Stable Diffusion XL Fine-Tuning

Prepared By: Adam Łucek

Video Walkthrough

Make YOUR OWN Images With Stable Diffusion - Fi…

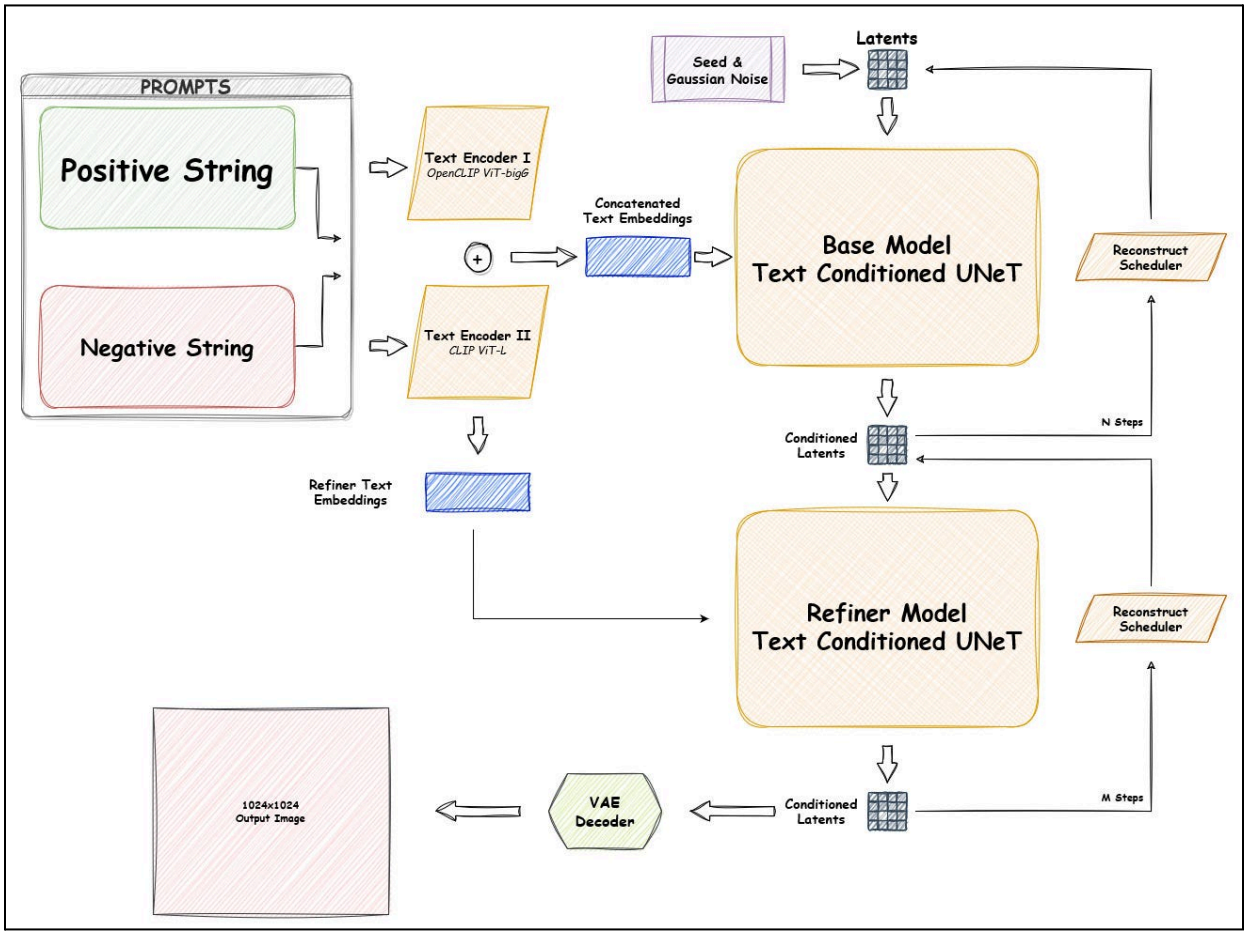
On Style

“Old Book Illustrations”

On Objects

“My Green Chair”

Understanding Stable Diffusion XL



https://towardsdatascience.com/the-arrival-of-sdxl-1-0-4e739d5cc6c7

Prompts: Inputs to the model consisting of a "Positive String" and a "Negative String", which guide the model on what to include or avoid in the generated image.

Text Encoder I & II: These are different encoders (e.g., OpenCLIP-VT/G and CLIP-VT/L) used to transform the text inputs into embeddings. The embeddings from these encoders are then concatenated to form a rich representation of the text input.

Refiner Text Embeddings: A refinement step that processes the concatenated embeddings to optimize or enhance the information they contain, making them more suitable for generating conditioned latents.

Seed & Gaussian Noise: Initial random inputs that, along with text embeddings, help in generating the initial latent representations of the image.

Base Model Text Conditioned UNeT: A UNeT-based model (a type of convolutional neural network that follows a U-shaped architecture, with an encoder for downsampling to capture context and a decoder for upsampling to precisely localize features) that takes the initial latents and the refined text embeddings to generate an initial version of the conditioned latents. This step includes a "Reconstruct Scheduler" that determines how the latent space is iteratively refined across several steps.

Refiner Model Text Conditioned UNeT: An additional refinement stage using a UNeT model that further processes the conditioned latents to enhance the final image output, involving multiple iterations as governed by another "Reconstruct Scheduler".

VAE Decoder: A variational autoencoder (VAE) decoder that converts the final conditioned latents into the pixel space, resulting in the generation of the final output image, typically at a high resolution like 1024x1024.

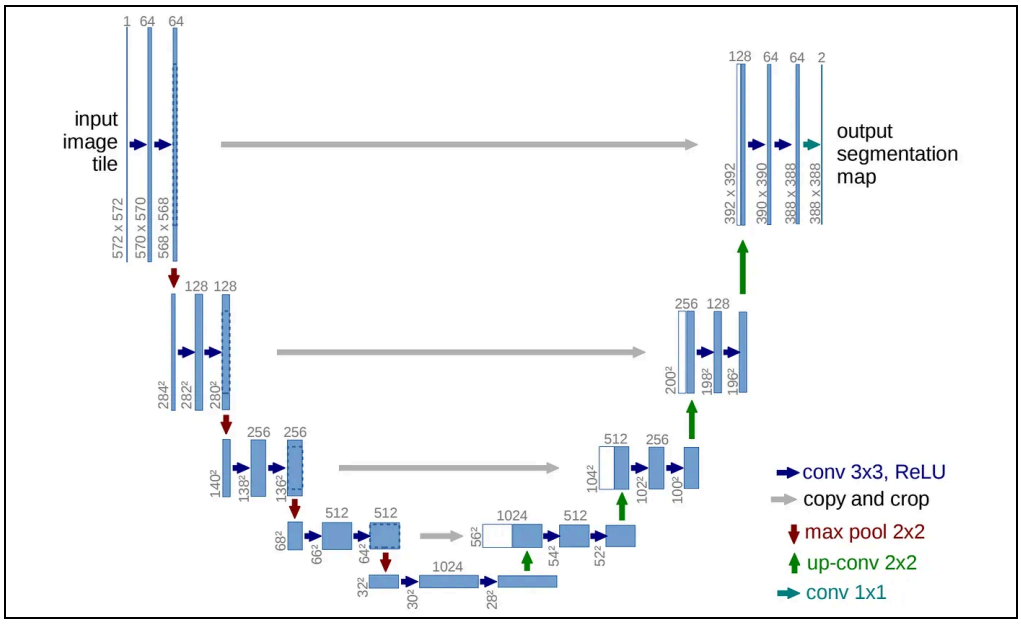
For our fine tuning efforts, we will be focusing on modifying three parts of this architecture to get our desired results

- **Base Model:** SDXL-Base-1.0
- **Text Encoder I:** OpenCLIP-VIT/G
- **Text Encoder II:** CLIP-VIT/L

By modifying just these parts, we'll be able to get noticeable results when training on our own dataset!

Fine Tuning Methodology

Text-to-Image Model Training

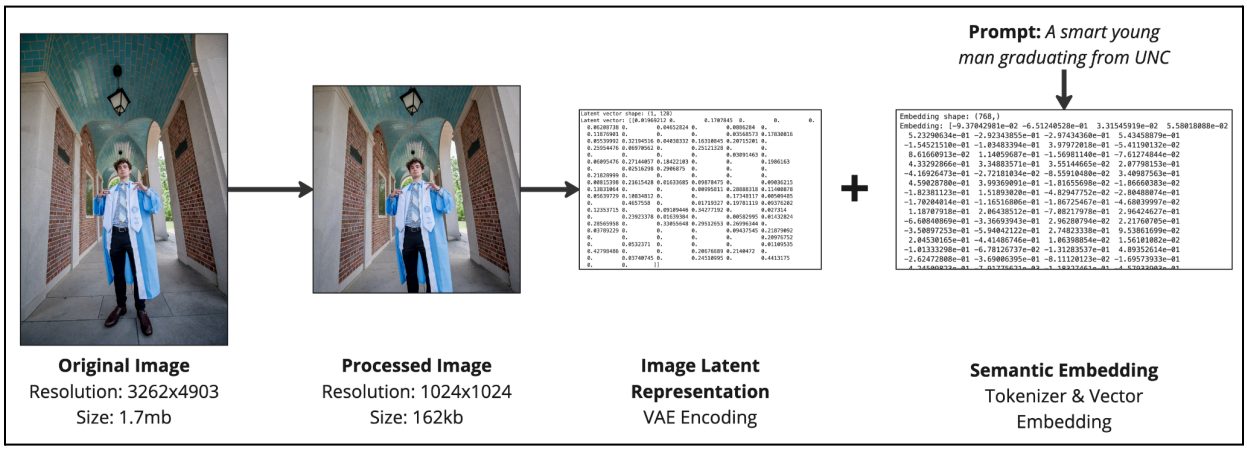


https://towardsdatascience.com/you-cant-spell-diffusion-without-u-60635f569579

The fine tuning process is mostly focused on training the UNeT, which in our case will be the base model shown in the diagram above. The overall process for training the entire UNeT tends to follow a process similar to this:

1. **Loading Pre-Trained Model**: As we're fine tuning the model, we want to re-use what components we can. In the case of SDXL, we want to load the main base UNeT, tokenizers, text encoders I & II, and the variational autoencoder.
2. **Pre-Process Data**: The training data will be two parts, an image and a corresponding caption. We want to ensure that the data is good to go through this process, so it first undergoes some pre-processing. This includes:
    a. Resizing, cropping, cutting, and flipping of the images to the desired resolution to ensure uniformity of training data.
    b. Pre-computing the embeddings of both the text and the images.
        i. Text data is first tokenized (broken down into pieces, typically words or subwords), and each token is then transformed into a dense vector embedding that represents semantic information about the text.
        ii. The VAE takes an image and encodes it into a latent space—a compressed representation that retains the critical features of the image but in a more abstract form.

iii. Pre-computing these embeddings before training speeds up the overall training process instead of calculating embeddings on the fly during each training step.



3. **Setting & Loading Hyperparameters**: Instantiating all of your training hyperparameters, like epochs, batch size, learning rate, etc. (or tuning them from prior runs)

4. **Training the UNeT**:

    a. **Sample Noise**: Adding noise to the latent space introduces variability and robustness into the model, helping it to generate diverse and realistic images by preventing overfitting.

    b. **Sample Timestep**: Sampling timesteps randomly for each image helps simulate various stages of adding noise, which the model learns to reverse

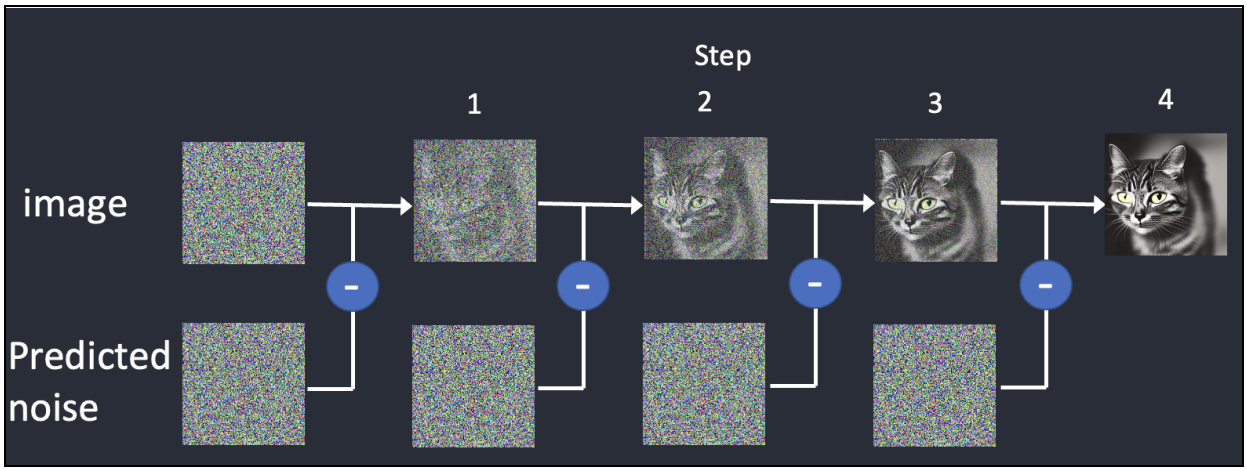
    c. **Add Noise to Image Input**: The noise added at each timestep simulates the forward diffusion process, and takes into consideration the magnitude of noise at the sampled timestep.

    d. **Predict Noise Residual**: The model predicts the difference between the noisy input and the clean image

    e. **Calculate Loss**: Measures how far the model's prediction of noise is from the actual noise that was added.

    f. **Backpropagate**: The gradients of the loss are calculated with respect to each parameter of the model. Using an optimizer like Adam, the model updates its weights based on the gradients to minimize the loss.

