Medium Search Write 12

Let's Implement LLM-JEPA from Scratch

azhar Following 12 min read · Just now

This post is a from-scratch implementation of the core idea behind LLM-JEPA, inspired by the paper *LLM-JEPA: Large Language Models Meet Joint Embedding Predictive Architectures*. This is **not** the official implementation released by the authors, and it does not claim to reproduce their reported numbers. Instead, I wrote a clean, minimal training script that captures the JEPA-style training objective for language: create two views of the same text, predict embeddings of masked spans, and train using a representation alignment loss. The goal of this article is practical understanding. I will walk through the full code line by line, explain the purpose of every function, and connect each piece back to the paper's intuition. Nothing is skipped, and every block of code shown is explained in detail so you can modify it confidently for your own experiments.

Before we proceed, let’s stay connected! Please consider following me on **Medium**, and don’t forget to connect with me on LinkedIn for a regular dose of data science and deep learning insights.” 🚀📊🤖

Code

LLM-JEPA (minimal, practical) training script in one file.

What it does
- Takes raw text
- Creates two views:
1) context view: full text with masked spans replaced by [MASK]
2) target view: the original text (unmasked), but we only supervise onmasked positions
- Context encoder (trainable) predicts target encoder representations at masked positions
- Target encoder is an EMA (momentum) copy of the context encoder (no grads)
- Loss is cosine distance between predicted embeddings and target embeddings on masked positions

```
Run (examples)
1) Tiny smoke test (no downloads, random init):
   python llm_jepa_train.py --smoke_test

2) Train with a HF model backbone:
   python llm_jepa_train.py --model_name distilbert-base-uncased --steps 200 --b

3) Train on your own text file:
   python llm_jepa_train.py --model_name distilbert-base-uncased --text_file dat
```

Notes

- This is a clean reference implementation, not the full repo codebase.

