ECE/CPSC 3520 Software Design Exercise #1: A Purely Functional Implementation of Selected Hopfield Recurrent Artificial Neural Network Functions Using `ocaml` *(Given Existing and Possibly Useful* `ocaml` *Code)*

Canvas submission only
Assigned 1/23/2020
(Actually released 1/21/2020)
Due 2/20/2020 11:59PM

Contents

1 Preface

SDE 1 exposes you to the application of functional programming (ocaml) to Artificial Neural Network (ANN) implementation. ANNs are currently relevant and very popular, e.g., a large fraction of the research and development on self-driving automobiles is based upon an elaborate architecture and training procedure based upon Connectionist Neural Networks (CNNs).

The overall objective of SDE1 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.

One novelty of the effort this semester is that I am giving you some elemental ANN code (in `ocaml`) to get you started and force 'code reading' (see text Section 1.6.2). You will then design, implement and test a set of extended functions.

Note that you are not free to choose any available construct or feature in `ocaml`; Section 8 outlines the restrictions.

2 Resources

It would be foolish to attempt this SDE without carefully exploring:

1. The text, especially the many ocaml examples in Chapter 11;
2. The `ocaml` and SDE1 lectures;
3. The background provided in this document; and
4. The `ocaml` manual.

We use some very elementary linear algebra constructs for representation (basically vectors, matrices, transpose and multiplication). Make sure you understand this formulation.

3 ANN Recurrent Network Concept, 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 (a weighted linear input combination (WLIC):

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

   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 net activation value, net_i, into the artificial unit output, o_i. Here you will implement the Hopfield activation or 'squashing' function.

3.2 The (Hopfield) Recurrent Network Architecture

The 'Hopfield' net, honors John Hopfield of Caltech, who seems to have popularized the strategy. In this SDE, you are implementing and training a somewhat basic 'Hopfield' recurrent net. 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. This is represented as the state vector, $\underline{o}$. **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.

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 weights w_{ij} are real numbers. With the constraints developed below, for a d-unit network there are $\frac{d(d-1)}{2}$ possibly nonzero and unique network weights.

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. Hopfield's original work postulated that **in the case of $net_i = 0$, the unit output is unchanged from its previous value**. We refer to this as the 'leave it alone' characteristic, given by:

$$o_i = \begin{cases} 1 & if \sum_{j;\ j\neq i} w_{ij}o_j > 0 \\ o_i \quad (previous\ value) & if \sum_{j;\ j\neq i} w_{ij}o_j = 0 \\ -1 & otherwise \end{cases} \tag{3}$$

Notice from Equation 3, the neuron activation characteristic is nonlinear.

3.2.3 Hopfield 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 only consider the case of $\{-1, 1\}$ unit outputs. A set of desired stored states $\underline{o}^s, s = 1, 2, ...n$, form the **training set** of stored states, $H = \{\underline{o}^1, \underline{o}^2, \ldots, \underline{o}^n\}$. The storage prescription you will implement is given by Hopfield:

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

with the additional 'no-self-activation' constraint

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

3.3 Network State Propagation or Evolution

3.3.1 Forming the Next State

Given a network with (learned) unit weights and a Hopfield activation function, the next state is computed[1]. This is the overall state propagation computation of the network simulation.

3.3.2 Network 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 of Energy

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 therefore stays the same or (more commonly) decreases.

3.3.3 Network Equilibrium

Any time the evolution of a recurrent Hopfield network leads to a next state which is identical to the previous state, the network is said to be in equilibrium. A network in equilibrium will have a constant energy function value, but the converse is not true[2].

[1]This is the so-called synchronous update case, where all current unit outputs are 'frozen' and used to compute the next unit activation values using Equation 1 which are subsequently squashed to form the new state vector.

[2]In other words, if the network energy is constant, this does not necessarily mean the network is in equilibrium.

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 Section 6.

Notes:

1. The entire effort must use only `ocaml`, Version 4.02 or newer.
2. Pay particular attention to the signatures and names of functions shown and the data structures used. If you implement one or more functions with signatures different from those specified, they will fail the testing script.
3. **All multiple-input functions to be developed have a tupled interface.**
4. Recall **ocaml is case sensitive.**
5. 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.
6. Pay particular attention to the restrictions in Section 8.

4.2 General Representation Issues

- We will use floating point numbers for unit weight values, net activation values and unit output values.
- 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. W will always be a square, symmetric real matrix with a zero diagonal. **Note the number of**

units in the Hopfield network is arbitrary. Your software must accommodate this.

4.3 Example Data and Use of the Network State Representation

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

let os1 = [1.0; -1.0; 1.0; -1.0];;
val os1 : float list = [1.; -1.; 1.; -1.]

let os2 = [-1.0; -1.0; 1.0; -1.0];;
val os2 : float list = [-1.; -1.; 1.; -1.]

let os3 = [-1.0; -1.0; 1.0; 1.0];;
val os3 : float list = [-1.; -1.; 1.; 1.]

