Neural Network in FlatWorld

Eric Schulte

13 December 2010

Abstract

A series of increasingly complex neural network architectures are used to control an agent navigating an a two dimensional simulated world. The design of each successive architecture is informed by the perceived weaknesses of the previous architectures. The pros and cons of various technical and design choices are discussed.

1 Introduction

This work uses a number of neural network architectures to control an agent in a simulated two dimensional environment. The goal of each architecture is to prolong the life of a simulated agent for as long as possible. The architectures have access to sensory information provided by the environment and interact with the environment through controlling the movement of the agent.

In this report we will review the technical framework with which this work was performed which is composed of both a Neural Network engine and the Flatworld two dimensional environment (Section 2), we then introduce a number of architectures including a design description as well as performance metrics for each (Section 3), finally we discuss the relative strengths and weaknesses of the various architectures (Section 4).

In addition to the source code and experimental results presented directly below the entire suite of code required to reproduce these results as well as video recordings of selected agent runs is available in the appendix.

2 Technical Approach

2.1 Neural Network DSL

All neural calculations used below are implemented using a Neural Network domain specific language implemented in the Clojure lisp dialect [1]. This language was implemented by the author to support this class work and this project and is included in the appendix of this work.

The benefit of using a domain specific language (DSL) for the specification of neural networks is the ability to concisely and naturally express networks in small amounts of code. Such code can be used both to program neural networks and also as a serialization language for persisting and sharing networks. Much more information on this language along with a variety of simple examples is available online [2].

2.2 Flatworld Integration

Clojure lisp runs on the Java Virtual Machine (JVM). This raises a significant barrier to integration with the flatworld API which is written in the C programming language. The integration involved the following steps

1. Extraction of an API from provided example files. There were a number of decisions made as to where to draw the boundary between C code run as part of the flatworld simulator and Lisp code run as part of the neural controller.

2. Compilation of the C code into shared `.so` libraries which could be called from within the JVM.

3. Construction of Lisp wrappers for all exposed C functions which could be called from within the JVM.

More details of the above process are available in the appendix.

2.3 Gnuplot Visualization

Integration with the graphical display elements of the flatworld simulation proved too difficult. However a simple alternate form of visualization was implemented using the gnuplot [3] graphing library. The code used to generate

[1] http://clojure.org/
[2] http://repo.or.cz/w/neural-net.git
[3] http://www.gnuplot.info/

the gnuplot commands is given in Figure 1. Movies of agent performance generated using this method are provided in the appendix.

```
(def fmt
  "set title 'metabolic charge: %f'
set key outside
set xrange [%f:%f]
set yrange [%f:%f]
set arrow 1 from %f,%f to %f,%f
set arrow 2 from %f,%f to %f,%f
plot \"poison\", \"food\", \"neut\"")

(defn to-rad [n] (* (/ n 360) (* 2 Math/PI)))

(let [re (re-pattern (apply str (interpose "[ \\t]+"
                                           (repeat 5 "([-\\d\\.]+)"))))]
  (loop []
    (when-let [line (read-line)]
      (when-let [match (re-matches re line)]
        (let [[x y bth hth metab] (map read-string (rest match))]
          (println (format fmt
                           metab
                           (- x 20) (+ x 20) (- y 20) (+ y 20)
                           x y
                           (+ x (* 2 (cos (to-rad bth))))
                           (+ y (* 2 (sin (to-rad bth))))
                           x y
                           (+ x (cos (to-rad (+ hth bth))))
                           (+ y (sin (to-rad (+ hth bth)))))))))
      (recur))))
```

Figure 1: Code implementing a three layer seeking network.

3 Architectures

Seven architectures were implemented and evaluated. For each architecture we present a detailed description of the design and mechanics of the architecture, an overview of the performance of the architecture and a rational is given for the inclusion of each new architectural element.

The seven architectures and their overall performances are shown in tables 1 and 2.

Table 1: Overview of life span by architecture

architecture	mean	standard deviation	max
stationary	1999	0	1999
random	1988.36	49.192577	2102
seeking	1712.23	177.23225	2435
food seeking	1694.03	336.17914	3016
hearing	1892.29	497.88254	3590
lazy	1908.12	526.2256	3752
picky	1904.15	600.96747	4444

Table 2: Overview of objects eaten by architecture

architecture	mean	standard deviation
stationary	0	0
random	0.22	0.43992653
seeking	2.52	4.7129307
food seeking	5.93	5.2996187
hearing	11.05	5.874745
lazy	9.84	6.344838
picky	10.83	7.4441423

3.1 Stationary

A stationary individuals which does not move or consume objects. This architecture and the following random architecture provide a baseline against which the effectiveness of all subsequent designs can be compared.

While this is the simplest possible control mechanism – the lack of any control – this architecture does have one distinct advantage. While all other architectures are forced to take some movement which incurs a disadvantage in terms of increased metabolic rate this architecture maintains an optimally small metabolic rate. This fact enabled this simplest possible architecture to boast the best mean life span of 1999 time steps or roughly 20 flatworld seconds. A graph of the average metabolism as a function of flatworld seconds is given in Figure 2.

