CS 489: Introduction to Natural Language Processing
Assignment 3: Probing BERT for Constituency Information

Instructor: Freda Shi (fhs@uwaterloo.ca)
TA: Sheng Yao
Released: Fri, Oct 30, 2026
Due: **Fri, Nov 13, 2026, 11:59 PM ET (Waterloo Time)**
Weight: 9% of final grade

You may use AI tools, but you are responsible for all submitted work and must be able to explain the code and results. Using standard PyTorch and Hugging Face components is allowed.

Warning: if we find evidence that you used AI tools for anything beyond grammatical checking without declaring that use in `writeup.pdf`, you will receive 0 on this assignment.

---

1. Overview

In this assignment, you will predict the constituent label of a given span using a small classifier trained on frozen BERT representations. For example, a span may be labeled `NP`, `VP`, or `PP`. Each example provides a sentence and the gold span boundaries.

Extract a representation for every span at every BERT layer, train a separate two-layer MLP for each layer, and compare their development-set accuracy. Repeat each layer's experiment with **five seeds** and plot the mean accuracy with its **standard error**. Use the same probe architecture and training settings across layers for a controlled comparison.

Total points: 100.

2. Data, Model, and Environment

2.1 Data format

Download the constituent-label data from `compling/cs489-f26-a3` on Hugging Face. Training split contains 39,832 sentences and 701,738 labeled spans; dev contains 1,700 sentences and 29,034 labeled spans. The TA holds the test split for final evaluation.

Each UTF-8 JSONL line is one sentence with three fields:

- `id`: a unique record identifier.
- `tokens`: the ordered list of word tokens, including punctuation.
- `constituents`: a list of spans, each with `start`, `end`, and `label`.

Here is a hand-written example:

```
{"id":"toy-0001","tokens":["The","cat","sleeps","."],
 "constituents":[{"start":0,"end":2,"label":"NP"},
                 {"start":0,"end":4,"label":"S"},
                 {"start":2,"end":3,"label":"VP"}]}
```

The display is wrapped for readability; the actual JSONL record occupies one line. `start` and `end` are zero-based word-token indices with an exclusive end, so the span is `tokens[start:end]`. Each span has one constituent label, given as a string in `label`.

`labels.json` is a JSON list of 26 label strings; a label's zero-based index in this list is its class ID. Use this label order unchanged for every split, probe, and artifact. Use the complete train/dev files as supplied and count every listed span during training and evaluation.

2.2 Frozen BERT

Use `google-bert/bert-base-cased` with the exact model and tokenizer revision in the provided configuration. Preserve the supplied tokens and their original casing. BERT-base has 12 Transformer blocks and hidden size 768. All supplied sentences fit its 512-WordPiece input limit, including special tokens. Set `output_hidden_states` to `True` to request all 13 representations:

- Layer 0 is the embedding module output, including position and token-type embeddings and LayerNorm.
- Layers 1–12 are the outputs of the corresponding Transformer blocks.

Use **all 13 layers**, with an independent probe for each layer and seed. Call `model.eval()`, disable gradients on BERT parameters, and extract features without autograd. Optimize only the probe parameters.

2.3 Environment and provided files

Starter code is available at `CS489-F26/a3/starter`. Use the starter's `uv`-managed Python environment and retain `pyproject.toml` and `uv.lock` in your submission. Copy all starter files into your project root. The starter README distinguishes fixed course helpers from the three implementation files: `features.py`, `run_part1.py`, and `plot_results.py`. Fixed helpers specify the data alignment, MLP architecture, artifact format, and evaluation routines. The TA restores fixed modules to the course versions for evaluation.

A single GPU is useful for frozen BERT extraction; the intended development environment is a Colab T4. For reference, extraction and all 65 training runs took about 74 minutes on an RTX 4090 Laptop GPU, with a measured process maximum RSS of 19.2 GiB (including resident memory-mapped pages). Cache pooled features once and reuse them for all probe training runs. The required float32 train/dev span arrays alone occupy about 27.2 GiB on disk, so check free storage before extraction.

3. Part 1: Layer-wise Constituent Probing *(100 pts)*

3.1 Span representations *(25 pts)*

Encode the **whole sentence** so that span vectors contain sentence context. Use the pinned fast tokenizer with `is_split_into_words=True` and its `word_ids()` mapping. Add the usual `[CLS]` and `[SEP]`, pad on the right, disable truncation, and pass the correct `attention_mask` to BERT. Exclude padding positions from attention and pooling.

