Optimizations in speech recognition

Daniel Povey

June, 2009

Thanks to Thomas Schaaf for comments and suggestions

Goals and overview

- This lecture will introduce some of the more important optimization techniques used in speech recognition, covering only Maximum Likelihood techniques.

- Will show side by side the mathematical notation used in papers, with partial, C++ code.

- Focus on the pure data modeling techniques, with no HMM involved (e.g. a one-state HMM). Generalization not hard.

- Topics covered:

  1. Discrete model
  2. Gaussian distribution estimation; mixture-of-Gaussians estimation
  3. Introduction to (bits of) vector/matrix calculus
  4. Adaptation: Maximum Likelihood Linear Regression (MLLR)
  5. Constrained MLLR; Semitied Covariance Transform
  6. Speaker Adaptive Training for MLLR
  7. Maximum A Posteriori (MAP) training.

Find the mistakes

- Each code segment has mistakes in it, some intentional, some not
- Prizes will be offered for pointing out the mistakes, not just in code but also in the text.
- This is open to all people present at the lecture (not just undergraduates)
- For particularly significant errors that I was not aware of (especially outside the code), double the prize is offered. Audience will help decide whether the errors merit double reward.
- Each prize has an expected value of $5. Prize is you have to guess a suit from a card randomly drawn from a pack. If you guess right you get $20.
- (Actually the expected value is a little more than $5 if we do not shuffle the pack after each pick and you are strategic).
- If the prize is doubled, you get two tries to guess the same card. The expected value of this is a little less than $10. (Is this right?)

Discrete model example: Data

- Data, in this context, means some collection of mathematical objects that we have to assign a probability to.
- Simple e.g.: a sequence of discrete values $x_1 \ldots x_N$, with $x_n \in \mathcal{S}$.
- The values x_n are members of the discrete set $\mathcal{S}$.

```
int N = 512; // Length N of sequence.
int S=1000 // Fix size of set S.
int *x = new  int[N];
// zero-based numbering (C++)
// This data is random.  Real data won't be.
for(int i=0;i<N;i++) x[i] = rand() % S;
```

Discrete model example: Parametric models

- Parametric models allow us to compute the probability of a particular set of data being generated.
- A very simple model for the discrete values $s \in \mathcal{S}$ is to have a probability for each of them: c_s, with $\sum_{s \in \mathcal{S}} c_s = 1$.
- We can model the sequence $\mathcal{X} = x_1 \ldots x_N$ by multiplying the probability of each of them: $P(\mathcal{X}) = \prod_{n=1}^{N} c_{x_n}$.
- Probability of sequence is simple product: independence assumption
- Parameters of the model are $\{c_s,\ s \in \mathcal{S}\}$.
- Parameters must be nonnegative, and must sum to one: $\sum_{s \in \mathcal{S}} c_s = 1$.

```
int S=1000;
float *c = new float[S];
// initialize to flat distribution.
for(int s=0;s<S) c[s] = 1.0 / S;
```

Discrete model example: Evaluating data probability given model

- Computing the probability $P(\mathcal{X}) = \prod_{n=1}^{N} c_{x_n}$.
- Compute this as a log value to avoid floating point underflow or overflow:
- $\log P(\mathcal{X}) = \sum_{n=1}^{N} \log c_{x_n}$.
- In a multi-class/classification context, might compare this value between different data models

```
float loglike(int *x, int N, float *c, int S){
  float ans=0.0, int n;
  for(n=0;n<N;n++){
    Assert(x[n]>=0&&x[n]<S);
    ans += log(c[x[n]]);
  }
  return ans;
}
```

Discrete model example: statistics

- Training means computing model parameters: in this case $c_s,\ s \in \mathcal{S}$.
- We want to maximize the probability of training data data, which is: $P(\mathcal{X}) = \prod_{n=1}^{N} c_{x_n}$.
- Equivalent to maximizing the log probability $\log P(\mathcal{X}) = \sum_{n=1}^{N} \log c_{x_n}$.
- This is equivalent to: $\sum_{s \in \mathcal{S}} n_s \log c_s$, where n_s is the count of s in the sequence $\mathcal{X}$.
- The "count" values n_s for $s \in \mathcal{S}$ (there would be 1000 of these) are the "statistics".
- We call them "sufficient statistics" because once we need them we have all we need to estimate the model (real definition is a little technical)

```
int *count = new int[S];
for(int s=0;s<S;s++) count[S]=0;
for(int i=0;i<N;i++) count[x[i]]++;
```

Discrete model example: update

- Want to maximize $\log P(\mathcal{X}) = \sum_{s \in \mathcal{S}} n_s \log c_s$, with sum-to-one constraint.
- Normal derivation relies on Lagrange multipliers. Easier derivation:
- Imagine quantities $\widehat{c}_s$ that don't necessarily sum to one (but don't sum to zero), so $c_s = \frac{\widehat{c}_s}{\sum_{s' \in \mathcal{S}} \widehat{c}_{s'}}$ (renormalizing).

$$\begin{aligned} P(\mathcal{X}) &= \sum_{s \in \mathcal{S}} n_s \left(\log \widehat{c}_s - \log \sum_{s' \in \mathcal{S}} \widehat{c}_{s'} \right) \\ \frac{\partial P(\mathcal{X})}{\partial \widehat{c}_s} &= \frac{n_s}{\widehat{c}_s} - \frac{\sum_{s' \in \mathcal{S}} n_{s'}}{\sum_{s' \in \mathcal{S}} \widehat{c}_{s'}} \end{aligned} \tag{1}$$

- Setting gradient to zero, and defining T as the total $\sum_{s \in \mathcal{S}} \widehat{c}_s$, we can work out $\widehat{c}_s = T \frac{n_s}{\sum_{s \in \mathcal{S}} n_s}$. Set T to 1 to make $\widehat{c}_s$ equivalent to c_s.

```
int totcount=0;
for(int i=0;i<S;i++) totcount += count[i];   // actually totcount==N now.
for(int i=0;i<S;i++)  c[i] = (float)(count[i]/totcount);
```

Discrete HMMs

- The model we described above was once the basis for state-of-the-art speech recognition.
- It was used in "Discrete HMM" systems.
- The speech feature based on cepstral coefficients was vector quantized, and each speech state had one of the models described above (i.e. each speech state had a set of weights $c_s, s \in \mathcal{S}$).
- Estimation in the HMM case involves the forward-backward algorithm.
- On each iteration, instead of integer counts n_s they would be continuous counts, i.e. weighted by posterior probabilities of HMM states.
- E.g. we would have data-counts $n_{js} = \sum_{t=1}^{T} \gamma_j(t)$, where $0 \leq \gamma_j(t) \leq 1$ are the posteriors of state j in the forward backward algorithm.

Probability density functions and likelihoods: scalar case

- A *probability density function*, or p.d.f., is a function of a continuous variable which says how likely a variable is to fall in a particular region.
- For scalar x, a p.d.f. $p(x)$ must satisfy $p(x) \geq 0$ for $x \in (-\infty, +\infty)$ (nonnegative) and $\int_{x=-\infty}^{+\infty} p(x)dx = 1$ (properly normalized).
- More proper as we are talking about the function p itself rather than its value $p(x)$ to say "a p.d.f. $p : \Re \rightarrow \Re$" ... but we are being informal.
- Meaning of p.d.f. is: $P(x \in (a, b)) = \int_{x=a}^{b} p(x)dx$.
- Notation: (a, b) is an "open range" $a < x < b$, whereas $[a, b]$ is a "closed range" $a \leq x \leq b$. Makes no difference here.
- A likelihood is a p.d.f. evaluated at a specific place, e.g. we might say $p(5) = 2$ for some function p. May be > 1 !
- A *cumulative distribution function*, or c.d.f., is like the integral from $-\infty$ of a p.d.f, i.e. $c(x) = \int_{y=-\infty}^{x} p(y)$. Meaning: $P(x < k) = c(k)$.
- A *distribution* says how likely a variable is to take particular values, we can talk about the *p.d.f. of a distribution* or the *c.d.f. of a distribution*. Applies for discrete case too.

