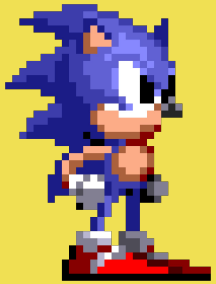


Small but Mighty

ONNX+ ONNXRT

In Production

vickiboykis.com

Our goals for this talk:

- Thinking about ML system design and tradeoffs
- Learn more about where smaller models work better
- Learn more about ONNX!

Building machine learning products that balance precision, recall, latency, and business needs is hard and requires good judgment and measurement

LinkedIn’s Job Search stack blends a number of models but it’s hard to get the product right

More filters coming soon

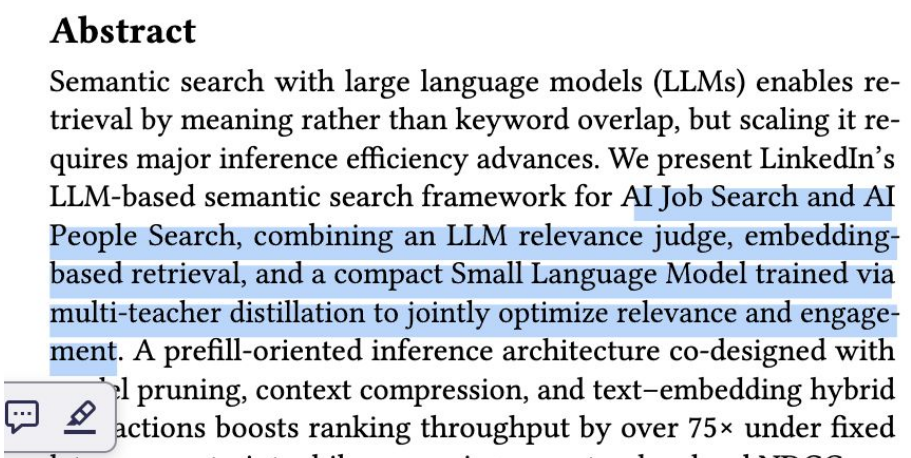
We understand that some members may still prefer filters, and that search results don’t always match what you’re looking for. We’re actively working to make results more relevant and bring back additional filters, including: **Verified job** (previously known as Has Verifications), **Sort by: Most recent / Most relevant** and **Hybrid / On-site**.

We’ll continue to update this article as changes are made and new functionality becomes available.

Here's a tip

Use **suggested filters** to quickly narrow your results. They appear right under the search bar and adapt based on your search. For example, searching “product manager jobs in San Francisco" might surface filters like Growth, B2B, SMB, or Digital, so you can easily find the roles that fit best. **Suggested filters are available globally in English, German, French, Spanish, and Portuguese, with more languages coming soon.**

Generative models currently make up only part of modern production machine learning inference workloads.



Abstract

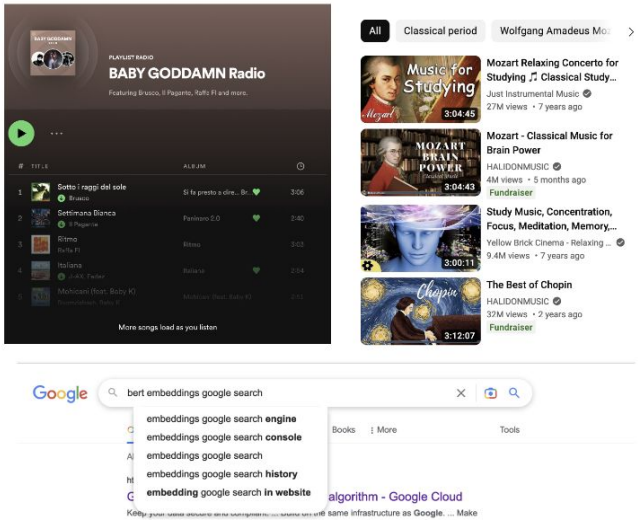
Semantic search with large language models (LLMs) enables retrieval by meaning rather than keyword overlap, but scaling it requires major inference efficiency advances. We present LinkedIn's LLM-based semantic search framework for AI Job Search and AI People Search, combining an LLM relevance judge, embedding-based retrieval, and a compact Small Language Model trained via multi-teacher distillation to jointly optimize relevance and engagement. A prefill-oriented inference architecture co-designed with l pruning, context compression, and text–embedding hybrid actions boosts ranking throughput by over 75× under fixed latency constraint while preserving near–teacher-level NDCG, enabling one of the first production LLM-based ranking systems with efficiency comparable to traditional approaches and delivering significant gains in quality and user engagement.

Source

We have an ML-driven retrieval product.

Should we use an ✨LLM✨?

Embedding-based retrieval was a hugely useful paradigm



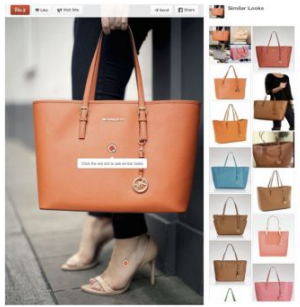


Figure 1: Similar Looks: We apply object detection to localize products such as bags and shoes. In this prototype, users click on objects of interest to view similar-looking products.

