Design and Implementation of Speech Recognition Systems

Spring 2012

Class 9: Templates to HMMs
20 Feb 2012

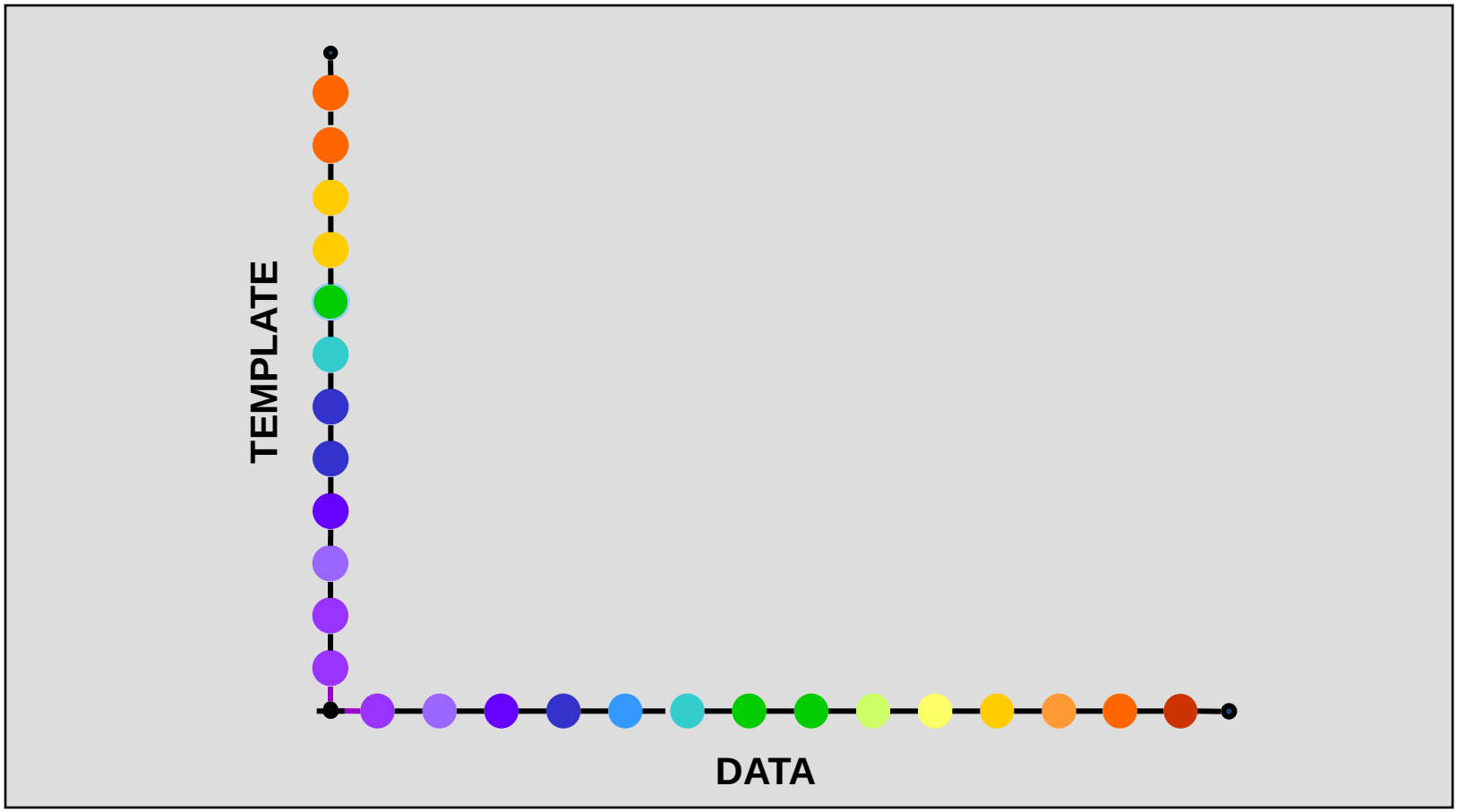
Recap

- Thus far, we have looked at dynamic programming for string matching,
- And derived DTW from DP for isolated word recognition
- We identified the search trellis, time-synchronous search as efficient mechanisms for decoding
- We looked at ways to improve search efficiency using pruning
  - In particular, we identified *beam pruning* as a nearly universal pruning mechanism in speech recognition
- We looked at the limitations of DTW and template matching:
  - Ok for limited, small vocabulary applications
  - Brittle; breaks down if speakers change

Today's Topics

- Generalize DTW based recognition
- Extend to multiple templates
- Move on to Hidden Markov Models
- Look ahead: The fundamental problems of HMMs
  - Introduce the three fundamental problems of HMMs
    - Two of the problems deal with *decoding* using HMMs, solved using the *forward* and *Viterbi* algorithms
    - The third dealing with estimating HMM parameters (seen later)
  - Incorporating *prior knowledge* into the HMM framework
  - Different types of probabilistic models for HMMs
    - Discrete probability distributions
    - Continuous, mixture Gaussian distributions

DTW Using A Single Template



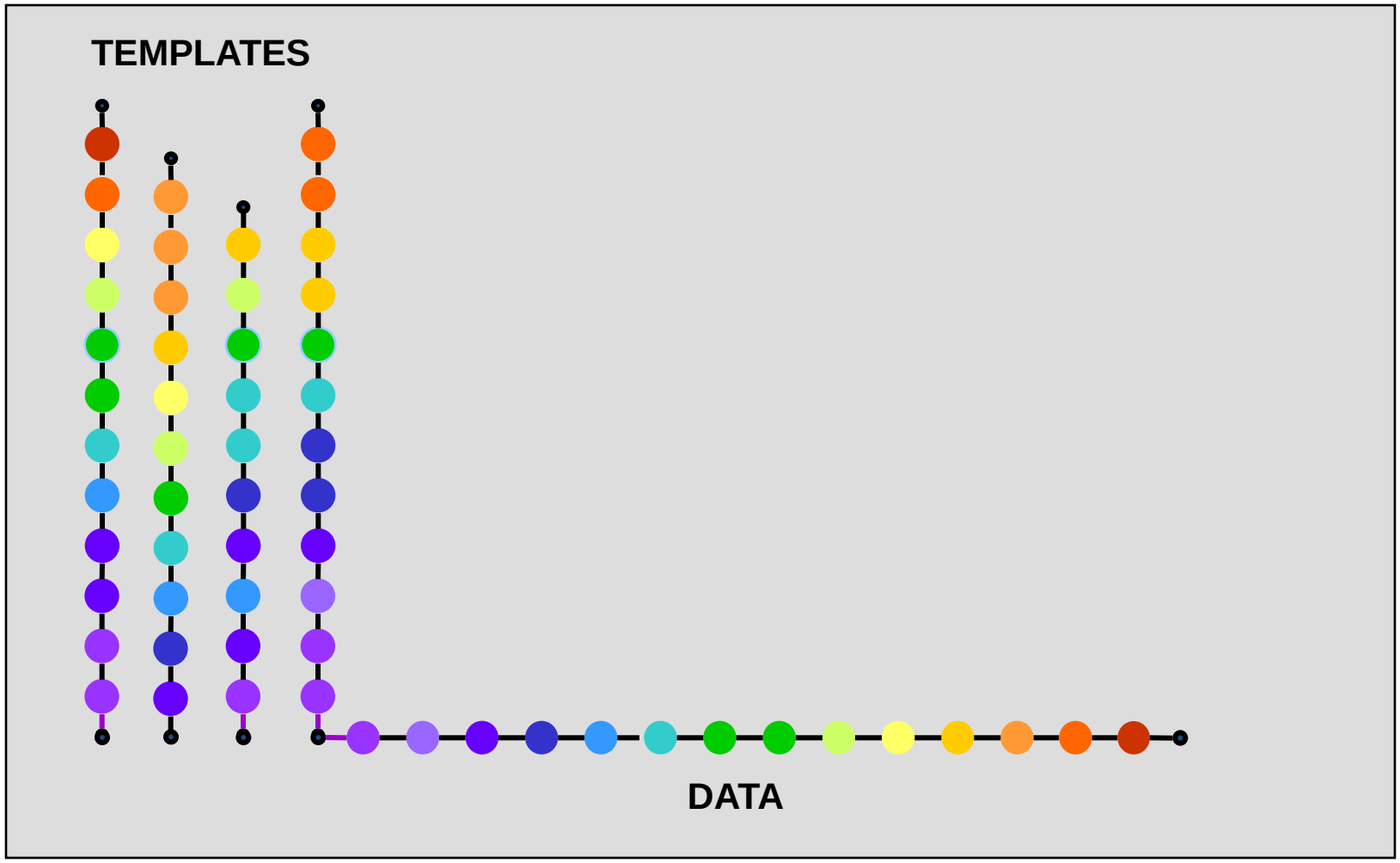
We've seen the DTW alignment of data to model

Limitations of A Single Template

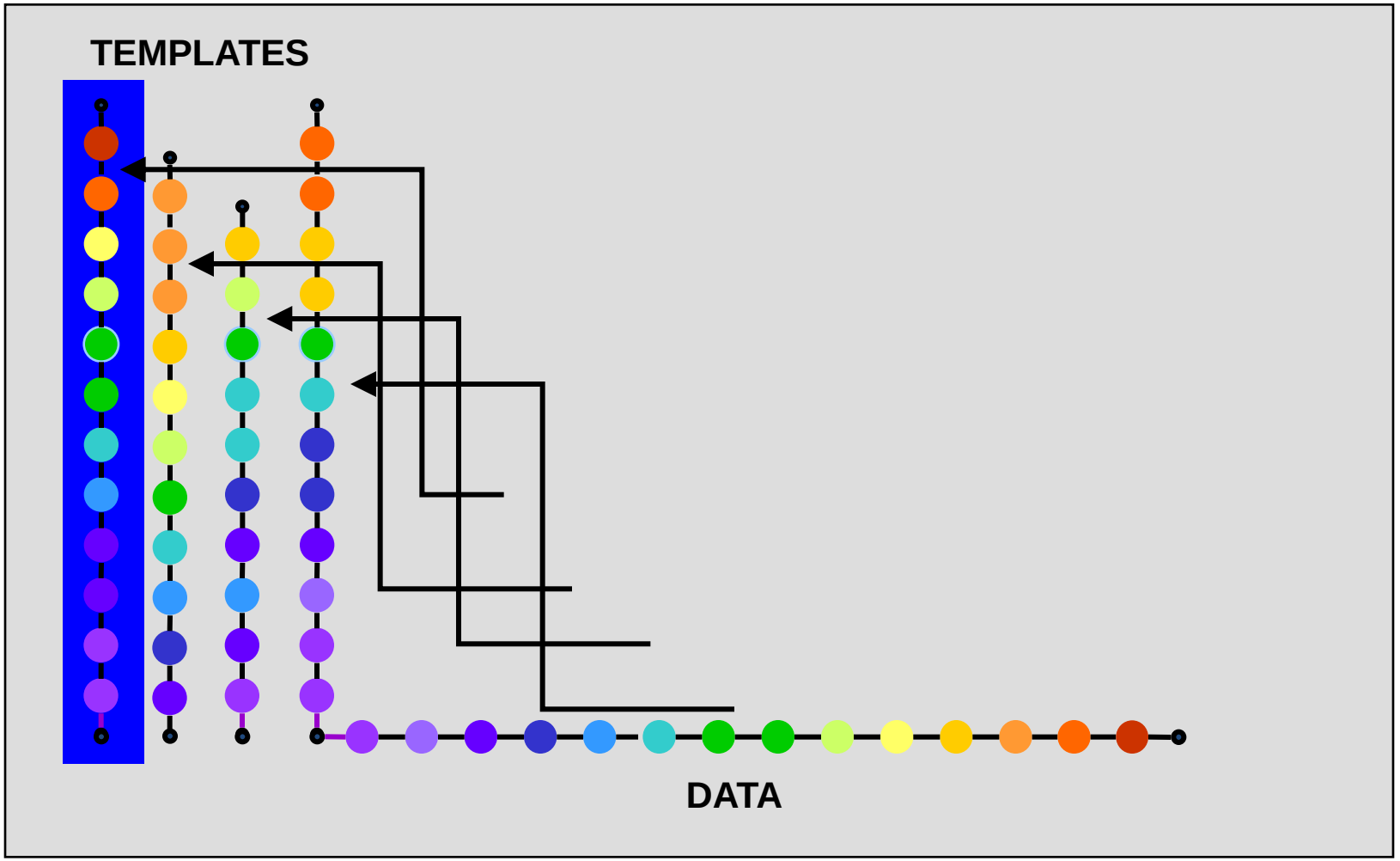
- As noted in the previous topic, a single template cannot capture all the variations in speech

- One alternative already suggested: use multiple templates for each word, and match the input against each one

DTW with multiple templates



DTW with multiple templates



Each template warps differently to best match the input; the best matching template is selected

Problem With Multiple Templates

- Finding the best match requires the evaluation of many more templates (depending on the number)
  - This can be computationally expensive
    - Important for handheld devices, even for small-vocabulary applications
    - Think battery life!
  - Need a method for reducing multiple templates into a single one
- Even multiple templates do not cover the *space* of possible variations
  - Need mechanism of generalizing from the templates to include data not seen before
- We can achieve both objectives by *averaging* all the templates for a given word

Generalizing from Templates

- Generalization implies going from the given templates to one that also represents others that we have not seen
- Taking the average of all available templates may represent the recorded templates less accurately, but will represent other unseen templates more robustly
- A general template (for a word) should capture all salient characteristics of the word, and no more
  - Goal: Improving accuracy
- We will consider several steps to accomplish this

Improving the Templates

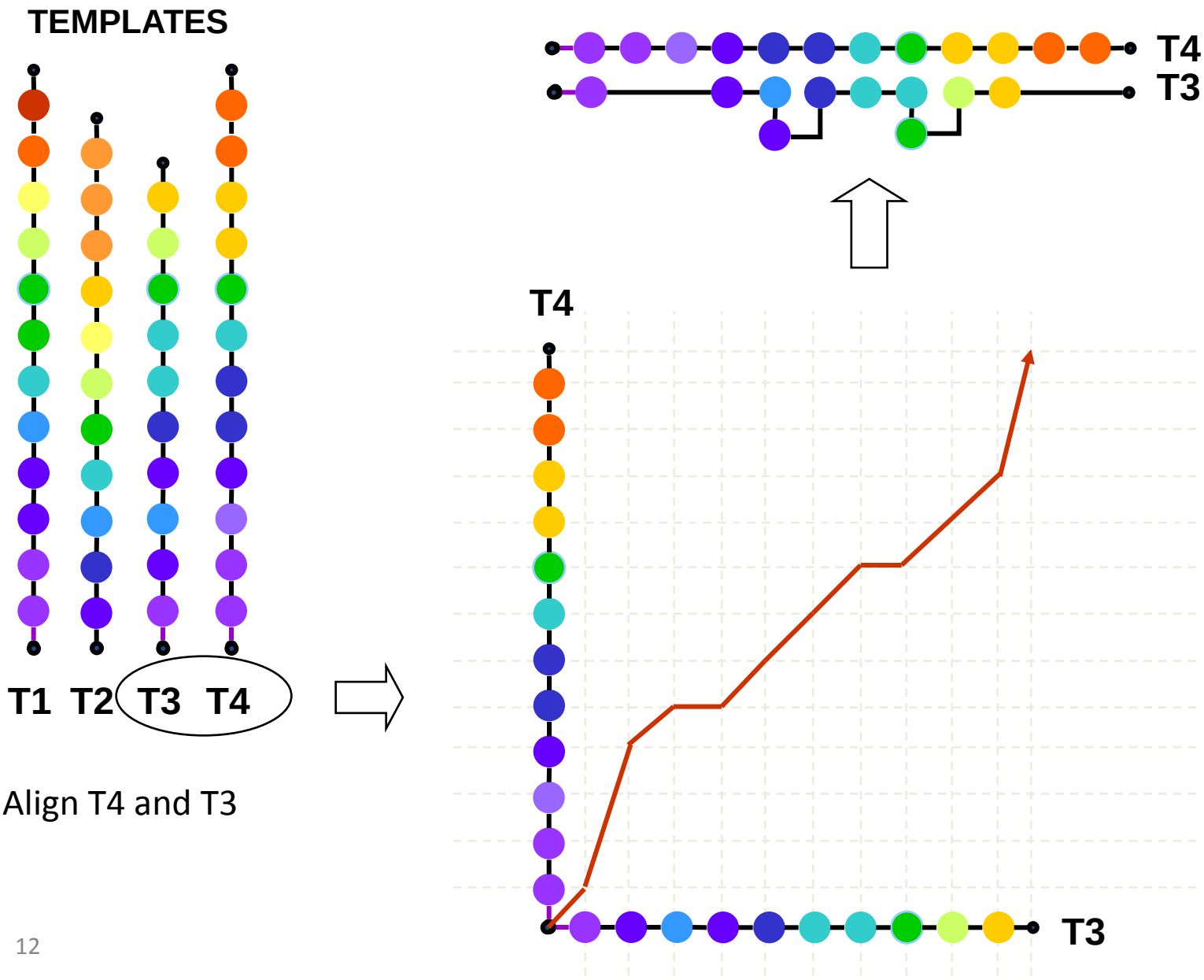
- Generalization by averaging the templates
- Generalization by reducing template length
- Accounting for variation within templates represented by the reduced model
- Accounting for varying segment lengths

Template Averaging

- How can we average the templates when they're of different lengths?
  - Somehow need to normalize them to each other
- *Solution*: Apply DTW (of course!)
  - Pick one template as a "master"
  - Align all other templates to it
    - *Note*: This requires not just finding the best cost, but the actual alignment between the template and input frame sequences, using the *back-pointers* described earlier
  - Use the alignments generated to compute their average

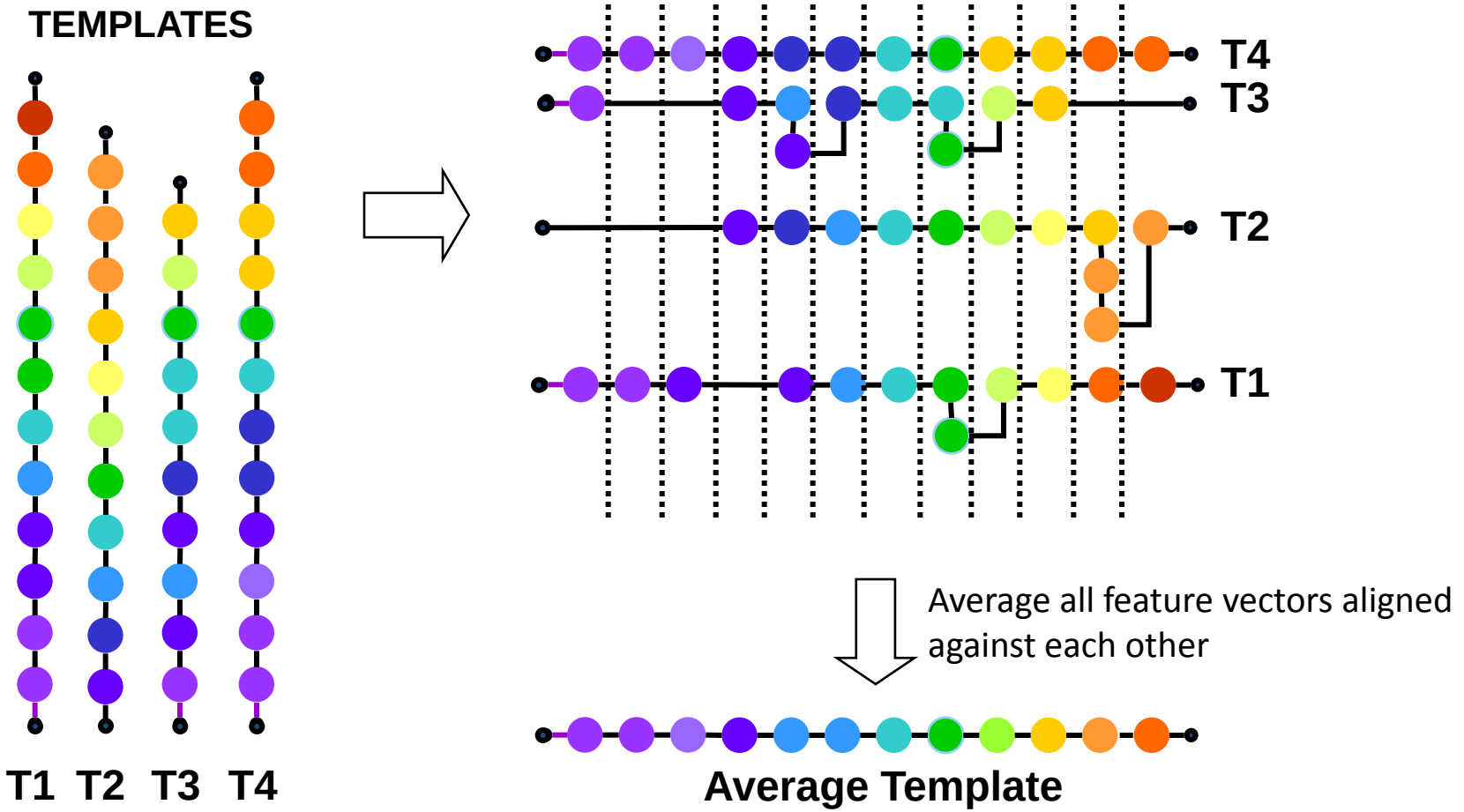
- *Note*: Choosing a different *master* template will lead to a different average template
  - Which template to choose as the master?
    - No definitive answer exists
    - Only trial and error solutions exist

DTW with multiple templates



Align T4 and T3

DTW with multiple templates



Align T4/T2 and T4/T1, similarly; then average all of them

Benefits of Template Averaging

- Obviously, we have eliminated the computational cost of having multiple templates for each word
- Using the averages of the aligned feature vectors *generalizes* from the samples
  - The average is representative of the templates, and more generally, assumed to be representative of future utterances of the word
- The more the number of templates, the better the generalization

Improving the Templates

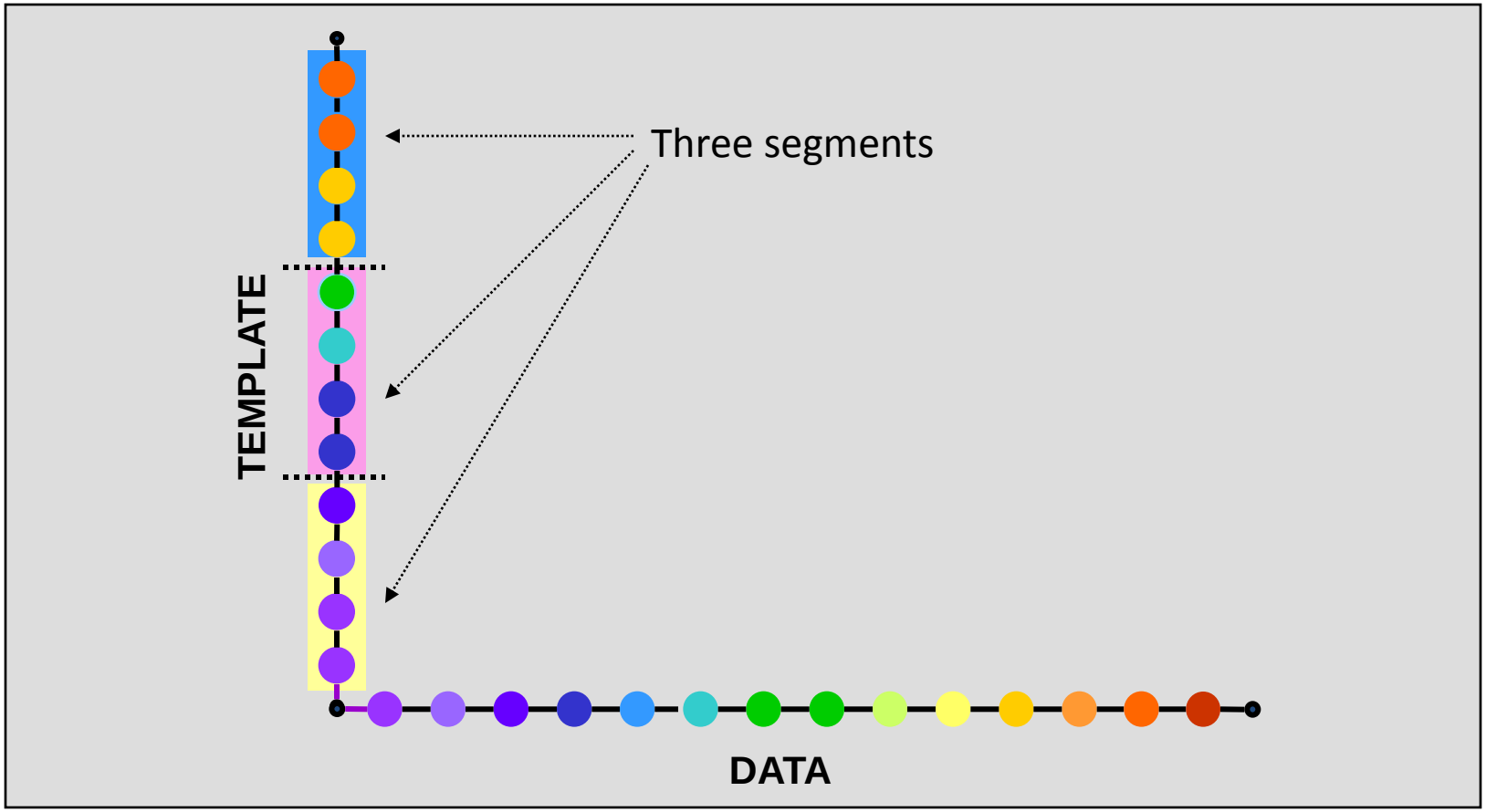
- Generalization by averaging the templates
- Generalization by reducing template length
- Accounting for variation within templates represented by the reduced model
- Accounting for varying segment lengths

Template Size Reduction

- Can we do better? Consider the template for "something":

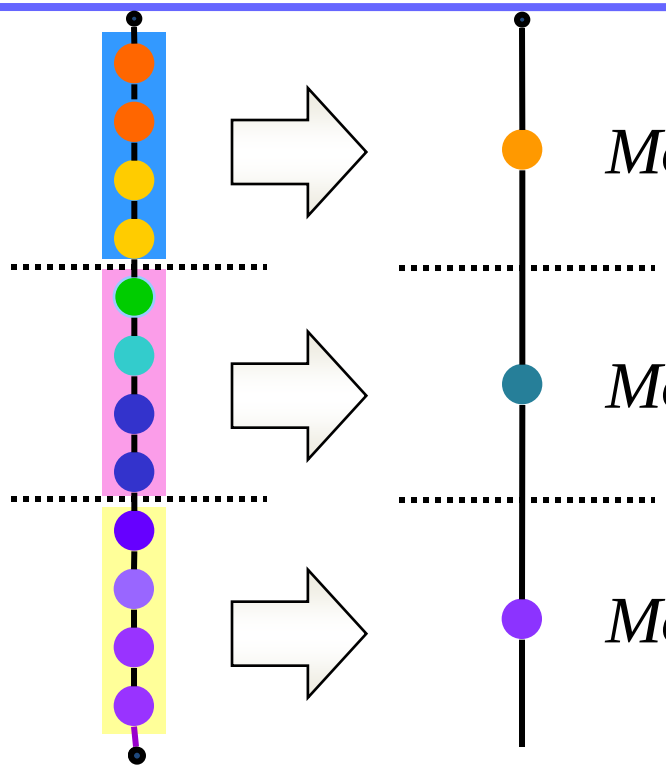
- Here, the template has been manually *segmented* into 6 segments, where each segment is a single phoneme
- Hence, the frames of speech that make up any single segment ought to be fairly alike
- If so, why not replace each segment by a *single* representative feature vector?
  - How? Again by averaging the frames within the segment
- This gives a reduction in the template *size* (memory size)

