Apr 8, 2025

Things I haven't learnt:

- Threading, Multiprocessing: Pool, map
    - From multiprocessing import ThreadPool, Pool (ProcessPool)
        - with ThreadPool(n) as pool: pool.map(functions, urls)
- Threadpoolexecutor, ProcessPoolexecutor: map
    - executor.map(function, urls)

Concurrency in Python: https://realpython.com/python-concurrency/

- Threading: I/O bound, Overlaps waiting times, Synchronous, Prone to issues like race conditions, Uses multiple threads, Preemptive multitasking
    - Threads can't run in parallel due to Global Interpreter Lock (GIL).
- AsyncIO: I/O bound, Overlaps waiting times, Asynchronous, Uses single thread, event loop, coroutines and non blocking functions, Co-operative multitasking, Generally faster than threading because coroutines are cheaper to create than thread.
- Multiprocessing: CPU bound, Executes multiple processes parallely, Synchronous.

AysncIO: https://realpython.com/async-io-python/

- Top level functions: run. create_task, gather, as_completed, to_thread, wait_for (with timeouts)
- Annotating the code with async and await.
    - async def , async with, aysnc for.
- For blocking functions use asyncio.to_thread() to send it to background.
- Asyncio's roots is in generators.
- Use of asyncio with Queues for producer, consumer problem. q.join(), c.cancel()
- We can chain coroutines in asyncio.

Global Interpretor Lock: https://realpython.com/python-gil/

- Python has reference counting system which requires a lock on the interpreter.
- It could place a lock on the data structures but that would cause deadlock.
- Thread safety due to GIL meant that a lot of C programs were easily rewritten for python leading to increased adoption.
- It is not easy to remove GIL without sacrificing single thread performance.

Questions:

- What is the difference between coroutines and threads ?
    - Coroutines are very lightweight. Threads have high overhead. (Eeach thread needs memory for stack, context switching is costly). Context switching in coroutines is controlled manually using await. It is controlled by OS for threads. Coroutines employs cooperative multitasking where as threads employ preemptive multitasking.
- What is Global Interpreter lock and why is it required in python ?
- What is pre-emptive vs co-operative multitasking ?

    - Pre-emptive multitasking means that OS controls which thread is going to run. In, Co-operative multitasking the coroutines voluntarily give up control using await.
- What are nonblocking and blocking functions ?
    - Blocking functions are those which don't give up control while they are waiting. When blocking functions are waiting the code execution stops and waits. Non-blocking functions voluntarily give up control when waiting and allows multiple tasks to progress without waiting for each other.
- What does it mean that threading and multiprocessing are sync while asyncio is async ?
    - Threading and multiprocessing are synchronous because they still block on operations. Asyncio is asynchronous because it never blocks, the tasks yield control when waiting allowing other tasks to run immediately.
- Why is threading prone to race conditions but asyncio not ?
    - There is only 1 thread in asyncio therefore it is less prone to race conditions. In threading, race conditions can happen anywhere where was in asyncio race conditions can happen only at await.
- What does as_completed do in ThreadPoolExecutor ?
    - Sorts the threads by order in which they finish. Its optional, doesn't change anything because we are having to wait for all threads anyway.
- When would threading be better than asyncio ?
    - AsyncIO is usually faster but not all functions can be non-blocking so sometimes its impractical to use asyncio. In such cases, it is recommended to use multithreading. Use asyncio if you can, Use multithreading if you must.
- Why do I still need to use locks in multiprocessing ?
- Why can't I use a regular dict instead of multiprocessor Manager dict ?

Apr 7, 2025

Evaluation Design

- Pretraining:
    - Perplexity.
    - Additional metrics to track during pre-training: https://chatgpt.com/share/6804ff7c-d77c-800a-b19e-05fb0b65c811
- Zero-shot & Few-shot benchmarks
    - Common Benchmarks:
        - General: MMLU, MMLU-Pro, HLE, TriviaQA, BigBenchHard, IFEval
        - Math: Math, HiddenMath, GSM8k, AIME, Omni Math, Frontier Math
        - Code: CodeGolf, HumanEval, MBPP, SWEBenchVerified, Natural2Code, SWELancer
        - Science: GPQA, GradQA
        - Reasoning: Hella Swag, ARC
        - Multilinguality: WMT '23, MGSM
        - Multimodality: MMMU, TextVQA
        - Factuality: Truthful QA, Simple QA
        - LongContext: Need in Haystack, Infinite Bench, ZeroScrolls, LooGLE.
    - Calibration: Expected Calibration Error.

    - Eval data leakage or contamination.
      Scaling laws for evaluation: Model number of flops with NLL of prediction on benchmark then model NLL with accuracy.
    - ELO Leaderboard or track by Win rates.
    - Different metrics: Accuracy, Pass @ k, Precision, Recall, F1, BLEU, ROUGE, METEOR, BLEURT etc.
- Autoraters: Evaluation Set, Benchmarking, Zero-shot vs Few-shot, Custom Few-shot, Rubrics, Evaluation Plan, Human Eval Collection & Filetring, Synthetic Data, SFT formulation, Rationales & Rubrics. Majority Voting& Confidence Estimation, Length & Position Bias, Hybrid Approach, Autoraters+Tool Use: Search + Code Interpreter.
- Human Evals
    - Use multiple raters per sample | Crowdsourcing vs In-House
    - Rubrics:Clear rubrics with calibration examples. | Latency vs Depth
    - Rater Quality Test, Diversity of Raters, Language Proficiency | Interleaved gold & dummy
- Inner loop vs Outer loop
- Additional metrics to track during RL:
  https://andyljones.com/posts/rl-debugging.html
- Talk about evaluating different capabilities instead of just one ??
    - Multilinguality, Multimodality
- Talk about Safety Evaluation & Redteaming.
- Robustness: Test on perturbed inputs: typos, negations, distractors

Apr 6, 2025

- Matrix Serialization and Deserialization:
  https://chatgpt.com/share/68058011-96cc-800a-a576-078167078fea
- Trillion Token Data Iterator:
  https://chatgpt.com/share/68058087-d82c-800a-a03b-0a4da0bcc429

Important things to add time in Design Interviews:
- Post-training:
    - Distillation

Apr 5, 2025

Supervised Fine-tuning Data:
- Sources of Data:
    - Human Annotated: Employ humans to write prompts and a suitable response for the prompt. To promote diversity in this dataset, we could come up with a set of topics for which we want the prompts and hire humans with expertise in that topic.
    - Synthetically Generated:

- Prompts with Verifiable Rewards: For use-cases such as math and coding, we need to collect prompts with verifiable answers. We can collect such prompt, answer pairs from textbooks, highschool exams, programming websites.

- Data Filtering Techniques:
  - Heuristic Based Filtering: Eyeball the data and remove specific things such as overuse of emojis
  - Difficulty Classifier: Use a few-shot model to categorize the prompts and responses by difficulty and sample a good mix of easy medium and difficult prompts.
  - Topic Classifier: Use a few-shot topic classifier on prompts to ensure coverage and diversity of different types of topics.
  - Model Based Filtering: Use a few-shot instruction tuned model to rate what is the quality of a response given the prompt.
  - Rejection Sampling: Use a reward model to score the prompt response pairs and keep the highest scores in the training data.
  - Semantic Deduplication: Cluster prompt + response pairs using RoBERTA. Within each cluster sort them by difficulty score x quality score. We then do greedy selection by iterating through all sorted examples.
- Capability specific Data:
  - Code:
    - Functions extracted from Github and their docstrings converted to prompts and Unit tests synthetically generated.
    - Code snippets extracted from various sources and the model asked the generate problems inspired by those code snippets. Generate solutions using a LLAMA model. Add general rules of programming in a particular language (similar to context distillation) to improve the quality of generated code. For each problem and solution, we prompt the model to generate unit tests. Run code through parser and linter to ensure syntactic correctness. When a solution fails at any step, we prompt the model to revise it. The prompt included the original problem description, the faulty solution, and feedback from the parser/linter/tester (stdout, stderr/ and return code). After a unit test execution failure, the model could either fix the code to pass the existing tests or modify its unit tests to accommodate the generated code.
    - Translating a piece of code from one language to another to generate more data in another language and check for correctness using syntax parsing, compilation, and execution.
    - Translating a piece of code to docstring and translating it back to code and checking if it passes all the test cases the original code passed. If yes, add it to the dataset.
    - System prompt steering: We used code specific system prompts to improve code readability, documentation, thoroughness, and specificity.
  - Safety:
    - Mixture of Unsafe and Borderline prompts.
    - Safety context distillation, Use the system prompt, “You are a safe and responsible assistant” to generate data and use that data without the system prompt for fine-tuning. Use this in conjunction with a reward model and only use the data if it increases the RM score.
  - Factuality:

        - Don’t add new information that model hasn’t already seen during pre-training. Generate responses from the model itself and filter it using a reward model or Factscore.
    - Multilinguality:
        - Translate and Translocalize the prompts and responses into new language.
        - **Multilingual data from NLP tasks rewritten into prompt response pairs eg. NER, dependency parsing etc.**
        - Ensure high quality language match rate between the prompt and response.
        - Use a multilinguality expert model to collect data. We train specific model for multilinguality with more multilingual tokens than our regular model.
    - Tool Use:
        - Synthetically annotate the responses with tool calls in a few-shot manner. Sample k positions where probability of generating tool call token is very high and make tool calls if the NLL loss on future tokens decreases as a result of that tool call.
    - Math:
        - Problems are paired with solutions in chain-of-thought (CoT), program-of-thought (PoT) and tool integrated reasoning format.
        - Source relevant pre-training data from mathematical contexts and converted it into a question-answer format which can then be used for supervised finetuning.
        - Teach model how to self refine using incorrect solution in context and suggested change and then the final answer as the response.
        - Generate intermediate reasoning steps and employ a process reward model to filter those intermediate steps.
    - Long Context:
        - Question Answering Data
        - Summarization
        - Long Context code reasoning: We remove one of these key files from a repository and prompt the model to identify which files depended on the missing file and to generate the necessary missing code.

Preference Data:

- Sources of Data:
    - Human Feedback: Human written prompts. The two responses generated by diverse models: different temperature, different sampling hyperparameters, different iteration of models, different stages of training etc.
    - AI Feedback: Human written prompts or Synthetically generated prompts tweaked from the original by modifying style etc. The two responses generated by 2 different models similar to previous step. Instead of using a Reward model trained on Human Feedback, we use a reward model trained on AI preferences. AI preferences are gathered using a few-shot instruction tuned model.
- Data Filtering Techniques:
    - Unclear intent or Vauge prompts: [Optionally] Remove prompts with unclear intent or prompts that are vague in nature.
    - Model based Filtering: Remove examples where both responses are poor quality.

    - Interannotator agreement or AI Feedback confidence: If there is high variance in labels, filter out those prompts.
    - Filter the data by the degree of preference: Filter out 1 on Likert scale.
    - Deduplication.
    - Timing Heuristics:
    - Only onboard raters who pass a short quiz on your domain.
- Capability specific Data:

Reinforcement Learning Data:
- Sources of Data:
    - Human written prompts:
    - Synthetically generated:
- Data Filtering techniques:
    - Difficulty Classification
    - Unclear intent or Vauge prompts
    - Topic Classification:
    - Quality Classification:
    - Semantic Deduplication.

Reasoning Data:
- Sources of Data:
    - RL Data:
        - Hendyrcks Math
        - US High School Competitions
        - IMO
        - OlympiadBench
        - Omni Math
        - AOPS
        - AtCoder
        - Codeforces

Apr 4, 2025

Graph
- Djistra’s
- Floyd Warshall Algorithm.
- Bellman Ford Algorithm
- Minimum Spanning Tree: Union Find with sorted edges.
- Topological Sort: Keep track of indegree.

Apr 3, 2025

Binary Search Problems:
- Find Peak Element: Comparator nums[m]>nums[m+1].

- Find First and Last Position of Element in Sorted Array: First position condition: nums[mid]>=target, Last position condition: nums[mid]>target. Use lo-1 if not equal to target.
- Binary Search: nums[mid]>=target
- Random Pick with Weight: nums[mid]>=target.
- Find kth positive: arr[mid] - (mid + 1) >= k
  - hi = len(arr)
  - lo<=hi, hi=mid-1, return lo+k
- Missing Element in Sorted Array: Same as Find kth positive.
- Cutting RIbbons: not isPossible(t), lo if isPossible(lo) else lo-1
- Search a 2D matrix: 2 level binary search:
  - matrix[mid][0] >= target, lo - 1 if matrix[lo][0] > target
  - matrix[row][mid] >= target

Apr 2, 2025

The N Implementation Details of RLHF with PPO
- The reward model and policy's value head take input as the concatenation of query and response.
- Pad with a special padding token and truncate inputs. OAI sets a fixed input length for query query_length; it pads sequences that are too short with pad_token and truncates sequences that are too long.
- Adjust position indices correspondingly for padding tokens.
- Response generation samples a fixed-length response without padding. The code would keep sampling until a fixed-length response is generated. Notably, even if it encounters EOS (end-of-sequence) tokens, it will keep sampling.
- Learning rate annealing for reward model and policy training.
- Use different seeds for different processes when using data parallelism.
- The reward model only outputs the value at the last token.
- Scale the logits by sampling temperature.
- Disable dropout
- Discount factor = 1
- Per-token KL penalty
- Reward Normalization.
- Adaptive KL: The KL divergence penalty coefficient β is modified adaptively based on the KL divergence between the current policy and the previous policy. If the KL divergence is outside a predefined target range, the penalty coefficient is adjusted to bring it closer to the target range.
- PyTorch Adam optimizer (torch.optim.Adam.html) has a different implementation compared to TensorFlow's Adam optimizer. Difference in epsilon term. In other words, pytorch adam goes for more aggressive gradient updates early in the training.
  - Because of the aggressive update, the ratio gets clipped 4.4x more often, and the approximate KL divergence is 6x larger.
- Use a really large batch size
- Log excessively:
  - Relative policy entropy
  - Kullback-Leibler divergence

    - Residual variance
    - Terminal correlation
    - Penultimate terminal correlation
    - Value target distribution
    - Reward distribution
    - Value distribution
    - Advantage distribution
    - Episode length distribution
    - Sample staleness
    - Step statistics
    - Gradient statistics
    - Gradient noise
    - Component throughput
    - Value trace
    - GPU stats
    - Reward per trajectory
    - Mean value
    - Policy and value losses

Weight initialization in Neural Networks:

- Xavier Initialization (Tanh Sigmoid)
    - Glorot initialization selects weight variance to keep the output signals and gradients stable throughout layers
    - Forward propagation stability → variance scales with 1/n_in
    - Backpropagation stability → variance scales with 1/n_out
    - Glorot combines these into a balanced choice using harmonic mean: 2/(n_in + n_out)
    - U[−a,a] has variance (a**2)/3 which gives us the uniform distribution version.
- He Weight Initialization (ReLU):
    - Double the variance since -ve values are lost. G (0.0, sqrt(2/n_in))

Apr 1, 2025

Code Review:

- Readability
    - Add docstrings to functions.
    - Annotate input and output types.
    - Check for type error in inputs.
    - If code length > 80 shorten those lines.
    - Descriptive Variable Names
    - Avoid Hardcoding Constants.
    - No Dead Code: Unused Imports or Variables

    - No Functions > 50-100 lines long
- Pythonic Elements:
    - Use list comprehension wherever possible.
    - Never use mutable objects (like `list`, `dict`, `set`) as default arguments.
    - Modifying a List While Iterating. mylist.remove(x): ou skip elements because the list is changing.
    - Variable names are builtins or imports.
    - Never use Bare Except. It swallows everything including Keyboard Interrupt. Catch specific exceptions.
    - One function/class tries to do everything — parsing, computing, printing, logging. ✅ Break down into smaller functions/modules.
    - Deep Nesting: More than 3 levels of indentation should be avoided.
    - Multiple blocks that look *almost* the same — extract to a function.
    - Class exposes too many internal details, violating encapsulation.
    - Avoid doing real work (like file I/O, DB calls) in constructors unless it's clearly documented or necessary.
    - Using `is` for Value Comparison. Use `==` unless comparing **identity**.
    - Avoid Lambdas in Loops
    - Don't modify global variables.
    - Inefficient Loops with indexing, Use element directly or enumerate.
    - Avoid using + in Loops for Strings. Instead use "".join(list)
    - Modifying inputs Unintentionally. Make a copy.
    - For code with randomness. Set Seed for reproducibility.
    - See if you can use vectorize the code using batches in numpy or torch.
    - Use modern strings: f"Test Accuracy: {acc:.2f}\n"
- Obvious Bugs
    - Check for division by zero.
    - Check for index out of bounds error.
    - Logic Bugs / Off-by-One Errors.
- Testing:
    - Unit tests for functions.
- File handling:
    - Use with open as f for file handling.
- Style & Maintainability:
    - Missing spaces: x=1 vs x = 1
    - Long lines > 79 or 88 characters.
    - Inconsistent indentation (spaces vs tabs).
- Specific packages:
    - Use pandas apply instead of for loop.

