RtoV - Deep Learning with Generated Data for Raster Vectorization

Matura Work by Lars Hoesli
Supervisor: Beat Temperli

December 2023

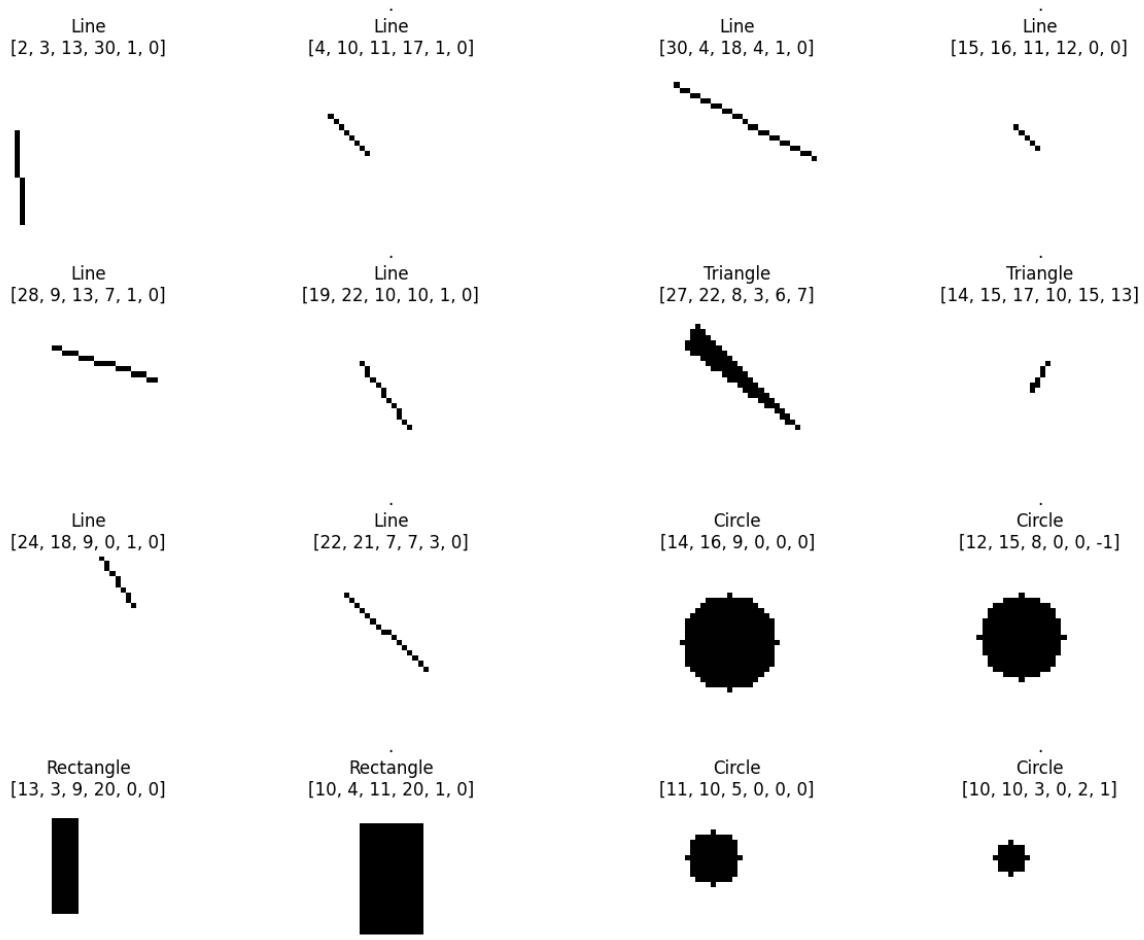


Figure 1: *An example output of the model used in the demonstration. Two images form a pair, the first being the input image and the second the model output, converted into a raster representation (source [I3])*

Abstract

The conversion from raster images to vector representations remains a challenging task to this day and is actively researched. Due to the inherent difficulties that algorithmic approaches face, such as finding shapes in noisy or blurred images under varying lighting conditions, more research is being done using machine learning, where the most apparent problem is the acquisition of appropriate training data. Therefore, automatically generating pairs of raster images and their corresponding vector representations can be advantageous. This paper explores this particular approach: Training a machine learning model on generated data pairs. A demonstration is conducted in a limited scope, where a deep learning model is trained on on-the-fly generated data in order to perform the conversion from raster images containing clearly visible shapes into vector representations. The model used in the demonstration is a simple multi-task learning model with a convolutional neural network as a front end and various back ends responsible for classifying the shape or determining the features of a particular shape.

The model, although trained on a limited number of images, was able to determine the shape with high accuracy and learned to approximate the data relevant for specifying the shapes *Line*, *Rectangle* and *Circle*. The vertices of the shape *Triangle* have proven to be more challenging to estimate and the model presented in the demonstration was not able to provide an accurate representation for this shape.

The code for the demonstration can be found online at github.com/lrshsl/RtoV

Contents

Chapter 1

Introduction

Images have become an important part of everyday life, most of which are stored digitally. This make the search for effective storage of images an essential, well-researched aspect of computer science. Many different formats and compression methods have emerged. The image formats used today can roughly be categorized into two main categories - vector and raster formats.

1.1 Raster and Vector Graphics

Raster and vector graphics are two fundamentally different approaches to representing the content of an image. Each has its advantages in representing certain types of images, but may be less appropriate in other situations. While raster images store the color values of small parts of a given picture - often called pixels - to approximate what the image looks like, vector formats store a specification for what shapes can be seen, similar to how humans receive images.

