Master Thesis

AI-driven animations of 2D rasterized graphics

FernUniversität in Hagen
Germany

Author: Hendrik Voßkamp
Student number: Q3042472

1st examiner: Prof. Dr.-Ing. habil. Gerke
2nd examiner & supervisor: Dr.-Ing. Peter Seibold

Faculty of mathematics and computer science
Chair of electrical engineering and information technology

The academic year 2021-2022

THIS PAGE IS INTENTIONALLY LEFT BLANK

Abstract

Manual keyframe animation is an integral part of the post-production process for 2D animations. Despite this, little research has been undertaken to utilize advances in artificial intelligence to improve this process, which still takes considerable user effort.

Inspired by recent progress in image manipulation and time series prediction through machine learning, this thesis examines the possibility of automatically creating animations between two rasterized graphics. The focus lies on small black and white 2D icon animations.

To this end, I create and compare different neural network architectures and present a multi-layered process built upon a convolutional neural network, called akaNet, for predicting the frames between a given input and output graphic. This approach gets augmented by a novel workflow using recursive triads for improved time series prediction results.

Acknowledgments

I am thankful for Prof. Dr.-Ing. habil. Gerke for accepting this thesis and Dr. Peter Seibold for helping me navigate this challenging topic and providing me with valuable feedback. Additionally, I would like to thank Hossein Alizadeh Moghaddam for being a great dialogue partner and helping me explore the idea of recursive triads. Finally, thank you, Nhung, for being a great support during difficult times.

Contents

List of acronyms

Adam	Adaptive moment estimation
akaNet	Automated keyframe animation network
ARNN	Autoregressive recurrent neural network
bbCNN	Bare bones CNN
BN	Batch normalization
CNN	Convolutional neural network
ConvLSTM	Convolutional long short-term memory
ConvVAE	Convolutional variational autoencoder
FPS	Frames per second
GPU	Graphics processing unit
GRU	Gated recurrent unit
LSTM	Long short-term memory
MLP	Multi-layer perceptron
MNIST	Modified National Institute of Standards and Technology
MSE	Mean squared error
Nadam	Nesterov-accelerated adaptive moment
ReLU	Rectified linear units
RNN	Recurrent neural network
SELU	Scaled exponential linear unit
SVG	Scalable vector graphic
Tanh	Hyperbolic Tangent

THIS PAGE IS INTENTIONALLY LEFT BLANK

1 Introduction

Animating 2D rasterized graphics is challenging, as the image comprises pixels and does not contain any layer or shape information. To animate distinct parts of a graphic, they must be manually separated into independent entities. This represents a time-consuming process. Animating graphics or icons can also be repetitive, especially when creating many small animations with similar content. In an entirely manual workflow, the animations must be put into motion by an animator. In practice, this is done with several keyframes added by the operator and an automatically added interpolation between these keyframes.

According to Li et al. [1] most video prediction machine learning models either require multiple previous frames or focus on only predicting a single frame into the future. Recent efforts have mainly focused on improving classical keyframe interpolation [2], morphing pictures [3], or generating generic looping videos from photos [4]. Research specifically on animating 2D graphics has been sparse, but advances have been made in automatically animating Scalable Vector Graphics (SVGs) [5], [6]. SVGs are graphics described through XML-based vectors. As this is essentially just a markup language [7], accessing the graphic's information is straightforward. This enables machine learning models to focus on the data, circumventing the challenge of interpreting the graphics first. However, an SVG is a rather specific format, and translating rasterized graphics into SVGs is a complex task.

This thesis proposes a machine learning model, which takes two rasterized graphics and fills in the animation in-between. The work starts by preparing the Moving MNIST dataset [8] as will be discussed in chapter 4.1 and using several web scraping techniques to collect a sizeable training dataset of icon animations. Then I will present multiple machine learning models utilizing different algorithms, ultimately choosing the best approach for this specific task.

After a model is trained up, it can be fed with unknown input graphics, on which it will create novel animations. In contrast to research that takes only one input image [4], this proposed model takes two input frames to create an animation between them. This enables the user to create animations between two desired frames easier and faster with fewer manual steps. The machine learning model's predictions are forced to adhere to the constraints set by the two provided frames. This will work similarly to today's keyframe animation in motion graphics and 3D animation. But it can be used on regular raster graphics without manual work such as preparing the graphics, masking out parts manually, or animating them by hand.

When using the final machine learning model for generating predictions, the user only needs to provide the input and output graphics to create the in-between frames automatically. The quality of these generated animations should suffice for preview purposes or the improvement of a manual animation process.

2 Terminology

2.1 Machine learning terms

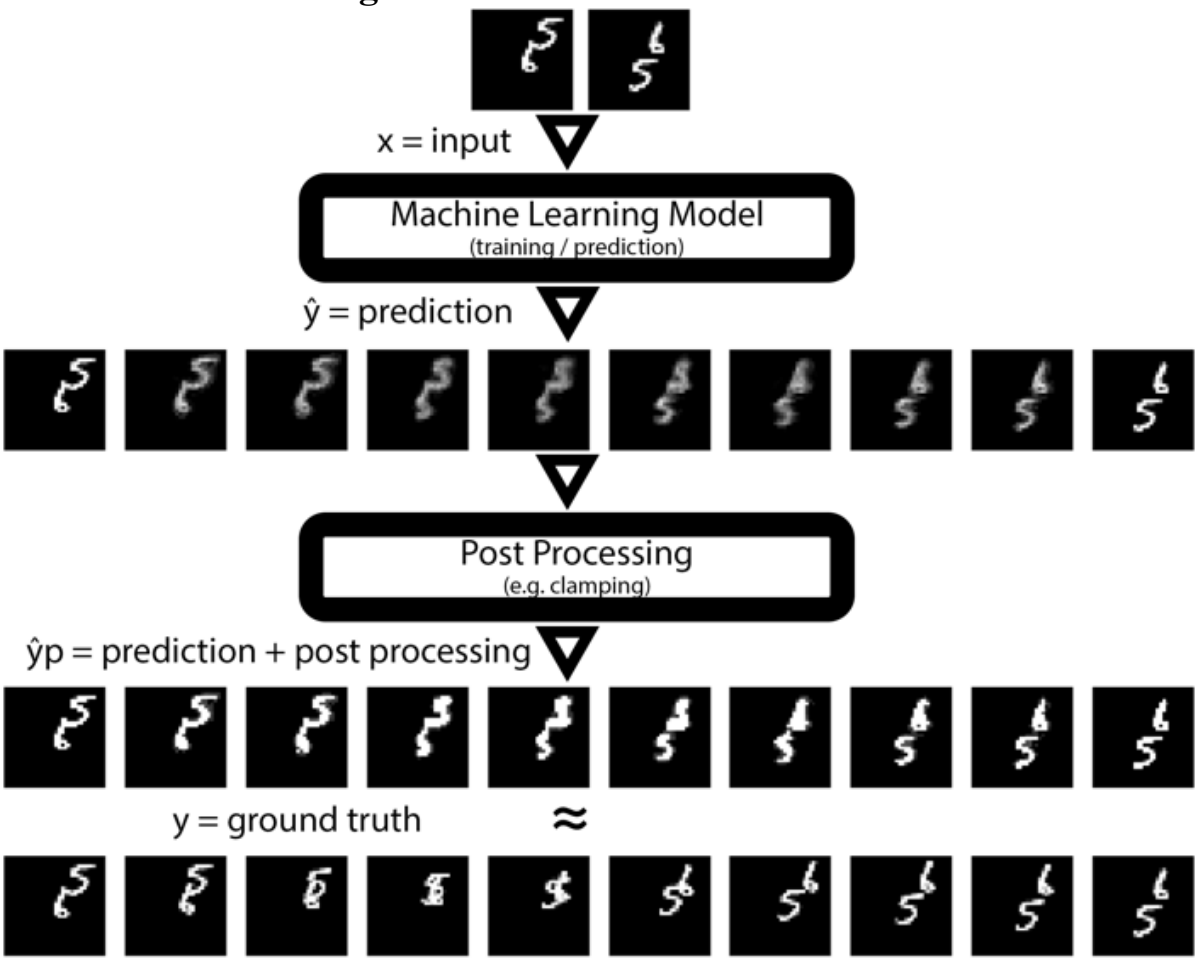


Figure 1: Graphical explanation of input data x, prediction ŷ, post-processed prediction ŷp and ground truth y.
Own image.

The goal is generating an animation from two rasterized input graphics. These input graphics compose the first and last frame of the generated sequence, hence the terms rasterized graphic, graphic, and frame will be used interchangeably.

A machine learning ***model*** is the final program after all work has been completed. It is the product that makes predictions at runtime. The ***architecture*** of a machine learning model describes its internal structure and building blocks.

As shown in Figure 1, the prepared input data used by the machine learning model for training and making new predictions is labeled ***x***. In the case of recursively trained models, as will be discussed in chapter 5.2.1, the input data only consists of the start and end graphics. In a conventional setting, the machine learning model expects the input data to be the same shape as the output video the model is asked to generate. In this case, the input data consists of sequences of the same length, but all frames except the input and output graphics are black.

The label for the ground truth is ***y*** and describes the expected outcome of a machine learning model's prediction. In this paper's context, *y* will refer to the original animation from the dataset. The results generated by the model are called ***ŷ*** or ***predictions***. The goal is for the predictions $\hat{y}$ to be as close as possible to the ground truth *y*. In this specific thesis, ***ŷp*** will be used for results that encompass any post-processing to differentiate them from $\hat{y}$. Post-processing primarily refers to cleaning up the predictions, as will be discussed in chapter 5.3.

Let $\chi_n, \hat{y}_n \in R^{w \times h \times c}$ be the n[th] frame in both the training and predicted sequence, respectively, with *w* denoting width, *h* denoting height, and *c* denoting the number of color channels of a frame. The final machine learning model will take two black & white 2D raster graphics as input frames $x = (x_0, x_{n-1})$ to predict an animation $\hat{y} = (x_0, \hat{y}_1, \ldots, \hat{y}_{n-2}, x_{n-1})$ in between, with *n* being the number of frames of the entire output sequence.

The restriction of black & white animations stems from limited available training data originating from a lack of research on generating 2D animations with machine learning. Since the training data requires no further human supervision or labeling, this model is ***self-supervised*** [9], [10]. This contrasts with classification tasks, where typically, a human operator or groups of people [11] must manually label the training data first. Especially for video frame prediction, the self-supervised approach is becoming increasingly popular [12], [13], [14]. It only requires the removal of frames on the original sequence to create training material where the model must predict the now missing frames. The original unaltered sequence serves as ground the ground truth, as demonstrated by Liu et al. in [15].

In Python, the training data is used in the form of multi-dimensional arrays, which will be referred to with regular parentheses; inside the machine learning framework TensorFlow the data is used in the form of ***tensors*** [16], which will be referred to with square brackets. Any conversion between these two is done automatically by the framework.

The terms ***training*** and ***fitting*** refer to how the model tries to find the best internal ***weights*** and ***biases*** to generate an output close to the ground truth *y*. These weights and biases are the values that will change during training and are the underlying mechanism by which machine learning models learn.

Fitting a model to its training data does not always yield the desired result of predicting a $\hat{y}$ that closely matches *y*. There are many possible reasons for this, such as using an ill-suited architecture, a training dataset that is too small, too homogeneous, or simply due to bugs. Unsatisfactory predictions can be broadly categorized into two categories: ***underfitting*** and ***overfitting*** [17]. Underfitting refers to a model that has not learned to predict data well. This will most likely manifest itself in a high loss value.

This will become visible in predictions that do not resemble *y*, either by being very noisy or containing very little data, as demonstrated in Figure 2. Overfitting is the opposite in that it fits the training data too well. Usually, this becomes evident with a validation loss much higher than the training loss. The training loss describes how well the model fits the training data, but also influences the training progress. The ***validation loss*** only measures the model's performance without affecting the training. Hence, validation loss is a better indicator of generalizability, as the goal is for the model to perform well on data it has never seen before.

Figure 2: Comparison of underfitting symptoms. ŷ1 = predicts only a small part of y, ŷ2 = predicts mostly noise.
Own image.

Figure 3 shows a simple, fully connected machine learning model with ten inputs (left), three ***hidden layers*** containing 16 neurons each (middle), and two outputs (right). A hidden layer describes a collection of neurons not visible from the outside. A neuron is depicted as a circle and can be described as a mathematical function which manipulates data. When using a library such as Keras, one hidden layer, such as the second row of neurons in Figure 3 could be composed of multiple operations. Keras refers to every operation inside a model as a layer. However, the layers that count towards the total number of hidden layers are only those capable of learning. Hence, when referring to layers in this thesis, only those capable of learning are meant. Everything else will be referred to as an operation or technique to avoid confusion.

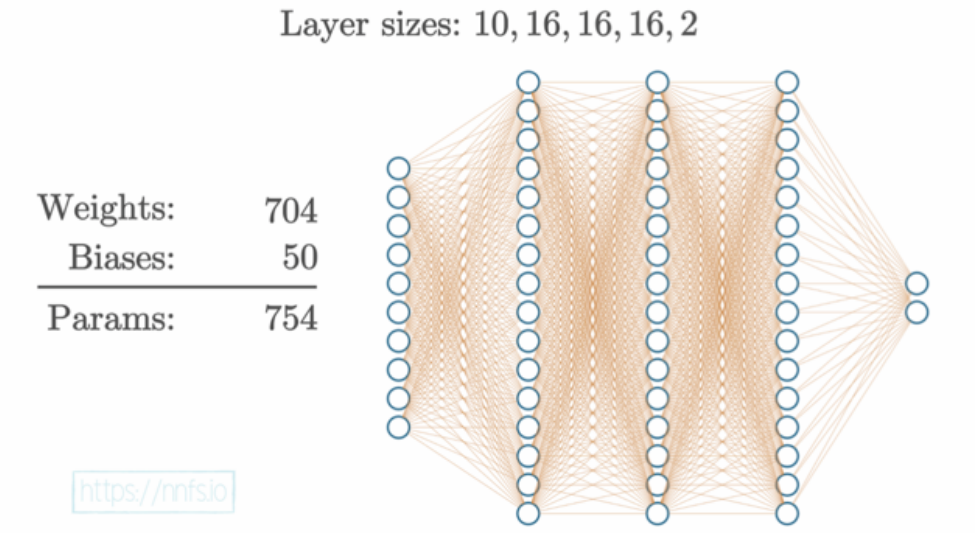


Figure 3: Example of a neural network with 3 hidden layers of 16 neurons each.
H. Kinsley and D. Kukieła, Neural Networks from Scratch in Python, 1st ed. (n.p.).

Every yellow line in Figure 3 connects two neurons and has an associated weight. Every circle is a neuron, and except for the ten input neurons, they each have a bias. In addition, every neuron aside from the input neurons uses an **activation function**. These activation functions can change the output of a neuron. By using a non-linear activation function, a neural network is able to map non-linear functions [18].

Different models work better with different activation functions. This is dependent on the model architecture as well as the dataset and problem to be solved. The models presented here will primarily rely on the ***rectified linear units*** activation function ***(ReLU)***,

$$f(x) = \begin{cases} x \; for \; x > 0 \\ 0 \; for \; x \leq 0 \end{cases} \tag{1}$$

the ***scaled exponential linear unit*** activation function (***SELU***),

$$f(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1) & for \; x < 0 \\ x & for \; x \geq 0 \end{cases} \tag{2}$$

and the ***hyperbolic tangent*** function (***tanh***).

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3}$$

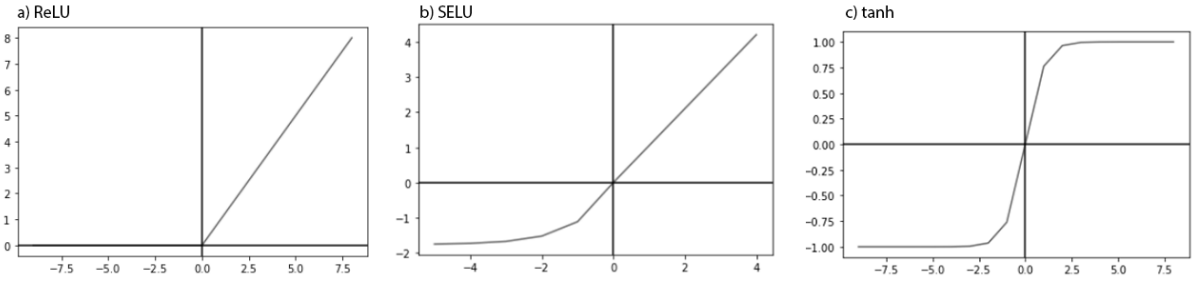


Figure 4: Comparison of ReLU (a), SELU (b) & tanh (c).
Sami, Thanga. "CNN and ANN Performance with Different Activation Functions like ReLU, SELU, ELU, Sigmoid, GELU Etc." Medium (blog), June 25, 2021. https://thangasami.medium.com/cnn-and-ann-performance-with-different-activation-functions-like-relu-selu-elu-sigmoid-gelu-etc-c542dd3b1365.

Figure 4 offers a graphical comparison of the ReLU, SELU and Tanh activation functions.

During training, the model will run for between 1 and n ***epochs***. An epoch describes a single training iteration during which the training data will propagate through the entire model once in a forward pass and a backward pass (backpropagation [19]). Different machine learning models require different settings for the number of epochs needed to converge. ***Convergence*** is achieved once the loss value does not decrease anymore, which signifies that the model is done with training. A small *loss* value suggests that $\hat{y}$ and y are close, i.e., the predicted animation looks very similar to the ground truth.

Conversely, a large loss value indicates poor performance. Optimally the model finds the global minimum. However, it can get stuck on a local minimum, as depicted in Figure 5. A model stuck on a local minimum will stop improving and needs to be restarted.

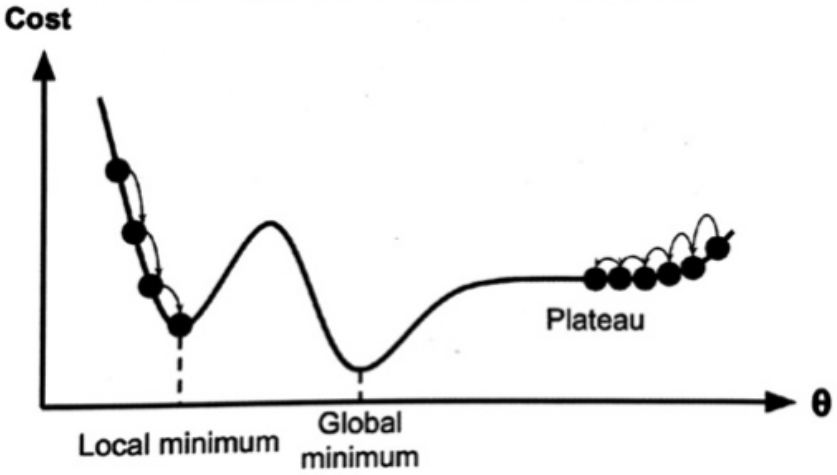


Figure 5: Gradient descent pitfalls.
Géron, Aurélien. Hands-on Machine Learning with Scikit-Learn, Keras, and TensorFlow. Second Edition. Sebastopol, CA: O'Reilly, 2019.

The optimizer is the algorithm modifying the network's weights and biases and uses a ***loss function*** to measure the difference between y and $\hat{y}$. There are many different loss functions available in TensorFlow. Most loss functions used for evaluating video predictions use metrics based on the image similarity [21]. One of the most widely used ones is the ***mean squared error*** loss (***mse***) [21].

$$\text{mse} = \frac{1}{N}\sum_{i=1} (y_i - \hat{y}_i)^2 \tag{4}$$

In equation (4) N stands for the total number of samples and i is the current sample. Essentially this function calculates the difference of the model's prediction $\hat{y}$ and the ground truth y, squares the result and then calculates the average across the entire dataset.

While all loss functions essentially judge a model's output $\hat{y}$ on how closely it matches y, they do this by prioritizing different factors. The mse loss function reports low errors by averaging all possible outcomes based on uncertainty. This can lead to blurry results by losing small details, leading to either entirely lost or blurry edges and outlines [20], [21]. Figure 6 demonstrates this with a deterministic animation at the top, which in this thesis's context corresponds to the ground truth y, and a probabilistic animation that is analogous to the predictions $\hat{y}$. The context frame (left) refers to the input frame, based on which the model predicts the probabilistic result. The probabilistic predictions of the machine learning model turn out blurry since the model is uncertain at what exact position the black square will be during future time steps. Hence it produces averaged, blurry results to cover all possibilities.

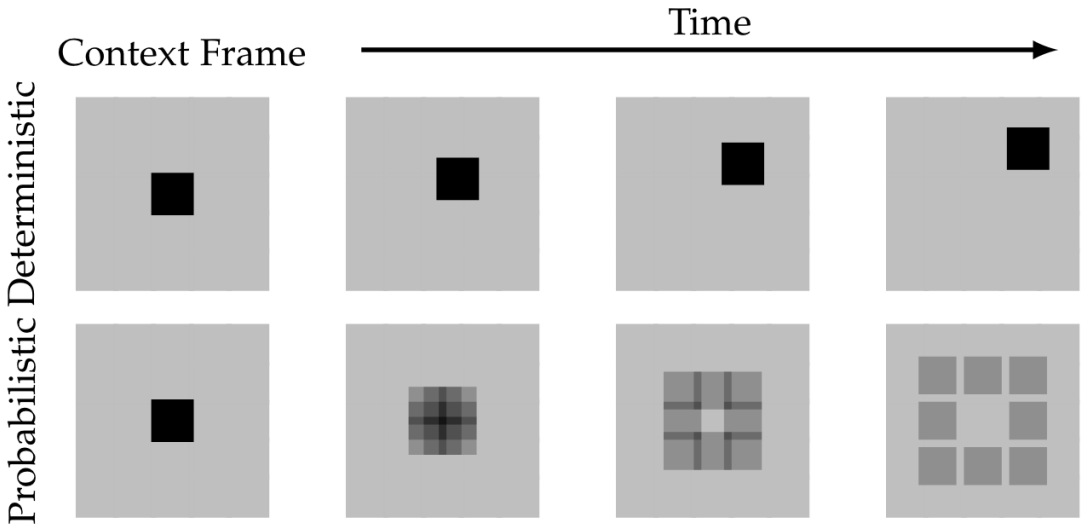


Figure 6: Deterministic animation (top), probabilistic animation (bottom). Darker areas correspond to higher probability outcomes.
Oprea, Sergiu, Pablo Martinez-Gonzalez, Alberto Garcia-Garcia, John Alejandro Castro-Vargas, Sergio Orts-Escolano, Jose Garcia-Rodriguez, and Antonis Argyros. "A Review on Deep Learning Techniques for Video Prediction." IEEE Transactions on Pattern Analysis and Machine Intelligence 44, no. 6 (April 15, 2020): 1–26. https://doi.org/10.1109/TPAMI.2020.3045007.

Solutions to the shortcomings of mse will be presented in chapter 5.3 about post-processing. It is also important to note that a good loss value for the Moving MNIST dataset should be below 0.01, as this value is also influenced by the large black portion of images that always match. Even poor results can achieve loss values below 0.02.

While the loss function calculates the loss value, the ***optimizer*** reduces this value during every epoch by adjusting the weights and biases of the model's neurons. There are different optimizers available, with contextual advantages and disadvantages. The most basic one is ***gradient descent,*** as shown in Figure 5. However, it has been effectively superseded by the ***stochastic gradient descent*** [22], so libraries like TensorFlow and Keras do not even offer classical gradient descent. This thesis relies mainly on the **adaptive moment estimation** (**Adam**) [23], which is one of the most widely used optimizers [24], and **Nesterov-accelerated adaptive moment estimation** (**Nadam**) [25], which proves to be more potent with some architectures. Both Adam and Nadam are gradient-based optimization functions[1].

All machine learning models presented in this thesis were trained in an ***offline*** fashion. This means that all the data for training is constant and the same between different architectures. Also, training is a single step in the pipeline; the models don't learn by making predictions outside of training. This makes for easy testing and comparing different models' performance. In ***online*** training, new data would continuously be used to keep training and improve the model. This would require a more complex setup.

[1] The following video offers a graphical explanation of gradient-based algorithms: [26].

2.2 Software & frameworks

Aside from the web scraping programs, all software for this thesis was written in ***JupyterLab*** [27]. JupyterLab is a web-based development environment that allows for very flexible working conditions. In machine learning research, where trial and error can take up a considerable part of the process, it is invaluable to easily separate and run specific code sections without running the entire program. Especially when considering large datasets that easily exceed 20 Gigabytes. The ***Anaconda*** Python distribution [28] makes this set-up even more flexible, as Python packages are not installed directly onto the local machine but into a separate Anaconda environment. This avoids having multiple packages, frameworks, and libraries conflict. It is easy to create a new environment when building a program that requires different frameworks or versions. These can also be backed up and sent to other machines. Exchanging these environments avoids having to install all the necessary dependencies manually.

All machine learning architectures and their training pipelines will be provided in the form of *JupyterLab* files, accompanied by the matching *Anaconda* environment. Furthermore, this thesis will use Google's ***TensorFlow*** framework [29] in conjunction with the open-source library ***Keras*** [30], which provides a more convenient interface. The most frequently used program library is ***NumPy*** [31]. It allows for the easy handling of vectors, matrices, and multi-dimensional arrays. Since NumPy's core is written in C, it offers significantly improved performance over native Python code but keeps Python's advantages regarding its ease of use. The final noteworthy library is Matplotlib [32] which will be used to display graphics. In addition, I have used the Netron [33] viewer to visualize parts of the presented machine learning architectures in more detail.

2.3 Hardware

All machine learning models presented in this thesis were trained on the same consumer-grade machine as listed in Table 1. This helps to provide comparable, real world, performance measurements.

Component	**Specification**
OS	Pop!_OS 21.04 (Ubuntu based)
CPU	I7-7700K @ 4.20GHz
RAM	32GB DDR4
GPU	NVIDIA GeForce GTX 1080 Ti @ 12GB

Table 1: OS & hardware used for training and testing all machine learning models in this thesis.

3 Related work

New research papers in video and image manipulation and animation with machine learning are regularly released. And by now, the research community has left the purely academic domain to provide novel solutions to real-world challenges.

3.1 3D animation

Approaches in 3D animation focus their efforts on teaching 3D characters to walk independently without having humans animate them. For example, [34] has achieved this by training the muscles and constraints of the characters. While the field of physics-based character animation is different from 2D animation, the end goal is still automation. The mentioned paper does not try to generate animations directly. Instead, the machine learning algorithm only gets to set constraints for the character's muscles. Over multiple iterations, the model adjusts these constraints to get closer to the desired walking animation. The constraints describe the degrees of freedom, meaning the number of independent ways a system can move. This might apply to 2D animations as well. Limiting the degrees of freedom could allow the generated animations to focus on specific areas, keeping other parts unchanged.

3.2 Keyframe animation

Other studies are researching better methods for classical keyframe interpolation as presented in [35]. The mentioned paper uses an autoregressive[2] recurrent neural network (ARNN) trained on the motion characteristics of physics-based example simulations. The proposed model can generate the motion between given keyframes while still following the style of the examples. Hence it is not just a simple linear interpolation between given keyframes but a more capable and motion-aware auto-completion.

3.3 Time step prediction

A pioneering study by [37] has developed a machine learning approach for precipitation forecasting, proposing a convolutional long short-term memory (ConvLSTM). They use previously observed sequences to forecast a series of fixed lengths into the future. As videos are nothing more than sequences of frames, the same approach could predict future frames in a video, as seen in a tutorial [38] for the Keras deep learning library.

Much effort by the research community continues to go into predicting future frames in various forms. Lotter et al. [39] present *PredNet,* which focuses on predicting one frame at a time but can be fed previous predictions recursively to predict multiple frames into the future. They treat previously generated predictions as inputs. *PredRNN* uses ten input frames to predict the subsequent 20 frames [40]. Some researchers present approaches to generate motion trajectories from a single image [41], while others

[2] An autoregressive network is one that produces its outputs in sequence [36].

manage to create animated output sequences from a single input image [1], [42], [43]. More papers present solutions that take multiple input frames for their future frame prediction, ranging from two input frames [44], [45], through five inputs [46] up to requiring ten frames [47], [48]. The LSTM model in [49] was trained to reconstruct ten input frames and then predict the next ten future frames.

An alternative approach was proposed by [50]. Their method consists of two steps. The first step is a frame reproduction to obtain features disentangled from content. The second step is the prediction of video frames using these disentangled features.

A very memorable paper has been [51], describing the model behind Google's *Bach Doodle*. It can take user-generated compositions, harmonize them, and fill in missing parts using *Coconet* [52]. *Coconet* is a generative model of musical counterpoint that can fill in music scores by predicting the missing notes.

3.4 Morphing

Papers researching the morphing or manipulation of pictures to create an animation between two different images, as in [3], are also of interest. The mentioned paper uses a convolutional neural network (CNN) to morph images of different semantic categories into one another. In their approach, the researchers look for *neural best buddies*, pairs of neurons that are mutual nearest neighbors.

Another relevant approach to animating photos is presented in [53]. That paper demonstrates Eulerian motion fields to animate pictures with fluid motion, such as flowing water and billowing smoke. The exciting part is that they use an image-to-image translation network. This enables the model to encode motion collected from online videos.

3.5 Optical flow

Optical flow dates back to the 1950s [54] and deals with the velocities of the movement of brightness patterns in images [55], [56]. It is already widely available in commercial video editing and visual effects software as a popular method for retiming video sequences [57], [58]. At the same time, it is still being actively researched and improved [59]. Furthermore, optical flow has also influenced the field of machine learning, with researchers using it as their benchmark [15], or incorporating it into machine learning models [60], [61].

Neural networks are opening up new avenues for optical flow, such as Walker et al. [62], who propose a model that can predict movement for a static image. But while optical flow can be used to create frames between two images, the requirement for stereo training data or existing flow fields as used in [63] is a significant downside, as creating training data is a non-trivial task. Since optical flow is mainly used to describe the motion between consecutive frames and relies on challenging to obtain ground truth data [64], [65], it is less suitable to generate animations between two disjointed input graphics.

3.6 Slow-motion

A topic closely related to the undertaking presented in this thesis is slow-motion, where the task is to generate new frames between two existing ones, in order to slow down a video [66]. The setup is quite similar, as it also has a start and end frame, where the model must predict what happens in-between these two. However, these approaches focus on regular videos and not animations. The thesis at hand presents methods that can create an animation between two different graphics, which is different from predicting an animation between two very similar video frames. However, these approaches show that a convolutional neural network is a very promising and popular candidate for the challenges presented in this thesis.

Furthermore, the AI-driven video interpolation of 2D sprite animations is getting attention with projects such as *Dain-App* [67], which can increase an animation's FPS based on Depth-Aware video frame interpolation as described in [2]. This approach synthesizes non-existent frames between the original frames using deep convolutional neural networks.

3.7 SVG animation

The most relevant papers that also served as inspiration for this thesis are [5] as well as [6], which present techniques for automatically animating SVGs (scalable vector graphics). The authors of [5] demonstrate a novel hierarchical generative network for SVG icon generation and interpolation. After learning how to reconstruct vector graphics accurately, they use a network that predicts sets of shapes in a non-autoregressive[3] fashion. It can perform interpolations and other latent space operations for animation purposes. This approach was further developed by [6], who built on top of [5]. They focus on further improving the quality of generated animations. Their work presents one unified pipeline from unprocessed input SVG to animated output.

However, SVGs are inherently different from rasterized graphics. The papers mentioned above can rely on reading out the vector information of the SVG graphics. This thesis will demonstrate an approach to achieving similar animation results not limited to SVGs and will work on generic rasterized graphics.

[3] A non-autoregressive network is one that produces its outputs in parallel [36].

4 Data acquisition & preparation

4.1 Moving MNIST

The primary dataset used in this paper is ***Moving MNIST,*** made public by the University of Toronto. It was created for the article Unsupervised Learning of Video Representations using LSTMs' [49] and is based on the classic handwritten digits MNIST dataset [68]. The mentioned paper uses different LSTM architectures to extend an input sequence of 10 frames into a target sequence of 20 frames. Figure 7 shows a single frame of the input sequence compared to the model's prediction.

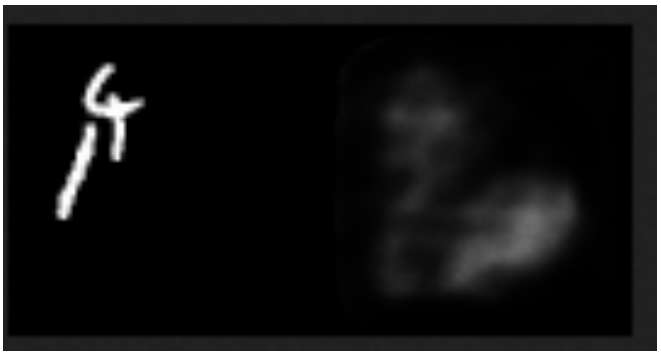


Figure 7: Long-term Future Prediction. Original animation (left), prediction (right).
Srivastava, Nitish, Elman Mansimov, and Ruslan Salakhutdinov. "Unsupervised Learning of Video Representations using LSTMs." http://www.cs.toronto.edu/~nitish/unsupervised video/..

