# Speech Processing

Undergraduate course code: LASC10061
Postgraduate course code: LASC11065

Slide pack 3 of 3: Automatic Speech Recognition

# Speech recognition

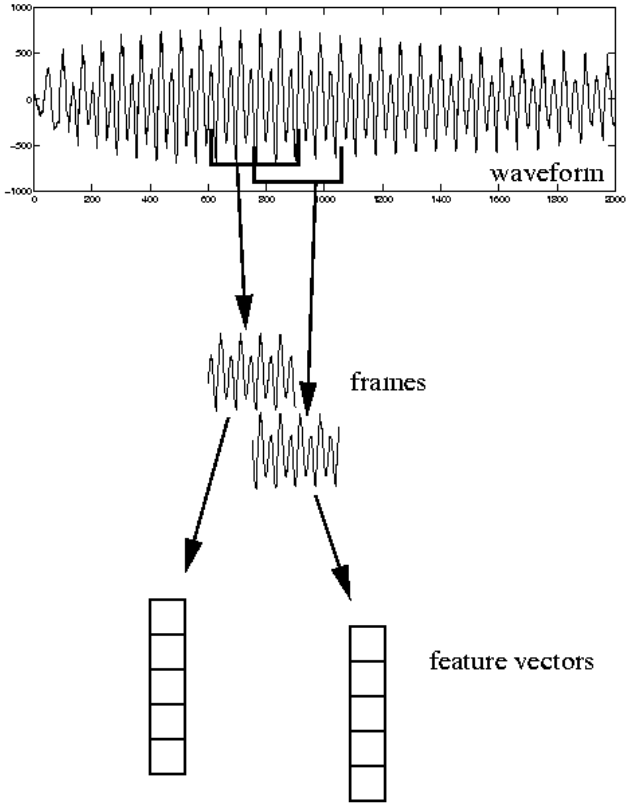
- Feature extraction
  - Fourier analysis of waveforms
  - From how the ear works, to perceptually-motivated processing
  - De-correlation: the Cepstral transform and Mel Frequency Cepstral Coefficients (MFCCs)
- Comparing frames of speech
  - Distance measures and probability
- Whole word templates
- Dynamic time warping (DTW)
- Hidden Markov models (HMMs)
- Sub-word models
- Putting it all together
- Training HMMs
- Language models

# Speech recognition - lecture 1 of 5

- frame-based analysis
- Fourier transform
- feature vectors
- Euclidean distance measure
- Dynamic Time Warping

# Frame based analysis of speech

- We assume that the speech signal is statistically stationary over the duration of the window (i.e., its spectral properties are constant)
- For each frame of speech, we extract a vector of parameters – typically 39 numbers that capture all the segmental information for that frame

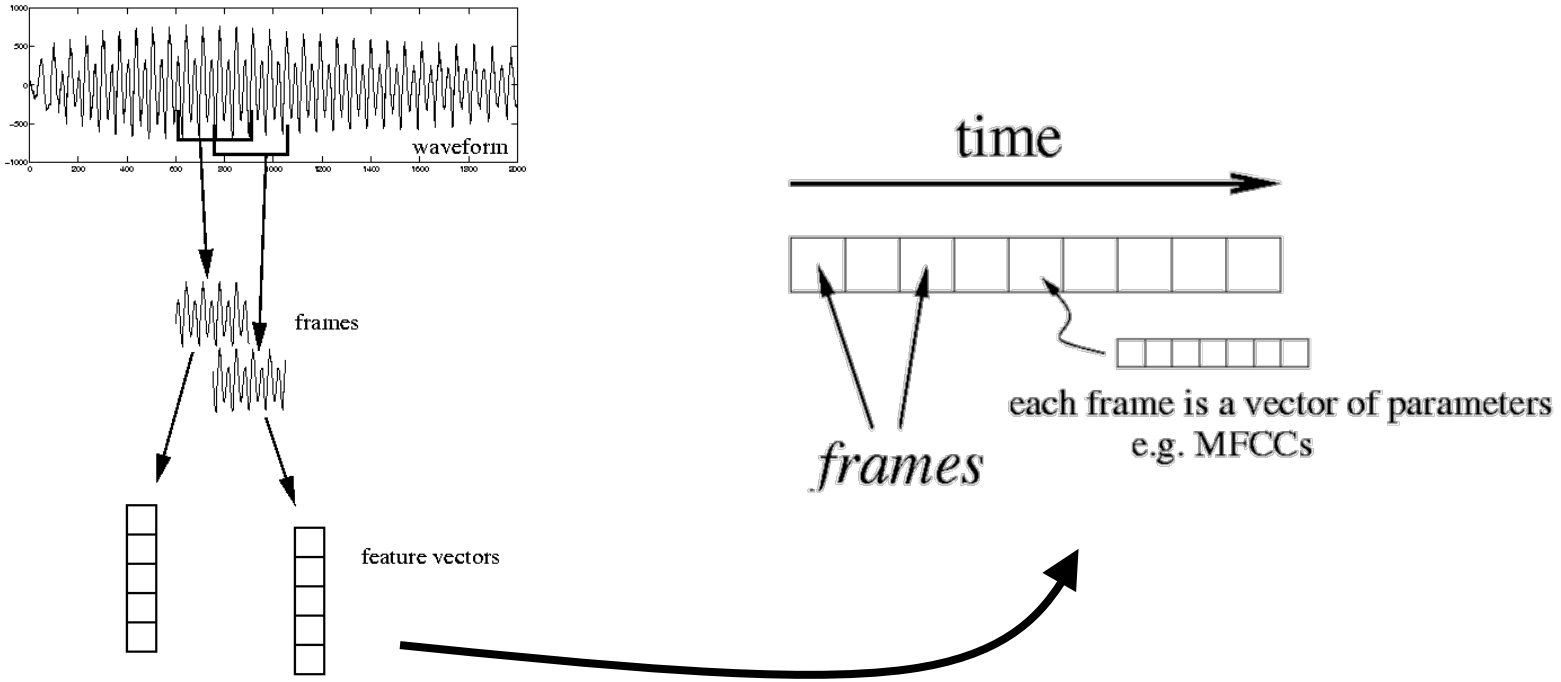


# Whole word templates

- We are going to work our way up to HMMs slowly. Let's start with a very simple problem: isolated word recognition using template matching.
- The scenario is as follows:
    - We have recorded a set of *reference* words, known as templates
    - Given a recording of an *unknown* word:
        - Match it against each *reference* word in turn
        - Find the closest match
        - Announce the result
- The key process is **finding the closest match**. The operation we must perform is: measure how similar two recorded words are (*reference* and *unknown*). The *reference* is sometimes called the *template.*

# Parameterising the speech signal

- From now on, we will only be dealing with parameterised speech signals
- The waveform is represented as:
  - A sequence of frames: each frame is a vector of parameters

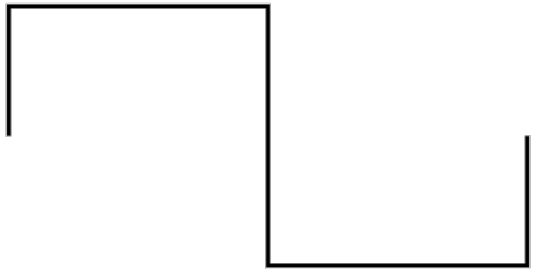


# Working in the frequency domain

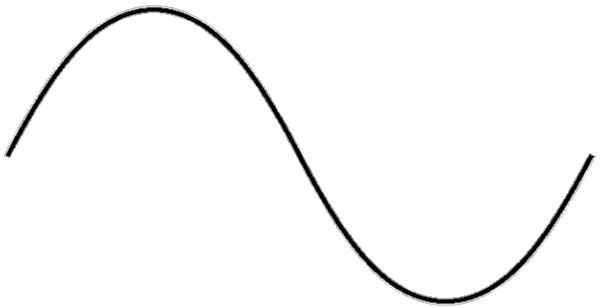
- For speech recognition, we want a description of the speech signal which:
  - Captures all the important information
  - Removes unwanted variability
    - Fundamental frequency not required (for many languages)
    - Speaker identity not required
  - Is fast to compute
- First, let's look at how we can build up any waveform by adding simple waveforms: Fourier analysis. This is how we transform a waveform to a spectrum

# Making a square wave

- Can we make this

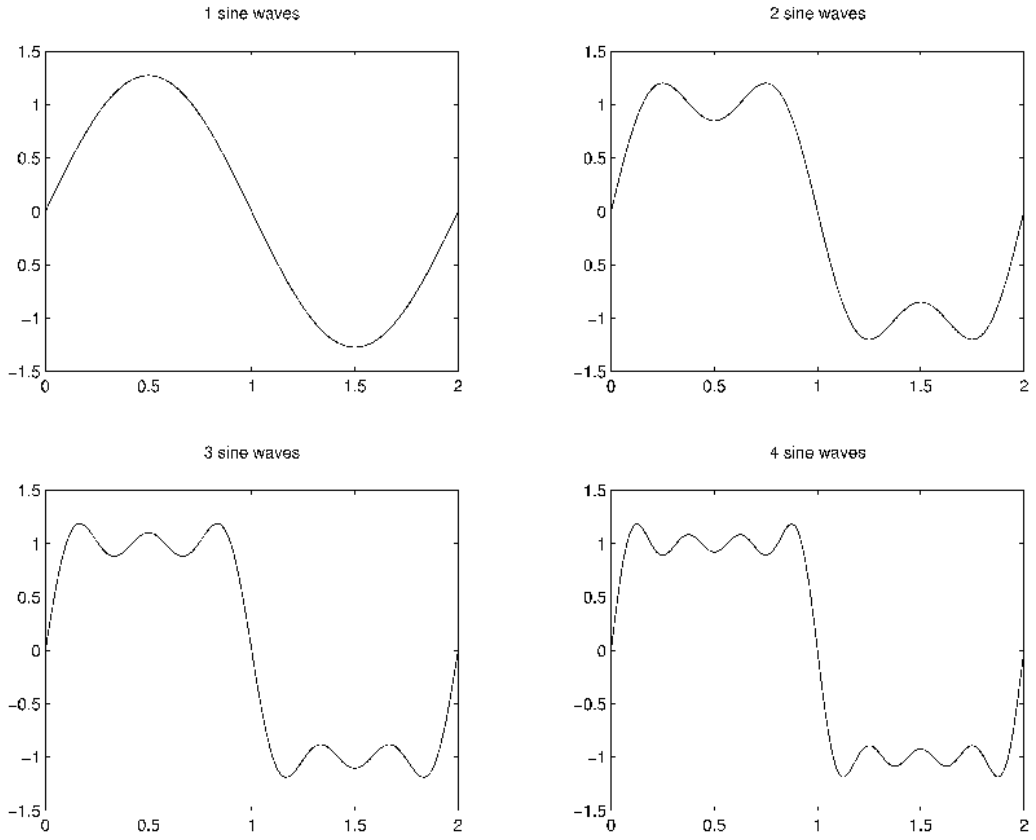


- by adding together waveforms like this?



# Fourier analysis of a square wave

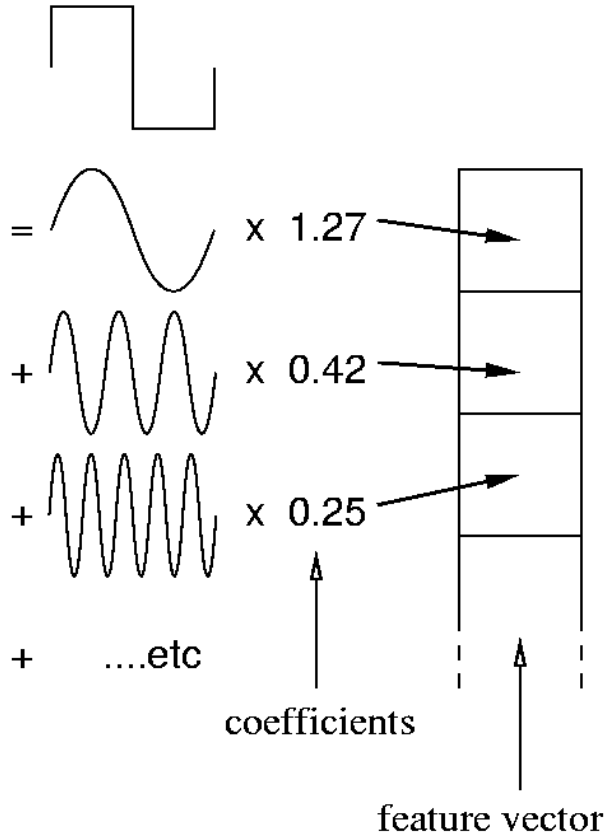
- Yes we can! At least we can approximate it. The approximation gets closer and closer the more waveforms we add in.




http://www.falstad.com/fourier/

# Decomposing the square wave

- We could make a feature vector of the weights used for each sine wave component

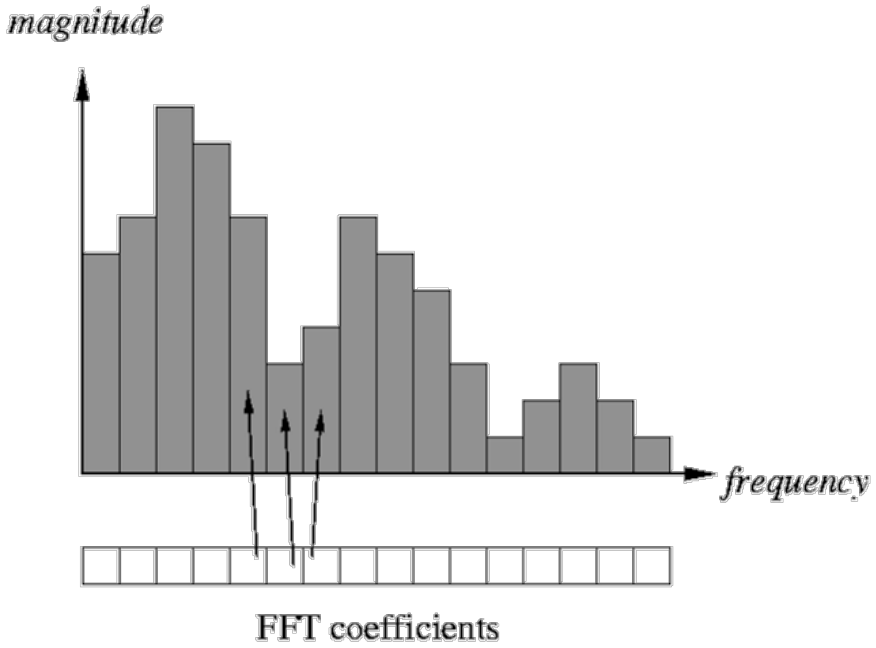


# Fourier transform

- The coefficients (weighting factors) of the components are a representation of the frequency contents of the waveform
- The Fourier transform computes these coefficients from the waveform
- We can vary the number of coefficients by varying the duration of the waveform being analysed
    - This varies the resolution of the frequency domain representation
- There is an inverse transform back to the time domain
- Terminology:
    - DFT: Discrete Fourier Transform
    - FFT: Fast Fourier Transform

# FFT spectrum of a signal

- The FFT coefficients show the proportions of different frequencies present in the signal. We can plot them:



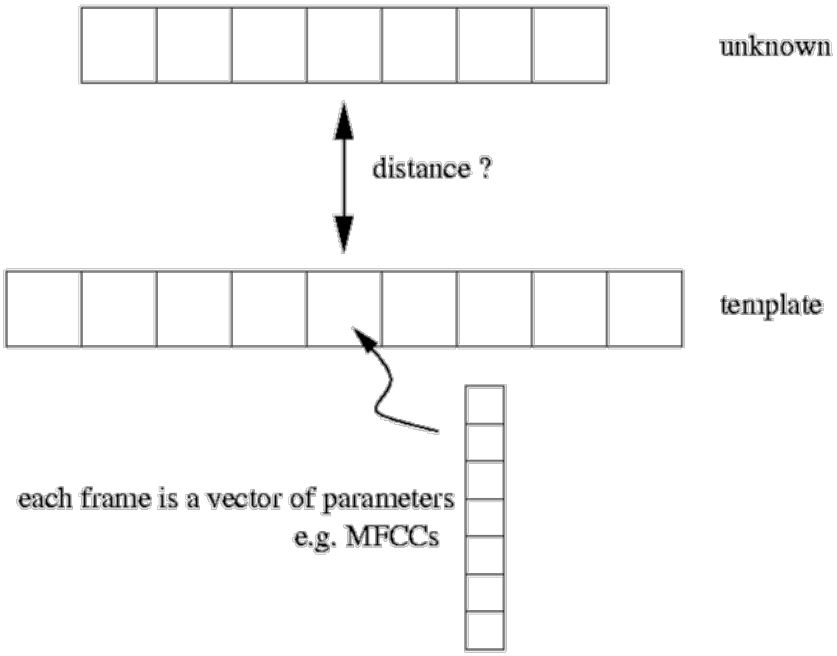

- This is the spectrum of the signal. For now, let's assume this is a suitable representation for doing speech recognition (we'll come back to this oversimplification a little later and fix it by replacing the FFT coefficients with MFCCs)

# Trading off time and frequency resolution

- Using the FFT, we can vary the number of coefficients, to vary the frequency resolution. But there is a price to pay…
- The number of coefficients depends on the number of samples in the waveform being analysed
- Long window
  - Poor time resolution
  - Good frequency resolution
- Short window
  - Good time resolution
  - Poor frequency resolution
- Feature extraction for speech recognition must be a compromise
- Typically, frames of 25ms duration are used, spaced every 10ms

# Measuring similarity

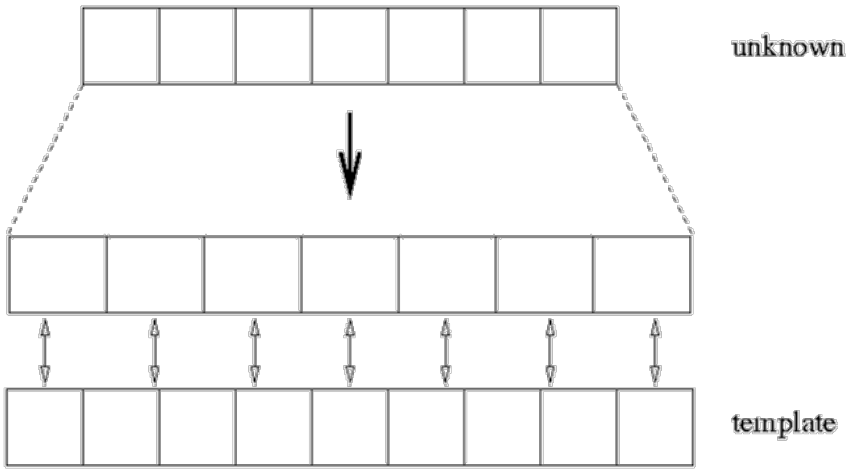
- We want a measure of the distance between two recorded words




- The distance between the two patterns is the sum of the local distances between corresponding frames

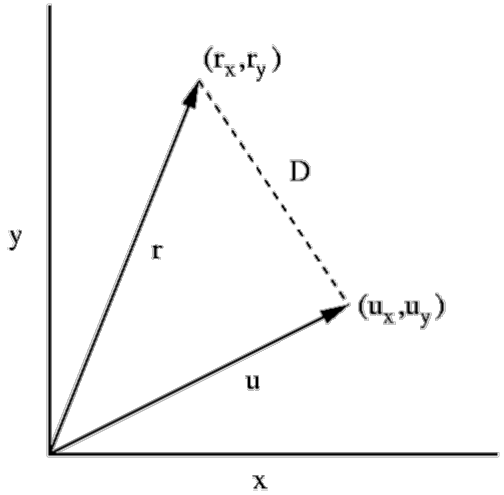
# Linear time warping

- There are two operations involved in finding the distance between two sequences of frames
  - Find an alignment between the two sequences of frames
  - Add up the local distances between aligned pairs of frames
- The simplest alignment is to stretch the shorter pattern



# Local distance measure

- Having aligned the two sequences, we need to compare individual frames
- Imaging that there are only two components in each vector – we can draw the vectors on a 2-dimensional (x-y) plot like this:




$$D = \sqrt{(r_x - u_x)^2 + (r_y - u_y)^2}$$

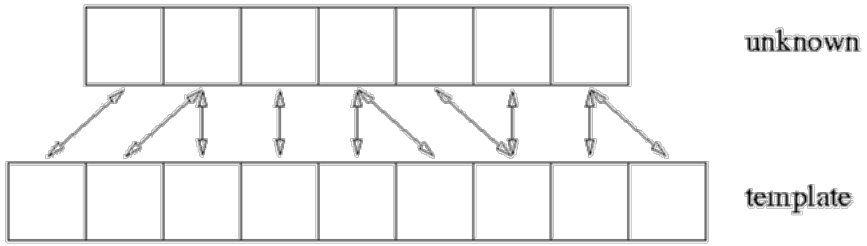
# What's wrong with our word recogniser?

