RtoV - Deep Learning with Generated Data for Raster Vectorization

Matura Paper by Lars Hoesli

Supervisor: Beat Temperli

December 2023

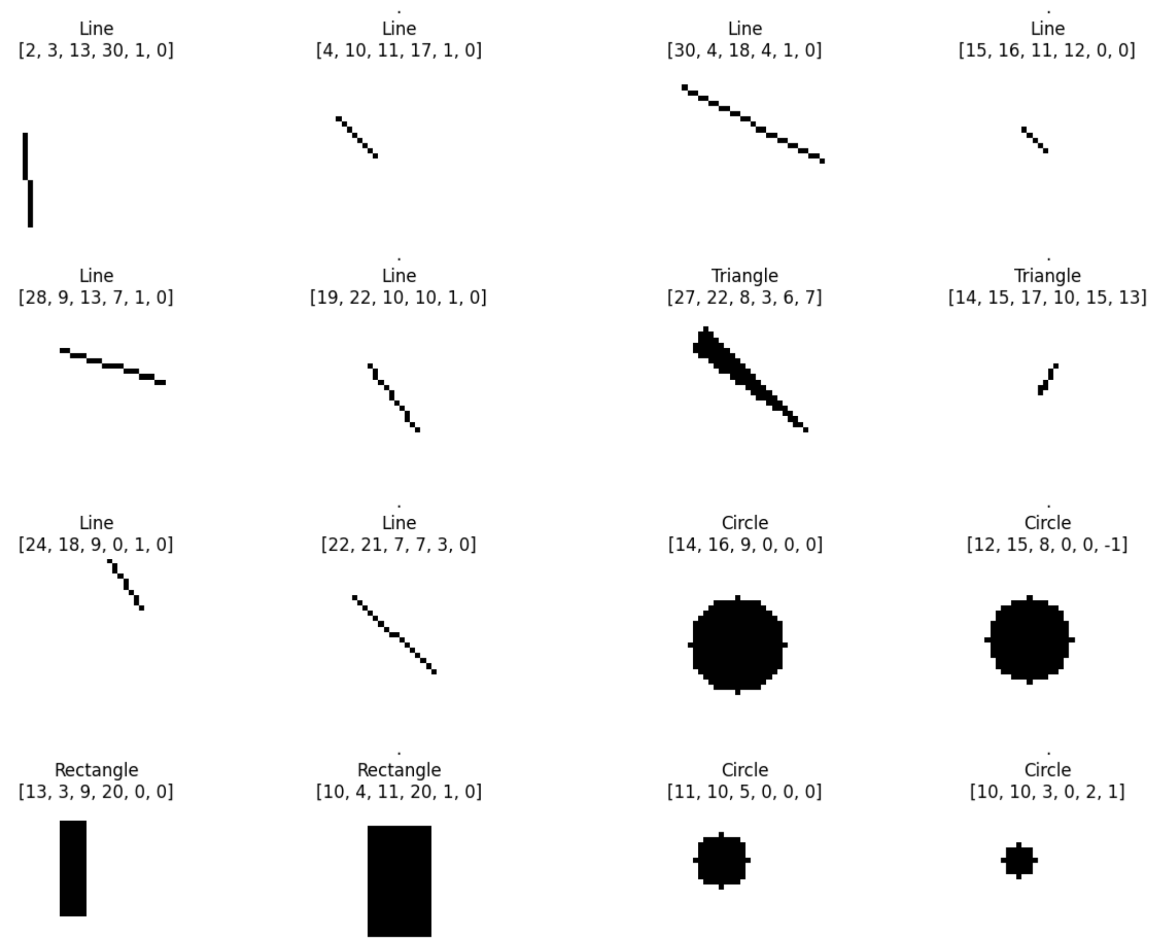


Figure 1: *An example output of the model used in the demonstration. Two images form a pair, the first being the input image and the second the model output, converted into a raster representation (source [I3])*

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

1.1 Raster and Vector Graphics

Raster and vector graphics are two fundamentally different approaches to representing the content of an image. While raster images store the color values of small parts of a given picture - usually referred to as 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. 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. To this class of images belong schemes, plots, logos, icons and many more. In these cases, vector graphics can be a more precise and efficient representation of the image than raster formats. A shape described in a vector specification can be stored with an infinite resolution, since there are no 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, landcape pictures or photorealistic paintings.

1.1.1 Problem Description

While format conversions among raster or vector and from vector to raster formats is being done by a multitude of programs, the conversion from a raster image to an accurate vector representation proves to be more challenging. This is especially because the shapes and their features seen in the input image have to be recognized in a grid of numbers, 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. However, 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 both beneficial and there is no universal solution to it. Deep learning is a powerful tool for this task, especially if the data to train a neural network could be generated indefinitely. This is why this thesis has been conducted on the topic of raster-to-vector conversion with deep learning, and focuses on the usage of generated training data to accomplish this task.

