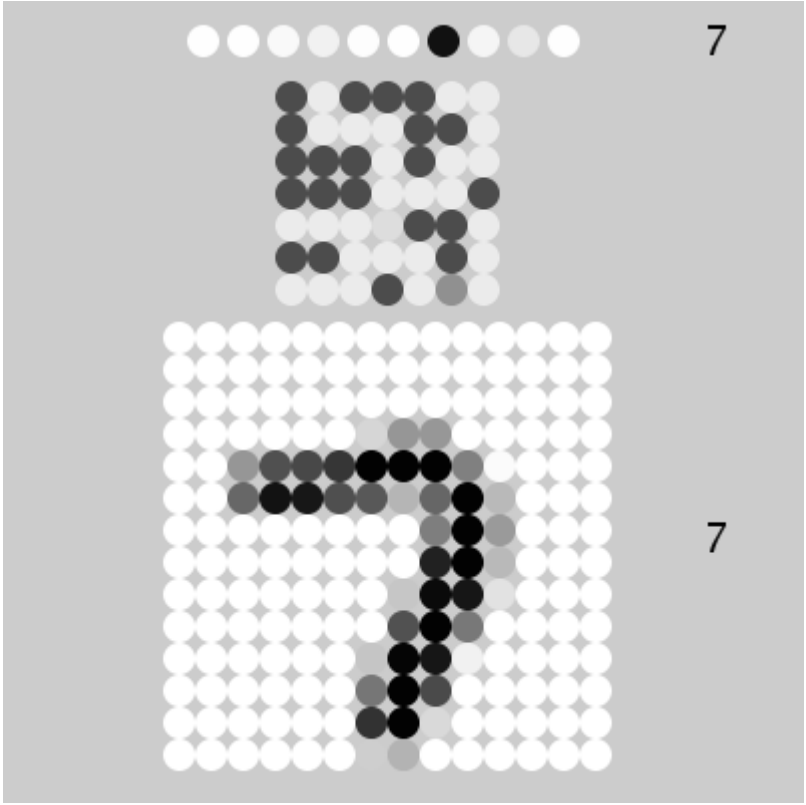
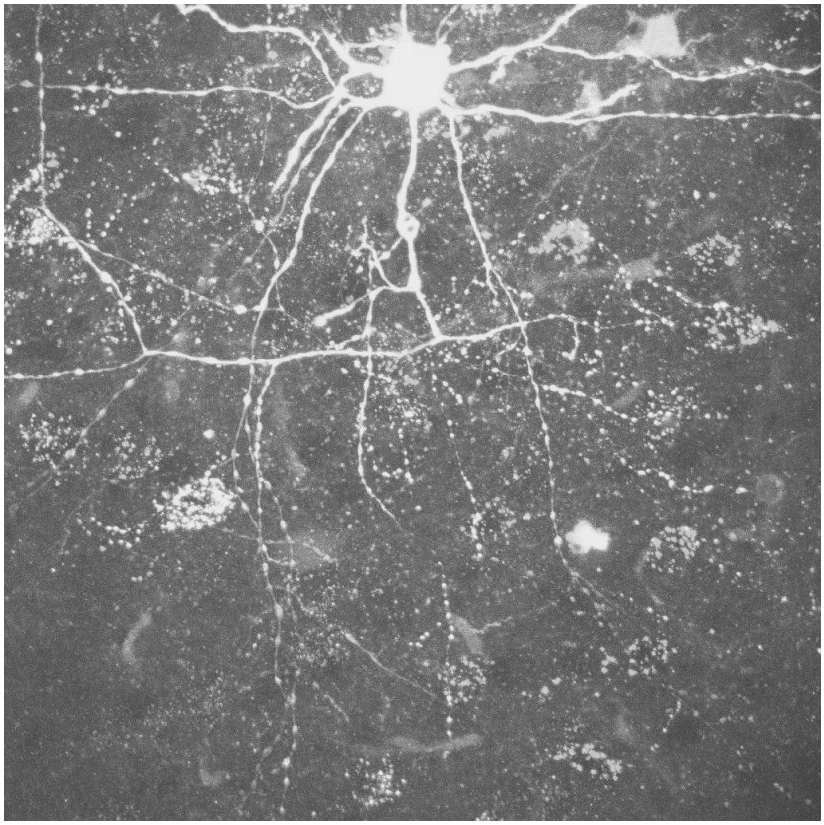
Week 6: Artificial Neural Networks

In this lesson, I would like to create an artificial neural network, based loosely on the way in which the brain functions. This particular neural network will attempt to identify numbers written by hand, and 'say' the result.



