Master of Science HES-SO in Engineering
Av. de Provence 6
CH-1007 Lausanne

Master of Science HES-SO in Engineering
Major: Information and Communication Technologies

GANime

Video generation of anime content conditioned on two frames

Under the supervision of
Prof. Dr. Stefano Carrino
At Haute Ecole Arc, Switzerland

Realised by
Farid Abdalla

Supported by Osaka Prefecture University
At Osaka, Japan
Under the supervision of Prof. Motoi Iwata

Osaka, Osaka Prefecture University, 2022

Acknowledgments

I would first like to thank Prof. Koichi Kise of Osaka Prefecture University who invited me as a research student allowing me to fulfill my dream to study in Japan.

Then my thesis advisor, Prof. Stefano Carrino of the Haute Ecole Arc for his guidance and availability throughout the work.

Also Prof. Motoi Iwata of Osaka Prefecture University who welcomed me at his lab and provided me sound advice.

My friend and ex-schoolmate Ludovic Herbelin from the Haute Ecole Arc with whom I was able to discuss ideas and new concept in the image/video generation field.

And finally, my friends in Switzerland and Japan who supported me during my work and allowed me to spend quality time when not working on my thesis.

Thank you.
Farid Abdalla

Abstract

Video synthesis shows little progress compared to image synthesis. Improvements in this field could be beneficial to the anime industry where content creation is time consuming and expensive. In this paper, we present a method that builds on VQ-GAN and GPT-2 transformer to generate videos frame by frame. These videos are conditioned on two frames, the first and the last frame, allowing better control on the generated sequence. The proposed model trained on a custom dataset made from scenes of the Kimetsu no Yaiba anime is able to generate videos with good temporal consistency from the first to the last frame.

The code of the project and videos are available as a GitHub project at the following address: https://github.com/Kurokabe/GANime.

Contents

List of Figures

List of Tables

1 Introduction

Image generation has tremendously improved in the recent years. Mostly thanks to Generative Adversarial Networks (GANs) [1] it is now possible to generate realistic pictures that would fool a human. A famous example is the StyleGAN [2] model from Nvidia which is able to generate realistic faces (example visible on the site https://this-person-does-not-exist.com). Methods other than GAN are available for generative models.

Diffusion models [4, 5, 6] work by successively adding Gaussian noise to the training data, then learns to recover the data by denoising the image. Random noise can then be used to generate new samples after the training.

Discrete representation learning [8, 9, 7] first learns discrete representation of the data (by learning to represent images as a sequence of codebook indices), then uses another model to generate codebook indices which can be decoded into a new image.

Generation can either be unconditional or conditional. Unconditional generation means that data will be generated "randomly", whereas conditional generation will use existing data (label, image, text) to generate novel data. Some examples of conditional generation are CGAN [10] where a label is used in addition to the noise, or the DALL-E model where an image is generated based on a textual description [7, 6]

Advances in the video generation field however are relatively slower. This is due to the increased complexity of the problem. In addition to be space coherent as in image generation, video generation needs to be time coherent. With the added dimension, it is also more computationally demanding.

Video generation differs from frame interpolation by the number of generated frame, which is generally lower. One use case of frame interpolation is to increase the number of frame per second (FPS). When generating the intermediate frame between each frame of a video, the number of frame per second is doubled. This problem can be seen as rather simple, since the middle frame is rather similar to the two adjacent frames. However some problems can arise, such as a blurry result [11, 12, 13].

Most image conditioned video generation are based on a single frame. [15, 14, 16] are some examples of image conditioned video generation. However, the problem which can occur is that the model does not have any information on how to animate the image. If the image represents a person standing, the person could be animated by running, jumping, waving his arms and so on.

Point-to-Point Video Generation[17] proposes to condition a video generation by giving not only the first frame but also the last frame of the sequence. The number of frames to generate in-between is also given to the model. The model is aware of the remaining number of frames to create and can adapt the generation accordingly.

In the manga or anime field, most of the work is based on image generation or editing (see *Curated list of papers on anime or manga* [18]). Some try to replicate a character pose on an anime body or face [19, 20], but they only take into account the character (no apparent background). The ones that generate video images based on the full content (characters + background) tend to generate only a couple of frames via interpolation / inbetweening [21, 22].

Creating anime content is time consuming and expensive. As an example, it costed ¥ 9-10 million ($ ∼70'000) to create one episode of the Kimetsu no Yaiba anime and ¥2 billion ($ ∼15 million) for the movie [23]. In terms of time, a slow paced anime is composed of around 2'000 drawn frames and a fast paced anime can go up to 10'000 drawn frames [24]. It takes around 9 months to create a 12 episode anime [25]. Although expensive, anime are profitable. For instance, the Kimetsu no Yaiba movie grossed around $500 million. Also, a famous anime helps selling the manga or derived products: the Kimetsu no Yaiba franchise generated ¥ 1 trillion ($ ∼7 billion) in estimated revenue sales in 2020.

Having a method to reduce the number of drawn frames could be beneficial for the anime industry. The existing methods are able to increase the number of FPS by generating a few in-between frames (1∼ 5 frames) via interpolation techniques. Using the capabilities of image generation in this field would allow a greater number of generated frames thus saving a lot of time and money for the anime producing companies.

2 Related work

Before diving into the proposed method, it is important first to explain some related work to better understand the continuity behind the resulting work.

2.1 Autoencoders

An autoencoder is a model composed of an encoder and decoder. The encoder learns to represent the data into what is called the latent space. The decoder learns to reconstruct the original data based on this space as seen on Fig. 1.

The modeling and the training are very simple. In the case of image processing, the encoder is often composed of multiple convolutions layer reducing the number of dimensions of the image. The decoder is then composed of the opposite transformations (deconvolution or up-sampling followed by a convolution, more details on the difference on Appendix B) to get back the original dimensions.

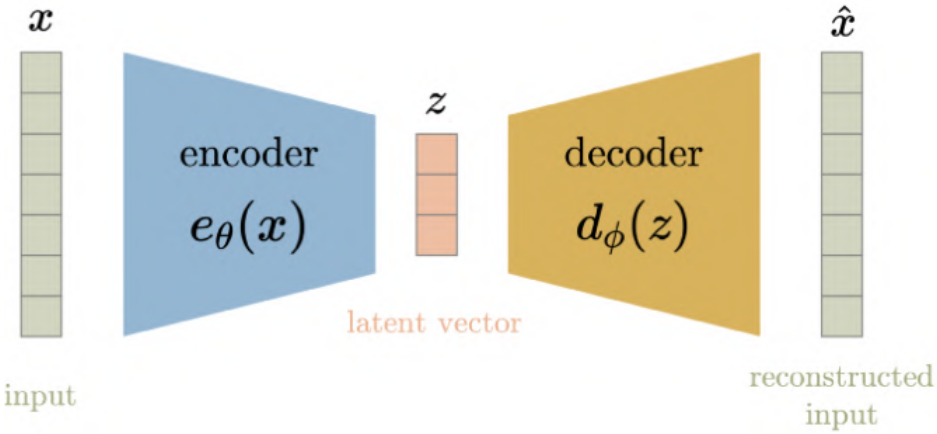


Figure 1: Architecture of an autoencoder (by Anwaar)

As for the training, the input of the model is some data (e.g. an image), and the target is the same data. The loss is the difference between the real data and the reconstructed data.

The loss function can be the Mean Squared Error (MSE):

$$loss = ||x - \hat{x}||_2 = ||x - d_\phi(z)||_2 = ||x - d_\phi(e_\theta(x))||_2 \tag{1}$$

Figure 2: Difference between a regular and irregular latent space (by Rocca)

With an autoencoder, it is possible to sample from the latent space to generate novel data. However, the problem with autoencoder is that the latent space is not regular. That means that random points inside the latent space does not necessarily have meaningful results. Fig. 2 shows the difference between a regular and irregular latent space.

2.2 Variational autoencoders (VAE)

A variational autoencoder aims to regularize the latent space. The first major difference with an autoencoder is that the discrete encoding of the latent space is replaced by a continuous one, i.e. a distribution composed of the mean and variance. Fig. 3 shows the architecture of the VAE.

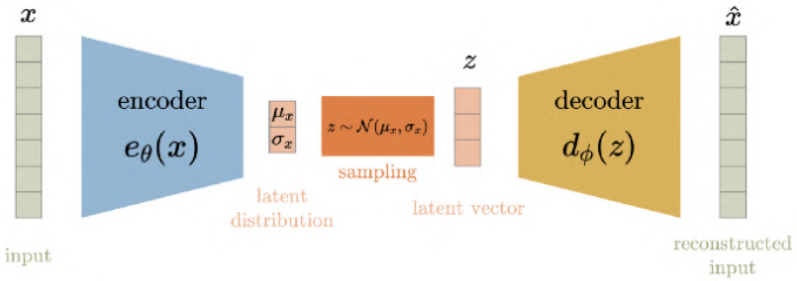


Figure 3: Architecture of a VAE (by Anwaar)

The second difference is that during the training we encourage the model to produce a latent space where each points are close to each other and that the distributions overlap, as described in Fig. 4. This is done by adding a similarity loss component, the Kullback-Leibler divergence between the latent space distribution and the standard gaussian (zero mean, one variance).

The loss function becomes:

$$\mathcal{L}_{reconstruction} = ||x - \hat{x}||_2 = ||x - d_\phi(z)||_2 = ||x - d_\phi(\mu_x + \sigma_x \epsilon)||_2$$

$$\mu_x, \sigma_x = e_\theta(x), \epsilon \sim \mathcal{N}(0, 1)$$

$$\mathcal{L}_{similarity} = KL\ Divergence = D_{KL}(\mathcal{N}(\mu_x, \theta_x)\ ||\ \mathcal{N}(0, 1))$$

$$loss = \mathcal{L}_{reconstruction} + \mathcal{L}_{similarity} \tag{2}$$

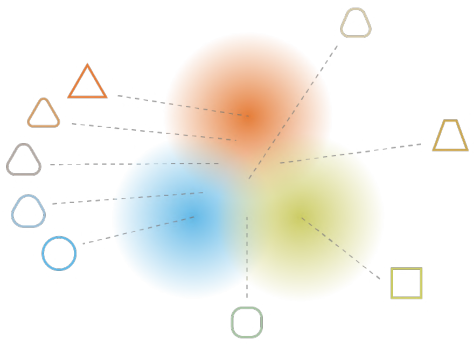


Figure 4: Regularized latent space where a point sampled from the space has a meaning relative to the other points (by Rocca)

2.3 Vector Quantized Variational Autoencoder (VQ-VAE)

Since most of the data available in the real world is discrete, e.g. an image can be thought as a list of discrete objects (car, dog, house, etc.) with discrete characteristics (shape, color, size, color, etc.), the continuous latent representation from VAE is replaced by a discrete one. Also, transformers work with discrete data, which will be a benefit later.

The main idea behind VQ-VAE [8] is that a discrete codebook is learned. The codebook is a list of vectors with their corresponding indices, see Fig. 5. Each vector of the encoder output is compared with all vectors from the codebook. The most similar vector (closest in terms of euclidean distance) will be selected as the quantized representation for the decoder.

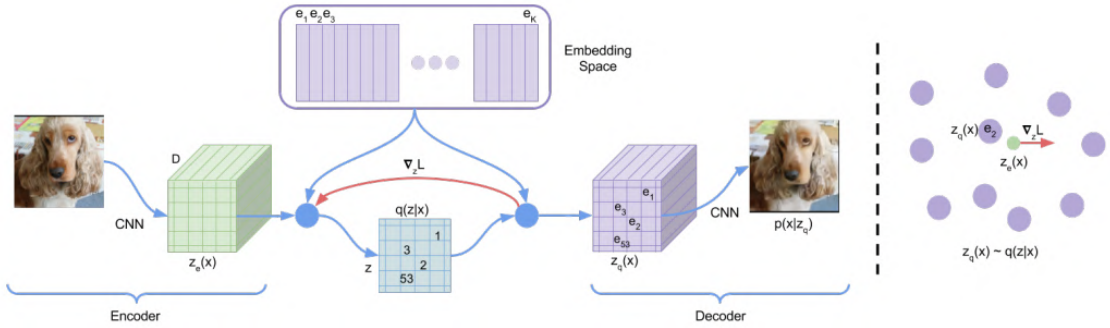


Figure 5: Architecture of the VQ-VAE (by Oord et al.)

During backpropagation, the gradient $\nabla_z L$ is transfered unaltered to the encoder. This gradient contains useful information on how the encoder needs to change its output to lower the reconstruction loss as seen on the right part of Fig. 5.

Loss Put more formally, there is an encoder E, a decoder G which together learn a discrete codebook $Z = \{z_k\}_{k=1}^{K} \subset \mathbb{R}^{n_z}$ where n_z is the dimensionality of codes. The image x is approximated to $\tilde{x} = G(z_q)$. The encoding $\tilde{z} = E(x) \in \mathbb{R}^{h \times w \times n_z}$ and a element-wise quantization $q(\cdot)$ of each spatial code $\tilde{z}_{ij} \in \mathbb{R}^{n_z}$ onto its closest codebook entry z_k is used to obtain z_q [9].

$$z_q = q(\tilde{z}) := \left(\underset{z_k \in \mathcal{Z}}{\operatorname{argmin}} ||\tilde{z}_{ij} - z_k|| \right) \in \mathbb{R}^{h \times w \times n_z} \tag{3}$$

The reconstruction $\tilde{x} \approx x$ is given by

$$\tilde{x} = G(z_q) = G(q(E(x))) \tag{4}$$

Since the quantization operation is not differentiable, backpropagation is done by copying the gradient from the decoder to the encoder (straight-through gradient estimator, as seen on appendix A) so that the model and codebook can be trained end-to-end via the loss function:

$$\mathcal{L}_{VQ}(E, G, \mathcal{Z}) = \underbrace{||x - \tilde{x}||^2}_{\mathcal{L}_{reconstruction}} + \underbrace{||z_q - sg[E(x)]||_2^2}_{\mathcal{L}_{codebook}} + \beta \underbrace{||sg[z_q] - E(x)||_2^2}_{\mathcal{L}_{commitment}} \tag{5}$$

The $\mathcal{L}_{codebook}$ goal is to have the chosen codebook vector as close as possible to the encoder output. The stop-gradient operation $sg[\cdot]$ is set to only update the codebook. The $\mathcal{L}_{commitment}$ purpose is

similar to the $\mathcal{L}_{codebook}$ but with the stop-gradient on the codebook vector in order to commit the encoder output as much as possible to its closest codebook vector.

