Contents

Additional reading on LLMs Fundamentals

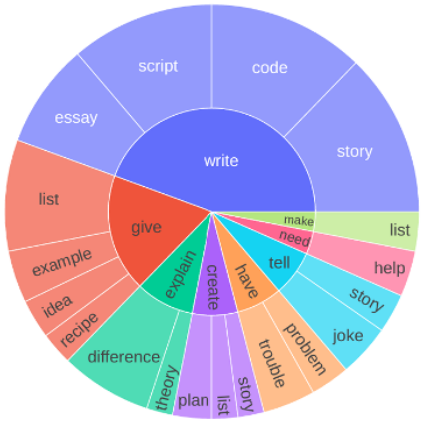
This would be too big to include into this file. This is more for additional reading or as additional information for creating a website.

- LLM alignment – as info from this doc, but in more detail: https://aman.ai/primers/ai/llm-alignment/
- Everything about PEFT in detail: https://aman.ai/primers/ai/parameter-efficient-fine-tuning/
- **Llama 2** from ground up: https://cameronrwolfe.substack.com/p/llama-2-from-the-ground-up
- GREAT recap of past and recent developments in **summarization**: https://cameronrwolfe.substack.com/p/summarization-and-the-evolution-of
- The **transformer architecture** [link]
- **Decoder-only transformer** architecture [link]
- **Language modeling** (next-token prediction) objective [link]
- **Prompting** language models [link]
- **Specialized language models** [link]
- **Decoding** (or inference) with a language model [link]
- **Development with LLMs**: https://www.youtube.com/watch?v=xZDB1naRUlk (LLM, RAG, agents, etc.)

LLM Basics

What types of tasks was it trained on?

Category	Count
Brainstorming	5245
Chat	3911
Classification	1615
Extract	971
Generation	21684
QA, closed	1398
QA, open	6262
Rewrite	3168
Summarization	1962
Other	1767



InstructGPT paper by OpenAI (March 2022)

Humpback paper by Meta (Aug 2023)

Deployment to downstream application

- **Further fine-tuning** on domain-specific data.
- In-context learning - textual prompts incl. **few-shot** exemplars or **RAG**.

What does it take to build these models?

- SoTA General Purpose LLM (i.e. GPT-4) $30 - 100 MM +
- Industry-Specific LLM (i.e. BloombergGPT, Replit) $10 - 50 MM +
- Smaller Single Task LLM $100k - 5 MM +
- Fine-Tune Existing Model $10k - 100k +

Enterprise use cases

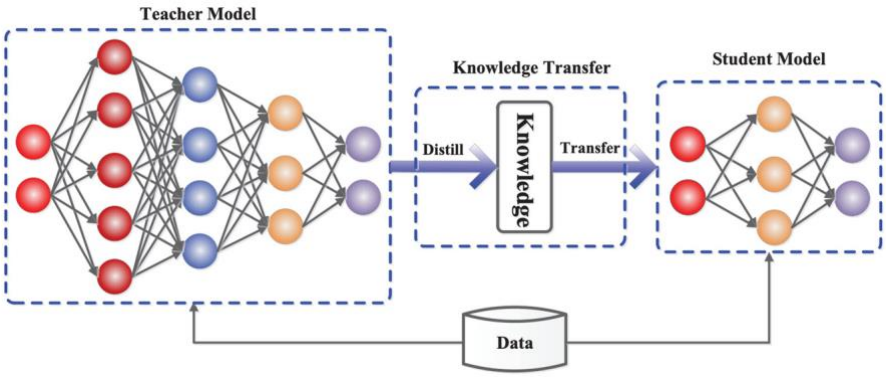
- **Summarization**: product reviews, reports, articles, insights from unstructured data
- **Conversational AI** – customer service bots, contact center solutions, enterprise Q&A,
- **Writing Assistant** – writer's block, writing assistant
- **Knowledge Mining** – domain-specific research, social media trends, cross-functional insights
- **Software Development** – faster coding, debugging, autocompletion, documentation
- **Image Generation** – marketing, logotypes, images / videos for ads (happy cleaner)

What can you do with a trained LLM

- Basic chat or using embeddings in downstream tasks
- Prompt Engineering (few shot, CoT, templates, chains, etc.)
- Retrieval Augmented Generation (RAG)
- Supervised Fine-Tuning (SFT)
- Autonomous Agents

QA with Private Data Protection – example: obfuscate name and credit card number, send to LLM to answer a question, receive an answer and use a map to go to the original name and credit card number.

Imitation Learning



LLaMa catalyzed an explosion of open-source LLM research ==> fine-tunes an LLM over outputs from a more powerful LLM (inspired by knowledge distillation):

- Collect dialogue examples from these models (e.g., using the OpenAI API).
- Perform (supervised) fine-tuning on this data (i.e., using a normal language modeling objective).

Alpaca fine-tunes LLaMA-7B by auto-collecting fine-tuning dataset from GPT-3.5. Cost - $600.
Vicuna fine-tunes LLaMA-13B over 70K dialogue examples from ChatGPT (derived from ShareGPT). Cost - $300.
Koala fine-tunes LLaMA-13B on dialogue examples from Alpaca fine-tuning set + other sources like ShareGPT, HC3, OIG, Anthropic HH, and OpenAI WebGPT/Summarization.
GPT4ALL fine-tunes LLaMA-7B on over 800K chat completions from GPT-3.5-turbo. Authors released training/inference code + quantized model weights for use on CPU.
These models claimed to achieve comparable quality w/ChatGPT and GPT-4. For example, Vicuna is found to maintain 92% of the quality of GPT-4, while Koala is found to match or exceed the quality of ChatGPT in many cases. This impressive performance fostered a promise for open-source LLMs.
However, more targeted evaluations of these models reveal that **they do not perform nearly as well as top proprietary LLMs like ChatGPT and GPT-4** – they just mimic ChatGPT and trick humans into believing they are good. The resulting models i*mprove performance on tasks that are heavily represented in the fine-tuning set* + pronounced tendency for hallucination.
To make imitation models better in general:

- Generating a much bigger and more comprehensive imitation dataset
- Creating a better base model to use for imitation learning

This was found to yield positive results. **Orca** - imitation model based upon LLaMA-13B, fine-tuned over higher-quality massive dataset collected from ChatGPT and GPT-4 (using complex instructions, e.g. CharGPT provides detailed explanations of its response etc., **5M examples from ChatGPT and 1M from GPT-4**) ==> Orca performs incredibly well vs. prior imitation models and narrows the gap between open-source imitation models and proprietary LLMs, but it is outperformed consistently by GPT-4. Nonetheless, Orca's impressive performance reveals that imitation learning is a valuable fine-tuning strategy**.**
On the other hand LIMA (Less is More for Alignment) – fine-tuning on just 1K examples.

Mixture of Experts(MOE)

MoEs replace the feed-forward layers of the transformer model with sparse MoE layers that contain a certain number of experts (e.g. 8), each is a neural net (usually FFN). Router/gate network picks which experts to send tokens to; it is composed of learned params and is pretrained w/the rest of the network.

Each expert in the sparse MoE layer is just a feed-forward neural network with its own independent set of parameters. The architecture of each expert mimics the feed-forward sub-layer used in the standard transformer architecture. The router takes each token as input and produces a probability distribution over experts that determines to which expert each token is sent.

Decoder-only architecture is composed of transformer blocks w/masked self-attention and feed-forward sub-layers.

Sparse activation. The MoE model has multiple independent neural networks (i.e., instead of a single feed-forward neural network) within each feed-forward sub-layer of the transformer. Given a list of tokens as input, we use a routing mechanism to sparsely select a set of experts to which each token will be sent, so we only use a portion of experts for each forward pass.

How does routing work? The routing mechanism used by most MoEs is a simple softmax gating function. We pass each token vector through a linear layer that produces an output of size N (i.e., the number of experts), then apply a softmax transformation to convert this output into a probability distribution over experts. From here, we compute the output of the MoE layer:

1. Selecting the top-K experts (i.e., k=2 for Grok)

2. Scaling the output of each expert by the probability assigned to it by the router

- Pretraining *w/far less compute* => MoE model achieves the same quality as its dense counterpart much faster during pretraining.
- MoEs are *much faster to pre-train* and have *faster inference* due to fewer activated parameters.
- They can outperform dense models with the same number of active parameters.
- MoE experts specialize in shallow concepts or groups of tokens, not specific topics.
- The above are examples of **pre-trained MoEs**. Two other types are described below
- **FrankenMoE** combines fine-tuned models w/same architecture into a MoE, potentially losing advantages of MoEs incl. load-balancing.
- **Upcycled MoE** uses a trained base model and replicates FFN multiple times to create multiple experts.

GPT-4 = 16 expert models, each with around 111B parameters totaling 1.76T parameters.

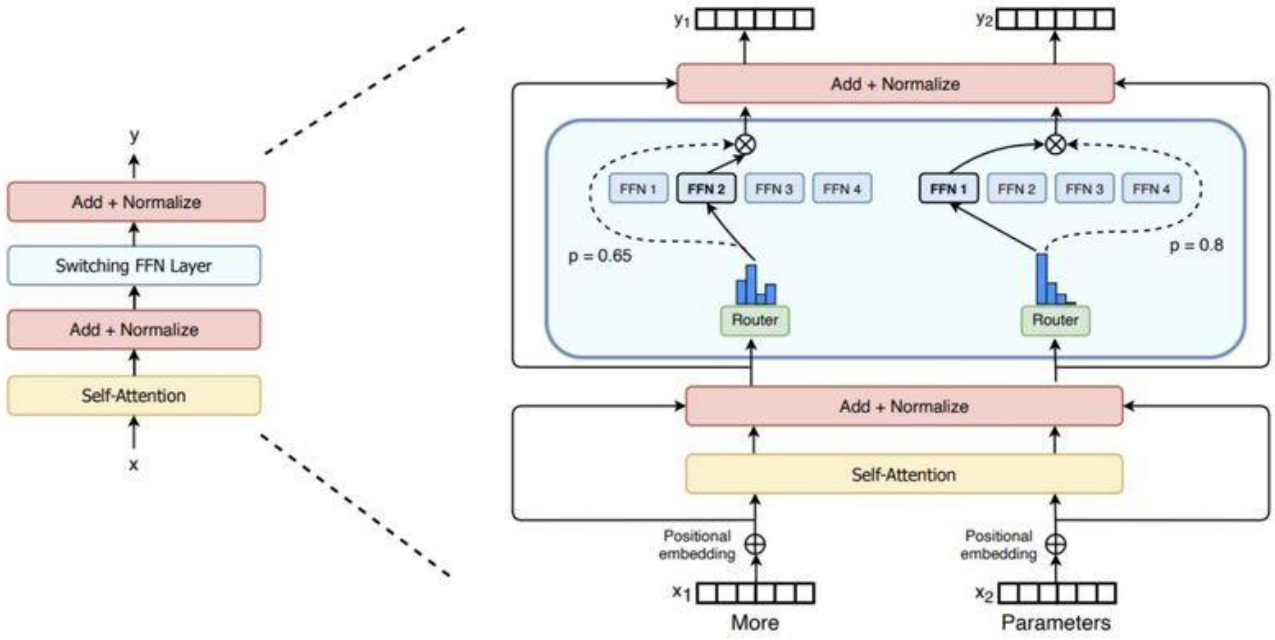
Mixtral 8x7B - 8 models with 7B params; high-quality sparse mixture of experts model (SMoE), decoder-only model. Outperforms Llama 2 70B and GPT-3.5, 6x faster inference, 32K tokens. Supports English, French, Italian, German and Spanish and shows strong performance in code generation. Has 46.7B total params, but uses 12.0B params per token. Processes input and generates output at the same speed and for the same cost as a 12.9B model.

Other MoEs: Google Gemini, Grok-1, Jamba, Databricks DBRX.

MoE Articles:

- https://lnkd.in/dbJuV5sC,
- https://www.linkedin.com/posts/philipp-schmid-a6a2bb196_mixture-of-experts-explained-activity-7179478562398187520-dbzM/,
- https://www.linkedin.com/posts/cameron-r-wolfe-ph-d-04744a238_now-that-grok-1-has-been-released-its-the-activity-7175524199036284928-PalQ/
- Mixtral 8x7B: https://lnkd.in/dKaWZhXY

Switch transformer:



[DBRX: A New SOTA Open LLM]

Latest Advanced Example of MoE (April, 2024) – Learn about these new terms and concepts.

We trained a 132B param MoE model on 12T tokens that beats all other open models and the original version of GPT-3.5.

We evaluated the crap out of this thing and I could regurgitate the claims point by point here, but what I think is most valuable to share is some perspective from building it.
People sometimes have this impression that big LLMs are like this space race where we stand around at whiteboards having breakthroughs and the company with the most talent or insights gets the best model. The reality is much less sexy than that. We're basically all getting the same power law scaling behavior, and it's just a matter of:

- Parameter count
- Training data quantity / epoch count
- Training data quality
- Whether it's an MoE model or not
- Hyperparameter tuning

The real innovation is in dataset construction and the systems you build to scale efficiently.
It also takes immense skill to *debug* the training, with all sorts of subtleties arising from all over the stack. E.g., a lot of weird errors turn out to be symptoms of expert load imbalance and the resulting variations in memory consumption across devices and time steps. Another fun lesson was that, if your job restarts at *exactly* the same rate across too many nodes, you can DDOS your object store—and not only that, but do so in such a way that the vendor's client library silently eats the error. I could list war stories like this for an hour (and I have).

Another non-obvious point about building LLMs that there are a few open design/hparam choices we all pay attention to when another LLM shop releases details about their model. This is because we all assume they ablated the choice, so more labs making a given decision is evidence that it's a good idea. E.g., seeing that MegaScale used parallel attention increased our suspicion that we could get away with it too (although so far we've always found it's too large a quality hit).
In that spirit, here's DBRX (pronounced[2] "D-B-R-X" or "D-B-Rex") in the low-dimensional space of LLM design choices. We ablated almost all of these at smaller scale.

- **LayerNorm**, not RMSNorm
- **QK clipping** to fix loss spikes, not QK normalization (AFAIK no one else does this because Vitaliy invented it)
- **LION optimizer**, not Adam
- A lot of **data curation**. I can't comment on what we trained on (or even what we didn't train on!) because lawsuits have started flying. But suffice it to say, we could have made certain numbers go up a lot more if we were less principled. We also didn't train on any test sets (at least intentionally), although we did do targeted dataset construction with certain benchmarks in mind.
- **RoPE rather than Alibi.** Alibi was never worse quality and we used it for MPT, but RoPE has better software support across hardware.
- **4-of-16 mixture of experts** instead of 2-of-8 using MegaBlocks. This granularity is consistent with the view of MoE as a sparse matrix product. The funny/frustrating story here is that we'd been adding to + using MegaBlocks for months when Mixtral came out and got all the brand recognition from it.
- **Serial Attention + FFNs** (not parallel)
- **No attempt to hide the system prompt**. Here it is, with comments.

Besides raw quality, DBRX is also fast. Since it's an MoE with only 36B active params for a given token, the number of FLOPs you have to do for inference is much lower than for a 70B model. Here's the GitHub repo, model, and interactive demo.

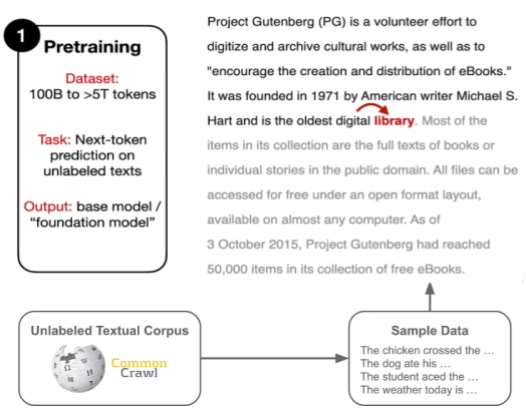
LLM Training and Fine-Tuning

https://cameronrwolfe.substack.com/p/data-is-the-foundation-of-language

One very important recent advancement – **model alignment** or teaching a model to satisfy human's goals, interact with users (steerability). A **small, curated dataset of high quality** can effectively achieve alignment (e.g. 1,000 high-quality response examples, as seen in the LIMA model ("Less Is More for Alignment")) ==> **Superficial Alignment Hypothesis** (SAH) = most knowledge is learned during pre-training, while alignment refines the delivery format. Quality (LIMA) and diversity of data is incredibly important to the alignment process (much more so than the data scale).

Unsupervised phase

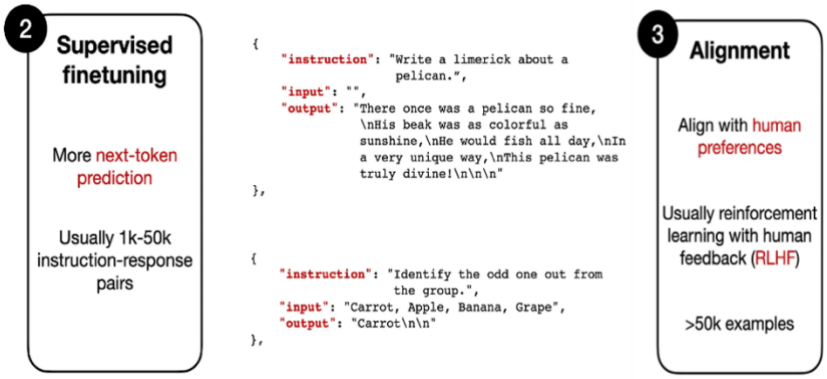
The costliest step, where model is pre-trained on a large corpus of unlabeled text, *learning general-purpose representations via self-supervised learning*. Model is trained to predict the next word.

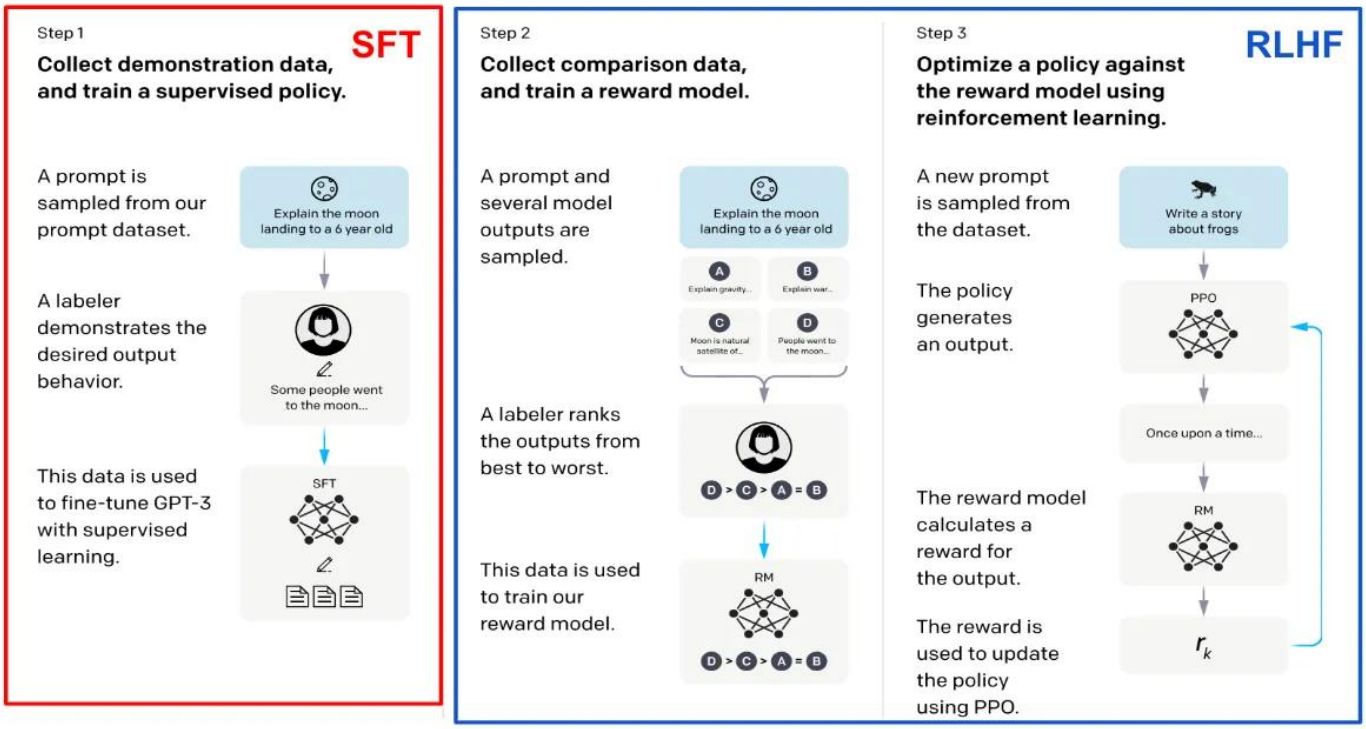


Supervised alignment phase

Using one or the other or both sequentially to better align to end tasks and user preferences:

- **Supervised Fine-Tuning** (SFT) teaches LLM to follow instructions by training it on many examples of accurate responses to instruction-based prompts. Good results require high-quality dataset. SFT datasets are available on HugginFace.
- **Reinforcement Learning from Human Feedback** (RLHF) - optimizes the model using human-provided feedback: multiple responses to the same prompts are generated and ranked by annotators => a reward model is trained with output = scalar reward is maximized (LLM is optimized) via the *Proximal Policy Optimization (PPO)* algo – SOTA for RL, special algos by OpenAI that utilize policy gradient methods - they search the space of policies rather than assigning values to state-action pairs. *Rejection sampling* is also used for RLHF (generates observations from a distribution).
- **Reward model** in RLHF takes a prompt w/full chat history + response as input and **predicts a preference score**, usually it's the **same architecture and weights as the LLM**, but its classification head (next token pred) is replaced with a **regression head** (preference estimation) and model is fine-tuned on preference data





Proximal policy optimization

Proximal Policy Optimization (PPO) used for RL: trains an agent to learn how to act in an environment in order to **maximize reward**. Introduced by OpenAI, became popular for effectiveness and relative simplicity vs other algos Trust Region Policy Optimization (TRPO).

1. **Policy Gradient Method**: directly adjusts policy (agent's strategy) by gradient ascent to maximize the reward.
2. **Objective Function**: clipped version of policy gradient objective - prevents large updates to ensure more stable and reliable improvement.
3. **Actor-Critic Method**: 'actor' updates policy based on feedback from 'critic' which evaluates the policy.
4. **Multiple Epochs per Update**: runs through data multiple times (epochs) - better sample efficiency.

Advantages: stability, reliability, efficiency, simplicity. Drawback: hyperparameter sensitivity, computationally demanding.

Rejection sampling – LLM

- PPO only takes one sample from the model per iteration (iterative updates after each sample). Rejection sampling **samples K responses from the LLM for each prompt**, scores each response using the **reward model**, and **fine-tunes on the best response** (Llama 2 70B was used for rejection sampling to train all other models).
- Rejection sampling fine-tuning uses the same model (i.e., at the beginning of the RLHF round) to **generate an entire dataset of high-reward samples** that are used for fine-tuning in a similar manner to SFT (rejection sampling fine-tuning (**RFT**)).
- For best performance, rejection sampling fine-tuning includes **best samples from all RLHF iterations**, not just the current one.
- **Rejection sampling** approach is adopted for the **first four rounds** of RLHF; *final round = rejection sampling fine-tuning + PPO* sequentially (PPO last)

RLAIF vs. RLHF

Instead of humans, AI ranks the quality of several generated responses for each prompt during the RL phase.

From SFT to OAIF (Online Feedback)

a. Fine-Tuning LLMs as a Dedicated Solution

Instead of using general-purpose LLMs like OpenAI's API, it's possible to leverage open-source models fine-tuned for specific tasks: Anyscale's user-friendly Fine Tuning service accelerates this transition by making it easy to craft accurate, efficient custom models. Despite initial investment to create labeled datasets, the resultant custom models are more efficient and accurate for a specific task = **dedicated solution** that outperforms the broader capabilities of foundational models. This highlights the value: fine-tuning + services like Anyscale = **specialized AI solutions**.

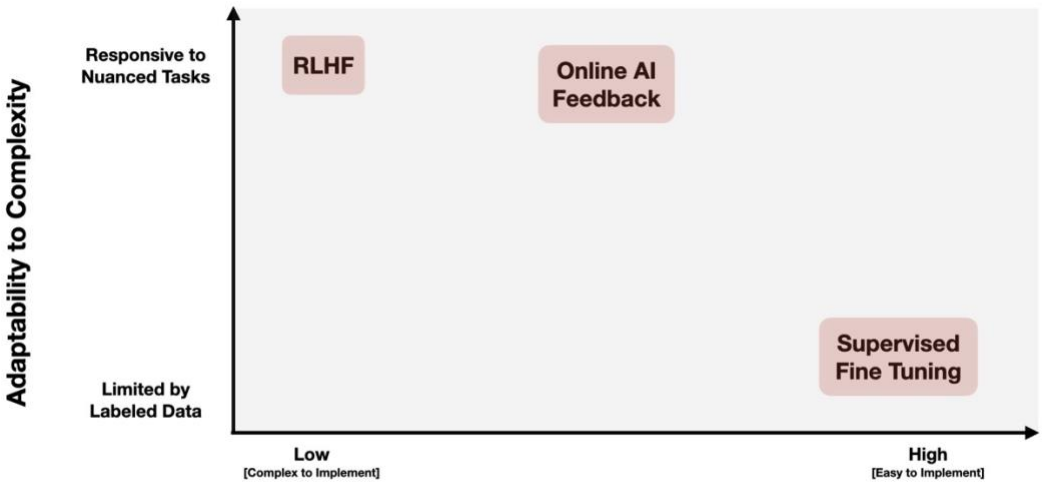
b. Supervised Fine Tuning and RLHF

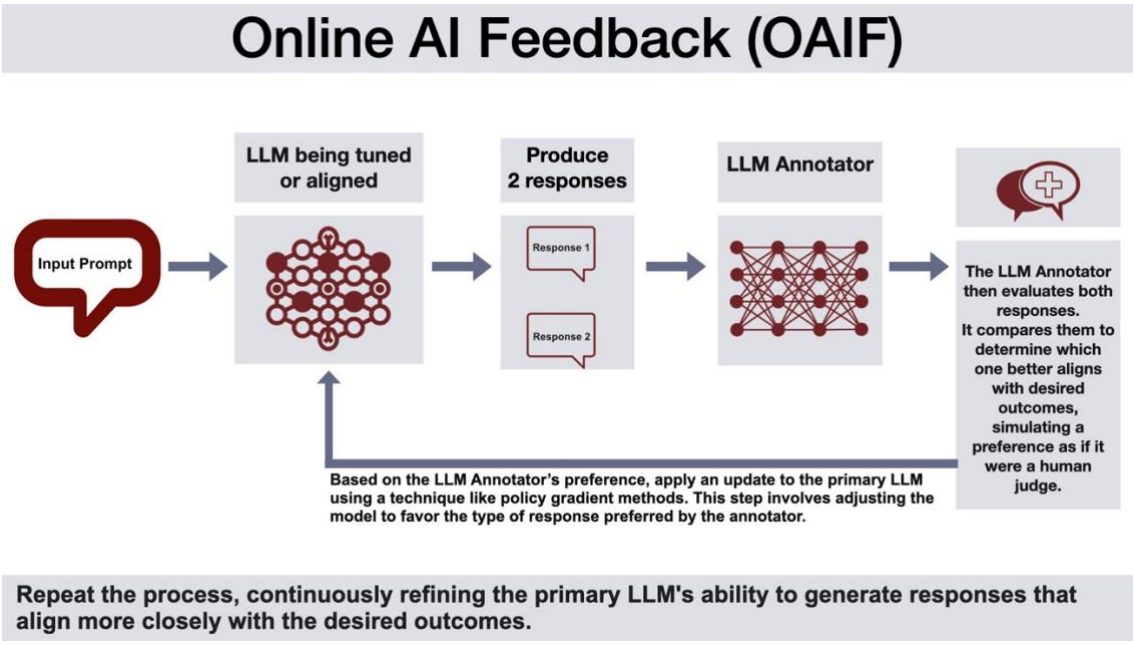
SFT is a *powerful* method for training models *but has limitations*, including the need for *large, high-quality labeled datasets* and potential performance degradation due to *distribution shifts* between training and real-world data. Techniques like adversarial training can enhance SFT's robustness but require more data. An alternative, **RLHF** is an *effective way to teach models new concepts with minimal data*. RLHF is particularly useful for subjective tasks or those requiring *alignment with human values*, where traditional labels may fall short. However, RLHF demands *significant human involvement and introduces training complexity*, necessitating a careful evaluation of trade-offs between SFT and RLHF based on specific use case requirements.

c. Online AI Feedback (OAIF)

Online AI Feedback (OAIF) from DeepMind offers a way to achieve similar benefits with greater efficiency and less human effort – it collects real-time preference judgments from a **separate annotator LM evaluating pairs of responses** from the model being trained. The annotator's feedback about which responses better align with goals is then used to update training in a streamlined manner, without needing a distinct reward model.

Compared to RLHF, OAIF simplifies the process by *extracting preferences directly from the AI annotator eliminating extensive human reward labeling*. This self-supervised approach removes friction in the feedback loop. OAIF also enables *easy customization of desired outcomes* through flexible annotator prompting.



