IT3708

SUBSYMBOLIC METHODS IN AI

ASSIGNMENTS 4 AND 5

A Neural Network Controller for Webots

Kjetil Valle
kjetilva@stud.ntnu.no

Olav Bjørkøy
olavfrih@stud.ntnu.no

NTNU, Trondheim
April 28, 2010

Abstract

This report describes the implementation of a generic framework for creating, training and testing artificial neural networks. Networks are generated through descriptions in script files, and can be tested by specifying case files. The framework is used to create a controller for a robot, which is trained to perform various tasks in a simulated environment. The robot learns to avoid obstacles in the environment, and to drive to blocks of a specified color.

1 Introduction

Artificial neural networks are powerful constructs in artificial intelligence. However, deciding on topology, arc weights and other parameters is often a process governed by trial and error. It is therefore beneficial to use a generic ANN framework, where the parameters are set in external script files, so that customization and tweaking remain as painless as possible.

The first part of this report describes the implementation of such a generic ANN framework. The framework uses script files for defining the network parameters, and case files for feeding predetermined input data into the network generated from a script. By using external resources for customization, the user is free to create any kind of ANN to solve the task at hand. The framework is tested for correctness and performance through the use on a few simple cases.

The second part of the report outlines how the framework was used to create a neural network controller for an EPuck robot simulated in the Webots simulator. The implemented robot controller uses two neural nets, one for avoiding obstacles, and one for targeting an element of specified color in the virtual world.

The report is ended by a description of the most important problems encountered during the project, ideas for further work, and a summary of the main conclusions reached when developing the generic ANN framework and robot controller.

2 The Generic ANN Framework

The generic ANN framework provides a flexible way of generating and running neural networks. Through the use of case files, scripts and task specifications, a user can specify the topology of a network, as well as optional training and testing data, without changing any of the framework's internal code.

Since choices regarding layers and nodes, arcs and links, training method, sensory input and expected output are properties set by the user, the framework remains generic and can be used for many purposes. This is beneficial as most neural nets are customized for the task at hand, with custom number of layers, nodes, connection schemes, learning and activation functions.

Through the use of simple text-files for specifying the nature of the network, small and large changes can be made to any aspect of the generated result. As noted in the section on creating an ANN controller for a Webot, the inevitable process of experimentation with different network properties during development demonstrated the value of such flexibility.

The framework is implemented using the Python programming language. In addition to being a language well suited for fast prototyping, using Python ensured an easy way to interface with the Webots application, the second part of this exercise.

An overview of the framework is found in Figure 1.

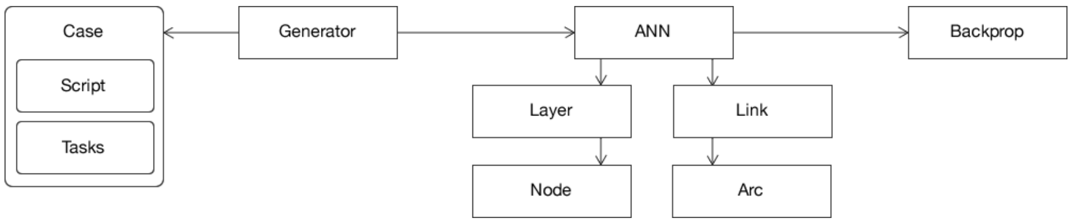

Figure 1: Class diagram: The API consists of the case and script files, the Generator and ANN class. The latter class has a collection of layers and links, which in turn contains a number of nodes and arcs. The Backprop class contains the logic necessary to train the network using the backpropagation method.

2.1 Features

The framework strives to support all functions needed to provide flexible construction of neural nets. For each layer, the user can set properties such as the node count, activation function, quiescence threshold, max settling rounds and whether it should be active during training. Other properties include whether the layer should have a bias node, and several settings relative to which activation function the user selects.

