



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	1
1.1	Research Questions	1
1.2	Related work	2
1.3	Infrastructure for Model Development	3
1.4	Training Data and Data Analysis	4
1.4.1	Data Acquisition	4
1.4.2	Data Cleaning	4
1.4.3	Patterns in the data	6
1.4.4	Data Lab plots	7
1.4.5	Data RGB plots	9
1.5	Image Transformers Models	10
1.5.1	Large Language Models (LLMs)	10
1.5.2	Adapting Transformer Architecture for Image Prediction	10
1.5.3	Column Image Transformer	11
1.5.4	Spiral Image Transformer	11
2	Experiment	13
2.1	The Foundation of the Models	13
2.1.1	Data Handling	13
2.1.2	Monitoring Training Progress	14
2.1.3	Multi GPU Training	14
2.2	Column Image Transformer	15
2.2.1	Get Data as Columns	15
2.2.2	Pixel Embedding	16
2.2.3	Positional Embedding	17
2.2.4	Transformer Layer	17
2.2.5	Output Layer	18
2.2.6	Sigmoid Compared to Clamp	18
2.2.7	Discoloration in the Model Output	18
2.2.8	Classification or Regression	19
2.2.9	Training the Model	19
2.2.10	Experimental Results	20
2.2.11	Challenges and Limitations	23
2.3	Spiral Image Transformer	24

2.3.1	Spiral Data Processing	24
2.3.2	Spiral generation	25
2.3.3	Model Architecture	27
2.3.4	Training the Model	28
2.3.5	Experimental Results	28
2.3.6	Challenges and Limitations	28
3	Conclusion	29
3.1	Evaluation of the models	29
3.1.1	performance	29
3.1.2	quality	29
3.2	Execution locally and on the cloud	29
3.3	Further research	29
3.3.1	Discriminator	29
3.3.2	LLM Scaling Laws	30
3.4	Stable diffusion/ GANs with convolutional neural network	30
A	Appendix	ii
A.1	Dataset Plots in the Lab Color Space	ii
A.2	Dataset Plots in the RGB Color Space	iv
A.3	Trained Models with hyperparameters	vi
A.4	Local workstation setup	vii
A.5	Transformer layer implementation	viii
B	Eidesstattliche Erklärung	

List of Figures

1	Combined Sources 512x512	5
2	Combined Sources 1024x1024	5
3	Python code to count the color frequency in a dataset	7
4	Color distribution of the dataset in the LAB color space	8
5	Python function declaration: getDataSet	13
6	Column Image Transformer Model	15
7	Python function to convert data into a columnar format	16
8	Python code snippet for the Positional Embedding Layer	17
9	Column Image Transformer Model	20
10	Color test with four different models	21
11	Zebra test vertical with two different models	22
12	Zebra test horizontal with two different models	22
13	Spiral Image Transformer Model	24
14	Get data in a spiral form for the SIT model	25
15	Python function to create a spiral index tensor	26
16	Image representation	27
17	7x7 Image flattened into a spiral form	27
18	Visualization of datasets plotted in the LAB color space	iii
19	Visualization of datasets plotted in the RGB color space	v

List of Tables

1	The Dataset collected for this thesis	6
2	Model Configuration Overview	vi
3	Local Workstation Setup	vii
4	Single compute node	vii

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, thereby providing a simple method for video game prototyping to generate texture assets. 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.

```
1  color_counts = {}
2  for i, (data, _) in enumerate(dataset):
3      # data is a tensor of shape (3, height, width)
4      pixel_rgb_array = (data.view(3, -1).t() * 255).to(torch.int32)
5
6      for pixel_color in map(tuple, pixel_rgb_array):
7          if color in color_counts:
8              color_counts[pixel_color] += 1
9          else:
10              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.

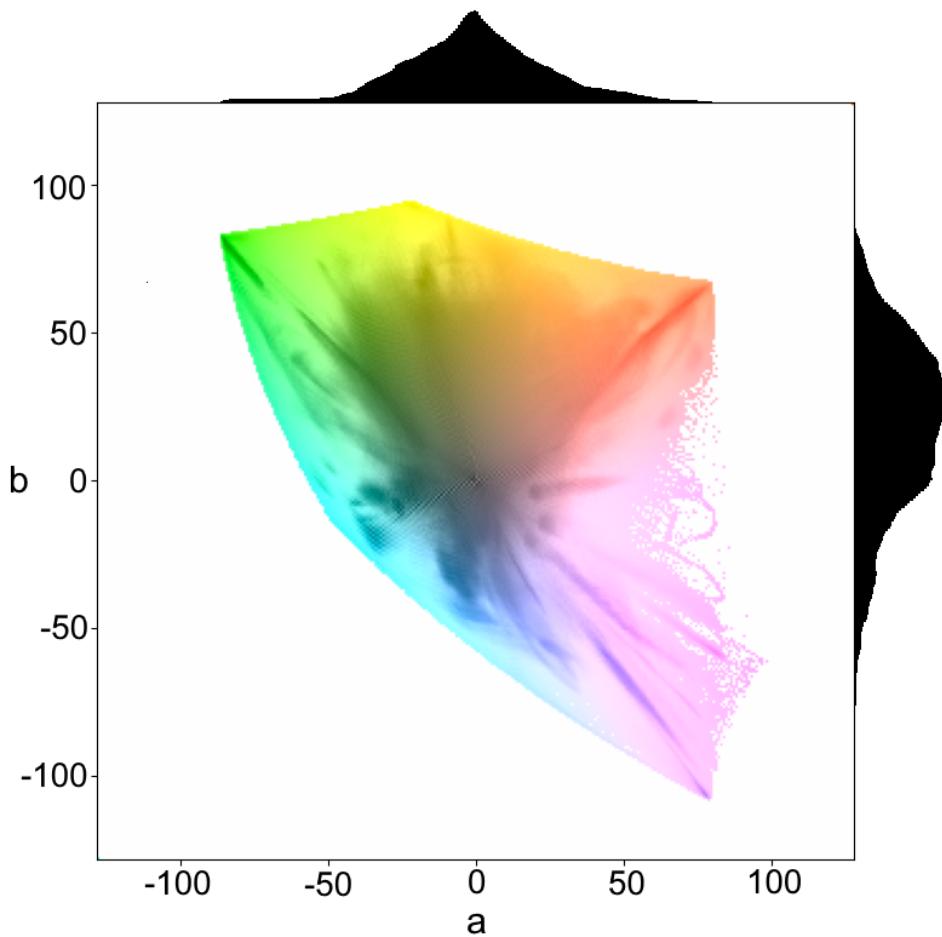


Figure 4: Color distribution of the dataset in the LAB color space

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 19 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.

2.1.3 Multi GPU Training

To enhance the training speed of larger models, utilizing multiple GPUs can be significantly more efficient. Consequently, the model's code is 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.

2.2 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.

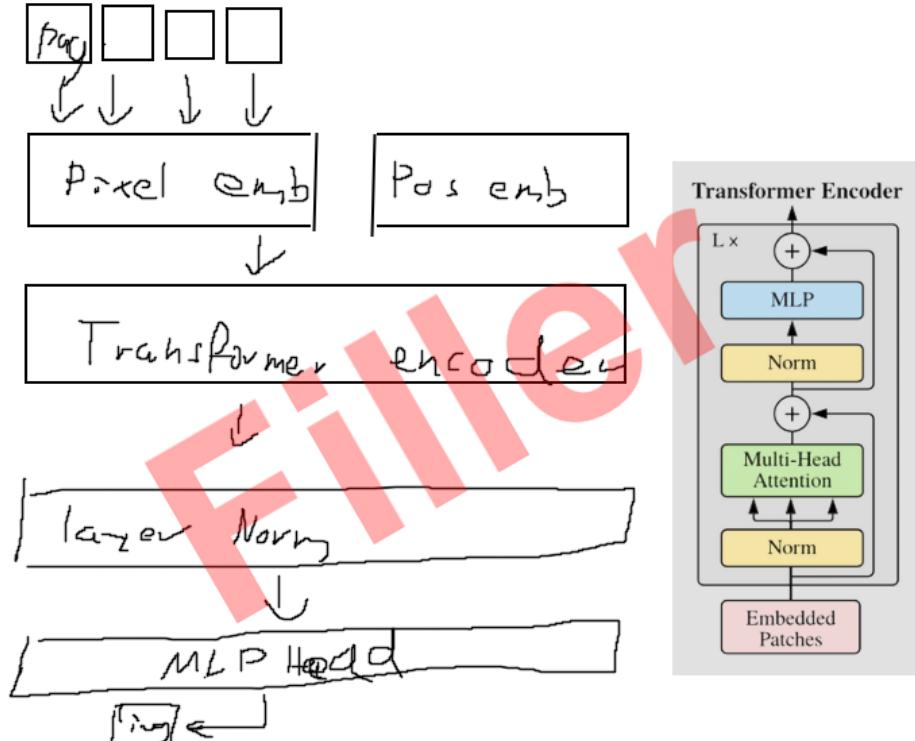


Figure 6: Column Image Transformer Model

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 Channels, 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 Channels). 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.

