CSCI3180 Assignment 1: OCaml

2024-2025, Term 2

Introduction

In this assignment, you'll develop a simple feed-forward neural network framework in OCaml by filling in missing code in the file `nn.ml`. There are a total of **100 points** for this assignment.

Guidelines

You should follow the below guidelines during completion of the assignment:

1. Modify *only* the code in the file `nn.ml`.
2. You may define additional functions if you find it helpful, but ensure that they are given *unique* names so that they don't shadow other values defined in `nn.ml`.
3. Do not modify any types, values, or functions already defined in `nn.ml` unless you have been asked to do so as part of an exercise.
4. Your code **should not use any records with mutable fields or references** (i.e., OCaml `ref`s). Violation of this guideline will result in a reduction of 30 points from the assignment.
5. Do not import any modules (i.e., by using `open`) into `nn.ml`.

Failure to adhere to the guidelines may result in a reduction of points from your assignment.

Submission Guidelines

The submission deadline for the assignment is **11:59pm on 1 March 2025**. There will be a late submission penalty of **1 point per 5-minutes late**, where the amount of time late is rounded *up* to the nearest 5 minutes. You should complete the following by the deadline:

- Ensure that you have completed the declaration at the top of `nn.ml`.
- On Blackboard, for Assignment 1, attach and submit *only* your completed version of `nn.ml`. (Do not zip or otherwise compress the file.) It *must* be called `nn.ml`.

You can submit as many times as you want, but **only the latest submission will be graded**. You are encouraged to submit early to prevent any issues!

Testing

If you've implemented everything correctly, you can run the tests in `tests.ml`. First, build a native executable running `ocamlopt -o tests nn.ml circle.ml tests.ml`. Then if you run the resulting file (called `tests` or `tests.exe`), you should get the following output:

```
Prop 1  (repeat        ): Passed!
Prop 2  (repeat        ): Passed!
Prop 3  (list_apply    ): Passed!
Prop 4  (weighted_sum  ): Passed!
Prop 5  (weighted_sum  ): Passed!
Prop 6  (weighted_sum  ): Passed!
Prop 7  (mk_node       ): Passed!
Prop 8  (mk_node       ): Passed!
Prop 9  (mk_layer      ): Passed!
Prop 10 (example       ): Passed!
```

```
Prop 11 (node_from_arch): Passed!
Prop 12 (nn_from_arch  ): Passed!
Prop 13 (nn_from_arch  ): Passed!
Prop 14 (example       ): Passed!
Prop 15 (example       ): Passed!
Prop 16 (nn_from_arch  ): Passed!
Prop 17 (circle_nn     ): Passed!
Prop 18 (empty_hidden  ): Passed!
Prop 19 (empty_hidden  ): Passed!
Prop 20 (empty_output  ): Passed!
```

Note that just because all these tests pass, it doesn't mean that you will necessarily get full marks! You are strongly encouraged to run your own tests.

Part 1: Warm-up (10 pts)

Before we get started, let's implement some utility functions that will come in handy later one.

Exercise 1 (4 pts)

Implement a function `repeat : int -> 'a -> 'a list` such that `repeat n v` creates a list of `v`s with length `n`.

```
let rec repeat n v = (* your code here *)
```

Exercise 2 (6 pts)

Implement a function `list_apply : ('a -> 'b) list > 'a list -> 'b list` such that it applies the i^{th} function of its first argument to the i^{th} value of its second:
`list_apply [ f_0 ;...; f _n ] [ x_0; ...; x_n ] = [ f_0 x_0; ...; f_n x_n ].`

```
let list_apply fs xs = (* your code here *)
```

Part 2: Representing Neural Networks (48 pts)

Neural Networks

A neural network is a function whose behavior is described via a a weighted directed acyclic graph (DAG) $G = (V, E, W)$, made up of a set of nodes V, a set of edges $E \subseteq V \times V$, and a function $W : E \to \mathbb{R}$ that maps from edges to real-valued weights.

