ChatGPT

# Gram-GAN Methodology vs Implementation

**Model architecture:** The Gram-GAN paper specifies that the *generator* should be an ESRGAN-style RRDB network, and the *discriminator* a VGG-style convolutional network [1]. In particular, the authors state "the generator uses the RRDB [11]... and the discriminator uses a variant of the VGG network" [1]. Any faithful implementation must replicate this. If the provided code instead uses a different generator (e.g. SRResNet or a smaller model) or a standard ESRGAN discriminator, it diverges from the paper. Likewise, the discriminator should output feature maps at intermediate layers (not just the final score) so that those features can be used in the perceptual loss [2]. In summary, the network should match the ESRGAN backbone for the generator and a VGG-style ConvNet for the discriminator as described [1].

**Patch-wise texture (Gram) loss:** Gram-GAN's key novelty is one-to-many supervision by matching textures via Gram matrices. Concretely, for each predicted patch $e_i$ the method constructs a *Gram matrix* $G(p)=p^\top p$ from original feature vectors of patches [3]. The paper forms a **candidate database** of patches containing (1) the ground-truth patch, (2) a downsampled ground-truth patch, and (3) randomly *affine-warped* versions of the ground-truth patch [4] [5]. It then selects the candidate patch whose Gram matrix is closest (in Euclidean norm) to the Gram of the predicted patch [6] [4]. The "patch-wise texture loss" is then computed between the predicted patch and this best candidate patch [5].

If the implementation omits this mechanism, it fails to capture the one-to-many supervision. For example, if the code simply uses the ground-truth patch (one-to-one) or selects a patch by raw pixel distance rather than Gram distance, it contradicts the paper's method [7] [5]. A faithful implementation should **compute Gram matrices for each patch**, include affine transformations of the GT patch in the candidate set, and then enforce a loss $$L_{patch} = |\,Gram(e_i) - Gram(p_i^*)|^2,$$ *where* $p_i^*$ is the selected patch as described (see Eq.4 in the paper) [5] [3]. If any of these steps (Gram computation, patch database construction, affine augmentation) are missing or simplified, the implementation deviates from the published method.

**Discriminator perceptual loss:** Gram-GAN adds a novel perceptual loss using the discriminator's hidden features. The paper states: "the discriminator in each iteration is used to extract features, and the discriminator perceptual loss... is represented as (5)" [2]. In practice, this means selecting certain convolutional layers of D (the paper uses the 5th and 11th conv layers) and penalizing differences between real and fake features at those layers. An implementation should forward both the generator output and the ground truth through D, capture the specified intermediate feature maps, and include a loss term $L_{disc_perc} = \sum_k |D_k(G(z)) - D_k(\text{GT})|^2$. If the code does not extract and compare discriminator features (for example, if it only uses the final discriminator score), then it omits this loss component. The absence of this term would contradict the claim that "discriminator features can highlight differences... to improve quality" [8]. A correct implementation must therefore hook the discriminator's 5th and 11th layer activations (after their nonlinearities) and include their MSE difference in the generator's loss, as per the paper [2].

**VGG perceptual and content losses:** In addition to discriminator features, the model uses traditional VGG-based perceptual loss and an $L_1$ content loss. The VGG loss compares features from pretrained VGG19

(layers conv2_2, conv4_2, conv4_4 with weights 1.0, 0.1, 0.01) [9] . The content loss is simply the $\ell_1$ distance between SR and HR images [10] . Any faithful code should implement these exactly: e.g. using a VGG19 model (frozen) and weighting each feature layer as specified. If the implementation uses different layers or omits the VGG loss entirely, that is a simplification. Likewise, replacing $\ell_1$ with $\ell_2$ would conflict with the rationale (the paper explicitly justifies $\ell_1$ to be "insensitive to outliers" in GAN training) [10] . These losses must be included with the paper's stated weights in the overall objective (Eq.12) [11] .

**Adversarial (RaGAN) loss:** Gram-GAN employs a **relativistic average GAN** loss with a special masking scheme [12] [13] . Specifically, Eqs.(7–9) in the paper define a Relativistic Average GAN discriminator loss, and Eq.(10) defines a binary mask $M(x)=\mathbf{1}{\sigma(\Omega(x))>\epsilon}$ based on the standard deviation $\sigma$ of a random patch of the image. In each iteration, real and fake images are partially masked (the "true" patch is masked out) according to $M$ before computing the GAN loss [13] . The implementation should therefore (1) compute this random patch mask at each step, (2) apply it to real and SR images, and (3) use the RaGAN loss formulas. If the provided code instead uses a plain GAN loss (e.g. non-relativistic BCE) or neglects the mask, it departs from the method. For example, the mask generation (std threshold = 0.005, patch size 4×4) is quite unusual and unlikely to appear unless explicitly coded. Any code lacking this mechanism should be noted as missing a key component of the SR-GAN design [13] .