Despite all subsequent architectures falling short of this baseline architecture in terms of mean life span, many of the changes were justified through

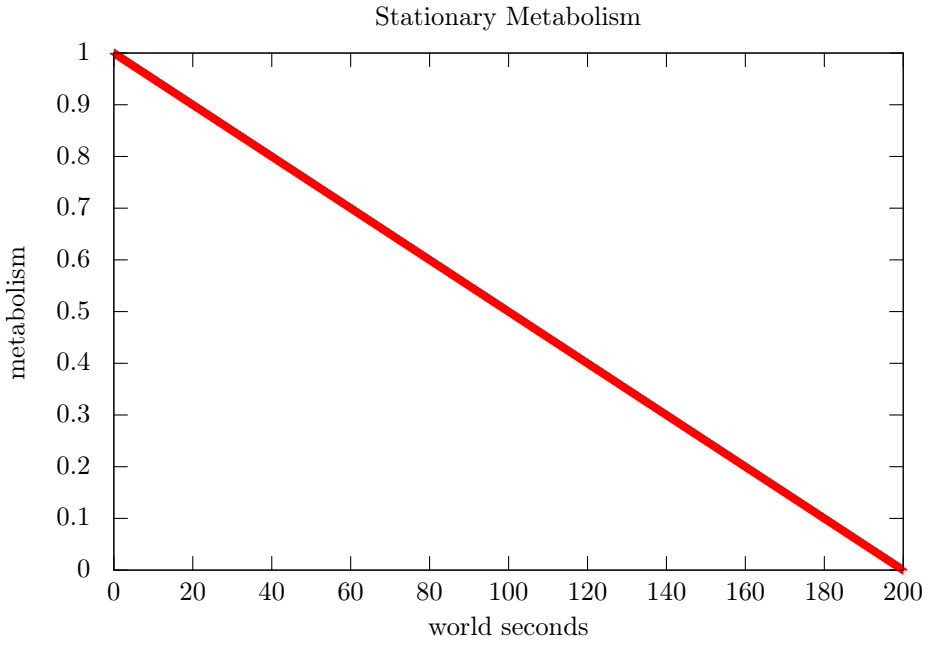

Figure 2: Metabolism of a stationary agent over time.

the potential they created for much longer lifespans. All subsequent architectures were able to outperform the stationary architecture in multiple runs.

3.2 Random

Perhaps a more reasonable choice of baseline architecture is the random architecture. Like nearly all subsequent architectures this agent moves constantly. The only control exerted by this agent is through turning by a random either right or left by a random value less than or equal to five degrees. This agent – and all subsequent agents unless explicitly stated otherwise – will automatically eat every food item with which it comes into contact.

The metabolism over time of these random agents is shown in Figure 3, and the average life spans of these agents are given in table 3.

3.3 Seeking

While the mean performance of the random agent is better than all subsequent individuals, the runs displayed very little variation and were uninter-

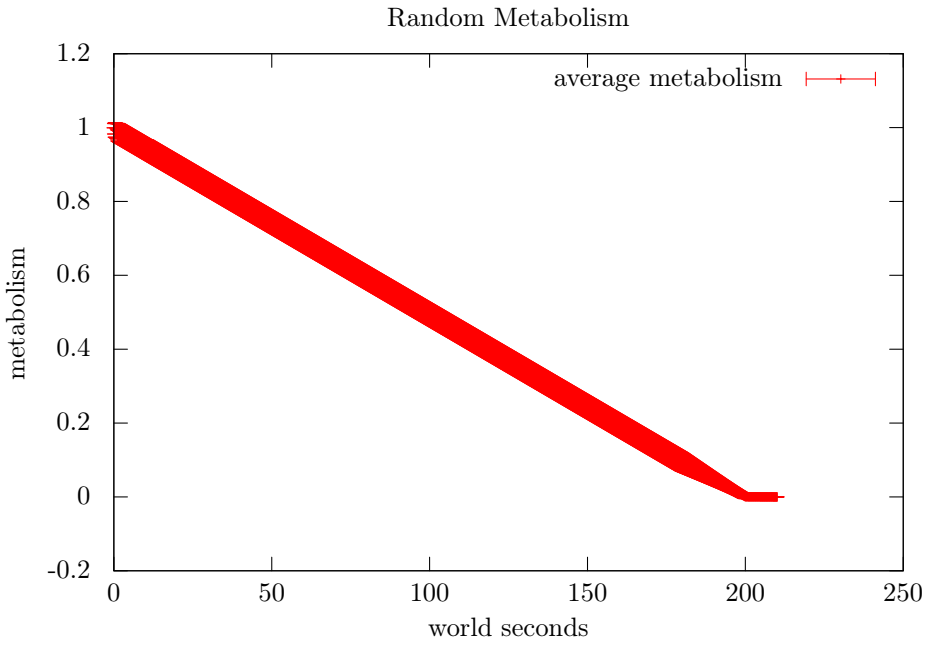

Figure 3: Average metabolism of a Random agent over time with standard deviation.

Table 3: Life span metrics

mean	standard deviation	max
1988.36	49.192577	2102

esting. To develop interesting behavior and increased interaction with the environment, the agent was enhanced with the ability to seek out objects in its environment.

