Solving Cutting and Packing Problems With Neural Networks

Wessel Sandtke
6261612

May 5th, 2021

Master Artificial Intelligence,
Utrecht University

Supervisor:
Dr. Till Miltzow (UU)

Second examiner:
Dr. Zerrin Yumak

Abstract

This thesis focuses on using neural networks to solve cutting and packing problems (C&P problems). C&P problems are a set of combinatorial optimization problems that generally aim to fit as many smaller objects into larger containers. These problems are common in manufacturing processes like leather cutting, car manufacturing, and container packing. In this thesis, we have attempted to improve upon commonly used methods to solve C&P problems as efficiently as possible. There are two main areas to improve. The first is the time it takes to solve a problem. The second is the final score, commonly measured by the percentage of the container filled. C&P problems have two aspects to them, checking methods and solution methods. Checking methods check whether different pieces do not overlap with each other or the container's border. Solution methods decide what item will be placed and where it will be placed in the container. We have conducted seven experiments. The first five experiments aimed to improve checking methods, and the last two experiments focused on improving solution methods. We have found that it is probably not feasible to replace current checking methods with neural networks due to the high precision required. We do believe, however, that solution methods might benefit from including neural networks. We have created a neural network based solution method that could not improve the score (percentage of the container filled) of non-neural network solution methods. However, it did manage to solve problems 61 times faster. These results indicate a strong possibility that solution methods can be improved upon by the use of neural networks, which would allow C&P problems to be solved faster and with higher scores.

Contents

1 Introduction

Cutting and packing problems are a set of combinatorial optimization problems that aim to fit items in containers in order to minimize empty (or wasted) space. These problems have a wide variety of applications in the real world and are a big part of sheet metal cutting, leather cutting, and the garment industries [11].

To illustrate the importance of minimizing wasted space during cutting problems, let us look at the leather cutting industry. For every 200kg of leather, 800kg of waste and 50,000 kg of wastewater is produced. Out of the 200kg leather produced, 15 to 20 percent is lost to inefficient cutting. Minimizing lost material due to increased efficiency in cutting would reduce costs for the leather industry and minimize environmental damage due to fewer waste products and emissions produced during the manufacturing process [16]. In this thesis, we aim to create new methods to solve cutting and packing problems. These improvements are made by designing and implementing neural networks to increase efficiency. If successful, this allows for future waste reduction in a wide variety of industries.

Over the years, a number of solution methods have been developed to solve cutting and packing problems (C&P problems). However, to the best of our knowledge, only a single one of these solution methods is based on the use of neural networks [4, 14]. This method, implemented by Hu et al. (2017), was designed to solve C&P problems in a three-dimensional space, leaving a gap in the literature that we intend to fill. This project will therefor focus on two-dimensional cutting and packing problems, solved with the help of neural networks. Specifically, we concentrate our efforts on a reasonably simple problem by fitting rectangular-shaped objects (also called pieces) on a two-dimensional square container (also called 'board').

C&P problems consist of two sub-problems. The first is checking whether a position is legal. This is done by checking methods. The second sub-problem decides what piece will be placed next and where it should be placed on the board. This is done by solution methods. In the first part of this thesis, we tried to create neural networks that could replace classical checking methods. Later, we found that this was not feasible. Therefore, we spend the second part of this thesis implementing solution methods. We created two different neural networks that try to simulate human insight by considering the current board-state and all the pieces that were yet to be placed.

This thesis is split into several parts. After this introduction, we define the C&P problems that we are looking at throughout this thesis. Afterward, we will write about the relevant literature to this thesis. We put this into practice in the section after that by designing and performing five experiments that attempt to create a checking method that rely on neural networks. Then there is a section with the other two experiments we performed that look at solution methods aided by neural networks. For each of those experiments, we explain how the data was generated, how the network was set up, trained, and our thoughts on why it performed the way it did. Finally, we will describe our concluding thoughts about our results and future research in the discussion and conclusion.

2 Problem Definition

Back in 1997, a typology was developed by Dyckhoff to differentiate between the many different C&P problems [11]. That typology was expanded upon in 2007 by Wäscher when a growing body of literature was not covered by the original typology [28]. We will use the definitions set in the latter of those two typologies since they cover the most ground and clear up some mistakes made in the typology by Dyckhoff.

In this thesis, we will focus on two C&P problems. First, the nesting problem. Second, the two-dimensional cutting problem, which Wascher categorizes as a SLOPP (Single Large Object Placement Problem). The nesting problem is a slightly more complicated version of the two-dimensional cutting problem. We will refer to the two-dimensional cutting problem as the cutting problem since this whole project is in two dimensions. At the start of this project, the goal was to work on the nesting problem exclusively, but as the work progressed, it was deemed unrealistically complex. The decision was then made to work on the cutting problem instead.

First, we will define the nesting problem, and after that, we will explain the differences with the strip packing problem.
As mentioned before, the nesting problem is a subset of C&P problems. It is concerned with placing objects onto a single two-dimensional square container with width W and length L. This container is called the 'board' and represented as B[1]. As input, the nesting problem takes a set of n pieces $p_1, \ldots, p_n$ that will be placed onto the board, denoted by B. The pieces vary in shape and size and specifically include at least one irregular-shaped piece.

Pieces can be moved both through translation, rotation or a combination of the two: $m := (t, R)$ where t is translation and R is rotation. If we have a piece p and motion $m := (t, R)$, then we define $p^m = \{x \in \mathbb{R}^2 | x = Ry + t$ where $y \in p\}$.

When rotating a point p with coordinates coordinates (p_x, p_y) by angle θ, the new position $p\prime$ is calculated by: $\begin{pmatrix} p\prime_x \\ p\prime_y \end{pmatrix} = R \times \begin{pmatrix} p_x \\ p_y \end{pmatrix}$, where R is the rotation matrix defined as $R = \begin{pmatrix} \cos(\theta) & \sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$.

A set of movements denoted as m needs to be found for each piece to find the minimal amount of space on the board $B = [0, \ell]^2$ that can fit as many pieces as possible. This means that we want to find the move m each piece p_i has to make p_i^m in order to fit inside the edges of the board B. For all the pieces combined, this is defines as

$$p_1^{m_1}, \ldots, p_i^{m_i} \subseteq B$$

[1]Within the operations research literature, there are some slight differences in how the board is defined. Some say that boards can have any shape [18], the consensus is that there is a single square board [28, 4]. This is the definition we use.

The constraint in this problem is that none of the pieces can overlap with any other piece: $\forall_{i,j} : p_i^{m_i} \cap p_j^{m_j} = \emptyset$.

As output, the nesting problem gives back a set of pieces and their position on the board with the goal of minimizing the space on the board that is not covered.

The cutting problem is the same as the nesting problem with 2 differences:

- The pieces that are used are all rectangular in shape (but can still differ in size).
- The pieces can only be moved through translation, not through rotation.

These two differences significantly reduces the complexity of the problem and thus make it easier to find meaningful solutions.

3 Related Work

To solve C&P problems, two smaller problems need to be solved.

The first problem is to see whether a position is legal. A legal position is one where no piece overlaps with any other piece, and all pieces are contained within the boundaries of the board. This is done by a checking method. The second problem is whether a final layout of pieces is the optimal layout for minimizing wasted space. A solution method is used to find this optimal layout.

In this section, we will first discuss the most common checking methods, and after that, we will discuss the most common categories of solution methods.

3.1 The Geometry of C&P Problems

To a person, it is easy to see whether two pieces overlap or not. To an algorithm, it is much more difficult. Even so, computers need to be precise when calculating overlap. If a faulty checking method is used in the leather cutting industry, people might end up with holes in their shoes because certain pieces of leather are missing pieces. Just as important is that a checking method is quick enough to be useful for general implementation. Even when a checking method is precise, it might be challenging to implement, which can lead to few people using it, as we shall see later with phi functions.

3.1.1 Raster Method

The simplest checking method is the raster method, also called the pixel method. It divides the board into discrete, equally sized areas. When a piece is placed, the values of the areas it covers are set to 1. When an area reaches a value of 2, it means that two pieces are both in the same area and thus overlap with each other [24]. The downside of this method is that it is not very precise. It only works properly if the pieces are integer-sized and have square corners. Otherwise, pieces of the raster are not completely filled and are still set to a value of 1. C&P problems with integer-sized rectangle pieces are a common sort of C&P

problem that we will focus on ourselves in the second half of this project. Making the areas smaller to increase the precision of the method is impractical because of the exponentially growing memory requirements for the growing matrices of values [3].

3.1.2 D-function

A more precise checking method looks at whether any line segments of one piece overlap with any line segment of another piece. Two line segments intersect with each other when the endpoints of one line segment straddle the other line segment and the other way around. There is a common edge case in which the endpoint of one line segments ends precisely on the other line segments, but since we allow that in our problems, we will not discuss that here [8]. When we say straddling, what we mean is that both endpoints of one line segment $\overline{p_1, p_2}$ are on opposite sides of another line segment $\overline{p_3, p_4}$ and the end points of that second line segment $\overline{p_3, p_4}$ are on opposite sides of line segment $\overline{p_1, p_2}$.

This can be calculated by calculating which way two consecutive directed line segments $\overrightarrow{p_1, p_2}$ and $\overrightarrow{p_2, p_3}$ turn at their shared point p_2. For this we compute the cross product $(p_1 - p_2) \times (p_3 - p_2) = (x_1 - x_2)(y_3 - y_2) - (y_1 - y_2)(y_1 - y_2)$ where each point p_i has coordinates (x_i, y_i). When the result of this cross product is negative, $\overrightarrow{p_1, p_2}$ makes a left turn at p_1 and if the cross product is positive, it makes a right turn at p_2.

To see if the end points of a line segment $\overline{p_1, p_2}$ straddle line segment $\overline{p_3, p_4}$, we use the cross product to calculate points p_1 and p_2 in relation to line segment $\overline{p_3, p_4}$. If one of these points is positive and the other negative, points p_1 and p_2 straddle line segment $\overline{p_3, p_4}$. The same thing is calculated for points p_3 and p_4. If they, too, straddle line segment $f\overline{p_1, p_2}$, the two line segments intersect. Figure 1 visualizes this.

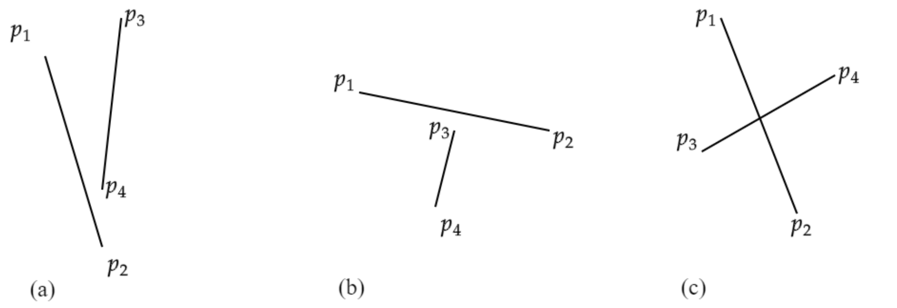


Figure 1: (a) Neither line segment is straddling the other line segment. (b) points p_1 and p_2 straddle $\overline{p_3 p_4}$, but since both points p_3 and p_4 are on the same side of $\overline{p_1 p_2}$, the line does not intersect. (c) the end points of both line segments straddle the other line segment, and as we can see, the line segments intersect.

The downside of using this method is that the number of computations grows exponentially with the number of edges of the pieces. A more efficient approach is by using bounding boxes to reduce the line segments that need to be compared with each other. First, a check is done to see whether the bounding boxes of two pieces overlap. If that is the case, see if the bounding boxes of each pair of edges from both items overlap. If that is the case, calculate whether the line segments intersect (as per the function above). Last, if that is not the case, one last check is done to see if one item is entirely inside the second item. This method reduces the needed calculations by at least 90% and sometimes more depending on the complexity of the pieces. Even with these improvements, using this method is still time-consuming, especially if the layout of the pieces is changed through iterative improvement (which we discuss later) [24].

3.1.3 No Fit Polygon

When pieces grow more complex, it will become more time-intensive to check whether any line segment of one piece intersects with any line segments from another piece. The no fit polygon (NFP) was designed to take two whole pieces and check whether they intersect with a single calculation. In its simplest form, the NFP is a polygon NFP_{AB} that shows the combined area of a tracing polygon B that has been moved around the edges of a fixed polygon A. The combined area shows where polygon B can legally be placed so that it touches the edge but does not overlap with A [18]. This total area is calculated by taking the edges of the fixed polygon in a counterclockwise direction and the edges of the tracing polygon in a clockwise direction. Those edges are sorted in order of slope and linked to each other in their sorted order to create NFP_{AB}. These steps are depicted in figure 2.

This checking method is more efficient than the D-function and just as accurate. However, it initially did not work with non-convex pieces. Several improvements have been made in order to resolve this.

The first of those improvements was the *sliding algorithm* by Mahadevan [20]. With the sliding algorithm, the tracing polygon moves along the shallowest slope of the fixed polygon. This is called the sliding edge. If there are concavities in either of the polygons, the tracing polygon will slide along the sliding edge, resulting in overlap between the two polygons. If that is the case, the D-function is used to determine the distance that the tracing polygon needs to be moved back along the sliding edge to not overlap with the fixed polygon.

This sliding algorithm could not notice holes in the fixed polygon, even if the tracing polygon fit entirely in that hole. This was solved by Whitwell [29] who proposed another addition to the NFP algorithm. Their solution was to check whether each edge of the fixed polygon had been traversed. If not, that would mean there was a hole in the fixed polygon, and the tracing polygon would use the untraveled edges as a new starting point to check for more legal placements.

Overall, NFP can check for intersections between polygons more efficiently while maintaining accuracy. A downside is that NFP is much more complicated

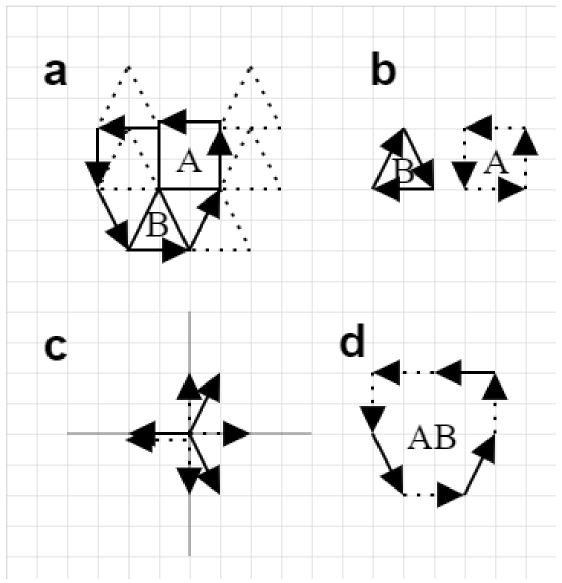


Figure 2: A: the fixed polygon A is traced by the tracing polygon B. B: Polygon A and polygon B with their edges pointed counterclockwise and clockwise, respectively. C: All the edges from polygon A and polygon B, ordered slopewise from a central point. D: The NFP_{AB} by reconstructing the edges of A and B in slopewise order.

to implement than the previously mentioned checking methods [3]. When using simple polygons (like we do in our experiments), there is no need to use NFP's.

3.1.4 Phi Function

The phi-function is the most recent development in checking methods. It is related to the NFP method in the way that, just like NFP, it takes as input fixed polygon A and tracing polygon B and creates a polygon AB, on the boundary of which, polygon B can be placed without overlapping with polygon A [27]. In its basis, phi functions split up the polygons into primary shapes (circles, rectangles, triangles, and other convex, regular shapes) and derives, through the D-function, the positions that are feasible for the two primary shapes to touch boundaries but not overlap.