Mar 31, 2025

LM Arena: Platform for crowdsourced human evaluation. The users can upload their own prompts and 2 anonymous models would be chosen and their response would be shown to the users. The users are then asked to compare the responses and choose between ‘A’ is better, ‘B’ is better, ‘Tie’ or ‘Both are Bad’.

- ELO Score: Elo is a widely used for competitive games such as chess. The difference in the ratings between two players serves as a predictor of the outcome of a match.

  E_A = 1/ (1 + 10 ** (R_B-R_A)/400)

  R_A = R_A + K (S_A - E_A)
  - An issue with this update is that it is sensitive to the order of battles.
  - Another way to predict ELO score is using maximum likelihood estimation using logistic regression. The labels predict whether A is a winner. The features are of size m where m is the number of models and they are set to +log(base) for model a and -log(base) for model b.
- An alternative to Elo would be to use average Win rate against other models. However, this method may not be as scalable as the Elo rating system because this method requires data from all model combinations.
- Datasets:
  - https://huggingface.co/datasets/lmsys/chatbot_arena_conversations
  - https://huggingface.co/datasets/lmsys/lmsys-chat-1m

Mar 30, 2025

Data Parallelism

- DataParallel: Single-process, multi-threaded. Works when the model fits on a single GPU. Keep a copy of the model on each GPU. Do forward and backward pass for a microbatch on each GPU separately. Average the gradients, perform weight update and send the updated copy of the weights to all GPUs. The main bottleneck is that it relies on **single-process, multi-threaded communication**, leading to inefficient inter-GPU communication and potential slowdowns due to CPU overhead. DataParallel runs in a single process with multiple threads, so it suffers from GIL contention. At the end of each minibatch, workers need to synchronize gradients or weights to avoid staleness. There are two main synchronization approaches and both have clear pros & cons.
  - Bulk synchronous parallels (BSP): Workers sync data at the end of every minibatch. It prevents model weights staleness and good learning efficiency but each machine has to halt and wait for others to send gradients.
  - Asynchronous parallel (ASP): Every GPU worker processes the data asynchronously, no waiting or stalling. However, it can easily lead to stale weights being used and thus lower the statistical learning efficiency. Even though it increases the computation time, it may not speed up training time to convergence.
- Distributed Data Parallel: Each GPU has its own process. Can work on multiple nodes/machines. Uses Ring all reduce algorithm avoiding a central bottleneck. DDP has lower communication overhead.
- ZeRO: ZeRO-DP has three main optimization stages:
  - Optimizer State Partitioning: 4x memory reduction, same communication volume as DP. Gradient computation can be done independently for each GPU and when it is being done for

parameters that are not on the current GPU, we incur a communication cost but this is the same as gradient averaging in DP.

    - Add Gradient Partitioning: 8x memory reduction, same communication volume as DP. This is similar to optimizer state partitioning in practice. Optimizer states are calculated per parameter anyway so this doesn't incur any extra cost.
    - Add Parameter Partitioning: Memory reduction is linear with DP degree Nd. For example, splitting across 64 GPUs (Nd = 64) will yield a 64x memory reduction. There is a modest 50% increase in communication volume. This works because at any time doing forward or backward, only a subset of parameters (in a layer) for example are required for the operation. At best you're gonna need memory equivalent to a layer size. The model parameters can be sliced in any manner (vertically or horizontally). The way its different from tensor parallelism or pipeline parallelism is that every computation still happens on each GPU using full tensors, just that the parameters are not all on single GPU.
- FSDP (Fully Sharded Data Parallel): It is same as ZeRO. Just a different name.
- DeepSpeed: Deepspeed is an open source implementation of Zero-DP.

Context Parallelism: Context Parallelism is about how to parallelize the sequence length into multiple GPUs.

- Blockwise Parallel Transformer for Large Context Models:
    - Ring Attention with Blockwise Transformers:
- Flash Attention:
- SELF-ATTENTION DOES NOT NEED O(n**2) MEMORY:
- LLAMA3:
- Context Parallelism for Scalable Million-Token Inference:

Expert Parallelism:

- Mixture of Experts:
- Device Balance Loss:
- Communication Balance Loss:
- Auxiliary Free Load Balancing:

Communication:

- All reduce
- All gather
- Reduce Scatter

Mar 29, 2025

Optimizers:

- Stochastic Gradient Descent: Do a forward pass with one example, calculate the gradients wrt loss and subtract the gradient multiplied by a learning rate from parameter. An issue with stochastic gradient descent is that it can get stuck in saddle point or a local minima. What if loss changes quickly in one

direction and slowly in another?: Very slow progress along shallow dimension, jitter along steep direction
  - Mini batches: Do a gradient update by accumulating loss on a small batch.
- SGD + Momentum: Continue moving in the general direction as the previous iterations. Build up “velocity” as a running mean of gradients. Momentum has a shot at escaping local minima (because the momentum may propel it out of a local minimum). In a similar vein, as we shall see later, it will also power through plateau regions better.

  V_t+1 = V_t + G(t)

  X_t+1 = X_t - alpha* V_t+1
- Adagrad: Adds element-wise scaling of the gradient based on the historical sum of squares in each dimension (without decay). Issue is that it is too slow because the sum of squares in each dimension add up fast.
- RMSProp: Adds element-wise scaling of the gradient based on the historical sum of squares in each dimension (with decay). “Per-parameter learning rates” or “adaptive learning rates”. Idea is that if a direction has taken too many big steps, we slow it down, otherwise if it is too slow, we speed it up. Progress along “steep” directions is damped. Progress along “flat” directions is accelerated.
- Adam: Combines momentum and RMSprop. Tracks first moment and second moment. Momentum helps accelerate the optimization process, while RMSprop helps adapt each parameter's learning rate. We perform bias correction in Adam because the initial values for momentum and variance is 0.
  - Weight Decay: In Adam, weight decay is often added as an L2 regularization term to the loss function: L(θ) + (λ/2) * ||θ||^2. However, adding this term to the loss affects the adaptive learning rates, which can hinder optimal convergence. Adaptive learning rate mechanisms can reduce or amplify the intended regularization effect. AdamW is a modification of Adam that incorporates decoupled weight decay, which improves generalization and training stability. In AdamW, Here, the weight decay term λ * θt is added directly to the update, without affecting the adaptive learning rates.
- Adadelta: Similar to RMSProp in that it also adds a decay factor to sum of gradient squares in each dimension. This is unique in the sense that you don’t have to set a learning rate. The method automatically computes a learning rate.

$$g_{t+1} = \gamma g_t + (1-\gamma)\bigtriangledown \mathcal{L}(\theta)^2$$

$$x_{t+1} = \gamma x_t + (1-\gamma) v_{t+1}^2$$

$$v_{t+1} = -\frac{\sqrt{x_t + \epsilon}\,\delta L(\theta_t)}{\sqrt{g_{t+1} + \epsilon}}$$

$$\theta_{t+1} = \theta_t + v_{t+1}$$

- Second Order Optimization Methods: Use gradient and Hessian to form quadratic approximation. Step to the minima of the approximation. It tells you how big of a step to take. Gradient just gives you a direction. Second order gives you exactly where to step. Hessian has O(n^2) elements. Taking inverse

means O(N^3) time complexity and N is in order of billions. Hence we avoid second order optimization for deep learning.
- Shampoo: AdaGrad and Adam constrain the preconditioning matrices to be diagonal. Shampoo bridges the gap between full matrix preconditioning and the diagonal version by approximating the full matrices by a Kronecker product. It computes **preconditioning matrices** along multiple tensor dimensions, rather than a single diagonal vector as Adam does. Typically, for a parameter tensor with dimensions W∈Rm×nW \in \mathbb{R}^{m \times n}W∈Rm×n, Shampoo keeps track of two matrices:
    - GtrowG_t^{row}Gtrow, capturing correlation along rows.
    - GtcolG_t^{col}Gtcol, capturing correlation along columns.
    - Gtrow=Gt−1row+gtgt⊤,Gtcol=Gt−1col+gt⊤gt
    - Ptrow=(Gtrow)−1/4,Ptcol=(Gtcol)−1/4
    - Wt+1=Wt−η⋅PtrowgtPtcol
- Adafactor:

Learning Rate Schedules:
- Linear Warmup: Big learning rates at the beginning of the training cause gradients to explode, so we start with a linear warmup.
- Inverse Square root Decay: alpha / sqrt(t)
- Cosine Decay: alpha * (1-t)/T
- Step Decay: Linear warmup followed by reducing the learning rate my multiplying it with a factor of 0.316 after 80% steps and multiplying it again by a factor of 0.316 after 90% steps.

Mar 28, 2025

Pytorch Basics:
- Common Used:
    - nn.Module -> forward.
    - nn.Linear()
    - F.softmax
    - F.silu
    - transpose(dim1, dim2)
    - @ for matrix multiplication
    - torch.einsum
    - torch.outer
    - torch.where : Return a tensor of elements selected from either `input` or `other`, depending on `condition`.
- Indexing: 3 dimensional indexing requires you to expand dim across the first dimension.
- One hot vectors: You can do torch.eye(num_classes)[y].
- Gather: You can replace this by indexing as well but over here the idea is to have a src matrix, a dst matrix and indices. Along the dimension you're gathering, indices could have 1:k dimension and the src matrix will get those dimensions.

- Scatter:

Mar 27, 2025

Scaling Laws:

- Scaling Laws for Neural Language Models:
    - Performance depends strongly on scale, weakly on model shape. Within reasonable limits, performance depends very weakly on other architectural hyperparameters such as depth vs. width.
    - Performance has a power-law relationship with each of the three scale factors N, D, C when not bottlenecked by the other two, with trends spanning more than six orders of magnitude.
    - Performance improves predictably as long as we scale up N and D in tandem, but enters a regime of diminishing returns if either N or D is held fixed while the other increases. Every time we increase the model size 8x, we only need to increase the data by roughly 5x to avoid a penalty.
    - When working within a fixed compute budget C but without any other restrictions on the model size N or available data D, we attain optimal performance by training very large models and stopping significantly short of convergence.
    -
    -
- LLAMA 3: Scaling laws are developed to determine optimal model size given the compute budget. It is also used to predict the performance on downstream benchmarks:
    - Compute optimal Model: For compute budget between 6x10^18 flops and 10^22 flops, we plot ISOFlop curves. On the X axis we have number of training tokens, Using flop budget and number of training tokens, we can compute the model size. On the y axis we have NLL corresponding to the model size. For each Flop budget, we approximate the ISOFlop curve using a second degree polynomial. We get the minimum of these ISOFlop curves as the optimum model size. We use the compute-optimal models we identified this way to predict the optimal number of training tokens for a specific compute budget. To do so, we then plot #Training tokens on the Y axis and the number of Flops on the X axis. We assume a power-law relation between compute budget, C, and the optimal number of training tokens, N(C) = AC^α. We find that (α, A) = (0.53, 0.29). Extrapolation of the resulting scaling law to 3.8 × 10^25 FLOPs suggests training a 402B parameter model on 16.55T tokens.
    - Predicting the performance on downstream benchmarks: A two stage method is used: First, we linearly correlate the (normalized) NLL of correct answer in the benchmark and the training FLOPs. We use this to estimate the NLL of the flagship model on the benchmark. Then NLL is correlated with accuracy using the scaling laws models and the older models trained on higher FLOPs. We establish a sigmoidal relation between the log-likelihood and accuracy using both the scaling law models and Llama 2 models. Using the predicted NLL in the previous step, we estimate the accuracy using the sigmoidal relation.

Apr 26, 2025

Long Context Extension:

- Modeling:
    - Position Interpolation: Down-scale the position indices so that the maximum position index matches the previous context window limit in the pre-training stage.

      $$f(x,m) = f(x,mL/L')$$

        - After finetuning on longer sequences, the perplexity slightly increases for short sequences compared with the original pretrained model.
    - NTK Methods: "NTK-aware" interpolation by taking the loss of high frequency into account.
        - Dynamic NTK:
        - NTK-by-parts:
    - Yarn:
    - CLEX:
    - LLAMA3: Increase the supported context length in increments in 6 stages, pre-training until the model has successfully adapted to the increased context length. This long-context pre-training stage was performed using approximately 800B training tokens.
    - NoPE: Removing positional embeddings from global attention layers.
    - Change the attention:
        - Sliding Window Attention (LM Infinite)
        - Block Diagonal Attention (Long LORA)
        - Capped Length (Self-Extend)
        - Binned Length (Landmark Attention)
    - Increasing the theta: We can increase the theta of the rope embeddings which can be thought of as similar to position interpolation. **Higher theta** → better long-context generalization, but worse resolution at short distances.
- Evaluation:
    - Needle in a Haystack: A keyword is hidden at a specific depth in the context and we check if the model is able to retrieve it or not. We plot this for different context length giving us a 2d plot where on the x axis we have the context length and on the y axis we have the depth %.
        - Averaged NLL by context position
        - Perplexity
    - ZeroSCROLLS/QuALITY:
    - InfiniteBench/En.MC:
    - L-Eval:
    - Bamboo:
    - RULER:
    - LongBench:
    - LooGLE:

Mar 25, 2025

Efficient Transformers:https://chatgpt.com/share/67e9c82c-bdcc-8010-a57e-e874caaa51a7

- BigBird: Uses a combination of local, random, and global attention patterns to reduce complexity to O(n).
- Longformer: Utilizes a combination of sliding window (local) attention and global attention to improve efficiency.

Mar 24, 2025

Inference Optimization techniques in LLMs:

- Reducing kV Cache:
    - Grouped Multi Query Attention: Reduce kv cache memory by grouping multiple queries with same keys and values.
    - Multi-head Latent Attention: Have a shared downsized representation for keys and values and a similarly downsized representation for queries.
    - Cross Layer KV-sharing: We tie the KV cache across neighboring attention layers, For global attention layers, we tie the KV cache of multiple global layers across blocks.
    - Interleaving Local and Global Attention Layers: 1 in every 4 or 6 layer is a global attention layer.
- Speculative Decoding: Use a smaller draft LLM for generating response and use the target LLM to verify the response.
- Distillation: Build a smaller, cheaper model (“student model”) to speed up inference by transferring skills from a pre-trained expensive model (“teacher model”) into the student.
- Quantization:
    - Post-Training Quantization (PTQ): A model is first trained to convergence and then we convert its weights to lower precision without more training. It is usually quite cheap to implement, in comparison to training. As the model size continues to grow to billions of parameters, outlier features of high magnitude start to emerge in all transformer layers, causing failure of simple low-bit quantization.
        - Mixed-precision quantization: The most straightforward approach for resolving the above quantization challenge is to implement quantization at different precision for weights vs activation.
            -
    - Quantization-Aware Training (QAT): Quantization is applied during pre-training or further fine-tuning.

Sources:

- https://research.character.ai/optimizing-inference/
- https://lilianweng.github.io/posts/2023-01-10-inference-optimization/

Mar 23, 2025

Pipeline Parallelism

- Naive Model Parallel: Partition the model by layers and put each partition on a separate GPU. The main deficiency and why this one is called “naive” MP, is that all but one GPU is idle at any given moment.

- Gpipe: Pipeline parallelism (PP) combines model parallelism with data parallelism to reduce inefficient time “bubbles’’. The main idea is to split one minibatch into multiple microbatches and enable each stage worker to process one microbatch simultaneously. Given m evenly split microbatches and d partitions, the bubble is

  $$(d-1)/(m+d-1)$$

  - Activation Recomputation: Only activations at partition boundaries are saved and communicated between workers. Intermediate activations at intra-partition layers are still needed for computing gradients so they are recomputed during backward passes. With activation recomputation, the memory cost for training M(l) is

    $$M(l) = O(l/d) + O(d) = O(\mathrm{sqrt}(l))$$
- Pipedream: It schedules each worker to alternatively process the forward and backward passes. PipeDream does not have an end-of-batch global gradient sync across all the workers, an naive implementation of 1F1B can easily lead to the forward and backward passes of one microbatch using different versions of model weights, thus lowering the learning efficiency. PipeDream proposed a few designs to tackle this issue: Weight stashing: Each worker keeps track of several model versions and makes sure that the same version of weights are used in the forward and backward passes given one data batch. Vertical Sync: The version of model weights flows between stage workers together with activations and gradients. Then the computation adopts the corresponding stashed version propagated from the previous worker. This process keeps version consistency across workers.
  - Pipedream-flush: PipeDream-flush adds a globally synchronized pipeline flush periodically, just like GPipe.
  - Pipedream-2BW: PipeDream-2BW maintains only two versions of model weights, where “2BW” is short for “double-buffered weights”. It generates a new model version every k microbatches and k should be larger than pipeline depth d. A newly updated model version cannot fully replace the old version immediately since some leftover backward passes still depend on the old version. In total only two versions need to be saved so the memory cost is much reduced.
- Breadth First Pipeline Parallelism: Looped pipeline with the principle of Gpipe constitutes breadth first search approach where as Looped pipeline with principle of 1F1B constitutes a depth first search approach.
- Zero Bubble Pipeline Parallelism: Split Backward pass into two: Backward for Input and Backward for weights. Backward for input needs to be done first, backward for weights can be done later.
  - ZB-H1: Bubble reduction is because B is initiated earlier across all workers compared to 1F1B, and the tail-end bubbles are filled by the later-starting W passes.
  - ZB-H2: We introduce more F passes during the warm-up phase to fill the bubble preceding the initial B. We also reorder the W passes at the tail, which changes the layout from trapezoid into a parallelogram, eliminating all the bubbles in the pipeline.
    - Bypassing optimizer synchronization: Use post-validation strategy to replace optimizer synchronization.
- LLAMA3: Current implementations of pipeline parallelism have batch size constraint, memory imbalance due to embedding layer and warmup microbatches and computation imbalance due to output & loss calculation making the last stage execution latency bottleneck. **They modify the pipeline schedule to run an arbitrary number of microbatches in each batch**. To balance the pipeline, we reduce one Transformer layer each from the first and the last stages, respectively. This means that the

first model chunk on the first stage has only the embedding, and the last model chunk on the last stage has only output projection and loss calculation.
- DeepSeek-V3: The key idea of DualPipe is to overlap the computation and communication within a pair of individual forward and backward chunks. It employs a bidirectional pipeline scheduling, which feeds micro-batches from both ends of the pipeline simultaneously and a significant portion of communications can be fully overlapped.

