Large Language Model Foundations - Technical Deep Dive

Transformer Architecture

What is a Transformer Model? | IBM What is a transformer model? Tags Artificial Intelligence 28 March 2025 Link copied Authors Cole Stryker Editorial Lead, AI Models Dave Bergmann Senior Writer, AI Models IBM What is a transformer model? The transformer model is a type of neural network architecture that excels at processing sequential data, most prominently associated with large language models (LLMs). Transformer models have also achieved elite performance in other fields of artificial intelligence (AI), such as computer vision, speech recognition and time series forecasting. The transformer architecture was first described in the seminal 2017 paper "Attention is All You Need" by Vaswani and others, which is now considered a watershed moment in deep learning. Originally introduced as an evolution of the recurrent neural network (RNN)-based sequence-to-sequence models used for machine translation, transformer-based models have since attained cutting-edge advancements across nearly every machine learning (ML) discipline. Despite their versatility, transformer models are still most commonly discussed in the context of natural language processing (NLP) use cases, such as chatbots, text generation, summarization, question answering and sentiment analysis. The BERT (or Bidirectional Encoder Representations from Transformers) encoder-decoder model, introduced by Google in 2019, was a major landmark in the establishment of transformers and remains the basis of most modern word embedding applications, from modern vector databases to Google search. Autoregressive decoder-only LLMs, such as the GPT-3 (short for Generative Pre-trained Transformer) model that powered the launch of OpenAI's ChatGPT, catalyzed the modern era of generative AI (gen AI). The ability of transformer models to intricately discern how each part of a data sequence influences and correlates with the others also lends them many multimodal uses. For instance, vision transformers (ViTs) often exceed the performance of convolutional neural networks (CNNs) on image segmentation, object detection and related tasks. The transformer architecture also powers many diffusion models used for image generation, multimodal text-to-speech (TTS) and vision language models (VLMs). The latest AI News + Insights Discover expertly curated insights and news on AI, cloud and more in the weekly Think Newsletter. Subscribe today Why are transformer models important? The central feature of transformer models is their self-attention mechanism, from which transformer models derive their impressive ability to detect the relationships (or dependencies) between each part of an input sequence. Unlike the RNN and CNN architectures that preceded it, the transformer architecture uses only attention layers and standard feedforward layers. The benefits of self-attention, and specifically the multi-head attention technique that transformer models employ to compute it, are what enable transformers to exceed the performance of the RNNs and CNNs that had previously been state-of-the-art. Before the introduction of transformer models, most NLP tasks relied on recurrent neural networks (RNNs). The way RNNs process sequential data is inherently serialized: they ingest the elements of an input sequence one at a time and in a specific order. This hinders the ability of RNNs to

capture long-range dependencies, meaning RNNs can only process short text sequences effectively. This deficiency was somewhat addressed by the introduction of long short term memory networks (LSTMs), but remains a fundamental shortcoming of RNNs. Attention mechanisms, conversely, can examine an entire sequence simultaneously and make decisions about how and when to focus on specific time steps of that sequence. In addition to significantly improving the ability to understand long-range dependencies, this quality of transformers also allows for parallelization: the ability to perform many computational steps at once, rather than in a serialized manner. Being well-suited to parallelism enables transformer models to take full advantage of the power and speed offered by GPUs during both training and inference. This possibility, in turn, unlocked the opportunity to train transformer models on unprecedentedly massive datasets through self-supervised learning. Especially for visual data, transformers also offer some advantages over convolutional neural networks. CNNs are inherently local, using convolutions to process smaller subsets of input data one piece at a time. Therefore, CNNs also struggle to discern long-range dependencies, such as correlations between words (in text) or pixels (in images) that aren't neighboring one another. Attention mechanisms don't have this limitation. Mixture of Experts | 18 July, episode 64 Decoding AI: Weekly News Roundup Join our world-class panel of engineers, researchers, product leaders and more as they cut through the AI noise to bring you the latest in AI news and insights. Watch the latest podcast episodes What is self-attention? Understanding the mathematical concept of attention, and more specifically self-attention, is essential to understanding the success of transformer models in so many fields. Attention mechanisms are, in essence, algorithms designed to determine which parts of a data sequence an AI model should "pay attention to" at any particular moment. Consider a language model interpreting the English text "on Friday, the judge issued a sentence. " The preceding word "the " suggests that "judge " is acting as a noun-as in, a person presiding over a legal trial-rather than a verb meaning to appraise or form an opinion.That context for the word "judge " suggests that "sentence " probably refers to a legal penalty, rather than a grammatical "sentence."The word "issued " further implies that "sentence " refers to the legal concept, not the grammatical concept.Therefore, when interpreting the word "sentence ," the model should pay close attention to "judge " and "issued. " It should also pay some attention to the word "the ." It can more or less ignore the other words. How does self-attention work? Broadly speaking, a transformer model's attention layers assess and use the specific context of each part of a data sequence in 4 steps: The model "reads" raw data sequences and converts them into vector embeddings, in which each element in the sequence is represented by its own feature vector(s) that numerically reflect qualities such as semantic meaning. The model determines similarities, correlations and other dependencies (or lack thereof) between each vector and each other vector. In most transformer models, the relative importance of one vector to another is determined by computing the dot product between each vector. If the vectors are well aligned, multiplying them together will yield a large value. If they're not aligned, their dot product will be small or negative. These "alignment scores" are converted into attention weights. This is achieved by using alignment scores as inputs to a softmax activation function, which normalizes all values to a range between 0-1 such that they

all add up to a total of 1. So for instance, assigning an attention weight of 0 between "Vector A" and "Vector B" means that Vector B should be ignored when making predictions about Vector A. Assigning Vector B an attention weight of 1 means that it should receive 100% of the model's attention when making decisions about Vector A. These attention weights are used to emphasize or deemphasize the influence of specific input elements at specific times. In other words, attention weights help transformer models focus on or ignore specific information at a specific moment. Before training, a transformer model doesn't yet "know" how to generate optimal vector embeddings and alignment scores. During training, the model makes predictions across millions of examples drawn from its training data, and a loss function quantifies the error of each prediction. Through an iterative cycle of making predictions and then updating model weights through backpropagation and gradient descent, the model "learns" to generate vector embeddings, alignment scores and attention weights that lead to accurate outputs. How do transformer models work? Transformer models such as relational databases generate query, key and value vectors for each part of a data sequence, and use them to compute attention weights through a series of matrix multiplications. Relational databases are designed to simplify the storage and retrieval of relevant data: they assign a unique identifier ("key") to each piece of data, and each key is associated with a corresponding value. The "Attention is All You Need" paper applied that conceptual framework to processing the relationships between each token in a sequence of text. The query vector represents the information a specific token is "seeking." In other words, a token's query vector is used to compute how other tokens might influence its meaning, conjugation or connotations in context.The key vectors represent the information that each token contains. Alignment between query and key is used to compute attention weights that reflect how relevant they are in the context of that text sequence.The value (or value vector) "returns" the information from each key vector, scaled by its respective attention weight. Contributions from keys that are strongly aligned with a query are weighted more heavily; contributions from keys that are not relevant to a query will be weighted closer to zero. For an LLM, the model's "database" is the vocabulary of tokens it has learned from the text samples in its training data. Its attention mechanism uses information from this "database" to understand the context of language. Tokenization and input embeddings Whereas characters-letters, numbers or punctuation marks-are the base unit we humans use to represent language, the smallest unit of language that AI models use is a token. Each token is assigned an ID number, and these ID numbers (rather than the words or even the tokens themselves) are the way LLMs navigate their vocabulary "database." This tokenization of language significantly reduces the computational power needed to process text. To generate query and key vectors to feed into the transformer's attention layers, the model needs an initial, contextless vector embedding for each token. These initial token embeddings can be either learned during training or taken from a pretrained word embedding model. Positional encoding The order and position of words can significantly impact their semantic meanings. Whereas the serialized nature of RNNs inherently preserves information about the position of each token, transformer models must explicitly add positional information for the attention mechanism to consider. With positional encoding, the model adds a vector of values to each token's embedding, derived

from its relative position, before the input enters the attention mechanism. The nearer the 2 tokens are, the more similar their positional vectors will be and therefore, the more their alignment score will increase from adding positional information. The model thereby learns to pay greater attention to nearby tokens. Generating query, key and value vectors When positional information has been added, each updated token embedding is used to generate three new vectors. These query, key and value vectors are generated by passing the original token embeddings through each of three parallel feedforward neural network layers that precede the first attention layer. Each parallel subset of that linear layer has a unique matrix of weights, learned through self-supervised pretraining on a massive dataset of text. The embeddings are multiplied by the weight matrix WQ to yield the query vectors (Q), which have dk dimensionsThe embeddings are multiplied by the weight matrix WK to yield the key vector (K), also with dimensions dk The embeddings are multiplied by the weight matrix WV to yield the value vectors (V), with dimensions dv Computing self-attention The transformer's attention mechanism's primary function is to assign accurate attention weights to the pairings of each token's query vector with the key vectors of all the other tokens in the sequence. When achieved, you can think of each token x as now having a corresponding vector of attention weights, in which each element of that vector represents the extent to which some other token should influence it. Each other token's value vector is now multiplied by its respective attention weight.These attention-weighted value vectors are all summed together. The resulting vector represents the aggregated contextual information being provided to token x by all of the other tokens in the sequence.Finally, the resulting vector of attention-weighted changes from each token is added to the token x's original, post-positional encoding vector embedding. In essence, x's vector embedding has been updated to better reflect the context provided by the other tokens in the sequence. Multi-head attention To capture the many multifaceted ways tokens might relate to one another, transformer models implement multi-head attention across multiple attention blocks. Before being fed into the first feedforward layer, each original input token embedding is split into h evenly sized subsets. Each piece of the embedding is fed into one of h parallel matrices of Q, K and V weights, each of which are called a query head, key head or value head. The vectors output by each of these parallel triplets of query, key and value heads are then fed into a corresponding subset of the next attention layer, called an attention head. In the final layers of each attention block, the outputs of these h parallel circuits are eventually concatenated back together before being sent to the next feedforward layer. In practice, model training results in each circuit learning different weights that capture a separate aspect of semantic meanings. Residual connections and layer normalization In some situations, passing along the contextually-updated embedding output by the attention block might result in an unacceptable loss of information from the original sequence. To address this, transformer models often balance the contextual information provided by the attention mechanism with the original semantic meaning of each token. After the attention-updated subsets of the token embedding have all been concatenated back together, the updated vector is then added to the token's original (position-encoded) vector embedding. The original token embedding is supplied by a residual connection between that layer and an earlier layer of the network. The resulting vector is fed into

