Image Super-Resolution using SRResNet for Kaggle Competition

Aryan Bhosale
210040024
kaggle id: supe.one.0

1 Introduction

In his report, I talk about the implementation and results of an image super-resolution (SR) project developed for a Kaggle competition. The primary objective was to reconstruct high-resolution (HR) gaming images from their corresponding low-resolution (LR) counterparts using a deep learning model. The project involved training an SRResNet model from scratch on the provided dataset, evaluating its performance using relevant metrics, and generating predictions for the competition's test set. The ranking in the competition was based on the quality of these predictions.

2 Model Architecture: SRResNet

The core of this project is the Super-Resolution Residual Network (SRResNet), a deep convolutional neural network architecture specifically designed for image super-resolution tasks.

2.1 Theoretical Description

SRResNet leverages several key concepts:

- **Residual Learning:** Instead of directly learning a mapping from LR to HR images, the network primarily learns the *residual* information – the difference between the HR image and a simpler upscaled version of the LR image (or features). This makes optimization easier, especially for deep networks, as it often involves learning high-frequency details.
- **Residual Blocks:** The network employs multiple residual blocks. Each block typically contains convolutional layers, batch normalization (BN), and activation functions (ReLU in this implementation). A skip connection adds the input of the block to its output (after convolutions and activations), allowing gradients to flow more easily through the network and enabling the training of deeper models without degradation.
- **Batch Normalization (BN):** BN layers are used after convolutional layers (before activation in this implementation's residual block) to stabilize training by normalizing the activations, reducing internal covariate shift.
- **PixelShuffle Upsampling:** Instead of traditional interpolation methods (like bicubic) or transposed convolutions, SRResNet often uses sub-pixel convolutional layers, commonly implemented via a `Conv2d` followed by `PixelShuffle`. This learns the upsampling process. A `Conv2d` layer outputs $C \times r^2$ channels, and `PixelShuffle` rearranges these elements into an output tensor with C channels and $r\times$ spatial dimensions (height and width), effectively upscaling the feature map by a factor of r. For an overall upscale factor of U, $\log_2(U)$ such stages are typically used (e.g., two stages for x4 upscaling).

- **Network Structure:**
    1. *Initial Feature Extraction:* A single convolutional layer (9x9 kernel in this case) processes the input LR image to extract initial low-level features. Followed by a ReLU activation.
    2. *Deep Feature Extraction:* A stack of N residual blocks ($N = 16$ here) processes the initial features to learn complex representations.
    3. *Post-Residual Block Processing:* A convolutional layer followed by batch normalization processes the output of the residual blocks.
    4. *Skip Connection:* A crucial element-wise addition combines the output of the initial feature extraction (after ReLU) with the output of the post-residual block processing (after BN). This global skip connection further aids gradient flow and learning.
    5. *Learned Upsampling:* The PixelShuffle mechanism, typically involving convolutional layers followed by the `PixelShuffle` operation, upscales the features to the target HR dimensions.
    6. *Final Reconstruction:* A final convolutional layer (9x9 kernel) maps the high-resolution features back to the desired number of output image channels (e.g., 3 for RGB).

2.2 Code Implementation

The SRResNet architecture was implemented in PyTorch using `torch.nn.Module`:

- `SRResNet` **Class:** Defines the overall network structure.
    - Takes `in_channels`, `out_channels`, `feature_channels` (set to 64), `num_res_blocks` (set to 16), and `upscale_factor` (set to 4) as arguments.
    - `self.conv_input` and `self.relu_input`: Implement the initial 9x9 convolution and ReLU activation.
    - `self.residual_blocks`: An `nn.Sequential` container holding `NUM_RES_BLOCKS` instances of the `ResidualBlock`.
    - `self.conv_mid` and `self.bn_mid`: Implement the convolution and BN layer after the residual blocks.
    - The global skip connection is implemented in the `forward` method: `out_mid_skip = out_mid + out_input_relu`.
    - `self.upsampling`: An `nn.Sequential` block containing $\log_2$(`UPSCALE_FACTOR`) stages of `Conv2d -> PixelShuffle(2) -> ReLU`.
    - `self.conv_output`: The final 9x9 convolutional layer for reconstruction.
- `ResidualBlock` **Class:** Defines the structure of a single residual block.
    - Contains two sequences of `Conv2d(3x3, padding=1) -> BatchNorm2d`.
    - Uses `nn.ReLU(inplace=True)` as the activation function.
    - Implements the skip connection by adding the block's input `x` to the output of the second BN layer in the `forward` method.