Since OCaml doesn't have a built-in representation of real numbers, we'll use the floating-point type `float` to model real numbers. We will use the built-in `list` type to model tuples, sequences, and sets of varying sizes. The type `nn` that we use to represent neural networks is defined below:

```
type nn = float list -> float list
```

Nodes (20 pts)

The nodes V of the neural network are called *neurons*, and neurons are themselves functions that take a set of pairs of a weight and a value. The weights and edges in the neural network describe which weights and values each neuron takes as an input. The type `node` that we use to represent neurons is defined below:

```
type node = (float * float) list -> float
```

Each neuron is parameterized by an *activation function* in $\mathbb{R} \to \mathbb{R}$ and a *bias* in $\mathbb{R}$. The function carried out by a neuron with activation function A and bias b that takes an input $\{(w_0, x_0), \ldots, (w_n, x_n)\}$

is described below:

$$A\Big(b + \sum_{i=0}^{n} w_i \cdot x_i\Big)$$

We can see that in the case where the input is of size 0, the neuron simply computes $A(b)$. In order to compute the function carried out by a neuron, we need to compute the sum of the bias and a weighted sum.

Exercise 3 (10 pts)

Implement a function `weighted_sum : float -> (float * float) list -> float` that takes a bias `bias` and a list of weight-and-value pairs `weights_and_vals`. The function `weighted_sum` should return the value that results from adding `bias` to the weighted sum of of the entries in `weights_and_vals`. Your implementation should use `List.fold_left`.

```
let weighted_sum bias weights_and_vals =
  (* your code here *)
```

Hints.

- There are two ways to write the addition operator for `float`s:

  `(+.) : float -> float -> float` and

  `Float.add : float -> float -> float`

  Operator `(+.)` is an infix operator, allowing you to write, e.g., `1. +. 2.`, which behaves the same as `Float.add 1. 2.`.

- There are two ways to write the multiplication operator for `float`s:

  `(*.) : float -> float -> float` and

  `Float.mul : float -> float -> float`

  Operator `(*.)` is an infix operator; e.g., `1. *. 2.` is the same as `Float.mul 1. 2.`.

Now that we have a way of computing a weighted sum, let's implement a function that creates `node`s from activation functions and biases.

Exercise 4 (10 pts)

In this exercise, you will implement the function `mk_node : (float -> float) -> float -> node` that creates a `node` from an activation function and a bias.

```
(* make a node *)
let mk_node activation_fn bias = function
  | weights_and_vals -> (* your code here *)
```

Part 1 (2 pts). Add explicit type annotations for the parameters and return type of `mk_node` reflecting the following type:

`(float -> float) -> float -> node`

Part 2 (8 pts). Implement the function `mk_node`.

Layers (28 pts)

Feed-forward neural networks are arranged into a sequence of *layers* of sequences of nodes $L_0, \ldots, L_n$. An edge (v_1, v_2) exists in E *only if* v_1 is in some layer L_i that immediately precedes the layer L_{i+1} that v_2 is in. For a neural network with $n+1$ layers that implements a function in $\mathbb{R}^{in} \to \mathbb{R}^{out}$, there will be *in* nodes in the input layer L_0 and *out* nodes in the output layer L_n. Layers that are neither input nor output layers are called *hidden layers*.