1.2 Fundamentals

To understand the concepts and terms used in this paper, a certain basic knowledge of neural networks and deep learning in general is required. In this section, the most relevant information is summarized. This includes general information as well as techniques that are used in the demonstration of this work or mentioned in the discussion. However, it is not meant to be comprehensive and conducting own research is strongly advised.

1.2.1 Neural Networks

Deep learning is done using *neural networks* with several layers of *neurons*. These neurons are connected by *weights* with other neurons. Each layer of a typical neural network has a *bias value* and an *activation function* associated with it. They are responsible for deciding whether a neuron should be activated or not. The network learns through making decisions, which are judged by loss functions. The model is subsequently either punished or rewarded, and its weights are adapted. In that manner, neural networks learn how to accomplish a task through trial and error.

Whilst the bias value is relatively trivial, the various activation and loss functions are important to understand.

1.2.2 Activation Functions

Activation functions are essential components in neural networks. They are, together with the bias value, responsible for deciding whether a given neuron should be activated or not based on the weighted sum of its inputs.

One widely adopted activation function is the rectified linear unit (ReLU) function. ReLU is a simple yet efficient activation function. Mathematically, it is defined as:

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

Which is graphically plotted in 1.1.

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 $[-1, 1]$.

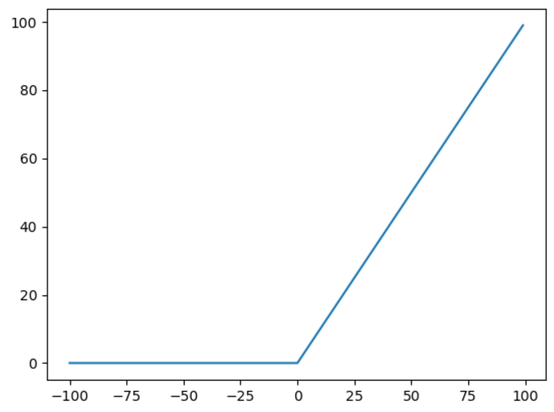


Figure 1.1: *ReLU function plotted with* $x \in [-100; 100)$ *(source [I2])*

1.2.3 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.

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:

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

This is the formula for MSE 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 the following formula applies:

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

Minimizing this output of a loss function encourages the output of the model to converge towards the true values 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 may be a better fit 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, a CNN 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. A 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.

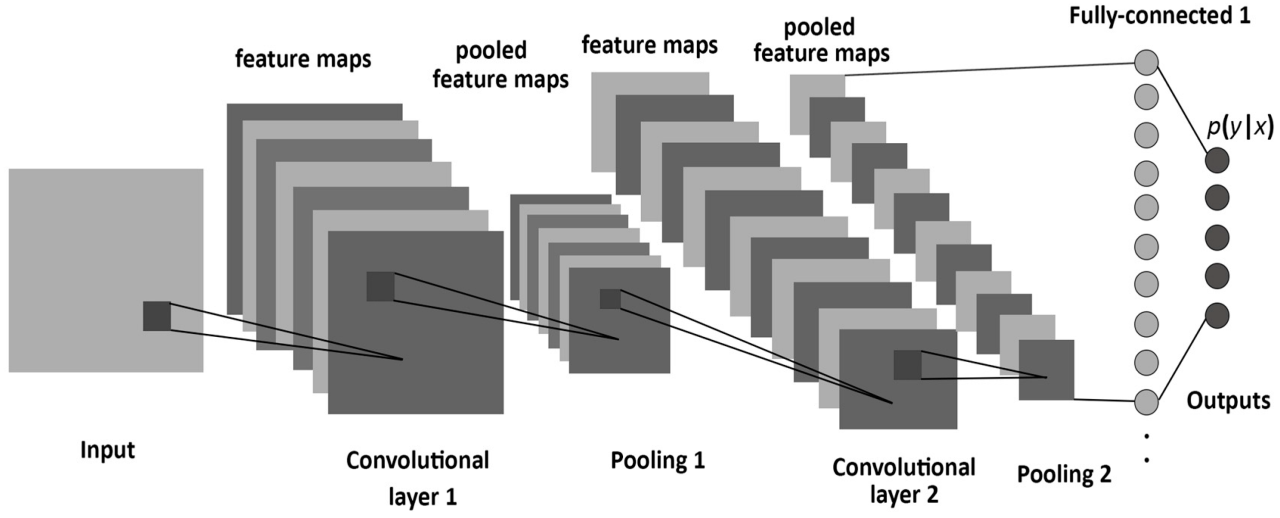


Figure 1.2: *Typical convolutional neural network architecture (source [I1])*

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 and thus for raster-to-vector conversion.