```

1 # data: (Batch, Color, Height, Width)
2 def get_batch(data):
3
4     source = data[:, :BLOCK_SIZE, :BATCH_SIZE]
5     targets = data[:, 1:BLOCK_SIZE+1, :BATCH_SIZE]
6
7     source = rearrange(source, 'c h b -> b h c')
8     targets = rearrange(targets, 'c h b -> b h c')
9
10    return source, targets

```

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_{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_{EMBD} ($\frac{1}{5}N_{EMBD}$), then increasing it to one-half of N_{EMBD} ($\frac{1}{2}N_{EMBD}$) until reaching N_{EMBD} . The pixel embedding layer is followed by a dropout layer with a dropout rate of 0.2. See Figure 8.

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_{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_{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.

```

1  class ColumnTransformer(nn.Module):
2      self.pixel_embedding = nn.Sequential(
3          nn.Linear(CHANNELS_IMG, N_EMBD//5, device = device),
4          nn.ReLU(),
5          nn.Linear(N_EMBD//5, N_EMBD//2, device = device),
6          nn.ReLU(),
7          nn.Linear(N_EMBD//2, N_EMBD, device = device),
8          nn.Dropout(DROPOUT),
9      )
10     self.position_embedding_table = nn.Embedding(BLOCK_SIZE, N_EMBD)
11     #[...]
12
13 def forward(self, source, targets=None):
14
15     # tok_emb: (B, H, N_EMBD)
16     tok_emb = self.pixel_embedding(source)
17
18     # pos_emb: (H, N_EMBD)
19     pos_emb = self.position_embedding_table(torch.arange(H))
20
21     # x: (B,H,N_EMBD)
22     x = tok_emb + pos_emb
23     #[...]
```

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.5 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. The sigmoid function typically 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 inputs need to be extremely large or small 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 the issue of vanishing gradients during backpropagation. A problem where gradients become so small during backpropagation that the model stops learning. This occurs because the derivative of the clamp function outside its bounds is zero, which stops the gradient flow and prevents weights from updating. Consequently, layers deeper in the network receive minimal or no updates, hindering the models training process.

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 became 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 evi-

dence 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 Training the Model

The training process starts by selecting the hyperparameters and arguments for the model. To test the basic functionality of the model, the training process is kept simple, and they are trained on a single GPU. Most of the training is done via the Slurm cluster, which uses a batch script to configure the hardware and training parameters. All the parameters can be found in detail in the appendix subsection A.3.

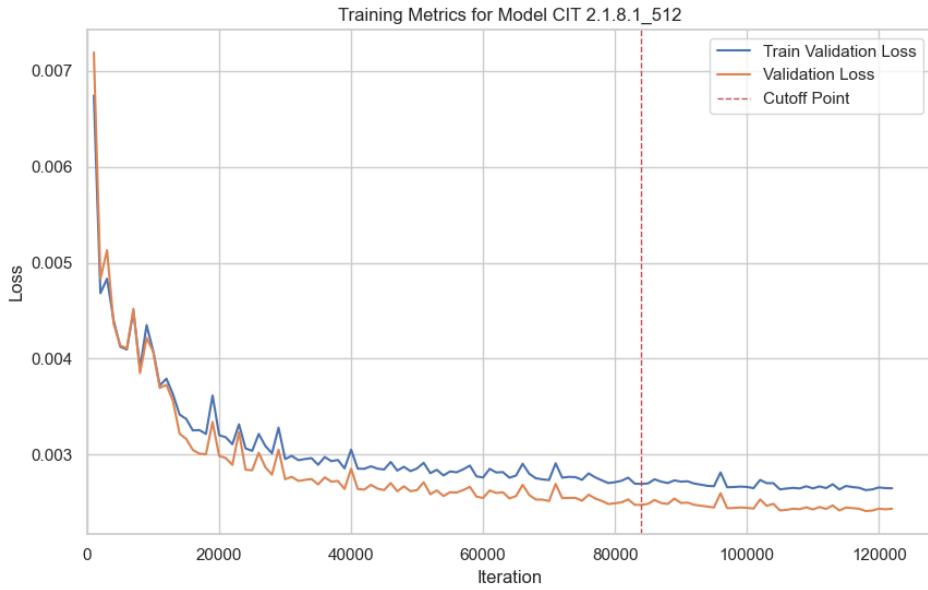


Figure 9: Column Image Transformer Model