The biggest problem with this method is that, as of writing, it is not available for general use, and every derivation is calculated by hand [3]. This is also the reason why we do not discuss it in depth. As soon as the method is made available for general use, the phi-function looks to be promising in fitting non-convex items in 2 and 3 dimensions [26], but until then, it is less relevant to our project.

3.1.5 Summary

Checking whether there is overlap between polygons takes a significant part of the computational time that is spent on finding the optimal solution for C&P problems [7]. This section has discussed several ways to check whether two polygons overlap as efficiently and accurately as possible.

NFP and the Phi function are more time-efficient when checking more complex polygons for overlap but more challenging to implement and not always beneficial for every project. In later sections, we will attempt to build neural networks that can see whether complex polygons overlap or not. We hope that despite its long training time, a network like this might be able to outperform the NFP or phi function on efficiency when classifying whether complex polygons overlap or not.

3.2 Solution Methods

As discussed earlier, the second part of solving C&P problems is to find a solution (also called layout) which is the final position of all the pieces on the board. This is done with a solution method.

Even with relatively few complex pieces, it can be very time-intensive to find the optimal layout of pieces [15]. Due to the incredibly high amount of possible final layouts, it is not useful to brute force solutions. This means that most solution methods are based on heuristics [9]. In this section, we will attempt to outline the different approaches to solution methods.

3.2.1 Partial Solutions

Partial solutions are solution methods that function by placing one piece at a time, using a heuristic to fit them as densely as possible. Most of these methods are quite time-efficient but are not highly optimized for packing pieces as densely as possible [4]. When designing a partial solution, two things are important to consider. First, the placement rule. The placement rule decides where a piece is placed on the board. Second, the placement sequence. The placement sequence decides in what order the pieces are placed on the board. We will start with placement rules, and afterward, we will discuss the placement sequence.

One of the simplest placement rules is called Bottom Left (BL). As the name suggests, each piece is slid down and left until they touch the edge of the board or another piece. This is quite an intuitive method since one piece will always be placed in the bottom left corner. It follows that another piece should be placed against the right side of that first piece in the newly created bottom left corner.

Placing pieces against each other in this way has the added benefit that they can be seen as a single composite piece which can be used as the single fixed polygon for an NFP. This composite piece allows for a quick calculation to find the legal placement for the next piece [10]. An improvement to the BL strategy is by trying to fill the empty space between already placed pieces. This is called a Bottom-Left-Fill solution. The pieces are placed in order from large to small, and once empty space opens up between the larger pieces, the algorithm tries to fit the next (smaller) piece into those empty spaces [7].

A different approach to placement is the floating placement method. Rather than immediately putting the pieces on the board, it adds pieces one by one to a floating group of pieces that, as a whole, is not allowed to exceed the width or height of the board. With this floating group of pieces, the frontier on which new pieces can be places is much larger. The new placement is the one that does not overlap with other pieces (naturally) and minimizes the total increase of the bounding box of all the pieces taken together [23].

As we saw with the Bottom-Left-Fill method, the sequence in which pieces are placed on the board can play an important role in the quality of the final layout. There are three major placement sequences.

The simplest and also one of the fastest sequences is a random sequence. Because of its speed, it can be used in iterative algorithms that run multiple times and choose the iteration with the best layout.

Another sequence is a predefined sequence according to some rule, for example, largest to smallest. This can be helpful because it might allow for the filling up empty spaces like mentioned earlier. When a predefined placement sequence is used to fit a heuristic used in the placement rule, it can be beneficial, as shown by Dowsland [10] who achieved the most time-efficient results when using predefined size-based placement sequences. Their placement strategy was large to small while checking whether smaller pieces fit in the holes between larger pieces.

Lastly and most complex of the major placement sequences is dynamic selection. This placement sequence allows each piece to be placed next, as long as it has not yet been placed. Dynamic selection is usually coupled with a search tree approach where backtracking is allowed to simultaneously explore multiple solutions [4].

3.2.2 Complete Solutions

Complete solutions look at the complete layout of all the pieces at the same time. This differs from partial solutions because those only looked at placing pieces one by one on the board. This also means that, rather than building up a solution as densely as possible, complete solutions try to achieve improvement by small iterative changes over the final layout.

When working with a complete layout, there are two main ways in which the solution is represented, through its placement sequence and its physical layout [3].

When searching through a sequence, each piece is enumerated. This decides in what order the pieces are placed. The actual placement will be done by a placement rule (for example, Bottom-Left). Improvements to the sequence are made by swapping, inserting, or rotating pieces within that placement sequence. When the placement rule then puts down the pieces, the final layout might have improved (or degraded). This process is repeated until a satisfactory result has been achieved [13].
Another way to try and make improvements is by using genetic algorithms. The enumerated sequence of pieces can be seen as genetic material (especially if multiple pieces have the same shape), and different placement sequences can be combined to find improvements. Mutations can be used as well to find better final layouts. These mutations can be a change in orientation or placement in the sequence [1].

The second way to use a complete layout is by moving the pieces while they are on the board.
When searching through the physical layout, pieces are represented as though they can be physically moved around the board. This allows for a more continuous movement of the pieces, where they can be rotated more arbitrarily. This usually does require more computations since whenever a change is made, a checking method has to make sure that none of the pieces overlap because of said change. However, feasibility is not a constant necessity while searching over a layout. Overlap can temporarily be permitted to allow the pieces to be packed more densely in the final layout. A problem with this can arise where the infeasible solution cannot be made feasible by small adjustments and needs larger changes which might reduce the density of the layout significantly [4].

A more general way layouts can be improved, is by ensuring that each piece is always touching at least one edge (either of the board or another piece). By using NFP as checking method, pieces can easily be placed on the boundary of other pieces or the board [2].
Another way is to find pieces that are on the frontier of the layout and, if possible, move them into empty spaces between the pieces that have already

been placed [5].
Yet another way is to use guided local search. It starts with a board that is larger than the actual board and places the pieces in random legal positions. Then, the length and width of the board will be reduced by a certain small percentage, and if any of the pieces are intersecting with the board's boundary, they are given a new random position on the board. If they overlap with any other pieces, the overlap is reduced in steps by moving the newly placed piece away from the piece that was already there [12].

In summary, there are many different ways to place pieces on a board to fit them together as densely as possible. The focus can lie on placing pieces one by one or improve the solution by moving pieces around. The sequence in which pieces are placed is important, as is the location in which they are placed.

3.3 Conclusion

In this section, we have discussed, among other things, several ways to check whether two polygons overlap: the Raster method, D-function, No Fit Polygon, and Phi Function. We have also discussed several categories that solution methods can fall into: partial solutions and complete solutions, each with its own rules and heuristics.

In the next section, we will attempt to use neural networks to improve the checking methods discussed here. After that, we will try and build a solution method using neural networks.

4 Checking Methods

In the previous section, we have discussed the two different aspects of C&P problems (checking methods and solution methods). In this section, we will try to solve those problems with the help of artificial intelligence. First, we will attempt to create a checking method, and later we will look at solution methods.

We have conducted five experiments that focus on checking problems. All five are described below. Our aim with these experiments was to create a checking method that can classify, with perfect accuracy, whether multiple pieces intersect with each other. For the first experiments, the goal was not yet to outperform the current commonly used checking methods, but to improve our understanding of C&P problems and as a first step to solve C&P problems entirely with neural networks.

4.1 2 Line Segments

For our first experiment, a neural network was trained to classify between intersecting line segments and non-intersecting line segments. This neural network was trained because, as previously mentioned, checking whether a position is feasible or not is the most computationally intensive part of finding solutions in C&P problems [7]. We hypothesized that maybe in simple board states a classical algorithm would outperform our neural network, but that our neural network would be faster when board states became more complex. This did require the classifier to be completely accurate, even in complicated situations. For now, we were looking for a basis on which we could built more and more complex networks.

4.1.1 Generating Data

To generate data, a program was build using the Python programming language. It generated 150,000 line segments S. Each line segment S_i was represented by a set of two points $\{p_1, p_2\}$ where each point had an x and y coordinate $p_i = (x_i, y_i)$. All the points were floating point numbers $n \in (0, 10)$.

Pairs of line segments $\{S_1, S_2\}$ were taken and checked to see whether they intersected or not. To calculate this, we used the D-function that we discussed in an earlier section. If the pair of line segments intersected, they were added to one list, and if they did not, they were added to another list. From each of those lists, 15,000 pairs were taken and added to a CSV file. Each line consisted of eight floating-point numbers, representing the two line segments, and one binary value where 0 represented the line segments not intersecting and 1 represented the line segments intersecting.

4.1.2 Network Structure

The neural network was built using the Pytorch library, specializing in machine learning for Python programming language. An extensive overview of the documentation of Pytorch can be found at their website: pytorch.org. The neural network was a fully connected feedforward network that started with an input layer of 8 neurons (representing the 8 floating-point numbers that comprised each example) and three hidden layers of 256, 128, and 128 neurons. The output layer had two neurons to represent the binary state of intersecting or not.[2]

The network was trained multiple times (more on this later) with different activation functions. We used both ReLU and LeakyReLU because according to a 2015 study by Xu, LeakyReLU slightly outperforms ReLU. This is because LeakyReLU has a slight negative slope when the input is lower than 0. This prevents the weight from dying off and stop learning [30]. Figure 4 and Figure 5 show the ReLU and Leaky ReLU activation functions. A dropout layer was added to prevent overfitting. Dropout layers set a percentage (usually 50% of the weights) to zero to prevent co-adaptation of patterns. Adding a dropout layer meant that the network takes longer to converge but had fewer problems with overfitting [25]. A dropout layer was introduced after the second hidden layer, setting half the weights of the second layer to zero every batch. We tried to

[2]We realize that only 1 neuron would suffice where 0 means no intersection and 1 means intersection, but the original experiment was conducted with 2 output neurons, where the first one indicated no intersection and the second one intersection

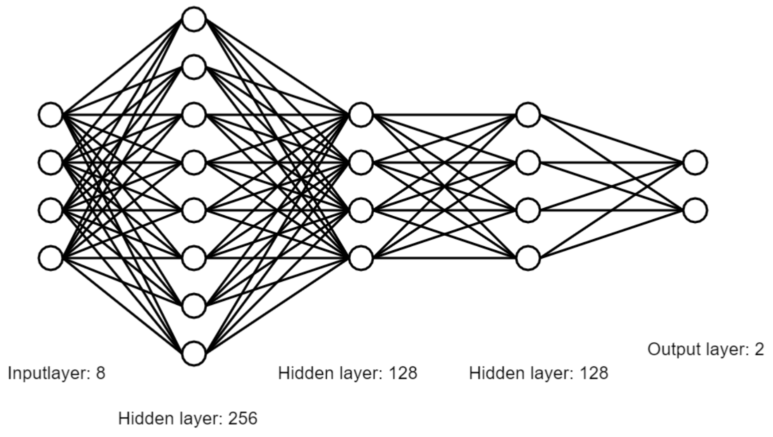


Figure 3: A representation of the classifier network. Do note that it only shows an approximation and not the actual amount of neurons in each layer.

use more dropout layers by adding them to every hidden layer, but this reduced the validation accuracy by 3%, so they were removed again. As activation function of the output layer, we used Softmax. Softmax takes the values from the output layer and turns them into probabilities that, together, sum up to 1. As loss function, we used cross entropy. Cross entropy loss is also called log loss. The loss function is used to adjust the weights of the network. The higher the loss, the larger the adjustment. Cross entropy uses the probabilities from the softmax layer as input and gives a 'loss value' as output. For binary cases, the loss value is calculated by the following formula: Loss $= -(y \log(p) + (1-y) \log(1-p))$ where y is the ground truth, and p is the probability that the network assigns to a label. It specifically penalizes confident (high probability) wrong classifications. As optimizer, we used Stochastic Gradient Descent (SGD), and the learning rate was 0.05.

4.1.3 Training

The dataset was shuffled and split into 3 parts for training: a training set, a validation set, and a test set. The training set consisted of 80% of the data, the validation set consisted of 15% of the data, and the test set consisted of 5% of the data. This split was consistent for all our experiments. The training set was used to train the neural network. The network trained 300 epochs. During each epoch, the network received the data in batches of 128. During each epoch, the whole training set was classified. For each wrongly classified example, the loss was calculated. That loss was propagated back through the network, adjusting neurons so that they would fit the ground truth better. After each epoch, the data from the validation set was tested to see how well the classifier could classify those. The weights were not trained after classifying the validation set.

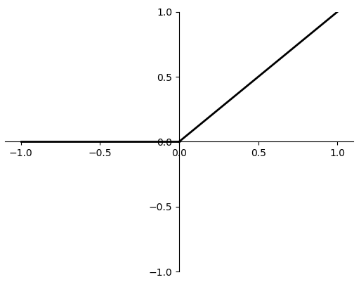


Figure 4: ReLU has a zero negative slope.

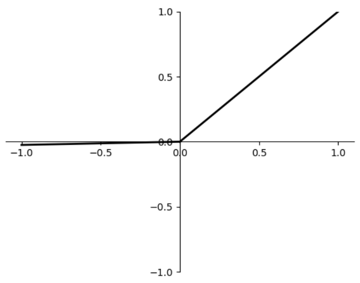


Figure 5: LeakyReLU has a non-zero negative slope.

This is commonly done to see whether the neural network is learning to solve a generalized problem or whether it is just starting to recognize and learn the solution to the training data. After all the epochs were completed, the test set was classified. The classifier had never seen this data before. The accuracy score from the test set was taken as the final accuracy of the network.// The training was done using an Asus Zenbook UX305U (used for all experiments, unless stated otherwise).

4.1.4 Results

As mentioned before, the network was trained twice. The first network, using ReLU as activation function for the hidden layers, correctly predicted whether two line segments intersected in 97.11% of the cases in the test set. The network that used the LeakyReLU activation layers was correct 97.55% of cases in the test set. There was a small amount of overfitting for the training data. A larger dataset would be used in later experiments to try and diminish the current rate of overfitting.

Figure 7 shows a more detailed look at the last 200 epochs of the network's training, using the LeakyReLU activation function. It shows that even in the later stages of the training, a small improvement is made. The reason that the network was not trained until the accuracy stops improving is because of time constraints. Currently, the network takes 7 minutes to be trained, and the last 2/3 of that only increases the validation accuracy by 1,5%. We decided not to let the network train longer than this since the validation accuracy increase would be marginal.

4.1.5 Misclassifications

With a test set accuracy of 97.55%, 37 pairs of line segments were misclassified out of a total of 1500 examples (5% of the total dataset). We randomly took some of these misclassified examples and visualized them to see what kinds of

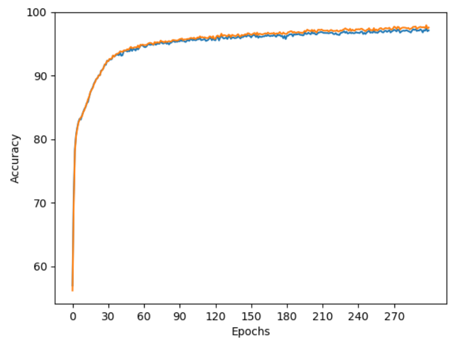


Figure 6: This figure described the accuracy of the neural network classifier over time. The orange line is the validation accuracy score and the blue line is the training accuracy score. Both in percentages.

