Reinforcement learning with sequence models

Pattern Recognition - Spring 2025

What is Reinforcement Learning with Sequence Models?

Reinforcement Learning (RL) with sequence models refers to the integration of sequence-based architectures (such as RNNs, LSTMs, GRUs, and Transformers) into RL algorithms. This combination is particularly useful for handling long-term dependencies, partial observability, and decision-making in sequential environments.

Partially Observable Environments (POMDPs):

- In many real-world tasks, the agent cannot fully observe the environment state at each timestep.
- Sequence models infer hidden states using past experiences.

Memory & Temporal Dependencies:

- Traditional RL algorithms like DQN rely on Markovian assumptions (i.e., the current state contains all necessary information). However, many environments (e.g., dialogue systems, robotic control, finance) require memory over long sequences of states and actions.
- RNNs, LSTMs, and Transformers help capture these dependencies.

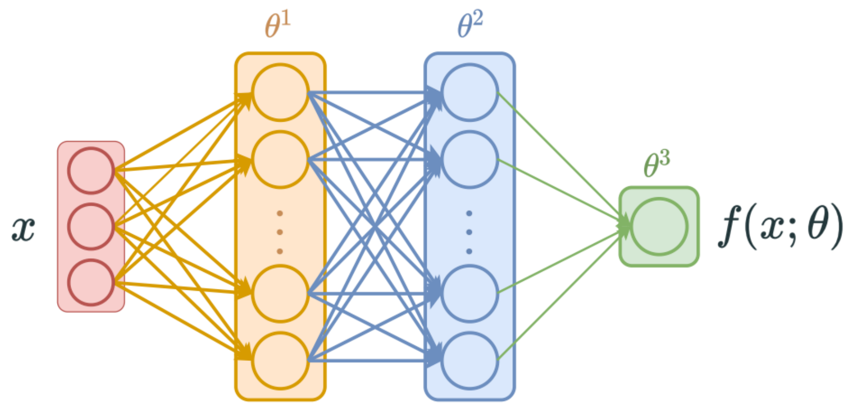
Offline & Sequence-based Decision Making:

- Some recent approaches model RL as a sequence modeling task, treating trajectories as tokenized sequences and using Transformers.

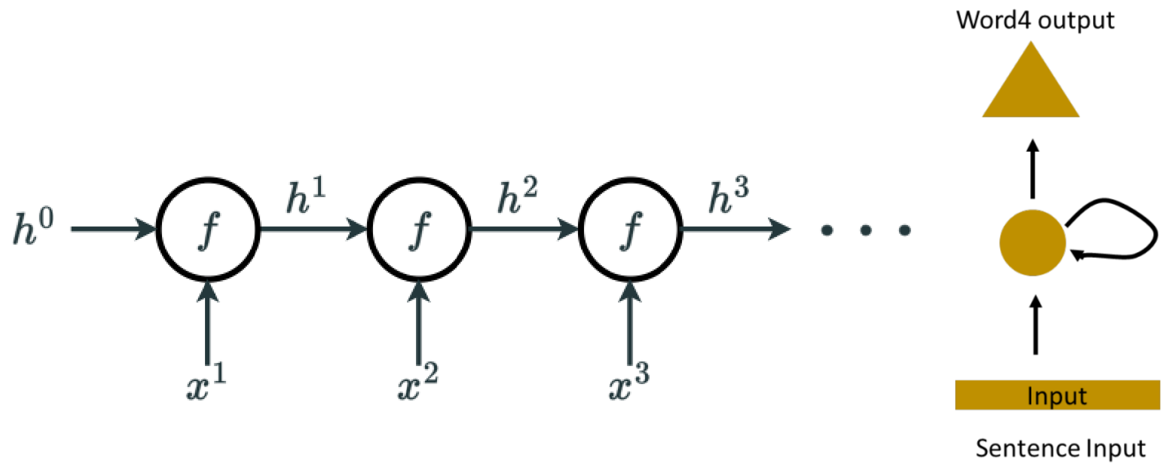
Review of sequence models

Recurrent neural networks

Recurrent neural networks (RNNs) are designed to process sequential data with several advantages: Learning sequences may require passing in the entire sequence as an input, which requires lots of parameters, It is also reasonable to assume a correlation between time points in a sequence, Sharing parameters might help pick up on recurring patterns in the data that occur at different timesteps.



