Politecnico Di Bari

Department of Electrical and Information Engineering

Master Degree in Computer Science Engineering

Report of Deep Leaning

PikaPikaGen: Generative Synthesis of Pokemon Sprites from Textual Descriptions

Professor:

Prof. Vito Walter Anelli

Student::

Mattia Tritto

Academic Year 2024-2025

Index

List of Tables

List of Figures

Chapter 1

Related work

1.1 Review of existing literature on the subject

Recent advancements in text-to-image generation provide a foundation for developing a generative model to create Pokémon sprites from textual descriptions.

Transformer-based architectures, introduced by Vaswani et al. [1], have revolutionized natural language processing (NLP) and are widely used for encoding textual prompts in text-to-image models like DALL·E [2].

Convolutional Neural Networks (CNNs) are effective for image synthesis due to their ability to capture spatial patterns [3]. **Deep Convolutional GANs (DCGANs)** [4] and the **Game Effect Sprite GAN (GESGAN)** [5] demonstrate the efficacy of CNN-based decoders for generating pixel art. GESGAN, in particular, achieves near real-time sprite generation with high visual quality.

Attention mechanisms are critical for aligning textual descriptions with generated image features. Cross-attention, as used in DALL·E and Stable Diffusion [6], enables the decoder to focus on relevant words (e.g., "blue scales") when generating specific sprite regions. This is particularly important for ensuring accurate and visually consistent sprite generation.

Chapter 2

Methodology

2.1 Model Architecture

This section presents the architecture of the text-to-image generation model, which takes a natural language description as input and outputs a corresponding 215x215 image.

2.1.1 Overview

Given a tokenized text sequence, the model performs the following high-level steps:

1. The **Text Encoder** transforms the text into a sequence of rich contextual embeddings using a BERT-based architecture followed by a Transformer encoder.

2. The **Attention Block** compresses the encoded sequence into a single global context vector that captures the semantic content of the text, by applying multi-head attention.

3. The **Image Decoder** generates an image from the global context vector and the full sequence of encoder outputs. It progressively upsamples latent representations from low to high resolution, injecting textual context via cross-attention at each stage.

2.1.2 Text Encoder

The text encoder is responsible for converting the input sequence of tokens into a high-dimensional representation suitable for downstream image synthesis. It consists of two primary components:

- **Pre-trained BERT-mini**: I utilize a compact version of the BERT model as the first stage of the encoder. This model outputs token-level contextual embeddings that are sensitive to the surrounding text.
- **Transformer Encoder Stack**: To further refine the embeddings and adapt them to the generation task, we apply a stack of Transformer encoder layers on top of the BERT output.

During training, optional Gaussian noise can be added to the BERT embeddings before they are passed to the Transformer encoder. This acts as a form of regularization, encouraging robustness and diversity in the learned representations.

Formally, given input token IDs $\mathbf{x} \in \mathbb{N}^{B \times T}$ and attention mask $\mathbf{m} \in \{0, 1\}^{B \times T}$, the encoder produces a sequence of embeddings:

$$\mathbf{H} = \texttt{TransformerEncoder}(\texttt{BERT}(\mathbf{x}, \mathbf{m})) \in \mathbb{R}^{B \times T \times D}$$

where B is the batch size, T is the sequence length, and D is the embedding dimension.

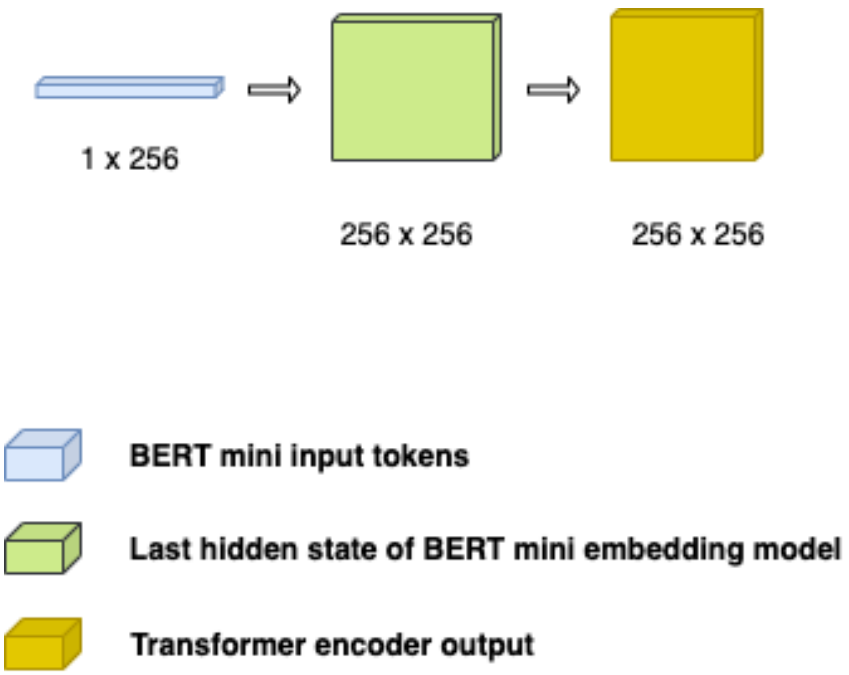


Figure 2.1: Encoder architecture

2.1.3 Attention Block

To summarize the sequence of encoder outputs into a single vector representation that captures the global semantics of the input text, I introduce an `AttentionBlock`. This module computes a weighted aggregation over the encoder sequence.