Figure 1: *Left to right: Products that use embeddings used to generate recommended items: Spotify Radio, YouTube Video recommendations, visual recommendations at Pinterest, BERT Embeddings in suggested Google search results*

Embedding-based retrieval is a hugely useful paradigm

Cursor

To power semantic search, Cursor builds a searchable index of your codebase when you open a project. For small projects, this happens almost instantly. But large repositories with tens of thousands of files can take hours to process if indexed naively, and semantic search isn't available until at least 80% of that work is finished.

Perplexity

X

Phoenix: Recommendation System

This repository contains the JAX code for the Phoenix recommendation system, which powers content ranking and retrieval. Phoenix uses transformer-based architectures for both **retrieval** (finding relevant candidates from millions of items) and **ranking** (ordering a smaller set of candidates by predicted engagement).

> **Note:** Earlier releases shipped a sample transformer ported from the Grok-1 open source release. This release ships the **production implementation itself**: the real model code, the real training step, and the real Rust serving engine, exported from the internal tree. What is *not* shipped is xAI-specific infrastructure (production data feeds, cluster orchestration, internal telemetry) — every such seam is replaced by a documented local equivalent, and synthetic data generators are included so the whole system runs end to end with nothing external. One training-recipe substitution is disclosed in TRAINING.md: for configs on the legacy dense-optimizer slot, the export ships standard AdamW rather than production's tuned internal variant. The flagship ranking configs and the nano twin train the production Muon recipe, which ships in full.

10 sources

does perplexity train its own embeddings for RAG to generate search results?

Finished 4s

Yes. Perplexity publicly states that it trains and serves its own embedding and ranking models for search, including its `pplx-embed` models. Its engineering post says: “We achieve state-of-the-art quality and latency by training and serving our own models, such as pplx-embed.” That’s direct evidence about its search infrastructure—not just an embedding API offered to developers. perplexity

Many information retrieval systems work by retrieving candidates from a large collection using similarity searches, then filtering and ranking those candidates to narrow the results.

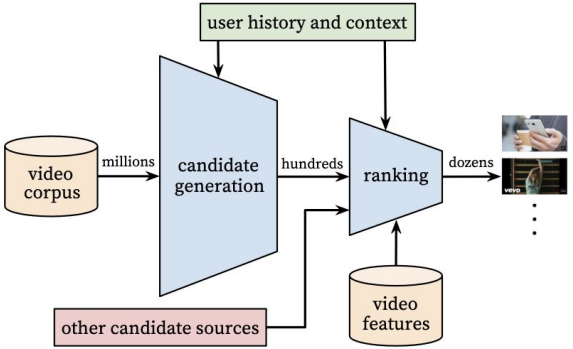
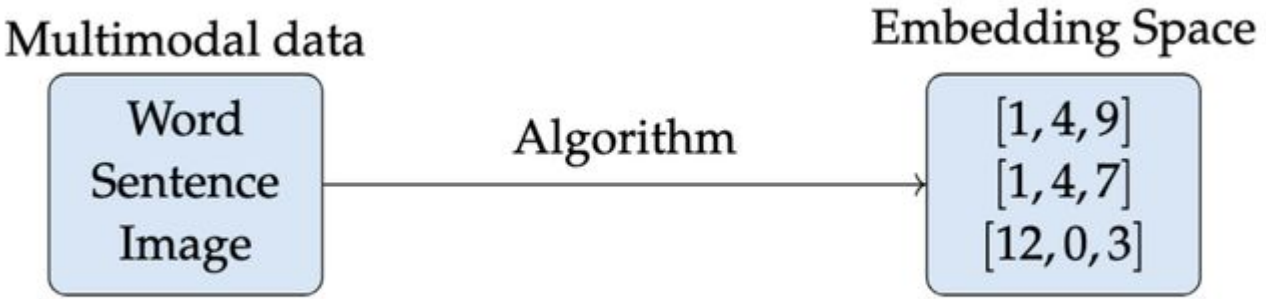


Figure 2: Recommendation system architecture demonstrating the "funnel" where candidate videos are retrieved and ranked before presenting only a few to the user.

Source

Embeddings are deep learning models' internal compressed representations of their input data that allow us to compare concepts or topics mathematically

