Member-only story

How to Build an LLM from Scratch

Data Curation, Transformers, Training at Scale, and Model Evaluation

Shaw Talebi

Published in Towards Data Science · 16 min read · Sep 21, 2023

This is the 6th article in a series on using large language models (LLMs) in practice. Previous articles explored how to leverage pre-trained LLMs via prompt engineering and fine-tuning. While these approaches can handle the overwhelming majority of LLM use cases, it may make sense to build an LLM from scratch in some situations. In this article, we will review key aspects of developing a foundation LLM based on the development of models such as GPT-3, Llama, Falcon, and beyond.

Photo by Frames For Your Heart on Unsplash

Historically (i.e. less than 1 year ago), training large-scale language models (10b+ parameters) was an esoteric activity reserved for AI researchers. However, with all the AI and LLM excitement post-ChatGPT, we now have an environment where businesses and other organizations have an interest in developing their own custom LLMs from scratch [1]. Although this is not necessary (IMO) for >99% of LLM applications, it is still beneficial to understand what it takes to develop these large-scale models and when it makes sense to build them.

Supplemental Video.

How much does it cost?

Before diving into the technical aspects of LLM development, let's do some back-of-the-napkin math to get a sense of the financial costs here.

Meta's Llama 2 models required about 180,000 GPU hours to train its 7b parameter model and 1,700,000 GPU hours to train the 70b model [2]. Taking orders of magnitude here means that a ~10b parameter model can take 100,000 GPU hours to train, and a ~100b parameter takes 1,000,000 GPU hours.

Translating this into commercial cloud computing costs, an Invidia A100 GPU (i.e. what was used to train Llama 2 models) costs around $1–2 per GPU per hour. That means a **~10b parameter model costs about $150,000 to train, and a ~100b parameter model costs ~$1,500,000.**

Alternatively, you can buy the GPUs if you don't want to rent them. The cost of training will then include the price of the A100 GPUs and the marginal energy costs for model training. An A100 is about $10,000 multiplied by 1000 GPUs to form a cluster. **The hardware cost is then on the order of $10,000,000.** Next, supposing the energy cost to be about $100 per megawatt hour and it requiring about 1,000 megawatt hours to train a 100b parameter model [3]. That comes to a **marginal energy cost of about $100,000 per 100b parameter model.**

These costs do not include funding a team of ML engineers, data engineers, data scientists, and others needed for model development, which can easily get to $1,000,000 (to get people who know what they are doing).

Needless to say, training an LLM from scratch is a massive investment (at least for now). Accordingly, there must be a significant potential upside that is not achievable via prompt engineering or fine-tuning existing models to justify the cost for non-research applications.

4 Key Steps

Now that you've realized you do not want to train an LLM from scratch (or maybe you still do, IDK), let's see what model development consists of. Here, I break the process down into 4 key steps.

1. **Data Curation**
2. **Model Architecture**
3. **Training at Scale**
4. **Evaluation**

Although each step has a bottomless depth of technical detail, the discussion here will stay relatively high-level, only highlighting a handful of key details. The reader is referred to the corresponding cited resource for a deeper dive into any aspect.

Step 1: Data Curation

Machine learning models are a product of their training data, which means **the quality of your model is driven by the quality of your data** (i.e. "garbage in, garbage out").

This presents a major challenge for LLMs due to the tremendous scale of data required. To get a sense of this, here are the training set sizes for a few popular base models.

- **GPT-3 175b:** 0.5T Tokens [4] (T = Trillion)
- **Llama 70b:** 2T tokens [2]
- **Falcon 180b:** 3.5T [5]

