# 11_Neuroevolution_Part1_JupyterExport

January 21, 2025

## 1 Neuroevolution

Neuroevolution uses evolutionary algorithms to optimise neural networks. Before we start with the implementation of neural networks, let's import required dependencies.

[1]:
```
import random
import math
from statistics import mean
import numpy as np
import matplotlib.pyplot as plt

from graphviz import Digraph
```

### 1.1 Neural Networks

Neural networks consist of neurons and weighted connections between these neurons. Each neuron represents a processing unit in which an activation function is applied to the weighted sum of all incoming connections. After a neuron has been activated, its activation signal is further propagated into the network.

[2]:
```
# Basic neuron definition from which all neuron genes will be derived.
class Neuron:

    def __init__(self, uid: str):
        self.uid = uid
        self.signal_value = 0
        self.activation_value = 0
        self.incoming_connections = []
```

#### 1.1.1 Input Neuron

Input neurons receive a signal from the environment (input feature) and propagate it into the network. Since we are interested in the raw input signal, we refrain from applying any activation functions within the input layer.

[3]:
```
class InputNeuron(Neuron):

    def __init__(self, uid: str):
        super().__init__(uid)
```

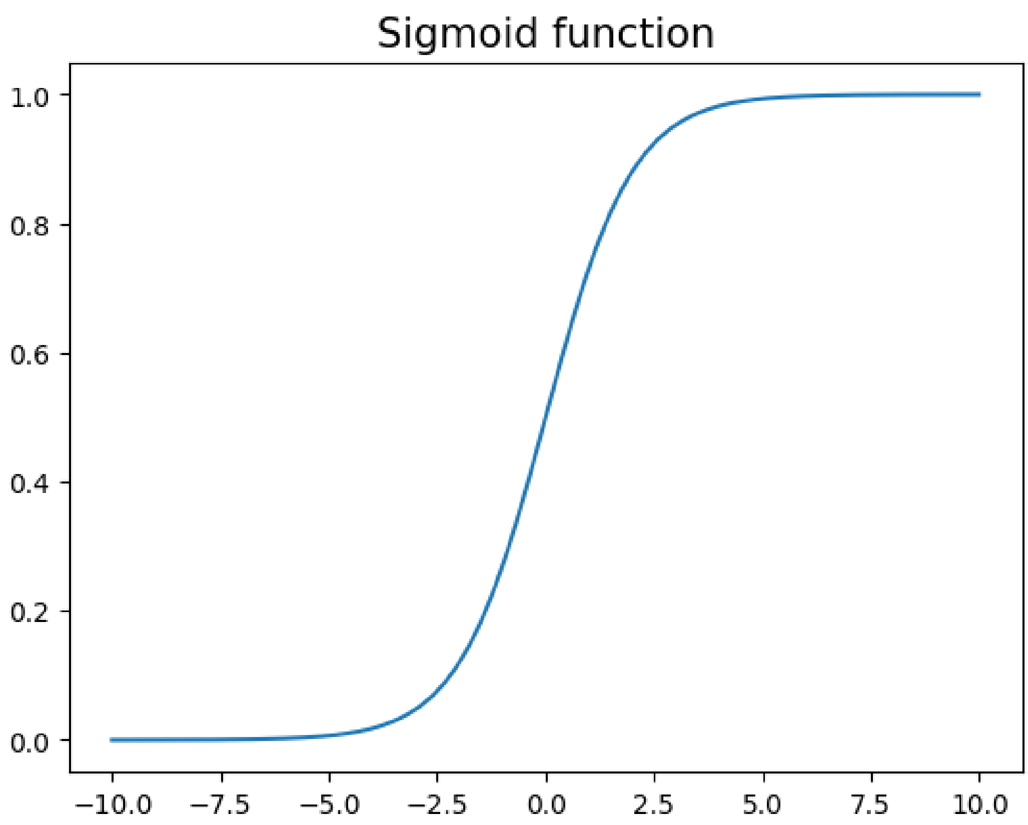
```
    def activate(self) -> float:
        self.activation_value = self.signal_value
        return self.activation_value
```

### 1.1.2 Hidden Neuron

Hidden neurons reside between input and output neurons. They increase the capacity of a neural network, in other words, we require more hidden neurons for complex tasks than for simple ones. The number and distribution of neurons is an optimisation task on its own and one of the main reasons to use neuroevolution. As an activation function, we will use the sigmoid function, which maps any input value to $[0, 1]$.

