Introduction to Neural Networks

Regression, least squares, gradient descent, Widrow-Hoff & backprop

Initialization

```
In[338]:= Off[SetDelayed::write]
          Off[General::spell1]
```

Introduction

Last time

Focused on classification:

Turning linear networks into classifiers with a hard threshold.

Perceptrons, perceptron learning rule.

Linear separability

Today

Linear regression & brain-style supervised learning -- the Widrow-Hoff learning rule

Multiple layers of weights -- backdrop learning, an example of non-linear regression

Linear regression and Widrow-Hoff learning

Introduction

We have been studying a linear matrix model of memory based on the storage of connection weights that follow a particular Hebbian rule. We have studied the "psychology" of some operations of linear algebra and have seen some interesting parallels to human memory, such as interference and pattern reconstruction.

We've learned that linear networks can be configured to be supervised or unsupervised learning devices. But linear mappings are severely limited in what they can compute. With a threshold non-linearity, the linear neuron model becomes a simple perceptron unit that makes decisions.

But we are now limited by the linearity of the decision surface, i.e. by the fact that it is a hyperplane. More complex networks can be built by adding layers making multiple layer networks (“multi-layer perceptrons”), but then we have another problem: how can we learn the weights in this more complicated setting?

Overview of our strategy

First we return to strictly linear networks (no threshold) which will allow us to introduce techniques (gradient descent learning in the context of finding the weights of a linear matrix transformation). We will learn concepts that will generalize to non-linear networks with multiple layers of weights. In particular, the technique of gradient descent will lead us to the Widrow-Hoff learning rule. By treating the linear case first, we will be able to see how the Widrow-Hoff learning rule relates to the classic problem of linear regression.

Learning, memory and generalization can be viewed from the point of view of non-linear statistical regression. The idea is to treat learning as an attempt to fit past input/output associations into a model that can be used both for recall and for future generalization.

The problem of regression in statistics is: given a set of vector inputs $\{\mathbf{x}_i\}$, and a set of corresponding vector “target” outputs $\{\mathbf{t}_i\}$, one tries to find a transformation **W** that will map $\mathbf{x} \to \mathbf{t}$ as closely as possible over the data available. For this we need a model for **W**, and a measure of *goodness of fit.* At first, we will assume a linear model for **W**, so for the discrete case, **W** is a matrix. Our measure of goodness of fit will be the *sum of the squared differences between predicted output and actual output.* In the linear case, this is linear *least squares regression*.

Then we’ll see how the Widrow-Hoff rule provides a clue how to generalize learning to non-linear, multiple layer networks with smooth non-linear squashing functions, with the same measure of goodness of fit. This is the error back-propagation algorithm.

In later lectures, we will develop a more powerful notion of goodness of fit -- *probability* of parameters given experience with past inputs and outputs, for which the sum of squared differences is a special case.

More terminology regarding supervised learning

Consider s**upervised learning**. We have a "training set" $\{\mathbf{x}_i, \mathbf{t}_i\}$ representing input vectors $\mathbf{x}_i$, and target outputs $\mathbf{t}_i$. The training set "samples" the larger space of possible input/output pairs $\{\mathbf{x}, \mathbf{t}\}$.   We would like to learn a general mapping: T: **x** -> **y** in such a way that T (= **W**) is a good fit to the training data (i.e. **y** is close to **t**), and generalizes well to novel inputs. The set of target data is the feedback for the "teacher".

In general, feedback can vary in the degree of precision it provides for learning. It can specify whether the mapping is correct or not. Or the feedback can provide information as to how far off the map T's prediction, i.e. how far is $T[\mathbf{x}_i]$) from $\mathbf{t}_i$? It is the latter case we pursue below.

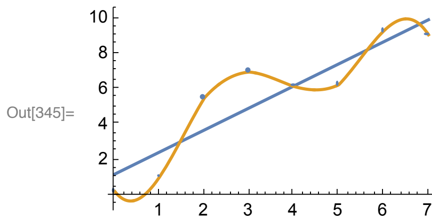
After training, one could require that T always maps members of the training set to exactly the target members, and hopefully generalizes appropriately for other inputs. This means that the learning should be *consistent*. ***Interpolation*** between data points on a graph is an example of consistent learning. A specific example of interpolation would be drawing lines connecting data points on a graph as illustrated below.

Or, we may require that the T maps the original members of the training set to outputs **y**, that are close,

but not necessarily equal to the original targets **t**. This is called ***approximation*** learning.

In[340]:=
```
Clear[data, x];
data = Table[{x, x + 4 * RandomReal[]}, {x, 0, 7}];
gdata = ListPlot[data];
approx = Fit[data, {1, x}, x];
interp = Interpolation[data];
Show[{Plot[{approx, interp[x]}, {x, 0, 7}], gdata}, ImageSize → Small]
```

Out[345]=



Linear regression is an example of *approximation* learning. Approximation doesn't necessarily exactly fit the data points, but the goal is that it should come close, but not at the expense of failing to generalize to new data--a problem called over-fitting. Note that in the above example, the approximation solution has only two parameters, but the interpolation requires specifying all the data.

► 1. Try Fit[data,{1,x,x^2,x^3,x^4,x^5},x]. Note how having more parameters gets one closer to an interpolation solution.

In addition to remembering, regression *generalizes,* i.e. provides the basis for *prediction.* So novel input values get mapped to predicted outputs based on past "experience". We've already seen how linear heteroassociative learning does this. Later we will look at how the type of generalization affects "bias" and "variance".

Let's take a close look at linear regression, a fundamental case of approximation learning.

Least squares regression - linear models

The idea behind least squares regression is given a set of N training pairs $\{\mathbf{x}_i, \mathbf{t}_i\}$, where i runs from 1 to N, we would like to find a function that given an input **x**, the function approximates well the output, i.e. **W.x ~ t**. If the function reproduces the association between input and output that it has seen before, this is like "remembering".

An example of linear regression

A fundamental assumption that affects the modeling and performance of any learning system which needs to generalize is that there is an underlying structure to the data--the relationship between associative pairs is not arbitrary.

When trying to understand how a relationship can be learned between a set of two patterns, it helps to have some understanding of the structure of the relationship. For example, one shouldn't try to fit a straight line to data when there is reason to believe that the underlying process produces data on a circle. We will study this more when we learn about the "bias/variance dilemma". So let us assume that the data have an underlying structure that we are going to try to discover or approximate using **W**.

We will study a simple "toy" problem that has the following very specific generative structure. The inputs are randomly located points on a 2D plane, and the outputs are heights above these points. The outputs lie approximately on a planar surface that runs through the origin and whose orientation is characterized by two parameters, **a** and **b**.

It may seem like overkill, but we are going to estimate **W** for the same set of data in *four* different ways to illustrate several points.

The first two methods are drawn from standard linear algebra (first, a least squares solution using transpose and inverse, and the second using the pseudoinverse). The third introduces the method of "gradient descent" a common numerical technique which is used in many applications, including non-linear regression. We will use it later in several contexts. The fourth method is the most relevant to neural network theory--we estimate **W** using a biologically plausible learning rule (the Widrow-Hoff rule).

Generative model: Synthetic training pairs

Although we will illustrate our examples with few dimensions, everything we do generalizes to higher dimensional inputs and outputs, and in fact the demonstration code below will work with higher dimensional input/output vectors.

Let x1 and x2 be the (scalar) inputs, and y be the (scalar) output:

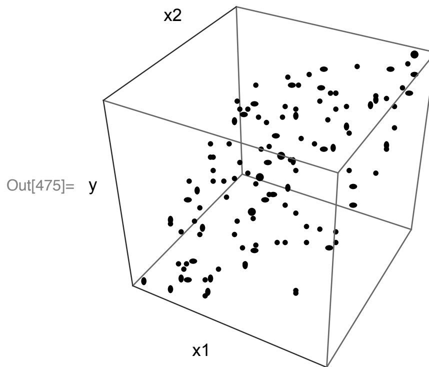
```
{x1, x2} → y,
and assuming the mapping is a plane through the origin and additive noise,
y = a x1 + b x2 + noise
```

Even without noise, this is a many-to-one mapping--many combinations of x1 and x2 can give the same value of y. This equation describes a plane oriented in 3D, and a fixed value of y describes a horizontal plane. Solutions are were the two planes intersect, which is a line.

We want to learn the parameters a and b from the training pairs: {{x1, x2}, y}. These parameters could be thought of as neural weights.

