Contextualized Embeddings: A Complete Academic Manual

Prepared from instructor slides — expanded

This document expands instructor slides into a thorough, self-contained reference. It fills in missing intuition, concrete examples, math, and practical tips.

Contents

1 Introduction and scope

This manual dives deep into contextualized embeddings: definitions, how transformers produce them, practical issues (similarity, anisotropy), and downstream uses (classification, NER, NLI). The goal is to make every step explicit so the reader won't have to ask follow-ups.

2 Definitions and core concepts

2.1 Contextualized embedding — precise definition

Given an input token sequence $x_1, \dots, x_N$, the output vector $h_i^{(L)} \in \mathbb{R}^d$ from the final layer L of a Transformer encoder is called the *contextualized embedding* (or vector) for the token instance x_i. It represents the meaning of that token *in the given sentence*.

Alternative construction A common variant is to define the representation of token x_i as the average of the last four layers:

$$\bar{h}_i = \tfrac{1}{4}(h_i^{(L)} + h_i^{(L-1)} + h_i^{(L-2)} + h_i^{(L-3)}).$$

This often yields more stable vectors for downstream tasks.

2.2 Static vs contextualized embeddings

- **Static embeddings:** one vector per vocabulary word type (e.g., word2vec, GloVe). Does not depend on context.
- **Contextualized embeddings:** one vector per *token instance* (each occurrence can have a different vector depending on surrounding words).

"bank" as a word type has multiple contextual meanings. Static embedding: one fixed vector for b̈ank.̈ Contextual embedding: different vectors in r̈iver bank̈vs s̈avings bank.̈

3 How Transformers produce contextualized embeddings

We sketch the computations from embeddings to contextualized outputs, explaining why and how context influences each token.

3.1 Input embedding layer

Each token id is mapped to a learned token embedding and summed with a learned positional embedding (and optional segment embedding):

$$X = [e(x_1) + p_1,\ e(x_2) + p_2,\ \dots,\ e(x_N) + p_N] \in \mathbb{R}^{N \times d}.$$

These are the initial vectors fed into the transformer stack.

3.2 Single transformer encoder layer (math)

For layer ℓ, with input matrix $H^{(\ell-1)} \in \mathbb{R}^{N\times d}$, multi-head self-attention produces:

$$
\begin{aligned}
Q &= H^{(\ell-1)}W_Q, \quad K = H^{(\ell-1)}W_K, \quad V = H^{(\ell-1)}W_V, \\
\text{Attention} &= \text{softmax}\Big(\frac{QK^\top}{\sqrt{d_k}}\Big)V, \\
H^{(\ell)} &= \text{LayerNorm}(H^{(\ell-1)} + \text{Attention}), \\
H^{(\ell)} &= \text{LayerNorm}(H^{(\ell)} + \text{FFN}(H^{(\ell)})).
\end{aligned}
$$

The attention weights mix values V_j from other positions into the representation at each position. Repeating over L layers yields final outputs $H^{(L)}$ whose rows are the contextualized vectors.

3.3 Intuition: queries, keys, values

- Q_i (query) encodes *what* token i needs at this layer.
- K_j (key) encodes what token j can provide.
- V_j (value) is the content to be mixed in when token j is attended to.

The dot-product $Q_i \cdot K_j$ measures relevance; softmax turns these into mixture weights $\alpha_{i,j}$ used to compute the new representation via $\sum_j \alpha_{i,j} V_j$.

3.4 Why the vector for token i changes with context

Because Q_i, K_j, V_j depend on $H^{(\ell-1)}$, and $H^{(\ell-1)}$ already encodes context from earlier layers, the attention mixes information across positions: the new vector aggregates content from other positions weighted by relevance. Thus the same token type in different sentences will follow different attention patterns and produce different final vectors.

Important: the learned projection matrices W_Q, W_K, W_V are *shared* across tokens (and sometimes across positions) within a layer; the *vectors* Q_i, K_i, V_i are different per token because they are produced by multiplying the token's current representation by these shared matrices.

4 Properties of contextualized embeddings

4.1 Layerwise behavior and interpretability

Empirical studies (probing) show early layers capture surface features (POS, local n-grams), middle layers capture syntax/phrase structure, and later layers encode semantics and task-relevant features. This is approximate and depends on model size and training data.

4.2 Isotropy / anisotropy problem

Contextualized embeddings often suffer from *anisotropy*: vectors lie in a narrow cone of the space and many pairwise cosines are large. This makes raw cosine similarity less informative.

4.2.1 Why anisotropy arises

Training objectives, layernorms, and shared positional biases can create large-magnitude components that dominate vector geometry. The phenomenon is well-observed in BERT and other large pretrained models. In practice, a small number of dimensions (¨rogue dimensions¨) often have very high variance and dominate cosines.

4.2.2 Consequences

Similarity measures (cosine) can return high values for semantically unrelated instances. This breaks nearest-neighbor retrieval and clustering if used naively.

5 Practical fixes for similarity computations

5.1 Standardization (z-scoring)

Given a corpus set C of contextual vectors $x \in \mathbb{R}^d$, compute the empirical mean μ and per-dimension standard deviation σ:

$$\mu = \frac{1}{|C|} \sum_{x \in C} x, \qquad \sigma_j = \sqrt{\frac{1}{|C|} \sum_{x \in C} (x_j - \mu_j)^2}.$$

Then standardize each vector componentwise:

$$z = \frac{x - \mu}{\sigma}.$$

This reduces the influence of rogue dimensions and makes cosine scores more meaningful.

5.2 Whitening / PCA-based normalization

Alternatively, compute the covariance matrix Σ and whiten vectors via $z = \Sigma^{-1/2}(x - \mu)$ or use PCA to remove top principal components (the top-k components often capture corpus-level frequency/stopword effects). Removing the top 1–5 principal components is a common practical trick.