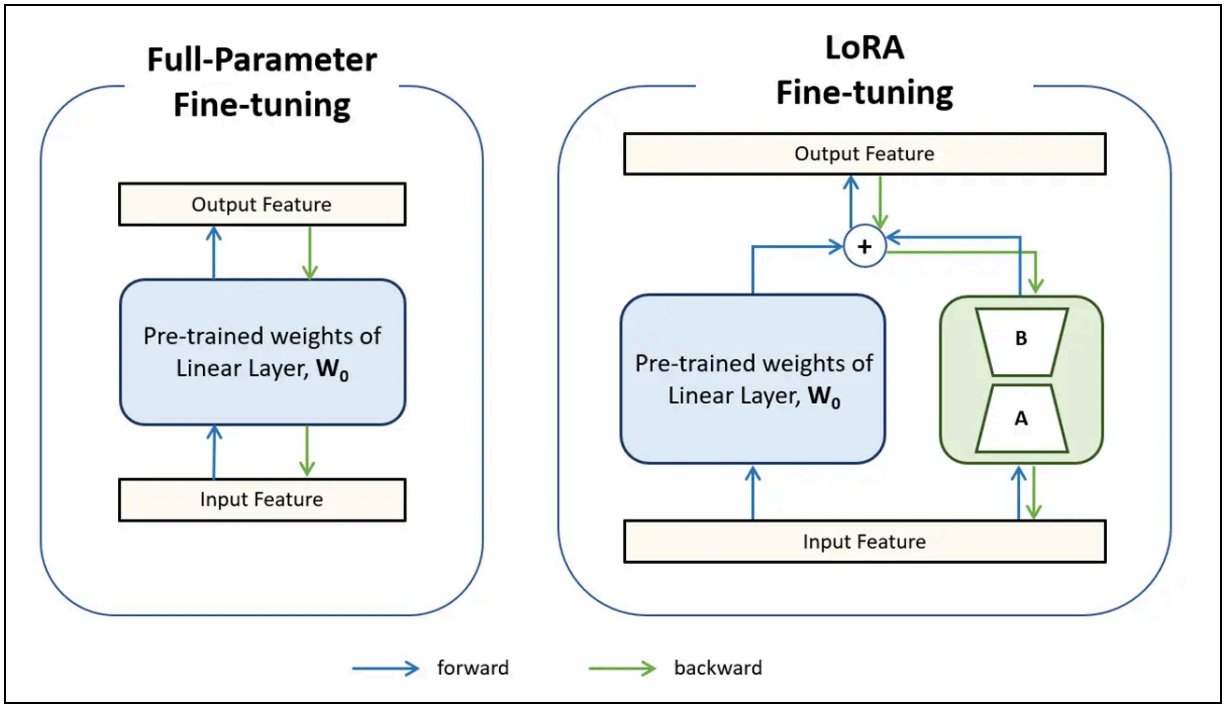
    g. **Repeat & Validate**: Repeating the training process with regular validations helps monitor the model's performance on unseen data, ensuring it generalizes well beyond the training set.



Following these general model training steps, we're able to perform text-to-image fine tuning with our own dataset on the entire base UNeT of the overall diffusion model.

Note that this is a process for training the **ENTIRE** model itself, in practice this can be expensive, tricky, and compute intensive. For more consumer available methods, we take advantage of "parameter efficient fine tuning" methods, defined below.

Low Rank Adaptation (LoRA) Fine-Tuning



As mentioned above in the training overview, it can be time, money, and resource intensive to fine tune an entire model's weights & biases, among other issues that can arise like:

- **Overfitting**: Training your weights too specifically on the training data, causing the model to reproduce or simply copy what it's been trained on. The goal for generative models is to "learn" from the data to generate new data on its own, not to copy the data itself.
- **Catastrophic Forgetting**: The tendency for neural networks to abruptly and drastically forget previously learned information upon learning new information.

To address these issues, researchers at Microsoft developed the Low Rank Adaptation, or **LoRA**, fine-tuning method.

Key Concepts of LoRA

Overview: LoRA (Low-Rank Adaptation) is a method designed to adapt large pre-trained models efficiently by modifying only a subset of their parameters. This allows these models to be fine-tuned for specific tasks without the extensive computational costs associated with traditional methods.

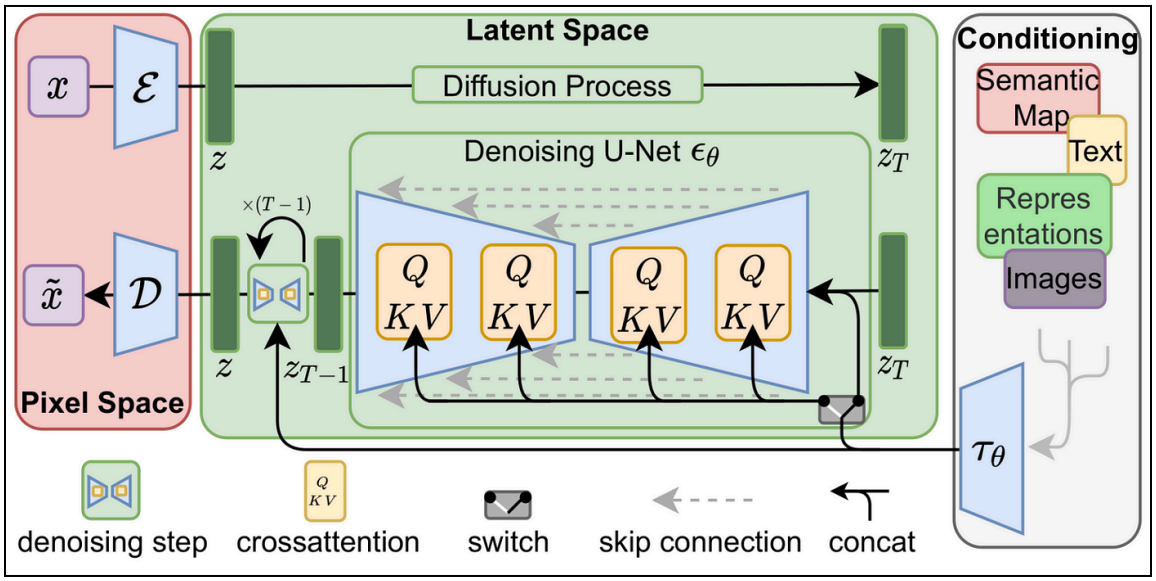
Core Components of LoRA:

1. **Pre-trained Model and LoRA Modules**:
    a. **Frozen Weights**: The core architecture of LoRA involves keeping the main pre-trained model weights unchanged (frozen), ensuring that the fundamental knowledge the model has learned remains intact.
    b. **Injectable Modules**: Rather than retraining all parameters, LoRA introduces trainable low-rank matrices into each layer of the Transformer architecture. These matrices are significantly smaller, reducing the computational load and simplifying the adaptation process.
2. **Low-Rank Parametrized Update Matrices**:
    a. **Full-Rank vs. Low-Rank**: Normally, neural networks utilize dense layers with full-rank weight matrices, which can be computationally intensive to update. LoRA, however, employs low-rank matrices that capture the essential transformations required for new tasks, effectively reducing the dimensionality and complexity of the updates.
    b. **Efficiency in Adaptation**: By using these low-rank matrices, LoRA can approximate the necessary changes in the model's behavior with fewer parameters, enhancing efficiency and reducing resource use.
3. **Task-Specific Adaptation without Full Fine-Tuning**:

a. **Selective Training**: LoRA focuses on training only the small, injected low-rank matrices, leaving the bulk of the model's weights unchanged.

b. **Reduced Resource Demands**: This targeted approach cuts down on the number of trainable parameters drastically, which lowers memory requirements and computational costs, making it feasible to adapt large models on less powerful hardware.

Application of LoRA in Transformers:



Adaptation of Self-Attention Mechanisms:

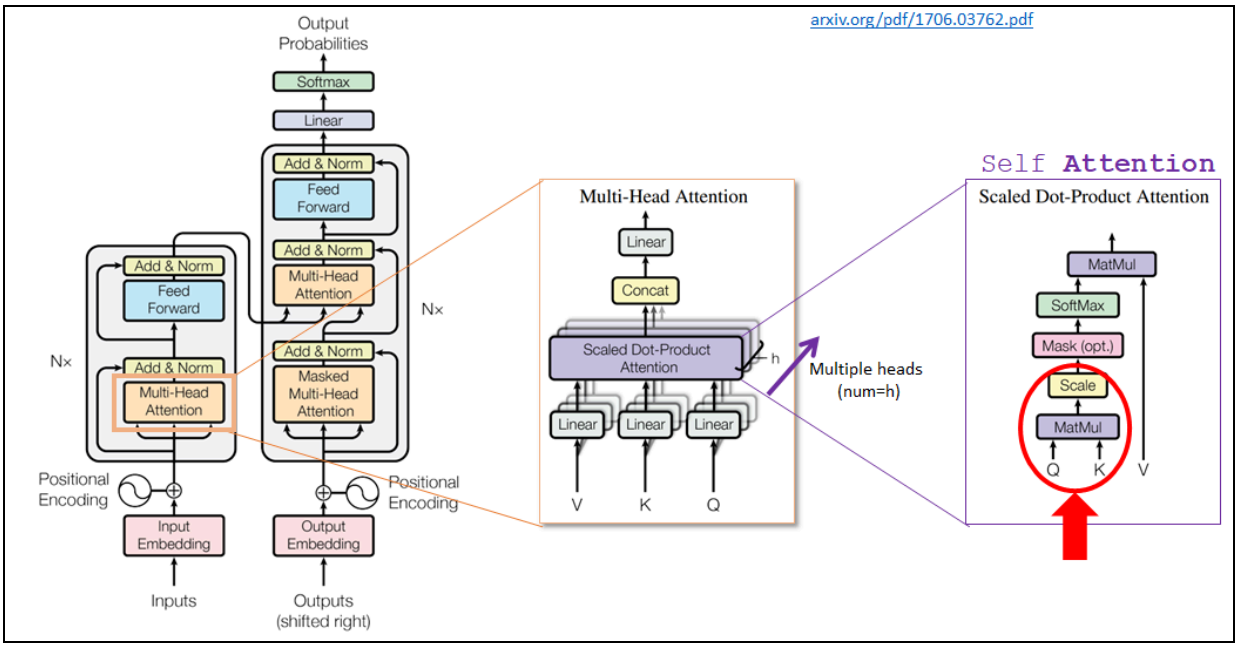
1. LoRA is specifically applied to the projection matrices within the Transformer's self-attention mechanism—namely the query, key, value, and output matrices. These matrices play crucial roles:

    a. **Query**: Determines how much attention to pay to each element in the input sequence.

    b. **Key**: Acts as a counterpart to queries to compute attention scores.

    c. **Value**: Contains the actual information from the input data that is retrieved after computing the attention.

    d. **Output**: Aggregates the weighted contributions from the 'value' matrix to form the output of the attention layer.



2. **Focus on Attention Weights**: The adaptation strategy primarily modifies the attention weights, which are critical for tailoring the model's responses to specific tasks. The MLP (multi-layer perceptron) modules remain unchanged to maintain the model's general capabilities while enhancing task-specific performance.