Each link layer has properties to adjust its source and destination, its plasticity, learning rate and learning rule and the connection topology. This topology can be fully connected, one-to-one, triangular, stochastic, or customized by manually setting each arc weight. If no weights are set, the initial weights of arcs can either be a fixed number or set to be a random number in some range.

A user can utilize the task structure feature to specify inputs and outputs for training and testing, as well as which mode the network should be in for each input. The framework supports unsupervised hebbian learning, and supervised learning in the form of the backpropagation algorithm.

2.2 API

A primary focus of the framework is to have a simple yet powerful API. The user should be able to specify the network, generate an ANN object and work with this object through a small number of well-defined processes. The API is comprised of the four main parts listed below.

Case files: These files specify which script to use for defining the network topology, and textual tasks the network can execute. A task has a setting for which mode the network should be in for this particular task, e.g. testing or backpropagation training. In addition, each task has input and optional output values for use during training and testing. Case files are found in the 'annlib/cases' folder.

Script files: A script file defines the topology of the network and provides settings for each layer and link layer. Each layer has settings such as the number of nodes, its activation function and the quiescence mode threshold. Each link layer has settings for the connection type, learning rate, initial weight boundaries, learning rule and plasticity. Finally, a script file specifies the ordering of layers during execution. Script files are found in the 'annlib/scripts' folder.

Generator: The generator takes a case file and generates a new network based on its script file. This class returns a new ANN object, with the proper configuration of layers and nodes, links and arcs, as specified in the script file.

Ann: The ANN class is the finished neural net as created by the generator. It has methods for running input, setting its mode and collecting output values. This class can also serialize networks, so that they can be saved for future use. Serialized networks are saved to the 'annlib/networks' folder.

See the Appendices A and B for examples of case and script files, respectively.

2.3 Internal Classes

The internal, non-visible parts of the framework are controlled by the ANN class, and follows the structure outlined in [Downing, 2009a]. These files make few assumptions, and rely on case files and scripts created by the user. Should some aspect of these configurations not be set, the internal classes fall back on default values, which are also specified in an external script file found in 'scripts/defaults.yml'. The most important internal classes are:

Layer: Contains all settings, properties and nodes belonging to a layer.

Node: Contains node settings and a host of activation functions. Instances of this class also have derivatives of these activation functions for use during backpropagation training.

Link: Contains all arc objects linking the nodes of two layers together. Arcs are organized in a 2D matrix created by the generator based on the link connection structure from the script file.

Arc: Each arc has its weight, previous weight, membrane potential and other helpful fields. It also contains several learning rules for training, such as the general hebbian rule and the oja rule.

Backprop: This is a utility class used by the ANN class for training the arcs using the backpropagation method. It works directly on layers and links, extracting all the functionality of this mode into a single, easily maintainable class. The backpropagation algorithm is implemented as described in [Downing, 2009b].

2.4 Testing the framework: Learning XOR

Before creating the webot controller, the framework was tested with a number of simple examples to ensure correctness and good performance. Learning how to compute the XOR of two inputs through the use of backpropagation training is a suitable example, as this relatively simple task shows that a generated network is capable of dealing with non-linearly separable outputs after training.

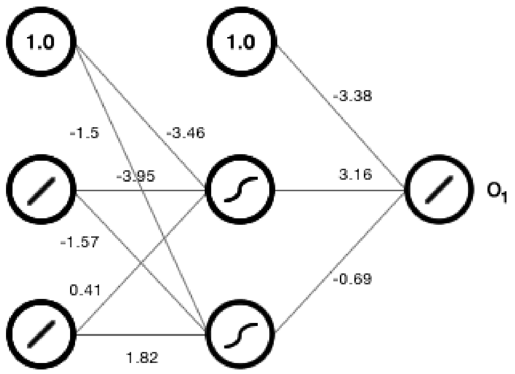

Figure 2: Topology from the script for learning XOR. Weights as trained by backpropagation. The top nodes in the first and second layers are bias nodes for shifting the sigmoid threshold.

