

CS 1810 Spring 2026 Section 9

Autoregressive Models and Hidden Markov Models

1 Introduction

So far in CS 1810 we have studied two flavors of unsupervised learning: *clustering* (discovering discrete group structure) and *dimensionality reduction* (finding compact continuous representations). Today we turn to the third flavor: **generative modeling**, in which we learn the underlying probability distribution $p(\mathbf{x})$ of the data itself.

Why does generative modeling matter? Given a good model of $p(\mathbf{x})$ we can

- **Generate** new realistic samples $\mathbf{x} \sim p(\mathbf{x})$ (text, images, audio, proteins).
- **Evaluate likelihoods** to detect anomalies or score hypotheses.

We have already seen several generative approaches: VAEs (latent-variable models trained with the ELBO), diffusion models (iterative denoising), GANs (adversarial training), and Energy-Based Models (unnormalized densities). This section studies two more approaches to generative sequence modeling.

The first is **autoregressive (AR) models**, which factorize the joint distribution of a sequence into a product of conditional distributions and learn each conditional directly. The second is **Hidden Markov Models (HMMs)**, which extend the idea by introducing an *unobserved* (hidden) state sequence that evolves over time and generates the observations. Both models answer the question “how do I model a sequence?”, but from different angles: AR models encode dependence among observations directly, while HMMs posit that a latent process drives the observations.

2 Autoregressive Modeling

2.1 The Generative Modeling Goal

Suppose our data consists of sequences $\mathbf{x} = (x_1, x_2, \dots, x_T)$. The fundamental goal is to model the *full joint distribution* $p(x_1, x_2, \dots, x_T)$. This is useful both for evaluating the likelihood of any given sequence and for generating new ones.

Directly parametrizing this joint is hard for two reasons:

1. The number of possible sequences grows *combinatorially*: for a vocabulary of size $|V|$ and sequences of length T , there are $|V|^T$ possible sequences ($|V|=50,000$ and $T=100$ already gives $50,000^{100}$ paths).
2. Observations at different positions are *strongly dependent* on each other, so simple factored models (e.g. naïve Bayes) will not work.

2.2 The Chain Rule as a Generative Model

The key idea is to use the **chain rule of probability**, which gives an *exact* factorization of any joint distribution:

$$\begin{aligned} p(x_1, x_2, \dots, x_T) &= p(x_1) p(x_2 | x_1) p(x_3 | x_1, x_2) \cdots p(x_T | x_{<T}) \\ &= \prod_{t=1}^T p(x_t | x_{<t}), \end{aligned} \quad (1)$$

where $x_{<t} = (x_1, \dots, x_{t-1})$ is the *prefix* (also called the *context*).

An **autoregressive model** is one that exploits this factorization: instead of modeling the full joint directly, we learn a family of conditional distributions $p_\theta(x_t | x_{<t})$. The term “autoregressive” comes from classical statistics (e.g., the AR(p) time-series model $x_t = \sum_{i=1}^p \phi_i x_{t-i} + \varepsilon_t$), where “auto” means the regression uses previous values of the same variable.

The appeal of the AR decomposition is that a hard *joint* modeling problem becomes a sequence of *conditional prediction* problems: given the prefix $x_{<t}$ predict the next element x_t . Each step looks exactly like a supervised classification problem—the prefix is the input and the next token is the label—but crucially *no labels are required*: the data supervises itself.

2.3 Training by Maximum Likelihood

Given a dataset of N sequences $\{x_{1:T_i}^{(i)}\}_{i=1}^N$, we train p_θ by **maximizing the (log-)likelihood** of the observed data. For a single sequence, taking the log of the AR factorization gives:

$$\log p(x_{1:T}) = \log \prod_{t=1}^T p(x_t | x_{<t}) = \sum_{t=1}^T \log p(x_t | x_{<t}). \quad (2)$$

That is, the log-likelihood of a sequence is simply the sum of next-step log-probabilities. Extending this over the entire training dataset, we get the full objective and its equivalent minimization form:

$$\max_{\theta} \sum_{i=1}^N \log p_\theta(x_{1:T_i}^{(i)}) = \max_{\theta} \sum_{i=1}^N \sum_{t=1}^{T_i} \log p_\theta(x_t^{(i)} | x_{<t}^{(i)}) \quad (3)$$

