Technology
Arts Sciences
TH Köln

Cologne University of Applied Sciences
Faculty of Computer Science and Engineering Science

M A S T E R T H E S I S

# Texture Asset generation through Transformer Models

Cologne University of Applied Sciences
Campus Gummersbach
Master Digital Sciences

written by:
Dennis Gossler
11140150

**First examiner:** Prof. Dr. Olaf Mersmann
**Second examiner:** Prof. Dr. Boris Naujoks

**Abstract**

TODO

# Contents

# List of Figures

# List of Tables

# 1 Introduction

Transformers are widely used in the generation of text for large language models (LLMs), such as GPT (Radford et al., 2019), LaMMA (Touvron et al., 2023), among others. This thesis aims to extend the application of traditional Transformer-based architectures to generate texture assets, a significant departure from the conventional image-generating methods that primarily rely on stable diffusion techniques or Generative Adversarial Networks (GANs).

The primary objective of this study is to develop a model capable of continuously generating textures for a given seed image or drawing. This approach is analogous to text generation in LLMs, where the model expands upon an initial sentence. To realize this, two distinct models have been developed and trained, each incorporating unique adaptations to leverage the inherent advantages of Transformer-based models in texture generation.

The first model named the Column Image Transformer (CIT), processes images by segmenting them into vertical columns. This design allows the model to predict subsequent pixels within a column using the context information provided by previous pixels.

The second model, the Spiral Image Transformer (SIT), analyzes pixels in a spiral pattern. This approach enables the model to consider both the immediate neighborhood and the broader context of the image, potentially enhancing the texture generation process.

This thesis showcases the steps involved in collecting and preprocessing the data and developing the models. The models are trained on a high-performance computing system, and their effectiveness is evaluated using various metrics. The end of the thesis concludes with an evaluation of the strengths and weaknesses of each model, offering insights into potential areas for further improvement.

## 1.1 Research Questions

Building on the developed models, this thesis explores several research questions to assess the viability and effectiveness of the two mentioned models. The investigation begins by examining whether the Column Image Transformer (CIT) and Spiral Image Transformer (SIT) are technically feasible for generating texture assets for video games, including identifying specific challenges in adapting Transformer architectures from text to image texture generation.

The study also evaluates the performance of these models in producing coherent and continuous textures, identifying appropriate metrics for assessing their effectiveness and comparing these results to traditional methods such as stable diffusion and GANs. Further, it examines the scalability of the models and discusses their potential deployment on a local machine or cloud-based infrastructure.

Lastly, the possibility of further improvements is discussed. The final question will explore potential modifications to the model architecture that could enhance the models performance and efficiency in generating textures for video games.

## 1.2 Related work

The exploration of machine learning models for image generation has been a significant area of research lately, with notable advancements from Generative Adversarial Networks (GANs) to state-of-the-art diffusion models like DALL-E, Midjourney and many more. This section reviews the seminal works and recent innovations in the field, particularly focusing on image/texture generation and the application of Transformer models in the context of images, laying the foundation for the current study's approach to image generating.

**Generative Adversarial Networks (GANs):** Since their introduction by (Goodfellow et al., 2014), GANs have been a cornerstone in the field of generative models, especially for image generation tasks. Works by (Radford et al., 2016), introducing the DCGAN architecture, demonstrated the potential of GANs in producing high-quality images. The adaptability of GANs has been explored in various contexts, including texture synthesis (Xian et al., 2018), showcasing their capability to generate seamless textures for different materials.

**Diffusion Models:** Diffusion models represent a cutting-edge development in the field of generative models, that demonstrate remarkable capabilities in image generation by iterative denoising a random signal to produce detailed images. The process, initially introduced by (Sohl-Dickstein et al., 2015), involves gradually adding noise to an image across several steps and then learning to reverse this process. Stable diffusion, a term often associated with these models, refers to the technique's ability to maintain stability throughout the noise addition and removal process, ensuring high-quality image synthesis. The paper "Diffusion Models Beat GANs on Image Synthesis" (Dhariwal & Nichol, 2021) further refined this concept with models like DDPM, showcasing exceptional fidelity in generated images. This approach contrasts traditional models by focusing on the controlled removal of noise, leading to the generation of coherent and visually impressive images.

**Transformers in Image classification:** The success of Transformer models in natural language processing, as seen with architectures like GPT (Radford et al., 2019) and LaMMA (Touvron et al., 2023), has inspired their application in image-related tasks. The Vision Transformer (ViT) by (Dosovitskiy et al., 2021) marked a significant leap, applying Transformers directly to sequences of image patches for classification tasks.

**Texture Generation with Transformers:** TransGAN by (Jiang et al., 2021) revolutionizes image generation with a Transformer-based GAN architecture, moving beyond traditional Convolutional neural network (CNN) approaches. It features a memory-efficient generator and multiscale discriminator, both utilizing transformer blocks. The TransGAN model produces high-quality images, showcasing the potential of Transformers

## 1.3 Infrastructure for Model Development

To develop and train the models in this thesis, a powerful computing infrastructure is necessary to manage the extensive datasets and the substantial computational requirements for model training. Unlike conventional development environments where a standard laptop or desktop may suffice, most of the models in this thesis demand a more capable infrastructure. Therefore, the high-performance computing system (NHR) at Zuse Institute Berlin (ZIP) is used for the model training processes. This system contains an array of (NVIDIA Tesla A100 80 GB) GPUs, (INTEL Ice Lake 8360Y) CPUs and a significant quantity of RAM. Such a configuration, especially the substantial GPU memory, enables the training and execution of larger models that would be possible on a home workstation. The development of these models is carried out using Python and PyTorch, with the code being developed in Visual Studio Code and managed through version control with Git. The model development and initial code testing are done on a local machine, reserving the high-performance system exclusively for the final training phases. This approach diverges from standard practices, where often both development and execution occur on the same development platform. Ensuring the code is free of errors prior to giving the task of training the model to the high-performance computing system is crucial, as discovering bugs in the training process can be exceedingly time-consuming. For instance, to endure a training session that extends for 24 hours, only to realize it terminated prematurely due to script errors.

## 1.4 Training Data and Data Analysis

This section describes the methods used for gathering, cleaning, and analyzing data for this thesis.

### 1.4.1 Data Acquisition

On the internet, a wide variety of game textures can be found, but not all of them are suitable for this task. The textures collected should be seamless, devoid of shadows, and free from any objects. Textures of floors, such as carpets, tiles, wood, concrete, and more, are utilized. Two approaches are employed to acquire the data for this thesis.

- Web Data Collection

  Some data for this project was obtained from various online sources. Numerous free texture providers are utilized for data acquisition. Due to the limitation of downloading one texture at a time from most websites, a series of scripts are developed to compile a list of suitable textures and automate the downloading process. These scripts are created using UiPath and Python.

- Video Game Textures

  The second approach involved using textures from video games. The advantage of this approach is that it allows for the acquisition of many textures simultaneously. However, a drawback is that these collections of textures include normal maps, height maps, and many other assets that are not suitable. And needed to be filtered and hand-sorted.