Example: Single Templates With Three Segments



The feature vectors within each segment are assumed to be similar to each other

Averaging Each Template Segment



$$Model\ Vector = \frac{1}{N} \sum_{i \in segment} vector(i)$$

$$Model\ Vector = \frac{1}{N} \sum_{i \in segment} vector(i)$$

$$Model\ Vector = \frac{1}{N} \sum_{i \in segment} vector(i)$$

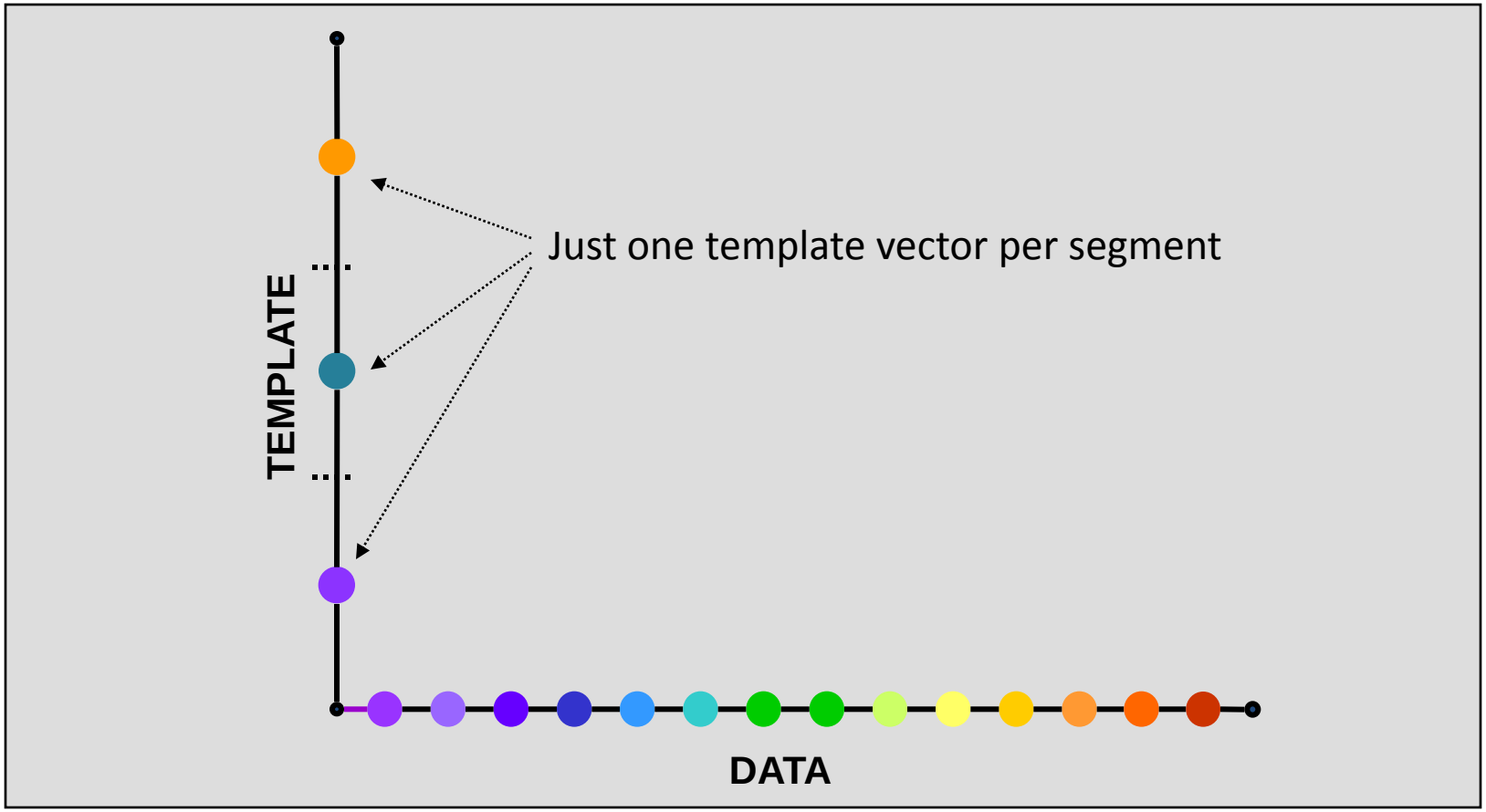
$$m_j = \frac{1}{N_j} \sum_{i \in segment(j)} x(i)$$

m_j is the model vector for the j^{th} segment

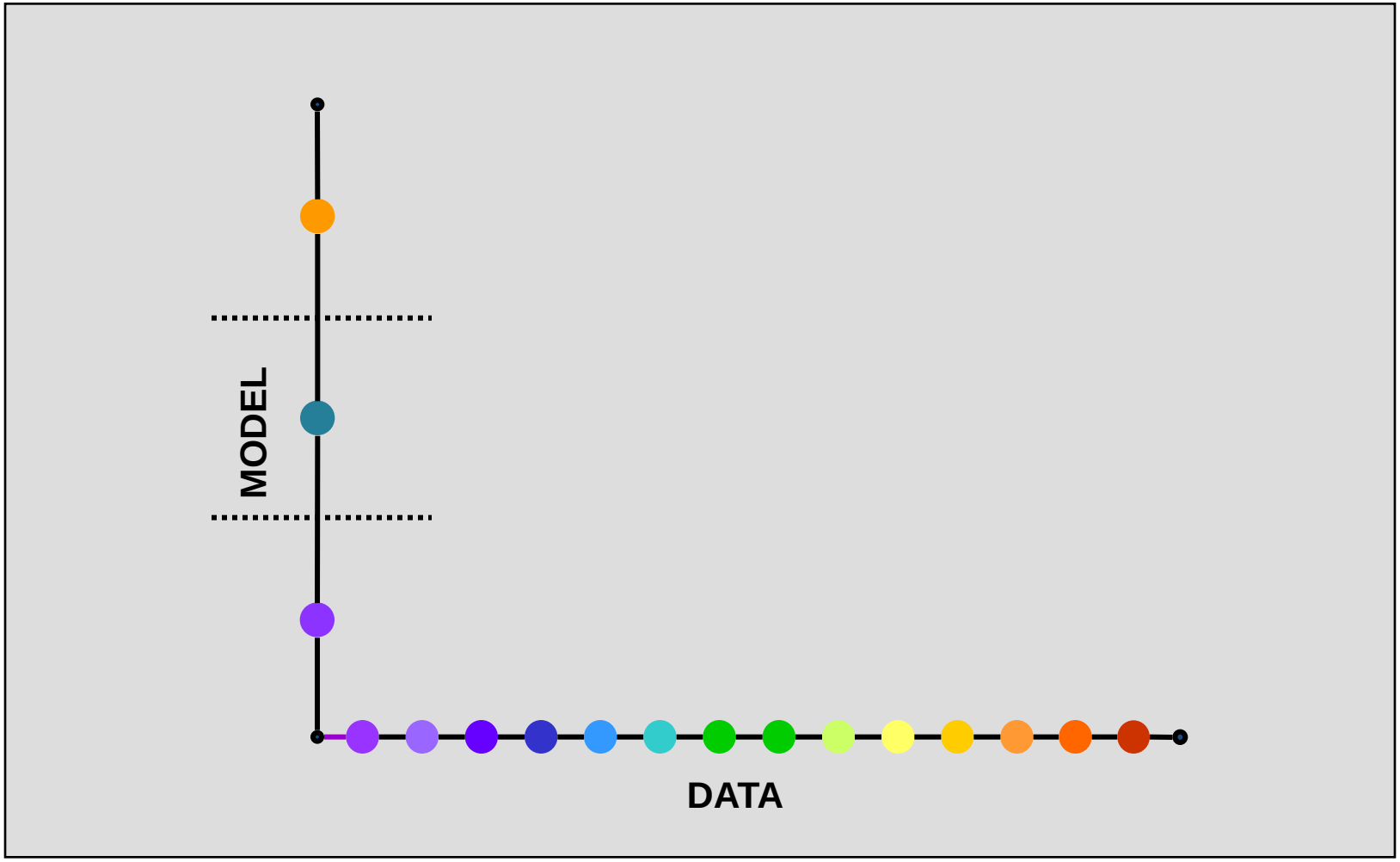
N_j is the number of vectors in the j^{th} segment

$x(i)$ is the i^{th} feature vector

Template With One Model Vector Per Segment

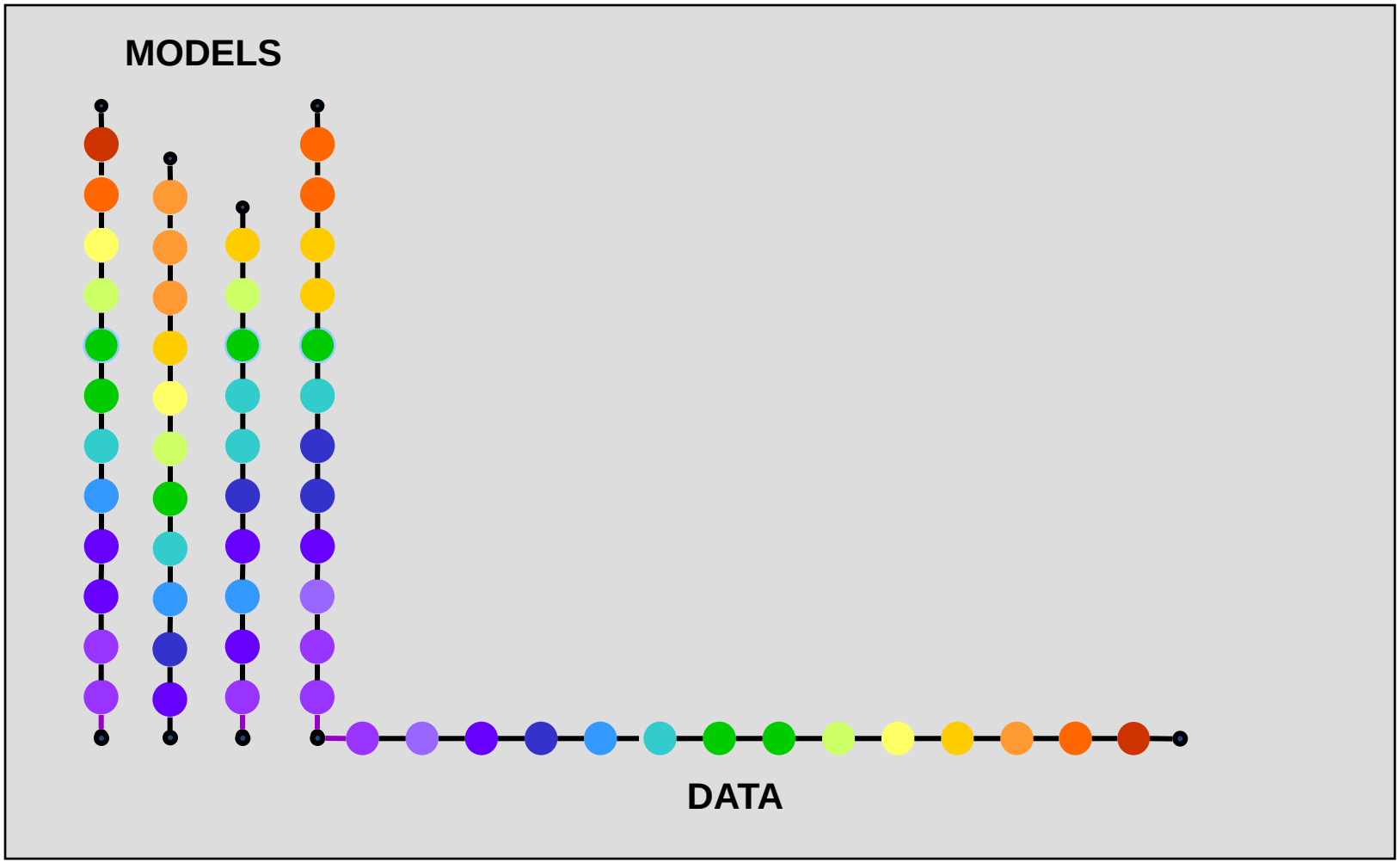


DTW with one model



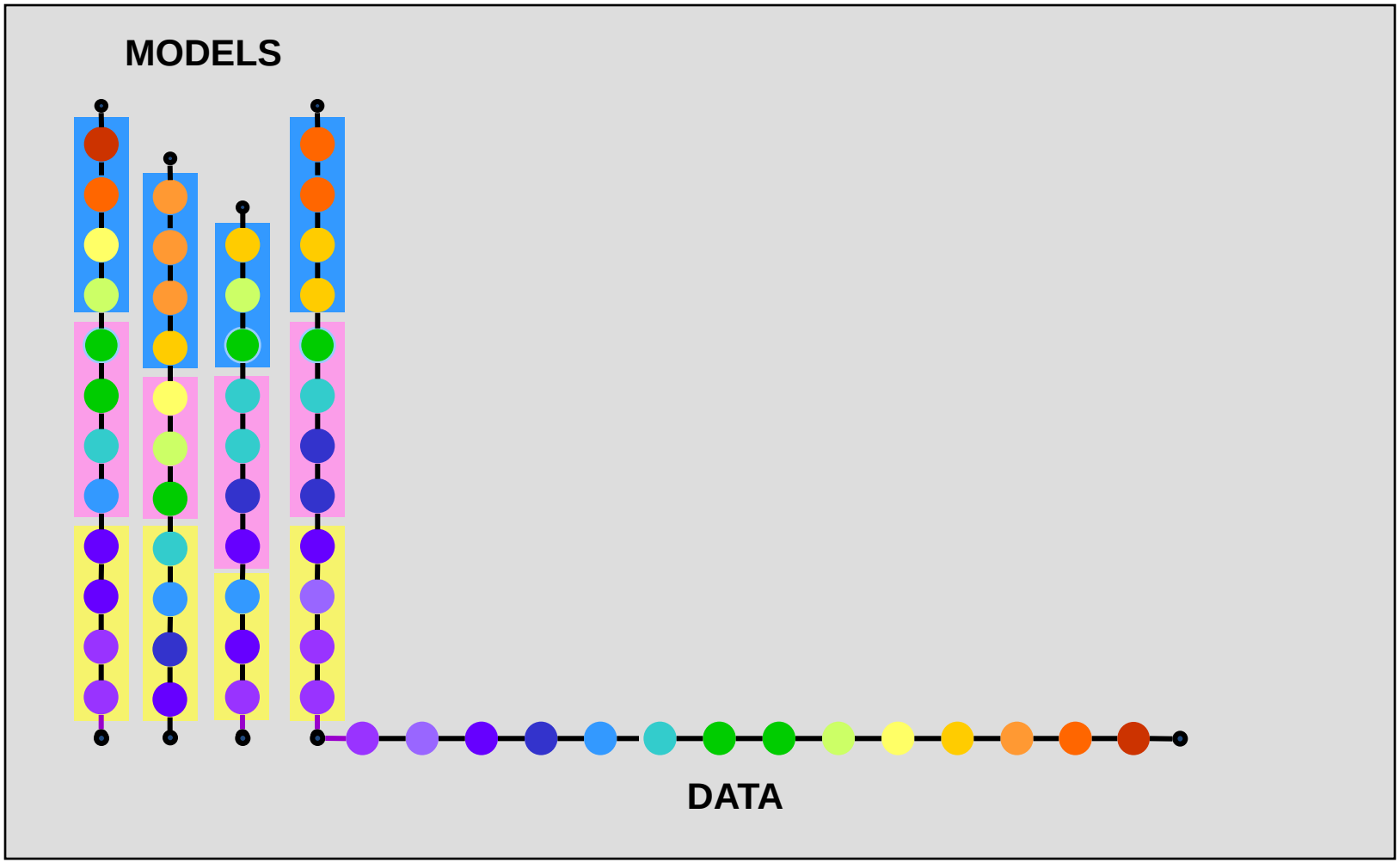
The averaged template is matched against the data string to be recognized
Select the word whose averaed template has the lowest cost of match

DTW with multiple models



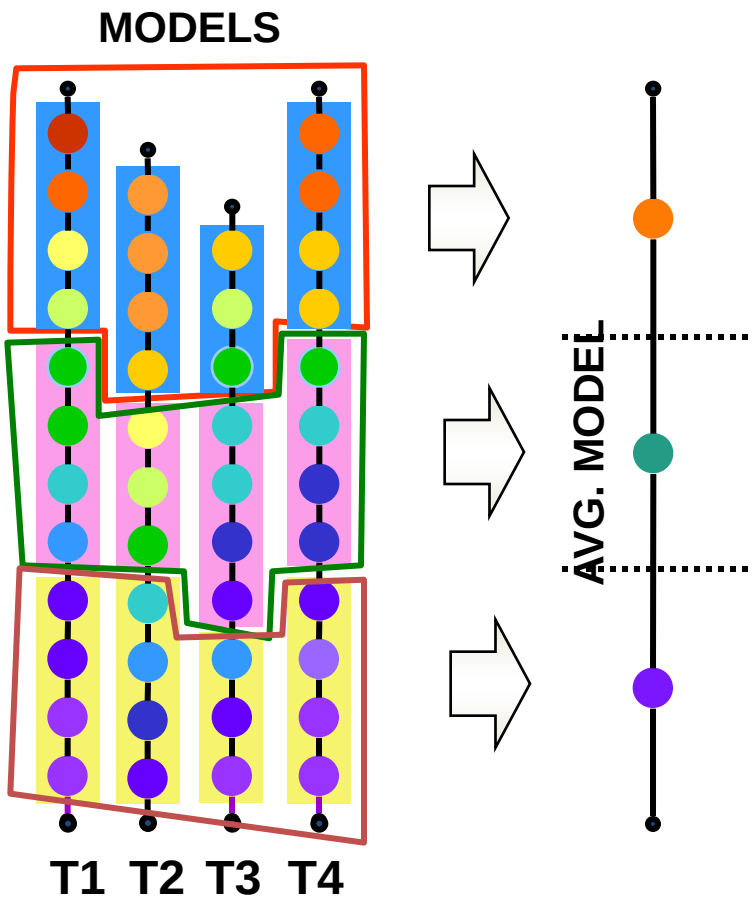
Segment all templates
Average each region into a single point

DTW with multiple models



Segment all templates
Average each region into a single point

DTW with multiple models



$$m_j = \frac{1}{\sum_k N_{k,j}} \sum_{i \in segment_k(j)} x_k(i)$$

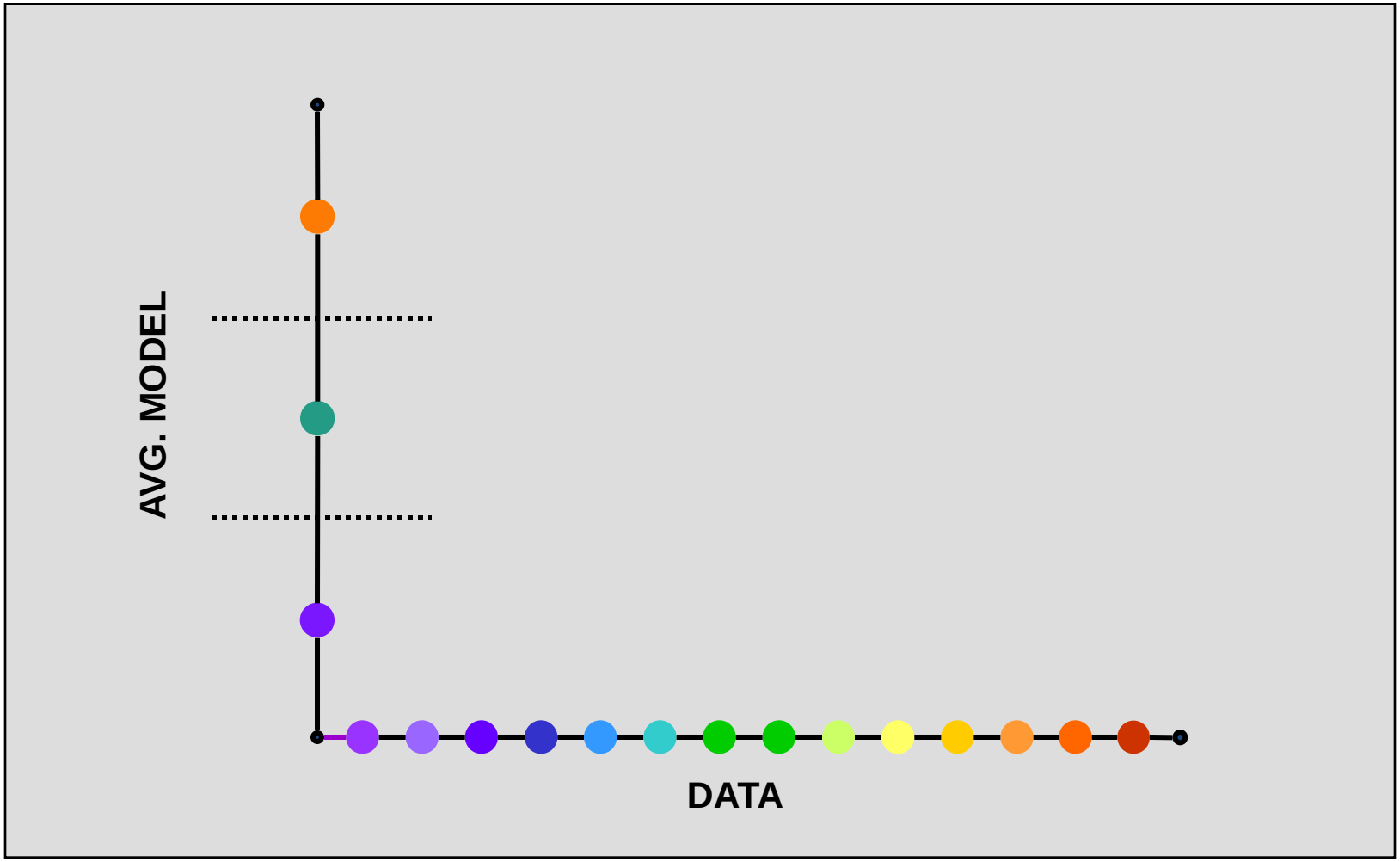
$segment_k(j)$ is the j^{th} segment of the k^{th} training sequence

m_j is the model vector for the j^{th} segment

$N_{k,j}$ is the number of training vectors in the j^{th} segment of the k^{th} training sequence

$x_k(i)$ is the i^{th} vector of the k^{th} training sequence

DTW with multiple models

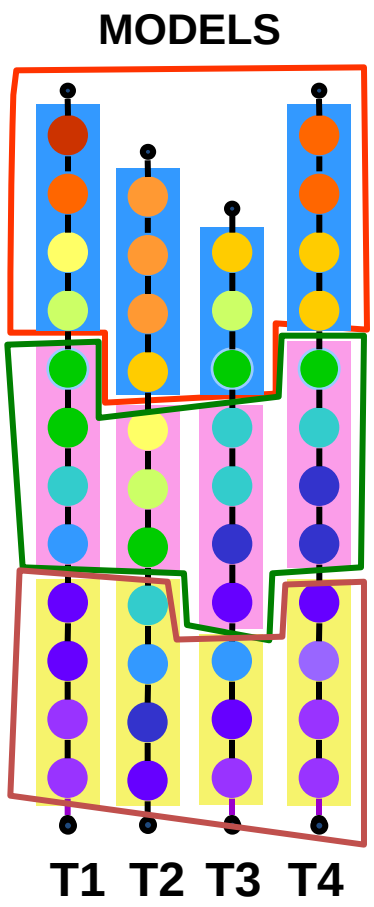


Segment all templates, average each region into a single point
To get a simple average model, which is used for recognition

Improving the Templates

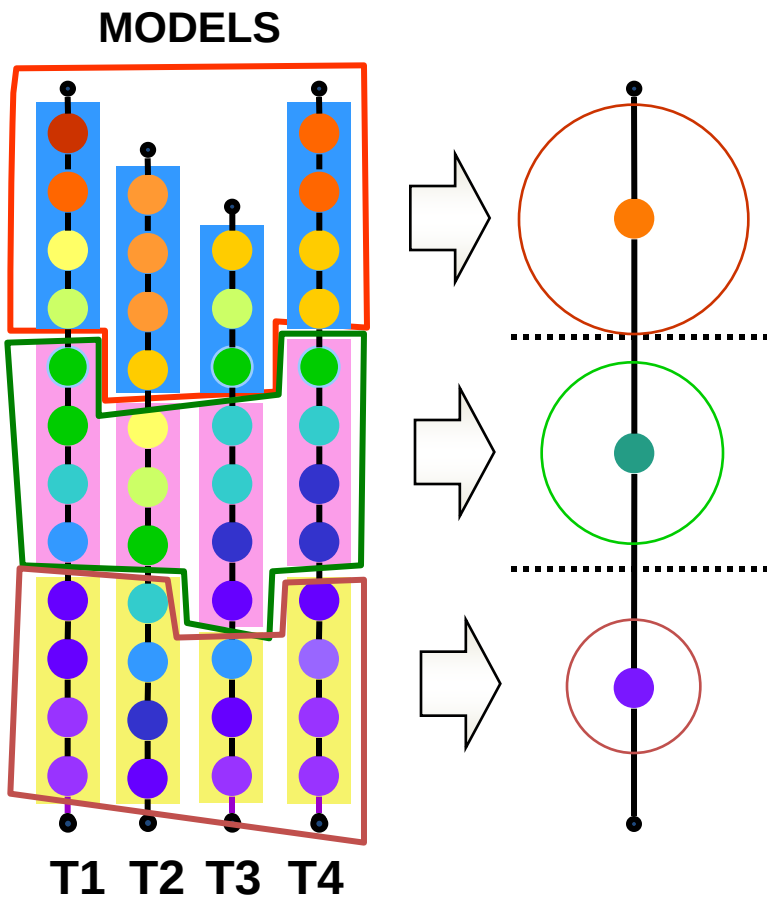
- Generalization by averaging the templates
- Generalization by reducing template length
- Accounting for variation within templates represented by the reduced model
- Accounting for varying segment lengths

DTW with multiple models



- The inherent variation between vectors is different for the different segments
  - E.g. the variation in the colors of the beads in the top segment is greater than that in the bottom segment
- Ideally we should account for the differences in variation in the segments
  - E.g, a vector in a test sequence may actually be more matched to the central segment, which permits greater variation, although it is closer, in a Euclidean sense, to the mean of the lower segment, which permits lesser variation

DTW with multiple models



We can define the covariance for each segment using the standard formula for covariance

$$C_j = \frac{1}{\sum_k N_{k,j}} \sum_{i \in segment_k(j)} \left(x_k(i) - m_j\right)\left(x_k(i) - m_j\right)^T$$

m_j is the model vector for the j^{th} segment

C_j is the covariance of the vectors in the j^{th} segment

DTW with multiple models

- The distance function must be modified to account for the covariance
- Mahalanobis distance:
  - Normalizes contribution of all dimensions of the data

$$d(x, m_j) = (x - m_j)^T C_j^{-1} (x - m_j)$$

  - x is a data vector, m_j is the mean of a segment, C_j is the covariance matrix for the segment
- Negative Gaussian log likelihood:
  - Assumes a Gaussian distribution for the segment and computes the probability of the vector on this distribution

$$Gaussian(x; m_j, C_j) = \frac{1}{\sqrt{(2\pi)^D |C_j|}} e^{-0.5(x-m_j)^T C_j^{-1} (x-m_j)}$$

$$d(x, m_j) = -\log\left(Gaussian(x; m_j, C_j)\right) = 0.5\log\left((2\pi)^D |C_j|\right) + 0.5(x - m_j)^T C_j^{-1} (x - m_j)$$

The Covariance

- The variance that we have computed is a *full covariance matrix*
  - And the distance measure requires a matrix inversion

$$C_j = \frac{1}{\sum_k N_k} \sum_k \sum_{i \in segment_k(j)} \left(x_k(i) - m_j\right)\left(x_k(i) - m_j\right)^T$$

$$d(x, m_j) = (x - m_j)^T C_j^{-1} (x - m_j)$$

- In practice we assume that all off-diagonal terms in the matrix are 0
- This reduces our distance metric to:

$$d(x, m_j) = \sum_l \frac{(x_l - m_{j,l})^2}{\sigma_{j,l}^2}$$

- Where the individual variance terms σ^2 are

$$\sigma_{j,l}^2 = \frac{1}{\sum_k N_k} \sum_k \sum_{i \in segment_k(j)} \left(x_{k,l}(i) - m_{j,l}\right)^2$$

- If we use a negative log Gaussian instead, the modified score (with the diagonal covariance) is

$$d(x, m_j) = 0.5 \sum_l \log(2\pi\sigma_{j,l}^2) + 0.5 \sum_l \frac{(x_l - m_{j,l})^2}{\sigma_{j,l}^2}$$

Segmental K-means

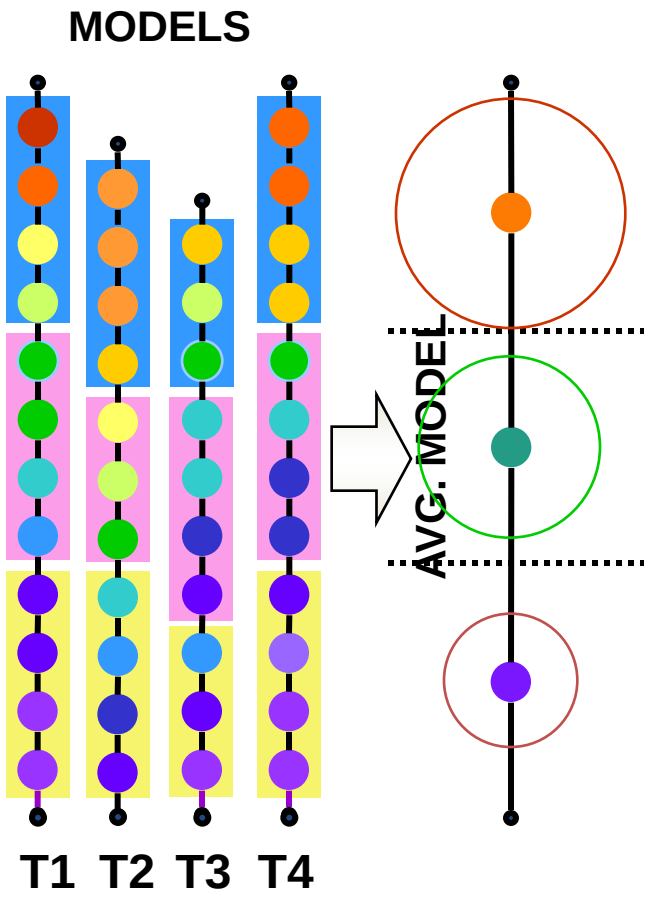
- Simple uniform segmentation of training instances is not the most effective method of grouping vectors in the training sequences

- A better segmentation strategy is to segment the training sequences such that the vectors within any segment are most alike
  - The total distance of vectors within each segment from the model vector for that segment is minimum
  - For a global optimum, the total distance of all vectors from the model for their respective segments must be minimum

- This segmentation must be estimated

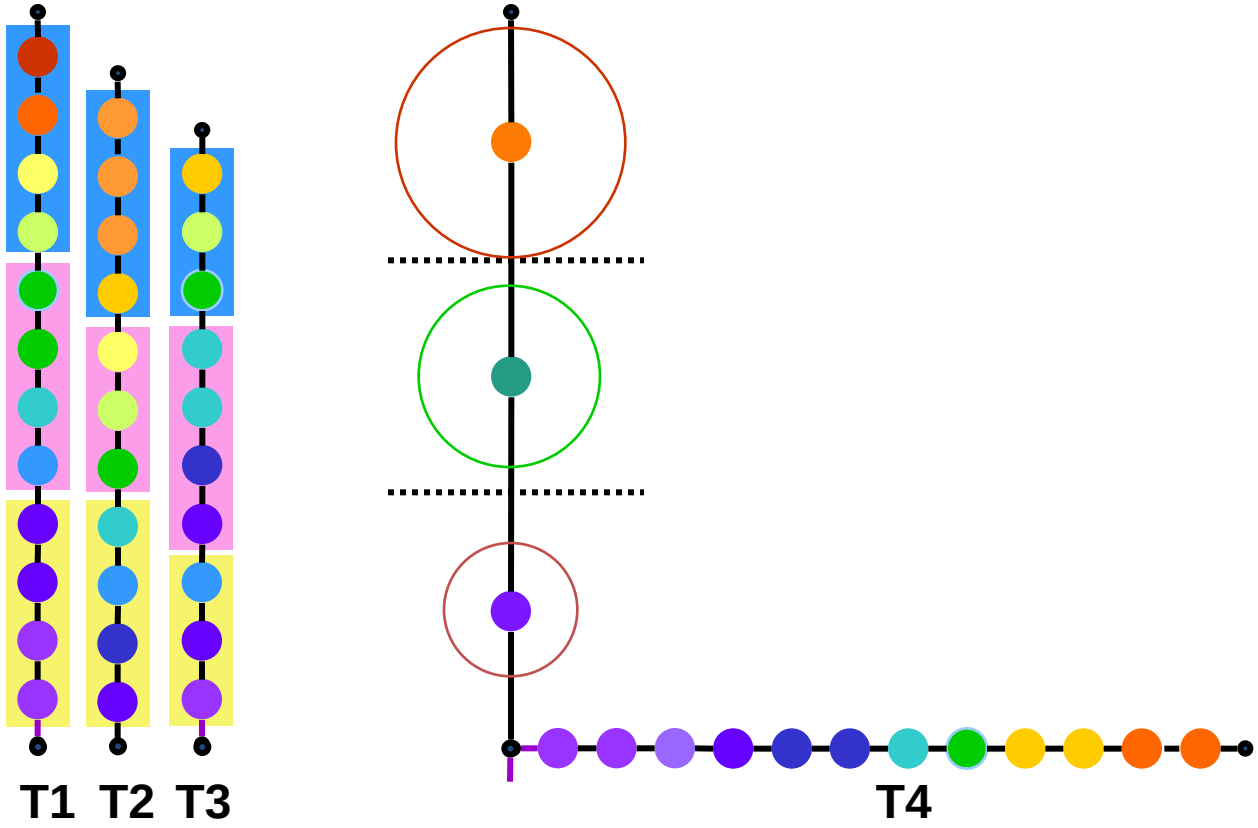
- The segmental K-means procedure is an iterative procedure to estimate the optimal segmentation

