CNS 187 - Neural Computation
Problem Sheet 1

Handed out:	28 Sept 00
Due:	6 Oct 00, 5pm

1.1 Computing with binary numbers using McCulloch & Pitts neurons (2 points)

One of the first milestones in the history of neural networks was Warren McCulloch and Walter Pitts' model of a neuron as a binary linear threshold unit. While the biological relevance of this model is dubious, in their 1943 paper, McCulloch and Pitts showed that any finite logical expression could be constructed using these units.

In this problem set, we will use McCulloch-Pitts units to build simple computational devices. For a more detailed discussion of linear threshold units, see Marvin Minsky's *Computation: Finite and Infinite Machines*. If these topics interest you, you might consider taking CNS188a and b.

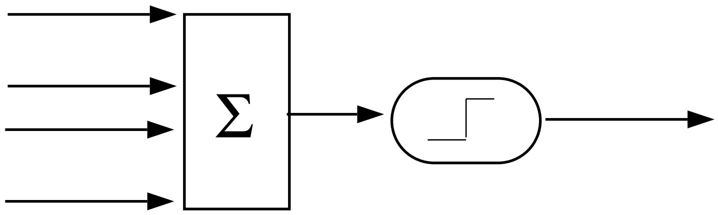


Figure 1: The workings of a McCulloch-Pitts unit

Every McCulloch-Pitts unit outputs a binary value (1 or 0, corresponding to 'active' or 'quiescent') based on the activity of its inputs. As shown in Figure 1, a McCulloch-Pitts unit performs two basic tasks in every time step. It first sums the activity of the input units, and then compares that sum to the unit's threshold value. When the total input is greater than or equal to the unit's threshold, the unit will be active in the next time step. Otherwise it will become inactive. The threshold is an intrinsic parameter of each unit, although it can vary among units.

Figure 2 shows a few simple McCulloch-Pitts units with their threshold specified in the center of the unit. The first two examples show how to build two of the standard logical gates with a M-P unit. An OR gate is simply a unit with a threshold of 1. Activity in either (or both) of the two inputs will cause the unit to output a 1. The AND gate looks about the same, but it has a threshold of 2, ensuring that only activity in both of the inputs will result in the unit's activation. Note that M-P units can have more than two inputs as well. To build an n-input OR or and n-input AND, we would need to put n input connections on a unit and set it's threshold to 1 (for OR) or n (for AND).

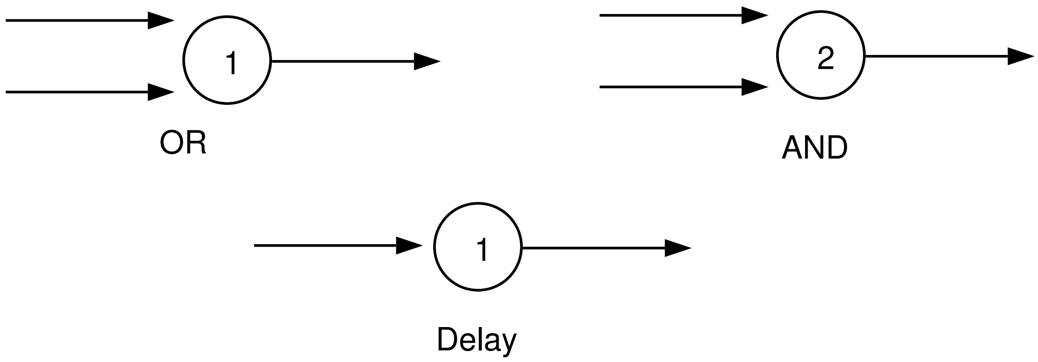


Figure 2: Some example McCulloch-Pitts units

The third example shows a delay circuit, that stores the value of its input during time t, and returns that same value, unchanged, at time $t+1$. Remember that the output of these units is based on the inputs from the previous time step. Therefore it takes one time cycle for information to propagate through any of these units. A simple delay could be useful if different bits of the input to a M-P unit are expected at different times. The bits that arrive earlier could be sent through a delay line, until the other inputs have arrived.

