Technische Universität München

Bachelor's Thesis

INSTITUTE FOR HUMAN-MACHINE COMMUNICATION
TECHNISCHE UNIVERSITÄT MÜNCHEN
Univ.-Prof. Dr.-Ing. habil. G. Rigoll

Bridging the Domain Gap in Tracking by Diffusion

Simon Okutan

Advisor: Philipp Wolters M.Sc., Fabian Herzog Ph.D.

Started on: 08.04.2024
Handed in on: 08.08.2024

Abstract

This thesis addresses the growing need for realistic synthetic data in computer vision, particularly for object detection and tracking tasks, where real data is expensive and/or difficult to obtain. With the upcoming possibilities of advanced latent diffusion models, such as the open model Stable Diffusion by StabilityAI [EKB+24], new opportunities have emerged to enhance synthetic datasets, making them more realistic while retaining advantages like easy semantic labeling of synthetic data generation.

The main contribution of this work is the development of an easy-to-extend pipeline leveraging the "Adversarial Supervision Makes Layout-to-Image Diffusion" (ALDM) [LKZK24] to transform existing synthetic images via its semantic segmentation labels into realistic and high-variance datasets. For postprocessing, this pipeline uses image-to-image processing guided by semantic segmentation labels and depth ground truth to generate images in varied weather, lighting, and scene conditions [RBL+22] [ZRA23]. The pipeline's effectiveness was evaluated through different experiments, particularly focusing on video analysis tasks like object detection and tracking.

Results indicate that while ALDM and its post-processing techniques are promising, challenges such as object distortion and noise amplification persist, limiting the overall accuracy of the generated data. Nevertheless, ALDM demonstrates robust spatial-temporal consistency on fixed camera scenes, particularly in maintaining background stability. These findings highlight the potential of diffusion-based synthetic data generation, underlining the need for further research focussing on image output with more training and integrating segmentation and depth information to enhance the resulting dataset quality.

`github.com/Nomiez/bridging-the-domain-gap-by-diffusion`

Contents

1 Introduction

1.1 Motivation

With the recent release of StabilityAI's Open Model Stable Diffusion 3 [EKB+24], latent diffusion models have pushed the boundaries of creating more realistic images even further. However, training these general-purpose models is extremely expensive, and they may reach a plateau where further enhancements require exponentially more data, as noted by [UPG+24]. Therefore, the focus should shift towards optimizing the use of existing models for specialized tasks and exploring new scientific fields where these models can be as effective as they are in other domains.

One example is the field of computer vision, which by itself has revolutionized numerous other fields, from autonomous driving and healthcare to surveillance and entertainment. Developing robust and accurate computer vision models often depends on the availability of high-quality, diverse datasets. Real-world data collection can be extremely expensive, time-consuming, and sometimes infeasible due to privacy concerns or safety issues. Consequently, synthetic data has become an important resource for bridging this gap.

Synthetic data offers high flexibility, enabling researchers to generate vast quantities of labeled images under controlled conditions. This controlled environment captures necessary variations, such as different lighting conditions, weather scenarios, and camera perspectives, which are often challenging to achieve with real-world data collection. Despite these advantages, synthetic data can lack the realism required to effectively train and evaluate high-performance models [ZBSL17].

Diffusion models could solve this problem, offering the capability to enhance synthetic image data, making it a more realistic representation of reality while retaining the advantage of complete control over the test environment. The resulting datasets enable training on rare events, improving the robustness of detection and tracking systems and allowing training in scenarios where real data is unavailable.

1.2 Problem Description

Synthetic data is essential for many computer vision tasks where real data is unavailable or limited, such as object detection and tracking. This thesis aims to utilize pre-trained neural networks and extensions based on Stable Diffusion [RBL$^+$22] to create a pipeline that transforms existing (synthetic) data from Synthehicle [HCT$^+$23] and the MUAD Dataset [FYB$^+$22] into high-variance realistic data. Synthehicle and other synthetic datasets provide semantic, instance, and depth ground truth, which can be used as image conditions in diffusion networks. In this work, the recently proposed ALDM [LKZK24] network with other post-processing operations will be used to produce images that are:

- realistic,
- high-variance (in terms of weather, lighting, and overall scene conditions) and
- consistent with the ground truth labels.

1.3 Contribution

The main contributions of this thesis are:

- Implementation of a diffusion pipeline based on ALDM [LKZK24] that transforms existing synthetic and real images as described above, and other enhancement methods, primarily using image-to-image technology guided by different ControlNets.
- Conducting a comprehensive analysis of the influence on performance when the generated data is used as training data for video analysis tasks focusing on object detection.
- Conduction tests around spatial-temporal image consistency, especially focusing on object tracking.
- Discussion of potentials and limitations of diffusion-based synthetic data generation and how certain emerging problems could be resolved.

1.4 Structure

1.4.1 Background

This chapter covers the foundational concepts and technologies relevant to the thesis. It includes detailed explanations of various neural network architectures, such as Artificial Neural Networks (ANNs), Convolutional Neural Networks (CNNs), Recurrent Neural Networks (RNNs), Transformer Models, Generative Adversarial Networks (GANs), and Latent Diffusion Models. It also discusses encoder-decoder structures and the ControlNet.

1.4.2 Related Work

This chapter reviews existing literature and previous work related to the thesis topic. It discusses various projects and models, such as Synthehicle [HCT+23], Adversarial Supervision Makes Layout-to-Image Diffusion Models Thrive (ALDM) [LKZK24], the paper "Applications of generative AI for sim-to-real data synthesis in driving" [ZWBR+24], the CARLA simulator [DRC+17], and Yolo [RDGF16] with its predecessor YoloX [GLW+21].

1.4.3 Own Work

This chapter details the research and contributions of this paper. It includes the project structure and functionality of the developed Stable Diffusion Pipeline. It also provides an in-depth look at the different modules created and describes the testing setups for object detection on the resulting images, as well as the consistency of resulting videos and object tracking.

1.4.4 Evaluation

This chapter presents the evaluation of the research work. It includes an assessment of ALDM [LKZK24] and image-to-image transformations regarding object detection and object tracking. The chapter also evaluates spatial-temporal image consistency and discusses diffusion-based synthetic data findings, potentials, and limitations.

1.4.5 Conclusion and Outlook

This final chapter summarizes the key findings and contributions of the thesis. It also suggests potential areas for future research, building on the results and insights gained from the current work.

2 Background

2.1 Artificial Neural Networks

An artificial neural network (ANN) [Ras16] [GS24] [ZLLS23] describes the architecture of neurons and their connection, which is orientated towards biological neurons to simulate a learning process.

ANNs consist of multiple layers, each of them containing a large number of neurons. The input layer consists of a node for each part of its input (e.g., a greyscale picture with the size of 400 x 400 pixels could contain 160,000 input nodes, one node for each pixel with an input between 0 and 255). The output layer is the last layer where the number of neurons maps directly to the desired output (e.g., if you want to decide if the animal in the picture is a dog or a cat, you could use one final neuron - if it activated, the ANN detected a cat, otherwise a dog). All layers in between are called hidden layers because they are not influenced by the outside. Each neuron has an activation function, determining if a neuron is activated by its input. Every neuron in layer k can get its input from other neurons' output from the layer $k - 1 = j$. An edge between two neurons has a weight w^{jk} assigned, influencing the sum of all the inputs from this edge. In mathematical terms, an artificial neuron is described by a function, also called an activation function. Over a layer's potentials and weights of the edges, a weighted sum is formed and projected via the activation function into a smaller space. Two often used functions are the Sigmoid function which projects all values into the interval $(0, 1)$

$$\phi(\mathbf{v}) = (1 + \exp(-\mathbf{v}))^{-1} \tag{2.1}$$

and the ReLU function, projecting to $[0, \infty)$:

$$\phi(\mathbf{v}) = \max(0, \mathbf{v}) \tag{2.2}$$

. Also, a bias (β) could be added to shift neurons activation potential.

To calculate the output of a neural network, a neuron's activation potential could be written down as a vector, multiplied by a matrix containing every edge that leads to the nodes in layer k. One row in the matrix corresponds to the edges between a neuron in layer j and a particular neuron in layer k. After that, a bias is added, and everything is wrapped into an activation function again, leading to the next node's potential [GS24] [ZLLS23]:

$$\boldsymbol{H}^k = \phi\left(\begin{bmatrix} w^{jk}_{0,0} & w^{jk}_{0,1} & \cdots & w^{jk}_{0,m} \\ w^{jk}_{1,0} & w^{jk}_{1,1} & \cdots & w^{jk}_{1,m} \\ \vdots & \vdots & \ddots & \vdots \\ w^{jk}_{n,0} & w^{jk}_{n,1} & \cdots & w^{jk}_{n,m} \end{bmatrix} \cdot \begin{bmatrix} a^j_0 \\ a^j_1 \\ \vdots \\ a^j_2 \end{bmatrix} + \begin{bmatrix} \beta^j_0 \\ \beta^j_1 \\ \vdots \\ \beta^j_2 \end{bmatrix}\right) \rightarrow \boldsymbol{H}^k = \phi(\boldsymbol{W}^{jk}\boldsymbol{A}^j + \boldsymbol{\beta}^j) \tag{2.3}$$

To make ANNs effective in predicting outputs, they have to be trained. "Backpropagation" describes the process of training the model to minimize the total cost by each output (or batch). For every result, the cost (also called "loss" or "error") in the output layer c_n is calculated by subtracting the actual output a_n from node n by the expected output e_n. Squaring the result for an individual node and taking the sum over all nodes results in the cost of one run. For all nodes, the cost is:

$$\boldsymbol{c} = \begin{bmatrix} (e_1 - a_1)^2 \\ (e_2 - a_2)^2 \\ \vdots \\ (e_n - a_n)^2 \end{bmatrix} \tag{2.4}$$

The average cost of multiple runs describes the performance of a neural network. Finding the global minimum of the cost function could be an option to minimize the cost. The problem with this approach is that the calculation cost is too high, making even calculations for small networks inefficient. Therefore, "gradient descent" is used, which approaches a local minimum iteratively. For backpropagation, the sensitivity of the cost function for a neuron with respect to every connected weight (and biases) is interesting because it represents the slope of the error function. The question is, how much do the weights of the edges have to be changed to correct the output. Mathematically speaking [ZLLS23], this is denoted by:

$$\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{W}^{jk}} \tag{2.5}$$

Looking at the network error function with the idea in mind that only the errors from the nodes that the neuron is connected to is interesting, the formula can be reformulated to:

$$\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{W}^{jk}} = \frac{\partial}{\partial \boldsymbol{W}^{jk}} + \boldsymbol{c}^k \tag{2.6}$$

As mentioned above the indices j and k reference the different layers, where $\boldsymbol{W}^{jk}$ is a matrix containing the weights connecting layer j and k. In the calculation, it is also assumed that k represents the final layer. Looking at $\boldsymbol{c}^k$, it is known that $\boldsymbol{e}^k$, the expected output, is not dependent on the function. With the chain rule and building the resulting derivatives, the following term emerges:

$$\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{W}^{jk}} = \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{a}^k} \cdot \frac{\partial \boldsymbol{a}^k}{\partial \boldsymbol{W}^{jk}} = -2\boldsymbol{c}^k \cdot \frac{\partial \boldsymbol{a}^k}{\partial \boldsymbol{W}^{jk}} \tag{2.7}$$

The second part of the equation heavily depends on the activation function because a^k is the activation of the node in layer k, in this case, in the final layer, and is calculated by $\phi(\sum_j \boldsymbol{W}^{jk} \cdot \boldsymbol{a}^j)$. Calculating the derivative of the last term with the chain rule results in:

$$\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{W}^{jk}} = -2\boldsymbol{c}^k \cdot \phi'\left(\sum_j \boldsymbol{W}^{jk} \cdot \boldsymbol{a}^j\right) \cdot \frac{\partial}{\partial \boldsymbol{W}^{jk}} \boldsymbol{a}^k = -2\boldsymbol{c}^k \cdot \phi'\left(\sum_j \boldsymbol{W}^{jk} \cdot \boldsymbol{a}^j\right) \cdot \boldsymbol{a}^j \tag{2.8}$$

This is the formula to calculate the change for the weights connecting the output. With the small modification of removing the factor 2 (when adding the direction to the weight later, it is scaled via a new factor also called "learning rate") and looking at $\boldsymbol{c}^k$ as the error of layer k (which is calculated by taking multiplying the transformed nodes from any layer k with the weights from jk), with j and k being an arbitrary layer, this results in the final function for updating the weights:

$$\frac{\partial \boldsymbol{c}}{\partial \boldsymbol{W}^{jk}} = -\boldsymbol{c}^k \cdot \phi'\left(\sum_j \boldsymbol{W}^{jk} \cdot \boldsymbol{a}^j\right) \cdot \boldsymbol{a}^j \tag{2.9}$$

For updating weights, the calculation is

$$\boldsymbol{W}^{jk} = \boldsymbol{W}^{jk} - \alpha \cdot \frac{\partial \boldsymbol{c}}{\partial \boldsymbol{W}^{jk}} \tag{2.10}$$

with α being the above-mentioned learning rate. Implementing such a neural network heavily relies on matrix multiplications which can be calculated very efficiently, especially with the help of modern graphic cards. Because of this, ANNs are in use to this day, often in combination with other network types that are presented below [GLW+21] [VSP+23].

2.2 Convolutional Neural Networks

Conventional Neural networks (CNNs) [ON15] are another type of network and are often combined with ANNs. They are used to extract features from an image by applying different "filters" to it.

A CNN consists of multiple layers as well. Like an ANN, a CNN typically includes an input layer with a size corresponding to the number of pixels in the input image and maps to a much smaller number of nodes. In a CNN, the activation of a neuron depends on a learnable kernel. A kernel is a matrix that convolves over the input vector, aiming to extract important features while diminishing unimportant ones. Convolution involves selecting a pooled vector $\boldsymbol{p}$, usually represented as a quadratic matrix $\boldsymbol{P}$ with a center value c, and a quadratic kernel $\boldsymbol{K}$. The pooled vector and kernel are multiplied together via the dot product, with the resulting value replacing c in the input vector. Through backpropagation, the kernel is trained to improve performance. [ON15]

Due to performance reasons, a CNN is often used to map a large input to a smaller dimension by adjusting the depth, the stride, and zero padding. Depth describes the number of convolution layers (a reduction leads to fewer neurons for the cost of recognition performance), the stride is the distance around different center points, and zero-padding is the *"simple process of padding the border of the input"* [ON15].

