Image Super-Resolution on Gaming Dataset using Enhanced CNN

Cherish Jain(22b3937) Aman Verma(22b3929)

1. Objective

The objective of this assignment is to perform image super-resolution on a custom dataset of gaming images. Each low-resolution image has a corresponding high-resolution ground truth. The goal is to evaluate the performance of a CNN-based Super-Resolution model in reconstructing high-quality images, emphasizing enhancements in textures, details, and overall fidelity.

2. Model Architecture

For this project, we have developed a novel and enhanced Super-Resolution architecture named **InsaneSRNet**, inspired by the Residual-in-Residual Dense Block (RRDB) concept introduced in ESRGAN. This architecture leverages dense connections, multi-level residual learning, and efficient pixel-shuffle-based upsampling to reconstruct high-resolution images from low-resolution inputs. Below is a breakdown of its core components:

- **Residual Dense Blocks (RDB):** Each RDB consists of multiple convolutional layers where the input to each layer is concatenated with the outputs of all previous layers. This dense connectivity encourages feature reuse and allows for more expressive and hierarchical representations. A 1x1 convolution compresses the concatenated features, and the block output is scaled and added to the original input to form a residual connection.
- **Residual-in-Residual Dense Blocks (RRDB):** Three RDBs are wrapped in a larger residual structure to form an RRDB. This deeper residual formulation enables stable training of deeper networks and better gradient flow, facilitating the learning of complex textures and details.
- **Upscale Blocks (Pixel Shuffle):** To increase spatial resolution, two Pixel Shuffle-based upscale blocks are employed sequentially. Each block increases the resolution by a factor of 2, resulting in a total upscale factor of 4. This method is computationally efficient and well-suited for high-quality super-resolution tasks.
- **Head and Tail Convolutions:** A head convolution layer first transforms the input image into a higher-dimensional feature space. After the RRDB trunk and upscaling, a tail convolution layer maps the features back into the RGB image space.
- **Global Skip Connection (Bilinear Interpolation):** A global skip connection is incorporated by upscaling the input image via bilinear interpolation and adding it to the final output. This helps the model preserve the overall structure and low-frequency information in the image.

The architecture consists of the following stages:

- **Input Layer:** Accepts a 3-channel RGB low-resolution image.
- **Feature Extraction:** Initial convolution extracts base features.
- **RRDB Trunk:** A sequence of Residual-in-Residual Dense Blocks (e.g., 5 blocks) extract and refine deep hierarchical features.
- **Upsampling:** Two pixel shuffle blocks progressively upscale the features to 4× the original resolution.
- **Output Generation:** A final convolution layer produces the super-resolved image, enhanced by a bilinear-upsampled skip connection from the input.

The model is implemented as follows:

Listing 1: InsaneSRNet Model Architecture

```
# Residual Dense Block (RDB)
class ResidualDenseBlock(nn.Module):
    def __init__(self, in_channels, growth_channels=32, num_layers=5):
        super(ResidualDenseBlock, self).__init__()
        self.layers = nn.ModuleList()
        for i in range(num_layers):
            self.layers.append(
                nn.Conv2d(in_channels + i * growth_channels,
                   growth_channels, kernel_size=3, padding=1)
            )
        self.lrelu = nn.LeakyReLU(negative_slope=0.2, inplace=True)
        self.conv_1x1 = nn.Conv2d(in_channels + num_layers *
           growth_channels, in_channels, kernel_size=1)

    def forward(self, x):
        inputs = [x]
        for conv in self.layers:
            concat = torch.cat(inputs, 1)
            out = self.lrelu(conv(concat))
            inputs.append(out)
        out = torch.cat(inputs, 1)
        out = self.conv_1x1(out)
        return out * 0.2 + x  # residual scaling

# Residual-in-Residual Dense Block (RRDB)
class RRDB(nn.Module):
    def __init__(self, in_channels, growth_channels=32, num_layers=5):
        super(RRDB, self).__init__()
        self.rdb1 = ResidualDenseBlock(in_channels, growth_channels,
           num_layers)
        self.rdb2 = ResidualDenseBlock(in_channels, growth_channels,
           num_layers)
        self.rdb3 = ResidualDenseBlock(in_channels, growth_channels,
           num_layers)

    def forward(self, x):
        out = self.rdb1(x)
        out = self.rdb2(out)
```

```
        out = self.rdb3(out)
        return out * 0.2 + x  # residual scaling

# Upscale block using PixelShuffle
class UpscaleBlock(nn.Module):
    def __init__(self, in_channels, upscale_factor=2):
        super(UpscaleBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, in_channels * upscale_factor *
           upscale_factor, 3, padding=1)
        self.pixel_shuffle = nn.PixelShuffle(upscale_factor)
        self.lrelu = nn.LeakyReLU(negative_slope=0.2, inplace=True)

    def forward(self, x):
        x = self.conv(x)
        x = self.pixel_shuffle(x)
        return self.lrelu(x)

# InsaneSRNet: Our advanced model
class InsaneSRNet(nn.Module):
    def __init__(self, num_rrdb=5, in_channels=3, channels=64,
       upscale_factor=4):
        super(InsaneSRNet, self).__init__()
        self.head = nn.Conv2d(in_channels, channels, 3, padding=1)
        self.trunk = nn.Sequential(*[RRDB(channels) for _ in range(
           num_rrdb)])
        self.trunk_conv = nn.Conv2d(channels, channels, 3, padding=1)
        self.upscale1 = UpscaleBlock(channels, upscale_factor=2)
        self.upscale2 = UpscaleBlock(channels, upscale_factor=2)
        self.tail = nn.Conv2d(channels, in_channels, 3, padding=1)

    def forward(self, x):
        skip = F.interpolate(x, scale_factor=4, mode='bilinear',
           align_corners=False)
        out = self.head(x)
        out = self.trunk(out)
        out = self.trunk_conv(out)
        out = self.upscale1(out)
        out = self.upscale2(out)
        out = self.tail(out)
        out += skip
        return torch.sigmoid(out)

# Instantiate the model
model = InsaneSRNet(num_rrdb=5, in_channels=3, channels=64, upscale_factor
   =4).to(DEVICE)
optimizer = Adam(model.parameters(), lr=1e-4)
criterion = nn.MSELoss()
```

This architecture is capable of producing highly detailed super-resolved images. The use of deep residual dense connections enables better texture recovery, while the combination of multi-level residual learning and efficient upscaling makes InsaneSRNet a powerful tool for high-fidelity image reconstruction