Theory

Firstly, let us very quickly sketch how the brain *does* function. A simple model of the brain has it as receiving inputs from the outside world via neurons, processing those inputs via further neurons, and acting in response. For example, if I see the number '7' (the input) and say aloud '7' in response (the output), we can think of the process from light entering the eye to the word '7' being formed. We might assume there is an image of the number '7' formed on the retina. The photons entering the eye trigger a chemical reaction, which cause neurons connecting the eye to the visual cortex in the brain to 'pulse' or 'fire', by transmitting an electrical signal along their length. These neurons connect to others via a web of dendrites. The tips of the dendrites release a chemical, which in turn causes other neurons in the visual cortex to 'fire', and the chain continues, with neurons either exciting or inhibiting other neurons. Eventually, activity wells up around the neurons which control breathing, the tongue and the muscles in the voice box, these too fire, chemicals are released which cause contraction of the muscles at the appropriate time, and the word '7' is pronounced.

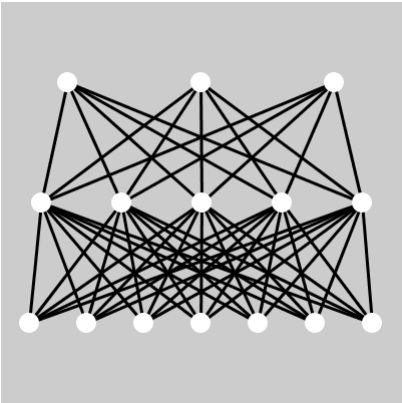


Neural soma with dendrites extending © neurollero, 2005

In this very simple description, some activity happens somewhere, before the result is passed onto a next stage. So there might be some activity in sensory neurons in the retina, some activity in inter-neurons in the visual cortex, some activity linking numeracy to language, followed by activity in Broca's area (responsible for language production), followed by some transmission to motor neurons controlling the mechanics of speech production. We might consider this as a transition through a number of 'layers' of neurons, each passing its result to the next area of the brain.

To make a simple artificial neural network model of this activity, we might construct a set of 'input' neurons, representing the sensory neurons in the retina, laid out in a grid. Then we might connect these neurons to a 'hidden' layer within the brain. Next, this hidden layer may connect to an 'output' layer, equivalent to a signal to motor neurons to vocalise a certain number.

The way we will connect the neurons together will be to connect all the input neurons to all the hidden layer neurons, and all the hidden layer neurons to the output neurons. Like this:



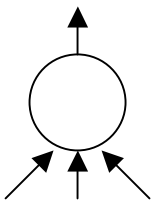
Output layer

Hidden layer

Input layer

Notice that unlike the neurons in the brain, the neurons in each layer do not link directly to each other. Each layer simply feeds forward into the next layer.

Rather than have neurons as such in each layer, we will have 'perceptrons'. A perceptron will take a number of inputs, sum them, and if their combined value is above some threshold, 'fire':



Output

Perceptron

Inputs

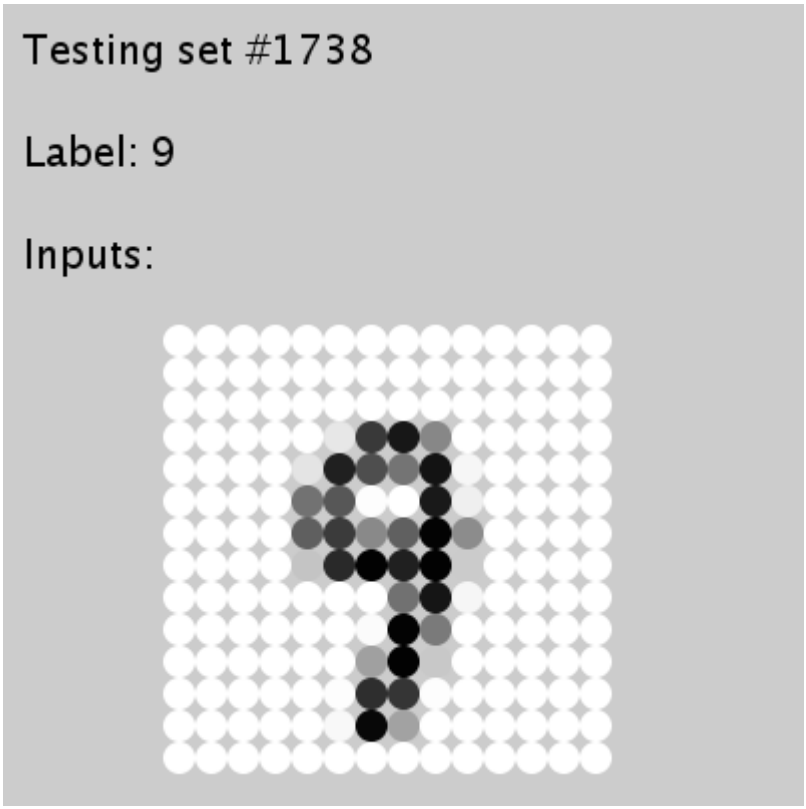
We will weight each input, so the sum is a weighted sum. Those inputs with higher weights will contribute more to reaching the threshold.

