Lecture 6:

Optimization & Supervised Mixtures

Sam Roweis

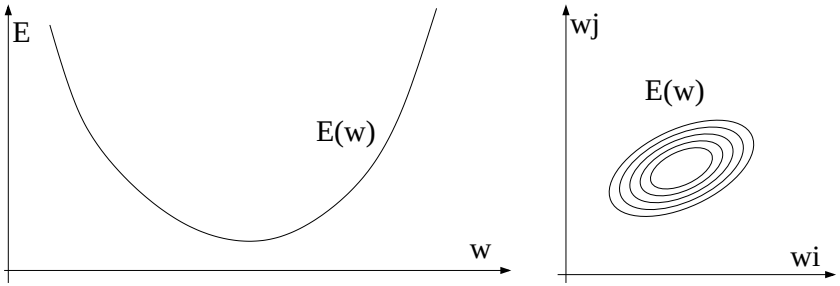
October 15, 2002

Partial Derivatives of Error

- Question: if we wiggle w_k *and keep everything else the same*, does the error get better or worse?
- Luckily, calculus has an answer to exactly this question: $\frac{\partial E}{\partial w_k}$.
- Use a differentiable cost function E and compute *partial derivatives* of each parameter with respect to this error: $\frac{\partial E}{\partial w_k}$
- 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. (e.g. using the *chain rule* and backprop).

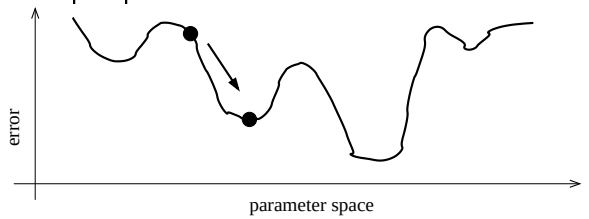
Error Surfaces and Weight Space

- Make a new "error function" $E(\mathbf{w})$ which we want to minimize.
- $E(\mathbf{w})$ can be the negative of the log likelihood function or, better yet, of a penalized log likelihood function.
- Consider a fixed training set. Now, for each setting of the weights there is some error: this defines an *error surface* in *weight space*.
- Learning == descending the error surface.



Convexity, Local Optima

- Unfortunately, many error functions while differentiable are not *convex*. Convexity means that the second derivative is always positive. No linear combination of weights can have greater error than the linear combination of the original errors.
- When using gradient descent we can get stuck in *local minima*. Where we end up depends on where we start.



- Things like linear regression, logistic regression, lasso, all have convex error functions. Neural networks, HMEs do not.

"Bold Driver" 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* grad. descent, but mini-batch or online may be better.

Quadratic Cost Functions

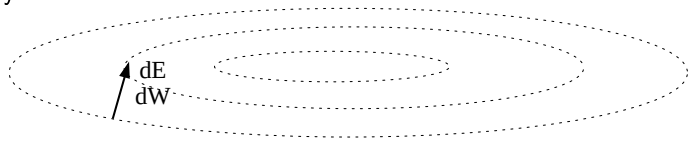
- A very common cost function is the quadratic: $E(\mathbf{w}) = \mathbf{w}^\top \mathbf{A}\mathbf{w} + 2\mathbf{w}^\top \mathbf{b} + c$.
- This comes up as the log probability when using Gaussians.
- Fact: sum of quadratics is another quadratic.
- Fact: it is easy to find the minimum of a quadratic bowl:

$$E(w) = a + bw + cw^2 \quad \Rightarrow \quad w^* = -b/2c$$

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

Curved Error Surfaces

- Notice: the error surface may be curved differently in different directions. This means that the gradient does not necessarily point directly at the nearest local minimum.



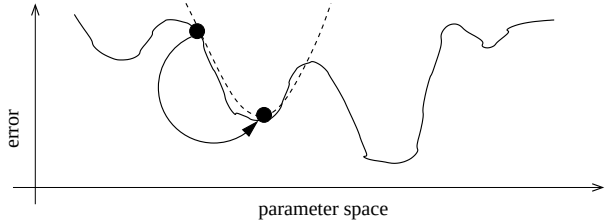
- The local geometry of curvature is measured by the *Hessian matrix* of second derivatives: $H_{ij} = \partial^2 E / \partial w_i w_j$.
- Eigenvectors/values of the Hessian describe the directions of principal curvature and the amount of curvature in each direction. Near a local minimum, the Hessian is positive definite.
- Maximum stepsize is $\frac{2}{\lambda_{max}}$
  Rate of convergence depends on $(1 - 2\frac{\lambda_{min}}{\lambda_{max}})$.

