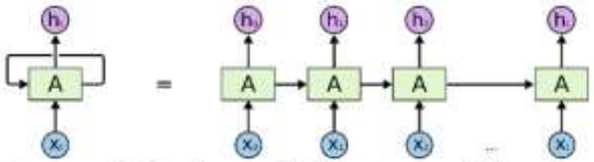
Neural Networks Designed for Sequential Data

(RNNs, LSTMs, Transformers)

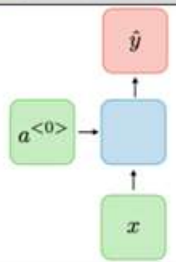
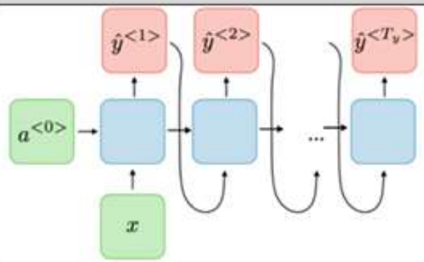
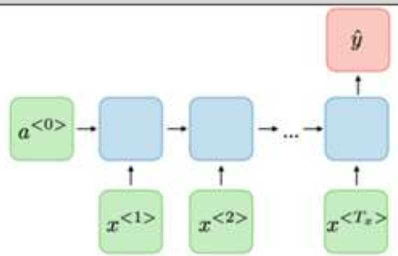
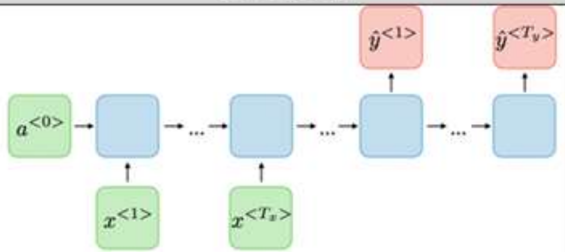
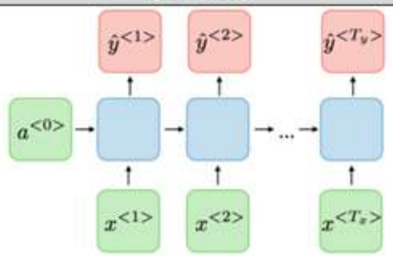
What can RNNs be used for, and how do they work?

- **Recurrent Neural Networks (RNNs)** allow you to work with nonfixed, variable-length sequential data, e.g. time series data or sentences, by building a hidden state over time.
- During training, we do **backpropagation through time**, for some number of timesteps that we choose. However, **vanishing/exploding gradients** is a significant problem.
- In practice, **long term dependencies** are an issue; it's difficult to "remember" previous context when the time step gap is very large, since repeatedly overwritihg a hidden state leads to data loss.
- In ordinary RNNs, there is no **bidirectionality**; for many-to-one or many-to-many settings, we can only use the inputs which we have already seen when making a prediction, and can't use later input elements.
  - However, there are ways to address this, e.g. Bidirectional RNNs which read left-to-right and right-to-left in parallel.
- RNNs and its variants are related to **Bayesian filtering**. Both work with sequential data/measurements to update an internal state, to predict a target variable.
  - Bayesian filters provide probabilities and are more "specialist", useful in settings where the noise and dynamics are well-characterized



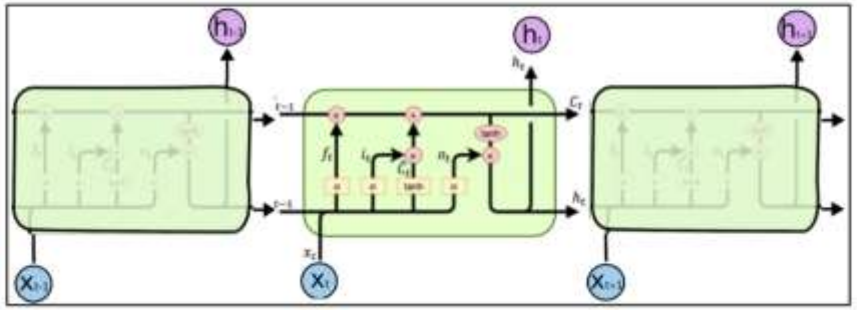
Composed of an internal hidden state which updates itself with each input x.
At some time steps, we will ask it to predict a vector.

- In general, $h_t = f_w(h_{t-1}, x_t) = hidden\ state\ at\ time\ t$
  - $f_w = some\ function\ with\ parameters\ w$
    - Weights and function are fixed – don't change with time
  - $h_{t-1} = previous\ time\ step's\ hidden\ state$
  - $x_t = input\ @\ time\ t$
- In general, $y_t = f_y(h_t) = output\ for\ time\ t$
- A "vanilla" version uses matrices & tanh nonlinearities for f_w, f_y
  - $h_t = tanh\,(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$
  - $y_t = W_{hy}h_t + b_y$

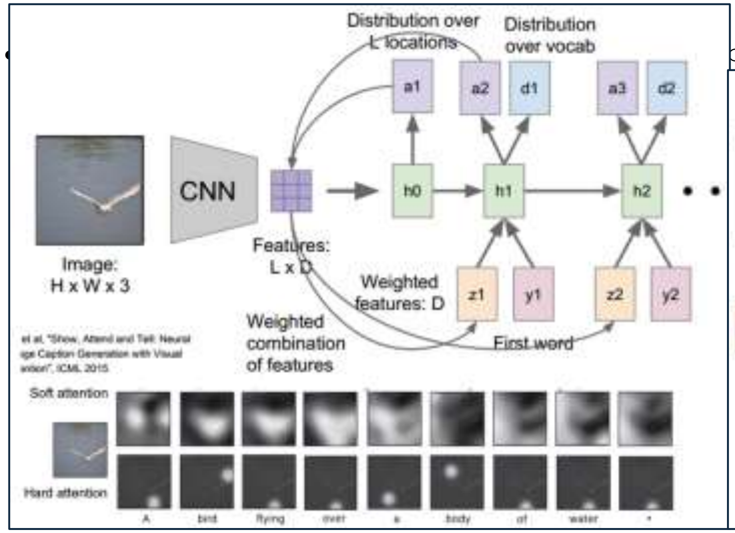
I/O Setup	One to One (vanilla feedforward network)	One to Many	Many to One	Many to Many (indirect)	Many to Many (direct)
Graph					
Remarks	A vanilla feedforward NN.	Previous output prediction fed back in as input ("**autoregression**"). Can be used as a sequential decoder.	Can be used as a sequential encoder.		
Example Applications	Image classification, monocular depth est, SVR	Image captioning, music generation from text.	Sentence sentiment classification, video classification.	Language translation, visual question answering (VQA; image+sentence input, sentence output)	Temporally smoothed video tracking or depth estimation

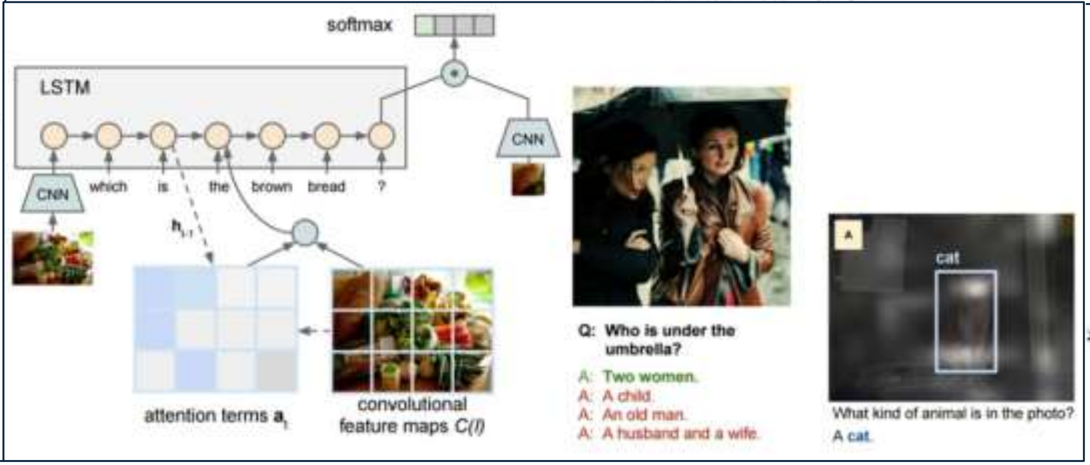
What are LSTMs used for and how do

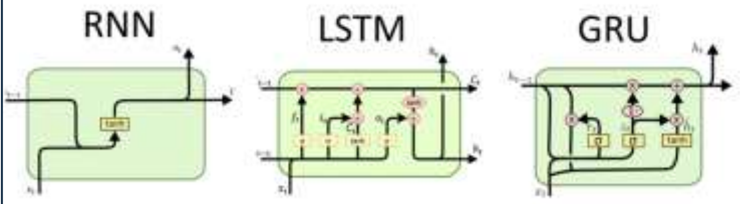
- **Long Short Term Memory (LSTM)** is a RNN variant solves issues with exploding/vanishing gradients by two modifications, which improve gradient flow (at a high level, similar to ResNets' residuals):
    - Incorporate a **cell state** (c) used only internally for each time step. The hidden state (h) is still used for the output.
    - Replacing the simple tanh/matrix multiplication update rule with a **gating mechanism** which updates both the cell and hidden states.
    - The gating mechanism determines what to forget, update, & reveal in the output (vs keep in cell state)
    - Allows for discarding irrelevant information
- There are many variants on LSTMs with different configurations, e.g. adding/removing gates, or only using hidden states (no cell state).
    - A popular example is the **Gated Recurrent Unit (GRU)**
    - Some research finds that GRUs do better, some say that they do the same. The consensus seems to be
- olving NLP, such as:



- An LSTM module has a hidden state (h, will be used for output) and a cell state (c, only used internally) for each time step
- There are three gates:
    - **f (forget gate):** Decides what information from the cell state should be thrown away or kept, using a sigmoid in [0,1].
        - $f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$
    - **i (input/information gate):** Decides which values should be updated in the cell state, using a sigmoid. Additionally, the update magnitude and direction is determined by a tanh.
        - $i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$
        - $\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C)$
    - **o (output gate):** Decides how much to reveal/filter the cell state to the hidden state, to be output. This is done with a sigmoid.
        - $o_t = \sigma(W_o [h_{t-1}, x_t] + b_o)$
- Overall, the cell state is updated with the forget gate and input gate outputs:
  $C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$
- The hidden state is updated with the current cell state and the output gate's filter:
  $h_t = o_t * \tanh(C_t)$

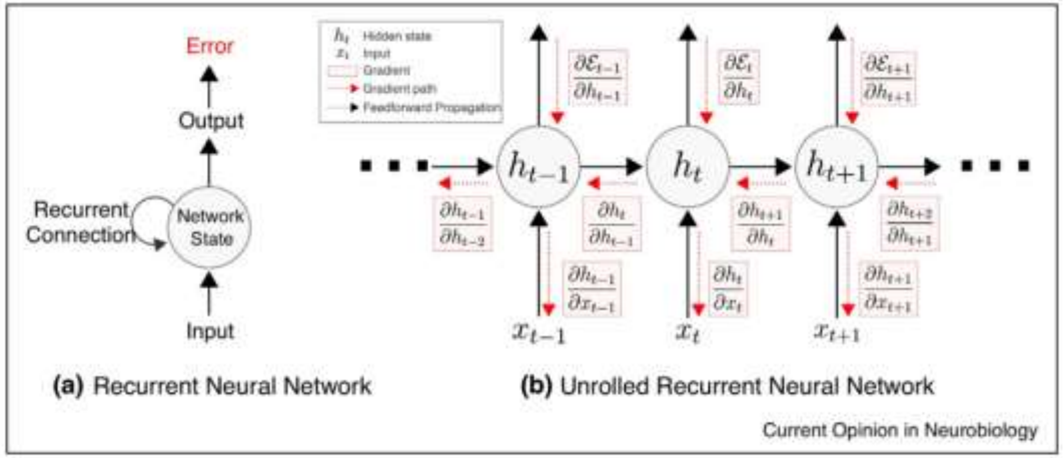


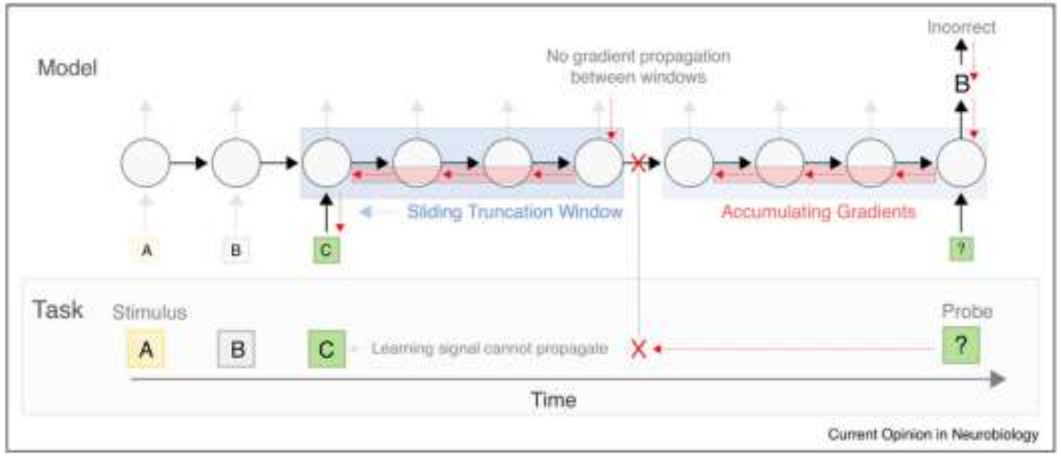




How does backpropogation through time (BPTT) work?

- Enables learning of temporal dependencies by computing gradients across all timesteps of an RNN forward pass
  - In contrast, regular backpropagation only cares about the current input-output mapping, not how past inputs influence the present.
  - Intuition: Learning a musical instrument. If you hit the wrong note in bar 8, the mistake could be because of something you did in bar 5 (e.g., bad hand positioning). You need to trace the error *back through time* to understand and correct it.
- Challenges:
  - Vanishing/exploding gradients as they pass through time steps
  - Computationally expensive
- **Truncated BPTT** limits the number of time steps we backprop through for efficiency & stability
  - Intuitively, we assume that the influence of things that happened more than N steps ago is small enough that we can ignore it for training purposes
  - Resulting model might fail to capture long-term dependencies
  - Previous hidden states still influence future ones – we just don't train the network to fully learn how changes in earlier states affect later ones



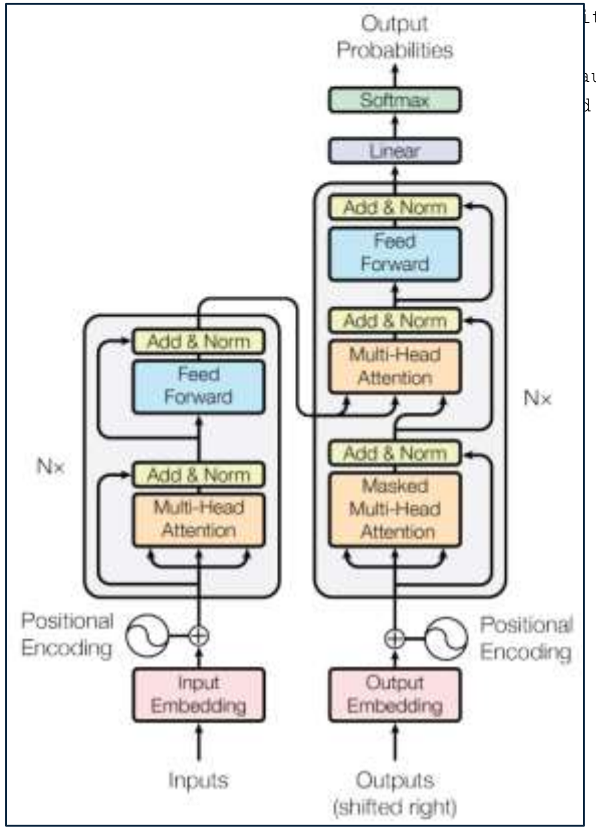


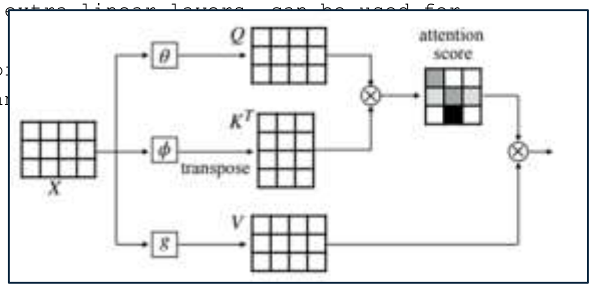
$$\frac{\partial E}{\partial h_t} = \frac{\partial E}{\partial h_T}\frac{\partial h_T}{\partial h_t} = \frac{\partial E}{\partial h_T}\prod_{k=t}^{T-1}\frac{\partial h_{k+1}}{\partial h_k}$$

