Imperial College London

BEng Individual Project

Imperial College London

Department of Computing

EC-LoRA-T: Engagement-Conditioned Low-Rank Adaptation for Time-Aware Aesthetic Generation

Author:
Geonwoo Park

Supervisor:
Dr. Thomas Lancaster

Second Marker:
Dr. Tom Crossland

June 13, 2025

Abstract

This project presents EC-LoRA-T, a generative framework that integrates Low-Rank Adaptation (LoRA) with an Engagement Conditioning Module (ECM) to modulate diffusion models using social media engagement and temporal context. The model was trained on 273 Instagram posts from the influencer `sungccess`, using ML-generated English captions and a two-dimensional conditioning vector encoding normalized engagement and post timing. A linear regression model trained on VAE-extracted latents achieved an R^2 score of 0.9769 and MSE of 0.0003, confirming that engagement is strongly encoded in visual features. However, when varying the engagement input across five target values (0.00 to 1.00), the predicted engagement of generated images remained narrowly distributed (0.0549–0.0827), and qualitative differences were visually subtle or inconsistent. These findings suggest that while EC-LoRA-T can model engagement in latent space, it struggles to translate it into controlled image generation without further architectural or training improvements.

Acknowledgements

First, I would like to extend my gratitude to my supervisor, Dr. Thomas Lancaster, for his constant support. He has taught me many lessons across the board, from research skills to work management. Meetings with him were really helpful as they guided me through the complexities of my research. I also wish to thank my second marker, Dr. Tom Crossland, for his helpful feedback and suggestions. His advice to focus on the research side of the project, particularly Low-Rank Adaptation, gave me a critical breakthrough in my project. Next, I would like to thank my personal tutor, Dr. Pancham Shukla, for his regular check-ins throughout my years at Imperial. Finally, I am very grateful to my family and friends for their support throughout my days at Imperial. Thank you.

Contents

Chapter 1

Introduction

Low-Rank Adaptation (LoRA) has emerged as a transformative technique for advancing generative AI, particularly in the context of text-to-image synthesis using models like Stable Diffusion. Traditional fine-tuning of large generative models required expensive computing resources, limiting their adaptability for specialised tasks. LoRA solved the problem by introducing a parameter-efficient approach. Instead of changing big weight matrices directly, it breaks the changes into smaller, low-rank pieces, reducing the use of required memory and computing power, while maintaining the quality of the output [1].

Studies exploring the application of LoRA to Stable Diffusion have illustrated its advantages in enabling efficient fine-tuning. Zhang et al. [2] conducted an innovative bridge aesthetics design using LoRA. With a real photo dataset of the Coral Bridge in Qingdao, they created thousands of novel bridge types with styles similar to the dataset. The study demonstrated LoRA's capability to capture key structural characteristics from the limited training images and rapidly generate creative architectural designs using modest hardware resources (an RTX4060Ti GPU). Similarly, Shrestha et al. [3] applied LoRA to fine-tune Stable Diffusion on a dataset of Calvin and Hobbes comics for style transfer. Their approach enabled the model to convert input images into the distinctive comic style using stable-diffusion-v1.5 with minimal training. Despite limited data and compute, the results were visually compelling, highlighting LoRA's effectiveness in enabling accessible and efficient personalization of generative models.

LoRA has shown strong results in making generative models more efficient for style and domain-specific tasks, yet a limitation remains in dynamic content environments like social media. Even with LoRA fine-tuning, diffusion models don't have a built-in way to learn from implicit human feedback (e.g., likes and comments). This deficiency results in models producing visually appealing content that may not necessarily resonate with or achieve social validation from an audience. Most current methods either stick to generic style changes or rely on explicit human ranking, which lacks scalability. This challenges AI to generate content that reflects social engagement and trends.

To bridge this gap, this project proposes a novel generative framework: Engagement-Conditioned Low-Rank Adaptation with Time Variance (EC-LoRA-T). Our central hypothesis is: "Can a LoRA-based diffusion model be dynamically conditioned on historical engagement metrics and timing to generate visual content that aligns with past high-performing aesthetics?" This endeavor elevates the work beyond a mere application of existing generative techniques, positioning it as a study into how implicit human feedback dynamically shapes and refines learned aesthetic representations within an AI framework.

1.1 Project Scope and Key Contributions

This project focuses on designing, building, and testing a new method called Engagement-Conditioned Low-Rank Adaptation with Time Variance (EC-LoRA-T). It builds on the existing LoRA technique — a popular way to fine-tune large models - and introduces new features tailored for diffusion models, particularly Stable Diffusion. The key idea is to adapt the model not just with low-rank updates, but also in a way that responds to engagement signals and time-based patterns, making the fine-tuning more context-aware and dynamic.

This project makes several key technical contributions:

- **Dynamic LoRA Scaling via Engagement Conditioning Module (ECM):** Diverging from conventional LoRA's fixed parameter additions, EC-LoRA-T introduces a dedicated Engagement Conditioning Module (ECM). This module dynamically modulates the influence of each LoRA layer based on time-aware engagement data, rendering the generative process directly sensitive to external behavioral signals and implicit human feedback.
- **Time Variance Integration:** The input to the ECM is a special vector that combines two things: normalised engagement data (like likes or comments) and a normalised time-related feature. By blending these two types of information, the EC-LoRA-T model learns how aesthetic trends change over time. This means the model can generate images that are not only based on engagement but also match the style or popularity patterns of a specific time period.
- **Parameter-Efficient Training:** During fine-tuning, the large pre-trained diffusion model stays completely unchanged — none of its original parameters are updated. Instead, training is limited to just the small LoRA matrices and the internal weights of the ECM. This focused approach keeps the training process lightweight and efficient, while still allowing the model to learn from the new engagement-based conditioning.

Collectively, these contributions facilitate a pioneering approach to integrating implicit human feedback and temporal dynamics directly into the parameter-efficient fine-tuning of generative models, moving beyond generic style adaptation towards contextually and socially informed content generation.

Chapter 2

Background

2.1 Fundamentals of Generative AI and Diffusion Models

Generative Artificial Intelligence (AI) refers to models that are trained to create new data similar to what they've seen during training [4]. Among these, diffusion models have emerged as a prominent paradigm, demonstrating exceptional capabilities in generating high-fidelity and diverse samples, particularly in the domain of image synthesis [5].

Diffusion models operate by iteratively transforming data from a simple prior distribution (typically Gaussian noise) into complex, structured data through a process known as reverse diffusion. This is underpinned by a forward diffusion process:

- **Forward Diffusion (Noising Process):** This process follows a fixed, Markovian structure, meaning each step depends only on the one before it. Over a series of timesteps, Gaussian noise is gradually added to the input data (like an image). As this continues, the image slowly loses its original details and becomes more and more distorted, until it eventually turns into pure random noise.
- **Reverse Diffusion (Denoising Process):** The generative part of the model comes from learning how to reverse the noise-adding process. A neural network (usually a U-Net) is trained to predict the noise that was added at each step. Then, by repeatedly subtracting this predicted noise from a noisy image, the model slowly reconstructs a clear, realistic image from pure noise.

2.1.1 Latent Diffusion Models (LDMs)

Latent Diffusion Models (LDMs), like Stable Diffusion, are a powerful type of generative model designed for creating high-quality images. What makes them special is that they don't try to generate images directly from pixels. Instead, they do most of the work in a smaller, compressed version of the image, known as the latent space [6]. This is done using a structure called an autoencoder, which has two main parts:

- **Encoder:** Takes a large image (for example, 512x512 pixels) and compresses it into a smaller set of numbers (called "latents"), such as 64x64. This captures the essential information while throwing away unnecessary details.
- **Decoder:** Later, this compressed version is turned back into the full image.

By doing the image generation in this compressed latent space, LDMs can work much more efficiently, especially when dealing with high-resolution outputs. To make this work, LDMs rely on three important components:

- **Variational Autoencoder (VAE):** This is the part that does the compression and decompression, helping the model operate in a smaller space.

- **U-Net:** This is the main network that learns how to remove noise from the latent image, step by step. It's trained to start from random noise and gradually turn it into something meaningful.

- **Text Encoder:** If we want to guide the model with a text prompt (like "a cat flying in space"), the text is first turned into a special numerical format using a language model (like CLIP's text encoder). These numbers are then used to help guide the image generation process, so the result matches the prompt.

During training, the model learns how to predict and remove the noise that was added to the image in earlier steps. By learning this process well, it can generate completely new images from scratch, just by starting from noise and optionally using a text prompt as guidance.

2.2 Low-Rank Adaptation (LoRA) Techniques

Fine-tuning large pre-trained models, such as transformers or diffusion models, often requires a lot of memory and storage. This is because traditional fine-tuning updates all the parameters in the model — which can be hundreds of millions — and stores a full copy of the model for each fine-tuned task.

Low-Rank Adaptation (LoRA) is a smarter and more efficient way to fine-tune these models. It allows us to train only a small number of extra parameters — without touching the original weights — while still achieving strong performance [1].

