UNIVERSITÄT
LEIPZIG

Universität Leipzig
Fakultät für Mathematik und Informatik
Institut für Informatik
Abteilung Datenbanken

Model training of a simulated self-driving vehicle using an evolution-based neural network approach

Bachelor Thesis

submitted by:
Jonas König
Informatik Bachelor of Science

supervised by: Dr. Thomas Burghardt, MSc. Tobias Jagla

Contents

1. Introduction

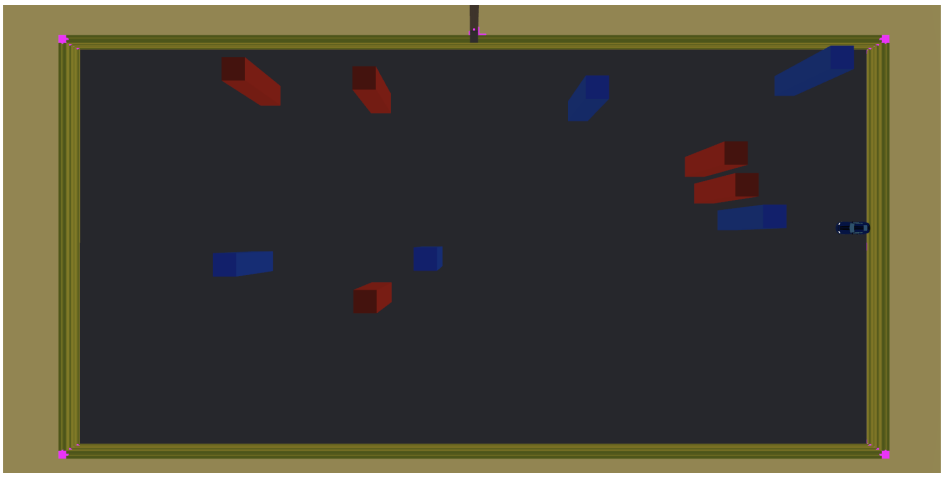
1.1. Motivation

The motivation for this Bachelor thesis comes from the desire to build a self-driving vehicle for the living lab at the SCADS. This vehicle should be able to maneuver the parkour of blocks from a start to a finish line. Because of the issues with training self-driving vehicles in the real world and the overall complexity of the task, which would include building the vehicle itself, this Bachelor thesis is about creating and training a neural network that can maneuver the self-driving vehicle around in a simulated environment.
Because of the future desire to transform the simulated car into an actual vehicle, the software has to be designed in such a way that the simulation can be replaced with real-world hardware.

1.2. Task

The task of the vehicle is to drive through a parkour, shown in fig. 1.1, of red and blue blocks scattered over the arena. These blocks indicate the driving left on red and right on blue. A time penalty is added if the vehicle fails to drive around the block the right way. The time from starting the parkour to finishing is stopped and added to the time penalty. The goal is to complete the course as fast as possible.

Figure 1.1.: Top view of the driving arena



1.3. Model training

Because training should be unsupervised, it will be done with an evolution-based training algorithm. This is necessary for solving this task because while the task is performed, it is not clear which move is right or wrong, so reinforcement learning is not an easy option.
While the evolution-based approach reduces complexity in the training algorithm, compared to reinforcement learning, the processing power needed is much higher.
The evolution-based neural network training approach is based on Darwin's theory of evolution. The algorithm is training the neural network by generation. In one generation are x amount of individuals. These individuals differ in parameters to control their behavior. All individuals of the generation have to perform the same task (meaning the exact positions of obstetrical). After one generation has completed the task, the individuals are ordered after the speed with which they have completed the task. This position then influences the passing on of their parameters to the next generation, which is randomly crossed with the parameters of other individuals to form the next generation.

1.4. Neural Network

The Deign of the neural network is essential for the success of model training. Neural Networks have to be designed in a way so that they are as small as possible while keeping up with the complexity of the task they are given.
This is challenging because estimating the complexity only from the task itself is impossible. Finding the right design and complexity of the neural network is also essential and will be discussed and tested throughout this thesis.

2. Related work

2.1. Self driving Vehicles

Training a self-driving car in a simulation is not new and has become more mainstream with more capable hardware. Most self-driving training is done with rule-based algorithms [1], restricting the possibility of learning directly from data and introducing costly hand-tuning. Most research on self-driving cars is done with the goal in mind to take part in public roads, resulting in a focus on rule building [2]. This work is concerned more about finding the optimal performance in the given driving task and not how this performance is archived. Meaning a manner that is unwanted on public grounds is not a concern. This means solving more complicated problems like following the lanes or dealing with lane changes, described by Mariusz Bojarski, are not a concern and will not be discussed [3].
Another problem with self-driving vehicles is processing input data, like lidar or images. These problems are often also a critical component for the design of the neural net and its features [2] [4]. Developing an image processing pipeline will not be part of the thesis to reduce the complexity further. Instead, the input data for the neural network is calculated with images mask.

2.2. Multi Layer Neural Network training

Uniting all these projects and research is the Multi-Layer Neural Network (MLNN), which generally consists of three layers: input, hidden, and output. Finding the proper weights and connections between these layers is the task of the training algorithm [5]. The backward propagation algorithm is commonly used for training MLNN, but these tend to get stuck at local optima and are sensitive to initial weights [6] [7]. To overcome this issue, in this thesis, a Population-based metaheuristic algorithm (PBMH) is used, which has been shown to overcome these problems [8] [9]. Evolution-based algorithms are a subset of PBMH and the most prominent evolutionary algorithm is the differential evolution algorithm (DE) [10]. DE has been found very efficient in finding optimal weights in MLNN. DE is based on mutation, crossover, and selection operators, where the mutation is responsible for generating a mutant individual based on scaled differences among individuals. Crossover combines the mutant individual with the parent one, and selection carries the better individuals to the next

population [7]. To further optimize these DE algorithms, [11] researched DE with multiple trial vectors and, more recently, [12] introduced a Quasi-Opposition DE algorithm, which returned promising data. Combining these [7] proposed a DE algorithm with a centroid-based strategy incorporating opposition-based (CenDE-DOBL). Tested against the UCI machine learning repository, the proposed algorithm outperforms all other tested algorithms. With these results, I decided to use CenDE-DOBL as the training algorithm.

2.3. Multi Layer Neural networks

Neural Networks are generally composed of three types of neurons: input, output, and hidden. Each neuron's weights and bias describe the flow through the network. A challenge faced with the given task is that the input is not static. There are three strategies for solving this. The first is to use a recurrent neural network. This neural network has a short time memory and can provide neural variable input sizes [13]. This approach has the downside that recurrent neural networks often need a special training algorithm like described in [14] and are generally more challenging to train. The second method uses null-filling to fill unused input neurons with null. This method results in overhead, as input neurons that are null/unused have to be calculated nevertheless. The third strategy is based on a [15] and [16], which proposed some preprocessor to reduce the input data to a known size. This preprocessor is static and repeats for the inputs. In this study, the preprocessor will also be a trainable neural network, which will evolve with the rest of the neural net.

A big influence on the performance of neural networks is the activation function. This function defines how and when the neuron frowards data in the neural network. [17] describes how different activation functions compare in certain task. [18] describes how deep learned neural networks can be improved by not using a static activation function instead of learning the activation function. While this is an exciting topic to research, for this thesis, a tanh is used as a static activation function because [19] and [17] indicate that tanh fits the task best.

3. General

3.1. Differential Evolution

Differential evolution is a method of optimizing a problem with a candidate solution. It has three leading operators: mutation, crossover, and selection. The mutation is done with the mutation Vector:

$$v_i = x_{r1} + F * (x_{r2} - x_{r3}) \tag{1}$$

where x_{r1}, x_{r2}, x_{r3} are distinct individuals, which are randomly selected from the input population. Crossover then combines the mutant, and the target vector [7]. A popular and used in this paper is the bionormal crossover:

$$u_{i,j} = \begin{cases} v_{i,j}, & rand(0,1) \leq \ CR \ or \ j == j_{rand} \\ x_{i,j} & otherwise \end{cases} \tag{2}$$

where $i = 1, ..., N_P, j = 1, ..., D,\ CR$ is the crossover rate, and j_{rand} is a random number in $[1; N_P]$ [7].
The selection operator then chooses the best individuals from their performance in the evaluation function.

3.2. Dynamic Quasi Opposite based learning

The initial population of an evolution-based strategy is generally constructed out of randomly generated individuals. It is then the task to find an optimum from this initial population. The opposite learning approach tries to amplify the rate with which the training algorithm performs by creating for every individual an opposite individual. This opposite individual can be described as the opposite guess in a defined range. In Theory, this can increase the guess rate by 50% and be shown in [20] to be superior over only using DE. An opposite number of x is described as:

$$\check{x_i} = a_i + b_i - x_i \tag{3}$$

with a_i, b_i beeing the upper und lower bounds of the search space [7].
Dynamic quasi opposite-based learning is a variant of opposite-based learning, which employs

quasi opposite numbers [20]. It is dynamic since the individual's maximum and minimum numbers are used as upper and lower bounds to generate the quasi opposite individual. The quasi opposite number $\check{x}$ is obtained by:

$$\check{x_i} = rand[\frac{a_i + b_i}{2}, a_i + b_i - x_i] \tag{4}$$

Where a_i, b_i are the upper and lower bound of the x_i rand $rand[n, m]$ being an uniform number between n, m.

