Super-Resolution Generative Adversarial Network

CONTENTS

1. Introduction

This report offers an in-depth examination of a Python codebase that implements two GAN-based image super-resolution pipelines—the original SRGAN architecture and an Enhanced variant built around Residual-in-Residual Dense Blocks (RRDB). Both models are trained on the DIV2K dataset, which provides high-quality photographic images. The goal is to learn a mapping from low-resolution (LR) bicubic-downsampled inputs back to their high-resolution (HR) counterparts, maximizing both pixel fidelity and perceptual realism.

---

2. Dependency Installation

The script begins by ensuring all required libraries are present:

```
pip install torch torchvision pillow scikit-image tqdm
```

- **PyTorch & torchvision**: Core deep-learning framework and vision utilities (datasets, transforms, pre-built layers).
- **Pillow**: Image I/O (opening, resizing, saving).
- **scikit-image**: Provides quantitative image-quality metrics—PSNR and SSIM—used during evaluation.
- **tqdm**: Lightweight progress bars around loops, giving real-time feedback during training and evaluation.

By installing these at the top, the code guarantees reproducibility across environments, and avoids runtime import errors.

---

3. Dataset Download & Low-Resolution Generation

DIV2K Download

```
if not os.path.isdir('DIV2K_train_HR'):
    !wget -q
https://data.vision.ee.ethz.ch/cvl/DIV2K/DIV2K_train_HR.zip
    !unzip -q DIV2K_train_HR.zip
    !rm DIV2K_train_HR.zip
```

- Checks for the presence of the `DIV2K_train_HR` directory.
- If absent, downloads the official DIV2K training images in ZIP form, extracts them, and deletes the archive to save disk space.

`create_lr` Function

```
def create_lr(hr_dir='DIV2K_train_HR', lr_dir='DIV2K_train_LR',
scale=4):
    os.makedirs(lr_dir, exist_ok=True)
    for fn in os.listdir(hr_dir):
        if not fn.lower().endswith(('png','jpg','jpeg')):
            continue
        hr = Image.open(f'{hr_dir}/{fn}').convert('RGB')
        w, h = hr.size
        # Downsample then upsample to simulate low-res input
        lr = hr.resize((w//scale, h//scale), Image.BICUBIC)
        lr = lr.resize((w, h), Image.BICUBIC)
        lr.save(f'{lr_dir}/{fn}')
create_lr()
```

- **Purpose**: Synthesizes LR inputs by bicubic downsampling (factor of 4) and then bicubic upsampling back to original dimensions.
- **Rationale**: Models learn to reverse the specific blur/artifacts introduced by bicubic resizing, which approximates common real-world downsampling.
- **Output**: A parallel directory `DIV2K_train_LR` filled with LR images matching the filenames of the HR set.

---

4. Custom PyTorch Dataset & DataLoader

```
class DIV2KDataset(Dataset):
    def __init__(self, hr_dir, lr_dir, transform=None):
        self.hr_dir, self.lr_dir = hr_dir, lr_dir
        self.fns = [f for f in os.listdir(hr_dir) if
f.lower().endswith(('png','jpg'))]
        self.transform = transform or Compose([
            ToTensor(),
            Normalize((0.5,)*3, (0.5,)*3)  # scales pixels to [-1,1]
        ])
```

```python
    def __len__(self):
        return len(self.fns)

    def __getitem__(self, i):
        hr = Image.open(f'{self.hr_dir}/{self.fns[i]}').convert('RGB')
        lr = Image.open(f'{self.lr_dir}/{self.fns[i]}').convert('RGB')
        return self.transform(lr), self.transform(hr)
```

- **Initialization**:
  - Scans the HR directory for image filenames.
  - Defines a default transform: convert to PyTorch tensor, then normalize each of the three color channels to mean = 0.5, std = 0.5 (mapping [0,1]→[–1,1]).
- **Length**: Returns the number of image pairs available.
- **Get Item**: Loads the LR/HR pair by filename, applies identical transforms, and returns a tuple `(lr_tensor, hr_tensor)`.