The β parameter is often set to 0.25 according to the VQ-VAE paper [8].

More details about the internal structure of VQ-VAE can be found in the Neural Discrete Representation Learning paper [8] and in the Keras code sample [28].

2.4 Taming transformers for high-resolution image synthesis (VQ-GAN)

The self-attention mechanism found in transformers [29] produces good results but is computationally expensive. To be able to use it for image synthesis, the images need to be expressed as sequences. However, representing an image as a sequence of pixels or patches would require too much processing power.

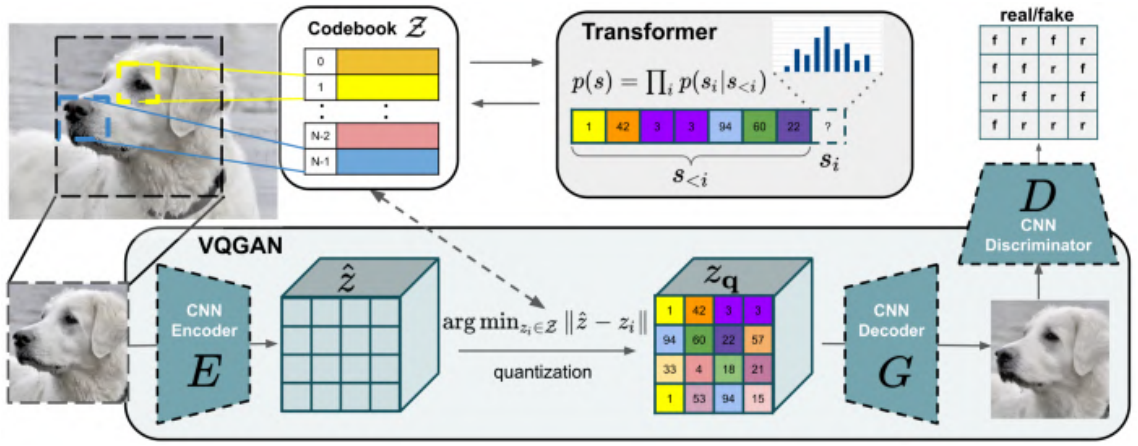


Figure 6: Architecture of the VQ-GAN (by Esser et al.)

VQ-GAN is an extension of VQ-VAE and uses the same idea of codebook encoding to represent the encoder output. Exactly like in VQ-VAE, the output of the encoder $\tilde{Z}$ is quantized to Z_q by using the closest codebook entry. This enables the image to be represented as a sequences of token as shown in Fig. 6 where the image of the dog could be represented as the sequence [1, 42, 3, 3, 94, 60, 22, 57, ...].

VQ-GAN changes the codebook representation learning in two ways compared to VQ-VAE. First, a **perceptual loss** [30] is added to the reconstruction loss. Then an **adversarial loss** as used in GANs [1] is added to sharpen the generated images (VQ-VAE results tend to be blurry).

Perceptual loss In addition of calculating the error pixel-wise between two images with the Mean Absolute Error, both images are given to a pretrained network, like VGG16, and high level features are used to compare the images as described on Fig. 7.

This allows two images that are visually similar, in terms of content, or same content but shifted by some pixels, to have a low loss, which would not be the case with the MSE and it produces visually more appealing results.

Adversarial loss In a Generative Adversarial Network (GAN), there is a generator and a discriminator. The generator will try to generate realistic pictures whereas the discriminator will distinguish real pictures from fake ones. They compete in a min-max game where the discrimin-

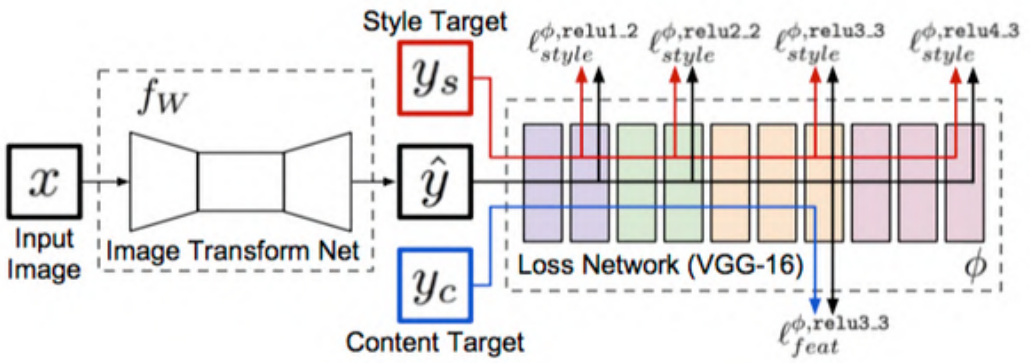


Figure 7: Overview of the perceptual loss model (by Johnson et al.)

ator will try to minimize its loss and become good at distinguishing real from fake pictures in contrast to the generator who will try to maximize the discriminator loss by creating realistic images that will fool the discriminator.

In the case of VQ-GAN, the prediction real/fake is not applied to the entire image, but to single patch, see top right of Fig. 6, by using a patchGAN architecture [31].

The adversarial loss can be expressed as follows, where D is the Discriminator:

$$\mathcal{L}_{GAN}(\{E, G, \mathcal{Z}\}, D) = [logD(x) + log(1 - D(\tilde{x}))] \tag{6}$$

The adversarial loss $\mathcal{L}_{GAN}(\{E, G, \mathcal{Z}\}, D)$ is scaled by an adaptative weight λ to balance the reconstruction and adversarial loss [9, 32, 33] :

$$\lambda = \frac{\nabla_{G_L}[\mathcal{L}_{reconstruction}]}{\nabla_{G_L}[\mathcal{L}_{GAN}] + \delta} \tag{7}$$

Where $\nabla_{G_L}[\cdot]$ is the gradient of its input with respect to the last layer L of the decoder, $\delta = 10^{-6}$ for numerical stability and $\mathcal{L}_{reconstruction}$ is the reconstruction (MAE + perceptual) loss.

Loss The complete VQ-GAN loss for the first stage is the following:

$$\mathcal{L}_{VQGAN} = \underset{E,G,\mathcal{Z}}{arg\,min}\ \underset{D}{max}\ \mathbb{E}_{x\sim p(x)}\left[\mathcal{L}_{VQ}(E, G, \mathcal{Z}) + \lambda\mathcal{L}_{GAN}(\{E, G, \mathcal{Z}\}, D)\right] \tag{8}$$

Two stage approach After the first stage of the training, the model has learned a codebook and is able to encode an image x into a sequence of indices Z_q and decode the indices Z_q into a realistic image $\tilde{x}$. Now, if instead of feeding to the decoder the sequence [1, 42, 3, 3, 94, 60, 22, 57, ...] (which corresponds to the picture of the dog in Fig. 6 we give the sequence [**5**, 42, 3, 3, 94, 60, 22, 57, ...], the resulting picture might add sunglasses on the dog's head if the index 5 corresponds to sunglasses.

For the second part of the training, a transformer model (GPT-2) is used to predict the next token with a cross-entropy loss. When given the input [1] it should predict [42]. When given [1, 42] it should predict [3] and so on as shown on Fig. 8.

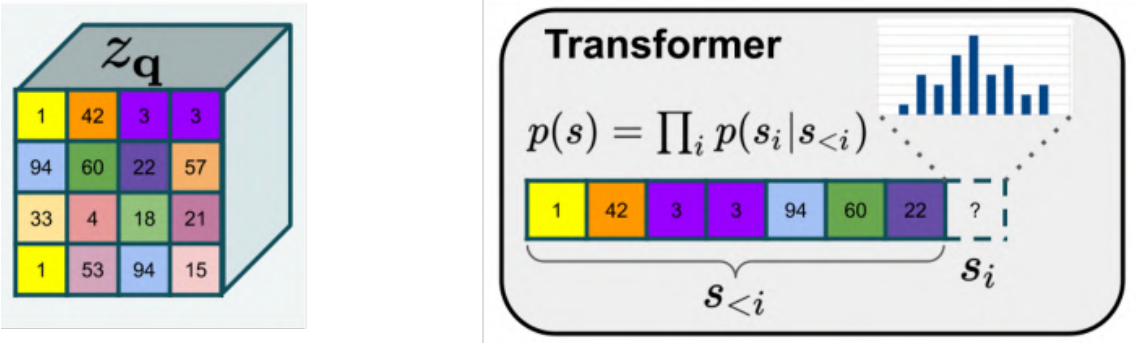


Figure 8: Transformer part of the VQ-GAN (by Esser et al.)

2.5 VideoGPT

VideoGPT is similar to VQ-GAN but for video generation. A 3D VQ-VAE is used for the first stage and a GPT-like transformer for the second stage [15]. It can generate conditional videos based on a class label or a starting image. However the problem when synthesizing a video based on a single frame is that the model has no indication on the movement. Many animations could be interpreted. For example, when giving the first image on the first row on Fig. 9, the resulting animation could be a jump or a waving.

Figure 9: Example of generation with VideoGPT (by Yan et al.)

Another thing that could be improved with VideoGPT is that the number of generated frames is not customizable. The 3D VQ-VAE can encode a video into a sequence of N indices times M frames then decode a sequence of indices to generate a video. However, the number of frames seems to be limited to 16 frames.

2.6 Point-to-Point Video Generation

The Point-to-Point Video Generation model creates video based on two control points, the start and end frame of a sequence. The model encode the frames into a latent space with a Variational Auto Encoder (VAE), uses Long Short Term Memory (LSTM) layers for temporal consistency and keep track of the number of remaining frames to generate (Time Counter). It uses multiple losses such as the Kullback-Leibler divergence between the prior and posterior, a reconstruction loss between the generated frame and the ground-truth and an alignment loss between the generated and target frame.

Compared to other models [15, 34] where the video is generated in one go with 3D convolutions, this model generates video frame by frame which allows better control on the number of generated frames.

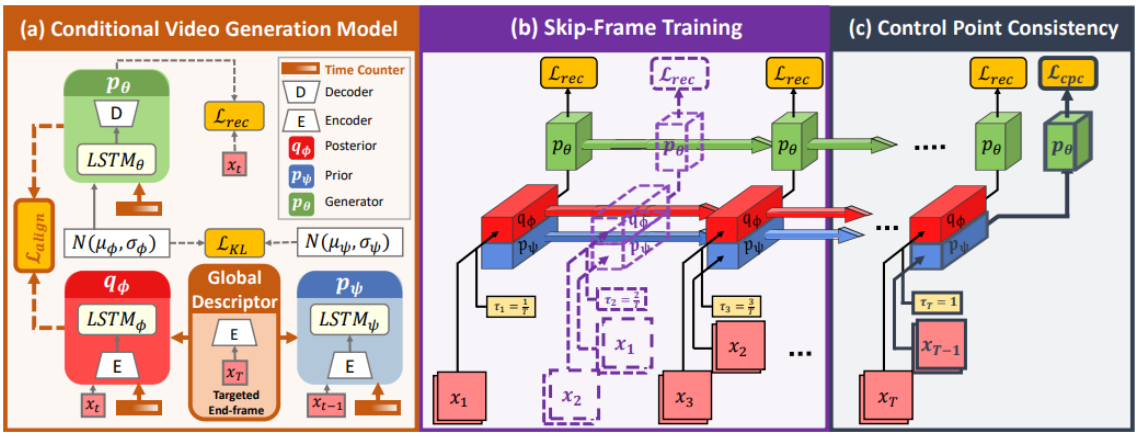


Figure 10: Architecture of the Point-to-Point Video Generation model (by Wang et al.)

2.7 Chosen method

Two approaches were possible to begin the project. Start from the VideoGPT and add the VQ-GAN improvements or start from the VQ-GAN and add the temporal dimension to generate videos. After looking at both source code, the VQ-GAN seemed simpler. Also, the generated video length with VideoGPT is not customizable, which would be a problem with the scene length disparity of an anime dataset. Those two reasons lead to start from VQ-GAN and generate videos frame by frame similar to the Point-to-Point video generation model.

3 Approach

The video generation is done in a two stage manner. First, a model is trained to auto-encode an image dataset in order to learn an effective codebook of image constituents. Then a transformer model is used to generate the next frame in the video sequence based on the codebook indices of the previous frames.

Section 3.1 presents the different datasets used for the experiments and how they were created. Section 3.2 presents the VQ-GAN model used to generate codebook indices based on an image (and vice-versa). Section 3.3 explains how a conditional model is trained to generate a video based on the first and last frame.

3.1 Datasets

During the experiments, two types of dataset have been tested. The first one is named Moving MNIST and is composed of white digit numbers on a black background. The second one is created from the Kimetsu no Yaiba anime (abbreviated into KNY). Each dataset type contains both an image and video subset. The image subset is used to train the VQ-GAN and the video subset is used to train the GPT-2 Transformer. All datasets are converted in TFRecord format to benefit from TFDataset performance when training with Tensorflow [35].

3.1.1 Image dataset

Pre-processing The images are saved in the [0, 255] range with uint8 type to save memory, however before being fed to the model they are normalized in the [-1, 1] range and random jitter is applied to artificially increase data variety in the KNY dataset. An image is first scaled from 64x128 to 72x142 then a random portion of the image is taken to have a 64x128 image. There is also a 50% chance that the image is horizontally flipped.

Moving MNIST The image dataset for Moving MNIST is made by simply taking all individual images from the Moving MNIST video dataset (see Section 3.1.2). The resulting dataset consists of 10'000 x 20 = 200'000 images of 64x64 pixels.

Kimetsu no Yaiba During the early tests, only the first episode of KNY was used. That would allow to have a smaller dataset which would be faster to train with. Later on, the 26 episodes of season 1, the 11 episodes of season 2 and the movie split into 7 episodes were used. That would consists of 44 episodes of around 24 minutes with about 34′000 frames per episode. In total, there is about 1056 minutes of video for almost 1.5 million frames. However, when looking at the frames, we can see that multiple same frames are used consecutively. The number of same frames can vary depending on the action.

Depending on the pace of the scene, consecutive frames will be more similar if it is slower, respectively different if faster. Fig. 11 shows the difference between a slow-paced and fast-paced scene. Fig. 11a shows a slow talking scene where an image is used 4-6 times in a row (except for the falling snow which is different on every image) to reduce the number of animation that needs to be done. Fig. 11b shows a fast running scene where the same frame is used only once or twice consecutively.