Both methods have pros and cons, and are more appropriate for certain situations than others. But vector formats have numerous advantages for storing images that are easily describable with shapes, especially those consisting of uniformly colored areas or easily describable gradients. In these cases, vector graphics can be a more precise and efficient representation of the image than raster formats. Vector formats can achieve an infinite resolution, since they do not work with grids of pixels, while being less storage intensive at the same time. Raster formats, in turn, are better suited for images without easily distinguishable shapes, such as portraits or landcape images.

Since the generation of raster images has become popular and the programs performing it more mature, programs for generating vector images are not yet so far.

1.1.1 Problem Description

While format conversions among raster or vector formats and from vector to raster graphics can be done with a multitude of programs, the conversion from a raster image to a vector representation proves to bee more challenging. This is especially because the shapes and their features seen in the input image have to be recognized, which is not an easily solvable problem. Many factors can make it more difficult, such as contrasts in different strengths, noise, and gradients that make it impossible to work with fixed thresholds to detect the contours of shapes. With different manipulations and strategies, such as *hough transform*, *skeletonization* or various edge detection techniques, it is possible to vectorize raster images algorithmically. However, by the time of writing, these approaches are still very limited.

With deep learning those problems can partially be addressed. The model itself can learn to recognize the shapes, in a similar manner as humans do. Deep learning models have their own difficulties. For supervised learning, accumulating sufficient training data is a significant challenge and often a limiting factor.

Therefore a way to convert raster images into a vector format is beneficial and there is no universal solution to it. Deep learning is a powerful tool, especially if the data to train a neural network could be generated indefinitely. This is why this thesis has been done on the topic of raster-to-vector conversion with deep learning, and focuses on the usage of generated training data.

1.2 Fundamentals

1.2.1 Activation Functions

Activation functions are essential components in neural networks that introduce non-linearities. These functions decide whether a neuron should be activated or not based on the weighted sum of its inputs and possibly a bias value if provided.

One widely adopted activation function is the Rectified Linear Unit (ReLU) function. ReLU, short for Rectified Linear Unit, is a simple yet efficient activation function. Mathematically, it is defined as:

$$ReLU(x) = \max(0, x)$$

As graphically plotted in 1.1.

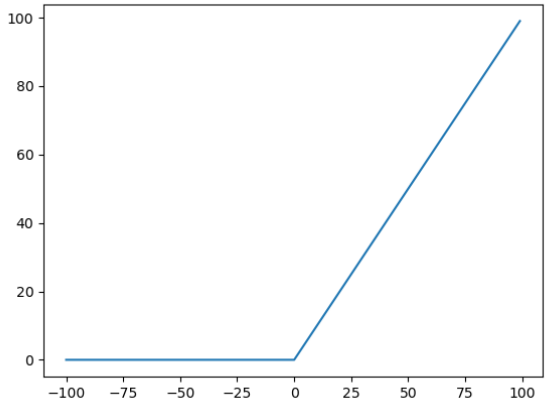


Figure 1.1: *ReLU function plotted with* $x \in [-100; 100)$ *(source [I2])*

In other words, if the input to the ReLU function is negative, it returns zero; otherwise the input value itself is returned. The simplicity of ReLU makes it computationally efficient while still introducing non-linearity. ReLU has become a popular choice in many neural network architectures.

Other popular activation functions include *Softmax*, *Sigmoid*, which squash the input into the range $(0, 1)$ and *Tanh* with a range of $[-1, 1]$.

1.2.2 Loss Functions

Loss functions provide a way to measure the quality of the predictions made by a neural network. Several loss functions exist, each comparing the output of the network with the label. The output usually consists of the values of the neurons in the last layer and is provided as a vector, array or matrix of numbers. The label either already is a vector or similar of the ideal distribution, or is being translated into it.

Loss functions play a crucial role in evaluating the accuracy of the predictions generated by a neural network. They compare the output of the network to the actual values (labels). The output, i. e. the values of the last layer of the network typically takes the form of a vector, array, or matrix. The label may already exist as an ideal distribution in the same form or be translated into one.

A common loss function is the *mean squared error* (MSE), which measures the difference between the predicted values and the actual values and squares it, so that the larger errors get more weight while removing the sign at the same time. One widely used loss function is the *mean squared error* (MSE), which measures the squared difference between the predicted and actual values. The squaring amplifies larger errors while eliminating the sign at the same time:

$$MSE(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

Where y is the true value, $\hat{y}$ is the predicted value, and n is the number of samples on which the function is applied.

Another prevalent loss function is *cross-entropy* or *categorical cross-entropy*. They measure the divergence between the predicted probability distribution and the true distribution of labels with the help of the log function. For categorical cross-entropy:

$$CE(y, \hat{y}) = -\sum_{i=1}^{n} y_i \cdot \log(\hat{y}_i)$$

Minimizing this output of a loss function encourages the predicted distribution to converge towards the true distribution and thus improves the accuracy of the neural network. Some loss functions are better suited for certain tasks than others. The MSE loss is especially suitable for regression tasks, while the CE loss is especially suitable for classification tasks.

1.3 Architectures of Neural Networks

Deep neural networks can be structured in different ways, leading to different kinds of traits that are beneficial in certain situations over others. For raster-to-vector conversion the following architectures are especially interesting.

1.3.1 Convolutional Neural Networks

A convolutional neural network (CNN) is a type of neural network that uses a convolutional layer to extract features from an input image.

Reducing the information of individual pixels to a sequence of distinct features proves beneficial, as fully connected layers can subsequently learn to establish connections based on these features. In order to extract the relevant features, it uses different *kernels*, which are moved

through the images using a certain *stride* value and are applied to the pixel values. A features map is produced. The *kernel* and *stride* values can be adapted to reduce the information for the following layers directly with the convolution layer. Alternatively, after each convolution, a pooling function can be applied to the feature maps, which reduces the number of features that need to be processed. The usual architecture consists of repeating convolution and pooling layers, which extract the relevant features, and finally fully connected neuronal layers that can logically process the information.