A `DataLoader` wraps this dataset:

```python
ds = DIV2KDataset('DIV2K_train_HR', 'DIV2K_train_LR')
dl = DataLoader(ds, batch_size=16, shuffle=True, num_workers=4,
pin_memory=True)
```

- **Batch Size (bs)**: 16 images per iteration balances GPU memory constraints with gradient stability.
- **Shuffling**: Ensures varied batches each epoch.
- **Workers**: Four parallel processes accelerate image loading/preprocessing.
- **pin_memory**: Speeds host→GPU transfer for CUDA training.

---

5. Model Architectures

5.1. Baseline SRGAN Components

- **ResidualBlock(c=64)**
  Two 3×3 conv layers, each followed by BatchNorm and PReLU. A skip-connection adds the input to the block's output, promoting gradient flow and preserving low-level details.
- **UpsampleBlock(c, scale=2)**
  A conv layer expands channels by `scale²`, then `PixelShuffle(scale)` rearranges

them into doubled spatial dimensions, followed by PReLU activation. Stacks of these blocks achieve the desired upsampling factor (e.g., ×4 with two blocks).

- **Generator(num_res=16, up=4)**
  1. Initial Conv(9×9) + PReLU
  2. 16 ResidualBlocks in sequence
  3. Conv(3×3) + BatchNorm and add skip from step 1
  4. UpsampleBlocks to scale by 4
  5. Final Conv(9×9) produces 3-channel output, followed by `tanh` scaled to [0,1].
- **Discriminator**
  A deep CNN that progressively halves spatial resolution while doubling channels. Each stage uses Conv3×3 → BatchNorm → LeakyReLU(0.2). Ends with Global Average Pooling, Flatten, two dense layers, and a single logit output for real/fake classification.

5.2. Enhanced Generator with RRDB

- **DenseResidualBlock**
  - Five sequential conv layers. Each layer's input concatenates all previous feature maps (dense connectivity), encouraging feature reuse.
  - A 1×1 conv fuses the concatenated maps back to the original channel size, and a scaled skip adds the block's input—this "residual-in-residual" design stabilizes training.
- **RRDB**
  Three DenseResidualBlocks cascaded, with an overall scaled skip connection. This forms the core trunk of the Enhanced generator.
- **EnhancedGenerator(rrdb_blocks=23)**
  - Uses one initial Conv(9×9) + PReLU.
  - Passes through 23 RRDBs.
  - A trunk conv merges back to 64 channels, adds the initial features.
  - Two upsampling stages (conv→PixelShuffle→PReLU) to achieve ×4 scaling.
  - A final Conv(9×9) and `tanh`→[0,1] yields the super-resolved output.

---

6. Training Procedures

6.1. `train_srgan`

```
def train_srgan(hr_dir, lr_dir, epochs=200, bs=16, lr=1e-4,
save_every=10):
    # Setup
```

```
    device = torch.device('cuda' if torch.cuda.is_available() else
'cpu')
    ds, dl = DIV2KDataset(hr_dir, lr_dir), DataLoader(...)

    G, D = Generator().to(device), Discriminator().to(device)
    optG = Adam(G.parameters(), lr=lr); optD = Adam(D.parameters(),
lr=lr)
    mse, bce = MSELoss(), BCEWithLogitsLoss()

    for e in range(1, epochs+1):
        G.train(); D.train()
        for lr_imgs, hr_imgs in tqdm(dl, desc=f"SRGAN Ep
{e}/{epochs}"):
            lr_imgs, hr_imgs = lr_imgs.to(device), hr_imgs.to(device)
            valid = torch.ones(len(lr_imgs),1, device=device)
            fake  = torch.zeros(len(lr_imgs),1, device=device)

            # Discriminator update
            gen_hr_detached = G(lr_imgs).detach()
            lossD = 0.5 * (bce(D(hr_imgs), valid) +
bce(D(gen_hr_detached), fake))
            optD.zero_grad(); lossD.backward(); optD.step()

            # Generator update
            gen_hr = G(lr_imgs)
            loss_content = mse(gen_hr, hr_imgs)
            loss_adv     = bce(D(gen_hr), valid)
            loss_pix     = mse(gen_hr, hr_imgs)
            lossG = loss_content + 1e-3 * loss_adv + 2e-6 * loss_pix
            optG.zero_grad(); lossG.backward(); optG.step()

        print(f"Completed epoch {e}/{epochs}")
        if e % save_every == 0:
            torch.save(G.state_dict(), f'gen_{e}.pth')
            torch.save(D.state_dict(), f'disc_{e}.pth')
```