Likelihoods: vector case

- For vector-valued features e.g. $\mathbf{x} \in \Re^n$ i.e. $\mathbf{x}$ is an n-dimensional vector..
- Notation: bold font for vector $\mathbf{x}$; $\Re$ the set of all real numbers; $\Re^n = \Re \times \Re \ldots \times \Re$ is the cartesian product of the set $\Re$, n times, i.e. the set of all n-tuples $(x_1 \in \Re, x_2 \in \Re, \ldots x_n \in \Re)$.
- Constraint: $\int p(\mathbf{x}) \mathrm{d}^n\mathbf{x} = 1$ (properly normalized distribution). Nonnegative.
- Notation (this is Riemann integral notation, which is the "normal" kind of integrals as far as non-mathematicians are concerned): $\mathrm{d}^n\mathbf{x}$ the volume of a small region: the same as the product $\mathrm{d}x_1\mathrm{d}x_2 \ldots \mathrm{d}x_n$ of the side lengths of a little hypercube.
- We would have just $\mathrm{d}\mathbf{x}$ for a line integral if we wanted the (vector) length of the little line segment.
- Note, integral above is the same as $\int_{x_1=-\infty}^{\infty} \int_{x_2=-\infty}^{\infty} \ldots \int_{x_n=-\infty}^{\infty} p(x) dx_1 dx_2 \ldots dx_n$. (Not separable like this for general volume integrals).
- Interpretion of likelihoods in vector case: $P(x \in W)$ with $\mathcal{W} \subset \Re^n$ is $\int_{\mathcal{W}} p(\mathbf{x}) d^n\mathbf{x}$.

Gaussian distribution

- Scalar: $\mathcal{N}(x; \mu, \sigma^2) = \exp\left(-\frac{1}{2}\left(\log 2\pi + \log \sigma^2 + \frac{(x-\mu)^2}{\sigma^2}\right)\right)$ (note, variance is σ^2).
- Vector $\mathbf{x} \in \Re^D$: $\mathcal{N}(\mathbf{x}; \mu, \mathbf{\Sigma}) = \exp\left(-\frac{1}{2}\left(D \log 2\pi + \log \det \mathbf{\Sigma} + (\mathbf{x}-\mu)\mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)\right)\right)$.
- Sometimes covariance $\mathbf{\Sigma}$ limited to be diagonal $\rightarrow$ independent scalar Gaussians in each dimension.

```
float diag_loglike(float *x, float *mu, float *sigmasq, int D){
   float ans=0.0;
   for(int i=0;i<D;i++)
     ans -= 0.5*(log(2*M_PI*sigmasq[i])+(mu[i]-x[i])*(mu[i]-x[i])/sigmasq[i]);
   return ans;
}
float full_loglike(float *x,float *mu,float **inv_sigmasq,float logdet,int D){
   float ans=-0.5*(D*log(2*M_PI) + logdet);
   for(int i=0;i<D;i++)
     for(int j=0;j<D;j++) ans += (mu[i]-x[i])*inv_sigmasq[i][j]*(mu[j]-x[j]);
   return ans;
}
```

Training Gaussian distributions

- Training one Gaussian distribution on a collection of (vector-valued) points $\mathbf{x}_1 \ldots \mathbf{x}_N$.
- Sufficient statistics are: (0th, 1st and 2nd order): $\gamma = N$, $\mathbf{m} = \sum_{n=1}^{N} \mathbf{x}_n$, $\mathbf{S} = \sum_{n=1}^{N} \mathbf{x}_n \mathbf{x}_n^T$ (not very standard notation).
- Easy to show by differentiation that we get a maximum of the likelihood when:
- $\mu = \frac{1}{\gamma}\mathbf{m}$
- $\mathbf{\Sigma} = \frac{1}{\gamma}\left(\mathbf{S} - 2\mathbf{m}\mu + \gamma\mu\mu^T\right)$

```
void reest(float gamma, float *m, float *S, float *mu, float **Sigma){
   for(int i=0;i<N;i++) mu[i] = m[i]/gamma;
   for(int i=0;i<N;i++) for(int j=0;j<N;j++)
     Sigma[i][j]=(S[i][j]-m[i]*mu[j]-mu[i]*m[j]-gamma*mu[i]*mu[j])/gamma;
}
// would then need to invert Sigma and compute determinant before using
// model, need matrix software for this.
```

Vector/matrix calculus

- Math required for the previous slide needs vector/matrix calculus for vector case.

- We generally only need to differentiate a *scalar* function with respect to vector or matrix valued quantities.

- e.g. $\frac{d}{d\mathbf{x}}(\mathbf{x}^T\mathbf{A}\mathbf{x})$. Answer will always be the same dimension as the transpose of the thing we are differentating with respect to (i.e. $\mathbf{x}$ in this case).

- E.g. gradient w.r.t. a column vector is a row vector.

- Meaning is: if $\mathbf{v}^T = \frac{dF}{d\mathbf{x}}$, $v_1 = \frac{\partial F}{\partial x_1}$. (Note, using curly ∂ for partial derivative since now more than one variables are involved).

- Answer generally corresponds somehow to scalar answer, e.g. $\frac{d}{d\mathbf{x}}\mathbf{x}^T\mathbf{A}\mathbf{x} = \mathbf{x}^T(\mathbf{A} + \mathbf{A}^T)$, whereas $\frac{d}{dx}x^2A = 2Ax$.

Vector/matrix derivatives - why the transpose?

- There are actually competing conventions regarding whether $\frac{df}{d\mathbf{A}}$ should be transposed w.r.t. $\mathbf{A}$.
- It will be generally be obvious from the equations which is the case.
- Rationale for convention used here is that (for vectors) $\frac{df}{d\mathbf{x}}$ is like a "co-vector" to $\mathbf{x}$, i.e. product between the two would make sense.
- E.g. consider $\frac{df}{d\mathbf{x}}\Delta$, where Δ is a change in $\mathbf{x}$. This expression makes sense (change in function value f); clearly $\Delta\mathbf{x}$ is the same kind of quantity as $\mathbf{x}$ (because you can get it from a difference between $\mathbf{x}$ and $\mathbf{x}'$).
- Use of this convention means we don't have to write $\left(\frac{df}{d\mathbf{x}}\right)^T(\Delta\mathbf{x})$ (or use an explicit dot product $\frac{df}{d\mathbf{x}} \cdot (\Delta\mathbf{x})$, which is the same thing).
- Either convention is OK as long as it is used consistently.

Vector conventions– columns vs rows.

- Most people when they write $\mathbf{x}$, assume that $\mathbf{x}$ is a column vector. To write a column vector they would write $\mathbf{x}^T$.

- Sometimes people define a named variable like $\mathbf{x}$ to be a row vector, but it should be stated in the text. This is not very normal.

Traces

- The trace operator is very useful in matrix/vector calculus.
- The trace of a square matrix is the sum of its diagonal elements.
- For a scalar x, this corresponds to x itself: $\text{tr}(x) = x, x \in \Re$.
- $\text{tr}(\mathbf{AB}) = \sum_{i,j} a_{ij} b_{ji}$.
- $\text{tr}(\mathbf{AB}^T) = \sum_{i,j} a_{ij} b_{ij}$. Like a matrix form of dot-product.
- $\text{tr}(\mathbf{AB}) = \text{tr}(\mathbf{BA})$. Can move things from beginning to end and vice versa, so $\text{tr}(\mathbf{ABCD}) = \text{tr}(\mathbf{BCDA})$ (bracket $\mathbf{BCD}$ to see why).
- $\text{tr}(\mathbf{A}) = \text{tr}(\mathbf{A}^T)$. Can transpose contents of trace operator.
- $\text{tr}(\mathbf{A}+\mathbf{B}) = \text{tr}(\mathbf{A})+\text{tr}(\mathbf{B})$ if $\mathbf{A}$ and $\mathbf{B}$ have same dimension (should hardly need stating).

