Lecture 5:

Objective Functions & Optimization

Sam Roweis

October 12, 2004

Maximum Likelihood

- Maximum likelihood asks the question: for which setting of the parameters 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 $\mathbf{w}$:

$$\begin{aligned}\ell(\mathbf{w}) &= \log p(y_1, \mathbf{x}_1, y_2, \mathbf{x}_2, \ldots, y_n, \mathbf{x}_n|\mathbf{w}) \\ &= \log \prod_b p(y_n, \mathbf{x}_n|\mathbf{w}) && \text{since iid} \\ &= \sum_n \log p(y_n, \mathbf{x}_n|\mathbf{w}) && \text{since } \log \prod = \sum \log\end{aligned}$$

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

$$\ell(\mathbf{w}) = -\frac{1}{2\sigma^2}\sum_n (y_n - f(\mathbf{x}_n; \mathbf{w}))^2 + \text{const}$$

Regression/Classification & Probabilities

- We assume our data are iid from an unknown joint distribution $p(y, \mathbf{x}|\mathbf{w})$ or an unknown conditional $p(y|\mathbf{x}, \mathbf{w})$.
- We see some examples $(y_1, \mathbf{x}_1)(y_2, \mathbf{x}_2) \ldots (y_n, \mathbf{x}_n)$ and we want to infer something about the parameters (weights) of our model.
- The most basic thing to do is to optimize the parameters using *maximum likelihood* or *maximum conditional likelihood*.
- A better thing to do is *maximum penalized (conditional) likelihood*, which includes regularization effects like factorization (independence assumptions), shrinkage, input selection, or smoothing (parameter priors).

Bayesian Programme

- Ideally, we would be Bayesian, introduce a prior $p(\mathbf{w})$, and use Bayes rule to compute $p(\mathbf{w}|y_1, \mathbf{x}_1, y_2, \mathbf{x}_2, \ldots, y_n, \mathbf{x}_n)$.
- This is the *posterior distribution* of the parameters given the data. A true Bayesian would *integrate over it* to make future predictions:

  $$p(y^{\text{new}}|x^{\text{new}}, Y, X) = \int p(y^{\text{new}}|x^{\text{new}}, \mathbf{w})p(\mathbf{w}|Y, X)d\mathbf{w}$$

  but often analytically intractable and computationally very difficult
- We can settle for *maximizing* and using the argmax $\mathbf{w}^*$ to make future predictions: this is called *maximum a-posteriori*, or MAP.
- Many of the penalized maximum likelihood techniques we used for regularization are equivalent to MAP with certain parameter priors:
  - quadratic weight decay (shrinkage) ⇔ Gaussian prior (var=$1/2\Lambda$)
  - absolute weight decay (lasso) ⇔ Laplace prior (decay $= 1/\Lambda$)
  - smoothing on multinomial parameters ⇔ Dirichlet prior
  - smoothing on covariance matrices ⇔ Wishart prior

Maximum A Posteriori

- MAP asks the question: which setting of the weights is most likely to be drawn from the prior $p(\mathbf{w})$ and *then* to generate the data from the conditional $p(X, Y|\mathbf{w})$?
- To answer this, it assumes that the training data are iid, computes the log posterior and forms a function $\ell(\mathbf{w})$ which depends on the fixed training set we saw and on the argument $\mathbf{w}$:

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

(e.g. MAP is equivalent to ridge regression, if the noise model is Gaussian, the weight prior is Gaussian, and the datapoints are iid:

$$\ell(\mathbf{w}) = -\tfrac{1}{2\sigma^2} \textstyle\sum_n (y_n - \mathbf{w}^\top \mathbf{x}_n)^2 + \lambda \sum_i w_i^2 + \mathrm{const}$$

Quadratic Error Surfaces and IID data

- 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, since if the noise model is Gaussian, each of the 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:

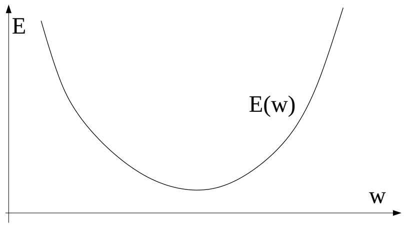
$$\begin{aligned}E(w) &= a + bw + cw^2 && \Rightarrow \quad w^* = -b/2c \\ E(\mathbf{w}) &= a + \mathbf{b}^\top \mathbf{w} + \mathbf{w}^\top \mathbf{C} \mathbf{w} && \Rightarrow \quad \mathbf{w}^* = -\frac{1}{2}\mathbf{C}^{-1}\mathbf{b}\end{aligned}$$