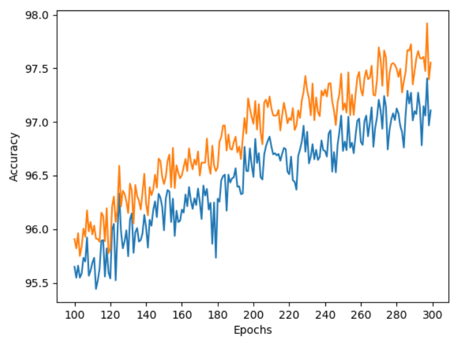
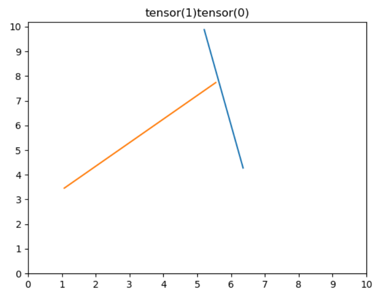
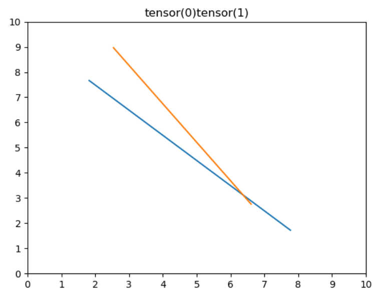


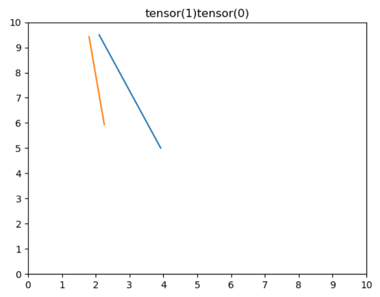
Figure 7: A more detailed figure of the last 200 epochs and the accuracy of the training and validation set. Both the accuracies are steadily rising, but this is two thirds of the training for only a 1,5% increase which indicates that the network is very close to convergence



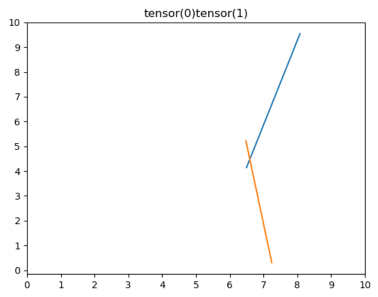
(a) Misclassified line segments



(b) Misclassified line segments



(c) Misclassified line segments



(d) Misclassified line segments

Figure 8: Four examples of misclassified line segment pairs from the neural network.

mistakes are made and how they can be prevented in the future. As shown in Figure 8, the classifier seems to have trouble with the endpoints of the line segments. When the endpoints intersect slightly, or an endpoint almost intersects with the other line segment, the classifier tends to make mistakes. As we mentioned before, it is essential to get 100% accuracy in order to be useful for general application. When these programs are used in, for example, sheet metal cutting, overlap between pieces might mean that there will be a hole in the hull of a ship would be problematic.
This is why our next experiments focus on precision and finding even the smallest amount of overlap.

4.2 Point inclusion Accuracy

After the first experiment, we realized our neural network would not be sufficient in more complex situations. Predicting whether two line segments intersected with 97.55% accuracy would not be enough when there were more complex pieces on the board. That is why this second experiment was designed with accuracy as the goal. The hypothesis was that a different neural network architecture might allow us to get a perfect prediction score on the previous test. A simpler test was designed to train the network faster and test for accuracy with a high degree of precision. This network could serve as a proof of concept. If this network architecture was able to achieve a perfect accuracy, we would try and implement it in the previous experiment design to see if we were able to improve the accuracy score.

The experiment is a simple point inclusion test. We generated a circle and points that were distributed inside and outside the boundary of that circle. The neural network would receive the coordinates of those points as input. As output, it had to classify whether they were inside or outside the boundary of the circle.

The main difference in the network architecture was the change in activation function. This change was made because ReLU and LeakyReLU are both classified as piece-wise linear functions. Activation functions that can approximate a curved decision boundary but can not curve accurately to replicate the decision boundary perfectly [22].

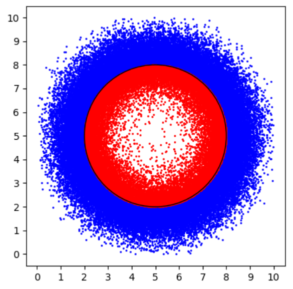
We designed an activation function based on a quadratic function which meant that it was not piece-wise linear. We hypothesized that this activation function would better be able to curve the decision boundary accurately. We will discuss the specifics of this activation function later.

4.2.1 Generating Data

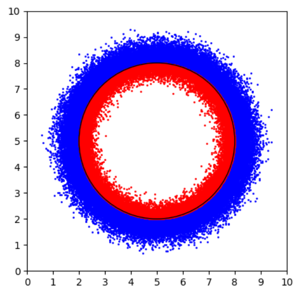
We generated multiple data sets for this task using Python. Each data set contained 50,000 points in a 2d space. Each point p was made up of two floating point numbers $p_{(i)} = (p_x, p_y)$ where each number $n \in (0, 10)$. On a 2d plane, a circle was drawn with the center $C_(x, y)$ at $(5, 5)$ and with a radius r of 3.

For each point, we calculated whether it was inside or outside the circle boundary using the following formula: $(x-5)^2+(y-5)^2 < 3$. When this formula returned true, p_i would be inside the circle boundary and would be given label '1'. When it would return false, p_i would be outside the circle boundary and given the label '0'. The first dataset consisted of the aforementioned 50,000 points with their label. Because of how the points were generated, they were uniformly distributed over the plane.

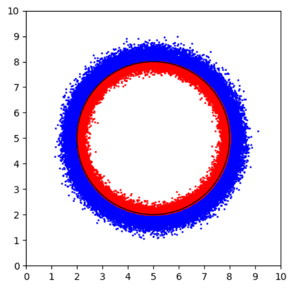
The second data set, which we call the 'First boundary-dense' data set, was generated by taking 3 times as many points as the previous data set and selecting 50,000 points located closer to the circle's boundary. The distribution of points we took was a normal distribution with the circle boundary as the average and a standard deviation of $r/2$. This dataset is visualized in figure 9a. The third, fourth, and fifth data sets were extensions of the second one (all with 50,000 points), but with progressively smaller standard deviations (one-fourth of the radius, one-sixth and one-eighth, respectively, as can be seen in figure 9b, 9c and 9d). We created these subsequent datasets to ascertain whether a network trained on data that was more closely surrounding the circle boundary would be more accurate.



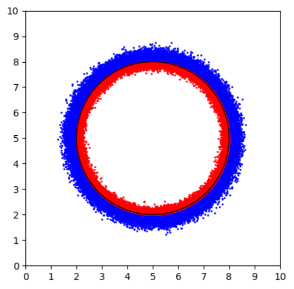
(a) First Boundary Dense Dataset



(b) Second Boundary Dense Dataset



(c) Third Boundary Dense Dataset



(d) Fourth Boundary Dense Dataset

Figure 9: The four boundary dense datasets were generated to train a neural network on more relevant data.

4.2.2 Network Structure

In terms of network architecture, this network resembled the classifier network that we discussed in the previous experiment. The number of hidden layers and amount of neurons was equal to the previous network, as shown in figure 10. The input layer was reduced to only 2 neurons, representing the x and y coordinates of each point. The output layer was reduced to a single neuron which represented the binary choice of whether a point was inside or outside the circle's boundary. The biggest difference in the network architecture was the use of a different activation function.

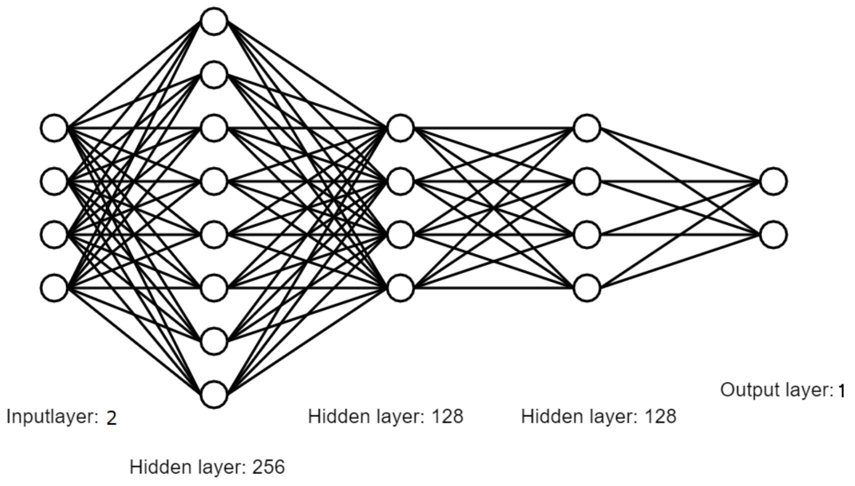


Figure 10: A representation of the classifier network. Do note that it only shows an approximation and not the actual amount of neurons in each layer.

As mentioned earlier, we built a quadratic activation function that could be implemented with the Pytorch library. A problem arose when we trained our neural network with this quadratic activation function. All the neuron weights would either grow exponentially or turn to 0 after a few batches. This was because quadratic equations, which can be described as $f(x) = ax + bx^2 + c$, includes a variable that is squared. This causes the input values to become unstable and grow or shrink exponentially. We attempted to cap the output values at 1, but this did not work either as all neuron weights eventually became either 0 or 1. Another attempt was made to make the activation function partially quadratic and partially linear by using a slight negative slope when the input value was less than zero. This did not work either, and neuron weights would either grow exponentially or shrink to zero.

Instead, the SoftSign activation function was used. SoftSign can be described by $y = \dfrac{x}{(1 + |x|)}$ which results in an equation that can be seen in figure 11. This activation function is called a smooth activation function because no part of it is linear, which should result in increased accuracy compared to the piecewise linear activation functions [22]. As optimizer, we used Stochastic Gradient Descent (SGD), and as activation function for the output layer, we used the SoftMax activation function. The loss function we used was CrossEntropyloss.

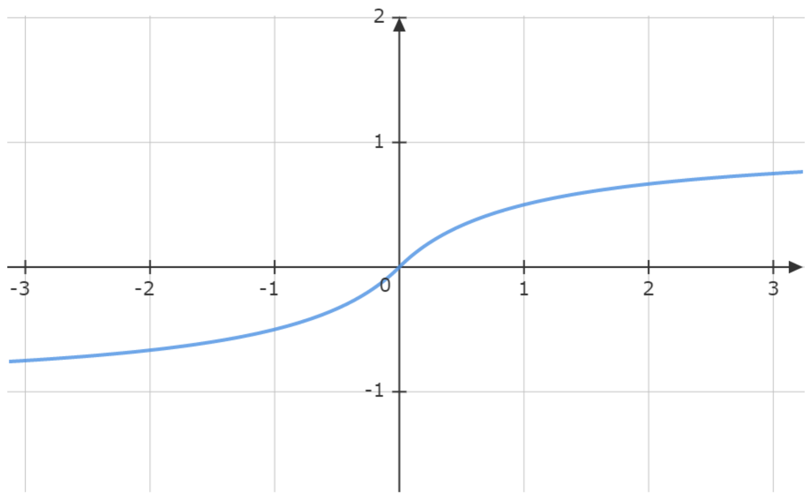


Figure 11: SoftSign activation function.

4.2.3 Training

During training, the dataset was shuffled and split into 3 parts. A trainingset, a validationset and a testset. The training set consisted of 80% of the data. The validation set consisted of 15% of the data, and the test set consisted of 5% of the data. The network ran for 1000 epochs with a batch size was 256, and a learning rate of 0.05. The learning rate was divided by 5 if the training accuracy had not improved after 50 epochs. If this happened 4 times, the training would stop even if it had not trained for 1000 epochs. This was implemented to prevent redundant training after convergence had been reached.

The network was trained four different times with different activation functions and datasets. First, it was trained using the uniformly distributed dataset and LeakyReLU as the activation function for the hidden layers to establish a baseline accuracy. The second test was performed with the uniformly distributed dataset, but this time with the Softsign activation function for the hidden layers. The third test was performed using the boundary-dense dataset initially. Rather than decreasing the learning rate every time the training set accuracy did not improve for 50 epochs, a progressively more boundary dense dataset replaced the previous training data. The subsequent datasets had data that clustered more closely around the circle boundary. We expected that it would help the

network to pinpoint the circle boundary more precisely. A second validation test was performed after each epoch using the uniformly distributed data to see whether the change in training data would affect the accuracy of an independent validation dataset.
The last test was performed using the boundary-dense dataset as training data and validation data. Like with the previous test, we used the uniformly distributed dataset as the second validation and test set. We wanted to know what would happen if the network was trained on data that would find the circle's boundary but validated using data that did not exactly mirror the training data. For example, the training data did not have points located in the corners of the board, while the validation set did.

4.2.4 Results

As mentioned before, the network was trained using both the LeakyReLU and Softsign activation functions. The network that used LeakyReLU as activation function reached a test accuracy of 98.80%, while the network that used Softsign as activation function reached a test accuracy of 98.93%. These results are deceptive, though, as the validation accuracy did not converge. Instead, it hovered between 93.13% and 99.70% without any indication that these scores would converge to a single point. This is shown in figure 12 and figure 13. Since both networks encountered this, we concluded that neither has a clear advantage over the other.

During the third test, more boundary dense data replaced the previous training data each time the training data did not improve for 50 epochs. However, rather than becoming more accurate with the higher precision data, the accuracy of the network started to worsen. We had expected the opposite. The results of this network can be seen in figure 14. The training and validation accuracy drop quite significantly and do not recover. The interesting thing is that the second uniformly distributed validation set had no drop in accuracy, although the variance in those scores seems to increase slightly.

The last test used the first boundary dense dataset as training set and validation set, but also had a second validation test running on the side that used the uniformly distributed dataset. This second validation set was to see whether the network would be able to classify points correctly that it had never seen before (for example in the corners of the board and the very center of the circle). The results, starting at epoch 200 and going until the end, can be seen in figure 15. As before, when the training accuracy does not increase for more than 50 epochs, the learning rate is decreased. The second time this happens (as denoted by the vertical red line), the validation score of the uniformly distributed data drops by about 0.3%. The training did not continue for long enough for this to recover, and the final accuracy was 97.18%. Like the results from the first two tests, these results are deceptive, and the validation score seems to be hovering between about 92.78% and 99.69%.

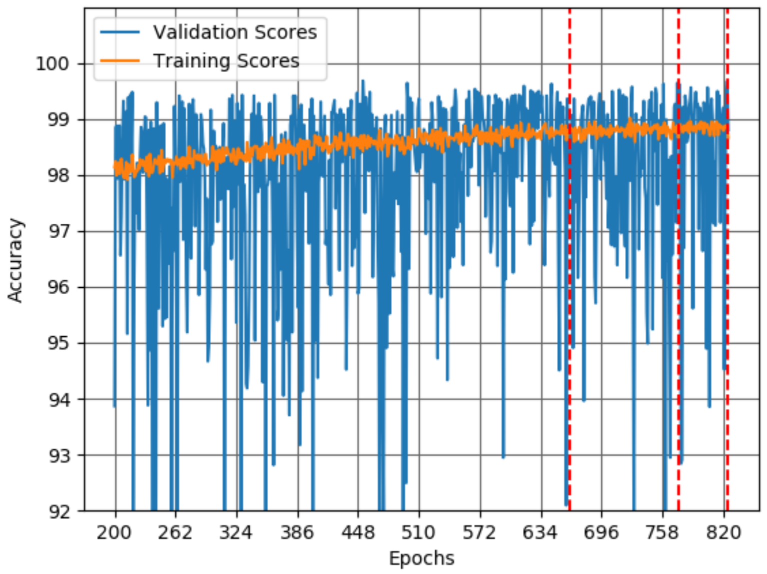


Figure 12: Accuracy scores over time when a neural network was trained on a uniformly distributed dataset while using LeakyReLU as activation function. The vertical red lines mark the moment at which the learning rate was diminished.

