VLG Summer Open Projects 2024

BY- Adarsh Yadav [21112010]

Denoising Image

Introduction

The aim of this project is to create an image denoising algorithm using deep learning techniques, specifically focusing on convolutional neural networks (CNNs). Image denoising is a critical preprocessing step in various image processing tasks, aimed at removing noise from images while preserving important details.

Dataset information

Our dataset comprises pairs of images, where each pair includes a noisy image and its corresponding clean version. Each noisy image is paired with its clean counterpart.

Dataset Preparation

- Noisy Images Folder: `path_to_noisy_images`
- Clean Images Folder: `path_to_clean_images`

Ensure that images in both folders are of the same size and properly aligned for training. This alignment is crucial for the model to learn the mapping from noisy to clean images effectively.

Modules and Libraries Used

- **os**: Handles directory and file operations.
- **cv2:** Provides image processing functions.
- **NumPy:** Supports array and matrix operations.
- **sklearn.model_selection:** Splits data into training and test sets.
- **TensorFlow :** Facilitates deep learning model creation and training.
- **tensorflow.keras.layers, models :** Defines neural network layers and models.
- **torch.utils.data :** Manages dataset and data loading in PyTorch.
- **torchvision.transforms :** Applies data transformations for image processing.
- **PIL.Image :** Handles image opening and manipulation.
- **skimage.metrics :** Calculates image quality metrics like PSNR.

Specification

- **Framework:** TensorFlow/keras
- **Layers:** Multiple convulation layers with activation functions, upsampling layers for increasing image dimension back to the original size.
- **Optimizer:** Adam optimizer
- **Loss function:** binary-cross entropy

Model Architecture

In this project, we implemented a convolutional neural network (CNN) architecture, which is commonly used for image processing tasks due to its ability to capture spatial hierarchies within images. Our CNN model features several convolutional layers, each followed by activation and pooling layers to progressively extract and condense image features.

- **Convolutional Layers(Conv2D layers with ReLU activation):** Extract features from the input images.
- **Activation Functions:** Employ Leaky ReLU to avoid the “dying ReLU” issue and improve gradient flow.
- **Pooling Layers(MaxPooling2D layers for downsampling):** Downsample feature maps to reduce spatial dimensions and computational load.
- **Upsampling Layers(MaxPooling2D layers for downsampling):** Increase the spatial dimensions of the feature maps to match the original image size.

Code Snippets for Model Creation

1. **Load the necessary libraries:**

```python
import os
import cv2
import numpy as np
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow.keras import layers, models
from torch.utils.data import Dataset, DataLoader
from torchvision import transforms
from PIL import Image
from skimage.metrics import peak_signal_noise_ratio as psnr
```

2. **Load images from the folder:**

   The function `load_images_from_folder` reads images from a specified folder, processes them, and returns a list of resized, normalized images. It iterates through each file in the folder, converts each image from BGR to RGB color space, resizes it to the specified dimensions (`img_size`), normalizes the pixel values to the range [0, 1], and appends the processed image to a list, which is then returned.

```python
def load_images_from_folder(folder, img_size=(128, 128)):
    images = []
    for filename in os.listdir(folder):
        img_path = os.path.join(folder, filename)
        img = cv2.imread(img_path)
        if img is not None:
            img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
            img = cv2.resize(img, img_size)
            img = img / 255.0
            images.append(img)
    return images
```

3. **Prepare datasets:**

This code sets up the data pipeline for training and testing an image denoising model using TensorFlow. Here is a step-by-step explanation:

1. Define Paths and Image Size:

- `noisy_folder_path` and `clean_folder_path` specify the directories containing the noisy and clean images, respectively.
- `img_size` sets the dimensions to which all images will be resized.

2. Load Images:

- `noisy_images` and `clean_images` are loaded from their respective folders using the `load_images_from_folder` function, which processes each image by resizing and normalizing it.

3. Split Data:

- The loaded images are split into training and testing sets using `train_test_split`, with 20% of the data reserved for testing. The random seed is set to 42 for reproducibility.

4. Preprocess Images:

- The `preprocess_image` function converts images to `tf.float32` data type, ensuring they are in the correct format for TensorFlow.

5. Create Datasets:

- `train_noisy_ds`, `train_clean_ds`, `test_noisy_ds`, and `test_clean_ds` are created by converting the image lists to TensorFlow datasets. Each dataset is then mapped to the `preprocess_image` function and batched with a size of 32.