The equivalent **minimization form** is the *negative log-likelihood* (NLL):

$$\min_{\theta} - \sum_{i=1}^N \sum_{t=1}^{T_i} \log p_\theta(x_t^{(i)} | x_{<t}^{(i)}) . \quad (4)$$

At each position t , the model outputs a distribution $p_\theta(x_t | x_{<t})$ over the vocabulary, and the true next token x_t^* acts as the target class. The loss at each step is therefore the **cross-entropy**:

$$\mathcal{L}_t = -\log p_\theta(x_t = x_t^* | x_{<t}). \quad (5)$$

Key takeaway: Maximum-likelihood AR training = cross-entropy on the next token, summed over all positions and all sequences.

2.3.1 Teacher Forcing and Exposure Bias

During training, we always feed the *true* previous tokens $x_1, \dots, x_{t-1}$ as context, even if the model predicted the wrong token at step $t-1$. This technique is called **teacher forcing**. It makes training stable and fully parallelizable (all positions can be computed in one forward pass), but it creates a mismatch with generation time:

- **Training:** every context comes from the ground-truth data.
- **Generation:** the model conditions on its *own* (possibly wrong) previous outputs.

This gap is called **exposure bias**: the model has never been trained on its own mistakes, so it may not know how to recover from them during generation.

2.4 Evaluation Metrics

Because AR models define an explicit joint probability $p(\mathbf{x}) = \prod_{t=1}^T p(x_t | x_{<t})$, they can be evaluated directly on held-out data via the average NLL. In practice, one of three normalized variants is reported:

- **Perplexity (PPL):** $\text{PPL} = \exp\left(-\frac{1}{T} \sum_{t=1}^T \log p(x_t | x_{<t})\right)$. The exponential of the average per-token NLL. Intuition: a PPL of K means the model is as uncertain as if it were choosing uniformly among K options at every step. Lower is better; PPL = 1 is perfect.
- **Bits-per-byte (BPB):** $\text{BPB} = -\frac{1}{B} \sum_{t=1}^T \log_2 p(x_t | x_{<t})$, where B is the byte count of the text string. Common in language modeling as a tokenizer-agnostic alternative to PPL. Lower is better; BPB = 0 is perfect.
- **Bits-per-dimension (BPD):** $\text{BPD} = -\frac{1}{D} \log_2 p(\mathbf{x})$, where D is the input dimension (e.g. $D = 3HW$ for an RGB image of height H and width W). Common in image modeling. Lower is better; BPD = 0 is perfect.

2.5 Generation (Sampling)

To *generate* a sequence from p_θ , we sample sequentially:

1. Sample $x_1 \sim p_\theta(x_1)$.
2. Sample $x_2 \sim p_\theta(x_2 | x_1)$.
3. Sample $x_3 \sim p_\theta(x_3 | x_1, x_2)$.
4. Continue until an end-of-sequence token is produced.

Generation is *inherently sequential*: each token requires the previously generated prefix. This is the main computational drawback of AR models: parallelism across token positions is not possible at generation time (though training is fully parallelizable).

Decoding strategies. Given the model's conditional distribution $p_\theta(x_t | x_{<t})$, we can choose x_t in several ways:

- **Greedy decoding:** $x_t = \arg \max_x p_\theta(x \mid x_{<t})$. Deterministic; good for tasks with unique correct answers (e.g. math), but often repetitive for open-ended generation.
- **Ancestral sampling:** $x_t \sim p_\theta(x \mid x_{<t})$. Stochastic; more diverse outputs.
- **Temperature sampling:** rescale logits by $1/\tau$ before softmax. Small $\tau < 1$ sharpens the distribution (more focused); large $\tau > 1$ flattens it (more random).
- **Top- k / nucleus sampling:** restrict sampling to the top- k tokens or the smallest set of tokens whose cumulative probability exceeds p (nucleus).