- Time alignment of reference (template) and unknown word
  - linear stretch is not appropriate for speech
  - because some sounds are more 'stretchy' than others
- Local distance measure
  - currently just a geometrical distance
  - gives equal weight to all components in the vector

- We'll deal with the alignment problem first

# Dynamic time warping

- Some speech sounds ‘stretch’ more than others, when we extend the duration of a word. We need a dynamic alignment which can deal with this




- There are many possible alignments
  - first, how do we **define** the correct alignment?
  - then, how do we **find** it?

# Defining the correct alignment

- We define the correct alignment as:
    - the one that results in the smallest total distance between the unknown word and the reference
- This is appropriate for speech
    - it will lead to alignments of the most similar sounds in the reference and unknown words
- So, the problem now is finding that alignment amongst all the possible alignments.
    - a search problem!
- **Search** is a key concept in speech recognition – it will turn up again

# Some important terms

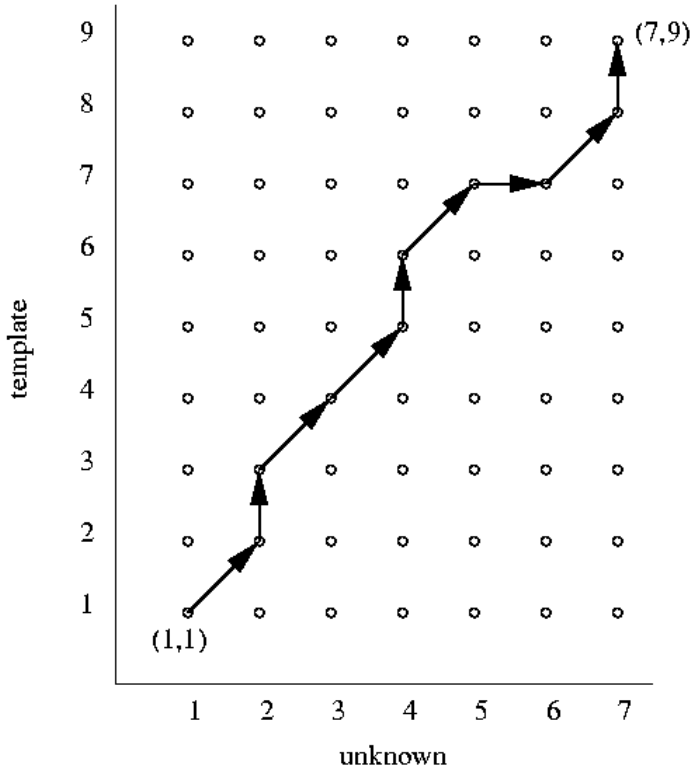
- When talking about matching patterns, we use the following terms
    - distance
    - cost
    - probability
    - likelihood
- Two similar patterns will have a small distance or cost.
- When we move on to HMMs later, these terms will be replaced by probabilities or likelihoods. High cost corresponds to low probability

# Searching for the alignment

- The simplest search method is just to try every possible alignment
- This algorithm is called **exhaustive search**
- Will this work (i.e., will it always find the correct alignment)?
    - Yes
- How many alignments are there?
    - Too many
- Homework: find all the alignments between a sequence of 7 frames and a sequence of 9 frames (only if you have nothing better to do…)

# A cunning plan

- So, there will generally be too many alignments for an exhaustive search. We'll have to be a bit more clever.
- Let's represent the problem visually:

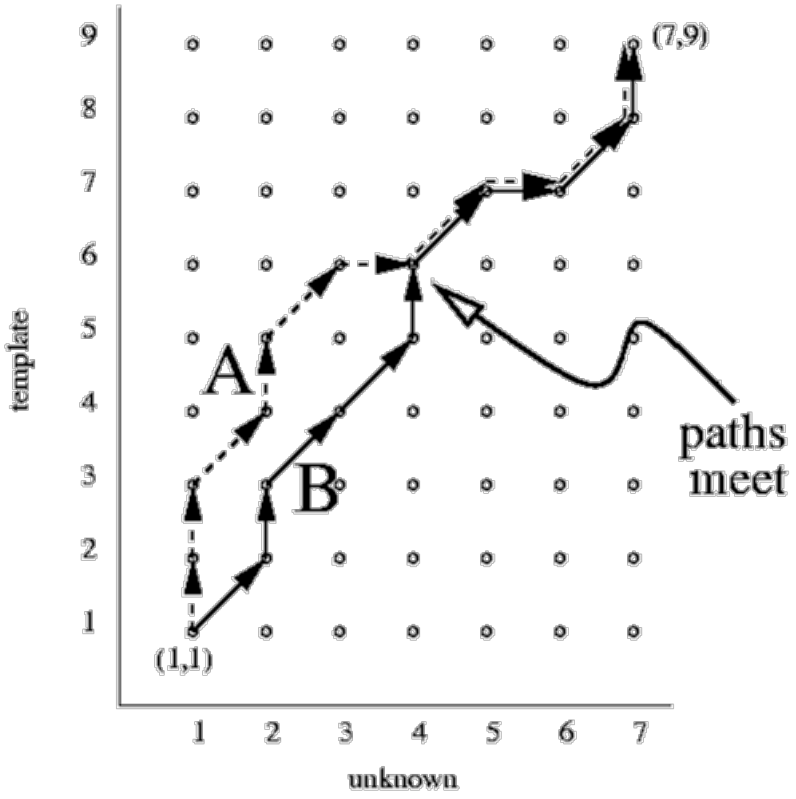


# Dynamic programming

- The path describes which frames of the unknown word are aligned with which frames of the reference
- The path cost is just the sum of the local distances along that particular path
- The problem is to
    - find the path through the grid with minimises the total cost of the path

# Dynamic programming

- Consider these two paths:




- **Only one** of the two sub-paths A or B is part of the cheapest (=best) overall path passing through point (4,6)

# The key to dynamic time warping

- Consider a point in the grid - point (4,6) in the previous diagram
  - the cheapest overall path which passes through this point **must** contain the cheapest path **into** this point
  - this means that when two (or more) possible paths arrive at a point in the grid, we only need to consider the cheapest one
  - we can forget about the other one(s)
- In terms of the search problem
  - we have pruned some of the possible paths, reducing the search space
- This particular type of pruning is **very special**
  - it will not introduce any errors: the **only** paths that get pruned are ones that would **never** have 'won' anyway.

# Dynamic time warping in action

- A simple DTW algorithm
- Pre-compute all the local distances between all possible pairs of frames in the reference and unknown words
- Store these distances at the points on the grid
- Start a path at (1,1)
- Repeat
    - Extend all possible paths forward one frame
    - Only keep best path into each grid point
- Until we reach the top right corner of the grid
- The best path arriving at that point is the winner - and its cost is the overall lowest cost alignment between reference and unknown

# Is DTW good enough for speech recognition?

- We can characterise the difficulty of a speech recognition task by the factors: speaker, vocabulary and conditions
- A typical application of DTW is:
    - Speaker-dependent (only works accurately for one particular person)
    - Small vocabulary
    - Isolated word recognition
- For example, voice-dialling on a mobile phone.
    - Typical error rates might be 1-2% for 50 place names, down to 0.5% for 10 digits

# Problems with DTW

- Number of local distances summed is path dependent, since paths vary in their length
    - Solution: weight diagonal steps differently to horizontal or vertical ones
- Can get unreasonable alignments; solution is to impose some constraints
    - local (slope)
    - global (stay near the main diagonal of the grid)
- Time variability handled by dynamic programming: not speech specific
- Spectral variability handled by distance measure: not speech specific
- Only uses a single reference copy of each word
- Needs accurate endpointing of both reference and unknown words

# Speech recognition - lecture 2 of 5

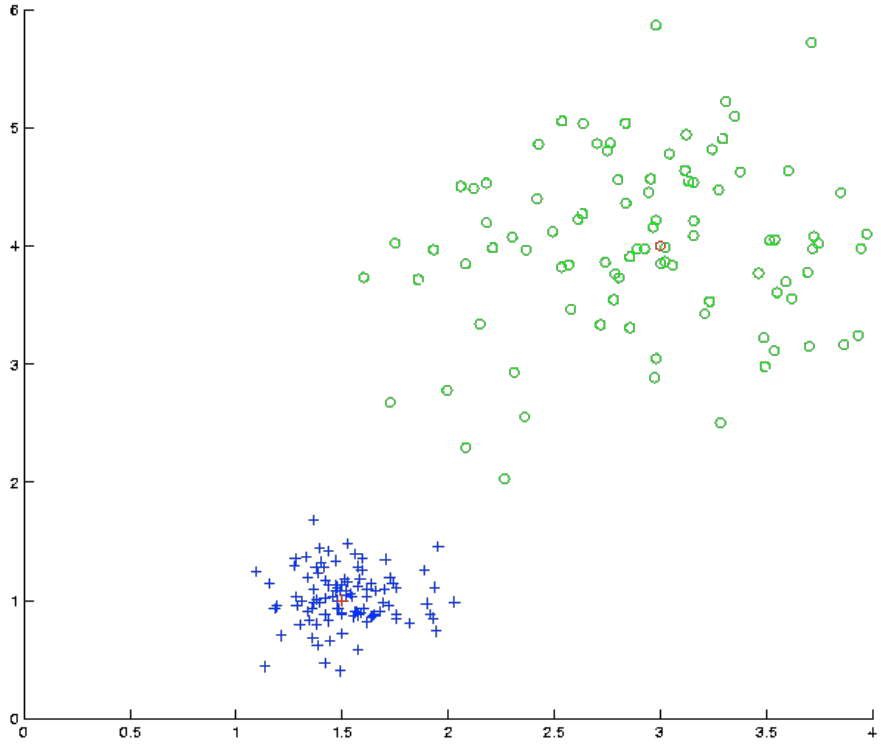
- probability distributions
- Gaussians
- MFCCs

# Distance measures and probability distributions

- Need to improve the Euclidean distance measure
  - It does not reflect the distribution (spread) of the data
- Need a distance measure derived from data
  - i.e., **learn from data**
- Probability distributions
  - Measure the distribution of some training data
  - Use this distribution estimate the probability of the test data

# What are (probability) distributions?

- Here are 2-dimensional feature vectors for a set of data. Each data point belongs to one class, either + or o



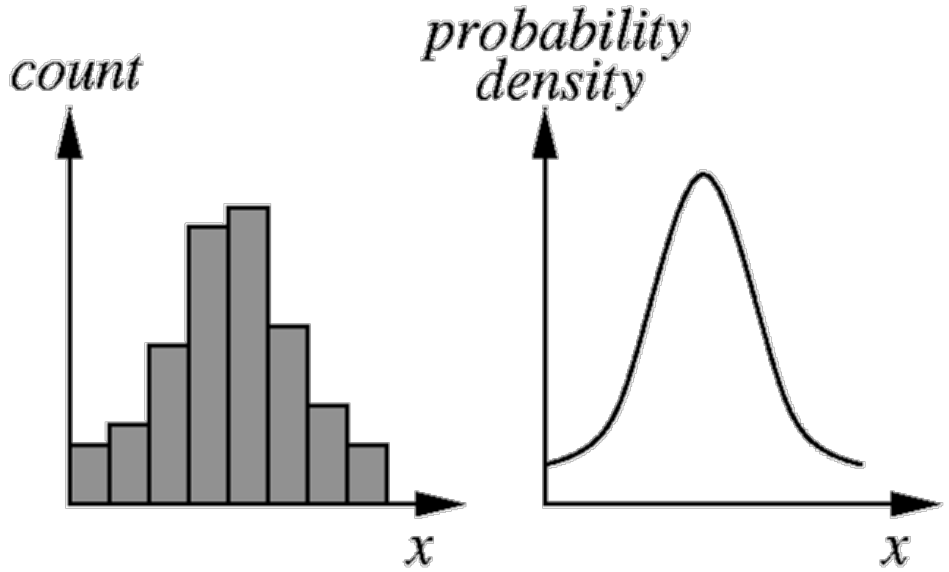

- the simple Euclidean distance measure will fail here

# Learning from data

- We want to use our training data to estimate a probability distribution, but how do we represent a probability distribution?
- As a set of examples
  - e.g., the actual points
- Discrete
  - e.g., a histogram or scatter plot
- Parametric
  - e.g., a Gaussian (same as the “normal distribution” or the “bell curve” you may have encountered if you’ve done a statistics course)

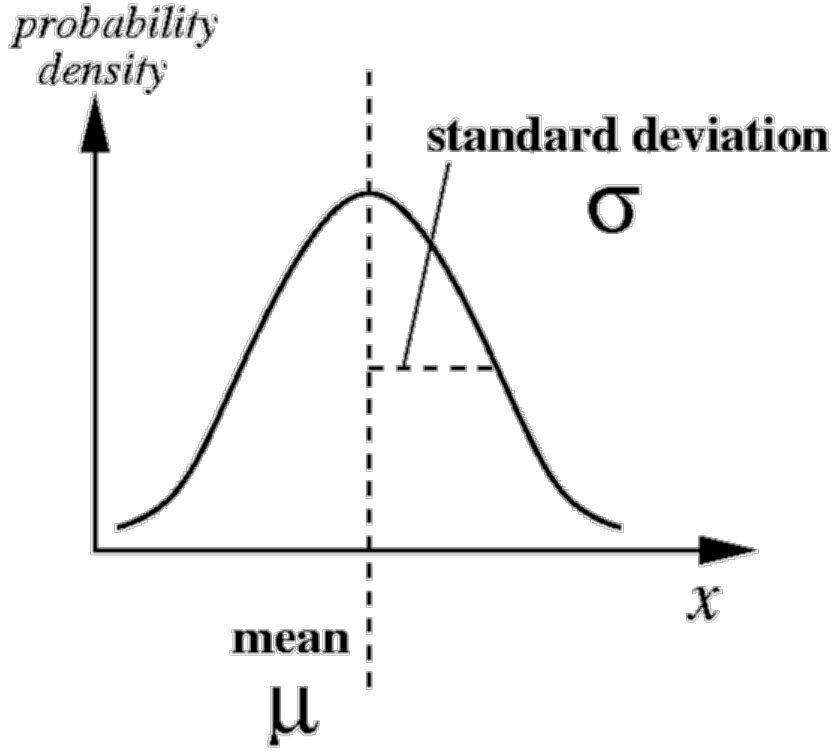
# Learning from data

- The Gaussian, or normal, distribution has just 2 parameters
  - mean and variance
- These represent the average value of the data, and the spread about that value
- If you know some statistics: variance = (standard deviation)$^2$



# The Gaussian distribution

- Univariate Gaussian:



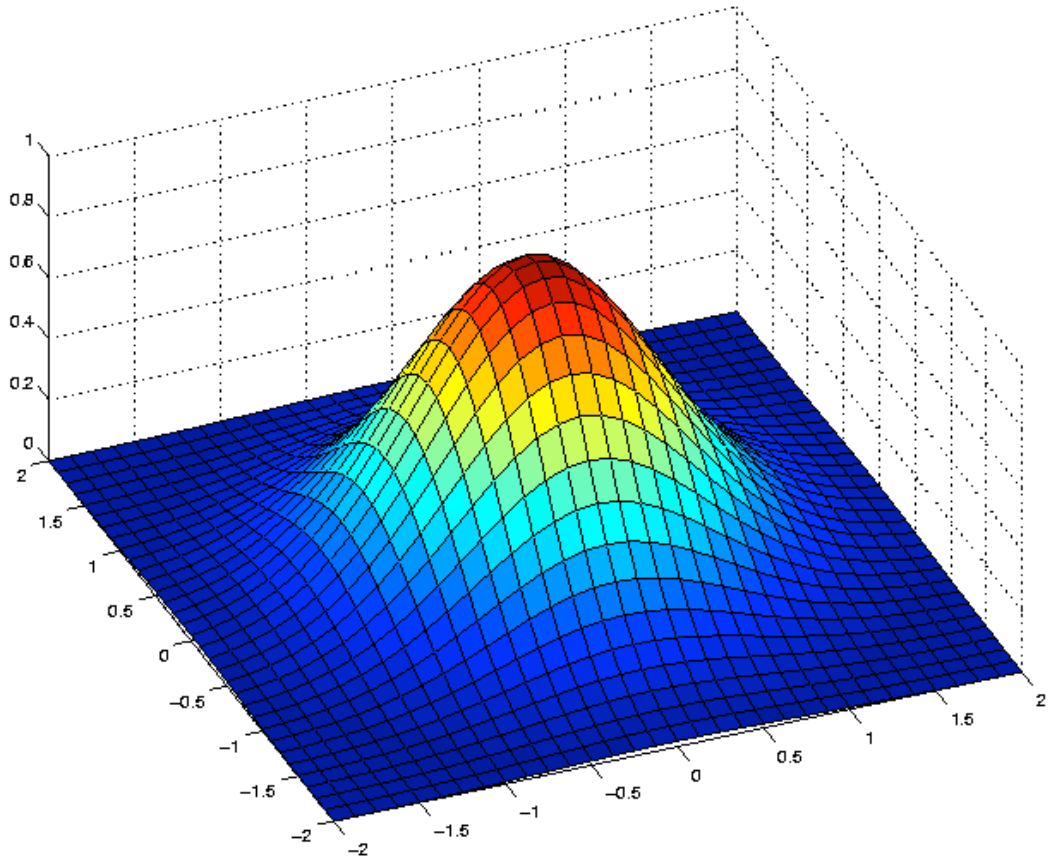

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

# What's so special about the Gaussian?

- Many convenient properties
    - e.g., multiply two random variables with Gaussian distributions and the result is a Gaussian distribution
- Central limit theorem
    - if we add up enough random variables, of whatever distribution, the distribution of the sum tends to be Gaussian
- Often a good approximation of observed data from the real world
- So from now on, when we talk about probability distributions of speech data, we will generally be thinking of Gaussian distributions.

