How to compute LLM embeddings 3X faster with model quantization

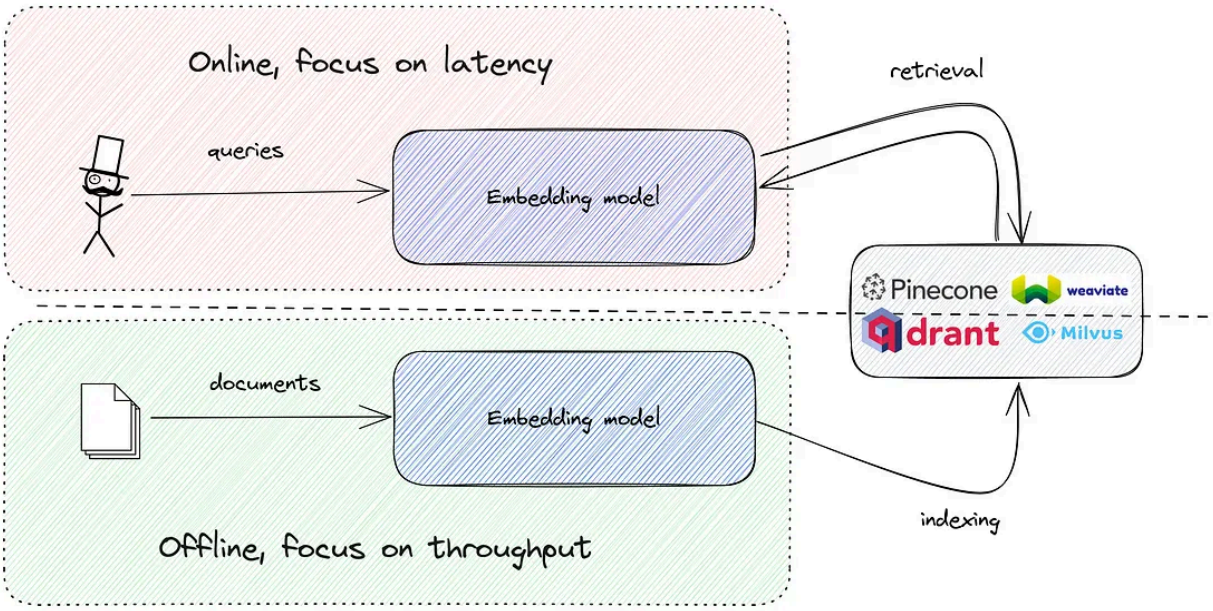
Roman Grebennikov · Follow

Published in Nixiesearch · 10 min read · Nov 13, 2023

Running LLM embedding models is slow on CPU and expensive on GPU. We will make it up to 3X faster with **ONNX model quantization**, see how different **int8 formats** affect performance on new and old hardware, and go even further with doing **ONNX transformer optimization** on top of the quantized model.

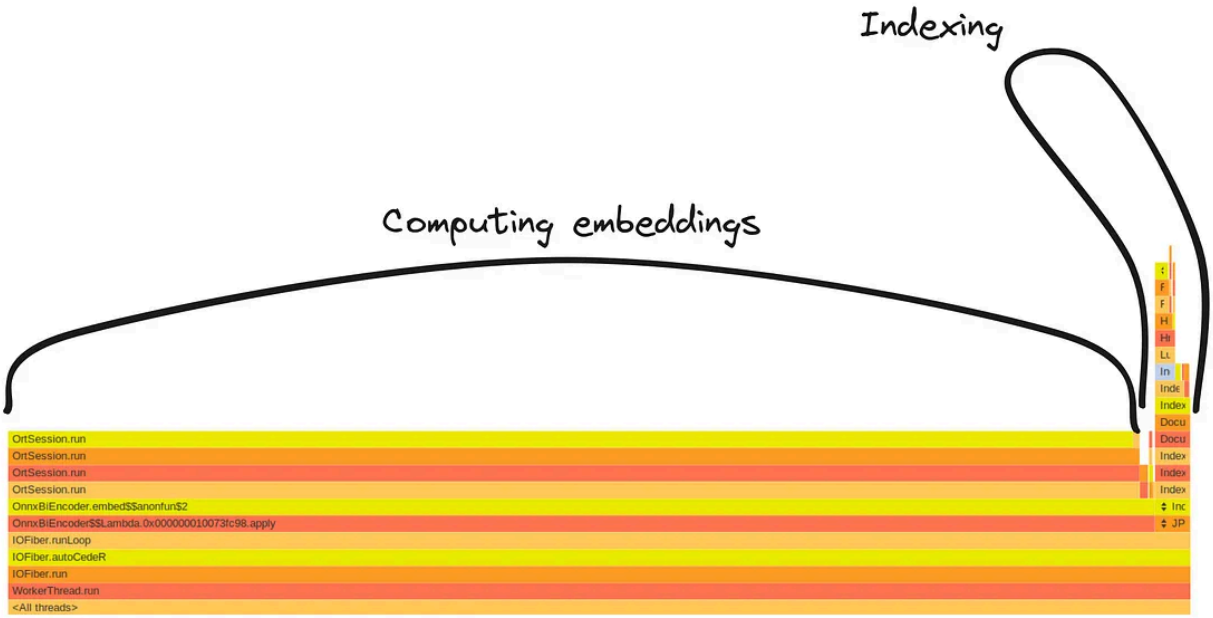
The vector search problem nobody talks about

To perform a semantic search with LLM embeddings, you must first compute these embeddings. With many vector search databases in the market, computing embeddings is unfortunately considered an afterthought and out-of-scope problem.



Embedding inference is a primary step of any semantic search system. Image by author.

We are working on a new open-source search engine, Nixiesearch, which *can fine-tune embeddings to your data*. As we handle embeddings on the server side, we were not surprised to see the tremendous performance impact of running embeddings on a CPU.



A flame graph of indexing process in Nixiesearch with the e5-small-v2 embedding model. Image by author.

The flame graph above shows that **95% of CPU** time is spent on computing embeddings. For sure, you can make it much faster by switching the indexing to GPU, but it's still a major bummer for people wanting to try Nixiesearch. Can we make it faster while still staying on CPU?

Model quantization

All the present deep neural networks are just glorified combinations of matrix operations.