Vector/matrix calculus - reduced axiomatization (simplified, and only allowing differentiation of scalar functions)

- Easy to show $\frac{d}{d\mathbf{A}}\text{tr}(\mathbf{AB}) = \mathbf{B}$ (1)
- $f(\mathbf{A}+\mathbf{\Delta}) \simeq f(\mathbf{A}) + \text{tr}(\mathbf{\Delta}\frac{\partial f}{\partial \mathbf{A}})$ (2)
- Informal version of our "special" product rule: each time $\mathbf{A}$ appars in in an expression, differentiate with respect to that $\mathbf{A}$ and treat everything else as a constant; add up the results of differentiating w.r.t. each $\mathbf{A}$.
- Formally: if $f(\mathbf{A}) = g(\mathbf{A})h(\mathbf{A})$, where g and h can be scalar or vector or matrix-valued functions, define $\bar{\mathbf{A}} = \mathbf{A}$, and $\frac{df}{d\mathbf{A}} = \frac{\partial}{\partial \mathbf{A}} g(\bar{\mathbf{A}})h(\mathbf{A}) + g(\mathbf{A})h(\bar{\mathbf{A}})$ (i.e. gradient with $\bar{\mathbf{A}}$ fixed; note use of partial derivative symbol ∂). (3)

Vector/matrix calculus - example

- Want $\frac{d}{d\mathbf{x}}\mathbf{x}^T\mathbf{A}\mathbf{x}$
- Add trace: $\frac{d}{d\mathbf{x}}\text{tr}(\mathbf{x}^T\mathbf{A}\mathbf{x})$
- Our "special" product rule (using $\bar{\mathbf{x}} = \mathbf{x}$): $\frac{d}{d\mathbf{x}}\text{tr}(\mathbf{x}^T\mathbf{A}\bar{\mathbf{x}}) + \frac{d}{d\mathbf{x}}\text{tr}(\bar{\mathbf{x}}^T\mathbf{A}\mathbf{x})$.
- Apply $\text{tr}(\mathbf{AB}) = \text{tr}(\mathbf{BA})$ and $\text{tr}(\mathbf{A}) = \text{tr}(\mathbf{A}^T)$ to get $\mathbf{x}$ on the left: $\frac{d}{d\mathbf{x}}\left(\text{tr}(\mathbf{x}\bar{\mathbf{x}}^T\mathbf{A}) + \text{tr}(\mathbf{x}\bar{\mathbf{x}}^T\mathbf{A}^T)\right)$.
- Apply (1) to get: $\bar{\mathbf{x}}^T(\mathbf{A}^T + \mathbf{A})$. Discard the bar at this point (because $\bar{\mathbf{x}} = \mathbf{x}$) to get $\mathbf{x}^T(\mathbf{A}^T + \mathbf{A})$ (the distinction between $\mathbf{x}$ and $\bar{\mathbf{x}}$ was only used for partial differentiation, they are the same variable).
- We derived $\frac{d}{d\mathbf{x}}\mathbf{x}^T\mathbf{A}\mathbf{x} = \mathbf{x}^T(\mathbf{A}^T + \mathbf{A})$.

Vector/matrix calculus example - determinant.

- Determinants: $\frac{d}{d\mathbf{A}} \det \mathbf{A} = \mathbf{I}$ where $\mathbf{A} \simeq \mathbf{I}$. Can prove this recursively using determinant formula and some easy-to-prove facts about determinants.

- Because determinants are multiplicative e.g. $\det(\mathbf{AB}) = \det \mathbf{A} \det \mathbf{B}$, can use:

- $\frac{d}{d\mathbf{A}} \det \mathbf{A}$ around $\mathbf{A} = \bar{\mathbf{A}}$ equals: $\frac{d}{d\mathbf{A}} \det(\mathbf{A}\bar{\mathbf{A}}^{-1}) \det \bar{\mathbf{A}}$,
$= \det(\mathbf{B}) \det \bar{\mathbf{A}}$ substituting $\mathbf{B} = \mathbf{A}\bar{\mathbf{A}}^{-1}$, with $\mathbf{B} = \mathbf{I}$ at the current point.

- We can then use $\frac{d}{d\mathbf{B}} \det \mathbf{B} = \mathbf{I}$ since $\mathbf{B} \simeq \mathbf{I}$

- Then use (2) to get $\det \mathbf{B} \simeq 1 + \mathrm{tr}((\mathbf{B} - \mathbf{I})\mathbf{I}) = \mathrm{tr}(\mathbf{B}) - D + 1$.

- So $\det \mathbf{A} \simeq (\det \bar{\mathbf{A}})(\mathrm{tr}(\mathbf{A}\bar{\mathbf{A}}^{-1}) - D + 1)$.

- Using (1): $\frac{d}{d\mathbf{A}} \det \mathbf{A} = (\det \mathbf{A})\mathbf{A}^{-1}$.

Deriving Gaussian update

- $\log p(\mathcal{X}) = K - 0.5\left(\gamma \det \boldsymbol{\Sigma} + \text{tr}\left(\boldsymbol{\Sigma}^{-1}(\mathbf{S} + \mu\mathbf{m}^T + \mathbf{m}\mu^T + \gamma\mu\mu^T)\right)\right)$. where $\gamma, \mathbf{m}, \mathbf{S}$ are zeroth,first, and second order statistics.
- Differentiate w.r.t μ and set to zero:
- $\frac{d}{d\mu}\text{tr}(\boldsymbol{\Sigma}^{-1}(\mathbf{S} + \mu\mathbf{m}^T + \mathbf{m}\mu^T + \gamma\mu\mu^T)) = 0$
- $(\mathbf{m}^T + \gamma\mu^T)(\boldsymbol{\Sigma}^{-1} + \boldsymbol{\Sigma}^{-T}) = 0$
- $\mu = \frac{1}{\gamma}\mathbf{m}$.
- Differentiate w.r.t. $\mathbf{T} = \boldsymbol{\Sigma}^{-1}$ and set to zero (use $\log\det\boldsymbol{\Sigma} = -\log\det\mathbf{T}$)
- $-0.5\left((\mathbf{S} + \mu\mathbf{m}^T + \mathbf{m}\mu^T + \gamma\mu\mu^T) - \gamma\mathbf{T}^{-1}\right) = 0$
- $\mathbf{T}^{-1} = \boldsymbol{\Sigma} = \frac{1}{\gamma}(\mathbf{S} + \mu\mathbf{m}^T + \mathbf{m}\mu^T + \gamma\mu\mu^T) = \frac{1}{\gamma}\mathbf{S} - \mu\mu^T$.

Mixture of Gaussians distribution

- Model data with: $p(\mathbf{x}) = \sum_{m=1}^{M} c_m \mathcal{N}(\mathbf{x}; \mu_m, \mathbf{\Sigma}_m)$.
- Will first state approach used to optimize the likelihood, then derive it.
- Training now does not "jump to answer", is iterative.
- For each data point $\mathbf{x}_n$ compute proportion of the likelihood $\gamma_m(n)$ accounted for by Gaussian m, store weighted statistics:
- $\gamma_m = \sum_{n=1}^{N} \gamma_m(n)$, $\mathbf{m}_m = \sum_{n=1}^{N} \gamma_m(n)\mathbf{x}_n$, $\mathbf{S}_m = \sum_{n=1}^{N} \gamma_m(n)\mathbf{x}_n\mathbf{x}_n^T$.
- Update equations are as before but indexed by Gaussian mixture index m.
- Derivation and update for weights is same as discrete case: $c_m = \frac{\gamma_m}{\sum_{m'} \gamma_{m'}}$.

Mixture of Gaussians distribution: Jensen's inequality (1 of 2)

- Jensen's inequality says for a real concave function ϕ (such as log function): $\phi\left(\frac{\sum a_i x_i}{\sum a_i}\right) \geq \frac{\sum a_i \phi(x_i)}{\sum a_i}$, for real x_i and real, nonnegative a_i.

- Or equivalently if $\sum a_i = 1$: $\phi(\sum a_i x_i) \geq \sum a_i \phi(x_i)$.