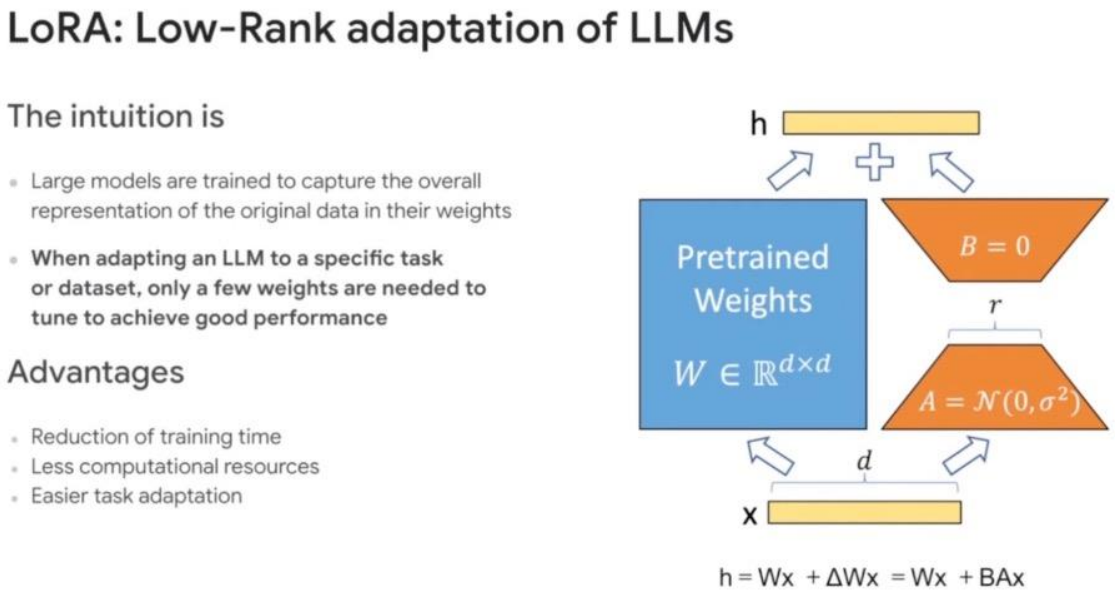


OAIF - more accessible method to *align models to human preferences with less effort*: combines scalability of SFT with responsiveness of RLHF, while automating repetitive judgment tasks. However, reliance on AI evaluation alone risks potential biases or integrity issues => *need oversight and periodic human review* of the annotator's decisions.

OAIF - promising new point between SFT and RLHF if some loss of precision from human-in-the-loop can be tolerated in favor of implementation simplicity.

Parameter Efficient Fine-Tuning (PEFT)

- Full fine-tuning
- Parameter efficient fine-tuning (PEFT): (Q)LORA



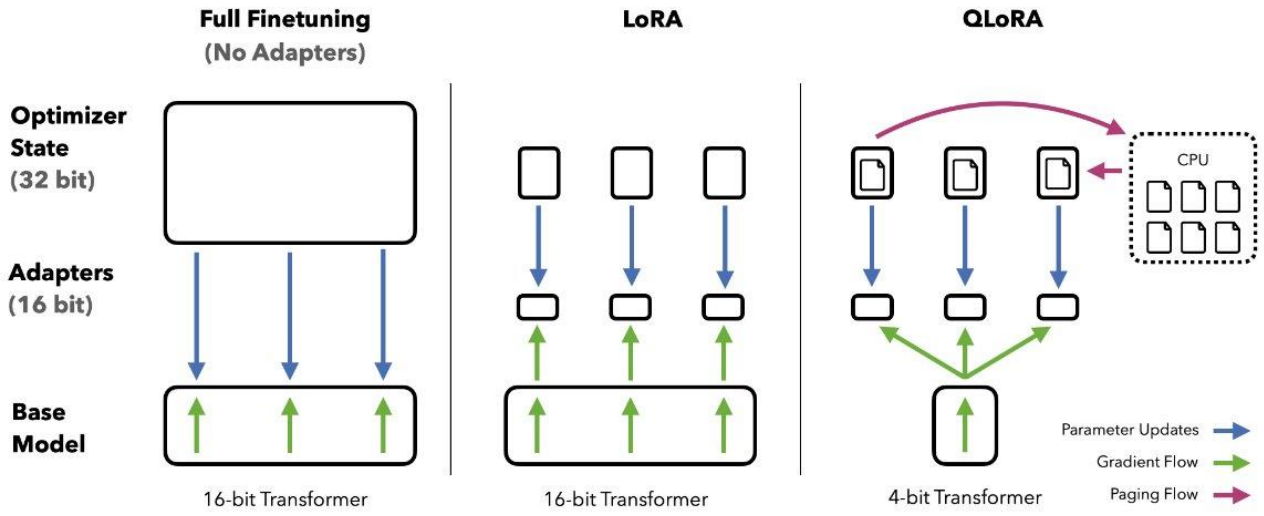
- Significantly **reduces the number of trainable parameters** - inserts a **smaller # new weights** into model and only these are trained => makes training with LoRA much **faster, memory-efficient**, and produces smaller model weights (a few hundred MBs) - easier to store and share.
- How exactly LoRA works: a) **identify layers** to apply LoRA to (see projection layers below), b) instead of directly fine-tuning all weights of identified layers, **LoRA introduces and applies low-rank matrices to these layers** – these m. are much smaller in size vs. original weight matrices, significantly reducing the number of trainable parameters. *The original model weights are kept frozen, and only the low-rank matrices' parameters are updated during fine-tuning*.
- **Matrix rank** – max # linearly independent column vectors or row vectors in the matrix; it determined how much info or complexity the matrix captures about the data or the transformations it is meant to perform. A low-rank approximation of a matrix involves finding a matrix of lower rank that is close to the original matrix, typically in terms of some distance measure like the Frobenius norm. This approximation reduces the computational complexity and memory usage, making it particularly useful for efficiently fine-tuning large pre-trained models by updating only a small subset of their parameters.

Example implementation:

```
import numpy as np
original_matrix = np.random.randn(100, 200)
rank = 5
U, S, VT = np.linalg.svd(original_matrix, full_matrices=False)
# Keep only the top 'rank' singular values/components
U_k = U[:, :rank]
S_k = np.diag(S[:rank])
VT_k = VT[:rank, :]
```

```
# Construct the low-rank approximation of the original matrix
low_rank_matrix = np.dot(np.dot(U_k, S_k), VT_k)  # Shape (100, 200)
```

Google Colab has limited memory, e.g. T4 GPU has only 16 GB of VRAM, which is barely enough to store Llama 2–7B's weights. This makes full fine-tuning not really possible. That is why **we need to use parameter-efficient fine-tuning (PEFT) techniques like low-rank adaptation (LoRA)** to reduce the large weights matrix into smaller low-rank matrixes allowing to freeze the pre-trained model's weights and train only a small number of model's parameters. This means we can have **multiple lightweight portable LoRA models for various downstream tasks** built on top of the pre-trained model. In this notebook we will use the QLoRA technique to fine-tune the model in 4-bit precision.



PEFT works by only updating a small subset of the model's parameters, making it much more efficient. Uses two classes in practice (my Llama 2 and Mistral notebooks):

BitsAndBytesConfig() class sets up *quantization parameters*: load_in_4bit to load the model weights in 4-bit precision, bnb_4bit_compute_dtype - data type for computation after quantization (16-bit floating-point numbers in my notebook - balances between computational efficiency and maintaining sufficient precision), double_quant, bnb_4bit_quant_type = 'nf4' or 4-bit NormalFloat (NF4) quantization - a custom data type exploiting the property of the normal distribution of model weights and distributing an equal number of weights (per block) to each quantization bin - enhancing the information density. Then you pass *BitsAndBytesConfig()* to *AutoModelForCausalLM()* (loads a pre-trained causal language model - predicts proba of next word).

Quantization is a process to *reduces the precision of model's weights*, which makes the model more efficient by significantly *decreasing the model's memory footprint and inference time*, but with a potential trade-off in accuracy - beneficial for deploying models on *resource-constrained devices* such as mobile phones or embedded systems.

These weights are represented as floating-point numbers, which can have varying levels of precision: 32-bit floating-point (float32) or 64-bit floating-point (float64). Quantization reduces the precision of the weights from a higher precision format (like float32) to a lower precision format (like int8 or float16). This involves mapping the continuous range of weight values to a discrete set of values – significantly *reduces the amount of memory* required to store the weights and can also *speed up computations*, as

operations with lower-precision integers are generally faster than with high-precision floating-point numbers.

How the mapping works: you transform the weights from a high-precision format (float32) which represent a vast range of values to a lower-precision format (*8-bit integers*) which can only represent *256 distinct values*. E.g. your float32 weight = 0.34567 in a range of weights from -1 to 1; quantization maps this continuous range to an 8-bit integer scale, e.g. from 0 to 255. You must represent 0.34567 within the new scale via scaling and rounding. For instance, you can map -1 to 0 and 1 to 255, and then scale the values in between, 0.34567 becomes 172. However, the approximation due to *rounding or truncation* can affect the model's performance, which is a trade-off between efficiency and accuracy.

LoraConfig() class defines the configuration for applying LoRA to a model. Params: r OR Lora attention dimension / rank of the adaptation matrices, lower rank - fewer parameters are added, which controls the model's complexity and the amount of computations. I used r=8 - relatively low-rank adaptation; target_modules to apply the adapter to using LoRA - typically *projection layers* within the transformer architecture, such as query (q_proj), key (k_proj), value (v_proj), and output (o_proj) projection layers, as well as other custom layers - adapting these layers allows the model to modify its attention mechanism and output processing in a task-specific manner; bias = "None" => the biases will not be updated as part of the LoRA process; task_type = "CAUSAL_LM".

SFTTrainer() parameter **gradient_checkpointing=True** enables gradient checkpointing to save memory at the expense of slower backward pass. G.ch. selectively stores only a subset of forward pass's intermediate activations for the backward pass (to compute gradients) by discarding most of them. If gradients are needed for an activation that was not stored, the algo recomputes the forward pass for that part of the network - significantly reduces memory usage, allows for the training of large models or the use of larger batch sizes within the same hardware constraints. Why do we need this: despite PEFT, the huge model can still pose challenges due to the high memory requirements for storing activations during training and this method allows to fine-tune models on available hardware by mitigating memory bottlenecks

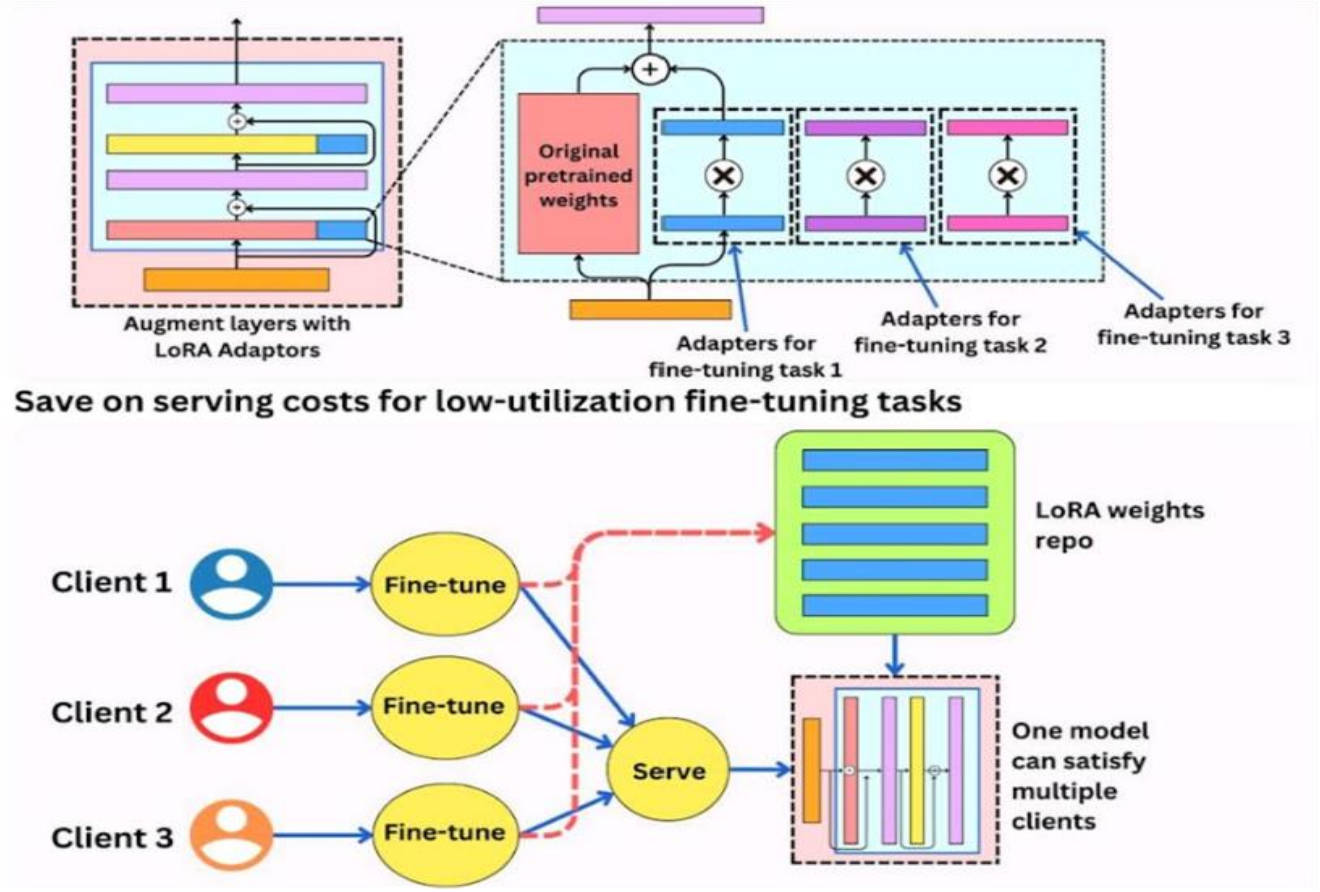
Then we use the **model.add_adapter(LoraConfig())** method to modify the model based on the LoRA configuration – it inserts low-rank matrices into the specified target modules, allowing these parts of the model to adapt to the new task with minimal additional parameters.

Alternative to add_adapter():

```
model = peft.prepare_model_for_kbit_training(model)
model = peft.get_peft_model(model, LoraConfig())
```

LoRA Exchange (LoRAX) by PrediBase

Utilize PEFT and (Q)LORA to build LoRAX to serve the large model just once with many different adapters (1 model for the cost of 100)



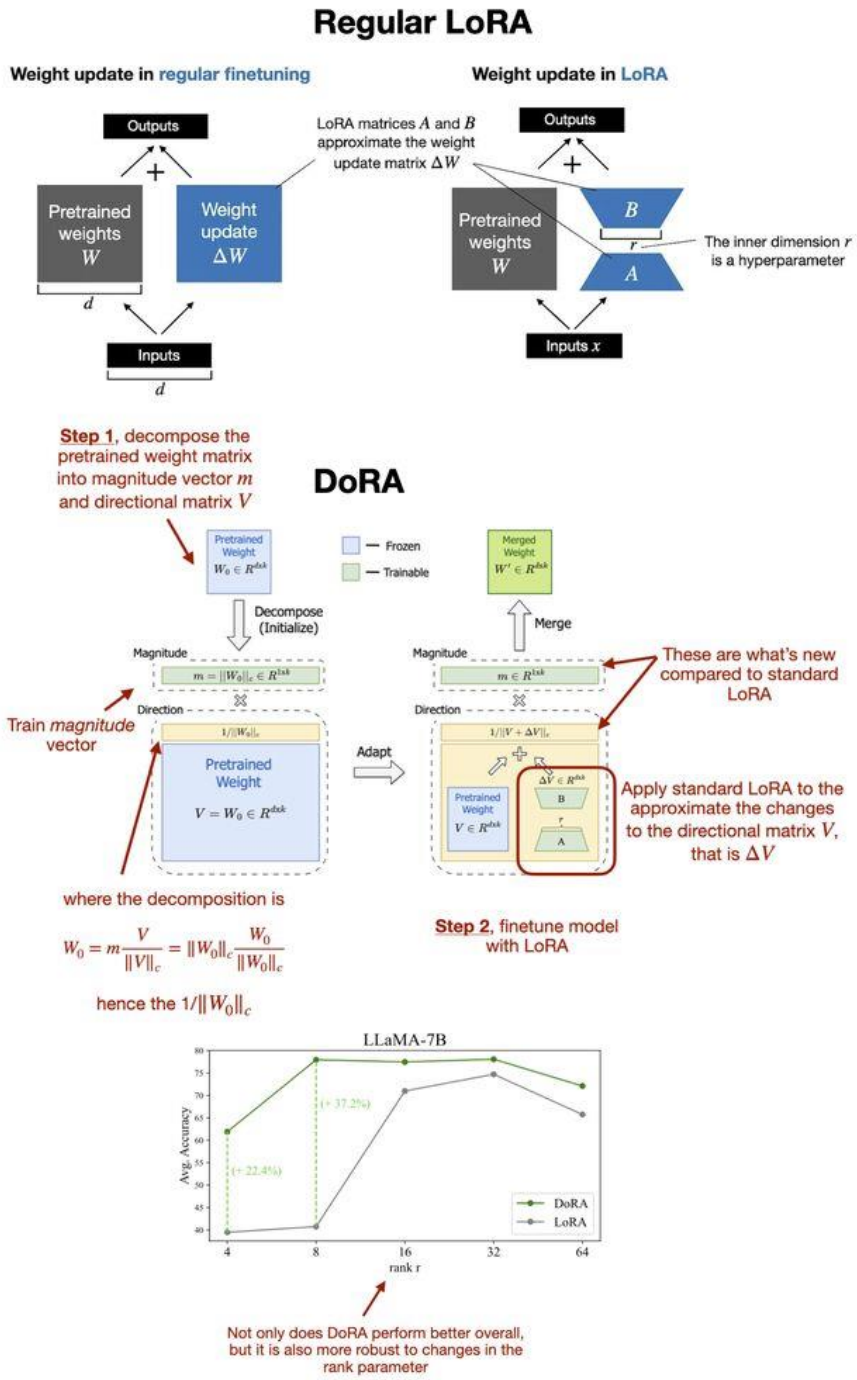
Weight-Decomposed Low-Rank Adaptation (DoRA)

A new PEFT technique that enhances the learning capacity and training stability of LoRA. Decomposes weights updates into a) magnitude, and b) direction. *Direction is handled by normal LoRA, whereas the magnitude is handled by a separate learnable parameter*. This can improve the performance of LoRA, especially at low ranks. For more information on DoRA, see https://arxiv.org/abs/2402.09353.

from peft import LoraConfig
config = LoraConfig(use_dora=True, ...)

DoRA consistently outperforms LoRA. DoRA only supports linear layers at the moment. DoRA should work with weights quantized with bitsandbytes ("QDoRA"). DoRA = bigger overhead than pure LoRA => merge weights for inference, see LoraModel.merge_and_unload().

Regular finetuning => **W' = W + ΔW** (where ΔW is weight change).
LoRA => uses low-rank matrices to approximate ΔW with B∗A, so **W' = W + BA**. (A and B are two low-rank matrices whose product approximates ΔW).
DoRA => decomposes the pretrained weight matrix into a magnitude vector (m) and a directional matrix (V). Then we apply LORA to directional matrix V, i.e. **W' = m (V + ΔV)/norm = m (W + BA)/norm**



GaLore

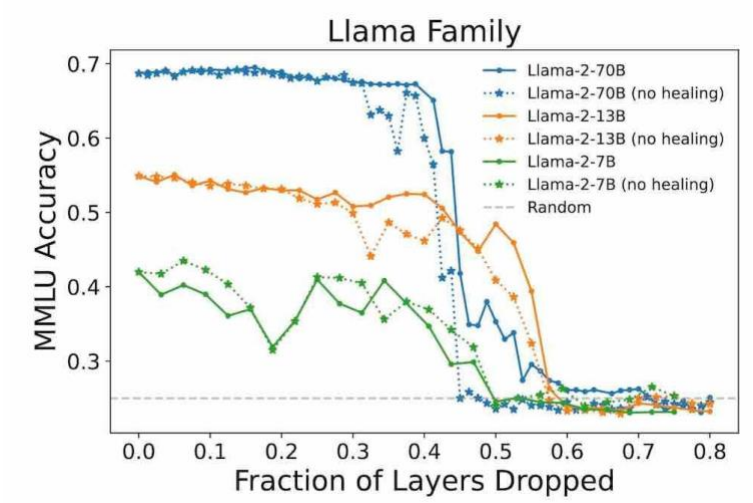
https://huggingface.co/blog/galore

Memory efficient fine-tuning technique for billion-parameter models like Llama 2 7B.
Optimizer state represents a significant portion of memory footprint during training. *GaLore projects gradients into a lower-dimensional subspace* before they are processed by the optimizer - reduces the memory to store these states (by 82.5%) but ensures effective optimization => train larger models or use larger batch sizes. Combined with 8-bit precision optimizers – even more savings. Outperforms LoRA on GLUE and Pretraining of Llama on C4. Integrated into HF Transformers, with **optim="galore_adamw"** or " galore_adamw_8bit" in **TrainingArguments()**, and then SFTTrainer().

Algorithm for 8-bit Optimization with GaLore

- **Gradient Projection**: full-precision *gradients projected into a low-rank subspace* using projection matrices - reduces the dimensionality of gradients.
- **Quantization**: projected gradients, model weights and optimizer states (e.g. moving averages in Adam) are quantized *from 32-bit floating-point to 8-bit int representations* - involves scaling floating-point values to 8-bit range and rounding to nearest int.
- **Optimizer Update**: 8-bit quantized gradients are used to update the model weights: a) *de-quantize gradients back to floating-point format*, b) apply optimizer's update rule (e.g., Adam's moment update and parameter adjustment), c) quantizing updated optimizer states back to 8-bit for storage.
- **De-quantization and Weight Update**: a) 8-bit quantized weights are de-quantized to a floating-point representation for processing (retaining 8-bit precision inherent to their quantized form) because standard operations in PyTorch do not support 8-bit ints, b) de-quantized low-rank updates are projected back to the original space, c) weight update is applied.

Pruning + PEFT QLORA



Meta, Cisco, and MIT researchers demonstrated that large language models (LLMs) could have up to 40%-50% of their layers pruned with minimal impact on accuracy.

Process:

1. Simple layer-pruning of open-weight pretrained LLMs - **minimal degradation of performance** on Q&A benchmarks when **up to 50%** the layers are removed.
2. **Identification of Layers for Pruning**: use *similarity score* to find redundant or less important layers - *layers w/lowest angular distance* between their representations.
3. **Pruning Strategy**: progressively delete layers that showed *minimal change in output* when compared to adjacent layers.
4. **Fine-tuning post-pruning**: to recover any lost performance on benchmarks like MMLU and BoolQ – *small amount of fine-tuning using PEFT, specifically quantization and Low Rank Adaptation (QLoRA)* on the C4 dataset, such that each of our experiments can be performed on a single A100 GPU.

This suggest that layer *pruning can complement other PEFT methods* to further reduce computational resources of finetuning + can improve inference memory and latency. Robustness of LLMs to pruning: shallow layers had a disproportionate importance, while *deep layers could be removed with negligible effects*. This points to potential *inefficiencies in how deep layers are utilized* => current pretraining methods are not properly leveraging the parameters in the deeper layers or that the shallow layers play a critical role in storing knowledge.

Impact:

- direct linear reduction in both memory and compute requirements for inference
- Llama 70B and Llama 13B showed slight accu loss after 40% and 50% layer pruning, respectively.
- Other models (Qwen, Mistral, and Phi families) - minimal accu decline w/20-30% layers removed.

Why This Matters: AI models might be more efficient than previously thought, opening paths to faster, cheaper AI and indicating a need to refine training methods to fully utilize model capacity.

Continual Pre-training of LLMs

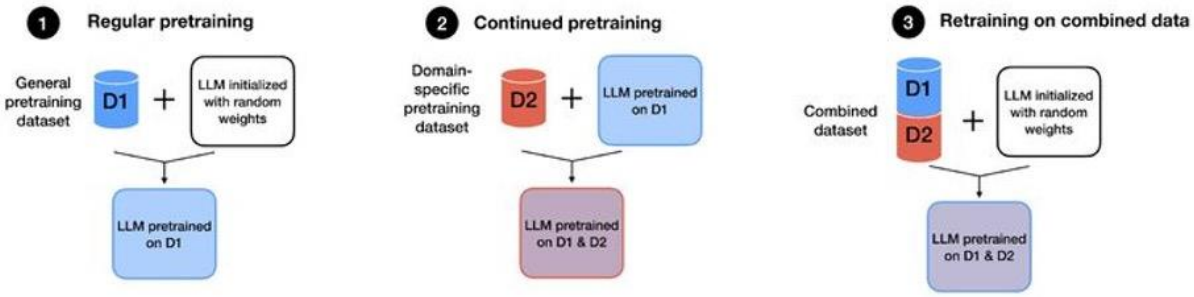
LLMs are pre-trained on billions of tokens, but *new data arrives* - the process is fully repeated (*re-training*). Much more efficient solution - *continually pre-train LLMs*, saving significant compute vs. re-training. But *distribution shift* caused by new data may result in degraded performance on previous data or poor adaptation to the new data.

In general, 3 ways to train LLMs:
1) *Regular pretraining*: initialize model w/random weights and pretrain on dataset D1
2) *Continued pretraining*: adopt pretrained model from 1) and pretrain it on dataset D2
3) *Retrain*: same as 1), but train on datasets D1 + D2 – 2x more expensive than continual pretraining.

Third option is commonly used in practice (e.g., BloombergGPT paper) - *helps find a good LR schedule* (usually a linear warmup with a half-cycle cosine decay) and *w/catastrophic forgetting*.

But **continual pretraining is much more efficient** and *matches the performance of full re-training from scratch*, as measured by final validation loss, downstream task performance and average score on several LLM eval benchmarks:

1) **Re-warm and re-decay LR** (re-applying the typical LR schedule)
2) **Add a small portion (e.g., 5%) of original pretraining data (D1)** to the new dataset (D2) to prevent catastrophic forgetting.



Direct Preference Optimization (DPO)

Video: https://www.youtube.com/watch?v=QXVCqtAZAn4&t=808s&ab_channel=DeepLearningAI

Direct Preference Optimization (DPO) - simpler, more robust, higher performing successor of RLHF used in HF's Zephyr, Intel's NeuralChat, etc. *RLHF - complex and unstable w/many hparams to tune* + *need 3 LLMs* (a lot of compute): first fitting a reward model for human preferences (1st LLM), and then fine-tuning the LLM using RL to maximize this estimated reward (2nd LLM), but making sure it is not too far from the original reference model (RM = 3rd LLM); the latter is ensured by *KL divergence penalty to prevent reward hacking* controlled by beta (r.h.=LLM learns to output some garbage that is rewarded highly like emojis or backslashes). Below is the equation for RLHF:

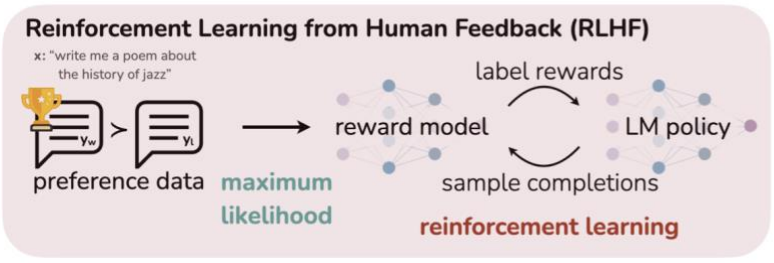
$$\max_{\pi_\theta} \mathbb{E}_{x\sim\mathcal{D}, y\sim\pi_\theta(y|x)} \left[r_\phi(x, y) \right] - \beta \mathbb{D}_{\mathrm{KL}} \left[\pi_\theta(y \mid x) \,||\, \pi_{\mathrm{ref}}(y \mid x) \right]$$

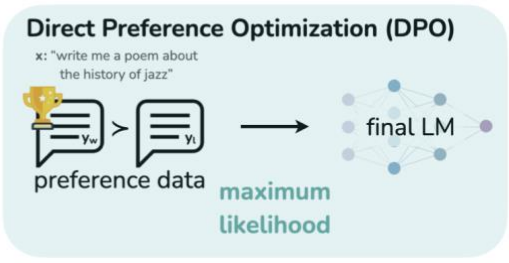
maximise rewards

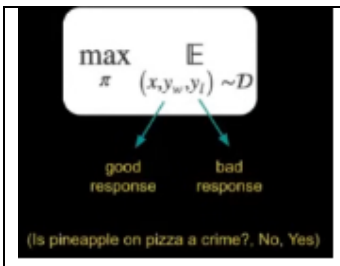
use KL-divergence penalty to prevent **reward hacking** (controlled by β)

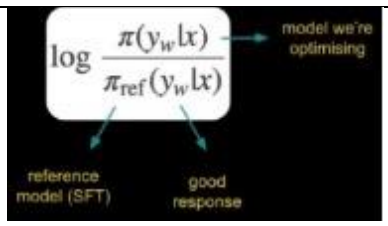
This constrained reward maximization problem can be optimized with DPO using a single stage of policy training by *learning a policy directly from collected data without a reward model* – applying a *simple loss function optimized directly on a dataset of preferences* {(*x*, *yw*, *yl*)} = {(prompt, preferred, dispreferred responses)} = *classification problem* on the human preference data. Result - stable, performant, computationally lightweight algo - promising alternative for aligning LLMs to human/AI preferences; no need for to fit a reward model / sample from LLM during fine-tuning, or performing significant hyperparameter tuning.

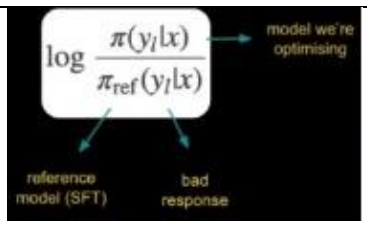
DPO aligns LLMs as well as or better than existing methods. It exceeds RLHF's in controlling sentiment of generations and improves response quality in summarization and single-turn dialogue while being substantially simpler to implement and train. DPO success prompted researchers to develop two new loss f(x).

