ECE 417 Lecture 20: MP5 Walkthrough

10/31/2019

Outline

- Background things that are done for you
  - Observations: mel-frequency cepstral coefficients (MFCC)
  - Token to type alignment
- Gaussian surprisal: set_surprisal
- Scaled Forward-Backward Algorithm: set_alphahat, set_betahat
- E-step: set_gamma, set_xi
- M-step: set_mu, set_var, set_tpm

Done for you: Mel Frequency Cepstral Coefficients (MFCC)

What you need to know:

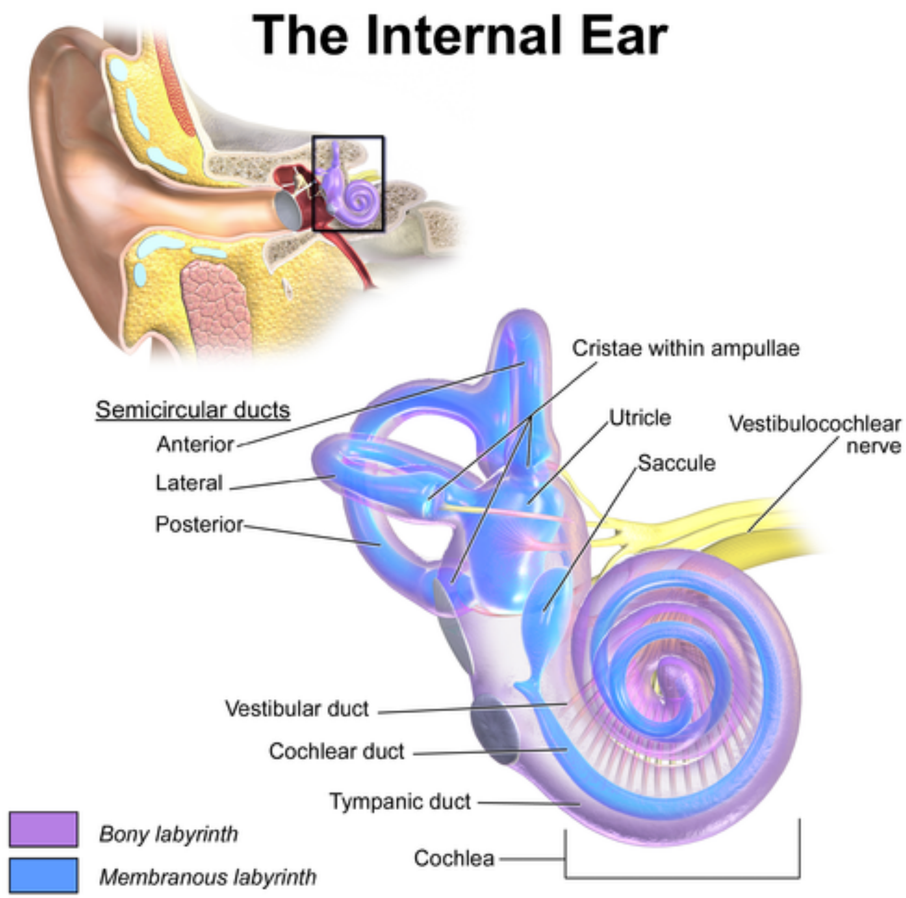
- MFCC is a **<u>low-dimensional vector</u>** (13 dimensions) that keeps most of the speech-relevant information from the MSTFT (**<u>magnitude short-time Fourier transform</u>**, 257 dimensions).

What you don't need to know, but here's the information in case you're interested: **<u>How it's done</u>**.

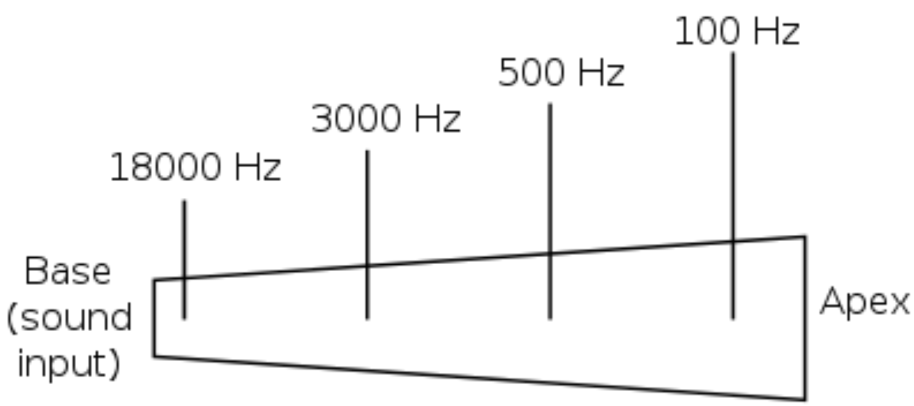
1. Compute the MSTFT, $X[t,k] = \left|X_t\left(e^{j\frac{2\pi k}{N}}\right)\right|$
2. Modify the frequency scale (human perception of pitch).
3. Take the logarithm (human perception of loudness).
4. Take the DCT (approximately decorrelates the features).

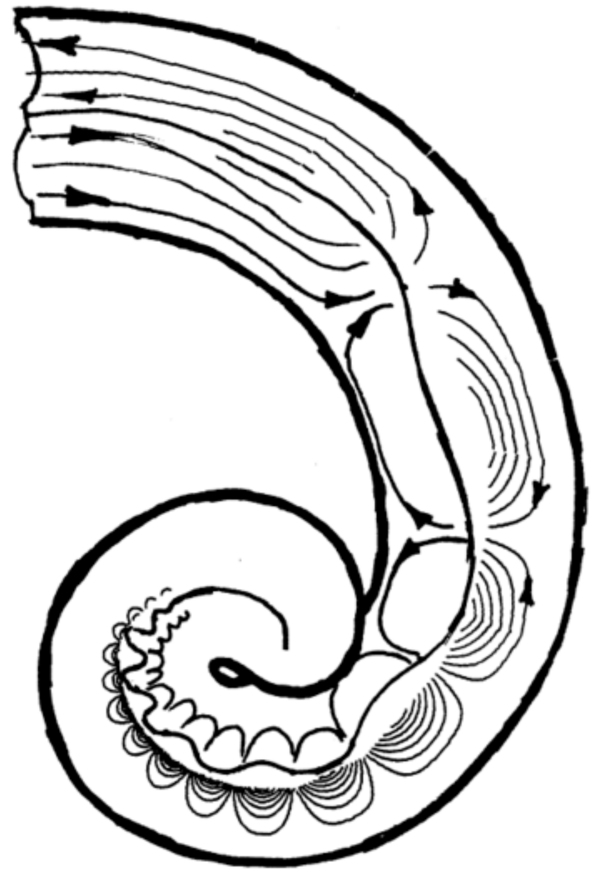
What frequency scale do people hear?