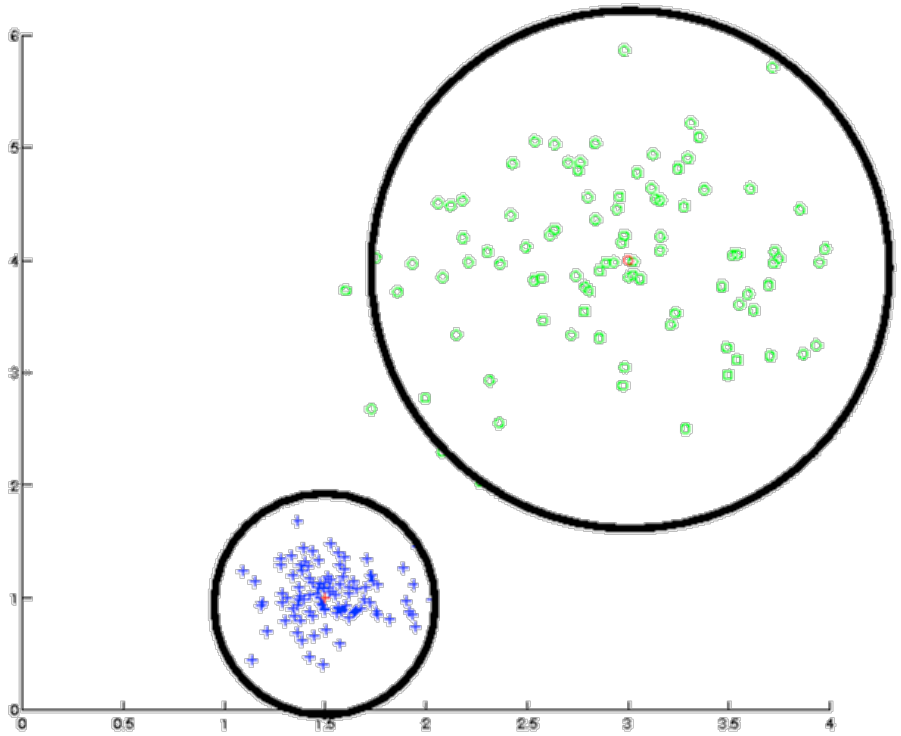
# How does using the Gaussian actually help

- Let's move up to two dimensions (the Gaussian can be extended to any number of dimensions) - it's now a **multivariate Gaussian**



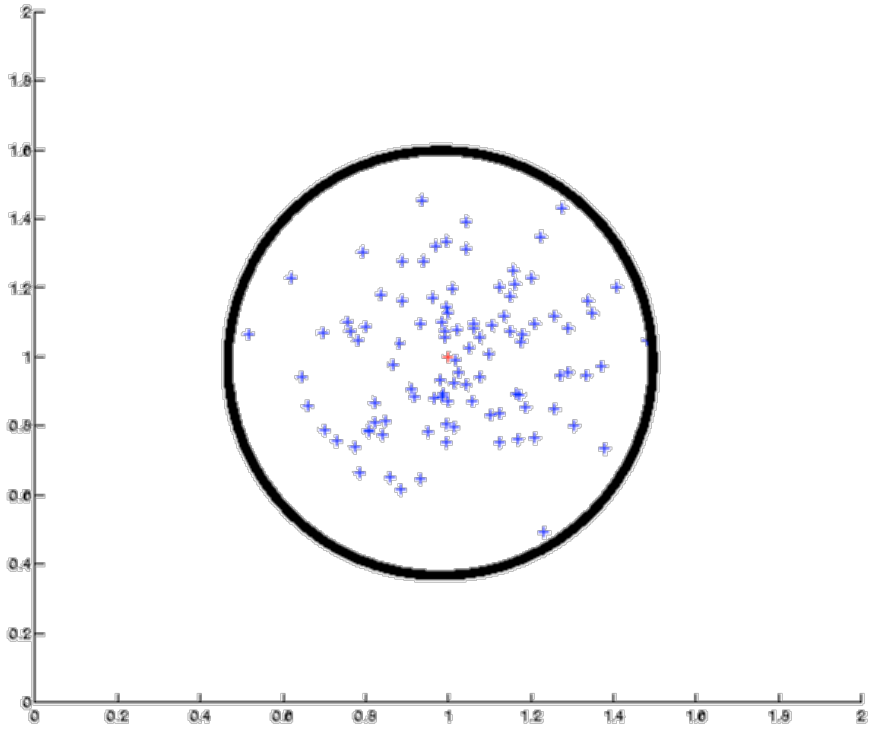
# Using Gaussians

- We can put Gaussians on our previous example. The large circles show two standard deviations away from the mean



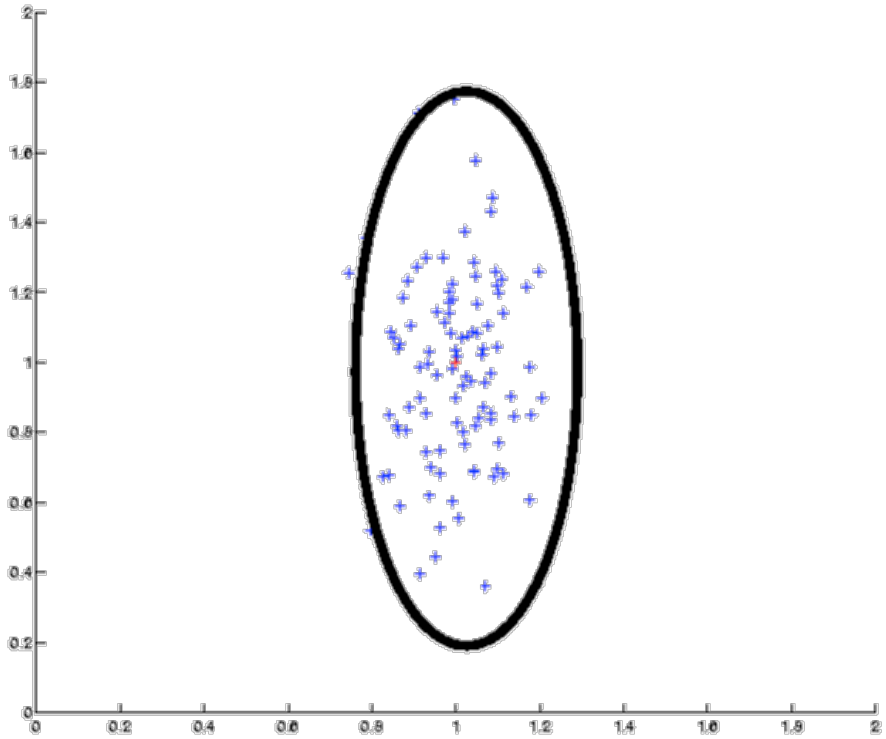
# Example 1: uniform distribution

- When the data is distributed uniformly in all dimensions, the Gaussian will be circular (spherical in 3 dimensions, …)



# Example 2: variance

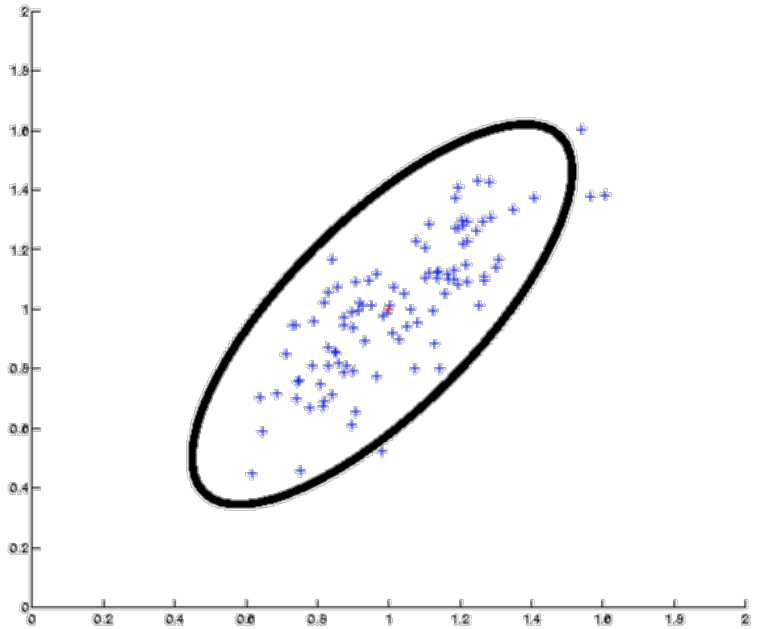
- In general, the variance will not be the same in all dimensions, so the Gaussian will not be spherical:




- Now, the variance of each dimension must be calculated and stored individually – the number of parameters increases a bit

# Example 3: covariance

- If the dimensions are correlated, then the Gaussian will be rotated:




- Now, even more parameters are needed: the covariances between every pair of dimensions – the number of parameters increases a lot

# Why is covariance a problem?

- If we make the assumption that components of the observation vector are independent (i.e., not correlated), then we need fewer parameters
    - but why is having fewer parameters such a good thing?
- To understand that, we need to know how to estimate the parameters of the Gaussians: we need to derive the values of the mean and variance of our Gaussians from some training data
    - Need multiple examples of each class
    - Need a formula for the parameters in terms of the data
    - More parameters to estimate means that more data are required
- Deriving the re-estimation formulae from first principles is beyond the scope of this course, but we need to understand what they mean.

# Estimating the parameters of the Gaussian

- There are many ways of deriving re-estimation formulae. The simplest and most common is:
    - Maximum likelihood, or simply ‘ML’
- What the method does is to

***adjust the model parameters to <u>maximise the likelihood</u> of the training data***

- For our Gaussian, this means picking the mean and variance that makes the training data most probable

# ML estimate of Gaussian parameters

- This result is simple enough:
  - The best estimate for the mean of the Gaussian is…
    - ...the mean of the training data
  - and the best estimate of the variance of the Gaussian is…
    - ...the variance of the training data

- This makes things pretty easy, provided that we have
  - training data
  - which are labelled with which class each observation belongs to

# Probabilistic modelling

- Remember the two parts of the DTW
    - compute local distances
    - align the two sequences
- and the associated problems of
    - accounting for variability (temporal and spectral)
    - differing importance of each component of the feature vector
- We want to learn everything from data
    - one solution is to use a **probabilistic model** (sometimes called a stochastic model)

# WAIT! We need to re-think the feature vector first.

- Given what we know about Gaussians, and in particular that
  - modelling covariance requires lots of additional parameters
- We must ask “Is the vector composed of the FFT coefficient values for one frame of speech a suitable feature vector for speech recognition?”
  - Does it capture the required information? **Yes**
  - Does it remove effects of F0 and other unwanted variability? **No, the harmonics of F0 are still present.**
  - Is it fast to compute? **Yes**
  - Are the components of the feature vector uncorrelated? **No, they are highly correlated!**

# A closer look at the ear

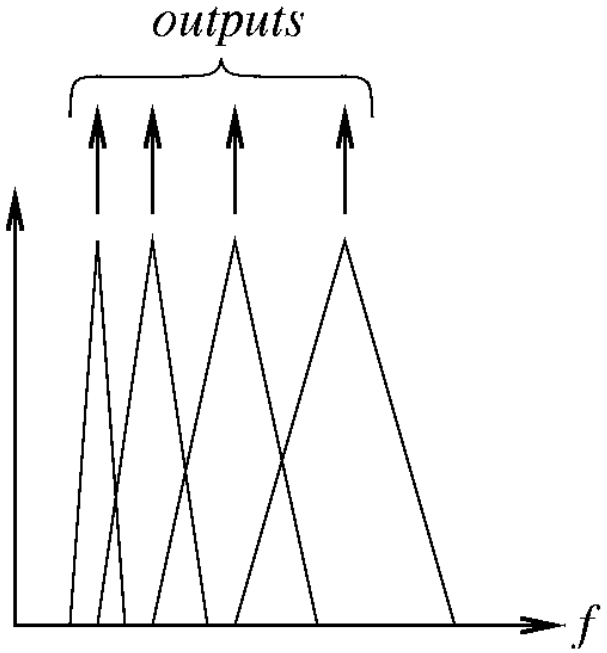
- The ear does more than a simple frequency analysis - it's not just doing something like an FFT
    - it performs various non-linear transformations on the signal
- Some facts about the human auditory system:
    - The loudest tolerable sound has an intensity 1 million million ($10^{12}$) times the lowest intensity sound we can hear
    - We can perceive frequencies from about 20Hz to 16kHz
        - Range decreases with age
    - Sensitivity varies with frequency
        - Most sensitive over frequencies found in speech
        - Frequency discrimination decreases at higher frequencies

# Perceptually motivated analysis

- The most useful properties of human hearing that we can apply to automatic speech recognition are:
  - frequency range is limited (say, to 8kHz for speech)
  - frequency discrimination decreases as frequency increases
  - amplitude scale is non-linear
- and we do this by
  - limiting the sampling rate of the digitised speech to 16kHz
  - warping the frequency scale
  - applying some non-linear compression to the amplitude

# Warping the frequency scale

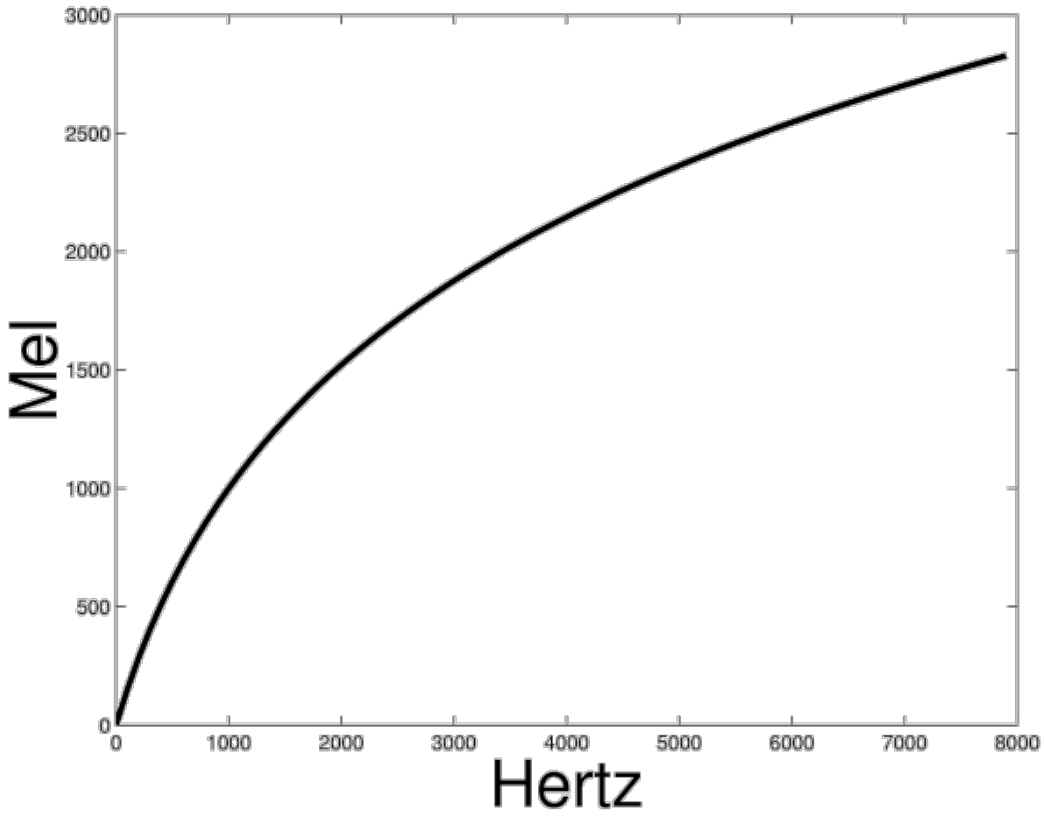
- How do we warp the frequency scale?
- The cochlea in the ear converts from the time domain to the frequency domain using a bank of filters. We can do the same:




- The centres of the filters are spaced on a non-linear scale, which reflects our knowledge of the human auditory system (e.g., the Mel scale)

# The Mel scale

- Perceptual experiments reveal the frequency scale used in the human auditory system. A close approximation to this is the Mel scale:



# Is the set of Mel-scale filterbank outputs a good representation for recognition?

- Does it capture the required information?
  - **Yes**, captures overall spectral shape
  - **Yes**, uses non-linear frequency scale
  - **Yes**, we can apply amplitude compression to the filterbank outputs
- Does it remove F0 effects?
  - **Yes**, it smoothes the harmonics away
- But, there's still one problem:
  - adjacent outputs of the filterbank are **highly correlated**

# Why is correlation a problem?

- Our representation of the speech signal is a vector of numbers for each frame
- Probability distributions of correlated data require extra parameters to describe the amount of correlation (covariance)
- Statistical independence between components of the vector is a **Good Thing**
- The last step is therefore to transform the filterbank outputs:
    - compress the dynamic range by taking the logarithm
    - reduce the correlation by taking a **cosine transform** of the log filterbank outputs

# What's a cosine transform?

- It's a bit like another Fourier transform
    - it captures the **shape of the spectrum** - in other words, the spectral envelope
    - the degree of detail which is captured can be controlled by the number of coefficients we use - and we wish to limit the amount of detail so that
        - only the important properties of the spectral envelope are retained (e.g., formants)
        - unnecessary fine detail is discarded (e.g., effects of F0)
- These are called Mel frequency cepstral coefficients (MFCCs).
- Typically, the first 13 coefficients of the cosine transform are retained.
- *Cepstrum is an anagram of spectrum*

# The feature extraction process from waveform to MFCCs

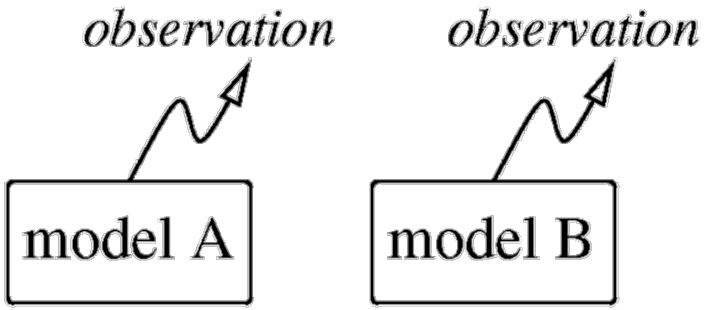
- Take frames of the time domain signal, apply a window
  - 25 ms duration @ 16kHz sampling rate = 400 samples
- Convert to frequency domain
  - take the FFT of each windowed frame
- Perceptual frequency scale warping, smooth away harmonics of F0
  - Mel-scale filterbank, typically 22 filters
- Decorrelate
  - cosine transform, typically retaining first 13 coefficients
- We have reduced the number of parameters needed to represent one frame of 16kHz speech from 400 to 13, by discarding unwanted information

# Speech recognition - lecture 3 of 5

- generative models
- from DTW to HMMs

# Generative models

- We have a model for each **class** (e.g., word) we want to recognise




- For an unknown word to be recognised (**classified**), first obtain the sequence of feature vectors for the incoming speech (*feature extraction*)
- Use each model in turn to generate this sequence
    - During generation, also compute the **probability** that the model generated the observation sequence
- Pick the most probable model

# Generative models

- This is a powerful framework for classification
  - need a model for each **class** (e.g., word or phoneme)
  - model randomly **generates** observations (e.g., sequences of MFCC vectors)
  - model can compute the **probability** that it generated a particular given observation sequence
- The models themselves are **not classifiers**
  - in other words, they do not directly perform the task of recognition
  - rather, they are used to compute the probability that an observation sequence for an unknown word could have been generated by that model
  - given the probabilities for different models, the classification decision can be made (by choosing the most probable model)

# Types of generative model

- The type of model we are looking for should
  - be able to generate all possible observations (so that it can always assign a non-zero probability to any particular given sequence)
  - generate observations from its **own class** with a **high** probability
  - generate observations from **other classes** with a **low** probability
  - be learnable from data

- A probability distribution, such as the Gaussian, has these properties

# Can we do speech recognition yet?

- What are the classes we are trying to recognise?
  - Words?
  - Phonemes?
  - Something smaller?
- Is estimating a single Gaussian for each class going to work?

