A Deep Dive into Syntactic Parsing for NLP Interviews

This guide provides a comprehensive overview of syntactic parsing, tailored for candidates preparing for Natural Language Processing and Data Science interviews. It covers the foundational theory of Constituency and Dependency parsing, traces the evolution from classical algorithms like CKY to modern neural architectures, and details practical implementations in Python using libraries such as NLTK, spaCy, and Stanza. The document culminates in a discussion of state-of-the-art transformer-based models from the Hugging Face ecosystem and presents a curated set of theoretical and coding interview questions with detailed solutions to solidify understanding and test practical skills.

Knowledge Section

1. The Fundamentals of Syntactic Parsing

Syntactic parsing is the process of analyzing a string of symbols—typically a sentence—to determine its grammatical structure according to a formal grammar. It is a cornerstone of Natural Language Understanding (NLU), transforming a flat sequence of words into a hierarchical structure that reveals who did what to whom.

1.1. The Two Core Paradigms: Constituency vs. Dependency Parsing

The field is dominated by two distinct philosophies for representing sentence structure.

Constituency Parsing (Phrase-Structure Parsing)

- **Core Idea:** Sentences are composed of nested phrases or "constituents" (e.g., Noun Phrase, Verb Phrase). The goal is to identify these phrases and their hierarchical relationships.
- **Underlying Grammar:** Based on Phrase Structure Grammars, most notably the **Context-Free Grammar (CFG)**. A CFG is a set of production rules specifying how constituents are formed, e.g., `S -> NP VP`.
- **Output:** A constituency parse tree, an ordered, rooted tree where leaf nodes are

words and internal nodes are phrasal categories.
- **Example:** "The cat sat on the mat."
  `(S (NP (DT The) (NN cat)) (VP (VBD sat) (PP (IN on) (NP (DT the) (NN mat)))))`

Dependency Parsing

- **Core Idea:** Focuses on the grammatical relationships between individual words. The structure is a set of directed, labeled links (dependencies) from a "head" word to a "dependent" word.
- **Underlying Grammar:** Based on Dependency Grammars, which view the verb as the central element governing its arguments. It abstracts away from word order to directly encode grammatical functions.
- **Output:** A dependency graph (typically a tree), where nodes are words and labeled edges represent dependency relations (e.g., `nsubj` for nominal subject, `dobj` for direct object).
- **Example:** "The cat sat on the mat."
  - `cat` --(`nsubj`)--> `sat`
  - `The` --(`det`)--> `cat`
  - `mat` --(`nmod`)--> `sat` (via `on`)
  - `on` --(`case`)--> `mat`
  - `the` --(`det`)--> `mat`

Feature	Constituency Parsing	Dependency Parsing
Core Unit	Phrases/Constituents (e.g., NP, VP)	Words and their relationships
Underlying Grammar	Phrase Structure Grammar (e.g., CFG)	Dependency Grammar
Output	Hierarchical tree with phrase labels	Directed graph (tree) with labeled arcs
Key Strength	Explicitly models phrasal structure	Directly models predicate-argument structure
Common Use Cases	Phrase-level sentiment analysis, Grammar checking	Information Extraction (S-V-O), Question Answering

Feature	Constituency Parsing	Dependency Parsing
Word Order	Better for fixed-order languages	More robust for free-order languages

2. Constituency Parsing: From Grammars to Neural Models

2.1. Core Concepts and Algorithms

Context-Free Grammar (CFG)

A CFG is defined by a 4-tuple $G = (N, \Sigma, R, S)$, where:

- N is a set of non-terminal symbols (e.g., `S`, `NP`, `VP`).
- Σ is a set of terminal symbols (the vocabulary of words).
- R is a set of production rules of the form $A \to \beta$, where $A \in N$ and $\beta \in (N \cup \Sigma)^*$.
- $S \in N$ is the start symbol.