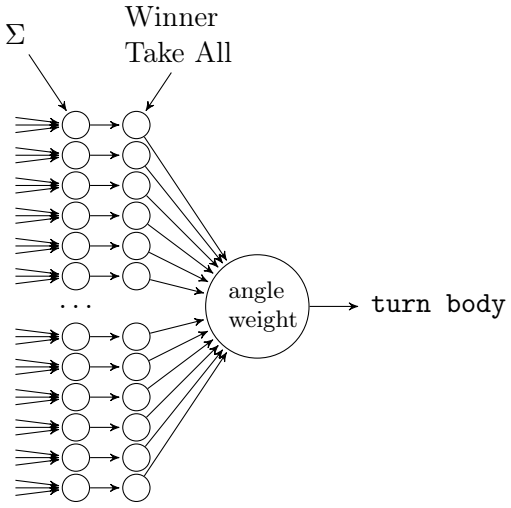

Figure 4: Three layer neural network. The first layer sums inputs over all bands for each eye, the second layer is a winner take all network which selects the brightest eye, and the third layer is a single neuron which weights the brightest eye by its angle around the head.

This object seeking behavior is accomplished through the use of a three layer neural network as shown in 4. This network uses the visual sensors of the agent to turn the agents body towards objects. The first layer of this network contains one neuron for each visual sensor on the agent, this neuron sums the values of all three bands of the visual spectrum. This first layer of neurons then feeds into a second layer of neurons connected in a "winner takes all" network, which activates only the brightest of the input neurons corresponding to the eye which is sensing the most light across all bands. The last layer of this neuron network consists of a single neuron which

```
[;; for each eye, sum over all bands
 (with-meta (repeat 31 {:phi (fn [n x]
                               (reduce + (map * (n :weights) x)))
                        :weights [1 1 1]})
   {:nn :direct})
 ;; winner take all for brightest receptor, if none then turn
 {:phi winner-take-all:phi
  :weights (repeat 32 1)
  :min 0.001}
 ;; weight winning receptor by its degrees around the head
 {:phi (fn [n x] (reduce + (map * (n :weights) x)))
  :weights (vec (map #(fw/eye-position a %) (range 31)))}]
```

Figure 5: Code implementing a three layer seeking network.

is connected to every neuron in the "winner take all" layer. Each of these connections is weighted with the angle in degrees of the corresponding sensor. The end result of this network is to return the angle of the brightest neuron effectively steering the agent into the light. The source code implementing this network is shown in Figure 5.

Table 4: Life span metrics

mean	standard deviation	max
1712.23	177.23225	2435

The increased interaction of these agents with their environment can be measured directly by counting the number of objects eaten by each agent. In this case the average number of objects eaten was 2.52 with a standard deviation of 4.7129307, this is significantly more than the 0.22 mean objects eaten by random runs. Metrics of the life span of the agent are shown in Figure 6 and Table 4.

3.4 Food Seeking

The previous architecture is successful at finding objects in flatworld, however despite this success agents controlled using the pure light seeking setup lived significantly shorter lives than randomly turning agents. The immediate explanation for this is that the results of eating poison objects is counteracting the results of eating food. This architecture seeks to resolve this

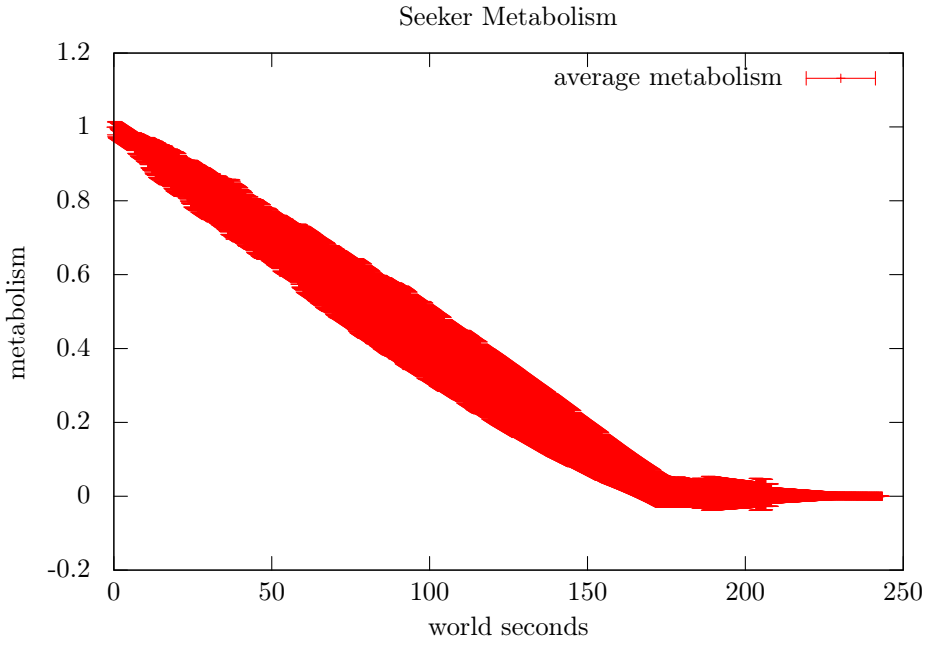

Figure 6: Average metabolism of a Seeking agent over time with standard deviation.

problem through differentiation of food sources, steering the agent towards food rather than poison.

The resulting architecture is shown in Figure 7. This network alters that presented above in Figure 4 through changing the behavior of neurons in the first layer – those that combine the three bands of input. Rather than simply summing their inputs these neurons are now implemented using a perceptron which has been trained to identify the combination of visual bands which indicate food items as opposed to either neutral or poison items. The source code implementing this network is shown in Figure 8.