- Uses Transformers for the encoder backbone.

```
import argparse
import math
import os
import random
from dataclasses import dataclass
from typing import List, Tuple, Optional

import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.utils.data import Dataset, DataLoader
```

```
try:
    from transformers import AutoTokenizer, AutoModel, AutoConfig
except Exception:
    AutoTokenizer = None
    AutoModel = None
    AutoConfig = None


# ----------------------------
# Utilities
# ----------------------------
def set_seed(seed: int):
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)


def pick_device(device_str: str) -> torch.device:
    if device_str == "auto":
        return torch.device("cuda" if torch.cuda.is_available() else "cpu")
    return torch.device(device_str)


# ----------------------------
# Span masking (simple + effective)
# ----------------------------
def sample_span_mask(
    seq_len: int,
    mask_ratio: float,
    mean_span_len: int,
    special_positions: Optional[set] = None,
) -> torch.BoolTensor:
    """
    Returns a boolean mask of length seq_len indicating which positions are mask
    We mask contiguous spans until we reach approximately mask_ratio of tokens.
    """
    if special_positions is None:
```

```
        special_positions = set()

    mask = torch.zeros(seq_len, dtype=torch.bool)
    if seq_len <= 0:
        return mask

    target_to_mask = max(1, int(round(seq_len * mask_ratio)))
    masked = 0

    attempts = 0
    max_attempts = seq_len * 4

    while masked < target_to_mask and attempts < max_attempts:
        attempts += 1

        span_len = max(1, int(random.expovariate(1.0 / max(1, mean_span_len))))
        span_len = min(span_len, seq_len)

        start = random.randint(0, seq_len - 1)
        end = min(seq_len, start + span_len)

        span_positions = [i for i in range(start, end) if i not in special_posit
        if not span_positions:
            continue

        newly = 0
        for i in span_positions:
            if not mask[i]:
                mask[i] = True
                newly += 1

        masked += newly

    return mask


def apply_mask_to_input_ids(
```

```
        input_ids: torch.LongTensor,
        attention_mask: torch.LongTensor,
        tokenizer,
        mask_ratio: float,
        mean_span_len: int,
    ) -> Tuple[torch.LongTensor, torch.BoolTensor]:
        """
        Masks spans inside non-special, non-padding tokens.
        Returns:
          masked_input_ids: input ids with masked tokens replaced by [MASK]
          pred_mask: boolean mask over positions where we apply JEPA loss
        """
        assert input_ids.dim() == 1
        seq_len = int(attention_mask.sum().item())

        # Identify special token positions (CLS, SEP, etc.) in the visible region
        special_positions = set()
        for i in range(seq_len):
            tid = int(input_ids[i].item())
            if tid in {
                tokenizer.cls_token_id,
                tokenizer.sep_token_id,
                tokenizer.pad_token_id,
            }:
                special_positions.add(i)

        pred_mask = sample_span_mask(
            seq_len=seq_len,
            mask_ratio=mask_ratio,
            mean_span_len=mean_span_len,
            special_positions=special_positions,
        )

        masked_input_ids = input_ids.clone()
        mask_token_id = tokenizer.mask_token_id
        if mask_token_id is None:
            raise ValueError("Tokenizer has no mask_token_id. Use a model with [MASK
```

```
    # Replace masked positions with [MASK]
    masked_input_ids[:seq_len][pred_mask] = mask_token_id

    # pred_mask should be full length (includes pads as False)
    full_mask = torch.zeros_like(attention_mask, dtype=torch.bool)
    full_mask[:seq_len] = pred_mask

    return masked_input_ids, full_mask


# -----------------------------
# Dataset
# -----------------------------
class TextLinesDataset(Dataset):
    def __init__(self, texts: List[str]):
        self.texts = [t.strip() for t in texts if t.strip()]

    def __len__(self) -> int:
        return len(self.texts)

    def __getitem__(self, idx: int) -> str:
        return self.texts[idx]


def load_texts_from_file(path: str, max_lines: Optional[int] = None) -> List[str
    texts = []
    with open(path, "r", encoding="utf-8") as f:
        for i, line in enumerate(f):
            if max_lines is not None and i >= max_lines:
                break
            texts.append(line.rstrip("\n"))
    return texts


def default_tiny_corpus() -> List[str]:
    return [
```

```
        "The cat sat on the mat and looked at the window.",
        "A quick brown fox jumps over the lazy dog.",
        "Deep learning models can learn useful representations from raw data.",
        "Rocket Learning builds AI tools for education in India.",
        "Transformers use attention to mix information across tokens.",
        "Self-supervised learning can reduce the need for labels.",
        "JEPA trains models to predict embeddings, not tokens.",
        "Bengaluru is a major tech hub in India.",
        "A good system design balances simplicity and scalability.",
        "Reading code carefully helps you understand how an idea is implemented.
    ]


@dataclass
class Batch:
    input_ids: torch.LongTensor          # [B, L]
    attention_mask: torch.LongTensor     # [B, L]