The process can be broken down into the following steps:

1. **Mean Pooling**: The encoder outputs are first mean-pooled along the sequence dimension, using the attention mask to ignore padding tokens. This produces a single vector per example that captures the overall content.

2. **Query Projection**: The pooled vector is projected to form a query vector of dimension `decoder_dim`.

3. **Multi-head Attention**: The query attends over the full encoder output sequence (used as keys and values) to compute a context vector.

4. **Output Projection**: The final context vector is optionally projected to ensure dimensional compatibility with downstream modules.

This results in a single vector $\mathbf{c} \in \mathbb{R}^{B \times d}$ that acts as a global semantic representation of the input text.

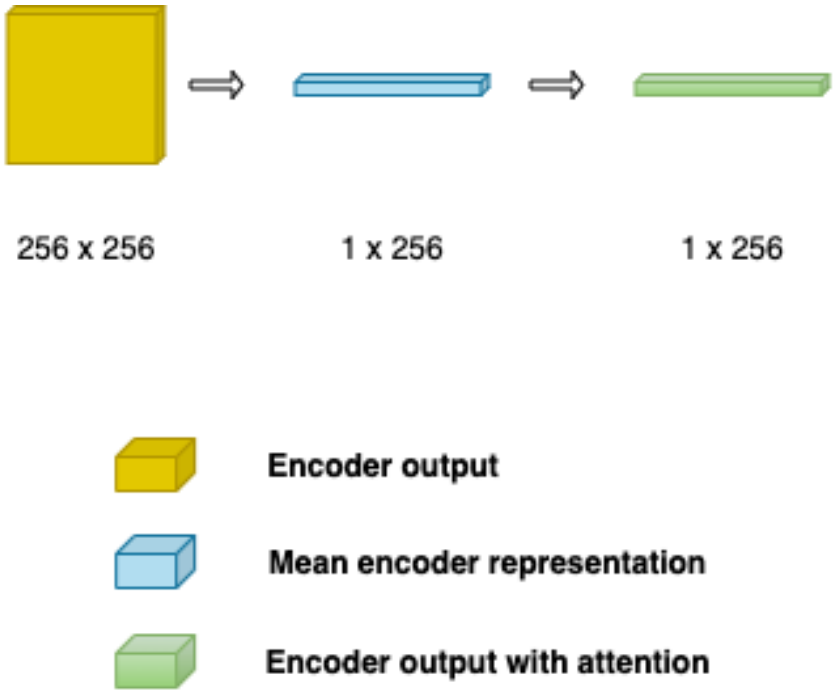


Figure 2.2: Attention architecture

2.1.4 Image Decoder

The image decoder, implemented in the `ImageGenerator` class, is a deep convolutional neural network that synthesizes an image from the text-derived context. Its architecture is inspired by generative adversarial networks (GANs) but extended with cross-attention at each decoding stage.

Input Preparation

The decoder starts by concatenating the global text context vector $\mathbf{c} \in \mathbb{R}^d$ with a randomly sampled Gaussian noise vector $\mathbf{z} \in \mathbb{R}^n$ to obtain a joint input:

$$\mathbf{h}_0 = \texttt{concat}(\mathbf{c}, \mathbf{z}) \in \mathbb{R}^{d+n}$$

This vector is passed through a linear projection to reshape it into a low-resolution feature map of size $8 \times 8 \times 1024$, which forms the starting point of the image generation pipeline.

Progressive Upsampling with Attention

The decoder consists of multiple transposed convolutional layers, each doubling the spatial resolution of the feature maps. At each stage, we inject textual context via an `AttentionBlock` that receives the full sequence of encoder outputs.

Each decoding block performs the following operations:

- Apply transposed convolution, batch normalization, and ReLU to upsample the feature map.
- Flatten the spatial features and compute a new context vector via cross-attention over the text encoder outputs.
- Project the context vector to match the feature map's channel dimension and add it to the feature map.

- Apply dropout for regularization.

This cross-modal conditioning allows the image generation to remain tightly aligned with the textual content throughout the decoding process.

Output Image

The final decoder layer performs a transposed convolution that maps the features to a 3-channel image (RGB). A `Tanh` activation is applied to scale the output values to the range $[-1, 1]$. The output image has dimensions $215 \times 215 \times 3$.