In[473]:=
```
rsurface[a_, b_] := N[Table[{x1 = 1 RandomReal[],
     x2 = 1 RandomReal[], a x1 + b x2 + 0.5 RandomReal[] - 0.25}, {120}], 2];
data = rsurface[2, 3];
Graphics3D[Point /@ data, AxesLabel → {"x1", "x2", "y"},
  Axes → True, Ticks → None, BoxRatios → {1, 1, 1}, ImageSize → Small]
Dimensions[data]
```

Out[475]=



Out[476]= {120, 3}

You can also use: **ListPointPlot3D[data]**

Let’s extract the y and x data, where **yy** represents the 1 dimensional *y* values, and **xx** the 2-dimensional input values *x* = {x1,x2}:

In[479]:=
```
yy = data[[All, 3]];
xx = data[[All, 1 ;; 2]];
```

1. Least squares regression to find W

Assume we want to find a matrix **W** that will come close to reproducing values **y**, given inputs **x**. Of course, because we generated the data, we know the underlying structure and what the matrix **W** should be. The correct answer should be a 1x2 matrix = {{a,b}} = **{{2,3}}**. But let's assume we don't know the answer, and want to discover the weights from **yy** and **xx**. The **yy** data contains the target values (**t** above).

Imagine you have knobs that you can twiddle that let you adjust each element of the matrix **W**. For particular settings of these knobs, you can calculate $\mathbf{W}.\mathbf{x}_i$ for any input/output pair. Think of $\mathbf{W}.\mathbf{x}_i$ as the current guess of what $\mathbf{y}_i$ is. In general you won't be so lucky as to have **W** predict your given target values, $\mathbf{y}_i=\mathbf{W}.\mathbf{x}_i$. So you take the opportunity to adjust the knobs to produce $\mathbf{W}'$ so that $\mathbf{W}'.\mathbf{x}_i$ is closer to $\mathbf{y}_i$. A familiar measure of "how close" is:

$$(\mathbf{y}_i - \mathbf{W}.\mathbf{x}_i).(\mathbf{y}_i - \mathbf{W}.\mathbf{x}_i) =$$

$$|\mathbf{y}_i - \mathbf{W}.\mathbf{x}_i|^2 = (\text{Norm}[\mathbf{y}_i - \mathbf{W}.\mathbf{x}_i])^2 = \text{EuclideanDistance}[\mathbf{y}_i, \mathbf{W}.\mathbf{x}_i]\text{^}2.$$

Of course, we usually have lots of data, so we could add up all the discrepancies between what **W** predicts, and what the "teacher" has provided in the answers in the training pair (i.e. y_i for all of the i's). In least squares regression, we try to find the values of the matrix that will minimize the sum of all the errors, e(**W**).

$$e(\mathbf{W}) = \sum_{i=1}^{N} |\mathbf{y}_i - \mathbf{W}.\mathbf{x}_i|^2$$

In many cases our error function will be over a very high dimensional space determined by the number of elements in **W**. However, it is useful to get an intuition for the problem in a low dimensional space. We can get a picture of the total error, **e(W)**, for our synthetic data as a function of the weight parameters **w1** and **w2**:

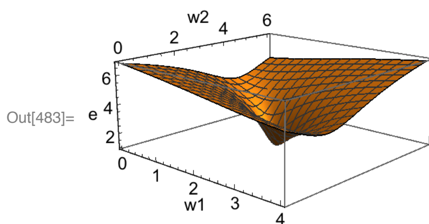
```
In[481]:= eW[w1_, w2_] := Sum[{yy[[i]] - {w1, w2}.xx[[i]]}.{yy[[i]] - {w1, w2}.xx[[i]]},
          {i, 1, Length[xx], 1}]
```

```
In[482]:= g = Plot3D[eW[w1, w2], {w1, 0, 4}, {w2, 0, 6}, ViewPoint → {1.78, -2.861, 0.312},
    AxesLabel → {"w1", "w2", "e"}, ImageSize → Small];
```

If you plot the log of the error function, you can get a better view of where the minimum is:

```
In[483]:= g = Plot3D[Log[eW[w1, w2]], {w1, 0, 4}, {w2, 0, 6},
    ViewPoint → {1.78, -2.861, 0.312}, AxesLabel → {"w1", "w2", "e"}, ImageSize → Small]
```

Out[483]=



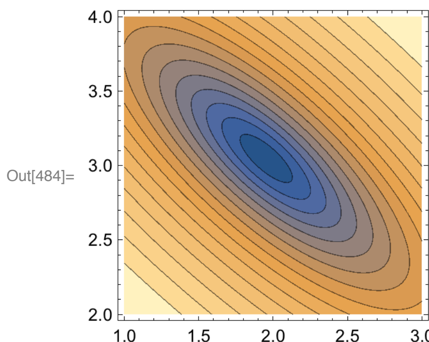
(Remember that log[] is a montonic function, so it won’t change the location of the minimum.)

Note that because our input data are 2-element vectors, and our output or target values are 1-D, the matrix **W** we seek is not square--it has 1 row and 2 columns. So we represent **W** above as a vector.

The minimum appears to be near **{2,3}**. It is easier to see the location of the minimum using **ContourPlot[]**.

```
In[484]:= gc = ContourPlot[Log[eW[w1, w2]],
    {w1, 1, 3}, {w2, 2, 4}, Contours → 16, ImageSize → Small]
```

Out[484]=



Note: You can get a rough idea of the coordinates near the middle by moving your computer mouse over the above graphic after initializing the cell below:

```
In[485]:= Dynamic[MousePosition["Graphics"]]
Out[485]= None
```

Of course, in high dimensional spaces, we can't visualize **W**, and we can't hope to manually "adjust knobs" or sliders. But we can find the exact location of the minimum by using calculus and linear algebra to find "the bottom of the bowl".

Calculus tells us that this should be where the gradient of the error function, e is zero. This is a generalization from the one-dimensional case in which the slope of the derivative of a graph of a function is zero where the curve is maximum or minimum.

To work out the solution term by term is a bit messy, and the derivation below isn't necessarily obvious without taking the time to work through the details. Although there are a lot of indices to worry about, the formalism actually looks like the familiar case in calculus where one finds the minimum by looking for where the derivative of a function f(x) is zero, i.e. the value of x where f'(x)=0. But now e(**W**) plays the role of f(x), and **W** is a matrix of variables. And instead of a single first derivative, we calculate the gradient, all the values of $\frac{\partial e}{\partial w_{ij}}$, which get stuffed in a matrix we call $\frac{\partial e}{\partial \mathbf{W}}$. Below you can see that the formalism looks similar to the application of the chain rule in calculus. Our eventual goal is to calculate the values of the elements of **W,** (w_{ij}), where all the $\frac{\partial e}{\partial w_{ij}}$=0, which we express in terms of $\mathbf{y}_i$ and $\mathbf{x}_i$. When we calculate the gradient, we treat $\mathbf{y}_i$ and $\mathbf{x}_i$ as fixed. The solution can be written concisely in terms of vector outer products, inversion, and matrix multiplication:

$$e(\mathbf{W}) = \sum_{i=1}^{N} \left|\mathbf{y}_i - \mathbf{W}\mathbf{x}_i\right|^2$$

$$\frac{\partial e}{\partial \mathbf{W}} = -2\sum_{i=1}^{N} (\mathbf{y}_i - \mathbf{W}\mathbf{x}_i)\mathbf{x}_i^T = 0$$

$$\sum_{i=1}^{N} \mathbf{y}_i \mathbf{x}_i^T - \mathbf{W}\sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T$$

$$W = \sum_{i=1}^{N} \mathbf{y}_i \mathbf{x}_i^T \left(\sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T\right)^{-1}$$

(Note that the first line above is a scalar function. The second line is matrix equation. See wiki entry for matrix calculus. The symbol 0 on the right indicates a matrix filled with 0's. The third and fourth lines are also matrix equations.)

So we see that the value of **W** where the gradient is "zero", (i.e. where all the $\frac{\partial e}{\partial w_{ij}}$=0), can be calculated by first adding up two sets of outer products, taking an inverse of one of them, and then multiplying.

Let's translate the last line above into *Mathematica* instructions:

```
In[486]:= sumX = Sum[Outer[Times,xx[[i]],xx[[i]]],{i,Length[xx]}];

sumYX = Sum[Outer[Times,{yy[[i]]}, xx[[i]]],{i,Length[xx]}];

W = sumYX.Inverse[sumX]

Out[490]= {{1.95365, 3.04769}}
```

The values for **W** come close to what we would expect from the structure of our data, a plane with parameters {a,b} = **{2,3}**.