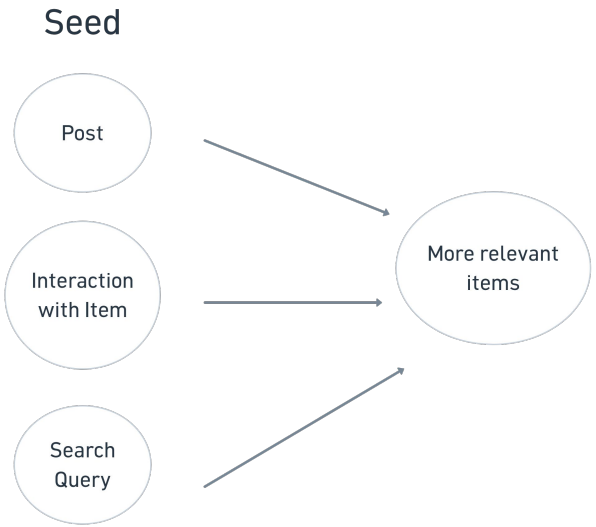


One embedding looks like this!

```
[tensor([-3.0241e-01, -1.5066e+00, -9.6222e-01,  1.7986e-01, -2.7384e+00,
                  -1.6749e-01,  7.4106e-01,  1.9655e+00,  4.9202e-01,
                   -2.0871e+00,
                  -5.8469e-01,  1.5016e+00,  8.2666e-01,  8.7033e-01,
                   8.5101e-01,
                   5.5919e-01, -1.4336e+00,  2.4679e+00,  1.3920e+00,
                   -3.9291e-01,
                  -1.2054e+00,  1.4637e+00,  1.9681e+00,  3.6572e-01,
                   3.1503e+00,
                  -4.4693e-01, -1.1637e+00,  2.8804e-01, -8.3749e-01,
                   1.5026e+00,
                  -2.1318e+00,  1.9633e+00, -4.5096e-01, -1.8215e+00,
                   3.2744e+00,
                   5.2591e-01,  1.0686e+00,  3.7893e-01, -1.0792e-01,
                   5.1342e-01,
                  -1.0443e+00,  1.7513e+00,  1.3895e-01, -6.6757e-01,
                   -4.8434e-01,
                  -2.1621e+00, -1.5593e+01,  1.5249e+00,  1.6911e+00,
                   -1.2916e+00,
                   1.2339e+00, -3.6064e-01, -9.6036e-01,  1.3226e+00,
                   1.6427e+00,
                   1.4588e+00, -1.8806e+00,  6.3620e-01,  1.1713e+00,
                   1.1050e+00, ...
                  2.1277e+00])
```

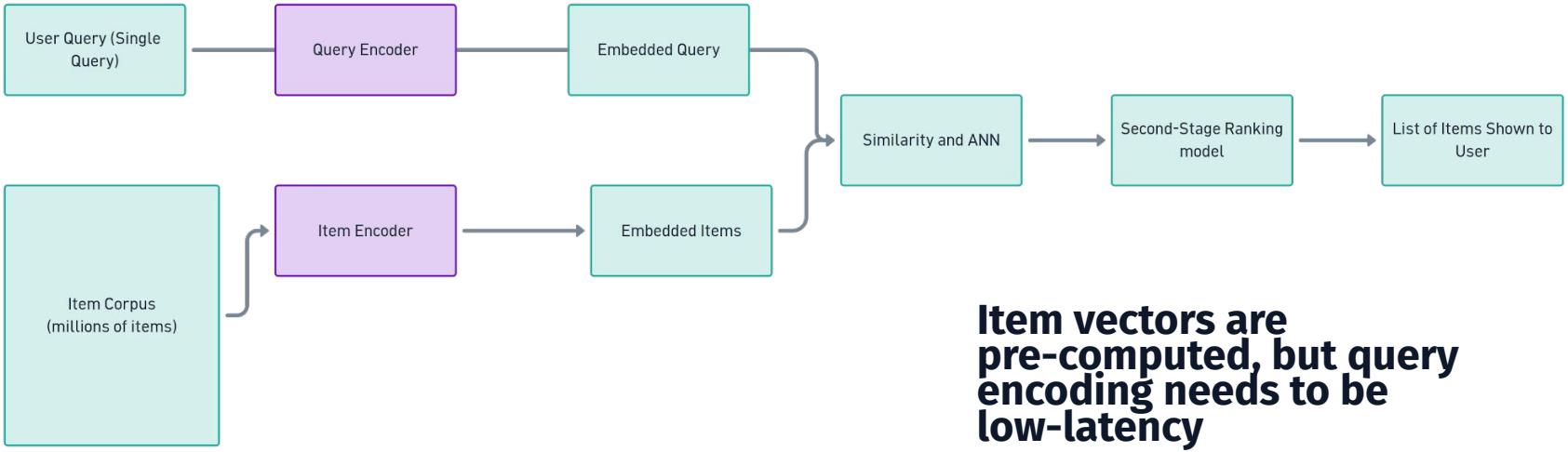
How do products like semantic search or a personalized social feed typically work?

We typically have a query and are trying to sort through our entire dataset to find relevant items



Two-tower embedding inference requires both batch and realtime inference of the same trained model artifact

Real-time inference

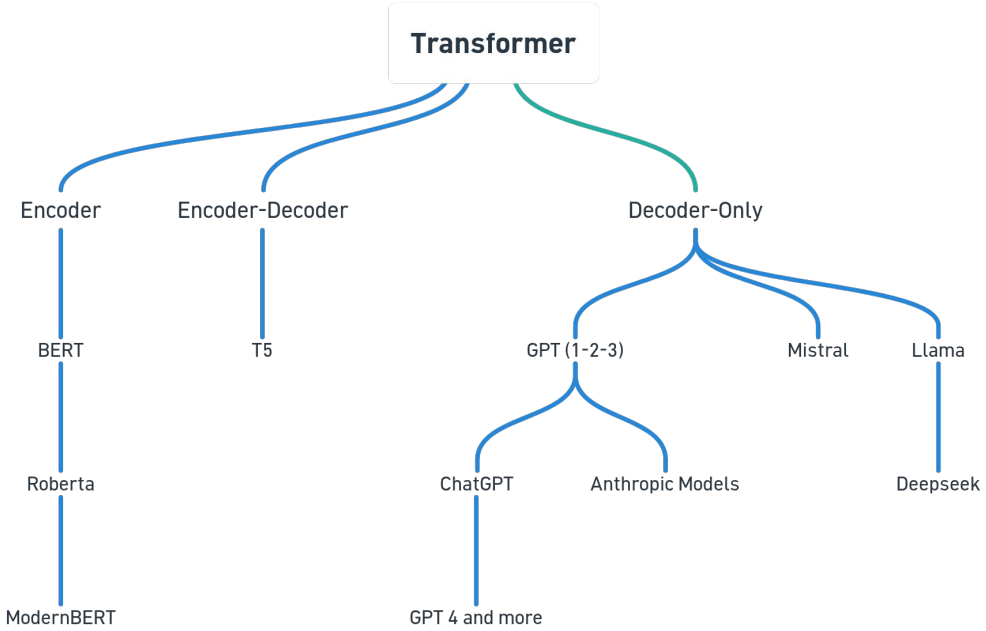


Item vectors are pre-computed, but query encoding needs to be low-latency

Batch inference

What's the difference between embedding models and LLMs?