The figure above displays the training metrics for the Column Image Transformer Model [3] CIT 2.1.8.1_512, illustrating the loss over iterations during the training phase. The blue line signifies the training validation loss, and the orange line represents the validation loss throughout the training iterations. The graph shows a consistent decline in both metrics, indicating that the model is learning and improving its predictive accuracy over time. The 'Cutoff Point,' indicated by the dashed red line, marks the iteration at which the model is used in the examples presented in this thesis since it has not shown further improvement beyond this point. An unusual behavior observed is that the validation loss is lower than the training validation loss, which is uncommon. This could be the case due to various factors, such as a high dropout rate and needs further investigation.

2.2.10 Experimental Results

Multiple experiments are conducted to evaluate the performance of the CIT model. These experiments are designed to test the model's ability to generate images, the quality of the generated images, and the model's performance across different scenarios. The results are analyzed to identify the strengths and weaknesses of the model.

In each test, the end of the seed pixels are marked with a purple line on the left side of the image.

- **Color test:** To assess the model's capability to generate specific colors, it is challenged with 11 different images with a width of 32 and a height of 128 pixels. Each test image is composed of a single color filling 128 positions in the model's context.

The model then generates the next 32 pixels for each image.

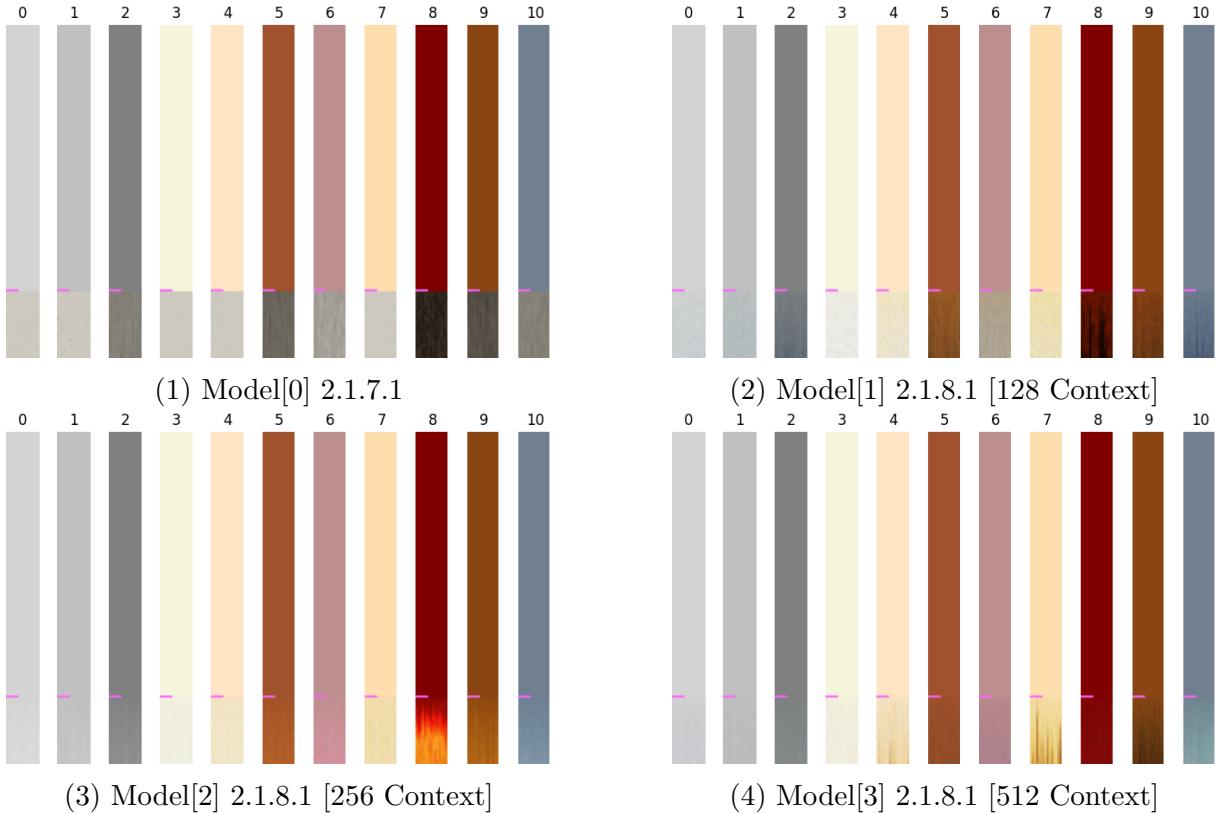


Figure 10: Color test with four different models

As shown in the images above, Model [0] struggles to generate the correct colors due to the limited size of its old small dataset compared to Models [1-3]. Model [1] generally produces accurate colors but faces difficulties with some. Model [2] generates all colors correctly, but inconsistencies start appearing in column 8. The model may begin changing colors due to its training to recognize patterns such as flowers or carpets in the training data. Model [3] consistently generates the correct colors but begins to create patterns in some columns, potentially indicating that the model is overfitting.

- **Zebra pattern test:** The second test assesses whether the model can generate a zebra pattern. The model is provided with both a vertical and a horizontal test image featuring black and white stripes, each stripe being 4 pixels wide. It then generates the next 32 pixels.

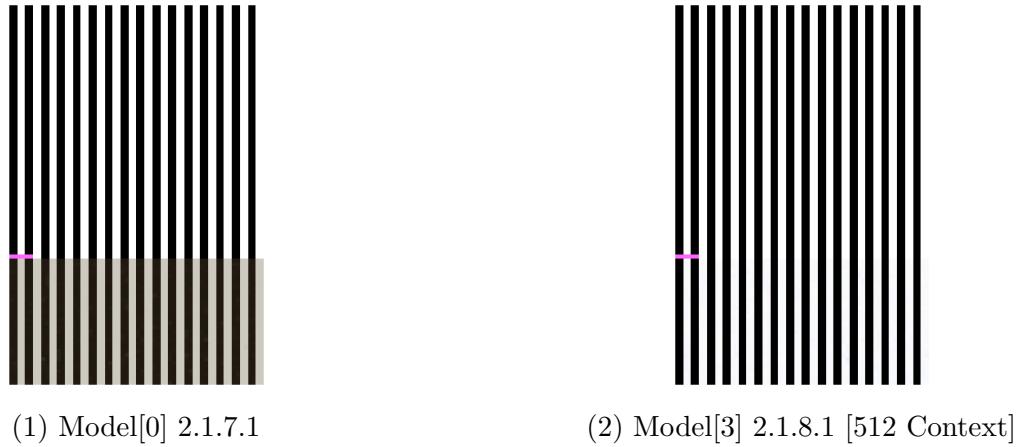


Figure 11: Zebra test vertical with two different models

As shown in the images above, the model with the newer dataset (Model[3]) generates the colors in the zebra pattern more accurately than the model with the older dataset (Model[0]). Models [1-3] perform similarly. The vertical test is particularly interesting to observe when the column model is altered to see pixels from both the left and the right sides of the current column, assessing whether it can still perfectly continue generating the pattern.

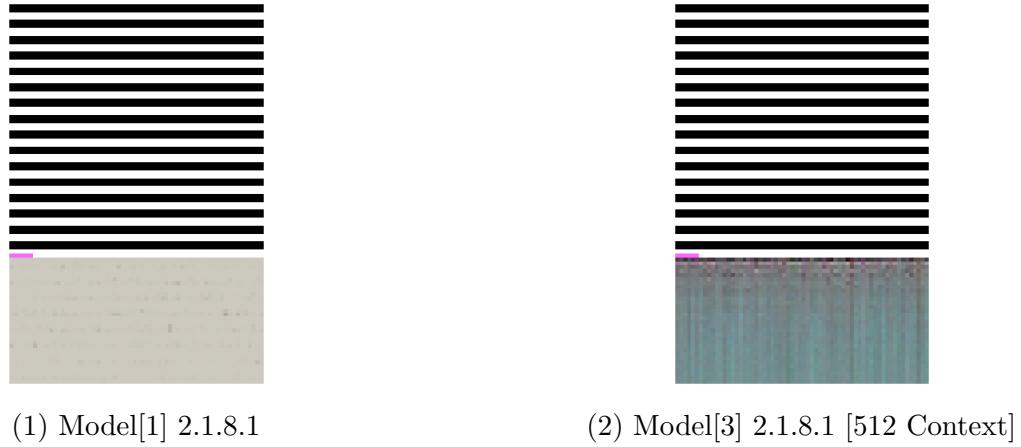


Figure 12: Zebra test horizontal with two different models

Unfortunately, the model struggles to generate the zebra pattern horizontally. The model with the newer dataset (Model[3]) more accurately renders the colors in the zebra pattern than the model with the older dataset (Model[0]), but both models struggle to correctly continue the pattern. Model [3] can recognize that the next pixel in the pattern should be dark but quickly loses context. This could be due to the dataset being too small or the model size being insufficient to fully understand the pattern. The images suggest that Model [3] is better than Model [1] but still fails to generate the pattern correctly.

