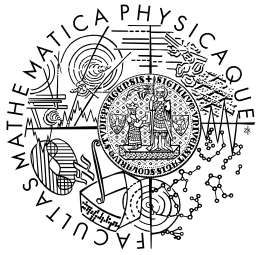

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

BACHELOR THESIS

Matěj Mrázek

Generative neural networks for sky image outpainting

Department of Software and Computer Science Education

Supervisor of the bachelor thesis: doc. RNDr. Elena Šikudová, Ph.D.
Study programme: Computer Science
Study branch: Artificial Intelligence

Prague 2023

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my family for their support, my supervisor, Elena Šikudová, for her guidance and feedback, the Computer graphics group at Charles University for providing me with the computational resources and data to train the models for this thesis, and finally, Milan Straka, for his amazing machine learning lectures that motivated me to pursue a degree in the field.

Title: Generative neural networks for sky image outpainting

Author: Matěj Mrázek

Department: Department of Software and Computer Science Education

Supervisor: doc. RNDr. Elena Šikudová, Ph.D., Department of Software and Computer Science Education

Abstract: Image outpainting is a task in the area of generative artificial intelligence, where the goal is to expand an image in a feasible way. The goal of this work is to create an machine learning algorithm capable of sky image outpainting by implementing several recently proposed techniques in the field. We train three models, a tokenizer for converting images to tokens and back, a masked generative transformer for performing outpainting on tokens and a super sampler for upscaling the result, all on a dataset of sky images. Then, we propose a procedure that combines the trained models to solve the outpainting task. We describe the results of training each model and those of the final algorithm. Our contribution consists mainly in providing a working, open-source implementation including the trained models capable of sky image outpainting.

Keywords: image transformer sky imagery outpainting

Contents

Introduction

Image outpainting is an interesting task in the area of generational artificial intelligence. It can be thought of as follows - we have an input image, and add any amount of white rows or columns. The task is then how to recolor the added white pixels in such a way that makes the whole image look feasible. In this thesis, we attempt to create an AI capable of outpainting sky images, in order to improve the capabilities of the SkyGAN model [REF. Also, is this correct?].

In the recent years, there were multiple forays into the area of image generation, most notably DALL-E 2 [research [Ramesh et al., 2022], web [DALL-E 2 web]], Midjourney [[Midjourney web]], and Stable diffusion [[Stable diffusion web]]. These models solve a more general problem than we do - they support a prompt input, describing what should be present in the image, and generate an image based on it. We recognize that there have been many recent advances in diffusion model architectures that achieve record scores on image generation and which could be modified to perform sky image outpainting, however, unlike DALL-E 2 and Stable diffusion with a good prompt, they do not support them out of the box.

In this work, we attempt to create an open-source image outpainting algorithm that can surpass DALL-E 2 and Stable diffusion on the very specific task of outpainting sky images. We use a model inspired by MaskGIT [Chang et al., 2022] for outpainting the images, and then a diffusion model based on [Saharia et al., 2022] to upscale the results and add some details. Because we only need to generate sky images, we can simplify our models by not dealing with any class-conditional generation and simply training everything exclusively with a sky image dataset, which we create ourselves from a large amount of webcam images.

In Chapter 1, we will introduce multiple topics from machine learning that will be later used in this thesis, including multi-layer perceptrons, convolutional networks, transformers, and diffusion models. Chapter 2 describes how we prepare the dataset before training. In Chapter 3, we present all models we use in this work - this contains the VQVAE tokenizer, which is required for training the second model, MaskGIT, and finally, the diffusion model super sampler. Chapter 4 then describes how the individual models comprise the final algorithm.

1. Background

In this chapter, we familiarize the reader with several topics from machine learning that are used in this thesis. This includes a brief introduction to neural networks and some more advanced architectures such as convolutional networks or transformers. Finally, we shortly introduce diffusion models, the model type we use for upscaling images.

1.1 Neural networks

Neural networks are a type of machine learning model, composed of multiple layers (each of which is parametrized by some weights) and activation functions, each one represented by a mathematical operation. Each layer either operates on the output from a previous layer or on the model input, making the whole model a composite function, producing an output given input and the model weights.

The model trains by tweaking the weights of its layers to minimize a loss function, which measures how well the model performs on a given task.

We first introduce the dense layer and ReLU activation function, on which we show the basics of how a model learns. After that, we show more advanced layers that will be later used for the models used in this work.

1.2 Multi-layer perceptron

The simplest neural network one can build is called a multi-layer perceptron. It consists of an input $I \in \mathbf{R}^n$, one dense hidden layer, and one dense output layer. A dense layer has a matrix of weights $W \in \mathbf{R}^{m \times n}$ and a vector of biases $b \in \mathbf{R}^n$, where m/n are the input/output dimensions respectively, and a specified activation function σ. The layer output y is then defined as $y = \sigma(x^T \cdot W + b)$, where σ is applied to each element of the output vector individually. A simple dense neural network is shown in Figure 1.1.

When illustrating complex neural networks, we do not want to visualize every single element of the output - we instead just represent individual operations, with arrows indicating which tensors are used as inputs. Consider the network from Figure 1.1 - a more compact representation is shown in Figure 1.2. All networks used in this work are presented in this way.

When we provide the input value x_{input}, we can compute the value of the first dense layer using its formula, then we repeat the same for the second dense layer, producing the network output.

1.3 Training

To train the network, we need a dataset containing examples that can be passed as input to the first layer and their respective outputs that define what the model should return for the given value. We also need to provide a loss function L, which serves as a measure of how far our model outputs are from the correct ones, defining what value should our model minimize. Commonly used loss functions include

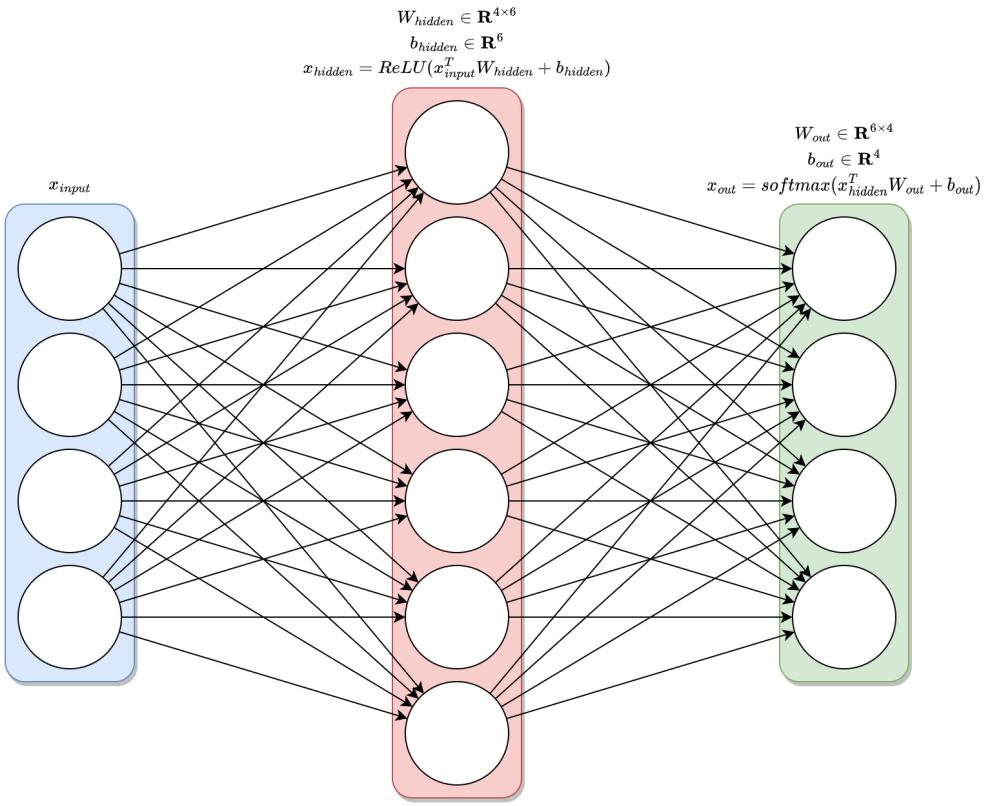

Figure 1.1: *A simple MLP network, consisting of three layers - one input layer with 4 units, one dense hidden layer with 6 units and a* ***ReLU*** *activation, and a dense output layer with 4 units and a* ***softmax*** *activation. Each layer is parametrized by weights W and biases b. Above each layer, we can see how its output is computed. Each arrow represents one connection - a single multiplication between a weight and a value.*

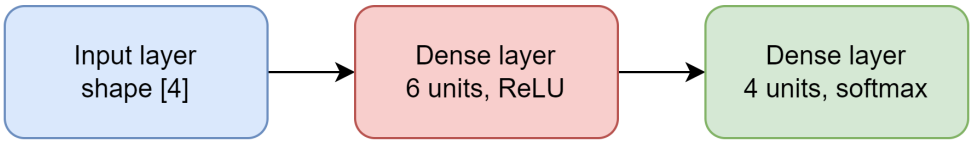

Figure 1.2: *A more compact representation of the MLP from Figure 1.1*

mean squared error for regression tasks and sparse categorical cross-entropy for classification.

Neural networks are trained using gradient descent - we want to compute the derivative of the loss function relative to each weight, and then change each weight slightly in the direction that reduces loss the most based on the current training example:

$$w' = w - \alpha \cdot \frac{\partial L}{\partial w}$$

The parameter α is called the learning rate, and it represents how fast our

model should learn.

We begin by passing the example to the network input, evaluating the outputs, and computing the loss function. After that, we compute the derivatives of all weights using the **back-propagation** algorithm, explained in more detail in the Deep learning book, further referred to as DLB, [Goodfellow et al., 2016], section 6.5.

In practice, there are two major distinctions - the gradient is averaged over multiple examples, called a batch, to produce a more accurate estimate, and the weights are updated using an optimizer - an algorithm that is provided with the gradient, which it can modify it in some way, and only then updates the weights. The optimizers we use in this work are all variations of Adam [described in Section 8.5.3 of DLB [Goodfellow et al., 2016]], which also tracks the momentum of the gradient over multiple batches, and takes this information into account when updating network weights.

1.4 Convolutional networks

1.4.1 Convolution basics

The aforementioned dense layers have several problems for image data - the image has to be flattened to a large one-dimensional vector first, completely losing any spatial data, and using another dense layer on top requires an enormous amount of weight parameters. To alleviate these issues, convolutional layers were introduced in AlexNet [Krizhevsky et al., 2012].

A convolutional layer can be viewed as an extension of the discrete convolution operation between two images. When using one input and one output layer, the output at a given position is computed by multiplying the surrounding pixels in the input by a convolutional matrix and applying an activation to the result, the values of which are the same for every position in the output and which represent the trainable weights. We visualize this operation in Figure 1.3. When using multiple input channels, each layer in the input has a separate convolution matrix, and the convolution results are summed before applying the activation. When using multiple outputs, each additional layer is computed in the same way as the first, just using a different matrix of weights.

Convolutional layers are great at working with data locally, but unless an extensive amount of layers is used, they have no way to propagate information to the other side of the image. For this reason, downscaling layers were introduced - by using a greater distance (called stride) between individual matrix placements in the input, they can downscale the image, compressing the information to a smaller size. When upscaling is required instead, a transposed convolution can be used - it can be thought of as inserting zeroes after each element in the input matrix, effectively upscaling it, and then using a basic convolution.

