Lecture 5:

Regression II

Sam Roweis

October 9, 2002

Maximum Likelihood

- Maximum likelihood asks the question: for which setting of the weights is the data we saw most likely?
- To answer this, it assumes that the training data are iid, computes the log likelihood and forms a function $\ell(\mathbf{w})$ which depends on the fixed training set we saw and on the argument $\mathbf{w}$:

$$\begin{aligned}\ell(\mathbf{w}) &= \log p(y_1, \mathbf{x}_1, y_2, \mathbf{x}_2, \ldots, y_n, \mathbf{x}_n | \mathbf{w}, \sigma^2) \\ &= \log \prod_b p(y_n, \mathbf{x}_n | \mathbf{w}, \sigma^2) && \text{since iid} \\ &= \sum_n \log p(y_n, \mathbf{x}_n | \mathbf{w}, \sigma^2) && \text{since } \log \prod = \sum \log \\ &= -\frac{1}{2\sigma^2} \sum_n (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \text{const}\end{aligned}$$

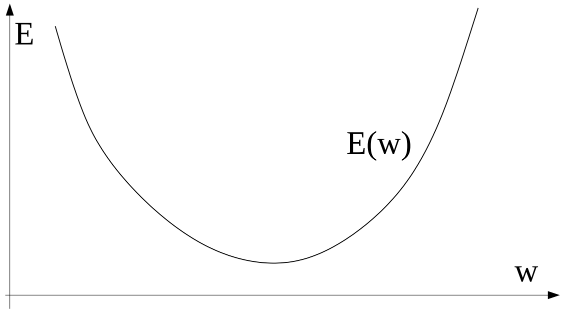
- Maximizing likelihood is equivalent to minimizing sum squared error, if the noise model is Gaussian and the datapoints are iid.

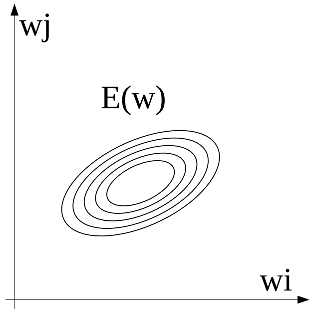
Reminder: regression and probabilities

- Assume data is of form $p(y|\mathbf{x}) = \mathcal{N}(y; \mathbf{w}^\top \mathbf{x}, \sigma^2)$.
- We see some data $(y_1, \mathbf{x}_1)(y_2, \mathbf{x}_2) \ldots (y_n, \mathbf{x}_n)$ and we want to infer something about the weights.
- Ideally, we would be Bayesian and use Bayes rule to compute $p(\mathbf{w}|y_1, \mathbf{x}_1, y_2, \mathbf{x}_2, \ldots, y_n, \mathbf{x}_n, \sigma^2)$.
- But so far we have been weak, and we have just been using *maximum likelihood*.
- Actually, we haven't been quite so stupid: we've been using *maximum penalized likelihood* to include regularization effects like shrinkage or maximum number of input features or smoothing.

Error Surfaces and Weight Space

- Make a new "error function" $E(\mathbf{w}) = -\ell(\mathbf{w})$ (to be minimized).
- Think not of input space but of weight space.
  Consider a fixed training set.
  Now, for each setting of the weights there is some error:
  this defines an *error surface*.
- Learning == descending the error surface.





Quadratic Errors and IID data

- Notice: If the data are IID, the error function E is a sum of error functions E_n, one per data point.
- If the noise model is Gaussian, each of these E_n is an upside-down parabola (called a "quadratic bowl" in higher dimensions.
- Fact: sum of parabolas (quadratics) is another parabola (quadratic).
- So the overall error surface is just a quadratic bowl.
- Fact: it is easy to find the minimum of a quadratic bowl:

$$E(w) = a + bw + cw^2 \quad \Rightarrow \quad w^* = -b/2c$$
$$E(\mathbf{w}) = a + \mathbf{b}^\top \mathbf{w} + \mathbf{w}^\top \mathbf{C}\mathbf{w} \quad \Rightarrow \quad \mathbf{w}^* = -\frac{1}{2}\mathbf{C}^{-1}\mathbf{b}$$

- Convince yourself that for linear regression: $\mathbf{C} = \mathbf{X}\mathbf{X}^\top$ and $\mathbf{b} = -2\mathbf{X}\mathbf{y}^\top$.

