An Expert's Guide to Part-of-Speech Tagging: From HMMs to Transformers

This guide provides a comprehensive overview of Part-of-Speech (POS) tagging, structured as an interview preparation resource for data scientists and NLP engineers. It begins with the fundamental concepts of POS tagging, explores the evolution of modeling techniques from classical statistical methods like Hidden Markov Models (HMMs) and Conditional Random Fields (CRFs), delves into the deep learning era with Bidirectional LSTMs (BiLSTM) and the BiLSTM-CRF architecture, and culminates with the current state-of-the-art using Transformer-based models like BERT. Each section is enriched with theoretical explanations, mathematical formulations, and practical Python implementations to build a deep and applicable understanding of the subject.

Knowledge Section

1. Fundamentals of Part-of-Speech (POS) Tagging

1.1. Defining the Task

Part-of-Speech (POS) tagging is a foundational Natural Language Processing (NLP) task that involves assigning a grammatical category—such as noun, verb, adjective, or adverb—to each word (token) in a text. The goal is to enrich the text with a layer of syntactic information, which is a prerequisite for many downstream NLP applications to comprehend the structure and meaning of human language.

The output is a sequence of `(token, tag)` tuples. For example, the sentence "Time flies like an arrow" would be tagged as:

```
[('Time', 'NN'), ('flies', 'VBZ'), ('like', 'IN'), ('an', 'DT'), ('arrow', 'NN')]
```

1.2. The Significance of POS Tagging

POS tagging is a critical preprocessing step that provides essential grammatical context for more complex NLP applications.

- **Syntactic Parsing & Information Extraction:** Identifying nouns, verbs, and other

constituents is essential for building parse trees and understanding subject-verb-object relationships.
- **Named Entity Recognition (NER):** POS tags provide strong clues for NER systems. For instance, sequences of proper nouns (NNP) often indicate a named entity.
- **Word Sense Disambiguation (WSD):** Knowing a word's grammatical function (e.g., "conduct" as a verb vs. a noun) helps resolve meaning ambiguity.
- **Machine Translation:** Understanding the grammatical role of each word is crucial for preserving sentence structure in the target language.

Errors at the POS tagging stage can propagate and degrade the performance of these downstream tasks, highlighting the need for highly accurate taggers.

1.3. The Core Challenge: Lexical Ambiguity

The primary challenge in POS tagging is **lexical ambiguity**, where a single word can belong to multiple grammatical categories depending on its context. A simple dictionary lookup is insufficient.

- **"book"**: Can be a noun ("I read a good **book**.") or a verb ("Please **book** my flight.").
- **"flies"**: Can be a verb ("Time **flies** like an arrow.") or a noun ("Fruit **flies** like a banana.").

Effective taggers must leverage contextual clues from surrounding words to resolve this ambiguity. The evolution of algorithms from rule-based systems to Transformers has been driven by the quest for more effective methods to model this context.

1.4. Understanding Tagsets

A **tagset** is the predefined collection of POS tags a model is trained to assign. The choice of tagset determines the granularity of grammatical detail.

- **The Penn Treebank (PTB) Tagset:** A highly influential, fine-grained tagset for English, containing 36 distinct POS tags (e.g., `NN` for singular noun, `NNS` for plural noun, `VB` for base form verb, `VBD` for past tense verb). This detail is invaluable for deep syntactic analysis of English.
- **The Universal Dependencies (UD) Tagset:** A coarse-grained, cross-linguistically

consistent tagset designed to facilitate multilingual NLP model development. It includes universal categories like `NOUN`, `VERB`, `ADJ`, `ADV`, making it easier to train models that generalize across languages.

Modern libraries like spaCy often provide both tagsets, offering the fine-grained detail of PTB and the cross-lingual consistency of UD.

2. Word Embeddings: From Static to Contextual

Before deep learning models can process text, words must be converted into numerical vectors. This is the role of word embeddings.

2.1. Static Embeddings (e.g., Word2Vec, GloVe)

Static embedding algorithms learn a single, fixed vector representation for each word in the vocabulary.

- **How they work:** They are trained on large text corpora, and their learning objective is typically based on a word's co-occurrence with its neighbors (context words). For instance, Word2Vec's "skip-gram" model learns to predict context words from a given center word.
- **Limitation:** They cannot handle ambiguity. The word "bank" has the exact same vector in "river bank" and "investment bank."

