Chapter 2

Neural networks

One approach to learning is to try to simulate the human brain in the computer. To do this, we need to know roughly how a brain works.

A brain neuron (see Figure 2.1) is a cell with two significant pieces: some **dendrites**, which can receive impulses, and an **axon**, which can send impulses. When the dendrites receive enough impulses, the neuron becomes excited and sends an impulse down the axon. In the brain, this axon is next to other neurons' dendrites. These neurons receive the impulse and may themselves become excited, propagating the signal further. This connection between an axon and a dendrite is a **synapse**.

Over time, the connections between axons and dendrites change, and so the brain "learns" as the conditions under which neurons become excited change. How exactly neurons map onto concepts, and how the changing of synapses represents an actual change in knowledge, remains a very difficult question for psychologists and biologists.

2.1 Perceptron

Using what we know about a neuron, though, it's easy to build a mechanism that approximates one. The simplest attempt at this is called the **perceptron**. Although the perceptron isn't too useful as a learning technique on its own, it's worth studying due to its role in artificial neural networks (ANNs), which we'll investigate in Section 2.2.

2.1.1 The perceptron algorithm

The perceptron has a number of inputs corresponding to the axons of other neurons. We'll call these inputs x_i for $i = 1, \ldots, n$ (where n is the number of inputs). It also has a weight w_i for each input (corresponding to the synapses). It becomes excited whenever

$$\sum_{i=1}^{n} w_i x_i > 0 \; .$$

When it is excited, it outputs 1, and at other times it outputs -1. The perceptron can output only these two values.

Let us return to classifying irises. As with linear regression, we'll add a constant-one attribute to give added flexibility to the hypothesis. We'll also make the label numeric by labeling an example 1 *if it is* versicolor *and* -1 *if it is not. Figure 2.2 contains the data we'll use.*

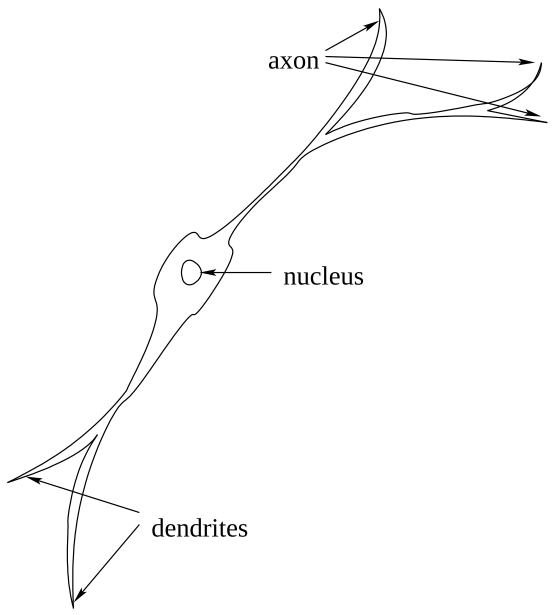


Figure 2.1: A depiction of a human neuron.

	constant one	sepal length	sepal width	petal length	petal width	is species *versicolor*
Iris A	1	4.7	3.2	1.3	0.2	-1
Iris B	1	6.1	2.8	4.7	1.2	1
Iris C	1	5.6	3.0	4.1	1.3	1
Iris D	1	5.8	2.7	5.1	1.9	-1
Iris E	1	6.5	3.2	5.1	2.0	-1

Figure 2.2: A prediction-from-examples problem of Iris classification.

We'll choose random weights with which to begin the perceptron. Say our perceptron begins with the weights $w_1 = 1$, $w_2 = 0$, $w_3 = 0$, $w_4 = 0$, $w_5 = 1$. *For the first prediction, we need to compute whether*

$$\sum_{i=1}^{5} w_i x_i = 1 \cdot 1 + 0 \cdot 4.7 + 0 \cdot 3.2 + 0 \cdot 1.3 + 1 \cdot 0.2 = 1.2$$

exceeds 0. *Since it does, the perceptron predicts that this example's label will be* 1.

To learn, a perceptron must adapt over time by changing its weights w_i over time to more accurately match what happens. A perceptron normally starts with random weights. But each time it makes a mistake by predicting 1 when the correct answer is -1, we change all weights as follows.

$$w_i \leftarrow w_i - r x_i$$

Here r represents the **learning rate**, which should be chosen to be some small number like 0.05. (If you choose r too large, the perceptron may erratically oscillate rather than settle down to something meaningful.)

