april-fallingblocks.md

The Falling Blocks Game

- Two versions of the falling blocks game has been implemented. A solution to the game is found using a simple GA without crossover, but no good solution is found when using NEAT. In this report I will describe the game, the solutions and the results.

Introduction

This section will give a general description of the game, and then each of its elements. After each description, a description of the implementation is given. The game consists of a 2-dimensional environment in which there are spawned a block and a paddle with sensors and motors. The neural network agent is supposed to control the paddle, to either catch or avoid the block. Whether the block is to be caught or avoided is to be determined based on observation of the block size.

The various tasks of the game:

Task 1:

- Catch blocks of size 1
- Avoid blocks of size 3

Task 2:

- Catch blocks of size 1
- Avoid blocks of size 2

Task 3:

- Catch blocks of size 1
- Catch blocks of size 4
- Avoid blocks of size 2
- Avoid blocks of size 3

Task 4:

- Catch blocks of size 3
- Catch blocks of size 6
- Avoid blocks of size 4
- Avoid blocks of size 5

The Environment

The environment in which the block and paddle are spawned is 2-dimensional, but the lateral edges are wrapped around. This means that the block and paddle can move from the right edge to the left edge with one step to the right(, and the other way around). The size of the environment is 36 units high and 16 units wide.

The Block

The block is always initialized at a random position on the upper edge of the environment, with a random falling-direction which is either straight down, diagonal down-left or diagonal down-right. During the game instance, the falling direction will never change. The block will be initialized with different sizes, where each size determines whether the agent should catch or avoid the block.

The block falls with one unit per time-step.

The Paddle

The paddle is initialized in the middle position at the lower edge of the environment. The paddle always has a size of 3 units. Where a sensor is placed at the first and third unit - the edges of the paddle. This means the paddle has a "blind spot" on its second unit. A sensor can see the closest object that is in a straight line upwards from the sensor. The motors of the paddle can be activated to propel it to the left or right, respectively. If both motors are activated simultaneusly the paddle will not move. The neural network agent's input nodes is connected to the sensors on the paddle, while its output is connected to the motors of the paddle. This implies that each neural network has 2 input nodes and 2 output nodes, while they have an unknown amount of hidden nodes.

The paddle can be moved one unit per timestep.

Implementation

Implementation A:

The following implementation of the game is written twice, once for NEAT and once as a single python script. The NEAT implementation A1:, and the single script will be called implementation B2.

Here the environment is implemented as a 2-dimensional numpy array of zeros. The block and paddle are initialized as ones, which are moved each timestep. The rules for movement are still one step per timestep. The block is always falling down, and has a falling direction of left, right or straight. Sensors light up if there's another index containing one in the column of the first or third unit of the paddle. The agent moves the paddle by activating one of the motors. Success is asserted by checking if there are, or are not, indices containing two in the bottom row.

Below are descriptions of the methods in the game-implementation for NEAT:

A1:: Move Paddle:

This method asserts the position of the block related to the position of the paddle, activates the network with an input and returns an output for the motor. Its a computationally heavy check, but should be foolproof in terms of logical errors.

```
def move_paddle(self, board, ann):
    idx = [i for i in range(len(board[0])) if board[-1,i]==1] # save the indices where the paddle is
    sens_in = [0,0]
    for x in range(len(board)-1): # check all rows
        idx2 = [i for i in range(len(board[0])) if board[x,i]==1] # save indices of block
        if idx2:
            # sends the height of the block to the sensor if it aligns
            sens_in[0] = len(board)-x if idx[0] in idx2 else 0
            sens_in[1] = len(board)-x if idx[2] in idx2 else 0
            break
    motor_out = ann.activate(sens_in)
    return motor_out
```

A1: Update Block:

This method simply takes the 2-dimensional array and the row number, then it moves the previous row to the current row, while using input_func() to move the block one step horisontally.

```
def update(self, board, row):
    # move the block
    board[row] = board[row] + self.input_func(copy.deepcopy(board[row-1]),self.direction)
    # flush previous line
    board[row-1] = np.zeros(len(board[row-1]))
```