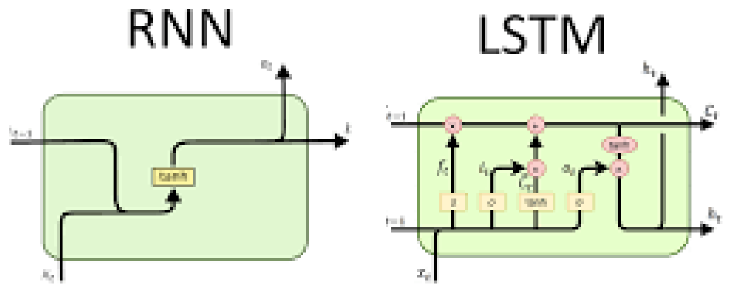
NN



RNN

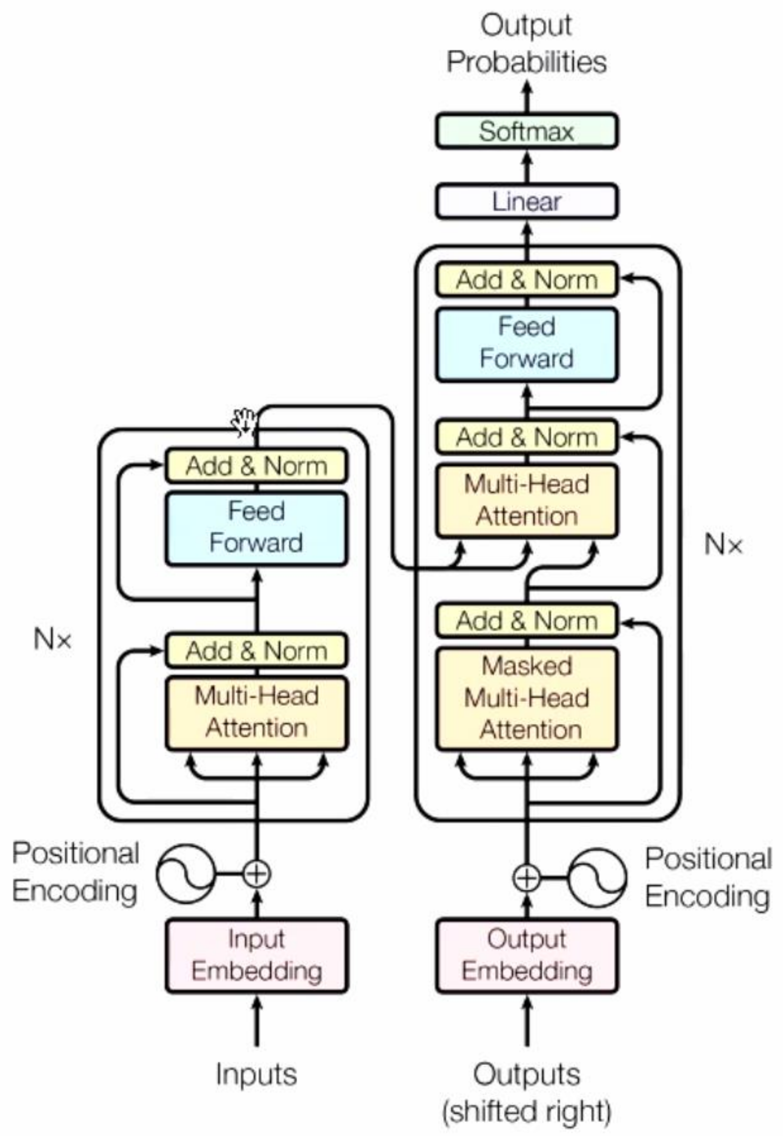
Long Short-Term Memory

Long Short-Term Memory (LSTM) networks are an advanced type of Recurrent Neural Network (RNN) designed to overcome the limitations of traditional RNNs, particularly their difficulty in learning long-term dependencies due to issues like vanishing and exploding gradients. While standard RNNs update their hidden state using a simple function of the current input and previous hidden state, LSTMs introduce a more complex memory cell along with gating mechanisms—input, forget, and output gates—that regulate the flow of information. This architecture enables LSTMs to selectively retain or discard information over long sequences.



Transformers

Transformers are a deep learning architecture that replaced recurrence with self-attention mechanisms to process sequential data more efficiently and effectively. Unlike RNNs and LSTMs, which handle sequences step-by-step, transformers process entire sequences in parallel, using self-attention to weigh the relevance of each word (or token) to every other word in the sequence. This allows them to model long-range dependencies without the sequential bottleneck, making training faster and more scalable. Transformers also incorporate positional encoding to retain information about the order of tokens. First introduced in the seminal paper *"Attention Is All You Need"*, transformers have since become the foundation for state-of-the-art models in NLP.



What are Variational Autoencoders

Latent variables are like the hidden ingredients in a secret recipe — they hold the key to the flavor of your model, but you can’t directly see them. In **latent variable models (LVMs)**, we use these hidden variables to capture complex patterns in data. In simple terms, **latent variables** are variables that you can’t measure or observe directly. However, they govern the behavior of observable data. For instance, in the case of generating text, the words you see on the screen are observable, but the underlying meaning or topic structure (which influences word choice) can be thought of as latent variables.

Variational Autoencoders (VAEs): These are a class of generative models that encode input data into a latent space and then decode from that space to generate new data. They work wonderfully in sequence generation tasks like text and music generation.

VAEs work using an **encoder-decoder architecture**:

Encoder: This part takes your input sequence (like a sentence) and compresses it into a lower-dimensional latent space. It doesn’t just encode any random data; it learns the most important, underlying patterns.

Decoder: After encoding, the decoder takes that compressed latent representation and tries to reconstruct the original sequence. Here’s where the magic happens: by tweaking the latent variables, the decoder can generate **new sequences** that still make sense because they follow the learned patterns.

What are Variational Autoencoders

Instead of encoding each input as a fixed point, VAEs encode it as a **probability distribution** — a range of possibilities. This allows the model to generate diverse outputs by sampling different points from this distribution.
Encoder: The encoder is an RNN (LSTM or GRU) that processes your sequence, one element at a time, and compresses it into a latent space.
Latent Variables: These are represented by two vectors — mean (mu) and log-variance (logvar). We sample from this latent space using a trick called the reparameterization trick, which allows us to backpropagate gradients and train the model.
Decoder: The decoder is another RNN that takes a latent vector (sampled from the latent space) and generates a new sequence. Because it's generating one element at a time, it captures the sequential dependencies.

The goal of training a VAE is to optimize something called the **Evidence Lower Bound (ELBO)**. The ELBO consists of two key terms:

1. **Reconstruction Loss**: This measures how well the decoder reconstructs the original sequence. You want the generated sequence to be as close as possible to the input.
2. **KL Divergence**: This term ensures that the latent space follows a standard normal distribution. Essentially, it forces the model to generalize better by keeping the latent space smooth and well-behaved.

Section 1

This section introduces how sequence models can be used in reinforcement learning (RL), particularly when dealing with partially observed Markov Decision Processes (POMDPs).