(a) Slow movement (b) Fast movement

Figure 11: Difference in terms of image duplicate between a slow-paced and fast-paced scene

When creating the image dataset based on all episodes, using all individual frames would have generated a large dataset with many duplicates. The total number of frames would be 44 episodes times 34'000 frames= 1'496'000 frames. To reduce the number and still have a fairly amount of content, only 1 frame over 5 was taken which resulted in around 300'000 individual frames.

The frames were resized from 480x848 pixels to 64x128 pixels in order to reduce the problem complexity and the training time. To preserve the real ratio, the size should have been 60x104 pixels, but in order to keep the size simple when down-sampling with the model (size consecutively divided by two), a power of two was chosen.

3.1.2 Video dataset

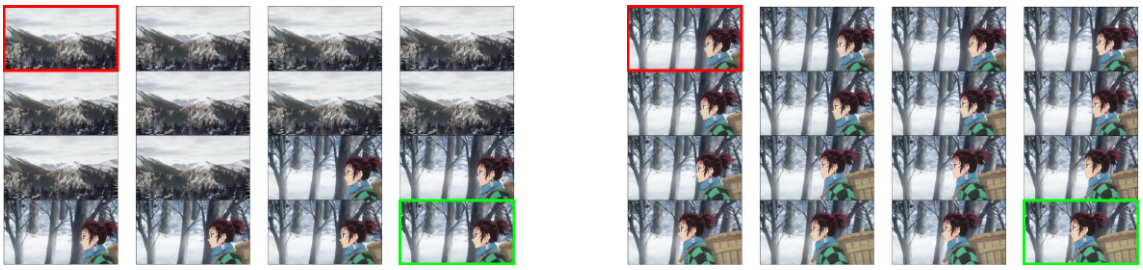
Pre-processing Similarly to the image dataset, videos are saved in the [0, 255] range with the uint8 type but converted to the [-1, 1] range, float32, type before being fed to the model. Other parameters such as the first frame, last frame, number of frames to generate and the number of remaining frames are created based on the video after the normalization.

Moving MNIST The moving MNIST dataset consists of 10'000 video sequences, each composed of 20 frames of 64x64 pixels. The original dataset [36] is of shape [20, 10000, 64, 64] (number of frames, number of samples, height, width). To be consistent with Keras / Tensorflow shapes and to be compatible with data coming from anime, the dataset has been modified to be of shape [10000, 20, 64, 64, 3] (number of samples, number of frames, height, width, channels).

Figure 12: Example of the Moving MNIST dataset (by Srivastava et al.)

Kimetsu no Yaiba

Generating the KNY video dataset is a bit trickier. In order for the model to learn the generation of intermediate frames between two frames, these two frames need to be rather similar. Fig. 13 shows the difference between a bad and good frame selection. When the first and last frame as in Fig. 13b are part of the same sequence, the model can learn how to generate the intermediate frames. However, when the first and last frame are part of two different scenes as shown in 13a, it is difficult for the model to learn the dependency between the two images.



(a) Bad selection of first and last frame

(b) Good selection of first and last frame

Figure 13: On the left, the first (red) and last (green) frame have nothing in common. On the right, the first and last frame are part of the same sequence.

The splitting of episodes into scenes is done with PySceneDetect [38]. It is a command line application which detects shot changes in video and generates multiple clips. Multiple scene detection methods are available:

- **Threshold**: detect changes in average frame intensity / brightness. Good for detecting fade-in / fade-out transitions
- **Content-aware**: detect changes between frames in the HSV color space. Good for detecting cuts between scenes.
- **Adaptive-content** Similar to content-aware, but with a rolling average of adjacent frame changes. Good when there is fast camera motion.

The content-aware and adaptive-content detection have been tried with different set of parameters. The available parameters are the *threshold* and *min-scene-length* for content-aware and adaptive-content and *frame-window* for adaptive-content. It is possible to generate a statistics file with the difference value of two subsequent frames. This allows to tweak the threshold value depending on some missed scenes.

However, it is difficult to decide the best set of parameters to correctly detect all scenes because it depends on the scene length and the elements in the scene. Some scenes are very short as in Fig. 14 where 4 consecutive scenes happen in 4 seconds (around 1 second per scene), whereas on Fig. 15, 3 scenes happen in around 7 seconds.

Some threshold parameters could be good for certain type of scenes but not be suitable for others. A too low threshold would unnecessary split a single scene into multiple sequences, and a too high threshold would not be able to split some scenes correctly.

After some tests on the first episode of KNY to get a better understanding of the parameters, the adaptive-content detection was able to keep quick scenes, but too often fused multiple scenes together. The content-aware detection sometimes dropped very quick scenes, but each scene that was kept (depending on the threshold parameter) was correctly identified as a unique scene.

The command line parameters used to generate the scenes from the different episodes of KNY is shown in Code 1.

Figure 14: Exemple of 4 consecutive scenes happening in a short amount of time

```
scenedetect --input {video_path} --output {output_folder} --downscale 4
--min-scene-len=0.5s --drop-short-scenes detect-content --threshold=15
list-scenes save-images split-video
```

Code 1: PySceneDetect command line used to generate the scenes on the KNY dataset

Where {video_path} is the path of the video that will be cut into scenes and {output_folder} is the path of the folder that will contain the splitted scenes.

All scenes shorter than 0.5 seconds are dropped. Without the drop-short-scenes parameter, shorter scenes are added to other scenes.

This resulted in between 300 to 500 scenes per episodes. However, some of these scenes had a few number of frames (half a second / around 15 frames), and some were longer (10 seconds and more). In order to have a rather uniform dataset without truncating to the shortest video, an ideal amount of frames was set as between 15 and 25 frames. Every video longer than this amount would be split into videos of 15 to 25 frames.

Table 1 shows for a different number of frames how longer scenes are splitted into shorter scenes.

Video length (frames)	Resulting scene lengths (frames)
25	25
29	29
30	15, 15
31	15, 16
32	16, 16
33	16, 17
50	25, 25
51	17, 17, 17
52	17, 17, 18

Table 1: Examples of splitting long scenes into smaller scenes

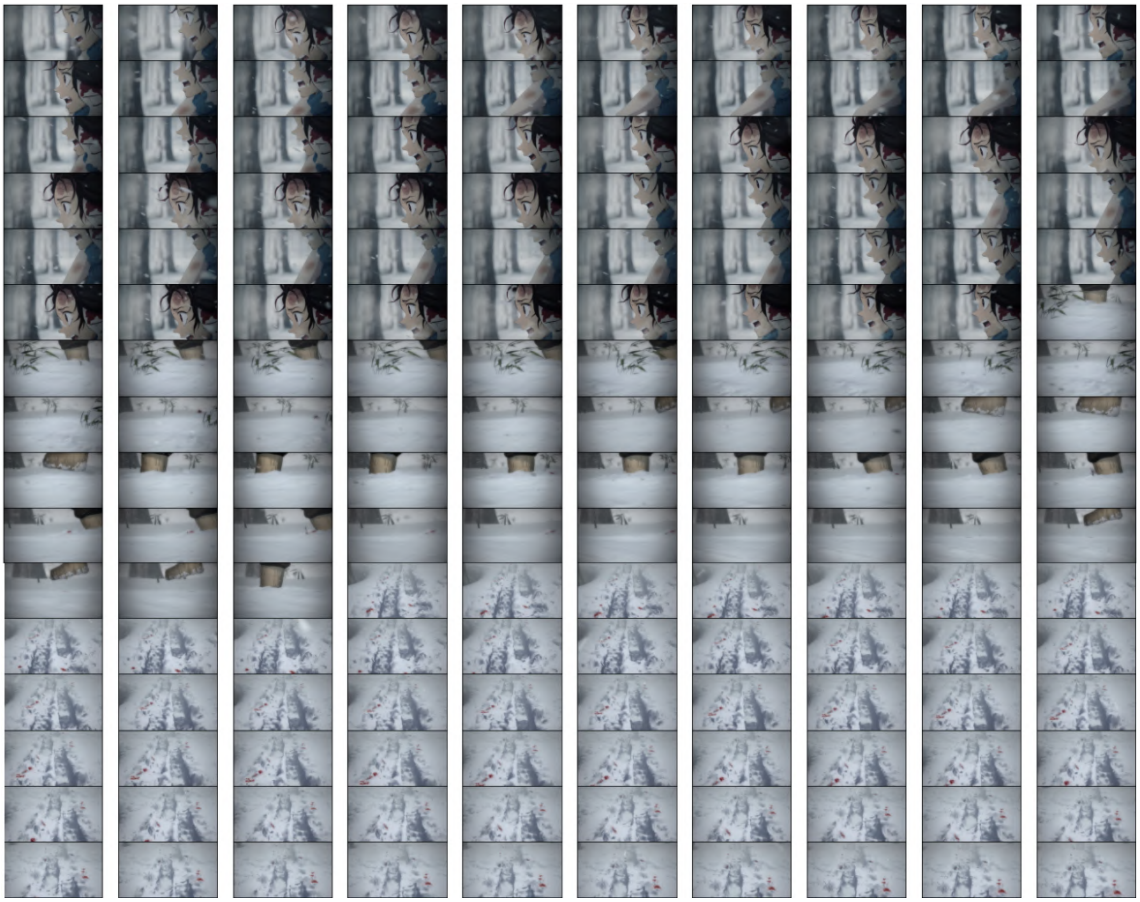


Figure 15: Exemple of 4 consecutive scenes happening in a short amount of time

After splitting all scenes in that manner, the number of videos per episode almost doubled to a total of around 35'000 scenes for the 44 episodes of KNY.

Note: After generating scenes with PySceneDetect, it sometimes happens that the first two frames comes from the previous scene. This is problematic when taking the first and last frame of a scene to train the model, the first frame will not be really correlated with the current scene as seen in Fig. 16.

(a) The first two frames does not belong to the current scene

(b) The first two frames are part of the current scene

Figure 16: Exemple of a bad (left) and good (right) split with PySceneDetect. Sometimes the first two frames come from the previous scene.

To avoid potential problems, the first two frames of each scene are removed when creating the TFRecord dataset (before splitting the video into smaller videos).

Variable number of frames Since the video dataset is not totally uniform, batching multiple videos is not possible without modifications. A simple solution would be to pad the videos with empty frames so that each batch elements has the same number of frames than the longest video. Since the effective number of frame is also known, it is possible when processing empty frames to use the masking property of the model to ignore the computation on these padded frames.

Another method is to reduce the number of frames so that all batch elements have the same number of frames than the shortest video. Since there are less frames to compute, processing a batch is faster than with the masking method. An example of this method is shown on Fig. 17

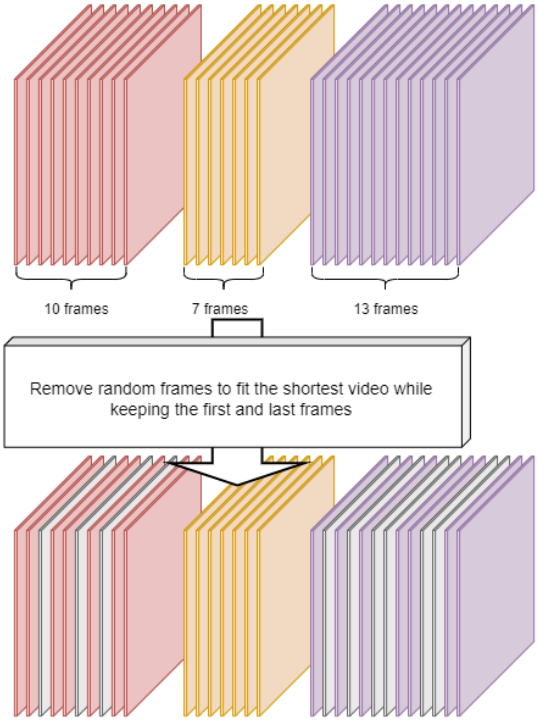


Figure 17: Creation of a batch when truncating to shortest video. The frames selected for removal are in gray.

Both methods have been tested for model training and seem to produce similar results. The truncating method was then preferred since it was faster to train.

3.2 Effective codebook of image constituents

This section details the first stage of the GANime model training in order to generate videos based on the first and last frame. The code was inspired from the PyTorch code of *Taming Transformers for High-Resolution Image Synthesis* [9] available on Github and was adapted in Tensorflow.

The first stage of GANime is identical to the first stage of VQ-GAN to generate a codebook of image constituents. A yaml config file is used to define the VQ-GAN model parameters to be trained on an image dataset. An example of such config file can be found at appendix C.1.

The VQ-GAN is composed of an encoder, vector-quantizer and decoder. Convolution layers are used before and after the vector-quantizer to ensure that the vector-quantizer inputs and outputs have respectively *num_embeddings* and *z_channels* number of channels as shown in figure 18.

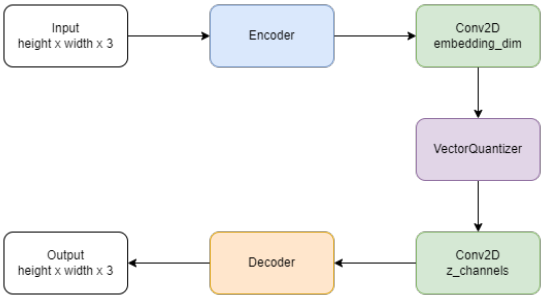


Figure 18: Implementation architecture of the VQ-GAN

The encoder is essentially a sequence of residual, attention and down sampling blocks. These three blocks are applied N times depending on the number of *channels_multiplier* specified in the config file. The channel number of the Nth sequence is equivalent to the previous number of channels multiplied by the current *channels_multiplier*. The downsample block is composed of an average pooling and convolutional layer in order to divide by two the image resolution. The attention block (with a dotted outlined on Fig. 19) is only applied if the current resolution is specified in the *attention_resolution* parameter.

