Carleton University

COMP 4905 – Honours Project

TactileNet: Exploring Model Optimization and Dataset Expansion

Youssif Ashmawy

Supervisor: **Dr. Majid Komeili**

Date: August 04, 2025

Abstract

In this work, we expanded the TactileNet dataset (1,200 tactile images) to incorporate feedback text with the help of 132 benchmark-evaluated examples by human experts. We added structured annotations and incorporated them into a reward-guided training pipeline using Stable Diffusion fine-tuned with LoRA adapters trained on our dataset `https://huggingface.co/datasets/youssifashmawy/training_dataset`. Multiple loss functions and reward weightings were tested, along with extended training at reduced learning rates. Experiments showed that CLIP-heavy weighting improved contour clarity for some classes, while a balanced weighting better preserved structural fidelity in others. Longer training runs yielded small stability gains but also signs of overfitting, which degraded tactile readability. Despite these advances, dataset scarcity and the reliance on a frozen CLIP model as a reward signal limited performance, as CLIP did not evaluate tactile quality or guideline compliance. These results provided a foundation for future work on expanding datasets, applying preference-based reinforcement learning, and leveraging large language models to generate feedback-guided image edits toward fully guideline-compliant tactile graphics.

Acknowledgments

I want to take this opportunity to express my thanks to all those who supported me throughout my project journey. I would like to express my deepest gratitude to my supervisor, Dr. Majid Komeili, for his invaluable guidance, feedback, and support throughout the course of this project. I am also sincerely thankful to Adnan Khan, whose mentorship, technical insights, and constant encouragement were essential in navigating the challenges of this research.

Contents

List of Figures

List of Tables

Listings

Chapter 1

1.1 Motivation

Access to graphical information remains a significant barrier for individuals who are blind and visual impaired (BLV). Tactile graphics are crucial for conveying visual concepts such as diagrams, maps, and illustrations, but their production is time-consuming, costly, and dependent on specialist expertise. Devices like the Dot Pad (Figure 1.1), the first smart tactile graphics display, now allow users to instantly access tactile graphics from digital sources by converting any input on the connected device into touchable graphics [VisonAid, 2025]. While this technology opens up new possibilities, scaling the generation of tactile graphics still requires not only advanced generative models but also a robust, well-structured dataset that reflects tactile readability standards.

Figure 1.1: Braille PAD Tactile Tablet from [VisonAid, 2025].

1.2 Background

TactileNet [Khan et al., 2025] is a deep learning framework designed to generate tactile graphics from text and image prompts using Stable Diffusion (SD) fine-tuned with LoRA and DreamBooth. Earlier versions of TactileNet produced outputs that visually aligned with reference images but often failed to meet tactile readability requirements, including excessive detail and missing essential tactile features. A shift from multiple

category-specific LoRA adapters to a single unified adapter further reduced tactile quality, revealing that dataset limitations rather than model architecture were the primary bottleneck.

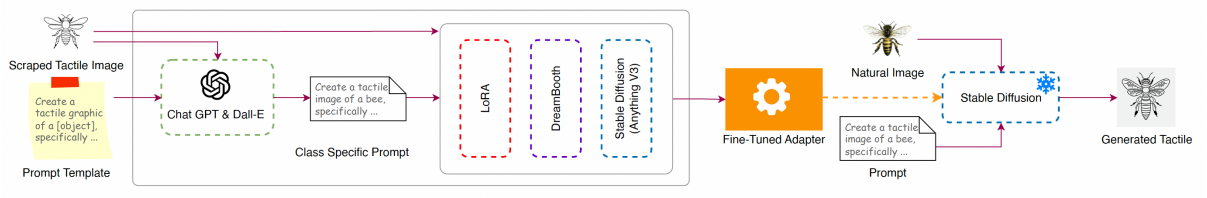


Figure 1.2: TactileNet Flow Diagram from [Khan et al., 2025].

1.3 Objectives

This project focuses on addressing the limitations of scalability and generalization by expanding and structuring the dataset and using it in the reinforcement learning (RL) inspired SD frameworks. Given the small sample of 132 expert evaluated images, the dataset was created containing over 1,200 image pairs, and each pair was annotated with targeted feedback from tactile professionals and classified into four evaluation categories: *Accept as Is*, *Minor Edit*, *Major Edit*, and *Reject*, according to some questions and guidelines that need to be met as seen in the sample size. This richer dataset now provides the foundation to experiment with different training strategies, including alternative loss functions, augmentation methods, and model refinements, with the objective of enhancing TactileNet's ability to generate tactile graphics that are both accessible and practical for visually impaired users.

1.4 Gap Analysis

The initial TactileNet framework demonstrated the feasibility of generating tactile graphics using Stable Diffusion (SD) fine-tuned with LoRA and DreamBooth. However, several limitations identified in that study created clear gaps that needed to be addressed for the system to progress beyond proof-of-concept.

First, the dataset was both small, containing only 1,029 tactile images across 66 classes. This scarcity restricted the model's ability to generalize and resulted in inconsistent quality across object types, particularly for organic shapes and classes with limited representation.

Second, the original approach relied on 66 separate LoRA adapters, one for each class, which achieved higher fidelity for individual categories but was not scalable. Attempts

to shift toward a single, unified adapter caused a noticeable drop in tactile readability, revealing that the dataset and feedback mechanisms were not sufficient to support a generalized model.

Third, while the evaluation process in the original study included expert feedback and a four-level quality rating that were mentioned earlier, these assessments were used only for reporting and not systematically incorporated into the dataset for model training or refinement. As a result, the model lacked targeted, structured feedback that could guide future iterations toward producing guideline-compliant tactile graphics.

Finally, the previous study proposed several avenues for improvement including expanding the dataset, experimenting with alternative loss functions, and refining data augmentation techniques, but these directions were left for future work and were not fully explored within the scope of that study.

This project directly addresses these gaps by:

1. **Expanding the feedback dataset tenfold** to over 1,200 image pairs and balancing weakly represented categories.
2. **Structuring each image pair with targeted feedback and evaluation categories** aligned with BANA guidelines.
3. **Experimenting with training strategies** including loss functions, augmentation methods, and unified adapter setups to explore potential improvements in tactile readability and scalability of the TactileNet framework.

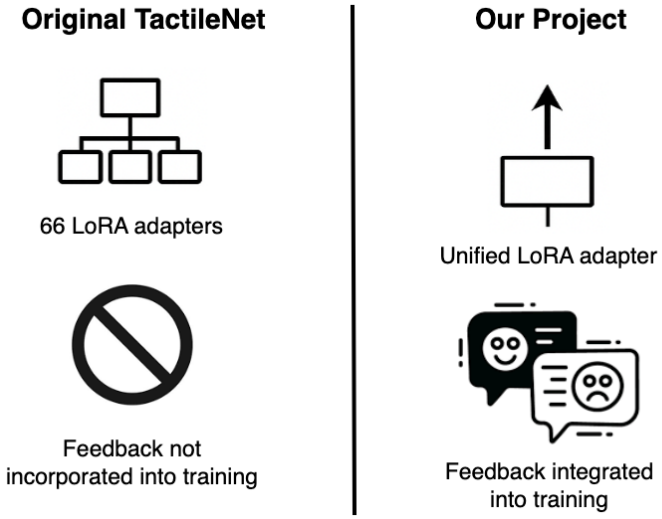


Figure 1.3: Original TactileNet vs Our Project.

1.5 Report Structure

The present report is organised into six chapters, including this Introduction chapter. An overview of each chapter is provided below:

Chapter 2 – Literature Review
This chapter reviews the key concepts, standards, and research relevant to generating tactile graphics for visually impaired users. It covers tactile graphics design guidelines and accessibility standards, the *TactileNet* evaluation protocol for assessing tactile images beyond conventional metrics, and how artificial intelligence is being applied through generative models, diffusion methods, reinforcement learning (RL)–based frameworks, and multimodal systems to produce clearer, touch-readable graphics.

Chapter 3 – Methodology
This chapter explains the overall system design, detailing dataset preparation, model setup, training strategies, and evaluation methods. It also introduces the reward model and custom loss functions integrated into the pipeline, as well as the final LoRA-based adapter configuration and the fine-tuned Stable Diffusion baseline.

Chapter 4 – Experimental Settings
This chapter describes the training and testing configurations, including hyperparameters, epoch counts, and loss function variations. The rationale behind these choices and how they relate to testing underfitting, overfitting, and model performance is also discussed. Different baseline models were also evaluated to observe their influence on overall generation quality and alignment with tactile feedback. Experiments further include the evaluation of the final LoRA-based adapter and the fine-tuned Stable Diffusion baseline trained on the expanded feedback dataset, alongside variations in reward weighting, extended training durations, and inference settings to assess their impact on tactile readability and adherence to BANA guidelines.

Chapter 5 – Results and Evaluation
This chapter presents quantitative and qualitative results from the experiments, comparing different models, loss functions, and training settings. It also evaluates the visual fidelity, CLIP reward scores, and the overall impact of each approach. Results include a detailed comparison between the final LoRA-based adapter and the fine-tuned Stable Diffusion baseline, as well as ablation studies on CLIP–LPIPS reward weighting, extended training durations, and inference parameter variations. Both numerical metrics and qualitative assessments are used to analyze tactile readability, structural fidelity, and compliance with BANA guidelines.

Chapter 6 – Summary and Conclusion

This chapter summarizes the key findings, discusses limitations, and draws final conclusions. It also outlines potential directions for future work.

Chapter 2

Literature Review

This chapter reviews the foundations and recent developments relevant to tactile graphics for people with visual impairments. It begins by outlining tactile graphics and accessibility standards that guide the creation of touch-readable materials. It then summarizes the evaluation protocol proposed in the *TactileNet* [Khan et al., 2025], which defines how the quality of tactile images should be assessed beyond traditional metrics. Finally, the chapter examines how artificial intelligence (AI) is being applied to tactile graphics, covering generative models for image simplification, diffusion-based methods for controllable image editing, and large language and multi-modal models that enable richer, more accessible descriptions. Together, these sections provide context for how tactile graphics are designed, evaluated, and increasingly automated using AI.

2.1 Tactile Graphics and Accessibility Standards

Tactile graphics are essential for providing access to visual information such as diagrams, charts, and illustrations for people with visual impairments. Easy access to such information improves literacy, education, employment, and independence for individuals who are blind or visually impaired [Edman, 1992]. Rosenblum and Herzberg (2015) found that students who used tactile graphics in their mathematics and science classes defined a "good" tactile graphic primarily by the clarity of the information presented.

Traditionally, creating tactile graphics is a manual and highly specialised process. Producing even a simple tactile diagram can take 10–15 minutes, while complex images may require hours of work by accessibility specialists. Tools like CorelDRAW, Adobe Illustrator, and embossers such as ViewPlus are commonly used, but the process remains time-consuming and costly [Khan et al., 2025].

There are also established resources and libraries for tactile graphics, including those from the American Printing House for the Blind (APH) [APH, 2024] and Perkins [Perkins, 2025]. The Braille Authority of North America (BANA) [BANA, 2025] provides formal guidelines to standardise Braille and tactile graphics. These guidelines cover requirements such as line thickness, symbol spacing, and overall clarity to ensure that tactile graphics are both usable and understandable. Teachers and instructors working with visually impaired students are expected to follow these standards when developing tactile materials.

This project draws directly on these guidelines to ensure that generated images and dataset annotations reflect best practices for tactile readability.

2.2 Evaluation Protocol

In the *TactileNet* paper, Khan et al. designed a custom evaluation protocol to assess the quality and usability of generated tactile graphics, since conventional metrics like FID and SSIM do not capture tactile-specific requirements such as line clarity, texture, and adherence to accessibility standards. Expert reviewers evaluated 132 tactile graphics (66 generated and 66 sourced) against 66 reference natural images using a side-by-side interface, without knowing whether the tactile graphic was generated or sourced.

Evaluators answered four key questions:

1. **Natural Features & Posture Alignment:** Does the tactile image accurately match the pose and key features of the reference image?
2. **Adherence to Tactile Graphics Guidelines:** Does the graphic follow BANA and similar tactile design standards?
3. **Quality Rating:** Categorized as *Accept as Is*, *Minor Edits*, *Major Edits*, or *Reject*.
4. **Optional Feedback:** Free-text comments suggesting specific improvements (e.g., line clarity, texture, or structure).

Natural images were carefully selected to pair with gold-standard tactile graphics from established libraries such as APH and Perkins. For each class, evaluators compared:

- **Sample 1:** A generated tactile graphic (created from original and paraphrased prompts).
- **Sample 2:** A sourced tactile graphic from trusted libraries.

This combined approach leveraged expert judgment along with limited SSIM checks to ensure that generated tactile graphics were not only structurally sound but also practically usable by individuals with visual impairments.

There are two examples below of tactile images that follow BANA and similar tactile design guidelines. In Figure 2.1, the dog illustration demonstrates several important concepts. The image is drawn from a side view, eliminating all 3D aspects. The ear, eye, and nose are represented with solid black shapes that are clearly separated from the body for distinction. The outer contour of the body is drawn with a thicker, darker line than the inner details. A texture pattern appears on the dog's body, but it is omitted from the far legs to make them distinguishable from the nearer legs. The tail is positioned slightly apart from the body to prevent confusion for tactile readers.

In Figure 2.2, the horse image also applies these principles. The image is also drawn from a side view, eliminating all 3D aspects. Light grey is used to indicate the far legs, while black defines the lower parts of the legs. A darker grey is used for the mane and tail to convey a texture different from the body. Finally, black is used again to mark the nostril, mouth, eyes, and ears for clear identification.

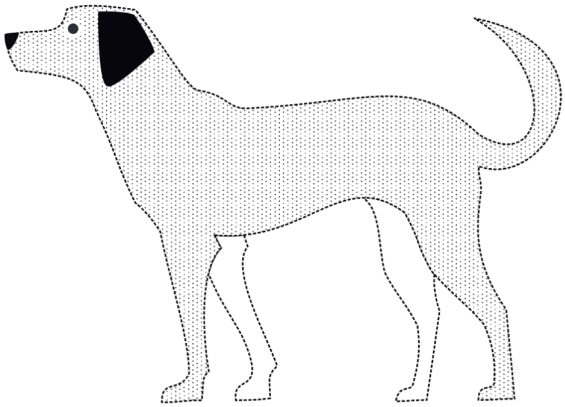


Figure 2.1: Tactile graphic of a dog.

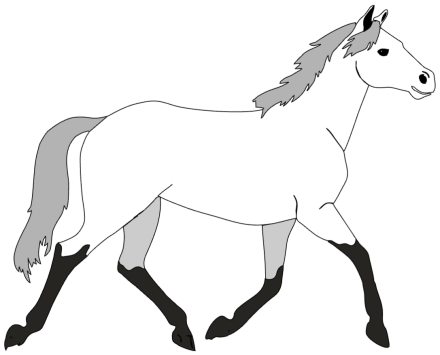


Figure 2.2: Tactile graphic of a horse.

2.3 AI in Tactile Graphics

Recent research has increasingly explored how generative AI can support the creation of tactile graphics for visually impaired users. This includes systems that generate tactile-ready images and interactive tools, methods for simplifying complex visuals using models like GANs, and newer diffusion-based approaches that allow precise, instruction-driven edits. Together, these advances show how AI is shifting tactile graphics from a manual

process to one that is more scalable and accessible.

2.3.1 AI Applications in Tactile Graphics

Dzhurynskyi et al. (2024) introduced a model that combines a Bidirectional and Auto-Regressive Transformer (BART) with a Vector Quantized Variational Auto-Encoder (VQ-VAE) to generate tactile graphics from text prompts. Their system was evaluated using metrics such as cross-entropy, perplexity, and CLIP Score, and testing with educational and rehabilitation institutions showed it could drastically reduce production time and the need for specialized expertise. This work highlights how advanced generative architectures can automate what has traditionally been a highly manual process.

Other studies have focused on converting existing visual materials into tactile forms. For instance, Pakenaite et al. (2024) presented Pic2Tac, a system that translates semantic content from photographs into tactile symbols, demonstrating strong results in user studies with both sighted and visually impaired participants. Similarly, Tanaka et al. (2023) explored how Generative AI and 3D printing can transform ordinary picture books into tactile picture books, opening up new opportunities for inclusive learning experiences.

Beyond purely visual-to-tactile conversion, researchers have also investigated ways to enhance tactile graphics with real-time interactivity. The TAURIS project [Zeinullin, 2025], a PhD study involving 20 participants, developed a fingertip-tracking system for mobile devices that provides instant audio descriptions as users explore tactile graphics. This system not only improved the speed and accuracy of interaction but also demonstrated a notable advantage in memory retention compared to screen readers.

Work has also begun to explore the use of large language models for multimodal educational tools. Wu et al. (2025) developed TaleVision, which uses LLMs to generate touchable images and sound effects for visually impaired children. In testing with 19 participants, nearly all reported high satisfaction and positive experiences, suggesting strong potential for combining tactile and auditory outputs to create richer learning tools.

Collectively, these projects illustrate the breadth of AI applications in tactile graphics: from automating graphic generation and enhancing printed materials, to introducing interactivity and multimodal content, marking an important shift toward scalable, user-centered solutions for accessibility.

2.3.2 Generative Models for Image Simplification

Generative Adversarial Networks (GANs) have become a cornerstone for image simplification tasks relevant to tactile graphics, such as converting photographs into line drawings

or stripping away visual clutter. Early GAN frameworks relied on end-to-end learning, generating images directly from uniform noise distributions. While effective, these early models lacked interpretability and offered little control over what aspects of the image were simplified.

Decomposing Structure and Style

More recent work has sought to decompose the generation process for greater structure and flexibility. For example, [Wang and Gupta, 2016] introduced the Style and Structure GAN, which splits image generation into two coordinated components: Structure-GAN, responsible for producing a surface normal map, and Style-GAN, which applies textures onto that structure. This separation of structure from style allows models to preserve essential shapes while modifying or simplifying textures, a capability highly relevant to tactile graphics, where clear boundaries must be retained even as unnecessary details are abstracted away.

Image-Driven Simplification Approaches

Other researchers have explored alternatives to purely geometry-driven simplification methods, which often strip away perceptually important information.
[Lindstrom and Turk, 2000] proposed an image-driven simplification approach that compares rendered images of the original and simplified models to guide the simplification process. This framework maintains high-fidelity silhouettes, removes visually insignificant regions, and strikes a balance between geometric and appearance-based trade-offs. Although developed for 3D models, the principle, prioritizing perceptually important structures while simplifying less critical elements, aligns closely with the goals of tactile graphics, where maintaining clear, touchable shapes matters more than reproducing every visual nuance.