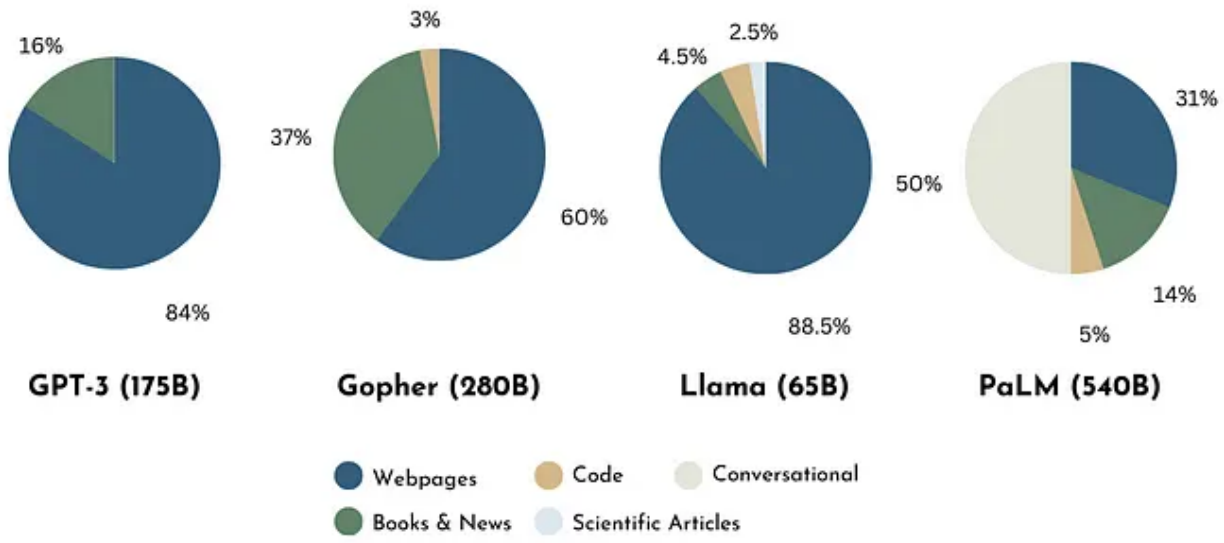
This translates to about a trillion words of text i.e. about 1,000,000 novels or 1,000,000,000 news articles. *Note: if you are unfamiliar with the term token, check out the explanation in a previous article of this series.*

Where do we get all these data?

The internet is the most common LLM data mine, which includes countless text sources such as webpages, books, scientific articles, codebases, and conversational data. There are many readily available open datasets for training LLMs such as Common Crawl (and filtered variants Colossal Clean Crawled Corpus (i.e. C4), and Falcon RefinedWeb), The Pile (a cleaned and diverse 825 GB dataset) [6], and many others on Hugging Face's datasets platform (and elsewhere).

An alternative to gathering human-generated text from the Internet (and other sources) is to have an existing LLM (e.g. GPT-3) generate a (relatively) high-quality training text corpus. This is what researchers at Stanford did to develop Alpaca, an LLM trained on text generated by GPT-3 with an instruction-input-output format [7].

Regardless of where your text is sourced, **diversity** is a key aspect of a good training dataset. This tends to **improve model generalization** for downstream tasks [8]. Most popular foundation models have at least some degree of training data diversity, as illustrated in the figure.



Comparison of training data diversity across foundation models. Inspired by work by Zhao et al. [8]. Image by author.

How do we prepare the data?

Gathering a mountain of text data is only half the battle. The next stage of data curation is to ensure training data quality. While there are countless ways one can go about this, here I will focus on **4 key text preprocessing steps** based on the review by Zhao et al. [8].

Quality Filtering — This aims to **remove "low-quality" text from the dataset** [8]. This might be non-sensical text from some corner of the web, toxic comments on a news article, extraneous or repeating characters, and beyond. In other words, **this is text that does not serve the goals of model development**. Zhao et al. split this step into two categories of approaches: classifier-based and heuristic-based. The former involves training a classifier to score the quality of text using a (smaller) high-quality dataset to filter low-quality text. The latter approach employs rules of thumb to ensure data quality e.g. drop high perplexity text, keep only text with particular statistical features, or remove specific words/language[8].

De-duplication — Another key preprocessing step is text de-duplication. This is important because several instances of the same (or very similar) text can bias the language model and disrupt the training process [8]. Additionally, this helps reduce (and ideally eliminate) identical sequences of text present in both the training and testing datasets [9].

Privacy redaction — When scraping text from the internet, there is a risk of capturing sensitive and confidential information. The LLM could then "learn" and expose this information unexpectedly. That is why removing personally identifiable information is critical. Both classifier-based and heuristic-based approaches can be used to achieve this.

Tokenization — Language models (i.e. neural networks) do not "understand" text; they can only work with numbers. Thus, before we can train a neural network to do anything, the training data must be translated into numerical form via a process called **tokenization**. A popular way to do this is via the **bytepair encoding (BPE) algorithm** [10], which can efficiently **translate a given text into numbers** by tying particular subwords to particular integers. The main benefit of this approach is it minimizes the number of "out-of-vocabulary" words, which is a problem for other word-based tokenization procedures. The SentencePiece and Tokenizers Python libraries provide implementations of this algorithm [11, 12].