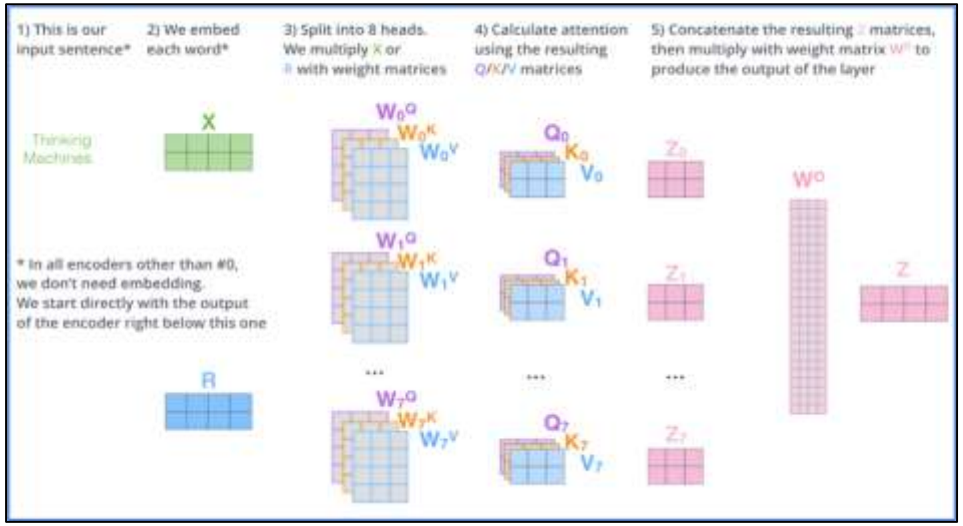
In NLP, what are Transformers and how do they work at a high level? How do they differ from LSTMs/RNNs?

There are several significant differences between vanilla Transformers and RNNs:

- They use an encoder-decoder NN structure, and encoder inputs are **fed all at once**; recurrent hidden states are completely removed
  - This allows for very **explicit modeling of long-range dependencies** compared to RNNs/LSTMs
  - Workarounds like "**chunking**" are necessary for inputs larger than the maximum size
  - Does not need to process the beginning of the sentence before the end, allowing for **parallelism** during training
  - Along with the use of skip connections, this partially **addresses the BPTT vanishing/exploding gradients**
  - To encode order/time series information, **positional encoding** is added to the inputs
- Adopts a **self-attention mechanism**, which differently weighs the significance of each part of the input data, dynamically on-the-fly during inference.
  - Compared to normal learned weights, these can be considered "**dynamic data-dependent weights**" or "fast weights"
  - Intuitively, this enables "selective" **memory** to attend to specific relevant parts of the input
- Two well-known models based on Transformers are:
  - **BERT** is encoder-only and is non-autoregressive. It can be thought of as an







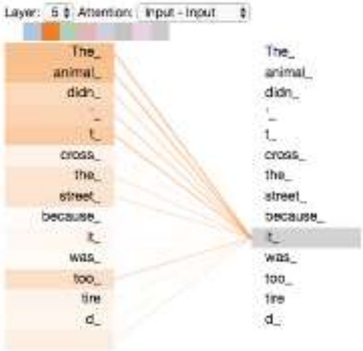
Encoder input preparation:

- Input words tokenized as fixed length vectors (e.g. 512 dims)
- Positional encodings are added to each word embedding
- With inputs in matrix X, this is fed into the first encoder blocks
- Output is a contexualized embedding, per-token that is richer than the initial word embeddings since it looks at the surrounding context

Decoder is similar to the encoder, with a few changes:

- Decoder performs **autoregression** repeatedly predicts the next word (with a linear layer which is the size of the corpus and softmax, at top), refeeding that prediction back into itself to obtain the next input
- **Masked self-attention layer** is only allowed to attend to earlier positions in the output sequence
- **Multi-Head Attention layer** works just like the encoder, except it creates its Queries matrix from the layer below it, and takes the Keys and Values matrix from the last encoder stack

- Useful analogy: Attention is an aggregation operation that allows tokens to communicate with each other (reduce); MLPs process each token individually (map). So it's a bunch of map-reduce operations.



Given a text sequence with L_t tokens each of a d_t dimensional embedding, and an image patch sequence with L_i tokens each of a d_i dimensional embedding, write the equations associated with text attending to image (# heads=h).

- $X_t \in \mathbb{R}^{L_t \times d_t}$: Text sequence with L_t tokens, each of a d_t dimensional embedding
- $X_i \in \mathbb{R}^{L_i \times d_i}$: Image patch sequence with L_i tokens, each of a d_i dimensional embedding
- $W_{Q,j} \in \mathbb{R}^{d_t \times d_q}$; (in pytorch, $d_q = d_v \coloneqq d_t$)
- $W_{K,j} \in \mathbb{R}^{d_i \times d_q}$
- $W_{V,j} \in \mathbb{R}^{d_i \times d_v}$
- $Q_j = X_t W_{Q,j} \in \mathbb{R}^{L_t \times d_q}$; intuitively, what you're currently focusing on or asking about (here, text)
- $K_j = X_i W_{K,j} \in \mathbb{R}^{L_i \times d_q}$; intuitively, the tags/label of what you're referencing (here, image patches)
- $V_j = X_i W_{V,j} \in \mathbb{R}^{L_i \times d_v}$; intuitively, the content / actual information of what you're referencing (here, image patches)
- $A_j = softmax\left(\frac{Q_j K_j^T}{\sqrt{d_q}}\right) \in \mathbb{R}^{L_t \times L_i}$; intuitively, what part of the image each word should focus on. Softmax collapsing rows, so that probabilities are over image tokens.
- $h_j = A_j V_j \in \mathbb{R}^{L_t \times d_v}$; intuitively, gathering weighted avg of visual image context, relevant to each word
- $W_O \in \mathbb{R}^{h d_v \times d_t}$
- $O = concat(h_1, \ldots, h_h) W_O \in \mathbb{R}^{L_t \times d_t}$
- (often, $d_t = d_i = d$ which then reduces to the same setup as self-attention)

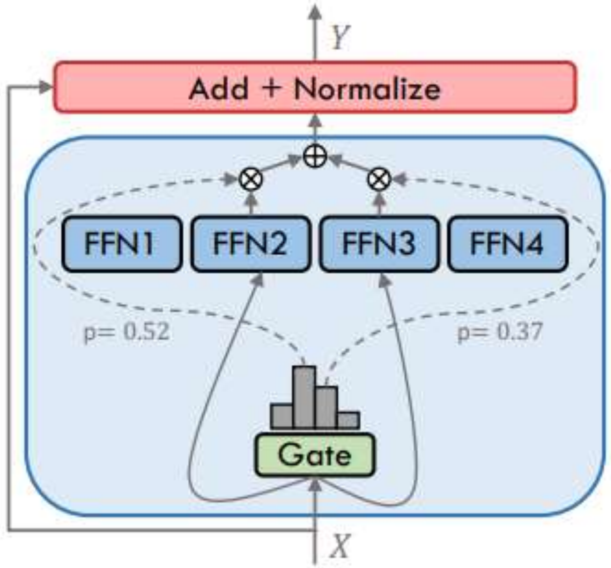
What’s the difference between models like BERT & GPT?

Feature	**GPT**	**BERT**
Name	*Generative Pre-trained Transformer*	*Bidirectional Encoder Representations from Transformers*
Architecture	Decoder-only Transformer	Encoder-only Transformer
Pretraining Task	Causal Language Modeling (predict next token using only past context; uses causal masking to ensure it can’t cheat by looking at future words)	Masked Language Modeling (predict missing tokens, eg “The [MASK] is blue” → predicts “sky”)
Training Direction	Left-to-right (unidirectional)	Bidirectional
Usage	Text generation: Chatbots, text generation, code generation, story writing, etc.	For when you want rich, contextualized embeddings per input token. Used for text understanding/comprehension, sentiment analysis, question answering, classification, named entity recognition, etc. Also used in retrieval-augmented generation to retrieve documents to later pass into encoder.

In principle generative models like GPT could also use an encoder to achieve better understanding of the input prompt, but is generally avoided for simplicity & scalability, and

Describe how mixture-of-experts (MoE) models are formulated.

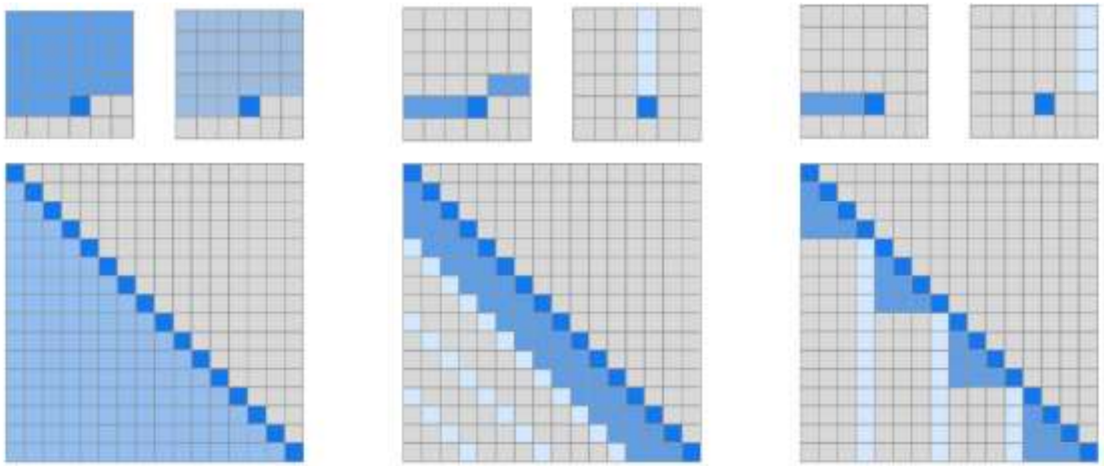
- Note that the vanilla transformer block's FFN is a dense network, since every component is activated on every forward pass; MoE layer provides a sparse model/network
- Router is a FFN that does a n-way classification for N experts. Can choose one expert or multiple and compute weighted mean
- Still need to load all parameters, but speeds up inference since you don't need to go through all weights
- Can lead to tricker training due to complexity and risk of overfitting, collapse, etc
    - Eg can introduce load balancing loss to encourage uniform routing



(b) Sparse MoE

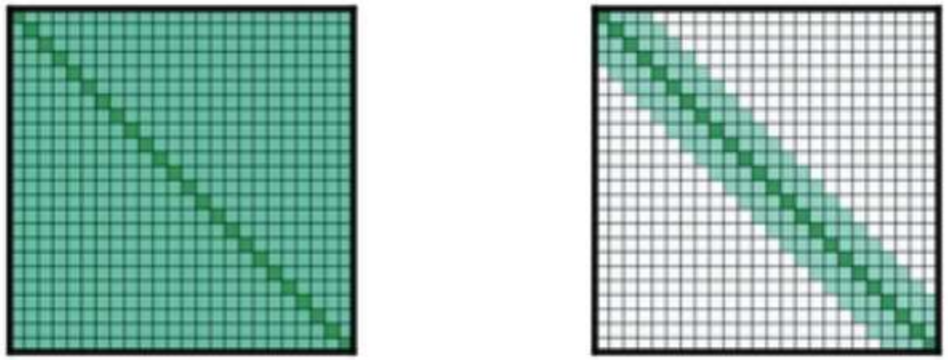
Explain how sparse/windowed attention works.

- Reduces the quadratic complexity of the standard attention mechanism by computing attention over a subset of token pairs rather than all token pairs
  - From $O(dn^2)$ to $O(dnlog(n))$ or $O(dn)$, where n is sequence length and d is the hidden size



(a) Transformer (b) Sparse Transformer (strided) (c) Sparse Transformer (fixed)

Figure 3. Two 2d factorized attention schemes we evaluated in comparison to the full attention of a standard Transformer (a). The top row indicates, for an example 6x6 image, which positions two attention heads receive as input when computing a given output. The bottom row shows the connectivity matrix (not to scale) between all such outputs (rows) and inputs (columns). Sparsity in the connectivity matrix can lead to significantly faster computation. In (b) and (c), full connectivity between elements is preserved when the two heads are computed sequentially. We tested whether such factorizations could match in performance the rich connectivity patterns of Figure 2.



(a) Full n^2 attention (b) Sliding window attention

Explain how KV Caching works.

- Unlike during training, with autoregressive inference we only use a single query token (the previously predicted output token)
- As a result, we only need to compute 3 vectors Q_t, K_t, V_t and can use cached values for $K_{1:t-1}, V_{1:t-1}$
- Results in $O(t)$ instead of $O(t)^2$ time per-token

With KV caching:

- **Cache** $K_1, \ldots, K_{t-1}$ and $V_1, \ldots, V_{t-1}$ from previous steps.
- At time t:
  - Compute only Q_t, K_t, V_t
  - Append K_t, V_t to cache:

$$K_{1:t} = \mathrm{concat}(K_{1:t-1}, K_t) \in \mathbb{R}^{t \times d_k}$$
$$V_{1:t} = \mathrm{concat}(V_{1:t-1}, V_t) \in \mathbb{R}^{t \times d_v}$$

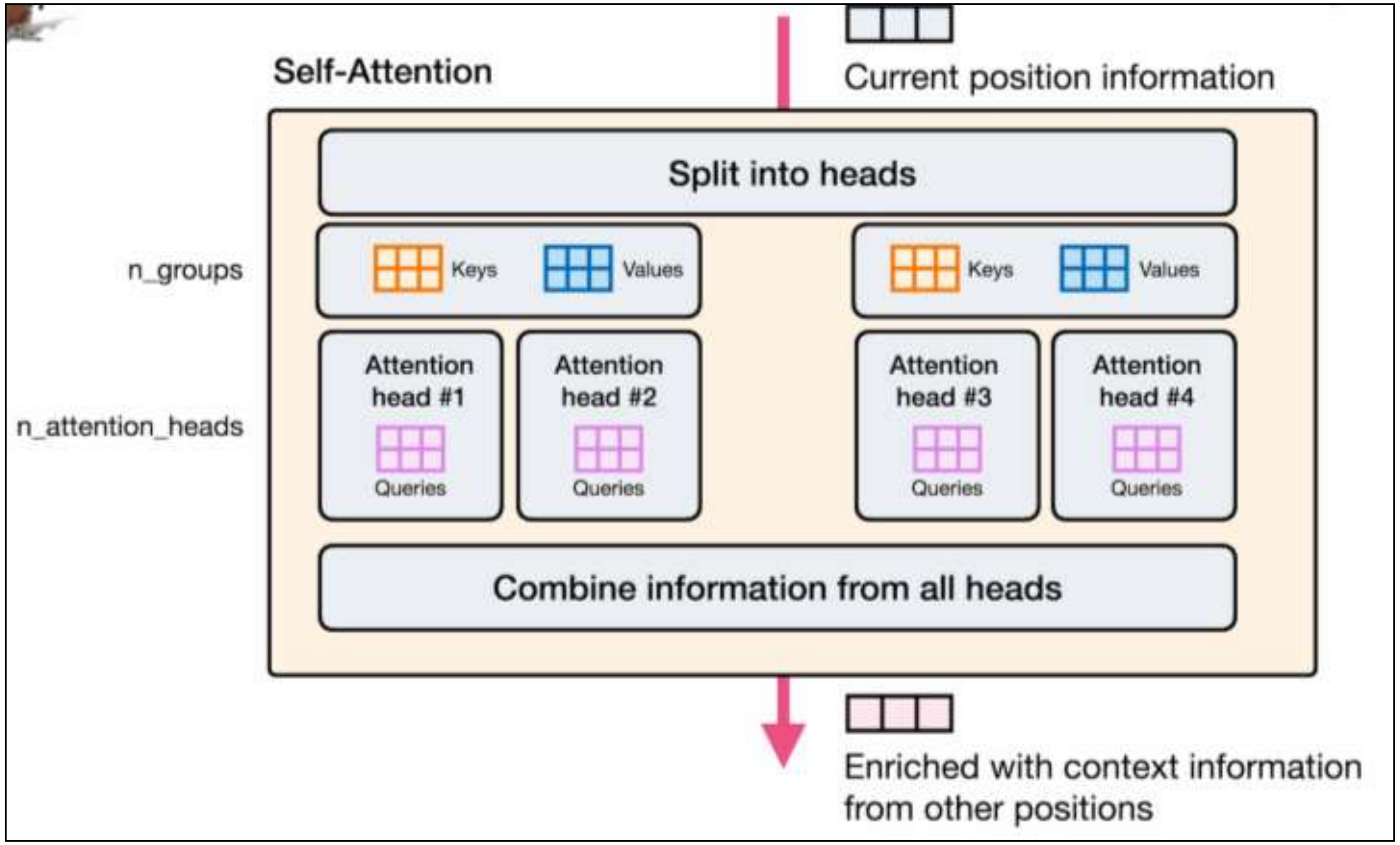
- Attention becomes:

$$\mathrm{Attention}_t = \mathrm{softmax}\left(\frac{Q_t K_{1:t}^{\top}}{\sqrt{d_k}}\right) V_{1:t}$$

You **never recompute** any K_i, V_i for $i < t$. You just **retrieve** them from cache.