A1: Move something one step:

```
def input_func(self, b, d):
    """
    Args:
        b: input array
        d: an array of two elements. The combination determines the direction for something to be moved.
    """
    b = list(b)
    if d[0] > d[1]: # move left
```

```
            return np.array([b[-1]] + b[0:-1])
        elif d[0] < d[1]: # move right
            return np.array(b[1:]+[b[0]])
        else:
            return np.array(b)
```

A1: Game Run:

```
    def run(self, animat):
        # reset game
        self.board = np.zeros((self.game_height, self.game_width))
        # initializes the paddle based on w=16 and h=36
        self.board[-1, 7:10] = 1
        # set block size with a "coin flip"
        p = random.randint(0,1)

        if p == 0:
            # block size 1
            beg = random.randint(0, self.game_width-2)
            end = beg+1
            self.board[0, beg:end] = 1
        else:
            # block size 3
            beg = random.randint(0, self.game_width-4)
            end = beg+3
            self.board[0, beg:end] = 1

        # the direction is later based on the bigger value
        self.direction = [random.randint(-1,1), random.randint(-1,1)]

        for h in range(1, self.game_height): # until the block is at the bottom of the board
            motor_out = self.move_paddle(self.board, animat)
            self.board[-1] = self.input_func(self.board[-1], motor_out) # moving paddle
            self.update(self.board, h)

        # check values in bottom line (0=nothing, 1=paddle/block, 2=paddle+block)
        u, c = np.unique(self.board[-1], return_counts=True)

        if p == 2 and 2 in u:
            return 1
        elif p == 1 and 2 not in u:
            return 1
        else:
            return 0
```

Below are descriptions of the methods in the game-implementation for NEAT: This implementation is essentially the same as the one above, but with a simple genetic algorithm written in the script itself.

A2 Creating the Genomea

```
    def org_seed(nodes):
        genome = list(np.random.randint(1, nodes+2, size=[1,nodes]))
        genome.extend(list(np.random.rand(9,nodes)))
        return genome
```

A2 Updating the Block

```
    def udate(a, x):
        a[x] = a[x] + input_func(copy.deepcopy(a[x-1]), np.random.randint(-1,2)) # move the block
        a[x-1] = y # flush previous line
```

A2 Running the ANN

```
    def output_func(c, ann):
        idx = [i for i in range(len(c[0])) if c[-1, i] == 1]
        dist = [0, 0]
        for x in range(len(c)-1):
            idx2 = [i for i in range(len(c[0])) if c[x, i] == 1]
            if idx:
                dist[0] = len(c)-x if idx[0] in idx2 else 0
                dist[1] = len(c)-x if idx[1] in idx2 else 0
                break
        d = ann.input_output(dist)
        return d
```

A2 Function for moving something one step

```
def input_func(b, d):
    # Function for moving something one step.
    # b = input array, d = direction
    if d > 0:
        return np.array([b[-1]]+b[0:-1])
    else:
        return np.array(b)
```

A2 Results

The current best result for this script when running a population size of 20 over 36000 generations is: Timestamp: 2020-05-14 10:52:38.030201 Iteration: 32082 of 36000, ANNs: 20, Score, mean: 91.00, max: 112.00 - took 2.37s *(Note that the algorithm is very slow, with over 2 minutes per generation!*

Implementation B1

The height and width of the environment is simply constraints limiting the number of possible positions and the duration of the game. The position of the block and the paddle are stored as integers in a vector, and updated at every timestep. The update is based on the direction of the block or the decision of the agent, and the update-functions contain several if-conditions that make up for the contraints of the environment. When a unit of the block is vertically aligned with the position of a sensor, the sensor will light up.