# Back to DTW

- In DTW, we were matching the stored sequence of feature vectors for a reference word (template) with the incoming sequence for an unknown word
- We are going to replace the stored sequences of feature vectors (i.e., **exemplars**) of the reference words with Gaussian probability distributions (i.e., **models**), trained on multiple examples
- One question is, “How many Gaussians do we need to model our word?”
- Or perhaps, “Are Gaussians alone a sufficient model?”

- The missing link seems to be that
  - a Gaussian can generate a single observation (i.e., a feature vector)
  - but how do we arrange these into a **sequence**

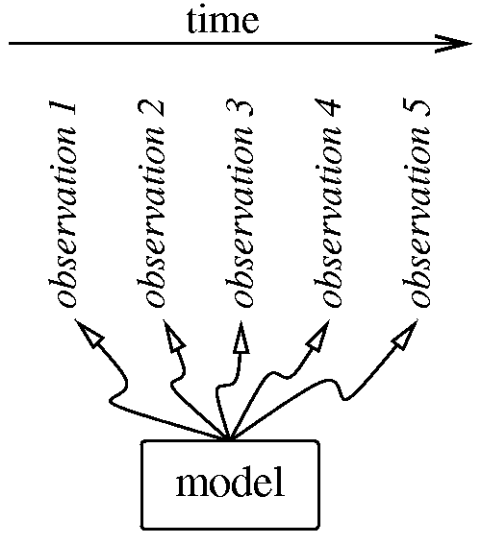
# From DTW to Hidden Markov Models

- So far we have covered
  - Dynamic time warping
  - Distance measures
- Improving the distance measure lead us to
  - Probability distributions
  - In particular: the Gaussian probability density function (pdf)
- We then introduced the concept of
  - Generative models

- Now we can tie this all together…

# The Gaussian as a generative model

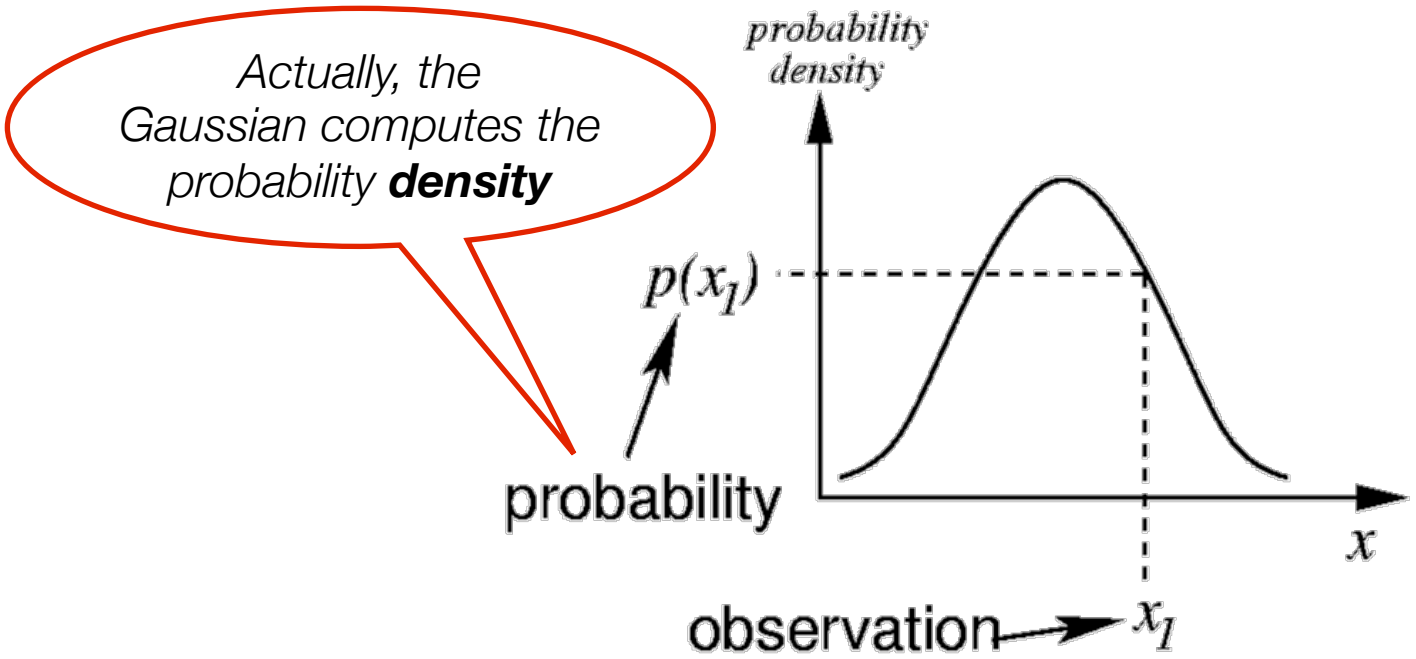
- We can generate a sequence of observations using a Gaussian probability density function simply by repeatedly generating individual observations




- The observations have a Gaussian distribution
- The distribution is constant - it doesn't vary with time (technically, we say that the observations are *independent and identically distributed* )

# Going the other way: given an observation, compute the probability

- For speech recognition, we are given the observation and need to know the probability that a particular Gaussian generated it.
- We can easily work this out:

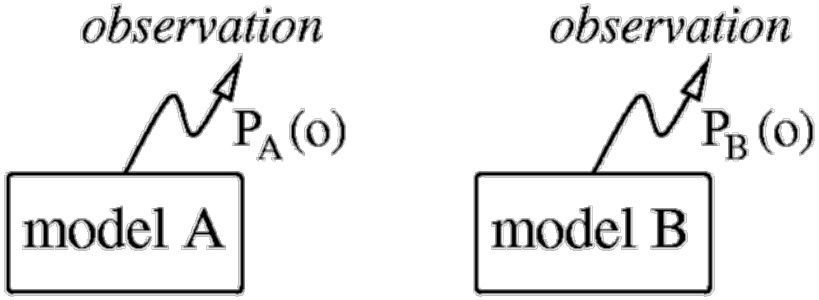


# Views of probability

- There are two ways to look at probability
- Classical or Frequentist (what you learn at school)
    - Count things
    - Repeat experiments multiple times
    - Probability is the fraction of the time that a certain outcome is reached
- Bayesian
    - Degree of belief/certainty, perhaps expressed as a pdf (probability density function)
    - Our degree of certainty can be changed by observing data
        - we combine prior belief with incoming evidence

# Bayesian view of Gaussian generative model

- This is the viewpoint that we will adopt
- The model computes the probability (degree of certainty) that it generated a particular observation
- This computation is easy for Gaussians




- Then, classification is as simple as choosing the most probable model

# Some simple probability

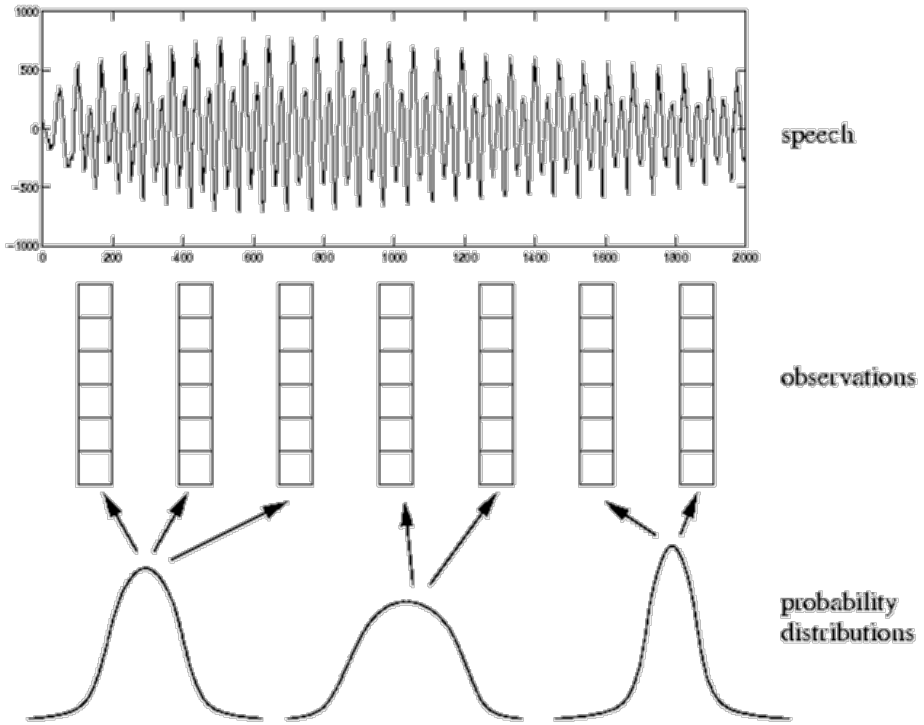
- Before going further we need just a little probability theory
- Imagine two independent events:
  X: It will rain tomorrow
  Y: I'll be wearing blue socks tomorrow
- **Random variables** are written in upper case: X and Y
- The probability of X taking the value x is written P(X=x)
- Or we can write P(X), which means the probability distribution of X
- The probability of both events occurring is computed by multiplying

$$P(X,Y) = P(X)P(Y)$$

  - …because they are **independent**. Independence makes the calculation simple.

# Generating sequences

- Our speech signal has been parameterised as a discrete sequence of observations
- A single Gaussian can generate any number of observations, one after the other



# An assumption of statistical independence

$$\mathbf{O} = \mathbf{o}_1, \mathbf{o}_2, \mathbf{o}_3, \ldots, \mathbf{o}_N$$

- We are going to assume these observations are statistically conditionally independent. That is if $o_1$ and $o_2$ were generated from the same Gaussian then
    - the value of $o_2$ does not depend on the value of $o_1$, given that we know the parameters of the Gaussian
- This is called conditional independence

# Conditional independence

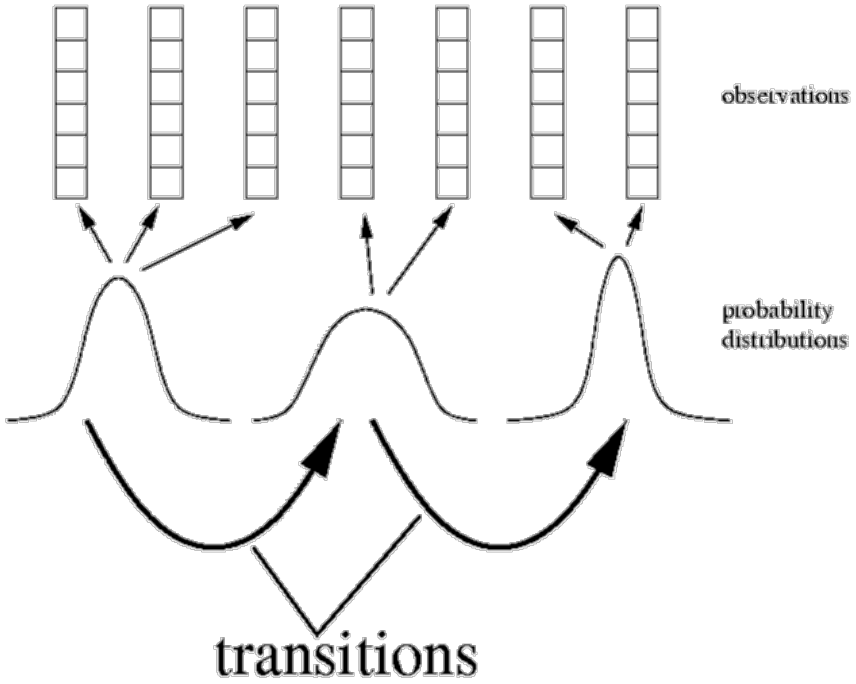
- We know the parameters of the Gaussian that generated $o_2$
- Therefore we know the pdf of $o_2$. This expresses our (Bayesian) beliefs about the sorts of values we might expect $o_2$ to take
- Knowing the value of $o_1$ does not change our beliefs about $o_2$ at all
- This makes the model simpler and the probability calculation easier:

$$P(\mathbf{O}) = P(\mathbf{o}_1)P(\mathbf{o}_2)P(\mathbf{o}_3)\ldots P(\mathbf{o}_N)$$

- But is this a reasonable assumption?
    - (not really, but it makes the model *much simpler* and anyway we'll try to mitigate any problems this causes later on)

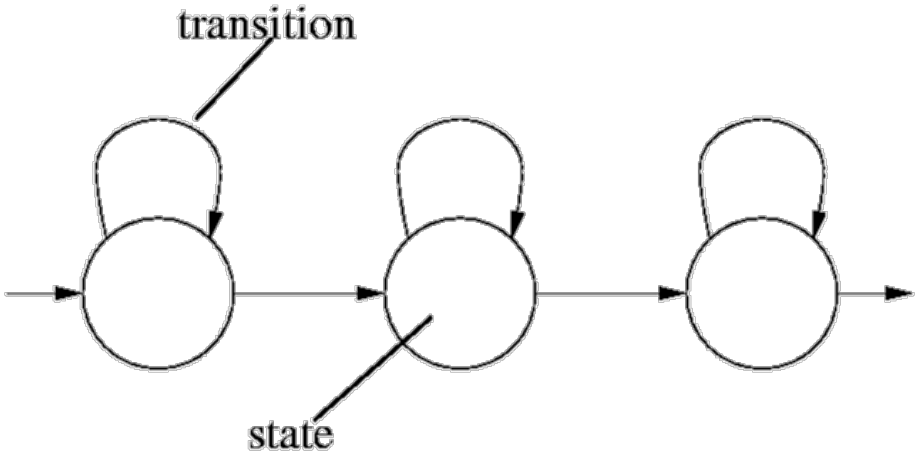
# What about duration?

- How many observations will each Gaussian emit?
  - when do we move from one Gaussian to the next?
- We need to model **duration**



# Finite state machines

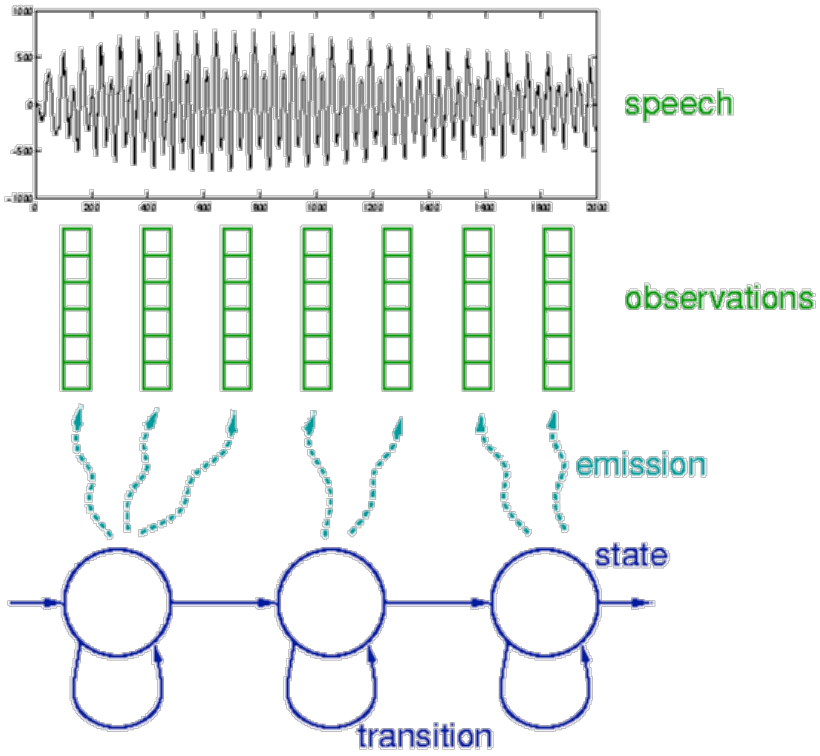
- We need some temporal constraints to
  - 1. use the Gaussians in the correct order (!)
  - 2. model duration




- Transitions have probabilities
- States contain the Gaussian pdfs.

# Generative Markov model

- The model now generates sequences of observations
- Duration of each state controlled by self-transition probability
- Each state emits observations with a Gaussian distribution



# What is Markov about the model?

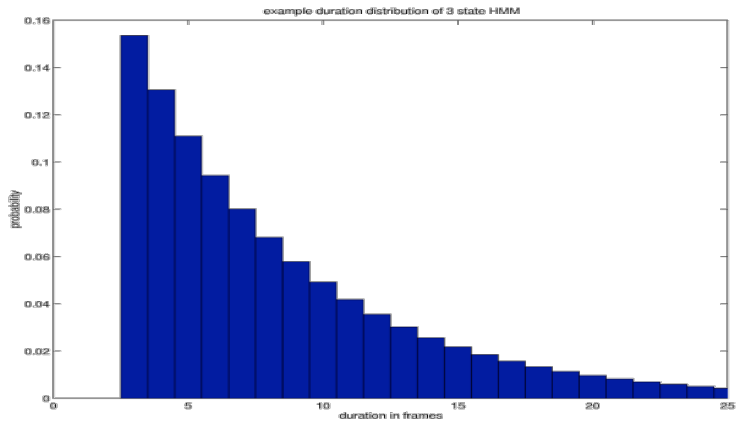
- Markov = “memoryless”

- Applied to our model, this means:
  - the probability of an observation depends **only on the current state** of the model and not on past or future states or observations
  - the probability of arriving in a particular state at time t depends only on which state we were in at time t-1

- These two limitations on our model make it *much* simpler to compute with.