6. Zip Datasets:

- `train_ds` and `test_ds` are created by zipping the noisy and clean datasets together, forming pairs of noisy and clean images for both training and testing. This zipping is essential for supervised learning, where the model learns to denoise the noisy images using the clean images as targets.

```
noisy_folder_path = 'path_to_noisy_images'
clean_folder_path = 'path_to_clean_images'
img_size = (128, 128)

noisy_images = load_images_from_folder(noisy_folder_path, img_size)
clean_images = load_images_from_folder(clean_folder_path, img_size)

train_noisy, test_noisy, train_clean, test_clean = train_test_split(
    noisy_images, clean_images, test_size=0.2, random_state=42
)

def preprocess_image(image):
    return tf.image.convert_image_dtype(image, tf.float32)

train_noisy_ds = tf.data.Dataset.from_tensor_slices(train_noisy).map(prepro
train_clean_ds = tf.data.Dataset.from_tensor_slices(train_clean).map(prepro
test_noisy_ds = tf.data.Dataset.from_tensor_slices(test_noisy).map(preproce
test_clean_ds = tf.data.Dataset.from_tensor_slices(test_clean).map(preproce

train_ds = tf.data.Dataset.zip((train_noisy_ds, train_clean_ds))
test_ds = tf.data.Dataset.zip((test_noisy_ds, test_clean_ds))
```

4. Create and compile the model:

This code defines and compiles a convolutional neural network (CNN) model for image denoising. Here is a detailed explanation:

1. **Function Definition**:

- **`create_denoising_model(input_shape)`:** This function creates a CNN model for image denoising with the given input shape.

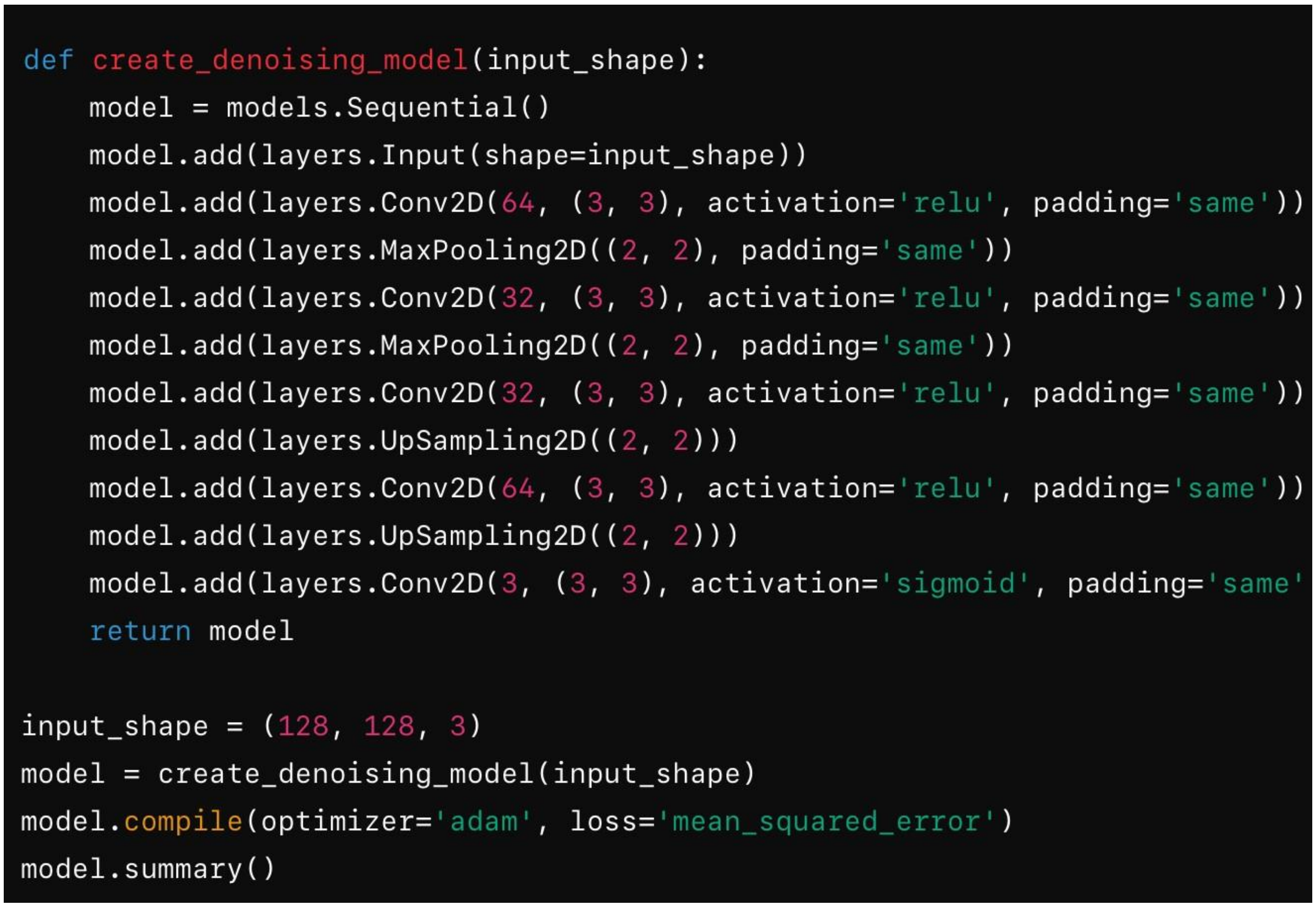
2. **Model Architecture**:

- **`model = models.Sequential()`:** Initializes a sequential model, which is a linear stack of layers.
- **`model.add(layers.Input(shape=input_shape))`:** Adds an input layer that takes images of the specified shape (128x128 with 3 color channels).

- **`model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))`:** Adds a convolutional layer with 64 filters, 3x3 kernel size, ReLU activation, and same padding.
- **`model.add(layers.MaxPooling2D((2, 2), padding='same'))`:** Adds a max pooling layer to downsample the feature maps, with same padding.
- **`model.add(layers.Conv2D(32, (3, 3), activation='relu', padding='same'))`**: Adds another convolutional layer with 32 filters.
- **`model.add(layers.MaxPooling2D((2, 2), padding='same'))`:** Adds another max pooling layer.
- **`model.add(layers.Conv2D(32, (3, 3), activation='relu', padding='same'))`:** Adds a third convolutional layer with 32 filters.
- **`model.add(layers.UpSampling2D((2, 2)))`:** Adds an upsampling layer to increase the spatial dimensions of the feature maps.
- **`model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))`:** Adds a convolutional layer with 64 filters.
- **`model.add(layers.UpSampling2D((2, 2)))`:** Adds another upsampling layer.
- **`model.add(layers.Conv2D(3, (3, 3), activation='sigmoid', padding='same'))`:** Adds a final convolutional layer with 3 filters and a sigmoid activation to produce the output image.

3. **Return the Model:** `return model`: Returns the constructed model.
4. **Model Creation and Compilation:**

- **`input_shape = (128, 128, 3)`:** Specifies the shape of the input images.
- **`model = create_denoising_model(input_shape)`:** Creates the model using the specified input shape.
- **`model.compile(optimizer='adam', loss='mean_squared_error')`:** Compiles the model with the Adam optimizer and mean squared error loss function, which is suitable for image reconstruction tasks.
- **`model.summary()`:** Prints a summary of the model architecture, showing the layers, their output shapes, and the number of parameters.



```
def create_denoising_model(input_shape):
    model = models.Sequential()
    model.add(layers.Input(shape=input_shape))
    model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2, 2), padding='same'))
    model.add(layers.Conv2D(32, (3, 3), activation='relu', padding='same'))
    model.add(layers.MaxPooling2D((2, 2), padding='same'))
    model.add(layers.Conv2D(32, (3, 3), activation='relu', padding='same'))
    model.add(layers.UpSampling2D((2, 2)))
    model.add(layers.Conv2D(64, (3, 3), activation='relu', padding='same'))
    model.add(layers.UpSampling2D((2, 2)))
    model.add(layers.Conv2D(3, (3, 3), activation='sigmoid', padding='same'
    return model

input_shape = (128, 128, 3)
model = create_denoising_model(input_shape)
model.compile(optimizer='adam', loss='mean_squared_error')
model.summary()
```

Training the Model:

This line of code trains the denoising model using the training dataset (`train_ds`) and evaluates its performance on the validation dataset (`test_ds`). Here's a detailed breakdown:

- model.fit : This function trains the model on the provided dataset.
- train_ds : The training dataset, which consists of pairs of noisy and clean images.
- epochs=50 : Specifies that the model will be trained for 50 iterations over the entire training dataset.

- validation_data=test_ds : Provides the validation dataset to evaluate the model's performance at the end of each epoch. This helps in monitoring the model's ability to generalize to unseen data.