2.2.11 Challenges and Limitations

The CIT model has some limitations and challenges that require attention. The primary issue is its limited context to only one column, which hampers its ability to generate complex patterns requiring a broader visual spectrum. Additionally, similar to some text-based Transformer models, the CIT model struggles with generating coherent long sequences, often using its recent outputs as a basis for predicting subsequent pixels. This can lead to errors accumulating over time, particularly evident in longer image sequences where the model fails to maintain the specific pattern.

To address these limitations, methods used in text-based models, such as scaling up the model size and expanding the training dataset, can be considered. Scaling the model could involve increasing the dimensionality of the embeddings or adding more layers, which might help the model capture more complex dependencies. However, expanding the dataset was not feasible in this thesis due to time constraints and the extensive duration of the dataset collection process.

Introducing a text prompt feature could significantly enhance the usability and flexibility of the CIT model. This feature would allow users to guide the generation process using descriptive language, making it easier to specify desired outcomes without relying solely on seed pixels. Such a capability would not only improve user interaction but could also potentially help the model deal with complex pattern generation by providing additional contextual information. Exploring this possibility could be a valuable direction for future research.

2.3 Spiral Image Transformer

The Spiral Image Transformer, as implied by its name, is a transformer-based model specifically designed to process image data by unrolling pixels in a spiral pattern. This approach stands in contrast to the CIT Model, which processes data using a column-by-column method.

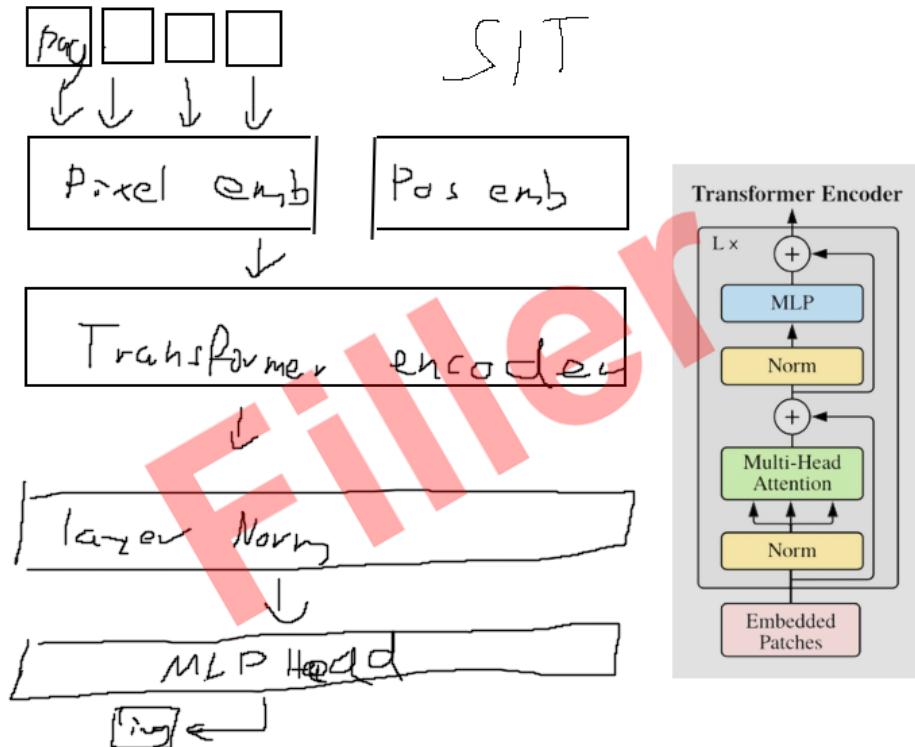


Figure 13: Spiral Image Transformer Model

The visual representation above illustrates the operational steps of the SIT model. Similar to the CIT model, it begins by loading and converting the data. However, it diverges in its method by transforming the data into a spiral format, instead of using the columnar approach employed by the CIT. Like its counterpart, this model embeds pixels, from 3 colors to a higher-dimensional embedding space represented by N_{EMBD} . Positional embeddings are then added. The embedded data is further processed by a (MLP), which converts it back into the original 3-color format. Finally, the spiral-formatted data is restructured into the original image layout.

2.3.1 Spiral Data Processing

To transform the data, which is loaded by the DataLoader and iterated over, into a spiral format, the width and height dimensions are first squeezed into a single dimension. This

resulting tensor has the shape (Batch, Color Channels, Spiral Length). The next step involves indexing the data into the spiral form. For more detailed information on this indexing process, refer to Figure 16. The restructured tensor is then processed like how the data is handled in the Column Image Transformer. The following code block illustrates the process used to convert the data into a spiral format.