► 2. Review the chain rule.

Let e, x, y, w all be scalars. Assume y and x are fixed. Let e(w) = (y - w*x)^2. What is $\frac{\partial e}{\partial w}$? Check your answer with: D[(y - w*x)^2,w].

► 3. Derive a solution with a 2x2 weight matrix, and just two training pairs.

Use either D[ ] or Grad[ ] to find the gradient, and Solve[ ] to find an expression for the weights in terms of the training values.

```
In[362]:= Clear[x1, y1, x2, y2, wt11, wt12, wt21, wt22]
Wt = {{wt11, wt12}, {wt21, wt22}};
xt1 = {x11, x21};
yt1 = {y11, y21};
xt2 = {x12, x22};
yt2 = {y12, y2};
```

```
In[367]:= ttt = D[(yt1 - Wt.xt1).(yt1 - Wt.xt1) + (yt2 - Wt.xt2).(yt2 - Wt.xt2),
    {{wt11, wt12, wt21, wt22}}];
```

```
In[368]:= Solve[ttt == {0, 0, 0, 0}, {wt11, wt12, wt21, wt22}];
```

2. Pseudoinverse to find W

For linear algebra afficianodos, there is another way of solving the linear least squares regression that uses the “pseudoinverse” of matrix to find the least-squares solution. For a *square* matrix **X**, the inverse of **X** is chosen so that $\mathbf{XX}^{-1}$ is equal to the identity matrix, **I**. The pseudoinverse, **X***, of a *rectangular* matrix **X** is chosen so that **XX*** is close to the identity matrix in the sense that the sum of the squares of all of the entries of **XX* - I** is least. If interested, you can look up the derivation of **X***. It can be written as:

$$\mathbf{X}^* = \left(\mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T$$

or as:

$$\mathbf{X}^* = \mathbf{X}^T\left(\mathbf{X}\mathbf{X}^T\right)^{-1}$$

For our simple generative model, let's take all of the input vectors **x**, and arrange them as columns in a matrix **X**:

$$\mathbf{X} = \begin{pmatrix} x_1^1 & x_1^2 & & x_1^N \\ x_2^1 & x_2^2 & \cdots & x_2^N \end{pmatrix}$$

Now do the same for the **y**'s:

$$\mathbf{Y} = \begin{pmatrix} y_1 & y_2 & y_3 \ldots y_N \end{pmatrix}$$

And what is the matrix that maps the x's to the y's with least total squared error? Again, you can look up the derivation or try to derive it yourself. The answer is $\mathbf{X}^*\mathbf{Y}$:

```
In[369]:= Inverse[Transpose[xx].xx].Transpose[xx].yy
Out[369]= {1.9876, 3.0013}
```

$\mathbf{X}^*$, the *pseudoinverse* is sometimes called the *generalized inverse* of the matrix **X.**

The **PseudoInverse[]** function is built into *Mathematica,* so now that we know about it, we can calculate the least-squares solution for our problem in one simple expression:

In[491]:=
```
PseudoInverse[xx].yy
```

Out[491]=
```
{1.95365, 3.04769}
```

► 4. What are the dimensions of the above PseudoInverse? Verify that **PseudoInverse[X].X** is the identity matrix with our data.

In[371]:=
```
Chop[PseudoInverse[xx].xx]//MatrixForm
```

Out[371]//MatrixForm=

$$\begin{pmatrix} 1. & 0 \\ 0 & 1. \end{pmatrix}$$

3. Gradient descent

Let’s go back to the original global error function that we used above for standard linear least squares regression. There we found the weights that gave the minimum total error by setting the gradient of the error function to zero, and then solving for the weights.

Gradient descent, is a more general way of finding the minimum of an error function. It can be used when the error function is much more complicated (e.g. it isn’t quadratic), and there is no linear solution. For optimization problems, gradient descent works best for concave error functions, where one doesn’t have to worry about multiple minima. Later we will run into this last kind of situation when we try to learn weights via supervised learning for a non-linear network with more than one layer of weights.

The idea is to start off at some location, $\mathbf{W_0}$ in weight space ($\mathbf{W_0}$ is just an initial guess), and iteratively move towards the minimum by always taking a step downhill. (See **Graphical illustration of Gradient** in **Appendix**). Again we take the elements of **W** and think of them as defining a point in weight space, i.e. a vector in weight space. We want to find a vector, say **Wbest**, that tells us which direction to move in weight space so that that the error measured at location $\mathbf{W_0}$ decreases by as much as possible. We want to go in the best direction, but we don't want to take too big of step or too small, so we need to control the length of **Wbest**, with a scalar η. In general, if we are at the ith point of an iteration, we would like:

$W^{i+1} = W^i + \eta$ (**Wbest** at W^i)

To derive a rule for **Wbest**, we assume that the error function is a smooth function of **W** so we can calculate the gradient values. The downhill direction is given by the negative gradient of the error function. (This can be seen by noting that $\nabla e \,.\, \hat{n} = |\nabla e| \cos(\theta)$, which is biggest when $\theta = 0$, in other words when $\hat{n}$ points in the direction of ∇e.)

So our gradient descent update rule is:

$$\mathbf{W}^{i+1} = \mathbf{W}^{i} - \eta \left.\frac{\partial e}{\partial \mathbf{W}}\right|_{\mathbf{W}^i} \tag{1}$$

As before, $\frac{\partial e}{\partial \mathbf{W}}$ is just short-hand for the matrix that has all the values of $\frac{\partial e}{\partial w_{ij}}$.

Note: Sometimes you might see the change in weights modeled as a dynamical system of differential equations, where time is a continuous variable:

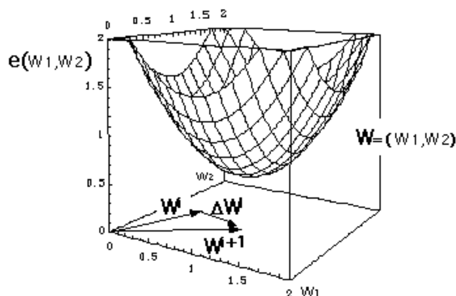
$$\frac{dW}{dt} = -\eta \frac{\partial e}{\partial \mathbf{W}}$$

We’ll see an example of this later. But as we saw with the recurrent lateral inhibition model, to simulate we have to approximate the differential equation by a discrete update rule as in equation (1).

So the result is that **Wbest** = $-\eta \frac{\partial e}{\partial \mathbf{W}} |_{W_i}$ evaluated at for the values of **W** in the ith iteration.

The figure below gives a graphical view in the small dimensional case of our example problem, where we have only two weights, and **W** can be represented as a vector:

$$\Delta\mathbf{W} = \boldsymbol{W}^{i+1} - \boldsymbol{W}^{i} = -\eta \frac{\partial e}{\partial \mathbf{W}}.$$



From the expression for the gradient which we wrote earlier in terms of outer products, we can obtain an expression for **ΔW**:

$$\frac{\partial e}{\partial \mathbf{W}} = -2\left(\sum_{i=1}^{N} \mathbf{y}_i \mathbf{x}_i^T - W \sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T\right)$$

```
In[492]:= Clear[wg];
eWgradient[wg_] :=
            Sum[Outer[Times,{yy[[i]]}, xx[[i]]], {i,Length[xx]}]-
            wg.Sum[Outer[Times,xx[[i]],xx[[i]]],
                {i,Length[xx]}];
```

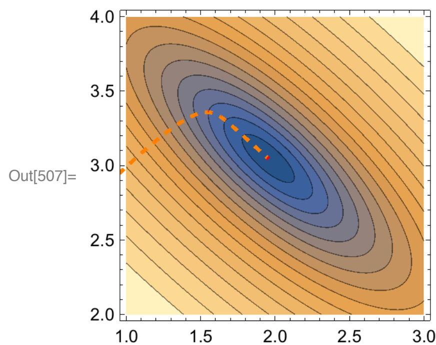
Now define a function T that specifies an updated weight matrix **W=wg**. You may have to adjust eta (= *η*) to make sure the steps are sufficiently small to get convergence, but not so small as to take long to converge.

$$\mathbf{W}^{i+1} = \mathbf{W}^{i} - \eta \left.\frac{\partial e}{\partial \mathbf{W}}\right|_{\mathbf{W}^i}$$

```
In[494]:= T[wg_] := wg + eta eWgradient[wg]
```