2.2.1 Core Principle of LoRA

LoRA is based on the idea that when a model is adapted to a new task, the changes needed in its weights are quite simple — they can be captured using low-rank (low-complexity) updates, rather than modifying the full weight matrices. Instead of directly fine-tuning the full weight matrices of a pre-trained model, LoRA introduces a small number of additional trainable parameters that capture these low-rank updates. For a pre-trained weight matrix $W_0 \in R^{d\times k}$, LoRA approximates its update ΔW by factorising it into two smaller matrices, $A \in R^{d\times r}$ and $B \in R^{r\times k}$, where $r \ll \min(d, k)$ is the chosen low rank. The adapted weight matrix W' is then expressed as:

$$W' = W_0 + BA$$

During training, the original weight matrix W_0 remains frozen. Only the newly introduced low-rank matrices A and B are trainable. The output of the adapted layer becomes $h' = W_0x + BAx$. A scaling factor, α/r, is typically applied to the output of the LoRA matrices, where α is a constant that influences the magnitude of the learned update. The full layer output then becomes:

$$h' = W_0x + \frac{\alpha}{r}BAx$$

This scaling ensures that the LoRA update is properly weighted, allowing α to be decoupled from r and often set to a value like r or $2r$.

2.2.2 Efficiency and Benefits

The primary advantage of LoRA lies in its remarkable efficiency. By training only a small fraction of the total parameters (the A and B matrices), the number of trainable parameters can be reduced by several orders of magnitude compared to full fine-tuning. This translates directly into:

- **Reduced Memory Footprint:** Only a small part of the model is trained, so it uses much less GPU memory.
- **Faster Training:** With fewer parameters being updated, training is quicker.
- **Smaller Checkpoints:** You only need to store the small A and B matrices for each fine-tuned version, instead of the entire model.

LoRA has proven to be effective in many areas. In natural language processing, it helps fine-tune large language models (like GPT or BERT) for specific tasks such as summarisation or translation. More recently, it has shown great results in image generation, especially for fine-tuning diffusion models (like Stable Diffusion) to learn new styles or visual concepts — for example, adapting the model to generate anime-style art or custom aesthetics with just a few training images.

2.3 Engagement-Conditioned and Feedback-Driven Generative Models

As generative AI continues to improve, it becomes increasingly important to develop ways to align its outputs with human preferences, especially when it comes to subjective qualities like aesthetic style or social impact. Traditional generative models often produce outputs without any direct feedback from real users, which means the results can be technically correct but miss the mark in terms of what people actually find meaningful or appealing. This section looks at different strategies for integrating human feedback into the generation process to make models more responsive and relevant.

2.3.1 Explicit Feedback Mechanisms

Explicit feedback mechanisms involve direct human intervention to guide or refine generative models. The most prominent example is **Reinforcement Learning from Human Feedback (RLHF)**, widely applied in large language models [7]. In RLHF, human annotators provide pairwise comparisons or rankings of model outputs, which are then used to train a separate reward model. This reward model subsequently serves as a proxy for human preferences, optimising the generative model via reinforcement learning algorithms (e.g., Proximal Policy Optimisation). While highly effective in aligning outputs with complex human preferences, RLHF is inherently resource-intensive, demanding significant human annotation effort, which limits its scalability.

Beyond language models, **Aesthetic Prediction Models** represent another form of explicit feedback [8]. These models are trained on datasets of images pre-scored by humans for aesthetic quality. Once trained, they can predict an "aesthetic score" for new images, which can then be used as a reward signal to fine-tune generative models towards higher aesthetic appeal. However, these models are constrained by the quality and diversity of their training data and often generalise poorly to novel or domain-specific aesthetics.

2.3.2 Implicit Feedback Integration

Unlike explicit feedback (like user ratings), implicit feedback relies on passively collected signals from user behaviour, making it much more scalable and cost-effective [9]. On social media platforms, for example, massive amounts of implicit feedback are generated every day through actions like likes, shares, and comments. While these signals aren't direct ratings of an image's aesthetic quality, they do reflect how much people engage with or respond to the content — which often indicates its perceived value or appeal.

2.3.3 Project Positioning: Leveraging Implicit Social Engagement

EC-LoRA-T distinguishes itself by directly integrating *implicit, quantitative social engagement metrics* and *temporal context* as a continuous conditioning mechanism during the fine-tuning of a latent diffusion model. Unlike methods relying on costly explicit human annotation loops or static aesthetic predictors, our framework transforms readily available behavioural data (e.g., `likes_count` and `comments_count`) and temporal information (e.g., `days_since_earliest_post`) into dynamic generative parameters. By doing so, the model learns to create content that aligns with socially validated aesthetics — styles and visuals that people are actually responding to — and can adapt to how those preferences change over time. This makes it possible to train generative models in a way that's both scalable and context-aware, offering a new way forward beyond traditional feedback-based methods.

2.4 Time-Aware Generative Models

In many real-world applications — like fashion, music, or social media — aesthetics and preferences change over time. To stay relevant, generative models need to understand and reflect these temporal trends. However, most traditional models generate content based on a static snapshot of data, which means their outputs can quickly become outdated or out of sync with current styles. Adding time-awareness helps generative models produce content that not only makes sense in context, but also aligns with specific moments or periods, such as seasonal trends or stylistic shifts over months or years.

2.4.1 Approaches to Temporal Integration

Different methods have been used to help generative models handle time, depending on the kind of data and the level of time sensitivity required:

- **Explicit Sequential Modelling:** For tasks where data naturally follows a sequence (like video, speech, or time series), models use structures like RNNs, LSTMs, temporal CNNs, or more recently, Transformers [10]. These architectures are built to process data in order, making them well-suited for generating sequences or predicting future steps.

- **Time Embeddings and Conditioning:** In diffusion models, it's common to include a timestep embedding to tell the model what stage of the noise-removal process it's in [11]. However, this relates to the model's internal process — not to real-world time (e.g., seasons, dates, or cultural trends). Some models do add simple time-based inputs like time-of-day or day-of-week, but these usually reflect repeating patterns, not evolving ones.

- **Temporal Generative Adversarial Networks (GANs) and Variational Autoencoders (VAEs):** Some variants of GANs and VAEs have been designed to model temporal changes by introducing latent sequences or using recurrent components [12]. These setups help in generating content that evolves over time, or even in forecasting what future content might look like.

2.4.2 Project Positioning: Implicit Temporal Conditioning

While earlier work has looked at time in generative models, EC-LoRA-T introduces a new way to handle it using implicit temporal conditioning within a diffusion model. It does this by feeding a normalised time feature — like the number of days since an influencer's first post — directly into the Engagement Conditioning Module (ECM). This value helps control how much influence the LoRA layers have at each step, effectively guiding the model to learn how an influencer's aesthetic changes over time. Unlike approaches that use fixed time embeddings or generate content in strict sequences, EC-LoRA-T can naturally pick up on gradual shifts in style or trends. As a result, it can generate content that feels timely and in sync with the period it's meant to represent within the broader timeline of the dataset.

2.5 Gaps in Existing Research and Project Positioning

Despite progress in generative AI and efficient fine-tuning methods, key gaps remain in generating content that truly reflects human preferences and evolving temporal trends.

2.5.1 Current Limitations

- **Absence of Implicit Feedback Integration:** While methods like Reinforcement Learning from Human Feedback (RLHF) and human ranking are effective at aligning models with human preferences, they are resource-heavy and hard to scale — especially for the large, constantly changing datasets found on social media. There is still a lack of efficient ways to directly leverage implicit feedback, such as likes or comments, despite its abundance.

- **Scalability Challenges in Feedback Loops:** The cost and complexity of collecting explicit human annotations make it difficult to apply feedback-driven generative models at scale in fast-moving content environments.

- **Limited Granularity in Temporal Conditioning:** Most existing time-aware generative models rely on sequences or fixed time features (like time-of-day), but they often struggle to learn subtle, continuous style changes from unordered data like individual social media posts.

- **Static LoRA Conditioning:** While LoRA is efficient, it applies static adaptations and lacks built-in support for dynamically adjusting its influence based on continuous signals like engagement levels or time.

2.5.2 Project Positioning

The EC-LoRA-T project is specifically positioned to address these identified limitations through its unique architectural and methodological contributions:

- **Novel Implicit Conditioning:** EC-LoRA-T introduces a pioneering approach to directly integrate *implicit, quantitative social engagement metrics* (`likes_count` and `comments_count`) and *continuous temporal context* (`normalized_time`) as dynamic conditioning signals. This bypasses the scalability constraints of explicit feedback loops.
- **Dynamic LoRA Modulation:** The core novelty lies in the **Engagement Conditioning Module (ECM)**, which dynamically computes scalar scaling factors for each LoRA layer based on the input engagement-time vector. This turns LoRA from a static method into a dynamically controllable system, allowing for fine-tuned control over the generated content.
- **Time-Aware Aesthetic Learning:** By feeding normalised time features into the ECM, the model learns how aesthetic trends evolve over time, enabling it to generate content that fits specific temporal contexts.
- **Efficiency and Generalisation:** Leveraging LoRA's efficiency, EC-LoRA-T enables complex conditional generation with only a small increase in trainable parameters, making it practical for real-world use and deployment.

