ECE/CPSC 352
Software Design Exercise #1:
Implementing Hopfield Networks in `ocaml`

Blackboard submission only

Assigned 1-23-2014

Due 2-13-2014 11:59PM

Contents

1 Preface

The topic of artificial neural networks (ANNs) is both relevant and popular. The overall objective of SDE #1 is to implement and test a version of an recurrent ANN in `ocaml`. All work is to be done on a linux platform and in `ocaml`, using a purely functional paradigm. Note that you are not free to choose any available construct or feature in `ocaml`; Section 7 outlines the restrictions. Significant in-class discussion will accompany this SDE; you would probably not want to miss it.

2 Resources

As discussed in class, it would be foolish to attempt this SDE without carefully exploring:

1. The text, especially the many ocaml examples in Chapter 11;
2. The background provided in this document;
3. The `ocaml` manual; and
4. In-class discussions.

3 ANN Recurrent Network Implementation and Training (Background)

3.1 A Simple, Single Artificial Neuron (Unit)

Many artificial neural unit models involve two important processes:

1. Forming a *unit net activation*, denoted by the scalar net_i for unit i, by (somehow) combining unit inputs. Most commonly, a weighted linear input combination (WLIC) is used:

$$net_i = \Sigma_j w_{ij} o_j = \underline{w}_i^T \underline{o} \tag{1}$$

   In Equation 1, $\underline{\mathrm{w}}$ and $\underline{\mathrm{o}}$ are (column) vectors of unit weights and outputs, respectively. Thus, Equation 1 represents an inner product computation.

2. Mapping this activation value into the artificial unit output. Here you will implement the Hopfield activation or 'squashing' function.

Figure 1 illustrates the generic two-part unit model concept.

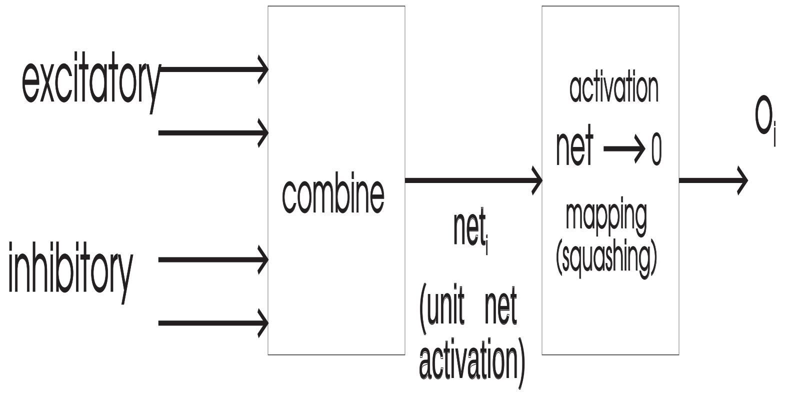

Figure 1: Two-Part Model for Unit Input Combination and Output Formation

3.2 The Recurrent Network

The 'Hopfield' net, honors John Hopfield of Caltech, who seems to have popularized the strategy. You are implementing a 'Hopfield' net.

Recurrent networks are networks which have closed loops in the network topology. Taking a group of isolated units and allowing full interconnection between all units yields a recurrent network structure. The outputs of all units comprise the network (or system) state. A recurrent ANN maps *states* into *states*.

3.2.1 Network Parameters

The following variables are defined:

o_i : the output state of the i^{th} neuron. Therefore, the vector $\underline{o}$ represents the outputs of all units and therefore the state of the entire network.

α_i : the activation threshold of the i^{th} neuron.

w_{ij}: the interconnection weight, i.e., the strength of the connection FROM the output of neuron j TO neuron i. Thus, $\Sigma_j w_{ij} o_j$ is the total input or activation (net_i) to neuron i. Assume $w_{ij} \in R$. With the constraints developed below, for a d-unit network there are $\frac{d(d-1)}{2}$ possibly nonzero and unique network weights.