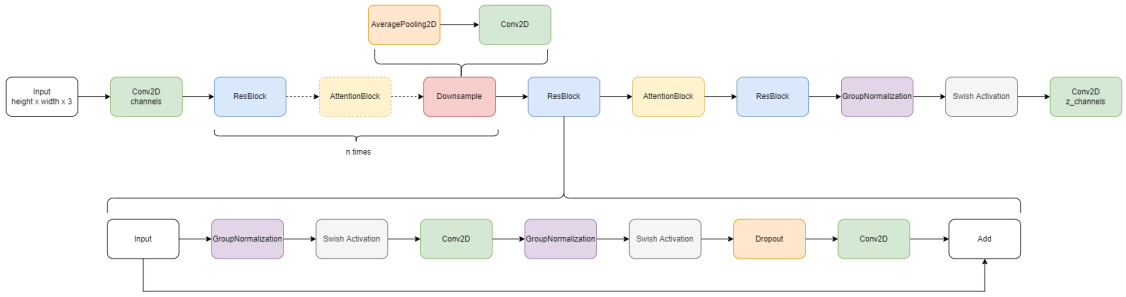


Figure 19: Architecture of the VQ-GAN encoder

Table 2 shows an example of the image shape at each step when the *channels* parameter is set to 128 in the config file, the *channels_multiplier* is set to [1, 2, 4], the *attention_resolution* is [16] and the input image is of shape 64x64x3:

Name	Output shape	Channels Multiplier
Input	(64, 64, 3)	
Conv2D	(64, 64, 128)	
ResBlock	(64, 64, 128)	1
Downsample	(32, 32, 128)	1
ResBlock	(32, 32, 256)	2
Downsample	(16, 16, 256)	2
ResBlock	(16, 16, 1024)	4
Attention	(16, 16, 1024)	4
Downsample	(8, 8, 1024)	4

Table 2: Visualization of the output shape during the first steps of the VQ-GAN encoder

The decoder elements are similar to the encoder but put somewhat in a reversed order. An up-sampling block is used instead of a down-sampling block and a *tanh* function is used at the output of the model. The channels multiplier is also used in reverse to decrease the number of channels.

Notice that the up-sampling layer is followed by a convolutional layer inside the up-sampling block instead of a deconvolutional layer on Fig. 20. This is to solve the checkboard effect, as mentioned in appendix B

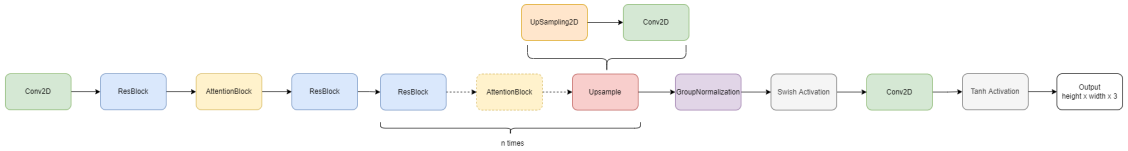


Figure 20: Architecture of the VQ-GAN decoder

The discriminator uses the same architecture as in the PatchGAN [31]. Multiple convolutions are done sequentially to double the number of channels (starting from 64 at the first convolution) while dividing the image resolution by 2 with a stride of 2. The down-sampling block is done N-1 times where N is the model parameter n_layers (usually set to 3). Fig. 21 shows the discriminator architecture and table 3 presents an example of the output shape at each steps when the input image is of size 64x128x3 and the n_layers parameter is set to 3.

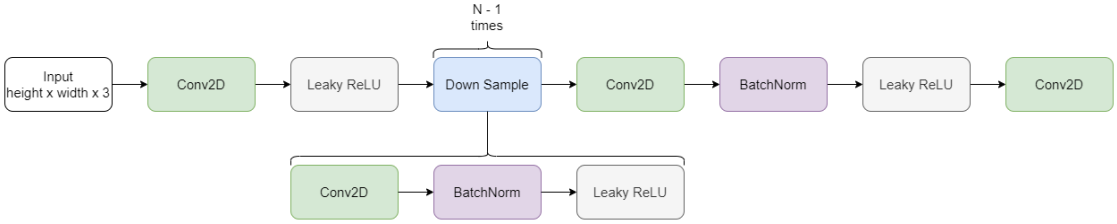


Figure 21: Architecture of the VQ-GAN discriminator

Name	Output shape
Input	(64, 128, 3)
Conv2D	(32, 64, 64)
Downsample 1	(16, 32, 128)
Downsample 2	(8, 16, 256)
Conv2D	(8, 16, 512)
Conv2D	(8, 16, 1)

Table 3: Visualization of the output shape at each steps of the discriminator when n_layers is 3.

Training The VQ-GAN generator is trained as shown on Fig. 22. The reconstructed image is used with the target image to calculate the reconstruction loss which is the sum of the MAE and perceptual loss. The discriminator will predict if patches of the image are real or fake. The cross entropy is calculated between the discriminator prediction and an array full of 1, since all prediction are fake and the generator job is to fool the discriminator. The adaptive weight is used to scale the discriminator loss based on the reconstruction loss (as described in section 2.4). The disc factor is used to activate or not the discriminator loss. It allows to use the discriminator loss after a certain number of steps. When this number of steps is not reached, the discriminator loss is scaled to 0. The total loss corresponds to the sum of the quantization loss, reconstruction loss and discriminator loss. More details and equations can be found in Section 2.3 and 2.4.

The training procedure of the VQ-GAN discriminator is shown on Fig. 23. The goal of the discriminator is to sucessfully distinguinsh real images from fake ones. The discriminator prediction

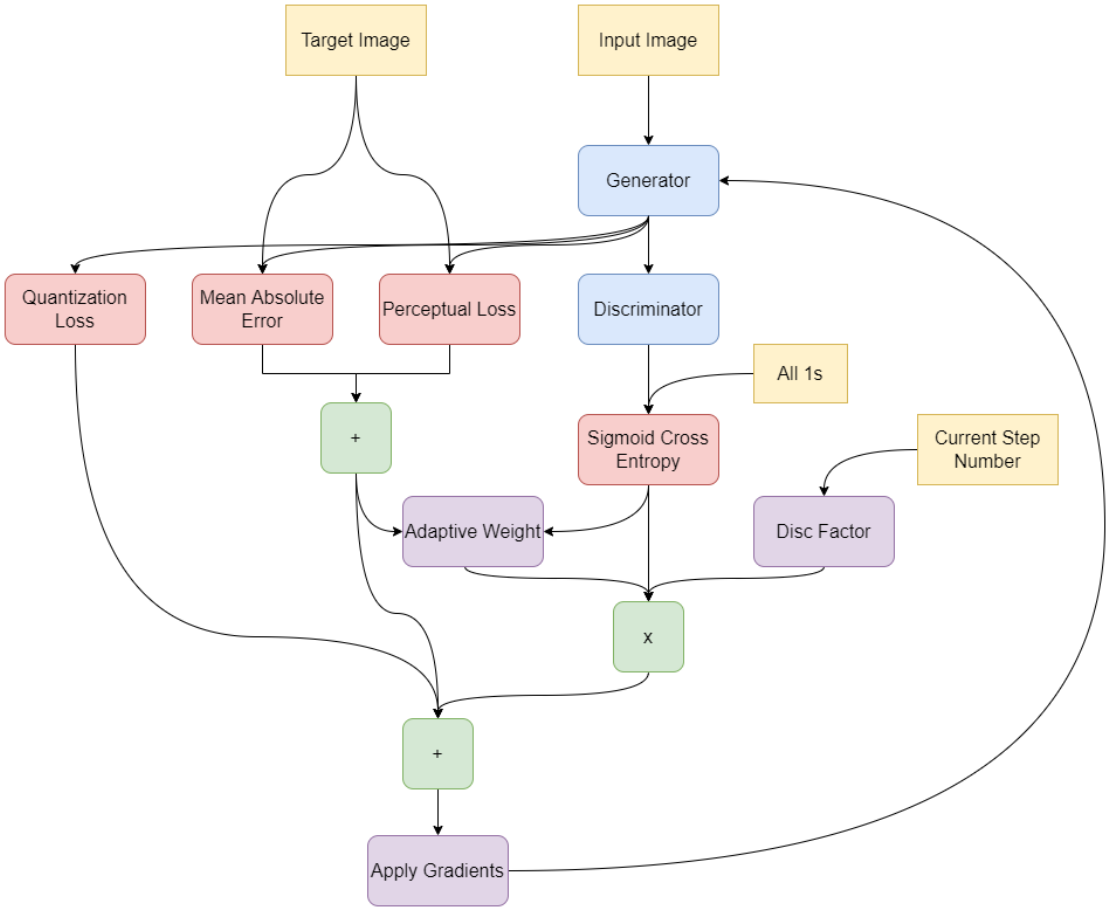


Figure 22: Training procedure of the VQ-GAN generator

is then compared with an array full of 1 for the real pictures and an array full of 0 for the fake pictures. This corresponds to the real and generated loss which are summed then scaled with the disc factor as in the generator training procedure.

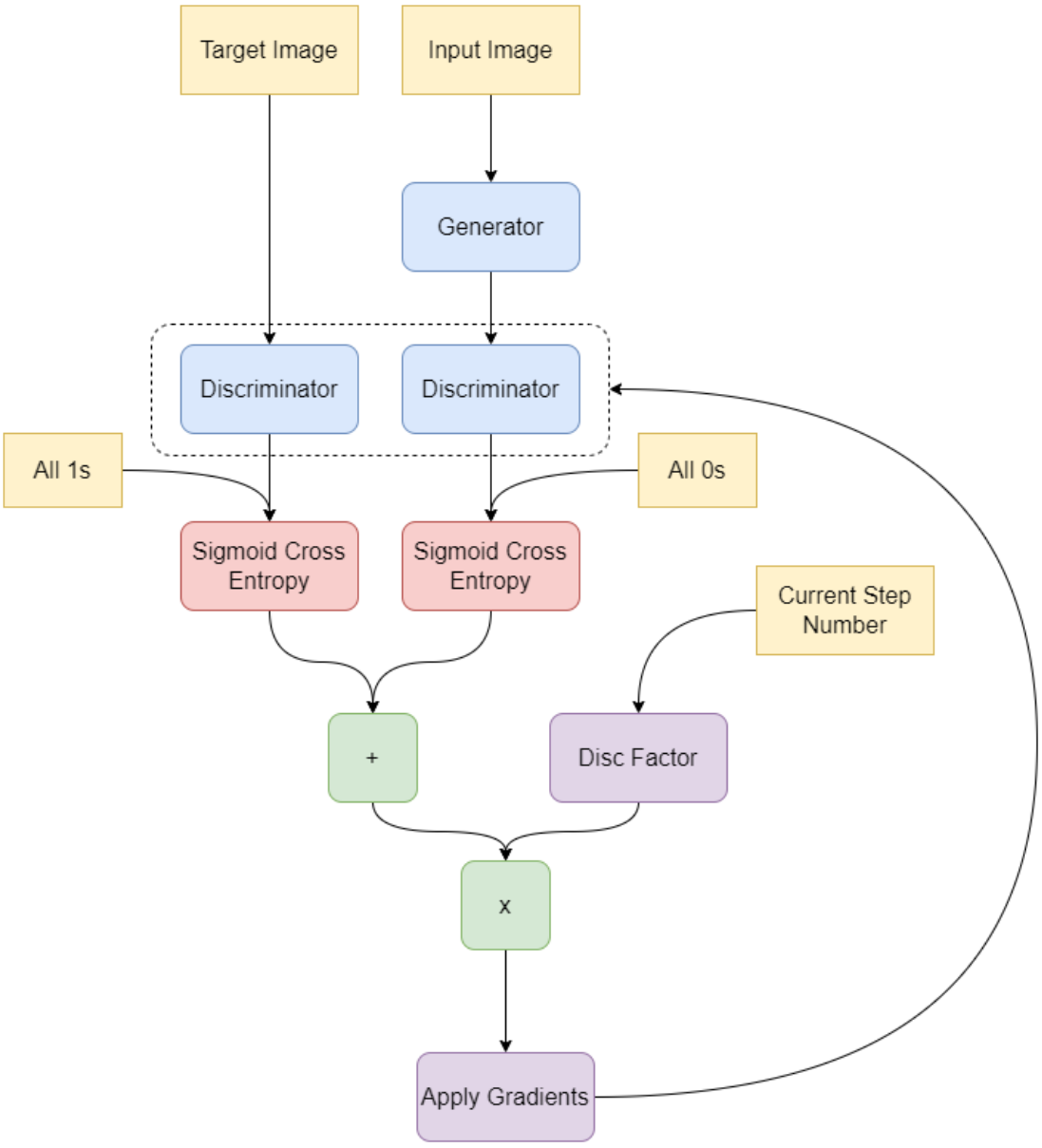
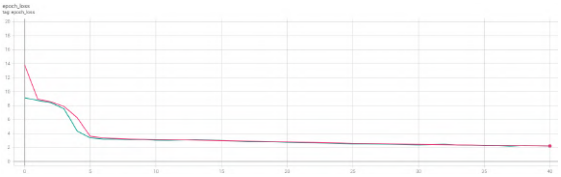


Figure 23: Training procedure of the VQ-GAN discriminator

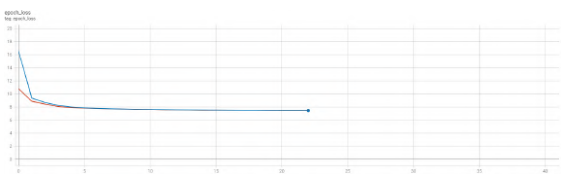
3.3 Video synthesis

The second part of the training uses a GPT-like transformer model to generate a video frame by frame. The MinGPT architecture was re-implemented as in the VQ-GAN paper but was replaced with the GPT-2 model from Huggingface [39] due to performance issues as seen on Fig. 24. The resulting videos were also visually better with the Huggingface model.

Table 4 shows the available parameters for the Huggingface GPT-2 model. Table 5 shows the different pretrained Huggingface GPT-2 models. Even though the pretrained GPT-2 model has been trained on English text, using a pretrained model allows faster results as seen on Fig. 26.



(a) Learning curve of the Huggingface GPT-2 transformer model



(b) Learning curve of the VQ-GAN transformer model

Figure 24: Both model were trained with the same parameters for the same amount of time. The Huggingface model (left) was able to do more epochs (40) and reached a better loss (∼2) than the VQ-GAN who did only 22 epochs with a resulting loss of ∼7.8.

Parameter	Description
vocab_size	Number of different tokens that can be represented
n_positions	Maximum sequence length that the model can be used with
n_layer	Number of hidden layers
n_head	Number of attention heads for each attention layer
n_embd	Dimensionality of the embeddings and hidden states
embd_pdrop	The dropout ratio for the embeddings
resid_pdrop	The dropout ratio for all dense layers
attn_pdrop	The dropout ratio for the attention

Table 4: List of available parameters for the transformer model

It has been proven as shown on Fig. 25 that the more parameters a transformer has (i.e. the deeper the network is), the better are the results. Due to memory issues, the largest transformer model that could be used was the **GPT-2 medium** with 345 million parameters.