To train the food classification perceptron the agent checks upon eating each object if its internal charge has increased, decreased or stayed even. The direction of change in internal charge is used to determine the desired value of the output of the neuron, namely $+1$ for food and -1 for poison. The brightest eye is then trained using this desired value and its most recent inputs.

This approach relies solely on visual inputs perceived immediately before consumption to guide all agent vision. This is based on the assumption

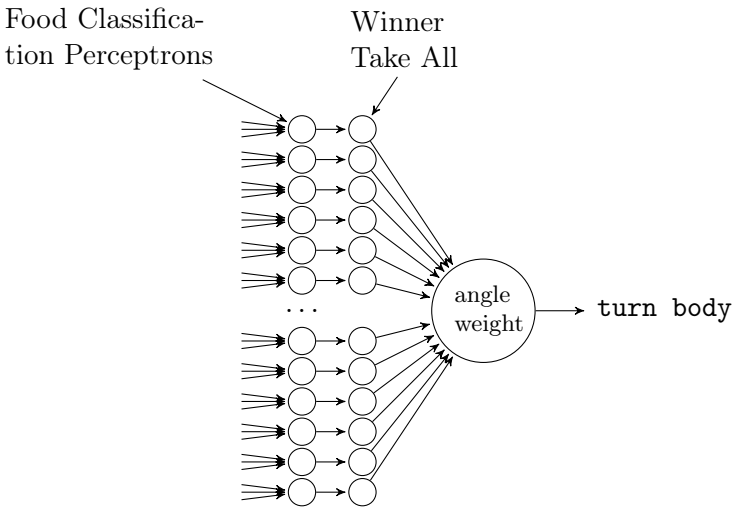

Figure 7: Three layer neural network. The first layer classifies its inputs over all bands for each eye as either food or not food, the second layer is a winner take all network which selects the eye with the strongest food classification, and the third layer is a single neuron which weights the winner by its angle around the head.

that the color signature of different objects is roughly invariant over various distances.

The results of this approach are presented in Figure 9 which shows the average metabolism of agents as a function of flatworld seconds, in table 5 which shows the average life span of these agents and also in table 6 which shows the average number of objects eaten by agent.

Table 5: Life span metrics for the food seeking agent

mean	standard deviation	max
1694.03	336.17914	3016

3.5 Hearing

Through observing runs (see Section 5.2) in the previous architectures one common reason for poor agent performance is wandering away from the objects. Visual sensors have a narrow range of perceptron meaning that

```
(def food-selector
  {:phi (fn [n x] (reduce + (map * (n :weights) x)))
   :weights [0.1608399379998446
             -0.3624796495889312
             -0.04009846952976671]
   :learn perceptron:learn
   :train (fn [n delta]
            (assoc n :weights (vec (map + (n :weights) delta))))
   :eta 0.1})

(def eye-net
  [;; winner take all for brightest receptor
   {:phi winner-take-all:phi
    :weights (repeat 32 1)
    :min 0.001}
   ;; weight winning receptor by its degrees around the head
   {:phi (fn [n x] (reduce + (map * (n :weights) x)))
    :weights (vec (map #(fw/eye-position a %) (range 31)))}])

(run eye-net (run (with-meta (repeat 31 food-selector)
                    {:nn :direct}) eyes))
```

Figure 8: Code implementing a three layer seeking network.

Table 6: Eaten metrics for the food seeking agent

mean	standard deviation
5.93	5.2996187

when an agent is pointed away from all objects in the world there is no sensor information available to the agent. To address this concern the aural sensors of the agent are used which have a larger range of perception.

This architecture embeds the previous network (referred to as "Food Seeker" in Figure 10) into a larger multipart network which combines both visual and aural sensor inputs to steer the agent. The same strategy employed in previous architectures consisting of an input layer followed by a winner take all layer and finally a layer which weights the winner by its position in degrees, is applied to the two aural inputs of the agent. Then the visual and aural inputs are feed into a short circuited `or` (neurons "a" "b" and "c") which returns the output of the visual portion of the network when available and the output of the aural portion of the output when the visual

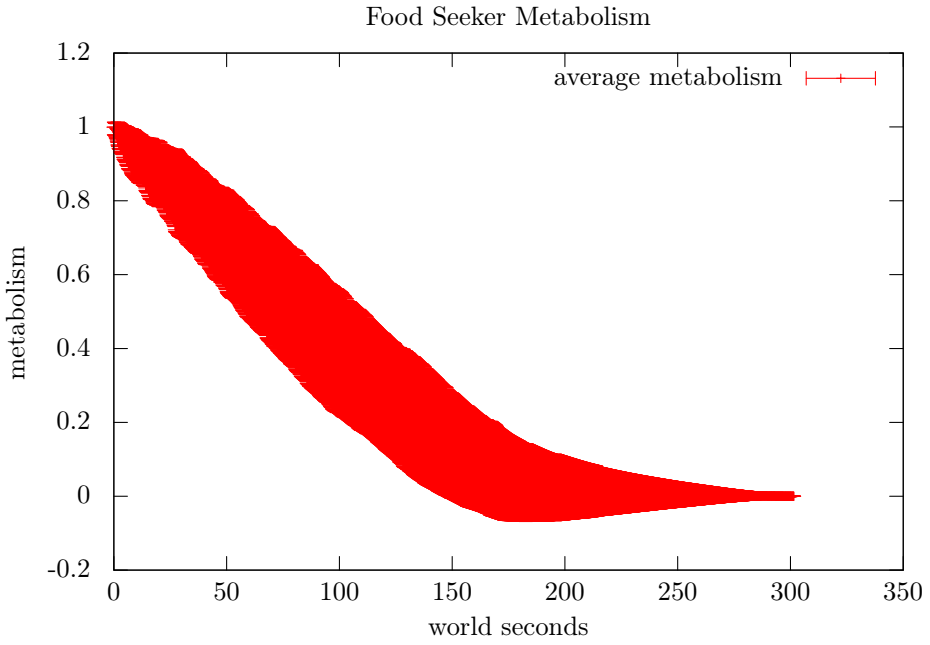

Figure 9: Average metabolism of a Seeking agent over time with standard deviation.

is not available. The code implementing this system is shown in figure 11.