1.4.2 Residual connections and normalization

Convolutional networks require many layers, each of which performs a small part of the whole computation. For large networks, possible issues like *van-*

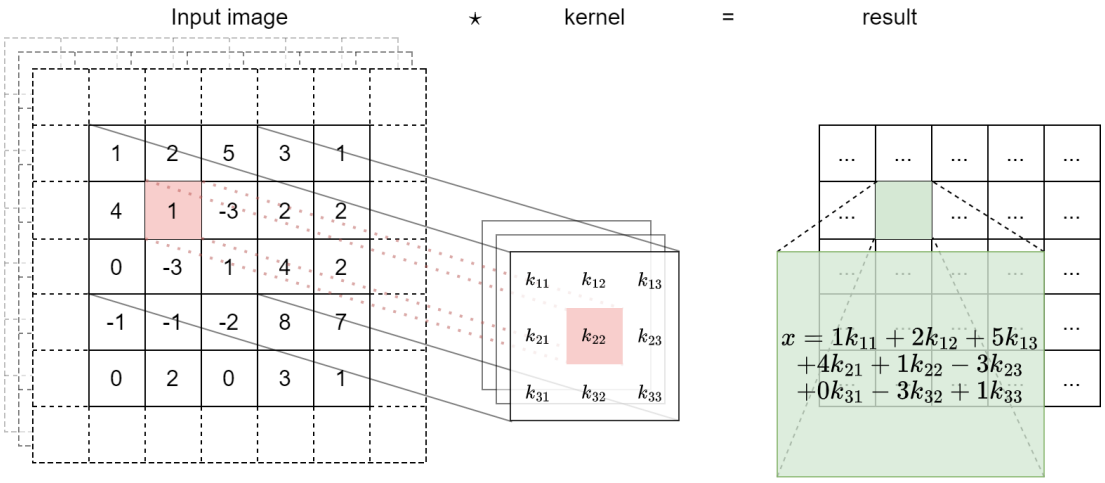

Figure 1.3: *Using a convolutional kernel at coordinates* $(1, 1)$*. The same kernel would be used for all other positions as well. Multiple input dimensions would each correspond to a separate kernel, forming a kernel stack; multiple output dimensions would use a different kernel stack each.*

ishing/exploding gradients[1] and increasingly more complex loss landscapes (visualized in [Li et al., 2017]) become more prominent. To address this, ResNet [He et al., 2015] proposes residual connections, which boil down to taking an input, applying a series of layers, and then adding the result to the original input, forming a residual block. In the context of convolutional networks. A possible configuration is shown in Figure 1.4.

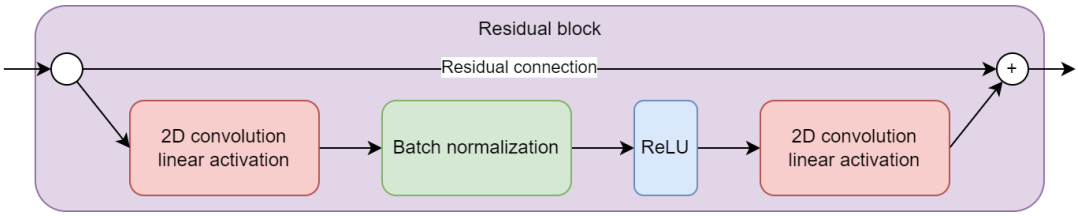

Figure 1.4: *A possible residual block configuration. The bottom, processing part consists of a 2d convolution operation, a normalization (batch norm is used, but layer norm is common as well), a ReLU activation, and another convolution. After that, the result is added to the input. In cases when we want to change the number of filters or downscale the image, we add a convolutional block on the residual connection with linear activation.*

Another thing helping with training is new forms of regularization in the form of normalization layers, which all take some input data and try to shift and scale it so that it has a unit variance and a zero mean. The first to be proposed was a batch normalization layer [Ioffe and Szegedy, 2015], tracking the mean and variance of each layer in a convolutional network, and shifting the values in

[1]During backpropagation, the gradient is multiplied by the weights of the current layer before being propagated back. When there are many layers, and their weights are either too large or too small, the gradient as a whole can become significantly smaller or larger during backpropagation, resulting in unstable or slow training.

each layer to conform to the properties mentioned above. More advanced forms of normalization appeared later, such as layer normalization [Ba et al., 2016] and group normalization [Wu and He, 2018], averaging values in each example individually and over groups of layers in an example respectively.

1.5 Transformers

Transformers were originally introduced in the field of natural language processing as a successor to recurrent networks (first mention was in the BERT article [Devlin et al., 2018]). A transformer layer allows working with sequences of elements, where any element of a sequence can share information with all other elements at once. A typical transformer layer consists of two parts, a multi-head self-attention, which is responsible for interactions between multiple elements, and a dense layer processing part, which processes each sequence element individually.

The attention is implemented in the following way - for each element in the sequence, the input is a vector $x \in \mathcal{R}^n$, n is dubbed `hidden_size`. The transformer parameters consist of three matrices, the key matrix $\mathbf{W^K} \in \mathcal{R}^{n \times d_k}$, value matrix $\mathbf{W^V} \in \mathcal{R}^{n \times d_v}$ and query matrix $\mathbf{W^Q} \in \mathcal{R}^{n \times d_k}$. These are used to compute the keys $\mathbf{K} = \mathbf{W^K}x$, values $\mathbf{V} = \mathbf{W^V}x$ and queries $\mathbf{Q} = \mathbf{W^Q}x$ - queries representing what we are looking for, keys representing what others should search for if they want to find the value in this place, and value, representing what will be found at this position. We can then compute the layer output as follows:

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

For reasons mentioned in previous sections, we also have a residual connection from x to layer output and apply a layer normalization afterward. We also use multiple attention heads - this means doing the attention operation multiple times in parallel, with different query, key, and value matrices, concatenating the result, and using one last dense layer on every sequence element to project the result to the required size, before adding the residual value. The whole transformer layer is shown in Figure 1.5.

The processing layer is much simpler - it simply applies a dense layer with a different count of units, called `intermediate_size`, then an activation function, and finally another dense layer with `hidden_size` units. We then have another residual connection from the first hidden layer input, add both values together, and apply another layer normalization, obtaining the transformer layer output.

1.6 Embeddings

We often need to represent discrete data and work with them in our neural networks. The way used in this work is by using embedding layers. These can represent m discrete values in n-dimensinal space, by storing an embedding matrix $E \in \mathbf{R}^{m \times n}$ values. When we need to convert a discrete value $t \in \mathcal{N}, 0 \leq t < m$ to the target dimension, we select the row E_{t*} from the embedding matrix and return it.

We can also use embeddings to encode information about positions in a se-

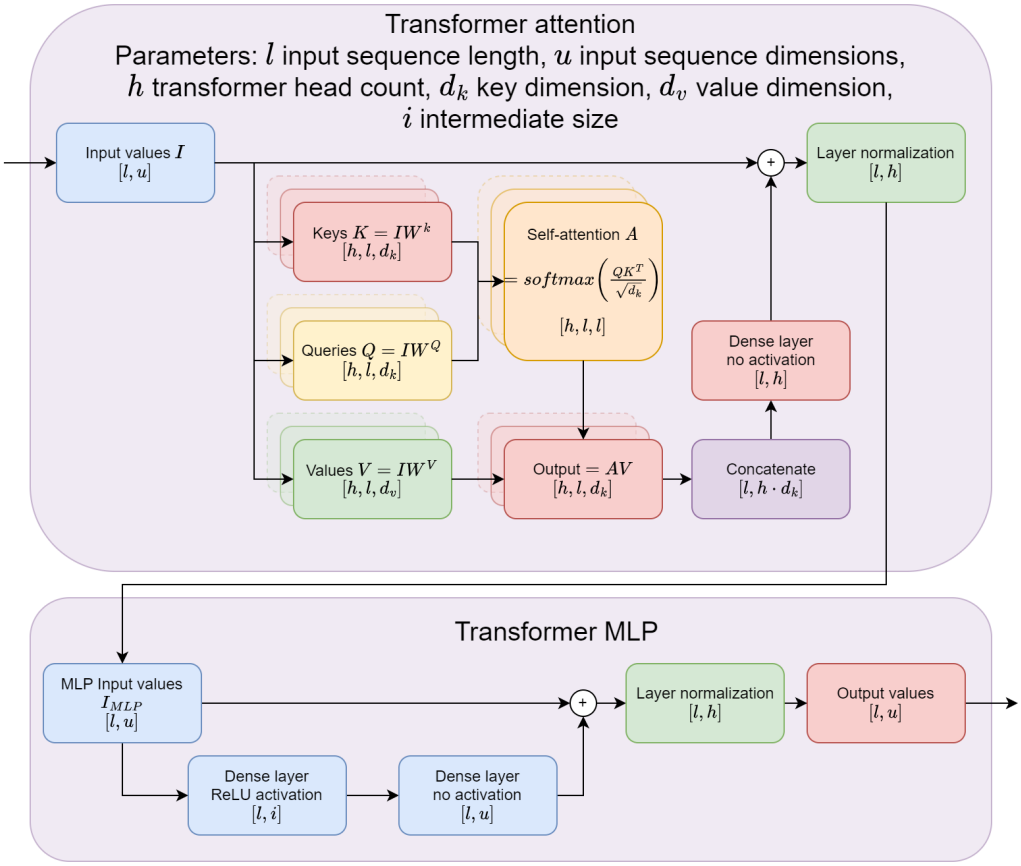

Figure 1.5: *A single transformer layer. The top block shows how the multi-head self-attention layer and the bottom shows the processing layer. Both of them combined constitute one transformer layer.*

quence or an image - by associating each index with a vector in the embedding matrix, the network can learn useful representations for each position. We often add this positional embedding to the input data before a transformer layer, so that we can distinguish between different sequence elements inside the transformer layer.

1.7 Diffusion models

Diffusion models (first mention [Ho et al., 2020], we use a more advanced version [Song et al., 2020]) are a type of generative neural networks - they learn the distribution of the data and can generate new samples. Before describing how generation works, we briefly touch on the underlying idea, a Gaussian process.

A Gaussian process consists of multiple steps, where each step adds a normally distributed noise to the data (we assume we work with images in this work). The amount of noise added in each step is described by a schedule, and over as the number of steps increases, the values in the image with noise added converge to a normal distribution. Now, if we had a way to remove noise from images, we could start with a noisy image nad denoise it multiple times to obtain a sharp copy, and denoising pure Gaussian noise would lead to a generated image, leading to a reverse Gaussian process. Diffusion networks put this idea into practice to

generate images, as described below.

Diffusion network can be viewed as a neural network that approximates the noise present in the image, which we can then use to estimate the noise for each step during the reverse Gaussian process. Instead of defining the noise to be added at individual steps, we create a parameter, $t, t \in [0, 1]$, and a diffusion schedule β, denoting the amount of noise in the image for each t (and it always holds that $\beta(0) = 1$, $\beta(1) = 0$). With this, we can create an image by estimating the reverse diffusion process by taking uniform steps over t, using the neural network to estimate the amount of noise, and subtracting it so that the noise/image ratio matches in the next step. Any amount of steps can be performed, with larger numbers leading to results of higher quality.

The most common diffusion network architectures are inspired by U-Net [Ronneberger et al., 2015], downscaling the image and upscaling it again. We also provide the network with the current noise magnitude, either directly or as a sinusoidal embedding. To train the network, we take a training image, select a random time from the decoding schedule β, mix it with pure Gaussian noise so the noise-to-signal ratio matches the schedule value, and minimize the difference between the noise prediction based on my image and the noise I introduced before. It is also possible to learn a conditional distribution of the data, for example, how to produce a sharp image from a blurred one, this is done by simply providing the conditional image as input to the model before predicting the noise.

2. Dataset

In this chapter, we describe how we prepare training data for all of the models we will eventually use for outpainting.

2.1 Dataset

For training the models, we use the data graciously provided by the Czech hydro-meteorological Institute (further referred to as CHMI) [hydro-meteorological institute, 2023a] and collected by the computer graphics group [CGG web] at Charles University. The data consists of 98 locations, each of which has a static webcam that saves an image of resolution 1600×1200 every 5 minutes. We use 94 of 98 locations available, excluding some with images of wrong dimensions or low-quality data. For each of them, images were being collected from March in the year 2021 to the time of writing, summing up to circa 200 thousand images per location, or approximately 19 million in total. We show some images from the dataset in Figure 2.1.