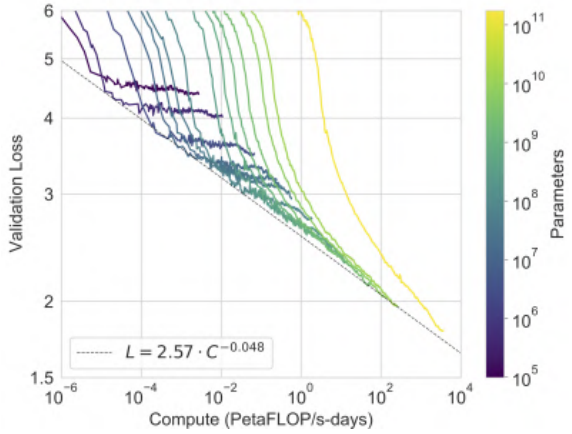
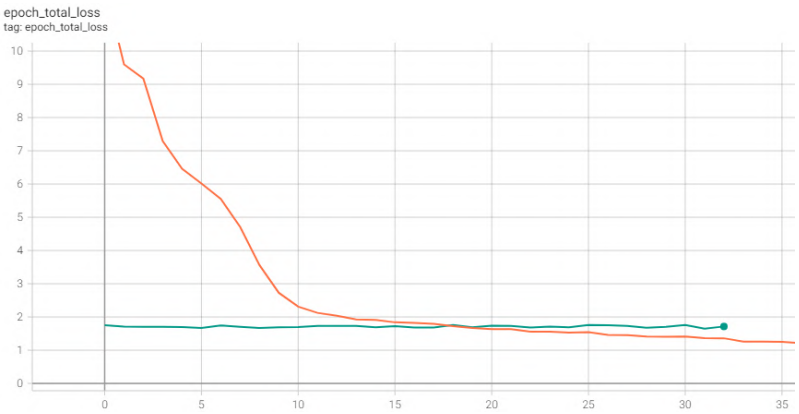


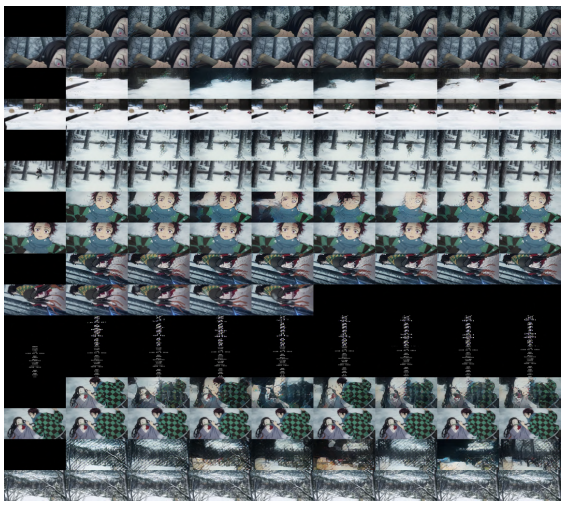
Figure 25: Relationship between the number of parameters of a transformer model and the validation loss [40].

Model name	Parameters
gpt2	12-layer, 768-hidden, 12-heads, 117M parameters.
gpt2-medium	24-layer, 1024-hidden, 16-heads, 345M parameters.
gpt2-large	36-layer, 1280-hidden, 20-heads, 774M parameters.
gpt2-xl	48-layer, 1600-hidden, 25-heads, 1558M parameters.

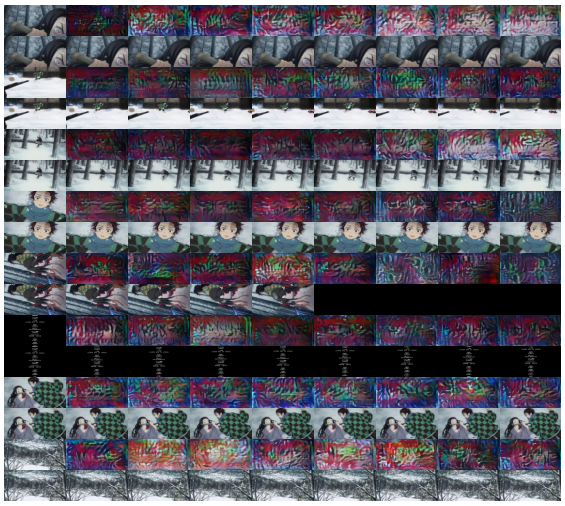
Table 5: List of pretrained Huggingface GPT-2 models with their parameters [41].



(a) Orange: training loss of a pretrained model.
Green: training loss of an untrained model



(b) Example of results with a pretrained GPT-2



(c) Example of results with an untrained GPT-2

Figure 26: For each pair of rows, the first one is generated while the second is the ground truth. Although the use case is different, using a pretrained model quickly produces results. The untrained model loss (in green) stagnates and the results are of poor quality, i.e completely different from the ground truth.

3.3.1 Conditioned synthesis

The video generation is done frame by frame in a similar fashion to the Point-to-Point Video Generation model. Fig. 27 shows how the training is done iteratively to generate a video frame by frame. With the last frame of the sequence and the previously generated frames, the next frame is predicted.

In a video sequence composed of n frames, to generate the frame F_{i+1}, the last frame F_n and the M previous frames $F_i, F_{i-1}, \cdots, F_{i-M}$ are encoded into the indices I_n and $I_i, I_{i-1}, \cdots, I_{i-M}$ by using the encoder part E of the VQ-GAN. The generated frame $\tilde{F}_{i+1}$ is the result of the transformer T with the input s_i once decoded with the decoder D part of the VQ-GAN:

$$I_i = E(F_i) \tag{9}$$

$$s_i = [frame\_counter, I_n, I_i, I_{i-1}, I_{i-2}, \cdots, I_{i-M}] \tag{10}$$

Where $frame\_counter$ is the number of remaining frames to generate and M is the number of previous frames to use.

$$\tilde{F}_i = D(T(s_i)) \tag{11}$$

The loss is defined as:

$$\mathcal{L} = \sum_{i=0}^{n-1} \mathcal{L}_{SCCE}(I_{i+1}, \tilde{I}_{i+1}) + \mathcal{L}_{Perceptual}(F_{i+1}, \tilde{F}_{i+1}) \tag{12}$$

Where $\mathcal{L}_{SCCE}(I_{i+1}, \tilde{I}_{i+1})$ is the Sparse Categorical Cross Entropy between real and generated indices and $\mathcal{L}_{Perceptual}(F_{i+1}, \tilde{F}_{i+1})$ is the perceptual loss between the real and generated frame.

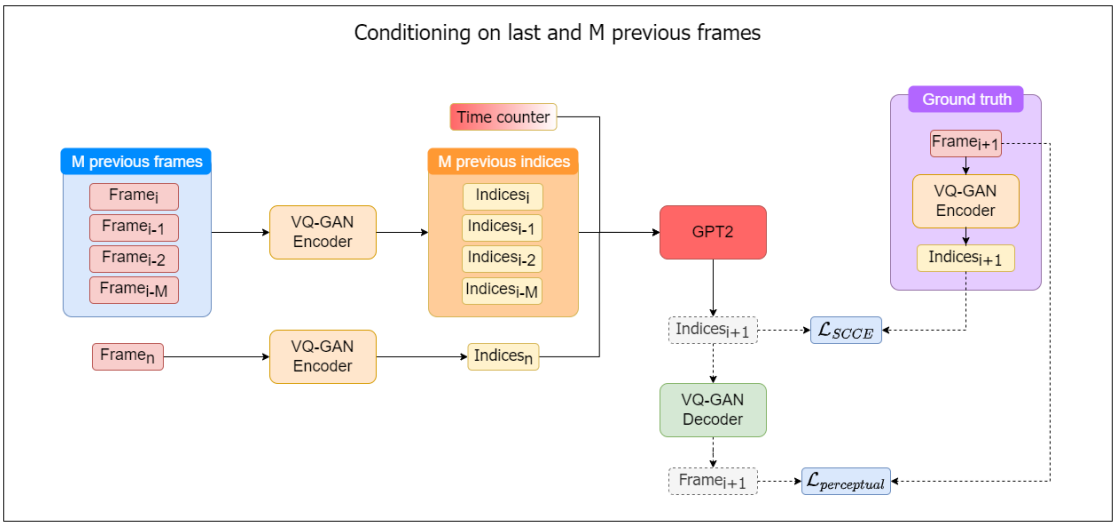


Figure 27: Training of the video generation model by using M previous frames and the last frame as input to predict the next frame

Table 6 shows the frames used when generating the next frame when M previous frames is 3.

Input frames	Generated frame
0, last	1
0, 1, last	2
0, 1, 2, last	3
1, 2, 3, last	4

Table 6: Example of input frames when M previous frames = 3

Since during the first training epochs the generated results are of bad quality, it does not make sense to feed the generated images back to the model as previous inputs. The model would learn to predict a certain frame based on the previously generated frames which are note close yet to the reality. Thus, the ground truth is used during a certain number of epochs (defined with the parameter $stop\_ground\_truth\_after\_epoch$) until the model starts generating good results. After this number of epoch, the generated frames are fed to the transformer.

Fig. 28 shows how the ground truth is used during the first step of the training until good results are produced. Afterwards, the generated frames are used as inputs to generate the subsequent frames as shown on Fig. 29. During inference, since only the first and last frames are given, the video is generated in the latter fashion.

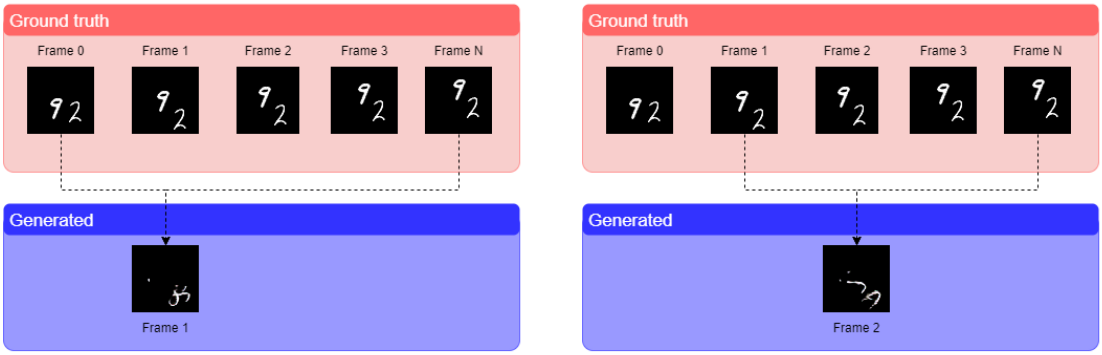


Figure 28: Use the ground truth as input frame during a certain number of epochs since the generated results are not good.

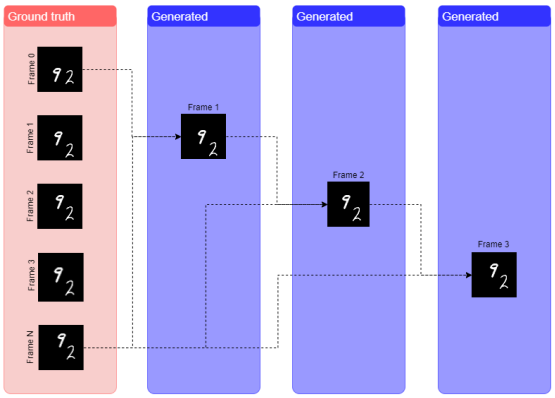


Figure 29: When the results are good enough (i.e. after a defined number of epochs), use the generated frame as input for the previous frames.

3.3.2 Remaining frames number

When generating a video from the first and last frame, it is important to know how many frames remain to be generated. The generated frame might differ a lot depending on the total number of frame and its position inside the video. Three different methods have been tried to include the temporal information to the model (i.e. the number of remaining frames).

Concatenation The first method consists of concatenating the remaining frame number, which is an integer, to the indices of the previous and last frames. The transformer input thus becomes an array holding [remaining_frame_number, previous_frames_indices, last_frame_indices].

Token type id The second method uses the parameter token_type_id of the Huggingface GPT-2 transformer. Normally this parameter is used by models that classify pairs of sentences to indicate which part of the input is from the first and second sentence. Code. 2 shows how the token type id is used inside the transformer. The remaining frame number passed as token_type_ids is encoded with the same embeddings as the inputs and added to the inputs before being fed to the subsequent layers.

```
def call(self, input_ids, token_type_ids, position_ids):
    inputs_embeddings = self.embedding(input_ids)
    position_embeddings = tf.gather(self.embedding, position_ids)
    token_type_embeddings = self.embedding(token_type_ids)

    hidden_states = input_embeddings
                    + position_embeddings
                    + token_type_embeddings

    #call of the subsequent layers
    ...
}
```

Code 2: How the token_type_id is used inside the model

Own embedding Instead of using the same embedding as the inputs, a different embedding is used to convert the remaining frame number. Code 3 shows how this new embedding is used in the call method of the transformer.

```
def call(self, input_ids, token_type_ids, position_ids, remaining_frames):
    inputs_embeddings = self.embedding(input_ids)
    position_embeddings = tf.gather(self.embedding, position_ids)
    token_type_embeddings = self.embedding(token_type_ids)
    remaining_frames_embeddings = self.embedding_remaining_frames(remaining_frames)

    hidden_states = input_embeddings
                    + position_embeddings
                    + token_type_embeddings
                    + remaining_frames_embeddings
    #call of the subsequent layers
    ...
}
```

Code 3: Modified call method to use a new embedding for the remaining frame number

Comparison between the three methods

Fig. 30 shows the loss comparison between the three methods to add the remaining frames number as a model feature. In red, it is the baseline, i.e, the model without the new feature. In orange, the remaining frames number concatenated to the input. In blue, the remaining frames number given as token type id. In grey the remaining frames number given as a new embedding. We can see that adding this new features helps the model achieve a better loss since the three different methods are under the red curve. However, the difference between the three methods is minimal. In terms of simplicity, the orange solution (concatenation) is easier to implement. The new embedding solution produces slightly better results but is the most difficult to implement. The adopted solution is the latter.

