Introduction to Neural Networks

Sculpting energy/cost landscapes: interpolation and gradient descent

Initialization

```
Off[SetDelayed::write]
Off[General::spell1]
```

Overview

Last time

Probability, statistics review

Today

Final projects

Exploring the “added value” of understanding neural processing from the point of view of statistical inference

From costs/energy to update rules

Bias/variance

In the near future: Supervised learning: neural networks in the context of statistics/machine learning
Belief propagation

Using energy and gradient descent to derive update rules

Earlier we studied how to set up the weights in a discrete response TLU network for the correspondence problem in stereopsis. The weights were determined by an analysis of the constraints needed to find a unique correspondence between the pixels in the left and right eyes. We didn't compute energy in that example, but pointed out that energy could play a useful role as an index to describe how well the network's state vector was moving towards the correct answer in state space. In particular, the energy function contains information about the stable points in state space.

In effect, we sculpted the energy landscape by hand-wiring the weights according to the constraints that were determined heuristically.

In the TIP examples, we reconstructed stored letters from partial information. In that case, the weights were determined by Hebbian learning. So the energy landscape was sculpted by state vectors to be stored.

But we can also do things the other way around. Rather than figuring out the weights for a Hopfield-style network that has a known relationship to an energy function, we first specify the energy function, and then figure out an update rule that will descend the energy landscape.

We followed an analogous strategy when we set up an error function in terms of weights, and then did gradient descent to find the weights that minimized the error to learn the weights. But one can also set up the analog to the error function (weights variable), that is an energy function of the state vector (weights fixed), and then use gradient descent to derive a rule to find minima of this energy function. From the point of view of neural networks, this update rule may look nothing like what neurons do. But it may be the best way to start--that is, by sculpting the energy function directly, and then see what emerges in terms of an update rule.

We are going to follow this strategy in this notebook on a simple problem of interpolation. It will turn out that our update rule is the simplest neural model--a linear summer. However, this kind of analysis provides a starting point for more complicated energy functions with correspondingly non-linear update rules.

The energy function is sometimes referred to as a cost function or objective function.

In the second main part of this notebook (Deriving learning rules), we return to the problem of deriving learning rules from cost functions over weights using gradient descent. We apply it to the problem of self-organization where the goal is to learn a decorrelating set of weight vectors, which (unlike PCA) are not necessarily orthogonal.

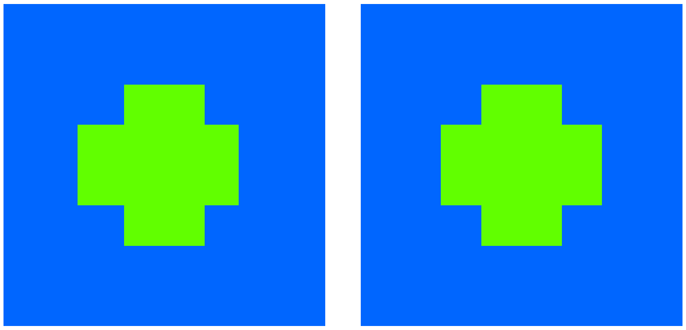
Our main tool is gradient descent. Given an objective or energy function, there are often better tools for finding the minimum (e.g. see Hertz et al.). But gradient descent is a simple and intuitive starting point.

Interpolation problems in perception

A major theoretical problem in vision has to do with the fact that local changes in image intensities are usually ambiguous in natural images. A change in shading can mean an change in shape (or a change in illumination, e.g. a cast shadow). A change in image color can mean a change in the reflectivity of a surface (or a change in the illumination--two quite different causes). Changes of intensity of pixels in time provide information about surface structure, and the viewer's relation to that surface. In order to solve problems such as those above, researchers have studied special cases: shape-from-shading, reflectivity from color (color constancy), optic flow field from the flow of intensities, and more. One recurring theme in these problems is that the data available in the image does not fully constrain the estimate of the surface or surface properties one would like to compute.

Earlier we studied the random dot stereogram in which each image had many local features densely packed, but there was ambiguity in matching left eye pixels to those in the right. Here another stereo example. This time there are few features sparsely packed.

```
backwidth = 50; backheight = 50;
     backwidth        backheight
x0 = ─────────; y0 = ──────────;
         2                2
         backwidth            backheight
vwidth = ─────────; vheight = ──────────;
             4                    8
        backwidth          backheight
vxoff = ─────────; vyoff = ──────────;
            2                  2
         backwidth
hwidth = ─────────;
             8
          backheight
hheight = ──────────;
              4
        backwidth
hxoff = ─────────;
            2
        backheight
hyoff = ──────────;
            2
gleft =
  Show[Graphics[{Hue[0.6`], Rectangle[{0, 0}, {backwidth, backheight}], Hue[0.27`],
     Rectangle[{x0 - vwidth, y0 - vheight}, {x0 + vwidth, y0 + vheight}], Hue[0.27`],
     Rectangle[{x0 - hwidth, y0 - hheight}, {x0 + hwidth, y0 + hheight}]}],
   AspectRatio → Automatic, DisplayFunction → Identity];
gright = Show[Graphics[{Hue[0.6`], Rectangle[{0, 0}, {backwidth, backheight}], Hue[
      0.27`], Rectangle[{x0 - vwidth - 1, y0 - vheight}, {x0 + vwidth - 1, y0 + vheight}],
     Hue[0.27`], Rectangle[{x0 - hwidth, y0 - hheight}, {x0 + hwidth, y0 + hheight}]}],
   AspectRatio → Automatic, DisplayFunction → Identity];
Show[GraphicsRow[{gright, gleft}], DisplayFunction → $DisplayFunction]
```



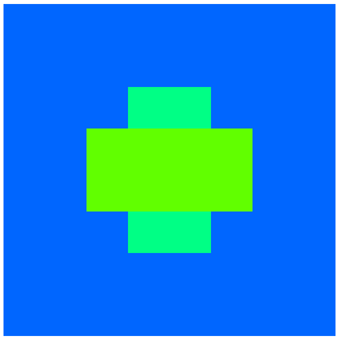
Try a motion analog: Show[gleft,DisplayFunction→ $DisplayFunction]; Show[gright,DisplayFunction→ $DisplayFunction]; and then group the two pictures, and play as a slow movie

---

If you can cross your eyes, so that the left image is in the right eye, and the right image in the left, you will see a green horizontal bar floating out in front of a green vertical bar. So-called "free-fusing" isn't easy, but when you've got it, you should see a total of three green crosses. The one in the middle is the one in which the two images are fused by your brain--and this is the one we are talking about.

