Lecture 6:

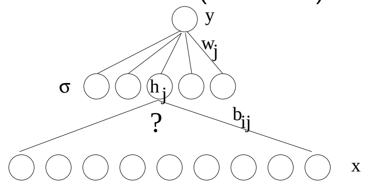
Regression II: Adaptive Basis Networks & Supervised Mixtures

Sam Roweis

October 14, 2003

Credit Assignment Problem

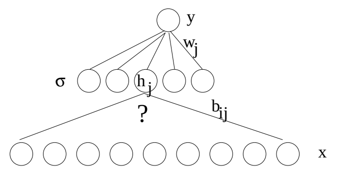
- Imagine that the basis functions were of the form $h_j(\mathbf{x}) = \sigma(\mathbf{b}_j^\top \mathbf{x})$.
- If we knew the desired values h_j^* of $h_j(\mathbf{x})$ we could find $\mathbf{b}_j$ by doing linear regression from $\mathbf{x}$ to $\sigma^{-1}(h_j^*)$.
- Problem: these values are unknown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

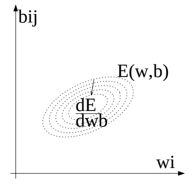
Adaptive Basis Regression

- Previously we considered the generalized linear model
$$y = \sum_j w_j h_j(\mathbf{x})$$
- Originally, the $h_j(\cdot)$ were fixed and we were finding w_j. Now we want to learn *both* the basis functions h_j and 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.



Partial Derivatives and Weight Space Again

- 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 inside the basis functions $h_j(\mathbf{x})$. Now compute the *partial derivative* of each parameter with respect to the error: $\frac{\partial E}{\partial h_j}$.
- Use the *chain rule* to compute gradient components $\frac{\partial E}{\partial b_{ij}} = \frac{\partial E}{\partial h_j}\frac{\partial h_j}{\partial b_{ij}}$



Adaptive basis regularization is all about descending the error surface in (w,b) weight space by following the gradient.

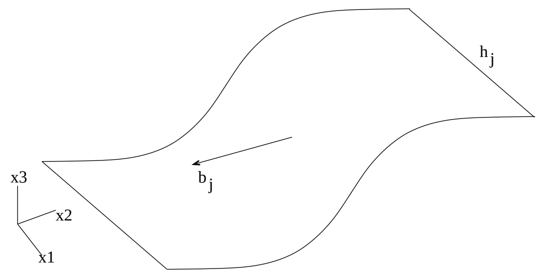
- The trick becomes computing these derivatives efficiently.

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: $y_k = \sigma(\mathbf{w}_k \mathbf{h})$.

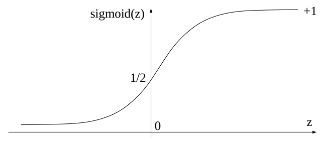
Geometry of ANNs

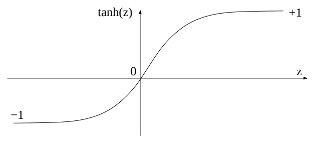
- ANNs can be thought of as generalized linear models, where the basis functions are sigmoidal shaped "cliffs".
- The cliff direction is determined by the input-to-hidden weights, and these bases are combined by the hidden-to-output weights.
- We include bias units of course, and these set where the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



Common Activation Functions

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





- 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?).
- Neural network models with these activations are often called *multi-layer perceptrons* (MLPs), in analogy with the classic perceptron that used a hard-threshold activation.

Backpropagation: error gradient in ANNs

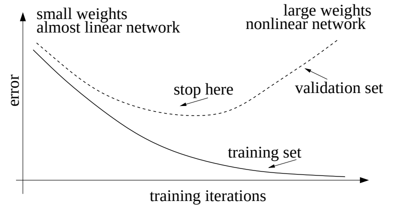
- In layered networks (even with more than two layers), there is an elegant, efficient recursive algorithm for computing the gradients of any differentiable error function with respect to each of the weights using message passing up and down the network.
- The calculations depend on which error function is used at output (e.g. squared) and on which nonlinear activation function is used at the hidden units (e.g. tanh).
- Batch training: sum gradients over all data cases (examples) in the training set, then update weights, then repeat.
Mini-batch: sum gradients over next n_{batch} cases, update.
Online training: update based on gradient for one case at a time.