another linear feedforward layer, where it's normalized back to a constant size before being passed along to the next attention block. Together, these measures help preserve stability in training and help ensure that the text's original meaning is not lost as the data moves deeper into the neural network. Generating outputs Eventually, the model has enough contextual information to inform its final outputs. The nature and function of the output layer will depend on the specific task the transformer model has been designed for. In autoregressive LLMs, the final layer uses a softmax function to determine the probability that the next word will match each token in its vocabulary "database." Depending on the specific sampling hyperparameters, the model uses those probabilities to determine the next token of the output sequence. Transformer models in natural language processing (NLP) Transformer models are most commonly associated with NLP, having originally been developed for machine translation use cases. Most notably, the transformer architecture gave rise to the large language models (LLMs) that catalyzed the advent of generative AI. Most of the LLMs that the public is most familiar with, from closed source models such as OpenAI's GPT series and Anthropic's Claude models to open source models including Meta Llama or IBM Granite, are autoregressive decoder-only LLMs. Autoregressive LLMs are designed for text generation, which also extends naturally to adjacent tasks such as summarization and question answering. They're trained through self-supervised learning, in which the model is provided the first word of a text passage and tasked with iteratively predicting the next word until the end of the sequence. Information provided by the self-attention mechanism enables the model to extract context from the input sequence and maintain the coherence and continuity of its output. Encoder-decoder masked language models (MLMs), such as BERT and its many derivatives, represent the other main evolutionary branch of transformer-based LLMs. In training, an MLM is provided a text sample with some tokens masked-hidden-and tasked with completing the missing information. While this training methodology is less effective for text generation, it helps MLMs excel at tasks requiring robust contextual information, such as translation, text classification and learning embeddings. Transformer models in other fields Though transformer models were originally designed for, and continue to be most prominently associated with natural language use cases, they can be used in nearly any situation involving sequential data. This has led to the development of transformer-based models in other fields, from fine-tuning LLMs into multimodal systems to dedicated time series forecasting models and ViTs for computer vision. Some data modalities are more naturally suited to transformer-friendly sequential representation than others. Time series, audio and video data are inherently sequential, whereas image data is not. Despite this, ViTs and other attention-based models have achieved state-of-the-art results for many computer vision tasks, including image captioning, object detection, image segmentation and visual question answering. To use transformer models for data not conventionally thought of as "sequential" requires a conceptual workaround to represent that data as a sequence. For instance, to use attention mechanisms to understand visual data, ViTs use patch embeddings to make image data interpretable as sequences. First, an image is split into an array of patches. For instance, a 224x224 pixel image can be subdivided into 256 14x14 pixel patches, dramatically reducing the number of computational steps required to process the image.Next, a linear projection layer maps

each patch to a vector embedding.Positional information is added to each of these patch embeddings, akin to the positional encoding described earlier in this article.These patch embeddings can now essentially function as a sequence of token embeddings, allowing the image to be interpreted by an attention mechanism. Ebook Unlock the power of generative AI + ML Learn how to confidently incorporate generative AI and machine learning into your business. Read the ebook Resources Explainer Neural networks from scratch Get an in-depth understanding of neural networks, their basic functions and the fundamentals of building one. Read the article Report IBM is named a Leader in Data Science & Machine Learning Learn why IBM has been recognized as a Leader in the 2025 Gartner Magic QuadrantTM for Data Science and Machine Learning Platforms. Read the report AI models Explore IBM Granite IBM GraniteTM is our family of open, performant and trusted AI models, tailored for business and optimized to scale your AI applications. Explore language, code, time series and guardrail options. Meet Granite Report AI in Action 2024 We surveyed 2,000 organizations about their AI initiatives to discover what's working, what's not and how you can get ahead. Read the report Ebook Unlock the power of generative AI and ML Learn how to confidently incorporate generative AI and machine learning into your business. Read the ebook Ebook How to choose the right foundation model Learn how to select the most suitable AI foundation model for your use case. Read the ebook Guide The CEO's guide to generative AI Learn how CEOs can balance the value generative AI can create against the investment it demands and the risks it introduces. Read the guide Guide Put AI to work: Driving ROI with gen AI Want to get a better return on your AI investments? Learn how scaling gen AI in key areas drives change by helping your best minds build and deliver innovative new solutions. Read the guide Related solutions IBM watsonx.ai Train, validate, tune and deploy generative AI, foundation models and machine learning capabilities with IBM watsonx.ai, a next-generation enterprise studio for AI builders. Build AI applications in a fraction of the time with a fraction of the data. Explore watsonx.ai AI for developers Move your applications from prototype to production with the help of our AI development solutions. Explore AI development tools AI consulting and services Reinvent critical workflows and operations by adding AI to maximize experiences, real-time decision-making and business value. Explore AI services Take the next step Get one-stop access to capabilities that span the AI development lifecycle. Produce powerful AI solutions with user-friendly interfaces, workflows and access to industry-standard APIs and SDKs. Explore watsonx.ai Book a live demo Footnotes 1 Google's BERT Rolls Out Worldwide (link resides outside ibm.com), Search Engine Journal, Dec 9, 2019

Pre-training Methodologies

Pre-training in LLM Development SolutionsDatasetsResearchResourcesCompanyTalk to usLog inLog inLog inToloka welcomes new investors Bezos Expeditions and Mikhail Parakhin in strategic funding round Learn moreToloka welcomes new investors Bezos Expeditions and Mikhail Parakhin in strategic funding round Learn moreToloka welcomes new investors Bezos Expeditions and Mikhail Parakhin in strategic funding round Learn morePre-training in LLM DevelopmentToloka TeamFebruary 22, 2024February 22, 2024Essential ML GuideEssential ML GuideCan your AI agent survive in the real world?Can your AI agent survive in the real

world?Can your AI agent survive in the real world?Training datasets are what it needs to reason, adapt, and act in unpredictable environmentsTraining datasets are what it needs to reason, adapt, and act in unpredictable environmentsTraining datasets are what it needs to reason, adapt, and act in unpredictable environmentsGet traning dataGet traning dataNatural Language Processing (NLP) has been revolutionized by the advent of pre-trained models. Most of today's best-known LLMs are pre-trained. Why is it so critical to pre-train AI models, and LLMs in particular? And what other indispensable steps should be taken to obtain effective language models that comprehend user preferences and can fulfill your business-specific tasks? We'll figure it out further in our article.What is pre-training?Pre-training involves training a neural network model on a large corpus of text data in an unsupervised manner. It is an initial phase of machine learning training that is a crucial step to equip an LLM with general language understanding capabilities. After pre-training it can be fine-tuned to accomplish the desired results.By leveraging past experiences rather than starting from scratch, language models are able to effectively address new tasks during fine-tuning, which results in a model benefiting from previous training.Humans possess similar inherent abilities to leverage prior knowledge, allowing us to avoid starting from scratch when faced with new challenges.However, pre-trained models have some core knowledge and are fully capable of undertaking a wide range of tasks, yet they do not possess any kind of specialization. To master proficiency levels in conversational skills, text generation, or creating other content on request the LLM requires several more stages of learning.Pre-training in LLMsPre-trained LLMs are not yet suitable for use in highly specialized areas since they do not have in-depth contextual knowledge in certain areas. Supervised fine-tuning (SFT) and Reinforcement Learning from Human Feedback (RLHF) are often called for to make a pre-trained model suitable for such a niche use.However, if the model was not pre-trained on large-scale unlabeled text corpora collected from various sources such as books, articles, websites, social media, and other textual resources, it would be even more time-consuming and resourceful to train it for specific needs.Fine-tuning is a supervised learning process, requiring a relatively small set of labeled data. The use of a model that is already pre-trained is precisely what makes it possible to employ a relatively small set of labeled data for additional training. That is, if you were to take a non-pre-trained deep learning model, you would have to collect a huge dataset to train it to do what you want it to do. Since the model already has the initial knowledge, it is easy to fine tune it on a smaller amount of data.Training a non-pre-trained deep learning model for a special use case from scratch would require more data, training time and resources. Consequently, the quickest and most cost-effective way to an improved model performance requires pre-training of LLM.Steps for LLM pre-trainingDuring the pre-training stage in ML, the model is learning to foretell the next word in the text in a mindful manner. It is called the model's pre-training objective. Pre-trained LLM cannot recognize the instructions or questions that are given to it yet. The SFT and RLHF steps are necessary to adapt it to a real-world AI application, for example, for using it as a chatbot. As we've already mentioned, pre-training helps to complete these steps faster and at a lower cost. Here is a step-by-step breakdown of how it can be implemented.Data CollectionData scientists