Stylization for Tactile-Friendly Outputs

GAN-based models have also been adapted for stylization tasks that naturally support tactile interpretation. CartoonGAN [Chen et al., 2018], for instance, converts real-world photos into cartoon-style images with bold edges and simplified textures, producing outputs that more closely resemble tactile-ready line art.

Pix2Pix and Its Improvements

Another foundational model, Pix2Pix, has been widely used for image-to-image translation tasks such as sketch-to-image and outline-to-photo conversion. However, standard Pix2Pix often loses critical structural information during encoding and decoding, and its training lacks constraints to maintain fine details. Zhao et al. (2023) addressed these limitations by integrating a U-Net generator with denser skip connections to reduce information loss and adding a differential image discriminator to improve supervision. Their improved Pix2Pix variant demonstrated significantly better image quality and structural preservation: an important advance for tactile graphics, where even the smallest lost edge or contour can compromise the tactile readability of the output.

2.3.3 Diffusion Models for Tactile and Accessible Image Generation

Diffusion models have rapidly become the dominant paradigm for high-quality image generation, offering state-of-the-art performance in controllable, prompt-driven synthesis. Unlike GANs, which map noise to images in a single step, diffusion models iteratively denoise random noise, providing more stable training and finer control over the generation process. This controllability has made diffusion approaches especially attractive for accessibility-focused research, where precision, structure, and repeatability are essential.

From Visual Fidelity to Tactile Content

Early research has started extending diffusion models beyond purely visual fidelity to tackle tactile and multimodal tasks. Dou et al. (2024) introduced *Tactile-Augmented Radiance Fields* (TARF), a system that unifies vision and touch into a shared 3D space. Their conditional diffusion model takes an RGB image and depth map from a neural radiance field and produces corresponding tactile "images." By training on the largest dataset of spatially aligned visual–tactile pairs to date, TARF demonstrated that diffusion models can accurately translate visual cues into tactile features, an important step toward AI-driven tactile data generation.

Instruction-Driven Editing

A second major development has been the use of diffusion models for instruction-driven image editing, where text commands guide specific changes to images. Paul (2023) presented the open-source project *Instruction-Tuning Stable Diffusion*, inspired by FLAN

[Wei et al., 2021] and InstructPix2Pix [Brooks et al., 2023], which fine-tunes Stable Diffusion (SD) to follow explicit instructions paired with input images (e.g., "apply a cartoon filter"). This ability to modify images through natural language commands is directly relevant to tactile graphics: simplification, outlining, or re-texturing for tactile use can be framed as targeted "instructions."

Brooks et al. (2023) expanded this idea with *InstructPix2Pix*, a conditional diffusion model trained on a large dataset of synthetic image–instruction pairs created using GPT-3 and Stable Diffusion. InstructPix2Pix performs edits in a single forward pass, without requiring fine-tuning or inversion for each image, making it extremely fast and flexible. This framework is highly applicable to tactile graphics generation, where edits like removing shading, emphasizing contours, or simplifying textures must be executed quickly and consistently.

Shagidanov et al. (2024) introduced *Grounded-Instruct-Pix2Pix*, building on Brooks et al.'s work by adding automatic "target grounding." This ensures that edits only affect the intended area (e.g., just the "cat" in a complex background). Such localized precision is crucial for tactile graphics, as it allows selective simplification or re-texturing of certain objects while leaving the surrounding scene intact.

Spatial Conditioning with ControlNet

While instruction-driven editing handles "what" to change, spatial conditioning handles "where" to change it. Zhang et al. (2023) addressed this with *ControlNet*, a neural architecture that adds spatial controls to large, pretrained diffusion models like Stable Diffusion. ControlNet preserves the base model's pretrained knowledge while adding "zero convolution" modules (zero-initialized layers that progressively learn how to respond to control inputs like edges, segmentation maps, or human poses). This means users can tightly guide the generation process with structural cues, an ability particularly promising for tactile graphics. For example, a ControlNet-based system could generate tactile graphics that precisely follow edge maps or segmentation boundaries, ensuring tactile clarity and alignment with visual structure.

Implications for Accessibility

Together, these developments position diffusion models as a powerful toolkit for accessibility research. Their ability to incorporate multimodal inputs, follow human-readable instructions, and respect structural guidance means they can not only generate high-quality images but do so in ways that meet the unique demands of tactile graphics. This opens the door to scalable, user-centered systems capable of producing tactile images that are

both accurate and touch-friendly.

2.3.4 Large Language and Multi-Modal Models for Accessibility

Large Language Models (LLMs) are increasingly being explored as tools for bridging text and vision, offering new ways to structure, describe, and prioritize visual information for accessibility. Rather than simply labeling images, these models can provide richer semantic context and enable new forms of interaction that are particularly relevant to tactile graphics.

Bertolotto et al. (2021) demonstrated how LLMs can enrich vision-based systems by generating semantic descriptions for class labels. Instead of treating classes as isolated categories, their method used an LLM to create descriptive text for each label, which was then embedded and used as the training target for image classification. This approach encouraged the model to learn a hierarchically structured feature space, leading to more semantically meaningful errors (e.g., confusing a "dog" for a "wolf" rather than for a "car"). For tactile graphics, such semantic structuring could help generative models decide which visual concepts to emphasize and how to frame prompts, ensuring that outputs preserve the most relevant and interpretable features for touch.

Building on this theme of richer interpretation, Ricci et al. (2024) expanded the scope of image captioning by introducing a dialogue-based approach using ChatGPT. Their system moved beyond the typical single-sentence caption, instead generating a multi-turn question-and-answer dialogue about an image and summarizing the exchange into enriched, context-aware captions. For tactile graphics, such a dialogue-driven captioning framework could generate layered explanations for complex images: first summarizing the overall scene, then adding detail about key objects or relationships, helping visually impaired users build a more comprehensive mental model of the content.

Collectively, these works highlight how LLMs and multi-modal systems can go beyond basic captioning or labeling, enabling structured, meaningful descriptions that could feed directly into tactile graphics generation pipelines, supporting not only clearer image simplification but also richer, more accessible explanations of visual content.

2.3.5 RL-based Frameworks

While GANs, diffusion models, and large language models provide powerful generative and editing capabilities, reinforcement learning (RL) offers a complementary paradigm: directly optimizing policies to maximize accessibility-focused objectives. A central challenge in applying RL to real-world problems is specifying goals that are both precise and

practical to obtain.

Christiano et al. (2017) introduced deep reinforcement learning from human preferences, in which agents learn a reward model from pairwise comparisons of short trajectory segments provided by non-expert annotators. This separates learning the reward function from learning the policy, enabling the agent to acquire complex behaviors with minimal human oversight. Ziegler et al. (2019) extended this paradigm to language models, demonstrating that fine-tuning from human preferences (RLHF) can align generative models with nuanced stylistic and semantic goals.

Rocamonde et al. (2024) further showed that pretrained vision language models (VLMs) can act as zero-shot reward models, producing a reward signal by measuring similarity between generated outputs and natural language task descriptions. Such VLM-based rewards could complement human ratings by automatically assessing whether a tactile output meets accessibility-focused prompts, for example "a simplified, high-contrast outline with Braille labels", and provide scalable and consistent feedback.

Together, these works suggest that RL-inspired optimization, whether through human preferences [Christiano et al., 2017; Ziegler et al., 2019] or multimodal reward models [Rocamonde et al., 2024], provides a promising route to develop tactile generation systems that are adaptive, user-centered, and scalable.

2.3.6 Summary

This chapter reviewed how AI is transforming the creation and use of tactile graphics for visually impaired users (Table 2.1). First, it explored applications of AI in tactile graphics, including systems that generate tactile-ready images, convert visual materials like books and photos into tactile forms, and add interactive features such as fingertip-tracked audio descriptions. Next, it examined generative models for image simplification, showing how GAN-based approaches like StyleGAN, Pix2Pix, and CartoonGAN simplify visuals while preserving essential structure for touch interpretation. It then discussed diffusion models, which offer instruction-driven editing, spatial control, and multimodal capabilities through frameworks like InstructPix2Pix, ControlNet, and TARF, opening new possibilities for precise, scalable tactile content creation. Finally, it highlighted the role of large language and multimodal models, demonstrating how LLMs like ChatGPT can enrich image descriptions and guide generative pipelines toward semantically meaningful, touch-friendly outputs. Building on this, reinforcement learning (RL)–based frameworks offer a complementary paradigm by directly optimizing models with accessibility-focused reward signals, whether derived from human preferences or multimodal evaluators.

Together, these advances mark a shift from manual, specialist-driven production of tactile

graphics to AI-powered, scalable systems capable of generating richer, clearer, and more accessible tactile experiences (Table 2.2).

Table 2.1: Topics of References cited across subsections of Section 2.3: AI in Tactile Graphics.

Topic	**# of References**
AI Applications in Tactile Graphics	5
Generative Models for Image Simplification	4
Diffusion Models for Tactile and Accessible Image Generation	5
Large Language and Multi-Modal Models for Accessibility	2
RL-based Frameworks	3
Total	**19**

Table 2.2: Summary of References Cited in Section: AI in Tactile Graphics.

Citation	**Description**
Dzhurynskyi et al. (2024)	Introduced a BART + VQ-VAE model for generating tactile graphics from text prompts.
Pakenaite et al. (2024)	Developed Pic2Tac, converting photos into tactile symbols using semantic information.
Tanaka et al. (2023)	Explored converting printed picture books into tactile books using generative AI and 3D printing.
Zeinullin (2025)	Developed TAURIS, a fingertip-tracking system for tactile graphics with audio feedback.
Wu et al. (2025)	Built TaleVision, using LLMs to generate touchable and audible content for visually impaired children.
Wang et al. (2016)	Proposed Style and Structure GAN to separate texture and shape during image generation.
Lindstrom (2000)	Introduced image-driven simplification by comparing rendered images to preserve perceptually important shapes.
Chen et al. (2018)	Created CartoonGAN for photo-to-cartoon conversion with bold outlines and flat textures.
Zhao et al. (2023)	Improved Pix2Pix using U-Net and differential image discriminator to enhance structure preservation.
Dou et al. (2024)	Proposed TARF: a conditional diffusion model to generate tactile features from RGB + depth inputs.
Paul (2023)	Released Instruction-Tuned Stable Diffusion for applying text-driven modifications to images.
Brooks et al. (2023)	Created InstructPix2Pix, enabling fast instruction-based edits without per-image fine-tuning.
Shagidanov et al. (2024)	Introduced Grounded-Instruct-Pix2Pix for localized, instruction-specific image editing.
Zhang et al. (2023)	Developed ControlNet, enabling spatial conditioning for diffusion models using edge/pose maps.
Bertolotto et al. (2021)	Used LLMs to create semantic-rich label embeddings, improving classification hierarchy.
Ricci et al. (2024)	Proposed dialogue-driven image captioning with ChatGPT for richer semantic outputs.
Christiano et al. (2017)	Introduced deep reinforcement learning from human preferences, enabling agents to learn reward models from pairwise human comparisons.

Summary of References Cited in Section: AI in Tactile Graphics (continued).

Citation	**Description**
Ziegler et al. (2019)	Applied preference-based reinforcement learning to fine-tune language models, aligning outputs with nuanced stylistic and semantic goals.
Rocamonde et al. (2024)	Demonstrated that vision–language models can serve as zero-shot reward models for reinforcement learning, enabling scalable multimodal reward estimation.

Chapter 3

Methodogy

3.1 Pipeline Overview

Referring to Figure 3.1, the workflow of this project can be divided into three main phases. The first phase is dataset expansion. Each image pair in the dataset contains four elements: the tactile image, the corresponding natural image, written feedback highlighting what is wrong with the tactile image based on evaluator questions from the *TactileNet* paper [Khan et al., 2025] (whether the tactile image accurately matches the pose and key features of the reference image, and whether it follows tactile graphics guidelines such as those from BANA), and finally a rating of the tactile image using the natural image as the reference.

The second phase is the training phase. We used a Python script to fine tune a Stable Diffusion model with LoRA adapters to generate tactile images using a reward guided approach inspired by reinforcemnet learing (RL) (Table 3.1). The script loads paired tactile and natural images, uses a CLIP reward model to score how well the generated images align with the provided instructions, and applies several custom loss functions within a tailored training loop. Nonessential parts of the model are frozen to improve efficiency, training progress is logged with Weights & Biases, and the resulting LoRA adapters are saved to create a streamlined pipeline for instruction guided and accessibility focused image generation.

The third phase is the testing phase. A separate Python script evaluates the LoRA fine tuned Stable Diffusion models by generating and scoring images from a test dataset. It processes a set of prompts and input images, applies each model, and uses the CLIP reward model to calculate semantic alignment scores between the generated images and their prompts. For each model, the generated images are saved and the evaluation results are compiled into a CSV file. This phase forms the evaluation and ablation stage of the

pipeline, providing a clear comparison of how different training strategies affect image quality and alignment with prompts.

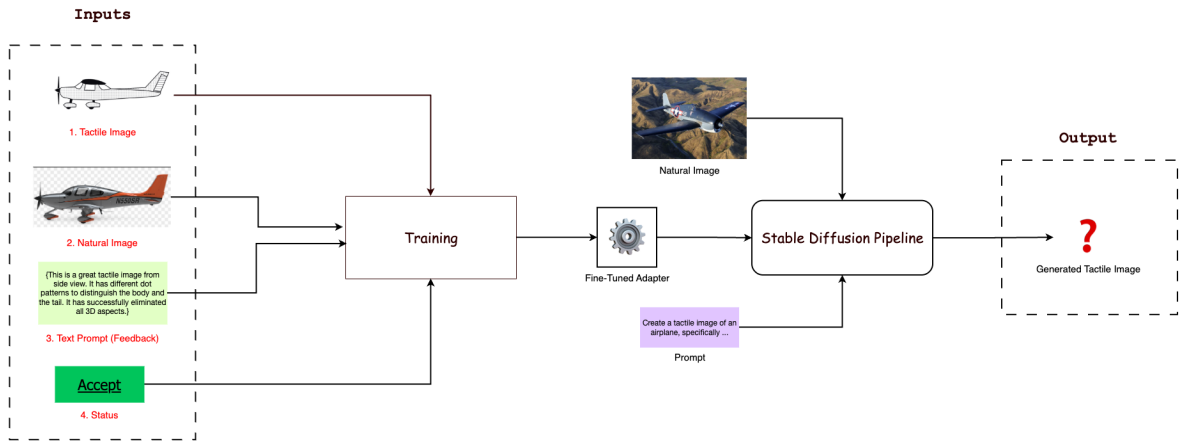


Figure 3.1: Initial Project's Flow Diagram.

Table 3.1: Bridging RL Concepts with Our Pipeline

RL Concept	**Our Setup**
Agent	Stable Diffusion model (with or without LoRA adapter)
Environment	Training pipeline with feedback-based dataset
Action	The generated tactile image
State	The natural image + instruction (prompt)
Reward	CLIP similarity score between generated image and instruction

3.2 Dataset Expansion

The *TactileNet* paper highlighted the need for a larger dataset, which led me to expand the collection by evaluating additional image pairs. Understanding how to evaluate these pairs was challenging, as the criteria became clearer only through the process itself, with each new example revealing gaps in my understanding and prompting refinements to the evaluation framework.

Two main sources guided this process: (1) earlier expert feedback and (2) the BANA (Braille Authority of North America) guidelines for tactile graphics. The expert feedback, while useful, was simplified and often vague, which made it less effective for training purposes. To address this, we reworked the feedback into clearer, more directed instructions and linked each comment to specific BANA standards when relevant.

3.2.1 Building and Structuring the Dataset

Originally, the expert feedback was stored in a Word document. We reformatted it into an Excel sheet to make it compatible with the baseline code and easier for future contributors to expand. This sheet was then converted into CSV and JSONL formats so the script could read the data directly. These files, along with the image folders, were uploaded to Google Drive and linked in the project's GitHub README for open access.

The dataset originally had four outcome labels: *Accept as is*, *Minor Edit*, *Major Edit*, and *Reject*. For simplicity and better training consistency, these were consolidated into two categories:

- **Accept:** Accept as is & Minor Edit
- **Reject:** Major Edit & Reject

Status	rating
Reject	reject
Major Edit	reject
Accept as is	accept
Minor Edit	accept

Figure 3.2: Status and its corresponding rating from the CSV file.

Each entry in the dataset now includes:

1. The natural image
2. The tactile image
3. A clear directed feedback statement explaining what is wrong and how to fix it
4. A rating (Accept/Reject)

Category	T Filename	N Filename	feedback_text	rating
Banana	1.png	1.jpg	Rotate image 45 degrees anticlockwise to match the posture of the original image.	accept

Figure 3.3: Example of a dataset row from the CSV file.

3.2.2 Challenges and Refinements

The most significant challenge was the vague nature of the expert feedback. It lacked the directness needed for a model to learn corrective actions. We rewrote much of this feedback to explain not only *what* was wrong but also *why* it was wrong and *how* to fix it.

Common problems found in tactile graphics included:

- Missing or misleading textures
- Inaccurate posture or object orientation
- Unnecessary shading that confused tactile
- Poor representation of key features compared to the natural image

Example of an Accepted Case:

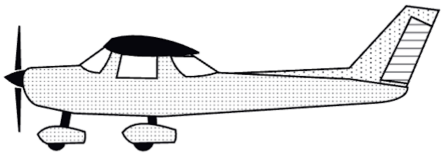
 **Airplane Tactile Image**	 **Airplane Natural Image**
Feedback: This is a great tactile image from side view. It has different dot patterns to distinguish the body and the tail. It has successfully eliminated all 3D aspects.	
Rating: accept	

Table 3.2: An example of an accepted data entry.

Example of Reject Cases:

 **Airplane Tactile Image**	 **Airplane Natural Image**
Feedback: Add the missing landing gear.	
Rating: reject	

Table 3.3: An example of a rejected data entry due to a missing feature.