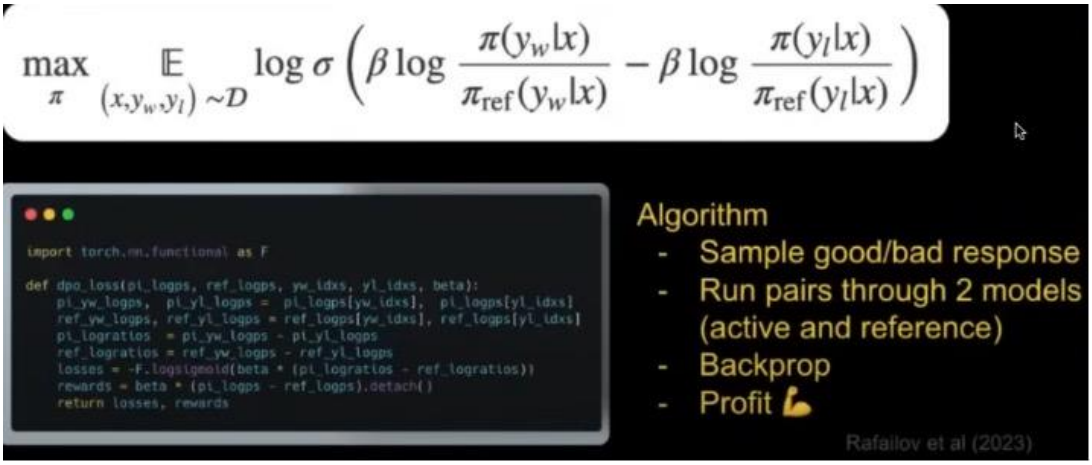












In the above DPO equation, we are trying to maximize the difference between the chosen response and the rejected one to make the model gets much better at predicting the chosen response. This algo is differentiable – we can just use backpropagation to optimize a model without RL (the equation for RL above isn't differentiable).

Fine-tune Llama 2 https://huggingface.co/blog/dpo-trl with:

- SFT: BitsAndBytesConfig() + AutoModelForCausalLM() + LoraConfig() + SFTTrainer
- DPO: model=AutoPeftModelForCausalLM() + model_ref=AutoPeftModelForCausalLM() + DPOTrainer(model, model_ref, …)

HF recommends the Chat Template for SFT before DPO (available in transformers lib). There are ~50 SFT datasets + 15 DPO datasets on HF. DPO dataset has additional fields: chosen and rejected

Iterative DPO or online DPO – take a reward model, use prompts to generate responses from chat model, use the reward model to rank the prompts and retrain another DPO model; then generate again on new prompts and rank and then do this a set of times and the model gets progressively better.

Stepwise DPO (sDPO) improves performance by doing *DPO in multiple steps* on a subset of the alignment data at each stage. Using an already aligned reference model in the 2nd DPO stage results in a stricter alignment (e.g. Step 1: DPO align using OpenOrca preference data, Step 2: DPO align using UltraFeedback preference data. Result: sDPO can improve LLM by 1.6% on H4; sDPO+Solar 10B outperforms Mixtral (5x more parameters).

Because DPO tends to quickly overfit on the preference dataset, **Identity Preference Optimization (IPO)**, *adds a regularization term to the DPO loss* and enables one to train models to convergence without early stopping (proposed by Google DeepMind).

Vs. RLFH and DPO, IPO *simplifies* the learning process and directly *learns pairwise preferences without requiring point-wise estimates*, such as ELO scores, which are used in RLHF and DPO (pointwise as in reward, ELO score – worse / better scores for different responses to the same prompt). This direct method of learning preferences is intended to *better exploit of KL-regularization* which prevents overfitting by ensuring the LLM remains close to the original non-aligned model while improving.

While RLHF and DPO can overfit the training data, their more general learning objective, Ψ preference optimization (ΨPO), encompasses both (RLHF, DPO, and IP are special cases of PsiPO). *DPO is prone to overfitting*, as it achieves perfect accuracy too quickly on training data, without generalizing well. IPO aims to address this by *focusing on pairwise preferences rather than the more complex ELO score-based logit-*

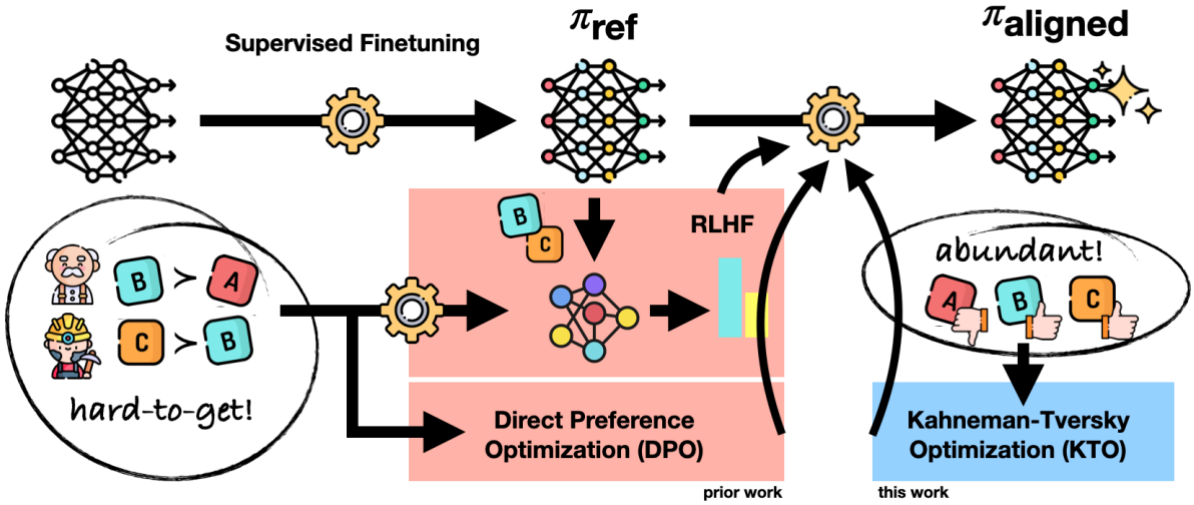
preferences used by RLHF and DPO. The authors performed an in-depth analysis of RLHF and DPO to identify their potential pitfalls, and considered another special case of PsiPO by setting Psi to Identity.

Ψ = [(p)sai]

Human-aware loss functions (HALOs) are used to align ML models with human preferences, grounded in the principles of prospect theory from behavioral economics. Dispensing with paired preferences, **Kahneman-Tversky Optimization (KTO)** dispenses w/binary preferences and *defines a new HALO loss function entirely in terms of individual examples that have been labelled as "good" or "bad"* (thumbs up or down). KTO uses an *adapted version of Proximal Policy Optimization (PPO) for simplified offline, one-step alignment using existing preference datasets*, bypassing the update of the reference model to enhance stability. RLHF and DPO – more time consuming and costly because they requires a dataset of paired / pairwise preferences {(*x*,*yw*,*yl*)}, where annotators label which response is better according to a set of criteria like helpfulness or harmfulness.

The technique is inspired by the work of economists Daniel Kahneman and Amos Tversky on *human decision-making*. KTO utilizes a *singleton feedback to determine if an output is desirable* based on criteria such as successful customer interactions. Such thumbs up or down *labels are much easier to acquire* in practice, and KTO is a promising way to continually update chat models running in prod env. The method is especially advantageous for organizations because it can *utilize readily available customer interaction data* to align LLMs to desirable outcomes (e.g., sales made) as opposed to undesirable ones (e.g., no sale made). By focusing on whether a given output is desirable in itself, KTO *simplifies the alignment process*, making it more accessible and feasible for a wider range of users and applications. This approach also *alleviates the subjective nature of human-rated preferences*, which can often be a source of conflicting data, thus streamlining the alignment process and potentially providing a big boost in LLM performance over both unaligned models and those fine-tuned with standard methods or DPO

KTO matches or surpasses the performance of DPO without relying on the comparative preferences.



Most important hyperparameter in these methods is β (weighting the preference of the reference model). Libraries like 🤗 **TRL** already provide these alternative method. HF implemented DPO, IPO, and KTO in **DPOTrainer()**.

More from the "Aligning LLMs with DPO" webinar by Lewis Tungsten:

- How do you **evaluate the quality of an open-source alignment dataset**? – Vibes testing by looking at the labels to find errors + using another LLM like GPT-4
- **Evaluating fine-tuned LLMs** – best eval: human. If not, HF uses *MT-Bench* (multi-turn dialogue data that evaluates reasoning, math, summarization, creativity, etc.) and *AlpacaEval* (rates your LLM against GPT-4). Other famous benchmarks: HumanEval (programming benchmark), MMLU (language understanding)
- **How much data is enough for alignment**? – Usually, 10-100K examples for SFT and 50K example for DPO. Recent Data algorithm can reduce that to approx. 10K for SFT and 50K for DPO.
- **Create your own dataset vs. use an open-source one** – it really depends on your use case; **if it's too specific**, e.g. a dataset for call center, then create your own dataset.
- **Does DPO work at scale because the largest model tried was 7B**? – There is already a released 70B model trained using DPO. DPO has more potential to work at scale vs. RL.

Odds Ratio Preference Optimization (ORPO)

Typically, LLM alignment = SFT + RLHF / DPO. ORPO *collapses these two steps into one*. Working with *preference data*, based on log odds ratio this method **introduces a penalty to the NLL loss f(x)** (negative log likelihood), **to favor generations in the chosen response sets**. ORPO shows faster training, lower memory usage and good results.
Note: Log odds ratio = natural log of odds ratio. An odds ratio - relationship between the exposure of one variable and the occurrence of the other. What are the odds of Y occurring given the exposure to X as opposed to no exposure.

Is DPO Superior to PPO for LLM Alignment?

https://www.linkedin.com/posts/sebastianraschka_while-i-was-eagerly-awaitingthe-technical-activity-7187072869334413312-aDHS/?utm_source=share&utm_medium=member_android

To summarize:

RLHF is one of the main pillars of the LLM training lifecycle: pretraining -> supervised instruction-finetuning -> RLHF. In recent months, the reward-free DPO method has become one of the most widely used alternatives to the reward-based RLHF with PPO because it doesn't require training a separate reward model and is thus easier to use and implement.

Consequently, most of the LLMs on top of public leaderboards have been trained with DPO rather than PPO. Unfortunately, there have not been any direct head-to-head comparisons where the same model was trained with either PPO or DPO using the same dataset until this paper came along.

It's a well-written paper with lots of experiments and results, but the main takeaways are that PPO is generally better than DPO, and DPO suffers more heavily from out-of-distribution data.

If you use DPO, make sure to perform supervised finetuning on the preference data first. Also, iterative DPO, which involves labeling additional data with an existing reward model, is better than DPO on the existing preference data.

For PPO, the key success factors are large batch sizes, advantage normalization, and parameter updates via an exponential moving average.

Fun fact: Based on what we know from the LLama 3 blog post, it doesn't have to be RLHF with either PPO or DPO. Llama 3 was trained with both! Pretraining -> supervised finetuning -> rejection sampling -> PPO -> DPO.

SimPO

Simple Preference Optimization (by Princeton) - new RLHF method **improves simplicity and training stability** for offline preference tuning while outperforming DPO or ORPO.

Very similar to DPO, but different:

1. Uses the **average log probability** of a sequence as the reward.
2. Does **not require a reference model**, reducing computational and memory requirements.
3. Employs a **length-normalized reward** formulation to penalize too longer responses.
4. Introduces *a target reward margin*, encouraging **a larger gap between chosen and rejected responses**.
5. Objective is to **maximize the cumulative reward**, which is the *difference between the average log probabilities of the chosen and rejected responses, minus the target reward margin*.

Insights

- Reduces time by ~20% and GPU memory by ~10% compared to DPO
- Outperforms DPO on key benchmarks (AlpacaEval 2, Arena-Hard)
- Up to 6.4 points improvement on AlpacaEval 2 and up to 7.5 points on Arena-Hard.
- SimPO does not significantly increase response length compared to DPO models
- Enforcing a larger target reward margin effectively improves reward accuracy
- Maintains performance on academic benchmarks like MMLU, GSM8k
- Built using Hugging Face TRL

Paper: https://lnkd.in/epEmda9d

Github: https://lnkd.in/eSBJAH6e

Models: https://lnkd.in/eSy5G9Ak

LangSmith

Unified framework for LLM EVALUATION and DevOps platform for developing, collaborating, testing, deploying, and monitoring LLM applications. Everything is described on the LangSmith website: you get API keys, spin off LangSmith instances (VMs) for your tasks, etc.

Unsloth

(promising technology)
Helps finetune LLMs like Mistral, Gemma, and Llama 2-5x faster using up to 70% less memory on NVIDIA GPUs. Employs LoRA, bitsandbytes, custom Triton kernels for efficient training acceleration w/out losing accuracy. Supports conversational models, raw text, and direct preference optimization (DPO) for reward modeling.

Deploy Llama 2 70B on AWS Inferentia2

Deploy Llama-2-70b on **AWS Inferentia2 (SageMaker)** using HF LLM Inf2 Container - a **special HF's Inference container** powered by **Text Generation Inference** (HF's toolkit for deploying and serving

LLMs) and **Optimum Neuron** (interface between Transformers lib and AWS Accelerators like Inferentia).

AWS inferentia (Inf2) - **EC2 for DL inference**. Llama 2 is deployed as an **end point** on instance type e.g., inf2.48xlarge where you define batch size, sequence length, and auto cast type. Expose it as a Gradio app. Then you can conduct throughput and latency tests.

RAG

In-Context Learning (Few-Shot or RAG)

GPT-3 and subsequent LLMs have been able to do something more powerful: **learn new tasks and skills** *simply from new examples* in the input, without any gradient updates or changes to the pre-trained model. This ability is known as **in-context learning (ICL)**. Exemplar ordering, distribution, and format may impact performance. It unlocks more flexible, general, and human-like intelligence in AI systems providing for:

- **Versatility** - single model learns a variety of skills (no re-training).
- **Generalization** - model learns underlying patterns from a few examples and generalizes to unseen data.
- **Efficiency** – no re-training.
- **Accessibility** - AI systems learn from users through simple demonstrations of task.

The **ability to "learn to learn"** is enabled by:

a) advanced **model architecture** (RNN not capable of this): Transformers can perform *optimization algorithms* (regression and gradient descent) => transformers can be trained to learn how to solve new tasks, instead of just specific task BECAUSE they have a *dual-form of gradient descent* - under right conditions the math of the Transformer's *attention mechanism* = the math of gradient descent - a general optimization algorithm. Calculating attention is approx. similar to gradient-based learning!

b) large **model scale** - bigger models (by neural network parameters) learn better and handle more complex tasks when learning in context because they ensure *more rule-based than exemplar-based generalization*. Also, bigger models can override prior semantic knowledge if the new task requires *learning new semantics*! Also, bigger models - *more complex in-context learning* (transformers w/2 layers perform 1 step of gradient descent, 8 layers - Ridge Regression, 12+ layers - Ordinary Least Squares). Also, bigger models are *more robust to noise*.

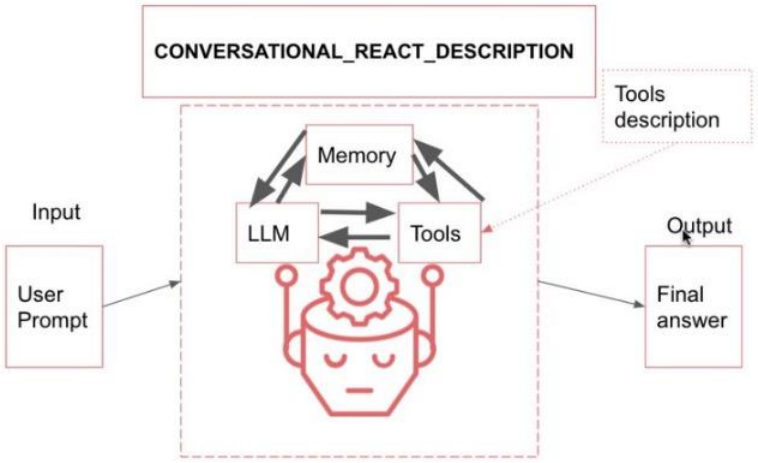
c) need certain **data distribution** to enable ICL – *long-tailed* (Zipfian) data distrib. (a lot of rare tokens), *rare tokens must appear in clusters* (topics?), tokens need to be dynamic (tokens need to have different meanings given different contexts) - dynamic tokens => model is forced to learn to infer the meaning from context - hence in-context learning. All three are available in natural language data. If the data distrib. is not complex enough, Transformers don't learn ICL.

ICL was never the goal in training transformer, but we got lucky with natural language data. ICL => profound implications for AI - models adapting to new tasks with minimal retraining. We can leverage this knowledge to build more capable ICL systems.

RAG w/LangChain

- **Loaders**: from websites, DBs, YouTube, arXiv, etc. for dtypes: PDFs, HTML, JSON, Word, ppt (return a list of `Document` obj).
- **Splitters**: CharacterTextSplitter(),TokenTextSplitter(), MarkdownHeaderTextSplitter(),SentenceTransformersTokenTextSplitter(), RecursiveCharacterTextSplitter() (recursively tries to split by different chars to find one that works), Language() (CPP, Python, Ruby, Markdown, NLTKTextSplitter(), SpacyTextSplitter() (sents).

- **Retrieval**: basic *semantic similarity*, maximum marginal relevance (get `fetch_k` most similar responses within which choose *k most diverse ones*), LLM Aided Retrieval (SelfQuery – *use LLM to convert user question into query*), using metadata
- **Compress** – use LLM to shrink retrieved chunks to only relevant info to pack as much info into context as possible (map reduce approach).
- **Rerank** retrieved chunks and keep only top n
- **Refine** retrieved chunks iteratively to have fewer of them sent to LLM as context.
- **Agent**: use an LLM to *determine which actions to take and in what order*. Action can be using a tool, observing its output, and deciding what to return to the user. Need a prompt template, LLM, and parser - *parses LLM output into AgentAction* or AgentFinish object.



RAG at AWS

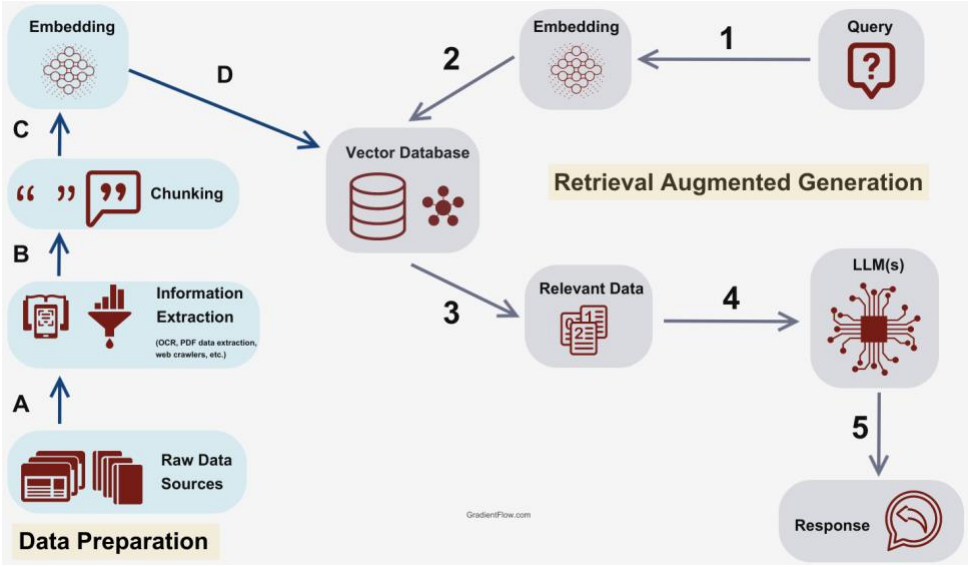
AWS - Unlock the potential of generative AI in industrial operations: https://aws.amazon.com/blogs/machine-learning/unlock-the-potential-of-generative-ai-in-industrial-operations/

Retrieval Augmented Generation Best Practices

RAG refers to the process of *providing an LLM with additional information retrieved from elsewhere* to improve the model's responses.

RAG produces more accurate and factual outputs by *grounding LLM in external knowledge sources* – it can pull relevant information from large DBs to include in generated text - *prevents hallucinated or incorrect information* in model outputs. Associated tools - vector databases, LLM endpoints, LLM orchestration tools, etc.

Best practices are still developing. Deploying an effective RAG system (even a simple one) requires *extensive experimentation to tune / optimize many different parameters*, components, and models, including data collection, model embeddings, chunking strategies, and more.



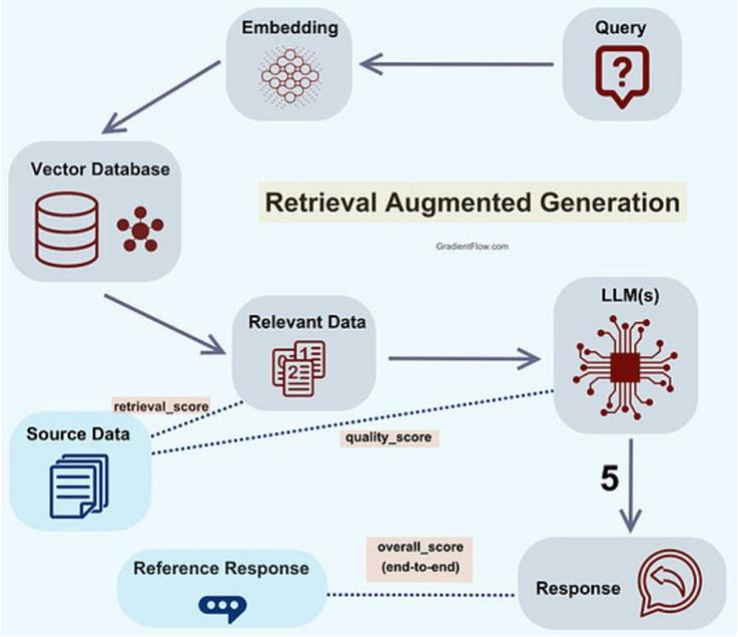
RAG Evaluation

Anyscale proposes a detailed two-step evaluation of RAG systems:

- **Component-wise evaluation** assesses individual RAG elements like the efficiency of the retrieval system in sourcing relevant content and the LLM's ability to generate quality responses from optimal sources.
- **End-to-end evaluation** that examines the overall system quality, incorporating the effectiveness of data sources used.

Key metrics: a) **Retrieval_Score** measures the relevance of retrieved content by comparing query embeddings with those of KB chunks, and b) **Quality_Score** evaluates the overall quality of responses based on relevance, coherence, and accuracy.

Implementing this evaluation framework requires procuring a *reference dataset* w/real-world representative questions, *relevant documents* or passages as ground truth sources, *responses through the RAG system* using these sources, and *human evaluators* scoring the responses based on relevance, accuracy, and factual consistency, following specific criteria.



Component-wise and end-to-end evaluation metrics introduced by Anyscale for RAG.

Data Quality in Knowledge Corpus

RAG retrieves information from this corpus to generate responses - poor-quality data can lead to inaccurate or uninformative outputs. An ideal setup involves automated tools for identifying data quality issues, akin to **Visual Layer** for computer vision and **CleanLab** for text data. Key factors affecting RAG system data impact include:

- **Data Quality**: Inaccuracies, incompleteness, or biases in data can cause misleading RAG responses.
- **Data Preparation**: Essential steps include cleaning data, removing duplicates, and ensuring data compatibility with the RAG system.
- **Data Sources**: A limited range of sources may restrict the comprehensiveness and informativeness of responses. Diverse sources enrich RAG outputs.
- **Metadata**: Providing metadata improves the Large Language Model's (LLM) context understanding, enhancing response quality.
- **Additional Context and Knowledge**: Incorporating knowledge graphs at query time offers extra context, improving the accuracy and informativeness of responses.

Chunking

= dividing input text into smaller, meaningful units to aid the retrieval system in finding contextually relevant passages for generating responses. *Effectiveness of RAG heavily depends on quality and structure of text chunks*, ensuring the precision of retrieved information for user queries.

Anyscale's research highlights the importance of chunking strategies over embedding optimization – do experiment with different chunking approaches to improve RAG performance:

- Tests show that **chunk sizes** *of 100, 300, 500, and 700 tokens* are beneficial while larger chunks' benefits taper off after a point when *additional context becomes noise*.

- Also, many open-source *embedding models have a limitation* - 512 sub-word tokens.
- The **number of chunks** affects retrieval and quality scores, but only up to a certain limit, with experiments showing diminishing returns beyond seven chunks due to the LLM's context length constraints.
- Effective strategies include **using smaller chunks and retrieving adjacent chunk content** or maintaining multiple embeddings for a document.
- Theoretically, **chunk overlapping** ensures that chunks passed to the LLM include context from neighboring chunks, potentially enhancing output quality by preserving surrounding context.

Embedding Model

Anyscale's experiments revealed the choice of *embedding model significantly impacts retrieval and quality scores* - **smaller models may outperform top-performers** from a leaderboard; depends on the case! The chunking strategy appears to have a slightly greater impact on the quality than the embedding model.

Off-the-shelf embedding models work well in most cases, but if you have a specific domain, you may want to **fine-tune an embedding model on the domain data** to improve the retrieval quality.

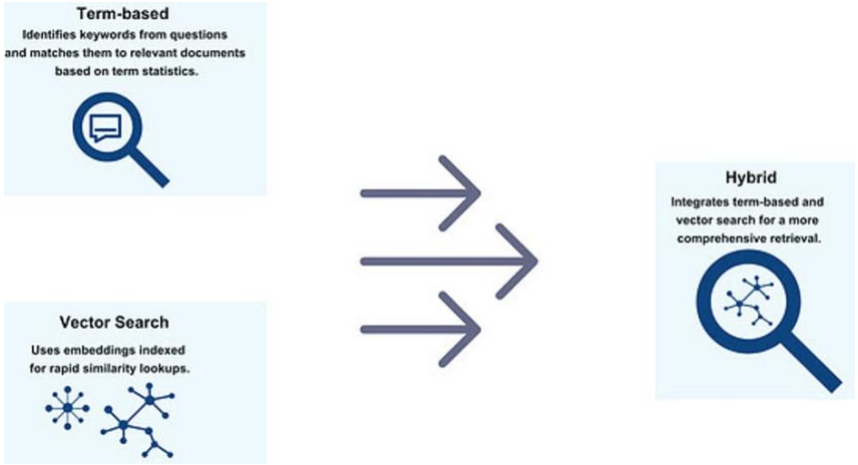
Information Retrieval and Data Management

Vector search is not limited to vector databases. It is supported by many data management systems including Postgres, near real-time or streaming systems (Rockset's), knowledge graphs (Neo4j's support for vector search means your RAG results will be easier to explain and visualize).

Efficient retrieval is key in RAG. There are several mechanisms:

- **term-based matching**, like BM25, which identifies keywords from questions and matches them to relevant documents based on term statistics; it is simpler and needs less preprocessing;
- **vector similarity search** that relies on embeddings that are indexed, allowing for rapid similarity lookups; vector similarity provides a deeper contextual relevance;
- **hybrid retrieval methods**.

For optimal outcomes with RAG, you will have to run experiments with your data and determine the most suited retrieval strategy.



LLM

It is crucial to choose the right LLM depending on the app's objectives (succinct summaries, advanced reasoning, etc.); larger models more coveted for reasoning tasks. Experiment w/various LLMs based on your app's unique demands - accuracy, latency, and cost, etc. Early design patterns include:

- **Mixture of Experts**: *supervisor model* amalgamates outputs from *several LLMs* or guides the input query to the most apt LLM.
- **Intent Classifiers**: combined with a naive RAG, they discern when a user query matches predefined canonical forms, enabling faster and precise responses - a conversational agent.

Tune Away

RAG is here to stay, and systematic exploration will be the key to its success. Hyperparameter optimization tools like **Ray Tune** could enable the systematic refinement of RAG systems. **Ray's** distributed computing framework is essential for parallelizing computationally intensive, combinatorial experiments (grid search), enabling exploration of many more configurations.

Benefits

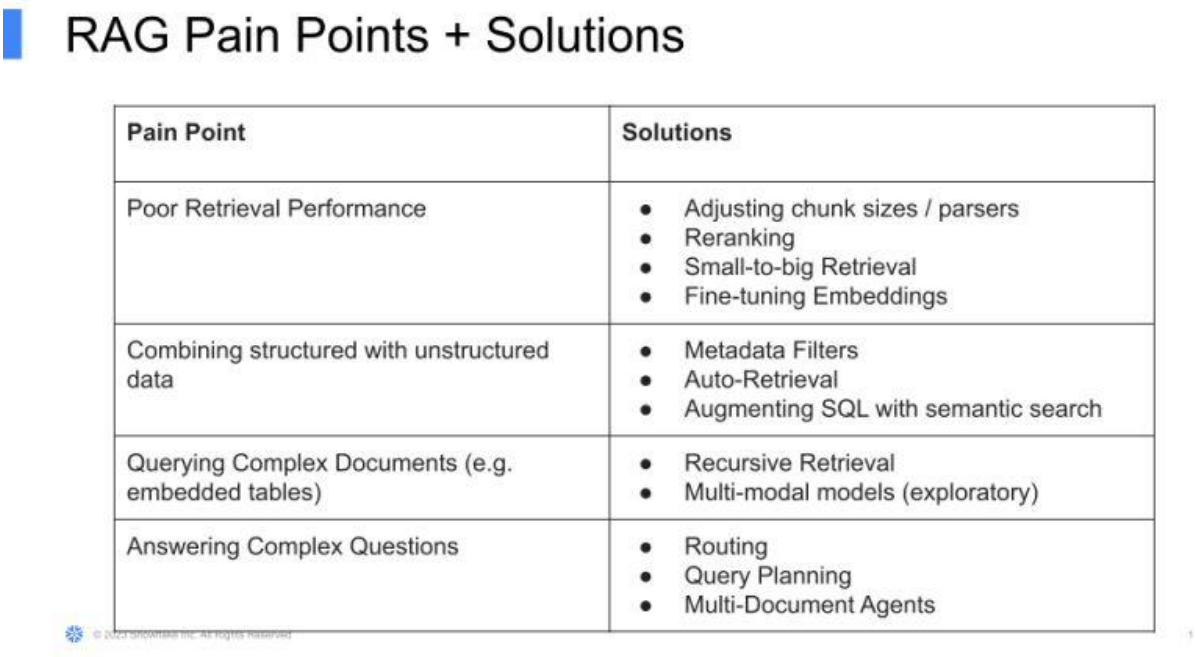
- **Reducing Hallucinations**: RAG significantly lowers the risk of LLMs generating incorrect information (even fine-tuned LLMs!) by providing direct access to reference data, thereby improving trust and the quality of LLM applications.
- **Explainability (Verification by User)**: it's hard to explain the outputs of an LLM; RAG provides references/citations from the retrieved doc chunks that explain the RAG output - highly explainable. RAG *enables users to verify the output of LLM*
- **Access to Up-to-date Information**: Unlike LLMs with fixed knowledge cutoff dates, RAG allows for the easy *incorporation of current information* into the LLM's output, bypassing the limitations of finetuning in updating LLM knowledge.
- **Data Privacy and Security**: RAG offers a safer *alternative to including confidential info or IP / proprietary data in LLM training sets*, which are susceptible to data extraction attacks or which is not acceptable to clients. With RAG, there is no need to separate confidential info out which is sometimes not an easy process.
- **Ease of Implementation**: Compared to finetuning, RAG is *simpler and more cost-effective* to implement, with the possibility of enhancing retrieval model quality without the need to train the LLM itself.
- **Lower cost, no special MLE expertise:** RAG does not require training or fine-tuning – usually no high cost associated with it (data collection / annotation, GPUs are expensive for fine-tuning). Fine-tuning requires significant expertise in DL and transformers (hparams like # epochs, how to avoid overfitting, etc.).
- **No overfitting and catastrophic forgetting**: the fine-tuned model "memorizes" the smaller fine-tuning dataset rather than "understands" it OR it forgets tasks it previously knew how to solve in favor of new ones.
- **Access control** – if employee doesn't have permission to access, don't show then this doc chunk during RAG.

