大纲

1. 比赛简介 / 榜单说明
2. 技术路线的迭代史
3. 我的心得体会
4. Top 方案分享
5. 经验总结

技术路线 (简单 + 2轮)

Text
- SDXL Turbo 模型选择
  - 高质量模型
  - 快速
- 蒸馏
  - 步骤
  - 卷积
  - 图片尺寸
- 改进提示词 (简单 易操作)
  - prompt
  - Lora
- 改 SVG 结构 (小规模 外部)
  - 结构顺序
  - 坐标运算方法
    - 数据类型
    - 浮点运算
    - 数据表示法
    - 字符表达方式

SVG

Image
- 指标迭代
  - OCR : (对抗幻觉)
    - 加空白
    - 加背景
  - Vqa
    - proxy 问题迭代
    - Hacking 问题 : 加分项
  - aes
    - 迭代
    - Hacking : 加惩罚

我的结果 :

榜1 : (46)

主要用 : SDXL-Turbo
- 1. 使用多个 lora, 每个榜单 1 个权重
- 2. 增加 SD2.1 , SDXL , SD35-medium , SD3.5-large 替换下原 SDXL Turbo 速度变慢 不符合赛题

图转 SVG : 基于公开源码的速度优化 (解决 numpy 瓶颎)

渲染 : ocr 识别 : 在图像左边加白边 用 YY

评估 :
- vqa 分 : 通过 proxy 问题 、 Does <lower portray "{description}"? Answer yes or no.
- aes 分 : 官方 评分函数 vs vqa 分 — 计算 Y Token 概率
- 二者调参平衡

几个操作点 :
1. OCR 识别
2. SD2.1 → SDXL
3. Lora
4. SDXL Turbo 增加出图数量 (aes 倾向)
5. vqa-proxy 分
6. 使用多个 lora

榜2 : 33/12

metric

SVG

Image

- 提高渲染速度 → ocr_score
- 其它模型替换 → aes_score ; vqa_score

1st Place Solution

New Team Name - 1st Place Solution

Congratulations to the other phenomenal finalists in Stage 3! Looking forward to pushing such a unique competition - it was a lot of fun to participate in! Thanks also to my teammate @kaggle.org and @kaggle.com !

TL;DR

The notebook for run top scoring submission can be found here.

1. Generate 64 baseline images with Argmax-OCR Baseline.
2. Convert each image to an SVG.
3. Add the SVG data to a number of other candidate strings to each SVG.
4. Compute the CLIP for images, select the image producing graphics, and calculate their aesthetic scores.
5. Select the SVG with highest aesthetic score.

OCR Exploits

Avoiding Hallucination

The model frequently hallucinates text on the corner of each SVG to score OCR hallucinations. Blurring models reduce these artifacts, but not enough to be competitive. Instead, I simply crop out the bottom 20% of the image before OCR. This significantly improves OCR accuracy, and produces the highest average aesthetic scores without requiring edit distance.

Using Additional Text

The most lightweight and reliable method of improving the OCR score is to add more text related to the prompt to the image. However, adding text to the image reduces the aesthetic score. To avoid this, I place the text on a large white background, and crop the image to the bounding box containing non-white pixels. This preserves the aesthetic score while improving OCR accuracy.

```
<svg viewBox="0 0 256 256" xmlns="http://www.w3.org/2000/svg">
  <g transform="
    translate(-1.7,-1.7)
    translate(-0.3,-0.3)
    translate(0.0,0.0)
    translate(0.3,0.3)
    translate(1.7,1.7)
    translate(128,128)
    scale(0.992,0.992)
    translate(-128,-128)
  ">
    <path d="M0,0 L256,0 L256,256 L0,256 Z" fill="rgb(255,255,255)" />
  </g>
</svg>
```


This results in text which is hidden in both the raw and partially processed image (since it is cropped out), but still scores, reveals the hidden text.

Image Generation and Conversion

Diffusion Models

We use Stanford's SDXL-Turbo to generate 64 images for the given description of the target SVG.

I initially ran SDXL-Turbo on Kaggle T4s following the following steps:

1. Replace the default VAE with madebyollin's fp16-fixed version for faster inference on T4s.
2. Use a CPU offload, multiple inference step, and bfloat16 tensors.
3. Enable Pytorch's SDP MATH, EFFICIENT and FLASH SDPA kernels.
4. Run torch.compile() on the UNet, VAE decoder, and text encoder.

