I. Understanding Language Models

Chapter 1: An Introduction to Large Language Models

This chapter provides a comprehensive overview of Large Language Models (LLMs) and the evolution of Language AI, marking humanity's inflection point with AI systems capable of human-like text generation.

The Evolution of Language AI

The chapter traces the development from simple bag-of-words representations in the 1950s to today's sophisticated models:

- **Bag-of-Words (1950s-2000s)**: Simple word counting approach that ignored semantic meaning
- **Dense Vector Embeddings (2013)**: Word2vec introduced semantic representations using neural networks
- **Attention Mechanisms (2014-2017)**: RNNs with attention, leading to the revolutionary Transformer architecture
- **Modern LLMs (2018+)**: BERT (encoder-only) and GPT (decoder-only) architectures

Two Main Model Categories

Representation Models (Encoder-Only)

- **Examples**: BERT
- Focus on understanding and representing language
- Excel at tasks like classification and semantic search
- Generate embeddings rather than text

Generative Models (Decoder-Only)

- **Examples**: GPT family
- Focus on text generation through autocompletion
- Trained using a two-step process: pretraining on vast text corpora, then fine-tuning for specific tasks

Key Concepts

- **Context Window**: Maximum number of tokens a model can process
- **Parameters**: Numerical values representing the model's language understanding (GPT-1: 117M → GPT-3: 175B)
- **Foundation Models**: Base models before task-specific fine-tuning

Applications and Considerations

LLMs excel at diverse tasks including classification, topic detection, document retrieval, chatbots, and multimodal applications. However, responsible development requires addressing bias, transparency, harmful content generation, and intellectual property concerns.

Practical Implementation

The chapter demonstrates generating text using Microsoft's Phi-3-mini model:

```python
from transformers import pipeline

# Create a pipeline
generator = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False,
    max_new_tokens=500,
    do_sample=False
)

# The prompt (user input / query)
messages = [
    {"role": "user", "content": "Create a funny joke about chickens."}
]

# Generate output
output = generator(messages)
print(output[0]["generated_text"])
```

Output: "Why don't chickens like to go to the gym? Because they can't crack the egg-sistence of it!"

Chapter 2: Tokens and Embeddings

This chapter explores the fundamental concepts of tokens and embeddings in Large Language Models (LLMs), covering how text is processed and represented numerically for machine learning.

LLM Tokenization

How Tokenizers Work

Tokenizers break down text into smaller pieces (tokens) before feeding them to language models. Here's a basic example:

```
from transformers import AutoModelForCausalLM, AutoTokenizer

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-
instruct")

prompt = "Write an email apologizing to Sarah for the tragic gardening
mishap. Explain how it happened.<|assistant|>"

# Tokenize the input prompt
input_ids = tokenizer(prompt,
return_tensors="pt").input_ids.to("cuda")

# Generate the text
generation_output = model.generate(
  input_ids=input_ids,
  max_new_tokens=20
)

# Print the output
print(tokenizer.decode(generation_output[0]))
```

Tokenization Methods

Four main approaches exist:

- **Word tokens**: Complete words (less common now)
- **Subword tokens**: Parts of words (most popular)
- **Character tokens**: Individual letters
- **Byte tokens**: Individual bytes representing Unicode characters

Comparing Different Tokenizers

The chapter demonstrates how various tokenizers handle the same text differently:

```
colors_list = [
    '102;194;165', '252;141;98', '141;160;203',
    '231;138;195', '166;216;84', '255;217;47'
]

def show_tokens(sentence, tokenizer_name):
    tokenizer = AutoTokenizer.from_pretrained(tokenizer_name)
    token_ids = tokenizer(sentence).input_ids
```

```python
    for idx, t in enumerate(token_ids):
        print(
            f'\x1b[0;30;48;2;{colors_list[idx % len(colors_list)]}m' +
            tokenizer.decode(t) +
            '\x1b[0m',
            end=' '
        )
```

Key Differences Observed

- **BERT**: Converts to lowercase, adds [CLS] and [SEP] tokens, struggles with emojis
- **GPT-2/GPT-4**: Preserves capitalization and newlines, handles Unicode better
- **StarCoder2**: Optimized for code with better whitespace handling
- **Specialized models**: Include domain-specific tokens (citations, reasoning, etc.)

Token Embeddings

Creating Contextualized Word Embeddings

Language models create context-aware embeddings that represent words differently based on their usage:

```python
from transformers import AutoModel, AutoTokenizer

# Load a tokenizer
tokenizer = AutoTokenizer.from_pretrained("microsoft/deberta-base")

# Load a language model
model = AutoModel.from_pretrained("microsoft/deberta-v3-xsmall")

# Tokenize the sentence
tokens = tokenizer('Hello world', return_tensors='pt')

# Process the tokens
output = model(**tokens)[0]
```

Text Embeddings for Documents

For representing entire sentences or documents as single vectors:

```python
from sentence_transformers import SentenceTransformer

# Load model
model = SentenceTransformer("sentence-transformers/all-mpnet-base-v2")

# Convert text to text embeddings
vector = model.encode("Best movie ever!")
```

Word Embeddings Beyond LLMs

Using Pretrained Word Embeddings

```
import gensim.downloader as api

# Download embeddings (66MB, glove, trained on wikipedia, vector size:
50)
model = api.load("glove-wiki-gigaword-50")

# Find similar words
model.most_similar([model['king']], topn=11)
```

Word2vec Algorithm

The word2vec algorithm uses:

- **Skip-gram**: Selecting neighboring words within a sliding window
- **Negative sampling**: Adding random negative examples to improve training
- **Contrastive training**: Learning to distinguish positive from negative word pairs

Embeddings for Recommendation Systems

Song Recommendation Example

Using playlists as "sentences" and songs as "words":

```
import pandas as pd
from urllib import request
from gensim.models import Word2Vec

# Load playlist data
data =
request.urlopen('https://storage.googleapis.com/maps-premium/dataset/
yes_complete/train.txt')
lines = data.read().decode("utf-8").split('\n')[2:]
playlists = [s.rstrip().split() for s in lines if len(s.split()) > 1]

# Train Word2Vec model on playlists
model = Word2Vec(
    playlists, vector_size=32, window=20, negative=50, min_count=1,
workers=4
)

# Function to print recommendations
def print_recommendations(song_id):
    similar_songs = np.array(
        model.wv.most_similar(positive=str(song_id),topn=5)
    )[:,0]
    return  songs_df.iloc[similar_songs]
```

Key Takeaways

- **Tokenization is crucial**: Different tokenizers produce different results, affecting model performance.
- **Design decisions matter**: Vocabulary size, special tokens, and training data all impact tokenizer behavior.
- **Contextualized embeddings**: Modern LLMs create dynamic word representations based on context.
- **Broad applications**: Embedding concepts extend beyond NLP to recommendation systems and other domains.
- **Foundation for understanding**: Tokens and embeddings are essential for understanding how LLMs process and generate text.

The chapter establishes the groundwork for understanding how language models transform text into numerical representations that enable computational language processing.

Chapter 3: Looking Inside Large Language Models

Overview

This chapter explores how Transformer language models work internally, focusing on text generation models and the mechanics of generative LLMs.

Initial Setup Code

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline

# Load model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-
instruct")

model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)

# Create a pipeline
generator = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False,
    max_new_tokens=50,
    do_sample=False,
)
```

Key Concepts

1. **Autoregressive Token Generation**
   - **Core principle**: LLMs generate text one token at a time, not all at once.
   - **Process**: Each generated token is appended to the input prompt, then the model generates the next token.
   - **Definition**: Models that use their previous predictions to make future predictions are called "autoregressive models."

Example Generation:

```
prompt = "Write an email apologizing to Sarah for the tragic gardening mishap. Explain how it happened."
output = generator(prompt)
print(output[0]['generated_text'])
```

1. **Model Architecture Components**

The Transformer LLM consists of three main parts:

- **Tokenizer**: Breaks text into tokens.
- **Stack of Transformer blocks**: Core processing (32 blocks in the example).
- **Language Modeling Head (LM head):** Converts outputs to probability scores.

Model Structure Inspection:

```
# View model architecture
print(model)  # Shows Phi3ForCausalLM structure with embed_tokens, layers, and lm_head
```

1. **Token Probability and Decoding**

The model outputs probability scores for each token in the vocabulary (~32,064 tokens). The decoding strategy determines which token to select:

```
prompt = "The capital of France is"

# Tokenize the input prompt
input_ids = tokenizer(prompt, return_tensors="pt").input_ids
input_ids = input_ids.to("cuda")

# Get the output of the model before the lm_head
model_output = model.model(input_ids)
```

```
# Get the output of the lm_head
lm_head_output = model.lm_head(model_output[0])

# Get top scoring token
token_id = lm_head_output[0,-1].argmax(-1)
print(tokenizer.decode(token_id))  # Output: "Paris"
```

Decoding Strategies

- **Greedy decoding**: Always pick the highest probability token.
- **Sampling**: Add randomness, sometimes choose the 2nd or 3rd highest probability token.

4. Parallel Processing and Context Length

- Each input token flows through its own processing stream.
- **Context length**: Maximum number of tokens the model can process at once (e.g., 4K tokens).
- Only the final token stream's output is used for the next token prediction.
- Previous streams' calculations are still needed for attention mechanisms.

5. Key-Value (KV) Caching

Critical optimization that dramatically speeds up generation by caching previous computations:

```
# Example timing with cache enabled
%%timeit -n 1
generation_output = model.generate(
  input_ids=input_ids,
  max_new_tokens=100,
  use_cache=True  # 4.5 seconds
)