Another way to reduce the output size is by using a pooling layer. In pooling layers, a kernel (typically 2 x 2) is used with a larger stride than $\frac{width_{\text{kernel}}}{2}$ to further minimize the output array. Only the largest number within the pooled vector is carried to the next layer for max pooling. Pooling layers can also be used for other optimizations, such as L1/L2 normalization [WLD+19] [GK20] and average pooling [GK20] [ON15].

2.3 Autoencoder

The Autoencoder architecture describes a possible solution for compressing high dimensional data to lower dimensions (encoder) and rebuilding it with information, the encoder removed (decoder). The idea was first proposed in the paper "Fully Convolutional Networks for Semantic Segmentation" [LSD15] as an efficient upsampling method after downsizing an image by a CNN.

As mentioned above, autoencoders reduce their input to a much smaller space, known as the "latent space". It later upscales the vector in the smaller space again to a possible new domain, such as segmentation maps for image classification. The downscaling/encoding is performed by a CNN, consisting of multiple kernel and pooling layers. In modern autoencoders, these memorized pooled indices are filtered by a *"trainable filter bank"*[BHC15] and later injected into the decoding process to get lost information by the encoding process back into the image. The decoder is also a CNN with small modifications, often called a "deconvolution network". Instead of convolution, deconvolution takes a smaller input and expands it into a larger output with the help of a trainable kernel. The pooling layers are reversed with the help of the filter bank.

An implementation of this approach, which is used to this day is the "UNET" [RFB15]. For the encoder part the network uses *"two 3x3 convolutions (unpadded convolutions), each followed by a rectified linear unit (ReLU) and a 2x2 max pooling operation with stride 2"* [RFB15] doubling the number of feature channels (each kernel produces an output for each input, which results in $\text{fchannels}_{\text{output}} = 2 \cdot \text{fchannels}_{\text{input}}$). The decoder part reverses this process, but instead of injecting the data via a "filter bank", it concatenates the corresponding feature map to each layer. It bisects it afterward via deconvolution. That results in the decoder side having one additional convolution, reducing the "feature channels" without any concatenation to its final result.

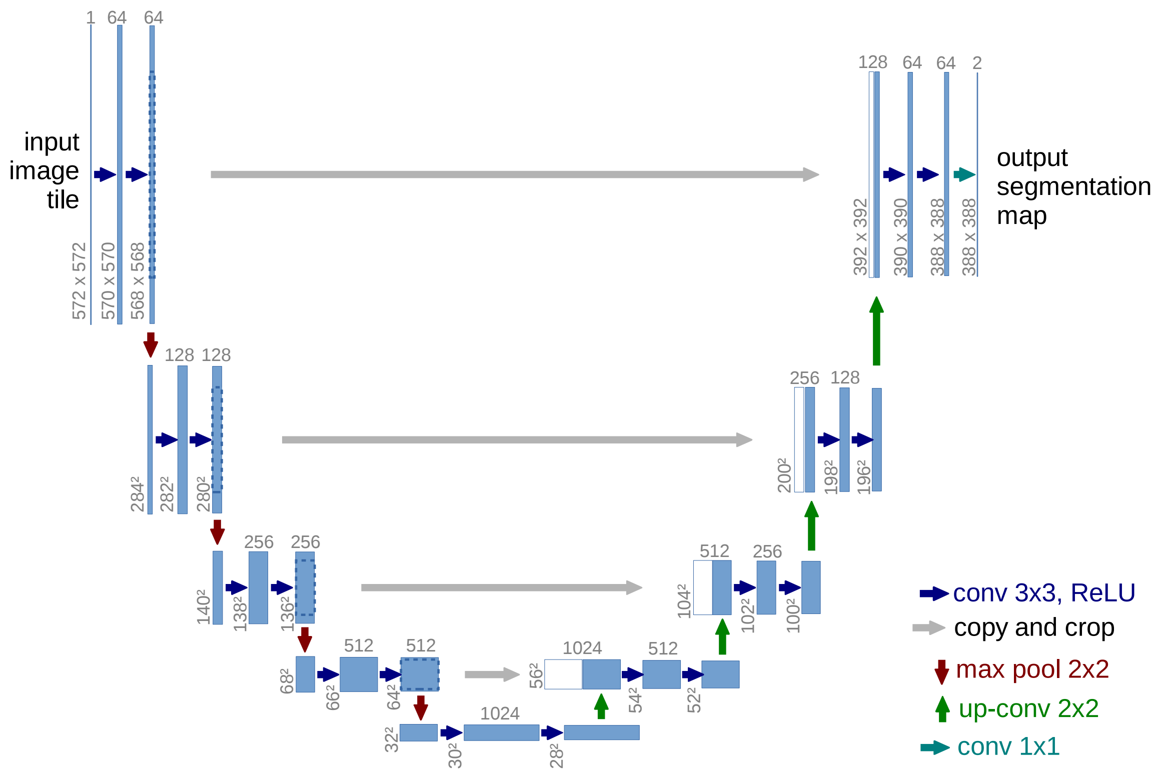

Figure 2.1: Structure of the UNET architecture. [RFB15]

2.4 GAN

Generative Adversarial Net (GAN) [GPAM+14] is an architecture for generating and validating images through a second instance. A generative model, G, and a discriminative model, D are trained together with G generating data to fool D. D distinguishes between "real" (Data from the training set) and generated, often called "fake" data. Both models are trained using backpropagation, avoiding complex inference methods. Various experiments demonstrate [ZWBR+24] [KALL18] that this adversarial process drives G to produce realistic data.

This raises the question: what are the advantages of playing such a game, and why not simply predict every pixel individually to produce a completed picture? This approach also exists, and these frameworks are called "Autoregressors" [vdOKK16]. They can be trained by removing information from an image and letting the model predict the missing information. The problem with this approach lies in the complexity of image generation. While generating embedding after embedding up to a few hundred is fine, generating a few million pixels is slow. The problem with developing a more significant number of pixels simultaneously is that Neuronal networks average their output, resulting in a blurry result.

GANs, therefore, play their adversarial game with the discriminator, allowing generation in one passthrough starting with a random vector.

Looking at the adverse game inside a GAN from a mathematical perspective, to calculate the loss for such a framework, the following value function is important: [GPAM+14]:

$$\mathcal{L} = \min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.11}$$

where G marks a parameterized function and D the discriminator, a function returning a scalar. The first part of the equation

$$\min_G \max_D V_I(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] \tag{2.12}$$

maximizes D for a given element from the data, and the second part

$$\min_G \max_D V_{II}(D,G) = \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))] \tag{2.13}$$

minimizes D for G from D's perspective and maximizes D from G's perspective if the image is a "fake".

2.5 Recurrent Neural Networks

Recurrent Neural Networks (RNN) [ZLLS23] enlarge ANNs by allowing each neuron to input its previous output again and thus maintain its state. Therefore, RNNs are well-suited for tasks like natural language processing, time series prediction, and speech recognition.

Looking into this from a mathematical point of view, the goal is to predict a token x at time t with a set of tokens that were injected into the model before t. They are denoted as x_{t-n} with $n \in \mathbb{N} \wedge t - n > 0$. Because it would be computationally too expensive to store a state of the network for each token, all of them are combined into a hidden state h. A hidden state is defined as a state inside a neuronal network that can only be calculated recursively on top of the previous result:

$$\mathbb{P}(x_t|x_{t-1}, ..., x_0) \approx \mathbb{P}(x_t|h_{t-1}) \tag{2.14}$$

Extending Equation 2.3 by the inner state with extra weights for control leads to:

$$\boldsymbol{H}_t = \phi(\boldsymbol{W}_l^{j-1}\boldsymbol{A} + \boldsymbol{W}_s^j\boldsymbol{H}_{t-1} + \boldsymbol{\beta}^{j-1}) \tag{2.15}$$

One of the problems of this approach is that for each iteration the weight $\boldsymbol{W}_s^j$ is multiplied again to $\boldsymbol{H}$. If this weight is larger than 1, over time, it leads to large gradients, also called gradient explosion. On the other hand, if it is smaller than 1, it leads to gradient vanishing [ZLLS23]. For that reason, RNNs were modified to different forms like long short-term memory [ZLLS23] [HS97] and gated recurrent networks [CGCB14], which use memory cells and gates to maintain and regulate information across long sequences.

2.6 Transformer Models & Self Attention

As mentioned above, the biggest problems regarding recurrent neural networks are gradient vanishing and explosion and not memorizing features over a large context size [ZLLS23]. To counter this, attention was created, an approach that allows finding features in early tokens and memorizing them over a long time. It was combined into a transformer, a deep-learning architecture proposed by researchers at Google. This architecture, unlike its predecessors, only uses attention to maintain its state, which increases performance regarding translation tasks and later led to the creation of Large Language Models [BMR$^+$20], as well as cross-attention and backbone architectures for latent diffusion algorithms [RBL$^+$22] [PX23].

A transformer has two different parts: an encoder and a decoder section 2.3. In this architecture, the encoder and decoder have a defined size of stacks, where each stack is again divided into two sub-layers, with the first being a multi-head self-attention mechanism and the second being a simple, position-wise, fully connected feed-forward network section 2.1. The decoder adds a third sub-layer to each stack, which performs multi-head attention over the output of the encoder stack and also masks the output tokens to not influence previous ones.[VSP$^+$23]

The core of the transformer model is the attention mechanism, which aims to enhance a token with the meanings of the other tokens around it. The idea is to map the tokens from the high dimensional embedding space to a smaller Query/Keyspace and multiply the attention pattern to the embedding. A query $\boldsymbol{q}_1$ is calculated by multiplying a matrix $\boldsymbol{W}_Q$, consisting of tunable parameters, with the token vector of an embedding $\boldsymbol{e_i}$. A key is calculated the same way, but instead of multiplying $\boldsymbol{W}_Q$ with the embedding, the key matrix $\boldsymbol{W}_K$, another matrix consisting of tunable parameters is used. After that, a new matrix is calculated by multiplying the dot product of each resulting key-value pair and multiplying it with the corresponding $\boldsymbol{v}_k$, resulting from multiplying another tunable matrix $\boldsymbol{W}_V$ by each embedding vector. $\boldsymbol{W}_V$ is normally split into two smaller matrices, as you can see later. [VSP$^+$23] [GS24]

$$\text{Attention}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{softmax}\left(\frac{\boldsymbol{Q}\boldsymbol{K}^T}{\sqrt{d^k}}\right)\boldsymbol{V} \tag{2.16}$$

$\boldsymbol{Q}$, $\boldsymbol{K}$ and $\boldsymbol{V}$ are vectors containing $[\boldsymbol{q}_1, \boldsymbol{q}_n]$, $[\boldsymbol{k}_1, \boldsymbol{k}_n]$ and $[\boldsymbol{v}_1, \boldsymbol{v}_n]$, while $\frac{1}{\sqrt{d^k}}$ counters the effect of the dot products growing large in magnitude and therefore *"pushing the softmax function into regions where it has extremely small gradients"*. [VSP$^+$23]. As mentioned above, masking is also applied so that succeeding tokens do not influence previous tokens, as the goal of a transformer is to predict these. To preserve the normalized columns calculated by the softmax function, all entries in $\boldsymbol{Q}\boldsymbol{K}^T$ representing the influence of a succeeding token are set to infinity. [VSP$^+$23]

For each token, the influences resulting from Attention$(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V})$ are added up, resulting in Δe_i. This marks the change, which is added to e_i.

To make this idea computationally more efficient, these attention heads and their corresponding changes are calculated multiple times in parallel, each attention head with its own $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$:

$$\text{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Concat}(head_1, ..., head_h)\boldsymbol{W}^O$$
$$\text{where } head_i = \text{Attention}(\boldsymbol{Q}\boldsymbol{W}_i^Q, \boldsymbol{K}\boldsymbol{W}_i^K, \boldsymbol{V}\boldsymbol{W}_i^V) \quad (2.17)$$

where $\boldsymbol{W}^O \in \mathbb{R}^{hd_v \times d_{\text{model}}}$, $\boldsymbol{W}^V \in \mathbb{R}^{d_{\text{model}} \times d_v}$ and h the number of attention layers. The previous matrix $\boldsymbol{W}^K$, which also exists h times, is split up into $\boldsymbol{W}^O$ (also called Output Matrix) and $\boldsymbol{W}^V$, where only $\boldsymbol{W}^V$ is part of the attention mechanism. With this mechanism, calculating the weighted sums on the smaller matrices from W^O and multiplying W^V is much less computationally difficult.[VSP+23] [GS24] [ENO+21].

Every layer also consists of a Fast Forward Neuronal Network (FNN, a term equivalent to ANN section 2.1). The dimensions of this network's input and output layer are d_{model}. It also has a larger hidden layer, which in the implementation of this paper [VSP+23] is scaled by factor 4.

While transformer models' most famous usage today is in "Generative Pretrained Transformers" like GPT-3 [BMR+20], most diffusion algorithms, see section 2.7 use attention mechanisms to extract meaning out of text embeddings for image generations.

Also, despite one of the biggest problems Transformers have, being their context size growing by the factor input^2 ($\text{embeddings} \cdot \text{query}, \text{embeddings} \cdot \text{key}, \text{embeddings} \cdot \text{output-matrix}$), there are implementations of modern diffusion models using transformers to replace the UNET [RFB15] [PX23].