 **Airplane Tactile Image**	 **Airplane Natural Image**
Feedback: Remove 3D aspects by showing a proper front view of the natural image. Adjust proportions to match the reference image. It doesn't resemble the reference airplane in key structural ways (engines, wing shape, windows, and emergency doors).	
Rating: reject	

Table 3.4: An example of a rejected data entry due to the presence of 3D aspects.

3.2.3 Summary

This section outlined the process of expanding and restructuring the TactileNet dataset to support more effective training and evaluation. The dataset was reformatted from scattered Word documents into structured CSV and JSONL files, with each entry containing a natural image, a tactile image, clear feedback, and an accept/reject rating. Feedback from experts was rewritten to provide actionable, targeted instructions. Labels were consolidated from four to two categories (Accept and Reject) for consistency. Common issues identified in the dataset included missing textures, 3D elements, and inaccurate object proportions. Accepted and rejected examples illustrated how these refinements were tested in an attempt to improve clarity and standardization, creating a more reliable and scalable dataset for model development.

3.3 Initial Training Pipeline Implementation

The training pipeline was implemented in Python and designed to fine-tune a Stable Diffusion model for generating tactile graphics using LoRA adapters and reward-guided optimization. The script `"train.py"` is structured into distinct stages, each handling a key component of the training workflow.

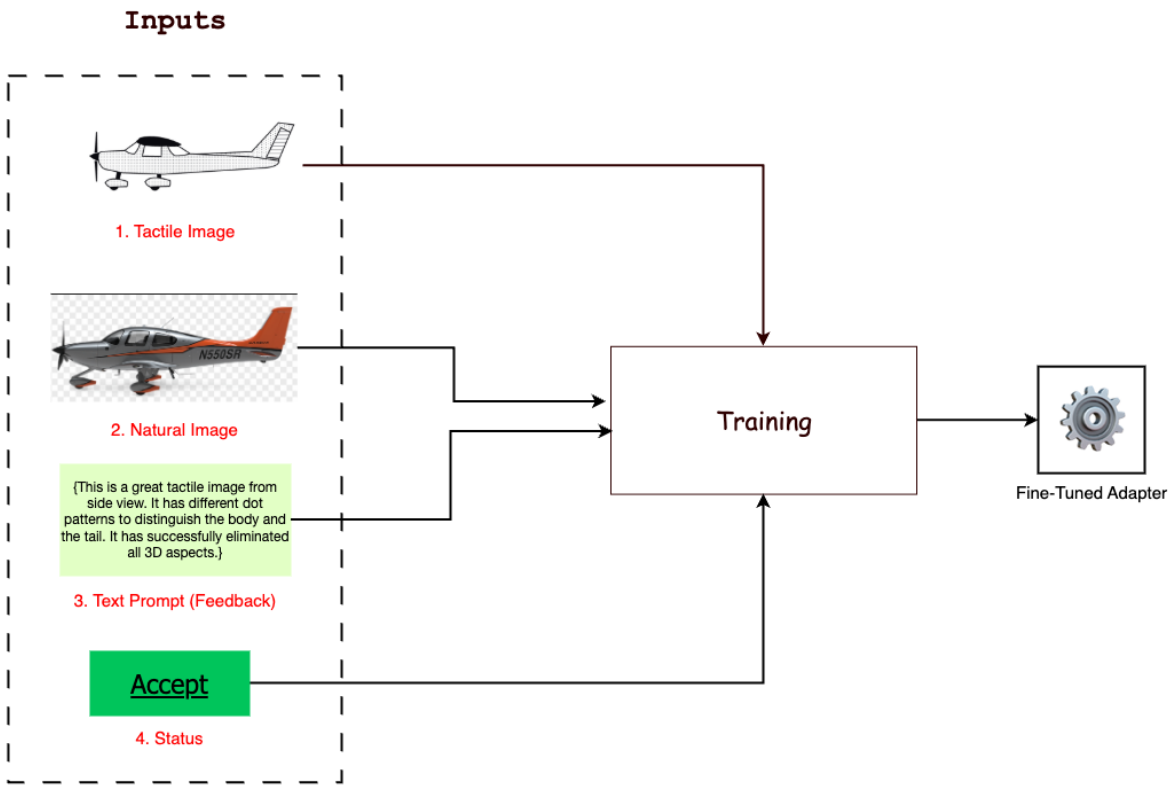


Figure 3.4: Training Phase from the Project's Flow Diagram.

3.3.1 Dataset Loading and Preprocessing

The pipeline begins by loading tactile–natural image pairs from a JSONL file. Each entry includes the tactile image, natural reference image, feedback text, and an acceptance rating. Images are resized to 512×512 pixels, converted to RGB, and preprocessed with normalization for the Variational Autoencoder (VAE). Augmentations such as horizontal flips and color jittering are applied to expand data variability.

```
def load_image_pairs(jsonl_path: Path) -> List[Dict]:
    pairs = []
    with open(jsonl_path) as f:
        for line in f:
            try:
                data = json.loads(line)
                rating = 1 if data["rating"].lower() == "accept"
                    else 0
                input_img_path = Path(data["input_image"])
                gen_img_path = Path(data["generated_image"])

                if not input_img_path.exists() or not gen_img_path
                    .exists():
                    continue
```

```
13
14                  pairs.append({
15                      "input_image": Image.open(input_img_path).
                           convert("RGB").resize(IMAGE_SIZE),
16                      "generated_image": Image.open(gen_img_path).
                           convert("RGB").resize(IMAGE_SIZE),
17                      "instruction": data["feedback_text"] if data["
                           feedback_text"].lower() != "nan" else "",
18                      "rating": rating
19                  })
20      return pairs
```

Listing 3.1: Loading and preprocessing image pairs

3.3.2 Reward Model Integration

To guide training, a CLIP-based reward model was implemented. The reward model calculates semantic similarity scores between generated images and corresponding text instructions, providing a quantitative "reward" signal. These scores are used later in the custom loss functions to prioritize alignment between model outputs and tactile feedback instructions.

```
1  class CLIPRewardModel:
2      def __init__(self):
3          self.model = CLIPModel.from_pretrained("openai/clip-vit-
               base-patch32").to(DEVICE)
4          self.processor = CLIPProcessor.from_pretrained("openai/
               clip-vit-base-patch32")
5
6      def calculate_reward(self, images: List[Image.Image], texts:
           List[str]) -> torch.Tensor:
7          inputs = self.processor(text=texts, images=images,
               return_tensors="pt", padding=True, truncation=True).to(
               DEVICE)
8          with torch.no_grad():
9              outputs = self.model(**inputs)
10             sims = outputs.logits_per_image
11             diagonal = sims.diagonal()
12             return diagonal
```

Listing 3.2: Reward model integration

3.3.3 Loss Function Variants

Several custom loss functions were implemented to experiment with different training dynamics. Each formulation is described with its purpose and variable definitions.

- **MSE Loss** – standard pixel-level loss.

$$L_{MSE} = \|\hat{\epsilon} - \epsilon\|^2$$

  Where:

  – $\hat{\epsilon}$: noise predicted by the UNet

  – ϵ: ground-truth noise sampled during training

```
def mse_loss_fn(pred, target, **kwargs):
    return F.mse_loss(pred, target)
```

Listing 3.3: MSE Loss

- **Clip-Weighted MSE** – adjusts the MSE based on the CLIP similarity score.

$$L_i = (1 - \text{CLIP}(x_i, t_i)) \cdot L_{MSE}$$

  Where:

  – x_i: generated tactile image

  – t_i: instruction or feedback text

  – $\text{CLIP}(x_i, t_i)$: similarity score from CLIP model (ranges from 0 to 1)

```
def clip_weighted_mse_loss(pred, target, ratings, **kwargs):
    accepted_mask = torch.tensor(ratings, device=DEVICE, dtype=torch.bool)
    loss_weights = torch.where(
        accepted_mask,
        torch.tensor(0.5, device=DEVICE),
        torch.tensor(2.0, device=DEVICE)
    )
    base_loss = F.mse_loss(pred, target)
    return base_loss * loss_weights.mean()
```

Listing 3.4: Clip-Weighted MSE

- **Reward-Weighted MSE** – scales the loss by rating-based weights.

$$L = w_i \cdot L_{MSE} \quad \text{where} \quad w_i = \begin{cases} 2.0, & \text{if rejected} \\ 0.5, & \text{if accepted} \end{cases}$$

  Where:

  - w_i: weight applied based on feedback (higher for rejected images)
  - L_{MSE}: standard Mean Squared Error loss

```
1    def reward_weighted_mse_loss(pred, target, rewards=None,
        **kwargs):
2        rewards = rewards.to(device=pred.device, dtype=pred.
            dtype)
3        base_loss = F.mse_loss(pred, target, reduction='none')
4        weight = rewards.reshape(-1, 1, 1, 1)
5        return (base_loss * weight).mean()
```

Listing 3.5: Reward-Weighted MSE

- **Inverse Reward Loss** – penalizes predictions inversely proportional to the reward values.

$$L_i = \frac{1}{\text{CLIP}(x_i, t_i) + \epsilon} \cdot L_{MSE}$$

  Where:

  - $\text{CLIP}(x_i, t_i)$: CLIP similarity score
  - ϵ: small constant to prevent division by zero
  - L_{MSE}: Mean Squared Error loss

```
1    def inverse_reward_loss(pred, target, rewards=None, **
        kwargs):
2        rewards = rewards.to(device=pred.device, dtype=pred.
            dtype)
3        base_loss = F.mse_loss(pred, target, reduction='none')
4        inverse_weight = (1.0 - rewards).unsqueeze(1).
            unsqueeze(2).unsqueeze(3)
5        return (base_loss * inverse_weight).mean()
```

Listing 3.6: Inverse Reward Loss

- **Margin Reward Loss** – applies a margin-based weighting strategy.

$$L = MSE(x_i, x_i') \cdot \max(0, m - r_i)$$

Where:

- x_i: ground truth noise
- x_i': model's predicted noise
- m: reward margin threshold (e.g. 0.2)
- r_i: CLIP reward score for image-text alignment

```
def margin_reward_loss(pred, target, rewards=None, **
    kwargs):
    rewards = rewards.to(device=pred.device, dtype=pred.
        dtype)
    base_loss = F.mse_loss(pred, target, reduction='none')
    margin = 0.2
    clipped = torch.clamp(margin - rewards.unsqueeze(1).
        unsqueeze(2).unsqueeze(3), min=0)
    return (base_loss * clipped).mean()
```

Listing 3.7: Margin Reward Loss

These loss functions allow exploration of how reward-guided training can improve tactile image generation.

3.3.4 Forward and Backward Diffusion

This stage describes how the diffusion process is applied during training to teach the model to denoise images and update its parameters. It involves three main components:

1. **Forward Diffusion:** Each input image is encoded into a latent representation using the VAE. Random noise is then added at varying timesteps, simulating the forward diffusion process that gradually destroys image information.
2. **Reverse Diffusion Prediction:** The UNet receives the noisy latent, timestep, and text instruction embeddings, and predicts the noise that should be removed, effectively learning to reverse the diffusion process.
3. **Backward Pass (Training):** The predicted noise is compared to the true noise using the selected loss function. Gradients are computed, and LoRA parameters are updated via backpropagation with the AdamW optimizer and a cosine scheduler.

This combined approach ensures that the model learns how to denoise (reverse diffusion) while integrating guidance from instructions, ratings, and rewards during training. Training progress is logged with Weights & Biases (W&B) for monitoring.

3.3.5 LoRA Fine-Tuning Setup

Stable Diffusion's UNet is prepared for LoRA fine-tuning by freezing the VAE and text encoder. A LoRA configuration (rank 8, dropout 0.1) targets key attention layers (`to_q`, `to_k`, `to_v`) across 16 layers, allowing the model to learn tactile-specific adjustments without overfitting or overwriting the base model's general capabilities.

```
    pipe.unet = prepare_unet_for_lora(pipe.unet)
    lora_config = LoraConfig(
        r=8,
        lora_alpha=32,
        target_modules=["to_q", "to_k", "to_v"],
        layers_to_transform=list(range(16)),
        lora_dropout=0.1,
        bias="none",
        fan_in_fan_out=False
    )
    pipe.unet = get_peft_model(pipe.unet, lora_config)
    pipe.unet = pipe.unet.to(device=DEVICE, dtype=torch.float32)
    pipe.vae.requires_grad_(False)
    pipe.text_encoder.requires_grad_(False)
```

Listing 3.8: LoRA Fine-Tuning Setup

After completing all epochs, the trained LoRA adapters are saved to disk, along with metadata including run name, loss type, number of epochs, training duration, and final loss. This ensures reproducibility and supports structured ablation studies on different training configurations.

3.3.6 Summary

This section outlines the full training pipeline for fine-tuning Stable Diffusion to generate tactile graphics. It covers dataset loading and preprocessing, where tactile–natural image pairs are structured and augmented, and the integration of a CLIP-based reward model to provide semantic guidance. Multiple custom loss functions are implemented to test different training strategies, and the diffusion process is detailed, explaining how the model learns to denoise (forward and backward diffusion). Finally, a LoRA fine-tuning setup is described, targeting attention layers while freezing the VAE and text encoder.

3.4 Initial Testing Pipeline Implementation

The testing pipeline, implemented in the "`test.py`" script, evaluates multiple LoRA-fine-tuned Stable Diffusion models on a held-out test dataset. It systematically generates images, scores them using a CLIP-based reward model, and compiles the results for comparison across different loss functions and training strategies.

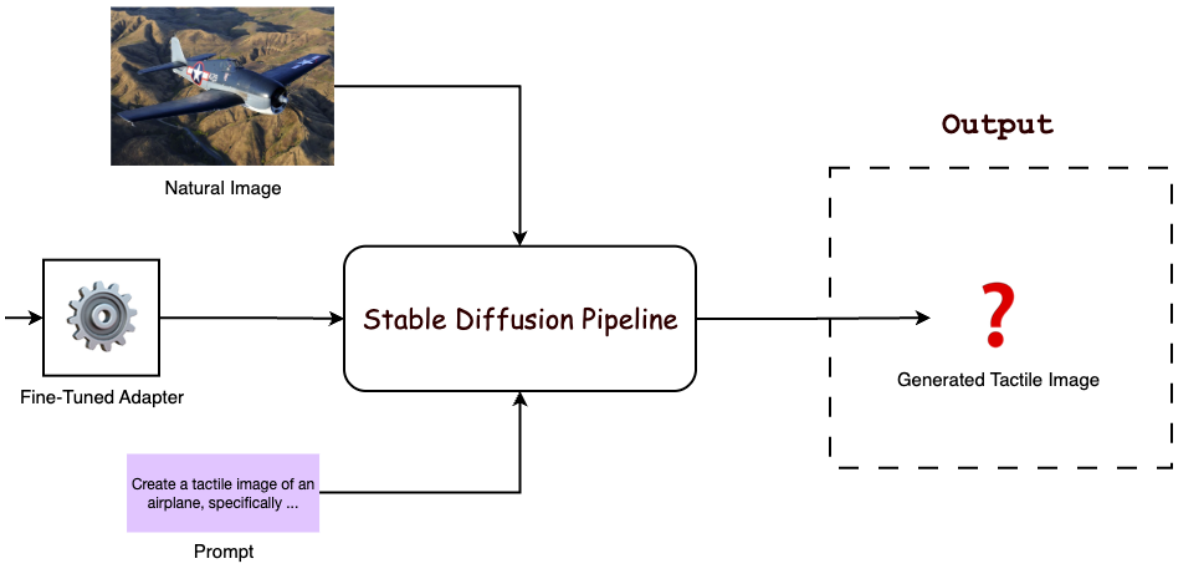


Figure 3.5: Testing Phase from the Project's Flow Diagram.

3.4.1 Model Setup and Configuration

The pipeline supports evaluating several models trained with different loss functions, including MSE, Clip-Weighted MSE, Reward-Weighted MSE, Inverse Reward Loss, and Margin Reward Loss. Each model is linked to its corresponding LoRA adapter path, while a baseline model is also included for comparison. Stable Diffusion v1-5 serves as the base model, and LoRA adapters are loaded dynamically after being created by the "`train.py`" script.

3.4.2 Test Dataset Loading

The test dataset is stored in a JSONL file including 60 entries, where each entry contains the natural image, feedback prompt, and a category. These samples are loaded and iterated over for evaluation.

```
1 def load_test_set(path):
2     samples = []
3     with open(path) as f:
```

```
        for line in f:
            data = json.loads(line)
            samples.append({
                "input_image": data["input_image"],
                "prompt": data["feedback_text"],
                "category": data.get("Category", "unknown")
            })
    return samples
```

Listing 3.9: Loading the test dataset

3.4.3 Image Generation and Evaluation

A CLIP model is used to provide semantic alignment scores between the generated images and the feedback prompts. These scores act as a quantitative measure of how well the model output aligns with textual instructions. For each model, the pipeline uses the Stable Diffusion Img2ImgPipeline to generate images based on the test inputs. The LoRA adapter is loaded, and images are generated using the prompt and the input image as guidance. Each output is scored with the CLIP reward model.

```
def generate_image(pipe, image_path, prompt):
    image = Image.open(image_path).convert("RGB").resize(
       IMAGE_SIZE)
    generator = torch.Generator(DEVICE).manual_seed(SEED)
    with torch.no_grad():
        result = pipe(prompt=prompt, image=image, strength=0.8,
           guidance_scale=7.5)
    return result.images[0].convert("RGB")
```

Listing 3.10: Image generation during testing

3.4.4 Results Compilation

Generated images are saved into model-specific directories, and all evaluation metrics (prompt, category, reward score, and image path) are written into CSV files for later comparison. This structure enables ablation studies by making it easy to assess how different loss functions and adapters impact image quality and semantic alignment.

This testing pipeline provides a structured evaluation framework for all LoRA-tuned models. It generates visual outputs, assigns reward-based scores, and organizes results for direct comparison. The process is essential for validating which training strategies

(loss functions, adapters) best support generating tactile graphics that align with feedback instructions and accessibility standards.

3.4.5 Summary

The testing pipeline systematically benchmarks all LoRA-tuned models by generating images, scoring them with a CLIP reward model, and compiling results for comparison. This structured evaluation ensures that the most effective loss functions and training strategies are clearly identified.

3.5 Our Final Tuned InstructPix2pix Baseline

Our baseline uses the InstructPix2Pix pipeline to translate a *natural image* and a *text prompt* into a *tactile-style image* (Fig. 3.6). The model conditions denoising on both the prompt and the input image, following the original InstructPix2Pix formulation.

