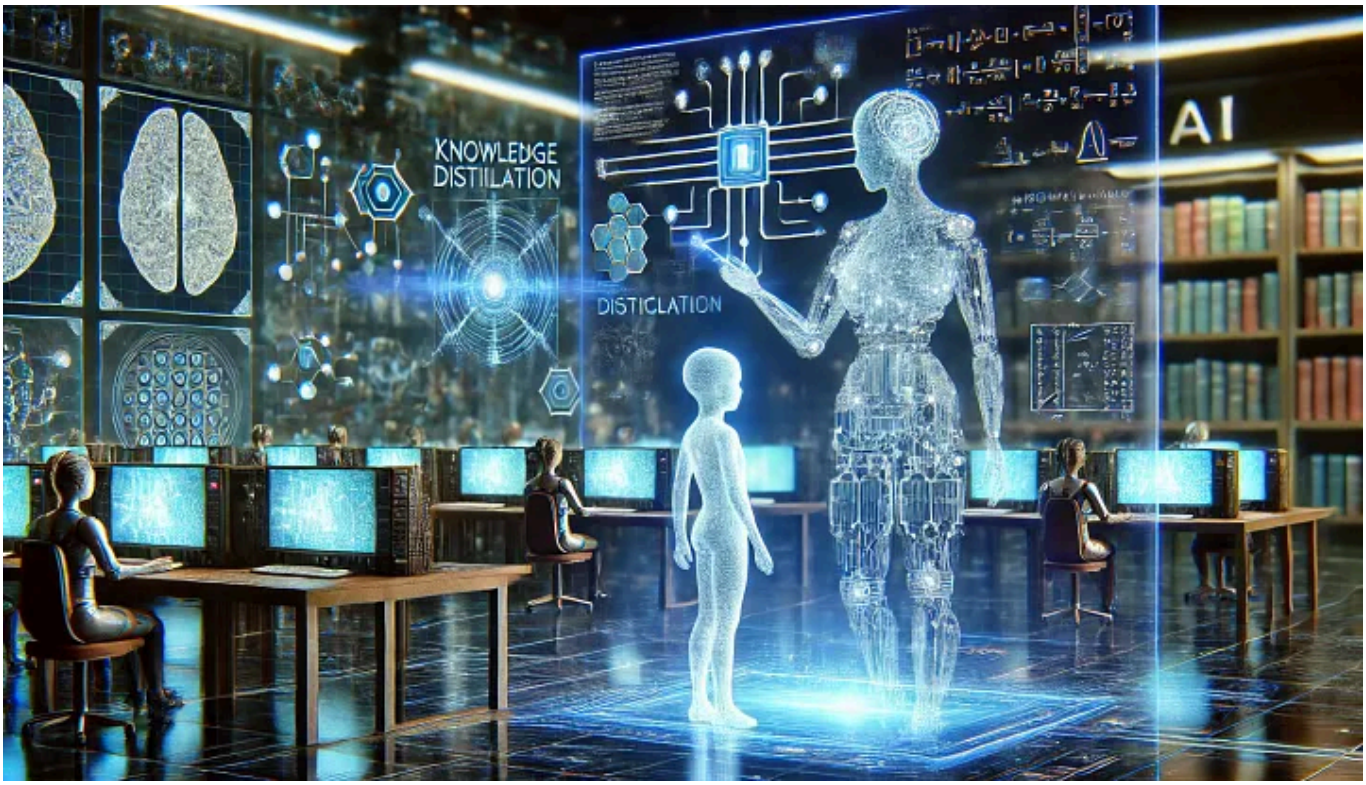
Member-only story

How to Distill a LLM: Step-by-step

The Google Paper that started efficient LLM distillation. Let's explore how it works, the math behind this technique, and how to implement it with code.

Cristian Leo · Following
Published in Data Science Collective · 25 min read · Just now



LLM teacher and a LLM student — Image generated by DALL-E

Large Language Models, or LLMs, have truly stormed onto the scene, haven't they? It feels like just yesterday we were marveling at simple chatbots, and now we have this sprawling intelligence capable of generating surprisingly coherent text, translating languages, and even writing code.

But, and it's a rather large 'but', these models are... well, large. Colossally so. We're talking about massive computational resources, specialized hardware, and energy consumption that makes you think twice about asking it to summarize your grocery list. For many of us, especially those of us tinkering outside of massive tech companies, the sheer scale of these models presents a real barrier. It's a bit like having access to the world's most powerful telescope, but only if you can afford to build a mountain to put it on.

This is where the idea of distillation comes into play. Imagine you're a master chef teaching an apprentice. You wouldn't expect them to replicate your Michelin-star dishes immediately, right? Instead, you'd focus on distilling the core techniques, the essential flavors, into something they can manage and master. LLM distillation is, in essence, similar. We're trying to take the vast knowledge and capabilities of these behemoth models and condense them into something smaller, more manageable, and crucially, more efficient. Traditional distillation methods have been around for a while, attempting to transfer knowledge, but often, it feels like we're losing some of the magic in the process. We get a smaller model, sure, but sometimes it feels... less intelligent, less capable of the nuanced reasoning we see in the big guys.

Now, a research paper titled "Distilling Step-by-Step!", by a group of Google researchers and academic, proposes a fascinating approach that not only shrinks these LLMs but, suggests that these smaller models can actually outperform their larger teachers. It's a bold claim, I know, and naturally, my data scientist senses are tingling with both curiosity and a healthy dose of

skepticism. Could it really be true? Can we create these 'genius in a shoebox' models that are not just smaller, but smarter?

What makes this "Distilling Step-by-Step" method particularly intriguing is its focus on reasoning. Instead of just treating the LLM as a source of labels, this method cleverly extracts the rationale behind its answers — the step-by-step thought process, if you will. It's like not just getting the answer to a math problem, but also seeing the detailed work. This rationale, it seems, becomes a powerful teaching tool, guiding the smaller model to learn not just **what** to predict, but **why**.