### 1.4.2 Data Cleaning

To ensure that the data is consistent and free from elements that could corrupt the model, various cleaning steps were applied. For example, all images containing 3D objects were removed. To obtain the cleaned textures from the video game sources, a Python script is used to discard images containing certain keywords like '_no.png', '_spec.png', and many others. Additionally, an image should contain a keyword like 'floor', 'wall', 'wood', or 'stone'. In summary, 129 good keywords and 56 bad keywords were utilized. The major challenge with this approach is that you can't filter with 100% accuracy, so all the textures are manually checked. In addition, some textures are only usable as overlays for 3D models and aren't supposed to be flat 2D images on their own. So, these are also filtered out. In total, only 20-30% of textures from a video game are viable for use in this thesis.

In the case of web-gathered textures, the textures are often in different folder structures, and it was necessary to standardize them across all data folders. Additionally, some of them had associated files that were irrelevant to this use case and needed to be discarded.

Finally, the images are cropped to 512x512 or 1024x1024 pixels. In total, 57454 images of size 512x512 were collected and 12600 being of size 1024x1024 pixels from 34 sources. The sources are summarized in Table 1.

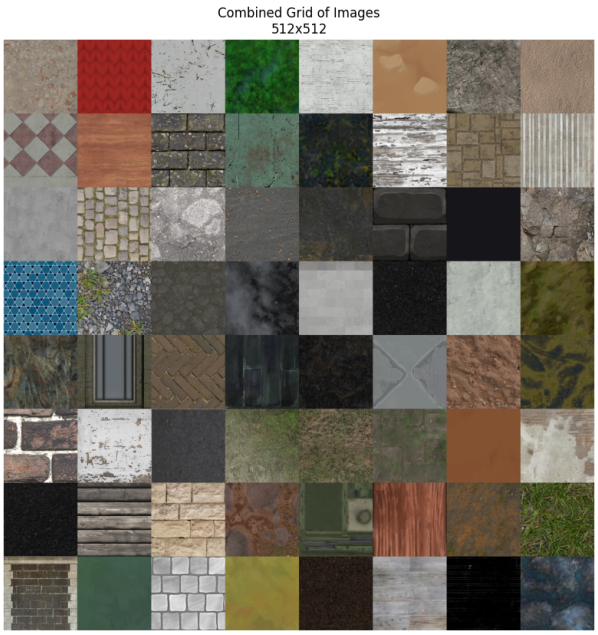



Figure 1: Combined Sources 512x512

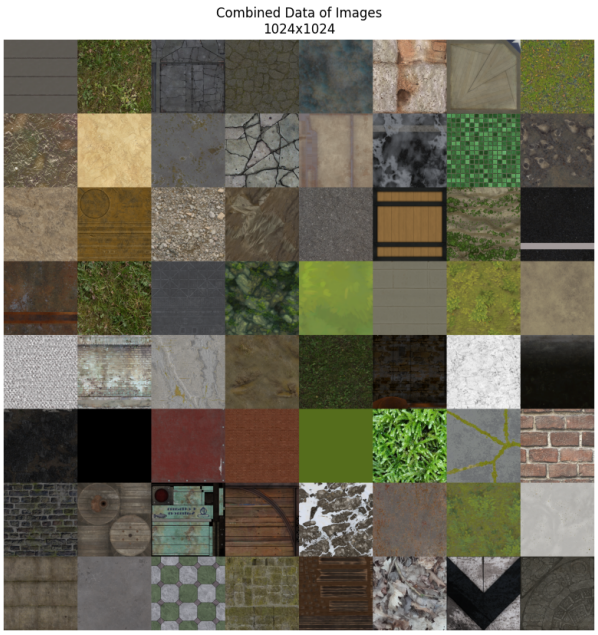



Figure 2: Combined Sources 1024x1024

| Source | Type | Number of Images [size 512x512] | Number of Images [size 1024x1024] |
|---|---|---|---|
| 0 | Game assets | 56 | 14 |
| 1 | Game assets | 3,236 | 809 |
| 2 | Game assets | 2,152 | 539 |
| 3 | Game assets | 184 | 46 |
| 4 | Game assets | 1808 | 452 |
| 5 | Game assets | 100 | 25 |
| 6 | Game assets | 309 | 71 |
| 7 | Game assets | 4154 | 809 |
| 8 | Game assets | 16 | 4 |
| 9 | Game assets | 1,084 | 271 |
| 10 | Website | 1,052 | 263 |
| 11 | Game assets | 102 | 20 |
| 12 | Game assets | 1,334 | 36 |
| 13 | Game assets | 409 | 24 |
| 14 | Game assets | 76 | 19 |
| 15 | Game assets | 5,789 | 628 |
| 16 | Game assets | 632 | 158 |
| 17 | Game assets | 1 | — |
| 18 | Game assets | 1,984 | 491 |
| 19 | Website | 196 | 49 |
| 20 | Website | 1,756 | 439 |
| 21 | Game assets | 643 | 110 |
| 22 | Game assets | 69 | — |
| 23 | Game assets | 1,656 | 414 |
| 24 | Game assets | 4,188 | 1,047 |
| 25 | Game assets | 24 | 6 |
| 26 | Game assets | 2,980 | 745 |
| 27 | Game assets | 996 | 14 |
| 28 | Game assets | 492 | 123 |
| 29 | Game assets | 2,096 | 524 |
| 30 | Website | 17,816 | 4,434 |
| 31 | Game assets | 8 | 2 |
| 32 | Game assets | 4 | 1 |
| 33 | Game assets | 52 | 13 |
| Total | Game assets | 36,634 | 7,415 |
| | Website | 20,820 | 5,185 |
| Total | | 57,454 | 12,600 |

Table 1: The Dataset collected for this thesis

### 1.4.3 Patterns in the data

To examine whether the dataset encompasses a broad spectrum of colors, multiple plots are created. These plots illustrate the color distribution within the datasets, providing insights into the diversity of colors present. Prior to plotting, the colors of the pixels are counted across all images. For instance, if an image features 10 pixels of the color $(255, 0, 0)$, this count is added to a dictionary. Should the subsequent image in the dataset contain 5 pixels of the same color, these are also incorporated into the dictionary, cumulating a total of 15 for that specific color. This process is repeated for each color encountered,

aggregating the counts to yield the overall color frequency within the dataset.

```
color_counts = {}
for i, (data, _) in enumerate(dataset):
    # data is a tensor of shape (3, height, width)
    pixel_rgb_array = (data.view(3, -1).t() * 255).to(torch.int32)

    for pixel_color in map(tuple, pixel_rgb_array):
        if color in color_counts:
            color_counts[pixel_color] += 1
        else:
            color_counts[pixel_color] = 1
```

Figure 3: Python code to count the color frequency in a dataset

After analyzing the dataset through this method, visual representations of the color distributions were produced using Python, Seaborn and Matplotlib.