- **Inputs:** a natural (reference) image, tactile image and an instruction prompt (e.g., "create a tactile image of a bird").
- **Output:** a generated tactile image aligned with the prompt and the input visual content.

Architecture and training configuration

- **Backbone:** Stable Diffusion InstructPix2Pix with an 8-channel UNet input (4 for noisy latents + 4 for the conditioned image embedding).
- **Frozen components:** VAE and CLIP text encoder (`requires_grad_(False)`).
- **Fine-tuned component: UNet** only (all UNet parameters optimized).
- **Objective:** mean–squared error on the predicted noise (standard diffusion loss).
- **Conditioning:** text tokens from the prompt (via the frozen CLIP text encoder) and latent features of the input image (via the frozen VAE). Optional conditioning dropout is supported to enable classifier-free guidance at inference.

Implementation and Usage

Training uses triplets $\langle$input image, target image, edit prompt$\rangle$ provided via a `metadata.jsonl` file. Images are resized and normalized, and prompts are tokenized with the CLIP tokenizer.

At inference, the pipeline takes a new natural image and prompt, generating a tactile image through a single `img2img` pass with fixed guidance and step settings for consistent comparison across experiments.

This baseline isolates the capabilities of a strong instruction-guided `img2img` model *without* reward modeling or LoRA adapters, serving as a controlled reference point for later feedback-guided fine-tuning and adapter-based variants.

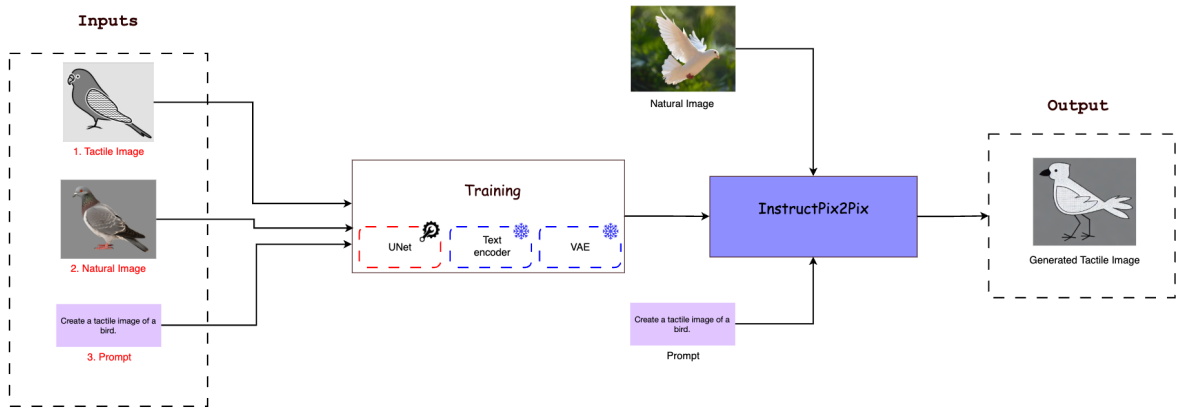


Figure 3.6: Project's Fine-Tuned Baseline Flow Diagram.

3.6 Our Final Adapter

We create a new adapter by fine-tuning Stable Diffusion with LoRA applied to both the UNet and the CLIP text encoder, while keeping the VAE frozen (Fig. 3.7). Compared to the earlier train.py pipeline, most dataset handling and training loop structure remain unchanged, and the hyperparameters that previously yielded the best results are reused. The main architectural change is that the text encoder is now unfrozen and modified to accept LoRA layers, allowing it to be trained jointly with the UNet. Training is guided by a modular reward suite—any of the four available reward models (CLIP, LPIPS, SSIM, Pixel) can be used individually or in combination, with weights controlling their influence. Data augmentation is adapted for each modality: the tactile image receives different transformations than the natural image, with the latter including saturation and Gaussian blur to enhance robustness.

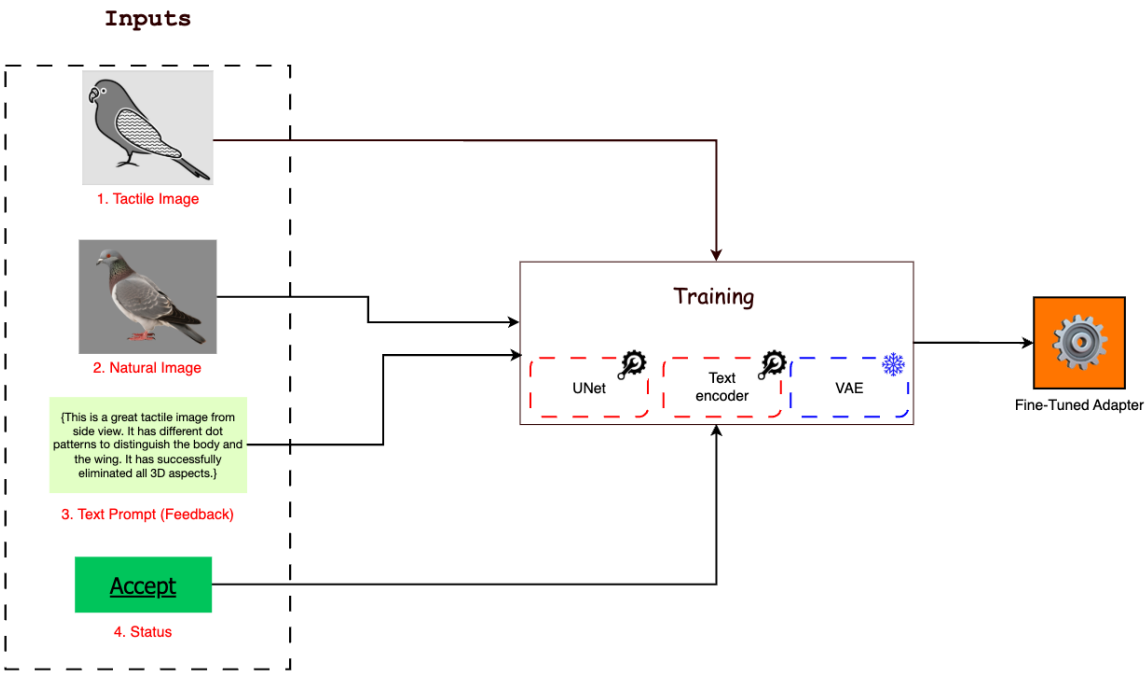


Figure 3.7: New Adapter Creation.

3.6.1 Inputs

- **Base model**: pre-trained Stable Diffusion (Img2Img) checkpoint.
- **Dataset**: The same JSONL file `training_dataset.jsonl` that was used in the initial training phase with records containing:
  - `input_image`: path to source (natural) image.
  - `generated_image`: path to target (tactile) image.
  - `feedback_text`: instruction/feedback string.
  - `rating`: `accept` / `reject`.
- **Training hyperparameters**: Best performing epochs, batch size, learning rate, scheduler from initial tests.
- **Reward configuration**: `REWARD_TYPE` (e.g., `clip+lpips`) and `REWARD_WEIGHTS`.

3.6.2 Outputs

- **LoRA adapters**:
  - `unet/`: UNet LoRA weights.
  - `text_encoder/`: CLIP text encoder LoRA weights.

- **Training log**: `results.jsonl` with *run name*, *loss type*, *reward type*, *epochs*, *final loss*, and timestamps.

3.6.3 Model Components and LoRA Setup

- **UNet**: prepared for LoRA (`prepare_model_for_kbit_training`), adapters attached to attention projections `to_q`, `to_k`, `to_v` across specified layers.
- **Text encoder (unfrozen)**: LoRA adapters attached to `q_proj`, `k_proj`, `v_proj`, `out_proj`. The text encoder is trainable in this method.
- **VAE**: frozen (`requires_grad_(False)`).

```
pipe.unet = prepare_unet_for_lora(pipe.unet)

lora_config_unet = LoraConfig(
    r=8, lora_alpha=32,
    target_modules=["to_q", "to_k", "to_v"],
    layers_to_transform=list(range(16)),
    lora_dropout=0.1, bias="none", fan_in_fan_out=False
)
pipe.unet = get_peft_model(pipe.unet, lora_config_unet)

lora_config_te = LoraConfig(
    r=8, lora_alpha=32,
    target_modules=["q_proj", "k_proj", "v_proj", "out_proj"],
    lora_dropout=0.1, bias="none", fan_in_fan_out=False
)
pipe.text_encoder = get_peft_model(pipe.text_encoder,
   lora_config_te)

pipe.vae.requires_grad_(False)          # freeze VAE
pipe.unet.train(); pipe.text_encoder.train()

# training loop (rewards, latents, denoising, loss, optimizer)

adapter_root = output_dir / "lora_adapters"
(pipe.unet).save_pretrained(str(adapter_root / "unet"))
(pipe.text_encoder).save_pretrained(str(adapter_root / "
   text_encoder"))
```

Listing 3.11: LoRA on UNet and Text Encoder; VAE frozen and adapters saved

3.6.4 Training Procedure

The training process begins with loading the paired natural and tactile images, applying modality-specific augmentations. Light transformations are applied to the tactile images, while additional saturation and Gaussian blur are applied to the natural images. Each sample is then assigned a reward score $r_i \in [0, 1]$ from the configured reward suite (CLIP similarity, $1 -$ LPIPS, SSIM, or pixel-based metrics). Both reference and target images are VAE-encoded into latents, with random noise added to the target latents according to a sampled diffusion timestep. Instructions are tokenized and processed by the trainable text encoder to produce conditioning embeddings, which guide the UNet in predicting the noise for denoising. The selected loss variant (e.g. reward-weighted MSE) uses r_i to scale the objective. Optimization updates only the LoRA parameters in the UNet and text encoder, while logging training metrics across epochs. Once training completes, the resulting LoRA adapters are saved for downstream evaluation and inference.

3.6.5 Testing Procedure

The testing phase aims to rigorously evaluate the performance, robustness, and generalization capabilities of the proposed feedback-guided LoRA adapter in comparison to the baseline Stable Diffusion model. Testing is conducted on a fixed, curated dataset, allowing reproducible comparison across different configurations. Two complementary components structure the evaluation: a *Custom Test Protocol*, designed specifically for tactile image generation tasks, and a *New Testing Setup* that enables systematic baseline–adapter comparisons under controlled conditions. Together, these components provide both targeted, qualitative insights and broad, quantitative performance metrics.

Custom Test Protocol

A tailored evaluation protocol was developed to rigorously assess the performance and generalization ability of the newly trained feedback-guided adapter for Stable Diffusion. Unlike generic benchmarking, this protocol was explicitly designed for the tactile image generation setting and integrates both seen and unseen examples, as well as controlled variations in prompts and reference images. The dataset contains 29 carefully curated entries spanning 14 classes, each including a natural image, its corresponding tactile image, a generation prompt, and a human rating. The protocol incorporates:

- **Held-out evaluation** for unseen examples with prompt adaptation to a baseline template.

- **Seen evaluation** using identical inputs from training to measure retention of learned mappings.
- **Targeted interpretation subset** selection for deeper qualitative inspection.
- **Prompt variation testing** to assess robustness to linguistic changes.
- **Reference variation testing** to evaluate sensitivity to visual changes.

This structured evaluation approach provides a reproducible framework for analyzing both fidelity and adaptability in feedback-driven tactile image generation systems.

New Testing Setup

The evaluation script is designed to systematically compare the baseline Stable Diffusion model with the newly trained LoRA adapter. It supports three test configurations, controlled by boolean flags:

- **baseline_only** – runs without any LoRA adapter.
- **adapter_only** – applies the LoRA adapter without merging weights.
- **both_together** – loads the LoRA adapter and merges its weights into both the UNet and the CLIP text encoder before testing.

This toggle system allows direct comparison of performance impacts from the adapter and the merge process. The merging step integrates LoRA-trained parameters into both the UNet and text encoder for potentially faster inference and unified weights during evaluation.

The setup of these configurations is defined as follows:

```
RUNS = {
    "baseline_only": {"use_lora": False, "merge": False, "
       lora_path": None},
    "adapter_only": {"use_lora": True,  "merge": False, "lora_path
       ": LORA_PATH},
    "both_together": {"use_lora": True, "merge": True,  "lora_path
       ": LORA_PATH},
}
```

Listing 3.12: Evaluation configuration toggles

For each configuration, the pipeline loads the appropriate weights, generates outputs for the fixed test set, and computes semantic alignment scores using a CLIP-based reward model. All images and scores are stored for later quantitative and qualitative comparison.

It loads the fixed test dataset that was mentioned earlier and generates tactile image outputs for each configuration using the same inference settings. A CLIP-based reward model then scores each output for semantic alignment with its prompt. Generated images and their scores are stored per configuration for later quantitative and qualitative analysis.

Chapter 4

Experimental Settings

This chapter outlines the experimental setup used to evaluate and refine the LoRA fine-tuned Stable Diffusion models for tactile graphic generation. The experiments were designed to explore how different training and testing parameters impact performance, with a focus on model quality, semantic alignment, and tactile readability.

4.1 Compute and Software Environment

All experiments were conducted on a local machine equipped with an **NVIDIA RTX 4090 GPU** with **24GB VRAM** and **CUDA version 12.2**. The system used **Python 3.10.16** and **PyTorch 2.7.0**.

The following major libraries were used:

- **Transformers** (v4.52.3): for CLIP and related model access
- **Diffusers** (v0.33.1): for Stable Diffusion and image generation workflows
- **Accelerate**, **PEFT**, **Bitsandbytes**: for efficient model loading and fine-tuning

All libraries were managed via a **Conda** environment.

4.2 Training Experiments

Initial Adapter Training

The `train.py` script was executed a total of 15 times to systematically train models across different settings. Five distinct loss functions were tested:

- Mse Loss
- Clip Weighted Mse
- Reward Weighted Mse
- Inverse Reward Loss
- Margin Reward Loss

Figure 4.1: 10-epochs models generated by "`train.py`".

Each loss function was trained for three different epoch counts (5, 8, and 10) using a fixed batch size of 4. This setup enabled evaluation of underfitting versus overfitting, with metrics including:

- Final training loss
- CLIP reward score
- Visual fidelity of generated images

New img2img Baseline Training

A new `InstructPix2Pix` img2img baseline was trained from the instruction-tuned Stable Diffusion model using paired natural–tactile image data. Two main variants were explored:

- **10k steps** training

- **15k steps** training

This allowed us to study whether extended training improves fidelity and semantic alignment without overfitting.

Final Adapter Training

The `train2.py` script was used to fine-tune the LoRA adapter jointly on the UNet and CLIP text encoder (VAE frozen) with reward-weighted objectives. Different reward weight distributions between **CLIP** and **LPIPS** were tested to determine their effect on output quality.

4.3 Testing Experiments

Initial Adapter Testing

Once training was completed, the "`test.py`" script was used to evaluate all LoRA fine-tuned models. Testing was performed three times, once for each trained epoch count (5, 8, and 10), to measure how the number of epochs affected model performance.

The evaluation used a test dataset containing 60 entries: 30 images that had been part of the training dataset used by "`train.py`," and 30 entirely unseen images that were not used during training. This split allowed assessment of both how well the models reproduced familiar examples and how well they generalized to new data.

Final Adapter & Baseline Testing

The `test2.py` script was used to evaluate:

1. **New Baseline Only:** Img2Img model without adapter.
2. **Adapter Only:** applying LoRA without merging weights.
3. **Merged Adapter + Baseline:** LoRA merged into both UNet and text encoder for faster inference.

The evaluation dataset contained 29 entries across 14 classes, each comprising a natural image, corresponding tactile image, generation prompt, and human rating. The protocol assessed performance on both seen and unseen examples, included a targeted high-quality subset for qualitative inspection, and introduced controlled variations in prompts and reference images to measure linguistic robustness and visual sensitivity.

4.4 Hyperparameters

To further explore the model behavior, additional testing runs varied key inference hyperparameters:

- **Guidance Scale:** Default value of 7.5 was adjusted to 5 and 10 (while keeping the other hyperparameters fixed) for the 10-epochs models.
- **Number of Inference Steps:** Default value of 30 was tested against 10 and 50 (while keeping the other hyperparameters fixed) for the 10-epochs models.

These controlled variations assessed how guidance strength and denoising depth influenced image quality, semantic alignment, and tactile readability.

Ablation Studies

Two ablation studies were conducted if using the best performing hyperparameters:

- **Reward Weight Impact:** varying CLIP vs LPIPS weights.
- **Extended Training:** 50 and 100 epochs at reduced learning rate to examine long-term training effects on stability, overfitting, and output quality.

Altogether, this experimental setup provides a comprehensive view of how training dynamics, loss design, and inference hyperparameters shape the effectiveness of Stable Diffusion for tactile graphic generation.

4.5 Model Selection

The baseline model used in initial experiments was **Stable Diffusion version 1.5**, a widely adopted general-purpose diffusion model. To better support instruction-based editing for tactile image generation, we replaced it with the **Instruction-Tuned Stable Diffusion model**. This version was fine-tuned on paired image–instruction datasets, enabling it to follow semantic editing prompts more reliably.

Chapter 5

Results and Evaluations

5.1 Overview of Evaluation

This chapter reports two evaluation phases. The first phase benchmarks five LoRA-fine-tuned models trained on the *original* Stable Diffusion v1.5 baseline using five loss functions (MSE, CLIP-Weighted MSE, Reward-Weighted MSE, Inverse Reward Loss, Margin Reward Loss). Models were trained for 5, 8, and 10 epochs and tested on a 60-entry set (30 seen, 30 unseen) to assess memorization vs. generalization. Performance was summarized with CLIP-based semantic alignment scores and qualitative inspection.

The second phase adopts a stronger *instruction-tuned* Stable Diffusion img2img baseline and evaluates both the baseline itself and the newly proposed LoRA adapter. A fixed test set of **29** entries spanning **14** classes is used; each entry includes a natural (reference) image, a corresponding tactile image, a generation prompt, and a human rating. This set also encodes controlled *prompt variations* (synonyms/feedback-guided refinements) and *reference-image variations* to probe robustness. Using consistent inference settings, we compare: (i) new baseline alone vs. new baseline + adapter, (ii) adapter variants trained with different reward weightings (CLIP/LPIPS and their combinations), and (iii) training durations (e.g., 10k vs. 15k img2img steps for the baseline, and extended runs such as 50/100 epochs for ablations). Results are reported with the same CLIP-based metric and targeted qualitative analysis.