What is Grouped Query Attention?

- Variant of multi-head attention, where several groups share the same keys and values; still one query per head.
- Reduces memory and latency, especially with KV-caching



How is training and autoregressive inference different for decoder-only transformer LLMs?

	Training	Inference
Causal Masking	Essential, or else network can cheat in the next token prediction	In principle, not required since there is no "cheating" by construction. However, we still don't want previously predicted tokens to attend to later predicted tokens after it because it'll cause a train-test mismatch
Queries	Use sequence of queries. Consequently, predicts next token at all positions	Only use the latest previous generated token as query. Consequently, predicts one token at a time.
KV Caching	Not used, since we are training over randomized batches	Used to speed up inference

How does RoPE work?

- Relative positional encoding is generally better than absolute
  - Lets model generalize better across sequence lengths
  - Learns "distance patterns" instead of fixed positions; better shift invariance
- Rotary Position embeddings (RoPE) allow relative positioning without any trainable parameters
  - Done inside self-attention, right after the linear projection and before computing attention scores
  - Unlike regular absolute embeddings which are baked/added into the input itself
  - Crucially, dot product of q_i and k_j become a function of relative positions

Let:

- $\mathbf{x}_m \in \mathbb{R}^d$: token embedding at position m,
- $\mathbf{q}_m = W_q \mathbf{x}_m$: query vector,
- $\mathbf{k}_n = W_k \mathbf{x}_n$: key vector,
- d is even.

We split $\mathbf{q}_m$ and $\mathbf{k}_n$ into $d/2$ consecutive 2D subvectors and apply a position-dependent rotation to each subvector using a block-diagonal matrix $\mathbf{R}_{\Theta,m} \in \mathbb{R}^{d \times d}$:

$$\mathbf{q}_m^{\mathrm{RoPE}} = \mathbf{R}_{\Theta,m}\mathbf{q}_m, \quad \mathbf{k}_n^{\mathrm{RoPE}} = \mathbf{R}_{\Theta,n}\mathbf{k}_n$$

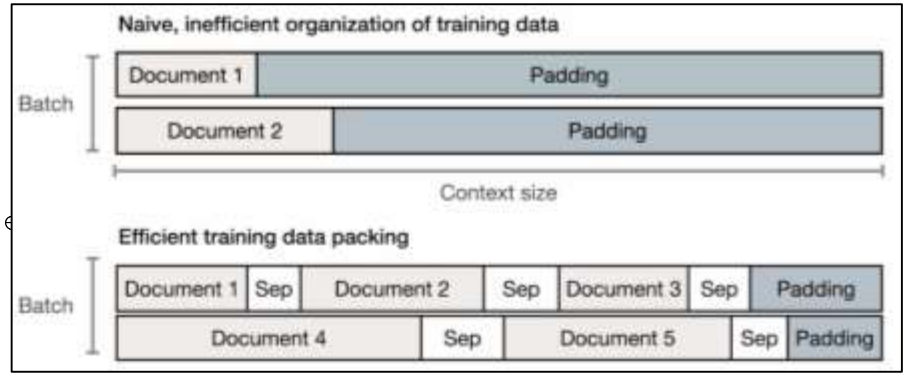
Where each 2D block in $\mathbf{R}_{\Theta,m}$ is a rotation matrix:

$$\begin{bmatrix} \cos(m\theta_i) & -\sin(m\theta_i) \\ \sin(m\theta_i) & \cos(m\theta_i) \end{bmatrix} \quad \text{for } i = 1, \ldots, d/2$$

and $\theta_i = 10000^{-2(i-1)/d}$

$$\mathrm{score}_{m,n} = \langle \mathbf{q}_m^{\mathrm{RoPE}}, \mathbf{k}_n^{\mathrm{RoPE}} \rangle = \mathbf{q}_m^\top \mathbf{R}_{\Theta,m}^\top \mathbf{R}_{\Theta,n}\mathbf{k}_n = \mathbf{q}_m^\top \mathbf{R}_{\Theta,n-m}\mathbf{k}_n$$

- YaRN … ective extr… ng train-test mism…



What is ReAct prompting?

- Approach that takes a frozen LLM trained without any tool use, and endows it with tool use without any retraining using prompt engineering & a controller
  - Controller parses output tokens, and when an api call is invoked, executes it, appends the text to the ongoing context window, and continues autoregressively decoding
- Philosophically emphasizes a tight synergy that interleaves acting & reasoning, mimicking humans
- Demonstrates tool use via manually written trajectories / text; leverages in-context learning / few shot prompting.
  - No new tokens introduced, just uses text
- Paper also explores optional fine-tuning using psuedolabeled ReAct trajectories

(1d) ReAct (Reason + Act)

Thought 1: I need to search Apple Remote and find the program it was originally designed to interact with.
Act 1: Search[Apple Remote]
Obs 1: The Apple Remote is a remote control introduced in October 2005 by Apple … originally designed to control the Front Row media center program …

Thought 2: Apple Remote was originally designed to control the Front Row media center program. I need to search Front Row next and find what other device can control it.
Act 2: Search[Front Row]
Obs 2: Could not find [Front Row]. Similar: ['Front Row Seat to Earth', 'Front Row Motorsports',' Front Row (software) ', …]

Thought 3: Front Row is not found. I need to search Front Row (software) .
Act 3: Search[Front Row (software)]
Obs 3: Front Row is a discontinued media center software …

Thought 4: Front Row (software) is controlled by an Apple Remote or the keyboard function keys. So the answer is keyboard function keys.
Act 4: Finish[keyboard function keys]

How is Qwen Trained?

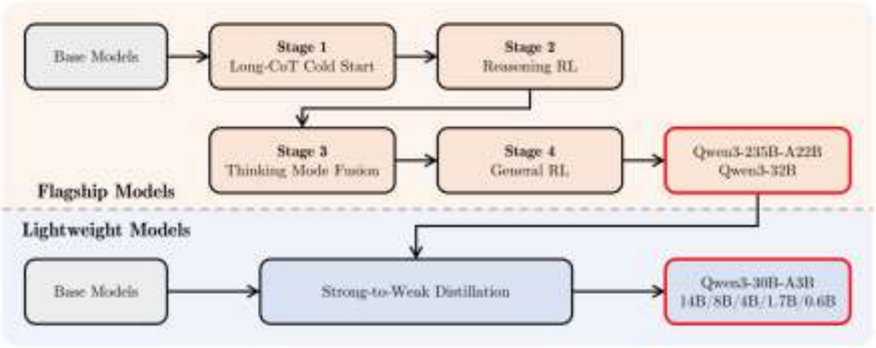
Pretrain

- Pretrains standard next-token prediction on diverse data, and also includes multi-task instructions during pretraining
  - Blurs boundary between pretraining & SFT; intuitively, primes network better for steerability while SFT data is still more curated & higher quality
- Does not tie input embedding and output projection (separate $embDim \times vocabSize$ matrices, unlike other approaches), to trade off efficiency for performance
- Training context length is 2048, but uses RoPE + dynamic NTK + windowed attention to allow good eval context of up to 8192
  - NTK-aware interpolation interpolates RoPE positions correctly, so that at test time a longer context still works without "positional sense distortion"
- Like other modern LLMs, uses pre-normalization (norm before attention) and RMSNorm instead of layernorm
  - $layernorm(x) = \frac{x-\mu}{\sigma} * \gamma + \beta$
  - $RMSNorm(x) = \frac{x}{\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2 + \epsilon}} * \gamma$
  - Intuitively, RMS is like the norm, but factors in the number of dimensions. Takeaway is that controlling & ensuring the scale is close to 1 is more important than making sure that the logits are centered

PostTrain

- SFT done using human-like conversational data
- Two stages in training reward model for RLHF (optimized using PPO)
  - First, PMP (preference model pretraining) trains model on large-scale paired preference data that is noisier
  - Then, reward model is obtained by fine-tuning PMP on higher quality data
- Uses ReAct prompting with fine-tuning to enable tool use

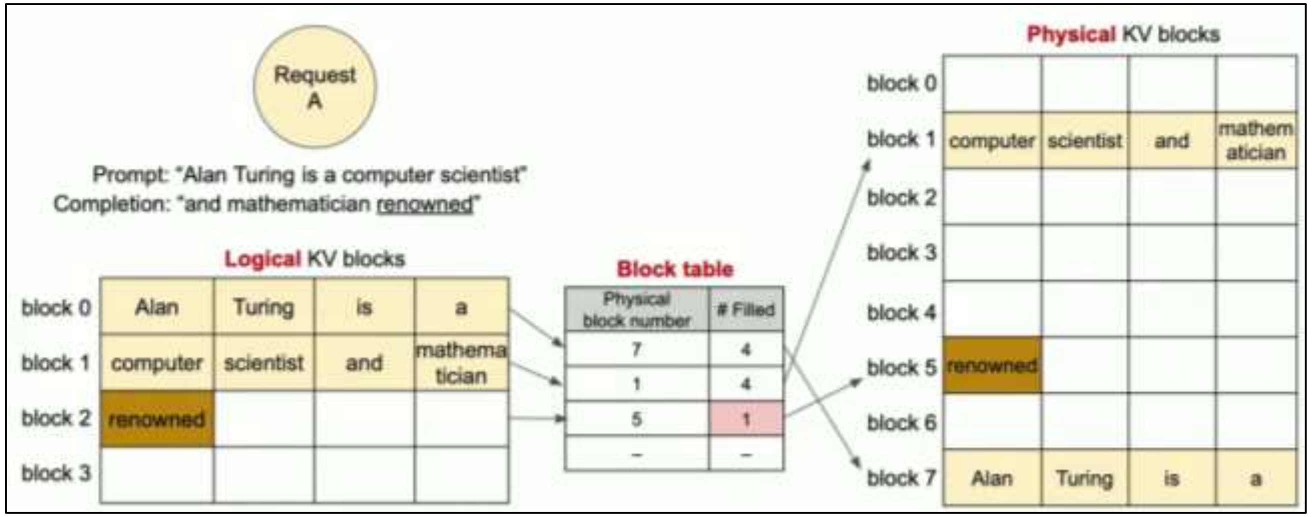
Improvements of Qwen 3

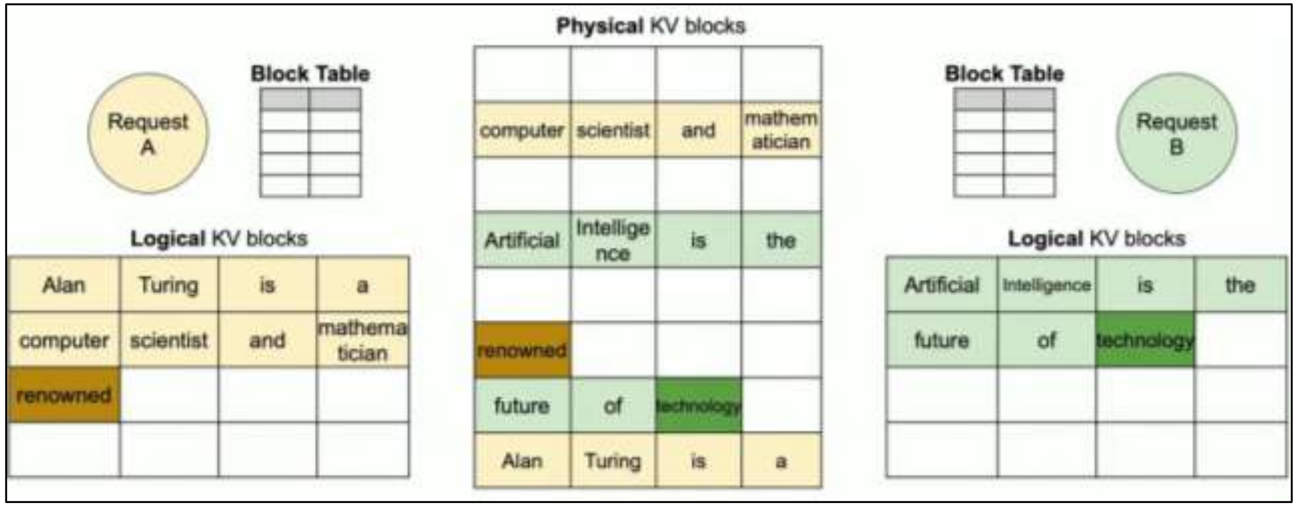


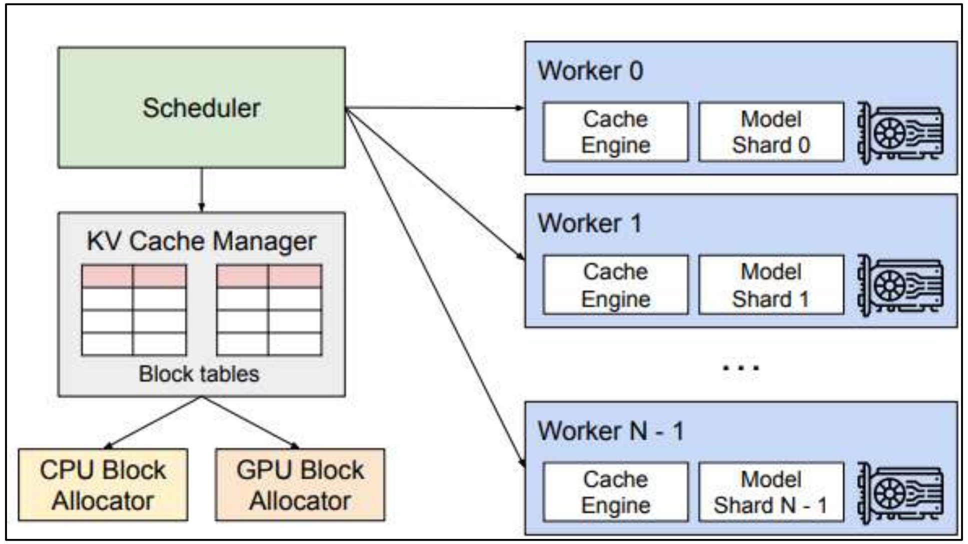
- Architecture: RoPE, SwiGLU, RMSNorm, QK-Norm, YARN, Dual Chunk Attenti
- Has MoE & dense variants
- Pretraining has three stages
  - **General:** 30T tokens, sequence length 4096
  - **Reasoning:** 5T tokens emphasizing STEM, sequence length 4096
  - **Long Context:** billions of tokens in long context corpora, sequence length 32768.
- Posttraining has four stages
  - **Long CoT Cold Start:** Standard SFT on filtered verifiable STEM dataset that has high quality annotated chain of thought / reasoning trace examples
  - **Reasoning RL:** Additional query-verifier pairs trained using GRPO
  - **Fusion:** Introduces "/think" "/no think" flag that enables dynamic switching of behavior based on user behavior within one model, trained using SFT
  - **General RL:** Align to human preferences. Doesn't state algorithm, but likely GRPO or DPO.
- To get smaller models, uses a strong-to-weak distillation pipeline, with two steps
  - **Off-Policy Distillation:** student learns/imitates from data generated by a teacher model
    - High quality supervision, very sample efficient, easy to train
  - **On-Policy Distillation:** student generates its own responses, and learns by comparing its output to the teacher's (using KL divergence between logits)
    - Student & teacher prompts sampled during training time
    - More exploration, more adaptative

Explain how PagedAttention works.