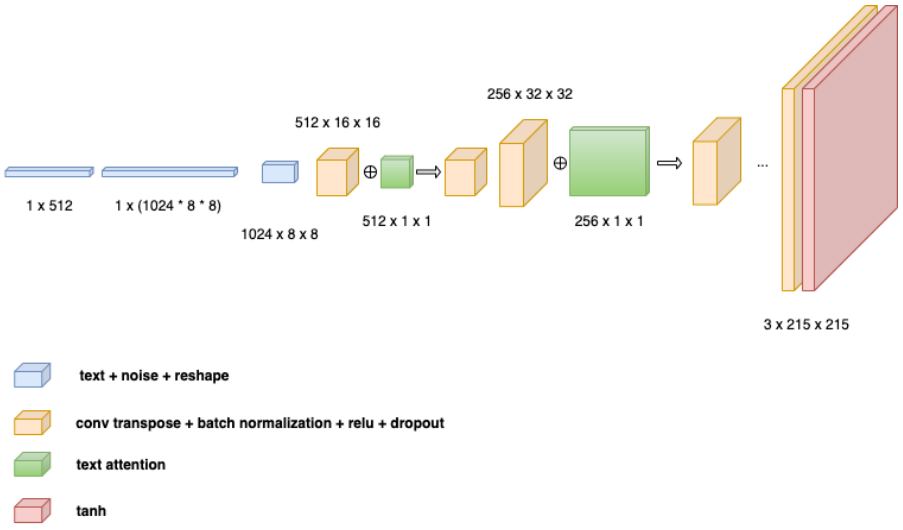


Figure 2.3: Decoder architecture

2.2 Dataset Preprocessing

To train the text-to-image model effectively, I designed a preprocessing pipeline that structures, augments, and splits the original Pokémon dataset into well-defined subsets. This section details the operations performed to prepare the dataset, including image augmentation, dataset splitting, and consistent CSV file generation for metadata tracking.

2.2.1 Input Data Description

The raw dataset consists of a CSV file containing Pokémon metadata and a corresponding sprite image directory:

- **Metadata File:** `pokemon_data.csv` contains one row per Pokémon, including fields such as name, type, and a `national_number` which maps to its sprite filename.
- **Sprite Directory:** Each Pokémon has a PNG image stored in `sprites/`, with filenames formatted as either padded three-digit numbers (e.g., `001.png`).

2.2.2 Train/Validation/Test Split

The dataset is randomly partitioned into three subsets: training, validation, and test, with configurable proportions. By default, 80% of the data is used for training, and 10% each for validation and testing.

Each subset is saved as a CSV file (e.g., `train.csv`, `val.csv`, `test.csv`) in semicolon-separated format.

2.2.3 Image Augmentation

To improve model generalization and increase training data diversity, the pipeline optionally applies deterministic rotation-based augmentation to training and validation images. The augmentation module performs the following steps:

- Each image is rotated n times (default $n = 20$), with angles evenly spaced between $0°$ and $360°$.
- Augmented images are saved with filenames like `025_aug1.png`, `025_aug2.png`, etc.
- Each augmented image is recorded as a new row in the corresponding CSV file, with the `national_number` field updated to reflect the augmented version.

2.2.4 Final Output

The preprocessing function produces the following artifacts:

- `train.csv`, `val.csv`, `test.csv`: Metadata CSVs listing the national numbers (including augmented identifiers) and other attributes.
- `train_sprites/`, `val_sprites/`, `test_sprites/`: Directories containing the sprite images corresponding to each subset.
- Console output summarizing the number of images per subset and any warnings about missing data.

2.2.5 Reproducibility and Parameterization

The preprocessing script is fully parameterized:

- `RANDOM_SEED` ensures deterministic shuffling and splitting.
- `AUGMENT` flag toggles image augmentation.
- `NUM_AUGMENTS` controls how many rotated versions are generated per original.

This design facilitates rapid experimentation while maintaining a consistent preprocessing workflow.

2.3 Training Pipeline

2.3.1 Overview

The training procedure for the Text-to-Image model is centered around learning a mapping from textual descriptions to visual representations of Pokémon sprites. The model is trained on a dataset of text-image pairs using supervised learning, where the objective is to minimize the difference between the generated and real images. The training loop is implemented to support monitoring, checkpointing, and extensibility for hyperparameter search.

2.3.2 Loss Functions

The primary loss used during training is the L1 loss, also known as mean absolute error. This loss is defined as:

$$\mathcal{L}_{\text{L1}} = \frac{1}{N} \sum_{i=1}^{N} |\hat{y}_i - y_i| \,,$$

where $\hat{y}_i$ is the predicted pixel value and y_i is the corresponding ground truth.

2.3.3 Training Process

The training loop iterates over a fixed number of epochs. At each epoch, the model is set to training mode, and the training data is processed in mini-batches. For each batch, the model receives tokenized textual input and generates an image prediction. The L1 loss is computed between the predicted and actual images. The model parameters are updated via backpropagation using the Adam optimizer.

After each epoch, the model is evaluated on the validation set, using the same loss metrics. If the validation loss at the current epoch is lower than any previously observed value, the model checkpoint is saved. This ensures that the best-performing model on the validation set is retained. Additionally, every ten epochs, a subset of predicted and real images is visualized and saved as a PDF file, allowing qualitative assessment of model performance over time.

2.3.4 Hyperparameter Search

To evaluate the impact of optimization parameters, a grid search is performed over different values of learning rate and weight decay. Specifically, the training loop is repeated for each combination of these hyperparameters. For each configuration, a new instance of the model is initialized and trained from scratch. Upon completion, the best validation loss achieved during training is recorded.

2.3.5 Visualization and Monitoring

To monitor training progress, the framework generates and saves learning curves showing training and validation loss over epochs. Moreover, visual comparisons between predicted and ground truth images are saved at regular intervals. These comparisons are compiled into PDFs for both training and validation sets, facilitating qualitative evaluation.

2.3.6 Implementation Details

The model is trained for 100 epochs with a batch size of 10. Images are processed at a resolution of 215×215 pixels. The training is conducted using a GPU if available. The optimizer used is Adam, and the model architecture supports extensions such as noise injection and dropout. A custom dataset class is used to load and preprocess the Pokémon sprites and their corresponding textual descriptions.