Map a word span $[l, r)$ to WordPiece positions $[l_{\text{wp}}, r_{\text{wp}})$ in the actual BERT input, accounting for the initial `[CLS]`. Include every WordPiece of every word in the span. For hidden-state matrix $H^{(k)}$ at layer k, compute

$$z_{l:r}^{(k)} = \frac{1}{r_{\text{wp}} - l_{\text{wp}}} \sum_{t=l_{\text{wp}}}^{r_{\text{wp}}-1} H_t^{(k)} \in \mathbb{R}^{768}.$$

This is **mean pooling over WordPieces**: slice `hidden_states[k][l_wp:r_wp]` for the sentence, then average over the token dimension. Exclude special tokens and padding; ordinary punctuation within the constituent remains part of the span. Use this 768-dimensional vector as the sole input

to the MLP.

Save pooled float32 features locally, keeping splits and layers separate, and reuse them across probe training runs. Regenerate the cached features whenever the input data or feature-extraction procedure changes. Process sentences in batches and load feature minibatches from disk for training.

3.2 The probe and five-seed training *(25 pts)*

The fixed probe is

$$\mathbb{R}^{768} \xrightarrow{\text{Linear}} \mathbb{R}^{1024} \xrightarrow{\text{ReLU}} \mathbb{R}^{1024} \xrightarrow{\text{Linear}} \mathbb{R}^{C},$$

where C is the number of course labels. **The probe has two affine layers with biases and one 1,024-unit ReLU hidden layer.** Pass the output logits directly to cross-entropy.

Train a freshly initialized probe and optimizer for each layer and each seed in `[0, 1, 2, 3, 4]`, giving **65 independent runs**. Use the same data, architecture, optimizer settings, and training budget for all runs. The supplied configuration fixes Adam, learning rate 10^{-3}, batch size 256, 10 epochs, and weight decay 0.

Reset the relevant Python, NumPy, PyTorch, and training-shuffle random states for each run. Seeds control probe initialization and training order; keep the data splits and cached BERT features fixed. Record the device and software versions; small floating-point differences across hardware are normal.

Update probe parameters using the train split. After every epoch, evaluate on the *entire* dev split in evaluation mode without gradients. Save the checkpoint with the highest dev accuracy; ties keep the earlier epoch. At the end of each run, reload that checkpoint and compute its final dev accuracy. Save local epoch-level train loss, dev loss, dev accuracy, and sample counts in `results/training.csv`. Online experiment tracking is optional.

3.3 Accuracy, standard error, and the layer curve *(20 pts)*

For layer k and seed s, compute span-level accuracy by accumulating integer counts over the full split:

$$a_{k,s} = \frac{\text{number of correctly classified spans}}{\text{number of evaluated spans}}.$$

For the five seeds, report

$$\bar{a}_k = \frac{1}{5}\sum_{s=1}^{5} a_{k,s}, \qquad \mathrm{SE}_k = \frac{1}{\sqrt{5}}\sqrt{\frac{1}{4}\sum_{s=1}^{5}(a_{k,s} - \bar{a}_k)^2}.$$

This summarizes variation across the five independently trained probes on the fixed dev set.

Create one plot with layer index 0–12 on the horizontal axis and mean dev accuracy on the vertical axis, with error bars or a band of **mean $\pm$ one SE**. Identify layer 0 as the embedding output. If displaying accuracy as a percentage, multiply both the mean and SE by 100; SE is then in percentage points.

Save all 65 best-checkpoint results in `results/runs.csv`. Save the 13 layer summaries in `results/layers.csv`, using the starter's field names. Store CSV accuracies as numeric fractions in $[0, 1]$. Before aggregation, verify that all 65 layer/seed pairs appear exactly once and share matching data and protocol fingerprints recorded by the evaluator. Generate `results/layer_accuracy.pdf` from these CSV results.

3.4 Short analysis *(20 pts)*

Submit `writeup.pdf`, at most **two pages including figures and tables**. Include the layer curve and a compact table of mean accuracy and SE, then discuss:

1. Which layer has the highest mean dev accuracy, and how do early, middle, and late layers compare? Support the comparison with your measured values.
2. How large are the differences between layers relative to the seed-to-seed variability shown by the error bars?
3. What does the probe reveal about constituent-label information recoverable from each layer, and what are the limits of this evidence about BERT's own use of that information?
4. How might gold boundaries, annotation quality, class imbalance, mean pooling, and probe capacity limit the conclusion?