Tensor Parallelism:
- Column-wise Parallel: A= [A1 | A2] O = [X@A1 | X@A2]
- Row-wise Parallel A = [A1, A2] X = [X1 | X2] O = [X1 @ A1 + X2 @ A2]
- Column_wise Parallel first followed by Row-wise Parallel which naturally expects to split input by columns.
- TP splits the model vertically, partitioning the computation and parameters in each layer across multiple devices, requiring significant communication between each layer. As a result, they work well within a single node where the inter-GPU communication bandwidth is high, but the efficiency degrades quickly beyond a single node.
- Megatron-LM: Open source implementation of Tensor Parallelism with Transformers.

Mar 22, 2025

Partial Reward Models:
- Motivation: Sometimes the model is thinking in the right direction but it makes a small calculation mistake somewhere and gets a zero reward. We want to provide these models with some credits so that they can learn that they are heading in the right direction. This is particularly important when the problems at hand are too difficult. These are also useful when the solution itself is not verifiable for example a mathematical proof.
- Assigning a partial reward model is a 2 step process:
  - We first use a few-shot technique to generate a grading scheme given a question and its solution.
  - We then use the question, grading scheme and the generated solution to assign a partial reward to the solution.
- Check for the correctness of the grading scheme:
- Reward Hacking with Partial Rewards:

Process Reward Models:
- Let’s Verify Step by Step:
  - PRM800k: 800k step level rewards to 75k solutions to 12k problems. Strategically choose which response to surface to the human annotators. Choose responses that scores high with current PRM but are wrong. This active learning strategy heavily biases the dataset towards wrong answer solutions.
  - Generator: We finetune the generator to produce the output in step by step format. Specifically, we few-shot generate solutions to MATH training problems, filter to those that reach the correct final answer, and fine-tune the base model on this dataset for a single epoch. This step is not

intended to teach the generator new skills; it is intended only to teach the generator to produce solutions in the desired format.

  - Baseline: Outcome Reward model (ORM) is trained to predict whether each solution is correct or incorrect. We determine correctness by automatically checking the final answer.
  - Training: We train PRMs using next token prediction to predict the correctness of each step after the last token in each step. This prediction takes the form of a single token, and we maximize the log-likelihood of these target tokens during training.
    - When we provide process supervision, we deliberately choose to supervise only up to the first incorrect step. This makes the comparison between outcome and process supervision more straightforward. For correct solutions, both methods provide the same information, namely that every step is correct. For incorrect solutions, both methods reveal the existence of at least one mistake, and process supervision additionally reveals the precise location of that mistake.
  - Inference: To determine the step-level predictions at test time, it suffices to perform a single PRM forward pass over the whole solution. We define the PRM score for a solution to be the probability that every step is correct under the PRM.
  - Evaluation Setup: Uses process reward models to do Best of N ranking and check accuracy. They don't use that as a reward for reinforcement learning.
  - Reward Hacking Behaviors:
    - If we use the product of the probabilities, the models would choose solutions that producing fewer steps as with each additional step, the probability of the solution will go down.
    - If we use the average of the probabilities, the models would choose solutions that have redundant steps that got a high score.
  - Comparing Outcome and Process Supervision on the same data: We use PRM_large to generate process supervision and outcome supervision data for training PRM and ORM respectively. We can also use accuracy check to obtain outcome supervision data. Out of the three PRM performs the best followed by ORM supervised by PRM_large and accuracy check supervision is the last.
- DeepSeekMath:
  - Deploy process reward model which provides a reward at the end of each reasoning step. These rewards are normalized by subtracting mean and dividing by standard deviation. The process supervision calculates the advantage of each token as the sum of the normalized rewards from the following steps.
- DeepSeekR1:
  - PRM has three main limitations that may hinder its ultimate success. While it demonstrates good ability to rank N candidates its advantages are limited compared to the additional computational overhead it introduces during the large-scale reinforcement learning process.
    - First, it is challenging to explicitly define a fine-grain step in general reasoning.
    - Second, determining whether the current intermediate step is correct is a challenging task. Automated annotation using models may not yield satisfactory results, while manual annotation is not conducive to scaling up.
    - Third, once a model-based PRM is introduced, it inevitably leads to reward hacking.

Scaling LLM Test-Time Compute Optimally can be More Effective than Scaling Model Parameters:

Anthropic Screening: https://aonecode.com/iq/docs/antropic/online-assessment/in-memory-db

Mar 21, 2025

Tokenization in LLMs:

- Word Tokenization: Split based on spaces and take care of punctuation. Vocab size = 260k for only English language in transformers XL.
- Character Tokenization: Loss of quality due to loss of semantic meaning. Harder to learn context independent representation. Also longer sequences and increase in computation.
- Subword Tokenization: Frequently used words should not be split into smaller subwords, but rare words should be decomposed into meaningful subwords. Subword tokenization allows the model to have a reasonable vocabulary size while being able to learn meaningful context-independent representations.
    - BPE: Uses pre-tokenization into words. Keep count of words. Start with a base vocabulary with all character. Consider pairs of elements in base vocabulary and count their occurrences, merge the one with the highest occurrence and add them into vocab.
        - Byte level BPE: Only consider bytes as the base vocabulary.
    - WordPiece: Similar to BPE, starts with a base vocab with all characters and chooses the symbol pair that maximizes the likelihood of the training data once added to the vocabulary.

      score=(freq_of_pair)/(freq_of_first_element×freq_of_second_element)

      By dividing the frequency of the pair by the product of the frequencies of each of its parts, the algorithm prioritizes the merging of pairs where the individual parts are less frequent in the vocabulary. Tokenization differs in WordPiece and BPE in that WordPiece only saves the final vocabulary, not the merge rules learned. Starting from the word to tokenize, WordPiece finds the longest subword that is in the vocabulary, then splits on it. hen the tokenization gets to a stage where it’s not possible to find a subword in the vocabulary, the whole word is tokenized as unknown — so, for instance, "mug" would be tokenized as ["[UNK]"], as would "bum" (even if we can begin with "b" and "##u", "##m" is not the vocabulary, and the resulting tokenization will just be ["[UNK]"], not ["b", "##u", "[UNK]"]). This is another difference from BPE, which would only classify the individual characters not in the vocabulary as unknown.
    - Unigram: Starts with a larger base vocab and trims down the vocab. The base vocabulary could for instance correspond to all pre-tokenized words and the most common substrings. There are several options to use to build that base vocabulary: we can take the most common substrings in pre-tokenized words, for instance, or apply BPE on the initial corpus with a large vocabulary size. At each training step, the Unigram algorithm defines a loss (often defined as the log-likelihood) over the training data given the current vocabulary and a unigram language model. Then, for each symbol in the vocabulary, the algorithm computes how much the overall loss would increase if the symbol was to be removed from the vocabulary. Unigram then

    removes p (with p usually being 10% or 20%) percent of the symbols whose loss increase is the lowest, i.e. those symbols that least affect the overall loss over the training data. This process is repeated until the vocabulary has reached the desired size. The Unigram algorithm always keeps the base characters so that any word can be tokenized. Unigram saves the probability of each token in the training corpus on top of saving the vocabulary so that the probability of each possible tokenization can be computed after training. The algorithm simply picks the most likely tokenization in practice, but also offers the possibility to sample a possible tokenization according to their probabilities. We use the viterbi algorithm to figure out the best path.
    - Sentence Piece:

Encodings:
- ASCII: 128 characters
- UTF-8: Backward compatible with ASCII. Uses 1-4 bytes for each character.
- UTF-16: Uses 2-4 bytes for each character.

Mar 20, 2025

Reward Hacking in Reinforcement Learning

What are some of the common reward hacking behaviors ?
- It outputs the possibility of an answer. Rubric based reward model. Model often that not model generates a table with rubrics.
- Accuracy based reward models are toughest to hack. Bug in math parsing. Boxed pad because the grader fails.
- Using autorater to check. It starts looking for latex formatting.

Mar 19, 2025

Pre-training Data:
- Sources of Data:
    - C4: FIltered Common Crawl
    - LLAMA: Common Crawl, C4, Github, arXiV, Wikipedia, Books, Stack Exchange
    - RedPajama: Common Crawl, C4, Github, arXiV, Wikipedia, Books, Stack Exchange, Web (Common Crawl)
    - Pile: PileCC, PubMed, Books3, OpenWebText2, ArXiv, Github, FreeLaw, Stack Exchange, US Patent and Trademark Office, Gutenberg, OpenSubtitles, HackerNews, BookCorpus2,, DM mathematics, OpenWebMath, EuroParl & others
    - GPT-2: WebText formed using scraping all outbound links from Reddit.
- Preprocessing:
    - C4: Retain lines that end in terminal punctuation mark. Discard any page that contains less than 3 lines and discard any lines that contain less than 5 words. Removing Dirty, Naughty words, Remove Javascript, Remove lorem ipsum, Removing pages with curly brackets, Remove

citation markers, Remove lines containing strings like terms of use, use of cookies, privacy policy.
    - LLama: For code, remove files with avg line longer than 100 characters and max line length longer than 1000 characters containing less than 25% alphanumeric characters. In WIkipedia, remove hyperlinks, comments and formatting boilerplate. In Arxiv, remove everything before the first section and bibliography. For stack exchange, remove HTML tags.
    - Redpajama: For code remove using line length and % of alphanumeric characters heuristics, File should be in a set of whitelisted extensions. Remove hyperlinks, comments and other boilerplate in wikipedia. Remove HTML tags for Stack Exchange.
    - Pile: Instead of using .wet files, they use raw HTML files and extract them using justext.
- Deduplication:
    - C4: Discarded all but one of any three-sentence span occurring more than once in the data set.
    - LLama: Deduplication at the line level. For Code, deduplication at file level. For books, deduplication at the book level.
    - Redpajama: Deduplication at snapshot level by CCNet pipeline. File level deduplication for Code. Simhash for deduplication of books. Remove preamble, comments, bibliography in arXiv.
    - DeepSeek-V1:
- Quality Filtering:
    - C4: None
    - LLama: Filters low quality content with an ngram language model. Linear model to classify pages as wikipedia vs non-wikipedia. For Github,
    - Redpajama: Use CCNet to assign “head”, “middle” and “tail” buckets based on the perplexity assigned by a 5-gram Kneser-Ney model trained on Wikipedia. Trained a unigram classifier using fastext to distinguish wikipedia vs non-wikipedia. For RedPajama V2, keep all head middle and tail buckets. Quality signals in Redpajama V2:
        - Natural Language: Fraction of all caps words or letters. Fraction of lines that end with ellipsis, fraction of unique words, whether or not a sentence ends in a terminal punctuation mark.
        - Repetitiveness: Fraction of characters appearing in most frequent n-gram (n=2,3,4), Fraction of characters appearing in any duplicated n-gram (n=5…..10).
        - Content-based: Number of sequences of words that are contained in a blocklist. Remove domains in the list of blocked URLs. Embedding Clusters is also a promising direction.
        - ML Heuristics: Fastext classifier based on unigram bag of words to classify content as wikipedia vs non-wikipedia.
        - Deduplication: Include Minhash signatures for fuzzy dedup. IDs of documents found to be exact duplicates using bloom filter.
    - Pile: Net upvotes on Reddit submissions as a proxy for outgoing link quality.
    - GPT-3: Common crawl filtering by training a logistic regression classifier using Wikipedia, OpenWebText and Books as positive examples and unfiltered common crawl as negative examples.
- Language Filtering:
    - C4: Remove pages that were not classified as English with a probability of atleast 0.99 using langdetect.

    - LLama: Perform language identification with a fastText linear classifier to remove non-English pages
    - Redpajama: For redpajama V2, apart from english language, they also keep French, german, Italian and spanish.
- Remixing:
    - LLama 2: Upsampling most factual sources to reduce hallucination.

Key Concepts in Data:

- MinHash: Idea is to reduce dimensionality of a set (sparse vector). You use m hash functions and for each hash function, you go over each element in your set and get a minimum hash value. You do this m times for m hash functions and obtain a m dimensional representation. Then the number of positions the two m dimensional vectors are equal to each other estimates Jaccard similarity. Hash functions are random permutations of vocab.
- Locality Sensitive Hashing: The Minhash signatures are converted into r rows of b bands and each band is put into buckets using hashing functions. If any of those buckets collide, they are treated as candidate pairs. This algorithm basically reduces the number of pairs for which we need to compute the Jaccard similarity.
- Bloom Filter for Exact match document deduplication.
- Importance Resampling: Train 2 unigram/bigram language models on target text and raw text and measure the log of ratio of probabilities assigned to target text using 2 models.

Mar 18, 2025

LLM as a Judge

- GPT-Score: The key idea is that higher-quality text should have a higher likelihood of being generated given a specific evaluation context. GPTSCORE is defined as the following conditional probability:

$$\text{GPTScore}(\boldsymbol{h}|d, a, \mathcal{S}) = \sum_{t=1}^{m} w_t \log p(h_t|\boldsymbol{h}_{<t}, T(d, a, \mathcal{S}), \theta)$$

  where wt is the weight of the token at position t. In our work, we treat each token equally.
- AlpacaEval: For each instruction, both a baseline model b (currently GPT-4 turbo) and the evaluated model m produce responses. A GPT-4 turbo-based evaluator then compares the responses head-to-head and outputs the probability of preferring the evaluated model. A win rate is then computed as the expected probability that the auto evaluator prefers the evaluated model's output on the 805 instructions.
    - Length-Controlled AlpacaEval: We can fit a logistic regression model to predict win rate using factors such as model_identiy, length difference and instruction difficulty. We can then set the coefficient of the length difference in this logistic regression to zero and get a debiased win rate estimation.
- G-Eval: We first input Task Introduction and Evaluation Criteria to the LLM, and ask it to generate a CoT of detailed Evaluation Steps. Then we use the prompt along with the generated CoT to evaluate the

NLG outputs in a form-filling paradigm. Finally, we use the probability-weighted summation of the output scores as the final score.

- Self Taught Evaluators: Using prompt x, generate a winning and a losing response. To generate a losing response use LLMs to slightly modify the prompt x to create x' and take the model response for x' as the losing response. (This is done simultaneously using one common template). Once we have x,y_w and y_l, generate judgements for which response is better using LLM as a Judge iteration i. If the response is correct, we add it to the training dataset for the next iteration i+1. In addition to producing the rating, we also produce a reasoning for why that rating is given.
- BERTScore:

Good Resources:

- https://www.evidentlyai.com/llm-guide/llm-as-a-judge
- https://www.confident-ai.com/blog/llm-evaluation-metrics-everything-you-need-for-llm-evaluation
- https://arxiv.org/pdf/2411.15594
- https://arxiv.org/pdf/2412.05579v1
- https://arxiv.org/pdf/2404.04475
- https://www.reddit.com/r/LocalLLaMA/comments/1afu08t/exploring_the_limitations_of_llmsasajudge/

Mar 17, 2025

Safety:

- Aspects of Safety:
  - Bias: Gender, Religion, Race, Sexual Orientation, Age, Nationality, Disability, Physical Appearance, Socioeconomic Status.
  - Toxic or Offensive Comments: Insults, Hate Speech, Threat, Bullying, Harassment, Sexually Arousing, Violent Behavior.
  - Dangerous Capabilities: Cybersecurity, Chemical, Biological, Radiological and Nuclear Risks, Medical Advice, Mental Health Advice, Persuasion, Manipulation, Deception, Weapons Acquisition, Political Opinions.
- Pretraining:
  - We apply filters to identify websites that likely contain personally identifiable information. We don't filter aggressively as models trained from less aggressively filtered pre-training data also require fewer examples to achieve reasonable safety-alignment.
- Data:
  - Prompts:
    - Human Written Prompts: For each risk category. We collect human-written prompts that are adversarial or borderline in nature. Adversarial prompts range from straightforward ones that directly elicit a harmful response to ones that incorporate sophisticated jailbreaking techniques. In addition to collecting adversarial prompts, we also gather a set of similar prompts, which we refer to as borderline prompts. These are closely related to the adversarial prompts but with a goal to teach the model to learn to provide helpful responses.