The interesting point here is that even though there is no local information in the image to support the percept of a horizontal occluding bar, observers still see an illusory completion. It looks a bit like the figure below, except that the color of the horizontal bar is changed here slightly just for illustration.

```
Show[Graphics[{Hue[0.6`], Rectangle[{0, 0}, {backwidth, backheight}], Hue[0.42`],
   Rectangle[{x0 - hwidth, y0 - hheight}, {x0 + hwidth, y0 + hheight}], Hue[0.27`],
   Rectangle[{x0 - vwidth, y0 - vheight}, {x0 + vwidth, y0 + vheight}]}],
 AspectRatio → Automatic, ImageSize → Small]
```



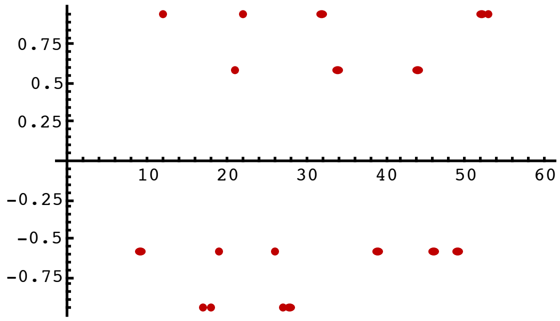
The visual system seems to *interpolate* a surface between salient points--in this case the salient points are the vertical edge segments of the horizontal bar.

(Side note: The perceptual interpolation in the above example is not straightforward. In fact, one's first guess might be that observers should not see the horizontal bar as a plane, but rather they would interpolate a surface that on the left is close to the viewer, but then descends back towards the depth of the vertical bar, stays there, and then comes back towards the viewer on the right. Why people usually don't see this has been studied by: Nakayama, K., & Shimojo, S. (1992).)

We will study a simpler case of interpolation--namely filling in a line between feature points. We will do this by constructing an energy function first, and then calculating an update rule by gradient descent. But the same principles apply to computational models of stereopsis, shape-from-shading, optic flow and other problems in early visual processing.

Energy: Data and smoothness terms

Suppose we have a set of points {i, di}, where i indexes the point, and di is the height.



We would like to find a smooth function, f[i] ~ di, that approximately fits the data points. However, we'll assume we don't have specific knowledge of the parametric form of the underlying function. In other words, we don't want to assume linear, quadratic, or general polynomial fits, or sinusoidal fits. We could just connect the dots. But if there is any wiggly noise, we'd have lots of wiggles that shouldn't be there. Further, the model probably wouldn’t generalize well. We want a "smoother fit".

We assume that there are two constraints that will guide the problem of sculpting an energy function to reconstruct a line (or surface):

1. Fidelity to the data;
2) Smoothness of the fit.

Suppose {d$_i$} are the data points, where the i's come from a subset, D of the total domain over which our reconstructed function, f is defined. Fidelity to the data can be represented by an energy or cost term that is big if the estimate of f is too far away from the data:

$$E_d = \sum_{i \in D} (f_i - d_i)^2$$

Smoothness can be represented by an energy cost in which near-by values of the estimate, f, are required to be close:

$$E_s = \sum_i (f_i - f_{i+1})^2$$

We combine two constraints by adding:

$$E = E_d + E_s = \sum_{i \in D} (f_i - d_i)^2 + \lambda \sum_i (f_i - f_{i+1})^2$$

λ is a free parameter that allows us to control how much the smoothing should dominate the data or fidelity term. Note we assume that the smoothness term is independent of the data, and is equivalent to assuming a Bayesian prior in statistics (Poggio et al., 1988; Kersten et al., 1987).

If we wish to start off with some initial guess for the values of f, we can successively improve our estimate by sliding down the slope of E in the steepest direction. As we saw in an earlier lecture, this says that the rate of change of f$_i$ in time should be proportional to the negative slope of E in the direction of f$_i$

$$\frac{df_i}{dt} = -\frac{\partial E}{\partial f_i}$$

Proof, in case you are still wondering:

$$\Delta E = \nabla E \bullet d\vec{f} = |\nabla E| |d\vec{f}| \cos\theta$$

For step Δt, the biggest decrease in E is when cosθ = -1. In other words, when the vectors $\overrightarrow{\nabla E}$ and $\overrightarrow{\text{df}}$ are pointing in opposite directions. So if we make a change df in proportion to -∇E for each time step Δt, we will reduce the energy.

$$\overrightarrow{\text{df}} \propto -\nabla \text{E}$$

In the limit Δt→0, we can set:

$$\left(\frac{\partial E}{\partial f_1}, \frac{\partial E}{\partial f_2}, \ldots\right) = -\left(\frac{df_1}{dt}, \frac{df_2}{dt}, \ldots\right)$$

As mentioned above, we've encountered gradient descent before when we calculated the derivative of the error function with respect to the weights when we studied linear regression and the widrow-hoff rule.

Taking the derivative of the above cost function:

$$\frac{df_i}{dt} \propto -(f_i - d_i) - \lambda(2f_i - f_{i+1} - f_{i-1})$$

(One can use *Mathematica* to verify the pattern of the terms in the derivative.) Or in discrete time steps of dt,

$$f_i(t + dt) = f_i + dt\{-2(f_i - d_i) - 2\lambda(2f_i - f_{i+1} - f_{i-1})\}$$

(Note: There is a slightly messy issue of book-keeping in that, we have to pay attention to when we are updating f's that don't have data support. We show one way to handle that below.)