Probabilistic Context-Free Grammar (PCFG)

Natural language is inherently ambiguous (e.g., "I saw a man with a telescope"). PCFGs address this by assigning a probability to each rule, $P(A \to \beta)$. The probability of a parse tree T is the product of the probabilities of all rules used to generate it:

$$P(T) = \prod_{(A \to \beta) \in T} P(A \to \beta)$$

Given a sentence, the parser's goal is to find the most likely parse tree: $\arg\max_T P(T)$.

Chomsky Normal Form (CNF)

A restricted but powerful form of CFG required by some algorithms like CKY. All production rules must take one of two forms:

1. $A \to BC$ (A non-terminal yields two non-terminals)
2. $A \to w$ (A non-terminal yields a single terminal/word)
   Any CFG can be converted into an equivalent CNF.

The CKY Algorithm (Cocke-Kasami-Younger)
A classic dynamic programming algorithm for parsing with a CFG in CNF. It fills a triangular chart (or table) `T` of size $n \times n$ for a sentence of length n. An entry `T[i, j]` stores the set of non-terminals that can generate the substring from word `i` to word `j`.

- **Initialization:** For each word w_i, fill `T[i, i]` with all non-terminals A such that $A \rightarrow w_i$ is a rule.
- **Inductive Step:** For increasing substring lengths `len` from 2 to `n`:
    - For each starting position `i` from 1 to `n-len+1`:
        - Calculate the end position `j = i+len-1`.
        - For each possible split point `k` from `i` to `j-1`:
            - If there is a rule $A \rightarrow BC$ and $B \in T[i, k]$ and $C \in T[k+1, j]$, then add A to `T[i, j]`.
- **Result:** The sentence is grammatical if the start symbol S is in `T[1, n]`.

2.2. Modern Neural Constituency Parsers

While classical methods are foundational, modern parsers are neural network-based, trained on large annotated "treebanks" like the Penn Treebank (PTB).

- **Stanza (Stanford NLP Group):** Offers high-accuracy, pre-trained parsers. It uses a sophisticated model, often incorporating transformers like BERT, to achieve state-of-the-art results.
- **spaCy + benepar:** spaCy is a production-focused library optimized for dependency parsing. The `benepar` extension seamlessly integrates a state-of-the-art, self-attention-based constituency parser into the spaCy pipeline, making it easy to access SOTA performance within a production-ready ecosystem.

3. Dependency Parsing: Modeling Word Relationships

3.1. Core Concepts

- **Dependency Relations:** The labeled arcs, such as `nsubj` (nominal subject), `dobj` (direct object), `amod` (adjectival modifier).
- **Universal Dependencies (UD):** A cross-linguistic project to create a standardized set of dependency relations and POS tags. This enables the creation of

multilingual models and is the standard for most modern parsers.

- **Projectivity:** A dependency tree is projective if its arcs can be drawn above the sentence without any of them crossing. While most English sentences are projective, non-projective structures exist (e.g., "The book which I bought was expensive.") and pose a challenge for simpler parsing algorithms.

3.2. Foundational Parsing Algorithms

Transition-Based Parsing (Shift-Reduce)

- **Analogy:** A state machine that processes a sentence incrementally.
- **Components:** A stack of partially processed words, a buffer of upcoming words, and a set of actions.
- **Actions (Arc-Standard System):**
    i. **Shift:** Move a word from the buffer to the stack.
    ii. **Left-Arc:** Create a `head <- dependent` arc between the top two words on the stack and pop the dependent.
    iii. **Right-Arc:** Create a `head -> dependent` arc between the top two words on the stack and pop the dependent.
- **Mechanism:** At each step, a classifier (often a neural network) predicts the next best action.
- **Trade-offs:** Very fast (often linear time, $O(n)$), but decisions are greedy and local, which can lead to error propagation.