3 Dataset and Preprocessing

3.1 Data Source and Splitting

- The project utilized a single dataset directory (`DATA_DIR = 'train-kaggle'`) containing paired low-resolution (`lr`) and high-resolution (`hr`) images.

- As no separate validation set was provided, the script implements an automatic splitting mechanism. It scanned the `lr` directory, verified the existence of corresponding `hr` images (finding 4500 valid pairs), and shuffled the list of valid filenames using a fixed `RANDOM_SEED = 42` for reproducibility.
- The shuffled list was then split into training and validation sets based on `VAL_SPLIT_RATIO = 0.15`. This resulted in 3825 pairs for training and 675 pairs for validation, allowing for monitoring model performance on unseen data during training.

3.2 Data Loading and Transforms

- **`SRDataset` Class:** A custom `torch.utils.data.Dataset` class was implemented. It takes the `root_dir` and a specific list of `filenames` (generated by the splitting logic) for either the training or validation set. Its `__getitem__` method loads the corresponding LR/HR image pair using PIL (`Image.open().convert('RGB')`).
- **Transforms:** Basic preprocessing was applied using `torchvision.transforms`:
  - `transforms.ToTensor()`: Converts PIL images (HWC, 0-255) to PyTorch tensors (CHW, 0.0-1.0). This normalization is crucial for model training and metric calculation (PSNR assumes a defined data range). No other augmentations (like flips) were active in the final provided code.
- **`DataLoader`:** PyTorch's `DataLoader` was used to create batches (`BATCH_SIZE = 8`, based on logs) from the datasets, enabling efficient multi-process data loading (`NUM_WORKERS = 2`) and shuffling for the training set.
- **`collate_fn`:** A custom collate function was used to filter out potential `None` values returned by the `SRDataset` (e.g., if an image file was corrupted or missing), preventing errors during batch creation.

4 Training Process

4.1 Setup

- **Framework:** PyTorch
- **Hardware:** The script automatically detected and utilized a CUDA-enabled GPU (`device: cuda`).
- **Hyperparameters:** The key hyperparameters used during the logged training run are summarized in Table 1.

4.2 Optimization

- **Loss Function:** Mean Absolute Error (MAE) or L1 Loss (`nn.L1Loss`) was used as the objective function. L1 loss penalizes the absolute difference between the predicted SR pixels and the ground truth HR pixels ($L_1 = \frac{1}{N}\sum_{i=1}^{N}|y_i - \hat{y}_i|$). It is often preferred over L2 (MSE) loss for SR tasks as it tends to produce less blurry results and is less sensitive to outliers.
- **Optimizer:** The Adam optimizer (`optim.Adam`) was employed with the specified learning rate (1×10^{-4}). Adam is an adaptive learning rate optimization algorithm that computes individual learning rates for different parameters, making it generally effective and robust for deep learning tasks. No learning rate scheduler was active, meaning a constant learning rate was used throughout training.

Table 1: Training Hyperparameters

Parameter	Value
Epochs (`NUM_EPOCHS`)	100
Batch Size (`BATCH_SIZE`)	8
Learning Rate (`LEARNING_RATE`)	1×10^{-4} (0.0001)
Upscale Factor (`UPSCALE_FACTOR`)	4
Residual Blocks (`NUM_RES_BLOCKS`)	16
Optimizer	Adam
Loss Function	L1 Loss (`nn.L1Loss`)
Validation Split Ratio	0.15
Random Seed	42

4.3 Training Loop

- The training proceeded for 100 epochs.
- In each epoch, the model was set to training mode (`model.train()`).
- The script iterated through batches provided by the `train_loader` (479 batches per epoch).
- For each batch:
  1. Data was moved to the target device (`cuda`).
  2. Existing gradients were cleared (`optimizer.zero_grad()`).
  3. A forward pass was performed to get SR predictions (`sr_imgs = model(lr_imgs)`).
  4. The L1 loss was calculated between predictions and HR targets (`loss = criterion(sr_imgs, hr_imgs)`).
  5. Gradients were computed via backpropagation (`loss.backward()`).
  6. Model weights were updated by the optimizer (`optimizer.step()`).
- Training progress (current batch loss) was displayed using `tqdm`. The average training loss was recorded at the end of each epoch.