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.

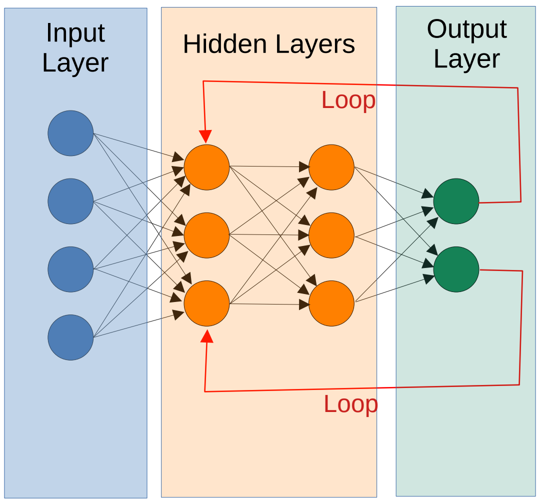


Figure 1.3: *Example recurrent neural network architecture (source [I7])*

RNNs have proven valuable in various applications that involve sequential data, allowing machines to comprehend and generate sequences more effectively than standard feed-forward networks. In the case of raster vectorization, this architecture could allow a neural network to remember which shape or vertex has previously been processed, allowing the model to decide which to focus on.

1.3.3 Reinforcement Learning

In reinforcement learning (RL), a neural network (*agent*) learns to make decisions by interacting with a given *environment*. This agent makes *actions* in the environment which modify a given *state*. Based on these 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 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].

The concept of reinforcement learning being very general, a specific use-case for raster-to-vector conversion is presented below. The proposed use of RL for vectorization in this paper is inspired by the work ”Marvel - Raster Manga Vectorization via Primitive-wise Deep Reinforcement Learning” ([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 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 separately. It also allows for information to be shared. For instance, the output of one block of layers might influence which other blocks are used for further processing the information.

Typically, multi-task learning model are used in large neural networks with several common initial layers. Those layers are often responsible for extracting the relevant data for the subsequent layers. After extracting those features, the model may split into several independent branches; blocks of layers, which are specific to a separate tasks each.

Multi-task learning is advantageous for raster-to-vector conversion. Typically, in raster vectorization models, comparatively much time is spent in the initial layers with feature extraction. The information that is 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 to reuse the output of these layers for the entire prediction.

Chapter 2

Goal and Hypothesis

A substantial amount of data is needed to train a machine learning model for vectorization of raster images. An ideal data-set 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?

IV. What are the limitations and pitfalls of this approach?

Chapter 3

Method

3.1 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, but also the flexibility and expected runtime performance of the language and its frameworks and libraries. 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 higher priority than the performance of the language. A rich and mature ecosystem in general were important as well. On this basis, the table 3.1 has been produced, which assigns points to each of the relevant factors. They were mulitiplied by a relevance factor to determine the best fit for the demonstration. Furthermore, a more in-depth case study of the different languages has been conducted, which provides insight on how the points were selected.

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*

*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 community and many libraries, such as *Matplotlib*, *Pandas* or *Pillow*, Python has earned itself 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. They are both compiled and can take advantage of type information and are therefore much faster, with little overhead when calling from and into Python.

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. The ecosystem in general is on a similar level as the one of Python.

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 even allow implementing the machine learning 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 safety. Much control, but also much responsibility, is given to the developer, which is not advantageous when fast iterations and changes have to be made.

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. It may be the best choice if the deep learning ecosystem was more mature.

Go

Ecosystem Go is a promising language, but its ecosystem is in a similar state as the one of Rust. Maschine learning libraries like *Gorgonia* might soon be as powerful as *Keras*, 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.

3.1.1 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 a simple language with a large ecosystem and very active community. An advantage that is often not considered is the ability of *Cython*, a Python superset and compiler, to call back and forth into C as well as Python code. This makes an application written in Python or Cython gradually optimizable if needed.

The next best choice might be Rust, but after some experimenting with different libraries, it was decided to stick with Python due to the robustness of its ecosystem. Large parts of Rust's ecosystem are still in an unstable state by the time of this work.

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.

3.2 Framework Choice

After deciding to write the demonstration in Python, the frameworks had to be evaluated in order to choose the best option. Since *Keras* has been merged into Tensorflow, the main choice was between Tensorflow and Pytorch. A point for point comparison was done, with no numerical evaluation 3.2.