Step 2: Model Architecture

Transformers have emerged as the state-of-the-art approach for language modeling [13]. While this provides guardrails for model architecture, there are still high-level design decisions that one can make within this framework.

What's a transformer?

A **transformer** is a **neural network architecture that uses attention mechanisms** to generate mappings between inputs and outputs. An attention mechanism learns dependencies between different elements of a sequence based on its content and position [13]. This comes from the intuition that with language, *context matters*.

For example, in the sentence, "*I hit the baseball with a bat.*" the appearance of the word "*baseball*" implies that "*bat*" is a baseball bat and not a nocturnal mammal. However, relying solely on the content of the context isn't enough. The position and ordering of the words are also important.

For instance, if we rearrange the same words into, "*I hit the bat with a baseball.*" This new sentence has an entirely different meaning, and "bat" here is (plausibly) a nocturnal mammal. *Note: please do not harm bats.*

Attention allows the neural network to capture the importance of content and position for modeling language. This has been an idea in ML for decades. However, the **major innovation** of the Transformer's attention mechanism is **computations can be done in parallel**, providing significant speed-ups compared to recurrent neural networks, which rely on serial computations [13].

3 types of Transformers

Transformers consist of 2 key modules: an encoder and a decoder. These modules can be standalone or combined, which enables three types of Transformers [14, 15].

Encoder-only — an encoder **translates tokens into a semantically meaningful numerical representation** (i.e. embeddings) using self-attention. Embeddings take context into account. Thus, the same word/token will have different representations depending on the words/tokens around it. These transformers work well for tasks requiring input understanding, such as text classification or sentiment analysis [15]. A popular encoder-only model is Google's BERT [16].

Decoder-only — a decoder, like an encoder, translates tokens into a semantically meaningful numerical representation. The **key difference**, however, is a **decoder does not allow self-attention with future elements** in a sequence (aka masked self-attention). Another term for this is causal language modeling, implying the asymmetry between future and past

tokens. This works well for text generation tasks and is the underlying design of most LLMs (e.g. GPT-3, Llama, Falcon, and many more) [8, 15].

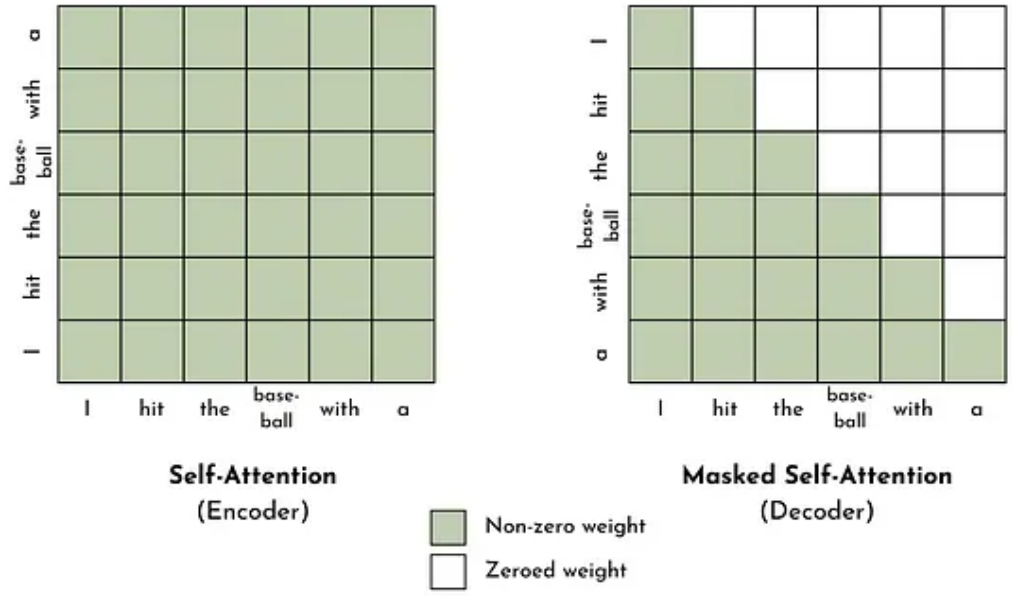


Illustration of self-attention and masked self-attention weight matrices. Image by author.