```

1 def get_batch(data):
2     # Batch_size, Color_Channels, Height, Width
3     B, C, H, W = data.shape
4
5     spiral_data = torch.zeros_like(data.view(B, C, -1))
6
7     spiral_data[:, :, spiral_indices.flatten()] = data.view(B, C, -1)
8
9     source = spiral_data[:, :, :BLOCK_SIZE]
10    label = spiral_data[:, :, 1:BLOCK_SIZE+1]
11
12    source = rearrange(source, 'b c h -> b h c')
13    label = rearrange(label, 'b c h -> b h c')
14
15    return source, label
16

```

Figure 14: Get data in a spiral form for the SIT model

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 training step. Therefore, a simple nested for loop is insufficient to rearrange the data into the desired form. In this example, fancy indexing is used to convert the data into a spiral form. Thus, the data tensor is indexed with a flattened 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.

```

1 def create_spiral(n): # n = width and height
2
3     matrix = [[0] * n for _ in range(n)] # Initialize n x n matrix
4
5     x, y = 0, 0
6     # Direction vectors (right, down, left, up)
7     dx = [0, 1, 0, -1]
8     dy = [1, 0, -1, 0]
9     direction = 0
10
11    for i in range(n * n - 1, -1, -1): # Start (35 for 6x6)
12        matrix[x][y] = i
13        nx = x + dx[direction]
14        ny = y + dy[direction]
15
16        # Change direction if the next position: out of bounds or filled
17        if nx < 0 or nx >= n or ny < 0 or ny >= n or matrix[nx][ny] != 0:
18            direction = (direction + 1) % 4 # Change direction
19            nx = x + dx[direction]
20            ny = y + dy[direction]
21
22        x, y = nx, ny
23
24    return torch.tensor(matrix)

```

Figure 15: 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.

The generated 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.

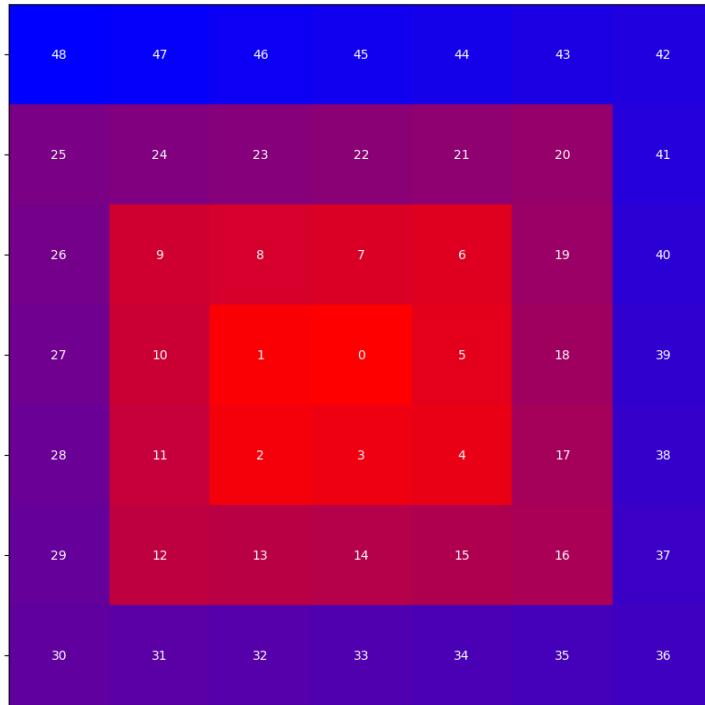


Figure 16: Image representation

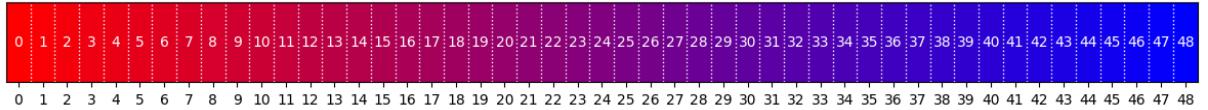


Figure 17: 7x7 Image flattened into a spiral form

As shown in Figure 16, the pixels of the image are labeled with their respective indices 0, ..., 43. The image is then unrolled into a single dimension, as shown in Figure 17. 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 a width of 7, yet the underlying process remains unchanged.

2.3.3 Model Architecture