3.3. Centroid based strategy

A centroid-based strategy is an optimization of a metaheuristic algorithm based on an individual's centroid. Its strategy is based on the Monte-Carlo simulation, which says the probability of an individuum being closer to an unknown solution is higher towards the center of the search space compared to random selection [7].
Inspired by [21], [7] implemented the centroid bases strategy as an individual created based on the N best individuals. The centroid based individual $\overrightarrow{x_{center}}$ is calculated as:

$$\overrightarrow{x_{center}} = \frac{\sum_{n=0}^{N} \overrightarrow{x_{bn}}}{N} \tag{5}$$

Where $\overrightarrow{x_{bn}}$ is the n best individual [7] [21].

4. Neural Network training with Evoulutionaray algorithms

4.1. Software setup

The neural network and the training algorithm are entirely separate from the Simulation itself because of the requirement that the neural network should be portable to another system. Because of this, the software to control, train and evaluate the evolution-based neural network training progress requires these different software parts to work together. These different parts are described here, and their execution is justified. All created code is available at https://github.com/jonaskonig/BachelorNN for the Python side and https://github.com/jonaskonig/carsim for the Unity side.

4.1.1. Neural Network

The self-driving vehicle's task is to drive through parkour that contains different colored obstacles to maneuver around. The blue obstacle indicates turning right, the red one turning left. A third color, yellow, is chosen to signal the border of the arena. Where and in which quantity these colors are visible in the picture should result in an acceleration and steering output.

To not further complicate the training and the neural network itself, recognizing where and how big the different colors are in the picture is not done by the neural network itself but rather by OpenCV. This has two benefits: The first is that confronting the vehicle with a never trained view that is entirely different from what it has trained on while in the simulator would most likely result in abysmal performance and vanish the opportunity to port the neural network. In the real world, a pre-trained neural network could be used to reduce the camera view into an array of obstacle positions. The second benefit is that there should be less neural network training with reduced complexity.

While OpenCV solves many problems for this task, it introduces one problem. With different camera angles, the count of different obstacles in the frame changes, and with it, the size of the input data for the neural network changes too. This is problematic because a traditionally neural network has a fixed count of inputs. Dealing with nonfixed

inputs in a neural network, in general, requires one of these two options: The first is to make the input of the neural network extensive enough so that the input size is never bigger than the available inputs. This is called null filling, and while resolving the problems, it introduces unwanted complexity because most of the inputs are not used but have to be taken into account in the feed-forward algorithm, which forces the design to never accepts more inputs than initially planned. The second option is a recurrent neural network (RNN). These can take variable inputs because of added memory. The RNNs come with the downside of being complex to integrate and very hard to train with deep learning algorithms.

Figure 4.1.: Input neural network

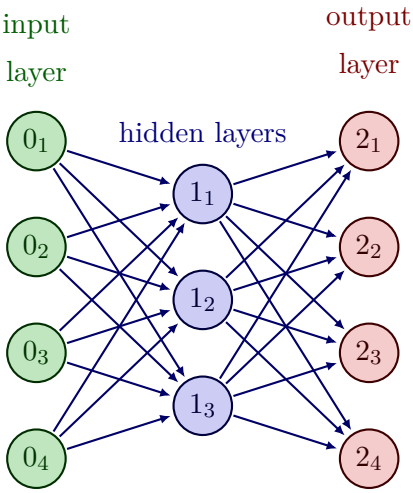
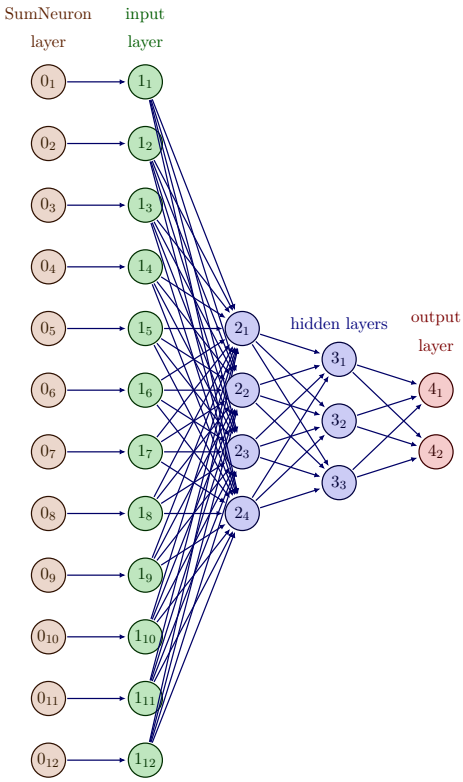


Figure 4.2.: Primary neural network



Because these solutions are not suitable for the task, a third option is proposed. The Neural network is broken up into two parts.

The first part is being able to process the input data. This part is a neural network and is unique once per color. So there is an input neural network for the blue blocks, one for the red, and one for the yellow border. These can then be reused for the count of the input. For example, suppose there are four blue, three red, and one yellow obstacle. In that case, four identical blue-input-neural networks are used to represent the blue blocks, three identical red input-neural networks are used for the red blocks, and one yellow-input-NN is used for the yellow border. The output of the input neural network is fed in groups by their color into the first layer of the primary neural network, shown in fig. 4.2, which is constructed of sum neurons. The task of the sum neuron is to add up the out-

puts of the different blocks by their color to form the input for the primary neural network. All the output data of the input neural networks that represent the blue, red, and yellow obstacles respectively get added together. Therefore, the primary neural network always has the same 12 inputs if the input neural network's output size is 4. The input neural networks handle the first interpretation of the data while not creating overhead as only so many of them are present as necessary. The primary neural network's task is to reduce the input to the steering and acceleration output.

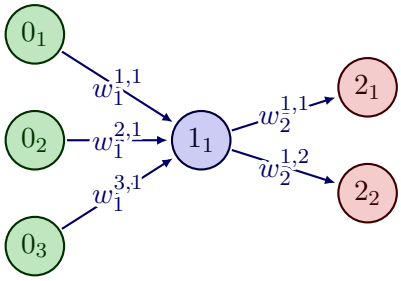
Both neural networks operate on the traditional feed-forward approach. It has been extended to first calculate the inputs for the main neural net with the input neural networks.

Another essential component of a neural net is the activation function. We want the output to be two values that are zero centered and between two known values, preferable between -1 and 1. As the steering operates between -180-180 and the acceleration is limited in both directions (negative acceleration resulting in breaking/ driving backward). As tanh fulfills these demands, it is used as an activation function.

4.1.1.1. Neural Network encoding

An encoding strategy is needed because the neural network must be stored as an array of floating-point numbers. As a base of this encoding strategy, [7] is used and extended to take the division into input neural network and primary neural network into account. A Neural Network is encoded by stringing together all of the neural network weights and biases, as shown in fig. 4.3. Describing the form of the Neural network is an Integer array. This is needed for calculating the offsets, where the weights and biases are located for each neural network. For example, an array [4,3,4] describes a Neural Network with four inputs, 3 Hidden, and four output layers as shown in fig. 4.1. While [12,4,3,2] would result in fig. 4.2 (The Sum Neuron are not encoded, as their position is always the same, and they do not have configurable data).

Figure 4.3.: Encoded neural network



$w_1^{1,1}$	$w_1^{1,2}$	$w_1^{1,3}$	$w_2^{1,1}$	$w_2^{1,2}$	$b(1_1)$	$b(2_1)$	$b(2_2)$

with b(x) is bias from neuron x, and $w_1^{1,1}$ being the weight between the first neuron of the first layer and the first neuron of the second layer

These encoded neural nets are then strung together to form the complete en-

coded neural network. For every neural network the describer is inside an array so [[4,3,4],[4,3,4],[4,3,4],[12,4,3,2]] indicates four fig. 4.1 feeding into fig. 4.2. The describer needs to have the same chronology as the encoded neural network, meaning, for this example, the weights and bias of the three input neural network followed by the weights and bias from the primary neural network. It has to be noted that the total output neuron count of the input neural network can not exceed the input count of the primary neural network.

4.1.2. OpenCV Object recognition pipeline

As described before, OpenCV is used to find the positions of the desired objects in the picture stream. To find these objects, Firstly OpenCV has to convert the image into the HSV(hue, saturation, lightness) color space. Lower and upper bounds for the different object colores are then defined. OpenCV can find these lower and upper bounds in the picture, creating image masks. Image noise gets introduced into the mask if these bounds' range is too broad. If the range is too tight, some blocks might fail to be recognized.

The image mask holds a set of pixels that fall into one color range. If the pixels are connected, they get grouped as one object. If a group of connected pixels then exceed a threshold in size, as the method is not perfect and image artifacts can get recognized, an array is created in which the upper left pixel and the width and height get saved. The whole process is shown in fig. 4.4. While this method is prone to failure when the lightning or the block color changes in a real-world scenario, testing is done inside the simulation, where all of these factors are controlled, and it has proven to work very well.