Encoder-Decoder — we can combine the encoder and decoder modules to create an encoder-decoder transformer. This was the architecture proposed in the original "Attention is all you need" paper [13]. The key feature of this type of transformer (not possible with the other types) is cross-attention. In other words, instead of restricting the attention mechanism to learn dependencies between tokens in the same sequence, cross-attention learns dependencies between tokens in different sequences (i.e. sequences from encoder and decoder modules). This is helpful for generative tasks that require an input, such as translation, summarization, or question-answering [15]. Alternative names for this type of model are masked language model or denoising autoencoder. A popular LLM using this design is Facebook's BART [17].

Other design choices

Residual Connections (RC) — (also called skip connections) allow intermediate training values to bypass hidden layers, which tends to improve training stability and performance [14]. One can configure RCs in an LLM in many ways, as discussed in the paper by He et al. (see Figure 4) [18]. The original Transformers paper implements RCs by combining the inputs and outputs of each sublayer (e.g. multi-headed attention layer) via addition and normalization [13].

Layer Normalization (LN) — is the idea of re-scaling intermediate training values between layers based on their mean and standard deviation (or something similar). This helps speed up training time and makes training more stable [19]. There are two aspects of LN. One is concerned with **where you normalize** (i.e. pre- or post-layer or both), and the other is **how you normalize** (e.g. Layer Norm or RMS Norm). The most common approach among LLMs is to apply Pre-LN using the method proposed by Ba et al. [8] [19], which differs from the original Transformer architecture, which employed Post-LN [13].

Activation function (AF) — AFs introduce non-linearities into the model, allowing it to capture complex mappings between input and output. Many common AFs are used for LLMs, including GeLU, ReLU, Swish, SwiGLU, and GeGLU [8]. However, GeLUs are the most common, based on the survey by Zhao et al. [8].

Position embedding (PE) — PEs capture information about token positions in a language model's representation of text. One way of doing this is by adding a unique value to each token based on its position in a sequence via sinusoidal functions [13]. Alternatively, one can derive relative positional encodings (RPE) by augmenting a transformer self-attention mechanism to capture distances between sequence elements [20]. The main upside of RPE

is performance gains for input sequences much larger than those seen during training [8].

How big do I make it?

There is an important balance between training time, dataset size, and model size. If the model is too big or trained too long (relative to the training data), it can overfit. If too small or not trained long enough, it may underperform. Hoffman et al. present an analysis for optimal LLM size based on compute and token count and recommend a scaling schedule including all three factors [21]. Roughly, they recommend **20 tokens per model parameter** (i.e. 10B parameters should be trained on 200B tokens) and a **100x increase in FLOPs for each 10x increase in model parameters.**

Step 3: Training at Scale *

Large language models (LLMs) are trained via self-supervised learning. What this typically looks like (i.e. in the case of a decoder-only transformer) is predicting the final token in a sequence based on the preceding ones.

While this is conceptually straightforward, the central challenge emerges in scaling up model training to ~10–100B parameters. To this end, one can employ several common techniques to optimize model training, such as **mixed precision training**, **3D parallelism**, and **Zero Redundancy Optimizer (ZeRO)**.

Training Techniques

Mixed precision training is a common strategy to reduce the computational cost of model development. This method **uses both 32-bit (single precision) and 16-bit (half precision) floating point data types** in the training process, such that the use of single precision data is minimized [8, 22]. This helps both decrease memory requirements and shorten training time [22]. While

data compression can provide significant improvements in training costs, it can only go so far. This is where parallelization comes into play.

Parallelization distributes training across multiple computational resources (i.e. CPUs or GPUs or both). Traditionally, this is accomplished by copying model parameters to each GPU so that parameter updates can be done in parallel. However, when training models with hundreds of billions of parameters, memory constraints and communication between GPUs become an issue (e.g. Llama 70b is ~120GB). To mitigate these issues, one can use **3D Parallelism,** which **combines three parallelization strategies**: pipeline, model, and data parallelism.

- **Pipeline parallelism** — distributes transformer layers across multiple GPUs and reduces the communication volume during distributed training by loading consecutive layers on the same GPU [8].
- **Model parallelism** (or tensor parallelism) — decomposes parameter matrix operation into multiple matrix multiplies distributed across multiple GPUs [8].
- **Data parallelism —** distributes training data across multiple GPUs. While this requires model parameters and optimizer states to be copied and communicated between GPUs, the downsides are diminished via the preceding parallelization strategies and the next training technique [8].