In our example, we just predicted that Iris A's label would be 1 *when in fact the correct label is* -1 *(according to Figure 2.2). So we'll update the weights.*

$$\begin{aligned}
w_1 &\leftarrow w_1 - r x_1 = 1 - 0.05 \cdot 1 &&= 0.95 \\
w_2 &\leftarrow w_2 - r x_2 = 0 - 0.05 \cdot 4.7 &&= -0.24 \\
w_3 &\leftarrow w_3 - r x_3 = 0 - 0.05 \cdot 3.2 &&= -0.16 \\
w_4 &\leftarrow w_4 - r x_4 = 0 - 0.05 \cdot 1.3 &&= -0.07 \\
w_5 &\leftarrow w_5 - r x_5 = 1 - 0.05 \cdot 0.2 &&= 0.99
\end{aligned}$$

These are the weights we'll use for the next training example.

Similarly, if the perceptron predicts -1 when the answer is 1, we change the weights again.

$$w_i \leftarrow w_i + r x_i$$

To compute the prediction for Iris B, we determine whether

$$\sum_{i=1}^{5} w_i x_i = 0.95 \cdot 1 + (-0.24) \cdot 6.1 + (-0.16) \cdot 2.8 + (-0.07) \cdot 4.7 + 0.99 \cdot 1.2 = -0.05$$

exceeds 0. *Since it does not, the perceptron predicts that Iris B is labeled* -1.

This is wrong: According to Figure 2.2, the correct label for Iris B is 1. *So we'll update the weights.*

$$\begin{aligned}
w_1 &\leftarrow w_1 + r x_1 = 0.95 + 0.05 \cdot 1 &&= 1.00 \\
w_2 &\leftarrow w_2 + r x_2 = -0.24 + 0.05 \cdot 6.1 &&= 0.07 \\
w_3 &\leftarrow w_3 + r x_3 = -0.16 + 0.05 \cdot 2.8 &&= -0.02 \\
w_4 &\leftarrow w_4 + r x_4 = -0.07 + 0.05 \cdot 4.7 &&= 0.17 \\
w_5 &\leftarrow w_5 + r x_5 = 0.99 + 0.05 \cdot 1.2 &&= 1.05
\end{aligned}$$

These are the weights we'll use for the next training example.

Let me make an intuitive argument for why this training rule makes sense. Say we get an example where the perceptron predicts 1 when the answer is -1. In this case, each input contributed $w_i x_i$ to a total that ended up being positive (and so the perceptron got improperly excited). After the training, the new weight is $w_i - r x_i$, and so if the perceptron saw the same example again, this input would contribute $(w_i - r x_i) x_i = w_i x_i - r x_i^2$. Since r is positive, and since x_i^2 must be positive regardless of x_i's value, this contribution is smaller than before. Thus if we repeat the same example immediately, the sum will be smaller than before — and hence closer to being labeled -1 correctly. (The same line of argument applies when the perceptron predicts -1 when the answer is 1.)

The perceptron algorithm will iterate over and over through the training examples until either it predicts all training examples correctly or until somebody decides it's time to stop.

To train the perceptron on the irises, we'd go through the examples several times. The following table demonstrates how the weights change while going through all the examples three times over.

iris	$\sum_i w_i x_i$	w_1	w_2	w_3	w_4	w_5
A	1.20	0.95	-0.24	-0.16	-0.07	0.99
B	-0.05	1.00	0.07	-0.02	0.17	1.05
C	3.39	1.00	0.07	-0.02	0.17	1.05
D	4.21	0.95	-0.22	-0.16	-0.08	0.96
E	0.50	0.90	-0.55	-0.32	-0.34	0.86
A	-2.94	0.90	-0.55	-0.32	-0.34	0.86
B	-3.88	0.95	-0.24	-0.18	-0.10	0.92
C	-0.16	1.00	0.04	-0.03	0.10	0.98
D	3.54	0.95	-0.25	-0.16	-0.15	0.89
E	-0.21	0.95	-0.25	-0.16	-0.15	0.89
A	-0.76	0.95	-0.25	-0.16	-0.15	0.89
B	-0.69	1.00	0.05	-0.02	0.08	0.95
C	2.80	1.00	0.05	-0.02	0.08	0.95
D	3.47	0.95	-0.24	-0.16	-0.17	0.85
E	-0.27	0.95	-0.24	-0.16	-0.17	0.85

At first glance, this looks quite nice. On the first pass through the examples, the perceptron classified only 1 of the 5 flowers correctly. On the second pass, it got 2 correct. And on the third pass, it got 3 correct.