3.2.1 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:

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 used libraries	Often used library	*More often used*

Table 3.2: *Framework evaluation table*

Simplicity and fast development speed in order to experiment with different architectures is invaluable for this work.

3.3 Overview of the Demonstration

The training and evaluation data for the model in the demonstration 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 shape 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 and adjusting of the model and its weights

3.3.1 Shape Generation

The training and evaluation data in the demonstration consists of white images with a single black shape on them. The shapes are one of the following four: *Line*, *Circle*, *Rectangle* and *Triangle*. The shapes and their parameters - such as position, size or their vertices - have been randomized. These features are stored as labels and provide the basis on which the model is trained.

A function has been developed for the random generation of each shape, returning a numpy array that describes the shape in a numerical representation. This allows it to be compared directly to the output of the model, without an 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 vertices and width: [x1, y1, x2, y2]. Width and color could be added.
- Circle: Center and radius: [x, y, r].
- Rectangle: One corner, width and height: [x1, y1, w, h]. A scalar controlling the rotation could be added.
- Triangle: Three corner points: [x1, y1, x2, y2, x3, y3].

When fully implemented, a color could be included in every shape, taking up four bytes of memory, for red, green, blue and alpha channels. In the unmodified demonstration however, 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.

The vertices, width and height values that have been 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 has not been considered. For instance, a triangle with its corners forming a line may lead to the same input image as a line with the same vertices 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 is written to a numpy array with the shape (100, 100, 3) or, in graphical terms, a white canvas of 100 times 100 pixels. 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.

3.3.2 Training

The training process is structured in epochs, which consist of several batches each. The batch size and the number of epochs are parameterized and can be set by the user, either as flags that are passed to the program or, in the case of the number of epochs, interactively after each training.

The training process on a batch can further be broken down into sub steps:

I. The training images and their labels are generated as explained above and received from the data-loader in a batch

II. The model processes the images from the batch

III. The shape loss is computed using the appropriate loss functions

IV. Neural network is being back-propagated with both losses

V. The losses are recorded for the statistics

The optimizer automatically updates the model according to the back-propagation of the losses. The most important factor for the effectiveness the training is the learning rate, which is a scalar value that determines the impact of the losses on the weights of the model. In the demonstration, no automatic learning rate optimizer is used. However, it can be specified interactively. Through that procedure, the model is trained on all batches.

3.3.3 Evaluation

For the evaluation of the model, its output is being compared to the ground truth. The model output consists of a numpy array containing a batch of shape predictions and second array containing a batch of the shape specifications. 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: *cross-entropy loss* (CE) for the shape prediction and *mean squared error* (MSE) for the prediction of the data specific to the shape.

3.4 Model Architecture

The model in the demonstration is a convolutional neural network, with the usual structure, with several branches as it is typical for multi-task learning. Each branch 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). 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 branches that is responsible for the classification of the shape. Depending on the shape that is being predicted in this branch, a branch specific for that shape is being used. For that, 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, as it is seen in 3.1.