By integrating implicit human feedback and temporal dynamics into a parameter-efficient framework, EC-LoRA-T pushes generative AI forward, enabling models to produce content that better reflects audience engagement and evolving aesthetic trends.

Chapter 3

Methodologies

This section outlines the key components of our EC-LoRA-T framework for fine-tuning latent diffusion models using implicit engagement signals and temporal context.

3.1 EC-LoRA-T Architecture Overview

Engagement-Conditioned Low-Rank Adaptation with Time Variance (EC-LoRA-T) is a new architecture designed to enhance fine-tuning for pre-trained latent diffusion models. It enables dynamic conditioning of generated outputs using implicit social engagement data and temporal context. The core innovation is the integration of a custom LoRA layer, an Engagement Conditioning Module (ECM), and a modified U-Net forward pass that brings everything together.

At a high level, the EC-LoRA-T architecture operates by receiving a textual prompt and a numerical engagement-time vector as input. The text prompt guides the semantic content of the generation, while the engagement-time vector modulates the learned stylistic adaptations within the U-Net. This modulation is achieved by dynamically scaling the contributions of numerous low-rank adaptation matrices embedded throughout the U-Net, thereby influencing the denoising process to align with desired engagement and temporal characteristics.

3.1.1 Engagement Conditioning Module (ECM)

The ECM serves as the primary mechanism for translating abstract conditioning signals into concrete, dynamic scaling factors. It is realised as a compact Multi-Layer Perceptron (MLP), formally denoted as f_ψ.

- **Input:** The ECM receives a two-dimensional time-aware engagement vector, $e_i = [\text{normalized\_engagement}, \text{normalized\_time}]$, as its input. Both components are normalised to a $[0, 1]$ range.
- **Architecture:** The ECM's internal structure consists of a sequence of linear layers interspersed with ReLU activations, culminating in a final linear layer with no explicit activation function (effectively an identity activation). All internal computations and parameter dtypes within the ECM are explicitly maintained in `torch.float32` for precision consistency. The hidden dimension of the ECM is 128.
- **Output:** The ECM generates a set of dynamic, scalar scaling factors, s_k, where k corresponds to the total number of LoRA layers integrated into the U-Net.

3.1.2 EC-LoRA-Linear Layer

The standard linear layers and `Conv1D` modules within the U-Net's critical components (e.g., attention blocks, feed-forward networks) are replaced by custom `EC_LoRA_Linear` layers. Each `EC_LoRA_Linear` instance is a bespoke LoRA implementation designed for dynamic scaling.

- **Structure:** Each `EC_LoRA_Linear` layer internally stores the original pre-trained weights (`original_weight`) as frozen, non-trainable parameters. It introduces two trainable low-rank matrices, $A \in R^{d \times r}$ and $B \in R^{r \times k}$, where r is the LoRA rank. Both A and B matrices are explicitly initialised and maintained in `torch.float32` precision.

- **Dynamic Scaling Mechanism:** Crucially, each `EC_LoRA_Linear` instance maintains a transient attribute, `_current_dynamic_batch_scale`. This attribute is populated with the specific scalar scaling factor s_k (derived from the ECM's output for that particular LoRA layer and batch) just prior to its forward pass. The LoRA update, BAx, is then scaled by this dynamic factor s_k in addition to the static α/r scaling, influencing the overall layer output: $W_{\text{new}}x = Wx + s_k \cdot (\alpha/r) \cdot (AB)x$.

3.1.3 LoRA Layer Injection and U-Net Patching

The integration of `EC_LoRA_Linear` layers and the dynamic conditioning mechanism requires precise modification of the U-Net architecture and its forward computation.

- **Layer Injection:** A recursive traversal function, `replace_module_recursively`, identifies and replaces specific target linear modules within the U-Net. These targets are typically found within attention projection layers (`to_q`, `to_k`, `to_v`, `to_out.0`) and feed-forward networks (`ff.net.0.proj`, `ff.net.2`). During this process, the base U-Net parameters are frozen, ensuring that only the injected LoRA matrices and ECM weights are trainable.

- **U-Net Forward Pass Patching:** The U-Net's native `forward` method is dynamically replaced with a custom function, `ec_lora_unet_forward` (or `ec_lora_unet_forward_inference` for generation), using `types.MethodType`. This patched forward pass orchestrates the dynamic scaling:

  1. It receives the `engagement_time_vector` (passed via `cross_attention_kwargs` for inference, or as a direct argument for training).
  2. The `engagement_time_vector` is fed into the ECM to compute `batch_scaling_factors`.
  3. These `batch_scaling_factors` are then individually assigned to the `_current_dynamic_batch_scale` attribute of each `EC_LoRA_Linear` layer within the current batch.
  4. The original U-Net's forward logic is then executed.
  5. Crucially, after the forward pass, the `_current_dynamic_batch_scale` attributes are reset to `None` to prevent state leakage between batches.

  All tensors manipulated within this patched forward pass are explicitly cast to `torch.float32` to maintain consistent precision throughout the model's computation graph.

This integrated architecture ensures that LoRA adaptations remain dynamic, continuously adjusted by engagement and temporal context at inference time. This enables finer control over the generated content.

3.2 Dataset Collection and Characteristics

The dataset for this project is a custom-curated collection from the Instagram account of Korean male influencer sungccess, used with his explicit consent. It features a diverse yet stylistically consistent set of images along with associated engagement metrics (e.g., likes and comments), supporting the training and evaluation of the EC-LoRA-T model.

3.2.1 Data Acquisition and Volume

The dataset comprises approximately 200 Instagram posts, many of which contain multiple high-quality photographs of the influencer. This results in a diverse collection of images varying in content, pose, and environment. All images are stored in a hierarchical directory structure, organised by post, to facilitate efficient access during data processing and training.

3.2.2 Raw Data Structure and Metadata

Each individual social media post is encapsulated within its own subdirectory, identified by a unique `post_shortcode`. Within each of these post-specific folders, the primary image is stored as a `.jpg` file. Accompanying each image is an individual `metadata.csv` file. This metadata file, manually compiled during the collection phase, contains essential quantitative information pertinent to the respective post:

- `shortcode`: A unique identifier for the social media post.
- `likes_count`: The total number of 'likes' received by the post.
- `comments_count`: The total number of 'comments' on the post.
- `timestamp`: The Unix timestamp indicating when the post was published.

Crucially, it is noted that these initial `metadata.csv` files *do not contain original captions*. The textual conditioning for this project is entirely derived from machine learning-generated descriptions, as detailed in Section 3.3.

```
.
sungccess/
   DCByOxRTVAa/
      465092590_8646157025441238_255656975607481380_n.jpg
      metadata.csv
   DCoD0lzvaa1/
      ...
   ... (other post folders and images)
```

Figure 3.1: Raw project directory structure for the collected Instagram post data. This hierarchical structure supports automated metadata consolidation and dataset preparation for the EC-LoRA-T training pipeline.

3.3 Enhanced Text Conditioning: ML-based English Description Generation

A critical aspect of this project's data preparation involved addressing the absence of original textual captions within the collected `sungccess` dataset. To ensure consistent, high-quality, and machine-readable textual conditioning for the diffusion model, an approach leveraging machine learning-based description generation was adopted.

3.3.1 Vision-Language Model (VLM) for Captioning

Instead of relying on potentially noisy or inconsistent user-generated captions, a pre-trained Vision-Language Model (VLM), specifically the Salesforce/blip-image-captioning-base model, was employed. This VLM possesses the capability to analyse an input image and generate a concise, descriptive English sentence encapsulating its visual content. The process involved:

1. Loading the pre-trained BLIP model and its associated processor.
2. Iterating through each image in the `sungccess` dataset.
3. Passing the raw image data through the BLIP model's inference pipeline to obtain an initial English description.

3.3.2 Custom Tagging and Temporal Context Integration

To further enhance the text conditioning and imbue it with project-specific stylistic and temporal signals, each ML-generated English description was augmented with two distinct custom tags:

- **Influencer Style Tag:** Each generated description was programmatically prefixed with the influencer's specific style tag: (`sungccess style`). This explicit token provides a direct conditioning signal to the model, guiding it towards the learned aesthetic associated with the `sungccess` persona.
- **Season Tag:** To incorporate crucial temporal context, the relevant season was appended to each description. The season was determined based on the image's Unix timestamp, following a predefined mapping for the Northern Hemisphere:
  - Spring: March - May
  - Summer: June - August
  - Fall: September - November
  - Winter: December - February

By integrating seasonal information, the model can potentially learn and reflect time-dependent visual traits such as clothing, lighting, or environment. Each image is paired with a structured `full_caption`, combining a style tag and a season tag, which serves as the conditioning input. This enriched textual prompt provides clear and controllable guidance to the diffusion model during generation.