On this basis, W is the overall system weight matrix, i.e.,

$$W = [w_{ij}] \tag{2}$$

The i^{th} row of W corresponds to the weight set of the i^{th} unit.

In the Hopfield network, every neuron is allowed to be connected to all other neurons, although the value of w_{ij} varies (it may also be 0 to indicate no unit interconnection). To avoid false reinforcement of a neuron state, the constraint $w_{ii} = 0$ is employed. *Thus, no self-feedback is allowed in the Hopfield formulation.* Equivalently, the diagonal entries of W are all identically 0.

3.2.2 Hopfield (-1,1) Unit Characteristic

Threshold or hardlimiter unit characteristics are commonly used, although in some cases this firing characteristic requires careful interpretation, since the network behavior may be different when $\alpha_i = 0^-$ vs. $\alpha_i = 0^+$. Hopfield's original work suggested a slight modification of the hardlimiter characteristic. Specifically, in the case of $net_i = \alpha_i$, the unit output is unchanged from its previous value. We refer to this as the 'leave it alone' characteristic, given by:

$$o_i = \begin{cases} 1 & if \sum_{j;\ j \neq i} w_{ij} o_j > \alpha_i \\ o_i \quad (previous\ value) & if \sum_{j;\ j \neq i} w_{ij} o_j = \alpha_i \\ -1 & otherwise \end{cases} \tag{3}$$

Notice from Equation 3, the neuron activation characteristic is nonlinear. Commonly, the threshold $\alpha_i = 0$.

3.2.3 Weight (Storage) Prescriptions for Desired Stored States

The storage prescription shown below leads to symmetric network interconnection matrices with zero diagonal entries. This is a popular form.

Units Outputs $\in \{-1, 1\}$. We treat the case of $\{-1, 1\}$ unit outputs and $\alpha_i = 0$. Given a set of desired stored states $\underline{o}^s, s = 1, 2, ...n$, from the the training set (stored states) $H = \{\underline{o}^1, \underline{o}^2, \ldots, \underline{o}^n\}$. The prescription given by Hopfield is:

$$w_{ij} = \sum_{s=1}^{n} o_i^s o_j^s \qquad i \neq j \tag{4}$$

with the additional 'no-self-activation' constraint

$$w_{ii} = 0 \tag{5}$$

Energy. It is paramount to note that network equilibrium states are related to minima of the following energy function:

$$E = -\frac{1}{2} \sum \sum_{i \neq j} w_{ij} o_i o_j = -\frac{1}{2} \underline{o}^T W \underline{o} \tag{6}$$

Assessment.

1. When employing the Hopfield network in certain constraint satisfaction problems, the energy of a state can be interpreted as the extent to which a combination of hypotheses or instantiations fit the underlying neural-formulated model. Thus, low energy values indicate a good level of constraint satisfaction.

2. The energy of a Hopfield network **cannot increase**, and (almost always) decreases.

4 `ocaml` Data Formats and Representations

4.1 Disclaimer

I've broken up some of the design into guided steps, as shown below. **You must design and implement these functions solely in ocaml and with the interfaces (esp. arguments) and behavior shown.** Of course, there may also be other functions to be developed. Your ocaml implementation must include the functions described in Sections 4.2.

Notes:

1. The entire effort must use only `ocaml`.
2. Pay particular attention to the signatures and names of functions shown and the data structures used.
3. Recall **ocaml is case sensitive.**
4. Notice you must work with the specified list data structures, i.e., you cannot redefine the input and output formats (or signatures) of these functions to suit your needs or desires.
5. Pay particular attention to the restrictions in Section 7.

4.2 General Representation Issues

- The system state vector, $\underline{o}$, is an `ocaml` float list.
- The `ocaml` data format for the entire set of network weights, i.e. matrix W, is that of an `ocaml` list of lists of float format. `ocaml` calls this 'float list list'. The i^{th} list corresponds to row i of W and is a float list indicating the weights for the i^{th} unit. **Note the number of units in the Hopfield network is arbitrary.** Your software must accommodate this.