2.6 Neural Autoregressive Models

The AR factorization is a probabilistic framework that can be implemented with a variety of architectures. Any model that computes $p_\theta(x_t \mid x_{<t})$ is autoregressive. The choice of architecture determines how the context $x_{<t}$ is *summarized* into a representation h_t :

$$h_t = f_\theta(x_{<t}), \quad p_\theta(x_t \mid x_{<t}) = \text{softmax}(Wh_t + b). \quad (6)$$

RNNs. Recurrent neural networks maintain a hidden state that compresses the prefix recursively:

$$h_t = \sigma(W_{hh}h_{t-1} + W_{xe}(x_{t-1}) + b), \quad p_\theta(x_t \mid x_{<t}) = \text{softmax}(W_o h_t + b_o), \quad (7)$$

where $e(x_{t-1})$ is an embedding of the previous token. RNNs can in principle use an infinite context window, but in practice suffer from vanishing gradients and limited memory.

Transformers with causal masking. A standard (bidirectional) Transformer attends to all positions simultaneously, which would *break the AR constraint*: the representation at position t would depend on future tokens $x_{>t}$. The fix is **causal (masked) self-attention**: position t may only attend to positions $\leq t$. This is implemented by adding $-\infty$ to the upper-triangular entries of the attention score matrix before the softmax, causing those entries to become zero after normalization. With causal masking, all positions can still be computed in *parallel* during training (since the true tokens are known), but the AR constraint is enforced.

Transformers dominate modern AR language modeling because they (i) handle long-range dependencies without vanishing gradients, (ii) parallelize training across all sequence positions, and (iii) scale favorably with data and parameters.

2.7 Comparison of Generative Models

The table below situates AR models alongside the other generative model families studied in this course. HMMs will be introduced in Section ??; they are included here for completeness.

Family	Key Idea	Likelihood	Sampling	Key Strength
VAE	Probabilistic latent variables	Approx. (ELBO)	Easy	Learns useful latent codes
GAN	Adversarial learning	No explicit likelihood	Easy	Sharp, high-fidelity samples
EBM	Energy function	Intractable normalizer	Hard (MCMC)	Very flexible
AR	Chain-rule factorization	Exact & tractable	Sequential	Exact likelihood; natural for sequences
HMM	Latent Markov chain + emissions	Exact (forward alg.)	Sequential	Interpretable latent states; exact inference

2.8 Exercises

Exercise 2.1. Consider an autoregressive model over the alphabet $\{A, B\}$ with

$$p(x_1 = A) = 0.4, \quad p(x_1 = B) = 0.6,$$

and the following conditional probabilities:

$$p(x_2 = A \mid x_1 = A) = 0.3, \quad p(x_2 = B \mid x_1 = A) = 0.7,$$

$$p(x_2 = A \mid x_1 = B) = 0.8, \quad p(x_2 = B \mid x_1 = B) = 0.2,$$

$$p(x_3 = B \mid x_1 = B, x_2 = A) = 0.9, \quad p(x_3 = A \mid x_1 = A, x_2 = B) = 0.4.$$

- Compute $p(B, A, B)$.
- Compute $\log p(B, A, B)$.
- Compute the perplexity of the sequence (B, A, B) under this model.

Exercise 2.2. Autoregressive training typically uses *teacher forcing*: at each position, the model is given the true prefix $x_{<t}$ as input.

- Explain why teacher forcing makes the training objective easy to compute in parallel across positions.
- Suppose that during generation, the model makes an incorrect prediction at time t . Why can this error affect later predictions?
- A student says: “If teacher forcing works well during training, then generation should also work well, since the model has already learned the correct conditional distributions.” What important issue is this argument missing?

Exercise 2.3. For a sequence $x = (x_1, x_2, x_3, x_4)$, consider two autoregressive factorizations of

the same joint distribution:

$$p(x_1, x_2, x_3, x_4) = \prod_{t=1}^4 p(x_t \mid x_{<t})$$

and

$$p(x_1, x_2, x_3, x_4) = p(x_4) p(x_3 \mid x_4) p(x_2 \mid x_3, x_4) p(x_1 \mid x_2, x_3, x_4).$$

- (a) Why are both factorizations valid for the *true* data distribution?
- (b) In practice, if we train one neural model left-to-right and another right-to-left, should we expect them to assign exactly the same likelihood to every test sequence? Explain.
- (c) Give one reason why left-to-right ordering is especially natural for language, but less obviously correct for images.