RAG References

More about RAG in Cameron Wolfe's blog here.: **A Practitioners Guide to Retrieval Augmented Generation (RAG).**

Add schematics for 2 FourthBrain RAG notebooks?

Skim through 2 ppt RAG presentations by LlamaIndex in this directory. Table below is from them:



RAG Pain Points + Solutions

Pain Point	Solutions
Poor Retrieval Performance	• Adjusting chunk sizes / parsers • Reranking • Small-to-big Retrieval • Fine-tuning Embeddings
Combining structured with unstructured data	• Metadata Filters • Auto-Retrieval • Augmenting SQL with semantic search
Querying Complex Documents (e.g. embedded tables)	• Recursive Retrieval • Multi-modal models (exploratory)
Answering Complex Questions	• Routing • Query Planning • Multi-Document Agents

RAG vs. Fine-Tuning

Per paper RAG vs Fine-tuning: Pipelines, Tradeoffs, and a Case Study on Agriculture, there is an "accuracy increase of over 6 percentage points when fine-tuning the model. RAG additionally increases accuracy by 5 percentage points more." Also, " the fine-tuned model leverages information from across geographies to answer specific questions, increasing answer similarity from 47% to 72%."

RAG combines a retriever with LLM to enhance responses using real-time data or external databases, ensuring responses are evidence-based and more accurate. *It is effective when data is contextually relevant* (e.g. farm data), but it can *significantly increase the prompt size* and become harder to steer.

Fine-tuning adjusts model's weights to improve its contextual, stylistic, or domain response, essential for cases needing *specific behavioral or language adaptation* (Google's example – need a real estate model speaking colloquial English). *Fine-tuning could be tuned for brevity* and can incur *less inference cost* when dealing with large datasets. The *challenge is the initial cost / effort* to build a new dataset and fine-tune models on it.

Overall, the suitability of each approach depends on the *specific application, the nature and size of the data, and available resources for model development*. Many other reports suggest *the combination of two approaches*. It would be also interesting to combine structured information from PDFs with images and captions to enable *multi-modal fine-tuning opportunities*. Before choosing an approach - assess the use-case:

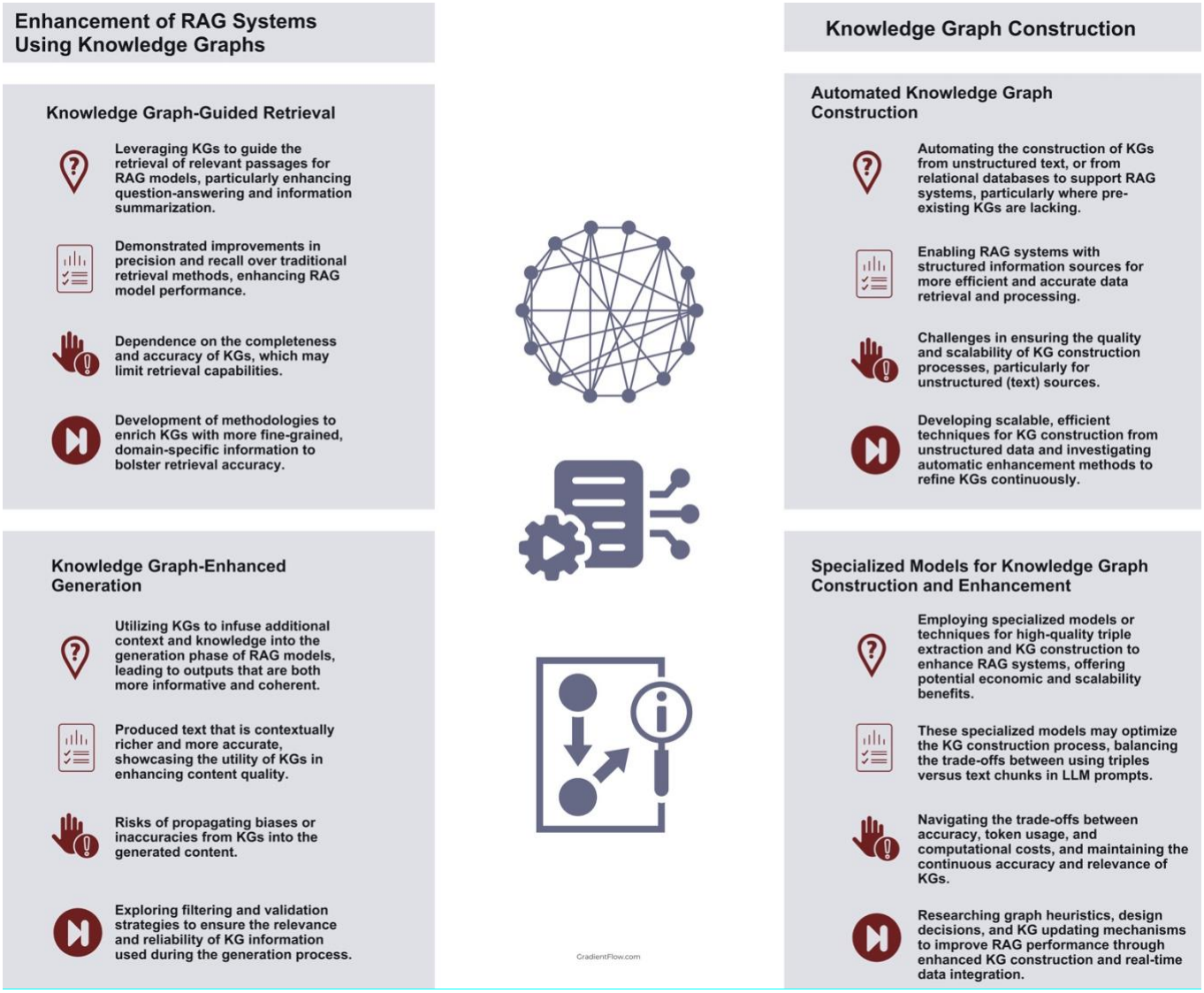
a. **finetuning is model adaption** - changing the *LLM's behavior (structure=weights), vocabulary, writing style, customizing model's tone or jargon* for a niche application, but not changing its knowledge, for example a pretrained LLM might find it challenging to summarize the transcripts of company meetings, because they might be using some internal vocabulary, OR LLM fine-tuned on medical texts to become more adept at answering healthcare-related questions – something that RAG can do too, but fine-tuning => shorter prompts => better steerability and savings in the long

run. It's like giving additional training to an already skilled worker to make them an expert in a particular area,

b. RAG is the other way around – **RAG relies on updated external data** to generate outputs **grounded to custom knowledge base** while the LLM's vocabulary and writing style are unchanged,
c. *if your app needs both* custom knowledge & LLM adaptation - use a **hybrid approach** (RAG + fine-tuning),
d. *if you don't need either*, **prompt engineering** is the way to go.

Knowledge Graphs + RAG

Recent work on the enhancement of RAG w/knowledge graphs (KGs) has shown promising results. Studies have demonstrated improvements in precision and recall over traditional retrieval methods, enhancing RAG model performance. However, the effectiveness of this approach depends on the completeness and accuracy of the KGs, potentially affecting retrieval efficacy.



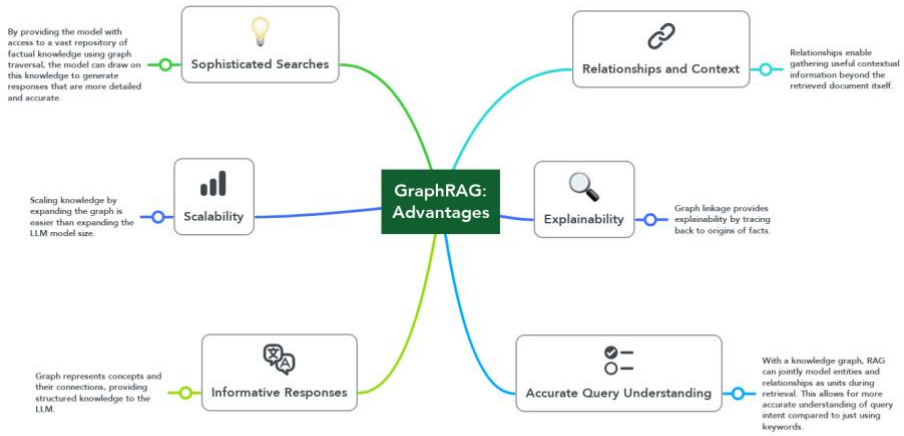
KGs are used mostly for augmenting context during the RAG's generation phase. Beware of transferring biases or inaccuracies from the KGs.

Automated KG construction from unstructured data is being explored to support RAG systems, particularly where pre-existing KGs are lacking. Specialized models to optimize KG construction are being employed for high-quality **triple extraction**, offering potential cost and scalability benefits. They balance **trade-offs between using triples versus text chunks** in LLM prompts.

GraphRAG: Design Patterns, Challenges, Recommendations

RAG is enhanced by integrating **structured, domain-specific knowledge graphs** (KGs) or graph DBs with LLMs to **provide better context to LLMs** for generating responses. Conventional vector search focuses on unstructured data using high-dimensional vectors while GraphRAG leverages KGs for **more nuanced understanding and retrieval** of interconnected, heterogeneous information.

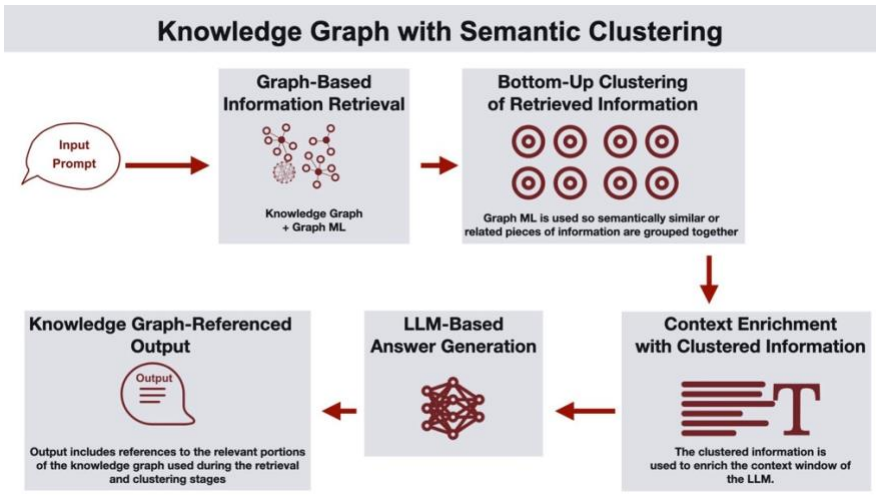
These are early days for GraphRAG - we barely know of any examples of production deployments and we need more benchmark datasets and evaluation methodologies. GraphRAG is here to stay - **using knowledge graphs provides structured information to enhance the LLM prompts at the generation stage**. Getting started: become familiar with graph databases, how to ingest data into them, and how to query them (**Kuzu** is an easy-to-setup option).



Common GraphRAG architectures:

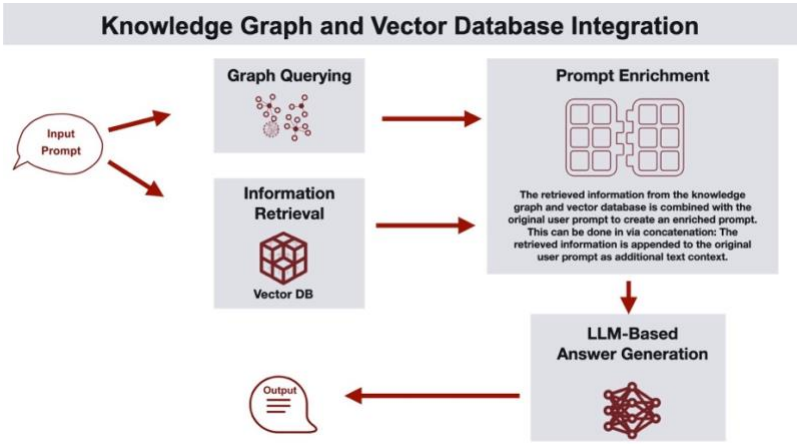
- **KG + Semantic Clustering**

User submitting query => system uses KG and graph ML to retrieve relevant info which is then organized into semantic clusters through graph-based clustering. Clustered info enriches LLM's context. Final answer includes references to KG for better data analysis.



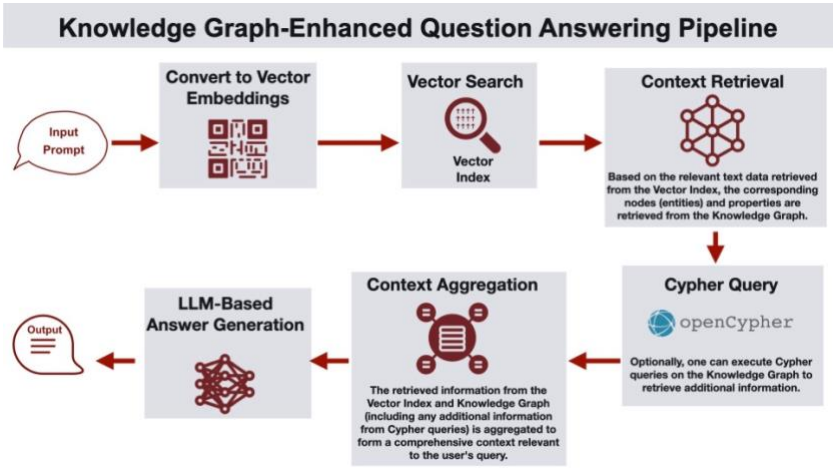
- **KG + Vector DB**

Leverages both KG and vector DB. KG is constructed to capture relationships between chunks of vectors, including document hierarchies, structured entity information in the neighborhood of the retrieved chunks. Use cases - **customer support, semantic search, personalized recommendations**.



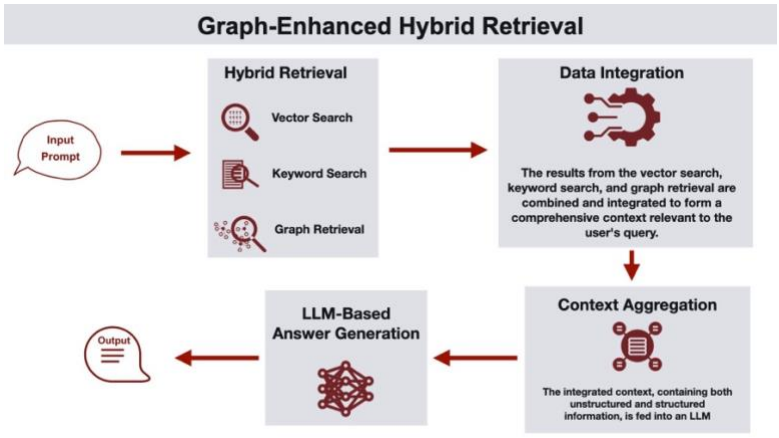
- **KG-Enhanced Q&A Pipeline**

KG is used downstream of vector DB to add additional facts. Use cases: **healthcare or legal**, where a standard response is always included along with the answer, based on the entities in the response.



- **Graph-Enhanced Hybrid Retrieval**

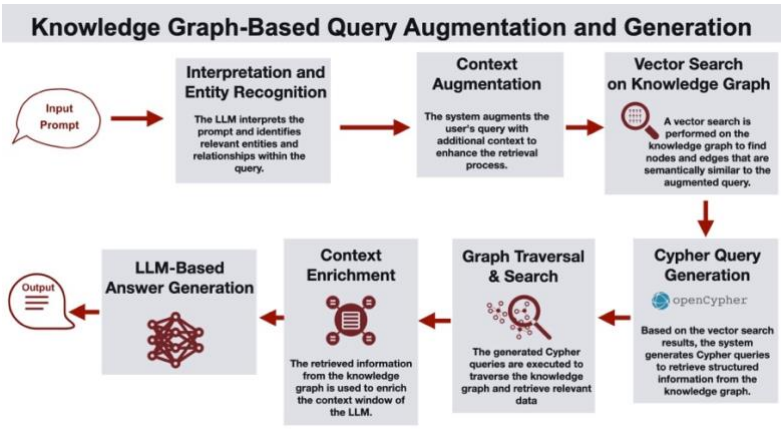
Vector search + keyword search + KG queries to retrieve relevant info efficiently. Retrieval from vector and keyword index can be enhanced w/reranking or rank fusion. All three search results are combined to augment the LLM's context. Use cases - **enterprise search, document retrieval, knowledge discovery**.



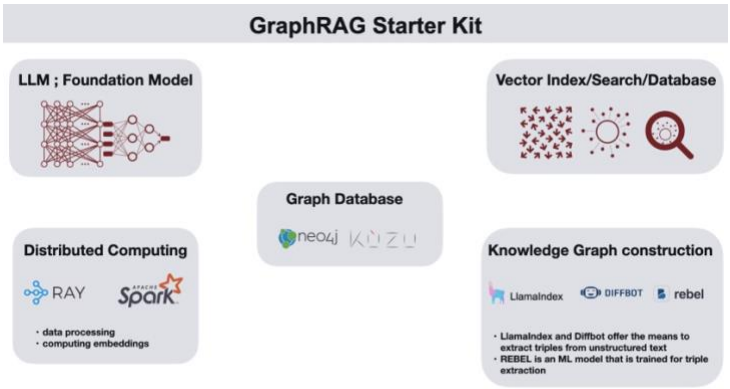
- **Knowledge Graph-Based Query Augmentation and Generation**

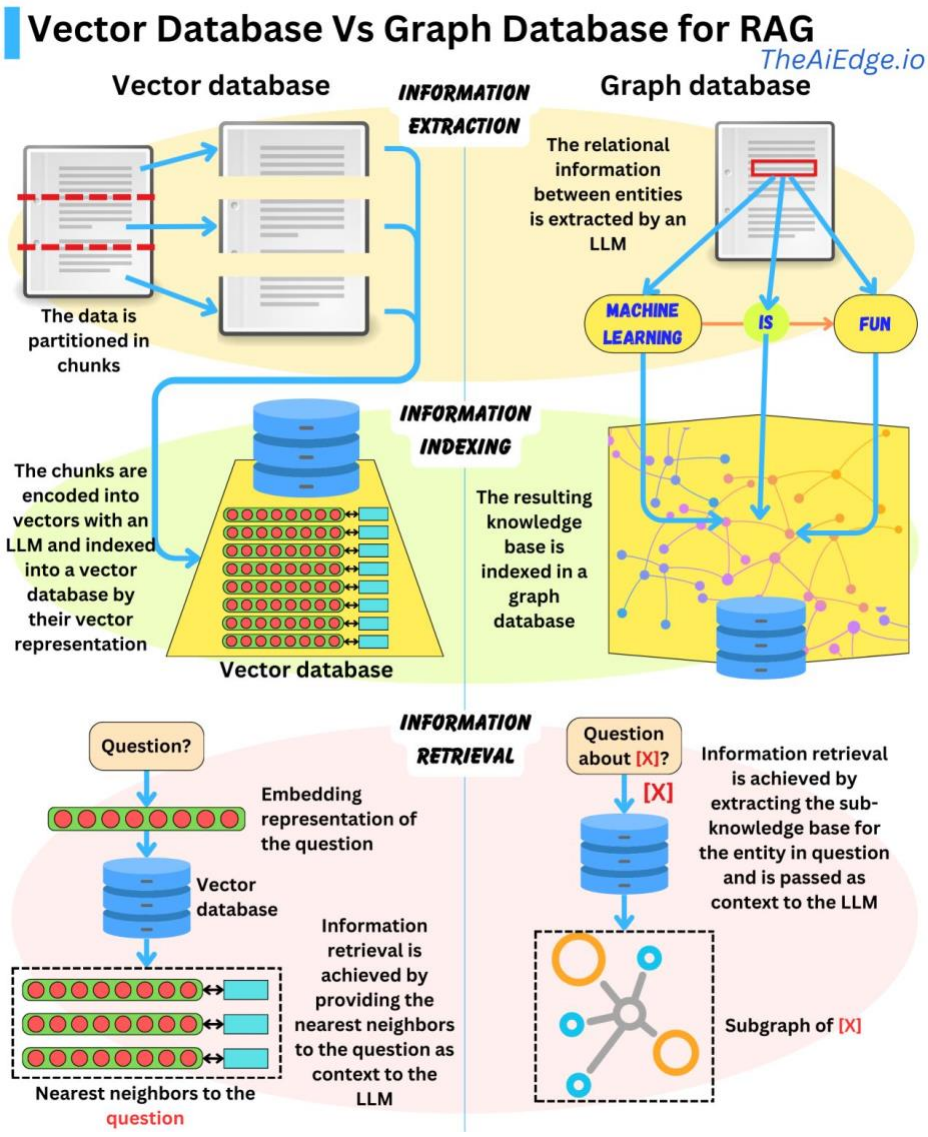
KG is used **prior to performing vector search.** Query augmentation (LLM extracts entities and relationships) => Vector search on KG node properties to narrow down nodes => Query rewriting - Cypher queries run on retrieved subgraph to further narrow down on relevant structured info => Retrieved data from KG traversal is used to enrich the LLM's context.

Suitable for **product lookups or financial report** generation where the relationships between entities are important.



Challenges - building an efficient knowledge graph requires **deep domain understanding** and expertise in graph modeling. LLM-based automation of this process s still in its early stages. RAG itself requires **computationally demanding experiments** to identify optimal information extraction, chunking, embedding, retrieval and ranking strategies. So does GraphRAG.





Practical link - [Using LlamaIndex to build KGs from LLMs]

Retrieval Augmented Fine-Tuning (RAFT)

https://arxiv.org/abs/2403.10131

RAFT trains the model to **answer the question (Q) from retrieved Documents (D)** to generate an answer (A), where A includes chain-of-thought (CoT). Documents (D) include **k distractors** to teach the model to ignore unhelpful (negative) information.

Post-training recipe to fine-tune LLMs for RAG in specialized domains by achieving domain adaptation and robustness against irrelevant text. It combines SFT w/RAG. *Training data contains questions, oracle documents w/answers, and distractor documents w/out answers*. The model is trained to generate answers from the provided documents by *ignoring irrelevant documents (distractors)* and getting the right sequences from relevant document that would help answer the question.

By incorporating a mix of relevant and irrelevant docs during training, RAFT improves the model's *robustness against inaccurate retrievals and fluctuations in # docs encountered during testing*. During testing, the model is provided with questions and top-k documents retrieved by the RAG pipeline and must discern the pertinent content.

RAFT's *CoT-style response* helps improve the model's ability to reason. By incorporating a reasoning process and citing sources in the answers, RAFT enhances the model's accuracy and prepares it for real-world applications in specialized domains.

Insights:

- Adding chain-of-thought (CoT) answers boosts performance, up to ~15%
- Adding distractor documents makes the model more robust
- RAFT Llama 2 7B outperforms GPT-3.5 on PubMed (Medical)

JavaScript RAG Web Apps with LlamaIndex

A course on DeepLearningAI - building full-stack RAG app in JavaScript, from the *API server* to a *React component* that queries your data. You assemble an *intelligent agent* running its own queries and learn to develop a *persistent chat that streams your answers* to an interactive front-end in real time.

Embedding quantization

Substantial cost and latency reductions in *retrieval* and *similarity search*: CPU-only retrieval demo with under 0.1 s latency across all of Wikipedia! *Embeddings are stored in fewer bits* instead of the standard 'float32' format.

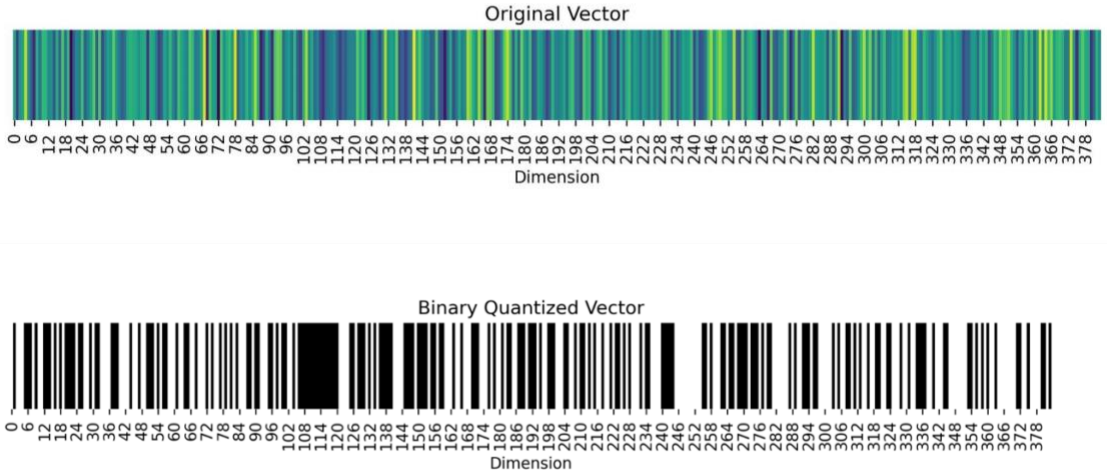
- **Binary Quantization**: 45x lower latency, 32x less memory, 96% of retrieval performance.
- **Scalar (int8) Quantization**: 4x lower latency, 4x less memory, 99.6% of retrieval performance.
- **Combining both techniques**: binary search for minimal latency and memory footprint + scalar rescoring for high performance = notable cost savings.

Binary Embedding / Vector Quantization

Video by Mehdi: https://www.youtube.com/watch?v=aqGVF2YFDkc&t=63s

The concept of binary quantization simplifies the encoding of vectors by retaining only their directionality. Each vector dimension is encoded with a single bit, indicating whether it's positive or negative. For instance, a vector like [12, 1, -100, 0.003, -0.001, 128, -1000, 0.0001] would be condensed into a single byte, resulting in the binary sequence [1,1,0,1,0,1,0,1]. This massively reduces the amount of space every vector takes by a factor of 32x by converting the number stored at every dimension from

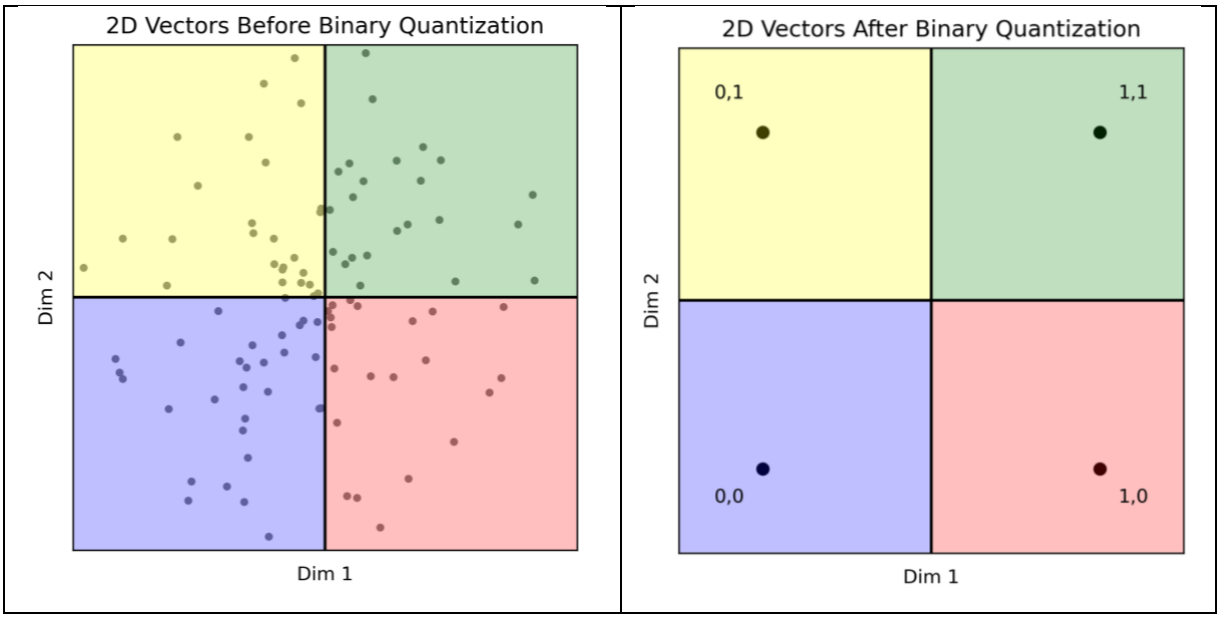
a float32 down to 1-bit. However, reversing this process is impossible - making this a lossy compression technique.



Distance between two binarized vectors - count the differing bits (Hamming distance) using bitwise operations, faster than calculating distances between non-binarized vectors. Example, vectors 11010101 and 11001101 differ by two bits, indicating a Hamming distance of 2.

