# Hidden Markov Models (HMMs)

This talk gives a brief tutorial overview of HMMs and their uses. It is designed to give an intuitive understanding of the workings of the models as well as to point out some common pitfalls when implementing them.

## Brief Outline

1. Hidden Markov Models:
   – some history
   – generative model
2. Evaluating the probability of a sequence
3. Learning model parameters
4. Practical Pitfalls

Sam Roweis

## Some History

- Markov ('13) and later Shannon ('48,'51) studied *Markov chains*.
- Baum and colleagues (BP'66, BE'67, BS'68, BPSW'70, B'72) developed much of the theory of "probabilistic functions of Markov chains".
- Viterbi ('67) (now Qualcomm) came up with an efficient optimal decoder for state inference.
- Applications to speech were pioneered independently by:
  - Baker ('75) at CMU (now Dragon)
  - Jelinek's group ('75) at IBM (now Hopkins)
  - communications research division of IDA (Ferguson '74 unpublished)
- Dempster, Laird & Rubin ('77) recognized a general form of the Baum-Welch algorithm and called it the *EM* algorithm.
- A landmark open symposium in Princeton ('80) hosted by IDA reviewed work till then.

## Generative Model

- plain-vanilla HMM $\equiv$
  "probabilistic function of a Markov chain":
  1. use a 1st-order Markov chain to generate a hidden state sequence (path)
  2. use a set of output prob. distributions $A_j(\cdot)$ (one per state) to convert this state path into a sequence $\sigma_t$ of observable symbols or vectors
- Notes:
  - the Markov chain is characterized by transition probabilities $T_{ij}$ and a distribution $T_{0j}$ over the initial state
  - even though hidden state seq. is 1st-order Markov, the output process is not Markov of *any* order [ex. 1111121111311121111131. . . ]
  - discrete state, discrete output models can approximate any continuous dynamics and observation mapping even if nonlinear; however lose ability to interpolate
  - to communicate $N$ bits of information into future, need $2^N$ states!

## Using HMMs for Recognition

- Use many HMMs for recognition by:
  1. training one HMM for each class
     (this requires *labelled* training data)
  2. evaluating the probability of an unknown sequence under each HMM
  3. classifying the unknown sequence by the HMM which gave it the highest likelihood
- This requires the solution of two problems:
  1. Given a model, evaluate the probability of a sequence. (We can do this exactly and efficiently.)
  2. Give some training sequences, estimate model parameters. (We can find the local maximum of parameter space nearest our starting point.)

## Probability of an Observed Sequence

- To evaluate the probability, we want:

$$\sum_{\text{all paths}} prob[\,\text{observed seq} \mid \text{path}\,]$$

  which looks like an extremely hard computation because the number of possible paths grows exponentially with number of states ($N$) and time steps ($\tau$): paths $= N^{\tau}$

- But: there exists a clever "forward recursion" to compute the sum efficiently. Define $\alpha_j(t)$:

$$\alpha_t(j) = prob[\,\sigma_1^t \quad \text{and state at time } t = j\,]$$

  Now induction comes to our rescue:

$$\alpha_1(j) = T_{0j} A_j(\sigma_1)$$

$$\alpha_{t+1}(k) = \left( \sum_{s_j \in \mathcal{S}} \alpha_t(j) T_{jk} \right) A_k(\sigma_{t+1})$$

  Which enables us to easily (cheaply) compute the desired likelihood $L$ since we know we must end in some possible state:

$$L = \sum_{s_k \in \mathcal{S}} \alpha_{\tau}(k)$$

# Bugs

- Naive algorithm:
  1. start a bug at state $0$, time $0$ holding value $0$
  2. move each bug forward in time by making copies of it and incrementing the value of each copy by the probability of the transition and symbol emission
  3. go to 2 until all bugs have reached time $\tau$
  4. sum up values on all bugs
- Clever recursion:
  adds a step between 2 and 3 above which says: at each node, replace all the bugs with a single bug carrying the sum of their values

# Model Training

- Intuition: if only we *knew* the true state path then ML parameter estimation would be trivial.
- And: we can *estimate* the state path using a trick similar to the one above.
- What if: we estimate the states, then compute params, then re-estimate states, and so on ...
- Turns out: this works and we can *prove* that it converges to local maximum.
- However: we can also *prove* that finding the ML parameters is NP complete, so initial conditions matter a lot and convergence is hard to tell.
- This is the *Baum-Welch* or *forward-backward* algorithm and it is a special case of the *EM* algorithm.