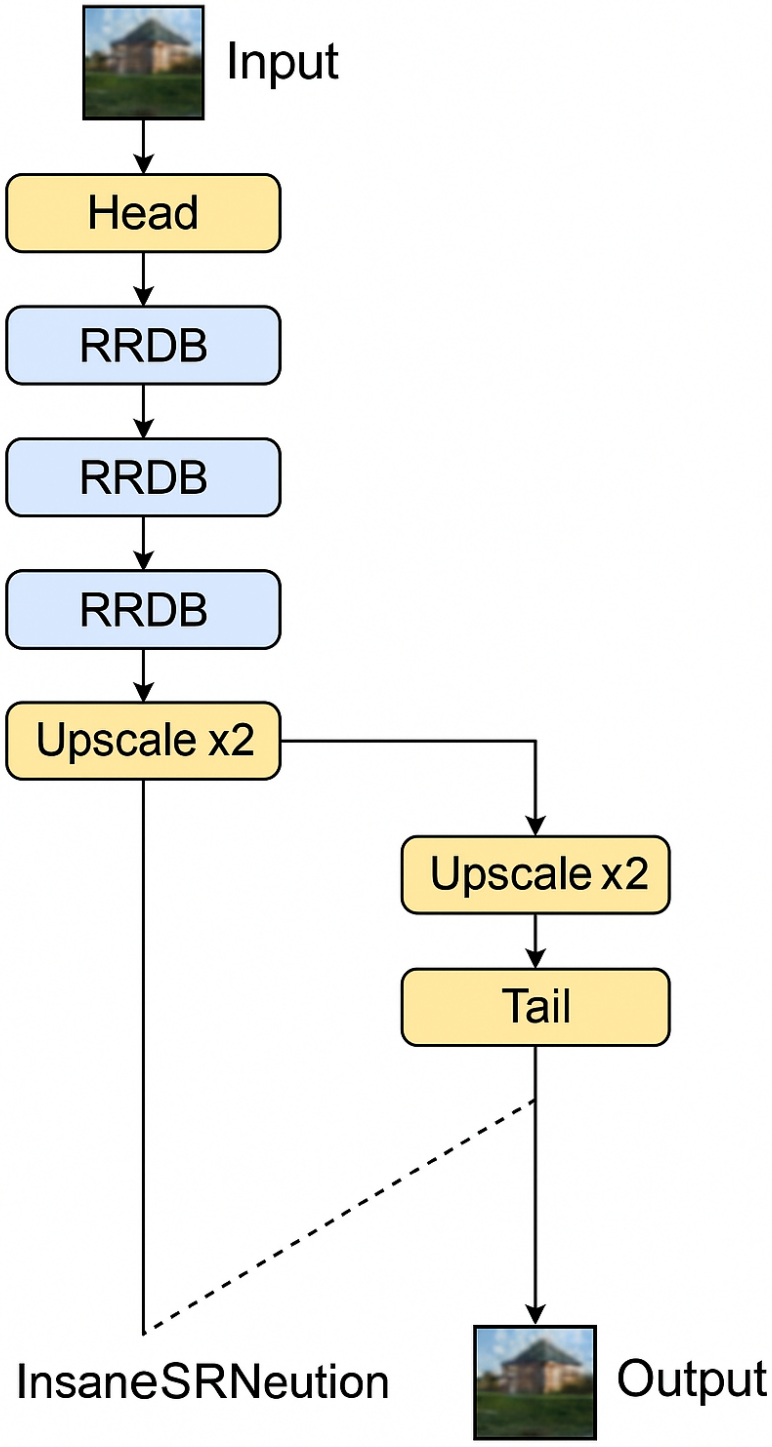

Figure 1: Enhanced Super-Resolution CNN Architecture

3. Dataset Description

- **Train Set:** Consists of pairs of low-resolution (LR) and high-resolution (HR) gaming images.

- **Test Set:** Contains LR images without HR ground truth; used for submission.
- **Train-Val Split:** The dataset was split into training and validation sets in a 90:10 ratio.

4. Training Details

The training process for the proposed *InsaneSRNet* model was designed to be efficient and robust, incorporating advanced training strategies to optimize performance and stability. The key components of the training procedure are outlined below:

- **Optimizer:** The Adam optimizer was used with a learning rate of 10^{-4} to update model weights. Adam combines the advantages of both AdaGrad and RMSProp, offering adaptive learning rates and momentum, making it well-suited for computer vision tasks like image super-resolution.
- **Loss Function:** We employed the Mean Squared Error (MSE) loss to measure reconstruction accuracy between the generated high-resolution (HR) images and the ground truth. MSE is widely used for regression problems and provides a stable signal for pixel-level prediction tasks.
- **Mixed Precision Training:** To accelerate training and reduce GPU memory usage, we utilized PyTorch's Automatic Mixed Precision (AMP). The `autocast` context manager was used during forward passes, and gradient scaling via `GradScaler` ensured stable updates without numeric underflow.
- **Progress Tracking:** Each training epoch included a progress bar via `tqdm`, dynamically displaying the training loss to provide live feedback and detect issues during training.
- **Validation and Checkpointing:** After each epoch, the model was evaluated on a validation set using a custom Joint Metric (JM). The model weights were saved in two formats:
  - `latest.pth` – contains weights from the most recent epoch.
  - `best.pth` – contains weights from the epoch with the best validation JM score.
- **Submission CSV Generation:** After each epoch, predictions on the test set were generated, and a CSV submission file was automatically saved. This allowed efficient tracking and versioning of output performance.
- **Memory Optimization:** GPU memory was cleared at the end of each epoch using `torch.cuda.empty_cach` to avoid accumulation and enable smoother multi-epoch training.