2.4 Evaluation Metrics

To assess the quality and relevance of the generated images, a widely adopted metric was employed: the **CLIP Score** This metric capture complementary aspects of image generation semantic alignment with textual input and image realism, respectively.

2.4.1 CLIP Score

The CLIP (Contrastive Language–Image Pre-training) Score leverages the CLIP model developed by OpenAI, which jointly embeds images and their textual descriptions into a shared latent space. Given a generated image and its corresponding textual prompt, the CLIP Score is computed as the cosine similarity between the CLIP-encoded image and text embeddings:

$$\text{CLIP Score} = \frac{\langle \phi_{\text{img}}, \phi_{\text{text}} \rangle}{\|\phi_{\text{img}}\| \cdot \|\phi_{\text{text}}\|}, \tag{2.1}$$

where ϕ_{img} and ϕ_{text} denote the normalized embeddings of the image and the text, respectively.

This score serves as a proxy for semantic alignment, measuring how well the generated image matches its corresponding textual description. Higher CLIP scores indicate better alignment.

2.5 Experimental Setup

All experiments were conducted using Python 3.11, with the required libraries and dependencies explicitly defined in a `requirements.txt` file to ensure reproducibility. The experiments were performed on a local machine with the following hardware specifications:

- **Processor:** Apple M2 chip (8-core CPU, up to 10-core GPU, 16-core Neural Engine)
- **Integrated GPU:** Apple M2 GPU
- **RAM:** 16 GB unified memory
- **Operating System:** macOS (Macintosh HD)

The Apple M2 chip's integrated GPU and Neural Engine provided enhanced computational capabilities compared to traditional CPU-only setups, enabling efficient training and inference. All scripts and utilities were run in a controlled environment, and seed values were fixed to ensure deterministic behavior wherever possible.The use of virtual environments and dependency locking further contributed to the reproducibility and isolation of the software environment across experimental runs.

Chapter 3

Results and Discussion

3.1 First Trial

The hyperparameters used in the first trial were:

- **ID**: 1
- **Augmentation used**: False
- **Epochs**: 100
- **Batch Size**: 10
- **Loss**: L1
- **Learning Rate**: 1.0×10^{-4}
- **Weight Decay**: 1.0×10^{-6}
- **Attention Heads encoder**: 1
- **Dropout Encoder**: Not used
- **Dropout Attention**: Not used
- **Dropout Decoder**: Not used

Train Loss	Validation Loss	Test Loss	CLIP Score
0.12	0.19	0.21	0.23

Table 3.1: Performance metrics for the first trial.

As observed from the learning curves, the model is able to **learn effectively during training**; however, it quickly **overfits** the data. This is evident as the training loss decreases steadily, while the validation and test losses remain higher, indicating poor generalization to unseen data.

Qualitatively, the model performs well on the training images, successfully learning their specific visual patterns. However, it fails to replicate similar quality on the validation images, suggesting that it has memorized the training set rather than learning generalizable features.