- PagedAttention (implemented via the vLLM library) is a memory management strategy for serving LLMs, to further improve KV caching
- LLMs are unlike traditional deep learning workloads, b/c its KV cache can dynamically grow/shrink over time as the model generates new tokens, and its lifetime and length are not known a priori
  - This causes naïve implementations to have contiguous pre-allocations based on max context lengths that cause wasteful fragmentation of memory
- PagedAttention is inspired by OS virtual memory paging, where contiguous physical memory is split up into blocks that can store a predefined number of keys/values for tokens, eg 16







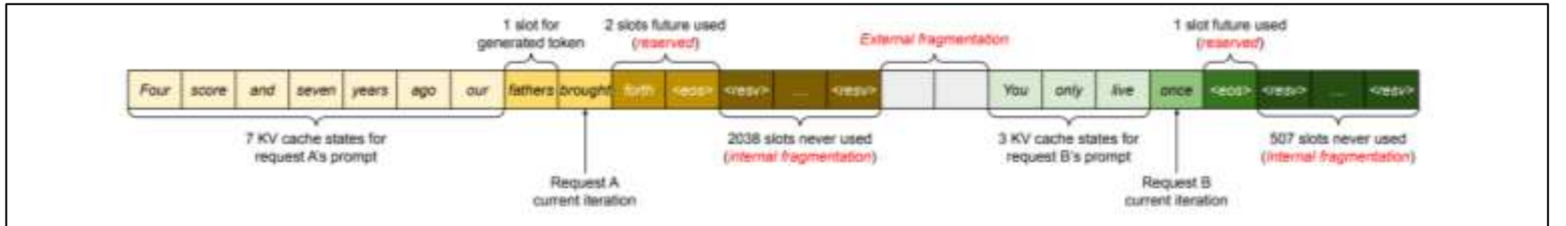


Figure 3. KV cache memory management in existing systems. Three types of memory wastes – reserved, internal fragmentation, and external fragmentation – exist that prevent other requests from fitting into the memory. The token in each memory slot represents its KV cache. Note the same tokens can have different KV cache when at different positions.

What is FiLM conditioning?

- FiLM (Feature-wise Linear Modulation) is a simple & general way to condition neural networks
- Works by performing feature-wise scaling/shifting on feature maps
- In original paper, parameters are from an text-conditioned RNN and applied to VQA, but this is a general method
  - Parameters can be predicted from a transformer, MLP, etc
  - Conditioning does not need to be from text
- Overall, lightweight, interpretable, and penetrative (eg, can condition many laters throughout network) method to do conditioning

A **FiLM layer** applies:

$$\text{FiLM}(F_{i,c} \mid \gamma_{i,c}, \beta_{i,c}) = \gamma_{i,c} F_{i,c} + \beta_{i,c}$$

Where:

- $F_{i,c}$: feature map for sample i and channel c
- $\gamma_{i,c}, \beta_{i,c}$: modulation parameters from a FiLM generator, often an RNN encoding a question

This allows question embeddings to **dynamically reprogram the visual feature extractor**.

Question Goes Here

- Answer goes here

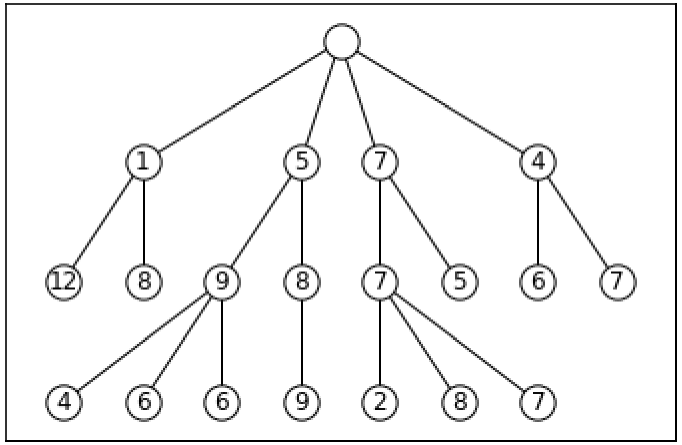
LLM Data Processing

What are the trade-offs of a larger or smaller vocabulary size?

- Larger vocabulary:
  - More compression, needs fewer tokens to express thoughts (cheaper cost of serving)
  - Improves context efficiency
  - Slower softmax, needs more memory due to larger embedding matrix
  - Can be harder to learn/generalize; leads to more rare tokens
- Smaller vocabulary:
  - Less compressed, needs more tokens to express thoughts
  - Less context efficiency
  - Faster softmax, smaller embedding matrix
  - More universal/generalizable
- Examples:
  - GPT-4 – around 100k tokens
  - Qwen – around 152k tokens

What are some common LLM decoding strategies?

- **Greedy Decoding**
  - At each step, select token with highest probability
  - Pros: Fast, deterministic
  - Cons: Can miss globally better sequences due to greedy local decisions; can result in repetitive or generic text
- **Beam Search**
  - Maintains k (beam_width) best sequences at each time step
  - Practical middle ground between greedy search and exhaustive search (since while the search space is exponential, beam search is a heuristic that restricts exploration to a linear slice of that space)
  - Pros: balances exploration and exploitation; produces higher-likelihood sequences than greedy decoding.
  - Cons: More computationally expensive. Large beams can sometimes decrease diversity.
- **Top-k Sampling (Truncated Sampling)**
  - At each step, restrict sampling to the top k most probable tokens
  - Pros: Introduces randomness and better diversity; prevents getting “off-manifold” & things going off the rails by not selecting rare tokens
  - Cons: Sensitive to choice of k, can still be repetitive/low-quality
- **Top-p Sampling (Nucleus Sampling)**
  - Select smallest set of tokens whose total probability exceeds p (eg p=0.9), then sample from that set
  - Pros: more adaptive than top-k, since the size of candidate pool changes. Can lead to more fluent/natural generation than fixed top-k
  - Cons: Sensitive to choice of p



Beam search examp

Explain the difference between a transformer LLM’s training & inference, with regards to how data is processed.

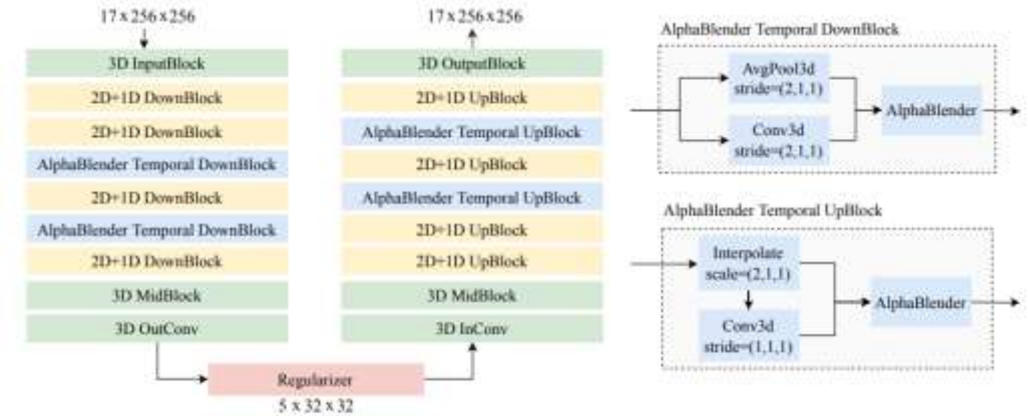
- **During training**, we compute a full query matrix — one query vector per token — because we have access to the entire input sequence
  - This allows the model to make next-token predictions at every position in parallel.
  - Combined with causal masking (which prevents attending to future tokens), this enables efficient batched forward and backward passes across full sequences.
  - This maximizes utilization of each training sample and GPU parallelism.
  - Computing next-token predictions at every position gives you T gradient signals from a single sequence of length T
- **During inference**, we generate one token at a time. At each step, we only compute a single query vector (from the most recent token), and use cached keys and values from previous steps for efficiency. This enables iterative autoregressive generation.

What are the differences between continuous and discrete tokens for transformers?

- Continuous tokens:
  - Higher fidelity, smooth latent space
  - No quantization errors
  - Simpler
  - Can lead to error accumulation/drift during autoregressive generation
  - Harder integration with LLMs (they're designed for next-token classification, not regression)
- Discrete tokens:
  - Compact, interpretable
  - Enables symbolic reasoning
  - No cumulative errors (each prediction is snapped, mitigating small errors from compounding)
  - Storage efficiency
  - Can lead to quantization error
  - Can lead to codebook collapse or under-utilization of codeobook

How does the VidTok video tokenizer work?

- Video tokenizer that converts raw video frames to compact latent tokens (both continuous or discrete)
- Architecture blend of 1D, 2D, and 3D convs, encoder-decoder style to get continuous tokens
- Uses **Finite Scalar Quantization (FSQ)** to turn continuous tokens to discrete, instead of codebook-based vector quantization
    - Lookup free – avoids codebook collapse and improves utilization
    - Works by quantizing/rounding each scalar element independently to a small set of pre-defined values
    - Implicit codebook; just all possible scalar values for each channel



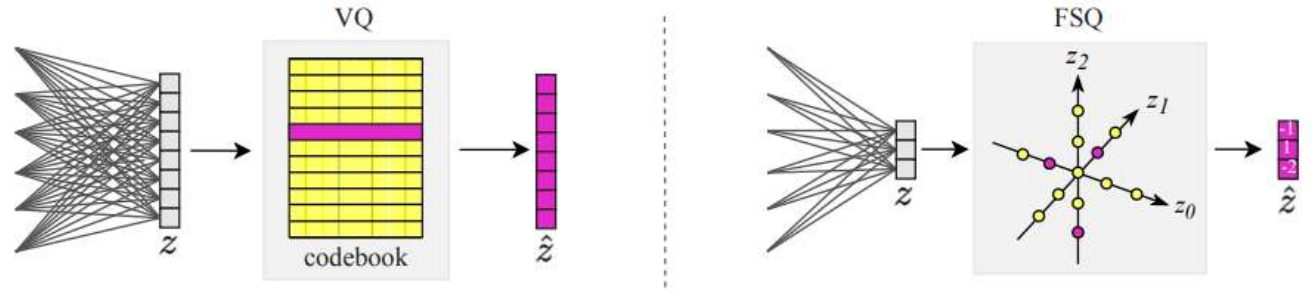


Figure 4: Left: Vector Quantization (VQ) employed in Vector Quantised-Variational AutoEncoder (VQ-VAE) (Van Den Oord et al., 2017). Right: Finite Scalar Quantization (FSQ) (Mentzer et al., 2024) utilized in our model.

Question Goes Here

- Answer goes here

NLP Metrics

How is perplexity calculated?

Perplexity measures the **inverse probability of the test set**, normalized by the number of tokens:

$$\text{Perplexity} = \exp\left(-\frac{1}{N}\sum_{t=1}^{N}\log P(w_t \mid w_1, \ldots, w_{t-1})\right)$$

- N: total number of tokens in the sequence
- w_t: the token at position t
- $P(w_t \mid w_1, \ldots, w_{t-1})$: probability assigned by the model to token w_t given the prior tokens

So, **perplexity is essentially** $e^{\text{cross-entropy}}$.

- Usually used to measure base model accuracy
- Cross entropy is in units of bits; the exp is for interpretability
  - For example, a perplexity of 10 means that the model is as uncertain as if it were choosing uniformly from 10 possible options

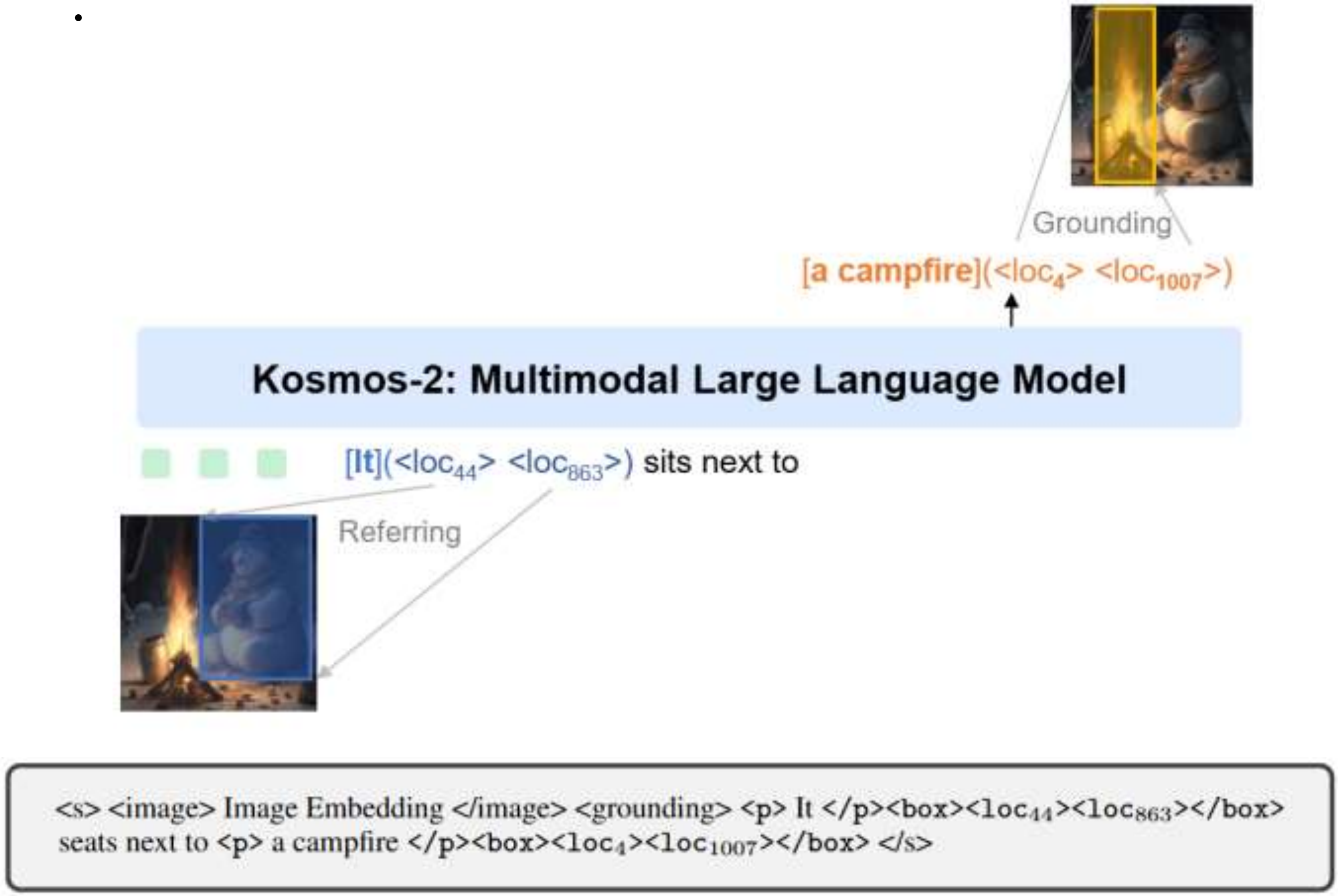
Question Goes Here

- Answer goes here

Vision Language Models

Describe the KOSMOS-2 model.