        - Synthetic Prompts: Beyond we also leverage synthetic data to improve the quality and coverage of our training datasets. using in-context learning with carefully crafted system prompts, Rainbow Teaming or Red-teaming using LLMs.
    - Responses:
        - Human Annotated Responses: Our annotation teams are instructed to meticulously craft responses to safety prompts based on our guidelines.
        - Synthetic Responses: Use an LLM to generate responses but only keep prompt response pairs which do not violate any policy, using LLM as a judge. This is also known as rejection sampling.
        - **Safety Context Distillation**: This involves generating safer model responses by prefixing a prompt with a safety preprompt, e.g., “You are a safe and responsible assistant,” and then fine-tuning the model on the safer responses without the preprompt, which essentially distills the safety preprompt (context) into the model. Context distillation increases the RM score significantly for samples that initially have a low score, but can also have a detrimental effect on samples that initially have a high score, producing vague responses. We therefore only apply context distillation on targeted samples when it increases RM score.
- Alignment:
    - Finetuning: SFT is highly effective in aligning the model when we strategically balance the ratio of adversarial to borderline examples. We put the focus on more challenging risk areas, with a higher ratio of borderline examples.
    - DPO / RLHF: Responses in preferences data are generated in a way such that they are orthogonal in embedding space.
- Special Cases:
    - Multilingual Safety: We use a combination of prompts written by native speakers, sometimes supplementing with translations from our English benchmarks.
    - Long Context Safety: Long-context models are vulnerable to many-shot jailbreaking attacks without targeted mitigation. To address this, we finetune our models on SFT datasets that include examples of safe behavior in the presence of demonstrations of unsafe behavior in context.
- Uplift Testing:
    - Cyberattacks: Novice and expert attackers were asked to carry out cyberattacks with access to open internet but with or without an LLM. The uplift was insignificant.
    - Chemical and Biological Weapons: Teams of two participants were asked to generate fictitious operational plans for either a biological or chemical attack with access to open internet but with or without an LLM (enabled with websearch, RAG and code execution). Each plan is evaluated across four stages of potential attacks (agent acquisition, production, weaponization, and delivery), generating scores for metrics such as scientific accuracy, detail, detection avoidance, and probability of success in scientific and operational execution.
- Eval Metrics: We measure violation rate on adversarial prompts. In addition, we also keep track of false refusal rates on borderline prompts. False refusal occurs when a model refuses to answer in a helpful way even when a plausible, safe response is possible.
- Evaluation Benchmarks:
    - ToxiGen:

    - XSTest:
    - CrowS Pairs
    - Winogender
    - RealToxicityPrompts
    - Winobias
    - BBQ
    - CyberSecEval:
        - Insecure Coding
        - Cyber attack Helpfulness
        - Prompt Injection
        - Code Interpreter Abuse
        - Vulnerability identification
    - Spear phishing benchmark: We evaluate model persuasiveness and success rate in carrying out personalized conversations designed to deceive a target into unwittingly participating in security compromises. LLAMA model serves as a victim as well as the attacker.
        - Malware Download
        - Security info Gathering
        - Data Theft
        - Credentials Theft
- Red Teaming with Human in the Loop:
    - Short and long-context English:Multi-turn refusal suppression, Hypothetical scenarios, Personas and Role Play, Gradually escalating violation, Adding disclaimers and warnings.
    - Multilingual: Mixing multiple languages in one prompt or conversation, Using Lower resource languages. Slang, specific context or cultural-specific references
    - Tool Use: Unsafe tool chaining, Forcing tool use, Modifying tool use parameters.
- Red Teaming with LLMs:
    - Rainbow Teaming: Builds on Quality Diversity Search algorithm called MAP Elites. In MAP Elites there are two functions, f and d and a grid of solutions. We fill the grid with random solutions. We sample one solution x, apply a mutation to get x', then we fill d(x') and f(x') accordingly. At each iteration of Rainbow Teaming, we sample 1) an adversarial prompt x from the archive with descriptor z, and 2) a descriptor z ′ for the new candidate prompt to be generated. Note that z and z ′ are different.2 We provide x and z ′ to the Mutator LLM to generate a new candidate prompt x ′ with descriptor z ′ . We then feed x ′ to the Target to generate a response. Finally, we ask a Judge LLM to compare the effectiveness of the candidate prompt x ′ to that of the archive's elite prompt – the prompt. stored in the archive with a descriptor z ′ . This comparison focuses on the criteria of interest, such as the toxicity of the Target response, to determine which of the two prompts more effectively meets the adversarial objective.
        - Risk Category: Violence and Hate, Sexual Content, Criminal Planning, Guns and Illegal Weapons, Regulated or Controlled Substances, Self Harm, Inciting or Abetting Discrimination, Frauds and Scams, Cybercrime and Hacking, Terrorism.
        - Attack Style: Slang, Technical Terms, Role Play, Authority Manipulations, Misspellings, Word Play, Emotional Manipulation, Hypotheticals, Historical Scenario, Uncommon Dialects.

- System Level Safety: Often LLMs are not used in isolation but with a safety classifier. which supplements model-level mitigations by providing more flexibility and control. This classifier is used to detect whether input prompts and/or output responses generated by language models violate safety policies on specific categories of harm.
    - Training Data: Utilize the data collected for safety finetuning SFT as negative labels and the rejected samples in preference data as positive labels. We increase the number of unsafe responses in the training set by doing prompt engineering to get the LLM to not refuse responding to adversarial prompts. We use Llama 3 to obtain response labels on such generated data.

Mar 16, 2025

Factuality:

- Pretraining:
    - Factuality Enhanced Language Models for Open-Ended Text Generation: This method consists of i) an addition of a TOPIC PREFIX that improves the awareness of facts during training, and ii) a sentence completion task as the new objective for continued LM training.
    - Unsupervised Improvement of Factual Knowledge in Language Models: Assign higher weight to named entity or factual tokens during pre-training task.
- SFT & RL
    - FLAME: Factuality aware SFT and RL.
        - SFT presents new or unknown information to LLMs which can inadvertently promote hallucination. If we use RAG augmented pretrained model responses to do the SFT, it will increase hallucination compared to using pre-trained model's responses for SFT filtered by FactScore.
            - Classify the prompts into Fact based vs Non-fact based. For non fact based, use human demonstrations, for fact based use pre-trained model responses generated by model in a RAG based few-shot setting.
        - Standard reward used in the RL stage often prefers longer and more detailed responses. . Consequently, a reward-hacking model ends up with a tendency to produce longer claims with more non-factual information. DPO finetuned with RAG augmented pretrained model responses as +ve examples promotes hallucination. If we use FactScore as reward model and select +ve & -ve examples from the model's own generated responses, that fares better.
            - In DPO generate multiple responses from SFT policy and employ 2 reward models RM_IF and RM_Fact to get instruction following and factuality rewards. RM_IF simply deploys an SFT model to give a reward, RM_Fact is built with retrieval augmentation.
    - Fine-tuning Language Models for Factuality: DPO on preference data collected by model confidence estimate or comparing with Wikipedia.
        - Reference based Truthfulness Estimation: FactScore as a representative method of reference based truthfulness scoring. To evaluate a piece of text, FactScore first extracts a list of the atomic claims present in the text using GPT-3.5. For each atomic claim, a smaller, more efficient model such as a Llama-1-7B model that has been fine-tuned for

fact-checking is then used to perform natural language inference to determine if a claim is supported by the reference text.

        - Reference-Free Confidence-Based Truthfulness Estimation: To eliminate the need for external knowledge, we leverage the fact that large language models are wellcalibrated. that is, a large language model's confidence in a generated answer is highly correlated with the probability that the answer is correct. More concretely, we first extract atomic claims from the text using GPT-3.5. We then use GPT-3.5 to convert each claim to a question testing knowledge of the particular fact. Careful rephrasing is necessary to ensure that the rephrased question is unambiguous. After each claim is converted to a minimally ambiguous question, we resample an answer 20 times, typically from the base model (e.g. Llama-1-7B) that is fine-tuned, to estimate the model's uncertainty over the answer. We use a few-shot prompt to encourage well-formed answers. We bin these answers by equivalence, using either heuristic string matching of the responses or using GPT-3.5 to assess if the answers are semantically equivalent. The fraction of responses falling into the largest bin is the final truthfulness score used for the fact, essentially representing the model's confidence for this fact.
        - Evaluation using Human Evaluators and GPT-4 outside of FactScore.
- Inference/Decoding:
    - Factual-nucleus sampling algorithm: We hypothesize that the randomness of sampling is more harmful to factuality when it is used to generate the latter part of a sentence than the beginning of a sentence. There is no preceding text at the start of a sentence, so it is safe for LM to generate anything as long as it is grammatical and contextual. However, as the generation proceeds, the premise become more determined, and fewer word choices can make the sentence factual. Given the example "Samuel Witwer's father is a Lutheran minister", the beginning of the sentence "Samuel Witwer's father is" is not nonfactual. However, the continuation of "Lutheran minister" makes the sentence nonfactual. Therefore, we introduce the factual-nucleus sampling algorithm that dynamically adapts the "nucleus" p along the generation of each sentence. In factual-nucleus sampling, the nucleus probability pt to generate the t-th token within each sentence is

$$pt = \max\{\omega, p \times \lambda\ t-1\}$$

        - Specifically, it has the following parts:
            - λ-decay: Given that top-p sampling pool is selected as a set of subwords whose cumulative probability exceeds p, we gradually decay the p value with decay factor λ at each generation step to reduce the "randomness" through time.
            - p-reset: The nucleus probability p can quickly decay to a small value after a long generation. So, we reset the p-value to the default value at the beginning of every new sentence in the generation (we identify the beginning of a new sentence by checking if the previous step has generated a full-stop). This reduces the unnecessary cost of diversity for any long generations.
            - ω-bound: If λ-decay is applied alone, the p-value could become too small to be equivalent to greedy decoding and hurt diversity. To overcome this, we introduce a lower-bound ω to limit how far p-value can be decayed
    - DOLA: Factual knowledge in an LLMs has generally been shown to be localized to particular transformer layers. Apply vocabulary head directly to the hidden representation of the middle

layers. It works because of residual connections, transformer layers make the hidden representations gradually evolve without abrupt changes. We calculate 2 sets of softmax, apply a contrastive function and then recompute softmax to get token probabilities..
        - Repetition Penalty
- Evaluation
    - FactScore: It breaks down a generation into a series of atomic facts and computes the percentage of atomic facts supported by a reliable knowledge source such as Wikipedia. FACTSCORE considers precision but not recall, e.g., a model that abstains from answering too often or generates text with fewer facts may have a higher FACTSCORE, even if these are not desired.
    - TruthfulQA: 817 questions covering 38 categories such as health, law, fiction. The questions are crafted in such a way that some humans might answer them incorrectly due to false belief or misconception.
    - SimpleQA:

Mar 15, 2025

Eval Data Contamination:
- Hash-Based Matching (Exact Match)
- nGram Overlap
- Embedding Similarity Search (Semantic Matching)
- Lexical Similarity (Jaccard / BLEU / ROUGE)
- Training Loss/Log-Probability Skew Detection
- How to Respond to Detected Contamination
    - Remove the contaminated eval samples or training examples. Retrain/fine-tune without the contaminated data.
    - Report contaminated performance separately and conduct evaluation on a **decontaminated version** of the benchmark

Mar 14, 2025

DeepSeekMath:
- Continued pre-training on DeepSeekCoder using 120B math tokens from common crawl.
- DeepSeekMath Corpus: A large-scale high-quality pre-training corpus comprising 120B math tokens. This dataset is extracted from the Common Crawl (CC) using a fastText-based classifier. The classifier is trained using 500,000 instances from OpenWebMath as positive examples, while incorporating a diverse selection of other web pages, 500,000 instances in total to serve as negative examples. We apply deduplication and near deduplication to obtain 40B documents in common crawl and apply the fastext classifier on these documents to get 40B tokens. After the first iteration of data collection, numerous mathematical web pages remain uncollected, mainly because the fastText model is trained on a set of positive examples that lacks sufficient diversity. We therefore find domains with >10% of webpages classified as math content. Manually annotate the URLs associated with mathematical

content within these identified domains. Web pages linked to these domains, yet uncollected, will be added to the seed corpus. This approach enables us to gather more positive examples, thereby training an improved fastText model capable of recalling more mathematical data in the subsequent iteration.

  - Decontamination: Any text segment containing a 10-gram string that matches exactly with any sub-string from the evaluation benchmarks is removed from our math training corpus.

- Instruction tuning: Mathematical instruction tuning to DeepSeekMath-Base. Our English collection covers diverse fields of mathematics, e.g., algebra, probability, number theory, calculus, and geometry. The total number of training examples is 776K. Problems are paired with solutions in chain-of-thought (CoT), program-of-thought (PoT) and tool integrated reasoning format.
  - Chain-of-thought
  - Program of thought: Models are prompted to solve each problem by writing a Python program where libraries such as math and sympy can be utilized for intricate computations.
  - Tool-integrated reasoning: This format interleaves rationales with program-based tool use. where the model creates rationale r1 for analysis, writes program a1 to call an external solver, obtains the execution output o1, and then generates rationale r2 to finalize the answer
- Reinforcement Learning: Group relative policy optimization. Major change from PPO is that they don't use the value network, instead they take N samples and use the mean reward of the samples to calculate the value function and subtract it from the reward for advantage estimate. (They also divide it by std). Instead of using KL penalty on reward, they add a KL penalty in the loss term.
  - Online vs Offline Training: Data sources can be divided into two categories: online sampling, and offline sampling. Online sampling denotes that the training data is from the exploration results of the real-time training policy model, while offline sampling denotes that the training data is from the sampling results of the initial SFT model. RFT and DPO follow the offline style, while Online RFT, PPO and GRPO follow the online style.
  - Outcome vs Process Supervision: Outcome supervision only provides a reward at the end of each output, which may not be sufficient and efficient to supervise the policy in complex mathematical tasks. Process supervision provides a reward at the end of each reasoning step. Process supervision calculates the advantage of each token as the sum of the normalized rewards from the following steps.
  - Single turn vs Iterative RL: In iterative RL, we generate new training sets for the reward model based on the sampling results from the policy model and continually train the old reward model using a replay mechanism that incorporates 10% of historical data. Then, we set the reference model as the policy model, and continually train the policy model with the new reward model.
  - GRPO vs Online RFT: GRPO uniquely adjusts its gradient coefficient based on the reward value provided by the reward model. This allows for differential reinforcement and penalization of responses according to their varying magnitudes. In contrast, Online RFT lacks this feature; it does not penalize incorrect responses and uniformly reinforces all responses with correct answers at the same level of intensity.
- Lessons Learnt:
  - Code training benefits Mathematical Reasoning.
  - RL enhances Maj@K's performance but not Pass@K. RL enhances the model's overall performance by rendering the output distribution more robust, in other words, it seems that the improvement is attributed to boosting the correct response from TopK rather than the enhancement of fundamental capabilities.

    - We should explore reinforcement learning algorithms that are robust against noisy reward signals. The uncertainty quantification in reward models could potentially act as a linking bridge between the weak reward model and the weak-to-strong learning algorithms.

LLAMA3 Math & Reasoning:

- Lack of prompts: Source relevant pre-training data from mathematical contexts and converted it into a question-answer format which can then be used for supervised finetuning. We create a taxonomy of mathematical skills to boost diversity.
- Lack of ground truth chain of thought: Model generated chain of thoughts filtered by what leads to a correct answer.
    - Incorrect intermediate steps in CoT: Use process reward models to filter data where intermediate steps are incorrect.
- Teaching models to use external tools and interleaving code and text: We prompt Llama 3 to solve reasoning problems through a combination of textual reasoning and associated Python code. Code execution is used as a feedback signal to eliminate cases where the reasoning chain was not valid, ensuring the correctness of the reasoning process.
- Discrepancy between training and inference where during inference the model may be expected to use feedback to improve its reasoning: To simulate human feedback, we utilize incorrect generations (i.e., generations leading to incorrect reasoning traces) and perform error correction by prompting Llama 3 to yield correct generations. The iterative process of using feedback from incorrect attempts and correcting them helps improve the model's ability to reason accurately and learn from its mistakes.