A graphical scheme is shown in figure 1.2.

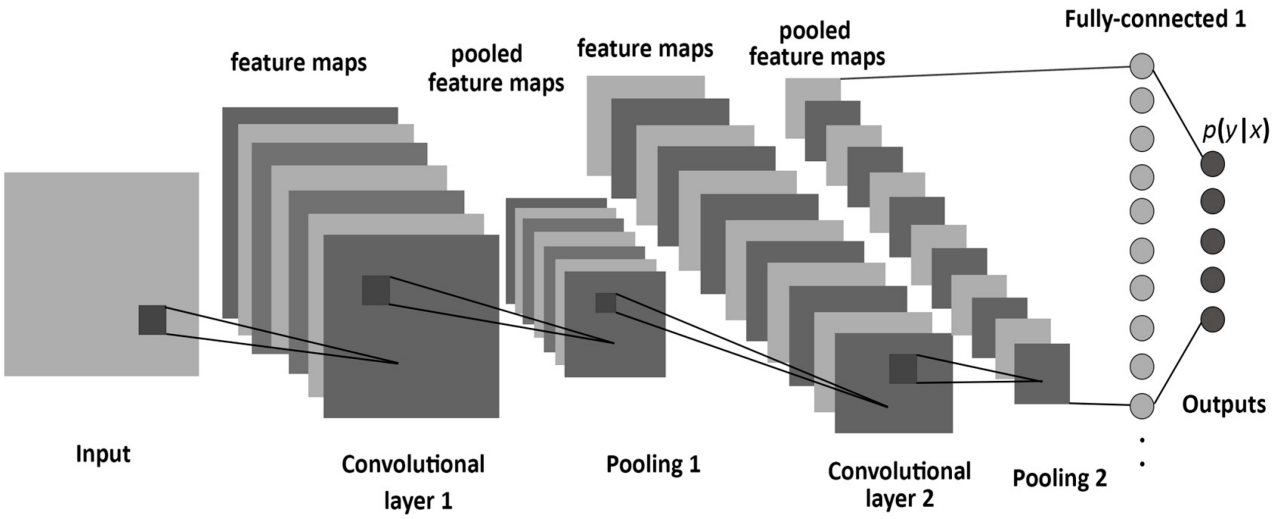


Figure 1.2: *Typical convolutional neural network architecture (source [I1])*

Convolutional neural networks posses the ability to automatically identify relevant features in images. As such, they are invaluable for computer vision, object detection, as well as for classification tasks. In the demonstration for this work, the CNN architecture is used to classify the image and extract the relevant features.

1.3.2 Recurrent Neural Networks

Recurrent neural networks (RNNs) are well suited to handle sequential data by remembering information about previous inputs. This kind of memory mechanism is achieved by having loops within the neural network (1.3), allowing recurrent neural networks to retain information and consider past in- and outputs when processing current ones. This makes them ideal for tasks like time series prediction, natural language processing, speech recognition, and more.

However, traditional RNNs can face issues like the vanishing or exploding gradient problem, hindering their ability to learn long-range dependencies effectively. To address those issues, improvements like Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) have been developed. These architectures use gating mechanisms that control the flow of information, enabling better long-term memory and gradient flow.

RNNs and their enhanced versions like LSTM and GRU have proven valuable in various applications that involve sequential data, allowing machines to comprehend and generate sequences more effectively than standard feedforward networks. In the case of raster vectorization, this architecture could allow a neural network to remember which shape has previously been processed.

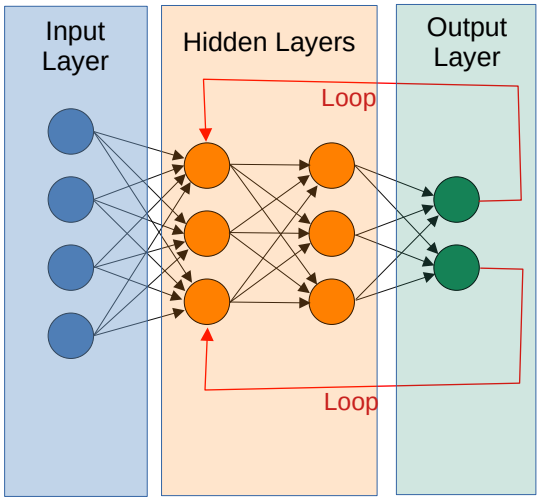


Figure 1.3: *Example Recurrent neural network architecture (source [I7])*

1.3.3 Reinforcement Learning

In Reinforcement Learning (RL), a neural network (agent) learns to make decisions by interacting with a given environment. The agent takes actions in the environment, and based on those actions, it receives feedback in the form of rewards or punishments. The goal of the agent is to learn how to maximize the reward.

The suggested use of RL for vectorization in this paper is inspired by the work "Marvel - Raster Manga Vectorization via Primitive-wise Deep Reinforcement Learning" ([T5]).

The agent learns by trial and error, exploring different actions and observing the consequences in terms of rewards. The goal is to discover an optimal policy that leads to the maximum cumulative reward over time. Reinforcement learning has been successfully applied to various domains, including game playing (e.g., AlphaGo), robotics, finance, and more. They have already proven valuable in raster-to-vector conversion [T5].

I. **Environment**: In the case of the raster-to-vector conversion, the environment is the raster image that is being converted.

II. **State**: The state is a copy of the original image, which is modified by the agent each time a new shape is predicted.

III. **Action**: The deletion of the shape that was predicted follows as a action that modifies the state.

IV. **Reward**: The reward is calculated by comparing the original image with the state. To do this comparison, both images would have to be converted into a raster presentation, so that the pixel values can be compared.