Example of the Representation.

```
(** some 4-D data for simulation/debugging *)

let os1 = [1.0; -1.0; 1.0; -1.0];;

let os2 = [-1.0; -1.0; 1.0; -1.0];;

let os3 = [-1.0; -1.0; 1.0; 1.0];;

# let w=hopTrain([os1]);;
val w : float list list =
  [[0.; -1.; 1.; -1.]; [-1.; 0.; -1.; 1.]; [1.; -1.; 0.; -1.];
   [-1.; 1.; -1.; 0.]]
```

5 Functions for Training and Implementing the Recurrent Network in `ocaml`

Prototypes are shown below. **Note all functions use a tupled argument interface.** You should also study the examples in Section 6.

5.1 Single Unit Related Functions

```
(** netUnit returns net activation for a single unit *)

netUnit(inputs, weights)

(* net activation computation for entire network; returns vector (list) of unit activations *)

netAll(state, weightMatrix)

(** Hopfield activation function for single (-1,1) unit; returns unit output *)

hop11Activation(net,alpha,oldo)
```

5.2 State Propagation Functions

```
(** Next state computation; returns next state *)

nextState(currentState, weightMatrix, alpha)

(**  Update state N time steps; returns network state after n time steps*)

updateN(currentState, weightMatrix, alpha, n)

(** Function findsEquilibrium returns true if network reaches an equilibruim state,
    false otherwise *)

findsEquilibrium(initialState, weightMatrix, alpha, range)
```

5.3 Implementing the Network Storage Prescription in `ocaml`

```
(** This returns weight matrix for only one stored state *)

hopTrainAstate(astate)

(** Weight matrix for a list of stored states *)

hopTrain(allStates)
```

5.4 Energy Function Implementation

This function corresponds to Equation 6.

```
(** state energy *)

energy(state,weightMatrix)
```

5.5 Function Signatures

Here's the signatures of the 9 `ocaml` functions you must design, implement and test:

```
val netUnit : float list * float list -> float = <fun>
val netAll : float list * float list list -> float list = <fun>
val hop11Activation : 'a * 'a * float -> float = <fun>
val hopTrainAstate : float list -> float list list = <fun>
val hopTrain : float list list -> float list list = <fun>
val nextState : float list * float list list * float -> float list = <fun>
val energy : float list * float list list -> float = <fun>
val updateN : float list * float list list * float * int -> float list =
  <fun>
val findsEquilibrium : float list * float list list * float * int -> bool =
  <fun>
```