2.2. Contextual Embeddings (e.g., ELMo, BERT)

Contextual embedding models generate a different vector for a word each time it appears, based on the specific sentence it's in.

- **How they work:** Models like BERT use deep bidirectional architectures (Transformers) to process the entire sentence at once. The embedding for a word is a function of all other words in the sentence.
- **Advantage:** They excel at resolving lexical ambiguity. The vector for "bank" in "river bank" will be significantly different from its vector in "investment bank," as it is conditioned on the surrounding context. This is the key reason for the superior performance of modern NLP models.

3. Classical Statistical Models

These models formed the backbone of NLP for many years and are essential to understand for their theoretical contributions.

3.1. Hidden Markov Models (HMMs)

HMMs are **generative** models that assume an observable sequence (words) is generated by an underlying sequence of hidden states (tags).

- **Theoretical Framework:** An HMM is defined by:
    1. **Emission Probabilities** $P(word|tag)$: The probability of observing a word given a tag.

    $$P(w_i|t_i) = \frac{\text{count}(w_i, t_i)}{\text{count}(t_i)}$$

    2. **Transition Probabilities** $P(tag_i|tag_{i-1})$: The probability of transitioning from one tag to the previous one. This is based on the **Markov assumption** that the current state only depends on the previous state.

    $$P(t_i|t_{i-1}) = \frac{\text{count}(t_{i-1}, t_i)}{\text{count}(t_{i-1})}$$

    3. **Initial State Probabilities** $P(tag_1)$: The probability of a sentence starting with a particular tag.
- **Inference (The Viterbi Algorithm):** Given a sequence of words, the goal is to find the most probable sequence of tags. The Viterbi algorithm is a dynamic programming approach that efficiently finds this optimal path without exhaustively checking all possibilities.
- **Limitations:**
    - **Strict Independence Assumptions:** The Markov assumption (tag depends only on previous tag) and the word independence assumption (word depends only on its tag) prevent the model from using long-range or richer contextual features.
    - **Generative Nature:** HMMs model the joint probability $P(\text{words}, \text{tags})$. This forces them to model the distribution of the

words themselves, which is complex and not strictly necessary for the tagging task.

3.2. Conditional Random Fields (CRFs)

CRFs are **discriminative** models that overcome the key limitations of HMMs. They are a major advancement for sequence labeling.

- **Theoretical Framework:** Instead of modeling the joint probability, a CRF models the conditional probability $P(\text{tags}|\text{words})$ directly. The probability of a tag sequence y given a word sequence x is defined as:

  $$P(y|x) = \frac{1}{Z(x)} \exp\left(\sum_{j=1}^{m}\sum_{i=1}^{n} \lambda_j f_j(y_{i-1}, y_i, x, i)\right)$$

  Where:

  - f_j is a feature function (e.g., "is the current word capitalized and the previous tag DT?").
  - λ_j is a learned weight for that feature.
  - $Z(x)$ is the **partition function**, a normalization factor that sums the scores of all possible tag sequences for the given input x. It is computationally expensive but ensures a valid probability distribution.

  $$Z(x) = \sum_{y'} \exp\left(\sum_{j=1}^{m}\sum_{i=1}^{n} \lambda_j f_j(y'_{i-1}, y'_i, x, i)\right)$$