Each image is generated at a resolution of 384x384 with a guidance scale of 0.0, using the following prompts:

Prompt

```
flat vector illustration, flat color blocks, hard color boundaries, solid fills, clipart style, 2-
d graphic, no texture, no shading, minimalist, simple design, vertical center, horizontal center,
"{TEXT}", {color_1} at {pos_1}, {color_2} at {pos_2}, {color_3} at {pos_3}
```

Negative Prompt

```
photorealistic, realistic, detailed, 3d, complex shading, gradients, soft edges, text, signature,
watermark, texture, painting, blurry, jpeg artifacts, thin lines, dark, cluttered
```

With multi-threading, image generation takes ~5 seconds for 64 samples. However, due to the small threading loop currently present in Kaggle notebooks, we spend a lot on single threading for more reliable performance. Loading the pipeline and compiling the models takes roughly 150 seconds.

Conversion to SVG

Each image is converted to an SVG using Vectorizer.AI using its default settings. At each of the 4 corners, we add a 5-pixel radius circle in the image's dominant color. This improves compressibility and ensures the SVG covers the full canvas. We convert PNG images to JPEG with a quality of 90 before uploading them, since Vectorizer.AI charges per megapixel uploaded. This reduces costs. Vectorizer.AI also occasionally returns 500 errors, so we retry up to 3 times with exponential backoff. This process takes ~2.5 seconds for a batch of 64 images.

Examples of Generated Images and Converted SVGs

Examples of Generated Images and Converted SVGs

Constructing the SVG

We create the SVG we extracted from the generated image down to 90% of its original size and place it in the top center of our final SVG. We then iterate through the 64 candidate SVGs, adding each path to our final SVG, and calculate the CLIP for the resulting image. We then select the SVG with the highest CLIP score.

Finally, we add a number of other artifacts to the SVG which are intended to improve the aesthetic score (although negatively impacting the OCR score).

1. Add 3 randomly sized circles.
2. 2-3 paths of colored squares (we test four different variations of these colored squares for each generated image).
3. Green text saying "YOU".
4. Green text saying "DOG".
5. Green text saying "ZOOM".
6. Green text saying "CRAWLE DE ARTIST" at sufficient space is available after the SVG description text.

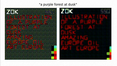
Each of these artifacts were found using either a genetic algorithm, a grid-style search, or occasionally a manual sweep and choice approach.

Examples of Fully Constructed SVGs (before and after processing)

Callback Drills

In case of timeouts or exceptions we also prepare a simple base solution. I initially used a single dot SVG hard-coded into the text from the SVG description with a phrase suffix ("Intentional Artwork 48, Self Expression") and a few SP the (#0000ff/#ffffff/#ff0000) and (#000000/#ffffff/#ff0000) color palettes.

We didn't submit much time refining these fallback SVGs, but the later stages of the competition as our highest scoring submissions didn't make any use of them.



Final SVG Selection

We convert some of the SVGs back to images and apply the original evaluation metric to them. We use a custom implementation of the image preprocessing and CLIP model from HuggingFace transformers library, and torch.compile() the image encoder. We also cache the text embeddings for the prompts and simply return the first one we have cached when we encounter a prompt we have already seen. This improves our score on the private leaderboard from 0.58 to 0.59. Although this doesn't matter for this competition, it also significantly improves our runtime, which is critical for multi-day competitions, since it allows us to iterate much more quickly.

What Didn't Work

We experimented with many different techniques and variations on the above approach that didn't result in improved scores.

1. Incorporating OLIP scores, VQA scores, etc. into the leaderboard. We weren't able to find OLIP for leaderboard.
2. Generating more than 64 images (although on the private LB this seemed to score quite well).
3. Using more than 2 paths for the SVG. Our score peaked at 2-3 paths, and degraded with more complex paths.
4. Different parameters for Vectorizer.AI, including different square strong thresholds, or the self-trace flag, etc.

Bonus: Old Meme Solution

For the last week we submitted a meme submission to find out if CLIP could assume that produced the highest aesthetic score with some small text. Based on my leaderboard hacking, this actually had a good score of 0.504 with an aesthetic score of -7.12.

2nd Place Solution

Thank you to Kaggle for hosting such an interesting competition and providing a great platform for everyone. I would also like to thank my two teammates @kaggle.com and @kaggle.com, they accomplished a great deal of incredible work (Thank you again!