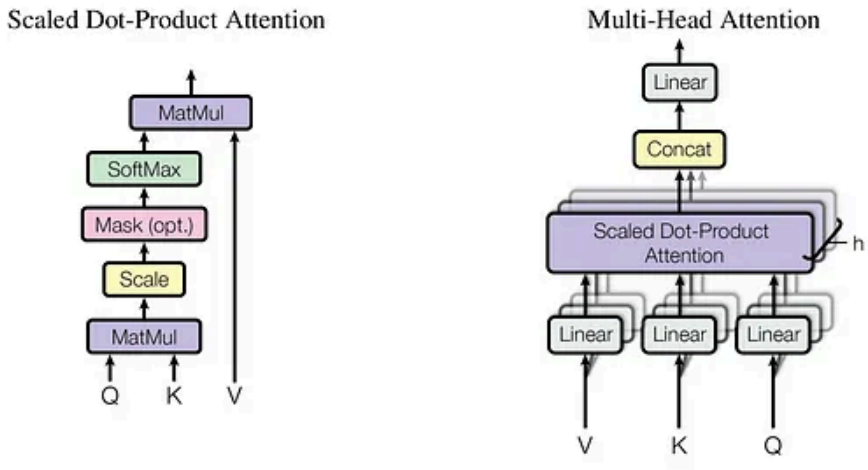
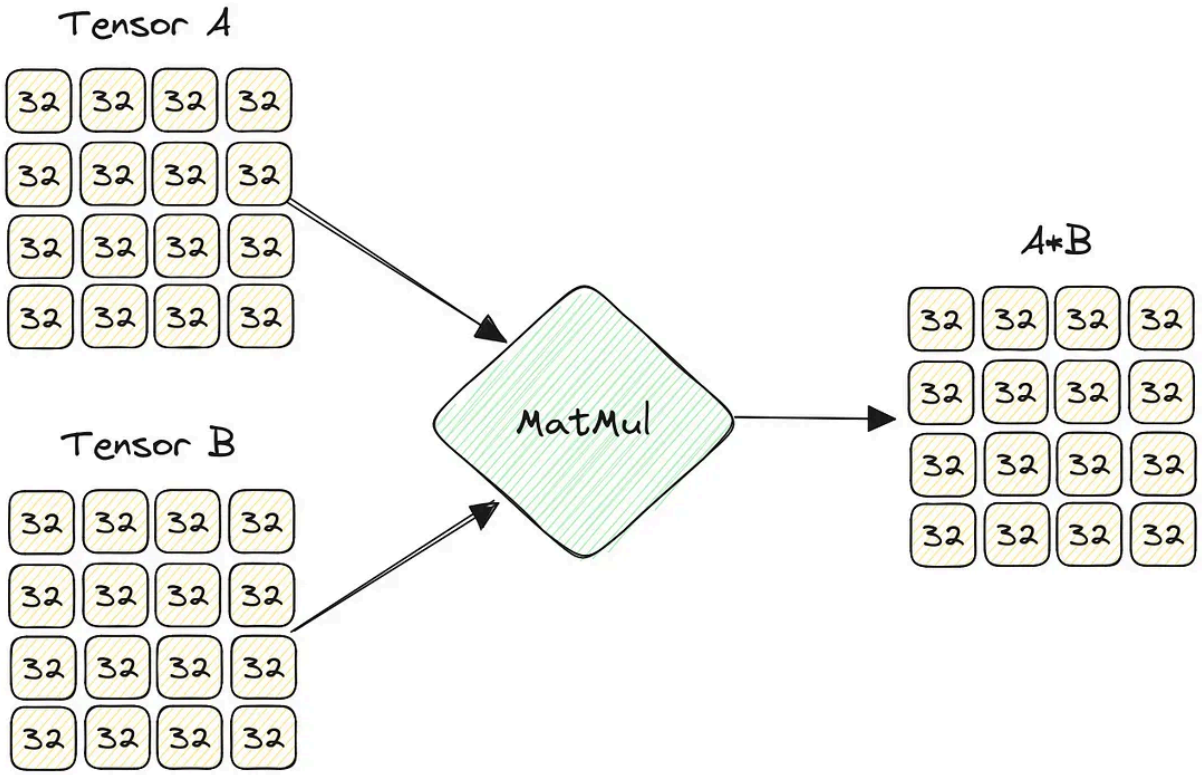


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

Attention layer matrix operations. Image from 'Attention Is All You Need' by Vaswani et al.

As shown in the diagram above, the attention layer from a transformer network is just a combination of trivial algebraic transformations on top of matrices. In a classical implementation, these matrices contain 32-bit float values. What if we are ready to trade a bit of precision for better performance, reducing the float size from 32 bytes down to 8?

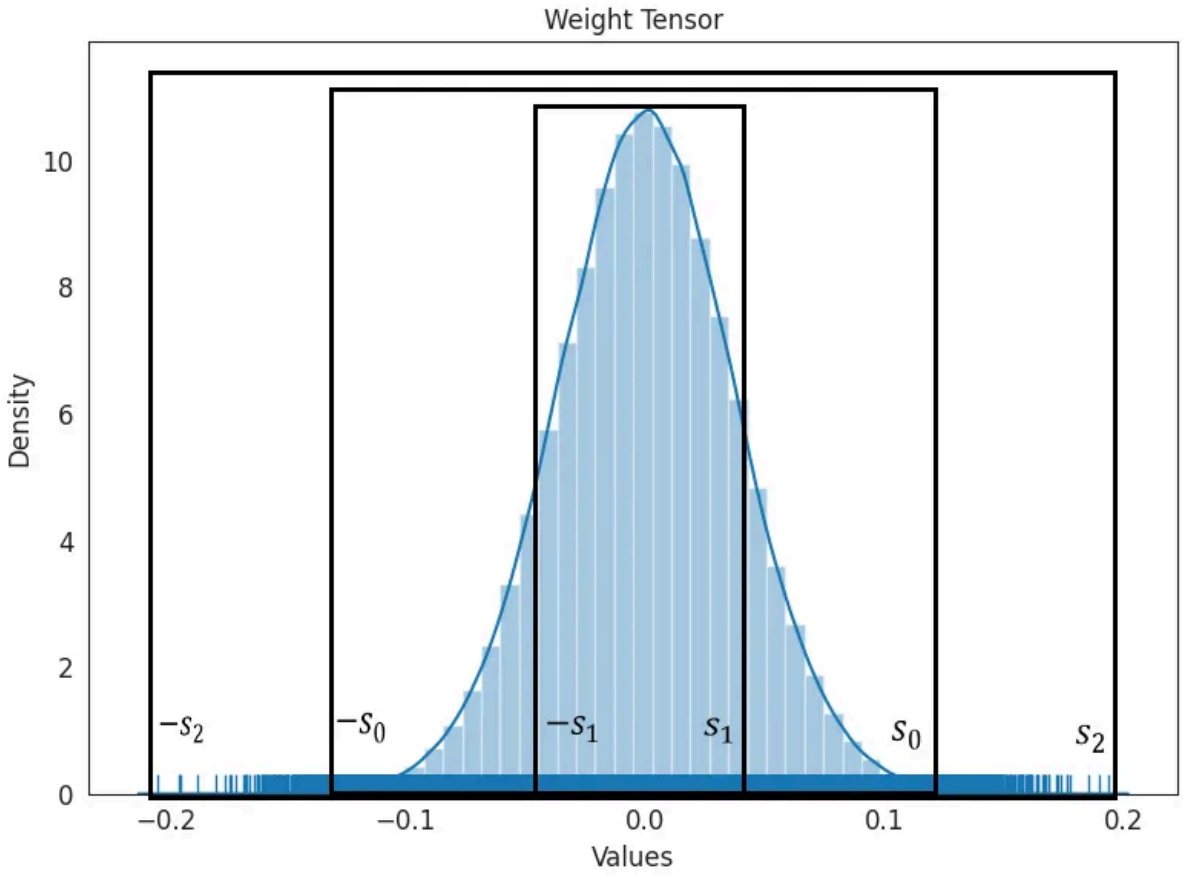


32-bit float matrix multiplication. Image by author.

