Lecture 24: Retrieval Augmented Generation

Grounding LLMs in External Knowledge 🔍

PSYC 51.07: Models of Language and Communication

Week 9

Today's Journey 🗺️

What we'll cover

1. **The Problem**: Why parametric memory isn't enough
2. **RAG Basics**: Retrieve, Augment, Generate
3. **Implementation**: Building RAG systems step-by-step
4. **Advanced Techniques**: Self-RAG, Corrective RAG, HyDE
5. **Production Challenges**: Making RAG work in the real world

The Limits of Parametric Memory 🧠

What are the fundamental limitations of storing knowledge in model parameters?

Problems with purely parametric models:

- ❌ **Knowledge cutoff**: No information after training date
- ❌ **Hallucinations**: Models confidently generate false information
- ❌ **No source attribution**: Can't cite where information comes from
- ❌ **Expensive updates**: Retraining for new information costs millions
- ❌ **Privacy concerns**: Sensitive data baked into parameters
- ❌ **Domain specificity**: Limited knowledge of specialized domains
- ❌ **Outdated facts**: World changes but model weights don't

Solution: Combine parametric knowledge with non-parametric retrieval! 🔍

Example: Knowledge Cutoff Problem 📅

User Query (Dec 2024)
"Who won the 2024 US Presidential election?"

Parametric-only LLM:

{❌} "I apologize, but my knowledge was last updated in April 2023, so I cannot tell you about the 2024 election results."

Or worse: Hallucinates an answer!

RAG-enhanced LLM:

{✅} "According to CNN (retrieved Nov 6, 2024), [actual winner] won the 2024 US Presidential election with [details]."

Provides: Fresh info + source!

Retrieval Augmented Generation: Definition 📚

RAG (Lewis et al., 2020)
A technique that enhances LLMs by retrieving relevant documents from an external knowledge base and using them to inform generation.

Core Idea: Instead of relying only on learned parameters, the model can "look things up"!

Traditional LLM:

- Question → Model → Answer
- Only parametric knowledge
- Fixed at training time
- No sources

RAG Pipeline:

RAG Architecture

Worked Example: "What causes the Northern Lights?"

Step	Action	Result
1. Embed	Convert query to vector	`[0.12, -0.45, 0.78, ...]` (384 dims)
2. Search	Find similar vectors in DB	Top-3 docs: scores 0.92, 0.87, 0.85
3. Retrieve	Get actual text chunks	"Aurora borealis occurs when..."
4. Augment	Add context to prompt	System + Context + Query
5. Generate	LLM produces answer	Grounded response with citations

RAG: Step-by-Step Walkthrough 🔍

Query: "What is the capital of Kazakhstan?"

```
1# Step 1: Embed the query
2query = "What is the capital of Kazakhstan?"
3query_embedding = embedding_model.encode(query)
4# Result: numpy array of shape (384,)
5
6# Step 2: Search vector database
7results = vector_db.search(query_embedding, top_k=3)
8# Returns: [
9#   {"text": "Astana is the capital of Kazakhstan...", "score": 0.94},
10#   {"text": "Kazakhstan's capital moved from Almaty...", "score": 0.89},
11#   {"text": "The city was renamed Nur-Sultan in 2019...", "score": 0.85}
12# ]
13
14# Step 3: Build augmented prompt
15context = "\n".join([r["text"] for r in results])
16prompt = f"""Answer based on the context below.
```

RAG: Step-by-Step Walkthrough 🔍

```
21# Step 4: Generate with LLM
22response = llm.generate(prompt)
23# "Astana (previously known as Nur-Sultan) is the capital of Kazakhstan."
```

...continued

RAG Components Deep Dive 🔬

1. Document Processing

```
1# Chunking example
2from langchain.text_splitter import RecursiveCharacterTextSplitter
3
4splitter = RecursiveCharacterTextSplitter(
5    chunk_size=500,      # Target size
6    chunk_overlap=50,    # Overlap between chunks
7    separators=["\n\n", "\n", ". ", " "]
8)
9
10chunks = splitter.split_text(long_document)
11# ["First chunk about topic A...",
12#  "Second chunk continues topic A...",
13#  "Third chunk about topic B..."]
```