A network with two input nodes, two hidden nodes and one output node, with full link connections between each layer was generated. The hidden layer had the logistic sigmoid activation function. The two first layers were given bias nodes to in order for the backpropagation algorithm to be able to shift the thresholds of the activation function in each neuron. The network was then trained until the error threshold was below 0.01.

The resulting network could reliably compute the XOR function with an error less than the given threshold setting. See Figure 6 and 9 in the appendix for the case and script files used in this example.

Figure 2 illustrates the trained network.

3 Controlling a Webot

The Generic ANN framework has been applied to create a controller for a simulated EPuck robot. The following section describe the task of the robot. Section 3.2 describe the implementation details of the controller. Lastly the results and performance of the controller are discussed.

3.1 Task Description

The task of the robot is twofold. It should drive around without crashing into everything in its environment. Its first task is therefore to **avoid any nearby obstacles** such as wall or blocks. It should also so something more purposeful than just wandering around aimlessly. The second goal of the robot is to **find and drive to the green block**.

The environment the robot is going to perform these tasks in is a simple one. It consists of a flat area surrounded by walls. In this area a number of blocks of different color is placed to serve as obstacles and target for the robot.

3.2 Implementation

This section describe how the webots controller is implemented. The overall architecture is first explained, and then each of its layers are described in more detail. Lastly the training process of the controller is discussed.

3.2.1 Architecture

The architecture of the webots controller is inspired by Brooks' layered control system as described in [Brooks, 1986].

The behavior of the robot is a result of the combined output of two neural network, each serving as one layer of the architecture. This makes sense since the task for the robot consist of two parts, one of which describe pretty basic behavior.

Layer one of the controller is responsible for avoiding any obstacles in the immediate area. The task of the second layer is to make the robot drive towards the green block, whenever it see it. An overview of the architecture is shown in Figure 3.

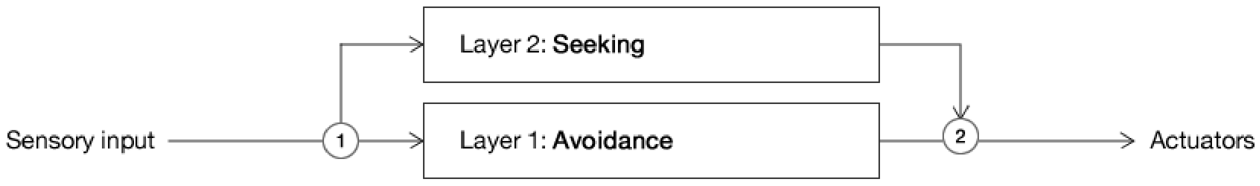

Figure 3: Overview of the controllers architecture. (1) is a component splitting the inputs between layers, and (2) is the code integrating the outputs of the layers into wheel speeds for the robot.

The two layers use different parts of the sensory inputs. This is handled by the component marked (1) in the figure, which is part of the controller code.

The component marked (2) is the one responsible for integrating the different outputs into actions for the robot. The outputs are mapped to wheel speeds for the robots. This is done using the following formulas, and then scaling them so that the largest one equals to 1.0.

$$R = A_1 + 0.1 * S_1 + 0.01$$
$$L = A_2 + 0.1 * S_2 + 0.01$$

L and R represents the left and right wheel speeds of the robot, respectively. A is the vector of outputs from the avoidance network, and S is the outputs from the network seeking green blocks.

The result of the above computation is that whenever the robot is close to anything, the first layer fires and takes the control of the robot. If the first layer is quiet, the second layer have the chance of influencing the robot if it detects any green blocks in the robots field of vision. When the second layer is quiet too, the last term of the equation ensures that the robot will continue driving straight ahead.

It should be mentioned that this architecture differs from Brooks' in that here the lower and more basic behavior always overrules that of the higher layers, whereas in his model the higher layers are able to suppress those below.

3.2.2 Layer 1: Avoid obstacles

The first layer in the controller is responsible for steering the robot clear of any obstacles in its surroundings. To do this it uses as its inputs values from the six forwardmost proximity sensors.