While the promise is certainly there — smaller, faster, potentially even better models — it's important not to get too carried away. Is this a silver bullet? Probably not. There are likely complexities to unpack, limitations to consider, and perhaps even scenarios where traditional methods might still hold advantages. For instance, how robust is this approach across different types of tasks? And what are the computational costs of generating these rationales in the first place? These are the kinds of questions that buzz around in my head, and I imagine in yours too if you're reading this.

So, let's delve into "Distilling Step-by-Step." We'll dissect the math that underpins this method, build a simplified version from scratch in Python to really get our hands dirty, and explore the results presented in the paper. Let's try to understand not just how it works, but also why it seems to be so effective, and where its boundaries might lie.

1. The Bottleneck of Big Models: Why Distillation?

Let's face it, the sheer scale of these Large Language Models is, in many ways, their superpower. That vastness, that almost incomprehensible number of parameters, is what allows them to perform those amazing feats

we've been talking about. It's like a sprawling neural network that has soaked up almost the entire internet, learning patterns and relationships that smaller models can only dream of. However, and this is a crucial point, size becomes a significant bottleneck when we try to actually use these models in practical applications. It's a bit like having a Formula 1 race car — incredible performance, no doubt, but try using it for your daily commute in rush hour traffic. Suddenly, its size and power become more of a hindrance than a help.

Think about the computational muscle required to run these behemoths. We're not just talking about your average laptop here. Serving a single, state-of-the-art LLM often demands specialized hardware, rooms full of GPUs, and a hefty electricity bill that could make your eyes water. It's an infrastructure challenge and a costly one at that. For smaller companies, for researchers working with limited resources, or even for just wanting to run these models on edge devices like phones or embedded systems? Forget about it. The computational demands are simply prohibitive. It's akin to needing a whole power plant just to illuminate a single lightbulb — effective, perhaps, but wildly inefficient.

And it's not just about the upfront cost. Latency, the time it takes for the model to generate a response, is another critical factor. Imagine interacting with a chatbot that takes minutes to reply to each of your questions. Frustrating, right? Large models, while powerful, can be slower due to the sheer amount of computation involved in each inference. For real-time applications, where speed is paramount, this latency can be a deal-breaker. It's like having a conversation with someone who is incredibly knowledgeable but takes ages to formulate each sentence — the flow of interaction is just…gone.

So, what's the alternative? This is where the concept of **LLM distillation** comes to the rescue. Distillation, in essence, is about knowledge transfer. We're trying to distill the essential "knowledge" and capabilities of a large, powerful LLM — often referred to as the teacher model — into a smaller, more efficient model, the student. Think of it like creating a concentrated extract. You start with a large volume of something rich and flavorful, and through a careful process, you reduce its size while retaining, and perhaps even enhancing its key qualities.

The core motivation behind distillation is clear: to create smaller models that can perform comparably to their larger counterparts, but with significantly reduced computational cost and faster inference times. It's about making powerful AI more accessible, more deployable, and frankly, more sustainable. Imagine being able to run sophisticated language models on your phone, in your smart home devices, or in applications where resources are inherently limited. That's the promise of distillation. It's about democratizing AI, taking it out of the server farms, and putting it into the hands of more people, in more places.

Now, traditional distillation techniques have been around for some time. Methods like knowledge distillation, often using "soft labels" from the teacher or trying to mimic intermediate representations, have shown some success. These approaches generally aim to train the student to replicate the teacher's output behavior. While these methods can indeed shrink models and improve efficiency, they often feel like they're trading off some crucial element, some essential spark of intelligence. It can be akin to photocopying a masterpiece — you get a copy, sure, but the subtle nuances, the depth, and the original vibrancy often get lost in translation. And when it comes to capturing the complex reasoning abilities of LLMs, traditional distillation

sometimes falls short. It's as if we're teaching the student to mimic the answers without truly understanding the reasoning behind them.

This is where "Distilling Step-by-Step" offers a potentially exciting leap forward. It's not just about making models smaller; it's about making them smaller and smarter, by focusing on distilling not just the outputs, but the very process of reasoning. And that, as we'll explore further, could be a game-changer.

2. Introducing Distilling Step-by-Step

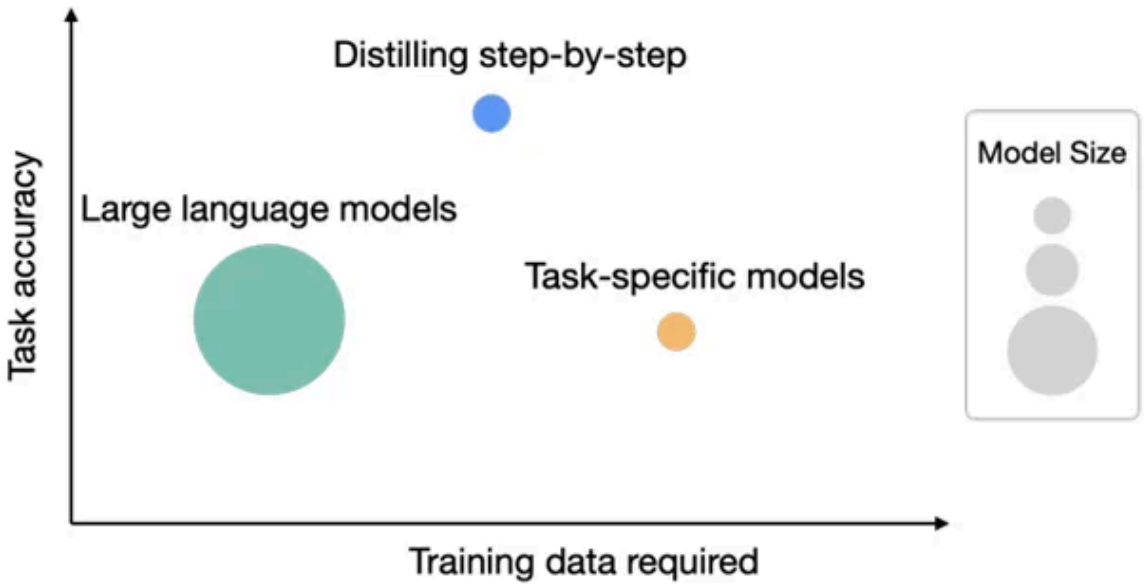


Chart extracted from original paper "Distilling Step-by-Step! Outperforming Larger Language Models with Less Training Data and Smaller Model Sizes"

So, if traditional distillation is like photocopying a masterpiece and potentially losing some of its essence, what does "Distilling Step-by-Step" do differently? Well, the core innovation lies in a fundamental shift in perspective. Instead of just viewing the Large Language Model as a black box

that spits out answers, this method recognizes and leverages something truly fascinating: the LLM's ability to reason.