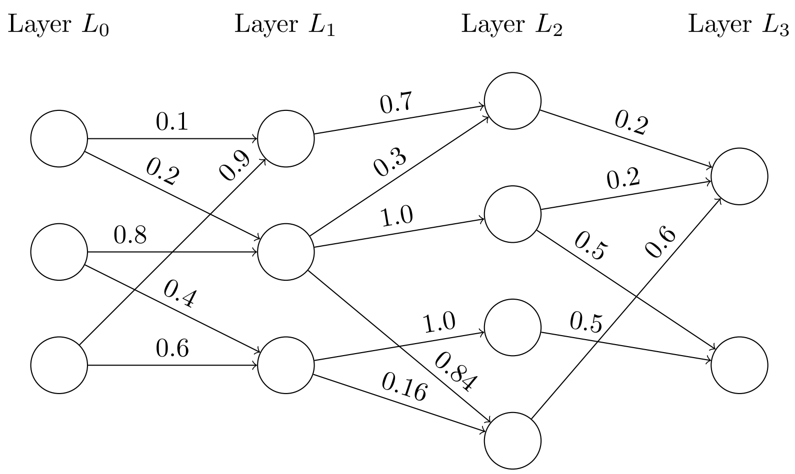

Figure 1: A feed-forward neural network with 2 hidden layers

We will represent the set of edges in the neural network in a per-layer manner. We define a type `connectivity`, which is simple a type alias for `int list`:

```
type connectivity = int list
```

For two adjacent layers $L_{i-1} = (v_0, \ldots, v_m)$ and $L_i = (v'_0, \ldots, v'_n)$, we model the edges between nodes of these layers as a `connectivity list` associated with layer L_i. The j^{th} element of the list is a `connectivity` containing the indices of the nodes in layer L_{i-1} that are connected to node v'_j: for each edge $(v_k, v'_j) \in E$, the `int` representing index k appears in the `connectivity` exactly once. As an example, below is a list representing the edges between the nodes in layers L_0 and L_1 in Fig. 1:

```
[ [ 0; 2 ]; [ 0; 1 ]; [ 1; 2 ] ]
```

We similarly represent the weights of these edges as a `float list list`. In particular, if we have that the edges between adjacent layers are represented by `connectivities : connectivity list` that is such that `List.nth (List.nth connectivities j) idx == k`, then for the list of weights `weights : float list list`, the expression `List.nth (List.nth weights j) idx` should be the `float` representing weight $W(v_k, v'_j)$. As an example, below is a list representing the weights of the edges between the nodes in layers L_0 and L_1 in Fig. 1:

```
[ [ 0.1; 0.9 ]; [ 0.2; 0.8 ]; [ 0.4; 0.6 ] ]
```

Exercise 5 (6 pt)

For this exercise, you will be considering the layers L_1 and L_2 of the neural network shown in Fig. 1.

Part 1 (3 pts) Define `example_connectivities : connectivity list` that describes the edges between the nodes depicted above.

```
let example_connectivity = (* your code here *)
```

Part 2 (3 pts) Define `example_weights : float list list` that describes the edges between the nodes depicted above.

```
let example_weights = (* your code here *)
```

The overall computation performed by the neural network proceeds layer-by-layer:

Input layer. For an input $\langle x_0, \ldots, x_{in} \rangle$ to the neural network, the neurons in the input layer $L_0 = (v_0, \ldots, v_{in})$ are such that each node v_k takes input $\{(1, x_k)\}$, has the identity function as its activation function, and has a bias 0. That is, each node v_k simply outputs the input x_k. This input layer can be thought of as implementing the identity function on $\mathbb{R}^{in}$:

$$\mathsf{id}(\langle x_0, \ldots, x_{in} \rangle) = \langle x_0, \ldots, x_{in} \rangle$$

Other layers. For any edge $(v_1, v_2) \in E$, the neuron v_2 takes the pair of the weight $W(v_1, v_2)$ and the already-computed output of v_1 as part of its inputs. The outputs of the nodes in the output layer provide the output of the entire neural network. Each non-input layer L_i can be thought of implementing a function $\ell_i : \mathbb{R}^{|L_{i-1}|} \to \mathbb{R}^{|L_i|}$.

We can therefore define the behavior of a neural network with layers $L_0, \ldots, L_n$ as $\ell_n \circ \cdots \circ \ell_1 \circ \mathsf{id}$. We will omit the input layer `id` to model neural network behavior as $\ell_n \circ \cdots \circ \ell_1$.

Exercise 6 (2 pts)

We will be modeling the behavior of the neural network as a composition of non-input layer functions ℓ_i. Give the type that `layer` should be an alias for another type, given that a `layer` should model a function ℓ_i implemented by a neural network layer:

```
type layer = (* your code here *)
```

Now we can implement the layer functions ℓ_i.