Alignment for training a model from multiple vector sequences



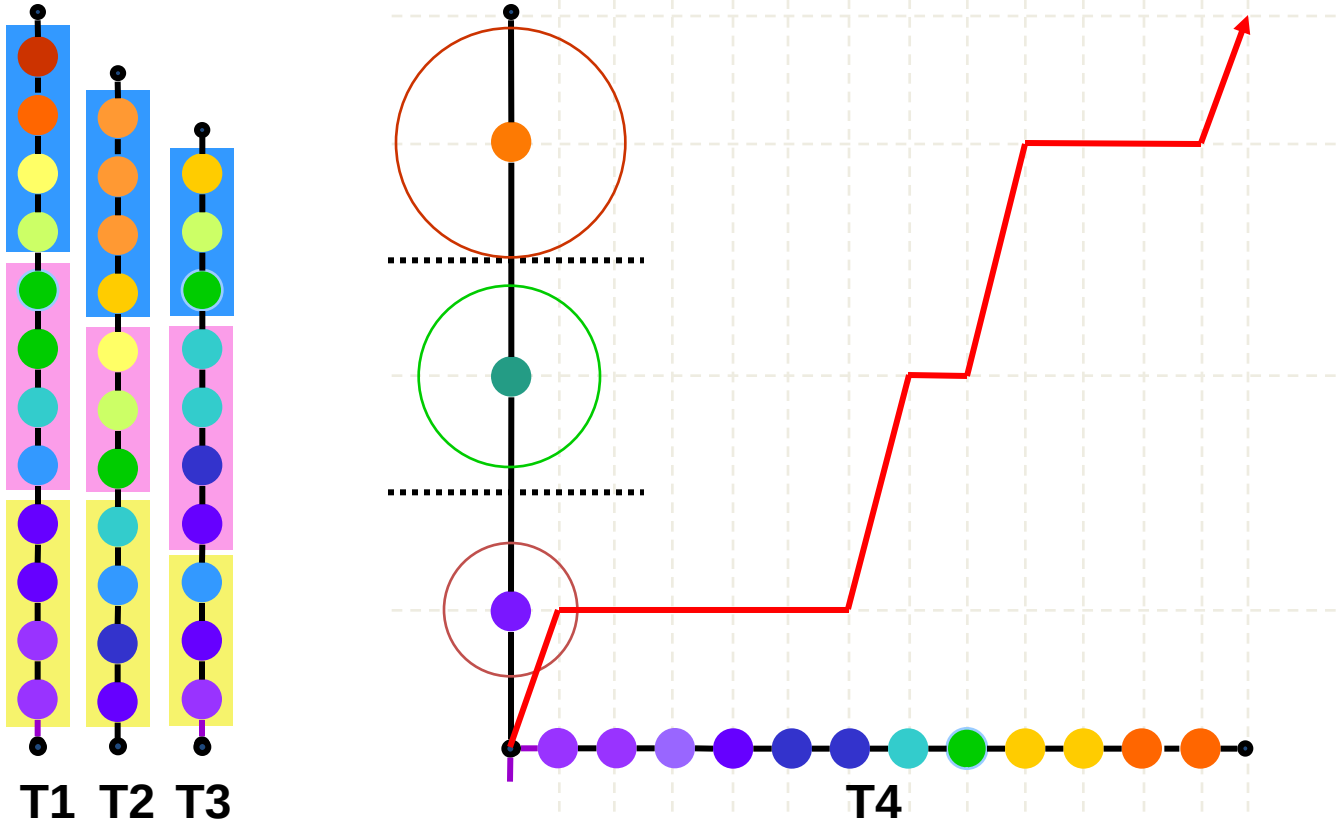
Initialize by uniform segmentation

Alignment for training a model from multiple vector sequences



Initialize by uniform segmentation

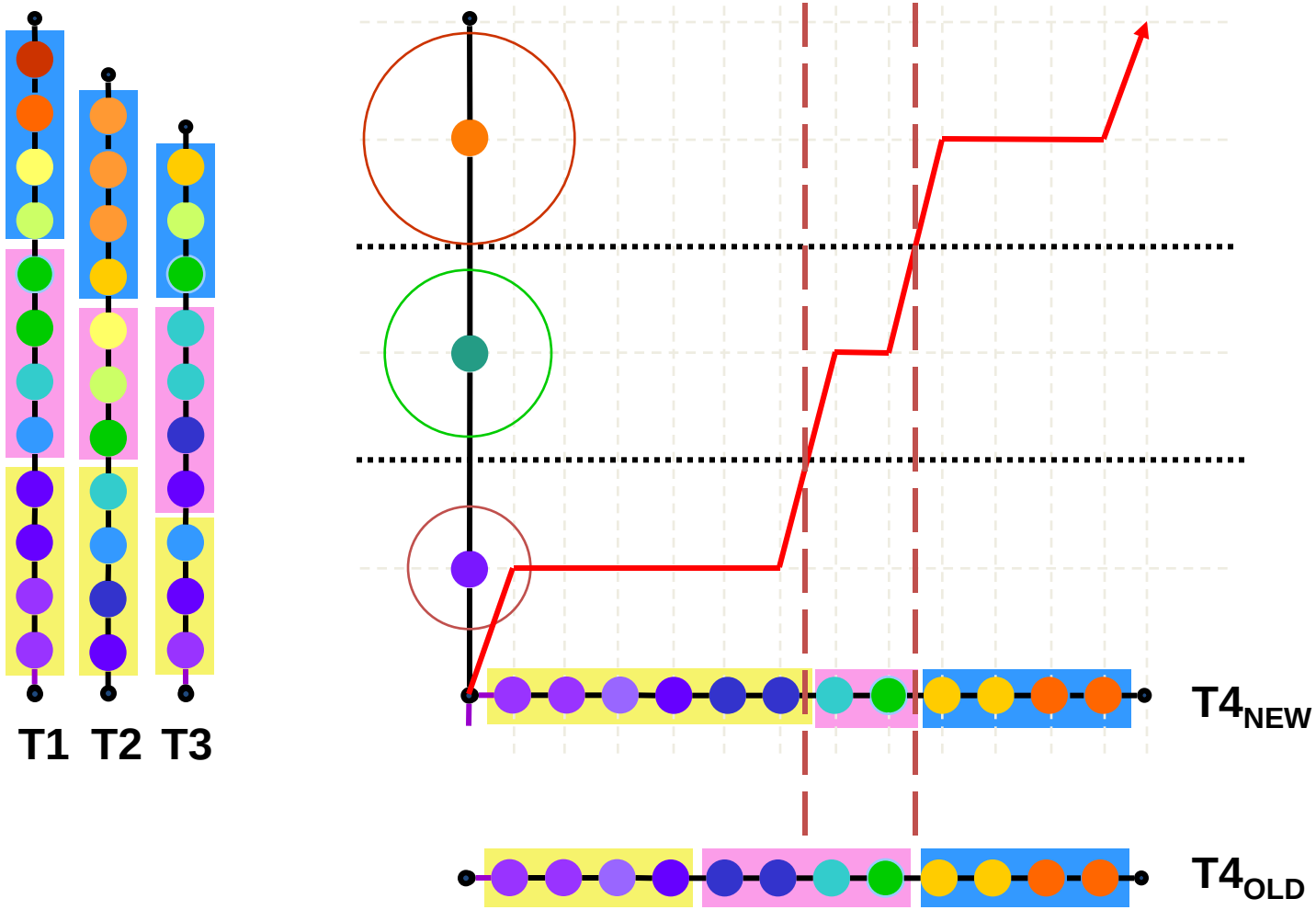
Alignment for training a model from multiple vector sequences



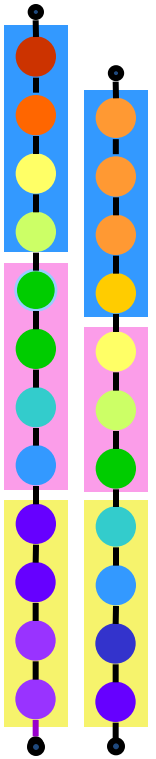
Initialize by uniform segmentation

Align each template to the averaged model to get new segmentations

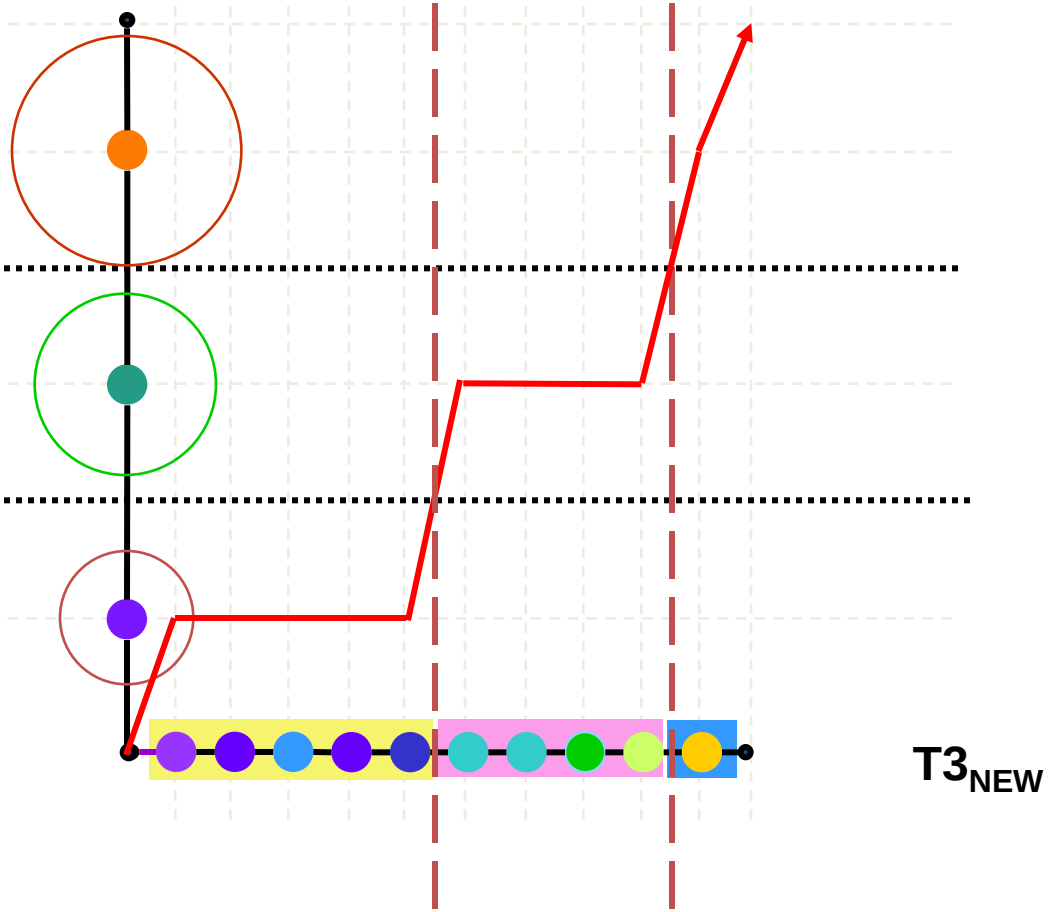
Alignment for training a model from multiple vector sequences



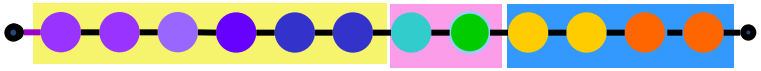
Alignment for training a model from multiple vector sequences



T1 T2

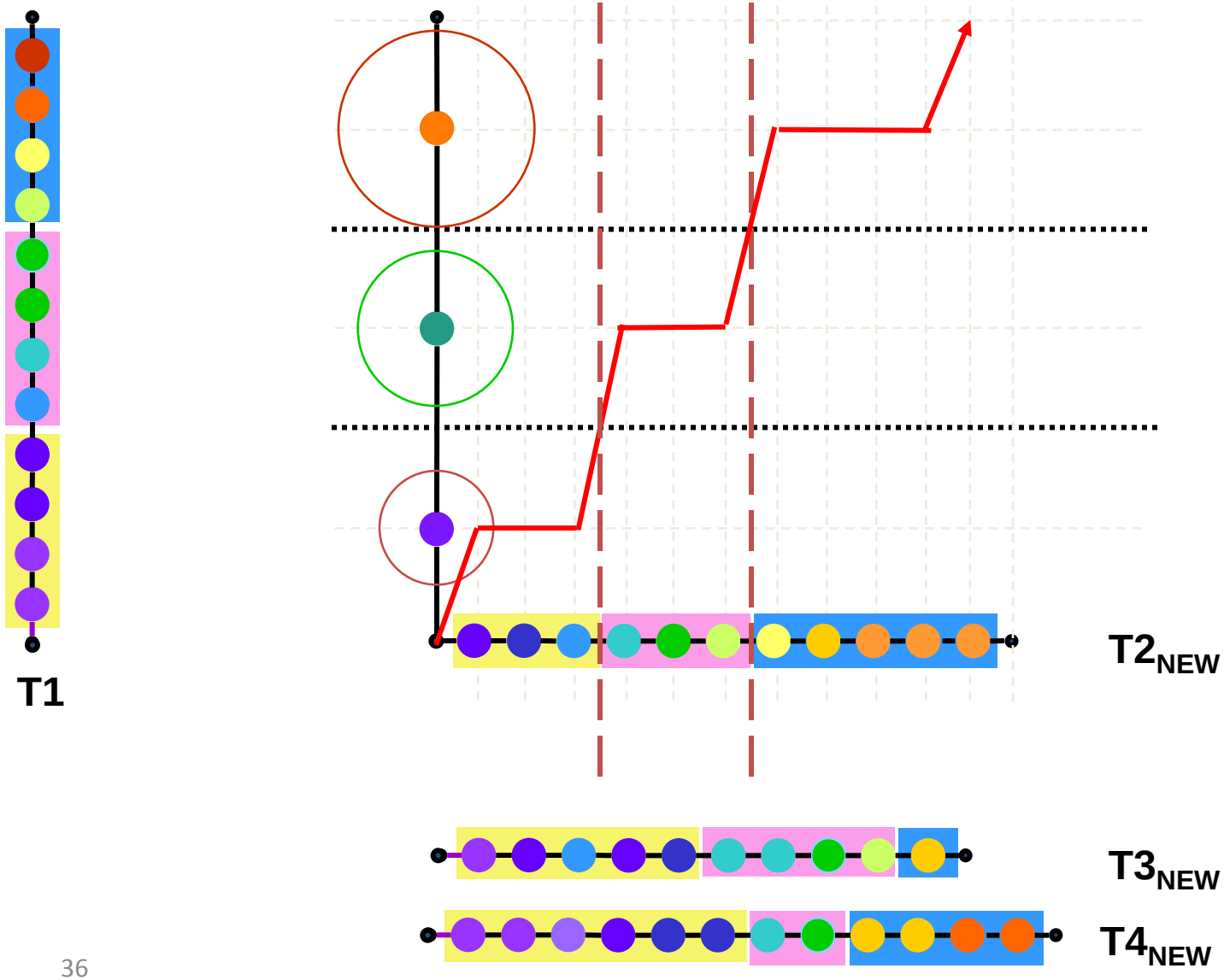


$T3_{NEW}$

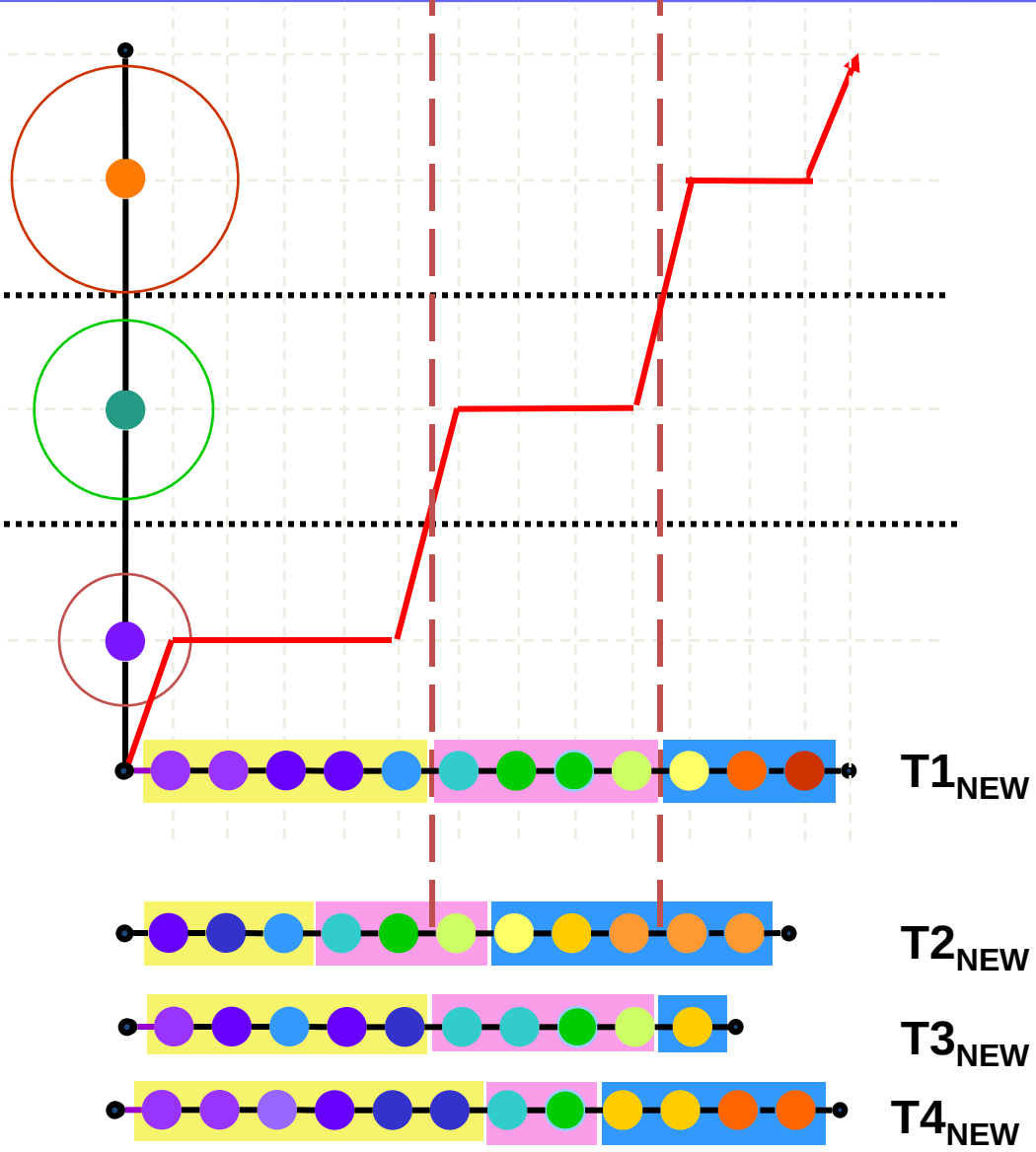


$T4_{NEW}$

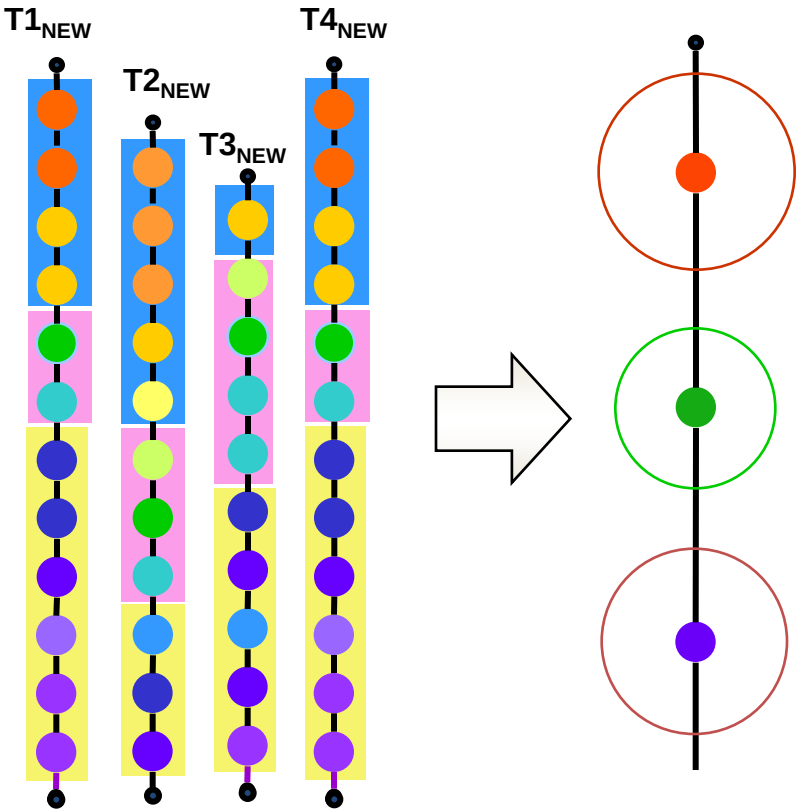
Alignment for training a model from multiple vector sequences



Alignment for training a model from multiple vector sequences

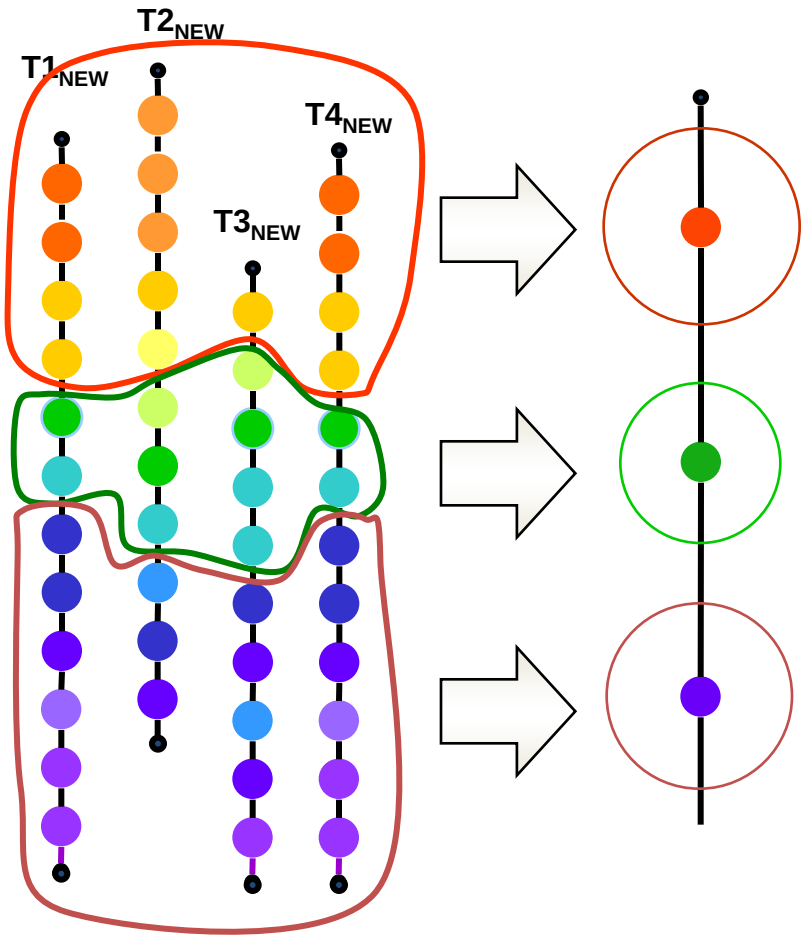


Alignment for training a model from multiple vector sequences

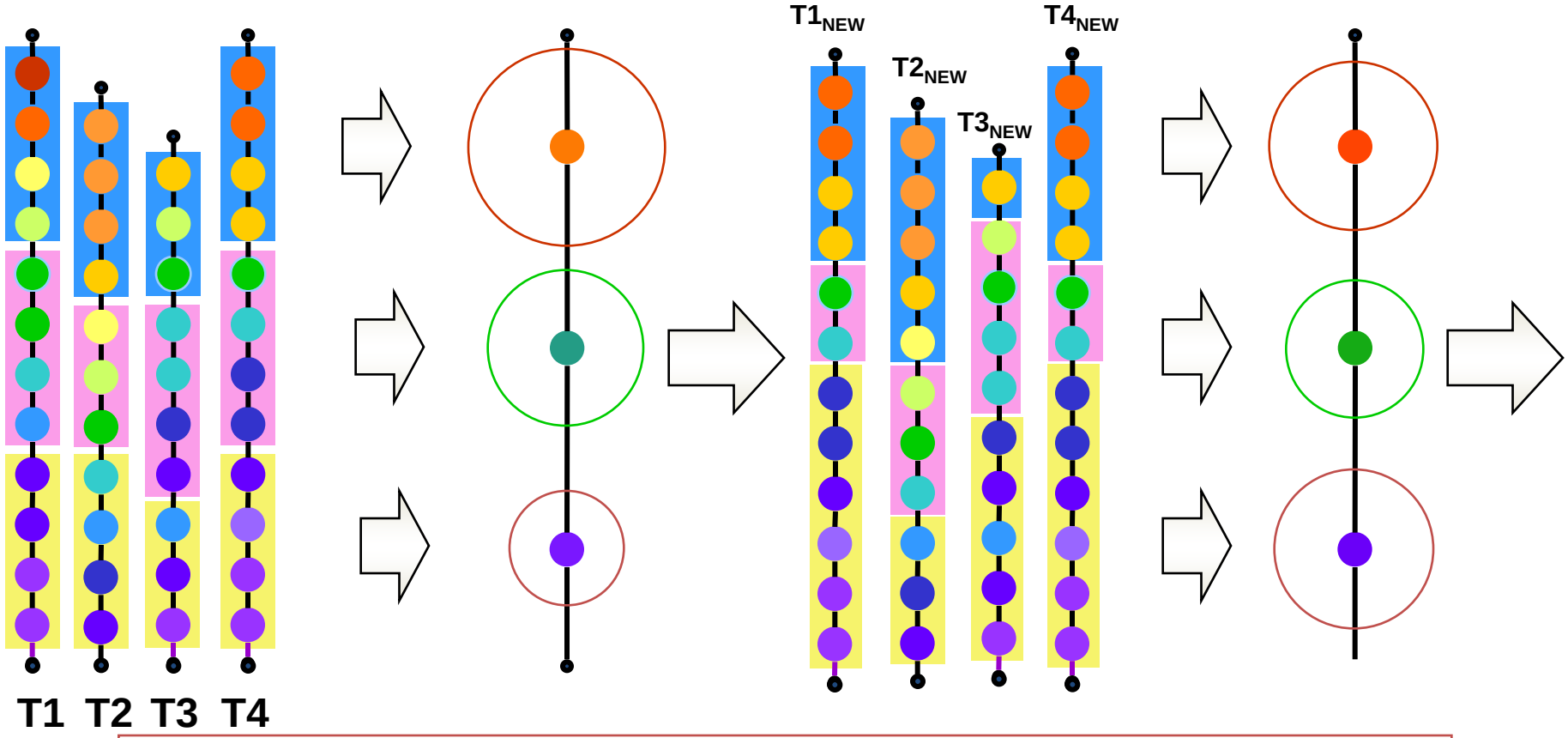


Initialize by uniform segmentation
Align each template to the averaged model to get new segmentations
Recompute the average model from new segmentations