Graph-Based Parsing

- **Analogy:** Finding the best spanning tree in a dense graph.
- **Mechanism:**
    i. Create a fully connected directed graph where nodes are words.
    ii. Use a model to score every possible directed arc (h, d) from a potential head h to a dependent d.
    iii. The problem becomes finding the **Maximum Spanning Tree (MST)** in this graph that is rooted at a special `ROOT` node. The Chu-Liu-Edmonds algorithm is used for this.
- **Trade-offs:** Finds a globally optimal tree, leading to higher accuracy, especially for long-distance dependencies. However, it is computationally more expensive (e.g., $O(n^2)$ or $O(n^3)$).

3.3. Modern Implementations

- **spaCy:** The industry standard for production NLP. It features a highly optimized and fast transition-based dependency parser as a core component. Its "batteries-included" approach makes it incredibly easy to use.
- **Stanza:** Provides a very high-accuracy graph-based parser, often topping academic leaderboards. It is an excellent choice when precision is the absolute priority.

4. The Transformer Era: State-of-the-Art Parsing

4.1. Implicit Syntax in Transformers and Probing

Research has shown that large pre-trained language models (PLMs) like BERT and Llama learn hierarchical syntactic structures implicitly during their pre-training (e.g., on Masked Language Modeling). This knowledge can be extracted via **probing**. A probe is a simple classifier (e.g., a linear layer) trained to predict a syntactic property (like a dependency relation) using the transformer's internal word embeddings. The high performance of these probes proves that the syntactic information is already encoded in the embeddings.

4.2. Key Architectures: Deep Biaffine Attention

The Deep Biaffine Attention mechanism (Dozat & Manning, 2017) is a cornerstone of modern graph-based dependency parsing. After encoding the sentence (e.g., with a BiLSTM or Transformer), the model generates specialized representations for each word as a potential head and as a potential dependent.

The score for a dependency arc from word j (dependent) to word i (head) is calculated as:

$$s_{i,j} = (\mathbf{h}_i^{(\text{head})})^T \mathbf{U} \mathbf{h}_j^{(\text{dep})} + \mathbf{w}^T [\mathbf{h}_i^{(\text{head})} \oplus \mathbf{h}_j^{(\text{dep})}] + b$$

Where:

- $\mathbf{h}_i^{(\text{head})}$ and $\mathbf{h}_j^{(\text{dep})}$ are MLP-transformed representations.
- $\mathbf{U}$ is a learnable tensor capturing the bilinear (interactive) term.
- $\mathbf{w}$ is a learnable weight vector for the linear term.
- $\oplus$ denotes concatenation.

This mechanism is highly effective and is used in SOTA parsers like those in Stanza and DiaParser.

4.3. Leveraging the Hugging Face Ecosystem

The Hugging Face Hub hosts many parsing models. However, parsing is not a standard task in the `transformers.pipeline` API because it requires complex, non-generic decoding logic (like transition-based systems or MST algorithms). Therefore, using these models typically involves:

1. **Integrated Libraries (Best Practice):** Using libraries like `Stanza`, `spaCy+benepar`, or `diaparser` that handle loading a transformer from the Hub and wrapping it in their own high-performance parsing decoders.
2. **Prompt-Based LLMs (Emerging):** Fine-tuning large generative models (like Llama 3) to produce a textual representation of a parse tree when given a specific prompt. This is powerful but requires careful prompt engineering and custom output parsing logic.

Interview Questions

Theoretical Questions

Question 1: Explain the core differences between Constituency and Dependency Parsing. When would you choose one over the other?

Answer:
The core difference lies in what they model.

- **Constituency Parsing** models **phrase structure**. It groups words into nested constituents like Noun Phrases (NP) and Verb Phrases (VP). Its output is a hierarchical tree that explicitly shows how the sentence is built from these blocks.
- **Dependency Parsing** models **word-to-word relationships**. It identifies the grammatical functions that words serve relative to each other (e.g., subject, object, modifier). Its output is a flatter graph that directly exposes the predicate-argument structure.