2.7 Latent Diffusion Models & Text to Image

Instead of using GANs to generate high-quality images, (latent) diffusion models could also be used. The idea of diffusion models is to counter the blurring when generating too many images simultaneously by decoupling pixels so that they are independent of each other. This is done by adding random noise. Mathematically speaking, having an image x_0, in the encoding process, noise $\epsilon \sim \mathcal{N}(0, 1)$ is applied T times onto the image resulting in x_T. In every iteration, more noise gets added. The following interpolation can calculate each intermediate step x_t:

$$x_t = \left(\frac{T-t}{T}\right) \cdot \epsilon + \left(1 - \frac{T-t}{T}\right) x_0 \quad (2.18)$$

The decoder is a function $\mathcal{E}_\theta$ with trained parameters to revert the noise added by the encoder step by step. This is done via the UNET [PX23], which is described above. To speed up this process during the training phase, evaluation of the result from denoising x_t is not done against $x_t - 1$ but against x_0, or to be more exact, it predicts the noise ϵ_t until ϵ_0. This is really efficient because that way, the model only learns how to denoise to a clean image and not all steps in between (this does not mean that denoising is done in one step later - it is only trained to do it in one step). Looking at the loss function, it is calculated as the "Mean Squared Error" between the actual noise applied and the current image x_t, which is, as mentioned above, put together out of noise ϵ_t and x_0.

The problems resulting from this approach are the long generation time (one denoising step equals one iteration through the network) and the large amount of training data needed to make the algorithm learn efficiently. To counter both of these problems, Robin Rombach proposed in his infamous paper "High-Resolution Image Synthesis with Latent Diffusion Models" [RBL+22] to shift calculation from the pixel space to the latent space of the UNET, which is looking back at Figure 2.1 at the bottom of the U-Shape.

To train the autoencoder, equivalent to the GAN approach (see section 2.4), a discriminator is used to distinguish between "fake" images produced by the autoencoder and the real images from the dataset - the resulting loss function from this approach now combines the diffusion loss function, the discriminator loss function and a third loss function, similar to the loss calculation of variational autoencoder (VAE) [KW22]:

$$\mathcal{L}_{\text{Autoencoder}} = \min_{\mathcal{E},\mathcal{D}} \max_{\psi} \left(\mathcal{L}_{\text{rec}}(x, \mathcal{D}(\mathcal{E}(x))) - \mathcal{L}_{\text{adv}}(\mathcal{D}(\mathcal{E}(x))) + \log \mathcal{D}_{\psi}(x) + \mathcal{L}_{\text{reg}}(x; \mathcal{E}, \mathcal{D}) \right) \tag{2.19}$$

The diffusion loss is calculated in $\mathcal{L}_{\text{rec}}(x, \mathcal{D}(\mathcal{E}(x)))$, while $\mathcal{L}_{\text{reg}}(x; \mathcal{E}, \mathcal{D})$ is the regularisation of the latent image by taking the encoder output $\mathcal{E}(x) = L$, with L being the image in the latent space and calculate the loss with respect to $\mathcal{N}(0, 1)$. The last two parts of this loss function

$$\min_{\mathcal{E},\mathcal{D}} \max_{\psi} \left(-\mathcal{L}_{\text{adv}}(\mathcal{D}(\mathcal{E}(x))) + \log \mathcal{D}_{\psi}(x) \right) \approx \min_{\mathcal{E},\mathcal{D}} \max_{\psi} \left([(\mathcal{D}_{\psi}(\mathcal{D}(\mathcal{E}(x))))] + \log \mathcal{D}_{\psi}(x) \right) \tag{2.20}$$

are part of the adverse game section 2.4 and calculate the loss for minimizing the discriminator's output by adjusting the $\mathcal{D}_{\psi}$ (detecting that $(\mathcal{D}(\mathcal{E}(x)))$ is a "fake" image) while trying to maximize the discriminator's output by adjusting $(\mathcal{D}(\mathcal{E}(x)))$ (making $(\mathcal{D}(\mathcal{E}(x)))$ looks real). The last part, $\log \mathcal{D}_{\psi}(x)$, aims to maximize the discriminator's output for real images. Therefore, it is identical to the above described GAN loss function Equation 2.11

Combining these two goals of the Latent Diffusion Model denoises the image to its state before and also prevents it from being detected by the discriminator, creating a well-defined latent space and allowing consistent image generation. Instead of running the denoising function $\mathcal{E}_{\theta}$ on the pixel-space representation x_T, it is more feasible to train and run it on the latent-space representation z_T until z_0. The loss function is equivalent to the one on the pixel space. The resulting latent representation z_0 is afterward put into the decoder to obtain the final image.

The last important part of most of these models is the conditioning. Conditioning is done by concatenating the tokens with the images or by cross-attention. For cross-attention, the input is tokenized/split up (here defined as $\mathcal{T}_{\theta}$) equivalent to self-attention described in section 2.6. But instead of mapping tokens to itself, the latent image z is mapped via the Query $\boldsymbol{W}^{Q_z}$ to the Keys $\boldsymbol{W}^{K_{\mathcal{T}_{\theta}}}$ and the Output $\boldsymbol{W}^{V_{\mathcal{T}_{\theta}}}$. To enhance the influence, classifier-free guidance [HS22][Die22] was introduced, where the image is generated two times, one time with and once without the condition, and only the difference is added:

$$z_i = z_{i+1} + (\mathcal{E}_\theta(z_{i+1} \oplus \mathcal{T}_\theta) - \mathcal{E}_\theta(z_{i+1})) \tag{2.21}$$

Ultimately, the images can upscaled by latent diffusion models via conditioning [RBL+22]. That allows the diffusion algorithm to generate the image on smaller inputs, resulting in higher calculation speed.

2.8 ControlNet

The ControlNet architecture [ZRA23] aims to enable latent diffusion models with additional conditioning inputs, avoiding the need to retrain parts of the original model, which would add significant computational costs. To achieve this, ControlNet duplicates the encoding layers of the diffusion network, treating each of them as a "trainable copy". The parameters of the original diffusion network are fixed, preventing any modifications, as symbolized by the lock in Figure 2.2.

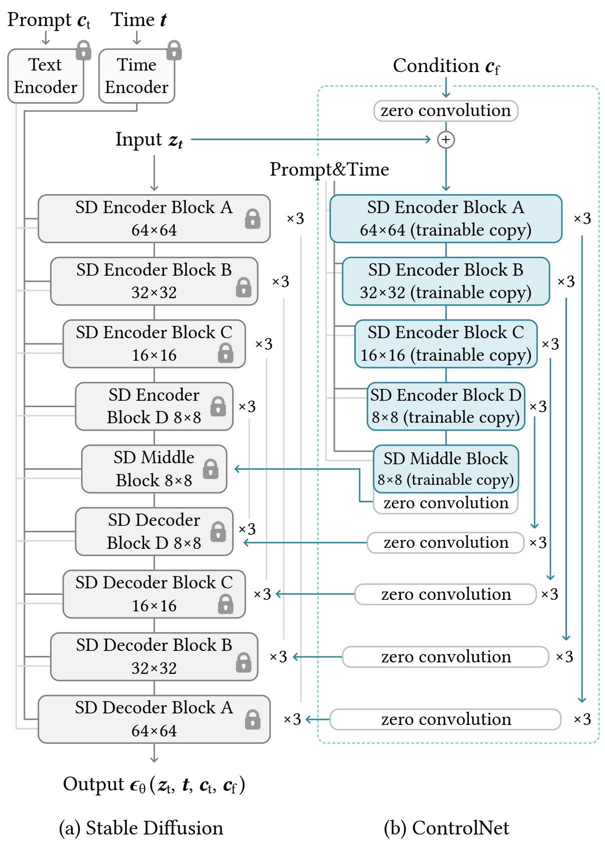

Figure 2.2: Structure of the ControlNet (right) and the Diffusion Network (left). [ZRA23]

The outputs from different layers of the ControlNet undergo a zero-convolution layer, a CNN where weights and biases are initialized to zero. However, using Gaussian weights to initialize nodes can degrade the quality. Each output from the ControlNet Encoder

Layer is incorporated into the Diffusion Decoder Layers to enhance the diffusion model with additional features. Because the ControlNet maintains a structure similar to its original form, the loss can be computed using the same method for diffusion, leading to efficient training times.

Interestingly, during training, there is a notable fixed point where the diffusion algorithm effectively utilizes the features injected by the ControlNet. This phenomenon is referred to as the *"convergence phenomenon"*.[ZRA23]

3

Related Work

3.1 Cityscapes

Cityscapes [COR+16] is a real-world dataset containing over 5000 different semantically labeled images from large German cities. These images are taken out of a car with the goal of adapting the camera angle of self-driving cars.

The train and validation labels are publicly available, while the test labels are private. Next to semantic segmentation also 2d bounding boxes and depth maps are provided in the "fine" format. Various parts of the dataset were later extended by synthetically generated weather condition scenes, e.g., different strengths of rain and fog.

Because of the dataset's size, availability, and realism, parts of the dataset have been selected as my test data set for the object detection tasks. [COR+16]

3.2 Carla

The CARLA simulator [DRC+17] is an open-source platform for research into autonomous driving and training. It provides realistic environments for testing and developing autonomous vehicle systems and offers features such as the simulation of urban driving scenarios, different weather conditions, and complex traffic situations. CARLA has different maps of urban environments for enough variety, including detailed street plans, traffic signals, buildings, and pedestrians. These environments are designed to accurately simulate real-world driving conditions. Users can create and customize driving scenarios to test specific behaviors of autonomous vehicles. This involves the ability to control weather conditions, lighting, and traffic patterns. CARLA has many sensors commonly used in autonomous vehicles, such as cameras, LiDAR, GPS, IMU (Inertial Measurement Unit), Depth Maps, Semantic Segmentation, and RADAR [DRC+17]. These sensors can be attached to the simulated vehicle to collect data for perception and navigation tasks and be controlled via a Python API. [DRC+17] In this paper, the presented pipeline leverages the CARLA python package to allow importing synthetic data directly from CARLA into the pipeline.

3.3 Synthehicle

Synthehicle [HCT$^+$23] is a synthetic dataset optimized for "multiple vehicle tracking and segmentation in multiple overlapping and non-overlapping camera views." The cameras capturing the cars are placed at high positions, like on traffic lights or next to streets, to simulate traffic control monitoring. The dataset was exported out of CARLA section 3.2 and contains 64 different environments in the four different scenes "day", "rain", "dawn", and "night". For all scenarios, 2d and 3d bounding boxen, as well as semantic segmentation, instance segmentation, and depth maps, are provided, all of them formatted as in the Cityscapes format. [HCT$^+$23] This dataset is the main used set in this thesis and takes part in the object detection tests as well as the temporal and spatial constancy tests.

3.4 Multiple Uncertainties for Autonomous Driving Dataset

The Multiple Uncertainties for Autonomous Driving Dataset (MUAD) [FYB$^+$22] is a synthetic dataset that provides highly situational events to enhance the current detection network from a car's perspective. Therefore, most of the images have realistic weather simulations, trying to recreate the real-world counterparts.

For this work, especially, the annotations are used as an alternative to the Synthehicle section 3.3 ones. The dataset consists of around 4000 images, split into training and validation sets. Each image is annotated with its segmentation, bounding boxes, and depth maps, making it a viable first-person test set for the different modules of the presented diffusion pipeline. [FYB$^+$22]

3.5 ALDM - Adversarial Supervision Makes Layout-to-Image Diffusion Models Thrive

Adversarial Supervision Makes Layout-to-Image Diffusion Models [LKZK24] is an alternative architecture to force Latent Diffusion Models (LDM) to solve Layout-to-Image tasks next to ControlNets (it is built on top of the ControlNet code) section 2.8. The core idea is instead of freezing the LDM weights to add an "adversarial game between the UNET [RFB15] and the segmenter" [LKZK24] and unroll generated noisy images later.

The training process is split into three parts: a slightly adjusted version of the "Traditional DM Training"[LKZK24], described in section 2.7, the "adversarial supervision via a segmenter-based discriminator"-part [LKZK24] to align on the ground through semantic segmentation and the "multistep unrolling strategy"[LKZK24] to guide the model during generation.

The first step is the "Traditional DM Training",[LKZK24] but instead of just training the UNET [RFB15] with the goal of noise reduction on the latent image, the semantic segmentations are also added at this training step. This allows more accurate noise prediction, leading to better output quality and more correctness regarding the semantic segmentation labels [LKZK24].

For the "adversarial supervision via a segmenter-based discriminator"-party a segmenter, in the ALDM case, the UperNet [XLZ+18], is trained alongside the UNET [RFB15] to distinguish "fake" pixels from "real" ones. The idea is that each pixel should be mapped to its ground truth label; if that is not possible, it is added to a "fake" class - this idea is denoted by a Cross-Entropy Loss function, which is used for such multi-class semantic segmentation problems [SSZ+21]:

$$\mathcal{L}_{\text{Dis}} = -\mathbb{E}\left[\sum_{c=1}^{N} \gamma_c \sum_{i,j}^{H\times W} y_{i,j,c} \log\left(\text{Dis}(x_0)_{i,jc}\right)\right] - \mathbb{E}\left[\sum_{i,j}^{H\times W} \log\left(\text{Dis}(\hat{x}_0^{(t)})_{i,j,c=N+1}\right)\right], \tag{3.1}$$

The first term:

$$-\mathbb{E}\left[\sum_{c=1}^{N} \gamma_c \sum_{i,j}^{H\times W} y_{i,j,c} \log\left(\text{Dis}(x_0)_{i,jc}\right)\right] \tag{3.2}$$

describes the cross-entropy loss for real images as a weighted sum over all classes for all pixels and the indicator $y_{i,j,c}$ (1 if pixel i, j is of class c else 0) and the second part

$$-\mathbb{E}\left[\sum_{i,j}^{H\times W} \log\left(\text{Dis}(\hat{x}_0^{(t)})_{i,j,c=N+1}\right)\right] \tag{3.3}$$

calculates the loss for fake images ($N + 1$, because N classes and 1 "fake" class). In summary, for real images, it calculates the weighted cross-entropy loss over all classes and pixels, encouraging the discriminator to correctly classify each pixel's true class, while for fake images, it calculates the cross-entropy loss for the "fake" class, encouraging the discriminator to identify fake images accurately.