### 1.4.4 Data Lab plots

The following plot illustrates the color distributions of the whole dataset in the lab color spectrum.

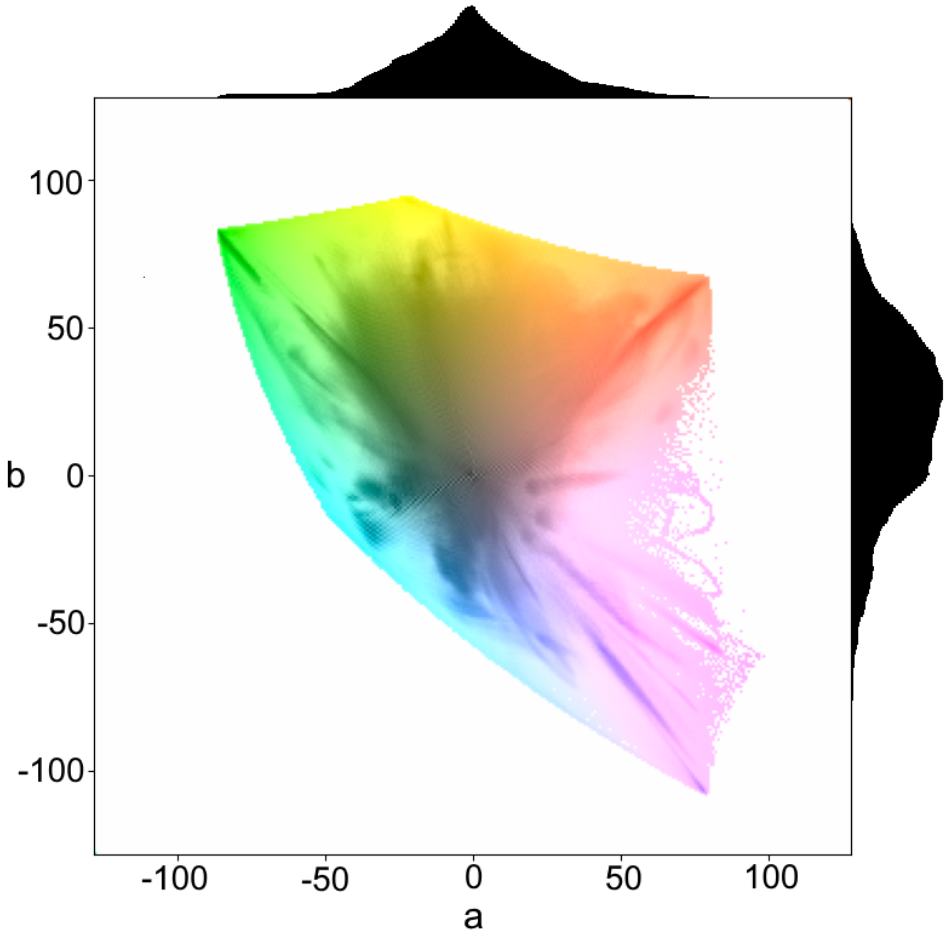



Figure 4: Color distribution of the dataset in the LAB color space

This visualization primarily consists of a central plot complemented by two histograms on its top and right sides. The central plot is a color map representing the distribution of colors from the original data in the CIELAB (Lab) color space, specifically focusing on the 'a' and 'b' components. The 'a' axis (horizontal) indicates the position between green and red colors, while the 'b' axis (vertical) shows the position between blue and yellow colors.

The color map itself is constructed by converting Lab color values back into the RGB color space for display. Colors are distributed across this plot based on their 'a' and 'b' values, with the lightness ('L' component of Lab) squeezed into the 2D plane. The areas with no data are shown in white.

The histograms on the top and right sides of the central plot provide additional context by showing the distribution of 'a' and 'b' values across the dataset. They are weighted by a logarithmic transformation of the count of colors. This logarithmic scaling helps in highlighting patterns in the data without allowing very high or very low counts to

dominate the visualization.

The plot demonstrates that the dataset encompasses a broad array of colors, with a significant quantity of colors spread throughout the entire spectrum. However, the histograms on the sides suggest that the dataset primarily consists of colors in the middle of the spectrum, indicating that it is not overly colorful but leans more toward the white/gray/black spectrum. This could be attributed to the fact that most of the textures are floor textures, which are typically not very colorful. All the various sources are presented as a lab plot in the appendix; see subsection A.1.

### 1.4.5 Data RGB plots

The second plots provide a three-dimensional view of the RGB color space, where the X, Y, and Z axes correspond to the Red, Green, and Blue color values, respectively, each ranging from 0 to 255.

$$\text{size} = \log(\text{count of color}) \times 20$$

The size of each plotted point is determined by the logarithm of the color count, scaled by a factor of 20. See Figure 14 for the combined color distribution of all sources. There, it is also clearly visible that most of the sources are not very colorful, leaning more towards the white/gray/black color spectrum.

## 1.5 Image Transformers Models

This chapter explores the standard Large Language Models (LLMs) and their transformative applications, and later delves into the modifications required to process images and generate images. Furthermore, this chapter introduces the Column Image Transformer (CIT) and the Spiral Image Transformer (SIT) models, which are the primary focus of this thesis.

### 1.5.1 Large Language Models (LLMs)

Large Language Models (LLMs) are a class of machine learning models that have gained significant attention in recent years due to their ability to generate coherent and contextually relevant text. These models are trained on vast amounts of text data, enabling them to understand and generate human-like text. The GPT models by OpenAI (Radford et al., 2019) is a prominent example of an LLM, capable of generating human-like text and performing a wide range of language-related tasks. Like LLMs, the models developed in this thesis predict the next thing in a sequence. But instead of predicting the next word in a sentence, they predict the next pixel in an image. This is achieved by treating the image as a sequence of pixels and using the transformer architecture to predict the next pixel in the sequence.

### 1.5.2 Adapting Transformer Architecture for Image Prediction

The Transformer architecture, introduced by (Vaswani et al., 2023), has significantly impacted the field of natural language processing (NLP). Its widespread adoption across a variety of language tasks, such as machine translation and text generation, highlights its transformative influence. At the heart of the Transformer's success is the self-attention mechanism, which allows the model to weigh different portions of the input data dynamically. This critical feature enables the detection of long-range dependencies and a deeper understanding of the context within the input, making the architecture highly effective for complex NLP tasks.

Building upon this foundation, this thesis explores the extension of the Transformer architecture from its traditional role in NLP to the domain of image prediction. This adaptation employs the architecture's fundamental principles, especially the self-attention mechanism. By treating images as sequences of pixels, the Transformer architecture is applied to predict subsequent pixels in an image sequence, showcasing its potential versatility beyond text-based applications.

### 1.5.3 Column Image Transformer

In the context of this thesis, a model termed the Column Image Transformer (CIT) has been conceptualized and developed. This model embodies an adaptation of the conventional transformer architecture. Distinctively, the CIT model diverges from traditional image processing techniques by segmenting the image into vertical slices or columns of pixels. This segmentation allows for a method where each column is processed on its own.