This approach is called quantization: reducing the numerical precision of matrices used to store model weights and neuron activation values.

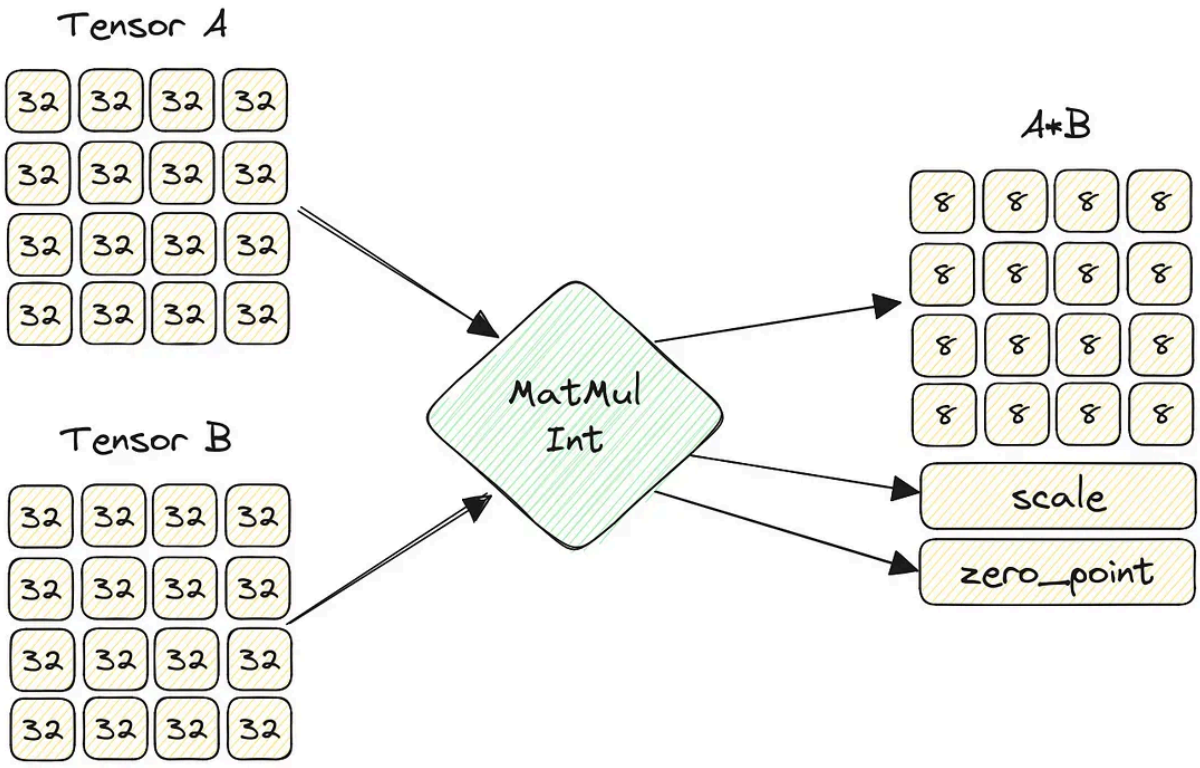
Going from 32 to 8 bits of precision will reduce RAM size needed to store model weights 4x and *hopefully* make it faster with modern CPUs.

But you cannot just fit a 32-bit float to a 8-bit integer without loss: with only 8 bits of storage per number, you can encode only 256 distinct values!



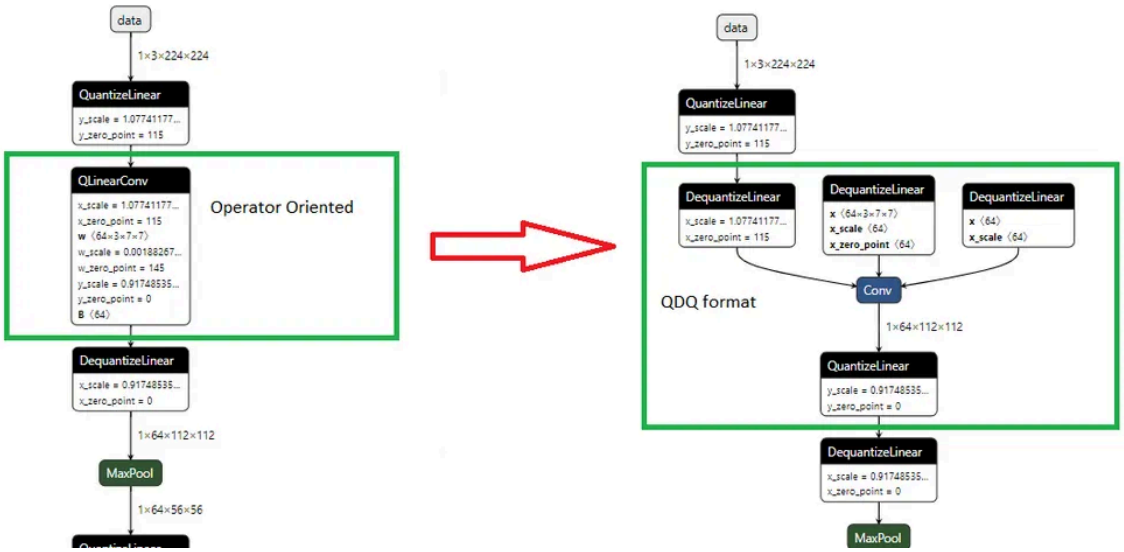
A weight distribution in trained TinyBERT. Image from KDLSQ-BERT: A Quantized Bert Combining Knowledge Distillation with Learned Step Size Quantization by J. Jin et al.

But weights and activation values inside the neural network are not random numbers! As you can see from the histogram above, they are usually close to zero and normally distributed. With this observation we can replace all the network operations with quantization-aware ones — tracking also **zero_point** and **scale** of the underlying numerical distribution.



Operator-based quantization in ONNX. Image by author.

This approach is called operator quantization and is considered the most straightforward approach to the problem. Another option would be a QDQ quantization where you still use the same 32-bit operators but inject a quantize-dequantize pair of operators before a regular one — which is usually much slower.



Operator vs QDQ quantization. An image from ONNX documentation — Quantize ONNX Models.

In the image above, you can see extra nodes injected into the graph in the QDQ mode, which usually results in worse performance:

- **Additional Quantize-Dequantize** operators are not free.
- The primary **operator runs on Float32** data as before, so you only save on RAM/VRAM usage, not performance.
- QDQ is only supported for **static quantization in ONNX runtime** — see the next chapter for details.

For the sake of simplicity we will target only operator quantization in this article.