Alignment for training a model from multiple vector sequences



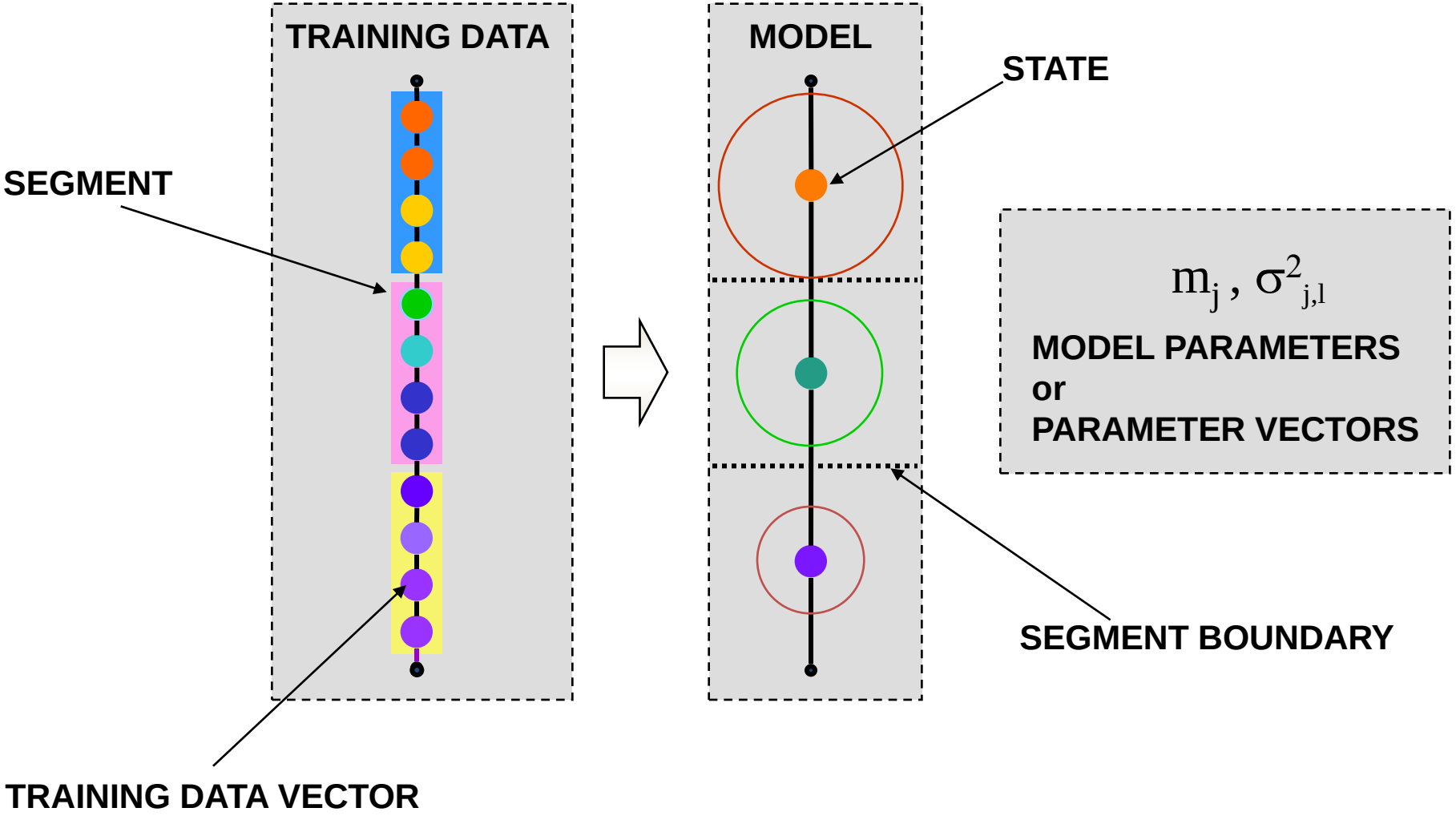
Alignment for training a model from multiple vector sequences



The procedure can be continued until convergence

Convergence is achieved when the total best-alignment error for all training sequences does not change significantly with further refinement of the model

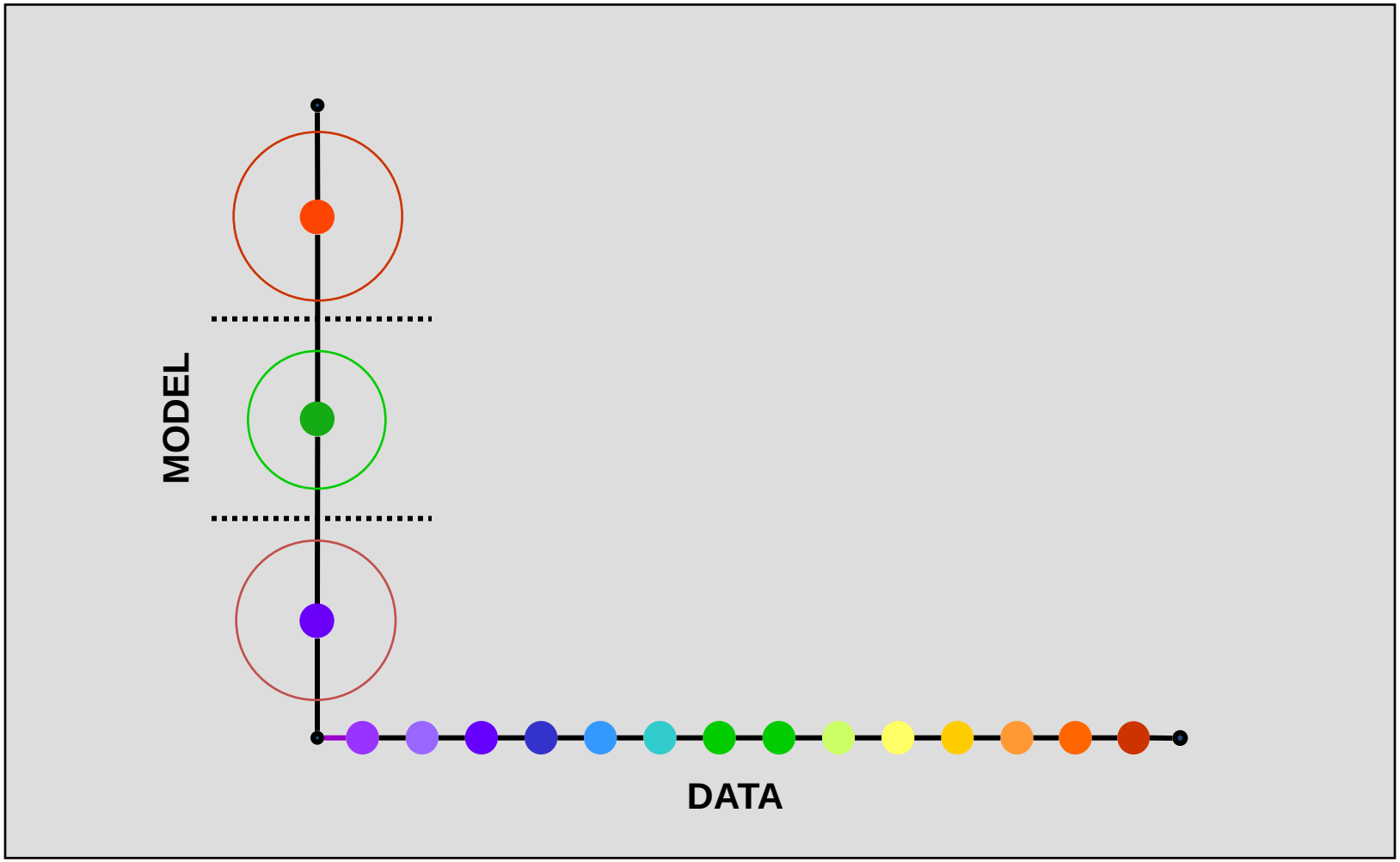
Shifted terminology



Improving the Templates

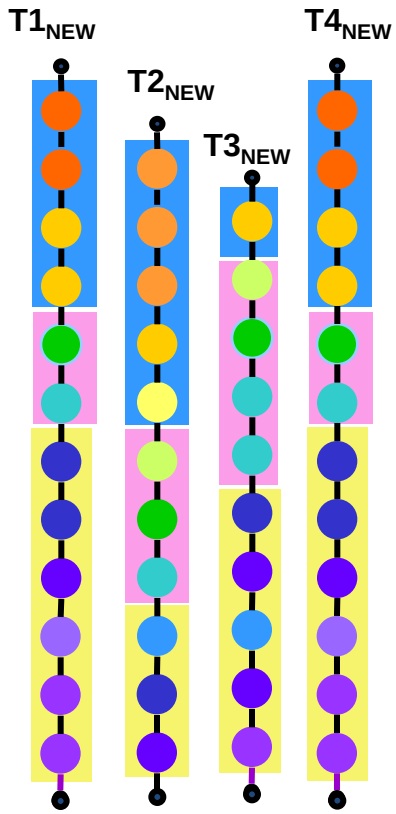
- Generalization by averaging the templates
- Generalization by reducing template length
- Accounting for variation within templates represented by the reduced model
- Accounting for varying segment lengths

Transition structures in models



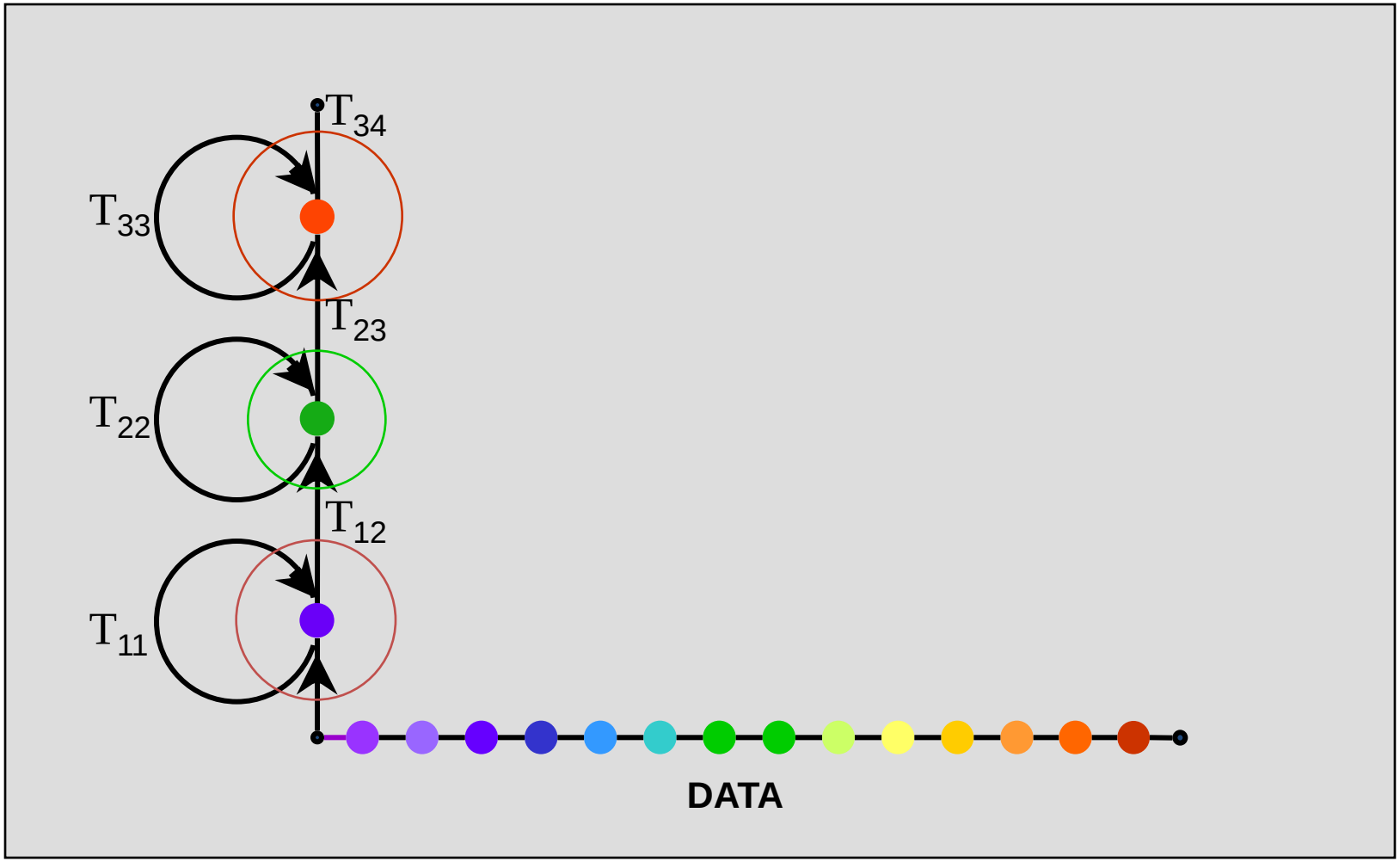
The converged models can be used to score / align data sequences
Model structure in incomplete.

DTW with multiple models



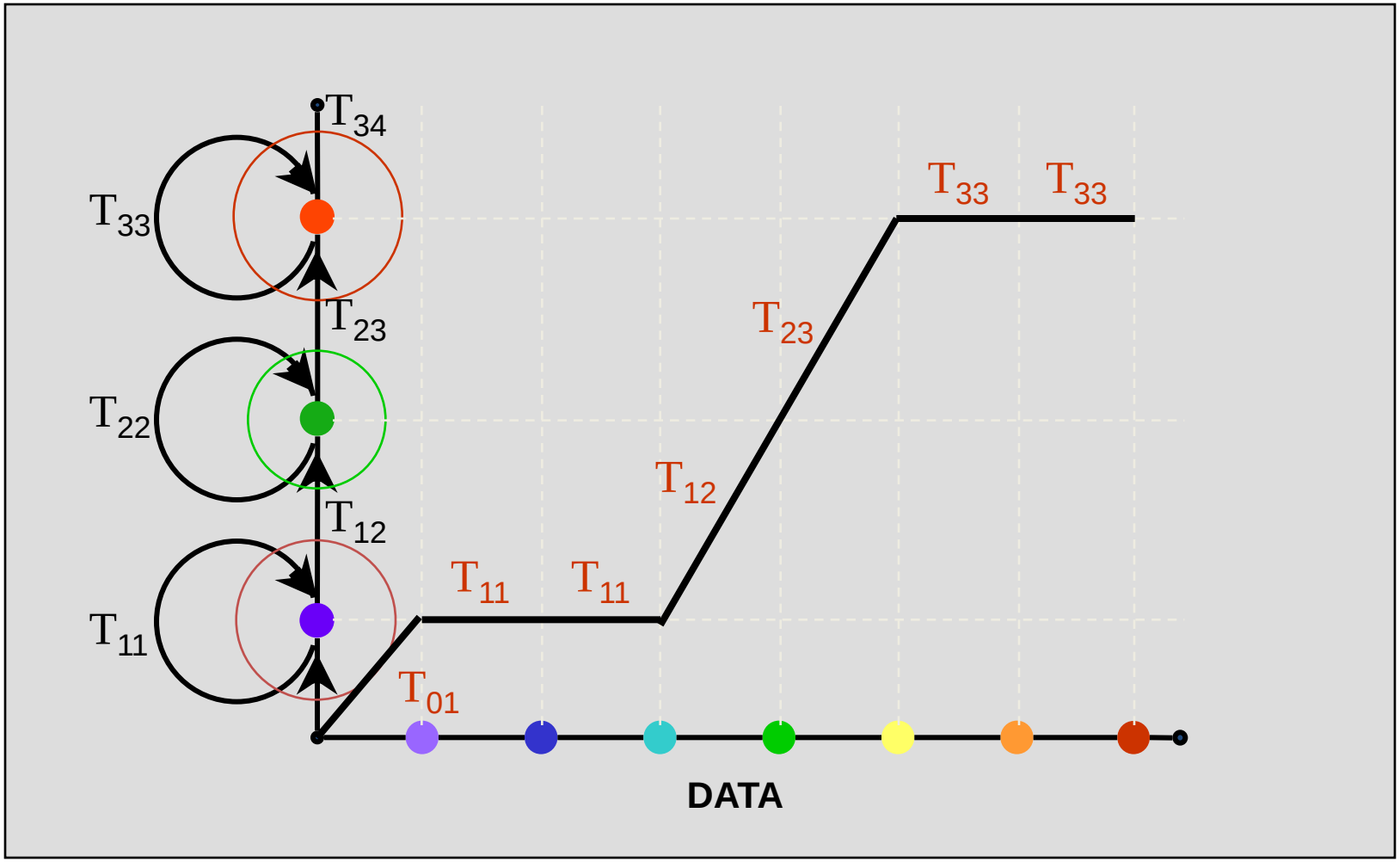
- Some segments are naturally longer than others
  - E.g., in the example the initial (yellow) segments are usually longer than the second (pink) segments
- This difference in segment lengths is different from the *variation* within a segment
  - Segments with small variance could still persist very long for a particular sound or word
- The DTW algorithm must account for these natural differences in typical segment length
- This can be done by having a state specific insertion penalty
  - States that have lower insertion penalties persist longer and result in longer segments

Transition structures in models



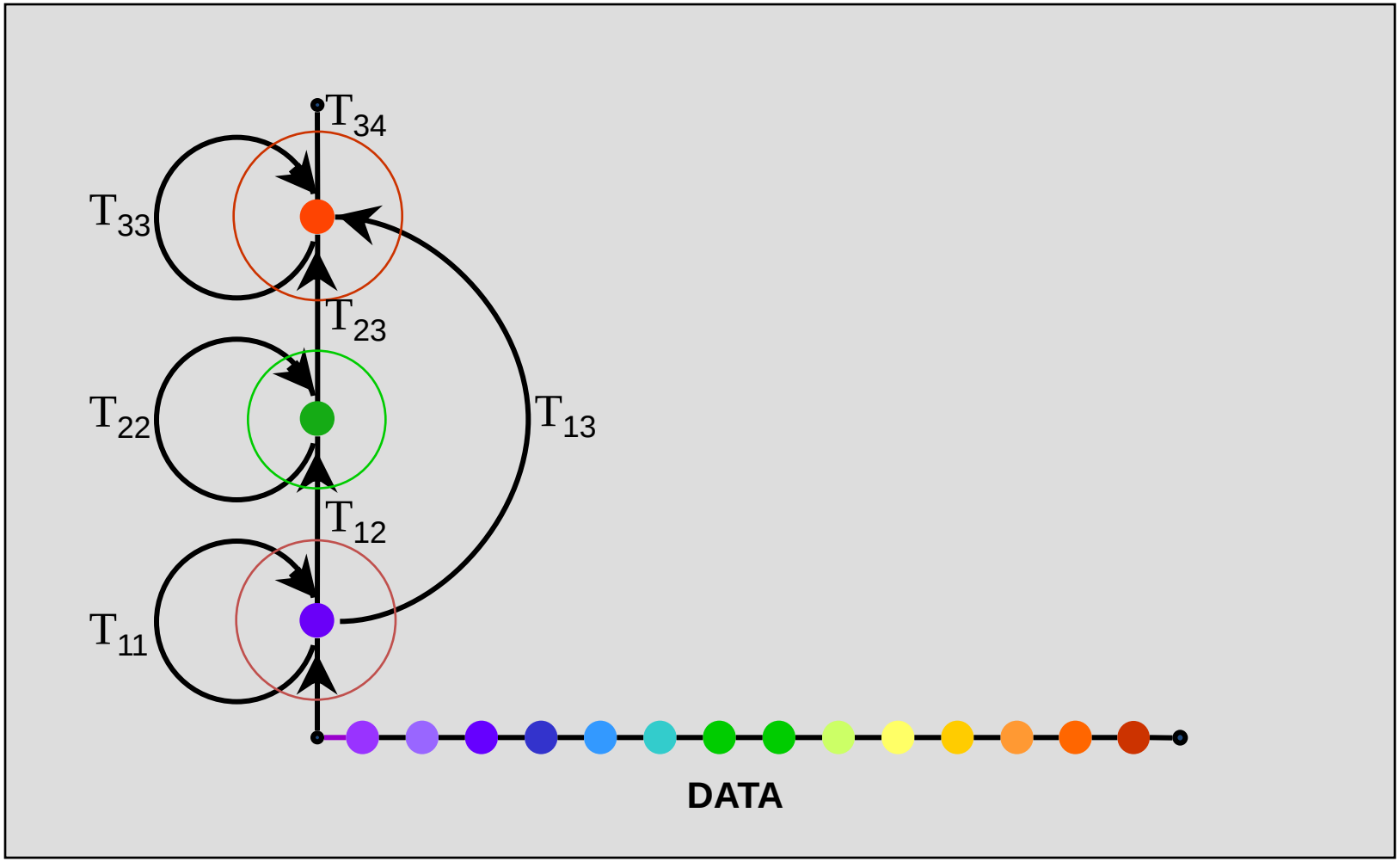
State specific insertion penalties are represented as self transition arcs for model vectors. Horizontal edges within the trellis will incur a penalty associated with the corresponding arc. Every transition within the model can have its own penalty.

Transition structures in models



State specific insertion penalties are represented as self transition arcs for model vectors. Horizontal edges within the trellis will incur a penalty associated with the corresponding arc. Every transition within the model can have its own penalty or score

Transition structures in models



This structure also allows the inclusion of arcs that permit the central state to be skipped (deleted)

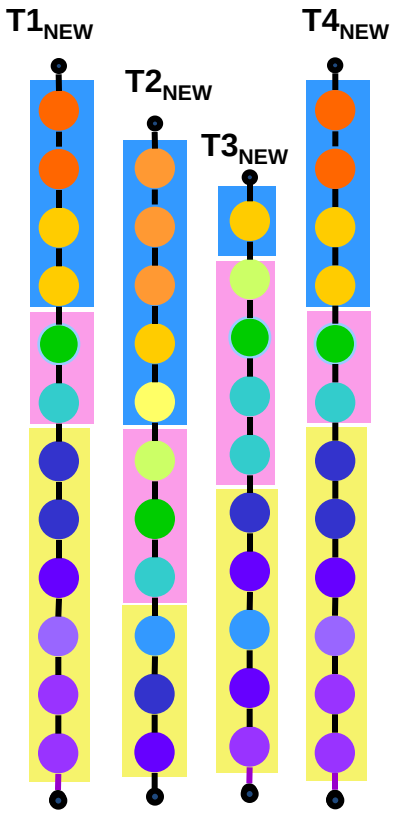
Other transitions such as returning to the first state from the last state can be permitted by inclusion of appropriate arcs

What should the transition scores be

- Transition behavior can be expressed with probabilities
  - For segments that are typically long, if a data vector is within that segment, the probability that the next vector will also be within it is high
  - If the i^{th} segment is typically followed by the j^{th} segment, but also rarely by the k^{th} segment, then, if a data vector is within the i^{th} segment, the probability that the next data vector lies in the j^{th} segment is greater than the probability that it lies in the k^{th} segment

- A good choice for transition scores are the negative logarithm of the probabilities of the appropriate transitions
  - T_{ii} is the negative of the log of the probability that if the current data vector belongs to the i^{th} state, the next data vector will also belong to the i^{th} state
  - T_{ij} is the negative of the log of the probability that if the current data vector belongs to the i^{th} state, the next data vector belongs to the j^{th} state
  - More probable transitions are less penalized. Impossible transitions are infinitely penalized

Modified segmental K-means AKA Viterbi training

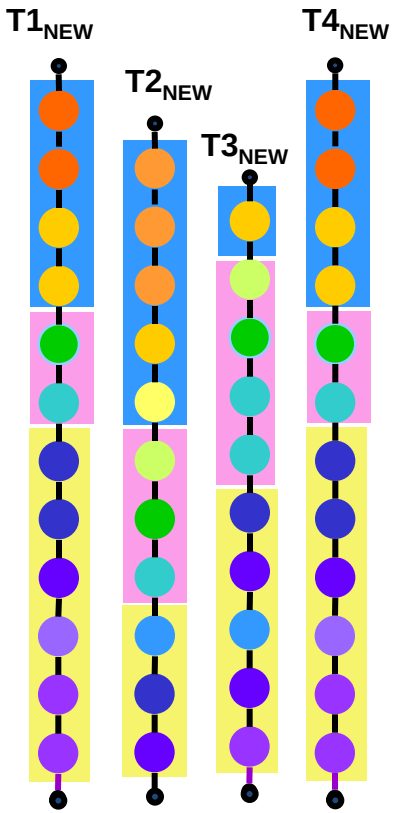


- Transition scores can be easily computed by a simple extension of the segmental K-means algorithm
- Probabilities can be counted by simple counting

$$P_{ij} = \frac{\sum_k N_{k,i,j}}{\sum_k N_{k,i}} \qquad T_{ij} = -\log(P_{ij})$$

- $N_{k,i}$ is the number of vectors in the i^{th} segment (state) of the k^{th} training sequence
- $N_{k,i,j}$ is the number of vectors in the i^{th} segment (state) of the k^{th} training sequence that were followed by vectors from the j^{th} segment (state)
  - E.g., No. of vectors in the 1st (yellow) state = 20
    No of vectors from the 1st state that were followed by vectors from the 1st state = 16
    $P_{11} = 16/20 = 0.8$; $T_{11} = -\log(0.8)$

Modified segmental K-means AKA Viterbi training



N = 4
N_{01} = 4
N_{02} = 0
N_{03} = 0

- A special score is the penalty associated with *starting* at a particular state
- In our examples we always begin at the first state
- Enforcing this is equivalent to setting $T_{01} = 0$, T_{0j} = infinity for j != 1
- It is sometimes useful to permit entry directly into later states
  - i.e. permit deletion of initial states
- The score for direct entry into any state can be computed as

$$P_j = \frac{N_{0j}}{N} \qquad T_{0j} = -\log(P_j)$$

- N is the total number of training sequences
- N_{0j} is the number of training sequences for which the first data vector was in the j^{th} state

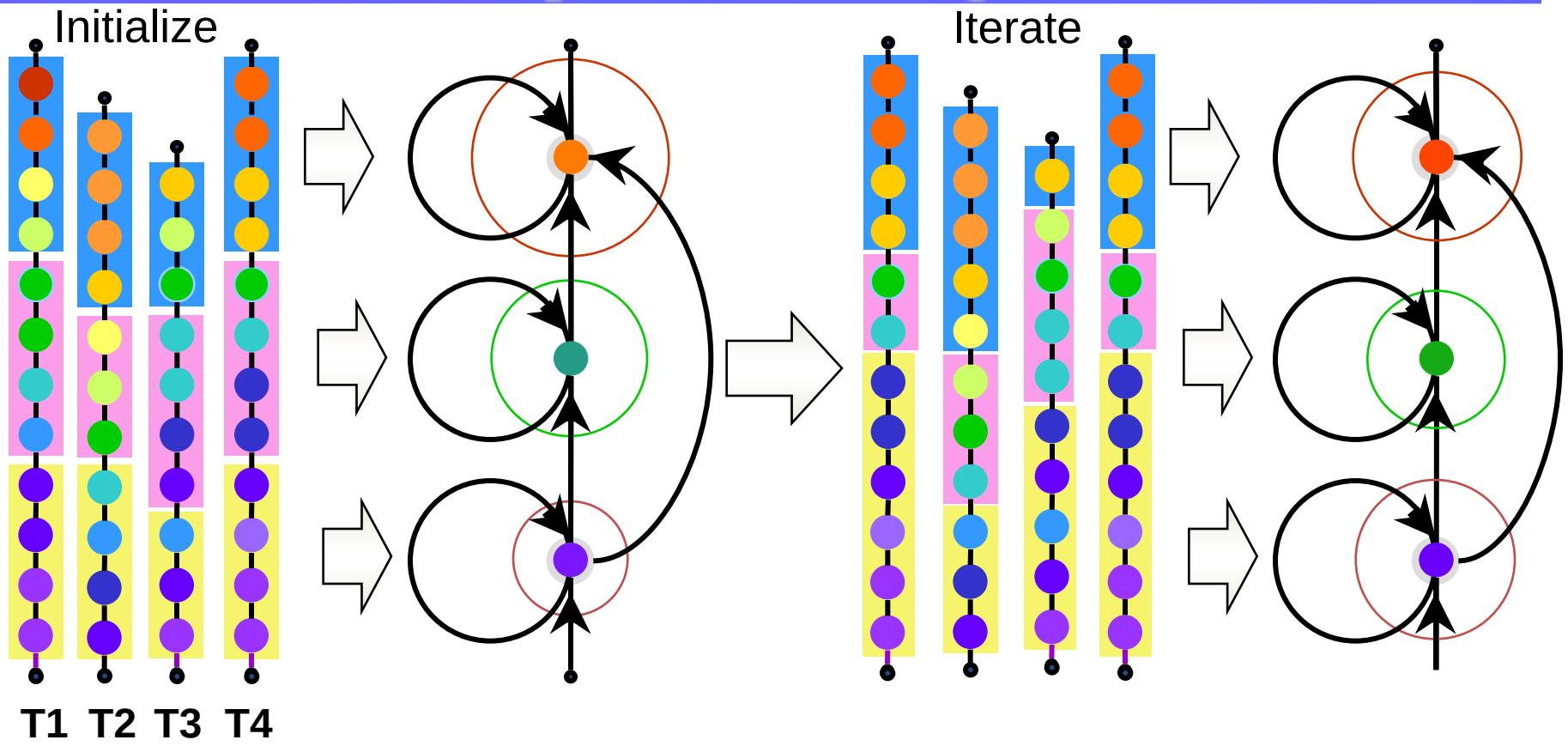
Modified segmental K-means AKA Viterbi training

- Initializing state parameters
  - Segment all training instances uniformly, learn means and variances
- Initializing T_{0j} scores
  - Count the number of permitted initial states
    - Let this number be M_0
  - Set all permitted initial states to be equiprobable: $P_j = 1/M_0$
  - $T_{0j} = -\log(P_j) = \log(M_0)$
- Initializing T_{ij} scores
  - For every state i, count the number of states that are permitted to follow
    - i.e. the number of arcs out of the state, in the specification
    - Let this number be Mi
  - Set all permitted transitions to be equiprobable: $P_{ij} = 1/M_i$
  - Initialize $T_{ij} = -\log(P_{ij}) = \log(M_i)$
- This is only one technique for initialization
  - You may choose to initialize parameters differently, e.g. by random values

Modified segmental K-means AKA Viterbi training

- The entire segmental K-means algorithm:
  1. Initialize all parameters
     - State means and covariances
     - Transition scores
     - Entry transition scores
  2. Segment all training sequences
  3. Reestimate parameters from segmented training sequences
  4. If not converged, return to 2

Alignment for training a model from multiple vector sequences

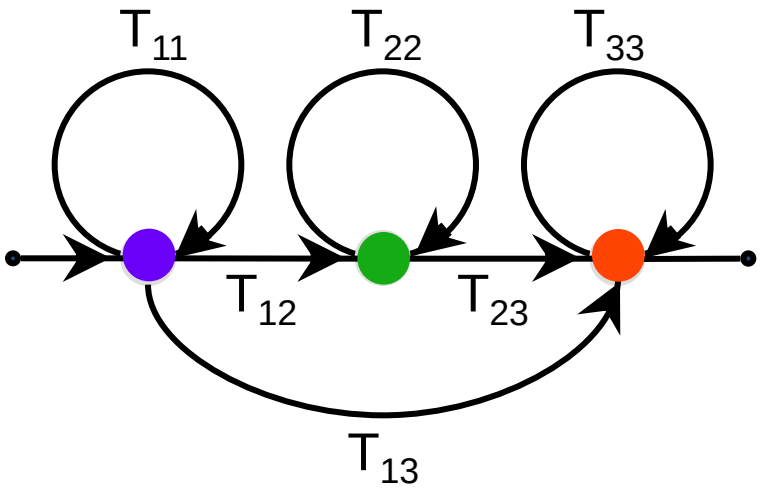


The procedure can be continued until convergence

Convergence is achieved when the total best-alignment error for all training sequences does not change significantly with further refinement of the model

The resulting model structure is also known as an HMM!

