C Final Test

Imperial College London

Department of Computing

Artificial Neural Networks

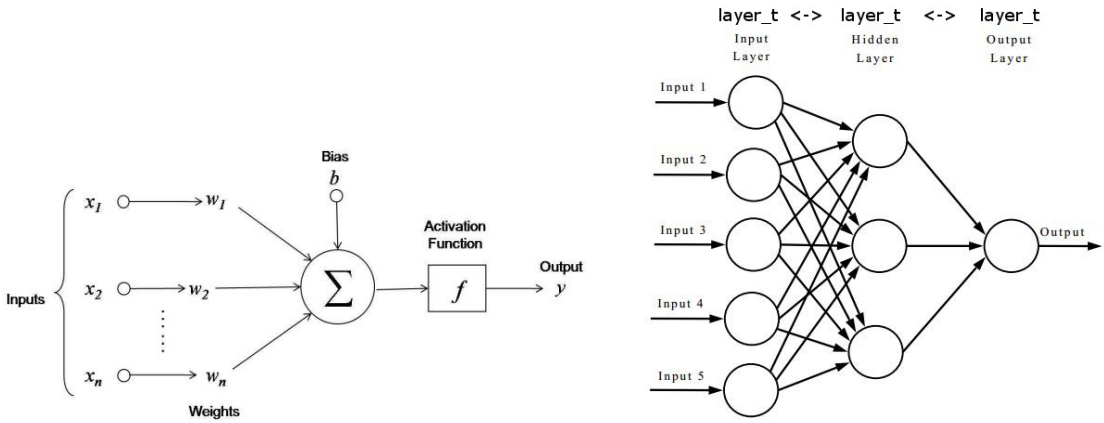
10:00 – 13:00
THREE HOURS
(including 10 minutes planning time)

- Please make your swipe card visible on your desk.
- After the planning time log in using your username as **both** your username and password.
- The maximum total is **30 marks**.
- Credit will be awarded throughout for clarity, conciseness, useful commenting, appropriate use of assertions and error checking.
- **Important:** THREE MARKS will be deducted from solutions that do not compile. Comment out any code that does not compile before you leave. Commented-out code will not be marked.
- The files can be found under the subfolder "**ann**" inside the Home folder of Lexis.

Monday 11 June 2018

Feed-forward Neural Networks

Artificial neural networks (ANNs) are computing systems inspired by the biological neural networks that constitute animal brains. Recently, they have been at the core of artificial intelligence (AI) tasks varying from speech recognition to playing the board game Go. A building block for all these complex tasks are a class of feed-forward neural networks called multilayer perceptrons (MLP). A perceptron is like a **neuron** that acts as a function mapping inputs to a single output.



(a) Artificial neuron.

(b) Feed-forward neural network.

Figure 1: Construction of feed-forward artificial neural networks.

Figure 1a represents a single neuron computation. This computation is identical for every neuron/perceptron in the network. All the **inputs** are multiplied by the **weights** and summed together with the **bias** then passed through the **activation function** to give the **output** result. It turns out this simple computation when chained together in a network architecture can learn complex non-linear functions.

Figure 1b shows a 3-layer neural network with multiple inputs and a single output. Each column of neurons is regarded as a single **layer** and layers are connected so that the output of one becomes the input of the next except for the input and output layers. These network architectures are called feed-forward as the values only propagate forward from the input layer to the output layer. These networks are **fully connected** such that between layers every output is connected to the input of every neuron in the next layer.

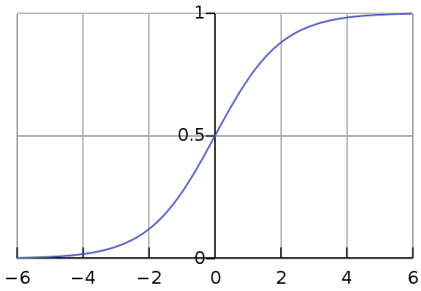


Figure 2: The logistic (sigmoid) function.

Figure 2 plots the logistic function or the **sigmoid** function which we will use as the **activation function** for the neurons. As inputs get larger the neuron *fires* and the output saturates towards 1.

Your Task

You will implement an artificial neural network library with the objective of making a 3 layer feed-forward neural network to learn the `XOR` function. You have been provided with a skeleton code for implementing a multilayer feed-forward network and a main function that will create the neural network using your library to train it. The structure of the source code is as follows:

layer.h Defines the `layer_t` type and function signatures for layer operations. **Do not modify.**

layer.c Write your answers to Part I in this file.

ann.h Defines the `ann_t` type and function signatures for ANN operations. **Do not modify.**

ann.c Write your answers to Part II in this file.

train.c Creates and runs a neural network with some debugging information to train XOR function. **Do not modify.**

rdata.c Generates random training data and contains memory leaks to be fixed in Part III.