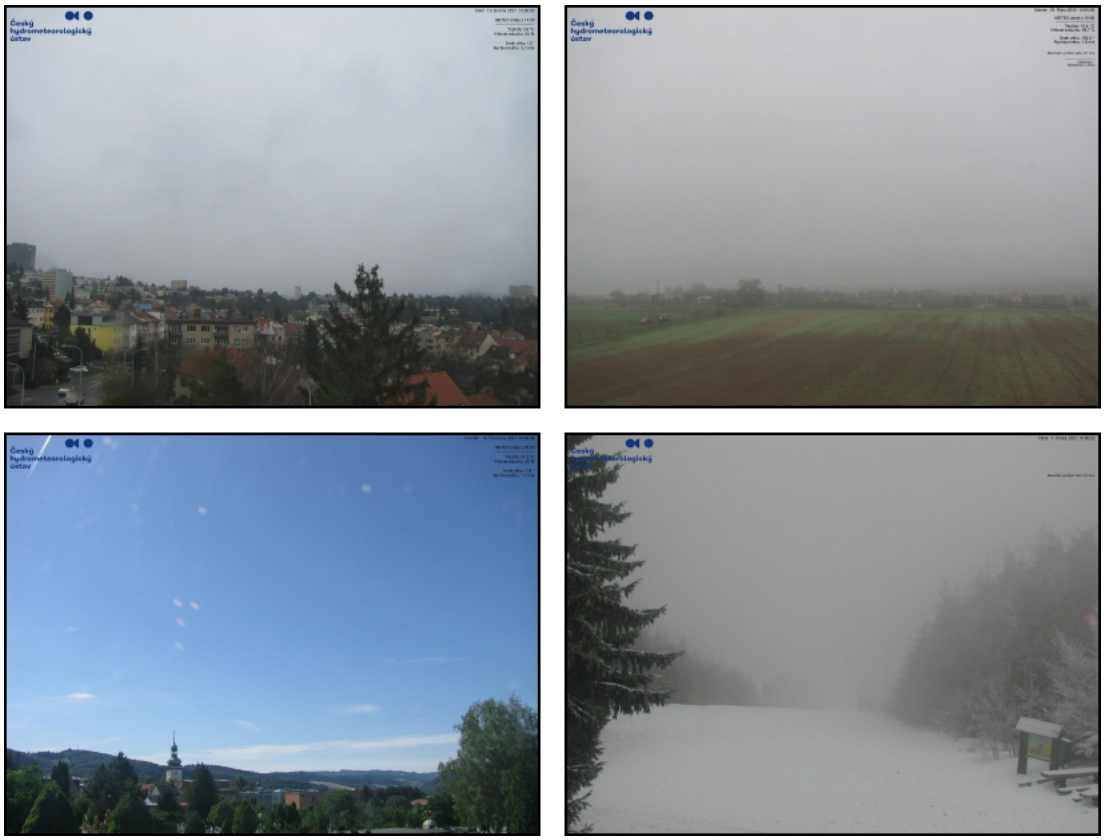

Figure 2.1: *Unprocessed dataset examples. All contain the landscape and the sky, and an overlay with the CHMI logo and weather text.*

All images contain some part of the landscape, the CHMI logo in the left top corner, and some weather measurements in the top right, in addition to the sky. For this reason, we start by performing a simple segmentation on each location, figuring out which pixels are part of the sky and which ones are not, assuming

that these do not change over the time of collecting the data. These masks are generated once before training, and their creation is described in more detail in the section below. When preparing the dataset for training a particular model, we downscale the images and masks and select a random part of the sky as model input, then filter the selected parts to condition the model to create more diverse images. In the following subsections, we first describe the segmentation algorithm in detail, then we describe how it is used to generate training data for each model.

2.2 Location segmentation

We base location segmentation on the following observation - during the day, the sky will change quite a lot, while the landscape stays mostly the same. The algorithm then boils down to detecting edges in an image, doing some processing, and then finding the places where edges are present many times during different times of the day. We describe the process of creating a segmentation mask, in which unmasked pixels are part of the sky, and masked ones are part of the landscape, the CHMI logo, or the weather information text.

I detect edges by computing the image gradient in the direction of the x and y axis and summing the absolute values of both directions and all RGB components. After that, we use a threshold, marking all values above it as parts of the landscape.

I then process the edge mask as follows - first, we perform the *morphological closing operation*[1] with a 3x3 square kernel five times in a row, making parts of the mask a lot more cohesive and filling in noisy areas. After that, we find all continuous non-landscape areas in the mask and mark all areas whose number of pixels is below a certain threshold (20000 pixels is used) as landscape. Then, we do the same for areas marked as landscape, discarding those with less than 300 pixels, considering them random noise.

After this, we hide the CHMI logo by masking a 270x115 rectangle in the top right and the weather measurements by masking a 250x250 rectangle in the top left. The process is shown in Figure 2.2.

During daytime with bright skies, this generally produces an acceptable mask, with just a few mistakes, such as masked parts of clouds and missing masks over patches of the landscape of the same color. To fix these issues, we compute separate masks for a hundred consecutive time steps from one day (starting at noon so most time steps are during the day). Then, for each pixel, we compute the largest consecutive amount of time steps it has been marked as an edge and the percentage of the total time it was an edge. For an edge to be in the final mask, the consecutive amount must be above a certain threshold, and the percentage must be above another threshold.

We didn't manage to find thresholds that would work well for all locations and weather conditions, so we instead chose the thresholds manually for all locations. The final segmentation masks are presented in Figure 2.3.

[1] A combination of the binary dilation and erosion operations. Erosion places the mask at all points in the image, and when there is a below the kernel that isn't true, it sets the result for the point as false. Dilation does the opposite, setting a pixel as true if any value below the kernel is true.

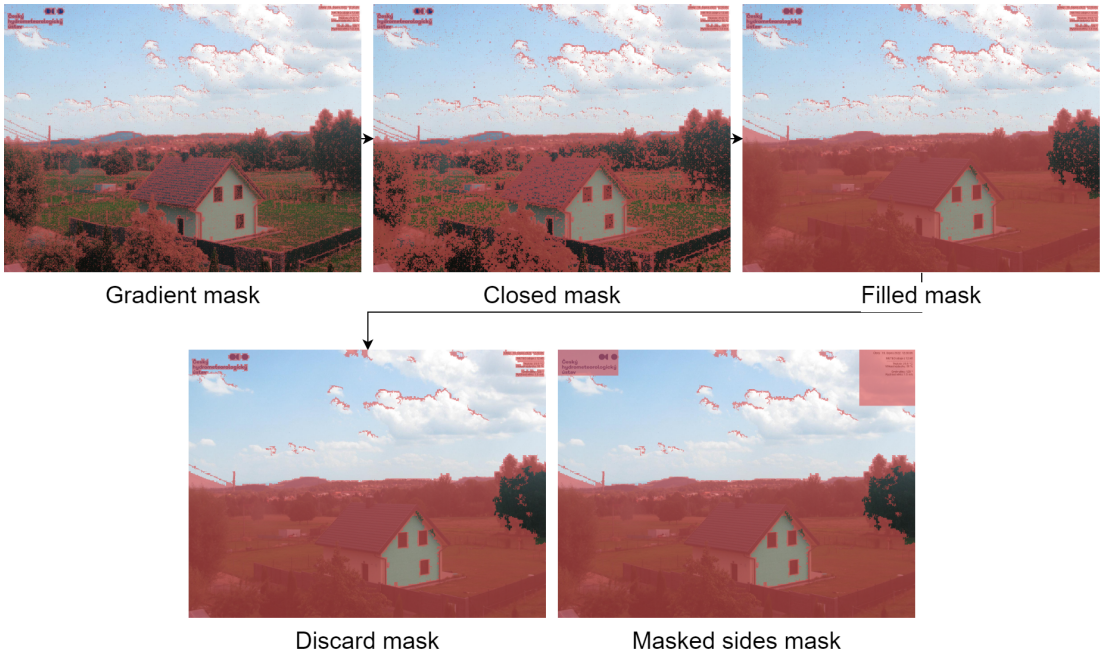

Figure 2.2: *Image segmentation on one image. Pixels colored red are parts of the mask. Gradient mask is obtained by masking all areas with a large enough gradient in the original image, closed mask by doing the closing operation, filled mask by filling small unmasked areas, discarded by discarding masked ones, masked sides by hiding the CHMI logo and the weather information.*

2.3 Generating training data

We now describe how to generate a training example, assuming we have a loaded image and its' segmentation mask, both of size 1600x1200. When mentioning color constants in the following paragraphs, we assume all images to be normalized with values between 0 and 1.

We generate training data by downscaling both the image and the mask by some factor based on the model being trained, and selecting a random subset that contains only sky pixels according to the provided mask. We believe this is better than simply scaling the image or using any part of it as it enables us to eliminate unwanted stretching artifacts while not having to worry about dealing with landscape pixels during training. It also conditions the networks to learn exclusively about the sky, which is what we want to generate during outpainting. The tokenizer and MaskGIT networks take inputs of size 128x128, so we found it useful to downscale the images by a factor of 4 before selecting the area, allowing the training to select a large part of the sky while still fitting multiple positions above the landscape. As the super sampler takes images of size 512x512, we don't downscale at all.

We also found it useful to filter the images created by the procedure above to condition networks to generate more interesting images. First, we discard all images that were recorded during the night - because the downscaled training data doesn't have a high enough resolution to capture any stars or planets in the night sky, all the images are pitch black, and there is nothing of interest to generate. We consider all images with a mean less than 0.2 to be those of the

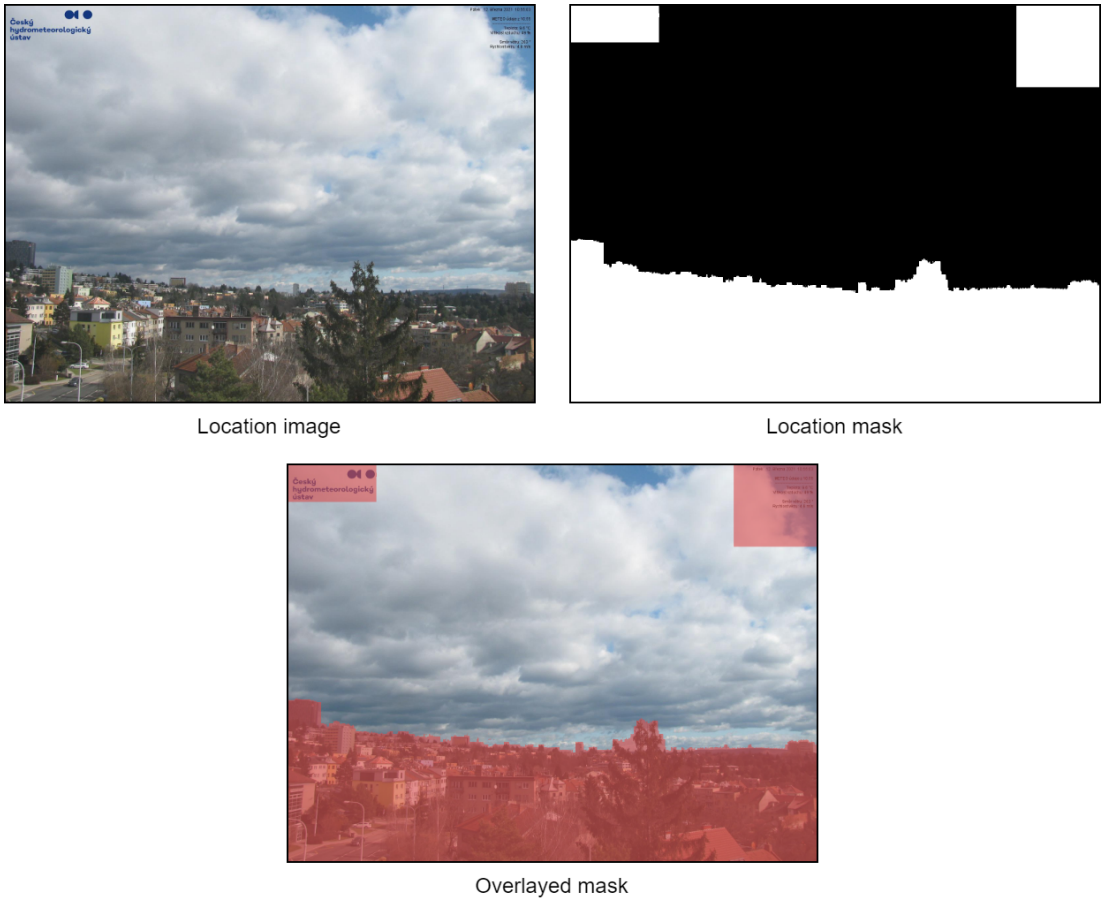

Figure 2.3: *Final segmentation. We show one sample image, the resulting mask, and the image with mask overlay for one location, Brno.*

night sky, and discard all of them, filtering out about 40% of all input images.

We also try to increase variability in the generated images by limiting the number of monochrome images present in the training data. We found that using the unrestricted dataset makes it highly likely that we will get a single-color image during outpainting as well. Because we want to drastically limit this behavior, but not eliminate generating monochrome images altogether, we choose to keep 10% of the monochrome images with a standard deviation from their mean of less than 0.05, and all others, filtering out another 50% of the dataset.

Although we eliminate quite a large chunk of the training data during this step, we find it worth the increased quality in generated samples, and no measurable decrease in performance, partly due to the abundance of data available. The example data from each part of the process is shown in Figure 2.4.