It's almost like we're moving from simply asking the teacher for the answer key to instead asking them to show their work, to explain their thought process. And it turns out, that these "thought processes," often elicited through clever prompting techniques like Chain-of-Thought (CoT), can be incredibly valuable in guiding a smaller model to learn more effectively. Imagine it like this: if you're learning to play chess, just memorizing winning moves might get you so far, but understanding the strategic reasons behind those moves, the underlying principles of chess, will make you a far more adaptable and ultimately, a better player. "Distilling Step-by-Step" attempts to impart this deeper understanding, this "strategic reasoning," to the smaller student model.

The process, as outlined in the research paper, unfolds in two distinct yet interconnected phases. Let's break them down.

First, we have the **Rationale Extraction Phase**. This is where we essentially interview our large, "teacher" LLM. We use a technique called Chain-of-Thought prompting, which is quite ingenious, really. Think of it as crafting a very specific set of questions designed to not just get the answer, but to coax out the step-by-step reasoning that leads to that answer. Essentially, we're not just asking "What's the answer?", but "How did you arrive at that answer? Can you explain your thinking, step-by-step?". The research paper utilizes "few-shot" CoT prompting, which means providing the LLM with a few examples of input-rational-label triplets to guide its generation. It's like showing the LLM a few worked examples before asking it to solve a new problem and explain its approach.

For example, consider a simple question like, “If a train travels at 60 mph for 2 hours, how far does it go?”. A traditional LLM prompt might just ask for the answer. But with CoT prompting, we encourage it to generate intermediate reasoning steps: “Speed = 60 mph, Time = 2 hours. Distance = Speed x Time. Distance = 60 mph x 2 hours = 120 miles.” The output isn’t just “120 miles,” but also the rationale: “Distance = Speed x Time” and the calculation steps. These rationales, these natural language explanations of the LLM’s reasoning, are the gold we’re mining in this phase.

The output of this phase is a valuable dataset. For each input, we now have not only the LLM’s predicted label (the answer), but also a natural language rationale justifying that label. It’s like creating a rich training dataset where each example is not just input-output, but input-reasoning-output.

Next comes the **Multi-Task Training Phase**. This is where we take our smaller, “student” model and put it to school, using the dataset we just created. And this is where the “multi-task” aspect becomes crucial. Instead of just training the student to predict the final label, as in traditional distillation, we train it to do two things simultaneously: predict the label and generate the rationale.

It’s a bit like teaching that chess apprentice not just to make winning moves, but also to explain why each move is strategically sound. We’re not just asking the student to mimic the teacher’s answers, but to understand and replicate the teacher’s reasoning process. The paper frames this as a multi-task learning problem. The student model is trained to minimize not one, but two loss functions: one for label prediction accuracy, and another for how well it generates the rationales. Both tasks are given weight, forcing the student to learn both the “what” (the answer) and the “why” (the reasoning).

A clever trick they use is “task prefixes.” During training, they prepend the input with “[label]” when the target is the label and “[rationale]” when the target is the rationale. It’s like explicitly telling the student, “Okay, for this input, I want you to focus on predicting the label,” and then, “Now, for this input, I want you to focus on generating the reasoning.” This helps the model disentangle the two tasks and learn to perform both effectively.

Now, it’s worth pausing here to consider why this approach might be more effective than traditional distillation or even standard fine-tuning. Traditional methods often focus on mimicking the surface behavior of the teacher — the outputs. “Distilling Step-by-Step”, on the other hand, aims to capture something deeper: the underlying reasoning process. By forcing the smaller model to generate rationales, we’re essentially encouraging it to learn a more abstract, more generalizable understanding of the task. It’s not just memorizing input-output pairs; it’s learning the principles that connect inputs to outputs.

This focus on reasoning is the key differentiator and the source of the potential data efficiency gains we’ll explore later. By learning to generate rationales, the smaller model might be able to generalize better from fewer examples, and potentially even surpass the performance of the teacher LLM in certain scenarios. It’s a compelling idea, and as we delve deeper into the math and implementation, we can start to unpack exactly how this step-by-step distillation magic works.

3. The Math Behind Distillation Step-by-Step

Alright, let’s pull back the curtain and peek at the mathematical machinery that powers “Distilling Step-by-Step.” Now, I know formulas can sometimes look intimidating, but trust me, we’ll break it down piece by piece, and hopefully, it will all start to make sense. We’ll explore not just what the

formulas say, but also what they imply, and where their strengths and potential limitations might lie.

3.1 Rationale Extraction with Chain-of-Thought (CoT) Prompting

While Chain-of-Thought prompting itself isn't strictly defined by a single equation, it's more of a methodology to guide the generation process of a Large Language Model. At its heart, it's about influencing the probability distribution of the LLM's output.

Imagine an LLM as a function that, given an input prompt P, generates an output sequence Y. In standard prompting, we aim to maximize the probability of a desired output y (the label), given the prompt P. We can represent this as:

$$P(Y = y|P)$$

Probability of generating a desired output y given a prompt P

CoT prompting subtly shifts this. We want to encourage the LLM to generate not just the final answer *y*, but also an intermediate *rationale r* that leads to *y*. We do this by crafting prompts that demonstrate this input-rationale-label structure, as we discussed earlier with the example triplets.