Local Quadratic Approximation

- By taking a Taylor series of the error function around any point in weight space, we can make a *local quadratic approximation* based on the value, slope and curvature:

$$E(\mathbf{w}-\mathbf{w}_0) \approx E(\mathbf{w}_0) + (\mathbf{w}-\mathbf{w}_0)^\top \frac{\partial E}{\partial \mathbf{w}} + (\mathbf{w}-\mathbf{w}_0)^\top \frac{\mathbf{H}(\mathbf{w}_0)}{2}(\mathbf{w}-\mathbf{w}_0)$$

  $\mathbf{H}$ is the *Hessian* matrix of second derivatives: $H_{ij} = \partial^2 E / \partial w_i w_j$



- Newton's method: jump to the minimum of this quadratic, repeat.

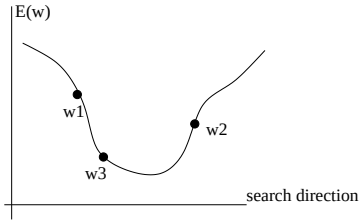
$$\mathbf{w}^* = \mathbf{w} - \mathbf{H}^{-1}(\mathbf{w})\frac{\partial E}{\partial \mathbf{w}}$$

Second Order Methods

- Newton's method is an example of a *second order* optimization method because it makes use of the curvature or Hessian matrix.
- Second order methods often converge much more quickly, but it can be very expensive to calculate and store the Hessian matrix.
- In general, most people prefer clever first order methods which need only the value of the error function and its gradient with respect to the parameters.

Line Search

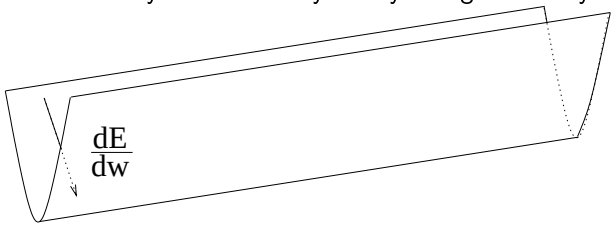
- Rather than take a fixed step in the direction of the gradient or the momentum-smoothed gradient, it is possible to do a search along that direction to find the minimum of the function.



- Usually the search is a bisection, which bounds the nearest local minimum along the line between any two points $\mathbf{w}_1$ and $\mathbf{w}_2$ such that there is a third point $\mathbf{w}_3$ with $E(\mathbf{w}_3) < E(\mathbf{w}_1)$ and $E(\mathbf{w}_3) < E(\mathbf{w}_2)$.

Momentum

- If the error surface is a long and narrow valley, grad. descent goes quickly down the valley walls but very slowly along the valley bottom.



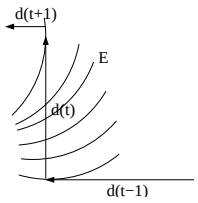
- We can alleviate this by updating our parameters using a combination of the previous update and the gradient update:

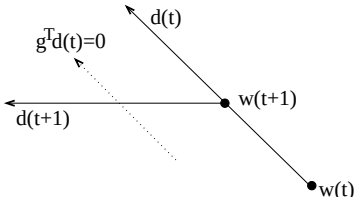
$$\Delta w_j^t = \lambda \Delta w_j^{t-1} + (1-\lambda)\,\epsilon\,\partial E/\partial w_j(\mathbf{w}^t)$$

- Usually, λ is quite high, about 0.95.
- When we have to retract a step, we set Δw_j to zero.
- Physically, this is like giving *momentum* to our weights.

Conjugate Gradients

- At the end of a line search, the new gradient is almost orthogonal to the direction we just searched in.
- So if we choose the next search direction to be the new gradient, we will always be searching successively orthogonal directions and things will be very slow.
- Instead, select a new direction so that, *to first order*, as we move in the new direction the gradient parallel to the old direction stays zero. This involves blending the current gradient with the previous search direction: $\mathbf{d}(t+1) = -\mathbf{g}(t+1) + \beta(t)\mathbf{d}(t)$.