Inner ear

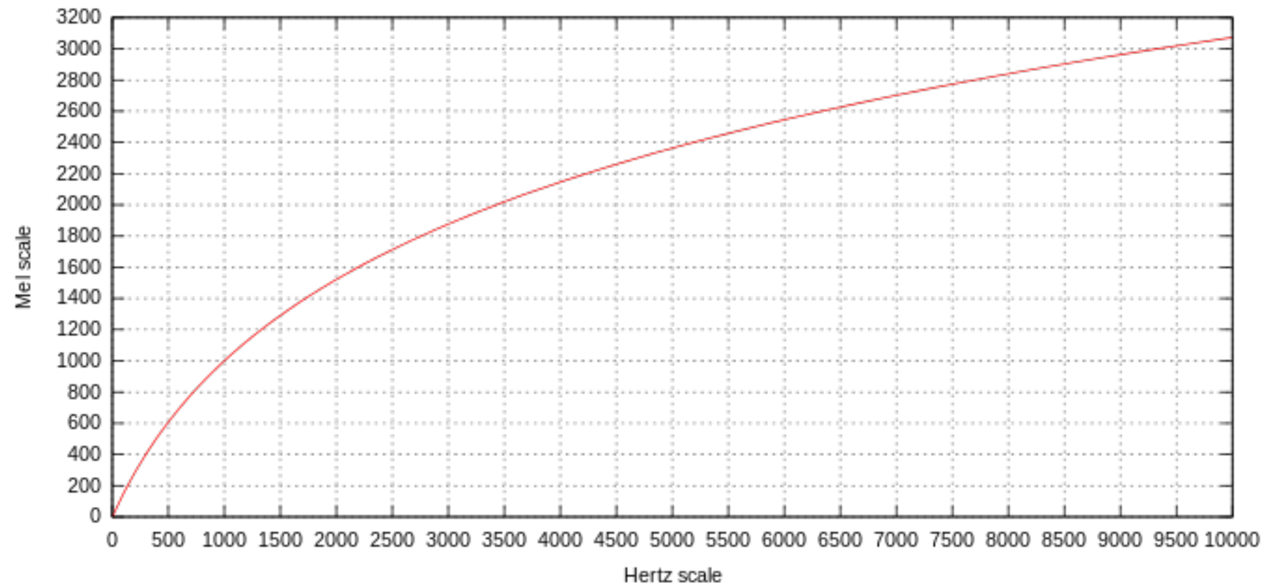

The Internal Ear
Cristae within ampullae
Semicircular ducts
Anterior
Lateral
Posterior
Utricle
Saccule
Vestibulocochlear nerve
Vestibular duct
Cochlear duct
Tympanic duct
Cochlea
Bony labyrinth
Membranous labyrinth

Basilar membrane of the cochlea = a bank of mechanical bandpass filters





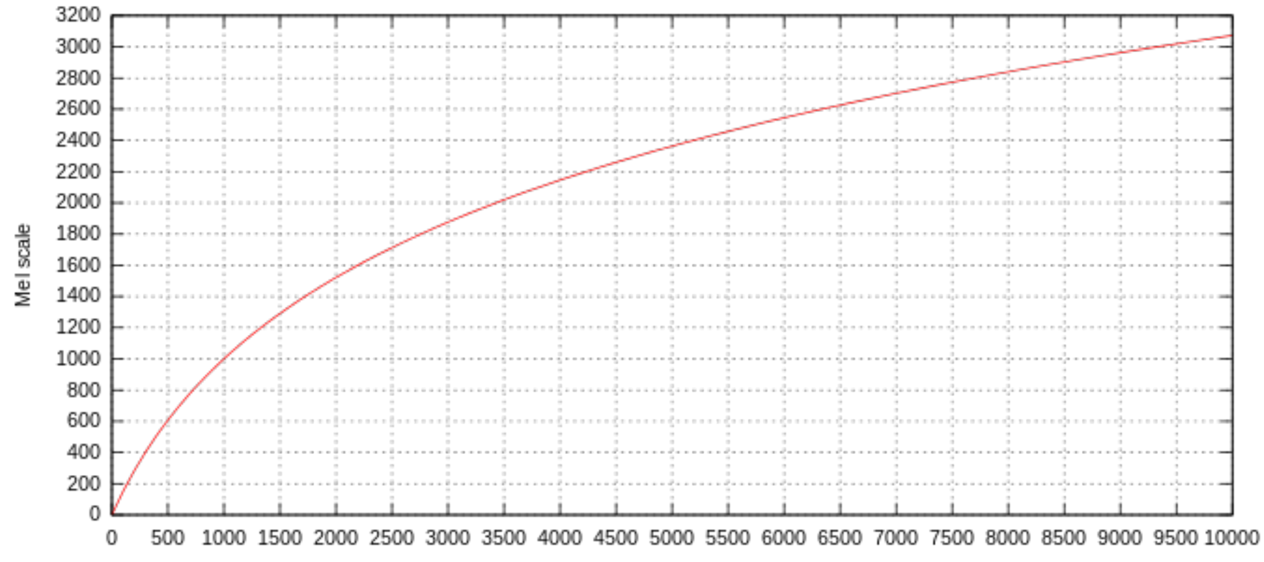
Mel-scale

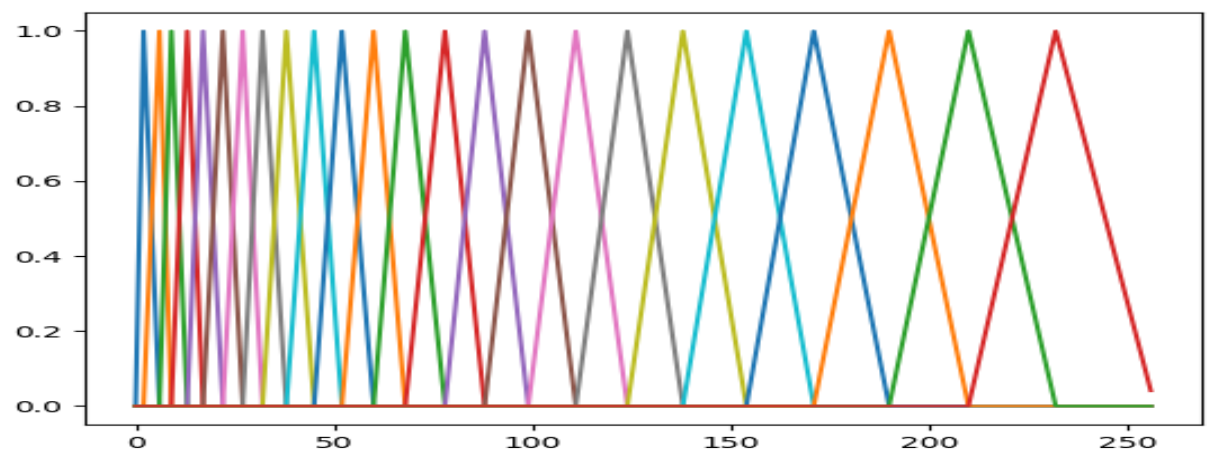


- The experiment:
  - Play tones A, B, C
  - Let the user adjust tone D until pitch(D)-pitch(C) sounds the same as pitch(B)-pitch(A)
- Analysis: create a frequency scale m(f) such that m(D)-m(C) = m(B)-m(A)
- Result: $m(f) = 2595\log_{10}\left(1 + \frac{f}{700}\right)$