- Multimodal LLM designed to ground text to images by linking phrases to bounding boxes
  - Here, grounding refers to explicitly establishing connections to real-world natural language concepts; can mitigate hallucinations & provide interpretability
- Trained on new GRIT dataset, with 90M+ image-text-bbox pairs
- Uses special tokens to relate text and image regions; uses prompt templates to train models to use these tokens
  - Overall trained on image-text pairs, monomodal text corpora, image-caption pairs, and interleaved image-text data
  -



- "What is <p> it </p><box><loc$_1$><loc$_2$></box>? It is {*expression*}."
- "What is <p> this </p><box><loc$_1$><loc$_2$></box>? This is {*expression*}."
- "Describe <p> this object </p><box><loc$_1$><loc$_2$></box>. This object is {*expression*}."
- "<p> It </p><box><loc$_1$><loc$_2$></box> is {*expression*}."
- "<p> This </p><box><loc$_1$><loc$_2$></box> is {*expression*}."
- "<p> The object </p><box><loc$_1$><loc$_2$></box> is {*expression*}."

Table 9: Instruction templates used for expression generation.

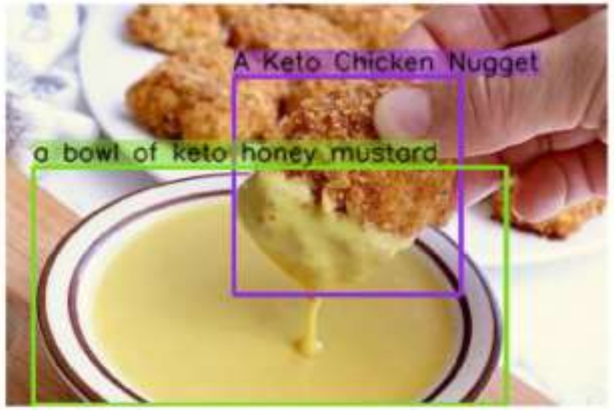
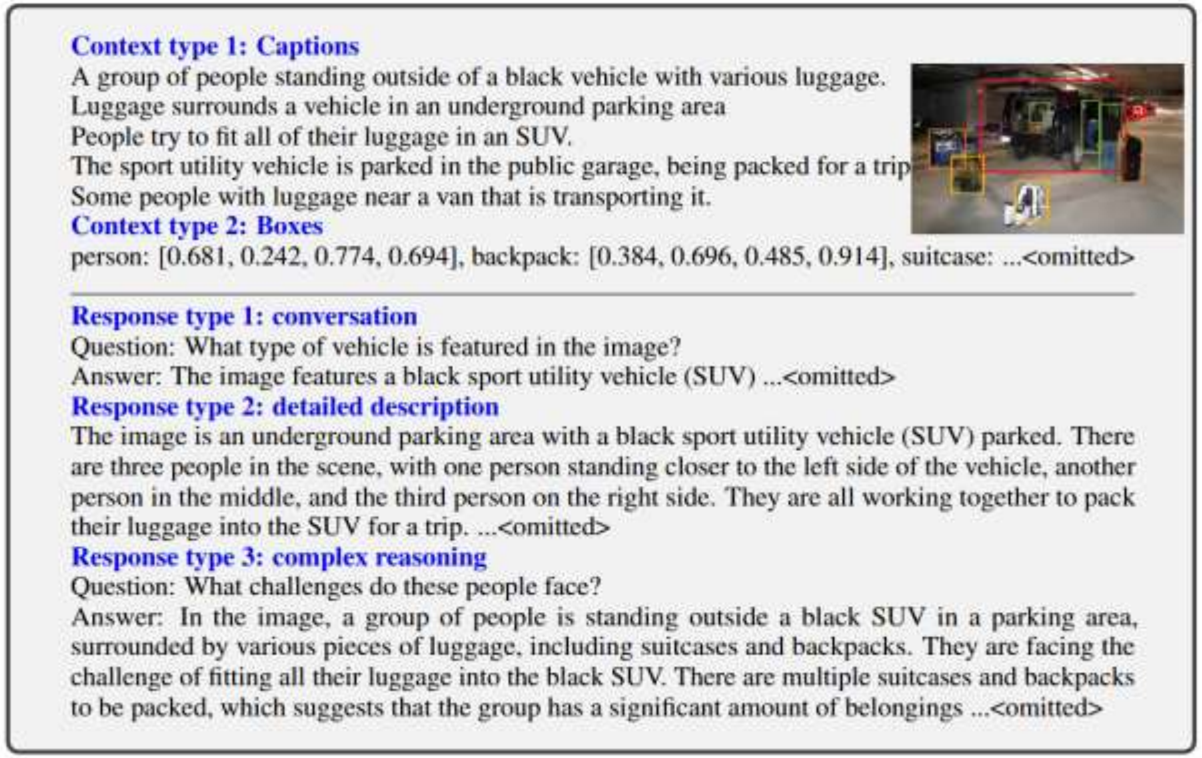


Figure 7: Example from GRIT. Caption: "*A Keto Chicken Nugget being dipped into a bowl of keto honey mustard.*".

Describe the LLaVA model.

- Endows a text-only LLM with image reasoning abilities via instruction tuning, to build a general-purpose visual assistant
- First, a **synthetic multi-turn VLM dataset** is generated by taking captioned image data & leveraging text-only GPT-4
    - Feeds in captions & bboxes as input
    - Asks conversation, description, and reasoning questions to get answers
    - These then become GT for the LLaVA dataset
- Then, we take the text-only Vicuna model (a fine-tuned LLaMA), add a CLIP-pretrained ViT visual encoder & projection layer. This is trained in two stages:
    - **First Stage: Caption Generation**
        - Train projection layer W only, on 500k captioned images to predict captions
        - Effectively trains visual tokenizer, that maps image features into token features
    - **Second Stage: Instruction Following (Visual Reasoning & Dialogue)**
        - Trains projection layer W and LLM parameters on synthetic multi-turn VLM dataset
    - ViT encoder is always frozen throughout; only used as a feature extractor



Context type 1: Captions
A group of people standing outside of a black vehicle with various luggage.
Luggage surrounds a vehicle in an underground parking area
People try to fit all of their luggage in an SUV.
The sport utility vehicle is parked in the public garage, being packed for a trip
Some people with luggage near a van that is transporting it.
Context type 2: Boxes
person: [0.681, 0.242, 0.774, 0.694], backpack: [0.384, 0.696, 0.485, 0.914], suitcase: ...<omitted>

Response type 1: conversation
Question: What type of vehicle is featured in the image?
Answer: The image features a black sport utility vehicle (SUV) ...<omitted>
Response type 2: detailed description
The image is an underground parking area with a black sport utility vehicle (SUV) parked. There are three people in the scene, with one person standing closer to the left side of the vehicle, another person in the middle, and the third person on the right side. They are all working together to pack their luggage into the SUV for a trip. ...<omitted>
Response type 3: complex reasoning
Question: What challenges do these people face?
Answer: In the image, a group of people is standing outside a black SUV in a parking area, surrounded by various pieces of luggage, including suitcases and backpacks. They are facing the challenge of fitting all their luggage into the black SUV. There are multiple suitcases and backpacks to be packed, which suggests that the group has a significant amount of belongings ...<omitted>

Example of synthetic data generated fro

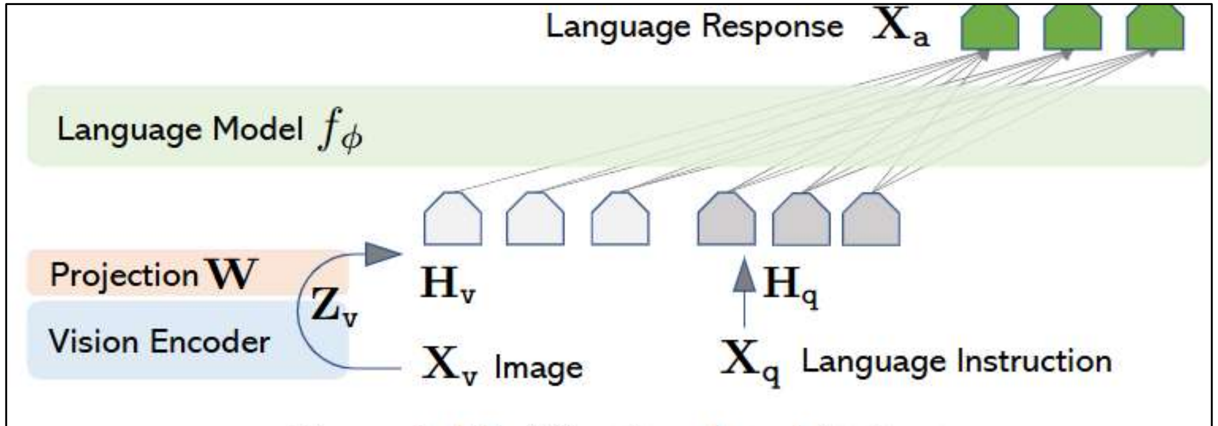


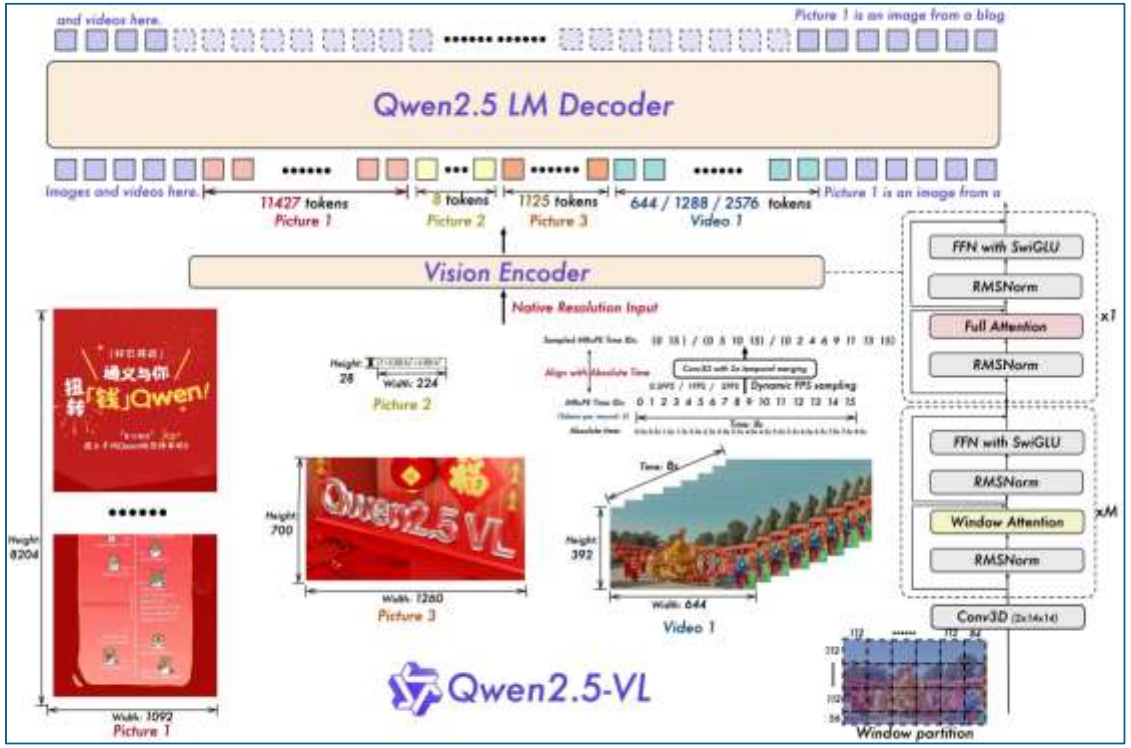
Figure 1: LLaVA network architecture.

Describe how Qwen 2.5-VL works.

- Pretraining stages (Image caption uses CLIP, rest uses standard next token prediction objective)

Stages	Visual Pre-Training	Multimodal Pre-Training	Long-Context Pre-Training
Data	Image Caption Knowledge OCR	+ Pure text Interleaved Data VQA, Video Grounding, Agent	+ Long Video Long Agent Long Document
Tokens	1.5T	2T	0.6T
Sequence length	8192	8192	32768
Training	ViT	ViT & LLM	ViT & LLM

- Posttraining stages (in both stages, ViT is frozen)
  - SFT
    - Includes grounding data (pointing / bbox data), VQA, document understanding, coding, agent GUI tasks, etc
    - Data format is more curated instruction-following, compared to pretraining
    - 50% pure text, 50% multimodal
  - DPO
    - Focuses on image-text & pure text data to align with human preferences
    - Rejection sampling is used to ensure quality of positive examples



What are VLAs?

- Vision Language Action (VLA) models are VLMs that can also directly autoregressively generate robot actions
- End-to-end framework, that doesn't separate perception, understanding, planning, and execution

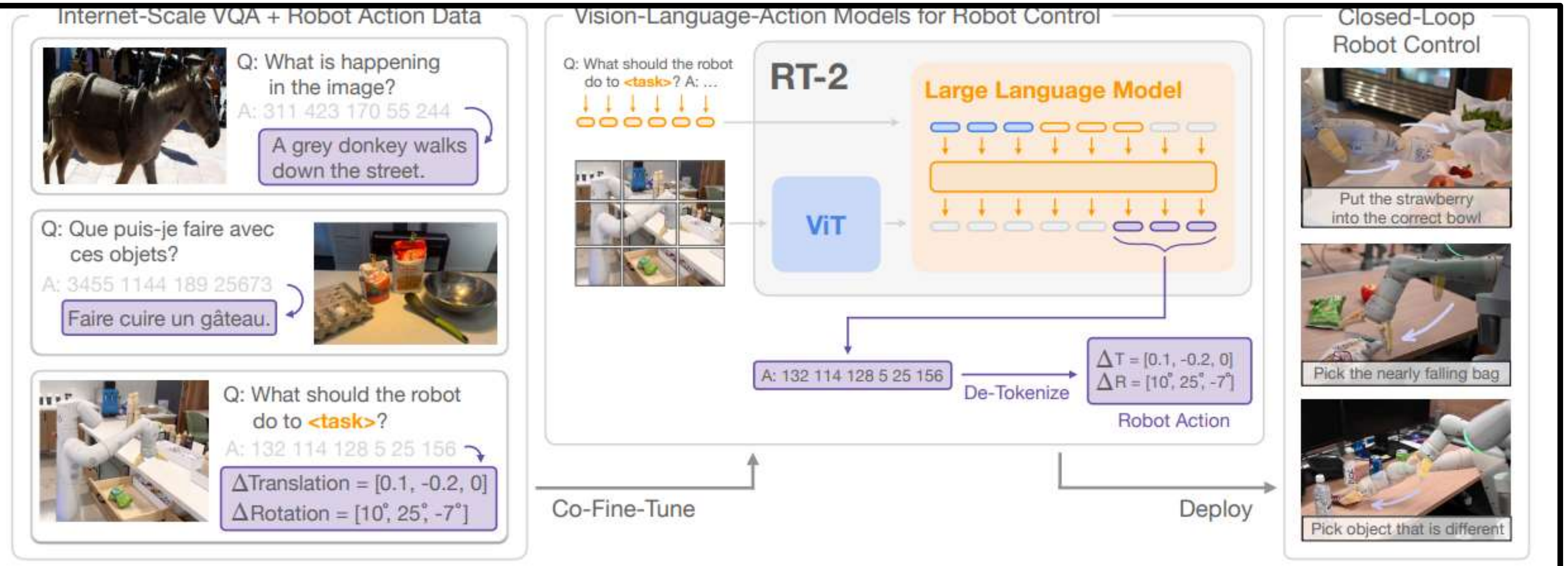


Figure 1 | RT-2 overview: we represent robot actions as another language, which can be cast into text tokens and trained together with Internet-scale vision-language datasets. During inference, the text tokens are de-tokenized into robot actions, enabling closed loop control. This allows us to leverage the backbone and pretraining of vision-language models in learning robotic policies, transferring some of their generalization, semantic understanding, and reasoning to robotic control. We demonstrate examples of RT-2 execution on the project website: robotics-transformer2.github.io.