2. Embedding & Storage

RAG Implementation Example 💻

Basic RAG with LangChain:

```
1from langchain.vectorstores import Chroma
2from langchain.embeddings import HuggingFaceEmbeddings
3from langchain.llms import HuggingFacePipeline
4from langchain.chains import RetrievalQA
5from langchain.document_loaders import TextLoader
6from langchain.text_splitter import RecursiveCharacterTextSplitter
7
8# 1. Load and chunk documents
9loader = TextLoader('knowledge_base.txt')
10documents = loader.load()
11
12text_splitter = RecursiveCharacterTextSplitter(
13    chunk_size=512,
14    chunk_overlap=50
15)
16chunks = text_splitter.split_documents(documents)
```

RAG Implementation Example 💻

```
21)
22vectordb = Chroma.from_documents(
23    documents=chunks,
24    embedding=embeddings,
25    persist_directory="./chroma_db"
26)
27
28# 3. Set up retriever
29retriever = vectordb.as_retriever(
30    search_type="similarity",
31    search_kwargs={"k": 3}  # Retrieve top 3 chunks
32)
```

...continued

RAG Implementation (cont.) 💻

```
# 4. Create LLM
llm = HuggingFacePipeline.from_model_id(
    model_id="meta-llama/Llama-2-7b-chat-hf",
    task="text-generation",
    model_kwargs={"temperature": 0.7, "max_length": 512}
)

# 5. Create RAG chain
qa_chain = RetrievalQA.from_chain_type(
    llm=llm,
    retriever=retriever,
    return_source_documents=True,
    chain_type="stuff"  # How to combine documents
)

# 6. Query the system
query = "What is retrieval augmented generation?"
result = qa_chain({"query": query})
```

RAG Implementation (cont.) 💻

```
21print("\nSources:")
22for doc in result['source_documents']:
23    print(f"- {doc.metadata['source']}: {doc.page_content[:100]}...")
```

...continued

Tutorial: HuggingFace Advanced RAG -

https://huggingface.co/learn/cookbook/advanced_rag

Evolution of RAG Approaches 📈

Approach	Key Innovation	When to Retrieve
Naive RAG	Always retrieve	Every query
Self-RAG	Model decides	Only when needed
Corrective RAG	Verify relevance	Always, but filter
Agentic RAG	Multi-step reasoning	Tool-based decisions

Trend

Moving from always-retrieve to **adaptive**, **self-correcting** retrieval systems!

Self-RAG: Adaptive Retrieval 🎯

Key Innovation: Model decides when to retrieve

Special Tokens Learned:

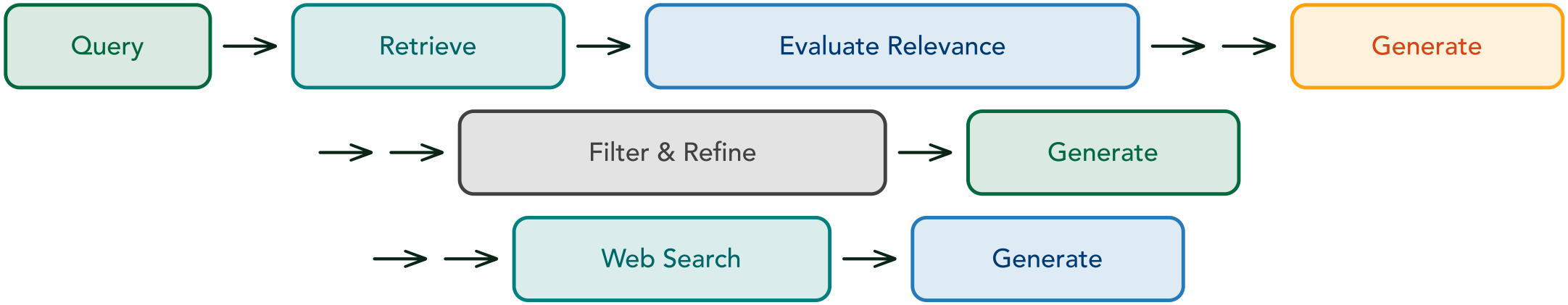
- `[Retrieve]` - Need external info?
- `[Relevant]` - Is retrieved doc useful?
- `[Support]` - Does doc support answer?
- `[Useful]` - Is answer helpful?