B1 Paddle update-function

The function for updating the paddle is implemented as follows:

```
def _update_paddle(self, motor_out):
    if motor_out[0] == 0 and motor_out[1] == 0:
        self.paddle_pos = self.paddle_pos # nothing happens
    elif motor_out[0] == 1 and motor_out[1] == 1:
        self.paddle_pos = self.paddle_pos # nothing happens
    elif motor_out[1] == 1: # move right
        if self.paddle_pos == self.game_width-1: # wrap-around
            self.paddle_pos = 0
        else:
            self.paddle_pos += 1
    elif motor_out[0] == 1: # move left
        if self.paddle_pos == 0:
            self.paddle_pos = self.game_width-1 # wrap-around
        else:
            self.paddle_pos -= 1
```

B1 Block update-function

The function for updating the block is implemented as follows:

```
def _update_block(self):
    self.block_pos[0] += 1 # move down
    if self.block_pos[1] == self.game_width -1: # handle edge-case on the right
        if self.direction == 1:
            self.block_pos[1] = 0 # wrap-around
        elif self.direction == -1:
            self.block_pos[1] = self.game_width - 2 # move left
        elif self.direction == 0:
            self.block_pos[1] = self.block_pos[1] # no horisontal movement / falling straight down
    elif self.block_pos[1] == 0: # handle edge-case on the left
        if self.direction == 1: # move right
            self.block_pos[1] = 1
        elif self.direction == -1: # wrap-around
            self.block_pos[1] = self.game_width -1
        elif self.direction == 0:  # no horisontal movement / falling straight down
            self.block_pos[1] = self.block_pos[1]
    else:
        self.block_pos[1] += self.direction
```

These update-functions should be efficient and error-free.

B1 Game Run function

I divide this explanation into two parts, before and after the block hits the lower edge.

B1 Before the block hits the lower edge.

The code below is simply a loop that first updates the block. Then the position of the block and paddle are compared to create input to the network. The network decides an output which determines the update of the paddle.

```
    def run(self, animat):
        '''
        Arg animat: The neural network object with 2 input and 2 output, needs a method activate()
        '''
        # Reset game
        self.block_pos = [0, random.randint(0,self.game_width-1)] # position is in height, width (rows, columns)
        # the block pos is always indicating the leftmost unit of the block
        # the paddle pos is always indicating the middle unit of the paddle
        p = random.randint(0,1) # coin flip
        if p == 1:
            self.block_size = 1
        else:
            self.block_size = 3

        self.paddle_pos = int(self.game_width/2) # along the x-axis / cols
        self.direction = random.randint(-1,1)

        while self.block_pos[0] < self.game_height-1: # until the block as at the bottom of the board
            self._update_block()
            # The horisontal position of the block and sensors are compared.
            if self.block_size == 1: #
                if self.block_pos[1] == self.paddle_pos - 1: # left sensor lights up
                    sens_in = [1,0]
                elif self.block_pos[1] == self.paddle_pos + 1: # right sensor lights up
                    sens_in = [0,1]
                else: # no sensors light up
                    sens_in = [0,0]
            if self.block_size == 3:
                if self.block_pos[1] == self.paddle_pos-1: # block of size 3 aligns perfectly with a paddle of size 3.
                    sens_in = [1,1]
                elif self.block_pos[1] == self.paddle_pos:
                    sens_in = [0,1]
                elif self.block_pos[1] == self.paddle_pos+1:
                    sens_in = [0,1]
                elif self.block_pos[1]+1 == self.paddle_pos-1:
                    sens_in = [1,0]
                elif self.block_pos[1]+1 == self.paddle_pos+1:
                    sens_in = [0,1]
                elif self.block_pos[1]+2 == self.paddle_pos-1:
                    sens_in = [1,0]
                elif self.block_pos[1]+2 == self.paddle_pos:
                    sens_in = [1,0]
                else:
                    sens_in = [0,0]

            output = animat.activate(sens_in)
            self._update_paddle(output)
```

B1 After the block hits the lower edge.