Adaptive Basis Regression

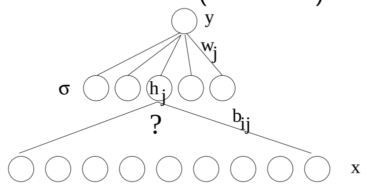
- Last lecture we considered the generalized linear model

$$y = \sum_j w_j h_j(\mathbf{x})$$

- Now: want to *learn* both the basis functions h_j and the weights w_j.
- Of course, for a single output we could just absorb w_j into h_j but for multiple outputs this is not possible.
- One way to learn is to select from a large set of possible h_j. But we want to consider a fixed number of *adaptive* basis functions.

Credit Assignment Problem

- Imagine that the basis functions were of the form $h_j(\mathbf{x}) = \sigma(\mathbf{b}_j^\top \mathbf{x})$.
- If we knew the desired values h_j^* of $h_j(\mathbf{x})$ we could find $\mathbf{b}_j$ by doing linear regression from $\mathbf{x}$ to $\sigma^{-1}(h_j^*)$.
- Problem: these values are unknown ("hidden"). Must find them.



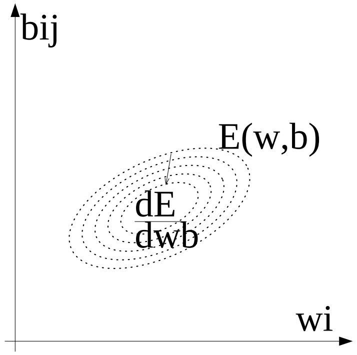
- Basic idea: try changing the basis functions' values and see how the final output y (and thus the error) is affected.
- If we change several basis functions and the output gets better, which one was responsible? This is the *credit assignment problem*.
- Solution: only change one at a time!

Partial Derivatives of Error

- Question: if we wiggle $h_j(\mathbf{x})$ *and keep everything else the same*, does the error get better or worse?
- Luckily, calculus has an answer to exactly this question: $\frac{\partial E}{\partial h_j}$.
- Plan: use a differentiable cost function E on the outputs and on the basis functions $h_j(\mathbf{x})$. Now compute the *partial derivative* of each parameter with respect to this error $\frac{\partial E}{\partial h_j}$.
- Use the *chain rule* to compute $\frac{\partial E}{\partial b_{ij}} = \frac{\partial E}{\partial h_j}\frac{\partial h_j}{\partial b_{ij}}$
- The vector of partial derivatives is called the *gradient* of the error. It points in the direction of steepest error descent in weight space.
- The trick becomes computing these derivatives efficiently.

Weight Space Again

- Think in weight space again. But now the weights are both w_j and basis weights b_{ij}.
- Adaptive basis regularization is all about descending the error surface in this extended space by following the gradient.



Artificial Neural Networks

- Lots of hype and relations to biology, but for our purposes, ANNs are just adaptive basis regression machines of the form:
$$y_k = \sum_j w_{kj}\sigma(\mathbf{b}_j^\top \mathbf{x}) = \mathbf{w}_k \mathbf{h}$$
where $h_j = \sigma(\mathbf{b}_j^\top \mathbf{x})$ are known as the *hidden unit activations*; y_k are the output units and x_i are the input units.
- The nonlinear scalar function σ is called an *activation function*. We usually use *invertible* and *differentiable* activation functions.
- If the activation function is *linear*, the whole network reduces* to a linear network: equivalent to linear regression.
[Only if there are at least as many hiddens as inputs and outputs.]
- It is often a good idea to add "skip weights" directly connecting inputs to outputs to take care of this linear component directly.
- Sometimes we put an activation on the outputs also: $y_k = \sigma(\mathbf{w}_k \mathbf{h})$.