Exercise 7 (20 pts)

In this exercise, you will implement the function `mk_layer : node list -> connectivity list -> float list list -> layer` that creates a `layer` from a list of nodes in the corresponding layer, a `connectivity list` called `c` that represents the edges from the previous layers' nodes, and a list `weights` representing the weights for the edges from the previous layers' nodes.

```
let mk_layer (nodes : node list) (c : connectivity list)
    (weights : float list list) : layer = function
  | inputs ->
      (* use if helpful *)
      let nth_input = List.nth inputs in
      (* optional -- define and use if helpful:
        let per_node_inputs : float list list =
          (* your code here *)
        in
      *)
      (* your code here *)
```

Hints.

1. You may find it helpful first to define `per_node_inputs : float list list` that gets the input *values* for each node from the input *indices* in `connectivity` and the layer input `inputs`.

2. You may wish to use the `list_apply` function you defined earlier.

Having defined `mk_layer`, we can now model neural networks. Below is the OCaml code modeling an example neural network with three inputs and one hidden layer (`layer1`):

```
(* some activation functions *)
let relu x = if x > 0. then x else 0.
let sigmoid x = Float.div 1. (1. +. Float.exp x)

(* example nn *)
let example : nn = function
  | input ->
      let layer1 =
        mk_layer
          (repeat 3 (mk_node relu 0.))
```

```
          [ [ 1 ]; [ 0; 1; 2 ]; [ 0; 2 ] ]
          [ [ 1. ]; [ 0.3; 0.3; 0.4 ]; [ 0.2; 0.8 ] ]
      in
      let layer2 =
        mk_layer
          (repeat 2 (mk_node relu 0.2))
          [ [ 0; 1; 2 ]; [ 0; 1; 2 ] ]
          [ [ 0.3; 0.4; 0.3 ]; [ 0.4; 0.3; 0.3 ] ]
      in
      layer1 input |> layer2
```

Part 3: Specification (42 pts)

Typically, neural network frameworks allow for separate specification of the neural network *parameters*, which comprise the nodes' biases and edges' weights, and the network *architecture*, which comprises everything else in the network. Parameters are typically learned, whereas the architecture is specified up-front. Later, these can be combined to construct and run a full neural network.

While you can now model neural networks using the OCaml functions you've implemented, it isn't convenient to use our functions as-is for specifying neural network architectures and parameters separately.

To make a more user-friendly framework, let's make some new ADTs for neural network architecture specification, where we allow a node to take either of the two kinds of activation functions we defined in the previous section:

```
(* nn architecture *)
type activation = ReLU | Sigmoid
type node_arch = Node of activation
type in_edge_indices = int list

(* nn architectures as compositions of one or more neural networks *)
type nn_arch =
  | Uniform of (int * node_arch * in_edge_indices)
  | Custom of (node_arch * in_edge_indices) list
  | Composition of nn_arch list
```

Our `nn_arch` ADT describes neural network architectures as compositions of one or more other architectures. In particular,

- `Uniform(n, a, c)` describes a neural network with one (non-input) layer of `n` nodes with architecture `a`, where each node has connectivity `c`.
- `Custom(acs)` describes a neural network with one (non-input) layer of `List.length(acs) nodes`, where each `(a, c)` in `acs` specifies a node with architecture `a` and connectivity `c`.
- `Composition(nns)` describes a neural network formed by the composition of the neural networks `nns`, where for `nns = [ nn_1; nn_2; ...; nn_k ]`, `nn_1` is applied first to the inputs of the neural network, `nn_2` to the output of `nn_1`, `nn_3` to the output of `nn_2`, and so on.

Every architecture has an *implicit input layer* that acts like the identity function.

Note that we can interpret these ADTs as ASTs for a language, so that OCaml terms of type `nn_spec` are ASTs in another language *embedded* in OCaml. This language, of course, is not a general purpose programming language, but is rather a *domain-specific language* (DSL) for specifying neural network architectures. DSLs that are implemented via embedding in another language are called *embedded DSLs* (eDSLs). Popular neural network frameworks like TensorFlow can be regarded as Python eDSLs[1].