The signals from the sensors are preprocessed before they are given as inputs to the network. A threshold is applied, with values below 100 regarded as 0, and those above as 1.

The topology of the network is shown at the left side of Figure 4. The network used is a feedforward network with 6 nodes in the hidden layer, and uses fully connected links between layers. Sigmoid is used as activation function for the hidden layer, and linear activation for input and output layers. The input layer and hidden layer each have one bias node, so that backpropagation is able to shift the threshold of the sigmoids.

The network's weights can be found in Appendix C.

3.2.3 Layer 2: Find the green box

The second layer of the controller uses the robot's visual inputs. The responsibility of this layer is to steer the robot towards any blocks of a specific color in its field of vision. Which color the robot should seek is easily configurable in the controller, and in this task green is used.

The color value of each pixel is calculated relative to the other colors present in the pixel. For green the resulting formula for the relative value V is the following.

$$V = G - \frac{R+B}{2}$$

The image signal is preprocessed before fed to the neural network. First the image is reduced to a row vector containing the maximum value of V in each column. Then this vector is reduced to a 5 element vector selecting the maximum of consecutive elements.

The elements of the resulting vector is then fed to the network, and the outputs used as indicated

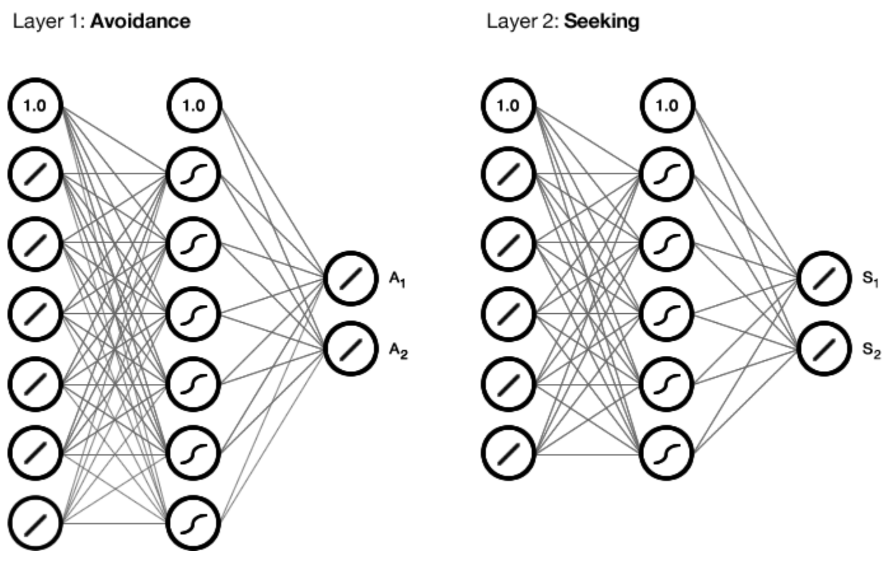

Figure 4: Topologies of the networks controlling controller. Left: layer 1, avoidance. Right: layer 2, seeking of the green block. Weights for the different arcs can be found in Appendix C.

previously.

The network used is similar to that used in layer 1 above, except it uses only 5 nodes for the input and hidden layers. An illustration of the network is given in Figure 4.

3.2.4 Training the Robot

The two networks in the controller were trained using supervised learning. By providing the backpropagation module of the framework with examples of how the robot should behave given various sensory inputs, the behavior of each net were learned.

Initially the training started out with only the most basic examples. The robot was then simulated until a situation it did not know how to handle were encountered, and a new training example covering this was added. This process was repeated until the robot handled its task well in the simulated environment.

The error threshold was set to 1%, which means that the training was stopped when the total error reached this value. This was done in order to avoid overfitting the training cases.

The final cases used to train the robot can be found in Appendix A.