Mel-scale filterbanks

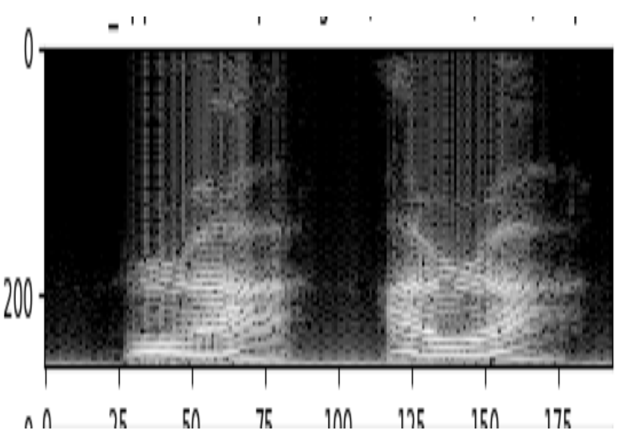
- Define filters such that each filter has a width equal to about 200 mels
- As a function of Hertz: narrow filters at low frequency, wider at high frequency





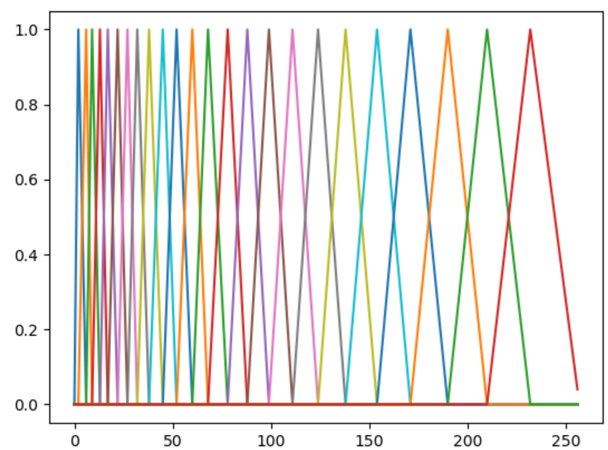
Mel-frequency filterbank features

Suppose X is a matrix representing the MSTFT, $X[t,k] \ = \ |X_t(e^{j\frac{2\pi k}{N}})|$. We can compute the filterbank features as $F = XH$, where H is the matrix of bandpass filters shown here:



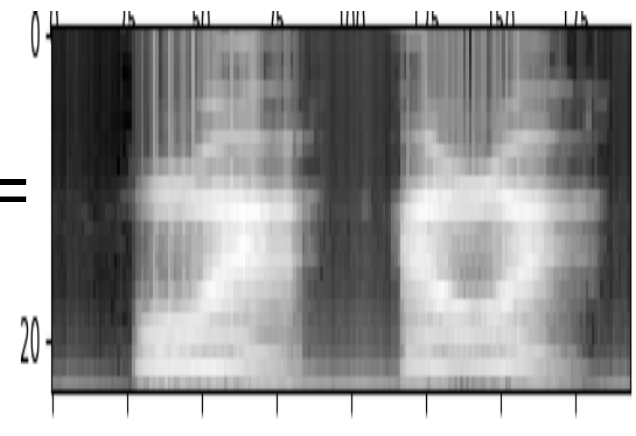
MSTFT, X
(an NFRAMESx257 matrix)

$\times$



Triangle filters, H
(a 257x24 matrix)

$=$



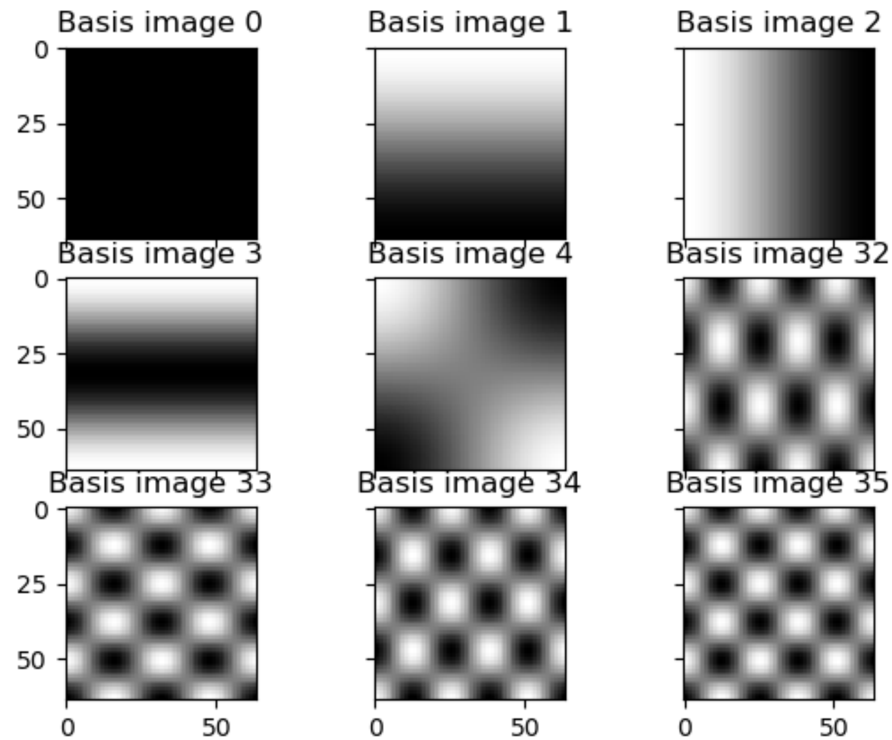
Filterbank features, $F = XH$
(an NFRAMESx24 matrix)

How can we decorrelate the features?
Answer: DCT!

Remember, the 2D DCT looked like this…

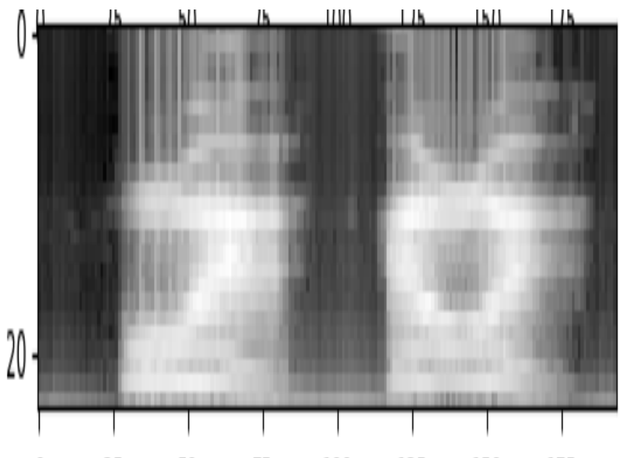
$$\cos\left(\frac{\pi k_1\left(n_1+\frac{1}{2}\right)}{N_1}\right)\cos\left(\frac{\pi k_2\left(n_2+\frac{1}{2}\right)}{N_2}\right)$$

With a 36th order DCT (up to k1=5,k2=5), we can get a bit more detail about the image.