[4]:
```
def sigmoid(x: float):
    return 1 / (1 + math.exp(-x))

x_range = np.arange(-10, 10, 0.01)
sigmoid_values = [sigmoid(x) for x in x_range]
plt.plot(x_range, sigmoid_values)
plt.title('Sigmoid function', fontsize=15)
plt.show()
```



[5]:
```
class HiddenNeuron(Neuron):

    def __init__(self, uid: str):
        super().__init__(uid)

    def activate(self) -> float:
        self.activation_value = sigmoid(self.signal_value)
        return self.activation_value
```

### 1.1.3 Output Neuron

For our output neurons, we have to choose an appropriate activation function that matches the given task. Later, we will try to solve a binary decision problem in which a robot can move to the right or left. Hence, we could use a sigmoid function, whose output is treated as a probability for choosing one of the two classes. However, due to the neuroevolution algorithm we will implement, we add two output neurons to the output layer, assign one output neuron to each class, and choose the class whose output neuron has the highest activation value.

[6]:
```
class OutputNeuron(Neuron):

    def __init__(self, uid: str):
        super().__init__(uid)
        self.uid = uid

    def activate(self) -> float:
        self.activation_value = self.signal_value
        return self.activation_value
```

### 1.1.4 Connections

Our neurons are rather useless as long as they are not connected. We define connections by their source and destination neurons. Furthermore, we assign a weight to each connection to specify the strength of a link between two neurons.

[7]:
```
class Connection:

    def __init__(self, source: Neuron, target: Neuron, weight: float):
        self.source = source
        self.target = target
        self.weight = weight
```

### 1.1.5 Network Definition

With our neurons and connections defined, we can now assemble both components in layers to build a neural network.

[8]:

```python
class Network:
    def __init__(self, layers: dict[float, list[Neuron]], connections: list[Connection]):
        self.layers = layers
        self.connections = connections
        self.generate()

    # Traverse list of connections and add incoming connections to neurons.
    def generate(self):
        for connection in self.connections:
            connection.target.incoming_connections.append(connection)
```

Let's build a network with five input neurons, three hidden neurons and two output neurons.

[9]:

```python
def gen_simple_network() -> Network:
    input_neurons = [InputNeuron(f"I{x}") for x in range(5)]
    hidden_neurons = [HiddenNeuron(f"H{x}") for x in range(3)]
    output_neurons = [OutputNeuron(f"O{x}") for x in range(5)]

    # Assemble in layers with 0 representing the input layer and 1 the output layer.
    layers = {0: input_neurons, 0.5: hidden_neurons, 1: output_neurons}

    # Generate connections from every input neuron to each hidden neuron
    connections = []
    for input_neuron in input_neurons:
        for hidden_neuron in hidden_neurons:
            weight = random.uniform(-1, 1)
            connections.append(Connection(input_neuron, hidden_neuron, weight))

    # Generate connections from every hidden neuron to each output neuron
    for hidden_neuron in hidden_neurons:
        for output_neuron in output_neurons:
            weight = random.uniform(-1, 1)
            connections.append(Connection(hidden_neuron, output_neuron, weight))

    return Network(layers, connections)


net = gen_simple_network()
```

We can visualise our generated network using the Graphviz library.

[10]:

```python
class Network(Network):
    def show(self) -> Digraph:
        dot = Digraph(graph_attr={'rankdir': 'BT', 'splines': "line"})
        # Use sub graphs to position input neurons are at the bottom and output neurons at the top.
```