    masked_input_ids: torch.LongTensor   # [B, L]
    pred_mask: torch.BoolTensor          # [B, L]  positions to compute loss on


def collate_jepa(
    batch_texts: List[str],
    tokenizer,
    max_length: int,
    mask_ratio: float,
    mean_span_len: int,
) -> Batch:
    toks = tokenizer(
        batch_texts,
        padding=True,
        truncation=True,
        max_length=max_length,
        return_tensors="pt",
    )
    input_ids = toks["input_ids"]               # [B, L]
    attention_mask = toks["attention_mask"]     # [B, L]
```

```
    masked_input_ids_list = []
    pred_mask_list = []

    for b in range(input_ids.size(0)):
        mi, pm = apply_mask_to_input_ids(
            input_ids[b],
            attention_mask[b],
            tokenizer,
            mask_ratio=mask_ratio,
            mean_span_len=mean_span_len,
        )
        masked_input_ids_list.append(mi)
        pred_mask_list.append(pm)

    masked_input_ids = torch.stack(masked_input_ids_list, dim=0)
    pred_mask = torch.stack(pred_mask_list, dim=0)

    return Batch(
        input_ids=input_ids,
        attention_mask=attention_mask,
        masked_input_ids=masked_input_ids,
        pred_mask=pred_mask,
    )


# ----------------------------
# Model: Encoder + Predictor + EMA target encoder
# ----------------------------
class PredictorMLP(nn.Module):
    def __init__(self, dim: int, hidden_mult: int = 4, dropout: float = 0.0):
        super().__init__()
        hidden = dim * hidden_mult
        self.net = nn.Sequential(
            nn.Linear(dim, hidden),
            nn.GELU(),
            nn.Dropout(dropout),
```

```
            nn.Linear(hidden, dim),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        return self.net(x)


class LLMJEPA(nn.Module):
    def __init__(self, encoder: nn.Module, dim: int, ema_m: float = 0.99, pred_h
        super().__init__()
        self.context_encoder = encoder
        self.target_encoder = self._copy_encoder(encoder)
        self.predictor = PredictorMLP(dim=dim, hidden_mult=pred_hidden_mult, dro
        self.ema_m = ema_m

        for p in self.target_encoder.parameters():
            p.requires_grad = False

    @staticmethod
    def _copy_encoder(enc: nn.Module) -> nn.Module:
        import copy
        return copy.deepcopy(enc)

    @torch.no_grad()
    def ema_update(self):
        m = self.ema_m
        for p_ctx, p_tgt in zip(self.context_encoder.parameters(), self.target_e
            p_tgt.data.mul_(m).add_(p_ctx.data, alpha=(1.0 - m))

    def forward(
        self,
        masked_input_ids: torch.LongTensor,
        input_ids: torch.LongTensor,
        attention_mask: torch.LongTensor,
        pred_mask: torch.BoolTensor,
    ) -> torch.Tensor:
        """
```

```
        Returns JEPA loss (scalar).
        We compute:
          z_ctx = context_encoder(masked_input)
          z_tgt = target_encoder(full input)
          pred = predictor(z_ctx)
          loss over positions in pred_mask
        """
        out_ctx = self.context_encoder(input_ids=masked_input_ids, attention_mas
        z_ctx = out_ctx.last_hidden_state  # [B, L, D]

        with torch.no_grad():
            out_tgt = self.target_encoder(input_ids=input_ids, attention_mask=at
            z_tgt = out_tgt.last_hidden_state  # [B, L, D]

        pred = self.predictor(z_ctx)  # [B, L, D]

        # Select masked positions
        # pred_mask: [B, L] bool
        masked_pred = pred[pred_mask]  # [N, D]
        masked_tgt = z_tgt[pred_mask]  # [N, D]

        if masked_pred.numel() == 0:
            # Safety: if a batch ends up with no masked tokens, return zero loss
            return pred.sum() * 0.0

        masked_pred = F.normalize(masked_pred, dim=-1)
        masked_tgt = F.normalize(masked_tgt, dim=-1)

        # Cosine distance
        loss = 1.0 - (masked_pred * masked_tgt).sum(dim=-1)
        return loss.mean()


# ----------------------------
# Training
# ----------------------------
def build_hf_encoder(model_name: str):
```

```
    if AutoModel is None:
        raise RuntimeError("transformers is not installed. pip install transform

    config = AutoConfig.from_pretrained(model_name)
    encoder = AutoModel.from_pretrained(model_name, config=config)
    dim = int(config.hidden_size)
    return encoder, dim


def build_random_encoder(vocab_size: int = 30522, dim: int = 256, layers: int =
    """
    For smoke tests only: small Transformer encoder (random init).
    Requires a tokenizer with vocab mapping for ids.
    """
    encoder_layer = nn.TransformerEncoderLayer(d_model=dim, nhead=heads, batch_f
    transformer = nn.TransformerEncoder(encoder_layer, num_layers=layers)

    class TinyEncoder(nn.Module):
        def __init__(self):
            super().__init__()
            self.emb = nn.Embedding(vocab_size, dim)
            self.pos = nn.Embedding(512, dim)
            self.enc = transformer

        def forward(self, input_ids, attention_mask):
            B, L = input_ids.shape
            pos_ids = torch.arange(L, device=input_ids.device).unsqueeze(0).expa
            x = self.emb(input_ids) + self.pos(pos_ids)

            # attention_mask: 1 for keep, 0 for pad
            # transformer expects src_key_padding_mask: True for pad
            pad_mask = attention_mask == 0
            h = self.enc(x, src_key_padding_mask=pad_mask)
            return type("Out", (), {"last_hidden_state": h})