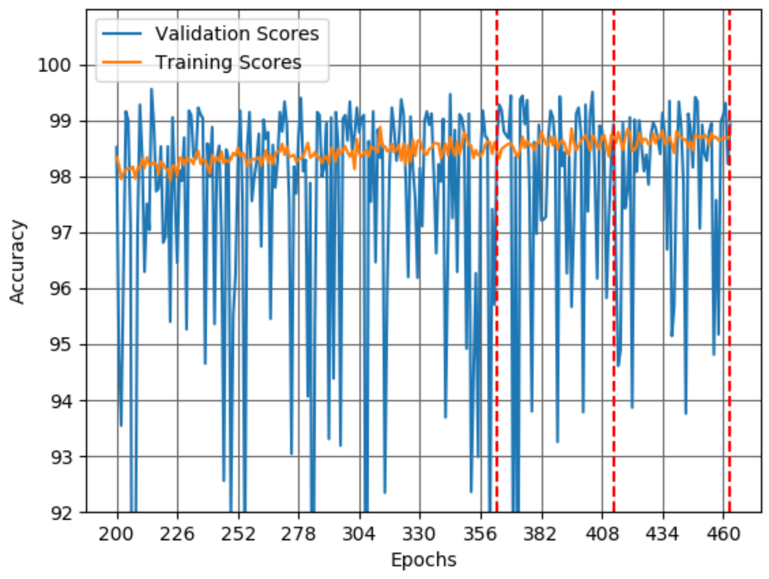


Figure 13: Accuracy scores over time when a neural network was trained on a uniformly distributed dataset while using Softsign as activation function. The vertical red lines mark the moment at which the learning rate was diminished.

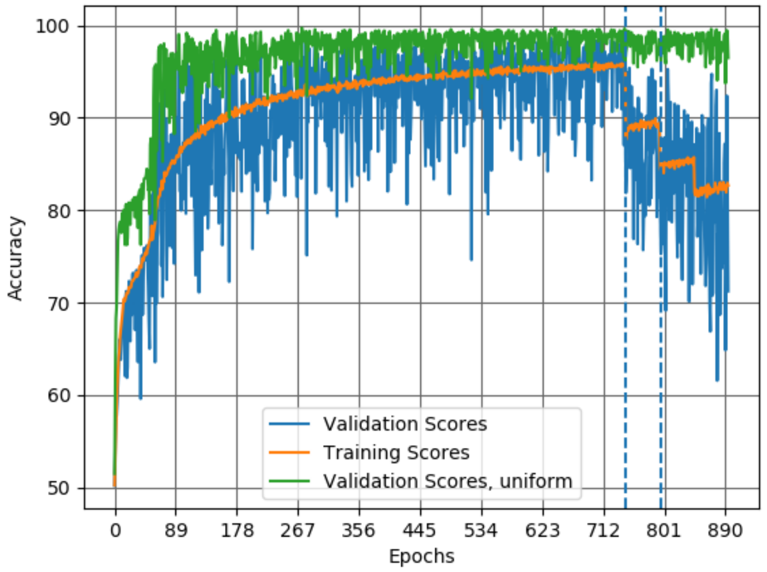


Figure 14: Accuracy scores of the network that trained on progressively more boundary dense datasets. Training data was replaced with a more boundary dense data set if the training accuracy of the network did not improve above its previous highest point for 50 epochs.

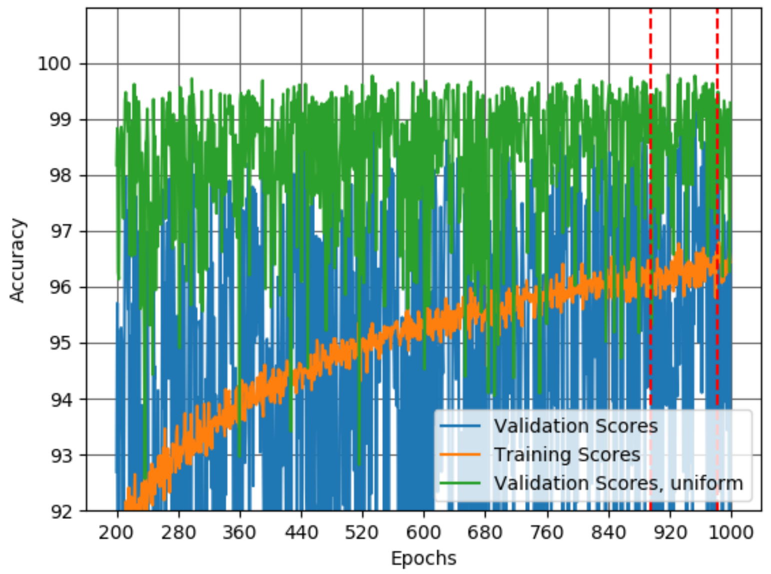


Figure 15: Accuracy scores of the network that trained the first boundary dense data set. A second validation set containing uniformly distributed data was used. This was to see whether the network, trained on a boundary dense data, would still be able to correctly classify the uniformly distributed data.

In conclusion, none of our tests reached the perfect accuracy that we were looking for. We are unsure whether we will be able to achieve 100% accuracy on this experiment. Different activation functions do not seem to lead to a significant increase in accuracy for this experiment. Since this is the simplest form of this experiment that we could design, we doubt that a different activation function would lead to different results in complex experiments. This might become a larger problem since checking methods must be highly accurate. All discussed checking methods except for the raster method have perfect accuracy, which means that any neural network with than perfect accuracy will be as useful. For the next experiments, we focused on slightly different experiments to see whether trying to solve more complex problems would give us insight into how to solve the simpler ones.

4.3 Increasing Input

The two previous experiments aimed to test the limits of accuracy for a neural network. This third experiment was designed to test the limits of the neural network regarding input size and board state complexity. Until now, the maximum size of the network's input layer has been 8 (representing two line segments). Despite not predicting with perfect accuracy whether two line segments intersected, one of the advantages of using neural networks can be their speed. Up until now we have looked at simple board state, but if a neural network is able to be reasonably accurate while looking at very complex board states, it might be more useful as a general checking method. This experiment tested whether the accuracy would stay as high as during the first experiment when the amount of input line segments (and thus the size of the input layer) increases. This was done by taking the line segment classifier and increasing the number of line segments used in each sample. Additionally, this meant that the network's output would increase since it would train to recognize the possible intersection between each possible pair of an n amount of line segments. We expected that the accuracy of the network would drop, but our goal was to keep that drop as small as possible.

4.3.1 Generating Data

We built upon the program used to generate line segments for the first experiment to generate data for this new experiment. The only difference was that rather than taking pairs of line segments, we took n amount of line segments. Each line segment was checked against every other line segment in the sample to see whether the two intersected.
We built multiple datasets. The first dataset consisted of 3 line segments per sample which meant that there were $\binom{3}{2} = 3$ output labels for every sample. Two more datasets were made. One with 5 line segments and $\binom{5}{2} = 10$

output labels and a last dataset with 10 line segments and $\binom{10}{2} = 45$ output labels. No effort was made to equalize the percentage of intersecting and non-intersecting line segments, which means that about 77% of the line segments did not intersect with each other. This percentage was consistent across the three generated datasets. All datasets consisted of a total of 50,000 samples.

4.3.2 Network Structure

The neural network architecture was substantially changed to the networks we used in the previous two experiments. This network had more hidden layers, and the layers had more neurons in them. To prevent overfitting the network, a second dropout layer was added. The first one was located after the second hidden layer and the second after the fifth hidden layer. In total, the network had seven hidden layers ranging from 128 to 512 neurons. The amount of neurons in each layer is specified and visualized in figure 16.

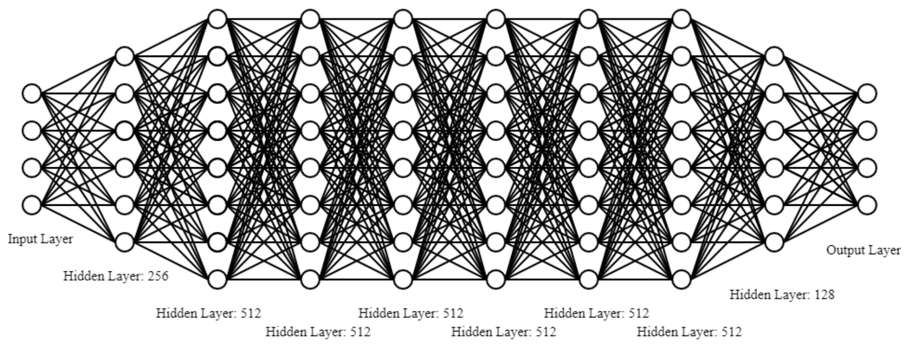


Figure 16: A representation of the neural network. Note that it only shows an approximation and not the actual amount of neurons in each layer.

The input layer consisted of neurons equal to the number of line segments multiplied by 4 since each line segment has two points p, each consisting of an x and y coordinate. The output layer consisted of the number of neurons equal to the length of the set of $\binom{2}{n}$ where n is the number of line segments. The output activation function was changed from a Softmax function to a Sigmoid function. This change was made because a Softmax function chooses a single option with the highest probability out of multiple options. In contrast, a Sigmoid function will give an output between 0 and 1 for each output neuron, allowing for a multi-label output. The Sigmoid function is described by $y = \frac{1}{1+e^{-x}}$.

The network had a batch size of 256 and a learning rate of 0.05, which was divided by 5 every time the training accuracy did not increase for 50 epochs.

If the learning rate had been divided 4 times, the training would end. If the network had trained for 1000 epochs, the training would end as well. The loss function used was BCELoss, and the optimizer was Stochastic Gradient Descent (SGD).

4.3.3 Training and Results

For training, the dataset was split into three parts: a training set that included 80% of the data, a validation set that included 15% of the data, and a test set that included the remaining 5%. The network was trained and tested several times, using the different datasets that were generated. First, it was trained on the dataset that included 3 line segments and 3 output labels.

The training results are displayed in figure 17 and show how well the classifier could predict line intersection when there are three lines and three possible line segment pair combinations. There was more overfitting than expected, and the training accuracy eventually reached 100% while the validation and test accuracy only reached about 92.72%.

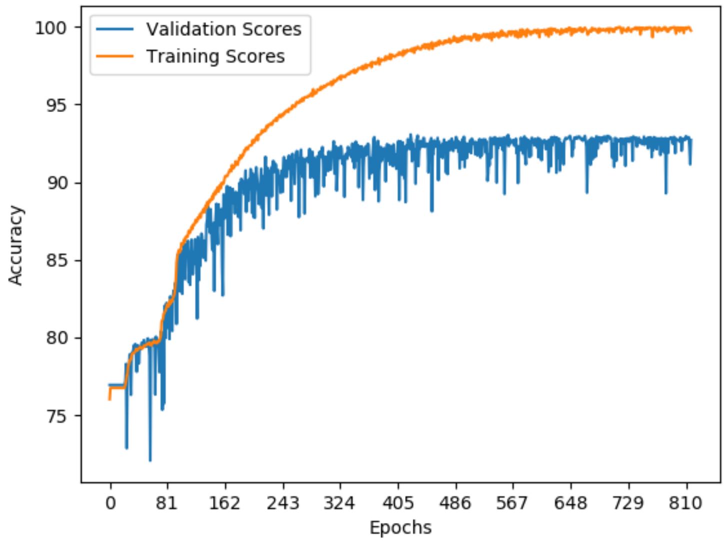


Figure 17: The accuracy over time of the training scores and validation scores of the line intersection classifier with 3 line segments.

The network overfitted more when using a more complex dataset with 5 line segments. The validation scores even started to decrease slightly after peaking at around 500 epochs. The test score ended up at 85.05%, as shown in figure 18. We have tried adding multiple dropout layers, but they did not decrease overfitting without decreasing the validation accuracy as well, and so they were removed again.

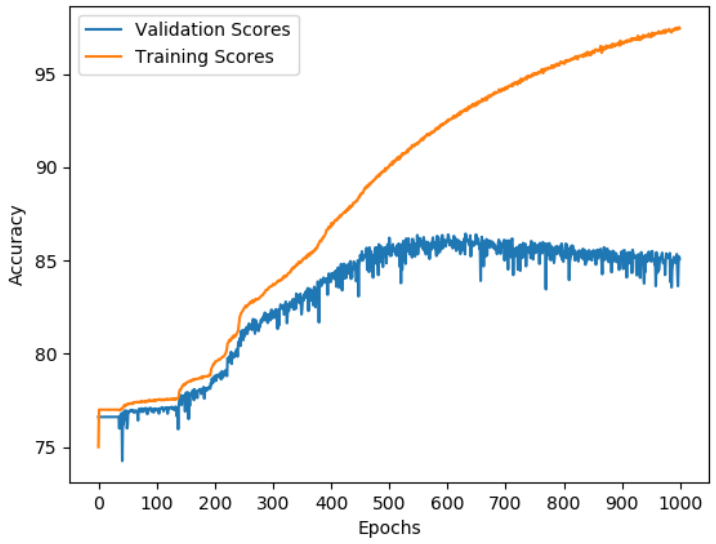


Figure 18: The accuracy over time of the training scores and validation scores of the line intersection classifier with 5 line segments.

When using the dataset with 10 line segments per sample, the network did not work and did not improve throughout the first 200 epochs, after which the training was cut short. We hypothesize that the 45 different output labels made learning too complicated for the network. This is a worrying trend since comparing 45 line segments with each other would be a lot less complex than even only a handful of relatively simple pieces in a real C&P problem.

4.3.4 Conclusion

The results from this experiment and the previous one show that our network is not able to achieve accuracy scores that would be useful in C&P problems. We have made two more attempts to create neural networks that could bring us towards a neural network based checking method. We will discuss those now.

4.4 Translation

The fourth experiment focused on the translation of line segments. Previously, we built a classifier that could detect the intersection of two line segments (although not with perfect accuracy). This experiment took the intersecting line segments and trained a neural network to recognize how far one line segment would have to move in each of the four axis-parallel directions to not intersect with the other line segment anymore. A network like this would solve an important part of C&P problems.

4.4.1 Generating Data

To generate data, a program was built using Python. This time we generated 50,000 pairs of intersecting line segments. Each line segment S_i was represented by a set of two points $\{p_1, p_2\}$ where each $p_i = (x_i, y_i)$. Each coordinate p_x or p_y was assigned a single floating point number where each number $n \in (0, 10)$.

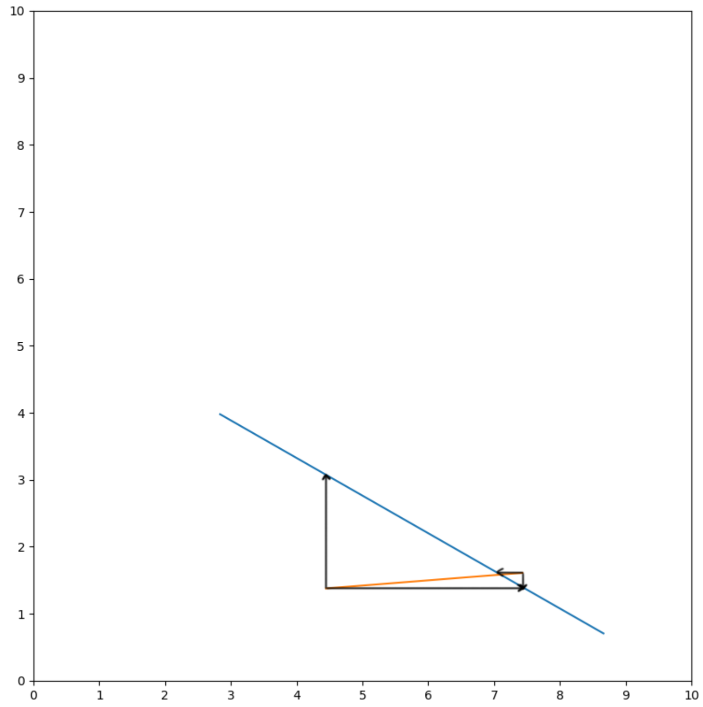


Figure 19: In blue the fixed line segment, in orange the moving line segment. Black arrows delineate the amount the moving line segment would have to move in each of the four axis-parallel directions in order not to intersect with the fixed line segment.