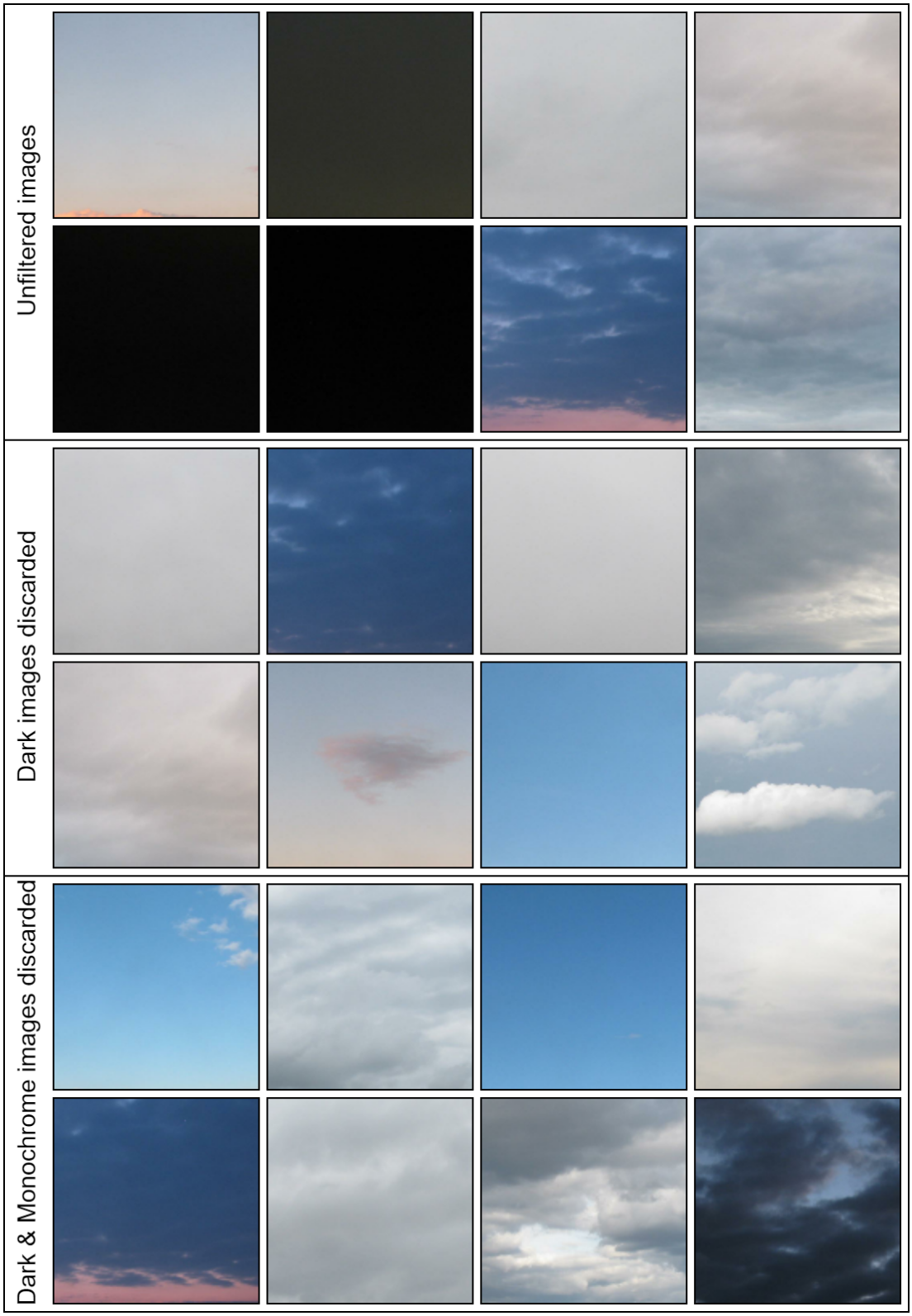

Figure 2.4: *Dataset filtering. The first row shows images from the unfiltered dataset, the second one from the dataset with night images filtered out, and the third row shows the final dataset, with both night and monochrome images discarded.*

3. Models

This chapter describes all models used, how they are trained, and how they are utilized in the outpainting algorithm.

Our outpainting algorithm consists of three distinct models - a VQVAE tokenizer, the MaskGIT transformer model, and a diffusion model super sampler. The tokenizer consists of two parts, an encoder that can convert images to a latent space consisting of discrete tokens, and a decoder, which attempts to reconstruct the original image given tokens. The MaskGIT model takes in an image converted to tokens using the tokenizer, with some tokens getting masked out, and tries to reconstruct the original tokens. The super sampler gets an image on input, upscales it, and tries to fill in the missing details. For clarity, all models are visualized in Figure 3.1.

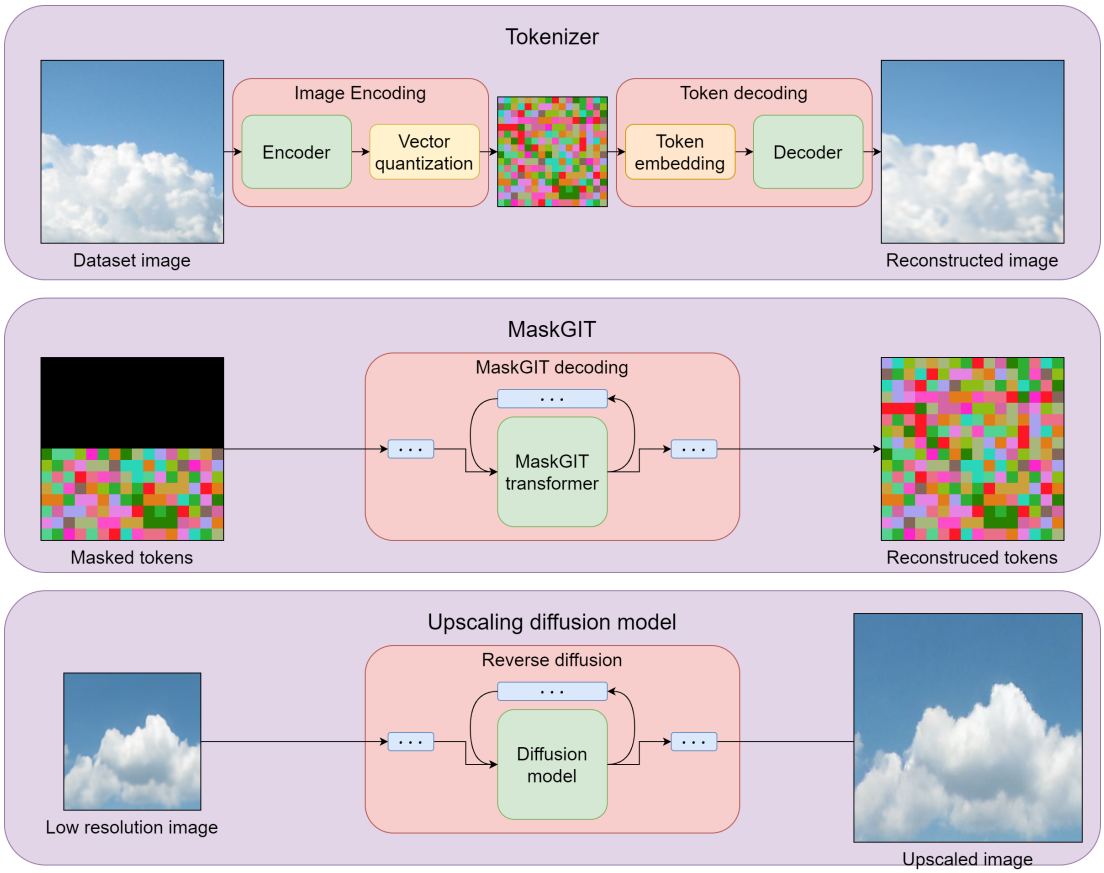

Figure 3.1: *All the models we use. On top, we have the tokenizer - it can convert images to discrete tokens, and tokens back to images. In the middle, we have MaskGIT - when provided with an image with some of its' tokens masked (marked as black in the image), it can replace the missing tokens with reasonable values. The bottom row shows the diffusion model. Provided with a low-resolution copy of the image, it can upscale it and add some details.*

During outpainting, we use the models as follows - first, we get the image we are trying to extend, pass it through the tokenizer encoder, receiving a set of tokens. After that, we repeatedly call the MaskGIT model, generating feasible tokens surrounding the original image, until a specified size is reached. When we

are done, we use the tokenizer decoder to convert the tokens back to an image, which is then upscaled using the super sampler to create the final result. The process is described in more detail later in the next chapter.

3.1 Tokenizer

The tokenizer is responsible for taking in images and converting them to a latent space consisting of discrete tokens. The architecture is based on the vector quantization variational autoencoder [van den Oord et al., 2017] and consists of three parts - an encoder, the vector quantization layer, and a decoder. The encoder takes images as input, processing them using multiple residual blocks and downscaling convolutions, and outputs a downscaled image of embedding vectors in an embedding space. The vector quantization layer then takes the vectors in embedding (or latent) space, and attempts to represent each of them using a discrete token, then it converts the tokens back to the embedding space. The decoder then uses more residual layers and transposed convolutions to attempt to reconstruct the original image from the vectors in the embedding space. Our variant of the VQVAE trains on images of size 128×128 and uses a latent space of dimensions 32×32. We describe the architecture and training process of all model parts in this section.

In the description of both the encoder and decoder, we use the variations on the residual block introduced in Figure 3.2.

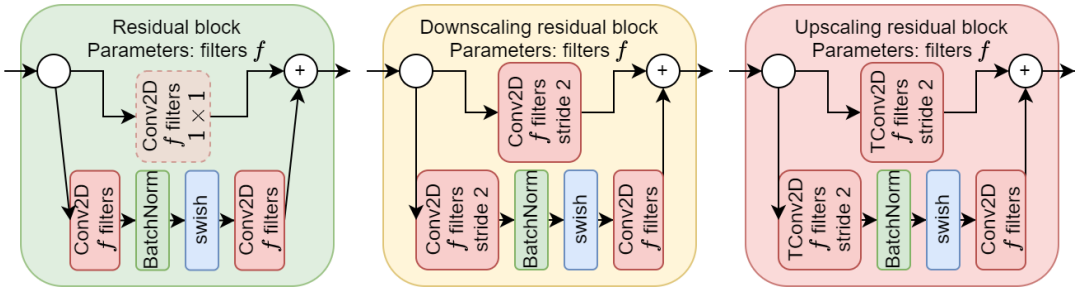

Figure 3.2: *Residual block configurations used in the tokenizer. All convolutions have* 3×3 *kernels and linear activations. The dashed block in default convolution is active only when the input image has a different number of filters than* f*, and uses a* 1×1 *kernel.*

We use the encoder architecture as shown in Figure 3.3:

The vector quantization layer can be viewed as an extension to the embedding layer with added support for converting vectors to discrete tokens. The inverse operation is implemented by simply taking the closest vector in the embedding matrix according to the L2 metric and using the associated token, while the conversion back is the same as in the original embedding layer. During the forward pass, we apply both operations in order - first the conversion to tokens, then embedding them back. To be able to train the network using backpropagation, we need to be able to compute the gradient of the loss relative to the layer inputs when provided with the gradient of the outputs. Thankfully, albeit our layer is not differentiable, as both the layer inputs and outputs share the same embedding space, VQVAE proposes to copy the gradient from output to input, propagating

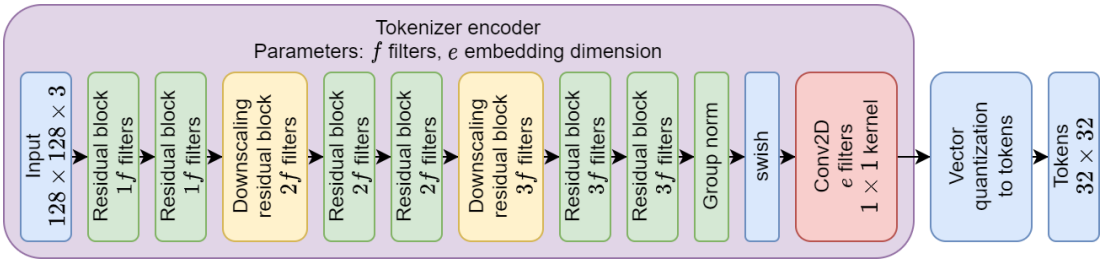

Figure 3.3: *Tokenizer encoder architecture. Using 2 downscaling blocks results in a 4-times downscale, and using the vector quantization layer on the output results in an output of* 32×32 *tokens*

useful information and enabling the model to be trained.

The decoder uses an architecture similar to the encoder, just using upscaling convolutions instead of downscaling ones, as presented in Figure 3.4.