Traditional RL settings often assume fully observed environments, where the state fully captures the configuration of the system. However, in many real-world applications, agents only receive partial observations that do not satisfy the Markov property, leading to the need for sequence modeling.

From Fully Observed to Partially Observed MDPs

A Markovian state contains all the necessary information to predict the future. Once the current state is known, past states provide no additional predictive power.

Markov property: $P(S_{t+1} = s | S_t = s_t \ , S_{t-1} = s_{t-1} \ , \ldots, S_1 = s_1) = \ P(S_{t+1} = s | S_t = s_t)$

In partially Observed MDPs (POMDPs), observations do not contain full state information. Past observations may be required to infer the current state. Examples:

- Cheetah chasing a gazelle: Image observations may not fully convey the true state (e.g., position, momentum).
- Self-driving cars: Vehicles in blind spots cause crucial information to be missing.
- Dialogue systems: A single message lacks full context; history is essential.

Formally, a POMDP is a 7-tuple (S, A, T, R, Ω, O) where Ω is the set of observations and O is a probability distribution p(o| s, a).

All real-world problems are partially observed to some degree. In some cases, partial observability is minor, so current observations can act as a proxy for state (e.g., Atari games).

Unique Phenomena in POMDPs

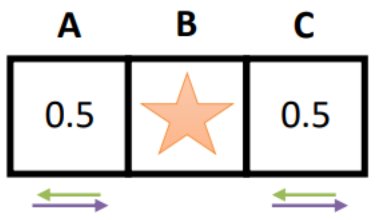
1) Information-Gathering Actions

In POMDPs, actions may be taken purely to gain information, not immediate rewards. Example: Peeking over a maze to understand its layout is optimal even if it doesn't move the agent closer to the goal.

2) Stochastic Optimal Policies

POMDPs may require stochastic policies to maximize expected reward. Example:

- 3-state MDP with states A, B, C; reward only at B.
- Starting in A or C with 0.5 probability each.
- Observation gives no info; deterministic policy (always left or right) may fail.
- Stochastic policy (50/50 split) outperforms any deterministic one.



Can Existing RL Algorithms Handle POMDPs?

We examine whether standard RL methods can “handle” POMDPs correctly.

Policy Gradient Methods:

Policy Gradient Estimator (∇ log π) is valid even without the Markov property.

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(a_{i,t}\middle|s_{i,t}\right)A(s_{i,t},a_{i,t}) \xrightarrow{?} \nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(a_{i,t}\middle|o_{i,t}\right)A(o_{i,t},a_{i,t})$$

Next we will show that the following formula does not use Markov property.

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta\left(a_{i,t}\middle|s_{i,t}\right)\sum_{t=1}^{T} r(s_{i,t},a_{i,t})$$

Proof That Reward To Go Does Not Use Markov Property

$$\theta^\star = \arg\max_\theta \underbrace{E_{\tau\sim p_\theta(\tau)}\left[\sum_t r(\mathbf{s}_t, \mathbf{a}_t)\right]}_{J(\theta)}$$

a convenient identity

$$\underline{p_\theta(\tau)\nabla_\theta \log p_\theta(\tau)} = p_\theta(\tau)\frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \underline{\nabla_\theta p_\theta(\tau)}$$

$$J(\theta) = E_{\tau\sim p_\theta(\tau)}[\underbrace{r(\tau)}_{\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t)}] = \int p_\theta(\tau) r(\tau) d\tau$$

$$\nabla_\theta J(\theta) = \int \underline{\nabla_\theta p_\theta(\tau)} r(\tau) d\tau = \int \underline{p_\theta(\tau)\nabla_\theta \log p_\theta(\tau)} r(\tau) d\tau = E_{\tau\sim p_\theta(\tau)}[\nabla_\theta \log p_\theta(\tau) r(\tau)]$$

Proof That Reward To Go Does Not Use Markov Property

$$\theta^\star = \arg\max_\theta J(\theta)$$

$$J(\theta) = E_{\tau\sim p_\theta(\tau)}[r(\tau)]$$

$$\nabla_\theta J(\theta) = E_{\tau\sim p_\theta(\tau)}[\nabla_\theta \underline{\log p_\theta(\tau)} r(\tau)]$$

$$\underbrace{p_\theta(\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T)}_{p_\theta(\tau)} = p(\mathbf{s}_1)\prod_{t=1}^{T}\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$$

log of both sides

$$\underline{\log p_\theta(\tau) = \log p(\mathbf{s}_1) + \sum_{t=1}^{T}\log\pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \log p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)}$$

$$\nabla_\theta\left[\cancel{\log p(\mathbf{s}_1)} + \sum_{t=1}^{T}\log\pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \cancel{\log p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)}\right]$$

$$\nabla_\theta J(\theta) = E_{\tau\sim p_\theta(\tau)}\left[\left(\sum_{t=1}^{T}\nabla_\theta\log\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\right)\left(\sum_{t=1}^{T}r(\mathbf{s}_t, \mathbf{a}_t)\right)\right]$$