3 Hidden Markov Models

3.1 Motivation: From Observed to Hidden Sequences

Autoregressive models capture statistical dependence *among observations* directly. But in many real-world settings the observations are not the whole story—there is an underlying process that we cannot see, and the observations are only noisy measurements of that process. For example:

- A patient's *true disease stage* is hidden; a clinician observes only symptoms and test results.
- A speaker's *intended phoneme* is hidden; a microphone records a continuous acoustic signal.
- The stock market's *true economic regime* (expansion, recession) is hidden; we see only prices.

In such settings, it is natural to posit a **latent state** z_t that evolves over time and generates the observation x_t . This is the core idea behind **Hidden Markov Models (HMMs)**.

HMMs vs. AR models. An AR model writes

$$p(x_{1:T}) = \prod_{t=1}^T p(x_t \mid x_{<t}),$$

capturing dependence among observations directly. An HMM instead writes

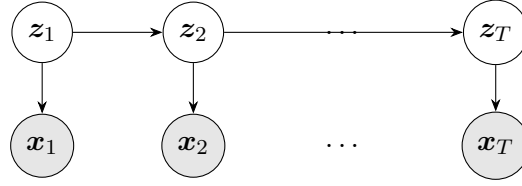
$$p(x_{1:T}) = \sum_{z_{1:T}} p(z_{1:T}, x_{1:T}),$$

marginalizing over all possible latent-state sequences. The joint factors as

$$p(z_{1:T}, x_{1:T}) = p(z_1) p(x_1 \mid z_1) \prod_{t=2}^T p(z_t \mid z_{t-1}) p(x_t \mid z_t),$$

so observations depend on each other *only through the hidden states*.

3.2 Graphical Model and Properties



We have a process that can be in one of K possible *states*, $C_1, \dots, C_K$, at each timestep. The state at a given timestep t is captured by the latent variable z_t . States cannot be observed, but they generate *observations* x_t which we can see. There are M possible values, $O_1, \dots, O_M$, that each observation can take on. The state the model is in at time $t + 1$ only depends on the state it was in at time t . A full HMM chain consists of T observations. We define a dataset in the context of HMMs as a collection of N chains: $\{\mathbf{x}^{(n)}\}_{n=1}^N$ where $\mathbf{x}^{(n)} = (x_1^{(n)}, \dots, x_T^{(n)})$.

The key properties of HMMs are:

- (*Markov property*) The next hidden state depends only on the current hidden state:

$$p(z_{t+1} \mid z_1, \dots, z_t, x_1, \dots, x_t) = p(z_{t+1} \mid z_t),$$

for $t = 1, 2, \dots, T - 1$.

- The current observation depends only on the current hidden state:

$$p(x_t \mid z_1, \dots, z_t, x_1, \dots, x_{t-1}) = p(x_t \mid z_t),$$

for $t = 1, 2, \dots, T$.

Both properties can be read off directly from the graphical model above.

3.3 Exercise: When to Use HMMs

Exercise 3.1. For each of the following scenarios, is it appropriate to use a Hidden Markov Model? Why or why not? What would the observed data be in each case, and what would the hidden states capture?

1. Stock market price data
2. Recommendations on a database of movie reviews
3. Daily precipitation data in Boston
4. Optical character recognition for identifying words

3.4 Parameterization

We can use the HMM assumptions to decompose the joint distribution of the hidden states and observations. Using the Markov property and the conditional independence of observations given states:

$$p(z_1, \dots, z_T) = p(z_1) \prod_{t=1}^{T-1} p(z_{t+1} \mid z_t),$$

and

$$p(\mathbf{x}_1, \dots, \mathbf{x}_T \mid \mathbf{z}_1, \dots, \mathbf{z}_T) = \prod_{t=1}^T p(\mathbf{x}_t \mid \mathbf{z}_t).$$

Combining these gives the full joint:

$$p(\mathbf{z}_1, \dots, \mathbf{z}_T, \mathbf{x}_1, \dots, \mathbf{x}_T) = p(\mathbf{z}_1) \prod_{t=1}^{T-1} p(\mathbf{z}_{t+1} \mid \mathbf{z}_t) \prod_{t=1}^T p(\mathbf{x}_t \mid \mathbf{z}_t). \quad (8)$$

From this decomposition, we see that three sets of parameters fully specify the HMM:

1. $\boldsymbol{\theta} \in \mathbb{R}^K$: defines the prior distribution over initial hidden states $\mathbf{z}_1$, corresponding to $p(\mathbf{z}_1)$.
2. $\mathbf{T} \in \mathbb{R}^{K \times K}$: **transition matrix**, where element T_{ij} is the probability of transitioning from latent state C_i to latent state C_j , corresponding to $p(\mathbf{z}_{t+1} \mid \mathbf{z}_t)$.
3. $\boldsymbol{\pi} \in \mathbb{R}^{K \times M}$: matrix of **emission probabilities**, where element π_{kl} is the probability of observing O_l given latent state C_k , corresponding to $p(\mathbf{x}_t \mid \mathbf{z}_t)$.

3.5 Using HMMs: Inference Tasks

The term *inference* here means any task where we feed observed data into the model and compute a quantity of interest. The major tasks are:

1. **Filtration**: $p(\mathbf{z}_t \mid \mathbf{x}_1, \dots, \mathbf{x}_t)$. What is the most likely state at the current time t given all observations so far?
2. **Smoothing**: $p(\mathbf{z}_t \mid \mathbf{x}_1, \dots, \mathbf{x}_T)$. What was the most likely state at time t given *all* observations (past and future)?
3. **Prediction**: $p(\mathbf{x}_{t+1} \mid \mathbf{x}_1, \dots, \mathbf{x}_t)$. What will be observed at the next timestep, given everything observed so far?
4. **Transition**: $p(\mathbf{z}_t, \mathbf{z}_{t+1} \mid \mathbf{x}_1, \dots, \mathbf{x}_T)$. What transition was most likely made from time t to $t+1$, given all observations?
5. **Joint of Observations**: $p(\mathbf{x}_1, \dots, \mathbf{x}_T)$. What is the likelihood of a particular observation sequence?
6. **Best Path**: $\max_{\mathbf{z}_1, \dots, \mathbf{z}_T} p(\mathbf{z}_1, \dots, \mathbf{z}_T \mid \mathbf{x}_1, \dots, \mathbf{x}_T)$. What is the most likely hidden state trajectory?

Computing these quantities naively by marginalizing over all latent-state sequences is exponentially expensive (K^T terms). We therefore use *variable elimination* and *dynamic programming* to compute them efficiently.

3.6 Forward-Backward Algorithm

The **forward-backward algorithm** computes two sets of intermediate values using dynamic programming: $\{\alpha_t(\mathbf{z}_t)\}_{t=1}^T$ (forward messages) and $\{\beta_t(\mathbf{z}_t)\}_{t=1}^T$ (backward messages).

Forward messages α_t . Define $\alpha_t(\mathbf{z}_t) = p(\mathbf{x}_1, \dots, \mathbf{x}_t, \mathbf{z}_t)$, the joint probability of the observation prefix $\mathbf{x}_1, \dots, \mathbf{x}_t$ and being in state $\mathbf{z}_t$ at time t .

Derivation:

$$\begin{aligned}
\alpha_t(\mathbf{z}_t) &= p(\mathbf{x}_1, \dots, \mathbf{x}_t, \mathbf{z}_t) \\
&= p(\mathbf{x}_t \mid \mathbf{z}_t, \mathbf{x}_1, \dots, \mathbf{x}_{t-1}) p(\mathbf{z}_t, \mathbf{x}_1, \dots, \mathbf{x}_{t-1}) && \text{(chain rule)} \\
&= p(\mathbf{x}_t \mid \mathbf{z}_t) p(\mathbf{z}_t, \mathbf{x}_1, \dots, \mathbf{x}_{t-1}) && \text{(obs. depends only on current state)} \\
&= p(\mathbf{x}_t \mid \mathbf{z}_t) \sum_{\mathbf{z}_{t-1}} p(\mathbf{z}_t, \mathbf{z}_{t-1}, \mathbf{x}_1, \dots, \mathbf{x}_{t-1}) && \text{(marginalize over } \mathbf{z}_{t-1}) \\
&= p(\mathbf{x}_t \mid \mathbf{z}_t) \sum_{\mathbf{z}_{t-1}} p(\mathbf{z}_t \mid \mathbf{z}_{t-1}) p(\mathbf{z}_{t-1}, \mathbf{x}_1, \dots, \mathbf{x}_{t-1}) && \text{(Markov property)} \\
&= p(\mathbf{x}_t \mid \mathbf{z}_t) \sum_{\mathbf{z}_{t-1}} p(\mathbf{z}_t \mid \mathbf{z}_{t-1}) \alpha_{t-1}(\mathbf{z}_{t-1}). && \text{(definition of } \alpha_{t-1})
\end{aligned}$$