1.3.4 Residual Neural Networks

Residual Neural Networks (ResNets) facilitate the development of large neural networks. It provides a way for some information to skip one or more layers by adding the input of the layers to their output according to the following formula:

$$Output = Activation\,(Weight \times Input + Shortcut(Input))$$

The only derivation from traditional neural networks is the $Shortcut(Input)$ term, which can be as simple as a one-to-one mapping of the input, or a projection in case the in- and output dimensions differ. Those shortcut connections help in addressing the vanishing gradient problem, which can occur in very deep networks during backpropagation. ResNets facilitate the training of deeper networks by allowing the gradient to pass directly through the skip connections, even in the case where some layers have nearly zero gradients.

1.3.5 Multi-Task Learning

Multi-task learning is an expression used to classify neural networks that solve several related tasks simultaneously in the same model. This allows for some layers or blocks of layers to be reused, and can lead to faster performance than training multiple neural networks. It also allows for information to be shared, such as the output of one block which might influence which other blocks are used for further processing of the information.

Typically for multi-task learning are large neural networks with one or more common layers (here referred to as *front end*), which extract the relevant data for the subsequent layers, followed by a branching into several independent blocks of layers, specific for separate tasks.

Multi-task learning is advantageous for raster-to-vector conversion because typically, much time is spent in the front end with feature extraction. The information extracted in this stage can be used by most if not all subtasks, such as image classification, color extraction and shape approximation. Therefore, it is efficient if this information can be reused for the entire prediction.

Chapter 2

Goal and Hypothesis

To train a machine learning model to vectorization of raster images, a substantial amount of data is needed. Ideal data would consist of large collections of pairs of raster and vector representations of the same image, which are typically difficult to obtain. Since the conversion of vector images to raster formats is far less difficult, it could be a valid approach to generate random vector images and then convert them into a raster format, to get those raster-vector pairs on which the model can be trained.

This work examines the effectiveness this approach by addressing the following questions:

I. Is it feasible to approach raster-to-vector graphics conversion with a neural network that is trained on randomly generated data?

II. What strategies or measures have to be taken in order to make this feasible?

III. What performance can be achieved, and what are the limitations and pitfalls of this approach?

Chapter 3

Method

3.1 Overview

The training and evaluation data for the model consists of thirty-two on thirty-two pixel images with three color channels, with exactly one shape on each image. The shape is being drawn on all color channels to the pixels that are situated within the shape, resulting in a black appearance on a white canvas when plotted. The generated shapes, as well as their sizes and positions, are randomized and vary with each function call.

The training occurs in several steps:

I. Generation of the training data

II. Training of the model

III. Evaluation of the model

3.1.1 Shape Generation

A function has been written for each shape, returning a numpy array that describes the shape in a numerical representation. This allows it to be directly compared to the output of the model, with no additional conversion step, while representing the relevant features of the image with a minimal number of bytes. Since the number of data points differs from each shape, the array is padded with zeros to match the maximum number of data points. This padding is not being used for determining the performance of the model.

Here is a breakdown of the different shapes and their data representation:

- Line: A line is defined through two points, width and color: [x1, y1, x2, y2, width, *color*]. Zeros are added for padding. Width and color are not used by the model in the demonstration.
- Circle: Center, radius and color: [x, y, r, *color*].
- Rectangle: One corner, width, height and color: [x1, y1, w, h, *color*].
- Triangle: Three corner points and color: [x1, y1, x2, y2, x3, y3, *color*].

When fully implemented, the color could take up four bytes, for red, green, blue and alpha values. In the unmodified demonstration, the color is not being used since the shape that is being drawn on the raster image is always black.

The function in the current configuration returns a one-dimensional numpy array with a length of six, which corresponds to the number of data points used for a triangle (without color).

The corner points, width and height values generated are constrained to ensure that the shape always remains within the image. However, the possibility that two different shapes can result in the same pixels being colored is not considered. This means that, for instance, a triangle with its corners forming a line may lead to the same input image as a line with the same corner points may produce. In such a case, the model could not possibly know the correct shape and might be punished.

The representation that is obtained after the previous step, then is written to a numpy array with the shape (100, 100, 3) or in graphical terms a white canvas. To convert from this specific representation to a raster image, a function has been developed (the function draw_on_image in data/draw_shape_on_image.py) which accomplishes this task using the *OpenCV2* library.

3.1.2 Training

3.1.3 Evaluation

For the evaluation of the model, its output, which is a numpy array containing a batch of shape predictions and one which contains a batch of the shape specifications is being compared to the ground truth. The shape specifications are numpy arrays with data in the same representation as the one the raster image is generated from. This is done using the following loss functions: *Crossentropy loss* (CE) for the shape prediction and *Mean Squared Error* (MSE) for the prediction of the data specific to the shape.

3.2 Language Choice

Deep learning is a very resource intensive task. Due to time and resource constraints, the language choice was primarily based on the expected development time and runtime performance of the language and its frameworks. However, since the computational intensive parts are done within the deep learning library, the availability of performant and established machine learning libraries or frameworks was a high priority. On this basis, the table 3.1 has been produced, which assigns points to each of the relevant factors. Furthermore, an in-depth analysis of the different parts have been conducted, which provides insight of how the points were selected.

*The point *Other ecosystem* is rating the availability of libraries that are important for the given task. Namely, libraries for working with raster and vector images and plotting libraries for inspecting the progress of the neural network.