With a little inspection, you can see that the value of f at time t+dt, is just the weighted sum of the values of f at time t, and the data, d. A weighted sum calculation is exactly what our linear model of a neuron does. So we've derived an update rule for our energy function that could be implemented with a standard linear neural model.

What is the weight matrix? Are the diagonals zero? Is the matrix symmetric?

It turns out that as long as the energy function is a quadratic function of the f's, the update rule will be linear. Why?

The power of this approach is that one can construct more complicated energy functions, and derive gradient descent update rules that are not linear but can be used to find solutions.

For example, the world we see is not constructed from one smooth surface, but is better modeled as a set of surfaces separated by discontinuities. An energy function can be constructed that explicitly models these discontinuities and their effects on the interpolated values. These are sometimes called *“weak membrane”* models.

The energy function in this case is no longer quadratic, and the update rule computes more than a weight sum.

Example: Reconstructing a smooth line, interpolation using smoothness

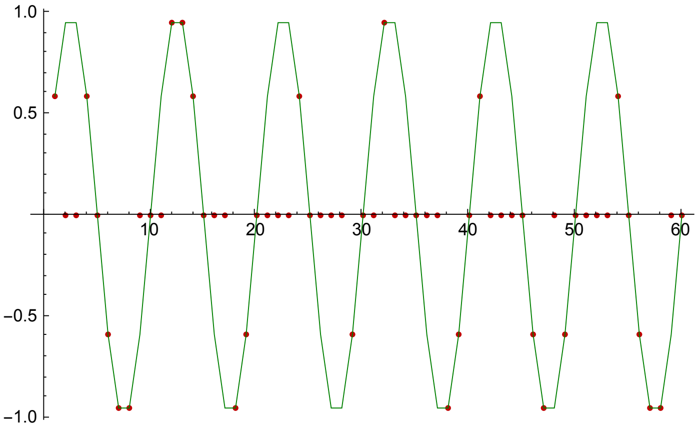
First-order smoothness

A generative model: Sine wave with random missing data points

Suppose we have sampled a function at a discrete random set of points **xs**. Multiplying the sine function by the vector **xs** picks out the values at the sample points at each location of the vector where **xs** is one, and sets the others to zero.

```
size = 60; xs = Table[RandomInteger[1], {i, 1, size}];
data = Table[N[Sin[2 π j/10] xs⟦j⟧], {j, 1, size}];
g3 = ListPlot[Table[N[Sin[2 π j/10]], {j, 1, size}], Joined → True,
   DisplayFunction → Identity, PlotStyle → {RGBColor[0, 0.5`, 0]}];
g2 = ListPlot[data, Joined → False, PlotStyle → {RGBColor[0.75`, 0.`, 0]},
   Prolog → AbsolutePointSize[5]];
```

```
Show[g2, g3]
```



We are not going to fit the dots on the abscissa. These are places where data is missing, analogous to the stereo interpolation regions with uniform regions where there are no points to match. The non-zero data are analogous to the disparity values from the two eyes.

We would like to find a smooth function, **f[]**, approximation to the non-zero data points, given the assumption that we don't know what the underlying function actually is.

We have one constraint already--the fidelity constraint that requires that the function **f** should be close to the non-zero data, **d**. We will use this to construct the "energy" term that measures how close they are in terms of the sum of the squared error.

We need another constraint--smoothness--to get the in-between points. There are many ways of doing this. If we had a priori knowledge that the underlying curve was periodic, we'd try fitting the data with some combination of sinusoids. Suppose we don't know this, but do have reason to believe that the underlying function is smooth in the sense of nearby points being close. As above, let's assume that the difference between nearby points should be small. That is, the sum of the squared errors, **f[i+1] - f[i]**, gives us the second part of our energy function.

Let's make up a small 8 element energy vector:

```
energyvector = 
Table[(f[i+1] - f[i])^2 + s[i] (d[i] - f[i])^2, 
             {i,1,8}];
```

```
energy = Sum[energyvector[[j]],{j,1,8}];
```

The **s[i]** term is the "filter" (the same as **xs** above) that only includes data points in the data part of the energy function. It is zero for i's where there are no data, and one for the points where there are data.

We would like to find the **f[]** that makes this energy a minimum. We can do this by calculating the derivative of the energy with respect to each component of **f**, and moving the state vector in a direction to minimize the energy--i.e. in the direction of the negative of the gradient.

It can be messy to keep track of all the indices in these derivatives, so let's let *Mathematica* calculate the derivative for f[3]. From this we can see the pattern for any index.

```
D[energy, f[3]]
```

```
2 (-f[2] + f[3]) - 2 (-f[3] + f[4]) - 2 (d[3] - f[3]) s[3]
```

```
Simplify[%]
```

```
-2 (f[2] + f[4] + d[3] s[3] - f[3] (2 + s[3]))
```

How to solve? Now we could generate the full set of derivatives, set them equal to zero and solve for **f**, using standard linear algebra to solve a set of linear equations. This will work because the energy function is quadratic in elements of **f**, and thus the derivatives are linear in **f**. The interpolation function is then a matrix operation on the data. As mentioned before, life won't always be that easy, and the situation often arises in which the energy function is not quadratic.

So the alternative, which can work in the non-linear case, is to use what we introduced above--gradient descent. We can do this by expressing the derivative in terms of two matrix operations: One on the function to be estimated, f, and one on the data.

Let's set up these two matrices, **Tm** and **Sm** such that the gradient of the energy is equal to:

Tm . f - Sm . f.