Gradient Descent

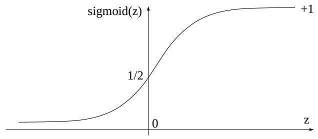
- Once we have the gradient of our error function, how do we minimize the weights? Follow it! But not too fast...
- **Algorithm Gradient Descent**

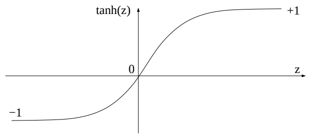
```
w ← GradientDescent(w0,x-train,y-train)   {
step=median(abs(w0(:)))/100;  errold=Inf;  grad=0;
while(step>0)
  w = w0 - step*grad;
  (err,grad) ← errorGradient(w,x-train,y-train)
  if(err>=errold)
    step=step/2; grad=gradold;
  else
    step=step*1.01; errold=err; w0=w; gradold=grad;
  end
end
                                                    }
```

- This algorithm only finds a *local* minimum of the cost.
- This is *batch* gd, but mini-batch or online are better.

Common Activation Functions

- Two common activation functions: *sigmoid* and *hyperbolic tangent*
$$\text{sigmoid(z)} = \frac{1}{1+\exp(-z)} \qquad \tanh(\text{z}) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$$

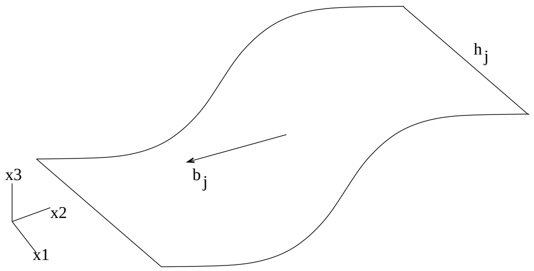




- For small weights, these functions will be operating near zero and their behaviour will be almost linear.
- Thus, for small weights, the network behaves essentially linearly.
- In general we want a saturating activation function (why?).

Geometry of ANNs

- ANNs can be thought of as generalized linear models, where the basis functions are sigmoidal "cliffs".
- The cliff direction is determined by the input-to-hidden weights, and these bases are combined by the hidden-to-output weights.
- We include bias units of course, and these set where the cliff is positioned relative to the origin.



Backpropagation: error gradient in ANNs

- Let a_{ji} be the i^{th} input to unit j.
  Let w_{ji} be the weight associated with i^{th} input to j.
  Let $q_j = \sum_i w_{ji} a_{ji}$ be the net input to unit j.
  Let o_j be the output of unit j.
  Let F_j be the set of units including j as input.
- For input units i, $o_i = x_i$ and $q_i, a_{i\cdot}, w_{i\cdot}$ are undefined.
- For output units k, $o_k = q_k$ if no final activation; F_k is undefined.

Multi-layer nets and representational capacity

- We have considered only one hidden layer but we can have more.
- However, there is a deep result, (Kolmogorov's theorem) which tells us that *any* real valued function of K real inputs can be created by composing addition and single input functions.
  [This answered Hilbert's 13th problem in the affirmative, which asked if specific 3-variable functions could be written as compositions of continuous functions of two or less variables.]
- In other words, there are *no* true functions of two variables.
  e.g. $xy = \exp(\log x + \log y) \quad \log xy = \log x + \log y \quad \ldots$
- This tells us that a one-hidden layer net will always do the job, in theory. But it may take *a lot* of hidden units.
- Want to know more?

V. Kurkova, "Kolmogorov's Theorem Is Relevant", Neural Computation, 1991, Vol. 3, pp. 617–622.

F. Girosi and T. Poggio. Kolmogorov's theorem is irrelevant. Neural Computation, 1(4):465–469, 1989.

Derivative of Activation

- For a sigmoid activation, $o_j = \sigma(q_j)$ and
$$\frac{\partial o_j}{\partial q_j} = o_j(1 - o_j)$$
- For a tanh activation, $o_j = \sigma(q_j)$ and
$$\frac{\partial o_j}{\partial q_j} = (1 - o_j^2)$$

Derivative of Output Error

- For squared error:

$$E = \sum_n E_n$$
$$E_n = \sum_k (y_{nk} - o_{nk})^2$$
$$\frac{\partial E_n}{\partial o_k} = -2(y_k - o_k)$$

- We dropped the n from y_k and o_k for ease.