4.1.3. Evolution based Neural Network training

The used Training algorithm, CenDE-DOBL, was proposed in [7] and combined the benefits of Centroid (CEN), Differential-Evolution (DE), and Dynamic Opposite Based Learning (DOBL) together to form a better-performing algorithm. The different strategies in the algorithm pursue different goals. The DOBL part of the algorithm is present in the algorithm to increase the exploration. The DE parts act like a well-established base and create the mutated crossover individuals. This results in a more diverse set of individuals, resulting in better performance than if only used DE. The introduced diversity by DOBL results in an artificially bigger population, where inbreeding is less common, and the algorithm is not prone to get stuck at local maxima. The Centroid approach increases the exploitation by creating

individuals out of the n best individuals that should in theorie, perform better. Not both strategies are called in every evolution, but the jumping rate controls their distribution. The complete algorithm can be seen in pseudocode in algorithms 1 and 2.

Algorithm 1: OBL(D, N_p, Pop, L, U)

Data: D: Dimension of individual N_p: Population size, Pop: initial Population, L, U: lower and upper bound.

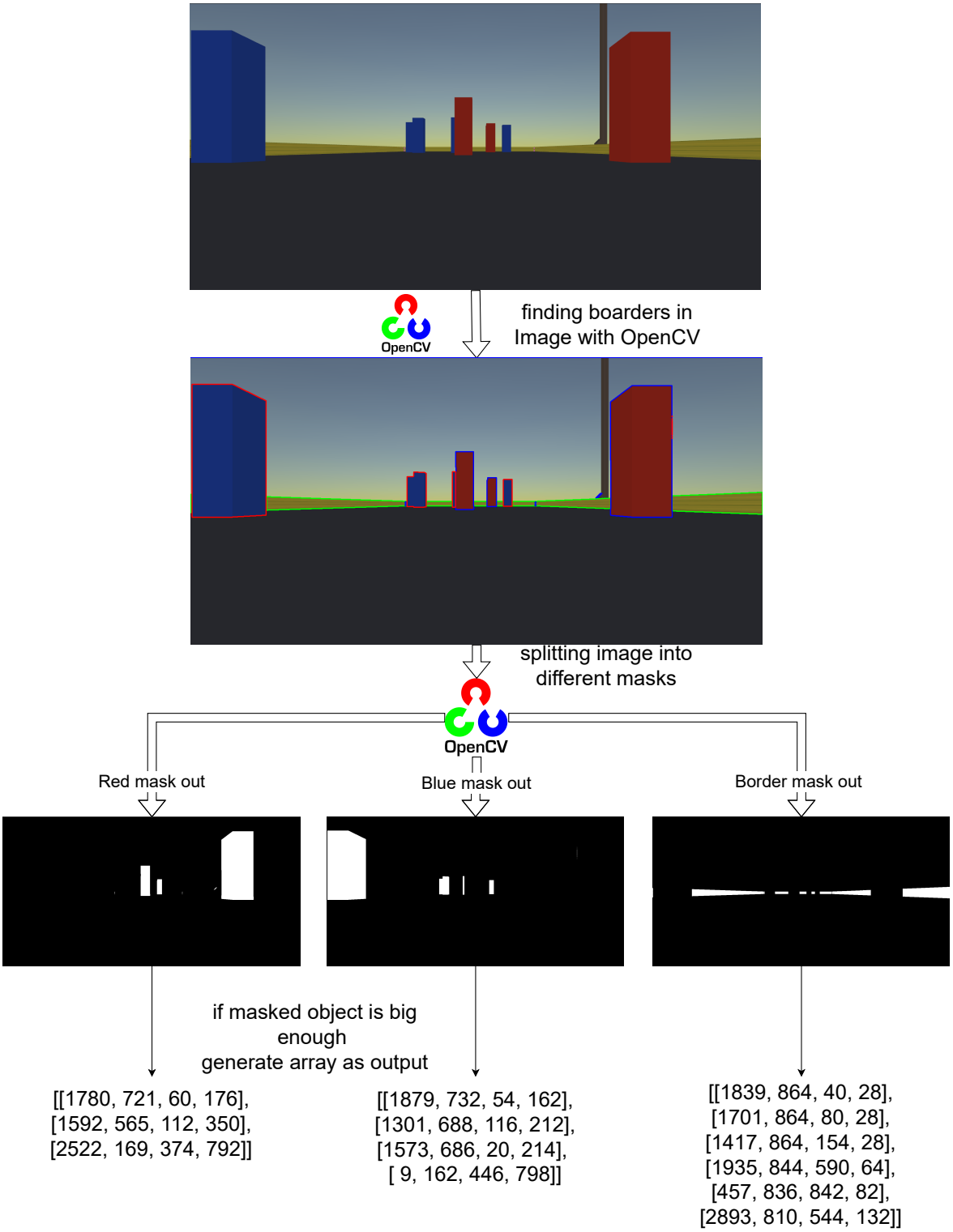
for *i from 0 to* N_p **do**
  for *j from 0 to* D **do**
    $OPop(i,j)) = rand[\frac{L_{i,j} + U_{i,j}}{2}, L_{i,j} + U_{i,j} - Pop(i,j)]$
$Pop \longleftarrow Select\ Np\ best\ individuals\ form\ \{Pop, OPop\}\ with\ evaluationfunction$

Algorithm 2: CenDE-DOBL$(D, N_p, j_r, N, C_R, L, U)$

Data: D: Dimension of individual N_p: Population size, j_r: jumpingrate, C_R: crossover rate, L, U: lower and upper bound.

$Create\ inital\ population\ with\ upper\ and\ lower\ bounds$
$Call\ OBL(D, N_p, PopL, U)$
while *not optimal solution* **do**
  $x_{r1}, x_{r1} \longleftarrow$ *two random individuals out of Pop with* $x_{r1} \neq x_{r1}$
  $x_{rb} \longleftarrow$ *best individual of Pop*
  for *i from 0 to* N_p **do**
    $v_i = Pop_i + F * (x_b - Pop_i) + F * (x_{r1} - x_{r2})$
    for *j from 0 to* D **do**
      if $rand[0,1] < C_R$ *or* $j == j_r$ **then**
        $u_{i,j} = v_{i,j}$
      else
        $u_{i,j} = Pop_{i,j}$
  $calculate\ performance\ of\ all\ individuals\ with\ evaluation\ functiom$
  for *i from 0 to* N_p **do**
    if $perf(u_i) > perf(x_i)$ **then**
      ; // $perf(x)$ `being the performance of` x
      $Pop_i = u_i$
  if $rand[0,1] < j_r$ **then**
    $L_n, U_n \longleftarrow$ *new Upper and Lower values calculated from individuals*
    $Call\ OBL(D, N_p, Pop, L_n, U_n)$
  else
    $\overrightarrow{x_{center}} = \frac{\sum_{n=0}^{N} \overrightarrow{x_{bn}}}{N}$
    $insert\ \overrightarrow{x_{center}}\ in\ Pop\ sort\ and\ reduce\ Pop\ to\ N_p$

Figure 4.4.: OpenCV Object recognition pipeline



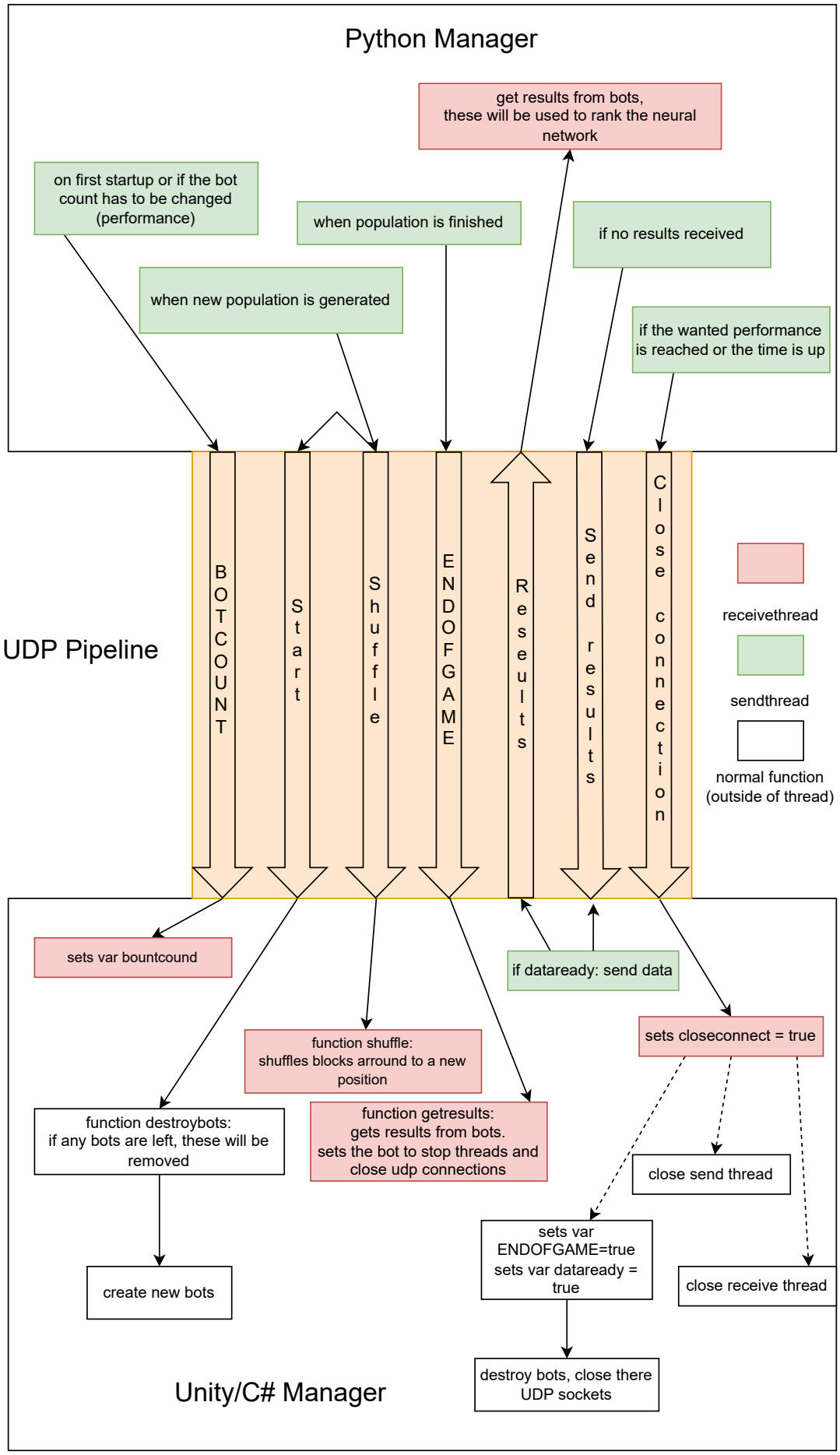
4.1.4. Communication between Python and C# / Unity