When to choose which:

- **Choose Constituency Parsing when:**
    - The task requires analyzing or extracting complete phrases. For example, in phrase-level sentiment analysis, you need to know if the sentiment of "disappointing" applies to "the movie" or "the ending".
    - You need to check for grammatical well-formedness based on a formal grammar.
    - The structure of constituents themselves is important for downstream tasks.
- **Choose Dependency Parsing when:**
    - The task requires understanding functional relationships. This is extremely common in modern NLP.
    - **Information Extraction:** Extracting subject-verb-object (SVO) triples is trivial with a dependency parse.
    - **Question Answering:** Understanding the relationship between words in a question is key to finding the answer.
    - **Machine Translation:** Preserving the core grammatical roles is crucial when translating.
    - **Speed and Efficiency:** Dependency parsers, especially transition-based ones, are often faster and more suitable for production systems.

Question 2: What is structural ambiguity in parsing? How does a Probabilistic Context-Free Grammar (PCFG) help resolve it?

Answer:
Structural ambiguity occurs when a single sentence can have more than one valid syntactic parse tree according to a given grammar. A classic example is the sentence: "I saw a man with a telescope."

This sentence has two possible interpretations, leading to two different constituency parse trees:

1. **Attachment to NP:** The prepositional phrase "with a telescope" modifies "man". The telescope is the instrument the man was holding.
   `(S (NP I) (VP (V saw) (NP (DT a) (NN man) (PP (IN with) (NP (DT a) (NN telescope`

2. **Attachment to VP:** The prepositional phrase "with a telescope" modifies "saw". The telescope is the instrument used for seeing.
   `(S (NP I) (VP (VP (V saw) (NP (DT a) (NN man))) (PP (IN with) (NP (DT a) (NN tel`

A standard CFG would consider both parses equally valid. A **PCFG** resolves this ambiguity by assigning a probability to each production rule. The probability of a complete parse tree is the product of the probabilities of all the rules used to generate it.

For instance, the grammar might have learned from a corpus that the rule `VP -> V NP PP` (attaching the PP to the verb phrase) is more probable than `NP -> NP PP` (attaching it to the noun phrase).

Let $P(T_1)$ be the probability of the first tree and $P(T_2)$ be the probability of the second. The PCFG parser will compute both and select the tree with the higher probability:

$$T_{best} = \arg\max_{T \in \{T_1, T_2\}} P(T)$$

By learning these rule probabilities from a large treebank, the parser can make a statistically informed decision and choose the most likely interpretation.

Question 3: Explain the trade-offs between transition-based and graph-based dependency parsing.

Answer:
This is a classic trade-off between speed and global optimality.

Feature	Transition-Based Parsing	Graph-Based Parsing
Algorithm	Greedy, local decisions (Shift, Arc) made sequentially.	Global optimization, scores all possible arcs and finds the Maximum Spanning Tree (MST).
Speed	**Very Fast.** Typically linear time, $O(n)$, where n is sentence length.	**Slower.** Typically polynomial time, $O(n^2)$ or $O(n^3)$.
Accuracy	Can be slightly less accurate. A single wrong decision early	Generally more accurate. It considers the entire sentence structure globally,

Feature	Transition-Based Parsing	Graph-Based Parsing
	on can lead to a cascade of errors (**error propagation**).	making it better at capturing complex or long-distance dependencies.
Projectivity	Basic versions can only produce projective trees. More complex algorithms are needed for non-projectivity.	Can naturally handle non-projective dependencies by simply allowing crossing arcs in the MST algorithm.
Suitability	Ideal for **production systems** where throughput and low latency are critical (e.g., spaCy's default parser).	Ideal for **research** or applications where achieving the highest possible accuracy is the main goal (e.g., Stanza's parser).

Question 4: Prove that the objective function for Logistic Regression is convex, ensuring that Gradient Descent will find a global minimum.