This architecture increases the longevity of the average agent through helping to keep agents from wandering off away from the center of objects in the world. This can be seen in Figure 12 and Tables 7 and 8.

Table 7: Life span metrics

mean	standard deviation	max
1892.29	497.88254	3590

3.6 Lazy

This agent builds upon the previous agent design with the addition of a *lazy* controller which inhibits movement unless the agent has less than 0.9 metabolism. This ensures that the agent does not waste energy unless there is a need for finding food.

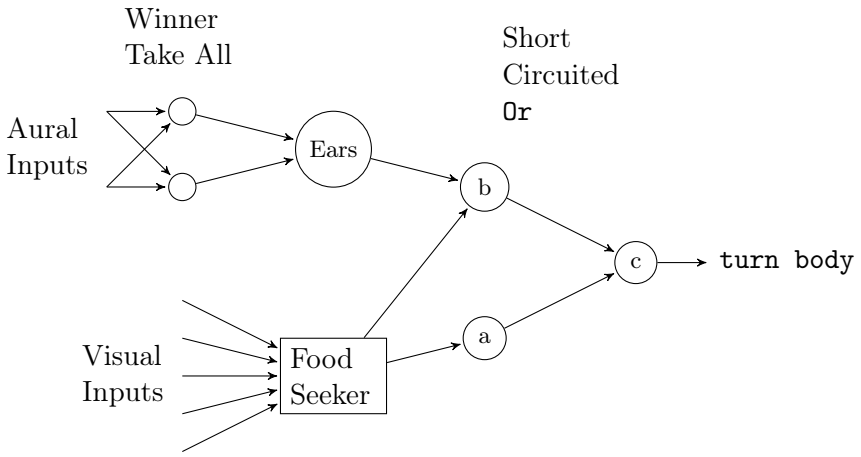

Figure 10: Combination of visual and aural inputs. The box labeled "Food Seeker" represents the entire network from Figure 7. A similar "winner take all" layer is used to select the ear which hears the most sound, this winner is then weighted by its position on the head. The neurons "a" "b" and "c" are used to return the angle returned by "Food Seeker" or to return the angle returned by "Ears" if no angle is produced by "Food Seeker".

Table 8: Eaten objects metrics

mean	standard deviation
11.05	5.874745

The implementation of this *lazy* addition to the network is shown in Figure 13. The existing neural network is retained intact and another network is added. This new network connected the agents internal metabolic charge sensor to the control the movement of the agent. If the charge is less than 0.9 then the neuron turns on body movement, and otherwise body movement is turned off. The code just for the new portion of this network is shown in Figure 14.

Despite the simplicity of this addition it has a significant effect on the fitness of the resulting agent. As can be seen in Figure 18 and in Table 9 in which both the mean and max life spans resulting from the architecture are a significant improvement over the previous architecture. As would be expected from reducing the activity of the agent this change results in slightly

less object consumption as shown in Table 10.

Table 9: Life span metrics for lazy agent

mean	standard deviation	max
1908.12	526.2256	3752

Table 10: Object eaten metrics for lazy agent

mean	standard deviation
9.84	6.344838

3.7 Picky

This agent adds a perceptron which attempts to classify food sources on a sliding scale from poison through neutral to food and only eats those objects on the food portion of the scale. All previous architectures ate all touched objects.

Figure 16 shows the architecture of this new picky eating network. This addition shown as the third separate component of the network is uses visual inputs to eat or ignore objects which the agent is in contact with. This new network uses a "winner take all" network similar to that introduced in previous networks to select the eye which most strongly sees the object. The three bands of this eye are then passed to a perceptron trained to identify food objects. The output of this perceptron is used to either eat or ignore the object at hand. The code for this new portion of the network is shown in figure 17.

The results of this final architecture are shown in Figure 18 and in Tables 11 and 12.

Table 11: Life span metrics for the picky agent

mean	standard deviation	max
1904.15	600.96747	4444

Table 12: Objects eaten by the picky agent

mean	standard deviation
10.83	7.4441423

4 Discussion

4.1 Technical Design

The single most important design decision made in the course of this investigation was the use of a neural network engine implemented in a separate programming language and execution environment than the flatworld simulation. This choice carried with it a number of challenges and benefits.

The main benefit of this engine included the ability to quickly and easily specify neural architectures, the actual design and implementation of neural architectures represented a small portion of the time spent in the execution of this work. There were also ancillary benefits associated with the interactive nature of the lisp programming language, including the ability to pragmatically inspect runs during execution, the ability to adjust parameters and perform runs from the REPL, and integration of lisp with a reproducible research framework [4] allowing for programmatic generation of all metrics and figures in this report.