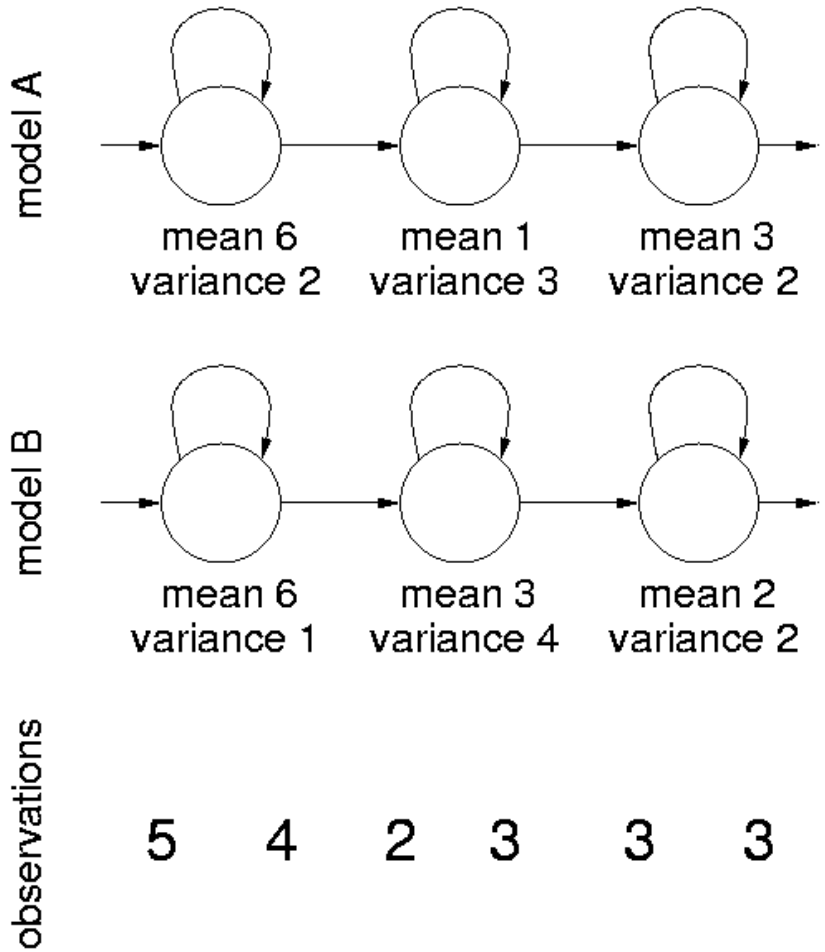
# Duration in Markov models

- The self transition on each state controls the distribution.
- High self transition probability → it is likely that the model will remain in a state for longer
- Low self transition probability → it is likely that the model will spend less time in a state
- The probability of remaining in a state for n frames depends on the self transition probability p multiplied by itself n-1 times. p is always less than 1
- The distribution looks like this:



# Example

- Before looking in more detail at HMMs, an example:



# Example

- Let's ignore the transition probabilities, for simplicity. The PDF of the Gaussian is:

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{(x-\mu)^2}{2\sigma^2}}$$

- Model A:
  Most probable state sequence is 1,1,2,3,3,3
  Emission probabilities 0.18, 0.16, 0.13, 0.20, 0.20, 0.20
  Total probability = 0.000030

- Model B:
  Most probable state sequence is 1,1,2,2,2,3
  Emission probabilities 0.31, 0.15,0.10,0.10,0.10,0.19
  Total probability = 0.000009

- *Note: we actually computed probability density, not probability, but it's OK*

# Multiple paths

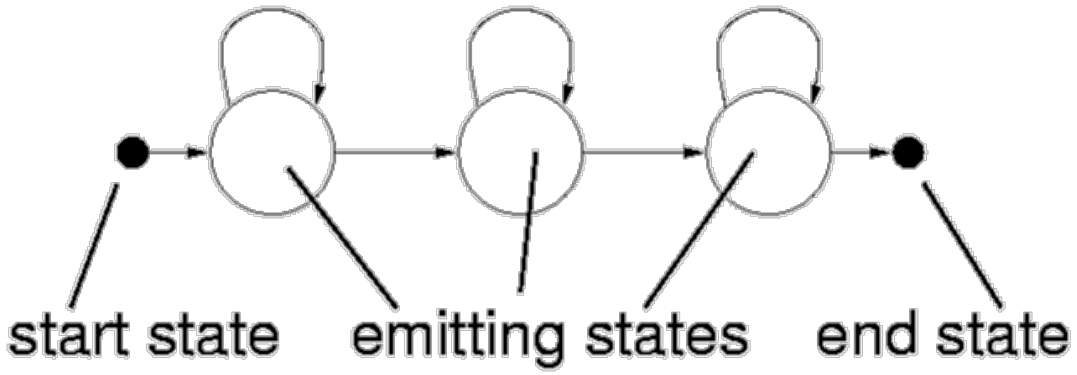
- For a given observation sequence, there are generally multiple paths (i.e., state sequences) through the model
    - it can generate **the same observation sequence in more than one way**
- The probability of each path will generally be different
- Therefore, the probability of a model generating a particular observation sequence is actually computed as the **sum over all possible paths**. This is called the total probability or the forward probability.
- We saw in DTW how many paths there can be (i.e., too many!) – so we can make an approximation:
    - only the most likely path is evaluated
- A path represents a particular alignment between states of the model and observations: we call this the **state sequence**

# What is hidden about the model?

- Because there are multiple paths which lead to the same observation sequence:
    - for any given observation sequence, the actual state sequence is not observable
    - so we say that the state sequence is **hidden**
- We can use dynamic programming (the same algorithm as DTW) to find the most likely path. More on this later…

# Hidden Markov Model

- We can now specify our model fully
- We have:
    - states, transitions, and observation probability density functions
- We will now adopt a useful convention (used in HTK) of dummy start and end states



# Relating this to the assessed practical

- The task is to:
  - Build an isolated digit recogniser
  - Test it
  - Improve it
- The main part of the practical is to make a speaker-dependent, isolated word recogniser.
- You then go on and make as many improvements as time allows:
  - multi-speaker, speaker-independent, connected-digit, anything else you can think of, …

# Data

- We will need some training data
- Quantity?
    - 7 examples of each token
- Labelling?
    - Using wavesurfer
- Parameterisation?
    - MFCCs
- Language?
    - English, as this makes speaker-independent easier (because you can use data from other students)

# Models

- We need to specify our models:
  - How many states?
  - What transitions?
- In HTK this is done by providing a template or *prototype* model
- You are provided with some prototypes, which you can edit if you need to.

# Labelling the data

- We are doing isolated word recognition, so we will label at the word level
- The label set is: zero, one, two, … , nine
- Important: there will be silence between the words in your recording. We don't want to train models on that, so we must label it as something else: use the label 'junk'

- It is very important to divide the data into **training** and **testing** sets
- You cannot test models on the same data they were trained on – that only tests the models' capacity to 'memorise' data
- We want to test the **generalisation** of models to new data

# Language model

- Even for isolated word recognition, HTK needs a language model – we'll see why later. We'll have a very trivial model, which allows sentences to be composed of exactly one digit; all the digits have equal probability.

- You may try connected digit recognition with a small modification to the language model

# Testing the models

- Measuring the performance on unseen data
- You can experiment with the models at this stage:
    - Using your own speech
    - On a new speaker of the same gender
    - On a speaker of a different gender
    - Many experiments are possible

# The HTK toolkit

- We are using version 3 of the HMM toolkit from Cambridge University
- The names of the programs in the toolkit start with **H**
- HCopy parameterises the data
- HInit initialises the models
- HRest trains the models (‘re-estimation’)
- HVite performs recognition using the Viterbi algorithm (i.e., dynamic programming)
- Scripts are provided to make the running of these programs easier
    - you don’t need to change any HTK programs
    - you will need to modify the scripts, in order to conduct your experiments

# General options for HTK tools

- Command line options to the HTK tools are single letters. Upper case letters apply to all the tools, such as
  - **-T 1** gives tracing information
  - **-S <filename>** provides a script file (a list of files to process)
  - **-C <config file>** specifies a configuration file (usually called CONFIG)

# An example

- Some conventions used in the tutorial sheet:
  - **bash $** represents the prompt. Don't type it
  - **<filename>** means fill in a filename here (don't type the < > part)
  - **** means command continues onto the next line

```
bash $ HCopy –T 1 \
-C resources/CONFIG_for_coding \
wav/train_data.wav mfcc/train_data.mfcc
```

# File types

- The filename endings depend on the type of file:

  **file.wav** waveform

  **file.mfcc** MFCC coefficients

  **file.lab** text labels

  **file.rec** recogniser output labels

# Speech recognition - lecture 4 of 5

- conditional probability
- Bayes' rules
- priors and posteriors
- the Viterbi algorithm implemented as token passing
- sub-word models
- language models

# Conditional probability

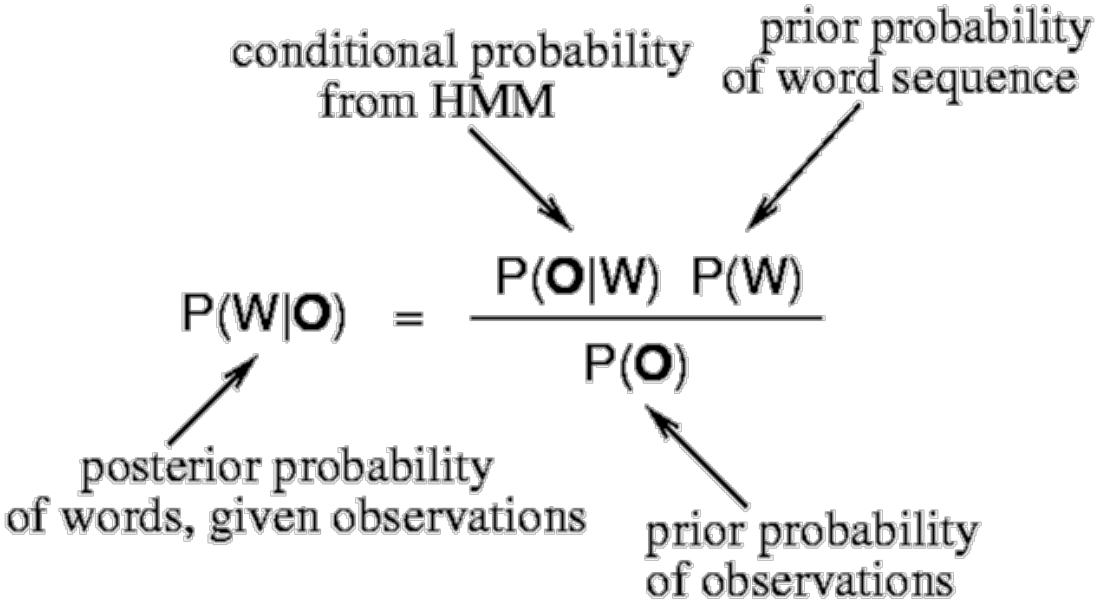
- So far, we have made lots of independence assumptions
- What happens when events are not independent?
    - for example, predict someone's height, given their gender
- We can express this in a Bayesian way: what is the probability that a person's height H is equal to the value h, **given that** their gender G is equal to g?
    - The notation for conditional probability is **P( H=h | G=g )**
    - or simply P(h|g)
- Our HMM also computes a conditional probability: what is the probability of a sequence of observations, given a particular word model?
    - We write that as **P( O | W )**

# Bayes' rule

- In speech recognition, what we actually want to know is the probability of a word, given the observation sequence: P(W|O)
- This is a problem, because we can only compute P(O|W) with the HMMs
- Fortunately there is a law of probability that says:
  - P(O,W) = P(O|W) P(W)
- and so by symmetry:
  - P(O,W) = P(W|O) P(O)
- Putting these together gives us Bayes' rule: $P(W|\mathbf{O}) = \frac{P(\mathbf{O}|W)P(W)}{P(\mathbf{O})}$

# Bayes' rule and speech recognition

- Note: W can be a single word, or a sequence of words (or any units). This formula works in all cases.



# Interpreting Bayes' rule

- There are 3 terms on the right hand side of Bayes' rule as used for speech recognition

- P(O|W) – the **likelihood** of the observations, computed by the HMM
- P(W) – the **prior** probability of the word (or word sequence)
- P(O) – the **prior** probability of the observations.
  *For any single observation sequence, this is constant, so can be ignored*

- We need to compute the first two terms in order to evaluate P(W|O)

# Bayes' rule

- There is no need to evaluate P(O) because it is constant for any given utterance we are recognising

$$P(W|\mathbf{O}) = \frac{P(\mathbf{O}|W)P(W)}{P(\mathbf{O})}$$

$$P(W|\mathbf{O}) \propto P(\mathbf{O}|W)P(W)$$

- The process of speech recognition can now be stated as:
  - find the **most likely** word sequence
  - in other words, find the value of W that maximises P(W|O)

# The prior probability

- The term P(W) is called the **prior** probability of the word sequence.
- Prior means that it can be computed before any observations have been made.
- In Bayesian terms:
    - it expresses our prior beliefs about what word sequences are likely
- In other words, it is our model of language

- We'll be looking at ways of computing this term later.

# The posterior probability

- The term P(W|O) is called the **posterior** probability of the word sequence.
- Posterior means that it is computed after the observations have been made
- In Bayesian terms:
    - it expresses our **revised** beliefs about W, now that we have received new information in O

# The Viterbi algorithm

- In DTW: many possible alignments between two sequences
- HMMs: many possible alignments between HMM states and observations
    - we call this alignment the **state sequence**
    - it's the counterpart of the **path** in DTW
- By definition, an HMM computes P(O|W) by adding up the total (forward) probability over all possible state sequences that could have generated O
- But we usually make an approximation: only the probability of the **single most likely** state sequence is computed
- We are assuming that the probability of the most likely path is a good approximation (or at least proportional to) the total probability. This turns out to be quite a good assumption

# Numerical issues

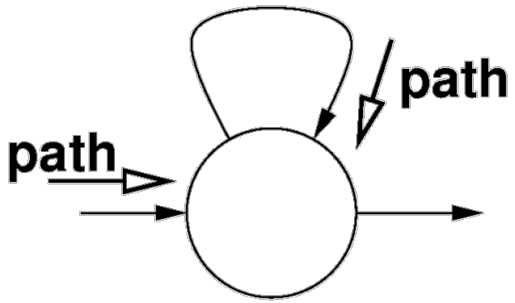
- As we saw last time, even with a trivial example the probabilities can get VERY small indeed.
    - because we are multiplying together lots of numbers (probabilities and probability densities) that are less than 1
- Problem: computers can only represent numbers with a certain precision (just like having a fixed number of decimal places on a pocket calculator)
- Result: “underflow”
- Solution: work in log probabilities
    - these are what you will see in HTK output

$$
\begin{aligned}
\log_{10} 100 &= 2 \\
\log_{10} 10 &= 1 \\
\log_{10} 1 &= 0 \\
\log_{10} 0.1 &= -1 \\
\log_{10} 0.01 &= -2 \\
\log_{10} 0.000001 &= -6 \\
\log_{10} 0.000000361 &\approx -6.442
\end{aligned}
$$

# The Viterbi criterion

- We have already seen this in DTW. Here is the Viterbi criterion for HMMs

The most likely complete path passing through a particular state at a particular time must contain the most likely path arriving at that state at that time

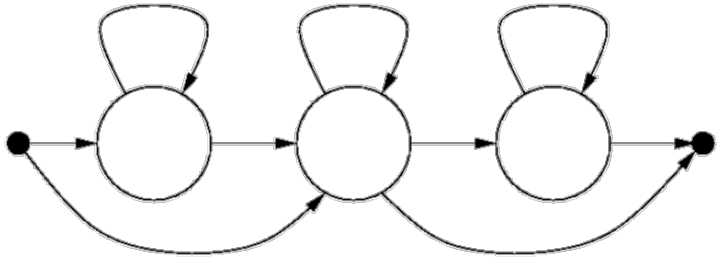


# Concepts in token passing

- Remember:
    - We are searching for a path through the model. During the search, we consider many partial paths: some are extended, others are abandoned.
- Tokens
    - Represent partial paths
    - Hold a (log) probability
    - Have a record of their path so far
- Token passing
    - Extends partial paths forward in time
    - Tokens move from state to state, along the arcs in the HMM

# Starting and ending probabilities

- An HMM consists of states (containing observation pdfs) and transitions
- We need to specify the probability of starting (and of ending) in each state

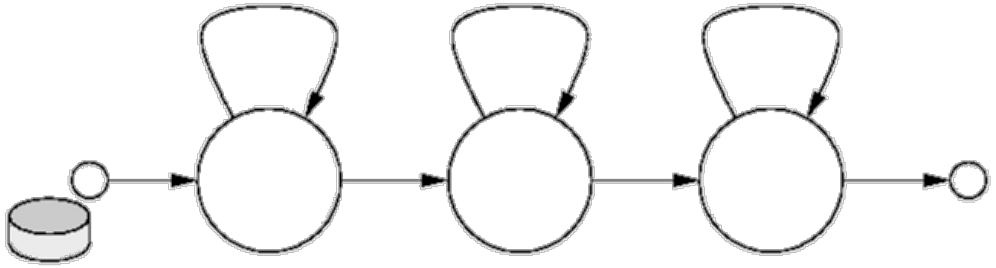


- We can represent this as a simple matrix. For example:

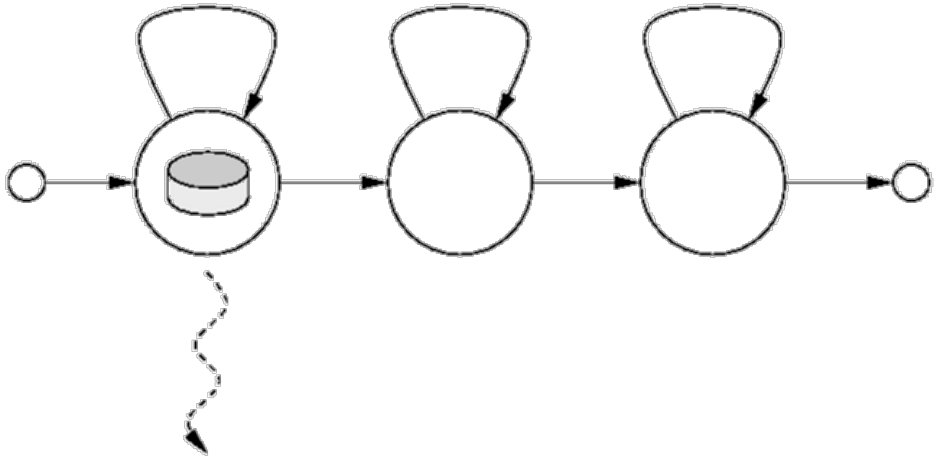
$$\begin{array}{ccccc} 0.0 & \mathbf{0.7} & \mathbf{0.3} & 0.0 & 0.0 \\ 0.0 & \mathbf{0.6} & \mathbf{0.4} & 0.0 & 0.0 \\ 0.0 & 0.0 & \mathbf{0.7} & \mathbf{0.2} & \mathbf{0.1} \\ 0.0 & 0.0 & 0.0 & \mathbf{0.8} & \mathbf{0.2} \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{array}$$

# Token passing in action (generative view)

- We need some initial conditions: there is a partial path (token) in the start state, with probability=1

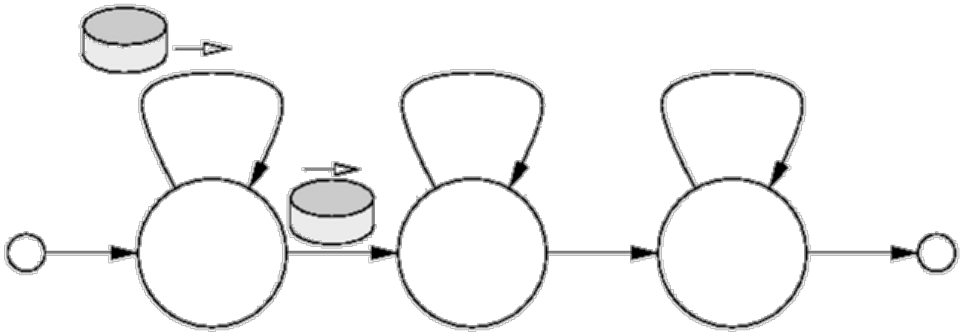


- At each time step (frame), we move the token along an arc, and then generate an observation

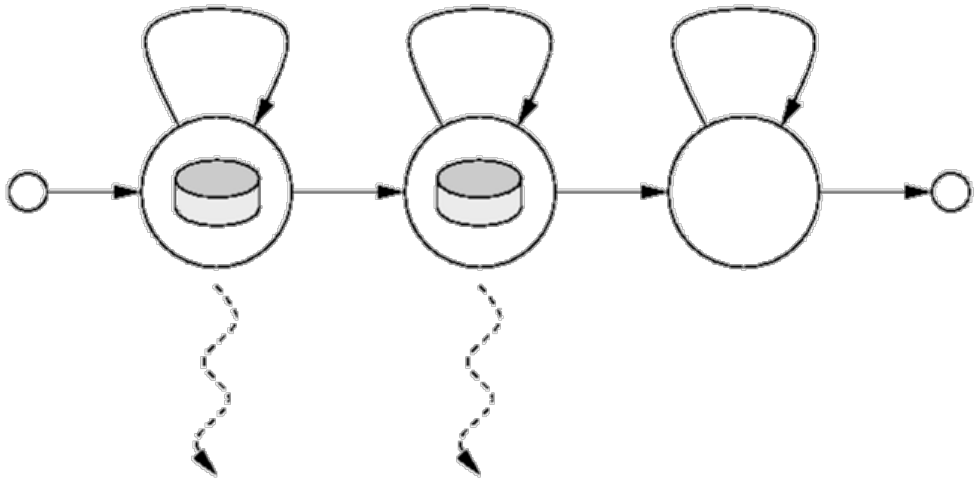


# Token passing in action

- At the second step, there are two arcs leaving state 2. We send copies of the token along both arcs

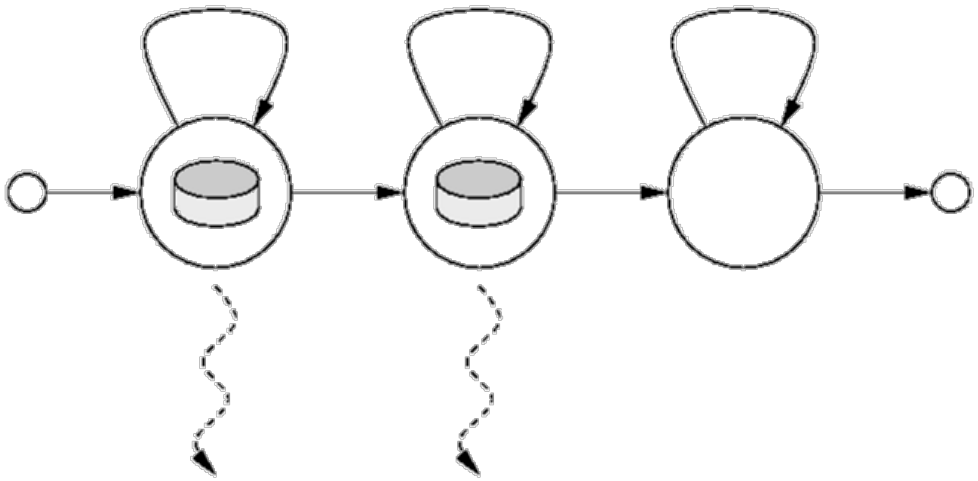


- and each token then generates the second observation:



# Token passing is a parallel search

- At a given time step (frame), each token generates the same observation.
  - e.g., at time t=2, observation $o_2$ is generated



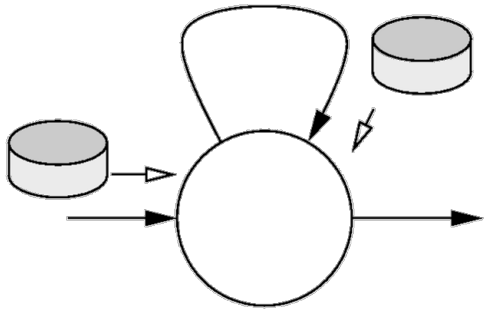
- Each token is exploring a different path through the model.
- Since there is generally more than one token “alive” at any given instant, this means that multiple paths are all being explored in parallel

# Token passing as recognition

- It is a **generative model**: an observation is generated as a token enters or re-enters a state.
- For recognition, what we actually do is calculate the probability that the state in question would generate the given observation
- We add this log probability to the log probability total stored in the token
- We also add the transition log probabilities to this score as we move from state to state

# Viterbi criterion for token passing

- Whenever two, or more, tokens arrive in a state at the same time, we only keep the most likely one:



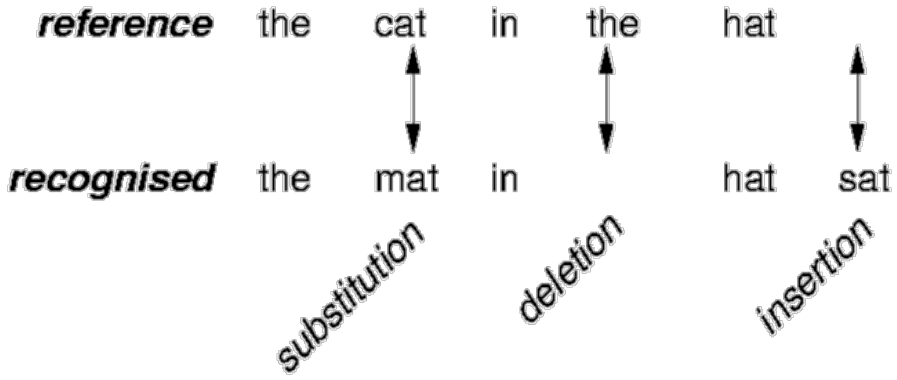
- Which is analogous to two paths arriving at the same point in the DTW grid

# Pruning

- Even with the Viterbi algorithm, recognition can still be too computationally expensive (i.e., too slow and/or uses too much memory)
- It is usual to do **additional pruning**. This involves discarding tokens whose probability has fallen below some defined level
- How do we define this level?
    - Relative to the best token (i.e. best partial path at this time)
        - Beam search
- Pruning makes the search faster, but it is now possible to prune the path that would have gone on to “win”, so error rates may increase
- It may be acceptable to slightly increase the error rate in return for a speed up in computation by a factor of 10 or 100

# Performance evaluation

- Standard measure is word error rate on a test set. For connected speech: three types of errors:




$$\text{Correct} = \frac{N - (del + subs)}{N} \times 100\%$$

$$\text{Accuracy} = \frac{N - (del + subs + ins)}{N} \times 100\%$$

- Where N is the number of words in the reference transcription

# Interpreting results

- It is easy to get very high percent correct, by making lots of random insertions.
- Accuracy is a better measure, since it take the number of insertion errors into account.
- Accuracy can be negative when there are a lot of insertions

- The word error rate is defined as:
  - WER (%) = 100 – Accuracy

- The alignment between the reference transcriptions and the recogniser output is determined by dynamic programming

# Continuous speech

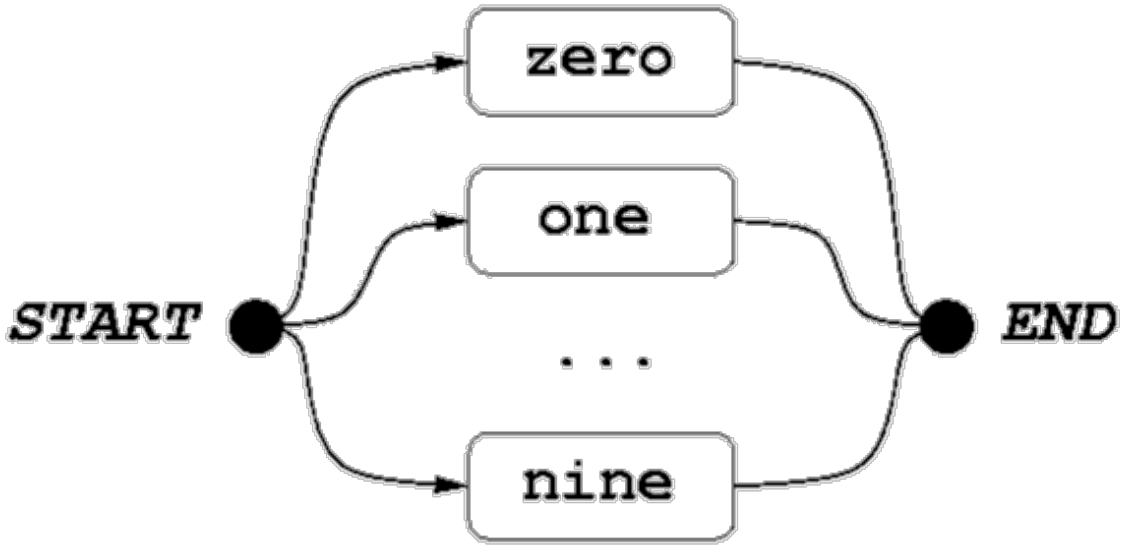
- Things we would like our recogniser to work with:
    - A larger vocabulary
    - Continuous speech (rather than isolated words)
    - New (unseen) words
- What changes do we have to make?

# Larger vocabulary: sub-word units

- Let's say that a word model needs 10 training examples.
  - Small vocabulary: it is practical to collect training data for every word
  - Larger vocabulary: this is no longer practical
- How do we train models for words we have no training data for?
- Use sub-word models
  - Typically phonemes
  - Can write the pronunciation dictionary by hand (just a mapping from words to phoneme strings)
  - Models of words will be built up by joining together models of phonemes

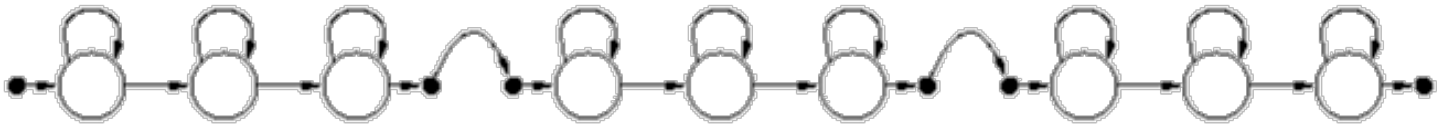
# Language model: first attempt

- We will start with a very simple language model.
  - e.g., for our digit recogniser:



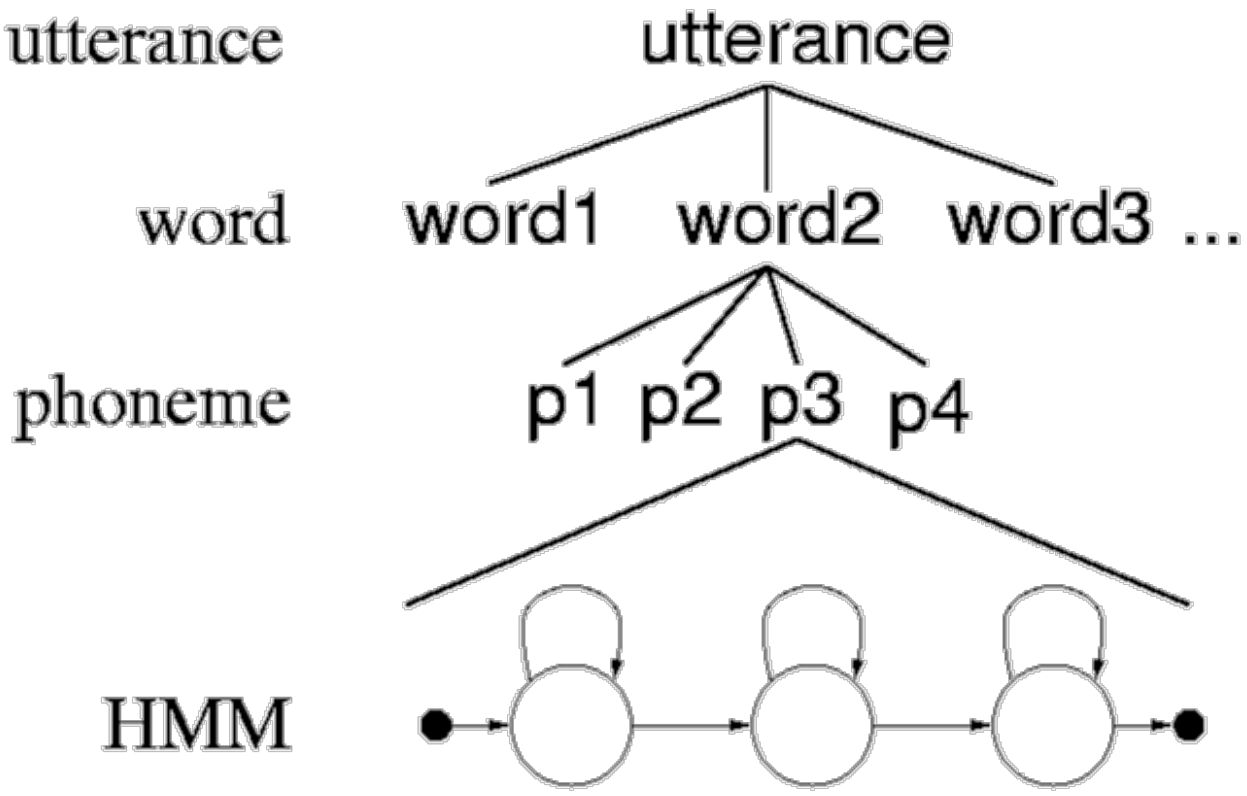
# A useful property of HMMs

- Now we come to a very important property of HMMs
  - They can be joined together to make larger HMMs
- Consider the word “cat” with pronunciation /k ae t/



- The model for the word “cat” is simply made by joining the models of the constituent phones /k/ /ae/ and /t/
- Because the result is just another HMM, we can use all the same training and recognition techniques.

# From HMMs of phones to HMMs of utterances



# Token passing for connected models

- Now we can see why token passing is such a useful view of Viterbi search
    - **The algorithm for connected word recognition is exactly the same as for isolated words**
- This is because we have joined our words together with arcs, just like the internal HMM arcs
- The arcs that join words are the language model and can have probabilities too.

# Back to Bayes

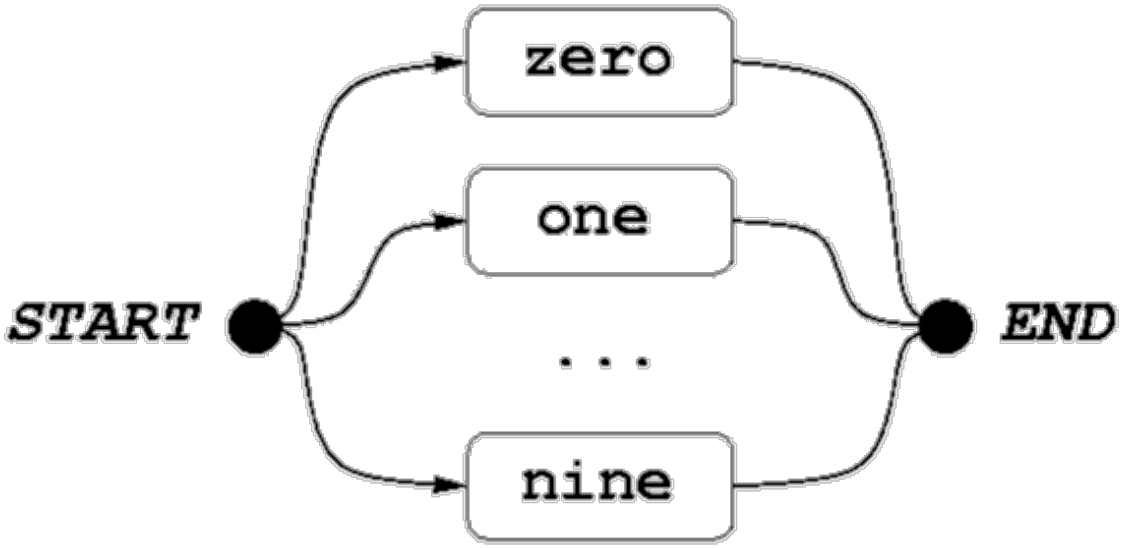
- Remember we are trying to compute:

$$P(W|\mathbf{O}) = \frac{P(\mathbf{O}|W)P(W)}{P(\mathbf{O})}$$

- P(W) is the probability of the word sequence W and is computed by the language model
- First, let's look at a very simple form of P(W): models where P(W) is either 0 or some constant non-zero value
    - We might call these non-probabilistic models
    - Word sequences are either allowed, or not allowed.