Moving MNIST consists of 10,000 black & white sequences with 20 frames, each with a height and width of 64 pixels, containing moving handwritten digits. The input pipelines for the machine learning models in this paper turn the Moving MNIST dataset into 20,000 sequences of 10 frames and shrink the height and width to 32 pixels. Doubling the number of sequences increases the training performance while shrinking it to 32x32 pixels significantly reduces the memory requirements.

The focus lies on the Moving MNIST dataset as it has many similar examples, making it ideal for testing and comparing different architectures. It is also the only publicly available dataset that is both large enough and matches the task of generating 2D icon animations[4]. Also, most researchers in machine learning already know the classic MNIST dataset. "*This set [MNIST] has been studied so much that it is often called the 'hello world' of Machine Learning* […]" [70]. In comparison, the Moving MNIST dataset is not as well-known and has only been used in a few research papers[5]. However, it is easy to understand Moving MNIST if one already has experience with the classic MNIST dataset, even though the complexity of Moving MNIST is considerably higher. Also, both MNIST and Moving MNIST are freely available to download. Hence it helps

[4] Oprea et al. have presented a comprehensive list of datasets for video prediction [69].
[5] Google scholar yields 66.400 results for "MNIST" and only 405 for "Moving MNIST" as of 12.05.2022.

with reproducibility as well as comparability. The movements of the Moving MNIST animations are also ideal; they consist of simple translations of digits that often overlap. On the one hand, this makes it a very simplistic dataset. On the other hand, it would be laborious to reproduce these results manually. This is especially true for examples that overlap. If two digits overlap at any point during the animation, some dedicated reconstruction work is required to separate them cleanly when talking about manual animation.

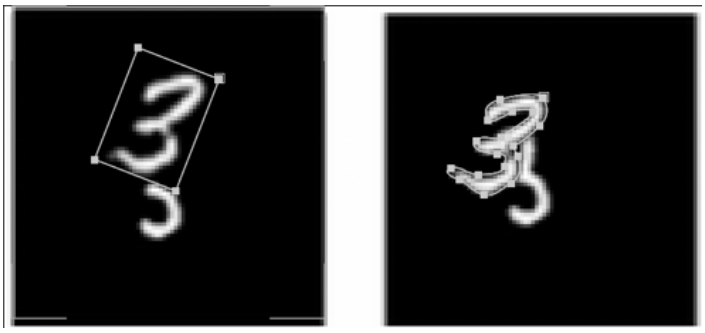


Figure 8: MNIST digits masked by hand. (Left) mask drawn with little regard for overlaps; (Right) mask drawn with attention to overlaps.
Own image.

Figure 8 compares a simple rectangular mask with no regard for overlap to a more carefully drawn mask. The left one causes one of the two digits to be cut off but requires less work. The right one avoids this problem but requires more vertices that must be created manually. Depending on the exact animation, these masks need at least two manually set keyframes in the best case and one manually set keyframe for every frame of the animation in the worst case. This can lead to the amount of work scaling with animation length. Hence the mask on the right takes much more time and effort.

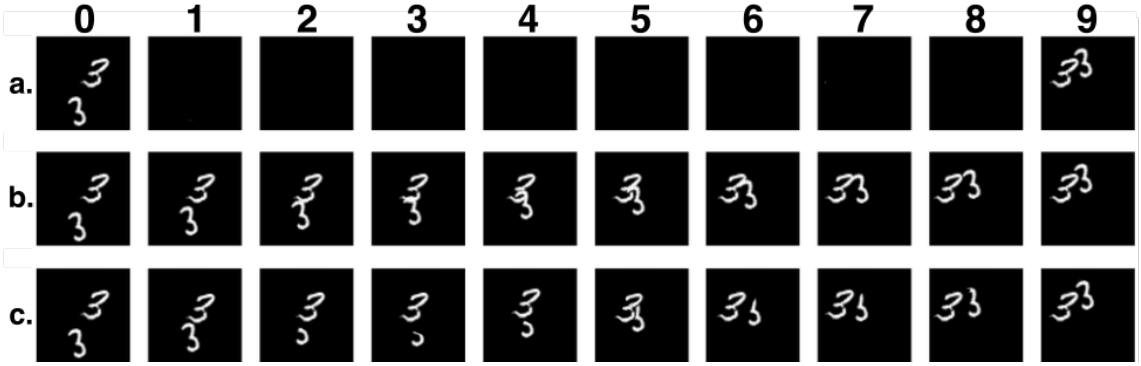


Figure 9: Moving MNIST digits animated by hand. a. two input frames; b. ground truth, c. manually masked and animated.
Own image.

Figure 9 demonstrates how a ten-frame animation with simple rectangular masks (c.) compares to the original animation (b.); (a.) depicts the available input frames from which the entire animation (b.) must be reconstructed by animating and masking the digits of the input frames. Frame 4 of (c.) shows the rectangular mask overlayed.
When looking at frame 2, a large part of the bottom three in (c.) is missing. Improving the masks in (c.) requires considerably more time.

4.2 Animations

While the focus lies on the Moving MNIST dataset, which is already challenging, web scraped animations will be used to better demonstrate generalizability. This dataset will be referred to as GIF or custom GIF dataset throughout this thesis.

Creating a functional machine learning model requires a large amount of training data. I have focused on finding animated GIFs and vector-based Lottie files to simplify this process. GIFs are a widely used standard web file format for images and animations, dating back to the 1980s. Lotties [71], on the other hand, are a novel JSON based file format made especially for animations. A Lottie describes an animation programmatically through code, as shown in Figure 10. However, it can only be played back with a specific Lottie player.

The reason for choosing GIFs and Lotties is that both formats are widely used for very short icon animations. This reduced the pre-processing required to find the desired part of an animation and narrowed down the subjects instead of using videos that are usually longer and less focused on small icons.

Figure 10: JSON representation of a Lottie animation. "JSON Lottie Files." Accessed January 12, 2021. https://lordicon.com/api/library/icons?loadData=1&premium=0&categoryId=0.

4.2.1 Web Scraping

Not all machine learning models work well with the same amount of training data. Hence an easy-to-use pipeline that allows for the easy and quick acquisition of more training data is beneficial. This resulted in multiple Python web scrapers and pipeline scripts.

4.2.1.1 Beautiful soup scraper

The first Scraper is a Python script utilizing the *Beautiful Soup* framework [72]. This allows downloading a website's entire HTML code with a simple GET request. Once downloaded, the file links to the animations can be extracted to download the data.

4.2.1.2 Selenium Scraper

The second Scraper needed a different approach as some websites use JavaScript to load their content dynamically, so it was not directly available. This made it impossible to read out the website's complete HTML code. Instead, I used *Selenium* [73], which works similarly to a human user. It opens the website in a (headless) browser and waits for the website to load all content. This script also works with websites that use infinite scrolling.

4.2.1.3 Direct download

A vanilla Python script suffices on some websites, where the files are directly accessible via URL. It simply iterates through the URL's last digit[6]. In this case, no frameworks were necessary.

4.2.2 Data processing

After downloading data, it still needs processing before being fed into any machine learning model. The downloaded data consists of two different formats with Lottie and GIF files and differences in dimension and color channels. Hence all downloaded animations must be normalized into one coherent format that the machine learning models can understand. For this paper, I have used NumPy arrays of predefined dimensions.

4.2.2.1 Lottie to GIF converter

Lottie files are very different from GIFs and other video and picture codecs, as they are made up of JSON code, not frames. Since this thesis presents machine learning models that create animations from input raster graphics, I had to convert the Lottie files. They were first converted into GIFs to process all the scraped data with the same pipeline in the following steps. The Lottie to GIF converter utilizes the *puppeteer-Lottie* npm library [74], which uses the *Gifski* [75] GIF converter library to turn the Lottie files into standard GIFs.

4.2.2.2 GIF to image sequence converter

During my research, I discovered that working directly on GIFs is problematic since they are prone to suffering from artifacts, displaying corrupted information, and breaking entirely when accessed by Python. So instead of using GIFs directly, I built a pipeline to convert them into png image sequences. This converter uses the command line tool *FFmpeg* [76] to convert the GIFs. The image sequences still keep their original dimensions at this point, but every GIF's frame rate gets converted into 25 fps.

[6] E.g., website.com/animations/0.

4.2.2.3 Image sequence to NumPy converter

Inside Python, I work with NumPy arrays of fixed dimensions such as:
(Total number of sequences, number of frames in a sequence, image width, image height, color channels).
The conversion script reads all image sequences and converts them into NumPy arrays. In this step, all sequences get normalized into the same format. Every array representing a sequence is 30 frames long. This is achieved by either cutting off longer sequences or repeating those that are too short.

All arrays must also be of the same dimensions, as ragged arrays[7] cause problems and are harder to work with. I have used the *open cv* [78] library to convert the images to 64x64 pixels. This size aligns with the animations of the *Moving MNIST* dataset and allows the use of the same input pipelines within all machine learning setups. This size can still be reduced if memory constraints are prohibitive when loading them into the training pipeline.

The result is one large file containing the NumPy array. As one NumPy file can be easily larger than 10GB, I have created multiple smaller files that can be loaded individually. The maximum file size has practical limitations set by the available RAM.

4.2.2.4 Data augmentation

Machine learning results can be vastly improved by utilizing data augmentation [79]. This has the effect of creating many slightly different variations of the same data point. In the case of animations, it could mean flipping them horizontally, vertically, or temporally and zooming in or rotating frames. But the focus lies on Moving MNIST, which works reasonably well without special data augmentation. The only step taken is the division of animations into 10-frame sequences.

Since the GIF dataset is much more heterogeneous, with vastly different animations and content, they will make use of data augmentation. This will take the form of reversing them temporally or flipping them both horizontally and vertically. The 30-frame long animations will also be turned into three ten-frame animations each.

[7] Ragged or jagged arrays are nested arrays containing arrays that can be of different lengths [77].

5 Methods

5.1 Pre-processing

Most images and videos are made up of pixels in the range of [0, 255] that must be rescaled for machine learning. Depending on the model's preferences, it gets either rescaled into the range of [0, 1].

$$data = \frac{data}{255} \tag{5}$$

Or it gets rescaled into the range of [-1, 1].

$$data = \frac{data - 127.5}{127.5} \tag{6}$$

In most machine learning tasks, a critical step is to separate the training data from the validation data. The validation data is used to assess the model's performance on yet-unseen data, as the goal is for a machine learning model to generalize well on new data. If the model has already seen the validation data during training, it can reproduce it to fake good results, achieving a low loss value. According to [70], it is common to use 80% of the data for training and hold out 20%.

In addition to the validation set used to measure the model's performance, I create a test set that will only be used for making predictions after all training has been concluded. All predictions presented in this thesis are taken from that test set. Hence the model has never seen those exact samples before. Data augmentation will be avoided as best as possible to ensure no blending between training, validation, and test sets.

The entire dataset will be split into a training set containing 80%, a validation set containing 15%, and a final test set containing 5% of the data.

5.2 Peri-processing: Recursive Triads

I discovered that using a recursive approach for training and prediction yields better results for several different machine learning architectures.

5.2.1 Introduction of recursive triads

The conventional approach of predicting an animation between two frames provides the model with frame 0 and frame *n*-1 and predicts all frames in-between in one go. Some networks also allow feeding in previous predictions recursively, although this usually still refers to consecutive frames as in [39].

Since arrays in Python start at Zero, the last frame of an animation will be denoted as *n*-1. My approach of recursive triads divides the dataset into *n* triads, each containing three frames. This approach uses multiple runs to generate one frame per run when used for predictions. Every triad of the ground truth consists of a start-frame, the frame in the middle, and an end-frame. The training data's triads also consist of a start and end frame, but the middle frame is black. The model is trained on predicting content for the black frame in the middle.

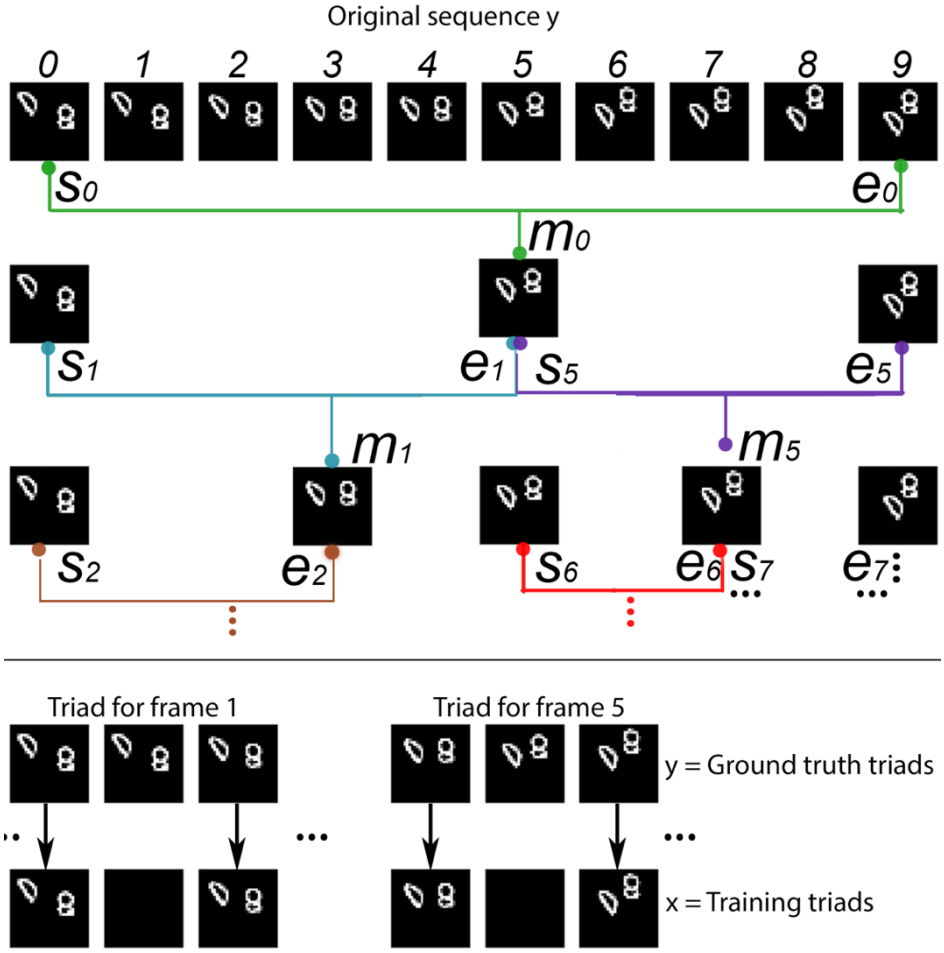


Figure 11: Converting a 10-frame animation into triads. Graphic only depicts the start of the algorithm.
Own image.

Figure 11 shows how these triads are generated recursively, based on two example triads of both the ground truth and training data. For a ten-frame long animation, this means generating ten triads. A reduced version of the Python code of the recursive triads algorithm only showing the generation of the ground truth triads can be found in the appendices chapter 10.1.

The animation length in frames being *n*, this algorithm's goal is to generate *n* triads each consisting of a start frame at index *s,* a middle frame at index *m*, and an end frame at index *e* of the original animation. This works in two directions, by calling the recursive function twice and updating the function parameters. For the initial function call, the indices will be s_0 = 0 and e_0 = n-1, resulting in $m_0 = e_0 - \left\lfloor \frac{e_0 - s_0}{2} \right\rfloor$.
Subsequent calls for $i \in \mathbb{N}, 0 < i < n - 2$ will contain two function invocations. Once a triad for the middle frame at index: $m_{i-1} - \left\lfloor \frac{m_{i-1} - s_{i-1}}{2} \right\rfloor$, followed by a triad for the middle frame at index: $e_{i-1} - \left\lfloor \frac{e_{i-1} - m_{i-1}}{2} \right\rfloor$.
For *i = n-2* there is a single call with $s_i = i$, e_i = *n*-1, and $m_i = n - 1$ creating the triad corresponding to the original end frame. For *i = n-1* there is also a single call with s_i = 0, e_i = 1, and $m_i = 0$ creating the triad for the original start frame.

For a ten-frame long animation, this approach creates ten triads containing frames corresponding to indices of the original animation as follows in Table 2: Triads containing indices corresponding to frames in the original animation.

:

Triad #	**(s_i, m_i, e_i)**
0	(0, 5, 9)
1	(0, 3, 5)
2	(0, 2, 3)
3	(0, 1, 2)
4	(3, 4, 5)
5	(5, 7, 9)
6	(5, 6, 7)
7	(7, 8, 9)
8	(8, 9, 9)
9	(0, 0, 1)

Table 2: Triads containing indices corresponding to frames in the original animation.

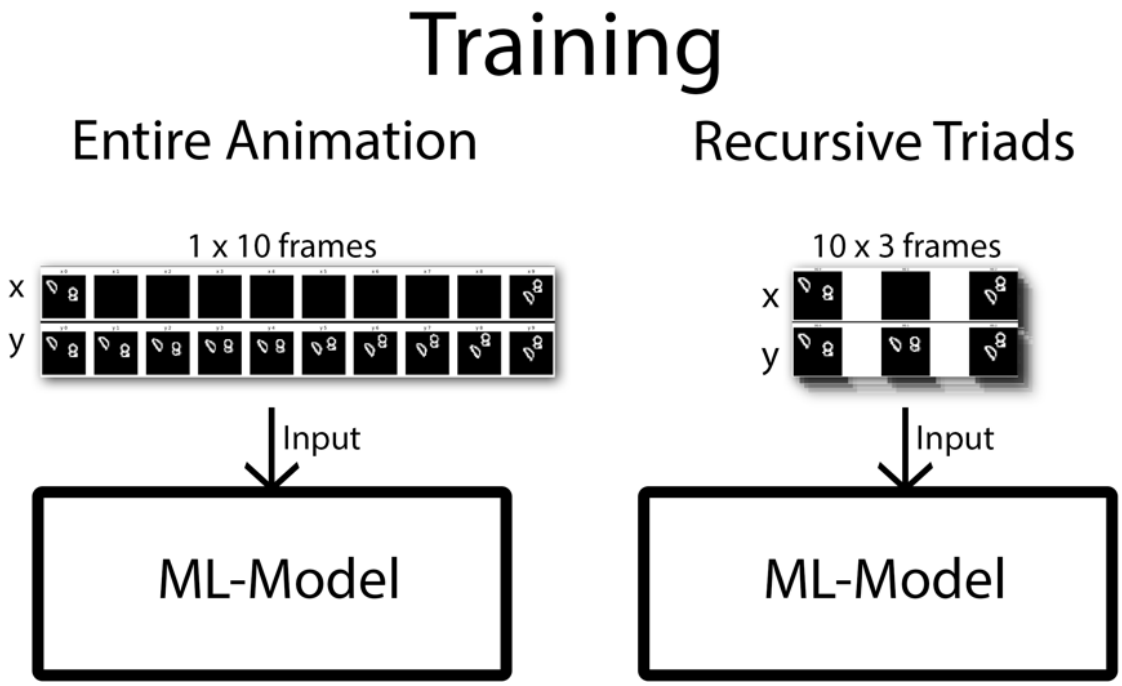


Figure 12: Training on the entire animation at once (left) vs. recursive triads (right). Own image.

Figure 12 shows the difference between the classic approach of training the model on the entire animation in one go and training it on multiple triads for a 10-frame long animation. A single black & white 10-frame animation has the NumPy array shape of (1, 10, 32, 32, 1) corresponding to:

Element #	**Item**	**Example**
0	Number of animations	1
1	Frames per animation	10
2	Frame width	32
3	Frame height	32
4	Color channels	1

Table 3: Elements of a NumPy array for training a conventional model.

The triads are of a similar shape with the addition of triads instead of single frames per animation. The shape for a single animation of triads is (1, 10, 3, 32, 32, 1), corresponding to:

Element #	**Item**	**Example**
0	Number of animations	1
1	Triads per animation	10
2	Triad (Three Frames)	3
3	Frame width	32
4	Frame height	32
5	Color channels	1

Table 4: Elements of a NumPy array for training a recursive model.

Every triad consists of the start frame, the frame in the middle, and the end frame. This approach adds a non-trivial amount of complexity and overhead to the run-time and RAM usage, although this can be further optimized.

Using the triads approach in the prediction phase is similar to the training phase. With the conventional method, the model is presented with the start and end frame and must predict all frames in-between in one go. The triad approach presents the model multiple times with start and end frames. But the model only predicts a single frame for each input triad. The predictions are processed recursively until all frames have been predicted. For example, the first prediction will serve as the end frame for the second prediction. Figure 13 offers a graphical comparison of both approaches for a 10-frame long animation.

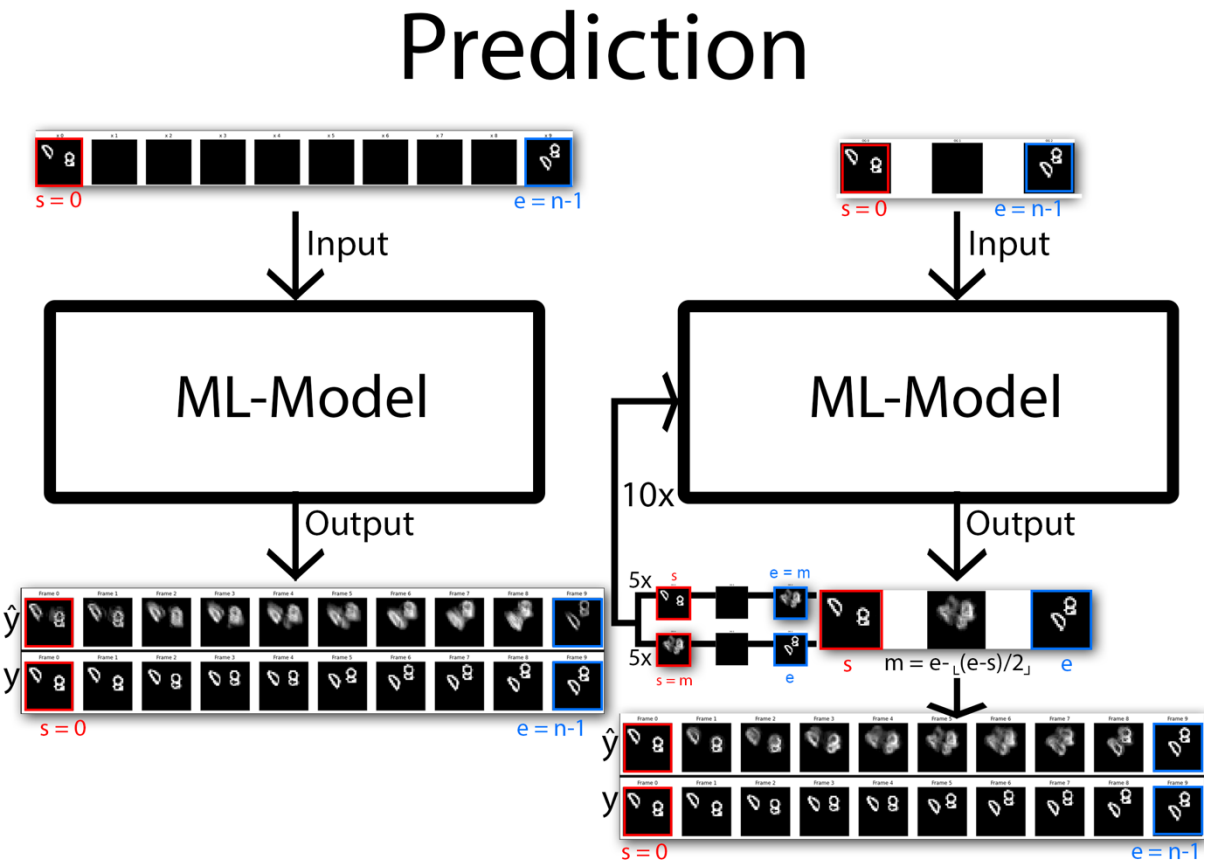


Figure 13: Predicting the entire animation at once (left) vs recursive triads (right). Own image.

Despite the extra overhead and complexity, once developed, the approach of recursive triads offers an easy opportunity to improve the predictions.

5.2.2 Comparison of conventional and recursive CNNs

Figure 14 and Figure 15 show the predictions generated by the same CNN architecture in the top row compared to the ground truth in the respective bottom rows. The only difference between the two CNNs is that in Figure 14, all frames were predicted at once, while Figure 15 uses the recursive triads approach. CNNs are further discussed in chapter 5.7.3.

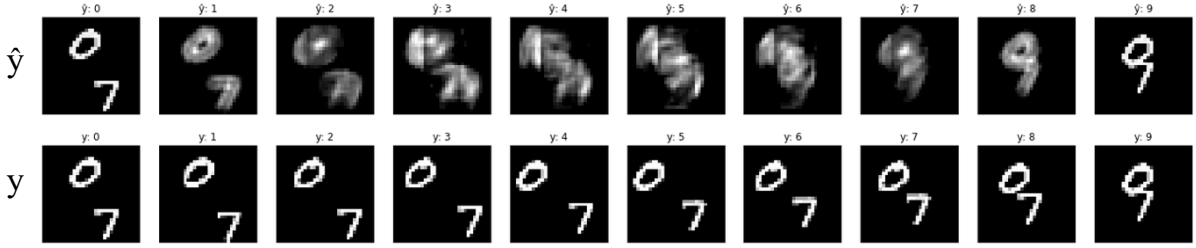


Figure 14: CNN trained conventionally; prediction (top), frames 0 and 9 are the original inputs; ground truth (bottom).
Own image.

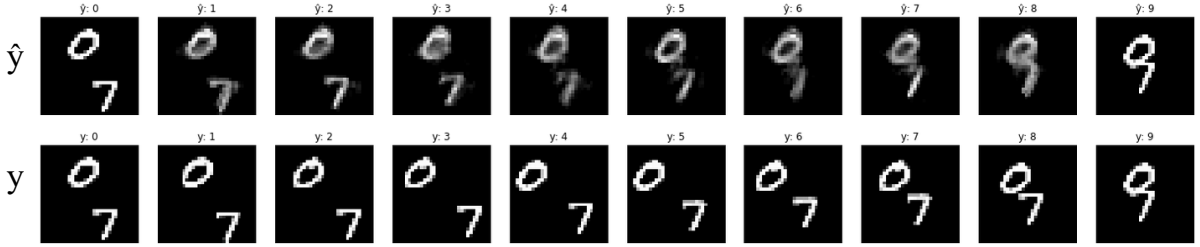


Figure 15: CNN trained using recursive triads; prediction (top), frames 0 and 9 are the original inputs; ground truth (bottom).
Own image.

The results of the conventional CNN in Figure 14 start degrading very quickly. The original digits become unrecognizable after only three frames. The same CNN model generates much clearer results when using recursive triads, as shown in Figure 15.

5.2.3 Comparison of conventional and recursive MLPs

Figure 16 and Figure 17 illustrate the same difference for a multi-layer perceptron (MLP) architecture, as showcased in chapter 4.1. The model itself is less promising than the CNN shown above, but the difference between the conventional and recursive approaches is even more stark.

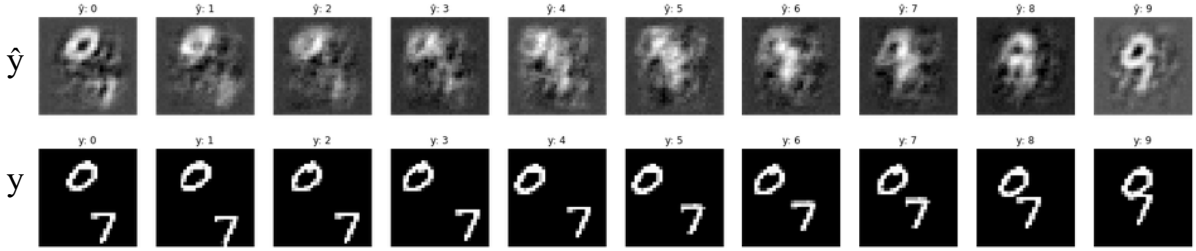


Figure 16: Multi-layer perceptron trained conventionally; prediction (top), ground truth (bottom).
Own image.

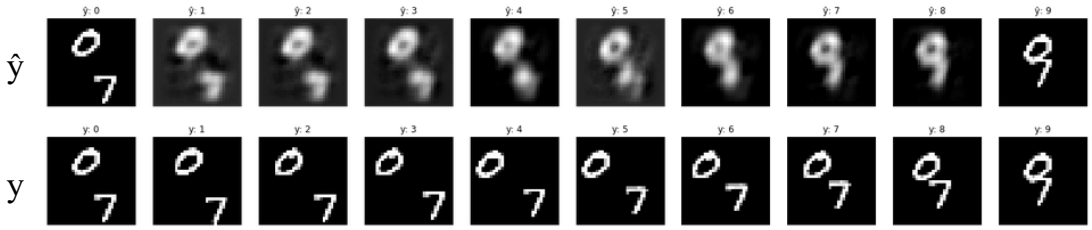
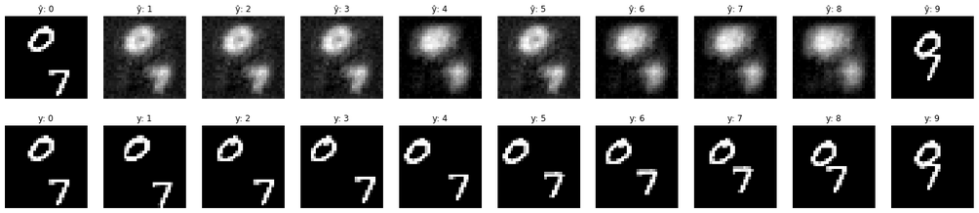


Figure 17: Multi-layer perceptron trained using recursive triads; prediction (top), ground truth (bottom).
Own image.

The predictions generated by an MLP using the conventional approach for training and predictions in Figure 16 are extremely noisy and hard to make out. The same model trained on triads that makes its predictions recursively has massively improved results, as seen in Figure 17.

5.2.4 Comparison of conventional and recursive RNNs

It is necessary to note that this approach cannot be blindly used on just any machine learning model. Recurrent architectures that rely heavily on timesteps tend to fail, as they are built from the ground up to work with sequences. In the case of an RNN (and other recurrent architectures, which will be discussed in chapters 5.6.1 to 5.7.1), the recursive approach cannot improve the model's predictions. The generated frames simply degrade the first input frame, staying in place without any actual translational motion. Figure 18: Recurrent neural network trained conventionally; prediction (top), ground truth (bottom).
Own image.



shows the outputs of a conventional recurrent neural network. The results very quickly lose any temporal coherence.

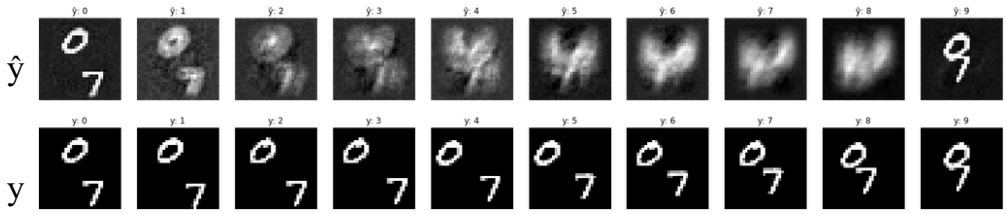


Figure 18: Recurrent neural network trained conventionally; prediction (top), ground truth (bottom).
Own image.

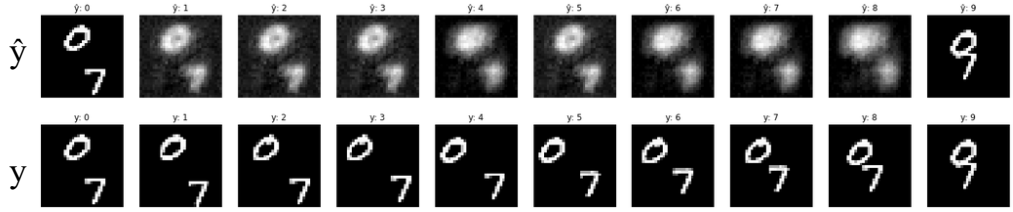


Figure 19: Recurrent neural network trained using recursive triads; prediction (top), ground truth (bottom).
Own image.

Figure 19 has cleaner results generated with the recursive approach. But frames one to eight are just degraded copies of the first input frame.

5.2.5 Conclusion regarding recursive triads

While the recursive triads offer a convenient option for improving machine learning models in this context, they must be compared to a traditional approach to ensure that they work with the specific architecture. Otherwise, there is a possibility of disregarding architectures that might have worked with a conventional method.

One key advantage of recursive triads is that neural networks generally get trained on a specific shape of data[8]. In a conventional approach, the model can only predict animations of ten frames if that is what it was trained on. With the recursive triads, however, the length of the animation is much less relevant. The model only ever predicts one frame at a time. The size of the sequence is a hyperparameter of the recursive triads algorithm and decoupled from the machine learning model itself. This means that the same model can predict animations of any length. Of course, the quality will still degrade with longer animations.

5.3 Post-processing

While using recursive triads is a somewhat involved process to be set up the first time, post-processing can be easily applied after every prediction. It can be used in conjunction with any approach and is applied as a separate step. As this thesis aims at creating animations from two input graphics for preview purposes, it would not suffice to rely only on one machine learning model to do all the heavy lifting, as it would be far too limiting and suffer from blurriness introduced by the *mse* loss function. Combining different approaches can produce a better result and alleviate the end-user's work, as he does not need to clean up the results manually.