Binary quantization (BQ) is not suited for all data types; it requires **normalized data** when using the cosine metric. Normalization ensures data is evenly distributed across predefined quantized regions, improving the effectiveness of BQ. In contrast, non-normalized data may lead all vectors to be encoded identically, which is problematic for differentiation.

In one-dimensional BQ, there are only two possible outcomes: vectors are encoded as 1 for positive and 0 for negative. Extending this to two dimensions, vectors lie within a unit circle and are divided into four regions, each represented by a two-bit code. This segmentation allows distinct and efficient representation of the data.



Collision between two vectors = vectors have the same representation once binarized (see above – all vectors in each of the 4 regions have only one 2D binary representation.
In **higher dimensions**, BQ enables an exponential increase in distinct regions (2^N), which **dramatically reduces the likelihood of collisions** (multiple vectors sharing the same binary code). For example, with 756 dimensions, the probability of collisions becomes negligible, even with trillions of vectors.
Performance Improvements with BQ
Improvements:

- Binary quantization enhances **performance** by allowing distances between vectors to be computed with simple XOR operations. In Weaviate, searching through binarized vectors is much faster than using uncompressed vectors, demonstrating significant improvements in processing time across various dimensions (e.g., 768, 1536, 4608 dimensions).
- **Indexing Time Improvements**: The use of binary quantization in indexing can drastically reduce the time and memory required. Comparisons of flat and HNSW indices show that incorporating BQ can halve indexing times and significantly decrease memory usage, providing a robust alternative to traditional methods.
- **Memory Footprint Improvements**: Different configurations of data storage (flat index with/without BQ and cache) show varied memory footprints. Storing compressed data reduces memory demands compared to storing uncompressed data or using proximity graphs, which is advantageous in large-scale applications.
- **Latency Improvements**: Analyses of latency versus recall for DBPedia vectors using different indices reveal that binary quantization can streamline brute-force searches, improving query speed while maintaining low memory usage. The flat index with cached compressed vectors strikes a balance between speed and memory efficiency, suitable for high-recall applications.

Embedding Truncation

Excellent notebook for training text embedding models whose **embeddings can be truncated with minimal performance loss**. This allows for **faster retrieval, clustering**, etc. Plus, you can train the model on your domain for better performance. Notebook: https://lnkd.in/ezRrRy7r.
Learn more about training embedding models here: https://sbert.net/

SELF-RAG

Self-RAG is an innovative enhancement of the Retrieval-Augmented Generation (RAG) system. It addresses limitations of RAG by introducing a self-reflective mechanism for better LLM response's quality and factuality through context understanding. The framework trains a single arbitrary LM that adaptively retrieves passages on-demand, and generates and reflects on retrieved passages and its own generations using special tokens, called reflection tokens. Self-RAG significantly outperforms ChatGPT and retrieval-augmented Llama2-chat on Open-domain QA, reasoning and fact verification tasks, and it shows significant gains in improving factuality and citation accuracy for long-form generations relative to these models.

Introduction:

SELF-RAG Algorithm Step-by-Step Description

1. **Input Prompt**: Receive an input prompt and generate initial segments.
2. **Evaluate Retrieval Need**: Predict if retrieval is necessary using a "Retrieve" token.
    - If "Retrieve" is "Yes", proceed to the next step.
    - If "Retrieve" is "No", generate the next segment without retrieval.
3. **Retrieve Documents**: Retrieve relevant text passages from a large-scale passage collection based on the input prompt and preceding generation.
4. **Generate Task Output**: For each retrieved passage, generate corresponding task outputs.
5. **Evaluate Relevance**: Predict the relevance of each passage to the task using an "ISREL" token (Relevant/Irrelevant).
6. **Evaluate Support**: Predict if the information in the generated segment is supported by the retrieved passage using an "ISSUP" token (Fully Supported/Partially Supported/No Support).
7. **Evaluate Usefulness**: Assign an overall utility score to the generated segment using an "ISUSE" token (Scale of 1-5).
8. **Rank and Select**: Rank all generated segments based on relevance, support, and usefulness scores.
9. **Generate Final Response**: Select the best segment(s) and generate the final response.

This process ensures that the generation is informed by relevant and supported information, enhancing the quality and factual accuracy of the output.

Corrective Retrieval-Augmented Generation (CRAG)

Introduction: CRAG, or Corrective Retrieval-Augmented Generation, is an advanced RAG technique designed to enhance response accuracy by refining the quality of retrieved documents before passing them to the LLM for response generation. Corrective mechanisms + advanced retrieval and refinement strategies = more accurate and reliable outputs.

Paper: https://lnkd.in/e2MA9qHY

Code github: https://lnkd.in/e5wCr7Ti

Motivation:

- **Problem with Vanilla RAG:** Retrieves many irrelevant or redundant documents, leading to poor response quality.
- **Solution:** Introduce an evaluator to filter and refine retrieved documents, ensuring only relevant, high-quality information is used.

CRAG Workflow:

1. **Initial Retrieval:** Retrieve documents D1 and D2 from the vector database based on the query.
2. **Evaluation:** Use a lightweight T5 model (7 million parameters) as an evaluator to assess the quality of retrieved documents.
    - **Actions:** Classify documents into three categories: Correct, Ambiguous, and Incorrect.
3. **Processing Based on Evaluation:**
    - **Correct Documents:** Directly passed to the LLM for response generation.
    - **Ambiguous Documents:** Supplemented with external web search results, re-evaluated, and only relevant parts are used.
    - **Incorrect Documents:** Replaced entirely with web search results. To overcome the limitations of static and limited corpora, CRAG incorporates large-scale web searches. This approach expands the scope of retrieval to include more diverse and potentially relevant documents
4. **Decompose and Recompose:** selectively focus on key info within retrieved docs while filtering irrelevant parts - split relevant documents into smaller chunks (strips), evaluate them again, and remove redundant or irrelevant information.

Advantages:

- **Flexibility:** No need to fine-tune the LLM; the evaluator is a separate, smaller model.
- **Ease of Integration:** Can be easily integrated into existing RAG systems without major changes.

Comparison with Self-RAG:

- **Self-RAG:** Requires fine-tuning the LLM for specific tasks, less flexible for changing tasks.
- **CRAG:** Uses a separate evaluator, making it more adaptable and easier to integrate with existing systems.

Performance:

- **CRAG vs. Self-RAG:** CRAG shows higher accuracy (e.g., 61% vs. 55% in certain datasets) and outperforms proprietary models like ChatGPT and Perplexity AI in some benchmarks.

Production Consideration:

- **CRAG:** Lightweight, easy to implement, and flexible, making it superior for production environments compared to the more comprehensive Self-RAG approach.

As tested on four datasets for short- and long-form generation tasks, CRAG significantly improved RAG performance – it’s effective in enhancing the robustness of generated texts.

GNN-RAG

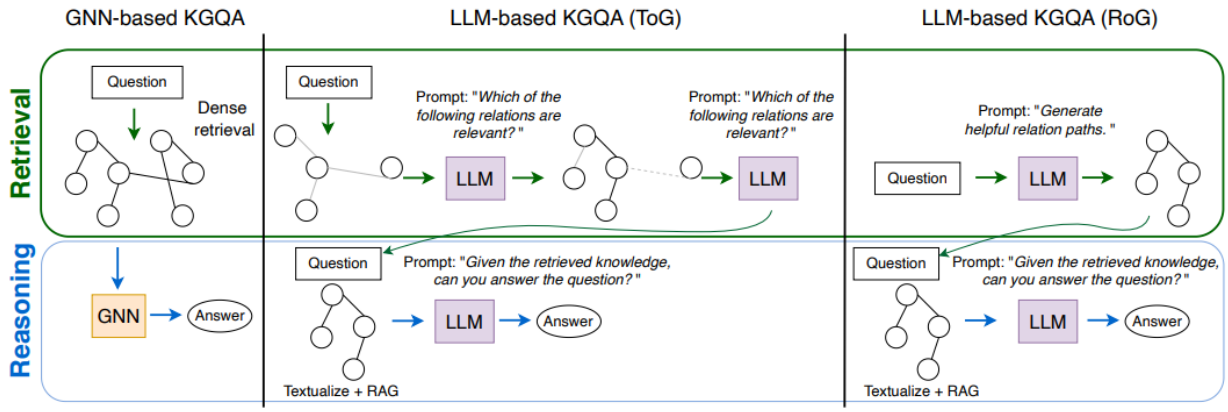


Figure 2: The landscape of existing KGQA methods. GNN-based methods reason on dense subgraphs as they can handle complex and multi-hop graph information. LLM-based methods employ the same LLM for both retrieval and reasoning due to its ability to understand natural language.

GNN-RAG innovatively combines the language understanding abilities of Large Language Models (LLMs) with the reasoning abilities of Graph Neural Networks (GNNs) in a retrieval-augmented generation (RAG) style, particularly for Question Answering over Knowledge Graphs (KGQA), significantly improving KGQA performance without incurring additional computational costs. Knowledge Graphs (KGs) represent human-crafted factual knowledge in the form of triplets (head, relation, tail), which collectively form a graph. The task of KGQA involves answering natural questions by grounding the reasoning in the information provided by the KG. KGQA methods are categorized into Semantic Parsing (SP) and Information Retrieval (IR) approaches. SP methods convert questions into logical queries, executing them over KGs for answers, but they rely on annotated queries and may generate non-executable ones. IR methods operate in weakly-supervised settings, retrieving KG information for question answering without explicit query annotations. Integrating Graph Neural Networks (GNNs) with RAG improves KGQA, outperforming existing methods by utilizing GNNs for retrieval and RAG for reasoning.
LLMs are renowned for their exceptional natural language understanding capabilities, derived from extensive pretraining on vast textual data. However, their adaptation to new or domain-specific knowledge can be limited, leading to inaccuracies. Conversely, GNNs are adept at handling complex graph information stored in KGs, making them suitable for KGQA tasks. In this context, GNN-RAG combines the strengths of both models to enhance performance. GNN-RAG improves vanilla LLMs on KGQA and outperforms or matches GPT-4 performance with a 7B tuned LLM.

In the GNN-RAG framework, a GNN reasons over a dense KG subgraph to retrieve answer candidates for a given question. The shortest paths in the KG that connect question entities and answer candidates are extracted to represent KG reasoning paths. These paths are then verbalized and given as input for LLM reasoning with RAG. This integration allows GNN-RAG to excel in handling complex multi-hop and multi-entity questions by retrieving and reasoning over intricate graph structures, outperforming competing approaches by 9-15% points at answer F1.

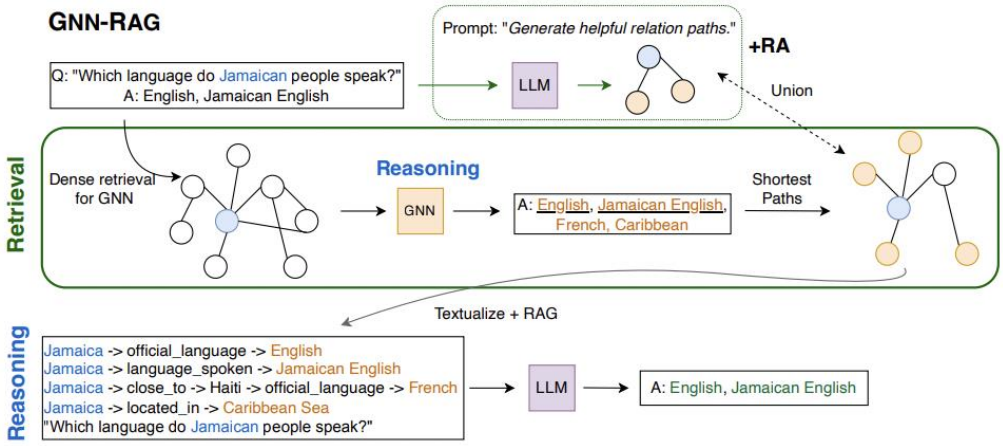


Figure 3: GNN-RAG: The GNN reasons over a dense subgraph to retrieve candidate answers, along with the corresponding reasoning paths (shortest paths from question entities to answers). The retrieved reasoning paths –optionally combined with retrieval augmentation (RA)– are verbalized and given to the LLM for RAG.

Furthermore, the framework employs a retrieval augmentation (RA) technique to further boost KGQA performance. By combining GNN and LLM-based retrievals, GNN-RAG maximizes answer diversity and recall, enhancing overall performance. Experimental results demonstrate that GNN-RAG achieves state-of-the-art performance on two widely used KGQA benchmarks (WebQSP and CWQ), outperforming or matching GPT-4 performance with a 7B tuned LLM. Notably, GNN-RAG excels in multi-hop and multi-entity questions, outperforming competing approaches by 8.9-15.5% points at answer F1.

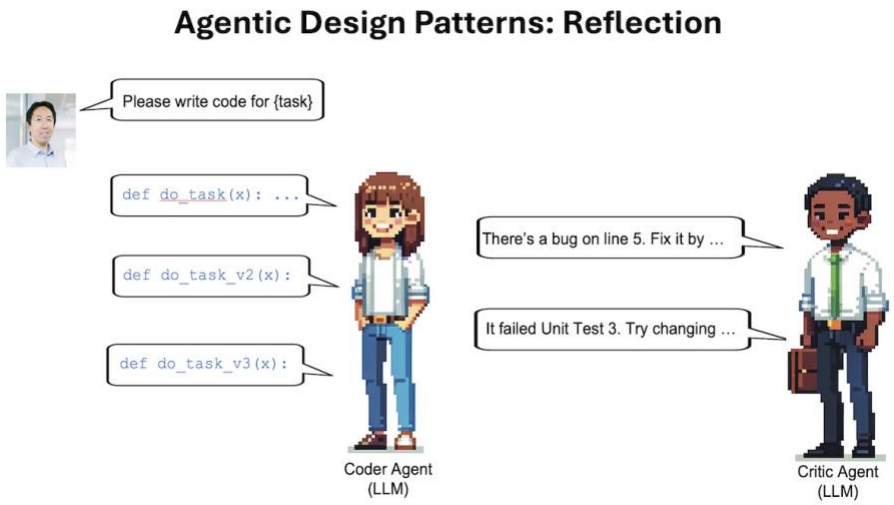
GNN-RAG showcases the recent trend of integrating tree and graph-based approaches with LLMs to enhance their reasoning abilities. Researchers from the University of Minnesota have shown that GNN-RAG can retrieve multi-hop information necessary for faithful LLM reasoning on complex questions.

(paper | tweet)

Agents

Agentic Workflows

(by Andrew Ng)



Instead of having an LLM generate its final output directly, an *agentic workflow* **prompts LLM multiple times**, giving it opportunities to *build higher-quality output step by step*. Four design patterns for AI agentic workflows:

- Reflection
- Tool use
- Planning
- Multi-agent collaboration.

Reflection

Imagine you prompt LLM, receive unsatisfactory output, and delivering critical feedback to LLM to improve its response. **Reflection automates the step of delivering critical feedback** => *model automatically criticizes its own output* and improves its response.

Example:

- **Ask LLM to generate desired code** (or write text, answer questions, etc.)
- **Prompt LLM for constructive criticism** by reflecting on its output: `Here's code for task X: [previously generated code]. Check it carefully for correctness, style, and efficiency, and give constructive criticism for how to improve it.`
- **Prompt the LLM to use feedback to rewrite code** by providing context (i) previously generated code (ii) constructive feedback.
- **Repeat criticism/rewrite process** to yield further improvements.

Beyond self-reflection, **we can give LLM tools to evaluate its output**: e.g. run its code through *unit tests* to see if test cases are passed or *search the web* to double-check text output. Then it can reflect on error found and improve results further.

Reflection w/multi-agent framework - create two agents, *one generates good outputs and the other is prompted for constructive criticism*. The discussion between the two leads to improved responses.

Reflection is simple, but it improves the final output. To learn more:

- Self-Refine: Iterative Refinement with Self-Feedback, by Madaan et al. (2023)
- Reflexion: Language Agents with Verbal Reinforcement Learning, by Shinn et al. (2023)
- CRITIC: Large Language Models Can Self-Correct with Tool-Interactive Critiquing, by Gou et al. (2024)

AIOS: The First LLM Agent Operating System

Designed for optimal operation of OS: resource allocation, facilitating context switches, concurrent execution, tool services for agents, access control, and providing a rich toolkit for developers. Based on **several key agents that orchestrate the others**: Agent Scheduler, Context Manager, etc. Those agents communicate with the AIOS SDK in an interactive mode, along with non-LLM tasks coming from the OS Kernel - integrating complex AI functionalities into traditional OS. This is a shift in the way we interact with machines. E.g. Apple's ReALM models capable of understanding not only conversation, but also on-screen and background jobs information - **new era of intelligent computing**.
Andrey Karpathy: *"Looking at **LLMs as chatbots** is the same as looking at **early computers as calculators**. We're seeing an emergence of a whole new computing paradigm, and **it is very early**."*

More Agents Is All You Need

Adding more agents increases LLMs accuracy using a simple sampling-and-voting technique - tested across various LLM benchmarks. Significant improvement of performance, especially for complex tasks. For instance, with 15 agents, Llama2-13B equals Llama2-70B's accuracy,

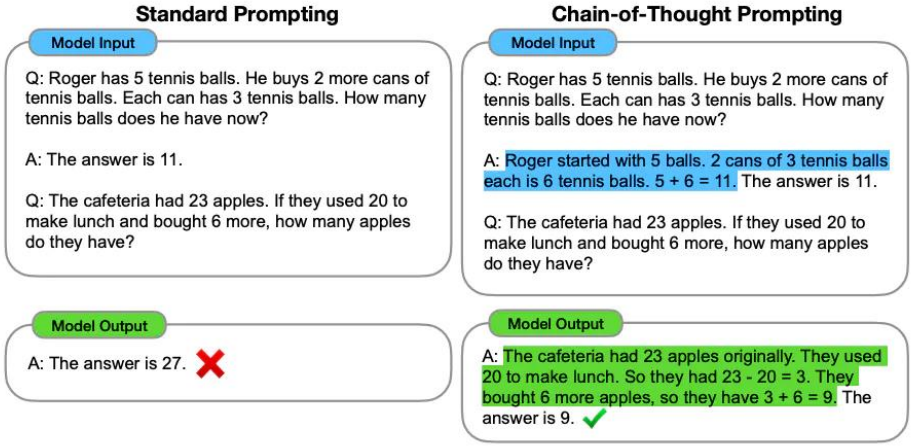
AI Agents for Automated Sports(IPL) News Generation using Llama 3, Crew AI

To summarize

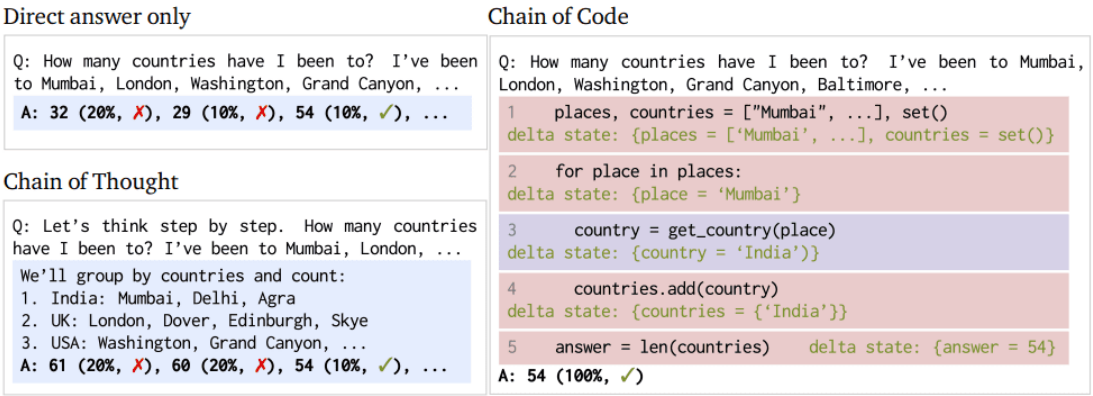
Principles of Prompt Engineering

Strategies

- **Standard prompting**: Answer the following question based on the prompt below
- **Automatic prompt engineering**: Use LLM to iterate on your prompt and improve its quality, for example based on your classifier's performance.
- **Chain of thought**: LLM is encouraged to solve problems step-by-step instead as a whole. Each example **explains how the problem is solved step-by-step**: the *problem is broken into small parts that are solved individually*. Averages few-shot learning by inserting several *examples of solved reasoning problems* within the prompt. Improves performance on reasoning tasks: commonsense or symbolic reasoning.



- **Chain of code** (outperforms CoT): *encourage LLM to format linguistic or arithmetic sub-tasks in a program* as flexible pseudocode which the compiler can either execute or catch undefined behaviors and hand off to emulate/simulate with an LLM (LMulator). This broadens the scope of problems LMs can tackle and demoes AI potential to handle more complex, real-world tasks that require nuanced thinking. Steps:
  - **Define task**: Determine a linguistic or arithmetic task that requires reasoning.
  - **Code**: LM writes code / pseudocode to outline a solution.
  - **Emulation of code**: for non-executable parts of code, the LM emulates the expected outcome, simulating the code execution.
  - **Combining outputs**: LM combines the results of actual code execution + emulation to form a comprehensive solution.



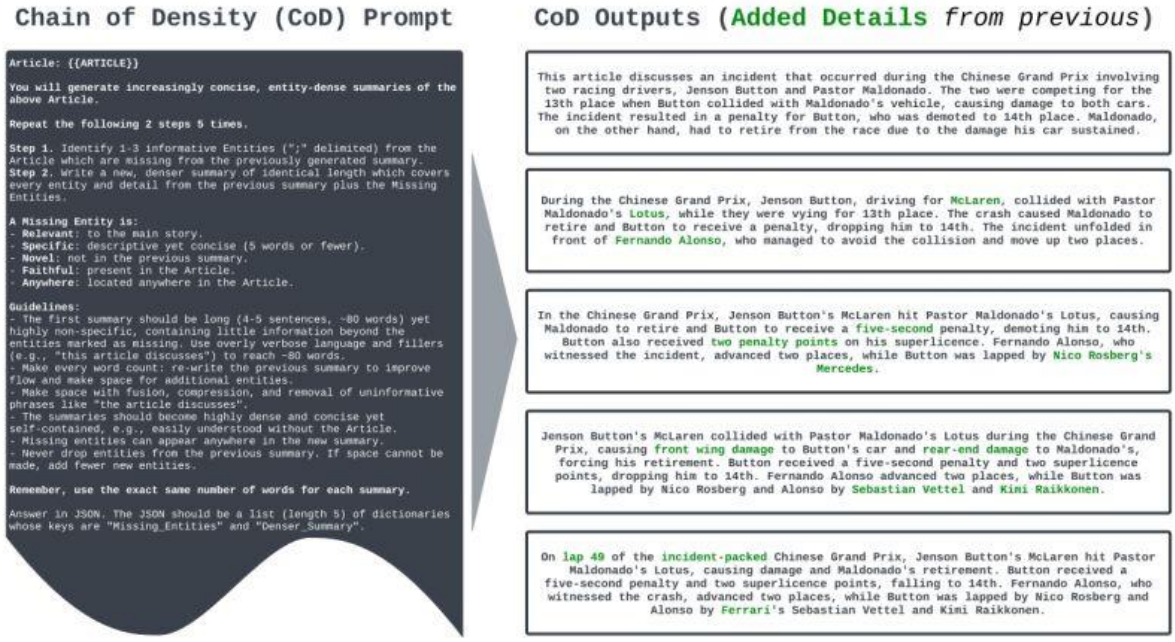
Chain of Density (CoD)

Helps generate **dense summaries with a higher concentration of information**. Too much info reduces readability and coherence => CoD strikes the **balance between informativeness and readability**.

1. *Initial summary* is generated and made increasingly entity dense.

2. In fixed # turns, the *density of previous summary is increased by adding essential details (missing entities)* from the source text while *keeping the length same* by removing non-essential information.

On average, intermediate CoD summaries can achieve clarity-informativeness balance.



Retrieval Augmented Thoughts (RAT)

RAG + COT ⇒ RAT - iterative CoT prompting with information retrieval to mitigate hallucinations.

- Prompt LLM w/ *zero-shot CoT* (using RAG?)
- Retrieve information and each CoT reasoning step.
- Revise CoT steps based on the retrieved context.
- Generate a response using *revised CoT steps and context*.

RAG alleviates hallucinations with within the intermediate reasoning process, while RAT does it for *sophisticated long-horizon reasoning* as a **progressive approach where LLMs produce the response step-by-step following the CoT (a series of subtasks)**, and *only the current thought step will be revised based on the information retrieved with task prompt, the current and the past CoTs*. This is analogous to human reasoning: utilizing outside knowledge to adjust the step-by-step thinking during complex long-horizon problem-solving.

Notes: A lot of LLM Calls per answer. RAT is better than simple RAG. Relatively increasing rating scores by 13.63% on code generation, 16.96% on mathematical reasoning, 19.2% on creative writing, and 42.78% on embodied task planning

Retrieval Augmented Generation + Chain of Thought (COT)

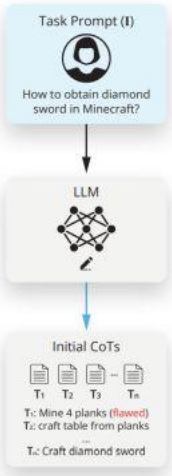
Step 0

Draft initial step-by-step zero-shot CoTs based on the task prompt.

A task prompt is given by a human user.

LLM makes zero-shot step-by-step reasoning based on the prompt.

This initial zero-shot CoT answer may be flawed.



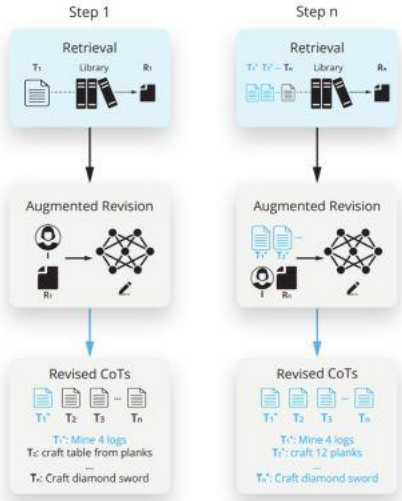
Step 1 - Step n

Retrieve relevant information and iteratively revise each CoT with all previous generations in context.

Retrieve with the task prompt and previous generated CoTs.

LLM revises the i-th CoT based on retrieved content.

The i-th CoT is replaced with the revised generation.



RankPrompt: Step-by-Step Comparisons Make LLMs Better Reasoners

https://arxiv.org/abs/2403.12373 (2024)

LLMs, even ChatGPT, are prone to logical errors during reasoning. To address this - RankPrompt, a new prompting method, enables *LLMs to self-rank their responses* using in-context learning, without additional resources:

- *Breaks down the ranking problem into a series of comparisons* among diverse responses,
- Leveraging the inherent capabilities of LLMs to *generate chains of comparison as contextual exemplars*.
- Experiments across 11 arithmetic and commonsense reasoning tasks - RankPrompt enhances the reasoning performance of ChatGPT and GPT-4 (up to 13%), excels in LLM-based automatic evaluations for open-ended tasks - aligning w/human judgments 74% of the time in the AlpacaEval dataset.
- RankPrompt - effective method to eliciting high-quality feedback from LLM.

WISER Framework

- **W** Who - Assign an identity or role
- **I** Instructions - Tell the model what to do
- **S** Sub-tasks - Break it down into simpler steps
- **E** Example - Provide examples (if applicable)
- **R** Review - Look at output / evaluation metrics and iterate

1. **Write Clear and Specific Instructions**

Provide clear, specific instructions: longer prompts, full of context and details, often yield more accurate results.

Tactics

- Use Delimiters for Clarity (triple backticks)
- Ask for structured output (JSON)
- Ask the model to check conditions (if then)
- Provide examples ("Few-shot" prompts)

2. **Give the model time to "think"**

So the model can think step by step before giving the final answer.

Tactics

- Specify the steps (first summarize, then translate, then list each name, fourth, output a JSON)
- Have the model to work out its own solution (first, create your solution to this math problem, then compare it to the student's response)

3. **Balance specificity with creativity** (be specific, but let the model be creative)

Tactics

- Providing exhaustive details and context
- **Top p** - high p increase randomness / creativity in next word prediction, while lower p - makes next-token selection more predictable (default 1.0) AND **temperature** – higher temperature increases randomness / creativity (default 1.0)

4. "**Act as**…": The hack of letting ChatGPT act as someone or something has proven to be **extremely powerful**.
5. Model can lie/hallucinate – **always double-check** (e.g. ask for documentation **as proof**)
6. Iterate to find more efficient prompts (change words)
7. **Itemize instructions** (better to understand vs. long paragraphs)
8. **Avoid negations** (confuses model)
9. **Chain-of-thought** prompting

Temperature Explained

Temperature is a hyperparameter that affects the probability distribution of the tokens generated by the GPT model. By adjusting the temperature, we can **control the diversity and creativity** of the generated text. Higher temperature values result in more creative outputs, lower values - more focused and deterministic generated text.

Mathematically, the temperature is **incorporated into the softmax function** (converts raw logits produced by the GPT model into a probability distribution):

$p_i = \frac{\exp(z_i)}{\sum_{j=1}^{N} \exp(z_j)}$	$p_i = \frac{\exp(x_i/\tau)}{\sum_{j=1}^{N} \exp(x_j/\tau)}$
$\sigma(\mathbf{z})_i = \frac{e^{\beta z_i}}{\sum_{j=1}^{K} e^{\beta z_j}}$ or $\sigma(\mathbf{z})_i = \frac{e^{-\beta z_i}}{\sum_{j=1}^{K} e^{-\beta z_j}}$ for $i = 1, \ldots, K$.	

The parameter tau is called temperature and controls the softness of proba distribution. When tau gets lower, the biggest value in x get more probability, when tau gets larger - the proba will be split more evenly on different elements.