Sm will be our filter to exclude non-data points. **Tm** will express the "smoothness" constraint.

```
Sm = DiagonalMatrix[xs];
Tm = Table[0,{i,1,size},{j,1,size}];
For[i=1,i<=size,i++,Tm[[i,i]] = xs[[i]]+2];
For[i=1,i<size,i++, Tm[[i+1,i]] = -1];
For[i=1,i<size,i++, Tm[[i,i+1]] = -1];
```

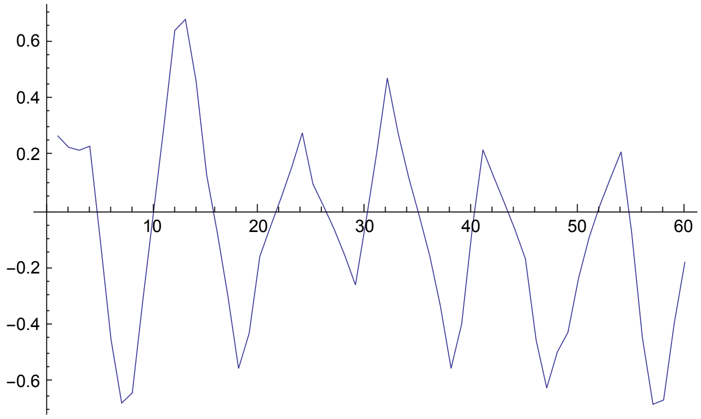
```
dt = 0.1;
Tf[f1_] := f1 - dt (Tm.f1 - Sm.data);
```

We will initialize the state vector to zero, and then run the network for 30 iterations:

```
f = Table[0,{i,1,size}];
result = Nest[Tf,f,30];
```

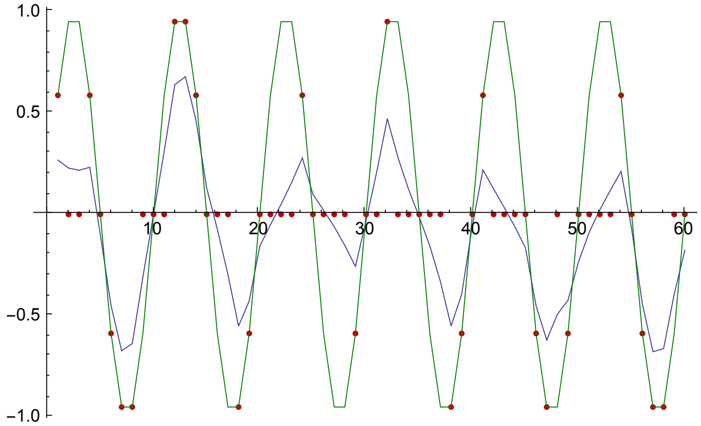
Here is our smoothed function:

```
g1 = ListPlot[result, Joined → True]
```



You can see below that the function was *not interpolated*--the fit doesn't not pass through the data points. If we wanted more fidelity to the data, we could control this by increasing the weight given to the data part of the energy term relative to the smoothness part.

```
Show[{g1, g2, g3}]
```



Exercise:

Decrease the parameter controlling smoothness to see if you get better fidelity.

Sculpting for interpolation using second order smoothness constraints

This section elaborates the energy function (and weight matrix) to require that both first and second order differences be small.

```
Clear[energy,energyvector,f,d,s,data];

energyvector =
Table[(f[i+1] - f[i])^2 + (f[i+2] - 2 f[i] + f[i+1])^2 + s[i] (d[i] - f[i])^2,{i,1,8}];

energy = Sum[energyvector[[j]],{j,1,8}];
```

By taking the derivative of the energy with respect to one of the interpolation depths, say f[3], we can see the pattern of the weights for the gradient descent update rule:

```
D[energy, f[3]]
```

```
2 (-f[2] + f[3]) + 2 (-2 f[1] + f[2] + f[3]) - 2 (-f[3] + f[4]) +
 2 (-2 f[2] + f[3] + f[4]) - 4 (-2 f[3] + f[4] + f[5]) - 2 (d[3] - f[3]) s[3]
```

```
Simplify[%]
```

```
-2 (2 f[1] + 2 f[2] - 8 f[3] + 2 f[4] + 2 f[5] + d[3] s[3] - f[3] s[3])
```

Now we simulate the sampled data, and then set up the weight matrix:

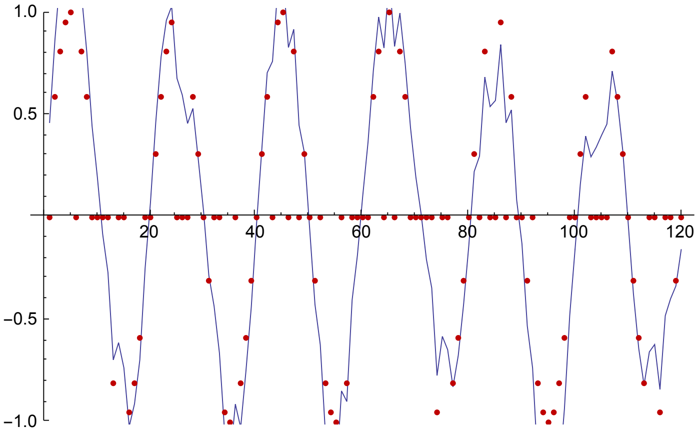
```
size = 120; xs = Table[RandomInteger[1], {i, 1, size}];
data = Table[N[Sin[2 π j/20] xs⟦j⟧], {j, 1, size}];
Sm = DiagonalMatrix[xs]; Tm = Table[0, {i, 1, size}, {j, 1, size}];
For[i = 1, i ≤ size, i++, Tm⟦i, i⟧ = xs⟦i⟧ + 8]; For[i = 1, i < size, i++, Tm⟦i + 1, i⟧ = -2];
For[i = 1, i < size, i++, Tm⟦i, i + 1⟧ = -2]; For[i = 1, i < size - 1, i++, Tm⟦i + 2, i⟧ = -2];
For[i = 1, i < size - 1, i++, Tm⟦i, i + 2⟧ = -2];
```