Python

Ecosystem Many well-established and heavily optimized machine learning libraries such as Tensorflow, Keras, Pytorch have been developed for Python. Because of its active com-

Language	Performance	Machine learning libraries	Other ecosystem*	Representative ML Libraries	Rating
Python	1	5	5	Pytorch, TF	***40***
Javascript	2	4	5	BrainJS, Ml5	*37*
C++	5	3	4	TF, mlpack, CNTK	*32*
C	5	1	2	Darknet, Libtorch	*16*
Rust	5	4	3	Burn, Tangram	*34*
Go	4	3	4	Gorgonia	*31*
Relevance	**1**	**5**	**3**	-	

Table 3.1: *Language evaluation table*

munity and many libraries, such as Matplotlib, Pandas or Pillow, it has the reputation of being the best fit for deep learning and data science.

Performance The language performs poorly on benchmark tests [T2], but Python libraries are written mostly in C and heavily optimized. Thus, the performance-critical parts are usually fast enough. Performance bottlenecks can also be gradually optimized by switching from Python to Cyphon or directly to C, which are compiled and can take advantage of type information and are therefore much faster.

Javascript

Ecosystem Many high-level machine learning libraries such as Brain.js, Ml5.js exist, but few provide the fine-grained control necessary for deep learning research.

Performance The language itself usually performs better than Python, but due to its dynamically typed nature, it still is remarkably slower than most statically typed languages.

Other remarks Javascript runs in browsers, which would facilitate deployment to the web.

C++

Ecosystem Many deep learning libraries such as *mlpack*, *caffe* and *Tensorflow* exist. The fast but high-level nature would allow implementing the machine algorithms manually, but since development speed is a priority, this might not be a good choice.

Performance C++ is a mature, performant and relatively high-level language. However, the development speed is not optimal. The language focuses on performance over ease of use and memory-safety. Much control, but also responsibility is given to the developer.

C

Ecosystem Despite many deep learning frameworks being written in C, they are often designed to be operated from higher-level languages. Machine learning libraries that focus on

providing an interface for C like *Darknet* are often very specialized and prioritize speed over ease of use.

Performance The language itself belongs to the best performant languages. But the lack of high-level features can slow down development speed. It is also not memory-safe, which can lead to bugs and vulnerabilities.

Rust

Ecosystem Rust is a promising but relatively new language, and so is its deep learning ecosystem. Comprehensive, performant deep learning frameworks such as Burn do exist, but are not yet in a stable state and have not yet been able to build up a large ecosystem around it. Calling C functions and thus using C libraries such as *libtorch* is possible, but not an optimal solution.

Performance Rust is very performant while still offering high-level features through zero-cost abstractions. As opposed to C++, Rust is also memory-safe due to its built-in borrow checker.

Go

Ecosystem Go is a promising language, but its ecosystem is in a similar state as the one of Rust. Libraries, like Gorgonia, might be as powerful as *Keras* soon, but have not yet been widely adopted and do not have such a large community behind it as *Tensorflow* or *Pytorch*.

Performance Go is faster than both Python and Javascript, and allows to easily add concurrency, which can increase the performance even more.

3.2.1 Conclusion

Python is known to be the most popular or even the best choice for machine learning, due to its various well established libraries and ecosystem. Thus, it has not come as a surprise that the evaluation in this work has come to the same conclusion. Python is an easy language with a large ecosystem and very active community. An advantage that is often not considered is the ability of Cython to call back and forth into C as well as Python code, thus making an application written in Python or Cython gradually optimizable.

However, the performed evaluation is based on experience and research of the author, as well as common knowledge and is thus not subjective nor comprehensive. It is meant to give an overview of the options and it was tried to make as objective as possible.

3.3 Framework Choice

3.3.1 Conclusion

After experimenting with both frameworks, *Pytorch* has been chosen for the demonstration. It is a popular and widely used framework and its goals align better with the goals of this work: Simplicity and fast development speed in order to experiment with different architectures.

Criteria	Tensorflow	Pytorch	Project Focus
Written in	C++	C	-
Main focus	Comprehensiveness	Simplicity, fast development in a Pythonic way	*Fast development*
Ecosystem	Huge ecosystem with many tools	Comparatively few tools available	*Many good tools*
Primary application	Application Production	AI Research	*AI Research*
Performance	Highly optimized for large-scale training	Focuses on flexible training and experimentation	*Flexibility, fast iterations and changes*
Community	One of the most often used libraries	Often used library	*More often used*

Table 3.2: *Framework evaluation table*

3.4 Data Generation

The data that the model uses for training consists of raster images containing one shape in each. The images are internally represented as numpy arrays, where each entry represents the color values of a pixel. The background is white (i. e. RGB values set to (1, 1, 1)), and the pixels that fall within the shape are set to (0, 0, 0), thus appearing black, so that the contrast between shape and background is maximized.

All data generation is implemented in rtov/data/ and its subdirectories. A class, LazyDataset, which inherits from the torch. utils .data.Dataset, provides an interface, which is a Dataloader (torch. utils .data.Dataloader) object can use later to get the next image. Therefore, the method __getitem__(self, i: int) is provided, which loads a numpy array, draws a random shape on it and transforms it into a vector.