Proof That Reward To Go Does Not Use Markov Property

For events $A_1, \ldots, A_n$ whose intersection has not probability zero, the chain rule states

$$\begin{aligned}
\mathbb{P}\left(A_1 \cap A_2 \cap \ldots \cap A_n\right) &= \mathbb{P}\left(A_n \mid A_1 \cap \ldots \cap A_{n-1}\right) \mathbb{P}\left(A_1 \cap \ldots \cap A_{n-1}\right) \\
&= \mathbb{P}\left(A_n \mid A_1 \cap \ldots \cap A_{n-1}\right) \mathbb{P}\left(A_{n-1} \mid A_1 \cap \ldots \cap A_{n-2}\right) \mathbb{P}\left(A_1 \cap \ldots \cap A_{n-2}\right) \\
&= \mathbb{P}\left(A_n \mid A_1 \cap \ldots \cap A_{n-1}\right) \mathbb{P}\left(A_{n-1} \mid A_1 \cap \ldots \cap A_{n-2}\right) \cdot \ldots \cdot \mathbb{P}(A_3 \mid A_1 \cap A_2) \mathbb{P}(A_2 \mid A_1) \mathbb{P}(A_1) \\
&= \mathbb{P}(A_1) \mathbb{P}(A_2 \mid A_1) \mathbb{P}(A_3 \mid A_1 \cap A_2) \cdot \ldots \cdot \mathbb{P}(A_n \mid A_1 \cap \cdots \cap A_{n-1}) \\
&= \prod_{k=1}^{n} \mathbb{P}(A_k \mid A_1 \cap \cdots \cap A_{k-1}) \\
&= \prod_{k=1}^{n} \mathbb{P}\left(A_k \,\middle|\, \bigcap_{j=1}^{k-1} A_j\right).
\end{aligned}$$

$$\underbrace{p_\theta(\mathbf{s}_1, \mathbf{a}_1, \ldots, \mathbf{s}_T, \mathbf{a}_T)}_{p_\theta(\tau)} = p(\mathbf{s}_1) \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t | \mathbf{s}_t) p(\mathbf{s}_{t+1} | \mathbf{s}_t, \mathbf{a}_t)$$

What Else Is Ok For $\mathrm{A}(s_{\mathrm{i,t}}, \mathrm{a}_{\mathrm{i,t}})$?

Causality trick (using rewards from t to T) is valid under POMDPs, because policy at time step t’ cannot effect reward at timestep t when t<t’.

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}[\nabla_\theta \log \pi_\theta(a_{i,t}|o_{i,t}) \sum_{t'=t}^{T} r(o_{i,t}, a_{i,t})]$$

Observation-based baselines are unbiased, though potentially less effective at variance reduction.

$$E[\nabla_\theta \log p_\theta(\tau) b] = \int p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) b\, d\tau = \int \nabla_\theta p_\theta(\tau) b\, d\tau = b\nabla_\theta \int p_\theta(\tau) d\tau = b\nabla_\theta 1 = 0$$

Using value approximators $V(o_t)$ is invalid because observations don't fully determine state.

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t})(r_{i,t} + \gamma V(o_{i,t+1}) - V(o_{i,t}))$$

Other Approaches

Value-Based Methods (e.g., Q-Learning)

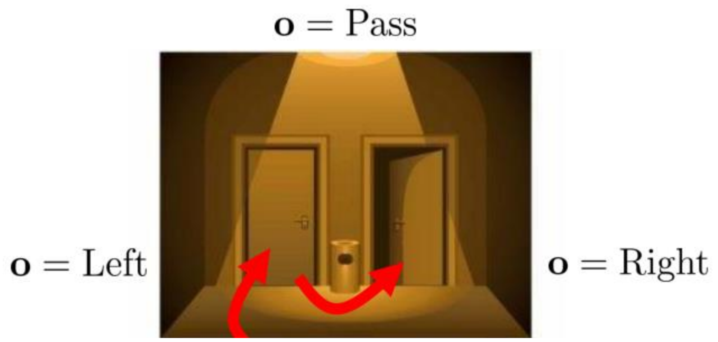
Q-learning assumes consistent value per state regardless of trajectory.

Observations violate this assumption, so Q(o, a) is not valid.

Deterministic policy extraction from Q-values fails in POMDPs, where stochastic policies may be optimal.

Model-Based Methods

Predictive models trained on (o, a) → o' are incorrect.


$\mathbf{o} = \text{Pass}$
$\mathbf{o} = \text{Left}$
$\mathbf{o} = \text{Right}$

Example:

- Two doors, one locked at random.
- Naively learned model would estimate a 0.5 success chance.
- In reality, retrying a locked door will always fail.
- The model incorrectly assumes I.I.D. transitions, which ignores crucial history (e.g., door state already tested).

Toward Better Handling of Partial Observability

Using Observation History as State

Define $s_t = (o_1, o_2, \ldots, o_t)$. This sequence obeys the Markov property because S_t contains S_{t-1}, so the future is conditionally independent of past beyond S_t:

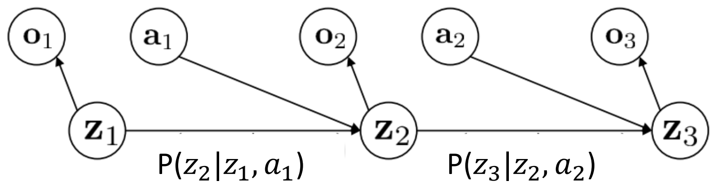
$$P(S_{t+1} = s | S_t = s_t \ , S_{t-1} = s_{t-1} \ , \ldots, S_1 = s_1) = \ P(S_{t+1} = s | S_t = s_t)$$

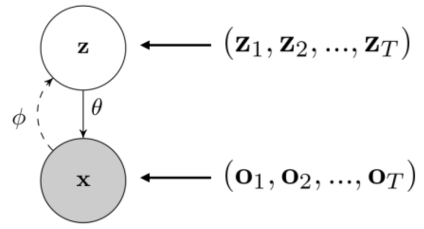
Valid to use Q(o_1,..., o_t, a) but must use sequence models to represent these long histories.

Using Learned Latent States (State-Space Models)

Sequence VAEs can model latent variables Z_t with Markovian dynamics from observation sequences. Trained with:

- Encoder: history → latent state Z_t




$\mathbf{z}$
$\mathbf{x}$
θ
ϕ
$(\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_T)$
$(\mathbf{o}_1, \mathbf{o}_2, \ldots, \mathbf{o}_T)$

Once trained, Z_t can be used in place of unobservable MDP states.