While we don't have a direct formula for CoT prompting itself, we can think of it as conditioning the LLM's generation on a specific *style* of output — one that includes explicit reasoning steps. It's like nudging the LLM's internal decision-making process to become more transparent, more step-by-step.

The magic of CoT lies in its emergent behavior. Large Language Models, when prompted correctly, demonstrate a surprising ability to mimic this

chain-of-thought process, even though they weren't explicitly trained to generate rationales in this way.

However, it's worth noting a few nuances. CoT prompting is still somewhat of an art. Crafting effective prompts requires intuition and experimentation. There's no guarantee that the generated rationales will always be perfect, or even entirely accurate. They are, after all, generated by a model that, while impressive, is not infallible. And the quality of these rationales will undoubtedly influence the effectiveness of the subsequent distillation process. It's a bit of a "garbage in, garbage out" scenario — if the teacher's reasoning is flawed, the student might learn flawed reasoning too.

Despite these caveats, CoT prompting provides a powerful way to tap into the reasoning capabilities of LLMs and extract valuable supervisory signals for training smaller models. The output of this phase, as we noted, is a dataset of input-rationale-label triplets (xi , $r\hat{}i$, $y\hat{}i$), ready for the next stage: multi-task learning.

3.2 Multi-Task Learning Objective

This is where the mathematical heart of "Distilling Step-by-Step" truly beats. We move from simply *extracting* rationales to *utilizing* them in a multi-task learning framework. As we discussed, the core idea is to train the student model to perform two tasks simultaneously: label prediction and rationale generation. This is formalized through a combined loss function.

3.2.1 Label Prediction Loss

The first task, and arguably the more traditional one, is label prediction. We want our student model f to accurately predict the correct label $y\hat{}i$ for a given input x_i. To measure how well it's doing, we use the standard **cross-**

entropy loss, denoted as *L_label*. For a dataset of *N* examples, the label prediction loss is calculated as:

$$L_{label} = \frac{1}{N} \sum_{i=1}^{N} \ell(f(x_i), \hat{y}_i)$$

Cross Entropy Loss Formula — Image by Author

Here, ℓ(f(x_i) , y^i) represents the cross-entropy loss between the student model's predicted probability distribution for the labels, *f*(*xi*), and the target label *y^i*. In essence, this loss function penalizes the model when its predicted probabilities deviate from the correct label. It's a well-established loss function in classification tasks, and its strength lies in its ability to effectively guide the model towards making accurate predictions. However, it's worth noting that cross-entropy loss primarily focuses on label accuracy and doesn't directly encourage the model to learn *reasoning* or generate explanations. That's where the next loss function comes in.

3.2.2 Rationale Generation Loss

This is the novel ingredient in "Distilling Step-by-Step." To encourage the student model to learn to *reason*, we introduce a second task: rationale generation. We want the model to not only predict the label but also to generate a plausible rationale *r^i* for input *x_i*. Again, we use the cross-entropy loss, but this time, applied to the sequence of tokens in the rationale:

$$L_{rationale} = \frac{1}{N} \sum_{i=1}^{N} \ell(f(x_i), \hat{r}_i)$$

Cross Entropy Loss on Rationale Formula — Image by Author

Here, $\ell(f(xi), r\hat{}i)$ is the cross-entropy loss calculated over the token sequences. The student model is now penalized not just for incorrect labels, but also for generating rationales that are dissimilar to the teacher-generated rationales $r\hat{}i$. This loss function directly encourages the model to learn to produce human-readable explanations for its predictions. It's a powerful way to inject the "reasoning style" of the teacher LLM into the smaller student. However, it's important to acknowledge that this loss function is based on *imitation*. We are essentially asking the student to mimic the teacher's rationales, which might not always be the optimal or most efficient form of reasoning. There could be alternative, perhaps even better, ways for the student model to arrive at the correct answer, but this loss function biases it towards replicating the teacher's reasoning process.

3.2.3 Combined Multi-Task Loss: LL

Finally, to train the student model to perform both tasks simultaneously, we combine these two loss functions into a single, joint loss function:

$$L = L_{label} + \lambda L_{rationale}$$

Combined Loss Function — Image by Author

This is a simple yet elegant combination. We sum the label prediction loss and the rationale generation loss, weighted by a factor λ. In the research paper, they often set λ=1, giving equal importance to both tasks. The weighting factor λ allows us to adjust the relative importance of rationale generation versus label prediction. If we want to prioritize label accuracy, we could decrease λ. Conversely, if we want to emphasize learning to reason and generate explanations, we could increase λ. However, in practice, setting λ=1 often strikes a good balance.

This combined loss function is the engine that drives the multi-task learning process in “Distilling Step-by-Step.” By minimizing this joint loss, the student model is incentivized to become proficient at both predicting labels and generating rationales. It’s a clever way to leverage the rich information contained in the teacher-generated rationales to guide the training of a smaller, more efficient, and potentially even more insightful student model.

Now that we’ve dissected the math, let’s get our hands dirty and see how we can bring these equations to life in Python code.

4. Building Distilling Step-by-Step in Python

Alright, enough theory! It’s time to get our hands dirty and translate those mathematical formulas into actual Python code. We will refer to the original GitHub repo published by Google Researchers:

Let’s dive in!

4.1. Data Loading and Preprocessing: `data_utils.py`

First of all, we need a method to load the data, which we will use to distill our student model. This is what we can see in the file `data_utils.py`, which:

- **Loads datasets from various sources:** Whether it’s using the Hugging Face `load_dataset` method or loading from a custom JSON file, our

`DatasetLoader` base class (and its subclasses) manage it.

- **Prepares inputs and targets:** For example, the `CQADatasetLoader` builds a combined input string from a question and its multiple-choice answers. It also removes unnecessary columns so that the downstream training sees only what matters.
- **Integrates LLM outputs:** Want to use external rationales and labels from an LLM? The loader can read these from JSON files (for either PaLM or GPT predictions) and parse them into structured columns.