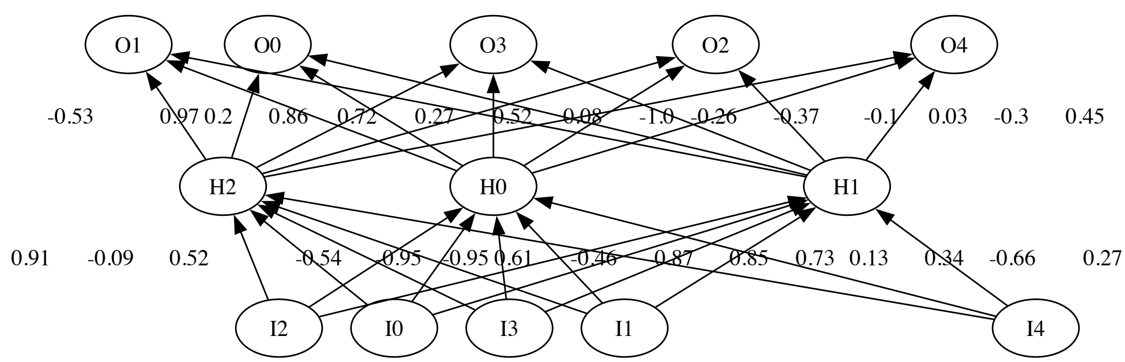
```
        input_graph = Digraph(graph_attr={'rank': 'min', 'splines': "line"})
        output_graph = Digraph(graph_attr={'rank': 'max', 'splines': "line"})
        hidden_graph = Digraph(graph_attr={'splines': "line"})

        # Traverse network from input to output layer and assign neurons to corresponding subgraph.
        layer_keys = list(self.layers.keys())
        layer_keys.sort()
        for layer in layer_keys:
            for neuron in self.layers.get(layer):
                if layer == 0:
                    input_graph.node(neuron.uid, color='black', fillcolor='white', style='filled')
                elif layer == 1:
                    output_graph.node(neuron.uid, color='black', fillcolor='white', style='filled')
                else:
                    hidden_graph.node(neuron.uid, color='black', fillcolor='white', style='filled')

        # Combine the sub graphs to a single graph
        dot.subgraph(input_graph)
        dot.subgraph(hidden_graph)
        dot.subgraph(output_graph)
        # Link the nodes based on the connection gene.
        for connection in self.connections:
            dot.edge(connection.source.uid, connection.target.uid,
                     label=str(round(connection.weight, 2)), style='solid')
        return dot
```

[11]:
```
gen_simple_network().show()
```

[11]:




Now that we have our network structure in place, we can implement the network activation that takes an input signal as an argument and outputs the result of the network after it has been activated. We can realise a network's activation by computing our neurons' activation values sequentially from the input to the output layer.

[12]:
```
class Network(Network):
    def activate(self, inputs: list[float]) -> list[float]:
        # Reset neuron values from previous executions
        for neuron_layer in self.layers.values():
            for neuron in neuron_layer:
                neuron.signal_value = 0

        # Load input features into input neurons
        input_neurons = self.layers.get(0)
        for i in range(len(inputs)):
            input_neurons[i].signal_value = inputs[i]
            input_neurons[i].activate()

        # Traverse the neurons of our network sequentially starting from the␣
↪hidden layer.
        layer_keys = list(self.layers.keys())
        layer_keys.sort()
        for layer in layer_keys:
            for neuron in self.layers.get(layer):
                if layer > 0:
                    # Calculate weighted sum of incoming connections
                    for connection in neuron.incoming_connections:
                        neuron.signal_value += connection.source.
↪activation_value * connection.weight
                neuron.activate()

        output_neurons = self.layers.get(1)
        return [o.activation_value for o in output_neurons]
```

[13]:
```
gen_simple_network().activate([0, 1])
```

[13]:
```
[0.4115740022496357,
 -0.6218943439217343,
 -0.0018156850109715128,
 -0.1409252268959789,
 0.39929351346955444]
```

## 1.2 Inverted Pendulum Problem

We will use our networks to solve the inverted pendulum (= pole-balancing) problem, a well-known benchmark task in the reinforcement learning community. In this problem scenario, a pole is centred on a cart that can be moved to the right and left. Obviously, any movement to the cart also impacts the pole. The task is to move the cart to the left and right such that the pole remains balanced. Whenever the cart position exceeds the boundaries of the track or the pole tips over 12 degrees, the balancing attempt is deemed a failure.