3.5 Model Architecture

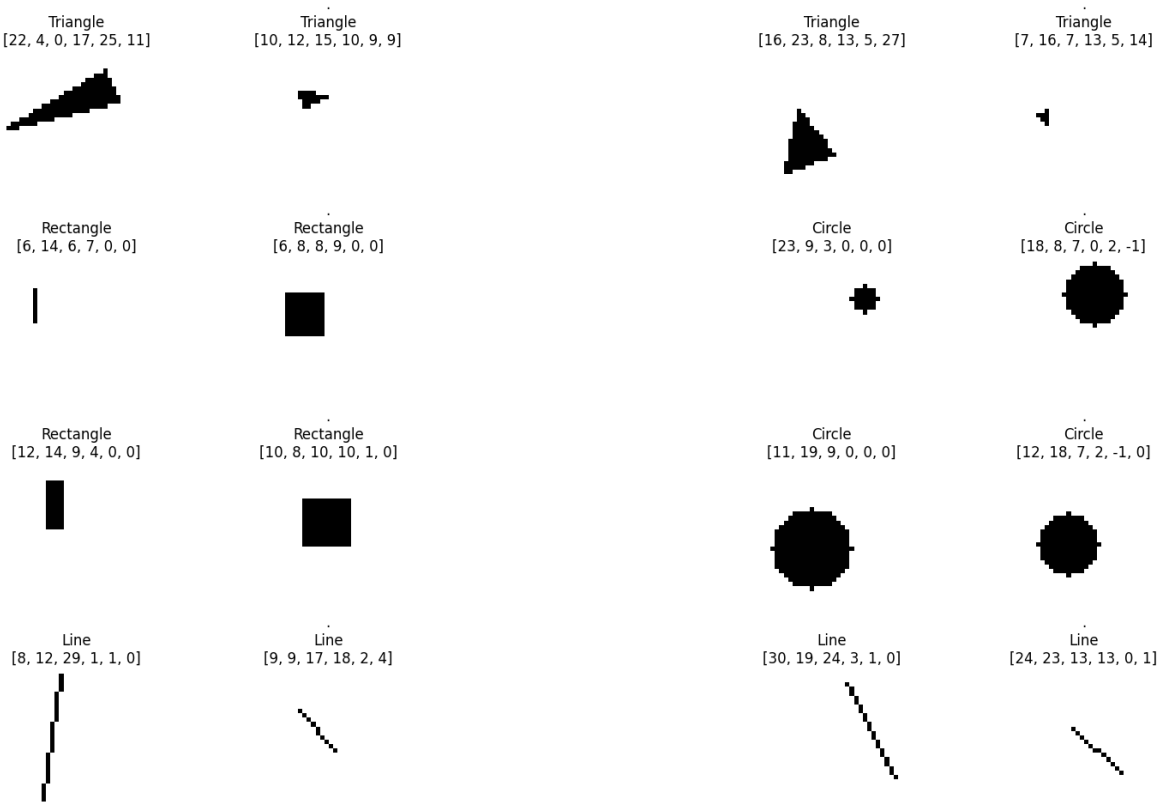
The model in the demonstration is a convolutional neural network with several back end. A back end is a collection of fully connected layers, which can be trained on the output of the front end (i. e. the convolutional and pooling layers). At first, the input image is fed through the convolutional front end: Two blocks with each a convolution layer with ReLU activation followed by a Max Pooling layer. Subsequently, the result is passed through one of the back end that is responsible for the classification of the shape. Depending on the shape that has been predicted in the previous step, the same output of the convolutional layers is passed through another block of fully connected layers which are responsible for the respective shape. Through this mechanism, a conditional form of multi-task learning is achieved.