# Without cache
%%timeit -n 1
generation_output = model.generate(
  input_ids=input_ids,
  max_new_tokens=100,
  use_cache=False  # 21.8 seconds - much slower!
)
```

Transformer Block Components

1. Feedforward Neural Network

- Houses the majority of the model's processing capacity.
- Responsible for memorization and interpolation.
- Stores information learned during training.
- Enables generalization beyond training data.

2. Attention Layer

The attention mechanism incorporates contextual information and works in two steps:

Step 1: Relevance Scoring

- Determines how relevant each previous token is to the current token.
- Uses query-key multiplication followed by softmax normalization.

Step 2: Information Combining

- Multiplies value vectors by relevance scores.
- Sums weighted vectors to produce the final output.

Multi-head Attention

- Runs multiple attention operations in parallel.
- Each "head" focuses on different types of patterns.
- Results are combined for the final output.

Key Matrices in Attention

- **Queries**: Information about the current position.
- **Keys**: Information for relevance scoring.
- **Values**: Information to be combined.

Recent Architectural Improvements

1. Efficient Attention Mechanisms

- **Sparse/Local Attention**: Only attend to limited previous tokens.
- **Multi-query Attention**: Share keys and values across all heads.
- **Grouped-query Attention**: Share keys/values across groups of heads (used in Llama 2/3).
- **Flash Attention**: GPU memory optimization for faster computation.

2. Enhanced Transformer Blocks

Modern improvements include:

- **Pre-normalization**: Layer normalization before attention/feedforward layers.
- **RMSNorm**: More efficient than the original LayerNorm.
- **SwiGLU activation**: Replaces the original ReLU activation.
- **Residual connections**: Skip connections around major components.

3. Positional Embeddings (RoPE)

- **Rotary Positional Embeddings**: Encode both absolute and relative position information.
- Applied during the attention step, not at input.
- Better handles document packing and variable context lengths.
- More efficient for training on mixed document lengths.

Matrix Dimensions Example

```
# View tensor shapes
print(model_output[0].shape)      # torch.Size([1, 6, 3072])
print(lm_head_output.shape)       # torch.Size([1, 6, 32064])
```

Summary of Key Takeaways

- **Sequential Generation**: LLMs generate one token at a time in an autoregressive manner.
- **Three-Part Architecture**: Tokenizer → Transformer blocks → LM head.
- **Parallel Processing**: Each token has its own processing stream.
- **Caching Critical**: KV caching provides ~5x speedup in generation.
- **Attention is Key**: Two-step process of relevance scoring and information combining.
- **Continuous Innovation**: Ongoing improvements in efficiency (Flash Attention, grouped-query attention) and architecture (RoPE, pre-normalization).

II. Using Pretrained Language Models

Chapter 4: Text Classification

This chapter covers various approaches to text classification using both representation and generative language models, demonstrated through sentiment analysis of movie reviews from the Rotten Tomatoes dataset.

Dataset Setup

```
from datasets import load_dataset

# Load our data
data = load_dataset("rotten_tomatoes")
data
```

Dataset Overview

The dataset contains 5,331 positive and 5,331 negative movie reviews split into train, validation, and test sets.

Text Classification with Representation Models

Task-Specific Models

Uses pre-trained models fine-tuned for specific tasks like sentiment analysis:

```python
from transformers import pipeline

# Path to our HF model
model_path = "cardiffnlp/twitter-roberta-base-sentiment-latest"

# Load model into pipeline
pipe = pipeline(
    model=model_path,
    tokenizer=model_path,
    return_all_scores=True,
    device="cuda:0"
)

# Run inference
y_pred = []
for output in tqdm(pipe(KeyDataset(data["test"], "text")),
total=len(data["test"])):
    negative_score = output[0]["score"]
    positive_score = output[2]["score"]
    assignment = np.argmax([negative_score, positive_score])
    y_pred.append(assignment)
```

Result

F1 score of 0.80.

Embedding Models with Supervised Classification

Two-Step Approach

1. Generate embeddings.
2. Train a classifier:

```python
from sentence_transformers import SentenceTransformer
from sklearn.linear_model import LogisticRegression

# Load model
model = SentenceTransformer("sentence-transformers/all-mpnet-base-v2")

# Convert text to embeddings
train_embeddings = model.encode(data["train"]["text"],
show_progress_bar=True)
test_embeddings = model.encode(data["test"]["text"],
show_progress_bar=True)

# Train a logistic regression on our train embeddings
clf = LogisticRegression(random_state=42)
clf.fit(train_embeddings, data["train"]["label"])
```

```
# Predict previously unseen instances
y_pred = clf.predict(test_embeddings)
```

Result

F1 score of 0.85.

Zero-Shot Classification with Embeddings

Classification without labeled training data using cosine similarity:

```
from sklearn.metrics.pairwise import cosine_similarity

# Create embeddings for our labels
label_embeddings = model.encode(["A negative review", "A positive
review"])

# Find the best matching label for each document
sim_matrix = cosine_similarity(test_embeddings, label_embeddings)
y_pred = np.argmax(sim_matrix, axis=1)
```

Result

F1 score of 0.78 (impressive for no labeled data!).

Text Classification with Generative Models

Flan-T5 (Encoder-Decoder Model)

Uses prompt engineering to guide the model:

```
# Load our model
pipe = pipeline(
    "text2text-generation",
    model="google/flan-t5-small",
    device="cuda:0"
)

# Prepare our data
prompt = "Is the following sentence positive or negative? "
data = data.map(lambda example: {"t5": prompt + example['text']})

# Run inference
y_pred = []
for output in tqdm(pipe(KeyDataset(data["test"], "t5")),
total=len(data["test"])):
    text = output[0]["generated_text"]
    y_pred.append(0 if text == "negative" else 1)
```

Result

F1 score of 0.84.

ChatGPT (Closed-Source Model)

Access through OpenAI's API:

```python
import openai

# Create client
client = openai.OpenAI(api_key="YOUR_KEY_HERE")

def chatgpt_generation(prompt, document, model="gpt-3.5-turbo-0125"):
    """Generate an output based on a prompt and an input document."""
    messages=[
        {
            "role": "system",
            "content": "You are a helpful assistant."
        },
        {
            "role": "user",
            "content": prompt.replace("[DOCUMENT]", document)
        }
    ]
    chat_completion = client.chat.completions.create(
        messages=messages,
        model=model,
        temperature=0
    )
    return chat_completion.choices[0].message.content

# Define a prompt template
prompt = """Predict whether the following document is a positive or
negative movie review:

[DOCUMENT]

If it is positive return 1 and if it is negative return 0. Do not give
any other answers.
"""

# Run predictions
predictions = [
    chatgpt_generation(prompt, doc) for doc in tqdm(data["test"]
["text"])
]
y_pred = [int(pred) for pred in predictions]
```

Result

F1 score of 0.91 (highest performance).

Evaluation Function

```
from sklearn.metrics import classification_report

def evaluate_performance(y_true, y_pred):
    """Create and print the classification report"""
    performance = classification_report(
        y_true, y_pred,
        target_names=["Negative Review", "Positive Review"]
    )
    print(performance)
```

Key Takeaways

- **Task-specific models** work well out-of-the-box but are limited to specific tasks.
- **Embedding models** offer flexibility and can be used for various tasks with lightweight classifiers.
- **Zero-shot classification** enables classification without labeled data using creative prompt design.
- **Generative models** require prompt engineering but can achieve high performance.
- **Performance hierarchy**:
  - ChatGPT (0.91)
  - Embeddings + Classifier (0.85)
  - Flan-T5 (0.84)
  - Task-specific (0.80)
  - Zero-shot (0.78)

The chapter demonstrates that there are multiple viable approaches to text classification, each with different trade-offs in terms of performance, computational requirements, and data needs.

Chapter 5: Text Clustering and Topic Modeling

This chapter explores unsupervised learning techniques for grouping semantically similar texts and discovering latent topics in document collections using modern language models.

Key Concepts

- **Text Clustering** aims to group similar texts based on semantic content and relationships. Unlike supervised classification, clustering doesn't require labeled data, making it valuable for exploratory data analysis, finding outliers, and speeding up labeling processes.

- **Topic Modeling** extends clustering by automatically identifying themes or topics within document collections, traditionally represented by keywords rather than single labels.

Common Text Clustering Pipeline

The chapter outlines a three-step pipeline that has gained popularity:

1. **Document Embedding**: Convert textual data to numerical representations that capture semantic meaning:

```
from sentence_transformers import SentenceTransformer

# Create embeddings for documents
embedding_model = SentenceTransformer("thenlper/gte-small")
embeddings = embedding_model.encode(abstracts, show_progress_bar=True)
```

1. **Dimensionality Reduction**: Reduce high-dimensional embeddings to lower dimensions for better clustering:

```
from umap import UMAP

# Reduce from 384 dimensions to 5 dimensions
umap_model = UMAP(
    n_components=5, min_dist=0.0, metric='cosine', random_state=42
)
reduced_embeddings = umap_model.fit_transform(embeddings)
```

1. **Clustering**: Group similar documents using density-based algorithms:

```
from hdbscan import HDBSCAN

# Cluster the reduced embeddings
hdbscan_model = HDBSCAN(
    min_cluster_size=50, metric="euclidean",
cluster_selection_method="eom"
).fit(reduced_embeddings)
clusters = hdbscan_model.labels_
```

BERTopic Framework

BERTopic is a modular topic modeling framework that combines the clustering pipeline with topic representation techniques.

Basic Implementation

```
from bertopic import BERTopic

# Train model with previously defined components
topic_model = BERTopic(
    embedding_model=embedding_model,
    umap_model=umap_model,
    hdbscan_model=hdbscan_model,
    verbose=True
).fit(abstracts, embeddings)

# Explore topics
topic_model.get_topic_info()
topic_model.get_topic(0)  # Inspect specific topic
```

Topic Representation Enhancement

BERTopic uses **c-TF-IDF** (class-based Term Frequency-Inverse Document Frequency) to generate meaningful topic representations by:

- Calculating word frequencies within entire clusters rather than individual documents.
- Weighting words based on their importance to specific clusters versus all clusters.

Advanced Representation Models

The chapter explores several "Lego blocks" for improving topic representations:

- **KeyBERTInspired**: Uses embedding similarity to rerank keywords:

```
from bertopic.representation import KeyBERTInspired

representation_model = KeyBERTInspired()
topic_model.update_topics(abstracts,
representation_model=representation_model)
```

- **Maximal Marginal Relevance (MMR)**: Reduces redundancy by selecting diverse keywords:

```
from bertopic.representation import MaximalMarginalRelevance

representation_model = MaximalMarginalRelevance(diversity=0.2)
topic_model.update_topics(abstracts,
representation_model=representation_model)
```

- **Text Generation Models**: Uses LLMs to generate interpretable topic labels:

```
from transformers import pipeline
from bertopic.representation import TextGeneration

prompt = """I have a topic that contains the following documents:
[DOCUMENTS]

The topic is described by the following keywords: '[KEYWORDS]'.