> *Note that model quantization differs from embedding quantization supported in all major vector search engines like Elasticsearch, Qdrant, Vespa, and Weaviate.* ***The quantized model still emits Float32 embeddings as before*** *— it just uses a more compact layout for weights and activations.*

Dynamic vs static quantization

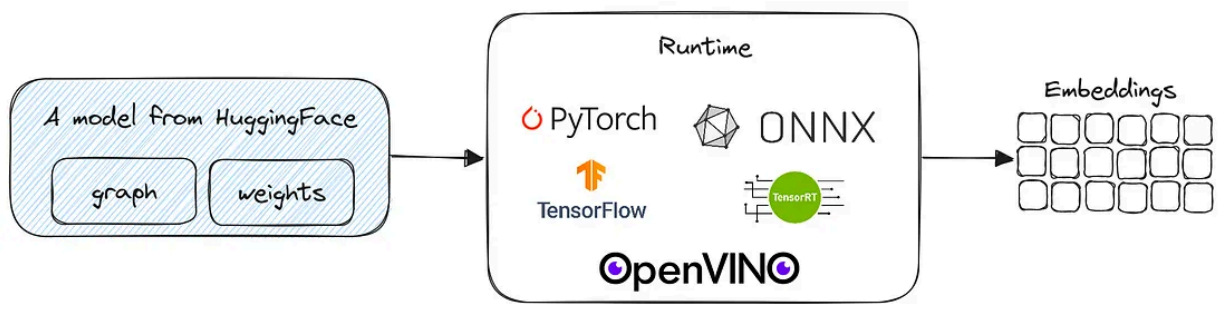
There are two options for computing **zero_point** and **scale** quantization parameters for each operation in the graph:

- **Dynamic**: each operator re-computes these parameters during runtime for each batch. It is more resource-heavy but also more precise — these parameters may drift between batches.
- **Static**: instead of re-computing these parameters each time for every batch, we do an offline calibration — feed-forward the network with a tiny dataset and record quantization parameters statically based on distributions observed. There is no extra overhead, but as quant parameters are static, they may be imperfect in a case of drift between batches.

The main drawback of static quantization is the need to perform calibration, which is an extra manual step. Since static quantization typically leads to worse precision, we will focus on **the dynamic quantization approach** in this article.

LLM inference runtime

Model files you find on HuggingFace Hub contain only a definition of matrix operations (an execution graph) you need to execute on top of model weights.



Execution runtime is the thing doing the actual matrix multiplication. Image by author.

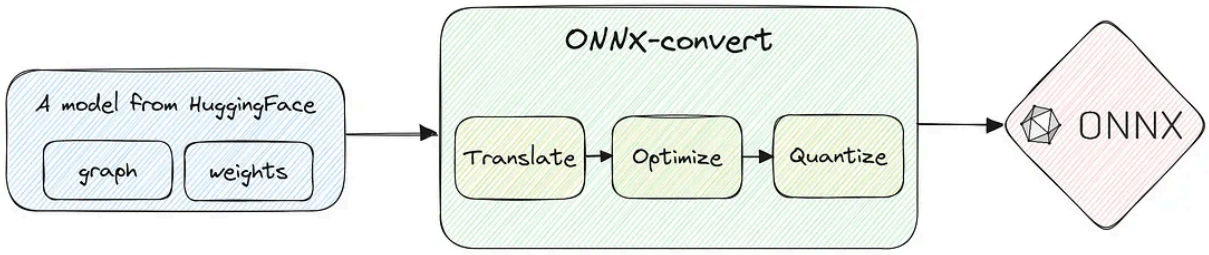
It is the execution runtime that interprets and executes this graph on top of your hardware:

- **PyTorch and TensorFlow** — used for model development, very Python-centric, but there are low-level bindings for other languages like C/C++. In practice, both are more optimized for training and batch processing on GPU.
- **OpenVINO** — a CPU-focused runtime from Intel, Python/C++ only.
- **ONNX** — an open multi-language (Java/JS/WASM/C++/Python) and multi-backend (CPU/GPU/TPU) runtime.
- **TensorRT** — an ONNX-compatible runtime from Nvidia specialized only in GPU execution.

Tied to Apache Lucene, Nixiesearch is implemented as a JVM application — and as other open-source search engines chose to use ONNX as the primary execution runtime for neural networks.

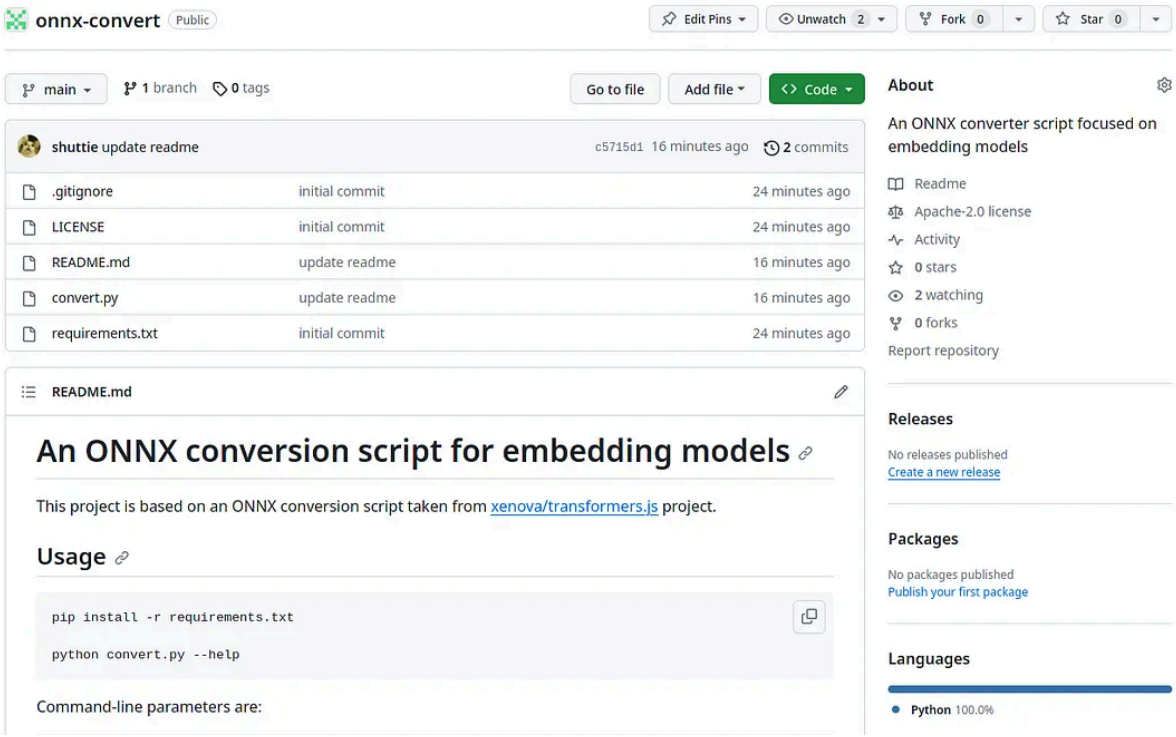
Converting models to ONNX

You need to convert in first to execute a model inside the ONNX runtime.



