REPORT

IMAGE - DENOISING

ARCHITETURE:

The architecture defined in the provided script is based on the Super-Resolution Convolutional Neural Network (SRCNN), a seminal deep learning model for image super-resolution. Here are the detailed components and structure of the architecture, along with an explanation of its working principle:

SRCNN Architecture

Components:

1. **Convolutional Layers:**
    - `conv1`: First convolutional layer with 3 input channels (RGB image), 64 output channels, kernel size of 9x9, and padding of 4.
    - `conv2`: Second convolutional layer with 64 input channels, 32 output channels, and kernel size of 1x1.
    - `conv3`: Third convolutional layer with 32 input channels, 3 output channels, kernel size of 5x5, and padding of 2.
2. **Activation Function:**
    - `ReLU`: Rectified Linear Unit (ReLU) activation function used after the first and second convolutional layers to introduce non-linearity.

Working Principle:

1. **Input Layer:**
    - The input to the network is a low-resolution image. This image is passed through the first convolutional layer (`conv1`).
2. **Feature Extraction (conv1):**
    - `conv1` extracts features from the input image using a large receptive field (9x9). This helps in capturing more contextual information from the

low-resolution image. The output is passed through the ReLU activation function to introduce non-linearity.

3. **Non-linear Mapping (conv2):**
   - The output from the first layer is passed through conv2, which performs a non-linear mapping. This layer has a kernel size of 1x1, which means it combines features extracted from the first layer in a pixel-wise manner, reducing the dimensionality to 32 feature maps. ReLU activation is again applied.
4. **Reconstruction (conv3):**
   - The final convolutional layer (conv3) reconstructs the high-resolution image from the feature maps obtained from the second layer. This layer has a smaller receptive field (5x5) and produces the final output with the same number of channels as the input image (3 channels for RGB).
5. **Output:**
   - The output of the network is a high-resolution image, which is the enhanced version of the input low-resolution image.

Model Specifications

- **Input Size:** The input is a low-resolution RGB image with dimensions (Height, Width, 3).
- **Output Size:** The output is a high-resolution RGB image with the same dimensions as the input but with enhanced quality.
- **Number of Parameters:** The total number of learnable parameters in the SRCNN model is determined by the convolutional layers:
  - First Layer: (9×9×3×64)+64 parameters
  - Second Layer: (1×1×64×32)+32 parameters
  - Third Layer: (5×5×32×3)+3 parameters

PNSR - for epoch 10-

- 34.03 for batch size of 16
- 30.71 for batch size of 32

```
Average MSE: 0.0009, Average PSNR: 30.71 dB, Average MAE: 0.0152
(0.0008595937397330999, 30.708303943757087, 0.015205301434522676)
```

Research paper :

https://books.google.co.in/books?hl=en&lr=&id=gK4SEAAAQBAJ&oi=fnd&pg=PA3&dq=super+resolution+research+paper+with+srcnn+model&ots=6vC6QeMLRs&sig=hIXPbAmFwcwZ_hejQmbvEEyCaNk

I have took mostly information from the paper and some from the other source.

DETAILS:

```
[2] from google.colab import drive
    drive.mount('/content/drive')

    Mounted at /content/drive

[3] import os
    import zipfile
    from google.colab import drive


    zip_file_path = '/content/drive/MyDrive/Train.zip'
    extract_dir = '/content/extracted_dataset'

    # Extract the ZIP file
    if not os.path.exists(extract_dir):
        with zipfile.ZipFile(zip_file_path, 'r') as zip_ref:
            zip_ref.extractall(extract_dir)
```

- Firstly I have used the train dataset and uploaded on the drive and accessed it from there by connecting drive to collab.
- I have extracted all the files from the zip folder as shown above.

```
[ ] pip install -q tensorflow numpy opencv-python
```

```
[ ] pip install -q scikit-learn
```

```
[ ] pip install  -q keras
```

```
[ ] from tensorflow.keras.preprocessing.image import img_to_array, load_img
```

Then installing all the necessary libraries as shown above like **tensorflow** for deep learning ,**scikit-learn** for machine learning to load the model and using all the necessary functions that are used in the project.

```
[ ] import torch
    import torch.nn as nn
    import torch.optim as optim
    from torch.utils.data import Dataset, DataLoader
    from torchvision import transforms
    from torchvision.datasets import ImageFolder
    from torchvision.utils import save_image
    from PIL import Image
```

torch: PyTorch is the main framework used for building and training the model.

torch.nn: Contains modules and functions to build neural networks.

torch.optim: Contains optimization algorithms like Adam.

torch.utils.data: Contains data loading utilities.

torchvision: Provides tools for image processing.