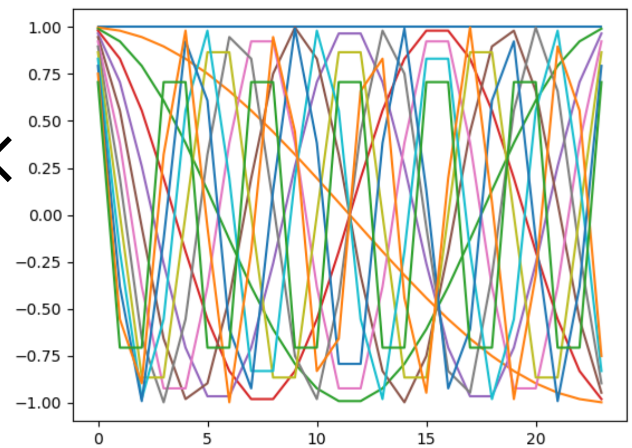


The 1D DCT looks like this:

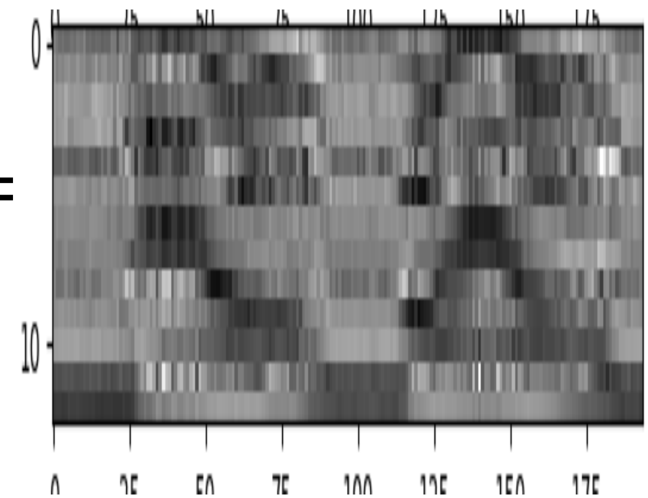
Suppose F is a matrix representing the mel-scale filterbank features, $F = XH$. We can compute the mel-frequency cepstral coefficients (MFCC) as $M = \ln(F)T$, where T is the DCT matrix:



×



=



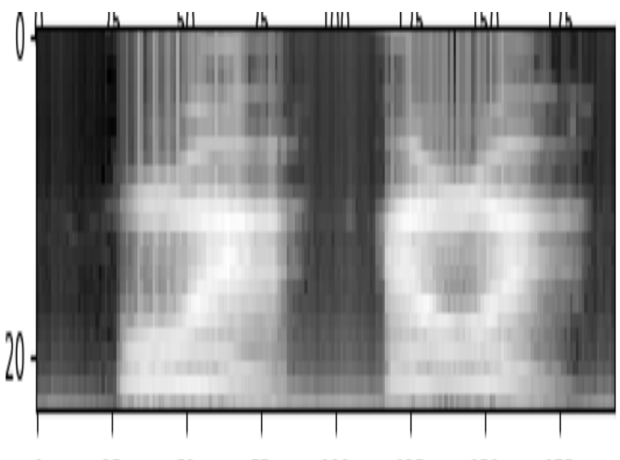
Log Filterbank features, $\ln F$
(an NFRAMESx24 matrix)

DCT matrix, T
(a 24x13 matrix)

MFCC, $\mathrm{M} = \ln(F)T$
(an NFRAMESx13 matrix)

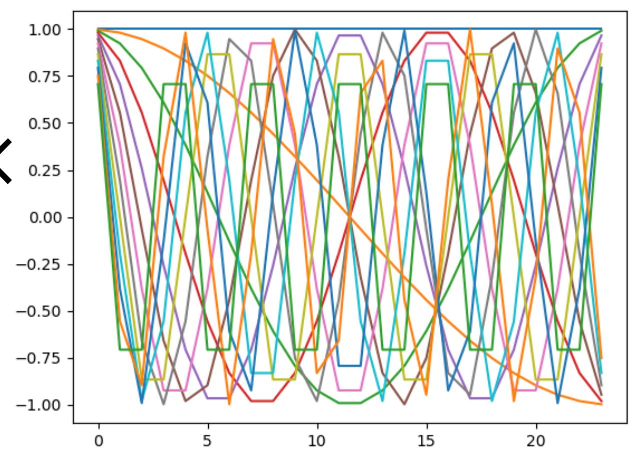
DCT works like PCA!! That’s why we use it.

- Filterbank features (left): neighboring frequency bands are highly correlated.
- MFCC (right): different cepstral coefficients are nearly uncorrelated.



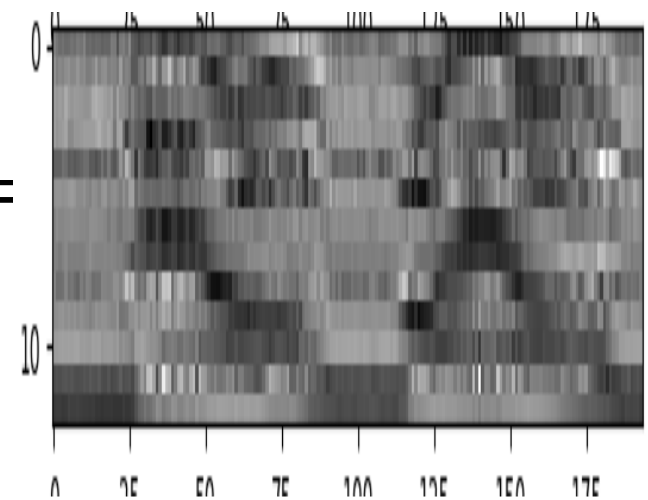
Log Filterbank features, $\ln F$
(an NFRAMESx24 matrix)

×



DCT matrix, T
(a 24x13 matrix)

=



MFCC, $\mathrm{M} = \ln(F)T$
(an NFRAMESx13 matrix)

Outline

- Background things that are done for you
  - Observations: mel-frequency cepstral coefficients = f(MSTFT)
  - Token to type alignment
- Gaussian surprisal, a.k.a. information: set_surprisal
- Scaled Forward-Backward Algorithm: set_alphahat, set_betahat
- E-step: set_gamma, set_xi
- M-step: set_mu, set_var, set_tpm

Token-to-type alignment

- We talked about it a great deal in Tuesday’s lecture.
- Here’s the code that does it:
  - self.model[‘phones’] = ' aelmnoruø‖ɘɤɨɯɵɹɺʉʘʙ’
  - self.tok2type = [str.find(self.model['phones'],x) for x in self.toks]

This defines the types (distinct phones that are present in the training data)

This creates an array tok2type:tok→type

submitted.py

```
def get_params_for_utt(self,u):
    '''Get local model parameters for the u'th utterance'''
    #
    # types[i] = type ID of the i'th tok in utterance[u]
    types = self.tok2type[self.starttok[u]:self.endtok[u]]
    #
    # mu[i,:] = mean vector of the i'th token in utterance[u]
    mu = self.model['mu'][types,:]
    #
    # var[i,:] = vector of variances of the i'th token in utterance u
    var = self.model['var'][types,:]
    #
    # A[i,j] = probability of a transition from the i'th to the j'th tok in utterance u
    A = np.array([[ self.model['tpm'][i,j] for j in types] for i in types])
    #
    return(mu,var,A,types)
U:**-  submitted.py   36% L113  Git:master  (Python)
```

This code cuts out the tok2type array for a particular utterance, u, and then computes:

- mu: matrix of mean vectors
- var: matrix of variance vectors
- A: transition probabilities among the tokens of the utterance

Outline

- Background things that are done for you
  - Observations: mel-frequency cepstral coefficients = f(MSTFT)
  - Token to type alignment
- Gaussian surprisal, a.k.a. information: set_surprisal
- Scaled Forward-Backward Algorithm: set_alphahat, set_betahat
- E-step: set_gamma, set_xi
- M-step: set_mu, set_var, set_tpm