Similar to the CIT model, the addition of positional embeddings is handled in the same manner. This is because the data is transformed into a spiral pattern, allowing the addition of positional embeddings to remain unchanged. Additionally, the transformer layer operates identically. For more information on the transformer layer, see the CIT model subsubsection 2.2.4.

2.3.4 Training the Model

2.3.5 Training Protocols Discuss the training process, highlighting any unique considerations for the spiral data format.

2.3.6 Optimization and Loss Functions Specific optimization techniques and loss functions that are effective for spiral-based models.

2.3.5 Experimental Results

Multiple experiments are conducted to evaluate the performance of the SIT model. The experiments are designed to test the models ability to generate images, the quality of the generated images, and the models performance on different datasets. The results are analyzed to identify the strengths and weaknesses of the model.

In each of the tests, the end of the seed pixels is marked with a purple line on the left side of the image.

- **Color test:** To test the models capability to generate certain colors the model is given a test image with one single color filling the whole context. Then the model generates the next flowing 32 pixels.

TODO: Add color test images models output

- **Zebra pattern test:** The second test is to see if the model is capable of generating a zebra pattern. The model is given a vertical and a horizontal test image with a black and white colored zebra pattern with a pixel width of 3. The model then generates the next 32 pixels.

TODO: Add zebra pattern test images models output

- **Spiral Pattern test:** The third test is to see if the model is capable of generating a full image. The model is given a test image with a 4x4 pixel pattern. The model then generates the next 32 pixels.

2.3.6 Challenges and Limitations

2.3.9 Identified Challenges Discuss specific challenges encountered during the development and implementation phases.

2.3.10 Limitations of the Spiral Transformer Critically analyze the limitations and potential drawbacks of using a spiral approach.

3 Conclusion

This section focuses on the evaluation of the performance of the models and offers detail on the further implementations and extenstions that would make sense.