Listing 2: Training Loop with Mixed Precision

```
scaler = GradScaler()
best_metric = -float('inf')

for epoch in range(EPOCHS):
    model.train()
    train_loss = 0.0
    progress_bar = tqdm(train_loader, desc=f"Epoch␣{epoch+1}/{EPOCHS}␣-␣
       Training")
```

```
    for lr_imgs, hr_imgs in progress_bar:
        lr_imgs = lr_imgs.to(DEVICE)
        hr_imgs = hr_imgs.to(DEVICE)
        optimizer.zero_grad()

        with autocast(device_type='cuda'):
            outputs = model(lr_imgs)
            loss = criterion(outputs, hr_imgs)

        scaler.scale(loss).backward()
        scaler.step(optimizer)
        scaler.update()
        train_loss += loss.item()
        progress_bar.set_postfix(loss=loss.item())

    avg_train_loss = train_loss / len(train_loader)
    print(f"Epoch␣{epoch+1}␣Training␣Loss:␣{avg_train_loss:.6f}")

    # Generate submission for test set
    generate_submission_csv(model, test_loader, test_filenames, epoch,
       SUBMISSION_DIR)

    # Evaluate model performance on validation set
    print("\nEvaluating␣on␣validation␣set:")
    avg_joint = evaluate_model(model, val_loader)

    # Save latest model
    torch.save(model.state_dict(), os.path.join(SUBMISSION_DIR, "latest.
       pth"))

    # Save best model if improved
    if avg_joint > best_metric:
        best_metric = avg_joint
        torch.save(model.state_dict(), os.path.join(SUBMISSION_DIR, "best.
           pth"))
        print(f"New␣best␣model␣saved␣at␣epoch␣{epoch+1}.")

    torch.cuda.empty_cache()
```

This training loop efficiently balances speed, accuracy, and memory management, ensuring that both intermediate and best-performing models are preserved. The use of automated validation and test submission generation also supports a reproducible and traceable training pipeline.

5. Experiments and Results

To monitor model performance, we evaluated the Training loss and validation Joint Metric at the end of each epoch. The figure below illustrates the progression across 10 epochs.

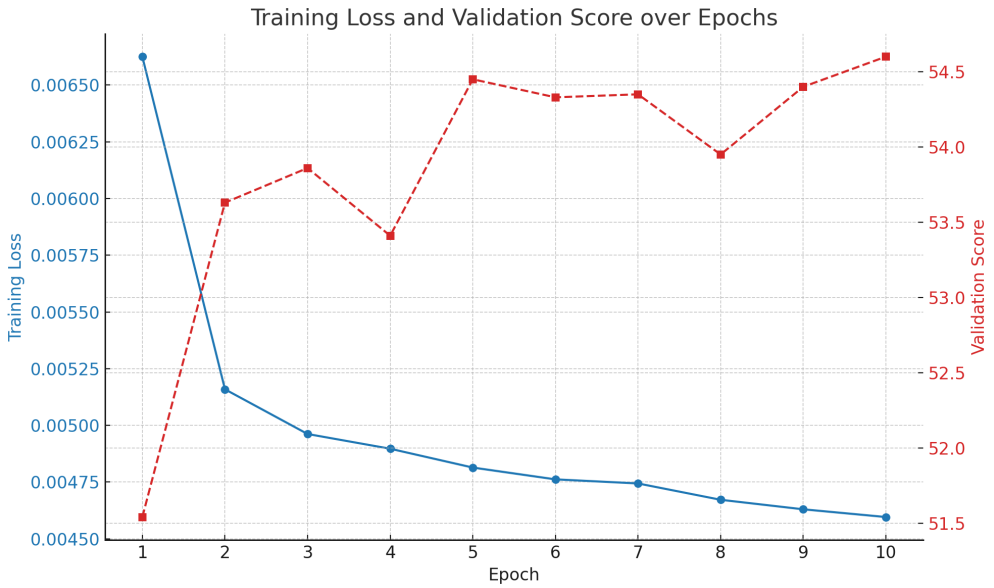

Table 1: Training Loss and Validation Score per Epoch

Epoch	Training Loss	Validation Score (JM)
1	0.006625	51.54
2	0.005159	53.63
3	0.004962	53.86
4	0.004897	53.41
5	0.004814	54.45
6	0.004762	54.33
7	0.004744	54.35
8	0.004672	53.95
9	0.004630	54.40
10	0.004596	54.60

5.1 Validation Score Calculation

The validation score used in this project, referred to as the **Joint Metric**, is designed to evaluate the perceptual and pixel-wise quality of the super-resolved images. It combines the Structural Similarity Index (SSIM) and the Peak Signal-to-Noise Ratio (PSNR) using a weighted sum, where higher emphasis is placed on perceptual quality.