At the last timestep there are no updates and the final positions of the block and the paddle are compared.

```
if self.block_pos[0] == self.game_height-1:
    # catch
    if self.block_size == 1:
        if self.paddle_pos == 0: # handle left edge-case wraparound
            if self.game_width-1 == self.block_pos[1] or 1 == self.block_pos[1] or 0 == self.block_pos[1]:
                return 1
            else:
                return 0
        elif self.paddle_pos == self.game_width-1:
            if self.game_width-2 == self.block_pos[1] or 0 == self.block_pos[1] or self.game_width-1==self.block_pos[1]:
                return 1
            else:
                return 0
        else:
            if self.paddle_pos-1 == self.block_pos[1] or self.paddle_pos == self.block_pos[1] or self.paddle_pos+1 == sel
                return 1 #self.game_width-3 # point
            else:
                return 0
    # avoid
    elif self.block_size == 3:
        if self.paddle_pos == 0:
            if self.block_pos[1]+2 >= self.game_width-1:
                return 0
            elif self.block_pos[1] <= 1:
                return 0
            else:
                return 1 # successfully avoided
        elif self.paddle_pos == self.game_width-1:
            if self.block_pos[1] >= self.game_width-3:
                return 0
            elif self.block_pos[1] <= 0:
                return 0
            else:
                return 1 # successfully avoided
        else:
            # is the left side of the block on the right side of the paddle
            if self.block_pos[1] > self.paddle_pos + 1:
                return 0
            elif self.block_pos[1] + 2 < self.paddle_pos - 1:
                return 0
            else:
                return 1 # successfully avoided
```

B1 Results

This implementation had good results when it was more simple, without wrap-around, with only one task. Wrap-around made is especially hard.

Getting stuck in local optima

- A problem with Task 1 is that when there's enough games there's statistically 50-50 change to get catch and avoid. For the catch there's a high probability that if you don't try to catch you will fail, and for the avoid there's a high probability that if you don't try you'll still succeed. These probabilities are more or less complimentary, so that over time you will get 50% fitness without trying. Most solution-searches tend to stagnate here...

Possible solutions

Tuning the GA to explore more hills (avoid local optima) with implementation B1

Hypthesis B1A

- A medium population size (200-400) with a low mutate rate will require many generations to find a good hill.

- An increased mutate rate will find the correct hill, but might also loose it.
- A high mutate rate with low interspecie compatibility might make the solutions stay on their hills.
- High stagnation treshold will let NEAT explore the hill for a long time.
- Low elitism rate will make space for new solutions within a species.

Hypothesis B1B

- A large population size (1000) with a low mutate rate could find a good hill in fewer generations.
- There is a high probability that each individual will find the same initial solution, and that they need to branch out afterwards, this is given that all genomes start out with no hidden nodes and no connections.
- It seems that it does not matter if number of generations is high or if population size is high.
- B1A has a medium pop with many generations, so then this, B1B will have a large pop with few generations.
- The rest will remain the same: low compatibility, high stagnation treshold, high mutate rate, low elitism rate.

Hypothesis B1C

- This should refect B1A while keeping the mutate rate low and increasing compability instead, assuming compability can be seen as crossover rate.
- Thus:
  - A low mutate rate with high compability, high stagnation and low elitism.

Death Laser

- This technique will kill of solutions that are exploits a mechanic to get a relatively high score even though they're wrong.
- The example shown here is used in the implementations below:

```
# at the last time-step
if self.block_pos[0] == self.game_height-1:
    # TESTING "KILLING LASER"
    if self.moves_cnt == 0:
        return 0
```

More advanced fitness function

- The current fitness function is only returning an the output of the numer of successful games run through an exponential function.
- A better approach might be to more evaluate the in-game performance more closely.