If you continue to see how it improves, though, the perceptron doesn't label 4 of the flowers correct until 34 times through the training set. It finally gets all 5 flowers right on the $1,358$th iteration through the examples. (The training rate doesn't affect this too much: Increasing the training rate even quite a bit doesn't speed it up much, and decreasing the rate only slows it a little.)

Incidentally, after going through all these iterations using $r = 0.05$, the final weights would be $w_1 = -1.80$, $w_2 = -0.30$, $w_3 = -0.19$, $w_4 = 4.65$, and $w_5 = -11.56$.

2.1.2 Analysis

Although the approaches are very different, it's instructive to compare linear regression with the perceptron. Their prediction techniques are identical: They have a set of weights, and they predict according to whether the weighted sum of the attributes exceeds a threshold.

We know that linear regression has a strong mathematical foundation, and we know that the perceptron hypothesis isn't any more powerful than that used by linear regression. So why would you ever use a perceptron instead?

You wouldn't. I don't know of any reason to use a single-perceptron predictor, when you could just as easily do linear regression. Perceptrons are easier to program, sure, and easier to understand. But they take a lot more computation to get the same answer, if you're lucky enough to get an answer. (Perceptrons aren't even guaranteed to converge to a single answer, unless there's a hyperplane that separates the data perfectly.)

So what's the point of studying perceptrons? They're inspired by human biology, and the human brain is the best learning device known to humankind. But we need to keep in mind that, though the human brain

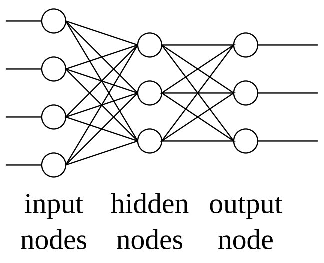


Figure 2.3: An artificial neural network.

is extraordinarily powerful, each individual neuron is relatively worthless. To get good performance, we're going to have to network neurons together. That's what we're going to look at next, and that's when we'll start to see some dividends from our study of perceptrons.
Advantages:

- Simple to understand.
- Generates a simple, meaningful hypothesis.
- Inspired by human biology.
- Adapts to new data simply.

Disadvantages:

- The hypothesis is a simple linear combination of inputs, just like linear regression, but linear regression has much more solid mathematical grounds.
- Only works for predicting yes/no values.

2.2 Artificial neural networks

Artificial neural networks (ANNs) are relatively complex learning devices. We'll look first at the overall architecture, then at the individual neurons, and finally at how the network predicts and adapts for training examples.

2.2.1 ANN architecture

Figure 2.3 pictures the layout of one ANN. This particular network has three layers. The first is the **input layer**, including four nodes in Figure 2.3. These nodes aren't really neurons of the network — they just feed the attributes of an example to the neurons of the next layer. (We'll have four inputs when we look at the irises, as each iris has four attributes.)

The next layer is the **hidden layer**. This layer has several neurons that should adapt to the input. These neurons are meant to process the inputs into something more useful, like detecting particular features of a picture if the inputs represent the pixels of a picture — but they'll automatically adapt, so what they detect isn't necessarily meaningful. Choosing the right number of neurons for this layer is an art. Figure 2.3 includes three hidden neurons in the hidden layer, with every input node connected to every neuron, but really an ANN could have any number of hidden neurons in any configuration.

The final layer is the **output layer**. It has an output neuron for each output that the ANN should produce.

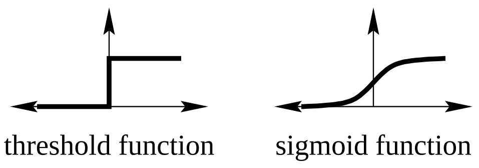
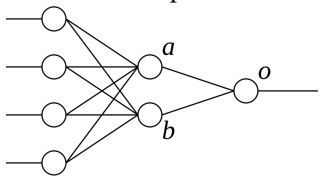


Figure 2.4: The threshold function and the sigmoid function.