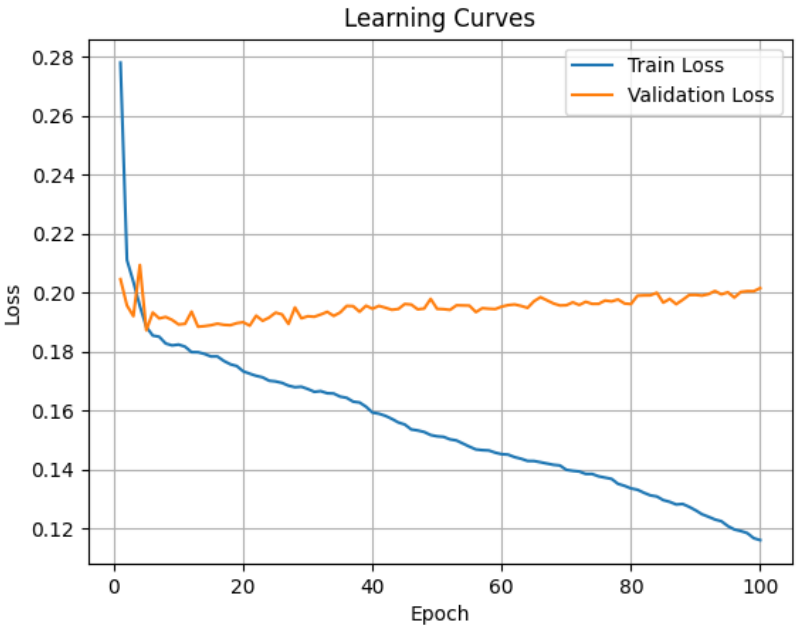
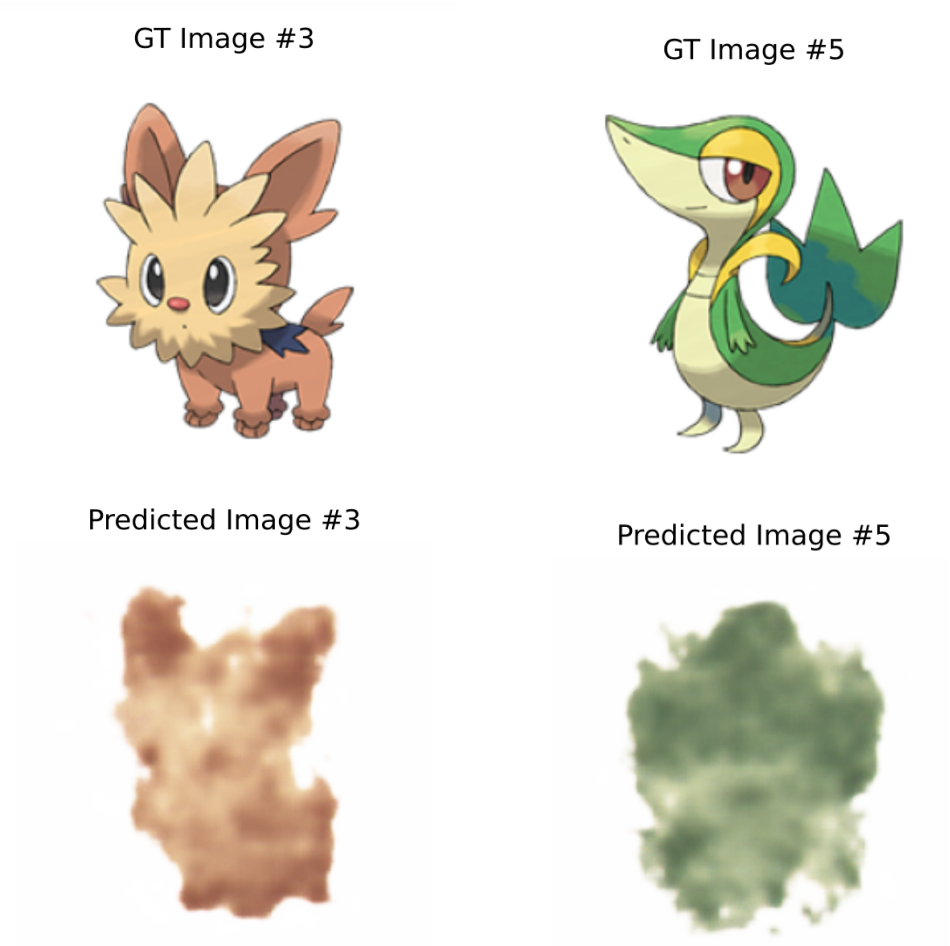


Figure 3.1: Train and validation loss, experiment ID 1



(a) Example on training set (ID 1)

(b) Example on validation set (ID 1)

Figure 3.2: Generated Pokémons from experiment ID 1 on training and validation sets.

3.2 Second trial with dropout added

In this second trial, I added dropout with a probability of 0.3 to all modules of the architecture to see if it would help reduce overfitting.

- **ID**: 1
- **Augmentation used**: False
- **Epochs**: 100
- **Batch Size**: 10
- **Loss**: L1
- **Learning Rate**: 1.0×10^{-4}
- **Weight Decay**: 1.0×10^{-6}
- **Attention Heads encoder**: 1
- **Dropout Encoder**: 0.3
- **Dropout Attention**: 0.3
- **Dropout Decoder**: 0.3

Train Loss	Validation Loss	Test Loss	CLIP Score
0.12	0.19	0.21	0.23
0.12	0.19	0.21	0.23

Table 3.2: First line is the first trial experiment without dropout, the second line is the second trial experiment, with dropout enabled.

According to the results, **dropout did not improve the model's ability to generalize to unseen images** on the validation and test sets. One possible explanation is that the model may already have sufficient regularization or capacity, and introducing dropout might not have helped prevent overfitting.

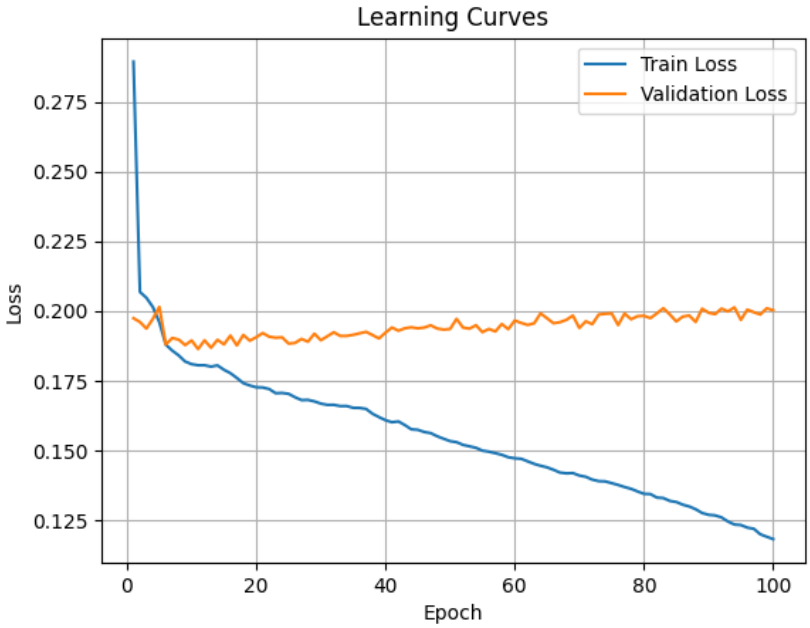


Figure 3.3: Train and validation loss, experiment ID 2

3.3 Third trial with CLIP loss

In this third trial, I **replaced the L1 loss with the CLIP loss** to observe its impact during training. Instead of minimizing the difference between the ground truth image and the predicted image, the model now **minimizes the misalignment between the predicted image and the input prompt**.