LLM and embedding models share a transformer backbone but differ in parameter size, training objectives, and outputs



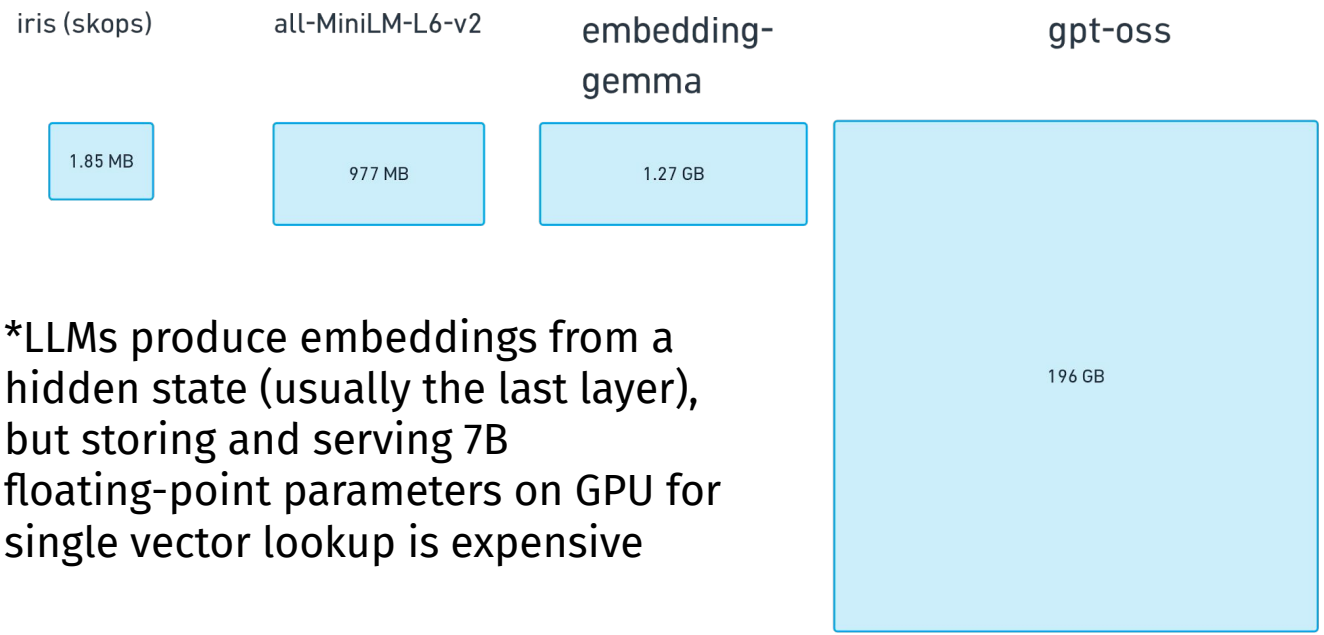
⚠️ Broad generalization, things are chagnging in this space, etc.

Embedding models and modern transformer architectures

	Embedding Model (Encoder-Only)	LLMs (Decoder-Only)
Input/Output	Text in , Vector Out	Text In, text out
Parameters	20M- 600M	7B-100B and beyond
Task	Small and specialized for understanding representations	Larger and more generalizable to many language tasks
Tasks	Classification, retrieval, semantic similarity, reranking, token labeling	Generation, summarization, completion, dialogue, code generation
Computation graph	Fixed, fully known	Dynamic shapes and control flow

⚠️ Broad generalization, things are chagnging in this space, etc.

Embedding models are much smaller than LLMs (though you could serve embeddings from an LLM)*