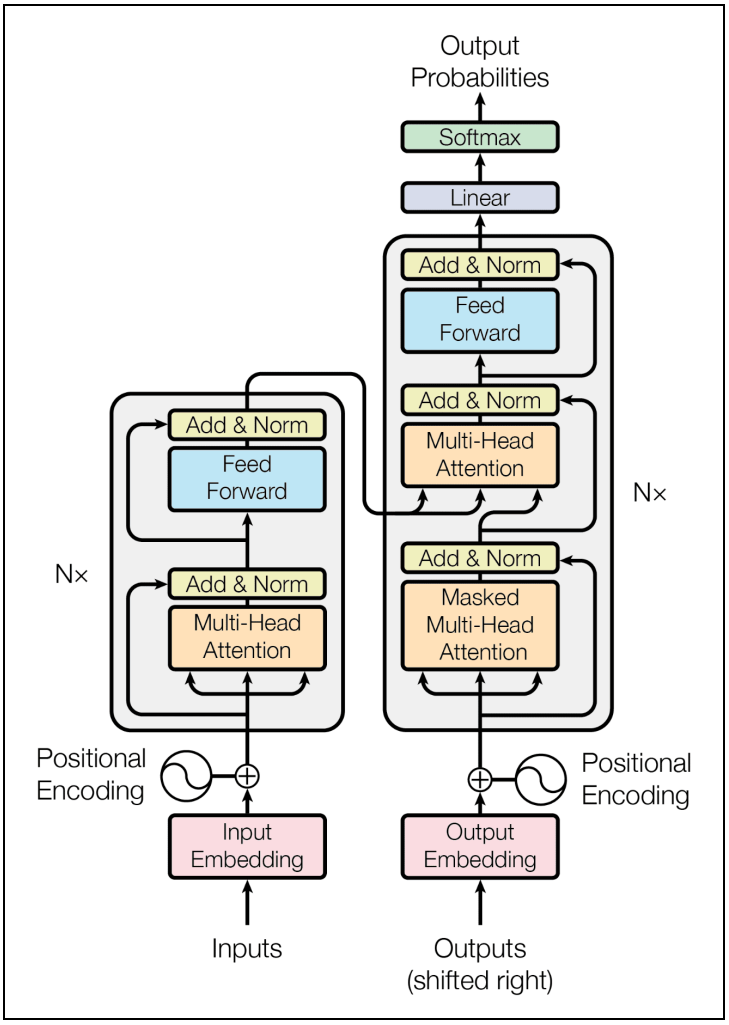
Supporting Context

1. **Transformer**:

    a. **Overview**: The Transformer is a type of neural network architecture that has become the foundation for many state-of-the-art models in natural language processing (NLP). It was introduced by Vaswani et al. in the paper "Attention is All You Need" in 2017.

    b. **Key Features**: Unlike previous models that relied heavily on sequence-based processing (e.g., RNNs and LSTMs), the Transformer uses a mechanism called self-attention to process all parts of the input data simultaneously. This allows for significantly improved parallelization during training and leads to better handling of long-range dependencies in data.

    c. **Impact**: Transformers have revolutionized the field of NLP, enabling the development of highly effective models like BERT, GPT series, and others that perform exceptionally well on a wide range of NLP tasks.

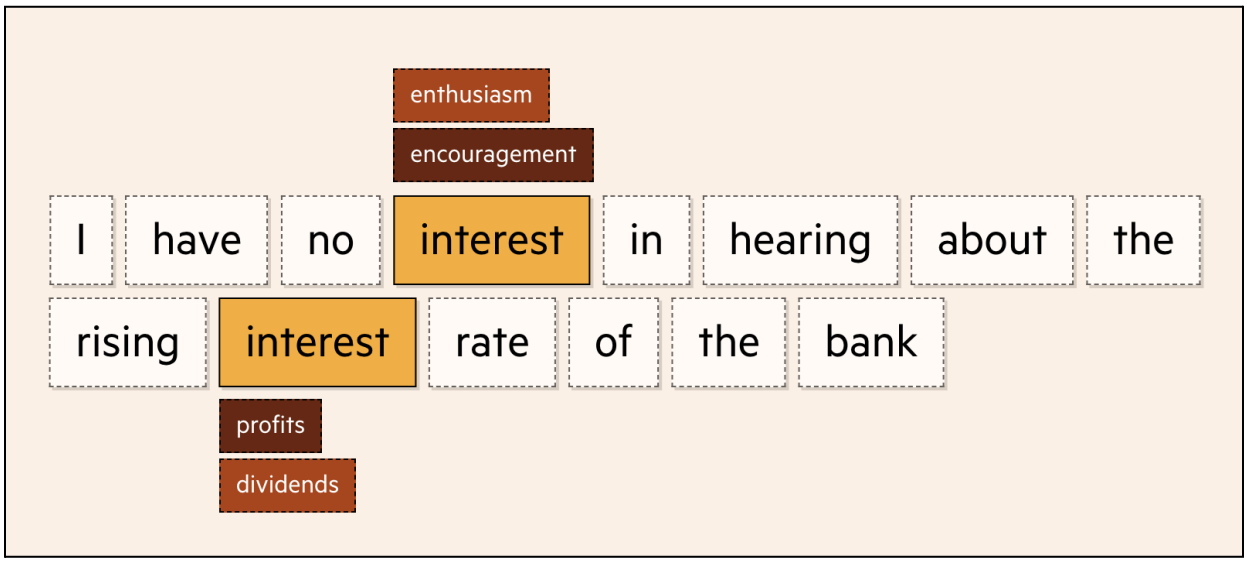


2. **Attention Mechanism**:

    a. **Purpose**: The attention mechanism allows models to focus on different parts of the input sequence when performing tasks like translation, summarization, or text generation. This mechanism is integral to the Transformer architecture.

    b. **How It Works**: In the context of Transformers, attention weights are computed between all pairs of input and output positions. The weights determine how much each part of the input should be considered for each output, allowing the model to dynamically prioritize which parts of the input are most relevant.

    c. **Self-Attention**: Specifically in Transformers, self-attention refers to attention mechanisms that relate different positions of a single sequence in order to compute a representation of the sequence. It helps the model to understand the context surrounding each word or token within the sequence.



3. **Rank:**

    a. **Mathematical Definition**: In linear algebra, the rank of a matrix is the maximum number of linearly independent column vectors in the matrix or, equivalently, the maximum number of linearly independent row vectors. Rank provides a measure of the information content of the matrix.

        i. **Simplified**: In the context of a matrix, which you can think of as a grid filled with numbers, the rank tells you how many different rows (or horizontal lines of numbers) or columns (vertical lines of numbers) you really need to keep the essential information in that grid. Each row or column must add new, unique information that isn't already provided by the others. So, the rank gives you the smallest number of rows or columns that are needed to maintain all the information in the matrix without any redundancy.

    b. **Relevance in Machine Learning**: In the context of machine learning, particularly in neural networks, reducing the rank of weight matrices (as in LoRA) can simplify the model by limiting the number of parameters that need to be trained. This reduction can lead to less overfitting and faster training, while still capturing the essential patterns in the data.

    c. **Low-Rank Approximations**: These are techniques used to approximate complex matrices with ones of lower rank. They are crucial in scenarios where computational efficiency is paramount, such as in deploying large models like GPT-3 using LoRA, where the intrinsic dimensionality of the problem allows for such approximations without significant loss of performance.

For example, the matrix A given by

$$A = \begin{bmatrix} 1 & 2 & 1 \\ -2 & -3 & 1 \\ 3 & 5 & 0 \end{bmatrix}$$

can be put in reduced row-echelon form by using the following elementary row operations:

$$\begin{bmatrix} 1 & 2 & 1 \\ -2 & -3 & 1 \\ 3 & 5 & 0 \end{bmatrix} \xrightarrow{2R_1+R_2\to R_2} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 3 \\ 3 & 5 & 0 \end{bmatrix} \xrightarrow{-3R_1+R_3\to R_3} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 3 \\ 0 & -1 & -3 \end{bmatrix}$$

$$\xrightarrow{R_2+R_3\to R_3} \begin{bmatrix} 1 & 2 & 1 \\ 0 & 1 & 3 \\ 0 & 0 & 0 \end{bmatrix} \xrightarrow{-2R_2+R_1\to R_1} \begin{bmatrix} 1 & 0 & -5 \\ 0 & 1 & 3 \\ 0 & 0 & 0 \end{bmatrix}.$$

The final matrix (in reduced row echelon form) has two non-zero rows and thus the rank of matrix A is 2.

LoRA simplifies the process of adapting large-scale models for new tasks by modifying only a fraction of the model's parameters, thus maintaining high efficiency without sacrificing performance.

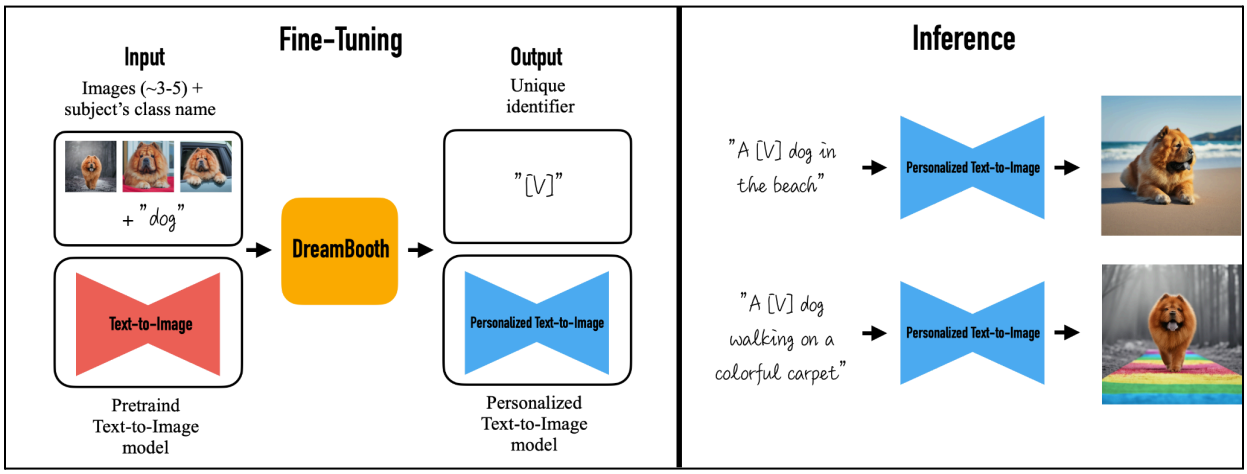
Our examples and applications for fine tuning will all take advantage of LoRA fine-tuning!

DreamBooth Fine-Tuning

While Text-to-Image training on the UNeT can be good for initially training a model to generate an image, or tune it closer towards a style/type/genre of image, it lacks one specific capability: reliable generation and recreation of specific subjects or objects. The DreamBooth method aims to take text-to-image training one step further to tackle this problem.

Key Concepts of DreamBooth

Overview: DreamBooth is a method from Google Research that takes text-to-image generation one step further by allowing personalized image synthesis with only minimal reference images. It extends traditional large-scale text-to-image models, which, while capable of generating diverse images from text prompts, generally fail to accurately recreate and recontextualize specific subjects, or objects.



1. **Embedding Subjects Using Unique Identifiers**

    a. **Image-Text Pairing**: The model is fine-tuned with text prompts that include the unique identifier along with a class noun that describes the subject (e.g., "a [unique_token] dog", commonly used tokens include: sks, [V], T0K). This pairs the visual features of the subject with the text-based identifier.

    b. **Impact**: This process embeds the subject into the model's output domain (set of possible outputs that the model can produce), enabling it to generate this subject in various scenarios when prompted with the identifier.

2. **Expanding the Language-Vision Dictionary**

    a. **Definition**: The Language-Vision Dictionary It's a conceptual framework within the model that links textual descriptions (language) to visual representations (vision). In the context of DreamBooth, this dictionary is expanded to include the new unique identifiers that are linked to specific subjects.

    b. **Expansion Technique**: By training the model on image-text pairs that include the unique identifiers, DreamBooth effectively "teaches" the model to associate these new tokens with specific visual characteristics of the subjects.

    c. **Generation**: When generating new images, the model uses these learned associations to accurately recreate the subject in response to prompts that include the unique identifier.