Answer:
Let's define the objective function for logistic regression, which is the negative log-likelihood (or cross-entropy loss). For a single training example (x, y) where $y \in \{0, 1\}$, the loss is:

$$L(w) = -[y \log(\sigma(w^T x)) + (1 - y) \log(1 - \sigma(w^T x))]$$

where $\sigma(z) = \frac{1}{1+e^{-z}}$ is the sigmoid function. Let $z = w^T x$.

To prove convexity, we need to show that the Hessian matrix of the loss function with respect to the weights w is positive semidefinite. The Hessian is the matrix of second partial derivatives.

First, let's find the first derivative (gradient). We use the fact that $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

$$\frac{\partial L}{\partial w_j} = -\left[y \frac{1}{\sigma(z)} \sigma'(z) x_j - (1 - y) \frac{1}{1 - \sigma(z)} \sigma'(z) x_j\right]$$

$$= -\left[y(1 - \sigma(z)) x_j - (1 - y)\sigma(z) x_j\right]$$

$$= -[y - y\sigma(z) - \sigma(z) + y\sigma(z)]x_j = -[y - \sigma(z)]x_j = (\sigma(w^T x) - y)x_j$$

This is the familiar gradient update rule.

Now, let's compute the second partial derivative to form the Hessian, H:

$$H_{jk} = \frac{\partial^2 L}{\partial w_j \partial w_k} = \frac{\partial}{\partial w_k}[(\sigma(w^T x) - y)x_j]$$

$$= \frac{\partial}{\partial w_k}[\sigma(w^T x)x_j] = x_j \frac{\partial}{\partial w_k}\sigma(w^T x)$$

$$= x_j[\sigma'(w^T x)\frac{\partial}{\partial w_k}(w^T x)] = x_j[\sigma(w^T x)(1 - \sigma(w^T x))x_k]$$

$$H_{jk} = \sigma(w^T x)(1 - \sigma(w^T x))x_j x_k$$

The full Hessian matrix for a single data point is:

$$H = \sigma(w^T x)(1 - \sigma(w^T x))xx^T$$

For the entire dataset of m points, the Hessian is the sum over all points:

$$H = \sum_{i=1}^{m} \sigma(w^T x_i)(1 - \sigma(w^T x_i))x_i x_i^T$$

To prove that H is positive semidefinite, we must show that for any non-zero vector v, the quadratic form $v^T H v \geq 0$.

$$v^T H v = v^T \left(\sum_{i=1}^{m} \sigma(w^T x_i)(1 - \sigma(w^T x_i)) x_i x_i^T \right) v$$

$$= \sum_{i=1}^{m} \sigma(w^T x_i)(1 - \sigma(w^T x_i)) v^T (x_i x_i^T) v$$

$$= \sum_{i=1}^{m} \sigma(w^T x_i)(1 - \sigma(w^T x_i))(v^T x_i)(x_i^T v)$$

$$= \sum_{i=1}^{m} \underbrace{\sigma(w^T x_i)(1 - \sigma(w^T x_i))}_{\geq 0} \underbrace{(v^T x_i)^2}_{\geq 0}$$

The term $\sigma(w^T x_i)$ is a probability, so its value is in $[0, 1]$. Therefore, $\sigma(z)(1 - \sigma(z))$ is always non-negative. The term $(v^T x_i)^2$ is a squared real number, which is also always non-negative.

Since $v^T H v$ is a sum of non-negative terms, $v^T H v \geq 0$. This proves that the Hessian is positive semidefinite. Therefore, the loss function for logistic regression is convex, which guarantees that gradient descent can find the single global minimum.

Practical & Coding Questions

Question 5: Using spaCy, parse the sentence "Apple is looking at buying a U.K. startup for $1 billion" and extract the subject, the root verb, and the direct object.