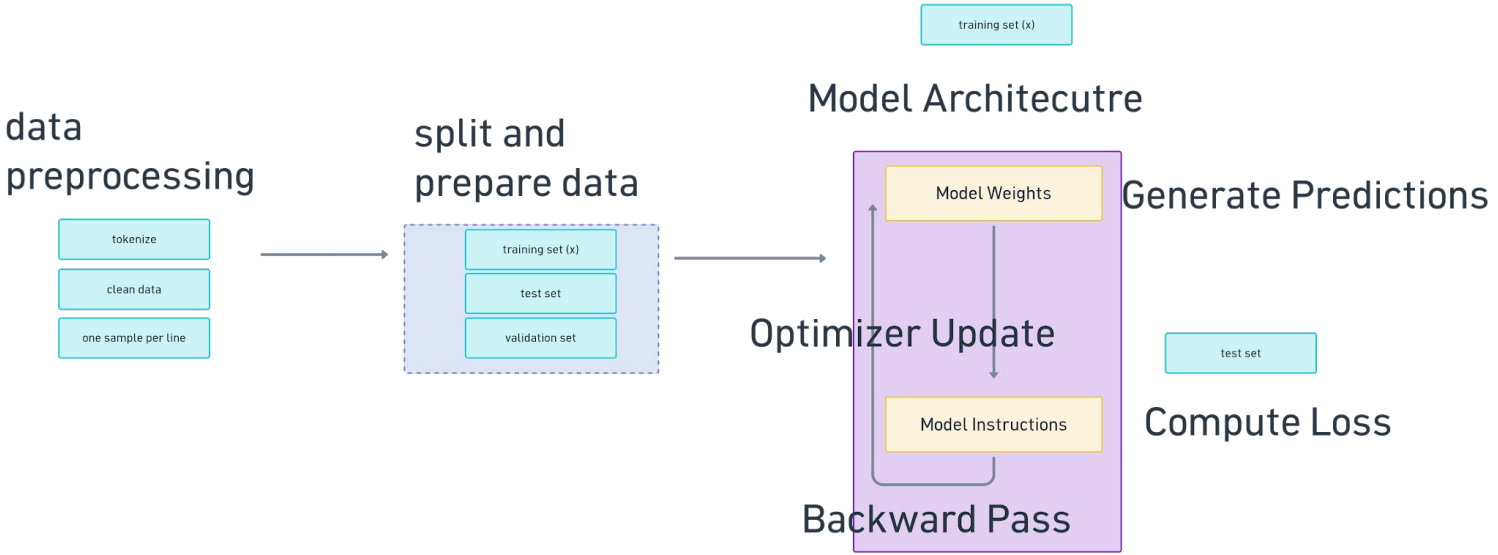
*LLMs produce embeddings from a hidden state (usually the last layer), but storing and serving 7B floating-point parameters on GPU for single vector lookup is expensive

Embedding model inference patterns are more predictable

	Embedding Models	Generative LLMs
Input	One prefill	Prefill, then iterative token generation
Output	Fixed-dimensional vector	Variable-length sequence of tokens
Bottleneck	fixed shape per request	The KV cache grows over an inference session
State	Stateless	Stateful KV cache between requests
Batching	Static batching	Continuous batching
Computation graph	Fixed, fully known	Dynamic shapes and control flow

How do we typically train and serve machine learning models?

When we train a model, we perform the following in a loop to get our model weights and model instructions



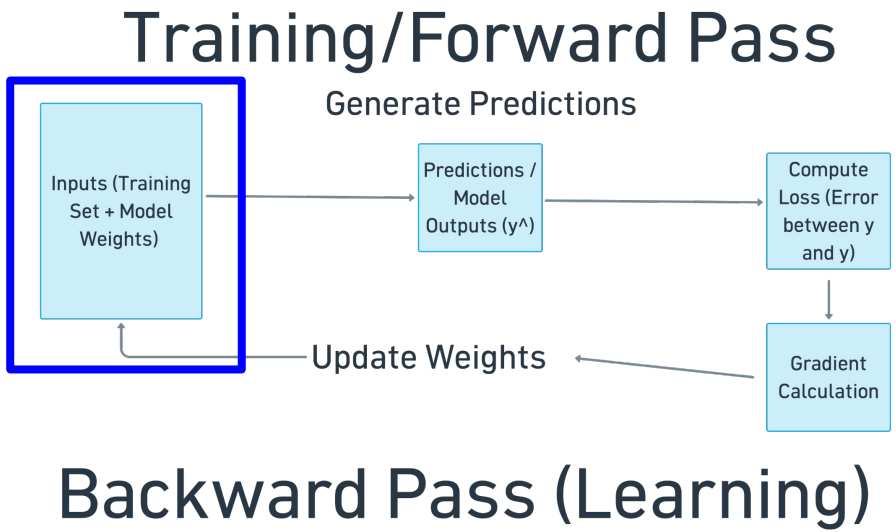
When we train a model, we initialize our weights as floats, our model architecture, and nn.Module in-memory execution set

```python
import torch
import torch.nn as nn
import torch.optim as optim

X = torch.tensor([[1.0], [2.0], [3.0], [4.0]], dtype=torch.float32)
y = torch.tensor([[2.0], [4.0], [6.0], [8.0]], dtype=torch.float32)
```

```python
class LinearRegression(nn.Module):
    def __init__(self):
        super(LinearRegression, self).__init__()
        self.linear = nn.Linear(1, 1)  # 1 input feature, 1 output feature

    def forward(self, x):
        return self.linear(x)
```



When we’re done training a model, we have the following artifacts:

- The model architecture
- The new set of model weights
- An executor (how the architecture is actually run)

The resulting trained model object contains the model metadata (OrderedDict), and weights (Pickle file)