Here's what this class looks like:

```
class DatasetLoader(object):
    def __init__(self, dataset_name, source_dataset_name, dataset_version, has_v
                 batch_size, train_batch_idxs, test_batch_idxs, valid_batch_idxs
        self.data_root = DATASET_ROOT
        self.dataset_name = dataset_name
        self.source_dataset_name = source_dataset_name
        self.dataset_version = dataset_version
        self.has_valid = has_valid
        self.split_map = split_map
        # (Additional setup omitted for brevity…)

    def load_from_source(self):
        if self.dataset_version is None:
            datasets = load_dataset(self.source_dataset_name)
        else:
            datasets = load_dataset(self.source_dataset_name, self.dataset_versi
        return datasets

    def to_json(self, datasets):
        for k, v in self.split_map.items():
            datasets[v].to_json(f'{self.data_root}/{self.dataset_name}/{self.dat

    # …plus methods for loading from JSON and parsing LLM/GPT outputs.
```

Each concrete loader (e.g. `CQADatasetLoader`, `SVAMPDatasetLoader`, etc.) implements dataset-specific logic. For instance, the CQA loader's `_post_process` method constructs an input string by combining a question with its answer choices:

```
def prepare_input(example):
    question = example['question']
    c_0 = example['choices'][0]
    # …other choices...
    input = f'{question}\nAnswer Choices:\n(a) {c_0}\n(b) {example["choices"][1]
    example['input'] = input
    example['label'] = example['answer']
    return example
```

This way, the pipeline is primed for multi-task training — where the model learns both to predict the answer *and* (via auxiliary signals) to generate a rationale.

4.2. Evaluation Metrics: `metrics.py`

No serious project is complete without robust evaluation. In `metrics.py`, we have functions that compute accuracy for both text and equation predictions. Notice that:

- **Text Accuracy:** Simply compares the decoded predictions against the labels.
- **Equation Accuracy:** Evaluates the string expressions (using Python's `eval` in a controlled way) to see if the computed answers match.

For example, our equation accuracy function looks like this:

```
def compute_equation_acc(preds, labels):
    preds = [eval_equation(pred) for pred in preds]
    labels = [eval_equation(label) for label in labels]
    return np.mean(np.array(preds) == np.array(labels))
```

And `eval_equation` tries to safely compute the result of an equation:

```
def eval_equation(equation):
    try:
        answer = eval(equation)
    except:
        answer = np.nan
    return answer
```

This module is vital when your tasks aren't just classification but involve more complex reasoning, such as mathematical problem-solving.

4.3. Multi-Task Model and Trainer Setup: `model_utils.py`

Here's where things get really interesting. Our pipeline supports multi-task training by using **task prefixes** to distinguish between prediction (answering) and explanation (generating rationales). The code uses some HuggingFace classes as parent classes and extends on top of them.

Custom Data Collator

The `TaskPrefixDataCollator` takes the batch of examples and splits it into two dictionaries:

- One for the primary prediction task.
- One for the explanation (auxiliary) task.

```
class TaskPrefixDataCollator(DataCollatorForSeq2Seq):
    def __call__(self, features, return_tensors=None):
        features_df = pd.DataFrame(features)
        pred_features = features_df.loc[:, ~features_df.columns.isin(['aux_label
        expl_features = features_df.loc[:, ~features_df.columns.isin(['labels',
            columns={'aux_labels': 'labels', 'expl_input_ids': 'input_ids', 'exp
        pred_features = super().__call__(pred_features, return_tensors)
        expl_features = super().__call__(expl_features, return_tensors)
        return {
            'pred': pred_features,
            'expl': expl_features,
        }
```

Custom Trainer

The `TaskPrefixTrainer` extends Hugging Face's `Seq2SeqTrainer` by overriding the `compute_loss` method. This allows us to combine the loss from the primary prediction task and the auxiliary explanation generation task using a weighted sum (controlled by the parameter `alpha`):

```
class TaskPrefixTrainer(Seq2SeqTrainer):
    def __init__(self, alpha, output_rationale, **kwargs):
        super().__init__(**kwargs)
        self.alpha = alpha
        self.output_rationale = output_rationale
```

```
    def compute_loss(self, model, inputs, return_outputs=False):
        pred_outputs = model(**inputs['pred'])
        expl_outputs = model(**inputs['expl'])
        loss = self.alpha * pred_outputs.loss + (1. - self.alpha) * expl_outputs
        return (loss, {'pred': pred_outputs, 'expl': expl_outputs}) if return_ou
```

This design elegantly fuses the two tasks — guiding the student model to learn not just the answer but also the reasoning process.

4.4. Running the Distillation Pipeline: `rain.py`

The main entry point, `rain.py`, orchestrates everything. It parses command-line arguments to select the dataset (CQA, SVAMP, ESNLI, ANLI1, or even ASDiv for data augmentation), the type of LLM predictions to use (PaLM or GPT), and other hyperparameters.

Step 1 — Dataset Preparation
Depending on the chosen dataset, the appropriate loader is instantiated. For example:

```
if args.dataset == 'cqa':
    dataset_loader = CQADatasetLoader()
elif args.dataset == 'svamp':
    dataset_loader = SVAMPDatasetLoader()
# …and so on.
```

Step 2 — Integrating LLM Predictions
If you're distilling from an LLM, the code loads external rationales and labels

and adds them as new columns:

```
datasets['train'] = datasets['train'].add_column('llm_label', train_llm_labels)
datasets['train'] = datasets['train'].add_column('llm_rationale', train_llm_rati
```

Step 3 — Tokenization and Task Prefixing

Using a pre-trained tokenizer (say, from `google/t5-v1_1-base`), the code tokenizes the examples. For task-prefix models, it even prepends "predict:" and "explain:" to the input text:

```
def tokenize_function(examples):
    model_inputs = tokenizer(['predict: ' + text for text in examples['input']],
    expl_model_inputs = tokenizer(['explain: ' + text for text in examples['inpu
    model_inputs['expl_input_ids'] = expl_model_inputs['input_ids']
    model_inputs['expl_attention_mask'] = expl_model_inputs['attention_mask']
    # (Encode labels and rationales as targets)
    return model_inputs
```

Step 4 — Training and Evaluation

Finally, `rain.py` calls the `train_and_evaluate` function (found in `train_utils.py`), passing in the tokenized datasets and the metric functions. This function sets everything up and kicks off training.