Initially, the most already, we used Stable Diffusion to generate images and Lucent Vectorizer to convert them into SVGs. We had three @kaggle.com submissions, one using Stable Diffusion 1.5, one using Stable Diffusion 2.1, and one using Stable Diffusion XL.

I started contributing some improvements. The idea was very simple: in the first version, the CPU spent a lot of time waiting for the GPU to finish processing the diffusion model, so we definitely needed to focus on CPU work and the only way to achieve a high score on the Kaggle GPU was to make the CPU process fast enough to keep the GPU busy. I optimized the code, switched to OpenCV, and implemented batch processing. This allowed us to generate more images in the same amount of time, which significantly improved our score. Later, I also added Flash Attention and torch.compile() to further reduce generation time. I also made the code more robust, added more error handling, and improved logging. This allowed us to run longer, more stable experiments and iterate much faster. Overall, these changes helped us move from the 1st place to the 2nd place on the leaderboard.

Later, I also added a simple evolutionary search on top of the pipeline. Instead of generating a single SVG per prompt, we started generating several candidates and selecting the best one based on a quick aesthetic evaluation. This gave us a small but consistent boost in quality without significantly slowing down the pipeline. Most of the time, the extra compute was used to explore slightly better outputs rather than just speeding up a single run.

Unfortunately, there was a bug in the evaluation code that I didn't notice at the time. Instead of using the official CLIP implementation, I used a custom version that was not exactly the same as the one used by the organizers. Because of this, my local validation scores were not fully reliable, and I ended up overfitting to a metric that was slightly different from the real one. In hindsight, I should have spent more time verifying the evaluation setup and relying more on the leaderboard feedback. Despite that, the overall approach still worked well enough to reach 2nd place, and I learned a lot about how small implementation details can matter a lot in competitions like this.

3rd place solution: VQA/AES=0.81/0.64 Diffusion model + differentiable SVG optimization

My specialty in competitions is usually the hacks that wins continuously (some hard to reproduce for others), making this competition interesting and challenging. But learned a lot about Kaggle generation, which we leverage that advice entering the competition.

Overall pipeline is simple:

Generate prompt -> generate image (Flux.1-schnell) -> trace image with DiffVG -> optimize curves -> convert to SVG

Images of each step

Original image generated by Flux

Original SVG image generated by DiffVG

SVG image after backpoint optimization

SVG image after backpoint optimization and SVG Shape Optimizer (final score for VQA evaluation)

We are excited to share our approach both because it's unusual (inspired by the "Painting with Light" competition). This competition challenges us to generate Portable Vector Graphics (SVG) from text prompts, adhering to strict constraints on file size and complexity, while maximizing visual fidelity and aesthetic appeal. The solution is a multi-stage pipeline that combines the strengths of diffusion models, gradient-based vector optimization, and differentiable rendering. We believe this approach offers a unique perspective on text-to-vector generation, leveraging techniques from computer vision and graphics optimization.

Overall Pipeline

Our core idea was to first generate high-quality bitmap images that capture the essence of the prompt, convert it into an initial SVG, and then refine and optimize this SVG to maximize its alignment to the prompt, its aesthetic quality, and its adherence to the competition's VQA metrics. We employ a multi-stage process:

1. **Bitmap Generation (Flux)**: We use the black-forest-labs/FLux.1-schnell model to generate candidate bitmap images.
   - **Prompt Engineering**: The prompt is enriched with keywords, randomly chosen from predefined lists, including:
     - "{description}", the raw prompt.
     - "{adjective} {description}", adding an adjective.
     - "Palette: {color1} {color2} {color3}", with colorful background, beautiful background.
     - "{adjective} {description} on {description2}", with background of vibrant {flat design illustration|3d render|oil painting|watercolor painting}.
   - We generate 2 images per prompt keywords, resulting in 8 bitmap images for each input description.
2. **Initial SVG Conversion** (vtracer):
   - Each bitmap image is converted to an SVG using the vtracer (0.0.32) binary with the following parameters: `--clustering_threshold 45 --mode polygon --filter_speckle 4 --path_precision 5 --corner_threshold 60 --length_threshold 4.0 --max_iterations 2 --splice_threshold 45 --layer_difference 16 --hierarchical stacked --color_precision 6`. These parameters aim for a balance between detail and simplicity.
   - These raw SVGs are stored in a dataset.
3. **Backpoint Optimization** (custom scripts):
   - This initial SVGs were evaluated using a custom aesthetic predictor based on CLIP-L/14 and a linear head, trained on the LAION Aesthetic V2 dataset. The predictor outputs a score between 0 and 10, reflecting the image's aesthetic quality.
   - These top 2 SVG, original bitmap and underlined CLIP aesthetic (normalized) (10 scale) are used for further optimization.
   - Two candidate SVG underwent backpoint optimization, a gradient-based optimization step inspired by the Painting with Light competition's approach.
4. **Backpoint Optimization** (custom scripts) & **Gradient Descent**:
   - This step utilizes PyTorch and differentiable rendering (DiffVG) to adjust the control points of the Bézier curves defining the SVG paths, with the goal of maximizing the aesthetic score and minimizing file size.
   - The loss function combines three components:
     - **Aesthetic Loss**: Negative output of the aesthetic predictor, encouraging higher aesthetic scores.
     - **Size Penalty**: Proportional to the number of points in the SVG paths, discouraging overly complex shapes.
     - **Curvature Regularization**: Penalizes sharp angles, promoting smoother curves.
   - Gradients are clipped to a maximum norm of 1.0 to prevent instability.
   - The optimizer is an Adam optimizer with a learning rate of 0.01.
   - The process runs for 300 iterations.
   - To bridge the gap between our optimization process and the competition's evaluation, we implemented StraightToSelectTorch. This function mimics pixelization and JPEG compression artifacts, ensuring that what we optimize is what the competition sees.
   - Crucially, these operations need differentiable. We provide differentiable versions of standard image operations (resize, blur, sharpen, JPEG compression) using PyTorch, enabling gradients to flow through the entire process.
   - Gradient-free versions of these operations are used for inference.
   - The top two candidates are then rendered to PNG using CairoSVG, resized to 256x256 pixels, and saved for the next stage.
5. **Final SVG Selection and Scoring**:
   - After optimization, the SVG underwent several post-processing steps to reduce file size and improve robustness to other rendering requirements.
   - This included:
     - Rounding coordinates to 1 decimal place.
     - Removing empty groups.
     - Removing default attributes.
     - Converting absolute path commands to relative.
     - Merging consecutive path elements with the same attributes.
     - Removing unused XML namespaces.
     - Minifying the SVG by removing whitespace.
     - Calculating the final score using the official competition metrics: LAION-Aesthetic score (CLIP-based aesthetic predictor) and OCR accuracy (PaddleOCR). The final score is a weighted combination of these two metrics.

Key Insights and Observations

- **Initial Quality Matters**: The quality of the initial bitmap significantly impacts the final result. Better initial images lead to better SVGs after optimization.
- **Gradient Descent Works**: Gradient descent can effectively optimize SVG paths to improve aesthetic scores and adherence to the VQA metrics.
- **Simplicity is Key**: The size penalty is important to prevent the optimization from generating overly complex SVGs that are difficult to render and score poorly on the size metric.
- **Robustness is Important**: The post-processing steps are crucial for ensuring that the SVGs are compatible with various rendering engines.

Acknowledgements

We would like to thank the organizers for their work in creating this challenging and interesting competition, especially since this marks a rapidly rising and so much more fun than a "Kaggle" competition. We also extend our gratitude to the developers of the following libraries and models: Flux.1-schnell, vtracer, DiffVG, LAION Aesthetic V2, PaddleOCR, and CairoSVG.

4th Place Solution: SD3.5M + DiffaFT + diffvg

Hey! Glad to share the engineering-heavy approach I used for this competition. It's more like a hacker's guide than a research paper. Nothing fancy, just a lot of GPU time and a few tricks to fix common failure modes.

Overall approach

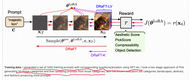
My pipeline is pretty straightforward: generate bitmap images, then trace them with DiffVG to optimize high-quality, smooth splines/paths. The magic comes from aesthetically-guided Best-of-N sampling and a custom OCR loss when optimizing with DiffVG.

Diffusion Model

After experimenting with a few models, I decided to use Stable Diffusion 3.5 Medium with a custom LoRA for faster inference. It has a relatively small memory footprint, which is important for free tuning.