Practice

At this stage, we have enough details to begin writing some code. There will be further details related to how this system can learn to recognise numbers later, but first, let us set up a conceptual scheme in Processing. Let us create five tabs:

1. Main tab. This will allow us to draw, 'train' and 'test' our neural network.
2. Load data. This tab will allow us to load a set of labelled 'flash cards'. Each flash card will have a handwritten number on it for the neural network to recognise, and a label on the back to tell us which number it actually is.
3. Neuron. This will be the code for an individual perceptron as drawn above.
4. Network. This will consist of a number of arrays of neurons. One for the input layer, one for the hidden layer, and one for the output layer.
5. Sigmoid. The purpose of this tab is to construct a 'how to fire a neuron' function. There will be more detail later.

The first thing I would like to do is simply load the 'flash cards' and display them. To display them, I will create the input layer of neurons. This layer will be arranged as a grid, and simply 'fire' according to the pattern on the flash card:



You will notice that my neurons 'fire' not just 'black' (on) and 'white' (off), but also shades of grey in between. We will skip over 'how is this possible', and simply say: our perceptrons will not just be 'on' or 'off', they will have a range from +1 (black, fully on) through 0 (grey, partially on, partially off) to –1 (white, fully off).

It is also worth noting that our neurons *stay* in this state. Unlike the brain, our artificial neural network will simply wait, neurons either on or off (or even partially-on partially-off), until the next flash card is displayed.

Loaddata

These details aside, to the data. I have already prepared a set of flash cards. Each one has a 14 x 14 pixelated version of a handwritten digit on it. The cards were origianally taken from the National Institute of Standards and Technology (USA). I have created resampled versions of the MNIST processed versions available from Yann LeCun and Corinna Cortes.

The cards are in a file called:

t10k-images-14x14.idx3-ubyte

('10k' is '10,000': I have 10,000 images in the set)

Alongside the cards, I also have a set of 10,000 labels in a file called:

t10k-labels.idx1-ubyte

You will need download these files and create a 'data' folder for your sketch to hold them. You can create your 'data' folder by showing the sketch folder (available on the Processing tools menu), and then making a new folder.

Now we can write some code to load these files. Firstly, the flash cards themselves: these we will split into two.

a) A 'training' set of 8,000 cards to show to our computer, and to teach it the numeric label for the card.
b) A 'test' set of 2,000 cards. These will be 'exam' questions for our computer. It will not know the answers, and will have to guess the correct label. It will not be told the correct solution.

```
class Card
{
  float [] inputs;
  float [] outputs;
  int label;
  Card()
  {
    inputs = new float [196];
    outputs = new float[10];
  }
}

Card [] training_set;
Card [] testing_set;
```

Notice that each card has a set of inputs, the 14 x 14 grid laid out in a row of 14 x 14 = 196 perceptrons, and it also has outputs, the answer the neural network is supposed to give as 10 'sample' perceptrons, one for each possible answer (the digits from 0 to 9). The label is simply a numeric backup to the outputs.

We will now add a function to read images which are presented as a set of raw byte values.

```
class Card
{
  // ...
  void imageLoad(byte [] images, int offset)
  {
    for (int i = 0; i < 196; i++) {
      inputs[i] = int(images[i+offset]) / 128.0 - 1.0;
    }
  }
}
```

`images` will be an array of bytes, each one representing a pixel on one of 10,000 14 x 14 images. That is, an array of 1,960,000 bytes. The 'offset' parameter says where this particular 14 x 14 image on this particular card starts in the array. Each byte runs from 0 to 255 and represents a greyscale

colour. We first each byte to an integer using the `int()` function. This is one of the rare cases where we must use the conversion *function*, not cast the byte to an integer (which is a more literal conversion). Once we have an integer, it is scaled from 0 to 2 by dividing through by 128, and then, finally, from +1 to –1 by subtracting 1. There are 196 of these neuron values for each image.

In addition to loading the images, we will also write a function to prepare the 'result' that the neural network should show, an array of ten outputs, with the correct answer highlighted as +1, and the other nine incorrect answers labelled –1:

```
class Card
{
  // ...
  void labelLoad(byte [] labels, int offset)
  {
    label = int(labels[offset]);
    for (int i = 0; i < 10; i++) {
      if (i == output) {
        outputs[i] = 1.0;
      }
      else {
        outputs[i] = -1.0;
      }
    }
  }
}
```

This time, `labels` is an array of 10,000 bytes, each one representing the label for the card in question. We convert this to an integer label again using the `int()` function, before then going through the 10 template outputs and highlighting the correct answer within them.