ONNX model conversion flow. Image by author.

For such conversion, we developed a tool nixiesearch/onnx-convert, which is an extended version of a conversion script from the original xenova/transformers.js project. The converted model files are compatible with any ONNX-flavored search engine and can be used in Elasticsearch Inference Processor, Vespa Embedder and directly in Nixiesearch.



nixiesearch/onnx-convert conversion tool. Image by author.

The ONNX conversion+quantization process has multiple important tunable parameters affecting both performance and quality:

- **Underlying quantization format**: it can be a signed/unsigned 8-bit integer and 16-bit float. How does this affect performance and precision?
- **ONNX transformer optimizer**: ONNX can fuse multiple typical operators into a single optimized core for things like attention blocks. Does it really matter, and how does its level affect end inference latency?

Apart from these variable parameters, we fix other less important ones to recommended constant values:

- **ONNX opset**: ONNX has multiple versions with different support for data types and operators. We chose the latest **opset=17** supported by the Python onnx package.
- **Per-channel quantization**: should the graph track a single per-tensor *scale* and *zero_point* values, or should it be done per channel, a slice in the

tensor? Turning it off may slightly increase performance and decrease precision — but with a cost of extra memory used, we chose to enable it by default.

- **7-bit scale**: as extra protection from numeric overflows, should 8-bit values be shrunk a bit? ONNX docs state that it may improve precision on older hardware without AVX-VNNI.

We will take an E5-v2 family of embedding models, taking **small**, **base**, and **large** variants to see the impact of each change on models of different sizes.

QUint8/QInt8/Float16 and inference latency

We convert the **e5-small-v2**, **e5-base-v2** and **e5–large-v2** to **QUint8**, **QInt8** and **Float16** numerical types, still without any optimization yet:

```
$> python convert.py --model_id intfloat/e5-large-v2 --quantize Float16 --opset 17 --optimize 0

Conversion config: ConversionArguments(model_id='intfloat/e5-large-v2', quantize='Float16',
output_parent_dir='/home/shutty/data/models', task='sentence-similarity', opset=17, device='cpu',
skip_validation=False, per_channel=True, reduce_range=True, optimize=0)
Exporting model to ONNX
Framework not specified. Using pt to export to ONNX.
Using the export variant default. Available variants are:
    - default: The default ONNX variant.
Using framework PyTorch: 2.1.0+cu121
Overriding 1 configuration item(s)
        - use_cache -> False
Post-processing the exported models...
Deduplicating shared (tied) weights...
Validating ONNX model /home/shutty/data/models/intfloat/e5-large-v2/model.onnx...
        -[✓] ONNX model output names match reference model (last_hidden_state)
        - Validating ONNX Model output "last_hidden_state":
                -[✓] (2, 16, 1024) matches (2, 16, 1024)
                -[✓] all values close (atol: 0.0001)
The ONNX export succeeded and the exported model was saved at: /home/shutty/data/models/intfloat/e5-
large-v2
Processing model file /home/shutty/data/models/intfloat/e5-large-v2/model.onnx
ONNX model loaded
No optimization enabled, quantizing to Float16

Export done
```

ONNX conversion tool. Image by author.

And run an embedding-benchmark suite:

- For each model of each size, compute embedding latency using onnxruntime in JVM.
- The suite is run on the last generation of AWS **M7I.2xlarge** instance with 8 VCPU supporting AVX-VNNI.

	e5-small-v2			e5-base-v2			e5-large-v2		
tokens	QUint8	QInt8	Float16	QUint8	QInt8	Float16	QUint8	QInt8	Float16
4	1.38	1.39	0.11	3.30	3.52	0.13	3.64	3.57	0.14
8	1.25	1.24	0.11	2.92	2.98	0.13	3.32	3.31	0.14
16	1.37	1.36	0.16	2.87	2.87	0.17	3.24	3.20	0.17
32	1.50	1.51	0.24	2.74	2.73	0.24	3.19	3.19	0.23
64	1.61	1.61	0.30	2.72	2.70	0.31	3.47	3.49	0.30
128	1.69	1.69	0.40	2.86	2.86	0.40	3.46	3.47	0.39
256	1.82	1.82	0.49	2.49	2.51	0.52	3.34	3.38	0.51

Relative inference time improvement between Float32 and other formats. Higher is better. Image by author.

The table above reads, "*for e5-small-v2 model the QUInt8 format for 4 tokens* ***is 1.38X faster*** *than Float32*". And the results we got are pretty surprising:

- The good news is that we got **more than 3X faster inference** for base and large models!
- There is **little difference between QUInt8 and QInt8** on VNNI hardware.
- The mixed-precision **Float16 format was unexpectedly 2x-7x SLOWER** than the baseline. This surprising fact comes from the lack of FP16/BF16 support in modern CPUs: only the latest Intel Sapphire Rapids Xeon CPUs with AMX support can handle these data types natively. CPUs without AMX do downcast-upcast operations each time they spot a Float16 type.

So you obviously should quantize your model, but what if your hardware has no VNNI support?

Impact of AVX-VNNI

We're explicitly mentioning AVX-VNNI support in CPU for a reason. While writing this article, the author was stuck with not being able to replicate performance numbers seen in the article "Accelerating Transformer-based Embedding Retrieval with Vespa" on model quantization. The root cause was the AMD Zen2 CPU not supporting the VNNI instruction set.

VNNI is an AVX extension for Vector Neural Network Instructions and supported on Intel CPUs made from 2019+ and on AMD Zen 4+:

Designer	Microarchitecture	Year	Support Level																
			F	CD	ER	PF	BW	DQ	VL	FP16	IFMA	VBMI	VBMI2	BITALG	VPOPCNTDQ	VP2INTERSECT	4VNNIW	4FMAPS	VNNI	BF16
Intel	Knights Landing	2016	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘
	Knights Mill	2017	✔	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✔	✘	✔	✔	✘	✘
	Skylake (server)	2017	✔	✔	✘	✘	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘	✘
	Cannon Lake	2018	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘
	Cascade Lake	2019	✔	✔	✘	✘	✔	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘	✘	✔	✘
	Cooper Lake	2020	✔	✔	✘	✘	✔	✔	✔	✔	✘	✘			✘	✘	✘	✘	✔	✔
	Tiger Lake	2020	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✔	✘	✘	✔	
	Rocket Lake	2021	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘	✘	✘	✔	
	Alder Lake	2021	✔	✔	✘	✘	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	✘	✘	✔	✔
	Ice Lake (server)	2021	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘	✘	✘	✔	✘
	Sapphire Rapids	2023	✔																	✔
AMD	Zen 4	2022	✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✔	✔	✔	✘	✘	✘	✔	✔
Centaur	CHA		✔	✔	✘	✘	✔	✔	✔	✘	✔	✔	✘	✘	✘	✘	✘	✘	✘	✘

VNNI CPU support. Image from https://en.wikichip.org/wiki/x86/avx512_vnni.