In[495]:=
```
i=0; eta = .005; wglist = {};

(*wg={{RandomReal[5.0],RandomReal[5.0]}}; *)

wg={{0,2}};

T[wg_] := wg + eta eWgradient[wg];

Clear[w1List];
w1List = NestList[T, wg, 100]; (*This is where the algorithm is!*)
gl = ListPlot[Flatten[w1List, 1], Joined → True,
   PlotStyle → {Orange, Dashed, Thick}];

Show[gc, gl]
w1List[[40]]
```

Out[507]=



Out[508]=
```
{{1.85191, 3.13931}}
```

We used **NestList[]** above to keep track of all the iteration values so we could graph them, but we could just output the final value after, say 50 steps, using Nest[]

In[510]:=
```
w1 = Nest[T, wg, 50]
```

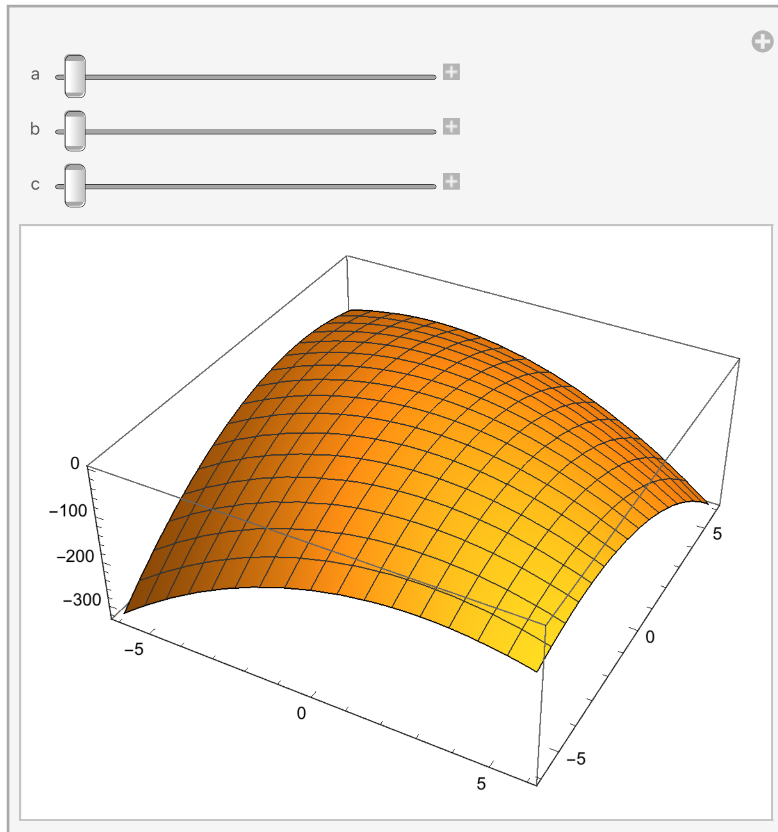
Out[510]=
```
{{1.89069, 3.10438}}
```

NOTE: There are often better algorithms for finding minima depending on the problem. For example, see the conjugate gradient method.

► 5. Graph a general 2D quadratic function. Explore values of a,b,c for which the function has a unique minimum or maximum.

```
In[390]:= Manipulate[Plot3D[a * x^2 + b * y^2 + c * x * y, {x, -6, 6}, {y, -6, 6}],
  {a, -3, 3}, {b, -3, 3}, {c, -3, 3}]
```

Out[390]=



4. "Brain-style" learning: Iterative Widrow-Hoff learning to estimate W

So far so good. But there are a several problems. First, we are interested in brain-style computation. Based on what we think we know about neurons, how could the brain compute transposes, do matrix inversion and multiplication?

Second, when we learn we don't seem to gather information on a whole set of training pairs, and then suddenly build a memory matrix. A more plausible assumption is that learning is *incremental, trial by trial*, rather than *batch.*

Another problem is purely computational. What if the dimensionality of the vectors is really big? Apart from the issue of neural plausibility, it is computationally expensive to invert large matrices. The above gradient descent procedure avoids the problem of inverting large matrices, but it involved computing a global error term over all the training pairs. We would like a method which would learn a regression mapping without having to store all the training pairs with the accompanying computation of a global error term. Instead, we'd like *to compute an error term incrementally, trial-by-trial*.

Can we discover the mapping **W** in such a way so as to be biologically plausible, and avoid having to invert a large matrix? The *Widrow-Hoff* rule provides an answer. The basic idea behind *Widrow-Hoff* learning is to update **W** iteratively with each new training pair. Let's start off with an arbitrary **W**, find out

which direction we would have to go in weight space to reduce the discrepancy between what **W** tells us **x** should map to and what it actually is, namely **y**. We take our clue from gradient descent, but apply it each time a new training pair comes along. We recompute the error term each time a new training pair comes along.

$$e(\mathbf{W}) = \left|\mathbf{y}_i - \mathbf{W}\,\mathbf{x}_i\right|^2$$

$$\frac{\partial e}{\partial \mathbf{W}} = -2\left(\mathbf{y}_i - \mathbf{W}\,\mathbf{x}_i\right)\mathbf{x}_i^T$$

$$\frac{\partial e}{\partial \mathbf{W}} = -\frac{d\mathbf{W}}{dt}$$

$$\mathbf{W}^{i+1} = \mathbf{W}^i + \eta_i\left(\mathbf{y}_i - \mathbf{W}^i\mathbf{x}_i\right)\mathbf{x}_i^T$$

Let's try out this update rule on our synthetic training pairs.

In[514]:=
```
ww1 = {{0,0}}; ww1list = {}; ww2list = {};
```

To visualize how the learning progresses, we are using ww1list and ww2list to store the first and second weights respectively, for each learning iteration.

In[515]:=
```
eta = 5;
```

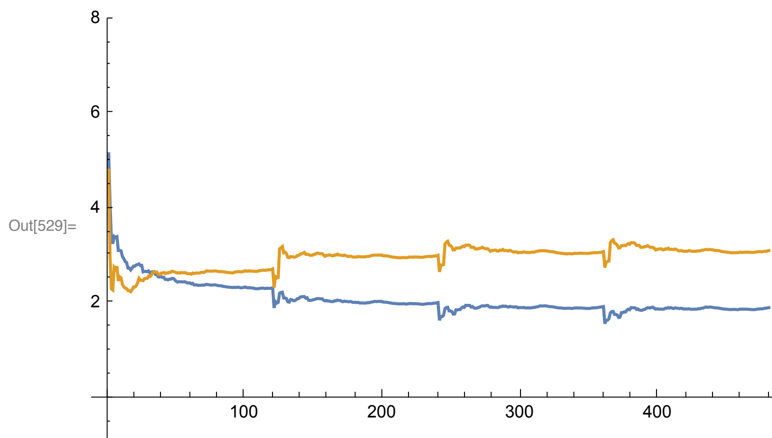
In[526]:=
```
i=0;
While[i<Length[data],
    ++i;
    in = {data[[i,1]],data[[i,2]]} ; out = {data[[i,3]]};
    ww1 = ww1 + (eta/i) Outer[Times,(out - ww1.in),in];
    ww1list = Append[ww1list,ww1[[1,1]]];
    ww2list = Append[ww2list,ww1[[1,2]]];
];
ww1
```

Out[528]=
```
{{1.89652, 3.10043}}
```

Keep executing the above input cell. You may have to run through the above loop several times before reaching stable convergence. We can plot up the two weights as a function of the iteration number to see how the Widrow-Hoff rule for weight modification eventually leads to two stable weights:

In[529]:=
```
ListPlot[{ww1list, ww2list}, PlotRange → {-1, 8}, AxesOrigin → {0, 0}, Joined → True]
```

Out[529]=



Note that we introduced a small change to the learning “constant” eta, replacing it by (eta/i)--it is no longer constant, but starts of large and decreases with each iteration.

Memory recall

We've seen several ways of finding the weights of a matrix that will approximately reproduce an output, given an input it has seen before. Let's try it out.

So in order to "recall" a response, from an input xx**[[6]]**, we run it through the "network" memory matrix **w1**:

```
In[530]:= w1.xx[[22]]
Out[530]= {1.97648}
```

And we can check to see how well it recalls by comparing with the target value:

```
In[531]:= yy[[22]]
Out[531]= 2.07983
```

Recall that the regression model of memory should generalize--*interpolate* and *extrapolate*. We saw a 3-dimensional example of interpolation in a graphical demonstration in the last lecture. But we can also extrapolate.