This gives the recursion:

$$\alpha_t(\mathbf{z}_t) = \begin{cases} p(\mathbf{x}_1 \mid \mathbf{z}_1) p(\mathbf{z}_1) & t = 1 \\ p(\mathbf{x}_t \mid \mathbf{z}_t) \sum_{\mathbf{z}_{t-1}} p(\mathbf{z}_t \mid \mathbf{z}_{t-1}) \alpha_{t-1}(\mathbf{z}_{t-1}) & 1 < t \leq T. \end{cases}$$

Intuition: $\alpha_t(\mathbf{z}_t)$ encodes “how likely are we to be in state $\mathbf{z}_t$ and have seen the observations so far?” We move **forward** through the sequence.

Backward messages β_t . Define $\beta_t(\mathbf{z}_t) = p(\mathbf{x}_{t+1}, \dots, \mathbf{x}_T \mid \mathbf{z}_t)$, the probability of future observations conditioned on being in state $\mathbf{z}_t$ at time t .

Derivation:

$$\begin{aligned}
\beta_t(\mathbf{z}_t) &= p(\mathbf{x}_{t+1}, \dots, \mathbf{x}_T \mid \mathbf{z}_t) \\
&= \sum_{\mathbf{z}_{t+1}} p(\mathbf{x}_{t+1}, \dots, \mathbf{x}_T, \mathbf{z}_{t+1} \mid \mathbf{z}_t) && \text{(marginalize over } \mathbf{z}_{t+1}) \\
&= \sum_{\mathbf{z}_{t+1}} p(\mathbf{x}_{t+1} \mid \mathbf{z}_{t+1}) p(\mathbf{x}_{t+2}, \dots, \mathbf{x}_T \mid \mathbf{z}_{t+1}) p(\mathbf{z}_{t+1} \mid \mathbf{z}_t) && \text{(Markov \& emission assumptions)} \\
&= \sum_{\mathbf{z}_{t+1}} p(\mathbf{z}_{t+1} \mid \mathbf{z}_t) p(\mathbf{x}_{t+1} \mid \mathbf{z}_{t+1}) \beta_{t+1}(\mathbf{z}_{t+1}). && \text{(definition of } \beta_{t+1})
\end{aligned}$$

This gives the recursion:

$$\beta_t(\mathbf{z}_t) = \begin{cases} 1 & t = T \\ \sum_{\mathbf{z}_{t+1}} p(\mathbf{z}_{t+1} \mid \mathbf{z}_t) p(\mathbf{x}_{t+1} \mid \mathbf{z}_{t+1}) \beta_{t+1}(\mathbf{z}_{t+1}) & 1 \leq t < T. \end{cases}$$

Intuition: $\beta_t(\mathbf{z}_t)$ encodes “how well does the future evidence fit, if we are in state $\mathbf{z}_t$ now?” We move **backward** through the sequence.

Inference tasks via α and β . Once forward and backward messages are computed, all inference tasks reduce to simple expressions:

- **Filtration:** $p(z_t | x_1, \dots, x_t) \propto \alpha_t(z_t)$
- **Smoothing:** $p(z_t | x_1, \dots, x_T) \propto \alpha_t(z_t) \beta_t(z_t)$
- **Prediction:** $p(x_{T+1} | x_1, \dots, x_T) \propto \sum_{z_T, z_{T+1}} \alpha_T(z_T) p(z_{T+1} | z_T) p(x_{T+1} | z_{T+1})$
- **Transition:** $p(z_t, z_{t+1} | x_1, \dots, x_T) \propto \alpha_t(z_t) p(z_{t+1} | z_t) p(x_{t+1} | z_{t+1}) \beta_{t+1}(z_{t+1})$
- **Joint of Observations:** $p(x_1, \dots, x_T) = \sum_{z_t} \alpha_t(z_t) \beta_t(z_t)$, for any t .
- **Best Path:** Computed via the Viterbi algorithm (see below).