The adaptation of self-attention for image prediction involves sequentially processing the image, similar to text in natural language processing. However, instead of words or characters, the sequence consists of pixels. In the Column Image Transformer (CIT) model, the image is divided into columns, and the self-attention mechanism is applied to understand the relationships between pixels within each column. This should enable the model to predict the properties of subsequent pixels in a column by considering the context provided by preceding pixels.

### 1.5.4 Spiral Image Transformer

The second approach is represented by the Spiral Image Transformer (SIT). Unlike its predecessor, the Column Image Transformer (CIT), the SIT model employs a contextually spiral pattern. This architecture enables the generation of images starting from a central point and expanding outward see subsubsection 2.3.2. In the SIT model, the batch dimensions correspond to distinct images, whereas the H dimension represents the spiral context. Similar to the CIT model, the C dimension denotes the color channels.

One of the pivotal enhancements of the SIT model is its ability to analyze adjacent pixels on the horizontal axis, in contrast to the CIT model's limitation to columnar pixel analysis. This feature is particularly beneficial for interpreting textures with intricate patterns, such as diagonal ones, thereby offering an advantage over the Column Image Transformer. However, it is important to note that the SIT model operates within a constrained area of the image due to its 2D context. This limitation necessitates the use of only a portion of the image area, specifically a sector determined by the square root of the total area available to the Column Image Transformer (CIT), with an equal context length.

In the Spiral Image Transformer (SIT) model, the self-attention mechanism is adapted to analyze pixels in a spiral pattern. This approach allows the model to evaluate the context in a manner that incorporates both the immediate neighborhood and the broader context of the image, facilitating the prediction of pixel properties in a way that captures complex, two-directional patterns and textures. The self-attention mechanism's ability to

dynamically focus on different parts of the spiral sequence should enable these models to generate coherent predictions for the next pixel, based on the learned importance of each pixel to the others.

# 2 Experiment

This section focuses on the implementation and the tests conducted to evaluate if the models work properly. This includes the data handling, logging, the training process, and the evaluation steps taken of the models.

## 2.1 The Foundation of the Models

To build the models, a set of Python libraries is utilized, mainly PyTorch, NumPy, TensorBoard, and Einops. PyTorch serves as the core tool for constructing and training the models.

The transformer model's code discussed in this thesis is based on a Python script by A. Karpathy named NanoGPT (Karpathy, 2023a). This script implements a straightforward LLM (Large Language Model) approach, and it has been modified to meet the specific needs of the discussed models. However, the underlying transformer architecture remains unchanged.

### 2.1.1 Data Handling

For data management, a script named dataSetCombiner is developed to load and return a combined dataset from specified image folders with optional transformations. The function combines the 33 different sources from multiple directories into a single dataset, also offering the option to apply various image transformations.

It allows for the selection between the 512x512 and the 1024x1024 pixel dataset, depending on the use case. Parameters can be adjusted to tailor the dataset to specific needs, such as image size, the particular dataset to load, and the option to include multiple instances of the dataset. Furthermore, it can randomly flip images vertically or horizontally to augment the dataset, thereby preventing overfitting and improving model generalization. It also supports color jittering, allowing for random adjustments in image brightness, contrast, saturation, and hue, which introduces further variability into the dataset when needed. Additionally, an option to convert images to grayscale is available.

```
1    def getDataSet(path, dataset_name, size_x, size_y, repeatData=1,
     random_vertical_flip=False, random_horizontal_flip=False, crop_type='
     random', grayscale=False, color_jitter=False, jitter_brightness=0,
     jitter_contrast=0, jitter_saturation=0, jitter_hue=0):
```

Figure 5: Python function declaration: getDataSet

### 2.1.2 Monitoring Training Progress

The training process is monitored using TensorBoard, a tool that helps visualize different aspects of training. In this thesis, TensorBoard is particularly useful for tracking training and validation loss through plots. It also allows for the viewing of the input images and the outputs generated by the models.

Secondly, the training progress is monitored by logging the training process. This is helpful because the console output is not accessible until the training has finished on the external Slurm cluster. Logging is performed using the Python logging library. Additionally, all hyperparameters are logged to the log file at the start of the scripts.

## 2.2 Column Image Transformer

The Column Image Transformer (CIT) is the first transformer-based model approach in this thesis. It processes input data in a columnar format as examined in the previous section in more detail see subsubsection 1.5.3.

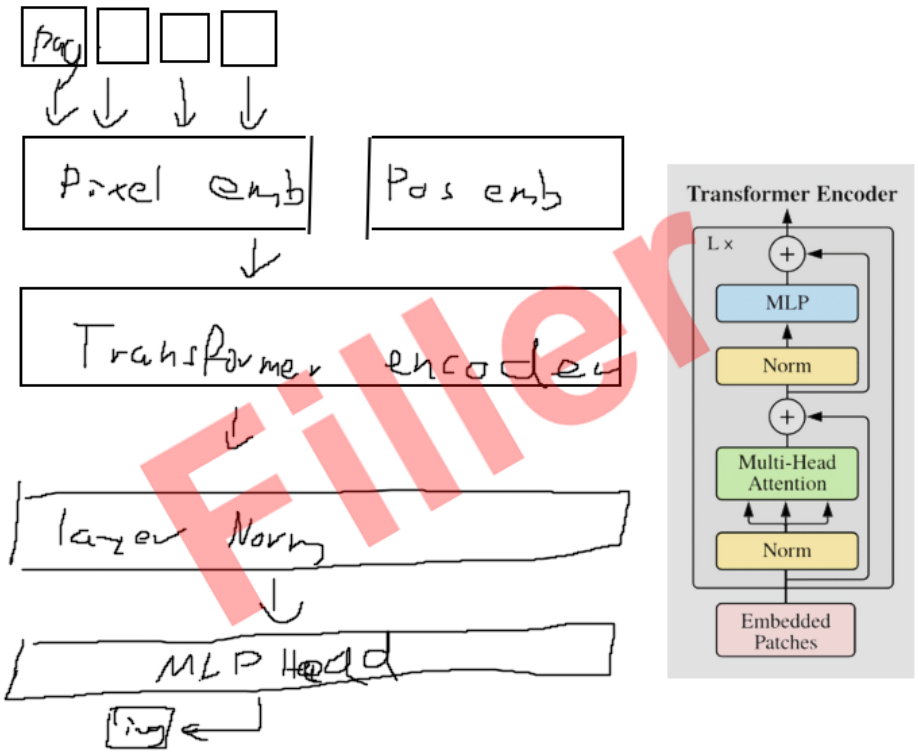



Figure 6: Column Image Transformer Model

The image above shows the steps carried out by the CIT model. The input data is reshaped into a columnar format, which is then embedded into a higher-dimensional space. The positional embedding layer adds positional context to the input data. The transformer layer processes the data, and the output is then transformed back into a columnar format. After going through a final layer normalization and another MLP that converts back from the embedded layer to 3 colors, the model is trained using a Mean Squared Error (MSE) loss function.