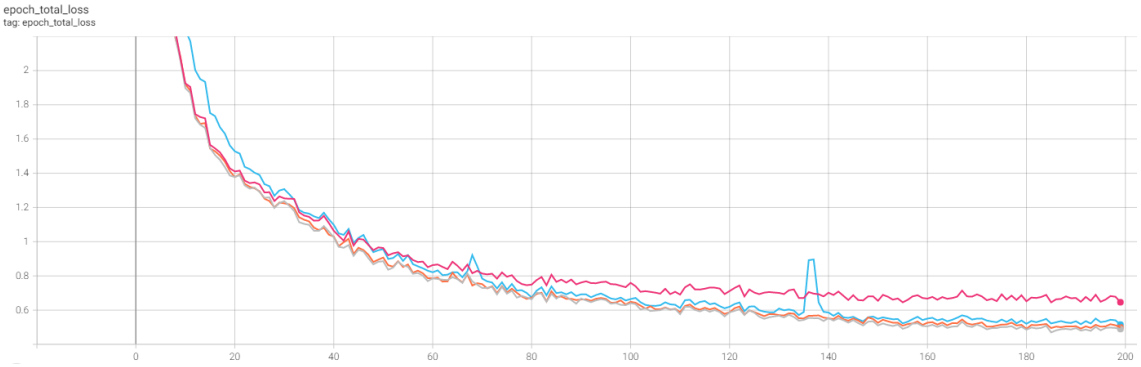


Figure 30: Training loss comparison between the different methods for the remaining frames. **In red**: without the remaining frame input. **In orange**: remaining frames number concatenated to the input. **In blue**: remaining frames number given as token type id. **In grey**: remaining frames number given as a new embedding

3.3.3 Comparison N frames before

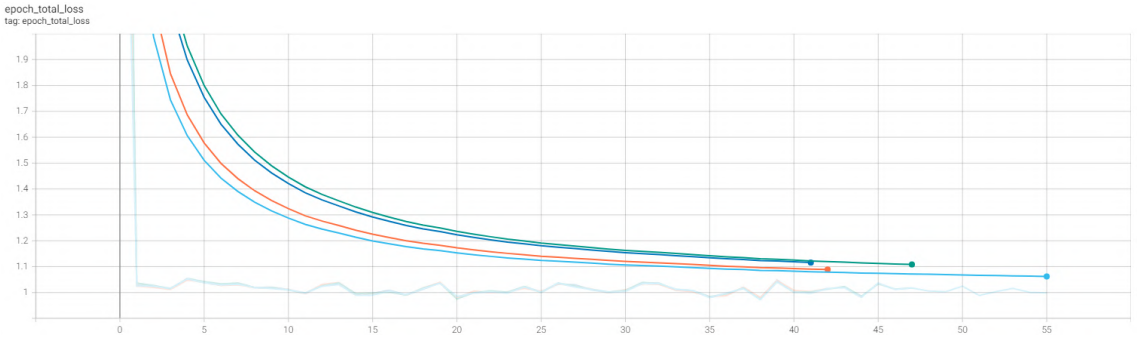
Different values of previous frames have been tested to study the impact on video generation. Fig. 31 shows the learning curve on the training and validation set. The curves are smoothed for better readability. Without smoothing, the impact in terms of value is minimal. With the smoothing, a

slight difference is visible. When increasing the number of previous frames, more memory is used and more time is required to process on epoch.

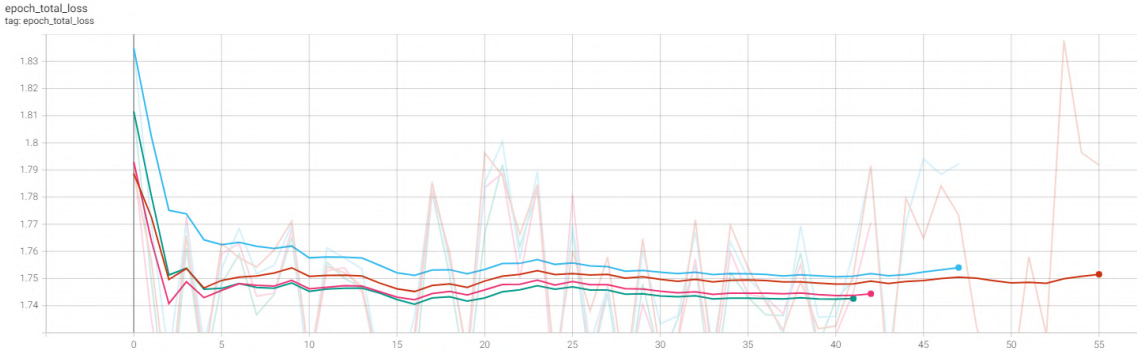
For the training as seen on Fig. 31a, the light blue curve corresponds to 1 frame before, the green curve to 2 frames, the orange to 5 frames and the dark blue to 8 frames. The best value corresponds to only one frame.

For the validation data on Fig. 31b, the red curve corresponds to a single previous frame, blue curve to 2 previous frames, the pink curve to 5 previous frames and the green curve to 8 previous frames. The best value is with 8 previous frames.

If the processing power and the available time allows it, the bigger the number of previous frame used the better. However, only using a single previous frame already gives good results and is much more time and memory efficient.



(a) Learning curve on the train set. the light blue curve corresponds to 1 frame before, the green curve to 2 frames, the orange to 5 frames and the dark blue to 8 frames.



(b) Learning curve on the validation set. The red curve corresponds to 1 previous frame, the blue to 2 previous frames, the pink to 5 previous frames and the green to 8 previous frames.

Figure 31: Impact of different number of previous frames given to the model.

3.3.4 Training details

The VQ-GAN was trained with the parameters defined on Appendix C.1 on 8 RTX3090 GPUs without discriminator for a duration of 1 day for MovingMNIST and 6 days for the Kimetsu no Yaiba dataset.

The GPT-2 medium transformer from Huggingface was trained with the parameters defined on Appendix C.2 on 8 RTX3090 GPUs for a duration of 1 day for MovingMNIST and 3 days for the Kimetsu no Yaiba dataset. For batching the dataset, videos are truncated to the lowest number of frames. The number of remaining frames is given to the model using the own embedding method

as described in Chapter 3.3.2. When generating a new frame, only the previous frame (with the last video frame) is used.

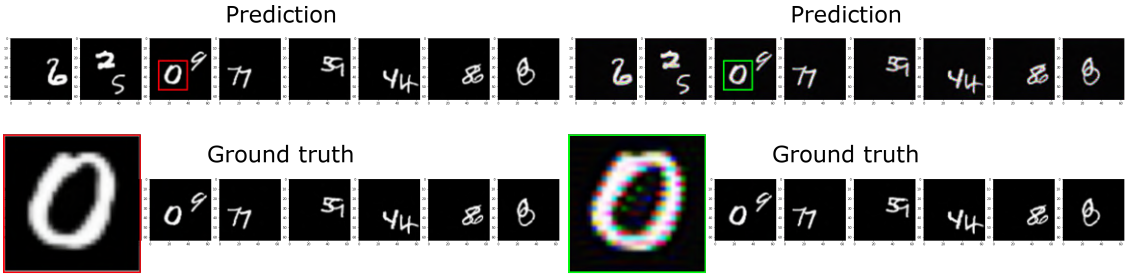
4 Results

This section will present the different results obtained on the first and second stage of the training with the MovingMNIST dataset and the Kimetsu no Yaiba dataset. For both datasets, 80% of the data was used for the training, 10% for validation and 10% for the test. The numerical results are calculated between 5 iterations on the entire train and test set to take into account the stochastic effect of shuffling and pre-processing (jitter and video batching). For the VQ-GAN, the Fréchet Inception Distance (FID), Structural Similarity (SSIM) and Peak Signal-to-Noise Ratio (PSNR) are calculated. For the video generation, the Fréchet Video Distance (FVD), Kernel Video Distance (KVD), SSIM and PSNR are calculated.

4.1 First stage (VQ-GAN)

4.1.1 MovingMNIST

Since the data is rather simple, training the VQ-GAN without the discriminator already produces good results. Similarly to the VQ-GAN paper, the model is first trained without the adversarial loss so that the generator can learn to autoencode the data, then the discriminator is applied after a certain number of steps that can be configured with the config file. However, after applying the discriminator, strange artifacts appears on the image as seen on Fig. 32b. Considering that the generated images were satisfying, the discriminator was not used for this model.



(a) Results when training without the discriminator (b) Results when training with the discriminator

Figure 32: Example of VQ-GAN results on the test set of the MovingMNIST dataset

Table 7 shows numerical results between the train and test set on the MovingMNIST dataset.

Dataset	FID (↓)	SSIM (↓)	PSNR (↓)
Train	17.8 ± 0.0	0.832 ± 0.000	23.69 ± 0.000
Test	18.1 ± 0.1	0.832 ± 0.000	23.69 ± 0.006

Table 7: Numerical results on image generation for the MovingMNIST dataset

4.1.2 Kimetsu no Yaiba

Comparable to the MovingMNIST dataset, the training was also done in two parts. The discriminator was not used for the first part of the training, then was applied after a certain number of steps.

An experiment was done with different types of perceptual loss taken from the *FILM: Frame Interpolation for Large Motion* paper [42, 11] where the style loss seems to produce sharper results. A perceptual loss using VGG19 instead of VGG16 was available as well. Since the code is also in Tensorflow, using these loss functions was an easy task.

Figure 33: Comparison of the style, VGG16 and VGG19 loss for training the VQ-GAN on the Kimetsu no Yaiba dataset

The Figure 33 shows the comparison between the style, VGG16 and VGG19 loss used as the perceptual loss. The style loss initially showed strange results with some element displacement, however it was found later that a term was missing from the reconstruction loss. The training loss is supposed to be the Mean Average Error + Perceptual loss, however only the Perceptual loss was used. Regardless of this mistake, VGG19 seems to produce better results than VGG16.

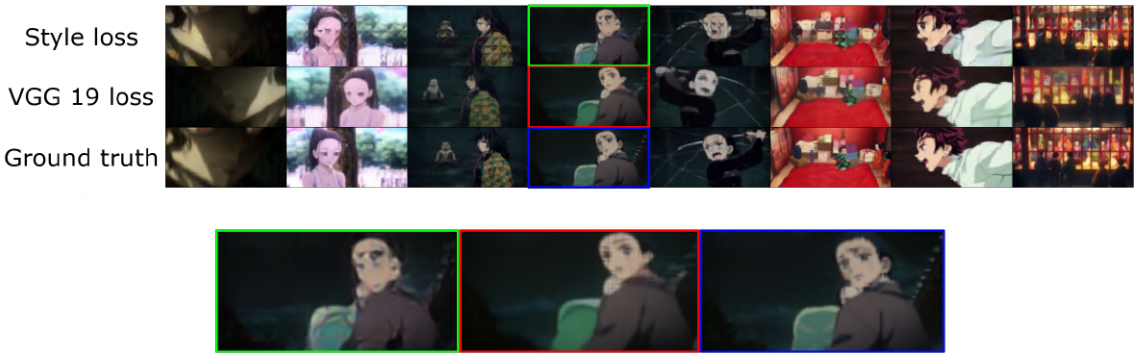


Figure 34: Comparison of the style and VGG19 loss on the Kimetsu no Yaiba dataset after correction of the reconstruction loss

The figure 34 shows the comparison of the style and VGG19 loss functions where the MAE term is correctly added to the perceptual loss. The symmetrical and framing difference between the different results is due to the data augmentation layer where a portion of the image is taken and randomly mirrored. We can see however that the VGG19 loss seems to produce more accurate results. The VGG19 loss was then used as the perceptual loss for the subsequent models.

After adding the adversarial loss with the discriminator, artifacts started appearing on the generated image as with the MovingMNIST dataset. An example of the artifacts is visible on Fig. 35.

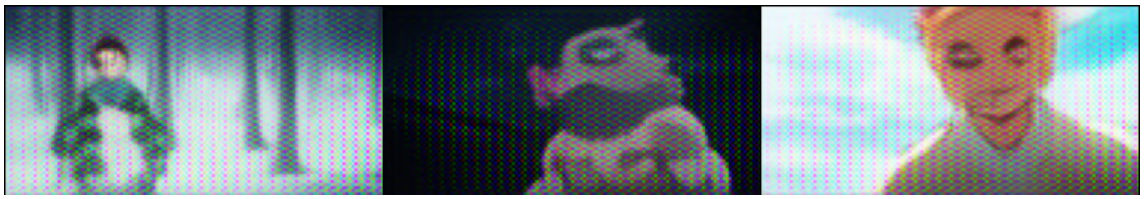


Figure 35: Artifacts appearing on the generated images when using the adversarial loss

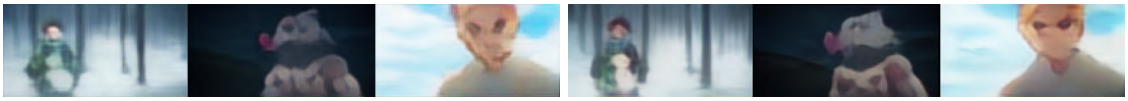
Initially, the discriminator from PatchGAN has 3 down-sampling layers. Changing the number of down-sampling layers to 2 or 4 does not remove the artifacts as seen on Fig. 36.

(a) Results when the discriminator has **2** down-sampling layers

(b) Results when the discriminator has **4** down-sampling layers

Figure 36: Artifacts still present when varying the number of down-sampling layers from the discriminator

The up-sampling method from the decoder uses the nearest neighbor. Changing from nearest to bilinear makes the generated image smoother as seen on Fig. 37. Other methods such as area, bicubic, gaussian, lanczos3, lanczos5, or mitchellcubic are available in Tensorflow 2.9 and could potentially improve the results but have not been tested since it was not available at the time of the experiment. Once the adversarial loss is added, the artifacts still appear as seen on Fig. 38. The artifacts are less visible on a dark background when there is 3 down-sampling layers as seen on Fig. 38b, but are still present.



(a) Original up-sampling method using the nearest neighbor

(b) Results when using the bilinear up-sampling method

Figure 37: Comparison of results when up-sampling with the nearest neighbor or the bilinear method where bilinear method produces smoother results.