State the training configuration, device, measured runtime, and AI-use declaration. Label the curve as *development-set performance*, since dev accuracy is used for checkpoint selection.

3.5 Reproducible implementation *(10 pts)*

Complete `run_part1.py` as the entry point for all 13 layers and five seeds. Put runtime settings and paths in YAML; protocol settings must match the course configuration. Interpret paths relative to the YAML file, and export a usable `artifacts/part1/config.yaml` with portable relative paths. Include your locked environment, supporting modules, and exact setup, training, evaluation, and plotting commands in `README.md`.

Deliverables: The 65 probe checkpoints and their configuration, label list, and manifest under `artifacts/part1/`; executable `run_part1.py`, `features.py`, `plot_results.py` and supporting files; the CSV files and layer figure under `results/`; and `writeup.pdf`, following Section 4. The TA loads your artifact, computes held-out accuracy, and grades the implementation, statistics, and analysis.

4. Submission and Evaluation

Submit a single ZIP on LEARN with this structure. Include all fixed starter modules and any supporting modules your implementation needs.

```
a3/
  pyproject.toml
  uv.lock
  README.md
  run_part1.py
  features.py
  plot_results.py
  <all fixed starter and supporting Python modules>
  artifacts/
    part1/
      config.yaml
      labels.json
      manifest.json
      checkpoints/
        layer_00/seed_0.pt ... seed_4.pt
        ...
        layer_12/seed_0.pt ... seed_4.pt
  results/
    training.csv
    runs.csv
    layers.csv
    layer_accuracy.pdf
  writeup.pdf
```

Checkpoints contain only CPU tensor `state_dict` weights and must load strictly into the provided `probe_model.py` architecture. Use the supplied artifact helpers and keep the course label order unchanged. The manifest identifies each checkpoint's layer, seed, best epoch, dev counts, and checksum, along with the data and experiment versions. The collection of 65 probes is approximately 200 MiB in float32; the exact size depends on the number of labels.

Start with `config.example.yaml`; the training entry point exports the resolved configuration under `artifacts/part1/`. The following commands must work from the project root, with the course data available under `data/`:

```
uv sync --locked
uv run python run_part1.py --config artifacts/part1/config.yaml \
    --output-root rerun-01
uv run python evaluator.py --artifacts artifacts/part1 \
    --data data/dev.jsonl --output results/runs.csv
uv run python plot_results.py --input results/runs.csv \
    --table results/layers.csv --output results/layer_accuracy.pdf
```

The evaluator reloads checkpoints and recomputes development-set accuracy using your feature extractor. The plotting command validates and summarizes those results. The optional `--output-root` selects a fresh directory for the reproduced artifacts and results, while reusing the configured data and cache. The training entry point must reject nonempty artifact/results directories.

The TA evaluates all 65 submitted probes on the private test split with the same accuracy and SE definitions. Use the dev split for checkpoint selection and layer comparisons, and report dev results in your submission. Grading uses the submitted checkpoints; the TA may rerun the code to check reproducibility. Reproducibility checks allow ordinary numerical variation across hardware; fabricated results and hand-edited artifacts are academic-integrity violations.

Package only the code, environment specifications, artifacts, results, and write-up shown above. Keep data, downloaded BERT weights, feature caches, credentials, and virtual environments outside the ZIP. The submission must run using the included code and environment together with

the course data and specified BERT checkpoint.

5. Policies

Collaboration and academic integrity

This is individual work. You may discuss high-level ideas, but must not copy another student's code, results, or report. Cite external code and declare AI assistance beyond grammatical checking in `writeup.pdf`. You must be able to explain the submitted implementation and claims. Use only the supplied train/dev data for probe training and development, and only the specified pretrained BERT checkpoint for features. See University of Waterloo Policy 71.

Late policy

The deadline is **Fri, Nov 13, 2026 at 11:59 PM Eastern Time**. The universal **72-hour grace period** accepts submissions without penalty until Mon, Nov 16, 2026 at 11:59 PM Eastern Time. No submissions are accepted after the grace period ends. Extensions beyond the grace period are considered only for medical emergencies: a valid Verification of Illness Form (VIF) must be submitted before the original deadline and cover both that deadline and the entire grace period.