3.4 Revised Engagement Calculation and Normalisation

To accurately represent and quantify audience interaction for conditioning the generative model, a revised methodology for calculating and normalising engagement metrics was implemented.

3.4.1 Weighted Total Engagement

`total_engagement` was calculated using a weighted formula to reflect different levels of user effort. Since comments often indicate deeper engagement than passive likes, they were given higher weight. The formula used is:

$$\text{total\_engagement} = \text{likes\_count} + (3 \times \text{comments\_count})$$

3.4.2 Min-Max Scaling for Normalisation

To ensure smooth integration into the Engagement Conditioning Module (ECM) and maintain numerical stability during training, the `total_engagement` scores were normalised using min-max scaling, mapping all values to a 0–1 range. This standardisation ensures that engagement signals of different magnitudes contribute proportionally to the model's conditioning input. The normalisation follows the formula:

$$\text{normalized\_engagement} = \frac{\text{total\_engagement} - \min(\text{total\_engagement})}{\max(\text{total\_engagement}) - \min(\text{total\_engagement})}$$

This `normalized_engagement` value forms the first component of the two-dimensional `engagement_time_vector`, providing a consistent and comparable measure of audience interaction across the entire dataset.

3.5 Time Variance Integration

3.5.1 Derivation of Normalised Time Feature

The raw temporal data for each post was initially available as a Unix `timestamp` within the collected metadata. To convert this into a continuous and meaningful feature suitable for neural network consumption, the following steps were performed:

1. **Timestamp Conversion:** Each Unix `timestamp` was converted into a `datetime` object, allowing for precise chronological calculations.

2. **Days Since Earliest Post:** The earliest `datetime` in the entire dataset was identified as the baseline (`min_timestamp`). For every subsequent post, the number of days elapsed since this `min_timestamp` was calculated. This yielded a raw `days_since_earliest_post` metric, representing each post's temporal position relative to the beginning of the dataset.

3.5.2 Min-Max Scaling for Normalised Time

Like the engagement scores, the `days_since_earliest_post` values were normalised using min-max scaling, converting them to a 0–1 range. This ensures the temporal feature is consistent and doesn't exert disproportionate influence due to large raw values. The normalisation is given by:

$$\text{normalized\_time} = \frac{\text{days\_since\_earliest\_post} - \min(\text{days\_since\_earliest\_post})}{\max(\text{days\_since\_earliest\_post}) - \min(\text{days\_since\_earliest\_post})}$$

The `normalized_time` value forms the second key component of the `engagement_time_vector`, providing a continuous temporal signal to guide the conditioning mechanism.

3.5.3 Formation of Engagement-Time Vector

The final `engagement_time_vector`, e_i, which serves as the primary input to the Engagement Conditioning Module (ECM), is a two-dimensional vector formed by the concatenation of the `normalized_engagement` and `normalized_time` features for each individual post:

$$e_i = [\text{normalized\_engagement}, \text{normalized\_time}]$$

This composite vector captures both audience engagement and temporal context, allowing the ECM to produce dynamic scaling factors that guide the generative process in a time-aware, engagement-conditioned way.

3.6 Data Pipeline for Training

This section outlines the data pipeline that transforms raw social media data into a stable, consistent format optimized for EC-LoRA-T training.

3.6.1 Data Consolidation and Feature Engineering

The first step was to combine all the raw data into one unified dataset. Individual `metadata.csv` files, stored in separate post folders, were merged into a single Pandas DataFrame. As part of this process, `image_path` was linked to its metadata, ensuring that every data entry is directly connected to its corresponding visual content.

Subsequently, the consolidated DataFrame was refined through feature engineering, adding enhanced text conditioning and updated engagement metrics to prepare the data for model training.

- **ML-generated `full_caption`:** As described in Section 3.3, each image's associated caption was derived from an ML-based Vision-Language Model, augmented with a `(sungccess style)` tag and a season tag.
- `normalized_engagement`: The weighted total engagement was computed and min-max scaled to a $[0, 1]$ range, as detailed in Section 3.4.
- `normalized_time`: The temporal information, derived from timestamps, was converted to `days_since_earliest_post` and similarly min-max scaled to a $[0, 1]$ range, as elaborated in Section 3.5.

3.6.2 Hugging Face Dataset Integration

To facilitate efficient data handling and seamless integration with the `diffusers` training framework, the Pandas DataFrame was converted into a Hugging Face `Dataset` object. This conversion explicitly defined the features, ensuring the `image` column was loaded as a `PIL.Image.Image` object from its path, and the `full_caption`, `normalized_engagement`, and `normalized_time` were correctly typed as `string`, `float32`, and `float32` respectively. This approach leverages the `datasets` library's optimisations for large-scale data management.

3.6.3 Image Transformations and Text Tokenisation

Prior to model ingestion, raw image data and textual captions underwent specific preprocessing steps:

- **Image Transformations:** The `make_train_transforms` function applies a sequence of `torchvision.transforms`:

  1. `Resize(512, interpolation=BILINEAR)`: Images were resized to 512×512 pixels, aligning with the VAE's expected input resolution.
  2. `CenterCrop(512)`: A central crop ensured consistent spatial dimensions.
  3. `ToTensor()`: Converts PIL images to PyTorch tensors, normalising pixel values to $[0, 1]$.
  4. `Normalize([0.5], [0.5])`: Scales pixel values from $[0, 1]$ to $[-1, 1]$, which is the standard input range for the Stable Diffusion U-Net.

  All pixel value tensors were explicitly cast to `torch.float32` to maintain precision consistency throughout the pipeline.

- **Text Tokenisation:** The `tokenize_captions` function, leveraging the `CLIPTokenizer` (loaded from `runwayml/stable-diffusion-v1-5`), converts textual captions into fixed-length sequences of token IDs. It applies padding and truncation to `tokenizer.model_max_length` and returns PyTorch tensors, preparing them for the CLIP Text Encoder.

3.6.4 Custom Dataset and DataLoader Setup

The prepared Hugging Face `Dataset` was interfaced with the PyTorch training loop via a custom `ECLoRATDataset` class and a `DataLoader`.

- `ECLoRATDataset`: This custom dataset class inherits from `torch.utils.data.Dataset`. Its `__getitem__` method is responsible for:

  1. Retrieving the pre-loaded `PIL.Image` object directly from the Hugging Face dataset.
  2. Applying the defined image transformations to obtain `pixel_values`.
  3. Fetching the `full_caption` and converting it to `input_ids` via `tokenize_captions`.
  4. Constructing the `engagement_time_vector` by combining `normalized_engagement` and `normalized_time` into a `torch.float32` tensor.

- `collate_fn`: A custom collate function was used to batch data samples and filter out any `None` entries (e.g., from loading errors), ensuring each batch remained clean and reliable.

- `DataLoader`: The `DataLoader` was configured with a specified `BATCH_SIZE` (e.g., 4), `NUM_WORKERS` (e.g., 4 for parallel data loading), `shuffle=True` for randomness across epochs, and `pin_memory=True` to accelerate data transfer to the GPU. This setup efficiently delivers processed batches to the training loop.

Chapter 4

Implementation

4.1 Development Environment and Tools

The development and execution of the EC-LoRA-T project were primarily conducted within the Google Colaboratory (Colab) environment, leveraging its cloud-based computational resources. This section outlines the specific tools, libraries, and environmental configurations employed throughout the project lifecycle.

4.1.1 Computational Infrastructure

- **Google Colaboratory:** Colab was selected as the primary development platform due to its accessible Python environment, integrated Jupyter Notebook interface, and direct provision of Graphical Processing Units (GPUs).
- **Google Colab Pro:** The project utilised Colab Pro instances to access higher-tier GPU resources, specifically NVIDIA A100 Tensor Core GPUs, which are essential for accelerating deep learning computations, coupled with increased RAM capacity for handling large models and datasets.
- **Google Drive:** All project data, including the `sungccess` image dataset, processed Hugging Face datasets, Python scripts, and trained model checkpoints, were persistently stored and accessed via Google Drive, ensuring data integrity and session resilience.
- **Programming Language:** Python was the sole programming language employed for all aspects of the project, including data processing, model implementation, training, and evaluation.

4.1.2 Key Libraries and Frameworks

The project extensively relied on a suite of open-source Python libraries, primarily from the deep learning and scientific computing ecosystems:

- **PyTorch:** As the foundational deep learning framework, PyTorch was used for all neural network constructions, tensor operations, and model training. A strict `torch.float32` precision environment was maintained throughout the model's forward and backward passes.
- **Hugging Face `diffusers`:** This library provided the core components of the Stable Diffusion v1.5 model, including the `UNet2DConditionModel`, `AutoencoderKL` (VAE),

and `DDPMScheduler`. Its modular design facilitated the custom architectural modifications required for EC-LoRA-T.