DTW and Hidden Markov Models (HMMs)



- This structure is a generic representation of a statistical model for processes that generate time series
- The "segments" in the time series are referred to as states
  - The process passes through these states to generate time series
- The entire structure may be viewed as *one* generalization of the DTW models we have discussed thus far
- In this example -- strict left-to-right topology
  - Commonly used for speech recognition

DTW -- Reversing Sense of "Cost"

- Use "Score" instead of "Cost"
  - The same cost function but with the sign changed (*i.e.* *negative* Euclidean distance (= $-\sqrt{\Sigma(x_i - y_i)^2}$; X and Y being vectors)
  - $-\Sigma(x_i - y_i)^2$; *i.e.* –ve Euclidean distance squared
  - Other terms possible:
    - Remember the Gaussian

Likelihood Functions for Scores

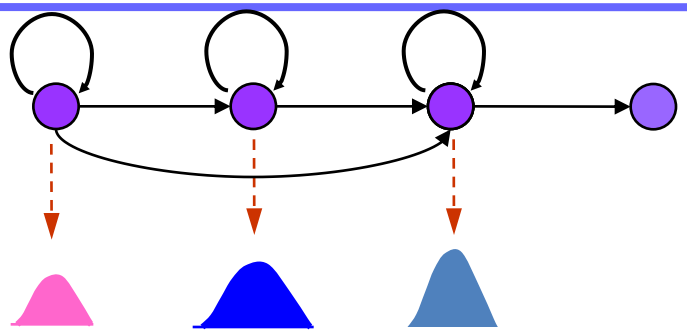
- HMM – inference equivalent to DTW modified to use a *probabilistic* function, for the local node or edge "costs" in the trellis
  - Edges have transition probabilities
  - Nodes have *output* or *observation probabilities*
    - They provide the probability of the observed input
    - The output probability may be a Gaussian
  - Again, the goal is to find the template with highest probability of matching the input

- Probability values as "costs" are also called *likelihoods*

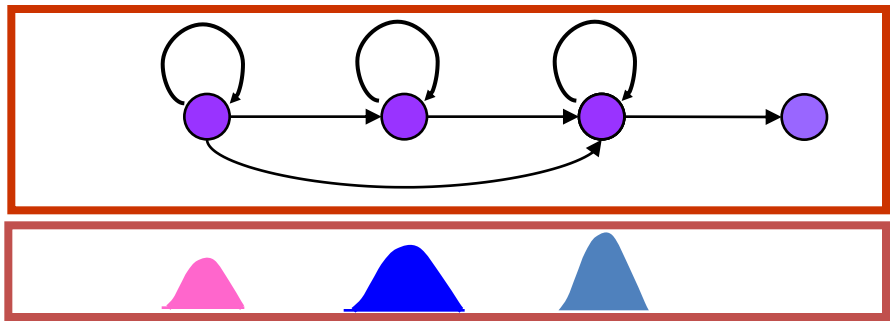
Log Likelihoods

- May use probabilities or likelihoods instead of cost
  - Scores combines multiplicatively along a path – cost of a path = Product_over_nodes(cost of node) * Product_over_edges(cost of edge)
- May use log probabilities
  - Scores add as in DTW
    - Max instead of Min
- May use *negative* log probabilities
  - *Cost* adds as in DTW
  - *More on this later*

Hidden Markov Models



- A Hidden Markov Model consists of two components
  - A state/transition backbone that specifies how many states there are, and how they can follow one another
  - A set of probability distributions, one for each state, which specifies the distribution of all vectors in that state



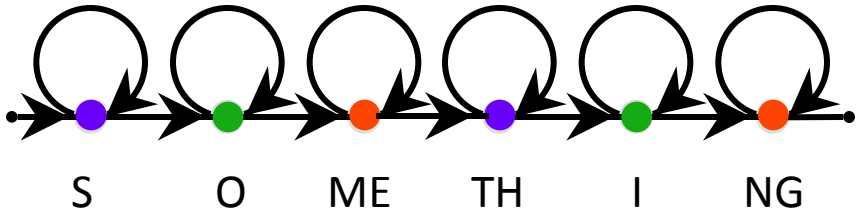
- This can be factored into two separate probabilistic entities
  - A probabilistic Markov chain with states and transitions
  - A set of data probability distributions, associated with the states

Determining the Number of States

- How do we know the number of states to use for any word?
  - We do not, really
  - Ideally there should be at least one state for each "basic sound" within the word
    - Otherwise widely differing sounds may be collapsed into one state
    - The average feature vector for that state would be a poor representation
  - For efficiency, the number of states should the minimum needed to achieve the desired level of recognition accuracy
  - These two are conflicting requirements, usually solved by making some educated guesses

Determining the Number of States

- For small vocabularies, it is possible to examine each word in detail and arrive at reasonable numbers:



- For larger vocabularies, we may be forced to rely on some *ad hoc* principles
  - *E.g.* proportional to the number of letters in the word
    - Works better for some languages than others
    - Spanish, Japanese (Katakana/Hiragana), Indian languages..

The State Output Distribution

- The state output distribution is a probability distribution associated with each HMM state
  - The negative log of the probability of any vector as given by this distribution would be the node cost in DTW
- The state output probability distribution could be any distribution at all
- We have considered Gaussian state output distributions

$$P(x \mid j) = \frac{1}{\sqrt{\prod_l 2\pi\sigma_{j,l}^2}} \exp\left(-0.5 \sum_l \frac{(x_l - m_{j,l})^2}{\sigma_{j,l}^2} \right)$$

$$d_j(v) = -\log(P(x \mid j)) = 0.5 \sum_l \log(2\pi\sigma_{j,l}^2) + 0.5 \sum_l \frac{(x_l - m_{j,l})^2}{\sigma_{j,l}^2}$$

Node cost for DTW (note change in notation)

- More generically, we can assume it to be a *mixture* of Gaussians

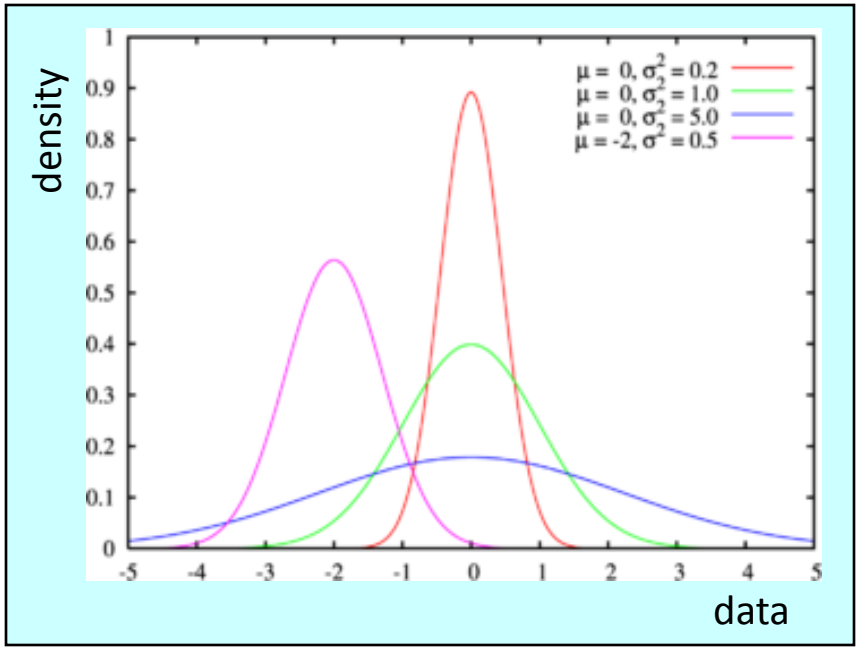
$$P(x \mid j) = \sum_k \frac{w_k}{\sqrt{\prod_l 2\pi\sigma_{j,k,l}^2}} \exp\left(-0.5 \sum_l \frac{(x_l - m_{j,k,l})^2}{\sigma_{j,k,l}^2} \right)$$

$$d_j(v) = -\log(P(x \mid j))$$

- More on this later

The Gaussian Distribution

- What does a Gaussian distribution look like?
- For a single (scalar) variable, it is a bell-shaped curve representing the density of data around the mean
- Example:



Four different scalar Gaussian distributions, with different means and variances

The mean is represented by μ, and variance by σ^2

μ and σ are the *parameters* of the Gaussian distribution

(Taken from Wikipedia)

The Scalar Gaussian Function

- The Gaussian density function (the bell curve) is represented by a somewhat complicated looking formula:

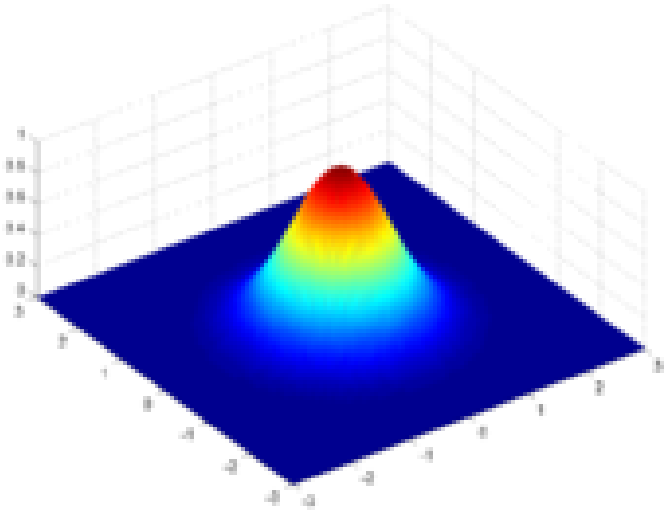
$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{1}{2\sigma^2}(x-\mu)^2}$$

- $p(x)$ is the density function of the variable x, with mean μ and variance σ^2

- The attraction of the Gaussian function (regardless of how appropriate it is!) comes from how easily the mean and variance can be estimated from *sample data* $x_1, x_2, x_3 \ldots x_N$
  - $\mu = (\text{Sum } x_i)/N$
  - $\sigma^2 = (\text{Sum } (x_i^2 - \mu^2))/N$

The 2-D Gaussian Distribution

- However, our speech data are not scalar values, but vectors!
- The Gaussian distribution for vector data becomes quite a bit more complex
- Let's first see what a 2-D Gaussian density function looks like, shown as a 3-D plot:
  - Same bell shape, but now in 2-D



- Distributions for higher dimensions are tough to visualize!

The Multidimensional Gaussian Distribution

- Instead of variance, the multidimensional Gaussian has a *covariance matrix*

- The multi-dimensional Gaussian distribution of a vector variable x with mean μ and covariance Σ is given by:

$$f(x) = (2\pi)^{-N/2} \det(\Sigma)^{-1/2} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right)$$

  - where N is the vector dimensionality, and *det* is the determinant function

- The complexity in a full multi-dimensional Gaussian distribution comes from the covariance matrix, which accounts for *dependencies* between the dimensions

The Diagonal Covariance Matrix

- In speech recognition, we frequently assume that the feature vector dimensions are *independent* of each other
- *Result*: The covariance matrix is reduced to a diagonal form

$$\Sigma = \begin{pmatrix} \sigma_1^2 & 0 & \cdots & 0 \\ 0 & \sigma_2^2 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & \sigma_N^2 \end{pmatrix}$$

$$f(x) = (2\pi)^{-N/2} \det(\Sigma)^{-1/2} \exp\left(-\frac{1}{2}(x-\mu)^T \Sigma^{-1}(x-\mu)\right)$$

$$= \frac{1}{\sqrt{(2\pi)^N \prod_i \sigma_i^2}} \exp\left(-\sum_i \frac{(x_i-\mu_i)^2}{2\sigma_i^2}\right) = \prod_i \frac{1}{\sqrt{2\pi\sigma_i^2}} \exp\left(-\frac{(x_i-\mu_i)^2}{2\sigma_i^2}\right)$$

- Further, each σ_i (the i-th digonal element in the covariance matrix) is easily estimated from x_i and μ_i like a scalar

Recap: What are Markov Models?

- Markov process: Process where the state at any time depends only on the state at the previous time instant
- Markov model: is a statistical model for describing time series of events or observations as outcomes of a Markov process
- The model consists of a finite set of *states* with *transitions* between them (including self transitions)
  - Thus, we can model state sequences using them
- Transitions can have probabilities associated with them
  - The probability of a transition from state *i* to state *j* depends only on state *i*, and not on the earlier history; *i.e.*

    $P(s_t \mid s_{t-1}, s_{t-2}, s_{t-3}, \ldots) = P(s_t \mid s_{t-1})$, where s_t is the state of the model at time t
    - This usually called the *Markovian* property of the model
  - The probabilities of all transitions out of any given state must sum to 1

What are Markov Models? (contd.)

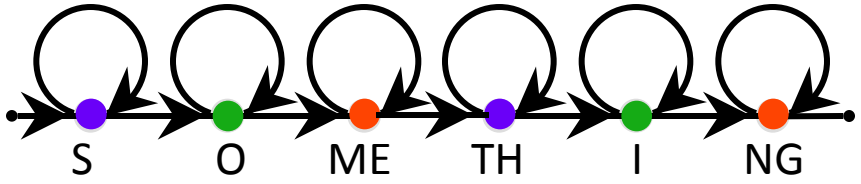
- The *states* collectively model a set of *events* or *observations*
  - The observations can be discrete or continuous valued
  - Each state has a probability distribution that defines which observations are produced with what probability
    - For continuous valued observations, this is a probability density function
      - Many medium and large vocabulary systems use the Gaussian probability density function
  - This is also often called the *emission* or *observation* probability for the state

What are *Hidden* Markov Models?

- In many real-world processes that generate time series data, it is not possible to know the state sequence that produced them
  - *i.e.* the actual state sequence is *hidden* from an observer
    - In fact, it may not even be possible to know *what* the *set of states* is, or the state transition structure
  - This is the primary difference between regular Markov and hidden Markov models
- The model has state transitions with the Markovian property
- Each state has a probabilistic model for the *generation* (or *emission*) of *events* (or *observations*)
- HMMs are *generative* models; they model the *production* or *emission* of the observed time series of events
  - Note that the *actual* process that produces the time series may not be a Markovian process at all

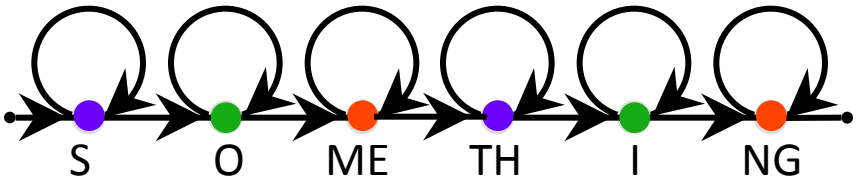
Modeling Speech With HMMs

- *Example*: the structure below, for capturing the production of the word "something", is that of an HMM



- We use the six states to model the six distinct segments that are predominantly uniform within themselves
- There is usually a *start state* and a *final state* (*S* and *NG* above)
- Each state has a probabilistic function that describes the sound produced when in that state
  - Thus, the state labelled *S* would have a very high probability associated with feature vectors for the *S* sound
- We could use fewer states, but then each would need a more complex probability model
- We could use more states to obtain a more precise model, *e.g.* for distinguishing between the first and second halves of any segment

Modeling Speech With HMMs (contd.)



- The transition structure permits the sequence of sounds that make up the word, with varying durations for each segment
  - Obviously, the model *allows* inordinately strange durations as well
  - This lack of a good *duration model* is one of the limitations of HMMs

- Note that this is only a *model* of the word being spoken
  - *i.e.* It is an *approximation* of the process within us humans that actually generates the word
  - It is clearly preposterous to think that we go through precisely six states in pronouncing the word, or that we make abrupt transitions from one state to another
- *Since we may not know the set of states or the transition structure, the main problem in using HMMs is to attempt to discover it, based solely on the observed sequences of events (feature vectors)*

Some Reasons for Using HMMs

- One of the main reasons for using HMMs is that efficient and mathematically well-understood algorithms exist for solving three fundamental HMM problems

- Mathematically elegant ways of incorporating other sources of knowledge about speech (than just acoustics)

- Highly flexible for use in a wide range of applications, small to very large vocabulary systems

Three Fundamental Problems

- HMMs require solutions to three basic problems:
  - *Likelihood evaluation*: Given an HMM M, and an observation (input) sequence $X = x_1, x_2, x_3, \ldots x_N$, find $P(X|M)$
    - $P(X|M)$ is variously called the conditional probability of X given M, the *likelihood* of X given M, and sometimes the converse
    - This is equivalent to the minimum cost problem in DTW
  - *State sequence decoding*: Given an HMM M, and an observation sequence X, find the most likely HMM state sequence for M and X

    i.e. $\text{argmax}(s_1, s_2, s_3 \ldots s_N)\ P(s_1, s_2, s_3 \ldots s_N \mid X, M)$
    - This is equivalent to finding the actual minimum cost path in DTW
  - *HMM estimation*: Given an HMM structure (*i.e.* set of states and transition structure) , and some *labeled training data*, estimate the HMM parameters that maximize the likelihood of the training data
    - This is the HMM *training* problem, the hardest one and the subject of a later talk

Problem 1: HMM Likelihood Evaluation

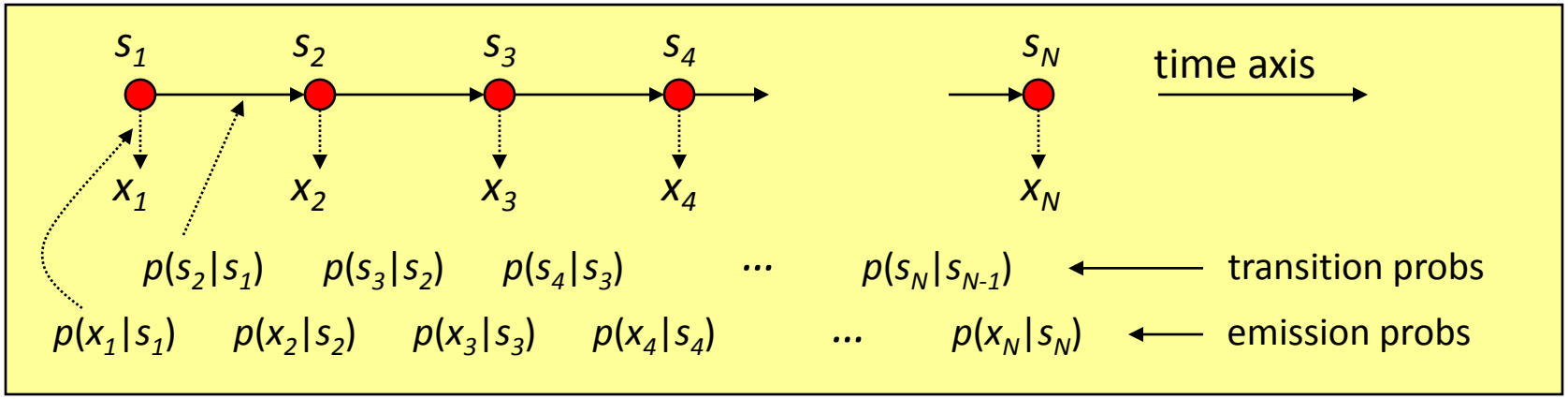
- Likelihood evaluation is the equivalent of finding least cost distance between a given template and an input sequence
  - In this case, between a given HMM and the observation sequence

- Since HMMs are probabilistic, and generative models, we want to find the probability or likelihood that a given HMM would generate the given observation sequence X
  - *i.e.* compute $P(X|M)$

- Thus, if we have HMMs for each word in a vocabulary, $word_1$, $word_2$, $word_3$,… and we get some unknown spoken input (the observation sequence X), we wish to compute $P(X|word_i)$ for every i, and choose the maximum

Forward Algorithm: $P(X \mid word)$

- How can we compute $P(X \mid word)$?
  - X is an N long feature vector sequence: $x_1, x_2, x_3 \ldots x_N$
  - *word* is an HMM
- Consider an N long *state sequence* through the HMM, $s_1, s_2, s_3 \ldots s_N$, from its start state to its final state
- We can compute $P(X, s_1, s_2, s_3 \ldots s_N \mid word)$; *i.e.* the probability of producing X by following the given state sequence:



- Thus, $P(X, s_1, s_2, s_3 \ldots s_N \mid word)$ = product of the individual emission and transition probabilities

Computing $P(X \mid word)$ (contd.)

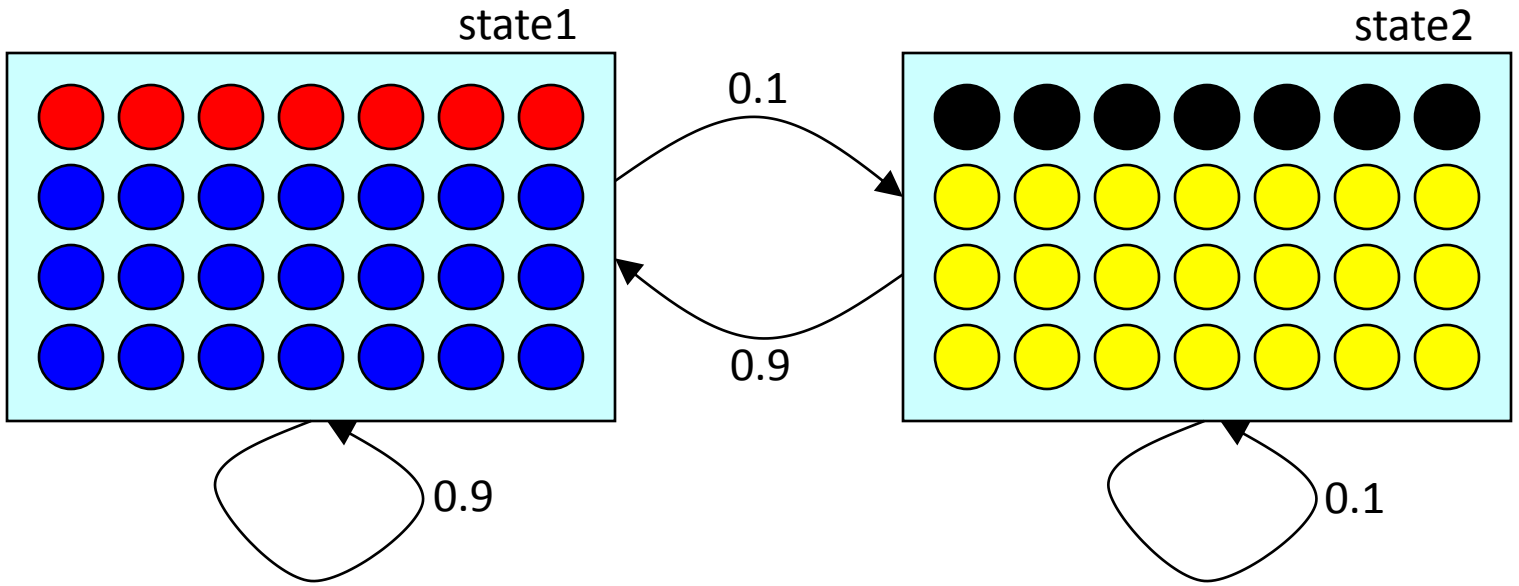
- The "actual" state sequence that generates the observed sequence is hidden in HMMs
  - i.e. we need to consider *all possible* state sequences:
  - $P(X \mid word) = \Sigma\ (P(X, s_1, s_2, s_3 \ldots s_N \mid word))$, summed over all possible state sequences $s_1, s_2, s_3 \ldots s_N$ through the HMM

- We now have a definition for $P(X \mid word)$, but it is computationally intractable as formulated
  - The number of possible state sequences of length N explodes exponentially with N

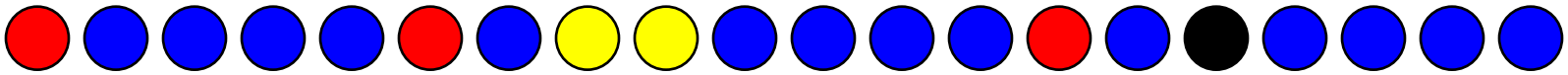
- Is there an efficient algorithm for computing $P(X \mid word)$?
  - Yes! The trellis to our rescue again

Example: Markov Model

Equal prob. of starting in either state



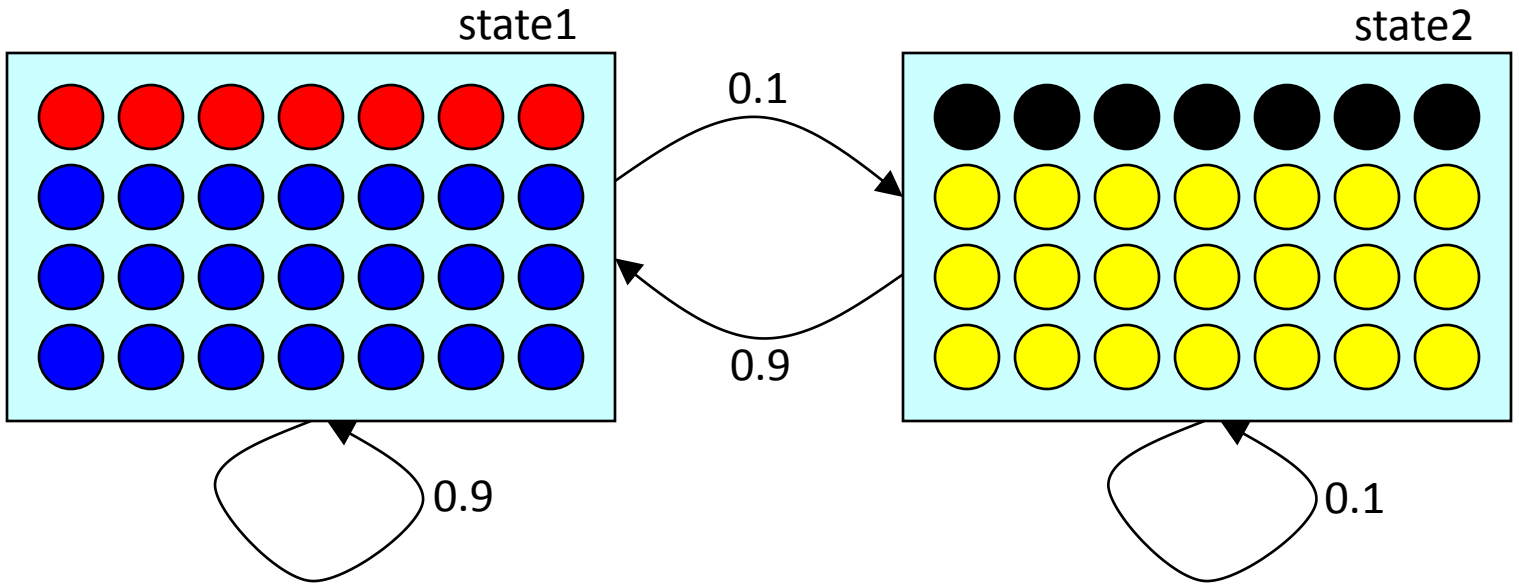
Observed data:



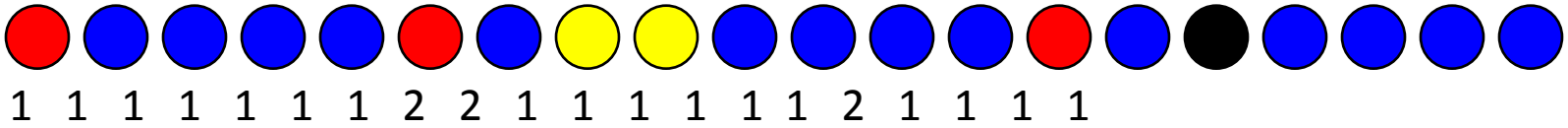
- Can you tell the state sequence that was taken, by looking at the data?