- **Key Advantage:** By conditioning on the entire input sequence `x`, CRFs can incorporate a rich, arbitrary, and overlapping set of features from anywhere in the sentence. This allows for much more sophisticated modeling of context than HMMs. Feature engineering (designing `f_j`) is critical. Common features include word identity, suffixes/prefixes, capitalization, and contextual words.
- **Python Implementation with `sklearn-crfsuite`:**
  This library provides a convenient wrapper for the powerful `CRFsuite` implementation. The process involves defining a feature extraction function and then training the model.

```python
# First, install the library: pip install sklearn-crfsuite
import sklearn_crfsuite
from sklearn.model_selection import train_test_split
from sklearn_crfsuite import metrics
import nltk

# 1. Load data and define feature extraction
nltk.download('treebank', quiet=True)
nltk.download('universal_tagset', quiet=True)
tagged_sentences = nltk.corpus.treebank.tagged_sents(tagset='universal')

def word2features(sent, i):
    word = sent[i][0]
    features = {
        'bias': 1.0,
        'word.lower()': word.lower(),
        'word[-3:]': word[-3:],
        'word.isupper()': word.isupper(),
        'word.istitle()': word.istitle(),
        'word.isdigit()': word.isdigit(),
    }
    if i > 0:
        prev_word = sent[i-1][0]
        features.update({
            '-1:word.lower()': prev_word.lower(),
            '-1:word.istitle()': prev_word.istitle(),
        })
    else:
        features['BOS'] = True # Beginning of Sentence

    if i < len(sent)-1:
        next_word = sent[i+1][0]
        features.update({
            '+1:word.lower()': next_word.lower(),
            '+1:word.istitle()': next_word.istitle(),
        })
    else:
        features['EOS'] = True # End of Sentence
    return features
```

```
def sent2features(sent):
    return [word2features(sent, i) for i in range(len(sent))]

def sent2labels(sent):
    return [label for token, label in sent]

# 2. Prepare data
X = [sent2features(s) for s in tagged_sentences]
y = [sent2labels(s) for s in tagged_sentences]
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random

# 3. Train the CRF model
crf = sklearn_crfsuite.CRF(
    algorithm='lbfgs', c1=0.1, c2=0.1,
    max_iterations=100, all_possible_transitions=True
)
crf.fit(X_train, y_train)

# 4. Evaluate
labels = list(crf.classes_)
y_pred = crf.predict(X_test)
f1_score = metrics.flat_f1_score(y_test, y_pred, average='weighted', labels=lab
print(f"F1 Score: {f1_score:.4f}")

# Example prediction
test_sent = [("The", "DET"), ("cat", "NOUN"), ("sat", "VERB")]
print("Predicted:", list(zip([t[0] for t in test_sent], crf.predict([sent2featu
```

4. Deep Learning Architectures for Sequence Tagging

Deep learning models automate the feature engineering process, learning rich representations directly from data.

4.1. Bidirectional LSTMs (BiLSTMs)

Recurrent Neural Networks (RNNs) process sequences, but simple RNNs suffer from the **vanishing gradient problem**, making it hard to capture long-range dependencies. **Long Short-Term Memory (LSTM)** networks solve this with a memory cell and gating mechanisms (input, forget, output) that control the flow of information.

A standard LSTM is unidirectional. A **Bidirectional LSTM (BiLSTM)** uses two LSTMs: one processes the sequence from left-to-right, and the other from right-to-left. At each time step, the hidden states from both are concatenated. This provides a rich representation of each word that is informed by its entire past and future context.

4.2. The BiLSTM-CRF Architecture: The Best of Both Worlds

While a BiLSTM is excellent at learning contextual features, adding a simple `softmax` layer on top makes independent tagging decisions for each word. This can lead to grammatically invalid tag sequences (e.g., `I-NOUN` following `B-VERB`).

The **BiLSTM-CRF** architecture combines the strengths of both worlds:

1. **BiLSTM Layer:** Acts as a powerful, automated feature extractor. It takes word embeddings and outputs a sequence of high-level feature vectors (emission scores), one for each word.
2. **CRF Layer:** Takes the entire sequence of emission scores from the BiLSTM. It learns a matrix of transition scores representing the likelihood of one tag following another. It then uses both emission and transition scores to find the globally optimal tag path using the Viterbi algorithm.

This synergy is highly effective: the BiLSTM learns *what features are important*, and the CRF learns the *rules of how tags should be structured*, enforcing output constraints.

- **Loss Calculation:** The model is trained by minimizing the negative log-likelihood of the correct tag sequence.

$$\text{Loss} = -\log P(y_{\text{gold}}|x) = \log\left(\sum_{y'} e^{\text{Score}(x,y')}\right) - \text{Score}(x, y_{\text{gold}})$$

  - The first term is the log of the partition function (total score of all possible paths), calculated efficiently via the forward algorithm.
  - The second term is the score of the ground-truth path.

5. The Transformer Era: Attention and Pre-trained Models

(Information current as of early 2024)
The Transformer architecture, introduced in "Attention Is All You Need" (2017), revolutionized NLP. Models like BERT (2018) established new state-of-the-art results on most tasks, including POS tagging.

5.1. The Self-Attention Mechanism

Unlike RNNs, which process sequences step-by-step, Transformers process all tokens in parallel. The core innovation is the **self-attention mechanism**, which allows the model to weigh the importance of all other words in the input when creating a representation for a given word.