- **Hugging Face `transformers`:** Utilised for accessing the `CLIPTextModel` (for text encoding) and the `CLIPTokenizer`, as well as the `Salesforce/blip-image-captioning-base` Vision-Language Model for automated English caption generation.

- **Hugging Face `datasets`:** Employed for efficient loading, processing, and management of the `sungccess` dataset, optimising data throughput for training.

- **`accelerate`:** A library from Hugging Face designed to simplify the training loop for PyTorch models, managing device placement, gradient accumulation, and enabling distributed training (though primarily used for single-GPU setup with gradient accumulation in this project).

- **`torchvision` and `Pillow`:** Essential for image loading, preprocessing, and applying various transformations required to prepare images for model input.

- **`scikit-learn`:** Used for developing and evaluating the linear regression model in the quantitative assessment phase, specifically for feature scaling, data splitting, and performance metrics.

- **`pandas` and `numpy`:** Core libraries for data manipulation, cleaning, and numerical operations across the dataset preparation and evaluation stages.

4.1.3 Environment and Version Management

A notable challenge throughout the project's development was ensuring compatibility across various Python libraries, particularly given the rapid evolution of the `diffusers` and `transformers` ecosystems. Persistent `ImportError`s (e.g., related to `huggingface_hub`'s `cached_download` function) and `RuntimeError`s (e.g., dimension mismatches within the U-Net's attention mechanism after custom patching, and warnings from `AttnProcessor2_0`) required a meticulous approach to environment management.

To maintain a stable and reproducible environment, a strategy of explicitly pinning library versions was adopted. This involved:

- Uninstalling existing `torch`, `torchvision`, and `torchaudio` to prevent conflicts.

- Installing PyTorch and its associated libraries with explicit CUDA version alignment (e.g., `cu121` for CUDA 12.x).

- Installing core Hugging Face libraries (`diffusers`, `transformers`, `accelerate`, `huggingface_hub`) to specific versions (e.g., `diffusers==0.20.2`, `transformers==4.30.2`, `accelerate==0.21.0`, `huggingface_hub==0.16.4`) that were verified to be compatible with the custom LoRA injection and the project's `float32` precision requirements.

- Consistently restarting the Colab runtime after each library installation step to ensure changes took effect.

This rigorous version control was crucial in resolving deep-seated incompatibilities and establishing a functional pipeline for model development and training.

4.2 Core Model Integration

The foundation of the EC-LoRA-T framework is built upon the robust architecture of the pre-trained Stable Diffusion v1.5 model. This section illustrates the integration of its core components, outlining their individual roles and initial configuration within the project's `float32` computational environment.

4.2.1 Stable Diffusion Components

The Stable Diffusion pipeline comprises several interconnected modules, each performing a specialized function within the latent diffusion process:

- **Tokenizer (`CLIPTokenizer`):** This component is responsible for converting human-readable text prompts into a sequence of numerical token IDs. These tokens form the initial representation of the textual conditioning information.
- **Text Encoder (`CLIPTextModel`):** Following tokenisation, the text encoder (a Transformer-based model derived from CLIP) transforms the token IDs into a rich, high-dimensional contextual embedding. This embedding serves as the semantic guidance for the image generation process, directing the U-Net towards desired visual attributes.
- **Variational Autoencoder (VAE) (`AutoencoderKL`):** The Variational Autoencoder (VAE) has two main components: an encoder and a decoder. During training, the encoder compresses high-resolution images (e.g., 512×512 pixels) into lower-dimensional latent representations (e.g., 64×64) that retain perceptually important features. During inference, the decoder reconstructs the final image from these denoised latent representations.
- **U-Net (`UNet2DConditionModel`):** This is the central neural network within the diffusion process. It is trained to iteratively predict and subtract noise from noisy latent representations, gradually transforming random noise into a coherent image latent. The U-Net's architecture incorporates cross-attention layers that integrate the text embeddings, allowing it to generate images conditioned by prompts.
- **Noise Scheduler (`DDPMScheduler`):** The noise scheduler controls both the forward (noising) and reverse (denoising) diffusion processes. It defines parameters—such as the noise magnitude at each timestep—that guide the progressive corruption of the input during the forward pass and support the U-Net's denoising predictions during the reverse pass.

4.2.2 Loading and Initial Configuration

All core Stable Diffusion components were loaded from the pre-trained `runwayml/stable-diffusion-v1-5` model checkpoint. A critical aspect of their initialisation involved ensuring adherence to the project's strict `torch.float32` precision requirement. Each component was explicitly loaded with `torch_dtype=torch.float32` to maintain numerical stability and consistency throughout the computational graph.

When the models were loaded, they went through an initial setup phase:

- **Device Placement:** The `VAE` and `Text Encoder` were immediately moved to the GPU (CUDA device), leveraging hardware acceleration for their operations.

- **Evaluation Mode and Parameter Freezing:** Both the `VAE` and `Text Encoder` were set to evaluation mode (`.eval()`) and their parameters were frozen (`.requires_grad_(False)`). This is standard practice as these components are pre-trained feature extractors and are not fine-tuned during the LoRA adaptation process.
- **U-Net Initialisation:** The `UNet` was also loaded in `torch.float32` precision and moved to the GPU. Its base parameters were subsequently frozen; only the custom `EC_LoRA_Linear` layers, injected as part of the EC-LoRA-T architecture (detailed in Section 4.4), and the `EngagementConditioningModule` (Section 4.3), are designated as trainable.

This comprehensive initial setup ensures that the foundational Stable Diffusion model is correctly configured, residing on the appropriate hardware, and prepared for the integration of the custom EC-LoRA-T modules.

4.3 Custom Module Implementation

The effectiveness of the EC-LoRA-T framework depends on the careful design and implementation of two key trainable modules: the `EC_LoRA_Linear` layer and the `EngagementConditioningModule` (ECM). These components, developed with a strict adherence to `torch.float32` precision, facilitate the dynamic conditioning of the generative process.

4.3.1 EC-LoRA-Linear Layer

The `EC_LoRA_Linear` class (`nn.Module`) serves as a direct, drop-in replacement for standard `nn.Linear` and `transformers.pytorch_utils.Conv1D` layers within the U-Net architecture. Its design integrates the principles of LoRA with the novel dynamic scaling mechanism.

- **`__init__`(self, original_layer, lora_rank, lora_alpha, ecm_idx):**
  - `original_layer`: The pre-existing `nn.Linear` or `Conv1D` module from the U-Net that this custom layer will replace. The constructor dynamically identifies its `in_features` and `out_features`.
  - `lora_rank` (r): An integer specifying the bottleneck dimension for the LoRA matrices.
  - `lora_alpha` (α): An integer defining the static scaling factor for the LoRA update.
  - `ecm_idx`: A unique integer index assigned to each `EC_LoRA_Linear` instance, corresponding to its position in the ECM's output vector of scaling factors.
  - **Frozen Original Weights:** The `weight` and `bias` (if present) of the `original_layer` are cloned, explicitly cast to `torch.float32`, and stored as `nn.Parameter`s with `requires_grad=False`. These constitute the immutable base layer.
  - **Trainable LoRA Matrices:** Two new `nn.Parameter`s, `lora_A` ($\in R^{\text{in\_features} \times r}$) and `lora_B` ($\in R^{r \times \text{out\_features}}$), are initialised. These matrices are explicitly created on the CUDA device with `torch.float32` dtype and `requires_grad=True`. `lora_A` is initialised using Kaiming uniform distribution, while `lora_B` is initialised to zeros.

- `static_lora_scaling`: Computed as `lora_alpha / lora_rank`.
- `_current_dynamic_batch_scale`: A transient attribute, initialised to `None`, designed to store the batch-specific dynamic scaling factor provided by the ECM during the U-Net's patched forward pass.

- `forward`**(self, x: torch.Tensor) -> torch.Tensor:**
  - The input tensor `x` is ensured to be in `torch.float32`.
  - **Base Output:** The output from the frozen original layer is computed via `F.linear(x, self.original_weight, self.original_bias)`.
  - **LoRA Update:** The low-rank update is calculated as `torch.matmul(torch.matmul(x, self.lora_A), self.lora_B)`.
  - **Scaling Application:** This `lora_update` is first scaled by `self.static_lora_scaling`. If `_current_dynamic_batch_scale` is set (i.e., by the ECM), the `lora_update` is further scaled by this dynamic factor, ensuring correct broadcasting across batch and sequence dimensions.
  - **Final Output:** The `base_output` is added to the dynamically scaled `lora_update`, yielding the final output of the `EC_LoRA_Linear` layer. All internal tensor operations are implicitly performed in `torch.float32`.

4.3.2 Engagement Conditioning Module (ECM)

The ECM is an essential component responsible for generating the dynamic scalar scaling factors that modulate the `EC_LoRA_Linear` layers.