gather a large and diverse corpus of text data from various sources such as books, articles, websites, and social media. The diversity of the data helps the model learn a wide range of language patterns and concepts.CleaningText data often contains noise, such as special characters or non-textual elements. Cleaning involves removing or replacing any non-textual elements and duplicated text samples from the raw data to ensure that it is consistent and meaningful. Techniques such as regular code scripts, artificial intelligence algorithms that are trained to automatically identify and clean data, and a final human review can be used for general text cleaning.TokenizationData scientists and NLP engineers tokenize the text data into smaller units such as words, subwords, or characters to form the input sequences for the machine learning model. Word tokenization is perhaps the most common form of tokenization, where the text is split into individual words based on whitespace or punctuation boundaries.Architecture SelectionA transformer-based architecture is often chosen for the model, as it works well with sequences in texts. Transformers have proven to be highly effective for natural language processing tasks due to their attention mechanism. It allows the model to weigh the importance of each word/token in the input sequence when computing representations. This enables the model to capture dependencies between distant words more effectively than traditional architectures.Pre-training processMachine learning engineers train the LLM according to its pre-training objective using the tokenized and preprocessed data. This step involves feeding a large dataset as input through the model to make it comprehend and create human-like sequences of text.After pre-trainingOnce all the stages of pre-training are complete, the pre-trained model is ready to be fine-tuned and go through the RLFH or SFT phase. There are certain differences between those approaches but they both require broad datasets of preferences that could be used to teach the model which answers are preferable in a given context. Unlike data for pretraining these preference datasets are smaller yet harder to obtain. Read our article on SFT to learn more or simply get in touch, if you need high-quality dataset for RLHF or SFT.The performance of the pre-trained and fine-tuned model is evaluated on various benchmarks and tasks to assess its generalization ability and effectiveness in understanding and generating human-like text. This so-called continuous model evaluation also incorporates human assessment of model quality.The Importance of Pre-TrainingPre-training in machine learning is important due to its numerous benefits. Here are some key reasons why pre-training is important:Transfer learningThe availability of pre-trained models enables a technique known as transfer learning, where a model trained on one task or dataset is employed to improve performance on a related task or dataset. Fine-tuning is a type of transfer learning that involves updating the parameters of the entire pre-trained model. Instead of training a model from scratch, transfer learning allows knowledge from pre-trained models to be reused, resulting in faster training and better performance on target tasks.Data Efficiency and Lower Training CostPre-training enables models to leverage large amounts of unlabeled data, which is often more abundant and accessible than labeled data. This reduces the need for extensive labeled data for training models on target tasks, making it feasible to train effective models even with limited labeled data. Such approach lowers the overall annotation costs associated with training machine learning models.Easy Customization for Specific TasksPre-training provides a flexible

starting point for adapting models to address specific tasks or domains. By fine-tuning pre-trained models on domain-relevant datasets or objectives, ML practitioners can tailor the output to the nuances and requirements of the target application.This process can happen more seamlessly than if one had to train a model for a narrow specialization from scratch. Fine-tuning and subsequent RLHF, which helps improve the model's output quality through human feedback, leads to improved performance and relevance.Numerous Use CasesPre-trained models serve as the basis for many AI applications, particularly in the field of NLP. Thanks to pre-training LLMs can be adjusted with fine-tuning or/and RLHF to:determine the sentiment expressed in texts;translate from one to several other languages;identify and classify named entities;produce realistic and diverse texts in various styles and genres;generate content for chatbots and conversational agents.Pre-Training: An Important Step of ML Training That Cannot Exist On Its OwnHowever effective and valuable the pre-training model may be, it cannot serve as an accomplished LLM because it does not have all the necessary properties that fine-tuning and RLFH provide. Also, another vital step for LLM output assessment called continuous model evaluation is introduced after all of the training stages are through.So, if you need to develop a truly customized and high-performing model, only pre-training wouldn't be enough. You need to fine-tune your LMM as well and introduce other steps like RLFH and model output evaluation. By incorporating these stages into the training pipeline, developers can create customized language models that meet the specific requirements and objectives of their applicationsThese additional steps complement pre-training and ensure that the model not only possesses the necessary knowledge but also adapts to changing conditions, learns from human feedback, and maintains high performance over time.Read more about other stages of LLM development:LLM training and fine-tuningWhy RLHF is the key to improving LLM-based solutionsEvaluating Large Language ModelsSubscribe to Toloka NewsCase studies, product news, and other articles straight to your inbox.SubscribeSubscribe to Toloka NewsCase studies, product news, and other articles straight to your inbox.SubscribeSubscribe to Toloka NewsCase studies, product news, and other articles straight to your inbox.SubscribeBack to topRecent articlesView all articlesDetecting hidden harm in long contexts: How Toloka built AWS Bedrock's advanced safety datasetJul 14, 2025Does Your Agent Work? AI Agent Benchmarks ExplainedJul 7, 2025What is data governance for AI, and why does it matter?Jul 4, 2025Detecting hidden harm in long contexts: How Toloka built AWS Bedrock's advanced safety datasetJul 14, 2025Does Your Agent Work? AI Agent Benchmarks ExplainedJul 7, 2025What is data governance for AI, and why does it matter?Jul 4, 2025LLM evaluation framework: principles, practices, and toolsJul 3, 2025More about TolokaWhat is Toloka's mission?Where is Toloka located?What is Toloka's key area of expertise?How long has Toloka been in the AI market?How does Toloka ensure the quality and accuracy of the data collected?How does Toloka source and manage its experts and AI tutors?What types of projects or tasks does Toloka typically handle?What industries and use cases does Toloka focus on?What is Toloka's mission?Where is Toloka located?What is Toloka's key area of expertise?How long has Toloka been in the AI market?How does Toloka ensure the quality and accuracy of the data collected?How does Toloka source and manage its experts and AI tutors?What types of projects or tasks does Toloka typically handle?What industries and use cases does Toloka

focus on?What is Toloka's mission?Where is Toloka located?What is Toloka's key area of expertise?How long has Toloka been in the AI market?How does Toloka ensure the quality and accuracy of the data collected?How does Toloka source and manage its experts and AI tutors?What types of projects or tasks does Toloka typically handle?What industries and use cases does Toloka focus on?SOLUTIONSData for text & reasoning skilsData for AI agentsData for Coding ModelsData for creative AIAI EvaluationAI Safety & Red TeamingHuman - synthetic dataMeet our expertsDATASETSOff-the-shelf DatasetsMultimodal ConversationsUniversity-level Math ReasoningRESEARCHToloka ResearchResponsible AIAI text detectionMath reasoningVQA benchmarkReSourcesBlogEventsCase StudiesSecurity and PrivacyCompanyAbout UsNewsroomCareersBrand Guidelines 2025 Toloka AI BVManage cookiesPrivacy NoticeTerms of UseCode of ConductSOLUTIONSData for text & reasoning skilsData for AI agentsData for Coding ModelsData for creative AIAI EvaluationAI Safety & Red TeamingHuman - synthetic dataMeet our expertsDATASETSOff-the-shelf DatasetsMultimodal ConversationsUniversity-level Math ReasoningRESEARCHToloka ResearchResponsible AIAI text detectionMath reasoningVQA benchmarkReSourcesBlogEventsCase StudiesSecurity and PrivacyCompanyAbout UsNewsroomCareersBrand Guidelines 2025 Toloka AI BVManage cookiesPrivacy NoticeTerms of UseCode of ConductSOLUTIONSData for text & reasoning skilsData for AI agentsData for Coding ModelsData for creative AIAI EvaluationAI Safety & Red TeamingHuman - synthetic dataMeet our expertsDATASETSOff-the-shelf DatasetsMultimodal ConversationsUniversity-level Math ReasoningRESEARCHToloka ResearchResponsible AIAI text detectionMath reasoningVQA benchmarkReSourcesBlogEventsCase StudiesSecurity and PrivacyCompanyAbout UsNewsroomCareersBrand Guidelines 2025 Toloka AI BVManage cookiesPrivacy NoticeTerms of UseCode of Conduct

Attention Mechanisms

Attention (machine learning) - Wikipedia Jump to content From Wikipedia, the free encyclopedia Machine learning technique Part of a series onMachine learningand data mining Paradigms Supervised learning Unsupervised learning Semi-supervised learning Self-supervised learning Reinforcement learning Meta-learning Online learning Batch learning Curriculum learning Rule-based learning Neuro-symbolic AI Neuromorphic engineering Quantum machine learning Problems Classification Generative modeling Regression Clustering Dimensionality reduction Density estimation Anomaly detection Data cleaning AutoML Association rules Semantic analysis Structured prediction Feature engineering Feature learning Learning to rank Grammar induction Ontology learning Multimodal learning Supervised learning(classification * regression) Apprenticeship learning Decision trees Ensembles Bagging Boosting Random forest k-NN Linear regression Naive Bayes Artificial neural networks Logistic regression Perceptron Relevance vector machine (RVM) Support vector machine (SVM) Clustering BIRCH CURE Hierarchical k-means Fuzzy Expectation-maximization (EM) DBSCAN OPTICS Mean shift Dimensionality reduction Factor analysis CCA ICA LDA NMF PCA PGD t-SNE SDL Structured prediction Graphical models Bayes net Conditional random field Hidden Markov Anomaly detection RANSAC k-NN Local outlier factor Isolation forest Neural networks Autoencoder Deep learning Feedforward neural network Recurrent neural network LSTM GRU ESN reservoir computing Boltzmann