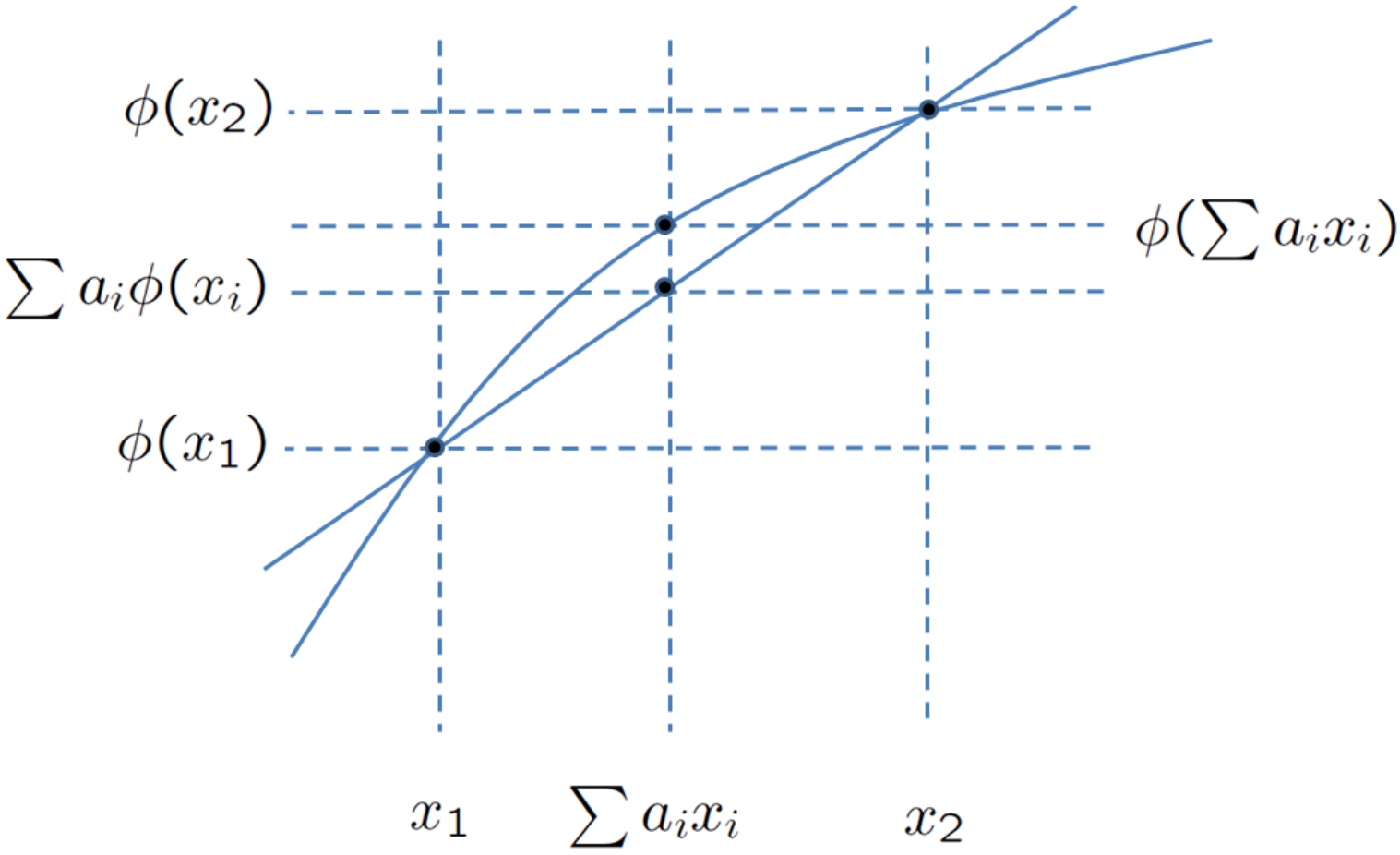
- In text: when taking a weighted average and applying a concave function, the answer is always more (or the same) if we apply the concave function *after* taking the weighted average.

- Remember: judge concavity or convexity of functions *from below*.

- This is only an equality when the x_i are all the same (for general concave functions that don't have straight lines in them).

- For the application of Jensen's inequality in optimization algorithms, remember the x_i always start out the same for all i.

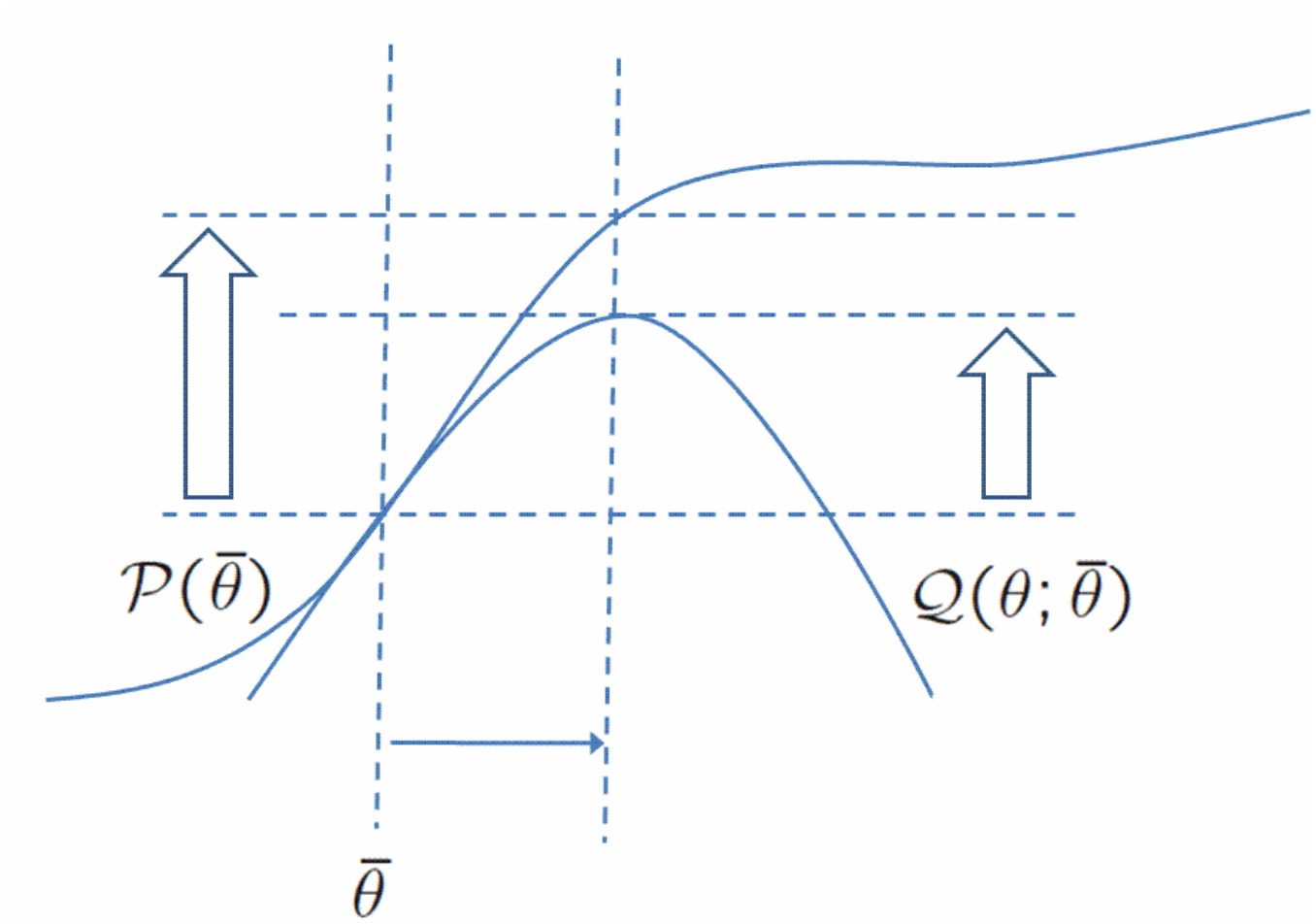
Mixture of Gaussians distribution: Jensen's inequality (picture)