3.3 Results

The achieved behavior of the robot is pretty much as expected. Unless hindered by any obstacles it will always drive straight ahead, unless it has a green block in its sight. As soon as the robot can see a green block, it will correct its heading and turn towards it.

This behavior was achieved by training the robots neural networks with example cases of how it should behave in different situations. For each of the two networks only 10 training cases were used. This means that the networks have generalized well from the training data.

Figure 5 shows a screenshot of the simulator with the robot heading for a green block.

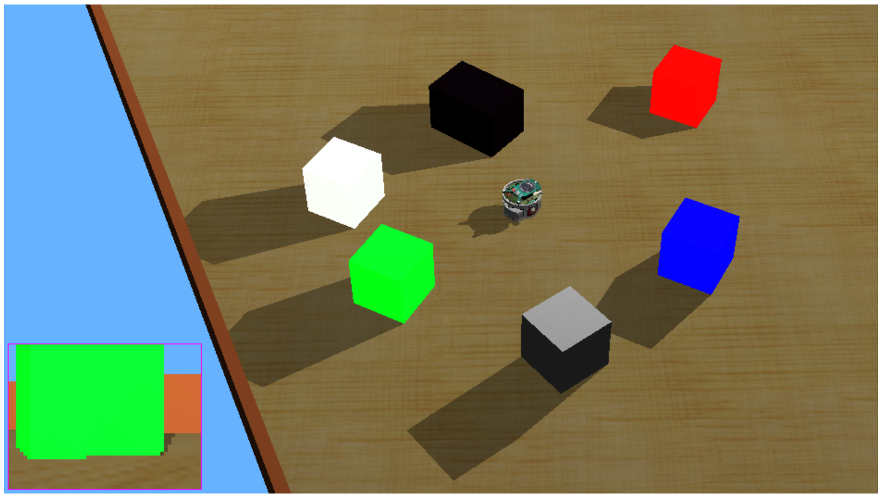

Figure 5: A screenshot of the robot just as it is heading towards the green block.

4 Problems encountered

The first problem encountered during this project was how to specify the script file format. The format needed to give the user enough flexibility to properly define as many aspects of the generated networks as possible. While the final resulting script structure allows such customization, arriving at the final syntax took some time.

Concluding that case files was the way to go in order to organize training and testing data also took some time. Initially simple python files were used to organize training and testing input, but this caused problems as the amount of data grew. It became apparent that some other solution was needed, and case files were created.

Apart from this, the most challenging part of the project was to implement and use backpropagation. At first a version of backpropagation were implemented without bias nodes, and this lead to some problems when trying to train the simple xor-network from section 2.4. After reading about the algorithm in [Floreano and Mattiussi, 2008], bias nodes were added, and training worked as expected.

A technical problem that had to be overcome was getting the Webots program to run as intended under MacOSX. The application fared fine during installation and initial setup, but then got confused about whether it should use 32 or 64 bit python. Once this problem was resolved, Webots proved to be a great simulation tool in which to test the controller.

5 Further work

There are several limitations to both the framework and the webots controller that it would be possible to refine in the future. Given the scope of the project, a lot of interesting functionality where implemented. However, a lot of possibilities remain unexplored and are outlined here as posibilities for future work.

First of, the framework currently do not support reinforcement learning. This is something that it would have been interesting to implement, and would have opened a lot of possibilities with the robot as well.

Visuallisation of the created networks also were not prioritized, and thus the framework only supports textual printing of the weights and activations of the networks at this time. This is an area where a lot could be done, and something that would greatly enhance the use of the framework.

A limitation of the controller is that the robot, when not provided by proximity or visual stimuli, only drives in a straight line. The architecture is easily extendable with a third layer, that could provide a wandering behavior. This layer should make the robot occationally turn around in order to look at its surroundings. When the correct block is spotted layer 2 would again take control.

The webots controller have not yet been tested on a physical robot. This is expected to be an interesting and challenging transition.