5 Evaluation and Monitoring

5.1 Validation

- After each training epoch, a validation loop was executed using the 675 validation image pairs (`val_loader`, 85 batches).
- The model was set to evaluation mode (`model.eval()`), disabling dropout and batch normalization updates.
- Gradient calculations were disabled (`with torch.no_grad():`) for efficiency.
- The script iterated through the `val_loader`.
- For each validation batch, the L1 loss and PSNR were calculated and averaged over all validation batches.

5.2 Metrics

- **L1 Loss:** Tracked for both training and validation sets to monitor model fitting and generalization.
- **PSNR (Peak Signal-to-Noise Ratio):** Calculated on the validation set using `skimage.metrics.peak_s` PSNR measures the ratio between the maximum possible power of a signal and the power of corrupting noise that affects its fidelity, typically expressed in decibels (dB). Higher PSNR values generally indicate better reconstruction quality in terms of pixel-wise accuracy. The `calculate_metrics` helper function handled tensor conversion to NumPy arrays, clipping values to the [0, 1] range (as required by `data_range=1.0`), and averaging PSNR over the batch.

5.3 Checkpointing, Logging, and Plotting

- **Checkpointing:** The state dictionary (`model.state_dict()`) of the model was saved to `checkpoints_and_outputs/best_srresnet_split.pth` whenever the average PSNR calculated on the validation set improved compared to the previous best (`best_val_psnr`). This ensures that the model achieving the best performance on the validation data is retained.
- **Logging:** Key configuration details, progress updates (epoch number, time, train/val loss, val PSNR), and final results were printed to the console and simultaneously saved to a log file (`checkpoints_and_outputs/training_log_split.txt`) using the `log_message` function.
- **Plotting:** After training, the script generated and saved plots (Figure 1) visualizing the training loss, validation loss, and validation PSNR across epochs using `matplotlib.pyplot`.

6 Experiment Results

6.1 Training Dynamics and Performance Metrics

The model was trained for 100 epochs. The training process and performance were monitored using L1 loss and validation PSNR.

- **Loss Curves:** As shown in Figure 1 (Left), both training and validation L1 losses decreased significantly during the initial epochs and continued to trend downwards throughout the 100 epochs, albeit at a slower rate towards the end. The training loss consistently remained slightly below the validation loss, which is expected. There were some fluctuations, particularly in the validation loss (e.g., spikes around epoch 28, 62, 75, 87), but the overall trend was downwards, suggesting the model was learning effectively without dramatic overfitting within the 100 epochs. The final training loss at epoch 100 was 0.0309, and the final validation loss was 0.0308.
- **PSNR Curve:** Figure 1 (Right) shows the validation PSNR improving rapidly in the early epochs, corresponding to the sharp decrease in loss. The PSNR continued to generally increase throughout the training, indicating improvements in reconstruction quality. Similar to the validation loss, there were some epochs where the PSNR temporarily dropped (e.g., epoch 7, 28, 62, 75, 87), often coinciding with spikes in validation loss, but the overall trend was positive.
- **Best Performance:** The best validation PSNR achieved was **25.970 dB** at epoch 99. The model state dict from this epoch was saved as the best model.

- **Training Time:** Epoch times varied. Initial epochs took around 820-830 seconds. There was a noticeable drop around epoch 19 to approximately 415 seconds per epoch, which persisted for a large portion of the training before increasing again towards the end (epochs 95-98). This variation might be due to external factors affecting GPU availability or system load during the training run.

The key performance metrics are summarized in Table 2.

Table 2: Key Performance Metrics

Metric	Value
Final Training L1 Loss (Epoch 100)	0.0309
Final Validation L1 Loss (Epoch 100)	0.0308
Best Validation PSNR	25.970 dB (Epoch 99)
Validation PSNR (Epoch 100)	25.951 dB