Mixture of Gaussians distribution: auxiliary function

- Derivation of mixture of Gaussians update is a maximization of: $\mathcal{P}(\theta) = \sum_{t=1}^{T} \log \sum_{m=1}^{M} f_m(\theta, t)$, with θ as model parameters.
- $f_m(\theta, t)$ is shorthand for $c_m \mathcal{N}(\mathbf{x}_t; \mu_m, \mathbf{\Sigma}_m)$.
- We will use Jensen's inequality to push the log inside the second summation which makes it the same problem as estimating the Gaussians one by one.
- Need $f_m(\theta, t) = a_m(\bar{\theta}, t) x_m(\theta, \bar{\theta}, t)$, need $x_m(\theta, \bar{\theta}, t)$ to be all the same at current value $\theta = \bar{\theta}$, i.e. $x_m(\bar{\theta}, \bar{\theta}, t) = \mathbf{x}_{m'}(\bar{\theta}, \bar{\theta}, t)$; can arbitrarily stipulate that all the $a_m(\bar{\theta}, t)$ sum to one (since they will get renormalized anyway).
- Define $a_m(\bar{\theta}, t) = \frac{f_m(\bar{\theta}, t)}{\sum_m f_m(\bar{\theta}, t)}$, $x_m(\theta, \bar{\theta}, t) = \frac{f_m(\theta, t)}{a_m(\bar{\theta}, t)}$, satisfies both conditions.
- We can then work out an "auxiliary function" $\mathcal{Q}(\theta; \bar{\theta})$ such that $\mathcal{Q}(\theta; \bar{\theta}) \leq \mathcal{P}(\theta)$ and $\mathcal{Q}(\bar{\theta}; \bar{\theta}) = \mathcal{P}(\bar{\theta})$.
- Thus, if the current parameters are $\bar{\theta}$ we can show that increasing the value of $\mathcal{Q}(\theta; \bar{\theta})$ will increase $\mathcal{P}(\bar{\theta})$ by at least as much.
- $\mathcal{Q}(\theta; \bar{\theta}) = K + \sum_{t=1}^{T} \sum_{m=1}^{M} \gamma_m(t, \bar{\theta}) \log f_m(\theta)$, with $\gamma_m(t, \bar{\theta}) = \frac{f_m(\bar{\theta})}{\sum_m f_m(\bar{\theta})}$.

Mixture of Gaussians distribution: auxiliary function (picture)



Mixture of Gaussians estimation: code (will only work if mean and variances sensibly initialized)

```
void reest_mixture(int D, int M, int T, int iters, const float **data,
    float **mu, float **var, float *weights){
  float *count_stats = new float[M], *loglikes = new float[M];
  float **mu_stats = alloc_matrix(M,D), **var_stats = alloc_matrix(M,D);
  for(int iter=0;iter<iters;iter++){
    // set count_stats, mu_stats and var_stats to zero here!
    for(int t=0;t<T;t++){
      float log_sum=-1.0e+10; // very negative->log(zero)
      for(int m=0;m<M;m++){
        loglikes[m] = diag_loglike(data[t],mu[m],var[m],D)+log(weights[m]);
        log_sum=log_add(log_sum,loglikes[m]);  }// log(exp(a)+exp(b));
      for(int m=0;m<M;m++){
        float gamma_m_t = exp(loglikes[m]-log_sum); // posterior of mix.
        count_stats[m] += gamma_m_t;
        for(int d=0;d<D;d++){
          mu_stats[m][d] += data[t][d]*gamma_m_t;
          var_stats[m][d] += data[t][d]*data[t][d]*gamma_m_t; }}}
    for(int m=0;m<M;m++){
      weights[m] = count_stats[m] / T;
      for(int d=0;d<D;d++){
        mu[m][d] = mu_stats[m][d]/count_stats[m];
        var[m][d] = var_stats[m][d]/count_stats[m] - mu[m][d]*mu[m][d]; }}}}
```

Maximum Likelihood Linear Regression (MLLR)

- Consider the mean transformation $\mu \rightarrow \mathbf{A}\mu + \mathbf{b}$.
- Equivalent to $\mu \rightarrow \mathbf{W}\mu^+$, with $\mathbf{W} = [\mathbf{A}; \mathbf{b}]$ and $\mu^+ = \begin{bmatrix}\mu^T & 1\end{bmatrix}^T$.
- Useful as a way of transforming models to new speakers or conditions using relatively few parameters.
- Likelihood function is: $\mathcal{P}(\mathbf{W}) = \sum_{t=1}^{T} \log \sum_{m=1}^{M} c_m \mathcal{N}(\mathbf{x}_t; \mathbf{W}\mu_m^+, \mathbf{\Sigma}_m)$.
- Auxiliary function is: $\mathcal{Q}(\mathbf{W}; \bar{\mathbf{W}}) = \sum_{t=1}^{T} \sum_{m=1}^{M} \gamma_m(t) \log \left(c_m \mathcal{N}(\mathbf{x}_t; \mathbf{W}\mu_m^+, \mathbf{\Sigma}_m) \right)$, where $\gamma_m(t) = \frac{c_m \mathcal{N}(\mathbf{x}_t; \bar{\mathbf{W}}\mu_m^+, \mathbf{\Sigma}_m)}{\sum_{m'} c_{m'} \mathcal{N}(\mathbf{x}_t; \bar{\mathbf{W}}\mu_{m'}^+, \mathbf{\Sigma}_{m'})}$.

```
void mllr_transform_models(int D, int M, float **means, float **W){
  float *tmp = new float[D];
  for(int m=0;m<M;m++){
    for(int d=0;d<D;d++) tmp[d]=means[m][d];
    for(int d=0;d<D;d++){
      float sum = W[d][D];
      for(int e=0;e<D;e++) sum += tmp[e]*W[d][e];
      means[m][d] = sum; }}}
```

Maximum Likelihood Linear Regression (MLLR): statistics

- Assuming variances $\mathbf{\Sigma}_m$ are diagonal, auxiliary function can be separated per row $\mathbf{w}_d$ of the transform:
  $\mathcal{Q}(\mathbf{W}) = K + \sum_{d=1}^{D} \mathbf{w}_d \cdot \mathbf{k}_d - 0.5\mathbf{w}_d^T \mathbf{G}_d \mathbf{w}_d$.

- Statistics are: $\mathbf{k}_d = \sum_{t,m} \gamma_m(t) \frac{x_{td}}{\sigma^2_{md}} \mu^+_m$, $\mathbf{G}_d = \sum_{t,m} \gamma_m(t) \frac{1}{\sigma^2_{md}} \mu^+_m \mu^{+T}_m$.

```
// If we are not on the first iteration, mu is pre-transformed.
void accu_mllr(int M, int D, int T, float **data, float **mu,
       float **var, float *weights, float ***G, float **k){
   float *loglikes = new float[M];
   for(int t=0;t<T;t++){
      float log_sum=-1.0e+10; float *x=data[t];
      for(int m=0;m<M;m++){
        loglikes[m] = diag_loglike(x,mu[m],var[m],D)+log(weights[m]);
        log_sum=log_add(log_sum,loglikes[m]); }
      for(int m=0;m<M;m++){
        float gamma_m_t = exp(loglikes[m]-log_sum);
        for(int d=0;d<D;d++)
          for(int e=0;e<D+1;e++){
            k[d][e] += gamma_m_t*x[d]/var[m][d] * (e==D?1:mu[m][e]);
            for(int f=0;f<D+1;f++)
              G[d][e][f]+=gamma_m_t/var[m][d]*(e==D?1:mu[m][e])*(f==D?1:mu[m][f]);
          }}}}
```

Maximum Likelihood Linear Regression (MLLR): update

- Update is very simple: $\mathbf{w}_d = \mathbf{G}_d^{-1}\mathbf{k}_d$.
- If any $\mathbf{G}_d$ are not invertible we cannot update (can happen if $\leq D$ means had nonzero counts).

```
// If we are not on the first iteration, mu is pre-transformed.
void update_mllr(int D, float **W_in, float ***G, float **k, int T){
  float tot_objf_impr=0.0;
  float **W = alloc_matrix(D,D+1), **Ginv = alloc_matrix(D+1,D+1);
  for(int d=0;d<D;d++){
    invert_matrix(Ginv,G[d],D+1,D+1); // Ginv:=inverse(G[d]);
    m_v_prod(W[d], Ginv, k[d]); // W[d] := Ginv * k_d.
    float objf_impr=dot_prod(W[d],k[d],D+1) -0.5*vmv_prod(W[d],G[d],W[d],D+1,D+1)
      -(k[d][d] -0.5 G[d][d][d]); // Subtract same with ‘‘default’’ matrix row.
    Assert(objf_impr>=0);
    tot_objf_impr += objf_impr;
  }
  printf("Objective improvement is %f\n", tot_objf_impr/T);
  float **W_in_plus = alloc_matrix(D+1,D+1); W_in_plus[D][D] = 1.0;
  for(int d=0;d<D;d++) for(int e=0;e<D+1;e++) W_in_plus[d][e]=W_in[d][e];
  m_m_prod(W_in, W, W_in_plus, D,D+1,D+1); // W_in := W*(W_in with extra row..)
}
```