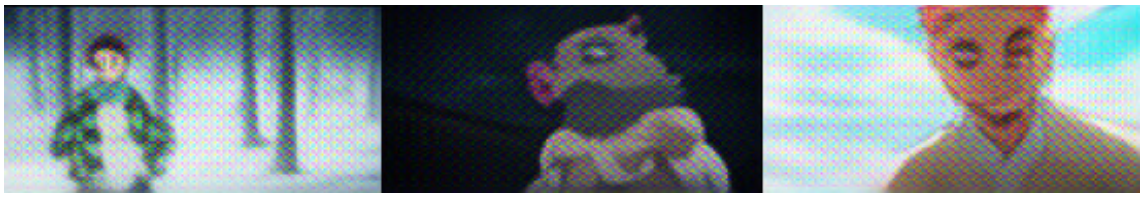
The PatchGAN discriminator works with images of 256x256 pixels. Resizing the image from 64x128 to 256x512 pixel before feeding to the discriminator does not generate the artifacts, however the model learning stagnates until the discriminator loss drops to 0 producing a failure mode.

Changing the discriminator architecture from a multi output (one value, true or fake, per image patch) to a single output (true or fake for the entire image) also generate the artifacts.

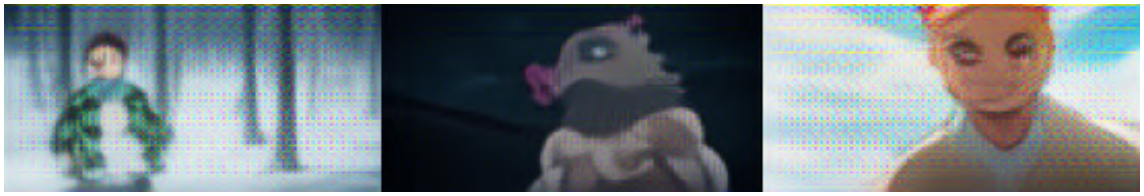
According to the *Long Video Generation with Time-Agnostic VQGAN and Time-Sensitive Transformer* paper [34] having a pure convolutional architecture for the encoder-decoder (i.e, removing the attention layers) can help with the failure mode. Due to the remaining time, this could not be tested but could be interesting for a later work.

Numerical results between the train and test set on the Kimetsu no Yaiba dataset are visible on table 8.

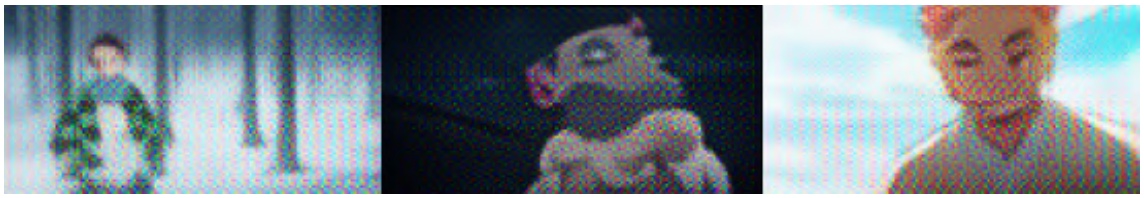
A comparison between the generated results and the ground truth is shown on Fig. 39.



(a) Bilinear up-sampling on the decoder, **2** down-sampling layers on the discriminator



(b) Bilinear up-sampling on the decoder, **3** down-sampling layers on the discriminator



(c) Bilinear up-sampling on the decoder, **4** down-sampling layers on the discriminator

Figure 38: Comparison of different number of down-sampling layers on the discriminator with the bilinear up-sampling on the decoder.

Figure 39: Example of final generated results on the Kimetsu no Yaiba dataset

Dataset	FID (↓)	SSIM (↓)	PSNR (↓)
Train	14.8 ± 0.1	0.660 ± 0.000	17.43 ± 0.007
Test	16.1 ± 0.1	0.660 ± 0.000	17.423 ± 0.044

Table 8: Numerical results on image generation for the Kimetsu no Yaiba dataset

4.2 Second stage (GPT-2 transformer)

4.2.1 MovingMNIST

Not a lot of time was spent training the transformer on this dataset, thus the results are not perfect. Even though the generated images does not always look like numbers, the general movement seems to be correctly learnt as seen on Fig. 40. For each pairs of rows, the first one is generated (starting with a grey square), the second one is the ground truth. The first generated frames looks close to the ground truth, then it starts to diverge while keeping a good positional coherence until the last frame that is correctly reconstructed. Which is a good sign since the model has access to the last frame indices.

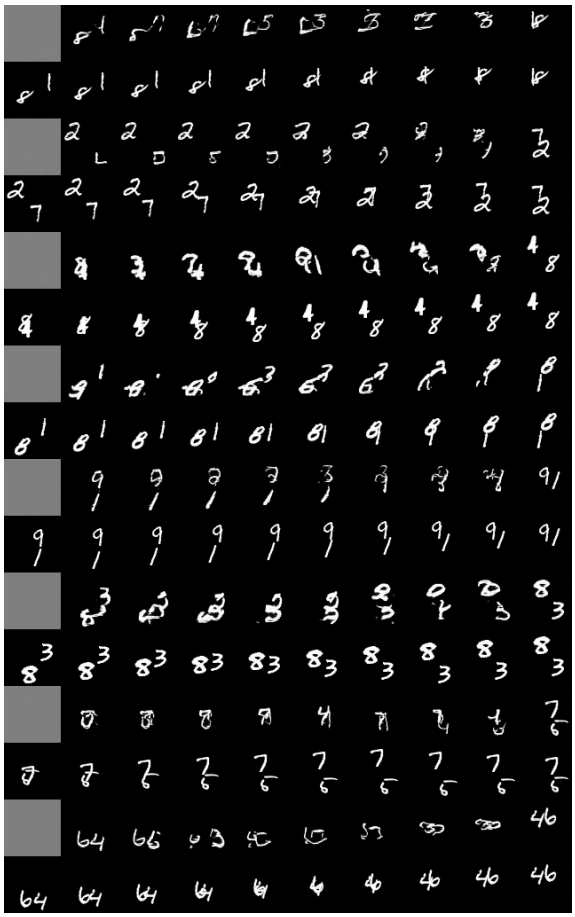


Figure 40: Video results on the moving MNIST dataset

Numerical results on video generation for the MovingMNIST dataset are shown on Table 9.

Dataset	FVD (↓)	KVD (↓)	SSIM (↓)	PSNR (↓)
Train	61.1 ± 0.4	7.9 ± 0.1	0.697 ± 0.000	21.985 ± 0.004
Test	64.9 ± 0.9	7.8 ± 0.3	0.698 ± 0.002	21.990 ± 0.029

Table 9: Numerical results on video generation for the MovingMNIST dataset

4.2.2 Kimetsu no Yaiba

Compared to the MovingMNIST dataset, the results are visually more coherent. The model has learnt to transition from the first to the last frame with some limitations. Figure 41 shows some examples of generation where each row starting with a gray frame is the generated video and the row under it is the ground truth. Table 10 shows numerical results for the video generation on the train and test set.

In the first generated video, the character head first goes down, then goes up. The head position on the last frame is higher than the first frame. The model generates only an upward movement even though the ground truth should have been downward then upwards, which is understandable since the model had no further information.

Figure 41: Video results on the Kimetsu no Yaiba dataset

On the second video, we can see the character raising her head then opening her eyes. The eyes during the transition from closed to open are blurry due to the VQ-GAN not able to generate all kind of pictures. However the generated results globally represent the action of opening the eyes.

On the third video, the character closes his mouth then opens it for the two last frames with a slight head movement. The generated animation corresponds to a closing mouth with a head movement. The model learned to animate a mouth even though from the first and last frame there is no information that the mouth would be closed.

The last example shows blood flowing from the upper right corner of the eye. The model learned to gradually make blood appear. However, the blood does not totally flow from the eyelid. Some spots appears from the middle of the eye but corresponds quite well in the end to the target picture.

Surprising results Although not that visible on paper, Fig. 42 shows some surprising results.

Figure 42: Surprising results on the Kimetsu no Yaiba dataset

The first video is supposed to be a slight camera tilt (moving down to up), however, the model generated a zoom in and out.

On the second video, the middle circle logo appeared suddenly but the generated result made it appears progressively.

For the third video, the character is talking (mouth animation), but the generated images are pulsating making it looks like a breathing.

On the last video, the ground truth has no animations, but the model generated slight eyelids movements.

Dataset	FVD (↓)	KVD (↓)	SSIM (↓)	PSNR (↓)
Train	77.6 ± 0.3	3.8 ± 0.0	0.589 ± 0.000	16.624 ± 0.014
Test	79.8 ± 2.7	3.9 ± 0.1	0.604 ± 0.001	17.114 ± 0.066

Table 10: Numerical results on video generation for the Kimetsu no Yaiba dataset

5 Discussion

Combining the VQ-GAN image generation with a GPT-2 transformer for generating video sequences seems promising. Despite being rather blurry, the generated videos show meaningful movements. The generated frames show good temporal consistency from the first to the last frame. However, small motions such as mouth animations, or head tilting are better represented than a whole scene change. The model is also capable of generating interesting animations when the first and last frames are identical. The numerical results, although not comparable with other models because of the novel dataset, are of similar magnitude to other video generation projects.

Another recent work [34] also combines a VQ-GAN with a GPT transformer for video generation with a slightly different approach. The VQ-GAN is modified to generate a video in one-shot and not frame by frame. Two transformers are used, one to generate key-frames and another to interpolate images between the key frames. And lastly, two discriminator are used. One for frame coherence and another for video coherence. This work proves that using a VQ-GAN in combination with a transformer is in line with the state of the art for video generation.

The resulting videos, while not perfect, globally show the intended movements between the frames. The model could be improved in multiple ways. The first and foremost is to properly train the VQ-GAN. Since it generates the resulting frames, it is important to have a model that can produce frames as similar to the target frame. A better investigation of why the discriminator produced artifacts on the resulting pictures might improve the VQ-GAN. One possible cause could be the attention layers in the encoder and decoder as mentioned in the *Long Video Generation with Time-Agnostic VQGAN and Time-Sensitive Transformer* paper [34].

Another improvement could be to use a transformer with even more parameters. Due to resources limitations, only the GPT-2 medium with 345M parameters was used, but Huggingface proposes other pretrained models such as GPT-2 large with 774M or GPT-2 xl with 1558M parameters. The server used has 8xRTX3090 with 24GB of memory each. Using GPUs with more memory would be able to handle bigger models. One alternative would also be to use PyTorch instead of Tensorflow since methods to reduce model memory such as gradient checkpointing, float16 or model parallelism seems easier to use. There are probably also more recent transformers architecture that could be used instead of GPT-2.

The dataset diversity could also be improved. Multiple scenes present in the Kimetsu no Yaiba dataset contained few or no motions at all. These scenes could be removed and would allow the model to better learn the animations. Also, taking scenes from a greater diversity of anime could result in a better generalization.

Capturing the intended movement only with two frames is still a complicated task. In the context of anime, the first and last frames can sometimes be identical with a number of different frames in between. More information could be added to the model, such as additional control frames or a textual description as in the DALL-E model [7].

6 Conclusion

We propose GANime, a model capable of generating videos of anime content based on two frames, the first and the last. This model is composed of a VQ-GAN for image generation and a GPT-2 transformer for temporal consistency. Videos are generated frame by frame by using the last frame of the sequence and the previously generated frame to predict the next frame. With this method, the video length can be specified allowing the generation of short or longer sequences. While there is still room for improvements, the resulting videos show coherent motions compared to the ground truth. Having two inputs, the first and last frame, helps for generating meaningful movements. However, when an object suddenly appears, or a movement which is not captured by the first or last frame occurs, the generated result tends to not be accurate. Additional inputs or a better segmentation of the data could help solve these problems. The model is also able to generate some motions when the first and last frames are identical, showing that the output can be creative.

Bibliography

[1] Ian J. Goodfellow et al. *Generative Adversarial Networks*. 2014. DOI: 10.48550/ARXIV.1406.2661. URL: https://arxiv.org/abs/1406.2661.

[2] Tero Karras, Samuli Laine and Timo Aila. *A Style-Based Generator Architecture for Generative Adversarial Networks*. 2018. DOI: 10.48550/ARXIV.1812.04948. URL: https://arxiv.org/abs/1812.04948.

[3] *This person does not exist*. URL: https://this-person-does-not-exist.com.

[4] *Introduction to Diffusion Models*. URL: https://www.assemblyai.com/blog/diffusion-models-for-machine-learning-introduction/.

[5] Jonathan Ho et al. *Video Diffusion Models*. https://github.com/lucidrains/video-diffusion-pytorch. 2022. DOI: 10.48550/ARXIV.2204.03458. URL: https://arxiv.org/abs/2204.03458.

[6] Aditya Ramesh et al. *Hierarchical Text-Conditional Image Generation with CLIP Latents*. 2022. DOI: 10.48550/ARXIV.2204.06125. URL: https://arxiv.org/abs/2204.06125.

[7] Aditya Ramesh et al. *Zero-Shot Text-to-Image Generation*. 2021. DOI: 10.48550/ARXIV.2102.12092. URL: https://arxiv.org/abs/2102.12092.

[8] Aaron van den Oord, Oriol Vinyals and Koray Kavukcuoglu. *Neural Discrete Representation Learning*. 2017. DOI: 10.48550/ARXIV.1711.00937. URL: https://arxiv.org/abs/1711.00937.

[9] Patrick Esser, Robin Rombach and Björn Ommer. *Taming Transformers for High-Resolution Image Synthesis*. 2020. DOI: 10.48550/ARXIV.2012.09841. URL: https://arxiv.org/abs/2012.09841.

[10] Mehdi Mirza and Simon Osindero. *Conditional Generative Adversarial Nets*. 2014. DOI: 10.48550/ARXIV.1411.1784. URL: https://arxiv.org/abs/1411.1784.

[11] Fitsum Reda et al. *Tensorflow 2 Implementation of "FILM: Frame Interpolation for Large Motion"*. https://github.com/google-research/frame-interpolation. 2022.

[12] Liying Lu et al. *Video Frame Interpolation with Transformer*. 2022. DOI: 10.48550/ARXIV.2205.07230. URL: https://arxiv.org/abs/2205.07230.

[13] Wenbo Bao et al. *MEMC-Net: Motion Estimation and Motion Compensation Driven Neural Network for Video Interpolation and Enhancement*. 2018. DOI: 10.48550/ARXIV.1810.08768. URL: https://arxiv.org/abs/1810.08768.