One could conceivably have more hidden layers, or delete or add links different from those diagrammed in Figure 2.3, or even use a more complex unlayered design. But the most common design is the **two-layer network**, like that of Figure 2.3. In this architecture, there is a layer of some number of hidden neurons, followed by a layer of some number of output neurons. It has each input connected to each hidden neuron, and it has each hidden neuron connected to each output neuron. People usually use two-layer networks because more complex designs just haven't proven useful in many situations.

In our walk-through example of how the ANN works, we'll use the following very simple two-layer network with two hidden units a and b and one output unit o.



As is commonly done with ANNs, each of the neurons in our network has an additional constant-one input. We do this instead of adding a new constant-1 attribute (as we did with linear regression and perceptrons), because that wouldn't give a constant-one input into the output unit o.

2.2.2 The sigmoid unit

Each unit of the network is to resemble a neuron. You might hope to use perceptrons, but in fact that doesn't work very well. The problem is one of feedback: In a neural network, we must sometimes attribute errors in the ANN prediction to mistakes by the hidden neurons, if the hidden neurons are to change behavior over time at all. (If they don't change behavior, there isn't much point in having them.) Researchers just haven't found a good way of doing this with regular perceptrons.

But researchers *have* figured out a practical way to do this attribution of error using a slight modification of the perceptron, called a **sigmoid unit**. Sigmoid units produce an output between 0 and 1, using a slightly different procedure from before. (Having the low output being 0 is just a minor difference from the perceptron, whose low output is -1. We'll just rework our training example labels by replacing -1 labels with 0.)

A sigmoid unit still has a weight w_i for each input x_i, but it processes the weighted sum slightly differently, using the **sigmoid function**, defined as

$$\sigma(y) = \frac{1}{1 + e^{-y}} \; .$$

This is a smoothed approximation to the threshold function used by perceptrons, as Figure 2.4 illustrates.

To make a prediction given the inputs x_i, the sigmoid unit computes and outputs the value

$$\sigma\left(\sum_i w_i x_i\right) \; .$$

Notice that this means it will output some number *between* 0 and 1. It will never actually be 0 or 1, but it can get very close.

The difference between a sigmoid unit and a perceptron is quite small. The only real reason to use the sigmoid unit is so that the mathematics behind the analysis — which we won't examine — works out to show that the neural network will approach some solution. The advantage of a sigmoid unit is that its behavior is smooth and never flat. This makes mathematical analysis easier, since it means we can always improve the situation by climbing one way or the other along the sigmoid curve. If the output is too high, we'll try to go downhill a bit. Too small? Go uphill a bit. But we won't get into the details of why it works.

We have three sigmoid units in our architecture: a, b, and o. The hidden units a and b have five inputs and hence five weights each — one from each input node, plus the constant-one input. The output unit o has three inputs and hence three weights — one from each hidden unit, plus the constant-one input. To begin the network, we initialize each of these weights using small random values as follows.

$w_{0a} = 0.0$	*weight of constant-one input into hidden unit a*
$w_{1a} = 0.1$	*weight for hidden unit a from first input node*
$w_{2a} = -0.1$	*weight for hidden unit a from second input node*
$w_{3a} = -0.1$	*weight for hidden unit a from third input node*
$w_{4a} = 0.0$	*weight for hidden unit a from fourth input node*
$w_{0b} = 0.0$	*weight of constant-one input into hidden unit b*
$w_{1b} = -0.1$	*weight for hidden unit b from first input node*
$w_{2b} = 0.2$	*weight for hidden unit b from second input node*
$w_{3b} = 0.1$	*weight for hidden unit b from third input node*
$w_{4b} = -0.1$	*weight for hidden unit b from fourth input node*
$w_{0o} = 0.1$	*weight of constant-one input into output unit o*
$w_{ao} = 0.2$	*weight for output unit o from hidden unit a*
$w_{bo} = -0.1$	*weight for output unit o from hidden unit b*

2.2.3 Prediction and training

Handling a single training example is a three-step process. We'll see how to do each of these steps in detail soon, but here's the overview.

1. We run the training example through the network to see how it behaves (the *prediction step*).

2. We assign an "error" to each sigmoid unit in the network (the *error attribution step*).

3. We update all the weights of the network (the *weight update step*).

The whole process is akin to the perceptron training process, except here we'll *always* update the weights. (Recall that the perceptron weights got updated only the perceptron erred.)