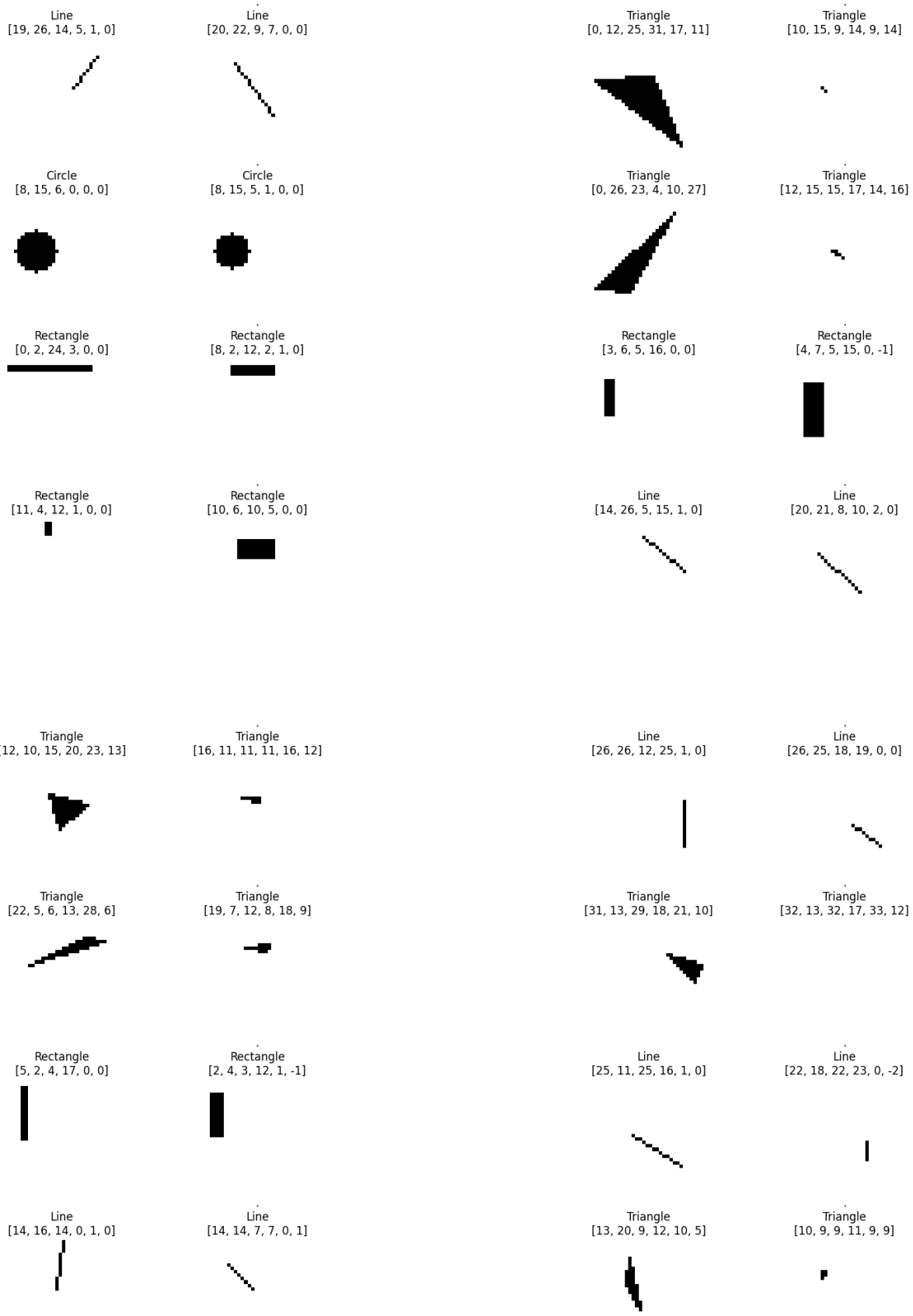
Chapter 4

Results

The model in the demonstration has been able to approximate the shapes it was trained on. The accuracy would need to be improved in order to provide usable results. The given model had problems with approximating triangles; Rather than estimate the positions of the vertices, the model has learned to place those points grouped around the center of the triangle.

The following illustrations are sources [I4], [I5] and [I6] . They are example outputs of the model evaluation (obtained through the *test* command in the demonstration). Two images each form a pair, the right one of which is the original raster representation, the left is the output of the model when evaluated on the right image. The model output has been converted into a raster format. The exact numerical data is shown above the respective image.




Line
[19, 26, 14, 5, 1, 0]
Line
[20, 22, 9, 7, 0, 0]
Triangle
[0, 12, 25, 31, 17, 11]
Triangle
[10, 15, 9, 14, 9, 14]
Circle
[8, 15, 6, 0, 0, 0]
Circle
[8, 15, 5, 1, 0, 0]
Triangle
[0, 26, 23, 4, 10, 27]
Triangle
[12, 15, 15, 17, 14, 16]
Rectangle
[0, 2, 24, 3, 0, 0]
Rectangle
[8, 2, 12, 2, 1, 0]
Rectangle
[3, 6, 5, 16, 0, 0]
Rectangle
[4, 7, 5, 15, 0, -1]
Rectangle
[11, 4, 12, 1, 0, 0]
Rectangle
[10, 6, 10, 5, 0, 0]
Line
[14, 26, 5, 15, 1, 0]
Line
[20, 21, 8, 10, 2, 0]
Triangle
[12, 10, 15, 20, 23, 13]
Triangle
[16, 11, 11, 11, 16, 12]
Line
[26, 26, 12, 25, 1, 0]
Line
[26, 25, 18, 19, 0, 0]
Triangle
[22, 5, 6, 13, 28, 6]
Triangle
[19, 7, 12, 8, 18, 9]
Triangle
[31, 13, 29, 18, 21, 10]
Triangle
[32, 13, 32, 17, 33, 12]
Rectangle
[5, 2, 4, 17, 0, 0]
Rectangle
[2, 4, 3, 12, 1, -1]
Line
[25, 11, 25, 16, 1, 0]
Line
[22, 18, 22, 23, 0, -2]
Line
[14, 16, 14, 0, 1, 0]
Line
[14, 14, 7, 7, 0, 1]
Triangle
[13, 20, 9, 12, 10, 5]
Triangle
[10, 9, 9, 11, 9, 9]

Chapter 5

Discussion

5.1 Opportunities and Limitations

The demonstration has been done in a limited scope and does not provide a basis for making general statements about the usage of generated data. That being said,

The model in the demonstration has shown that a neural network that has been trained on randomly generated raster-vector representation pairs can produce results that resemble the original raster image. In that regard, the demonstration has confirmed the initial hypothesis.

No limiting factors habe been observed that would hinder the scaling of this approach, and there are no apparent reasons for why a neural network with sufficient complexity and training should not be able to perform vectorization of simple raster images.

However, it should to be considered that the data that is used in the demonstration is very limited and has not yet reached a point where it could be used in real world applications. The training and evaluation data consist images containing a single shape, and there are a small number of shapes available. Furthermore, only only data related to the position and size of these shapes. The model outputs a numerical representation of the image, which first would have to be converted into an actual vector format such as SVG.

The demonstration specifically lacks the ability to determine the color of a shape, and cannot be used to convert images containing more than one shape and subsequently never learned how to recognize relations between the shapes in an image.

Those are limitations of demonstration and not necessarily of the approach itself. The model used in the demonstration cannot answer the question of whether those obstacles can be overcome or not.

Those observations could be made:

I. Even a model trained on a CPU can quickly learn to extract shapes from raster images using the generated data

II. To scale and improve the model, many optimizations could still be taken in the sections model architecture, training data and training parameter optimization

III. No obstacles have been observed when increasing the complexity of the training data. There is no apparent reason why it should not be possible to make a model learn more complex data.

IV. A more sophisticated model may be needed for further experiments

5.2 Proposed Model Architecture

There are various strategies and architectures for converting raster images to vector images using deep learning. Based on the experience gained during the progress of this work, an architecture is proposed for projects extending beyond this thesis. A comprehensive structure is given that incorporates elements from various deep learning strategies, combining Recurrent Convolutional networks with reinforcement learning while keeping the multi-task learning nature used in the demonstration.

I. A random array of numbers is generated, which describe the image and its shapes in a vector-like format.

II. The input is a raster image represented as an array.

III. The first layers are convolution and pooling layers. They are applied directly to the input image to extract the relevant features.