### 2.2.1 Get Data as Columns

The data is loaded into a DataLoader object as described in subsubsection 2.1.1 and iterated over to extract it as a 4-dimensional tensor (Batch, Color, Height, Width). This data is reshaped into a columnar format, resulting in two tensors: the *source* and *targets*, both with the shape (Batch, Height, Color). The target tensor is shifted one pixel downwards

to the prediction of subsequent pixels by the model. This reshaping and shifting method is commonly employed in transformer-style learning because it enables the model to process data more efficiently in terms of computational speed and resource utilization. The source tensor is then fed into the model, and its output is compared against the targets using the Mean Squared Error (MSE) loss function.

```
# data: (Batch, Color, Height, Width)
def get_batch(data):

    source = data[:, :BLOCK_SIZE, :BATCH_SIZE]
    targets = data[:, 1:BLOCK_SIZE+1, :BATCH_SIZE]

    source = rearrange(source, 'c h b -> b h c')
    targets = rearrange(targets, 'c h b -> b h c')

    return x, y
```

Figure 7: Python function to convert data into a columnar format

### 2.2.2 Pixel Embedding

The *source* data is embedded into a higher-dimensional space using pixel embedding layers. This MLP is a straightforward linear layer that maps the input data to a higher-dimensional space, transitioning from 3 color channels to $N_{\text{EMBD}}$. This layer consists of a linear transformation followed by a ReLU activation function and is succeeded by a dropout layer.

Through testing and experimentation, it seems that the optimal configuration for the pixel embedding layer involves a sequence of dimensionality increases. The most effective pathway identified begins by scaling the dimensionality of the input from the original image channels $C$ to one-fifth of $N_{\text{EMBD}}$ ($\frac{1}{5}N_{\text{EMBD}}$), then increasing it to one-half of $N_{\text{EMBD}}$ ($\frac{1}{2}N_{\text{EMBD}}$) until reaching $N_{\text{EMBD}}$. The pixel embedding layer is followed by a dropout layer with a dropout rate of 0.2.

Todo: Add Code or Image of the Pixel Embedding Layer

It is crucial to highlight the importance of not using a layer normalization (layernorm) layer in the initial few layers of the model. Incorporating a layernorm layer too early results in outputs that are predominantly black and white. This phenomenon occurs because the color channels $C$, which are comprised of Red, Green, and Blue, are normalized by the layernorm layer to exhibit uniform values across these channels. As a consequence, this normalization process hinders the model's ability to learn the distinct colors of the pixels,

relegating it to only wrong grayscale variations.

### 2.2.3 Positional Embedding

The positional embedding layer in this model utilizes a learnable embedding matrix of size BLOCK_SIZE times $N_{\text{EMBD}}$, where BLOCK_SIZE represents the context length, or the height of an image. This layer operates by assigning each position within this vertical context a unique embedding vector, thereby mapping the original pixel positions to a high-dimensional space characterized by $N_{\text{EMBD}}$ dimensions. The integration of these positional embeddings is achieved through their direct addition to the output of the pixel embeddings, effectively enhancing the model's ability to maintain positional awareness across the image height.

```
class ColumnTransformer(nn.Module):

    self.position_embedding_table = nn.Embedding(BLOCK_SIZE, N_EMBD)
    #[...]

def forward(self, idx, targets=None):

    # tok_emb: (B, H, N_EMBD)
    tok_emb = self.pixel_embedding(source)

    # pos_emb: (H, N_EMBD)
    pos_emb = self.position_embedding_table(torch.arange(H))

    # x: (B,H,N_EMBD)
    x = tok_emb + pos_emb
    #[...]
```

Figure 8: Python code snippet for the Positional Embedding Layer

### 2.2.4 Transformer Layer

The Transformer layer forms the core of the model's architecture and is crucial for processing the data. It utilizes self-attention mechanisms (Vaswani et al., 2023) to weigh the importance of different pixels relative to each other, allowing the model to focus on relevant parts of the input when predicting the next pixel in a column.

In this model, each Transformer layer receives the input from the previous layer. The layer then processes this input using a multi-head attention mechanism, which allows the model to capture various aspects of the input at different positions simultaneously. This is followed by a series of normalization and feed-forward layers. All are combined into a

block that is repeated multiple times. See subsection A.4 for the code snippet.

### 2.2.5 Output Layer

The output layer is like the pixel embedding a simple MLP that converts the data back to the original color space. It has the same structure but reversed like the Pixel Embedding layer but with the output dimensionality set to 3, representing the Red, Green, and Blue color channels. After the shrinkage of the dimensionality, the output is passed through one final sigmoid to map the resulting colors to 0-1.

### 2.2.6 Sigmoid Compared to Clamp

Due to discoloration issues in the model output, the performance has been evaluated using two different activation functions: sigmoid and linear clamp. Typically, the sigmoid function maps input values to a range between 0 and 1. However, in the context of color values, this compression into the [0, 1] range can prevent achieving true black and white colors, as extremely large or small inputs are needed to reach the extremes of 0 or 1. Initially, the linear clamp function, which restricts input values to a specified range without compression, seemed to better preserve the distinct black and white colors. However, it has several notable disadvantages, such as ...(please fill in the biggest problem with backpropagation).

TODO: Add more information about the Problem with the clamp function

In discussions about this problem, it was suggested that the lack of training data featuring significant black and white samples might be a contributing factor. After collecting more data and expanding the training set, the issue was resolved, and the model performance with the sigmoid activation function is very similar, showing improvement comparable to the clamp function.

### 2.2.7 Discoloration in the Model Output

The model faces similar discoloration issues as mentioned in section subsubsection 2.2.6 with certain colors. When the model starts with a random pixel, it often struggles to consistently generate the intended pattern in that color or starting pattern. Some evidence suggests that the training set may be too small, as the issue was less noticeable with a larger dataset. Generally, transformers require significantly more data to improve performance (Chen et al., 2022). However, this problem has not yet been resolved, and further research is needed.

### 2.2.8 Classification or Regression

To see if the method (Classification / Regression) has a big impact on the performance of the model both methods are tested. For comparison grayscale input and outputs are used. This strategy is particularly advantageous because it minimizes the complexity arising from the vast number of potential output possibilities in classification tasks. With RGB color representation, each pixel can be classified into one of 16,581,375 distinct color combinations (255 possibilities for each of the Red, Green, and Blue channels).

Utilizing grayscale input and outputs, simplifies the classification task, reducing the number of potential outcomes to a manageable range. Grayscale images, with their single intensity channel representing luminance, offer a more straightforward basis for analysis. This simplification allows for a more focused evaluation and a faster check if the model performs differently or better.

TODO: Add more information about the Classification and Regression Results

### 2.2.9 Multi GPU Training

To enhance the training speed of larger models, utilizing multiple GPUs can be significantly more efficient. Consequently, the model's code has been adapted to support multi-GPU training while maintaining the original core structure. Necessary adjustments were made to the data DataLoader, logger, and trainer. For distributed training across multiple GPUs, the PyTorch library Distributed-Data-Parallel (DDP) is used. This implementation draws on guidelines from the PyTorch documentation (Subramanian, 2023). The larger models are trained on an external Slurm cluster equipped with four A100 Tesla GPUs.