machine Restricted GAN Diffusion model SOM Convolutional neural network U-Net LeNet AlexNet DeepDream Neural field Neural radiance field Physics-informed neural networks Transformer Vision Mamba Spiking neural network Memtransistor Electrochemical RAM (ECRAM) Reinforcement learning Q-learning Policy gradient SARSA Temporal difference (TD) Multi-agent Self-play Learning with humans Active learning Crowdsourcing Human-in-the-loop Mechanistic interpretability RLHF Model diagnostics Coefficient of determination Confusion matrix Learning curve ROC curve Mathematical foundations Kernel machines Bias-variance tradeoff Computational learning theory Empirical risk minimization Occam learning PAC learning Statistical learning VC theory Topological deep learning Journals and conferences AAAI ECML PKDD NeurIPS ICML ICLR IJCAI ML JMLR Related articles Glossary of artificial intelligence List of datasets for machine-learning research List of datasets in computer vision and image processing Outline of machine learning vte Attention mechanism, overview In machine learning, attention is a method that determines the importance of each component in a sequence relative to the other components in that sequence. In natural language processing, importance is represented by "soft" weights assigned to each word in a sentence. More generally, attention encodes vectors called token embeddings across a fixed-width sequence that can range from tens to millions of tokens in size. Unlike "hard" weights, which are computed during the backwards training pass, "soft" weights exist only in the forward pass and therefore change with every step of the input. Earlier designs implemented the attention mechanism in a serial recurrent neural network (RNN) language translation system, but a more recent design, namely the transformer, removed the slower sequential RNN and relied more heavily on the faster parallel attention scheme. Inspired by ideas about attention in humans, the attention mechanism was developed to address the weaknesses of using information from the hidden layers of recurrent neural networks. Recurrent neural networks favor more recent information contained in words at the end of a sentence, while information earlier in the sentence tends to be attenuated. Attention allows a token equal access to any part of a sentence directly, rather than only through the previous state. History[edit] See also: Timeline of machine learning Academic reviews of the history of the attention mechanism are provided in Niu et al.[1] and Soydaner.[2] 1950s 1960s Psychology biology of attention. cocktail party effect[3] - focusing on content by filtering out background noise. filter model of attention,[4] partial report paradigm, and saccade control.[5] 1980s sigma pi units,[6] higher order neural networks[7] 1990s fast weight controller.[8][9][10][11] Neuron weights generate fast "dynamic links" similar to keys & values.[12] 2014 seq2seq with RNN + Attention.[13] Attention mechanism was added onto RNN encoder-decoder architecture to improve language translation of long sentences. See Overview section. 2015 Attention applied to images [14][15][16] 2017 Transformers [17] = Attention + position encoding + MLP + skip connections. This design improved accuracy and removed the sequential disadvantages of the RNN. See also: Transformer (deep learning architecture) History Overview[edit] This article possibly contains original research. Please improve it by verifying the claims made and adding inline citations. Statements consisting only of original research should be removed. (June 2025) (Learn how and when to remove this message) The modern era of machine attention was revitalized by grafting an attention mechanism (Fig 1. orange) to an Encoder-Decoder.[citation needed] Animated

sequence of language translation Fig 1. Encoder-decoder with attention.[18] Numerical subscripts (100, 300, 500, 9k, 10k) indicate vector sizes while lettered subscripts i and i 1 indicate time steps. Pinkish regions in H matrix and w vector are zero values. See Legend for details. Legend Label Description 100 Max. sentence length 300 Embedding size (word dimension) 500 Length of hidden vector 9k, 10k Dictionary size of input & output languages respectively. x, Y 9k and 10k 1-hot dictionary vectors. x -> x implemented as a lookup table rather than vector multiplication. Y is the 1-hot maximizer of the linear Decoder layer D; that is, it takes the argmax of D's linear layer output. x 300-long word embedding vector. The vectors are usually pre-calculated from other projects such as GloVe or Word2Vec. h 500-long encoder hidden vector. At each point in time, this vector summarizes all the preceding words before it. The final h can be viewed as a "sentence" vector, or a thought vector as Hinton calls it. s 500-long decoder hidden state vector. E 500 neuron recurrent neural network encoder. 500 outputs. Input count is 800-300 from source embedding + 500 from recurrent connections. The encoder feeds directly into the decoder only to initialize it, but not thereafter; hence, that direct connection is shown very faintly. D 2-layer decoder. The recurrent layer has 500 neurons and the fully-connected linear layer has 10k neurons (the size of the target vocabulary).[19] The linear layer alone has 5 million (500 10k) weights - ~10 times more weights than the recurrent layer. score 100-long alignment score w 100-long vector attention weight. These are "soft" weights which changes during the forward pass, in contrast to "hard" neuronal weights that change during the learning phase. A Attention module - this can be a dot product of recurrent states, or the query-key-value fully-connected layers. The output is a 100-long vector w. H 500100. 100 hidden vectors h concatenated into a matrix c 500-long context vector = H * w. c is a linear combination of h vectors weighted by w. Figure 2 shows the internal step-by-step operation of the attention block (A) in Fig 1. Figure 2. The diagram shows the attention forward pass calculating correlations of the word "that" with other words in "See that girl run." Given the right weights from training, the network should be able to identify "girl" as a highly correlated word. Some things to note: This example focuses on the attention of a single word "that". In practice, the attention of each word is calculated in parallel to speed up calculations. Simply changing the lowercase "x" vector to the uppercase "X" matrix will yield the formula for this. Softmax scaling qWkT / 100 prevents a high variance in qWkT that would allow a single word to excessively dominate the softmax resulting in attention to only one word, as a discrete hard max would do. Notation: the commonly written row-wise softmax formula above assumes that vectors are rows, which runs contrary to the standard math notation of column vectors. More correctly, we should take the transpose of the context vector and use the column-wise softmax, resulting in the more correct form (X W v) T [(W k X T) (x _ W q) T] s m {\displaystyle {\begin{aligned}(XW_{v})^{T}*{[(W_{k}X^{T})*{({\underline {x}}W_{q})^{T}}]_{sm}}\end{aligned}}} . This attention scheme has been compared to the Query-Key analogy of relational databases. That comparison suggests an asymmetric role for the Query and Key vectors, where one item of interest (the Query vector "that") is matched against all possible items (the Key vectors of each word in the sentence). However, both Self and Cross Attentions' parallel calculations matches all tokens of the

K matrix with all tokens of the Q matrix; therefore the roles of these vectors are symmetric. Possibly because the simplistic database analogy is flawed, much effort has gone into understanding attention mechanisms further by studying their roles in focused settings, such as in-context learning,[20] masked language tasks,[21] stripped down transformers,[22] bigram statistics,[23] N-gram statistics,[24] pairwise convolutions,[25] and arithmetic factoring.[26] Interpreting attention weights[edit] In translating between languages, alignment is the process of matching words from the source sentence to words of the translated sentence. Networks that perform verbatim translation without regard to word order would show the highest scores along the (dominant) diagonal of the matrix. The off-diagonal dominance shows that the attention mechanism is more nuanced. Consider an example of translating I love you to French. On the first pass through the decoder, 94% of the attention weight is on the first English word I, so the network offers the word je. On the second pass of the decoder, 88% of the attention weight is on the third English word you, so it offers t'. On the last pass, 95% of the attention weight is on the second English word love, so it offers aime. In the I love you example, the second word love is aligned with the third word aime. Stacking soft row vectors together for je, t', and aime yields an alignment matrix: I love you je 0.94 0.02 0.04 t' 0.11 0.01 0.88 aime 0.03 0.95 0.02 Sometimes, alignment can be multiple-to-multiple. For example, the English phrase look it up corresponds to cherchez-le. Thus, "soft" attention weights work better than "hard" attention weights (setting one attention weight to 1, and the others to 0), as we would like the model to make a context vector consisting of a weighted sum of the hidden vectors, rather than "the best one", as there may not be a best hidden vector. Variants[edit] Comparison of the data flow in CNN, RNN, and self-attention Many variants of attention implement soft weights, such as fast weight programmers, or fast weight controllers (1992).[8] A "slow" neural network outputs the "fast" weights of another neural network through outer products. The slow network learns by gradient descent. It was later renamed as "linearized self-attention".[12] Bahdanau-style attention,[13] also referred to as additive attention, Luong-style attention,[27] which is known as multiplicative attention, Early attention mechanisms similar to modern self-attention were proposed using recurrent neural networks. However, the highly parallelizable self-attention was introduced in 2017 and successfully used in the Transformer model, positional attention and factorized positional attention.[28] For convolutional neural networks, attention mechanisms can be distinguished by the dimension on which they operate, namely: spatial attention,[29] channel attention,[30] or combinations.[31][32] These variants recombine the encoder-side inputs to redistribute those effects to each target output. Often, a correlation-style matrix of dot products provides the re-weighting coefficients. In the figures below, W is the matrix of context attention weights, similar to the formula in Overview section above. 1. encoder-decoder dot product 2. encoder-decoder QKV 3. encoder-only dot product 4. encoder-only QKV 5. Pytorch tutorial Both encoder & decoder are needed to calculate attention.[27] Both encoder & decoder are needed to calculate attention.[33] Decoder is not used to calculate attention. With only 1 input into corr, W is an auto-correlation of dot products. wij = xi xj.[34] Decoder is not used to calculate attention.[35] A fully-connected layer is used to calculate attention instead of dot