Many papers work around the problem of blurry results via more comprehensive and complex architectures. Multiple strategies were presented by Matthieu et al. [20]. But in the context of animating small 2D graphics, a simpler post-processing strategy suffices without making the core machine learning architectures more complex.

[8] I.e., the array dimensions of the input data.

5.3.1 K-Means

"*The goal is to group similar instances into clusters*" [70].

K-Means is an algorithm that can quickly cluster a dataset and was first used in pulse-code modulation (PCM) [80]. As the Moving MNIST dataset is essentially black & white with some aliasing, K-Means with a *K* of two or three will cluster the predictions and eliminate most of the noise. Keras defaults to an improved version, K-Means++, which uses a smarter initialization to avoid convergence at a sub-optimal solution [81].

The basic K-Means algorithm:
„1. Arbitrarily choose an initial k centers C = {c1, c2, · · · , ck}.
2. For each i ∈{1, . . . , k}, set the cluster C_i to be the set of points in X that are closer to c_i than they are to c_j for all j ≠ i.
3. For each i ∈{1, . . . , k}, set c_i to be the center of mass of all points in C_i: $c_i = \frac{1}{|C_i|}\sum_{x \in C_i} x$.
4. Repeat Steps 2 and 3 until C no longer changes.“ [81].

The K-Means++ algorithm as explained by D. Arthur and S. Vassilivitskii [81]:
“[...] let D(x) denote the shortest distance from a data point to the closest center we have already chosen. Then, we define the following algorithm, which we call k-means++.
1a. Take one center c1, chosen uniformly at random from X.
1b. Take a new center ci, choosing $x \in X$ with probability $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$
1c. Repeat Step 1b. until we have taken k centers altogether.
2-4. Proceed as with the standard k-means algorithm.
We call the weighting used in Step 1b simply ‘D^2 weighting’.“

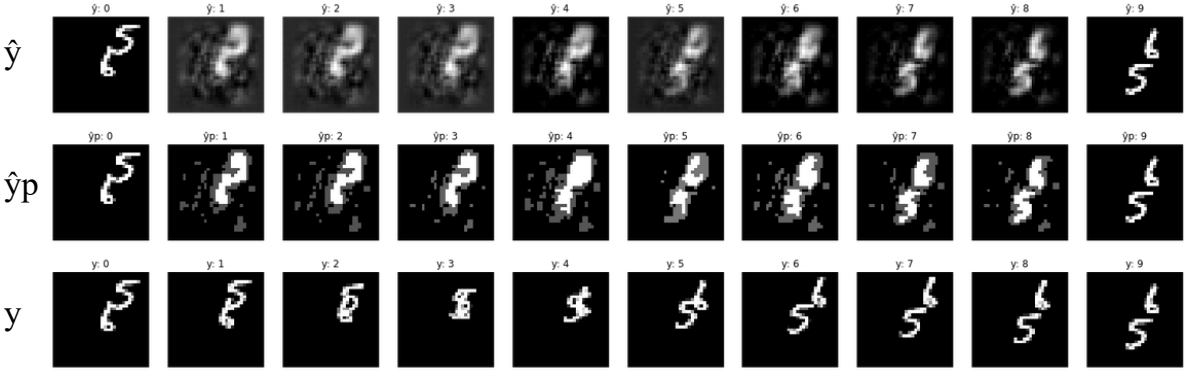


Figure 20: MLP trained using recursive triads; ŷ (top), ŷp (middle, K-Means++; k=3), y (bottom).
Own image.

Figure 20 shows the result of a multi-Layer perceptron in *ŷ* and the same after going through the K-Means++ algorithm in *ŷp*, with a value of three for the number of clusters *k*. This reduces the grayscale pixel values to only three remaining options: black, white, and one shade of grey. While this approach reduces the noise and can bring the grey pixel values closer to *y,* the quality is disappointing. When using a value of two for the clusters *k,* the results lose more noise, but it does not increase their recognizability, as seen in Figure 21.

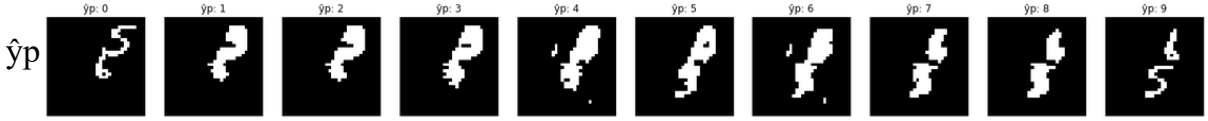


Figure 21: MLP trained using recursive triads; ŷp (K-Means++; k=2).
Own image.

```python
from sklearn.cluster import KMeans

def kmeans_cluster(data):
    y_hat_p = np.zeros(data.shape)
    ary = np.array(data)
    for i, frame in enumerate(ary):
        test = frame
        vectorized = test.reshape(-1,1)
        vectorized = np.float32(vectorized)
        kmeans = KMeans(n_clusters=3).fit(vectorized)
        segmented_image = kmeans.cluster_centers_[kmeans.labels_]
        segmented_image = segmented_image.reshape(test.shape)
        y_hat_p[i] = segmented_image
    return y_hat_p
```

Figure 22: Python code for applying K-Means++ algorithm.
Own image.

Considering that K-Means requires a training step itself, it presents too much work and overhead for too little gain. Figure 21 provides a code snippet using Keras' K-Means implementation. The fourth line inside the for loop confirms that this algorithm needs to call a fit[9] function for training.

5.3.2 Clamping

K-Means is a machine learning approach in and of itself but does not yield impressive results for cleaning up generated animations. A much simpler procedure is to clamp the results. The presented clamping algorithm is much less demanding as it does not require any training for itself and can be easily tweaked with only two parameters. It consists of a threshold *theta* and an *epsilon* around *theta*. *Theta* denotes the cut-off point to set small values to zero and large values to one. This greatly reduces the noise in black and white images. Both the theta and epsilon values were found through manual experimentation.

$$\hat{y}p = \boldsymbol{f}(\hat{y}) = \begin{cases} \mathbf{0}, & \hat{y} < \theta - \varepsilon \\ \mathbf{1}, & \hat{y} \geq \theta + \varepsilon \\ \hat{y}, & \theta - \varepsilon \leq \hat{y} < \theta + \varepsilon \end{cases} \tag{7}$$

```python
def clamp_image_array(y_hat, theta=.3, epsilon=.1):
    y_hat_p = np.where(y_hat < theta - epsilon, 0, y_hat)
    y_hat_p = np.where(y_hat_p >= theta + epsilon, 1, y_hat_p)
    return y_hat_p
```

Figure 23: Python & NumPy code for clamping animations.
Own image.

Figure 23 provides a Python implementation of the clamping algorithm as presented in equation (7).

[9] As discussed in chapter 2.1, to fit a machine learning model means training it.

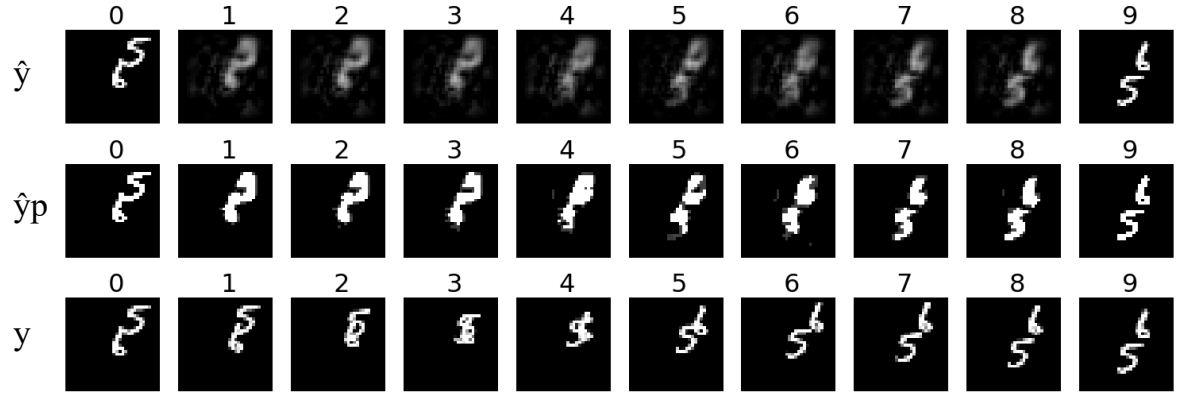


Figure 24: MLP trained using recursive triads; ŷ (top), ŷp (middle), y (bottom). Own image.

Figure 24 compares the original predictions $\hat{y}$ of a multi-layer perceptron with the clamped results *ŷp.* While a simple clamping algorithm cannot turn unsatisfactory results into something impressive, it can nonetheless add some improvement on top. The main shortcoming is blurred digits, where noise resides in inappropriate places that cannot be removed without damaging other digit areas. A poignant example of this is frame 6 in Figure 24. The blurred noise in *ŷp's six* cannot be reduced, without further degrading the five, as the pixel values in both digits would fall below *theta*. Hence a cleaner six results in an essentially removed five.

Overall, the presented clamping algorithm provides slightly better performance for the Moving MNIST dataset than K-Means++ while introducing less complexity and overhead. The *epsilon* value helps smooth over the results by retaining details that help render the digits more recognizable than in the K-Means approach with *k=2*, without leaving in a large amount of noise as in K-Means with *k*=3.

5.4 Comparison and evaluation of different hyperparameters

When building a machine learning model, one is presented with near-infinite possibilities of using and combining different hyperparameters[10]. The most significant parameters I have used are the number of layers, the number of units[11] per layer, the activation function for hidden layers, the activation function for the output layer, adding different types of layers, and the optimizer.

Trying out all combinations manually is not practical. Instead, I utilized the hyperparameter optimization framework *KerasTuner* [83]. Using this framework makes it easier to automatically try out lists of different hyperparameters that will be compared by the achieved loss values. It is important to note that a low loss value indicates only an average over the entire dataset used for training. It is possible, even likely, that a model with a slightly lower loss value still produces some outputs that look worse than

[10] Hyperparameters are parameters that can be manually adjusted to control a machine learning model's learning process. They are not learned through training [82].
[11] The units of a layer refer to the number of neurons or filters, depending on the type of layer.

a model with a higher loss value. But on average, the predictions will likely be better if the model has not overfit the data.

Using KerasTuner makes it easier to narrow down the architecture to a promising setup. This avoids building architectures by randomly guessing while building the intuition for what works in a specific context. But even when only running on half a dataset and with a partial set of hyperparameters to compare, this process can take up to 40 hours of pure run time, as was the case for the convolutional variational autoencoder presented in chapter 5.7.8.

After the initial search for hyperparameters, I narrow down the hyperparameters for a second run of the tuner. Depending on the results, I will manually try different hyperparameter setups to commit to a final one.

5.5 Multilayer perceptron (MLP) architecture

A multilayer perceptron is one of the most basic but also one of the most widely used types of neural networks [84]. It consists of one input layer, one or multiple hidden layers, and one output layer [85]. MLPs generated some buzz with the introduction of backpropagation[12] in 1986 [88]. The backpropagation algorithm made training neural networks efficient for the first time [85]. MLPs are fully connected, meaning that every node of a layer is connected to every node in the previous and the following layer, as depicted in Figure 25.

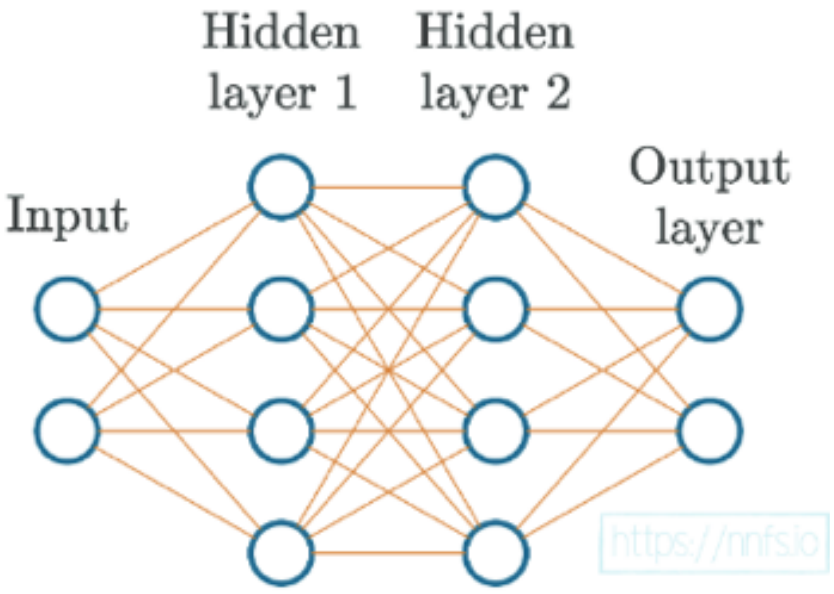


Figure 25: Example basic neural network
H. Kinsley and D. Kukieła, Neural Networks from Scratch in Python, 1st ed. (n.p.).

The widespread use of deep neural networks in recent years is rooted in two main developments. According to [89], GPUs became more powerful and accessible, and large labeled datasets became more readily available.

A GPU's usefulness in this context derives from the fact that the computationally heavy parts of deep learning are adjustments to the weights and biases. These are achieved primarily through vector and matrix operations. These are tasks at which GPUs excel [90].

[12] During forward propagation [86], the data passes through the network starting at the input layer, going through the hidden layers to finally reach the output layer. The network's output $\hat{y}$ is compared to y to determine the loss. Backpropagation allows propagating this loss value back through the network to tune the network's weights to reduce the loss [87].

Since MLPs are easier to understand and implement than other more sophisticated machine learning architectures, they constitute a good starting point. Even though a multi-layer perceptron is a very basic setup, it surprisingly manages to predict animations. I have achieved the best results with a single-layer architecture, trained using recursive triads. The presented MLP with one hidden layer achieves a validation loss value of 0.0088. As it is a rather simplistic as well as shallow network [91], training took only ~86.5 seconds. Adding a second layer degrades the results and increases validation loss to 0.009 and training time to ~105.7 seconds.

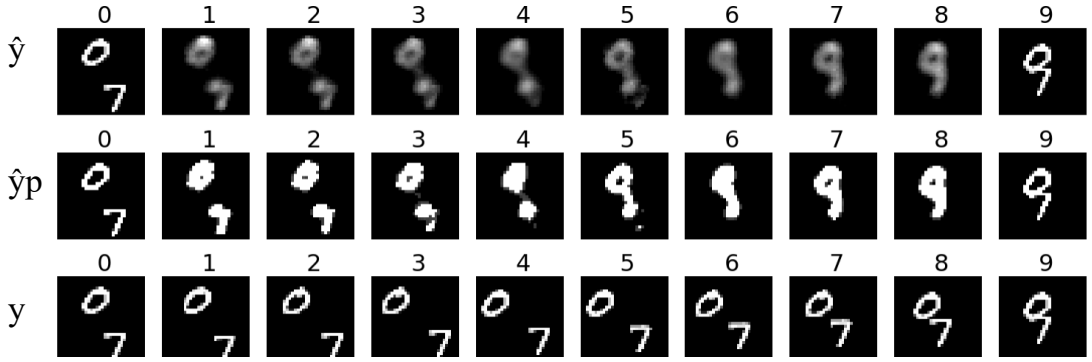


Figure 26: Predictions of a recursively trained MLP with one hidden layer; ŷ (top), ŷp (middle), y (bottom).
Own image.

Figure 26 shows the results of a one-layer deep MLP, *ŷ* is the model's prediction, and *ŷp* went through the clamping algorithm. Figure 28 displays this model's architecture consisting of one hidden layer, composed of a dense layer with the SELU activation function followed by batch normalization. The output layer is also dense but uses the ReLU activation function, and the optimizer is Adam. Figure 27 shows a close-up of the hidden layer.

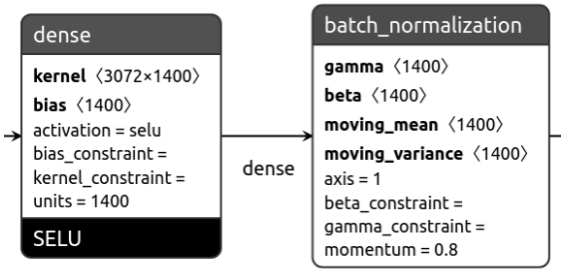


Figure 27: Hidden layer of the MLP.
Own image.

A dense layer is a fully connected collection of neurons as displayed in Figure 25. The batch normalization operation [92], which will be further investigated in chapter 5.7.5 in the context of convolutional neural networks, helps stabilize and accelerate training. The flatten operation at the very top of the MLP architecture of Figure 28 turns the 3-dimensional input triad with the shape of (3, 32, 32) into one dimension of (3072): $3 \times 32 \times 32 = 3072$. This is necessary as the MLP's dense layer can only work with one-dimensional data. The *None* in shapes displayed by Keras, i.e. (None, 3072) refers to the batch size and can be ignored.

Layer (type)	Output Shape	Param #	Trainable
flatten_11 (Flatten)	(None, 3072)	0	Y
dense_11 (Dense)	(None, 1400)	4302200	Y
batch_normalization_11 (BatchNormalization)	(None, 1400)	5600	Y
Dense_Output (Dense)	(None, 3072)	4303872	Y
reshape_11 (Reshape)	(None, 3, 32, 32, 1)	0	Y

Total params: 8,611,672
Trainable params: 8,608,872
Non-trainable params: 2,800

Figure 28: MLP architecture with one hidden layer.
Own image.

The most significant downside of an MLP for this task is that the best architecture is only a one-layer deep neural network. Larger, more complex models consistently decreased performance, likely due to neural networks becoming more challenging to train with depth [93]. This limits the prospects of finding a better MLP architecture. Unsurprisingly this architecture does not perform well when trained on a custom GIF dataset, as showcased in Figure 29. Even when applying data augmentation to provide more training data, the network's output $\hat{y}$ simply creates a fading transition between the first and last frame of y.

Figure 29: MLP trained on a custom augmented GIF dataset.
Own image.

While the multi-layer perceptron can generate somewhat matching translational results on the Moving MNIST dataset, the quality of the digits is poor. On a GIF dataset, it is the exact opposite; the animation translation is basically missing, while the icons themselves are recreated somewhat well. Since the MLP architecture was not conceived for time series prediction, it is still impressive that it could produce an image sequence at all.

5.6 Recurrent machine learning architectures

A substantial hurdle in generating animations is a lack of temporal consistency [94]. To some extent, this gets counteracted by the recursive triads approach. But the multi-layer perceptron's predictions still lack quality. Recurrent architectures on the other hand are explicitly designed to forecast time series data [95]. In the field of computer vision, most papers focusing on recurrent architectures train their models unidirectionally on multiple, concurrent input frames [40]. Some also use previous predictions by feeding them back into the model recursively [39], but still in a concurrent fashion.

The recurrent architectures, as presented in chapters 5.6.1 to 5.6.3, bring with them one obstacle that convolutional and MLP architectures in Keras omit, however. Where convolutional networks can work directly on image and video data, and MLPs only need a step inside the model that flattens the entire input array, recurrent models require additional preparation before handling animation data. They only accept 3D tensors with the following shape [batch, timesteps, features] [96], where the timesteps refer to the frames and the features to the content of a single frame. Hence, they cannot directly work on image or animation datasets. This limitation requires a minor workaround. The entire input dataset gets first converted into a three-dimensional space. A training dataset with the original array shape of (24000, 10, 32, 32, 1) will get transformed into the shape of (24000, 10, 1024), which TensorFlow can convert into a 3D tensor of the shape [24000, 10, 1024]. Table 5 explains the elements of the transformed data.

Element #	**Item**	**Example**
0	Number of animations	24000
1	Frames per animation	10
2	$Width \times height \times color\ channels$	1024

Table 5: Elements of the training data transformed into a 3D tensor.

Every frame contains three data points. The width, height, and color channels get combined into a single element (#2). This approach works very well and is lossless.
Figure 30 shows one sequence of the Moving MNIST dataset that first gets transformed into a 3D tensor as described above and is subsequentially reconstructed into its original shape without any loss or artifacts.

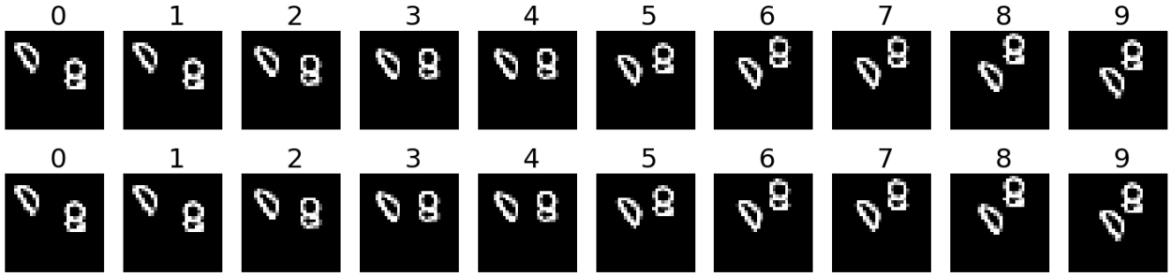


Figure 30: Original animation (top), transformed and subsequently reconstructed animation (bottom).
Own image.

5.6.1 Recurrent neural network (RNN)

The main idea behind an RNN is the memory cell which differentiates it from other architectures such as MLPs and CNNs [97].

"*Since the output of a recurrent neuron at time step t is a function of all the inputs from previous time steps, you could say it has a form of memory. A part of a neural network that preserves some state across time steps is called a memory cell*" [97].

Figure 31 on the left depicts a cell inside a neural network that uses the input *x* as well as the cell state *h* from the previous time step to create a new output. This means that each time step is influenced by the previous one. For example, frame two will be influenced by frame one and so on. On the right is shows the same cell 'unrolled' throughout three time steps.

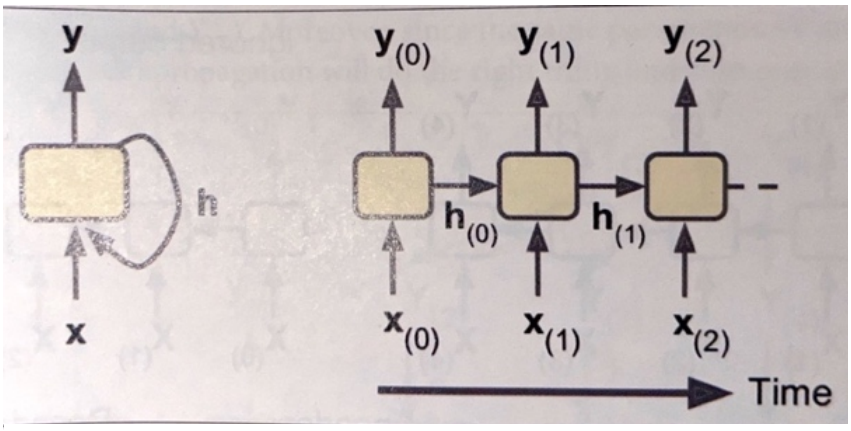


Figure 31: A cell's hidden state and its output may be different. Géron, A, Hands-On Machine learning with Scikit-Learn, Keras & Tensorflow.

Figure 32 provides a graphical representation of a deep recurrent neural network containing multiple memory cells. Every neuron is influenced by its own state from the previous time step. This allows an RNN to better keep temporal consistency since it has access to its previous state.

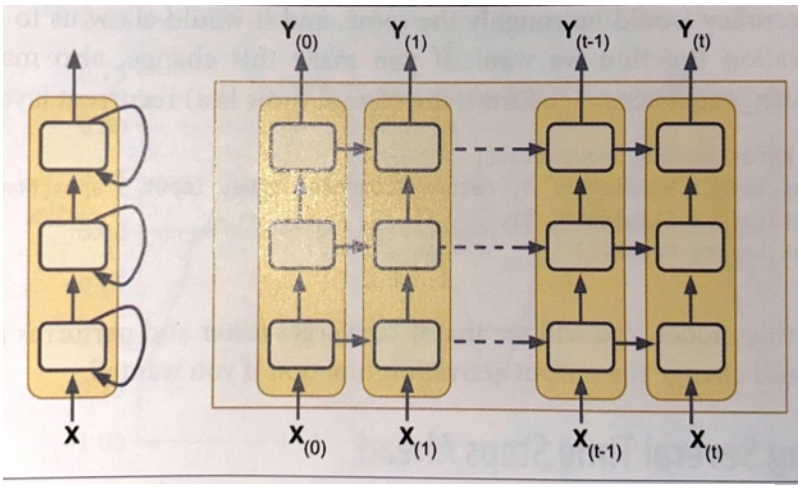


Figure 32: Deep RNN (left) unrolled through time (right).
Géron, A, Hands-On Machine learning with Scikit-Learn, Keras & Tensorflow.

Intuition suggests that neural networks with memory cells can make predictions without recursive triads. This is how they are used in many cases for time-series predictions working with sequence data [91], [98].

Layer (type)	Output Shape	Param #	Trainable
simple_rnn_3 (SimpleRNN)	(None, 10, 1024)	2098176	Y
simple_rnn_4 (SimpleRNN)	(None, 10, 1024)	2098176	Y
simple_rnn_5 (SimpleRNN)	(None, 10, 1024)	2098176	Y

Total params: 6,294,528
Trainable params: 6,294,528
Non-trainable params: 0

Figure 33: RNN architecture with three hidden layers.
Own image.

Figure 34 and Figure 35 compare the results of a conventionally trained and a recursively trained RNN, respectively. Both make use of the same model architecture as shown in Figure 33 with two hidden RNN layers and an RNN output layer, each using the tanh activation function. The model uses the Adam optimizer.

The conventionally trained network achieves a validation loss value of only 0.0277 and generates predictions that quickly turn into noise, losing any resemblance to y. The recursively trained network achieves a better validation loss value of 0.0155 and predicts digits closer to y. However, it lacks any temporal conformity with y, as the generated animation only consists of repetitions of the first input frame while slowly degrading in quality.

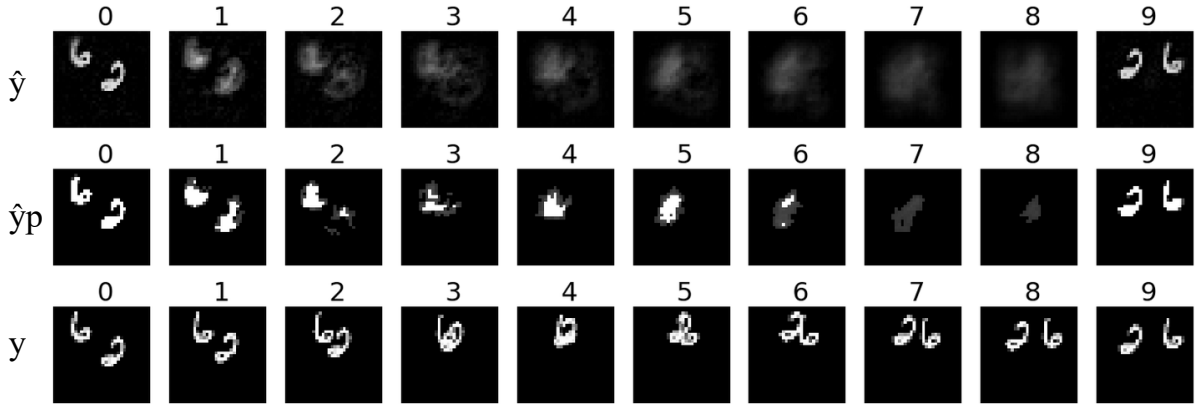


Figure 34: RNN trained conventionally.
Own image.

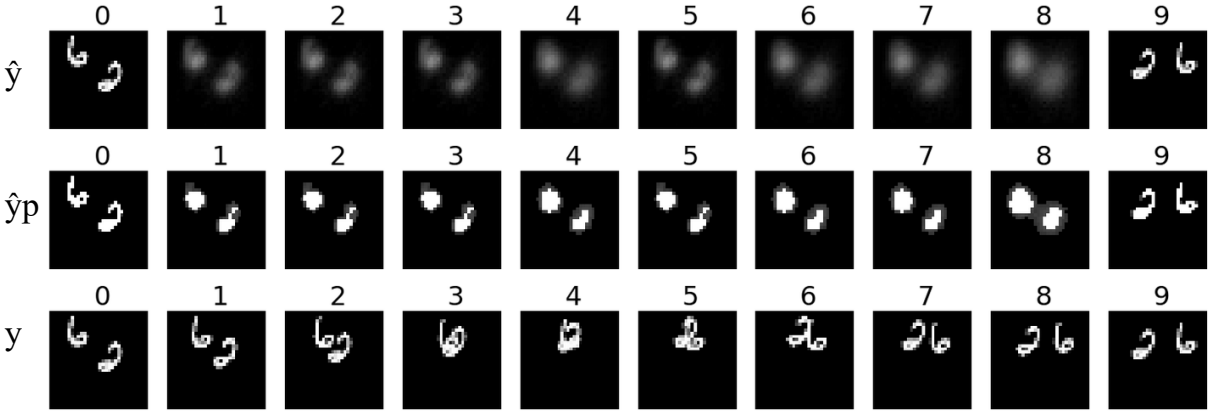


Figure 35: RNN trained recursively.
Own image.

It is no surprise that the recurrent neural network fails to predict an animation from a more complex custom GIF dataset, even if augmented, as presented in Figure 36.

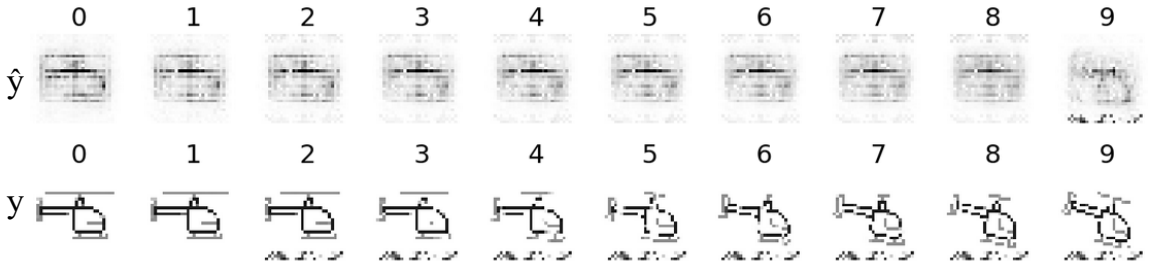


Figure 36: RNN trained conventionally on an augmented GIF dataset.
Own image.

5.6.2 Long Short-Term Memory (LSTM)

With the RNN's disappointing results, I decided to train an LSTM [99]. Long short-term memory networks are better at keeping previous time steps in memory [100]. The output of an LSTM at each time step depends on three factors: The current long-term memory, the output of the previous time step, and the input at the current time step. Figure 37 displays a simplified overview of a single LSTM cell. Denoted by $c_{(t-1)}$ is the long-term memory state of the previous time step, which at the current time step t will be affected by both the previous cell state $h_{(t-1)}$ as well as the new input data $x_{(t)}$. The cell can add new information to the memory if deemed important while dropping information that is not. The new hidden cell state $h_{(t)}$ is influenced by $h_{(t-1)}$, $x_{(t)}$ as well as $c_{(t)}$. The cell's output at the current time step is $y_{(t)}$.

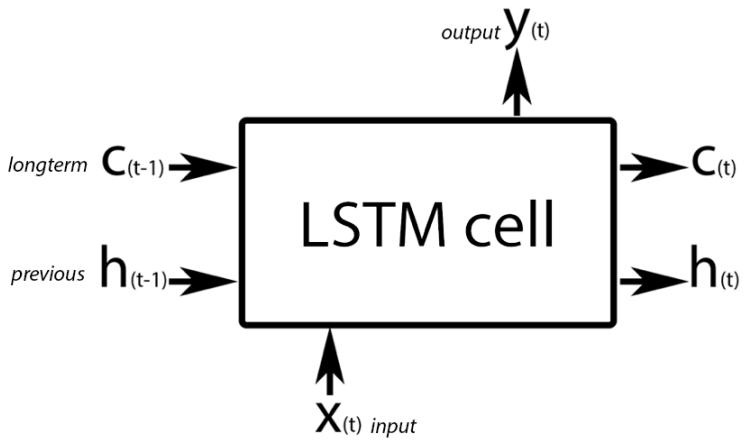


Figure 37: Simplified LSTM cell.
Own image.

The general setup was the same as for the RNN mentioned previously. The only difference is the single LSTM layer making use of the SELU activation function replacing all RNN layers, as displayed in Figure 38. The optimizer is Adam again. This model was also fed with a transformed array that gets reconstructed after making predictions. This is entirely analog to the recurrent neural network workflow.

```
_________________________________________________________________
 Layer (type)                 Output Shape              Param #
=================================================================
 lstm (LSTM)                  (None, 3, 1024)           8392704

=================================================================
Total params: 8,392,704
Trainable params: 8,392,704
Non-trainable params: 0
_________________________________________________________________
```

Figure 38: LSTM architecture with one hidden layer.
Own image.