4.5. The Training Loop: `train_utils.py`

In `train_utils.py`, the heavy lifting of training is performed.
The function `get_config_dir(args)` builds a directory path for saving checkpoints and logs, based on the current hyperparameters. This makes it easy to track different runs.

The T5 model is loaded via `T5ForConditionalGeneration.from_pretrained(args.from_pretrained)`. If needed, the model can be parallelized across GPUs.

We use `Seq2SeqTrainingArguments` to specify the training parameters (e.g., learning rate, batch size, evaluation frequency).

Depending on whether we're using a task prefix model or a standard one, we instantiate either our custom `TaskPrefixTrainer` (which supports the dual loss) or the vanilla `Seq2SeqTrainer`.

Lastly, training is started with:

```
trainer.train()
```

This entire process — from data loading to model training — embodies our step-by-step distillation philosophy: guiding a student model to learn not only the correct outputs but also the reasoning behind them.

5. Outperforming the Teacher

Now, the moment of truth. All this talk about rationales, multi-task learning, and step-by-step distillation — does it actually *work* in practice? Does it

deliver on its promise of creating smaller, more efficient models that can rival, or even surpass, their larger teachers? The research paper, “Distilling Step-by-Step!”, presents some compelling empirical evidence, and it’s these results that truly make this approach noteworthy. Let’s dive into the key findings, focusing on data efficiency and model size, and see what story the numbers tell.

5.1 Data Efficiency: Learning More with Less

One of the most striking claims of “Distilling Step-by-Step” is its improved **data efficiency**. In essence, it suggests that smaller models trained with this method can achieve comparable, and often better, performance than models trained with traditional methods, but using significantly *less* training data. This is a huge deal, especially in the world of Large Language Models where data scarcity and the cost of data annotation can be major bottlenecks.

Figure 4 and Figure 5 from the paper visually demonstrate this data efficiency advantage. In these experiments, they compared “Distilling Step-by-Step” to two common approaches: **Standard Fine-tuning** (when human-labeled data is available) and **Standard Task Distillation** (when only unlabeled data is available). They trained 220M T5-Base models using varying amounts of training data for each method and evaluated their performance on several NLP benchmark datasets.

The results are quite telling. Across datasets like e-SNLI, ANLI, CQA, and SVAMP, “Distilling Step-by-Step” consistently outperformed both Standard Fine-tuning and Standard Task Distillation when trained on *reduced* datasets. In some cases, the gains were dramatic. For example, on e-SNLI, the paper reports that “Distilling Step-by-Step” trained on just **12.5%** of the full dataset could outperform Standard Fine-tuning trained on **100%** of the same dataset!

That's a massive reduction in data requirements for achieving the same level of performance.

Similarly, when compared to Standard Task Distillation (using unlabeled data), "Distilling Step-by-Step" again showed superior data efficiency. It consistently achieved better performance with less unlabeled data. For instance, on the ANLI dataset, they observed that "Distilling Step-by-Step" needed only **12.5%** of the full unlabeled dataset to surpass the performance of Standard Task Distillation trained on the entire 100% dataset.

To visualize this, imagine two students learning a new subject. Student A (Standard Fine-tuning/Distillation) diligently studies all the textbook chapters and practice problems. Student B ("Distilling Step-by-Step") studies only a fraction of the material but has access to detailed explanations of the key concepts and problem-solving strategies from a master teacher (the LLM rationales). The results suggest that Student B, with less raw material but better guidance, can often learn more effectively and even outperform Student A who simply grinds through everything.

This data efficiency boost is likely attributable to the richer supervisory signal provided by the rationales. By learning to generate reasoning steps, the smaller model is not just memorizing input-output mappings but is internalizing a more generalizable understanding of the task. This deeper understanding allows it to learn effectively from fewer examples, generalizing better to unseen data.

However, it's worth noting that data efficiency gains might vary depending on the task and dataset. The paper shows impressive results on the benchmarks they tested, but it's possible that for some tasks, the benefits might be less pronounced. And, of course, the quality of the teacher-

generated rationales plays a crucial role. If the rationales are noisy or uninformative, the data efficiency advantage might diminish. It's not a magic bullet, but the evidence strongly suggests that "Distilling Step-by-Step" offers a significant step forward in training more data-efficient language models.

5.2 Model Size Reduction: Tiny Students, Giant Performance

Beyond data efficiency, the research paper also highlights the potential for significant **model size reduction**. The goal of distillation, after all, is to create smaller, more deployable models. But "Distilling Step-by-Step" goes a step further, suggesting that these smaller models, trained with rationales, can not only be more efficient but also achieve surprisingly high performance, sometimes even outperforming the massive teacher LLMs themselves.

Figure 6 and Figure 7 in the paper illustrate this point dramatically. They compare the performance of T5 models of varying sizes (from 220M T5-Base to 11B T5-XXL) trained with "Distilling Step-by-Step" and Standard Fine-tuning/Distillation to the performance of the 540B parameter PaLM LLM using Few-shot CoT prompting and PINTO tuning. The x-axis represents the model size (in parameters), and the y-axis shows the task performance.

The graphs reveal a striking trend. "Distilling Step-by-Step" consistently outperforms Standard Fine-tuning and Distillation across all model sizes. But the real eye-opener is how smaller models trained with "Distilling Step-by-Step" stack up against the massive PaLM LLM.

In several cases, the paper demonstrates that T5 models, orders of magnitude smaller than PaLM, can achieve comparable or even *better* performance. For instance, on the ANLI dataset (Figure 6), a **770M** parameter T5-Large model trained with "Distilling Step-by-Step" surpasses

the performance of the **540B** parameter PaLM LLM using Few-shot CoT. That's a model that is over **700 times smaller** achieving better results! Similarly, on e-SNLI, a **220M** T5-Base model outperforms PaLM, a model over **2000 times larger**.