As it is unwanted that the neural network is dependent on the simulation, they have split apart. The Python back end handles all object recognition with OpenCV, the feed-forward algorithm, the neural network, and the Evolutionary learning algorithm. In contrast, Unity handles all of the simulations. While this setup is very convenient for later porting the code to a different platform, as Python is known to run anywhere, it requires a connection between Python and Unity to exchange data. There is no well-established way to run Python functions from C# / Unity; a UDP connection is used. A UDP connection is superior for this application, as it has much less overhead, and if some packages get lost, it is not as important. Other socket connectoions where not used as the UDP prodtocol is easily implemantable with Python and C#.

4.1.4.1. Connection Manager

With the optimization task, the connection manager represents the always present connection between Python and C#. It provides the possibility for Python to start, stop the game, change the count of individuals that are simulated at the same time, and create or load a new set of obstacles throughout the course. After a game has been stopped, the manager will destroy all bots controlled by the neural network in the arena while getting their performance. All performances are then sent back to the Python side, where they can be evaluated. All commands, as seen in fig. 4.5 are tunneled through a UDP pipeline. All data is encoded in UTF-8 so both sides can decode and encode it into a known format. Both sides act asynchronously, meaning that when the Python side sends the signal to start, it can only assume that the command was received and performed. The only exception to this is the results pipeline because it happened that the Python manager missed the results. In this case, it can ask for the results again. All other commands have been shown to work fine and without any missed information. Efforts to further account for error correction resulted in more problems and slower performance.

Figure 4.5.: Block diagram of the connection manager

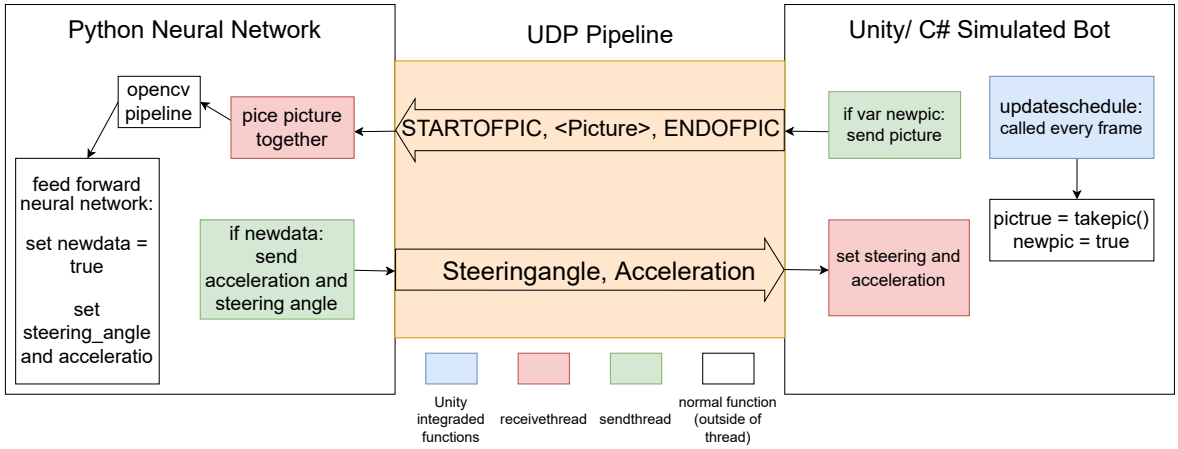


4.1.4.2. Neural Network Bot connection

For every individuals, a connection between the simulation side (Unity) and the Python side, where the image processing and the neural network feed-forward processing are done, has to be accomplished. This is done by the communication scheme shown in fig. 4.6. Both sides are created and know their corresponding send and receive ports. The Unity side then starts taking pictures, which happens five times per second as the framerate is fixed at 5 FPS. The captured picture is then stored in a byte array, and split into the max byte size that can be sent within one data frame. Before the picture starts, a start string is sent to Python to inform a new picture has started. This is followed by the bytes of the picture and ends with the top string. The Python side can now pice the picture back together and feed it into the OpenCV pipeline. The resulting output is then processed in the feed-forward algorithm. The neural network output is then strung together, converted to UTF-8, and sent back to the Unity side. Unity then uses this data to calculate the new position of the car.

This complete process happens independently from each other. This means Unity has no

Figure 4.6.: Block diagram of the communication between neural work and the Unity Bot



information on which captured picture corresponds to which steering commands. To not create a queue at the Python side, when the OpenCV and neural network pipeline is too slow, the buffer size, so the maximum queue length is set to two full packages. The oldest package is dropped when a new one comes in. Combined with the tactic of only putting the picture back together when all the date from the start- to the end string is received should prevent Unity from receiving outdated steering commands.

Testing has shown that this works fine. Some frames are dropped due to out-of-order or missing packages, but as the inputs on the Unity are not touched until new steering information

is received, it is not as crucial that every frame results in a steering output. If no new steering information is received, it is interpreted as if the current course is wanted to be continued.

4.2. Data acquisition

While training and evaluating the results by watching the simulator is possible, it is limited in determining the performance. For evaluating the data, the individuals that survived the selection get saved to a file for every generation. This data contains the performance of the individual in unit/ms, the encoded neural network of the individual, the parent of the neural net, the generation the neural network was generated, and the upper and lower bound. This data will allow analyzing the different individuals over time. The raw data generated by these tests can be found at https://github.com/jonaskonig/BachelorRawData. The tests are performed in the same arenas, meaning that every generation from every test sees the same parkour. This ensures that there are no unfair conditions. All individuals have 30 seconds to find a way through the parkour. If they manage to drive through the parkour faster, the time is stopped in advance. The distance between the start position and the end position, the maximum being the finish line, is then taken on one axis, as only the position at the y axis is relevant. This value then gets divided by time. In order to find the best combination of neural network complexity, design, and population size, the tests described in the following paragraph are performed.

4.2.1. Neural Network complexity

A more complex neural network can perform a more complex task. However, estimating the complexity is not accurate and possible. With more complex neural networks, the feed-forward algorithm, as it has to go over more iterations, gets more computationally costly; furthermore, the added complexity can harm the training performance as more data points have to be optimized. If the neural network size is not chosen right, it will result in lousy training performance with consistent or worse performance compared to a less complex neural network. If the neural network is chosen too small, the initial training performance is high, but the more complex neural network will overtake with more evolutions because it is better suited for the task.
For testing this an encoded neural network with the design [[4,3,4],[4,3,4],[4,3,4],[12,4,3,2]] is compared to a more complex neural network with the design [[4,3,4],[4,3,4],[4,3,4],[12,7,5,3,2]]

and a more simple neural network with the disign [[4,3,4],[4,3,4],[4,3,4],[12,4,2]]. These neural networks will train in the same conditions over 100 evolutions with a population size of 15.

4.2.2. Neural network design adaptations

The design of the neural network is also an essential part of the success of the training. It allows for shifting tasks to different parts of the neural network. For this task, we can modify the output size of the first neural network and the input size of the second neural network. This will force the input-neural network to reduce the initial input to a suggestion of direction and acceleration, while the primary neural network will only weigh the different inputs.
For testing this, an neural network with the design [[4,3,4],[4,3,4],[4,3,4],[12,4,3,2]] will be compared with a neural network [[4,3,2],[4,3,2],[4,3,2],[6,4,3,2]]. These neural networks will be trained over 100 evolutions in population size of 15 and will face the same tasks in every evolution.