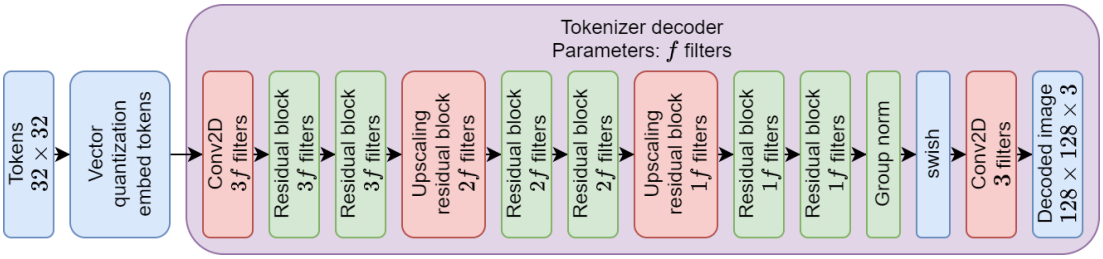

Figure 3.4: *Tokenizer decoder architecture. Again, two upscaling convolutions are used, upscaling the tokens four-fold while attempting to reconstruct the original colors.*

Using the terminology from VQVAE (x is the model input, $z_e(x)$ is the encoder output, $z_q(x)$ is the vector quantization layer output), we train the model using the following loss:

$$L = K \log p(x|z_q(x)) + \alpha||sg(z_e(x)) - e||_2^2 + \beta||z_e(x) - sg(e)||_2^2 + \gamma c_t||e_t - v_t||_2^2$$

The first three terms are from the VQVAE article, their rationale is as follows:

- $K \log p(x|z_q(x))$ - The reconstruction loss multiplied by a constant, K. When we interpret the model outputs as means of a normal distribution for each pixel, this term can be derived to be equal to the mean squared error between decoder outputs and encoder inputs, multiplied by a constant, K, which we set to the inverse of the variance in the input data, $K = 1/Var(x)$.

- $\alpha||sg(z_e(x)) - e||_2^2$ - The VQVAE embedding loss. Using the stop gradient sg operator, it changes the codebook vectors in the vector quantization layer to be closer to encoder outputs.

- $\beta||z_e(x) - sg(e)||_2^2$ - VQVAE commitment loss. Attempts to move encoder predictions closer to embedding vectors to increase training stability.

We found that the original VQVAE tended to leave some vectors unused, missing out on potential generation quality. For this reason, we introduce the following loss: $\gamma \sum_{t \in T} c_t ||e_t - v_t||_2^2$. Going over all VQVAE tokens T, c_t is 1 if the token was used for embedding the last batch, 0 otherwise. e_t is the closest encoder result vector from the last batch, and v_t is the token embedding in the codebook. The loss moves unused embedding vectors closer to encoder outputs in the last step, increasing their chance of being used the next time.

In the following sections, we use the following terminology: Using the tokenizer/encoder for conversion to tokens means applying the encoder and the first part of the VQ layer. Analogously, using the decoder to convert tokens to an image means embedding the tokens using VQ, and using the decoder to get an image.

3.2 MaskGIT

The masked generative transformer model [Chang et al., 2022] is responsible for doing outpainting on tokens - given an image with half or a quarter of its' tokens masked out, it can replace the hidden tokens with ones that feasibly complete the image. We base our model structure on the original MaskGIT article and only slightly modify the training process. We first describe the model architecture, how we use it when generating masked tokens, and finally, how we train it.

The MaskGIT architecture is greatly inspired by BERT [Devlin et al., 2018] from the area of NLP, just generalized for images. It takes an array of tokens as an input, some of which may be replaced with a special symbol, `MASK_TOKEN`, to signify that the model should try to predict the token in this place, and outputs, for each place, a set of logits, each one corresponding to a token from the tokenizer codebook.

The exact architecture used is presented below in Figure 3.5.

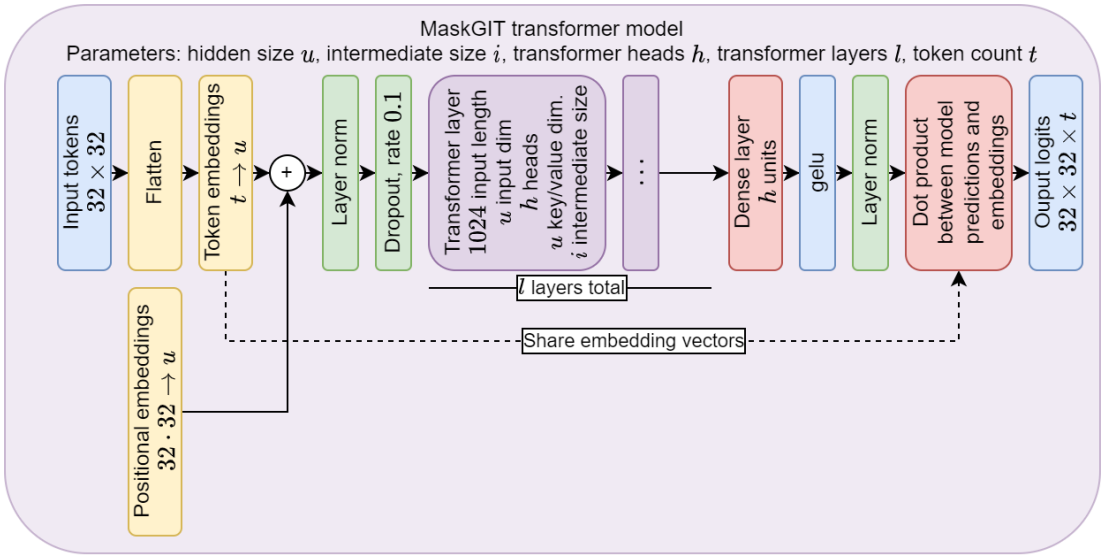

Figure 3.5: *MaskGIT architecture. The token embedding layer contains trainable embeddings for each token, while the positional embeddings contain one for each sequence position. The token embedding of a* `MASK_TOKEN` *is* 0.

We now continue by describing the way new tokens are generated, the process used is the same as in the original article, but we provide a brief summary. In theory, the model would be able to generate all tokens in a single pass, but we can get samples of much better quality if we allow multiple generation steps. We use the cosine mask scheduling function $\gamma(x) = cos(2 * pi * x)$ from the original article and set the `decode_steps` and `generation_temperature` parameters, specifying how many times we call the model when generating the image and how creative should the model be when guessing tokens respectively. Starting with an input token array where some tokens are equal to `MASK_TOKEN`, we run the following algorithm:

N = number of unknown tokens in input tokens
For $t = 1$ to `decode_steps`:

- Use the MaskGIT model to predict masked token logits. If we already know a token, we replace its logit with infinity.
- For each position, compute softmax over all logits to obtain token probabilities. Sample one token at random for every position - because we set the token logit to infinity, this will preserve the input tokens.
- Compute the number of tokens that should be masked after this step, $K = \gamma(\frac{t}{decode\_steps}) * N$
- We obtain confidence values by adding logits corresponding to the sampled tokens and random values from the Gumbel distribution with a scale equal to `generation_temperature`.
- We take the K lowest confidence values and replace their values in sampled tokens with the `MASK_TOKEN`. Then, we set the input tokens for the next step to sampled tokens.

When the cycle finishes, we are left with the newly created tokens.

During outpainting, we generally know a part of the columns or rows in the input token image and need to generate the tokens adjacent to them. As this is the only thing we will use the model for, we incorporate it into the training process, modifying the original MaskGIT. We define a `mask_ratio` constant, defining the percentage of rows/columns that will be masked. During outpainting, we either know the unmasked rows/columns, or we know both, and the only missing part is one corner. Based on this, we define training masks, 8 in total, describing all possible usages of the model, shown in Figure 3.6.

We modify the training process to use our training masks as follows:

- We sample a ratio $t \in [0, 1]$
- Using the cosine scheduling function from the section above, we obtain a `mask_ratio` $= \gamma(t)$
- We take an image and keep all tokens that are marked as unmasked in the training mask. We mask all other tokens with probability `mask_ratio`.
- We minimize the negative log-likelihood of the masked tokens.

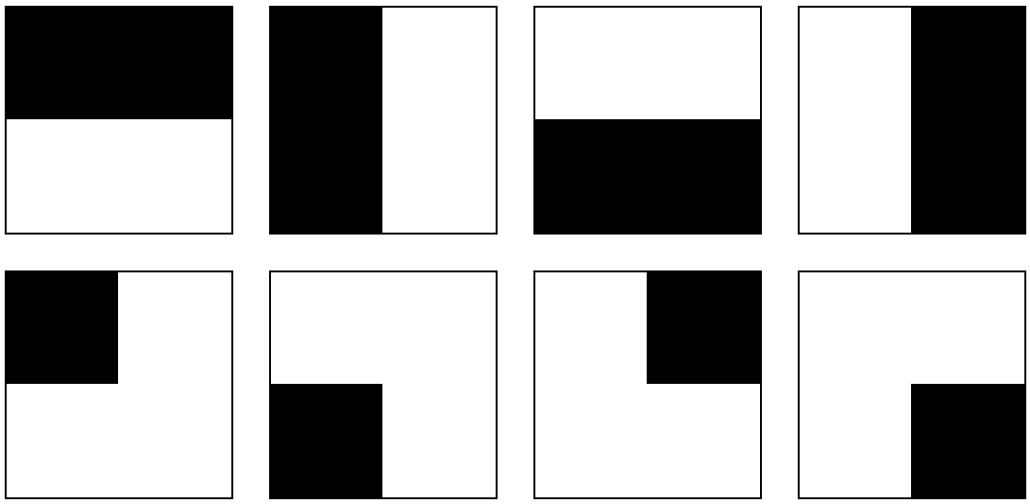

Figure 3.6: *MaskGIT training masks. White and black pixels represent known and masked tokens respectively.*

3.3 Diffusion model super sampler

Because the tokenizer tends to lose some details when decoding from tokens, we use an upscaling model loosely based on [Saharia et al., 2022] to add details to the final image and upscale it. For this, we use a diffusion model with a cosine generation schedule, where the signal we try to predict is edges - values that should be added to the blurry image to generate a sharp version. We describe the architecture of the model here, and how it is trained.

The model inputs are the blurred 512x512 images, the ratio of the noise component relative to the signal as a sinusoidal embedding, and the edges, with a noise component added. We train the model to predict the noise component in generated edges. The architecture we use is based on U-Net [Ronneberger et al., 2015], and it uses the same residual blocks as the tokenizer for encoding/decoding. At each resolution, we use residual blocks to process the data, then a downscaling block into more filters, until images of size 8x8. We then do the inverse, using upscaling convolutions into fewer filters, concatenate with the processed images at the same level created during downscaling, use a residual block on the whole stack, and upscale again, until we reach the target resolution of 512x512. We then predict the final noise in the image using a linear layer. To illustrate the architecture, we first present some building blocks used in Figure 3.7, then the architecture itself in Figure 3.8.

For the training procedure, we need to find the target edges. by taking a high-resolution image, downscaling it to the tokenizer input size, passing it through the tokenizer, and upscaling it again, obtaining a blurry image. We get the edges by subtracting the blurry image from the original, high-resolution one. With this, we can train the model like a generic diffusion - we mix the target edges with a random amount of noise according to a random time in our diffusion schedule, and the model attempts to approximate the noise.