Prediction step In a two-layer network like that of Figure 2.3, the prediction step is straightforward: We take the example attributes, feed them into the hidden sigmoid units to get those units' output values, and then we feed these hidden units' outputs to the output units. The output units' outputs are the ANN's prediction.

Given Iris A, we first compute the output of hidden node a.

$$\begin{aligned} o_a &= \sigma(w_{0a} + w_{1a}x_1 + w_{2a}x_2 + w_{3a}x_3 + w_{4a}x_4) \\ &= \sigma(0.0 + 0.1 \cdot 4.7 + (-0.1) \cdot 3.2 + (-0.1) \cdot 1.3 + 0.0 \cdot 0.2) \\ &= \sigma(0.02) = \frac{1}{1 + e^{-0.02}} = 0.5050 \end{aligned}$$

Similarly, we compute the output of hidden node b.

$$\begin{aligned} o_b &= \sigma(w_{0b} + w_{1b}x_1 + w_{2b}x_2 + w_{3b}x_3 + w_{4b}x_4) \\ &= \sigma(0.0 + (-0.1) \cdot 4.7 + 0.2 \cdot 3.2 + 0.1 \cdot 1.3 + (-0.1) \cdot 0.2) \\ &= \sigma(0.28) = \frac{1}{1 + e^{-0.28}} = 0.5695 \end{aligned}$$

Finally, we can compute the output of the output node o.

$$\begin{aligned} o_o &= \sigma(o_{0o} + w_{ao}o_a + w_{bo}o_b) = \sigma(0.1 + 0.2 \cdot 0.5050 + (-0.1) \cdot 0.5695) \\ &= \sigma(0.1440) = \frac{1}{1 + e^{-0.1440}} = 0.5359 \end{aligned}$$

Thus the computed prediction for Iris A is 0.5359.

Error attribution step The error attribution step uses a technique called **backpropagation**. What we'll do is to look at the entire network's output (made by the output layer) and determine the error of each output unit. Then we'll move backward and attribute errors to the units of the hidden layer.

To assign the error of an output unit, say the desired output for the unit is t_o, but the actual output made by the unit was o_o. We compute the error of that output unit as $o_o(1 - o_o)(t_o - o_o)$.

After we compute the errors of all outputs units, we backpropagate to the hidden layer. Consider a hidden unit h, whose weight connecting it to an output unit o is w_{ho}. Moreover, call h's output o_h. (This is the output that was sent forward to the output node when making a prediction.) The error that we'll attribute to the hidden node h from o is $o_h(1 - o_h)w_{ho}\delta_o$. (If there are multiple output units, we'll sum these errors over all outputs to get the total error attributed to h.)

In our example, the correct answer t_o was 0 (Iris A is not versicolor*), while the network output o_o was 0.5359. Thus the error of unit o (which we'll represent by δ_o) is*

$$\delta_o = o_o(1 - o_o)(t_o - o_o) = 0.5359(1 - 0.5359)(0 - .5359) = -0.1333 \,.$$

Now that we have errors attributed to the output layer, we can compute the error of the hidden units. We compute δ_a, the error attributed to hidden unit a.

$$\delta_a = o_a(1 - o_a)w_{ao}\delta_o = 0.5050(1 - 0.5050)0.2 \cdot (-0.1333) = -0.0067$$

And we compute the error δ_b of the hidden unit b.

$$\delta_b = o_b(1 - o_b)w_{bo}\delta_o = 0.5359(1 - 0.5359)(-0.1) \cdot (-0.1333) = 0.0033$$

Weight update step The last step of handling the training example is updating the weights. We update *all* weights as follows (both those going from inputs to hidden nodes, and those going from hidden nodes to output nodes).

Consider one input to a sigmoid unit in the ANN. Say x is the value being fed into the input during the prediction step, and δ is the error attributed during the error-attribution step to the sigmoid unit receiving the input. We'll add $r\delta x$ to the weight associated with this input.