Also, the Diffusion Loss is extended by:

$$\mathcal{L}_{\text{adv}} = -\mathbb{E}\left[\sum_{c=1}^{N} \gamma_c \sum_{i,j}^{H\times W} y_{i,j,c} \log\left(\text{Dis}(\hat{x}_0^{(t)})_{i,j,c}\right)\right]. \tag{3.4}$$

Instead of the ground truth label map x_0, for the generator/DM minimizing the adversarial loss $\mathcal{L}_{\text{adv}}$ for the generated image $\hat{x}_0^{(t)}$) is the goal, see section 2.4. The final function

$$\mathcal{L}_{\text{DM}} = \mathcal{L}_{\text{noise}} + \lambda_{\text{adv}}\mathcal{L}_{\text{adv}}, \tag{3.5}$$

is used to train the generator in an adversarial framework with $\mathcal{L}_{\text{noise}}$ describing the Loss function for the denoiser and λ_{adv} as a weighting factor.

The third step, "multistep unrolling strategy"[LKZK24], is used to not rely on the model's ability to denoise an image in just one step. The image is denoised multiple times - each new image is again fed into the discriminator to guide the generation even further, also at early denoising steps.

This architecture forms the foundation of this paper's presented pipeline and generates the first draft of the image via an input segmentation map. This thrives from the paper's compliance with the layout condition, which is relevant to the discussed subject. [LKZK24]

3.6 Exploring Generative AI for Sim2Real in Driving Data Synthesis

This recently published paper [ZWBR+24] explores and compares different AI generation strategies to convert segmentation-based data into real-world-looking datasets. The three selected methods are Pix2PixHD (GAN-based) [WLZ+18], OASIS [SSZ+21] (GAN-based), and ControlNet (diffusion-based, see section 2.8). The qualitative results indicate that Pix2PixHD performs the worst among the evaluated methods, producing blurred and distorted images with black artifacts that lack meaningful structure. OASIS produces images closer to real images but with significant distortions, and both GAN-based methods have problems with occlusion relationships. In contrast, ControlNet accurately represents occlusions and produces better textures, colors, and shapes, although it produces less similar images compared to the Cityscapes dataset images.

Quantitatively, ControlNet performs worse on image quality and perceptual tasks with Cityscapes labels, likely due to the superior design of the GAN-based methods, which include a loss in feature matching and content-based regularization. However, ControlNet shows better robustness and generalization. ControlNet's diffusion-based process helps avoid high-frequency distortions and checkerboard patterns, which are common in GAN outputs. Although not all models can replicate the exact color information of CARLA-generated images, ControlNet's results highlight its potential for improved texture and structural accuracy when generating synthetic data.

One of the main differences between this thesis and the one presented is the focus on object detection as well as fine-tuning. Additionally, this paper includes special weather conditions, allowing for a more focused evaluation in challenging settings. Another aspect is the use of ALDM in this thesis and the approach to combine it with other information, particularly depth, to observe the outcomes. The results from the paper "Exploring Generative AI for Sim2Real in Driving Data Synthesis" were one of the reasons for this paper to focus on diffusion-based algorithms [ZWBR+24].

3.7 Yolo and YoloX

Yolo [RDGF16] is an object detection neuronal network that performs object detection and object identification via a pre-trained backbone model combined with multiple CNNs and Feed-Forward Neuronal Networks (FNNs/ANNs). The CNN layers are used for detection and authentication, while the FNNs specialize in one of them. Later versions also use anchor bounding boxes next to further progressing backbone technology and a coupled detection head. Anchors are starting points for predicting the bounding boxes of objects, helping the model not to start from the beginning and rather have a form to begin with.

YoloX [GLW+21] is an advanced, anchor-free version of Yolo that uses a multi-head (see Figure 3.1) instead of a single-head approach to minimize convergence time. YoloX is shipped in 7 different versions, each of them with a different number of parameter sizes, making the smallest one especially attractive for less powerful hardware. It originates from the Yolo v3 because making this version anchor-free is easier than doing it with future releases where the pipeline depends more on them. For the current models, the Yolo v5 backbone models were used. In tests, YoloX outperforms its Yolo counterparts by a small margin [GLW+21], making it one of the most powerful object detection networks.

The tests ran for this thesis heavily depend on Yolo and YoloX: for object detection and fine-tuning, YoloX-x was used, while object tracking was done by Yolo v8.

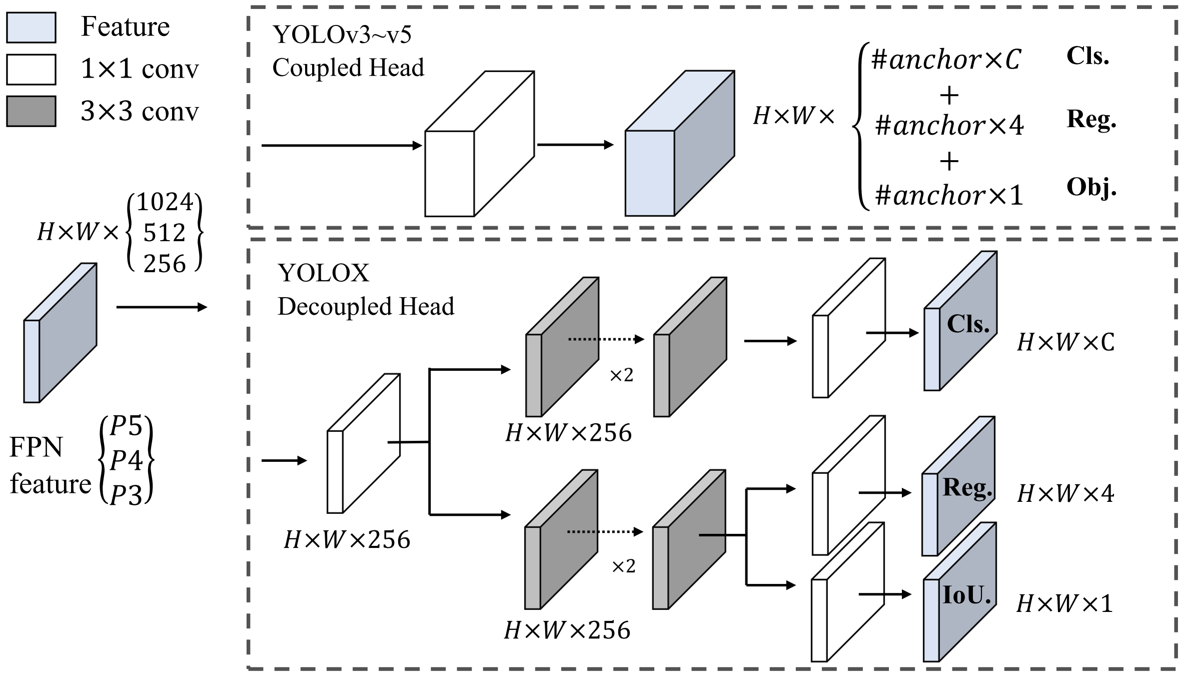

Figure 3.1: Comparison between Yolo and YoloX [GLW+21]. As mentioned, YoloX decouples its heads, allowing more accurate results for classification and bounding box detection.

4

Own Work

4.1 Stable Diffusion Pipeline

One of the thesis's main goals was to build a diffusion-based pipeline to enhance synthetic tracking data like from the Synthehicle dataset section 3.3 and evaluate them afterward. This section will focus on the software engineering side behind the pipeline implementation, explaining core features and functionality.

4.1.1 Programming language and project structure

For the programming language, Python was chosen as it has great libraries for image parsing ("Pillow") and the "diffusers" package, which allows fast integration of various hugging face models, especially in the Stable Diffusion space. Also, one of the core modules, ALDM [LKZK24], is also mostly written in Python, which makes integrating this project into the pipeline much easier.

Python allows the split of the project into multiple packages Figure 4.1, which later could be distributed easily via Conda-Forge or PyPi. Therefore, the pipeline is structured in a highly modular way, ensuring easy extendability while keeping installation requirements minimal. Each module is currently maintained within a mono repository, managed by "PDM" which is installed on top of a "Conda" environment. This unconventional package management approach offers three main advantages:

First, Conda enables the installation of packages from Conda Forge and binaries, which is particularly beneficial for optimized, platform-specific package builds.

Second, PDM assists in managing dependencies through its lock file and mono repository support.

Third, this unconventional approach allows for the installation of packages before the PDM install, facilitating the resolution of package dependencies such as PyTorch to compile instructions from YoloX.

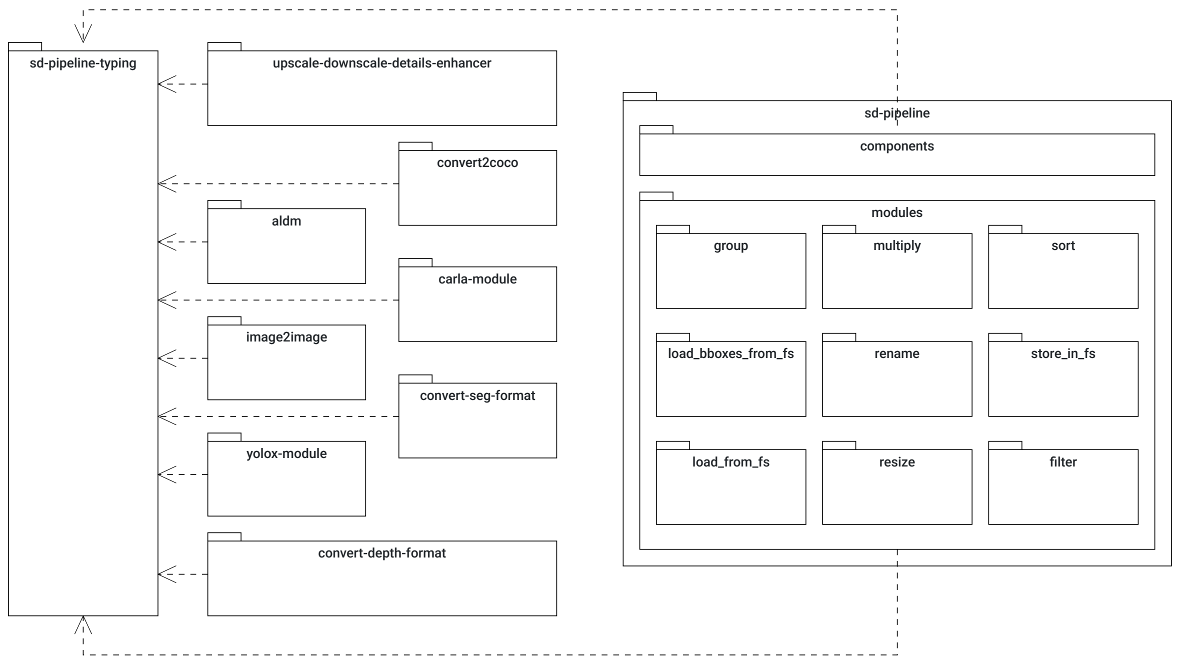

Figure 4.1: Package Diagram for the implemented modules

4.1.2 Dataflow and Stream Structure

Each function that is encapsulated in different pipeline methods accesses the pipeline's data stream. The data stream contains the image that should be processed, metadata, and other information, like bounding boxes or depth information. The implementation of the stream allows three data types:

```
stream:    Dict[str, str | PIL.Image] |
           Tuple[Dict[str, str | PIL.Image]] |
           None
```

Figure 4.2: Stream Data Type (The type should be *PIL.Image.Image*, but due to its length, it will be abbreviated to *PIL.Image* throughout the rest of this thesis.

The dictionary is the fundamental structure that contains all the important information for an image. In this context, the ”Tuple” datatype was chosen as the container for managing multiple images or inputs in one stream. Tuples were selected over lists to highlight the importance of order within the stream and to emphasize the functional concept of immutable variables.

4.1.3 Functionality

The **Pipeline** class is the heart piece of the application and contains all functions for building and executing different image processing pipelines Figure 4.3. The implementation mostly follows the builder pattern [GHJV95] by building the pipeline with different methods and getting the result after executing the run method. Therefore, a pipeline is represented through class methods that are chained together, with each method describing a step in the pipeline. Each of them returns the class object itself with a modified state by adding a function to its function array. All functions are executed in the end by the *run* method.

The **SubPipeline** class is a wrapper class around the "private" constructor of the Pipeline class (Python does not allow private constructors and does not have any equivalent to C++ friend classes. The underlying implementation, therefore, does not forbid access to the constructor from outside of the Pipeline class) and returns an empty Pipeline. This class should be used to initialize SubPipelines for different Pipeline methods.