The drawbacks of using a neural network engine implemented in a separate programming language were significant. The process of defining the interface barrier between the C simulation and the Lisp controller was on going. Beyond implementing control and sensing primitives in C, it was often preferable to provide C wrappers around built-in flatworld functions for example returned actual float values rather than float array pointers which can be difficult to safely access from within Java. Having this interface often in flux and buggy had a significant impact on the design aspect of this work. One specific bug in the interface resulted in three attempted solutions through agent design before the buggy agent behavior was discovered to be caused by the C interface.

Once the interface was solidified and working the benefits of using a flexible programming language for the specification of neural networks were sufficient to justify the effort of integration.

[4] http://orgmode.org/

4.2 Architectures

Although none of the architectures presented in Section 3 were able to best the mean life span of the motionless agent, the benefits of the subsequent design choices can be seen in the progression of steady increasing mean life span from the simplest *seeking* agent onward, and in the steadily increasing maximum life span. Along with generally increasing performance, the subsequent architectures demonstrated increasing spread of performance indicating that the increasingly complex architectures lead to increasingly unpredictable and variable agent behaviors.

Surprisingly the two architectural changes of questionable value were also the two which required the training of perceptrons. These two additions (the food identifier for long range movement and the food identifier for near range eating) both represented decreases in the average life span of agents from their predecessors.

In the case of the long range identification system this can be attributed to the assumption that the visual spectrum of objects is invariant over distances, if this assumption is not valid then the poor performance of the system is not surprising.

In the case of the short range identification system, the overall performance impact was negligible. This may be due to the way in which the system was applied, namely using only the brightest eye at the time of object consumption to determine object classification. A system in which multiple eyes were used could possibly have been more robust to differences resulting from the orientation of the agent and the object on contact.

The addition of hearing to the agent for purposes of global orientation towards the center of objects in the plane had a significantly positive impact on agent performance. Even with this impact it is likely that this modification could have been improved. Through manual inspection of runs it was often the case that when turned directly away from the object field, the aural inputs to the agent would balance s.t. it was impossible for symmetry to break and the agent to turn to any particular side to return to the objects. It is not clear whether this behavior is to be expected from the design of the hearing apparatus or if this is due to bugs in the C integration.

A general issue in the execution of this work was the lack of certainly in the presence of anomalous behavior whether the to investigate the code implementing the simulated environment or the agent. At different points each was to blame, and the lack of certainty inhibited the design process.

5 Appendix

5.1 Source Code

The source code for the neural network DSL is available online [5].

The source code for the experiments presented in this paper is also available online [6]. This includes the code used to perform all experiments included in this paper, the text of this paper as well as the code used to generate all metrics presented in this paper. Included in this package are both working notes on the integration of the neural network engine and the flatworld simulation as well as the actual source code accomplishing the integration.

5.2 Example Runs

Recordings of example runs are provided online [7]. One run is given for each of the neural frameworks. Both the agent performance and the sophistication of the graphing engine can be seen to improve through over the course of the 5 selected runs.

[5] http://repo.or.cz/w/neural-net.git
[6] http://gitweb.adaptive.cs.unm.edu/nn-flatworld.git
[7] http://cs.unm.edu/~eschulte/data/nn-flatworld-runs/

```
(def food-selector
  {:phi (fn [n x] (reduce + (map * (n :weights) x)))
   :weights [0.16213777155207942
             -0.3622200826925108
             -0.03983890263334627]
   :learn perceptron:learn
   :train (fn [n delta]
            (assoc n :weights (vec (map + (n :weights) delta))))
   :eta 0.1})

(def eye-net
  [;; winner take all for brightest receptor
   {:phi winner-take-all:phi
    :weights (repeat 32 1)
    :min 0.001}
   ;; weight winning receptor by its degrees around the head
   {:phi (fn [n x] (reduce + (map * (n :weights) x)))
    :weights (vec (map #(fw/eye-position a %) (range 31)))}])

(def ear-net
  {:phi (fn [n x]
          ((fn [act]
             (cond
              (> act 25)  10
              (> act 0)    1
              (< act 25) -10
              (< act  0)  -1
              true 0))
           (reduce + (map * (n :weights) x))))
   :weights [100 -100]})

(let [eye-dir (run eye-net (run (with-meta (repeat 31 food-selector)
                                           {:nn :direct}) eyes))
      ear-dir (run ear-net ears)]
  (if (not (= eye-dir 0)) eye-dir ear-dir))
```

Figure 11: Code implementing a three layer seeking network.