Worked Example:

```
1Q: What's 2+2?
2[Retrieve: No]  # No retrieval needed
3A: 4
4
```

Corrective RAG (CRAG) 🔧

Problem: Sometimes retrieved documents are irrelevant or misleading!



Worked Example:

```
1# Query: "Latest COVID vaccine recommendations"
2retrieved_docs = retriever.search(query)  # Returns old 2021 docs
3
4# Evaluator scores relevance
5scores = evaluator.score(query, retrieved_docs)
```

Comparing RAG Approaches 📊

Approach	When Retrieve	Filtering	Latency	Best For
Naive RAG	Always	None	Low	Simple Q&A
Self-RAG	Model decides	Self-reflection	Medium	Adaptive needs
Corrective RAG	Always + verify	Relevance scoring	High	High precision
HyDE	Via hypothesis	Similarity	Medium	Complex queries
Agentic RAG	Tool-based	Multi-step	Highest	Complex workflows

Trade-offs Example:

```
1Simple FAQ bot → Naive RAG (fast, cheap)
2Medical diagnosis assistant → Corrective RAG (accuracy critical)
3Research assistant → Agentic RAG (multi-step reasoning needed)
```

Chunking Strategies

How you split documents dramatically affects retrieval quality!

Fixed-size (Simple)

```
1# Split every 500 chars
2chunks = [text[i:i+500]
3          for i in range(0, len(text), 500)]
4# Problem: "The mitochondria is the power-"
5# "house of the cell." <- split mid-sentence!
```

Recursive (Better)

```
1splitter = RecursiveCharacterTextSplitter(
2    separators=["\n\n", "\n", ". ", " "],
3    chunk_size=500
4)
```

Embedding Models for Retrieval 🎯

Choosing the Right Embedding Model:

Model	Dims	Size	Speed	Quality
all-MiniLM-L6-v2	384	90MB	Fast	Good
BGE-large-en	1024	1.3GB	Medium	Excellent
OpenAI text-embedding-3-small	1536	API	Fast	Excellent

Code Example: Dense vs Hybrid Retrieval

```
1# Dense retrieval (semantic similarity)
2from sentence_transformers import SentenceTransformer
3model = SentenceTransformer('all-MiniLM-L6-v2')
4query_vec = model.encode("What causes headaches?")
5# Finds: "Migraines are often triggered by..." (semantically similar)
```

Vector Databases 🗄️

Purpose: Fast similarity search over millions of embeddings

Quick Start with ChromaDB:

```
import chromadb

# Create client and collection
client = chromadb.Client()
collection = client.create_collection("my_docs")

# Add documents (auto-embeds!)
collection.add(
    documents=["Paris is in France",
               "Berlin is in Germany"],
    ids=["doc1", "doc2"]
)

# Query
```

Prompt Engineering for RAG 📝

Template for Grounded Generation:

```
RAG_PROMPT = """You are a helpful assistant. Answer the question based ONLY on
the context provided below. If the answer is not in the context, say
"I don't have that information."

Context:
{context}

Question: {question}

Instructions:
- Use only information from the context above
- Cite sources using [1], [2], etc.
- Be concise and accurate

Answer:"""
```

Prompt Engineering for RAG 📝

```
21
22question = "When was the Eiffel Tower built?"
23
24response = llm.generate(RAG_PROMPT.format(context=context, question=question))
25# "The Eiffel Tower was completed in 1889 for the World's Fair [1]."
```

...continued

Key Elements

1. Explicit grounding instruction, 2. Source citation format, 3. Fallback for missing info

Production Challenges 🏭

Performance Challenges:

- 💰 **Cost**: Embedding generation + storage + inference
- ⚡ **Latency**: Retrieval adds 50-200ms
- 📏 **Context limits**: LLM window size
- 🎯 **Quality**: Retrieval accuracy
- 🔄 **Freshness**: Keeping index up-to-date

Solutions:

- Cache embeddings
- Semantic caching (similar queries)
- Incremental indexing
- Re-ranking pipelines

Evaluation Metrics for RAG 📊

How to measure RAG quality:

Retrieval Quality:

- **Recall@k**: Are relevant docs in top-k?
- **MRR (Mean Reciprocal Rank)**: Where is first relevant doc?
- **NDCG (Normalized Discounted Cumulative Gain)**: Ranked quality

Generation Quality:

- **Factual accuracy**: Are answers correct?
- **Faithfulness**: Does answer match retrieved docs?
- **Relevance**: Does answer address the question?
- **Citation quality**: Are sources correctly attributed?