Based on the documents and keywords, what is this topic about?"""
```

```
generator = pipeline("text2text-generation", model="google/flan-t5-
small")
representation_model = TextGeneration(
    generator, prompt=prompt, doc_length=50, tokenizer="whitespace"
)
topic_model.update_topics(abstracts,
representation_model=representation_model)
```

- **OpenAI Integration**: For more sophisticated topic labeling:

```
import openai
from bertopic.representation import OpenAI

client = openai.OpenAI(api_key="YOUR_KEY_HERE")
representation_model = OpenAI(
    client, model="gpt-3.5-turbo", exponential_backoff=True,
chat=True, prompt=prompt
)
topic_model.update_topics(abstracts,
representation_model=representation_model)
```

Visualization and Analysis

The chapter includes methods for visualizing clusters and topics:

```
# Interactive topic visualization
fig = topic_model.visualize_documents(
    titles,
    reduced_embeddings=reduced_embeddings,
    width=1200,
    hide_annotations=True
)

# Various visualization options
topic_model.visualize_barchart()
topic_model.visualize_heatmap(n_clusters=30)
topic_model.visualize_hierarchy()
```

Key Advantages

- **Modularity**: Each component (embedding, dimensionality reduction, clustering, representation) can be swapped independently.
- **Efficiency**: Topic representation optimization occurs once per topic rather than per document.
- **Flexibility**: Supports various algorithmic variants including guided, supervised, hierarchical, and dynamic topic modeling.
- **Multiple Perspectives**: Different representation models can be used simultaneously for the same topics.

Chapter 6: Prompt Engineering

This chapter covers the fundamentals and advanced techniques of prompt engineering for large language models (LLMs), focusing on generative pre-trained transformers (GPTs).

Using Text Generation Models

Model Selection and Loading

The chapter recommends starting with smaller foundation models like **Phi-3-mini** (3.8B parameters) for learning, as scaling up is easier than scaling down.

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline

# Load model and tokenizer
model = AutoModelForCausalLM.from_pretrained(
    "microsoft/Phi-3-mini-4k-instruct",
    device_map="cuda",
    torch_dtype="auto",
    trust_remote_code=True,
)
tokenizer = AutoTokenizer.from_pretrained("microsoft/Phi-3-mini-4k-
instruct")

# Create a pipeline
pipe = pipeline(
    "text-generation",
    model=model,
    tokenizer=tokenizer,
    return_full_text=False,
    max_new_tokens=500,
    do_sample=False,
)
```

Controlling Model Output

Key parameters for controlling randomness and creativity:

- **Temperature**: Controls randomness/creativity (0 = deterministic, higher = more creative).

```
# Using a high temperature
output = pipe(messages, do_sample=True, temperature=1)
```

- **top_p**: Nucleus sampling that controls token subset consideration.

```
# Using a high top_p
output = pipe(messages, do_sample=True, top_p=1)
```

Basic Prompt Engineering

Prompt Components

Essential elements of effective prompts:

- **Instruction**: The specific task.
- **Data**: Information related to the task.
- **Output indicators**: Guide response format.

Advanced Components

For complex prompts:

- **Persona**: Role the LLM should take.
- **Context**: Additional background information.
- **Format**: Desired output structure.
- **Audience**: Target audience level.
- **Tone**: Voice and style.

```
# Prompt components
persona = "You are an expert in Large Language models. You excel at
breaking down complex papers into digestible summaries.\n"
instruction = "Summarize the key findings of the paper provided.\n"
context = "Your summary should extract the most crucial points that
can help researchers quickly understand the most vital information of
the paper.\n"
data_format = "Create a bullet-point summary that outlines the method.
Follow this up with a concise paragraph that encapsulates the main
results.\n"
audience = "The summary is designed for busy researchers that quickly
need to grasp the newest trends in Large Language Models.\n"
tone = "The tone should be professional and clear.\n"
text = "MY TEXT TO SUMMARIZE"
data = f"Text to summarize: {text}"

# The full prompt
query = persona + instruction + context + data_format + audience +
tone + data
```

Advanced Techniques

In-Context Learning

Providing examples to demonstrate desired behavior:

- **Zero-shot**: No examples.
- **One-shot**: Single example.
- **Few-shot**: Multiple examples.

```
# Use a single example of using the made-up word in a sentence
one_shot_prompt = [
    {
        "role": "user",
        "content": "A 'Gigamuru' is a type of Japanese musical
instrument. An example of a sentence that uses the word Gigamuru is:"
    },
    {
        "role": "assistant",
        "content": "I have a Gigamuru that my uncle gave me as a gift.
I love to play it at home."
    },
    {
        "role": "user",
        "content": "To 'screeg' something is to swing a sword at it.
An example of a sentence that uses the word screeg is:"
    }
]
```

Chain Prompting

Breaking complex problems into sequential steps:

```
# Create name and slogan for a product
product_prompt = [
    {"role": "user", "content": "Create a name and slogan for a
chatbot that leverages LLMs."}
]
outputs = pipe(product_prompt)
product_description = outputs[0]["generated_text"]

# Based on a name and slogan for a product, generate a sales pitch
sales_prompt = [
    {"role": "user", "content": f"Generate a very short sales pitch
for the following product: '{product_description}'"}
]
outputs = pipe(sales_prompt)
sales_pitch = outputs[0]["generated_text"]
```

Reasoning Techniques

Chain-of-Thought

Encouraging step-by-step reasoning before answering:

```
# Answering with chain-of-thought
cot_prompt = [
    {"role": "user", "content": "Roger has 5 tennis balls. He buys 2
more cans of tennis balls. Each can has 3 tennis balls. How many
tennis balls does he have now?"},
    {"role": "assistant", "content": "Roger started with 5 balls. 2
cans of 3 tennis balls each is 6 tennis balls. 5 + 6 = 11. The answer
is 11."},
    {"role": "user", "content": "The cafeteria had 23 apples. If they
used 20 to make lunch and bought 6 more, how many apples do they
have?"}
]
```

- **Zero-Shot Chain-of-Thought**: Using prompts like "Let's think step-by-step."

```
# Zero-shot chain-of-thought
zeroshot_cot_prompt = [
    {"role": "user", "content": "The cafeteria had 23 apples. If they
used 20 to make lunch and bought 6 more, how many apples do they have?
Let's think step-by-step."}
]
```

- **Tree-of-Thought**: Exploring multiple reasoning paths through expert discussion.

```
# Zero-shot tree-of-thought
zeroshot_tot_prompt = [
    {"role": "user", "content": "Imagine three different experts are
answering this question. All experts will write down 1 step of their
thinking, then share it with the group. Then all experts will go on to
the next step, etc. If any expert realizes they're wrong at any point
then they leave. The question is 'The cafeteria had 23 apples. If they
used 20 to make lunch and bought 6 more, how many apples do they
have?' Make sure to discuss the results."}
]
```

Output Verification

Providing Examples for Structure

Guiding output format through examples:

```
# One-shot learning: Providing an example of the output structure
one_shot_template = """Create a short character profile for an RPG
game. Make sure to only use this format:

{
  "description": "A SHORT DESCRIPTION",
  "name": "THE CHARACTER'S NAME",
  "armor": "ONE PIECE OF ARMOR",
  "weapon": "ONE OR MORE WEAPONS"
}
"""
```

- **Grammar Constrained Sampling**: Using libraries like llama-cpp-python to enforce JSON format.

```
from llama_cpp.llama import Llama

# Load Phi-3
llm = Llama.from_pretrained(
    repo_id="microsoft/Phi-3-mini-4k-instruct-gguf",
    filename="*fp16.gguf",
    n_gpu_layers=-1,
    n_ctx=2048,
    verbose=False
)

# Generate output with JSON constraint
output = llm.create_chat_completion(
    messages=[
        {"role": "user", "content": "Create a warrior for an RPG in
JSON format."},
    ],
    response_format={"type": "json_object"},
    temperature=0,
)['choices'][0]['message']["content"]
```

Key Takeaways

- **Iterative Process**: Prompt engineering requires experimentation and refinement.
- **Modular Design**: Break prompts into components that can be mixed and matched.
- **Context Matters**: Order, specificity, and examples significantly impact output quality.
- **Reasoning Enhancement**: Techniques like chain-of-thought can dramatically improve complex problem-solving.
- **Output Control**: Various methods exist to ensure structured, valid, and appropriate responses.

The chapter emphasizes that prompt engineering is both an art and a science, requiring creativity, experimentation, and understanding of how LLMs process and respond to different types of instructions.

Chapter 7 Summary: Advanced Text Generation Techniques and Tools

This chapter explores advanced methods to enhance LLM performance without fine-tuning, building upon prompt engineering concepts from the previous chapter. The techniques are implemented using the LangChain framework and focus on four key areas:

Model I/O: Loading Quantized Models

The chapter introduces quantized models (GGUF format) that compress LLMs by reducing the number of bits needed to represent parameters. This reduces memory requirements while maintaining most accuracy.

```
from langchain import LlamaCpp

# Load quantized Phi-3 model
llm = LlamaCpp(
    model_path="Phi-3-mini-4k-instruct-fp16.gguf",
    n_gpu_layers=-1,
    max_tokens=500,
    n_ctx=2048,
    seed=42,
    verbose=False
)
```

For users without local compute resources:

```
from langchain.chat_models import ChatOpenAI

# Alternative cloud-based option
chat_model = ChatOpenAI(openai_api_key="MY_KEY")
```

Chains: Extending LLM Capabilities

Chains connect LLMs with additional components like prompt templates, tools, or other LLMs to create more powerful systems.