An example architecture for the neural network `example` defined in Part 2 using our architecture specification language is shown below:

```
(* example nn architecture *)
let example_arch =
  let layer1 =
    Custom
```

[1] As is done in this blog post: `https://julialang.org/blog/2017/12/ml-pl/`

```
      [ (Node ReLU, [ 1 ]); (Node ReLU, [ 0; 1; 2 ]); (Node ReLU, [ 0; 2 ]) ]
  in
  let layer2 = Uniform (2, Node ReLU, [ 0; 1; 2 ]) in
  Composition [ layer1; layer2 ]
```

We will also provide some types for describing what parameter specifications should look like:

```
(* nn params *)
type layer_params = { weights : float list list; biases : float list }
type nn_params = layer_params list
```

Now that we have ways of specifying architectures and parameters, we need to define a way of combining them to get a neural network.

Exercise 8 (5 pts)

Implement the function `node_from_arch : node_arch -> float -> node` using `mk_node`.

```
let node_from_arch (Node activation) bias =
  (* your code here *)
```

Exercise 9 (25 pts)

Implement the function `nn_from_arch : nn_arch -> nn_params -> nn`. It should return an exception `InvalidParams` if the given `nn_params` do not match the given architecture (i.e., there are too few/many biases or weights). Note that because of the implicit input layer assumption we made earlier, when `nn_from_arch` is applied to an empty architecture, the output `nn` should behave like an identity function on its inputs.

```
exception InvalidParams of nn_arch * nn_params

let nn_from_arch arch params inputs =
  (* your code here *)
```

This function should be such that the code below successfully defines a `nn` that produces the same results as the `example` that we defined at the end of Part 2:

```
(* apply nn architecture to weights *)
let example_v2 : nn =
  let layer1_params =
    {
      biases = [ 0.; 0.; 0. ];
      weights = [ [ 1. ]; [ 0.3; 0.3; 0.4 ]; [ 0.2; 0.8 ] ];
    }
  in
  let layer2_params =
    {
      biases = [ 0.2; 0.2 ];
      weights = [ [ 0.3; 0.4; 0.3 ]; [ 0.4; 0.3; 0.3 ] ];
    }
  in
  nn_from_arch example_arch [ layer1_params; layer2_params ]
```

Now we can use everything so far to implement a neural network that does something a bit more interesting.

Exercise 10 (12 pts)

Define a neural network architecture `circle_arch` for a neural network with two inputs and three hidden layers, where each layer is described below:

> **Layer 1** has five neurons, all of which use a ReLU activation function and have incoming edges from both of the input layer's neurons.
>
> **Layer 2** has five neurons, all of which use a ReLU activation function and have incoming edges from all five of the previous layer's outputs.
>
> **Layer 3** has five neurons, all of which use a ReLU activation function and have incoming edges from all five of the previous layer's outputs.
>
> **Layer 4** (the output layer) has one neuron that uses a sigmoid activation function and has incoming edges from all five of the previous layer's outputs.

```
let circle_arch =
  (* your code here *)
```

The file `circle.ml` creates the `nn` that results from combining the architecture `circle_arch` and parameters learned for the architecture described in Exercise 10. The file `draw_circle.ml` uses runs this resulting `nn` on several points to produce an ASCII drawing. After implementing everything, you should be able to build an executable `draw_circle` or `draw_circle.exe` by running the following:

```
ocamlopt -o draw_circle nn.ml circle.ml draw_circle.ml
```

Running the resulting executable should yield the following output:

```
o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o
o o o o o o o . . o o o o o o o
o o o o o . . . . . . o o o o o
o o o o o . . . . . . o o o o o
o o o o . . . . . . . . o o o o
o o o o . . . . . . . . o o o o
o o o o o . . . . . . o o o o o
o o o o o . . . . . . o o o o o
o o o o o o o . . o o o o o o o
o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o
o o o o o o o o o o o o o o o o
```