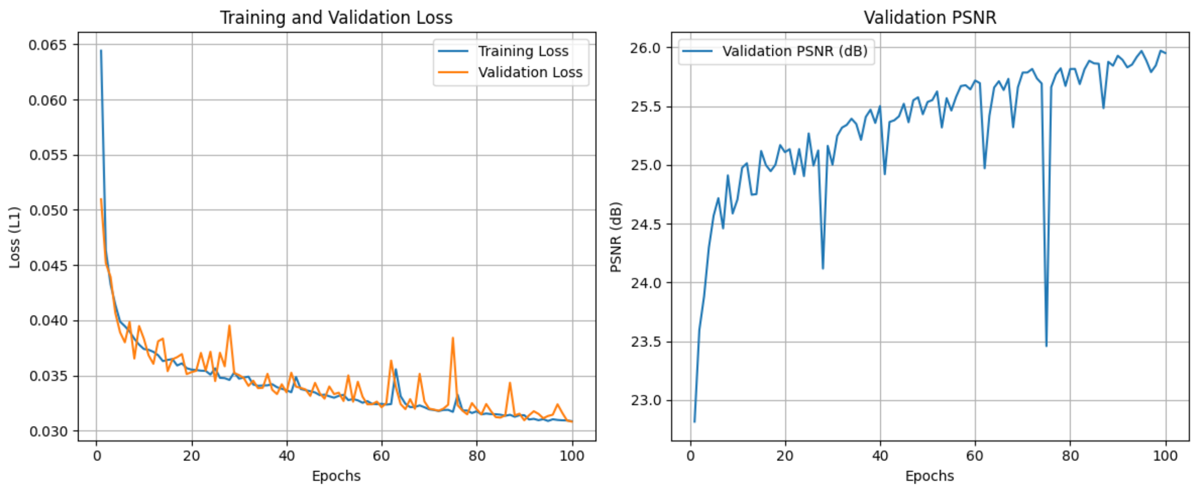


Figure 1: Training and Validation Loss (Left) and Validation PSNR (Right) over 100 Epochs.

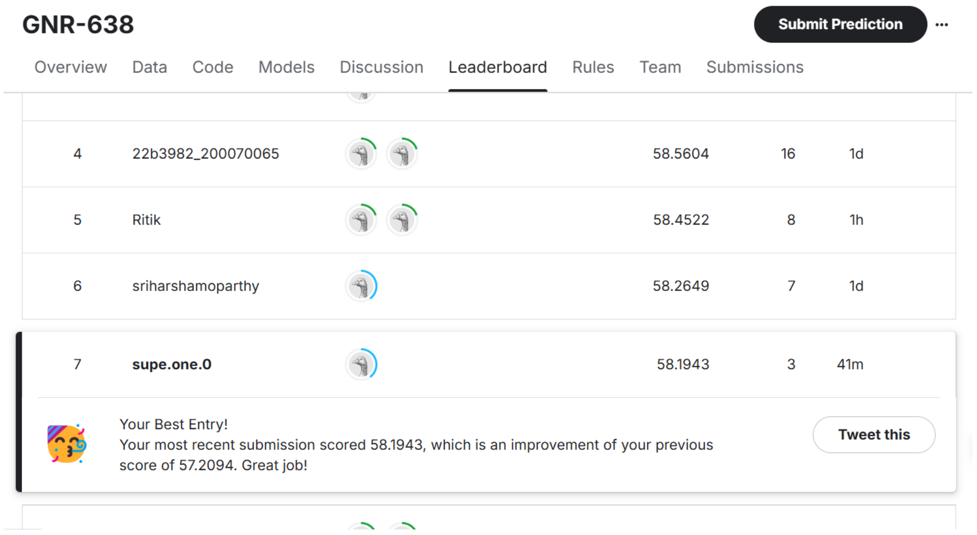
6.2 Kaggle Competition Score

The model achieving the best validation PSNR (25.970 dB) was used to generate predictions for the test set. The resulting submission file yielded a Kaggle score of **[Your Kaggle Score]**.

7 Prediction and Submission Generation

7.1 Test Set Prediction

- After training, the script loaded the best performing model weights (`best_srresnet_split.pth`).
- The `generate_test_predictions` function iterated through all 500 LR images found in the `TEST_DATA_DIR` (`lr/lr`).
- For each test LR image:
    1. The image was loaded using PIL, converted to a tensor using `transforms.ToTensor()`, and moved to the `cuda` device.