- **Loss Breakdown**:
  - **Content Loss** (`mse(gen_hr, hr)`) penalizes pixel-wise differences.
  - **Adversarial Loss** (`bce(D(gen_hr), valid)`) forces the generator to produce images that fool the discriminator.
  - **Pixel Loss** (another MSE term, scaled very low) provides a slight extra pull towards the ground truth.

6.2. train_enhanced

Follows the same overall loop but:

- Initializes `EnhancedGenerator()`.
- Optionally loads a pretrained SRGAN checkpoint into the RRDB trunk for warm start.
- Adjusts adversarial/pixel loss weights (e.g., `0.01 * loss_adv`, `0.006 * loss_pix`) to account for the deeper network's different convergence properties.

Both functions leverage GPU when available and save model checkpoints at regular intervals for later evaluation or fine-tuning.

---

7. Evaluation Routine

```
def evaluate(ckpt, hr_dir, lr_dir, enhanced=False):
    device = torch.device('cuda' if ...)
    G = EnhancedGenerator() if enhanced else Generator()
    G.load_state_dict(torch.load(ckpt, map_location=device))
    G.eval()

    ds = DIV2KDataset(hr_dir, lr_dir)
    dl = DataLoader(ds,1, shuffle=False)
    ps, ss = [], []

    for lr, hr in tqdm(dl, desc="Eval"):
        lr, hr = lr.to(device), hr.to(device)
        with torch.no_grad():
            out = G(lr)
```

```
        out_np =
(out.squeeze().permute(1,2,0).cpu().numpy()*255).astype(np.uint8)
        hr_np  =
(hr.squeeze().permute(1,2,0).cpu().numpy()*255).astype(np.uint8)

        ps.append(psnr(hr_np, out_np, data_range=255))
        ss.append(ssim(hr_np, out_np, multichannel=True,
data_range=255))

    print(f'Average PSNR: {np.mean(ps):.4f}, Average SSIM:
{np.mean(ss):.4f}')
```

- **PSNR (Peak Signal-to-Noise Ratio)** quantifies absolute pixel-value fidelity—the higher, the closer to the ground truth.
- **SSIM (Structural Similarity Index)** measures perceived structural similarity, accounting for luminance, contrast, and local correlations—a higher SSIM indicates better perceptual realism.
- By iterating on each LR image one at a time, the script ensures accurate metric computation without batch artifacts.

---

8. Example Workflow

At the bottom of the script, users are shown how to invoke training and evaluation:

```
# Train baseline SRGAN for 30 epochs
train_srgan('DIV2K_train_HR','DIV2K_train_LR', epochs=30)

# Train Enhanced SRGAN with warm-start from epoch-30 generator
train_enhanced('DIV2K_train_HR','DIV2K_train_LR','gen_200.pth',
epochs=30)

# Evaluate the enhanced model at epoch 30
evaluate('enh_gen_100.pth','DIV2K_train_HR','DIV2K_train_LR',
enhanced=True)
```

This sequence demonstrates:

1. Baseline training from scratch.
2. Enhanced training leveraging a pretrained checkpoint.
3. Quantitative assessment of the enhanced model's performance.

---

9. Conclusion & Potential Extensions

- **Modularity**: Clear separation of concerns—data preparation, model definition, training, evaluation—facilitates experimentation and reuse.
- **Flexibility**: Swapping architectures via subclassing makes it straightforward to implement new generator/discriminator variants.
- **Metrics**: Combined PSNR/SSIM evaluation addresses both pixel accuracy and perceptual quality.