Upscaling tokenizer outputs can be done simply - we first upscale the image using bicubic interpolation and use the diffusion model iterative decoding to es-

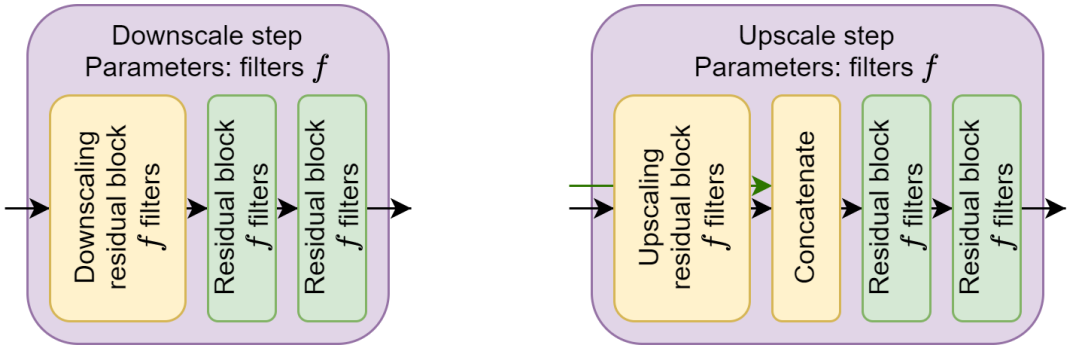

Figure 3.7: *Super sampler downscaling and upscaling blocks. The upscaling block uses two input parameters, denoted by arrows of different colors.*

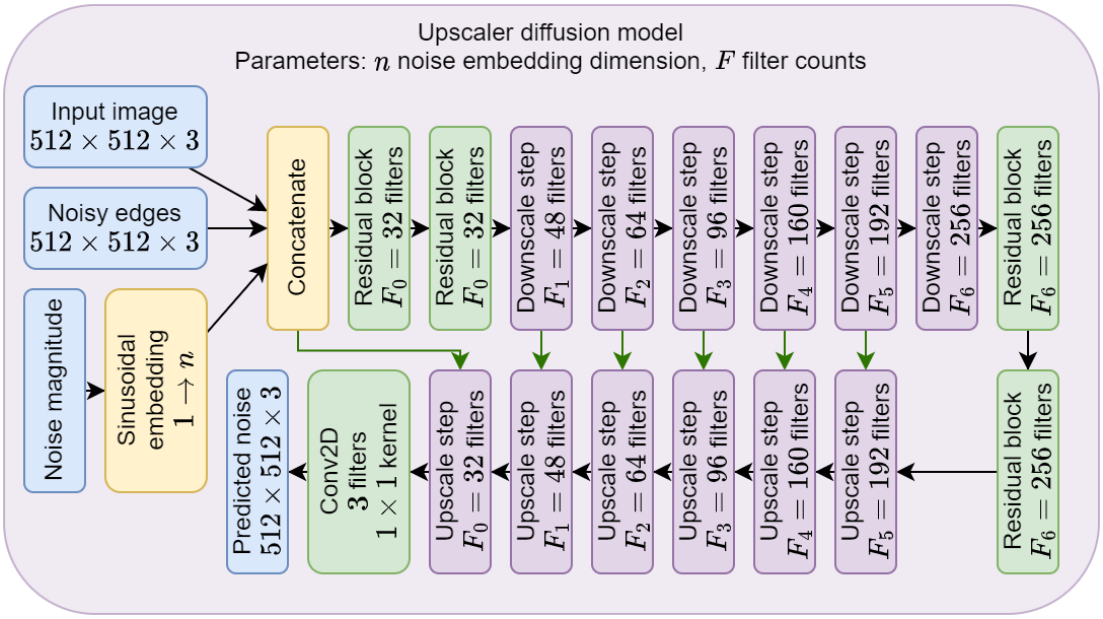

Figure 3.8: *Super sampler architecture. Input image, noisy edges, and noise variance are all used as model inputs, predicted noise is the only output. To make the illustration more compact, the model execution starts to the right, but continues to the left below, in the direction of the arrows. During processing, the model downscales the image 6 times, to a resolution of 8×8, then scales it back up. For clarity, we show the amounts of filters used in the final model, even though this is a modifiable parameter.*

timate the edges in it. After it is done, we add the edges to the blurry image, and clamp all values between 0 and 1, obtaining a sharp copy.

4. Outpainting

We now describe how to perform outpainting using the trained models. We assume we start with an image of the same size as the tokenizer input and want to expand it beyond its borders. We allow specifying the `outpaint_range` and `samples` parameters - the outpaint range defines how many outpainting steps the model should take, and the samples define the quality of decoding. We use the tokenizer to convert the input image to tokens, then we use the MaskGIT to perform outpainting on the created tokens, and finally, we use the super sampler to generate high-quality samples. We visualize the whole process in Figure 4.1, each step is described in more detail below.

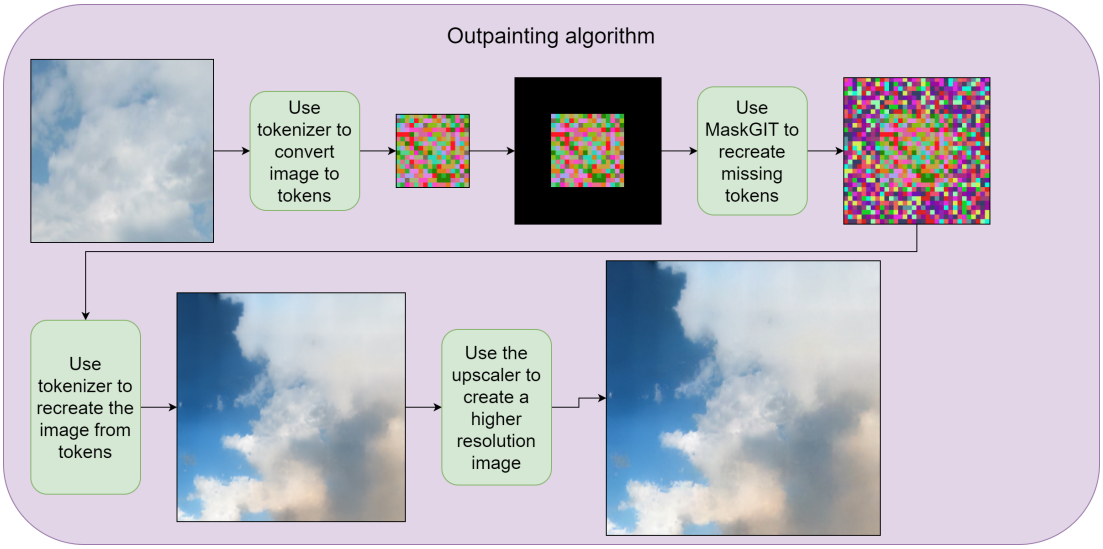

Figure 4.1: *An overview of the outpainting algorithm. MaskGIT, tokenizer and super sampler are called multiple times during the last three steps, to cover the whole outpainted image.*

After converting the input image we reserve space around it for the tokens we want to generate and set them all to `MASK_TOKEN`. After that, we move around the perimeter of known tokens in the order from Figure 4.2 and use MaskGIT to generate new tokens around it, until the whole space is filled.

When all tokens have been generated, we want to convert them back to an image. Because calling the tokenizer decoder directly on neighboring blocks of tokens can introduce artifacts (as we have no guarantee that the borders will be smooth), we instead use a sliding window, call the decoder with many overlapping squares of tokens, and average the results. The centers of the generating blocks are on a regular grid, and the *samples* parameter specifies how many more times we call the decoder, relative to neighboring blocks. This process produces multiple overlapping images - to compute the final image, we use a weighted averaging of the colors on a given position. Using weights equal to 1 everywhere can sometimes still produce visible edges in the final image in places where one block ends. To eliminate this, we use smaller weights for colors near the edge of their output image, while those at the center have large ones.

The previous step created an outpainted image, but due to averaging and data

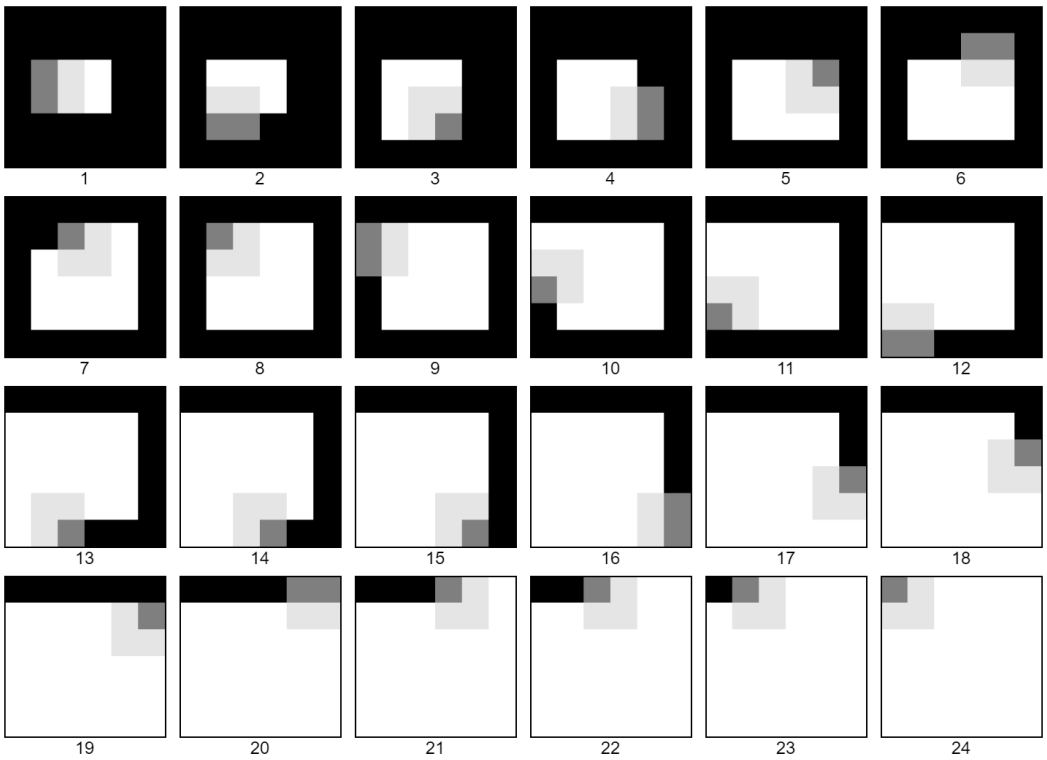

Figure 4.2: *Outpainting order. We start with known tokens in the middle (white) and unknown everywhere else (black). Gray tokens are the ones currently being outpainted, light gray are known tokens currently being fed into the MaskGIT to create new ones.*

loss when using the tokenizer, the output is often blurry. We fix this with the last network, the super sampler, which can add details to the outpainted image and upscale it. We once again work on a grid, this time, we only introduce small overlaps to avoid inconsistencies between neighboring parts of the image. Then we gradually go over all grid positions and call the super sampler for each one. If we already know some part of the result, because it was computed before, we alter the diffusion model generation slightly - during each decoding step, when we separate the edges into noise and signal data, and estimate the image in the next step using the two, we replace the estimated signal values with the values we already know. After every position is done, we are left with the upscaled image.

5. Results

We present the hyperparameters we used to train all three models, and the results achieved during training and when running the whole algorithm.

We show the parameters used to train all three models in Figure 5.1, then present the results for each model separately.

Tokenizer architecture parameters	
f, filters	32
e, token embedding dim	32
token count	128
Tokenizer training parameters	
K, reconstruction loss	10.93
α, embedding loss weight	1
β, commitment loss weight	0.25
γ, entropy loss weight	0.01
batch size	32
epochs	1.5
learning rate	10^{-4}
learning rate decay	exponential, 0.8 every 10^6
optimizer	Adam

MaskGIT architecture parameters	
u, hidden size	256
i, intermediate size	512
t, token count	128
h, transformer heads	4
l, transformer layers	8
mask ratio	0.5
MaskGIT training parameters	
batch size	32
epochs	2
learning rate	10^{-4}
learning rate decay	exponential, 0.8 every 10^6
optimizer	Adam

Upscaler architecture parameters	
n, noise dimension	32
$F_0 \dots F_6$, filter counts	$32, 48, 64, 96, 160, 192, 256$
Upscaler training parameters	
batch size	4
epochs	2, using a smaller dataset of $0.5M$ examples
learning rate	10^{-3}
learning rate decay	exponential, 0.5 every 10^6
weight decay	10^{-4}
optimizer	AdamW

Figure 5.1: *Model parameters. We stop training manually after the validation loss stops decreasing - causing fractional amount of epochs sometimes. The super sampler has been trained on a smaller number of images, but we didn't expect any significant improvement on the full one, so we didn't retrain it on the full dataset.*

5.1 Tokenizer result

The tokenizer gets an image as input, converts it to tokens and back, losing some information in the process, we are mainly interested in how much information was lost during the conversion. First, we present sample images and their reconstructions in Figure 5.2, then, the loss progression during training below in Figure 5.3.

5.2 MaskGIT results

We train the model to predict the masked tokens. The results we illustrate represent an image being converted to tokens using a tokenizer encoder, hiding some tokens by applying any MaskGIT training mask, using the MaskGIT to reconstruct the missing tokens, then using the tokenizer decoder to convert the result back to an image. We show both the results when decoding during one step, and when using `decode_steps`= 12 and `generation_temperature`= 1.0, in Figure 5.4 for border masks and in Figure 5.5 for corner masks.

During training, we track multiple metrics, most importantly validation accuracy on predicted tokens and validation loss, presented in Figure 5.6.