# Finite state networks

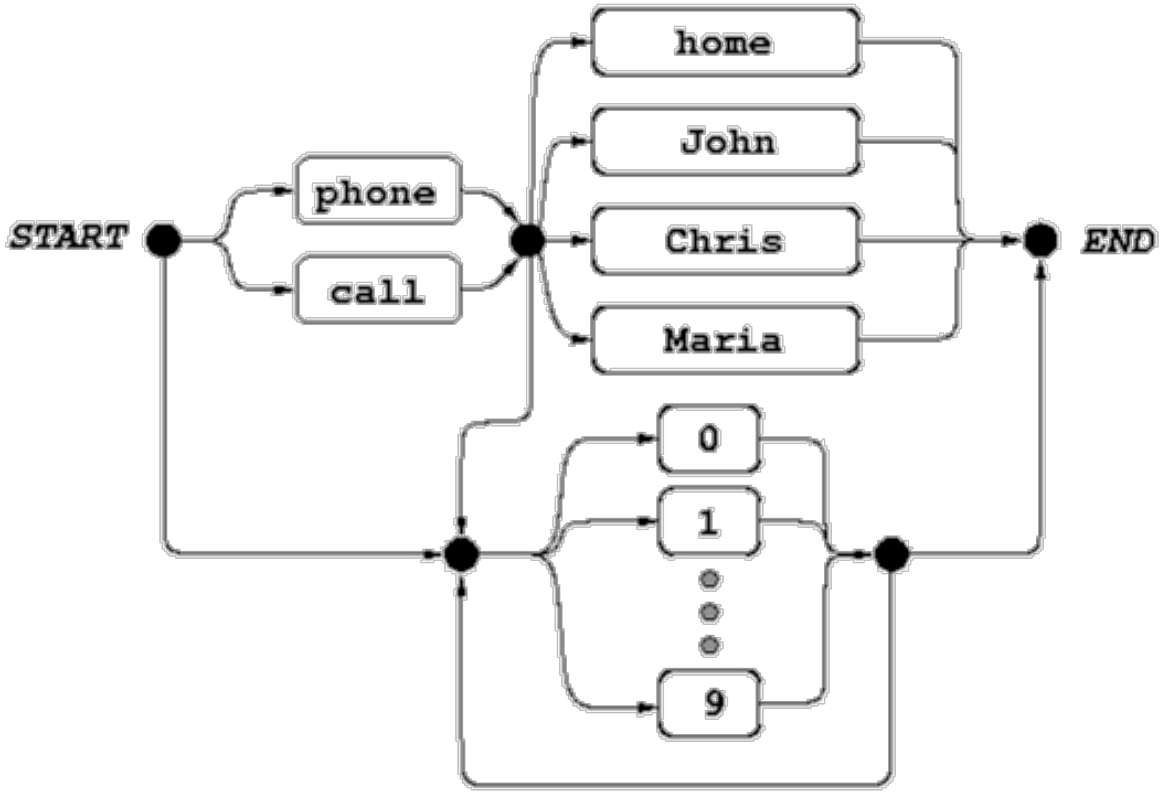
- A natural way to express constraints like this is a finite state network
  - e.g., for our isolated digit recogniser:




- We can simply write this down by hand

# Another finite state example: voice dialling

- Again, we can write this down by hand – no need to train on data



# Word-pair models

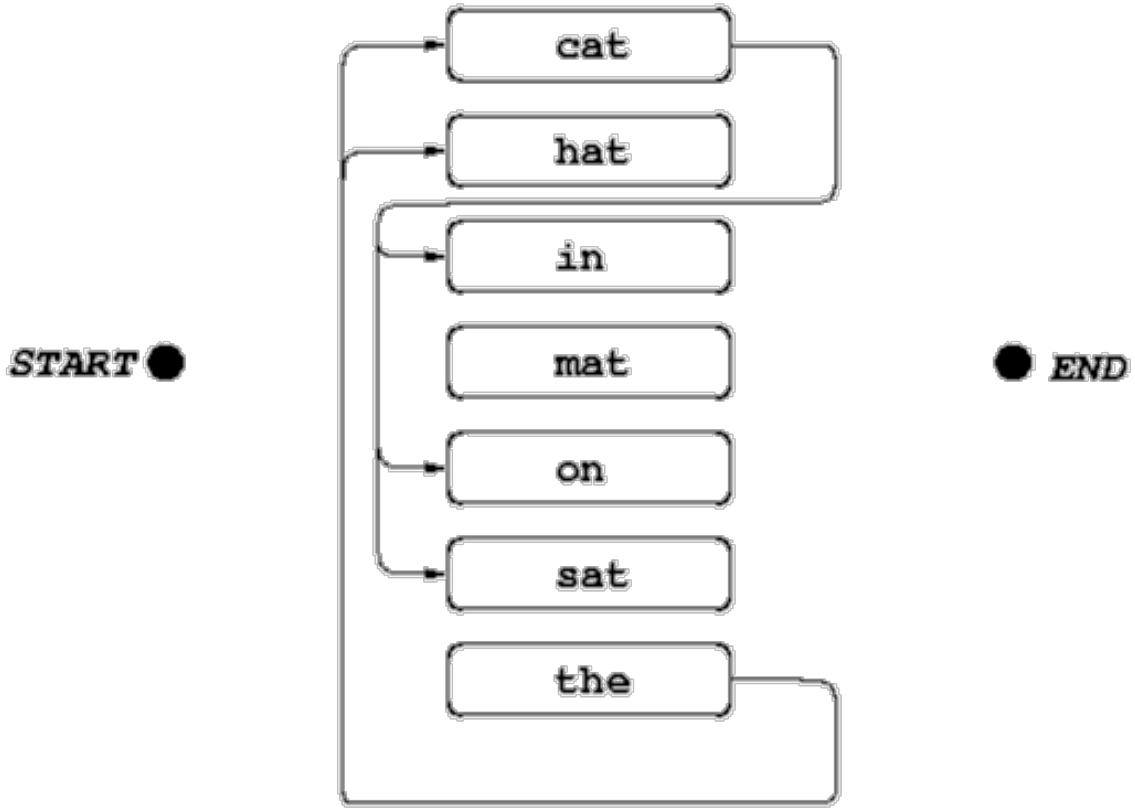
- Writing these networks by hand will quickly become tedious for larger tasks.
- One alternative is a word pair model:
- For each word in the vocabulary, we just list all the allowed following words:

| word | allowed followers |
|---|---|
| the | cat, hat |
| cat | in, on, sat |
| sat | in, on |

- We could learn this from data, or write it by hand

# Word-pair finite state model

- The word-pair model can be represented by a finite state network
- Here is a fragment

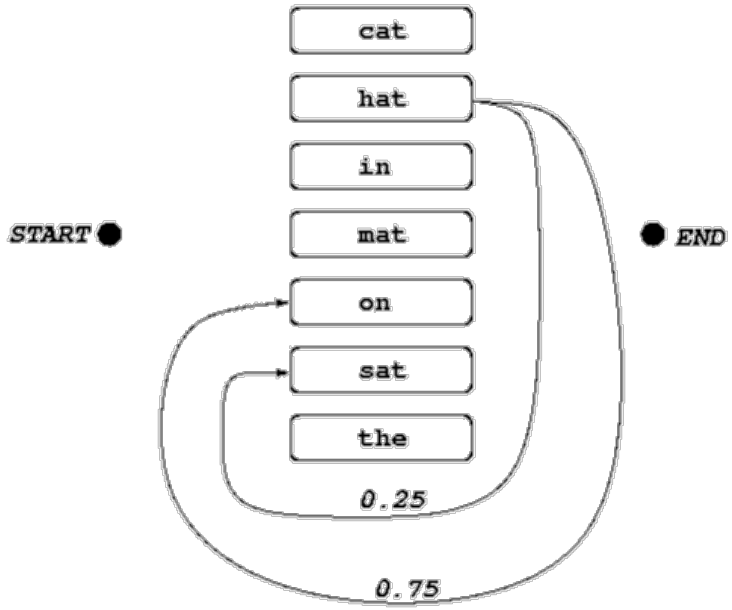


# What do we want from a language model?

- Intuition: as we add more pairs to the word-pair model, it becomes weaker.
- In the limit:
  - All pairs are allowed
  - Therefore all word sequences are allowed
  - The language model (LM) then does nothing useful
- We want our LM to **discriminate** between different word sequences
- Remember, we are trying to compute the P(W) term
- **Probabilistic models**

# Probabilistic word-pairs

- Here's part of a finite state probabilistic word-pair model:




- If any word can be followed by any other word, and all arcs have probabilities, this is a **bigram language model**
    - Details can be found in the readings

# Speech recognition - lecture 5 of 5

- training HMMs

# Training HMMs

- Recap: we already know how to estimate parameters of a Gaussian PDF
- Existing concept: maximum likelihood (ML) estimation
- Problem in extending this to HMM training: state sequence is hidden
- Solution: an iterative training algorithm
- New concept: **expectation-maximisation** (EM)
- We'll start with the simpler case: Viterbi training (as used in HInit)
    - using the single most likely state sequence
- Then EM training using the Baum-Welch algorithm (as used in HRest)
- We'll assume isolated word data & whole word models. Connected speech is left for the ASR course next semester

# Estimating the parameters of a Gaussian

- We already saw that it's easy to get ML estimates for these parameters from training data $O=\{o_1,o_2,\ldots,o_N\}$. We can state (*a proof is beyond our scope*):
  - ML estimate of the mean is: the mean of the training data:

$$\hat{\mu} = \frac{1}{N}\sum_{i=1}^{N} o_i$$

  - ML estimate of the variance is: the variance of the training data:

$$\hat{\sigma}^2 = \frac{1}{N}\sum_{i=1}^{N} (o_i - \hat{\mu})^2$$

# The training problem

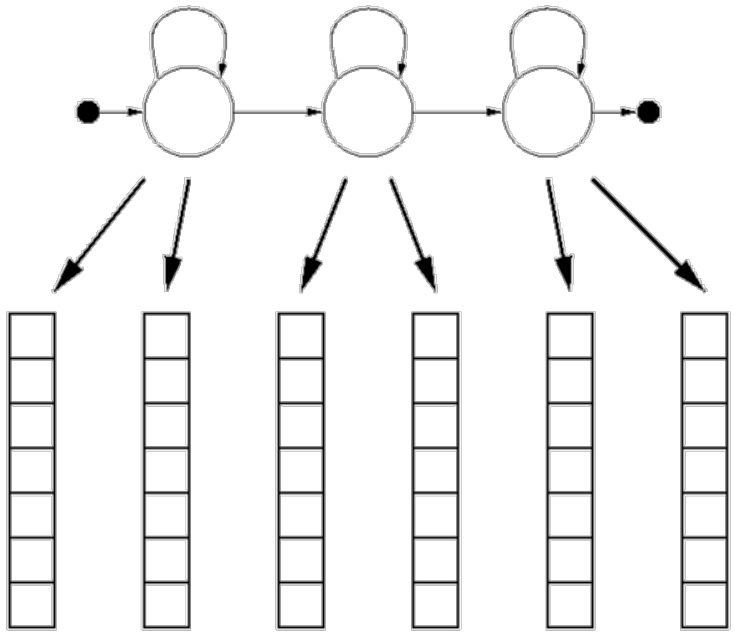
- The Hidden Markov model is a generative model
- States generate observations with distributions according to their probability density function (typically a Gaussian)
- But **the state sequence is hidden**
    - we say “hidden” because there are many possible state sequences that can generate the same observation sequence
- We can estimate parameters of each Gaussian only if we have some observations that we know were generated by that Gaussian
- Catch-22
    - we need to know which state generated which observation!

# Two simple training methods

- We have a seemingly impossible task, because the state sequence is hidden
- A first approximation is to consider only a single state sequence (just like we do during recognition) - the state sequence is no longer hidden
- Uniform segmentation
    - A one-step solution
- Viterbi training
    - An iterative method
- We'll focus on estimating the observation densities, and ignore the transition probabilities

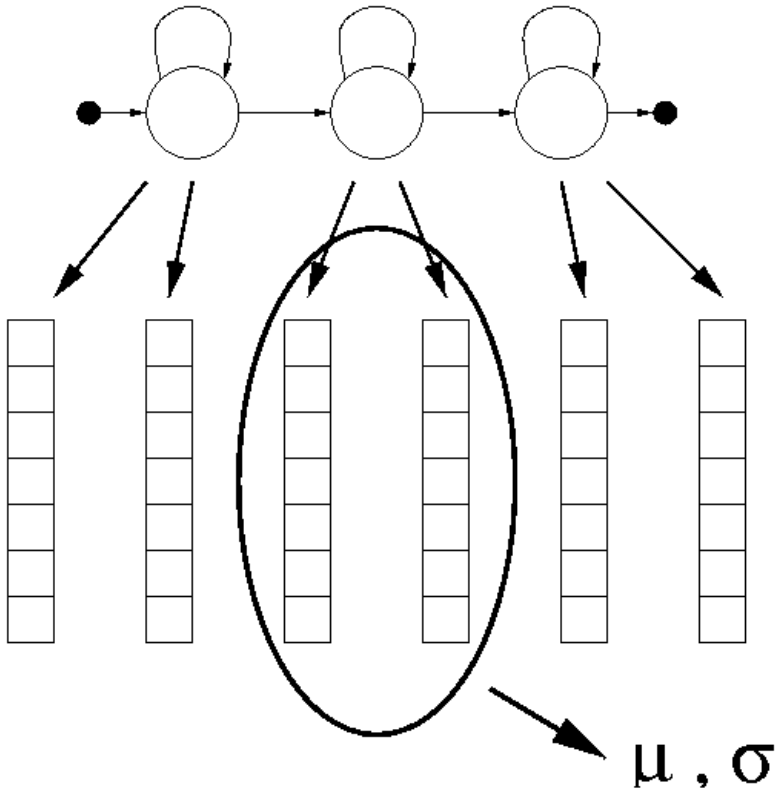
# Uniform segmentation: alignment

- Consider a model with 3 states and a single training example with 6 frames
- Segmentation: align each observation with exactly one state – the simplest thing is to divide the observations as uniformly as we can amongst the states:



# Uniform segmentation: parameter estimation

- The parameters of the Gaussian for each state are now computed from the observations aligned with that state
    - no point repeating this procedure because we'd get the same outcome every time



# New concept: iterative methods

- There is no simple solution for the ML estimate of the parameters of our HMM
- Clearly, a uniform segmentation is sub-optimal
- We will have to use an iterative method
    - Take a model
    - Somehow increase the likelihood of the data given the model
    - Repeat until likelihood doesn't increase any further
- In this way, we can get iteratively closer and closer to the "best" parameters for our model
- We define "best" as those parameters that maximise the likelihood of the training data – that is, the ML estimate

# Viterbi training

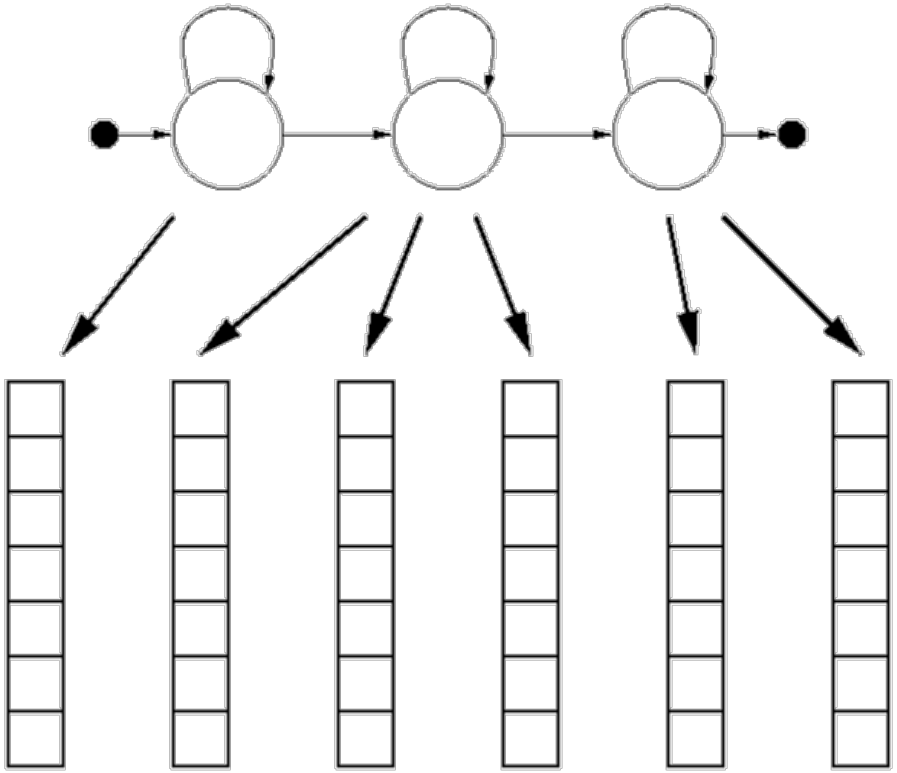
- An iterative method
- Can we find a better alignment than uniform segmentation?
    - Of course we can: the Viterbi algorithm finds the most likely alignment of states and observations
- So, given a model, we can
    - align the observations with the states
    - use this alignment to re-estimate the Gaussian means and variances
    - and so get better estimates of the Gaussian means and variances

# Viterbi training

- Take a model
- Align the observations to states
    - *Important: note that the alignment we find depends on the model parameters*
- For each state
    - Take the observations aligned to it
    - Compute their mean and variance
    - Update the parameters of the Gaussian to be that mean and variance
- Iterate