5.3 Layer selection and pooling strategies

Choices greatly affect downstream performance:

- Use final layer $h^{(L)}$ for specificity.
- Average last four layers for stability (reduces noise from last-layer specialization).
- Use mean pooling over token vectors to represent sequences.
- Use `[CLS]` token vector when model pretraining used it as a sequence summary (but test empirically: sometimes mean pooling or max pooling work better).

6 Extracting embeddings in practice (code/pseudocode)

```
# pseudocode (PyTorch-like):
# model: pretrained encoder that returns hidden states per layer
# tokens: tokenized input batch
outputs = model(tokens, output_hidden_states=True)
# outputs.hidden_states is a tuple length L+1: embeddings and each layer output
# choose pooling: final layer
h_final = outputs.hidden_states[-1]  # shape: (batch, seq_len, d)
# example: mean-last-4 pooling
h_last4 = torch.stack(outputs.hidden_states[-4:], dim=0)  # (4, batch, seq, d)
h_mean = h_last4.mean(dim=0)  # (batch, seq, d)
# standardize across a corpus: compute mu and sigma offline and apply
```

7 Downstream tasks and fine-tuning

This section explains how contextualized embeddings are used for common supervised tasks and how to fine-tune models.

7.1 Sequence classification

Goal: map an input sequence to one label (e.g., sentiment).

Common approaches:

1. Use the `[CLS]` token vector $h_{CLS}^{(L)}$ as sequence representation.
2. Mean-pool the final-layer token vectors to get a sequence vector.

Classifier head: Add a weight matrix $W_{cls} \in \mathbb{R}^{d \times C}$ (C=number of classes) and compute logits

$$u = h_{seq} W_{cls}, \qquad p = \text{softmax}(u).$$

Train with cross-entropy loss; fine-tune optionally updates both W_{cls} and the pretrained model's parameters (often full or partial).

7.1.1 Practical tips

- Use a small learning rate (e.g., 1e-5 to 5e-5) for pretrained weights and a slightly larger LR for the classifier head.
- Use early stopping and monitor validation loss; overfitting can be quick on small datasets.
- Consider freezing lower layers and only finetuning top layers for small datasets.

7.2 Pairwise sequence classification (NLI, paraphrase)

Input formatting: ¨[CLS] A [SEP] B [SEP]. The model processes the concatenated tokens; $h_{CLS}^{(L)}$ is fed to a classifier head for 2/3-way labels.

7.3 Sequence labeling (NER, POS)

Setup: label every token. Use final-layer token vectors $h_i^{(L)}$ and apply a token-level classifier $W_{tok} \in \mathbb{R}^{d \times K}$ producing distributions per token.

BIO tagging: for span tasks (NER), use BIO labels. Optionally add a CRF on top for structured prediction to enforce valid label sequences. The classifier per token can be trained with cross-entropy per token or with CRF loss for sequence constraints.

7.4 Fine-tuning strategies and regularization

- Weight decay and dropout help generalization.
- Gradual unfreezing: train classifier head first, then fine-tune top layers, then all layers.
- Parameter-efficient tuning: adapters or LoRA allow small updates without modifying full network.

8 Evaluation, probing, and interpretability

8.1 Probing classifiers

Probe whether a linguistic property is encoded in a layer by training a lightweight classifier (probe) on top of frozen layer outputs. Examples: POS, chunking, syntactic dependency.

8.2 Intrinsic vs extrinsic evaluation

- Intrinsic: embedding similarity, isotropy measures, clustering purity.
- Extrinsic: downstream task performance after finetuning (accuracy, F1).

8.3 Dimensionality reduction for visualization

Use PCA or t-SNE/UMAP on standardized vectors. Remove top principal components first if they capture corpus-level artifacts.

9 Worked numeric example (detailed)

We provide a full numeric example so the reader can compute by hand and see every transformation.

Toy setup: Vocabulary: The, cat, sat, on, mat. Embedding dim d=2 for simplicity.
Embeddings (initial):

$$e(\text{The}) = [1, 0], \quad e(\text{cat}) = [0, 1], \quad e(\text{sat}) = [1, 1].$$

Positional embeddings p1=[0.1,0], p2=[0,0.1], p3=[0.1,0.1].
Input sequence: T̈he cat saẗ. Compute X: rows are e+pos.
Assume a single-head attention with projection matrices equal to identity and no scaling (toy):

$$Q = K = V = X.$$

Compute scores S = Q $K^T (3x3), apply row-wise softmax to get attention matrices, then compute attention output$
$softmax(S)V. The final vectors are Attn (skip FFN for toy). The reader can compute explicit numeric values and se$
Concrete numbers (optional exercise): compute S, exponentiate, normalize, and obtain numeric attn vectors. This exercise makes concrete how context mixes tokens.

10 Best practices checklist

- When comparing contextual vectors use standardization or PCA-based decorrelation.
- Try last-4-layer averaging for stable features.
- For sequence classification compare mean-pooling vs CLS empirically.
- Use adapters/LoRA for parameter-efficient fine-tuning on small datasets.
- Monitor overfitting with small LR and weight decay when fine-tuning full model.

11 Further reading and references

- Devlin et al., B̈ERT: Pre-training of Deep Bidirectional Transformers for Language Understanding(̈2018).

- Ethayarajh, Ḧow contextual are contextualized word representations?̈(2019) — isotropy analysis.
- Hewitt & Manning, probing syntactic trees in BERT layers.

If you want, I can: (A) add detailed tikz diagrams that show attention flows layer-by-layer, (B) insert runnable PyTorch snippets that extract and standardize vectors on a real model, or (C) expand the numeric toy into full hand-calculable numbers. Which would you like next?