IV. The output of the convolutional layers is directly passed to one or more small, independent fully connected layers. These layers are used to classify the shape as well as the color of the shape and background. A separate CNN might be useful for those layers, but this has to be investigated further. Here, the model could incorporate features from Residual Neural Networks, such that the output of the first CNN layers is directly passed to following layers, without going through the entire CNN block.

V. Afterwards, the output of the CNN layers is passed to another set of fully connected layers, which are specific for the shape that has already been predicted. These layers are used to extract the shape-specific data, such as position, rotation, corner points or size.

VI. The output is then converted into a raster format, which then is compared to the input image

VII. For the loss function two fundamentally different approaches are considered:

- Comparing the output of the model directly with the label. This approach is used in the demonstration and is computationally very efficient.
- Converting the output, together with the previous predictions, into a raster image, and comparing it to the input image. This could lead to more precise results and provides more meaningful feedback to the model.

VIII. Recurrent capabilities can be incorporated:

- Long term memory, in order to make the model remember which shapes have already been predicted. This could help the model to learn relations between the shapes in a given image. However, using reinforcement learning would facilitate this further, since this information can be stored in the *state*.
- Short term memory can be very advantageous in the shape-specific layers. This could resolve the difficulties the model had with approximating the vertices of triangles in the demonstration.

5.3 Conclusion

The demonstration and research have shown that deep learning with generated vector-raster image pairs can be a useful technique for raster vectorization, at least in the first stage of training a neural network. The generation of the random images helps to obtain valuable labeled image pairs. However, the demonstration has shown that approximating a raster image in a vector format remains a challenging task even with an infinite amount of training data available. The model in the demonstration has been able to learn how to classify the four shapes, and could extract the relevant information to roughly approximate most of them; triangles have proven more challenging.

Furthermore, an architecture has been proposed that might be able to leverage the advantages of various deep learning architectures and techniques, which could address those challenges. It might be a starting point for future works, and successfully implementing such a neural network and training process might as well result in an efficient way to vectorize raster images in practical applications.

Chapter 6

Sources and References

Text Sources and References

[T1] Maria Dziuba et al. *Image Vectorization: a Review*. June 10, 2023. DOI: 10.48550/arXiv.2306.06441. arXiv: 2306.06441[cs]. URL: http://arxiv.org/abs/2306.06441 (visited on 09/28/2023).

[T2] GoodManWEN. *Programming-Language-Benchmarks-Visualization*. original-date: 2021-05-14T14:47:47Z. Nov. 18, 2023. URL: https://github.com/GoodManWEN/Programming-Language-Benchmarks-Visualization (visited on 11/20/2023).

[T3] Phillip Isola et al. *Image-to-Image Translation with Conditional Adversarial Networks*. Nov. 26, 2018. arXiv: 1611.07004[cs]. URL: http://arxiv.org/abs/1611.07004 (visited on 11/03/2023).

[T4] Christian Ledig et al. *Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network*. May 25, 2017. arXiv: 1609.04802[cs,stat]. URL: http://arxiv.org/abs/1609.04802 (visited on 11/03/2023).

[T5] Hao Su et al. *MARVEL: Raster Manga Vectorization via Primitive-wise Deep Reinforcement Learning*. July 18, 2023. arXiv: 2110.04830[cs]. URL: http://arxiv.org/abs/2110.04830 (visited on 09/06/2023).

Image Resources

[I1] *ImageCNNArchitecture*. URL: `https://miro.medium.com/v2/resize:fit:4000/0*-1Pad7loK_dFOUvS.png` (visited on 11/21/2023).

[I2] Author L. H. *Graph for ReLU Function*. Plotted with Python and Matplotlib by the Author. 2023.

[I3] Author L. H. *Graphical Results Demonstration 0 (condensed)*. Own creation, output of model in demonstration, plotted with Matplotlib. 2023.

[I4] Author L. H. *Graphical Results Demonstration 1*. Own creation, output of model in demonstration, plotted with Matplotlib. 2023.

[I5] Author L. H. *Graphical Results Demonstration 2*. Own creation, output of model in demonstration, plotted with Matplotlib. 2023.

[I6] Author L. H. *Graphical Results Demonstration 3*. Own creation, output of model in demonstration, plotted with Matplotlib. 2023.

[I7] Author L. H. *Image RNN Architecture*. Own creation. 2023.

AI Tools

[A1] Exafunction. *Codeium: A Free AI-Powered Toolkit*. Used for code. 2023. URL: `https://codeium.com/`.

[A2] Wolfram—Alpha LLC. *Wolfram Alpha: A Free AI-Powered Toolkit*. Used for plotting graphs. 2023. URL: `https://wolframalpha.com/`.

[A3] OpenAI. *ChatGPT: A Chatbot for Large Language Models*. Used for code and formulations. 2023. URL: `https://chat.openai.com/`.

Hiermit bestätige ich, Lars Hösli, meine Maturaarbeit selbstständig verfasst und alle verwendeten Quellen wahrheitsgetreu angegeben zu haben.

Ich nehme zur Kenntnis, dass meine Arbeit zur Überprüfung der korrekten und vollständigen Angabe der Quellen mit Hilfe einer Software geprüft wird. Zu meinem eigenen Schutz wird die Software auch dazu verwendet, später eingereichte Arbeiten mit meiner Arbeit elektronisch zu vergleichen und damit Abschriften und eine Verletzung meines Urheberrechts zu verhindern. Falls Verdacht besteht, dass mein Urheberrecht verletzt wurde, erkläre ich mich damit einverstanden, dass die Schulleitung meine Arbeit zu Prüfzwecken herausgibt.

Glarus, 03.12.2023

_____ _____

Ort, Datum und Unterschrift