- **Query, Key, Value (Q, K, V):** For each input token, the model creates three vectors: a Query, a Key, and a Value.
    i. The **Query (Q)** represents the current word asking for information.
    ii. The **Keys (K)** from all other words represent their "advertised" information.
    iii. The **Values (V)** from all other words represent the actual information they hold.
- **Scaled Dot-Product Attention:** The attention score is calculated by taking the dot product of the current word's Query with all other words' Keys. This score is scaled, passed through a softmax to get weights, and then used to create a weighted sum of all the Value vectors.

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

  This process allows the model to build deeply contextual representations by dynamically focusing on the most relevant parts of the input sequence for each token.

5.2. Fine-Tuning BERT for Token Classification

POS tagging is framed as a **Token Classification** task. The standard approach is:

1. Take a pre-trained Transformer encoder (like BERT).
2. Add a single linear classification layer on top.
3. Fine-tune the entire model on a task-specific labeled dataset (e.g., CoNLL-2003).

- **The Subword Alignment Challenge:** Transformers use **subword tokenization** (e.g., WordPiece), where an infrequent word like "tagging" might be split into `['tag', '##ging']`. This creates a mismatch with word-level labels.
- **The Solution:**
   i. Assign the correct label only to the *first* subword token of each original word.
   ii. Assign a special ignore index (`-100`) to all subsequent subword tokens and special tokens like `[CLS]` and `[SEP]`.
   iii. The loss function (e.g., `CrossEntropyLoss`) automatically ignores these `-100` labels during training.
- **Fine-tuning with Hugging Face `transformers`:**
  The `transformers` library provides a high-level API for this process.

```
# pip install transformers datasets evaluate seqeval
from datasets import load_dataset
from transformers import AutoTokenizer, AutoModelForTokenClassification, Traini
import numpy as np
import evaluate

# 1. Load data and tokenizer
raw_datasets = load_dataset("conll2003")
pos_feature = raw_datasets["train"].features["pos_tags"]
label_names = pos_feature.feature.names
id2label = {i: label for i, label in enumerate(label_names)}
label2id = {label: i for i, label in enumerate(label_names)}

model_checkpoint = "bert-base-cased"
tokenizer = AutoTokenizer.from_pretrained(model_checkpoint)

# 2. Tokenize and align labels
def tokenize_and_align_labels(examples):
    tokenized_inputs = tokenizer(examples["tokens"], truncation=True, is_split_
    labels = []
    for i, label in enumerate(examples["pos_tags"]):
        word_ids = tokenized_inputs.word_ids(batch_index=i)
        previous_word_idx = None
        label_ids = []
        for word_idx in word_ids:
            if word_idx is None:
                label_ids.append(-100)
            elif word_idx != previous_word_idx:
                label_ids.append(label[word_idx])
            else:
                label_ids.append(-100)
            previous_word_idx = word_idx
        labels.append(label_ids)
    tokenized_inputs["labels"] = labels
    return tokenized_inputs

tokenized_datasets = raw_datasets.map(tokenize_and_align_labels, batched=True)

# 3. Configure Trainer
```

```
data_collator = DataCollatorForTokenClassification(tokenizer=tokenizer)
seqeval = evaluate.load("seqeval")

def compute_metrics(p):
    predictions, labels = p
    predictions = np.argmax(predictions, axis=2)
    true_predictions = [
        [label_names[p] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]
    true_labels = [
        [label_names[l] for (p, l) in zip(prediction, label) if l != -100]
        for prediction, label in zip(predictions, labels)
    ]
    results = seqeval.compute(predictions=true_predictions, references=true_lab
    return {"f1": results["overall_f1"], "accuracy": results["overall_accuracy"

model = AutoModelForTokenClassification.from_pretrained(
    model_checkpoint, num_labels=len(label_names), id2label=id2label, label2id=
)

training_args = TrainingArguments(
    output_dir="bert-finetuned-pos", learning_rate=2e-5,
    per_device_train_batch_size=16, num_train_epochs=3,
    weight_decay=0.01, evaluation_strategy="epoch",
    save_strategy="epoch", load_best_model_at_end=True,
)

trainer = Trainer(
    model=model, args=training_args,
    train_dataset=tokenized_datasets["train"], eval_dataset=tokenized_datasets[
    tokenizer=tokenizer, data_collator=data_collator,
    compute_metrics=compute_metrics,
)

# 4. Train
# trainer.train() # Uncomment to run training
```

6. Evaluation Metrics for Sequence Tagging

- **Accuracy:** The percentage of tokens that are assigned the correct tag. It can be misleading if some tags are much more frequent than others.