6 Conclusions

A generic framework for creating artificial neural networks have been created. The framework is general enough to create a wide variety of ANNs, and supports unsupervised learning through hebbian rules, and supervised learning with the Backpropagation algorithm.

Networks are created through script files describing their structure and properties. The networks are trained and tested using cases. Both script and case files can be configured independent of the framework without touching any code.

The framework has been used to create neural networks for a controller for a simulated EPucks robot. The robot is simulated using the Webots simulation tool.

Backpropagation was used to train the robot to avoid obstacles in the environment, and to look for and drive to blocks of a specified color.

References

[Brooks, 1986] Brooks, R. (1986). A robust layered control system for a mobile robot. *IEEE journal of robotics and automation*, 2(1):14–23.

[Downing, 2009a] Downing, K. (2009a). Building a generic artificial neural network. Available from `http://www.idi.ntnu.no/emner/it3708/lectures/generic-ann.pdf`, accessed 26.04.2010.

[Downing, 2009b] Downing, K. (2009b). Supervised learning in neural networks. Available from `http://www.idi.ntnu.no/emner/it3708/lectures/learn-sup.pdf`, accessed 26.04.2010.

[Floreano and Mattiussi, 2008] Floreano, D. and Mattiussi, C. (2008). *Bio-Inspired Artificial Intelligence.* The MIT Press.

Appendix

The appendix contains case files, scripts and weight values for different networks described in the main text.

A Case Files

A case file is the main structure run by a network. It contains the name of the script to be used and a set of tasks used during training and testing.

```
class: text
script: learn_xor
tasks:

  - mode: training-backpropagation
    epochs: 3000
    error_threshold: 0.01
    input:
      - '0 0'
      - '0 1'
      - '1 0'
      - '1 1'
    output:
      - '0'
      - '1'
      - '1'
      - '0'

  - mode: testing
    input:
      - '0 0'
      - '0 1'
      - '1 0'
      - '1 1'
    output:
      - '0'
      - '1'
      - '1'
      - '0'
```

Figure 6: Case file with tasks for learning the XOR function through back-propagation training.

```
class: webot
script: avoidance
tasks:

  - mode: training-backpropagation
    epochs: 3000
    error_threshold: 0.001
    input:
      - '0 0 0 0 0 0' # no walls
      - '1 0 0 0 0 0' # left side wall
      - '1 1 0 0 0 0' # left side, left wall
      - '0 1 1 0 0 0' # left, front wall
      - '0 0 1 1 0 0' # front wall
      - '0 1 1 1 1 0' # corner
      - '1 1 1 1 1 1' # hypercorner
      - '0 0 0 1 1 0' # right, front wall
      - '0 0 0 0 1 1' # right, right side wall
      - '0 0 0 0 0 1' # right side wall
    output:
      - '0 0'
      - '1 .7'
      - '1 .5'
      - '1 -.3'
      - '1 -1'
      - '1 -1'
      - '1 -1'
      - '-.3 1'
      - '.5 1'
      - '.7 1'

  - mode: testing
    input:
      - '0 0 0 0 0 0' # no walls
      - '1 0 0 0 0 0' # left side wall
      - '1 1 0 0 0 0' # left side, left wall
      - '0 1 1 0 0 0' # left, front wall
      - '0 0 1 1 0 0' # front wall
      - '0 1 1 1 1 0' # corner
      - '1 1 1 1 1 1' # hypercorner
      - '0 0 0 1 1 0' # right, front wall
      - '0 0 0 0 1 1' # right, right side wall
      - '0 0 0 0 0 1' # right side wall
    output:
      - '0 0'
      - '1 .7'
      - '1 .5'
      - '1 -.3'
      - '1 -1'
      - '1 -1'
      - '1 -1'
      - '-.3 1'
      - '.5 1'
      - '.7 1'
```