## 2.3 Spiral Image Transformer

As explained in the previous section (see Section 1.5.4), the spiral model is a transformer-based model that employs a spiral architecture to process input data.

### 2.3.1 Spiral

To convert the 3-dimensional data (color channels, image height, image width) into a spiral form, the data must be unrolled into one dimension, resulting in an output format of (color channels, spiral length). Consequently, the data width and the height dimensions must be squeezed into a single dimension. This new form can then be utilized similarly to the data in the Column Image Transformer. As the data is flattened into a single dimension, it allows for an easy addition of positional embedding. It is the same process as in the Column Image Transformer see subsubsection 2.2.3.

### 2.3.2 Spiral generation

The conversion of data into a spiral form needs to be highly efficient because the code block will execute for every image (batch) in the dataset. Therefore, a simple nested for loop is insufficient. In this example, fancy indexing is used to convert the data into a spiral form. Thus, the data tensor is indexed with a two-dimensional tensor containing the indices of the spiral form.

At the start of the model training script, one indexing spiral is created to be used for all images in the dataset. The following code block illustrates the creation of the spiral index tensor.

```python
def create_spiral(n): # n = width and height

    matrix = [[0] * n for _ in range(n)] # Initialize n x n matrix

    x, y = 0, 0
    # Direction vectors (right, down, left, up)
    dx = [0, 1, 0, -1]
    dy = [1, 0, -1, 0]
    direction = 0

    for i in range(n * n - 1, -1, -1):  # Start (35 for 6x6)
        matrix[x][y] = i
        nx = x + dx[direction]
        ny = y + dy[direction]

        # Change direction if the next position: out of bounds or filled
        if nx<0 or nx>=n or ny<0 or ny>=n or matrix[nx][ny]!=0:
            direction = (direction + 1) % 4  # Change direction
            nx = x + dx[direction]
            ny = y + dy[direction]

        x, y = nx, ny

    return torch.tensor(matrix)
```

Figure 9: Python function to create a spiral index tensor

The code above generates a square matrix of size n by n, then fills it with numbers in a spiral pattern, starting from the outer edge and spiraling inwards clockwise. Each cell of the matrix is assigned a unique number, beginning from the highest value in the top-left corner and decreasing by one with each step along the spiral path until reaching zero at the center or the end of the spiral. The spiral formation is achieved by moving right, then down, then left, then up, and repeating this sequence, adjusting direction whenever the next step would go out of bounds or into a cell that's already been filled.

## 2.4 Fancy Indexing into the Spiral form

The spiral index tensor is then used to index the data tensor, effectively converting it into a spiral form. The following image illustrates the process of converting a 7x7 image.

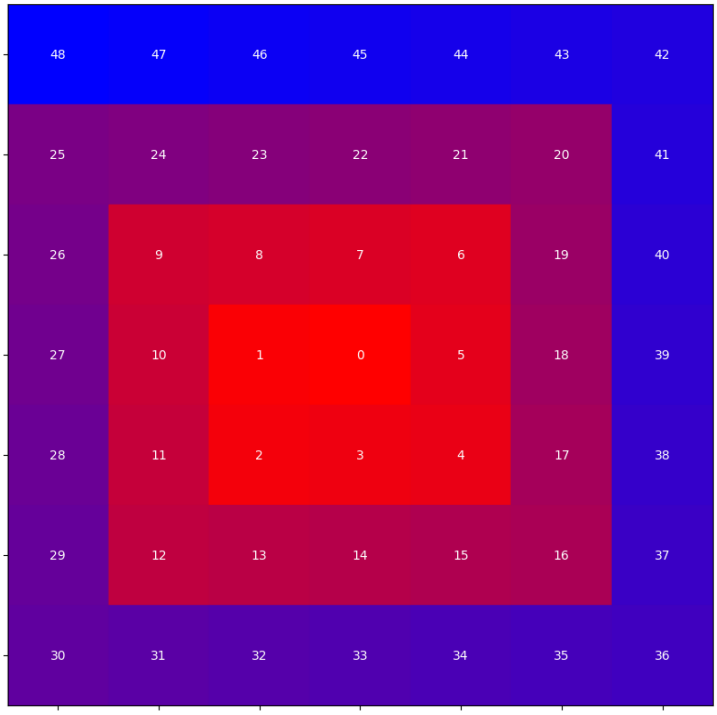



Figure 10: Image representation

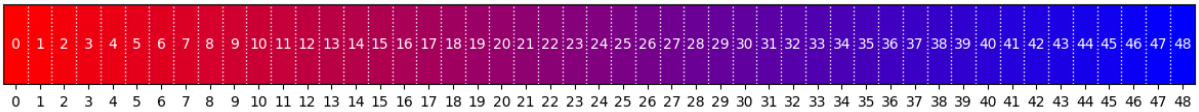



Figure 11: 7x7 Image flattened into a spiral form

As you can see in Figure 10, the pixels of the image are labeled with their respective indices $0, \ldots, 43$. The image is then unrolled into a single dimension, as shown in Figure 11. The centering pixel is the first element of the spiral, and the spiral continues counterclockwise from there. In the model script, the dimensions for width and height typically exceed 7, yet the underlying process remains unchanged.

```
spiral_indices = torch.tensor(create_spiral(IMAGE_SIZE))

# [...]
# Batch_size, Color_channels, Height, Width
B, C, H, W = data.shape

spiral_data = torch.zeros_like(data.view(B, C, -1))

spiral_data[:,:,spiral_indices.flatten()] = data.view(B, C, -1)

# [...]
```

Figure 12: Fancy indexing to convert data into a spiral form

# 3 Conclusion

This section focuses on the evaluation of the performance of the CIT and the SIT.

## 3.1 Evaluation of the models

As a general conclusion it can be said that the data hints that it is possible to generate quickly new assets trow this approach. Due to the fact that the models are relatively small and the training dataset is not that big the output of the models is

### 3.1.1 performance

One the big question is if it is possible to generated assets for video game. Here we can split it into two parts. The first part is to generate assets beforehand in the development cylce and in the second part is is the model so efficient that it is possible to generate assets in real time on a local machine.

### 3.1.2 quality

## 3.2 Execution locally and on the cloud

## 3.3 Further research

### 3.3.1 Discriminator

To get a better result a discriminator can be used to enhance the output of the model. In this case, a discriminator is added after the model prediction.