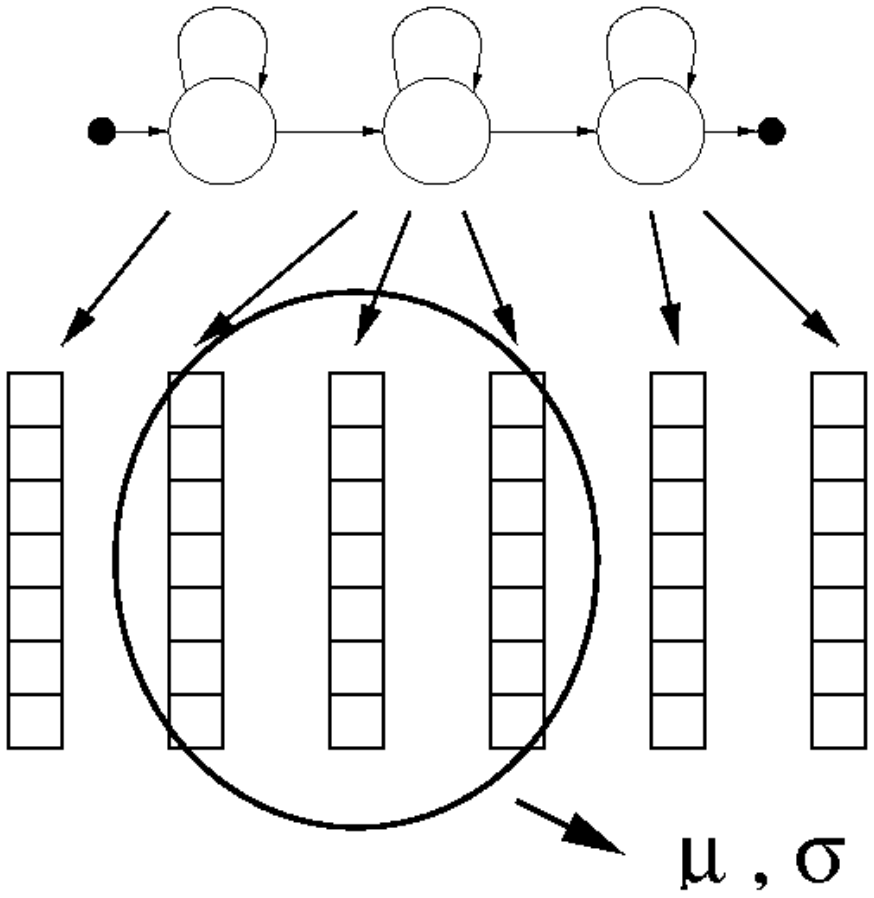
# Viterbi training: alignment

- Find the most likely alignment of states and observations



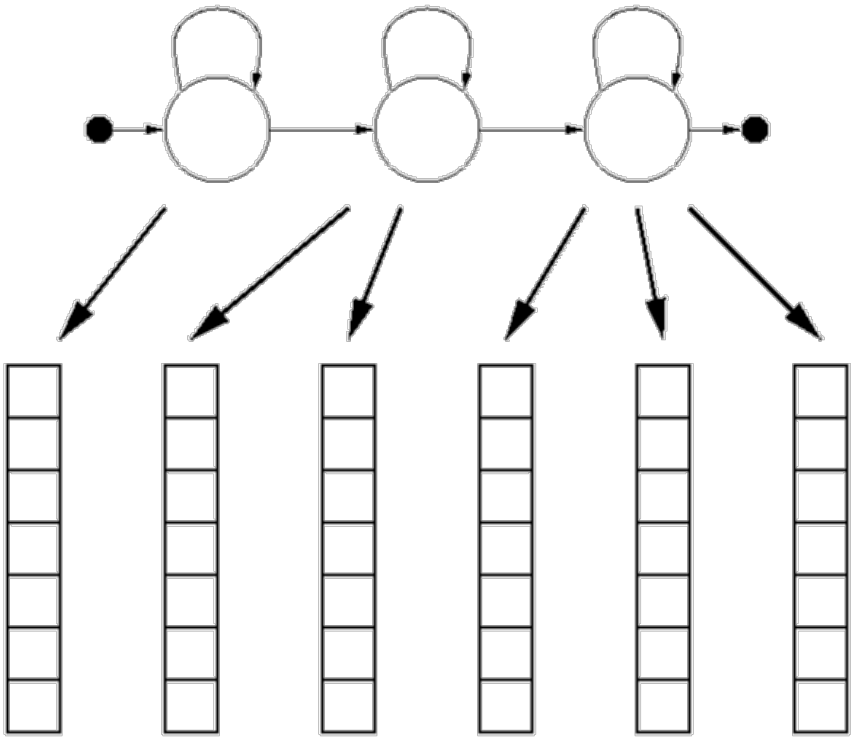
# Viterbi training: parameter update

- Update parameters:



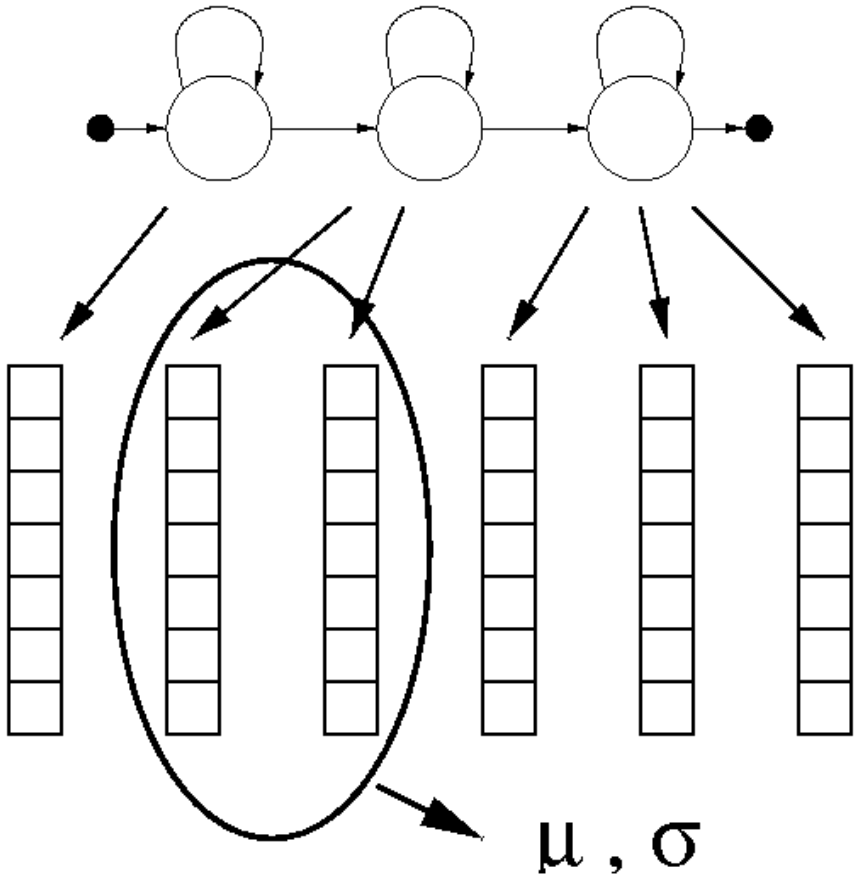
# Viterbi training: iterated alignment

- With the new model parameters, find the most likely alignment of states and observations:



# Viterbi training: iterated parameter update

- Update parameters again



# A better method

- So far, we have only considered the single, most likely, state sequence
    - What about all the other state sequences?
- We will now take a look at the standard training algorithm for HMMs, without any mathematical derivation (*that's left  for the ASR course, but if you have the mathematical ability you might want to look at a textbook now*)
- We know by now:
    - There are many state sequences that can generate any given observation sequence
    - In other words, the state sequence is a hidden variable
- and we also know:
    - Each of these state sequences has an **associated probability**

# Models with hidden variables

- Our model has an observed variable (the observation sequence) and a hidden variable (the state sequence)
- If we knew the values of the hidden variable, we could do simple ML estimation (that is what we just did in Viterbi training)
- A common solution for models with hidden variables is called **Expectation-Maximisation**
- The idea is to:
    - Average over all possible values of the hidden variables, using the current model parameters (that's the **Expectation** bit)
    - Update the model parameters to maximise training data likelihood (that's the **Maximisation** bit)

# Back to Viterbi training for a moment

- The observation sequence O={$o_1$,$o_2$,…,$o_N$} is aligned with the states
  - Here, each observation aligns with exactly one state
- For a particular state *s*, our estimate of the mean is the mean of the observations aligned with that state:

$$\hat{\mu}_s = \frac{1}{M} \sum_{i=1}^{N} o_i P(i)$$

- Where P(*i*) = 1 if the observation *i* was aligned with state s
- And P(*i*) = 0 otherwise
- And M is a normalising factor

# Consider all state sequences

- We could consider all state sequences and weight each one by the probability that it generated the observation sequence
- Previously, for a particular state *s* we had this:

$$\widehat{\mu}_s = \frac{1}{M} \sum_{i=1}^{N} o_i P(i)$$

- which might become:

$$\widehat{\mu}_s = \frac{1}{M} \sum_{i=1}^{N} o_i P(\text{of being in state } s \text{ at frame } i)$$

- We have weighted each observation in the sum (i.e., the computation of the mean) by the probability that this state (*s*) generated it

# Expectation-Maximisation

- Now we have an iterative algorithm to estimate the parameters of our HMM
- There are two phases to each iteration:
    - **E step**: compute the expectation (“averages”), given the current model parameters and the training data
    - **M step**: update the model parameters
- The algorithm is iterative and goes: E step, M step, E step, M step, ...
- The EM algorithm guarantees to increase (in fact, *not decrease*) the likelihood of the training data each iteration, until convergence
- The increases tend to get smaller and smaller as the algorithm runs

# EM for HMMs

- The EM algorithm as applied to HMMs is called the **Baum-Welch algorithm**
- The E step consists of computing expectations such as the expected probability of being in each state at each time
- The M step updates the parameters (means and variances of the Gaussians, transition probabilities)

- *We are not going to cover the update formulae for the transition probabilities*

# What is convergence: when does training stop?

- The EM algorithm only guarantees to incrementally improve the likelihood of the training data
    - this likelihood converges to some value as training proceeds
    - training stops when the improvement per iteration becomes small
- But, EM cannot guarantee to increase the likelihood of the test data
- So, there is no guarantee of good performance
    - **We** must engineer good performance into the system
    - typically, by ensuring that the training data are representative of the sort of test data we expect to see.

# The complete speech recogniser

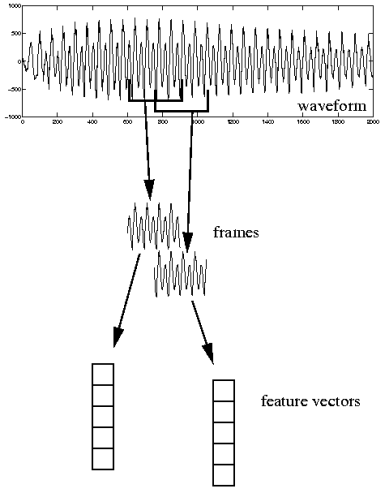
- Parameterisation of speech
- Acoustic models
- Dictionary (*only when using sub-word units - not the focus of this course*)
- Language model
- Architecture
- Search
- Performance evaluation
- Integration with other systems

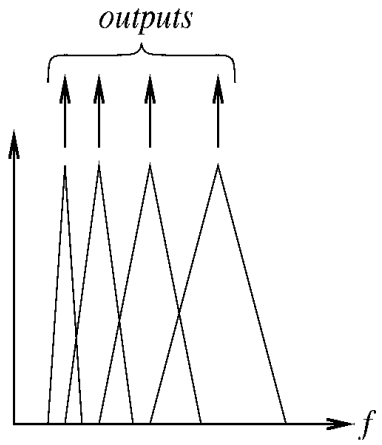
# Two views

- Two different views of speech recognition
    - Flowchart
    - Generative model
        - Recogniser as probabilistic generator of observations
        - Utterance model generates words
        - Word model generates phones
        - Phone model generates observations
- The second view fits nicely with our Bayesian formulation

# Parameterisation

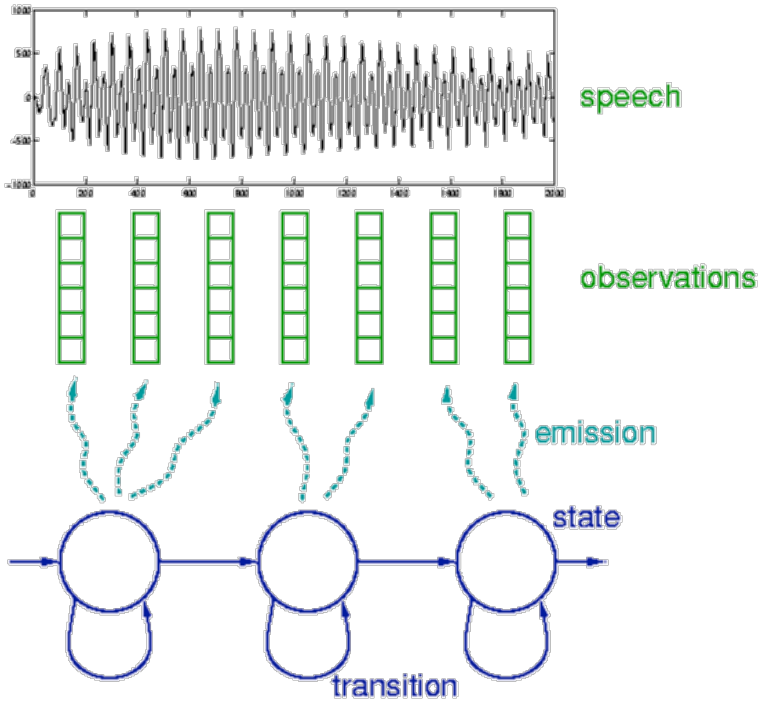
- Sampling: must use high enough sample rate to capture all important frequencies
- Frame based analysis:
  - assume short-term stationarity
  - choose a frame duration (25 msec) and frame spacing (10 msec) to suit speech
- Smooth away F0 and apply perceptual weighting: Mel-scale filterbank
- Amplitude compression: take logs
- Reduce dimensionality & decorrelate: cosine transform
- The result: MFCCs



# Acoustic models

- Hidden Markov models
- Assume observations are independent samples from Gaussian distributions
- Any state can generate any observation
  - state sequence is hidden
- Left-to-right model topology for speech



# Language model

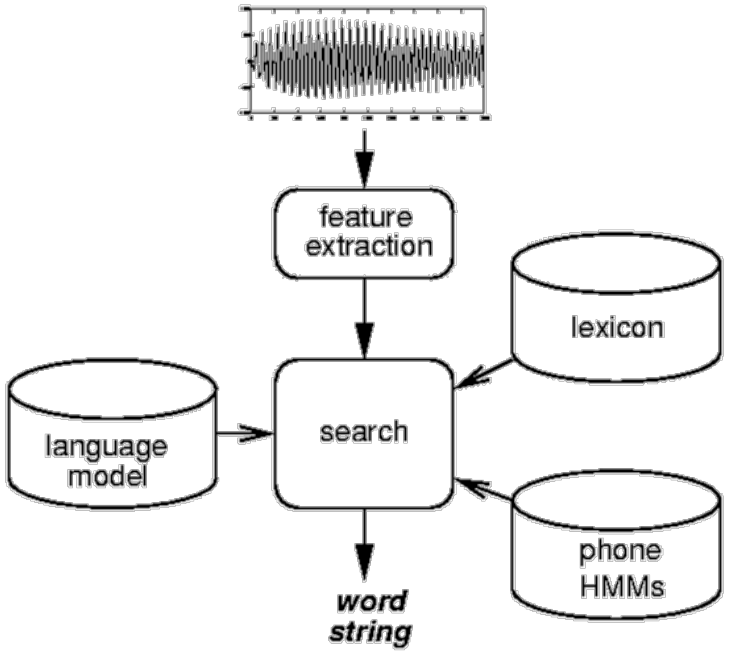
- Very simplistic models used in ASR
  - N-grams
- Trained from data (*not covered in this course*)
- Potentially very large number of parameters
  - Real systems use a mixture of 3- 2- and 1-grams (a backed-off model)
  - e.g. typical backed-off trigram model for Wall Street Journal task takes 100+ Mb of storage and contains:
    - 20k unigram
    - 5 million bigrams
    - 6.7 million trigrams

# Architecture

- Need to implement the recogniser
- Elegant implementation for HMMs with finite state LMs
    - We can compile together the language model, pronunciation models and hidden Markov models into one big finite state network
    - …and use token passing
- Problems with this approach:
    - The compiled network will get very large for big language models
    - Only works for time-synchronous search (more about this in the ASR course)

# A flow chart

- If you like that sort of thing…

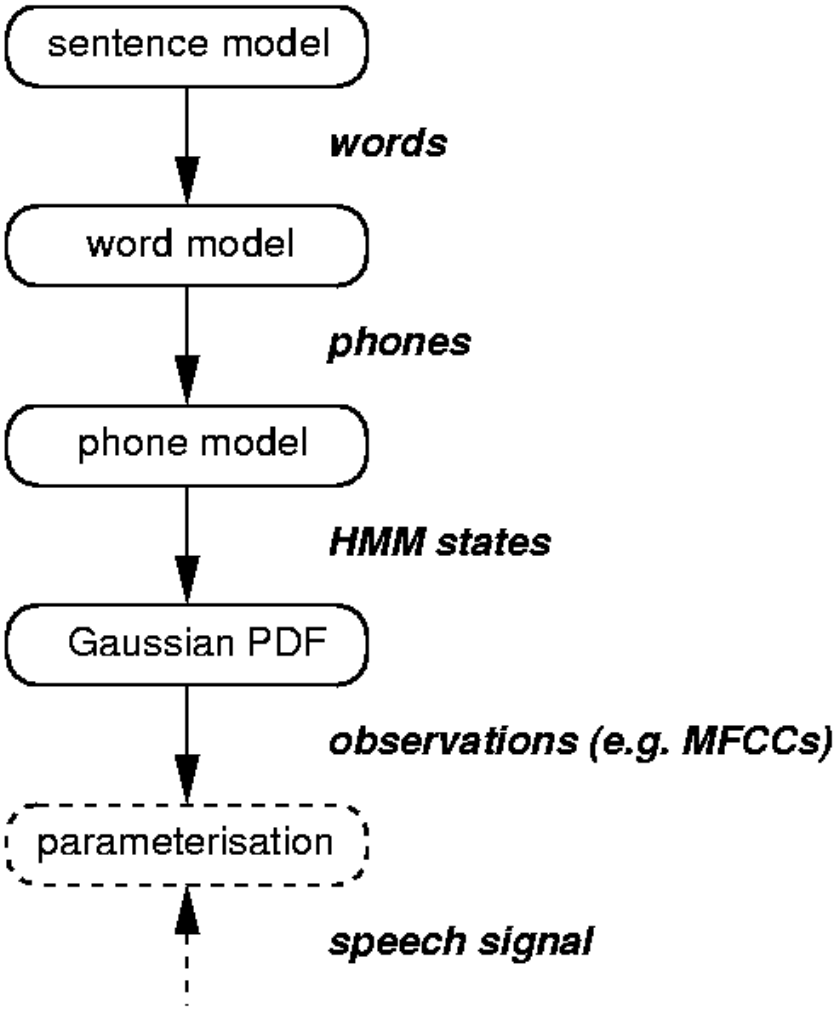



- The LM, lexicon and phone models are constraints on the search space

# A unifying view

- Bayes' formula allowed us to write down a probability we need to compute in terms of things we can compute
- This lead to an implementation in which we compiled together our models at sentence, word and phone levels.
- The result was a single model
    - More importantly, a single generative model
- So, rather than think of a flowchart, think in terms of a generative model of speech

# The generative model view



# Search

- With all our models compiled into a single network, the problem becomes one of performing a search for the most likely word sequence
- The Viterbi criterion is the key
    - It is applied right down to the HMM state level
- Is it enough?
    - Can we make the search faster?
    - For large vocabulary systems with trigram language models, the search will still be slow unless we do quite a lot of additional pruning

# Pruning

- Without pruning, there is always one token in every state of the model
  - Many tokens will have low likelihoods
  - We can discard some tokens
  - This reduces the amount of computation
  - Recognition is faster
- Common pruning method
  - beam search
- There is always a risk of discarding the correct word sequence (pruning error)