Independent events: Diagonal covariance Gaussian

Suppose that $\vec{o} = [o_1, \ldots, o_D]$ is a D-dimensional observation vector, and the observation dimensions are uncorrelated (e.g., MFCC). Then we can write the Gaussian pdf as

$$b_j(\vec{o}) = \frac{1}{\sqrt{|2\pi\Sigma_j|}} e^{-\frac{1}{2}(\vec{o}-\vec{\mu}_j)^T \Sigma_j^{-1}(\vec{o}-\vec{\mu}_j)} = \prod_{d=1}^{D} \frac{1}{\sqrt{2\pi\sigma_{jd}^2}} e^{-\frac{1}{2}\frac{(o_d-\mu_{jd})^2}{\sigma_{jd}^2}}$$

Complexity of inverting a DxD matrix: $O\{D^3\}$

One scalar operation for each of the D dimensions: Complexity = $O\{D\}$

Claude Shannon, "A Mathematical Theory of Communication," 1948

1. An event is informative if it is unexpected. The information content of an event, e, must be **<u>some (as yet unknown) monotonically decreasing function, f(), of its probability</u>:**

$$i(e) = f(p(e))$$

2. The information provided by two independent events, e_1 and e_2, is the **<u>sum of the information provided by each</u>**:

$$i(e_1, e_2) = i(e_1) + i(e_2)$$

There is only one function, f(), that satisfies both of these criteria:

$$i(e) = -\log p(e)$$
$$i(e_1, e_2) = -\log p(e_1)p(e_2) = -\log p(e_1) - \log p(e_2) = i(e_1) + i(e_2)$$

Surprisal

The “information” provided by observation $\vec{o}$ is $i(\vec{o}) = -\log p(\vec{o})$.

But the word “information” has been used for so many purposes that we hesitate to stick with it. There is a more technical-sounding word that is used only for this purpose: “surprisal.”

$i(\vec{o}) = -\log p(\vec{o})$ is the “surprisal” of observation $\vec{o}$, because it measures the degree to which we are surprised to observe $\vec{o}$.

- If $\vec{o}$ is very likely ($p(\vec{o}) \approx 1$) then we are not surprised ($i(\vec{o}) \approx 0$).
- If $\vec{o}$ is very unlikely ($p(\vec{o}) \approx 0$), then we are very surprised ($i(\vec{o}) \approx \infty$).

Gaussian is computationally efficient, but numerically AWFUL!!

10d observation vector

Gaussian probability

Surprisal

```
~ — python — 99×20
>>>
>>> import numpy as np
>>> for z in range(11):
...     o = z*np.ones(10)
...     p = np.exp(-0.5*np.sum(np.square(o)))/np.sqrt(np.power(2*np.pi,10))
...     i = -np.log(p)
...     print('o = %s, p = %s, i = %s'%(o,p,i))
...
o = [0. 0. 0. 0. 0. 0. 0. 0. 0. 0.], p = 0.00010211761384541831, i = 9.189385332046728
o = [1. 1. 1. 1. 1. 1. 1. 1. 1. 1.], p = 6.880630697635048e-07, i = 14.189385332046728
o = [2. 2. 2. 2. 2. 2. 2. 2. 2. 2.], p = 2.104800896922658e-13, i = 29.189385332046726
o = [3. 3. 3. 3. 3. 3. 3. 3. 3. 3.], p = 2.9231356703387792e-24, i = 54.189385332046726
o = [4. 4. 4. 4. 4. 4. 4. 4. 4. 4.], p = 1.8430711707236542e-39, i = 89.18938533204673
o = [5. 5. 5. 5. 5. 5. 5. 5. 5. 5.], p = 5.275825471471384e-59, i = 134.18938533204673
o = [6. 6. 6. 6. 6. 6. 6. 6. 6. 6.], p = 6.856364784305663e-83, i = 189.18938533204673
o = [7. 7. 7. 7. 7. 7. 7. 7. 7. 7.], p = 4.045317301362102e-111, i = 254.18938533204673
o = [8. 8. 8. 8. 8. 8. 8. 8. 8. 8.], p = 1.0835936445670234e-143, i = 329.18938533204675
o = [9. 9. 9. 9. 9. 9. 9. 9. 9. 9.], p = 1.3177574718327724e-180, i = 414.18938533204675
o = [10. 10. 10. 10. 10. 10. 10. 10. 10. 10.], p = 7.275447423157845e-222, i = 509.18938533204675
>>>
```

Observations: reasonable numbers, easy to work with in floating point

Probability densities: **Unreasonable numbers**, very hard to work with in floating point!

Surprisal: reasonable numbers, easy to work with in floating point

WARNING: Don’t calculate surprisal using the method on this slide!!!
Use the method on the next slide!!!

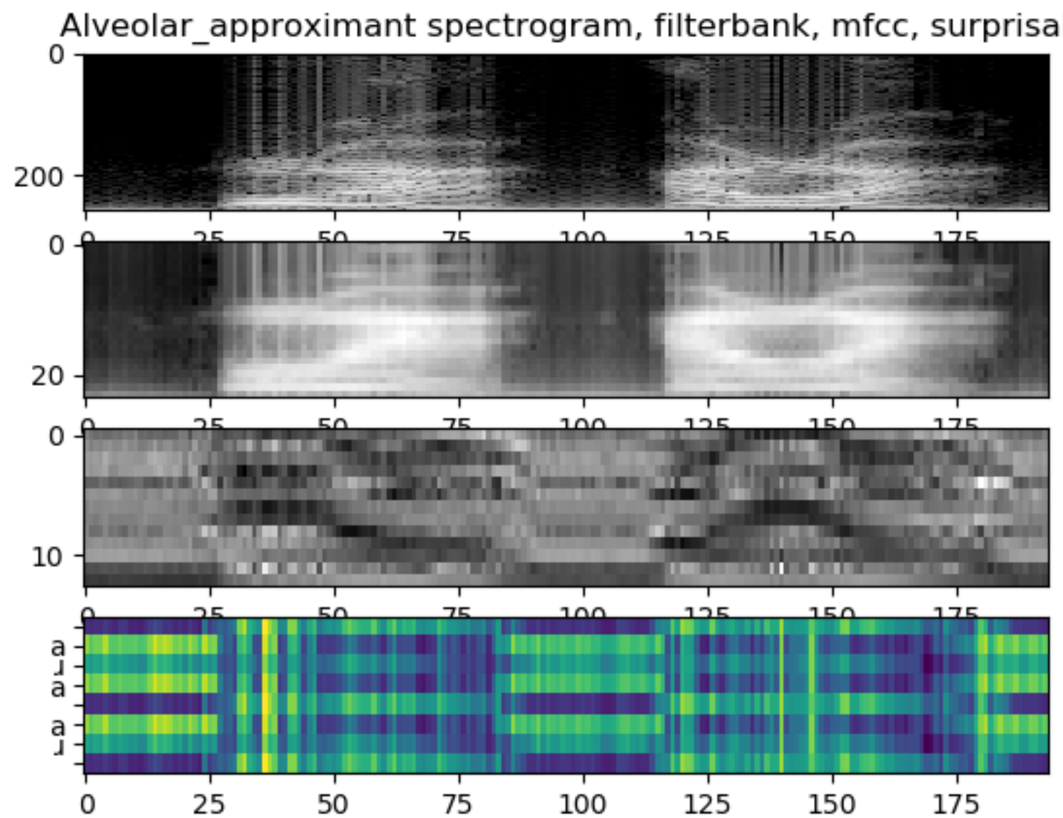
How to calculate surprisal without calculating probability first

$$i_j(\vec{o}) = -\ln b_j(\vec{o}) = -\ln \prod_{d=1}^{D} \frac{1}{\sqrt{2\pi\sigma_{jd}^2}} e^{-\frac{1}{2}\frac{\left(o_d-\mu_{jd}\right)^2}{\sigma_{jd}^2}}$$

$$= \frac{1}{2} \sum_{d=1}^{D} \left(\frac{\left(o_d - \mu_{jd}\right)^2}{\sigma_{jd}^2} + \ln 2\pi\sigma_{jd}^2 \right)$$