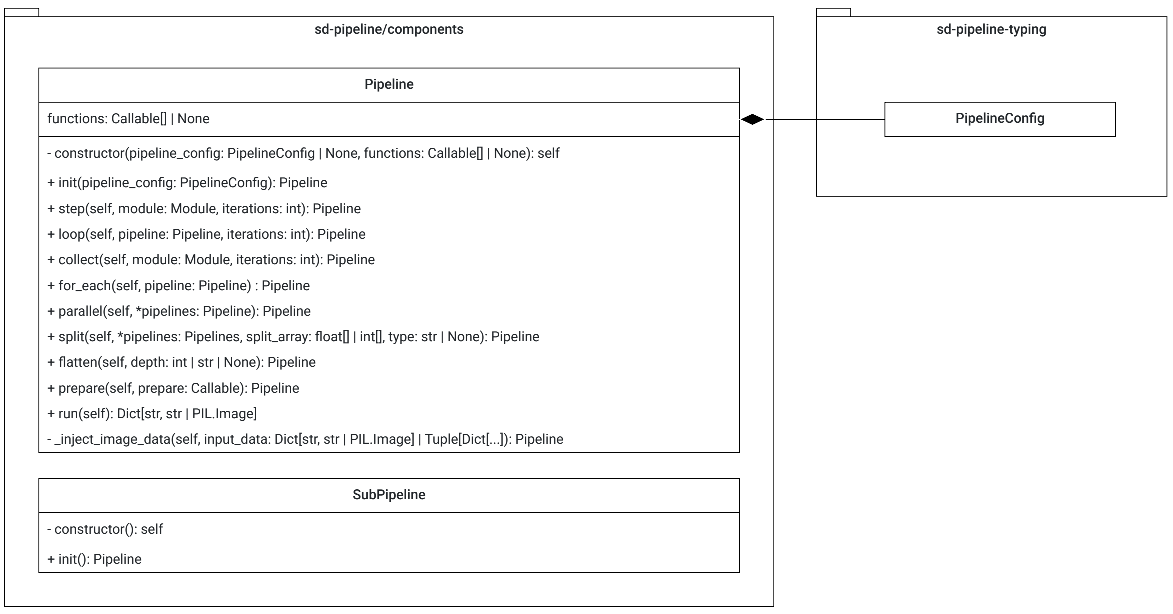
sd-pipeline/components
Pipeline
functions: Callable[] | None
- constructor(pipeline_config: PipelineConfig | None, functions: Callable[] | None): self
+ init(pipeline_config: PipelineConfig): Pipeline
+ step(self, module: Module, iterations: int): Pipeline
+ loop(self, pipeline: Pipeline, iterations: int): Pipeline
+ collect(self, module: Module, iterations: int): Pipeline
+ for_each(self, pipeline: Pipeline) : Pipeline
+ parallel(self, *pipelines: Pipeline): Pipeline
+ split(self, *pipelines: Pipelines, split_array: float[] | int[], type: str | None): Pipeline
+ flatten(self, depth: int | str | None): Pipeline
+ prepare(self, prepare: Callable): Pipeline
+ run(self): Dict[str, str | PIL.Image]
- _inject_image_data(self, input_data: Dict[str, str | PIL.Image] | Tuple[Dict[...]]): Pipeline
SubPipeline
- constructor(): self
+ init(): Pipeline
sd-pipeline-typing
PipelineConfig

Figure 4.3: Class Diagram for the pipeline component (functions with the self attribute are class methods, otherwise they are static methods)

To provide a large amount of functionality, the pipeline offers different methods to manipulate the stream of data that is pushed from function to function.

- *init()*: The init method is the only static function of the pipeline class and allows the initialization of an empty Pipeline. It helps with the injection of a pipeline config that is injected into each module 4.2,

- *loop()*: The loop method allows the execution of its SubPipeline for a defined amount of time. The output of the SubPipeline is the input for the next iteration.
- *collect()*: The collect method also allows the execution of its SubPipeline within a defined amount of time. Here, each iteration gets the same input, and the results are returned in a tuple of tuples or dictionaries.
- *for_each()*: The for_each method is often combined with the step method if specific modules can only have one image dictionary as its input. The method allows the execution of a given SubPipeline for each element in a Tuple.
- *parallel()*: The parallel method runs multiple pipelines parallel with the same input and stores its output in a tuple (the functions inside the function array are still called sequentially - the reason for that is the Graphic card storage that would be a bottleneck for most computers).
- *split()*: The split method splits its input stream into different output streams, each going into a separate SubPipeline. The ratio can be defined in absolute numbers or percentages.
- *flatten()*: The flatten method flattens the output streams by unpacking tuples for positive depth parameters. For negative depth parameters, it's the other way around; the input is wrapped in tuples.
- *run()*: The run method is the final step of building a pipeline. This method iterates over the statement array and calls the different functions added by all methods before.
- *_inject_image_data()*: This method manipulates the image stream by inserting new image data into it. Mostly, it is only used by other functions to copy a stream into a SubPipeline.

The last two functions that are not explained in this section, *step()* and *prepare()*, will be part of the next section, which focuses on exploring the module architecture.

4.2 Modules

Modules contain the main logic of the Pipeline and support one specific feature. Every module's structure is defined by the module superclass, which is part of the "sd-pipeline-typing" package, also displayed in Figure 4.1. Modules are differentiated into base modules, all modules implemented inside the sd-pipeline package, and package modules, where each package contains a different module. The base modules contain relevant IO features or easy formatting features, while the package modules contain larger features like complicated mapping tasks or difficult-based transformations.

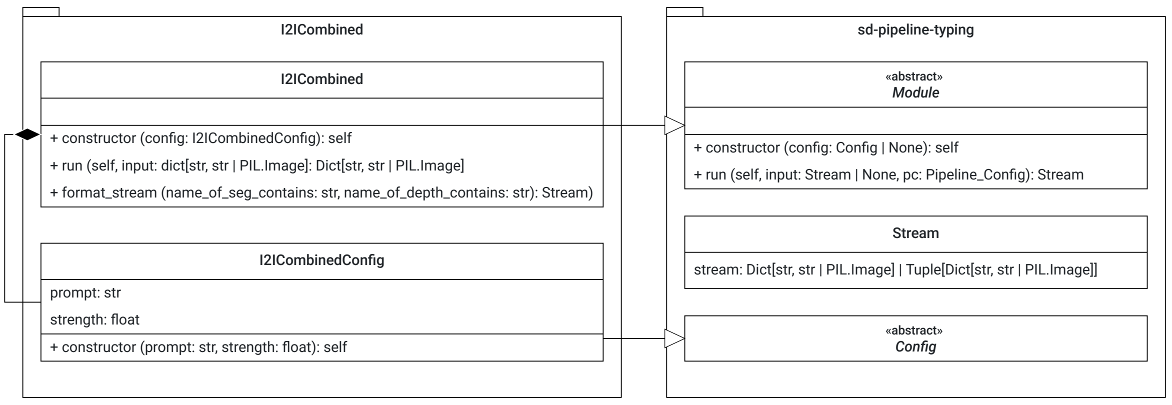
I2ICombined
I2ICombined
+ constructor (config: I2ICombinedConfig): self
+ run (self, input: dict[str, str | PIL.Image]: Dict[str, str | PIL.Image]
+ format_stream (name_of_seg_contains: str, name_of_depth_contains: str): Stream)
I2ICombinedConfig
prompt: str
strength: float
+ constructor (prompt: str, strength: float): self
sd-pipeline-typing
«abstract»
Module
+ constructor (config: Config | None): self
+ run (self, input: Stream | None, pc: Pipeline_Config): Stream
Stream
stream: Dict[str, str | PIL.Image] | Tuple[Dict[str, str | PIL.Image]]
«abstract»
Config

Figure 4.4: Class Diagram describing the module architecture

The module's interface is defined by two methods: the constructor, which initializes the module with the configuration object during pipeline programming, and the run method, which manipulates the data stream. This interface is established through the abstract class Module in the "sd-pipeline-typing" module. Additionally, the interface for configuration classes is provided within this package. All interfaces are implemented through inheritance, illustrated in Figure 4.4. Some modules also contain a static method named format_stream for data stream manipulation, i.e. to assign segmentation or depth information to the appropriate images.

To integrate a module into the sd-pipeline, the *prepare()* and *step()* methods can be used. While the *prepare()* method takes a callable as its only argument, allowing the format_stream() method to gain access to the data stream, the *step()* method takes a module as its parameter, adding the run method to the functions array and injecting the pipeline configuration.

4.2.1 Base Modules: pipeline-specific modules

Some of the modules are directly implemented into the pipeline package. These modules contain core features and build the pipeline's data flow manipulation foundation.

The first and second ones are the I/O modules *Load_from_fs* and *Store_to_fs*. They are responsible for loading images into the Pipeline via the library "Pillow" and storing them afterward.

The third I/O module *Load_bboxes_from_fs* loads bounding boxes from a JSON file into the pipeline stream, therefore providing metadata for the images that have to be matched with the help of other modules.

Rename and *Resize* are two modules to manipulate an image by changing its name via a lambda function and resizing images with the help of the "Pillow" library.

The last four modules *Group*, *Multiply*, *Sort*, and *Filter* focus on manipulating not the images of the stream directly but changing the stream's behavior. *Sort* sorts the dictionaries inside a stream by a given key, *multiply* multiplies each dictionary by a defined number and *group* groups n streams together by combining all dictionaries from the same positions into a new tuple and filter removes image dictionaries that fail a predicate function.

4.2.2 Package Modules

The package modules can be installed separately, and they are the main processing units of the pipeline. Therefore, this thesis will explain each of these modules in detail, also providing examples for the generated images and comparisons with a focus on object detection in chapter 5.

4.2.2.1 CARLA

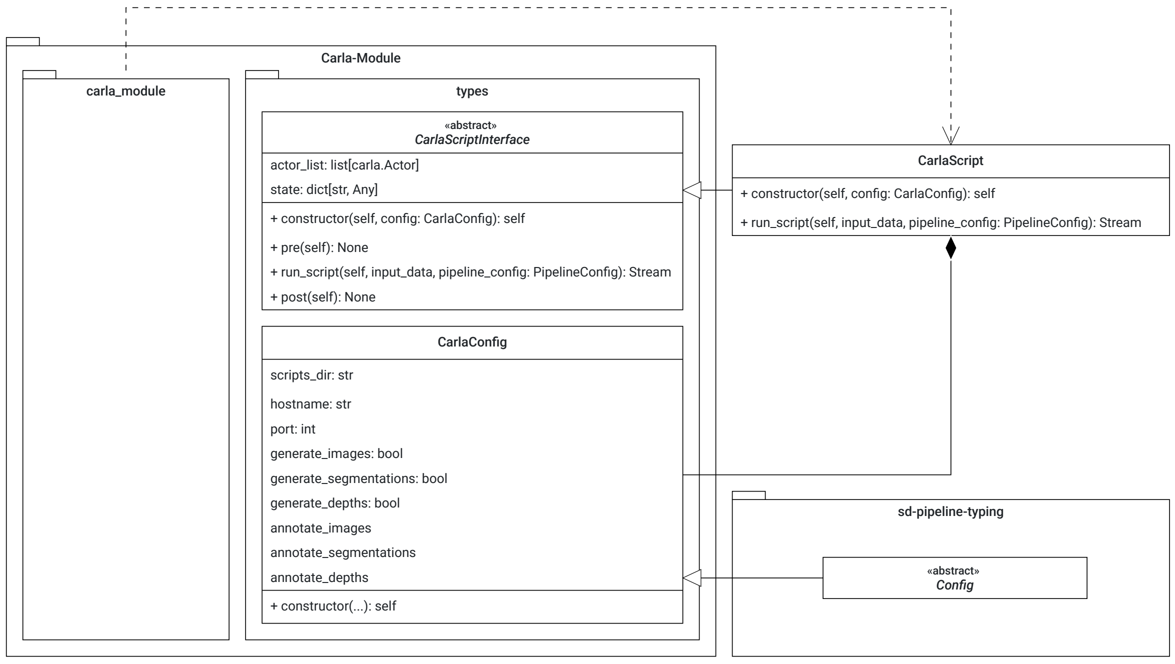

Figure 4.5: Class Diagram describing the Carla script import architecture. The Carla module package is here presented as a black box. For more detail on how modules are structured, refer to Figure 4.4

The CARLA module is logically the first module and facilitates data extraction from the CARLA Simulator (see section 3.2). Using the CARLA Python package, it is possible to configure CARLA's simulation and export data such as semantic segmentation, depth information, and bounding boxes. To provide the pipeline user with complete flexibility for configuration, the module allows the loading of custom scripts called "Carla Scripts." These scripts are used to program the simulation. The interface "CarlaScriptInterface" is provided to help the user maintain the necessary structure (see Figure 4.5).

The module provides the functions *pre()* and *post()*, which are called before and after the *run_script()* function to set up the basic environment and clean up. The *run_script()* method itself is called from the carla_module during the pipeline run, executing the script.

Safety is a crucial topic, and it is important to acknowledge that this module should not be used in a public environment. The "Carla Scripts" are loaded without validation, which could allow malicious code to be executed during a pipeline run.

4.2.2.2 ALDM

ALDM (see section 3.5) is the first step of the image generation process, allowing a segmentation map to be transformed into a realistic, detailed image with the help of a Stable Diffusion algorithm and a control net. To integrate the code into the pipeline, it had first been made installable as a package. Therefore, rewriting the original ALDM code and creating a wrapper around it to be called from my pipeline was necessary. Additionally, the configuration settings are extracted into a configuration class to make them accessible at the pipeline programming level.

4.2.2.3 Image2Image

The Image2Image module extends the ALDM processing by enabling further image manipulation after its initial creation. As described in section 2.8, image-to-image processes take an existing image as their base and generate a new one from it. This generation can be guided by ControlNets. This module implements multiple variations, including image-to-image without a ControlNet, the ControlNet trained on semantic segmentation by Lvmin Zhang [ZRA23], the ControlNet trained on depth by Lvmin Zhang [ZRA23], and a combination of both. All modules work with Stable Diffusion 1.5 [RBL$^+$22] as it was the only available Stable Diffusion checkpoint with a ControlNet trained on both inputs.

4.2.2.4 Convert Seg Format & Convert Depth Format

To provide correctly formatted segmentation and depth information for the ControlNets in the image2image module, the segmentation maps and depth maps must be properly modified. The convert-seg-format module offers methods to transform a semantic segmentation map from the Cityscapes dataset [COR$^+$16] format to the ade20k dataset [ZZP$^+$19] format and vice versa. This is essential because Sythehicle [HCT$^+$23] provides and ALDM [LKZK24] consumes Cityscape-formatted segmentation maps, while the ControlNet requires ADE-formatted segmentation maps. The convert-depth-format module provides two functionalities. The first one inverts the depth map (Sythehicle provides depth information with the nearest objects being the darkest, while the depth control net requires the reverse), and the second one includes a denormalizer that allows a function to run on each pixel of a depth map, reverting possible normalizations.

4.2.2.5 Upscale-Downscale-Details-Enhancer

The upscale downscale module tries to enrich image data with more details by first upscaling the image with the help of a latent diffusion network [RBL+22] and then downscaling it with "Pillow". This should help with missing details and broken proportions by adding/repairing them during the upscale process and not losing them completely during the downscale. In combination with the loop method provided by the pipeline, which can be found in subsection 4.1.3, this module could enhance the image multiple times before passing it to the next pipeline method. Its idea originates from the DLSS-Algorithm developed by NVIDIA [Wat20] for PC-Games, which also takes a low-resolution image, upscales, and afterward downscales it to the monitor resolution to provide better image quality.