Prompt Injections (jailbreaking)

Allow to access LLM fine-tuning instructions and uploaded files by *tricking the chatbot into behaving in a way it was told not to*. Early prompt injections asked to ignore instructions not to produce hate speech or other harmful content. More sophisticated prompt injections have used *multiple layers of deception* or hidden messages in images and websites to show how attackers can steal people's data. The creators of LLMs have put rules in place to stop common prompt injections from working, but there are no easy fixes. It is very easy to exploit vulnerabilities - sometimes requires only basic proficiency in English. *The work to defend bots against prompt injection issues is ongoing, as people find new ways to hack chatbots* and avoid their rules – the jailbreak game is never-ending.

Function Calling

Developers can now **describe functions** to LLM, and have the model intelligently choose to output a JSON object containing arguments to call those functions. Example: define function for weather as json output = { 'city': city, 'state': state, 'country': country, 'temperature': degrees Farenheit}. By providing

schemas for "functions", the LLM will choose one and attempt to **output a response matching that schema**.

Ethical Concerns of Foundation Models

- "Hallucinations" - False answers. Statistical approximation. No actual understanding of content.
- Training data issues - obsolete, biased, lack author consent, terms of use, etc.
- IP & copyright infringements - proprietary training data, Github Copilot lawsuit
- Deep fakes - models can generate fake content in needed style
- Fraud and abuse - false reviews, false papers, WormGPT, spam, phishing, etc.
- Not checking output - users not verifying information generated by the model before using
- Blackbox nature - actions in model are opaque, difficult to understand model reasoning
- Exposure of information - confidential information or PII exposed in training data

LLM Evaluation

- Regular metrics (Accuracy, etc)
- Reference matching - semantic similarity OR ask [another] LLM: "Are these two answers factually consistent", etc
- "Which is better" - ask [another] LLM: "Which of the two answers is better?" (according to any criteria you want)
- Ask an LLM whether the new answer incorporates the feedback on the old answer
- Static metrics - verify the output structure (json), ask an LLM to grade the answer (on a scale 1-5), etc.
- LLM leaderboards and test sets. HumanEval (programming benchmark), MMLU (lang understand)

LLM Distillation

Using ChatGPT zero-shot preds to provide labels, then training a smaller model on these labels

LLM Deployment

Vertex AI + Llama 2

Popular Multi-Star Course on LLMs

Includes fine-tuning Llama 2, Mistral, quantization in Google Colab and many more!

TinyLlama-1.1B-Chat-v1.0

A compact 1.1 billion parameter model, pretrained in 90 days on 16 A100-40G GPUs, is compatible with Llama 2, optimized for various applications. It offers enhanced dialogue generation, accessible through Hugging Face Transformers for advanced text generation tasks.

mixtral-offloading (☆ 1.3k)

Run Mixtral-8x7B models on free Colab or consumer desktops by enhancing model inference through mixed quantization and a unique MoE offloading strategy, fitting models within combined GPU and CPU memory. Includes a demo notebook.

Proxy-tuning

(More details including code examples: https://lightning.ai/lightning-ai/studios/improve-llms-with-proxy-tuning?view=public)

Proxy-tuning is a way to adapt LLMs without changing the model's weights. This is especially attractive if a given LLM is too resource-intensive to train or if a user doesn't have access to the LLM's weights. Steps:

1. **Select a base LLM** (e.g., an untuned 7B Llama 2 model) smaller and cheaper than the target LLM (e.g., a 10x larger, untuned 70B Llama 2 model).

2. **Finetune the base LLM** to obtain a small finetuned LLM (e.g., instruction-finetune a 7B Llama 2 model to get a finetuned 7B model).

3. **Compute the output difference** between the base model (step 1) and the tuned base model (step 2).

4. **Add this difference in outputs** to the target LLM's outputs.

5. **Normalize** the modified outputs from step 4, and **then generate the answer**.

I tried the following query: "If I have 5 apples and eat 2, but then find 3 more on my way home, how many do I have?" The proxy-tuned model was indeed able to answer correctly, whereas the base models failed: "You start with 5 apples and eat 2, so you have 5 - 2 = 3 apples left. Then, you find 3 more apples on your way home, so you have 3 + 3 = 6 apples in total."

Using the same approach, it was also possible to give Llama 2 13B coding abilities via CodeLlama 7B. Short code example (longer examples at the above link):

```
generated_tokens = []

input_txt = (
  "If I have 5 apples and eat 2, but then find 3 more"
  " on my way home, how many do I have?"
)
input_ids = tokenizer.encode(input_text)

for _ in range(max_length):
    # Obtain logits
    logits_base = model_base(input_ids).logits # Llama 7B Base
    logits_tuned = model_tuned(input_ids).logits # Llama 7B Chat
    logits_target = model_target(input_ids).logits # Llama 70B Base

    # Apply proxy-tuning
    logits = (
        logits_target + (logits_tuned - logits_base)
    )

    # Normalize logits and obtain token
    predictions = torch.softmax(logits[:, -1, :], dim=-1)
    next_token_id = torch.argmax(predictions).unsqueeze(0)
    generated_tokens.append(next_token_id.item())

generated_text = tokenizer.decode(generated_tokens)
print(generated_text)

# Output:
# You start with 5 apples and eat 2,
# so you have 5 - 2 = 3 apples left.
# Then, you find 3 more apples on your way home,
# so you have 3 + 3 = 6 apples in total.
```

TRANSFORMERS

Reference: https://towardsdatascience.com/transformers-explained-visually-part-1-overview-of-functionality-95a6dd460452

Encoder-only architecture (BERT, RoBERTa, DeBERTa) – **converts input text into a rich numerical representation** for further text classification, NER, Q&A, summary, etc. The computed representation for each token depends both on the left (before the token) and the right (after the token) contexts – bidirectional attention.

Decoder-only architecture (GPT). The model was trained using generative pre-training – it **autocompletes the sequence by iteratively predicting the most probable next word (**as a softmax distrib. over the entire vocab.). Text generation, Q&A. The computed representation for each token depends only on the left context = causal or autoregressive attention.

Encoder-decoder architecture (T5, BART) – complex **mapping from one text sequence to another** for machine translation, summarization.

Subword tokenization (e.g. Wordpiece) – split rare words into smaller units to deal with **complex words** and **misspellings**, but keeping frequent words as unique entities (more efficient + managing the size).

Transformer contains a stack of Encoder layers (Encoders) and Decoder layers. To avoid confusion we will refer to the individual layer as an Encoder or a Decoder and will use Encoder stack or Decoder stack for a group of Encoder layers.

The Encoder stack and the Decoder stack each have their corresponding Embedding layers for their respective inputs. Finally, there is an Output layer to generate the final output.

Data inputs for both the Encoder and Decoder, which contains:

- ✓ Embedding layer
- ✓ Position Encoding layer

Each **Encoder** in the encoder stack contains:

- ✓ *Multi-Head Attention layer*
- ✓ Feed-forward layer

Each **Decoder** in the decoder stack contains:

- ✓ *Two Multi-Head Attention layers*
- ✓ Feed-forward layer

Output (top right) — generates the final output, and contains:

- ✓ Linear layer
- ✓ Softmax layer.

Summary: Transformer vs. LSTMs

July 4, 2023 by Emily Rosemary Collins
https://blog.finxter.com/transformer-vs-lstm/
More in-depth source: https://nlp.seas.harvard.edu/annotated-transformer/

Transformer advantages

- **Parallel processing** => transformers process input seq in parallel, not sequentially, because they use *self-attention mechanisms that can process multiple tokens simultaneously* instead of relying on recurrent connections that process the next input based on the prev input (in a sequential manner) - in order to encode 2nd token in a seq we need hidden states for 1st token, therefore need to compute that first => transformers are highly efficient, scalable, perfect for handling large-scale NLP tasks. This means they have faster training times on GPU vs. sequential RNN / LSTM.
- More effective handling of **long-range dependencies**, due to the **self-attention mechanism** that weighs importance of tokens in input seq to help learn relations between words, whereas LSTMs might struggle with retaining info from distant positions in longer sequences. **Attention mechanism lets one focus on different parts of input sequence simultaneously and without regard to distance** (not sequentially). RNN / LSTM - **sequential processing** when the word's encoding strongly affects only the next word's representation => its influence fades after a few time steps (may be even lost in the final context vector). **Bi-directional models** encode same seq from start to end, then from end to star - words at the end of a sentence have stronger influence on the hidden representation, but is still a workaround, not a real solution for very long dependencies.
- **Flexibility**: easily adapted for different tasks. They have been used successfully in a wide range of applications, from text generation to image recognition.
- **CNNs** solve dependencies by **applying different kernels to same sentence**: kernel of size 2 learns relationships between bigrams, kernel of size 3 – trigrams => # different kernels to capture dependencies among all possible exponentially growing combinations of words in a sentence would be enormous and unpractical.

Transformer disadvantage:

- **Memory constraints - require more memory and computation power** vs. RNNs and LSTMs due to extensive use of **self-attention mechanisms** => using Transformers on **edge devices** w/limited CPU and memory can be challenging + for deep Transformers, **training requires significant computational resources** – may not always be feasible. Choice of model should be based on specific reqs and constraints at hand.

Although not a disadvantage, but a limitation - transformers can capture only dependencies within the **fixed input size** max_len (if 50 tokens, then only 50). Transformer-XL tries to overcome this by kinda re-introducing recursion by storing hidden states of encoded sentences.

Architecture: encoder and decoder stacks composed of multiple layers, each layer w/two components: multi-head self-attention and position-wise feed-forward networks (FFN).

- **Encoder** processes input seq and generates continuous representation preserving the contextual info.
- **Decoder** takes this encoder representation and generates the target seq. It has multi-head self-attention mechanism + encoder-decoder attention mechanism helping to focus on different parts of input seq.

Positional embeddings - unlike RNNs / LSTMs *transformers* don't process data sequentially => they *do not understand the order* of the input tokens (which is crucial for NLU). To address this, positional embeddings are added to the input embeddings before feeding them into the transformer model. Input embeddings represent tokens (e.g., words) in a high-dimensional space where similar tokens are closer together, but they do not contain token order info. *Positional embeddings = vectors that encode the position of each token in seq.*, created by applying sine and cosine f(x)-s of different frequencies to each position index - these functions will generate *unique values for each position*, and *by varying the frequencies, the model can learn to distinguish between different positions in seq*. The use of sine and cosine f(x)-s also ensures that the positional embeddings are *continuous* and *can generalize well to sequence lengths unseen during training*. Positional embeds + input embeddings = transformer model understands both meaning of individual tokens and their position in seq. This allows the model to effectively process sequences of data, such as sentences in natural language processing tasks, and make decisions based on both the content and order of the tokens.

Residual connections allow model to preserve info from earlier layers and help w/vanishing gradient – each sub-layer (multi-head self-attention + feed-forward network) *has a residual connection followed by layer normalization step*. This means that **the output of each sub-layer is added to its input**, and this sum is then normalized before being fed to the next sub-layer.

Why do we need **normalization**: from paper - when stacking more layers in a network, the issue of **vanishing / exploding gradients** arises which hampers convergence. This was addressed by normalized initialization and **intermediate normalization layers** which enable networks with tens of layers to start converging for SGD w/backpropagation.

Why do we need **residual connections**: to address the fact that with greater network depth (when adding more layers to deep model), accuracy gets saturated and degrades rapidly and this is not caused by overfitting. Original mapping denoted H(x), residual mapping: F(x) + x

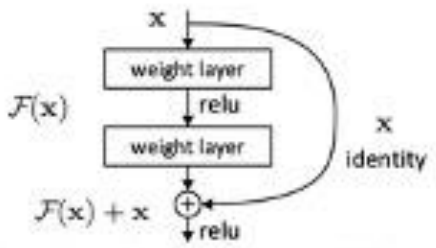


Figure 2. Residual learning: a building block.

The formulation of F (x) + x can be “shortcut connections” in feedforward networks (Fig. 2) = connections that skip one or more layers. In our case, shortcut connections perform identity mapping - their outputs added to outputs of stacked layers (Fig. 2) – no extra parameters or computational complexity, network still trains by SGD w/backpropagation.

The actual implementation from https://nlp.seas.harvard.edu/annotated-transformer/:

```
class LayerNorm(nn.Module):
    "Construct a layernorm module (See citation for details)."

    def __init__(self, features, eps=1e-6):
        super(LayerNorm, self).__init__()
        self.a_2 = nn.Parameter(torch.ones(features))
        self.b_2 = nn.Parameter(torch.zeros(features))
        self.eps = eps

    def forward(self, x):
        mean = x.mean(-1, keepdim=True)
        std = x.std(-1, keepdim=True)
        return self.a_2 * (x - mean) / (std + self.eps) + self.b_2
```

That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. We apply dropout (cite) to the output of each sub-layer, before it is added to the sub-layer input and normalized.

To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

```
class SublayerConnection(nn.Module):
    """
    A residual connection followed by a layer norm.
    Note for code simplicity the norm is first as opposed to last.
    """

    def __init__(self, size, dropout):
        super(SublayerConnection, self).__init__()
        self.norm = LayerNorm(size)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, sublayer):
        "Apply residual connection to any sublayer with the same size."
        return x + self.dropout(sublayer(self.norm(x)))
```

Attention Explained

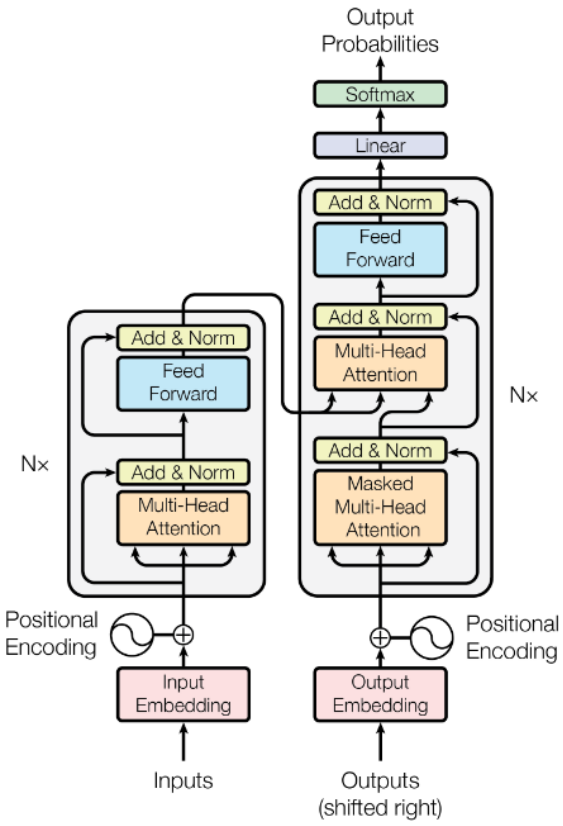


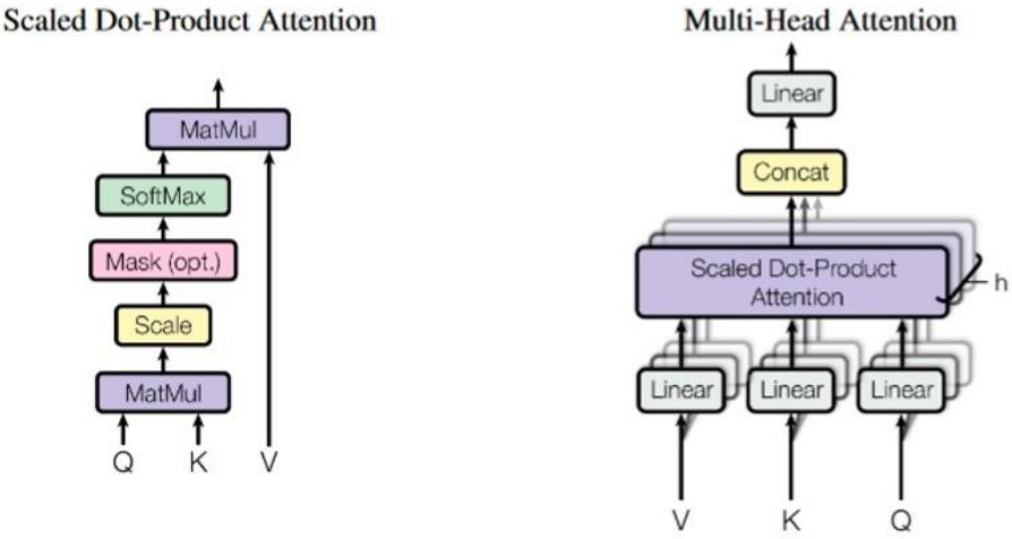
Figure 1: The Transformer - model architecture.

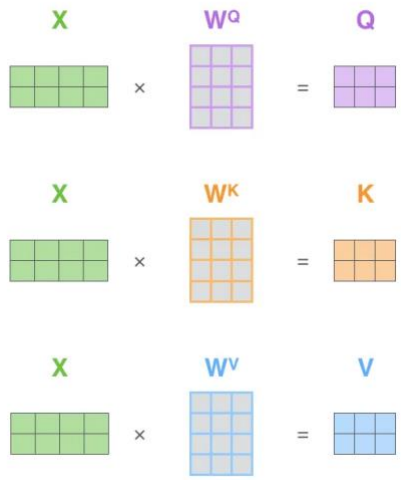
Attention mechanism allows models to **selectively focus on certain parts of input seq**, esp. useful in seq2seq tasks. From "Attention is all you need": Attention f(x) = **maps a query and a set of key-value pairs to an output** (all 4 are vectors).

This helps generate *more relevant output* by computing weighted sum of values dependent on query and corresponding keys (query determines which values to focus on = query "attends" to the values), weights obtained by comparing each input to the rest of the inputs in the same seq - model implicitly learns the **relationships between elements in seq**.

Output = weighted sum of the values where the *weight assigned to each value* is computed by compatibility f(x) of the query with the corresponding key.

Queries compare input elements, keys and values represent the relationship between the elements. **Softmax** f(x) applied to computed attention weights to form probability distribution, **emphasizing the most relevant elements** in the sequence. Self-attention solves the problems of parallelization and loss of contextual information for distant words within a seq.





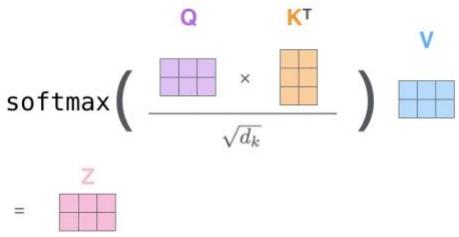
Step 1 - Linear Transformations: seq of embeddings passed through transformer's input layers (encoder and decoder) and **each embedding** experiences **three separate linear transformations** (by multiplying it by 3 weight matrices) resulting in three vectors — **query, key, and value**. The proper **weights are learned** through training.

Each weight matrix is **initialized randomly** => resultant vectors each learn some different information about the embedding. This is important when calculating the attention score since we don't want to just derive the dot product of the vector with itself.

Once we have these 3 vectors for each embedding in seq, we can calculate **attention score** which measures the **strength of relationship between a word** in the seq **with all other words**:

1. Take the **query vector** for a word and calculate it's dot product with the transpose of the **key vector** of each word in the sequence — including itself. This is the attention score / weight.
2. Then **divide** each of the results **by square root of the dimension of key vector**. This is the **scaled attention score**.
3. Pass them through a **softmax function**, so that values are contained between 0 and 1.
4. Take **each value vectors** and calculate the **dot product with output of softmax f(x)**.
5. **Add all weighted value vectors** together.

Notice in the figure below that we are doing matrix operations on seq_length x embedding_size matrices. This shows a toy example of two words with an embedding size of 3.



The dot product results in a seq_length x seq_length matrix. Think correlation matrix where the relationships between any two members can be found by their intersections. In this case, the members are words and their intersections are attention scores.

Multiplying value matrix by attention matrix results once again in a seq_length x embedding_size matrix. This matrix holds the **contextual information** for each embedding.

What does it mean scaled?

Scaled Dot-Product Attention

Introduced by Vaswani et al. in Attention Is All You Need

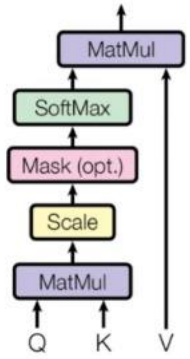
Scaled dot-product attention is an attention mechanism where the dot products are scaled down by $\sqrt{d_k}$. Formally we have a query Q, a key K and a value V and calculate the attention as:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

If we assume that q and k are d_k-dimensional vectors whose components are independent random variables with mean 0 and variance 1, then their dot product, $q \cdot k = \sum_{i=1}^{d_k} u_i v_i$, has mean 0 and variance d_k. Since we would prefer these values to have variance 1, we divide by $\sqrt{d_k}$.

Source: Attention Is All You Need

Read Paper | See Code



Multi-Head Attention (**Multiple Attention Heads)** – simultaneous focus on different subsets of input data => model learns multiple contextually rich representations of data in parallel to capture various aspects of input seq – there are **several attention heads**, each **with its own queries, keys, and values**. In other words, the Attention module **repeats its computations multiple times in parallel**. Each of these is called an **Attention Head**. The Attention module splits its Query, Key, and Value parameters N-ways and passes each split independently through a separate Head. All of these similar Attention calculations are then **combined together** to produce the **final Attention score**. This is called Multi-head attention and gives the Transformer greater power to encode multiple relationships and nuances for each word. Each head processes the input **independently** and the resulting representations are **combined through concatenation and linear transformation** (to return to the correct dimensions).

The query, key, and value vectors are divided by the number of heads, and each segment is passed through a different head. This results in eight vectors which are then concatenated together and transformed by multiplying with another weight matrix so that the resultant vector is the size of the input embedding vector.

https://sebastianraschka.com/blog/2023/self-attention-from-scratch.html (Raschka)

https://towardsdatascience.com/natural-language-processing-the-age-of-transformers-a36c0265937d (second source)

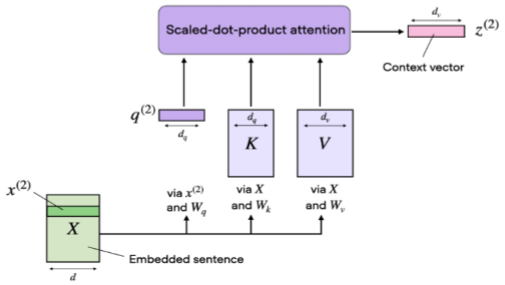
Attention came from an effort to improve RNNs when the input and output sequences do not match token per token exactly (e.g. word order in the same sentence in two different language). But eventually attention superseded the need for RNNs. Self-attention mechanism enables the model to **weigh the importance of different elements** in input sequence and dynamically adjust their influence on the output.

Most popular and most widely used attention mechanism - scaled-dot product (self-attention).

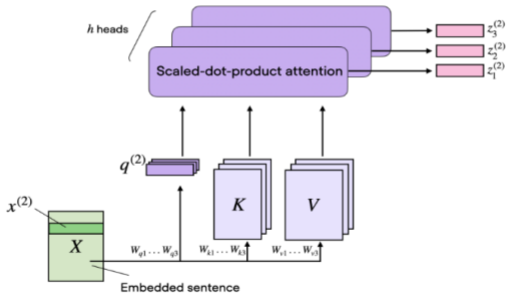
Multi-Head Attention

In the very first figure, at the top of this article, we saw that transformers use a module called *multi-head attention*. How does that relate to the self-attention mechanism (scaled-dot product attention) we walked through above?

In the scaled dot-product attention, the input sequence was transformed using three matrices representing the query, key, and value. These three matrices can be considered as a single attention head in the context of multi-head attention. The figure below summarizes this single attention head we covered previously:



As its name implies, multi-head attention involves multiple such heads, each consisting of query, key, and value matrices. This concept is similar to the use of multiple kernels in convolutional neural networks.



To illustrate this in code, suppose we have 3 attention heads, so we now extend the $d' \times d$ dimensional weight matrices so $3 \times d' \times d$:

Cross attention = transformer's encoder-decoder attention

In self-attention, we work with the same input sequence. In cross-attention, we **combine two different input sequences**. In the case of the original transformer architecture below, that's the sequence returned by the encoder module on the left and the input sequence being processed by the decoder part on the right. Note that in cross-attention, the two input sequences x1 and x2 can have **different numbers of elements, but their embedding dimensions must match**.

Encoder-Decoder Attention - encoder processes input seq and generates a context vector, while decoder generates output seq based on this context vector. **Encoder's output** = keys and values, while **decoder's hidden states** = queries => **Decoder attends to** different parts of **encoded input seq**, promoting greater understanding of input relationships and generating more accurate output sequences (improved translation and seq generation).

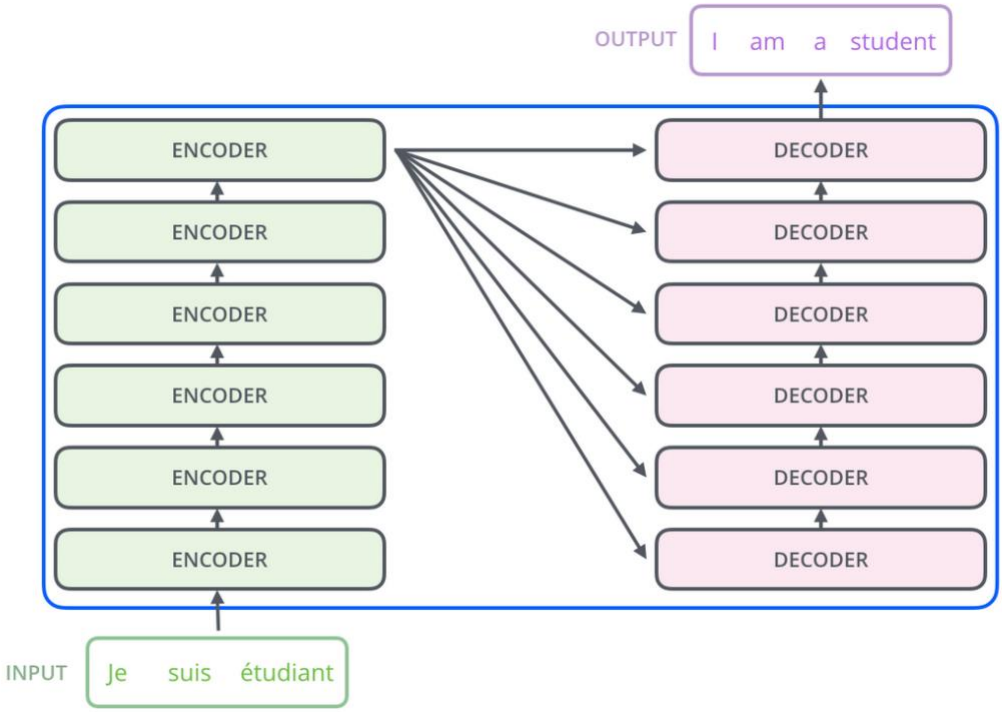
Part 2. Second source summarized

- Encoder self-attention - the input sequence pays attention (attends) to itself. Scaled dot-product of three **vectors (K, V, and Q) coming from the same sequence**. Purpose - to **learn relationships between different words in the sentence**.

- Decoder self-attention - target sequence attends to itself. Same definition as above.
- **Transformer's Encoder-Decoder Attention** in Decoder - target sequence attends to input sequence. Scaled *dot product of K (keys), V (values), and Q (queries) vectors* + *softmax* to decide which values get most attention. Encoder has multiple layers with hidden states h_i to label the final hidden state of the last Encoder layer for each w_i. Decoder also has multiple layers (num equal to that of the Encoder). All of the *Encoder hidden states h_i are fed as inputs to each of the six layers of the Decoder* => **Transformer's Encoder-Decoder Attention**.

Queries, keys, and values – if you are shopping to cook a dish: each ingridient is a query, the labels on the shelves are keys, the items that you take from the shelf are values.

"**Attention Mask**" - array of 1s and 0s indicating which tokens are padding and which aren't. Tells BERT's "self-attention" mechanism not to use the PAD tokens to interpret the sentence.



Google Releases New Infinite Context Method

Code: https://github.com/dingo-actual/infini-transformer

What's New

Can LLMs Handle Unlimited Context? Google researchers introduced a new concept called Infini-attention in their latest paper, enabling LLMs to process inputs of any length.

Comparison with Traditional Transformers: Typical transformers reset their attention memory after each context window to manage new data, losing previous context. For example, in a 500K token document split into 100K token windows, each segment starts fresh without memory from the others.

Infini-attention's Approach: Infini-attention retains and compresses the attention memory from all previous segments. This means in the same 500K document, each 100K window maintains access to the full document's context.

The model compresses and reuses key-value states across all segments, allowing it to pull relevant information from any part of the document.

How Infini-attention Works:

- Utilizes standard local attention mechanisms found in transformers.
- Integrates a global attention mechanism through a compression technique.
- Merges both local and global attention to manage extended contexts efficiently.
- In other words, the method effectively gives each window a view of the entire document, achieving what's termed as "infinite context."

Key Performance Metrics:

- **1B Model:** Effectively manages sequences up to 1 million tokens.
- **8B Model:** Achieves state-of-the-art results in tasks like summarizing books up to 500K tokens in length.

Key Highlights:

- **Memory Efficiency:** Constant memory footprint regardless of sequence length.
- **Computational Efficiency:** Reduces computational overhead compared to standard mechanisms.
- **Scalability:** Adapts to very long sequences without the need for retraining from scratch.

Why This Matters

Elvis Saravia: *"Given how important long-context LLMs are becoming having an effective memory system could unlock powerful reasoning, planning, continual adaption, and capabilities not seen before in LLMs. Great paper!"*

Community Feedback

swyx: *"with Griffin and Infini-attention, it increasingly feels like Google leapfrogged Together and RWKV in the race for scaling up linear attention, and they shared a watercooler conversation with Anthropic or something"*

Raúl Avilés Poblador: *"Sounds good on paper but scaling this to near-infinity would also mean inference would require crazy hardware resources, isn't it?"*

Input Representation

- **Tokenization** = breaking down input seq into smaller units – tokens (words, subwords or chars (punctuation)).
- **Embeddings** map tokens to high-dimensional vectors capturing semantic and syntactic info about words.
- **Positional embeddings** - capture position of each token within input seq (as Transformer processes input simultaneously, not sequentially as RNNs and LSTMs. Pos encodings calculated w/sine and cosine f(x) are added to word embeddings => combined representation capturing both meaning and position.