4.2.3. Population size

As the population represents the genetic pool from which every new generation is formed, it has a significant influence on the performance of the training algorithm. The more individuals are in the genetic poll, the possibility for new mutations and crossovers between individuals who are not related to each other is higher. This, in general, results in a better solution to the problem but comes with two risks:
With a greater population size, the evaluation and sorting of the individuals get far more costly in terms of time and compute performance.
Another problem is that the complexity of the neural network has to scale with the population size as the possible solutions are also limited by the neural net. By choosing the population size greater than necessary, the crossover will, at some point, only generate solutions that were generated before.
For testing out an optimal population size, populations containing 15, 30, and 60 individuals are tested over 100 generations.

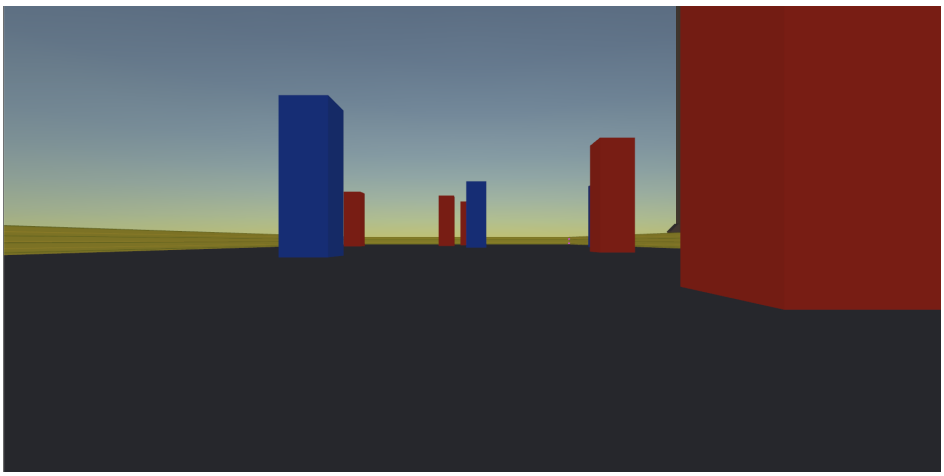
4.2.4. Comparaing CenDE-DOBL with DE

As the CenDE-DOBL algorithm is derived from DE, it is obvious to test their training performance against each other. For training, a population size of $N_p = 15$ and a training duration of 100 generations is chosen. This will show how useful the introduced mechanisms by CenDE-DOBL are in improving the overall training success.

4.2.5. Comparison against Human Pilot

As the best Neural Network resulting from the above test should be ranked in some form, a comparison against a human pilot is performed. Comparison against a static programmed algorithm would be preferable. Because such an algorithm would require substantial work with programming smoothing functions that allow not to generate abrupt steering and a function that can create steering output out of a picture, it is out of scope for this thesis. As a compromise, the human pilot is used. The human pilot will get instructions on the task and one training round which can be used to figure out steering and acceleration. The human pilot will only see the parkour and not anything else. The view out of the car can be seen in fig. 4.7. The car can be controlled by the four arrow keys, which represent acceleration, deceleration, steering left, and steering right. For every test, the driver can drive three times. There are three tests in three separate parkours to ensure that the resulting data is representable. The 100th generation is used for the neural network to perform the test. All individuals will perform all three tests and can then be compared to the human pilot. Comparing different training scenarios and algorithms, the $N_p = 15$, $N_p = 30$, $N_p = 60$ and the DE population are also tested.

Figure 4.7.: View out of the vehicle seen by the human pilot and the neural network



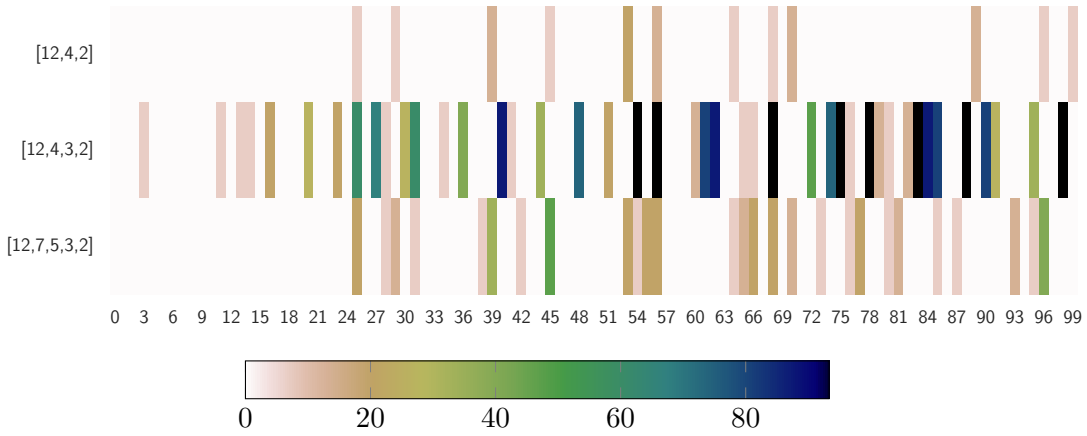
4.3. Data evaluations

There is something to keep in mind while evaluating the resulting data. The task is very dynamic, which means that the course can change a lot between runs, and it is not uncommon that the performance can take a hit. This generally does not mean that something went wrong but indicates a very challenging parkour. A Challenging parkour could be, for example, that the view of the car is blocked by two opticals standing together. This is considered ok as when humans place the blocks, this behavior is expected and somewhat wanted as the neural network has to withstand blocks placed by humans.

4.3.1. Neural Network complexity

As discussed before, the complexity of the neural network design is crucial for the success of the learning phase and can decide if it is successful or not. fig. 4.8 compares the introduced neural network structures. As can be seen, the neural network with minor complexity [12.4.3] is performing the worst, followed by the most complex neural network [12,7,5,3,2]. This indicates that the optimal complexity of the neural network lies in between these neural networks. Because the more complex neural network performed better, more to the complex side, as the [12,4,3,2] neural network is in between these neural networks, it can be assumed that the chosen complexity is well suited for this task. While there may be a far more optimal

Figure 4.8.: Comparison between individuals that completet the parkour in 30s or less



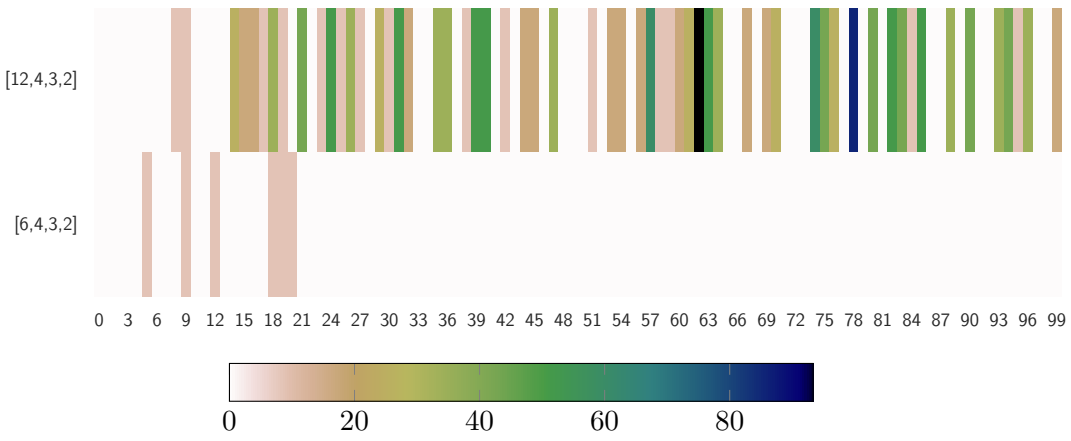
complexity factor for the neural network, testing all of them is impossible and would reach over the scope of this thesis. This data, however, shows that the neural network that is chosen

to perform most of the task and is used as a base in every other test is well suited for the task.