## State Path Estimation

- If we knew the total prob. of all paths going through state $j$ at time $t$ we could compute the *conditional* prob. of being in state $i$ at time $t$ given the observations:

$$\gamma_t(i) = \frac{\alpha_t(i)\beta_t(i)}{L}$$

where we defined:

$$\beta_t(j) = prob[\sigma_{t+1}^{\tau} \quad \text{and state at time } t = j]$$

- $\alpha_t(i)$ gives the total *inflow* of probability to node $t(i)$ while $\beta_t(i)$ gives the total probability of *outflow*
- Bugs again: we just let the bugs run forward from time $0$ to $t$ and backward from time $\tau$ to $t$.

# Single Best Path

- The numbers $\gamma_t(j)$ above gave the probability distribution over all states at any time.
- By choosing the state $\gamma_t^*$ with the largest probability at each time, we can make a "best" state path. This is the path with the maximum expected number of correct states.
- But it *is not* the single path with the highest likelihood of generating the data. In fact it may be a path of prob. zero!
- To find the single best path, we do *Viterbi decoding* which is just Bellman's dynamic programming algorithm applied to this problem.
- The recursions look the same, except with $\max$ instead of $\sum$.
- Bugs once more: same trick except at each step kill all bugs but the one with the highest value at the node.
- There is also a modified Baum-Welch training based on the Viterbi decode.

# New Parameters are just Ratios of Frequency Counts

- The initial state distribution is the expected number of times in state $i$ at time 1:

$$\gamma_1(i)$$

- The output distributions are the expected number of times we observe a particular symbol in a particular state:

$$A_j(x) = \sum_{t|\sigma_t = x} \gamma_t(j) \,/\! \sum_{t=1}^{\tau} \gamma_t(j)$$

or merely the state-probability weighted mean and variance for a continuous Gaussian model.

- The expected number of transitions from state $i$ to $j$ which begin at time $t$ is:

$$(1/L)\alpha^t(i) T_{ij} A_j(\sigma_{t+1}) \beta_{t+1}(j)$$

so the estimated transition probabilities are:

$$T_{ij} = (1/L) \sum_{t=1}^{\tau-1} \alpha^t(i) T_{ij} A_j(\sigma_{t+1}) \beta_{t+1}(j) \,/\! \sum_{t=1}^{\tau-1} \gamma_t(j)$$

## Practical Pitfalls

- Numerical Scaling: the probability values that the bugs carry around are tiny and so can easily overflow. There is a good trick to re-scale these values during the calculations while keeping almost the same equations.
- Computational Load: you spend most of your cycles working out useless contributions that are tiny. "Sparse" variants of Baum-Welch compute most cells only every once in a while.
- Training Data Requirements: full covariance matrices in high dimensions or discrete symbol models with many symbols have lots of parameters. To estimate these accurately requires a lot of data. Instead, we often use mixtures of diagonal covariance Gaussians or tie states together in other ways.
- Multiple observation sequences: can be dealt with by averaging numerators and averaging denominators in the ratios given above. Note the serious error on p.370 of Rabiner & Juang.

## Relationship to Kalman Filters

- Kalman filter models (linear dynamical systems with Gaussian noise) are exactly the continuous state analogue of Hidden Markov Models.
- The forward algorithm corresponds precisely to what is known as the Discrete Kalman Filter.
- Think about why there is no equivalent of Viterbi decoding.
- Think about why it is harder to use mixtures of Gaussians in place of single Gaussians.