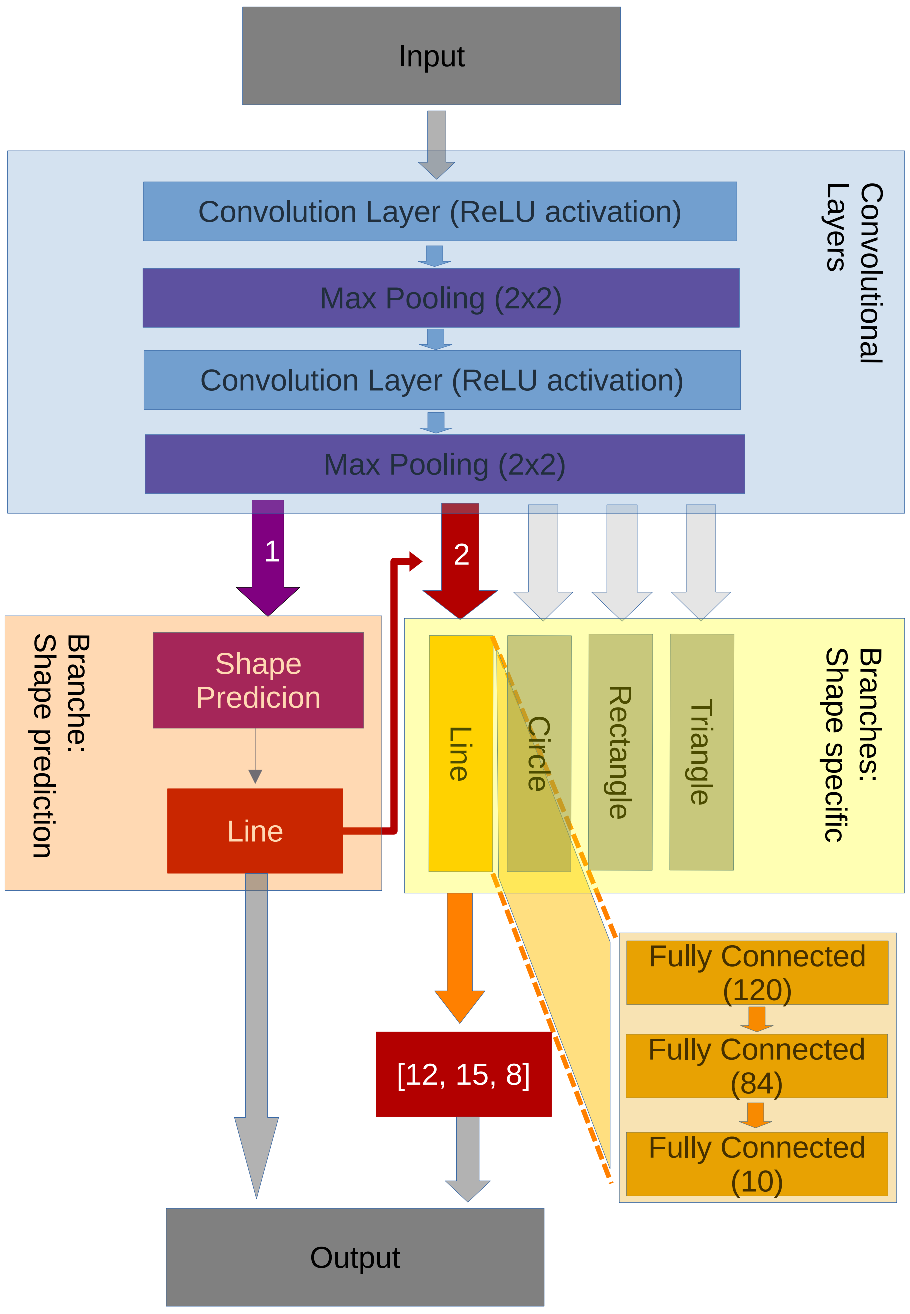


Figure 3.1: *Model architecture of the demonstration [18]. The gray boxes are the in- and outputs of the model; Bold arrows indicate the passing of the output of one layer to another. The red boxes are the predictions, while the other boxes represent the separate layers. The numbers under the fully connected layers show the number of neurons in the respective layer.*

Chapter 4

Results

The model in the demonstration has been able to approximate the shapes it was trained on. The vertex coordinates usually have a derivation from the ground truth of around 5 pixels for the shapes line, circle or rectangle. 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.

Three illustrations can be found in the appendix for example outputs of the model: Figures [I4], [I5] and [I6]. These outputs can be reproduced by following the documentation at *https://github.com/lrshsl/rtov/tree/main/docs* or scanning the QR-Code in the appendix.

Chapter 5

Discussion

5.1 Results of the Demonstration

A interesting observation was that the neural network had problems with approximating the shape *triangle*.

There is an explanation for why it was the triangle that the model could not learn to estimate correctly: It is likely due to the nature of the data it needed to extract from the image. What makes the data of the triangle different from the other shapes is that it consisted of three independent vertices, out of which the model could have chosen any at any place. The comparison through the MSE loss was not developed enough to determine that a correct vertex prediction was correct, if it was a different order than the label. Thus, the model may not have had the opportunity to make a guess that is rated as correct with a probability higher than $\frac{1}{3}$.

The combination of learning rate and this probability might have caused that problem to arise. From the model's point of view, guessing the real vertices must have not been worth it compared to approximating the center of the triangle. With this strategy, the model tried to minimize the punishment received through the quadratic nature of the MSE loss function.

This seems to be a problem that is very specific for the demonstration and is likely no an inherent issue with the more general approach.

Otherwise, the problems can be attributed to the time and resource constrains faced during this work.

5.2 Goals and Questions

The minimal goal has been achieved and the first question has been answered. It is a possible to approximate shapes in a raster image with a vector representation using a neural network that is trained on automatically generated data, as the demonstration has shown. The deep learning model had some difficulties with determining the vertices of triangles. It also could not achieve usable accuracy with the other shapes, but this can be attributed to the time and resource constrains.

This leads to the second question. *What strategies or measures have to be taken to make this [approach] feasible?* This question could be answered as well. The convolutional neural network with multi-task learning was not capable to correctly approximating all four shapes. With minor changes to the model and the training process, and with more training being conducted overall, it might have been possible. With those measures, as well as through increasing the complexity of the model, neural networks should be able to be trained effectively on generated data to learn to do raster vectorization.