MP5 walkthrough: what surprisal looks like (after 1 epoch of training)

- Dark blue: small surprise
  - Silence model during silences: zero surprise
  - Vowel model during vowels: zero surprise
- Bright green: large surprise
  - Vowel model during silences: high surprise
  - Silence model during vowels: high surprise



Outline

- Background things that are done for you
  - Observations: mel-frequency cepstral coefficients = f(MSTFT)
  - Token to type alignment
- Gaussian surprisal, a.k.a. information: set_surprisal
- Scaled Forward-Backward Algorithm: set_alphahat, set_betahat
- E-step: set_gamma, set_xi
- M-step: set_mu, set_var, set_tpm

Forward-Backward Algorithm

$$\alpha_t(j) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} b_j(\vec{o}_t) = \sum_{i=1}^{N} \alpha_{t-1}(i) a_{ij} e^{-i_j(\vec{o}_t)}$$

Oh NO! The very small number came back again!

Solution: Scaled Forward-Backward

- The key idea: define a scaled alpha probability, alphahat ($\hat{\alpha}_t(j)$), such that

$$\sum_{j=1}^{N} \hat{\alpha}_t(j) = 1$$

- We can compute alphahat simply as

$$\hat{\alpha}_t(j) = \frac{\sum_{i=1}^{N} \hat{\alpha}_{t-1}(i) a_{ij} e^{-i_j(\vec{o}_t)}}{\sum_{j=1}^{N} \sum_{i=1}^{N} \hat{\alpha}_{t-1}(i) a_{ij} e^{-i_j(\vec{o}_t)}}$$

Solution: Scaled Forward-Backward

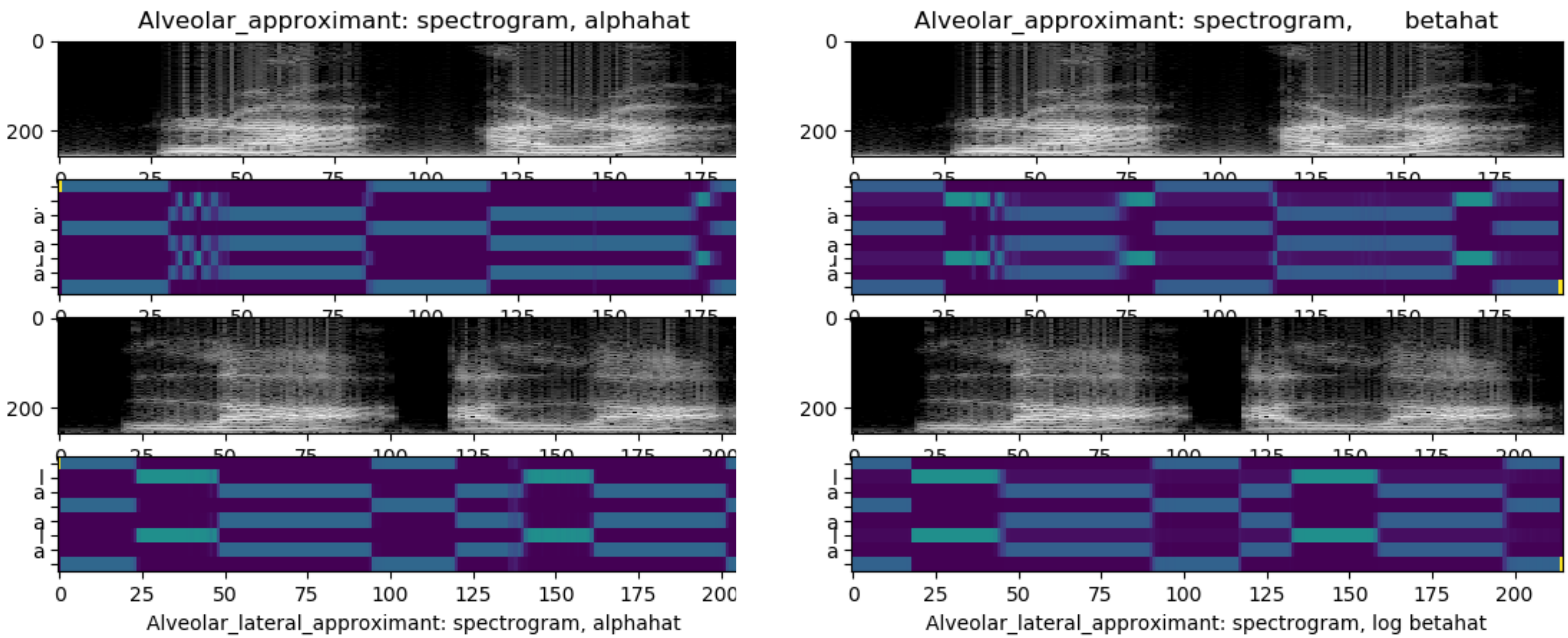
- Similarlym define a scaled betahat ($\hat{\beta}_t(i)$), such that

$$\sum_{i=1}^{N} \hat{\beta}_t(i) = 1$$

- We can compute betahat simply as

$$\hat{\beta}_t(i) = \frac{\sum_{j=1}^{N} a_{ij} e^{-i_j(\vec{o}_{t+1})} \hat{\beta}_{t+1}(j)}{\sum_{i=1}^{N} \sum_{j=1}^{N} a_{ij} e^{-i_j(\vec{o}_{t+1})} \hat{\beta}_{t+1}(j)}$$

MP5 Walkthrough: What alphahat and betahat look like



Why does scaling work?

Notice that the denominator is independent of i or j. So the difference between $\alpha_t(j)$ and $\hat{\alpha}_t(j)$ is a scaling factor (let's call it g_t) that doesn't depend on j:

$$\hat{\alpha}_t(j) = \frac{1}{g_t} \sum_{i=1}^{N} \hat{\alpha}_{t-1}(i) a_{ij} e^{-i_j(\vec{o}_t)} = \cdots = \frac{\alpha_t(j)}{\prod_{\tau=1}^{t} g_\tau}$$

Likewise, the difference between $\beta_t(i)$ and $\hat{\beta}_t(i)$ is some other scaling factor (let's call it h_t) that doesn't depend on i:

$$\hat{\beta}_t(i) = \frac{1}{h_t} \sum_{j=1}^{N} a_{ij} e^{-i_j(\vec{o}_{t+1})} \hat{\beta}_{t+1}(j) = \cdots = \frac{\beta_t(i)}{\prod_{\tau=t+1}^{T} h_\tau}$$

Why does scaling work?