Constrained Maximum Likelihood Linear Regression (MLLR)

- Consider the feature transformation: $\mathbf{x} \rightarrow \mathbf{A}\mathbf{x} + \mathbf{b}$.
- Equivalent to $\mathbf{x} \rightarrow \mathbf{W}\mathbf{x}^+$, with $\mathbf{W} = [\mathbf{A}; \mathbf{b}]$ and $\mathbf{x}^+ = \begin{bmatrix}\mathbf{x}^T & 1\end{bmatrix}^T$.
- We have obtained a Gaussian Mixture Model somehow, and want to train $\mathbf{W}$ to maximize data likelihood on a data sequence $\mathcal{X} = \mathbf{x}_1 \ldots \mathbf{x}_T$.
- Likelihood function is: $\mathcal{P}(\mathbf{W}) = \sum_{t=1}^{T} \log|\det \mathbf{A}| + \log \sum_{m=1}^{M} c_m \mathcal{N}(\mathbf{W}\mathbf{x}_t^+; \mu_m, \mathbf{\Sigma}_m)$.
- Need for the extra term $\log|\det \mathbf{A}|$ can be derived from viewing $\mathbf{A}$ as a model transformation (or considering effect on term $d^n\mathbf{x}$ in expression to derive probabilities from likelihoods).
- Auxiliary function is: $\mathcal{Q}(\mathbf{W}; \bar{\mathbf{W}}) = \sum_{t=1}^{T} \sum_{m=1}^{M} \gamma_m(t) \log\Big(c_m \mathcal{N}(\mathbf{W}\mathbf{x}_t^+; \mu_m, \mathbf{\Sigma}_m)\Big)$, where $\gamma_m(t) = \frac{c_m \mathcal{N}(\bar{\mathbf{W}}\mathbf{x}_t^+; \mu_m, \mathbf{\Sigma}_m)}{\sum_{m'} c_{m'} \mathcal{N}(\bar{\mathbf{W}}\mathbf{x}_t^+; \mu_{m'}, \mathbf{\Sigma}_{m'})}$.

```
float do_fmllr_transform(int D, float *Wx, const float **W, const float **x){
  for(int d=0;d<D;d++){
     float sum=W[d][D];
     for(int e=0;e<D;e++) sum += W[d][e] * x[e];
     Wx[d] = sum; }}
```

Constrained MLLR: accumulation

- If $\mathbf{\Sigma}_m$ are diagonal, $\mathcal{Q}(\mathbf{W}; \bar{\mathbf{W}})$ this can be separated out per row of the transform $\mathbf{w}_d$ (note, $\mathbf{w}_d$ is column vector) to get:
- $\mathcal{Q}(\mathbf{W}; \bar{\mathbf{W}}) = K + \beta |\det A| + \sum_{d=1}^{D} \mathbf{w}_d^T \mathbf{k}_d - 0.5 \mathbf{w}_d^T \mathbf{G}_d \mathbf{w}_d$.
- Sufficient statistics are: $\beta = T, \mathbf{G}_d = \sum_{t,m} \gamma_m(t) \frac{1}{\sigma_{m,d}^2} \mathbf{x}_t^+ \mathbf{x}_t^{+T}$, $\mathbf{k}_d = \sum_{t,m} \gamma_m(t) \frac{\mu_{m,d}}{\sigma_{m,d}^2} \mathbf{x}_t^+$.

```
void accu_fmllr(int M, int D, int T, float **W, float **data, float **mu,
      float **var, float *weights, float ***G, float **k){
  float *Wx = new float[D], *loglikes = new float[M];
  for(int t=0;t<T;t++){
     float log_sum=-1.0e+10; float *x=data[t];
     do_fmllr_transform(Wx, W, x); // Wx = W*x^+
     for(int m=0;m<M;m++){
       loglikes[m] = diag_loglike(Wx,mu[m],var[m],D)+log(weights[m]);
       log_sum=log_add(log_sum,loglikes[m]); }
     for(int m=0;m<M;m++){
       float gamma_m_t = exp(loglikes[m]-log_sum);
       for(int d=0;d<D;d++) for(int e=0;e<D+1;e++){
         k[d][e] += gamma_m_t*mu[m][d]/var[m][d] * (e==D?1:x[e]);
         for(int f=0;f<D+1;f++)
           G[d][e][f]+=gamma_m_t/var[m][d]*(e==D?1:x[e])*(f==D?1:x[f]); }}}}
```

Constrained MLLR: update (1 of 2)

- Estimation given statistics β, G_d and $\mathbf{k}_d$ is iterative, row by row.
- Each time we work out the optimal value for a row given the other rows.
- Log-determinant term: use the fact that $\log|\det\mathbf{A}| = \log|\det\bar{\mathbf{A}}| + \log|\det\mathbf{B}|$, where $\mathbf{B} = \mathbf{A}\bar{\mathbf{A}}^{-1}$.
- If only d'th row of $\mathbf{A}$ differs from $\bar{\mathbf{A}}$, only d'th row of $\mathbf{B}$ is non-unit and $\det\mathbf{B} = \mathbf{B}_{dd} = \mathbf{a}_d \cdot \mathbf{c}_d$ where $\mathbf{a}_d$ is d'th row of $\mathbf{A}$ and $\mathbf{c}_d$ is d'th column of $\bar{\mathbf{A}}^{-1}$. Can work this out from recursive determinant formula.
- Ignoring constant terms, auxiliary function in d'th row of $\mathbf{W}$, i.e. $\mathbf{w}_d$, is: $\mathcal{Q}(\mathbf{w}_d) = \beta \log|\mathbf{w}_d \cdot \mathbf{c}_d^{+0}| + \mathbf{w}_d \cdot \mathbf{k}_d - 0.5\mathbf{w}_d^T\mathbf{G}_d\mathbf{w}_d$.
- Differentiating w.r.t. $\mathbf{w}_d$, transposing and setting to zero: $\frac{\beta}{\mathbf{w}_d \cdot \mathbf{c}_d^{+0}}\mathbf{c}_d^{+0} + \mathbf{k}_d - \mathbf{G}_d\mathbf{w}_d = 0$.
- Defining $f = \frac{\beta}{\mathbf{w}_d \cdot \mathbf{c}_d^{+0}}$, can work out $\mathbf{w}_d = \mathbf{G}_d^{-1}(\mathbf{k}_d + f\mathbf{c}_d^{+0})$.
- Substituting into definition of f, $f = \frac{\beta}{\mathbf{c}_d^{+0T}\left(\mathbf{G}_d^{-1}(\mathbf{k}_d + f\mathbf{c}_d^{+0})\right)}$. ...

Constrained MLLR: update (2 of 2)

- Rearranging: $f\mathbf{c}_d^{+0T}\mathbf{G}_d^{-1}\mathbf{k}_d + f^2\mathbf{c}_d^{+0T}\mathbf{G}_d^{-1}\mathbf{c}_d^{+0} - \beta = 0$. Quadratic in f.

- Defining $a = \mathbf{c}_d^{+0T}\mathbf{G}_d^{-1}\mathbf{c}_d^{+0}, b = \mathbf{c}_d^{+0T}\mathbf{G}_d^{-1}\mathbf{k}_d, c = -\beta$, we have $f = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$. Safe to take plus sign only.

- Then put f into the formula for $\mathbf{w}_d$ to work out the updated row.