We can simulate the cart and pole system using the following two equations that describe the acceleration of the pole $\ddot{\theta}_t$ and the cart $\ddot{p}_t$ at a given point in time $t$:

$$\ddot{p}_t = \frac{F_t + m_p l(\dot{\theta}_t^2 sin\theta_t - \ddot{\theta}_t\ cos\theta_t)}{m} \qquad (1) \ddot{\theta}_t = \frac{mg\ sin\theta_t - cos\theta_t(F_t + m_p l\dot{\theta}_t^2\ sin\theta_t)}{\frac{4}{3}ml - m_p l\ cos^2(\theta_t)} \qquad (2)$$

where - $p$: Position of the cart - $\dot{p}$: Velocity of the cart - $\ddot{p}$: Acceleration of the cart - $\theta$: Angle of the pole - $\dot{\theta}$: Angular velocity of the pole - $\ddot{\theta}$: Angular acceleration of the pole - $l$: Length of the pole - $m_p$: Mass of the pole - $m$: Mass of the pole and cart - $F$: Force applied to the cart - $g$: Gravity acceleration

In our simulation, we will update both systems using the discrete-time equations

$$p(t+1) = p(t) + r\dot{p}(t)\ (3) \theta(t+1) = \theta(t) + r\dot{\theta}(t)\ (4) \dot{p}(t+1) = \dot{p}(t) + r\ddot{p}(t)\ (5) \dot{\theta}(t+1) = \dot{\theta}(t) + r\ddot{\theta}(t)\ (6)$$

with the discrete time step $r$ set to 0.02 seconds.

[14]:
```
GRAVITY = 9.8  # m/s^2
MASS_CART = 1.0  # kg
MASS_POLE = 0.1  # kg
TOTAL_MASS = (MASS_CART + MASS_POLE)  # kg
POLE_LENGTH = 0.5  # m
POLEMASS_LENGTH = (MASS_POLE * POLE_LENGTH)
FORCE = 10  # N
STEP_SIZE = 0.02  # sec
FOURTHIRDS = 1.333333


def cart_pole_step(action: int, p: float, p_vel: float, theta: float, theta_vel:
 ↪ float) -> (float, float, float, float):
    force_dir = FORCE if action > 0 else -FORCE

    cos_theta = math.cos(theta)
    sin_theta = math.sin(theta)

    temp = (force_dir + POLEMASS_LENGTH * theta_vel * theta_vel * sin_theta) /␣
 ↪TOTAL_MASS

    # Equation 2
    theta_acc = (GRAVITY * sin_theta - cos_theta * temp) / (
                POLE_LENGTH * (FOURTHIRDS - MASS_POLE * cos_theta * cos_theta /␣
 ↪TOTAL_MASS))
    # Equation 1
    p_acc = temp - POLEMASS_LENGTH * theta_acc * cos_theta / TOTAL_MASS
```

```
    # Compute new states
    p = p + STEP_SIZE * p_vel  # Equation 3
    theta = theta + STEP_SIZE * theta_vel  # Equation 4
    p_vel = p_vel + STEP_SIZE * p_acc  # Equation 5
    theta_vel = theta_vel + STEP_SIZE * theta_acc  # Equation 6

    return p, p_vel, theta, theta_vel
```

Let's test our cart pole system by executing a few steps starting from a clean state where all state variables are set to zero. Given a balanced pole ($\theta = 0$), we expect the pole to always move in the opposite direction of the applied force. Moreover, we expect increasing velocity values as long as we apply a force in the same direction and a decrease in velocity as soon as the force direction changes.