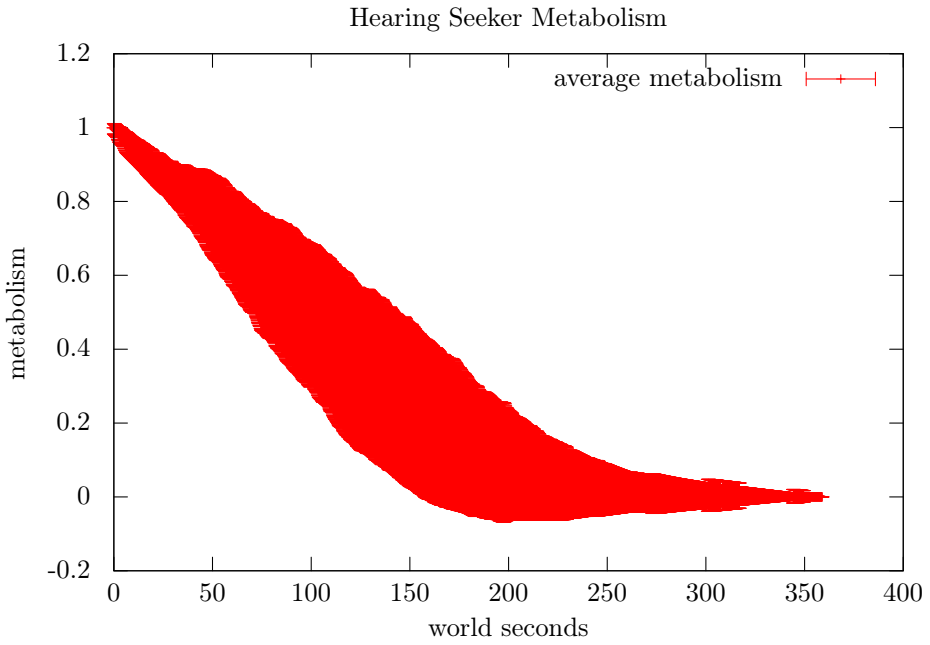

Figure 12: Average metabolism of a hearing and seeking agent over time with standard deviation.

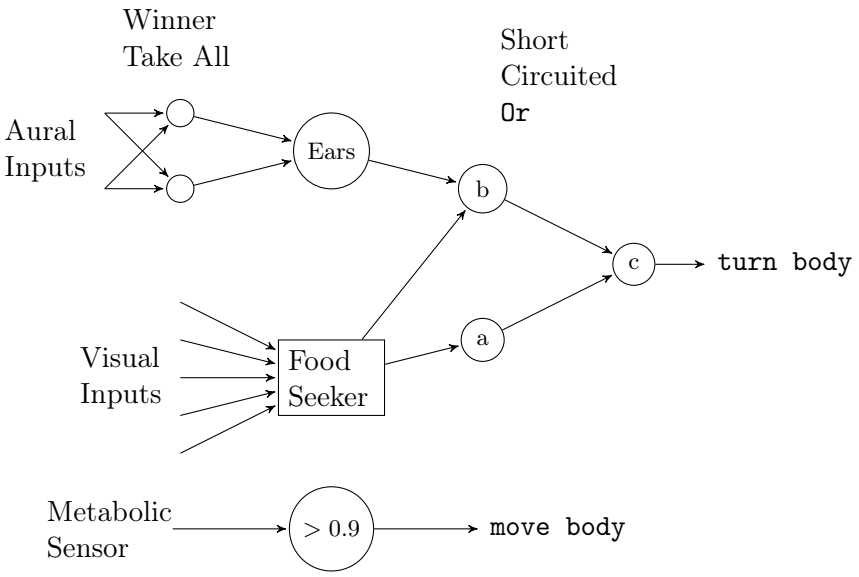

Figure 13: See the caption of Figure 10 for a description of the upper network. The lower network is responsible for moving the agent whenever its metabolic charge is lower than 0.9 and for holding the agent's body still otherwise.

```
{:phi (fn [n x] (if (> (reduce + (map * (n :weights) x)) 0.9) false true))
 :weights [1]}
```

Figure 14: Code implementing a three layer seeking network.

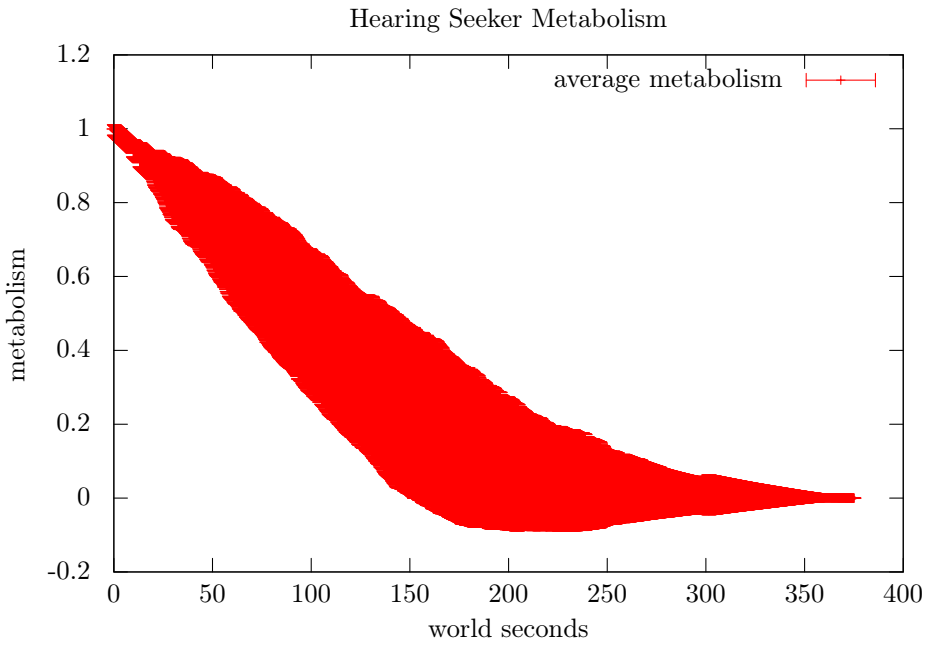

Figure 15: Average metabolism over time of a hearing and seeking agent which only moves when its metabolism is below 0.9.