3. **Implementing Class-Specific Prior Preservation Loss**:

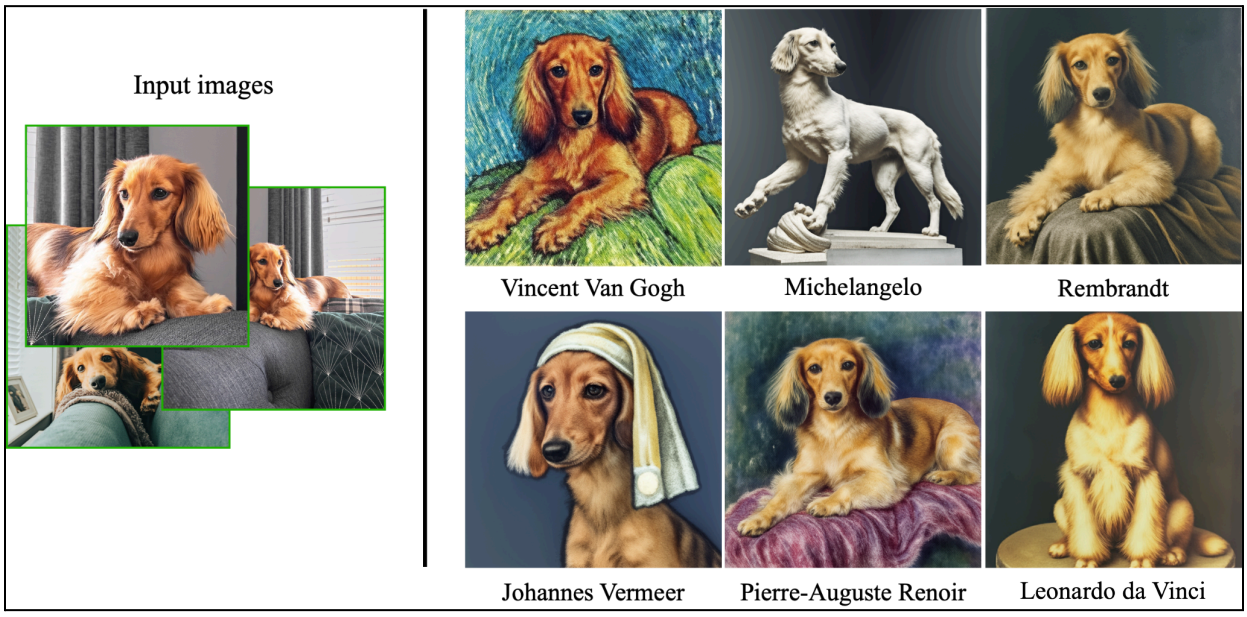
    a. **Implementation**: The CS-PPL loss function is designed to balance the model's ability to generate the specific subject with its general capability to produce diverse images from the same class. It operates by comparing the outputs generated from prompts with and without the unique identifiers. It ensures that while the model becomes better at generating the specific subject, it does not forget how to generate other plausible instances from the same class.

    b. **Purpose**: To prevent the model from losing its ability to generate other instances of the same class/subject (language drift) and to maintain high diversity in the outputs (no overfitting). The loss adjusts the model's training process to ensure that the introduction of a unique identifier does not skew the model's overall output diversity or its understanding of the class.

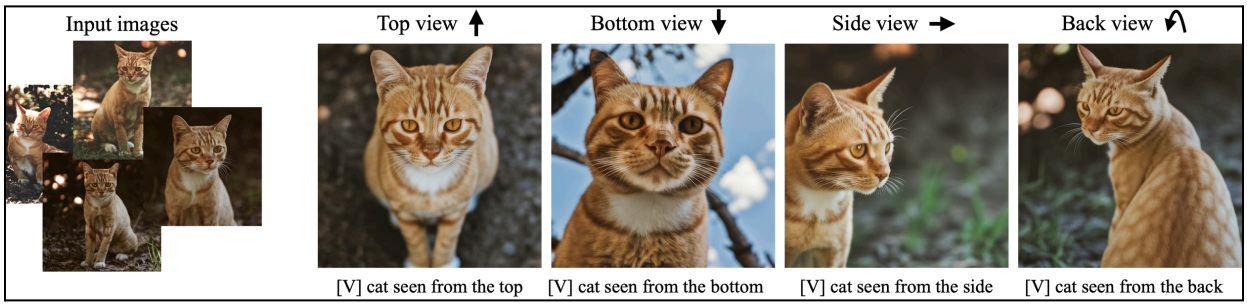
Applications of DreamBooth

Recontextualization: Generating novel images of specific subjects in varied contexts with realistic scene integration.

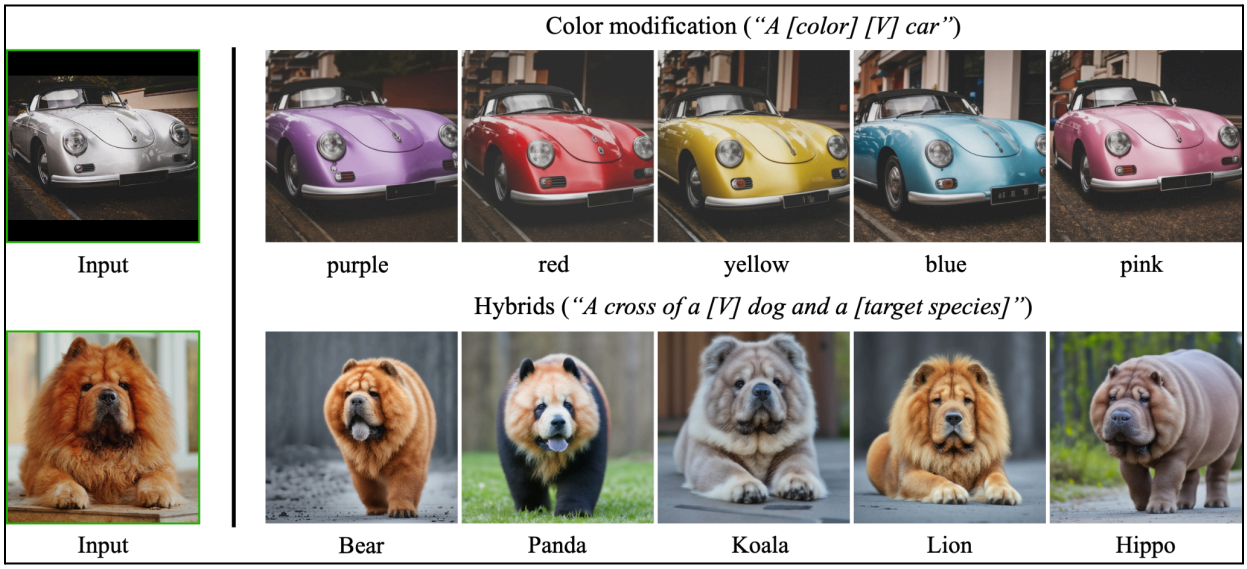
Art Renditions: Creating artistic interpretations of subjects in the styles of famous painters or sculptors, offering meaningful variations while preserving subject identity.



Novel View Synthesis: Rendering subjects from new perspectives, extrapolating from limited views to generate images from unobserved angles.



Property Modification: Altering subject properties such as color or species, blending unique features with new characteristics in a contextually appropriate manner.



Accessorization: Outfitting subjects with a variety of accessories tailored to specific scenarios, preserving the subject's core identity.

DreamBooth now allows us to maintain a subject’s likeness during image generation across multiple applications. Using these tweaks of the Text-to-Image training method, we can apply this with existing LoRA techniques for subject permanence across scenes during generation!

Additional Methods & Ideas

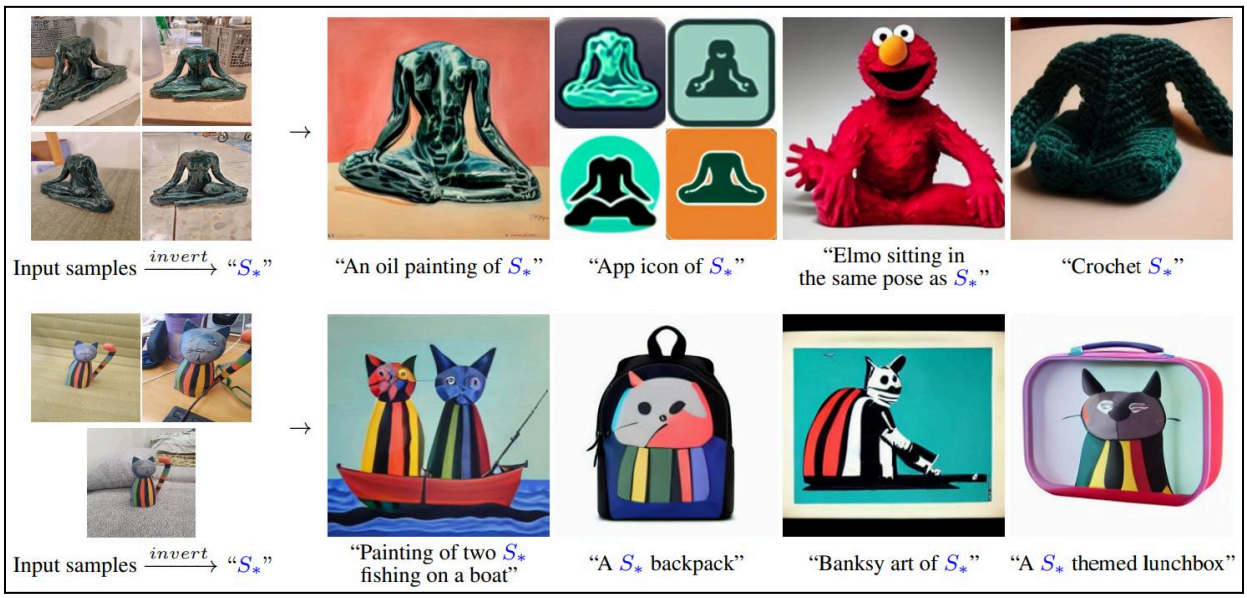
Many other different training methods and trained models have been developed for image generation models, here is a non-exhaustive list with some brief descriptions and resources to look further into.

Unconditional Image Generation

Description: Training models to generate images that are representative of the distribution seen in the training data, without being steered by any specific textual or image-based conditions.

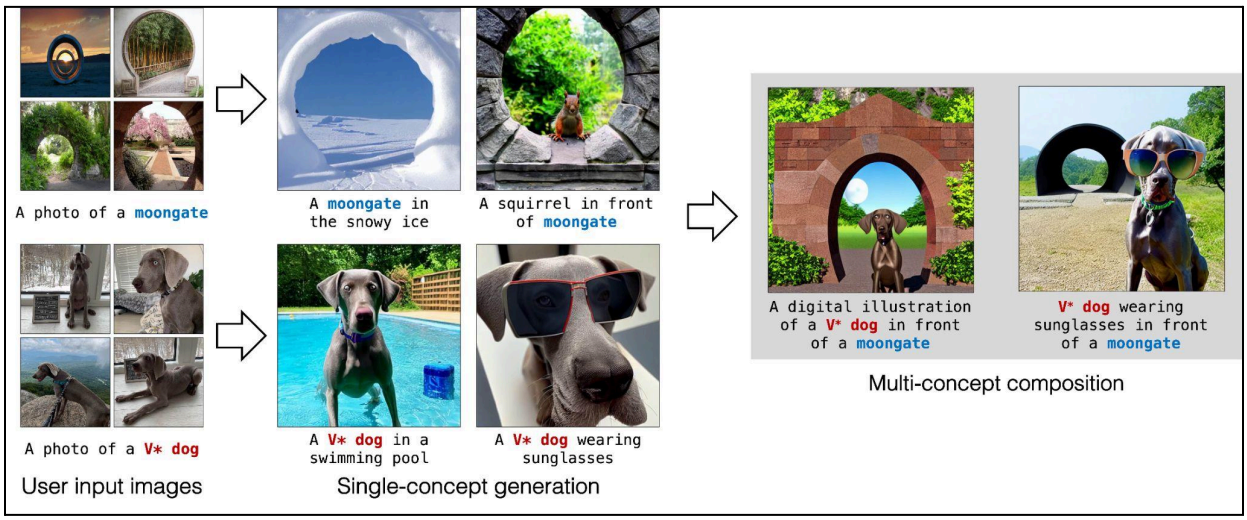
Textual Inversion

Description: Technique to personalize and refine the process of creating specific visual concepts, such as unique objects or artistic styles. It operates by introducing new "words" into the embedding space of a pre-trained text-to-image model. These words are learned from a small set of images (typically 3-5) that depict the concept the user wants to generate. Once learned, these new embeddings function like any other word in the model's vocabulary, allowing them to be used in sentences to guide the generation of images that feature the learned concepts, similar to the DreamBooth method.