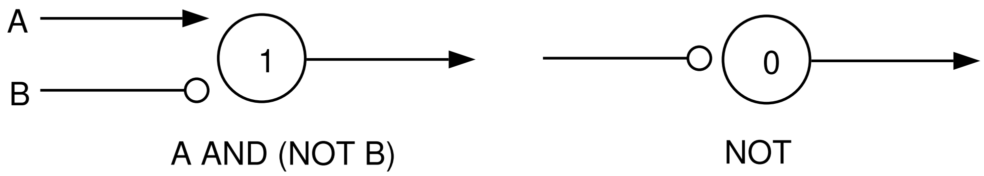


Figure 3: Inhibitory inputs

The M-P units, as we have described them so far, are rather limited because there are some very basic operations that a single unit cannot perform (e.g., NOT, XOR). Like real neurons, however, M-P units are not only restricted to excitatory connections. M-P units can have inhibitory connections (as shown in Figure 3). Some paradigms use absolute inhibition (a unit will output 0 any time one or more of its inhibitory inputs are active), and some use relative inhibition (an active inhibitory inputs sums as a -1; alternatively it can be thought of as raising the threshold by 1).

Formally, we can fully specify a network of n units by giving a vector θ of thresholds and a matrix W of connections, where W_{ij} is the number of excitatory connections from unit j to unit i, minus the inhibitory connections. For a connection with relative inhibition, W_{ij} is a negative integer; for a connection with absolute inhibition, we can use $W_{ij} = -\infty$, with the convention that $\infty \cdot 0 = 0$. Then the dynamics can be written:

$$g_i(t+1) = \sigma\left(\sum_{j=1}^{n} W_{ij} g_j(t) - \theta_i\right)$$

where $g_i(t)$ is the activity of unit i at time t, and $\sigma(x) = \{1 \text{ iff } x \geq 0; 0 \text{ otherwise } \}$.

1. A neural network can easily implement addition of two arbitrarily long numbers, assuming that the numbers are presented to the network starting with the lowest order digit first (from least to most-significant bit). Design a network that can perform such an addition using McCulloch & Pitts neurons with absolute inhibition (that is, a single active inhibitory input prevents the neuron from firing). Your network should have two input neurons, one for each input bit stream, representing the two number to be added. It should output a single bit stream, corresponding to the sum of the two inputs. The output can be delayed several time steps with respect to the input, but every subsequent time step should produce a new bit of output. Try to keep the network size small–you should be able to do this using 6 units.

   Hand in a diagram of your circuit, making sure you specify the threshold value of each unit. Show how your circuit will handle the following inputs: $A =$ `11101` $B =$ `01110` *(at t=1, input A is* `1` *and input B is* `0`*, at t=2, A is* `0` *and B is* `1`*,...)* producing (if the delay were 3 steps) the output $C = A + B$ (delayed) $=$ `0101011`**`000`**

2. Can you design another network with fewer units using relative inhibition (a unit fires if net excitation minus net inhibition equals or exceeds the threshold t)? Hand in a diagram of your new network, and work through the same example as before.

3. Now we will look at the multiplication of bit streams. Wire up a network of M-P neurons that can multiply two 2-bit long numbers, producing output as a bitstream (of up to 4 bits of content)? For this problem you can assume that in addition to the two input lines (A and B), your network has access to a clock signal that will have value `1` at t=1 (i.e., the first timestep of the input signal) and will be `0` at any other time. You can also assume that the input streams will have value `0` after the first two time steps. Hand in a diagram of your network and a description of how it works (you can either go through examples, or just verbally describe how it works).

4. Can you describe a similar finite network of McCulloch & Pitts neurons that performs multiplication of two arbitrarily long numbers, input as bit streams into two neurons? Justify your answer if you can or can't. Assume now that you know the two numbers are n bits long. Find and prove a lower bound for the number of internal states the network will need to be able to store to compute their product. (You don't have to prove that it is the best (biggest) lower bound, but the bound should be at least 2^n, within a constant factor.) If you generalize your 2-bit network to n bits, how many states does it have?

1.2 Gradient Descent (1.5 points)

You will soon be writing code to simulate artificial neural networks. Very often, this involves some form of parameter optimization with discrete steps, most commonly using a method called "gradient descent." The idea behind this is as follows: suppose you want to find the minimum of some multidimensional surface. Often the mathematical expression for the surface is too complicated to permit a direct algebraic solution, so you have to search for the minimum numerically. If you know the gradient of the surface (and thus the direction in which it goes down most steeply), then a seemingly sensible idea is to start with a "guess" for where the minimum is, and then improve the guess by moving it in the direction of steepest descent; when you can't improve the guess anymore, then you are at a minimum (possibly a local one, of course). We write "seemingly" because we will see in later problem sets that this is often very slow.