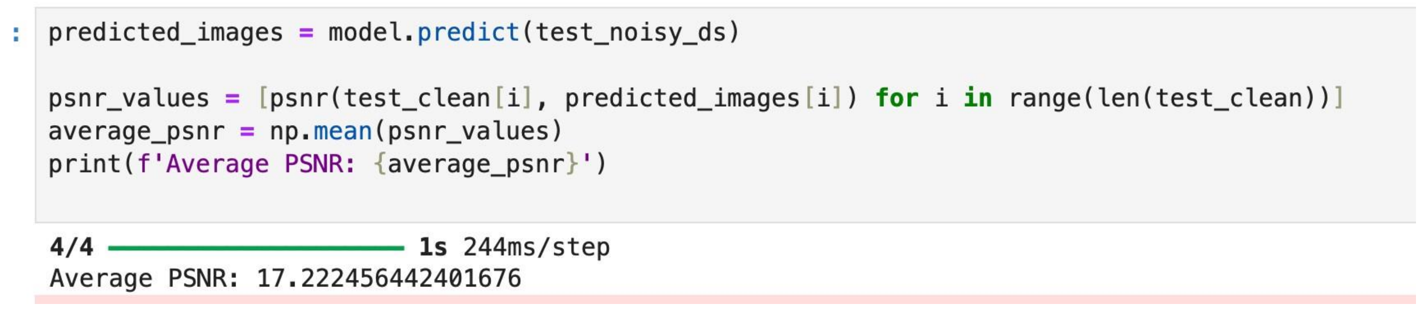
The training process will update the model's weights to minimize the loss function, and the validation performance will be tracked to prevent overfitting. The training history, including the loss and validation loss for each epoch, will be stored in the `history` object.

```
history = model.fit(train_ds, epochs=50, validation_data=test_ds)
```

Results:

The training log shows the progression of a denoising model over 50 epochs. Initially, the training loss is 0.0475 and the validation loss is 0.0393, indicating that the model starts with a relatively high error in reconstructing clean images from noisy inputs. As training progresses, the losses decrease, demonstrating that the model is learning to effectively denoise the images. By the 50th epoch, the training loss reduces to 0.0233 and the validation loss to 0.0237, indicating a significant improvement in the model's performance. The close values of training and validation losses suggest that the model is generalizing well to unseen data without overfitting.

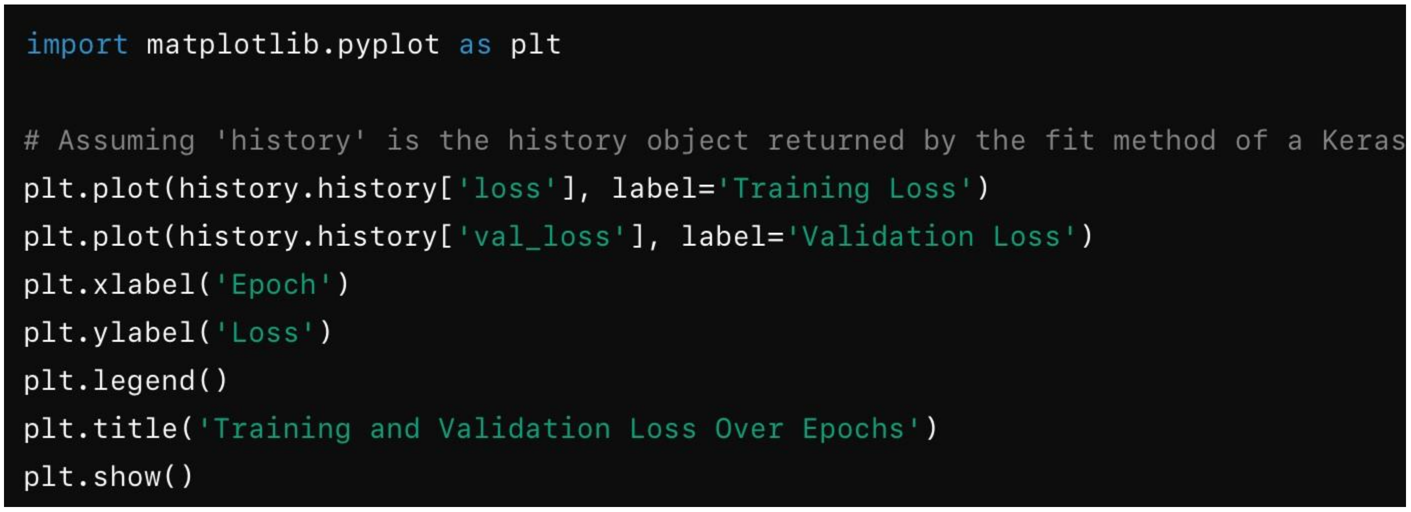
```
Epoch 1/50
13/13 ━━━━━━━━━━━━━━━━━━━━ 15s 527ms/step - loss: 0.0475 - val_loss: 0.0393
...
Epoch 50/50
13/13 ━━━━━━━━━━━━━━━━━━━━ 5s 370ms/step - loss: 0.0233 - val_loss: 0.0237
```



```
: predicted_images = model.predict(test_noisy_ds)

  psnr_values = [psnr(test_clean[i], predicted_images[i]) for i in range(len(test_clean))]
  average_psnr = np.mean(psnr_values)
  print(f'Average PSNR: {average_psnr}')
```

```
4/4 ━━━━━━━━━━━━━━━━━━━━ 1s 244ms/step
Average PSNR: 17.222456442401676
```