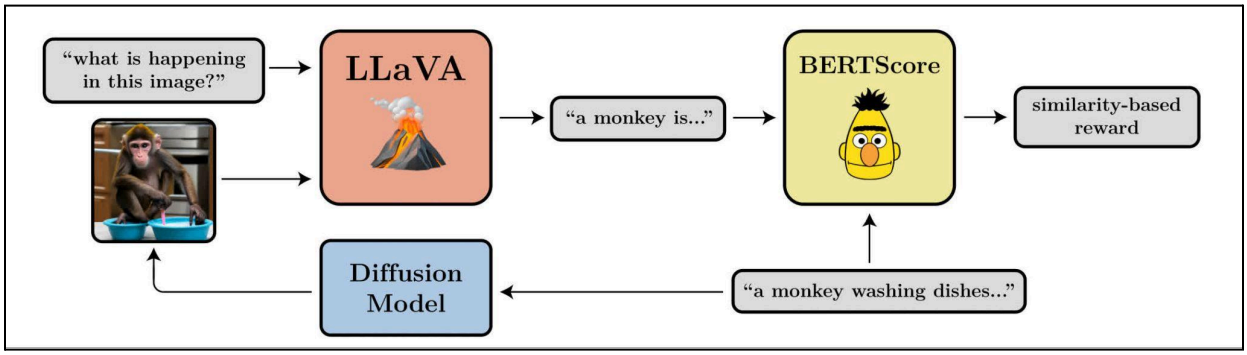
Custom Diffusion

Description: Method that enables diffusion models to quickly adapt to and synthesize new user-defined concepts like personal items, pets, or family members. It utilizes a retrieval system to gather real images with captions closely matching the new concepts provided by the user. This collection forms the basis of a specialized training dataset for fine-tuning the model. A modifier token is prefixed to general category names to denote personal concepts during the training process.



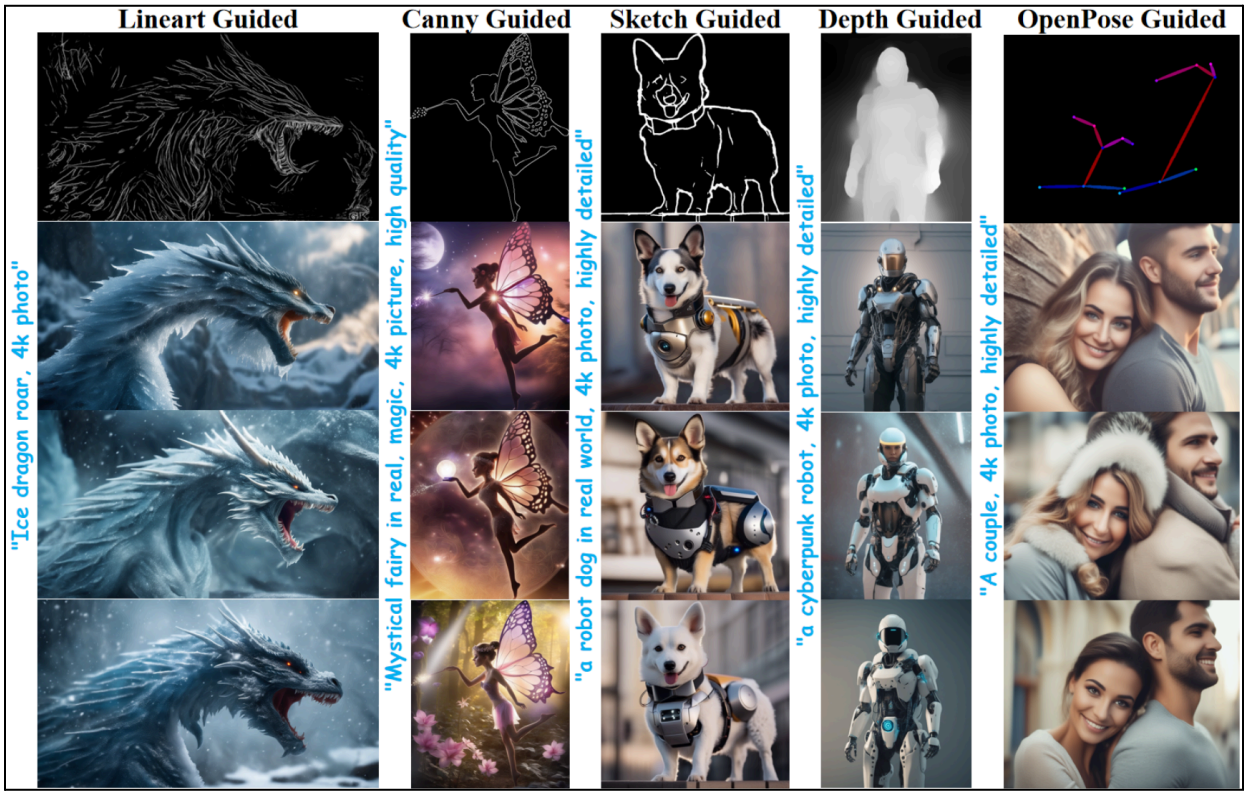
RL with Denoising Diffusion Policy Optimization (DDPO)

Description: Denoising Diffusion Policy Optimization (DDPO) treats the denoising process in diffusion models as a multi-step decision-making problem. DDPO employs policy gradient algorithms, demonstrating greater efficacy than conventional reward-weighted likelihood methods. The application of DDPO allows for the refinement of text-to-image diffusion models to meet specific objectives like enhancing image compressibility and improving image aesthetics based on human feedback, without the direct reliance on explicit prompting methods. Moreover, DDPO can utilize feedback from vision-language models, circumventing the need for extensive data collection or manual human annotation.



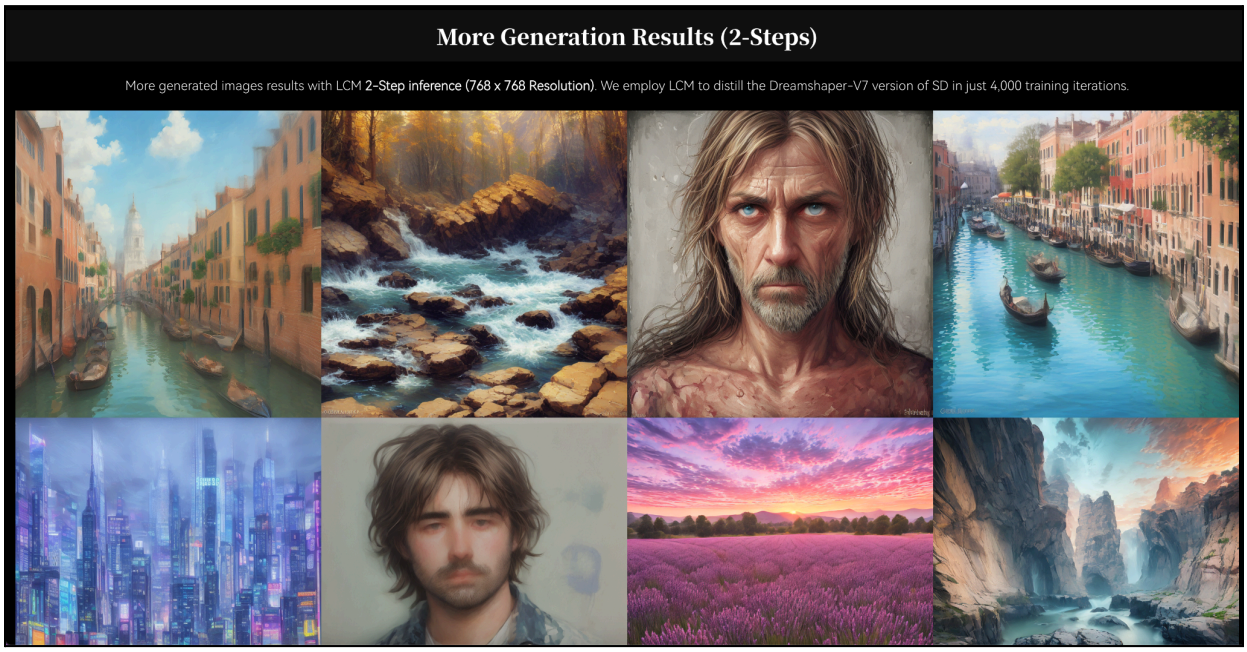
T2I-Adapters

Description: Method allowing for more precise adjustments in attributes like color and structure. It introduces the concept of T2I-Adapters, which are simple, lightweight modules that learn to align the model's internal knowledge with external control signals. These adapters enable fine-grained control over the image generation process by allowing the original Text-to-Image models to remain unchanged while various adapters are trained to cater to different control conditions. This achieves a more tailored and controlled generation of images.



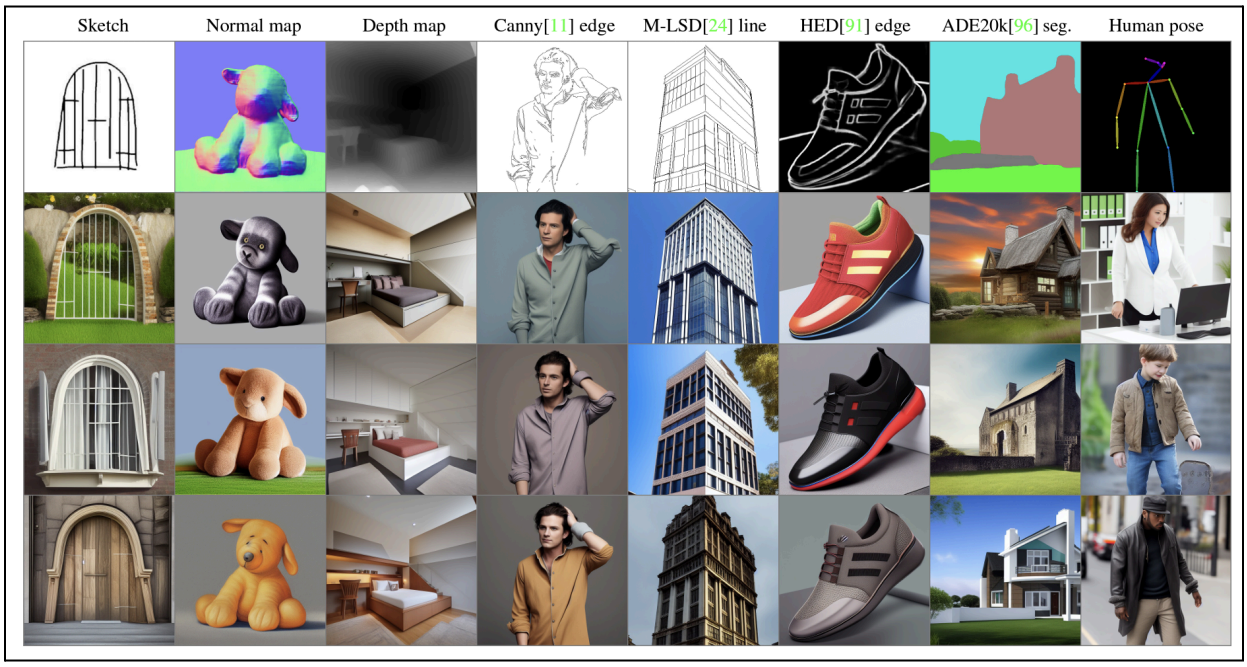
Latent Consistency Distillation

Description: Latent Consistency Models increase the efficiency of Latent Diffusion Model by streamlining the process of generating high-resolution images through minimizing the sampling steps. This approach enables the models to directly predict outcomes in the latent space, allowing a 768x768 image to be synthesized in just 2 to 4 steps.



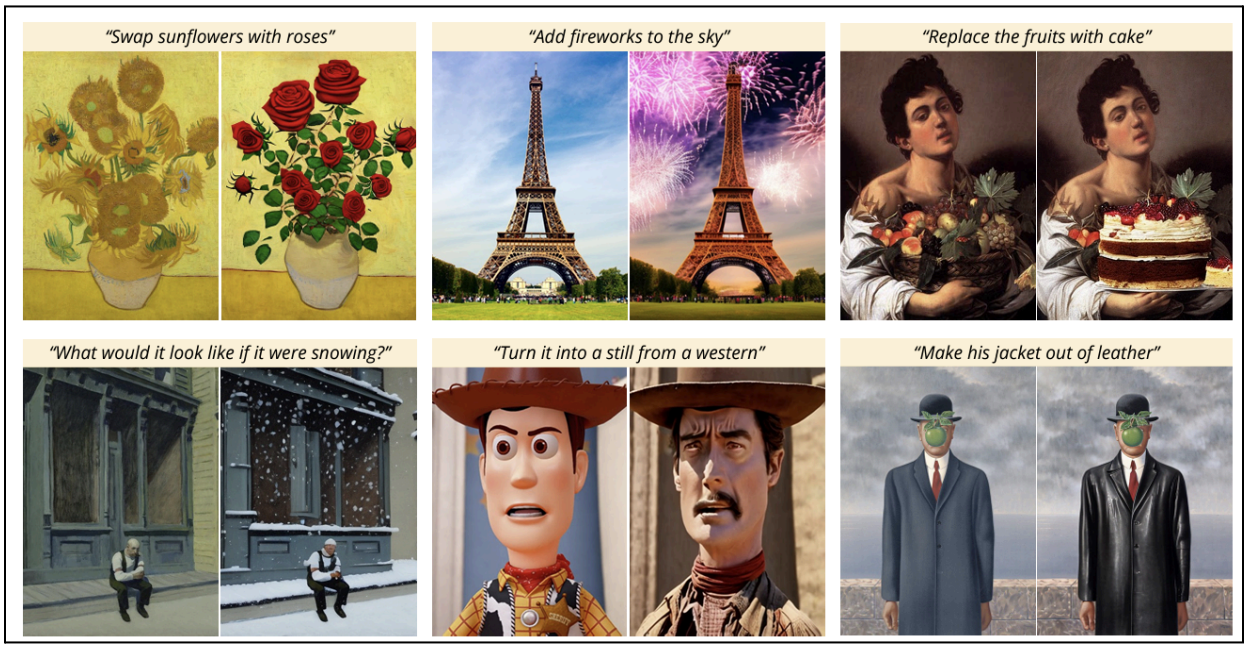
ControlNet

Description: This architecture allows for the application of various conditioning controls, such as edges, depth, segmentation, and human poses, which can be used singly or in combination, with or without accompanying text prompts.