B2: Scoring based on tracking, not wins

- Score is accumulated on each timestep, and the added value is amplified by the inverted distance.
- Tracking means that the sensor is vertically aligned with the block, at any timestep.
- In every iteration, the input to the network is [0,0] if the sensors are not aligned with the block, and amplified by the increasing block height if they align.

B2: Implementation

Below is the entire function game.run(), broken down into sections.

B2: Preparing the game:

As for every implementation, the position of the block and paddle are reset. The direction of the block is set.

```
def run(self, animat, d=False):
    '''
    It takes an animat player to play the game.
    '''

    #RESET GAME
    self.block_pos = [0, random.randint(0,self.game_width-1)] # postion in y,x / rows, cols
    # Set block size with a "coin flip"
    p = random.randint(0,1)
    if p == 1:
        self.block_size = 1
    else:
        self.block_size = 3 #self.board[self.block_pos[0]][self.block_pos[1]] = 1 self.paddle_pos = random.randint(0, sel
```

B2: The timesteps while the block is falling

In this implementation the sensor input is determined by the use of the modulo function, to assure alignment in wrap-around situations. The input to the sensor is amplified by the block height, as the blocks y-position (which is increasing). After the agent has made its decision and the paddle has been updated the gamescore is increased. The score is only increased if the agent is currently doing what it should do, that is aligning or avoiding, depending on the block size. The increase is amplified by the block height (which is increasing).

```
    while self.block_pos[0] < self.game_height-1: # until the block is at the bottom of the board
        self._update_block()

        w = self.game_width

        start = self.block_pos[1]
        end = start+self.block_size
        left_sens = self.paddle_pos-1
        right_sens = self.paddle_pos+1

        sens_in = [0,0]

        for i in range(start, end, 1):
            if (i%w) == left_sens%w:
                sens_in[0] = self.block_pos[0]
            if (i%w) == right_sens%w:
                sens_in[1] = self.block_pos[0]

        if d == True:
            self._print_game()
            print(sens_in)

        output = animat.activate(sens_in)
        self._update_paddle(output)

        # check for crash to give score for tracking
        start = self.block_pos[1]
        end = start+self.block_size
        crash = False
        paddle_start = self.paddle_pos-1
        paddle_end = self.paddle_pos+1

        for i in range(start, end, 1):
            for j in range(paddle_start, paddle_end+1, 1):
                crash = (i%w) == (j%w)
                if crash: break
            if crash: break

        if crash:
            if self.block_size == 1:
                score += self.block_pos[0] # score increases when the distance to the paddle decreases
        else:
            if self.block_size == 3:
                score += self.block_pos[0] # score increases when the distance to the paddle decreases

        if d == True:
            self._print_game()
            print(sens_in)
```

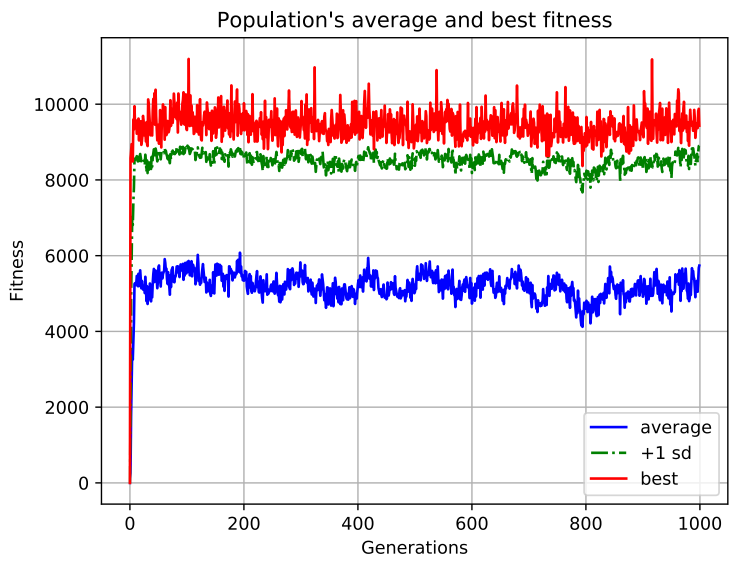
B2: The last time step

There's a mechanic called "killing laser" which is there to kill off solutions which no movement to be a good solution. Finally the positions are checked for alignment, and this score update is the strongest.