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 these terms, and there are $|W|$ weights so the naive procedure is $O(|W|^2)$.]
- Backprop uses messages and caching to share forward and backward calculations across the network and so is roughly $O(3|W|)$.
  This is the same time complexity as to make a single forward pass!
- The original backprop paper is:
  Learning representation by backpropagating errors,
  Rumelhart, Hinton and Williams; (Nature, 1986)

Network Regularlization

- We have to regularize ANNs and RBFs 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 (hence the name).
- Set λ by (cross-)validation.
- Can also start with small weights (and thus a linear network) and stop training when the (cross-)validation error starts to increase.
  This is called *early stopping*, and effectively keeps weights small.

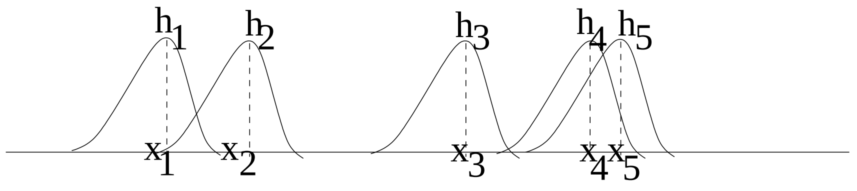


Radial Basis Networks

- 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.



Multi-layer nets and representational capacity

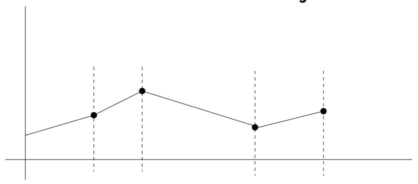
- We have considered only one hidden layer but we can have more.
- However, there is a deep result, (Kolmogorov's theorem) which tells us that *any* real valued function of K real inputs can be created by composing addition and single input functions.
  [This answered Hilbert's 13th problem in the affirmative, 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?

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

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

Divide and Conquer

- One important idea in regression, similar to that in decision trees for classification, is to divide the problem into many regions so that within each region a simple model (e.g. linear) will suffice.
- If we divide into fixed regions, we have a general basis model (e.g. splines, RBF's). But what if we want to adjust the boundaries also?



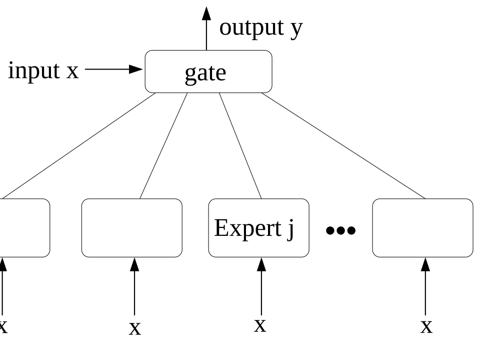
- Problem: since the cost function is *piecewise constant* with respect to these boundaries, taking derivatives will not work.
- Solution: use "soft" splits of the data space, rather than hard splits.

How the Experts and Gate Interact

- The gate must look at the input and the proposed outputs of each expert to come up with an overall output.
  (Remember that the true output is unknown at test time!)
- The gate gives each expert a score, representing how well it thinks the expert can predict the output on this input vector.
- How should the gate be used to combine the experts?
  - Select the best expert (max score) and report its output.
  - Linearly combine the expert outputs based on the expert scores.
  - Stochastically select an expert based on the scores.

Mixtures of Experts

- Basic idea: there are a collection of "experts", each of which can produce sensible outputs $\mathbf{y}$ for a small region of the input space $\mathbf{x}$.
- There is also a "manager" or "gate" which decides which expert should be called upon for a particular set of inputs.
- We must train *both* the gate and the experts.