InstructPix2Pix

Description: Method for editing images based on human instructions. The model operates by receiving an input image alongside written instructions that detail the desired edits. Leveraging the capabilities of two large pretrained models—a language model (GPT-3) and a text-to-image model (Stable Diffusion)—InstructPix2Pix uses these resources to generate a substantial dataset of image-editing examples for training. This training allows the model to generalize effectively to new, real images and user-provided instructions during inference. Unlike many other image editing models, InstructPix2Pix executes edits in a single forward pass without the need for per-example fine-tuning or inversion, resulting in rapid processing times.



Many other approaches and concepts are out there, but these are all documented well for individuals using the HuggingFace Diffusers package, which we will dig into next!

HuggingFace Diffusers Package

The HuggingFace Diffusers package makes it super easy to interface with and experiment with diffusion models (Github).

The library has three main components:

1. **Pipelines**: A simple way to run state-of-the-art diffusion models in inference by bundling all of the necessary components (multiple independently-trained models, schedulers, and processors) into a single end-to-end class. Pipelines are flexible and they can be adapted to use different schedulers or even model components.
2. **Noise Schedulers**: Interchangeable noise schedulers for balancing trade-offs between generation speed and quality.
    a. A scheduler takes a model's output (the sample which the diffusion process is iterating on) and a timestep to return a denoised sample. The timestep is important because it dictates where in the diffusion process the step is; data is generated by iterating forward n timesteps and inference occurs by propagating backward through the timesteps. Based on the timestep, a scheduler may be

discrete in which case the timestep is an int or continuous in which case the timestep is a float.

3. **Pretrained Models**: Pretrained models that can be used as building blocks, and combined with schedulers, for creating your own end-to-end diffusion systems.

These all allow for great flexibility over building, optimizing, and inferencing diffusion models of all modalities, both automatically with bundled pipelines, or piece by piece in a building block style.

They also provide great tools, and premade scripts, for training diffusion models, as we have discussed in the prior informational sections.

Below, we will break down the setup and execution of two premade scripts, Text-to-Image LoRA Training, and DreamBooth LoRA training for SDXL Base 1.0.

Text-To-Image SDXL LoRA Training Script Arguments

Diffusers Text-to-Image Training Documentation

README

Direct Link to Script in GitHub

Script Argument	Conditions
--pretrained_model_name_or_path	**Value Type**: String **Default Value**: None **Required**: True **Description**: Path to pretrained model or model identifier from huggingface.co/models.
--pretrained_vae_model_name_or_path	**Value Type**: String **Default Value**: None **Description**: Path to pretrained VAE model with better numerical stability
--revision	**Value Type**: String **Default Value**: None **Required**: False

Script Argument	Conditions
	Description: Revision of pretrained model identifier from huggingface.co/models.
--variant	**Value Type**: String **Default Value**: None **Description**: Variant of the model files of the pretrained model identifier from huggingface.co/models, 'e.g.' fp16
--dataset_name	**Value Type**: String **Default Value**: None **Description**: The name of the Dataset (from the HuggingFace hub) to train on (could be your own, possibly private, dataset). It can also be a path pointing to a local copy of a dataset in your filesystem, or to a folder containing files that 🤗 Datasets can understand.
--dataset_config_name	**Value Type**: String **Default Value**: None **Description**: The config of the Dataset, leave as None if there's only one config.
--train_data_dir	**Value Type**: String **Default Value**: None **Description**: A folder containing the training data. Folder contents must follow the structure described in https://huggingface.co/docs/datasets/image_dataset#imagefolder. In particular, a `metadata.jsonl` file must exist to provide the captions for the images. Ignored if `dataset_name` is specified.
--image_column	**Value Type**: String **Default Value**: image **Description**: The column of the dataset containing an image, as stated in the metadata.
--caption_column	**Value Type**: String **Default Value**: text **Description**: The column of the dataset containing a caption or a list of captions, as stated in the metadata.
--validation_prompt	**Value Type**: String **Default Value**: None

Script Argument	Conditions
	Description: A prompt that is used during validation to verify that the model is learning.
--num_validation_images	**Value Type**: Integer **Default Value**: 4 **Description**: Number of images that should be generated during validation with `validation_prompt`.
--validation_epochs	**Value Type**: Integer **Default Value**: 1 **Description**: Run fine-tuning validation every X epochs. The validation process consists of running the prompt `validation_prompt` multiple times: `num_validation_images`
--max_train_samples	**Value Type**: Integer **Default Value**: None **Description**: For debugging purposes or quicker training, truncate the number of training examples to this value if set.
--output_dir	**Value Type**: String **Default Value**: sd-model-finetuned-lora **Description**: The output directory where the model predictions and checkpoints will be written in your filesystem.
--cache_dir	**Value Type**: String **Default Value**: None **Description**: The directory where the downloaded models and datasets will be stored.
--seed	**Value Type**: Integer **Default Value**: None **Description**: A seed for reproducible training.
--resolution	**Value Type**: Integer **Default Value**: 1024 **Description**: The resolution for input images in pixels, all the images in the train/validation dataset will be resized to this resolution
--center_crop	**Default Value**: False

Script Argument	Conditions
	Action: If present in script gets set to True. **Description**: Whether to center crop the input images to the resolution. If not set, the images will be randomly cropped. The images will be resized to the resolution first before cropping.
--random_flip	**Action**: If present in script gets set to True. **Description**: whether to randomly flip images horizontally
--train_text_encoder	**Action**: If present in script gets set to True. **Description**: Whether to train the text encoder as well. If set, the text encoder should be float32 precision.
--train_batch_size	**Value Type**: Integer **Default Value**: 16 **Description**: Batch size (per device) for the training dataloader. Specifies the number of samples processed together on each device during training. A larger batch size can speed up training but requires more memory
--num_train_epochs	**Value Type**: Integer **Default Value**: 100 **Description**: Sets the total number of training cycles through the entire dataset. 1 Epoch means the model has seen every training example once.
--max_train_steps	**Value Type**: Integer **Default Value**: None **Description**: Total number of training steps to perform. If provided, overrides num_train_epochs. One step is one example (or batch) being processed
--checkpointing_steps	**Value Type**: Integer **Default Value**: 500 **Description**: Save a checkpoint of the training state every X updates. These checkpoints can be used both as final checkpoints in case they are better than the last checkpoint, and are also suitable for resuming training using `--resume_from_checkpoint`.
--checkpoints_total_limit	**Value Type**: Integer **Default Value**: None **Description**: Max number of checkpoints to store.

Script Argument	Conditions
--resume_from_checkpoint	**Value Type**: String **Default Value**: None **Description**: Whether training should be resumed from a previous checkpoint. Use a path saved by' `--checkpointing_steps`, or `latest` to automatically select the last available checkpoint.'
--gradient_accumulation_steps	**Value Type**: Integer **Default Value**: 1 **Description**: Number of updates steps to accumulate before performing a backward/update pass. Defines how many forward passes to accumulate before computing backward updates, reducing memory usage at the cost of slower updates.
--gradient_checkpointing	**Action**: If present in script gets set to True. **Description**: Whether or not to use gradient checkpointing to save memory at the expense of slower backward pass. Gradient checkpointing is a technique that trades computation for lower memory usage by storing a subset of intermediate activations and recomputing others during the backward pass.
--learning_rate	**Value Type**: Float **Default Value**: 1e-4, **Description**: Initial learning rate (after the potential warmup period) to use. Determines the step size at which the model updates its weights in response to the error it observes, affecting how quickly or slowly it learns.
--scale_lr	**Action**: If present in script gets set to True. **Default Value**: False **Description**: Scale the learning rate by the number of GPUs, gradient accumulation steps, and batch size.
--lr_scheduler	**Value Type**: String **Default Value**: constant **Choices**: linear, cosine, cosine_with_restarts polynomial, constant, constant_with_warmup **Description**: The scheduler type to use. - **Linear**: Gradually decreases the learning rate from the initial value to zero in a straight line.

Script Argument	Conditions
	- **Cosine**: Adjusts the learning rate following a cosine curve, smoothly decreasing to zero. - **Cosine with Restarts**: Applies a cosine learning rate decay but restarts the rate at intervals, potentially leading to better convergence in some cases. - **Polynomial**: Decreases the learning rate according to a polynomial power, generally sharper than linear. - **Constant**: Maintains a fixed learning rate throughout training, useful for fine-tuning when smaller adjustments are preferred. - **Constant with Warmup**: Starts with a lower learning rate and warms up to a constant value, helping to stabilize early training stages.
--lr_warmup_steps	**Value Type**: Integer **Default Value**: 500 **Description**: Number of steps for the warmup in the lr scheduler. Specifies the number of initial training steps during which the learning rate increases gradually, easing the model into learning to prevent early large gradient updates.
--snr_gamma	**Value Type**: Float **Default Value**: None Description: SNR weighting gamma to be used if rebalancing the loss. Recommended value is 5.0. More details here: https://arxiv.org/abs/2303.09556. Signal-to-Noise Ratio (SNR) measures the level of desired signal to the background noise, helping in assessing the clarity and usability of the data in model training.
--allow_tf32	**Action**: If present in script gets set to True. **Description**: Whether or not to allow TF32 on Ampere GPUs. Can be used to speed up training. For more information, see https://pytorch.org/docs/stable/notes/cuda.html#tensorfloat-32-tf32-on-ampere-devices, can accelerate training by reducing precision but maintaining suitable model accuracy.
--dataloader_num_workers	**Value Type**: Integer **Default Value**: 0 **Description**: Number of subprocesses to use for data loading. 0 means that the data will be loaded in the main process. Dataloader workers are subprocesses used to

Script Argument	Conditions
	load and preprocess data in parallel, reducing I/O wait times and speeding up input feeding to the training process.
--use_8bit_adam	**Action**: If present in script gets set to True. **Description**: Whether or not to use 8-bit Adam from bitsandbytes. More on the Adam optimizer here.
--adam_beta1	**Value Type**: Float **Default Value**: 0.9 **Description**: The beta1 parameter for the Adam optimizer. The beta1 parameter for the Adam optimizer controls the exponential decay rate for the first moment estimates, helping to adjust how quickly the optimizer forgets past gradients.
--adam_beta2	**Value Type**: Float **Default Value**: 0.999 **Description**: The beta2 parameter for the Adam optimizer. The beta2 parameter for the Adam optimizer controls the exponential decay rate for the second moment estimates, which influences the adjustment of the learning rate based on recent gradient variances.
--adam_weight_decay	**Value Type**: Float **Default Value**: 1e-2 **Description**: Weight decay to use. Weight decay is a regularization technique used in the Adam optimizer to prevent overfitting by penalizing large weights during training.
--adam_epsilon	**Value Type**: Float **Default Value**: 1e-08 **Description**: Epsilon value for the Adam optimizer. Epsilon is a small value added to the denominator in the Adam optimizer's update rule to prevent division by zero, ensuring numerical stability.
--max_grad_norm	**Value Type**: Float **Default Value**: 1.0 **Description**: Max gradient norm. Max gradient norm is a parameter used in gradient clipping, which limits the size of