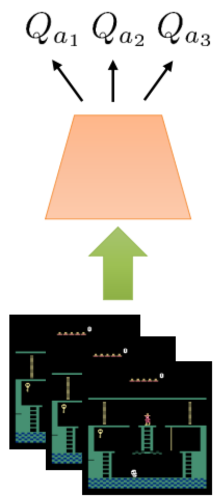
Implementation of Sequence-Based RL

1) Short History Approach

Concatenation of fixed-length window of past observations (e.g., last 4 frames in Atari).

Works well in mildly partially observable settings.

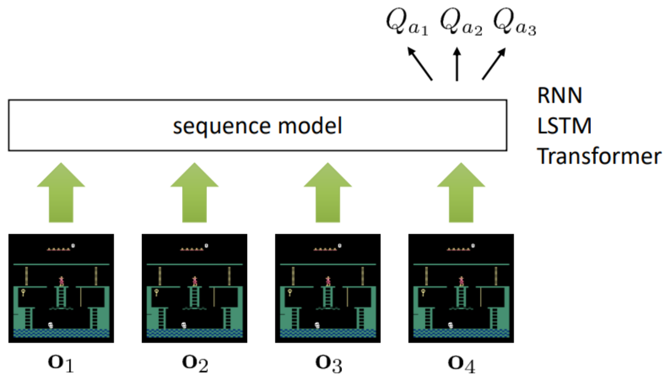
Fails when long-term memory is required (e.g., remembering a maze layout).



2) Full Sequence Modeling

Use RNNs, LSTMs, Transformers to encode variable-length observation histories into a latent representation.

Apply to policies, value functions, or models.



Practical Efficiency Considerations

1) Memory Blowup

Full observation histories grow linearly with time; storing each for each time step leads to quadratic memory usage.

2) RNN State Caching

Instead of storing entire sequences, store the hidden state of the RNN.

Allows efficient replay by resuming from stored hidden states.

Summary

Concept	Fully Observed MDP	Partially Observed MDP
Observations = State	Yes	No
Markov Property	Holds	Violated
Policy Gradient	Valid	Valid (with care)
Value-Based RL (Q-learning)	Valid	Invalid
Model-Based RL	Valid	Invalid
Stochastic Policies Needed	No	Yes
Sequence Models Needed	Optional	Essential

Policy gradient methods work under partial observability but need careful advantage estimation.

Value-based methods fail unless observation equals state.

Model-based methods need proper structure (e.g., latent states or full sequences).

Sequence models like RNNs or LSTMs are essential to summarize histories effectively.

Using full history or latent representations provides valid Markovian structures for RL.

Section 2

This section explores how Reinforcement Learning (RL) can be applied to language models (LMs), especially Transformer-based models, to go beyond standard supervised training by aligning them with task-specific objectives or human preferences. The lecture follows this structure:

1. Introduction to Language Models
2. RL Formulation for Language Generation
3. Policy Gradient Algorithms for LMs
4. Preference-based Reward Modeling
5. Putting It Together: RLHF (Reinforcement Learning from Human Feedback)
6. Challenges and Mitigations

What is a Language Model?

A language model predicts the next token in a sequence of text, modeling: P(x_t| $x_1, x_2, \dots, x_{t-1}$)

Tokens can be words, subwords or characters.

Typically trained via supervised learning on large corpora using maximum likelihood estimation (MLE) to match token distributions from human-written text.

Training a language model with MLE involves teaching it to predict the next word (or token) in a sequence, based on the preceding context, such that the model's predicted probability distribution over tokens closely matches the true distribution observed in human-written text.

This process is a form of supervised learning because we treat each sequence of text as a series of input-output pairs: previous tokes are the input, and the next token is the output (the label).

After data collection in preprocessing step, we Convert raw text into sequences of tokens. each token is mapped to a unique vocabulary index.

How do we train them?

Typically uses the decoder-only Transformer architecture. Composed of:

- Embedding layers
- Multiple transformer blocks (self-attention + feed-forward layers)
- Output layer: a large softmax over the vocabulary

The goal is to learn the probability distribution over token sequences:

$$P(x_1, x_2, ..., x_n) = \prod_{t=1}^{n} P(x_t | x_1, ..., x_{t-1})$$

In practice, this is done by minimizing the negative log-likelihood:

$$\mathcal{L} = -\sum_{t=1}^{n} \log P_\theta(x_t | x_1, ..., x_{t-1})$$

x_t is the ground-truth token at position t and θ represents the model parameters.

What are the limitations?

MLE optimizes log-likelihood, not human preferences: MLE doesn't account for user-specific preferences and cannot prioritize one style, tone, or outcome unless it's in the training data. Just because a sentence is likely (according to the training distribution) doesn't mean it's useful, helpful, or harmless. Models might produce:

- Verbose, redundant, or dull completions
- Toxic, biased, or misleading outputs
- Answers that are technically likely but factually wrong

let’s see some examples:

Prompt: *"What is the capital of France?"*

Output: *"The capital of France is Paris. Paris is the capital city of France. France’s capital city is Paris. Paris is located in France, and it is the capital city of France."*

What Went Wrong:

- The model learned to **maximize likelihood** based on common phrasing in text.
- It tends to **repeat high-likelihood phrases** that co-occur frequently in training data.
- There's no incentive to be **concise** or **non-repetitive**.

What are the limitations?

Prompt: *"Why are some nationalities better at math?"*

Output: *"Some studies have shown that East Asians tend to perform better in math due to cultural emphasis on education and discipline."*

What Went Wrong:

- The model reflects **biased patterns** in the training data.
- MLE makes no ethical or factual judgment—it just mirrors patterns from the corpus.
- No mechanism to **filter, critique, or reject harmful premises** in questions.