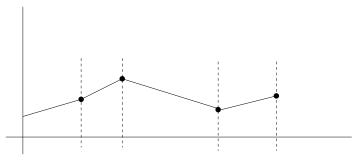


Hard Selection

- Let the score assigned by the gate to expert j be $\eta_j(\mathbf{x})$.
  Let the output of each expert be $\mathbf{y}_j$.
  Assume there is Gaussian noise with covar Σ on the outputs.
- If the gate selects the best expert, it is just doing a hard partitioning of the input space:

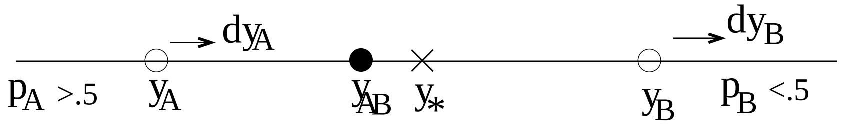
$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; \mathbf{y}_k, \Sigma) \qquad \text{iff } g_k = \max \ \{g_j\}$$

- As we have seen, this makes the cost function piecewise constant with respect to gate parameters, so training the gate using gradients becomes very difficult.



Linear Combination

- Let the score assigned by the gate to expert j be $\eta_j(\mathbf{x})$.
  Let the associated probabilities be $g_j = \exp \eta_j / \sum_k \exp \eta_k$.
  Let the output of each expert be $\mathbf{y}_j$.
- Linear combination of the experts' outputs:
$$p(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; \textstyle\sum_j g_j \mathbf{y}_j, \Sigma)$$
- Seems like an obvious solution, but it actually encourages co-operation instead of specialization.
- This can lead to crazy gradients.

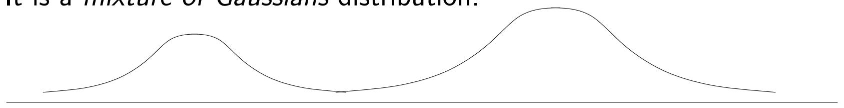


MOE Components

- We typically use a *linear logistic regression* model for the gate:
$$p(j|\mathbf{x}) = g_j = \frac{\exp(\eta_j)}{\sum_k \exp(\eta_k)} = \frac{\exp(\mathbf{v}_j^\top \mathbf{x})}{\sum_k \exp(\mathbf{v}_k^\top \mathbf{x})}$$
- We can use a linear model for each expert, or a linear model with a nonlinearity, or a neural network or anything else:
$$p(\mathbf{y}|\mathbf{x}, j) = \mathcal{N}(\mathbf{y}; \mathbf{U}_j \mathbf{x}, \Sigma) \qquad \text{or}$$
$$p(\mathbf{y}|\mathbf{x}, j) = \mathcal{N}(\mathbf{y}; f(\mathbf{U}_j \mathbf{x}), \Sigma) \qquad \text{or } \ldots$$
- In fact, radial basis networks and multilayer perceptrons/neural networks can be thought of as very simple mixtures of experts which use linear combination to fuse the experts and have a gate which is a constant function (represented by the last layer of weights).

Stochastic Selection

- Gate picks a single expert, with probabilities dictated by scores:
$$p(\mathbf{y}|\mathbf{x}) = \sum_j g_j \mathcal{N}(\mathbf{y}; \mathbf{y}_j, \Sigma)$$
- The output distribution is no longer unimodal!
  It is a *mixture of Gaussians* distribution.



- Now, there is no co-operation. Each expert tries to get the answer right on their own, but the size of their learning signal depends on how likely the gate is to select them.
- Your first exposure to a very important idea: the difference between linearly combining means and linearly combining distributions.

Objective Function for MOEs

- Consider the log likelihood function for the linear MOE architecture:
$$\ell(\mathbf{v}, \mathbf{U}) = \sum_n \log p(\mathbf{y}_n|\mathbf{x}_n)$$
$$= \sum_n \log \sum_j \frac{\exp(\mathbf{v}_j^\top \mathbf{x}_n)}{\sum_k \exp(\mathbf{v}_k^\top \mathbf{x}_n)} \mathcal{N}(\mathbf{y}_n; \mathbf{U}_j \mathbf{x}_n, \Sigma)$$
- For maximum likelihood learning, we want to take derivatives of this objective function with respect to the parameters.
- This looks hard! The objective function is a log of a sum.
- But there is another fundamental idea here: *sum over all possible ways in which the model could have produced the data.*
- Think of a *latent* or *hidden* unobserved random variable associated with each data case indicating which expert was responsible for generating its output given the input.