Conjugate Gradients

- To first order, all three expressions below satisfy our constraint that along the new search direction $\mathbf{g}^\top \mathbf{d}(t) = 0$:

$$\mathbf{d}(t+1) = -\mathbf{g}(t+1) + \beta(t)\mathbf{d}(t)$$

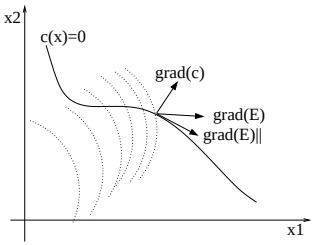
$$\beta(t) = \frac{\mathbf{g}^\top(t+1)(\mathbf{g}(t+1) - \mathbf{g}(t))}{\mathbf{d}^\top(t)(\mathbf{g}(t+1) - \mathbf{g}(t)} \quad \text{Hestenes-Stiefel}$$

$$\beta(t) = \frac{\mathbf{g}^\top(t+1)(\mathbf{g}(t+1) - \mathbf{g}(t))}{\mathbf{g}^\top(t)\mathbf{g}(t)} \quad \text{Polak-Ribiere}$$

$$\beta(t) = \frac{\mathbf{g}^\top(t+1)\mathbf{g}(t+1)}{\mathbf{g}^\top(t)\mathbf{g}(t)} \quad \text{Fletcher-Reeves}$$

Lagrange Multipliers

- Imagine that our parameters have to live inside the constraint surface $c(\mathbf{w}) = 0$.
- To optimize our function $E(\mathbf{w})$, we want to look at the component of the gradient that lies *within* the surface, i.e. with zero dot product to the normal of the constraint. At the constrained optimum, this gradient component is zero, in other words the gradient of the function is parallel to the gradient of the constraint surface.



Constrained Optimization

- Sometimes we want to optimize with some *constraints* on the parameters.
  e.g. variances are always positive
  e.g. priors are non-negative and sum to unity (live on the simplex)
- There are two ways to get around this.
  First, we can reparametrize so that the new parameters are unconstrained.
  e.g. use log(variances) or use softmax inputs for priors.
- The other way is to explicitly incorporate the constraints into our cost function.

Lagrange Multipliers

- At the constrained optimum, *the gradient of the function is parallel to the gradient of the constraint surface*:

$$\partial E/\partial \mathbf{w} = \lambda \partial c/\partial \mathbf{w}$$

  the constant of proportionality is called the *Lagrange multiplier*. Its value can be found by forcing $c(\mathbf{w}) = 0$.
- In general, the *Lagrangian* function

$$L(\mathbf{w}, \lambda) = E(\mathbf{w}) + \lambda^\top \mathbf{c}(\mathbf{w})$$
$$\partial L/\partial \mathbf{w} = \partial E/\partial \mathbf{w} + \lambda^\top \partial \mathbf{c}/\partial \mathbf{w}$$
$$\partial L/\partial \lambda = \mathbf{c}(\mathbf{w})$$

  has the property that when its gradient is zero, the constraints are satisfied and there is no gradient within the constraint surface.

Quadratic Cost with Linear Constraints

- Example: find the maximum over $\mathbf{x}$ of the quadratic form:

$$E(\mathbf{x}) = \mathbf{b}^\top \mathbf{x} - \frac{1}{2}\mathbf{x}^\top \mathbf{A}^{-1}\mathbf{x}$$

subject to the K conditions $c_k(\mathbf{x}) = 0$.

- Answer: use Lagrange multipliers:

$$L(\mathbf{x}, \lambda) = E(\mathbf{x}) + \lambda^\top \mathbf{c}(\mathbf{x})$$

Now set $\partial L/\partial \mathbf{x} = 0$ and $\partial L/\partial \lambda = 0$. Result:

$$\mathbf{x}^* = \mathbf{A}\mathbf{b} + \mathbf{A}\mathbf{C}\lambda$$
$$\lambda = -4(\mathbf{C}^\top \mathbf{A}\mathbf{C})\mathbf{C}^\top \mathbf{A}\mathbf{b}$$

where the kth column of $\mathbf{C}$ is $\partial c_k(\mathbf{x})/\partial \mathbf{x}$

Quadratic Programming

- QP optimizes a quadratic cost function subject to linear constraints:

$$E(\mathbf{w}) = \mathbf{w}^\top \mathbf{A}\mathbf{w} + 2\mathbf{w}^\top \mathbf{b}$$
$$\mathbf{G}\mathbf{w} < \mathbf{g}$$
$$\mathbf{C}\mathbf{w} = \mathbf{c}$$