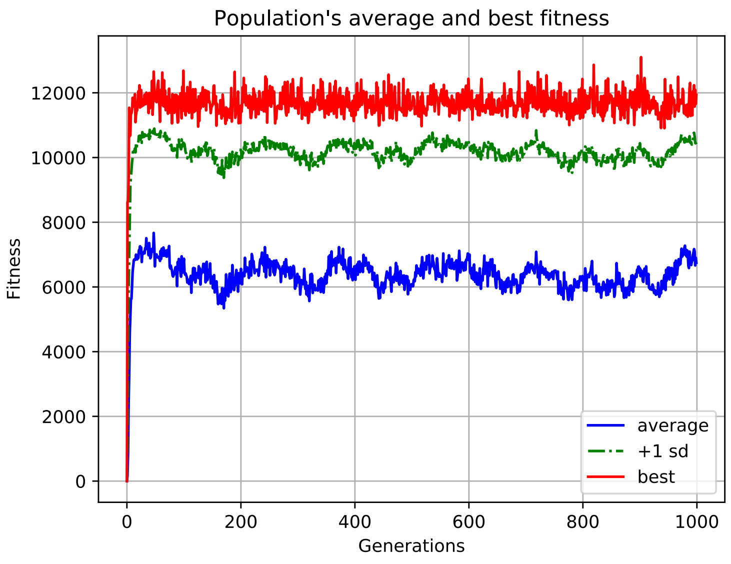
```
    # at the last time-step
    if self.block_pos[0] == self.game_height-1:
        # TESTING "KILLING LASER"
        if self.moves_cnt == 0:
            return 0

        # check for crash
        start = self.block_pos[1]
        end = start+self.block_size
        crash = False
        paddle_start = self.paddle_pos-1
        paddle_end = self.paddle_pos+1

        for i in range(start, end, 1):
            for j in range(paddle_start, paddle_end+1, 1):
                crash = (i%w) == (j%w)
                if crash: break
            if crash: break
        if crash:
            if self.block_size == 1:
                score += self.block_pos[0]
        else:
            if self.block_size == 3:
                score += self.block_pos[0]

        if d == True:
            print(score)

        return score
```

B2: Results

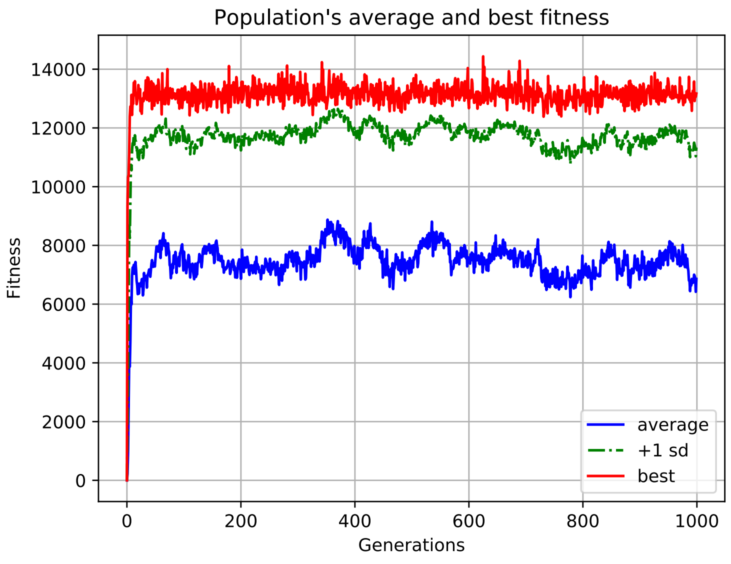
The experiment was divided into B2, B3, B4, B5 and B6. The division was made to test a hypothesis regarding the probability of the block sizes. More specifically, if there's an equal ratio between the blocks, a 50% will be guaranteed, and the solutions could stagnate at 50%. The experiments below show that the fitness indeed does change with the ratio of the blocks. B5 and B6 has exclusively blocks of size 1 and of size 3 respectively, and these results indiciate that avoiding blocks of size 3 might be easier than catching blocks of size 1.