For example, **{11,15}** wasn't in the training set, but the expected output is:

```
In[532]:= w1.{11,15}
Out[532]= {67.3634}
```

The network has "learned" a surface, (e.g. for one particular training set, **y = 1.9x1 + 3.07x2** -- the coefficients of **x1** and **x2** will vary with each new random set of data) through the points specified in the training set.

A crucial point is that the linear associator will try to fit a plane (or hyperplane) through the data. If the data are not fitted well by that model, then the memory and generalization will not be good.

An obvious generalization is to fit hypersurfaces, rather than hyperplanes. And that is the direction we will head. But first let us look at linear regression from a point of view that you may not have thought of before.

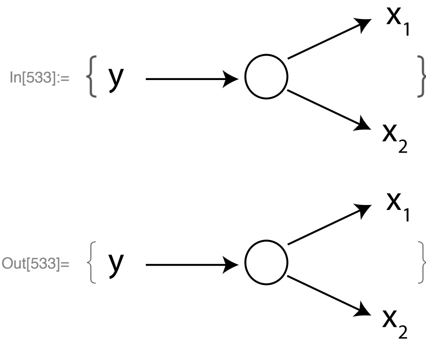
Underconstrained problems and redundancy

Learning a one-to-many mapping:

input dimensionality < output dimensionality

In[533]:=

Out[533]=



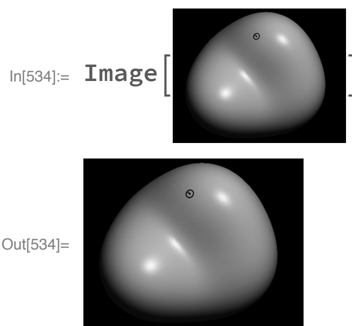
Motivation from vision

This example is very similar to the preceding example, except that we are going to try to learn **2D responses from 1D stimulus inputs**. At first this doesn't seem to make sense, because the mapping from 1D to 2D in general would be expected to be underconstrained by the data. But this is exactly what is needed in certain kinds of inverse problems that are said to be "ill-posed" (Poggio et al., 1985). They are made "well-posed" when there is some underlying regularity (sometimes called "smoothness") in the high-dimensional output space (e.g. Kersten et al., 1987).

In the figure below, the data reaching the eye can be represented as an intensity value at each pixel. However, you perceive a shape, and can even adjust the orientation of a “gauge figure” consisting of circle tangent to the surface at a point, and a vector perpendicular to the tangent plane (like a “thumb tack”, cf. Nefs et al., 2006). This is an example of the shape-from-shading problem in vision. The gauge figure has two degrees of freedom at each point in the image.

In[534]:= `Image[ ]`

Out[534]=

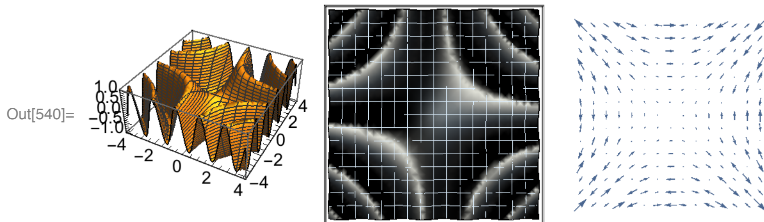


Thus "shape-from-shading" can be formulated as a problem of mapping N pixel intensities to N surface normals, where each surface normal is specified by two numbers--thus the mapping goes from N to 2N. E.g. given a 15x15 grid of intensities, estimate the 15x15 grid of 2D vectors that are normal to each point of the underlying surface. Thus one might start off with 225 numbers as input, but outputs 550 numbers.

In[535]:=
```
GradientFieldPlot[f_, {x_, xmin_, xmax_},
   {y_, ymin_, ymax_}, opts : OptionsPattern[]] :=
  VectorPlot[Evaluate[D[f, {{x, y}}]], {x, xmin, xmax}, {y, ymin, ymax}, opts]
```

In[536]:=
```
ssurface[x_, y_] := Sin[x y];
ggss = Plot3D[ssurface[x, y], {x, -4, 4}, {y, -4, 4}, MeshFunctions → {#1 &, #3 &}];
gsss = Plot3D[ssurface[x, y], {x, -4, 4}, {y, -4, 4}, ViewPoint → {0, 0, 30},
   Axes → False, Lighting → {{"Directional", White, {{1, 0, 1}, {1, 1, 0}}}},
   Mesh → 15, MeshStyle → Directive[Thin, LightBlue], ColorFunction → "GrayTones"];
gf = GradientFieldPlot[ssurface[x, y], {x, -4, 4}, {y, -4, 4}, Frame → False];
Show[GraphicsRow[{ggss, gsss, gf}]]
```

Out[540]=



The left panel illustrates an underlying surface that generates the image intensities shown in the middle. The shape-from-shading problem tries to take the graylevel values at points (approximated by grid boxes) in the middle panel, and output a 2-D vector for that grid, as shown in the right-hand panel.

If one sets up the shape-from-shading problem in terms of linear equations that have to be satisfied, one runs into the problem that there are too many unknowns for the number of equations.

Let's try a simple version of learning a mapping from a low to high-dimensional case. We'll use low-dimensional synthetic data whose generative process we'll keep hidden for now.

Synthetic data -- try not to look closely...

Here is the generative model:

input = x1;
output = yy = {y1, y2} = {a x1, b x1 + noise} ;

We want to learn the parameters {a,b} = {w1,w2}, from training pairs of inputs and outputs:
{x1, {{y1, y2}}}

In[556]:=
```
r3Dline[a_, b_] :=
  N[Table[{x1 = 1 RandomReal[], a x1, b x1 + 0.5 RandomReal[] - 0.25}, {30}], 2];
data = r3Dline[2, 3];
xx = data[[All, 1]];
yy = data[[All, 2 ;; 3]];
```

So the input data is a list of 1D scalars, and the output data a list of 2D vectors. Let's apply the Widrow-Hoff algorithm to learn the relationship between the input stimuli, xx, and the output responses, yy.

Iterative Widrow-Hoff learning

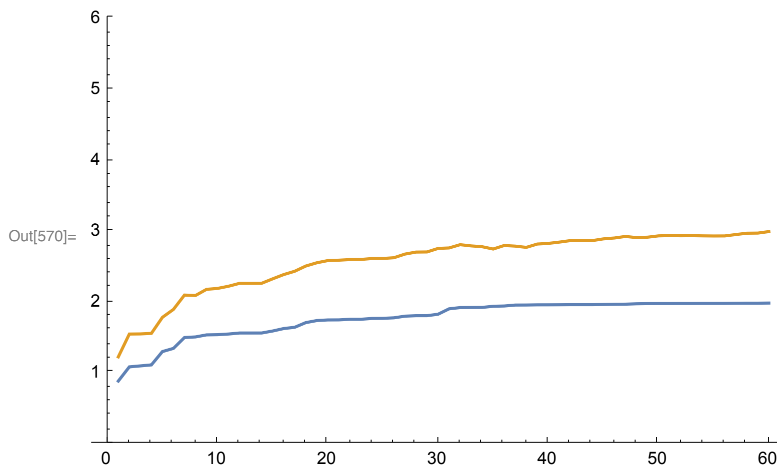
In[560]:=
```
w1 = {{0},{0}}; w1list = {}; w2list = {}; eta = 4.0;
Dimensions[w1];
```

In[567]:=
```
i=0;
While[i<Length[data],
    ++i;
    out = {data[[i,2]],data[[i,3]]} ;
    in = {data[[i,1]]};
    w1 = w1 + (eta/i) Outer[Times,(out - w1.in),in];
    w1list = Append[w1list,w1[[1]][[1]]];
    w2list = Append[w2list,w1[[2]][[1]]];
];
{xx[[1]],Flatten[w1.{xx[[1]]}]}
```

Out[569]= {0.332625, {0.660305, 0.995571}}

In[570]:=
```
ListPlot[{w1list, w2list}, PlotRange → {0, 6}, AxesOrigin → {0, 0}, Joined → True]
```

Out[570]=

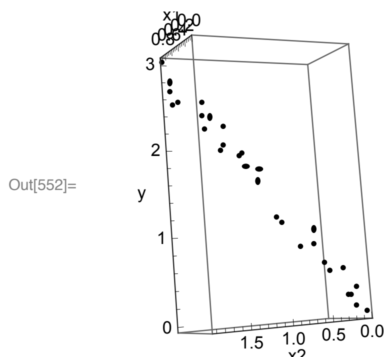


Visualizing the underlying structure

So why does it work to learn to predict a 2D value from a 1D input? The reason, of course, is that the underlying structure of the output data is highly constrained, and in fact lies close to a straight line in 3-space.