This problem and the next one ("Euler Integration") are intimately related, so you might want to look them over before starting in earnest; that will give you an idea of what techniques to use.

Let's first try this with a *very* simple surface, a one-dimensional bowl whose height is:

$$E = x^2/2 \tag{1}$$

(Clearly, the minimum is at $x = 0$, but pretend you don't know that right now – in future problem sets you will test this technique on surfaces you can't solve analytically, rest assured!)

1. Write a program, using `gradesc.m` provided in the MATLAB package, that implements gradient descent for this surface. Initialize x at a random position. Then at each time step,

   - find the gradient $\frac{\partial E}{\partial x}$ at the current position.
   - take a step in a direction negative to the gradient, such that

   $$x_{n+1} = x_n - \eta \frac{\partial E}{\partial x} \tag{2}$$

   η is called the "time step". Try running this program for various values of η. Do not hand in Matlab code unless the problem set specifically asks for it.

   Hand in three plots of x versus n (for different values of η) that show qualitatively different behaviors. From your numerical experiments, for what range of values of η does this program converge to the correct result?

2. Analyze the procedure to give a theoretical prediction for the values of η which would lead to such results. Does it coincide with your numerical values?

3. Now let's get just a little fancier. Define a new surface by

   $$E(x,y) = \frac{1}{25}(13x^2 + 24xy + 13y^2) \tag{3}$$

   Plot the surface in MATLAB (but don't hand in this plot). Where is the global minimum?

4. Simulate gradient descent for it, so that now

   $$\begin{aligned} x_{n+1} &= x_n - \eta \frac{\partial E}{\partial x} && (4) \\ y_{n+1} &= y_n - \eta \frac{\partial E}{\partial y} && (5) \end{aligned}$$

   Hand in three plots, each for a different η, to show qualitatively different behaviors.

5. Repeat the numerics that you did for $x^2/2$, so as to find the values of η for which the procedure doesn't converge. Find a theoretical prediction for this value. (Hint: find x_n and y_n as a function of n.)

   Since we've gone this far, we might as well go a little further and consider a general surface $E(\vec{x})$ where $\vec{x}$ is a column vector with N components $(x_1,\ x_2,\ \ldots x_N)$. (The problem you just did would be an example of such a surface, with $\vec{x}$ having 2 components.) Define a vector $\frac{\partial E}{\partial \vec{x}}$ whose i-th component is

$$\left(\frac{\partial E}{\partial \vec{x}}\right)_i = \frac{\partial E}{\partial x_i} \tag{6}$$

   This vector is just a shorthand way of writing the gradient. The gradient descent procedure would now be

$$\vec{x}_{n+1} = \vec{x}_n - \eta \frac{\partial E}{\partial \vec{x}} \tag{7}$$

   Let

$$H_{ij} \equiv \frac{\partial^2 E}{\partial x_i \partial x_j} \tag{8}$$

   The matrix $\vec{H} = [H_{ij}]$ is called the *Hessian.*

   The Taylor series expansion for E about some point $\vec{x}_0$ is

$$E(\vec{x}) = E(\vec{x}_0) + \left(\frac{\partial E}{\partial \vec{x}}\right)^T (\vec{x}-\vec{x}_0) + \frac{1}{2}(\vec{x}-\vec{x}_0)^T \vec{H}\,(\vec{x}-\vec{x}_0) + \text{higher order terms} \tag{9}$$

   (with both the gradient and the Hessian evaluated at $\vec{x}_0$.)

6. If $\vec{x}_{min}$ is a local minimum, and we are close enough to it that we can ignore the higher order terms in the Taylor expansion, predict the maximum value of η for convergence in the gradient descent procedure, in terms of the eigenvalues of the Hessian $\vec{H}$ at $\vec{x}_{min}$. You can assume, without loss of generality, that $\vec{x}_{min} = \vec{0}$; this makes manipulating the equations a little more convenient. (Reminder: what's the value of the gradient vector at the minimum?)

