



Top-Level Functions

The **main** program in **learn.c**

1. reads the options from the command line (**option.c**)
2. reads the training and test set (**getdata.c**)
3. runs the learning algorithm on the training set (**perceptron.c**)
4. calculates the loss on the training and test set (**perceptron.c**)

Activation Functions

`activation.c` implements a number of activation functions.

For each continuous function (e.g., `tanh`), its derivative (e.g., `dtanh`) is also implemented. It turns out that derivative is easier to compute from the output than from the input.

Noncontinuous functions (e.g., `signum`) do not have a useful derivative. Typically, 1 is used.

Neurons

`neuron.c` implements neurons.

`create_neuron` creates neurons.

`run_neuron` runs a neuron on inputs.

`backprop_neuron` calculates derivatives/updates.

`update_neuron` applies updates.

Learning Algorithms

To implement a different learning algorithm, just replace `perceptron.c`.

In `learn.c`, your network has type `void *`.

`perceptron.c` implements the functions that are declared in `network.h`.

`perceptron.c` uses casts to convert between `void *` and `Perceptron_Network` (aka PN).

Implementing Perceptrons

PN is a pointer to a structure, which includes an array of neurons.

`network_defaults` assigns defaults for options (called by `get_options`)

`network_create` creates and fills in a PN, and uses `create_neuron`.

`network_01_loss` calculates the 0-1 loss for a dataset. `network_loss` should calculate the loss the algorithm minimizes.

Implementing the Perceptron Learning Rule

`network_learn` implements the perceptron learning rule. For each example, for each neuron

1. obtain the output using `run_neuron`
2. obtain the desired output (changes 0 to -1)
3. if not equal, apply the learning rule by:
 - (a) calling `backprop_neuron` with the error derivative (opposite of direction of update)
 - (b) calling `update_neuron` to apply updates