Single Chain with Prompt Template

```
from langchain import PromptTemplate

# Create Phi-3 compatible prompt template
template = """<s><|user|>
{input_prompt}<|end|>
<|assistant|>"""
prompt = PromptTemplate(
    template=template,
    input_variables=["input_prompt"]
)

# Create the chain
basic_chain = prompt | llm

# Use the chain
basic_chain.invoke({
    "input_prompt": "Hi! My name is Maarten. What is 1 + 1?",
})
```

Multiple Chain Example: Story Generation

The chapter demonstrates creating sequential chains for complex tasks like story generation:

```
from langchain import LLMChain

# Title generation chain
template = """<s><|user|>
Create a title for a story about {summary}. Only return the title.<|
end|>
<|assistant|>"""
title_prompt = PromptTemplate(template=template,
input_variables=["summary"])
title = LLMChain(llm=llm, prompt=title_prompt, output_key="title")

# Character description chain
template = """<s><|user|>
Describe the main character of a story about {summary} with the title
{title}. Use only two sentences.<|end|>
<|assistant|>"""
character_prompt = PromptTemplate(
    template=template, input_variables=["summary", "title"]
)
character = LLMChain(llm=llm, prompt=character_prompt,
output_key="character")

# Story generation chain
template = """<s><|user|>
Create a story about {summary} with the title {title}. The main
character is: {character}. Only return the story and it cannot be
longer than one paragraph. <|end|>
<|assistant|>"""
story_prompt = PromptTemplate(
    template=template, input_variables=["summary", "title",
"character"]
)
story = LLMChain(llm=llm, prompt=story_prompt, output_key="story")

# Combine all chains
llm_chain = title | character | story
```

Memory: Helping LLMs Remember Conversations

LLMs are stateless by default, but memory can be added to maintain conversation context.

Conversation Buffer Memory

Stores the complete conversation history:

```
from langchain.memory import ConversationBufferMemory

# Updated prompt template with chat history
template = """<s><|user|>Current conversation:{chat_history}

{input_prompt}<|end|>
<|assistant|>"""

prompt = PromptTemplate(
    template=template,
    input_variables=["input_prompt", "chat_history"]
)

# Create memory and chain
memory = ConversationBufferMemory(memory_key="chat_history")
llm_chain = LLMChain(
    prompt=prompt,
    llm=llm,
    memory=memory
)
```

Windowed Conversation Buffer Memory

Retains only the last k conversations to manage token limits:

```
from langchain.memory import ConversationBufferWindowMemory

# Retain only the last 2 conversations
memory = ConversationBufferWindowMemory(k=2,
memory_key="chat_history")
llm_chain = LLMChain(
    prompt=prompt,
    llm=llm,
    memory=memory
)
```

Conversation Summary Memory

Uses an LLM to summarize conversation history, reducing token usage while preserving context:

```
from langchain.memory import ConversationSummaryMemory

# Create summary prompt template
summary_prompt_template = """<s><|user|>Summarize the conversations
and update with the new lines.
```

```
Current summary:
{summary}

new lines of conversation:
{new_lines}

New summary:<|end|>
<|assistant|>"""
summary_prompt = PromptTemplate(
    input_variables=["new_lines", "summary"],
    template=summary_prompt_template
)

# Create summarizing memory
memory = ConversationSummaryMemory(
    llm=llm,
    memory_key="chat_history",
    prompt=summary_prompt
)
llm_chain = LLMChain(
    prompt=prompt,
    llm=llm,
    memory=memory
)
```

Agents: Creating Systems of LLMs

Agents use LLMs to determine which actions to take and in what order, following the **ReAct** (Reasoning and Acting) framework with three iterative steps:

1. **Thought**: What should I do next and why?
2. **Action**: Execute a specific tool or task.
3. **Observation**: Analyze the results.

ReAct Agent Implementation

```
import os
from langchain_openai import ChatOpenAI
from langchain.agents import load_tools, Tool, AgentExecutor,
create_react_agent
from langchain.tools import DuckDuckGoSearchResults

# Use more powerful LLM for agent tasks
os.environ["OPENAI_API_KEY"] = "MY_KEY"
openai_llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)

# Create ReAct template
react_template = """Answer the following questions as best you can.
You have access to the following tools:
```

```
{tools}

Use the following format:

Question: the input question you must answer
Thought: you should always think about what to do
Action: the action to take, should be one of [{tool_names}]
Action Input: the input to the action
Observation: the result of the action
... (this Thought/Action/Action Input/Observation can repeat N times)
Thought: I now know the final answer
Final Answer: the final answer to the original input question

Begin!

Question: {input}
Thought:{agent_scratchpad}"""

prompt = PromptTemplate(
    template=react_template,
    input_variables=["tools", "tool_names", "input",
"agent_scratchpad"]
)

# Define tools
search = DuckDuckGoSearchResults()
search_tool = Tool(
    name="duckduck",
    description="A web search engine. Use this to as a search engine
for general queries.",
    func=search.run,
)

tools = load_tools(["llm-math"], llm=openai_llm)
tools.append(search_tool)

# Create and execute agent
agent = create_react_agent(openai_llm, tools, prompt)
agent_executor = AgentExecutor(
    agent=agent, tools=tools, verbose=True, handle_parsing_errors=True
)

# Example usage
agent_executor.invoke({
    "input": "What is the current price of a MacBook Pro in USD? How
much would it cost in EUR if the exchange rate is 0.85 EUR for 1 USD."
})
```

Key Takeaways

The chapter emphasizes that these techniques show their true power when combined rather than used in isolation. Each method addresses different aspects of LLM limitations:

- **Chains** enable modular, reusable components and complex multi-step processes.
- **Memory** provides conversation continuity with trade-offs between speed, memory usage, and accuracy.
- **Agents** create autonomous systems that can interact with external tools and make decisions.

Chapter 8: Semantic Search and Retrieval-Augmented Generation

Overview

This chapter explores three key applications of language models in search systems:

- **Dense Retrieval**: Using text embeddings for semantic similarity search.
- **Reranking**: Improving search result ordering using language models.
- **RAG (Retrieval-Augmented Generation)**: Combining search with text generation for factual answers.

Dense Retrieval

Dense retrieval converts text into embeddings (numeric vectors) and finds documents with embeddings nearest to the query embedding. This enables semantic search based on meaning rather than just keyword matching.

Basic Dense Retrieval Example

```
import cohere
import numpy as np
import pandas as pd
from tqdm import tqdm

# Paste your API key here. Remember to not share publicly
api_key = ''

# Create and retrieve a Cohere API key from os.cohere.ai
co = cohere.Client(api_key)

# Example text from Interstellar Wikipedia article
text = """
Interstellar is a 2014 epic science fiction film co-written, directed,
and produced by Christopher Nolan.
```

```
It stars Matthew McConaughey, Anne Hathaway, Jessica Chastain, Bill
Irwin, Ellen Burstyn, Matt Damon, and Michael Caine.
Set in a dystopian future where humanity is struggling to survive, the
film follows a group of astronauts who travel through a wormhole near
Saturn in search of a new home for mankind.

Brothers Christopher and Jonathan Nolan wrote the screenplay, which
had its origins in a script Jonathan developed in 2007.
Caltech theoretical physicist and 2017 Nobel laureate in Physics[4]
Kip Thorne was an executive producer, acted as a scientific
consultant, and wrote a tie-in book, The Science of Interstellar.
Cinematographer Hoyte van Hoytema shot it on 35 mm movie film in the
Panavision anamorphic format and IMAX 70 mm.
Principal photography began in late 2013 and took place in Alberta,
Iceland, and Los Angeles.
Interstellar uses extensive practical and miniature effects and the
company Double Negative created additional digital effects.

Interstellar premiered on October 26, 2014, in Los Angeles.
In the United States, it was first released on film stock, expanding
to venues using digital projectors.
The film had a worldwide gross over $677 million (and $773 million
with subsequent re-releases), making it the tenth-highest grossing
film of 2014.
It received acclaim for its performances, direction, screenplay,
musical score, visual effects, ambition, themes, and emotional weight.
It has also received praise from many astronomers for its scientific
accuracy and portrayal of theoretical astrophysics. Since its
premiere, Interstellar gained a cult following,[5] and now is regarded
by many sci-fi experts as one of the best science-fiction films of all
time.
Interstellar was nominated for five awards at the 87th Academy Awards,
winning Best Visual Effects, and received numerous other accolades"""

# Split into a list of sentences
texts = text.split('.')

# Clean up to remove empty spaces and new lines
texts = [t.strip(' \n') for t in texts]

# Get the embeddings
response = co.embed(
  texts=texts,
  input_type="search_document",
).embeddings

embeds = np.array(response)
print(embeds.shape)  # Output: (15, 4096)

# Build search index with FAISS
```

```
import faiss
dim = embeds.shape[1]
index = faiss.IndexFlatL2(dim)
print(index.is_trained)
index.add(np.float32(embeds))

# Search function
def search(query, number_of_results=3):

  # 1. Get the query's embedding
  query_embed = co.embed(texts=[query],
                input_type="search_query",).embeddings[0]

  # 2. Retrieve the nearest neighbors
  distances , similar_item_ids =
index.search(np.float32([query_embed]), number_of_results)

  # 3. Format the results
  texts_np = np.array(texts) # Convert texts list to numpy for easier
indexing
  results = pd.DataFrame(data={'texts': texts_np[similar_item_ids[0]],
                              'distance': distances[0]})

  # 4. Print and return the results
  print(f"Query:'{query}'\nNearest neighbors:")
  return results

# Example search
query = "how precise was the science"
results = search(query)
results
```

Keyword Search Comparison (BM25)

```
from rank_bm25 import BM25Okapi
from sklearn.feature_extraction import _stop_words
import string

def bm25_tokenizer(text):
    tokenized_doc = []
    for token in text.lower().split():
        token = token.strip(string.punctuation)

        if len(token) > 0 and token not in
_stop_words.ENGLISH_STOP_WORDS:
            tokenized_doc.append(token)
    return tokenized_doc

tokenized_corpus = []
for passage in tqdm(texts):
    tokenized_corpus.append(bm25_tokenizer(passage))

bm25 = BM25Okapi(tokenized_corpus)

def keyword_search(query, top_k=3, num_candidates=15):
    print("Input question:", query)

    ##### BM25 search (lexical search) #####
    bm25_scores = bm25.get_scores(bm25_tokenizer(query))
    top_n = np.argpartition(bm25_scores, -num_candidates)[-
num_candidates:]
    bm25_hits = [{'corpus_id': idx, 'score': bm25_scores[idx]} for idx
in top_n]
    bm25_hits = sorted(bm25_hits, key=lambda x: x['score'],
reverse=True)

    print(f"Top-3 lexical search (BM25) hits")
    for hit in bm25_hits[0:top_k]:
        print("\t{:.3f}\t{}".format(hit['score'],
texts[hit['corpus_id']].replace("\n", " ")))
```

Chunking Strategies

The chapter covers various approaches for splitting long documents:

- **One vector per document**: Embed entire documents or representative parts.
- **Multiple vectors per document**: Chunk documents into smaller pieces.
- **Overlapping chunks**: Include surrounding context to preserve meaning.
- **Sentence-based chunking**: Each sentence as a chunk.
- **Paragraph-based chunking**: Each paragraph as a chunk.