We'll use a learning rate r of 0.1 here. Here's how the weights are changed for this training example.

$$\begin{array}{lclcl}
w_{0a} \leftarrow w_{0a} + r\delta_a 1 &=& 0.0 + 0.1 \cdot (-0.0067) \cdot 1 &=& -0.0007 \\
w_{1a} \leftarrow w_{1a} + r\delta_a x_1 &=& 0.1 + 0.1 \cdot (-0.0067) \cdot 4.7 &=& 0.0969 \\
w_{2a} \leftarrow w_{2a} + r\delta_a x_2 &=& -0.1 + 0.1 \cdot (-0.0067) \cdot 3.2 &=& -0.1021 \\
w_{3a} \leftarrow w_{3a} + r\delta_a x_3 &=& -0.1 + 0.1 \cdot (-0.0067) \cdot 1.2 &=& -0.1009 \\
w_{4a} \leftarrow w_{4a} + r\delta_a x_4 &=& 0.0 + 0.1 \cdot (-0.0067) \cdot 0.2 &=& -0.0001 \\
w_{0b} \leftarrow w_{0b} + r\delta_b 1 &=& 0.0 + 0.1 \cdot 0.0032 \cdot 1 &=& 0.0003 \\
w_{1b} \leftarrow w_{1b} + r\delta_b x_1 &=& -0.1 + 0.1 \cdot 0.0032 \cdot 4.7 &=& -0.0985 \\
w_{2b} \leftarrow w_{2b} + r\delta_b x_2 &=& 0.2 + 0.1 \cdot 0.0032 \cdot 3.2 &=& 0.2010 \\
w_{3b} \leftarrow w_{3b} + r\delta_b x_3 &=& 0.1 + 0.1 \cdot 0.0032 \cdot 1.2 &=& 0.1004 \\
w_{4b} \leftarrow w_{4b} + r\delta_b x_4 &=& -0.1 + 0.1 \cdot 0.0032 \cdot 0.2 &=& -0.0999 \\
w_{0o} \leftarrow w_{0o} + r\delta_o 1 &=& 0.1 + 0.1 \cdot (-0.1333) \cdot 1 &=& 0.0867 \\
w_{ao} \leftarrow w_{ao} + r\delta_o o_a &=& 0.2 + 0.1 \cdot (-0.1333) \cdot 0.5050 &=& 0.1933 \\
w_{bo} \leftarrow w_{bo} + r\delta_o o_b &=& -0.1 + 0.1 \cdot (-0.1333) \cdot 0.5695 &=& -0.1076
\end{array}$$

These are the weights we'll use for our next training example.

Conclusion This is *all* just for a single training example. Like in training a perceptron, we'd do all of this for each of the examples in the training set. And we'd repeat it several times over. It's not the sort of thing you can do by hand, though a computer can do it pretty easily for small networks.

Just to demonstrate that we've made progress, let's see what the network would predict if we tried Iris A again. We propagate its attributes through the network.

$$\begin{array}{rcl}
o_a &=& \sigma(-0.0007 + 0.0969 \cdot 4.7 + (-0.1021) \cdot 3.2 + (-0.1009) \cdot 1.3 + (-0.0001) \cdot 0.2) \\
&=& \sigma(-0.0032) = 0.4992 \\
o_b &=& \sigma(0.0003 + (-0.0985) \cdot 4.7 + 0.2010 \cdot 3.2 + 0.1004 \cdot 1.3 + (-0.0999) \cdot 0.2) \\
&=& \sigma(0.2911) = 0.5724 \\
o_o &=& \sigma(0.0867 + 0.1933 \cdot 0.4992 + (-0.1076) \cdot 0.5724) \\
&=& \sigma(0.1440) = 0.5304
\end{array}$$

Thus the computed prediction for Iris A is 0.5304, closer to the correct answer of 0 than the previously predicted 0.5359. This provides some evidence that the ANN has learned something *through this training process.*

2.2.4 Example

Computationally, backpropagated ANNs are so complex that it's difficult to get a strong handle on how they work. It's instructive to look at a more industrial-strength example to see how they might actually be used. Let's consider the full iris classification example of Fisher [Fis36].

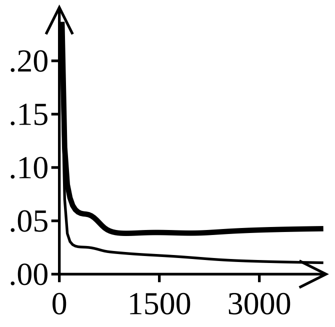


Figure 2.5: ANN error against iterations through training set. The thick line is the average error per evaluation example, and the thin line is the average error per training example.

Recall that this set of examples includes 150 irises, with each iris having four numeric attributes and a label classifying it among one of three species. Of the 150 irises, we choose a random subset of 100 as training examples and the remaining 50 for evaluation purposes.