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. Especially the CIT Models is able to generate new assets in a short amount of time due to the fact that it is not seeing the whole image at once. This is also a big disadvantage because it is not able to envision the hole context. In contrast to this is the Spiral Model which is able to see the whole image at once but is not able to generate new assets in a short amount of time du to the fact that a context window will get exponeltioly big $\hat{2}$ widht height. The best approach would be to find a middle ground between the two models. For example to use a wider width then on in the CIT model or to feed additional information to the model like the x position of the to generate pixel.

another addition to the model could be to use a text input as an aditional information. This would be an easy approch to generate more specific assets and it is easier to start form a smaller seedpixel.

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

The following images show generated assets from the CIT and the SIT model.

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

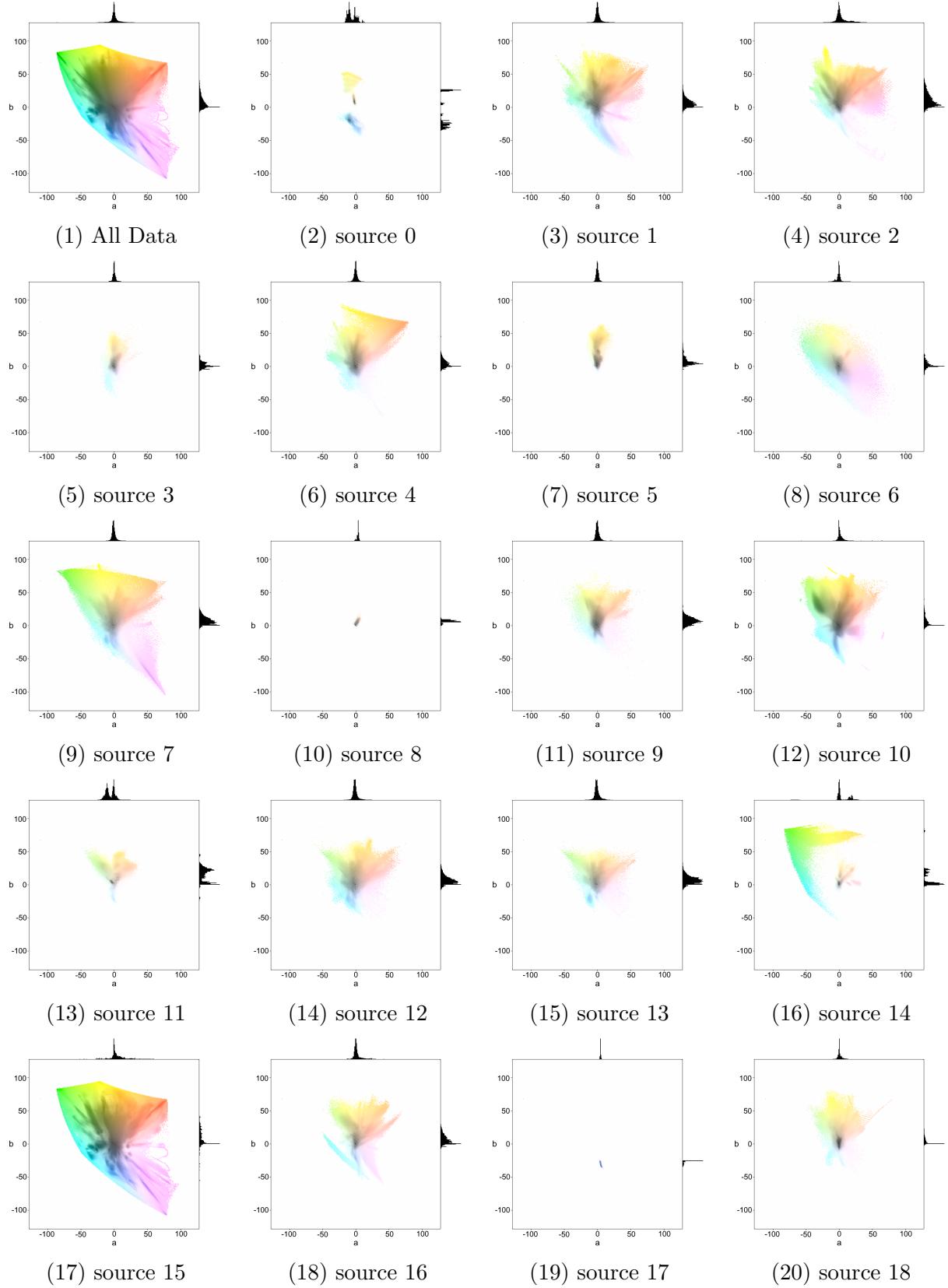
content is artifice or an original image. In a perfect scenario the discriminator can't distinguish between the generated content and the original content and the loss will balance out at 50

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



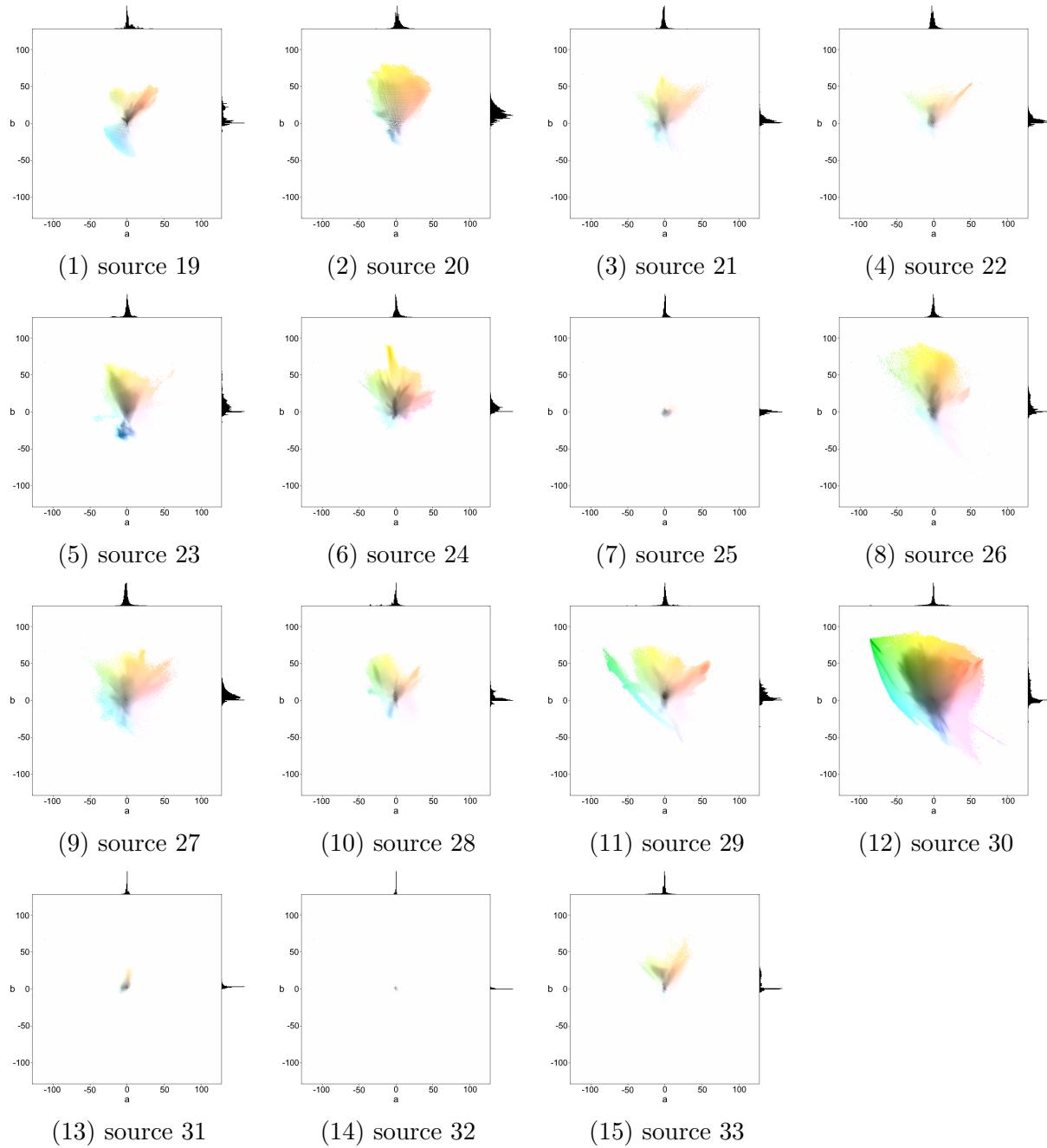
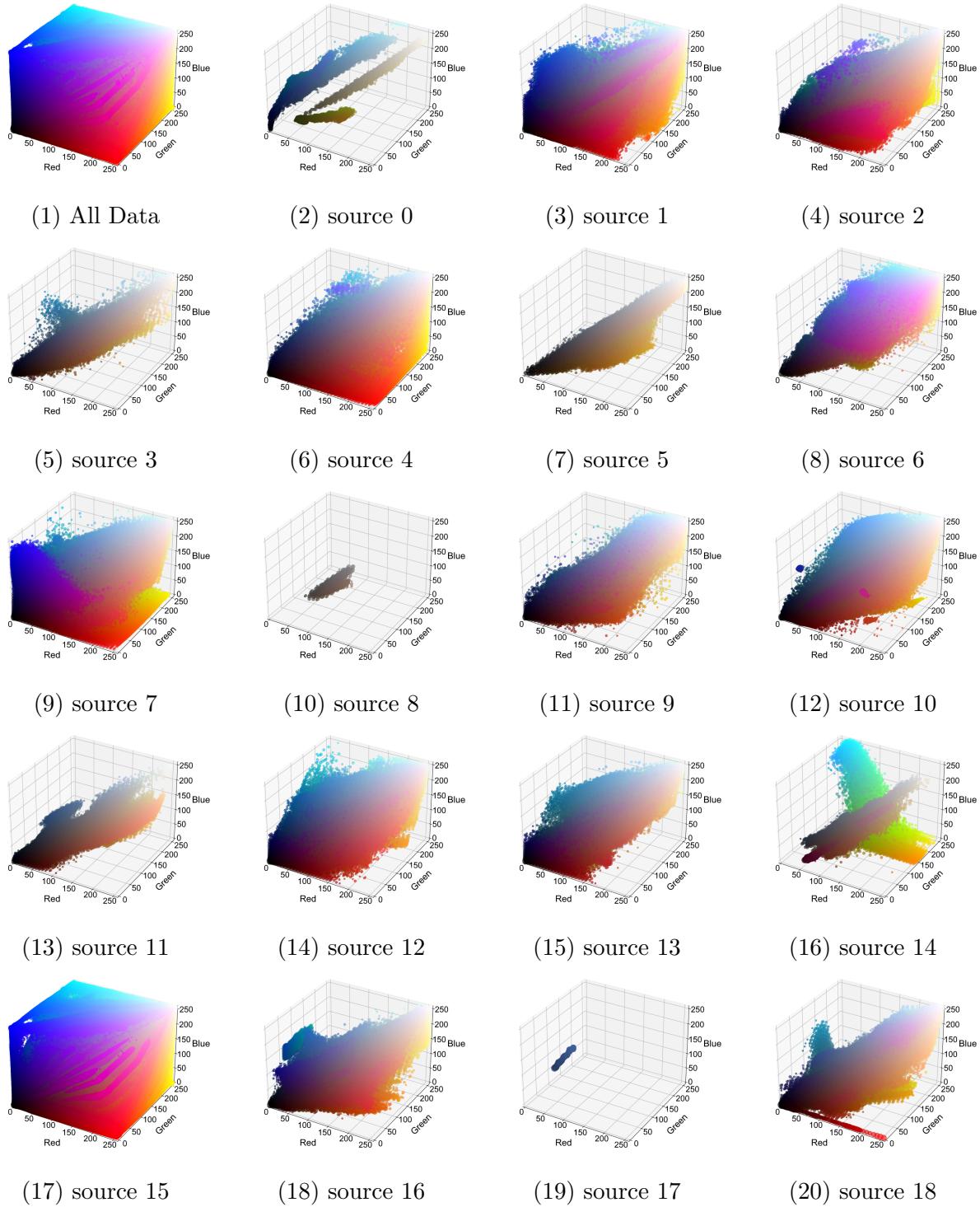


Figure 18: Visualization of datasets plotted in the LAB color space

A.2 Dataset Plots in the RGB Color Space



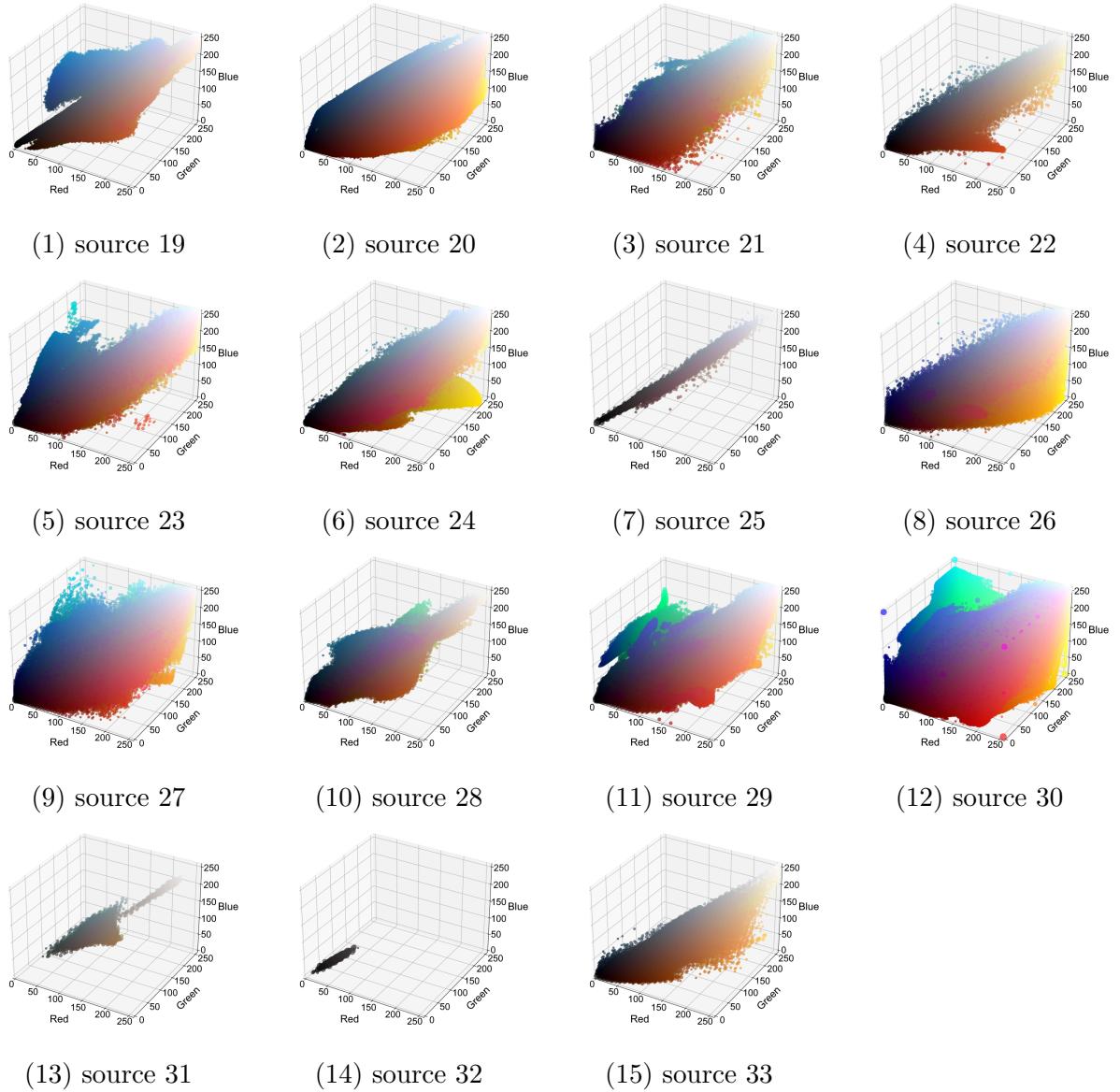


Figure 19: Visualization of datasets plotted in the RGB color space

A.3 Trained Models with hyperparameters

A.3 Trained Models with hyperparameters

A APPENDIX

Table 2: Model Configuration Overview

MODEL	Nr.	VER.	BATCH_SIZE	BLOCK_SIZE	N_EMBD	N_HEAD	N_LAYER	PARAMETER*	GPUS	DATASET	VAL_LOSS**
CLT	0	2.1.7.0	128	128	64	6	6	6	1.22 m	1	old Dataset
	1	2.1.8.0	64	64	64	6	6	6	1.21 m	1	x512 Dataset
	2	2.1.8.0 Classify	64	128	128	6	6	6	1.25 m	1	x512 Dataset
	3	2.1.8.0	64	256	512	8	8	8	25.66 m	4	x512 Dataset
	4	2.1.8.0	32	511	640	9	9	9	45.09 m	4	x512 Dataset
LIS	5	5.0.1.0	8	4,095	128	6	6	6	1.73 m	1	x512 Dataset
	6	5.0.1.0	4	4,095	512	8	8	8	27.62 m	1	x512 Dataset

* Figures in millions

** Average of the last 5 validation loss values

A.4 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 3: 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 4: Single compute node

A.5 Transformer layer implementation