Reranking

Rerankers take search results and reorder them by relevance to improve search quality.

Reranking Example

```
# Basic reranking
query = "how precise was the science"
results = co.rerank(query=query, documents=texts, top_n=3,
return_documents=True)

# Print results
for idx, result in enumerate(results.results):
  print(idx, result.relevance_score , result.document.text)
```

Keyword Search + Reranking Pipeline

```
def keyword_and_reranking_search(query, top_k=3, num_candidates=10):
    print("Input question:", query)

    ##### BM25 search (lexical search) #####
    bm25_scores = bm25.get_scores(bm25_tokenizer(query))
    top_n = np.argpartition(bm25_scores, -num_candidates)[-
num_candidates:]
    bm25_hits = [{'corpus_id': idx, 'score': bm25_scores[idx]} for idx
in top_n]
    bm25_hits = sorted(bm25_hits, key=lambda x: x['score'],
reverse=True)

    print(f"Top-3 lexical search (BM25) hits")
    for hit in bm25_hits[0:top_k]:
        print("\t{:.3f}\t{}".format(hit['score'],
texts[hit['corpus_id']].replace("\n", " ")))


    #Add re-ranking
    docs = [texts[hit['corpus_id']] for hit in bm25_hits]

    print(f"\nTop-3 hits by rank-API ({len(bm25_hits)} BM25 hits re-
ranked)")
    results = co.rerank(query=query, documents=docs, top_n=top_k,
return_documents=True)
    for hit in results.results:
        print("\t{:.3f}\t{}".format(hit.relevance_score,
hit.document.text.replace("\n", " ")))
```

Retrieval-Augmented Generation (RAG)

RAG combines search with text generation to provide factual answers while citing sources.

Basic RAG with API

```
query = "income generated"

# 1- Retrieval
results = search(query)

# 2- Grounded Generation
docs_dict = [{'text': text} for text in results['texts']]
response = co.chat(
    message = query,
    documents=docs_dict
)

print(response.text)
```

Local RAG Implementation

```
# Download model
!wget https://huggingface.co/microsoft/Phi-3-mini-4k-instruct-
gguf/resolve/main/Phi-3-mini-4k-instruct-fp16.gguf

# Load generation model
from langchain import LlamaCpp

llm = LlamaCpp(
    model_path="Phi-3-mini-4k-instruct-fp16.gguf",
    n_gpu_layers=-1,
    max_tokens=500,
    n_ctx=2048,
    seed=42,
    verbose=False
)

# Load embedding model
from langchain.embeddings.huggingface import HuggingFaceEmbeddings

embedding_model = HuggingFaceEmbeddings(
    model_name='thenlper/gte-small'
)

# Create vector database
from langchain.vectorstores import FAISS

db = FAISS.from_texts(texts, embedding_model)

# Create prompt template
from langchain import PromptTemplate

template = """<|user|>
Relevant information:
{context}

Provide a concise answer the following question using the relevant
information provided above:
{question}<|end|>
<|assistant|>"""

prompt = PromptTemplate(
    template=template,
    input_variables=["context", "question"]
)

# RAG pipeline
from langchain.chains import RetrievalQA

rag = RetrievalQA.from_chain_type(
    llm=llm,
    chain_type='stuff',
    retriever=db.as_retriever(),
    chain_type_kwargs={
        "prompt": prompt
    },
    verbose=True
)

# Query the RAG system
rag.invoke('Income generated')
```

Advanced RAG Techniques

- **Query Rewriting**: Improve vague or conversational queries for better retrieval.
- **Multi-query RAG**: Break complex questions into multiple searches.
- **Multi-hop RAG**: Sequential queries for complex reasoning.
- **Query Routing**: Direct queries to appropriate data sources.
- **Agentic RAG**: LLMs acting as agents with multiple tools and data sources.

Evaluation

Search Evaluation - Mean Average Precision (MAP)

The chapter explains MAP as a key metric for evaluating search systems, considering both relevance and ranking position.

RAG Evaluation

RAG systems require evaluation across multiple dimensions:

- **Fluency**: Text quality and coherence.
- **Perceived Utility**: Helpfulness of answers.
- **Citation Recall**: Proportion of statements supported by citations.
- **Citation Precision**: Proportion of citations that support statements.
- **Faithfulness**: Consistency with provided context.
- **Answer Relevance**: How well the answer addresses the question.

Key Takeaways

- Semantic search using embeddings enables meaning-based rather than keyword-based search.
- Rerankers can significantly improve search results with minimal integration effort.
- RAG combines the best of search and generation, reducing hallucinations while providing sources.
- Chunking strategies are crucial for handling long documents effectively.
- Evaluation requires multiple metrics to assess different aspects of system performance.

- Advanced techniques like query rewriting and multi-hop reasoning can handle complex information needs.

Chapter 9: Multimodal Large Language Models

This chapter explores how Large Language Models can be extended beyond text to handle multiple data types (modalities) like images, making them more versatile and capable.

Key Concepts

- **Multimodality** refers to models that can process different types of data - text, images, audio, video, or sensor data. While models may accept multiple input modalities, they don't necessarily generate output in all those modalities.
- The chapter demonstrates how multimodal capabilities can unlock new potential in LLMs, as real-world communication involves more than just text - body language, facial expressions, and visual context all enhance understanding.

Vision Transformers (ViT)

The foundation for visual processing in LLMs comes from Vision Transformers, which adapt the original Transformer architecture for computer vision tasks. Instead of tokenizing text, ViT:

1. Splits images into patches (typically 16×16 pixels)
2. Treats these patches like tokens in text
3. Linearly embeds the patches to create numerical representations
4. Passes these embeddings to the encoder, where they're processed identically to text tokens

This approach allows the same Transformer architecture to handle both text and images seamlessly.

Multimodal Embedding Models

CLIP (Contrastive Language-Image Pre-training)

CLIP creates embeddings for both images and text in the same vector space, enabling direct comparison between visual and textual content. The training process involves:

- **Contrastive Learning**: Using image-caption pairs to maximize similarity between matching pairs and minimize similarity between non-matching pairs
- **Shared Vector Space**: Both image and text embeddings exist in the same dimensional space
- **Applications**: Zero-shot classification, clustering, search, and image generation

OpenCLIP Example Code:

```
from urllib.request import urlopen
from PIL import Image
```

```python
# Load an AI-generated image of a puppy playing in the snow
puppy_path = "https://raw.githubusercontent.com/HandsOnLLM/Hands-On-
Large-Language-Models/main/chapter09/images/puppy.png"
image = Image.open(urlopen(puppy_path)).convert("RGB")

caption = "a puppy playing in the snow"

from transformers import CLIPTokenizerFast, CLIPProcessor, CLIPModel

model_id = "openai/clip-vit-base-patch32"

# Load a tokenizer to preprocess the text
clip_tokenizer = CLIPTokenizerFast.from_pretrained(model_id)

# Load a processor to preprocess the images
clip_processor = CLIPProcessor.from_pretrained(model_id)

# Main model for generating text and image embeddings
model = CLIPModel.from_pretrained(model_id)

# Tokenize our input
inputs = clip_tokenizer(caption, return_tensors="pt")

# Create a text embedding
text_embedding = model.get_text_features(**inputs)

# Preprocess image
processed_image = clip_processor(
    text=None, images=image, return_tensors="pt"
)["pixel_values"]

# Create the image embedding
image_embedding = model.get_image_features(processed_image)

# Calculate similarity
text_embedding /= text_embedding.norm(dim=-1, keepdim=True)
image_embedding /= image_embedding.norm(dim=-1, keepdim=True)

text_embedding = text_embedding.detach().cpu().numpy()
image_embedding = image_embedding.detach().cpu().numpy()
score = np.dot(text_embedding, image_embedding.T)
```

Simplified with sentence-transformers:

```python
from sentence_transformers import SentenceTransformer, util

# Load SBERT-compatible CLIP model
model = SentenceTransformer("clip-ViT-B-32")

# Encode the images
```

```
image_embeddings = model.encode(images)

# Encode the captions
text_embeddings = model.encode(captions)

#Compute cosine similarities
sim_matrix = util.cos_sim(
    image_embeddings, text_embeddings
)
```

Multimodal Text Generation: BLIP-2

BLIP-2 (Bootstrapping Language-Image Pre-training) connects vision and language by integrating a pretrained image encoder with a pretrained Large Language Model (LLM) through a "Querying Transformer" (Q-Former). This method is computationally efficient as it only trains the bridge component.

BLIP-2 Architecture

- **Frozen Vision Transformer**: Processes images.
- **Q-Former Bridge**: Connects vision and text modalities.
- **Frozen LLM**: Generates text responses.

Training Process

1. **Stage 1**: Learn visual-text representations using image-text pairs.
2. **Stage 2**: Convert learned embeddings into "soft visual prompts" for the LLM.