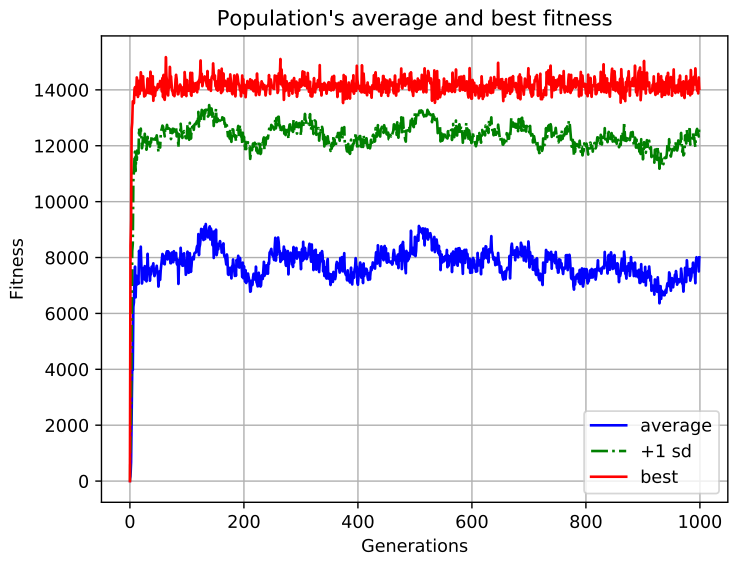
B2: Tracking, with a 50/50 ratio between blocks of size 1 and of size 3. Max score is 17280.



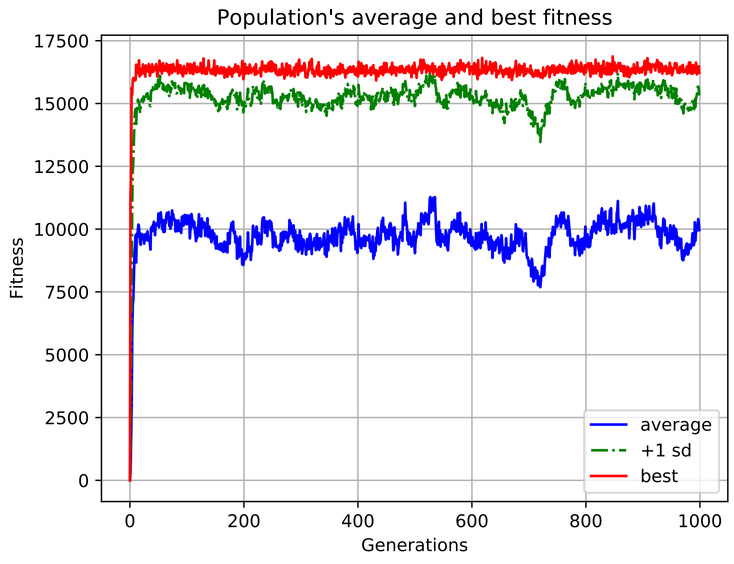
B3: Tracking, with a 75/25 ratio between blocks of size 1 and of size 3. Max score is 17280.



B4: Tracking, with a 25/75 ratio between blocks of size 1 and of size 3. Max score is 17280.



B5: Tracking, with a 100/0 ratio between blocks of size 1 and of size 3. Max score is 17280.



B6: Tracking, with a 0/100 ratio between blocks of size 1 and of size 3. Max score is 17280.