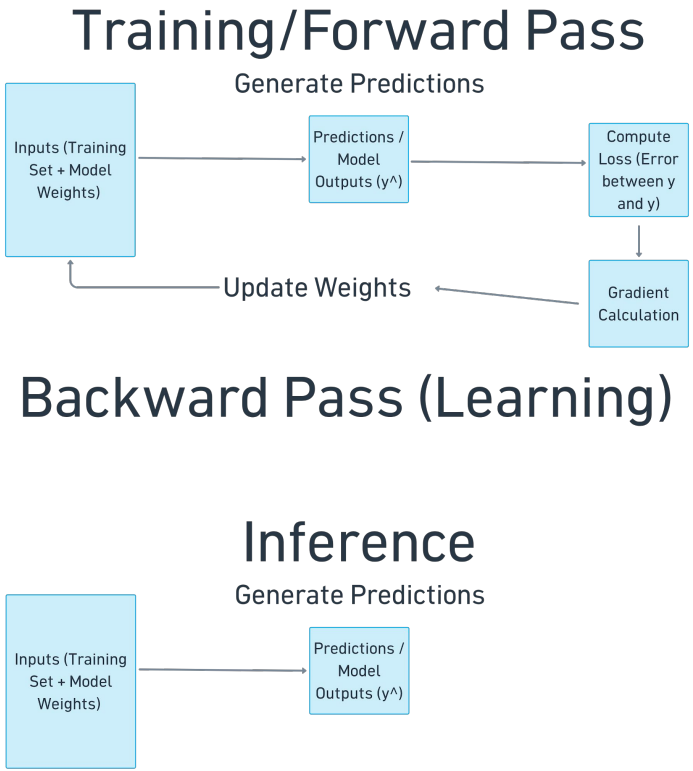
```
model = LinearRegression()
print(model.state_dict())

OrderedDict([('linear.weight', tensor([[0.5408]])), ('linear.bias',
tensor([-0.8195]))])

for param_tensor in model.state_dict():
    print(param_tensor, "\t", model.state_dict()[param_tensor].size())

linear.weight    torch.Size([1, 1])
linear.bias      torch.Size([1])
```

After training, we freeze the torch artifact. Model inference is training with model.eval() and torch.no_grad()



No: Backward pass, no gradient update, no dropout, and no loss computation

PyTorch ML model artifacts are currently standardized on safetensors

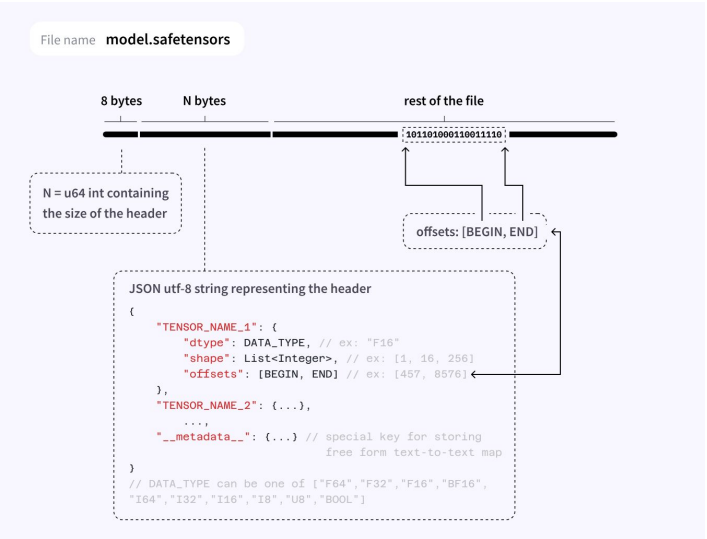
Pickle files (weights)

|

Safetensor files
(JSON header + weights)

|

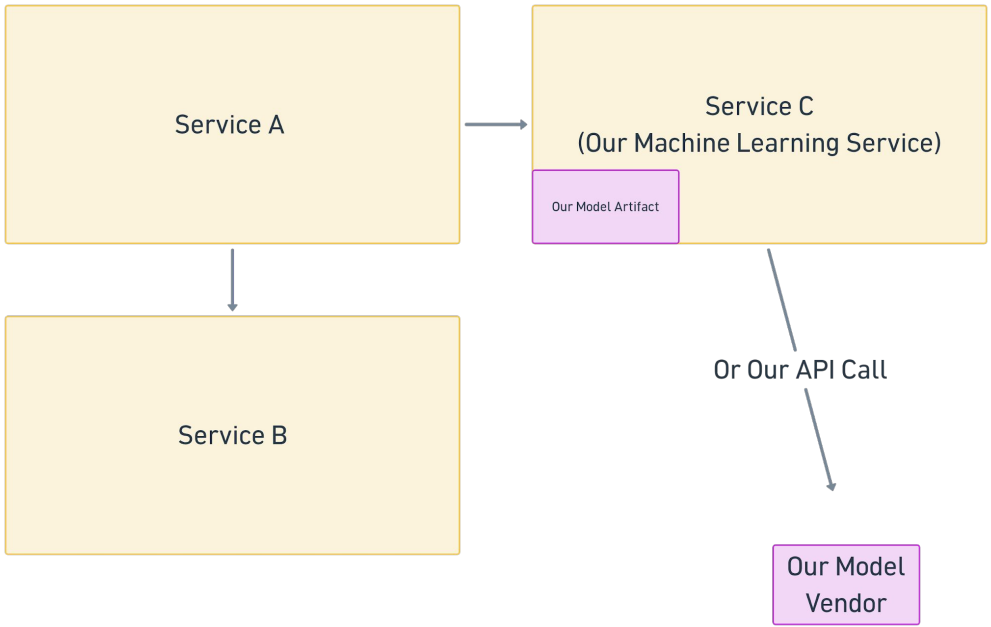
ONNX + ONNXRT (computational graph)



Source

How do we use a machine learning model in our service architecture?

In production, we will need to inference our embeddings either via artifact or API



What are the considerations of having machine learning models as service dependencies?

Real world issues with large-scale production system inference with self-hosted models

- Safetensors inference takes up 30% of our route roundtrip
- Torch dependencies
- Persisting a large bundle of artifacts between machines in distributed systems is hard

Why ONNX for embedding inference?