The goal of the data generation was to take two line segments, one would be designated as the fixed line segment and the other as the moving line segment. We then calculated the minimal amount the line segment would need to move up, down, left, and right to not intersect with the fixed line segment anymore. A visualization of this can be seen in figure 19. This was done by calculating the slope m of both line segments $m = (p_{2_y} - p_{1_y}) - (p_{2_x} - p_{1_x})$. The endpoints of both line segments were saved, and with that, the minimal amount of movement could be calculated. For example, when moving a line segment to the right, you only need to know the height of the right end of that line segment. That height can be put into the slope equation of the fixed line segment, allowing you to calculate the difference (and thus the minimal distance) the moving line segment has to move. This was done for each of the four axis-parallel directions an each line segment pair. It gave us the minimal distance the moving line would have to move in each of the four axis-parallel directions to not intersect with the fixed line segment. Together with the four distance measures, the coordinates of both line segments were saved in a text file.

4.4.2 Network Structure

This neural network consisted of 7 hidden layers with 256, 512, 512, 512, 512, 512, and 128 layers. Three dropout layers were added after the second, fourth, and fifth hidden layers. Each of these dropout layers set half the neurons of the previous layer to zero to prevent co-adaptation of patterns, as discussed in previous sections. Each hidden layer used SoftSign as activation function.
The input layer consisted of 8 neurons to represent the 8 coordinates of the 2 line segments. The output layer consisted of 4 neurons representing each of the axis-parallel directions of the moving line segment. The output layer of the network consists of four neurons. The activation function for the output layer was LeakyReLU. This activation function was chosen because of the continuous output that the network was expected to give.
The network was trained for a maximum of 1000 epochs. There was a possibility of the training being cut short if the accuracy score did not improve for 60 epochs in a row. The first three times that happened, the learning rate, which started at 0.05, would be divided by 5. After the fourth time, the training was cut short. As optimizer, we used Stochastic Gradient Descent (SGD), and the used loss function was MSELoss.

4.4.3 Training and Results

As before, the dataset was split into three parts before training. 80% went into a training set, 15% into a validation set, and 5% into a test set. The accuracy of this network was not decided like before by a percentage of right and wrong classifications, instead the network used continuous output labels. The success of the neural network was measured by taking the absolute difference between the label output and the ground truth of each of the four output neurons. Those four values were summed up and used as the total misjudged distance. The goal of the network was to minimize the total misjudged distance.

After training for 1000 epochs, the test set scored a total of 1.59 misjudged distance. So the total distance over the four directions was off by 1.59. This came down to about 0.4 misjudged distance in each direction. A graph with the decreasing total misjudged distance over time can be seen in figure 20. When studying some of the mistakes made by the network, we found that, like in the first experiment, the endpoints of the line segments were most often misjudged. The network managed to move the moving line segment by nearly the right

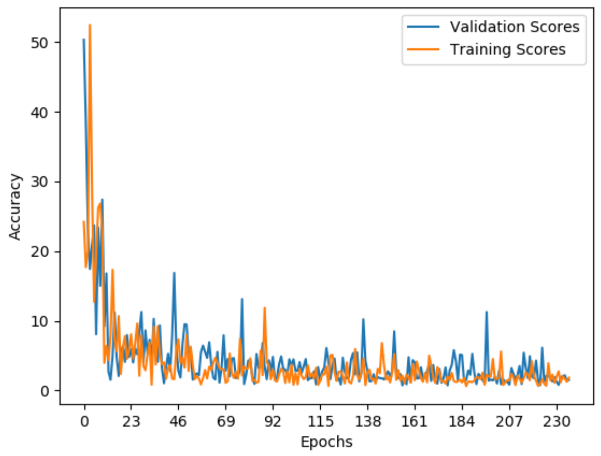


Figure 20: The misjudged distance scores over time.

amount, but the precision was lacking. This was difficult because we had been unable to improve on precision, even in simpler experiments.

4.4.4 conclusion

As mentioned before, there can be absolutely no overlap between pieces, and with an average of about 0.4 (1.59 divided by the four axis-parallel directions) there would be either overlap or wasted space between the different pieces. The results of this experiment, like the previous ones, are not sufficient to a point where the network can be succesfully used in a more general application.

4.5 Complex Polygon Intersection

Despite not reaching desired results, we decided to train one last network that would focus on complex polygons. We conducted this experiment since initially the aim was to be able to see overlap between complex polygons and so we wanted to at least try to built a network that does that. Even though we were aware that the previous networks did not work and so the chances of this one reaching perfect accuracy were not high.

4.5.1 Generating Data

For this experiment, we generated 120,000 8 sided polygons. For every polygon we would generate 12 points p_i, each with coordinates (x_i, y_i). Each of these coordinates was a floating-point number $n \in (0, 5)$. After generating those points, we connected them using Delaunay triangulation [19]. The outermost edges of these triangulated points formed the convex hull [21]. If this convex hull had eight sides, we would save it, but more often than not, the convex hull would have 6 to 9 sides. If this were the case, the line segments that make up the convex hull were added to a ‘frontier’ list. If the convex hull had too many sides, we would iterate over the points that made up the convex hull to see if there was any point that was only connected to two other points in the convex hull. That point would then be taken away, leaving only a single edge where there were two before. If the convex hull had too few sides, we would take two points of the convex hull and remove the edge between them, leaving two edges because of the Delaunay triangulation. An example of a point in the convex hull connected to only two other points in the convex hull, and an edge in the convex hull are shown in figure 21.

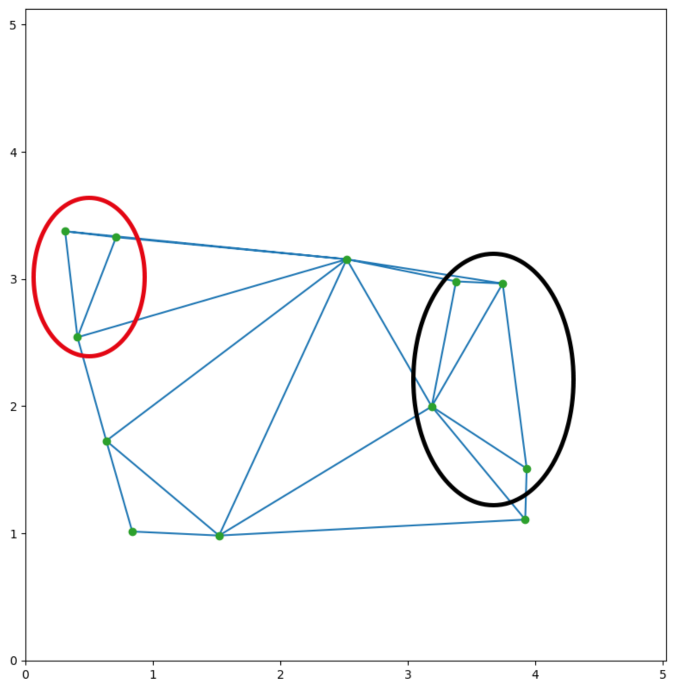


Figure 21: The red circle shows a point in the convex hull connected to only two other points in the convex hull. The black circle shows an edge in the convex hull that, when removed, allows two other edges to become a part of the convex hull.

This process was repeated until the polygon had the desired amount of sides. After this, the polygons we ended up with had eight sides and were more often than not non-convex. The last step was to take two polygons and assign them a random position on a board that was 10 by 10 in width and length. We checked to see whether the polygons overlapped with each other, using the improved D-function that we described in our literature overview. This method checked whether the bounding boxes of the two polygons overlapped. If it did, the bounding box of each line segment would be compared to each line segment of the other polygon. If those bounding boxes, too, overlapped, the D-function was used to see whether the two line segments intersected. Lastly, we selected 25,000 polygon pairs that did overlap with each other and 25,000 polygon pairs that did not overlap with each other. Each was given a label 1 or 0, respectively, and saved to a text file.

4.5.2 Network Structure

We used an input layer of 16 neurons for the neural network, 3 hidden layers with 128, 256, and 256 neurons, respectively, and an output layer with only 1 neuron. This output layer gave back a binary value whether the two polygons overlapped or not. Like in previous networks, we used the Softsign activation function for the hidden layers and the Sigmoid activation function for the output layer. A dropout layer was added after the second layer. The batch size was 256, and the learning rate 0.05. As optimizer, we used Stochastic Gradient Descent, and the loss function was BCELoss. As before, we let the network train for 1000 epochs but divided the learning rate by 5 if the training accuracy was not improving for 60 epochs in a row. If that happened 4 times, the training was cut short.

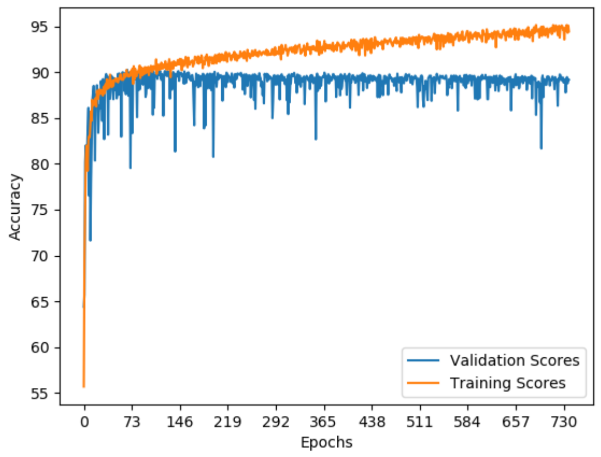


Figure 22: This figure shows the training and validation scores from the neural network that classifies whether complex polygons intersect or not.

4.5.3 Training and Results

The dataset was split into three parts before training: 80% went into a training set, 15% into a validation set, and 5% into a test set. The model's score was measured by looking at the percentage of correctly classified samples in the test set. The training results over time are shown in figure 22. The model started to overfit after about 100 epochs, and it was difficult to keep this from happening. When more dropout layers (a second dropout layer after the third hidden layer) was added, or more data was added (100,000 instead of 50,000), the validation scores went down. So we decided to use the original dataset of 50,000 and only a single dropout layer. After testing the network with different hyperparameters, the highest result we were able to produce was a final test score of 89.18%. This result, like before, was not accurate enough to be useful in general applications. These results caused us to change directions and look at a different part of C&P problems.

4.6 Conclusion

After conducting several experiments, the same problem kept appearing. It was our goal to program several neural networks to recognize and solve all different facets of C&P problems. One of those facets was the checking method to see whether a position is legal. To be generally applicable, a checking method needs to be 100% accurate. We hoped to create a network that would be 100% accurate and eventually faster than some existing methods because a neural network could look once at the entire board state while checking methods usually have to check each line segment or piece on the board. Sadly, accuracy was not high enough to equal, let alone surpass, classical algorithms commonly used as checking methods.

In this next phase of the thesis, rather than trying to replace classical algorithms entirely, we tried to supplement them. Classical checking methods are more precise than any network we were able to train, and so, going forward, we used those checking methods and instead focused our efforts on the second part of C&P problems: solution methods. We attempted to train a neural network that decides which piece goes where and in what order.

5 Solution Methods

After conducting several experiments on checking methods, we were never able to achieve perfect accuracy with neural networks. Checking methods that use classical algorithms can efficiently achieve perfect accuracy, so we decided to focus our attention elsewhere.

In the second part of this thesis, we have concentrated our efforts on solution methods. Checking methods might be the most time-intensive part of C&P problems, but solution methods are rarely able to achieve perfect results. Even in relatively simple problems, it can be challenging to find the most optimal solution [15]. So instead of trying to replace checking methods that already work quite well, we will spend the rest of this thesis trying to train a neural network to answer the more difficult question: How do you decide which piece goes where and in what order? To illustrate the complexity of the problem, let us think about how to pack a car for a holiday. Let us say 20 items need to be packed. There are some big bags, some playthings like balls and also some camping furniture. With just 20 items, there would be 2 quintillion different orders in which the items can be placed. Then we are not even talking about where in the car they should go.

Humans might know that it is smart to pack up the bigger items first because they can more easily fit smaller items in gaps between other items. Maybe the first items should be sturdy enough to carry the weight of the other items. Computers have a lot more trouble deciding what order is best. We hope that we can recreate that kind of insight in a solution method by training a neural network. Inspiration for how this neural network is designed comes from a chess engine called Giraffe which we will discuss very briefly. We will discuss the two experiments we conducted more extensively after that.

5.1 Giraffe

This section will briefly discuss the Giraffe chess engine since we have used the underlying idea of the chess engine to build our solution method. The core of Giraffe, build by Matthew Lai in 2015 [17], is a positional evaluator. Most chess engines have been built using expert knowledge, but Giraffe is not given any expert knowledge about what makes a good or a bad chess position. Instead, it solely relies on previously played games and later on playing against a version of itself. One of the strongest chess engines in the world, Stockfish 13, has been built by human experts who decided on hundreds of specific features to say whether a position is strong or weak. Things like material score, a bonus or penalty for each piece's position in relation to other pieces, and the mobility that each piece has. In contrast, Giraffe only looked at the board-state at random points in a game and trained a neural network to estimate the strength of that board-state. Doing this created a positional evaluator that was able to classify positions as strong or weak. After they built a program that could differentiate strong from weak positions, they built a Monte Carlo Tree Search algorithm (more on this later) which was able to check different moves going forward from a certain board-state in search of the best next move to make.

All of this made for a very strong chess engine, and the idea of looking at the board-state and learning whether it is strong or not inspired us to do something similar. We generated random game states and calculated the maximum achievable score from that game state. That data was given to a neural network which then learned what makes a strong and a weak board-state. To explain how we got to our positional evaluator, we start by explaining how we created the puzzles that would later be solved by our solution method. After that, we will talk about the Monte Carlo Tree Search (MCTS algorithm), and lastly, we will train two neural networks that will function as positional evaluators.

5.2 Generating Puzzles

To generate the data for the MCTS algorithm, a program was written using the Python programming language (version 3.6).

We took the size of the board and used it as the size of the first piece. We then cut that first piece into iteratively smaller random rectangle-shaped pieces. In a moment, we will explain exactly how we did that. The pieces that we ended up with were packed together again by our MCTS algorithm. The data generated from that was used to train a neural network. But we will describe everything step by step, starting with generating the pieces.

Several variables were defined at the start of the program. 'x_max' and 'y_max' signified the board's dimensions, 'pieces' signified the number of pieces desired, and 'samples' signified the number of puzzles that were generated. For each puzzle, a list of lists 'current pieces' was made. Each of the lists within 'current pieces' contained four values: The x and y coordinates which made up the bottom left corner of the piece, the height of the piece, and the width of the piece. At the start, the list of current pieces contained one piece. This piece was the size of the board and had values: [0,0,x_max, y_max].

Then, as long as the amount of desired pieces was higher than the number of pieces in the list 'current pieces', one of the three largest pieces (sorted by surface area) was randomly taken and removed from the 'current pieces' list. We used two different functions to split these chosen pieces. Which function would be used was decided by a random value that would, 66,6% of the time, choose the flower cut and otherwise the guillotine cut. More on those in a moment. This process was repeated until a satisfactory amount of pieces was produced. When that number was reached, the 'current pieces' list was added to a text file which saved all of the puzzles.