Viterbi Algorithm (Best Path). Finding the single most likely hidden-state sequence,

$$z^* = \arg \max_{z_{1:T}} p(z_{1:T} | x_{1:T}),$$

cannot be done by picking the most likely state independently at each time step; this would ignore correlations between states. Instead we use the **Viterbi algorithm**, which replaces the $\sum$ in the forward recursion with $\max$.

Define

$$\delta_t(z_t) = \max_{z_1, \dots, z_{t-1}} p(z_1, \dots, z_{t-1}, z_t, x_1, \dots, x_t),$$

the probability of the *best* partial path ending in state z_t at time t .

Forward pass (compute δ_t and backpointers):

$$\delta_t(z_t) = \begin{cases} p(x_1 | z_1) p(z_1) & t = 1 \\ p(x_t | z_t) \max_{z_{t-1}} [p(z_t | z_{t-1}) \delta_{t-1}(z_{t-1})] & 1 < t \leq T. \end{cases}$$

At each step, store the backpointer $z_{t-1}^\dagger(z_t) = \arg \max_{z_{t-1}} [p(z_t | z_{t-1}) \delta_{t-1}(z_{t-1})]$.

Backward pass (recover the best path):

$$z_T^* = \arg \max_{z_T} \delta_T(z_T), \quad z_{t-1}^* = z_{t-1}^\dagger(z_t^*).$$

The key distinction from the forward algorithm is: forward messages *sum* over all predecessor states; Viterbi *maximizes* over them.

3.7 Training HMMs with Expectation Maximization

Given data $\{x^{(n)}\}_{n=1}^N$, we want to learn the parameters $w = \{T, \theta, \pi\}$ by maximizing the marginal log-likelihood

$$\mathcal{L}(w) = \sum_{n=1}^N \log p(x^{(n)}; w) = \sum_{n=1}^N \log \sum_{z^{(n)}} p(x^{(n)}, z^{(n)}; w).$$

Why direct optimization is hard. If the hidden states $z^{(n)}$ were observed, the log-likelihood would decompose into a sum of terms, each depending on a single parameter, and MLE would reduce to counting. Because $z^{(n)}$ is *unobserved*, the sum over $z^{(n)}$ is inside the log, making direct maximization intractable.

The complete-data log-likelihood. If we *did* know the hidden states, we could write the *complete-data log-likelihood*:

$$\log p(\mathbf{x}^{(n)}, \mathbf{z}^{(n)}; \mathbf{w}) = \log p(z_1^{(n)}) + \sum_{t=1}^{T-1} \log p(z_{t+1}^{(n)} | z_t^{(n)}) + \sum_{t=1}^T \log p(\mathbf{x}_t^{(n)} | z_t^{(n)}).$$

This has a simple structure: each term depends on at most one or two consecutive states.

The EM idea: soft assignment of hidden states. Since the hidden states are unknown, we cannot maximize the complete-data log-likelihood directly. Instead, **Expectation Maximization (EM)** replaces the unknown hidden states with their *expected* values under the current parameter estimate. This is the key conceptual difference from, say, *K*-Means, which makes a *hard* assignment (each point belongs to exactly one cluster): EM makes a *soft* (probabilistic) assignment where each observation belongs to each state with some probability.

Concretely, EM iterates two steps:

3.7.1 E-Step: Computing the Expected Sufficient Statistics

Holding parameters $\mathbf{w}$ fixed, we compute the distribution over latent states given the observed data. This is exactly the inference we have already derived via the forward-backward algorithm. We introduce two proxy quantities:

1. **State occupancy** $q_{t,k}^{(n)}$: the posterior probability that the hidden state at time t in sequence n equals C_k :

$$q_{t,k}^{(n)} = p(z_t^{(n)} = C_k | \mathbf{x}^{(n)}; \mathbf{w}) \propto \alpha_t(C_k) \beta_t(C_k).$$

Here $\mathbf{q}^{(n)}$ is a $T \times K$ matrix, one row per timestep and one column per state. This is exactly the **smoothing** quantity from Section 3.5.