PIL (Python Imaging Library): Used for opening and manipulating images.

os: Provides functions to interact with the operating system.

```python
# Define a simple convolutional neural network for super-resolution
class SRCNN(nn.Module):
    def __init__(self):
        super(SRCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, kernel_size=9, padding=4)
        self.conv2 = nn.Conv2d(64, 32, kernel_size=1, padding=0)
        self.conv3 = nn.Conv2d(32, 3, kernel_size=5, padding=2)
        self.relu = nn.ReLU()

    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.relu(self.conv2(x))
        x = self.conv3(x)
        return x
```

SRCNN Model: The model consists of three convolutional layers(also explained above)

- **conv1:** Extracts features with a 9x9 filter.
- **conv2:** Performs non-linear mapping with a 1x1 filter.
- **conv3:** Reconstructs the high-resolution image with a 5x5 filter.
- **ReLU:** Activation function applied after the first two convolutional layers.

```python
# Set up transforms for input images (you can adjust these as needed)
transform = transforms.Compose([
    transforms.ToTensor(),
])

# Load dataset using ImageFolder
train_dataset = ImageFolder(root='/content/extracted_dataset/Train', transform=transform)

# Set up dataloader
train_dataloader = DataLoader(train_dataset, batch_size=32, shuffle=True, num_workers=4)

# Training function
def train_model(model, criterion, optimizer, dataloader, num_epochs=50):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    model.train()

    for epoch in range(num_epochs):
        running_loss = 0.0
        for i, (lr_images, hr_images) in enumerate(dataloader):
            hr_images = lr_images    # Assuming LR images are stored as HR images for simplicity in ImageFolder

            lr_images = lr_images.to(device)
            hr_images = hr_images.to(device)

            optimizer.zero_grad()

            outputs = model(lr_images)
            loss = criterion(outputs, hr_images)
            loss.backward()
```

```python
            loss.backward()
            optimizer.step()

            running_loss += loss.item()

            if i % 10 == 9:    # Print every 10 mini-batches
                print('[%d, %5d] loss: %.3f' %
                      (epoch + 1, i + 1, running_loss / 10))
                running_loss = 0.0

    print('Finished Training')
```

train_model: Trains the model.

- **device:** Uses GPU if available, otherwise CPU.
- **Training Loop:** For each epoch, iterates through the data loader, performs forward and backward passes, and updates model parameters.
- **Loss Printing:** Prints the loss every 10 batches for monitoring.

I have used sub batches so as to get more accurate and loss in results.

```python
def evaluate_model(model, criterion, dataloader):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    model.eval()

    mse_loss = nn.MSELoss()
    mae_loss = nn.L1Loss()
    total_mse = 0.0
    total_psnr = 0.0
    total_mae = 0.0
    count = 0

    with torch.no_grad():
        for lr_images, _ in dataloader:
            hr_images = lr_images    # Assuming LR images are stored as HR images for simplicity in ImageFolder

            lr_images = lr_images.to(device)
            hr_images = hr_images.to(device)

            outputs = model(lr_images)
            mse = mse_loss(outputs, hr_images)
            psnr = 10 * torch.log10(1 / mse)
            mae = mae_loss(outputs, hr_images)

            total_mse += mse.item()
            total_psnr += psnr.item()
            total_mae += mae.item()
            count += 1

            print(f'Image {count}: PSNR = {psnr:.2f} dB')
```

```python
    avg_mse = total_mse / count
    avg_psnr = total_psnr / count
    avg_mae = total_mae / count

    print(f'Average MSE: {avg_mse:.4f}, Average PSNR: {avg_psnr:.2f} dB, Average MAE: {avg_mae:.4f}')

    return avg_mse, avg_psnr, avg_mae
```

Evaluate_model: Evaluates the model on the dataset.

- **Metrics:** Calculates MSE, PSNR, and MAE for each image pair.
- **Averaging:** Computes average metrics over all images.
- **Printing:** Prints the average MSE, PSNR, and MAE.

Here I have tried various batch size like 16,32 and check the PSNR score for them .

For epoch 10-(For batch size 16 PSNR-34.03,batch size 32 PSNR-32.01)

For epoch 50-(For Batch size 32 PSNR-38.59(shown below with MSE AND MAE)

```
# Initialize model, loss function, and optimizer
model = SRCNN()
criterion = nn.MSELoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the model on 'train' dataset
train_model(model, criterion, optimizer, train_dataloader, num_epochs=50)

# Evaluate on training dataset (optional for demonstration)
print("Evaluation on training dataset:")
evaluate_model(model, criterion, train_dataloader)
```

Training, evaluation, and testing processes.

- **Transforms:** Defines transformations to apply to images (converting them to tensors).
- **Model Initialization:** Initializes the SRCNN model, loss function, and optimizer.
- **Training:** Trains the model using the `train_model` function.
- **Evaluation:** Evaluates the model on the training dataset.