Evaluation Benchmarks:

- AIME

Sources of Data:

- RL Data:
    - Math
    - AIME
    - US High School contests
    - Math League
    - Olympiad Bench
    - AtCoder
    - Codeforces
    - Puzzles
    - IMO Questions
    - AlphaCode
    - TopCoder
    - AOPS
- SFT Data:

New Approaches:

- Best of N Re-ranking: Model a reranking function that selects or aggregates the final answer given N model-generated attempts.
- Process Reward Models: Use reward models that rate each step of the thinking process rather than the final accuracy based reward.

Mar 13, 2025

Codex:

- Pre-training Data: 100 billion tokens (54 million public software repositories hosted on GitHub, containing 179 GB of unique Python files under 1 MB).
    - Filter out average line length > 100 and max line length > 1000 or if the % of alphanumeric characters is less.
- SFT Data:
    - 10000 problems from competitive programming and interview preparation websites, using problem description as docstring. Unit tests obtained by submitting incorrect solutions.
    - Find a project using Travis CI and Tox. Run integration tests in a sandbox while tracing function calls. Extract function inputs, outputs, and docstrings. Create a programming problem using the function's docstring as the prompt. Use the traced function as the response and additionally record test cases.
- Filtering SFT Data: Sample 100 responses using the pre-trained model, if none of the samples pass the unit tests, filter out those examples.
- Vocab: GPT-3 tokenizer is not very efficient at encoding whitespaces. So we add an additional set of tokens for representing whitespace runs of different lengths. This allows us to represent code using approximately 30% fewer tokens.
- Eval: On human eval, the pre-trained / tail-patched model has 28.8% accuracy but SFTed model with prompt response pairs has 37.7% accuracy.
    - pass @ k : k code samples are generated per problem, a problem is considered solved if any sample. However, computing pass@k in this way can have high variance. . Instead, to evaluate pass@k, we generate n ≥ k samples per task (in this paper, we use n = 200 and k ≤ 100), count the number of correct samples c ≤ n which pass unit tests, and calculate the unbiased estimator.

$$\text{pass@}k := \mathop{\mathbb{E}}_{\text{Problems}} \left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}} \right]$$

    - Filtered pass @ k: Generate 1000 solutions, filter those that pass 3 units tests given with the competition problems. Sample k solutions from them and then check if any of them passes all unit tests.
    - Higher temperatures are optimal for larger k, because the resulting set of samples has higher diversity, and the metric rewards only whether the model generates any correct solution.
    - When we have to choose 1 out of the k samples, mean logp performs better than sum logp.
- Limitations:
    - Doesn't follow chained instructions when the docstring gets longer.

CodeLLama:

- Data:
    - Code Llama: 500B tokens + 20B tokens for long context
    - Code Llama - Instruct: Code Llama + 5B tokens for instruction following
        - Data from manual annotation in SFT and rejection sampling from RLHF v5. However collecting such data for code is expensive as it requires feedback from professional developers.
        - Data collected with execution feedback: Generate interview style programming questions by few-shot prompting the LLAMA 2 model. Deduplicate. Generate unit tests and 10 python solutions by prompting. Execute unit tests on solutions and add the first solution that passes the unit tests to the dataset.
    - Code Llama - Python: 500B tokens + 100B python tokens + 20B long context tokens.
- Modeling:
    - Code Infilling: They use a causally masked language modeling objective where they introduce mask token where the masked tokens are and move the masked tokens at the end and predict them autoregressively.
    - Long Context: Increase context length from 4k to 16k by increasing the base period θ from 10,000 to 1,000,000 for fine-tuning.
- Eval:
    - Code generation
        - HumanEval: 164 programming problems with 7.7 unit tests per problem on an average.
        - MBPP: 1,000 crowd-sourced Python programming problems, consists of a task description, code solution and 3 automated test cases. Test cases are given along with the description in the prompt.
        - MultiPL-E benchmark: HumanEval and MBPP translated to 18 other programming languages. MultiPL-E translates unit tests, doctests, Python-specific terminology, and type annotations.
    - Long Context Eval:
        - Needle in Haystack: Set the value of some scalar in the code and ask the model to generate the assert statement checking the value of that scalar.

LLAMA 3:

- Pretraining Data: Target webpages with code interleaved with natural language. Customized heuristics for filtering.
- Expert Training: Branch the main pre-training run and train it on additional 1T coding tokens. This model is SFTed and DPOed on code specific prompts. This expert is then used to collect data for SFT via rejection sampling.
- Synthetic data via Execution Feedback
    - Problem description generation: To achieve this diversity, we sample random code snippets from various sources and prompt the model to generate programming problems inspired by these examples.

    - Solution Generation: Then, we prompt Llama 3 to solve each problem in a given programming language. We observe that adding general rules of good programming to the prompt improves the generated solution quality. Also, we find it is helpful to require the model to explain its thought process in comments.
    - Correctness Analysis:
        - Unit test generation: For each problem and solution, we prompt the model to generate unit tests.
        - Static analysis: Run code through parser and linter to ensure syntactic correctness.
    - Error feedback and iterative self-correction: When a solution fails at any step, we prompt the model to revise it. The prompt included the original problem description, the faulty solution, and feedback from the parser/linter/tester (stdout, stderr/ and return code). After a unit test execution failure, the model could either fix the code to pass the existing tests or modify its unit tests to accommodate the generated code.
- Synthetic data generation: programming language translation: Prompt the LLAMA 3 model to translate the code from one language to another and check for correctness using syntax parsing, compilation, and execution.
- Synthetic data generation: Backtranslation: Generate comments and docstrings for code snippet. Using those, attempt to produce original code. Using the original code as a reference, we prompt the Llama 3 to determine the quality of the output (e.g., we ask the model how faithful the backtranslated code is to the original). We then use the generated examples that have the highest self-verification scores in SFT.
- System prompt steering: We used code specific system prompts to improve code readability, documentation, thoroughness, and specificity.
- Filtering training data with execution and model-as-judge signals. Model as a judge is required when the code itself is interleaved with text and isn't executable.

DeepSeek-Coder:

- Data:
    - 2T tokens: 90% source code, 10% code related english data from Github Markdown and stack exchange.
    - Filtering approaches:
        - Filter out files with an average line length exceeding 100 characters or a maximum line length surpassing 1000 characters.
        - Remove files with fewer than 25% alphabetic characters.
        - For the XSLT programming language, we further filter out files where the string "<?xml version=" appeared in the first 100 characters.
        - For HTML files, We retain files where the visible text constitutes at least 20% of the code and is no less than 100 characters.
        - For JSON and YAML files, which typically contain more data, we only keep files that have a character count ranging from 50 to 5000 characters. This effectively removes most data-heavy files.
    - First attempt to incorporate repository-level data construction during the pre-training phase of our models. Topological sort for Dependency Parsing and then arrange these files in an order that ensures the context each file relies on is placed before that file in the input sequence. To

incorporate file path information, a comment indicating the file's path is added at the beginning of each file. This method ensures that the path information is preserved in the training data.

- Repo-Level Deduplication instead of file level dedup to avoid disrupting the structure of the repository: duplication is even more present in code datasets, since it is common practice to reuse and clone code repositories of others.
    - Near-deduplication: We implement near-deduplication in our pre-processing pipeline on top of exact deduplication. We first split the files into words/tokens based on non-alphanumeric characters and remove files with fewer than 10 tokens. Next, we compute the MinHash (Broder, 2000; Lee et al., 2021) with 256 permutations of all documents, and use Locality Sensitive Hashing (Har-Peled et al., 2012) to find clusters of duplicates. We further reduce these clusters by ensuring that each file in the original cluster is similar to at least one other file in the reduced cluster. We consider two files similar when their Jaccard similarity exceeds 0.85.
- Quality screening and decontamination: we also employ a compiler and a quality model, combined with heuristic rules, to further filter out low-quality data. This includes code with syntax errors, poor readability, and low modularity. To prevent test data contamination, we use 10-gram filtering approach.

- Modeling:
    - Next token prediction objective
    - Fill in Middle objective
- Evaluation:
    - Leetcode Contest Benchmark: 180 Leetcode daily questions from Jul 2023 to Jan 2024.
    - Math datasets: AIME, MATH, GSM8k etc.

DeepSeek-Coder-V2:

- Data: 6T tokens. 338 programming languages.
    - To collect code-related and math-related web texts from Common Crawl, , we select coding forums such as StackOverflow , library sites such as PyTorch documentation , and mathematics website such as StackExchange as our initial seed corpus. Using this seed corpus, we train a fastText model to recall more coding-related and math-related web pages.
- Modeling:
    - Long Context: 128k using Yarn
- Instruction tuning:
    - SFT Data: 20k examples collected using DeepSeek Coder and 30k examples using DeepSeek Math. 300M tokens of general instruction following SFT data.
    - RL Data: 40k prompts with test cases.
    - Reward Model: Trained on compiler signal. Has better generalization ability, in comparison with raw compiler signal.
- Eval:
    - RepoBench: This dataset is constructed from a diverse set of real-world, open-sourced, permissively licensed repositories in two popular programming languages: Python and Java.
    - Code Fixing:

        - SWEBench: SWE-bench is a comprehensive benchmark designed to evaluate the performance of large language models in addressing real-world software issues sourced from GitHub. The benchmark presents a codebase alongside a specific issue, challenging the language model to generate a patch that effectively resolves the described problem.
        - Aider: Aider's code editing benchmark evaluates the LLM's ability to modify Python source files, completing 133 distinct coding tasks. This benchmark not only tests the LLM's coding skills but also checks its consistency in producing code edits according to the specifications in the prompt.
    - CodeUnderstanding and Reasoning: Given a function and an input predict the output. Or Given a function and an output predict the input.

Survey Paper: https://arxiv.org/pdf/2406.00515

Mar 12, 2025

Toolformer:

- Data: Given a dataset of plain text, we convert this dataset into a dataset augmented with API calls. We first exploit the in-context learning capability of the model to sample a large number of potential API calls. We execute these API calls and finally check whether the obtained responses are helpful for predicting future tokens and use that as a filtering criteria.
    - During sampling, we sample upto k positions for doing API calls by predicting the probability of API token. For each of these k positions, we sample m API calls.
    - For each API call, We calculate the log probabilities of the tokens generated after the response to the API call and we compare it to the log probabilities of the tokens generated if there was no API call or at or if there was no response to the API call. If the former is higher than the latter by a specified threshold, we keep that data point.
    - Finally, we merge these API calls and interleave them with original inputs. That is, for an input text x = x1, . . . , xn with a corresponding API call and result (ci , ri) at position i, we construct the new sequence x* = x1:i−1, e(ci , ri), xi:n
- Modeling: The data is modeling using the usual next token prediction loss. A key insight here is that the model is learning what API calls to make and how to use the answers of API call in the final response at the same time.
- Inference: When generating text with M after finetuning with our approach, we perform regular decoding until M produces the "→" token, indicating that it next expects the response for an API call. At this point, we interrupt the decoding process, call the appropriate API to get a response, and continue the decoding process after inserting both the response and the token. We let the model start an API call not just when is the most likely token, but whenever it is one of the k most likely tokens.
- Tools: Atlas Question Answering, Calculator, Wikipedia Search, Machine Translation System, Calendar.
- Evals: Evaluated in a zero-shot setting without demonstration of how to use tools.
    - LAMA: SQuAD, GoogleRE and T-REx subsets. These examples were constructed to evaluate Masked language models and therefore have a mask token. Only the examples were mask

token is the last token are considered. For each of these subsets, the task is to complete a short statement with a missing fact (e.g., a date or a place). Toolformer uses QA tool for 98.5 % of these examples.
    - Math: ASDiv, SVAMP and MAWPS: These are all datasets that contain simple math word problems answer to which are all numbers. 98% of the time, toolformer decides to use calculator.
    - Question Answering: WebQuestions, Natural Questions, TriviaQA. QA tool is disabled. Toolformer relies on wikipedia search 99.3% of the times.
    - Multilingual: MLQA where the paragraph is in english but the answer is in a different language. Mixed results on these because due to finetuning on CCNet the model might be losing on multilingual capability.
    - Language Modeling: Perplexity is not increased due to toolformers (with tools disabled). This ensures that language modeling performance of Toolformer does not degrade through our finetuning with API calls.
- Interesting Questions:
    - How do you disable tools in toolformer ? By manually setting the probability of API token to 0. To disable individual tools, you'll set the probability of the next token after API token to be 0 for that particular tool.
    - What about the noise in the data ? However, some amount of noise in the API calls that are not filtered can actually be useful as it forces the model finetuned on C* to not always blindly follow the results of each call it makes.
- Limitations: Since for each tool, API calls are sampled separately, , there are no examples of chained tool use in the finetuning dataset. It does not allow the LM to use a tool in an interactive way. Toolformer currently does not take into account the tool-dependent, computational cost incurred from making an API call.

LLama 3:
- Tools: Search Engine, Python Interpreter, Mathematical Computation Engine (Wolfram API).
- Data:
- Evals:

ToolUse Evals:
- BFCL
- Nexus
- TAU-bench: retail and airline

Survey Paper: https://arxiv.org/pdf/2405.17935

Mar 11, 2025

SLIC:
- Additional stage of Sequence Likelihood Calibration following pre-training and finetuning. This stage involves decoding k samples from the SFT model and employ 2 loss functions: calibration loss and regularization loss. The calibration loss aligns the sequence likelihood of the generated sequences

according to their similarity with target answer and regularization loss ensures that the model weights do not go too far away from SFT weights. Calibration loss can be rank based, margin based, list-wise rank based or expected reward loss.
- SLIC-HF:

IPO:

- We consider a general non-decreasing function Ψ : [0, 1] → R, a reference policy πref ∈ ∆X Y , and a real positive regularisation parameter τ ∈ R ∗ +, and define the Ψ-preference optimisation objective (ΨPO):

$$\max_{\pi} \mathop{\mathbb{E}}_{\substack{x\sim\rho \\ y\sim\pi(.|x) \\ y'\sim\mu(.|x)}} [\Psi(p^*(y \succ y'|x))] - \tau D_{\mathrm{KL}}(\pi \mid\mid \pi_{\mathrm{ref}})\,.$$

- When, Ψ(q) = log(q/(1 − q)), this objective function reduces to the RLHF objective. And DPO already shows that the objective function of DPO and RLHF objective are similar. This for Ψ(q) = log(q/(1 − q)), ΨPO, DPO and RLHF all have identical objective function. [Proof]
- Closed form solution of DPO and RLHF can be written as:

$$\pi^*(y) \propto \pi_{\mathrm{ref}}(y) \exp\left(\tau^{-1}\mathbb{E}_{y'\sim\mu}[\Psi(p^*(y \succ y'))]\right)$$

- The highly non-linear nature and unboundedness of Ψ means that small increases in preference probabilities already close to 1 are just as incentivized as larger increases in preference probabilities around 50%, which may be undesirable.
- If p* (y ≻ y ′) = 1, , π* (y ′) = 0 irrespective of KL term. Strength of the KL-regularisation becomes weaker and weaker the more deterministic the preferences. The weakness of the KL-regularisation becomes even more pronounced in the finite data regime, where we only have access to a sample estimate of the preference. **This means that overfitting can be a substantial empirical issue**, especially when the context and action spaces are extremely large as it is for large language models.
- Standard RLHF is more robust to this problem. In practice when empirical preference probabilities are in the set {0, 1}, the reward function ends up being underfit. The optimal rewards in the presence of {0, 1} preference probabilities are infinite, but these values are avoided. DPO, in avoiding the training of the reward function, loses the regularisation of the policy that the underfitted reward function affords.
- This motivates choices of Ψ which are bounded, ensuring that the KL regularisation remains effective even in the regime of {0, 1} valued preferences, as it is often the case when working with empirical datasets.
- Ψ = Identity function: Loss function becomes

Algorithm 1 Sampled IPO

```
Require: Dataset $\mathcal{D}$ of prompts, preferred and dis-
    preferred generations $x$, $y_w$ and $y_l$, respectively. A
    reference policy $\pi_{\mathrm{ref}}$
1: Define
       $$h_\pi(y, y', x) = \log\left(\frac{\pi(y|x)\pi_{\mathrm{ref}}(y'|x)}{\pi(y'|x)\pi_{\mathrm{ref}}(y|x)}\right)$$
2: Starting from $\pi = \pi_{\mathrm{ref}}$ minimize
       $$\mathop{\mathbb{E}}_{(y_w,y_l,x)\sim D}\left(h_\pi(y_w, y_l, x) - \frac{\tau^{-1}}{2}\right)^2.$$
```

-

DPO:

- They first find the optimal policy which maximizes the reward penalized by KL divergence between old and new policy. Then they express the reward in terms of optimal policy. Then they substitute the reward term in reward model loss (according to bradley terry model) with this new parametrization. Thus the objective function of the reward model is expressed in terms of model parameters which can then be optimized using gradient descent.
- Implementation of DPO is straightforward. Keep a trainable model and a frozen SFT (reference) model. Make a forward pass of both models using chosen sample and rejected sample. You'll have 4 sets of logits. Calculate the sum of logprobs for the sequence using torch.gather on input indices shifted by 1. Calculate logprobs_chosen - logprobs_chosen_ref - (logprobs_rejected - logprobs_rejected_ref) and scale it by a factor of beta. Take logsigmoid and then mean over the entire batch.
    - No grad for reference model.
    - To calculate log probs, get logits upto :-1, shift labels and mask 1:
- To calculate probabilities of the entire sequence, the sum of the log probs makes more sense as it multiplies probabilities of each individual token giving the probability of the entire sequence. One potential issue is that what if the chosen responses are too long and the rejected responses are too short. But this issue is taken care of by the division by reference policy. We can also use a flag which decides whether to take mean or sum.
- DPO showed that when Bradley terry model perfectly fits the preference data and the optimal reward is obtained from loss function

$$\mathcal{L}(r) = -\mathbb{E}_{(x,y_w,y_l)\sim\mathcal{D}}\left[\log\left(p(y_w \succ y_l|x)\right)\right].$$

  then the policy optimizing the RLHF objective and DPO objective perfectly coincide.