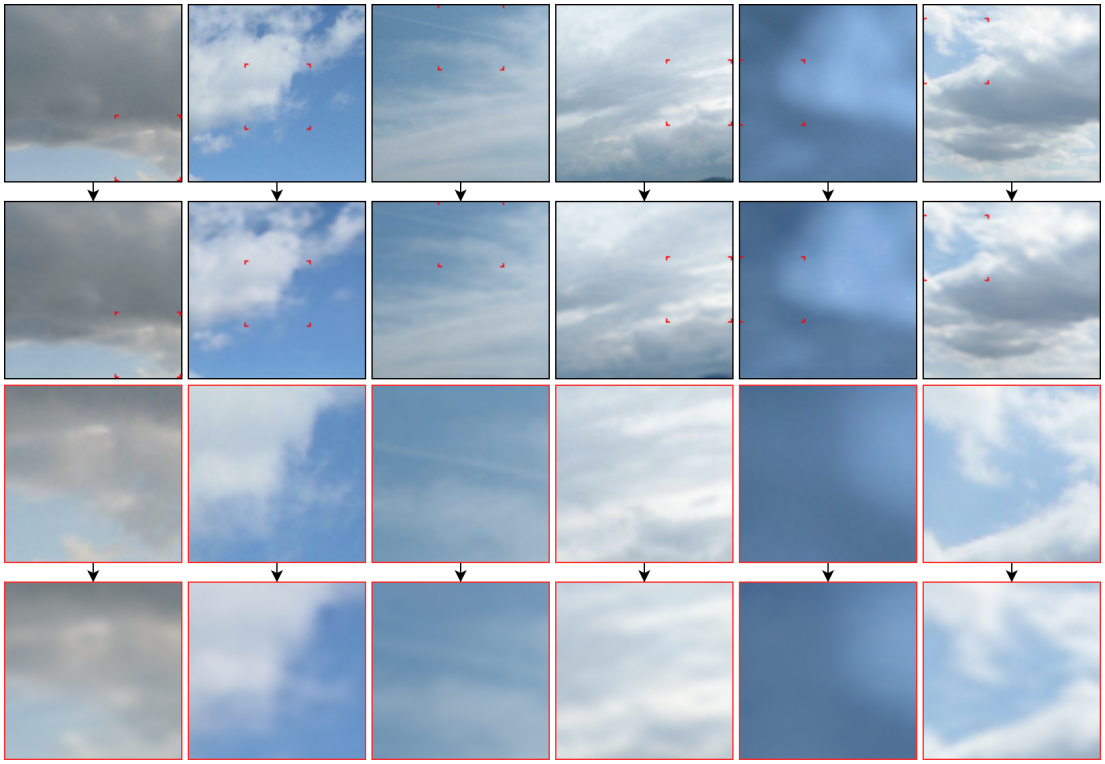

Figure 5.2: *Tokenizer results. The top row consists of dataset images, the second one contains reconstructed ones. Third and fourth row show the contents of the red boundaries in the original and reconstruction respectively, showing the loss of local details, images are significantly more blurry.*

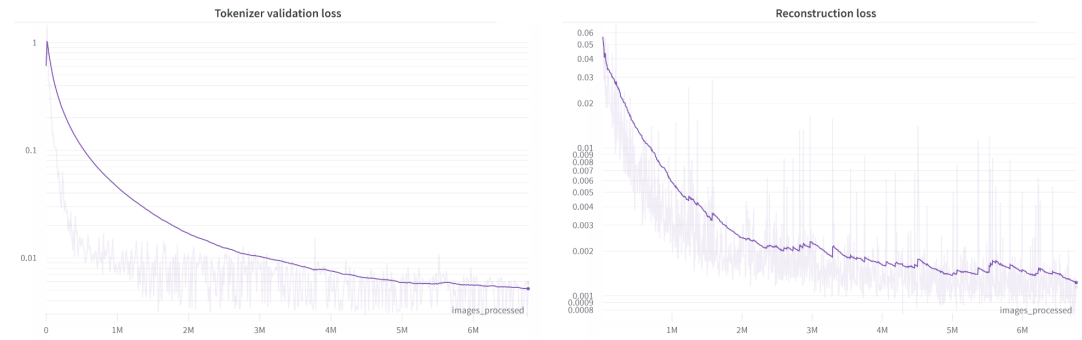

Figure 5.3: *Tokenizer losses during training. The solid line shows smoothed values using the exponential moving average technique with $k = 0.97$. The left graph shows total validation loss (sum of VQVAE reconstruction loss, embedding loss, commitment loss and entropy loss), the right shows reconstruction loss.*

5.3 Super sampler results

For the super sampler results, we present the original image, the low-resolution version, and how we manage to recreate the upscaled image, in Figure 5.7.

During training, the most important metric is the validation loss when predicting noise in an image - during training, it develops as shown in Figure 5.8.

Figure 5.4: *MaskGIT results, border masks. The first row shows images, just passed through the tokenizer. The second row shows which tokens (black) I discard before attempting to reconstruct them using MaskGIT. The third row shows how MaskGIT can reconstruct the missing parts in one step, and the last row shows how it can recreate them using 12 steps, with the generation temperature set to 1.0. The last row contains relatively few details, this is due to the fairly low generation temperature. We perform experiments with various values when analyzing the outpainting results.*

5.4 Outpainting results

We perform experiments with the parameters shown in Figure 4.5:

[!!!Outpainting experiments waiting for MaskGIT training to finish]

We show the original image, then two outpainted versions with the original image placed at the center - one without any upscaling, and one after applying the super sampler with sharpening.

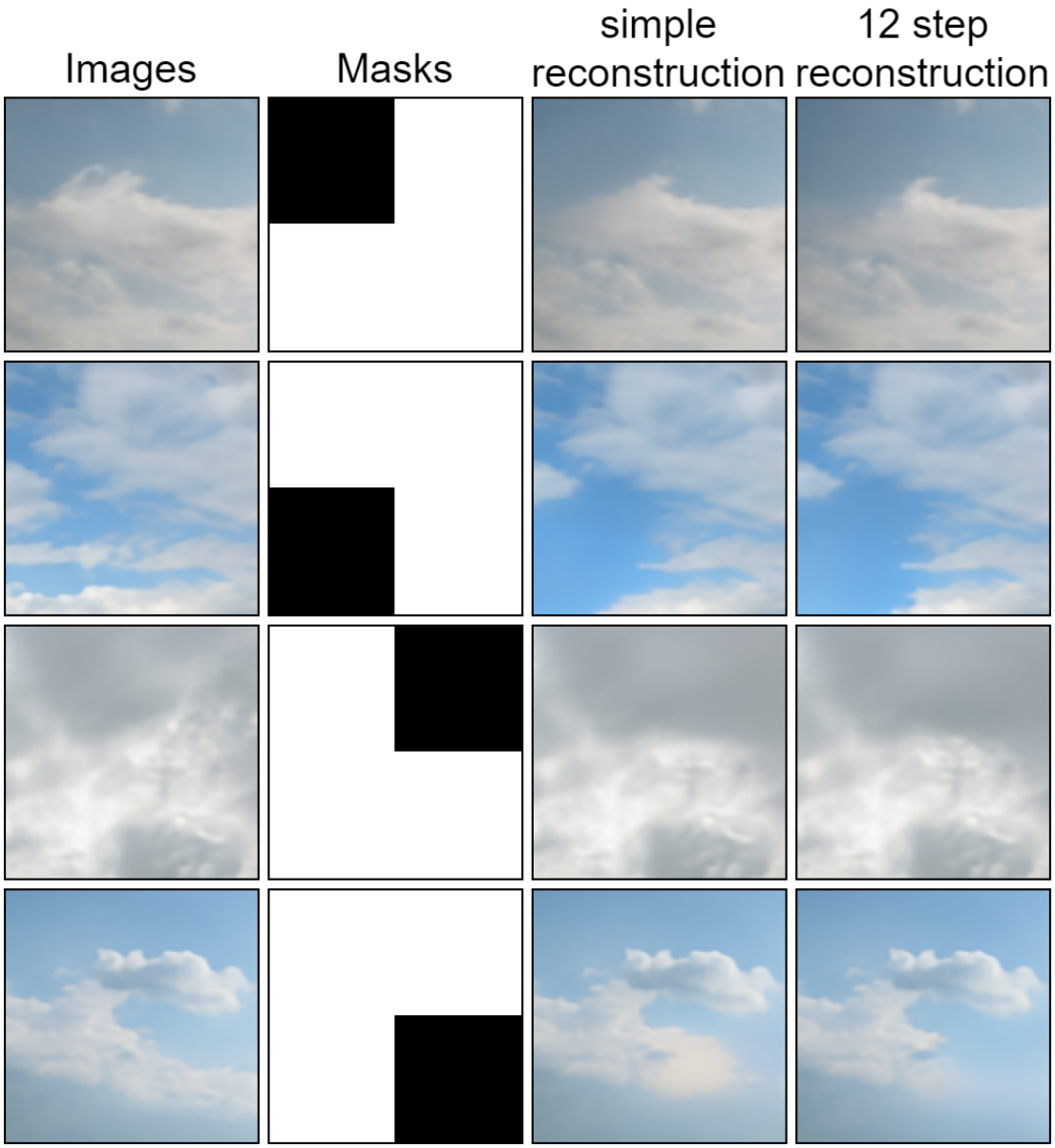

Figure 5.5: *MaskGIT results, corner masks. Rows have the same meaning as in Figure 5.4*

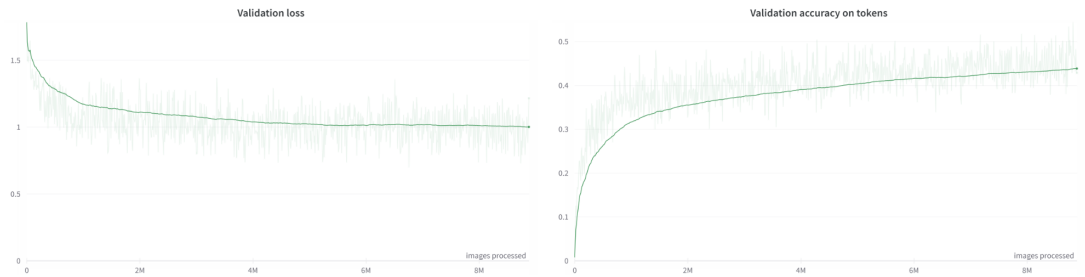

Figure 5.6: *MaskGIT losses. We track the validation accuracy when estimating all tokens discarded according to a training mask and validaton loss on the same set of tokens.*

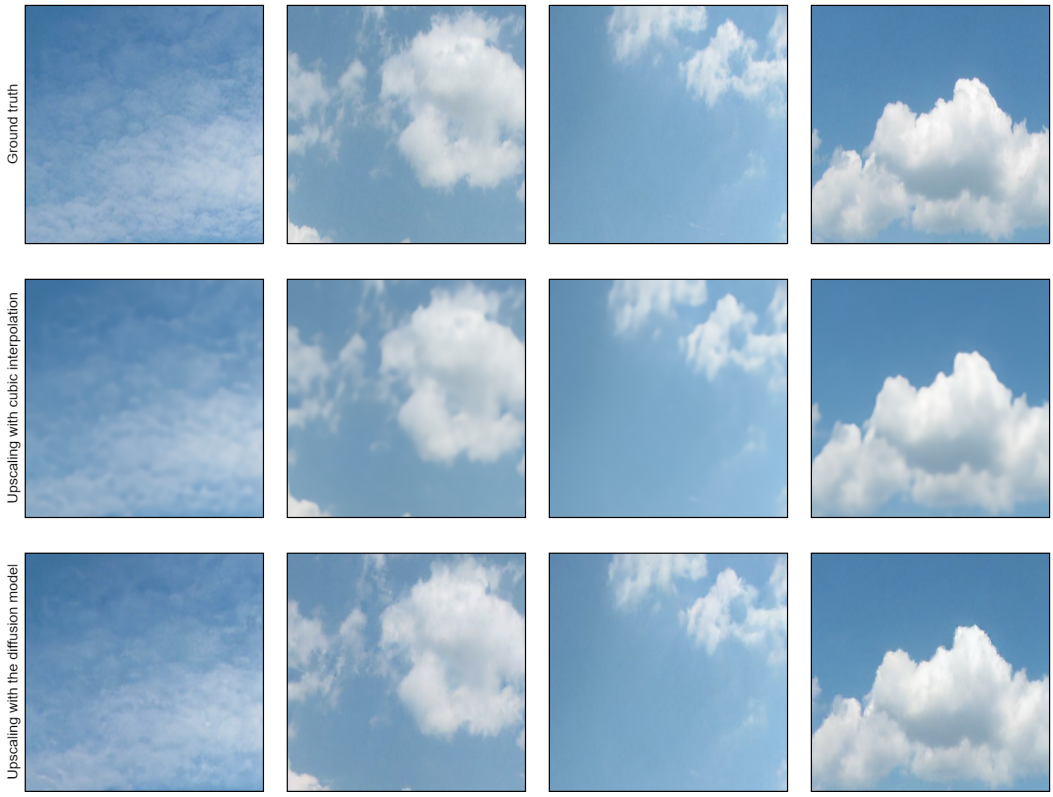

Figure 5.7: *Super sampler results. The first row is the original image. For the other two rows, we downscale the image, pass it through the tokenizer, and attempt to upscale it. The second row shows the upscaling done via bicubic interpolation, and the last one shows how the super sampler performs.*