product correlation.[36] Legend Label Description Variables X, H, S, T Upper case variables represent the entire sentence, and not just the current word. For example, H is a matrix of the encoder hidden state-one word per column. S, T S, decoder hidden state; T, target word embedding. In the Pytorch Tutorial variant training phase, T alternates between 2 sources depending on the level of teacher forcing used. T could be the embedding of the network's output word; i.e. embedding(argmax(FC output)). Alternatively with teacher forcing, T could be the embedding of the known correct word which can occur with a constant forcing probability, say 1/2. X, H H, encoder hidden state; X, input word embeddings. W Attention coefficients Qw, Kw, Vw, FC Weight matrices for query, key, value respectively. FC is a fully-connected weight matrix. , , vector concatenation; , matrix multiplication. corr Column-wise softmax(matrix of all combinations of dot products). The dot products are xi * xj in variant #3, hi* sj in variant 1, and column i (Kw * H) * column j (Qw * S) in variant 2, and column i (Kw * X) * column j (Qw * X) in variant 4. Variant 5 uses a fully-connected layer to determine the coefficients. If the variant is QKV, then the dot products are normalized by the d where d is the height of the QKV matrices. Optimizations[edit] Flash attention[edit] The size of the attention matrix is proportional to the square of the number of input tokens. Therefore, when the input is long, calculating the attention matrix requires a lot of GPU memory. Flash attention is an implementation that reduces the memory needs and increases efficiency without sacrificing accuracy. It achieves this by partitioning the attention computation into smaller blocks that fit into the GPU's faster on-chip memory, reducing the need to store large intermediate matrices and thus lowering memory usage while increasing computational efficiency.[37] FlexAttention[edit] FlexAttention[38] is an attention kernel developed by Meta that allows users to modify attention scores prior to softmax and dynamically chooses the optimal attention algorithm. Self-Attention and Transformers[edit] The major breakthrough came with self-attention, where each element in the input sequence attends to all others, enabling the model to capture global dependencies. This idea was central to the Transformer architecture, which completely replaced recurrence with attention mechanisms. As a result, Transformers became the foundation for models like BERT, GPT, and T5.[17] Applications[edit] Attention is widely used in natural language processing, computer vision, and speech recognition. In NLP, it improves context understanding in tasks like question answering and summarization. In vision, visual attention helps models focus on relevant image regions, enhancing object detection and image captioning. Mathematical representation[edit] Standard Scaled Dot-Product Attention[edit] For matrices: Q R m d k , K R n d k {\displaystyle \mathbf {Q} \in \mathbb {R} ^{m\times d_{k}},\mathbf {K} \in \mathbb {R} ^{n\times d_{k}}} and V R n d v {\displaystyle \mathbf {V} \in \mathbb {R} ^{n\times d_{v}}} , the scaled dot-product, or QKV attention is defined as: Attention (Q , K , V) = softmax (Q K T d k) V R m d v {\displaystyle {\text{Attention}}(\mathbf {Q} ,\mathbf {K} ,\mathbf {V})={\text{softmax}}\left({\frac {\mathbf {Q} \mathbf {K} ^{T}}{\sqrt {d_{k}}}}\right)\mathbf {V} \in \mathbb {R} ^{m\times d_{v}}} where T {\displaystyle {}^{T}} denotes transpose and the softmax function is applied independently to every row of its argument. The matrix Q {\displaystyle \mathbf {Q} } contains m {\displaystyle m} queries, while matrices K , V {\displaystyle \mathbf {K} ,\mathbf {V} } jointly contain

an unordered set of n {\displaystyle n} key-value pairs. Value vectors in matrix V {\displaystyle \mathbf {V} } are weighted using the weights resulting from the softmax operation, so that the rows of the m {\displaystyle m} -by- d v {\displaystyle d_{v}} output matrix are confined to the convex hull of the points in R d v {\displaystyle \mathbb {R} ^{d_{v}}} given by the rows of V {\displaystyle \mathbf {V} } . To understand the permutation invariance and permutation equivariance properties of QKV attention,[39] let A R m m {\displaystyle \mathbf {A} \in \mathbb {R} ^{m\times m}} and B R n n {\displaystyle \mathbf {B} \in \mathbb {R} ^{n\times n}} be permutation matrices; and D R m n {\displaystyle \mathbf {D} \in \mathbb {R} ^{m\times n}} an arbitrary matrix. The softmax function is permutation equivariant in the sense that: softmax (A D B) = A softmax (D) B {\displaystyle {\text{softmax}}(\mathbf {A} \mathbf {D} \mathbf {B})=\mathbf {A} \,{\text{softmax}}(\mathbf {D})\mathbf {B} } By noting that the transpose of a permutation matrix is also its inverse, it follows that: Attention (A Q , B K , B V) = A Attention (Q , K , V) {\displaystyle {\text{Attention}}(\mathbf {A} \mathbf {Q} ,\mathbf {B} \mathbf {K} ,\mathbf {B} \mathbf {V})=\mathbf {A} \,{\text{Attention}}(\mathbf {Q} ,\mathbf {K} ,\mathbf {V})} which shows that QKV attention is equivariant with respect to re-ordering the queries (rows of Q {\displaystyle \mathbf {Q} }); and invariant to re-ordering of the key-value pairs in K , V {\displaystyle \mathbf {K} ,\mathbf {V} } . These properties are inherited when applying linear transforms to the inputs and outputs of QKV attention blocks. For example, a simple self-attention function defined as: X Attention (X T q , X T k , X T v) {\displaystyle \mathbf {X} \mapsto {\text{Attention}}(\mathbf {X} \mathbf {T} _{q},\mathbf {X} \mathbf {T} _{k},\mathbf {X} \mathbf {T} _{v})} is permutation equivariant with respect to re-ordering the rows of the input matrix X {\displaystyle X} in a non-trivial way, because every row of the output is a function of all the rows of the input. Similar properties hold for multi-head attention, which is defined below. Masked Attention[edit] When QKV attention is used as a building block for an autoregressive decoder, and when at training time all input and output matrices have n {\displaystyle n} rows, a masked attention variant is used: Attention (Q , K , V) = softmax (Q K T d k + M) V {\displaystyle {\text{Attention}}(\mathbf {Q} ,\mathbf {K} ,\mathbf {V})={\text{softmax}}\left({\frac {\mathbf {Q} \mathbf {K} ^{T}}{\sqrt {d_{k}}}}+\mathbf {M} \right)\mathbf {V} } where the mask, M R n n {\displaystyle \mathbf {M} \in \mathbb {R} ^{n\times n}} is a strictly upper triangular matrix, with zeros on and below the diagonal and {\displaystyle -\infty } in every element above the diagonal. The softmax output, also in R n n {\displaystyle \mathbb {R} ^{n\times n}} is then lower triangular, with zeros in all elements above the diagonal. The masking ensures that for all 1 i < j n {\displaystyle 1\leq i<j\leq n} , row i {\displaystyle i} of the attention output is independent of row j {\displaystyle j} of any of the three input matrices. The permutation invariance and equivariance properties of standard QKV attention do not hold for the masked variant. Multi-Head Attention[edit] Decoder multiheaded cross-attention Multi-head attention MultiHead (Q , K , V) = Concat (head 1 , . . . , head h) W O {\displaystyle {\text{MultiHead}}(\mathbf {Q} ,\mathbf {K} ,\mathbf {V})={\text{Concat}}({\text{head}}_{1},...,{\text{head}}_{h})\mathbf {W} ^{O}} where each head is computed with QKV attention as: head i = Attention (Q W i Q , K W i K , V W i V

) {\displaystyle {\text{head}}_{i}={\text{Attention}}(\mathbf {Q} \mathbf {W} _{i}^{Q},\mathbf {K} \mathbf {W} _{i}^{K},\mathbf {V} \mathbf {W} _{i}^{V})} and W i Q , W i K , W i V {\displaystyle \mathbf {W} _{i}^{Q},\mathbf {W} _{i}^{K},\mathbf {W} _{i}^{V}} , and W O {\displaystyle \mathbf {W} ^{O}} are parameter matrices. The permutation properties of (standard, unmasked) QKV attention apply here also. For permutation matrices, A , B {\displaystyle \mathbf {A} ,\mathbf {B} } : MultiHead (A Q , B K , B V) = A MultiHead (Q , K , V) {\displaystyle {\text{MultiHead}}(\mathbf {A} \mathbf {Q} ,\mathbf {B} \mathbf {K} ,\mathbf {B} \mathbf {V})=\mathbf {A} \,{\text{MultiHead}}(\mathbf {Q} ,\mathbf {K} ,\mathbf {V})} from which we also see that multi-head self-attention: X MultiHead (X T q , X T k , X T v) {\displaystyle \mathbf {X} \mapsto {\text{MultiHead}}(\mathbf {X} \mathbf {T} _{q},\mathbf {X} \mathbf {T} _{k},\mathbf {X} \mathbf {T} _{v})} is equivariant with respect to re-ordering of the rows of input matrix X {\displaystyle X} . Bahdanau (Additive) Attention[edit] Attention (Q , K , V) = softmax (tanh (W Q Q + W K K) V) {\displaystyle {\text{Attention}}(\mathbf {Q} ,\mathbf {K} ,\mathbf {V})={\text{softmax}}(\tanh(\mathbf {W} _{Q}\mathbf {Q} +\mathbf {W} _{K}\mathbf {K})\mathbf {V})} where W Q {\displaystyle \mathbf {W} _{Q}} and W K {\displaystyle \mathbf {W} _{K}} are learnable weight matrices.[13] Luong Attention (General)[edit] Attention (Q , K , V) = softmax (Q W K T) V {\displaystyle {\text{Attention}}(\mathbf {Q} ,\mathbf {K} ,\mathbf {V})={\text{softmax}}(\mathbf {Q} \mathbf {W} \mathbf {K} ^{T})\mathbf {V} } where W {\displaystyle \mathbf {W} } is a learnable weight matrix.[27] Self Attention[edit] Self-attention is essentially the same as cross-attention, except that query, key, and value vectors all come from the same model. Both encoder and decoder can use self-attention, but with subtle differences. For encoder self-attention, we can start with a simple encoder without self-attention, such as an "embedding layer", which simply converts each input word into a vector by a fixed lookup table. This gives a sequence of hidden vectors h 0 , h 1 , ... {\displaystyle h_{0},h_{1},\dots } . These can then be applied to a dot-product attention mechanism, to obtain h 0 = A t t e n t i o n (h 0 W Q , H W K , H W V) h 1 = A t t e n t i o n (h 1 W Q , H W K , H W V) {\displaystyle {\begin{aligned}h_{0}'&=\mathrm {Attention} (h_{0}W^{Q},HW^{K},HW^{V})\\h_{1}'&=\mathrm {Attention} (h_{1}W^{Q},HW^{K},HW^{V})\\&\cdots \end{aligned}}} or more succinctly, H = A t t e n t i o n (H W Q , H W K , H W V) {\displaystyle H'=\mathrm {Attention} (HW^{Q},HW^{K},HW^{V})} . This can be applied repeatedly, to obtain a multilayered encoder. This is the "encoder self-attention", sometimes called the "all-to-all attention", as the vector at every position can attend to every other. Masking[edit] Decoder self-attention with causal masking, detailed diagramFor decoder self-attention, all-to-all attention is inappropriate, because during the autoregressive decoding process, the decoder cannot attend to future outputs that has yet to be decoded. This can be solved by forcing the attention weights w i j = 0 {\displaystyle w_{ij}=0} for all i < j {\displaystyle i<j} , called "causal masking". This attention mechanism is the "causally masked self-attention". See also[edit] Recurrent neural network seq2seq Transformer (deep learning architecture) Attention Dynamic neural network References[edit] ^ Niu, Zhaoyang; Zhong, Guoqiang; Yu, Hui (2021-09-10). "A review on the attention mechanism of deep learning". Neurocomputing. 452: 48-62.