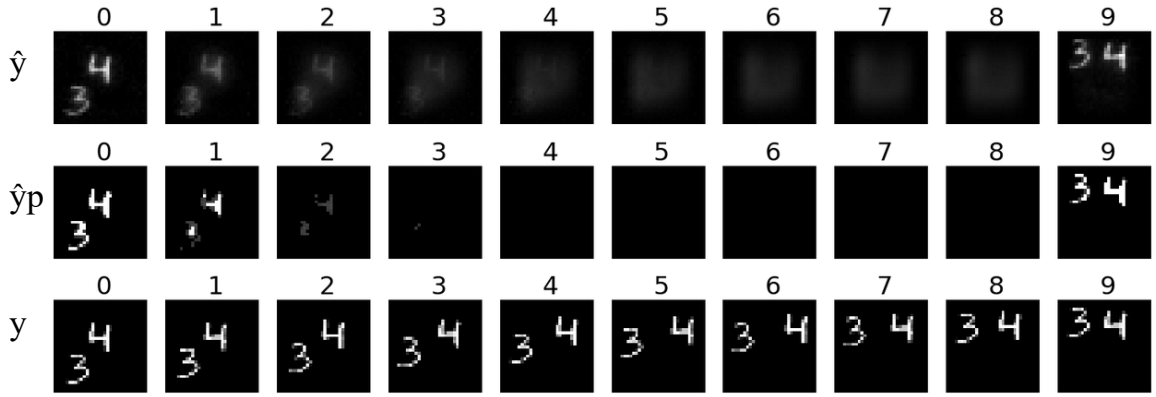


Figure 39: LSTM trained conventionally.
Own image.

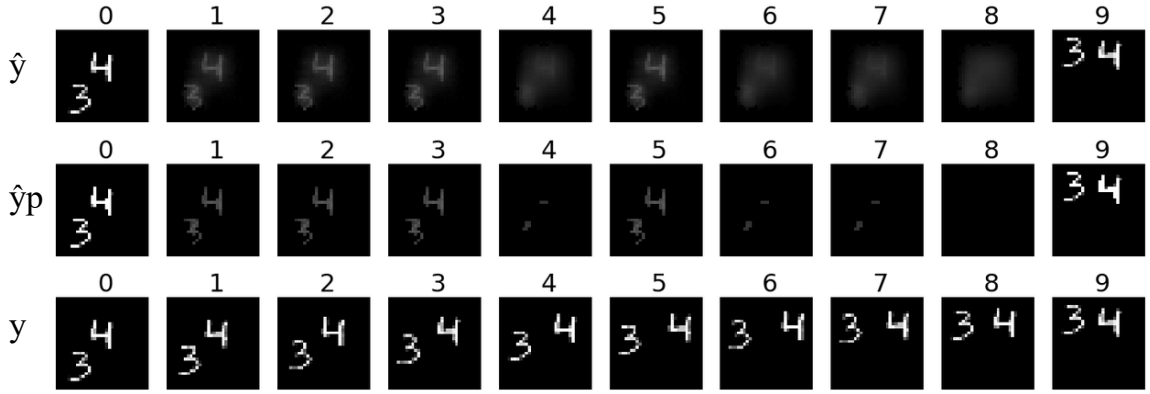


Figure 40: LSTM trained recursively.
Own image.

Both the predictions of the conventionally trained LSTM, as seen in Figure 39, and the recursively achieved results in Figure 40 fail at generating an animation between two frames. The former achieves a validation loss value of 0.0288 while the latter manages to get to 0.0125. It is surprising that the results are as bad as those of the RNN since architectures such as LSTMs that better utilize memory cells have mostly superseded RNN architectures [101].

5.6.3 Gated Recurrent Unit (GRU)

Before fully disregarding standard recurrent architectures for the presented challenge, I also tried training a GRU [102]. A *Gated Recurrent Unit* (GRU) is a simplified version of the LSTM architecture. When comparing both architectures, as shown in Figure 37 and Figure 41, it is apparent that the GRU has fewer components, which is achieved by merging both the LSTM cell's state *h* as well as the long-term memory *c* into a single entity. The two main advantages of a GRU are its simpler architecture, making it easier to understand combined with a performance boost to convergence speed.

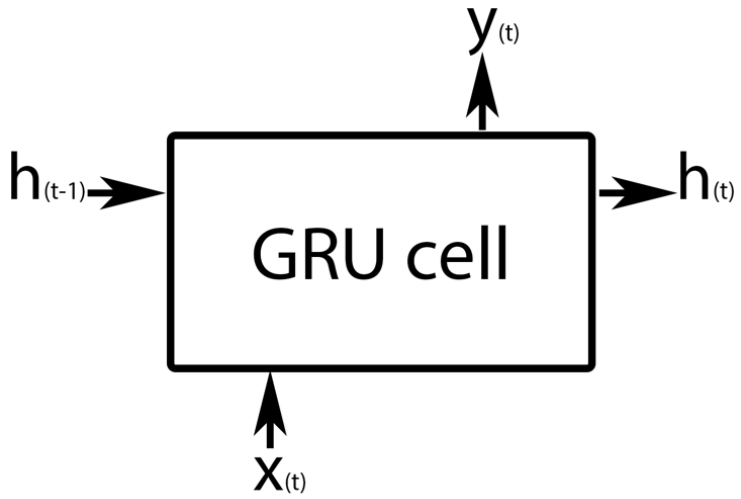


Figure 41: Simplified GRU cell.
Own image.

"*The structure of GRU is simpler. It has one gate less than LSTM, which reduces matrix multiplication, and GRU can save a lot of time without sacrificing performance.*" [103]

The GRU uses the same setup as the LSTM and RNN, including the flattening and reconstruction of the input animations. Figure 42 displays the architecture of the presented GRU, consisting of a single GRU layer with the SELU activation function. The only difference is that the GRU uses the Nadam optimizer.

```
 Layer (type)                 Output Shape              Param #
=================================================================
 gru (GRU)                    (None, 3, 1024)           6297600

=================================================================
Total params: 6,297,600
Trainable params: 6,297,600
Non-trainable params: 0
```

Figure 42: GRU architecture with one layer.
Own image.

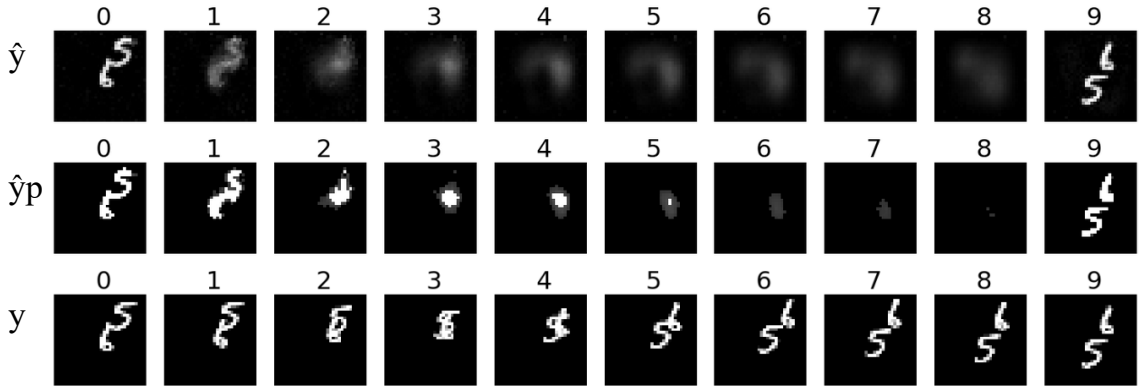


Figure 43: GRU trained conventionally.
Own image.

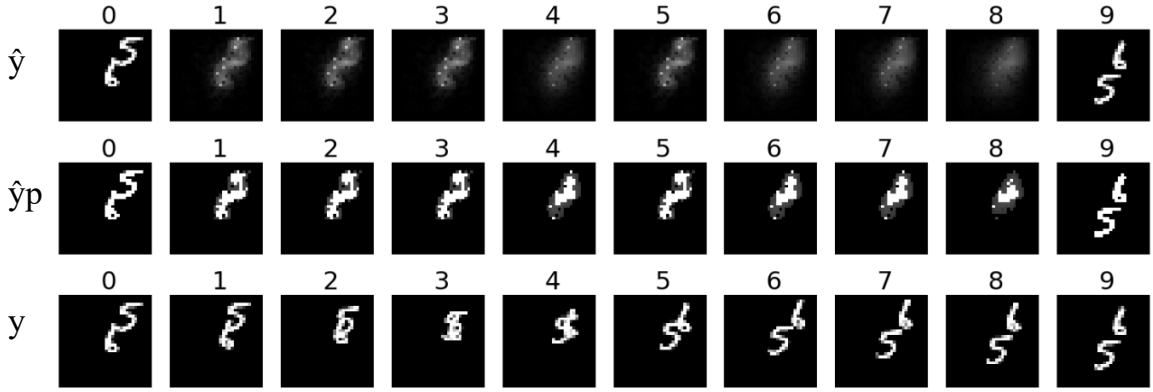


Figure 44: GRU trained recursively.
Own image.

Disappointingly, the results of neither a conventionally trained GRU as in Figure 43 nor the predictions of a recursively trained GRU as in Figure 44 display any promising attributes. The non-recursively trained GRU reaches convergence at a validation loss of 0.0283, while the recursively trained one goes down to 0.0106.

5.7 Convolutional machine learning architectures

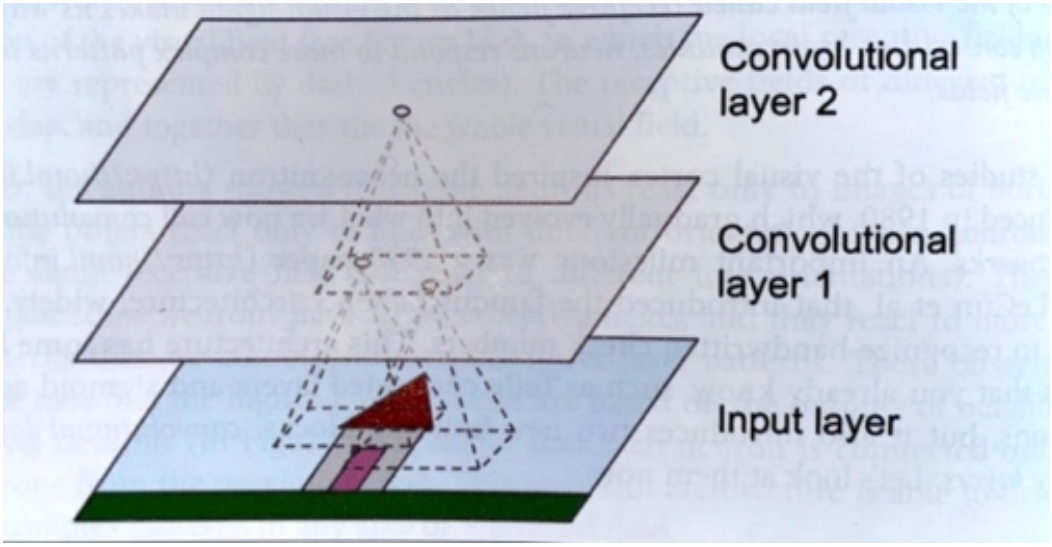


Figure 45: CNN layers with rectangular local receptive fields.
Géron, A, Hands-On Machine learning with Scikit-Learn, Keras & Tensorflow.

Deep neural networks that depend on convolutional layers[13] are expressly meant to work with image data [105] and can easily work with animations. Figure 45 shows a graphical representation of convolutional layers. The input layer (bottom of Figure 45) contains the entire picture, and the filters in subsequent layers look only at sections of that image. This concept makes it possible for a convolutional network to learn about abstract shapes and detect prominent features.

The filters[14] of a convolutional neural network can be roughly compared to the neurons in a fully dense layer, as multiple filters are stacked inside a single convolutional layer. In contrast to dense layers, convolutional layers are not fully connected. This means that the neurons in a convolutional layer are not connected to every single pixel in the input image or neuron in the preceding layer, but instead, they have a small, rectangular perceptive field called a kernel [107]. This kernel allows for filters in convolutional layers to focus on small areas of the preceding layer.

Figure 46 compares a fully connected layer, as is the case for dense layers discussed in chapter 5.5 about the multi-layer perceptron, to a convolutional layer. In the fully connected layer, every neuron in the second layer (top) is connected to every single neuron in the first layer (bottom). The neurons of the convolutional layer, in contrast, are only connected to the neurons corresponding to their respective receptive field. In this example, the receptive field, i.e., kernel size, is three neurons wide, and the number of filters is five for the layer on the bottom and three for the layer on the top. The number filters and the kernel size are the two main hyperparameters by which a convolutional layer gets adjusted.

[13] [104] offers a more in depth and visual explanation of convolutional neural networks.
[14] [106] offers a more in depth and visual explanation of filters.

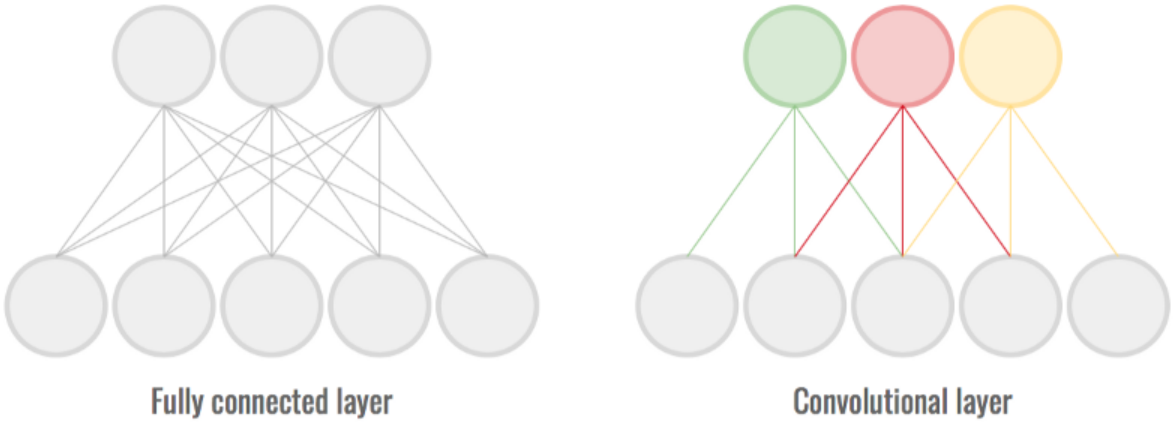


Figure 46: Fully connected layer (left); convolutional layer (right).
Bellary, Sunny Arokia Swamy. "Naive Bees - Deep Learning with Images." Sunny's Home page, August 17, 2019. https://sunnybellary.com/project/naive-bees-deep-learning-with-images/.

CNNs are not only able to recognize an image they have already seen in its entirety. This is due to their ability to use small parts of known images to recognize new ones. Figure 47 offers a more detailed view of how single filters in a later layer (top) sample data from a small region consisting of multiple filters from a preceding layer (bottom). The kernel size representing the perceptive field in this example is 3x3. Zero padding means adding zeros around the inputs (bottom) in order for the following layer (top) to keep the same height and width of the input image as in the preceding layer.

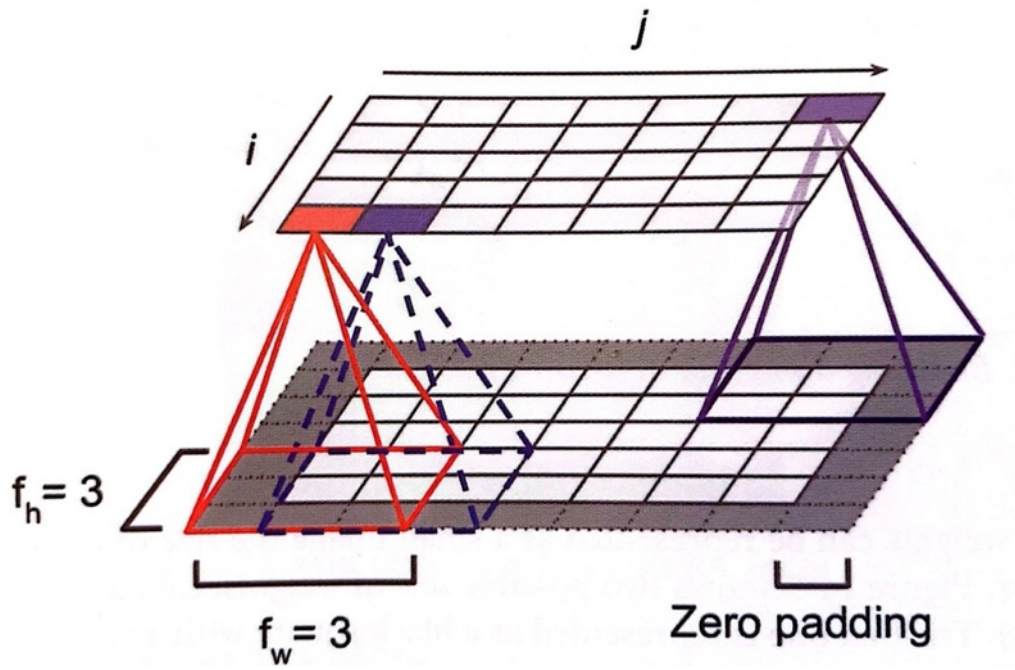


Figure 47: Connections between layers and zero padding.
Géron, A, Hands-On Machine learning with Scikit-Learn, Keras & Tensorflow.

5.7.1 Convolutional Long Short-Term Memory (ConvLSTM)

As mentioned in the related work section in chapter 3, a convolutional LSTM network, as shown in [37], can predict sequences into the future. And as shown in [38], it can even be used on GIFs. Hence, a ConvLSTM seems posed to be a good fit for predicting animations. As opposed to a regular LSTM or other recurrent networks, it incorporates the crucial convolutional layers into its architecture. It is a combination of a recurrent network as presented in chapter 5.6.2 and convolutional layers as discussed on the previous two pages.

One limitation of ConvLSTMs was that the models quickly became too large to fit into GPU memory when using more than four layers. Within this constraint, the best model was a three-layer deep convolutional LSTM as shown in Figure 48, with a kernel size of one, separated by batch normalization for each layer. The best activation function was ReLU, and Nadam was used as the optimizer.

```
Layer (type)                                 Output Shape                              Param #
=================================================================================================
conv_lstm2d (ConvLSTM2D)                     (None, 10, 32, 32, 64)                    16896

batch_normalization (BatchNormalization)     (None, 10, 32, 32, 64)                    256

dropout (Dropout)                            (None, 10, 32, 32, 64)                    0

conv_lstm2d_1 (ConvLSTM2D)                   (None, 10, 32, 32, 64)                    33024

batch_normalization_1 (BatchNormalization)   (None, 10, 32, 32, 64)                    256

dropout_1 (Dropout)                          (None, 10, 32, 32, 64)                    0

conv_lstm2d_2 (ConvLSTM2D)                   (None, 10, 32, 32, 64)                    33024

conv3d (Conv3D)                              (None, 10, 32, 32, 1)                     1729

=================================================================================================
Total params: 85,185
Trainable params: 84,929
Non-trainable params: 256
```

Figure 48: Convolutional LSTM architecture.
Own image.

Figure 49 shows a single hidden layer of that network, made up of a convolutional LSTM layer, followed by batch normalization and dropout. Dropout is a simple regularization technique that forces the neural network to randomly ignore neurons during training epochs, helping the network to avoid overfitting [108].

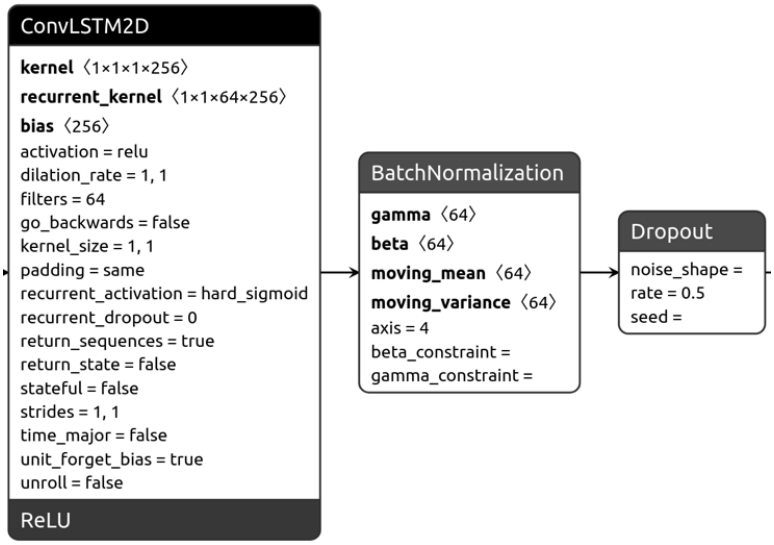


Figure 49: Close up of a single hidden layer of the ConvLSTM architecture.
Own image.

However, regardless of the architectural setup, the networks always favor simply fading the images instead of generating actual translations. Figure 50 display the results of a conventionally trained model with a validation loss of 0.1349. Recursively trained ConvLSTMs achieve a validation loss of 0.0518, but only produce a strange fade as the predictions shown in Figure 51 demonstrate. These results confirm that recurrent architectures are not well suited for predicting future frames based on non-consecutive input graphics. The convolutional LSTM suffers from very similar problems as the RNN, LSTM, and GRU models presented previously.

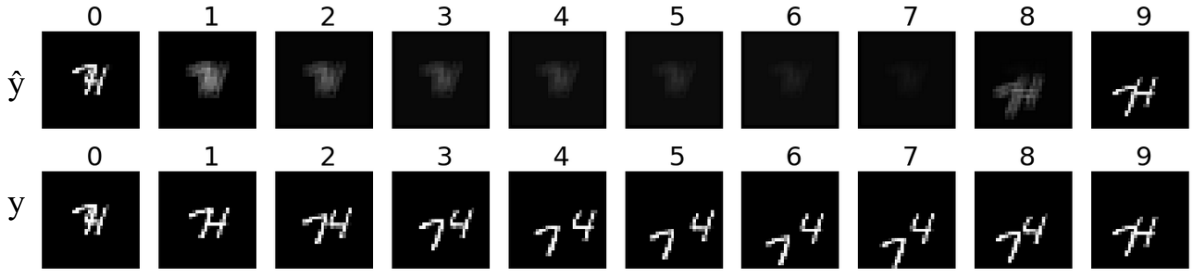


Figure 50: Results of a conventionally trained ConvLSTM.
Own image.

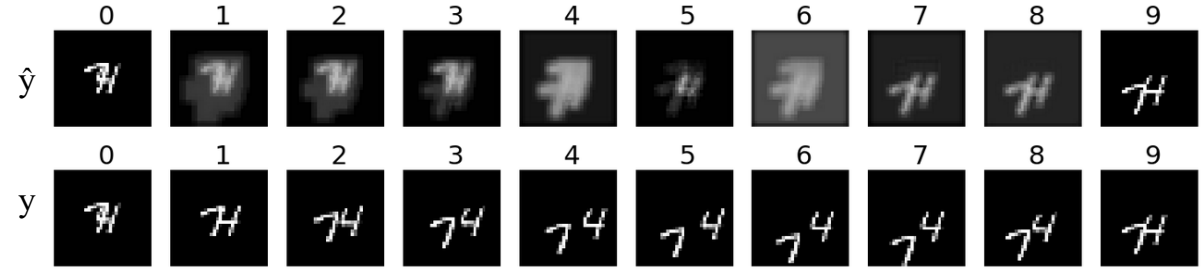


Figure 51: Results of a recursively trained ConvLSTM.
Own image.

5.7.2 ResNet-34

The other architectures presented in this thesis were explicitly created for this research and are based primarily on standard layers. Furthermore, while most of these machine learning architectures are deep neural networks by definition, they contain far fewer layers than ResNet [93]. ResNet is the winner of the 2015 ILSVRC[15] challenge using a *Residual Network*. This is a CNN with different variants ranging from 34 to 152 layers deep.

The key contribution of ResNet was the so-called *residual unit* that can be circumvented by a *skip connection,* as seen in Figure 52 on the right side. A residual unit is simply a stack of convolutional layers that the network can omit in its entirety to accelerate training.

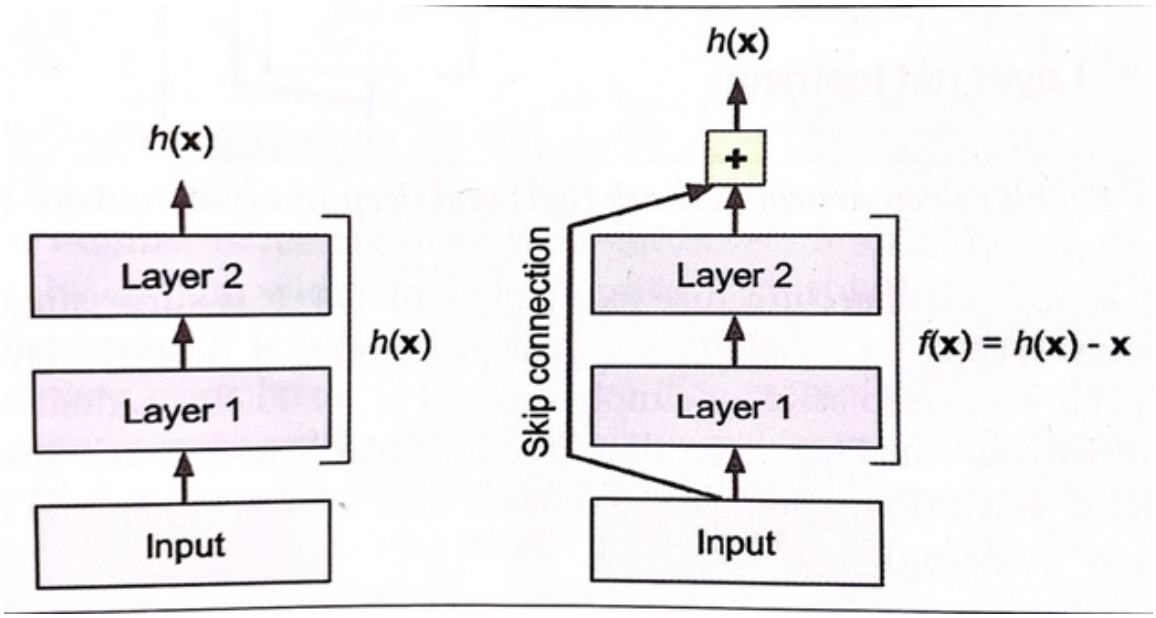


Figure 52: Residual learning.
Géron, A, Hands-On Machine learning with Scikit-Learn, Keras & Tensorflow.

These skip connections help a deep neural network progress faster, even if some layers are not yet learning. This is made possible by a stack of residual units, as shown in Figure 53. A single layer that is not learning can stop a model's entire progress in a regular neural network. However, a residual network can omit this problem by utilizing skip connections.

[15] ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [109] was a yearly challenge from 2010-2017 to evaluate algorithms for image classification and object detection [110].

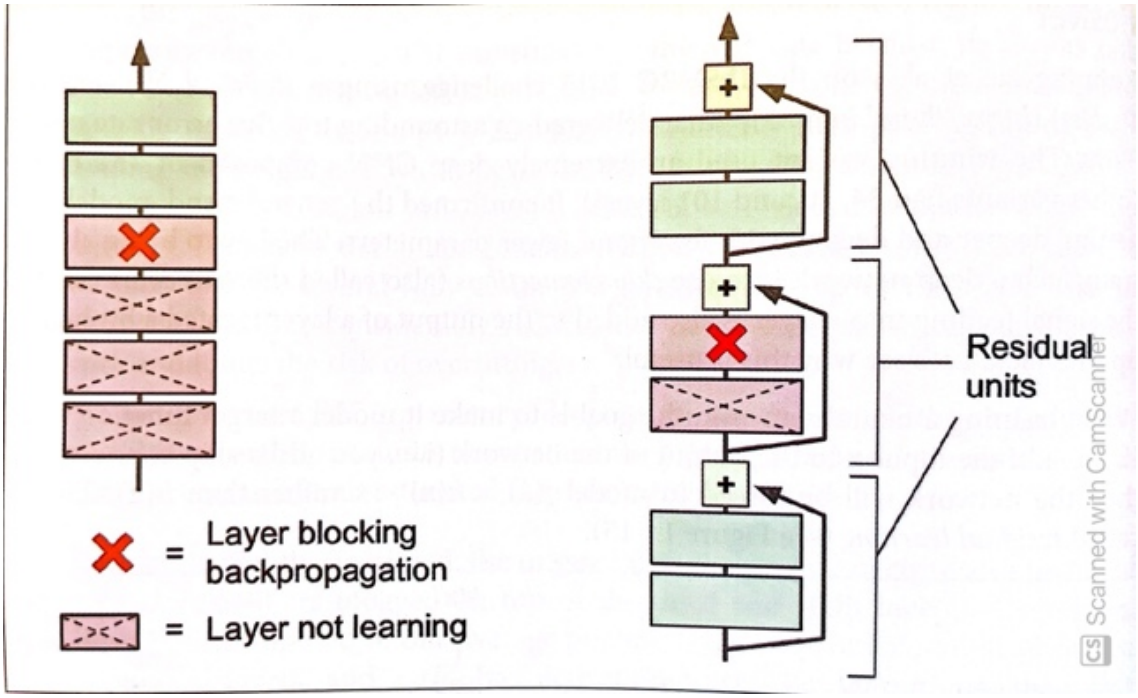


Figure 53: Regular deep neural network (left) and deep residual network (right).
Géron, A, Hands-On Machine learning with Scikit-Learn, Keras & Tensorflow.

ResNet was created for the task of image recognition. To use it on sequence data, I had to apply some key changes to its architecture. The 2D convolutional layers were replaced with 3D convolutional layers to train the model on animation data; the same goes for the 2D max-pooling and 2D global average pooling operations, which were replaced with 3D max-pooling and 3D global average pooling operations, respectively. The pooling operations are used to down-sample the image size, as will be discussed in greater detail in chapter 5.7.4. And instead of having one dense output layer consisting of ten neurons for image classification[16], this version has a dense layer with 3072 neurons representing a triad containing three 32x32 pixel frames. This is followed by a reshape layer to output an array with the original shape of the animation. The hidden layers keep the original ReLU activation function, but the output layer is modified to use tanh instead of the original softmax activation function. Finally, the optimizer is Adam.

Figure 54 displays a part of the original ResNet-34 architecture and showcases the residual units nicely. Every block of two convolutional layers is part of one residual unit, allowing the network to skip it. The skip connections are visualized as arrows in the center column of the figure. The red arrow denotes a skip connection between residual units whose convolutional layers have different filter sizes. In ResNet, the filter size of the convolutional layers doubles every time the input image is subsampled.

[16] In classification tasks the output layer has the same number of neurons as possible predictions. If a neural network is built to decide whether a given picture contains a dog or not, it would have a single output neuron. If the picture does contain a dog, the neuron activates, else it stays inactive.

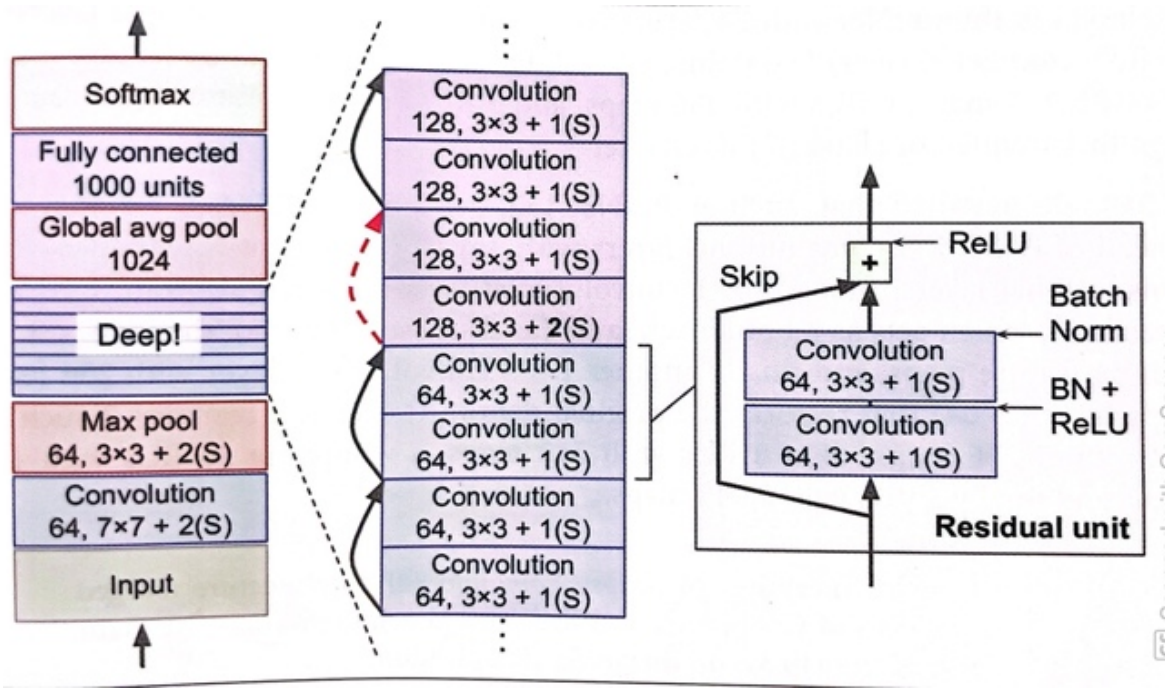


Figure 54: ResNet architecture.
Géron, A, Hands-On Machine learning with Scikit-Learn, Keras & Tensorflow.