5.2.1 Guillotine Cut

The guillotine cut is the simplest way to cut a piece. It splits the chosen piece into two pieces by randomly deciding whether to cut the piece horizontally or vertically. After it decides in what direction a cut will be made, it will then take either the height (if cut horizontally) or the width (if cut vertically) and take a random value between 1 and the width or height -1. The piece will be cut along the chosen line, creating two new pieces that are then added to the list of current pieces. A piece can not be cut vertically if it has a width smaller than 2 and can not be cut horizontally if it has a height smaller than 2. If that is the case, the piece will be added back to the list of current pieces and a new piece will be chosen.

5.2.2 Flower cut

The flower cut is the more complicated of the two. It will take the chosen piece and cut it into five pieces. If there are at least 5 more pieces necessary to get to the desired amount, a flower cut will be made. An example of a piece being cut into five pieces using the flower cut can be seen in figure 23. If 5 new pieces would end up as more than the desired amount, a guillotine cut will be performed instead.

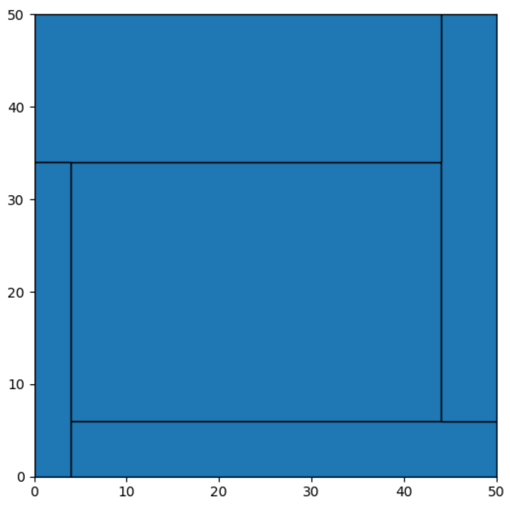


Figure 23: An example of a single piece being cut into 5 using the flower cut.

The flowercut takes several steps to perform. First, a centerpiece is formed by randomly selecting 2 values between 0 and the length of the chosen piece (the x values) and 2 values between 0 and the height of the chosen piece (the y values). These values can be no smaller than 1 and no larger than the length/width of the piece minus one. For the two x and two y values, the smallest of both x and y values becomes the bottom left point of the centerpiece, and the largest of the x and y values becomes the upper right corner. Then, from each corner of the centerpiece, starting at the bottom left and moving in a clockwise direction, a line is drawn vertically, horizontally, vertically, and horizontally again from the corner to the edge of the chosen piece. This creates 4 new pieces. All five pieces, the centerpiece and the four surrounding pieces, are added to the ‘current pieces’ list.

Using the methods described above, we generated 10,000 puzzles. The piece distribution of those puzzles is shown in figure 24. The distribution of pieces will later become relevant. We can now take the puzzles and try to put them back together using a Monte Carlo Tree Search algorithm. This algorithm will allow us to use the puzzles we just generated and create a dataset that we can use to train our neural networks.

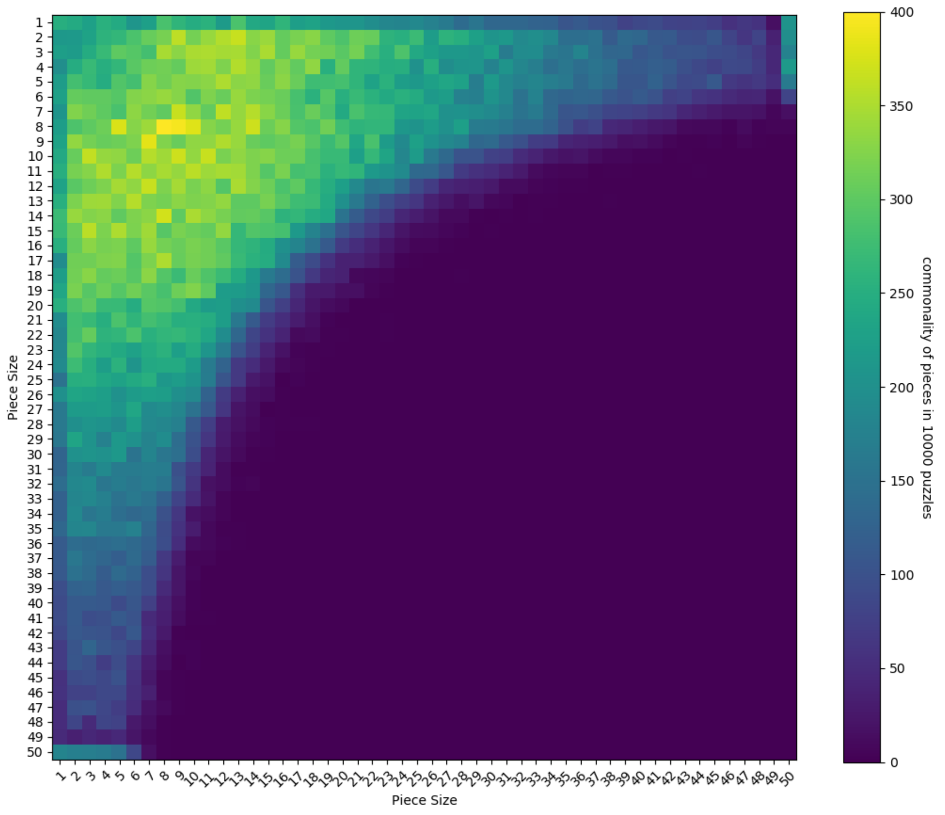


Figure 24: This figure shows the distribution of pieces in regards to piece size.

5.3 Monte Carlo Tree Search

The Monte Carlo Tree Search (MCTS) algorithm is an algorithm that tries to solve decision problems and can be used in games with a finite amount of moves [6]. The MCTS algorithm looks semi-randomly at many different action sequences from the current game state to the end of the game. It compiles those scores to estimate which action would most likely end in a positive result.

In its simplest form, an MCTS is a treelike structure where every node on the tree represents a different game state. Each of those game states has two extra pieces of information. How often that node has been visited, and the total score is that the node has achieved (more on that in a second). The MCTS goes through four phases:

1. Selection: The search tree is traveled from the root node onto a leaf node until a node is encountered, which is not a part of the tree. Nodes are selected by balancing exploration and exploitation. Commonly a method called 'Epsilon Greedy' is used wherein 95% of cases the node with the highest average score will be selected (dividing the total score that node reached divided by the amount of times it has been visited) and in 5% of the cases a random node will be selected.

2. Expansion: The selected node is added to the search tree.

3. Simulation: Random moves are made from the selected node until a terminal state is reached. These are also refered to as rollouts.

4. Backpropagation: Calculate the score for that terminal state and backpropagate it upwards through every node it has traversed, adding 1 to the number of times that those nodes have been visited and adding the score-value to the total score that the node has achieved in the past.

An example of this can be seen in figure 25. After a predetermined amount of simulations have been run, the move with the highest average score is made by the program. This move will cause a new current game state which will serve as the new root node. The whole process then repeats until the current game state is a terminal state. Described above is the most basic form of the MCTS, but every problem requires small adjustments to let the MCTS run as efficiently as possible. In the next section, we will describe some strategies we implemented to let our MCTS solve the previously generated puzzles as quickly and as accurately (high density, little empty space) as possible.

5.3.1 Our MCTS

Using the most general MCTS (as described in the previous section) would be extremely slow on any specific problem. Therefore, we made some changes and improvements to the MCTS to make it more efficient for the specific problem we are trying to solve. For example, for our project, it is important to bring the branching factor down as much as possible. The branching factor is the

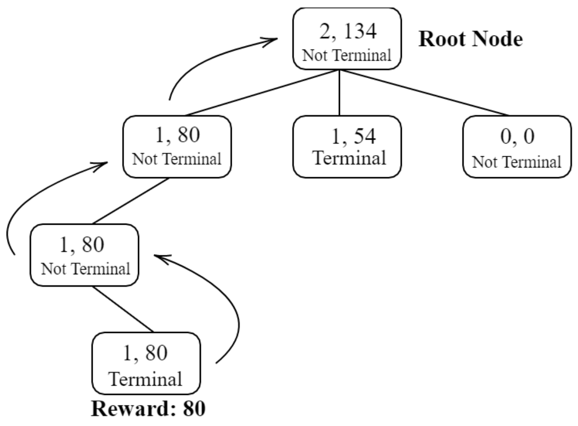


Figure 25: An example of the MCTS where the reward is propagated back through the path of nodes it has taken.

number of different moves that are legal from a current game state. The fewer branches, the faster and further the algorithm can go down each one to find out which move sequence would be the best. Even considering that we used integer coordinates to place our pieces, the fact is that if each piece can be placed in any spot on the board, there are 50,000 possible board states for the first move. This way, even solving a single puzzle would take nearly an hour, so we had to bring the branching factor down as far as possible.

This brings us to the most important change that we made. Rather than allowing the MCTS to check every piece fit and whether it fits in every place on the board, the algorithm was only looked in the bottom left corner. We know that all the pieces in each puzzle should be able to form a perfect square with no empty space. That means that at least one piece fits in the bottom left corner of the board. After that, another piece needs to be placed on the right side against that first piece, and so on.

Speed and accuracy were very important. The puzzles that this MCTS algorithm solved would be used to train our positional evaluator network. The more efficient our MCTS ran, the more data we had for our neural network, and the higher the accuracy of each puzzle solved, the more reliable the input data for our network. While the Bottom Left placement strategy was the most impactful on our MCTS, some smaller changes were made to increase the efficiency and accuracy of the algorithm.

5.3.2 Initial Placement

Initial Placement is a strategy where, if a piece has the width or the length of the entire board, it is placed before the first turn of the MCTS. In that case, the original root node of the MCTS would not be an empty board but would already have that piece on it. If, after placing that first piece, another piece existed with a length or width equal to the board minus the previously placed piece, that piece was placed as well. This procedure was repeated until there were no eligible pieces left. A very unlikely example of this can be seen in Figure 26

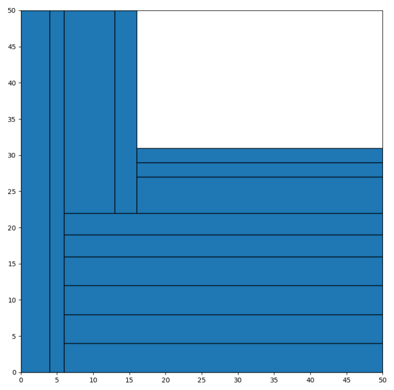


Figure 26: A very extreme example of the initial placement strategy in use.

5.3.3 Exhaustive Search

With only 8 pieces remaining, the branching factor is low enough that it is possible to look at every future board state. Looking at every possibility with 20 pieces left would take an unreasonably long time with our current hardware. So, as soon as there are only 8 pieces left, rather than doing more simulations, an exhaustive search was done by checking every possible position and choosing the best one as the final layout. This strategy was added to improve the final part of the solution.

5.3.4 Smart Frontier

Finding the bottom left corner in a square might sound straightforward, but is more complicated than that. Our solution to this was to build a 'frontier' that contained all points where new pieces could be placed. Any time a piece was placed, points were added to this frontier to represent the new places at which a new piece could be placed. Our frontier went through three main iterations before finding a version that was both accurate and fast enough not to hamper our MCTS.

The first iteration was a 'naive frontier' that just added a point on the bottom right corner of the newly placed piece and also on the top left corner, but only if the top left corner touched another piece or the side of the board. Otherwise, a frontier point was against the left border of the board, at the height equal to the top of the newly placed piece. The problem that arose with this frontier was that it would only create a new frontier point if two pieces created a corner, as shown in figure 27a. What was missing here was a frontier point that allowed a piece to slide over the rightmost piece against the tall piece, creating a small empty space, but utilizing the board more effectively.

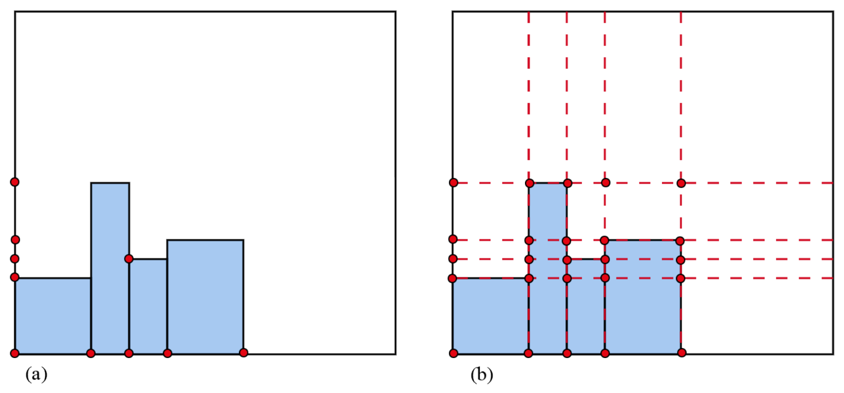


Figure 27: (a) The first iteration of our frontier method, the 'naive frontier', puts a frontier point at the bottom right and top left corner unless there is no second piece at the top left corner. In that case, a point is added on the board's border at the height of the placed piece. (b) The exhaustive frontier puts a frontier point at every crossing between the top coordinate and rightmost coordinate of each placed piece.

To ensure that a piece could slide over another piece until it touched another piece, we built a second iteration of our frontier named 'the exhaustive frontier'. Where this frontier placed its frontier points is visualized in 27b. Here you can see that we ensured that every possible position would be utilized by crossing every rightmost coordinate of every piece with every highest coordinate of every piece. Sorting these from bottom to top and from left to right ensured that the first available empty space would be found and utilized to place new pieces. The problem with this kind of frontier was that it grew exponentially and each time, the MCTS algorithm would check frontier points that had already been filled up. This caused the program to slow down more and more as more pieces were placed down. For each point, the MCTS needed to check whether apiece had already been placed there. If not, it needed to check whether any piece that had not been placed yet would fit in the open space. The problem starts to increase as more pieces are placed on the board. The representation in figure 27 does not show that frontier points will eventually end up inside pieces, which never need to be checked in the first place.

To prevent all this unnecessary checking, we created a 'smart frontier', the third and last iteration of our frontier method. The first frontier point is, like always, (0,0). When a piece is placed after that, the frontier point at which the

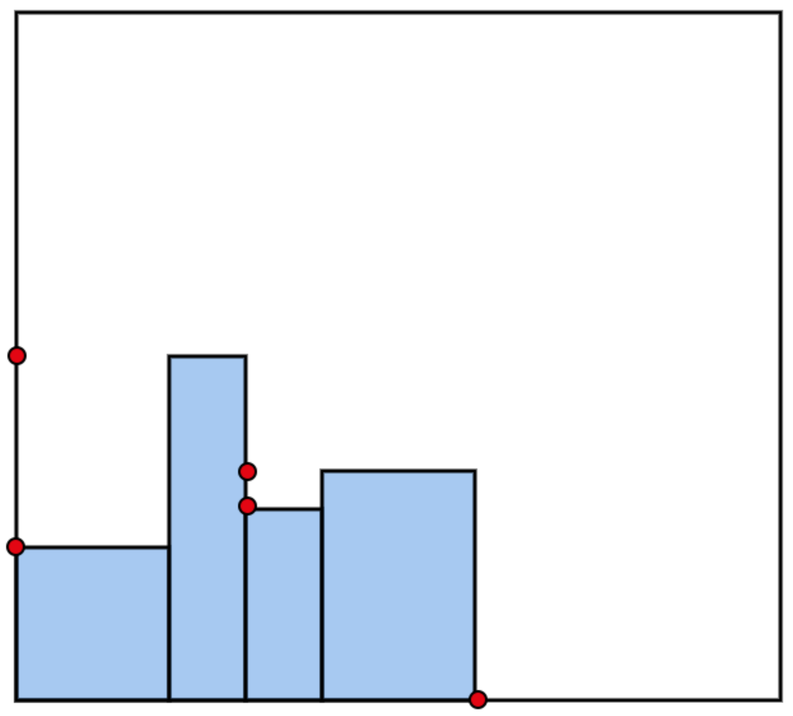


Figure 28: The Smart Frontier puts two frontier points for every piece it places while removing the frontier point at which the piece has been placed. A frontier point is added to the bottom right of the placed piece and against the right edge of the first taller piece it encounters on its left side. If no taller piece is encountered to the left side of the placed piece, a frontier point will be added to the board's border.

piece is placed (for example (0,0)), will be removed from the frontier, and two new frontier points will be added, one at the bottom right corner of the placed piece and another at the top left corner but only if the placed piece has a taller left side neighbor. Otherwise, the frontier point from the top left corner moves further left until it encounters a taller piece or the board's border. All this can be seen in figure 28. Lastly, if a newly placed piece covers one or more frontier points, those points will be moved to the right side of that newly placed piece.