doi:10.1016/j.neucom.2021.03.091. ISSN 0925-2312. ^ Soydaner, Derya (August 2022). "Attention mechanism in neural networks: where it comes and where it goes". Neural Computing and Applications. 34 (16): 13371-13385. arXiv:2204.13154. doi:10.1007/s00521-022-07366-3. ISSN 0941-0643. ^ Cherry EC (1953). "Some Experiments on the Recognition of Speech, with One and with Two Ears" (PDF). The Journal of the Acoustical Society of America. 25 (5): 975-79. Bibcode:1953ASAJ...25..975C. doi:10.1121/1.1907229. hdl:11858/00-001M-0000-002A-F750-3. ISSN 0001-4966. ^ Broadbent, D (1958). Perception and Communication. London: Pergamon Press. ^ Kowler, Eileen; Anderson, Eric; Dosher, Barbara; Blaser, Erik (1995-07-01). "The role of attention in the programming of saccades". Vision Research. 35 (13): 1897-1916. doi:10.1016/0042-6989(94)00279-U. ISSN 0042-6989. PMID 7660596. ^ Rumelhart, David E.; Hinton, G. E.; Mcclelland, James L. (1987-07-29). "A General Framework for Parallel Distributed Processing" (PDF). In Rumelhart, David E.; Hinton, G. E.; PDP Research Group (eds.). Parallel Distributed Processing, Volume 1: Explorations in the Microstructure of Cognition: Foundations. Cambridge, Massachusetts: MIT Press. ISBN 978-0-262-68053-0. ^ Giles, C. Lee; Maxwell, Tom (1987-12-01). "Learning, invariance, and generalization in high-order neural networks". Applied Optics. 26 (23): 4972-4978. doi:10.1364/AO.26.004972. ISSN 0003-6935. PMID 20523475. ^ a b Schmidhuber, Jurgen (1992). "Learning to control fast-weight memories: an alternative to recurrent nets". Neural Computation. 4 (1): 131-139. doi:10.1162/neco.1992.4.1.131. S2CID 16683347. ^ Christoph von der Malsburg: The correlation theory of brain function. Internal Report 81-2, MPI Biophysical Chemistry, 1981. http://cogprints.org/1380/1/vdM_correlation.pdf See Reprint in Models of Neural Networks II, chapter 2, pages 95-119. Springer, Berlin, 1994. ^ Jerome A. Feldman, "Dynamic connections in neural networks," Biological Cybernetics, vol. 46, no. 1, pp. 27-39, Dec. 1982. ^ Hinton, Geoffrey E.; Plaut, David C. (1987). "Using Fast Weights to Deblur Old Memories". Proceedings of the Annual Meeting of the Cognitive Science Society. 9. ^ a b Schlag, Imanol; Irie, Kazuki; Schmidhuber, Jurgen (2021). "Linear Transformers Are Secretly Fast Weight Programmers". ICML 2021. Springer. pp. 9355-9366. ^ a b c Bahdanau, Dzmitry; Cho, Kyunghyun; Bengio, Yoshua (2014). "Neural Machine Translation by Jointly Learning to Align and Translate". arXiv:1409.0473 [cs.CL]. ^ Vinyals, Oriol; Toshev, Alexander; Bengio, Samy; Erhan, Dumitru (2015). "Show and Tell: A Neural Image Caption Generator". pp. 3156-3164. ^ Xu, Kelvin; Ba, Jimmy; Kiros, Ryan; Cho, Kyunghyun; Courville, Aaron; Salakhudinov, Ruslan; Zemel, Rich; Bengio, Yoshua (2015-06-01). "Show, Attend and Tell: Neural Image Caption Generation with Visual Attention". Proceedings of the 32nd International Conference on Machine Learning. PMLR: 2048-2057. ^ Bahdanau, Dzmitry; Cho, Kyunghyun; Bengio, Yoshua (19 May 2016). "Neural Machine Translation by Jointly Learning to Align and Translate". arXiv:1409.0473 [cs.CL]. (orig-date 1 Sep 2014) ^ a b Vaswani, Ashish; Shazeer, Noam; Parmar, Niki; Uszkoreit, Jakob; Jones, Llion; Gomez, Aidan N; Kaiser, ukasz; Polosukhin, Illia (2017). "Attention is All you Need" (PDF). Advances in Neural Information Processing Systems. 30. Curran Associates, Inc. ^ Britz, Denny; Goldie, Anna; Luong, Minh-Thanh; Le, Quoc (2017-03-21). "Massive Exploration of Neural Machine Translation Architectures". arXiv:1703.03906 [cs.CV]. ^ "Pytorch.org seq2seq tutorial". Retrieved December 2, 2021. ^ Zhang, Ruiqi (2024). "Trained Transformers Learn Linear Models In-Context" (PDF). Journal of Machine

Learning Research 1-55. 25. arXiv:2306.09927. ^ Rende, Riccardo (2024). "Mapping of attention mechanisms to a generalized Potts model". Physical Review Research. 6 (2) 023057. arXiv:2304.07235. Bibcode:2024PhRvR...6b3057R. doi:10.1103/PhysRevResearch.6.023057. ^ He, Bobby (2023). "Simplifying Transformers Blocks". arXiv:2311.01906 [cs.LG]. ^ Nguyen, Timothy (2024). "Understanding Transformers via N-gram Statistics". arXiv:2407.12034 [cs.CL]. ^ "Transformer Circuits". transformer-circuits.pub. ^ Transformer Neural Network Derived From Scratch. 2023. Event occurs at 05:30. Retrieved 2024-04-07. ^ Charton, Francois (2023). "Learning the Greatest Common Divisor: Explaining Transformer Predictions". arXiv:2308.15594 [cs.LG]. ^ a b c Luong, Minh-Thang (2015-09-20). "Effective Approaches to Attention-Based Neural Machine Translation". arXiv:1508.04025v5 [cs.CL]. ^ "Learning Positional Attention for Sequential Recommendation". catalyzex.com. ^ Zhu, Xizhou; Cheng, Dazhi; Zhang, Zheng; Lin, Stephen; Dai, Jifeng (2019). "An Empirical Study of Spatial Attention Mechanisms in Deep Networks". 2019 IEEE/CVF International Conference on Computer Vision (ICCV). pp. 6687-6696. arXiv:1904.05873. doi:10.1109/ICCV.2019.00679. ISBN 978-1-7281-4803-8. S2CID 118673006. ^ Hu, Jie; Shen, Li; Sun, Gang (2018). "Squeeze-and-Excitation Networks". 2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition. pp. 7132-7141. arXiv:1709.01507. doi:10.1109/CVPR.2018.00745. ISBN 978-1-5386-6420-9. S2CID 206597034. ^ Woo, Sanghyun; Park, Jongchan; Lee, Joon-Young; Kweon, In So (2018-07-18). "CBAM: Convolutional Block Attention Module". arXiv:1807.06521 [cs.CV]. ^ Georgescu, Mariana-Iuliana; Ionescu, Radu Tudor; Miron, Andreea-Iuliana; Savencu, Olivian; Ristea, Nicolae-Catalin; Verga, Nicolae; Khan, Fahad Shahbaz (2022-10-12). "Multimodal Multi-Head Convolutional Attention with Various Kernel Sizes for Medical Image Super-Resolution". arXiv:2204.04218 [eess.IV]. ^ Neil Rhodes (2021). CS 152 NN-27: Attention: Keys, Queries, & Values. Event occurs at 06:30. Retrieved 2021-12-22. ^ Alfredo Canziani & Yann Lecun (2021). NYU Deep Learning course, Spring 2020. Event occurs at 05:30. Retrieved 2021-12-22. ^ Alfredo Canziani & Yann Lecun (2021). NYU Deep Learning course, Spring 2020. Event occurs at 20:15. Retrieved 2021-12-22. ^ Robertson, Sean. "NLP From Scratch: Translation With a Sequence To Sequence Network and Attention". pytorch.org. Retrieved 2021-12-22. ^ Mittal, Aayush (2024-07-17). "Flash Attention: Revolutionizing Transformer Efficiency". Unite.AI. Retrieved 2024-11-16. ^ "FlexAttention: The Flexibility of PyTorch with the Performance of FlashAttention - PyTorch". ^ Lee, Juho; Lee, Yoonho; Kim, Jungtaek; Kosiorek, Adam R; Choi, Seungjin; Teh, Yee Whye (2018). "Set Transformer: A Framework for Attention-based Permutation-Invariant Neural Networks". arXiv:1810.00825 [cs.LG]. External links[edit] Olah, Chris; Carter, Shan (September 8, 2016). "Attention and Augmented Recurrent Neural Networks". Distill. 1 (9). Distill Working Group. doi:10.23915/distill.00001. Dan Jurafsky and James H. Martin (2022) Speech and Language Processing (3rd ed. draft, January 2022), ch. 10.4 Attention and ch. 9.7 Self-Attention Networks: Transformers Alex Graves (4 May 2020), Attention and Memory in Deep Learning (video lecture), DeepMind / UCL, via YouTube vteArtificial intelligence (AI)History (timeline)Concepts Parameter Hyperparameter Loss functions Regression Bias-variance tradeoff Double descent Overfitting Clustering Gradient descent SGD Quasi-Newton method Conjugate gradient method Backpropagation Attention Convolution Normalization Batchnorm Activation Softmax Sigmoid Rectifier Gating Weight initialization Regularization Datasets