5.2 Quantitative Results

To systematically evaluate model performance, quantitative experiments were carried out using the CLIP reward score as the primary metric. This score reflects how closely generated tactile graphics align with the provided textual prompts and serves as a consistent

benchmark across all tests. Results were aggregated for each LoRA-fine-tuned model under different training and testing conditions, including variations in epoch count, inference steps, and guidance scale. Together, these experiments provide a structured view of how different hyperparameters and loss functions influence output quality and alignment.

5.2.1 Effect of Number of Epochs

To evaluate the influence of the number of epochs on model performance, the "`test.py`" script was run on models trained with each loss function across three training durations: 5, 8, and 10 epochs. The resulting CLIP reward scores for each setting are summarized in Table 5.1.

Model	**5 Epochs**	**8 Epochs**	**10 Epochs**
Baseline	29.17	29.17	29.17
Margin Reward Loss	29.17	29.17	29.17
MSE	28.35	27.80	27.64
Clip-Weighted MSE	28.17	27.81	27.79
Reward-Weighted MSE	28.13	27.55	27.76
Inverse Reward Loss	22.64	20.93	21.57

Table 5.1: Average CLIP reward scores for all loss functions across 5, 8, and 10 epochs.

Observations and Trends

Several key trends emerge from comparing results across training durations:

- **Baseline and Margin Reward Loss remain stable.** The baseline model and the Margin Reward Loss model consistently achieved the same highest mean score (29.17) across all epochs, showing that fine-tuning with this loss did not harm semantic alignment.
- **Minimal improvement after 5 epochs.** For most loss functions, the 5-epoch models achieved slightly higher or similar scores compared to 8 and 10 epochs. For instance, MSE Loss started at **28.35** (5 epochs) but slightly declined to **27.64** (10 epochs).
- **Inverse Reward Loss consistently underperformed.** Across all epochs, Inverse Reward Loss produced the lowest scores, dropping from **22.64** (5 epochs) to **20.93** (8 epochs), showing that the approach negatively impacts alignment regardless of training duration.

- **Signs of overfitting beyond 5 epochs.** While 5-epoch results were generally strongest for MSE Loss, Clip-Weighted MSE, and Reward-Weighted MSE, slight declines at 8 and 10 epochs suggest that extending training did not lead to better generalization and may have caused mild overfitting.

Key Findings

- **5 epochs provided the best balance.** Shorter training preserved generalization and delivered the highest or near-highest CLIP scores for most loss functions.
- **Margin Reward Loss maintained strong performance across all epochs.** Unlike other losses, it neither improved nor degraded significantly, making it a stable choice for future work.
- **Longer training (8–10 epochs) did not yield major benefits.** Scores plateaued or declined slightly, particularly for MSE Loss and Reward-Weighted MSE.
- **Inverse Reward Loss should be avoided.** It was consistently the lowest-performing loss, regardless of training length.

5.2.2 Effect of Number of Inference Steps

To evaluate the influence of inference steps on model performance, the `"test.py"` script was run on the models trained for 10 epochs using three different inference step counts: 10, 30 (default), and 50. The CLIP reward scores for each setting are summarized in Table 5.2.

Model	**10 Steps**	**30 Steps**	**50 Steps**
Baseline	26.53	29.17	28.75
Margin Reward Loss	26.53	29.17	28.75
MSE	26.95	27.64	27.59
Clip-Weighted MSE	26.77	27.79	27.40
Reward-Weighted MSE	27.12	27.76	27.42
Inverse Reward Loss	21.80	21.57	21.54

Table 5.2: Average CLIP reward scores across different inference step counts for models trained for 10 epochs.

Observations

- **30 steps offered the best overall balance.** Models using the default 30-step setting consistently achieved the highest scores, particularly for the Baseline and Margin Reward Loss models (29.17).
- **Fewer steps (10) caused noticeable quality degradation.** All models saw a sharp drop when only 10 steps were used, with the Baseline and Margin Reward Loss scores falling from 29.17 (30 steps) to 26.53.
- **Increasing to 50 steps provided no major benefit.** Scores slightly declined or plateaued compared to the default 30-step setting, suggesting that additional steps did not translate into better alignment.
- **Inverse Reward Loss remained consistently low.** Regardless of the step count, Inverse Reward Loss produced the weakest results (around 21.5–21.8).

Key Takeaway

The default 30 inference steps strike the optimal balance between computational cost and image quality. Reducing the steps significantly degrades performance, while increasing beyond 30 yields no meaningful improvement.

5.2.3 Effect of Guidance Scale

The impact of guidance scale on model performance was also assessed using the 10-epochs models. The "`test.py`" script was run three times, each with a different guidance scale: 5, 7.5 (Default), and 10. The CLIP reward scores are shown in Table 5.3.

Model	Guidance 5	Guidance 7.5	Guidance 10
Baseline	29.10	29.17	29.13
Margin Reward Loss	29.10	29.17	29.13
MSE	27.08	27.64	27.86
Clip-Weighted MSE	26.99	27.79	27.73
Reward-Weighted MSE	27.05	27.76	27.69
Inverse Reward Loss	21.75	21.57	21.44

Table 5.3: Average CLIP reward scores for models trained for 10 epochs under different guidance scale values.

Observations

- **The baseline and margin loss models were largely unaffected.** Scores stayed almost flat across scales (29.1–29.2).
- **For most loss types, performance peaked around guidance 10.** MSE and Reward-Weighted MSE gained a small bump in score (27.86 and 27.69) at guidance 10.
- **Guidance 5 slightly reduced performance.** All models except Baseline and Margin Reward Loss saw marginal dips when guidance dropped to 5.
- **Inverse Reward Loss stayed weak across the board.** It consistently lagged, scoring around 21.4–21.8 regardless of scale.

Key Takeaway

Guidance scale adjustments had only subtle effects. The default 7.5 already provided balanced performance, while moving to 10 yielded minor gains for some loss functions (especially MSE), and reducing to 5 slightly degraded output alignment.

5.2.4 Different Baseline Models

Table 5.4 compares the performance of models trained with different loss functions using two baseline variants: Stable Diffusion v1.5 and the instruction-tuned SD model.

Table 5.4: CLIP scores across different loss functions using two baseline models.

Model	Stable Diffusion v1.5	Instruction-tuned SD
Baseline	29.17	30.27
Margin Reward Loss	29.17	30.27
CLIP Weighted MSE	27.79	30.29
Reward Weighted MSE	27.76	30.37
MSE	27.64	30.20
Inverse Reward Loss	21.57	30.87

Observations

- The instruction-tuned SD baseline consistently outperforms the Stable Diffusion v1.5 across all loss functions.

- The inverse reward loss shows the largest improvement when switching to the instruction-tuned SD model, jumping from 21.57 to 30.87.
- Loss functions that performed moderately on the original baseline (e.g., CLIP Weighted MSE, Reward Weighted MSE) achieve significantly better results on the instruction-tuned model.

Key Takeaway

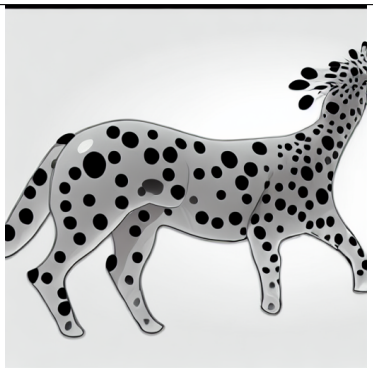
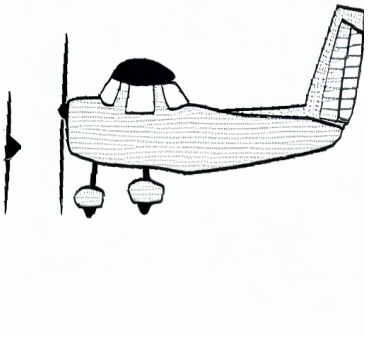
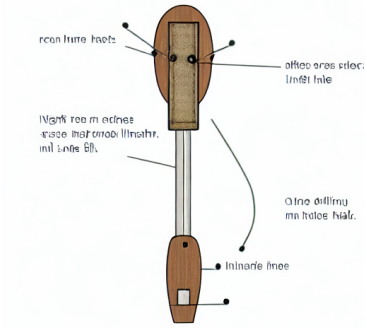
This indicates that a stronger base model allows the learning signal from reward-based losses to be better utilized.

5.3 Qualitative Observations

The qualitative results in Table 5.5 demonstrate a clear advantage of the Instruction-Tuned SD model over the standard Stable Diffusion v1.5. While the baseline model often introduces artifacts or semantically unrelated content, such as animal patterns or annotated diagrams, the instruction-tuned model generates outputs that better preserve the semantics and structural characteristics of the original images. This makes it a more appropriate foundation for tactile representation tasks.

Building on these observations, we now present results obtained with the new instruction-tuned img2img baseline under different training regimes, as well as its combination with the proposed LoRA adapter. These experiments focus on qualitative analysis, given the improved semantic alignment and structural fidelity of the new baseline.

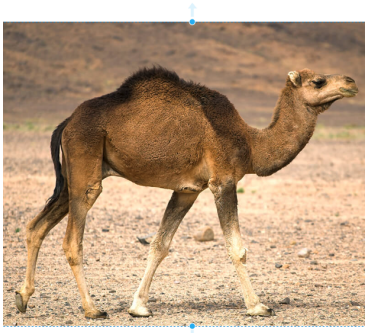
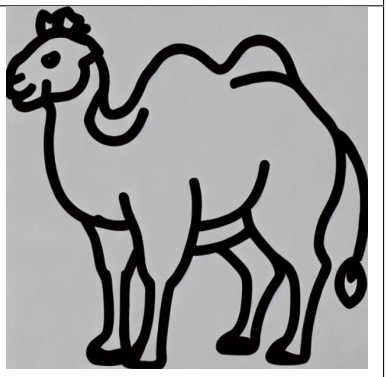
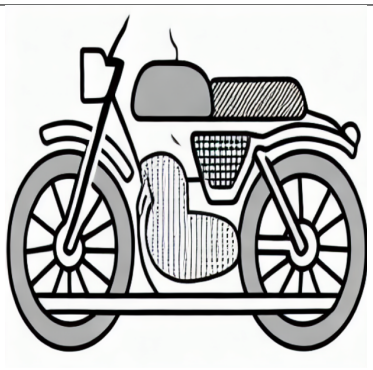
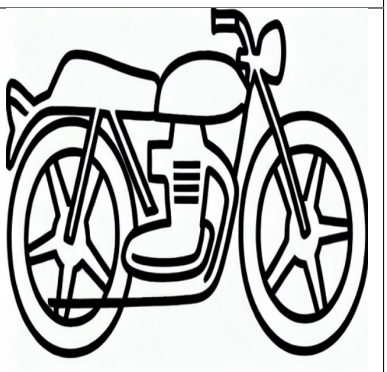
Table 5.5: Qualitative comparison between the original image, Stable Diffusion v1.5, and Instruction-Tuned SD model.

Natural Image	Stable Diffusion v1.5 Generated Image	Instruction-Tuned SD Generated Image
		
		

5.3.1 Effect of Training Steps for Instruct-Tuned Img2Img Baseline (10k vs 15k)

As shown in Table 5.6, the 15k-step baseline provides clearer semantic fidelity and finer structural details than the 10k-step counterpart. For example, in the *camel* class, the 15k model better preserves species-specific traits and local contours, whereas the 10k model exhibits simplified outlines. In the *motorcycle* class, the 10k model fails to capture key texture cues and the reference posture, while the 15k model more faithfully reflects both the surface patterning and overall geometry. Overall, increasing training to 15k steps yields qualitatively closer matches to the source images before adapter integration.

Table 5.6: Qualitative comparison between the original image, 15k steps, and 10k steps of Instruction-Tuned SD model.

Natural Image	Trained for 15k steps	Trained for 10k steps
		
		

5.3.2 Effect of Adding Final Adapter to Instruct-Tuned Img2Img Baseline

We evaluate the qualitative effect of adding the LoRA adapter to the instruction-tuned img2img baseline. Across both prompt modifications and changes in the reference image, the adapter improves semantic faithfulness.

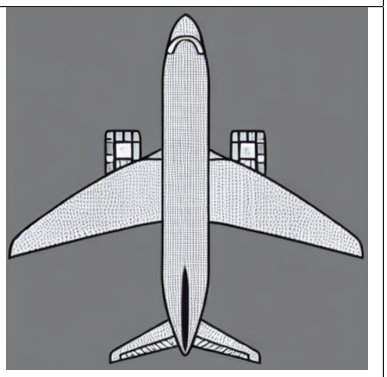
Case 1: Changing the prompt (Table 5.7)

- **Hut**: The adapter produces clearer separation between roof and walls, with thinner roof-texture strokes and a door rendered as an outline only, better matching the modified prompt.
- **Airplane**: The adapter sharpens the body/tail alignment, enforces single triangular wings, and renders engines as solid filled shapes, reducing spurious contours seen in the baseline.

Case 2: Changing the reference image (Table 5.8)

- **Hut**: When the reference image changes, the adapter preserves correct proportions and texture separation, producing more consistent roof patterns and door outlines despite variations in the source photo.
- **Watch**: The adapter corrects the distorted hand positions present in the baseline, while preserving the overall posture of the reference image and the casing shape. However, the generated image introduces some noise and extra details.

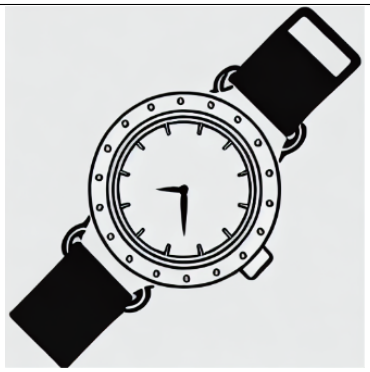
Table 5.7: Qualitative comparison between the original image, baseline only, and after adding the adapter based on modified prompts.

Natural Image	Baseline Only	Added Adapter
		
		

Modified Prompts:

Hut Class – Turn the natural image of a hut into a tactile-style drawing with a distinct base and top, using thinner lines for texture differentiation, and show the door only as an outline with no doorknob.
Airplane Class – Render edges and parts for tactile recognition of an airplane on a white background, with a distinct body and tail, triangular single main wings on each side, and solid black-filled engines.

Table 5.8: Qualitative comparison between the original image, baseline only, and after adding the adapter based on having different reference image.

Natural Image	Baseline Only	Added Adapter
		
		

5.3.3 Effect of Reward Weighting on Adapter Performance

We compared CLIP–LPIPS reward weighting schemes to determine which yields the best *tactile* output quality: crisp contours, correct part geometry, and high legibility. As shown in Fig. 5.1, the airplane class, the CLIP-heavy setting (0.8 CLIP) produces the *crispest and most legible* tactile rendering: clean silhouettes, reduced extraneous strokes, and better preservation of engine/body geometry. The balanced setting shows a slightly more accurate cockpit window but introduces extra line noise on the engines. Increasing LPIPS weight does not confer a clear advantage in our setting; it tends to thicken outlines, add spurious contours, and even drifts part shapes (e.g., engine mismatch). For the hut class, the balanced setting (0.6 CLIP / 0.4 LPIPS) yields the most faithful structure of the grass roof and respects the prompt requirement to render the door only as an outline with no doorknob. Higher LPIPS weight adds unnecessary thickness and visual clutter, while the CLIP-heavy setting simplifies details but slightly loses roof structure fidelity. We therefore adopt the 0.6 CLIP/ 0.4 LPIPS weighting in our adapter.

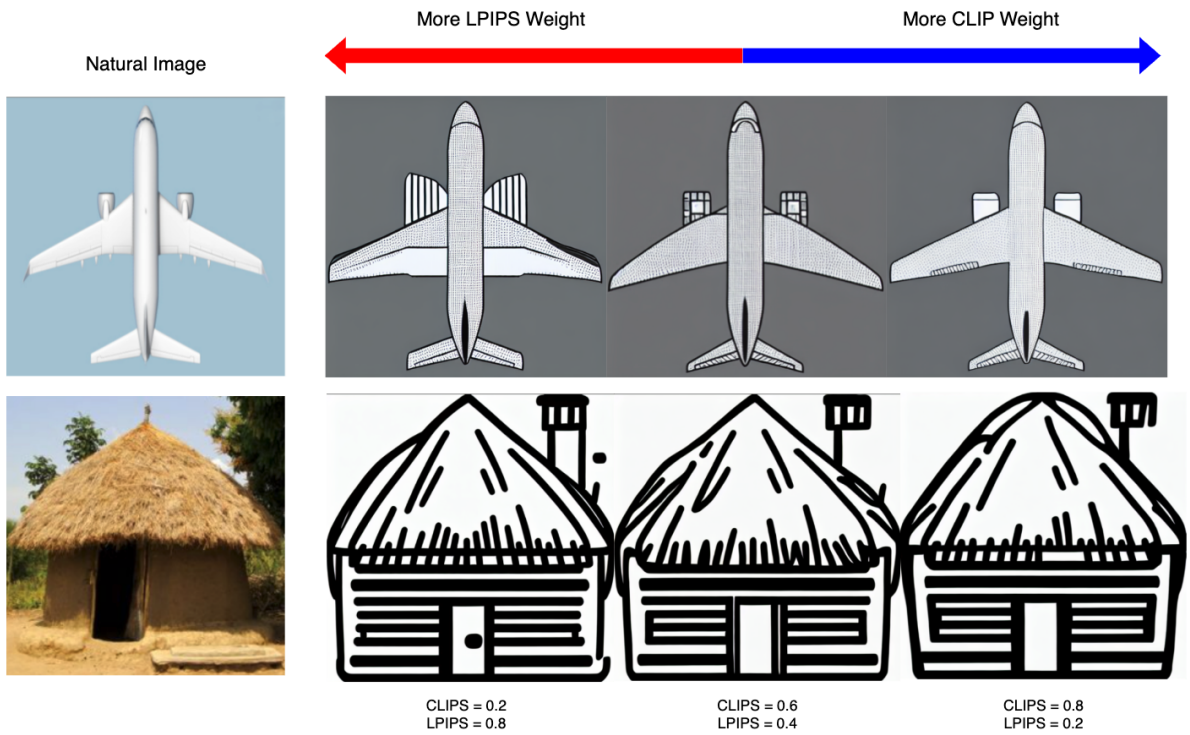


Figure 5.1: Qualitative comparison under different CLIP–LPIPS reward weightings.