These simple rules ensure that each possible location gets checked to see whether a new piece fits there without taking exponentially more time.

5.4 Using our MCTS for Data Generation

After implementing the improvements mentioned above, we tested our MCTS by making it solve the puzzles that were generated before. The MCTS is able to solve puzzles with an average final score of 88.34%, meaning that 88% of the board is covered by the end and 11.66% is wasted space. Each puzzle took an average of 159 seconds to solve. It takes this long because of the many simulations it had to perform before deciding on a next move. For every move, our MCTS performed a number of rollouts equal to 50 times the branching factor of that board state.

Spending 159 seconds on a single puzzle would be too slow to generate a sufficiently large dataset for a neural network to train. Fortunately, a neural network needs data representative of the task it performs. So, rather than solving puzzles from an empty board until a terminal state, we generated random board states. Our MCTS would then have to use that random board state as the starting point and solve the puzzle as best as possible. The final score from that puzzle would represent the best available solution achievable from the random starting board state. Taking the random board state together with the maximum achievable score would give us an idea of how 'good' board states were.

To generate random board states, we created a 'random start' function. This 'random start' function takes a 'counting' value which goes up by 0.6/number of total pieces any time a piece is placed (the pieces are still placed in the bottom left corner according to the smart frontier). Any time a piece is placed, a random value between 0 and 0.8 will be generated, and if the counter value is higher than the random value, the game state will be returned and used as the root node for the MCTS. The MCTS will then try to pack the remaining pieces as densely as possible on the partially filled board. An important note is that the data that this MCTS generated was not perfect. Of course, the randomly placed pieces from the random start function made it usually impossible for the MCTS to perfectly fill the rest of the board. However, even when faced with an empty board, the MCTS could not fit all 20 pieces perfectly onto the board. Like mentioned before, it was able to get to about an 88.34% score. Meaning that 88.34% of the board was filled, and the rest was left empty. If one were to place pieces completely randomly (although still using the Bottom Left strategy), it would only fill the board about 33.77%. Using the random start, the board got filled about 82.91%. It was partly our goal to find a puzzle that the MCTS would not be able to solve perfectly every time. The reason for this was because it would give us an easy benchmark to test our neural network against. If, when using a trained neural network, we would be able to achieve a higher score than 88.34%, we would have beaten the MCTS. If the MCTS could solve the puzzles perfectly, there would be no need for a neural network based solution method, like we saw before with neural network based checking methods.

That raised the question for us whether the generated data would be consistent on its own. The MCTS we implemented is semi-random because of the epsilon greedy algorithm we put in the selection process. A quick reminder, the epsilon greedy algorithm picks the average highest score 95% of the time while picking a random node to expand on 5% of the time. We wondered whether the epsilon of 0.05 would be able to reach a stable score or whether the MCTS would sometimes get stuck in the local optimum without finding the global optimum.

With this in mind, we solved 10 different puzzles 20 times in a row using 4 different epsilon values. Each puzzle was solved multiple times so that we would be able to see whether it would find the same solution or whether the final layouts would differ every time. The average score, standard deviation from that score, and epsilon value were saved and are depicted in figure 29. The highest score and the lowest standard deviations appear when an epsilon value of 0.7 is used. This means that in 30% of cases, the highest average score will be chosen, and in 70%, a random node will be chosen to explore. This ratio ensures the most consistent, highest scores while maintaining the lowest standard deviation and indicates that this ratio allows the MCTS to get stuck in a local optimum the least amount of times.

After ensuring that the data we generate is as consistent as possible, we used the 10,000 puzzles of 50 by 50 with 20 rectangle-shaped pieces that we previously generated.

Each puzzle would pass through the 'random start' function to create a random board state. That random board state and the pieces that were yet to be placed were given to the MCTS, which used the random board state as its root node. Each turn, the MCTS performed a number of rollouts equal to $50 * \#$ the branching factor. After the rollouts, the move with the highest average score was chosen as the new board state. This process repeated until there are only 8 pieces left, after which we used our exhaustive search function to test all possible layouts to find the best one.

Once a puzzle was solved to the best of our MCTS's ability, the board-state after the 'random start' function, the pieces that had not been placed after the 'random start function', and the final score of the puzzle, solved by the MCTS, were saved in a text file.

5.5 Our Positional Evaluator

The 10,000 solved and saved puzzles from the previous section were used as the dataset for the neural network that we trained as positional evaluator. The plan

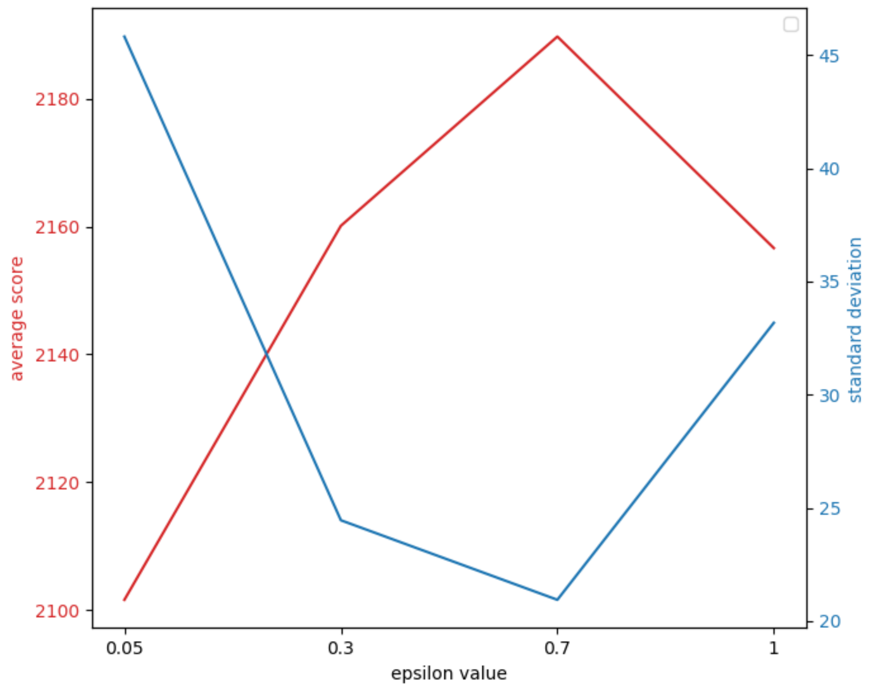


Figure 29: A graph that shows the consistency of the scores that the MCTS achieves when using different epsilon values for its epsilon greedy algorithm which decides the percentage of exploitation/exploration.

was to train the positional evaluator so that afterward, we could use it in an algorithm based on the MCTS. The difference with the MCTS was that this algorithm would only check one move ahead and evaluate those moves, where the MCTS checked until a terminal state.

We tested many different architectures for our neural network but settled on using an input layer of 5000 neurons. 2500 of those represented the current board-state of the 50 by 50 board, pixelated. The other 2500 neurons represented all the pieces that were still available to be placed. The first of those 2500 neurons would represent a 1x1 piece and the 2500th a 50x50 piece. Of course, the larger pieces did not appear in the dataset since there was no way that a 50x50 board could fit 20 pieces when one was already the full 50x50, so a part of those 2500 neurons was always set to 0. The distribution of pieces could be seen in figure 24.

We used three hidden layers that had 5000, 7000, and 9000 neurons, respectively. The output layer had only a single neuron since it only had to predict a single value that represented the network's guess about what percentage of the board would be filled up. We used the LeakyReLU activation function for the hidden layers. We did not use an activation function for the output layer but used the summed average input from the previous layer. The output would represent the network's estimation of what percentage of the board would be filled up by the end, looking at the current board-state and available pieces. The learning rate was way smaller than for the previous experiments at 0.00001. As optimizer, we used Adam with a small weight decay of 0.005. The loss function we used was MSELoss.

5.5.1 Training

As before, the data was split into three parts. 80%, 10%, and 10% for training, validating, and testing, respectively. The data was randomly shuffled before being divided. An important difference was that this network was not trained using the aforementioned Asus Zenbook. Instead, we used Google Colab and used the GPU option to to train the network using Colab's free-to-use Tesla K80 GPU's. This change sped up the training by about twenty-fold. Rather than an average of 58 seconds, one epoch only took about 2.5 seconds. The training lasted for 100 epochs, and during that time, the learning rate was not adjusted like it was in the previous experiments.

5.5.2 Results

The network's result was measured by the difference between the estimated final score of the board and the score that the MCTS achieved. This score was normalized between 0 and 1. Before training, the program misjudged the strength of a current board state by an average of 0.34. After training for 100 epochs, the error rate had decreased to an average error of 0.064. The average training and validation error has been visualized in figure 30.

As can be seen in the results, the validation scores stabilized after about 50

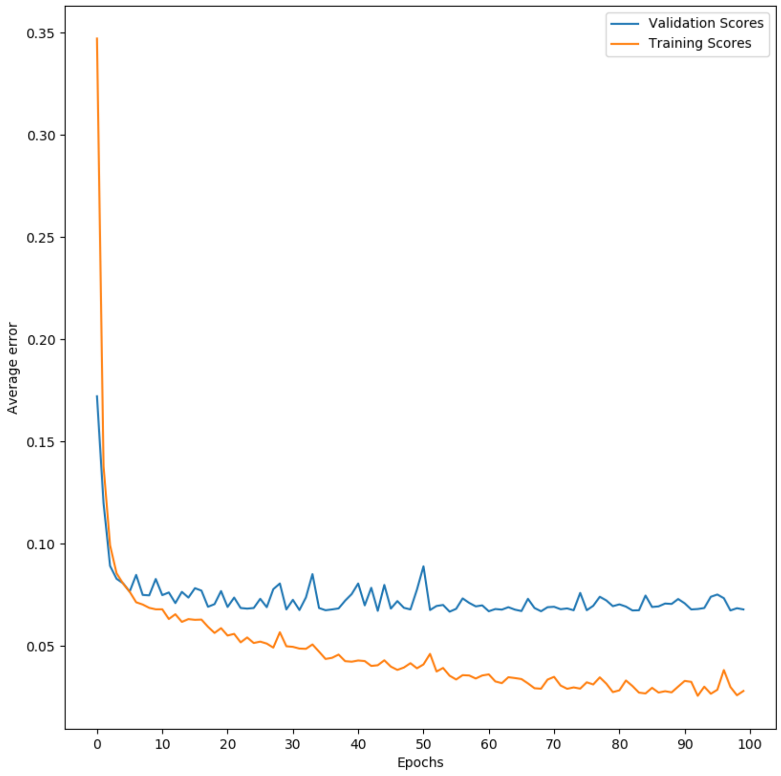


Figure 30: Training and validation error scores of the positional evaluator network over 100 epochs.

epochs and did not improve much after that point. The training scores continued to improve steadily, which means that the network started to overfit. This was possible caused by training the network on too little data, but due to the long time it took for the MCTS to solve single puzzles, we could not create sufficiently more data to improve the network. With more hardware, we hypothesize that it would be possible to improve the results from this experiment.

When manually going through results, it became evident that the network had devised certain strategies to pack a puzzle as densely as possible. For example, the network would give a higher evaluation to putting larger pieces early on the board. It also became clear that the network lacked a genuine understanding of what would make a good final layout. For example, two pieces of equal height would fit well next to each other, but the network rarely seemed to place pieces that way. Along with that, we noticed that the network would ascribe lots of possible layouts with virtually the same score. In practice, we hypothesize, the network might have a broad sense of how to pack pieces together but might not be fine-tuned enough to distinguish between a merely decent and good placement. These results might be caused by the imperfections in the dataset. The MCTS also placed larger pieces on the board first and was not always able to recognize that two pieces of the same height could be conveniently placed next to each other. These problems might be resolved by either creating a better MCTS that is able to pack puzzles perfectly, or simplifying the puzzles so that the MCTS can perfectly pack them without needing any improvements itself. As we discussed earlier, we have implemented this network in a tree search algorithm where it will evaluate all possible next moves before making the move that got the highest evaluation. Before talking about how this network performed as a solution method, we will discuss the training of another network that worked similar to this one. Although, rather than evaluating the strength of the current board state, we have devised a network that will try and predict what the next best piece is to be placed on the board. As output it will give what piece it thinks would be best to place next on the board.

5.6 What Piece to Place

The previous network we trained estimated the strength of the board state. For this last experiment, we trained a second neural network that would take the board state and yet to be placed pieces as input, and as output would give a single piece that should be placed next. As before, the piece size should be placed in the bottom left corner. Our goal with this network was twofold. First, we figured out how we could more easily generate data of a higher quality than we did for the previous network. Second, having a single piece as output would allow us to only pass the data through the network once, rather than classifying every possible board state. This should significantly speed up finding the next move to make.

5.6.1 Data Generation

We start with the same complete puzzles that were generated for the previous neural network. 10,000 boards of 50 by 50 with 20 pieces that fit perfectly on the board without empty space. Rather than taking the pieces off the board and trying to reconstruct the perfect layout with a MCTS, we took pieces from the board one by one. We started at the upper right corner this time. This made sure that the last piece taken away, would be the correct piece that needed to be placed next in the current layout if it were build from the bottom left corner up. In simple terms: If one piece is taken away from a complete puzzle, it is certain that that piece should go back in the only empty spot to complete the puzzle once more. This same idea can be applied when deconstructing one of our generated puzzles. The last piece taken away is the piece that should be placed back in the spot that it has been taken from. A small note on this data is that, while it is correct, it is not always the only correct way. There is a possibility that placing different pieces in one spot would still allow for a perfect final solution. That is the only thing that this data generation strategy does not take into account. Other than that, it is faster, easier and delivers more consistent data.

For our previous experiment we used a 'random start' function to simulate a random board state. For the data set using this new method, we took a random amount (between 1 and 19) of pieces away rather than randomly building up a board state. Pieces were taken away, starting at the top right corner and working our way to the bottom left corner. This meant that the layouts would look exactly the same as if they had been build by our MCTS algorithm. The last piece that was taken away, the current board state, and the pieces that had not yet been placed were saved to a textfile. In total, we generated 80,000 samples for our network to train.

5.6.2 Network structure

We used the same amount of hidden layers and neurons as we did in the previous experiment. We tried different network designs, but this proved to be the most effective. The biggest difference was the output layer which did not consist of a single neuron but consisted of 2500 neurons. Each neuron represented a single piece size from the first neuron representing a 1x1 piece to the last neuron representing a 50x50 piece. The target piece would have a value of 1, while all the other piece sizes would have a value of 0. For the output function, we used a Softmax activation function. The loss function that was used was BCELoss. We did keep the Adam optimizer and the learning rate of 0.00001, but lowered the weight decay to 0.000001.