Augmentation Prompt engineering Reinforcement learning Q-learning SARSA Imitation Policy gradient Diffusion Latent diffusion model Autoregression Adversary RAG Uncanny valley RLHF Self-supervised learning Reflection Recursive self-improvement Hallucination Word embedding Vibe coding Applications Machine learning In-context learning Artificial neural network Deep learning Language model Large language model NMT Reasoning language model Model Context Protocol Intelligent agent Artificial human companion Humanity's Last Exam Artificial general intelligence (AGI) ImplementationsAudio-visual AlexNet WaveNet Human image synthesis HWR OCR Computer vision Speech synthesis 15.ai ElevenLabs Speech recognition Whisper Facial recognition AlphaFold Text-to-image models Aurora DALL-E Firefly Flux Ideogram Imagen Midjourney Recraft Stable Diffusion Text-to-video models Dream Machine Runway Gen Hailuo AI Kling Sora Veo Music generation Suno AI Udio Text Word2vec Seq2seq GloVe BERT T5 Llama Chinchilla AI PaLM GPT 1 2 3 J ChatGPT 4 4o o1 o3 4.5 4.1 o4-mini Claude Gemini chatbot Grok LaMDA BLOOM DBRX Project Debater IBM Watson IBM Watsonx Granite PanGu- DeepSeek Qwen Decisional AlphaGo AlphaZero OpenAI Five Self-driving car MuZero Action selection AutoGPT Robot control People Alan Turing Warren Sturgis McCulloch Walter Pitts John von Neumann Claude Shannon Shun'ichi Amari Kunihiko Fukushima Takeo Kanade Marvin Minsky John McCarthy Nathaniel Rochester Allen Newell Cliff Shaw Herbert A. Simon Oliver Selfridge Frank Rosenblatt Bernard Widrow Joseph Weizenbaum Seymour Papert Seppo Linnainmaa Paul Werbos Geoffrey Hinton John Hopfield Jurgen Schmidhuber Yann LeCun Yoshua Bengio Lotfi A. Zadeh Stephen Grossberg Alex Graves James Goodnight Andrew Ng Fei-Fei Li Ilya Sutskever Alex Krizhevsky Ian Goodfellow Demis Hassabis David Silver Andrej Karpathy Ashish Vaswani Noam Shazeer Aidan Gomez Francois Chollet Architectures Neural Turing machine Differentiable neural computer Transformer Vision transformer (ViT) Recurrent neural network (RNN) Long short-term memory (LSTM) Gated recurrent unit (GRU) Echo state network Multilayer perceptron (MLP) Convolutional neural network (CNN) Residual neural network (RNN) Highway network Mamba Autoencoder Variational autoencoder (VAE) Generative adversarial network (GAN) Graph neural network (GNN) Portals Technology Category Artificial neural networks Machine learning List Companies Projects Retrieved from "https://en.wikipedia.org/w/index.php?title=Attention_(machine_learning)&oldid=1299557670" Category: Machine learningHidden categories: Articles with short descriptionShort description matches WikidataArticles that may contain original research from June 2025All articles that may contain original researchAll articles with unsourced statementsArticles with unsourced statements from June 2025 Search Search Attention (machine learning) 14 languages Add topic

Fine-tuning Strategies

Finetuning LLMs - truefoundry Docstruefoundry Docs home pageSearch...KAsk AISearch...NavigationLLM FinetuningFinetuning LLMsHomeML EngineeringAI GatewayLLM EngineeringAPI ReferenceChangelogLLM DeploymentDeploying LLMsDeploying NVIDIA NIM ModelsBenchmarking LLMsLLM FinetuningFinetuning LLMsPrompt ManagementPrompt ManagementLLM TracingOverviewGetting StartedConfigurationsFrameworksDistributed TracingOn this pageQLoRAPre-requisitesSetting up the Training DataChatCompletionUpload to a TrueFoundry

ArtifactUpload to a cloud storageFine-Tuning a LLMHyperparametersFine-Tuning using a NotebookFine-Tuning using a JobDeploying the Fine-Tuned ModelAdvanced: Merging LoRa Adapters and uploading merged modelLLMs are pre-trained on massive datasets of text and code. This makes them versatile for various tasks, but they may not perform optimally on your specific domain or data. Finetuning allows you to train these models on your data, enhancing their performance and tailoring them to your unique requirements. Fine-tuning with TrueFoundry allows you to bring your data, and fine-tune popular Open Source LLM's such as Llama 2, Mistral, Zephyr, Mixtral, and more. This is made easy, as we provide pre-configured options for resources and use the optimal training techniques available. You can choose to perform fine-tuning either using Jobs or Notebooks. You can further, easily track the progress of finetuning through ML-Repositories. Supported Architectures and SizesWe support model sizes of up to 70B for the following model architectures llama mistral qwen qwen2 gemma gemma2 phi phi3 phi4 Following architectures are supported on best effort basis mixtral gemma3 falcon deci mpt gpt_bigcode gpt_neox QLoRA For fine-tuning, TrueFoundry embraces the QLoRA technique, a cutting-edge technique that revolutionizes fine-tuning by balancing power and efficiency. This technique uses clever tricks to stay compact, so you can fine-tune on smaller hardware (even just one GPU), saving time, money, and resources, all while maintaining top performance. Pre-requisites Before you begin, ensure you have the following: Workspace: To deploy your LLM, you'll need a workspace. If you don't have one, you can create it using this guide: Create a Workspace or seek assistance from your cluster administrator. Setting up the Training Data We support two different data formats: Chat Data needs to be in jsonl format with each line containing a whole conversation in OpenAI Chat format Each line contains a key called messages. Each messages key contains a list of messages, where each message is a dictionary with role and content keys. The role key can be either user, assistant or system and the content key contains the message content. Example: Chat Type DatasetCopyAsk AI{"messages": [{"role": "system", "content": "Marv is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "What's the capital of France?"}, {"role": "assistant", "content": "Paris"}, {"role": "user", "content": "Can you be more sarcastic?"}, {"role": "assistant", "content": "Paris, as if everyone doesn't know that already."}]} {"messages": [{"role": "system", "content": "Marv is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "Who wrote 'Romeo and Juliet'?"}, {"role": "assistant", "content": "William Shakespeare"}, {"role": "user", "content": "Can you be more sarcastic?"}, {"role": "assistant", "content": "Oh, just some guy named William Shakespeare. Ever heard of him?"}]} {"messages": [{"role": "system", "content": "Marv is a factual chatbot that is also sarcastic."}, {"role": "user", "content": "How far is the Moon from Earth?"}, {"role": "assistant", "content": "384,400 kilometers"}, {"role": "user", "content": "Can you be more sarcastic?"}, {"role": "assistant", "content": "Around 384,400 kilometers. Give or take a few, like that really matters."}]} ... Completion Data needs to be in jsonl format with each line containing a json encoded string containing two keys prompt and completion. Example: Completion Type DatasetCopyAsk AI{"prompt": "What is 2 + 2?", "completion": "The answer to 2 + 2 is 4"} {"prompt": "Flip a coin", "completion": "I flipped a coin and the result is heads!"} {"prompt": "<prompt text>", "completion": "<ideal generated text>"} ... You can further split your data into