DiffaFT Fine-Tuning

I trained a LoRA on top of the base SD3.5 model using DiffaFT, which proposes Diffusion Adversarial Post-Training (DAPT). I chose a two-stage approach of first training with MSE loss for 1000 steps, then switching to the full DiffaFT loss for another 1000 steps. This helped stabilize the initial training and improve the final quality.



Training data: I generated a set of 1000 training images using the base SD3.5 model with the prompts provided by the organizers. Crucially, I also saved the initial noise tensors for each image. This allows me to perform the DAPT update efficiently, as I can calculate the reward for the generated image and use it to update the model without needing to run the full diffusion process.

Reward functions: I used two general reward functions that I have worked well from my previous work on DiffaFT:

- Human Preference Score v2
- PickScore

These models were trained on human preference data. I also added three competition-specific rewards:

- The competition aesthetic score
- A binary OCR reward (0/1) for whether the text is legible (using PaddleOCR). This was faster and sometimes easier to work better than using the raw OCR score.
- A negative L2 loss between the image and a grayscale, to encourage more vibrant colors.

Results: This fine-tuning improved the competition aesthetic score from 6.2 to 6.4. The percent of valid outputs (no illegible text) increased from 0.6 to 0.8 (specifically on the training prompts). I found that using a high CFG scale (9.0) and a low number of steps (10) was optimal for this task, as it maximized the reward signal. There are some examples of generations before/after the fine-tuning using the same random seed:

SVG Conversion

I used a simple bitmap tracing pipeline based on OpenCV contours and Douglas-Peucker simplification. The optimization happens in DiffVG, where we can apply gradients from our aesthetic and OCR losses to the control points of the splines. This is much more powerful than just tracing the pixels.

Best of n

For each prompt, my notebook keeps generating SVGs until roughly 90 seconds have passed, with some weights towards a whether it is before or after the 60 second mark. Then it selects the best one based on a weighted sum of the aesthetic score and the OCR score. This ensures we get a good result quickly, but still have time to try for a better one if the first few attempts are difficult.

diffvg : bitmap → svg

可微渲染器 (梯度渲染器)

进行优化

5th place solution - VTracer and DiffVG optimization

I want to thank the organizers for providing such an interesting and challenging competition, especially since this marks a rapidly rising and so much more fun than a "Kaggle" competition.

My final solution is based on generating a bunch of images with Flux.1-schnell, converting them to SVG, and then adding a bunch of random shapes to the SVG. I then optimize the SVG using DiffVG, with a loss function that combines the competition aesthetic score and OCR accuracy. Finally, I select the best image based on the final score.

Adding text to the image and using DiffVG to optimize the text improved OCR accuracy significantly.

The VQA score was suffering by adding the patch, partly because it uses around 224x224pixels which cannot be used further in the original image. Unfortunately and because it's a big chunk of image, it pays. However, the model that scores will be higher by quite a lot, especially when the patch is not visible. I tried to remove the patch but it hurt the score.

Final solution logic and library versions are in the codebase/logic.

Example:

Generating Images

I used Flux.1-schnell. I just asked GPT4o mini to generate a bunch of images. I tried Sadtalker other models. Featuring SDXL turbo for a conference session using the competition score, featuring using BERT, and image clusters. GPT mini was as cheap as.

The best overall generation I could get was using Flux. I tried quantized versions, hoping to get a significant speed. Naturally I tried Flux.1-schnell-gguf-q4_0 and Flux.1-schnell-gguf-q8_0. Unfortunately, the speed improvement was not significant, and the quality was worse. I tried Flux.1-schnell-int8 and Flux.1-schnell-int4 which, like before the results, were worse using the benchmark.

What prompts focus on shapes, SVG-like images that can be easily converted to SVG using Vtrace. Also, explicitly adding "(2d, 2d2, vector, vector)" to the prompt improves consistency. Improves the results a little bit.

Noise to vector

Instead of using the generated images to SVGs, I initialized the images generated by primitives per the round shifter than Prompt when round starting. But DiffVG is slower and good enough so it is on time. I have over 30 seconds to generate 50 SVG. Optimize. So a single image takes around 1.5 seconds. I tried 100 steps, but it didn't improve the score.