$$\text{Accuracy} = \frac{\text{Number of Correctly Tagged Tokens}}{\text{Total Number of Tokens}}$$

- **Precision, Recall, and F1-Score:** These are often calculated per-class (for each tag) and then aggregated (e.g., using a macro or weighted average).
    - **Precision (for a class C):** Of all tokens predicted as class C, what fraction were actually class C?

$$P = \frac{TP}{TP + FP}$$

    - **Recall (for a class C):** Of all tokens that were actually class C, what fraction did the model correctly predict as class C?

$$R = \frac{TP}{TP + FN}$$

    - **F1-Score:** The harmonic mean of precision and recall, providing a single metric that balances both.

$$F1 = 2 \cdot \frac{P \cdot R}{P + R}$$

For sequence tagging, these metrics are typically calculated using libraries like `seqeval`, which correctly handles entity-level evaluation for tasks like NER but can also report token-level metrics suitable for POS tagging.

Interview Questions

Theoretical Questions

1. What is the fundamental difference between a generative model like a Hidden Markov Model (HMM) and a discriminative model like a Conditional Random Field (CRF) for POS tagging?

Answer:
The fundamental difference lies in the probability distribution they model.

- **HMM (Generative):** An HMM is a generative model that learns the **joint probability** of the observations (words) and the labels (tags), $P(\text{words}, \text{tags})$. To do this, it makes strong independence assumptions: the current tag depends only on the previous tag (Markov assumption), and the current word depends only on the current tag. It models how the data was "generated."
- **CRF (Discriminative):** A CRF is a discriminative model that learns the **conditional probability** of the labels given the observations, $P(\text{tags}|\text{words})$. It does not attempt to model the distribution of the words themselves. This frees it from the strict independence assumptions of HMMs. A CRF can use a rich set of overlapping features from the entire input sentence to make a tagging decision at any position.

In summary: An HMM asks "What is the most likely sequence of tags and words?", while a CRF asks "Given this specific sequence of words, what is the most likely sequence of tags?". This direct approach generally leads to higher accuracy for CRFs in sequence labeling tasks.

---

2. What is a major limitation of using a BiLSTM with a softmax layer for sequence tagging, and how does the BiLSTM-CRF model solve it?

Answer:
A major limitation of a standard BiLSTM with a final softmax layer is that it makes **locally normalized, independent decisions** for each token. The softmax calculates a probability distribution over tags for word `i` conditioned on the BiLSTM's hidden state at that position, but it does so without explicitly considering the tag chosen for word `i-1` or `i+1`.

This can lead to grammatically invalid tag sequences. For example, in a chunking task, the model might predict an `I-NP` (inside a Noun Phrase) tag immediately following a `B-VP` (beginning of a Verb Phrase), which is syntactically impossible.

The BiLSTM-CRF model solves this by adding a CRF layer on top of the BiLSTM.

- The **BiLSTM** acts as a powerful feature extractor, producing context-aware emission scores for each word.

- The **CRF layer** does not make local decisions. Instead, it takes the entire sequence of emission scores and adds a matrix of learned **transition scores** (the likelihood of transitioning from one tag to another). It then finds the single highest-scoring tag path for the *entire sentence* using the Viterbi algorithm.

This combination ensures that the final output is not just locally plausible but **globally optimal** and structurally coherent.

---

3. Explain the self-attention mechanism in a Transformer. What are Query, Key, and Value vectors?

Answer:
Self-attention is the core mechanism of the Transformer architecture that allows it to build context-aware representations for each token by weighing the importance of all other tokens in the sequence.

For each input token's embedding, the model learns to project it into three separate vectors:

1. **Query (Q):** Represents the current token's state as it "queries" or looks for information from other tokens. It's like asking, "What information is relevant to me?"
2. **Key (K):** Represents a token's "label" or what kind of information it can provide. It's the counterpart to the Query.
3. **Value (V):** Represents the actual content or substance of the token. This is the information that will be passed on if this token is deemed important.

The process works as follows:

1. **Score:** The Query vector of the current token is multiplied (dot product) with the Key vectors of all tokens in the sequence (including itself). This produces a score indicating the relevance of each token to the current one.
2. **Scale:** The scores are scaled by dividing by the square root of the dimension of the key vectors ($\sqrt{d_k}$) to stabilize gradients.
3. **Softmax:** A softmax function is applied to the scaled scores to get a set of attention weights that sum to 1. These weights represent the importance distribution.