We will give the smoothness term a little more weight relative to the fidelity term:

```
dt = 0.1;
λ=.25;
Tf[f1_] := f1 - dt (λ*Tm.f1 - Sm.data);
```

```
f = Table[0,{i,1,size}];
result = Nest[Tf,f,30];
```

```
g1 = ListPlot[result, Joined → True, PlotRange → {-1, 1}, DisplayFunction → Identity];
g2 = ListPlot[data, Joined → False, PlotRange → {-1, 1},
  PlotStyle → {RGBColor[0.75`, 0.`, 0]}, Prolog → AbsolutePointSize[5],
  DisplayFunction → Identity]; Show[g1, g2, DisplayFunction → $DisplayFunction]
```



Try different values of the smoothing weight

Example: low-level vision “modules”

In 1985, Poggio, Torre and Koch showed that solutions to many of computational problems of early vision could be formulated in terms of maximum a posteriori estimates of scene attributes if the generative model could be described as a matrix multiplication, where the image I is matrix mapping of a scene vector S:

$$I = \mathbf{A}S$$

$$E = (I - \mathbf{A}S)^T (I - \mathbf{A}S) + \lambda S^T \mathbf{B} S$$

Then a solution corresponded to minimizing a cost function E, that simultaneously tries to minimize the cost due to reconstructing the image from the current hypothesis S, and a prior "smoothness" constraint on S. λ is a (often free) parameter that determines the balance between the two terms. If there is reason to trust the data, then λ is small; but if the data is unreliable, then more emphasis should be placed on the prior, thus λ should be bigger.
For example, S could correspond to representations of shape, stereo, edges, or motion field, and smoothness be modeled in terms of nth order derivatives, approximated by finite differences in matrix B.

The Bayesian interpretation comes from multivariate gaussian assumptions on the generative model:

$$p(I \mid S) = k \times \exp\left[-\frac{1}{2\sigma_n^2} (I - \mathbf{A}S)^T (I - \mathbf{A}S) \right]$$

$$p(S) = k' \times \exp\left[-\frac{1}{2\sigma_s^2} S^T \mathbf{B} S \right]$$

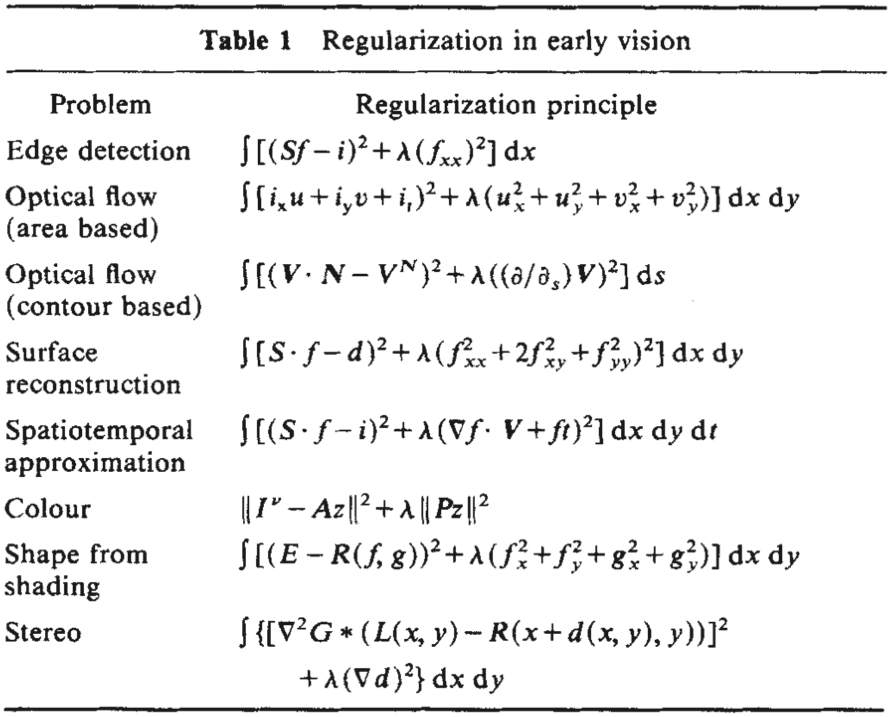


Table 1 Regularization in early vision

Problem	Regularization principle
Edge detection	$\int [(Sf - i)^2 + \lambda (f_{xx})^2]\, \mathrm{d}x$
Optical flow (area based)	$\int [i_x u + i_y v + i_t)^2 + \lambda (u_x^2 + u_y^2 + v_x^2 + v_y^2)]\, \mathrm{d}x\, \mathrm{d}y$
Optical flow (contour based)	$\int [(\boldsymbol{V} \cdot \boldsymbol{N} - V^N)^2 + \lambda ((\partial/\partial_s) \boldsymbol{V})^2]\, \mathrm{d}s$
Surface reconstruction	$\int [S \cdot f - d)^2 + \lambda (f_{xx}^2 + 2f_{xy}^2 + f_{yy}^2)^2]\, \mathrm{d}x\, \mathrm{d}y$
Spatiotemporal approximation	$\int [(S \cdot f - i)^2 + \lambda (\nabla f \cdot \boldsymbol{V} + ft)^2]\, \mathrm{d}x\, \mathrm{d}y\, \mathrm{d}t$
Colour	$\lVert I^\nu - Az \rVert^2 + \lambda \lVert Pz \rVert^2$
Shape from shading	$\int [(E - R(f, g))^2 + \lambda (f_x^2 + f_y^2 + g_x^2 + g_y^2)]\, \mathrm{d}x\, \mathrm{d}y$
Stereo	$\int \{[\nabla^2 G * (L(x, y) - R(x + d(x, y), y))]^2$ $+ \lambda (\nabla d)^2\}\, \mathrm{d}x\, \mathrm{d}y$

From Poggio, Torre & Koch, 1985

The notation above is: $\frac{\partial f}{\partial x} \longleftrightarrow f_x$.