Gradients for a Mixture of Experts

- Actually the gradients come out quite nicely:

$$\partial\ell/\partial\mathbf{U}_j = \sum_n p(j|\mathbf{x}_n, \mathbf{y}_n)\left(\mathbf{y}_n - \mathbf{U}_j\mathbf{x}_n\right)\mathbf{x}_n^\top$$

$$\partial\ell/\partial\mathbf{v}_j = \sum_n \left(p(j|\mathbf{x}_n, \mathbf{y}_n) - p(j|\mathbf{x}_n)\right)\mathbf{x}_n$$

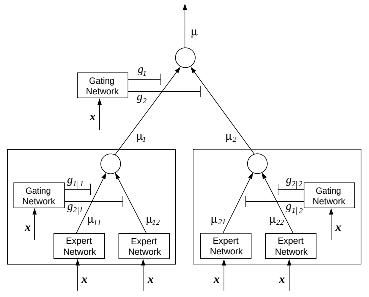
  The gradients include the *posterior* probability of each expert:

$$p(j|\mathbf{x}_n, \mathbf{y}_n) = \frac{p(j|\mathbf{x}_n)p(\mathbf{y}_n|j, \mathbf{x}_n)}{\sum_k p(k|\mathbf{x}_n)p(\mathbf{y}_n|k, \mathbf{x}_n)}$$

- The gradient for each expert is like a normal regression problem but with the data weighted by the expert posterior.
- The gradient of the gate parameters depends on the difference between the prior probability of an expert (as predicted by the gate) and the posterior (as predicted by how well they predict the true output.)

Hierarchical Mixtures of Experts

- The simple MOE idea can be extended to a hierarchical architecture in which each expert is itself a HME, until the last layer in which each expert actually makes a prediction.



- Very similar to a decision tree for regression but trained using maximum likelihood rather than greedy minimization of impurity.

MOE at Test Time

- Assume we can train MOEs How do we use them at test time?
- The correct thing to do is to get the test inputs $\mathbf{x}_{test}$, probabilistically select an expert using the gate, report that expert's output, *and repeat the process many times*.
- This gives you a whole distribution $p(\mathbf{y}|\mathbf{x}_{test})$ over outputs given the single test input.
- However, we might want to *summarize* this distribution.
- We can choose the *mode* of this distribution, corresponding to the output of the most likely expert.
- Or we can choose the *mean* of this distribution, corresponding to the weighted sum of the expert outputs.
- Notice: even though we might use these tricks to summarize the outputs, the underlying model is still one of stochastic selection, and so our training will not be messed up.

An Idea...

- What if instead of taking the gradient, we just solved a weighted least-squares problem and a weighted logistic-regression problem, both of which are convex.
- Find the "optimum" parameters given the current posterior weightings and then recalculate the weights.
- Excellent idea! It is called *Expectation Maximization* or EM, and we will see it soon.
- EM is a form of bound optimization.

Bound Optimization

- A completely different way to do optimization is to come up with consecutive upper (lower) bounds on your objective function and optimize those bounds.
- Assume we can find functions $Q(\mathbf{w}, \mathbf{z})$ and $\mathbf{z}(\mathbf{w})$ such that:

  $Q(\mathbf{w}, \mathbf{z}(\mathbf{w})) = E(\mathbf{w}) \leq Q(\mathbf{w}, \mathbf{z}^*)$ for any $\mathbf{w}$ and any $\mathbf{z}^* \neq \mathbf{z}(\mathbf{w})$

  $\arg\min_{\mathbf{w}} Q(\mathbf{w}, \mathbf{z}(\mathbf{w}^*))$ can be found easily for any $\mathbf{w}^*$
- Now iterate: $\mathbf{w}^{t+1} = \arg\min_{\mathbf{w}} Q(\mathbf{w}, \mathbf{z}(\mathbf{w}^t))$
- Guarantee:

$$\begin{aligned} E(\mathbf{w}^{t+1}) &= Q(\mathbf{w}^{t+1}, \mathbf{z}(\mathbf{w}^{t+1})) \\ &\leq Q(\mathbf{w}^{t+1}, \mathbf{z}(\mathbf{w}^t)) \\ &\leq Q(\mathbf{w}^t, \mathbf{z}(\mathbf{w}^t)) = E(\mathbf{w}^t) \end{aligned}$$