The next thing to do is to load the data from the files into the two arrays of cards, one training set and one test set.

The code to achieve this is as follows:

```
class Card()
{
  // ...
}

Card [] training_set;
Card [] testing_set;

void loadData()
{
  byte [] images = loadBytes("t10k-images-14x14.idx3-ubyte");
  byte [] labels = loadBytes("t10k-labels.idx1-ubyte");

  training_set = new Card [8000];
  int tr_pos = 0;
  testing_set = new Card [2000];
  int te_pos = 0;

  for (int i = 0; i < 10000; i++) {
    if (i % 5 != 0) {
```

```
      training_set[tr_pos] = new Card();
      training_set[tr_pos].imageLoad(images,16 + i * 196);
      training_set[tr_pos].labelLoad(labels,8 + i);
      tr_pos++;
    }
    else {
      testing_set[te_pos] = new Card();
      testing_set[te_pos].imageLoad(images,16 + i * 196);
      testing_set[te_pos].labelLoad(labels,8 + i);
      te_pos++;
    }
  }
}
```

I will not dwell on this code too long. It uses the Processing 'loadBytes' function to load both images and labels. It then goes through both simultaneously, matching up image and labels with a new card. Four out of every five cards is assigned to the training set, while every fifth card is assigned to the test set instead.

We can set the main tab to load the data as follows:

```
void setup()
{
  loadData();
}
```

If you play this sketch, it should run successfully without errors. Although we cannot set it yet, the data has been loaded.

Neuron and network (part 1: input layer)

For the next step, I would like to create a set of neurons to display the inputs on the card. We will create a grid of 14 x 14 input neurons, which will do nothing else apart from directly 'output' the inputs.

These simple neurons will look like this:

```
class Neuron()
{
  float m_output;
  void draw()
  {
    fill(128 * (1 - m_output));
    ellipse(0,0,16,16);
  }
}
```

That is, they will just perform the inverse operation of loading the data: they will convert a number scaled from +1 to –1 to a number between 0 and 255! Notice though that there is an inversion: +1 is shown as black (0) and –1 as white (255). This is to achieve a pen (black) on paper (white) effect.

For now, this will be the entire neuron. Next we will create a neural network of them:

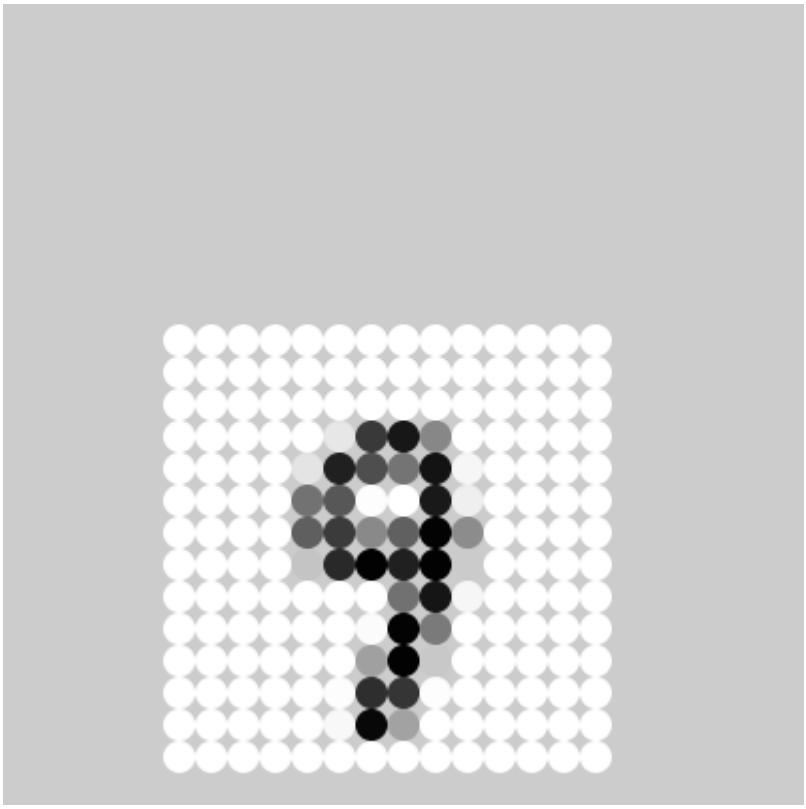
```
class Network
```

```
{
  Neuron [] m_input_layer;
  Network(int inputs)
  {
    m_input_layer = new Neuron [inputs];
    for (int i = 0; i < m_input_layer.length; i++) {
      m_input_layer[i] = new Neuron();
    }
  }
  void respond(Card card)
  {
    for (int i = 0; i < m_input_layer.length; i++) {
      m_input_layer[i].m_output = card.inputs[i];
    }
  }
  void draw()
  {
    for (int i = 0; i < m_input_layer.length; i++) {
      pushMatrix();
      translate(
        (i%14) * width / 25.0 + width * 0.22,
        (i/14) * height / 25.0 + height * 0.42);
      m_input_layer[i].draw();
      popMatrix();
    }
  }
}
```