The last questions could only partially be answered: *What performance can be achieved?* and *What are the limitations and pitfalls of this approach?* For the performance only a minimal example can be given, however, better performance can certainly be achieved. No severely limiting factors have been observed, and the only pitfall or challenge with the approach is the scaling when working with triangles or polygons in the future. The problem that occurred with the triangles would have to be addressed properly when predicting more than three vertices with no given order. One solution is the usage of a neural network architecture with some kind of internal state such as RNNs or reinforcement learning models as explained in *Proposed Model Architecture*.

5.3 Evaluating the Approach

The demonstration is too limited to make general statements about the usability and scaling of this the approach. However, those observations could be made:

I. Even a model trained on a CPU can quickly learn to extract shapes from raster images using simple 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 once the vertices of polygons can be extrated.

IV. A more sophisticated model may be needed for further experiments. Thus, an example architecture has been proposed.

5.4 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 determining the color of the shape and background. A separate CNN might be useful for those layers, but this has to be investigated further.

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. For the evaluation of the model two fundamentally different approaches can be considered:

- Comparing the output of the model directly with the label. This approach is used in the demonstration and is computationally very efficient. It might be beneficial to use this in the first stages of training.
- 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, but at a higher computation cost. This approach is well-suited for the fine-tuning of a neural network.

VII. 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.

5.5 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 converting a raster image into 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; But triangles have proven more challenging since the order of their vertices is unpredictable.

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.

[I8] Author L. H. *Scheme RtoV 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/`.

Chapter 7

Appendix

Code The code for the model can be found at the following link or when scanning the QR-Code below: *github.com/lrshsl/RtoV*

This Paper This paper is available as PDF and latex files at *github.com/lrshsl/matura-doc*

Documentation for RtoV The documentation for RtoV is available at *github.com/lrshsl/RtoV/tree/main/docs/*

Figure 7.1: *Link to the github repository with the code for the demonstration*

Figure 7.2: *Link to the documentation for the RtoV demonstration*

The following illustrations are example outputs of the *test* command of 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.