The **train.c** file contains a main function that utilises the library to create a network and train it using all the functions. You can use the given `make` rules to either run or memory leak check as you implement the questions. We encourage you to run `make runtrain` after incremental changes to see some debugging output about the layer weights etc.

```
make runtrain # compiles and runs ./train
make runrdata # compiles and runs ./rdata
make checktrain # compiles and memory leak checks ./train
make checkrdata # compiles and memory leak checks ./rdata
```

We would recommend you to attempt the questions in each part in a linear fashion as they are intended to be of increasing difficulty. At any point make your assumptions clear using the standard `assert(expression)` function; for example, when you expect an argument to be non-NULL.

Part I Layers

In this part you will complete the functions that comprise a *single layer* that is a vertical column in Figure 1b. To make things more compact, this library does not have an explicit neuron implementation but stores the properties of one in dynamically allocated arrays. For example, `outputs[i]` contains the output of the ith neuron in the layer. Listing 1 shows the structure of a layer. All the functions are located inside **layer.c**.

Listing 1: Definition of a single layer inside layer.h

```
1  /* Represents a single neural layer. */
2  typedef struct layer {
3    /* Number of inputs and outputs (neurons).*/
4    int num_inputs, num_outputs;
5    /* Output of EACH neuron. */
6    double *outputs;
7    /* Pointers to previous and next layer if any. */
8    struct layer *prev;
9    struct layer *next;
10   /* Incoming weights of EACH neuron. */
11   double **weights;
12   /* Biases of EACH neuron. */
13   double *biases;
14   /* Delta errors of EACH neuron. */
15   double *deltas;
16 } layer_t;
```

Complete the following functions:

1. `double sigmoid(double x) { ... }`

   This function implements the sigmoid function, Figure 2, following the equation:

   $$y = \frac{1}{1 + e^{-x}}$$

   Use the maths library function `double exp(double x)` to implement e^x; the library is already included in the header files.

   [1 Marks]

2. `layer_t *layer_create() { ... }`

   This function returns a new, heap-allocated *empty* layer. It only allocates `layer_t` setting all the integer properties to 0 and the pointers to `NULL`. If the layer allocation fails, it returns `NULL`.

   [2 Marks]

3. `bool layer_init(layer_t *layer, int num_outputs, layer_t *prev) { ... }`

   The initialisation function sets the properties of the given layer. As arguments it receives the layer to initialise, the number of outputs that layer has and the pointer to the previous layer, **prev**, which will be `NULL` if we are creating the input layer of a network. The function sets the number of outputs and allocates an array for the outputs of each neuron. If it is not the input layer, then it sets the number of inputs to the number of outputs of the previous layer and allocates the weights, biases and deltas arrays. Every neuron has a single bias and delta value. Outputs, biases and deltas arrays are set to 0s while weights are initialised randomly using the `ANN_RANDOM()` function. If any of the allocations fail, it returns `true` for failure; otherwise, the function returns `false` on success.

   [4 Marks]

4. `void layer_free(layer_t *layer) { ... }`

   Given a pointer to a `layer_t`, this function frees the resources allocated by `layer_init` function and the *layer itself. Hint: make sure you free anything you allocated in* `layer_init`.

   [2 Marks]

5. `void layer_compute_outputs(layer_t const *layer) { ... }`

   This function computes the outputs of a single layer according to the following equation described visually in Figure 1a:

   $$O_j = \sigma(B_j + \sum_i W_{ij} O_i) \tag{1}$$

   Equation 1 describes the *previous* layer's output O_i is weighted using the connection weight W_{ij} which joins neuron i from the previous layer with neuron j in the current layer. Then the bias B_j is added. The result is passed pass through the activation function σ to get the neuron output. Recall that we are using the `sigmoid` function as σ. This function is **not** called on an input layer as its outputs are the given inputs to the network.

   [3 Marks]

We recommend you attempt these after looking at PART 2 or PART 3.