- `__init__`**(self, input_dim, output_dim, hidden_dim=128):**
  - `input_dim`: Set to 2, corresponding to the dimensions of the `engagement_time_vector` (normalised engagement, normalised time).
  - `output_dim`: Set to the total number of `EC_LoRA_Linear` layers injected into the U-Net (e.g., 128 layers). Each dimension of the ECM's output corresponds to a unique scaling factor for one LoRA layer.
  - **Network Architecture:** The ECM is implemented as a `nn.Sequential` stack of `nn.Linear` layers. It typically comprises two hidden layers with `ReLU` activations, followed by a final linear layer (with no explicit activation function, effectively an Identity activation). All `nn.Linear` layers within the ECM are explicitly defined with `dtype=torch.float32`.

- `forward`**(self, engagement_time_vector) -> torch.Tensor:**
  - The input `engagement_time_vector` is ensured to be in `torch.float32`.
  - The method passes the `engagement_time_vector` through the defined sequential network and returns the resulting batch of scaling factors.

Together, these custom modules form the trainable core of the EC-LoRA-T framework, enabling parameter-efficient and dynamically conditioned generation within the Stable Diffusion architecture.

4.4 LoRA Layer Injection and U-Net Forward Patching

The EC-LoRA-T framework relies on smoothly integrating the custom `EC_LoRA_Linear` layers into the pre-trained U-Net and modifying its forward pass. This section explains how the LoRA layers are injected and how the U-Net's forward method is patched to support the new architecture.

4.4.1 LoRA Layer Injection Process

To selectively adapt the U-Net for engagement and time conditioning, specific `nn.Linear` and `transformers.pytorch_utils.Conv1D` layers were identified as targets for replacement by `EC_LoRA_Linear` instances. This process is initiated after the base U-Net parameters have been frozen, ensuring that only the injected LoRA components are trainable.

- **Target Module Identification:** The selection of layers for LoRA injection is guided by a predefined set of `target_linear_module_patterns`. These patterns correspond to key projection and feed-forward layers within the U-Net's transformer blocks, specifically: `"to_q"`, `"to_k"`, `"to_v"` (query, key, and value projections in attention mechanisms), `"to_out.0"` (the output projection of attention blocks), `"ff.net.0.proj"`, and `"ff.net.2"` (layers within feed-forward networks).
- **Recursive Replacement Function:** A custom recursive function, `replace_module_recursively`, traverses the U-Net's module hierarchy. For each module encountered, it checks if it is an instance of `nn.Linear` or `Conv1D` and if its full hierarchical name matches any of the `target_linear_module_patterns`.
- **Layer Replacement and Indexing:** Upon identifying a target module, it is replaced at its original location with a new `EC_LoRA_Linear` instance. Each `EC_LoRA_Linear` is assigned a unique `ecm_idx` (managed by a global counter, `ecm_idx_counter_list`), which links it to a specific output dimension of the Engagement Conditioning Module (ECM). All newly injected layers are immediately moved to the CUDA device. References to these injected layers are stored in a global list, `all_ec_lora_linear_layers`, crucial for later access during the patched forward pass.

4.4.2 U-Net Forward Pass Patching

To integrate the dynamic scaling functionality of the ECM and `EC_LoRA_Linear` layers, the U-Net's standard `forward` method is overridden by a custom function. This ensures that the `engagement_time_vector` actively modulates the U-Net's behaviour during both training and inference.

- **Method Replacement:** The U-Net's `forward` method is replaced using `types.MethodType` with a custom function, `ec_lora_unet_forward` (or `ec_lora_unet_forward_inference` for generation). A reference to the original U-Net's unbound class method (`UNet2DConditionModel.forward`) is retained to enable its execution within the patched function.
- **Custom Forward Logic:** The patched function receives the standard U-Net inputs (`sample`, `timestep`, `encoder_hidden_states`) along with the critical `engagement_time_vector`.
  - **Argument Handling:** During training, `engagement_time_vector` is passed as a direct positional argument. For inference, it is passed within the `cross_attention_kwargs` dictionary.

- **ECM Invocation:** The `engagement_time_vector` is processed by the `engagement_conditioning_module` (ECM) to yield `batch_scaling_factors`.
- **Dynamic Scale Distribution:** These `batch_scaling_factors` are then precisely distributed. For each `EC_LoRA_Linear` layer, its corresponding scalar from `batch_scaling_factors` is assigned to its transient `_current_dynamic_batch_scale` attribute. This assignment occurs prior* to the original U-Net's forward pass.
- **Original Forward Execution:** The original U-Net's `forward` method is then called, and as it internally executes the `EC_LoRA_Linear` layers, they apply their dynamically set scales.
- **Cleanup:** Crucially, after the original forward pass completes, the `_current_dynamic_batch_scale` attribute of all `EC_LoRA_Linear` layers is reset to `None` to prevent state leakage between subsequent batches.

- **Precision Consistency:** All tensors involved in the patched forward pass, including inputs and intermediate computations, are explicitly cast to `torch.float32` to maintain the project's strict precision requirements.

This comprehensive patching strategy enables the EC-LoRA-T model to dynamically modulate its generative output based on continuous engagement and temporal signals, which is central to its core functionality.

4.5 Training Configuration

Training EC-LoRA-T required a clear set of hyperparameters and environment setups. This section covers the configurations I used during optimisation, including the tweaks made to help the model better learn from engagement signals and time-based patterns.

4.5.1 Hyperparameters

The following hyperparameters were set to control how the EC-LoRA-T model learns during training:

- **LoRA Parameters:**
  - **LoRA Rank (`LORA_RANK`):** Set to `16`. A higher rank allows the LoRA matrices to capture more complex and nuanced adaptations, providing greater capacity for the model to embed the intricate visual features associated with varying engagement levels.
  - **LoRA Alpha (`LORA_ALPHA`):** Set to `64`. This parameter scales the magnitude of the LoRA updates. Maintaining a ratio where `LORA_ALPHA` is often $2\times$ or $4\times$ `LORA_RANK` ensures effective scaling of the learned adaptations.
- **Training Schedule Parameters:**
  - **Number of Epochs (`NUM_EPOCHS`):** Set to `200`. Increasing the number of epochs provides the model with more iterative opportunities to converge and learn the subtle correlations between visual features and the engagement-time vector.
  - **Learning Rate (`LEARNING_RATE`):** A learning rate of `1e-4` was utilised. This value was chosen as a common starting point for fine-tuning large models.

- **Batching Parameters:**
  - **Batch Size (`BATCH_SIZE`):** Set to `4` samples per mini-batch. This value balances GPU memory consumption with training efficiency, particularly for 512×512 pixel image inputs in a `float32` environment.
  - **Gradient Accumulation Steps (`GRADIENT_ACCUMULATION_STEPS`):** Set to `1`. While this does not simulate a larger effective batch size, it simplifies the training loop by ensuring gradient updates occur after every micro-batch.
- **Precision Setting:**
  - **Mixed Precision (`MIXED_PRECISION`):** Explicitly set to `"no"`. This configuration mandates that all model parameters and tensor operations are performed in full `torch.float32` precision. This was a deliberate choice to prioritise numerical stability and consistency throughout the custom architectural modifications, rather than opting for memory efficiency or speed benefits offered by lower precision formats like `fp16` or `bf16`.
- **Reproducibility:**
  - **Random Seed (`SEED`):** A fixed seed of `42` was employed to ensure reproducibility of random operations, including weight initialization and data shuffling, thereby facilitating consistent experimental results.

4.5.2 Optimizer and Learning Rate Scheduler

The optimisation process was orchestrated using standard deep learning components:

- **Optimizer:** The `AdamW` optimizer was selected for its robust performance in training large neural networks. It was configured with default betas of (0.9, 0.999), a weight decay of `1e-2`, and an epsilon of `1e-8`. Crucially, the optimizer was configured to update only the trainable parameters, specifically those of the injected `EC_LoRA_Linear` layers and the `EngagementConditioningModule`.
- **Learning Rate Scheduler:** A `cosine` learning rate scheduler was employed. This scheduler is designed to gradually reduce the learning rate following a cosine curve throughout the training process, often leading to better convergence. A warm-up period, set to 10% of the total training steps, was implemented to allow the model to stabilise early in training before the learning rate decay commences.

4.5.3 Accelerator Integration and Output Management

The Hugging Face `accelerate` library played a central role in simplifying the training loop and managing the computational environment.

- `accelerate.Accelerator`: The `Accelerator` object was initialised with the specified `MIXED_PRECISION` and `GRADIENT_ACCUMULATION_STEPS`. It facilitates automatic device placement, gradient accumulation handling, and logging integration with TensorBoard.
- `accelerator.prepare()`: This critical function wrapped the `U-Net`, `optimizer`, `train_dataloader`, `lr_scheduler`, and `engagement_conditioning_module`, effectively moving them to the appropriate GPU device and preparing them for the training regimen.