- We want to serve trained transformer architecture embedding models across commodity machines with standardized build and deploy steps
- Torch models take a while to cold-start. We'd love to make them smaller.
- We'd also love to be able to serve without torch or GPU dependencies

We trained a model in safetensors. Should we just inference that?

We could start, but we'd need to tune it

- torch.compile()
- torch.inference_mode()
- Model quantization - makes it fast but changes the decimal representation
- Move to GPU Inference

These all take time to test and experiment with (and GPUs are more expensive!)

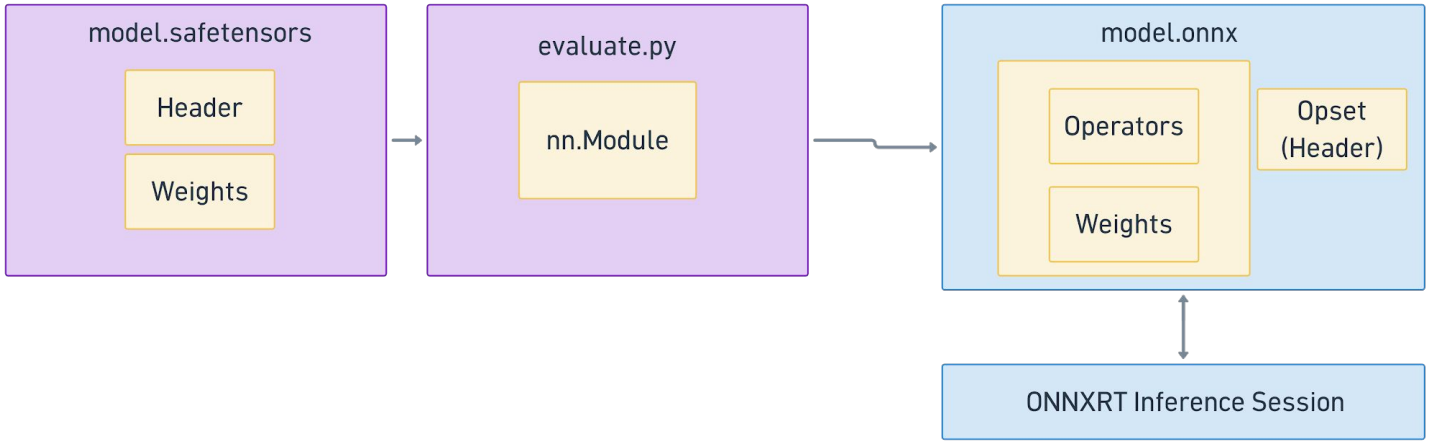
ONNX + ONNXRT main pluses are interop and optimizations

- **ONNX is an** open file specification **for machine learning models that can be used across ML frameworks, runtimes, and compilers.**

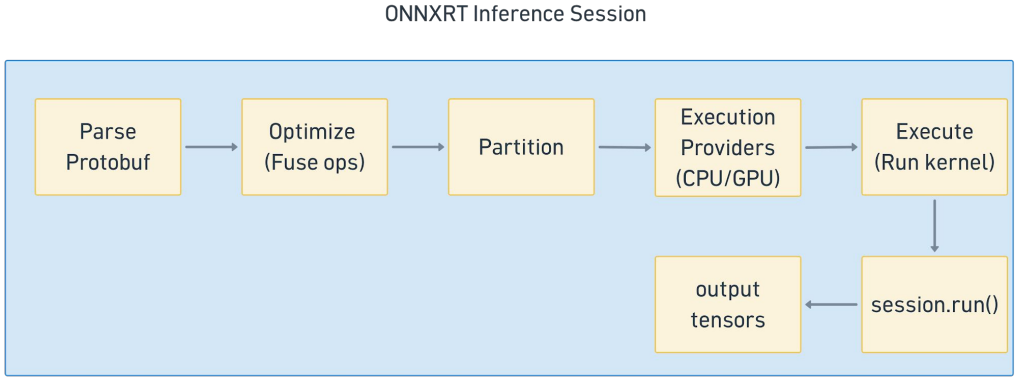
- **ONNX is** serialized using protobuf **and includes the computational graph and the model weights as tensors. It can be served on almost any hardware and doesn’t require torch dependencies**

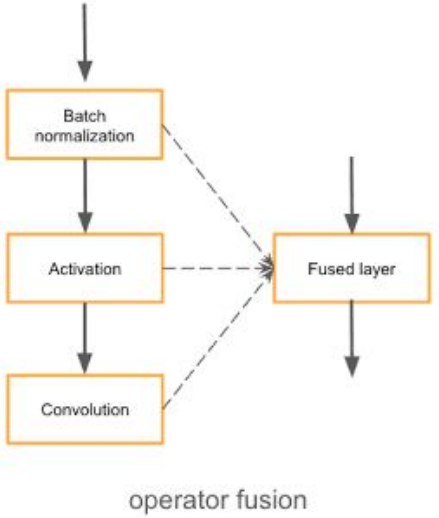
- **ONNXRT is the executor and provides graph fusion across providers (GPU and CPU) to serve the graph optimally and quickly.**

We convert our existing safetensors artifact into a model.onnx file and the graph into an ONNXRT inference session. We still use an external tokenizer.