6 Examples

You can easily check the values in these examples by hand.

```
# let w=hopTrain([os1]);;
val w : float list list =
  [[0.; -1.; 1.; -1.]; [-1.; 0.; -1.; 1.]; [1.; -1.; 0.; -1.];
   [-1.; 1.; -1.; 0.]]
```

```
# netAll(os1,w);;
- : float list = [3.; -3.; 3.; -3.]

# nextState(os1,w,0.0);;
- : float list = [1.; -1.; 1.; -1.]

# energy(os1,w);;
- : float = -6.

# energy(os2,w);;
- : float = -0.

# nextState(os2,w,0.0);;
- : float list = [1.; -1.; 1.; -1.]

# os2;;
- : float list = [-1.; -1.; 1.; -1.]

# energy(nextState(os2,w,0.0),w);;
- : float = -6.

# let w2 = hopTrain([os1;os2;os3]);;
val w2 : float list list =
  [[0.; 1.; -1.; -1.]; [1.; 0.; -3.; 1.]; [-1.; -3.; 0.; -1.];
   [-1.; 1.; -1.; 0.]]

# nextState(os1,w2,0.0);;
- : float list = [-1.; -1.; 1.; -1.]

# nextState(nextState(os1,w2,0.0),w2,0.0);;
- : float list = [-1.; -1.; 1.; -1.]

# energy(os1,w2);;
- : float = -4.

# energy(os2,w2);;
- : float = -6.

# energy(os3,w2);;
- : float = -4.

# updateN(os1,w,0.0,2);;
- : float list = [-1.; -1.; 1.; -1.]
```

```
# updateN(os1,w,0.0,0);;
- : float list = [1.; -1.; 1.; -1.]

# updateN(os1,w,0.0,1);;
- : float list = [-1.; -1.; 1.; -1.]

# energy(os1,w);;
- : float = -4.

# energy(updateN(os1,w,0.0,1),w);;
- : float = -6.

# findsEquilibrium(os1,w,0.0,10);;
- : bool = true

# findsEquilibrium(os3,w,0.0,10);;
- : bool = true

(* note the change of network size (dimension) below *)

# let we=[[0.0;-1.0];[-1.0;0.0]];;
val we : float list list = [[0.; -1.]; [-1.; 0.]]

# let oi=[-1.0;-1.0];;
val oi : float list = [-1.; -1.]

# updateN(oi,we,0.0,1);;
- : float list = [1.; 1.]

# updateN(oi,we,0.0,2);;
- : float list = [-1.; -1.]

# updateN(oi,we,0.0,3);;
- : float list = [1.; 1.]

# updateN(oi,we,0.0,4);;
- : float list = [-1.; -1.]

# findsEquilibrium(oi,we,0.0,2);;
- : bool = false

# findsEquilibrium(oi,we,0.0,5);;
- : bool = false

# findsEquilibrium(oi,we,0.0,50);;
```

```
- : bool = false

# findsEquilibrium(oi,we,0.0,500);;
- : bool = false
```