4.2.2.6 Convert to COCO

As the package name already states, this module converts an input stream into the COCO format [LMB+15], exporting the label files as JSON. With the help of the *split()* method, subsection 4.1.3, and some prepossessing functions, the input stream can be split to allow the dataset to contain the train, test, and validation labels. The purpose of this module is primarily to help with the evaluation through the *yolox* module.

4.2.2.7 YoloX

The YoloX module is the main evaluator module providing object detection on COCO datasets [LMB+15]. YoloX provides different models with different layer sizes, offering large flexibility for training. The module wraps the YoloX package [GLW+21], allowing it to be integrated into pipelines seamlessly.

4.2.2.8 Finetuning

A key feature of YoloX is the ease with which the model can be trained or fine-tuned. The pipeline is designed to facilitate the fine-tuning of existing checkpoints using newly generated data, ensuring that the model can be enhanced by that.

4.2.2.9 Resulting checkpoints and evaluation

After every epoch, YoloX provides a new checkpoint with the updated weights and training results consisting of the precision and recall values for each class. The module also exports them, providing the possibility to compare different pipeline configurations and use the trained checkpoint for other projects chapter 5.

4.3 Visual and Object detection testing

Another part of this thesis is the evaluation of the different modules and a detailed comparison between the results of different pipelines. Multiple tests regarding object detection have been run to test the performance and quality of the different results, and the output has been analyzed visually.

4.3.1 Test setup

For the major generation modules, small tests have been conducted to determine the best configurations. Following this, a dataset containing 300 segmentation maps was generated from six different settings within the Synthicle dataset (Town01-N-..., C01 - CO6, 50 segmentation maps from the selected scene) (section 3.3) and the MUAD dataset (300 images out of the validation set, because annotations for the test set was not available) (section 3.4). Four main themes, daytime, dawn, rain, and night were applied to the 300 segmentation maps, resulting in 1,200 images. This process was repeated twice, ultimately producing 2 datasets, each containing a total of 2,400 finalized images.

Due to time constraints (the generation for every dataset took over 3 days), the images from other datasets were used as a base for new ones. For example, when using the image2image module, the output from the ALDM Module was taken as the new input instead of regenerating the images resulting from the ALDM modules. While this could carry graphical errors through the tests, new generations could also introduce new graphical errors, making this an efficient trade-off between time and dataset size.

The dataset generation and the later conducted tests run on two machines, with the first one having two NVIDIA Titans and the other one having an NVIDIA RTX 4070. On both machines, the tests were run with the same YoloX settings.

4.3.1.1 Object detection with pre-trained model

With the resulting images, object detection for the train images was conducted to measure the quality of the different data sets by looking at the average precision (AP) and average recall (AR) the pre-trained COCO weights for the YoloX model can achieve (more details on these measurements in subsection 5.1.2).

4.3.1.2 Object detection model finetuning

With the resulting datasets, the YoloX reconfigured COCO weights were further finetuned and evaluated with 600 random images from the Cityscapes dataset to test if the module could be further optimized through data generated with a Stable Diffusion algorithm. As mentioned in section 3.1 the dataset was enhanced by adding foggy and rainy scenes to make detecting cars harder.

4.4 Consistency and tracking testing

Maintaining spatial and temporal consistency between different image generations is a problem most diffusion modules have. Four different test approaches were conducted to visualize the problem and test how spatial and temporal consistency could be preserved. For each test, the same six starting images were used.

4.4.1 Test setup Baseline

The Synthhicle dataset was used for these tests, as the images originate from videos and are, therefore, sequential. Per scene, 100 images were taken, and we are building the baseline for the other tests that will modify these images.

4.4.2 Test setup ALDM

This test focuses on generating 100 images with ALDM. The input was 100 segmentation maps that followed one another, resulting in a 10-second-long video with 10 frames per second.

4.4.3 Test setup image to image, segmentation ControlNet

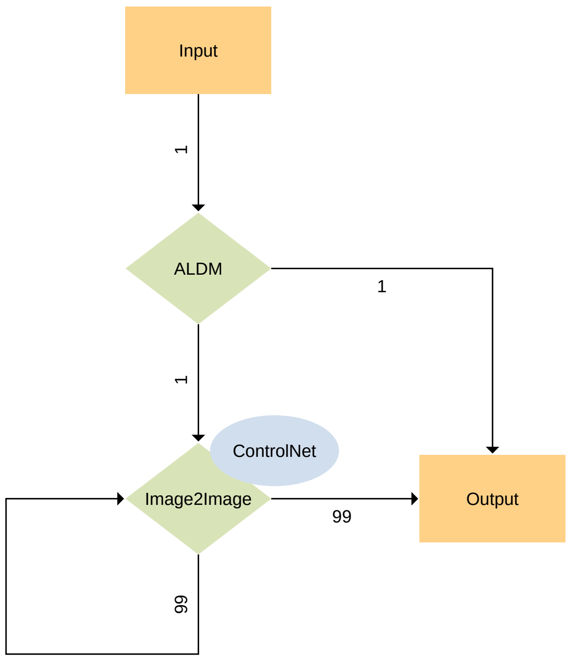

Figure 4.6: Visitation of the i2i consistency pipeline

For the next test, as shown in Figure 4.6, only the first segmentation map is used as input for the ALDM module. The resulting image is the starting point for the image-to-image process, with every step generating a new frame with the previous output as its next input. To guide the generation for the following frame, the segmentation ControlNet is used together with the next fame segmentation map. The resulting video is 10 seconds long with 10 frames per second. During different tests, the prompt of the image-to-image module was changed to cover multiple scenarios. For the final result, the seed and the strength parameter were kept the same (see subsubsection 5.2.1.3 and subsubsection 5.2.1.4 for more details).

4.4.4 Test setup image-to-image, segmentation & depth ControlNet

This test has a similar test setup as the previous one, only differing in the used ControlNets, combining segmentation maps with depth information.

4.4.5 Image constancy evaluation

To evaluate the consistency of these different videos, the previous image is compared to the current image via the Root Mean Square Error (RMSE; for calculation details, refer to the used implementation) to see how big the change between two consecutive frames is. This is compared to the synthetic image video (the baseline).

4.4.6 Object tracking evaluation

With the help of Ultralytics, Bytetrack [ZSJ+22], and pertained Yolo v8 [RKHD24] weights, every vehicle in the video was tracked during the 10 seconds and labeled with a unique ID. The IDs are assigned in ascending order, increasing by 1 for each subsequent vehicle. If the tracking algorithm loses the car, a new ID is assigned. In the end, the difference between the number of IDs counted on the synthetic image video (the baseline) and the number of IDs is used as a metric for the quality of the generated cars and their stability between different frames.

5

Evaluation

5.1 Object detection

5.1.1 Visual classification of the output images

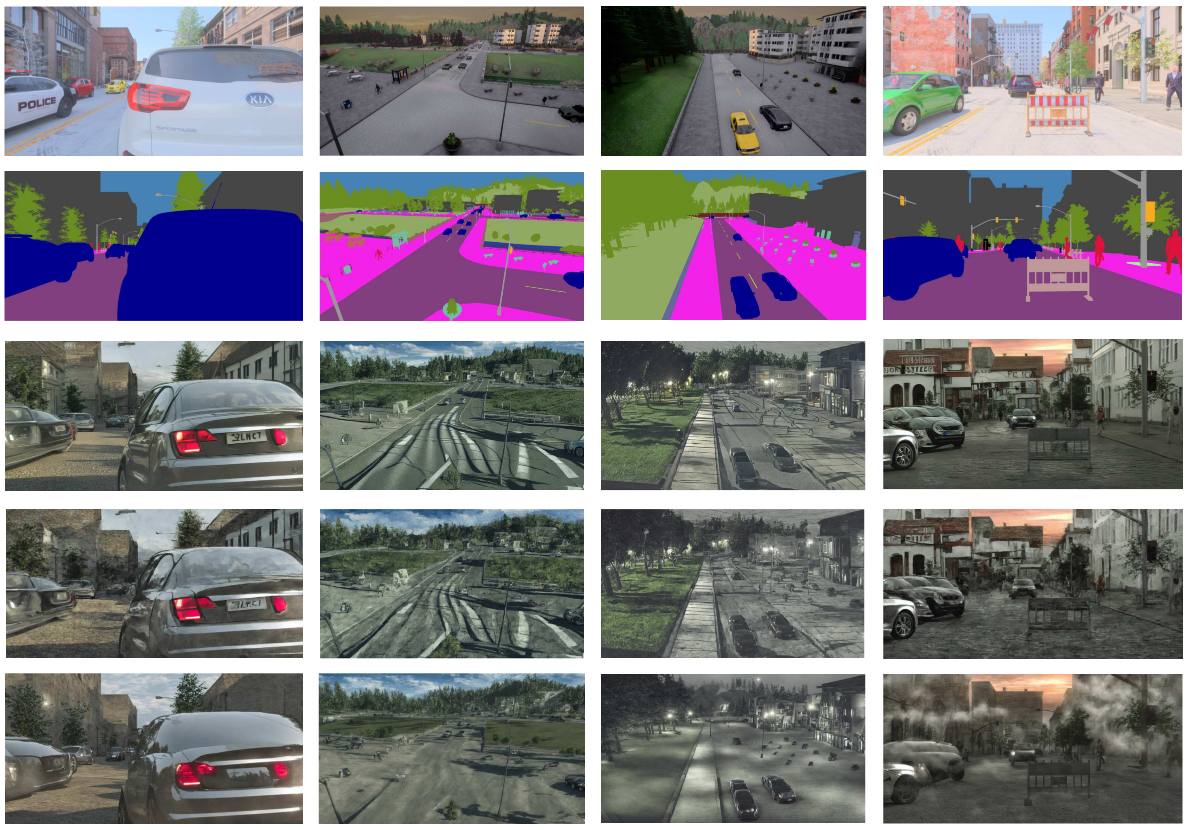

Figure 5.1: Comparing the different results of the generation. From top row to bottom row: synthetic data from MUAD and Synthehicle, segmentation maps, Output ALDM, Output ALDM + I2I Seg, Output ALDM + I2I Seg + Depth

All generations often have problems finding the right car proportions, leading to curved edges and unnatural shapes. This happens especially with objects close to the camera, hinting that ALDM has not been trained in this type of situation. Looking at Figure 5.1, ALDM dominates the car generation, especially for the out-of-the-car view (from now on called first-person view), while for the top-street view (from now on called third-person view), Stable Diffusion 1.5 with the image-to-image module is able to correct the outline of cars and streets, especially if the cars are small. For closer cars, details are lost because of the added noise from the Stable Diffusions image-to-image process, which is not completely removed. This is especially noticeable in the image-to-image generation with only segmentation maps. Adding depth maps and a slightly higher strength parameter (for more details, see subsubsection 5.2.1.3), which is possible due to the extra guidance through the depth maps, the noise can be heavily reduced, often resulting in random clouds in the image (see in Figure 5.1).

Stable Diffusion 1.5 with ControlNets really amplifies weather conditions and lighting. While ALDM's lighting and weather conditions are mostly slightly hinted at, ControlNet overdoes it sometimes, leading to a more interesting but less realistic image. Comparing images in Figure 5.2, one can spot the new lightning and reflection added by Stable Diffusion 1.5.

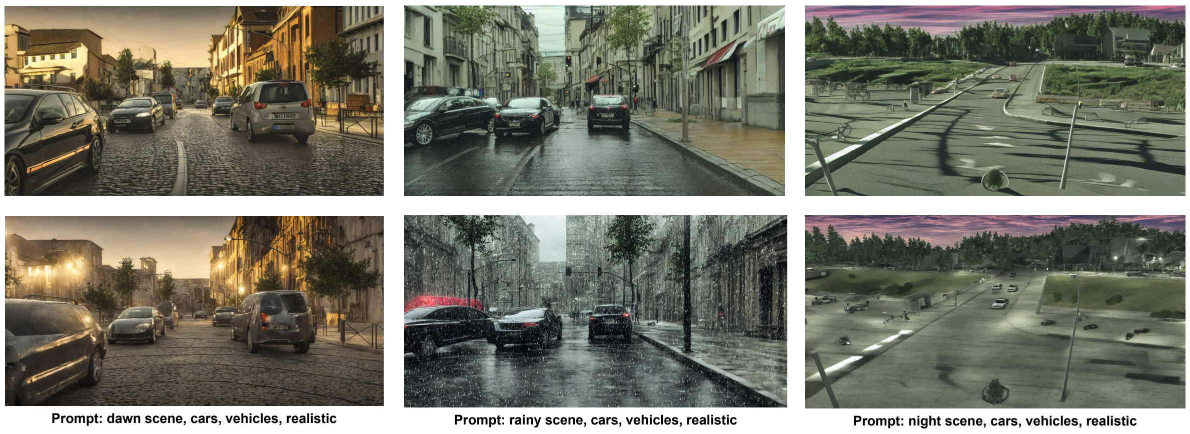
Prompt: dawn scene, cars, vehicles, realistic
Prompt: rainy scene, cars, vehicles, realistic
Prompt: night scene, cars, vehicles, realistic

Figure 5.2: Looking at different prompts, Stable Diffusion 1.5 with segmentation and depth ControlNet (bottom) highlights and enhances the weather and lightning effects of ALDM (top)

5.1.2 Test results object detection

To evaluate test results for object detection, two matrices are highly important: the average precision (AP):

$$AP = \frac{\text{True Positive}}{\text{True Positive} + \text{False Positive}}$$

and the average recall (AR):

$$AR = \frac{\text{True Positive}}{\text{True Positive} + \text{False Negative}}$$

The average precision, which is the main metric used to evaluate performance in YoloX, shows the ratio of rightfully detected pixels for a class to all detected pixels. The average recall shows the ratio of how many pixels of one class were detected from all pixels that are part of the class.