Figure 7: Webot case file for learning avoidance.

```
class: webot
script: seek
tasks:

  - mode: training-backpropagation
    epochs: 10000
    error_threshold: 0.001
    input:
      - '0 0 0 0 0'
      - '0 0 1 0 0'
      - '0 0 0 0 1'
      - '0 0 0 1 1'
      - '1 1 0 0 0'
      - '1 0 0 0 0'
      - '0 1 1 1 0'
      - '0 1 1 0 0'
      - '0 0 1 1 0'
      - '1 1 1 1 1'
    output:
      - '0 0'
      - '1 1'
      - '1 .3'
      - '1 .7'
      - '.7 1'
      - '.3 1'
      - '1 1'
      - '.9 1'
      - '1 .9'
      - '1 1'

  - mode: testing
    input:
      - '0 0 0 0 0'
      - '0 0 1 0 0'
      - '0 0 0 0 1'
      - '0 0 0 1 1'
      - '1 1 0 0 0'
      - '1 0 0 0 0'
      - '0 1 1 1 0'
      - '0 1 1 0 0'
      - '0 0 1 1 0'
      - '1 1 1 1 1'
    output:
      - '0 0'
      - '1 1'
      - '1 .3'
      - '1 .7'
      - '.7 1'
      - '.3 1'
      - '1 1'
      - '.9 1'
      - '1 .9'
      - '1 1'
```

Figure 8: Webot case file for learning seeking.

B Script files

A script file defines the properties of a neural net, and is used by the generator for creating ANN instances.

```
layers:
  input:
    size: 2
    activation_function: linear
    bias_node: true
  hidden:
    size: 2
    activation_function: logistic
    bias_node: true
  output:
    size: 1
    activation_function: linear

links:
  input_hidden:
    pre: input
    post: hidden
    connection_type: full
    learning_rate: 0.2
    weights:
      - 0.3
      - 0.7
  hidden_output:
    pre: hidden
    post: output
    connection_type: full
    learning_rate: 0.2
    weights:
      - 0.3
      - 0.7

order:
  - input
  - hidden
  - output
```

Figure 9: Script file for creating a network capable of learning the XOR function.

```
layers:
  input:
    size: 6
    activation_function: linear
    bias_node: true
  hidden:
    size: 6
    activation_function: logistic
    bias_node: true
  output:
    size: 2
    activation_function: linear

links:
  input_hidden:
    pre: input
    post: hidden
    connection_type: full
    display: true
    weights:
      - 0.3
      - 0.7
  hidden_output:
    pre: hidden
    post: output
    connection_type: full
    display: true
    weights:
      - 0.3
      - 0.7

order:
  - input
  - hidden
  - output
```

Figure 10: Script file for creating the avoidance network.

```
layers:
  input:
    size: 5
    activation_function: linear
    bias_node: true
  hidden:
    size: 5
    activation_function: logistic
    bias_node: true
  output:
    size: 2
    activation_function: linear

links:
  input_hidden:
    pre: input
    post: hidden
    connection_type: full
    display: false
    learning_rate: 0.3
    weights:
      - 0.3
      - 0.7
  hidden_output:
    pre: hidden
    post: output
    connection_type: full
    display: false
    learning_rate: 0.3
    weights:
      - 0.3
      - 0.7

order:
  - input
  - hidden
  - output
```

Figure 11: Script file for creating the seeking network.

C Network Arc Weights

The following tables show arc weights in different networks after training.

input_hidden	0.6	-0.72	0.98	0.36	0.71	0.15	-1.11
	0.97	-1.1	-3.67	1.72	0.75	1.11	-1.39
	0.28	0.83	0.17	-0.11	-0.04	0.81	-0.16
	0.43	-0.55	0.65	0.13	0.73	0.28	-0.84
	0.75	0.13	0.58	0.04	-0.08	0.69	-0.16
	2.23	1.39	2.18	-0.11	-0.55	1.58	-0.02

hidden_output	0.21	0.09	-0.79	-0.12	-0.04	2.54	-0.91
	-1.73	3.44	0.19	-1.48	-0.13	0.99	-0.33

Figure 12: Weights of avoidance network after training.