The exact numbers of components vary between the different ResNet variants. The number of hidden layers is displayed in the respective variant's name. Resnet-34 contains 34 hidden layers, while ResNet-152 totals 152 hidden layers.

One major difference is that ResNet versions deeper than the ResNet-34 version implement a slightly different residual unit, consisting of three convolutional layers. Aside from this, all variants work the same.

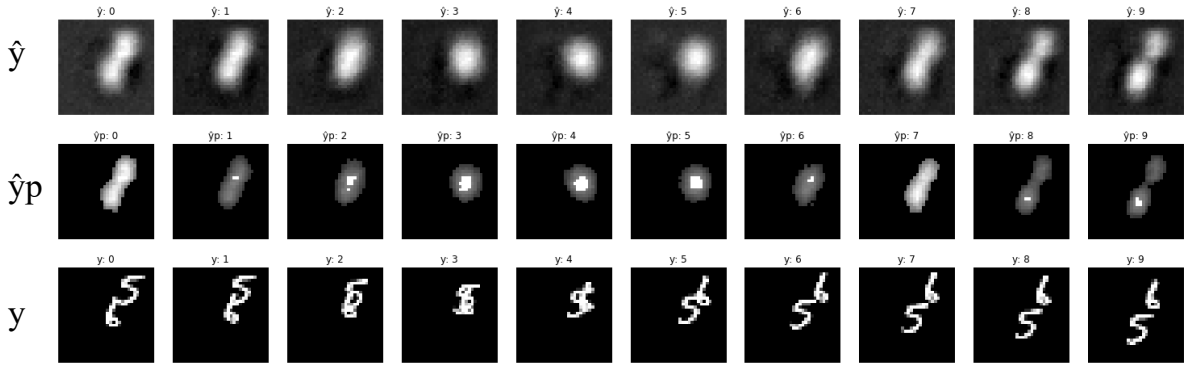


Figure 55: Conventionally trained ResNet-34.
Own image.

The conventionally trained modified ResNet-34 does not perform well, as shown in Figure 55. The results $\hat{y}$ do show some temporal motion matching y, but the digits are noise patches. This result changes drastically when applying recursive triads to the same model, as seen in Figure 56.

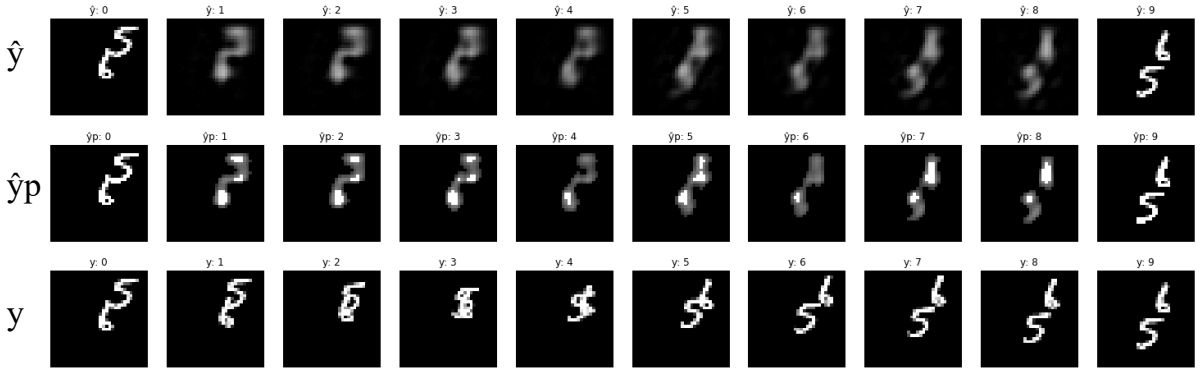


Figure 56: Recursively trained ResNet-34.
Own image.

While the predictions of a modified ResNet-34 are not convincing, they still show some promise. The results are blurry and start degrading quickly, but they represent discernible digits with a motion resembling *y*. The ResNet-34 variant achieves a validation loss of 0.0221 and took ~31.3 minutes to train.

The other ResNet architectures that contain more residual units were disappointing, however. ResNet-18, 50, 101, and 152 could not generate decent results, as displayed in Figure 57, Figure 58, Figure 59, and Figure 60. A guess at the reason for this failure would be that ResNet-18 is too shallow to work with animations, while ResNet-50 may already be too deep.

Both [111] and especially [112] mention that very deep neural networks can suffer a degradation problem. This is caused by the increased number of layers and manifests itself by first saturating and subsequentially degrading model performance. He et al. [93] verify that this problem is not caused by overfitting but rather by adding too many layers, leading to difficulties in optimization.

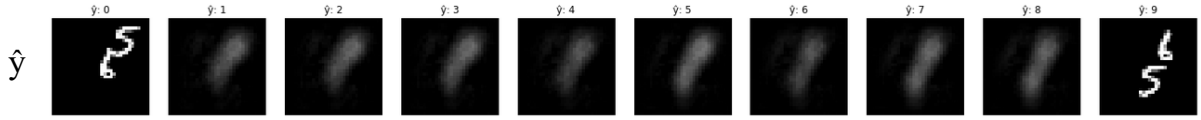


Figure 57: Recursively trained ResNet-18.
Own image.

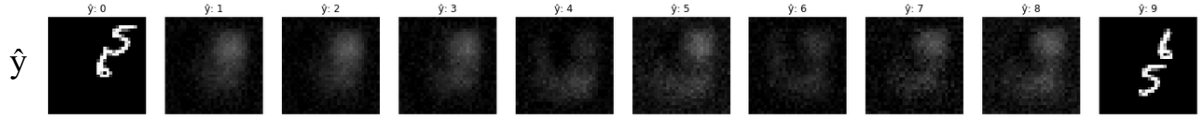


Figure 58: Recursively trained ResNet-50.
Own image.

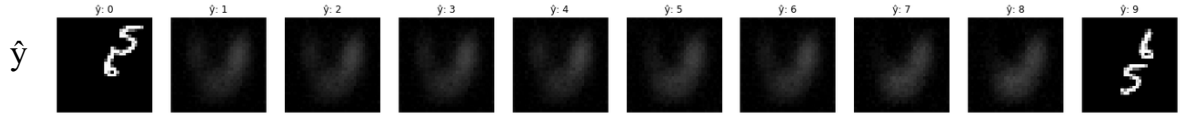


Figure 59: Recursively trained ResNet-101.
Own image.

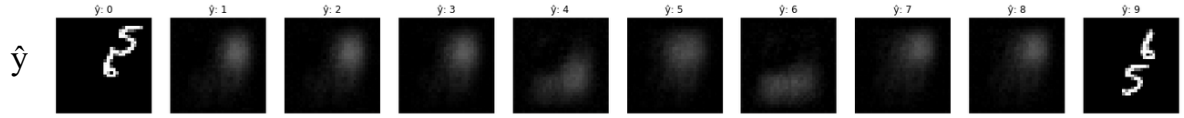


Figure 60: Recursively trained ResNet-152.
Own image.

The entire architecture of the ResNet-34 adapted for frame prediction is showcased in Figure 62 with the residual units displayed only as single entries. Every residual unit can be skipped. Figure 61 shows the internals of the first residual unit of the architecture. Other residual units may differ slightly in their number of filters and sub-sampling.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv3d_125 (Conv3D)          (None, 3, 32, 32, 64)     1728

batch_normalization_125 (BatchNormalization  (None, 3, 32, 32, 64)     256
)

re_lu_5 (ReLU)               (None, 3, 32, 32, 64)     0

conv3d_126 (Conv3D)          (None, 3, 32, 32, 64)     110592

batch_normalization_126 (BatchNormalization  (None, 3, 32, 32, 64)     256
)

=================================================================
Total params: 112,832
Trainable params: 112,576
Non-trainable params: 256
_________________________________________________________________
```

Figure 61: Architecture of the first residual unit.
Own image.

Layer (type)	Output Shape	Param #
conv3d_89 (Conv3D)	(None, 2, 16, 16, 64)	21952
batch_normalization_89 (BatchNormalization)	(None, 2, 16, 16, 64)	256
activation_2 (Activation)	(None, 2, 16, 16, 64)	0
max_pooling3d_2 (MaxPooling3D)	(None, 1, 8, 8, 64)	0
residual_unit_32 (ResidualUnit)	(None, 1, 8, 8, 64)	221696
residual_unit_33 (ResidualUnit)	(None, 1, 8, 8, 64)	221696
residual_unit_34 (ResidualUnit)	(None, 1, 8, 8, 64)	221696
residual_unit_35 (ResidualUnit)	(None, 1, 4, 4, 128)	673280
residual_unit_36 (ResidualUnit)	(None, 1, 4, 4, 128)	885760
residual_unit_37 (ResidualUnit)	(None, 1, 4, 4, 128)	885760
residual_unit_38 (ResidualUnit)	(None, 1, 4, 4, 128)	885760
residual_unit_39 (ResidualUnit)	(None, 1, 2, 2, 256)	2690048
residual_unit_40 (ResidualUnit)	(None, 1, 2, 2, 256)	3540992
residual_unit_41 (ResidualUnit)	(None, 1, 2, 2, 256)	3540992
residual_unit_42 (ResidualUnit)	(None, 1, 2, 2, 256)	3540992
residual_unit_43 (ResidualUnit)	(None, 1, 2, 2, 256)	3540992
residual_unit_44 (ResidualUnit)	(None, 1, 2, 2, 256)	3540992
residual_unit_45 (ResidualUnit)	(None, 1, 1, 1, 512)	10754048
residual_unit_46 (ResidualUnit)	(None, 1, 1, 1, 512)	14159872
residual_unit_47 (ResidualUnit)	(None, 1, 1, 1, 512)	14159872
global_average_pooling3d_2 (GlobalAveragePooling3D)	(None, 512)	0
flatten_2 (Flatten)	(None, 512)	0
Dense_Output (Dense)	(None, 3072)	1575936
reshape_2 (Reshape)	(None, 3, 32, 32, 1)	0

Total params: 65,062,592
Trainable params: 65,045,568
Non-trainable params: 17,024

Figure 62: Architecture of ResNet-34, adapted for frame prediction.
Own image.

5.7.3 Convolutional Neural Network (CNN / bbCNN)

With all recurrent architectures that have been presented failing to make any meaningful predictions and even ResNet performing poorly, it is time to look at a regular convolutional neural network. The presented model is a simplistic ten-layer deep network with 32 filters per layer and a kernel size of 3, using the Adam optimizer. It does not use subsampling and accordingly does not contain any max-pooling layers or stride. In addition, this model also forgoes using any normalization layers such as the popular batch normalization [92]. This is to make it an easy-to-understand baseline model for easier comparisons later. The topics of subsampling and normalization will be revisited in chapters 5.7.4 and 5.7.5, respectively. The goal for training this bbCNN is to reliably reach a low validation loss.

Table 6 shows the results of different numbers of 3D convolutional layers. These variants were trained on only 20% of the dataset, accelerating convergence time. The optimum of convolutional layers is ten. Using either more or fewer layers will decrease overall model performance.

Number of Conv3D layers	Validation loss	Training time
6	0.0064	~13.3
9	0.0062	~23.0
10	***0.0060***	***~27.8***
11	0.0059	~32.8
12	0.0060	~34.5
14	0.0060	~38.3

Table 6: Different numbers of Conv3D layers. The best one is highlighted in bold and italic.

Figure 63 shows a close-up of a single 3D convolutional layer as used in these tests. The results of Table 6 seem to suggest that a CNN with eleven layers performs best. Yet, further investigation shows that the eleven layers deep CNN is less stable than the ten-layer deep variant. When trained on the entire dataset the ten-layer deep model achieves a validation loss of 0.0055 in ~98 minutes. The eleven-layer deep network manages to achieve only a slightly better validation loss of 0.0054 but needs ~141.7 minutes to finish training. The deeper model displayed a clear problem to reach converge. When trained on the augmented GIF dataset, for several epochs during training the results of the eleven-layer variant keep getting higher validation errors. In contrast, the ten-layer deep network trained on GIFs moves straight towards convergence in a repeatable fashion across a multitude of tests. Considering both the time requirements as well as the stability, I consider a ten-layer deep CNN the best architecture for a baseline CNN. Figure 64 displays the final CNN architecture which will be referred to as ***bbCNN*** (***bare-bones CNN***).

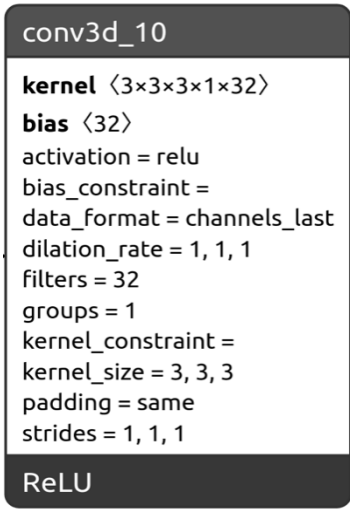


Figure 63: A single Conv3D layer.
Own image.

Layer (type)	Output Shape	Param #
conv3d_30 (Conv3D)	(None, 3, 32, 32, 32)	896
conv3d_31 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_32 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_33 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_34 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_35 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_36 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_37 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_38 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_39 (Conv3D)	(None, 3, 32, 32, 32)	27680
Output (Conv3D)	(None, 3, 32, 32, 1)	865