Markov Model

Equal prob. of starting in either state

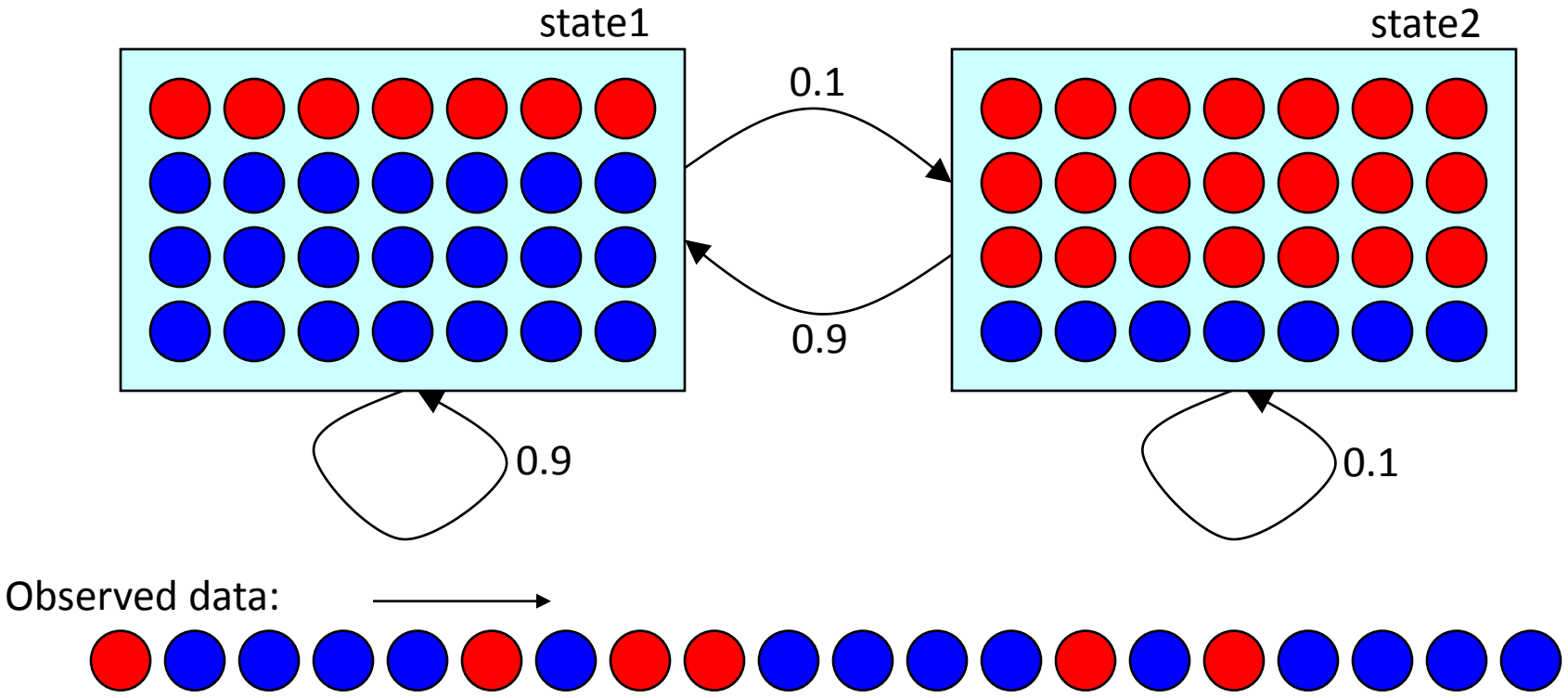


Observed data:



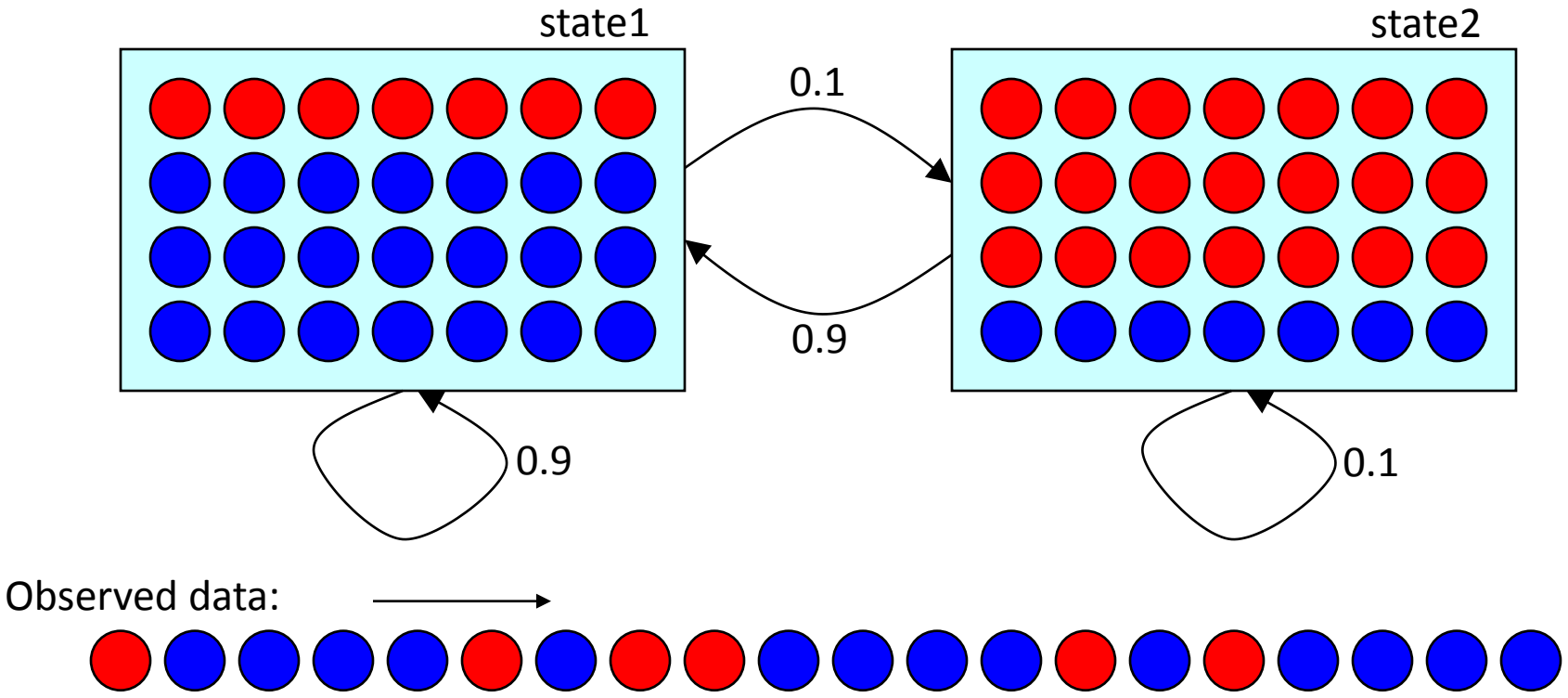
- YES!! You can tell the state sequence
- And so, this Markov model is a *plain* Markov model

Hidden Markov Model



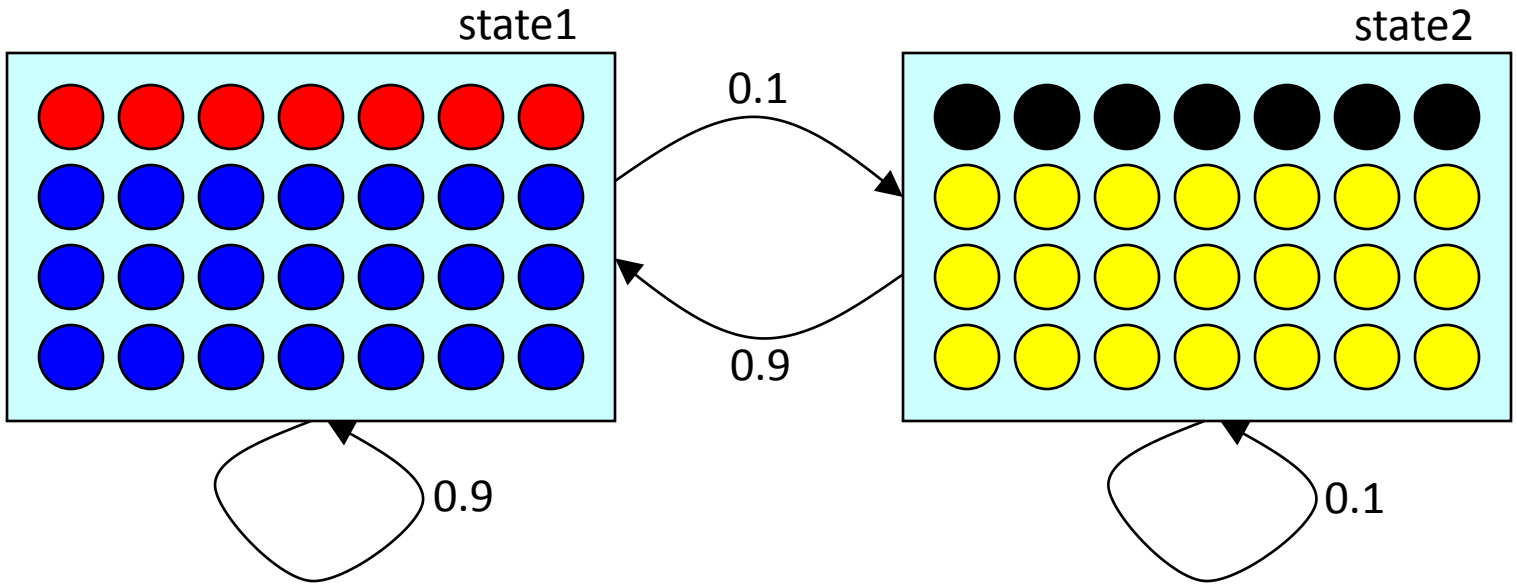
- Can you tell the state sequence that was taken, by looking at the data?

Hidden Markov Model



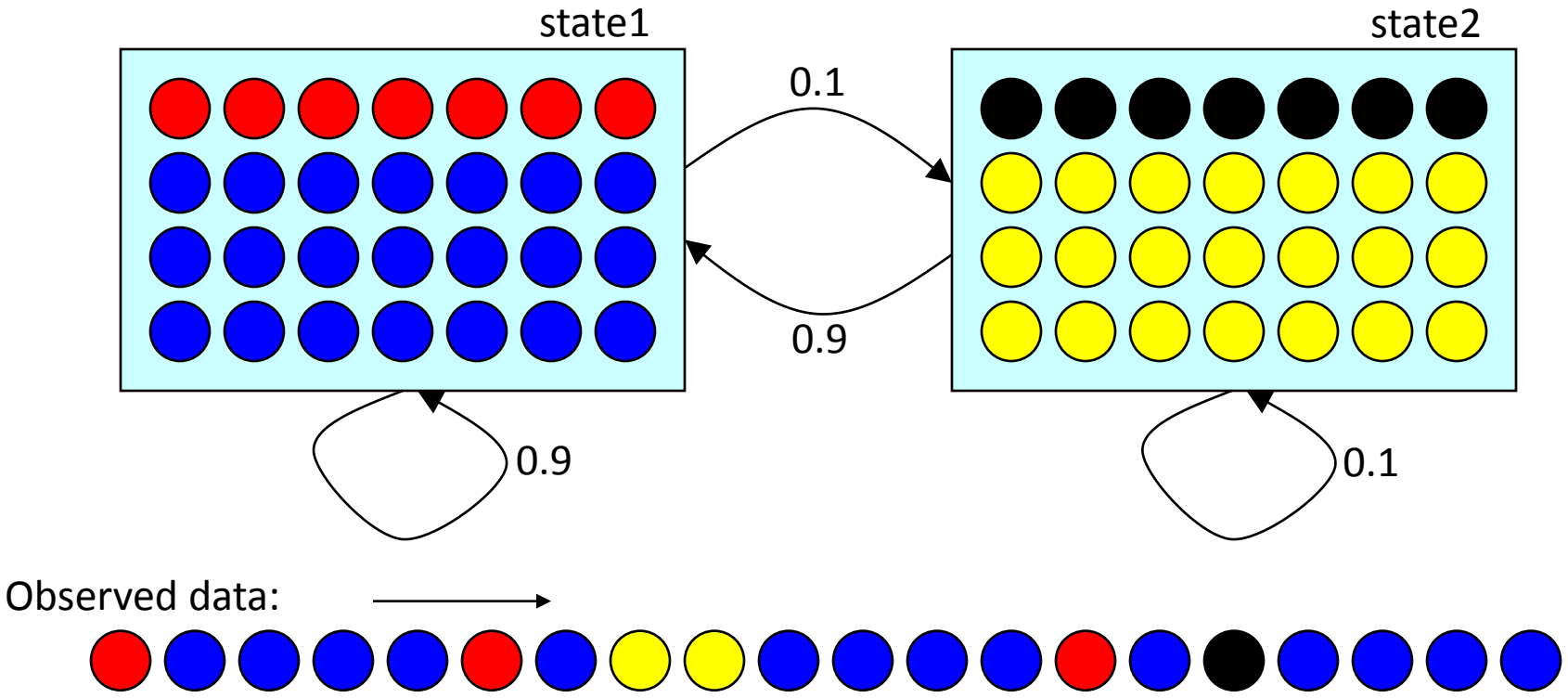
- NO!! You can NOT tell the state sequence
- And so, this is a *hidden* Markov model

Markov Model



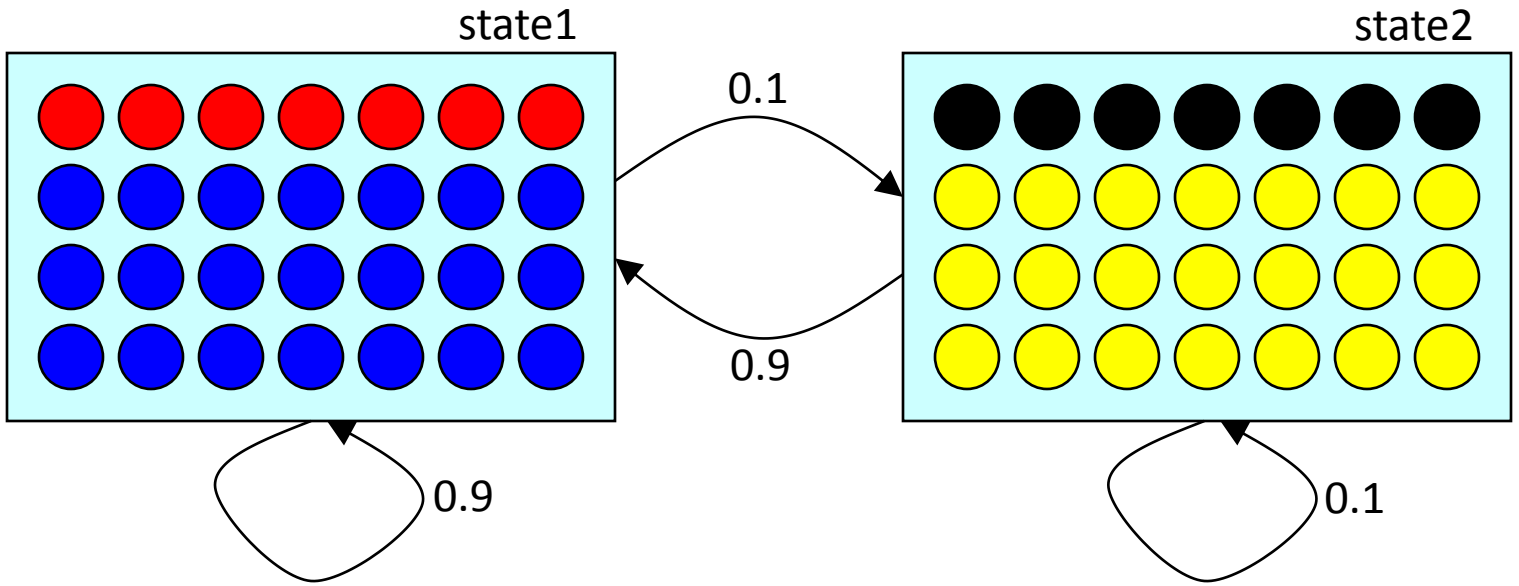
- *Q*: What is prob(ball2 = blue), *knowing* that ball1 = red?
- If ball1 = red, we know we're in state 1 at that time
- For ball2 to be blue, we have to STAY in state 1, AND pick blue ball
- So, prob = ???

Markov Model

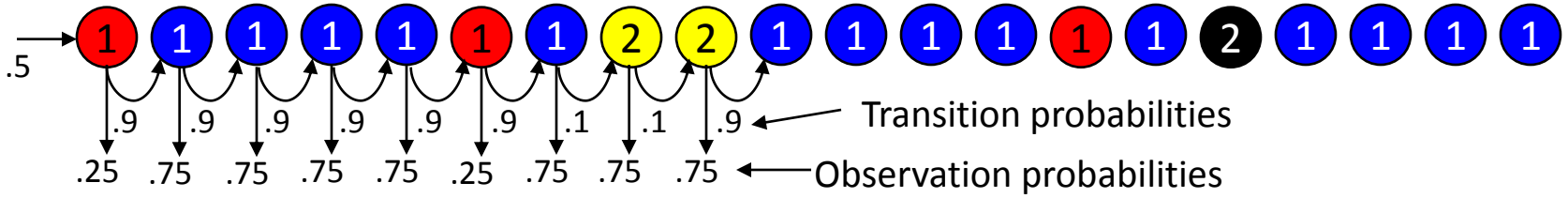


- What is probability of observing this entire sequence above?
  - Remember, equal prob of starting in either state!

Markov Model

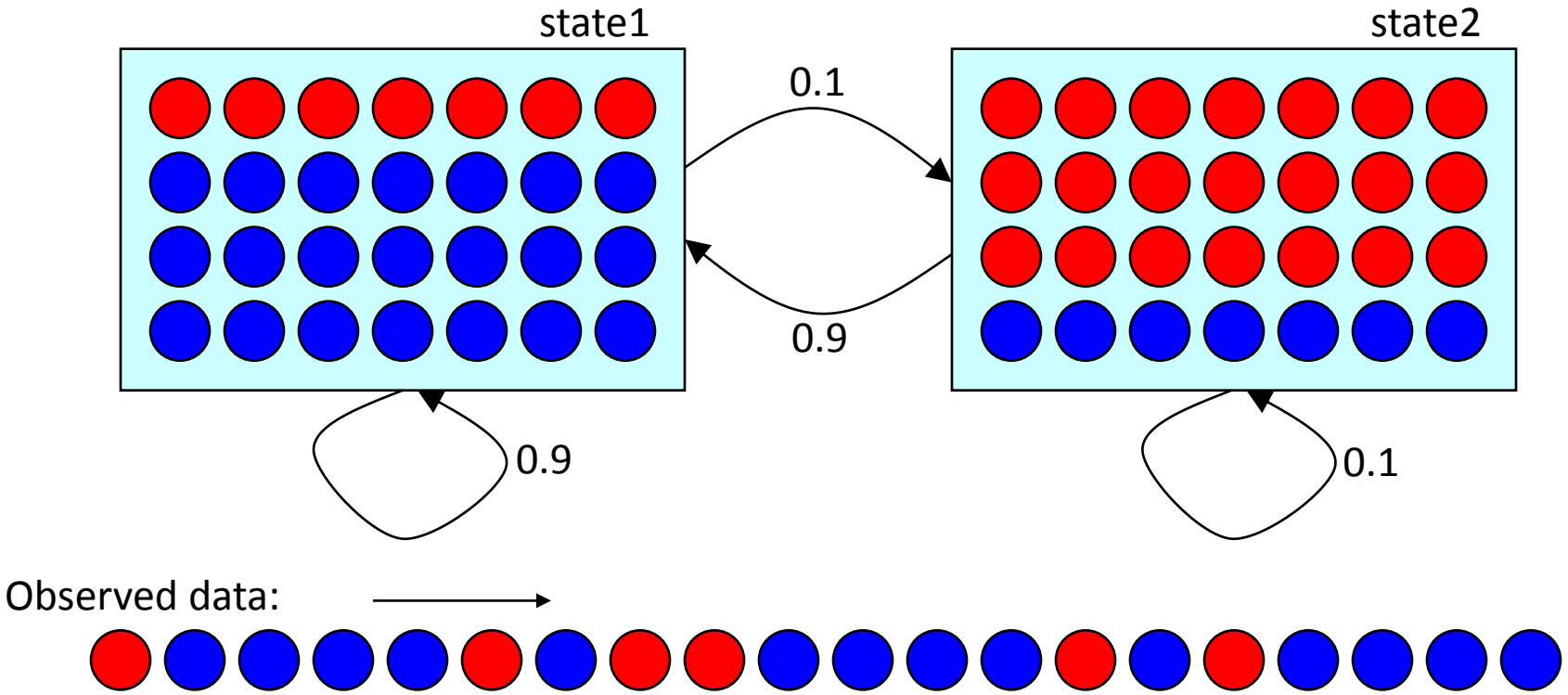


Observed data:



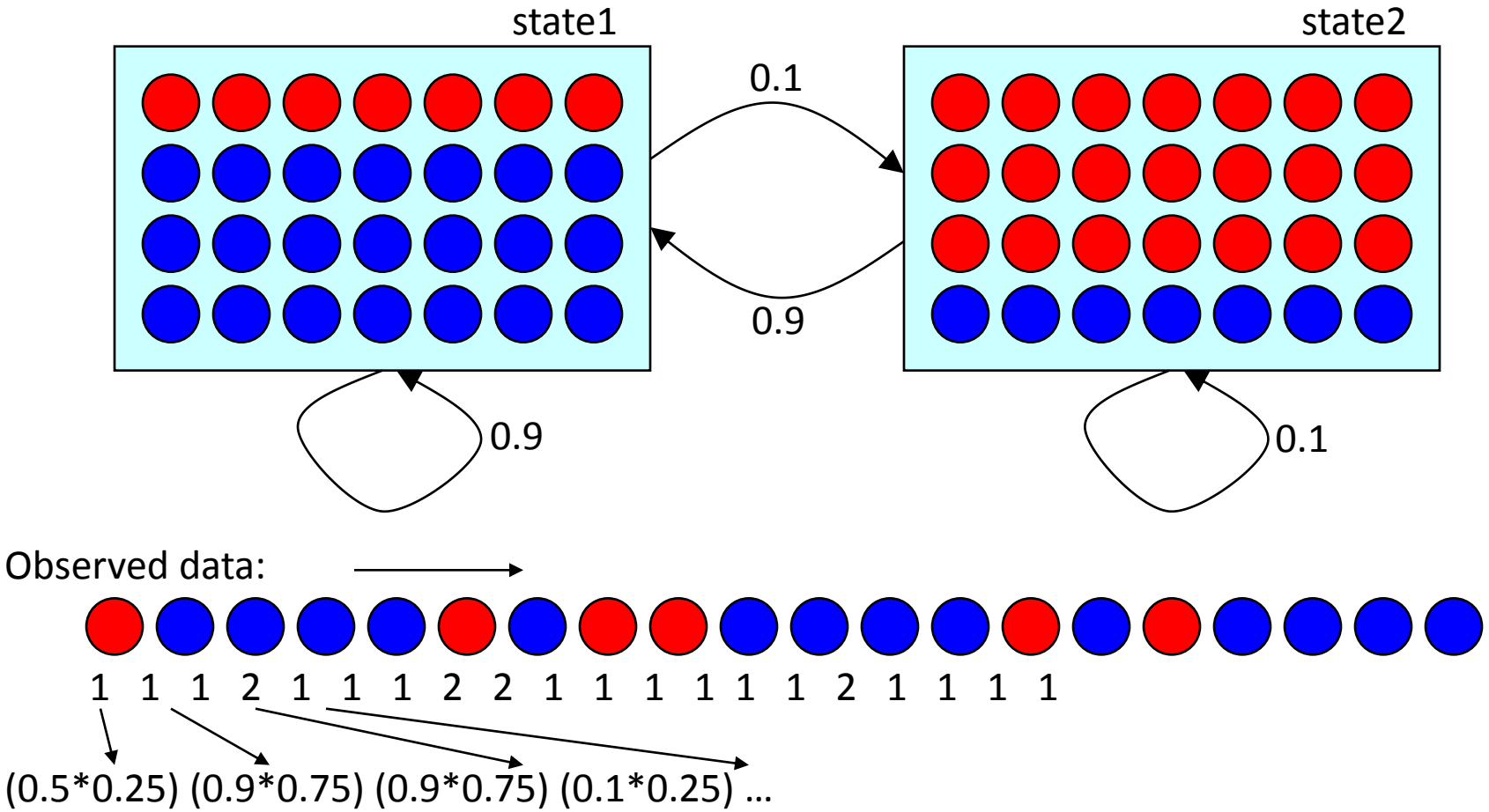
$(0.5*0.25)\ (0.9*0.75)^4\ (0.9*0.25)\ (0.9*0.75)\ (0.1*0.75)^2\ (0.9*0.75)^4\ (0.9*0.25)\ (0.9*0.75)\ (0.1*0.25)\ (0.9*0.75)^4$

Hidden Markov Model



- What is probability of observing this entire sequence above?
  - Big trouble! Many possible paths exist; not just a single path
- Need to consider ALL POSSIBLE paths and sum their probs!

Hidden Markov Model



- If we *assume* a particular state sequence, we can find the probability of the observation using that state sequence

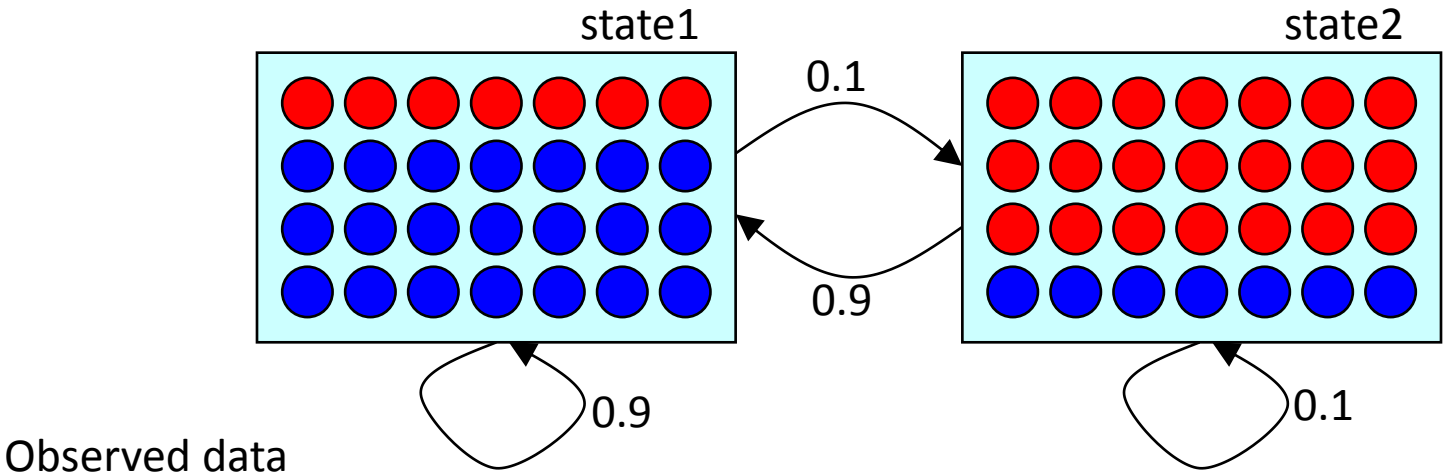
Hidden Markov Model

- So, we know how to compute probability of input and taking a particular linear path through the HMM

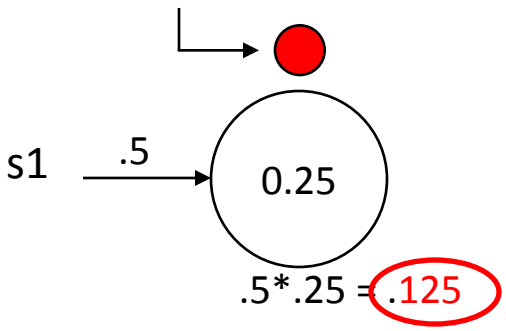
- Let us see how we can compute the probability of the input sequence, without constraining us to any single path
  - *i.e.* by considering ALL possible paths through the HMM

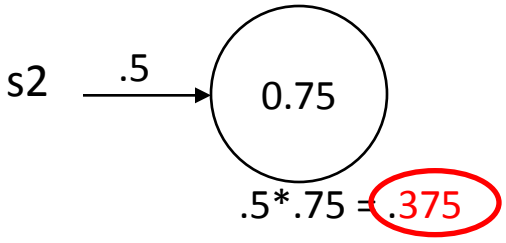
- Consider the input sequence one symbol at a time…