    return TinyEncoder(), dim
```

```
def save_checkpoint(path: str, model: LLMJEPA, optimizer: torch.optim.Optimizer,
    os.makedirs(os.path.dirname(path), exist_ok=True)
    torch.save(
        {
            "step": step,
            "context_encoder": model.context_encoder.state_dict(),
            "target_encoder": model.target_encoder.state_dict(),
            "predictor": model.predictor.state_dict(),
            "optimizer": optimizer.state_dict(),
        },
        path,
    )


def main():
    parser = argparse.ArgumentParser()
    parser.add_argument("--model_name", type=str, default="distilbert-base-uncas
    parser.add_argument("--text_file", type=str, default="", help="Path to a new
    parser.add_argument("--max_lines", type=int, default=50000)
    parser.add_argument("--max_length", type=int, default=128)
    parser.add_argument("--mask_ratio", type=float, default=0.3)
    parser.add_argument("--mean_span_len", type=int, default=5)
    parser.add_argument("--ema_m", type=float, default=0.99)
    parser.add_argument("--pred_hidden_mult", type=int, default=4)

    parser.add_argument("--batch_size", type=int, default=8)
    parser.add_argument("--lr", type=float, default=2e-5)
    parser.add_argument("--weight_decay", type=float, default=0.01)
    parser.add_argument("--steps", type=int, default=500)
    parser.add_argument("--warmup_steps", type=int, default=50)
    parser.add_argument("--log_every", type=int, default=25)
    parser.add_argument("--save_every", type=int, default=200)
    parser.add_argument("--save_path", type=str, default="checkpoints/llm_jepa.p

    parser.add_argument("--device", type=str, default="auto")
    parser.add_argument("--seed", type=int, default=42)
```

```
parser.add_argument("--smoke_test", action="store_true", help="No downloads,
args = parser.parse_args()

set_seed(args.seed)
device = pick_device(args.device)

if args.smoke_test:
    if AutoTokenizer is None:
        raise RuntimeError("transformers is required even for smoke_test (fo
    tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
    # Ensure mask token exists
    if tokenizer.mask_token_id is None:
        raise ValueError("Tokenizer must support [MASK]. Use a masked LM tok

    texts = default_tiny_corpus()
    ds = TextLinesDataset(texts)

    encoder, dim = build_random_encoder(vocab_size=int(tokenizer.vocab_size)
    model = LLMJEPA(encoder=encoder, dim=dim, ema_m=0.95, pred_hidden_mult=2

    lr = 1e-4
else:
    if AutoTokenizer is None:
        raise RuntimeError("transformers is not installed. pip install trans
    tokenizer = AutoTokenizer.from_pretrained(args.model_name)
    if tokenizer.mask_token_id is None:
        raise ValueError(
            "This tokenizer has no [MASK]. Pick a masked-encoder model (BERT
        )

    if args.text_file:
        texts = load_texts_from_file(args.text_file, max_lines=args.max_line
    else:
        texts = default_tiny_corpus()

    ds = TextLinesDataset(texts)
```

```
    encoder, dim = build_hf_encoder(args.model_name)
    model = LLMJEPA(encoder=encoder, dim=dim, ema_m=args.ema_m, pred_hidden_

    lr = args.lr

# DataLoader
def _collate(batch_texts):
    return collate_jepa(
        batch_texts=batch_texts,
        tokenizer=tokenizer,
        max_length=args.max_length,
        mask_ratio=args.mask_ratio,
        mean_span_len=args.mean_span_len,
    )

dl = DataLoader(ds, batch_size=args.batch_size, shuffle=True, drop_last=True

# Optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=lr, weight_decay=args.w

# Simple warmup + cosine schedule
def lr_at(step: int) -> float:
    if step < args.warmup_steps:
        return float(step + 1) / float(max(1, args.warmup_steps))
    progress = (step - args.warmup_steps) / float(max(1, args.steps - args.w
    progress = min(max(progress, 0.0), 1.0)
    return 0.5 * (1.0 + math.cos(math.pi * progress))

model.train()
running = 0.0
step = 0
data_iter = iter(dl)

while step < args.steps:
    try:
        batch = next(data_iter)
    except StopIteration:
```

```
        data_iter = iter(dl)
        batch = next(data_iter)

    # Move to device
    input_ids = batch.input_ids.to(device)
    attention_mask = batch.attention_mask.to(device)
    masked_input_ids = batch.masked_input_ids.to(device)
    pred_mask = batch.pred_mask.to(device)

    # LR schedule
    scale = lr_at(step)
    for pg in optimizer.param_groups:
        pg["lr"] = lr * scale

    loss = model(
        masked_input_ids=masked_input_ids,
        input_ids=input_ids,
        attention_mask=attention_mask,
        pred_mask=pred_mask,
    )

    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    torch.nn.utils.clip_grad_norm_(model.parameters(), 1.0)
    optimizer.step()

    # EMA update after optimizer step
    model.ema_update()

    running += float(loss.item())
    step += 1

    if step % args.log_every == 0:
        avg = running / float(args.log_every)
        running = 0.0
        print(f"step {step:6d} | loss {avg:.4f} | lr {optimizer.param_groups
```

```
            if step % args.save_every == 0:
                save_checkpoint(args.save_path, model, optimizer, step)
                print(f"saved checkpoint to {args.save_path} at step {step}")

    save_checkpoint(args.save_path, model, optimizer, step)
    print(f"training done. final checkpoint: {args.save_path}")


if __name__ == "__main__":
    main()
```