4.3.2. Neural Network design adaptations

As described before in this test, two design strategies were tested against each other. The first is the normal strategy [[4,3,4],[4,3,4],[4,3,4],[12,4,3,2]] where the input neural nets will reduce the input and then decrease it to form the input to the sum neuron. The second strategy, [[4,3,2],[4,3,2],[4,3,2],[6,4,3,2]], forces to reduce the input to only two outputs. These two outputs then get fed into the 6 sum neurons of the primary network.
As can be seen in fig. 4.9 the 6 sum neuron approach starts more successful in the training but fastly gets overtaken by the 12 sum neuron approach at generation 15. Around this generation,

Figure 4.9.: Comparison between individuals that complete the parkour in 30s or less

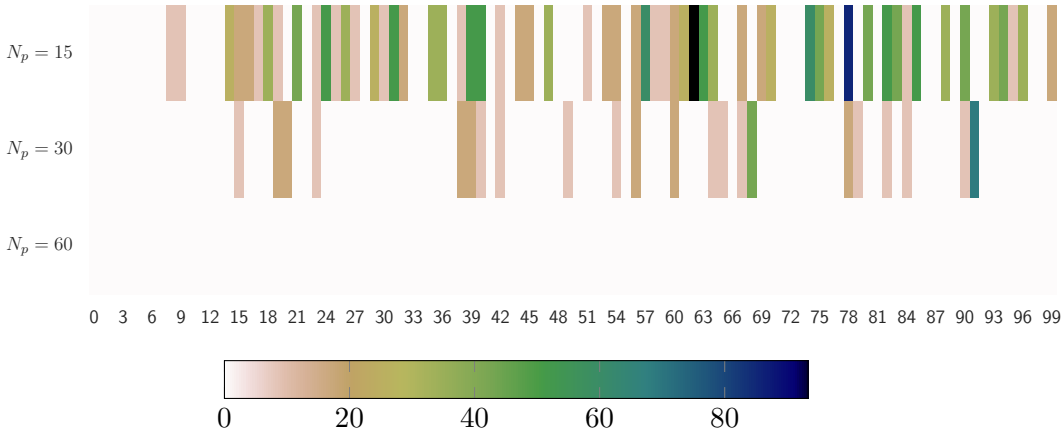


it can be seen in the simulation that the neural networks start to understand what the task is and begins to avoid the blocks. This indicates that the encoding and decoding strategy of the input neural network [4,3,4] is better suited for pre-evaluating the input data. This makes sense as the inspiration of the input neural network was a decode encode neural network, whose ability is to transform the input vector containing the position, length, and width of the neural network to an input that can be summed up without losing information. On the other hand, the [4,3,2] network has only the decoding part reducing the output. The encoded output seems not to withstand the sum neuron and performs worse.
A possible solution would be to change the design to [4,3,4,2], but as shown before, adding more complexity does not result in better data in general. Also, the added complexity would result in more compute usage as this neural is replicated for each input.

4.3.3. Population size

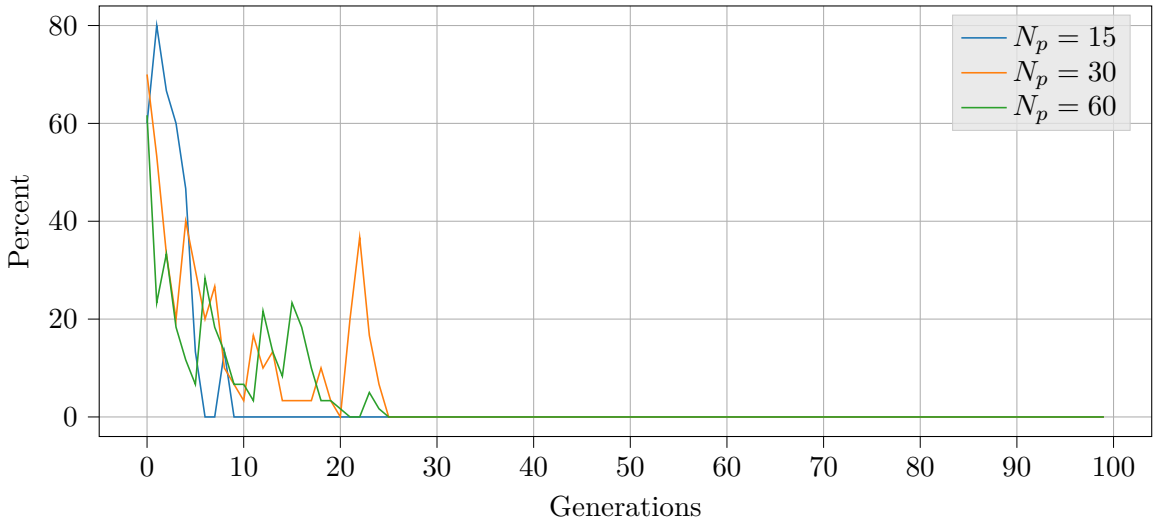
To compare the population size, we can first look at fig. 4.10. The heatmap compares the

Figure 4.10.: Comparison of 15 best individuals that have finished the parkour



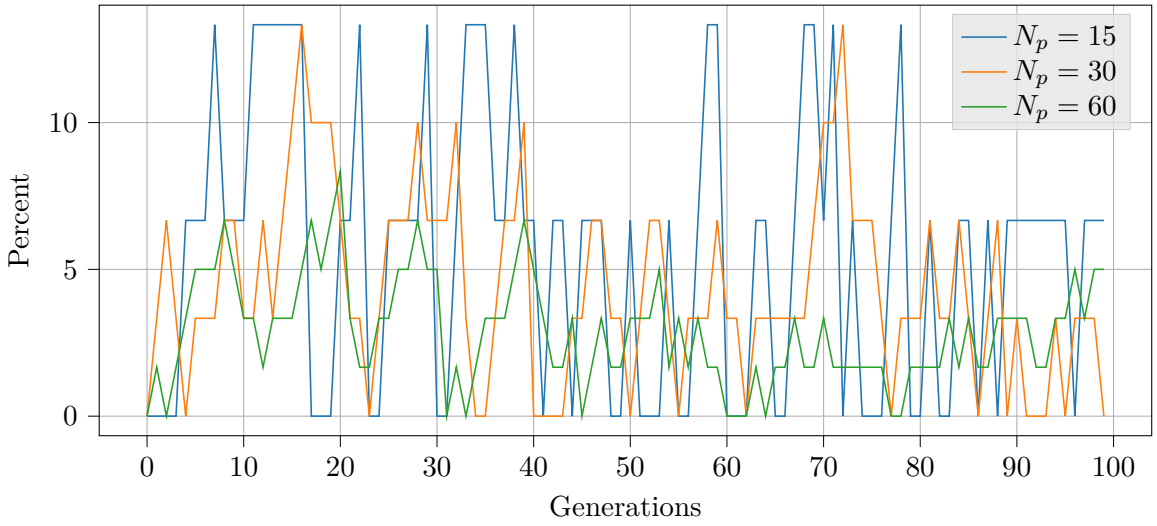
15 best individuals for each population size and shows how many percent of the 15 best individuals finished the parkour in under 30 seconds. As clearly can be seen, the population of size 15 performed much better than the $N_p = 30, N_p = 60$ populations. The $N_p = 15$ got better over time, frequently getting more than 40% of the indiums finishing the task in less than 30 seconds. On the other hand, the $N_p = 30, N_p = 60$ maps are much weaker despite their bigger population size. As can be seen, as the population gets bigger, the success rate drops with the $N_p = 60$ population not managing to finish the task once. This might seem off but is explainable by the nature of the algorithm. The algorithm uses DE OBL and CEN. OBL is present in the algorithm for the exploration of the algorithm. This is especially important at the start of the learning curve, as can be seen in fig. 4.11. While the $N_p = 15$ curve shows that after the 8th generation, the opposite individuals are completely gone from the population. $N_p = 30$ *and* $N_p = 60$ have to use opposite individuals until the 25th generation. As these individuals are more useful for narrowing down the search space and not raising the quality of the individuals, naturally, the $N_p = 15$ population has an advantage by narrowing down the search space earlier. This consequently means that the DE and CEN generated individuals are less used. As the centroid-based strategy enhances the exploitation and therefore tries to increase the quality of the individual with the algorithm, these individuals will perform better at the task. The percent of centroid-based individuals over the generations is shown in fig. 4.12. It reveals that the amount of centroid individuals in the $N_p = 60$ population is meager and is not able to surpass 7% compared to the $N_p = 15, N_p = 30$ populations. While the $N_p = 30$ population has a high amount of centroid individuals, the $N_p = 15$ population

Figure 4.11.: Comparison of opposite generated individuals present in each generation



manages to exploit the individuals even more, resulting in general higher performance. fig. 4.14 seems to confirm this result in a comparison, where the 100th evolution has to compare to each other in parkour that was not used while training. While fig. 4.14 compares the best individuals, fig. 4.15 compares the average performance and shows, that while in test one and two the $N_p = 15$ is superior, to $N_p = 60$ the gap is small and in test 3 the $N_p = 60$ performs on average better then both $N_p = 15$ *and* $N_p = 30$. This could indicate that while the single performance is better with $N_p = 15$, the average performance of the whole population is better with bigger populations. This could mean that if the population is trained past 100 generations, it may be possible that the performance could surpass the $N_p = 15$ population. Nevertheless, performing a longer test, especially with this large population, requiring a lot of computing power, which is beyond the scope of this work.