- **ID**: 1
- **Augmentation used**: False
- **Epochs**: 100
- **Batch Size**: 10
- **Loss**: CLIP loss
- **Learning Rate**: 1.0×10^{-4}
- **Weight Decay**: 1.0×10^{-6}
- **Attention Heads encoder**: 1
- **Dropout Encoder**: Not used
- **Dropout Attention**: Not used
- **Dropout Decoder**: Not used

Train Loss	Validation Loss	Test Loss	CLIP Score
0.86	0.87	0.85	0.21

Table 3.3: Third trial, using CLIP loss instead of L1 Loss.

According to the results, the CLIP loss leads to **unstable training**, as the model fails to learn, evidenced by the fact that the training loss does not decrease.

One possible reason for this behavior is the initialization strategy: **instead of using BERT-mini vectors, the model should be initialized with embeddings from the CLIP model itself**. This would ensure better compatibility with the CLIP loss and facilitate more effective learning throughout the pipeline.

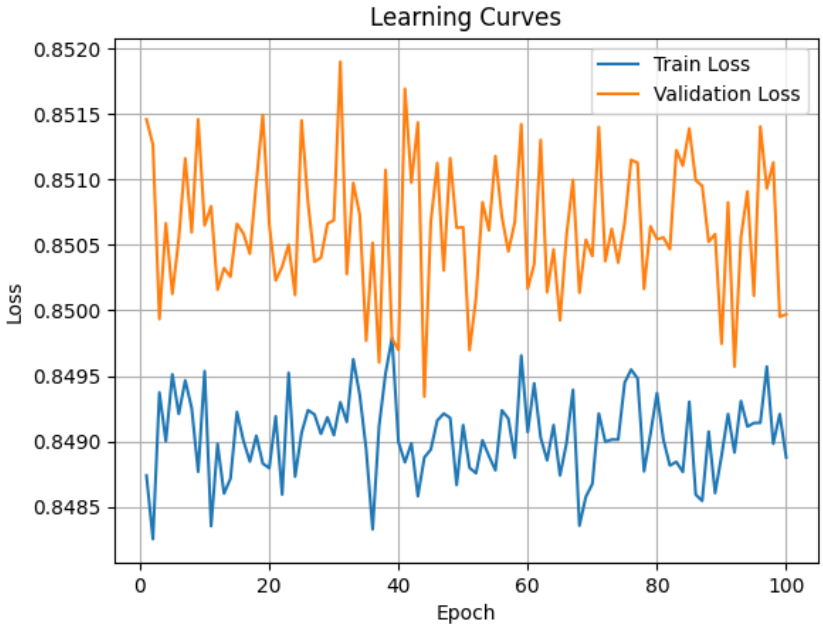


Figure 3.4: Train and validation loss, experiment ID 3

3.4 Fourth trial with data augmentation

In this fourth trial, I augmented the image dataset by **rotating the images**, increasing the dataset size by a factor of 8.

- **ID**: 1
- **Augmentation used**: True
- **Epochs**: 100
- **Batch Size**: 10
- **Loss**: L1 loss
- **Learning Rate**: 1.0×10^{-4}
- **Weight Decay**: 1.0×10^{-6}
- **Attention Heads encoder**: 1
- **Dropout Encoder**: Not used
- **Dropout Attention**: Not used
- **Dropout Decoder**: Not used

Train Loss	Validation Loss	Test Loss	CLIP Score
0.06	0.19	0.23	0.23

Table 3.4: Fourth trial, using dataset augmentation.

According to the results, augmenting the dataset **did not help the model generalize better**. The performance on the validation set, both visually and based on evaluation metrics, remained almost the same as without augmentation. Moreover, the learning curves suggest that the model may have **overfitted the**

training data, as the training loss decreased significantly while the validation performance did not improve.

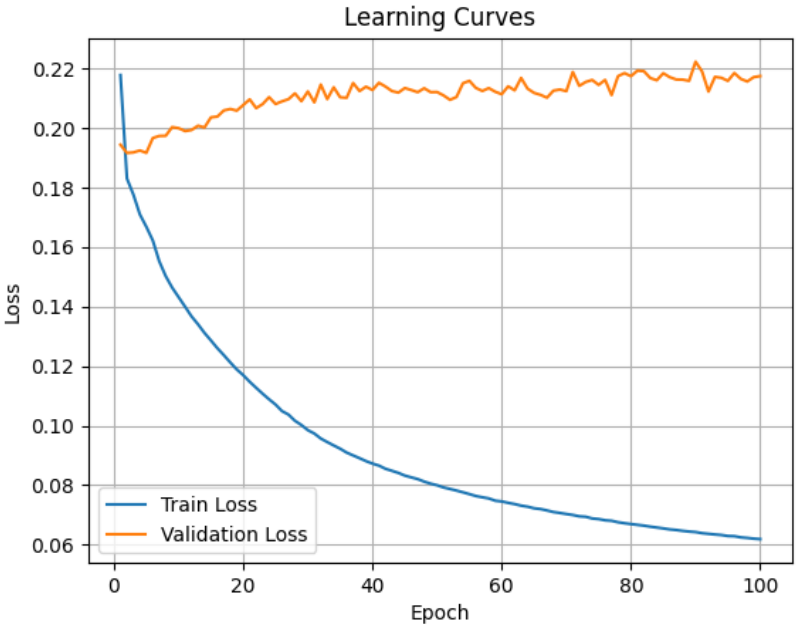


Figure 3.5: Train and validation loss, experiment ID 4

3.5 Fifth Trial with an Enriched Description

In this fifth trial, I used an enriched description that includes additional classification details, such as the primary and secondary types of the Pokémon. Below is an example of the prompt format:

There is a plant seed on its back from the day this Pokémon is born. The seed gradually grows larger as it develops. This Pokémon is classified as a Seed Pokémon. Its primary type is Grass, and its secondary type is Poison.

- **ID**: 5
- **Augmentation used**: False
- **Epochs**: 100
- **Batch Size**: 10
- **Loss**: L1 loss
- **Learning Rate**: 1.0×10^{-4}
- **Weight Decay**: 1.0×10^{-6}
- **Attention Heads encoder**: 1
- **Dropout Encoder**: Not used
- **Dropout Attention**: Not used
- **Dropout Decoder**: Not used

Train Loss	Validation Loss	Test Loss	CLIP Score
0.13	0.19	0.21	0.24

Table 3.5: Fifth trial, using an enriched description.

According to the results, using an enriched description **did not improve the model's ability to generalize**, as the validation loss remained unchanged.

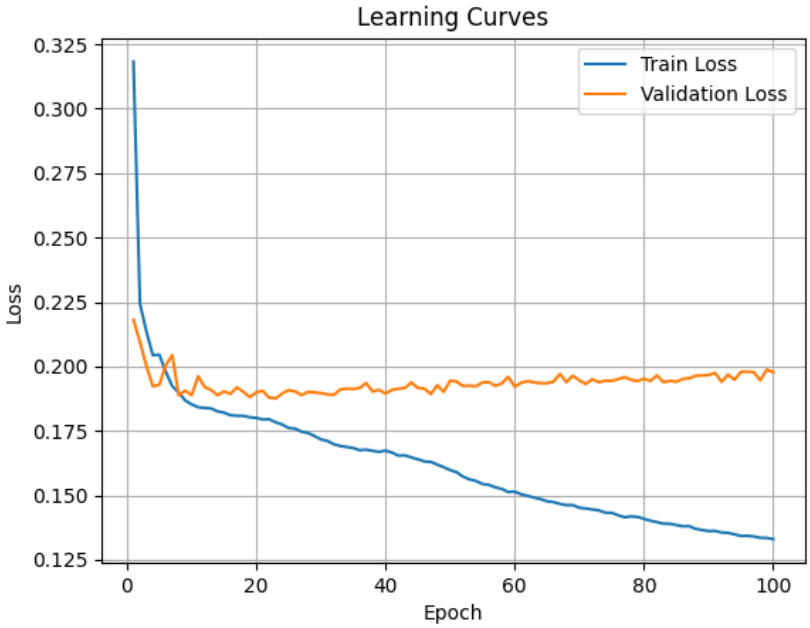


Figure 3.6: Train and validation loss, experiment ID 5

3.6 Hyperparameter Optimization and Final Model Selection

As a final experiment, I conducted a grid-search-style exploration of several hyperparameters to identify the best configuration. The hyperparameters evaluated include:

- **Learning rate**: 1×10^{-4}, 1×10^{-5}
- **Weight decay**: 1×10^{-5}, 1×10^{-6}
- **Number of attention heads in the encoder**: 2, 4
- **Dimension of the feedforward encoder**: 512, 1024
- **Number of transformer encoder layers**: 2, 3

For each configuration, I recorded the best validation loss, test loss, and CLIP score on the test set.

Learning rate	Weight Deacy	N. heads attention encoder	Dim. feedforward encoder	Transfomer encoder layers	Best val. loss L1	Test loss L1	CLIP score
0,00010	0,000001	2	512	2	0,188	0,204	0,229
0,00010	0,000001	2	512	3	0,185	0,205	0,230
0,00010	0,000001	2	1.024	2	0,185	0,204	0,232
0,00010	0,000001	2	1.024	3	0,187	0,205	0,232
0,00010	0,000001	4	512	2	0,186	0,208	0,228
0,00010	0,000001	4	512	3	0,187	0,209	0,232
0,00010	0,000001	4	1.024	2	0,188	0,204	0,232
0,00010	0,000001	4	1.024	3	0,186	0,203	0,226
0,00010	0,000010	2	512	2	0,187	0,208	0,227
0,00010	0,000010	2	512	3	0,187	0,207	0,227
0,00010	0,000010	2	1.024	2	0,187	0,205	0,230
0,00010	0,000010	2	1.024	3	0,185	0,205	0,228
0,00010	0,000010	4	512	2	0,187	0,205	0,233
0,00010	0,000010	4	512	3	0,185	0,204	0,229
0,00010	0,000010	4	1.024	2	0,185	0,205	0,226
0,00010	0,000010	4	1.024	3	0,187	0,205	0,227
0,00001	0,000001	2	512	2	0,192	0,211	0,224
0,00001	0,000001	2	512	3	0,193	0,210	0,228
0,00001	0,000001	2	1.024	2	0,192	0,212	0,223
0,00001	0,000001	2	1.024	3	0,191	0,215	0,226
0,00001	0,000001	4	512	2	0,191	0,211	0,225
0,00001	0,000001	4	512	3	0,193	0,212	0,227
0,00001	0,000001	4	1.024	2	0,198	0,216	0,228
0,00001	0,000001	4	1.024	3	0,198	0,215	0,229
0,00001	0,000010	2	512	2	0,190	0,212	0,223
0,00001	0,000010	2	512	3	0,196	0,219	0,228
0,00001	0,000010	2	1.024	2	0,195	0,208	0,229
0,00001	0,000010	2	1.024	3	0,191	0,217	0,227
0,00001	0,000010	4	512	2	0,193	0,216	0,228
0,00001	0,000010	4	512	3	0,193	0,213	0,230
0,00001	0,000010	4	1.024	2	0,191	0,215	0,227
0,00001	0,000010	4	1.024	3	0,191	0,209	0,226