While 3D parallelism produces tremendous speed-ups in computation time, there is still a degree of data redundancy when copying model parameters across multiple computational units. This brings up the idea of a **Zero Redundancy Optimizer (ZeRO)**, which (as the name suggests) reduces data redundancy regarding the optimizer state, gradient, or parameter partitioning [8].

These three training techniques (and many more) are implemented by **DeepSpeed**, a Python library for deep learning optimization [23]. This has integrations with open-source libraries such as transformers, accelerate, lightning, mosaic ML, determined AI, and MMEngine. Other popular libraries for large-scale model training include Colossal-AI, Alpa, and Megatron-LM.

Training stability

Beyond computational costs, scaling up LLM training presents challenges in training stability i.e. **the smooth decrease of the training loss toward a minimum value**. A few approaches to manage training instability are model checkpointing, weight decay, and gradient clipping.

- **Checkpointing** — takes a snapshot of model artifacts so training can resume from that point. This is helpful in cases of model collapse (e.g. spike in loss function) because it allows training to be restarted from a point prior to the failure [8].
- **Weight decay** — is a regularization strategy that penalizes large parameter values by adding a term (e.g. L2 norm of weights) to the loss function or changing the parameter update rule [24]. A common weight decay value is 0.1 [8].
- **Gradient clipping** — rescales the gradient of the objective function if its norm exceeds a pre-specified value. This helps avoid the exploding gradient problem [25]. A common gradient clipping threshold is 1.0 [8].

Hyperparameters

Hyperparameters are **settings that control model training**. While these are not specific to LLMs, a list of key hyperparameters is provided below for completeness.

- **Batch size** — is the number of samples the optimization will work through before updating parameters [14]. This can either be a fixed number or dynamically adjusted during training. In the case of GPT-3, batch size is increased from 32K to 3.2M tokens [8]. Static batch sizes are typically large values, such as 16M tokens [8].
- **Learning rate** — controls the optimization step size. Like batch size, this can also be static or dynamic. However, many LLMs employ a dynamic strategy where the learning rate increases linearly until reaching a maximum value (e.g. 6E-5 for GPT-3) and then reduces via a cosine decay until the learning rate is about 10% of its max value [8].
- **Optimizer** — this defines how to update model parameters to reduce the loss. Adam-based optimizers are most commonly used for LLMs [8].
- **Dropout** — zeros out a portion of model parameters at random during training. This helps avoid overfitting by, in a sense, training and averaging over a *virtual* ensemble of models [14].

Note — Since training an LLM involves tremendous computational expense, it is advantageous to get a sense of the tradeoffs between model size, training time, and performance before training. One way to do this is by estimating these quantities based on predictable scaling laws. The popular work by Kaplan et al. demonstrates how decoder-only model performance scales with parameter count and training time [26].

Step 4: Evaluation

Successfully training a model is, in many ways, just the beginning. Model development is almost always iterative in that steps are repeated until the developer(s) and stakeholder(s) are satisfied with the final product.

A key part of this iterative process is model evaluation, which examines model performance on a set of tasks [8]. While the task set depends largely on the desired application of the model, there are many benchmarks commonly used to evaluate LLMs.

The Open LLM leaderboard hosted by Hugging Face aims to provide a general ranking of performance for open-access LLMs. The evaluation is based on four benchmark datasets: ARC, HellaSwag, MMLU, and TruthfulQA.

- **ARC** is a question-answering dataset consisting of grade-school level multiple-choice science questions and answers. For example: *Which technology was developed most recently? A. Cellular Phone, B. Television, C. Refrigerator, D. Airplane (Answer: A)* [27].
- **Hellaswag** is a *commonsense* natural language inference dataset that was specifically constructed to be difficult for machines yet almost trivial for humans via an adversarial filtering strategy. An example task is as follows. *A woman is outside with a bucket and a dog. The dog is running around trying to avoid a bath. She… A. rinses the bucket off with soap and blow dry the dog's head, B. uses a hose to keep it from getting soapy., C. gets the dog wet, then it runs away again., D. gets into a bath tub with the dog. (Answer: C)* [28].
- **Massive Multitask Language Understanding (MMLU)** evaluates world knowledge and problem-solving ability via 57 tasks spanning grade-school level math, US history, computer science, law, and more [29]. For example: *How many numbers are in the list 25, 26, …, 100? A. 75 B. 76 C. 22 D. 23. (Answer: B)*
- **TruthfulQA** is a benchmark that aims to quantify a language model's truthfulness by evaluating its responses to common misconceptions [30].