Big picture: what this script trains

This script is a **JEPA-style representation predictor for text.**

You feed in normal text lines. For each example, you create two "views":

1. **Masked view (context view)**
   Same sentence, but some spans are replaced by `[MASK]`.
2. **Original view (target view)**
   The original sentence with no masking.

Then you do this:

- Run the masked view through a **trainable context encoder.**

- Run the original view through a **non-trainable target encoder**.
- Train a **predictor** so that the context encoder's representations can predict the target encoder's representations, but only on masked positions.
- Keep the target encoder stable using **EMA (exponential moving average)** updates.

This encourages the model to learn representations that "fill in" meaning rather than predicting exact tokens.

1) `set_seed(seed: int)`

```
def set_seed(seed: int):
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed_all(seed)
```

What it does

This ensures your run is reproducible.

- `random.seed(seed)` fixes Python random operations (span masking uses `random`).
- `torch.manual_seed(seed)` fixes randomness in PyTorch on CPU.
- `torch.cuda.manual_seed_all(seed)` fixes randomness in CUDA kernels if GPU is used.

Why it matters

Span masking and model initialization are random. Without seeds, results change every run.

2) `pick_device(device_str: str)`

```
def pick_device(device_str: str) -> torch.device:
    if device_str == "auto":
        return torch.device("cuda" if torch.cuda.is_available() else "cpu")
    return torch.device(device_str)
```

What it does

Returns a PyTorch device.

- If `--device auto`, it chooses GPU if available, else CPU.
- If you pass `--device cpu` or `--device cuda`, it uses that.

Why it matters

Your tensors and model must be on the same device.

3) `sample_span_mask(...)`

```
def sample_span_mask(seq_len, mask_ratio, mean_span_len, special_positions=None)
```

This is one of the most important functions in the whole script.

Goal

Create a boolean mask of which positions in the sequence should be masked.

Inputs

- `seq_len` : number of real tokens (excluding padding).
- `mask_ratio` : what fraction of tokens to mask, like 0.3.
- `mean_span_len` : average length of contiguous masked spans.
- `special_positions` : positions you should never mask (CLS, SEP, PAD).

Key logic inside

a) Create an all-false mask

```
mask = torch.zeros(seq_len, dtype=torch.bool)
```

b) Compute how many tokens we aim to mask

```
target_to_mask = max(1, int(round(seq_len * mask_ratio)))
```

Even short sequences will mask at least 1 token.

c) Repeatedly sample spans until enough tokens are masked

It runs a loop and keeps masking spans.

d) Span length sampling

```
span_len = max(1, int(random.expovariate(1.0 / max(1, mean_span_len))))
```

This draws span lengths from an exponential distribution. That creates a realistic span distribution: many short spans, some longer spans.

Then it clips the span length to max `seq_len`.

e) Sample a start index

```
start = random.randint(0, seq_len - 1)
end = min(seq_len, start + span_len)
```

f) Filter special tokens out

```
span_positions = [i for i in range(start, end) if i not in special_positions]
```

If the span overlaps CLS or SEP, those are skipped.

g) Mark positions True

```
if not mask[i]:
    mask[i] = True
    newly += 1
masked += newly
```

Why this function matters

Masking strategy heavily affects representation learning quality. Span masking encourages the model to infer missing meaning from surrounding context.