4. **Weighted Sum:** The Value vectors of all tokens are multiplied by their corresponding attention weights and summed up. This produces the final output for the current token—a new representation that is a weighted blend of all other tokens' information, with more "attention" paid to the most relevant ones.

The formula is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

---

4. Why do models like BERT use subword tokenization, and what challenge does this create for token classification tasks like POS tagging? How is it typically solved?

Answer:
Models like BERT use subword tokenization (e.g., WordPiece, BPE) for two primary reasons:

1. **Managing Vocabulary Size:** Instead of needing an enormous vocabulary for every possible word, the model can have a smaller, fixed-size vocabulary of common words and subword units.
2. **Handling Out-of-Vocabulary (OOV) Words:** An unknown word can be broken down into known subword units. This allows the model to infer the meaning of the new word from its constituent parts, making it much more robust to rare or unseen words than models with a fixed word vocabulary.

Challenge for Token Classification:
This creates a **mismatch between tokens and labels**. We have labels for each original word, but the model sees a sequence of subword tokens. For example, the word "unbelievably" might be tokenized into `['un', '##believ', '##ably']` but we only have one label (`ADV`) for the original word.

Solution:
The standard solution is a label alignment strategy:

1. **Assign the label only to the first subword token** of the original word. In our example, `un` would get the `ADV` tag.
2. **Assign a special ignore index (typically -100) to all subsequent subword**

tokens (`##believ` , `##ably`).

3. This special index `-100` is automatically ignored by the loss function during training, so the model is only penalized for its prediction on the first subword of each word. This ensures the model learns to associate the label with the start of a word while correctly handling the tokenization mismatch.

---

5. Prove that the objective function for logistic regression is convex. Why is this property important?

Answer:
The objective function for logistic regression is the negative log-likelihood of the data. For a single data point (x, y) where $y \in \{0, 1\}$, the likelihood is given by $h_\theta(x)^y (1 - h_\theta(x))^{1-y}$, where $h_\theta(x) = \sigma(\theta^T x)$ is the sigmoid function. The negative log-likelihood, or cross-entropy loss, is:

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

To prove convexity, we need to show that its Hessian matrix (the matrix of second partial derivatives) is positive semidefinite.

Let $z = \theta^T x$. The sigmoid function is $\sigma(z) = \frac{1}{1+e^{-z}}$. Its derivative is $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

The first derivative of $J(\theta)$ with respect to a single parameter θ_j is (using the chain rule):

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

$$= -\left[y(1 - \sigma(z)) - (1-y)\sigma(z) \right] x_j = -[y - \sigma(z)] x_j = (\sigma(\theta^T x) - y) x_j$$

The second derivative (an element of the Hessian) is:

$$\frac{\partial^2 J}{\partial \theta_k \partial \theta_j} = \frac{\partial}{\partial \theta_k} [(\sigma(\theta^T x) - y) x_j]$$

$$= x_j \frac{\partial}{\partial \theta_k} \sigma(\theta^T x) = x_j \cdot \sigma'(\theta^T x) \cdot \frac{\partial}{\partial \theta_k}(\theta^T x) = x_j \cdot \sigma(\theta^T x)(1 - \sigma(\theta^T x)) \cdot x_k$$

So, the Hessian matrix H is given by:

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

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

$$v^T H v = v^T (\sigma(\theta^T x)(1 - \sigma(\theta^T x)) x x^T) v$$

The term $\sigma(z)(1 - \sigma(z))$ is a scalar. Since $\sigma(z) \in (0, 1)$, this term is always positive. Let's call it c.

$$v^T H v = c \cdot v^T (x x^T) v = c \cdot (v^T x)(x^T v) = c \cdot (v^T x)^2$$

Since $c > 0$ and $(v^T x)^2 \geq 0$, we have $v^T H v \geq 0$. Thus, the Hessian is positive semidefinite, and the objective function is **convex**.

Importance: Convexity guarantees that any local minimum found by an optimization algorithm (like Gradient Descent) is also the **global minimum**. This means we can be confident that we have found the best possible set of parameters θ for the model, without worrying about getting stuck in suboptimal local minima.