This class has three functions. The first is simply to set up a network with a given number of inputs (in this case it will be 196). There is then a function to respond to a card it has been shown. In this case, the inputs on the card are simply copied directly to the outputs of this layer of neurons. Finally, there is a function to draw 196 neurons as a 14 x 14 grid. This uses the same layout principle that we used last time for displaying the genetic algorithm solutions.

We can test it all works by altering the code in the main tab:

```
Network neuralnet;

void setup()
{
  size(400,400);
  loadData();

  neuralnet = new Network(196);

  neuralnet.respond(testing_set[1738]);

  background(220,204,255);
  noStroke();
  smooth();
  neuralnet.draw();
}
```



Neuron (part 2: feed forward network)

Now that we have a working input layer, I would like to extend the neural network so that it feeds the input layer into a 'hidden' layer, and the 'hidden' layer into an output layer.

To do this, I will extend the neuron class to respond to the weighted sum of its inputs. That response will be fed to the next layer.

Firstly, let us change the neuron class so that it can be initialised in one of two ways: either as an input neuron, which as no inputs itself, or as a hidden or output layer neuron, both of which will have inputs from the layer below.

```
class Neuron
{
  Neuron [] m_inputs;
  float [] m_weights;
  float m_output;
  Neuron()
  {
  }
  Neuron(Neuron [] inputs)
  {
    m_inputs = new Neuron [inputs.length];
    m_weights = new float [inputs.length];
    for (int i = 0; i < inputs.length; i++) {
      m_inputs[i] = inputs[i];
      m_weights[i] = random(-1.0,1.0);
    }
  }
  void respond()
  {
```

```
    float input = 0.0;
    for (int i = 0; i < m_inputs.length; i++) {
      input += m_inputs[i].m_output * m_weights[i];
    }
    m_output = lookupSigmoid(input);
  }
  void draw()
  {
     // ...
  }
}
```

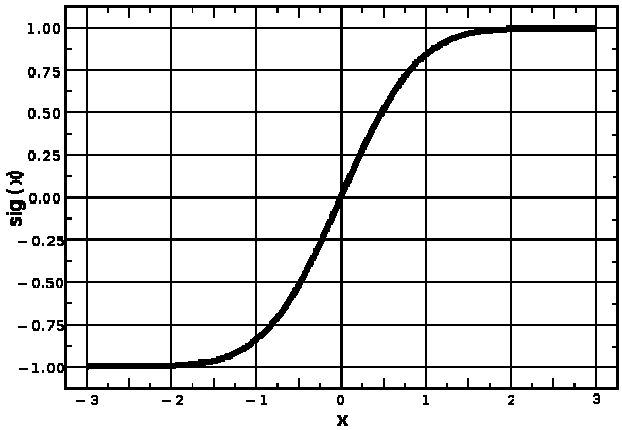
The first constructor is used for the input layer neurons: each of these simply responds directly to the flash card it is shown. However, both the hidden layer and output layer are more complicated. They have a string of neurons and weights, as well as a threshold above which they 'fire'.

This is the effect of the respond function. Each current input is related to the weighted sum of the outputs of the preceding layer. Rather than be too precise about how the threshold works, for this example, I will simply suggest that the sum of the inputs has some output associated with it. For now, I have assumed that the neuron fires according to a function called 'lookupSigmoid', based on its input.

Let us quickly take a detour, and implement a sigmoid function:

Detour: the sigmoid function

The sigmoid function is a curious addition to a neural network. When neural networks first started out, perceptrons were literally 'on' or 'off', there was no in between. However, as people wanted to train perceptrons, they realised that a simple 'on' or 'off' makes it difficult to adjust. The very 'on' / 'off' nature of the perceptrons meant that results swung rapidly as the inputs were adjusted. Instead, they wanted a continuous output. The function they chose is called a 'sigmoid'. It looks like this:



That is, when the weighted sum of the inputs is around 3, then the output is around '1' (firing). When the weighted sum of the inputs is around –3, then the output is around '–1' (not firing). At around 0, then we get the 'partially on' 'partially off' situation again.