4) `apply_mask_to_input_ids(...)`

```
def apply_mask_to_input_ids(input_ids, attention_mask, tokenizer, mask_ratio, me
```

Goal

Take tokenized ids for one sample and produce:

- `masked_input_ids` : same shape, with masked positions replaced by `[MASK]` .
- `pred_mask` : boolean mask for loss computation positions.

Key parts

a) Calculate visible sequence length

```
seq_len = int(attention_mask.sum().item())
```

Because attention_mask is 1 for real tokens and 0 for padding.

b) Identify special token positions

```
if tid in {tokenizer.cls_token_id, tokenizer.sep_token_id, tokenizer.pad_token_i
```

You do not want to mask CLS or SEP because that can destabilize models.

c) Sample span mask for visible region

Uses `sample_span_mask`.

d) Replace masked positions with `tokenizer.mask_token_id`

```
masked_input_ids[:seq_len][pred_mask] = mask_token_id
```

Only in visible region.

e) Convert to full length mask

You return a mask shaped like input length (including padding), but padding positions are False.

Why return pred_mask

Because you only compute JEPA loss on positions that were masked. All other positions are ignored.

Dataset: `TextLinesDataset`

```
class TextLinesDataset(Dataset):
    def __init__(self, texts):
        self.texts = [t.strip() for t in texts if t.strip()]
```

What it does

Very simple dataset: holds a list of text lines and returns them.

- Removes empty lines
- Strips whitespace

`__len__` returns number of lines.

`__getitem__` returns one text string.

`load_texts_from_file(path, max_lines)`

Reads a file line-by-line.

- Stops at `max_lines` if provided.
- Returns a list of strings.

Used when you pass `--text_file`.

`default_tiny_corpus()`

Provides a built-in dataset for quick testing.

`Batch` dataclass

```
@dataclass
class Batch:
    input_ids
    attention_mask
    masked_input_ids
    pred_mask
```

Why it exists

Cleaner than returning a tuple. Makes code readable and self-documenting.

`collate_jepa(...)`

This is the function used by DataLoader to create batches.

Input

- list of raw texts from dataset.

Steps

1. Tokenize the batch with padding/truncation:

```
toks = tokenizer(batch_texts, padding=True, truncation=True, max_length=max_leng
```

This produces `input_ids` and `attention_mask`.

1. For each sample in batch:

- call `apply_mask_to_input_ids` to produce:
- masked input ids
- pred mask

1. Stack them to tensors of shape `[B, L]`.
2. Return a `Batch`.

Why collate matters

DataLoader reads examples one-by-one but training needs batches. This is where batching and masking happen.

`PredictorMLP`

This is the predictor head.

```
nn.Linear(dim, hidden)
nn.GELU()
nn.Dropout()
nn.Linear(hidden, dim)
```

Meaning

It maps context representations to target representation space.

You can think of it like a learned adapter that helps align embeddings.

`LLMJEPA` model class

This is the main model wrapper.

Components

1. `context_encoder` : trainable transformer encoder

2. `target_encoder` : deep copy, not trainable
3. `predictor` : MLP
4. `ema_m` : EMA momentum factor

`_copy_encoder(enc)`

Uses `copy.deepcopy` . This ensures target starts identical to context.

`ema_update()`

```
p_tgt = m * p_tgt + (1 - m) * p_ctx
```

This slowly updates target encoder weights.

- If `m=0.99` , target changes very slowly.

- This stabilizes training and reduces collapse risk.

`forward(...)`

Inputs:

- `masked_input_ids` : masked view
- `input_ids` : original view
- `attention_mask`
- `pred_mask` : positions to compute loss

Steps:

1. Context forward pass (trainable)

```
z_ctx = context_encoder(masked_input_ids).last_hidden_state
```

2. Target forward pass (no gradients)

```
z_tgt = target_encoder(input_ids).last_hidden_state
```

3. Predictor

```
pred = predictor(z_ctx)
```

4. Select masked positions:

```
masked_pred = pred[pred_mask]
masked_tgt = z_tgt[pred_mask]
```

So instead of `[B, L, D]`, you get `[N, D]` where N is total masked tokens.