More examples (some are redundant) follow:

```
# #use"hopfield.caml";;
val oi : float list = [-1.; -1.]
val we : float list list = [[0.; -1.]; [-1.; 0.]]
val os1 : float list = [1.; -1.; 1.; -1.]
val os2 : float list = [-1.; -1.; 1.; -1.]
val os3 : float list = [-1.; -1.; 1.; 1.]
val os4 : float list = [1.; 1.; 1.; 1.]
val os5 : float list = [-1.; -1.; -1.; -1.]
val os6 : float list = [1.; 1.; -1.; -1.]
val tsr : float list list =
  [[1.; -1.; 1.; -1.]; [1.; -1.; 1.; -1.]; [1.; -1.; 1.; -1.];
   [-1.; -1.; 1.; -1.]; [-1.; -1.; 1.; 1.]]

# let w=hopTrain([os1]);;
val w : float list list =
  [[0.; -1.; 1.; -1.]; [-1.; 0.; -1.; 1.]; [1.; -1.; 0.; -1.];
   [-1.; 1.; -1.; 0.]]

# netAll(os1,w);;
- : float list = [3.; -3.; 3.; -3.]

# netUnit(os1,(hd w));;
- : float = 3.

# hop11Activation(-3.0,0.0,1.0);;
- : float = -1.

# hop11Activation(0.0,0.0,-1.0);;
- : float = -1.

# hop11Activation(0.0,0.0,1.0);;
- : float = 1.

# hop11Activation(50.0,0.0,-1.0);;
- : float = 1.

# hop11Activation(50.0,50.0,-1.0);;
```

```
- : float = -1.

# nextState(os1,w,0.0);;
- : float list = [1.; -1.; 1.; -1.]

# os1;;
- : float list = [1.; -1.; 1.; -1.]

# nextState(os2,w,0.0);;
- : float list = [1.; -1.; 1.; -1.]

# nextState(os3,w,0.0);;
- : float list = [1.; 1.; -1.; -1.]

# nextState(nextState(os3,w,0.0),w,0.0);;
- : float list = [-1.; -1.; 1.; 1.]

# nextState(nextState(nextState(os3,w,0.0),w,0.0),w,0.0);;
- : float list = [1.; 1.; -1.; -1.]

# energy(os1,w);;
- : float = -6.

# energy(os2,w);;
- : float = -0.

# energy(os3,w);;
- : float = 2.

# netAll(os3,w);;
- : float list = [1.; 1.; -1.; -1.]

# os3;;
- : float list = [-1.; -1.; 1.; 1.]

# updateN(os3,w,0.0,1);;
- : float list = [1.; 1.; -1.; -1.]

# netAll(updateN(os3,w,0.0,1),w);;
- : float list = [-1.; -1.; 1.; 1.]

# updateN(os3,w,0.0,2);;
- : float list = [-1.; -1.; 1.; 1.]

# energy([1.0;1.0;-1.0;-1.0],w);;
```

```
- : float = 2.

# updateN(os3,w,0.0,3);;
- : float list = [1.; 1.; -1.; -1.]

# updateN(os3,w,0.0,4);;
- : float list = [-1.; -1.; 1.; 1.]

# findsEquilibrium(os3,w,0.0,100);;
- : bool = false

# let w2 = hopTrain([os1;os2;os3]);;
val w2 : float list list =
  [[0.; 1.; -1.; -1.]; [1.; 0.; -3.; 1.]; [-1.; -3.; 0.; -1.];
   [-1.; 1.; -1.; 0.]]

# energy(os1,w2);;
- : float = -4.

# energy(os2,w2);;
- : float = -6.

# energy(os3,w2);;
- : float = -4.

# findsEquilibrium(os1,w2,0.0,100);;
- : bool = true

# findsEquilibrium(os2,w2,0.0,100);;
- : bool = true

# findsEquilibrium(os3,w2,0.0,100);;
- : bool = true

# nextState(os1,w2,0.0);;
- : float list = [-1.; -1.; 1.; -1.]

# os1;;
- : float list = [1.; -1.; 1.; -1.]

# updateN(os1,w2,0.0,2);;
- : float list = [-1.; -1.; 1.; -1.]

# updateN(os1,w2,0.0,3);;
- : float list = [-1.; -1.; 1.; -1.]
```

```
# energy([-1.; -1.; 1.; -1.],w2);;
- : float = -6.

# os2;;
- : float list = [-1.; -1.; 1.; -1.]

# updateN(os2,w2,0.0,1);;
- : float list = [-1.; -1.; 1.; -1.]

# updateN(os3,w2,0.0,1);;
- : float list = [-1.; -1.; 1.; -1.]

# energy(os3,w2);;
- : float = -4.

# let w3 = hopTrain([os1;os2;os3;os4;os5;os6]);;
val w3 : float list list =
  [[0.; 4.; 0.; 0.]; [4.; 0.; -2.; 2.]; [0.; -2.; 0.; 2.]; [0.; 2.; 2.; 0.]]

# energy(os1,w3);;
- : float = 2.

# energy(os2,w3);;
- : float = -6.

# energy(os3,w3);;
- : float = -6.

# energy(os4,w3);;
- : float = -6.

# energy(os5,w3);;
- : float = -6.

# energy(os6,w3);;
- : float = -6.

# updateN(os1,w3,0.0,1);;
- : float list = [-1.; -1.; 1.; -1.]

# updateN(os1,w3,0.0,2);;
- : float list = [-1.; -1.; 1.; -1.]

# findsEquilibrium(os1,w3,0.0,100);;
```

```
- : bool = true

# findsEquilibrium(os2,w3,0.0,100);;
- : bool = true

# findsEquilibrium(os3,w3,0.0,100);;
- : bool = true

# findsEquilibrium(os4,w3,0.0,100);;
- : bool = true

# findsEquilibrium(os5,w3,0.0,100);;
- : bool = true

# findsEquilibrium(os6,w3,0.0,100);;
- : bool = true

# energy([-1.; -1.; 1.; -1.],w3);;
- : float = -6.

# os2 = [-1.; -1.; 1.; -1.];;
- : bool = true

# let w4 = hopTrain(tsr);;
val w4 : float list list =
  [[0.; -1.; 1.; -3.]; [-1.; 0.; -5.; 3.]; [1.; -5.; 0.; -3.];
   [-3.; 3.; -3.; 0.]]

# energy(os1,w4);;
- : float = -16.

# energy(os2,w4);;
- : float = -6.

# energy(os3,w4);;
- : float = -0.

# updateN(os1,w4,0.0,1);;
- : float list = [1.; -1.; 1.; -1.]

# updateN(os2,w4,0.0,1);;
- : float list = [1.; -1.; 1.; -1.]

# updateN(os3,w4,0.0,1);;
- : float list = [-1.; -1.; 1.; -1.]
```

```
# updateN(os3,w4,0.0,2);;
- : float list = [1.; -1.; 1.; -1.]

# updateN(os3,w4,0.0,3);;
- : float list = [1.; -1.; 1.; -1.]
```