It's almost counterintuitive, isn't it? How can a tiny student outshine its giant teacher? The researchers suggest that by distilling the *reasoning process*, "Distilling Step-by-Step" is creating models that are not just smaller, but also more *specialized* and *focused* for the specific tasks they are trained on. The larger LLMs, while incredibly versatile, are also general-purpose models, potentially carrying a lot of "extra weight" that isn't strictly necessary for a specific task. The distilled, step-by-step trained models, on the other hand, are leaner, more task-optimized, and perhaps, in some ways, more efficient at leveraging the relevant information for the task at hand.

However, it's crucial to maintain a balanced perspective. Outperforming LLMs in these specific benchmarks doesn't necessarily mean that these smaller models are universally "better" in all aspects. LLMs still retain their strengths in terms of zero-shot generalization, few-shot learning across a wider range of tasks, and overall versatility. The "Distilling Step-by-Step" approach seems to excel at creating highly efficient, high-performing models for *specific* tasks, but may not replicate the broad, general intelligence of massive LLMs. It's more about creating specialized "geniuses" in a narrow domain, rather than general-purpose "polymaths."

5.3 Minimum Resources for Maximum Impact

Perhaps the most compelling aspect of "Distilling Step-by-Step" is its potential to minimize both data requirements *and* model size while achieving top-tier performance. Figures 8 and 9 in the paper explore this "resource efficiency frontier." They visualize the trade-off between the

amount of training data used, the final task performance, and the size of the resulting model.

These figures show that “Distilling Step-by-Step” often occupies a sweet spot in the resource-performance trade-off space. It can achieve LLM-level performance (and sometimes surpass it) with both significantly smaller models *and* less training data compared to standard methods. For example, on e-SNLI, “Distilling Step-by-Step” can outperform PaLM with a model over 2000x smaller and using only 0.1% of the full dataset!

This simultaneous reduction in both data and model size is a powerful combination. It suggests that “Distilling Step-by-Step” offers a path towards creating highly effective language models that are not only more computationally efficient but also more accessible and sustainable to train and deploy. It’s about achieving maximum impact with minimum resources, a particularly appealing prospect in the resource-constrained world of AI development.

Of course, further research is needed to fully understand the scope and limitations of “Distilling Step-by-Step.” But the initial results are undeniably promising. It seems to offer a genuinely innovative approach to distillation, one that goes beyond simple mimicry and taps into the deeper reasoning capabilities of Large Language Models to create smaller, smarter, and more data-efficient AI systems. And that, in my book, is a step in a very exciting direction.

Next up, we’ll wrap things up with a conclusion, summarizing the key takeaways and pondering the broader implications of this step-by-step distillation revolution.

improvement in model compression. It feels like a potentially significant shift in how we think about training and deploying powerful language models.

The implications of these advantages are far-reaching. “Distilling Step-by-Step” could be a key enabler in **democratizing access to powerful AI.** This could unlock a wave of innovation, allowing smaller companies, researchers, and individuals to leverage the power of advanced language AI without requiring vast infrastructure.

Furthermore, the increased efficiency could have significant **environmental benefits**. Training and running massive LLMs consumes a considerable amount of energy. Smaller, more efficient models, achieving comparable performance, could contribute to a more sustainable AI ecosystem. It’s about getting more “intelligence per watt,” a crucial consideration as AI becomes increasingly integrated into our lives.

However, it’s important to reiterate that “Distilling Step-by-Step” is not a universal panacea. As with any technique, it has its potential limitations and areas for further exploration.

Despite these open questions, “Distilling Step-by-Step” represents a genuinely exciting and promising direction in LLM research. It’s a step away from simply scaling up models and towards a more nuanced approach of distilling not just knowledge, but also the very process of reasoning.

The journey from massive, unwieldy LLM giants to potentially more agile, “genius in a shoebox” models is just beginning. But “Distilling Step-by-Step” offers a compelling roadmap, suggesting that the future of powerful AI may not just be about size, but about smarter, more efficient, and more accessible intelligence for everyone. And that’s a future I’m genuinely excited to see unfold.

References

- Google Research. (2024). *Distilling Step-by-Step!*. GitHub Repository. arXiv preprint arXiv:abs/2305.02301.
- Hinton, G., Vinyals, O., & Dean, J. (2015). *Distilling the Knowledge in a Neural Network*. arXiv preprint arXiv:1503.02531.
- Wei, J., et al. (2022). *Chain of Thought Prompting Elicits Reasoning in Large Language Models*. arXiv preprint arXiv:2201.11903. https://arxiv.org/abs/2201.11903
- Vaswani, A., et al. (2017). *Attention is All You Need*. Advances in Neural Information Processing Systems (NeurIPS). https://arxiv.org/abs/1706.03762
- Jang, Y., et al. (2023). *Can Distilled Models Be Better Than Their Teachers?* Proceedings of the International Conference on Learning Representations (ICLR). https://openreview.net/forum?id=example

Recommended Readings

Transformer-Squared: Stop Finetuning LLMs

The architecture behind self-adaptive LLMs, the math and code behind Transformer-Squared, and Single Value...

levelup.gitconnected.com

Don't Do RAG: Cache is the future

CAG or RAG? Let's explore Cached Augmented Generation, its math, and trade-offs. Let's dig into its research paper to...

levelup.gitconnected.com

The Math Behind Transformers

Deep Dive into the Transformer Architecture, the key element of LLMs. Let's explore its math, and build it from scratch...

medium.com

Data Science | Artificial Intelligence | Programming | Python | Machine Learning

Published in Data Science Collective

57 Followers · Last published just now

Follow

Advice, insights, and ideas from the Medium data science community

Written by Cristian Leo

34K Followers · 11 Following

Following

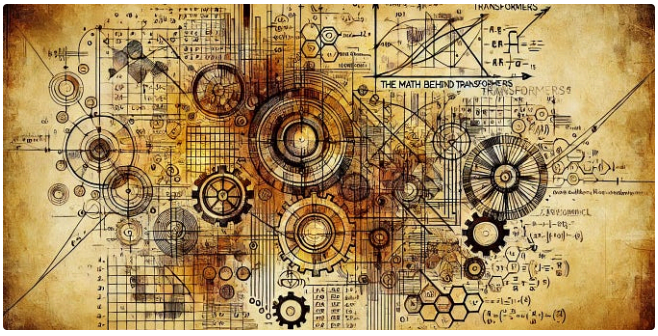
Data Scientist @ Amazon with a passion about recreating all the popular machine learning algorithm from scratch.