- **Output Directory (`OUTPUT_DIR`):** All training outputs, including model checkpoints (containing both LoRA and ECM weights), were systematically saved to a designated directory within Google Drive. Checkpoints were periodically saved at the end of every epoch, ensuring a granular record of the training progress and enabling easy resumption or post-training analysis.

This comprehensive configuration aims to provide a stable, efficient, and reproducible training environment for optimising the EC-LoRA-T model's capacity to learn subtle engagement-conditioned aesthetics.

4.6 Training Loop Implementation

The core of the EC-LoRA-T project lies within its iterative training loop, which orchestrates the forward and backward passes to optimise the trainable parameters of the LoRA layers and the Engagement Conditioning Module (ECM). This section details the sequence of operations performed within each training iteration.

4.6.1 Epoch and Batch Processing

The training process is structured as an outer loop iterating through a predefined number of `NUM_EPOCHS` (e.g., 200). Within each epoch, an inner loop traverses batches of data provided by the `train_dataloader`. A progress bar (`tqdm`) is employed to monitor training progress, and the `U-Net` is explicitly set to training mode (`unet.train()`) at the beginning of each epoch. Robust batch handling includes a check for empty batches, which are skipped if encountered.

4.6.2 Forward Pass Computation

Each batch undergoes the following sequence of computations within an `accelerator.accumulate(unet)` context, which manages gradient accumulation based on `GRADIENT_ACCUMULATION_STEPS`:

1. **Image Encoding to Latent Space:**
    - Pixel values from the batch, originally normalised to $[-1, 1]$, are optionally scaled to $[0, 1]$ (`(pixel_values + 1.0) / 2.0`) before being explicitly cast to `torch.float32`.
    - The `VAE` then encodes these pixel values into a compressed latent representation.
    - Latent features are sampled from the VAE's latent distribution (`latent_dist.sample()`) and scaled by `vae.config.scaling_factor`. The resulting `latents` are ensured to be in `torch.float32` precision.
2. **Noise Sampling:**
    - Gaussian noise of the same shape as the `latents` is sampled using `torch.randn_like()` and explicitly cast to `torch.float32`.
3. **Timestep Sampling:**
    - For each sample in the batch, a random timestep is sampled uniformly from the range defined by the `noise_scheduler`'s number of training timesteps. The timesteps are explicitly converted to `long` type.

- **Noisy Latent Creation:** The sampled noise is added to the `latents` according to the noise schedule. This is performed manually by:
  - Retrieving `alphas_cumprod` from the `noise_scheduler`.
  - Calculating `sqrt_alpha_prod` and `sqrt_one_minus_alpha_prod` for the sampled timesteps, ensuring these are on the correct device and in `torch.float32`.
  - Broadcasting these factors across the latent dimensions via `view(-1, 1, 1, 1)`.
  - Computing `noisy_latents` using the formula: `noisy_latents = sqrt_alpha_prod * latents + sqrt_one_minus_alpha_prod * noise`.

4. **Text Embeddings Generation:**
   - The `input_ids` (tokenised captions) from the batch are moved to the `text_encoder`'s device.
   - The `text_encoder` (operating within a `torch.no_grad()` context as its parameters are frozen) transforms these IDs into `encoder_hidden_states`, explicitly cast to `torch.float32`.
5. **Engagement-Time Vector Preparation:**
   - The `engagement_time_vector` from the batch is moved to the `accelerator.device` and ensured to be in `torch.float32`.
6. **U-Net Noise Prediction:**
   - The core of the forward pass involves invoking the `U-Net` with the prepared inputs: `noisy_latents`, `timesteps`, `encoder_hidden_states`, and the crucial `engagement_time_vector`.
   - The call syntax explicitly passes `engagement_time_vector` as a direct positional argument, aligned with the patched `ec_lora_unet_forward` signature (as established in Section 4.4). The `return_dict=False` argument ensures the U-Net returns a tuple, from which the predicted noise tensor is extracted.

4.6.3 Loss Calculation and Optimization

Following the U-Net's prediction, the model's trainable parameters are updated:

- **Loss Calculation:** The Mean Squared Error (`F.mse_loss()`) is computed between the `noise_pred` (predicted noise residual) and the original `noise` that was added to the latents. This loss function quantifies the model's accuracy in predicting the noise.
- **Backpropagation:** `accelerator.backward(loss)` performs the backpropagation step, computing gradients for all trainable parameters.
- **Parameter Update:** The `optimizer.step()` function updates the parameters of the `EC_LoRA_Linear` layers and the `ECM` using the calculated gradients.
- **Learning Rate Schedule:** The `lr_scheduler.step()` updates the learning rate according to the defined cosine schedule.
- **Gradient Reset:** `optimizer.zero_grad()` clears the gradients for the next iteration.

Chapter 5

Results and Evaluation

This chapter evaluates the EC-LoRA-T framework in terms of its ability to generate visually coherent outputs, capture the target influencer's style, and respond meaningfully to engagement and temporal conditioning. Both quantitative and qualitative methods were used, with a focus on validating the model's design assumptions and identifying its current limitations.

5.1 Evaluation Methodology

5.1.1 Objectives

The evaluation was structured around three core objectives:

- **Visual Quality and Aesthetic Coherence:** Assess the overall fidelity and aesthetic plausibility of generated images, ensuring they are free from visual artefacts.
- **Stylistic Adherence:** Determine whether the model effectively reproduces the unique visual characteristics of the `sungccess` influencer.
- **Effectiveness of Engagement and Temporal Conditioning:** Evaluate whether variations in `normalized_engagement` and `normalized_time` inputs lead to meaningful and consistent changes in the generated images.

5.1.2 Latent Regression as Diagnostic Tool

Before evaluating EC-LoRA-T's generation ability, a regression model was trained to validate whether engagement can be predicted from VAE-extracted latent features. This diagnostic step assesses whether engagement signals are even present in the visual space learned by the VAE.

A `LinearRegression` model was trained on flattened VAE latents from real training images, with their corresponding `normalized_engagement` scores. The model achieved a low mean squared error (MSE = 0.0003) and a high R-squared score ($R^2 = 0.9769$), confirming that engagement is strongly linearly correlated with visual features captured by the VAE. This result supports the feasibility of conditioning generation on engagement.

5.1.3 Quantitative Evaluation of EC-LoRA-T

To test whether EC-LoRA-T can generate images that reflect engagement conditioning, synthetic images were generated under controlled settings. The prompt was fixed to:

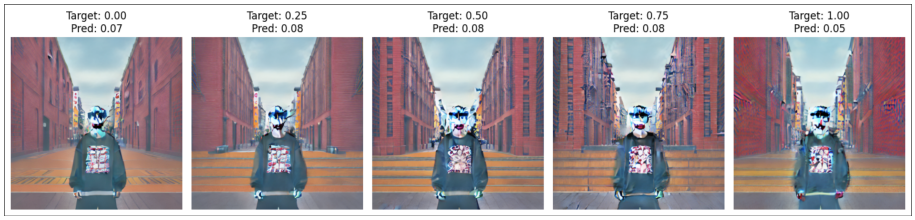


Figure 5.1: Quantitative Evaluation of EC-LoRA-T.

`"(sungccess style) a Korean man, confident and charismatic, dramatic lighting, wearing a fashionable oversized streetwear outfit, urban street scene, bokeh background, highly detailed, professional photography (summer)"`

A constant `normalized_time` value (0.5) and fixed random seed (42) were used, while the `normalized_engagement` input was varied across five values: 0.00, 0.25, 0.50, 0.75, 1.00.

Each generated image was passed through the VAE, and its latent representation was fed into the trained engagement regressor to obtain a predicted engagement score. The results are as follows:

Target Engagement	**Predicted Engagement**
0.00	0.0668
0.25	0.0767
0.50	0.0827
0.75	0.0809
1.00	0.0549

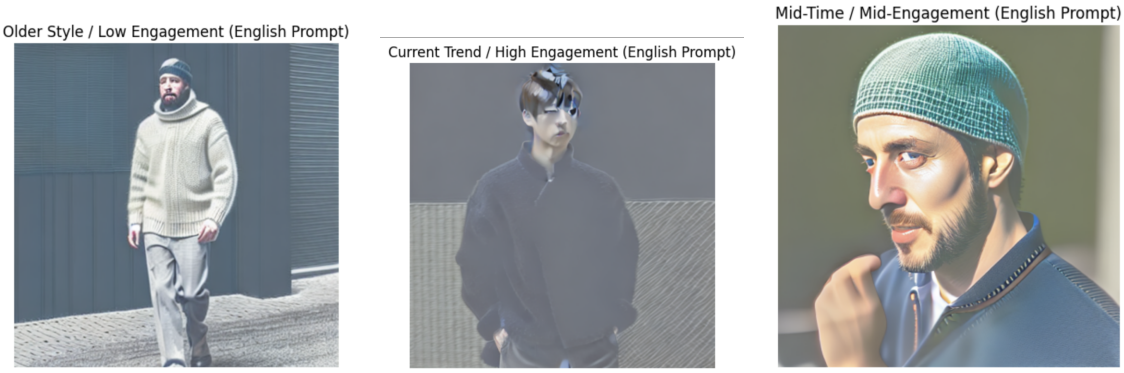
While the predictions show a mild upward trend from 0.00 to 0.50, they do not increase consistently thereafter. The predicted scores fall within a narrow band (approximately 0.05 to 0.08), indicating that the model failed to meaningfully embed the engagement signal into the generated images.