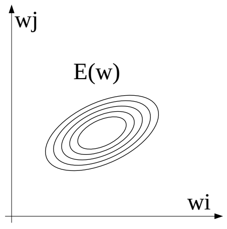
- Convince yourself that for linear regression with Gaussian noise:

$$\mathbf{C} = \mathbf{X}\mathbf{X}^\top \qquad \text{and} \qquad \mathbf{b} = -2\mathbf{X}\mathbf{y}^\top$$

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 or log posterior.
- Consider a fixed training set; think in weight (not input) space. At each setting of the weights there is some error (given the fixed training set): this defines an *error surface* in *weight space*.
- Learning == descending the error surface.
- Notice: If the data are IID, the error function E is a sum of error functions E_n, one per data point.





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: $\frac{\partial E}{\partial w_k}$.
- Plan: use a differentiable cost function E and compute *partial derivatives* of each parameter with respect to this error: $\frac{\partial E}{\partial w_k}$
- Use the *chain rule* to compute the derivatives.
- The vector of partial derivatives is called the *gradient* of the error. It points in the direction of steepest error descent in weight space.
- Three crucial questions:
  - How do we compute the gradient ∇E efficiently?
  - Once we have the gradient, how do we minimize the error?
  - Where will we end up in weight space?

“Bold Driver” Gradient Descent

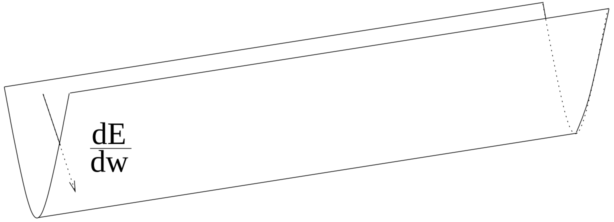
- Once we have the gradient of our error function, how do we minimize the weights? Follow it! But not too fast...
- **Algorithm Gradient Descent**

```
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.

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.



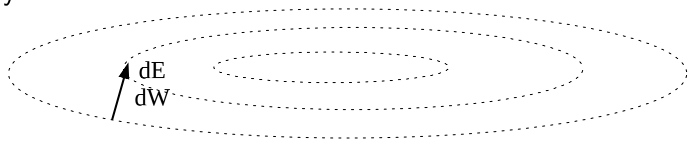
- We can alleviate this by updating our parameters using a combination of the previous update and the gradient update:

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

- Usually, β 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.

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 sensible stepsize is $\frac{2}{\lambda_{max}}$

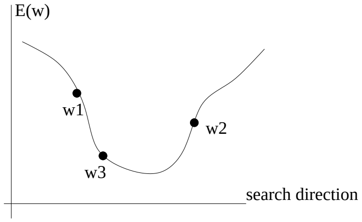
  Rate of convergence depends on $(1 - 2\frac{\lambda_{min}}{\lambda_{max}})$.

Mini-Batch and Online Optimization

- When our data is big, computing the exact gradient is expensive.
- This seems wasteful, since the only thing we are going to use the gradient for is to make a small change to the weights and then throw it away and measure it again at the new weights.
- An approximate gradient is just as useful as long as it is somewhat in line with the true gradient.
- One very easy way to do this is to use only a small batch of examples (not the whole data set), compute the gradient and make an update, then move to the next batch of examples. This is *mini-batch* optimization.
- In the limit, we can use only one example per batch, this is called *online gradient descent*, or *stochastic gradient descent*.
- These methods are often much faster than exact gradient descent, and are very effective when combined with momentum.

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 $E(\mathbf{w}_3) < E(\mathbf{w}_2)$.

Second Order Methods

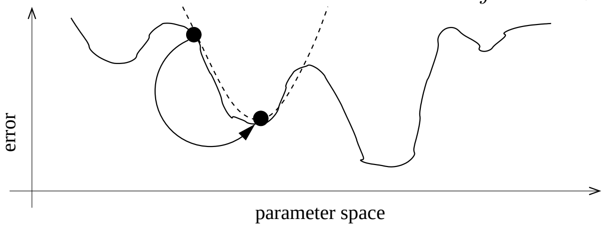
- Newton's method is an example of a *second order* optimization method because it makes use of the curvature or Hessian matrix.
- Second order methods often converge much more quickly, but it can be very expensive to calculate and store the Hessian matrix.
- In general, most people prefer clever first order methods which need only the value of the error function and its gradient with respect to the parameters. Often the sequence of gradients (first order derivatives) can be used to *approximate* the second order curvature. (This may even be better than the true Hessian, because we can constrain our approximation to be always positive definite.)