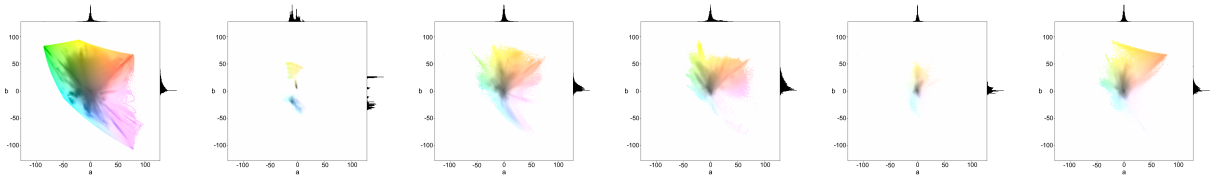
For these XX new rows are generated and then the discriminator checks if the generated content is artifice or an original image. In a perfect scenario the discriminator can't distinguish between the generated content and the original content and the loss will balance out at 50

### 3.3.2 LLM Scaling Laws

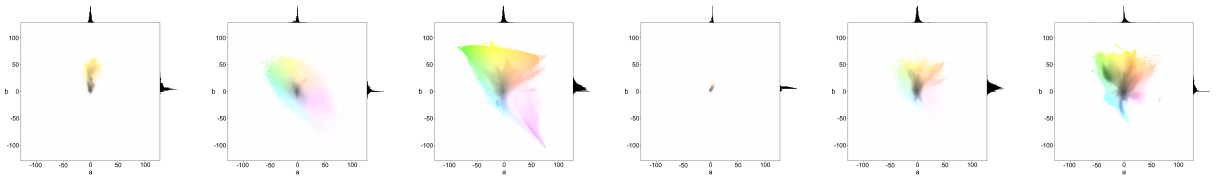
## 3.4 Stable diffusion/ GANs with convolutional neural network

# A Appendix

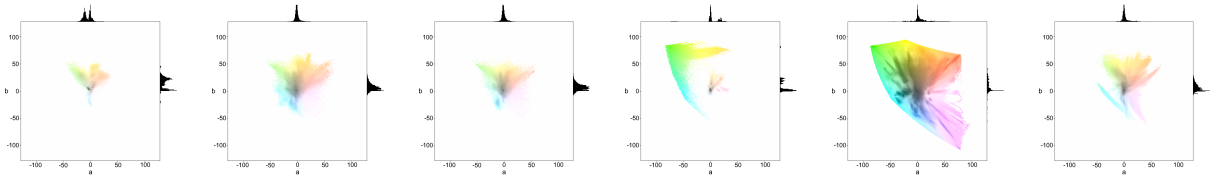
## A.1 Dataset Plots in the Lab Color Space



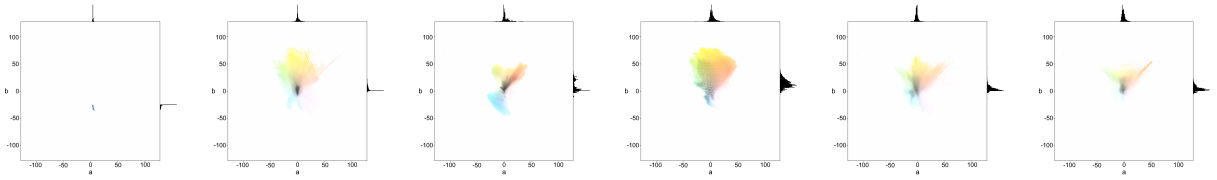
(1) All Data (2) source 0 (3) source 1 (4) source 2 (5) source 3 (6) source 4



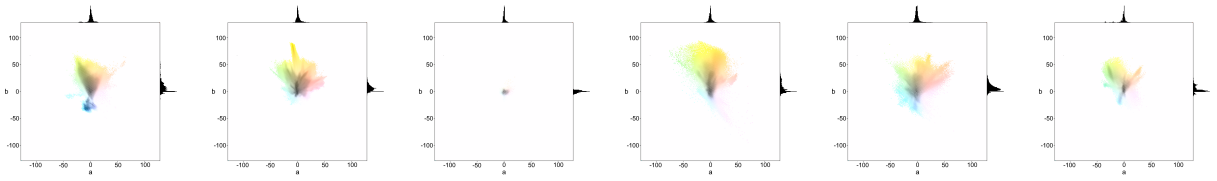
(7) source 5 (8) source 6 (9) source 7 (10) source 8 (11) source 9 (12) source 10



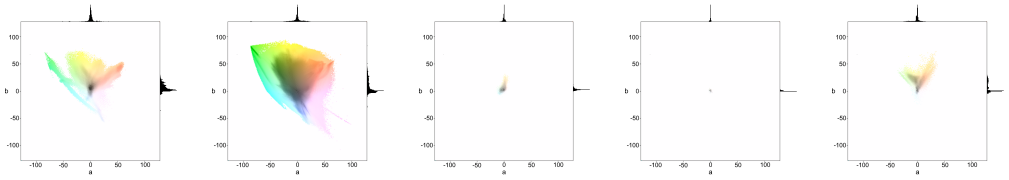
(13) source 11 (14) source 12 (15) source 13 (16) source 14 (17) source 15 (18) source 16



(19) source 17 (20) source 18 (21) source 19 (22) source 20 (23) source 21 (24) source 22



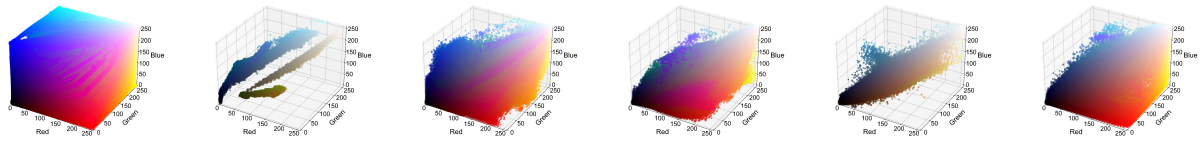
(25) source 23 (26) source 24 (27) source 25 (28) source 26 (29) source 27 (30) source 28



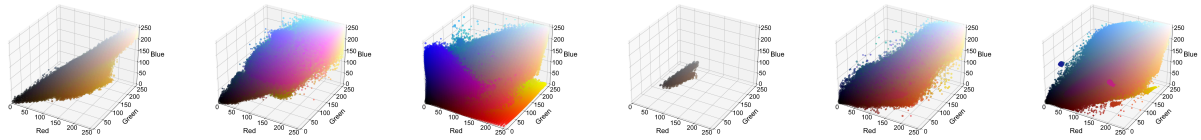
(31) source 29 (32) source 30 (33) source 31 (34) source 32 (35) source 33

Figure 13: Visualization of datasets plotted in the LAB color space

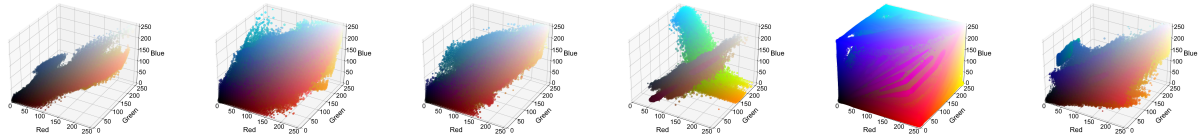
## A.2 Dataset Plots in the RGB Color Space