BLIP-2 Implementation:

```
from transformers import AutoProcessor, Blip2ForConditionalGeneration
import torch

# Load processor and main model
blip_processor = AutoProcessor.from_pretrained("Salesforce/blip2-opt-
2.7b")
model = Blip2ForConditionalGeneration.from_pretrained(
    "Salesforce/blip2-opt-2.7b",
    torch_dtype=torch.float16
)

# Send the model to GPU to speed up inference
device = "cuda" if torch.cuda.is_available() else "cpu"
model.to(device)
```

Use Cases

1. Image Captioning

```python
# Load an AI-generated image of a supercar
image = Image.open(urlopen(car_path)).convert("RGB")

# Convert an image into inputs and preprocess it
inputs = blip_processor(image, return_tensors="pt").to(device,
torch.float16)

# Generate image ids to be passed to the decoder (LLM)
generated_ids = model.generate(**inputs, max_new_tokens=20)

# Generate text from the image ids
generated_text = blip_processor.batch_decode(
    generated_ids, skip_special_tokens=True
)
generated_text = generated_text[0].strip()
```

2. Visual Question Answering

```python
# Visual question answering
prompt = "Question: Write down what you see in this picture. Answer:"

# Process both the image and the prompt
inputs = blip_processor(image, text=prompt,
return_tensors="pt").to(device, torch.float16)

# Generate text
generated_ids = model.generate(**inputs, max_new_tokens=30)
generated_text = blip_processor.batch_decode(
    generated_ids, skip_special_tokens=True
)
generated_text = generated_text[0].strip()
```

3. Interactive Chatbot

```python
from IPython.display import HTML, display
import ipywidgets as widgets

def text_eventhandler(*args):
  question = args[0]["new"]
  if question:
    args[0]["owner"].value = ""

    # Create prompt
    if not memory:
      prompt = " Question: " + question + " Answer:"
    else:
      template = "Question: {} Answer: {}."
      prompt = " ".join(
          [
              template.format(memory[i][0], memory[i][1])
              for i in range(len(memory))
          ]
      ) + " Question: " + question + " Answer:"

    # Generate text
    inputs = blip_processor(image, text=prompt, return_tensors="pt")
    inputs = inputs.to(device, torch.float16)
    generated_ids = model.generate(**inputs, max_new_tokens=100)
    generated_text = blip_processor.batch_decode(
        generated_ids,
        skip_special_tokens=True
    )
    generated_text = generated_text[0].strip().split("Question")[0]

    # Update memory
    memory.append((question, generated_text))

    # Assign to output
    output.append_display_data(HTML("<b>USER:</b> " + question))
    output.append_display_data(HTML("<b>BLIP-2:</b> " +
generated_text))
    output.append_display_data(HTML("<br>"))

# Prepare widgets
in_text = widgets.Text()
in_text.continuous_update = False
in_text.observe(text_eventhandler, "value")
output = widgets.Output()
memory = []

# Display chat box
display(
    widgets.VBox(
        children=[output, in_text],
        layout=widgets.Layout(display="inline-flex",
flex_flow="column-reverse"),
    )
)
```

Key Takeaways

- **Advancement in AI**: Multimodal LLMs mark a significant leap in AI capabilities, allowing models to comprehend and reason about visual content in conjunction with text.

- **Powerful Combinations:** The integration of Vision Transformers, contrastive learning methods like CLIP, and bridging architectures such as BLIP-2 results in robust systems.
- **Applications**: These technologies facilitate applications in image captioning, visual question answering, and multimodal conversations.
- **New Opportunities**: The ability to process both visual and textual information opens up new possibilities in various fields that require comprehensive understanding.

III. Training and Fine-Tuning Language Models

Chapter 10: Creating Text Embedding Models

This chapter covers the fundamentals of creating and fine-tuning text embedding models, which are essential for converting unstructured text into numerical representations that capture semantic meaning.

Key Concepts

- **Embedding Models**: These models convert textual data into numerical vectors (embeddings) that represent the semantic meaning of documents. Similar documents should have similar embeddings in vector space.

- **Contrastive Learning**: This is the primary training technique that teaches models about similarity and dissimilarity by presenting pairs of similar and dissimilar documents. This approach helps models learn distinctive characteristics through contrast.

SBERT (Sentence-BERT)

SBERT addresses computational overhead issues associated with cross-encoders by employing a Siamese architecture with shared weights and mean pooling. This allows for the efficient generation of fixed-size embeddings.

Creating Embedding Models from Scratch

Basic Training Setup

```
from datasets import load_dataset
from sentence_transformers import SentenceTransformer, losses
from sentence_transformers.evaluation import
EmbeddingSimilarityEvaluator
from sentence_transformers.trainer import SentenceTransformerTrainer
from sentence_transformers.training_args import
SentenceTransformerTrainingArguments

# Load MNLI dataset from GLUE
```

```
train_dataset = load_dataset("glue", "mnli",
split="train").select(range(50_000))
train_dataset = train_dataset.remove_columns("idx")

# Create evaluator using STSB
val_sts = load_dataset("glue", "stsb", split="validation")
evaluator = EmbeddingSimilarityEvaluator(
    sentences1=val_sts["sentence1"],
    sentences2=val_sts["sentence2"],
    scores=[score/5 for score in val_sts["label"]],
    main_similarity="cosine",
)
```

Loss Functions

1. Softmax Loss (Baseline)

```
embedding_model = SentenceTransformer('bert-base-uncased')
train_loss = losses.SoftmaxLoss(
    model=embedding_model,

sentence_embedding_dimension=embedding_model.get_sentence_embedding_di
mension(),
    num_labels=3
)
# Result: 0.59 Pearson cosine score
```

2. Cosine Similarity Loss

```
# Convert NLI labels: entailment=1, neutral/contradiction=0
mapping = {2: 0, 1: 0, 0:1}
train_dataset = Dataset.from_dict({
    "sentence1": train_dataset["premise"],
    "sentence2": train_dataset["hypothesis"],
    "label": [float(mapping[label]) for label in
train_dataset["label"]]
})

train_loss = losses.CosineSimilarityLoss(model=embedding_model)
# Result: 0.72 Pearson cosine score
```

3. Multiple Negatives Ranking (MNR) Loss

```
# Prepare triplets with anchor, positive, and negative examples
import random
from tqdm import tqdm

mnli = load_dataset("glue", "mnli",
split="train").select(range(50_000))
mnli = mnli.filter(lambda x: True if x["label"] == 0 else False)

train_dataset = {"anchor": [], "positive": [], "negative": []}
soft_negatives = mnli["hypothesis"]
random.shuffle(soft_negatives)
for row, soft_negative in tqdm(zip(mnli, soft_negatives)):
    train_dataset["anchor"].append(row["premise"])
    train_dataset["positive"].append(row["hypothesis"])
    train_dataset["negative"].append(soft_negative)

train_loss =
losses.MultipleNegativesRankingLoss(model=embedding_model)
# Result: 0.80 Pearson cosine score
```

Training Template

```
args = SentenceTransformerTrainingArguments(
    output_dir="embedding_model",
    num_train_epochs=1,
    per_device_train_batch_size=32,
    per_device_eval_batch_size=32,
    warmup_steps=100,
    fp16=True,
    eval_steps=100,
    logging_steps=100,
)

trainer = SentenceTransformerTrainer(
    model=embedding_model,
    args=args,
    train_dataset=train_dataset,
    loss=train_loss,
    evaluator=evaluator
)
trainer.train()
```

Fine-Tuning Approaches

Supervised Fine-Tuning

```
# Use pretrained model instead of BERT base
embedding_model = SentenceTransformer('sentence-transformers/all-
MiniLM-L6-v2')
# Result: 0.85 Pearson cosine score
```

Augmented SBERT (for Limited Data)

Four-Step Process

```
# Step 1: Train cross-encoder on gold dataset
from sentence_transformers.cross_encoder import CrossEncoder

cross_encoder = CrossEncoder("bert-base-uncased", num_labels=2)
cross_encoder.fit(
    train_dataloader=gold_dataloader,
    epochs=1,
    show_progress_bar=True,
    warmup_steps=100,
    use_amp=False
)

# Step 2: Create unlabeled sentence pairs
silver = load_dataset("glue", "mnli",
split="train").select(range(10_000, 50_000))
pairs = list(zip(silver["premise"], silver["hypothesis"]))

# Step 3: Label with cross-encoder
output = cross_encoder.predict(pairs, apply_softmax=True,
show_progress_bar=True)
silver_labels = np.argmax(output, axis=1)

# Step 4: Train bi-encoder on combined gold + silver data
data = pd.concat([gold, silver], ignore_index=True, axis=0)
```

Unsupervised Learning: TSDAE

The Transformer-based Sequential Denoising Auto-Encoder (TSDAE) trains without labels by reconstructing original sentences from noisy versions. This approach allows the model to learn meaningful representations of text data without the need for labeled datasets.

```
import nltk
nltk.download("punkt")

from sentence_transformers.datasets import DenoisingAutoEncoderDataset

# Create noisy data
flat_sentences = mnli["premise"] + mnli["hypothesis"]
damaged_data = DenoisingAutoEncoderDataset(list(set(flat_sentences)))

# Use CLS pooling instead of mean pooling
from sentence_transformers import models
word_embedding_model = models.Transformer("bert-base-uncased")
```

```
pooling_model =
models.Pooling(word_embedding_model.get_word_embedding_dimension(),
"cls")
embedding_model = SentenceTransformer(modules=[word_embedding_model,
pooling_model])

# Denoising loss with tied encoder-decoder weights
train_loss = losses.DenoisingAutoEncoderLoss(embedding_model,
tie_encoder_decoder=True)
# Result: 0.70 Pearson cosine score (unsupervised!)
```

Evaluation

For comprehensive evaluation, use the **Massive Text Embedding Benchmark (MTEB)**. This benchmark provides a standardized framework to assess the performance of text embedding models across various tasks and datasets, ensuring a thorough understanding of their capabilities and limitations.

```
from mteb import MTEB
evaluation = MTEB(tasks=["Banking77Classification"])
results = evaluation.run(model)
```

Key Takeaways

- **MNR Loss**: MNR loss generally outperforms other loss functions in training text embedding models.
- **Hard Negatives**: Incorporating hard negatives (similar but incorrect answers) enhances model performance compared to using easy negatives.
- **Batch Sizes**: Larger batch sizes are more effective when using MNR loss.
- **Pretrained Models**: Starting with pretrained models significantly boosts performance on specific tasks.
- **Augmented SBERT**: Augmented SBERT allows for effective training even with limited labeled data.
- **TSDAE**: The Transformer-based Sequential Denoising Auto-Encoder (TSDAE) offers strong capabilities for unsupervised learning.
- **Domain Adaptation**: Domain adaptation can be achieved through adaptive pretraining followed by fine-tuning, allowing models to better generalize to specific domains.