And now for some low-dimensional character recognition:

```
# #use"chars.caml";;
val x : float list = [1.; -1.; 1.; -1.; 1.; -1.; 1.; -1.; 1.]
val o : float list = [1.; 1.; 1.; 1.; -1.; 1.; 1.; 1.; 1.]
val i : float list = [-1.; 1.; -1.; -1.; 1.; -1.; -1.; 1.; -1.]
val z : float list = [1.; 1.; 1.; -1.; 1.; -1.; 1.; 1.; 1.]
val px : float list = [1.; -1.; 1.; -1.; 1.; -1.; 1.; -1.; -1.]
val po : float list = [1.; 1.; 1.; -1.; -1.; 1.; 1.; 1.; 1.]
val pi : float list = [-1.; 1.; -1.; -1.; 1.; -1.; -1.; 1.; 1.]
val pz : float list = [1.; 1.; 1.; -1.; 1.; -1.; 1.; -1.; 1.]

# let w = hopTrain([x;o;i;z]);;
val w : float list list =
  [[0.; 0.; 4.; 0.; 0.; 0.; 4.; 0.; 4.];
   [0.; 0.; 0.; 0.; 0.; 0.; 0.; 4.; 0.];
   [4.; 0.; 0.; 0.; 0.; 0.; 4.; 0.; 4.];
   [0.; 0.; 0.; 0.; -4.; 4.; 0.; 0.; 0.];
   [0.; 0.; 0.; -4.; 0.; -4.; 0.; 0.; 0.];
   [0.; 0.; 0.; 4.; -4.; 0.; 0.; 0.; 0.];
   [4.; 0.; 4.; 0.; 0.; 0.; 0.; 0.; 4.];
   [0.; 4.; 0.; 0.; 0.; 0.; 0.; 0.; 0.];
   [4.; 0.; 4.; 0.; 0.; 0.; 4.; 0.; 0.]]

# energy(x,w);;
- : float = -40.
# energy(i,w);;
- : float = -40.
# energy(z,w);;
- : float = -40.
# energy(o,w);;
- : float = -40.

# nextState(x,w,0.0);;
- : float list = [1.; -1.; 1.; -1.; 1.; -1.; 1.; -1.; 1.]

# nextState(i,w,0.0);;
```

```
- : float list = [-1.; 1.; -1.; -1.; 1.; -1.; -1.; 1.; -1.]

# nextState(z,w,0.0);;
- : float list = [1.; 1.; 1.; -1.; 1.; -1.; 1.; 1.; 1.]

# nextState(o,w,0.0);;
- : float list = [1.; 1.; 1.; 1.; -1.; 1.; 1.; 1.; 1.]

# updateN(po,w,0.0,1);;
- : float list = [1.; 1.; 1.; 1.; -1.; 1.; 1.; 1.; 1.]

# updateN(px,w,0.0,1);;
- : float list = [1.; -1.; 1.; -1.; 1.; -1.; 1.; -1.; 1.]

# updateN(pi,w,0.0,1);;
- : float list = [-1.; 1.; -1.; -1.; 1.; -1.; -1.; 1.; -1.]

# updateN(pz,w,0.0,1);;
- : float list = [1.; -1.; 1.; -1.; 1.; -1.; 1.; 1.; 1.]

# updateN(pz,w,0.0,2);;
- : float list = [1.; 1.; 1.; -1.; 1.; -1.; 1.; -1.; 1.]

# updateN(pz,w,0.0,3);;
- : float list = [1.; -1.; 1.; -1.; 1.; -1.; 1.; 1.; 1.]

# updateN(pz,w,0.0,4);;
- : float list = [1.; 1.; 1.; -1.; 1.; -1.; 1.; -1.; 1.]

# updateN(pz,w,0.0,5);;
- : float list = [1.; -1.; 1.; -1.; 1.; -1.; 1.; 1.; 1.]

# findsEquilibrium(pz,w,0.0,1000);;
- : bool = false
```