In edge detection, noise can create spurious edges. The way to deal with that is by blurring the image and then applying a spatial derivative. The above constraint says to assume there is an underlying “image”, f, that has the “true perfectly sharp edges”, which have gotten smoothed out by filter S to produce i; and because there is noise, we should find the find f that is consistent with this generative assumption, but restricted to the f which is most smooth. This latter constraint is measured by the square of the second spatial derivative of f: f_{xx}.

For optic flow (area based), the gradient constraint is what we have seen before: $i_x u + i_y v + i_t = 0$. The smoothness constraint here is expressed as:

$$u_x^2 + u_y^2 + v_x^2 + v_y^2,$$

which discourages rapid spatial changes in the optic flow vectors.

Yuille, A. L., & Grzywacz, N. M. (1988) proposed including all derivatives, which would include this, as well as the “slow” and “smooth” assumptions in the work by Weiss et al. (2002).

Above we looked at the one-dimensional analog of surface reconstruction where we contrasted the cost function minimization, similar to that above, with a probablistic formulation solved through belief-propagation.

A key point is that the maximum a posteriori solution based on equations 1 and 2 above is linear. Thus given the “right” representation, a broad range of estimation problems can be modeled as simple linear networks. However, we noted early on that there are also severe limitations to linear estimation.

The bias/variance dilemma

The problem we now consider is how to choose the function that both remembers the relationship between **x** and **y**, and generalizes given new values of **x**. At first one might think that it should be as general as possible to allow all kinds of mappings.

For example, if one is fitting a apparently complicated curve, you might wish to use a very high-order polynomial, or a back-prop network with lots of hidden units. There is a drawback, however, to the flexibility afforded by extra degrees of freedom in fitting the data. We can get drastically different fits for different sets of data that are randomly drawn from the same underlying process. The fact that we get different fit parameters (e.g. slope of a regression line) each time means that although we may exactly fit the data each time, we introduce variation between the average fit (over all data sets) and the fits over the ensemble of data sets.

We could get around this problem with a huge amount of data, but the problem is that the amount of required data can grow exponentially with the order of the fit--an example of the so-called "curse of dimensionality".

On the other hand, if the function is restrictive, (e.g. straight lines through the origin), then we will get similar fits for different data sets, because all we have to adjust is one parameter--the slope. The problem here, is that the fit is only good if the underlying process is in fact a straight line through the origin. If it isn't a straight line for instance, there will be a fixed error or **bias** that will never go away, no matter how much data we collect.

Statisticians refer to the trade-off between simple but biased fits, and complex but data-dependent variation, as the *bias/variance* dilemma.

To sum up, lots of parameter flexibility (or lots of hidden units) has the benefit of fitting anything, but at the cost of sensitivity to variability in the data set--there is *variance* introduced by the fits found over multiple training sets (e.g. of a small fixed size).

A fit with very few parameters is not as sensitive to the inevitable variability in the training set, but can give large constant errors or *bias* if the data do not match the underyling model.

There is no general way of getting around this problem, and neural networks are no exception. We generalized linear regression to non-linear fits using error back-propagation. Because back-propagation models can have lots of hidden layers with many units and weights, they form a class of very flexible approximators and can fit almost any function. But these models can show high variability in their fits from one data set to the next, even when the data comes from the same underlying process. Lots of hidden units can mean low bias, but at a high cost in variance.

Demonstration of bias/variance for regression

Suppose we have an unknown, underlying generative model given by: p(y|x) and p(x). From this we obtain a set of samples {x_i, y_i}, i=1...N.

Keep in mind that a set of samples is not in general representive of the whole distribution p(x,y). Later samples may suggest a different model.

We posit some estimator to fit the data: y_i ~ f(x_i). This function f has some associated parameters that need to be estimated. And these parameters get re-estimated each time with get a new set of samples. So there is variation in f.

In general, there will be some cost assigned to errors in f's ability to predict the y's. E.g. the expected value of the squared difference between the fits and the true expected value of y, call it $\hat{y}$. It is insightful to write this cost as the sum of two terms:

$$E[(f-\hat{y})^2] = (E[f]-\hat{y})^2 + E[(f-E[f])^2]$$

The first term on the right is the bias (squared), and the second term the variance. The bias is the "constant error" which tells us how far off we'll be--could be non-zero even with an unlimited supply of data. The second term, the "variance", tells us how much variation we have in the ensemble of fits f about the average of all the fits, E[f].

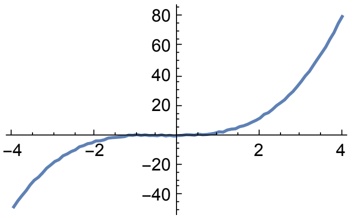
Let's demonstrate the effects of the bias/variance trade-off by estimating the values of $\tilde{f}$ and $\hat{y}$, given a generative model.

Mathematica's regression package

Go to Help, and find the Linear Regression package. Look up **LinearModelFit[]**. We are going to use Regress as our learning model. We could have used our errro-back prop network, or other learning algorithms that produce a set of fit parameters. The principles would be the same.

```
In[94]:= ff[x_, α_] := α.{1, x, x^2, x^3} + RandomReal[];
         α = {0, 0, 1, 1};
         xd = Table[{x, ff[x, α]}, {x, -4, 4, 0.1}];
         ListPlot[xd, AxesOrigin → {0, 0}, Joined → True, ImageSize → Small]
         lm = LinearModelFit[xd, {1, x, x^2, x^3}, x];
         lm[{"ParameterTable", "RSquared"}]
```

Out[97]=



```
              Estimate      Standard Error  t-Statistic  P-Value
          1   0.411398      0.0503723       8.16715      4.84557×10^-12
Out[99]= {x   -0.00919821   0.0359216       -0.256063    0.798585            , 0.999867}
          x^2 1.00896       0.00686874      146.892      4.67141×10^-96
          x^3 0.999897      0.00334682      298.76       9.39969×10^-120
```