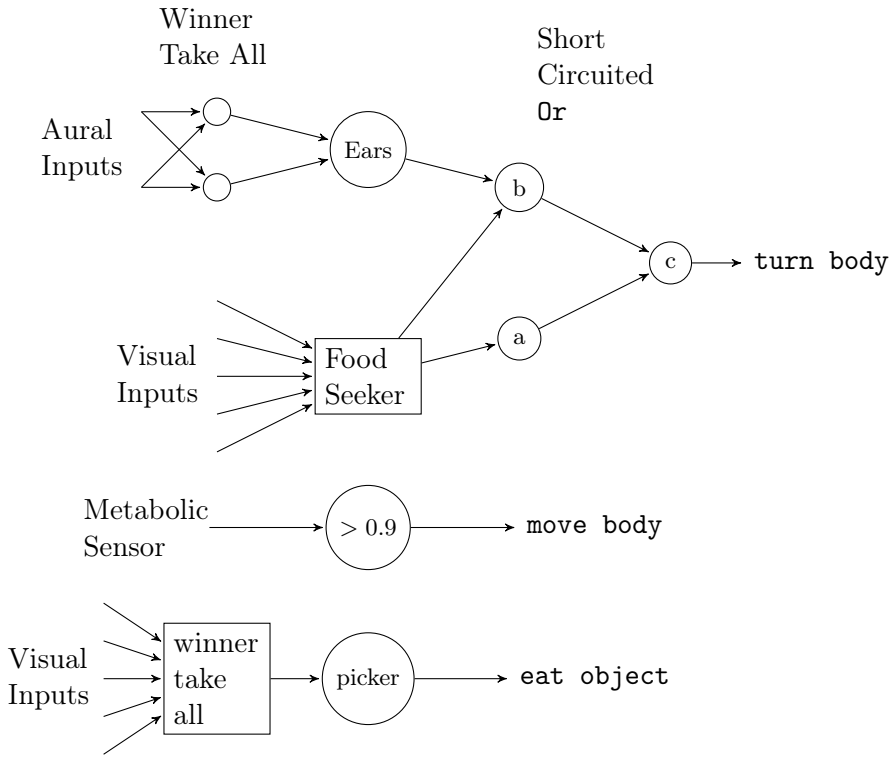

Figure 16: See the captions of Figures 10 and 13 for a descriptions of the upper two networks. The lowest network is responsible for selectively eating only those objects which appear to be food. The same visual inputs used for steering are used in this selection process. The brightest eye is found using a "winner take all" layer, the three color band inputs to this eye are then fed to a perceptron which has been trained to identity food, poison and neutral objects on a sliding scale. If this perceptron classifies the inputs on the food side of the scale, then the agent eats the object at hand.

```
(def food-picker
  {:phi (fn [n x] (reduce + (map * (n :weights) x)))
   :weights [0.9345053465185924 -0.2077465774877346 0.11463460257142993]
   :learn perceptron:learn
   :train (fn [n delta]
            (assoc n :weights (vec (map + (n :weights) delta))))
   :eta 0.001})

(when (> (run food-picker eye) 0)
  (fw/agent-eat (:raw world) a))
```

Figure 17: Code implementing a three layer seeking network.

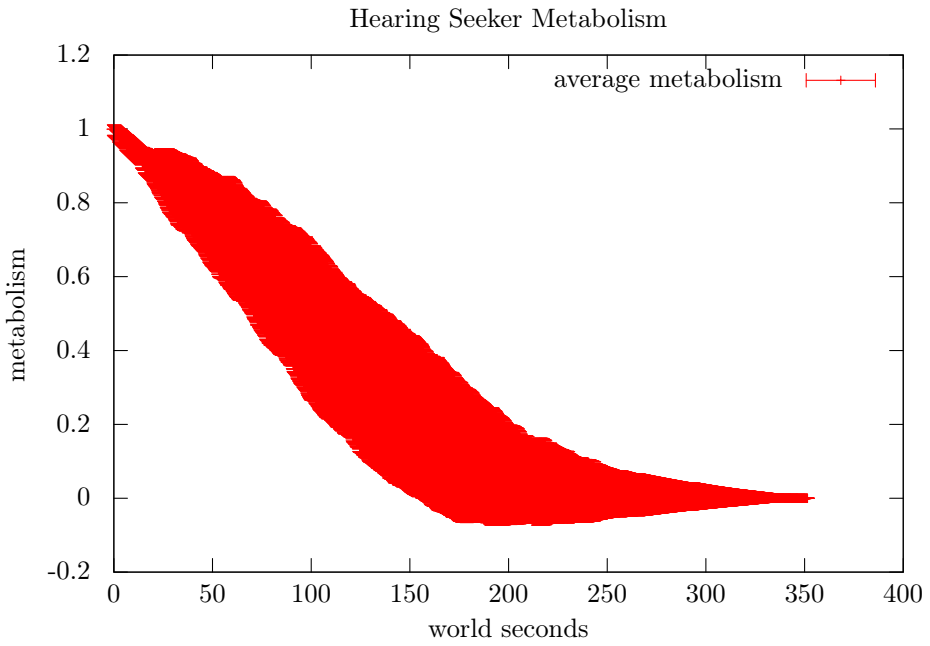

Figure 18: Average metabolism over time of a hearing and seeking agent which only moves when its metabolism is below 0.9.