[15]:
```
p = 0
p_vel = 0
theta = 0
theta_vel = 0
# Move the cart for 10 steps to the right.
for i in range(10):
    [p, p_vel, theta, theta_vel] = cart_pole_step(1, p, p_vel, theta, theta_vel)
    print(f"Iteration right: {i}\nCart Pos: {p}\nCart Vel: {p_vel}\nPole Angle:␣
 ↪{theta}\nPole AVel: {theta_vel}")
    print("--------------------")

# Move the cart for 15 steps to the left.
for i in range(15):
    [p, p_vel, theta, theta_vel] = cart_pole_step(-1, p, p_vel, theta,␣
 ↪theta_vel)
    print(f"Iteration left: {i}\nCart Pos: {p}\nCart Vel: {p_vel}\nPole Angle:␣
 ↪{theta}\nPole AVel: {theta_vel}")
    print("--------------------")
```

```
Iteration right: 0
Cart Pos: 0.0
Cart Vel: 0.1951219547888171
Pole Angle: 0.0
Pole AVel: -0.29268300535397695
--------------------
Iteration right: 1
Cart Pos: 0.0039024390957763423
Cart Vel: 0.3902439095776342
Pole Angle: -0.005853660107079539
Pole AVel: -0.5853660107079539
--------------------
Iteration right: 2
Cart Pos: 0.011707317287329027
Cart Vel: 0.5854473662672043
```

```
Pole Angle: -0.017560980321238616
Pole AVel: -0.8798872190960108
--------------------
Iteration right: 3
Cart Pos: 0.023416264612673113
Cart Vel: 0.7808034482987078
Pole Angle: -0.03515872470315883
Pole AVel: -1.178038896588744
--------------------
Iteration right: 4
Cart Pos: 0.03903233357864727
Cart Vel: 0.9763639418404654
Pole Angle: -0.05871950263493371
Pole AVel: -1.4815329625443723
--------------------
Iteration right: 5
Cart Pos: 0.05855961241545658
Cart Vel: 1.172151077414685
Pole Angle: -0.08835016188582115
Pole AVel: -1.7919612011435535
--------------------
Iteration right: 6
Cart Pos: 0.08200263396375028
Cart Vel: 1.3681454441981398
Pole Angle: -0.12418938590869222
Pole AVel: -2.1107473332066196
--------------------
Iteration right: 7
Cart Pos: 0.10936554284771308
Cart Vel: 1.5642715914153589
Pole Angle: -0.1664043325728246
Pole AVel: -2.4390888082002777
--------------------
Iteration right: 8
Cart Pos: 0.14065097467602025
Cart Vel: 1.76038115934149
Pole Angle: -0.21518610873683017
Pole AVel: -2.7778872734265025
--------------------
Iteration right: 9
Cart Pos: 0.17585859786285005
Cart Vel: 1.956233861883977
Pole Angle: -0.27074385420536023
Pole AVel: -3.127668490894963
--------------------
Iteration left: 0
Cart Pos: 0.21498327510052959
Cart Vel: 1.7632652933902342
```

```
Pole Angle: -0.3332972240232595
Pole AVel: -2.9273895035470296
--------------------
Iteration left: 1
Cart Pos: 0.25024858096833424
Cart Vel: 1.571344914406432
Pole Angle: -0.3918450140942001
Pole AVel: -2.7515365246297137
--------------------
Iteration left: 2
Cart Pos: 0.28167547925646286
Cart Vel: 1.3805005689274916
Pole Angle: -0.44687574458679435
Pole AVel: -2.5992441491311347
--------------------
Iteration left: 3
Cart Pos: 0.3092854906350127
Cart Vel: 1.1907126899876923
Pole Angle: -0.49886062756941707
Pole AVel: -2.469569739043521
--------------------
Iteration left: 4
Cart Pos: 0.33309974443476653
Cart Vel: 1.0019309505127758
Pole Angle: -0.5482520223502875
Pole AVel: -2.3615648963925833
--------------------
Iteration left: 5
Cart Pos: 0.35313836344502203
Cart Vel: 0.8140860856326241
Pole Angle: -0.5954833202781391
Pole AVel: -2.274325969546335
--------------------
Iteration left: 6
Cart Pos: 0.3694200851576745
Cart Vel: 0.6270978571284449
Pole Angle: -0.6409698396690658
Pole AVel: -2.2070281641767515
--------------------
Iteration left: 7
Cart Pos: 0.3819620423002434
Cart Vel: 0.44088004947248216
Pole Angle: -0.6851104029526008
Pole AVel: -2.1589473496805938
--------------------
Iteration left: 8
Cart Pos: 0.39077964328969306
Cart Vel: 0.2553432345191425
```

```
Pole Angle: -0.7282893499462127
Pole AVel: -2.1294729185112797
--------------------
Iteration left: 9
Cart Pos: 0.3958865079800759
Cart Vel: 0.07039587671876138
Pole Angle: -0.7708788083164384
Pole AVel: -2.1181143011627457
--------------------
Iteration left: 10
Cart Pos: 0.39729442551445115
Cart Vel: -0.11405579905980248
Pole Angle: -0.8132410943396933
Pole AVel: -2.1245030706087547
--------------------
Iteration left: 11
Cart Pos: 0.3950133095332551
Cart Vel: -0.29810887869795544
Pole Angle: -0.8557311557518683
Pole AVel: -2.1483920226503734
--------------------
Iteration left: 12
Cart Pos: 0.38905113195929597
Cart Vel: -0.4818655664058769
Pole Angle: -0.8986989962048758
Pole AVel: -2.189652192573561
--------------------
Iteration left: 13
Cart Pos: 0.37941382063117846
Cart Vel: -0.6654355960582027
Pole Angle: -0.942492040056347
Pole AVel: -2.2482684480876864
--------------------
Iteration left: 14
Cart Pos: 0.3661051087100144
Cart Vel: -0.8489391605252926
Pole Angle: -0.9874574090181008
Pole AVel: -2.324334063461633
--------------------
```