In[552]:=
```
Show[Graphics3D[Point /@ data], Axes → True,
 AxesLabel → {"x1", "x2", "y"}, ImageSize → Small]
```

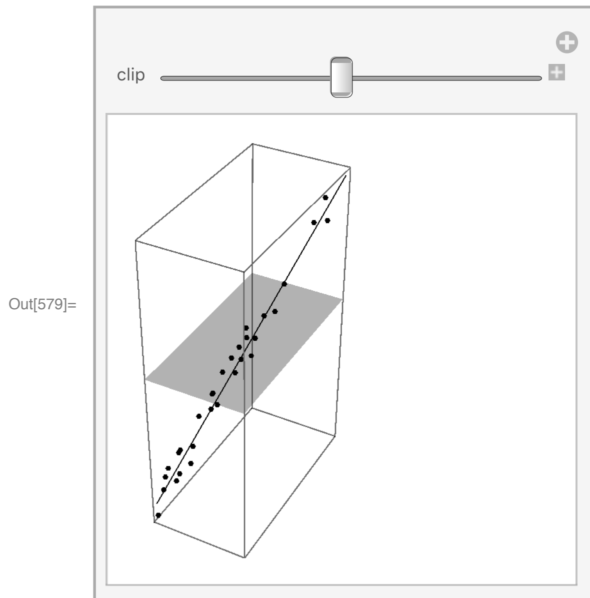
Out[552]=



Below you can visualize how specifying a specific an input value for y determines the height of the plane. The line intersecting the plane determines the pair of out put values {x1,x2}.

In[577]:=
```
trueline = Graphics3D[Line[{{0, 0, 0}, {1, 2, 3}}]];
trueline = Line[{{0, 0, 0}, {1, 2, 3}}];
Manipulate[Graphics3D[{{Point /@ data}, trueline}, ClipPlanes → {0, 0, 1, clip},
  ClipPlanesStyle → Opacity[0.3], ImageSize → Small], {clip, -3, 0}]
```

Out[579]=



Here is the generative model for our synthetic data:

x1 -> {x2, y},

where the underlying structure is a straightline through the origin with some additive noise. The output {x2, y} is given by:

{x2, y} = {a x1, b x1 + noise}

An input, a scalar quantity x1 specifies a plane. We've found a line that intersects the plane and for which points in our data set our close to that line. The intersection point gives us the coordinates {x2, y} for our regression fit, or "recall".

We learned the parameters {a,b}, corresponding to the weights, from training pairs of inputs and outputs: {x1, {{x2, y}}}

r3Dline[a_, b_] := N[Table[{x1 = 1 RandomReal[], a x1, b x1 + 0.5` RandomReal[] - 0.25`}, {30}], 2];

data = r3Dline[2,3];

It produced the coordinates of a noisy line in 3-space.

Making estimates of high-dimensional functions from lower dimensional inputs happens a lot, and is common to many so called "early vision" problems (Poggio et al., 1985). See Freeman et al., (2000) and Barron, J. T., & Malik, J. (2015).

Introduction to multi-layer nets

This was an earlier exercise. For linear networks, no computational power is gained by having extra

layers:

```
y1 := W0.y0;
y2 := W1.y1;
```

is equivalent to:

```
y2 := W1.(W0.y0) := W1.W0.y0 := W3.y0;
```

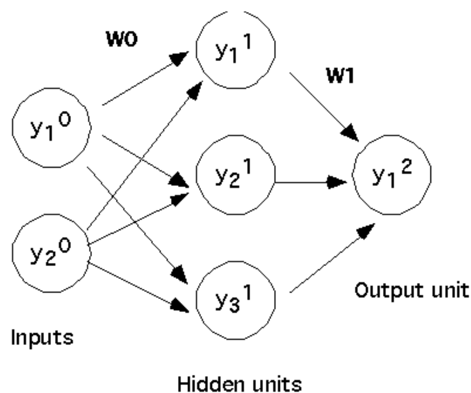
where **W3** is just another matrix. However, if the inner product is followed by a non-linear transformation, then concatenating layers of neural elements allows one to compute more complex transformations:

```
y2 := σ[W1.σ[(W0.y0)]];
```

where ***σ*[ ]**, for example, is a sigmoidal "squashing function" or *logistic function*:

```
σ[x_] := 1 / (1 + Exp[- (x - .5)]);
```

6. 2 input units, 3 hidden units, 1 output unit




```
In[592]:= Clear[y2, σ2];
σ2[x_] := N[1 / (1 + Exp[- (x - .5) * 100])];
W1 = {{-13.2328, 6.06398, 6.04958}};
W0 = {{-0.937564, -1.09841}, {-9.95589, 3.85642}, {3.79034, -10.1737}};
y2[y0_] := Chop[σ2[W1.σ2[(W0.y0)]]];
```

7. What logical function does this network compute?

```
In[597]:= {y2[{0, 0}], y2[{0, 1}], y2[{1, 0}], y2[{1, 1}]}

Out[597]= {{0}, {1.}, {1.}, {0}}
```

8. What logical function will the above network graph compute with the following weights?

```
In[598]:= W0 = {{1.9009195689645, 1.9251997561975753}, {2.177312432892121, 2.198392078532626},
  {-2.8297064197335953, -2.8648165030618693}};
  W1 = {{1.1241018372719358, 1.7600034188676417, -10.048562036295957}};

In[600]:= Clear[y2];
y2[y0_] := Chop[σ2[W1.σ2[(W0.y0)]]];

In[602]:= {y2[{0, 0}], y2[{0, 1}], y2[{1, 0}], y2[{1, 1}]}

Out[602]= {{0}, {1.}, {1.}, {1.}}
```

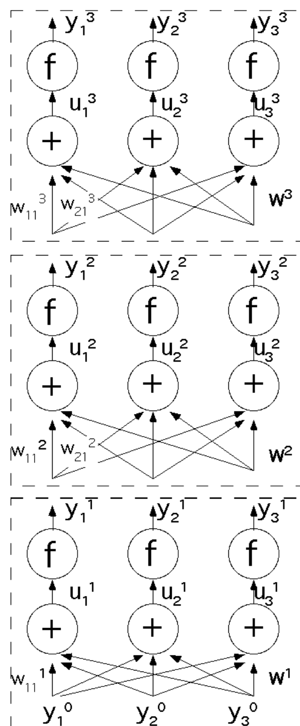
The demonstration illustrates that we can compute XOR as well as OR using a network with more than

one layer of weights.

Multi-layer nets: Learning and the error back-propagation algorithm

So we've seen that the non-linear generic neuron allows increased computational power when we add an extra layer of weights.

Suppose we have a multi-layer network with 3 layers of weights. The output from the first layer is: $\mathbf{y}^1 = f[\mathbf{u}^1] = f[\mathbf{W}^1.\mathbf{y}^0]$. The output of the second layer is: $\mathbf{y}^2 = f[\mathbf{u}^2] = f[\mathbf{W}^2.\mathbf{y}^1]$. And so forth. To simplify a bit, we'll assume each layer has the same number of units and weights:



NOTE: the symbol f replaces σ.

The problem is: how to assign the weights? We'd like to find them through supervised learning.

For any complex system that is required to achieve a target goal, for the system to work, each component must contribute towards the goal. If the goal is not met, one has to figure out which component needs to be fixed. If the goal is met, each component contributed something towards the goal. How does one assign the credit for success or failure to a component?

This problem is called the *credit-assignment problem*. In particular, for the above multi-layer network, how do we know which weights to adjust by how much appropriate to learning a given input/output relation?

$$\{x^p, t^p\},\ \ y^0 = x^p,\ \ \text{training pairs p} = 1,\ldots,\text{M}$$

For a given input $\{y^0 = x^p\}$, we feed forward the information to the last layer (layer L) to produce an

output {$\mathbf{y}=\boldsymbol{y}^L$}. We compare the output to the target value supplied by the "teacher" {t = t^p}, and compute the error as the sum of squared differences:

$$E(\{w_{ij}^{\lambda}\}) = \sum_{k=1}^{N} (y_k(y^0; w_{ij}^{\lambda}) - t_k)^2$$

where the sum is over all N output units. (For simplicity, we left out the superscript p in t_k^p. The subscript k in t_k^p means the component of the corresponding vector of activity t^p.) λ indexes the weight layers going from λ=1 to L. Note that the y's at any point after the input depend on the u's (the weighted sum before the non-linearity), each of which in turn depends on all the w_{ij}^{λ}s before it.