Question Goes Here

- Answer goes here

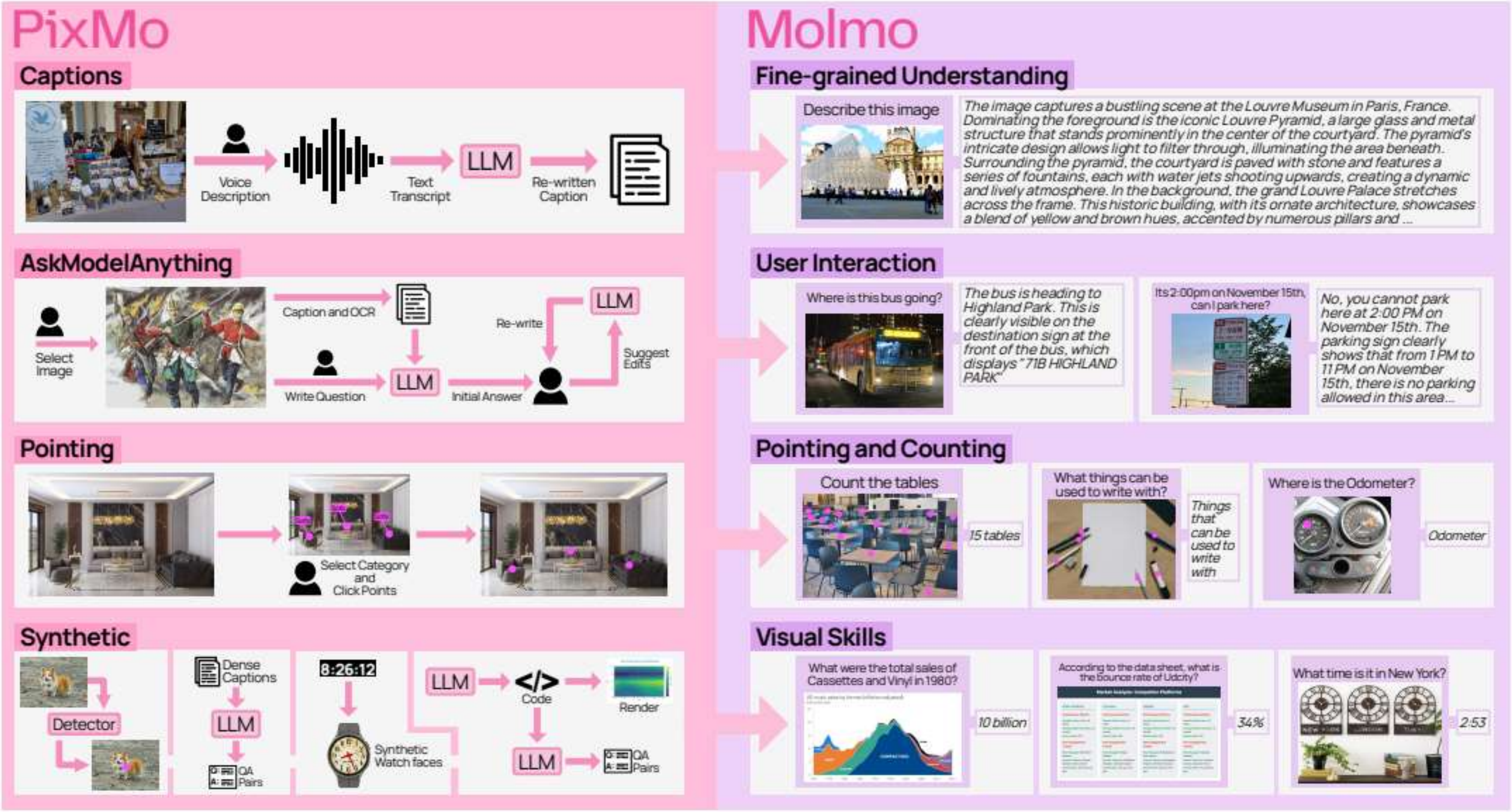


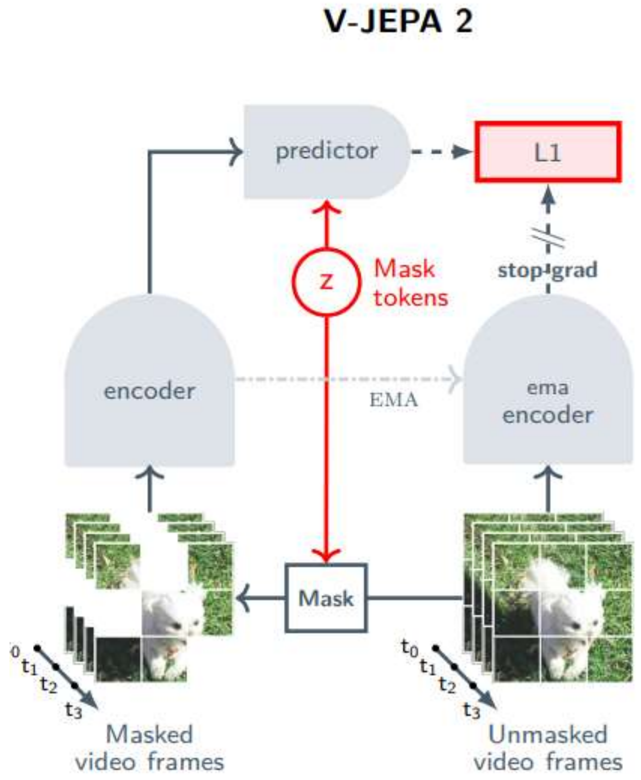
Figure 1. Datasets in **PixMo** (left) and the capabilities they enable in **Molmo** (right). PixMo consists of three annotated datasets and four synthetic datasets, all constructed without the use of VLMs. The annotated datasets include: dense captions (for pre-training), instruction following (for fine-tuning), and pointing (for fine-tuning, to support grounding and counting). The four synthetic datasets augment these datasets by targeting additional skills (*e.g.*, clock reading, document understanding).

What is teacher forcing, and what are its implications? What are some ways to address?

- When training LLMs, teacher forcing is the practice of giving the model GT tokens from the previous timestep as input, rather than using its own predictions.
- Pros
  - Makes training more stable, guarantees “correctness” in input during training
  - Errors don’t compound
  - Simpler to implement, & enables parallel compute
- Cons
  - At inference we do autoregressive generation, not teacher forcing; can lead to a train-test mismatch
  - As a result, can at test time go "off-manifold" if a bad token is sampled, and degenerate into bad trajectories
- Mitigations
  - **Data augmentation/noising**
  - **Scheduled sampling**: curriculum learning technique where with some increasing probability, sample predicted tokens instead of GT
  - **Reinforcement Learning**
    - In RL, model generates full sequences and that is evaluated wholistically on a distribution level – not on a per-token basis
    - Better long-term credit rewarding: token level supervision can’t teach the model the concept that early mistakes lead to bad outcomes, while RL can

How is V-JEPA 2 trained?

- V-JEPA2 (Joint Embedding Predictive Architecture) is a video self-supervised model trained on internet-scale video data
- Encoder extracts embeddings from input video clip patches; these are mased out, and a predictor predicts embeddings for the masked patches
    - Encoder and predictor are separate ViTs
    - Predictor is supervised by an EMA encoder teacher
- Frozen V-JEPA2 encoder also used to provide video embeddings, alongside a projection/connector layer, to tokenize and use as input to a VLM for VQA.
- This student/teacher EMA + stop-grad setup is also used in other self-supervised methods (DINO, BYOL) to provide a moving target, and prevent collapse(like predicting the same embedding, causing a trivial loss of 0



What are the contributions from Molmo?

- Molmo is a fully open-source VLM (no distilling or use of propriety VLMs) based on the LLaVA design.
- Components:
  - A preprocessor converts the image into multiscale, multi-crop images
  - A ViT encoder computes per-crop features (independently)
  - Patch features are generated by pooling features and a connector MLP maps to the LLM’s embedding space
- Some tricks/notes:
  - There is overlap in the cropping to allow for more context
  - Token dropout in the GT answer/target text, to incentivize looking at and attending to the image more
  - For caption generation, 90% of the time, the prompt includes a length hint to guide the model’s output length (improves pre-training quality)
  - Trained on in-house PixMo dataset and internet datasets. Some internet-gathered dataset have different styles; a style-prompt is used for certain datasets since their answers can have different quirks.

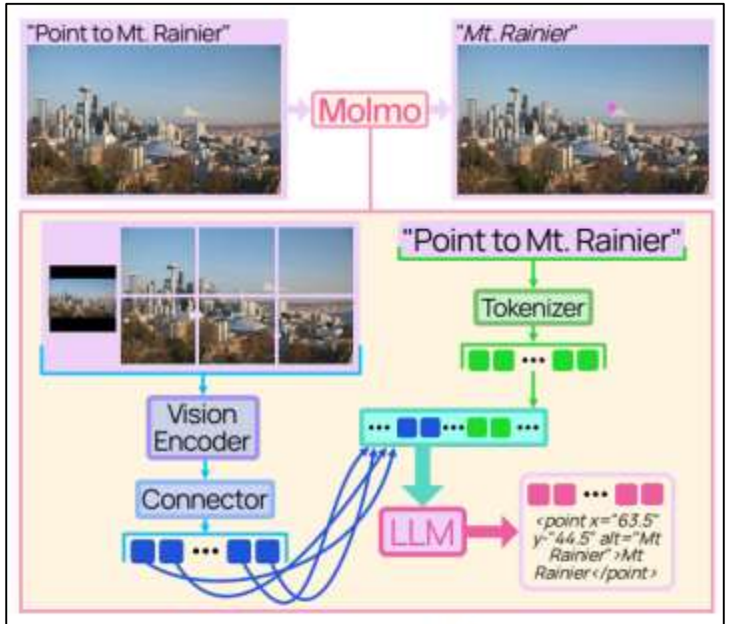
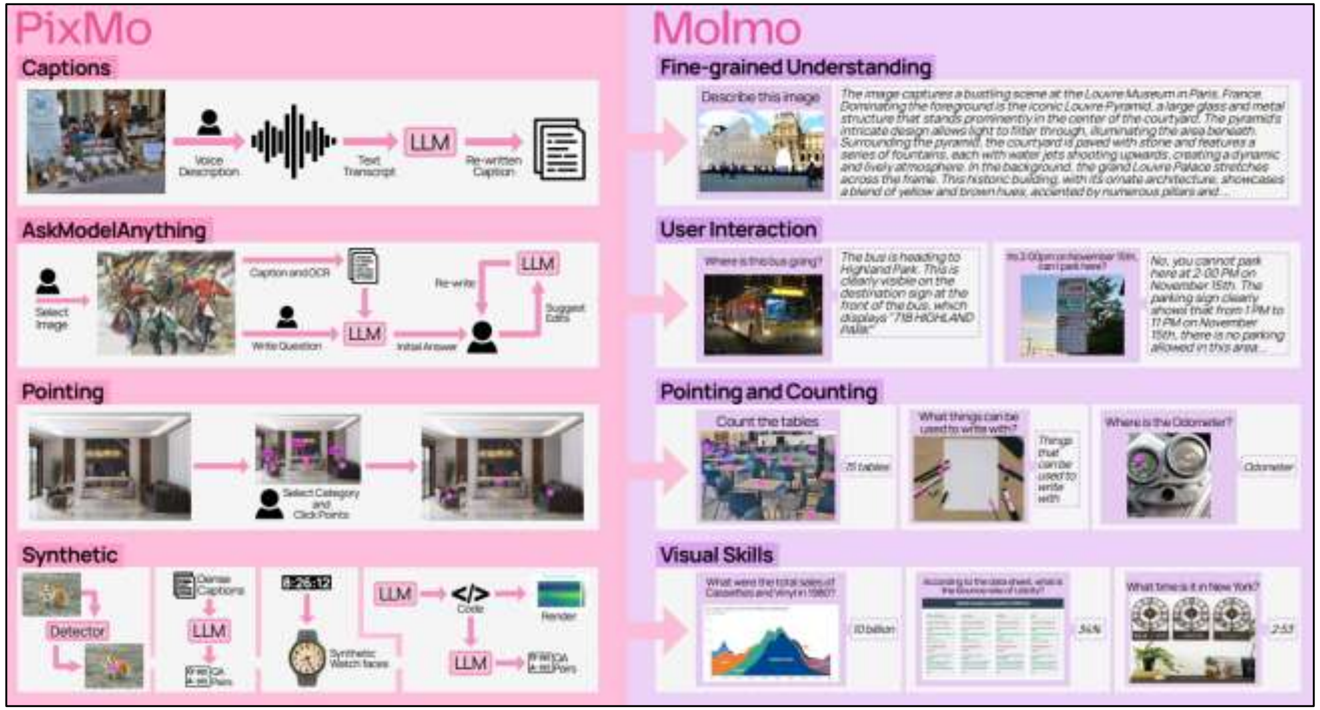


Figure 2. **Molmo** follows the simple and standard design of connecting a vision encoder and a language model.



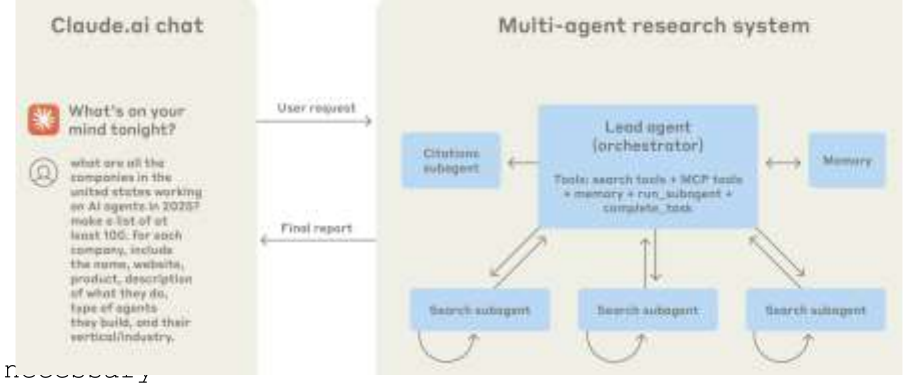
Question Goes Here

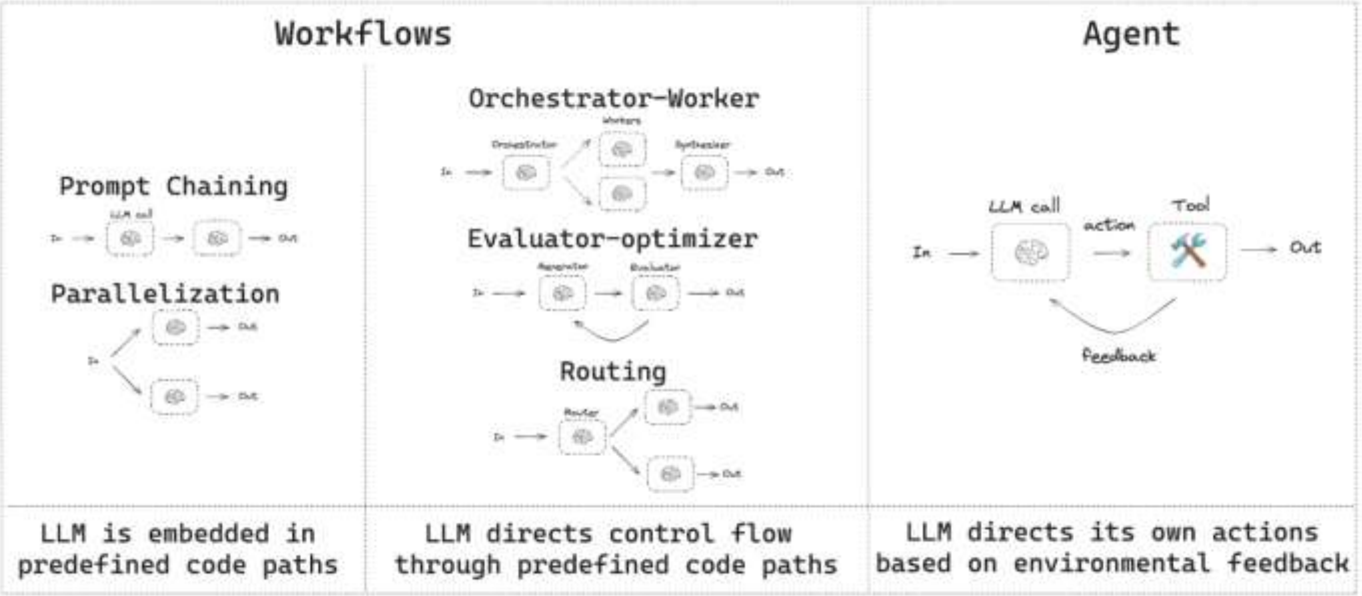
- Answer goes here

Agents

What is LangChain / LangGraph?