On a non-VNNI AMD Zen2 CPU, the results are quite different:

	e5-small-v2			e5-base-v2			e5-large-v2		
tokens	QUint8	QInt8	Float16	QUint8	QInt8	Float16	QUint8	QInt8	Float16
4	1.32	1.38	0.11	1.75	1.84	0.09	1.92	2.00	0.10
8	1.24	1.30	0.12	1.46	1.57	0.10	1.61	1.67	0.11
16	1.12	1.19	0.15	1.24	1.35	0.12	1.28	1.40	0.13
32	1.06	1.15	0.23	1.06	1.21	0.21	1.07	1.25	0.22
64	1.02	1.09	0.28	1.04	1.18	0.28	1.05	1.24	0.29
128	1.04	1.13	0.38	1.02	1.21	0.37	1.05	1.25	0.40
256	1.04	1.15	0.48	1.01	1.19	0.51	1.03	1.22	0.53

Non-VNNI CPU relative inference difference with Float32 baseline. Lower is better. Image by author.

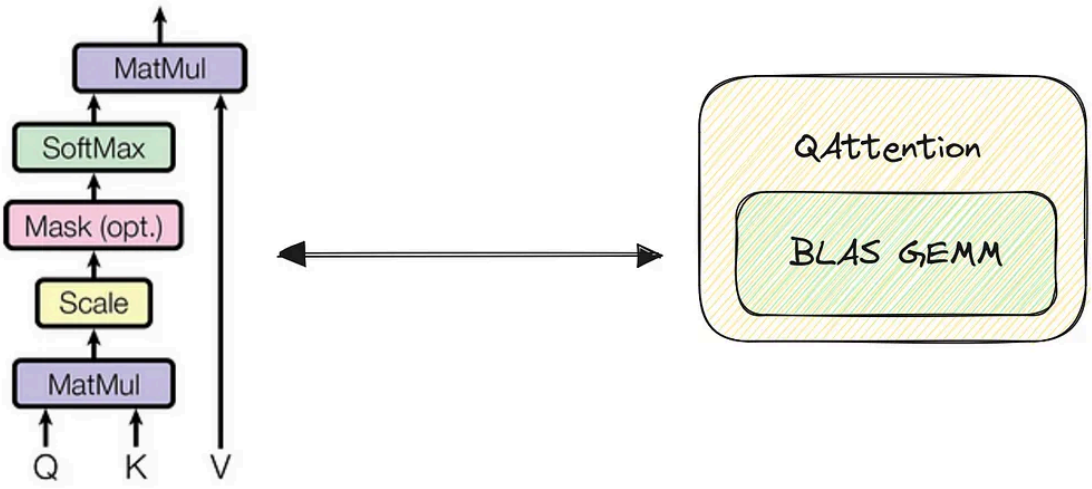
There are two main important observations:

- For the **QInt8/QUInt8, the speedup of 1.2x-1.6x is less dramatic** than for VNNI CPU.
- Compared to VNNI CPU, there is a difference between QInt8 and QUInt8: signed **QInt8 is almost always around 15% faster** than unsigned QUInt8.

So, even without VNNI support, the latency improvement of quantization is still worth it.

Optimizing the model

All the results above were made with 1–1 PyTorch to ONNX translated models. But ONNX has a set of specially fused kernels for transformer LLMs, like QAttention. A fused kernel is a group of matrix operations performed together, usually in a heavily-optimized way.



A chain of matrix ops replaced with a QAttention block implemented as a single BLAS GEMM method. Image by author.

So instead of performing multiple separate operations like MatMul-Scale-Mask-etc with many intermediate tensors, you can just replace them with a single QAttention block, which is technically a single BLAS GEMM function call.

ONNX has 4 optimizer levels:

- **0:** perform 1–1 translation without any optimization.
- **1:** do graph-only fusion like QAttention described above.
- **2:** as 1, but also apply fusion to Python-only blocks.
- **99:** as 2, but with RELU approximation.

	e5-base-v2											
	Float32			Float16			QInt8			QUInt8		
tokens	1	2	99	1	2	99	1	2	99	1	2	99
4	1.09	1.01	0.99	0.73	1.02	0.99	1.26	1.00	0.98	1.25	0.98	1.01
8	1.08	1.01	1.00	0.74	1.01	1.00	1.24	1.03	0.97	1.29	0.97	1.00
16	1.08	1.03	0.99	0.76	1.02	1.00	1.28	1.03	0.99	1.31	1.00	0.98
32	1.12	1.00	1.00	0.81	1.02	1.00	1.36	1.02	1.00	1.36	1.01	1.01
64	1.11	1.03	0.99	0.87	1.02	0.99	1.33	1.04	1.01	1.37	1.01	0.99
128	1.08	1.05	0.99	0.95	1.01	1.00	1.24	1.03	1.00	1.24	1.03	1.00
256	1.09	1.05	0.99	1.11	1.03	1.00	1.19	1.03	1.00	1.20	1.04	0.99

Stacked relative improvement, on top of each level. Image by author.

The table above can be read as *"for QInt8 format with 64 token input, optimization with level 1 gives +33% faster inference, and level 2 adds extra +4% on top of level 1"*.

The main observations are:

- ONNX optimization makes a difference: **QInt8/QUInt8 became ~30% faster!** There is still ~10% improvement, even for the non-quantized model.
- Level 1 significantly improves latency, and **level 2 almost always offers an extra 3–4% on top**.
- **Level 99 slightly degrades performance by 1–2%** compared to level 2.

So, you should also perform ONNX optimization as it makes your quantized model even faster with no extra cost.

Embedding quality on BEIR/MTEB

As the quantization process is not lossless, how does it affect embedding quality? We will use a subset of the BEIR-MTEB reference benchmark to see how quantized models degrade in quality.

	per-chan	7-bit	SciFact	NFCorpus	FiQA2018
Float32	yes	yes	0.6615	0.3222	0.3594
Float16	yes	yes	0.6614	0.3218	0.3593
QUInt8	yes	yes	0.6579	0.3146	0.3518
QInt8	yes	yes	0.6537	0.3177	0.3544
QInt8	yes	no	0.6462	0.3192	0.3494
QInt8	no	yes	0.5999	0.2988	0.2886
QInt8	no	no	0.633	0.3169	0.3379

NGCG@10 on 3 BEIR datasets. Float32 is a baseline without any quantization. Only small model and only three datasets due to too long running time of quantized models on CPU. Image by author.

There is a couple of surprising conclusions we can make:

- **Float16** quantization has **almost no degradation** on these three datasets. But considering it's severe performance implications while running on CPU, it's not a good option.
- **QUint8** and **QInt8** give slightly worse results, but the drop can still be considered tolerable, given how fast they are.
- **Per-channel quantization and 7-bit scaling** are two essential parameters: not using them during quantization often results in a significant drop in precision.

So, it would be best if you were extra accurate while choosing parameters for quantization: doing it wrong may result in a severe quality penalty. We advise using the onnx-convert tool to save yourself from shooting in the foot.