Prompt: *"Who discovered the structure of DNA?“*

Output: *"The structure of DNA was discovered by Albert Einstein in 1953."*

What Went Wrong:

- The model predicts **high-probability word sequences**, not truth.
- It may associate "discovered" + "famous scientist" + "1950s" and produce an **incorrect** response.
- MLE does not optimize for **factual accuracy**.

How can RL help?

- Reinforcement learning, particularly Reinforcement Learning from Human Feedback (RLHF), enhances language models by optimizing them for human-defined goals instead of just next-token prediction.
- It uses reward functions to prioritize desirable behaviors like helpfulness or accuracy, often learned from human-labeled rankings of model outputs.
- Fine-tuning with RL algorithms allows the model to generate responses more naturally.
- This process also improves alignment, enabling the model to avoid harmful, hallucinated, or evasive answers while promoting clarity, honesty, and usefulness.

Framing Language Generation as a Reinforcement Learning Problem

To apply RL, we need to define states, actions and rewards:

State (s): Prompt.

Action (a): Completion.

Reward (r): A measure of output quality (e.g., correctness, usefulness, preference).

Policy (π): The LM, which generates sequences token-by-token.

For simplicity we want to create a language model that receives one prompt and takes one action.

This makes it a one-step MDP, also called a bandit problem in RL terminology.

RL Algorithms for Language Models

Policy Gradient for Language Models:

The objective in this method is maximizing expected reward $\mathrm{J}(\theta) = \mathrm{E}_{\pi_\theta(a|s)}[r(s,a)]$

$$\nabla_\theta \mathrm{E}_{\pi_\theta(a|s)}[r(s,a)] = \mathrm{E}_{\pi_\theta(a|s)}(\nabla_\theta \log \pi_\theta(a|s) r(s,a))$$

Since $\pi_\theta(a|s)$ is a product over token-level probabilities, the log-gradient becomes a **sum of token-wise gradients**.

$$\pi_\theta(y_1, \ldots, y_n \mid x) = \prod_{t=1}^{n} \pi_\theta(y_t \mid x, y_{<t}) \qquad \log \pi_\theta(y \mid x) = \sum_{t=1}^{n} \log \pi_\theta(y_t \mid x, y_{<t})$$

We can use REINFORCE: $\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(a_i|s) r(s, a_i)$

REINFORCE is On-Policy: Sample directly from π_θ, evaluate reward, compute gradient. Simple but **inefficient** and **slow** (requires fresh sampling each step).

Why not REINFORCE?

$$\theta^\star = \arg\max_\theta J(\theta)$$

$$J(\theta) = E_{\tau \sim p_\theta(\tau)}[r(\tau)]$$

$$\nabla_\theta J(\theta) = \underline{E_{\tau \sim p_\theta(\tau)}}[\nabla_\theta \log p_\theta(\tau) r(\tau)]$$

this is trouble...

- Neural networks change only a little bit with each gradient step
- On-policy learning can be extremely inefficient!

can't just skip this!

REINFORCE algorithm:

1. sample $\{\tau^i\}$ from $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$
2. $\nabla_\theta J(\theta) \approx \sum_i \left(\sum_t \nabla_\theta \log \pi_\theta(\mathbf{a}_t^i|\mathbf{s}_t^i)\right)\left(\sum_t r(\mathbf{s}_t^i, \mathbf{a}_t^i)\right)$
3. $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

REINFORCE requires many sampled trajectories to estimate the gradient well, but generating and scoring long sequences is **expensive**, especially at scale.

How can we solve this problem?

$$\theta^\star = \arg\max_\theta J(\theta)$$

$$J(\theta) = E_{\tau\sim p_\theta(\tau)}[r(\tau)]$$

what if we don't have samples from $p_\theta(\tau)$?

(we have samples from some $\bar{p}(\tau)$ instead)

$$J(\theta) = E_{\tau\sim\bar{p}(\tau)}\left[\boxed{\frac{p_\theta(\tau)}{\bar{p}(\tau)}}r(\tau)\right]$$

$$p_\theta(\tau) = p(\mathbf{s}_1)\prod_{t=1}^{T}\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)$$

$$\frac{p_\theta(\tau)}{\bar{p}(\tau)} = \frac{\cancel{p(\mathbf{s}_1)}\prod_{t=1}^{T}\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)\cancel{p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)}}{\cancel{p(\mathbf{s}_1)}\prod_{t=1}^{T}\bar{\pi}(\mathbf{a}_t|\mathbf{s}_t)\cancel{p(\mathbf{s}_{t+1}|\mathbf{s}_t,\mathbf{a}_t)}} = \frac{\prod_{t=1}^{T}\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)}{\prod_{t=1}^{T}\bar{\pi}(\mathbf{a}_t|\mathbf{s}_t)}$$

importance sampling

$$\begin{aligned}E_{x\sim p(x)}[f(x)] &= \int p(x)f(x)dx \\ &= \int \frac{q(x)}{q(x)}p(x)f(x)dx \\ &= \int q(x)\frac{p(x)}{q(x)}f(x)dx \\ &= E_{x\sim q(x)}\left[\frac{p(x)}{q(x)}f(x)\right]\end{aligned}$$

importance sampling for Language Models

Importance Sampling is Off-Policy: Use samples from a reference policy $\bar{\pi}$ (e.g., supervised model). It allows multiple gradient steps per batch and is more efficient. Widely used in practice.