Linear Programming

- LP optimizes a linear cost function subject to linear constraints:

$$E(\mathbf{w}) = \mathbf{w}^\top \mathbf{a} + \mathbf{b}$$
$$\mathbf{G}\mathbf{w} < \mathbf{g}$$
$$\mathbf{C}\mathbf{w} = \mathbf{c}$$

- Can always be transformed to *standard form* using slack variables:

$$E(\mathbf{w}) = \mathbf{w}^\top \mathbf{a}$$
$$\mathbf{w} > 0$$
$$\mathbf{C}\mathbf{w} = \mathbf{c}$$

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:

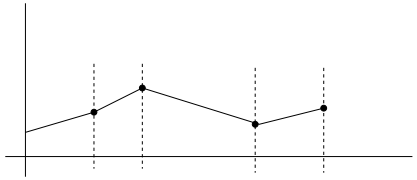
$Q(\mathbf{w}, \mathbf{z}(\mathbf{w})) = E(\mathbf{w}) \leq Q(\mathbf{w}, \mathbf{z}^*)$ for any $\mathbf{w}$ and any $\mathbf{z}^* \neq \mathbf{z}(\mathbf{w})$
$\arg\min_{\mathbf{w}} Q(\mathbf{w}, \mathbf{z}(\mathbf{w}^*))$ can be found easily for any $\mathbf{w}^*$

- Now iterate: $\mathbf{w}^{t+1} = \arg\min_{\mathbf{w}} Q(\mathbf{w}, \mathbf{z}(\mathbf{w}^t))$
- Guarantee:

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

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?



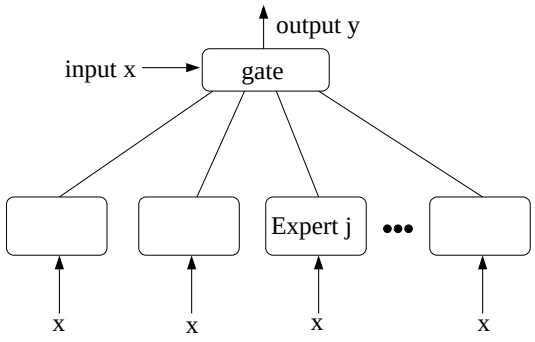
- Problem: since the cost function is *piecewise constant* with respect to these boundaries, taking derivatives will not work.
- Solution: use "soft" splits of the data space, rather than hard splits.

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.

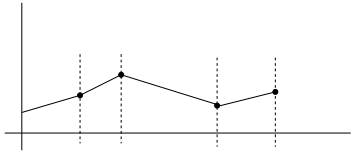


Hard Selection

- 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.

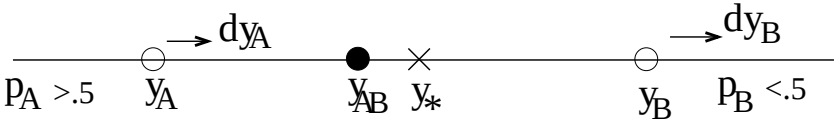


Linear Combination

- 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 Parametrization

- 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:

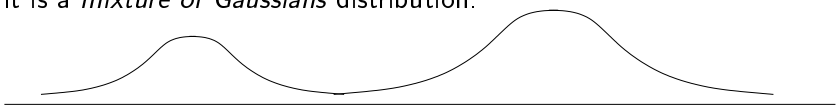
$$p(\mathbf{y}|\mathbf{x}, j) = \mathcal{N}(\mathbf{y}; \mathbf{U}_j \mathbf{x}, \Sigma) \qquad \text{or}$$
$$p(\mathbf{y}|\mathbf{x}, j) = \mathcal{N}(\mathbf{y}; f(\mathbf{U}_j \mathbf{x}), \Sigma) \qquad \text{or } \ldots$$

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.

Gradients for a Mixture of Experts

- Actually the gradients come out quite nicely:

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

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

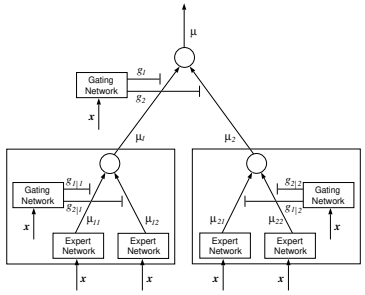
  The gradients include the *posterior* probability of each expert:

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

- 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.)

Heirarchical Mixtures of Experts

- The simple MOE idea can be extended to a heirarchical architecture in which each expert is iteself 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.