Output Units

- For output unit(s) o_k:

$$\begin{aligned}\frac{\partial E_n}{\partial q_k} &= \frac{\partial E_n}{\partial o_k}\frac{\partial o_k}{\partial q_k} \\ &= -2(y_k - o_k)o_k(1 - o_k) \\ \frac{\partial E_n}{\partial w_{kj}} &= -2(y_k - o_k)o_k(1 - o_k)a_{kj}\end{aligned}$$

- This is *with* activations on outputs.

Derivative with respect to a weight

- For any weight:

$$\begin{aligned}\frac{\partial E_n}{\partial w_{ji}} &= \frac{\partial E_n}{\partial q_j}\frac{\partial q_j}{\partial w_{ji}} \\ &= \frac{\partial E_n}{\partial q_j}a_{ji}\end{aligned}$$

- Let's assume squared error and sigmoid activations.

Hidden Units

- Let $\delta_j = \frac{\partial E_n}{\partial q_j}$ for all units.

$$\begin{aligned}\delta_j = \frac{\partial E_n}{\partial q_j} &= \sum_{i \in F_j} \frac{\partial E_n}{\partial q_i}\frac{\partial q_i}{\partial q_j} = \sum_{i \in F_j} \delta_i \frac{\partial q_i}{\partial q_j} \\ &= \sum_{i \in F_j} \delta_i \frac{\partial q_i}{\partial o_j}\frac{\partial o_j}{\partial q_j} \\ &= \frac{\partial o_j}{\partial q_j} \sum_{i \in F_j} \delta_i w_{ij} \\ &= o_j(1 - o_j) \sum_{i \in F_j} \delta_i w_{ij}\end{aligned}$$

Recursion

- The previous equations lead to a recursion for computing δs:
$$\delta_j = o_j(1 - o_j) \sum_{i \in F_j} \delta_i w_{ij}$$
- Given the δs we can compute the required gradients:
$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j a_{ji}$$

Backprop

- The final backprop procedure does this for each training case:
  1. Propagate the inputs forward through the network:
     inputs: $o_i \leftarrow x_i$
     hiddens & outputs: $o_j \leftarrow \sigma(\sum_i w_{ji} x_j)$
  2. Compute the output error $E_n = \sum_k (y_k - o_k)^2$ for this case.
  3. Propagate the deltas backwards through the network:
     outputs: $\delta_k \leftarrow -2(y_k - o_k) o_k (1 - o_k)$
     hiddens: $\delta_j \leftarrow o_j(1 - o_j) \sum_{i \in F_j} \delta_i w_{ij}$
  4. Compute the gradients: $\frac{\partial E_n}{\partial w_{ji}} = \delta_j a_{ji}$
- Batch: sum errors and gradients over all training cases.
  Mini-batch: sum errors and gradients over next n_{batch} cases.
  Online: return gradient and error for this one case.

Backprop is efficient

- What's so deep about the backprop algorithm?
  Isn't is just the chain rule applied to the ANN model?
- Yes, but it does all gradient computations in $O(|W|)$ rather than $O(|W|^2)$.
- If you just write down the derivatives, each one contains some forward activation terms. It takes $O(|W|)$ to compute the forward activations, and there are $|W|$ weights so the naive procedure is $O(|W|^2)$.
- Backprop shares the forward and backward calculations across the network and so is roughly $O(3|W|)$.

Regularlization

- Of course, we have to regularize ANNs just like all other models.
- Shrinkage is the standard approach: here it is called *weight decay*.
- Add a term $\lambda \sum_{ij} w_{ij}^2$ to the cost function.
- This adds $-2\lambda w_{ij}$ to the gradient of each weight.
  Makes large weights get smaller.
- Set λ by (cross-)validation.
- Can also start with small weights and use *early stopping* by stopping training when (cross-)validation error starts to increase.
  This effectively keeps weights small.

Radial Basis Networks

- Instead of sigmoidal hidden units, we can use Gaussian shaped "bumps" as our basis functions:

$$h_j(\mathbf{x}) = \exp\left[-\frac{1}{2\sigma^2}\|\mathbf{x} - \mathbf{z}_j\|^2\right]$$

- The goal now is to *learn* the bump centres $\mathbf{z}_j$.
- How? You got it! Take the derivative using backprop.
- Now, the activation function derivatives will be different, but the basic algorithm is exactly the same.
- Of course, we still have to set σ^2 somehow.