5.1.3 Test results object detection on train set

Looking at Table 5.1, the raw synthetic dataset has the highest AP of both datasets, resulting from better-defined car shapes and proportions compared to the diffusion approaches. ALDM I2I Seg + Depth outperforms the base ALDM and I2I Seg in MUAD and Synthehicle, especially regarding its AR, outperforming the pure synthetic data as well. Looking at the AP performance, the gain of image-to-image is much less for the first-person dataset because the images generated by ALDM already have better proportions there. This only results in more added noise by the image-to-image module, explaining the large gap in the raw synthetic data. Also, Synthehicle's data is much less detailed, giving another advantage to the diffusion approaches and resulting in closer AP values on Synthehicle's tests. The good AR performance in the ALDM I2I Seg + Depth approach could also result from the higher contrast of image-to-image described in subsection 5.1.1 leading to more cars getting detected.

	MUAD		Synthehicle	
Metrix	AP	AR	AP	AR
Raw Synthetic Data	**55.20**%	**60.00**%	**18.49**%	23.57%
ALDM	38.30%	45.44%	15.38%	24.48%
ALDM + I2I Seg	34.19%	41.75%	11.67%	22.09%
ALDM + I2I Seg + Depth	44.30%	50.49%	16.85%	**29.19**%

Table 5.1: Results of running YoloX-x with the default coco weights on the train sets of the generated datasets.

5.1.4 Evaluating the test set for fine-tuning

The result set based on Cityscapes is used across all resulting checkpoints, helping to compare the results of different generation methods. The detection of cars seems difficult for the model, as the base weights for YoloX-x only archive an AP of 33.45%. Reasons for that could be the low contrast of the images, the variety of different scenarios, and the different weather conditions that are part of this set. Also, the weather conditions are strong, making object detection difficult even for human eyes. Combined with the long training times, this leads to only finetuning the existing COCO weights, which should yield a better result. It is also important to mention that the colored segmentation maps of Synthehicle only use one label to describe vehicles. For that reason, during fine-tuning, the model has to relearn to classify the other COCO classes like *Bus* and *Truck* as *Cars*, being a new challenge.

5.1.5 Test results fine-tuning

	MUAD		Synthehicle	
Metrix	max AP (val)	AP (test)	max AP (val)	AP (test)
Raw Synthetic Data	63.07%	**30.68%**	**88.96%**	**5.95%**
ALDM	67.95%	25.77%	77.09%	3.42%
ALDM + I2I Seg	56.70%	24.32%	64.39%	2.43%
ALDM + I2I Seg + Depth	**69.71%**	27.19%	79.35%	3.82%

Table 5.2: Results of fine-tuning the COCO weights with the different generated datasets. The AP(tests) results are taken from the epoch with the highest AP in the value sets.

This test does not aim to compare the two datasets with each other because the test setup heavily favors MUAD. One reason for this is that the Cityscapes dataset, used as the evaluation set, is also a first-person view dataset. Another reason is that ALDM employs weights trained on Cityscapes, which enhances its performance on first-person generation tasks. Also, the selection of the MUAD dataset is much more diverse in terms of different scenarios, resulting in an even larger advantage. This test focuses more on comparing the performance of all four test setups inside the datasets. As displayed by Table 5.2, preparing data with Stable Diffusion algorithms does not perform better than pure synthetic data because of the inconsistency regarding image quality, as seen above, and the distortion of car proportions. While this may not be that bad for detecting the different vehicles, for training, where clean data is really important [LKKV14], this results in worse performances.

Comparing the different diffusion settings, also in this test, ALDM + I2I Seg + Depth outperforms the other two approaches in AP (val) and AP (test) at MUAD and Synthehicle. An explanation for this can be found in subsection 5.1.3.

5.1.6 Discussion: Potentials and limitations of diffusion-based synthetic data

Bringing the results together, it can be concluded that ALDM and the post-processing algorithms used are not yet accurate enough to replace synthetic data. The biggest problem can be traced back to the proportions of the generated objects, which are often distorted and do not change enough after post-processing with the help of depth information. The tests also show that training makes the biggest difference as the APs and ARs are generally better with MUAD than with Synthehicle, which is mainly due to ALDMs training on Cityscapes. Another discovery is that depth information usually improves a previously generated image, outperforming generations with segmentation only. It would, therefore, be interesting to modify ALDM to allow the input of both segmentation and depth information, which could significantly increase the performance.

5.2 Spatial-temporal image consistency

5.2.1 Overall findings

5.2.1.1 The importance of the prompt

As various tests have shown, the prompt selection for the ControlNet to pick up all significant elements of the picture is essential. As displayed in Figure 5.3, the overall setting and structure are captured mainly by the ControlNet, only slightly changing in the background. However, details, especially the cars, are not picked up if they are not mentioned in the prompt. On the other hand, if mentioned in the prompt, things not labeled as cars often transform into them over time. As seen in the pictures, with the help of depth control, most of these issues can be fixed.

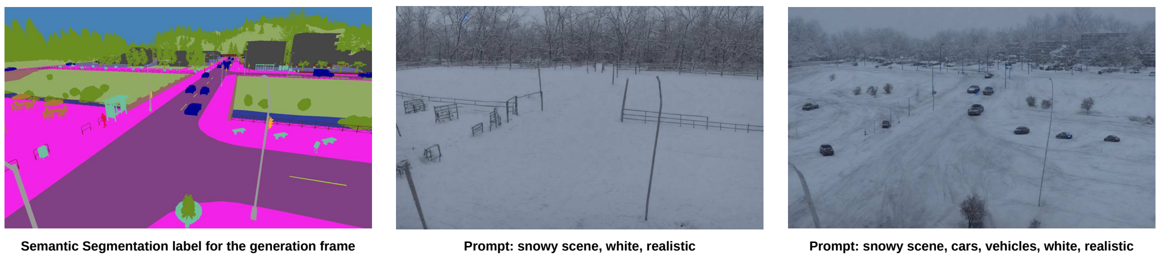
Semantic Segmentation label for the generation frame
Prompt: snowy scene, white, realistic
Prompt: snowy scene, cars, vehicles, white, realistic

Figure 5.3: The results for different prompts on the image.

5.2.1.2 Quality of the starting image

The quality of the starting image for the image-to-image approaches is not essential for the generation by itself. As different generations show, in most cases, the generation fixes most errors over time, allowing smooth generation for later frames. This effect is also strengthened by the high-strength parameter discussed before.

5.2.1.3 High score for the strength parameter

The strength parameter defines how much the input image has influenced the output image by changing the number of noising steps before the denoising iterations. Temporal consistency is broken if the score is too low, leading to no moving cars and heavy distortion because there is not enough noise added to move the cars. Unintuitively, for tests with the strength parameter set to 0.6, it sometimes happens that the cars disappear in later frames - part of the explanation could be that with 0.6, there is enough noise to let the car vanish but not enough denoising steps for the control to actively influence the generation sufficiently. Therefore, a score of 0.8 was used for the generation as it leads to consistent results in terms of temporal consistency (with the cost of spatial consistency as discussed below).

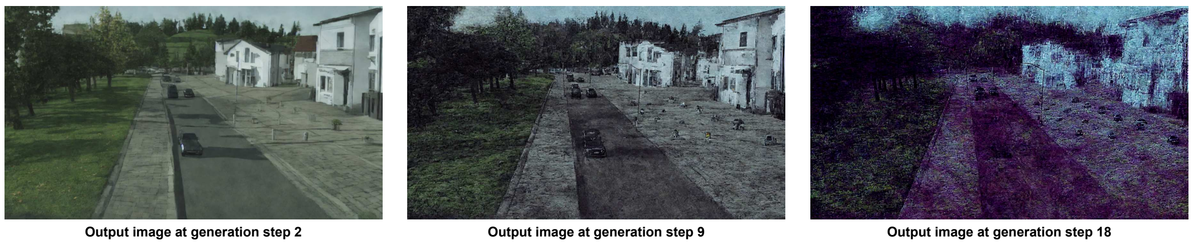
Output image at generation step 2
Output image at generation step 9
Output image at generation step 18

Figure 5.4: Image distortion for a strength score of 0.2.

5.2.1.4 Color shifting of the scenes

Stable diffusion 1.5 amplifies an image's overall color tone, especially shifting it into the red scope. If not defined by the prompt by specifying a different color or excluding red by the negative prompt, the image will turn red for every generation step, an error the network cannot fix. This happens to almost all scenes and is not changed by the base tone of the starting image. To resolve this issue, next to the prompt, a seed must be set for the generation and has to remain the same for all generation steps. This removes the red-shift as well as other effects that otherwise would appear (e.g., on rare occasions, generating a depth map generates a vignette effect around the edges of the picture)

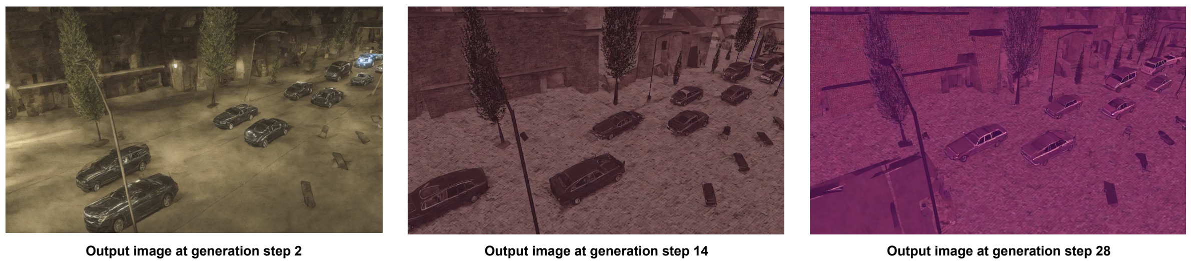
Output image at generation step 2
Output image at generation step 14
Output image at generation step 28

Figure 5.5: Red-Shift progression over time.

5.2.2 Test results

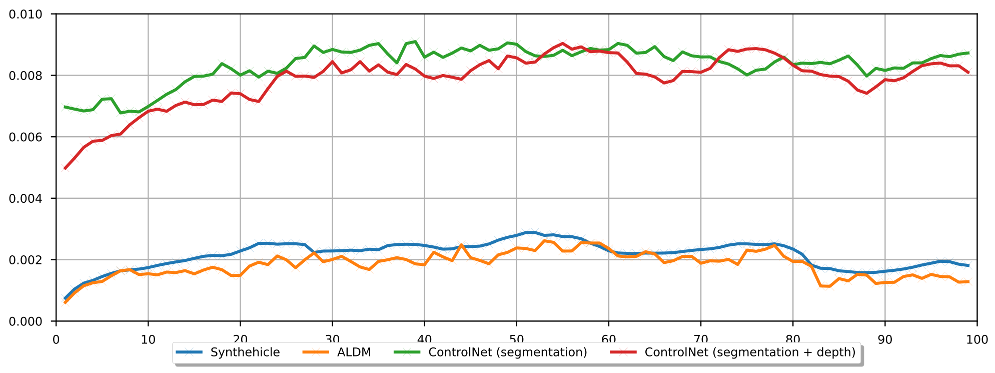

Figure 5.6: RMSE between two following frames. Nearer to the Sythehicle baseline video means better consistency between two frames.

Video	Syntheticle	ALDM	I2I ControlNet (seg)	I2I ControlNet (seg + depth)
Scene 1	7	**18**	36	24
Scene 2	15	**20**	29	22
Scene 3	2	**4**	14	**4**
Scene 4	8	**8**	23	24
Scene 5	8	**7**	63	13
Scene 6	9	15	–	**14**
Avg	8.2	**12.0**	–	16.8
Avg -1	8.2	**10.2**	29	14

Table 5.3: Results of object detection tests running on the six scenes. The original video marks the baseline. Avg -1 describes the average of all results, but the baseline score replaces the worst score of a generation method.

5.2.2.1 ALDM

As the test results show, using ALDM for generation yields results that are nearly equivalent to the original video in terms of image stability as well as in the object detection tests. As depicted in Figure 5.6, the ALDM model dominates the other approaches regarding RMSE between the following frames, showing how stable the images are. Compared to the original video, the RMSE is almost equal. Looking through the videos, spatial consistency is broken at two things: lightning and cars. Especially the cars, the only moving element in the videos, cannot persist in color or shape, even when the same seed is used. This comes with no surprise because of the change in the segmentation map; different pixels are now denoised to generate the cars, leading to other cars.

Also, this approach lacks information from the previous frames, making feature-sharing impossible. This also partially explains the gap between the baseline video from Synthehicle and ALDM in the object detection results Table 5.3. Another problem with ALDM discussed earlier as well, is its training on first-person view data, making it extra hard for the model to generate realistic shapes for the cars from a third-person perspective.

For special constancy, the videos generated by ALDM follow the underlying segmentations and, therefore, are fully consistent with the ground truth.