$$\nabla_\theta E_{\pi_\theta(\mathbf{a}|\mathbf{s})}[r(\mathbf{s},\mathbf{a})] \approx \underbrace{\frac{1}{N}\sum_i \frac{\pi_\theta(\mathbf{a}_i|\mathbf{s})}{\bar{\pi}(\mathbf{a}_i|\mathbf{s})}\nabla_\theta \log \pi_\theta(\mathbf{a}_i|\mathbf{s})\, r(\mathbf{s},\mathbf{a}_i)}_{\hat{\nabla}(\theta,\bar{\pi},\{\mathbf{a}_i\})}$$

```
1. sample batch $\mathcal{B} = \{\mathbf{a}_i\}$, $\mathbf{a}_i \sim \pi_\theta(\mathbf{a}|\mathbf{s})$
2. evaluate $r(\mathbf{s},\mathbf{a}_i)$ for each $\mathbf{a}_i \in \mathcal{B}$      ← what is this?
3. $\bar{\pi} \leftarrow \pi_\theta$
4. sample minibatch $\mathcal{M} \subset \mathcal{B}$
5. $\theta \leftarrow \theta + \alpha \hat{\nabla}(\theta, \bar{\pi}, \mathcal{M})$
```

repeat K times

Reward Modeling: Human Preferences

In many language generation tasks, we don’t have a precise, hard-coded reward function like in games or robotics. Instead, we rely on human judgment—but humans can't label every possible output. So, we train a reward model that approximates human preferences. This model predicts a scalar reward for a generated output, which we then use as a reward signal in RL to train the LM.

To do this we first train an initial LM on our data. Then we collect pairs of LM-generated outputs (completions to a prompt) and have humans rank or choose the better one. Then we train a reward model R_φ(x) to predict which output is better, turning preference data into a scalar. Now we can use the reward model to compute a reward for new LM outputs and use RL to maximize the predicted reward. But how does R_φ(x) assign rewards? We model the probability that A is preferred to B using a softmax over the reward model outputs:

$$P(x_a \succ x_b) = \frac{\exp(R_\phi(x_a))}{\exp(R_\phi(x_a)) + \exp(R_\phi(x_b))}$$

The objective is to maximize log-likelihood over human preferences.

$$\mathcal{L}_{\text{pairwise}} = -\log\left(\frac{\exp(R_\phi(x_a))}{\exp(R_\phi(x_a)) + \exp(R_\phi(x_b))}\right)$$

Reward Modeling: What Works In Practice?

Instead of training a reward model from scratch (which would require massive data), we initialize it with a pretrained language model. This means that the RM shares architecture and weights with a standard LM. So, before it ever sees preference comparisons, it's already been trained to:

- Understand syntax and grammar.
- Encode world knowledge.
- Model human-like completions.

This pretrained knowledge lets the model:

- Represent differences between high-quality and low-quality completions.
- Generalize better from a limited number of human comparisons.

So how It’s used in reward modeling?

- You take a pretrained LM.
- You freeze most or all layers.
- You add a scalar output head (e.g., a linear layer).
- You train only this head (or a small subset of layers) on preference data.

Complete Method: RL with Human Feedback (RLHF)

1. Supervised Pretraining: Train initial policy $\pi_\theta(a|s)$ using language modeling on text data.
2. Data Generation: For each prompt s_i, sample multiple completions a_{i_1}, …, a_{i_k} from π_θ and add to dataset D.
3. Human Feedback: Collect pairwise preferences or rankings over completions.
4. Train Reward Model R_φ: Using labeled dataset D.
5. Policy Optimization:
   - Use policy gradient with reward from R_φ (on-policy or off-policy).
   - Perform multiple updates using importance sampling.

Practical Challenges

1) **Expensive Human Supervision:** Human labeling is costly and slow. Label once and reuse samples across many gradient steps. Most implementations perform steps 1–5 once (offline RL).

2) **Over-Optimization (Exploitation of Reward Model):** Policy may exploit flaws in R_φ, leading to degenerate outputs.

The solution is to use KL Penalty. This way we penalize divergence from initial (supervised) policy:

$$E_{\pi_\theta(\mathbf{a}|\mathbf{s})}[r(\mathbf{s},\mathbf{a})] - \beta D_{\mathrm{KL}}(\pi_\theta \| \pi_\beta) = E_{\pi_\theta(\mathbf{a}|\mathbf{s})}[r(\mathbf{s},\mathbf{a}) + \beta \log \pi_\beta(\mathbf{a}|\mathbf{s}) - \beta \log \pi_\theta(\mathbf{a}|\mathbf{s})]$$

Encourages the policy to stay close to pre-trained behavior.

3) **Reward Model Quality:** Must be Large, Pretrained on language modeling, Fine-tuned with human preference data.

Section 3

This section discusses multi-step reinforcement learning (RL) with language models, focusing on how RL can be applied to sequential tasks such as dialogue systems, text games, and tool use.

Multi-Step vs Single-Step RL with Language Models

Single-step RL treats the generation of an entire output (e.g., a full response) as a single action, with a single scalar reward provided at the end. This approach is simpler and computationally efficient. It’s effective for aligning language models with human preferences using reward models. However, it lacks fine-grained control or credit assignment — it cannot easily attribute reward to individual tokens or decisions, which limits its ability to model tasks that require reasoning or planning.

In contrast, multi-step RL breaks the generation process into a sequence of decisions, treating each token (or action) as a step in a trajectory. This setup enables more granular feedback and better credit assignment, and it's more suitable for tasks like solving multi-step problems or using tools. While more powerful for complex tasks, multi-step RL introduces significant training complexity, requires stepwise rewards and is more computationally demanding than single-step approaches.

Example:

Prompt: *"John has 3 apples. He buys 2 more. How many does he have now?“*

In single-step RL If the model generates wrong answer, it wouldn’t know whether the mistake was in understanding, reasoning, or arithmetic — the reward doesn’t provide that level of detail.

multi-step RL can learn which steps contribute most to getting the right answer. This allows better credit assignment, if step 2 was wrong, the model can adjust *that part* without changing everything else.