Constrained MLLR: update code

```
void update_fmllr(int D, float **W, float **k, float ***G, float beta){
  float tot_objf_change=0;
  float **Ainv = alloc_matrix(D,D); float **Ginv = alloc_matrix(D+1,D+1);
  float *c0=new float[D+1],*tmp=new float[D+1],*wnew=new float[D+1]; c0[D]=0.0;
  for(int iter=0;iter<10;iter++){
    for(int d=0;d<D;d++){ // iterate over rows.
      invert_matrix(Ainv,W,D,D);  // Ainv := inverse(W[0:D-1,0:D-1])
      invert_matrix(Ginv,G[d],D+1,D+1); // Ginv:=inverse(G[d])
      for(int e=0;e<D;e++) c0[d] = Ainv[e][d]; // c0 := d'th column of Ainv.
      float a=vmv_prod(c0,Ginv,c0,D+1,D+1), b=vmv_prod(c0,Ginv,k[d],D+1,D+1),
         c=-beta, f = (-b + sqrt(b*b - 4*a*c))/(2*a);
      Assert(f>0);
      for(int e=0;e<D+1;e++) tmp[e]=f*c0[e] + k[d][e]; // tmp := k[d] + f*c0.
      m_v_prod(wnew,Ginv,tmp,D+1,D+1); // wnew:=Ginv*tmp.
      float objf_change += beta*log(beta/f)  // log-det term
           + dot_prod(wnew,k[d],D+1) - dot_prod(W[d],k[d],D+1)
           -0.5*(vmv_prod(wnew,G[d],wnew)-vmv_prod(W[d],G[d],W[d]));
      Assert(objf_change >= -1.0e-05); tot_objf_change += objf_change;
      for(int e=0;e<D+1;e++) W[d][e] = wnew[e];
    }
    printf("On iter %d, cumulative objf change is %f\n", iter, tot_objf_change);
  }
}
```

Semi-tied covariance transform (STC)

- Related to other techniques: HLDA (Heteroscedastic Linear Discriminant Analysis), which is a dimension-reduction form of STC; MLLT (which is an alternative basically equivalent formulation of a globally shared STC).
- Transformation: normally applied as $\mathbf{x} \rightarrow \mathbf{A}\mathbf{x}, \mu \rightarrow \mathbf{A}\mu$. Transform the features and means with the same transformation. Used for training (not adaptation).
- Equivalently, $\mathbf{\Sigma} \rightarrow \mathbf{A}^T\mathbf{\Sigma}\mathbf{A}$. Only useful if $\mathbf{\Sigma}$ is diagonal.
- Likelihood function is: $\mathcal{P}(\mathbf{A}) = \sum_{t=1}^{T} \log|\det \mathbf{A}| + \log \sum_{m=1}^{M} c_m \mathcal{N}(\mathbf{A}\mathbf{x}_t; \mathbf{A}\mu_m, \mathbf{\Sigma}_m)$.
- Auxiliary function is: $\mathcal{Q}(\mathbf{A}; \bar{\mathbf{A}}) = \sum_{t=1}^{T} \sum_{m=1}^{M} \gamma_m(t) \log\left(c_m \mathcal{N}(\mathbf{A}\mathbf{x}_t; \mathbf{A}\mu_m, \mathbf{\Sigma}_m)\right)$, where $\gamma_m(t) = \frac{c_m \mathcal{N}(\bar{\mathbf{A}}\mathbf{x}_t; \bar{\mathbf{A}}\mu_m, \mathbf{\Sigma}_m)}{\sum_{m'} c_{m'} \mathcal{N}(\bar{\mathbf{A}}\mathbf{x}_t; \bar{\mathbf{A}}\mu_{m'}, \mathbf{\Sigma}_{m'})}$.
- Sufficient statistics are the $D \times D$ matrices $\mathbf{G}_d$, $1 \leq d \leq D$: $\beta = T, \mathbf{G}_d = \sum_{t=1}^{T} \gamma_m(t)(\mathbf{x}_t - \mu_m)(\mathbf{x}_t - \mu_m)^T \sigma_{md}^2$. Like simplified constrained MLLR.
- Auxiliary function $\mathcal{Q}(\mathbf{A}; \bar{\mathbf{A}}) = \beta \log|\det \mathbf{A}| - 0.5 \sum_{d=1}^{D} \mathbf{a}_d^T \mathbf{G}_d \mathbf{a}_d$, if $\mathbf{a}_d$ are the rows of $\mathbf{G}$.

Semi-tied covariance transform (STC): accumulation code

- Assumes any existing STC transform has already been applied to the means.

```
void stc_accu(int M, int D, int T, float **data, float **A, float **mu,
              float **var, float *weights, float ***G){
  float *Ax = new float[D], *loglikes = new float[M];
  for(int t=0;t<T;t++){
    float log_sum=-1.0e+10; float *x=data[t];
    do_stc_transform(Ax, A, x); // Ax = A*x^+
    m_v_prod(Ax, A, x, D,D); // Ax := A * x;
    for(int m=0;m<M;m++){
      loglikes[m] = diag_loglike(Ax,mu[m],var[m],D)+log(weights[m]);
      log_sum=log_add(log_sum,loglikes[m]); }
    for(int m=0;m<M;m++){
      float gamma_m_t = exp(loglikes[m]-log_sum);
      for(int d=0;d<D;d++)
        for(int e=0;e<D;e++)
          for(int f=0;f<D;f++) // in reality would do f<=e: symmetric.
            G[d][e][f]+=gamma_m_t*(Ax[e]-mu[m][e])*(Ax[f]-mu[m][f])
              /var[m][d];
    }
  }
}
```

Semi-tied covariance transform (STC): update

- Derivation and update is the same as fMLLR except no terms $\mathbf{k}_d$.
- Since the statistics above were estimated with the *transformed* data and means, we are estimating a new transform $\tilde{\mathbf{A}}$ that is applied after any existing transform.
- We estimate $\tilde{\mathbf{A}}$ starting from the unit matrix, and will multiply $\mathbf{A} := \tilde{\mathbf{A}}\mathbf{A}$.
- $\mathbf{A}$ left-multplies features so $\tilde{\mathbf{A}}$ going *after* $\mathbf{A}$ means $\tilde{\mathbf{A}}$ needs to be on the left. (Imagine an $\mathbf{x}$ on the right: $\tilde{\mathbf{A}}\mathbf{A}\mathbf{x}$).
- We need to transform the means but they will already be transformed by the existing part of the transform $\mathbf{A}$ so we need to multiply only by the new part $\tilde{\mathbf{A}}$.
- This is only one way of implementing STC and it does not converge very fast.
- Alternative methods (e.g. used in HLDA estimation) are based on accumulating full-covariance statistics $\gamma_m, \mathbf{m}_m, \mathbf{S}_m$ for each Gaussian and within memory, alternating the accumulation and update phase we describe (accumulating from the statistics), with updating the model's diagonal variances. But this is memory intensive for large models.

Semi-tied covariance transform (STC): update code

```
void stc_upd(int D, float **A_in, float ***G, float beta, float **means, int M){
  float **A = alloc_matrix(D,D), **Ainv = alloc_matrix(D,D),
    **GInv = alloc_matrix(D,D);
  float *cd=new float[D],*tmp=new float[D],*anew=new float[D];
  for(int d=0;d<D;d++) A[d][d]=1.0; // Set A to unit matrix.
  float tot_objf_change=0;
  for(int iter=0;iter<10;iter++){
    for(int d=0;d<D;d++){
      invert_matrix(Ainv,W,D,D);  // Ainv := inverse(A);
      invert_matrix(Ginv,G[d],D,D); // Ginv:=inverse(G[d]);
      for(int e=0;e<D;e++) cd[e] = Ainv[e][d]; // c := d'th column of Ainv.
      float a=vmv_prod(cd,Ginv,cd,D,D), c=-beta, f=(-0 + sqrt(0*0 -4*a*c))/(2*a);
      for(int e=0;e<D;e++) tmp[e]=f*cd[e]; // tmp := c*f.
      m_v_prod(anew,Ginv,tmp,D,D); // anew:=Ginv*tmp.
      float objf_change = beta*log(beta/f) // log-det term
        -0.5*(vmv_prod(anew,G[d],anew)-vmv_prod(A[d],G[d],A[d]));
      Assert(objf_change >= 0); tot_objf_change += objf_change;
      for(int e=0;e<D;e++) A[d][e] = anew[e];
    }} // should print objf value on each iter.
  for(int m=0;m<M;m++) // Transform means by new part of transform.
    m_v_prod(means[m],A,means[m],D,D); //assume function works w/ repeat args.
  // Assume A_in is unit if first iteration.
  m_m_prod(A_in, A, A_in, D,D,D); // A_in:=A*A_in.  Assume works w/ repeat args.
}
```