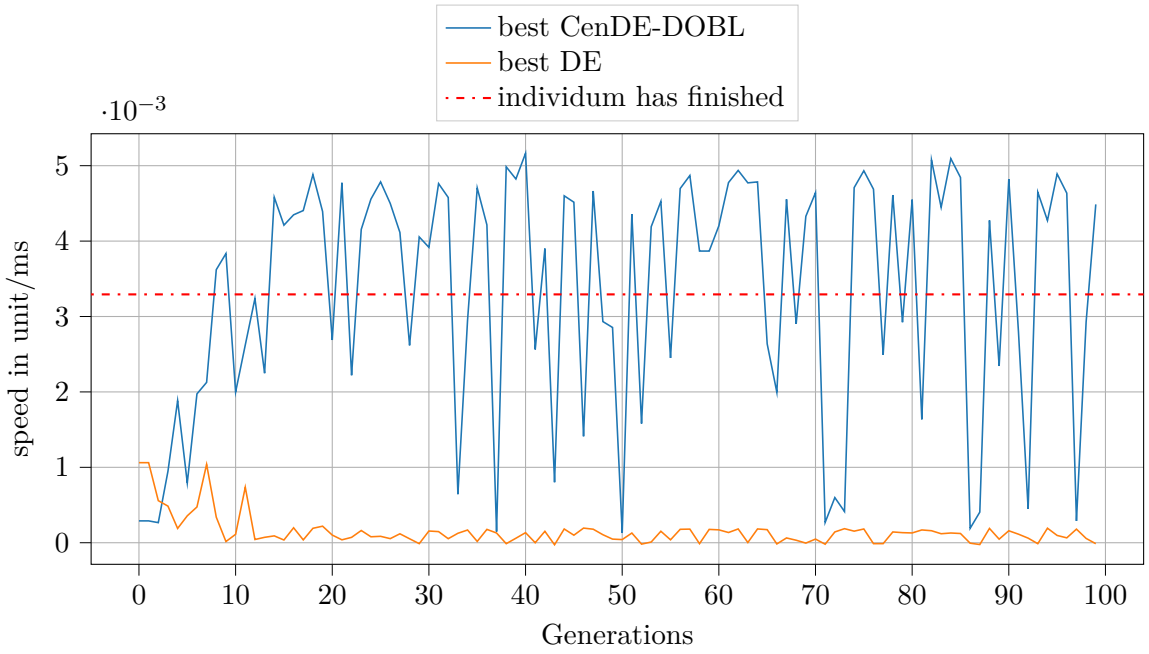
Figure 4.12.: Comparison of centroid generated individuals present in each generation



4.3.4. Comparaing CenDE-DOBL with DE

Showing the performance improvements of CenDE-DOBL over DE, both algorithms are compared in fig. 4.13. It shows that the CenDE-DOBL performance of the best individuals is far

Figure 4.13.: Comparision best individuals of DE and CenDE-DOBL with $N_p = 15$

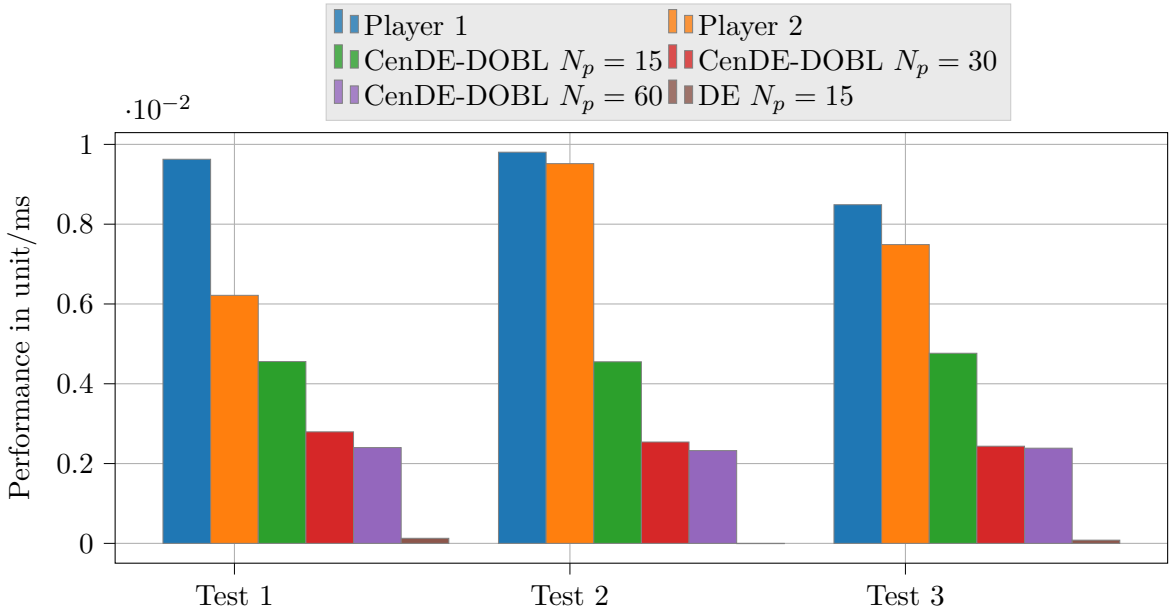


superior to the best individuals of the DE algorithm, where no individual in no generation managed to finish the task before the test was stopped. This aligns with the findings of [7] where the algorithm performed almost two times faster than DE. The reasoning behind this is quite apparent and is shown in figs. 4.11 and 4.12. The added diversity of the DOBL and CEN individuals gives it a considerable advantage. figs. 4.14 and 4.15 show this behavior as well, with the DE algorithm not being able to match the performance of the CenDE-DOBL algorithm. It should be noted that the comparison is not fair to the full extent. The DE algorithm only generates one new generation per evolution, while the CenDE-DOBL algorithm can generate an extra centroid-based individual or an entire dynamic opposite population. However, even if we only look at the first 50 evolutions of the CenDE-DOBL algorithm and compare it to the last 50 generations of the DE algorithm, it is clear that compensation for this disadvantage would hardly result in better results.

4.3.5. Human Pilot

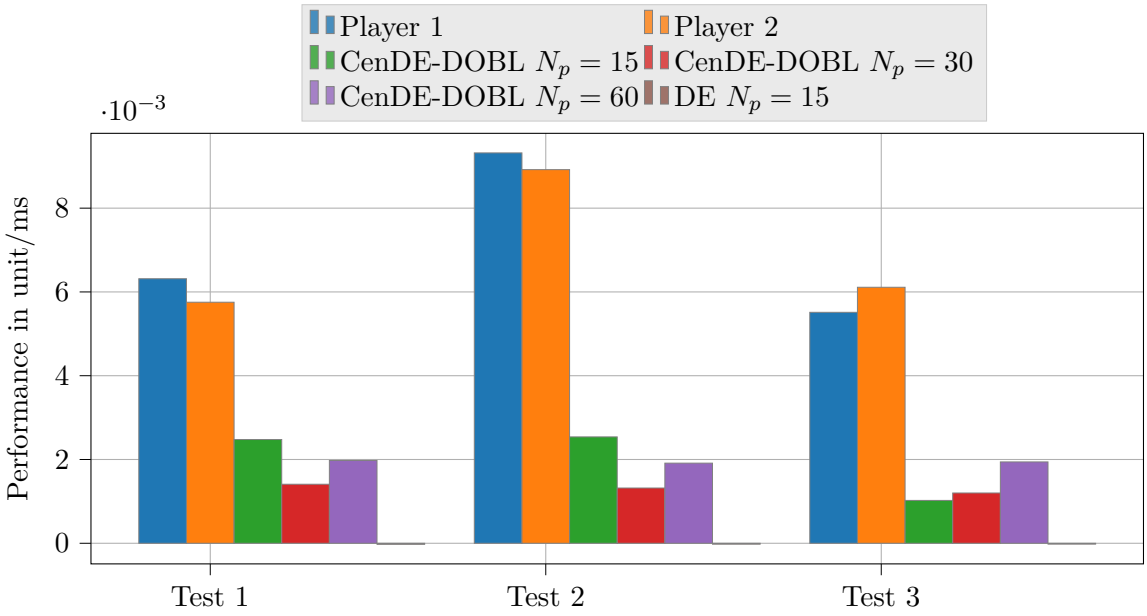
In fig. 4.14 the best results of a human pilot and the neural networks are compared. The neural network's performance is noticeably worse than the human pilot's. Nevertheless, it is impressive how good these results are, given that the training duration was only 100 generations. Comparing the different population sizes and DE also further confirms the results from previous tests and shows again how superior the CenDE-DOBL is compared to just DE. This test however can also show a problem with the evolution based trained neural networks.

Figure 4.14.: Comparing best performance from Human Pilot to neural network



In fig. 4.15 the average performance is compared, showing especially in test 3 that the best performance does not always reflect the performance of the population. A neural network that is best in one test might not be the best in another test. This could result in a problem when the neural network is transferred to an actual vehicle, as only one neural network can be used. Which neural network out of the population has the best average performance must be considered while porting.

Figure 4.15.: Comparing average performance Human Pilot to neural network



5. Outlook and Evaluation

5.1. Evaluation

The generated data has shown that the task of driving through a parkour of obstacles with a neural network as a pilot being trained with an evolution-based training algorithm was, in general successful. The resulting neural network can steer the car through most parkours with general success. In this section, different architectural decisions that were made will be evaluated.

5.1.1. Parcour design and training duration

The parkour design, in general, was relatively straightforward and was driven by the actual life model. The task was chosen to demonstrate possibilities of what machine learning can do and was also driven by the thought of presenting these possibilities, resulting in a not optimal training environment. Parkours that are not equal in difficulty resulted in a challenging training environment as some obstacles were placed close together for the car not being able to drive through. Solving this could be by selecting parkours with raising challenges, reducing the obstacles to a minimum at the beginning, and then slowly raising the count with the evolutions. While this could be a better way to do it, it also comes with a considerable amount of working overhead and the possibility of training something unwanted as the distribution and generation of the parkour is not randomly generated, giving it a bias. In hindsight, the randomized obstacle placement should have been more restrictive, making the space between obstacles more significant, and giving the car the possibility to drive through.
However, as shown in the results, the neural network could learn even under more challenging conditions with good learning results.