On the other hand, I was able to convert an image using a single color in less than 2 seconds, while the results were almost as good. After adding some random shapes, also clipping, and tuning the parameters a bit, I was able to generate good enough SVGs with around 60 seconds. Adding a patch of random color to the image's background, and crop it out on Vtracer

Optimizing the text image

After generating all the SVGs, the remainder of the time was spent using some functions with the VQA score. I simply removed the method Wordsearch this paper was used for "Text" or "Not" generated by the competition. "Triangle/square" or Spiral/Circle" Basic SVGs. Image optimization was the best for "Text" and "Not".

This actually turned out a lot better than generating positions, colors, and shapes using DiffVG and using the original SDE score. I believe this is because the prompt forces it much more closed when we can only evaluate a few questions, but I could be wrong and by question.

Although only three tasks optimize the SVG, this can be done also combined with the main one. Any original SVG made SVG. Add text always better, even if much slower and using less prompts because of that.

Optimizing the pipeline

After I noticed bottlenecking would be quite powerful, I had to find ways to efficiently use all bigger resources. I was mainly concerning Office Hours, and I noticed that the bottleneck was **not** Flux VTrace, generating images with SDXL turbo was actually faster than running Vtrace on a single image.

In the end, I was able to use almost 50% of the resources by running Vtrace processes in the background

```
./diffvg_venv/bin/ python
export DIFFVG_CUDA=1
export LD_LIBRARY_PATH=/usr/lib/x86_64-linux-gnu:$LD_LIBRARY_PATH
./vtracer_venv/bin/ python
```

Especially increasing the first with the main instances. For example, whenever a few images are ready from SDXL, we start converting using Vtrace. Although, I didn't notice any improvement, I thought it would be better to do so.

Also, I was able to use all available CPU cores. Thus, I was able to run 20 processes at the same time. Although, I didn't notice any improvement, I thought it would be better to do so.

Finding the aesthetic score

In order to find the patch, I tried many differentiable versions of every step of the competition's aesthetic score, but DiffVG, With this differentiable version, I was able to find the patch. Although, it was very slow, it was worth it.

- Optimized on score directly. Patch found in a few minutes.
- Add text to the image and optimize it. Patch found in under an hour.
- Find random patches and optimize the aesthetic score alongside it. Patch found.
- Optimize a proxy function (ssim/mse) with the VQA model to find the best SVG to use.
- Optimize a proxy function (ssim/mse) with the VQA model to find the best SVG to use.
- Find the Triangle/Circle/Square/Spiral/Random Patch and optimize it with the differentiable VQA score.

As of those solutions do work, but they are either too slow, requires too much GPU memory to run on Kaggle, generate SVGs way too large. Thus, I ran out of time to make sure they will work well. In the end, I think it is something similar to what other people did in optimizing shapes, but I was able to make it work well and fast enough to be competitive.

The best solution I could find was to add around a small subset of patch that, when added and the background of the end score, will maximize the aesthetic score. I added 10 of the Triangle/Square/Circle/Spiral/Random Patches. The number of patches was found by trial and error. Then, I optimized the image and the patches for 100 steps. Because of the cropping, the portion of the patch that is visible is optimized to be aesthetically pleasing, and the rest is optimized for OCR.

Because the patch covers a lot of pixels, I had to impose a penalty also on that pixels. The bigger the patch, the less information it has could use to convert the SVG. I was able to get patches with over 0.8 aesthetic score by using much more pixels, but then the VQA score went too high.

Although, I tried many different ways to optimize the patch, the best one was just using the original aesthetic score. I tried to add a penalty about amount of pixels that it is covering, aesthetic, aesthetic with total pixels in a grid the size of 100x100 pixels, and all possible errors. This **hackish** shows the process. It has to have found a form a good aesthetic patch since it is extremely important in the final score.

Bonus

I think the crucial thing I think was to plug the differentiable pipeline into **BEST** and optimize the diffusion process directly against that loss. It would be a huge engineering lift, but it would be the most "correct" way to do it. Instead of just using the final image, you would backpropagate through all the steps of the sampler, so the model would learn to generate latents that are specifically designed to be easy for the VTracer and DiffVG to process and score well on the aesthetic metric.

The **notebook** shows the improvement from the first generated image from SDXL and the images in between, optimizing the metric alongside the way to the diffusion process itself.

经验总结

1. 比赛早期加入
2. 第一个月 : 读目 + 调参 + 无脑想也行
3. 第二个月 : 搭通完善 DL 迭代优化 稳定性
4. 从公开 code 区学 Discussion 区 寻找突破
5. 工程能力