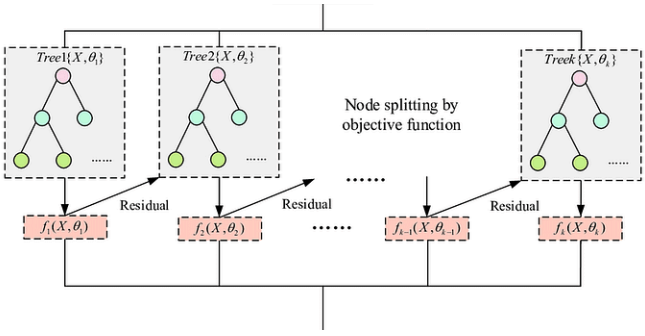
No responses yet

What are your thoughts?

Respond

More from Cristian Leo and Data Science Collective





Cristian Leo

The Math Behind Transformers

Deep Dive into the Transformer Architecture, the key element of LLMs. Let's explore its...

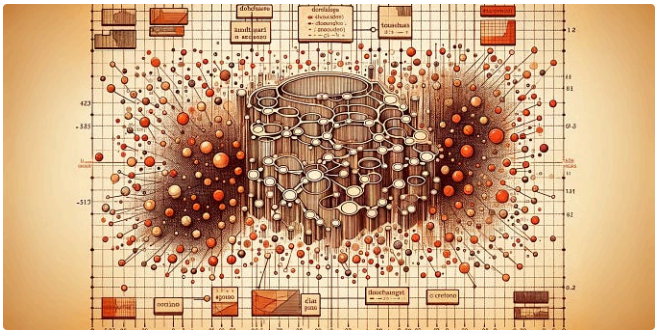
Cristian Leo

The Math Behind XGBoost

Building XGBoost from Scratch Using Python

Jul 25, 2024 957 8

Jan 10, 2024 510 5



In TDS Archive by Cristian Leo

Reinforcement Learning 101: Building a RL Agent

Decoding the Math behind Reinforcement Learning, introducing the RL Framework, an...

Feb 19, 2024 893 10

In TDS Archive by Cristian Leo

The Math Behind K-Means Clustering

Why is K-Means the most popular algorithm in Unsupervised Learning? Let's dive into its...

Feb 13, 2024 729 3

See all from Cristian Leo

See all from Data Science Collective

Recommended from Medium

Daniel Avila

Step-by-Step: Running DeepSeek locally in VSCode for a Powerful,...

This step-by-step guide will show you how to install and run DeepSeek locally, configure it...

Feb 2 342 11

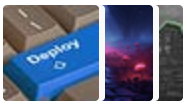
In Towards AI by Krishan Walia

Fine-tuning DeepSeek R1 to respond like Humans using Python!

Learn to Fine-Tune Deep Seek R1 to respond as humans, through this beginner-friendly...

Feb 2 200 3

Lists



Predictive Modeling w/ Python

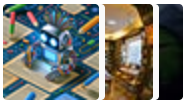
20 stories · 1823 saves

Practical Guides to Machine Learning

10 stories · 2195 saves

Coding & Development

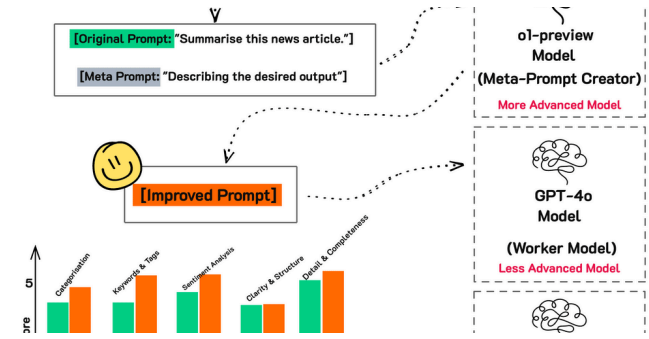
11 stories · 1002 saves



Natural Language Processing

1930 stories · 1582 saves

Henry Navarro



Cobus Greyling

Ollama vs vLLM: which framework is better for inference? 👊 (Part II)

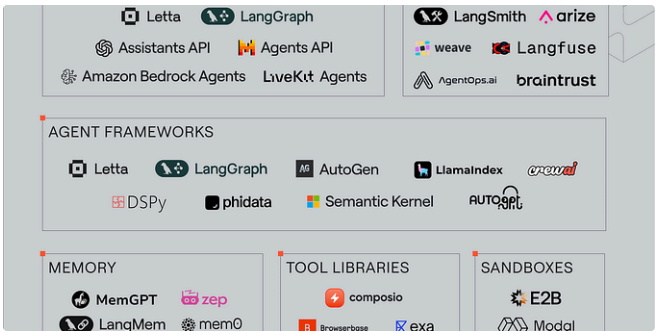
Exploring vLLM: The Performance-Focused Framework

Feb 3 126 1

Meta Prompting: A Practical Guide to Optimising Prompts...

Discover how meta prompting can enhance your results by using advanced models to...

5d ago 218



Vipra Singh

AI Agents: Introduction (Part-1)

Discover AI agents, their design, and real-world applications.

Feb 2 261 7

Luke Skyward

I Built 3 Apps with DeepSeek, OpenAI o1, and Gemini—Here's...

Results of My Week-Long Experiment

Feb 3 384 15

See more recommendations