```
Total params: 250,881
Trainable params: 250,881
Non-trainable params: 0
```

Figure 64: Architecture of a CNN with ten convolutional layers. Own image.

As mentioned, trained on the full dataset this model achieves a validation loss of 0.0055 and needs ~1.6 hours to train. Figure 65 presents one of the animations generated by the bbCNN when trained on the Moving MNIST dataset. The model's output $\hat{y}$ is not only recognizable but stays consistent with y for the entire duration of the animation. The results that went through the clamping algorithm in a post-processing step $\hat{y}p$ are getting very close to y. If the digits do not overlap during the animation and are clearly defined, the bbCNN can create a very reasonable approximation of the original animation.

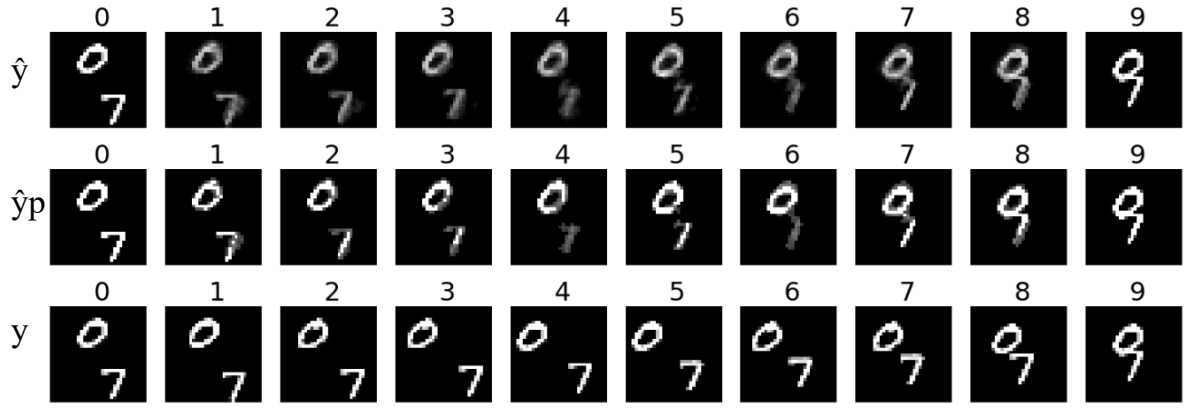


Figure 65: Predictions of a recursively trained CNN. Own image.

But the model does not perform as well as it does for the digits seen in Figure 65 for all digits of the *Moving MNIST* dataset or for custom data. Especially problematic are overlapping digits, which seem to be a blind spot for this model. Figure 67 shows the model's predictions for handwritten digits I created and scanned in. In contrast to the training dataset, the digits of the start and end frames are slightly different, and it would be expected that during the animation both digits overlap. Figure 66 shows a picture of the numbers written down initially before being processed to match the Moving MNIST training data more closely. The model's output for these custom inputs shows the digits morphing into each other. While this is not exactly the desired behavior, it does demonstrate that this architecture can automatically adapt to different input data and still achieve somewhat presentable results.

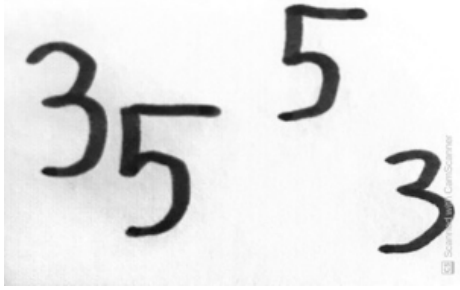


Figure 66: Custom digits.
Own image.

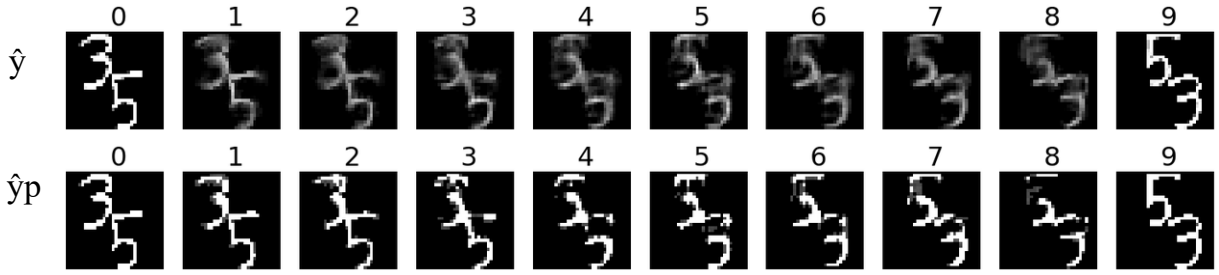


Figure 67: CNN prediction of digits written by the author.
Own image.

Using a convolutional neural network shows excellent promise. This model can be a great time-saver, especially when working with a uniform dataset, as is the case with Moving MNIST, where all samples are very similar. The predictions are less impressive when running it on a heterogeneous custom GIF dataset, as almost every animation contains different content. This forces the model to find the lowest common denominator for an animation, which in most cases is a simple, overlapping transition.

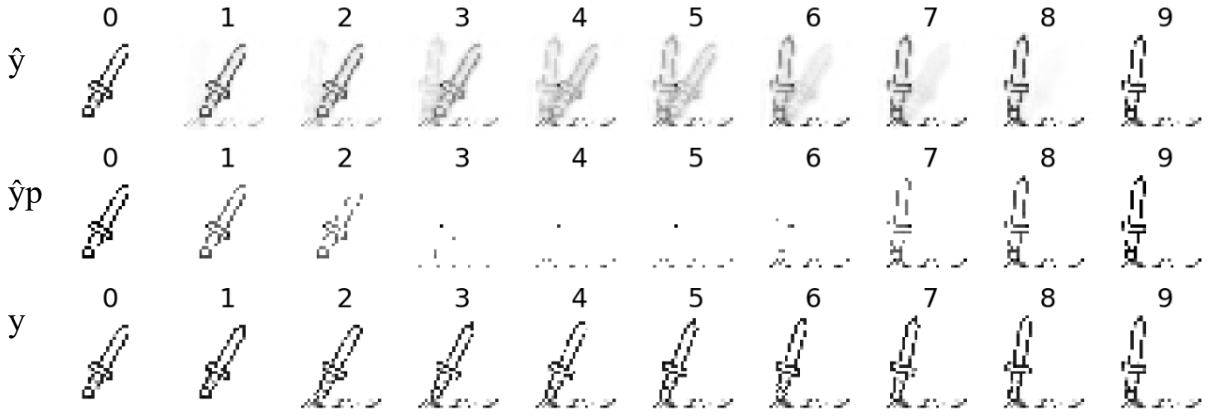


Figure 68: CNN trained on a very diverse GIF dataset.
Own image.

The bbCNN model had never seen a sword before being asked to animate one. Unsurprisingly this does not work well as noticeable in Figure 68. By comparison, the models trained on the *Moving MNIST* examples are presented with a dataset of 16.000 very similar animations during training.

But when using data augmentation on the GIF dataset, the model can do more than just fade two images. As seen in Figure 69 it can generate a matching animation. This shows that the main limitation is not necessarily the architecture itself but the dataset. Even though the results $\hat{y}$ still suffer from blurriness introduced by the mse loss function. This does however get counteracted to a large extent by post-processing leading to a cleaner $\hat{y}p$.

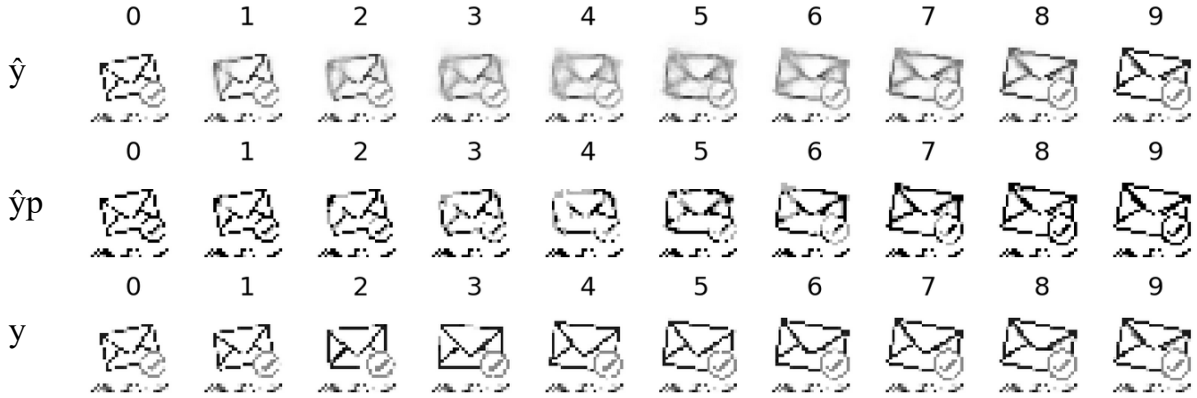


Figure 69: Predictions made by a CNN trained on an augmented GIF dataset that closely resemble y.
Own image.

Perhaps even more impressive is this model's ability to create entirely new animations. Figure 70 demonstrates this ability.

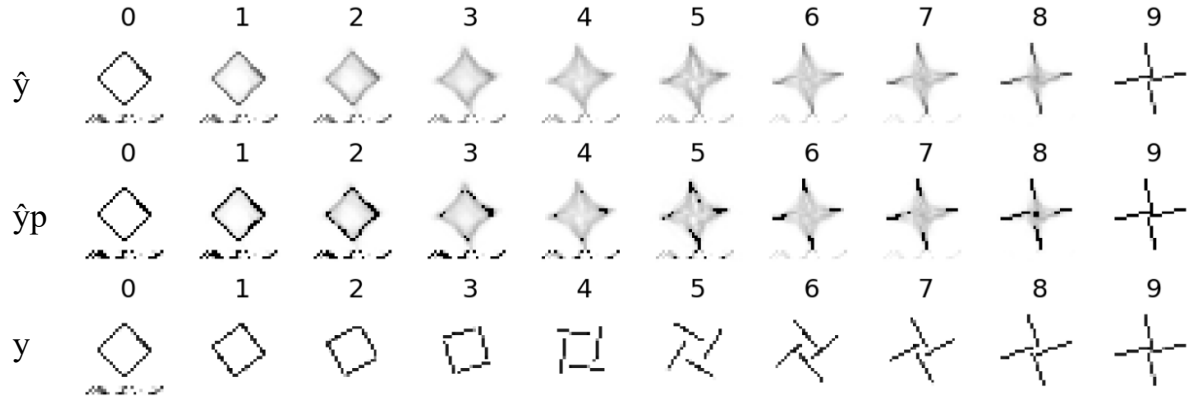


Figure 70: Predictions made by a CNN trained on an augmented GIF dataset that do not closely resemble y.
Own image.

As the model is only provided with the start and end frames, it makes up its own animation to bridge the gap between them. This result will work fine if the goal is to create a quick preview animation.

Suppose the goal is to create an animation more closely matching y. In that case, this prediction can be divided into multiple sub-steps by adjusting the recursive triads prediction algorithm to create only five frames at a time and run it twice. Once with frame 0 as the start frame and frame 4 as the end frame, and then with frame 5 as the start frame and frame 9 as the end frame. This creates two five-frame long animations that are more rigidly constrained, as demonstrated in Figure 71 and Figure 72.

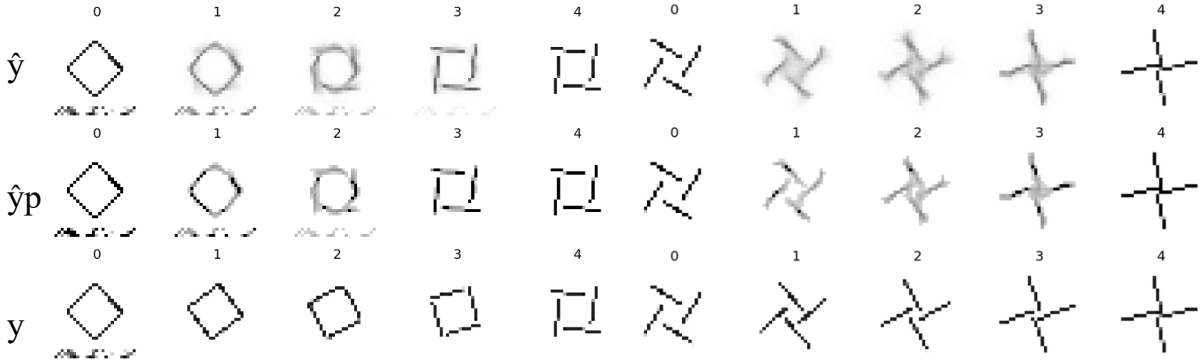


Figure 71: Predictions made by a CNN trained on an augmented GIF dataset with frames 0 and 4 as start and end frame.
Own image.

Figure 72: Predictions made by a CNN trained on an augmented GIF dataset with frames 5 and 9 as start and end frame.
Own image.

For this approach to work, the new start and end frames must be available. This is not a problem when using this setup as an augmentation to a manual animation process. This shows the flexibility of using recursive triads, as this would not be possible when only using the machine learning model itself, as it can only generate sequences of a predefined length.

5.7.4 Subsampling

Having presented a very promising architecture with bbCNN, the next step is seeing whether its shortcomings can be improved. The first downside to be addressed is the lack of subsampling, which refers to the act of shrinking the input image. This would improve the model's training performance and make it better suitable for working with larger image sizes, as it decreases the GPU memory cost. Missing out on subsampling entirely would be unfortunate, as subsampling is often used to significantly benefit a model's training run while reducing the risk of overfitting [113].

There are generally two approaches to implementing subsampling into a convolutional neural network: strides and pooling. Pooling layers are ubiquitous in modern CNN architectures, often in the form of a 2x2 max-pooling operation [114]. There are different variants of pooling operations, but the most popular one is max-pooling, as used in many famous architectures such as *AlexNet* [110], *GoogLeNet* [115], and *ResNet* [93]. When shrinking the image, the max-pooling layer selects the pixels with the highest activation, which are the brightest pixels.

"An ideal pooling method is expected to extract only useful information and discard irrelevant details." [116]

Some researchers, such as Scherer et al. [117], suggest that a max-pooling layer is superior to simple subsampling, which only shrinks the image without any specialized pixel selection. Figure 73 depicts a max-pooling layer that reduces an image's size by selecting only a single pixel, with the highest activation, from every 2x2 kernel. The square on the bottom layer outlined in red represents a 2x2 kernel in which the pixel with the highest activation, in this case, 5, is selected for the filter in the top layer that is depicted as a solid red rectangle. The bottom photograph on the right side of the figure shows the original image corresponding to the bottom layer. After that image has propagated to the top layer, its size is halved by the max-pooling layer, with the image's width being reduced by an additional pixel. This extra pixel is lost because the kernel width is even, but the image width is odd and does not utilize padding. The image's brightness has increased due to max-pooling, which only selects the pixels with the highest value and drops the rest.

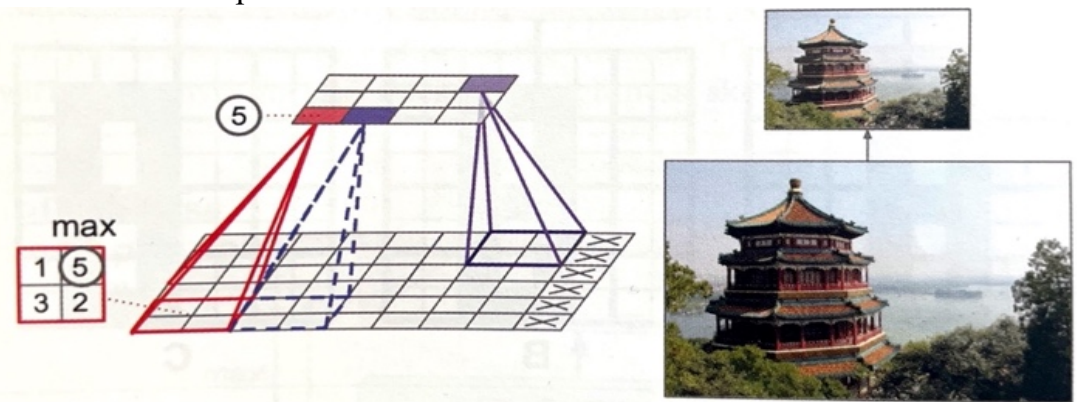


Figure 73: Max-pooling layer (2 x 2) pooling kernel, stride 2, no padding.
Géron, A, Hands-On Machine learning with Scikit-Learn, Keras & Tensorflow.

Goodfellow et al. describe pooling layers usefulness as making the input data representation invariant to small changes in translation. This means that small translational changes of the inputs do not change the output values of most pooling layers [118].

However, when working not just with images but with animations, utilizing max-pooling layers potentially degrades the model's output quality. Making a model more invariant to changes in the input data is excellent when detecting or classifying objects but does not necessarily benefit every use case [119]. Whether a digit is at the top left corner or at the center of an image should be irrelevant for a classification model that only needs to label the presented digit. But when creating animations, translation is an essential factor, and the model must not be invariant to it.
Research like *striving for simplicity* [120] even goes so far as to demonstrate that very simple architectures for convolutional neural networks can match or even outperform more complex networks by only using the strides of the convolutional layers for subsampling, instead of dedicated pooling layers. Subsampling via strides works by simply skipping pixels.

In an AMA (ask me anything) on Reddit, Geoffrey E. Hinton [121], [122] has even gone as far as to state: "*The pooling operation used in convolutional neural networks is a big mistake and the fact that it works so well is a disaster.*" [123]. A sentiment he reaffirms in his talk *What is wrong with convolutional neural nets?* [124].

With the research community in disagreement on the topic of subsampling, I had to find the optimal approach via experiments using the bbCNN architecture as the baseline. I compared this baseline to modified architectures, which contain different setups for subsampling. I also compare models that utilize max-pooling to those simply using strides. In accordance with *Striving for Simplicity: The All Convolutional Net* [120], I replaced the max-pooling layers with new 3D convolutional layers that make use of strides but also compared this to simply dropping the max-pooling layers and using strides on already present convolutional layers.

Table 7 summarizes the results. The position of pooling layers or the strides operation at different places inside the architecture can substantially impact performance. These models were trained on only 40% of the data to speed up the convergence time.

Entry #1 is the bbCNN serving as a baseline by which the other variants will be judged. Entries #2-5 display the results of using max-pooling layers, and entries #6-11 do the same, using strides inside 3D convolutional layers instead of pooling operations. Entries #6-9 simply drop the max-pooling layers and make use of strides in the convolutional layers that are already present, while entries #10-11 replace the max-pooling layers with new convolutional layers, which is the approach recommended in [120]. A full breakdown of every architecture presented in this table can be found in the appendices in chapter 10.3. These breakdowns more clearly outline the differences between architectures.

Entry #	**Description**	**Layer positions**	**Validation loss**	**Training time (m)**
1	bbCNN (baseline)	None	0.0058	~51.6
2	1 max-pooling layer	1st	0.0118	~18.2
3	1 max-pooling layer	2nd	0.0062	~18.3
4	1 max-pooling layer	5th	0.0058	~37.1
5	2 max-pooling layers	4th & 8th	0.0069	~34.5
6	2 Conv3D layers with strides	3rd & 6th	0.0063	~24.5
7	1 Conv3D layer with strides	5th	0.0060	~28.2
8	1 Conv3D layer with strides	1st	0.0058	~22.2
9	***1 Conv3D layer with strides***	***2nd***	***0.0056***	***~23.4***
10	1 additional Conv3D layer with strides	2nd	0.0057	~24.3
11	2 additional Conv3D layers with strides	4th & 8th	0.0063	~33.9

Table 7: Comparison of different subsampling approaches. The best result is highlighted in bold and italic.

Almost all models using max-pooling layers perform worse than bbCNN, albeit drastically reducing training time. The only exception is #4, which achieved the same validation loss as bbCNN while also reducing training time. Using strides instead of pooling layers, however, works better overall.

In contrast to Springenberg et al. [101] replacing the max-pooling layers with new convolutional layers (#10, #11) actually performs worse than adding strides to existing layers (#6, #9). This reconfirms the findings of chapter 5.7.3, that ten conovlutional layers are the optimum in this setup for a CNN. Entry #9 performs the best, only adding strides to the 2nd 3D convolutional layer in the model. It slightly boosts the quality of the outputs by decreasing the validation loss by 0.0002 while simultaneously reducing training time by a massive ~54.7% compared to the bbCNN model.

Figure 74 presents the predictions generated by entry #3 with one max-pooling layer. Figure 75 displays the predictions made by entry #9 using only strides. The difference demonstrates that strides are preferable in this context and shows that a difference in validation loss of 0.006 can lead to a massive difference in perceived quality.

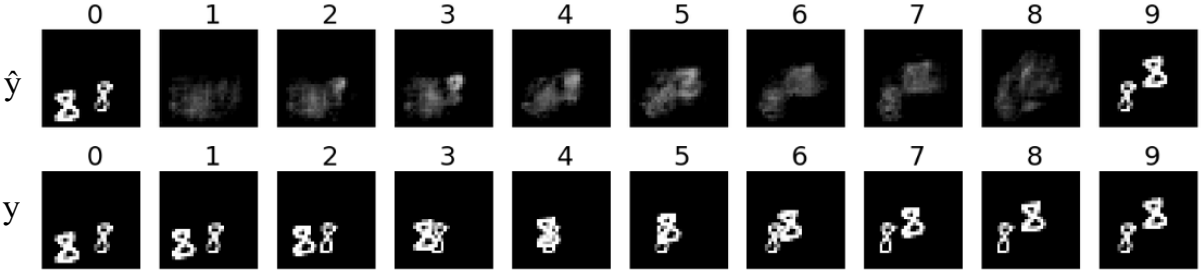


Figure 74: Predictions made by entry #3 with one max-pooling layer, achieving a validation loss of 0.0062.
Own image.

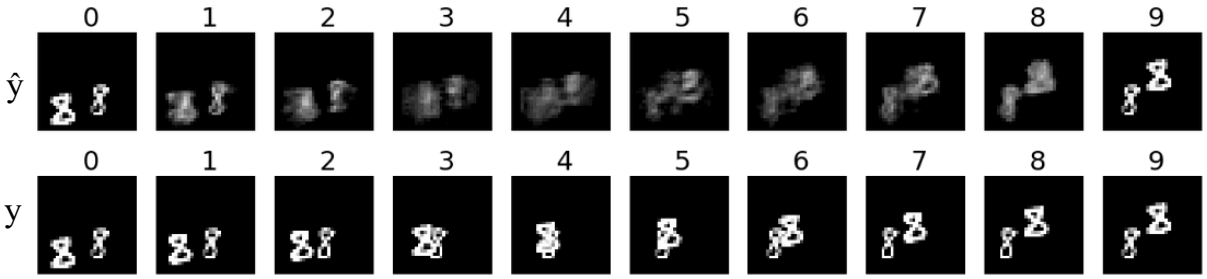


Figure 75: Predictions made by entry #3 with one Conv3D layer using strides, achieving a validation loss of 0.0058
Own image.

5.7.5 Normalization

Similar to how the input data gets normalized before being fed into the machine learning model, as described in chapter 5.1, the output of a hidden layer can be normalized before being propagated to the next layer. Not using any normalization technique has the drawback that the model does not make use of its drastically accelerated training time [92].

5.7.5.1 Batch normalization

A prominent form of normalization used in recent machine learning architectures is batch normalization (BN).

"*Batch Normalization has become very popular and is used in almost all modern CNN architectures.*" [125]

BN was proposed by Ioffe et al. [92] in 2015 and has become increasingly popular. Batch normalization offers a simple way to make deep neural networks faster and more stable [126].

$$BN(x) = \gamma \frac{x - E(x)}{\sqrt{Var(x) + \epsilon}} + \beta \tag{8}$$

Batch normalization can be expressed as in equation (8). Here *x* represents the features to be standardized, *E(x)* and *Var(x)* are the mean and variance estimates of the current batch, respectively, while both γ and β are learnable parameters. A small constant ϵ is added for numerical stability.

While many machine learning applications benefit from using batch normalization, my experiments show that for the task at hand, it clearly hurts both convergence time as well as the quality of predictions made by convolutional models. Most researchers focus on the benefits of using batch normalization [92], [126], and how it improves neural networks when compared to those without [127]. Very little research has gone into scrutinizing the possible downsides of BN. The first study about omitting BN for image recognition was released in 2021 by Brock et al. [128], closely followed by the paper *Rethinking "Batch" in BatchNorm* [129]. Possibly the first paper putting forward these deliberations for the task of video prediction was released in March of 2022 by Rivoir et al. [125]. Rivoir et al. extensively analyze the downsides of batch normalization for end-to-end video predictions. While ubiquitous in modern CNNs, the authors note that batch normalization layers are a source of performance pitfalls and even bugs. Furthermore, they present machine learning models without batch normalization that perform better than those that make use of it for video generation. These findings align well with those presented in this thesis.

A likely reason for batch normalization's poor performance in frame prediction is that images within one batch are highly correlated [125]. This per-batch correlation gets amplified when working with smaller batch sizes [130], [128], [131]. Batch normalization presumes that batches are representative of the entire training data, and according to [125] BN only performs well with large enough batch sizes.

5.7.5.2 Layer normalization

Layer normalization [132] offers an alternative to BN independent of the batches and the corresponding correlations. Layer normalization computes the mean and variance from all neurons in a layer for a single training case. This approach does not rely on the training batches and is invariant to batch size. Layer normalization was introduced specifically for use in recurrent neural networks and can substantially reduce training time. However, the researchers presenting this technique discourage the use of layer normalization in convolutional neural networks [132]. This statement, however, does not fully align with my findings. Using layer normalization in the context of CNNs did increase training time considerably, but at the same time it also improves the validation loss dramatically.

Table 8 shows a comparison of using different normalization techniques. The bbCNN model serves as a baseline again, making no use of any normalization methods. Analog to the experiments concerning subsampling, these models, too, were trained on a subset of the original data. In this case, a minimum of 40% of the dataset was required. Using fewer training examples led to some models not converging or getting stuck in local minima. I have tested batch-, group-, instance-, and layer normalization.

Normalization layers	One norm layer for each	Validation loss	Time (m)
bbCNN	None	0.0058	~51.6
Batch	1 conv layer	0.0060	~52.2
Group	1 conv layer	0.0056	~69.5
Instance	1 conv layer	0.0056	~73.3
Layer	1 conv layer	0.0055	~83.3
Layer	***2 conv layers***	***0.0054***	***~70.2***
Layer	3 conv layers	0.0057	~69.9
Layer	5 conv layers	0.0056	~60.2

Table 8: Comparison of different normalization techniques. The best version is highlighted in bold & italic

With layer normalization producing the lowest validation loss value, I tested further variants with different numbers of normalization layers. As both group-, and instance normalization fall short of layer normalization in terms of validation loss, these techniques will not be further investigated. Every model in Table 8 has exactly ten convolutional layers. After every one, two, three, or five convolutional layers, a normalization operation is added.

Surprisingly, using layer normalization after every second convolutional layer provided the best quality, even though the original paper presenting them [132] discouraged their use in convolutional architectures. However, improved prediction quality with layer normalization comes at the cost of increased training time by ~36.1% compared to the baseline model.

Figure 76 shows one predicted animation made by the bbCNN, while Figure 77 shows the results of an augmented version of the bbCNN that uses layer normalization after every second convolutional layer.

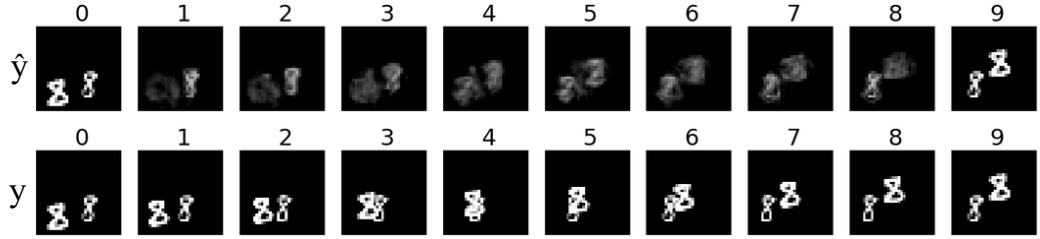


Figure 76: bbCNN with no normalization.
Own image.

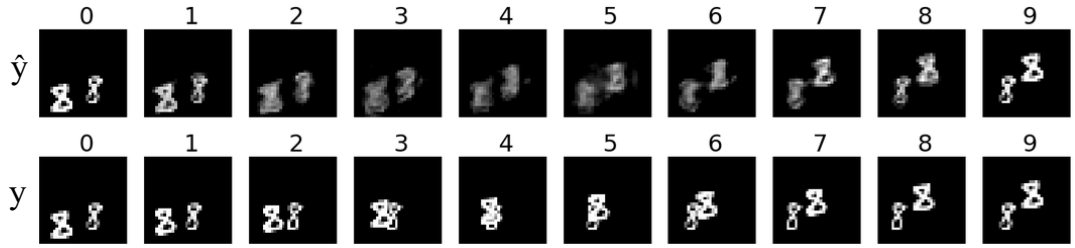


Figure 77: bbCNN enhanced by one layer normalization operation for every two convolutional layers.
Own image.

5.7.6 Skip connections

The experiments for bbCNN have shown that ten 3D convolutional layers are the optimal number for this task, as deeper neural networks performed worse. But as discussed in chapter 5.7.2 about ResNets, using skip connections can enable the use of more layers than otherwise possible.

ResNet itself performs poorly on the task of generating animations between two graphics, and the entire architecture with residual units is quite complex. I have achieved improved results, however, by reusing the logic of skip connections to enable the training of a deeper convolutional neural network based on bbCNN. To this extent, I adapt the bbCNN architecture and wrap eight of its layers into a residual unit which gets repeated twice. Both these units are connected via skip connections. This architecture has eighteen 3D convolutional layers in total. One convolutional layer at the start, one at the end, and eight convolutional layers in each of the two residual units. This setup achieves a validation loss of 0.0047 with ~2 hours of training time.

5.7.7 Automated keyframe animation network (akaNet)

After delving deeper into subsampling, normalization as well as skip connections I present a machine learning architecture that is an aggregation of these learnings, which I call *akaNet*. It is a continuation of the very simple bbCNN presented in 5.7.3, but it improves upon its downsides. The akaNet model is still simpler than ResNet and no more complex than a convolutional variational autoencoder as presented in chapter 5.7.8, while achieving higher quality predictions than both.

The core of akaNet is based on 3D convolutional layers but in contrast to bbCNN, it adds layer normalization after every second or third convolutional layer, makes use of subsampling, and wraps this entire setup into a residual unit, which gets repeated twice. Increasing the number of residual units in the akaNet architecture achieves steadily improved loss values. However, this leads the model to start overfitting the training data. This becomes evident by a divergence between training and validation loss. When overfit, it can make very high-quality predictions in some cases, but in many other cases, it produces extremely blurred digits. While skip connections improve the training of deeper networks, He et al. [93] argue that aggressively deep neural networks start suffering from overfitting on smaller datasets. Something like a 50-layer deep neural network may be too excessive for datasets such as Moving MNIST.

Every residual unit includes a subsample operation within the second convolutional layer and an up-sample operation within the final convolutional layer of the residual unit. This setup has the positive side effect that the residual units are entirely self-contained, which makes changing the amount of those units very easy. If the subsampling and upsampling operations were not self-contained, changing the network architecture would be more complex. Figure 78 and Figure 79 show akaNet's entire architecture.

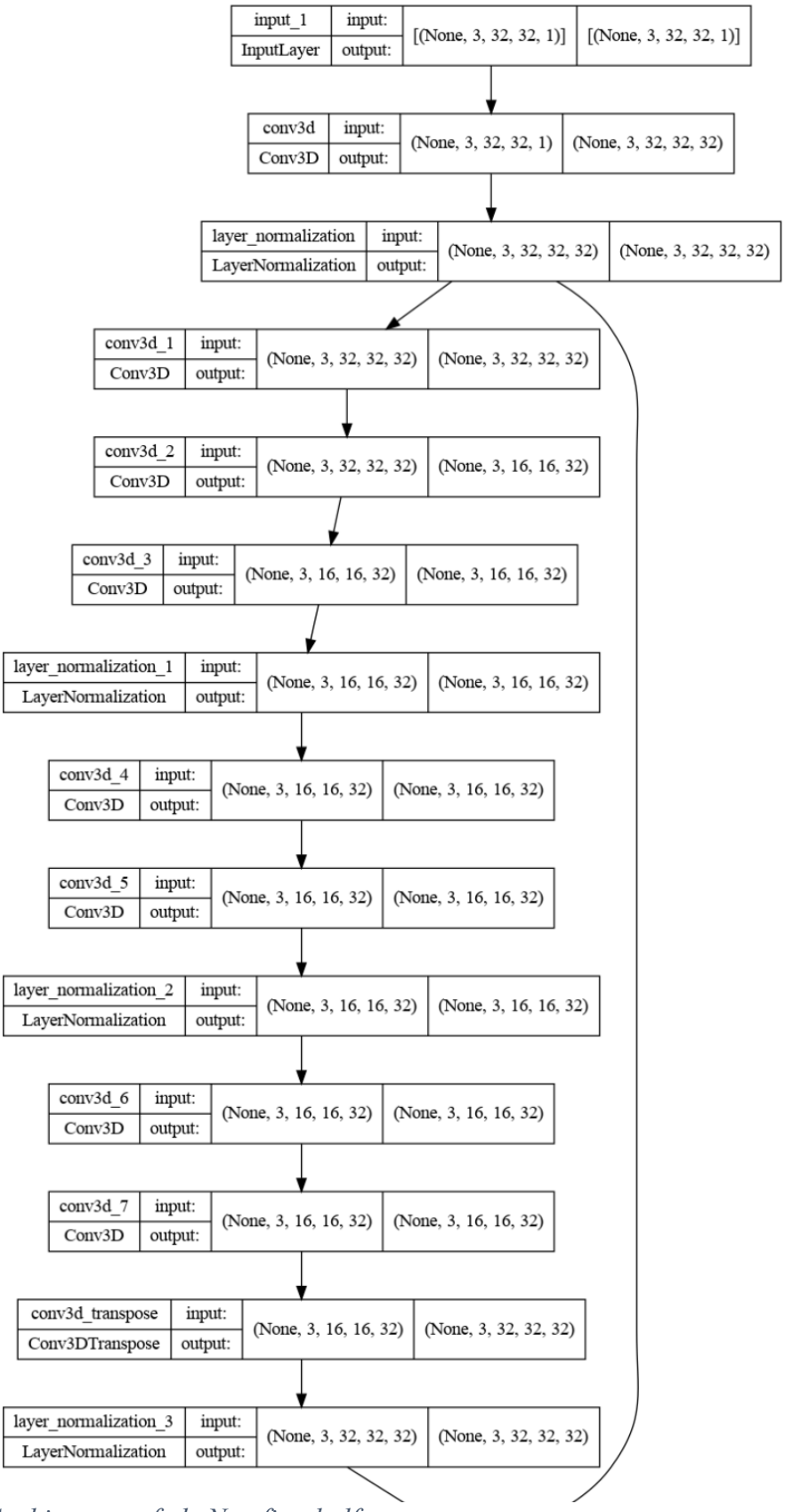


Figure 78: Architecture of akaNet, first half.
Own image.

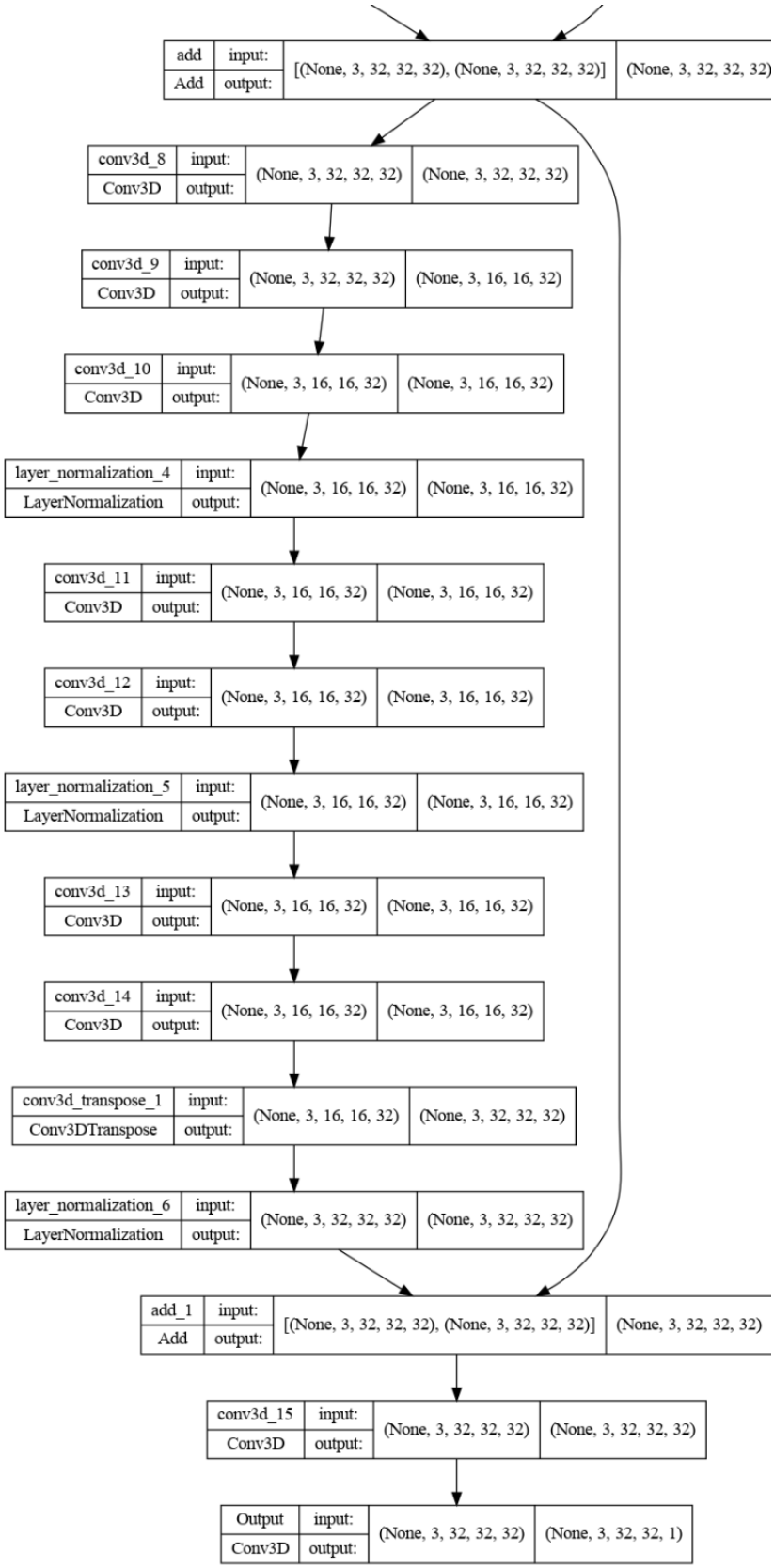


Figure 79: Architecture of akaNet, second half.
Own image.

The final akaNet model achieves a validation loss of 0.0045 and needs ~111.1 minutes for training. Figure 80 displays one predicted sequence by akaNet. It still suffers from the same problems of digits morphing into each other when dealing with overlapping shapes as was discovered with the bbCNN model in chapter 5.7.3, but the quality of results is higher. And while the digits still morph into each other, they very clearly follow a matching trajectory and suffer less from blurriness. The upward trajectory of the digit six until frame three in *y* is, of course, ignored, as the two input graphics (frame 0 and frame 9) do not contain any information that would lead the model to follow that exact trajectory. With the data given, this machine learning model manages to generate a believable animation.

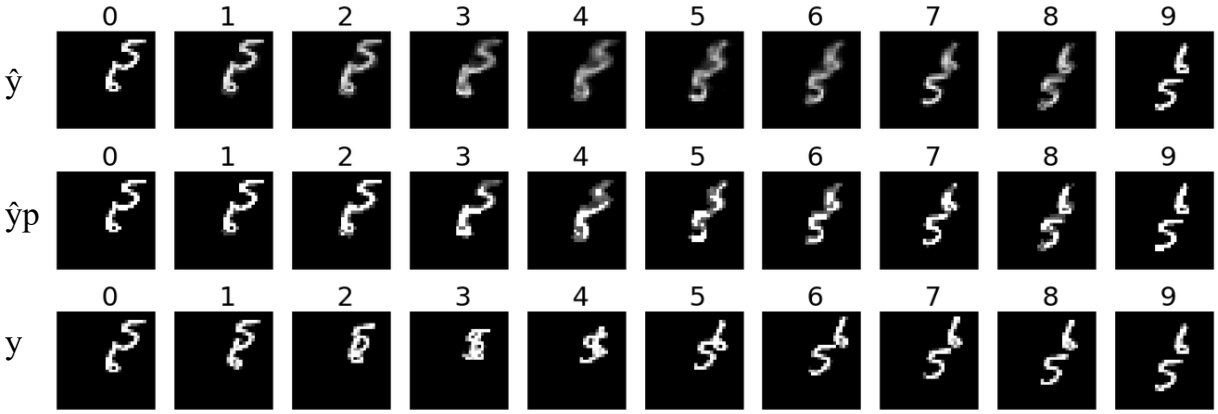


Figure 80: Prediction made recursively by akaNet.
Own image.

When trained on an augmented GIF dataset, the predictions of akaNet moderately outperform those of the bbCNN. This gets demonstrated by the prediction of akaNet shown in Figure 81, which is slightly less blurry than a prediction of bbCNN as seen in Figure 69 for the same input graphics.

Figure 81: Prediction of akaNet when trained on an augmented GIF dataset.
Own image.

5.7.8 Convolutional variational autoencoder (ConvVAE)

An autoencoder [133] is a type of neural network composed of two distinct sub-networks. The first network is called an encoder that encodes the input data as $h = f(x)$, the resulting h is called *codings*. The second network is a decoder that generates a reconstruction $r = g(h)$ out of these codings, with $g(f(x)) \neq x$, as the network is not supposed to simply copy the data [134]. To prevent the network from copying the input data it contains a bottleneck in the middle. Autoencoders containing multiple hidden layers are also called stacked autoencoders [135]. Figure 82 shows the hidden layers of such a stacked autoencoder. The red hidden layer 2 in the middle with 30 units functions as the bottleneck. The input data of 784 units must squeeze through this bottleneck by discarding most of its information. The decoder only works with the codings containing 30 units to reconstruct the total data totaling 784 units. This enables autoencoders to learn simpler representations for high-dimensional data such as photos.

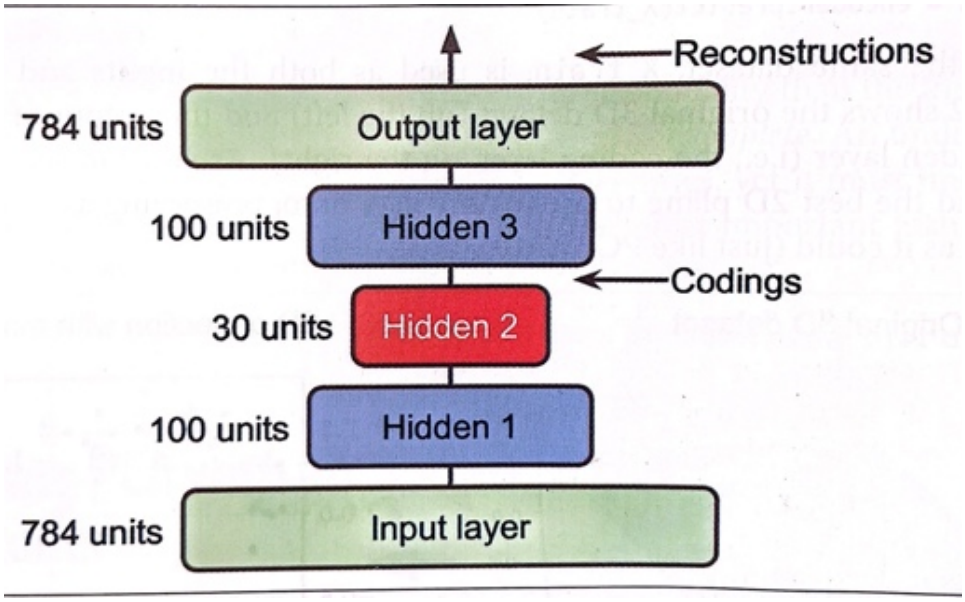


Figure 82: Stacked autoencoder.
Géron, A, Hands-On Machine learning with Scikit-Learn, Keras & Tensorflow.

Variational autoencoders [136] were introduced in 2013 and quickly became popular due to their two main properties, which differentiate them from other types of autoencoders.

"*They are probabilistic autoencoders, meaning that their outputs are partly determined by chance* […]. *Most importantly, they are generative autoencoders, meaning they can generate new instances that look like they were sampled from the training set.*" [137]

When comparing different autoencoder architectures, only the convolutional variational autoencoder could create recognizable results. Other architectures such as stacked autoencoders, convolutional autoencoders, and variational autoencoders had severe problems with underfitting.

Table 9 compares the validation losses of different autoencoders trained recursively on the full Moving MNIST dataset. Every architecture presented in this table also uses batch normalization but does not contain subsampling or residual units.

Autoencoder type	**Validation loss**	**Training time (m)**
Stacked autoencoder	0.0496	~5.3
Convolutional autoencoder	0.0186	~671.1
Variational autoencoder	0.1856	~3.1
Convolutional variational autoencoder	***0.0084***	***~284.6***

Table 9: Different autoencoder architectures. The best one is highlighted in bold and italic.

Only the combination of convolutional layers and a variational autoencoder showed promise when used in conjunction with recursive triads. The difference between a regular autoencoder and a variational autoencoder is that the codings of the encoder in the variational version don't get directly fed into the decoder, as would be the case in a stacked autoencoder, as shown in Figure 82. In a variational autoencoder, the codings first go through an intermediary step, adding some randomness, before being sent to the decoder.

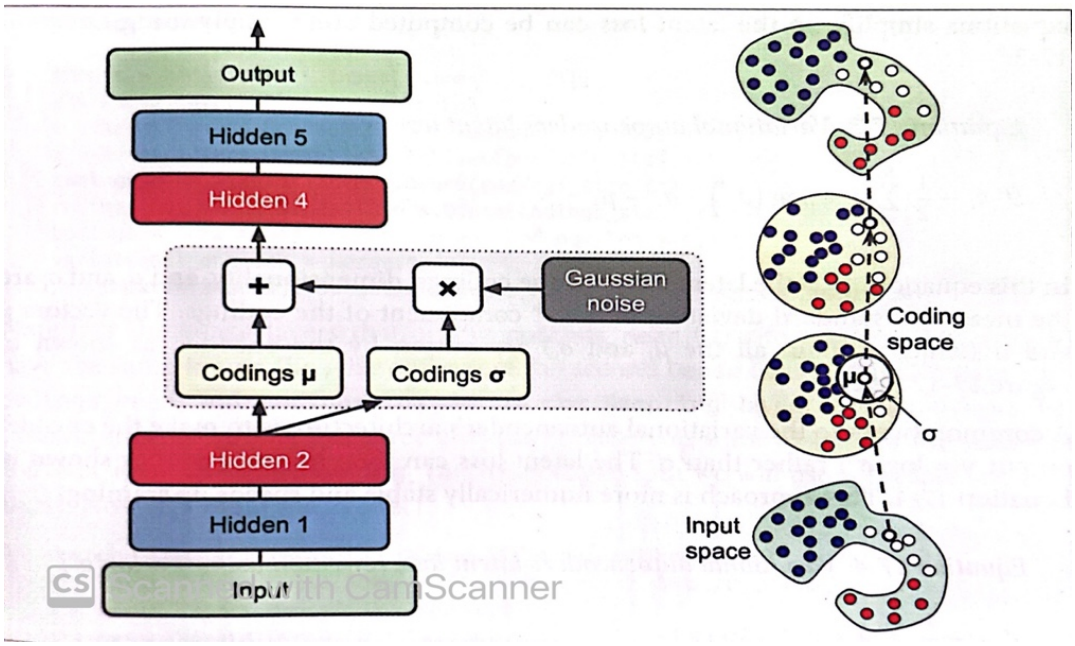


Figure 83: Variational autoencoder (left) and an instance going through it (right).
Géron, A, Hands-On Machine learning with Scikit-Learn, Keras & Tensorflow.

Figure 83 shows the setup of a variational autoencoder, containing the added logic between encoder and decoder. This extra logic creates a *mean coding* $\boldsymbol{\mu}$ and a *standard deviation* $\boldsymbol{\sigma}$. This allows for the codings to be sampled from a gaussian distribution. A benefit of this approach is that new instances can simply be sampled from that distribution. A variational autoencoder uses a cost function to create codings that look like they are part of the input data.

This cost function has two responsibilities. Firstly, it provides a *reconstruction loss*, showing the difference between the generated output and the original data. Secondly, it provides the *latent loss* as described in equation (9) that makes the codings look like from a gaussian distribution.

$$\mathcal{L} = -\frac{1}{2}\sum_{i=1}^{K} 1 + log(\sigma_i^2) - \sigma_i^2 - \mu_i^2 \tag{9}$$

The internals of both the encoder and decoder are inspired by the bbCNN from chapter 5.7.3 as well as the findings concerning normalization from chapter 5.7.5. The ConvVAE shown in Table 9 that utilizes batch normalization reaches a validation loss of 0.0084, but when using layer normalization, this value is drastically improved, achieving a validation loss of 0.0047. Unfortunately, this architecture did not work well with subsampling. A prediction of a conventionally trained ConvVAE can be seen in Figure 84, which severely underfits the training data.

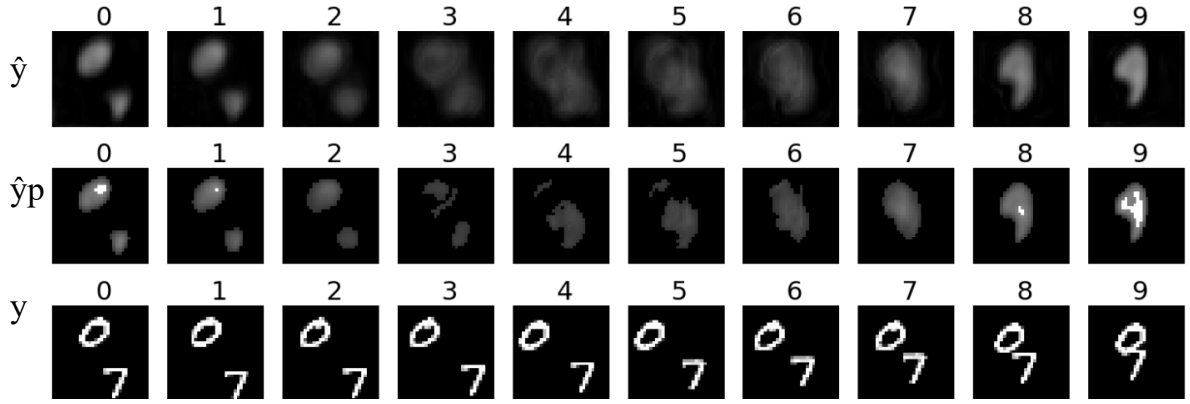


Figure 84: Conventionally trained ConvVAE.
Own image.

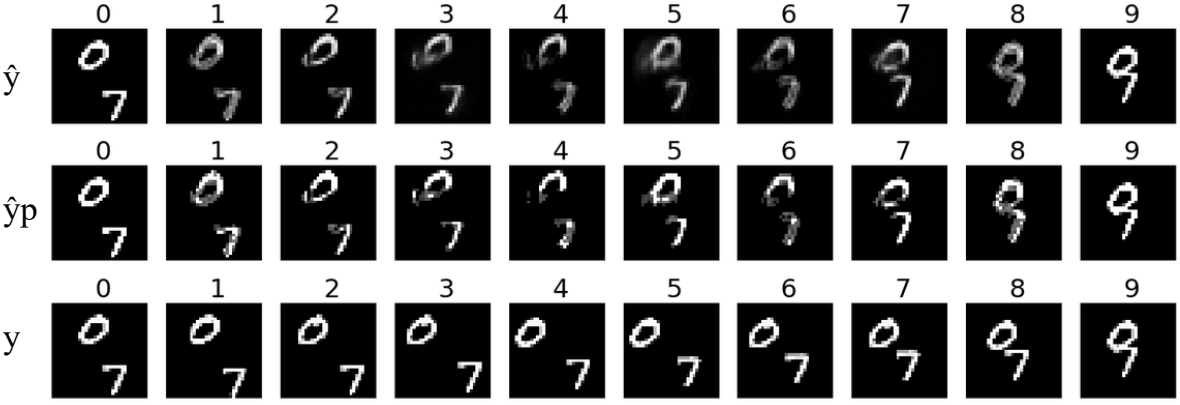


Figure 85: Recursively trained ConvVAE.
Own image.

The results of a recursively trained ConvVAE, as presented in Figure 85, align much better with *y*. In terms of quality, the ConvVAE performs slightly better than bbCNN and slightly worse than akaNet.

Figure 86 shows the results of the convVAE when trained on the augmented GIF dataset. These results are faintly less blurry than those of bbCNN shown in Figure 69 and marginally blurrier than the predictions of akaNet presented in Figure 81.

Figure 86: Predictions made by a ConvVAE trained on an augmented GIF dataset. Own image.

The final ConvVAE achieves a validation loss of 0.0047, which is just slightly lower than the 0.0045 of akaNet. However, the training time for this ConvVAE was significantly higher with ~14.7 hours, making it non-competitive. Especially when considering the considerably deeper and somewhat more complex architecture of the ConvVAE. The akaNet architecture is easier to understand, maintain and adjust. The two main components of this ConvVAE are the eight-layer deep encoder, as shown in Figure 87 followed by a sixteen-layer deep decoder, as seen in Figure 88. The two main components are 3D convolutional layers in combination with layer normalization.

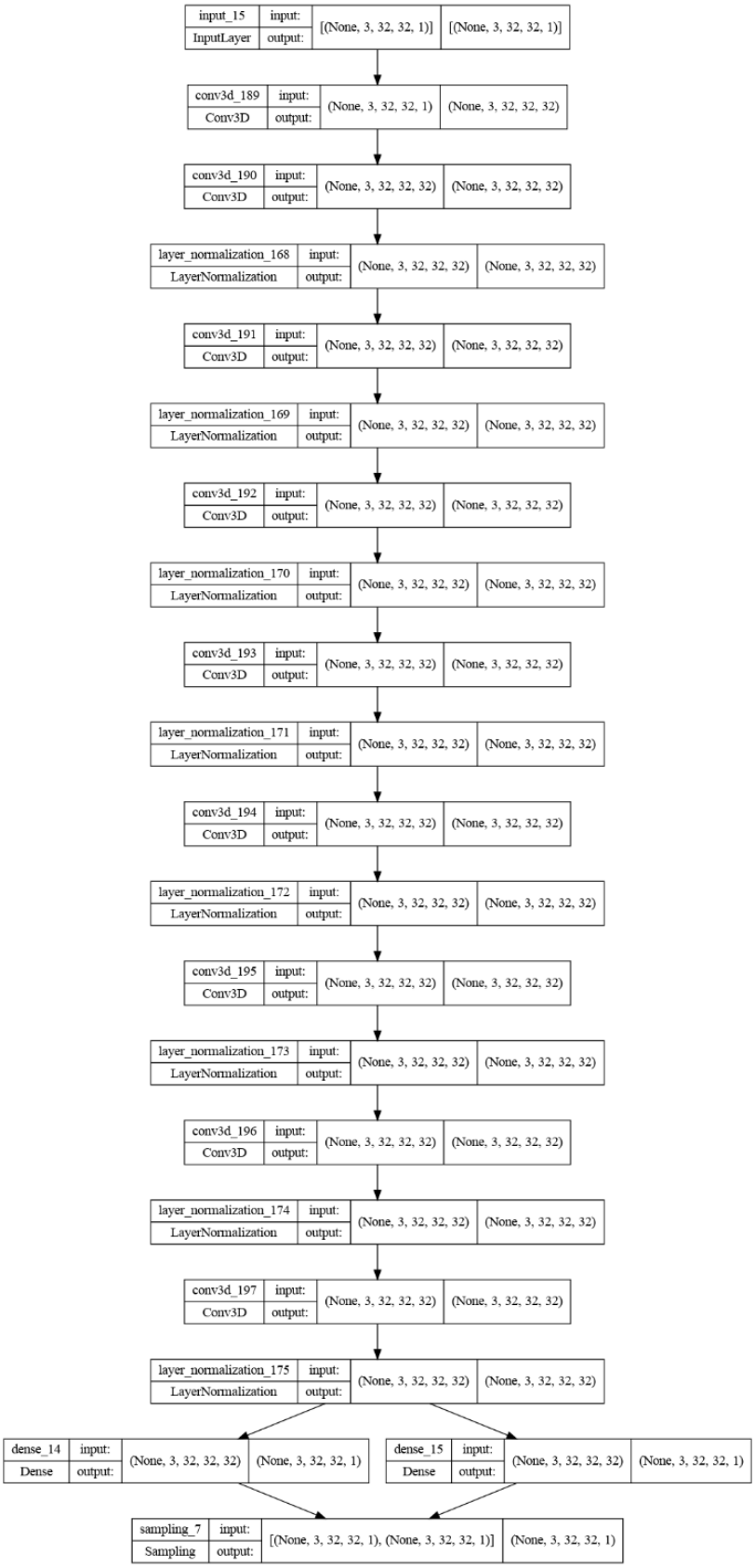


Figure 87: Architecture of the encoder of the ConvVAE.
Own image.

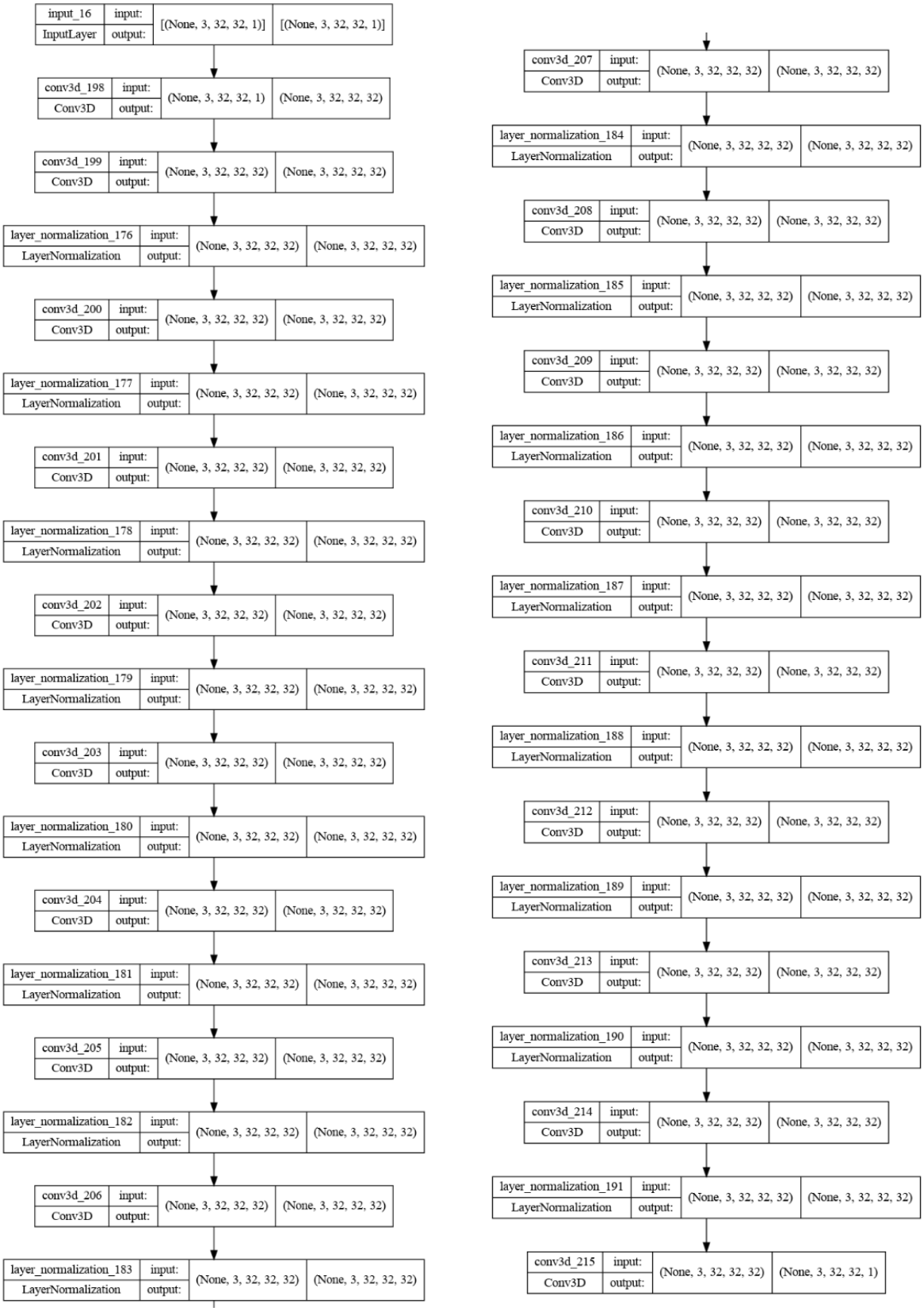


Figure 88: Architecture of the decoder of the ConvVAE. The left graphic shows the first half, and the right graphic shows the second half of the architecture.
Own image.

6 Results:

A sizeable amount of academic work has focused on suggesting new machine learning models that predict future frames of a video sequence based upon past frames. There are *Folded Recurrent Neural Networks* as presented in [138], Convolutional LSTM architectures like *Next-Frame* [38], generative adversarial networks (GANs) [139] such *FutureGAN* [61], or new combinations of existing architectures as presented in [140] that use an *Encoder-Decoder* architecture as their base to decouple a video's motion from its content. Most new studies focus on achieving slight enhancements over previous results. There is, however, a lack of tackling entirely new problem formulations, as was done in this thesis.

Previous studies in time series prediction have primarily focused on taking multiple existing time steps and using these to predict one or multiple following steps. Predicting time steps between a start and an endpoint has not gathered as much attention, aside from work on slow-motion methods for video as mentioned in chapter 3.6. This state is compounded by the fact that frame prediction in the sub-field of computer vision has just recently started taking off since machine learning methods on simple image data have become more prevalent. This thesis shows that predicting multiple frames between two rasterized graphics is possible. Various contrasting approaches have shown that this task is far from trivial. Training a machine learning model to create spatially and temporally consistent outputs is challenging. But the presented akaNet as discussed in chapter 5.7.7 has shown great promise. This CNN generated high-quality results, especially in combination with the exhibited approach of recursive triads. When discussing the stated goal of creating animations for preview purposes, akaNet's quality already suffices, under the condition that enough similar training material is available for the task at hand.

This can be compared to the standard workflow in video post-production and animation of using proxy files when talking about preview quality. Proxy video files are of a lower resolution than the original files and allow for faster processes due to smaller file sizes. They will only be exchanged for the original full-resolution material at the end when a project is finalized. A similar approach can be made possible with combination of recursive triads and akaNet. When starting an animation project, the animator can use akaNet to generate quick previews that get replaced with more fine-tuned or hand-made animations later. This could potentially save significant amounts of time, as the animator spends less time on reworking animations when receiving new feedback.

The presented novel workflow of using recursive triads, as presented in chapter 5.2 has shown a remarkable ability to influence the results of multiple different machine learning architectures positively. Another discovery made when comparing the various machine learning models presented in this paper is that architectures based on convolutional layers offer the best performance. They are better qualified to keep spatial consistency in their predictions. While other architectures can achieve results that display temporal progress, this alone is not enough for animations. But unanticipatedly, the use of max-pooling layers for subsampling, which are very common in many

convolutional architectures, often leads to a decreased output quality, as was exhibited in chapter 5.7.4. A better strategy is the use of strides inside convolutional layers.

One caveat regarding the results presented in this thesis is that these were achieved in an academic setup and not necessarily under real-world circumstances. The training stage utilized the Moving MNIST dataset, which was envisioned precisely for being used in a machine learning research scenario. It is challenging to collect a large enough dataset while keeping uniformity between its elements. But as this paper has demonstrated that automatically generating frames between two rasterized graphics is possible, collecting real-world datasets should be well worth it.

The sub-chapters of 5.6 have also demonstrated that recurrent architectures are ill-suited for predicting future frames based on non-consecutive frames. When only being fed with a start and end graphic, they failed to complete the animation in between. This was the case for both conventionally and recursively trained networks, and this finding is reinforced by the fact that even the Conv-LSTM from chapter 6.7.1 performs similarly poorly to a basic RNN. In contrast, a straightforward standard CNN like bbCNN in chapter 5.7.3 performs very well on this task.

Based on intuition, the shortcomings of recurrent networks are likely because the core of these architectures, the memory cell, expects the state of the previous timestep to correspond to the previous frame and that the data in its entirety is consecutive. With the recursive triads being independent of one another, the recurrent part of the network is essentially ignored [142].

It is also noteworthy that video frame prediction noticeably benefits from multifaceted approaches, as demonstrated in chapters 5.2 and 5.3 concerning the recursive triads and post-processing. When considering only a model's results, it was often not readily apparent whether the predictions were acceptable in quality. Only after augmenting the training and prediction process with recursive triads and the final results with post-processing did it become evident whether the results were actually appealing or not.

Table 10 compares all the final models, trained recursively. The training time measures only the time it takes to fit the model but may contain epochs after reaching convergence, as the model will only notice after a set number of epochs whether it has reached convergence. Both the loss value and the validation loss value are taken from the best epoch, with the lowest validation loss, which is not necessarily the last epoch.

Model	Loss	Validation loss	Training time (minutes)	Chapter
MLP	0.0085	0.0088	~1.4	5.5
RNN	0,0114	0,0115	~3.9	5.6.1
LSTM	0,0158	0,0125	~4.2	5.6.2
GRU	0,0105	0,0106	~6.2	5.6.3
ConvLSTM	0,0523	0,0518	~68.6	0
ResNet-34	0,0203	0,0221	~31.3	5.7.2
bbCNN	0,0055	0,0055	~98	5.7.3
akaNet	***0.0044***	***0.0045***	***~111.12***	***5.7.7***
ConvVAE	0,0046	0,0047	~880.2	5.7.8

Table 10: Comparison of the results of all presented models on the Moving MNIST dataset. The best model is highlighted in bold and italic.