Chapter 11: Fine-Tuning Representation Models for Classification

This chapter explores various methods for fine-tuning pretrained models like BERT for classification tasks, moving beyond the use of frozen pretrained models to actively training them on specific tasks.

Supervised Classification

Basic Fine-Tuning

Fine-tuning involves updating both the pretrained model and the classification head during training. This approach typically leads to better performance compared to using frozen models, as it allows the model to adapt more effectively to the specific characteristics of the classification task.

```
from datasets import load_dataset
from transformers import AutoTokenizer,
AutoModelForSequenceClassification

# Prepare data and splits
tomatoes = load_dataset("rotten_tomatoes")
train_data, test_data = tomatoes["train"], tomatoes["test"]

# Load model and tokenizer
model_id = "bert-base-cased"
model = AutoModelForSequenceClassification.from_pretrained(
    model_id, num_labels=2
)
tokenizer = AutoTokenizer.from_pretrained(model_id)

from transformers import DataCollatorWithPadding

# Pad to the longest sequence in the batch
data_collator = DataCollatorWithPadding(tokenizer=tokenizer)

def preprocess_function(examples):
   """Tokenize input data"""
   return tokenizer(examples["text"], truncation=True)

# Tokenize train/test data
tokenized_train = train_data.map(preprocess_function, batched=True)
tokenized_test = test_data.map(preprocess_function, batched=True)

import numpy as np
from datasets import load_metric

def compute_metrics(eval_pred):
   """Calculate F1 score"""
   logits, labels = eval_pred
   predictions = np.argmax(logits, axis=-1)

   load_f1 = load_metric("f1")
   f1 = load_f1.compute(predictions=predictions, references=labels)
["f1"]
   return {"f1": f1}

from transformers import TrainingArguments, Trainer

# Training arguments for parameter tuning
training_args = TrainingArguments(
   "model",
   learning_rate=2e-5,
   per_device_train_batch_size=16,
   per_device_eval_batch_size=16,
   num_train_epochs=1,
   weight_decay=0.01,
   save_strategy="epoch",
   report_to="none"
)

# Trainer which executes the training process
trainer = Trainer(
   model=model,
   args=training_args,
   train_dataset=tokenized_train,
   eval_dataset=tokenized_test,
   tokenizer=tokenizer,
   data_collator=data_collator,
   compute_metrics=compute_metrics,
)
```

Full Fine-Tuning

Full fine-tuning achieved an F1 score of **0.85**, compared to **0.80** with frozen pretrained models.

Freezing Layers

You can freeze specific layers of the model to reduce computational costs while still allowing some learning to occur. This strategy helps balance the trade-off between performance and resource efficiency, enabling the model to retain useful features from the pretrained weights while adapting to the new task.

```
# Freeze everything except classification head
for name, param in model.named_parameters():
     # Trainable classification head
     if name.startswith("classifier"):
        param.requires_grad = True
      # Freeze everything else
     else:
        param.requires_grad = False

# Freeze first 10 encoder blocks, train the rest
model_id = "bert-base-cased"
model = AutoModelForSequenceClassification.from_pretrained(
    model_id, num_labels=2
)
tokenizer = AutoTokenizer.from_pretrained(model_id)

# Encoder block 11 starts at index 165 and
# we freeze everything before that block
for index, (name, param) in enumerate(model.named_parameters()):
    if index < 165:
        param.requires_grad = False
```

Results

Results showed that freezing only the classification head resulted in an F1 score of **0.63**, while freezing the first 10 blocks achieved an F1 score of **0.80**. This demonstrates the trade-off between computation and performance in model fine-tuning.

Few-Shot Classification with SetFit

SetFit enables efficient fine-tuning with minimal labeled data through a three-step process:

1. **Generate Positive/Negative Sentence Pairs**: Create pairs of sentences from the labeled data, distinguishing between positive and negative examples.
2. **Fine-Tune Embeddings**: Use contrastive learning to fine-tune the embeddings based on the generated sentence pairs.
3. **Train a Classifier**: Train a classifier on the fine-tuned embeddings to perform the classification task effectively.

```
from setfit import sample_dataset

# We simulate a few-shot setting by sampling 16 examples per class
sampled_train_data = sample_dataset(tomatoes["train"], num_samples=16)

from setfit import SetFitModel

# Load a pretrained SentenceTransformer model
model = SetFitModel.from_pretrained("sentence-transformers/all-mpnet-
base-v2")

from setfit import TrainingArguments as SetFitTrainingArguments
from setfit import Trainer as SetFitTrainer

# Define training arguments
```

```
args = SetFitTrainingArguments(
    num_epochs=3, # The number of epochs to use for contrastive
learning
    num_iterations=20  # The number of text pairs to generate
)
args.eval_strategy = args.evaluation_strategy

# Create trainer
trainer = SetFitTrainer(
    model=model,
    args=args,
    train_dataset=sampled_train_data,
    eval_dataset=test_data,
    metric="f1"
)

# Training loop
trainer.train()
```

Few-Shot Performance

With only **32 labeled documents** (16 per class), SetFit achieved an F1 score of **0.84**, demonstrating impressive few-shot performance.

Continued Pretraining with Masked Language Modeling

This approach adds a third step between pretraining and fine-tuning: continue pretraining on domain-specific data. This step helps adapt the model's representations to better align with the specific characteristics and vocabulary of the target domain, enhancing overall performance in subsequent fine-tuning tasks.

```
from transformers import AutoTokenizer, AutoModelForMaskedLM

# Load model for masked language modeling (MLM)
model = AutoModelForMaskedLM.from_pretrained("bert-base-cased")
tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

def preprocess_function(examples):
   return tokenizer(examples["text"], truncation=True)

# Tokenize data
tokenized_train = train_data.map(preprocess_function, batched=True)
tokenized_train = tokenized_train.remove_columns("label")
tokenized_test = test_data.map(preprocess_function, batched=True)
tokenized_test = tokenized_test.remove_columns("label")

from transformers import DataCollatorForLanguageModeling

# Masking Tokens
data_collator = DataCollatorForLanguageModeling(
    tokenizer=tokenizer,
    mlm=True,
    mlm_probability=0.15
)

# Save pre-trained tokenizer
tokenizer.save_pretrained("mlm")

# Train model
trainer.train()

# Save updated model
model.save_pretrained("mlm")
```

Continued Pretraining Results

Testing the continued pretraining demonstrated that the model learned domain-specific vocabulary. For example, it predicted "movie," "film," and "comedy" for the input "What a horrible [MASK]!" instead of more general terms like "idea" or "day."

Named-Entity Recognition (NER)

Named-Entity Recognition (NER) fine-tunes models for token-level classification, enabling them to identify entities such as:

- **People**
- **Organizations**
- **Locations**

This process enhances the model's ability to recognize and classify specific entities within text, making it valuable for various applications in natural language processing.

```
# The CoNLL-2003 dataset for NER
dataset = load_dataset("conll2003", trust_remote_code=True)

label2id = {
    "O": 0, "B-PER": 1, "I-PER": 2, "B-ORG": 3, "I-ORG": 4,
    "B-LOC": 5, "I-LOC": 6, "B-MISC": 7, "I-MISC": 8
}
id2label = {index: label for label, index in label2id.items()}

from transformers import AutoModelForTokenClassification

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

# Load model
model = AutoModelForTokenClassification.from_pretrained(
    "bert-base-cased",
    num_labels=len(id2label),
    id2label=id2label,
    label2id=label2id
)
```

The key challenge is aligning word-level labels with subtoken-level predictions:

```
def align_labels(examples):
    token_ids = tokenizer(
        examples["tokens"],
        truncation=True,
        is_split_into_words=True
    )
    labels = examples["ner_tags"]

    updated_labels = []
    for index, label in enumerate(labels):

        # Map tokens to their respective word
        word_ids = token_ids.word_ids(batch_index=index)
        previous_word_idx = None
        label_ids = []
        for word_idx in word_ids:

            # The start of a new word
            if word_idx != previous_word_idx:

                previous_word_idx = word_idx
                updated_label = -100 if word_idx is None else
label[word_idx]
                label_ids.append(updated_label)

            # Special token is -100
            elif word_idx is None:
                label_ids.append(-100)

            # If the label is B-XXX we change it to I-XXX
            else:
                updated_label = label[word_idx]
                if updated_label % 2 == 1:
                    updated_label += 1
                label_ids.append(updated_label)

        updated_labels.append(label_ids)

    token_ids["labels"] = updated_labels
    return token_ids

tokenized = dataset.map(align_labels, batched=True)
```

```
import evaluate

# Load sequential evaluation
seqeval = evaluate.load("seqeval")

def compute_metrics(eval_pred):
    # Create predictions
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=2)

    true_predictions = []
    true_labels = []

    # Document-level iteration
    for prediction, label in zip(predictions, labels):

      # Token-level iteration
      for token_prediction, token_label in zip(prediction, label):

        # We ignore special tokens
        if token_label != -100:
          true_predictions.append([id2label[token_prediction]])
          true_labels.append([id2label[token_label]])

    results = seqeval.compute(
    predictions=true_predictions, references=true_labels
)
    return {"f1": results["overall_f1"]}

from transformers import DataCollatorForTokenClassification

# Token-classification DataCollator
data_collator =
DataCollatorForTokenClassification(tokenizer=tokenizer)

from transformers import pipeline

# Save our fine-tuned model
trainer.save_model("ner_model")

# Run inference on the fine-tuned model
token_classifier = pipeline(
    "token-classification",
    model="ner_model",
)
token_classifier("My name is Maarten.")
```

The chapter demonstrates that fine-tuning provides significant performance improvements over frozen models, with various techniques available depending on data availability and computational constraints.

Chapter 12: Fine-Tuning Generation Models

Overview

This chapter covers the process of fine-tuning pretrained text generation models to adapt them for specific tasks and behaviors. Fine-tuning transforms base models into more useful, instruction-following systems through two main approaches: supervised fine-tuning and preference tuning.