So we can calculate gamma as:

$$\gamma_t(j) = \frac{\alpha_t(j)\beta_t(j)}{\sum_{k=1}^{N} \alpha_t(k)\beta_t(k)} = \frac{\alpha_t(j)\beta_t(j) / \prod_{\tau=1}^{t} g_\tau \prod_{\tau=t+1}^{T} h_\tau}{\sum_{k=1}^{N} \alpha_t(k)\beta_t(k) \,/\, \prod_{\tau=1}^{t} g_\tau \prod_{\tau=t+1}^{T} h_\tau}$$

$$= \frac{\hat{\alpha}_t(j)\hat{\beta}_t(j)}{\sum_{k=1}^{N} \hat{\alpha}_t(k)\hat{\beta}_t(k)}$$

In other words, the scaling (of the scaled forward-backward algorithm) has no effect at all on the calculation of gamma and xi!!

Outline

- Background things that are done for you
  - Observations: mel-frequency cepstral coefficients = f(MSTFT)
  - Token to type alignment
- Gaussian surprisal, a.k.a. information: set_surprisal
- Scaled Forward-Backward Algorithm: set_alphahat, set_betahat
- E-step: set_gamma, set_xi
- M-step: set_mu, set_var, set_tpm

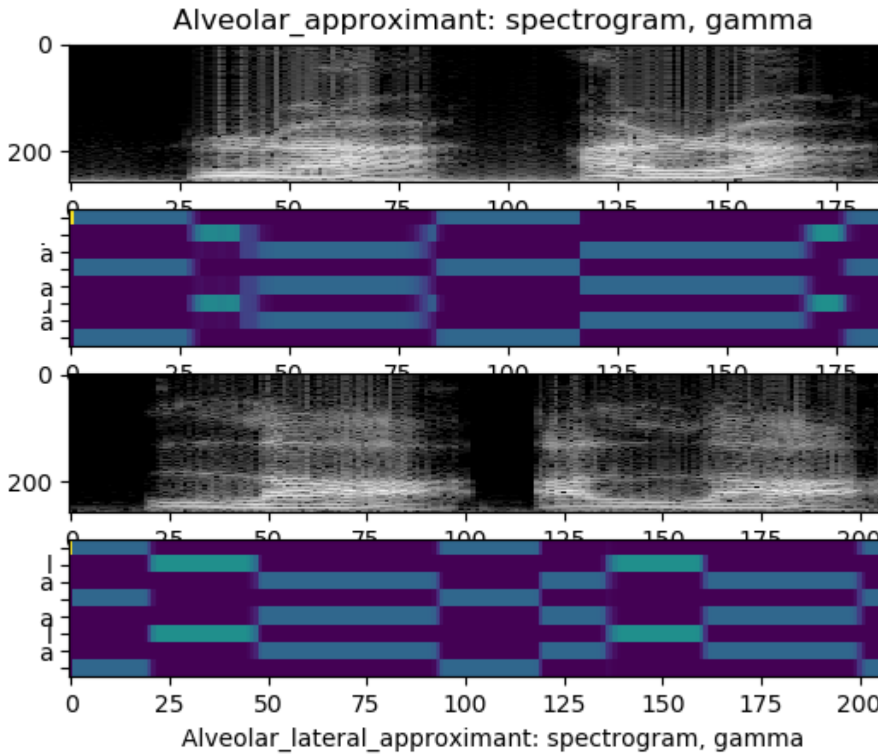
E-Step: set_gamma, set_xi

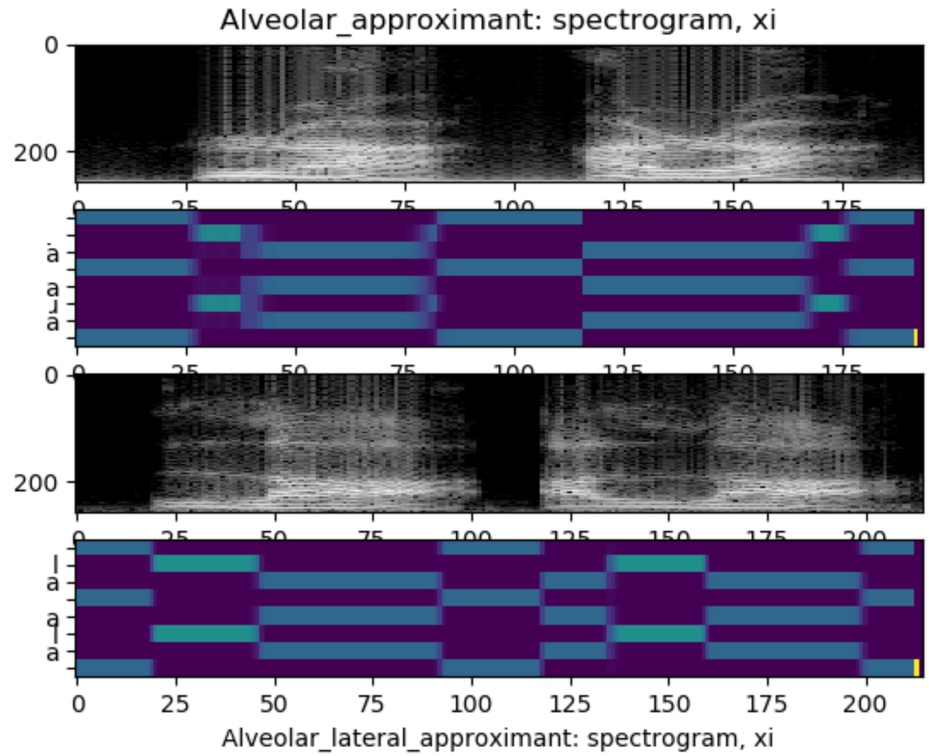
In other words, the scaling (of the scaled forward-backward algorithm) has no effect at all on the calculation of gamma and xi!!

$$\gamma_t(j) = \frac{\hat{\alpha}_t(j)\hat{\beta}_t(j)}{\sum_{k=1}^{N} \hat{\alpha}_t(k)\hat{\beta}_t(k)}$$

$$\xi_t(i,j) = \frac{\hat{\alpha}_t(i)a_{ij}e^{-i_j(\vec{o}_{t+1})}\hat{\beta}_{t+1}(j)}{\sum_{k=1}^{N}\sum_{l=1}^{N} \hat{\alpha}_t(k)a_{kl}e^{-i_l(\vec{o}_{t+1})}\hat{\beta}_{t+1}(l)}$$

MP5 Walkthrough: What gamma and xi look like





Outline

- Background things that are done for you
  - Observations: mel-frequency cepstral coefficients = f(MSTFT)
  - Token to type alignment
- Gaussian surprisal, a.k.a. information: set_surprisal
- Scaled Forward-Backward Algorithm: set_alphahat, set_betahat
- E-step: set_gamma, set_xi
- M-step: set_mu, set_var, set_tpm

M-Step: set_mu, set_var, set_tpm

Define the following index variables:

- u =Utterance ID
- t =Frame number
- i, j =Token indices
- m, n =Type indices

And, for convenience,

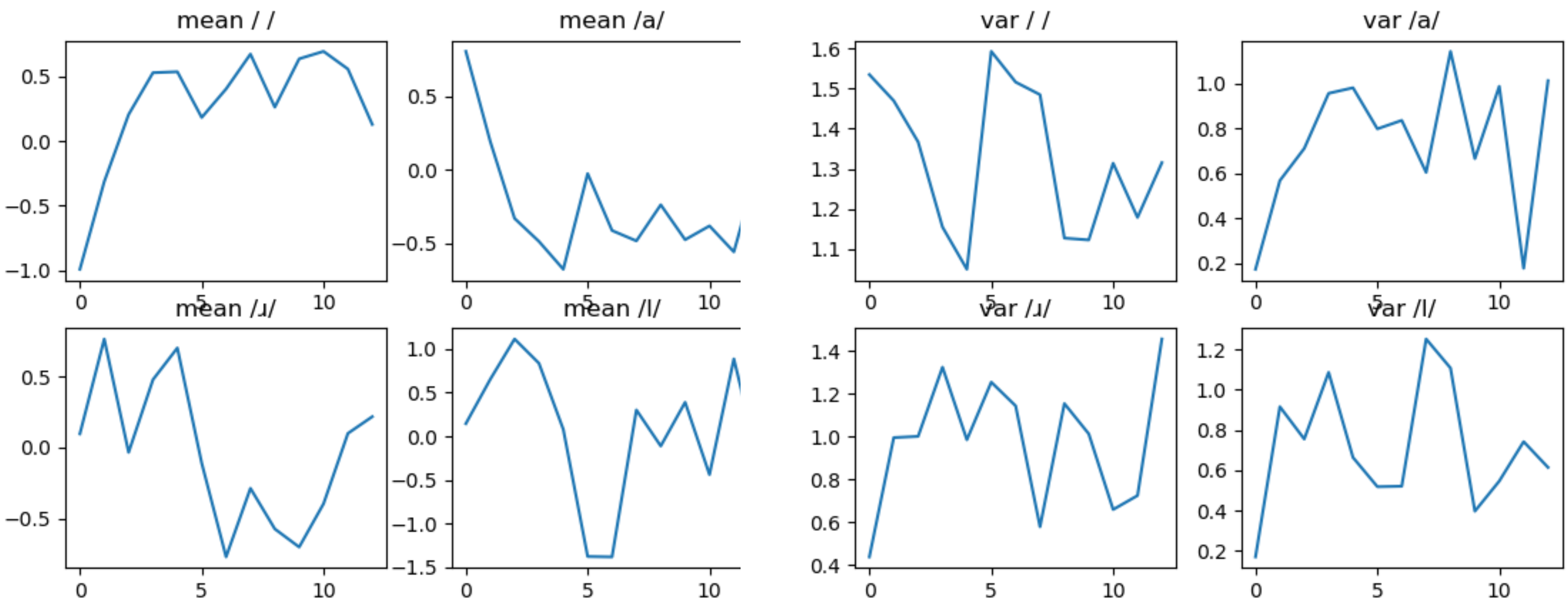
- $\vec{\sigma}_m^2$ =Variance vector for the m'th type
- $TPM(m,n)$ =Transition probability from type m to type n

$$\vec{\mu}_m = \frac{\sum_{u=1}^{U} \sum_{t=1}^{T} \sum_{j:type(j)=m} \gamma_t(j) \vec{o}_t}{\sum_{u=1}^{U} \sum_{t=1}^{T} \sum_{j:type(j)=m} \gamma_t(j)}$$

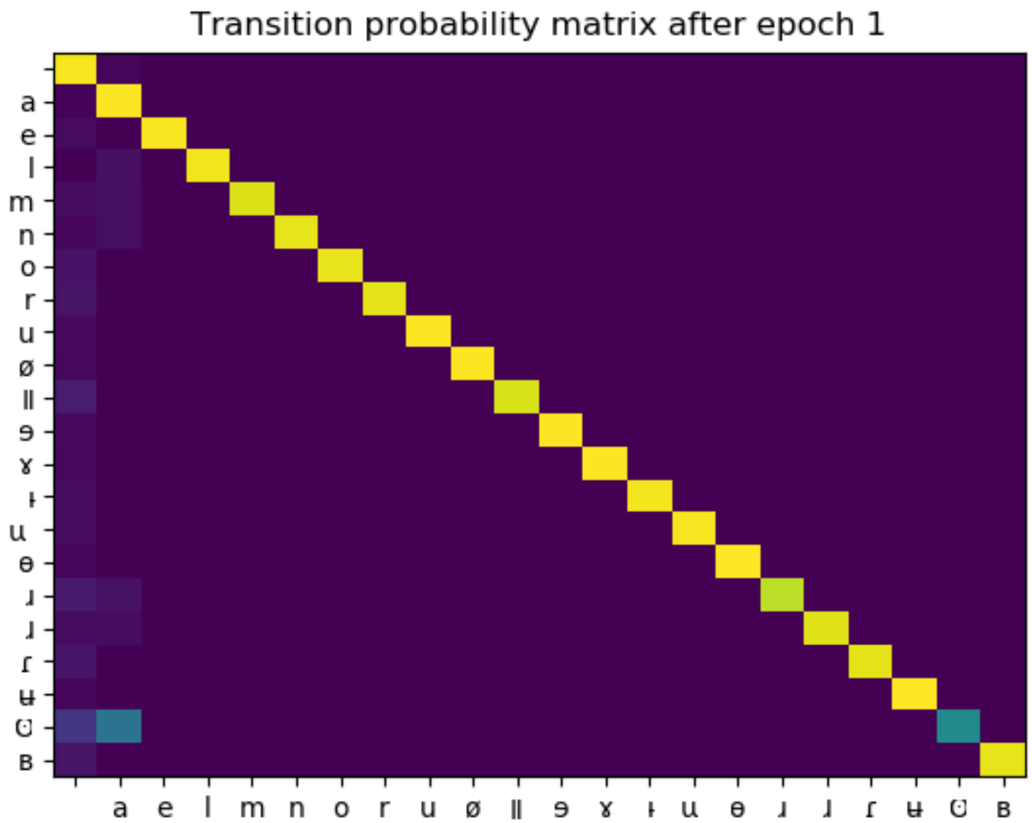
$$\vec{\sigma}_m^2 = \frac{\sum_{u=1}^{U} \sum_{t=1}^{T} \sum_{j:type(j)=m} \gamma_t(j) (\vec{o}_t - \vec{\mu}_m)^2}{\sum_{u=1}^{U} \sum_{t=1}^{T} \sum_{j:type(j)=m} \gamma_t(j)}$$

$$TPM(m,n) = \frac{\sum_{u=1}^{U} \sum_{t=1}^{T} \sum_{i,j:type(i,j)=(m,n)} \xi_t(i,j)}{\sum_{u=1}^{U} \sum_{t=1}^{T} \sum_{i,j:type(i)=(m)} \xi_t(i,j)}$$

MP5 Walkthrough: What mu and var look like



MP5 Walkthrough: What TPM looks like



Conclusions

- Step 0, set_surprisal: use the formula on slide 22 to compute $i_j(\vec{o})$ directly, without computing $b_j(\vec{o})$
- Steps 1 and 2, set_alphahat and set_betahat: use the formulas on slides 26 and 27, this allows you to immediately normalize alphahat and betahat so that they each sum to 1.
- Steps 3 and 4, set_gamma and set_xi: use the formulas on slide 32, you get $\gamma_t(j)$ and $\xi_t(i,j)$ directly from $\hat{\alpha}_t(i)$ and $\hat{\beta}_{t+1}(j)$, despite the scaling!
- Steps 5-7, set_mu, set_var, and set_tpm: use the formulas on slide 35, the only trick is that you have to be careful about token-to-type mapping.

… and the final speech recognition result: How well did it work?   About 90% accurate! (testing on the training data, though!)