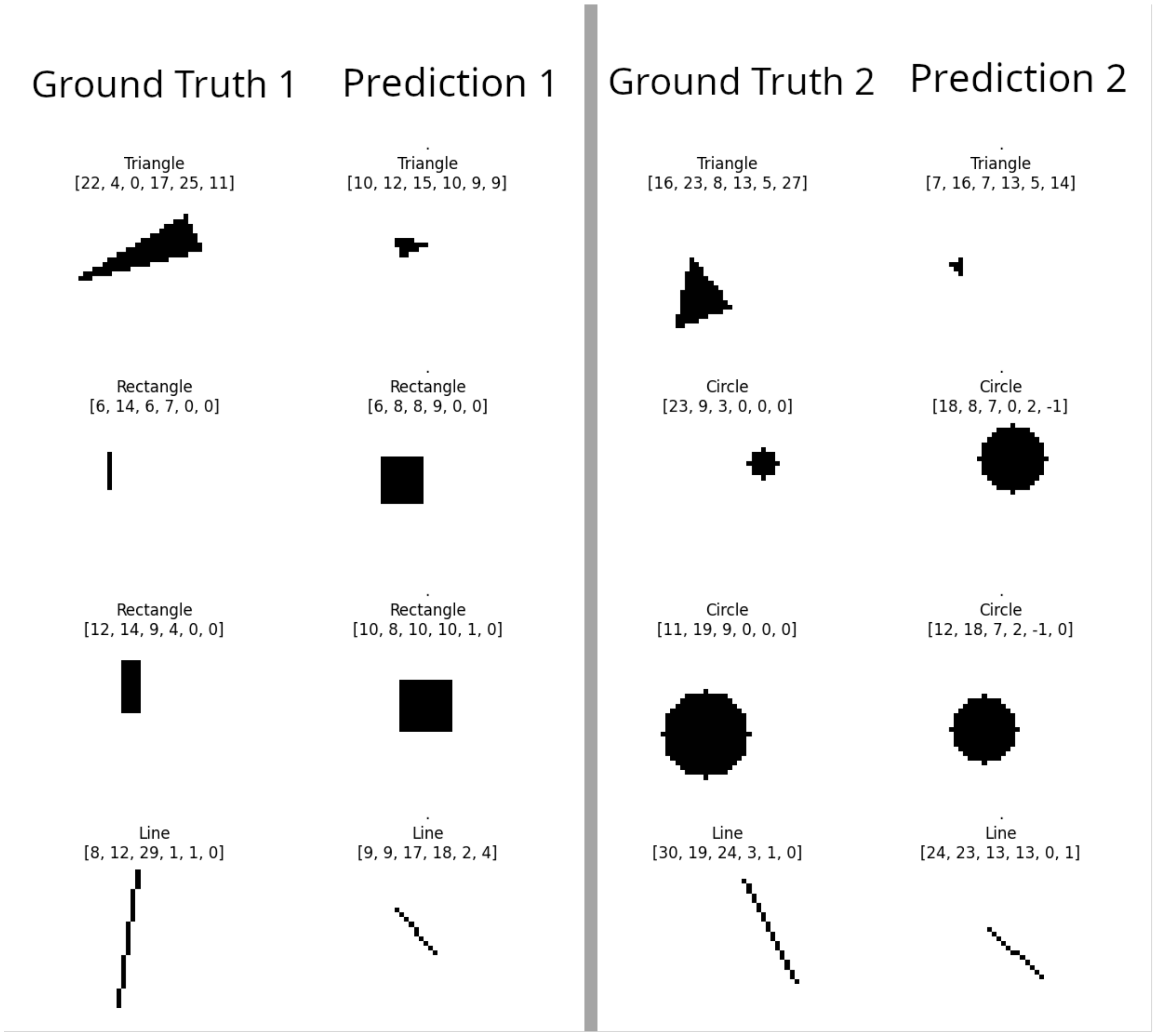
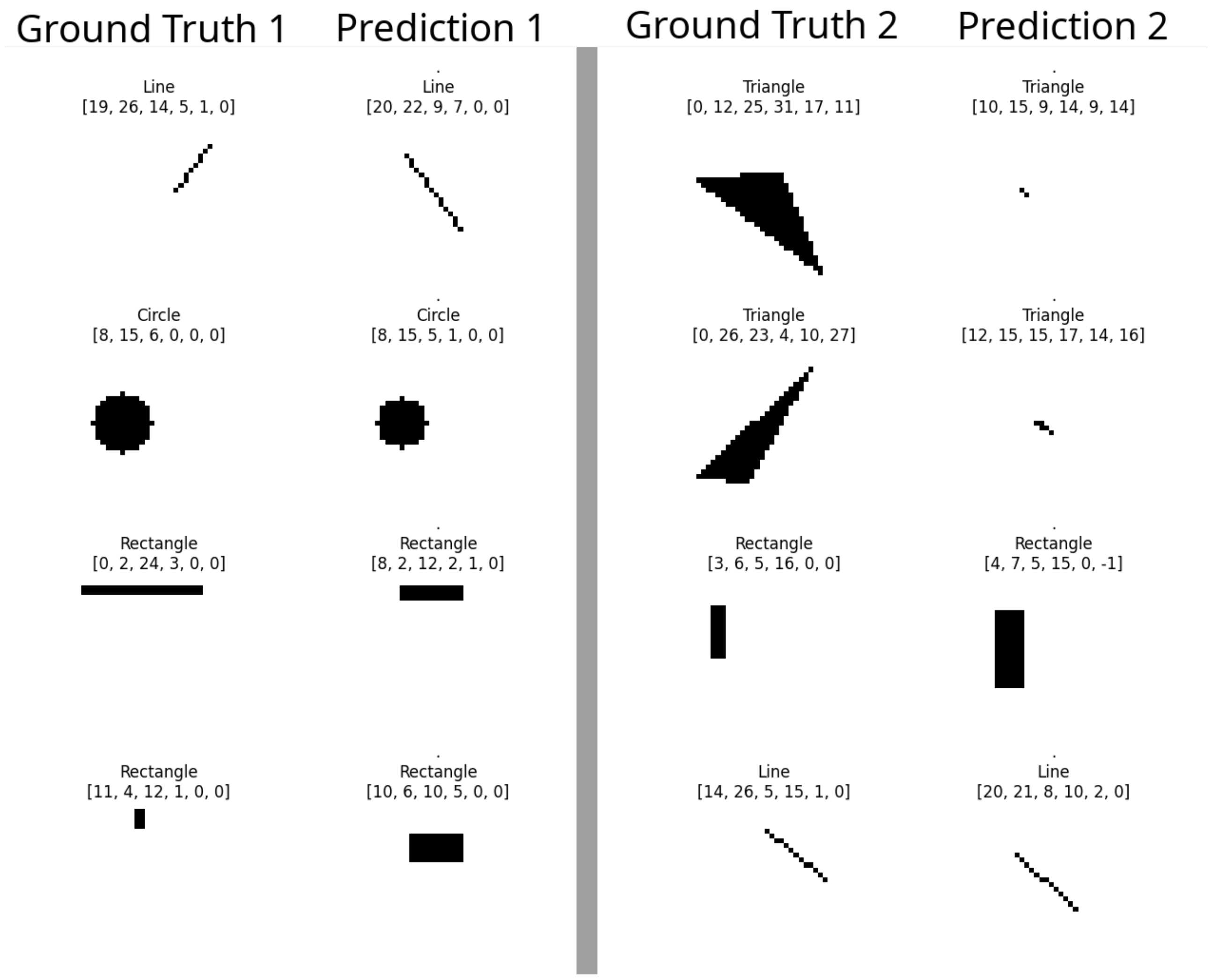
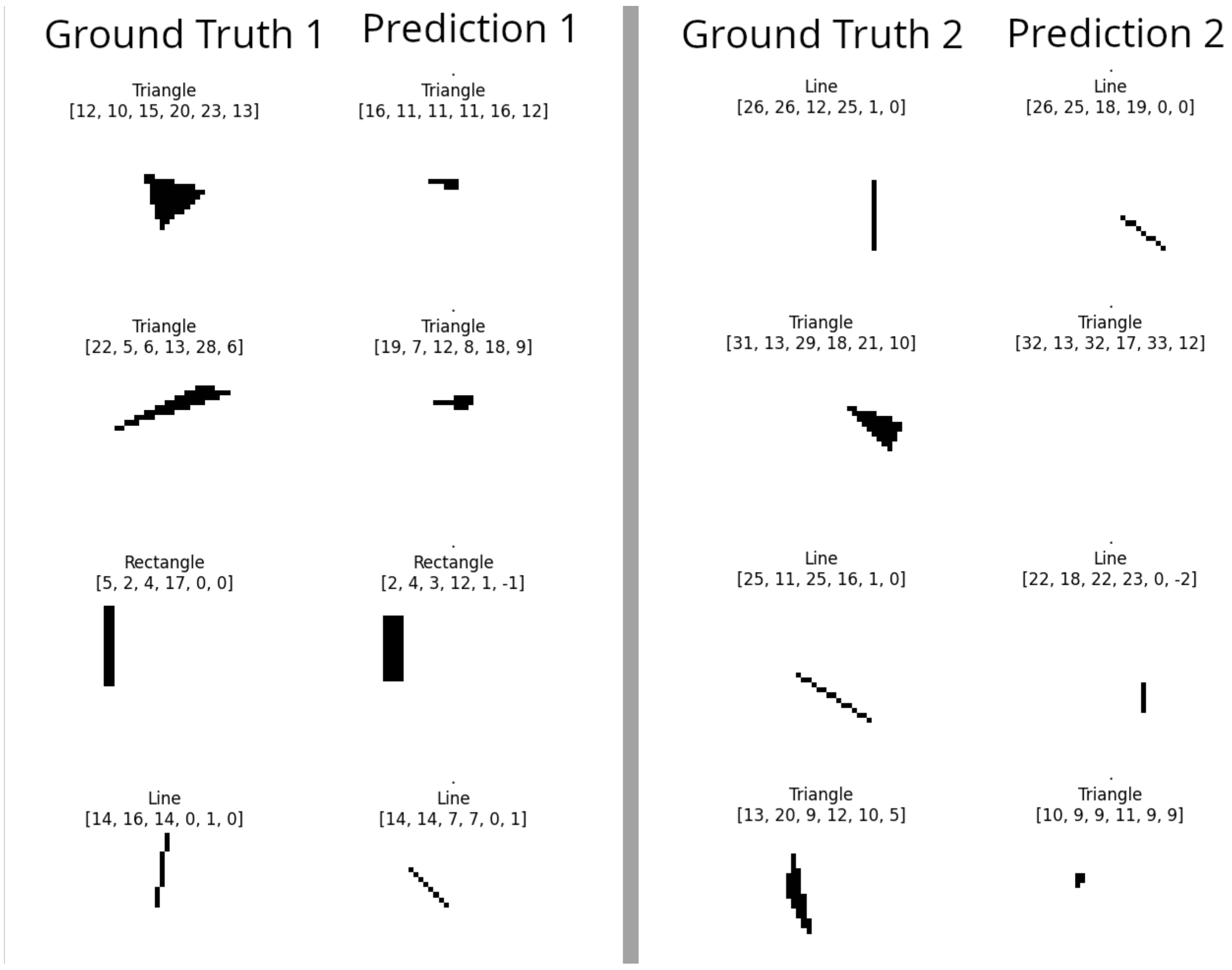


Figure 7.3: *Output of the model*


Ground Truth 1
Prediction 1
Ground Truth 2
Prediction 2
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]


Ground Truth 1
Prediction 1
Ground Truth 2
Prediction 2
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]

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