Training Time

Transformers have **faster training time** vs. LSTMs due to **better parallelization** during training because the self-attention mechanism **doesn't rely on sequential computations** like in LSTMs. + parallelization enables **better GPU utilization** (speeds up training).

Efficiency
Transformers are **more efficient** than LSTMs for handling **long-range dependencies in sequences** - self-attention mechanism allows direct access to any part of the input seq while LSTMs process seq step-by-step => transformers can better model complex relationships between distant tokens.
At least in one study for speech recognition transformers had a higher tendency to overfit vs. LSTM (generalization issue).

Applications or Seq2Seq models: machine translation, text summarization, conversation modeling (chatbots), time series, sentiment analysis, text classification, Q&A.

Time series forecasting - LSTMs have long been a popular choice - capture both short and long-term dependencies in data. Transformers are effective thanks to their attention mechanism - focuses on important parts of input sequence.

Transformer-XL (w/extra-long context) overcomes issues w/long-range dependencies (translation and language modeling) by implementing a recursive mechanism that connects different segments => allows to store and access info from prev segments.

Vision Transformers (ViT) treat an image as a sequence of patches, used for image classification, challenges the prevalent use of CNNs, achieves SOTA results on ImageNet.

FAQ

What are the key differences between LSTM and Transformer?
LSTM is a type of RNN that addresses the vanishing gradient problem, captures somewhat long dependencies in sequences. Transformers utilize self-attention mechanisms to process sequence inputs, handling long-range dependencies more efficiently - the most notable difference is the absence of RNN cells in Transformer architecture, which allows it to process inputs in parallel, resulting in faster computation.

How do LSTM and Transformer models compare in terms of speed and performance?
Transformers outperform LSTMs in some tasks, particularly those involving longer input sequences. Also, Transformers can process sequence data in parallel - faster training and inference times.

Are Transformers more suitable for certain tasks compared to LSTMs?
Transformers excel in long-range dependencies handling and parallel processing, such as machine translation, text summarization, and natural language understanding. LSTMs can still perform well on shorter sequences in tasks not requiring the full power and complexity of a Transformer model, e.g. sentiment or time-series prediction.

Why might one choose Transformer over LSTM in specific applications?
Choosing a Transformer model over LSTM could be motivated by several factors, such as:

- **Parallel processing capabilities**: Transformers can process sequence data concurrently, which is beneficial for computational efficiency and shorter training times.
- **Long-range dependency handling**: Transformers excel at understanding dependencies across larger sequences, making them ideal for complex tasks like machine translation or text summarization.

- **Scalability**: Due to their parallel processing capability, Transformers can handle larger input sequences more effectively than LSTMs.

Can LSTM and Transformer models be combined effectively?
Yes - **hybrid architectures**. Example – using **LSTM** layer to capture **local dependencies within a Transformer.**

How do the architectures of LSTM and Transformer models differ?
LSTM models consist of **RNN cells** designed to store and manipulate info across time steps. Transformers contain a stack of encoder and decoder layers, each consisting of self-attention and feed-forward neural network components. This architecture allows Transformers to process sequence data without the need for recurrent connections or cells, enabling **parallel processing** and **more efficient long-range dependency handling**.

10. WORD EMBEDDINGS, CONTEXTUAL EMBEDDINGS, ENCODER-DECODER MODEL, ATTENTION, BEAM SEARCH (COPY FROM DJ)

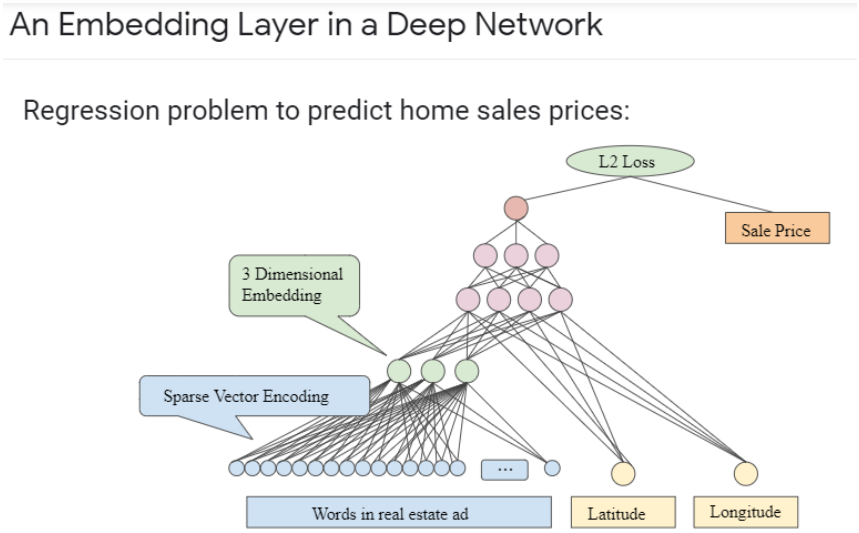
WORD EMBEDDINGS

Embeddings in general (video which is retold on subsequent webpages - excellent): https://developers.google.com/machine-learning/crash-course/embeddings/video-lecture

Embeddings translate high-dimensional vectors for items (e.g. movies, text,...) into **low-dimensional real vectors** in a way that **[semantically] similar items are close to each other**. Joint embeddings of diverse data types (e.g. text, images, audio, ...) define a similarity between them. An embedding can be learned and reused across models. Embeddings **make it easier** to do machine learning on large inputs like **sparse vector representation** of text (e.g. when vocab is vectorized by assigning integers to each word, represent sentences as lists of ints).

Learned Embeddings (DL)

No separate training; **embedding layer = hidden layer with one unit per dimension**. *Intuitively* the hidden units discover how to organize the items in the d-dimensional space in a way to best optimize the final objective (using backpropagation).



Num dimensions - *hyperparameter tuning* problem. Too few - insufficient representation, too many - overfitting and slower training. Rule of thumb: num dimensions = **fourth root** (num possible values)

Word2Vec: https://www.tensorflow.org/tutorials/text/word2vec

Two methods for learning word representations:

- **Continuous Bag-of-Words** Model - *predicts the middle word* based on **surrounding context** words (context = a few words before and after; **order** of words in the context is **not important** => BoW).
- **Continuous Skip-gram** Model which predict words in a range before and after current word in the same sentence. **Skip-gram** = n-gram with tokens skipped. Context can be represented as **skip-gram pairs** (target_word, context_word) where context_word = any word within a range (2, 3, etc.) from

target_word. **Negative sampling** - the same skip-gram pairs, but the target_word is not from the context (used for training the model)

GloVe (Global Vectors for Word Representation): trained on **non-zero entries** of a global **word-word co-occurrence matrix** (tabulates how frequently words co-occur with one another in the corpus). Model combines **global matrix factorization & local context window** methods, leverages statistical information by training only on nonzero elements in a word-word cooccurrence matrix (not the entire sparse matrix or individual context windows)

CONTEXTUAL EMBEDDINGS

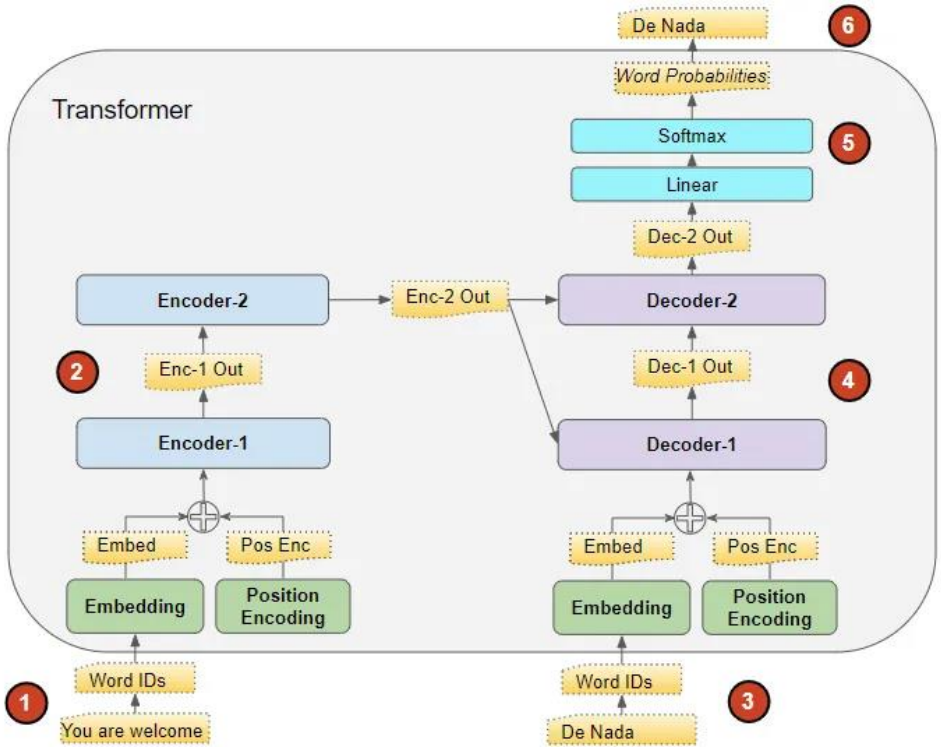
ELMO (Embeddings from LMs) = computes contextualized word representations/embeddings using **character-based word representations & bidirectional LSTMs**, as described in the paper "Deep contextualized word representations", dynamically **adjusting word embedding** based on the context (solving the problem of polysemous words). ELMo comes up with the contextualized embedding through *grouping together the hidden states* (and initial embedding) in a certain way (**concatenation** followed by **weighted summation**)

ULM-FiT introduced an **LM** and a process to effectively **fine-tune** that language model for various tasks. LM trained on a special **LSTM** architecture with **dropouts** at every possible level.

BERT embeddings = output from some of the 12 hidden layers. Examples from https://medium.com/@dhartidhami/understanding-bert-word-embeddings-7dc4d2ea54ca – **concatenate the output from the last 4 layers** (3072 dimensions) or **sum up** the same (768 dim). These are contextualized embeddings: bank will have different vectors depending on its meaning. To get sentence embeddings - **average the second to last hidden layer** of each token producing a single 768 length vector.

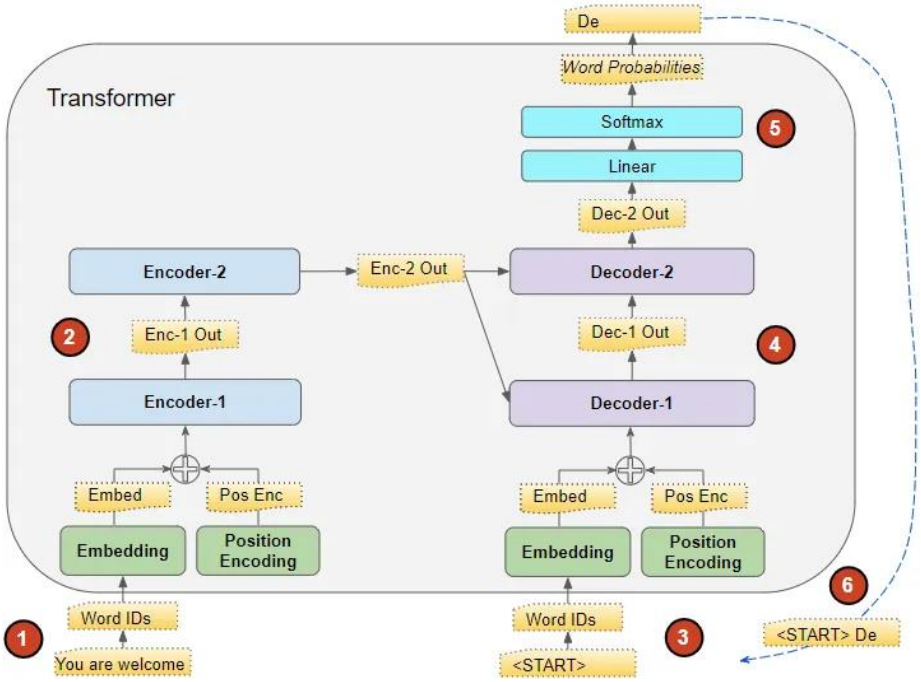
OpenAI **GPT-3** (Generative Pre-trained Transformer) trained on a **massive corpus** from Common Crawl, Wikipedia, etc. with more than **175 B parameters** = **generative language model** predicting the probability of the next word; given an input **prompt**, **generates text** that has the potential to be practically **indistinguishable from human-written text**.

Training transformers



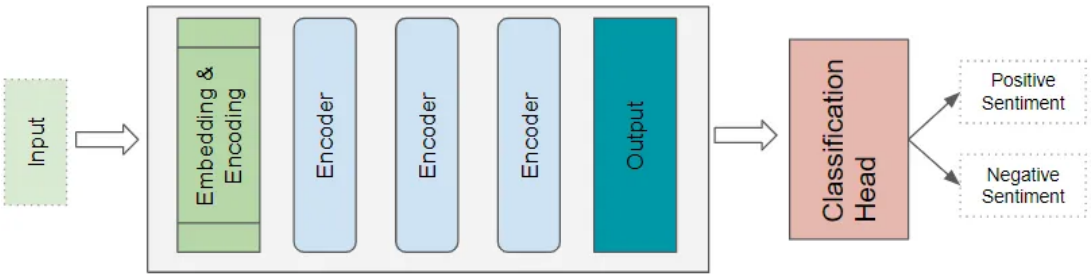
- ✓ The input sequence is converted into Embeddings (with Position Encoding) and fed to the Encoder.
- ✓ The stack of Encoders processes this and produces an encoded representation of the input sequence.
- ✓ The target sequence is prepended with a start-of-sentence token, converted into Embeddings (with Position Encoding), and fed to the Decoder.
- ✓ The stack of Decoders processes this along with the Encoder stack's encoded representation to produce an encoded representation of the target sequence.
- ✓ The Output layer converts it into word probabilities and the final output sequence.
- ✓ The Transformer's Loss function compares this output sequence with the target sequence from the training data. This loss is used to generate gradients to train the Transformer during back-propagation.

Inference by transformers



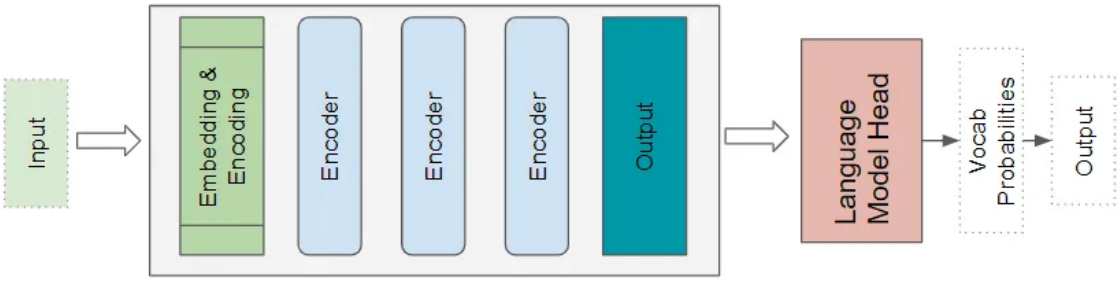
- ✓ The input sequence is converted into Embeddings (with Position Encoding) and fed to the Encoder.
- ✓ The stack of Encoders processes this and produces an encoded representation of the input sequence.
- ✓ Instead of the target sequence, we use an empty sequence with only a start-of-sentence token. This is converted into Embeddings (with Position Encoding) and fed to the Decoder.
- ✓ The stack of Decoders processes this along with the Encoder stack's encoded representation to produce an encoded representation of the target sequence.
- ✓ The Output layer converts it into word probabilities and produces an output sequence.
- ✓ We take the last word of the output sequence as the predicted word. That word is now filled into the second position of our Decoder input sequence, which now contains a start-of-sentence token and the first word.
- ✓ Go back to step #3. As before, feed the new Decoder sequence into the model. Then take the second word of the output and append it to the Decoder sequence. Repeat this until it predicts an end-of-sentence token. Note that since the Encoder sequence does not change for each iteration, we do not have to repeat the furst two steps each time.

Classification architecture



Transformer Language Model architecture

Takes the Transformer's output and generates a probability for every word in the vocabulary. The highest probability word becomes the predicted output for the next word in the sentence.



GPT vs. BERT

Generative Pretrained Transformers:

- **Autoregressive (causal) language model** designed to handle sequential data with the attention mechanism, generative model that generates text given some input (*predicts the next word* as a softmax distrib. over the entire vocab.). Unsupervised pretraining + supervised fine-tuning for specific tasks + RLFH for dialog apps.
- **Unidirectional** text processing – consider **left context** to predict the next word. Causal LM

Bidirectional Encoder Representations from Transformers:

- **Bidirectional** text processing - takes into account **both the words that come before and after the given word** when understanding context: more accurate representations for each word.
- Pre-training: **masked language model** (15% of tokens are masked - predicting the masked tokens based on their context) and **next sentence prediction (**predicting whether one sentence logically follows another sentence).
- BERT isn't generative - used for understanding the meaning of text: sentiment analysis, NER, Q&A.

BERT

(see full article in a separate file)

BERT (Bidirectional Encoder Representations from Transformers) - **state-of-the-art results** in NLP tasks, e.g. Q&A, inference, etc. **Bidirectionally trained LM** - deeper sense of language context and flow + **attention** mechanism.

Transformer = *attention mechanism that learns both left and right contextual relations between words (or sub-words)* = encoder (reads text input) + decoder (predicts). BERT generates LM - **only** the **encoder** is necessary. Encoder **reads the entire sequence of words at once**, and not sequentially *as RNNs* (= **bidirectional** OR **non-directional)**.

Attention helps to draw connections between any parts of the sequence, so long-range dependencies are not a problem anymore. With transformers, **long-range dependencies have the same likelihood of being taken into account** as any other short-range dependencies

The **input** sequ. of tokens is **embedded** into vectors and **processed** by neural network - **output** = sequence of **vectors of size H**, each vector **corresponds to an input token** *with the same index*.

BERT developers created two models:

- ✓ BASE: # transformer blocks (L): 12 layers, 768 hidden units or size (H), Attention heads (A): 12
- ✓ LARGE: # transformer blocks (L): 24 layers, 1024 hidden units or size (H), Attention heads (A): 16
- ✓ For comparison: GPT-3 has 175 B params, 96 layers

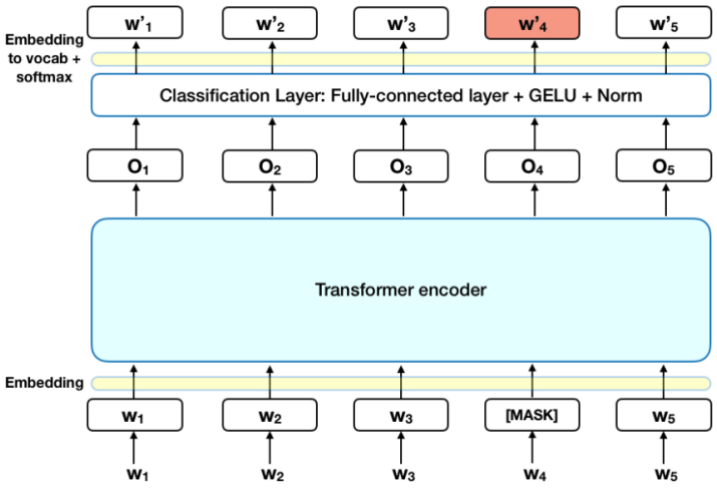
Multiple Attention Heads: the Attention module **repeats its computations multiple times in parallel**. Each of these is called an **Attention Head**. The Attention module splits its Query, Key, and Value parameters N-ways and passes each split independently through a separate Head. All of these similar Attention calculations are then **combined together** to produce a **final Attention score**. This is called Multi-head attention and gives the Transformer greater power to encode multiple relationships and nuances for each word.

LM training challenge: define a prediction goal (e.g. next word = limited context). **BERT - two training strategies**:

- ## Masked LM (MLM)

15% of input words are replaced with **[MASK]** token. The model **predicts** the value of **masked words** based on the context of non-masked words in the sequence and ignores predictions of non-masked words => it **converges slower** than directional models, but this is offset by its **increased context awareness**. Usage:

- Add **classification layer** on top of encoder output.
- **Multiply output vectors by embedding matrix** transforming them into vocab. dimension.
- Calculate **probability of each word** in vocab. with **softmax**.

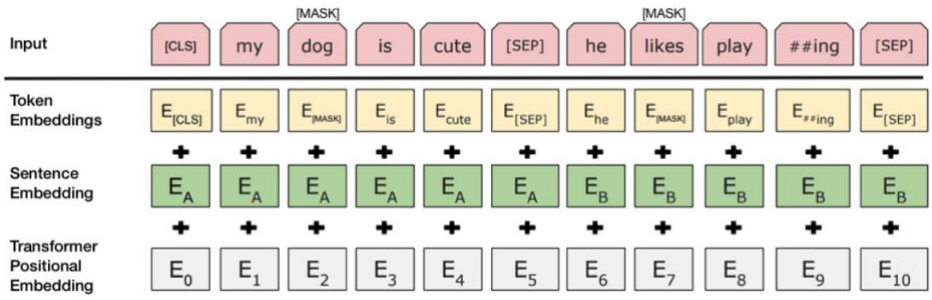


- ## Next Sentence Prediction (NSP)

Input: **pairs of sentences**; the model learns to predict if the second sent is **subsequent sent** in the original document. In **50%** of pairs a **random** (disconnected) sentence is chosen from the corpus instead of the subsequent sent.

Preprocessing of input data:

- **[CLS]** token inserted before 1st sent + **[SEP]** token inserted at the end of each sent.
- **Sent embedding** for sent A or sent B added to each token (sent embeddings ~ token embeddings with vocab. = 2).
- **Positional embedding** added to each token (indicates token's position in the sequence). See the Transformer paper for implementation of positional embedding.



Prediction if 2nd sent is connected:

- Entire **input** sequence goes **through the Transformer** model.
- **Output of [CLS] token** transformed into a **2×1 shaped vector** using simple classification layer (learned matrices of weights and biases).
- Calculate **probability of IsNextSequence** with **softmax**.

MLM and NSP are trained together to **minimize the combined loss function**.

Fine-tuning BERT

Add small layer to the core model:

- **Classification** (e.g. *sentiment analysis*) - similarly to Next Sentence classification = add a **classification layer** on top of Transformer output for [CLS].

- **Q&A** tasks (e.g. SQuAD v1.1) - model receives a question re a text sequence and marks the answer in the sequence. Training Q&A model - **two extra vectors** marking the **beginning and end of answer**.

- In **Named Entity Recognition** (NER) - model receives a text sequence and **marks various types of entities** (Person, Organization, Date, etc). Training NER model - feeding **output vector** of each token into classification layer **predicting NER label**.

Most hyper-parameters stay the same; paper gives specific guidance on partial **hyper-parameter tuning ((**Section 3.5). Paper mentions the use of BERT to achieve state-of-the-art results in NLP (Section 4).

Conclusion

1. **BERT** = **breakthrough** = allows **fast fine-tuning**. Read *full article* and ancillary articles referenced in it for a deeper dive. Another reference: *BERT source code and models for 103 languages* released by the research team.

- **BERT_large** = 345 M parameters demonstrably outperforms on small-scale tasks **BERT_base** = 110 M parameters.

- ***More training steps == higher accuracy***.

- *BERT's bidirectional (MLM)* ***converges slower*** *than left-to-right approaches* (only 15% of words are predicted in each batch), but still outperforms left-to-right training after a small number of pre-training steps.

TO CLARIFY - **Word masking**: BERT predicts randomly picked 15% of input tokens. These tokens are pre-processed: 80% replaced with [MASK], 10% with random word, and 10% use original word.

Example of fine-tuning BERT: https://www.tensorflow.org/tutorials/text/classify_text_with_bert , https://www.tensorflow.org/official_models/fine_tuning_bert

From this article on BERT: https://mccormickml.com/2019/07/22/BERT-fine-tuning/

GPT-3

Unsupervised language model. **Decoder** architecture. The model was trained using generative pre-training – it **autocompletes the sequence by iteratively predicting the most probable next word**. The computed representation for each token depends only on the left context = causal or autoregressive attention.

Trained on Common Crawl, Wikipedia, books – almost all available data. Requires **prompting -** a new kind of programming, where the prompt is now a "program" which **programs GPT-3 to do new things**. Can perform new tasks for which it was not trained. **Few shot learning** – GPT-3 showed that unsupervised language models trained with enough data can multitask, without fine-tuning, to the level of fine-tuned SOTA models by seeing just a few examples of the new tasks.

GPT uses a **one model for all downstream tasks** (no fine-tuning per task). It uses a paradigm which allows zero, one, or a few examples to be prefixed to the input of the model. In the few-shot scenario, the model is presented with a task description, a few dozen examples, and a prompt. GPT-3 then **takes all this information as the context** and starts to predict output token by token.

What can it do:

- ✓ Text Generation
- ✓ Translation
- ✓ Text Classification
- ✓ Sentiment Extraction
- ✓ Reading Comprehension
- ✓ NER
- ✓ Q&A

ChatGPT

Trained using a 3-step approach:

1. **Supervised Fine Tune** (SFT)

The goal of this step is to fine tune a pretrained GPT3 model using human generated responses. Hiring a lot of **humans to write high quality prompts and responses** (prompts also came from the API, but reponses were all human), then trained a pretrained GPT3.5 model on the **[concatenated] prompts and responses** - improving the network by forcing the model to keep fitting on input and output data. Inverse back-propagation adjusts the parameters of the model using these new data.

2. **Train a Reward Model** (RM)

- ✓ Take the **SFT** model from phase 1, remove the last layer called unembedding, **add a linear layer** (regressor)
- ✓ Throw **a prompt** multiple times (4–9 times) at it to generate **multiple responses**.
- ✓ Create **pairs** of responses from a given prompt (**one prompt – two reponses**).
- ✓ Ask a **human to rank individual responses** within each pair (indicate which response is better).
- ✓ Higher reward - human is more likely to prefer that response; a lower reward indicates the opposite.
- ✓ Update the model to align the generation with human-preference by doing the following:
  - ◆ Concatenate the prompt with each of the responses in the pair - one winning prompt-response and one losing prompt-response pair.
  - ◆ Pass the **winning prompt-response** to the reward model and generate a **winning reward**.

- ◆ Pass the l**osing prompt-response** to the same reward model and generate a **losing reward**.
- ◆ Take the difference between the two rewards and update the model so as to **maximize this difference**.
- ◆ The loss is a difference in log-likelihood of a high reward and low reward response. Intuitively, the **reward model is learning to maximize the difference between a winning and a losing response**.
- ◆ Once the model is trained, one can simply concatenate the prompt with a response and get a **scalar reward** that indicates the degree to which a human will prefer that response for the given prompt. **That's all this outputs – no text is generated!**

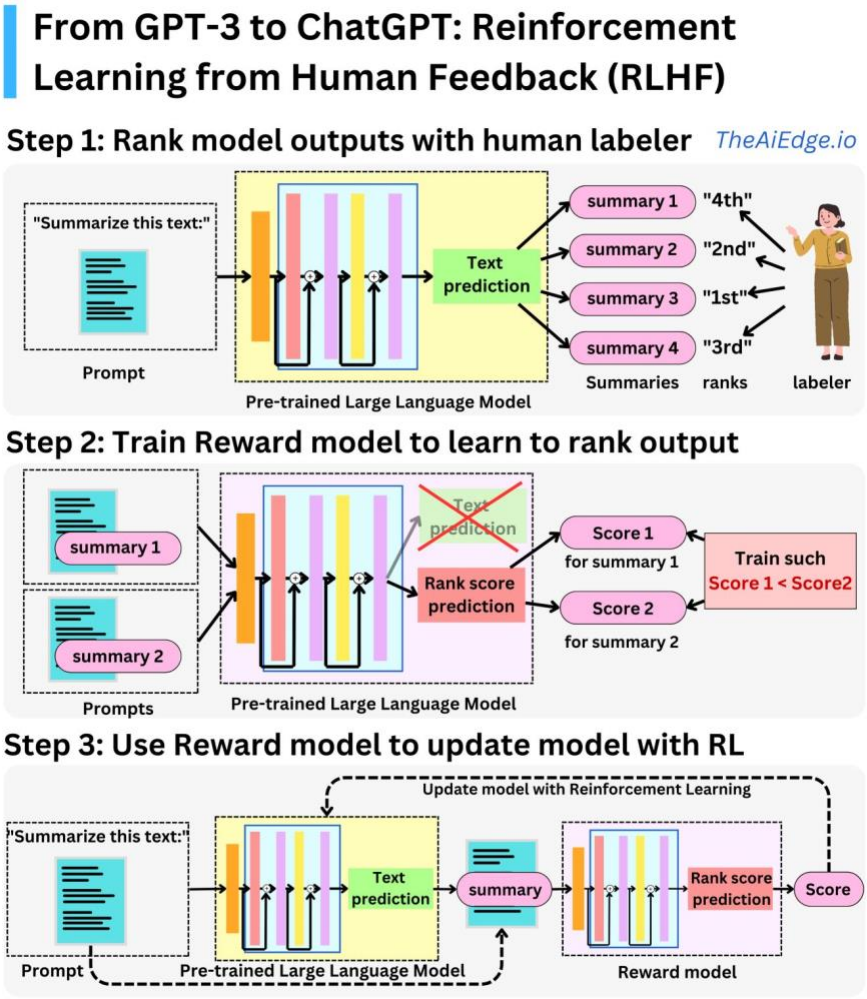
3. **Train a Reinforcement Learning Model (RL)**

Now we need to utilizes the prev. 2 models to produce another model capable of generating text per human preferences. Cost f(x) – maximizing expectation of reward from prompt-response pairs

- ✓ **31k prompts w/out reponses** were used for training.
- ✓ Throw each prompt at the model.
- ✓ The model generates a response.
- ✓ Concatenate the **prompt-response and pass it to the reward model** to get a reward score for the pair.
- ✓ Plug the values for reward and probabilitites in the **gradient equation of the cost f(x)**.
- ✓ Use this gradient to **update the model parameters**.