Practical & Coding Questions

1. Implement a single head of scaled dot-product attention from scratch using PyTorch.

Answer:
This implementation will create a function that takes Query, Key, and Value matrices as input and returns the attention output and the attention weights.

```python
import torch
import torch.nn as nn
import torch.nn.functional as F
import math

def scaled_dot_product_attention(query, key, value, mask=None):
    """
    Computes scaled dot-product attention.

    Args:
        query (torch.Tensor): Query tensor, shape (batch_size, num_queries, d_k)
        key (torch.Tensor): Key tensor, shape (batch_size, num_keys, d_k)
        value (torch.Tensor): Value tensor, shape (batch_size, num_keys, d_v)
        mask (torch.Tensor, optional): Mask tensor, shape (batch_size, num_queries, num_ke

    Returns:
        torch.Tensor: The output of the attention mechanism, shape (batch_size, num_queri
        torch.Tensor: The attention weights, shape (batch_size, num_queries, num_keys).
    """
    # Get the dimension of the key vectors
    d_k = key.size(-1)

    # 1. Compute dot product of query and key
    # (batch, num_queries, d_k) @ (batch, d_k, num_keys) -> (batch, num_queries, num_keys
    attention_scores = torch.matmul(query, key.transpose(-2, -1))

    # 2. Scale the scores
    attention_scores = attention_scores / math.sqrt(d_k)

    # 3. Apply mask (if provided)
    # The mask is typically used to prevent attention to future tokens in decoders
    # or to padding tokens.
    if mask is not None:
        # Where mask is 0, set scores to a very small number (-1e9)
        # so they become 0 after softmax
        attention_scores = attention_scores.masked_fill(mask == 0, -1e9)

    # 4. Apply softmax to get attention weights
    # The softmax is applied on the last dimension (num_keys) to get weights that sum to 
    attention_weights = F.softmax(attention_scores, dim=-1)

    # 5. Compute the weighted sum of Value vectors
    # (batch, num_queries, num_keys) @ (batch, num_keys, d_v) -> (batch, num_queries, d_v
    output = torch.matmul(attention_weights, value)

    return output, attention_weights

# --- Example Usage ---
batch_size = 4
seq_len = 10
d_k = 64  # Dimension of key/query
d_v = 128 # Dimension of value

# Create random tensors for Q, K, V
query = torch.randn(batch_size, seq_len, d_k)
key = torch.randn(batch_size, seq_len, d_k)
value = torch.randn(batch_size, seq_len, d_v)

# Compute attention
attention_output, attention_weights = scaled_dot_product_attention(query, key, value)

print("Shape of Attention Output:", attention_output.shape) # Expected: (4, 10, 128)
print("Shape of Attention Weights:", attention_weights.shape) # Expected: (4, 10, 10)
```

---

2. You need to train a CRF model for POS tagging. Implement the Python function that extracts features for a single word in a sentence, considering its context.

Answer:

This function is the core of feature engineering for a CRF. It should extract lexical, morphological, and contextual features for a given word at position `i` within a sentence `sent`.

```python
def word2features(sent, i):
    """
    Extracts features for a word at position i in a sentence.
    'sent' is expected to be a list of (word, tag) tuples.
    """
    word = sent[i][0]

    # Base features for the current word
    features = {
        'bias': 1.0,
        'word.lower()': word.lower(),
        'word.suffix_3': word[-3:],
        'word.suffix_2': word[-2:],
        'word.prefix_3': word[:3],
        'word.isupper()': word.isupper(),
        'word.istitle()': word.istitle(),
        'word.isdigit()': word.isdigit(),
        'word.has_hyphen': '-' in word,
    }

    # Features for the previous word (context)
    if i > 0:
        prev_word = sent[i-1][0]
        features.update({
            '-1:word.lower()': prev_word.lower(),
            '-1:word.istitle()': prev_word.istitle(),
            '-1:word.isupper()': prev_word.isupper(),
        })
    else:
        # Indicate this is the beginning of a sentence
        features['BOS'] = True

    # Features for the next word (context)
    if i < len(sent) - 1:
        next_word = sent[i+1][0]
        features.update({
            '+1:word.lower()': next_word.lower(),
            '+1:word.istitle()': next_word.istitle(),
            '+1:word.isupper()': next_word.isupper(),
        })
    else:
        # Indicate this is the end of a sentence
        features['EOS'] = True

    return features

# --- Example Usage ---
example_sentence = [
    ("The", "DT"),
    ("quick", "JJ"),
    ("brown", "JJ"),
    ("fox", "NN"),
    ("jumps", "VBZ"),
    ("over", "IN"),
    ("the", "DT"),
    ("lazy", "JJ"),
    ("dog", "NN"),
    (".", "."),
]

# Get features for the word "jumps" (at index 4)
features_for_jumps = word2features(example_sentence, 4)

import json
print(json.dumps(features_for_jumps, indent=2))
```