The trick is to find out how to assign credit (and blame) for the error to each of the weights. Gradient descent provides the answer. Adjust the weights such that:

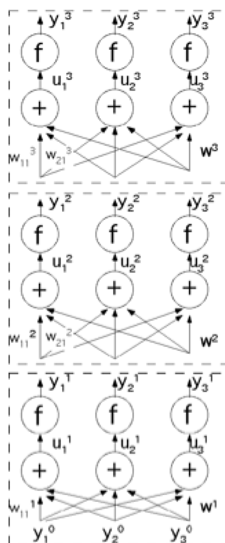
$$\text{new } \mathbf{w}_{ij}^{\lambda} = \text{previous } \mathbf{w}_{ij}^{\lambda} + \Delta\mathbf{w}_{ij}^{\lambda}$$

where

$$\Delta w_{ij}^{\lambda} = -\eta \frac{\partial E}{\partial w_{ij}^{\lambda}}$$

Again, this formula means that if we calculate the gradient $\nabla E = \partial_{w_{ij}^{\lambda}} E$, its negative direction points in the direction of steepest descent. Thus if we update the weight vector to point in this direction, then at the next step we should in general have a lower value of E. I.e.

E(new w_{ij}^{λ}) < E(previous w_{ij}^{λ}).



The appendix shows how to calculate a weight adjust with just one layer of weights. This case is related to the area of Generalized Linear Regression (not to be confused with the General Linear Model!).

We won’t go through the math here for the multiple layer case--it is really just a very complicated application of taking derivatives where the complication arises in keeping track of all the indices. Instead we assume we have the rule that tells us what the effective error term--the delta value-- and adjustment that needs to be for each weight, and see how it gets put together to converge on weights that tend to

minimize the overall output errors.

Summary of backprop algorithm

1. Initialize the weights to small random values

2. Pick a pattern from the input/output collection, say the pth pattern: $\{x^p, t^p\}$. Run the input vector, x^p, feedforward through the network. This will generate a set of values u_i^L and y_i^L in all the nodes of the network. Keep in mind that u_i^L is the linear weighted sum of its inputs arriving from layer L-1: $u_i^L = \sum_{j=1}^{3} w_{ij}^L y_j^{L-1}$. And $y_i^L = f(u_i^L)$.

Calculate a delta term (analogous to the Widrow-Hoff rule) for the output layer L:

$$\partial_i^L = \left(t_i^p - y_i^L\left(x^p\right)\right) f'\left(u_i^L\right)$$

Note why it is important to have an expression for the *derivative* of the squashing function f(). The derivative has a particularly nice form when $f(u) = 1/(1+e^{-u})$. (see Appendix, Logistic function - a smooth, differentiable non-linearity)

3. Propagate the errors back through the layers:

$$\partial_i^\lambda = f'(u_i^\lambda) \sum_{k=1}^{N} \partial_k^{\lambda+1} w_{ki}^{\lambda+1} \qquad \lambda = L-1, \ldots, 1$$

...the error back propagation or "back prop" part.

4. Calculate weight adjustments (analogous to the outer product part of the Widrow-Hoff) and update using:

$$\Delta w_{ij}^\lambda = \eta \partial_i^\lambda y_j^{\lambda-1}$$

5. Repeat steps 2 to 4 until convergence.

One can accumulate the weight adjustments for each training pair, and then update them all at once.

$$(\Delta w_{ij}^\lambda)_p = \eta \partial_i^\lambda y_j^{\lambda-1}$$

$$\Delta w_{ij}^\lambda = \sum_{p=1}^{M} (\Delta w_{ij}^\lambda)_p$$

In practice, updating the weights after each training pair often works better than accumulating a bunch of input/output pairs, and then computing the cumulative global error. The reason is that by randomly sampling a training pair, the "descent" may actually climb the global error function defined by the entire set. As we will see later with the Boltzmann machine, occasional climbing is useful to avoid local minima.

There are a number of derivations and illustrations on the web that you might find useful. See for example, http://galaxy.agh.edu.pl/~vlsi/AI/backp_t_en/backprop.html

Also, Andrew Ng has several excellent coursera videos on error backpropagation, which are also accessible via youtube.

Backprop simulation example: XOR

With appropriate weights, 2 weight layers with 3 hidden units can solve the XOR problem. But this is still a tough problem to learn, mainly because it requires that two very different inputs map to the same

output. See the supplementary material for a Mathematica demo that learns the weights for solving the XOR problem.

Next

Non-linear networks & "energy": Hopfield networks

Probabilistic models. From energy to probability. Boltzmann machines

Appendix

Pseudoinverse solution for the second (underconstrained redundancy) case above

```
In[438]:= r3Dline[a_, b_] :=
  N[Table[{x1 = 1 RandomReal[], a x1, b x1 + 0.5 RandomReal[] - 0.25}, {30}], 2];
data = r3Dline[2, 3];
xx = data[[All, 1]];
yy = data[[All, 2 ;; 3]];
```

We can calculate the memory matrix using the **PseudoInverse** in this case too:

```
In[442]:= Dimensions[xx];
Dimensions[yy];
```

```
In[444]:= matmem = Transpose[PseudoInverse[Transpose[{xx}]].yy]
Out[444]= {{2.}, {3.02372}}
```

Recall

Let's check out a few values to see how well the memory matrix can recall a 2 dimensional output, given a one dimensional input.

```
In[445]:= {{"recall: ",matmem.Transpose[{xx}][[5]], matmem.Transpose[{xx}][[13]], matmem.Transpose[{
{"true values: ", yy[[5]], yy[[13]], yy[[22]]}}//Grid
Out[445]=
   recall:     {1.38803, 2.0985}  {1.45095, 2.19364}  {0.163506, 0.247198}
true values:   {1.38803, 2.32505}  {1.45095, 2.26087}  {0.163506, 0.178338}
```

Another function for calculating the least-squares solution

Regression and learning in linear neural networks. Last time we showed 4 different ways to find the generating parameters {2,3} for the following data:

```
In[446]:= rsurface[a_, b_] := N[Table[{x1 = 1 RandomReal[],
     x2 = 1 RandomReal[], a x1 + b x2 + 0.5 RandomReal[] - 0.25}, {120}], 2];
data = rsurface[2, 3];
yy = data[[All, 3]];
xx = data[[All, 1 ;; 2]];
```

Linear regression is so common that *Mathematica* has added the following function to find the least squares parameters directly:

In[450]:=
```
LeastSquares[xx, yy]
```

Out[450]=
```
{2.03325, 3.03118}
```

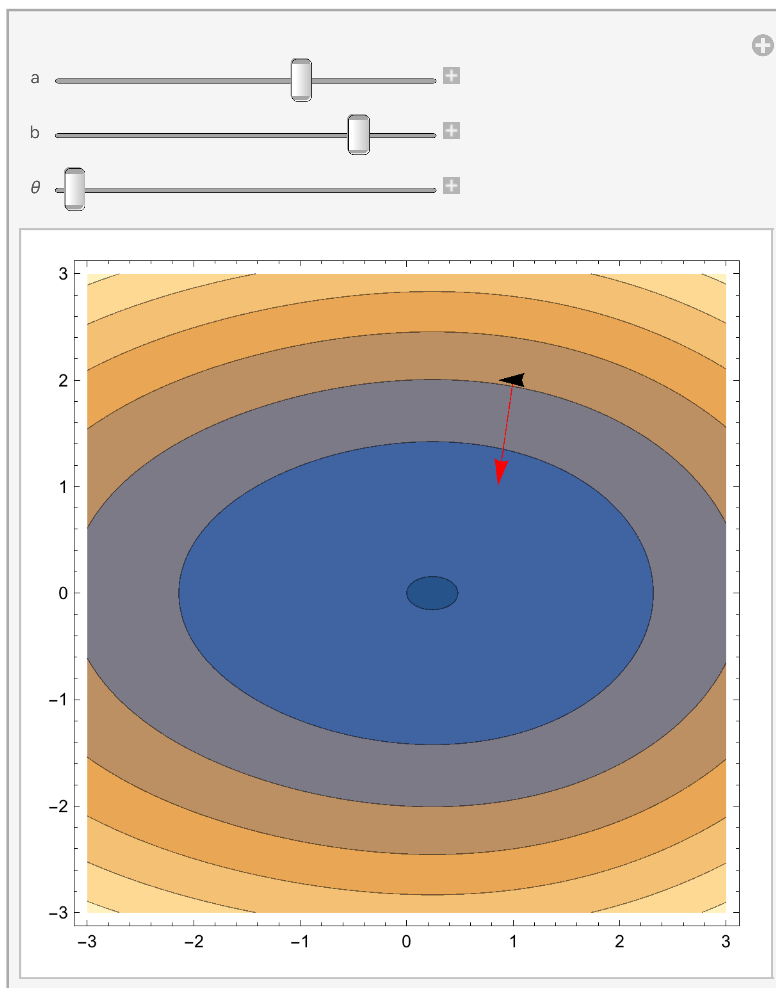
Graphical illustration of Gradient

The function f(x,y) is plotted as a contour plot. The demo below calculates and plots the vector **gv** = **∇f**. After normalized to unit length, it is plotted as red arrow. The red arrow should always point down the "hill" in the contour plot. You can manipulate the position of the point at which the gradient gets evaluated with the a and b sliders.