6. `void layer_compute_deltas(layer_t const *layer) { ... }`

   To train the network we need a mechanism to know in which direction to update the weights. The *back-propagation algorithm* provides us a way to do that by using the chain rule of derivatives. Essentially, it tells what the *gradients* for a given weight and a given bias are so that we can adjust the corresponding weights and biases to minimise that error. This function computes those gradients, indicated by the *delta* symbol Δ, for each neuron in the layer according to the following derivation:

   $$\Delta_i = \sigma'(O_i) \sum_j W_{ij} \Delta_j \tag{2}$$

   Equation 2 weights the gradients from the *next* layer Δ_j using the connection weight W_{ij} and then multiplies with its own activation gradient $\sigma'(O_i)$. In other words, the partial derivatives are multiplied using the chain rule, i.e. back-propagates. An interesting property of the sigmoid function is that it's derivative can be expressed using itself; use the provided `sigmoidprime` for σ'. *Hint: gradient calculation doesn't use the incoming weights of the current layer.*

   [2 Marks]

7. `void layer_update(layer_t const *layer, double l_rate) { ... }`

   This function updates the weights and the biases of the layer so as to minimise the error; hence, it *learns*! Once we know what the gradient is, it is straightforward to update the weights using *gradient descent*:

   $$W_{ij} = W_{ij} + \alpha \times O_i \times \Delta_j \tag{3}$$
   $$B_j = B_j + \alpha \times \Delta_j \tag{4}$$

   Equation 3 *nudges* the weight W_{ij} in the direction that minimises the error by the given learning rate, indicated by the α symbol and passed as an argument `l_rate`. Note that the update is dependent on the *previous* layer output O_i that the weight connects from. The biases are updated according to equation 4.

   [1 Marks]

Part II Artificial Neural Network

In this part, you will put together layers to create an artificial neural network. The layers inside an `ann` form a **doubly linked list**; hence, traversing the list in the forward direction computes the output of the network while traversing backwards updates the weights during training.

To get an insight into what is going on you can always `make runtrain` to see the outputs of hidden layers, biases and whether you get any `segfaults`. We recommend you do this after incremental changes.

Listing 2: Definition of an ANN inside ann.h

```
1 /* Represents a N layer artificial neural network. */
2 typedef struct ann {
3     /* The head and tail of layers doubly linked list. */
4     layer_t *input_layer;
5     layer_t *output_layer;
6 } ann_t;
```

Complete the following functions:

1. `ann_t *ann_create(int num_layers, int *layer_outputs) { ... }`

   The function creates a new artificial neural network (ANN) with `num_layers` layers and the array `layer_outputs` number of neurons in each layer. For example, `layer_outputs[0]` gives the number of neurons in the input layer and `layer_outputs[1]` for the first hidden layer etc. The function first allocates memory for the `ann_t` itself, then allocates and initialises layers using `layer_create` and `layer_init` respectively. Assume those functions are working correctly without relying on your implementation. When creating the layers, `ann_create` also ensures they form a **doubly linked list**, making sure `prev` and `next` pointers are set correctly. If at any point allocations fail, it returns `NULL` to signal failure. *Hint: after allocating the ann, start with allocating the input layer.*

   [4 Marks]

2. `void ann_free(ann_t *ann) { ... }`

   Similar to `layer_free`, this function frees all the resources allocated by `ann_create` and also frees the *ann itself. Hint: follow the linked list to free the layers.*

   [2 Marks]

3. `void ann_predict(ann_t const *ann, double const *inputs) { ... }`

   This function follows the linked list structure and computes the outputs of layers given the initial `inputs`. It first sets the input layer outputs to the given inputs and then propagates the computation over the list of layers. For a visual representation, refer to Figure 1b. Use `layer_compute_outputs` assuming it is correct. If you implemented the layer functions, you can try to run **train** to see if it outputs random numbers. *Hint: the outputs of the input layer are the inputs to the neural network.*

   [2 Marks]

   We recommend you attempt PART 3 before attempting the following question.

4. `void ann_train(ann_t *ann, double *inputs, double *targets, double l_rate) { }`

   This function trains the neural network so that it *learns* to map the given `inputs` to the desired `targets`. After computing the current outputs, the function computes the gradients at the output layer as follows:

   $$\Delta_j = \sigma'(O_j)(T_j - O_j) \tag{5}$$

   Intuitively, equation 5 tells the output layer how far away it is from the desired target T_j. After computing the output layer deltas, the training function computes all the deltas in the *hidden layers* and then updates the weights of all layers *except the input layer*. Use `layer_compute_deltas` and `layer_update` assuming they are correct. If you did implement them, the **train** should work with final outputs giving the XOR function.

   [3 Marks]

Part III Memory Leaks

The skeleton code that you have, unfortunately, has a memory leak. Your task is to find and fix the memory leak inside **rdata.c** and you *only need to modify* that file for this part.

1. Inside **rdata.c** there is a memory leak. Use whatever means you prefer to locate the leak and fix the broken code. We suggest you use Valgrind's memcheck for which there is a make rule that will invoke Valgrind:

   ```
   $ make checkrdata # will compile and memcheck rdata.c
   ```

   Important: Place a comment by any change you make containing the keyword `LEAKFIX` so we can easily locate your change.

   [**4 Marks**]

You can use this paper for planning

You can use this paper for planning

You can use this paper for planning