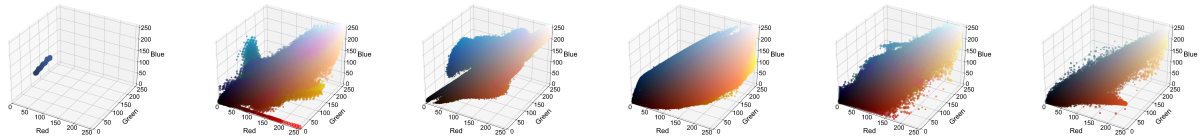
(1) All Data (2) source 0 (3) source 1 (4) source 2 (5) source 3 (6) source 4



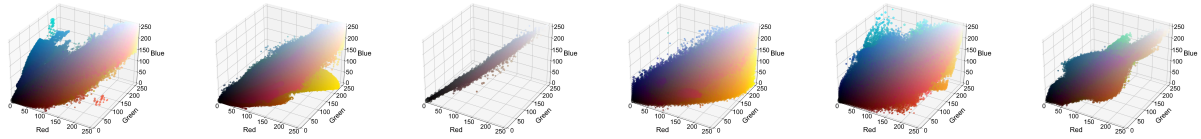
(7) source 5 (8) source 6 (9) source 7 (10) source 8 (11) source 9 (12) source 10



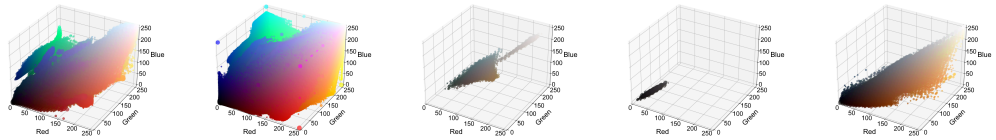
(13) source 11 (14) source 12 (15) source 13 (16) source 14 (17) source 15 (18) source 16



(19) source 17 (20) source 18 (21) source 19 (22) source 20 (23) source 21 (24) source 22



(25) source 23 (26) source 24 (27) source 25 (28) source 26 (29) source 27 (30) source 28



(31) source 29 (32) source 30 (33) source 31 (34) source 32 (35) source 33

Figure 14: Visualization of datasets plotted in the RGB color space

## A.3 Local workstation setup

| Component | Specification |
|---|---|
| Operating System | Microsoft Windows 11 Pro |
| CPU | AMD Ryzen 9 7900X 12 cores, 4.7 GHz |
| GPU | NVIDIA RTX 3090 |
| RAM | 64 GB (DDR5-6000) |
| SSD | 2TB Samsung 990 PRO M.2 PCIe 4.0 |

Table 2: Local Workstation Setup

| Component | Specification |
|---|---|
| CPU | 2x Intel Xeon "Ice Lake" Platinum 8360Y (36 cores per socket, 2.4 GHz) |
| GPU | 4x Nvidia A100 (80GB HBM2, SXM) |
| RAM | 1 TB RAM (DDR4-3200) |
| SSD | 7.68 TB NVMe local SSD |

Table 3: Single compute node

## A.4 Transformer layer implementation

```
class Head(nn.Module):
""" one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(N_EMBD, head_size, bias=False)
        self.query = nn.Linear(N_EMBD, head_size, bias=False)
        self.value = nn.Linear(N_EMBD, head_size, bias=False)
        self.register_buffer('tril', torch.tril(torch.ones(BLOCK_SIZE, BLOCK_SIZE)))

        self.dropout = nn.Dropout(DROPOUT)

    def forward(self, x):
        # input of size (batch, time-step, channels)
        # output of size (batch, time-step, head size)
        B,T,C = x.shape
        k = self.key(x)   # (B,T,hs)
        q = self.query(x) # (B,T,hs)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 # (B, T, hs) @ (B, hs, T) -> (B, T, T)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (B, T, T)
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)
        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,hs)
        out = wei @ v # (B, T, T) @ (B, T, hs) -> (B, T, hs)
        return out

class MultiHeadAttention(nn.Module):
""" multiple heads of self-attention in parallel """

    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(head_size * num_heads, N_EMBD)
        self.dropout = nn.Dropout(DROPOUT)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out

```

```
class FeedFoward(nn.Module):
""" a simple linear layer followed by a non-linearity """

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(DROPOUT),
        )

    def forward(self, x):
        return self.net(x)

class Block(nn.Module):
""" Transformer block: communication followed by computation """

    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd
    like
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x
```

This code snippet shows the implementation of the transformer layer in the Column Image Transformer and the Spiral Image Transformer. The code above is heavily inspired by the implementation of the script by A. Karpathy named NanoGPT (Karpathy, 2023a) (Karpathy, 2023b)

# B Eidesstattliche Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht.
Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

---

(Ort, Datum, Unterschrift)

# References


Chen, X., Cao, Q., Zhong, Y., Zhang, J., Gao, S., & Tao, D. (2022). Dearkd: Data-efficient early knowledge distillation for vision transformers.

Dhariwal, P., & Nichol, A. (2021). Diffusion models beat gans on image synthesis.

Dosovitskiy, A., Beyer, L., Kolesnikov, A., Weissenborn, D., Zhai, X., Unterthiner, T., Dehghani, M., Minderer, M., Heigold, G., Gelly, S., Uszkoreit, J., & Houlsby, N. (2021). An image is worth 16x16 words: Transformers for image recognition at scale.

Goodfellow, I. J., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., & Bengio, Y. (2014). Generative adversarial networks.

Jiang, Y., Chang, S., & Wang, Z. (2021). Transgan: Two pure transformers can make one strong gan, and that can scale up.

Karpathy, A. (2023a). GitHub - karpathy/nanoGPT: The simplest, fastest repository for training/finetuning medium-sized GPTs. https://github.com/karpathy/nanoGPT/tree/master

Karpathy, A. (2023b). GitHub - nanogpt-lecture. https://github.com/karpathy/ng-video-lecture/blob/master/gpt.py

Radford, A., Metz, L., & Chintala, S. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). Language models are unsupervised multitask learners.

Sohl-Dickstein, J., Weiss, E. A., Maheswaranathan, N., & Ganguli, S. (2015). Deep unsupervised learning using nonequilibrium thermodynamics.

Subramanian, S. (2023). Multi GPU training with DDP — PyTorch Tutorials 2.2.2+cu121 documentation. https://pytorch.org/tutorials/beginner/ddp_series_multigpu.html?utm_source=youtube&utm_medium=organic_social&utm_campaign=tutorial

Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., Bikel, D., Blecher, L., Ferrer, C. C., Chen, M., Cucurull, G., Esiobu, D., Fernandes, J., Fu, J., Fu, W., ... Scialom, T. (2023). Llama 2: Open foundation and fine-tuned chat models.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L., & Polosukhin, I. (2023). Attention is all you need.

Xian, W., Sangkloy, P., Agrawal, V., Raj, A., Lu, J., Fang, C., Yu, F., & Hays, J. (2018). Texturegan: Controlling deep image synthesis with texture patches.