Speaker adaptive training for MLLR

- For most speaker adaptation techniques, Speaker Adaptive Training (SAT) simply means training the HMM on appropriately adapted features (e.g. Constrained MLLR).
- This ensures models that are "compatible" with the form of adaptation.
- For (unconstrained) MLLR, it is different.
- Consider training a single GMM on speakers $s = 1 \dots S$, with mean transforms $\mathbf{W}^{(s)}$.
- Each speaker has training samples $\mathcal{X}^{(s)} = \mathbf{x}_1^{(s)} \dots \mathbf{x}_{N^{(s)}}^{(s)}$.
- Likelihood function in means $\mathcal{M} = \{\mu_1 \dots \mu_M\}$ is: $\mathcal{P}(\mathcal{M}) = \sum_{s,t} \log \sum_{m=1}^{M} c_m \mathcal{N}(\mathbf{x}_t^{(s)}; \mathbf{W}^{(s)} \mu_m^+, \mathbf{\Sigma}_m)$.
- Auxiliary function is $\mathcal{Q}(\mathcal{M}) = K + \sum_{s,t,m} \gamma_m^{(s)}(t) \mathcal{N}(\mathbf{x}_t^{(s)}; \mathbf{W}^{(s)} \mu_m^+, \mathbf{\Sigma}_m)$.
- $\mathcal{Q}(\mathcal{M}) = K' - 0.5 \sum_{s,t,m} \gamma_m^{(s)}(t) (\mathbf{x}^{(s)} - \mathbf{W}^{(s)} \mu_m^+)^T \mathbf{\Sigma}_m^{-1} (\mathbf{x}^{(s)} - \mathbf{W}^{(s)} \mu_m^+)$.
- Statistics: for each mixture m, store linear term $\mathbf{v}_m = \sum_{s,t,m} \gamma_m^{(s)}(t) \mathbf{A}^{(s)^T} \mathbf{\Sigma}_m^{-1} (\mathbf{x}^{(s)} - \mathbf{b}^{(s)})$ and quadratic term $\mathbf{G}_m = \sum_{s,t,m} \gamma_m^{(s)} \mathbf{A}^{(s)^T} \mathbf{\Sigma}_m^{-1} \mathbf{A}^{(s)}$.

Speaker adaptive training for MLLR- mean-stats accumulation code

```
// call this for each speaker.
void accu_mllr_sat_mean(int T, int M, int D, float **data, float **W,
      float **mu, float **var, float **v, float ***G){
  // mu provided to this function is pre-transformed mean (W mu^+).
  float *loglikes=new float[M], *offset=new float[D], vtmp=new float[D];
  for(int t=0;t<T;t++){
    float log_sum=-1.0e+10; // very negative->log(zero)
    for(int m=0;m<M;m++){
      loglikes[m] = diag_loglike(data[t],mu[m],var[m],D)+log(weights[m]);
      log_sum=log_add(log_sum,loglikes[m]);  }// log(exp(a)+exp(b));
    for(int m=0;m<M;m++){
      float gamma_m_t = exp(loglikes[m]-log_sum); // posterior of mix.
      for(int d=0;d<D;d++) // offset=Sigma^{-1} * (x-b)
    offset[d]=(data[t][d]-w[d][D])/var[m][d];
  m_v_prod(vtmp,W,offset,D,D); // vtmp:=A*offset.
      for(int d=0;d<D;d++) v[m] += gamma_m_t*vtmp;
      for(int d=0;d<D;d++)
        for(int e=0;e<D;e++) //
          for(int f=0;f<D;f++) // G += gamma_m_t * A^T Sigma^{-1} A
    G[m][e][f] += (gamma_m_t/var[m][d])*W[d][e]*W[d][f];
    }
  }
}
```

Speaker adaptive training for MLLR continued

- Mean update: $\mu_m = \mathbf{G}_m^{-1}\mathbf{v}_m$.

```
invert_matrix(G[m], Ginv, D,D);
m_v_prod(mu[m], Ginv, v[m], D,D);
```

- Variance: accumulation and update must be done separately from mean (theoretically)

- Variance statistics: $\gamma_m = \sum_{s,t,m} \gamma_m^{(s)}(t)$, $\mathbf{S}_m = \sum_{s,t,m}(\mathbf{x}^{(s)} - \mathbf{W}^{(s)}\mu_m^+)(\mathbf{x}^{(s)} - \mathbf{W}^{(s)}\mu_m^+)^T$

```
// full-variance case:
S[m][d][e] += gamma_m_t*(data[t][d]-mu[m][d])*(data[t][e]-mu[m][e]);
//or, diagonal case:
S[m][d] += gamma_m_t*(data[t][d]-mu[m][d])*(data[t][d]-mu[m][d]);
```

- Variance update: $\mathbf{\Sigma}_m = \frac{1}{\gamma_m}\mathbf{S}_m$.

```
var[m][d][e] = S[m][d][e]/count[m]; // full case.
var[m][d] = S[m][d]/count[m];  // Diagonal case.
```

Maximum A Posteriori (MAP) training.

- Maximum A Posteriori (MAP) has a generic meaning, from Bayesian statistics:
  - Refers to estimating something with a point estimate *given evidence*, i.e. after seeing some kind of observation (combining it with prior).
  - E.g. choosing x to maximizie $P(x|y) = P(x)P(y|x)$ if y is some kind of observation.
- MAP has a specific meaning in speech (if used wihtout further explanation): refers to adapting the mean and variance to a new speaker or condition, but "backing off" to the original parameters if there is not enough data.
- Original paper is by Gauvain and Lee and described a rather complicated technique.
- When people refer to MAP in speech they often mean a simplified "Cambridge-style" MAP that uses a parameter called τ to control backoff for means (an option in HTK code).

Maximum A Posteriori (MAP) training: HTK-style/Cambridge-style MAP.

- HTK code contains the following update for means:

$$\widehat{\mu} = \frac{\tau\mu + \mathbf{x}}{\tau + \gamma}$$

  where $\gamma, \mathbf{x}, \mathbf{S}$ are zeroth,first and second order statistics.

- Controlled by parameter τ (equivalent to a number of frames/observations), if $\tau = 10$, it takes 10 observations before we go halfway to the "data" estimate.

- Variance update (not in HTK code, but in a similar style), would be:

$$\widehat{\sigma_d^2} = \frac{\tau(\sigma_d^2 + (\mu_d - \widehat{\mu}_d)^2) + (s_{dd} - \mathbf{x}_d\mu_d + \gamma\widehat{\mu}_d^2)}{\tau + \gamma}$$

  .

- Think of it as adding "fake statistics" with count τ and same mean and variance as the Gaussian we are backing off to: $\widehat{\mathbf{S}} = \mathbf{S} + \tau(\Sigma + \mu\mu^T)$, $\widehat{\mathbf{x}} = \mathbf{x} + \tau\mu$, $\widehat{\gamma} = \gamma + \tau$. (Only true if using same τ for means and variances).

- Can easily imagine a similar update for mixture weights. In a mixture with M components, a sensible update might be: $\widehat{c}_m = \frac{\gamma_m + M\tau c_m}{M\tau + \sum_m \gamma_m}$.

Maximum A Posteriori (MAP) training: code

```
// Note: only the mean part of this is ‘‘standard’’ but the rest is reasonable.
void map_update(int M, int D, float **mu, float **var, float *weight,
     float **mu_stats, float **var_stats, float **count_stats,
     float tau_means=10, float tau_vars=20, tau_weights=10){
  float tot_count=0;
  for(int m=0;m<M;m++) tot_count+=count_stats[m];
  for(int m=0;m<M;m++){
    weight[m] = (count_stats[m] + tau_weights*M*weight[m])
                 / (tot_count + tau_weights*M);
    for(int d=0;d<D;d++){
      float mu_hat = (mu_stats[m][d] + tau*mu[m][d]) / (count_stats[m]+tau);
      float backoff_term = tau*(var[m][d]+(mu_hat-mu[m][d])*(mu_hat-mu[m][d]));
  float data_term = (var_stats[m][d] - 2*mu_hat*mu_stats[m][d]
                            + gamma*mu_hat*mu_hat);
      var[m][d] = (backoff_term + data_term)/(tau*gamma);
      mu[m][d] = mu_hat;
    }
  }
}
```