2. **Transition occupancy** $Q_{t,t+1,k,l}^{(n)}$: the posterior probability of transitioning from C_k to C_l between times t and $t+1$:

$$Q_{t,t+1,k,l}^{(n)} = p(z_t^{(n)} = C_k, z_{t+1}^{(n)} = C_l | \mathbf{x}^{(n)}; \mathbf{w}) \propto \alpha_t(C_k) p(C_l | C_k) p(\mathbf{x}_{t+1}^{(n)} | C_l) \beta_{t+1}(C_l).$$

This is exactly the **transition** quantity from Section 3.5.

Intuitively, $\mathbf{q}$ and $\mathbf{Q}$ are soft (probabilistic) guesses for the hidden states and transitions, analogous to the soft assignments r_{ij} in *K*-Means' E-step, but here derived from a proper probabilistic model.

3.7.2 M-Step: Updating the Parameters

Using the expected sufficient statistics from the E-step, we maximize the expected complete-data log-likelihood with respect to each parameter. Applying Lagrange multipliers (to enforce that probabilities sum to one) yields closed-form updates:

Initial state distribution:

$$\theta_k = \frac{\sum_{n=1}^N q_{1,k}^{(n)}}{N},$$

the average posterior probability of starting in state C_k .

Transition matrix:

$$T_{k,l} = \frac{\sum_{n=1}^N \sum_{t=1}^{T-1} Q_{t,t+1,k,l}^{(n)}}{\sum_{n=1}^N \sum_{t=1}^{T-1} q_{t,k}^{(n)}},$$

the expected number of $C_k \rightarrow C_l$ transitions divided by the expected number of times we are in C_k (excluding the last step).

Emission matrix:

$$\pi_{k,m} = \frac{\sum_{n=1}^N \sum_{t=1}^T q_{t,k}^{(n)} x_{t,m}^{(n)}}{\sum_{n=1}^N \sum_{t=1}^T q_{t,k}^{(n)}},$$

the expected fraction of time in state C_k that observation O_m is produced.

After updating all parameters, we repeat the E- and M-steps until convergence.

3.8 Exercise: Parameter Estimation in Supervised HMMs

Exercise 3.2. You are trying to predict the weather using an HMM. The hidden states are the weather of the day, which may be sunny or rainy, and the observable states are the color of the clouds, which can be white or gray. You have data on the weather and clouds from one sequence of four days (note: the hidden states are observed here):

Day	Weather	Clouds
1	Sunny	White
2	Rainy	Gray
3	Rainy	Gray
4	Sunny	Gray

1. Draw a graphical model representing the HMM.
2. Give the values of N, T, K and of the one-hot vectors $\mathbf{z}_1^{(1)}, \dots, \mathbf{z}_4^{(1)}, \mathbf{x}_1^{(1)}, \dots, \mathbf{x}_4^{(1)}$.

3. Estimate and interpret the values of the parameters θ, T, π using the M-step update equations from Section ??.

3.9 Exercise: Finding the Best Path

Exercise 3.3. Consider an HMM with hidden states {Rainy, Sunny} and observations {Walk, Shop, Clean}. The model parameters are

$$p(z_1) = \begin{cases} 0.6 & z_1 = \text{Rainy} \\ 0.4 & z_1 = \text{Sunny} \end{cases}, \quad p(z_t | z_{t-1}) = \begin{cases} 0.7 & (\text{Rainy} \rightarrow \text{Rainy}) \\ 0.3 & (\text{Rainy} \rightarrow \text{Sunny}) \\ 0.4 & (\text{Sunny} \rightarrow \text{Rainy}) \\ 0.6 & (\text{Sunny} \rightarrow \text{Sunny}) \end{cases}$$
$$p(x_t | z_t) = \begin{cases} 0.1 & (\text{Walk} | \text{Rainy}) \\ 0.4 & (\text{Shop} | \text{Rainy}) \\ 0.5 & (\text{Clean} | \text{Rainy}) \\ 0.6 & (\text{Walk} | \text{Sunny}) \\ 0.3 & (\text{Shop} | \text{Sunny}) \\ 0.1 & (\text{Clean} | \text{Sunny}) \end{cases}$$

Given the observation sequence $(x_1, x_2, x_3) = (\text{Walk}, \text{Shop}, \text{Clean})$, use the Viterbi algorithm to compute the best path (z_1^*, z_2^*, z_3^*) .