7. (optional) Try this for fun: suppose we are at some point $\vec{x}_n$, and that $\vec{x}_n$ is indeed close enough to the minimum that higher order terms in the Taylor expansion can be ignored, and that we know what $\vec{H}$ is at $\vec{x}_n$. Assume that $\vec{H}$ is invertible. Can you find a closed-form expression for how much we should add to $\vec{x}_n$ so as to jump *directly* to the minimum, in a single bound?

1.3 Euler integration (1.5 points)

It's time to look in more detail at the algorithm you used in the previous problem to implement gradient descent. In the case of the two dimensional surface, you might have imagined water flowing downhill, or a ball rolling downhill to the bottom. This illustrates an important theme in this class, namely, finding physical systems whose intrinsic dynamics carry out a computation. In the case of gradient descent, this way of thinking suggests a continuous version of gradient descent, where

$$\frac{\partial \vec{x}(t)}{\partial t} = -\frac{\partial E(\vec{x}(t))}{\partial \vec{x}} \tag{10}$$

In a sense, this differential equation represent the *true* dynamics we are trying to incorporate into the algorithm used in problem 1.2, which we now see boils down to numerically integrating the differential equation using a difference equation approximation. In the more general context of ordinary differential equations, this naive algorithm is known as *Euler's method.* It approximates any differential equation of the form

$$\frac{dy}{dt} = f(y) \tag{11}$$

with the difference equation

$$\frac{\Delta y}{\Delta t} = f(y) \tag{12}$$

which can be trivially solved for Δy,

$$\Delta y = f(y)\Delta t \tag{13}$$

Based on this difference equation, we can iterate our simulation forward one step at a time.

$$y_{t+\Delta t} = y_t + f(y_t)\Delta t \tag{14}$$

The value of Δt is known as the *step size* of the simulation. As $\Delta t \to 0$, the difference equation becomes a more and more accurate model of the real differential equation (until the effects of finite machine precision come into play).

We can convert any high–order ordinary differential equation[1] to a multidimensional first–order ordinary differential equation, and thus simulate it with Euler's method. This is done by creating a state vector

$$\vec{y} = \begin{pmatrix} y \\ y' \\ y'' \\ \vdots \end{pmatrix} \tag{15}$$

The primes denote differentiation by time. Notice that Euler's method can be applied to vectors just as easily as scalar variables:

$$\Delta\vec{y} = f(\vec{y})\Delta t \tag{16}$$

For example, the linear differential equation $y'' + y' + y = 0$ would be written in vector form as

$$\begin{aligned} \vec{y}' &= \vec{M}\vec{y} \\ \begin{pmatrix} y' \\ y'' \end{pmatrix} &= \begin{pmatrix} 0 & 1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} y \\ y' \end{pmatrix} \end{aligned} \tag{17}$$

[1] or even a multidimensional higher–order ODE

1. Show that in the general linear case $\vec{y}' = \vec{M}\vec{y}$, where $\vec{y} = (y_1, y_2, y_3, \ldots, y_N)^T$,

$$\vec{y}_{n\Delta t} = (\vec{I} + \vec{M}\Delta t)^n \vec{y}_0 \tag{18}$$

2. Convert the following differential equations into the state vector form above:

   a) $y'' + y = 0,$ $\quad y(0) = 1,\ y'(0) = 0$
   b) $y'' + 0.1y' + y = 0,$ $\quad y(0) = 1,\ y'(0) = 0$
   c) $y'' + 101y' + 100y = 0$ $\quad y(0) = 1,\ y'(0) = -1$

   Find the closed form solutions for these equations using the given initial conditions.

3. Using the program `eulerint.m` provided in the MATLAB package, write a program which can simulate the differential equations above using the given initial conditions. Plot $y(t)$ versus $y'(t)$ for each equation, using a reasonable value for Δt.

4. For each equation, what is the largest step size ($\Delta t_{\max}$) for which you get non-divergent behavior? Determine this numerically by trying different values of Δt.

   There are two related but distinct issues to consider here: instability (i.e. unboundedness as time increases) and accumulated error during stable operation. The second issue is much more involved than the first, and the relationship between discrete and continuous dynamical systems has evolved into a branch of applied mathematics in its own right. For the purposes of this homework set, as will become clear when you have solved 1), 2), and 3) analytically, we are only concerned here with the first issue: instability.