**Training strategy and scheduler:** The paper reports training with Adam ($\beta_1=0.9,\beta_2=0.999$), batch size 8, input patches of size 48×48 (HR 192×192), with random flips/rotations [14] . Critically, they use a **cyclical learning rate**: 600k iterations total, restarting every 200k, with warm-up and cosine decay from an initial $1\times10^{-4}$ [15] . Implementation should mirror this: e.g. a PyTorch scheduler like `CosineAnnealingWarmRestarts` or a manual warm-up schedule. If the code uses a fixed learning rate or a simple step decay, that is a simplification. It should also perform the stated data augmentation (flip/rotate), which is standard. The mention that "the magnitude of random affine transform was 0.003" [15] likely refers to the slight affine warp of patches in the supervision database – if the code omits generating random affine variations for patch supervision, this is another missing detail.

**Patch data handling:** According to the paper, DIV2K training images were cropped by a sliding window into non-overlapping 192×192 patches (yielding 44,226 patches) [16] . The code should either pre-crop the dataset or generate equivalent random crops on-the-fly. More importantly, for the one-to-many patch supervision, the paper implies constructing a small per-patch database (GT patch, down-sampled, affined). If the implementation simply loads HR patches and never explicitly forms this "patch database," then the one-to-many mechanism isn't realized. In summary, the code must handle patches of size 4×4 (for mask and Gram) and include the generation of affine variants of each GT patch for supervision [15] [4] ; absence of these steps is inconsistent with the method.

**Inconsistencies and missing components:** Based on the above points, any deviations in the code are critical. In particular, if the code lacks the Gram-matrix-based patch selection, it has effectively reduced the model to one-to-one supervision (like a standard SRGAN) rather than the intended one-to-many [7] [5] . If discriminator perceptual loss is not implemented, the "discriminator perceptual loss" contribution is missing [2] . If the adversarial mask is omitted, the RaGAN loss used is simpler than claimed [13] . And if the LR schedule does not use warm-up and cosine restarts, training dynamics differ from the paper's description [17] .

**Recommendations:** To fully match the paper, the implementation should be modified as follows:

- **Patch supervision:** Compute $G(e_i)$ and $G(p)$ (Gram matrices) and choose the patch $p^$ *minimizing $|G(e_i)-G(p)|$ from the candidate set (GT, downscaled GT, and randomly affined GT patches)* [3] [5]. *Add this patch-wise loss $L_{patch}$ = |e_i - p^|^2$* (or Gram-difference) to the generator loss as in Eq.(4). Generate affine variants by applying small random affine warps (variance 0.003) to the GT patch [4].

- **Discriminator features:** Hook the discriminator's 5th and 11th convolutional layer outputs during each forward pass. Compute $L_{disc_perc} = |D_5(G(z)) - D_5(\text{GT})|^2 + |D_{11}(G(z)) - D_{11}(\text{GT})|^2$ and add it (with appropriate weight) to the total generator loss [2]. Ensure gradients do not update D during this loss.

- **Adversarial mask:** For each real/fake image pair, select a random 4×4 patch location, compute its pixel-std, and form the binary mask $M(x)=\mathbf{1}{\sigma(\text{patch})>0.005}$. Apply $M$ to the real image and $1-M$ to the fake (or vice versa, as per Eq.9). Then compute the RaGAN losses using masked inputs [13]. In PyTorch, this could be done by elementwise multiplying images by the mask. This implements Eqs.(7–10).

- **Learning rate schedule:** Use an Adam optimizer ($\beta_1=0.9,\beta_2=0.999$) with an LR scheduler that performs warm-up from 0 to $10^{-4}$, then cosine decay resetting every 200k iterations [15].

By adding these components, the code will faithfully implement the Gram-GAN methodology. Failure to include any of the above would mean the implementation only partially realizes the paper's contributions. Each modification above is grounded in the original description [5] [2] [13] and is necessary to replicate the reported results.

**Sources:** The analysis above references the Gram-GAN paper's descriptions of architecture, losses, and training protocol [1] [3] [5] [2] [9] [12] [10] [15], which should be compared line-by-line against the implementation to identify any inconsistencies.

---

[1] [2] [3] [4] [5] [6] [7] [8] [9] [10] [11] [12] [13] [14] [15] [16] [17] Gram-GAN: Image Super-Resolution Based on Gram Matrix and Discriminator Perceptual Loss
https://www.mdpi.com/1424-8220/23/4/2098