Answer:

```python
import spacy

# Load a pre-trained English model
nlp = spacy.load("en_core_web_sm")

# Process the text
text = "Apple is looking at buying a U.K. startup for $1 billion"
doc = nlp(text)

# Initialize variables to store the results
subject = None
root_verb = None
direct_object = None

# Find the root of the sentence (usually the main verb)
for token in doc:
    if token.dep_ == "ROOT":
        root_verb = token
        break

# If a root is found, find its subject and direct object
if root_verb:
    for child in root_verb.children:
        if child.dep_ == "nsubj":
            subject = child
        # Direct objects are often attached to verbs that are children of the root
        # In this case, 'buying' is the action, and its object is 'startup'
        if child.dep_ in ("xcomp", "ccomp"): # Clausal complement
            for grand_child in child.children:
                if grand_child.dep_ == "dobj":
                    direct_object = grand_child

# A more robust way for this specific sentence structure
# The direct object of 'buying' is 'startup'
for token in doc:
    if token.lemma_ == "buy" and token.dep_ == "xcomp":
        for child in token.children:
            if child.dep_ == "dobj":
                direct_object = child
```

```
# Let's find the subject of the main verb 'looking'
for token in doc:
    if token.text == 'looking':
        for child in token.children:
            if child.dep_ == 'nsubj':
                subject = child

print(f"Original Sentence: '{doc.text}'")
print("-" * 30)
print(f"Subject: {subject.text if subject else 'Not Found'}")
print(f"Root Verb: {root_verb.text if root_verb else 'Not Found'}")
print(f"Direct Object: {direct_object.text if direct_object else 'Not Found'}")

# Visualize the parse tree to confirm
from spacy import displacy
print("\nDependency Parse Tree:")
# To run in a Jupyter notebook/Colab, this line is sufficient.
# To serve locally, use displacy.serve(doc, style="dep")
displacy.render(doc, style="dep", jupyter=True, options={'distance': 110})
```

Output Visualization will be rendered in a notebook.

Question 6: Using `spaCy` and the `benepar` extension, parse a sentence and extract all Noun Phrases (NPs) and Verb Phrases (VPs).

Answer:

```
import spacy
import benepar

# Note: You may need to run these first:
# pip install benepar
# python -m spacy download en_core_web_md
# benepar.download('benepar_en3')

# Load a spaCy model
nlp = spacy.load('en_core_web_md')

# Add the benepar component to the pipeline
# The syntax differs slightly for spaCy v2 vs v3+
try:
    nlp.add_pipe("benepar", config={"model": "benepar_en3"})
except ValueError:
    print("Benepar component may already be in the pipeline.")

text = "The quick brown fox jumps over the lazy dog."
doc = nlp(text)

# The constituency parse is available on each sentence span
sentence = list(doc.sents)[0]

print(f"Sentence: '{sentence.text}'")
print(f"Full Parse Tree:\n{sentence._.parse_string}\n")

noun_phrases = []
verb_phrases = []

# Iterate through all constituents in the sentence
for const in sentence._.constituents:
    # The ._.labels attribute is a tuple of labels
    if const._.labels == ('NP',):
        noun_phrases.append(const.text)
    elif const._.labels == ('VP',):
        verb_phrases.append(const.text)

print("--- Extracted Noun Phrases (NPs) ---")
```

```
for i, np in enumerate(noun_phrases):
    print(f"{i+1}. {np}")

print("\n--- Extracted Verb Phrases (VPs) ---")
for i, vp in enumerate(verb_phrases):
    print(f"{i+1}. {vp}")
```

Question 7: Conceptually describe how you would "probe" a pre-trained Transformer like BERT for syntactic knowledge. Then, write a PyTorch code snippet to implement the probing classifier.