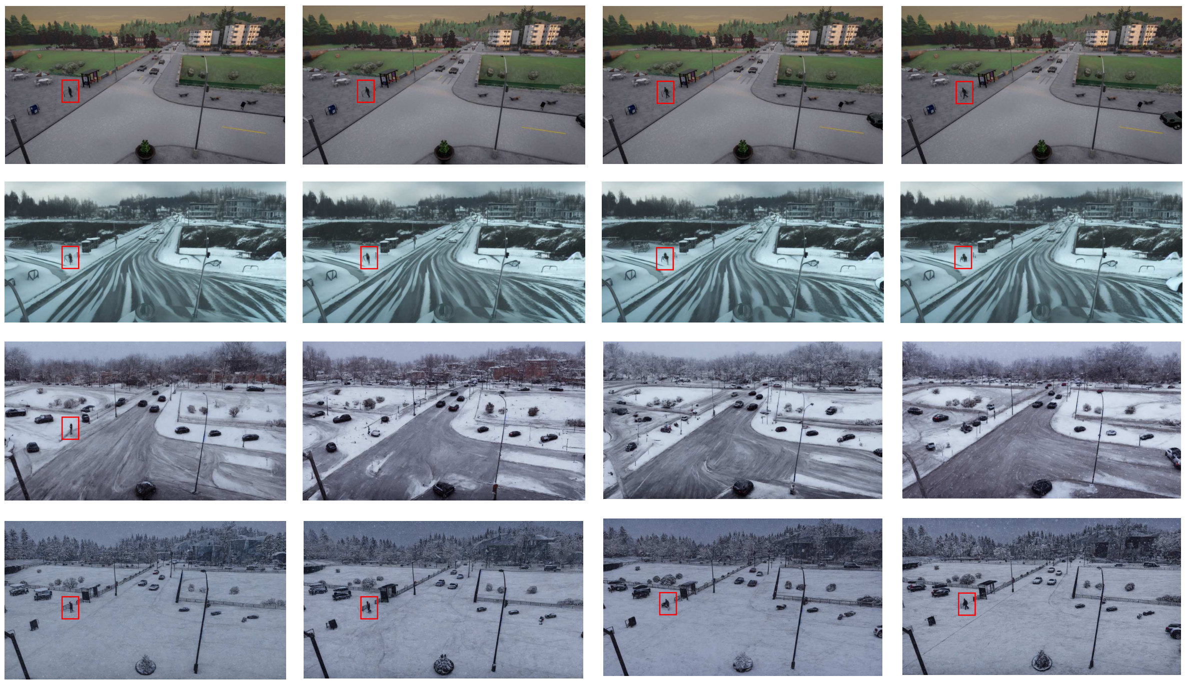

Figure 5.7: Frames 21-24 of Scene 1. From top row to bottom row: synthetic data from Synthehicle, Output ALDM, Output I2I Seg, Output I2I Seg + Depth. The red boxes mark where a pedestrian can be seen clearly.

5.2.2.2 Image-to-Image with the Segmentation ControlNet

Looking at the RMSE, image-to-image with a segmentation ControlNet overall performs the worst out of all tested approaches. The information from the previous image combined with the segmentation is too little information to obtain a consistent background, resulting in new generations in each frame.

This is especially bad for the car generation because, together with the prompt, objects that are distorted by previous generations are often converted to cars, resulting in really bad performance in object tracking.

5.2.2.3 Image-to-Image with the Depth and Segmentation ControlNet

For the depth images, it is important to notice how much better details are captured during generation. Looking at the person in Scene 1 Figure 5.7, in contrast to generation only with the segmentation map, the person is visible and moving correctly.

This behavior is also captured by the object tracking test. While the depth map does not help that much with maintaining a stable image (see Figure 5.6), the performance for car tracking drastically improves over the ControlNet with segmentation, sometimes getting even closer to the original Syntehicle video than ALDM.

5.2.3 Discussion: Spatial-temporal image consistency

ALDM dominates in this area with its capabilities to maintain a very good spatial and temporal consistency, while the image-to-image approaches fall short of expectations. Particularly noteworthy here is the preservation of the background, which is almost identical to the previous image. However, it is important to note that all these attempts are based on still cameras. This means that the same noise remains in the same place, which makes it much easier to keep the background stable. With a static camera, it should also be possible to transfer the properties of the moving objects to the subsequent frames with relatively little effort and thus make the cars consistent. With a moving camera, ALDM would have to be fundamentally modified. An example of how this could work would be the TemporalNet, which can generate the subsequent frame for another frame by being trained on time series data. Even if the presented results have strong artifacts, TemporalNet's outputs are much more stable than Stable Diffusion outputs. Another possibility would be to go the way of OpenAI Sora [LZL+24], which trains its entire model on time series and outputs its result at once and not iteratively. The problem with Sora's approach is the high training cost, long training time, and low data availability.

6 Conclusion and Outlook

6.1 Conclusion

In conclusion, this thesis introduced a user-friendly pipeline that integrates an intuitive interface with Stable Diffusion and evaluation frameworks, enabling the assessment of current advancements in Stable Diffusion algorithms. With its different modules, it currently allows large pipelines to generate different datasets used for evaluation. The created datasets are COCO annotated regarding their bounding boxes, with segmentation labeling and depth information available. All of these datasets are available for download and testing.

The findings from the conducted tests suggest that while ALDM and the associated post-processing algorithms have potential, they are not yet accurate enough to fully replace synthetic data. A key challenge lies in the distortion of generated objects, particularly in their proportions, which remains problematic even after depth information is applied during post-processing.

The experiments underline the importance of training, with MUAD showing superior Average Precision (AP) and Average Recall (AR) scores compared to Synthehicle, likely due to ALDM's training on the Cityscapes dataset. An additional insight is that depth information combined with segmentation enhances the quality of a previously generated image and, therefore, consistently outperforms segmentation-only approaches. This suggests that modifying ALDM to accept both segmentation and depth information as inputs could significantly optimize the image quality.

Looking at the other tests, ALDM demonstrates strong temporal and spatial consistency, maintaining stable backgrounds and object continuity across frames, especially with stationary cameras. However, distorted car contours remain unresolved, indicating a need for further refinement.

6.2 Future Work

As shown in the conclusion, there are still a lot of topics that have to be investigated further. The future work topics can be split into two categories: the programming and the evaluation.

6.2.1 Programming Part

The current pipeline supports most of the features used in the evaluation part, especially on the generational part. However, some metrics, especially the spatial-temporal image consistency parts, only exist as code fragments that can be adapted into modules. Overall, in the current space of LDMs, there is so much going on, leading to new technical findings that could also be adapted into more modules.

Also, user experience is currently limited. Configuration must be done in code, and the functional programming style is intuitive for computer science students but probably not for most others. A user interface a la ComfyUI would be a good thesis topic, especially addressing the security problems mentioned above. Also, performance is a significant point, especially parallelism, which could be used with enough computing power.

Moving away from my pipeline, enhancing ALDM could be another interesting topic. As mentioned above, expanding it with the input of depth maps or making it possible to infer information from previously generated images could be really interesting, resulting probably in results close to or even succeeding pure synthetic data.

6.2.2 Evaluation Part

While the current evaluation methods and settings are chosen by carefully analyzing different smaller samples, there are so many configuration options that it is not practical to explore all possible combinations. More tests can be quickly adapted to the existing pipelines, potentially yielding different results from those presented here. One option could be to use Stable Diffusion XL [PEL+23] instead of Stable Diffusion 1.5, which is often regarded as a superior diffusion model. New ControlNets have been released recently, enabling their integration into the pipeline as the image-to-image base. Additionally, training custom models could significantly improve some models' performance and is worth exploring, especially for the ALDM module, but also for addressing the temporal-spatial consistency problem discussed above.

Furthermore, other evaluation methods could be employed, particularly those with a greater focus on the image quality of the outputs from the different models. Examples of metrics could be taken from section 3.6.

Bibliography

[BHC15] Vijay Badrinarayanan, Ankur Handa, and Roberto Cipolla. Segnet: A deep convolutional encoder-decoder architecture for robust semantic pixel-wise labelling, 2015.

[BMR^{+}20] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners, 2020.

[CGCB14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling, 2014.

[COR^{+}16] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Rehfeld, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset for semantic urban scene understanding, 2016.

[Die22] Sander Dieleman. Guidance: a cheat code for diffusion models, 2022.

[DRC^{+}17] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. Carla: An open urban driving simulator, 2017.

[EKB^{+}24] Patrick Esser, Sumith Kulal, Andreas Blattmann, Rahim Entezari, Jonas Müller, Harry Saini, Yam Levi, Dominik Lorenz, Axel Sauer, Frederic Boesel, Dustin Podell, Tim Dockhorn, Zion English, Kyle Lacey, Alex Goodwin, Yannik Marek, and Robin Rombach. Scaling rectified flow transformers for high-resolution image synthesis, 2024.

[ENO+21] Nelson Elhage, Neel Nanda, Catherine Olsson, Tom Henighan, Nicholas Joseph, Ben Mann, Amanda Askell, Yuntao Bai, Anna Chen, Tom Conerly, Nova DasSarma, Dawn Drain, Deep Ganguli, Zac Hatfield-Dodds, Danny Hernandez, Andy Jones, Jackson Kernion, Liane Lovitt, Kamal Ndousse, Dario Amodei, Tom Brown, Jack Clark, Jared Kaplan, Sam McCandlish, and Chris Olah. A mathematical framework for transformer circuits. *Transformer Circuits Thread*, 2021. https://transformer-circuits.pub/2021/framework/index.html.

[FYB+22] Gianni Franchi, Xuanlong Yu, Andrei Bursuc, Angel Tena, Rémi Kazmierczak, Séverine Dubuisson, Emanuel Aldea, and David Filliat. Muad: Multiple uncertainties for autonomous driving, a benchmark for multiple uncertainty types and tasks. In *33rd British Machine Vision Conference 2022, BMVC 2022, London, UK, November 21-24, 2022*. BMVA Press, 2022.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[GK20] Hossein Gholamalinezhad and Hossein Khosravi. Pooling methods in deep neural networks, a review, 2020.

[GLW+21] Zheng Ge, Songtao Liu, Feng Wang, Zeming Li, and Jian Sun. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430*, 2021.

[GPAM+14] Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks, 2014.

[GS24] 3blue1brown Grant Sanderson. Playlist: Neural networks, 2018/2024.

[HCT+23] Fabian Herzog, Junpeng Chen, Torben Teepe, Johannes Gilg, Stefan Hörmann, and Gerhard Rigoll. Synthehicle: Multi-vehicle multi-camera tracking in virtual cities. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision (WACV) Workshops*, pages 1–11, January 2023.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[HS22] Jonathan Ho and Tim Salimans. Classifier-free diffusion guidance, 2022.

[KALL18] Tero Karras, Timo Aila, Samuli Laine, and Jaakko Lehtinen. Progressive growing of gans for improved quality, stability, and variation, 2018.

[KW22] Diederik P Kingma and Max Welling. Auto-encoding variational bayes, 2022.

[LKKV14] David Lazer, Ryan Kennedy, Gary King, and Alessandro Vespignani. The parable of google flu: Traps in big data analysis. *Science*, 343(6176):1203–1205, March 2014.

[LKZK24] Yumeng Li, Margret Keuper, Dan Zhang, and Anna Khoreva. Adversarial supervision makes layout-to-image diffusion models thrive. In *The Twelfth International Conference on Learning Representations*, 2024.

[LMB$^+$15] Tsung-Yi Lin, Michael Maire, Serge Belongie, Lubomir Bourdev, Ross Girshick, James Hays, Pietro Perona, Deva Ramanan, C. Lawrence Zitnick, and Piotr Dollár. Microsoft coco: Common objects in context, 2015.

[LSD15] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation, 2015.

[LZL$^+$24] Yixin Liu, Kai Zhang, Yuan Li, Zhiling Yan, Chujie Gao, Ruoxi Chen, Zhengqing Yuan, Yue Huang, Hanchi Sun, Jianfeng Gao, Lifang He, and Lichao Sun. Sora: A review on background, technology, limitations, and opportunities of large vision models, 2024.

[ON15] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks, 2015.

[PEL$^+$23] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. Sdxl: Improving latent diffusion models for high-resolution image synthesis, 2023.

[PX23] William Peebles and Saining Xie. Scalable diffusion models with transformers, 2023.

[Ras16] T. Rashid. *Make Your Own Neural Network: Tariq Rashid*. CreateSpace Independent Publishing Platform, 2016.

[RBL$^+$22] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022.

[RDGF16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection, 2016.

[RFB15] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation, 2015.

[RKHD24] Dillon Reis, Jordan Kupec, Jacqueline Hong, and Ahmad Daoudi. Real-time flying object detection with yolov8, 2024.

[SSZ$^+$21] Vadim Sushko, Edgar Schönfeld, Dan Zhang, Juergen Gall, Bernt Schiele, and Anna Khoreva. You only need adversarial supervision for semantic image synthesis, 2021.

[UPG$^+$24] Vishaal Udandarao, Ameya Prabhu, Adhiraj Ghosh, Yash Sharma, Philip H. S. Torr, Adel Bibi, Samuel Albanie, and Matthias Bethge. No "zero-shot" without exponential data: Pretraining concept frequency determines multimodal model performance, 2024.

[vdOKK16] Aaron van den Oord, Nal Kalchbrenner, and Koray Kavukcuoglu. Pixel recurrent neural networks, 2016.

[VSP$^+$23] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2023.

[Wat20] Alexander Watson. Deep learning techniques for super-resolution in video games, 2020.

[WLD$^+$19] Shuang Wu, Guoqi Li, Lei Deng, Liu Liu, Dong Wu, Yuan Xie, and Luping Shi. $l1$ -norm batch normalization for efficient training of deep neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 30(7):2043–2051, July 2019.

[WLZ$^+$18] Ting-Chun Wang, Ming-Yu Liu, Jun-Yan Zhu, Andrew Tao, Jan Kautz, and Bryan Catanzaro. High-resolution image synthesis and semantic manipulation with conditional gans, 2018.

[XLZ$^+$18] Tete Xiao, Yingcheng Liu, Bolei Zhou, Yuning Jiang, and Jian Sun. Unified perceptual parsing for scene understanding, 2018.

[ZBSL17] Tinghui Zhou, Matthew Brown, Noah Snavely, and David G. Lowe. Unsupervised learning of depth and ego-motion from video, 2017.

[ZLLS23] Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola. *Dive into Deep Learning*. Cambridge University Press, 2023. `https://D2L.ai`.

[ZRA23] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. Adding conditional control to text-to-image diffusion models, 2023.

[ZSJ$^+$22] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Fucheng Weng, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. Bytetrack: Multi-object tracking by associating every detection box, 2022.

[ZWBR$^+$24] Haonan Zhao, Yiting Wang, Thomas Bashford-Rogers, Valentina Donzella, and Kurt Debattista. Exploring generative ai for sim2real in driving data synthesis, 2024.

[ZZP$^+$19] Bolei Zhou, Hang Zhao, Xavier Puig, Tete Xiao, Sanja Fidler, Adela Barriuso, and Antonio Torralba. Semantic understanding of scenes through the ade20k dataset. *International Journal of Computer Vision*, 127(3):302–321, 2019.