A noteworthy factor in these results is that they were achieved on a single consumer-level computer as listed in chapter 2.3. Reproducing these results does not require the presence of a large GPU render farm or resorting to cloud resources. While the presented models were all trained in an offline approach, online training on either a local machine or a master system would be a logical next step to improve model performance continually. Whenever a user manually creates a new animation, this could be fed into the machine learning model to improve it.

7 Discussion:

To my knowledge, this was the first research delving into the automatic generation of animations between an input and an output frame through machine learning, specifically for rasterized graphics. I have presented three main contributions. The first one is the algorithm for training a machine learning model with recursive triads in chapter 5.2. The proposed recursive triads should be further investigated in other time-series prediction tasks or even unrelated tasks. The second contribution is the comparison of different machine learning architectures for the task of creating animations between two graphics in chapters 5.5 to 5.7. And the third contribution is the presentation of akaNet, a very promising convolutional neural network specifically constructed for this task in chapter 5.7.7.

These contributions aid in understanding AI-driven animation and hopefully present an incentive to incorporate more machine learning discoveries into animation software. Also, the presented results might be seen as a starting point for further research into improving these results, as has happened in previous years for the problem formulation of frame prediction based on sequences. In his videos about the latest machine learning

papers, Dr. Károly Zsolnai-Fehér [143] succinctly puts it: "*Just imagine what we will be able to do just a couple more papers down the line.*" [144], regarding the general progress made between research papers in a very short time frame.

Further research should be conducted to test the performance of generative adversarial networks (GANs) [139], which are popular for generating images, capsule networks [145], which were at least in part conceived as a replacement for CNNs, and the increasingly popular diffusion approach [146]. The diffusion approach is becoming progressively more popular [147] and has reached wide attention with OpenAI's [148] DALL·E 2 [149], [150], which has been presented to the public in April 2022.

Where the animation process was advanced from drawing every single frame by hand to only drawing keyframes, with an automatic interpolation generated by the computer in between, an AI-driven animation process could be the next advancement for this industry. This study's akaNet architecture has shown the capacity to automate parts of an animator's task.

The biggest hurdle to overcome is collecting a large, preferably homogonous, dataset. Many commercial products in the post-production industry already collect vast amounts of user data. Adobe is one of the most prominent software proprietors in this space and states the following in their official privacy policy:

"*We collect information about how you use our Services and Software, including when you use a desktop app feature that takes you online (such as a photo syncing feature). Depending on the Services and Software, this information may be associated with your device or browser or it may be associated with your Adobe account. It includes: […] Analysis of your content (e.g., documents, photos, videos, activity logs, direct feedback from you, and metadata about your content)* […]." [151].

It is conceivable to incorporate data acquisition into commercial products such as Adobe After Effects to capture manually created animations and use them to train a machine learning model. Once trained on a large real-world dataset, it could be utilized by the software's users to enhance new animations. This is not a far leap since Adobe is already invested in machine learning with their Sensei AI tool [152]. While the necessary form of data acquisition is already covered under Adobe's privacy policy and hence possible, one should still consider the implications on the user's privacy and how such a system could affect copyright or intellectual property [153], [154].

Consistently collecting new user data to improve machine learning models would be similar to how GitHub's Copilot [155] can aid software developers by automatically generating entire code blocks. Copilot was trained on public code and text on the internet, GitHub itself being one of the largest databases of open-source repositories. And it keeps learning from the programmers using it to improve suggestions. While training on an extensive database such as GitHub is a very advantageous starting position, there is no reason to believe that a similar approach would not work in the animation industry.

A different process would be to keep the model training local and have post-production companies train these models only on their own data. This would result in an overall less powerful machine learning model but keep the data on-premises. There is a non-negligible number of people in the post-production industry who prefer to keep all their data out of the cloud. The main reasons are data privacy and security concerns [156], [157], [158].

"*Other industries have been using the cloud for years, but broadcasters have been reluctant to embrace it—until COVID-19 forced the issue of enabling remote work. Still, it's an uneasy embrace, and misconceptions about and mistrust of the cloud still linger in the industry."* [159]

Another issue is costs associated with high utilization of the cloud [160]. S. Wang and M. Casado observe a trend of some large enterprises like Dropbox "*repatriating*" the majority of work back on-premise or adapting a hybrid compromise [161]. The general trend, however, is to use cloud resources more and more [162], [163]. All machine learning architectures presented in this thesis can easily work on-premise and in the cloud.

8 Conclusion:

This thesis demonstrates that the automatic creation of animations between two frames of rasterized graphics is possible by utilizing relatively simple machine learning algorithms. I have compared several approaches and highlighted which promising directions to investigate further. During this process, I have presented the architecture for an auspicious convolutional neural network, akaNet as presented in chapter 5.7.7, that can generate animations. This result has been augmented by the proposed approach of recursive triads as displayed in chapter 5.2, which can be applied to various architectures and significantly improve predictions.

I also make a case for keeping machine learning architectures simple and not blindly following existing or popular setups. Rejecting the use of popular techniques such as max-pooling and batch normalization can yield surprisingly good results while reducing complexity. I have also demonstrated that different versions of recurrent neural networks are not up to the task. In contrast, CNN-based networks made for object detection can be adapted to creating animations.

9 References

[1] Y. Li, C. Fang, J. Yang, Z. Wang, X. Lu, and M.-H. Yang, 'Flow-Grounded Spatial-Temporal Video Prediction from Still Images', in *Computer Vision – ECCV 2018*, vol. 11213, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds. Cham: Springer International Publishing, 2018, pp. 609–625. doi: 10.1007/978-3-030-01240-3_37.

[2] W. Bao, W.-S. Lai, C. Ma, X. Zhang, Z. Gao, and M.-H. Yang, 'Depth-Aware Video Frame Interpolation', Apr. 2019, Accessed: Dec. 12, 2021. [Online]. Available: http://arxiv.org/pdf/1904.00830v1

[3] K. Aberman, J. Liao, M. Shi, D. Lischinski, B. Chen, and D. Cohen-Or, 'Neural Best-Buddies: Sparse Cross-Domain Correspondence', *ACM Trans. Graph.*, vol. 37, no. 4, pp. 1–14, Aug. 2018, doi: 10.1145/3197517.3201332.

[4] A. Holynski, B. Curless, S. M. Seitz, and R. Szeliski, 'Animating Pictures with Eulerian Motion Fields'. arXiv, Nov. 30, 2020. doi: 10.48550/arXiv.2011.15128.

[5] A. Carlier, M. Danelljan, A. Alahi, and R. Timofte, 'DeepSVG: A Hierarchical Generative Network for Vector Graphics Animation', *ArXiv200711301 Cs*, Oct. 2020, Accessed: Nov. 23, 2021. [Online]. Available: http://arxiv.org/abs/2007.11301

[6] J. Langenbahn, *SVG Logo Animation using Machine Learning*. 2021. Accessed: Nov. 23, 2021. [Online]. Available: https://github.com/JakobLangenbahn/animate_logos

[7] 'SVG: Scalable Vector Graphics | MDN'. https://developer.mozilla.org/en-US/docs/Web/SVG (accessed Jun. 24, 2022).

[8] Nitish Srivastava, Elman Mansimov, Ruslan Salakhutdinov, 'Unsupervised Learning of Video Representations using LSTMs', *unsupervised_video*. http://www.cs.toronto.edu/~nitish/unsupervised_video/ (accessed May 10, 2022).

[9] ICML IJCAI ECAI 2018 Conference Videos, *AAAI 20 / AAAI 2020 Keynotes Turing Award Winners Event / Geoff Hinton, Yann Le Cunn, Yoshua Bengio*, (Feb. 10, 2020). Accessed: Jun. 13, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=UX8OubxsY8w

[10] 'Self-supervised learning: The dark matter of intelligence'. https://ai.facebook.com/blog/self-supervised-learning-the-dark-matter-of-intelligence/ (accessed Jun. 13, 2022).

[11] J. C. Chang, S. Amershi, and E. Kamar, 'Revolt: Collaborative Crowdsourcing for Labeling Machine Learning Datasets', in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*, New York, NY, USA, May 2017, pp. 2334–2346. doi: 10.1145/3025453.3026044.

[12] C. Vondrick, H. Pirsiavash, and A. Torralba, 'Anticipating Visual Representations from Unlabeled Video'. arXiv, Nov. 29, 2016. Accessed: Jun. 13, 2022. [Online]. Available: http://arxiv.org/abs/1504.08023

[13] C. Vondrick, A. Shrivastava, A. Fathi, S. Guadarrama, and K. Murphy, 'Tracking Emerges by Colorizing Videos'. arXiv, Jul. 27, 2018. Accessed: Jun. 13, 2022. [Online]. Available: http://arxiv.org/abs/1806.09594

[14] S. Oprea *et al.*, 'A Review on Deep Learning Techniques for Video Prediction', *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 6, pp. 1–26, Jun. 2022, doi: 10.1109/TPAMI.2020.3045007.
[15] Z. Liu, R. A. Yeh, X. Tang, Y. Liu, and A. Agarwala, 'Video Frame Synthesis using Deep Voxel Flow'. arXiv, Aug. 05, 2017. Accessed: Jun. 11, 2022. [Online]. Available: http://arxiv.org/abs/1702.02463
[16] 'Introduction to Tensors | TensorFlow Core', *TensorFlow*. https://www.tensorflow.org/guide/tensor (accessed Jun. 21, 2022).
[17] W. Koehrsen, 'Overfitting vs. Underfitting: A Complete Example', *Medium*, Jan. 28, 2018. https://towardsdatascience.com/overfitting-vs-underfitting-a-complete-example-d05dd7e19765 (accessed Jun. 29, 2022).
[18] H. Kinsley and D. Kukieła, *Neural Networks from Scratch in Python*, 1st ed. (n.p.).
[19] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard and L. D. Jackel, 'Backpropagation Applied to Handwritten Zip Code Recognition', in *Neural Computation*, AT&T Bell Laboratories Holmdel, NJ 07733 USA, 1989, vol. 1, pp. 541–551. Accessed: Jun. 14, 2022. [Online]. Available: http://yann.lecun.com/exdb/publis/pdf/lecun-89e.pdf
[20] M. Mathieu, C. Couprie, and Y. LeCun, 'Deep multi-scale video prediction beyond mean square error'. arXiv, Feb. 26, 2016. Accessed: Jun. 11, 2022. [Online]. Available: http://arxiv.org/abs/1511.05440
[21] S. Oprea *et al.*, 'A Review on Deep Learning Techniques for Video Prediction', *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 6, p. 17, Apr. 2020, doi: 10.1109/TPAMI.2020.3045007.
[22] S. Mandt, M. D. Hoffman, and D. M. Blei, 'Stochastic Gradient Descent as Approximate Bayesian Inference', *ArXiv170404289 Cs Stat*, Jan. 2018, Accessed: May 11, 2022. [Online]. Available: http://arxiv.org/abs/1704.04289
[23] D. P. Kingma and J. Ba, 'Adam: A Method for Stochastic Optimization', *ArXiv14126980 Cs*, Jan. 2017, Accessed: May 11, 2022. [Online]. Available: http://arxiv.org/abs/1412.6980
[24] H. Kinsley and D. Kukieła, in *Neural Networks from Scratch in Python*, 1st ed., (n.p.), p. 304.
[25] T. Dozat, 'Incorporating Nesterov Momentum into Adam', pp. 1–6.
[26] AlphaOpt, *Introduction To Optimization: Gradient Based Algorithms*, (Mar. 29, 2017). Accessed: Jul. 03, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=n-Y0SDSOfUI
[27] 'Project Jupyter'. https://jupyter.org (accessed May 10, 2022).
[28] 'Anaconda | The World's Most Popular Data Science Platform', *Anaconda*. https://www.anaconda.com/ (accessed May 10, 2022).
[29] 'TensorFlow', *TensorFlow*. https://www.tensorflow.org/ (accessed May 10, 2022).
[30] 'Keras: the Python deep learning API'. https://keras.io/ (accessed May 10, 2022).
[31] 'NumPy'. https://numpy.org/ (accessed May 10, 2022).

[32] ‘Matplotlib — Visualization with Python’. https://matplotlib.org/ (accessed May 11, 2022).
[33] ‘Netron’. https://netron.app/ (accessed May 21, 2022).
[34] T. Geijtenbeek, M. van de Panne, and A. F. van der Stappen, ‘Flexible muscle-based locomotion for bipedal creatures’, *ACM Trans. Graph.*, vol. 32, no. 6, pp. 1–11, 2013, doi: 10.1145/2508363.2508399.
[35] X. Zhang and M. van de Panne, ‘Data-driven autocompletion for keyframe animation’, in *Proceedings of the 11th Annual International Conference on Motion, Interaction, and Games*, Limassol Cyprus, Nov. 2018, pp. 1–11. doi: 10.1145/3274247.3274502.
[36] D. Weber and C. Gühmann, ‘Non-Autoregressive vs Autoregressive Neural Networks for System Identification’. arXiv, May 05, 2021. Accessed: Jun. 29, 2022. [Online]. Available: http://arxiv.org/abs/2105.02027
[37] X. Shi, Z. Chen, H. Wang, D.-Y. Yeung, W. Wong, and W. Woo, ‘Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting’, *ArXiv150604214 Cs*, Sep. 2015, Accessed: Dec. 08, 2021. [Online]. Available: http://arxiv.org/abs/1506.04214
[38] K. Team, ‘Keras documentation: Next-Frame Video Prediction with Convolutional LSTMs’. https://keras.io/examples/vision/conv_lstm/ (accessed Dec. 15, 2021).
[39] W. Lotter, G. Kreiman, and D. Cox, ‘Deep Predictive Coding Networks for Video Prediction and Unsupervised Learning’. arXiv, Feb. 28, 2017. Accessed: Jun. 11, 2022. [Online]. Available: http://arxiv.org/abs/1605.08104
[40] Y. Wang, M. Long, J. Wang, Z. Gao, and P. S. Yu, ‘PredRNN: Recurrent Neural Networks for Predictive Learning using Spatiotemporal LSTMs’, p. 10.
[41] J. Walker, C. Doersch, A. Gupta, and M. Hebert, ‘An Uncertain Future: Forecasting from Static Images using Variational Autoencoders’. arXiv, Jun. 25, 2016. Accessed: Jun. 11, 2022. [Online]. Available: http://arxiv.org/abs/1606.07873
[42] T. Xue, J. Wu, K. L. Bouman, and W. T. Freeman, ‘Visual Dynamics: Probabilistic Future Frame Synthesis via Cross Convolutional Networks’, p. 9.
[43] C. Vondrick, H. Pirsiavash, and A. Torralba, ‘Generating Videos with Scene Dynamics’. arXiv, Oct. 26, 2016. Accessed: Jun. 10, 2022. [Online]. Available: http://arxiv.org/abs/1609.02612
[44] M. Babaeizadeh, C. Finn, D. Erhan, R. H. Campbell, and S. Levine, ‘Stochastic Variational Video Prediction’. arXiv, Mar. 06, 2018. Accessed: Jun. 12, 2022. [Online]. Available: http://arxiv.org/abs/1710.11252
[45] A. X. Lee, R. Zhang, F. Ebert, P. Abbeel, C. Finn, and S. Levine, ‘Stochastic Adversarial Video Prediction’. arXiv, Apr. 04, 2018. Accessed: Jun. 12, 2022. [Online]. Available: http://arxiv.org/abs/1804.01523
[46] E. L. Denton, ‘Unsupervised Learning of Disentangled Representations from Video’, p. 10.

[47] E. Denton and R. Fergus, ‘Stochastic Video Generation with a Learned Prior’. arXiv, Mar. 02, 2018. Accessed: Jun. 12, 2022. [Online]. Available: http://arxiv.org/abs/1802.07687
[48] N. Kalchbrenner *et al.*, ‘Video Pixel Networks’. arXiv, Oct. 03, 2016. Accessed: Jun. 11, 2022. [Online]. Available: http://arxiv.org/abs/1610.00527
[49] N. Srivastava, E. Mansimov, and R. Salakhutdinov, ‘Unsupervised Learning of Video Representations using LSTMs’, *ArXiv150204681 Cs*, Jan. 2016, Accessed: May 04, 2022. [Online]. Available: http://arxiv.org/abs/1502.04681
[50] J. Lee, J. Lee, S. Lee, and S. Yoon, ‘Mutual Suppression Network for Video Prediction using Disentangled Features’, *ArXiv180404810 Cs*, Jul. 2019, Accessed: Dec. 08, 2021. [Online]. Available: http://arxiv.org/abs/1804.04810
[51] C.-Z. A. Huang *et al.*, ‘The Bach Doodle: Approachable music composition with machine learning at scale’, *ArXiv190706637 Cs Eess Stat*, Jul. 2019, Accessed: Dec. 08, 2021. [Online]. Available: http://arxiv.org/abs/1907.06637
[52] C.-Z. A. Huang, T. Cooijmans, A. Roberts, A. Courville, and D. Eck, ‘Counterpoint by Convolution’, *ArXiv190307227 Cs Eess Stat*, Mar. 2019, Accessed: Dec. 09, 2021. [Online]. Available: http://arxiv.org/abs/1903.07227
[53] A. Holynski, B. Curless, S. M. Seitz, and R. Szeliski, ‘Animating Pictures with Eulerian Motion Fields’, Nov. 2020, Accessed: Jan. 23, 2022. [Online]. Available: http://arxiv.org/pdf/2011.15128v1
[54] James L. Gibson, *The Perception of the Visual World*. Houghton Mifflin, 1950.
[55] B. K. P. Horn and B. G. Schunck, ‘Determining optical flow’, *Artif. Intell.*, vol. 17, no. 1–3, pp. 185–203, Aug. 1981, doi: 10.1016/0004-3702(81)90024-2.
[56] S. S. Beauchemin and J. L. Barron, ‘The computation of optical flow’, *ACM Comput. Surv.*, vol. 27, no. 3, pp. 433–466, Sep. 1995, doi: 10.1145/212094.212141.
[57] ‘Change Clip Speed and Duration in Adobe Premiere Pro’. https://helpx.adobe.com/premiere-pro/using/duration-speed.html (accessed Jun. 16, 2022).
[58] ‘OFlow Retiming’. https://learn.foundry.com/nuke/9.0/content/comp_environment/temporal_operations/oflow_retiming.html (accessed Jun. 16, 2022).
[59] J. Revaud, P. Weinzaepfel, Z. Harchaoui, and C. Schmid, ‘EpicFlow: Edge-Preserving Interpolation of Correspondences for Optical Flow’. arXiv, May 19, 2015. Accessed: Jun. 16, 2022. [Online]. Available: http://arxiv.org/abs/1501.02565
[60] N. Sedaghat, M. Zolfaghari, and T. Brox, ‘Hybrid Learning of Optical Flow and Next Frame Prediction to Boost Optical Flow in the Wild’. arXiv, Apr. 07, 2017. Accessed: Jun. 16, 2022. [Online]. Available: http://arxiv.org/abs/1612.03777
[61] S. Aigner and M. Körner, ‘FutureGAN: Anticipating the Future Frames of Video Sequences using Spatio-Temporal 3d Convolutions in Progressively Growing GANs’. arXiv, Nov. 26, 2018. Accessed: Jun. 10, 2022. [Online]. Available: http://arxiv.org/abs/1810.01325

[62] J. Walker, A. Gupta, and M. Hebert, ‘Dense Optical Flow Prediction from a Static Image’, *ArXiv150500295 Cs*, Dec. 2015, Accessed: Nov. 25, 2021. [Online]. Available: http://arxiv.org/abs/1505.00295
[63] E. Herbst, S. Seitz, and S. Baker, ‘Occlusion Reasoning for Temporal Interpolation using Optical Flow’, p. 41.
[64] D. J. Butler, J. Wulff, G. B. Stanley, and M. J. Black, ‘A Naturalistic Open Source Movie for Optical Flow Evaluation’, in *Computer Vision – ECCV 2012*, vol. 7577, A. Fitzgibbon, S. Lazebnik, P. Perona, Y. Sato, and C. Schmid, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 611–625. doi: 10.1007/978-3-642-33783-3_44.
[65] S. Niklaus, L. Mai, and F. Liu, ‘Video Frame Interpolation via Adaptive Convolution’. arXiv, Mar. 22, 2017. Accessed: Jun. 11, 2022. [Online]. Available: http://arxiv.org/abs/1703.07514
[66] H. Jiang, D. Sun, V. Jampani, M.-H. Yang, E. Learned-Miller, and J. Kautz, ‘Super SloMo: High Quality Estimation of Multiple Intermediate Frames for Video Interpolation’, in *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, Salt Lake City, UT, Jun. 2018, pp. 9000–9008. doi: 10.1109/CVPR.2018.00938.
[67] G. Poetsch, *Dain-App: Application for Video Interpolationsl*. 2020. Accessed: Feb. 10, 2022. [Online]. Available: https://github.com/BurguerJohn/Dain-App
[68] ‘MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges’. http://yann.lecun.com/exdb/mnist/ (accessed Feb. 06, 2022).
[69] S. Oprea *et al.*, ‘A Review on Deep Learning Techniques for Video Prediction’, *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 44, no. 6, pp. 7–8, Apr. 2020, doi: 10.1109/TPAMI.2020.3045007.
[70] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*, Second Edition., vol. 2, 2 vols. Sebastopol, CA: O’Reilly, 2019.
[71] ‘What is a Lottie animation? - LottieFiles’. https://lottiefiles.com/what-is-lottie (accessed Jan. 28, 2022).
[72] ‘Beautiful Soup Documentation — Beautiful Soup 4.4.0 documentation’. https://beautiful-soup-4.readthedocs.io/en/latest/ (accessed Dec. 12, 2021).
[73] ‘Selenium’, *Selenium*. https://www.selenium.dev/ (accessed Dec. 12, 2021).
[74] ‘puppeteer-lottie’, *npm*. https://www.npmjs.com/package/puppeteer-lottie (accessed Dec. 12, 2021).
[75] ‘gifski — highest-quality GIF converter’. https://gif.ski/ (accessed Dec. 12, 2021).
[76] ‘FFmpeg’. https://ffmpeg.org/ (accessed Dec. 26, 2021).
[77] ‘Jagged array’, *Wikipedia*. Mar. 11, 2022. Accessed: Jul. 03, 2022. [Online]. Available: https://en.wikipedia.org/w/index.php?title=Jagged_array&oldid=1076413658
[78] *opencv/opencv*. OpenCV, 2021. Accessed: Dec. 26, 2021. [Online]. Available: https://github.com/opencv/opencv

[79] A. Mikołajczyk and M. Grochowski, *Data augmentation for improving deep learning in image classification problem*. 2018, p. 122. doi: 10.1109/IIPHDW.2018.8388338.
[80] S. Lloyd, 'Least squares quantization in PCM', *IEEE Trans. Inf. Theory*, vol. 28, no. 2, pp. 1–9, Mar. 1982, doi: 10.1109/TIT.1982.1056489.
[81] D. Arthur and S. Vassilvitskii, 'K-Means++: The Advantages of Careful Seeding', Jan. 2007, vol. 8, pp. 2–3. doi: 10.1145/1283383.1283494.
[82] DeepLearningAI, *Parameters vs Hyperparameters (C1W4L07)*, (Aug. 25, 2017). Accessed: Jun. 27, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=VTE2KlfoO3Q
[83] K. Team, 'Keras documentation: KerasTuner'. https://keras.io/keras_tuner/ (accessed May 20, 2022).
[84] M.-C. Popescu, V. Balas, L. Perescu-Popescu, and N. Mastorakis, 'Multilayer perceptron and neural networks', *WSEAS Trans. Circuits Syst.*, vol. 8, p. 587, Jul. 2009.
[85] A. Géron, 'The Multilayer Perceptron and Backpropagation', in *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*, Second Edition., vol. 2, 2 vols, Sebastopol, CA: O'Reilly, 2019, pp. 289–290.
[86] DeepLearningAI, *Forward Propagation in a Deep Network (C1W4L02)*, (Aug. 25, 2017). Accessed: Jun. 27, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=a8i2eJin0lY
[87] DeepLearningAI, *Forward and Backward Propagation (C1W4L06)*, (Aug. 25, 2017). Accessed: Jun. 27, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=qzPQ8cEsVK8
[88] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, 'Learning Internal Representations by Error Propagation', CALIFORNIA UNIV SAN DIEGO LA JOLLA INST FOR COGNITIVE SCIENCE, Sep. 1985. Accessed: Apr. 12, 2022. [Online]. Available: https://apps.dtic.mil/sti/citations/ADA164453
[89] 'Machine learning - Breakthrough science and technologies: Transforming our future conference series | Royal Society'. https://royalsociety.org/science-events-and-lectures/2015/05/breakthrough-science-technologies-machine-learning/ (accessed May 15, 2022).
[90] J. Dsouza, 'What is a GPU and do you need one in Deep Learning?', *Medium*, Dec. 26, 2020. https://towardsdatascience.com/what-is-a-gpu-and-do-you-need-one-in-deep-learning-718b9597aa0d (accessed May 15, 2022).
[91] H. Salehinejad, S. Sankar, J. Barfett, E. Colak, and S. Valaee, 'Recent Advances in Recurrent Neural Networks'. arXiv, Feb. 22, 2018. Accessed: Jun. 20, 2022. [Online]. Available: http://arxiv.org/abs/1801.01078
[92] S. Ioffe and C. Szegedy, 'Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift'. arXiv, Mar. 02, 2015. Accessed: Jun. 12, 2022. [Online]. Available: http://arxiv.org/abs/1502.03167
[93] K. He, X. Zhang, S. Ren, and J. Sun, 'Deep Residual Learning for Image Recognition', arXiv, arXiv:1512.03385, Dec. 2015. doi: 10.48550/arXiv.1512.03385.

[94] C. Lei, Y. Xing, and Q. Chen, ‘Blind Video Temporal Consistency via Deep Video Prior’. arXiv, Oct. 22, 2020. Accessed: Jun. 27, 2022. [Online]. Available: http://arxiv.org/abs/2010.11838
[95] L. Medsker and L. C. Jain, ‘Overview’, in *Recurrent Neural Networks: Design and Applications*, CRC Press, 1999, p. 1. Accessed: Jun. 19, 2022. [Online]. Available: https://books.google.de/books?hl=de&lr=&id=ME1SAkN0PyMC&oi=fnd&pg=PA1&dq=recurrent+neural+networks&ots=7cxvgN8OXj&sig=L23gLQ8sJ6nV72ggXlHdYwrI5ro#v=onepage&q&f=false
[96] K. Team, ‘Keras documentation: SimpleRNN layer’. https://keras.io/api/layers/recurrent_layers/simple_rnn/ (accessed Jun. 21, 2022).
[97] A. Géron, ‘Memory Cells’, in *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*, Second Edition., vol. 2, 2 vols, Sebastopol, CA: O’Reilly, 2019, pp. 500–501.
[98] Z. C. Lipton, J. Berkowitz, and C. Elkan, ‘A Critical Review of Recurrent Neural Networks for Sequence Learning’. arXiv, Oct. 17, 2015. Accessed: Jun. 20, 2022. [Online]. Available: http://arxiv.org/abs/1506.00019
[99] S. Hochreiter and J. Schmidhuber, ‘Long Short-term Memory’, *Neural Comput.*, vol. 9, pp. 1735–80, Dec. 1997, doi: 10.1162/neco.1997.9.8.1735.
[100] O. Levy, K. Lee, N. FitzGerald, and L. Zettlemoyer, ‘Long Short-Term Memory as a Dynamically Computed Element-wise Weighted Sum’. arXiv, May 09, 2018. Accessed: Jun. 19, 2022. [Online]. Available: http://arxiv.org/abs/1805.03716
[101] A. Sherstinsky, ‘Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network’, *Phys. Nonlinear Phenom.*, vol. 404, p. 40, Mar. 2020, doi: 10.1016/j.physd.2019.132306.
[102] K. Cho, B. van Merrienboer, D. Bahdanau, and Y. Bengio, ‘On the Properties of Neural Machine Translation: Encoder-Decoder Approaches’. arXiv, Oct. 07, 2014. Accessed: Jun. 27, 2022. [Online]. Available: http://arxiv.org/abs/1409.1259
[103] S. Yang, X. Yu, and Y. Zhou, ‘LSTM and GRU Neural Network Performance Comparison Study: Taking Yelp Review Dataset as an Example’, in *2020 International Workshop on Electronic Communication and Artificial Intelligence (IWECAI)*, Shanghai, China, Jun. 2020, p. 4. doi: 10.1109/IWECAI50956.2020.00027.
[104] CodeEmporium, *Convolution Neural Networks - EXPLAINED*, (Feb. 07, 2018). Accessed: Jun. 28, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=m8pOnJxOcqY
[105] Y. LeCun, L. Bottou, Y. Bengio, and P. Ha, ‘Gradient-Based Learning Applied to Document Recognition’, *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998, doi: 10.1109/5.726791.
[106] CodeEmporium, *What do filters of Convolution Neural Network learn?*, (Oct. 19, 2020). Accessed: Jun. 28, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=eL80Im8Hq0k

[107] A. Géron, ‘Convolutional Layers’, in *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*, Second Edition., vol. 2, 2 vols, Sebastopol, CA: O’Reilly, 2019, pp. 448–450.
[108] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, ‘Dropout: A Simple Way to Prevent Neural Networks from Overfitting’, *J. Mach. Learn. Res.*, vol. 15, no. 56, pp. 1929–1958, 2014.
[109] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, and Sean Ma, ‘ImageNet Large Scale Visual Recognition Challenge (ILSVRC)’, *IMAGENET*, May 24, 2017. https://image-net.org/challenges/LSVRC/ (accessed Jun. 14, 2022).
[110] A. Krizhevsky, I. Sutskever, and G. E. Hinton, ‘ImageNet classification with deep convolutional neural networks’, *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.
[111] K. He and J. Sun, ‘Convolutional Neural Networks at Constrained Time Cost’. arXiv, Dec. 04, 2014. Accessed: Jun. 25, 2022. [Online]. Available: http://arxiv.org/abs/1412.1710
[112] R. K. Srivastava, K. Greff, and J. Schmidhuber, ‘Training Very Deep Networks’. arXiv, Nov. 23, 2015. Accessed: Jun. 25, 2022. [Online]. Available: http://arxiv.org/abs/1507.06228
[113] A. Géron, ‘Pooling Layers’, in *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*, Second Edition., vol. 2, 2 vols, Sebastopol, CA: O’Reilly, 2019, pp. 456–458.
[114] B. Graham, ‘Fractional Max-Pooling’. arXiv, May 12, 2015. Accessed: Jun. 15, 2022. [Online]. Available: http://arxiv.org/abs/1412.6071
[115] C. Szegedy *et al.*, ‘Going deeper with convolutions’, in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Boston, MA, USA, Jun. 2015, pp. 1–9. doi: 10.1109/CVPR.2015.7298594.
[116] H. Gholamalinezhad and H. Khosravi, ‘Pooling Methods in Deep Neural Networks, a Review’, p. 16.
[117] D. Hutchison *et al.*, ‘Evaluation of Pooling Operations in Convolutional Architectures for Object Recognition’, in *Artificial Neural Networks – ICANN 2010*, vol. 6354, K. Diamantaras, W. Duch, and L. S. Iliadis, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 92–101. doi: 10.1007/978-3-642-15825-4_10.
[118] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. Cambridge, Massachusetts: The MIT Press, 2016.
[119] S. Parva, ‘Do we really need the Pooling layer in our CNN architecture? | LinkedIn’, *LinkedIn*, Jul. 29, 2020. https://www.linkedin.com/pulse/do-we-really-need-pooling-layer-our-cnn-architecture-parva-shah/ (accessed Jun. 07, 2022).
[120] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, ‘Striving for Simplicity: The All Convolutional Net’. arXiv, Apr. 13, 2015. Accessed: Jun. 07, 2022. [Online]. Available: http://arxiv.org/abs/1412.6806

[121] ‘Geoffrey E Hinton - A.M. Turing Award Laureate’. https://amturing.acm.org/award_winners/hinton_4791679.cfm (accessed Jun. 16, 2022).
[122] B. Marr, ‘Who’s Who: The 6 Top Thinkers In AI And Machine Learning’, *Forbes*. https://www.forbes.com/sites/bernardmarr/2017/05/09/whos-who-the-6-top-thinkers-in-ai-and-machine-learning/ (accessed Jun. 16, 2022).
[123] geoffhinton, ‘AMA Geoffrey Hinton’, *r/MachineLearning*, Nov. 07, 2014. www.reddit.com/r/MachineLearning/comments/2lmo0l/ama_geoffrey_hinton/ (accessed Jun. 16, 2022).
[124] trwappers, *Geoffrey Hinton talk ‘What is wrong with convolutional neural nets ?’*, (Apr. 03, 2017). Accessed: Jun. 16, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=rTawFwUvnLE
[125] D. Rivoir, I. Funke, and S. Speidel, ‘On the Pitfalls of Batch Normalization for End-to-End Video Learning: A Study on Surgical Workflow Analysis’. arXiv, Mar. 15, 2022. Accessed: Jun. 15, 2022. [Online]. Available: http://arxiv.org/abs/2203.07976
[126] S. Santurkar, D. Tsipras, A. Ilyas, and A. Ma, ‘How Does Batch Normalization Help Optimization?’, p. 11.
[127] N. Bjorck, C. P. Gomes, B. Selman, and K. Q. Weinberger, ‘Understanding Batch Normalization’, p. 12.
[128] A. Brock, S. De, S. L. Smith, and K. Simonyan, ‘High-Performance Large-Scale Image Recognition Without Normalization’. arXiv, Feb. 11, 2021. Accessed: Jun. 15, 2022. [Online]. Available: http://arxiv.org/abs/2102.06171
[129] Y. Wu and J. Johnson, ‘Rethinking “Batch” in BatchNorm’. arXiv, May 16, 2021. Accessed: Jun. 15, 2022. [Online]. Available: http://arxiv.org/abs/2105.07576
[130] Y. Wu and K. He, ‘Group Normalization’. arXiv, Jun. 11, 2018. Accessed: Jun. 16, 2022. [Online]. Available: http://arxiv.org/abs/1803.08494
[131] S. Uppal, ‘Curse of Batch Normalization’, *Medium*, Jul. 25, 2020. https://towardsdatascience.com/curse-of-batch-normalization-8e6dd20bc304 (accessed Jun. 13, 2022).
[132] J. L. Ba, J. R. Kiros, and G. E. Hinton, ‘Layer Normalization’. arXiv, Jul. 21, 2016. Accessed: Jun. 16, 2022. [Online]. Available: http://arxiv.org/abs/1607.06450
[133] D. Bank, N. Koenigstein, and R. Giryes, ‘Autoencoders’. arXiv, Apr. 03, 2021. Accessed: Jun. 29, 2022. [Online]. Available: http://arxiv.org/abs/2003.05991
[134] I. Goodfellow, Y. Bengio, and A. Courville, ‘Autoencoders’, in *Deep learning*, Cambridge, Massachusetts: The MIT Press, 2016, pp. 499–500. Accessed: Feb. 14, 2022. [Online]. Available: https://www.deeplearningbook.org/contents/autoencoders.html
[135] A. Géron, ‘Stacked autoencoders’, in *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*, Second Edition., vol. 2, 2 vols, Sebastopol, CA: O’Reilly, 2019, p. 572.

[136] D. P. Kingma and M. Welling, ‘Auto-Encoding Variational Bayes’, arXiv, arXiv:1312.6114, May 2014. doi: 10.48550/arXiv.1312.6114.
[137] A. Géron, ‘Variational Autoencoders’, in *Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow*, Second Edition., vol. 2, 2 vols, Sebastopol, CA: O’Reilly, 2019, p. 586.
[138] M. Oliu, J. Selva, and S. Escalera, ‘Folded Recurrent Neural Networks for Future Video Prediction’, in *Computer Vision – ECCV 2018*, vol. 11218, V. Ferrari, M. Hebert, C. Sminchisescu, and Y. Weiss, Eds. Cham: Springer International Publishing, 2018, pp. 745–761. doi: 10.1007/978-3-030-01264-9_44.
[139] I. J. Goodfellow *et al.*, ‘Generative Adversarial Networks’, *ArXiv14062661 Cs Stat*, Jun. 2014, Accessed: Feb. 06, 2022. [Online]. Available: http://arxiv.org/abs/1406.2661
[140] R. Villegas, J. Yang, S. Hong, X. Lin, and H. Lee, ‘Decomposing Motion and Content for Natural Video Sequence Prediction’. arXiv, Jan. 07, 2018. Accessed: Jun. 10, 2022. [Online]. Available: http://arxiv.org/abs/1706.08033
[141] S. Nah, S. Son, and K. M. Lee, ‘Recurrent Neural Networks With Intra-Frame Iterations for Video Deblurring’, in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Long Beach, CA, USA, Jun. 2019, pp. 8094–8103. doi: 10.1109/CVPR.2019.00829.
[142] ‘How does a recurrent neural network work for non-sequential data?’, *Quora*, Feb. 26, 2019. https://www.quora.com/How-does-a-recurrent-neural-network-work-for-non-sequential-data (accessed Jun. 05, 2022).
[143] ‘About’, *Károly Zsolnai-Fehér - Research Scientist*, Jan. 30, 2014. https://users.cg.tuwien.ac.at/zsolnai/about/ (accessed Jun. 25, 2022).
[144] Two Minute Papers, *DeepMind’s New AI Thinks It Is A Genius!* 🤖, (May 22, 2022). Accessed: Jun. 25, 2022. [Online Video]. Available: https://www.youtube.com/watch?v=aPiHhJjN3hI
[145] S. Sabour, N. Frosst, and G. E. Hinton, ‘Dynamic Routing Between Capsules’. arXiv, Nov. 07, 2017. Accessed: Jul. 01, 2022. [Online]. Available: http://arxiv.org/abs/1710.09829
[146] J. Ho, A. Jain, and P. Abbeel, ‘Denoising Diffusion Probabilistic Models’. arXiv, Dec. 16, 2020. Accessed: Jul. 01, 2022. [Online]. Available: http://arxiv.org/abs/2006.11239
[147] P. Dhariwal and A. Nichol, ‘Diffusion Models Beat GANs on Image Synthesis’. arXiv, Jun. 01, 2021. doi: 10.48550/arXiv.2105.05233.
[148] ‘About OpenAI’, *OpenAI*, Dec. 11, 2015. https://openai.com/about/ (accessed Jul. 01, 2022).
[149] A. Ramesh, P. Dhariwal, A. Nichol, C. Chu, and M. Chen, ‘Hierarchical Text-Conditional Image Generation with CLIP Latents’. arXiv, Apr. 12, 2022. Accessed: Jul. 02, 2022. [Online]. Available: http://arxiv.org/abs/2204.06125
[150] ‘DALL·E 2’, *OpenAI*. https://openai.com/dall-e-2/ (accessed Jul. 01, 2022).
[151] ‘Adobe Privacy Center’. https://www.adobe.com/privacy/policy.html (accessed May 23, 2022).

[152] ‘Adobe Sensei’. https://www.adobe.com/de/sensei.html (accessed May 23, 2022).
[153] D. Somaya and L. R. Varshney, ‘Embodiment, Anthropomorphism, and Intellectual Property Rights for AI Creations’, in *Proceedings of the 2018 AAAI/ACM Conference on AI, Ethics, and Society*, New Orleans LA USA, Dec. 2018, pp. 278–283. doi: 10.1145/3278721.3278754.
[154] R. Hilty, J. Hoffmann, and S. Scheuerer, ‘Intellectual Property Justification for Artificial Intelligence’. Rochester, NY, Feb. 11, 2020. Accessed: Jun. 26, 2022. [Online]. Available: https://papers.ssrn.com/abstract=3539406
[155] ‘GitHub Copilot · Your AI pair programmer’, *GitHub Copilot*. https://copilot.github.com/ (accessed May 22, 2022).
[156] ‘Addressing Post-Production Security Concerns in Remote Workflows’. https://www.avid.com/resource-center/addressing-post-production-security-concerns-in-remote-workflows (accessed Jun. 26, 2022).
[157] ‘Making the Media S2E03: Safe and Sound’. https://www.avid.com/resource-center/making-the-media-s2e03-safe-and-sound (accessed Jun. 26, 2022).
[158] A. Bondarenko, ‘The Problems of Cybersecurity in the Film & Media Industry’. https://www.cm-alliance.com/cybersecurity-blog/the-problems-of-cybersecurity-in-the-film-media-industry (accessed Jun. 26, 2022).
[159] ‘Making the Media S1E08: Who’s Afraid of the Big Bad Cloud?’ Accessed: Jun. 26, 2022. [Online]. Available: https://www.avid.com/resource-center/making-the-media-s1e08-whos-afraid-of-the-big-bad-cloud
[160] ‘Chancen und Herausforderungen bei Remote-Produktionen’, *film-tv-video.de*, Jan. 26, 2022. https://www.film-tv-video.de/productions/2022/01/26/chancen-und-herausforderungen-bei-remote-produktionen/ (accessed Jun. 26, 2022).
[161] S. Wang and M. Casado, ‘The Cost of Cloud, a Trillion Dollar Paradox’, *Andreessen Horowitz*, May 27, 2021. https://a16z.com/2021/05/27/cost-of-cloud-paradox-market-cap-cloud-lifecycle-scale-growth-repatriation-optimization/ (accessed Jun. 26, 2022).
[162] A. Pennington, ‘Postproduction continues to move at pace to the cloud’. https://www.redsharknews.com/postproduction-continues-to-move-at-pace-to-the-cloud (accessed Jun. 26, 2022).
[163] P. Poti, ‘The cloud is coming home’, *DIGITAL PRODUCTION*, Sep. 25, 2020. https://www.digitalproduction.com/2020/09/25/the-cloud-is-coming-home/ (accessed Jun. 26, 2022).

10 Appendices

10.1 Simplified Python code sample of the recursive triads algorithm