- LangChain: framework to connect LLMs with tools, memory, APIs, and data sources.
  - Used for more simple, but structured pipelines like RAG, chatbot, document summarizers, etc
- LangGraph: Module on top of LangChain to build cyclical, state graphs for LLM workflows
  - Can orchestrate multiple agents
  - Can have human in the loop & fault recovery; pausing, oversight, state inspection, and resumption after interruptions
  - Main benefits (eg over one-shot LLM)
    - Reflection/feedback loops to improve reliability & accuracy
    - Interpretable reasoning and more through logging
    - Human-in-the-loop guardrails
    - Extendibility, modularity, scalability, flexibility
  - Research Assistant example:
    - Planning agent plans sub-questions
    - Multiple agents search the web/documents and summarizes the results in parallel
    - Gaps/inconsistencies are reflected upon; queries are refined and looped back if necessary
    - Outputs full structured report
  - Code Review Example
    - Independent agents “experts” analyze performance, sec
    - Coordinator agent consolidates feedback and resolves
    - Routes to human reviewer
    - Auto generates commit comments for github





Question Goes Here

- Answer goes here

Question Goes Here

- Answer goes here

Question Goes Here

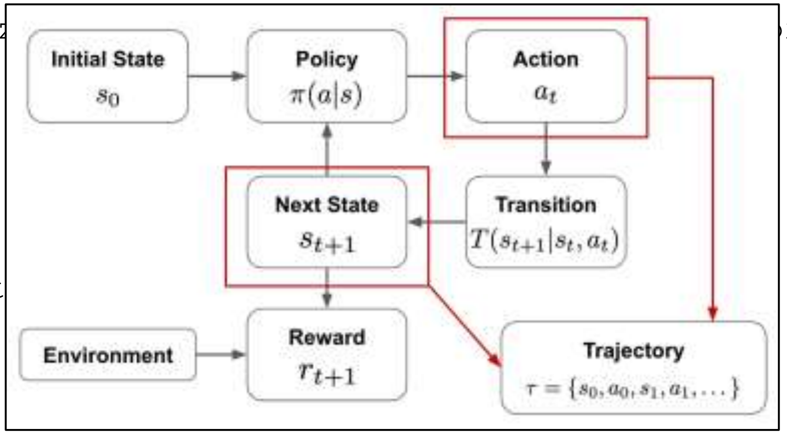
- Answer goes here

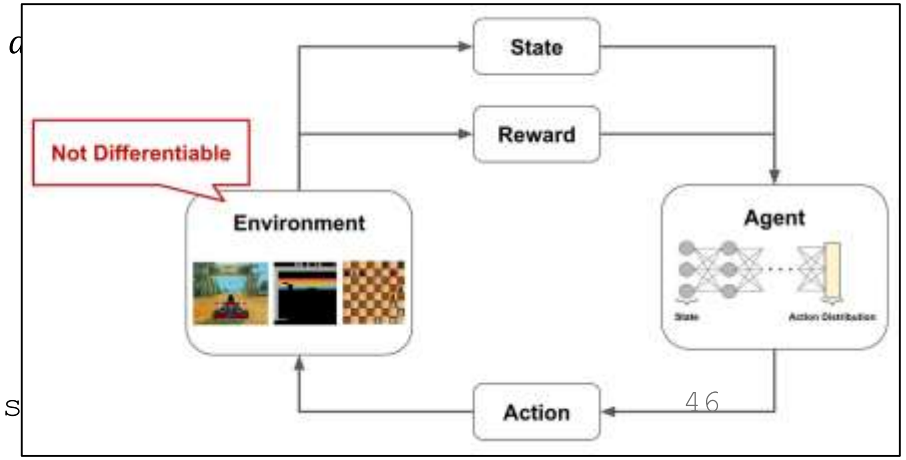
Reinforcement Learning

At a high level, what is the standard setup for training RL models?

- Similar to how humans learn, RL trains neural networks through trial and error
  - Producing outputs, receiving feedback, and adjusting weights
  - Agent learns to make decisions by interacting with an environment in order to maximize a reward signal over time
- RL generally the agent takes actions in an environment and indirectly receives delayed and sparse feedback in the for of rewards, without being told the correct action.
  - It's told how good the action was, but not the exact correct action; model has to discover this through exploration and learning
  - Generally environment is a black box and is not differentiable
  - In contrast, in supervised learning the feedback is direct & correct, and everything is typically backpropagable
  - Rewards is not associated with every action; they can have a long horiz ns to generate any positive reward
- Formally modeled by a Markov Decision Process (MDP)
  - $s \in S$, **State**
    - Set of all possible situations the agent can find itself in
    - Eg in robotics the position/velocity/orientation; in video game, t
  - $a \in A$, **Action**
    - Eg {up, down, left, right} or {accelerate, brake, turn left, etc}
  - $r_t \in \mathbb{R}$, **Reward from reward function** $R(s_t, a_t, s_{t+1})$
    - Gives agent a reward when transitioning from s_t to s_{t+1} via action a
  - $T(s_{t+1}|s_t, a|t)$, **Transition**
    - Defines dynamics of the environment
  - $\pi(a|s)$, **Policy**

  - $\tau \in \{s_0, a_0, s_1, a_1, \dots, s_t, a_t\}$, **Trajectory**
  - $R(\tau) = \sum_t \gamma^t r_t$, **Return**
    - $\gamma^t \in [0,1]$ is the discount factor, which can prefer near-term results





What are some strengths and weaknesses of RL?

- Strengths
  - **Learns by interaction:**
    - Through trial and error, ideal for sequential decision-making tasks
  - **No need for labeled data:**
    - Only need a reward signal, ideal for cases where labeling for direct GT is impractical or not well-defined
  - **Long term planning:**
    - Considers delayed rewards and can optimize actions over time, enabling strategic decision-making
  - **Optimizing for right objective:**
    - Generally, SFT only optimizes for a proxy of what you're actually trying to do. RL can optimize for that (eg human preference), even if it's a black box.
  - **Has had success in complex tasks, eg AlphaGo**

- Weaknesses
  - **Sample inefficiency:**
    - Requires many interactions with environment to learn effectively, which is usually only possible in sim. Due to large action/state spaces.
  - **Exploration vs. exploitation:**
    - Balancing trying new actions (exploration) vs. sticking with known good ones (exploitation) is non-trivial and can hinder learning.
  - **Reward design is hard:**
    - Poorly shaped or sparse rewards can lead to undesired behaviors or very slow learning.
  - **General instability and sensitivity**

What’s the difference between on-policy & off-policy RL?

On-policy (eg SARSA, policy gradient methods like TRPO, PPO, GRPO)

- Same policy used to generate behavior (collect data) and to improve itself
- Learns from data it generates by following its current policy
- Generally more stable training, but requires continual data collection as policy changes and less sample efficient since old data becomes stale

Off-policy (eg Q-learning, Soft Actor-Critic)

- Agent learns about one policy (target policy) using data collected from a different policy (behavior policy)
- More sample-efficient (can reuse past experiences, eg experience replay)
- Enables learning from humans/expert demonstrations
- More complex / tricky, can need corrections like importance sampling. But more flexible, maximizes use of data.

Derive the vanilla/basic policy gradient, and intuitively explain what it means.

$$\underset{\text{Policy}}{J(\pi_\theta)} = \mathbb{E}_{\overset{\text{Trajectory}}{\tau} \sim (\pi_\theta, \underset{\text{Transition Function}}{T})} \overset{\text{Return}}{[R(\tau)]}$$

$$\mathcal{D} = \{\tau_0, \tau_1, \dots\} \quad \text{(Sampled Trajectories)}$$

$$\overline{\nabla_\theta J(\pi_\theta)} = \frac{1}{|\mathcal{D}|} \sum_{\tau \in \mathcal{D}} \sum_{t=0}^{T} [\nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau)]$$

Using chain rule of probability and log-derivative trick:

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \nabla_\theta \mathbb{E}_{\tau \sim (\pi_\theta, T)} [R(\tau)] \\ &= \nabla_\theta \int_\tau P(\tau|\theta) R(\tau) \\ &= \int_\tau \nabla_\theta P(\tau|\theta) R(\tau)\end{aligned}$$

$$\theta_{t+1} = \theta_t + \underset{\text{Learning Rate}}{\alpha} \overbrace{\nabla_\theta J(\pi_\theta)|_{\theta_t}}^{\text{Gradient of objective}}$$

$$\underset{\text{Trajectory Probability}}{P(\tau|\theta_t)} = \overset{\text{Probability of Initial State}}{\rho(s_0)} \prod_{t=0}^{T} \underset{\text{Transition Function}}{T(s_{t+1}|s_t, a_t)} \overset{\text{Policy}}{\pi_{\theta_t}(a_t|s_t)}$$

- Tells us how to adjust the policy's parameters to make actions that led to good outcomes more likely in the future
- ∇_θ tells us how the probability of particular actions a_t change with respect to the model parameters θ
- Combined effect depends on $R(\tau)$
  - If positive, then we increase chance of choosing that action
  - If negative, then we decrease chance of choosing that action
- Gradient shifts probability mass to push agent to repeat lucky successes, based on observed outcomes
- Note, VPG is very data inefficient

Basic Policy Gradient

$$\boxed{\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim (\pi_\theta, T)} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) R(\tau) \right]}$$

Define On-Policy/Optimal Value/Action-Value functions, and how they lead to advantage functions.

- ***On-Policy Value Function***: $V^{\pi}(s) = \mathbb{E}_{\tau \sim (\pi, T)}\left[R(\tau) \mid s_0 = s\right]$
- Expected return if you start in state s and act according to policy π afterwards.

- ***On-Policy Action-Value Function***: $Q^{\pi}(s,a) = \mathbb{E}_{\tau \sim (\pi, T)}\left[R(\tau) \mid s_0 = s, a_0 = a\right]$
- Expected return if you start in state s, take some action a (may not come from the current policy), and act according to policy π afterwards.

- ***Optimal Value Function***: $V^{\star}(s) = \max_{\pi} \mathbb{E}_{\tau \sim (\pi, T)}\left[R(\tau) \mid s_0 = s\right]$
- Expected return if you start in state s and always act according to the optimal policy afterwards.

- ***Optimal Action-Value Function***: $Q^{\star}(s,a) = \max_{\pi} \mathbb{E}_{\tau \sim (\pi, T)}\left[R(\tau) \mid s_0 = s, a_0 = a\right]$
- Expected return if you start in state s, take some action a (may not come from the current policy), and act according to the optimal policy afterwards.

- The advantage function measures how much better/worse an action is compared to the average action in a given state

$$A^{\pi}(s,a) = \underbrace{Q^{\pi}(s,a)}_{\text{On-Policy Action-Value Function}} - \underbrace{V^{\pi}(s)}_{\text{On-Policy Value Function}} \quad \text{(Advantage Function)}$$

- Used in the bellman equations, which say that the value of a state is just the immediate reward plus the value of the future
- Encodes the recursive nature of reward accumulation and form the basis for how most RL algorithms incremental improve estimates.

$$V^{\pi}(s) = \mathop{\mathrm{E}}_{\substack{a \sim \pi \\ s' \sim P}} \left[r(s,a) + \gamma V^{\pi}(s') \right],$$

$$Q^{\pi}(s,a) = \mathop{\mathrm{E}}_{s' \sim P} \left[r(s,a) + \gamma \mathop{\mathrm{E}}_{a' \sim \pi} \left[Q^{\pi}(s',a') \right] \right],$$

$$V^{*}(s) = \max_{a} \mathop{\mathrm{E}}_{s' \sim P} \left[r(s,a) + \gamma V^{*}(s') \right],$$

$$Q^{*}(s,a) = \mathop{\mathrm{E}}_{s' \sim P} \left[r(s,a) + \gamma \max_{a'} Q^{*}(s',a') \right].$$

What is Generalized Advantage Estimation (GAE)?

$$A^{\pi}(s,a) = \underbrace{Q^{\pi}(s,a)}_{\text{On-Policy Action-Value Function}} - \underbrace{V^{\pi}(s)}_{\text{On-Policy Value Function}} \quad \text{(Advantage Function)}$$

- In general, we can't directly compute the advantage function – it's intractable
- Temporal distances (TD) only involve a one-step lookahead estimate, and everything is available at time t to get δ_t

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$$

- GAE takes this a step further & blends multiple estimates using decaying weights, to place more weigh on short-term signals

$$\hat{A}_t^{\mathrm{GAE}(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}$$

- In practice, a neural network V_ϕ regresses the value function using MSE loss
- Data source is sampled trajectories; tuples of $(s_t, a_t, r_{t+1}, s_{t+1})$
- The supervision target is $\hat{y} = r_{t+1} + \gamma V_\phi(s_{t+1})$
  - Bootstrapping technique: over time, will converge to a fixed point that satisfies the bellman equation
  - Not chicken-and-egg, even though V_ϕ appears on both sides of the target

$$\mathcal{L}(\phi) = (V_\phi(s_t) - \hat{y}_t)^2$$

What is Trust Region Policy Optimization (TRPO)?

- Motivated by how data inefficient VPG is, TRPO updates the policy under a constraint based on the KL divergence. Some notes:
  - Instead of performing gradient descent, we solve a constrained maximization problem to generate each new policy. Performs larger updates
  - KL divergence in parameter space prevents overly large steps
  - Term in expectation modified to express probability of an action as the ratio between old and updated policy
  - Advantage function is used instead of just the award function, to focus updates only on better-than-expected actions; helps policy learn to favor relatively good actions w/o overacting to randomness
- Follows an actor-critic model with 2 different neural networks
  - **Actor**
    - Represents policy $\pi_\theta(a|s)$, optimized via an estimate of the advantage
  - **Critic**
    - Approximates value function V, which estimates how good a state is under the current policy
    - Trained using regression (using temporal difference TD error / generalized advantage estimation GAE)
    - Helps train actor by providing advantage estimates

$$\theta_{k+1} = \mathbf{argmax}_\theta \; \mathbb{E}_{(s,a)\sim(\pi_{\theta_k},T)} \left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} \overbrace{A^{\pi_{\theta_k}}}^{\text{Advantage Function}}(s,a) \right]$$

$$\text{such that } \underbrace{\overline{D}_{\text{KL}}(\theta||\theta_k)}_{\text{KL Divergence}} < \delta$$

What is Proximal Policy Optimization (PPO)?

$$L^{\mathrm{CLIP}}(\theta) = \mathbb{E}_t\left[\min(r_t(\theta)A_t, \mathrm{CLIP}(r_t(\theta), 1-\epsilon, 1+\epsilon)A_t)\right] \qquad r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)}$$

- Improvement upon TRPO that is simpler & converges faster
- Like TRPO, uses ratio r_t; >1 when action is more probable under currently policy than old policy, and in [0,1) when less probable
- However, this ratio can lead to overly large gradient steps. One way to mitigate is to build stabilizing properties into a surrogate objective function
  - Clip limits ratio to a range, eg (0.8, 1.2)
  - We build an asymmetric feature via the clip & min components -- that if we do the right thing, don't take to big of a step, but if we do the wrong thing, we are free to take a larger step
- Because of the designed objective, we can run multiple epochs of gradient ascent on your samples. As we run more, we'll start hitting clipping limits, so training will gradually stop until we move onto the next iteration to collect new samples

Algorithm 1 PPO, Actor-Critic Style

for iteration$=1,2,\ldots$ **do**
  for actor$=1,2,\ldots,N$ **do**
    Run policy $\pi_{\theta_{\text{old}}}$ in environment for T timesteps
    Compute advantage estimates $\hat{A}_1,\ldots,\hat{A}_T$
  end for
  Optimize surrogate L wrt θ, with K epochs and minibatch size $M \le NT$
  $\theta_{\text{old}} \leftarrow \theta$
end for