1. Normalize vectors and compute cosine distance:

```
loss = 1 - (masked_pred * masked_tgt).sum(dim=-1)
return loss.mean()
```

Why normalize

Cosine similarity focuses on direction of embedding vectors, not magnitude.

`build_hf_encoder(model_name)`

Loads a Hugging Face encoder model and returns:

- encoder
- hidden dimension

The dim is read from config.hidden_size.

`build_random_encoder(...)`

Used for smoke test.

Creates a tiny transformer encoder from scratch:

- embedding layer
- positional embeddings
- transformer encoder stack

Important: This is NOT a masked LM. It is just a transformer encoder architecture.

Also it returns an object with `.last_hidden_state` to match HF output style.

This implementation deliberately favors clarity over completeness. It avoids custom attention masks, multi-view datasets, and hybrid objectives so the learning dynamics remain easy to inspect. As a result, it should be viewed as a **reference implementation**, not a production-ready system. The original LLM-JEPA paper goes further by combining JEPA with token prediction and by exploiting naturally paired views such as text and code. Those design choices are important for strong downstream performance, but they also add complexity that can obscure the underlying mechanism.

Llm Jepa Meta Transformer Alternative Bert

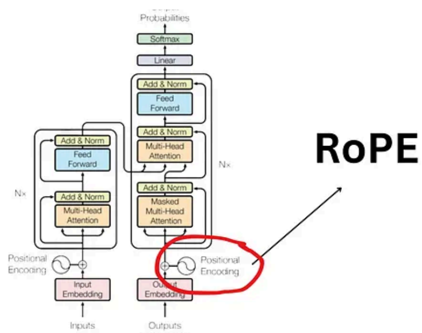
Written by azhar

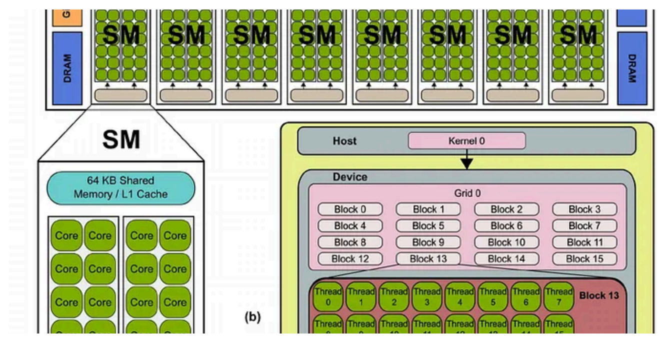
783 followers · 31 following

Following

Data Scientist | Exploring interesting (research paper / concepts). LinkedIn : https://www.linkedin.com/in/mohamed-azharudeen/

More from azhar





In azhar labs by azhar

In azhar labs by azhar

Rotary Positional Embeddings: A Detailed Look and Comprehensiv...

Since the "Attention Is All You Need" paper in 2017, the Transformer architecture has been...

Jan 11, 2024 316 9

Understanding GPU Architecture: Basics and Key Concepts

Graphics Processing Units (GPUs) have evolved from being specialized hardware for...

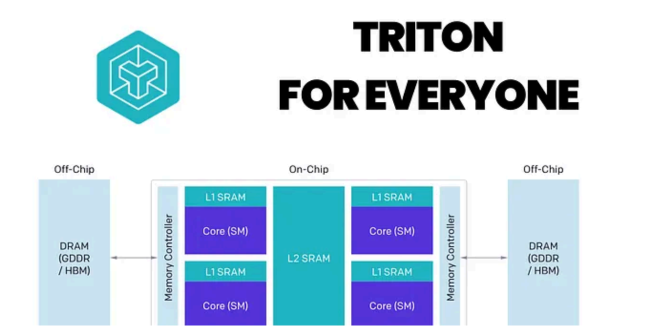
Mar 9, 2025 74 1

In azhar labs by azhar

Comprehensive Guide to Setting Up Claude Code on Windows Usin...

Setting up Claude Code, Anthropic's AI-driven coding assistant, on a Windows...

Mar 27, 2025 89 2



In azhar labs by azhar

Getting Started with Triton: A Step-by-Step Tutorial

Triton is an open-source GPU programming language and compiler that simplifies writin...