- Point of Possible Confusion: Newton's method is often described as a method of multidimensional root finding, which is a much harder problem: $x_{t+1} = x_t - f(x_t)/f'(x_t)$. In that case, it is trying to set the gradient vector $f(x) = \nabla E(x)$ to be the zero vector.

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.

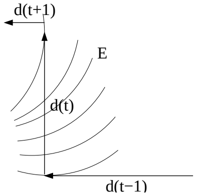
$$\mathbf{w}^* = \mathbf{w} - \mathbf{H}^{-1}(\mathbf{w})\frac{\partial E}{\partial \mathbf{w}}$$

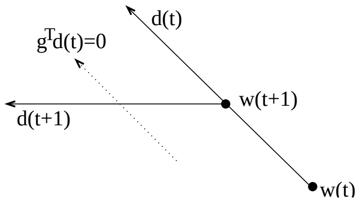
Newton and Quasi-Newton Methods

- Broyden-Fletcher-Goldfarb-Shanno (BFGS); Conjugate-Gradients (CG); Davidon-Fletcher-Powell (DVP); Levenberg-Marquardt (LM)
- All approximate the Hessian using recent function and gradient evaluations (e.g by averaging outer products of gradient vectors, but tracking the "twist" in the gradient; by projecting out previous gradient directions...).
- Then they use this approximate gradient to come up with a new search direction in which they do a combination of fixed-step, analytic-step and line-search minimizations.
- Very complex area (see reading) but we will go through in deatail only the CG method, although my current favourite optimizer is the limited-memory BFGS, which is like a multidimensional version of secant (actually false-position) optimization.

Conjugate Gradients

- Observation: 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 $\mathbf{g}^\top \mathbf{d}(t) = 0$:

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

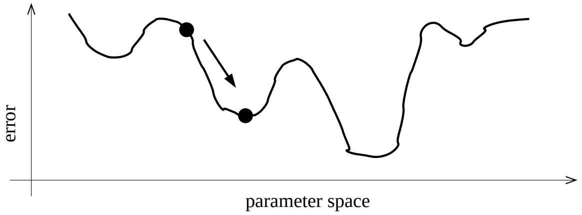
$$\beta(t) = \frac{\mathbf{g}^\top(t+1)(\mathbf{g}(t+1)-\mathbf{g}(t))}{\mathbf{d}^\top(t)(\mathbf{g}(t+1)-\mathbf{g}(t)} \qquad \text{Hestenes-Stiefel}$$

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

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

Convexity, Local Optima

- Unfortunately, many error functions while differentiable are not unimodal. When using gradient descent we can get stuck in *local minima*. Where we end up depends on where we start.



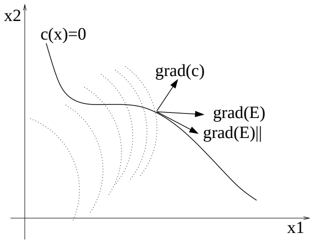
- Some very nice error functions (e.g. linear least squares, logistic regression, lasso) are *convex*, and thus have a unique (global) minimum. Convexity means that the second derivative is always positive. No linear combination of weights can have greater error than the linear combination of the original errors.
- But most settings do not lead to convex optimization problems.

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

- 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.



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

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

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:

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

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

Linear & Quadratic Programming

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

$$\begin{aligned} E(\mathbf{w}) &= \mathbf{w}^\top \mathbf{b} + \mathbf{b}_0 \\ \mathbf{G}\mathbf{w} &< \mathbf{g} \\ \mathbf{C}\mathbf{w} &= \mathbf{c} \end{aligned}$$

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

$$\begin{aligned} E(\mathbf{w}) &= \mathbf{w}^\top \mathbf{b} \\ \mathbf{w} &> 0 \\ \mathbf{C}\mathbf{w} &= \mathbf{c} \end{aligned}$$

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

$$\begin{aligned} E(\mathbf{w}) &= \mathbf{w}^\top \mathbf{A}\mathbf{w} + 2\mathbf{w}^\top \mathbf{b} + \mathbf{b}_0 \\ \mathbf{G}\mathbf{w} &< \mathbf{g} \\ \mathbf{C}\mathbf{w} &= \mathbf{c} \end{aligned}$$

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:

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