Answer:

Conceptual Description:
Probing is a diagnostic method to determine what kind of linguistic information is encoded in a model's internal representations (embeddings). To probe BERT for syntactic knowledge (e.g., dependency relations), we would follow these steps:

1. **Freeze BERT:** We take a pre-trained BERT model and freeze all its weights. We will not be fine-tuning it; we are only using it as a feature extractor.
2. **Prepare a Labeled Dataset:** We need a dataset where each pair of words in a sentence is labeled with their syntactic relationship (e.g., `nsubj`, `dobj`, or `None`). This data comes from a treebank.
3. **Extract Embeddings:** For each sentence in our dataset, we feed it through the frozen BERT model to get the final hidden state (contextual embedding) for every token.
4. **Train a Simple Classifier:** We design a very simple "probe" classifier, typically just a linear layer followed by a softmax. This classifier takes the embeddings of two words (a potential head and dependent) as input and tries to predict their dependency relation.
5. **Evaluate:** We train this probe classifier on our labeled dataset. If the simple probe achieves high accuracy, it implies that the necessary information for the task was already present in the BERT embeddings. If it performs poorly, it suggests BERT did not encode that specific syntactic information well. The simplicity of the probe is key—if a complex probe works, it might just be the probe itself learning the task, not BERT.

PyTorch Implementation Snippet:

```python
import torch
import torch.nn as nn
from transformers import BertModel, BertTokenizer

# --- 1. Setup Models (BERT is frozen) ---
tokenizer = BertTokenizer.from_pretrained('bert-base-uncased')
bert_model = BertModel.from_pretrained('bert-base-uncased')

# Freeze all parameters in the BERT model
for param in bert_model.parameters():
    param.requires_grad = False

bert_model.eval() # Set to evaluation mode

# --- 2. Define the Probing Classifier ---
# This probe will take two word embeddings (head and dependent) and classify their relatio
class DependencyProbe(nn.Module):
    def __init__(self, hidden_size, num_dep_relations):
        """
        Args:
            hidden_size (int): The size of BERT's hidden embeddings (e.g., 768).
            num_dep_relations (int): The number of possible dependency relations (our outp
        """
        super(DependencyProbe, self).__init__()
        # We concatenate the head and dependent embeddings
        self.classifier = nn.Linear(hidden_size * 2, num_dep_relations)

    def forward(self, head_embedding, dependent_embedding):
        """
        Args:
            head_embedding (Tensor): The embedding of the potential head word.
            dependent_embedding (Tensor): The embedding of the potential dependent word.
        """
        # Concatenate the embeddings
        combined_embedding = torch.cat((head_embedding, dependent_embedding), dim=-1)
        # Pass through the linear layer to get logits
        logits = self.classifier(combined_embedding)
        return logits
```

```python
# --- 3. Example Usage in a Training Loop (Conceptual) ---
# Assume we have a dataset with sentences, head/dependent indices, and relation labels
sentence = "The cat sat on the mat"
head_idx = 2  # "sat"
dep_idx = 1   # "cat"
relation_label = 5 # Let's say 5 corresponds to 'nsubj' in our label mapping

# Get BERT embeddings
inputs = tokenizer(sentence, return_tensors="pt")
with torch.no_grad():
    outputs = bert_model(**inputs)
    # Get the last hidden state
    last_hidden_states = outputs.last_hidden_state.squeeze(0) # Remove batch dim

# Extract embeddings for the specific head and dependent words
# Note: Tokenizer might create subwords, so indices need careful mapping.
# For simplicity, we use the given indices directly.
head_emb = last_hidden_states[head_idx]
dep_emb = last_hidden_states[dep_idx]

# Instantiate the probe
# Example: 37 Universal Dependency relations + 1 for 'None'
num_relations = 38
probe = DependencyProbe(hidden_size=bert_model.config.hidden_size, num_dep_relations=num_
optimizer = torch.optim.Adam(probe.parameters(), lr=1e-3)
loss_fn = nn.CrossEntropyLoss()

# --- Training Step ---
probe.train()
optimizer.zero_grad()

# Get logits from the probe
logits = probe(head_emb.unsqueeze(0), dep_emb.unsqueeze(0)) # Add batch dim

# Calculate loss and perform backpropagation
# Note: target label must be a tensor
target = torch.tensor([relation_label])
loss = loss_fn(logits, target)
loss.backward()
```

```
optimizer.step()

print(f"Training Loss: {loss.item()}")
print("Probe classifier updated. BERT remains frozen.")
```