Conclusion

So if you're looking for a way to improve the inference latency of your embedding models, you **should definitely consider ONNX quantization and optimization combo**. Here are absolute before-and-after numbers we got for **e5–base-v2** model:

	e5-small-v2		e5-base-v2		e5-large-v2	
tokens	Float32	QInt8	Float32	QInt8	Float32	QInt8
4	2.29	1.187	9.83	2.351	37.297	9.087
8	2.42	1.418	10.182	2.738	38.432	10.189
16	3.725	1.977	13.088	3.779	49.135	12.641
32	7.129	3.311	20.549	6.085	76.184	18.696
64	10.785	4.427	29.969	9.171	109.822	25.887
128	19.65	7.941	49.961	15.537	186.172	41.736
256	40.798	17.274	106.586	38.04	393.489	96.671

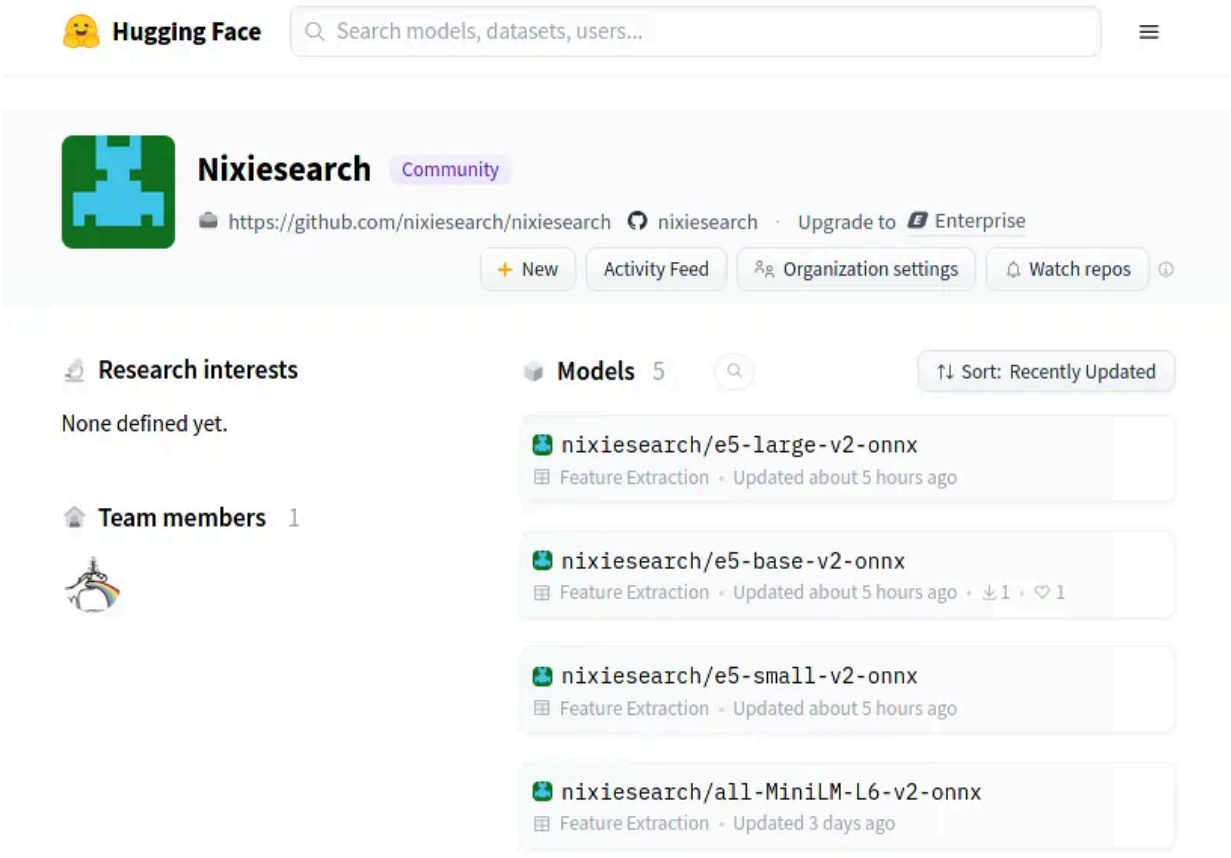
Absolute inference numbers in milliseconds, Float32 with optimize=0, QInt8 with optimize=2. Image by author.

With such an approach, we could go **down from 50ms to 15ms** for long documents on the e5-base-v2 model, **which is a 3.5x improvement in latency!**

Model quantization is an excellent way of improving the inference latency of embedding models:

- **2x-4x inference improvement is possible**, which comes at a price of minor quality degradation.
- **Prefer the QInt8 format** due to better performance on non-VNNI CPU.
- Stick to **ONNX optimization levels of 1 and 2** — they are a reasonable choice with no extra cost.
- Prefer **per-channel quantization with 7-bit scaling:** this results in the most minor model quality degradation.

All the E5 models tested in this article are published on https://huggingface.co/nixiesearch in three flavors: the raw ONNX non-optimized version, optimized version, and optimized+quantized version.



huggingface.co/nixiesearch namespace. Image by author.

But with the onnx-convert tool you can convert both open-source and proprietary models to the optimized and quantized ONNX format to

Links

- HuggingFace Optimum concept guide on quantization.
- The Basics of Quantization in Machine Learning (ML) for Beginners blog post.
- The How to accelerate and compress neural networks with quantization blog post.
- ONNX model quantization documentation page.

Machine Learning | Data Science | Llm | Search | Mlops

Published in Nixiesearch

15 Followers · Last published Oct 10, 2024

Follow

Here we write about modern search in the era of LLMs.

Written by Roman Grebennikov

554 Followers · 84 Following

Follow

Metarank maintainer

Responses (1)

Write a response

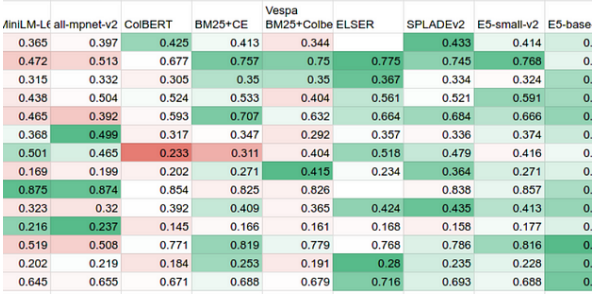
What are your thoughts?

katopz
Nov 25, 2023

Can you compare with this one 👉 https://github.com/huggingface/text-embeddings-inference