5.1.2. Communication solution

The solution to tunnel all of the data through a UDP connection was also driven by the desire to split the neural network from the rest of the system. It has shown to be effective but also comes with its challenges. As for every connection endpoint, there had to be a thread that sends/ received messages, which means handling opening and closing from many threads. This

was especially challenging on the Unity and C# side. Because Unity is a simulator built to run out of order execution, some threads would not close properly, resulting in a crash of the whole program. While this was eventually solved, the overall performance of Unity was not consistent, freezing at times and not being responsible. Some threads of Unity detached from the main thread, resulting in the program not noticing that the simulation was not running. Overall if the task had to be done again, picking another simulator that supports Python code natively would be endorsed instead of splitting the program apart and programming one part in C#.

5.1.3. Neural Network and training implementation

As the neural network is in a unique design, and the training algorithm is not mainstream, as it was only shown in 2021, all implementations have been done without any high-level APIs. While this is not optimal, as the neural network can now not be easily trained with other algorithms, there was only one other option of implementing this algorithm into an existing framework. Because evolution-based neural network training is straightforward to implement, it was decided not to do so. The workload would be much higher as the new functions had to be implemented into the framework. The structure and inner workings of the framework had to be understood first, adding much overhead.

5.2. Outlook

The model is now trained and ready to use in other projects and ideas. Here I will present some ideas and insights about what can be done with the generated data and model.

5.2.1. Bringing the Model to the real world

Because the design of the whole project was constructed for the neural network to split from the simulation, it is obvious to port it to an actual model car. Porting should be straightforward as on the input side, the only thing that has to change are the HSV ranges, which define the different obstacles in the parkour. Another possibility would be to replace the OpenCV pipeline internally and use some other recognition system, for example, a pre-trained neural network that is trained to perform this task.
The output and steering handling of the vehicle is slightly more challenging, as the output

is between -1,1. Some controller has to be developed to translate these inputs into motor commands that can drive the vehicle. The size between the vehicle and obstacles must also be taken into account. Overall it would be interesting to what extent porting the neural network is successful and what is needed even to get it driving.

5.2.2. Replacing OpenCV

Another idea can be to replace OpenCV from the pipeline, as it can quickly introduce errors by not detecting obstacles in shadow or show other artifacts. It could be replaced by another neural network trained by first generating many images out of the simulator and finding out the obstacle position in the picture with the simulator and then training the neural network with methods like reinforcement learning. Then the performance, like how much noise is introduced and how many obstacles are recognized, between this method and OpenCV could be compared.
Thinking even further would be to use a compression algorithm that can compress the resulting neural networks into one, reducing the overall size and potentially raising the overall performance. This could also be tested against each other to see if the approach was successful.

5.2.3. Extending the training algorithm

While the training algorithm is very performant, it can be further optimized. One way would be implementing activation function training as proposed in [22]. This will train the activation function, which is generally a fixed function, with the neural network resulting in more accurate results. Another idea would be to optimize the training by dynamically changing the jumping and crossover rates, controlling how exploitation/explorations the training algorithm is. This can be done by looking at the performance over a set period and registering if the algorithm gets stuck. Then, looking at how the population's pedigree is constructed of CEN and DOBL individuals determines how the jumping rate must be adjusted. That could potentially improve the performance by giving new tools where the algorithm is in the search space, making it easier to optimize to the optimum.

6. Selbstständigkeitserklärung

Ich versichere, dass ich die vorliegende Arbeit mit dem Thema:

Model training of a simulated self-driving vehicle using an evolution-based neural network approach

selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel angefertigt habe, insbesondere sind wörtliche oder sinngemäße Zitate als solche gekennzeichnet. Mir ist bekannt, dass Zuwiderhandlung auch nachträglich zur Aberkennung des Abschlusses führen kann.

Leipzig, den 10.06.2022

Jonas König

Bibliography

[1] A. Bewley, J. Rigley, Y. Liu, J. Hawke, R. Shen, V. Lam, and A. Kendall, "Learning to drive from simulation without real world labels," *CoRR*, vol. abs/1812.03823, 2018.

[2] F. Codevilla, M. Muller, A. Lopez, V. Koltun, and A. Dosovitskiy, "End-to-end driving via conditional imitation learning," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, May 2018.

[3] M. Bojarski, D. D. Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," 2016.

[4] D. Pomerleau, "ALVINN: an autonomous land vehicle in a neural network," in *Advances in Neural Information Processing Systems 1, [NIPS Conference, Denver, Colorado, USA, 1988]* (D. S. Touretzky, ed.), pp. 305–313, Morgan Kaufmann, 1988.

[5] S. J. Mousavirad, G. Schaefer, S. M. J. Jalali, and I. Korovin, "A benchmark of recent population-based metaheuristic algorithms for multi-layer neural network training," in *GECCO '20: Genetic and Evolutionary Computation Conference, Companion Volume, Cancún, Mexico, July 8-12, 2020* (C. A. C. Coello, ed.), pp. 1402–1408, ACM, 2020.

[6] S. Amirsadri, S. J. Mousavirad, and H. Ebrahimpour-Komleh, "A levy flight-based grey wolf optimizer combined with back-propagation algorithm for neural network training," *Neural Comput. Appl.*, vol. 30, no. 12, pp. 3707–3720, 2018.

[7] S. J. Mousavirad, D. Oliva, S. Hinojosa, and G. Schaefer, "Differential evolution-based neural network training incorporating a centroid-based strategy and dynamic opposition-based learning," in *2021 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1233–1240, 2021.

[8] S. Mirjalili, S. M. Mirjalili, and A. Lewis, "Let a biogeography-based optimizer train your multi-layer perceptron," *Inf. Sci.*, vol. 269, pp. 188–209, 2014.

[9] S. J. Mousavirad, A. A. Bidgoli, H. Ebrahimpour-Komleh, G. Schaefer, and I. Korovin, "An effective hybrid approach for optimising the learning process of multi-layer neural networks," in *Advances in Neural Networks - ISNN 2019 - 16th International Symposium on Neural Networks, ISNN 2019, Moscow, Russia, July 10-12, 2019, Proceedings, Part I*

(H. Lu, H. Tang, and Z. Wang, eds.), vol. 11554 of *Lecture Notes in Computer Science*, pp. 309–317, Springer, 2019.

[10] R. Storn and K. Price, "Differential evolution – a simple and efficient heuristic for global optimization over continuous spaces," *J. of Global Optimization*, vol. 11, dec 1997.

[11] A. Slowik, "Application of an adaptive differential evolution algorithm with multiple trial vectors to artificial neural network training," *IEEE Transactions on Industrial Electronics*, vol. 58, no. 8, pp. 3160–3167, 2011.

[12] S. J. Mousavirad and S. Rahnamayan, "Evolving feedforward neural networks using a quasi-opposition-based differential evolution for data classification," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 2320–2326, 2020.

[13] F. Cheng and J. Zhao, "A novel process monitoring approach based on feature points distance dynamic autoencoder," in *29th European Symposium on Computer Aided Process Engineering* (A. A. Kiss, E. Zondervan, R. Lakerveld, and L. Özkan, eds.), vol. 46 of *Computer Aided Chemical Engineering*, pp. 757–762, Elsevier, 2019.

[14] H. Song, G. Yang, and X.-J. Wang, "Reward-based training of recurrent neural networks for cognitive and value-based tasks," 08 2016.

[15] S. Messalti, A. Harrag, and A. Loukriz, "A new variable step size neural networks mppt controller: Review, simulation and hardware implementation," *Renewable and Sustainable Energy Reviews*, vol. 68, pp. 221–233, 2017.

[16] R. Kamaleswaran, R. Mahajan, and O. Akbilgic, "A robust deep convolutional neural network for the classification of abnormal cardiac rhythm using single lead electrocardiograms of variable length," *Physiological Measurement*, vol. 39, p. 035006, mar 2018.

[17] S. Sharma, S. Sharma, and A. Athaiya, "Activation functions in neural networks," *towards data science*, vol. 6, no. 12, pp. 310–316, 2017.

[18] F. Agostinelli, M. D. Hoffman, P. J. Sadowski, and P. Baldi, "Learning activation functions to improve deep neural networks," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2015.

[19] B. Karlik and A. V. Olgac, "Performance analysis of various activation functions in generalized mlp architectures of neural networks," *International Journal of Artificial Intelligence and Expert Systems*, vol. 1, no. 4, pp. 111–122, 2011.

[20] S. Rahnamayan, H. R. Tizhoosh, and M. M. Salama, "Quasi-oppositional differential evolution," in *2007 IEEE Congress on Evolutionary Computation*, pp. 2229–2236, 2007.

[21] S. J. Mousavirad and S. Rahnamayan, "A novel center-based differential evolution algorithm," in *2020 IEEE Congress on Evolutionary Computation (CEC)*, pp. 1–8, 2020.

[22] G. Bingham, W. Macke, and R. Miikkulainen, "Evolutionary optimization of deep learning activation functions," *CoRR*, vol. abs/2002.07224, 2020.

List of Figures