```
NUM_FRAMES = 10
RESOLUTION = 32
CHANNELS = 1

def create_triad_dataset(dataset):
    y_triad_dataset = np.zeros((len(dataset),
    NUM_FRAMES, 3, RESOLUTION, RESOLUTION, CHANNELS))

    for i, animation in enumerate(dataset):
        y_triads = create_triad_animation(animation, False)
        y_triads = np.concatenate(
            [y_triads, create_single_triad(
                animation, NUM_FRAMES - 2, NUM_FRAMES - 1, NUM_FRAMES - 1, False)],
axis=0)
        y_triads = np.concatenate(
            [create_single_triad(animation, 0, 0, 1, False), y_triads], axis=0)
        y_triad_dataset[i] = y_triads

    return y_triad_dataset

def create_triad_animation(animation, train):
    animation_triads = np.zeros((0, 3, RESOLUTION, RESOLUTION, CHANNELS))
    animation_triads = create_triad(animation, animation_triads, 0, len(animation) - 1, train)
    return(animation_triads)

def create_triad(animation, animation_triads, start, end, train):
    if start >= end - 1 or end <= start - 1:
        return animation_triads

    middle = end-((end - start)//2)
    triad = create_single_triad(animation, start, middle, end, train)

    animation_triads = np.concatenate([animation_triads, triad], axis=0)
    animation_triads = create_triad(animation, animation_triads, start, middle, train)
    animation_triads = create_triad(animation, animation_triads, middle, end, train)
    return animation_triads

def create_single_triad(animation, start, middle, end, train):
    triad = np.zeros((1, 3, RESOLUTION, RESOLUTION, CHANNELS))
    triad[0][0] = animation[start]
    if train:
        triad[0][1] = 1
    else:
        triad[0][1] = animation[middle]
    triad[0][2] = animation[end]
    return triad

dataset = np.zeros((1, 10, RESOLUTION, RESOLUTION, 1))
triads = create_triad_dataset(dataset)
```

10.2 Provided Files

File location on CD	Note
00_master_thesis/thesis_hendrik_vosskamp.pdf	Master Thesis as PDF
00_master_thesis/thesis_hendrik_vosskamp.docx	Master Thesis as Word document
01_software/00_web_scraping/	Web scraping and conversion scripts as described in chapter 4.2.1
01_software/01_machine_learning/00_MLP/	MLP architectures as discussed in chapter 5.5
01_software/01_machine_learning/01_RNN/	RNN architectures as discussed in chapter 5.6.1
01_software/01_machine_learning/02_LSTM/	LSTM architectures as discussed in chapter 5.6.2
01_software/01_machine_learning/03_GRU/	GRU architectures as discussed in chapter 5.6.3
01_software/01_machine_learning/04_ConvLSTM/	ConvLSTM architectures as discussed in chapter 0
01_software/01_machine_learning/05_ResNet/	ResNet architectures as discussed in chapter 5.7.2
01_software/01_machine_learning/06_bbCNN/	bbCNN architectures as discussed in chapter 5.7.3
01_software/01_machine_learning/07_akaNet/	akaNet architectures as discussed in chapter 5.7.7
01_software/01_machine_learning/08_ConvVAE/	ConvVAE architectures as discussed in chapter 5.7.8
01_software/02_conda_keras_env.yml	Anaconda environment used for training the presented machine learning models
02_custom_digits/	Custom digits as used in chapter 5.7.3

10.3 Architectures of the subsampling experiments

Layer (type)	Output Shape	Param #
max_pooling3d (MaxPooling3D)	(None, 3, 16, 16, 1)	0
conv3d (Conv3D)	(None, 3, 16, 16, 32)	896
conv3d_1 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_2 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_3 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_4 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_5 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_6 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_7 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_8 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_transpose (Conv3DTranspose)	(None, 3, 32, 32, 32)	27680
Output (Conv3D)	(None, 3, 32, 32, 1)	865

Total params: 250,881
Trainable params: 250,881
Non-trainable params: 0

Figure 89: Entry #2, one max-pooling layer at the 1st position. Validation loss of 0.0118. Own image.

Layer (type)	Output Shape	Param #
conv3d (Conv3D)	(None, 3, 32, 32, 32)	896
max_pooling3d (MaxPooling3D)	(None, 3, 16, 16, 32)	0
conv3d_1 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_2 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_3 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_4 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_5 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_6 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_7 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_8 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_transpose (Conv3DTranspose)	(None, 3, 32, 32, 32)	27680
Output (Conv3D)	(None, 3, 32, 32, 1)	865

Total params: 250,881
Trainable params: 250,881
Non-trainable params: 0

Figure 90: Entry #3, one max-pooling layer at the 2nd position. Validation loss of 0.0062. Own image.

Layer (type)	Output Shape	Param #
conv3d (Conv3D)	(None, 3, 32, 32, 32)	896
conv3d_1 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_2 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_3 (Conv3D)	(None, 3, 32, 32, 32)	27680
max_pooling3d (MaxPooling3D)	(None, 3, 16, 16, 32)	0
conv3d_4 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_5 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_6 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_7 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_8 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_transpose (Conv3DTranspose)	(None, 3, 32, 32, 32)	27680
Output (Conv3D)	(None, 3, 32, 32, 1)	865

Total params: 250,881
Trainable params: 250,881
Non-trainable params: 0

Figure 91: Entry #4, one max-pooling layer at the 5th position. Validation loss of 0.0058.
Own image.

Layer (type)	Output Shape	Param #
conv3d (Conv3D)	(None, 3, 32, 32, 32)	896
conv3d_1 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_2 (Conv3D)	(None, 3, 32, 32, 32)	27680
max_pooling3d (MaxPooling3D)	(None, 3, 16, 16, 32)	0
conv3d_3 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_4 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_5 (Conv3D)	(None, 3, 16, 16, 32)	27680
max_pooling3d_1 (MaxPooling3D)	(None, 3, 8, 8, 32)	0
conv3d_6 (Conv3D)	(None, 3, 8, 8, 32)	27680
conv3d_7 (Conv3D)	(None, 3, 8, 8, 32)	27680
conv3d_transpose (Conv3DTranspose)	(None, 3, 16, 16, 32)	27680
conv3d_transpose_1 (Conv3DTranspose)	(None, 3, 32, 32, 32)	27680
Output (Conv3D)	(None, 3, 32, 32, 1)	865

Total params: 250,881
Trainable params: 250,881
Non-trainable params: 0

Figure 92: Entry #5, two max-pooling layers at the 4th & 8th position. Validation loss of 0.0069.
Own image.

Layer (type)	Output Shape	Param #
conv3d_22 (Conv3D)	(None, 3, 32, 32, 32)	896
conv3d_23 (Conv3D)	(None, 3, 32, 32, 32)	27680
3rd_Conv3D_with_strides (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_24 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_25 (Conv3D)	(None, 3, 16, 16, 32)	27680
6th_Conv3D_with_strides (Conv3D)	(None, 3, 8, 8, 32)	27680
conv3d_26 (Conv3D)	(None, 3, 8, 8, 32)	27680
conv3d_27 (Conv3D)	(None, 3, 8, 8, 32)	27680
conv3d_transpose_6 (Conv3DTranspose)	(None, 3, 16, 16, 32)	27680
conv3d_transpose_7 (Conv3DTranspose)	(None, 3, 32, 32, 32)	27680
Output (Conv3D)	(None, 3, 32, 32, 1)	865

Total params: 250,881
Trainable params: 250,881
Non-trainable params: 0

Figure 93: Entry #6, two Conv3D layers with strides at the 3rd & 6th position. Validation loss of 0.0063.
Own image.

Layer (type)	Output Shape	Param #
conv3d (Conv3D)	(None, 3, 32, 32, 32)	896
conv3d_1 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_2 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_3 (Conv3D)	(None, 3, 32, 32, 32)	27680
5th_Conv3D_with_strides (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_4 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_5 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_6 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_7 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_transpose (Conv3DTranspose)	(None, 3, 32, 32, 32)	27680
Output (Conv3D)	(None, 3, 32, 32, 1)	865

Total params: 250,881
Trainable params: 250,881
Non-trainable params: 0

Figure 94: Entry #7, one Conv3D layer with strides at the 5th position. Validation loss of 0.0060.
Own image.

Layer (type)	Output Shape	Param #
1st_Conv3D_with_strides (Conv3D)	(None, 3, 16, 16, 32)	896
conv3d (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_1 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_2 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_3 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_4 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_5 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_6 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_7 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_transpose (Conv3DTranspose)	(None, 3, 32, 32, 32)	27680
Output (Conv3D)	(None, 3, 32, 32, 1)	865

Total params: 250,881
Trainable params: 250,881
Non-trainable params: 0

Figure 95: Entry #8, one Conv3D layer with strides at the 1st position. Validation loss of 0.0058.
Own image.

Layer (type)	Output Shape	Param #
conv3d_8 (Conv3D)	(None, 3, 32, 32, 32)	896
2nd_Conv3D_with_strides (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_9 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_10 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_11 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_12 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_13 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_14 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_15 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_transpose_1 (Conv3DTranspose)	(None, 3, 32, 32, 32)	27680
Output (Conv3D)	(None, 3, 32, 32, 1)	865

Total params: 250,881
Trainable params: 250,881
Non-trainable params: 0

Figure 96: Entry #9, one Conv3D layer with strides at the 2nd position. Validation loss of 0.0056.
Own image.

Layer (type)	Output Shape	Param #
conv3d (Conv3D)	(None, 3, 32, 32, 32)	896
2nd_Conv3D_with_strides (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_1 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_2 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_3 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_4 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_5 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_6 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_7 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_8 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_transpose (Conv3DTranspose)	(None, 3, 32, 32, 32)	27680
Output (Conv3D)	(None, 3, 32, 32, 1)	865

Total params: 278,561
Trainable params: 278,561
Non-trainable params: 0

Figure 97: Entry #10, one additional Conv3D layer with strides at the 2nd position. Validation loss of 0.0057.
Own image.

Layer (type)	Output Shape	Param #
conv3d (Conv3D)	(None, 3, 32, 32, 32)	896
conv3d_1 (Conv3D)	(None, 3, 32, 32, 32)	27680
conv3d_2 (Conv3D)	(None, 3, 32, 32, 32)	27680
4th_Conv3D_with_strides (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_3 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_4 (Conv3D)	(None, 3, 16, 16, 32)	27680
conv3d_5 (Conv3D)	(None, 3, 16, 16, 32)	27680
8th_Conv3D_with_strides (Conv3D)	(None, 3, 8, 8, 32)	27680
conv3d_6 (Conv3D)	(None, 3, 8, 8, 32)	27680
conv3d_7 (Conv3D)	(None, 3, 8, 8, 32)	27680
conv3d_transpose (Conv3DTranspose)	(None, 3, 16, 16, 32)	27680
conv3d_transpose_1 (Conv3DTranspose)	(None, 3, 32, 32, 32)	27680
Output (Conv3D)	(None, 3, 32, 32, 1)	865

Total params: 306,241
Trainable params: 306,241
Non-trainable params: 0

Figure 98: Entry #11, two Conv3D layers with strides at the 4th and 8th position. Validation loss of 0.0063.
Own image.

11 Selbstständigkeitserklärung

FernUniversität in Hagen

Selbstständigkeitserklärung

Name: Hendrik Voßkamp

Matrikel-Nr.: 3042472

Fach: Praktische Informatik

Modul: Masterarbeit

Thema: AI driven animations of 2D rasterized graphics

Ich erkläre, dass ich die Abschlussarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Arbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Arbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

Datum: _____ **Unterschrift:** _____