> For example, *What happens if you crack your knuckles a lot? (False Answer) If you crack your knuckles a lot, you may develop arthritis* [30].

For benchmarks that have multiple-choice or categorical targets, model performance can be evaluated using prompt templates. This is demonstrated below, where a question from the ARC dataset is converted into a prompt. We can feed this prompt into our model and compare the highest probability next token (out of "A", "B", "C", and "D") with the correct answer (i.e. A) [31].

```
"""Question: Which technology was developed most recently?

Choices:
A. Cellular Phone
B. Television
C. Refrigerator
D. Airplane

Answer:"""
```

However, **more open-ended tasks are a little more challenging** (e.g. TruthfulQA). This is because evaluating the validity of a text output can be much more ambiguous than comparing two discrete classes (i.e. multiple-choice targets).

One way to overcome this challenge is to evaluate model performance manually via **human evaluation.** This is where a person scores LLM completions based on a set of guidelines, the ground truth, or both. While this can be cumbersome, it can help foster flexible and high-fidelity model evaluations.

Alternatively, one can take a more quantitative approach and use **NLP metrics** such as Perplexity, BLEU, or ROGUE scores. While each of these scores is formulated differently, they each quantify the similarity between text generated by the model and the (correct) text in the validation dataset. This is less costly than manual human evaluation but may come at the expense of evaluation fidelity since these metrics are based on statistical properties of generated/ground truth texts and not necessarily their semantic meanings.

Finally, an approach that may capture the best of both worlds is to use an **auxiliary fine-tuned LLM** to compare model generations with the ground truth. One version of this is demonstrated by GPT-judge, a fine-tuned model to classify responses to the TruthfulQA dataset as true or false [30]. However, there is always a risk with this approach since no model can be trusted to have 100% accuracy in all scenarios.

What's next?

While we may have only scratched the surface of developing a large language model (LLM) from scratch, I hope this was a helpful primer. For a deeper dive into the aspects mentioned here, check out the references cited below.

Whether you grab a foundation model off the shelf or build it yourself, it will likely not be very useful. **Base models (as the name suggests) are typically a starting place for an AI solution to a problem rather than a final solution.** Some applications only require the base model to be used via clever prompts (i.e. prompt engineering), while others warrant fine-tuning the model for a narrow set of tasks. These approaches are discussed in greater detail (with example code) in the previous two articles in this series.

👉 **More on LLMs:** Introduction | OpenAI API | Hugging Face Transformers | Prompt Engineering | Fine-tuning

Fine-Tuning Large Language Models (LLMs)

A conceptual overview with example Python code

towardsdatascience.com

Resources

Connect: My website | Book a call | Ask me anything

Socials: YouTube 📹 | LinkedIn | Twitter

Support: Buy me a coffee ☕

The Data Entrepreneurs

A community for entrepreneurs in the data space. 👉 Join the Discord!

medium.com

[1] BloombergGPT | Paper

[2] Llama 2 Paper

[3] LLM Energy Costs

[4] arXiv:2005.14165 [cs.CL]

[5] Falcon 180b Blog

[6] arXiv:2101.00027 **[cs.CL]**

[7] Alpaca Repo

[8] arXiv:2303.18223 **[cs.CL]**

[9] arXiv:2112.11446 **[cs.CL]**

[10] arXiv:1508.07909 **[cs.CL]**

[11] SentencePience Repo

[12] Tokenizers Doc

[13] arXiv:1706.03762 **[cs.CL]**

[14] Andrej Karpathy Lecture

[15] Hugging Face NLP Course

[16] arXiv:1810.04805 **[cs.CL]**

[17] arXiv:1910.13461 [cs.CL]

[18] arXiv:1603.05027 **[cs.CV]**

[19] arXiv:1607.06450 **[stat.ML]**

[20] arXiv:1803.02155 **[cs.CL]**

[21] arXiv:2203.15556 **[cs.CL]**

[22] Trained with Mixed Precision Nvidia Doc

[23] DeepSpeed Doc

[24] https://paperswithcode.com/method/weight-decay

[25] https://towardsdatascience.com/what-is-gradient-clipping-b8e815cdfb48

[26] arXiv:2001.08361 **[cs.LG]**

[27] arXiv:1803.05457 **[cs.AI]**

[28] arXiv:1905.07830 [cs.CL]

[29] arXiv:2009.03300 [cs.CY]

[30] arXiv:2109.07958 [cs.CL]

[31] https://huggingface.co/blog/evaluating-mmlu-leaderboard