(** show Eqns (4) and (5) in action *)

# 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.]]
```

The last example directly above shows the network weight matrix, W, represented as a list of list structure in `ocaml`.

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

Prototypes are shown below. **Note all functions use a tupled argument interface and none have side effects.** You should study the examples of use and verify the results by hand.

5.1 Basic Unit-Related Functions (Given)

These are contained in the posted CANVAS file:
`sde1_sp2020_given_functions.caml`

You use these functions, you do not modify or develop them. They provide good tutorial information. Do not include them with your submission; we will load this file as part of the evaluation process.

5.1.1 `netUnit`

```
(** Returns net activation (scalar) for a single unit using our
list-based input and weight representation and Eqn (1) *)

Prototype: netUnit(inputs, weights)
Signature: val netUnit : float list * float list -> float = <fun>

Examples:

# netUnit([-1.; -1.; 1.; -1.],[1.; 0.; -3.; 1.]);;
- : float = -5.

# netUnit([-1.; -1.],[1.; 0.]);;
- : float = -1.

# netUnit(os1,[1.;2.;3.;1.]);;
- : float = 1.
```

5.1.2 `netAll`

```
(* Returns net activation computation for entire network
   as a vector (list) of individual unit activations *)

Prototype: netAll(state, weightMatrix)
Signature: val netAll : float list * float list list -> float list = <fun>

Examples:

(data from section 4.3)

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.]
```

```
# netAll(os2,w);;
- : float list = [3.; -1.; 1.; -1.]
```

5.1.3 hop11Activation

```
(** Returns 'squashed' unit output.
    Implements Hopfield activation function
    corresponding to Eqn (3) for single (-1,1) unit *)

Prototype: hop11Activation(net,oldo)
Signature: val hop11Activation : float * float -> float = <fun>

Examples:

# hop11Activation(-3., 1.);;
- : float = -1.
# hop11Activation(3., 1.);;
- : float = 1.
# hop11Activation(0., 1.);;
- : float = 1.
```

5.1.4 hop11ActAll

```
(** Returns a vector (list) of all 'squashed' unit outputs,
    given a tuple consisting of a vector of net activations
    and a vector of previous (current) unit outputs
 *)

Prototype: hop11ActAll(net,oldo)
Signature: val hop11ActAll : float list * float list -> float list = <fun>

Example:

# let os1 = [1.0; -1.0; 1.0; -1.0];;
val os1 : float list = [1.; -1.; 1.; -1.]

# hop11ActAll([-10.;34.5;0.0;-1.0],os1);;
- : float list = [-1.; 1.; 1.; -1.]
```

6 Functions You Will Design, Implement and Test

This is your work. Prototypes and examples of the use of each function are shown below. **Note all functions use a tupled argument interface and none have side effects.** Like the given functions, you should study the examples of use and verify the results by hand.

6.1 State Propagation-Related `ocaml` Functions

6.1.1 `nextState`

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

Prototype: nextState(currentState, weightMatrix)
Signature: val nextState : float list * float list list -> float list = <fun>

Examples:

data:

val w : float list list =
  [[0.; -1.; 1.; -1.]; [-1.; 0.; -1.; 1.]; [1.; -1.; 0.; -1.];
   [-1.; 1.; -1.; 0.]]

val w2 : float list list =
  [[0.; 1.; -1.; -1.]; [1.; 0.; -3.; 1.]; [-1.; -3.; 0.; -1.];
   [-1.; 1.; -1.; 0.]]

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

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

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

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

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

6.1.2 `updateN`

```
(** Returns network state after n time steps;
    i.e.,  update of N time steps *)

Prototype: updateN(currentState, weightMatrix, n)
Signature: val updateN : float list * float list list * int -> float list = <fun>

Examples:

(* 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,1);;
- : float list = [1.; 1.]

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

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

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

Interestingly, the above example shows the Hopfield network in a cycle.

6.1.3 `findsEquilibrium`

```
(** Function findsEquilibrium returns true if network reaches
    an equilibrium state within range steps, false otherwise *)

Prototype: findsEquilibrium(initialState, weightMatrix, range)
Signature: val findsEquilibrium : float list * float list list
                                  * int -> bool = <fun>

Examples:
```

```
val w : float list list =
  [[0.; -1.; 1.; -1.]; [-1.; 0.; -1.; 1.]; [1.; -1.; 0.; -1.];
   [-1.; 1.; -1.; 0.]]
# findsEquilibrium(os1,w,3);;
- : bool = true

# 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.]
# findsEquilibrium(oi,we,5);;
- : bool = false
```

6.1.4 `energy`

This function implements Equation 6.

```
Prototype: energy(state,weightMatrix)
Signature: val energy : float list * float list list -> float = <fun>

Example:

# w;;
- : float list list =
[[0.; -1.; 1.; -1.]; [-1.; 0.; -1.; 1.]; [1.; -1.; 0.; -1.];
 [-1.; 1.; -1.; 0.]]
# os1;;
- : float list = [1.; -1.; 1.; -1.]
# energy(os1,w);;
- : float = -6.
```