[!!!Outpainting experiments waiting for MaskGIT training to finish]

Figure 5.8: *The figure shows the validation noise loss when training the super scaler. The model was trained earlier than the tokenizer and MaskGIT, and it uses a smaller subset of locations. Since the validation loss didn't seem to go down significantly at the end of the first epoch, we didn't retrain it on the full dataset.*

6. Code

We chose Python 3.10.9 as the main language for implementing this project, due to its popularity in the area of machine learning and the high quality frameworks available. We make the project available on GitHub, and use the following libraries:

- **Tensorflow 2.11.1** [Tensorflow] - The main machine learning library we use. For defining all models, and their training.
- **Tensorflow-probability 0.19.0** [Tensorflow-probability] - Probability utilities for tensorflow, we use it to sample from some distributions
- **Numpy 1.23.5** [Numpy] - Numpy, for several mathematical operations on arrays
- **Pillow 9.4.0** [Pillow] - Pillow, for loading images
- **Wandb 0.15.0** [Weights and biases] - Weights and biases, a library for logging training metrics, models, and more.
- **Opencv-python 4.7.0.72** [OpenCV 2 python] - An image processing library. We use it during image segmentation
- **Scikit-image 0.20.0** [Scikit image] - An image processing library. We use it during image segmentation
- **Paramiko 3.1.0** [Paramiko] - a library for SSH communication. We use it for fecthing location data from a server when creating segmentation masks.

We follow up with a short guide for installing the required libraries, then we describe how to run outpainting, and finally, how to train new models.

6.1 Installation

We provide a short guide that should fit all users below:

1. Install a compatible version of Python. 3.10.9 is recommended. We suggest using a virtual environment for the following steps - Anaconda is recommended, but any other environment manager should work.
2. Install Tensorflow 2.11. The process for installing the GPU version is quite arduous, since it requires installing CUDA and other libraries. Refer to the guide at the tensorflow website for more information.
3. Install all other python libraries. This can be done simply by using the command `pip install numpy tensorflow-probability Pillow wandb python-opencv scikit-image paramiko`. If any package-specific problems occur, specify the exact version.

Before running the project, we recommend downloading the models provided as `.zip` attachments, namely the MaskGIT and the super sampler. Extract the zip files and place them in the `models/maskgit` and `models/sharpen` folders respectively. It is also possible to train the models from scratch according to the guide provided in section 6.3.2.

6.2 Architecture

Before delving into how to run the code, we describe which parts of the project are present and what each of them does, briefly. We mark the most important python sources in red, supporting modules in green, sources ran independently in orange, and used directories in blue:

- `tokenizer.py` For training the VQVAE tokenizer
- `maskgit.py` For training the MaskGIT model
- `diffusion_model_upscale.py` For training the super sampler
- `outpainting.py` For running outpainting
- `build_network.py` Utilities to make building networks in tensorflow simpler
- `dataset.py` Image dataset loading, filtering, and segmentation according to known masks
- `log_and_save.py` Logging training metrics to wandb, useful callbacks
- `tf_utilities.py` Function for initializing tensorflow
- `utilities.py` Multiple functions for working with files, getting filenames and more
- `check_dataset.py` Go over all images in a dataset and notify the user about those with broken formatting
- `clean_up_wandb.py` Delete old model versions from weights and biases
- `segmentation.py` Create segmentation masks
- `./masks` Contains masks for all locations. During training, only the locations from here will be used.
- `./data` During the first run, we search for all training data matching `(dataset_location)/place/*/*.jpg`, where place has an available mask at `./masks/(place)_mask.png`. This creates two files, `./data/masks.npy.gz`, and `masks/filename_dataset.data`, which will be used in all subsequent runs
- `./models` All models will be loaded from or saved to this directory
- `(dataset_location)` Specified as a parameter - defines the location of all training images

6.3 Running scripts

We use weights and biases library [Weights and biases] for visualising the results. When running a script for the first time, you will be prompted to either run in anonymous mode, in which case the results of the run will be stored for a week, or to create an account to store them for longer.

Before we go through all the scripts to describe how to run them, we list some parameters shared between multiple scripts, for clarity. In all listings, we display parameters that require additional care as red.

- `--use_gpu` The GPU to use, if multiple are present in the system
- `--seed` The random seed
- `--threads` CPU threads tensorflow can use
- `--img_size` Size of input images. If any model is being loaded in this script, the size must match
- `--batch_size` The size of batch
- `--dataset_location` Where the data is found. The path to each image should match `dataset_location/*/*/*.jpg"`, corresponding to `dataset_location/(pla` If no value is specified, the value of the `IMAGE_OUTPAINTING_DATASET_LOCATION` environment variable is used. Since the data is assumed to be raw webcam data, for it to be used, a mask of matching name, `./masks/(place)_mask.png` must be present. During outpainting, there is an additional parameter to support using images in a folder, without masks, while still using this path.

6.3.1 Running outpainting

Outpainting is implemented in the `outpainting.py` file. It is assumed that there are already trained MaskGIT and super sampler models in the `models` directory. It takes in images as input, produces outpainted images, and logs them to weights and biases. It takes the following arguments (marked in red are those requiring some attention):

- `--attempt_count` How many examples to produce for one input image
- `--example_count` How many images to process, at most
- `--outpaint_range` How far out to outpaint
- `--generation_temp` Generation temperature
- `--samples` The quality of image decoding
- `--decoding full` or `simple` - the type of MaskGIT decoding to use, 1-step or with `decoding_steps`
- `--maskgit_steps`, =`decoding_steps` to use when outpainting tokens
- `--diffusion_steps` how many steps to use when upscaling

- `--generate_upsampled` whether to use the super sampler
- `--maskgit_run` the name of the MaskGIT model to use. It should exist at the path `./models/maskgit_(maskgit_run)`. The provided model will be loaded without specifying this parameter.
- `--sharpen_run` the name of the super sampler model to use. It should exist at the path `./models/sharpen_(sharpen_run)`. The provided model will be loaded without specifying this parameter.
- `--outpaint_step` What part of the image to fill when outpainting during one step. Should match the one specified in MaskGIT.
- `--dataset_outpaint_only` when set as true, the `--dataset_location` will be assumed to contain images directly, instead of training data which needs to be segmented first.

6.3.2 Running training

We present a short guide for training the models, but we do not provide the dataset used. Some recent webcam images are available publicly at the [hydrometeorological institute, 2023b], from where it should be possible to scrape the data and obtain a dataset of a reasonable size over several weeks. For most of the czech locations, masks are available in the GitHub repository. We assume that we will be working with these locations only; creating masks for other places will require some small extensions to our code. We present a brief summary below, then we deal with the actual training, assuming one has the data ready.

1. We discuss mask creation in this step, it can be skipped when using the masks for czech locations. The mask segmentation is performed using the `segmentation.py` file. As we ran the segmentation locally and most of our dataset was on a remote server, we first fetch images for new locations using SSH, and only then perform the segmentation. This is the only supported mode of operation out of the box. If you need to create masks locally, you can import only the `image_segmentation` method from the file and use it to compute the mask. All results should be checked manually before being used in training, as our algorithm can sometimes fail to find the correct mask when the weather conditions on a the day being analyzed are suboptimal. If the mask is only slightly wrong, you can use the `finalize_masks` method to tweak the final values for masks created in the previous steps. All masks and temporary data will be saved in the `masks` folder automatically.
2. We recommended running the `dataset.py` file to ensure data is loaded as planned and masks are working correctly - it will print how many images were loaded and log some dataset examples to weights and biases. When running for the first time, it will also go through `dataset_location` and match all files with their respective masks, and save the result, so loading is faster next time.
3. With the data prepared in previous steps, we can train a model. MaskGIT and super sampler require a trained tokenizer during training, the tokenizer

can be trained immediately. When loading a tokenizer model during training, it must be available in the models `./models/tokenizer_(tokenizer_run)`, where `tokenizer_run` should be specified as a parameter. Each of the training scripts support multiple different parameters that tweak the model architecture, losses, and more. To get all supported parameters for a given training script, you can either call `python (training_script_filename).py --help` from the command line, or view the source code directly.

4. We can see the progression of the training on the weights and biases website, as well as how the model performs with actual images. Models are saved automatically during training in the models folder.

Conclusion

[!!!Not done yet]

Bibliography

Anaconda. Anaconda website, 2023. URL https://www.anaconda.com. Accessed on 2023-05-04.

Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. Layer normalization. *arXiv preprint arXiv:1607.06450*, 2016.

CGG web. Computer graphics group at charles university, web page, 2023. URL https://cgg.mff.cuni.cz. Accessed on 2023-05-03.

Huiwen Chang, Han Zhang, Lu Jiang, Ce Liu, and William T Freeman. Maskgit: Masked generative image transformer. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 11315–11325, 2022.

DALL-E 2 web. Dall-e 2 web page, 2023. URL https://openai.com/product/dall-e-2. Accessed on 2023-05-03.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. *CoRR*, abs/1810.04805, 2018. URL http://arxiv.org/abs/1810.04805.

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015. URL http://arxiv.org/abs/1512.03385.

Jonathan Ho, Ajay Jain, and Pieter Abbeel. Denoising diffusion probabilistic models. *CoRR*, abs/2006.11239, 2020. URL https://arxiv.org/abs/2006.11239.

Czech hydro-meteorological institute. Czech hydro-meteorological institute, web page, 2023a. URL https://www.chmi.cz. Accessed on 2023-05-03.

Czech hydro-meteorological institute. Czech hydro-meteorological institute, available cameras, 2023b. URL https://www.chmi.cz/files/portal/docs/meteo/kam/. Accessed on 2023-05-04.

Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015. URL http://arxiv.org/abs/1502.03167.

Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems*, volume 25. Curran Associates, Inc., 2012. URL https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

Hao Li, Zheng Xu, Gavin Taylor, and Tom Goldstein. Visualizing the loss landscape of neural nets. *CoRR*, abs/1712.09913, 2017. URL http://arxiv.org/abs/1712.09913.

Midjourney web. Midjourney web page, 2023. URL https://www.midjourney.com/home/. Accessed on 2023-05-03.

Numpy. Numpy website, 2023. URL https://numpy.org. Accessed on 2023-05-03.

OpenCV 2 python. Opencv 2 python website, 2023. URL https://pypi.org/project/opencv-python/. Accessed on 2023-05-03.

Paramiko. Paramiko website, 2023. URL https://www.paramiko.org. Accessed on 2023-05-04.

Pillow. Pillow documentation, 2023. URL https://pillow.readthedocs.io/en/stable/. Accessed on 2023-05-03.

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

Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. *CoRR*, abs/1505.04597, 2015. URL http://arxiv.org/abs/1505.04597.

Chitwan Saharia, Jonathan Ho, William Chan, Tim Salimans, David J Fleet, and Mohammad Norouzi. Image super-resolution via iterative refinement. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022.

Scikit image. Scikit image website, 2023. URL https://scikit-image.org. Accessed on 2023-05-03.

Jiaming Song, Chenlin Meng, and Stefano Ermon. Denoising diffusion implicit models. *CoRR*, abs/2010.02502, 2020. URL https://arxiv.org/abs/2010.02502.

Stable diffusion web. Stable diffusion web page, 2023. URL https://stability.ai/stable-diffusion. Accessed on 2023-05-03.

Tensorflow. Tensorflow website, 2023. URL https://www.tensorflow.org. Accessed on 2023-05-03.

Tensorflow-probability. Tensorflow website, 2023. URL https://www.tensorflow.org/probability. Accessed on 2023-05-03.

Aäron van den Oord, Oriol Vinyals, and Koray Kavukcuoglu. Neural discrete representation learning. *CoRR*, abs/1711.00937, 2017. URL http://arxiv.org/abs/1711.00937.

Weights and biases. Weights and biases website, 2023. URL https://wandb.ai/site. Accessed on 2023-05-03.

Yuxin Wu and Kaiming He. Group normalization. *CoRR*, abs/1803.08494, 2018. URL http://arxiv.org/abs/1803.08494.

List of Figures

A. Attachments

A.1 First Attachment