Script Argument	Conditions
	gradients to a defined threshold during backpropagation, aiding in stabilizing and improving the training of neural networks.
--push_to_hub	**Action**: If present in script gets set to True. **Description**: Whether or not to push the model to the HuggingFace Hub.
--hub_token	**Value Type**: String **Default Value**: None **Description**: The token to use to push to the Model Hub. Can be made or found in your account settings here.
--prediction_type	**Value Type**: String **Default Value**: None **Choices**: 'epsilon' or 'v_prediction' or leave `None`. If left to `None` the default prediction type of the scheduler: `noise_scheduler.config.prediction_type` is chosen. **Description**: The prediction_type that shall be used for training. - **Epsilon**: Utilizes a margin of error (epsilon) in predictions, typically to maintain a buffer that accounts for noise and variability in data. - **V-Prediction**: Involves predicting a future state or value based on a series of past values, often using more complex statistical or machine learning models to extrapolate trends or patterns.
--hub_model_id	**Value Type**: String **Default Value**: None **Description**: The name of the repository to keep in sync with the local `output_dir`, becomes the name of the model on HuggingFace.
--logging_dir	**Value Type**: String **Default Value**: logs **Description**: TensorBoard log directory. Will default to output_dir/runs/CURRENT_DATETIME_HOSTNAME.
--report_to	**Value Type**: String **Default Value**: tensorboard **Choices**: tensorboard, wandb, comet_ml, all

Script Argument	Conditions
	Description: 'The integration to report the results and logs to. Supported platforms are tensorboard, wandb, comet_ml. Use `all` to report to all integrations.'
--mixed_precision	**Value Type**: String **Default Value**: None **Choices**: no, fp16, bf16 **Description**: Whether to use mixed precision. PyTorch >= 1.10.and an Nvidia Ampere GPU. Default to the value of accelerate config of the current system or the flag passed with the `accelerate.launch` command. Use this argument to override the accelerate config. Mixed precision training involves using a mix of different numerical precisions (floating-point formats) during model training to optimize memory usage and computational speed without significantly affecting model accuracy. - **No**: Disables mixed precision, using the standard precision set by the training environment. - **FP16**: Uses 16-bit floating point precision, reducing memory consumption and potentially speeding up training on compatible hardware. - **BF16 (Bfloat16)**: Also a 16-bit format but retains more precision in the exponent part compared to FP16, beneficial for training deeper models; requires newer hardware and specific software support.
--local_rank	**Value Type**: Integer **Default Value**: -1 **Description**: For distributed training, local rank specifies the unique identifier of a process within a single machine when performing distributed training, helping to allocate resources and manage tasks across multiple GPUs or nodes.
--enable_xformers_memory_efficient_attention	**Action**: If present in script gets set to True. **Description**: Enables the use of memory-efficient attention mechanisms from the xformers library, which are designed to reduce memory consumption during training of transformer models, allowing for larger models or longer sequences.
--enable_npu_flash_attention	**Action**: If present in script gets set to True. **Description**: Enables the use of NPU Flash Attention, which optimizes attention mechanisms for Neural

Script Argument	Conditions
	Processing Units (NPUs) to enhance performance and efficiency in processing large-scale transformer models.
--noise_offset	**Value Type**: Float **Default Value**: 0 **Description**: Noise offset adjusts the scale of noise introduced into the training process, which can be used to enhance model robustness or to simulate variations in input data.
--rank	**Value Type**: Integer **Default Value**: 4 **Description**: Rank specifies the dimensionality of the low-rank matrices used in LoRA (Low-Rank Adaptation) updates, influencing the complexity and the capacity of the model adjustments during training.
--debug_loss	**Action**: If present in script gets set to True. **Description**: debug loss for each image, if filenames are available in the dataset

---

DreamBooth SDXL LoRA Script Arguments & Differences

Diffusers DreamBooth Training Documentation

README

Direct Link to Script in GitHub

NOTE: The following tables show the differences between this script and the prior. Existing arguments from the above Text-To-Image script remain true, along with these new arguments, unless specified removed in the Removed Arguments table.

New Arguments	
Script Argument	**Conditions**
--instance_data_dir	**Value Type**: String **Default Value**: None **Description**: A folder in the filesystem containing the training data.
--repeats	**Value Type**: Integer **Default Value**: 1 **Description**: How many times to repeat the training data. Specifies the number of times the training dataset is repeated during a single epoch, effectively increasing the amount of training without adding new data, to promote better model generalization.
--class_data_dir	**Value Type**: String **Default Value**: None **Required**: False **Description**: A folder in the filesystem containing the training data of class images.
--instance_prompt	**Value Type**: String **Default Value**: None **Required**: True **Description**: The prompt with identifier specifying the instance, e.g. 'photo of a **TOK** dog', 'in the style of **TOK**'
--class_prompt	**Value Type**: String **Default Value**: None **Description**: The prompt to specify images in the same class as provided instance images. Defines the textual prompt used to generate or specify images belonging to the same class as the provided instance images, aiding in prior preservation loss computation.
--do_edm_style_training	**Default Value**: False **Action**: If present in script gets set to True. **Description**: Flag to conduct training using the EDM formulation as introduced in https://arxiv.org/abs/2206.00364. Activates training using the EDM (Elucidating Diffusion Models) formulation, which enhances model performance by incorporating noise

New Arguments	
	scheduling and denoising objectives directly into the training process, as detailed in the specified research paper.
--with_prior_preservation	**Default Value**: False **Action**: If present in script gets set to True. **Description**: Enables the addition of prior preservation loss during training, which helps maintain the model's ability to generate outputs consistent with the original data distribution. This technique ensures that the model does not forget previously learned information while learning new patterns, improving its overall stability and performance.
--prior_loss_weight	**Value Type**: Float **Default Value**: 1.0 **Description**: The weight of prior preservation loss. Prior loss weight determines the influence of the prior preservation loss in the overall training objective, balancing the importance of maintaining previously learned information against learning new data patterns.
--num_class_images	**Value Type**: Integer **Default Value**: 100 **Description**: Specifies the minimum number of class images required for computing prior preservation loss. If the class_data_dir lacks sufficient images, additional images will be generated based on the class_prompt to meet the required count.
--output_kohya_format	**Action**: If present in script gets set to True. **Description**: Flag to additionally generate final state dict in the Kohya format so that it becomes compatible with A111, Comfy, Kohya, and other User Interface integrations.
--sample_batch_size	**Value Type**: Integer **Default Value**: 4 **Description**: Batch size (per device) for sampling images. Specifies the number of images to be processed simultaneously in each batch during the sampling phase. A larger batch size can speed up sampling but requires

New Arguments	
	more memory, with each device handling one batch at a time.
--text_encoder_lr	**Value Type**: Float **Default Value**: 5e-6 **Description**: Text encoder learning rate to use.
--lr_num_cycles	**Value Type**: Integer **Default Value**: 1 **Description**: Sets the number of times the learning rate is reset to its initial value in the cosine_with_restarts scheduler, allowing the model to escape local minima and potentially improve training convergence.
--lr_power	**Value Type**: Float **Default Value**: 1.0 **Description**: Power factor of the polynomial scheduler. Defines the power factor for the polynomial learning rate scheduler, controlling the rate of decay for the learning rate. A higher power results in a steeper decay.
--optimizer	**Value Type**: String **Default Value**: AdamW **Choices**: AdamW, prodigy **Description**: The optimizer type to use. - **AdamW**: An adaptive learning rate optimization algorithm that includes weight decay, improving generalization by preventing overfitting. - **Prodigy**: A state-of-the-art optimizer designed for efficient training, potentially offering improved convergence and performance over traditional optimizers.
--prodigy_beta3	**Value Type**: Float **Default Value**: None **Description**: coefficients for computing the Prodigy stepsize using running averages. If set to None uses the value of square root of beta2. Ignored if optimizer is adamW. Beta3 is a coefficient in the Prodigy optimizer that influences the calculation of the adaptive step size using running averages of past gradients and their squares.
--prodigy_decouple	**Value Type**: Boolean

New Arguments	
	Default Value: True **Description**: Indicates whether to use AdamW style decoupled weight decay in the Prodigy optimizer. Decoupled weight decay separates the weight decay term from the gradient update, leading to more effective regularization and better model performance.
--prodigy_use_bias_correction	**Value Type**: Boolean **Default Value**: True **Description**: Turn on Adam's bias correction. True by default. Ignored if optimizer is adamW. Adam's bias correction adjusts the estimates of the first (mean) and second (variance) moments of gradients to account for their initialization biases, particularly during the initial training steps.
--prodigy_safeguard_warmup	**Value Type**: Boolean **Default Value**: True **Description**: Remove learning rate from the denominator of D estimate to avoid low learning rate related instability issues during warm-up stage. True by default. Ignored if optimizer is adamW.
--prior_generation_precision	**Value Type**: String **Default Value**: None **Choices**: no, fp32, fp16, bf16 **Description**: Choose prior generation precision between fp32, fp16 and bf16 (bfloat16). Bf16 requires PyTorch 1.10.and an Nvidia Ampere GPU. Default to fp16 if a GPU is available else fp32. Prior generation precision refers to the numerical format used for calculations when generating prior distributions or initial conditions during the training process. - **No**: Disables prior generation precision adjustments. - **FP32**: Uses 32-bit floating point precision, providing the highest accuracy at the cost of more memory and computational resources. - **FP16**: Uses 16-bit floating point precision, reducing memory usage and potentially speeding up computation on compatible hardware. - **BF16 (Bfloat16)**: Uses 16-bit precision with a larger exponent range, beneficial for deep learning tasks; requires specific hardware and software support.

Removed Arguments	
--train_data_dir	**Value Type**: String **Default Value**: None **Description**: A folder containing the training data. Folder contents must follow the structure described in https://huggingface.co/docs/datasets/image_dataset#imagefolder. In particular, a `metadata.jsonl` file must exist to provide the captions for the images. Ignored if `dataset_name` is specified.
--max_train_samples	**Value Type**: Integer **Default Value**: None **Description**: For debugging purposes or quicker training, truncate the number of training examples to this value if set.
--prediction_type	**Value Type**: String **Default Value**: None **Choices**: 'epsilon' or 'v_prediction' or leave `None`. If left to `None` the default prediction type of the scheduler: `noise_scheduler.config.prediction_type` is chosen. **Description**: The prediction_type that shall be used for training. - **Epsilon**: Utilizes a margin of error (epsilon) in predictions, typically to maintain a buffer that accounts for noise and variability in data. - **V-Prediction**: Involves predicting a future state or value based on a series of past values, often using more complex statistical or machine learning models to extrapolate trends or patterns.
--noise_offset	**Value Type**: Float **Default Value**: 0 **Description**: Noise offset adjusts the scale of noise introduced into the training process, which can be used to enhance model robustness or to simulate variations in input data.
--debug_loss	**Action**: If present in script gets set to True. **Description**: debug loss for each image, if filenames are available in the dataset

Script Examples

Text-to-Image LoRA SDXL

```
accelerate launch train_text_to_image_lora_sdxl.py \
  --pretrained_model_name_or_path="stabilityai/stable-diffusion-xl-base-1.0" \
  --pretrained_vae_model_name_or_path="madebyollin/sdxl-vae-fp16-fix" \
  --dataset_name="AdamLucek/oldbookillustrations-small" \
  --validation_prompt="An inventor tinkers with a complex machine in his workshop, oblivious to the setting sun outside" \
  --num_validation_images=4 \
  --validation_epochs=1 \
  --output_dir="output/sdxl-base-1.0-oldbookillustrations-lora" \
  --resolution=1024 \
  --center_crop \
  --random_flip \
  --train_text_encoder \
  --train_batch_size=1 \
  --num_train_epochs=10 \
  --checkpointing_steps=500 \
  --gradient_accumulation_steps=4 \
  --learning_rate=1e-04 \
  --lr_warmup_steps=0 \
  --report_to="wandb" \
  --dataloader_num_workers=8 \
  --allow_tf32 \
  --mixed_precision="fp16" \
  --push_to_hub \
  --hub_model_id="sdxl-base-1.0-oldbookillustrations-lora"
```

Breakdown of Arguments:

--pretrained_model_name_or_path=stabilityai/stable-diffusion-xl-base-1.0

Uses the Stable Diffusion XL Base 1.0 model from Stability AI as the starting point for training. This pretrained model provides a solid foundation, leveraging previously learned features to improve training efficiency and outcomes.

--pretrained_vae_model_name_or_path="madebyollin/sdxl-vae-fp16-fix"

Uses a specific VAE (Variational Autoencoder) model for image encoding/decoding, which helps in improving image quality and reducing artifacts during training, especially with mixed precision (fp16).

--dataset_name="AdamLucek/oldbookillustrations-small"
Specifies a dataset of old book illustrations for training. This dataset defines the visual style and content that the model will learn to generate.

--validation_prompt="An inventor tinkers with a complex machine in his workshop, oblivious to the setting sun outside"
Sets a specific prompt for generating validation images. This helps in evaluating how well the model can generate images based on the given textual description at various stages of training.

--num_validation_images=4
Generates 4 images during each validation phase. This provides a small sample to assess the model's performance without significantly impacting training time.