2 Reply

More from Roman Grebennikov and Nixiesearch



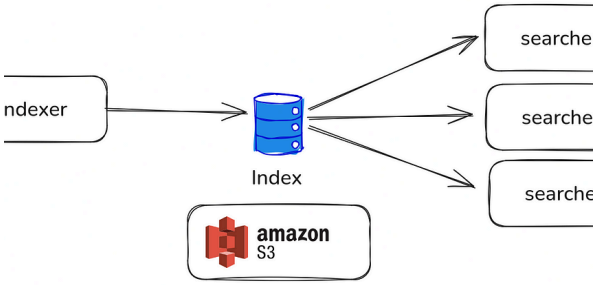
MiniLM-L6	all-mpnet-v2	ColBERT	BM25+CE	Vespa BM25+Colbe	ELSER	SPLADEv2	E5-small-v2	E5-base
0.365	0.397	0.425	0.413	0.344		0.433	0.414	0.
0.472	0.513	0.677	0.757	0.75	0.775	0.745	0.768	0.
0.315	0.332	0.305	0.35	0.35	0.367	0.334	0.324	0.
0.438	0.504	0.524	0.533	0.404	0.561	0.521	0.591	0.
0.465	0.392	0.593	0.707	0.632	0.664	0.684	0.666	0.
0.368	0.499	0.317	0.347	0.292	0.357	0.336	0.374	0.
0.501	0.465	0.233	0.311	0.404	0.518	0.479	0.416	0.
0.169	0.199	0.202	0.271	0.415	0.234	0.364	0.271	0.
0.875	0.874	0.854	0.825	0.826		0.838	0.857	0.
0.323	0.32	0.392	0.409	0.365	0.424	0.435	0.413	0.
0.216	0.237	0.145	0.166	0.161	0.168	0.158	0.177	0.
0.519	0.508	0.771	0.819	0.779	0.768	0.786	0.816	0.
0.202	0.219	0.184	0.253	0.191	0.28	0.235	0.228	0.
0.645	0.655	0.671	0.688	0.679	0.716	0.693	0.688	0.

In Metarank by Roman Grebennikov

From zero to semantic search embedding model

A series of articles on building an accurate Large Language Model for neural search fro...

Jun 12, 2023 1K 15

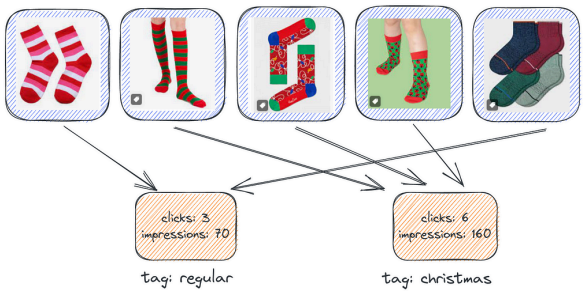


In Nixiesearch by Roman Grebennikov

Nixiesearch: running Lucene over S3, and why we're building a new...

A new search engine in 2024? Yes, but stateless—index on S3, serverless—no...

Oct 10, 2024 2

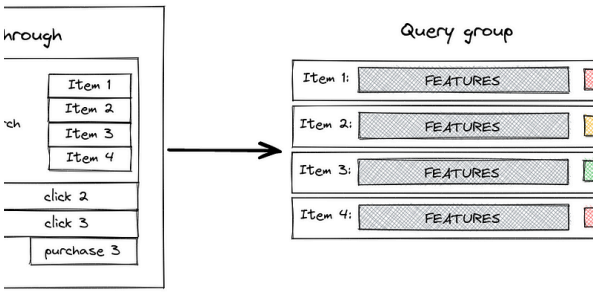


In Metarank by Roman Grebennikov

Solving a search cold-start problem with aggregated CTR

How do you rank a new item in search results when you have no visitor feedback? It was...

Mar 14, 2023 57



In Findify by Roman Grebennikov

DIY ML feature store

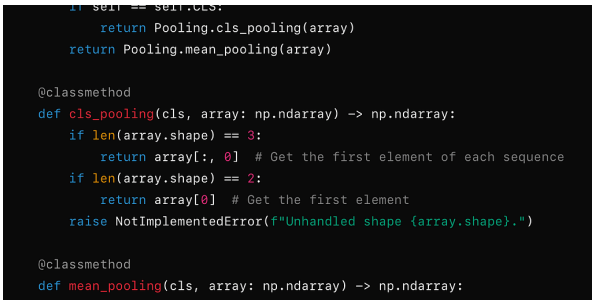
At Findify, all the product listings and search results we generate go through a hand-...

Jan 3, 2022 16 1

See all from Roman Grebennikov

See all from Nixiesearch

Recommended from Medium

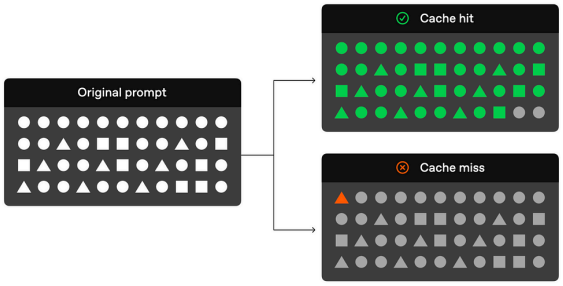


Suvasis Mukherjee

Pooling in Embedding

Pooling is typically used in natural language processing (NLP) to reduce the...

Sep 27, 2024 2

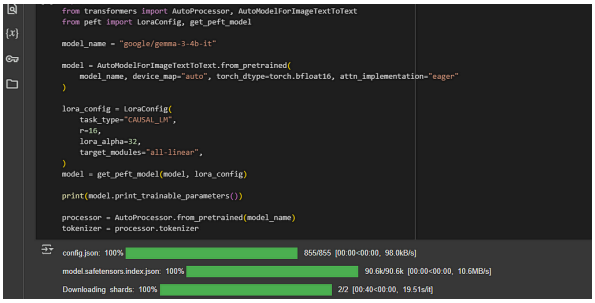


Nishi Ajmera

Risks of Prompt Caching

Many LLM providers have implemented prompt caching to improve response latenc...

Feb 20



In Artificial Intelligence in Plain Engli... by Eric Ris...

Fine-Tuning Google's Gemma-3-12B for Reasoning: How GRPO...

Artificial intelligence can speak fluently, create art, and even pass exams—but logica...

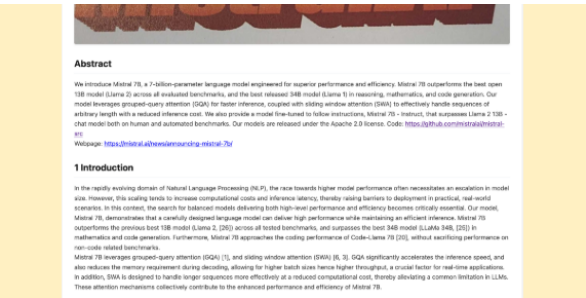
Mar 13 61

In The Deep Hub by Jorgecardete

Rust— A New Titan in Data Science

Revolutionizing Machine Learning with high-performance computation

Feb 20, 2024 2.1K 19



In Data And Beyond by TONI RAMCHANDANI

Mistral OCR: The Future of Document Understanding

In today's fast-paced digital world, turning paper documents, PDFs, and images into...

Mar 18 50

In Stackademic by Mohit Bajaj

How I Optimized a Spring Boot Application to Handle 1M...

Discover the exact techniques I used to scale a Spring Boot application from handling 50K...

Mar 2 654 24

See more recommendations