Common RAG Failure Modes ⚠️

1. **Retrieval Failures**
    - Wrong documents retrieved

- Relevant docs not in knowledge base
- Poor query formulation

2. **Context Problems**
    - Too much irrelevant context

- Context too long for LLM
- Important info not in retrieved chunks

3. **Generation Issues**
    - Ignores retrieved context

- Hallucinates despite good context

Multimodal RAG 🖼️📄

Beyond text: Retrieving images, tables, code, etc.

- **Vision + Text**
- Use CLIP embeddings for images
- Retrieve relevant diagrams, charts
- Generate answers referencing visual content

  \item **Code Retrieval**

    - Embed code snippets

- Retrieve relevant functions/examples
- Code completion and debugging

  \item **Structured Data**

Graph-Based RAG

Combining knowledge graphs with RAG:

Traditional RAG: Flat document chunks

Graph RAG: Documents + relationships

Advantages:

- Capture entity relationships
- Multi-hop reasoning (A → B → C)
- Better for complex queries
- Explicit knowledge structure

Implementation:

1. Extract entities and relations from documents

HyDE: Hypothetical Document Embeddings 💭

Clever trick: Generate a hypothetical answer first, then retrieve!

```
1# Standard RAG: Query -> Retrieve -> Generate
2query = "What causes the aurora borealis?"
3# Direct embedding may not match scientific docs well
4
5# HyDE: Query -> Generate Hypothesis -> Embed Hypothesis -> Retrieve -> Generat
6hypothesis = llm.generate(f"Write a short explanation: {query}")
7# "The aurora borealis occurs when charged particles from the sun
8#  interact with gases in Earth's atmosphere, causing them to glow."
9
10# Now embed the HYPOTHESIS (an answer-like text)
11hypo_embedding = embed(hypothesis)
12docs = vector_db.search(hypo_embedding)  # Better match to scientific docs!
13
14# Finally generate with real retrieved docs
15final_answer = llm.generate(query, context=docs)
```

RAG vs Fine-Tuning 🤔

When should you use RAG vs fine-tuning your model?

Use RAG when:

- ✅ Knowledge changes frequently
- ✅ Need citations/provenance
- ✅ Privacy concerns (data in DB, not weights)
- ✅ Large knowledge base
- ✅ Multi-domain applications
- ✅ Want to update without retraining

Use Fine-Tuning when:

- ✅ Need specific style/behavior

Future of RAG 🔮

Emerging trends and research directions:

1. **Agentic RAG**
    - LLM decides retrieval strategy
    - Multi-step reasoning with retrieval
    - Tool use (web search, APIs, databases)
2. **Long-context RAG**
    - Models with 1M+ token windows
    - Entire books as context
    - Retrieval still useful for efficiency
3. **Personalized RAG**
    - User-specific knowledge bases

Key Takeaways 🔑

1. **RAG solves fundamental LLM limitations**
    - Knowledge cutoff, hallucination, no citations
2. **Core pipeline: Retrieve → Augment → Generate**
    - Vector search for relevant documents

- Incorporate into prompt

3. **Many variants exist**
    - Naive RAG → Self-RAG → Corrective RAG → Agentic RAG
4. **Key components matter**
    - Chunking strategy, embedding model, vector DB
5. **Production requires careful engineering**
    - Latency, cost, quality evaluation

Readings 📖

Required:

1. **Lewis et al. (2020)**: Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks
   [arXiv]
2. **Asai et al. (2023)**: Self-RAG: Learning to Retrieve, Generate, and Critique
   [arXiv]

Recommended:

- Yan et al. (2024): Corrective RAG [arXiv]
- Gao et al. (2022): Precise Zero-Shot Dense Retrieval (HyDE) [arXiv]
- HuggingFace RAG Tutorial [Tutorial]
- LangChain RAG Docs [Docs]

Questions? 💬

Next: Mixture of Experts!