- Online implementations:
    - https://github.com/ahmed-alllam/Direct-Preference-Optimization/blob/main/src/train.py
        - No shifting of the inputs while taking log probs.
        - Takes mean instead of sum.
- Theoretical analysis of DPO:

Doubts:

- Direct preference optimization: population form of the loss as discussed in Equation (5) of IPO paper.
- What does estimated $p(y \succ y') = 1$ mean in practice ?

Mar 10, 2025

RL Algorithms:

- Value based vs Policy Based: In value based, we use the network to learn value functions which are the expected values of discounted reward and then we take the action that takes us to a state with maximum value function. If we are using neural network, the output will be the estimated value functions for each action given a state. In a policy based method, instead of learning a value function, the agent directly learns a policy which maps states to action. If we are using a neural network, the output will be probability distribution over actions.
    - Value based: Q-learning
    - Policy based: REINFORCE
- Value Function Learning Strategies:
    - Temporal Difference: After each step, using estimated TD target.
    - Monte Carlo: After each episode using actual discounted rewards.
- Q-learning: Off Policy, Value-based and Temporal Difference Approach. This is considered off policy because we use an epsilon greedy policy for inference but a greedy policy for estimated the TD target. Sarsa is on-policy.
    - Epsilon-greedy strategy: Chooses the action with the highest expected reward with a probability of 1-epsilon. Chooses a random action with a probability of epsilon. Epsilon is typically decreased over time to shift focus towards exploitation.
- DeepQ-learning: We formulate the loss as MSE between the estimated Q value and the estimated TD target. To estimate TD target, we first take the action using epsilon greedy policy and get the Q value of the next state. Then we choose the TD target according to the greedy policy using max of the Q values. To stabilize the training, following approaches are used:
    - Experience Replay: The agent may suffer from catastropic forgetting due to sequential nature of training. It has two phases: sampling and training. During sampling, we perform actions and store the observed experience tuples in a replay memory. During training, Select a small batch of tuples randomly and learn from this batch.
    - Fixed Q-Target: Since we are using the parameters in the network to calculate TD target, at every step of training, both our Q-values and the target values shift. We're getting closer to our target, but the target is also moving. It's like chasing a moving target! This can lead to significant oscillation in training. Instead, Use a separate network with fixed parameters for estimating the TD Target. Copy the parameters from our Deep Q-Network every C steps to update the target network.
    - Double Deep Q-Learning: We don't have enough information about the best action to take at the beginning of the training. Therefore, taking the maximum Q-value (which is noisy) as the best action to take can lead to false positives. Standard DQN tends to overestimate Q-values because it uses the maximum Q-value directly for target calculation. If Q-values are noisy or incorrect, taking the max will overestimate the true value. We use two networks to decouple the action selection from the target Q-value generation. Use our DQN network to select the best action to take for the next state (the action with the highest Q-value). Use our Target network to calculate the target Q-value of taking that action at the next state. This approach reduces the Q-Values overestimation.
- Advantages of Policy based methods over Value based methods:
    - Policy based methods are more effective for continuous action spaces because for continuous action spaces, the argmax over all actions is intractable and discretizing the action space loses precision. Policy-based methods can output continuous actions directly.

    - In the policy based methods, the concept of exploration and exploitation is baked in as we output a probability distribution over actions. We also get rid of the problem of perceptual aliasing. Perceptual aliasing is when two states seem (or are) the same but need different actions.
    - Policy-gradient methods have better convergence properties. In policy-gradient methods, stochastic policy action preferences (probability of taking action) change smoothly over time. Value based methods on the other hand can change dramatically if the maximum value is changed.
- The difference between policy-based and policy-gradient methods:
    - Policy-based methods: We can optimize the parameter θ indirectly by maximizing the local approximation of the objective function.
    - we optimize the parameter θ directly by performing the gradient ascent on the performance of the objective function J(θ).
- Policy Gradient Theorem: It is used to compute the gradient of the expected discounted rewards which we are trying to maximize. We compute the gradient and then derive the loss function for algorithms like REINFORCE such that the gradient of this loss function is same as the gradient of the expected discounted rewards. We use summation for practical implementations like reinforce. The notation without summation is used to denote the gradient at each time step. Both are correct just different ways of expressing the idea.

$$\nabla\theta J(\theta)=E\pi[Gt\nabla\theta\log\pi(at\mid st;\theta)]$$
$$\text{Or }\nabla\theta J(\theta)=E\pi[t=0\sum TGt\nabla\theta\log\pi(at\mid st;\theta)]$$

- Curiosity: Intrinsic reward mechanism which is calculated as the error of our agent in predicting the next state, given the current state and action taken. If the agent spends a lot of time on these states, it will be good at predicting the next state (low Curiosity). On the other hand, if it's in a new, unexplored state, it will be hard to predict the following state (high Curiosity).
- Policy Gradient for continuous action spaces: Estimate the parameters of the probability distribution such as for gaussian estimate mean and standard deviation. Mean can be modeled with a linear layer and standard deviation can be modeled with a linear layer followed by exp to keep it always +ve.
- Variance in REINFORCE: Given the stochasticity of the policy, trajectories can lead to different returns, which can lead to high variance. The solution is to mitigate the variance by using a large number of trajectories. However, increasing the batch size significantly reduces sample efficiency.
- Actor-Critic Methods: Combination of Policy-Based and Value-Based methods. Actor is a policy function controls how our agent acts. Critic is a value function that assists the policy update by measuring how good the action taken is. The loss function for the actor replaces the discounted rewards with estimated q value. The loss function for the critic is MSE loss between the estimated q values and the estimated TD target. The estimated TD target is computed by using the action taken by the actor at time step t+1.
    - Advantage-Actor Critic: Replaces the Value function with an Advantage function which calculates the relative advantage of an action compared to the others possible at a state: how taking that action at a state is better compared to the average value of the state. It's subtracting the mean value of the state from the state action pair. This advantage value essentially comes down to the TD error for the mean value of the state. As such since the mean value of the state is not dependent on action space and can work for continuous action spaces as well.

    - Advantages: Improved Sample Efficiency, Faster Convergence and Works for both discrete and continuous action spaces. Reduces Variance in Policy Gradient methods because Gt was calculated using Monte Carlo estimates using rollout of the entire episode whereas Value function or advantage function is calculated using TD estimate using only 1 step reducing variance.
- Proximal Policy Optimization (PPO): The log probability term is replaced with a ratio of new probabilities and old probabilities. ~~This importance sampling term is introduced to make the algorithm on policy.~~ The gradient function still has a log probability term in it and is multiplied by ratio of new probabilities and old probabilities. Additionally in PPO, we clip the ratio to ensure that we do not have a too large policy update because the current policy can't be too different from the older one. This is a soft constraint baked into the objective function. After clipping, we take the min of the unclipped and clipped objective. This essentially divides the objective function into 6 different regions. When the ratio is above 1+e and the advantage function is +ve, we don't want to get too greedy so we clip the objective function and the gradient is set to zero. When the ratio is below 1-e and the advantage function is +ve, we should encourage our current policy to increase the probability of taking that action in that state. Similarly, when the ratio is below 1-e and the advantage function is negative, we don't want to push down the probabilities too much, so we clip the gradient. But when the ratio is above 1+e and the advantage function is -ve, we want to decrease the probability of taking that action at that state. In the range between 1-e and 1+e, if the advantage function is +ve, we want to increase the probabilities and if the advantage function is -ve, we want to decrease the probabilities. In addition, PPO also adds an entropy bonus to ensure sufficient exploration. In addition to the clipped surrogate loss function, PPO has a MSE loss for value network and an entropy loss that encourages exploration. The value network is initialized from RM.
    - Why take the minimum ? Because, if we don't take the minimum in +ve advantage and ratio < 1-e, the gradient will become zero which we don't want. We want to bring the probabilities up in that case. Minimum gives us 6 nicely defined regions in loss function which makes intuitive sense.
    - To mitigate over-optimization of the reward model, the standard approach is to add a per-token KL penalty from a reference model in the reward at each token (Ouyang et al., 2022), i.e.,

$$rt = r\varphi(q, o{\le}t) - \beta \log \pi\theta(ot \mid q, o{<}t) / \pi ref\,(ot \mid q, o{<}t)\ ,$$

    - PPO-KL: Soft KL penalty in the objective function instead of clipping. β is a penalty coefficient that adjusts the KL regularization. If the KL divergence is too high, β is large to slow down learning and focus more on KL term. If KL divergence is too low, β is decreased to increase exploration. Uses simple first-order optimization.

$$D\_KL(P \,||\, Q) = \sum (P(x) * \log(P(x) / Q(x)))$$

- Trust Region Policy Optimization (TRPO): Similar to PPO, log probability term is replaced with a ratio of new probabilities and old probabilities. Uses a hard KL divergence constraint outside the objective function to constrain the policy update. It is strictly constrained to be below a threshold δ. TRPO uses a second-order optimization method. But this method is complicated to implement and takes more computation time.

$$D_{KL}(\theta) \approx 1/2*(\theta-\theta_{old})^T * H * (\theta-\theta_{old})$$

where H is the **Fisher Information Matrix (FIM)**, which approximates the Hessian of the KL divergence. Why second order methods ? Because of the hard KL constraint. First-order methods do not properly account for the curvature of the KL constraint. We need a method that accounts for the geometry of KL divergence

- Group Relative Policy Optimization (GRPO): The inclusion of the value model in PPO often comparable in size to the policy model, increases memory and computational requirements during training. Do away with the value function. Instead take G samples and use the mean of the G samples to estimate the baseline (or value function). They are also normalized by dividing by the group standard deviation. Similar to PPO, log probability term is replaced with a ratio of new probabilities and old probabilities. They also use clipped objective but in addition to that introduce a soft KL divergence penalty in the loss between old and new policies.

$$D_{KL}\ \pi\theta || \pi ref = \pi ref\,(o_{i,t}\ |q,\ o_{i,<t}) \,/\, \pi\theta(o_{i,t}\ |q,\ o_{i,<t}) - \log\,(\pi ref\,(o_{i,t}\ |q,\ o_{i,<t}) \,/\, \pi\theta(o_{i,t}\ |q,\ o_{i,<t})) - 1$$

  - Process Supervision: process reward model is used to score each step of the outputs, yielding corresponding rewards: R = {{r1 $index$ (1) , · · · ,r1 $index$ (K1) }, · · · , {rG $index$ (1) , · · · ,r $index$G (KG) }}. r_i $index$ (j) = (r_i $index$ (j) −mean(R))/std(R) . It calculates the advantage of each token as the sum of the normalized rewards from the following steps. This is as if the discount factor (gamma) is 1.

- REINFORCE + KL: It is a policy gradient algorithm with a value-function baseline, with annealed per-step exact Kullback-Leibler regularization, and some additional tricks. The objective is to maximize the total reward while not going too far from the original supervised model. It uses a learned value function that is subtracted to the Monte-Carlo estimate of the total reward. It requires 3 sets of weights: the policy, the value, and the anchor policy (the base supervised model or the best policy available, used to compute the KL).
  - KL divergence is computed across all of logits as opposed to classical sampled KL. This reduces variance and improves training stability.
  - By annealing alpha, we allow the policy to move away smoothly from the original policy towards a reward-maximizing policy.

Doubts:

- How do you arrive at the loss function for Policy Gradient: You take the gradient of the objective function and then derive the loss function based on that gradient.
- Summation in derivation but not in statement of theorem: Both statements are correct and are use in their own ways The theorem is true for all values of t. Gradient is summed up over all values of t.
- PPO in RLHF:
  - Its similar to regular PPO only difference is that we use Monte Carlo learning to calculate advantages instead of TD learning.

    - Its easy to add the batch_dim, only a mask needs to be calculated for loss for the sentences that terminate early. Discounted rewards are also calculated differently.
    - To take multiple samples per prompt, the same prompts are interleaved.
- Q learning loss: It is calculated using MSE because that is what our objective is. Also because nowhere we are trying to calculate probabilities.
- How does actor critic reduce variance in policy gradient methods such as reinforce: Instead of Monte Carlo learning, we do temporal difference learning and hence the variance is reduced because due to stochasticity, the variance can accumulate over the entire episode.
- On policy vs Off policy algorithms: In off policy, the policy that is being used for inference is different from that used for training. In on policy methods, the same policy is used for acting and updating. Off-policy RL refers to algorithms that learn from experience collected by a different policy than the one currently being optimized.
- Online RL vs Offline RL: In online RL, the data is collected live using the policy we are trying to optimize for. In offline RL, we collect the data using different policies and use them for training a new policy. Offline RL is a stricter setting where the RL agent learns exclusively from a fixed dataset of pre-collected experiences without any further interaction with the environment.
- Offline RL vs Off Policy RL: Offline RL is a **subset** of off-policy RL. All offline RL methods are off-policy, but not all off-policy RL methods are offline.
- How is the KL divergence term defined for the term that is subtracted from rewards in PPO and the KL divergence term in GRPO ? So the term that is subtracted from rewards is exactly the same as the KL divergence term defined usually the summation over y sampled from policy is subsumed in the expectation term. In GRPO, KL divergence is estimated with an unbiased estimator as shown in http://joschu.net/blog/kl-approx.html log (p(x) / q(x)) has high variance due to negative values. Therefore -log (q(x)/p(x)) + q(x)/p(x) - 1 is used to estimate KL divergence as it is always +ve and doesn't change the expected value.
- Is the old policy used in KL divergence term and ratios for PPO always the SFT policy or is it updated ? According to our implementation, rewards should be given to the current policy so the outputs and log probs should be from the current model. Maybe the ratio is between updated old policy and new policy but the **KL divergence term is between new policy and SFT policy.**
- Why are all policy gradient methods on policy ? The vanilla policy gradient update, expectation is over trajectories sampled from the current policy. If you don't sample from current policy, your gradient estimator is **biased**. importance-weighted tricks can let us cheat a bit and go off-policy but they have high variance.
- What is generalized advantage estimation ?

Future readings:

- https://lilianweng.github.io/posts/2018-04-08-policy-gradient/

Importance Sampling in RL:

- If there be a mismatch between the policy log probs and the sample log probs & The sample network starts generating tokens the policy network assigns very large nll.
    - Reasons for mismatch and solutions:
        - Token Dropping in MOEs (Token dropping logic depends on the content of the entire batch rather than individual sequences): Increase capacity factor from 1.5 to 16.

        - Floating point deltas: Use float32 instead of bfloat16.
    - Why does the grad norm grow and why is policy loss unusually negative ?
- There are two possible strategies to deal with this problem: either ensure the policy and sample networks have consistent forward passes, or employ methods designed to improve off-policy training. Given that training with cf=16 is prohibitively expensive, we propose the latter.
    - Introduce the importance weights on policy gradient multiply the gradients / loss by a factor of policy network / sample network.
    - Capping the gradient of policy network to be a constant.

Mar 9, 2025

Pre-training

DeepSeek-V3:
- Data: 14.8 trillion tokens
- Modeling:
    - MOEs: Slightly different from DeepSeek-V2, DeepSeek-V3 uses the sigmoid function to compute the affinity scores, and applies a normalization among all selected affinity scores to produce the gating values.
        - Auxiliary Free Loss for Load Balancing: Instead of having a expert balance loss, we introduce a bias term bi for each expert and add it to the corresponding affinity scores si,t to determine the top-K routing. During training, we keep monitoring the expert load on the whole batch of each training step. At the end of each step, we will decrease the bias term by γ if its corresponding expert is overloaded, and increase it by γ if its corresponding expert is underloaded, where γ is a hyper-parameter called bias update speed.
        - Complementary Sequence-Wise Auxiliary Loss: Similar to expert balance loss applied at a sequence level. α is a hyper-parameter, which will be assigned an extremely small value for DeepSeek-V
        - No token dropping: DeepSeek-V3 does not drop any tokens during training or inference.
    - Multi-token prediction: Standard practice is to predict n future tokens using n independent heads. To save memory, the forward and backward are computed sequentially and gradients are accumulated which are then back propagated. . Multi-token prediction enables self-speculative decoding.