```

1  class Head(nn.Module):
2      """ one head of self-attention """
3
4      def __init__(self, head_size):
5          super().__init__()
6          self.key = nn.Linear(N_EMBD, head_size, bias=False)
7          self.query = nn.Linear(N_EMBD, head_size, bias=False)
8          self.value = nn.Linear(N_EMBD, head_size, bias=False)
9          self.register_buffer('tril', torch.tril(torch.ones(BLOCK_SIZE,
10                                         BLOCK_SIZE)))
11
12          self.dropout = nn.Dropout(DROPOUT)
13
14      def forward(self, x):
15          # input of size (batch, time-step, channels)
16          # output of size (batch, time-step, head size)
17          B,T,C = x.shape
18          k = self.key(x)    # (B,T,hs)
19          q = self.query(x) # (B,T,hs)
20          # compute attention scores ("affinities")
21          wei = q @ k.transpose(-2,-1) * k.shape[-1]**-0.5 # (B, T, hs) @
22          (B, hs, T) -> (B, T, T)
23          wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-inf')) # (
24          B, T, T)
25          wei = F.softmax(wei, dim=-1) # (B, T, T)
26          wei = self.dropout(wei)
27          # perform the weighted aggregation of the values
28          v = self.value(x) # (B,T,hs)
29          out = wei @ v # (B, T, T) @ (B, T, hs) -> (B, T, hs)
30
31      return out
32
33
34  class MultiHeadAttention(nn.Module):
35      """ multiple heads of self-attention in parallel """
36
37      def __init__(self, num_heads, head_size):
38          super().__init__()
39          self.heads = nn.ModuleList([Head(head_size) for _ in range(
40          num_heads)])
41          self.proj = nn.Linear(head_size * num_heads, N_EMBD)
42          self.dropout = nn.Dropout(DROPOUT)
43
44      def forward(self, x):
45          out = torch.cat([h(x) for h in self.heads], dim=-1)
46          out = self.dropout(self.proj(out))
47
48      return out

```

```

43 class FeedFoward(nn.Module):
44     """ a simple linear layer followed by a non-linearity """
45
46     def __init__(self, n_embd):
47         super().__init__()
48         self.net = nn.Sequential(
49             nn.Linear(n_embd, 4 * n_embd),
50             nn.ReLU(),
51             nn.Linear(4 * n_embd, n_embd),
52             nn.Dropout(DROPOUT),
53         )
54
55     def forward(self, x):
56         return self.net(x)
57
58 class Block(nn.Module):
59     """ Transformer block: communication followed by computation """
60
61     def __init__(self, n_embd, n_head):
62         # n_embd: embedding dimension, n_head: the number of heads we'd
63         like
64         super().__init__()
65         head_size = n_embd // n_head
66         self.sa = MultiHeadAttention(n_head, head_size)
67         self.ffwd = FeedFoward(n_embd)
68         self.ln1 = nn.LayerNorm(n_embd)
69         self.ln2 = nn.LayerNorm(n_embd)
70
71     def forward(self, x):
72         x = x + self.sa(self.ln1(x))
73         x = x + self.ffwd(self.ln2(x))
74
75     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., Babaee, 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.