The Joint Metric is defined as:

$$\text{Joint Metric} = 40 \times \text{SSIM} + \text{PSNR} \tag{1}$$

Here, SSIM assesses perceptual similarity between the predicted image and the ground truth, while PSNR measures pixel-wise fidelity. The weight of 40 assigned to SSIM reflects its importance in capturing human-perceived quality.

This metric was computed on the validation set after each epoch to monitor model performance. The model checkpoint achieving the highest validation joint metric was saved as the best model.

6. Prediction and Visualization

Below is a sample visualization of the model's output on training and test images.

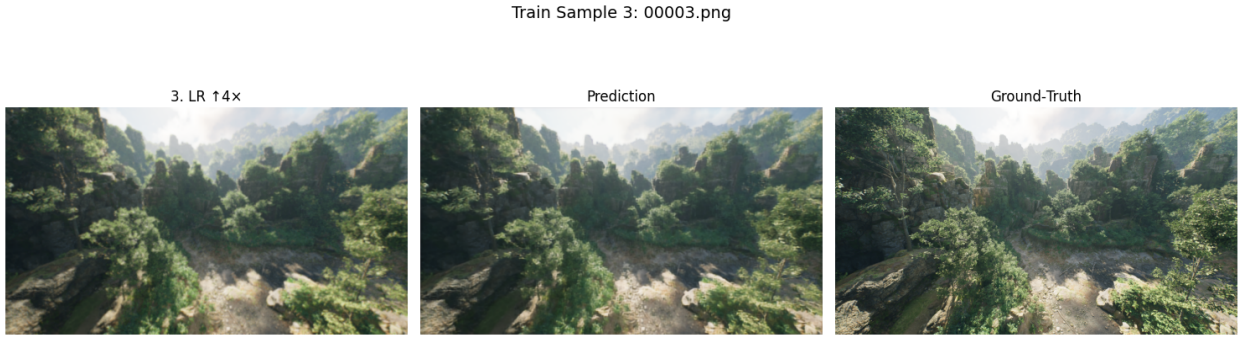

Figure 2: Left: Upsampled LR Image, Center: Model Prediction, Right: Ground Truth

Figure 3: Left: Upsampled LR Image, Center: Model Prediction, Right: Ground Truth

Figure 4: Left: Upsampled LR Image, Center: Model Prediction, Right: Ground Truth