Hidden Markov Model

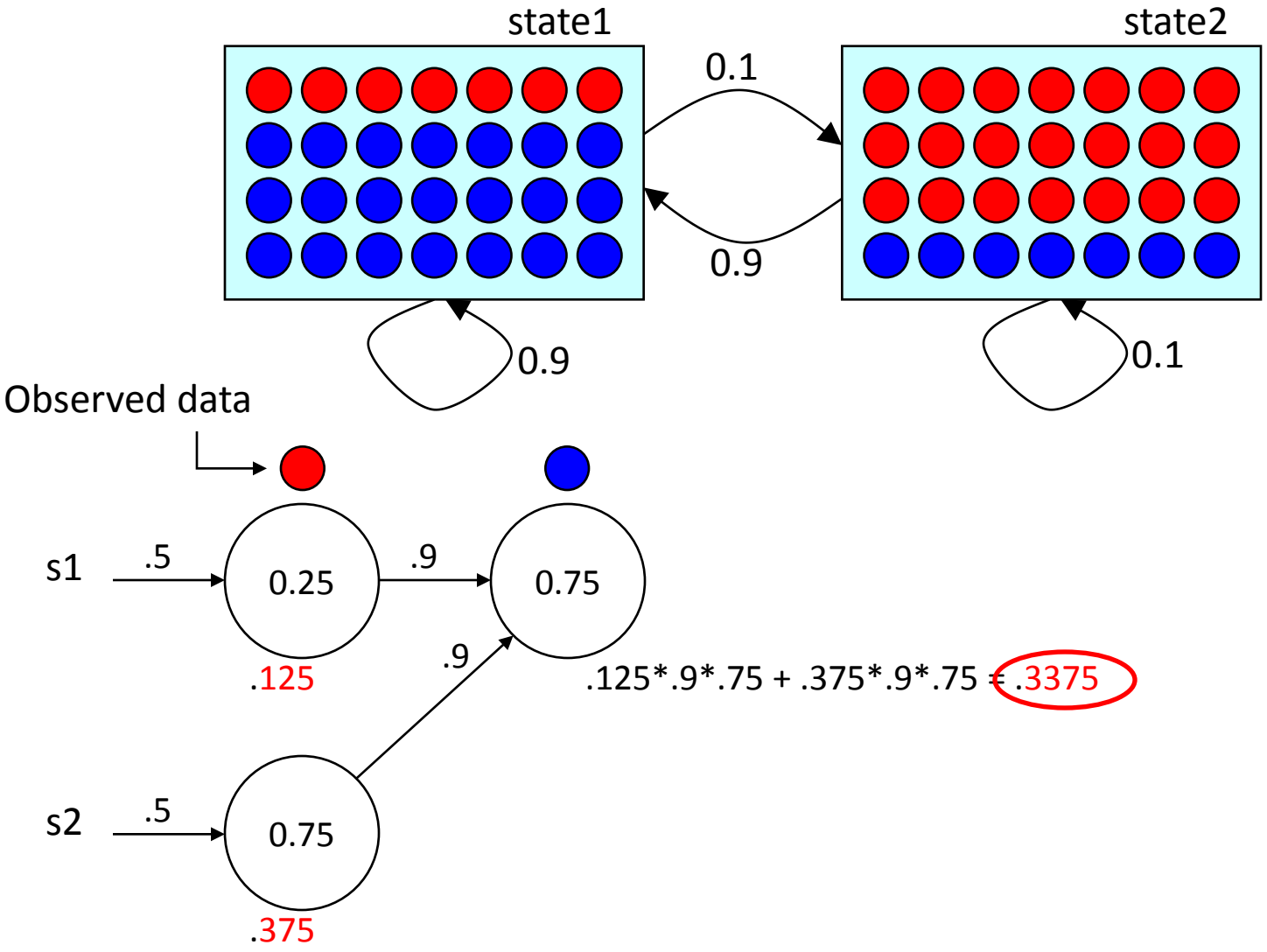


Observed data

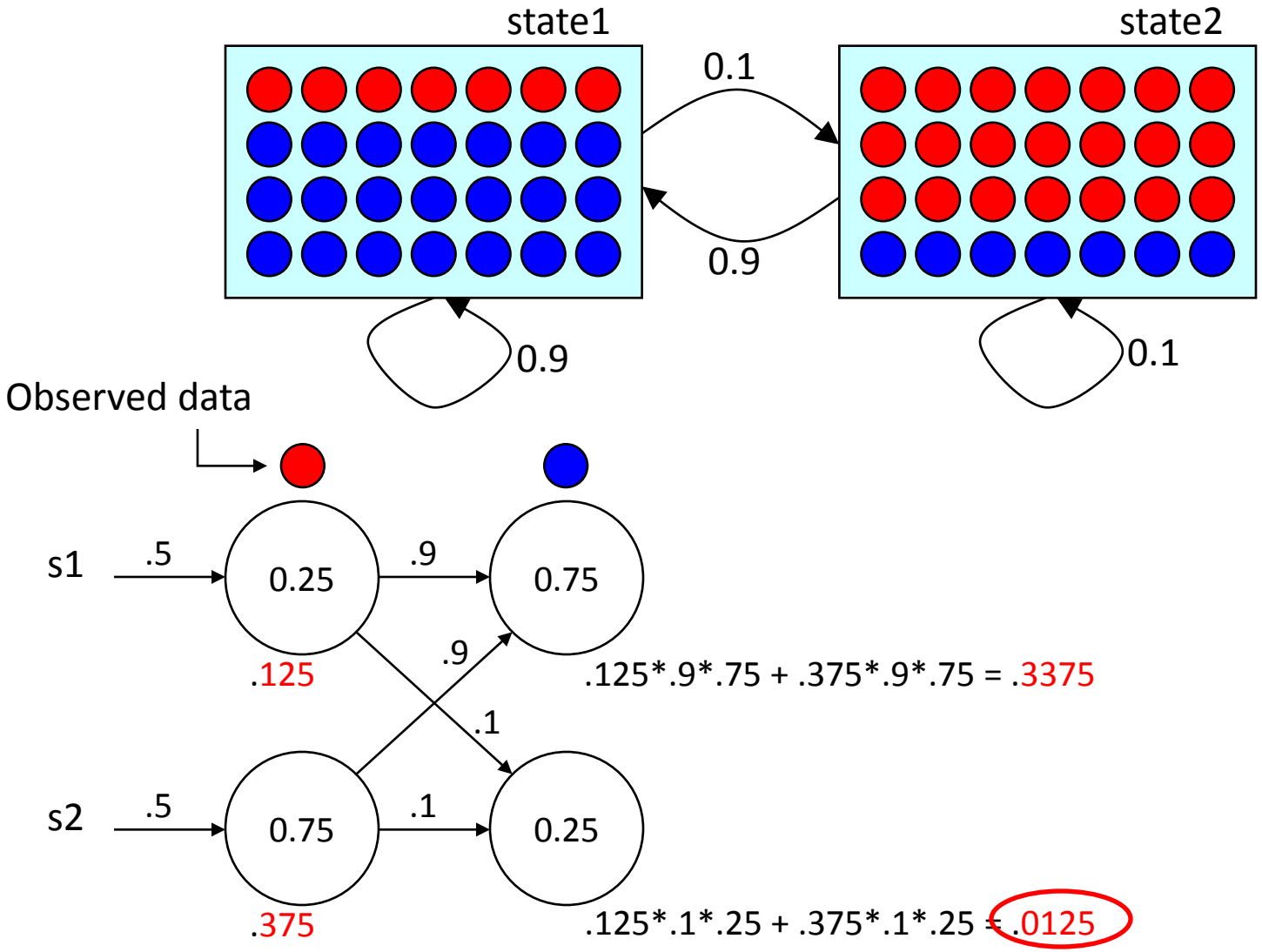




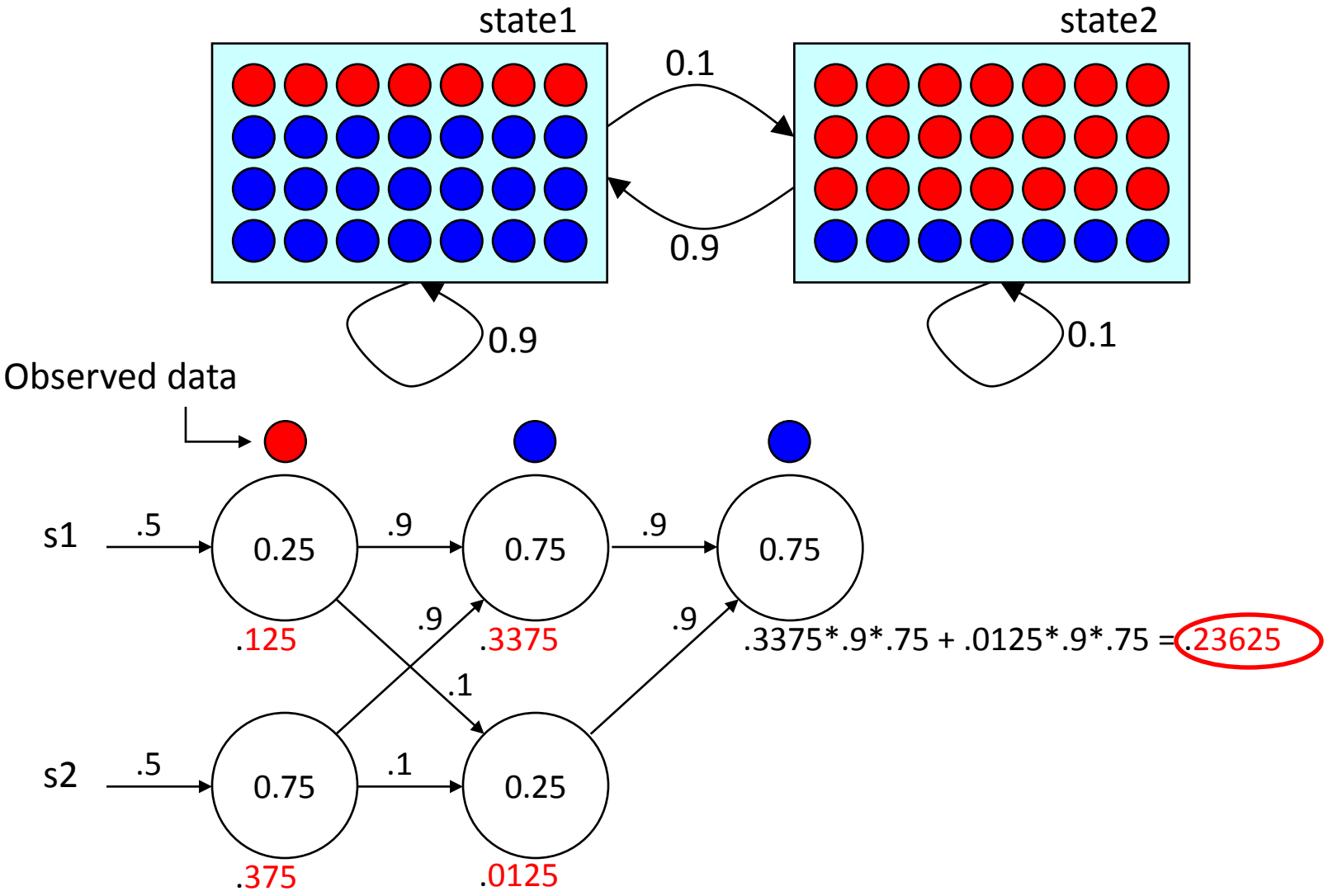
Hidden Markov Model



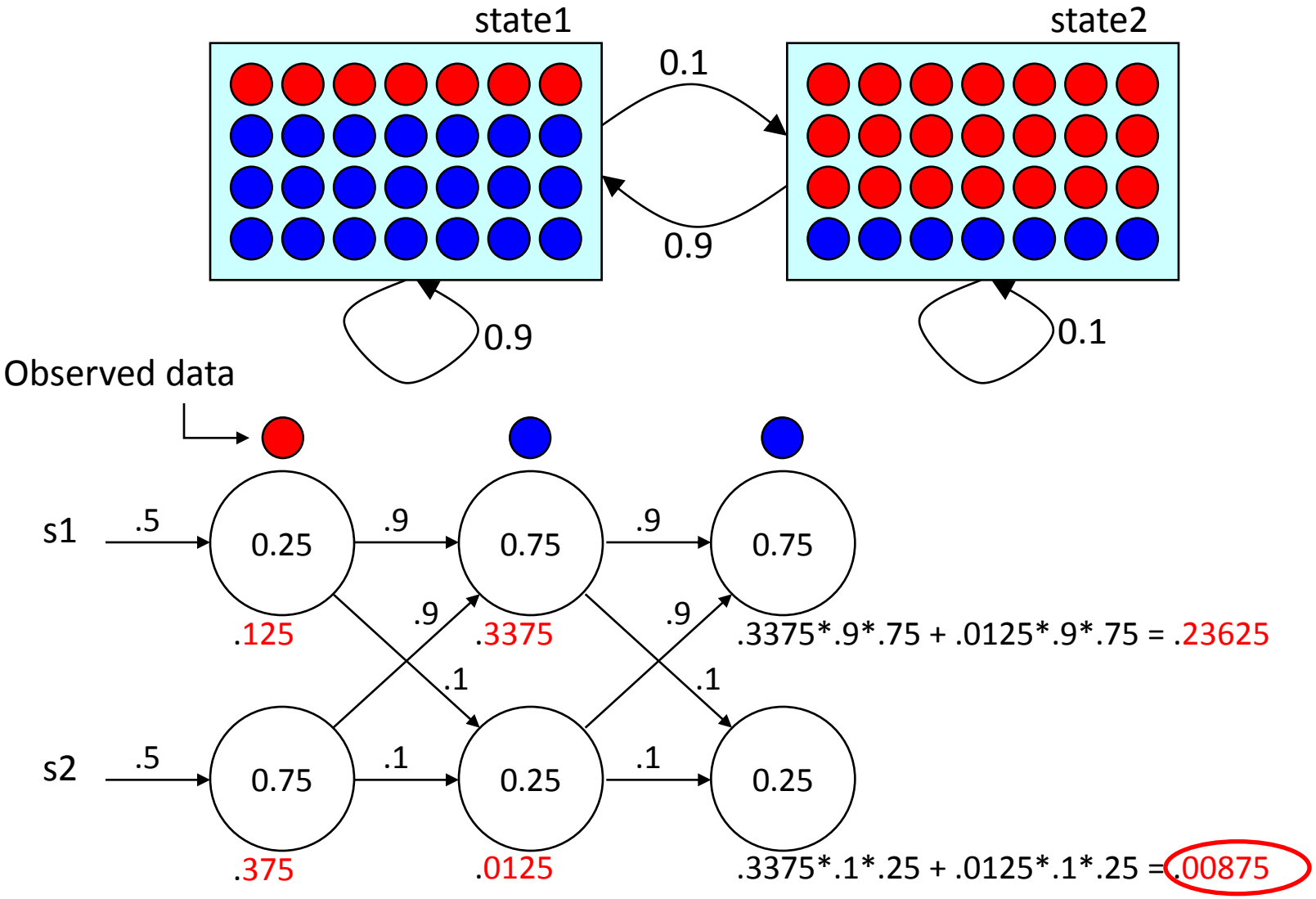
Hidden Markov Model



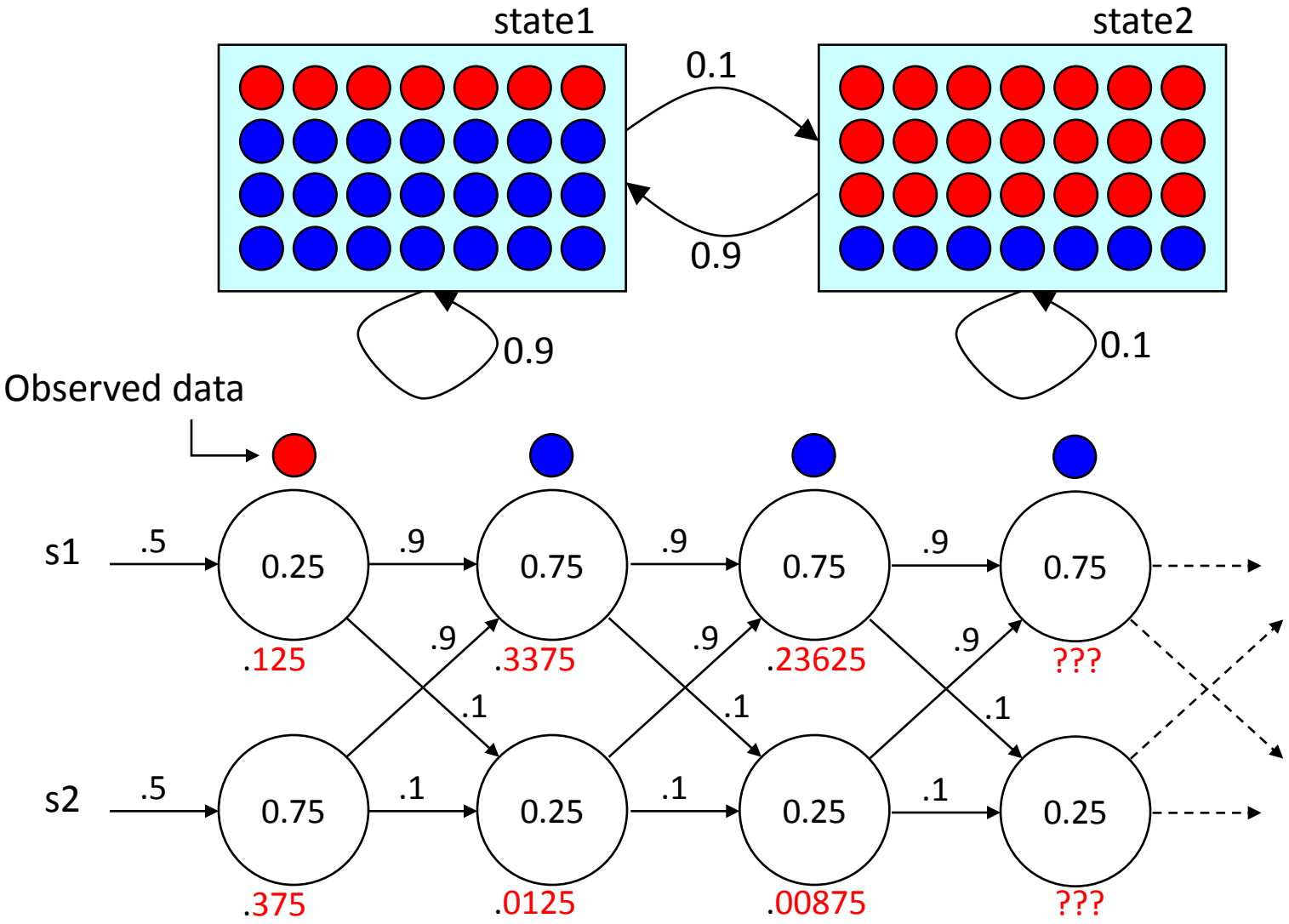
Hidden Markov Model



Hidden Markov Model



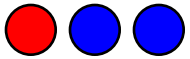
Hidden Markov Model



Hidden Markov Model

- What is going on here? TRELLIS!

- Verify by looking at all possible 3-long state sequences ending in s1, when input is:
  - s1 s1 s1: .5 * .25 * .9 * .75 * .9 * .75
  - s2 s1 s1: .5 * .75 * .9 * .75 * .9 * .75
  - s1 s2 s1: .5 * .25 * .1 * .25 * .9 * .75
  - s2 s2 s1: .5 * .75 * .1 * .25 * .9 * .75
  - Sum = 0.23625 (same as in trellis computation)



HMM: Forward Algorithm

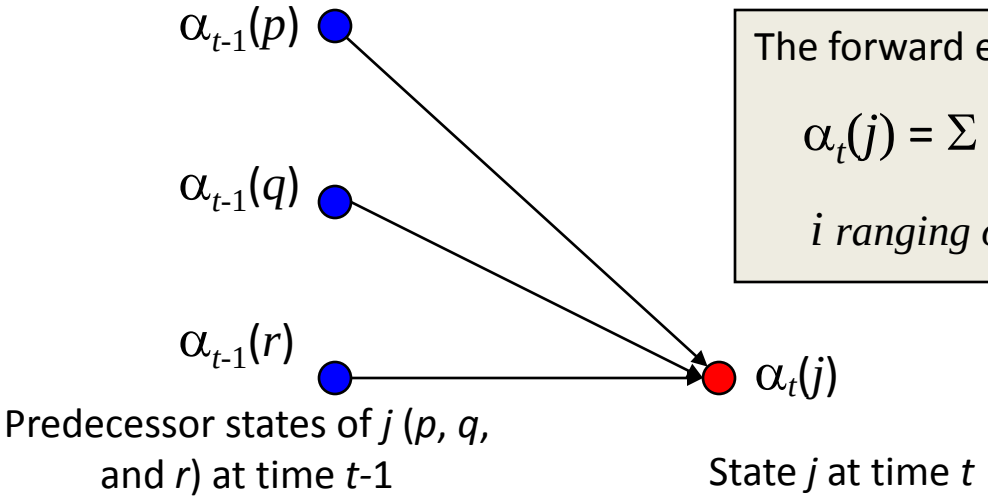
- We now have a way of matching an HMM and an input
  - Similar to matching a template and input

- Again, given input of length *N* observations:
  1. Consider all possible paths (state sequences) of length *N* through HMM, and ending in its final state
  2. Compute probability of each path (multiply together individual edge and local node probabilities)
  3. Sum all path probabilities

- Algorithmically:
  - Use a trellis once again, to avoid exponential explosion of considering all possible paths
  - This is called the *forward algorithm*

Forward Algorithm: $P(X \mid HMM)$

- Let us use $\alpha_t(j)$ to mean: the probability of observing the partial stream of observations $x_1, x_2, x_3 \ldots x_t$, and ending up at state j
  - It is the sum of the probabilities for all paths leading up to state j, while observing the partial sequence
- If we can define $\alpha_t(j)$ in terms of α_{t-1}(all predecessors of j), as in DP or DTW, we have an efficient solution:



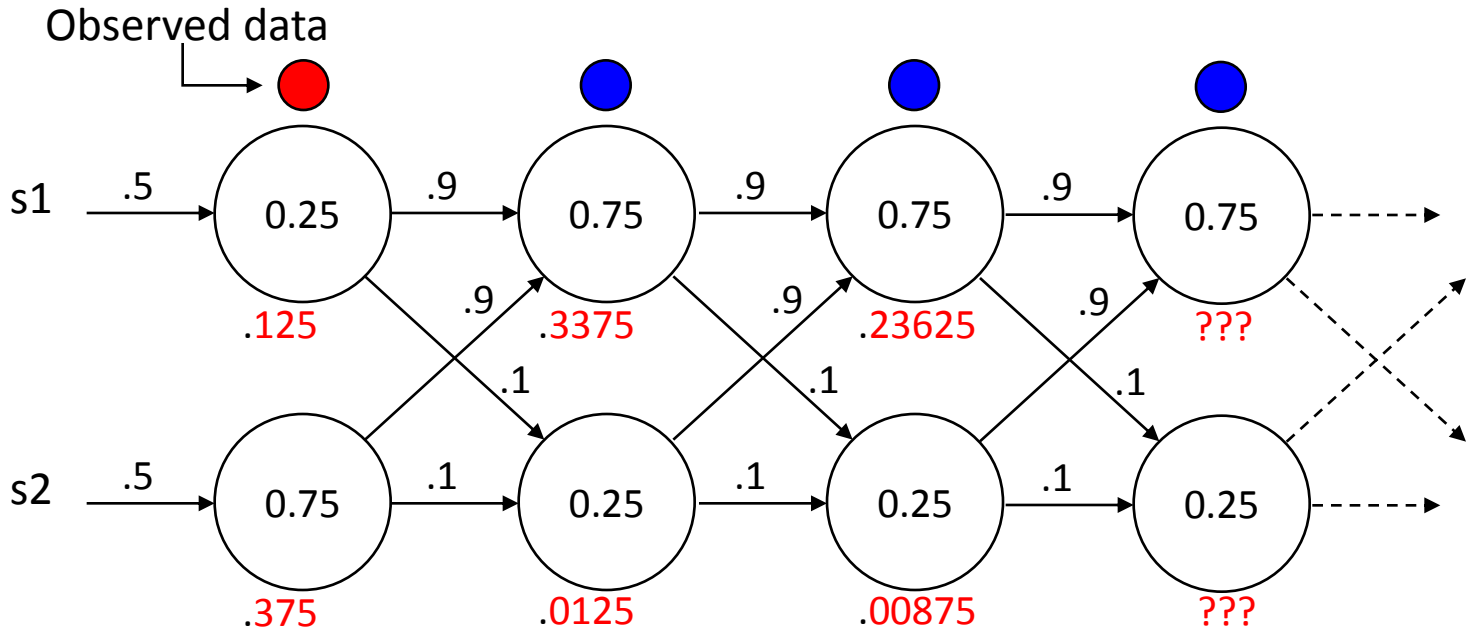
The forward equation:

$$\alpha_t(j) = \sum_i (\alpha_{t-1}(i)\, p(j|i)\, p(x_t|j))$$

i ranging over all predecessor states of j

Forward Algorithm: *P(X | HMM)*

- Review previous example:



The forward equation:

$$\alpha_t(j) = \sum_i (\alpha_{t-1}(i)\, p(j|i)\, p(x_t|j))$$

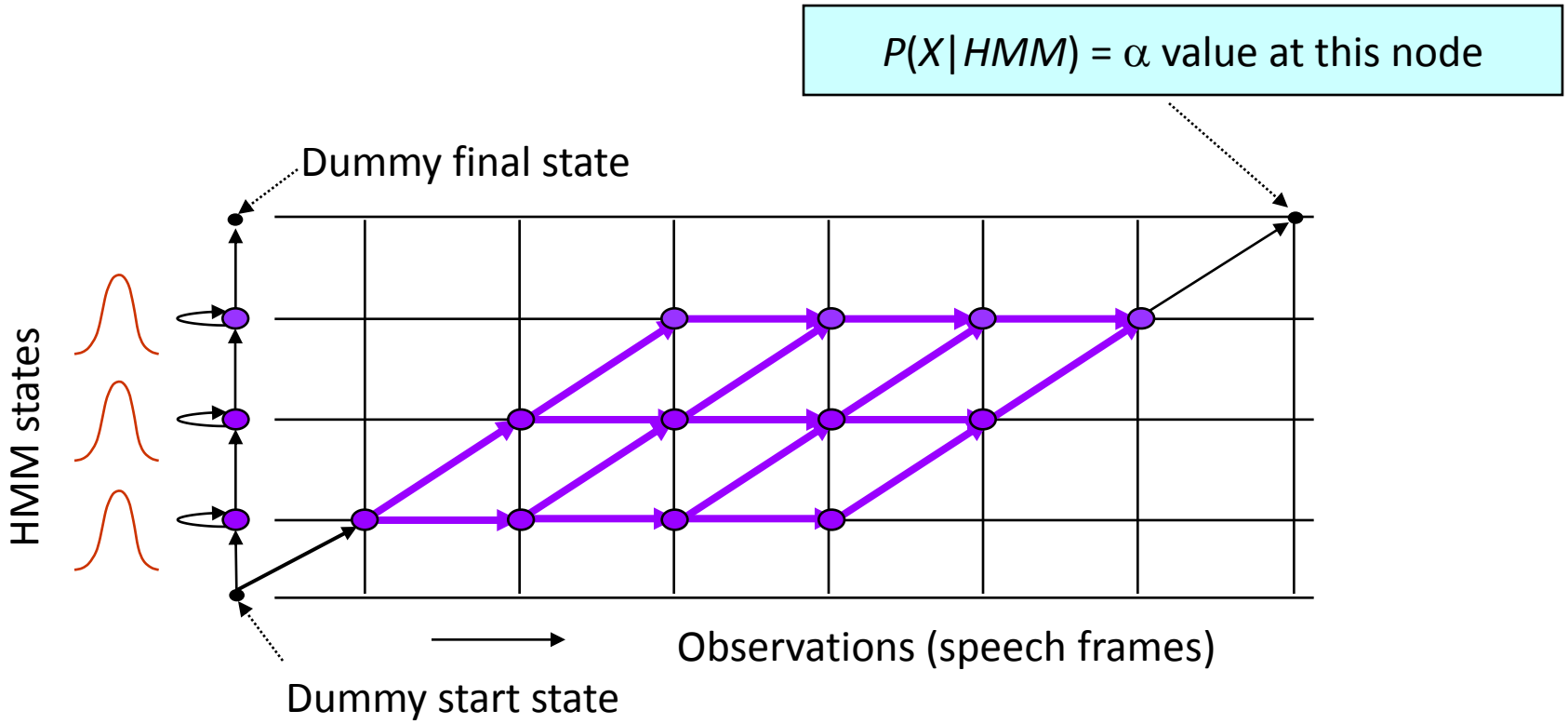
i ranging over all predecessor states of j

Forward Algorithm: *P*(*X* | *HMM*)

- Hence, *P*(*X* | *word*) can be computed using a trellis, somewhat similar to DP and DTW
  - The first difference: since this is a probabilistic model, the component edge and node probabilities are *multiplied*, rather being summed
  - The second difference: Partial path likelihoods arriving at a node are *summed*, rather than the max (or min) being chosen

Computing $P(X \mid HMM)$

- *Example*: Consider the following 3-state HMM and a 6-long input observation sequence:



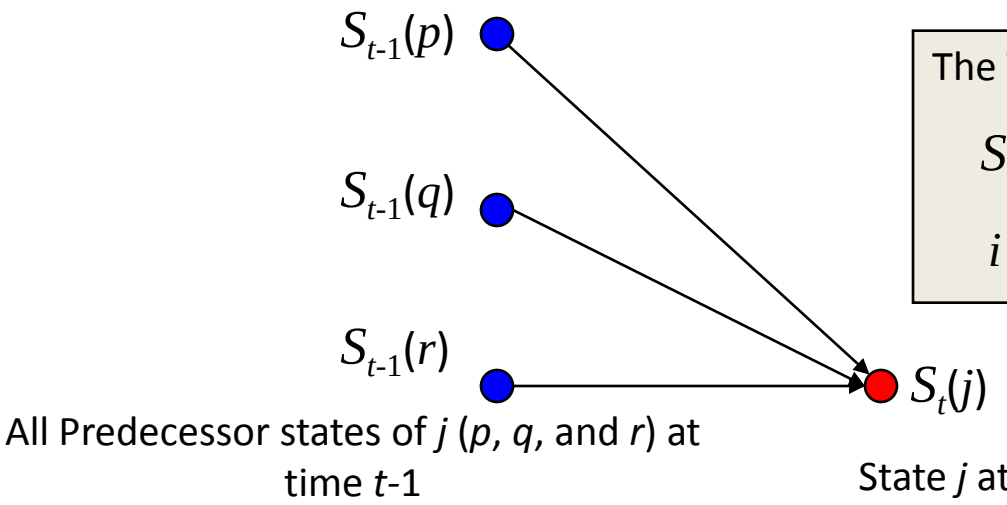
- All the possible paths in the blue network must be considered

Finding Best State Sequence

- Given an observation sequence, finding the most likely state sequence through an HMM is almost identical to DP
- The algorithm for this is called *Viterbi decoding*, after Andrew Viterbi
- In short:
  - Again, we have a trellis, with edges determined by the HMM structure and edge and local node likelihoods determined by the HMM state parameters and the input observations
  - Edge and node likelihoods are multiplied to obtain path likelihoods
  - At each node, we take the *max* of all incoming partial path likelihoods

Viterbi Decoding (contd.)

- Formally, let $S_t(j)$ mean: the likelihood of the best path that accounts for observations x_1, x_2, x_3 ... x_t, and ends up at state j
  - Note that $S_t(j)$ is the likelihood of a *single* path, a linear state sequence
  - $\alpha_t(j)$ was the likelihood summed over all paths leading up to state j
- Central idea in decoding the state sequence, the Viterbi equation:



The Viterbi equation:

$$S_t(j) = \max_i \left(S_{t-1}(i)\, p(j|i)\, p(x_t|j)\right)$$

i ranging over all predecessor states of j

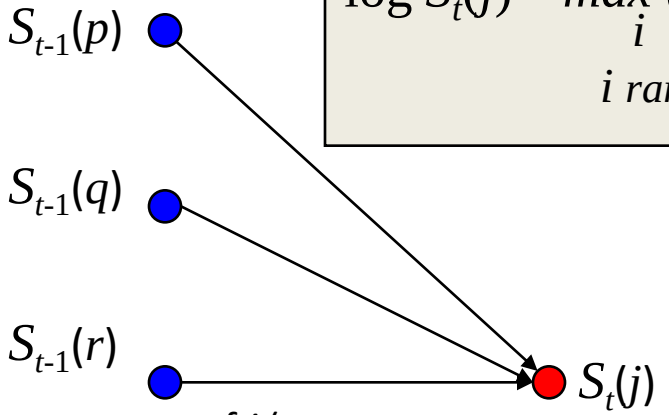
Viterbi Decoding (contd.)