We can formulate our pole-balancing task as a reinforcement learning problem by implementing a simulation in which an agent's goal is to balance the pole for as long as possible. In our scenario, our agents are neural networks that decide in which direction the cart should be moved at a given state $[p, \theta, \dot{p}, \dot{\theta}]$. Since we have four input features, the networks will have four input neurons. Furthermore, these input features will be normalised over the following ranges:

$$p : [-2.4, 2.4] \dot{p} : [-1.5, 1.5] \theta : [-12, 12] \dot{\theta} : [-60, 60]$$

We model the pole-balancing problem as a binary classification task using two output neurons, each representing one action. An action is chosen by selecting the action that belongs to the neuron with the highest activation value. Once the pole tips over 12 degrees or the 4.8 meter track is left by the cart, the trial ends. Finally, an agent's performance is measured by the number of steps it managed to survive. A balancing attempt is deemed successful whenever an agent balances the pole for 120,000 time steps, which is equivalent to 40 minutes in real time.

For a more challenging second scenario, we add an option that randomises the starting positions of the four input states.

[16]:
```
MAX_STEPS = 120000
TWELVE_DEGREES = 0.2094395  #conversion to rad (12*pi)/180

def evaluate_agent(network: Network, random_start=False) -> int:
    # Define starting state
    if random_start:
        p = random.uniform(-2.4, 2.4)  # -2.4 < p < 2.4
        p_vel = random.uniform(-1.5, 1.5)  # -1.5 < p_acc < 1.5
        theta = random.uniform(-TWELVE_DEGREES, TWELVE_DEGREES)  # -12 < theta␣
↪< 12
        theta_vel = random.uniform(-1, 1)  # -60 < theta_acc < 60 (in rad)
    else:
        p, p_vel, theta, theta_vel = 0.0, 0.0, 0.0, 0.0

    # Simulation loop
    steps = 0
    while steps < MAX_STEPS:
        # Normalise inputs
        inputs = [None] * 5
        inputs[0] = (p + 2.4) / 4.8
        inputs[1] = (p_vel + 1.5) / 3
        inputs[2] = (theta + TWELVE_DEGREES) / (2 * TWELVE_DEGREES)
        inputs[3] = (theta_vel + 1.0) / 2.0
        inputs[4] = 0.5

        # Activate the network and interpret the output, and advance the state.
        net_output = network.activate(inputs)
        action = -1 if net_output[0] > net_output[1] else 1
        p, p_vel, theta, theta_vel = cart_pole_step(action, p, p_vel, theta,␣
↪theta_vel)

        # Check if the attempt is still valid
        if p < -2.4 or p > 2.4 or theta < -TWELVE_DEGREES or theta >␣
↪TWELVE_DEGREES:
            return steps

        steps += 1
```

```
    # At this point the agent survived for the maximum number of steps
    return steps
```

Let's create a sample network and test it in the pole-balancing environment.

[17]:
```
net = gen_simple_network()
evaluate_agent(net, random_start=False)
```

[17]: 8