Mar 17, 2025 10 1

See all from azhar

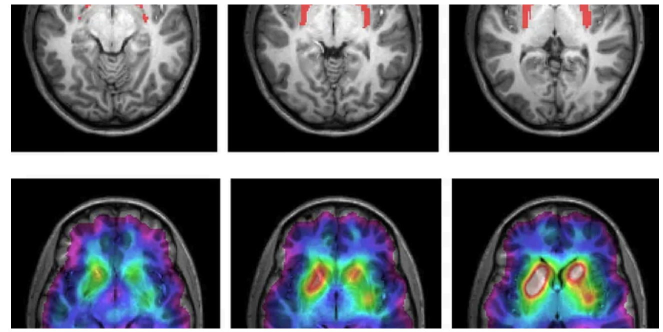
Recommended from Medium

Rice Yang

How Transformers Become Faster And Smarter: From KV Cache to...

Explore the fundamental evolution of the LLM decoder inference

Aug 16, 2025 8

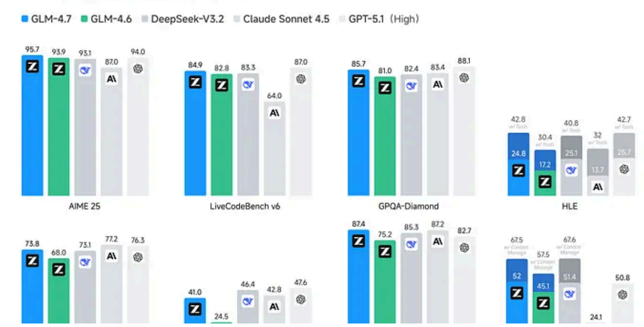


In Write A Catalyst by Dr. Patricia Schmidt

As a Neuroscientist, I Quit These 5 Morning Habits That Destroy You...

Most people do #1 within 10 minutes of waking (and it sabotages your entire day)

Jan 14 21K 355

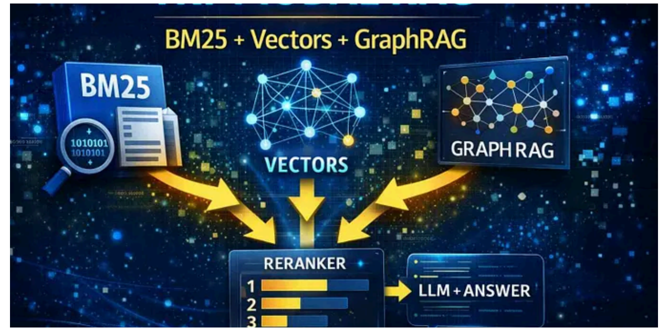


Agent Native

Local LLMs That Can Replace Claude Code

Small team of engineers can easily burn >$2K/mo on Anthropic’s Claude Code...

Jan 20 248 9



In Python in Plain English by Tarun Singh

Beyond Hybrid RAG That Actually Works: Vector + BM25 + GraphRA...

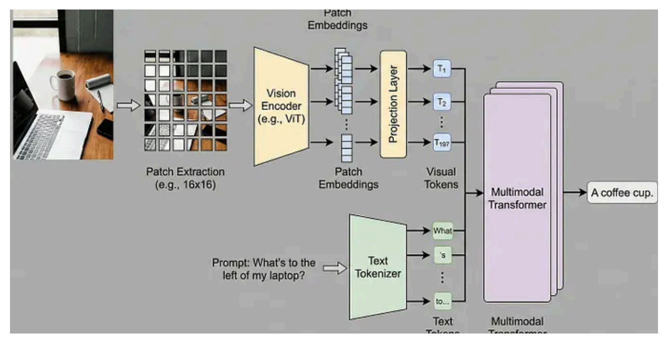
If you’re already using GraphRAG + Vector RAG, you’re ahead of most people.

Jan 18 30

Barack Obama

A Wake-Up Call for Every American

The killing of Alex Pretti is a heartbreaking tragedy. It should also be a wake-up call to...



Togo AI Labs

Understanding Visual Tokenization and the Gap Between Pixels and...

How Vision-Language Models Actually ‘See’

4d ago 55K 762

6d ago 137 2

See more recommendations