6.2 Network Storage Prescription Functions

6.2.1 `hopTrainAstate`

```
(** Returns weight matrix for only one stored state,
    used as a 'warmup' for the next function *)

Prototype: hopTrainAstate(astate)
Signature: val hopTrainAstate : float list -> float list list = <fun>

Examples:

# hopTrainAstate(os1);;
```

```
- : float list list =
[[0.; -1.; 1.; -1.]; [-1.; 0.; -1.; 1.]; [1.; -1.; 0.; -1.];
 [-1.; 1.; -1.; 0.]]
# hopTrainAstate(os2);;
- : float list list =
[[0.; 1.; -1.; 1.]; [1.; 0.; -1.; 1.]; [-1.; -1.; 0.; -1.];
 [1.; 1.; -1.; 0.]]
# hopTrainAstate(os3);;
- : float list list =
[[0.; 1.; -1.; -1.]; [1.; 0.; -1.; -1.]; [-1.; -1.; 0.; 1.];
 [-1.; -1.; 1.; 0.]]
```

6.2.2 `hopTrain`

```
(** This returns weight matrix, given a list of stored states
    (shown previously) using Eqns (4) and (5) *)

Prototype: hopTrain(allStates)
Signature: val hopTrain : float list list -> float list list = <fun>

Examples:

# 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.]]

# 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.]]
```

7 Function Signatures

Here's a summary of the 10 signatures of the ANN `ocaml` functions you will work with:

```
val netUnit : float list * float list -> float = <fun>
val netAll : float list * float list list -> float list = <fun>
val hop11Activation : float * float -> float = <fun>
val hop11ActAll : float list * float list -> float list = <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 list = <fun>
val energy : float list * float list list -> float = <fun>
val updateN : float list * float list list * int -> float list = <fun>
val findsEquilibrium : float list * float list list * int -> bool = <fun>
```

8 `ocaml` Functions and Constructs Not Allowed

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. To this end, we impose the following constraints on the solution.

8.1 No `let` for Local or Global Variables

So that you may gain experience with functional programming, 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. To force you into a purely applicative style, **`let` can only be used for function naming**. `let` or the keyword `in` cannot be used in a function body. Reread the following sentence. Loops and 'local' or 'global' variables or nested function definitions is strictly prohibited.

8.2 Only Functions in the Pervasives Module and the Following Functions in Other Modules are Permitted in Your Solution

The allowable functions in SDE1 are those non-imperative functions in the Pervasives module and these 3 individual functions listed below:

```
List.hd
List.tl
List.nth
```

No modules (other than Pervasives) may be opened. In other words, you cannot open any modules. There cannot be an `open List;;` or `open String;;`, or any similar statement in your source file. Each of the above 3 functions

must be used with the proper Module name. This also eliminates namespace ambiguity.
This constraint actually helps most of the class. Note you may not need all of these functions. Since no other functions are allowed, you can stop searching the ocaml libraries for something to trivialize the function development challenges in this assignment.

8.3 No Sequences

The use of sequence (6.7.2 in the ocaml manual) is not allowed. Do not design your functions using sequential expressions or begin/end constructs. Here is an example of a sequence in a function body:

```
let print_assignment = function(student,course,section) ->
print_string student; (* first you evaluate this*)
print_string " is assigned to "; (* then this *)
print_string course;  (* then this *)
print_string " section " ; (* then this *)
print_int section;  (* then this *)
print_string "\n";; (* then this and return unit*)
```

8.4 No (Nested) Functions

`ocaml` allows 'functions defined within functions' definitions (another 'illegal' `let` use for SDE1). Here's an example of a nested function definition:

```
# let f a b =
    let x = a +. b in
    x +. x ** 2.;;
```

9 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 developed in Section 6.

```
(** to test new functions *)
#use"sde1_sp2020_given_functions.caml";; (* WE supply this file *)
#use"sde1_sp2020.caml";; (* you supply this file with only your new functions *)
#use"testData.caml";;       (** various size data and cases -- our choice *)
#use"testing-sde1_sp2020.caml"     (** our test cases executed by script *)
```

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

10 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_sp2020.caml`). Note this file must include the functions defined in Section 6, as well as any additional functions you design and implement. **Do not include the functions in file `sde1_sp2020_given_functions.caml`.**

3. An ASCII log file showing 2 sample uses of each of the functions required in Section 4.2. The format of entries in this file is the same as the examples shown previously; `ocaml` interpreter response (output) must be shown. 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 which meets the specifications and constraints in this document.

11 Final Remarks

11.1 A Really Big Word of Advice

If your `ocaml` source file cannot be compiled by the `ocaml` interpreter, i.e., the input

```
   #use"sde1\_sp2020.caml";;
```

raises exceptions, we can't grade your solution. **Before you submit your archive, CHECK YOUR WORK** in a clean directory.

11.2 Deadlines Matter

energy
This remark was included in the course syllabus and SDE1. Since multiple submissions to Canvas are allowed[3], if you have not completed all functions, you should submit a freestanding archive of your current success before the deadline. This will allow the possibility of partial credit. **Do not attach any late submissions to email and send them to either me or the graders.**

[3]But we will only download and grade the latest (or most recent) one, and it must be submitted by the deadline.