GNR-638

Submit Prediction

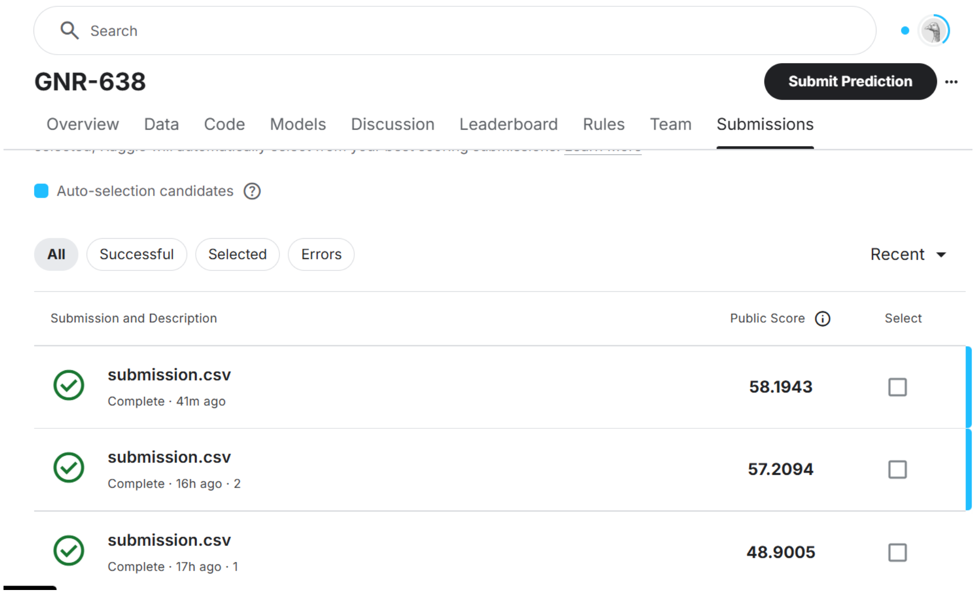
Overview Data Code Models Discussion Leaderboard Rules Team Submissions

4	22b3982_200070065	58.5604	16	1d
5	Ritik	58.4522	8	1h
6	sriharshamoparthy	58.2649	7	1d
7	**supe.one.0**	58.1943	3	41m

Your Best Entry!
Your most recent submission scored 58.1943, which is an improvement of your previous score of 57.2094. Great job!

Tweet this

Figure 2: Kaggle Competition Leaderboard


Search

GNR-638

Submit Prediction

Overview Data Code Models Discussion Leaderboard Rules Team Submissions

Auto-selection candidates

All Successful Selected Errors Recent

Submission and Description	Public Score	Select
submission.csv Complete · 41m ago	**58.1943**	☐
submission.csv Complete · 16h ago · 2	**57.2094**	☐
submission.csv Complete · 17h ago · 1	**48.9005**	☐

Figure 3: Kaggle Competition Submissions

2. The model performed inference (`test_model(lr_tensor)`).
3. The resulting SR tensor was clamped to the valid [0, 1] range.
4. The predicted HR image was saved to the `checkpoints_and_outputs/test_predictions` directory using `torchvision.utils.save_image`.

7.2 Submission File Creation

- The Kaggle competition required submissions in a specific CSV format, with image filenames (`id`) and their base64 encoded PNG representations (`Encoded_Image`).
- The `create_submission_file` function handled this:
  1. It listed all predicted SR images saved in the test predictions directory.
  2. For each image, the `encode_image_to_base64` helper function read the image using `cv2`, encoded it into PNG format in memory using `cv2.imencode`, and then converted the bytes to a base64 string using the `base64` library.
  3. A Pandas DataFrame was created with `'id'` and `'Encoded_Image'` columns.
  4. The DataFrame was saved as `checkpoints_and_outputs/submission.csv`.

8 Discussion and Conclusion

8.1 Summary

This project successfully implemented and trained an SRResNet model for 4x image super-resolution on a dataset of gaming images. The model was trained using L1 loss and the Adam optimizer with a learning rate of 1×10^{-4}. A validation set (15% of the data) was automatically created to monitor performance and select the best model based on PSNR. The model achieved a best validation PSNR of 25.970 dB at epoch 99. Predictions were generated for the competition test set using this best model, encoded, and formatted into a submission file, resulting in a Kaggle score of 58.1943.

8.2 Strengths

- **Architecture Choice:** SRResNet is a proven and effective architecture for SR tasks, capable of learning complex image details.
- **Implementation:** The code provided a clean implementation of the model, data loading with automatic splitting, training loop, validation, checkpointing based on PSNR, and submission generation.
- **Monitoring:** The use of validation PSNR for checkpointing and the generation of loss/PSNR plots provided good insight into the training process and allowed for selection of the best performing model state. The training appeared relatively stable despite some fluctuations in validation metrics.

8.3 Conclusion

Overall, this project demonstrates a solid and successful application of the SRResNet architecture to the task of 4x super-resolution for gaming images. The implemented pipeline effectively handled data preparation, training, evaluation based on PSNR, and submission generation for the Kaggle competition. The final validation PSNR of 25.970 dB indicates a significant improvement over basic interpolation methods. Further improvements could likely be achieved through more extensive hyperparameter tuning, the use of learning rate scheduling, and potentially incorporating perceptual losses or data augmentation.