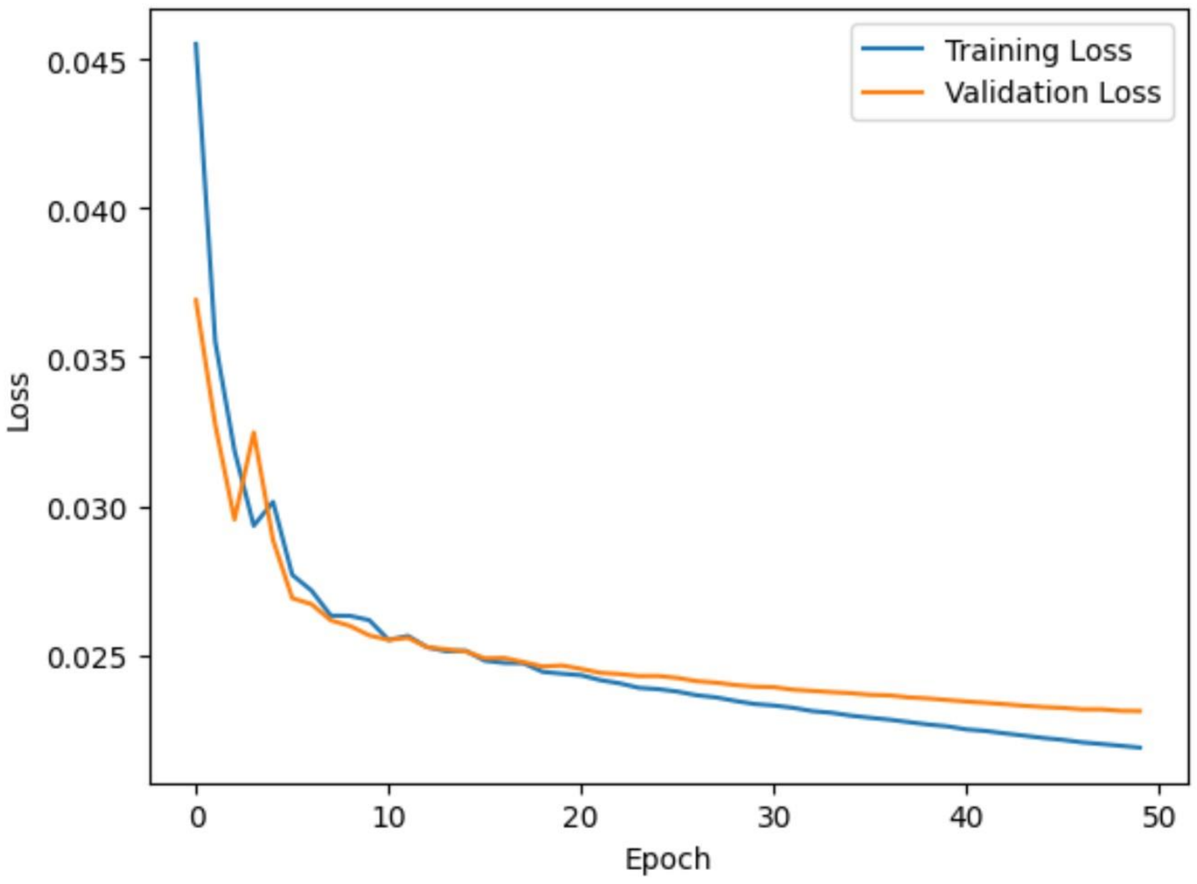


```
import matplotlib.pyplot as plt

# Assuming 'history' is the history object returned by the fit method of a Keras
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Training and Validation Loss Over Epochs')
plt.show()
```

The plot shows the training and validation loss of a machine learning model over 50 epochs. The blue line represents the training loss, while the orange line represents the validation loss. Both losses decrease over time, indicating that the model is learning. Initially, the training loss drops rapidly and then continues to decrease more gradually. The validation loss follows a similar pattern but starts slightly higher and decreases alongside the training loss, suggesting good generalization of the model without significant overfitting.

AVERAGE PSNR: **17.222456442401676**


Training Loss
Validation Loss
Loss
Epoch
0.045
0.040
0.035
0.030
0.025
0
10
20
30
40
50