5. Using the same techniques as in problem 1.2, analyze your procedure to obtain the theoretical maximum stepsize, $\Delta t_{\max}$, for which the Euler method converges. That is, we want to know the $\Delta t_{\max}$ for which the solution doesn't explode (go infinite) as $t \to \infty$. How do the analytically obtained values compare with the empirical ones obtained from simulation?

For more thoughts on accumulated error, consult *Numerical Recipes in C*, by W. Press et al. (Cambridge: Cambridge University Press).

1.4 APPENDIX: NUMERICAL INTEGRATION

(N.B. This section contains no questions for you to answer, merely some additional notes on integrating ODEs. Your CNS187 TAs would be remiss if we let you solve problem 1.3 without a discussion of some numerical techniques other than the Euler method (which is simple, but not very accurate). This discussion is intended as a supplement to problem 1.3. You should answer the questions in problem 1.3 as written, but it would be instructive (and not that much work) to try out some of the techniques presented here. For more detail, consult chapter 8 of *Elementary Differential Equations and Boundary Value Problems* by W.E. Boyce and R.C. DiPrima. For even more detail, see chapter 7 of *Introduction to Numerical Analysis* by J. Stoer and R. Bulirsh.

Consider the initial value problem

$$y' = f(y), y(x_0) = y_0, \tag{19}$$

which we will denote IVP. This is just a single first order equation, but all the techniques discussed here apply to systems (where y is a vector) as well. Let the solution of this ODE be

$$y_n = \phi(x_n) \tag{20}$$

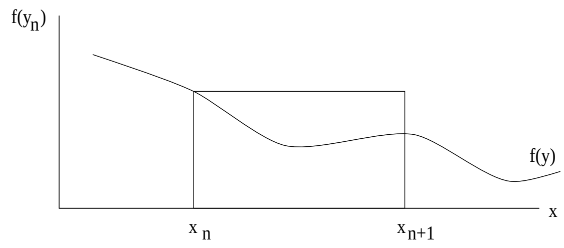
We can integrate IVP from one step to the next and obtain

$$\phi(x_{n+1}) = \phi(x_n) + \int_{x_n}^{x_{n+1}} f(\phi(x))dx \tag{21}$$

If we knew what the value of the integral was, we would have a solution for ϕ. Since we don't, we have to resort to approximation. The Euler method makes the simplest possible approximation:

$$\int_{x_n}^{x_{n+1}} f(\phi(x))dx \approx hf(y_n), \tag{22}$$

where $h = x_{n+1} - x_n$ is the stepsize (denoted Δt in problem 1.2). Graphically,



One obvious improvement on this is to use a trapezoidal approximation:

$$\int_{x_n}^{x_{n+1}} f(\phi(x))dx \approx (h/2)(f(y_n) + f(y_{n+1})) \tag{23}$$

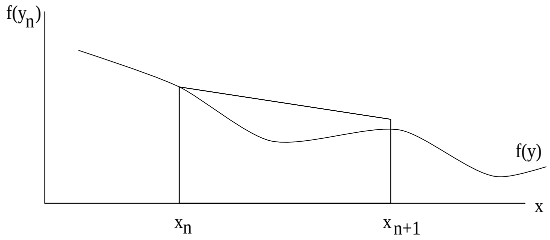
How do we know $f(x_{n+1})$ you're wondering? Once again, we make an approximation (look familiar?):

$$f(y_{n+1}) \approx f(y_n + hf(y_n)) \tag{24}$$

Putting all this together, we obtain the *Heun* formula

$$y_{n+1} = y_n + (h/2)[f(y_n) + f(y_n + hf(y_n))], \tag{25}$$

which, remarkably, has a discretization error of $O(h^3)$ as opposed to $O(h^2)$ for the Euler method. (An approximation is called n-th order if its error term is $O(h^{n+1})$. Thus, Euler is *first-order*, Heun is *second-order*.) Graphically,



Higher order methods can be obtained using similar approximations. Numerical accuracy and efficiency are always a concern when doing simulations. When solving problem 1.3, it would be helpful to write your code so that you can try various integrations techniques just by calling different functions. Even higher order methods, such as fourth order Runge-Kutta, although somewhat tricky to derive, are not difficult to implement.