The θ slider controls the direction of a second vector **un** (in black) whose length is determined by the projection: **∇f.un.** Note how the length
of the projection is always biggest in the direction of the gradient (red arrow).

In[603]:=
```
Clear[a, b, gv, un, gm, g2, g3, x, y];
f[x_, y_] := 2 x^2 + 5 y^2 - Sin[x];
g1 =
  ContourPlot[f[x, y], {x, -3, 3}, {y, -3, 3}, PlotPoints → 50, ImageSize → Medium];
Manipulate[
 gv = N[D[2 x^2 + 6 y^2 - Sin[x], {{x, y}}] /. {x → a, y → b}];
 gn = -Normalize[gv];
 un = {Cos[θ], Sin[θ]};
 gm = (un.gn) un;
 g2 = Graphics[{Red, Arrow[{{a, b}, gn + {a, b}}]}];
 g3 = Graphics[Arrow[{{a, b}, gm + {a, b}}]];
 Show[{g1, g2, g3}], {{a, 1}, -3, 3}, {{b, 2}, -3, 3}, {θ, 0, 2 * Pi}]
```

Out[606]=



Logistic function - a smooth, differentiable non-linearity

As we saw above, it can be useful to have a non-linearity which is smooth enough to be differentiable. **D[]** returns an expression for the derivative, so to define a function that is the derivative of another we write:

In[455]:=
```
Df1[x_] := D[f[t],t] /.t->x
Df1[x_] := Evaluate[D[f[t],t]]
```

The more direct way is to use "operators" or functionals that take functions as inputs and return functions as outputs. **Derivative[]**, or **f'[x]** return functions:

In[457]:=
```
(*Df[x_] := Derivative[f[x],x]*)
Df[x_] := f'[x]
```

Note that the derivative has a particularly simple expression in terms of **f[x]**, which you can verify by comparing **Simplify[Df[x]] with Simplify[hh[x]], or Simplify[Df[x]-hh[x]]:**

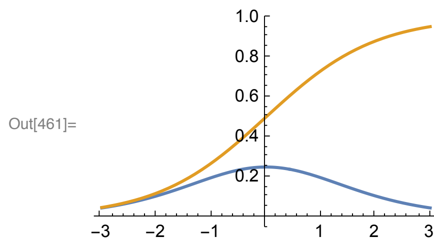
In[459]:=
```
f[x_] := 1 / (1 + Exp[-x]);
```

In[460]:=
```
hh[x_] := f[x](1-f[x]);
```

Here is a plot of the sigmoid and its derivative:

In[461]:=
```
Plot[{Df[x], f[x]}, {x, -3, 3}]
```

Out[461]=



Exercise: calculate the gradient for the last layer in a back-prop network

Let's see how to calculate the gradient for the last layer. Then we will generalize the result to an expression that tells us how to adjust the ijth weight for the connection in the λth layer.

In[462]:=
```
Remove["Global`*"]
```

We start with the error term for the jth training pair. Note that y_j depends on the inputs x, but these are fixed. At this point, we care about the variable weights.

In[463]:=
```
(1 / 2) * (tᵢ - yᵢ[wᵢⱼ]) ^2
```

Out[463]= $\frac{1}{2}\left(t_i - y_i[w_{ij}]\right)^2$

Take the derivative with respect to w_{ij}

In[464]:=
```
eq1 = D[(1 / 2) * (tᵢ - yᵢ[wᵢⱼ]) ^2, wᵢⱼ]
```

Out[464]= $-\left(t_i - y_i[w_{ij}]\right) y_i'[w_{ij}]$

y depends on u which in turn depends on the w's, so using the chain rule in calculus, we can write $y_i'(w_{ij})$ as:

In[465]:=
```
eq2 = D[yᵢ[uᵢ[wᵢⱼ]], wᵢⱼ]
```

Out[465]= $u_i'[w_{ij}]\, y_i'[u_i[w_{ij}]]$

But $y_i'(u_i)$ is the derivative of the squashing function f: $y_i'(u_i)$=f'(u_i) So we can substitute f' for y in eq2

In[466]:=
```
eq3 = eq2 /. yᵢ'[uᵢ[wᵢⱼ]] → f'[uᵢ]
```

Out[466]= $f'[u_i]\, u_i'[w_{ij}]$

Now u is given by:

In[467]:= $u_i = \sum_{k=1}^{N} x_k\, w_{i,k}$

Out[467]= $\sum_{k=1}^{N} x_k\, w_{i,k}$

Let's pick an arbitrary input input, say i=5. Suppose we take the derivative with respect to $w_{j,i}$, with i = 5? Then with respect to $w_{j,5}$, we have:

In[468]:= $D\left[\sum_{k=1}^{10} x_k\, w_{i,k},\ w_{i,5}\right]$

Out[468]= x_5

So from this we can guess that in general (i.e. any i) we should have:

$$\Delta w_{ij} = \left(t_i - y_i(\,x;\ w_{ij})\right) f'\,(u_i)\, x_j$$

where x_j is the jth "input" to the last layer. This is the *delta-rule*. But this delta rule only works for the output layer. We need to know how to update all the weights.

We don't have direct access to the hidden units, and the problem is how to find the error signals for the hidden layers. It turns out that the delta error terms can be propagated back in terms of a weighted sum of the delta terms at the level above. We will see how this update rule gets applied to the weights starting from the top--i..e. those closest to the outputs, and then applied recursively down towards the inputs. Hence "back-propagation". We will relabel $x_k$to be y_k^λ, where λ indicates the layer.

So y_k^0corresponds to the set of inputs to the network (bottom of figure).

To derive the recursive rules to relate deltas at an earlier layer to the next higher layer takes some work

and careful book keeping with indices. We won't go through the details here. Let's look at a summary of the algorithm that results.

References

Barron, J. T., & Malik, J. (2015). Shape, Illumination, and Reflectance from Shading. IEEE Transactions on Pattern Analysis and Machine Intelligence, 37(8), 1670–1687. http://doi.org/10.1109/TPAMI.2014.2377712

Bishop, C. M. (1995). Neural Networks for Pattern Recognition. Oxford: Oxford Univeristy Press.

Duda, R. O., & Hart, P. E. (1973). Pattern classification and scene analysis. New York.: John Wiley & Sons.

Duda, R. O., Hart, P. E., & Stork, D. G. (2001). Pattern classification (2nd ed.). New York: Wiley.

William T. Freeman, Egon C. Pasztor and Owen T. Carmichael (2000). Learning Low-Level Vision. International Journal of Computer Vision 40, 25-47

Kersten, D., O'Toole, A. J., Sereno, M. E., Knill, D. C., & Anderson, J. A. (1987). Associative learning of scene parameters from images. Appl. Opt., 26, 4999-5006.

Knill, D. C., & Kersten, D. (1990). Learning a near-optimal estimator for surface shape from shading. CVGIP, 50(1), 75-100.

Nefs, H.T., Koenderink, J.J. & Kappers, A.M.L. (2006). Shape-from-shading for matte and glossy objects. Acta Psychologica, 121 (3), 297-316

Poggio, T., Torre, V., & Koch, C. (1985). Computational vision and regularization theory. Nature, 317, 314-319.

Geman, S., Bienenstock, E., & Doursat, R. (1992). Neural networks and the bias/variance dilemma. Neural Computation, 4(1), 1-58.

© 1998-2016 Daniel Kersten, Computational Vision Lab, Department of Psychology, University of Minnesota.