Learning from one data set

Underlying model space

```
In[198]:= Clear[ff];
          ff[x_, α_] := α.{1, x, x^2, x^3};
```

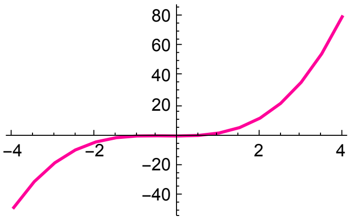
Generative data process: true plus some noise

```
In[200]:= noise = 15;
          ffn[x_, α_] := ff[x, α] + 1.5 * RandomReal[{-noise, noise}];
```

Choose domain and true model parameters. Calculate a set of samples from a true (noise free) model ffp, evaluated at xp.

```
In[222]:= xp = Table[x, {x, -4, 4, 0.5}];
          α = {0, 0, 1, 1};
          ffp = (ff[#1, α] &) /@ xp;
          gffp = ListPlot[Transpose[{xp, ffp}],
            PlotStyle → {PointSize[0.02], Hue[0.9]}, Joined → True, ImageSize → Small]
```

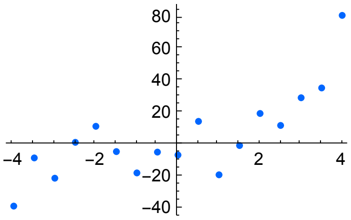
Out[225]=



Run one experiment to collect y values for data process:

```
In[226]:= y = (ffn[#1, α] &) /@ xp;
          gy = ListPlot[Transpose[{xp, y}],
            PlotStyle → {PointSize[0.02], Hue[0.6]}, ImageSize → Small]
```

Out[227]=



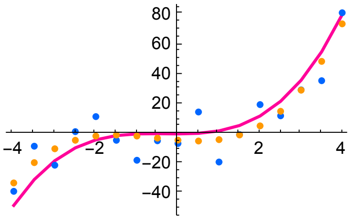
Estimate model parameters using polynomial regression. Return model parameters, and predicted responses, fsquiggle

```
In[228]:= xd = Table[{x, ff[x, α]}, {x, -4, 4, 0.1}];
```

```
In[229]:= αD = LinearModelFit[Transpose[{xp, y}], {1, x, x^2, x^3}, x];
          fsquiggle = Table[{x, αD[x]}, {x, -4, 4, 0.5}];
          gfsquiggle =
            ListPlot[fsquiggle, PlotStyle → {PointSize[0.02], Hue[0.1]}, ImageSize → Small];
```

```
In[232]:= Show[gffp, gy, gfsquiggle]
```

Out[232]=



Repeat the above, and notice how the model parameters and the fit changes. Try changing the basis functions used for fitting.

Demonstrating the key idea

We'd like some idea of how learning generalizes depending on model complexity

The "right" model

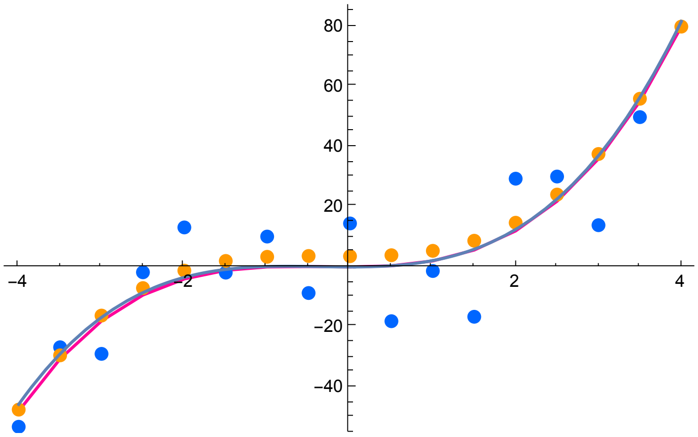
First, assume by some incredibly lucky guess, we've chosen the right model {x^2, x^3}, and want to find the parameters.

```
In[152]:= y = ffn[#1, α] & /@ xp;
```

```
In[153]:= αD2 = LinearModelFit[Transpose[{xp, y}], {x^2, x^3}, x];
          fsquiggle = Table[{x, αD2[x]}, {x, -4, 4, 0.5}];
          gfsquiggle =
            ListPlot[fsquiggle, PlotStyle → {PointSize[0.02], Hue[0.1]}, ImageSize → Small];
```

```
In[156]:= gffp = ListPlot[Transpose[{xp, ffp}],
             PlotStyle → {PointSize[0.02], Hue[0.9]}, Joined → True];
           gsmooth = Plot[Fit[Transpose[{xp, y}], {x^2, x^3}, x] /. x → x2, {x2, -4, 4}];
           Show[gffp, gy, gfsquiggle, gsmooth]
```