--validation_epochs=1
Runs the validation process after every epoch. This frequent validation helps in monitoring the model's progress and identifying potential issues early.

--output_dir="output/sdxl-base-1.0-oldbookillustrations-lora"
Specifies the directory where all training outputs, including models and logs, will be saved. This keeps training results organized and easily accessible.

--resolution=1024
Sets the resolution of the training and validation images to 1024x1024 pixels. Higher resolution improves image detail but requires more computational resources.

--center_crop
Crops images to the center before training, ensuring uniformity in image size and focusing on the central part of the images, which is often the most relevant.

--random_flip
Applies random horizontal flips to training images, augmenting the dataset and helping the model generalize better by learning from more varied image orientations.

--train_text_encoder
Enables training of the text encoder along with the image generator. This improves the model's ability to understand and generate images based on textual descriptions.

--train_batch_size=1
Processes one image per batch per device during training. Smaller batch sizes can reduce memory usage but may slow down the training process.

--num_train_epochs=10
Trains the model for 10 epochs, meaning the entire dataset will be passed through the model 10 times. More epochs can improve learning but require more time.

--checkpointing_steps=500

Saves a checkpoint of the model every 500 steps. This allows for saving intermediate states, making it possible to resume training or rollback if needed.

--gradient_accumulation_steps=4

Accumulates gradients over 4 steps before performing a backward pass. This effectively simulates a larger batch size, helping to stabilize training without requiring more memory.

--learning_rate=1e-04

Sets the initial learning rate to 0.0001. The learning rate controls how much to adjust the model's weights with respect to the loss gradient. A smaller learning rate can lead to more stable but slower convergence.

--lr_warmup_steps=0

No warm-up period for the learning rate. Warm-up steps can help prevent large updates early in training, which can destabilize the model.

--report_to="wandb"

Uses Weights & Biases for logging and monitoring the training process. This provides visualization tools and tracking capabilities for better experiment management.

--dataloader_num_workers=8

Uses 8 subprocesses for data loading. More workers can speed up data loading and preprocessing, reducing bottlenecks and improving training efficiency.

--allow_tf32

Enables TF32 computations on Ampere GPUs, potentially speeding up training by allowing lower precision calculations while maintaining acceptable accuracy.

--mixed_precision="fp16"

Uses 16-bit floating point precision for training, which reduces memory usage and can speed up training on compatible hardware without significantly affecting model accuracy.

--push_to_hub

Pushes the trained model to the Hugging Face Hub, making it accessible for sharing and deployment. This integrates the model into a broader ecosystem for collaboration.

--hub_model_id="sdxl-base-1.0-oldbookillustrations-lora"

Sets the identifier for the model on the Hugging Face Hub, organizing it under a specific name and version for easy access and management.

DreamBooth LoRA SDXL

```
accelerate launch train_dreambooth_lora_sdxl.py \
  --pretrained_model_name_or_path="stabilityai/stable-diffusion-xl-base-1.0"  \
```

```
--dataset_name="AdamLucek/green-chair" \
--pretrained_vae_model_name_or_path="madebyollin/sdxl-vae-fp16-fix" \
--output_dir="lora-trained-xl" \
--train_text_encoder \
--instance_prompt="a photo of sks chair" \
--resolution=1024 \
--train_batch_size=1 \
--gradient_accumulation_steps=4 \
--learning_rate=1e-4 \
--lr_scheduler="constant" \
--lr_warmup_steps=0 \
--max_train_steps=500 \
--validation_prompt="A photo of sks chair in an apartment" \
--validation_epochs=5 \
--seed="0" \
--hub_model_id="sdxl-base-1.0-greenchair-dreambooth-lora" \
--push_to_hub
```

Breakdown of Arguments:

--pretrained_model_name_or_path="stabilityai/stable-diffusion-xl-base-1.0"

Utilizes the Stable Diffusion XL Base 1.0 model from Stability AI as the starting point for training. This pretrained model offers a robust foundation, leveraging prior knowledge to enhance training efficiency and outcomes.

--dataset_name="AdamLucek/green-chair"

Specifies the dataset of green chair images for training. This dataset will determine the specific visual features and style that the model will learn to replicate.

--pretrained_vae_model_name_or_path="madebyollin/sdxl-vae-fp16-fix"

Uses a particular VAE (Variational Autoencoder) model for encoding and decoding images, which helps improve image quality and reduce artifacts, especially when using mixed precision (fp16).

--output_dir="lora-trained-xl"

Sets the directory where all training outputs, including models and logs, will be saved. This keeps the training results organized and easily accessible.

--train_text_encoder

Enables the training of the text encoder alongside the image generator. This improves the model's ability to understand and generate images based on textual descriptions.

--instance_prompt="a photo of sks chair"

Provides a specific text instance prompt to generate images with the prompt identifier 'a photo of sks chair' during training. This ensures the model focuses on generating the desired training results when the trigger phrase "a photo of sks chair" is inserted into the prompt.

--resolution=1024

Sets the image resolution to 1024x1024 pixels for training and validation. Higher resolution enhances image detail but requires more computational resources.

--train_batch_size=1

Processes one image per batch per device during training. Smaller batch sizes can reduce memory usage but may slow down the training process.

--gradient_accumulation_steps=4

Accumulates gradients over 4 steps before performing a backward pass. This effectively simulates a larger batch size, helping to stabilize training without requiring more memory.

--learning_rate=1e-4

Sets the initial learning rate to 0.0001. The learning rate controls the magnitude of updates to the model's weights. A smaller learning rate can lead to more stable but slower convergence.

--lr_scheduler="constant"

Uses a constant learning rate throughout the training process, which helps maintain consistent updates to the model's weights without adjustments.

--lr_warmup_steps=0

No warm-up period for the learning rate. Warm-up steps can help prevent large updates early in training, which can destabilize the model.

--max_train_steps=500

Limits the training process to a maximum of 500 steps. This cap helps to prevent overfitting and manage computational resources effectively.

--validation_prompt="A photo of sks chair in an apartment"

Sets a specific prompt for generating validation images. This helps in evaluating how well the model can generate images based on the given textual description and instance prompt at various stages of training.

--validation_epochs=5

Runs the validation process every 5 epochs. Frequent validation helps monitor the model's progress and identify potential issues early.

--seed="0"

Sets the random seed to 0 for reproducibility. This ensures that the training process can be replicated exactly, which is important for debugging and verifying results.

--hub_model_id="sdxl-base-1.0-greenchair-dreambooth-lora"

Assigns an identifier for the model on the Hugging Face Hub. This organizes the model under a specific name and version for easy access and management.

--push_to_hub

Pushes the trained model to the Hugging Face Hub, making it accessible for sharing and deployment. This integrates the model into a broader ecosystem for collaboration.

Running A Script Example

The Setup

Pick your favorite cloud computing service, and snag a GPU. Here's some popular spots in no particular order or endorsement (someone please sponsor me):

Easy for Individual Users:

Runpod - https://www.runpod.io/
Lambda Labs - https://lambdalabs.com/
Tensordock - https://www.tensordock.com/
Hyperstack - https://www.hyperstack.cloud/
Vast - https://vast.ai/

Little More Involved:

Coreweave - https://www.coreweave.com/
Paperspace - https://www.paperspace.com/
Google Cloud Platform - https://cloud.google.com/?hl=en
Amazon Web Service - https://aws.amazon.com/
Microsoft Azure - https://azure.microsoft.com/en-us

Nonexhaustive list, find the option that works for your budget and GPU requirements, for example the two scripts were trained on 1x A100 each, with 40GB of VRAM. You can skip validation inference runs to free up memory, and other parameters that can be added/modified for lesser systems are:

--use_8bit_adam	**Action**: If present in script gets set to True. **Description**: Whether or not to use 8-bit Adam from bitsandbytes. More on the Adam optimizer here.
--gradient_checkpointing	**Action**: If present in script gets set to True. **Description**: Whether or not to use gradient checkpointing to save memory at the expense of slower backward pass. Gradient checkpointing is a technique that trades computation for lower memory usage by storing a subset of intermediate activations and recomputing others during the backward pass.
--enable_xformers_memory_efficient_attention	**Action**: If present in script gets set to True.

	Description: Enables the use of memory-efficient attention mechanisms from the xformers library, which are designed to reduce memory consumption during training of transformer models, allowing for larger models or longer sequences.
--gradient_accumulation_steps	**Value Type**: Integer **Default Value**: 1 **Description**: Number of updates steps to accumulate before performing a backward/update pass. Defines how many forward passes to accumulate before computing backward updates, reducing memory usage at the cost of slower updates.
--mixed_precision	**Value Type**: String **Default Value**: None **Choices**: no, fp16, bf16 **Description**: Whether to use mixed precision. PyTorch >= 1.10.and an Nvidia Ampere GPU. Default to the value of accelerate config of the current system or the flag passed with the `accelerate.launch` command. Use this argument to override the accelerate config. Mixed precision training involves using a mix of different numerical precisions (floating-point formats) during model training to optimize memory usage and computational speed without significantly affecting model accuracy. - **No**: Disables mixed precision, using the standard precision set by the training environment. - **FP16**: Uses 16-bit floating point precision, reducing memory consumption and potentially speeding up training on compatible hardware. - **BF16 (Bfloat16)**: Also a 16-bit format but retains more precision in the exponent part compared to FP16, beneficial for training deeper models; requires newer hardware and specific software support.

Training Script Execution

Step 1:

Connect to your environment remotely and navigate to the command line interface

Step 2:

Download Required Packages

```
git clone https://github.com/huggingface/diffusers
cd diffusers
pip install -e .
```

```
cd examples/text_to_image
```

```
pip install -r requirements_sdxl.txt
```

As a note, sometimes your environment may need additional setup, however most of the GPU renting services are preset with basic deep learning packages. Generally an error will be displayed when attempting to run the script that will tell you if an additional package is missing, or a package is incompatible.

Step 3:

Log into HuggingFace, and Weights and Biases if using for logging/tracking

```
huggingface-cli login
```

And pass your HuggingFace Access Token

```
pip install wandb
wandb login
```

And pass your Weights and Biases API key

Step 4:

Setup your accelerate configuration.

```
accelerate config default
```

Step 5:

Run the script! Make sure to update the output directory and model ID if copying the script below to your own.

```
accelerate launch train_text_to_image_lora_sdxl.py \
  --pretrained_model_name_or_path="stabilityai/stable-diffusion-xl-base-1.0" \
  --pretrained_vae_model_name_or_path="madebyollin/sdxl-vae-fp16-fix" \
  --dataset_name="AdamLucek/oldbookillustrations-small" \
  --validation_prompt="An inventor tinkers with a complex machine in his workshop, oblivious to the setting sun outside" \
  --num_validation_images=4 \
  --validation_epochs=1 \
  --output_dir="INSERT OUTPUT DIRECTORY HERE" \
  --resolution=1024 \
  --center_crop \
```

```
--random_flip \
--train_text_encoder \
--train_batch_size=1 \
--num_train_epochs=10 \
--checkpointing_steps=500 \
--gradient_accumulation_steps=4 \
--learning_rate=1e-04 \
--lr_warmup_steps=0 \
--report_to="wandb" \
--dataloader_num_workers=8 \
--allow_tf32 \
--mixed_precision="fp16" \
--push_to_hub \
--hub_model_id="INSERT MODEL ID HERE"
```

As always, tuning your hyperparameters and figuring out what training method works best on your system or setup is nothing short of an art. Tinker around, see what error codes get thrown, monitor your training progress, and eventually you'll have a model training the way you want.

Step 5:

Load and use your model! The above training script took roughly 3 and a half hours to fully train on my dataset. In the next section, we have notebooks outlining the code used to load and run inference with the trained model.

Outcomes & Code Notebooks

Text-to-Image LoRA Results - Old Book Illustrations

SDXL LoRA OldBookIllustrations.ipynb

HuggingFace Hub:
AdamLucek/sdxl-base-1.0-oldbookillustrations-lora

Dataset:
AdamLucek/oldbookillustrations-small

Example Results:

“A dachshund walks confidently down a dirt path”

“A sailboat sailing the ocean”

“A diagram of a wolf skeleton”

“A carnival in the distance”

DreamBooth LoRA Results - Green Chair

SDXL LoRA DreamBooth GreenChair.ipynb

HuggingFace Hub:
AdamLucek/sdxl-base-1.0-greenchair-dreambooth-lora

Dataset:
AdamLucek/green-chair

Reference Image:

Example Results:

“A photo of sks chair in front of the great pyramids”

“A photo of sks chair in the middle of a new england fall field, with autumn leaves all around and pumpkins”

“A photo of sks chair in a deep ruby red color”

“A photo of sks chair in the middle of a snowy tundra with trees”