[14] Carl Vondrick, Hamed Pirsiavash and Antonio Torralba. *Generating Videos with Scene Dynamics*. 2016. DOI: 10.48550/ARXIV.1609.02612. URL: https://arxiv.org/abs/1609.02612.

[15] Wilson Yan et al. *VideoGPT: Video Generation using VQ-VAE and Transformers*. 2021. DOI: 10.48550/ARXIV.2104.10157. URL: https://arxiv.org/abs/2104.10157.

[16] Aidan Clark, Jeff Donahue and Karen Simonyan. *Adversarial Video Generation on Complex Datasets*. 2019. DOI: 10.48550/ARXIV.1907.06571. URL: https://arxiv.org/abs/1907.06571.

[17] Tsun-Hsuan Wang et al. *Point-to-Point Video Generation*. 2019. DOI: 10.48550/ARXIV.1904.02912. URL: https://arxiv.org/abs/1904.02912.

[18] *Curated list of papers on anime or manga*. URL: https://github.com/SerialLain3170/AwesomeAnimeResearch/.

[19] Yang Zhou et al. 'MakeltTalk'. In: 39.6 (Dec. 2020), pp. 1–15. DOI: 10.1145/3414685.3417774. URL: https://arxiv.org/abs/2004.12992.

[20] Koichi Hamada et al. *Full-body High-resolution Anime Generation with Progressive Structure-conditional Generative Adversarial Networks*. 2018. DOI: 10.48550/ARXIV.1809.01890. URL: https://arxiv.org/abs/1809.01890.

[21] Li Siyao et al. *Deep Animation Video Interpolation in the Wild*. 2021. DOI: 10.48550/ARXIV.2104.02495. URL: https://arxiv.org/abs/2104.02495.

[22] Wang Shen et al. *Enhanced Deep Animation Video Interpolation*. 2022. DOI: 10.48550/ARXIV.2206.12657. URL: https://arxiv.org/abs/2206.12657.

[23] *Cost of one episode of the Kimetsu no Yaiba anime*. URL: https://animegalaxyofficial.com/demon-slayer-season-2-budget-for-the-making-of-single-episode-is-way-more-expensive/.

[24] *Number of frames for an anime*. URL: https://coconala.com/magazine/3042.

[25] *Time required for a studio to create a 12 episode long anime*. URL: https://www.quora.com/How-long-does-it-take-animators-to-make-a-12-episode-season-of-anime.

[26] Aqeel Anwaar. *Difference between AutoEncoder (AE) and Variational AutoEncoder (VAE)*. URL: https://towardsdatascience.com/difference-between-autoencoder-ae-and-variational-autoencoder-vae-ed7be1c038f2.

[27] Joseph Rocca. *Understanding Variational Autoencoders (VAEs)*. URL: https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73.

[28] Paul Sayak. *VQ-VAE Keras implementation*. 2021. URL: https://keras.io/examples/generative/vq_vae/.

[29] Ashish Vaswani et al. *Attention Is All You Need*. 2017. DOI: 10.48550/ARXIV.1706.03762. URL: https://arxiv.org/abs/1706.03762.

[30] Justin Johnson, Alexandre Alahi and Li Fei-Fei. *Perceptual Losses for Real-Time Style Transfer and Super-Resolution*. 2016. DOI: 10.48550/ARXIV.1603.08155. URL: https://arxiv.org/abs/1603.08155.

[31] Phillip Isola et al. *Image-to-Image Translation with Conditional Adversarial Networks*. 2016. DOI: 10.48550/ARXIV.1611.07004. URL: https://arxiv.org/abs/1611.07004.

[32] Vasily Zadorozhnyy, Qiang Cheng and Qiang Ye. *Adaptive Weighted Discriminator for Training Generative Adversarial Networks*. 2020. DOI: 10.48550/ARXIV.2012.03149. URL: https://arxiv.org/abs/2012.03149.

[33] u/chasep255. *[D] Adaptive loss weight in VQGAN paper*. 2021. URL: https://www.reddit.com/r/MachineLearning/comments/q6ilpd/d_adaptive_loss_weight_in_vqgan_paper/.

[34] Songwei Ge et al. *Long Video Generation with Time-Agnostic VQGAN and Time-Sensitive Transformer*. 2022. DOI: 10.48550/ARXIV.2204.03638. URL: https://arxiv.org/abs/2204.03638.

[35] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[36] *Moving MNIST dataset*. URL: http://www.cs.toronto.edu/~nitish/unsupervised_video/.

[37] Nitish Srivastava, Elman Mansimov and Ruslan Salakhutdinov. 'Unsupervised Learning of Video Representations using LSTMs'. In: *CoRR* abs/1502.04681 (2015). arXiv: 1502.04681. URL: http://arxiv.org/abs/1502.04681.

[38] Brandon Castellano. *PySceneDetect*. URL: https://github.com/Breakthrough/PySceneDetect.

[39] Thomas Wolf et al. 'Transformers: State-of-the-Art Natural Language Processing'. In: *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. Online: Association for Computational Linguistics, Oct. 2020, pp. 38–45. URL: https://www.aclweb.org/anthology/2020.emnlp-demos.6.

[40] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. DOI: 10.48550/ARXIV.2005.14165. URL: https://arxiv.org/abs/2005.14165.

[41] *Huggingface pretrained models*. URL: https://huggingface.co/transformers/v2.2.0/pretrained_models.html.

[42] Fitsum Reda et al. *FILM: Frame Interpolation for Large Motion*. 2022. DOI: 10.48550/ARXIV.2202.04901. URL: https://arxiv.org/abs/2202.04901.

[43] The AI Epiphany. *VQ-VAEs: Neural Discrete Representation Learning — Paper + PyTorch Code Explained*. 2021. URL: https://youtu.be/VZFVUrYcig0?t=1393.

[44] Augustus Odena, Vincent Dumoulin and Chris Olah. 'Deconvolution and Checkerboard Artifacts'. In: *Distill* (2016). DOI: 10.23915/distill.00003. URL: http://distill.pub/2016/deconv-checkerboard.

[45] Aaron van den Oord et al. *Conditional Image Generation with PixelCNN Decoders*. 2016. DOI: 10.48550/ARXIV.1606.05328. URL: https://arxiv.org/abs/1606.05328.

Appendix

A Straight-through gradient estimator

In Keras / Tensorflow, passing the gradient from decoder to the encoder in the VectorQuantizer layer is done by the following line:

```
quantized = x + tf.stop_gradient(quantized - x)
```

Since $x - x$ cancel out, the quantizer output will be the variable *quantized*. However, during backpropagation, $tf.stop\_gradient(quantized - x)$ will be ignored, so the gradient output will be x (the input) [28, 43].

B Problem with Conv2DTranspose

When generating images with a model, it is not uncommon to go from a low dimension space (e.g. latent vector of size 512) to a higher dimension space (e.g. image of size 64x64x3). The method often used to increase the dimension is called deconvolution or transposed convolution (Conv2DTranspose in Keras).

However, when the kernel size of the deconvolution is not divisible by the number of stride, an overlap can be created as shown on figure 43

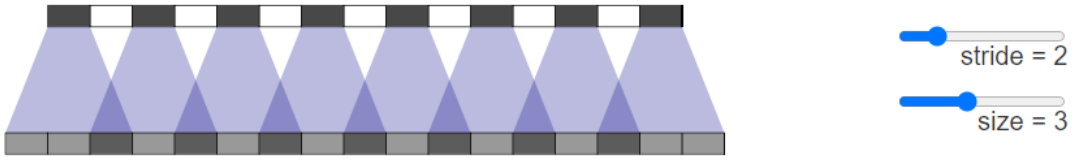


Figure 43: Overlapping with a stride of 2 and kernel of 3 (by Odena et al.)

This problem, applied in two dimensions create an overlap known as the checkboard effect [44] as shown on figure 44.

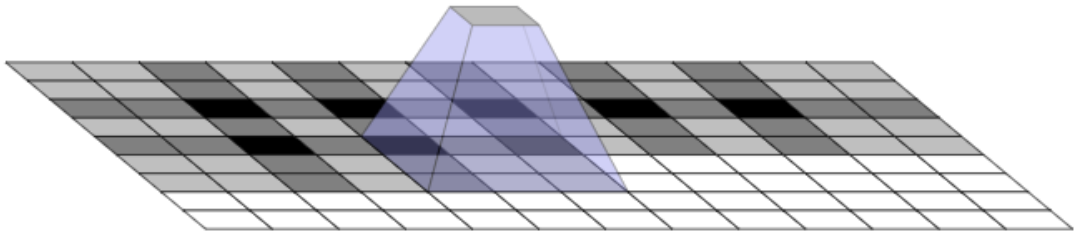


Figure 44: Checkboard effect (by Odena et al.)

During the training of the first stage of the VQ-GAN on the Kimetsu no Yaiba dataset, a checkboard effect was present as seen on Fig. 45)

The method suggested to remove this effect is to replace the deconvolution layer by an up-sampling layer then a normal convolution [44]. In keras, it is equivalent to replace the Conv2DTranspose layer

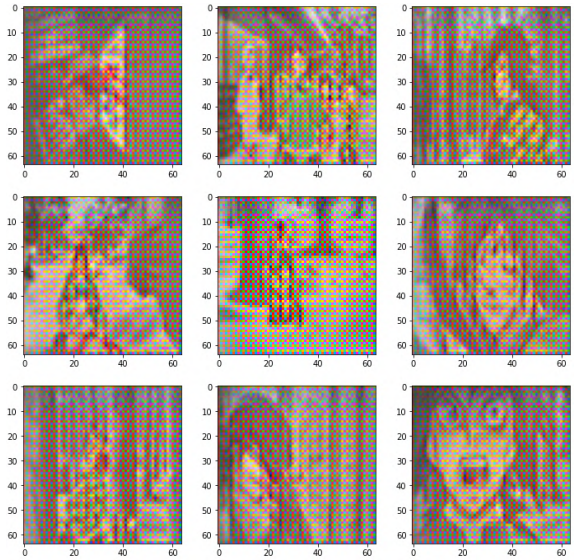
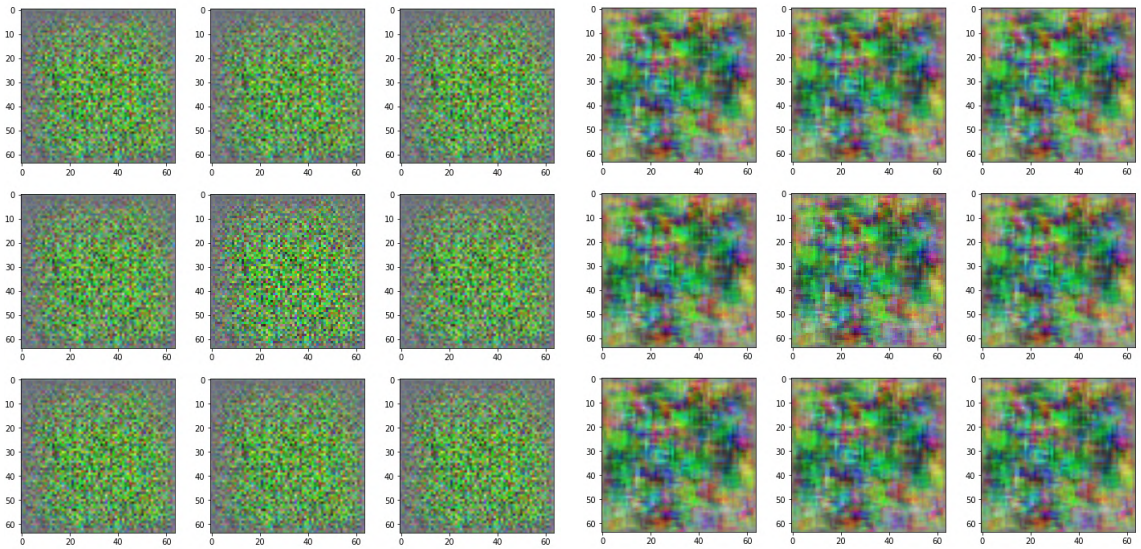


Figure 45: Checkboard effect when generating images of the anime Kimetsu no Yaiba

by an UpSampling2D followed by a Conv2D. The comparison between up-sampling + convolution vs deconvolution is visible without training the model and can be seen on Fig. 46.



(a) With deconvolution

(b) With up-sampling + convolution

Figure 46: On the left, checkboard effect already visible without training when using a decovolution layer. On the right, no artifacts visible before training when using upsampling + convolution

C YAML config files

C.1 VQ-GAN Config file

```yaml
model:
  checkpoint_path: ../../../checkpoints/kny_image/checkpoint
  vqvae_config:
    beta: 0.25
    num_embeddings: 50257
    embedding_dim: 128
  autoencoder_config:
    z_channels: 512
    channels: 32
    channels_multiplier:
    - 2
    - 4
    - 8
    - 8
    num_res_blocks: 1
    attention_resolution:
    - 16
    resolution: 128
    dropout: 0.0
  discriminator_config:
    num_layers: 3
    filters: 64

  loss_config:
    discriminator:
      loss: "hinge"
      factor: 1.0
      iter_start: 500000
      weight: 0.8
    vqvae:
      codebook_weight: 1.0
      perceptual_weight: 4.0
    perceptual_loss: "vgg19" # one between "vgg16", "vgg19", "style"

trainer:
  batch_size: 64
  n_epochs: 1000
  gen_lr: 3e-5
  disc_lr: 5e-5
  gen_beta_1: 0.5
  gen_beta_2: 0.9
  disc_beta_1: 0.5
```

```
  disc_beta_2: 0.9
  gen_clip_norm: 1.0
  disc_clip_norm: 1.0
```

C.2 GPT-2 Config file

```
model:
  transformer_config:
    # checkpoint_path: ../../../checkpoints/kny_video/checkpoint
    remaining_frames_method: "own_embeddings"
    transformer_type: "gpt2-medium"
  first_stage_config:
    # Here same parameters as in the vq-gan config file
    checkpoint_path: ...
    vqvae_config: ...
    autoencoder_config: ...
    discriminator_config: ...
    loss_config: ...

train:
  batch_size: 64
  accumulation_size: 1
  n_epochs: 500
  len_x_train: 28213
  warmup_epoch_percentage: 0.15
  lr_start: 5e-6
  lr_max: 1e-4
  perceptual_loss_weight: 1.0
  n_frames_before: 1
  stop_ground_truth_after_epoch: 200
```