5.3.4 Extended Training Ablation

We examined adapter training beyond the default schedule by continuing for 50 and 100 epochs at a reduced learning rate. As shown in Fig. 5.2, the 30-epoch model produces the cleanest, most stable outlines. Pushing to *50 epochs* slightly smooths interior dot textures but degrades global geometry (e.g., wing proportions and engine shapes become more blocky), and some contours soften. At *100 epochs* clear signs of overfitting appear: spurious structures emerge (e.g., a dorsal intake-like shape), outlines thicken, and part alignment drifts (wing and engine misplacement). Overall, extended training yielded no qualitative gains and instead increased artifacts and semantic drift.

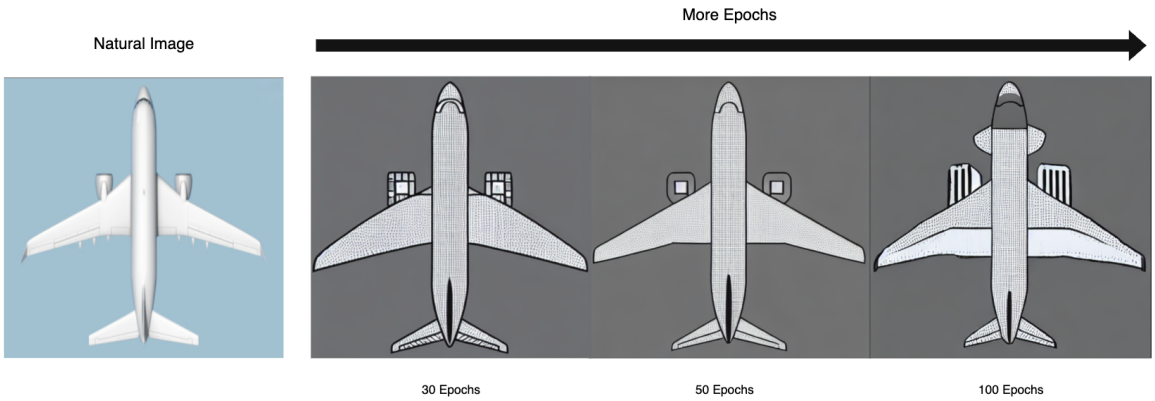


Figure 5.2: Effect of prolonged adapter training (30 vs. 50 vs. 100 epochs) on a fixed airplane example.

Chapter 6

Summary and Conclusion

This project advanced the TactileNet framework by addressing key limitations in dataset size, training strategy, and evaluation methodology. The dataset was expanded by a factor of ten and restructured with standardized feedback aligned to BANA guidelines, providing a more consistent and reliable foundation for model learning. A reward-guided training pipeline was implemented using Stable Diffusion fine-tuned with LoRA adapters, integrating multiple loss functions and forward/backward diffusion processes.

Extensive experiments compared training durations, loss types, and inference hyperparameters, revealing how these factors influence tactile image quality, CLIP reward scores, and structural readability. The instruction-tuned img2img Stable Diffusion baseline consistently outperformed the original Stable Diffusion v1.5 across most configurations, particularly when paired with reward-weighted objectives. Adapter integration improved semantic alignment and contour clarity in many cases. However, optimal results required careful balancing of reward weights (0.6 CLIP / 0.4 LPIPS) and restraint in training duration to avoid overfitting or semantic drift.

6.1 Limitations

A central limitation of this work is **data scarcity**. The available tactile graphics dataset remains small in both scale and class diversity, which restricts the model's exposure to varied structural patterns and hinders its ability to generalize beyond the seen classes. This bottleneck cascades into multiple issues, such as overfitting to frequent patterns, poor handling of rare objects, and limited robustness to prompt or input variations. Another limitation lies in the heavy reliance on CLIP's similarity score as a fixed, frozen proxy reward. This metric, computed as the cosine similarity between CLIP's vision and text encoder embeddings, offers only a coarse measure of image–text alignment. It does

not evaluate tactile quality, structural clarity, or compliance with BANA guidelines, and it cannot adapt to the tactile graphics domain over time. Consequently, images that scored well under CLIP were sometimes unsuitable for tactile interpretation, revealing its role as a weak oracle rather than a reliable evaluator of tactile graphic quality.

6.2 Future Work

Potential future work includes expanding the dataset to increase coverage and class diversity, thereby improving model robustness in the tactile graphics domain. Preference-based reinforcement learning optimization techniques could be explored, with this dataset serving as an initial step to create paired examples for fine-tuning. Furthermore, off-the-shelf models such as ChatGPT [ChatGPT, 2025] and Google Gemini [Gemini, 2025] could be employed to automatically generate edited versions of images based on the feedback text in the dataset, enabling scalable data augmentation and more effective alignment with tactile design requirements.

Bibliography

American Printing House for the Blind (APH), *The Tactile Graphics Image Library*, 2024. https://imagelibrary.aph.org/

Braille Authority of North America (BANA), *Braille Authority of North America*, 2025. https://www.brailleauthority.org/

Bertolotto, S., Panisson, A., & Perotti, A. (2021). LLM-Generated Class Descriptions for Semantically Meaningful Image Classification. In *Proceedings of the 1st International Conference on Explainable AI Neural Symbolic Methods (EXPLAINS)* (pp. 50–61). Porto, Portugal.

Brooks, T., Holynski, A., & Efros, A. A. (2023). InstructPix2Pix: Learning to follow image editing instructions. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)* (pp. 18392–18402).

Chen, Y., Lai, Y. K., & Liu, Y. J. (2018). CartoonGAN: Generative adversarial networks for photo cartoonization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 9465–9474).

Daniel M. Ziegler, Nisan Stiennon, Jeffrey Wu, Tom B. Brown, Alec Radford, Dario Amodei, Paul Christiano, and Geoffrey Irving, *Fine-tuning language models from human preferences*, arXiv preprint arXiv:1909.08593, 2019.

Dot Pad: The First Smart Tactile Graphics Display. Retrieved August 2, 2025, from https://www.visionaid.co.uk/dot-pad

Dzhurynskyi, Y., Mayik, V., & Mayik, L. (2024). Enhancing accessibility: Automated tactile graphics generation for individuals with visual impairments. *Computation, 12*(12), 251.

Edman, P. (1992). *Tactile graphics.* American Foundation for the Blind.

Google, *Gemini*, 2025. `https://gemini.google.com/app`

Jason Wei, Maarten Bosma, Vincent Y. Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M. Dai, and Quoc V. Le, *Finetuned language models are zero-shot learners*, arXiv preprint arXiv:2109.01652, 2021.

Juan Rocamonde, Victoriano Montesinos, Elvis Nava, Ethan Perez, and David Lindner, *Vision-Language Models are Zero-Shot Reward Models for Reinforcement Learning*, arXiv preprint arXiv:2310.12921, 2024.

Khan, A., Choubineh, A., Shaaban, M.A., Akkasi, A., & Komeili, M. (2025). *TactileNet: Bridging the Accessibility Gap with AI-Generated Tactile Graphics for Individuals with Vision Impairment.*

Lindstrom, P., & Turk, G. (2000). Image-driven simplification. *ACM Transactions on Graphics (TOG), 19*(3), 204–241.

OpenAI, *ChatGPT*, 2025. `https://chat.openai.com/`

Pakenaite, K., Kamperou, E., Proulx, M. J., Sharma, A., & Hall, P. (2024, February). Pic2Tac: Creating accessible tactile images using semantic information from photographs. In *Proceedings of the Eighteenth International Conference on Tangible, Embedded, and Embodied Interaction* (pp. 1–12).

Paul F. Christiano, Jan Leike, Tom Brown, Miljan Martic, Shane Legg, and Dario Amodei, *Deep reinforcement learning from human preferences*, Advances in Neural Information Processing Systems, 30, 2017.

Paul, S. (2023). Instruction-tuning Stable Diffusion with InstructPix2Pix. *Hugging Face Blog.* Retrieved from `https://huggingface.co/blog/instruction-tuning-sd`.

Perkins School for the Blind, *Tactile Graphics Library*, 2025. `https://www.perkins.org/resource/tactile-graphics-library/`

Ricci, R., Bazi, Y., & Melgani, F. (2024). Machine-to-Machine Visual Dialoguing with ChatGPT for Enriched Textual Image Description. *Remote Sensing, 16*(3), 441. `https://doi.org/10.3390/rs16030441`.

Rosenblum, L. P., & Herzberg, T. S. (2015). Braille and tactile graphics: Youths with visual impairments share their experiences. *Journal of Visual Impairment & Blindness, 109*(3), 173–184.

Shagidanov, A., Poghosyan, H., Gong, X., Wang, Z., Navasardyan, S., & Shi, H. (2024, April). Grounded-Instruct-Pix2Pix: Improving instruction-based image editing with automatic target grounding. In *ICASSP 2024 – IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (pp. 6585–6589). IEEE.

Tanaka, K., Fushimi, T., Tsutsui, A., & Ochiai, Y. (2023). Text to haptics: Method and case studies of designing tactile graphics for inclusive tactile picture books by digital fabrication and generative AI. In *ACM SIGGRAPH 2023 Labs* (pp. 1–2).

Wang, X., & Gupta, A. (2016). Generative image modeling using style and structure adversarial networks. In *European Conference on Computer Vision* (pp. 318–335). Cham: Springer International Publishing.

Wu, H., Yang, H., Chang, F., Zhu, D., & Liu, Z. (2025). AI-generated tactile graphics for visually impaired children: A usability study of a multimodal educational product. *International Journal of Human-Computer Studies*, 103525.

Zhang, L., Rao, A., & Agrawala, M. (2023). Adding Conditional Control to Text-to-Image Diffusion Models. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)* (pp. 3836–3847).

Zeinullin, M. (2025). *Improving exploration of tactile graphics by visually impaired people: Theoretical advances and a novel mobile application* (Doctoral dissertation).

Zhao, X., Yu, H., & Bian, H. (2023). Image to Image Translation Based on Differential Image Pix2Pix Model. *Computers, Materials & Continua, 77*(1).

Appendix A

Source Code

"train.py"

```
import torch
from torchvision import transforms
import json
from pathlib import Path
from typing import List, Dict
from PIL import Image
from datasets import Dataset, Features, Image as ImageFeature,
    Value
from transformers import CLIPProcessor, CLIPModel, get_scheduler
from diffusers import StableDiffusionImg2ImgPipeline
from peft import LoraConfig, prepare_model_for_kbit_training,
    get_peft_model
from torch.nn import functional as F
from torch.utils.data import DataLoader
import wandb
from tqdm import tqdm
import random
from torchvision.transforms import Compose, ToTensor, Normalize
import time
from datetime import datetime
import os

 # train.py
# Config
MODEL_PATH = "runwayml/stable-diffusion-v1-5"
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
```

```
25 BATCH_SIZE = 4
26 IMAGE_SIZE = (512, 512)
27 NUM_EPOCHS = 10
28
29 wandb.init(project="tactile-dpo-training")
30
31 # ----------------------
32 # 1. Load Dataset
33 # ----------------------
34 def load_image_pairs(jsonl_path: Path) -> List[Dict]:
35     pairs = []
36     with open(jsonl_path) as f:
37         for line in f:
38             try:
39                 data = json.loads(line)
40                 rating = 1 if data["rating"].lower() == "accept"
                       else 0
41                 input_img_path = Path(data["input_image"])
42                 gen_img_path = Path(data["generated_image"])
43
44                 if not input_img_path.exists() or not gen_img_path
                       .exists():
45                     continue
46
47                 pairs.append({
48                     "input_image": Image.open(input_img_path).
                           convert("RGB").resize(IMAGE_SIZE),
49                     "generated_image": Image.open(gen_img_path).
                           convert("RGB").resize(IMAGE_SIZE),
50                     "instruction": data["feedback_text"] if data["
                           feedback_text"].lower() != "nan" else "",
51                     "rating": rating
52                 })
53             except Exception as e:
54                 print(f"Error: {e}")
55     return pairs
56
57 dataset_features = Features({
58     "input_image": ImageFeature(),
59     "generated_image": ImageFeature(),
60     "instruction": Value("string"),
```

```
    "rating": Value("int32")
})

vae_preprocess = Compose([
    ToTensor(),
    Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
])
# ----------------------
# 2. Reward Model
# ----------------------
class CLIPRewardModel:
    def __init__(self):
        self.model = CLIPModel.from_pretrained("openai/clip-vit-
            base-patch32").to(DEVICE)
        self.processor = CLIPProcessor.from_pretrained("openai/
            clip-vit-base-patch32")

    def calculate_reward(self, images: List[Image.Image], texts:
        List[str]) -> torch.Tensor:
        inputs = self.processor(text=texts, images=images,
            return_tensors="pt", padding=True, truncation=True).to(
            DEVICE)
        with torch.no_grad():
            outputs = self.model(**inputs)
            sims = outputs.logits_per_image
            diagonal = sims.diagonal()
            return diagonal

reward_model = CLIPRewardModel()

augment = transforms.Compose([
    transforms.RandomHorizontalFlip(),
    transforms.ColorJitter(0.1, 0.1, 0.1),
])

def collate_fn(batch: List[Dict]) -> Dict:
    augmented = []
    for item in batch:
        augmented.append(item)
        augmented.append({
            "input_image": augment(item["input_image"]),
```

```
97              "generated_image": augment(item["generated_image"]),
98              "instruction": item["instruction"],
99              "rating": item["rating"]
100         })
101
102     rewards = reward_model.calculate_reward(
103         [x["generated_image"] for x in augmented],
104         [x["instruction"] for x in augmented]
105     )
106
107     return {
108         "input_images": [x["input_image"] for x in augmented],
109         "generated_images": [x["generated_image"] for x in
              augmented],
110         "instructions": [x["instruction"] for x in augmented],
111         "ratings": [x["rating"] for x in augmented],
112         "rewards": rewards.squeeze().detach().cpu()
113     }
114
115 # ----------------------
116 # 3. Custom Training Loop
117 # ----------------------
118 def prepare_unet_for_lora(unet):
119     return prepare_model_for_kbit_training(unet)
120
121 # ----------------------
122 # Loss Variants
123 # ----------------------
124 def mse_loss_fn(pred, target, **kwargs):
125     return F.mse_loss(pred, target)
126
127 def clip_weighted_mse_loss(pred, target, ratings, **kwargs):
128     accepted_mask = torch.tensor(ratings, device=DEVICE, dtype=
          torch.bool)
129     loss_weights = torch.where(
130         accepted_mask,
131         torch.tensor(0.5, device=DEVICE),
132         torch.tensor(2.0, device=DEVICE)
133     )
134     base_loss = F.mse_loss(pred, target)
135     return base_loss * loss_weights.mean()
```

```
136
137
138 def reward_weighted_mse_loss(pred, target, rewards=None, **kwargs)
       :
139     rewards = rewards.to(device=pred.device, dtype=pred.dtype)  #
           <- FIX: move to same device
140     base_loss = F.mse_loss(pred, target, reduction='none')  #
           shape: [B, C, H, W]
141     weight = rewards.reshape(-1, 1, 1, 1)  # broadcast shape
142     return (base_loss * weight).mean()
143
144 def inverse_reward_loss(pred, target, rewards=None, **kwargs):
145     rewards = rewards.to(device=pred.device, dtype=pred.dtype)
146     base_loss = F.mse_loss(pred, target, reduction='none')
147     inverse_weight = (1.0 - rewards).unsqueeze(1).unsqueeze(2).
           unsqueeze(3)
148     return (base_loss * inverse_weight).mean()
149
150 def margin_reward_loss(pred, target, rewards=None, **kwargs):
151     rewards = rewards.to(device=pred.device, dtype=pred.dtype)
152     base_loss = F.mse_loss(pred, target, reduction='none')
153     margin = 0.2
154     clipped = torch.clamp(margin - rewards.unsqueeze(1).unsqueeze
           (2).unsqueeze(3), min=0)
155     return (base_loss * clipped).mean()
156
157 # Update map:
158 loss_fn_map = {
159     "mse": mse_loss_fn,
160     "clip_weighted_mse": clip_weighted_mse_loss,
161     "reward_weighted_mse": reward_weighted_mse_loss,
162     "inverse_reward_loss": inverse_reward_loss,
163     "margin_reward_loss": margin_reward_loss
164 }
165
166 def custom_train_loop(pipe, dataset, reward_model, run_name,
       loss_type="clip_weighted_mse", num_epochs=NUM_EPOCHS):
167     dataloader = DataLoader(dataset, batch_size=BATCH_SIZE,
           shuffle=True, collate_fn=collate_fn)
168     optimizer = torch.optim.AdamW(pipe.unet.parameters(), lr=1e-5)
```

```
169     scheduler = get_scheduler("cosine", optimizer,
           num_warmup_steps=10, num_training_steps=len(dataloader) *
           num_epochs)
170     pipe.unet.train()
171
172     start_time = time.time()  # Start timing
173     for epoch in range(num_epochs):
174         for batch in tqdm(dataloader, desc=f"Epoch {epoch+1}/{
               num_epochs}"):
175             input_images = [vae_preprocess(img).to(device=DEVICE,
                   dtype=torch.float32) for img in batch["input_images
                   "]]
176             input_images = torch.stack(input_images)  # shape: [B,
                    3, 512, 512]
177             with torch.no_grad():
178                 latents = pipe.vae.encode(input_images).
                       latent_dist.sample()
179                 latents = latents * 0.18215
180             latents = latents.to(dtype=torch.float32)
181             timesteps = torch.randint(
182                 0,
183                 pipe.scheduler.config.num_train_timesteps,
184                 (latents.shape[0],),
185                 device=DEVICE,
186                 dtype=torch.long
187             )
188             noise = torch.randn_like(latents).to(dtype=torch.
                   float32, device=DEVICE)
189             latents = latents.to(device=DEVICE, dtype=torch.
                   float32)
190             noisy_latents = pipe.scheduler.add_noise(latents,
                   noise, timesteps)
191             text_inputs = pipe.tokenizer(batch["instructions"],
                   padding="max_length", max_length=pipe.tokenizer.
                   model_max_length, truncation=True, return_tensors="
                   pt").to(DEVICE)
192             with torch.no_grad():
193                 encoder_hidden_states = pipe.text_encoder(**
                       text_inputs).last_hidden_state
194             encoder_hidden_states = encoder_hidden_states.to(
                   device=DEVICE,dtype=torch.float32)
```

```
195             noise_pred = pipe.unet(
196                 noisy_latents,
197                 timesteps,
198                 encoder_hidden_states=encoder_hidden_states
199             ).sample
200             loss_fn = loss_fn_map[loss_type]
201             loss_fn = loss_fn_map[loss_type]
202 
203             # Dynamically check if loss function takes "rewards"
204             loss_kwargs = {}
205             if "rewards" in loss_fn.__code__.co_varnames:
206                 loss_kwargs["rewards"] = batch["rewards"]
207             if "ratings" in loss_fn.__code__.co_varnames:
208                 loss_kwargs["ratings"] = batch["ratings"]
209             final_loss = loss_fn(noise_pred, noise, **loss_kwargs)
210             final_loss.backward()
211             optimizer.step()
212             optimizer.zero_grad()
213             scheduler.step()
214             wandb.log({"loss": final_loss.item()})
215 
216     end_time = time.time()  # End timing
217     total_time = end_time - start_time
218     output_dir = Path(f"./outputs_10_epochs/{run_name}") #Changing
            the folder's name for each epoch set
219     output_dir.mkdir(parents=True, exist_ok=True)
220     adapter_path = output_dir / "./lora_adapters"
221     pipe.unet.save_pretrained(str(adapter_path))
222 
223     # Save training metadata
224     result_log = {
225         "run_name": run_name,
226         "loss_type": loss_type,
227         "num_epochs": num_epochs,
228         "total_time_sec": round(total_time, 2),
229         "final_loss": final_loss.item(),
230         "timestamp": datetime.now().isoformat()
231     }
232 
233     with open(output_dir / "results.jsonl", "a") as f:
234         f.write(json.dumps(result_log) + "\n")
```

```
235
236     print(f"Training completed. Adapter saved to {adapter_path}")
237     print(f"Log written to {output_dir / 'results.jsonl'}")
238
239 # ----------------------
240 # 4. Main
241 # ----------------------
242 def main():
243     run_name = input("Enter run name (no spaces): ").strip()
244     jsonl_path = Path("./training_dataset.jsonl")
245     pairs = load_image_pairs(jsonl_path)
246     if not pairs:
247         raise ValueError("No valid pairs found.")
248     dataset = Dataset.from_list(pairs, features=dataset_features)
249     pipe = StableDiffusionImg2ImgPipeline.from_pretrained(
250         MODEL_PATH,
251         torch_dtype=torch.float32
252     ).to(DEVICE)
253
254     pipe.unet = prepare_unet_for_lora(pipe.unet)
255     lora_config = LoraConfig(
256         r=8,
257         lora_alpha=32,
258         target_modules=["to_q", "to_k", "to_v"],
259         layers_to_transform=list(range(16)),
260         lora_dropout=0.1,
261         bias="none",
262         fan_in_fan_out=False
263     )
264     pipe.unet = get_peft_model(pipe.unet, lora_config)
265     pipe.unet = pipe.unet.to(device=DEVICE, dtype=torch.float32)
266     pipe.vae.requires_grad_(False)
267     pipe.text_encoder.requires_grad_(False)
268     loss_type = "margin_reward_loss"  # change loss function here.
269     # options: "mse", "clip_weighted_mse", "reward_weighted_mse",
        "inverse_reward_loss", "margin_reward_loss"
270     custom_train_loop(pipe, dataset, reward_model, run_name=
        run_name, loss_type=loss_type)
271
272 if __name__ == "__main__":
273     main()
```