Out[158]=



A simple, but wrong model

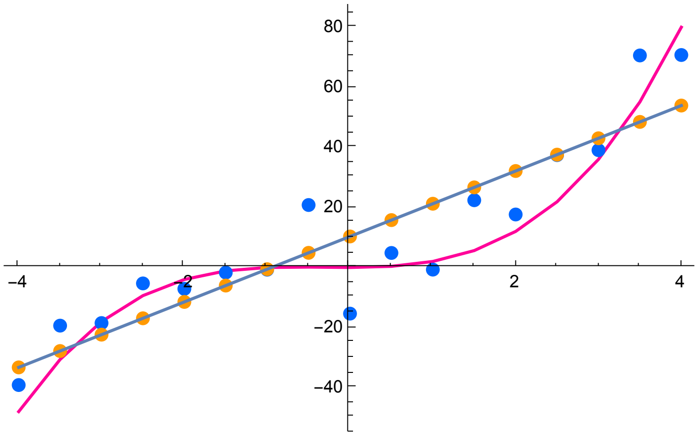
But now suppose that we are trying to fit the data with an inappropriate model. In particular, suppose that it is weak, say a linear model:

```
In[159]:= y = ffn[#1, α] & /@ xp;

           αD3 = LinearModelFit[Transpose[{xp, y}], {1, x}, x];
           fsquiggle = Table[{x, αD3[x]}, {x, -4, 4, 0.5}];
           gfsquiggle =
             ListPlot[fsquiggle, PlotStyle → {PointSize[0.02], Hue[0.1]}, ImageSize → Small];

In[163]:= gffp = ListPlot[Transpose[{xp, ffp}],
             PlotStyle → {PointSize[0.02], Hue[0.9]}, Joined → True];
           gy = ListPlot[Transpose[{xp, y}], PlotStyle → {PointSize[0.02], Hue[0.6]}];
           gsmooth = Plot[Fit[Transpose[{xp, y}], {1, x}, x] /. x → x2, {x2, -4, 4}];
           Show[gffp, gy, gfsquiggle, gsmooth]
```

Out[166]=



The bias is the squared difference between the average y values (blue) and the model fits (orange). The variance is the squared difference between the predicted responses (blue) and the true (red line). So we can see that the bias is high. The variance (represented by the square root of the variance, black points in the graph) is not zero. But how does it compare with a model with lots of parameters?

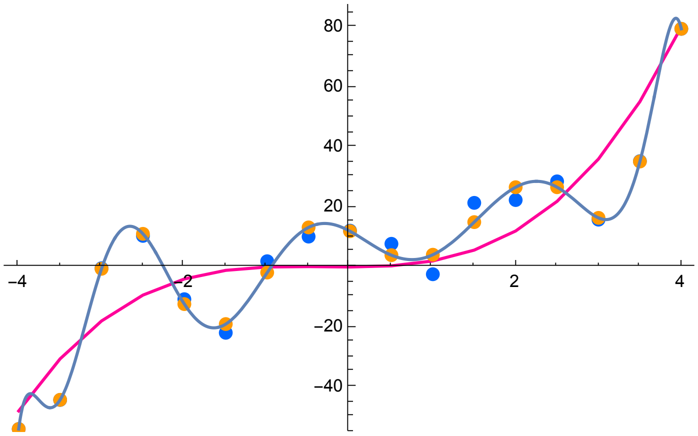
A complex model, with too many parameters

Now let's try over-fitting. (Analogous to having lots of hidden units and/or layers in a non-linear feedforward network).

```
In[233]:= y = ffn[#1, α] & /@ xp;
          αD4 = LinearModelFit[Transpose[{xp, y}],
             {1, x, x^2, x^3, x^4, x^5, x^6, x^7, x^8, x^9, x^10, x^11, x^12}, x];
          fsquiggle = Table[{x, αD4[x]}, {x, -4, 4, 0.5}];
          gfsquiggle =
            ListPlot[fsquiggle, PlotStyle → {PointSize[0.02], Hue[0.1]}, ImageSize → Small];
```

```
In[237]:= gffp = ListPlot[Transpose[{xp, ffp}],
             PlotStyle → {PointSize[0.02], Hue[0.9]}, Joined → True];
          gsmooth = Plot[Fit[Transpose[{xp, y}], {1, x, x^2, x^3, x^4, x^5, x^6,
                 x^7, x^8, x^9, x^10, x^11, x^12}, x] /. x → x2, {x2, -4, 4}];
          gy = ListPlot[Transpose[{xp, y}], PlotStyle → {PointSize[0.02], Hue[0.6]}];
          Show[gffp, gy, gfsquiggle, gsmooth]
```

Out[240]=



Note how the bias (discrepancy between the orange and blue) is lower than with too few parameters. But we have higher variance than with the "right model" family.

Now try comparing learning over several data sets to observe the increased variance in the fits.

Dealing with over-fitting

Drop-out & error backpropagation in multi-layer networks

See: http://arxiv.org/pdf/1207.0580v1.pdf

Later we’ll talk about inference in hierarchical structured graphs.

Bayesian model selection

MacKay (1992)

Cross-validation

References

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. (Amazon.com)
Hertz, J., Krogh, A., & Palmer, R. G. (1991). *Introduction to the theory of neural computation* (Santa Fe Institute Studies in the Sciences of Complexity ed. Vol. Lecture Notes Volume 1). Reading, MA: Addison-Wesley Publishing Company.
Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. R. (2012). Improving neural networks by preventing co-adaptation of feature detectors. arXiv Preprint arXiv:1207.0580.
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.
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.
Kersten, D. J. (1991). Transparency and the Cooperative Computation of Scene Attributes. In M. Landy, & A. Movshon (Ed.), Computational Models of Visual Processing (pp. 209-228). Cambridge, Massachusetts: M.I.T. Press.
Kersten, D., & Madarasmi, S. (1995). The Visual Perception of Surfaces, their Properties, and Relationships. In I. J. Cox, P. Hansen, & B. Julesz (Eds.), Partitioning Data Sets: With applications to psychology, vision and target tracking, (pp. 373-389): American Mathematical Society.
Koch, C., Marroquin, J., & Yuille, A. (1986). Analog "neuronal" networks in early vision. Proc. Natl. Acad. Sci. USA, 83, 4263- 4267.) See also, Hertz et al., page 82-87.
Nakayama, K., & Shimojo, S. (1992). Experiencing and perceiving visual surfaces. Science, 257, 1357-1363.
MacKay, D. J. C. (1992). Bayesian interpolation. *Neural Computation, 4*(3), 415-447.
Weiss, Y., Simoncelli, E. P., & Adelson, E. H. (2002). Motion illusions as optimal percepts. Nature Neuroscience, 5(6), 598–604. doi:10.1038/nn858

© 1998, 2001, 2009, 2011, 2014 Daniel Kersten, Computational Vision Lab, Department of Psychology, University of Minnesota.