For epoch 50-

```
Average MSE: 0.0001, Average PSNR: 38.59 dB, Average MAE: 0.0065
(0.00014345441905466l8, 38.58813661144626, 0.006488588713710347)
```

```python
import os
class TestImageDataset(Dataset):
    def __init__(self, low_res_dir, transform=None)
        self.low_res_dir = low_res_dir
        self.low_res_files = os.listdir(low_res_dir)
        self.transform = transform

    def __len__(self):
        return len(self.low_res_files)

    def __getitem__(self, idx):
        img_name = self.low_res_files[idx]
        img_path = os.path.join(self.low_res_dir, img_name)
        image = Image.open(img_path)

        if self.transform:
            image = self.transform(image)

        return image, img_name


if not os.path.exists('/content/extracted_dataset/test/predicted'):
        os.makedirs('/content/extracted_dataset/test/predicted')



test_dataset = TestImageDataset(low_res_dir='/content/extracted_dataset/test', transform=transform)
test_dataloader = DataLoader(test_dataset, batch_size=1, shuffle=False, num_workers=0)
```

Imports:

- `os`: Used for interacting with the operating system, particularly for file and directory operations.
- `Dataset and DataLoader`: PyTorch classes used for handling datasets and loading data in batches.
- `Image` from PIL: Used for opening and processing images.
- `transforms` from torchvision: Provides common image transformations.

TestImageDataset Class:

- **Initialization (`__init__`)**:
  - `low_res_dir`: Directory containing low-resolution test images.
  - `transform`: Optional transformations to apply to the images (e.g., converting to tensors, normalization).
  - `self.low_res_files`: List of all files in the `low_res_dir` directory, filtered to include only files (to exclude directories).
- **Length (`__len__`)**:
  - Returns the number of image files in the directory.
- **Get Item (`__getitem__`)**:
  - Given an index `idx`, retrieves the corresponding image file name.
  - Constructs the full path to the image file.

This snippet sets up a custom dataset for low-resolution images stored in a specified directory, ensuring that only image files are included. It prepares a DataLoader to iterate over these images one by one. It also ensures that a directory exists to save the predicted output images. This setup is commonly used in image processing tasks, such as super-resolution, where a model processes images to improve their resolution or quality.

```
[ ] device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    model.eval()


    with torch.no_grad():
            for lr_image, img_name in test_dataloader:
                lr_image = lr_image.to(device)
                output = model(lr_image)
                output_image_path = os.path.join('/content/extracted_dataset/test/predicted', img_name[0])
                save_image(output, output_image_path)
```

This code snippet is part of a process where a trained deep learning model is used to generate predictions on a set of test images. The steps involved include:

- Setting up the device for computation (GPU if available, otherwise CPU).
- Putting the model in evaluation mode.
- Iterating over the test dataset to get batches of low-resolution images.
- Moving the images to the appropriate device.
- Running the model to get high-resolution predictions.
- Saving each predicted image to a designated directory with its original filename.

This is commonly used in image processing tasks like super-resolution, where the goal is to improve the quality or resolution of input images using a trained neural network model.

```python
def evaluate_psnr(model, criterion, dataloader):
    device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
    model.to(device)
    model.eval()

    mse_loss = nn.MSELoss()
    mae_loss = nn.L1Loss()
    total_mse = 0.0
    total_psnr = 0.0
    total_mae = 0.0
    count = 0

    with torch.no_grad():
        for lr_images, _ in test_dataloader:
            hr_images = lr_images    # Assuming LR images are stored as HR images for simplicity in ImageFolder

            lr_images = lr_images.to(device)
            hr_images = hr_images.to(device)

            outputs = model(lr_images)
            mse = mse_loss(outputs, hr_images)
            psnr = 10 * torch.log10(1 / mse)
            mae = mae_loss(outputs, hr_images)

            total_mse += mse.item()
            total_psnr += psnr.item()
            total_mae += mae.item()
            count += 1

            print(f'Image {count}: PSNR = {psnr:.2f} dB')
```

```python
    avg_psnr = total_psnr/count
    print(f'Average PSNR: {avg_psnr:.2f} dB')

    return avg_psnr

evaluate_psnr(model,criterion,test_dataloader)
```

This function evaluates a trained model by:

- Setting the device and model to evaluation mode.
- Iterating over the test dataset.
- Performing inference to get high-resolution outputs.
- Calculating MSE, PSNR, and MAE for each image.
- Accumulating and printing the metrics for each image.
- Computing and returning the average PSNR across the test dataset.

This process is crucial for assessing the performance of models in image processing tasks like super-resolution, where metrics like PSNR are commonly used to evaluate image quality.

PSNR value of assumed test dataset-

I have taken a part of test images from the low data so that to predict and that data i have not included in training data and calculate its PSNR score and i got PSNR of value 40.67 .

Findings and solutions-

The code is well-structured but i think some modifications can be done-

- It assumes low-resolution images are stored as high-resolution images for simplicity. This might not always be the case and should be clarified.
- This code lacks error handling, which can be crucial for identifying and managing runtime issues.
- The evaluation function works well but in my opinion can be optimized for better performance and clarity.

By Babandeep Singh

22113035