Listing A.1: train.py

"test.py"

```
import torch
from diffusers import StableDiffusionImg2ImgPipeline
from peft import PeftModel
from transformers import CLIPProcessor, CLIPModel
from PIL import Image
from pathlib import Path
import json
import csv
import os
from tqdm import tqdm

# test.py

# --- Config ---
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
IMAGE_SIZE = (512, 512)
SEED = 42

torch.manual_seed(SEED)

# --- Paths ---
MODELS = {
    "baseline": None,
    "mse": "/home/student/khan/youssif/reward_model/
       outputs_10_epochs/mse_e10/lora_adapters",
    "clip_weighted_mse": "/home/student/khan/youssif/reward_model/
       outputs_10_epochs/clip_weighted_mse_e10/lora_adapters",
    "reward_weighted_mse": "/home/student/khan/youssif/
       reward_model/outputs_10_epochs/reward_weighted_mse_e10/
       lora_adapters",
    "inverse_reward_loss": "/home/student/khan/youssif/
       reward_model/outputs_10_epochs/inverse_reward_loss_e10/
       lora_adapters",
    "margin_reward_loss_10": "/home/student/khan/youssif/
       reward_model/outputs_10_epochs/margin_reward_loss_e10/
```

```
        lora_adapters"
}

MODEL_PATH = ""
TEST_JSONL = "./test_dataset.jsonl"
OUTPUT_DIR = Path("./misc_test")
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)

# --- CLIP Reward Model ---
class CLIPRewardModel:
    def __init__(self):
        self.model = CLIPModel.from_pretrained("openai/clip-vit-
            base-patch32").to(DEVICE)
        self.processor = CLIPProcessor.from_pretrained("openai/
            clip-vit-base-patch32")

    def score(self, images, texts):
        inputs = self.processor(text=texts, images=images,
            return_tensors="pt", padding=True, truncation=True).to(
            DEVICE)
        with torch.no_grad():
            outputs = self.model(**inputs)
        return outputs.logits_per_image.squeeze().tolist()

# --- Load Test Data ---
def load_test_set(path):
    samples = []
    with open(path) as f:
        for line in f:
            data = json.loads(line)
            samples.append({
                "input_image": data["input_image"],
                "prompt": data["feedback_text"],
                "category": data.get("Category", "unknown")
            })
    return samples

# --- Inference Function ---
def generate_image(pipe, image_path, prompt):
    image = Image.open(image_path).convert("RGB").resize(
        IMAGE_SIZE)
```

```
64      generator = torch.Generator(DEVICE).manual_seed(SEED)
65      with torch.no_grad():
66          result = pipe(prompt=prompt, image=image, strength=0.8,
               guidance_scale=7.5, num_inference_steps=30, generator=
               generator)
67      return result.images[0].convert("RGB")
68
69  # --- Main Evaluation ---
70  def main():
71      test_samples = load_test_set(TEST_JSONL)
72      scorer = CLIPRewardModel()
73
74      for name, lora_path in MODELS.items():
75          print(f"\nEvaluating model: {name}")
76          model_output_dir = OUTPUT_DIR / name
77          model_output_dir.mkdir(exist_ok=True)
78
79          pipe = StableDiffusionImg2ImgPipeline.from_pretrained(
               MODEL_PATH, torch_dtype=torch.float32).to(DEVICE)
80          if lora_path:
81              pipe.unet = PeftModel.from_pretrained(pipe.unet,
                   lora_path).to(DEVICE)
82          pipe.unet.eval()
83          pipe.vae.eval()
84          pipe.text_encoder.eval()
85
86          results = []
87
88          for idx, sample in enumerate(tqdm(test_samples)):
89              try:
90                  gen_img = generate_image(pipe, sample["input_image
                       "], sample["prompt"])
91                  score = scorer.score([gen_img], [sample["prompt"
                       ]])
92                  out_path = model_output_dir / f"{idx:04d}.png"
93                  gen_img.save(out_path)
94                  results.append({
95                      "idx": idx,
96                      "category": sample["category"],
97                      "prompt": sample["prompt"],
98                      "score": score,
```

```
                    "image_path": str(out_path)
                })
            except Exception as e:
                print(f"Error on sample {idx}: {e}")

        # Save CSV
        csv_path = model_output_dir / "results.csv"
        with open(csv_path, "w", newline="") as csvfile:
            writer = csv.DictWriter(csvfile, fieldnames=results
                [0].keys())
            writer.writeheader()
            writer.writerows(results)

if __name__ == "__main__":
    main()
```

Listing A.2: test.py

"analyzer.py"

```
import pandas as pd
import os
from pathlib import Path

# analyzer.py
# Path to the directory where model outputs are stored
OUTPUT_DIR = Path("//home/student/khan/youssif/
    instruct_reward_model/test_results/30_epochs_test") # change
    the input for analysis

def load_all_results(output_dir):
    records = []
    for model_dir in output_dir.iterdir():
        csv_path = model_dir / "results.csv"
        if not csv_path.exists():
            print(f"Skipping {model_dir.name}, no results.csv
                found.")
            continue
        df = pd.read_csv(csv_path)
        df["model"] = model_dir.name
        records.append(df)
```

```
19      return pd.concat(records, ignore_index=True) if records else
            pd.DataFrame()
20
21  def main():
22      df_all = load_all_results(OUTPUT_DIR)
23
24      if df_all.empty:
25          print("No results found. Make sure the evaluation outputs
                exist.")
26          return
27
28      # --- Summary Logs ---
29      print("=" * 60)
30      print("CLIP Score Interpretation")
31      print(" - Each score measures how well the generated tactile
            image aligns with the instruction.")
32      print(" - Higher is better (stronger visual-text alignment).")
33      print(" - Scores in your setting typically range 20-35.")
34      print(" - This helps compare how well each model follows human
            feedback.")
35      print("=" * 60)
36
37      # --- Mean Score per Model ---
38      mean_scores = df_all.groupby("model")["score"].mean().
            sort_values(ascending=False)
39      max_score = df_all["score"].max()
40      min_score = df_all["score"].min()
41
42      print("\n=== Mean CLIP Score per Model (Higher is Better) ==="
            )
43      print(mean_scores.round(2))
44      print(f"\n Score Range Across All Models: min={min_score:.2f},
            max={max_score:.2f}")
45
46      # --- Category-wise Mean Scores ---
47      print("\n=== Category-wise Mean Scores per Model ===")
48      pivot = df_all.pivot_table(index="category", columns="model",
            values="score", aggfunc="mean")
49      print(pivot.round(2))
50
51      # --- Optional Save ---
```

```
52      pivot.to_csv("/home/student/khan/youssif/instruct_reward_model
           /analysis_results/30_epochs_test_by_category.csv")  #change
            the name of the file
53      mean_scores.to_csv("/home/student/khan/youssif/
           instruct_reward_model/analysis_results/30
           _epochs_test_overall.csv") #change the name of the file
54
55  if __name__ == "__main__":
56      main()
```