The Three LLM Training Steps

1. **Language Modeling (Pretraining)**
   - Base models are pretrained on massive text datasets using next-token prediction.
   - This is a self-supervised method that learns linguistic and semantic representations.
   - Produces foundation models that are harder for end users to interact with directly.
2. **Supervised Fine-Tuning (SFT)**
   - Adapts base models to follow instructions using labeled instruction-response data.
   - Uses next-token prediction but with user input as context.
   - Transforms generative models into instruction/chat models.
3. **Preference Tuning**
   - Further aligns models with human preferences and AI safety expectations.
   - Uses preference data to improve output quality beyond basic instruction following.
   - Distills human preferences into the model's behavior.

Fine-Tuning Approaches

Full Fine-Tuning

- Updates all model parameters using smaller, labeled datasets.
- More effective but computationally expensive and slow.
- Requires significant storage and compute resources.

Parameter-Efficient Fine-Tuning (PEFT)

Adapters

- Add small modular components inside Transformer blocks.
- Fine-tune only 3.6% of parameters while achieving comparable performance.
- Can be specialized for specific tasks and swapped between models.

Low-Rank Adaptation (LoRA)

- Creates smaller matrices that approximate large weight matrices.
- Dramatically reduces trainable parameters (e.g., 150M → 197K per block).

- Exploits the low intrinsic dimensionality of language models.

Quantization (QLoRA)

- Reduces memory requirements by lowering bit precision (16-bit → 4-bit).
- Uses blockwise quantization and distribution-aware blocks.
- Maintains accuracy while significantly reducing VRAM usage.

Practical Implementation

Instruction Tuning with QLoRA

Data Preparation:

```
from transformers import AutoTokenizer
from datasets import load_dataset

# Load a tokenizer to use its chat template
template_tokenizer = AutoTokenizer.from_pretrained(
    "TinyLlama/TinyLlama-1.1BChat-v1.0"
)

def format_prompt(example):
    """Format the prompt to using the <|user|> template TinyLLama is
using"""

    # Format answers
    chat = example["messages"]
    prompt = template_tokenizer.apply_chat_template(chat,
tokenize=False)

    return {"text": prompt}

# Load and format the data using the template TinyLLama is using
dataset = (
    load_dataset("HuggingFaceH4/ultrachat_200k", split="test_sft")
      .shuffle(seed=42)
      .select(range(3_000))
)
dataset = dataset.map(format_prompt)
```

Model Quantization:

```
import torch
from transformers import AutoModelForCausalLM, AutoTokenizer,
BitsAndBytesConfig

model_name = "TinyLlama/TinyLlama-1.1B-intermediate-step-1431k-3T"

# 4-bit quantization configuration - Q in QLoRA
bnb_config = BitsAndBytesConfig(
    load_in_4bit=True,  # Use 4-bit precision model loading
    bnb_4bit_quant_type="nf4",  # Quantization type
    bnb_4bit_compute_dtype="float16",  # Compute dtype
    bnb_4bit_use_double_quant=True,  # Apply nested quantization
)

# Load the model to train on the GPU
model = AutoModelForCausalLM.from_pretrained(
    model_name,
    device_map="auto",
    # Leave this out for regular SFT
    quantization_config=bnb_config,
)
model.config.use_cache = False
model.config.pretraining_tp = 1

# Load LLaMA tokenizer
tokenizer = AutoTokenizer.from_pretrained(model_name,
trust_remote_code=True)
tokenizer.pad_token = "<PAD>"
tokenizer.padding_side = "left"
```

LoRA Configuration:

```
from peft import LoraConfig, prepare_model_for_kbit_training,
get_peft_model

# Prepare LoRA Configuration
peft_config = LoraConfig(
    lora_alpha=32,  # LoRA Scaling
    lora_dropout=0.1,  # Dropout for LoRA Layers
    r=64,  # Rank
    bias="none",
    task_type="CAUSAL_LM",
    target_modules=  # Layers to target
     ["k_proj", "gate_proj", "v_proj", "up_proj", "q_proj", "o_proj",
"down_proj"]
)

# Prepare model for training
model = prepare_model_for_kbit_training(model)
model = get_peft_model(model, peft_config)
```

Training Configuration:

```
from transformers import TrainingArguments

output_dir = "./results"

# Training arguments
```

```python
training_arguments = TrainingArguments(
    output_dir=output_dir,
    per_device_train_batch_size=2,
    gradient_accumulation_steps=4,
    optim="paged_adamw_32bit",
    learning_rate=2e-4,
    lr_scheduler_type="cosine",
    num_train_epochs=1,
    logging_steps=10,
    fp16=True,
    gradient_checkpointing=True
)
```

Training:

```python
from trl import SFTTrainer

# Set supervised fine-tuning parameters
trainer = SFTTrainer(
    model=model,
    train_dataset=dataset,
    dataset_text_field="text",
    tokenizer=tokenizer,
    args=training_arguments,
    max_seq_length=512,
    # Leave this out for regular SFT
    peft_config=peft_config,
)

# Train model
trainer.train()

# Save QLoRA weights
trainer.model.save_pretrained("TinyLlama-1.1B-qlora")
```

Merge Weights:

```python
from peft import AutoPeftModelForCausalLM

model = AutoPeftModelForCausalLM.from_pretrained(
    "TinyLlama-1.1B-qlora",
    low_cpu_mem_usage=True,
    device_map="auto",
)

# Merge LoRA and base model
merged_model = model.merge_and_unload()
```

Evaluation Methods

Word-Level Metrics

- **Perplexity**: Measures how well a model predicts text.
- **ROUGE, BLEU, BERTScore**: Compare generated tokens with reference datasets.
- **Limitations**: These metrics are limited in assessing consistency, fluency, creativity, or correctness.

Benchmarks

Common benchmarks include:

- **MMLU**: 57 different language tasks.
- **GLUE**: General language understanding evaluation.
- **TruthfulQA**: Measures the truthfulness of generated text.
- **GSM8k**: Grade-school math problems.
- **HellaSwag**: Common-sense inference.
- **HumanEval**: Programming problem evaluation.

Automated Evaluation

- **LLM-as-a-judge**: Uses separate LLMs to evaluate output quality.
- **Pairwise Comparison**: A third LLM judges between two model outputs.
- **Scalability**: Scales with LLM improvements but may not capture domain-specific needs.

Human Evaluation

- Considered the gold standard for evaluation.
- **Chatbot Arena**: Community-based voting system using Elo ratings.
- Most relevant for specific use cases but time-intensive.

Preference Tuning / Alignment

Traditional Approach (RLHF with PPO)

1. **Collect Preference Data**: Prompts with accepted/rejected generations.
2. **Train Reward Model**: A separate model that scores generation quality.
3. **Fine-Tune with PPO**: Uses reinforcement learning to optimize rewards.

Direct Preference Optimization (DPO)

- Eliminates the need for a separate reward model.
- Uses the LLM itself as the reference model.
- More stable and accurate than PPO-based methods.

Preference Tuning with DPO

Data Preparation:

```
from datasets import load_dataset

def format_prompt(example):
    """Format the prompt to using the <|user|> template TinyLLama is
using"""

    # Format answers
    system = "<|system|>\n" + example["system"] + "</s>\n"
    prompt = "<|user|>\n" + example["input"] + "</s>\n<|assistant|>\n"
    chosen = example["chosen"] + "</s>\n"
    rejected = example["rejected"] + "</s>\n"

    return {
        "prompt": system + prompt,
        "chosen": chosen,
        "rejected": rejected,
    }

# Apply formatting to the dataset and select relatively short answers
dpo_dataset = load_dataset(
    "argilla/distilabel-intel-orca-dpo-pairs", split="train"
)
dpo_dataset = dpo_dataset.filter(
    lambda r:
        r["status"] != "tie" and
        r["chosen_score"] >= 8 and
        not r["in_gsm8k_train"]
)
dpo_dataset = dpo_dataset.map(
    format_prompt,  remove_columns=dpo_dataset.column_names
)
```

DPO Training:

```
from trl import DPOConfig, DPOTrainer

# Training arguments
training_arguments = DPOConfig(
    output_dir=output_dir,
    per_device_train_batch_size=2,
    gradient_accumulation_steps=4,
    optim="paged_adamw_32bit",
    learning_rate=1e-5,
    lr_scheduler_type="cosine",
    max_steps=200,
    logging_steps=10,
    fp16=True,
    gradient_checkpointing=True,
    warmup_ratio=0.1
)

# Create DPO trainer
dpo_trainer = DPOTrainer(
    model,
    args=training_arguments,
    train_dataset=dpo_dataset,
    tokenizer=tokenizer,
    peft_config=peft_config,
    beta=0.1,
    max_prompt_length=512,
    max_length=512,
)

# Fine-tune model with DPO
dpo_trainer.train()

# Save adapter
dpo_trainer.model.save_pretrained("TinyLlama-1.1B-dpo-qlora")
```

Merging Multiple Adapters:

```
from peft import PeftModel

# Merge LoRA and base model
model = AutoPeftModelForCausalLM.from_pretrained(
    "TinyLlama-1.1B-qlora",
    low_cpu_mem_usage=True,
    device_map="auto",
)
sft_model = model.merge_and_unload()

# Merge DPO LoRA and SFT model
dpo_model = PeftModel.from_pretrained(
    sft_model,
    "TinyLlama-1.1B-dpo-qlora",
    device_map="auto",
)
dpo_model = dpo_model.merge_and_unload()
```

Key Takeaways

- **Efficiency**: Parameter-Efficient Fine-Tuning (PEFT) methods like LoRA combined with quantization dramatically reduce computational requirements while maintaining performance.

- **Two-Stage Process**: Supervised Fine-Tuning (SFT) creates instruction-following models, while preference tuning aligns them with human preferences.
- **Evaluation Challenges**: No single metric perfectly captures model quality; domain-specific evaluation is crucial for accurate assessment.
- **Modern Simplifications**: Methods like Direct Preference Optimization (DPO) and Online Reinforcement Preference Optimization (ORPO) combine multiple training stages, reducing complexity.
- **Practical Considerations**: Parameter optimization requires experimentation; techniques like QLoRA make fine-tuning accessible on consumer hardware.

```
!pip install nbconvert[webpdf]
!sudo apt-get install texlive-xetex texlive-fonts-recommended texlive-
plain-generic
```