The following adjustments had to be made:

- ✓ Minimize divergence from SFT model (otherwise, unstable results - model generates gibberish)
- ✓ Maximize likelihood of observing sequences seen during pretraining GPT-3 (otherwise GPT-3 outperformed the new model on some tasks)



Statistical Model (ChatGPT) + Symbolical Logic (Wolfram Alpha)

More here: https://writings.stephenwolfram.com/2023/03/chatgpt-gets-its-wolfram-superpowers/

Wolfram Alpha (WA) - **computational knowledge engine** developed by Wolfram Research: answers factual queries by computing answers from externally sourced data.
Users submit requests via a text field (NLU component). WA computes answers and visualizations from a **KB of curated, structured data** that come from other sites and books (*the CIA's The World Factbook, the United States Geological Survey, a Cornell University Library publication called All About Birds, Chambers Biographical Dictionary, Dow Jones, the Catalogue of Life, CrunchBase, Best Buy, and the FAA*). It displays its "input interpretation" of such a question. **Mathematical symbolism** is used to parse the input.
Released in 2009. Written in the **Wolfram Language**, implemented in proprietary Mathematica. At different times it was used in Microsoft Bing, DuckDuckGo, Apple's Siri, Amazon Alexa. Spanish version was launched in 2022.

WA plugin improves reliability of ChatGPT by providing **factual information** rather than imaginative content

WA = ChatGPT plugin acting as a brain implant for ChatGPT; gives ChatGPT the ability to deliver accurate, curated knowledge and data—as well as correct, nontrivial computations. Has two entry points: Wolfram|Alpha (accepts natural language input) and Wolfram Language (requires precise code). Wolfram Language is more powerful and flexible.

Under the hood ChatGPT interacts w/WA plugin by rewriting the query, switching between Wolfram|Alpha and Wolfram Language, using a plugin prompt. WA plugin process Wolfram|Alpha input / evaluates Wolfram Language code, sends it to a special LLM API endpoint. The output can include text, graphics, or math formulas. **ChatGPT interprets the WA output and generates a response for the user**. Users can check the actual code sent to WA plugin and the returned output.

Wolfram Language offers a more structured way of expressing computational ideas, represents both abstract and real-world concepts. The goal of Wolfram Language is to enable users to express themselves computationally, just as mathematical notation allows for expressing mathematical ideas. It **is designed for both humans and computers to read and understand**. Instead of being centered on computer-oriented instructions, it focuses on automating the process of implementing human-oriented concepts. While natural language is useful to some extent, it can be unwieldy for specifying complex ideas.

With ChatGPT, Wolfram Language becomes even more important as it can **build up computational ideas using natural language**, while allowing users to understand the generated code. This collaboration is already working effectively with the Wolfram plugin in ChatGPT.

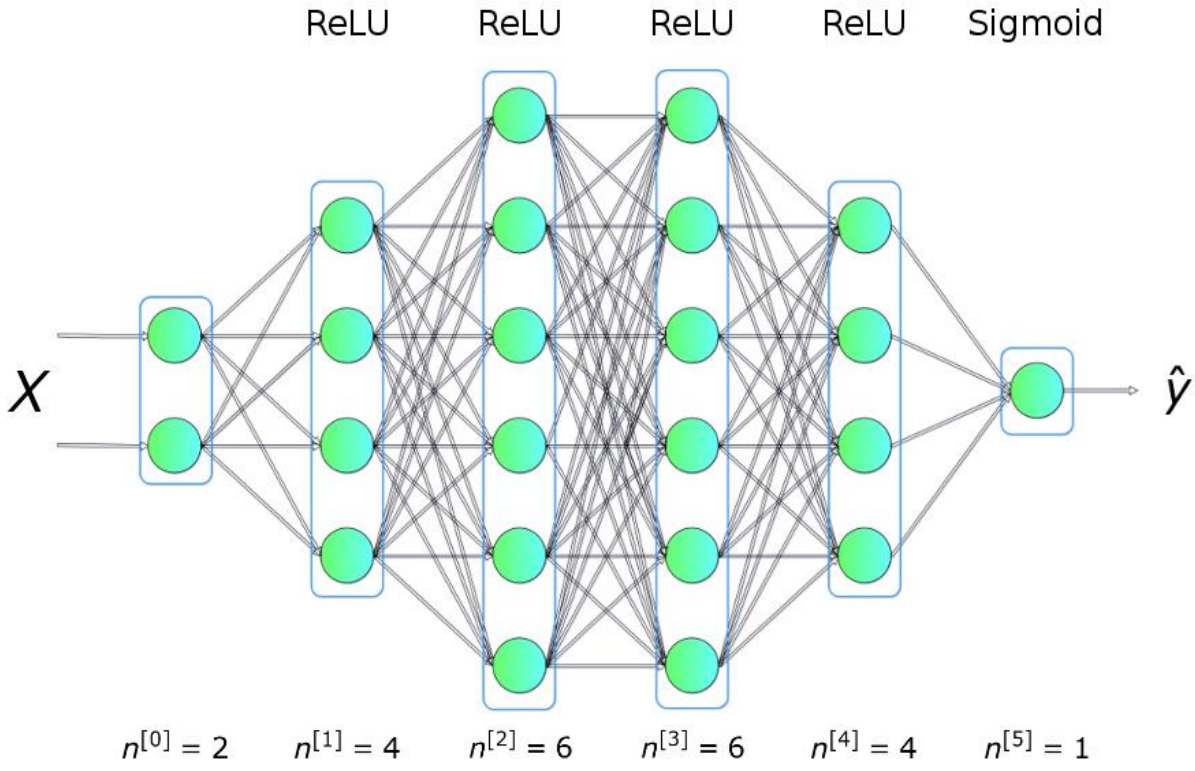
This is a historic moment: for half a century the statistical and symbolic approaches to AI evolved separately. But in ChatGPT + Wolfram they're being brought together (just the beginning).

Sample queries:
GeoDistance[{40.7128, -74.0060}, {34.0522, -118.2437}]
WordCounts["one fish, two fish, red fish, blue fish"] => <| fish →3, one→ 1, etc. |>
WordCounts["the fox jumped over the hare.", 2] => <| ('the', 'hare') → 1, ('the', 'fox') → 1, etc.)

Deep Learning

(Complete copy from "Basics of ML and DL.docx". Delete any time)

Neural network topologies:
Feed-forward network layers (CNN): input l., hidden processing layers, dropout, output l.
Pooling – reducing matrix from an earlier convolutional layer to a smaller matrix – taking the max or average value across the pooled area. Pooling mainly helps in **extracting sharp, low-level features** like edges, points (max pool) and **smooth features** (average pool). It is also done to **reduce variance and computations**.



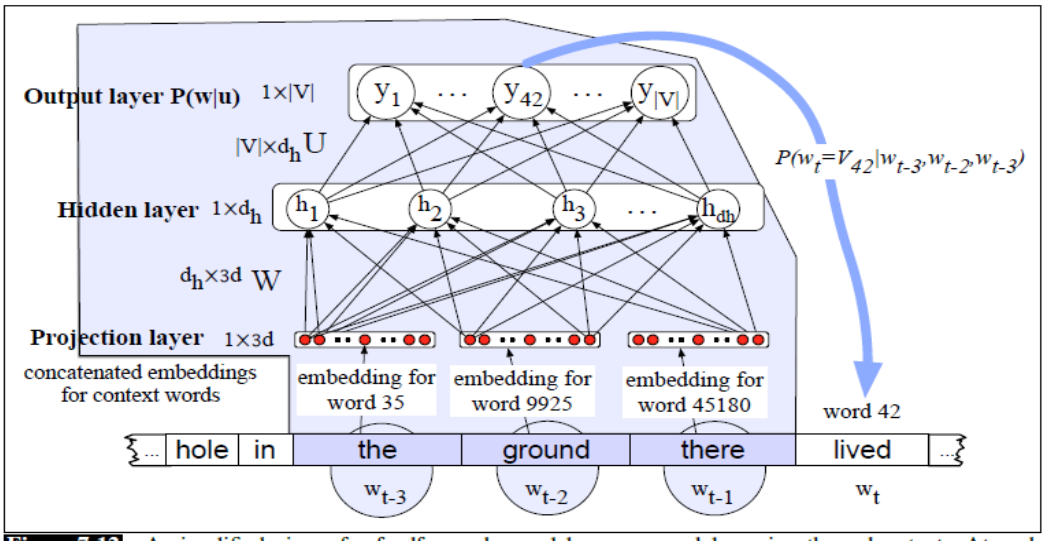


Figure 7.12 A simplified view of a feedforward neural language model moving through a text. At each timestep t the network takes the 3 context words, converts each to a d-dimensional embedding, and concatenates the 3 embeddings together to get the $1 \times Nd$ unit input layer x for the network. These units are multiplied by a weight matrix W and bias vector b and then an activation function to produce a hidden layer h, which is then multiplied by another weight matrix U. (For graphic simplicity we don't show b in this and future pictures.) Finally, a softmax output layer predicts at each node i the probability that the next word w_t will be vocabulary word V_i. (This picture is simplified because it assumes we just look up in an embedding dictionary E the d-dimensional embedding vector for each word, precomputed by an algorithm like word2vec.)

$$z = w_1x_1 + w_2x_2 + w_3x_3 + \ldots + w_nx_n = \mathbf{w}^T \cdot \mathbf{x}$$

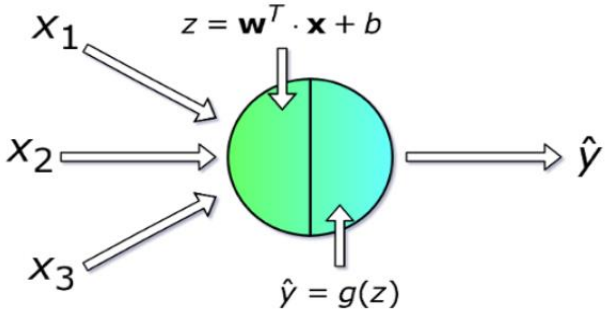
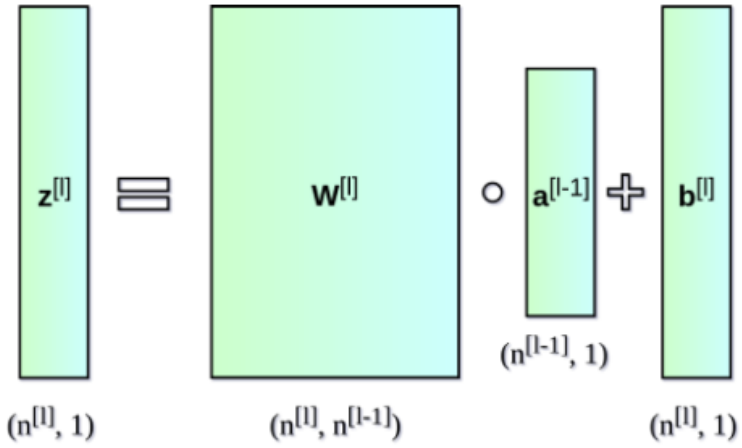


Figure 4. Single neuron

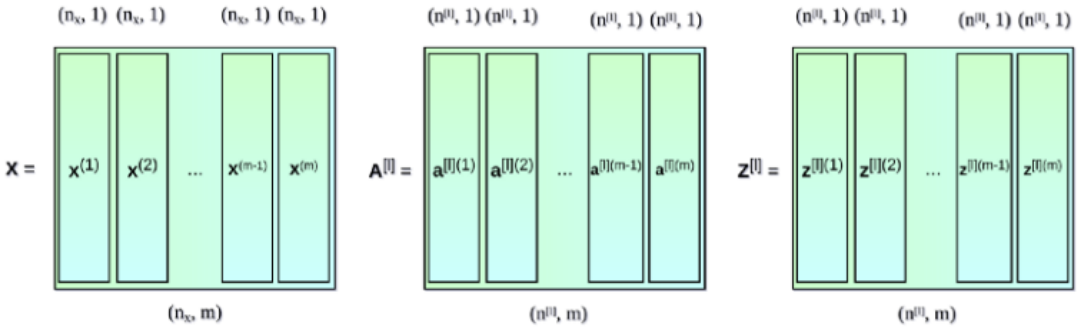
This is how it looks for one of the above layers (x - activation for layer 0, $\mathbf{a^n}$ - activation for layer n):

$$\mathbf{z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{a}^{[l-1]} + \mathbf{b}^{[l]} \qquad \mathbf{a}^{[l]} = g^{[l]}(\mathbf{z}^{[l]})$$



Vectorizing across multiple examples

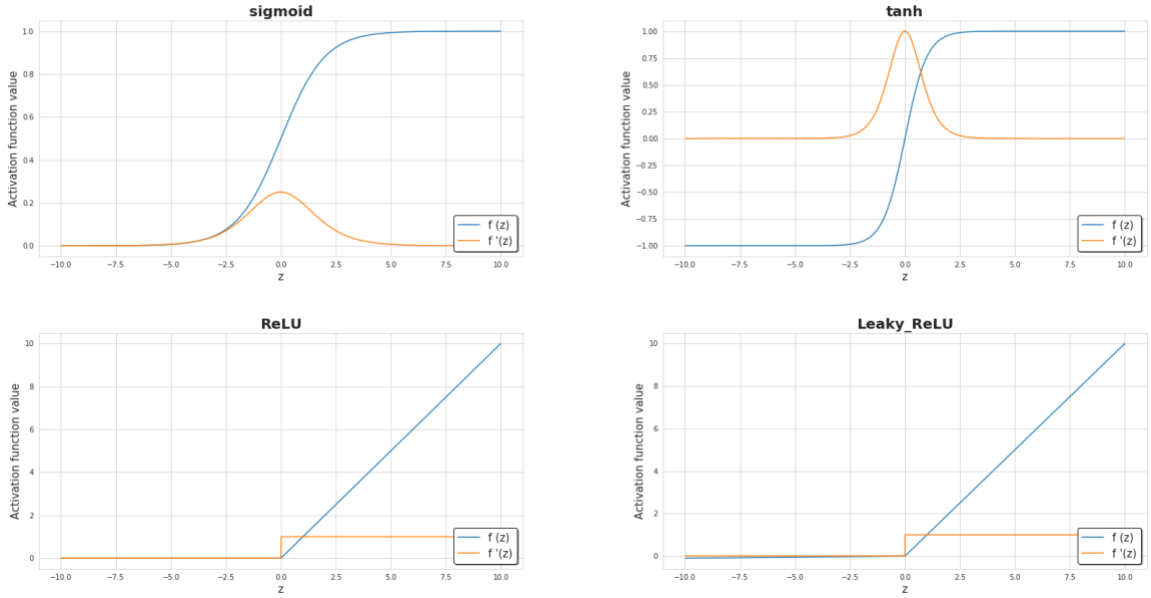
The equation that we have drawn up so far involves only one example. During the learning process of a neural network, you usually work with huge sets of data, up to millions of entries. The next step will therefore be vectorisation across multiple examples. Let's assume that our data set has *m* entries with *nx* features each. First of all, we will put together the vertical vectors **x**, **a**, and **z** of each layer creating the **X**, **A** and **Z** matrices, respectively. Then we rewrite the previously laid-out equation, taking into account the newly created matrices.



$$\mathbf{Z}^{[l]} = \mathbf{W}^{[l]} \cdot \mathbf{A}^{[l-1]} + \mathbf{b}^{[l]} \qquad \mathbf{A}^{[l]} = g^{[l]}(\mathbf{Z}^{[l]})$$

Activation function

Without them, neural network = combination of linear functions = linear function itself => limited expansiveness similar to logistic regression. The **non-linearity element** allows for **greater flexibility** and creation of complex functions during the learning process.



Loss function

Shows how far we are from the 'ideal' solution - with each iteration the value of the loss function decreases and accuracy increases (red&blue plots). Example of cross-entropy:

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L(\hat{y}^{(i)}, y^{(i)})$$

$$L(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y}))$$

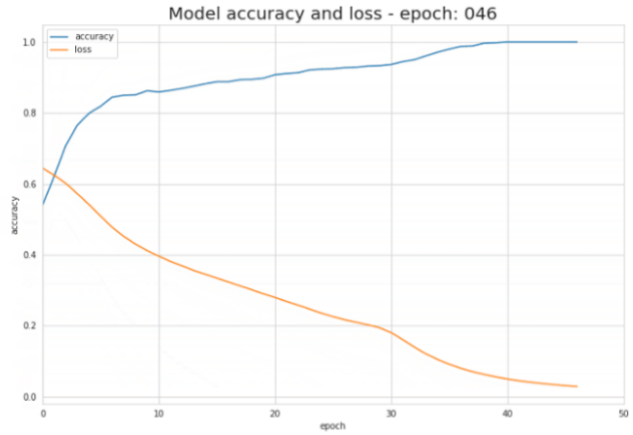


Figure 7. Change of accuracy and loss values during learning process

How do neural networks learn?

The learning process is about **updating** the values of weights matrix **W** and biases vector **b** to find a **loss function minimum** by using gradient descent through partial **derivatives of loss function**. Derivative describe the slope of the function, and thus we know how to manipulate variables in order to **move downhill**.

Alpha = learning rate (important to set it right). **W & b are adjusted to get down the slope in the right direction**.

Backpropagation

Backpropagation is an algorithm that allows us to calculate a very complicated gradient, like the one we need. The parameters of the neural network are adjusted according to the following formulae.

$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \mathbf{dW}^{[l]}$$
$$\mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \mathbf{db}^{[l]}$$

In the equations above, α represents learning rate - a hyperparameter which allows you to control the value of performed adjustment. Choosing a learning rate is crucial — we set it too low, our NN will be learning very slowly, we set it too high and we will not be able to hit the minimum. **dW** and **db** are calculated using the chain rule, partial derivatives of loss function with respect to **W** and **b**. The size of **dW** and **db** are the same as that of **W** and **b** respectively. Figure 9. shows the sequence of operations within the neural network. We see clearly how forward and backward propagation work together to optimize the loss function.

$$\mathbf{dW}^{[l]} = \frac{\partial L}{\partial \mathbf{W}^{[l]}} = \frac{1}{m} \mathbf{dZ}^{[l]} \mathbf{A}^{[l-1]T}$$
$$\mathbf{db}^{[l]} = \frac{\partial L}{\partial \mathbf{b}^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} \mathbf{dZ}^{[l](i)}$$
$$\mathbf{dA}^{[l-1]} = \frac{\partial L}{\partial \mathbf{A}^{[l-1]}} = \mathbf{W}^{[l]T} \mathbf{dZ}^{[l]}$$
$$\mathbf{dZ}^{[l]} = \mathbf{dA}^{[l]} * g'(\mathbf{Z}^{[l]})$$

In partial differentiation, only the variable with respect to which the the function is being differentiated is considered as variable and other variables are considered as constants.

Ex: consider $f = 4x^2 + 3y + z$
partial derivative of f with respect to x is $f' = 8x$

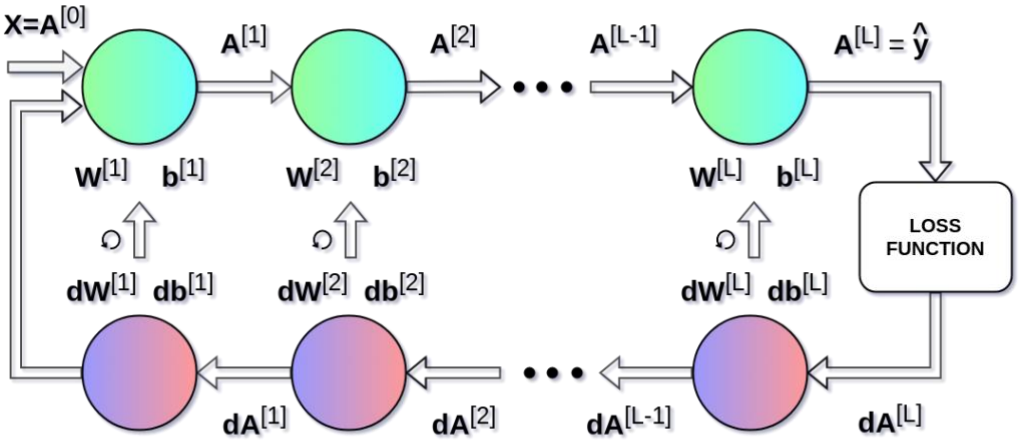
Here you only differentiate x and other variables, viz., y and z, are considered constants, so their derivatives are zero.

In ordinary differentiation, all the variables are differentiated with respect to the considered variable.

Ex: consider the above function
It's differentiation with respect to x is $f' = 8x + 3\frac{dy}{dx} + \frac{dz}{dx}$

Note:
Backpropagation involves taking the **error rate** (e.g. MSE) between true values and computed values of a forward propagation and iteratively **feeding it backward** through the neural network layers **to adjusts the weights and biases until the error rate reaches a minimum** so that a NN can use randomly allocated weights and biases to produce the correct output.
After each forward pass, the backward pass adjusts weights and biases **to minimize the cost function C**. This is done by **computing the gradient of C** with respect to each weight which tells you **how much the parameter x needs to change** to minimize C (see gradient) and whether **to increase** (negative gradient) or **decrease** (positive gradient) the weights.

FORWARD PROPAGATION



BACKWARD PROPAGATION

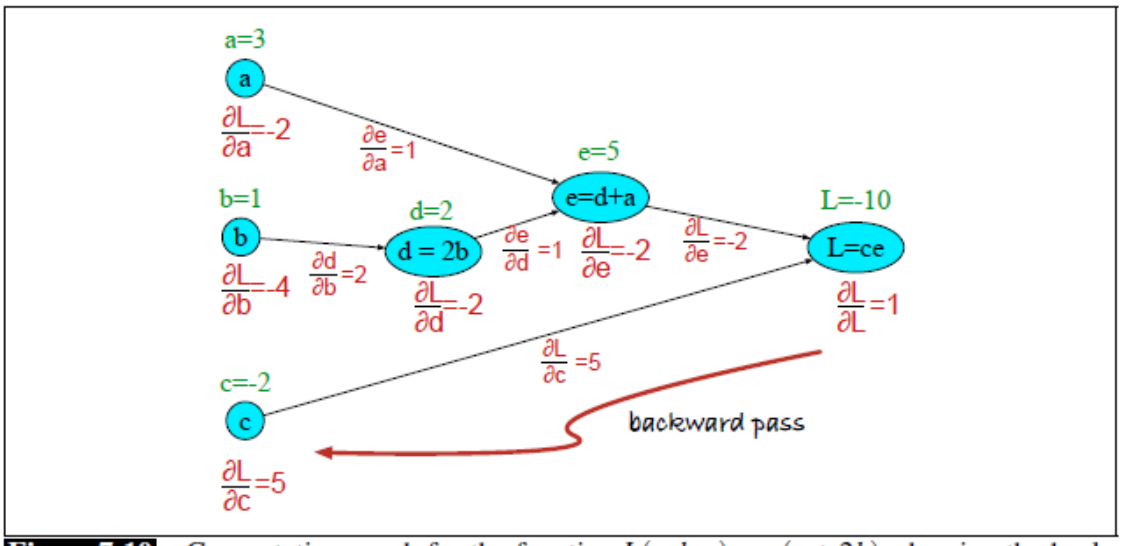


Figure 7.10 Computation graph for the function $L(a,b,c) = c(a+2b)$, showing the backward pass computation of $\frac{\partial L}{\partial a}$, $\frac{\partial L}{\partial b}$, and $\frac{\partial L}{\partial c}$.

RNN - information cycles through a loop, network **remembers recent previous outputs**. RNNs have two common issues: exploding gradients (can be fixed?) and vanishing gradients (harder to fix).

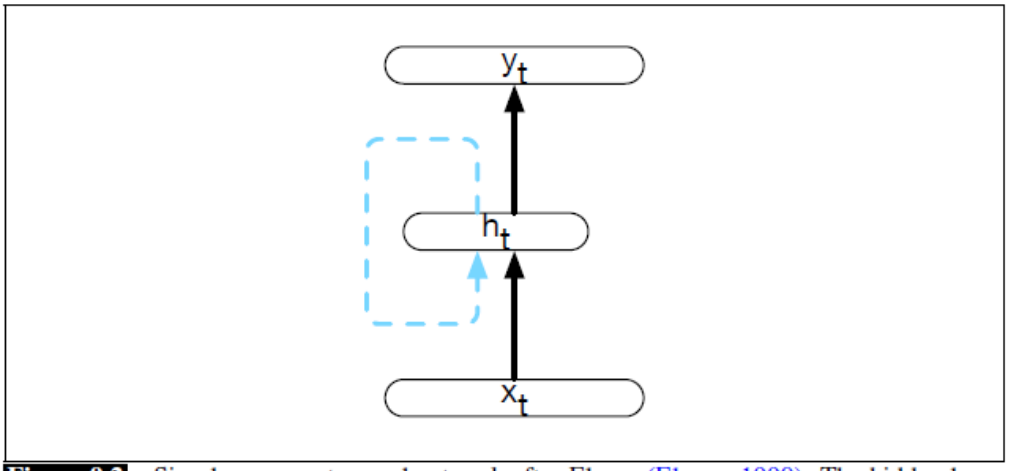


Figure 9.2 Simple recurrent neural network after Elman (Elman, 1990). The hidden layer includes a recurrent connection as part of its input. That is, the activation value of the hidden layer depends on the current input as well as the activation value of the hidden layer from the previous time step.

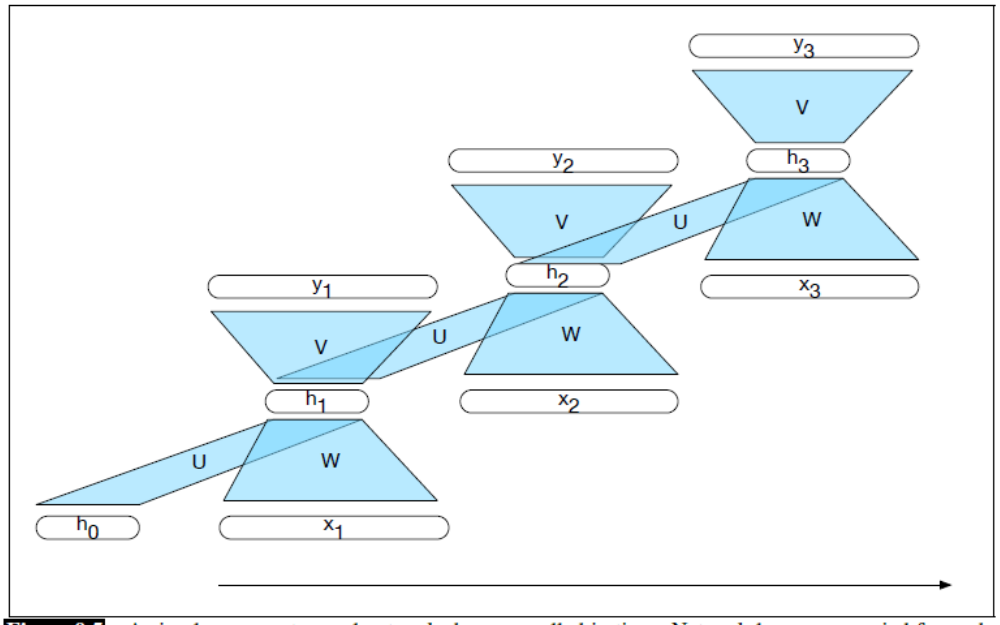
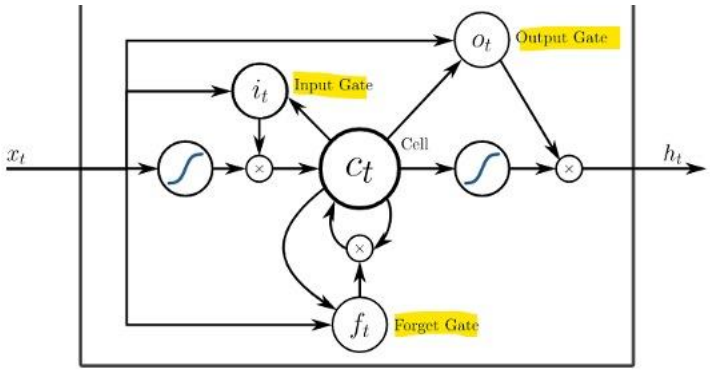


Figure 9.5 A simple recurrent neural network shown unrolled in time. Network layers are copied for each time step, while the weights U, V and W are shared in common across all time steps.

Sequence labeling (POS), NER (IOB), classification w/softmax. Stacked RNNs, bidirectional RNNs

LSTM



RNNs - during backpropagation, **gradients** can become **extremely small (vanish) or extremely large (explode)**, making it difficult to train model. LSTMs manage this by using a system of gates to retain relevant info and forget irrelevant data. They also have a more robust and effective way of handling sequence data than traditional RNNs.

LSTMs - used for seq2seq tasks (translation, text generation), designed to **address the vanishing gradient problem** of RNNs and limitations of standard RNNs in **handling long-term dependencies** - series of gates and hidden states helps remember long-term dependencies in data and controls the flow of info across time steps:

- **Input gate**: selects info to add to current context to be *stored in the cell state* - sigmoid activation:. (values from 0 (retain nothing) to 1 (retain everything)).
- **Forget gate**: deletes unneeded info from the context (decides how much of *prev cell state should be forgotten*) by a weighted sum of the previous state's hidden layer and the current input, passes that through a sigmoid, multiplies by the context vector. Sigmoid activation, values from 0 (forget more) to 1 (forget less).
- **Output gate**: decides what cell state info should be *sent to the next layer* - combination of sigmoid activation (deciding what information to pass) and tanh (to scale the values).

RNNs and LSTMs struggle with capturing long-range dependencies in input data due to their sequential nature - makes it difficult to learn complex patterns over long sequences, especially significantly long ones.

Gated recurrent units (GRUs)

Simplify calc by dispensing separate context vector, and reducing gates to 2 - **reset & update gate**.

End of complete copy from "Basics of ML and DL.docs"