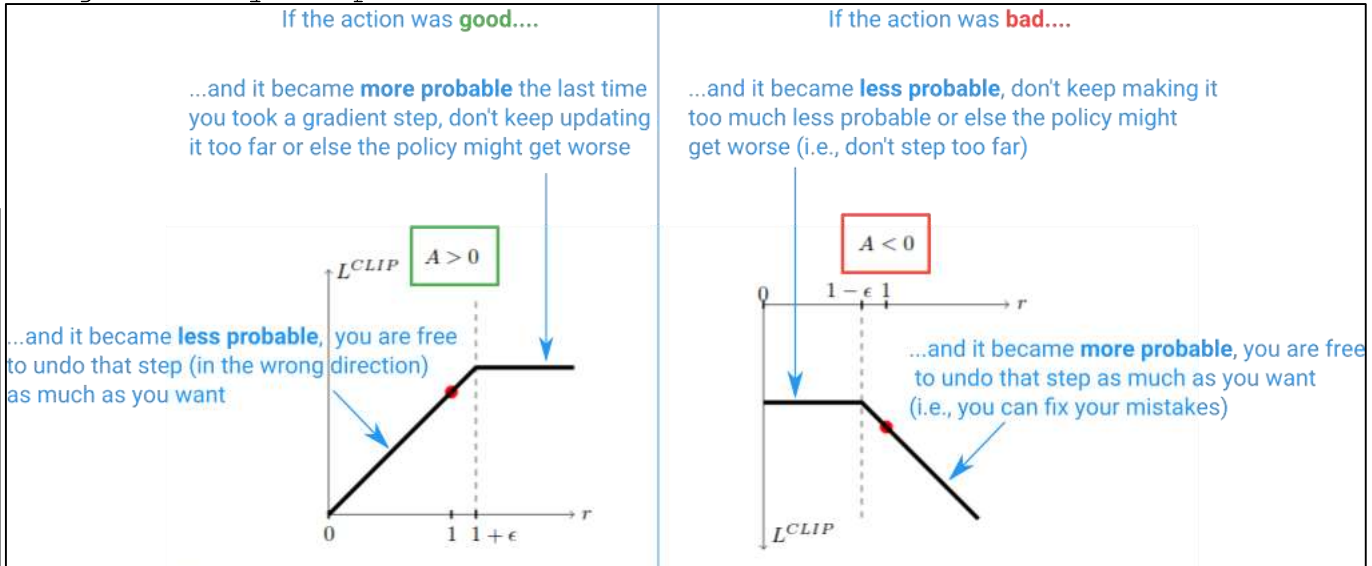
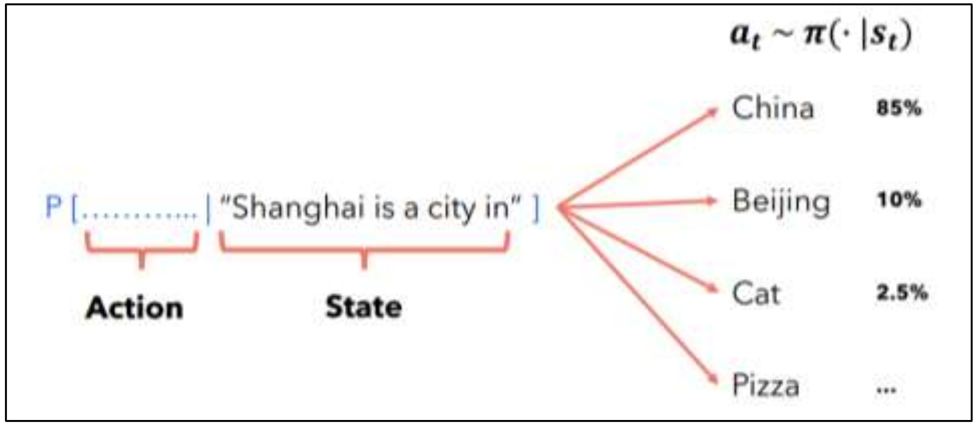


Figure 1: Plots showing one term (i.e., a single timestep) of the surrogate function L^{CLIP} as a function of the probability ratio r, for positive advantages (left) and negative advantages (right). The red circle on each plot shows the starting point for the optimization, i.e., $r = 1$. Note that L^{CLIP} sums many of these terms.

How can LLM alignment be formulated as an RL problem?

- RL allows us to align LLM outputs to human preferences and generate more helpful responses
  - In this final stage, we don't have concrete GT targets; there can be multiple plausible outputs, so the RL framework makes sense
  - Also, the autoregressive sampling process (eg greedy, beam search, top-k, etc) is not differentiable; output text is discrete
- Data: Instead of annotating raw reward scores of independent prompts/ flakey), annotate relative pairwise preferences with two answers
  - Tuples of $(prompt, x^+, x^-)$
- With data, train a transformer reward model R_ϕ to regress reward
  - Loss: $L_{RM} = -log\sigma\left(R_\phi(x^+) - R_\phi(x^-)\right)$
- Formulation:
  - **States**: The language model prompts
  - **Action**: The next token chosen autoregressively
  - **Policy/Agent**: The LLM itself; $\pi_\theta(a|s)$
  - **Episode**: prompt → sampled response → award



$$\hat{A}_t = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l}$$

- Optimization:
  - Sample prompts, and generate multiple responses per prompt
  - Compute their rewards using reward model
  - Compute advantage estimates: how much better a response was compared to expected using a learned critic model $V_\phi(s_t)$
    - Advantages $\hat{A}_t$ are per-response-timestep, and tell us how much better or worse that token choice was than expected
    - Estimates which tokens contributed more/less to the final award & a good/bad response, enabling credit assignment
    - Gives us a way to assign credit across timestamps even when only the full sequence is awarded
  - Update policy using PPO's surrogate objective (allows for small steps, less chance for reward hacking)

How does GRPO work?

$$\mathcal{J}_{GRPO}(\theta) = \mathbb{E}[q \sim P(Q), \{o_i\}_{i=1}^{G} \sim \pi_{\theta_{old}}(O|q)]$$
$$\frac{1}{G}\sum_{i=1}^{G}\left(\min\left(\frac{\pi_\theta(o_i|q)}{\pi_{\theta_{old}}(o_i|q)}A_i, \text{clip}\left(\frac{\pi_\theta(o_i|q)}{\pi_{\theta_{old}}(o_i|q)}, 1-\varepsilon, 1+\varepsilon\right)A_i\right) - \beta\mathbb{D}_{KL}\left(\pi_\theta||\pi_{ref}\right)\right)$$

$$A_i = \frac{r_i - \text{mean}(\{r_1, r_2, \cdots, r_G\})}{\text{std}(\{r_1, r_2, \cdots, r_G\})}$$

$$\mathbb{D}_{KL}\left(\pi_\theta||\pi_{ref}\right) = \frac{\pi_{ref}(o_i|q)}{\pi_\theta(o_i|q)} - \log\frac{\pi_{ref}(o_i|q)}{\pi_\theta(o_i|q)} - 1$$

- $q \sim P(Q)$: sampling question q from dataset
- $\{o_i\}_{i=1}^{G} \sim \pi_{\theta_{old}}(O|q)$: Sample G different outputs o_i from LLM
- $\pi_\theta(o_i|q)$: Probability of response o_i under the current LLM policy (obtained by multiplying the predicted token probabilities)
- $\pi_{\theta_{old}}(o_i|q)$: Probability of response o_i under an older version of the LLM policy, say 10 GRPO steps ago
- π_{ref}: A fixed reference model used throughout GRPO

3. Training Loop (High Level)

1. **Sample group trajectories** from the current policy.
2. **Compute rewards** for each trajectory.
3. **Normalize rewards to advantages** A_i within each group.
4. **Compute the GRPO loss** (clipped ratio + KL penalty).
5. **Backpropagate and apply gradient descent** to update θ.

- In **Group Relative Policy Optimization (GRPO)**, we want to adjust parameters θ of an LLM to maximize objective $J_{GRPO}(\theta)$. (In equation, expectation is in brackets)
- A_i is advantage of o_i in group; how much we should prefer the i-th response relative to others
  - r_i is reward for answer o_i for prompt q; simply 1 if right answer, 0 if wrong (this formulation called "sparse rewards"; dense per-token rewards also possible.
  - $mean(\{r_1, r_2, \ldots, r_g\})$ is called the **baseline**; normalizes for how "impressive" a particular response is compared to others
  - Std normalizes for "random guessing" vs a single special correct prompt. Hedges how seriously we should take prompt
  - Encapsulates the "Group" in GRPO; fundamentally A_i depends on these relative group comparisons; this approximation of advantage is much simpler than other RL methods
- Behavior of the "naïve" **policy gradient** $\pi_\theta(o_i|q) * A_i$:
  - If $A_i > 0$, we want to optimize to increase the probability $\pi_\theta(o_i|q)$; reinforcing this good behavior
  - If $A_i < 0$, we want to optimize to decrease the probability $\pi_\theta(o_i|q)$; making it less likely
- Other terms are scaffolding, hacks/tricks to hedge for the policy gradient:
  - We formulate a ratio $\pi_\theta/\pi_{\theta_{old}}$ to measure the diff from the past, and clip it by ϵ, so probability incentives are limited
  - Clip+min constraints incentivize focusing on others in group/expectation. This prevents **reward hacking** (regurgitating the same, highly scored response)
  - Reverse KL divergence (performed over token distributions over a trajectory) to incentivize mode-seeking.
- These hacks reflect a general conservatism to stay close to current beliefs.
  - RL technique can generally be fundamentally noisy due to the sparse signal we get after generating a long trajectory - unclear what led to 0/1 reward

What is Direct Preference Optimization

$$\mathcal{L}_{\text{DPO}}(\pi_\theta; \pi_{\text{ref}}) = -\mathbb{E}_{(x, y_w, y_l) \sim \mathcal{D}} \left[\log \sigma \left(\beta \log \frac{\pi_\theta(y_w \mid x)}{\pi_{\text{ref}}(y_w \mid x)} - \beta \log \frac{\pi_\theta(y_l \mid x)}{\pi_{\text{ref}}(y_l \mid x)} \right) \right]$$

- PPO can be difficult to use when aligning LLM models
  - Requires training a reward model
  - Training takes longer, need to generate/sample in-the-loop during training
  - Involves more complexity, RL machinery, tuning
- DPO allows us to align LLMs without RL
  - Directly use pairwise preference annotations and optimize using gradient descent
- In loss function, we have a negative log sigmoid, so we want the winning ratio to be high, and the losing ratio to be low
  - Probabilities $\pi(y|x)$ computed by multiplying per-token probabilities when running through LLM
  - Loss derived from the Bradley-Terry model, where $y_w > y_l$ reads as "y_w preferred to y_l"

$$P(y_w > y_l) = \frac{e^{r^*(x, y_w)}}{e^{r^*(x, y_w)} + e^{r^*(x, y_l)}}$$

- Some weaknesses of DPO compared to PPO:
  - Cannot train on arbitrary unannotated prompts (no reward models)
  - DPO is off-policy training (model not exposed to its own policy during training)
  - Limited to pairwise comparisons, cannot optimize scalar rewards

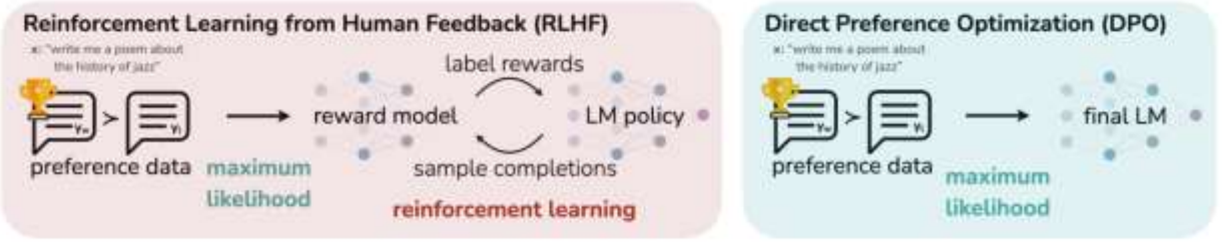
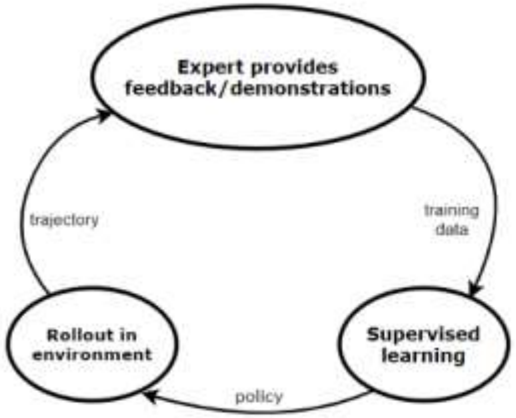
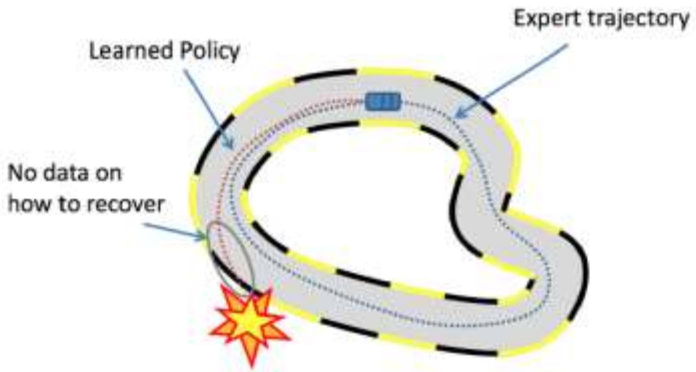


Figure 1: **DPO optimizes for human preferences while avoiding reinforcement learning.** Existing methods for fine-tuning language models with human feedback first fit a reward model to a dataset of prompts and human preferences over pairs of responses, and then use RL to find a policy that maximizes the learned reward. In contrast, DPO directly optimizes for the policy best satisfying the preferences with a simple classification objective, fitting an *implicit* reward model whose corresponding optimal policy can be extracted in closed form.

What is imitation learning?

- In imitation learning, an agent learns by observing & mimicking expert behavior
- Simplest form is **behavior cloning**, which uses supervised learning over a dataset of expert demonstrations (state-action pairs)
  - There are no explicit rewards, unlike RL. Treats the expert to learn the implicit reward function
  - Not trial-and-error based; does not have the same potential for superhuman performance that RL agents can discover
- Extensions use techniques from RL to improve generalizability
  - **Dagger (Dataset Aggregation)** iteratively gathers data, trains a policy, and deploys the policy in the environment to collect new states for the expert to annotate
  - **Inverse RL** learns a reward function to explain the expert's behavior, and then uses RL to find a policy
- Applicability
  - IL: Scenarios where solution is complex but can be demonstrated by expert. Reward is sparse and not well defined
  - RL: Good for problems where reward structure is clearly defined but solution is complex/unknown





What are some trade-offs and differences between Offline RL (eg DPO) and Online RL (eg PPO, GRPO)?

- **Offline RL (eg DPO)**
  - Can incorporate human preferences (unlike SFT), eg for tone, helpfulness, safety
  - Doesn’t require reward model
  - More sample efficient
  - Stable and tractable to train
  - Uses relative preference data, not absolute scores
  - Limited to existing data, can’t explore new behaviors
  - Can have higher risk of overfitting to static data
- **Online RL (eg PPO, GRPO)**
  - Model actively generates new responses in-the-loop and trained using RL with reward model
  - Dynamic feedback loop – if model starts making new kinds of mistakes, training loop can respond immediately (assuming good reward model) unlike static/frozen offline datasets
  - Less Sample efficient
  - Harder to train, more unstable
  - Enables exploration, can enable generalization beyond fixed data
  - Susceptible to reward hacking
- Note in practice, these are often combined as multiple steps in a LLM training recipe
- It is also possible to make offline RL more adaptive, by periodically generating the dataset (called iterative offline learning)

Unused

Question Goes Here

- Answer goes here