- Optimizations:
    - DualPipe:
    - FP8 mixed precision training:

DeepSeek-V2:

- Data: 8.1T tokens.
- Modeling:
  - Multi-head Latent Attention: The idea is to compress keys and values for efficient inference. In MHA, 2d elements per token need to cached. In MLA, instead of learning 2 weight matrices WQ and WV that maps input to keys and values, we learn 1 weight matrix WDKV that maps input to a compressed dimension vector and 2 separate weight matrices WUK and WUV for k & v to map it back to the original dimension. During inference, only the compressed vector needs to be cached, reducing cache to d_c elements per token. In order to reduce the activation memory during training, low rank compression is performed for queries as well. In addition, during inference, WUK can be absorbed into WQ , and WUV can be absorbed into WO , we even do not need to compute keys and values out for attention. MHA's KV cache is equal to GQA with only 2.25 groups
  - Decoupled Rotary Position Embedding: RoPE is incompatible with low-rank KV compression. If we apply RoPE for the keys directly WUK will be coupled with a position-sensitive RoPE matrix so that WUK cannot be absorbed into WQ any more during inference, since a RoPE matrix related to the currently generating token will lie between WQ and WUK and matrix multiplication does not obey a commutative law. A decoupled rope strategy that concatenates rope embeddings to original keys and queries is thereby proposed. We used multi-head queries and shared key to carry RoPE. Key is derived from raw input vector where queries are derived from compressed query vector.
  - DeepSeekMOE: We do this for economical training. MoE architectures substitute the Feed-Forward Networks (FFNs) in a Transformer with MoE layers. Each MoE layer consists of multiple experts, with each structurally identical to a standard FFN, and each token is assigned to one or two experts. Issues in current MOE: Knowledge Redundancy and Knowledge Hybridity. Knowledge Hybridity refers to the phenomenon that an expert tends to assemble vastly different types of knowledge in its parameters which are hard to utilize simultaneously. Also, multiple experts may converge in acquiring shared knowledge in their respective parameters, thereby leading to redundancy in expert parameters. DeepSeek MOE suggests two key changes for better specialization and less redundancy:
    - Shared Expert Isolation: Certain experts serve as shared experts that are always activated.
    - Fine-Grained Expert Segmentation: Less expert fwd dim, more experts and more activated experts. This would ensure that a single expert doesn't have to have vastly different types of knowledge as in Top 1 routing strategy. we segment each expert FFN into m smaller experts by reducing the FFN intermediate hidden dimension to $1/m$ times its original size. Since each expert becomes smaller, in response, we also increase the number of activated experts to m times to keep the same computation cost.
    - Expert Level Balance Loss

      LExpBal = α1* ∑' i=1toN' $fiPi$,

      fi = $N\,'$ / $K\,'T$ ∑ t=1toT 1(Token t selects Expert i),

      Pi = 1 / T ∑ t=1toT si,t
    - Device Level Balance Loss: In addition to the expert-level balance loss, we additionally design a device-level balance loss to ensure balanced computation across different devices.

$$\mathrm{LDevBal} = \alpha 2 \sum D\ i{=}1\ f'\ i\ \mathrm{P'}\ \mathrm{i}$$
$$\mathrm{f'i} = 1 / |\mathrm{Ei}| \sum \mathrm{j\ in\ Ei\ f\ j}$$
$$\mathrm{P'i} = \sum j{\in}\ \mathrm{E}i\mathrm{i\ Pj}$$

- ■ Communication Balance Loss: Finally, we introduce a communication balance loss to ensure that the communication of each device is balanced.

$$\mathrm{LCommBal} = \alpha 3 \sum D\ i{=}1\ f''\ i\ P''\ i$$
$$f''\ i = D/MT \sum t{=}1\mathrm{toT}\ 1(\text{Token } t \text{ is sent to Device } i)$$
$$P''\ i = \sum j{\in}\ \mathrm{E}i\ P\ \mathrm{j}$$

- ■ Device-Limited Routing: When expert parallelism is applied with fine grained expert segmentation, the routing costs can be high as number of activated experts are more. To limit the communication costs, they select M devices which have experts with highest affinity scores and then they limit the topK selection among these devices.
- ■ Token-Dropping Strategy: In order to further mitigate the computation wastage caused by unbalanced load, we introduce a device-level token-dropping strategy during training. We drop tokens with the lowest affinity scores on each device until reaching the computational budget. . In addition, we ensure that the tokens belonging to approximately 10% of the training sequences will never be dropped to have flexibility on whether to drop token during inference.
- ■ DeepSeek MOE: Warmup and Step Decay: Warmup for 2k steps, 0.316 at 80% maxsteps, x0.316 at 90% maxsteps.
- ■ Analysis on Expert Specialization:
    - ● Mask a certain ratio of experts with the highest routing probability, and then select top-K experts from the remaining routed experts. This effects deepseek more suggesting lower redundancy.
    - ● Disable shared expert and activate one more routed expert. Significant increase in pile loss.
- ■ For larger MOEs: Substitute all FFNs except for the first layer with MoE layers, since we observe that the load balance status converges especially slower for the first layer.
- ■ For DeepSeek MOE 16B: The batch size is set to 4.5K, and with a maximum sequence length of 4K, each training batch contains 18M tokens. Correspondingly, the total number of training steps is set to 106,449 to achieve 2T training tokens.
- ■ Tensor parallelism (Korthikanti et al., 2023; Narayanan et al., 2021; Shoeybi et al., 2019), ZeRO data parallelism (Rajbhandari et al., 2020), PipeDream pipeline parallelism (Harlap et al., 2018), and more specifically, expert parallelism (Lepikhin et al., 2021) by combining data and tensor parallelism.

- ○ Hyperparameters: The low-rank compression and fine-grained expert segmentation will impact the output scale of a layer. Therefore, in practice, we employ additional RMS Norm layers after the compressed latent vectors, and multiply additional scaling factors at the width bottlenecks (i.e., the compressed latent vectors and the intermediate hidden states of routed experts) to ensure stable training. We also use a batch size scheduling strategy, where the batch size is gradually increased from 2304 to 9216 in the training of the first 225B tokens, and then keeps 9216 in the remaining training.
- ○ Long Context Extension: We additionally train the model for 1000 steps, with a sequence length of 32K and a batch size of 576 sequences. Although the training is conducted solely at the

    - sequence length of 32K, the model still demonstrates robust performance when being evaluated at a context length of 128K.
- Optimizations:
    - Infrastructure: 16 way zero bubble pipeline parallelism, 8 way expert parallelism, Zero-1 data parallelism. No tensor parallelism, thereby decreasing the communication overhead.
    - Inference Efficiency: In order to efficiently deploy DeepSeek-V2 for service, we first convert its parameters into the precision of FP8. In addition, we also perform KV cache quantization for DeepSeek-V2 to further compress each element in its KV cache into 6 bits on average.
- Evaluation:
    - MMLU
    - CEval
    - HellaSwag
    - PIQA
    - ARC
    - BigBench Hard
    - TriviaQA
    - NaturalQuestions
    - RACE
    - DROP
    - C3
    - CMRC
    - WinoGrande
    - CLUEWSC
    - Pile (Language Modeling)
    - GSM8K
    - MATH
    - HumanEval
    - MBPP
    - AGIEval

Deepseek-LLM
- Data:
- Modeling:
- Evaluation:

Mar 8, 2025

Safety Evaluation of LLMs:
- Aspects:
    - Representational Bias: Gender, Religion, Race, Sexual Orientation, Age, Nationality, Disability, Physical Appearance, Socioeconomic Status
    - Toxic or Offensive Content: Insults, Hate speech, Threat, Violent Behavior, Bullying or Harassment, Sexually Arousing,

    - Dangerous Capabilities: Cyber-offense, Chemical, Biological, Radiological and Nuclear (CBRN) Risks, Weapons acquisition, Deception, Persuasion & manipulation, Mental Health Advice, Medical Advice, Political Comments or Opinions
- Safety Policies:
    -
- Benchmarks:
    - CrowSPairs: This dataset has 1500 examples that allows to measure biases in 9 categories: gender, religion, race/color, sexual orientation, age, nationality, disability, physical appearance and socioeconomic status. Each example is composed of a stereotype and an anti-stereotype. Examples were collected using human annotation on Mturk. To measure the biases of the models, we can compute the perplexity of the model on the stereotype.
    - WinoGender: Checks for gender bias in coreference resolutions. There are 120 sentences with an occupation, a participant and a pronoun. The sentences have been constructed in such a way that half the time the pronoun refers to the occupation and half the time it refers to the participants. For each sentence, we use male, female and neutral pronouns. If the model performs better on coreference resolution using neutral pronouns, then there is a bias in the model.
    - WinoBias:
    - BBQ:
    - RealToxicityPrompts: Sentences form Open WebText corpus are scored for toxicity. The scores are categorized into 4 buckets and 25k examples are sampled in each of the buckets. 50% of the prompts are toxic and the rest are non-toxic. Once an LLM completes a generation, it is scored by the perspective API.
    - SafetyBench: 11,435 diverse multiple choice questions spanning across 7 distinct categories of safety concerns.
    - PMIYC (Persuade me if you can):
- Human Evaluation:
    - Prompt Collection:
    - Guidelines for Rating:
    - Rating Template:
- Autoraters:
    - Zero-Shot Autoraters:
    - Training Data:
    - Modeling:
    - Evaluation:
- RedTeaming:
- Two Fold Eval Strategy: There is a trade off between the speed at which the evals need to be done vs the coverage of the evals. To manage this trade off we propose a two fold eval strategy with faster inner loop eval and a more complete outer loop eval.
    - Inner Loop Evals: Smaller set of evals run frequently and often alongside model training to aid development of the models and faster iteration. This would be 2-3 benchmarks along with the safety autoraters.
    - Outer Loop Evals: Complete set of evals run every 2-3 weeks or at the end of a revision cycle. The goal is to track progress. This would be all the benchmarks, human evals and autoraters.

Mar 7, 2025

Pre-training

LLAMA 1:

- Data: 1.4 Trillion tokens using the following sources:
    - English CommonCrawl:
        - Deduplication at line level,
        - language ID with fastext linear classifier to remove non-English pages.
        - Filter low quality content with an n-gram language model.
        - Linear model to classify pages used as references in wikipedia vs randomly sampled pages and discarded pages not classified as references.
    - C4:
        - Quality filtering based on punctuation marks, number of words or sentences in the page.
    - Github
        - Filter out low quality data based on heuristics such as line length or proportion of alphanumeric characters.
        - Remove boilerplate such as headers with regex.
        - Deduplicate at file level, with exact matches.
    - Wikipedia:
        - Remove hyperlink, comments and formatting boilerplate.
    - Gutenberg and Books3:
        - Deduplication at book level: Removing books with >90% overlap.
    - ArXiV:
        - Removed everything before the first section and bibliography.
        - Remove comments from tex files.
        - Remove user written macros
    - Stack Exchange:
        - Remove HTML tags
        - Sort answers by score.
- Modeling:
    - Architecture: Transformer decoder with following changes:
        - Pre-normalization with RMSNorm instead of LayerNorm
        - SwiGLU Activation Function in MLP.
        - Rotary Position Embeddings
    - Hyperparameters:
        - Cosine Learning Rate Schedule
        - AdamW Optimizer
        - Context Length = 2k
- Evaluation: Evaluated on free-form text generation and multiple choice questions in zero-shot and few-shot settings. For multiple choice questions, we select the completion with the highest likelihood given the provided context. On tasks that involve choosing one correct completion from several options

(multiple choice), we provide K examples of context plus correct completion, followed by one example of context only, and compare the LM likelihood of each completion.

- Reasoning:
    - HellaSwag:
    - BoolQ: Yes/No Questions. 1: Example yes/no questions from the BoolQ dataset. Each example consists of a question (Q), an excerpt from a passage (P), and an answer (A) with an explanation added for clarity. Questions are gathered from anonymized aggregated google search queries. Yes/No questions are heuristrically filtered. A question is selected if top 5 search result is a wikipedia article. Question and the article are given to annotator to identify if the question is meaningful, what paragraph in the article is sufficient to answer this question and the answer yes/no to the question.
    - PIQA: Given a goal and two solutions, which solution makes more sense. Questions that require physical commonsense knowledge.
    - SIQA: Multiple choice questions that require social commonsense knowledge.
    - WinoGrande
    - ARC easy and challenge
    - OpenBookQA
- General:
    - MMLU
    - TriviaQA
    - Natural Questions
    - RACE
- Code:
    - MBPP
    - HumanEval
- Math:
    - MATH
    - GSM8k
- Bias, Toxicity and Misinformation:
    - RealToxicityPrompts
    - CrowS-Pairs
    - WinoGender

LLAMA 2:

- Data: Increased the size of pre-training corpus by 40% resulting in 2T tokens.
    - Remove data from certain sites known to contain a high volume of personal information about private individuals.
    - Up-sampling the most factual sources in an effort to increase knowledge and dampen hallucinations.
- Modeling:
    - Architecture:
        - Grouped Multi-Query Attention
        - 2x Context Length = 4k
- Evaluation:

    - General:
        - SQuAD:
        - QUAC:
        - AGI Eval:
        - Big Bench Hard:
    - Reasoning:
        - CommonsenseQA:
    - Safety:
        - ToxiGen
        - BOLD
    - LongContext:
        - NarrativeQA
        - Qasper
        - ZeroSCROLLS/QuALITY
        - QMSum
        - ContractNLI
        - SQuAD
- Safety:
    - Remove data from certain sites known to contain a high volume of personal information about private individuals.
    - For informational purposes, we compute statistics for Demographic Representation of Pronouns and identities such as Religion, Gender and Sexual Orientation, Nationality, Race and Ethnicity.
    - Data Toxicity: We use HateBert classifier to assign toxicity score to each sentence in the document, the average of which is the score for the document. Only <0.2% of the documents have a toxicity score of > 0.5. We keep these documents in the pre-training data , to allow Llama 2 to be more widely usable across tasks (e.g., it can be better used for hate speech classification).
    - Language Identification: Using fastext language identification tool and a threshold of 0.5 for the language detection.
    - We observe that models trained from less aggressively filtered pre-training data also required fewer examples to achieve reasonable safety-alignment.

LLAMA 3:

- Data: 15.6T multilingual tokens compared to 2T tokens in LLAMA2. Much of the data we utilize is obtained from the web. We apply several de-duplication methods and data cleaning mechanisms on each data source to obtain high-quality tokens.
    - PII and safety filtering: we implement filters designed to remove data from websites are likely to contain unsafe content, domains that have been ranked as harmful according to a variety of Meta safety standards, and domains that are known to contain adult content.
    - Text Extraction and Cleaning:
    - De-duplication:
        - URL-level de-duplication: We keep the most recent version for pages corresponding to each URL

        - Document-level de-duplication: We perform global MinHash de-duplication across the entire dataset to remove near duplicate documents.
        - Line-level de-duplication: We remove lines that appeared more than 6 times in each bucket of 30M documents. It removes leftover boilerplate from various websites such as navigation menus, cookie warnings etc.
    - Heuristic filtering:
    - Model-based quality filtering:
    - Code and reasoning data:
    - Multilingual data:
    - Determining the Data Mix:
    - Annealing Data:
- Modeling:
    - Hyperparameters:
        - Context length: 8k -> 128k
- Evaluation:
    - General:
        - IFEval
        - MMLU-Pro
    - Code:
        - MBPP EvalPlus
    - ToolUse:
        - BFCL
        - Nexus
    - LongContext:
        - InfiniteBench/En.MC
        - NIH/Multi-needle
    - Multilingual:
        - MGSM
- Scaling Laws: Scaling laws are developed to determine optimal model size given the compute budget. It is also used to predict the performance on downstream benchmarks:
    - Compute optimal Model: For compute budget between 6x10^18 flops and 10^22 flops, we plot ISOFlop curves. On the X axis we have number of training tokens, Using flop budget and number of training tokens, we can compute the model size. On the y axis we have NLL corresponding to the model size. For each Flop budget, we approximate the ISOFlop curve using a second degree polynomial. We get the minimum of these ISOFlop curves as the optimum model size. We use the compute-optimal models we identified this way to predict the optimal number of training tokens for a specific compute budget. To do so, we then plot #Training tokens on the Y axis and the number of Flops on the X axis. We assume a power-law relation between compute budget, C, and the optimal number of training tokens, N(C) = AC^α. We find that (α, A) = (0.53, 0.29). Extrapolation of the resulting scaling law to 3.8 × 10^25 FLOPs suggests training a 402B parameter model on 16.55T tokens.
    - Predicting the performance on downstream benchmarks: A two stage method is used: First, we linearly correlate the (normalized) NLL of correct answer in the benchmark and the training FLOPs. We use this to estimate the NLL of the flagship model on the benchmark. Then NLL is

    correlated with accuracy using the scaling laws models and the older models trained on higher FLOPs. We establish a sigmoidal relation between the log-likelihood and accuracy using both the scaling law models and Llama 2 models. Using the predicted NLL in the previous step, we estimate the accuracy using the sigmoidal relation.