The sigmoid function is written mathematically like this:

$$y = 2 / (1 + e^{-2x}) - 1$$

'x' is the input and 'y' is the output, in terms of the input. The function has an interesting property that we will use, in that its differential is a function of the value of y, that is, the amount it is changing is determined by the current output. When the output is close to –1, it changes very slowly. The amount of change increases as the output reaches 0, and then it drops off again until the output is near +1, when it changes very slowly once again. The differential (the rate at which the curve changes) is:

$$dy/dx = (1 - y)(1 + y)$$

Now, the sigmoid function is quite difficult to calculate, so what people tend to do is to precalculate, and then look up a value close to the one they want. We will do exactly that in the sigmoid tab:

```
float [] g_sigmoid = new float [200];

void setupSigmoid()
{
  for (int i = 0; i < 200; i++) {
    float x = (i / 20.0) - 5.0;
    g_sigmoid[i] = 2.0 / (1.0 + exp(-2.0 * x)) - 1.0;
  }
}

float lookupSigmoid(float x)
{
  return
    g_sigmoid[constrain((int)floor((x + 5.0) * 20.0),0,199)];
}
```

This looks quite complicated, but it is simply the code to evaluate the sigmoid function from weighted inputs of –5 to +5, and to store that in an array called '`g_sigmoid`'. Whenever we need to find a value for the sigmoid function, we simply look it up, rather than perform a complex calculation.

In order to ensure it has been precalculated, we need to add the setupSigmoid function to the setup function in the main tab:

```
void setup()
{
  loadData();
  setupSigmoid();
  // ...
}
```

With the sigmoid in place, we can now return to the feed forward network:

Network (part 2: feed forward network)

We already have a neuron capable of taking its inputs from the layer below and preparing an output using the sigmoid function. Now we simply need to join those layers together.

We will need to adjust the network class to add hidden and output layers, as well as adjusting the 'respond' function, so each layer of neurons responds to those in the layer below.

Firstly, the construction:

```
class Network
{
  Neuron [] m_input_layer;
  Neuron [] m_hidden_layer;
  Neuron [] m_output_layer;
  Network(int inputs, int hidden, int outputs)
  {
    m_input_layer = new Neuron [inputs];
    m_hidden_layer = new Neuron [hidden];
    m_output_layer = new Neuron [outputs];
    for (int i = 0; i < m_input_layer.length; i++) {
      m_input_layer[i] = new Neuron();
    }
    for (int j = 0; j < m_hidden_layer.length; j++) {
      m_hidden_layer[j] = new Neuron(m_input_layer);
    }
    for (int k = 0; k < m_output_layer.length; k++) {
      m_output_layer[k] = new Neuron(m_hidden_layer);
    }
  }
  // ...
}
```

Note how every neuron in each layer uses the previous layer of neurons as its input.

Next, the response is fed forward through the layers:

```
class Network
{
  // ...
  void respond(Card card)
  {
    float [] responses = new float [m_output_layer.length];
    for (int i = 0; i < m_input_layer.length; i++) {
      m_input_layer[i].m_output = card.inputs[i];
    }
    // now feed forward through the hidden layer
    for (int j = 0; j < m_hidden_layer.length; j++) {
      m_hidden_layer[j].respond();
    }
    for (int k = 0; k < m_output_layer.length; k++) {
      m_output_layer[k].respond();
    }
  }
  // ...
}
```

Note that the input layer copies directly from the card inputs as before, but that the hidden layer and then the output layer respond using the neuron's respond function.

Finally, we should draw the entire network:

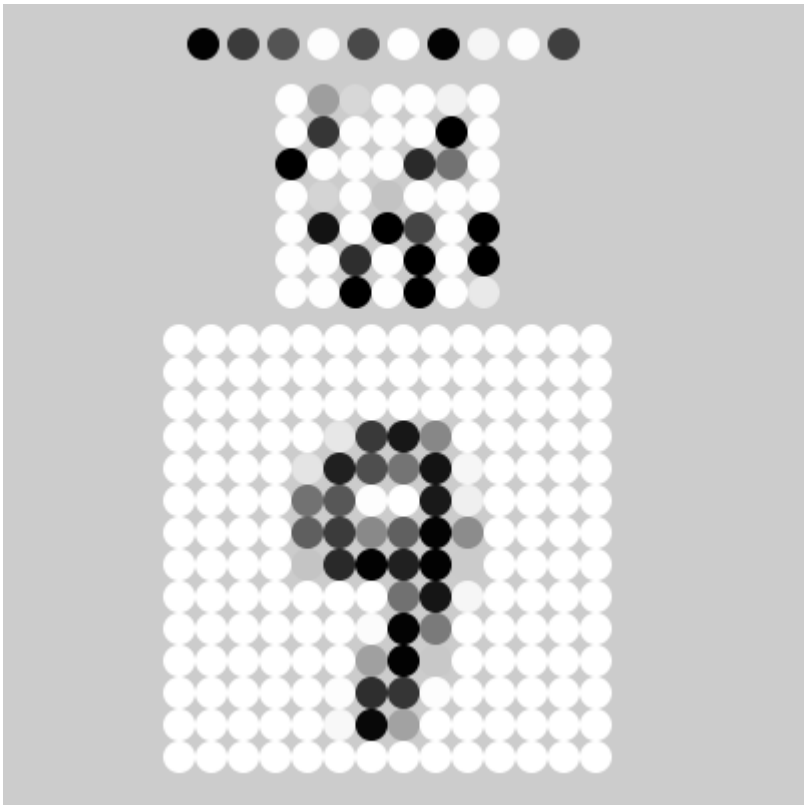
```
class Network
{
  // ...
  void draw()
  {
    for (int i = 0; i < m_input_layer.length; i++) {
      pushMatrix();
      translate(
        (i%14) * width / 25.0 + width * 0.22,
        (i/14) * height / 25.0 + height * 0.42);
      m_input_layer[i].draw();
      popMatrix();
    }
    for (int j = 0; j < m_hidden_layer.length; j++) {
      pushMatrix();
      translate(
        (j%7) * width / 25.0 + width * 0.36,
        (j/7) * height / 25.0 + height * 0.12);
      m_hidden_layer[j].draw();
      popMatrix();
    }
    for (int k = 0; k < m_output_layer.length; k++) {
      pushMatrix();
      translate(
        ((k+9)%10) * width / 20.0 + width * 0.25,
        height * 0.05);
      m_output_layer[k].draw(true);
      popMatrix();
    }
  }
}
```

The first two layers as drawn are relatively straightforward. They use the same gridding procedure as before. The output layer has a slight twist: the modulo operator as applied has the effect of drawing the neurons in the order 1,2,3,4,5,6,7,8,9,0 rather than 0,1,2,3,4,5,6,7,8,9 as they would normally be drawn. This is helpful when we come to the next stage, the trained network, as it will allow a more natural way to read the output number (in the order, for example, as the numbers on your keyboard).

All that now needs adding is a change in the setup function to call the revised Network constructor:

```
void setup()
{
  // ...
  neuralnet = new Network(196,49,10);
  // ...
}
```

The output should now looks something like this:



We do not draw all the linkages between the layers in this diagram, as there would be far too many to see anything meaningful. What we see is the current state of the neurons in each layer. In the input layer, as we might expect, the values on the flash card itself are drawn. The next grid up shows the hidden layer, with outputs based on multiplying the inputs by weights. Finally, above that, in a row, is shown the output layer. This is meant to show the neural network's response to the input. It is currently showing a mess. We might interpret this display as, "I think maybe it is a 1, or maybe a 2, quite possibly a 3, not a four, could well be a 5, not a six, could well be a 7, not an eight or a nine, but it could be a 0". Quite clearly, the net is guessing, and guessing badly. This looks like a '9' to me!

The next step must be to train the network, so that if it sees a '9', it says out loud a '9'.

Neuron (part 3: back propagation)

The training of neural networks is quite contentious, mainly because it is completely arbitrary. It bears no relationship whatsoever to how the brain actually works, and is simply intended as a good working way to achieve a result.

The basic idea is to use Newton's method of optimisation. That is to follow a gradient down towards the correct answer. The gradient towards the answer is given by differentiating the answer you currently have, and this is where the properties of the sigmoid are helpful. Well, actually, helpfulish. That it is a sigmoid does not really matter too much, it just makes the mathematics look a little pretty, and perhaps more thought out than they actually are.

So, the training works like this: go through each neuron and adjust the input weights so that the result you would have achieved is closer to the actual result you want.

For example, in the case of card number 1738, the set of values for each neuron in the output layer we want is:

–1	–1	–1	–1	–1	–1	–1	–1	+1	–1

That is 'not a 0, not a 1, …, not an 8, *definitely a 9*, not a 0'.

At the moment, the output probably looks something like:

+0.9	+0.8	+0.6	–1.0	+0.5	–0.9	+1.0	–0.7	–0.9	+0.6

If we look at the first output neuron, we see it is giving a +0.9 where a –1.0 is desired. We need to move along the sigmoid curve towards the –1.0 we desire.

The amount to move is achieved by looking at the error: –1.0 – 0.9 = –1.9. We then multiply this up by the gradient on the sigmoid. At this extreme, the gradient is actually very low: (1–0.9)(1+0.9) = 0.19. Finally, we also multiply it by a learning rate, which effectively says, move slowly towards the goal so you do not overshoot. We will use a learning rate of 0.01. This gives the change as –0.0036.

Not very much, but a beginning. We go through and adjust the weights according to this change: we move the weights according to the amount of input we received from the layer below.