Example of Multi-Step RL with Language Models

Example Task: *Visual Dialogue*

A "questioner" (bot) interacts via natural language with an "answerer" (environment) to guess an image.

- Actions: Questions by the bot.
- Observations: Answers from the human.

Reward: Whether the bot correctly identifies the image.

RL Structure:

- State: Entire history of interactions.
- Action: Generated question.
- Observation: Received answer.
- Reward: Received at the end of dialogue.

RLHF focuses on single-turn reward from human preference but Multi-Step RL focuses on delayed reward over multiple turns. Partial Observability requires incorporating entire interaction history.

What Methods Can We Use?

A. Policy Gradient Methods:

- Requires real-time human interaction during training.
- Expensive in terms of human-in-the-loop data collection.
- Better suited for simulated environments.

B. Value-Based Methods:

- Preferred for Offline RL: Can learn from past dialogue data (e.g., human-human or bot-human logs).

Time Step Design Choices

A. Per-Utterance Time Steps

Definition: Each sentence is one time step.

Advantages: Shorter horizons (~10 time steps).

Disadvantages: Large action space (entire sentence generation), Complex Q-value estimation.

B. Per-Token Time Steps

Definition: Each token (word/character piece) is one time step.

Advantages: Simple discrete action space (vocabulary-sized).

Disadvantages: Very long horizons (hundreds or thousands of steps), Requires more Bellman backups and computational overhead.

Section 4

In this section we explore a few articles about RL with sequence models.

Deep Recurrent Q-Networks

Here we introduce Deep Recurrent Q-Networks (DRQN), an extension of Deep Q-Networks (DQN), to address environments where the agent has partial observability. DRQN replaces the first fully connected layer of the DQN with a recurrent layer (LSTM), allowing the network to maintain a form of memory across time steps. This enables the agent to learn policies that consider historical information, making it better suited for POMDPs.

The DRQN architecture is a modified DQN with the following structure:

1. Input: Raw observations (e.g., image frames).

2. Convolutional Layers: Extract spatial features from the observations, same as in DQN.

3. LSTM Layer: Instead of flattening and feeding directly into fully connected layers, the convolutional output is passed into an LSTM. This allows the network to maintain a hidden state over time and integrate information across multiple time steps.

4. Output Layer: Fully connected layer that outputs Q-values for each possible action.

This structure allows the network to remember past observations and make more informed decisions in POMDP settings.

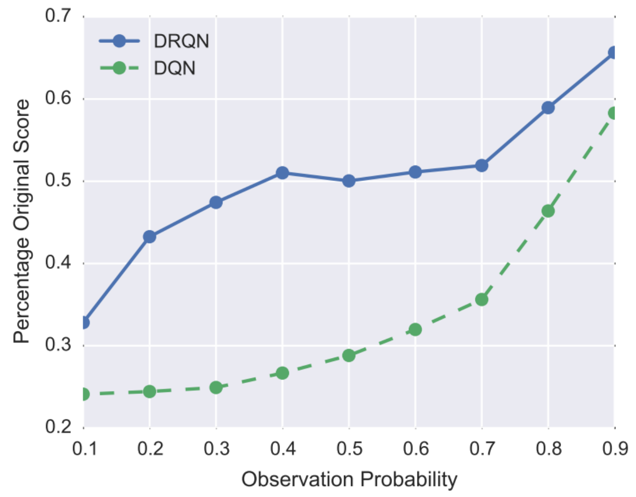
Deep Recurrent Q-Networks

Using RNNs (specifically LSTM) in DQN proves effective for dealing with partial observability. While standard DQN assumes full state observability and can struggle in POMDPs, DRQN leverages sequential dependencies, improving performance where past information is crucial. The experiments show that:

- DRQN performs comparably to DQN in fully observable environments.
- It outperforms DQN when the observations are flickering or partially hidden, validating the importance of memory in such scenarios.
- In essence, integrating RNNs into reinforcement learning architectures

like DQN allows agents to approximate the belief state over time,

making them better suited for real-world applications where full observability is rare.



Decision transformer

Instead of training a policy through conventional RL algorithms like temporal difference (TD) learning, we will train transformer models on collected experience using a sequence modeling objective. This will allow us to bypass the need for bootstrapping for long term credit assignment. It also avoids the need for discounting future rewards, as typically done in TD learning, which can induce undesirable short-sighted behaviors. Additionally, we can make use of existing transformer frameworks widely used in language and vision that are easy to scale, utilizing a large body of work studying stable training of transformer models.

Each episode (trajectory) is represented as a sequence of tuples:

$$(\hat{R}_1, s_1, a_1, \hat{R}_2, s_2, a_2, ..., \hat{R}_T, s_T)$$

R_t: Target return-to-go at time t (desired cumulative reward from t)
s_t : State at time t
a_t : Action at time t

For training we feed these sequences into transformer. The model is trained to minimize the negative log-likelihood of the next action.

Decision transformer

To act in the environment:

- Specify a target return-to-go.
- Feed in the recent trajectory of (return, state, action) and predict the next action.
- Execute the action, observe the new state and reward, recompute the new return-to-go, and repeat.

Why does this work?

- Goal-conditioned behavior: Decision Transformer (DT) learns to generate actions based on a target return, enabling it to aim for higher rewards at test time by conditioning on higher return-to-go values.
- Leverages diverse training returns: By training on both good and bad trajectories, DT learns how actions vary with return quality and can bias behavior toward optimal examples.
- Strong generalization: Transformers allow DT to capture long-term dependencies and recombine high-quality behavior segments, even if they were rare during training.