7 `ocaml` Functions and Constructs Not Allowed

As noted, you will be designing, implementing and testing `ocaml` with a set of constraints intended to force purely functional programming without a reliance on libraries or modules which trivialize the effort.

Of extreme significance is the restriction of the paradigm to pure functional programming (no side effects). **No `ocaml` imperative constructs are allowed.** Recursion must dominate the function design process.

So that we may gain experience with functional programming:

1. *Only the applicative (functional) features of* `ocaml` *are to be used.* **Please reread the previous sentence.** This rules out the use of `ocaml`'s imperative features. See Section 1.5 'Imperative features' of the manual for examples of constructs not to be used.

2. **To force you into a purely applicative style, `let` can be used only for function definition**. `let` cannot be used in a function body.

3. Loops and local or global variables are prohibited.

4. **Outside of the Pervasives module**, the only module functions you may use are `List.hd`, `List.tl` and `List.nth`. This means you may not use the Array Module, or any other Library module.

5. Finally, **the use of sequence (6.7.2 in the ocaml manual) is not allowed**. **Do not design functions using sequential expressions or begin/end constructs.**

   If you are in doubt, ask in class and I'll provide a 'private-letter ruling'.

The objective is to obtain proficiency in functional programming, not to try to find built-in `ocaml` functions which simplify or trivialize the effort.

8 How We Will Build and Evaluate Your ocaml Solution

The sample ocaml script (excerpt only) below shows how varying input files and test vectors are used to test the functions in Section 4.2.

```
open List;;
#use"ourTestData.caml";;         (** various size data-- our choice *)
#use "sde1.caml";;          (** THIS IS YOUR ocaml SOURCE -- all functions *)
(** function invocation = testing of functions  -- samples only *)
(** note dimension of vectors (network size) may vary *)
# netAll(os2,w);;
# nextState(os2,w,0.0);;
# energy(os2,w);;
<etc>....
```

The grade is based upon a correctly working solution satisfying the constraints herein.

9 Format of the Electronic Submission

The final **zipped** archive is to be named **`<yourname>-sde1.zip`**, where `<yourname>` is your (CU) assigned user name. You will upload this to the Blackboard assignment prior to the deadline.

The minimal contents of this archive are as follows:

1. A `readme.txt` file *listing the contents of the archive and a brief description of each file.* Include 'the pledge' here. Here's the pledge:

   > **Pledge:**
   > On my honor I have neither given nor received aid on this exam.

   This means, among other things, that the code you submit is **your** code.

2. The `ocaml` source for your implementation (named `sde1.caml`). Note this file must include the functions defined in Section 4.2, as well as any additional functions you design and implement.

3. An ASCII log file showing 2 sample uses of each of the functions required in Section 4.2. Name this log file `sde1.log`.

Your `ocaml` implementation must meet the constraints posed herein and provide the correct functionality. Furthermore, the use of `ocaml` should not generate any errors or warnings. The grader will attempt to interpret your ocaml source and check for correct functionality. We will also look for offending `let`, sequence and function use. Recall the grade is based upon a correctly working solution.