At the same time, though, I will also tell the layer below that it contributed to this error (a sort of apportionment blame passing back through the layers). This blame is in proportion to the weight currently in place.

The code for the neuron looks like this:

```
float LEARNING_RATE = 0.01;

class Neuron
{
  // ...
  int m_error;
  // ...
  void respond()
  {
     //...
     // reset the error every time you respond
     m_error = 0.0;
  }
  void setError(int desired)
  {
     m_error = desired - m_output;
  }
  void train()
  {
    float delta =
```

```
      (1.0 - m_output) * (1.0 + m_output) *
      m_error * LEARNING_RATE;
    for (int i = 0; i < m_inputs.length; i++) {
      m_inputs[i].m_error += m_weights[i] * m_error;
      m_weights[i] += m_inputs[i].m_output * delta;
    }
  }
  // ...
}
```

Notice that I have to tell my inputs about their error, and only then adjust my own weights according to the amount to change.

Network (part 3: back propagation)

Now the neuron is capable of feeding back values to its inputs, I simply need to connect together the network so that it can go back-to-front through the layers, updating them:

```
class Network
{
  // ...
  void train(float [] outputs)
  {
    // adjust the output layer
    for (int k = 0; k < m_output_layer.length; k++) {
      m_output_layer[k].setError(outputs[k]);
      m_output_layer[k].train();
    }
    // propagate back to the hidden layer
    for (int j = 0; j < m_hidden_layer.length; j++) {
      m_hidden_layer[j].train();
    }
    // the input layer doesn't learn:
    // it is simply the inputs
  }
}
```

Now the network is ready, all we need to do is to train the network. Then we can test it with random samples from the test set.

Main program

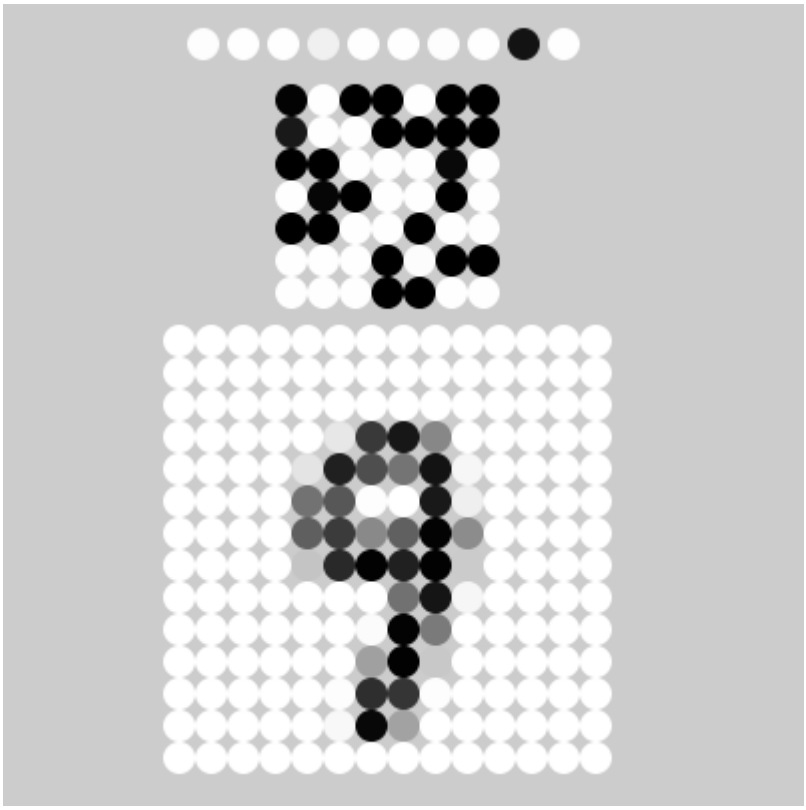
Switching back to the main tab, we can create a piece of code that can train or test according to the mouse button that we select:

```
Network neuralnet;

void setup()
{
  size(400,400);
  loadData();
  setupSigmoid();

  neuralnet = new Network(196,49,10);

  noLoop();
}
```

```
void draw()
{
    background(204);
    noStroke();
    smooth();
    neuralnet.draw();
}

void mousePressed()
{
  if (mouseButton == LEFT) {
    for (int i = 0; i < 500; i++) {
      int row = (int) floor(random(0,training_set.length));
      neuralnet.respond(training_set[row]);
      neuralnet.train(training_set[row].outputs);
    }
  }
  else {
    int row = (int) floor(random(0,testing_set.length));
    neuralnet.respond(testing_set[row]);
  }
  redraw();
}
```

If the left mouse is clicked, then the neural network is trained on 500 random training samples, if the right mouse is clicked, then the net is tested against a single flash card from the test set.

After a while, test card 1738 should give us a better result:



That is, “almost definitely a ‘9’”.