Figure 3.7: Grid search results

As shown in Figure 3.7, the performance across different configurations remains relatively consistent. Specifically, the validation loss ranges between 0.18–0.19, the test loss between 0.20–0.21, and the CLIP score between 0.20–0.22. This suggests

that the overall performance is not highly sensitive to hyperparameter variations within the explored ranges.

Nevertheless, if a single configuration must be selected, I recommend the one that achieved the lowest validation and test loss:

- **Learning rate**: 1×10^{-4}
- **Weight decay**: 1×10^{-5}
- **Number of attention heads (encoder)**: 4
- **Feedforward encoder dimension**: 1024
- **Number of transformer encoder layers**: 2

Details of the final selected configuration are as follows:

- **ID**: 5
- **Augmentation used**: True (8)
- **Epochs**: 100
- **Batch size**: 10
- **Loss function**: L1 Loss
- **Learning rate**: 1×10^{-4}
- **Weight decay**: 1×10^{-5}
- **Attention heads (encoder)**: 4
- **Dropout (encoder)**: 0.3
- **Dropout (attention)**: 0.3
- **Dropout (decoder)**: 0.3
- **Enriched description used**: Yes

Conclusions

In conclusion, the current architecture **does not appear to be particularly effective for generating Pokémon sprites**. Across all configurations, the model consistently exhibits a **tendency to overfit**, failing to generalize well on both the validation and test sets: this represents a significant limitation.

(a) Prediction image (b) Ground truth image

Figure 3.8: Training set example, prediction VS ground truth

(a) Prediction image (b) Ground truth image

Figure 3.9: Another training set example, prediction VS ground truth

(a) Prediction image (b) Ground truth image

Figure 3.10: Test set example, prediction VS ground truth

(a) Prediction image (b) Ground truth image

Figure 3.11: Another test set example, prediction VS ground truth

I strongly believe that the core issue lies in the architecture itself. To achieve meaningful improvements, it would be necessary to redesign it entirely. The most promising direction would be to adopt a **Stable Diffusion-based model**, which has been extensively studied and proven effective for generating images from textual prompts.

However, before proceeding with a complete architectural overhaul, there are still a few avenues worth exploring to attempt incremental improvements:

- **Data augmentation with textual prompts:** A powerful augmentation technique involves not only manipulating images but also enhancing textual descriptions [7]. This can be achieved by paraphrasing prompts using a large language model (LLM), potentially improving the model's generalization capabilities.

- **A combined loss function:** Incorporating a loss function that combines L1 loss with CLIP loss might be worth exploring. While I am not fully confident in its effectiveness, it could provide more semantically aligned image generation.

- **Staged training:** Rather than fine-tuning the entire model end-to-end from the start, it may be beneficial to train the model in separate stages. First, the encoder could be trained, followed by the decoder, and finally the full model with the attention mechanism. This progressive training strategy might help stabilize learning and improve overall performance.

Further experimentation along these lines could help validate whether such techniques offer measurable improvements before committing to a complete redesign.

Bibliography

[1] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[2] Aditya Ramesh, Prafulla Dhariwal, Alex Nichol, Casey Chu, and Mark Chen. Hierarchical text-conditional image generation with clip latents. *arXiv preprint arXiv:2204.06125*, 2022.

[3] Saad Albawi, Tareq Abed Mohammed, and Saad Al-Zawi. Understanding of a convolutional neural network. *2017 International Conference on Engineering and Technology (ICET)*, 2017.

[4] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015.

[5] Hao Yu, Xuan Chen, Qianru Zhang, Yuecong Liu, and Dimitris N Metaxas. Gesturegan for hand gesture-to-gesture translation in the wild. *arXiv preprint arXiv:2308.06399*, 2023.

[6] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. *arXiv preprint arXiv:2112.10752*, 2022.

[7] Sarvesh Yadav, Deepak Gupta, Asma Ben Abacha, and Dina Demner-Fushman. Paraphrasing to improve the performance of electronic health records

question answering. *AMIA Summits on Translational Science Proceedings*, pages 627–636, 2020.

[8] C. Yu et al. Game effect sprite generation with minimal data via conditional gan. *ScienceDirect*, 2024.

[9] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual models from natural language supervision. *arXiv preprint arXiv:2104.08718*, 2021.