We'll model this as a two-layer ANN. For each of the four numeric attributes, we'll include an input node in the input layer. In the hidden layer, we'll choose to use two sigmoid units. And the output layer will have a unit for each of the three possible labels. When we interpret the outputs, we'll find the output unit emitting the greatest output and interpret the ANN to be predicting the corresponding label.

We use a learning rate of 0.1. For training purposes, a label is encoded as $\langle 0.9, 0.1, 0.1\rangle$ for *setosa*, $\langle 0.1, 0.9, 0.1\rangle$ for *versicolor*, or $\langle 0.1, 0.1, 0.9\rangle$ for *virginica*. We use 0.1 and 0.9 instead of 0 and 1 because a sigmoid unit never actually reaches an output of 0 or 1; if we encode labels using 0s and 1s, the weights will gradually become more and more extreme.

When we train, we go through the 100 training examples several times. It's interesting to see how the ANN improves as this continues. We'll look at the average sum-of-squares error: If the ANN outputs $\langle o_1, o_2, o_3\rangle$ and the desired output is $\langle t_1, t_2, t_3\rangle$, the sum-of-squares error is

$$\sum_{i=1}^{3}(t_i - o_i)^2 \,.$$

This is an odd quantity to investigate. We choose it because the mathematics behind the error attribution and weight update is motivated by trying to decrease the sum-of-squares error for any given training example. (And the researchers who did the derivation chose it basically because they found a technique for decreasing it. That circularity — they discovered that the mathematics worked if they just chose to try to minimize this peculiar error function — is the spark of genius that makes backpropagation work.)

Figure 2.5 graphs the sum-of-squares error on the y-axis and the number of iterations through the data set on the x-axis. The thick line is the important number — it's the average sum-of-squares error over the 50 evaluation examples. The thin line is the average sum-of-squares error over the 100 training examples.

Since the mathematics behind the backpropagation update rule works with a goal of minimizing the sum-of-squares error, we expect that the thin line in this graph is constantly decreasing. But, then, that improvement is on the training examples: It's not indicative of general performance. The thick line represents performance on the evaluation examples the ANN never trains on, which *is* indicative of general performance. You can see that it flattens out after 1,000 iterations through the examples and then takes a upward turn after another 1,000 iterations, despite continued improvement on the training examples.

What's happening beyond the 2,000th iteration is *overfitting*. The neural net is adapting to specific anomalies in the training data, not the general picture. If we were running the data, we'd want to stop around this point rather than continue onward.

Of course, one could argue that what we *really* care about is the number of evaluation examples classified correctly, not the average sum-of-squares error. But this supports the same conclusion we reached using the sum-of-squares error.

- By the 100th iteration through the examples, the ANN got 47 of the 50 correct.
- By the 1,000th, it was getting 49 of 50.
- Around the 2,000th iteration, the ANN was back to 48 of 50 and remains there.

So it was probably best to stop after the first 1,000 iterations. (This is just for a single run-through over the data. Different runs give slightly different numbers, due to the small random numbers chosen for the initial weights in the network.)

It's also worthwhile looking at how the behavior changes with different numbers of hidden units. Adding additional hidden units in this case keeps the picture more or less the same — maybe even slightly worse than 2 units. Only one hidden unit isn't powerful enough — the network never gets above 40 correct.

Typically, people find that there's some critical number of hidden units. Below this, the ANN performs poorly. At the critical number, it does well. And additional hidden units provide only marginal help if it helps at all, at the added expense of much more computation.

2.2.5 Analysis

ANNs have proven to be a useful, if complicated, way of learning. They adapt to strange concepts relatively well in many situations. They are one of the more important results coming out of machine learning.

One of the complexities with using ANNs is the number of parameters you can tweak to work with the data better. You choose the representation of attribute vectors and labels, the architecture of the network, the training rate, and how many iterations through the examples you want to do. The process is much more complicated than simply feeding the data into a linear regression program. The extended example presented in this chapter is intended to illustrate a realistic example, where we had to make a variety of decisions in order to get a good neural network for predicting iris classification.

Advantages:

- Very flexible in the types of hypotheses it can represent.
- Bears some resemblance to a very small human brain.
- Can adapt to new data with labels.

Disadvantages:

- Very difficult to interpret the hypothesis as a simple rule.
- Difficult to compute.