- Parallelism: To scale training for our largest models, we use 4D parallelism—a combination of four different types of parallelism methods—to shard the model.
    - Tensor parallelism: Tensor parallelism splits individual weight tensors into multiple chunks on different devices. This is also referred to as horizontal parallelism as the splitting happens on horizontal level.
    - Pipeline parallelism: Pipeline parallelism partitions the model vertically into stages by layers, so that different devices can process in parallel different stages of the full model pipeline. We encountered several challenges with existing implementations of pipeline parallelism and suggest the following changes:
        - Current implementations of pipeline parallelism have batch size constraint, memory imbalance due to embedding layer and warmup microbatches and computation imbalance due to output & loss calculation making the last stage execution latency bottleneck. **They modify the pipeline schedule to run an arbitrary number of microbatches in each batch**. To balance the pipeline, we reduce one Transformer layer each from the first and the last stages, respectively. This means that the first model chunk on the first stage has only the embedding, and the last model chunk on the last stage has only output projection and loss calculation.
    - Context parallelism: It divides the input context into segments, reducing memory bottleneck for very long sequence length inputs. The keys and values need to be communicated across nodes.
    - Data parallelism: We use fully sharded data parallelism which shards the model, optimizer, and gradients while implementing data parallelism which processes data in parallel on multiple GPUs and synchronizes after each training step.
- Infrastructure:

Mar 6, 2025

Transformers: Transformers is composed of an encoder and decoder. Encoder decoder consists of N layers each of which has 2 sublayers: MHA and MLP. Around each of the two sublayers, a residual connection followed by layer normalization is added. Decoder also has N layers but it adds another sublayer which performs cross-attention over the encoder outputs. In decoder, we use masked attention to prevent tokens from attending to future tokens.

- Attention is All You Need:
    - Why multi-head, why not single head with larger dimension ? If we have single headed attention, we'll have only 1 attention map. Having multi-head attention leads to n_head attention maps and each could learn a different relation. Due to the reduced dimension of each head, the total computational cost is similar to that of single-head attention with full dimensionality.

    - Why divide attention by sqrt of dk ?: For large values of dk, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients 4 . To counteract this effect, we scale the dot products by √ 1 / dk . Assuming q and k have independent components with zero mean and unit variance, Variance of the dot product is scaled by a factor of dk, therefore we divide it by √ 1 / dk so that the variance doesn't add up and is independent of dk.
    - Additive Attention vs Dot Product Attention: In additive attention, the attention weight is computed by applying a feedforward network over the concatenation of query and key vector. This feedforward layer is a learnable parameter. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code. It also doesn't require extra learnable parameters.
    - Cross-attention: In "encoder-decoder attention" layers, the queries come from the previous decoder layer, and the keys and values come from the output of the encoder. This allows every position in the decoder to attend over all positions in the input sequence.
    - Feed-Forward Networks: MLP layer in Transformers consist of 2 linear transformations with a ReLU activation in between. The first linear layer increases the dimension to 4x that of model_dim and the second linear layer brings it back to model_dim.
    - Sinusoidal Position Encodings: It allows the model to easily learn to attend by relative positions, since for any fixed offset k, PE(pos+k) can be represented as a linear function of PE(pos). An advantage of the sinusoidal positional embedding over learned positional embedding is that it allows the model to extrapolate to sequence lengths longer than the ones encountered during training.
    - Hyperparameters: Learning rate schedule: Linear increase upto warmup steps=4000, inv sqrt of number of steps afterwards. Dropout before each sublayer output is added and normalized. Label smoothing with p=0.1. They also use checkpoint averaging using the last 10 or 20 ckpts.
- GPT-2:
    - Hyperparameters: Cosine Learning Rate Schedule: max(0.01, max_lr * 0.5 * (1 + math.cos(math.pi * steps_after_warmup/(max_steps - warmup)))). Anneals to a minimum. Residual, embedding and attention dropout of 0.1.
    - GELU activation function: The GELU activation function is xΦ(x), where Φ(x) the standard Gaussian cumulative distribution function
    - Layer normalization was moved to the input of each sub-block and an additional layer normalization was added after the final self-attention block.
    - A modified initialization which accounts for the accumulation on the residual path with model depth is used. We scale the weights of residual layers at initialization by a factor of 1/ √ N where N is the number of residual layers.
    - Alternating dense and locally banded sparse attention patterns in the layers of the transformer.
- LLAMA2:
    - Bias is set to False in all linear layers. This is done mainly for simplicity and the model works well even without the bias.
    - No dropout is used. In LLMs more generally, the need for regularization is reduced because we train for only 1 epoch on the pre-training data. RMSNorm along with weight decay provides sufficient regularization.

    - SwiGLU(w1,w2,w3,x) = (Swish(w1(x)) * w3(x)) w2(x). Since there are 3 weight matrices instead of 2, in GLU activations, the hidden dim is set to ⅔ of the usual hidden_dim so that number of parameters remains the same.
    - Grouped Multi-Query Attention: There are lesser k,v heads than q heads. To implement this k,v heads are repeated to match q heads. To repeat, we expand x[:,:,None,:,:] to n_rep and then reshape. This is done to reduce kv cache leading to faster inference.
    - RMSNorm: RMSNorm which only focuses on re-scaling invariance and regularizes the summed inputs simply according to the root mean square (RMS) statistic. It simplifies LayerNorm by totally removing the mean statistic. It works well in conjunction with zero bias.
    - Rotary Position Embeddings (RoPE):

Important Resources:
- https://nlp.seas.harvard.edu/2018/04/03/attention.html
- https://jalammar.github.io/illustrated-transformer/

Mar 5, 2025

Multimodal Pre-training: For multimodal pre-training, we need a lot of image text pairs. We can treat each image text pair as a document in itself. To use the next token prediction loss as we usually use in text pre-training, we'd need to convert the images into hard tokens. To achieve that, we train Vision Encoders.
- Vision Transformers (ViT): Pretrained on lots and lots of supervised image classification data. Each image is converted to N patches of dim PxP which is flattened and then converted to dimension D with a learnable linear layer. It uses a transformer with alternating MHA and MLP with layer normalization before and residual connections. The input to the vision transformer could be raw images or it could be output of CNN layer itself. As the model size increases, the benefit of having representations from a CNN model goes down. Vision Transformers can be used to obtain image representations for use in models like CoCa.
- Vision Encoders:
    - Contrastive Captioners (CoCa): Combines two loss functions: contrastive loss between the image and text encoding and captioning loss on the image captioning task. Cross-attention from the image encoder is omitted for the lower layers of the decoder to learn unimodal text representations on which the contrastive loss is applied. They add a separate [CLS] token to text whose encoding is used as the text encoding for the contrastive loss. They use attentional poolers with n_query = 256 for captioning loss and n_query = 1 for contrastive loss.
    - VQVAE: Proposed training VAEs with discrete latent variables. There is a codebook of variables and the encoder representation is mapped to the nearest neighbor in the codebook. This step is not differentiable but it is assumed that the encoder representation is learnt in such a way that it really close to its nearest neighbor in the codebook and thus the gradient of the two are the same. The codebook embeddings do not receive any gradients from the reconstruction loss. Instead, we use the l2 error to move the embeddings towards the encoder outputs. There is another loss term which encourages the encoder output to stay close to the selected embedding.
    - VQGAN: Combines the best of both worlds: the local spatial understanding of CNNs with global attention of transformers. It first uses a CNN encoder to reduce the dimension of images. The

    encoding is then mapped to the nearest neighbor in a learnable codebook, then we have a CNN decoder which tries to reproduce the image. There is a CNN based discriminator network which checks whether the image is real or fake. Simultaneously, once they obtain the discrete codebook representations, they also model a next token predict task on those discrete tokens using a transformer. All in all there are 4 losses: Image reconstruction loss: VQ loss which uses stop gradient alternatively on encodings and codebook representations, there is GAN loss and there is a next token prediction loss for transformers. For class-conditioned image synthesis, a class-id token is prepended before the image tokens
    - ViT-VQGAN: ViT-VQGAN replaces the CNN in encoder and decoder of VQGAN with Vision Transformer (ViT). The encoder of ViT maps 256x256 images into 32x32 tokens using 8x8 non-overlapping patches. They use a larger codebook size without top-p and top-k sampling. They apply L2 normalization for codebook and encoded latent variables. They use factorized codes which can be viewed as decoupling code lookup and code embedding. Factorized codes reduce dimension of lookup space from 768 to 32. It uses a linear projection from the output of the encoder to a low dimensional latent variable space for code index lookup. This prevents dead codebook entries.
    - Contrastive Captioners: It adds a VQ module to CoCa to discretize the embeddings from ViT encoder. VQ module has an encoder, a decoder and a codebook. The VQ module can be interpreted in one of the 2 ways: the encoder is simply doing the nearest neighbor search and producing discrete ids from the cookbook, the decoder is simply doing a lookup of those discrete ids from the cookbook. There is an additional embedding reconstruction loss that pushes the codebook embeddings to be closer to the encoder output. Another way to look at it would be that encoder and decoder are both combinations of FW and MHA and convert the input to a hidden representation on which we do the nearest neighbor seach which is discretized, then we take the codebook embeddings of those discrete representations, put it through a decoder which gives us the output of the VQ module. We then want these outputs to be closer to the input to the VQ module so we add a embedding reconstruction loss along with a codebook loss which encourages the embeddings to be closer to encoder outputs.
    - CLIP: https://arxiv.org/pdf/2103.00020
    - unCLIP: https://cdn.openai.com/papers/dall-e-2.pdf
- Generative Pretraining from Pixels: Pre-training Transformer Decoder with either an autoregressive next token prediction loss or masked language modeling loss. They rescale the images to 32x32 and then they have 32x32x3 context size with num_classes = 256. Another way is to cluster the RGB values to 512 clusters, that reduces the context length to 32x32 and increases num_classes = 512. Each cluster or RGB value is treated as an id and there is an embedding lookup to obtain its representation.
- Diffusion Model for Image Generation (Pictor):

Mar 4, 2025

Batch Normalization

- What is the internal covariate shift ? It refers to the phenomenon where distribution of inputs of the next layer changes during training as the network weights are updated. It impacts the stability of the training. Batch Normalization (BN) mitigates internal covariate shift by normalizing the inputs to each layer.

- Why do we want the input distribution to be unit gaussian ? The vanishing/exploding gradient problem is less severe when the input is normalized, the weight initialization techniques assume unit gaussian inputs. It also makes it easier for optimization algorithms like SGD to work efficiently.
- Why do we scale and shift in batch normalization ? It gives the network ability to choose whether or not it wants the normalization. If the network decides that the raw inputs are better, gamma can be standard deviation and beta can be the mean to restore the raw inputs.

Mar 3, 2025

Autoraters for LLM Evaluation

- LLMs are fine-tuned on large scale human ratings to train autoraters. Human ratings are expensive and time consuming to obtain. Autoraters provide for cheaper evaluation for faster development of the models.
- Data: The data could either be pointwise of the form: prompt, response and ratings or It could be pairwise and have the form: prompt, response A, response B, rating. Since its hard to evaluate LLM responses, pairwise data is preferred as comparing two responses becomes an easier task rather than scoring one response.
    - The autorater model strongly depends on the quality of training data. The coverage of eval prompts, the quality and diversity of model responses, noise in the human feedback can all impact the quality of trained autorater models.
- Modeling: The training is similar to that of Supervised finetuning stage where prompt, response A and response B along with rater instructions are given as the prompt to the model and the rating is the response of the model. The objective is simply the next token prediction which makes it easy to train such models.
- Meta-Evaluation: To measure the quality of these models, we track correlation with human judgement. In particular for the pairwise task, we track item-wise accuracy and projectwise spearman correlation.
- Challenges:
    - Distributional Shift: One thing to keep in mind while training autorater models is that the models will improve over time leading to a distribution shift between train and test time. To combat these, autorater models need to be refreshed periodically.

Mar 2, 2025

Scaling LLM Test-Time Compute

- Chain of Thought: Language models are guided through intermediate reasoning steps to solve complex problems. This behavior emerges in sufficiently large models. It also provides interpretable solutions, facilitating debugging and understanding of model outputs.
- Self-Refine: The model generates response and feedback which is then used to refine the response. The quality improves with the number of refinements made to the response. The output is preferred over Best of N approach. There are however recent papers which suggest that this alone is not enough to improve the model response.

- Multiagent Debate: Each model generates responses, critiques others' answers, and iteratively refines its own response, leading to more accurate and consensus-driven outputs. We see quality improvements across reasoning and factuality tasks with and without Chain of Thought. The quality increases with number of agents and number of debate rounds.
- Best of N Sampling: Sampling N outputs in “parallel” from a base LLM and selecting the one that scores the highest per a learned verifier or a reward model. Reward model can either be outcome supervised or process supervised. We can also use majority voting.
- Process Reward Models: Process supervised reward models are trained in an SFT fashion where labels of each step are provided during a single training example and all the steps are labeled in one inference call.
- Which approach to use: It depends on the nature of the problem and the capabilities of base LLM. On easier problems, for which the base LLM can already readily produce reasonable responses, iteratively refining the answers is more effective. On the other hand, with more difficult problems, re-sampling new responses independently in parallel or deploying search against a process-based reward model is more effective.
- For easy to medium problems, scaling test time compute with smaller models is a more effective strategy. For the most challenging problems, scaling pre-training still remains the most effective strategy.

Mar 1, 2025

SWEBench: Can Language Models Resolve Real World Github Issues
- Issue and Code snapshot is provided. Model is supposed to generate a PR that is evaluated against real tests.
- Template: Instruction, Issue, Code {File names and file content}, Example Patch.
- Since all of code wouldn’t fit into context, they perform BM25 retrieval or Oracle retrieval.
- SWEBenchVerified: SWE-bench dataset contains some tasks that are impossible to solve without additional context outside of the GitHub issue. It is a 500 problem subset of SWE-bench that has been reviewed by humans to make sure they are solvable
- SWEBenchSonnet: Agent=Model+Scaffolding (with Tools). Sonnet gets 49% resolution on SWE bench using 2 tools: bash and edit.

TODO

- https://research.character.ai/optimizing-inference/
- Gpipe, Gshard
- https://github.com/karpathy/nanoGPT
- Sasha Rush puzzles: https://github.com/srush?tab=repositories&q=puzzles&type=&language=&sort=
- https://kipp.ly/transformer-inference-arithmetic/
- The FLOPs Calculus of Language Model Training

- https://irhum.github.io/blog/pjit/
- Long Context: https://arxiv.org/pdf/2306.15595, https://arxiv.org/pdf/2309.00071
- https://arxiv.org/pdf/2502.05171
- https://www.reddit.com/r/LocalLLaMA/comments/1inch7r/a_new_paper_demonstrates_that_llms_could_think_in/
- Liian Wang
  - 03/29: https://lilianweng.github.io/posts/2023-01-10-inference-optimization/
  - 03/30: https://lilianweng.github.io/posts/2024-07-07-hallucination/
  - 03/30: https://lilianweng.github.io/posts/2020-04-07-the-transformer-family/
  - 03/30: https://lilianweng.github.io/posts/2023-01-27-the-transformer-family-v2/
  - 03/31: https://lilianweng.github.io/posts/2021-09-25-train-large/
  - 04/02: https://lilianweng.github.io/posts/2022-06-09-vlm/
  - 04/04: https://lilianweng.github.io/posts/2024-11-28-reward-hacking/
  - 04/05: https://lilianweng.github.io/posts/2023-10-25-adv-attack-llm/
  - 04/05: https://lilianweng.github.io/posts/2021-03-21-lm-toxicity/
  - 04/08: https://lilianweng.github.io/posts/2020-06-07-exploration-drl/
  - 04/08: https://lilianweng.github.io/posts/2020-01-29-curriculum-rl/
  - 04/08: https://lilianweng.github.io/posts/2018-04-08-policy-gradient/
  - 04/08: https://lilianweng.github.io/posts/2018-02-19-rl-overview/
  - 04/08: https://huggingface.co/blog/the_n_implementation_details_of_rlhf_with_ppo
  - 04/12: https://lilianweng.github.io/posts/2024-02-05-human-data-quality/
- Other things to study:
  - https://huggingface.co/blog/moe#load-balancing-tokens-for-moes
  - https://arxiv.org/pdf/2309.00071
  - https://arxiv.org/pdf/2402.04792
- Softmax Derivative: https://chatgpt.com/share/67fb11e0-83b0-8010-ba07-c83bce0b3bb2