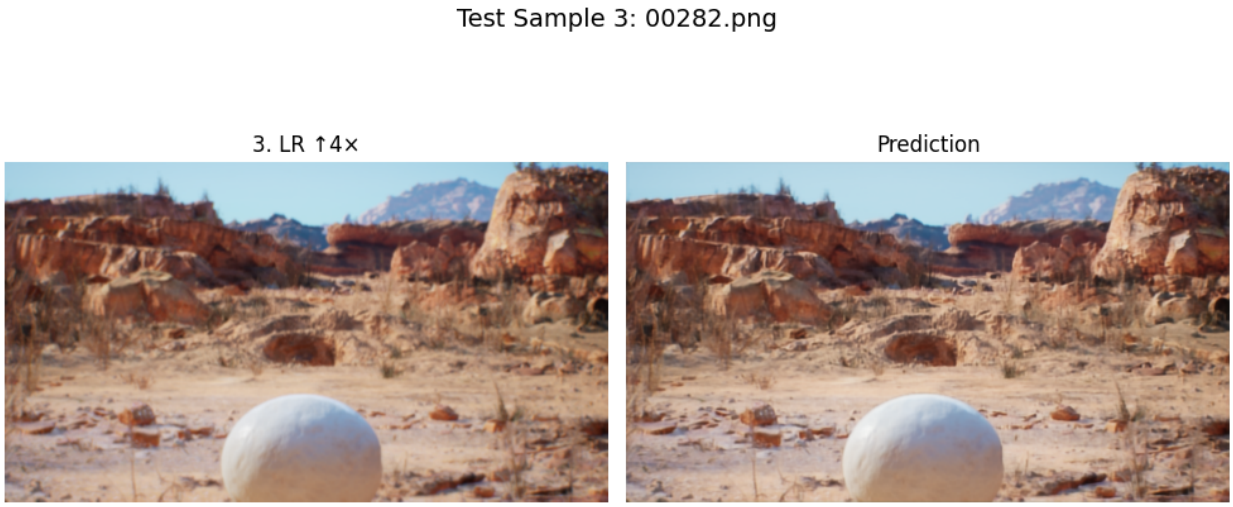

Figure 5: Left: Upsampled LR Image, Center: Model Prediction

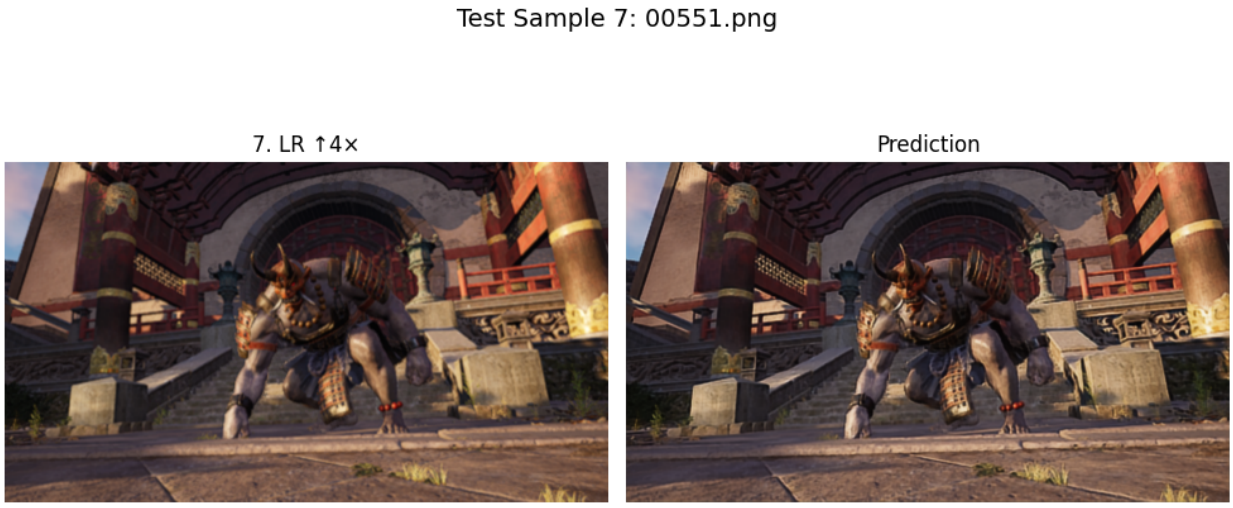

Figure 6: Left: Upsampled LR Image, Center: Model Prediction

Figure 7: Left: Upsampled LR Image, Center: Model Prediction

7. Submission Format

The output of the model is encoded into a base64 string and stored in a CSV format for evaluation.

Listing 3: Base64 Encoding for Submission

```
def encode_image_to_base64(image_path):
    image = cv2.imread(image_path)
    _, buffer = cv2.imencode('.png', image)
    return base64.b64encode(buffer).decode('utf-8')
```

The CSV contains two columns:

- **id** – Image filename
- **Encoded_Image** – Base64 encoded predicted image

8. Evaluation Metric

The evaluation metric is a combination of PSNR and SSIM:

$$\text{Joint Metric} = 40 \times \text{SSIM} + \text{PSNR}$$

Where:

- **PSNR** – Peak Signal-to-Noise Ratio
- **SSIM** – Structural Similarity Index Measure

Listing 4: Evaluation Function

```
def calculate_joint_metric(img1, img2):
    psnr_value = psnr(img1, img2, data_range=255)
    ssim_value, _ = ssim(img1, img2, full=True, multichannel=True)
    return 40 * ssim_value + psnr_value
```