5.1.4 Qualitative Evaluation

Qualitative inspection supported the quantitative findings. Images were generated using English prompts under controlled conditions, with varying `normalized_engagement` and `normalized_time` vectors. While some variation is visible—particularly in clothing and facial structure—the observed differences were inconsistent and not strongly correlated with the intended conditioning inputs.

For example, the image generated with a low engagement and early-time vector shows a traditional, conservative fashion style, while the high engagement / recent time vector leads to a more modern appearance, though degraded by visible facial artifacts. The mid-level conditioning input yields a cleaner output but lacks distinctive stylistic progression from the others.

Overall, these qualitative results reinforce the quantitative analysis: despite architectural support for engagement and temporal conditioning, the generated outputs lack a consistent or interpretable response to these control signals. This indicates the need for further architectural or training improvements.



Prompt: `classic minimalist outfit (winter)`
Vector: [0.1, 0.1]
Conservative style, low expressiveness.

Prompt: `stylish asian man, vivid color (winter)`
Vector: [0.99, 0.99]
Trendier look but with facial artifacts.

Prompt: `man with a cap, natural light (spring)`
Vector: [0.5, 0.5]
Clean output, limited variation.

Figure 5.2: Qualitative evaluation of EC-LoRA-T with English prompts and varying engagement-time vectors. Despite controlled conditions, outputs show limited visual differentiation.

5.2 Summary of Findings

While the regression model showed that engagement could be predicted from latent features, the EC-LoRA-T model did not successfully reflect this in its image generation. Even with extended training, added LoRA layers, and engagement-conditioning mechanisms, changes in the input engagement vector had little visible impact on the output. The model struggled to translate abstract engagement scores into meaningful stylistic or compositional changes in the generated images.

This suggests that while the model architecture has sufficient capacity, further work is required on the conditioning mechanism, loss formulation, or training supervision to ensure the engagement input influences generation more directly.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

The EC-LoRA-T project set out to test whether generative diffusion models can be conditioned using implicit signals like social media engagement and temporal context, in a way that leads to controllable, visually distinct outputs. While the architecture and training pipeline were successfully implemented, and the model was trained at scale with meaningful latent supervision, the final results suggest a clear gap between theoretical potential and practical outcome.

The core framework (combining EngagementConditioningModule (ECM) and custom EC_LoRA_Linear layers) was integrated into a stable diffusion backbone with a functional custom forward pass. The data pipeline, which included ML-generated captions, structured metadata, and normalised conditioning vectors, worked reliably and supported the training loop without major bottlenecks.

On the evaluation side, the regression analysis confirmed that engagement is indeed a learnable signal in the VAE's latent space—achieving an R^2 of 0.9769. However, when the same conditioning was used during image generation, the EC-LoRA-T model failed to produce outputs that reflected those inputs in any consistent or interpretable way. The predicted engagement scores of generated images remained nearly flat across varying targets, and the images themselves were visually similar despite changing the conditioning vector.

This discrepancy highlights an important insight: just because a signal is learnable in isolation (via regression) doesn't mean it is being embedded effectively during generative training. That limitation frames both the outcome of this work and the direction for future development.

6.2 Future Work

While EC-LoRA-T did not fully achieve its original goal, it established a solid technical foundation to build on. The following directions offer concrete next steps:

6.2.1 Model-Level Improvements

- **Make ECM More Expressive:** Try deeper or non-linear architectures or explore multiplicative interactions that could push the model to learn stronger or even an-

tagonistic modulations.

- **Fine-tune the Text Encoder:** If computationally feasible, LoRA-fine-tune the CLIP Text Encoder to better align prompt semantics with your target influencer's visual style.

6.2.2 Data and Conditioning Signal Quality

- **Larger + More Balanced Dataset:** Many of the failures could stem from underfitting due to limited data size or skewed engagement distributions.
- **Use Richer Engagement Signals:** Go beyond likes/comments—include comment sentiment, image-based cues, or even manual annotations to define "engagement."
- **Multimodal Conditioning:** Feed ECM not just abstract numbers but auxiliary image features, text context, or external knowledge embeddings.

This project makes it clear that conditioning generative models on abstract social signals is non-trivial, but also promising. While EC-LoRA-T didn't fully deliver on its original ambition, it built the tools, infrastructure, and diagnostic insights needed to move one step closer to socially intelligent generative systems.

Declarations

Use of Generative AI

Generative AI tools were used throughout this project to support learning and development. Specifically:

- **ChatGPT** (OpenAI) was used for debugging support, architectural clarification, and reviewing best practices in PyTorch, LoRA, and diffusion model implementation.
- **Gemini** (Google) and **Liner** were used for retrieving relevant research papers, summarising technical documentation, and exploring model tuning techniques.

All use of AI tools was for learning, reference, or productivity support purposes.

Ethical Considerations

This project involved collecting and processing Instagram content from the influencer `sungccess`. Although the data was publicly available, explicit consent was obtained from the creator for academic use. No personally identifiable information beyond what was already public was extracted or processed. All images used were hosted and accessed locally and were not redistributed. The project complies with ethical guidelines regarding the responsible use of social media data and informed consent.

Sustainability

Training was conducted efficiently using Google Colab Pro with GPU sessions of limited runtime. Care was taken to minimize idle computation and redundant retraining. Batch sizes and model checkpointing were optimized to reduce resource usage. No external compute clusters were used. Lightweight configurations (e.g., LoRA and float32 precision) were intentionally chosen to limit training cost and energy consumption.

Availability of Data and Materials

The full source code used in the project is available at: https://github.com/roypark337/EC-LoRA-T.

The dataset, including raw images, metadata, and training logs, is hosted privately on Google Drive and can be made available upon request for academic verification purposes. Selected representative samples and statistics are included within this report. All training scripts and configuration files are available in the repository above.

Bibliography

1. Hu EJ, Shen Y, Wallis P, Allen-Zhu Z, Li Y, Wang S, et al. LoRA: Low-Rank Adaptation of Large Language Models. 2021 Jun 17.

2. Zhang L, Tian X, Zhang C, Zhang H. Aided design of bridge aesthetics based on Stable Diffusion fine-tuning. 2024 Sep 24.

3. Shrestha S, Sripada VS, Venkataramanan A. Style Transfer to Calvin and Hobbes comics using Stable Diffusion. 2023 Dec 7.

4. Ramalakshmi S, Asha G. Exploring generative AI: models, applications, and challenges in data synthesis. Asian J Res Comput Sci. 2024;17(12):123–36.

5. Dhariwal P, Nichol A. Diffusion models beat GANs on image synthesis. arXiv [Preprint]. 2021 [cited 2025 Jun 13]. Available from: https://arxiv.org/abs/2105.05233

6. Zhang C, Motwani S, Yu M, Hou J, Juefei-Xu F, Tsai S, et al. Pixel-Space Post-Training of Latent Diffusion Models. 2024 Sep 26.

7. What is Reinforcement Learning from Human Feedback (RLHF)? [Internet]. 2025 Jan [cited 2025 Jun 13]. Available from: https://www.alation.com/blog/what-is-rlhf-reinforcem

8. Liao Z, Liu X, Qin W, Li Q, Wang Q, Wan P, et al. HumanAesExpert: Advancing a multi-modality foundation model for human image aesthetic assessment. arXiv [Preprint]. 2025 Mar 31 [cited 2025 Jun 13]. Available from: https://arxiv.org/abs/2503.23907

9. What is implicit feedback in recommender systems? [Internet]. Milvus; 2025 [cited 2025 Jun 13]. Available from: https://milvus.io/ai-quick-reference/what-is-implicit-feedbac

10. Kulkarni V, Tagasovska N, Vatter T, Garbinato B. Generative models for simulating mobility trajectories. arXiv [Preprint]. 2018 Nov 30 [cited 2025 Jun 13]. Available from: https://doi.org/10.48550/arXiv.1811.12801

11. Ma Z, et al. Efficient diffusion models: a comprehensive survey from principles to practices. IEEE Trans Pattern Anal Mach Intell. 2025. doi: 10.1109/TPAMI.2025.3569700

12. Brophy E, Wang Z, She Q, Ward T. Generative adversarial networks in time series: a survey and taxonomy. arXiv [Preprint]. 2021 Jul 23 [cited 2025 Jun 13]. Available from: https://doi.org/10.48550/arXiv.2107.11098