Two other additions were a masking layer that we added to the final layer of the network, and a signal boost that we added to the data. The signal boost was implemented to create stronger neural pathways during training. A possible difficulty was that since only one out of 2500 neurons was activated, the signal would be too weak, and the network would not learn. To prevent this, we increased the value of the pieces that were one size smaller and larger than the target piece. For example, if the target piece was size 12x8, we would turn the neurons representing piece size 11x7, 11x8, 11x9, 12x7, 12x9, 13x7, 13x8 and 13x9 from 0 to 1. The target piece itself would also become one higher and its value would be boosted from 1 to 2. After the activation function, we added a masking layer to prevent these boosted neurons from being the highest-scoring neurons in the output layer. This masking layer would set all scores of the output layer to 0, except the scores of the pieces that were available to be placed in the current board state.

5.6.3 Training and Results

As before, we split the data into three parts. A training set consisting of 80% of the data, a validation set consisting of 10% of the data, and a test set consisting of the remaining 10% of the data. The data was shuffled before being split into these three groups. We trained this network using Google Colab rather than our personal hardware, which substantially increased training speed. Using Google Colab, each epoch took 34.3 seconds on average. Using an Asus Zenbook UX305U (personal hardware), a single epoch took over 7 minutes. The network trained for 200 epochs, and we did not make changes to the in learning rate throughout that training. The accuracy and validation scores that were obtained during the training are shown in figure 31.

As can be seen in the results, the validation scores of this network converged at around 35% with the highest validation score being 36.23%. The training scores show that the network is heavily overfitting, but when we tried to combat this by adding a dropout layer, the validation scores decreased by about 8%. We also tried to add more training data, but it caused the training time to slow down substantially without any improvements in validation or test scores. At the end of the training, the test set revealed that the network chose the correct target piece 34.84% of the time. 68.52% of the time, the target piece was in the top three choices of the network.

When reviewing the network's mistakes, it was noticeable that this network too was more likely to favor the placement larger pieces over smaller ones. It seems unable to realize that two pieces of equal size would fit perfectly next to each other. The results are interesting, but it is still difficult for both this network and the previous one to differentiate between a decent layout and a good one.

5.7 Neural Network Assisted Tree Search

We have trained two neural networks to serve as a solution method for C&P problems. The first evaluates how good a current board state is. This network can look at different possible board states and decide which one will be able to achieve the highest result. The second neural network can look at a board state and the available pieces and decide which piece would best be placed next. We have implemented both of these neural networks into a tree search algorithm

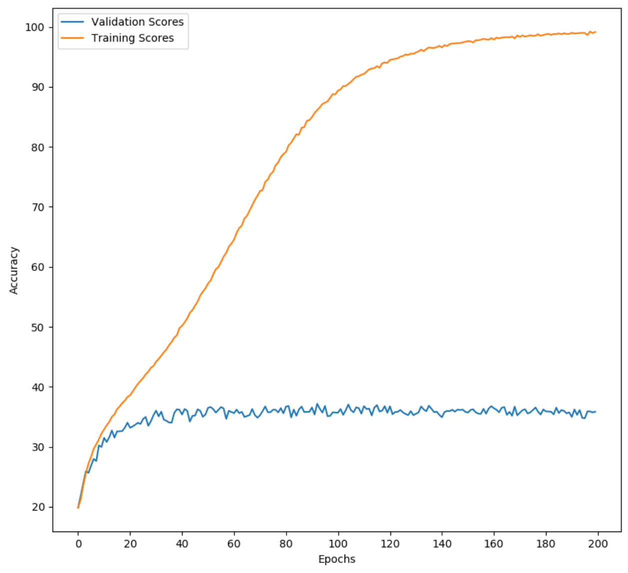


Figure 31: Training and validation accuracy scores of the network that decided what piece should be placed next.

to see whether they perform better than the previously implemented MCTS algorithm. The difference between these networks and the MCTS is that our neural networks only looked one move ahead, while the MCTS uses simulations to play the game multiple times until a terminal state.

We checked the bottom left corner of the current layout for the positional evaluator as per our smart frontier. If any pieces could legally be placed there, we would simulate possible board states by placing the legal pieces one by one in the bottom left corner and let the positional evaluator check how good that new board state was. The best of those board positions would be chosen and become the next current board state. This process would be repeated until there were no pieces that could legally be placed on the board. We tested this method by solving 1000 puzzles. The final coverage of the final puzzles was 68.6% which means that only 68.6% of the board was filled with pieces. This score is about 25% less than the MCTS was able to achieve. The upside of this method was that we did not need to do any simulations which caused each puzzle to be solved in 3.4 seconds on average. This means that the neural network is 47 times as fast as the MCTS.

We implemented the second neural network into a search tree algorithm as well. Like the previous algorithm, it would look at the current board state and all the available pieces. It would then decide what piece would be best to place next. That piece would then be put down, changing the board state. This process was repeated until there were no legal moves that could be made, either because the board was filled up or because there were no pieces that could legally be placed on the board. Since this approach only needed to look at a single board state rather than a number equal to the branching factor, this second network could solve puzzles in an average time of 2.6 seconds per puzzle. This was 61 times as fast as the MCTS algorithm. However, the scores were substantially lower, and this network could only cover 64.0% of the board.

6 Discussion

We set out at the start of this thesis to find checking and solution methods for C&P problems that incorporated neural networks. Throughout this project, we have made a lot of progress in achieving this goal. Despite those difficulties, which we will discuss in a moment, we have made some strong headway in several different aspects of C&P problems. In the results section, we will discuss our results in relation to the literature discussed throughout this thesis. After that, we will discuss our limitations, and finally, we will talk about some concluding thoughts on the project.

6.1 Results

Bennell et al. (2009) [3] describes the fundamentals of C&P problems and how those have been solved in the past. Our goal was to take the methods from that paper and translate them into neural networks. C&P problems consist of two main parts, according to Bennell, checking methods and solution methods. We attempted to solve both of those with neural networks. We hoped that replacing checking methods with a neural network would improve the speed at which C&P problems could be solved since the checking methods are the most time-intensive part of C&P problems [15]. While this might be the most time-intensive part, the precision of classical algorithms could not be beaten by our neural networks. The accuracy of checking methods (to make sure that none of the pieces overlap with each other or cross the board's boundary) is vitally important to solve C&P problems successfully.

After five different experiments, we were not able to achieve the precision that classical algorithms can achieve. We have tried to tweak our networks in dozens of small ways, using different architectures, hyper-parameters, and datasets. After all this, we feel fairly confident that, without significant improvements to the hardware used or significant changes to the neural networks used, checking methods using classical algorithms are not replaceable by checking methods purely based on neural networks. This is because some of the experiments we conducted focused on accuracy by doing tasks that were simpler than actual C&P problems. Since we could not outperform classical algorithms on a simple task like the point inclusion experiment, we feel that creating a neural network based checking method will be a complicated task with a low chance of success.

The second set of experiments we conducted was about solution methods. The solution methods described by Bennell et al. (2009) and Dowsland (1995) [9, 3] are all based on heuristics. Whether that's placing pieces from large to small, placing pieces to create the smallest increase in the bounding box of the composite piece, or placing pieces simply in the bottom left corner. All of these methods are rule-based and inflexible. Neural networks can be more flexible, making different kinds of decisions based on current information. In this aspect of C&P problems, we have made more progress towards an actual improvement on classical algorithms. A similar progress can be seen in chess, where neural networks have started to outperform classical algorithms [17]. The ability of neural networks to deviate from expert knowledge can create an advantage in chess, and we expected it would be similar in C&P problems. As of now, our neural networks are not quite able to improve on the classical algorithms in terms of score (percentage of the board covered by pieces). Currently, our best network is about 25% worse than our MCTS. However, our neural network can solve puzzles 61 times faster than the MCTS can. We expect that we can use that increased speed and exchange it for higher scores by increasing the depth limit or increasing the complexity of the positional evaluator. We will be making some suggestions on future research possibilities in the next section. What this all means is that we have found a way forward for neural networks in C&P problems.

6.2 Limitations and future work

The three biggest limitations to this process were mainly practical limitations. The biggest obstacle we encountered was the fact that there is only a limited amount of time that can be spent on a project like this. There are always more ideas and questions that can be answered. Different algorithms to implement, network architectures to try, and yet more hyperparameters to tweak. There are only so many hours that are allotted for a project like this, and we have long since surpassed that number.

The reason that time has been such an obstacle is because to our knowledge, only one study has been conducted that tries to use neural networks in order to improve on current solution methods for C&P problems [14]. The big difference between our project is that they worked with a three-dimensional C&P problem while we have been focusing our attention on two-dimensional problems. What this means is that we had to start with very little information of what would and would not work. This lack of knowledge resulted in us taking several avenues of inquiry that proved to be dead ends. One example of these dead ends are the first five experiments where we tried to replace classical checking methods with neural networks. These dead ends sometimes took weeks or even months of work which severely limited the project's scope.

These limitations have been difficult for us but do not necessarily have to impose problems for future researchers. We hope that this project can help people find an avenue of inquiry that allows them to find a promising approach to using neural networks to solve C&P problems. Our last neural network, where we generated consistent data by deconstructing puzzles, was in our minds the first experiment that had a chance to improve on classic solution methods. If someone were to continue this line of research, we would recommend looking into generating datasets that are much larger than the one we used. This will only be possible if the researcher has hardware available to train a neural network with a dataset of that size.

Another interesting idea might be to change the way that the to-be-placed pieces are represented. Currently, all the pieces have been represented by a binary matrix with a size equal to the number of possible piece sizes. We feel that this representation creates a needle in a haystack situation for a neural network that has to choose 1 out of 2500 options, even though there is only a maximum of 20 pieces.

The last way forward in our eyes is to exchange speed for depth in the search tree. Currently, we used our neural network to check a single layer of possibilities within our MCTS, selecting the best option and using that as the next move. This increased the speed of the program by a factor of 61. Giving up some of that speed in favor of an increased depth limit might result in higher final scores.

6.3 Conclusion

In conclusion, this thesis has tried to find ways to solve C&P problems by using neural networks. We have found that checking methods are probably not a good candidate to be replaced by neural networks because of the need for high precision. However, our results indicate that there is a strong possibility that solution methods can be improved upon by using neural networks. More research is yet to be done, but the addition of neural networks might allow solution methods to move away from heuristics and towards a more flexible way of decision making. If future researchers can improve solution methods, this would be beneficial to many real-world industries and could decrease the cost and environmental damage caused by manufacturing processes.

References

[1] A Ramesh Babu and N Ramesh Babu. A generic approach for nesting of 2-d parts in 2-d sheets using genetic and heuristic algorithms. *Computer-Aided Design*, 33(12):879–891, 2001.

[2] Julia A Bennell, Kathryn A Dowsland, and William B Dowsland. The irregular cutting-stock problem—a new procedure for deriving the no-fit polygon. *Computers & Operations Research*, 28(3):271–287, 2001.

[3] Julia A Bennell and Jose F Oliveira. The geometry of nesting problems: A tutorial. *European journal of operational research*, 184(2):397–415, 2008.

[4] Julia A Bennell and José F Oliveira. A tutorial in irregular shape packing problems. *Journal of the Operational Research Society*, 60(sup1):S93–S105, 2009.

[5] J Błażewicz, P Hawryluk, and Rafal Walkowiak. Using a tabu search approach for solving the two-dimensional irregular cutting problem. *Annals of Operations Research*, 41(4):313–325, 1993.

[6] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[7] Edmund Burke, Robert Hellier, Graham Kendall, and Glenn Whitwell. A new bottom-left-fill heuristic algorithm for the two-dimensional irregular packing problem. *Operations Research*, 54(3):587–601, 2006.

[8] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein. *Introduction to algorithms*. MIT press, 2009.

[9] Kathryn A Dowsland and William B Dowsland. Solution approaches to irregular nesting problems. *European journal of operational research*, 84(3):506–521, 1995.

[10] Kathryn A Dowsland, Subodh Vaid, and William B Dowsland. An algorithm for polygon placement using a bottom-left strategy. *European Journal of Operational Research*, 141(2):371–381, 2002.

[11] Harald Dyckhoff. A typology of cutting and packing problems. *European Journal of Operational Research*, 44(2):145–159, 1990.

[12] Jens Egeblad, Benny K Nielsen, and Allan Odgaard. Fast neighborhood search for two-and three-dimensional nesting problems. *European Journal of Operational Research*, 183(3):1249–1266, 2007.

[13] A Miguel Gomes and José F Oliveira. A 2-exchange heuristic for nesting problems. *European Journal of Operational Research*, 141(2):359–370, 2002.

[14] Haoyuan Hu, Xiaodong Zhang, Xiaowei Yan, Longfei Wang, and Yinghui Xu. Solving a new 3d bin packing problem with deep reinforcement learning method. *arXiv preprint arXiv:1708.05930*, 2017.

[15] Donald R Jones. A fully general, exact algorithm for nesting irregular shapes. *Journal of Global Optimization*, 59(2-3):367–404, 2014.

[16] Karel Kolomaznik, M Adamek, I Andel, and M Uhlirova. Leather waste—potential threat to human health, and a new technology of its treatment. *Journal of Hazardous materials*, 160(2-3):514–520, 2008.

[17] Matthew Lai. Giraffe: Using deep reinforcement learning to play chess. *arXiv preprint arXiv:1509.01549*, 2015.

[18] Aline AS Leao, Franklina MB Toledo, José Fernando Oliveira, Maria Antónia Carravilla, and Ramón Alvarez-Valdés. Irregular packing problems: a review of mathematical models. *European Journal of Operational Research*, 2019.

[19] Der-Tsai Lee and Bruce J Schachter. Two algorithms for constructing a delaunay triangulation. *International Journal of Computer & Information Sciences*, 9(3):219–242, 1980.

[20] Anantharam Mahadevan. Optimization in computer-aided pattern packing (marking, envelopes). 1984.

[21] Duncan McCallum and David Avis. A linear algorithm for finding the convex hull of a simple polygon. *Information Processing Letters*, 9(5):201–206, 1979.

[22] Ilsang Ohn and Yongdai Kim. Smooth function approximation by deep neural networks with general activation functions. *Entropy*, 21(7):627, 2019.

[23] José F Oliveira, A Miguel Gomes, and J Soeiro Ferreira. Topos–a new constructive algorithm for nesting problems. *OR-Spektrum*, 22(2):263–284, 2000.

[24] José Fernando C Oliveira and José A Soeiro Ferreira. Algorithms for nesting problems. In *Applied simulated annealing*, pages 255–273. Springer, 1993.

[25] Nitish Srivastava. Improving neural networks with dropout. *University of Toronto*, 182(566):7, 2013.

[26] Yuri Stoyan, Aleksandr Pankratov, and Tatiana Romanova. Quasi-phi-functions and optimal packing of ellipses. *Journal of Global Optimization*, 65(2):283–307, 2016.

[27] Yurij Stoyan, Guntram Scheithauer, Nikolay Gil, and Tatiana Romanova. Phi-functions for complex 2d-objects. *Quarterly Journal of the Belgian, French and Italian Operations Research Societies*, 2(1):69–84, 2004.

[28] Gerhard Wäscher, Heike Haußner, and Holger Schumann. An improved typology of cutting and packing problems. *European journal of operational research*, 183(3):1109–1130, 2007.

[29] Glenn Whitwell. *Novel heuristic and metaheuristic approaches to cutting and packing*. PhD thesis, University of Nottingham, 2004.

[30] Bing Xu, Naiyan Wang, Tianqi Chen, and Mu Li. Empirical evaluation of rectified activations in convolutional network. *arXiv preprint arXiv:1505.00853*, 2015.