training data and evaluation data. Once your data is prepared, you need to store the data somewhere. You can choose where to store your data: TrueFoundry Artifact: Upload it as a TrueFoundry artifact for easy access. Cloud Storage: Upload it to a cloud storage service. Local Machine: Save it directly on your computer. Upload to a TrueFoundry Artifact If you prefer to upload your training data directly to TrueFoundry as an artifact, follow the Add Artifacts via UI, and Upload your .jsonl training data file. Upload to a cloud storage You can upload your data to a S3 Bucket using the following command: ShellCopyAsk AIaws s3 cp "path-to-training-data-file-locally" s3://bucket-name/dir-to-store-file Once done you can generate a pre-signed URL of the S3 Object using the following command: ShellCopyAsk AIaws s3 presign s3://bucket-name/path-to-training-data-file-in-s3 Output of uploading file to AWS S3 and getting the pre-signed URL Now you can use this pre-signed URL in the fine-tuning job / notebook. Similarly, you can also upload to AZURE BLOB and GCP GCS. Fine-Tuning a LLM Now that your data is prepared, you can start the fine-tuning. Once your data is ready, you can now start fine-tuning your LLM. Here you have two options, deploying a fine-tuning notebook for experimentation or launching a dedicated fine-tuning job. Notebooks: Experimentation Playground Notebooks offer an ideal setup for explorative and iterative fine-tuning. You can experiment on a small subset of data, trying different hyperparameters to figure out the ideal configuration for the best performance. Thanks to the interactive setup, you can analyze the intermediate results to gain deeper insights into the LLM's behavior and response to different training parameters. Therefore, notebooks are strongly recommended for early-stage exploration and hyperparameter tuning. Jobs: Reliable and Scalable Once you've identified the optimal hyperparameters and configuration through experimentation, transitioning to a deployment job helps you fine-tune on whole dataset and facilitates rapid and reliable training. It ensures consistent and reproducible training runs, as well as built-in retry mechanisms automatically handle any hiccups, ensuring seamless training without manual intervention Consequently, deployment jobs are the preferred choice for large-scale LLM finetuning, particularly when the optimal configuration has been established through prior experimentation. Hyperparameters Fine-tuning an LLM requires adjusting key parameters to optimize its performance on your specific task. Here are some crucial hyperparameters to consider: Epochs: This determines the number of times the model iterates through the entire training dataset. Too many epochs can lead to overfitting, and too few might leave the model undertrained. You should start with a moderate number and increase until the validation performance starts dropping. Learning Rate: This defines how quickly the model updates its weights based on errors. Too high can cause instability and poor performance, and too low can lead to slow learning. Start small and gradually increase if the finetuning is slow. Batch Size: This controls how many data points the model processes before adjusting its internal parameters. Choose a size based on memory constraints and desired training speed. Too high can strain resources, and too low might lead to unstable updates. Lora Alpha and R: These control the adaptive scaling of weights in the Lora architecture, improving efficiency for large models. These are useful parameters for generalization. High values might lead to instability, low values might limit potential performance. Max Length : This defines the maximum sequence length the model can process

at once. Choose based on your task's typical input and output lengths. Too short can truncate context, and too long can strain resources and memory. The optimal values for these hyperparameters depend on your specific LLM, task, and dataset. Be prepared to experiment and iteratively refine your settings for optimal performance. Fine-Tuning using a Notebook Fine-Tuning using a Job Before you start, you will first need to create an ML Repo (this will be used to store your training metrics and artifacts, such as your checkpoints and models) and give your workspace access to the ML Repo. You can read more about ML Repo's here Now that your ML Repo is set up, you can create the fine-tuning job. Deploying the Fine-Tuned Model Once your Fine-tuning is complete, the next step is to deploy the fine-tuned LLM. You can learn more about how to send requests to your Deploy LLM using the following guide Advanced: Merging LoRa Adapters and uploading merged model Currently the finetuning Job/Notebook only converts the best checkpoint to a merged model to save GPU compute time. But sometimes we might want to pick an intermediate checkpoint, merge it and re-upload it as a model. Upcoming ImprovmentsWe are working on building this feature as part of the platform. Till then the code in this section can be run locally or in a Notebook on the platform First, let's make sure we have following requirements installed CopyAsk AI--extra-index-url https://download.pytorch.org/whl/cu121 torch==2.3.0+cu121 tokenizers==0.19.1 transformers==4.42.3 accelerate==0.31.0 peft==0.11.1 truefoundry[ml]>=0.5.5,<1.0.0 Make sure you are logged in to TrueFoundry CopyAsk AItfy login --host <Your TrueFoundry Platform URL> Next, save this script merge_and_upload.pyCopyAsk AIimport os import math import os import re import shutil from typing import Any, Dict, Optional import numpy as np from huggingface_hub import scan_cache_dir import argparse import json from truefoundry.ml import get_client, TransformersFramework from transformers import AutoModelForCausalLM, AutoTokenizer from peft import PeftModel def get_or_create_run(ml_repo: str, run_name: str, auto_end: bool = False, create_ml_repo: bool = False): client = get_client() if create_ml_repo: client.create_ml_repo(ml_repo=ml_repo) try: run = client.get_run_by_name(ml_repo=ml_repo, run_name=run_name) except Exception as e: if "RESOURCE_DOES_NOT_EXIST" not in str(e): raise run = client.create_run(ml_repo=ml_repo, run_name=run_name, auto_end=auto_end) return run def log_model_to_truefoundry(run, model_name: str, model_dir: str, hf_hub_model_id: str, metadata: Optional[Dict[str, Any]] = None, step: int = 0,): metadata = metadata or {} print("Uploading Model...") hf_cache_info = scan_cache_dir() files_to_save = [] for repo in hf_cache_info.repos: if repo.repo_id == hf_hub_model_id: for revision in repo.revisions: for file in revision.files: if file.file_path.name.endswith(".py"): files_to_save.append(file.file_path) break # copy the files to output_dir of pipeline for file_path in files_to_save: match = re.match(r".*snapshots\/[^\/]+\/(.*)", str(file_path)) if match: relative_path = match.group(1) destination_path = os.path.join(model_dir, relative_path) os.makedirs(os.path.dirname(destination_path), exist_ok=True) shutil.copy(str(file_path), destination_path) else: print("Python file in hf model cache in unknown path:", file_path) metadata.update({ "pipeline_tag": "text-generation", "library_name": "transformers", "base_model": hf_hub_model_id, "huggingface_model_url": f"https://huggingface.co/{hf_hub_model_id}" }) metadata = { k: v for k, v in metadata.items() if isinstance(v, (int, float, np.integer, np.floating)) and

math.isfinite(v) } run.log_model(name=model_name, model_file_or_folder=model_dir, framework=TransformersFramework(pipeline_tag="text-generation", library_name="transformers", base_model=hf_hub_model_id), metadata=metadata, step=step,) print(f"You can view the model at {run.dashboard_link}?tab=models") def merge_and_upload(hf_hub_model_id: str, ml_repo: str, run_name: str, artifact_version_fqn: str, saved_model_name: str, dtype: str = "bfloat16", device_map: str = "auto",): import torch client = get_client() if device_map.startswith("{"): device_map = json.loads(device_map) artifact_version = client.get_artifact_version_by_fqn(artifact_version_fqn) lora_model_path = artifact_version.download() tokenizer = AutoTokenizer.from_pretrained(hf_hub_model_id) model = AutoModelForCausalLM.from_pretrained(hf_hub_model_id, device_map=device_map, torch_dtype=getattr(torch, dtype)) model = PeftModel.from_pretrained(model, lora_model_path) model = model.merge_and_unload(progressbar=True) merged_model_dir = os.path.abspath("./merged") os.makedirs(merged_model_dir, exist_ok=True) tokenizer.save_pretrained(merged_model_dir) model.save_pretrained(merged_model_dir) run = get_or_create_run(ml_repo=ml_repo, run_name=run_name, auto_end=False, create_ml_repo=False,) log_model_to_truefoundry(run=run, model_name=saved_model_name, model_dir=merged_model_dir, hf_hub_model_id=hf_hub_model_id, metadata={ "checkpoint": artifact_version_fqn, }, step=artifact_version.step,) def main(): parser = argparse.ArgumentParser() parser.add_argument("--hf_hub_model_id", type=str, required=True) parser.add_argument("--ml_repo", type=str, required=True) parser.add_argument("--run_name", type=str, required=True) parser.add_argument("--artifact_version_fqn", type=str, required=True) parser.add_argument("--saved_model_name", type=str, required=True) parser.add_argument("--dtype", type=str, default="bfloat16", choices=["bfloat16", "float16", "float32"]) parser.add_argument("--device_map", type=str, default="auto") args = parser.parse_args() merge_and_upload(hf_hub_model_id=args.hf_hub_model_id, ml_repo=args.ml_repo, run_name=args.run_name, artifact_version_fqn=args.artifact_version_fqn, saved_model_name=args.saved_model_name, dtype=args.dtype, device_map=args.device_map,) if __name__ == "__main__": main()
Finally, run this script. Run --help to see help python merge_and_upload.py --helpCopyAsk AIusage: merge_and_upload.py [-h] --hf_hub_model_id HF_HUB_MODEL_ID --ml_repo ML_REPO --run_name RUN_NAME --artifact_version_fqn ARTIFACT_VERSION_FQN --saved_model_name SAVED_MODEL_NAME [--dtype {bfloat16,float16,float32}] [--device_map DEVICE_MAP] options: -h, --help show this help message and exit --hf_hub_model_id HF_HUB_MODEL_ID HuggingFace Hub model id to merge the LoRa adapter with. E.g. `stas/tiny-random-llama-2` --ml_repo ML_REPO ML repo to log the merged model to --run_name RUN_NAME Name of the run to log the merged model to. If the run does not exist, it will be created. --artifact_version_fqn ARTIFACT_VERSION_FQN Artifact version FQN of the LoRa adapter to merge with the HF model --saved_model_name SAVED_MODEL_NAME Name of the model to log to MLFoundry --dtype {bfloat16,float16,float32} Data type to load the base model --device_map DEVICE_MAP device_map to use when loading the model. auto, cpu, or a dictionary of device_map E.g.
usage which merges checkpoint artifact:truefoundry/llm-experiments/ckpt-finetune-2024-03-14T05-00-55:7 with its base

model stas/tiny-random-llama-2 and uploads it as finetuned-tiny-random-llama-checkpoint-7 to run finetune-2024-03-14T05-00-55 in ML Repo llm-experiments Grab the checkpoint's artifact version FQN from the Artifacts Tab of your Job Run Output CopyAsk AIpython merge_and_upload.py \ --hf_hub_model_id stas/tiny-random-llama-2 \ --ml_repo llm-experiments \ --run_name finetune-2024-03-14T05-00-55 \ --artifact_version_fqn artifact:truefoundry/llm-experiments/ckpt-finetune-2024-03-14T05-00-55:7 \ --saved_model_name finetuned-tiny-random-llama-checkpoint-7 \ --dtype bfloat16 \ --device_map auto Was this page helpful?YesNoBenchmarking LLMsPrompt ManagementAssistantResponses are generated using AI and may contain mistakes.