The ONNXRT inference session is an in-memory object that handles our inference state and optimization





Source

Exporting to ONNX requires a model, opset, and weights and biases

```
with torch.no_grad():
        torch.onnx.export(
            model,
            sample_input['input_ids'],
            str(onnx_path),
            input_names=['input'],
            output_names=['output'],
            dynamic_axes={'input': {0: 'batch_size'}, 'output': {0: 'batch_size'}},
            opset_version=14,
            do_constant_folding=True,
            export_params=True,
            verbose=False
        )
```

Pytorch nn.Module

Output location

A sample tensor so PyTorch can infer shapes/sizes

Human-legible output labels

ONNX Operator version

Shrink model to speed up inference

Export weights and biases

An opset is ONNX's API (Versions of Operators the graph can use)

ai.onnx | ai.onnx.ml | ai.onnx.preview | ai.onnx.preview.training

operator	versions	differences
Abs	13, 6, 1	13/6, 13/1, 6/1
Acos	22, 7	22/7
Acosh	22, 9	22/9
Add	14, 13, 7, 6, 1	14/13, 14/7, 13/7, 14/6, 13/6, 7/6, 14/1, 13/1, 7/1, 6/1
AffineGrid	20	
And	7, 1	7/1
ArgMax	13, 12, 11, 1	13/12, 13/11, 12/11, 13/1, 12/1, 11/1
ArgMin	13, 12, 11, 1	13/12, 13/11, 12/11, 13/1, 12/1, 11/1
Asin	22, 7	22/7
Asinh	22, 9	22/9
Atan	22, 7	22/7
Atanh	22, 9	22/9
Attention	25, 24, 23	25/24, 25/23, 24/23
AveragePool	22, 19, 11, 10, 7, 1	22/19, 22/11, 19/11, 22/10, 19/10, 11/10, 22/7, 19/7, 11/7, 10/7, 22/1, 19/1, 11/1, 10/1, 7/1
BatchNormalization	15, 14, 9, 7, 6, 1	15/14, 15/9, 14/9, 15/7, 14/7, 9/7, 15/6, 14/6, 9/6, 7/6, 15/1, 14/1, 9/1, 7/1, 6/1
Bernoulli	22, 15	22/15
BitCast	26	
BitShift	28, 11	28/11

We can now use our artifact for inference

```python
# set the session options
sess_options = onnxruntime.SessionOptions()
sess_options.execution_mode = onnxruntime.ExecutionMode.ORT_SEQUENTIAL
sess_options.inter_op_num_threads = settings.INTER_OP_NUM_THREADS
sess_options.intra_op_num_threads = settings.INTRA_OP_NUM_THREADS


# Initializing inference session
ort_session = onnxruntime.InferenceSession(
    str(local_path), sess_options=sess_options
)



# Tokenization
inputs_transformers = tokenizer(
    text=data.query,
    return_tensors="pt",
    padding="max_length",
    truncation=True,
).to(device)


# Input/Output
onnx_input = {
    embedding_model.get_inputs()[0].name: inputs_transformers[
        "input_ids"
    ]
    .cpu()
    .numpy()
    .astype(numpy.int64)

text_embedding = embedding_model.run(None, onnx_input)[0]
```

Once we have our artifact, we will want to benchmark vector accuracy

```
 # our torch model set to infernce results
with torch.no_grad():
    pytorch_output = wrapper.forward(inputs_transformers['input_ids'])

# Setting up our ONNX inference session
onnx_input = {ort_session.get_inputs()[0].name:
inputs_transformers['input_ids'].cpu().numpy().astype(np.int64)}
onnx_output = ort_session.run(None, onnx_input)[0]

diff = np.abs(pytorch_output.cpu().numpy() - onnx_output).max()
logger.info(f"Max difference PyTorch vs ONNX: {diff}")
logger.info(f"Output shape: {onnx_output.shape}")

logger.info(f"{pytorch_output.cpu().numpy()}")
logger.info(f"{onnx_output}")

if diff < 1e-4:
    logger.info("ONNX export successful")
    return True
else:
    logger.info("Warning: Significant difference detected")
    return False
```

Once we have our artifact, we will want to benchmark inference latency

Run #	Median Time (ms)
1	137.05
2	132.87
3	133.85
4	133.87
5	130.2

Creating a test suite is important and easier than ever!

Outcomes after moving to ONNX

- Faster: Shaves 50ms off realtime inference with a budget of ~500ms for the request path
- Compatible: Creates a portable artifact free from torch dependencies
- Smaller: Compresses embedding model size from 813 MB optimized torch to 441 MB - 45% savings with quantization

Tradeoffs of moving to ONNX

- Changing vector precision, particularly in production, requires a lot of testing and work
- We now need to manage model state upon model creation (model artifactory)
- If we have high traffic volume, we need to orchestrate batch inference (build your own inference engine!)

By design, protobuf caps out at 2GB and moves weights to an external artifact (`.onnx_data`) which means we have more artifacts to manage

This explains why in native platforms it is important to make sure you **never rename the external data file**. If you rename the external data file, the “location” in the protobuf file will mismatch the actual file name, and ONNX Runtime will fail to load the model.

Considerations

- Save time for experimentation with different model architectures, opsets and session settings for CPU inference
- You'll still need to ship your tokenizer artifact
- Create a model tracking repository and versioning scheme
- You'll need to create a rigorous test suite in production for latency timings
    - Your local machine is often better than what you get in the cloud!

Thanks!

vickiboykis.com