- Most speech recognizers actually use *log-likelihoods*
  - Avoids the multiplications and exponentiations of the Gaussian function
- When using log-likelihoods, the Viterbi equation becomes:

The Viterbi equation:

$$\log S_t(j) = \max_i \left(\log S_{t-1}(i) + \log p(j|i) + \log p(x_t|j)\right)$$

i ranging over all predecessor states of j



All Predecessor states of *j* (*p*, *q*, and *r*) at time *t*-1

State *j* at time *t*

Detour: Costs, Probabilities & LogProbs…

- Trellis computation in DP/DTW/Viterbi:

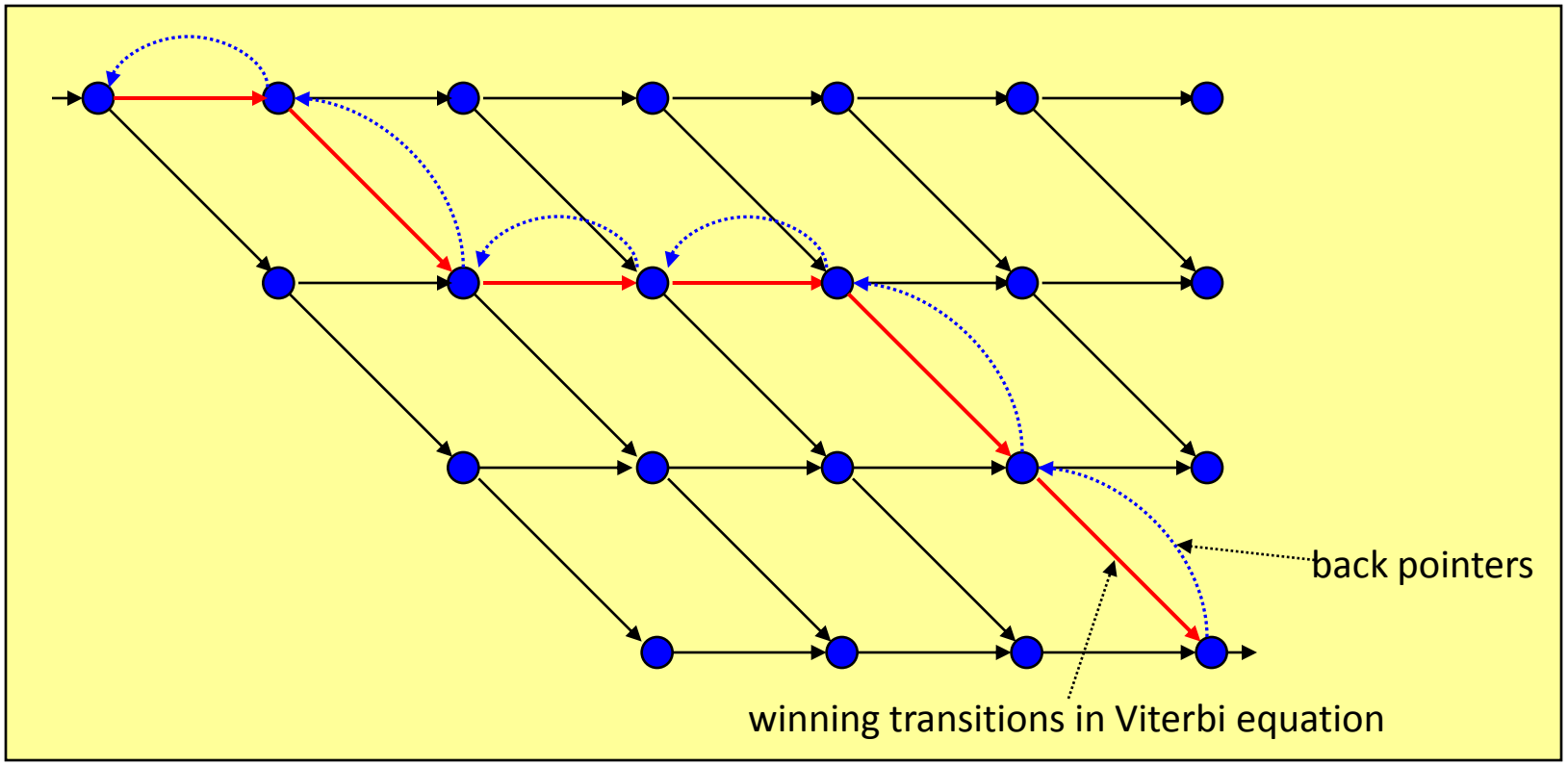
	Min/Max path cost?	Sum/multiply edge/node scores to get path score?
Distance/Cost	Min	Sum
-ve Distance/-ve Cost	Max	Sum
Probability/Likelihood	Max	Multiply
Log-Prob/Log-Likelihood	Max	Sum
-ve Log-Prob/-ve Log-Likelihood	Min	Sum

- In forward algorithm:

Probability/Likelihood	Sum	Multiply

Viterbi Decoding (contd.)

- To obtain the actual state sequence, when updating each node in the trellis, we maintain a *back-pointer* to its best predecessor
- In the end, we *trace back* from the final node of the lattice to determine the optimal HMM state sequence



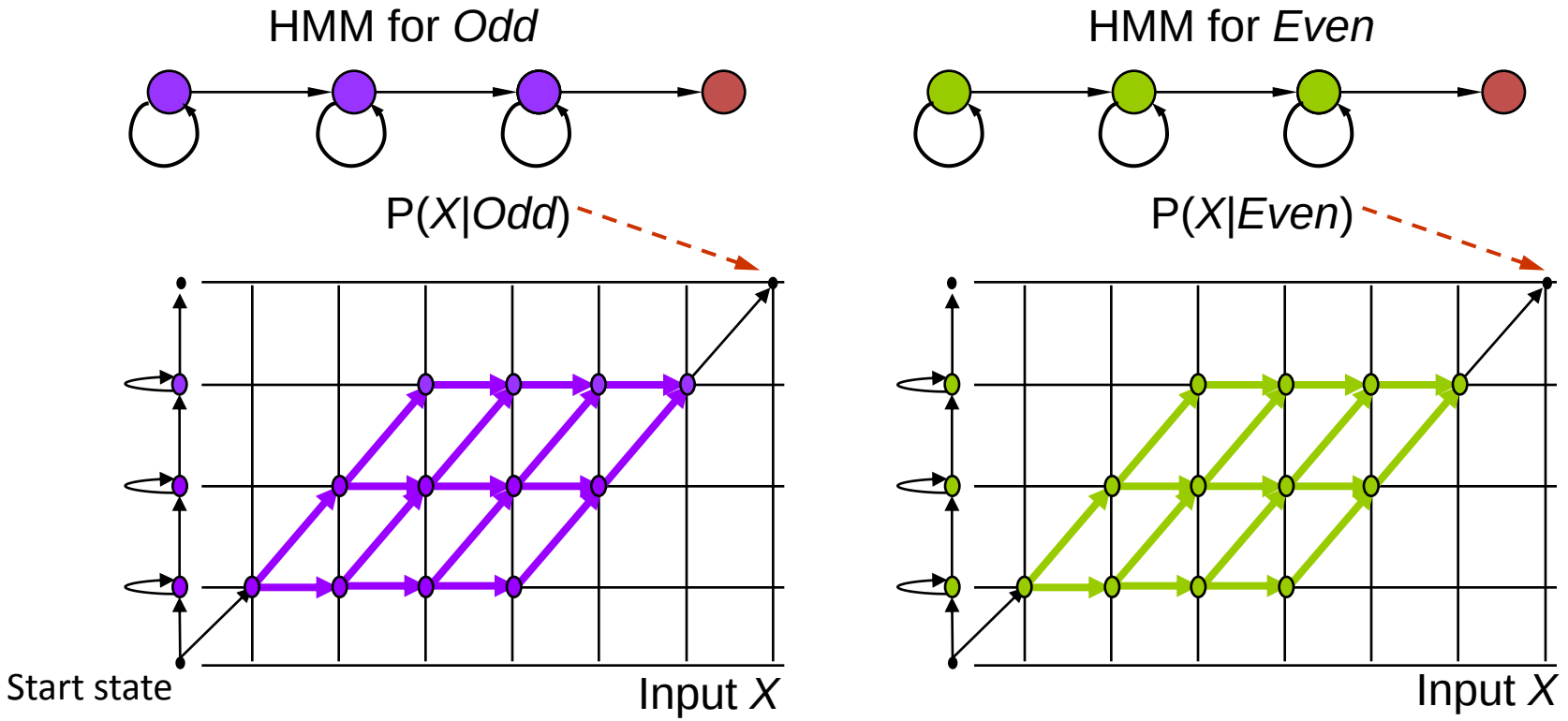
Time Synchronous Evaluation

- As we can easily imagine, both the forward and Viterbi algorithms can be executed in a time-synchronous manner, similar to DP/DTW
  - The advantages of such evaluation have already been seen

- Henceforth, we will assume time synchronous execution of either algorithm, unless otherwise stated

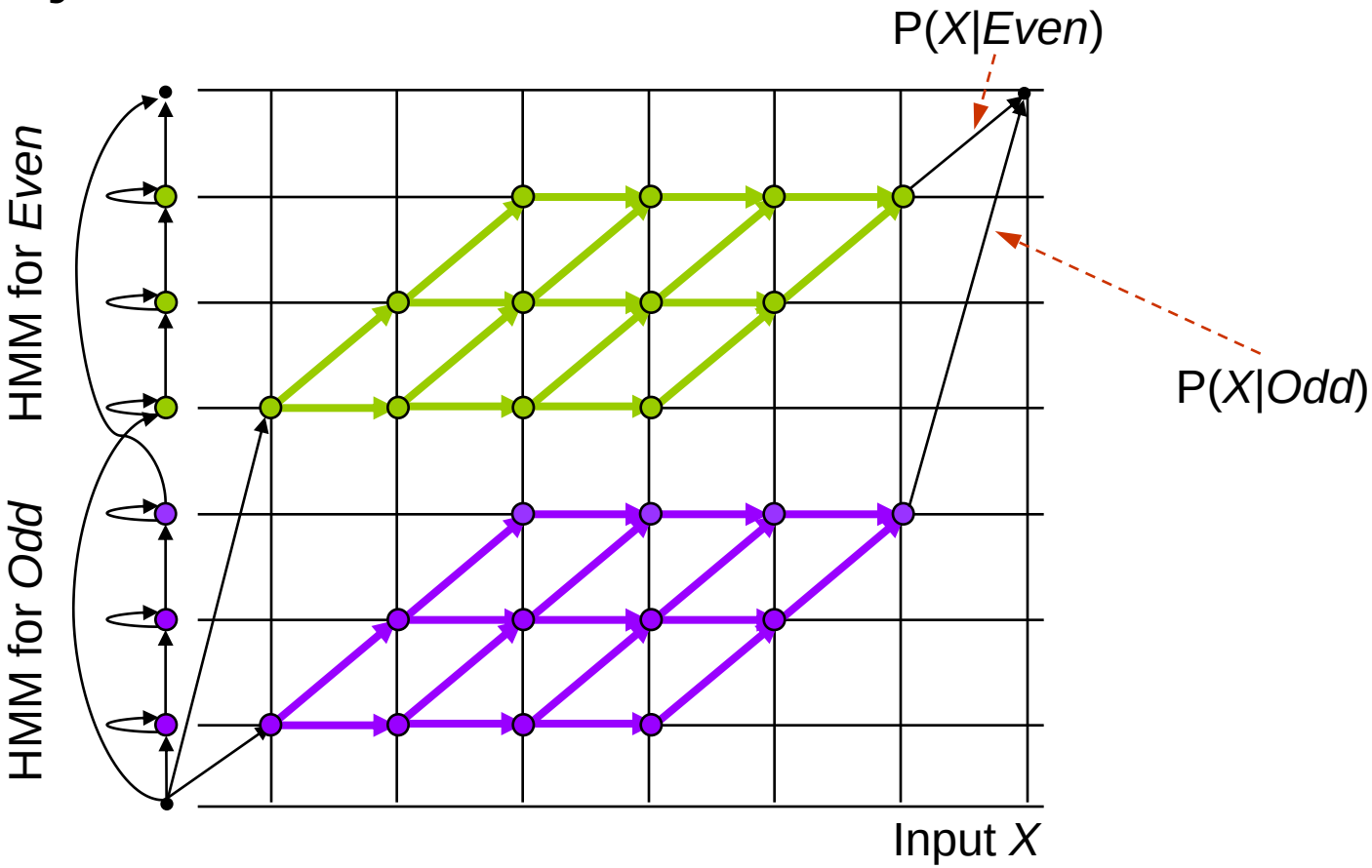
Isolated Word Recognition Using HMMs

- It should be straightforward to build an HMM-based isolated word recognizer at this point: *e.g.* 2 words: *Odd* and *Even*
  - Given input *X*, choose whichever has the higher forward likelihood



Isolated Word Recognition Using HMMs

- Time synchronous version:



Scaling Issues With HMMs

- Both the forward and the Viterbi algorithms compute long sequences of probabilities
- The accumulated path likelihoods can easily underflow any machine representation
- Two common ways of dealing with this problem:
  - *Scaling* path scores: At each frame, all the accumulated likelihoods at the trellis node are scaled (multiplied) by a fixed constant
    - Typically, such that the highest likelihood value after scaling is 1
    - It can be shown that such scaling does not affect the recognition
  - Using log-likelihoods instead of likelihoods
    - Multiplications are reduced to additions
    - However, adding log-likelihood path scores in the forward algorithm is tricky
      - Can use table look-up or other approximations
    - Ideal for the Viterbi algorithm that uses max instead of sum

Viterbi *vs* Forward Algorithm

- As just mentioned, it is easier to use log-likelihood values with Viterbi decoding
- We can obtain a best state-sequence (alignment) using Viterbi decoding
- In practice, with well trained HMMs, the best path usually dominates all else (likelihood-wise)
  - In such cases, it is acceptable to use the Viterbi likelihood as an approximation to the forward likelihood

Beam Search

- Not surprisingly, it is possible to use pruning techniques with either Viterbi or the forward algorithm
  - In particular, beam search
- When using likelihoods, the beam threshold is a multiplicative factor applied to the best score in the current frame
  - If the best scoring trellis node at the current time has path likelihood S, the pruning threshold is ST, where $T < 1$
  - Trellis nodes with path likelihoods < ST are pruned away
- When using log-likelihoods, it is an additive factor
  - If the best scoring trellis node at the current time has log-likelihood S, the pruning threshold is S-T, where $T > 0$
    - (Or, equivalently, $S+T$, where $T < 0$)
  - Trellis nodes with path log-likelihoods < S-T are pruned away

Incorporating Prior Knowledge

- Often we may have prior knowledge of the relative frequencies of the vocabulary words
  - *E.g.* Names on a cell phone; some names may be called much more frequently than others
  - Such knowledge is usually called *prior* knowledge
  - The known probabilities of names on cell phones are *prior* probabilities
    - Such information is usually available beforehand

- How can such information be used in speech recognition?
  - *E.g.* If two templates (HMMs) match the input equally well, the one with the higher prior probability should win
  - Is there a formalism that describes the optimal way of incorporating such knowledge in a speech recognition system?

Bayesian Classification

- Consider the problem of classifying a given input as belonging to one of several *classes*
  - In our case, the classes are the word HMMs
  - The input is the observed speech (utterance)

- In general:
  - Let the classes be C_1, C_2, C_3 …, and the input X
  - Assume we know $P(X|C_j)$ for all X and all j
    - These are the HMM forward probabilities
  - Assume also we know $P(C_j)$, the prior probability of each class
    - *i.e.*, the relative frequency of each word

- To maximize correctness, we want to maximize $P(C_j|X)$; *i.e.* identify the class C_j with highest probability of generating X

$= \text{argmax}_j\ (P(C_j/X))$

Bayesian Classification (contd.)

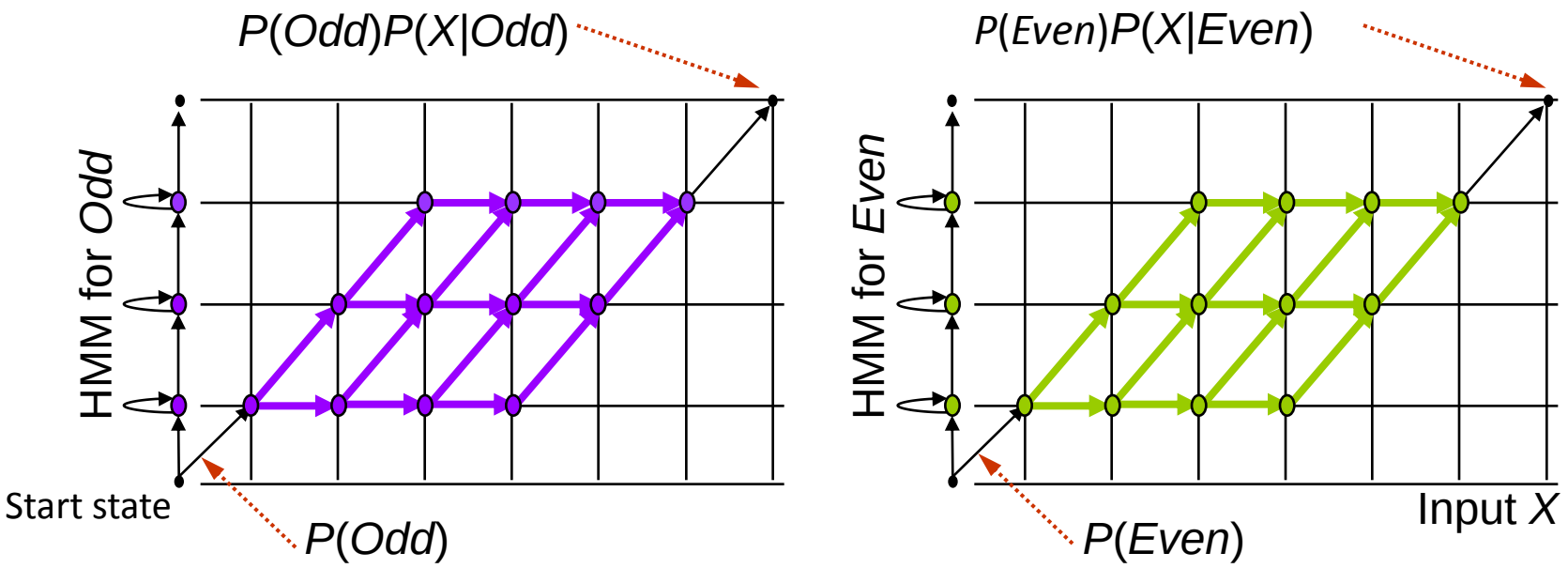
- By Bayes theorem:
  - $P(C_j|X) = P(C_j)P(X|C_j) / P(X)$
- Since we want to maximize $P(C_j|X)$ over all j, this is equivalent to maximizing $P(C_j)P(X|C_j)$
  - $P(X)$ is constant for (independent of) all C_j and can be ignored
- Since both $P(C_j)$ and $P(X|C_j)$ are known, we can identify the highest probability class generating X
- Thus, we get:

$$\text{argmax}_j\,(P(C_j|X)) = \text{argmax}_j\,(P(C_j)P(X|C_j))$$

  - This is the fundamental equation in speech recognition
- Since $P(C_j|X)$ combines both prior knowledge and the current input, it is called the *posterior probability* of C_j given X

Bayesian Classification (contd.)

- Thus, we can now incorporate prior knowledge into our forward and Viterbi algorithms:
  - Include $P(Odd)$ and $P(Even)$ as initial transitions to start states



- How do we do the same for the time-synchronous version

Isolated Words Based Dictation

- Should be a piece of cake..

General HMM State Distributions

- In the models considered so far, the state output probability distributions have been assumed to be Gaussian

- Actually, these distributions can be anything
  - The Gaussian is actually a rather coarse (smooth) representation
  - The actual distribution can have arbitrarily complex shape
  - If we model the output distributions of states better, we can expect the model to be a better representation of the data

- *Mixture* Gaussian distributions are good models for the distribution of classes of speech feature vectors

- Models can also be *simpler*
  - *E.g. discrete*, rather than *continuous valued*
  - Useful for systems with limited computational resources

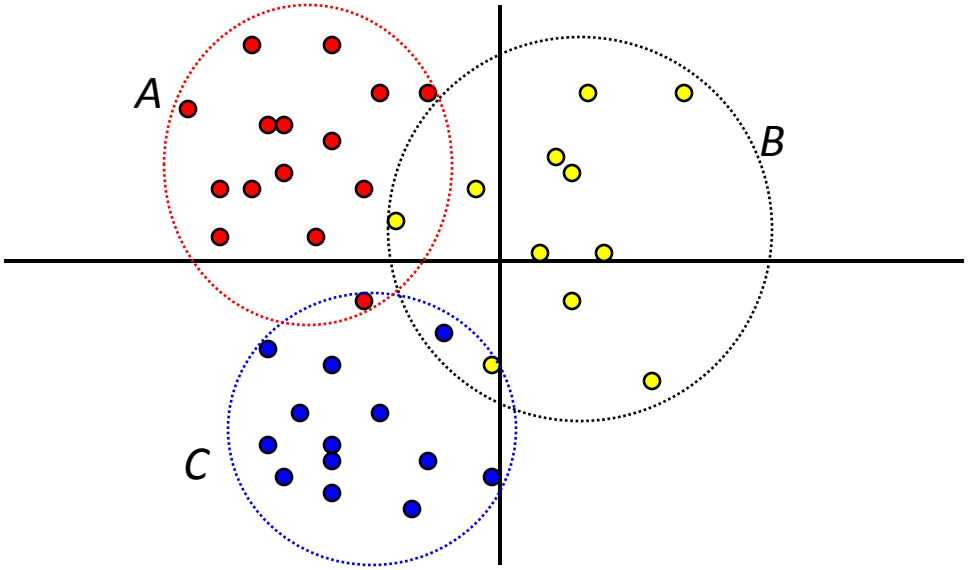
Discrete HMMs

- Discrete HMM systems are characterized by the following:
  - The observations are a finite set V of discrete *symbols*, $V = \{v_1, v_2, v_3, \ldots, v_N\}$
  - The state output probability functions are probability distributions over this set of symbols
    - *i.e.* $P_s(v_i)$, for state s, such that $\Sigma\, P_s(v_i) = 1$ (summed over s)
- The advantages are
  - The inputs can be scalar values (actually, just symbols) rather than real-valued vectors
  - The state output probability computation is dramatically simpler than evaluating a multi-dimensional Gaussian function
- The disadvantage is that such models may be too inaccurate for medium or large vocabulary systems

- Problem: speech is an inherently continuous valued stream
  - How can we use discrete HMMs to model speech?

Vector Quantization (VQ)

- A method of approximating real-valued vectors by a set of discrete symbols
- Consider the following 2-D example:



- Basic idea: group the data into a finite set of *clusters*, and replace each group by some single representative, often called its *centroid*
  - The discrete symbols are the cluster identities, *A*, *B*, and *C*
  - The set of representatives is called the *VQ codebook*

Vector Quantization (contd.)

- Several algorithms for VQ exist; *e.g. k-means clustering*

Discrete HMMs Using VQ Codebooks

- Once we have a VQ codebook, we can build discrete HMMs
  - Every input item (feature vectors) is quantized by finding its "closest" vector in the VQ codebook
  - The identity of this VQ codebook entry now becomes the observation
  - *E.g.* if we have a codebook with 256 entries, we can use an 8-bit codebook index values as the discrete observations
- The rest of the HMM formulation should be straightforward
- *Footnote*: Discrete HMMs are too inaccurate
  - Useful only for small devices with computational and memory limitations
  - Easy to estimate the resource requirements by examining the trellis and VQ codebook sizes

Complex HMM State Models

- We now consider the other side: HMM state output probability models that are *more* complex than simple Gaussian functions

Gaussian Mixtures

- A Gaussian Mixture is literally a mixture of Gaussians. It is a weighted combination of several Gaussian distributions

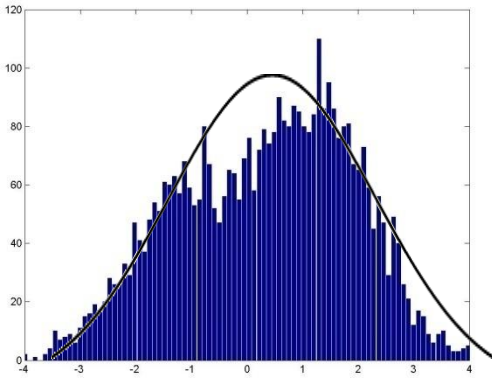
$$P(x) = \sum_{i=0}^{K-1} w_i Gaussian(x; m_i, C_i)$$

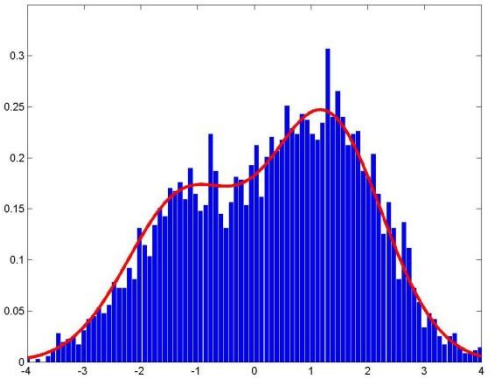
  - x is any data vector. P(x) is the probability given to that vector by the Gaussian mixture
  - K is the number of Gaussians being mixed
  - w_i is the mixture weight of the i^{th} Gaussian. m_i is its mean and C_i is its covariance

- The Gaussian mixture distribution is also a distribution
  - It is positive everywhere.
  - The total volume under a Gaussian mixture is 1.0.
  - Constraint: the mixture weights w_i must all be positive and sum to 1

Gaussian Mixtures

- A Gaussian mixture can represent data distributions far better than a simple Gaussian
- The two panels show the histogram of an unknown random variable
- The first panel shows how it is modeled by a simple Gaussian
- The second panel models the histogram by a mixture of two Gaussians
- Caveat: It is hard to know the optimal number of Gaussians in a mixture distribution for any random variable





The K-means algorithm

- The K-means algorithm is an iterative algorithm for clustering similar data from a data set
  - Where similarity is defined in terms of a user specified distance metric between clusters and data vectors
    - E.g. distance from the mean of the cluster
    - Negative log probability of the vector given by the distribution of the cluster
    - Distance from a linear regression for the cluster
- The goal of the algorithm is to cluster data such that the average distance between data vectors and their respective clusters is minimized
- The basic algorithm follows the following procedure:
  - Initialize all clusters somehow (the number of clusters is assumed)
  - For each training vector, find the closest cluster
  - Reassign training vectors to their closest clusters
  - Iterate the above two steps until the total distance of all training vectors from their clusters converges
    - Convergence can be proved for most distance measures

K-Means training Gaussian Mixtures

- The K-means algorithm can be used to estimate Gaussian mixture distributions for a data set
- Each of the K Gaussians is assumed to represent a separate cluster of the data
- The j^{th} cluster is characterized by
  - Its covariance C_j
  - Its mean vector m_j
  - A mixture weight w_j that specifies what portion of the total data belongs to that cluster
- Define the distance between a vector and the j^{th} cluster as

$$d(v, j) = -0.5\log\left(2\pi\left|C_j\right|\right) - 0.5(v - m_j)^T C_j^{-1}(v - m_j) - \log(w_i)$$

  - If the clusters are viewed as classes, the distance measure above is the log of the joint probability of the data vector and the class

K-Means training Gaussian Mixtures

1. Initialize means, covariances and mixture weights of all clusters
2. For each training vector *v*, find the cluster *j* for which d(v,j) is minimum
   - Let this cluster be *j*(*v*)
   - Mark *v* as belonging to *j*(*v*)
   - The distance for this vector is d(*v*,*j*(*v*))
3. Once all training vectors are clustered, re-estimate cluster means, covariances and mixture weights
4. If the sum of d(*v*,*j*(*v*)) for all vectors has converged, stop. Otherwise return to 2

- This algorithm minimizes the average distance of training vectors from their clusters. i.e., it maximizes the average of the log probability value given to data vectors by their cluster distributions

K-Means: Estimating parameters for a cluster

- The parameters for a cluster are its mixture weight, mean vector and covariance matrix. These are computed as follows:

$$m_j = \frac{1}{N_j} \sum_{v: j(v)=j} v$$

  - N_j is the number of vectors that have been tagged as belonging to cluster j
  - The summation is over all vectors who have been tagged as belonging to j

$$C_j = \frac{1}{N_j} \sum_{v: j(v)=j} (v - m_j)(v - m_j)^T$$

$$w_j = \frac{N_j}{N}$$

- N is the total number of training vectors for all clusters

K-Means for Gaussian Mixtures

- Initialization: There are several ways of initializing the K-means algorithm. One common method is:
  - Set $w_i = 1/K$ for all clusters
  - Set m_i = random({v}) : i.e. a randomly selected vector from the training data
  - Set C_j = the global covariance of the entire training data

- There are other ways of initializing the K-means algorithm
  - Some of these may be better than the initialization procedure described above, but are more complicated
  - Revisit later

The set of cluster means, variances and mixture weights constitute the parameters of the Gaussian mixture distribution for the data

$$P(v) = \sum_j w_j Gaussian(v; m_j, C_j)$$

HMMs with Gaussian mixture state distributions

- The parameters of an HMM with Gaussian mixture state distributions are:
  - π the set of initial state probabilities for all states
  - ***T*** the matrix of transition probabilities
  - A Gaussian mixture distribution for every state in the HMM. The Gaussian mixture for the i^{th} state is characterized by
    - K_i, the number of Gaussians in the mixture for the i^{th} state
    - The set of mixture weights $w_{i,j}$ $0<j<K_i$
    - The set of Gaussian means $m_{i,j}$ $0<j<K_i$
    - The set of Covariance matrices $C_{i,j}$ $0 < j <K_i$

Segmenting and scoring data sequences with HMMs with Gaussian mixture state distributions

- The procedure is identical to what is used when state distributions are Gaussians with one minor modification:

- The distance of any vector from a state is now the negative log of the probability given to the vector by the state distribution

- The "penalty" applied to any transition is the negative log of the corresponding transition probability

Modeling speech sounds with HMMs with Gaussian mixture state distributions

- When HMMs model speech sounds, the following structures have been found useful:
- The HMM must be entered from the first state
  - The initial state probability is 0 for all but the first state
- The HMM topology is left to right
  - All transition probabilities T_{ij} are 0 if $j < 1$
- Gaussians in the Gaussian mixtures are assumed to have diagonal covariance matrices
  - All off-diagonal terms are 0
  - This simplifies both, computation of probabilities and estimation of Gaussians

Summary of HMMs

- HMMs are a class of graphical models, consisting of states and transitions, for modeling time series data such as speech
- HMM states model the observed input data
  - We began by assuming a Gaussian distribution underlying each model
- HMM transitions model the time progression
- Their Markovian property allows elegant solutions to complex problems of model evaluation and estimation
- We have seen the three fundamental problems of HMMs
  - Evaluation of the model given some input data – Forward algorithm
  - Finding the best state sequence for some input data – Viterbi algorithm
- Looked at using log-likelihoods to simplify computation
- Adapting time synchronous beam search to HMM based decoding
- We developed the fundamental speech recognition equation to optimally incorporate prior knowledge into HMM based systems
- Finally, we looked at different types of HMM state output probability distributions: discrete and mixture Gaussians

Summary of HMMs (contd.)

- HMMs are a powerful mechanism for robust acoustic modeling over a wide range of system sizes
  - Small, medium, large vocabulary systems
- The framework is adaptable to using very small amounts of training data (*e.g.* a few samples of each word), to extremely large amounts (*e.g.* hundreds of hours of speech)
- The algorithms for training and decoding are elegant, with provable guarantees of optimality
  - The elegance provides simplicity of implementation and efficiency
  - The guarantee of optimality provides robustness; they do not break down in unexpected ways

- HMMs are used in speech recognition virtually universally

Summary of HMMs (contd.)

- Still, HMMs leave a lot of choices to the designer
  - The HMM structure: states and connectivity
  - What linguistic units to use HMMs for
    - We have considered word models, but we will see the use of phonetic models for large vocabulary recognition
  - The type of probability functions to be used as state models
    - Discrete, mixture Gaussians, others (shared mixture Gaussians)
    - For mixture Gaussians, the number of component Gaussians / mixture
    - For discrete systems, the size of VQ codebooks
  - Whether to use full covariances with Gaussians or simplify to diagonal ones

- The choices depend on the application, computational resources, and on the training data available
  - Experience with trying all varieties is invaluable in building expertise

- The design of HMM-based systems is still a bit of an art-form