Expected Output:

```json
{
  "bias": 1.0,
  "word.lower()": "jumps",
  "word.suffix_3": "mps",
  "word.suffix_2": "ps",
  "word.prefix_3": "jum",
  "word.isupper()": false,
  "word.istitle()": false,
  "word.isdigit()": false,
  "word.has_hyphen": false,
  "-1:word.lower()": "fox",
  "-1:word.istitle()": false,
  "-1:word.isupper()": false,
  "+1:word.lower()": "over",
  "+1:word.istitle()": false,
  "+1:word.isupper()": false
}
```

---

3. Using `matplotlib`, write a Python function to visualize the transition matrix of a trained CRF model. What insights can you gain from this visualization?

Answer:
This function will take the trained `sklearn-crfsuite` model object and visualize its learned transition probabilities as a heatmap. This is extremely useful for model interpretation.

```python
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from collections import Counter

def plot_crf_transitions(crf_model):
    """
    Visualizes the transition matrix of a trained sklearn-crfsuite model.

    Args:
        crf_model: A trained sklearn_crfsuite.CRF object.
    """
    # Get the learned transition scores
    transitions = crf_model.transition_features_

    # Get the labels in the correct order
    labels = crf_model.classes_

    # Create a matrix to hold the transition scores
    # The dictionary keys are (from_tag, to_tag)
    transition_matrix = np.zeros((len(labels), len(labels)))

    for (from_tag, to_tag), weight in transitions.items():
        # Find the index for each tag
        from_idx = labels.index(from_tag)
        to_idx = labels.index(to_tag)
        transition_matrix[from_idx, to_idx] = weight

    # Plotting the heatmap
    plt.figure(figsize=(12, 10))
    sns.heatmap(
        transition_matrix,
        annot=True,
        fmt=".2f",
        cmap='coolwarm',
        xticklabels=labels,
        yticklabels=labels,
        linewidths=.5
    )
    plt.title("CRF Learned Transition Scores", fontsize=16)
    plt.xlabel("To Tag", fontsize=12)
    plt.ylabel("From Tag", fontsize=12)
    plt.tight_layout()
    plt.show()

# --- Example Usage ---
# We need a trained CRF model. Let's use the one from the Knowledge Section.
# Assuming the CRF training code from section 3.2 has been run and `crf` is a trained mode

# If you don't have the trained model, you can run the training code here first.
# For demonstration, we'll assume `crf` exists and is trained.
# For example, let's just create a dummy object with the right attributes if we can't tra
class DummyCRF:
    def __init__(self):
        self.classes_ = ['NOUN', 'VERB', 'DET', 'ADJ']
        self.transition_features_ = {
            ('DET', 'NOUN'): 2.5, ('DET', 'ADJ'): 1.5, ('DET', 'VERB'): -3.0,
            ('ADJ', 'NOUN'): 3.0, ('ADJ', 'ADJ'): 0.5, ('ADJ', 'VERB'): -2.0,
            ('NOUN', 'VERB'): 2.0, ('NOUN', 'NOUN'): -1.0,
            ('VERB', 'DET'): 1.8, ('VERB', 'NOUN'): 1.0,
        }
# Use a dummy model for visualization if a real one isn't available
dummy_crf = DummyCRF()
plot_crf_transitions(dummy_crf)
```

What insights can you gain?

This visualization reveals the grammatical rules the model has learned.

- **High Positive Scores (Warm colors, e.g., red):** Indicate strong, likely transitions. For example, the score for `(DET, NOUN)` (transitioning from a Determiner to a Noun) will be high and positive, as in "the cat". Similarly, `(ADJ, NOUN)` will be high ("quick fox").
- **High Negative Scores (Cool colors, e.g., blue):** Indicate unlikely or forbidden transitions. The score for `(DET, VERB)` should be highly negative because a verb rarely follows a determiner directly in English.
- **Near-Zero Scores:** Indicate transitions that are neither strongly encouraged nor

discouraged.

- **Diagonal:** The scores on the diagonal show the likelihood of a tag repeating (e.g., `(ADJ, ADJ)` for multiple adjectives: "a big, red ball").

By inspecting this heatmap, you can verify if the model has learned sensible linguistic patterns or if it has picked up spurious correlations from the training data. It's a powerful tool for model debugging and interpretation.