Listing A.3: analyzer.py

"train2.py"

```
1   import torch
2   from torchvision import transforms
3   import json
4   from pathlib import Path
5   from typing import List, Dict
6   from PIL import Image
7   from datasets import Dataset, Features, Image as ImageFeature,
       Value
8   from transformers import CLIPProcessor, CLIPModel, get_scheduler
9   from diffusers import StableDiffusionImg2ImgPipeline
10  from peft import LoraConfig, prepare_model_for_kbit_training,
       get_peft_model
11  from torch.nn import functional as F
12  from torch.utils.data import DataLoader
13  import wandb
14  from tqdm import tqdm
15  import random
16  from torchvision.transforms import Compose, ToTensor, Normalize
17  import time
18  from datetime import datetime
19  import os
20  import lpips
21  import piq
22  import itertools
23
24
25  # Config
```

```
MODEL_PATH = "/home/student/khan/youssif/pix2pix_baseline/15k_steps"
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
BATCH_SIZE = 4
IMAGE_SIZE = (512, 512)
NUM_EPOCHS = 30
LEARNING_RATE = 5e-5 #1e-5

# Reward setup
REWARD_TYPE = "clip+lpips"      # "clip", "lpips", "ssim", "pixel", "clip+lpips", "clip+ssim", "clip+pixel"
REWARD_WEIGHTS = {
    "clip": 0.6, # adjust this back to 0.6
    "lpips": 0.4, #adjsut this back to 0.4
    "ssim": 0.4,
    "pixel": 0.4,
}

wandb.init(project="tactile-dpo-training")

# ----------------------
# 1. Load Dataset
# ----------------------
def load_image_pairs(jsonl_path: Path) -> List[Dict]:
    pairs = []
    with open(jsonl_path) as f:
        for line in f:
            try:
                data = json.loads(line)
                rating = 1 if data["rating"].lower() == "accept" else 0
                input_img_path = Path(data["input_image"])
                gen_img_path = Path(data["generated_image"])

                if not input_img_path.exists() or not gen_img_path.exists():
                    continue

                pairs.append({
                    "input_image": Image.open(input_img_path).convert("RGB").resize(IMAGE_SIZE),
```

```
62                      "generated_image": Image.open(gen_img_path).
                            convert("RGB").resize(IMAGE_SIZE),
63                      "instruction": data["feedback_text"] if data["
                            feedback_text"].lower() != "nan" else "",
64                      "rating": rating
65                  })
66              except Exception as e:
67                  print(f"Error: {e}")
68      return pairs
69
70  dataset_features = Features({
71      "input_image": ImageFeature(),
72      "generated_image": ImageFeature(),
73      "instruction": Value("string"),
74      "rating": Value("int32")
75  })
76
77  vae_preprocess = Compose([
78      ToTensor(),
79      Normalize([0.5, 0.5, 0.5], [0.5, 0.5, 0.5])
80  ])
81
82  # ----------------------
83  # 2. Reward Model
84  # ----------------------
85  class CLIPRewardModel:
86      def __init__(self):
87          self.model = CLIPModel.from_pretrained("openai/clip-vit-
                base-patch32").to(DEVICE)
88          self.processor = CLIPProcessor.from_pretrained("openai/
                clip-vit-base-patch32")
89
90      def calculate_reward(self, images: List[Image.Image], texts:
            List[str]) -> torch.Tensor:
91          inputs = self.processor(text=texts, images=images,
                return_tensors="pt", padding=True, truncation=True).to(
                DEVICE)
92          with torch.no_grad():
93              outputs = self.model(**inputs)
94              sims = outputs.logits_per_image
95              diagonal = sims.diagonal()
```

```
96              return diagonal
97
98  reward_model = CLIPRewardModel()
99
100 augment_ref = transforms.Compose([
101     transforms.RandomHorizontalFlip(),
102     transforms.ColorJitter(brightness=0.1, contrast=0.1,
        saturation=0.1),
103     transforms.GaussianBlur(kernel_size=3),
104 ])
105
106 augment_tgt = transforms.Compose([
107     transforms.RandomHorizontalFlip(),
108     transforms.ColorJitter(brightness=0.1, contrast=0.1),  # no
        saturation jitter here
109 ])
110
111
112 def collate_fn(batch: List[Dict]) -> Dict:
113     augmented = []
114     for item in batch:
115         augmented.append(item)
116         augmented.append({
117             "input_image": augment_ref(item["input_image"]),
118             "generated_image": augment_tgt(item["generated_image"
                ]),
119             "instruction": item["instruction"],
120             "rating": item["rating"]
121         })
122
123     return {
124         "input_images": [x["input_image"] for x in augmented],
125         "generated_images": [x["generated_image"] for x in
            augmented],
126         "instructions": [x["instruction"] for x in augmented],
127         "ratings": [x["rating"] for x in augmented],
128     }
129
130 class RewardSuite:
131     def __init__(self):
132         self.clip = CLIPRewardModel()
```

```
133         self.lpips = lpips.LPIPS(net='vgg').to(DEVICE).eval()
134         self.ssim = piq.ssim
135 
136     @staticmethod
137     def pil_to_tensor_01(img: Image.Image) -> torch.Tensor:
138         return transforms.functional.to_tensor(img).unsqueeze(0).
               to(DEVICE)
139 
140     @staticmethod
141     def pil_to_tensor_vae(imgs: List[Image.Image]) -> torch.Tensor
           :
142         t = [vae_preprocess(im) for im in imgs]
143         return torch.stack(t).to(DEVICE)
144 
145     def get_clip_reward(self, gen_imgs, texts) -> torch.Tensor:
146         return self.clip.calculate_reward(gen_imgs, texts)
147 
148     def get_lpips_reward(self, inp_imgs, gen_imgs) -> torch.Tensor
           :
149         with torch.no_grad():
150             d = []
151             for a, b in zip(inp_imgs, gen_imgs):
152                 A = transforms.functional.to_tensor(a).unsqueeze
                       (0).to(DEVICE)  # 0..1
153                 B = transforms.functional.to_tensor(b).unsqueeze
                       (0).to(DEVICE)
154                 A = A * 2 - 1  # -> [-1,1]
155                 B = B * 2 - 1
156                 d.append(self.lpips(A, B).squeeze())
157             d = torch.stack(d).to(DEVICE)
158             d = torch.clamp(d, 0.0, 1.0)
159             reward = 1.0 - d
160         return reward
161 
162 
163     def get_pixel_reward(self, inp_imgs, gen_imgs) -> torch.Tensor
           :
164         with torch.no_grad():
165             A = self.pil_to_tensor_vae(inp_imgs)
166             B = self.pil_to_tensor_vae(gen_imgs)
```

```
167             mse = F.mse_loss(A, B, reduction='none').mean(dim
                    =(1,2,3))
168             mse = torch.clamp(mse, 0.0, 1.0)
169             reward = 1.0 - mse
170         return reward
171
172     def get_ssim_reward(self, inp_imgs, gen_imgs) -> torch.Tensor:
173         with torch.no_grad():
174             vals = []
175             for a, b in zip(inp_imgs, gen_imgs):
176                 A = self.pil_to_tensor_01(a)
177                 B = self.pil_to_tensor_01(b)
178                 vals.append(self.ssim(A, B, data_range=1.0).
                        squeeze())
179             return torch.stack(vals).to(DEVICE)
180
181     def get_saturation_reward(self, imgs) -> torch.Tensor:
182         # High when images are closer to grayscale
183         with torch.no_grad():
184             vals = []
185             for im in imgs:
186                 t = transforms.functional.to_tensor(im).to(DEVICE)
                          # [3,H,W], 0..1
187                 mx, mn = t.max(dim=0).values, t.min(dim=0).values
188                 delta = mx - mn + 1e-6
189                 v = mx + 1e-6
190                 s = (delta / v).mean()       # mean HSV-like
                        saturation proxy
191                 vals.append(1.0 - s)         # higher reward for
                        lower saturation
192             return torch.stack(vals).to(DEVICE)
193
194     def get_combined_reward(self, inp_imgs, gen_imgs, texts, kind:
        str, weights: Dict[str, float]) -> torch.Tensor:
195         kind = kind.lower()
196         if "+" not in kind:
197             if kind == "clip":  return self.get_clip_reward(
                    gen_imgs, texts)
198             if kind == "lpips": return self.get_lpips_reward(
                    inp_imgs, gen_imgs)
```

```
199             if kind == "pixel": return self.get_pixel_reward(
                    inp_imgs, gen_imgs)
200             if kind == "ssim":  return self.get_ssim_reward(
                    inp_imgs, gen_imgs)
201             if kind == "sat":    return self.get_saturation_reward
                    (gen_imgs)
202             raise ValueError(f"Unknown REWARD_TYPE: {kind}")
203 
204         parts = [k.strip() for k in kind.split("+")]
205         w_sum = sum(weights.get(p, 0.0) for p in parts)
206         if w_sum <= 0:
207             raise ValueError("Combined reward has zero total
                    weight.")
208         norm = {p: weights.get(p, 0.0)/w_sum for p in parts}
209 
210         total = None
211         for p in parts:
212             if p == "clip":   r = self.get_clip_reward(gen_imgs,
                    texts)
213             elif p == "lpips": r = self.get_lpips_reward(inp_imgs,
                     gen_imgs)
214             elif p == "pixel": r = self.get_pixel_reward(inp_imgs,
                     gen_imgs)
215             elif p == "ssim":  r = self.get_ssim_reward(inp_imgs,
                    gen_imgs)
216             elif p == "sat":   r = self.get_saturation_reward(
                    gen_imgs)
217             else: raise ValueError(f"Unknown combo part: {p}")
218             total = r*norm[p] if total is None else total + r*norm
                    [p]
219         return total
220 
221 reward_suite = RewardSuite()
222 
223 # ----------------------
224 # 3. Custom Training Loop
225 # ----------------------
226 def prepare_unet_for_lora(unet):
227     return prepare_model_for_kbit_training(unet)
228 
229 # ----------------------
```

```
230 # Loss Variants
231 # ----------------------
232 def mse_loss_fn(pred, target, **kwargs):
233     return F.mse_loss(pred, target)
234 
235 def clip_weighted_mse_loss(pred, target, ratings, **kwargs):
236     accepted_mask = torch.tensor(ratings, device=DEVICE, dtype=
        torch.bool)
237     loss_weights = torch.where(
238         accepted_mask,
239         torch.tensor(0.5, device=DEVICE),
240         torch.tensor(2.0, device=DEVICE)
241     )
242     base_loss = F.mse_loss(pred, target)
243     return base_loss * loss_weights.mean()
244 
245 
246 def reward_weighted_mse_loss(pred, target, rewards=None, **kwargs)
    :
247     rewards = rewards.to(device=pred.device, dtype=pred.dtype)
248     base_loss = F.mse_loss(pred, target, reduction='none')
249     weight = rewards.reshape(-1, 1, 1, 1)
250     return (base_loss * weight).mean()
251 
252 def inverse_reward_loss(pred, target, rewards=None, **kwargs):
253     rewards = rewards.to(device=pred.device, dtype=pred.dtype)
254     base_loss = F.mse_loss(pred, target, reduction='none')
255     inverse_weight = (1.0 - rewards).unsqueeze(1).unsqueeze(2).
        unsqueeze(3)
256     return (base_loss * inverse_weight).mean()
257 
258 def margin_reward_loss(pred, target, rewards=None, **kwargs):
259     rewards = rewards.to(device=pred.device, dtype=pred.dtype)
260     base_loss = F.mse_loss(pred, target, reduction='none')
261     margin = 0.2
262     clipped = torch.clamp(margin - rewards.unsqueeze(1).unsqueeze
        (2).unsqueeze(3), min=0)
263     return (base_loss * clipped).mean()
264 
265 # Update map:
266 loss_fn_map = {
```

```
267     "mse": mse_loss_fn,
268     "clip_weighted_mse": clip_weighted_mse_loss,
269     "reward_weighted_mse": reward_weighted_mse_loss,
270     "inverse_reward_loss": inverse_reward_loss,
271     "margin_reward_loss": margin_reward_loss
272 }
273 
274 def custom_train_loop(pipe, dataset, run_name, loss_type="
       clip_weighted_mse", num_epochs=NUM_EPOCHS):
275     dataloader = DataLoader(dataset, batch_size=BATCH_SIZE,
           shuffle=True, collate_fn=collate_fn)
276     trainable_params = itertools.chain(
277         (p for p in pipe.unet.parameters() if p.requires_grad),
278         (p for p in pipe.text_encoder.parameters() if p.
               requires_grad),
279     )
280     optimizer = torch.optim.AdamW(trainable_params, lr=
           LEARNING_RATE)
281 
282     scheduler = get_scheduler("cosine", optimizer,
           num_warmup_steps=10, num_training_steps=len(dataloader) *
           num_epochs)
283     pipe.unet.train()
284     pipe.text_encoder.train()
285 
286 
287     start_time = time.time()  # Start timing
288     for epoch in range(num_epochs):
289         for step, batch in enumerate(tqdm(dataloader, desc=f"Epoch
               {epoch+1}/{num_epochs}")):
290             input_imgs_pil = batch["input_images"]
291             gen_imgs_pil   = batch["generated_images"]
292             instructions   = batch["instructions"]
293             ratings        = batch["ratings"]
294 
295             rewards = reward_suite.get_combined_reward(
296                 input_imgs_pil, gen_imgs_pil, instructions,
297                 kind=REWARD_TYPE, weights=REWARD_WEIGHTS
298             )
299             assert rewards.shape[0] == len(gen_imgs_pil), f"
                   rewards {rewards.shape} vs batch {len(gen_imgs_pil)
```

```
            }"


        # input_images = [vae_preprocess(img).to(device=DEVICE
            , dtype=torch.float32) for img in batch["
            input_images"]]
        # input_images = torch.stack(input_images)  # shape: [
            B, 3, 512, 512]

        # VAE encode both; NOISE THE TARGET
        ref_images = torch.stack([vae_preprocess(x) for x in
            input_imgs_pil]).to(DEVICE)
        tgt_images = torch.stack([vae_preprocess(x) for x in
            gen_imgs_pil]).to(DEVICE)

        with torch.no_grad():
            ref_latents = pipe.vae.encode(ref_images).
                latent_dist.sample() * 0.18215
            tgt_latents = pipe.vae.encode(tgt_images).
                latent_dist.sample() * 0.18215

        # latents = latents.to(dtype=torch.float32)
        timesteps = torch.randint(
            0,
            pipe.scheduler.config.num_train_timesteps,
            (tgt_latents.shape[0],),
            device=DEVICE,
            dtype=torch.long
        )
        noise = torch.randn_like(tgt_latents).to(dtype=torch.
            float32, device=DEVICE)
        noisy_tgt = pipe.scheduler.add_noise(tgt_latents,
            noise, timesteps)
        # latents = latents.to(device=DEVICE, dtype=torch.
            float32)
        # noisy_latents = pipe.scheduler.add_noise(latents,
            noise, timesteps)
        text_inputs = pipe.tokenizer(batch["instructions"],
            padding="max_length", max_length=pipe.tokenizer.
            model_max_length, truncation=True, return_tensors="
            pt").to(DEVICE)
```

```
326             encoder_hidden_states = pipe.text_encoder(**
                    text_inputs).last_hidden_state
327             encoder_hidden_states = encoder_hidden_states.to(
                    device=DEVICE, dtype=torch.float32)
328
329             # --- choose correct UNet input (4 or 8 chans)
330             in_ch = pipe.unet.config.in_channels
331             if in_ch == 4:
332                 latent_model_input = noisy_tgt
333             elif in_ch == 8:
334                 latent_model_input = torch.cat([noisy_tgt,
                        ref_latents], dim=1)
335             else:
336                 raise ValueError(f"Unexpected in_channels={in_ch}"
                        )
337
338             noise_pred = pipe.unet(
339                 latent_model_input,
340                 timesteps,
341                 encoder_hidden_states=encoder_hidden_states
342             ).sample
343
344             # --- one-time debug (avoids log spam)
345             if epoch == 0 and step == 0:
346                 tqdm.write(f"UNet in_channels: {in_ch}")
347                 tqdm.write(f"latent_model_input: {tuple(
                        latent_model_input.shape)}")
348                 tqdm.write(f"noise_pred/noise: {tuple(noise_pred.
                        shape)} {tuple(noise.shape)}")
349             assert noise_pred.shape == noise.shape, "UNet output
                    must match noise target."
350
351
352             loss_fn = loss_fn_map[loss_type]
353             # Dynamically check if loss function takes "rewards"
354             loss_kwargs = {}
355             if "rewards" in loss_fn.__code__.co_varnames:
356                 loss_kwargs["rewards"] = rewards.detach().to(
                        noise_pred.dtype).to(noise_pred.device)
357
358             if "ratings" in loss_fn.__code__.co_varnames:
```

```
359                 loss_kwargs["ratings"] = ratings
360
361             final_loss = loss_fn(noise_pred, noise, **loss_kwargs)
362             final_loss.backward()
363             optimizer.step()
364             optimizer.zero_grad()
365             scheduler.step()
366             wandb.log({"loss": final_loss.item()})
367
368     end_time = time.time()  # End timing
369     total_time = end_time - start_time
370     output_dir = Path(f"/home/student/khan/youssif/
            instruct_reward_model/train_results/{run_name}") #Changing
            the folder's name for each epoch set
371     output_dir.mkdir(parents=True, exist_ok=True)
372
373     adapter_root = output_dir / "lora_adapters"
374     (adapter_root / "unet").mkdir(parents=True, exist_ok=True)
375     (adapter_root / "text_encoder").mkdir(parents=True, exist_ok=
            True)
376
377     pipe.unet.save_pretrained(str(adapter_root / "unet"))
378     pipe.text_encoder.save_pretrained(str(adapter_root / "
            text_encoder"))
379
380
381     # Save training metadata
382     result_log = {
383         "run_name": run_name,
384         "loss_type": loss_type,
385         "reward_type": REWARD_TYPE,
386         "num_epochs": num_epochs,
387         "total_time_sec": round(total_time, 2),
388         "final_loss": final_loss.item(),
389         "timestamp": datetime.now().isoformat()
390     }
391
392     with open(output_dir / "results.jsonl", "a") as f:
393         f.write(json.dumps(result_log) + "\n")
394
395     print(f"Training completed. Adapter saved to {adapter_root}")
```

```
396     print(f"Log written to {output_dir / 'results.jsonl'}")
397
398 # ----------------------
399 # 4. Main
400 # ----------------------
401 def main():
402     run_name = input("Enter run name (no spaces): ").strip()
403     jsonl_path = Path("./training_dataset.jsonl")
404     pairs = load_image_pairs(jsonl_path)
405     if not pairs:
406         raise ValueError("No valid pairs found.")
407     dataset = Dataset.from_list(pairs, features=dataset_features)
408     pipe = StableDiffusionImg2ImgPipeline.from_pretrained(
409         MODEL_PATH,
410         torch_dtype=torch.float32
411     ).to(DEVICE)
412
413     pipe.unet = prepare_unet_for_lora(pipe.unet)
414     lora_config = LoraConfig(
415         r=8,
416         lora_alpha=32,
417         target_modules=["to_q", "to_k", "to_v"],
418         layers_to_transform=list(range(16)),
419         lora_dropout=0.1,
420         bias="none",
421         fan_in_fan_out=False
422     )
423     pipe.unet = get_peft_model(pipe.unet, lora_config)
424     pipe.unet = pipe.unet.to(device=DEVICE, dtype=torch.float32)
425
426     lora_config_te = LoraConfig(
427         r=8,
428         lora_alpha=32,
429         target_modules=["q_proj", "k_proj", "v_proj", "out_proj"],
              # CLIPTextModel attn projections
430         lora_dropout=0.1,
431         bias="none",
432         fan_in_fan_out=False,
433     )
434     pipe.text_encoder = get_peft_model(pipe.text_encoder,
        lora_config_te)
```

```
435     pipe.text_encoder = pipe.text_encoder.to(device=DEVICE, dtype=
            torch.float32)
436
437     pipe.vae.requires_grad_(False)
438     # pipe.text_encoder.requires_grad_(True)
439     loss_type = "reward_weighted_mse"  # change loss function here
            .
440     # options: "mse", "clip_weighted_mse", "reward_weighted_mse",
            "inverse_reward_loss", "margin_reward_loss"
441     custom_train_loop(pipe, dataset, run_name=run_name, loss_type=
            loss_type)
442
443 if __name__ == "__main__":
444     main()
```

Listing A.4: "train2.py" for final adapter creation

"test2.py"

```
1  import torch
2  from diffusers import StableDiffusionInstructPix2PixPipeline
3  from peft import PeftModel
4  from transformers import CLIPProcessor, CLIPModel
5  from PIL import Image
6  from pathlib import Path
7  import json
8  import csv
9  import os
10 from tqdm import tqdm
11
12 # --- Config ---
13 DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
14 IMAGE_SIZE = (512, 512)
15 SEED = 42
16
17 torch.manual_seed(SEED)
18
19 def load_lora_adapters(pipe, lora_root: str, merge: bool = True,
       device: str = "cuda"):
20     """
21     lora_root should contain:
```

```
22        - lora_root/unet
23        - lora_root/text_encoder
24    """
25    root = Path(lora_root)
26    unet_dir = root / "unet"
27    te_dir = root / "text_encoder"
28
29    if unet_dir.exists():
30        pipe.unet = PeftModel.from_pretrained(pipe.unet, str(
              unet_dir)).to(device)
31    else:
32        print(f"[WARN] UNet adapter not found at: {unet_dir}")
33
34    if te_dir.exists():
35        pipe.text_encoder = PeftModel.from_pretrained(pipe.
              text_encoder, str(te_dir)).to(device)
36    else:
37        print(f"[WARN] Text-encoder adapter not found at: {te_dir}
              ")
38
39    if merge:
40        # Merge LoRA weights into the base modules for faster
              inference
41        if hasattr(pipe.unet, "merge_and_unload"):
42            pipe.unet = pipe.unet.merge_and_unload()
43        if hasattr(pipe.text_encoder, "merge_and_unload"):
44            pipe.text_encoder = pipe.text_encoder.merge_and_unload
                  ()
45
46    return pipe
47
48 LORA_PATH = "/home/student/khan/youssif/instruct_reward_model/
       train_results/30_epochs_clip+lpips/lora_adapters"
49 # --- Paths ---
50 RUNS = {
51     "baseline_only": {"use_lora": False, "merge": False, "
          lora_path": None},
52     "adapter_only": {"use_lora": True,  "merge": False, "lora_path
          ": LORA_PATH},
53     "both_together": {"use_lora": True, "merge": True,  "lora_path
          ": LORA_PATH},
```

```
}

MODEL_PATH = "/home/student/khan/youssif/pix2pix_baseline/15k_steps"
TEST_JSONL = "/home/student/khan/youssif/instruct_reward_model/test/test.jsonl"
OUTPUT_DIR = Path("/home/student/khan/youssif/instruct_reward_model/test_results/30_epochs_test/")
OUTPUT_DIR.mkdir(parents=True, exist_ok=True)

# --- CLIP Reward Model ---
class CLIPRewardModel:
    def __init__(self):
        self.model = CLIPModel.from_pretrained("openai/clip-vit-base-patch32").to(DEVICE)
        self.processor = CLIPProcessor.from_pretrained("openai/clip-vit-base-patch32")

    def score(self, images, texts):
        inputs = self.processor(text=texts, images=images, return_tensors="pt", padding=True, truncation=True).to(DEVICE)
        with torch.no_grad():
            outputs = self.model(**inputs)
        return outputs.logits_per_image.squeeze().tolist()

# --- Load Test Data ---
def load_test_set(path):
    samples = []
    with open(path) as f:
        for line in f:
            data = json.loads(line)
            samples.append({
                "input_image": data["input_image"],
                "prompt": data["feedback_text"],
                "category": data.get("Category", "unknown")
            })
    return samples

# --- Inference Function ---
def generate_image(pipe, image_path, prompt):
```

```
88      image = Image.open(image_path).convert("RGB").resize(
            IMAGE_SIZE)
89      generator = torch.Generator(DEVICE).manual_seed(SEED)
90      with torch.no_grad():
91          result = pipe(prompt=prompt, image=image, strength=0.8,
                guidance_scale=10, num_inference_steps=30, generator=
                generator)
92      return result.images[0].convert("RGB")
93
94  # --- Main Evaluation ---
95  def main():
96      test_samples = load_test_set(TEST_JSONL)
97      scorer = CLIPRewardModel()
98
99      for name, cfg in RUNS.items():
100         print(f"\nEvaluating: {name}")
101         model_output_dir = OUTPUT_DIR / name
102         model_output_dir.mkdir(exist_ok=True)
103
104         pipe = StableDiffusionInstructPix2PixPipeline.
                from_pretrained(
105              MODEL_PATH,
106              torch_dtype=torch.float32,
107              safety_checker=None
108         ).to(DEVICE)
109
110         if cfg["use_lora"]:
111             pipe = load_lora_adapters(pipe, cfg["lora_path"],
                    merge=cfg["merge"], device=DEVICE)
112
113         pipe.unet.eval()
114         pipe.vae.eval()
115         pipe.text_encoder.eval()
116
117         results = []
118         for idx, sample in enumerate(tqdm(test_samples)):
119             try:
120                 gen_img = generate_image(pipe, sample["input_image
                        "], sample["prompt"])
121                 score = scorer.score([gen_img], [sample["prompt"
                        ]])
```

```
122                 out_path = model_output_dir / f"{idx:04d}.png"
123                 gen_img.save(out_path)
124                 results.append({
125                     "idx": idx,
126                     "category": sample["category"],
127                     "prompt": sample["prompt"],
128                     "score": score,
129                     "image_path": str(out_path)
130                 })
131             except Exception as e:
132                 print(f"Error on sample {idx}: {e}")
133 
134         if results:
135             csv_path = model_output_dir / "results.csv"
136             with open(csv_path, "w", newline="") as csvfile:
137                 writer = csv.DictWriter(csvfile, fieldnames=
                        results[0].keys())
138                 writer.writeheader()
139                 writer.writerows(results)
140         else:
141             print(f"No results to save for {name}")
142 
143 if __name__ == "__main__":
144     main()
```

Listing A.5: "test2.py" for final adapter and instruct-tuned img2img baseline testing