multimodel_rag

December 11, 2024

1 Setup

[ ]:
```
%pip install -U "unstructured[all-docs]" pillow lxml pillow
%pip install -U chromadb tiktoken
%pip install -U langchain langchain-community langchain-openai langchain-groq
%pip install -U python_dotenv
%pip install -U langchain-ollama
%pip install -U transformers
%pip install -qU "langchain-chroma>=0.1.2"
%pip install -qU langchain-openai
%pip install PyMuPDF
```

[1]:
```
from dotenv import load_dotenv
load_dotenv()
```

[1]:
```
True
```

2 Data Extraction from PDF

2.1 Partition PDF tables, text, and images

[2]:
```
from unstructured.partition.pdf import partition_pdf

output_path = "./pdf/"
file_path = output_path + 'attention_is_all_you_need.pdf'

# Reference: https://docs.unstructured.io/open-source/core-functionality/
 ↪chunking
chunks = partition_pdf(
    filename=file_path,
    infer_table_structure=True,            # extract tables
    strategy="hi_res",                     # mandatory to infer tables

    extract_image_block_types=["Image"],   # Add 'Table' to list to extract␣
 ↪image of tables
    # image_output_dir_path=output_path,   # if None, images and tables will␣
 ↪saved in base64
```

```
    extract_image_block_to_payload=True,   # if true, will extract base64 for␣
↪API usage

    chunking_strategy="by_title",          # or 'basic'
    max_characters=10000,                  # defaults to 500
    combine_text_under_n_chars=2000,       # defaults to 0
    new_after_n_chars=6000,

    # extract_images_in_pdf=True,          # deprecated
)
```

```
/Users/a2024/miniforge3/envs/multimodelrag/lib/python3.12/site-
packages/tqdm/auto.py:21: TqdmWarning: IProgress not found. Please update
jupyter and ipywidgets. See
https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

[3]:
```
len(chunks)
```

[3]:
```
17
```

[4]:
```
[str(type(x)) for x in chunks]
```

[4]:
```
["<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.Table'>",
 "<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.Table'>",
 "<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.Table'>",
 "<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.Table'>",
 "<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.CompositeElement'>"]
```

[5]:
```
#the valuable data tables and compositeElement
set([str(type(el)) for el in chunks])
```

[5]:
```
{"<class 'unstructured.documents.elements.CompositeElement'>",
 "<class 'unstructured.documents.elements.Table'>"}
```

```
[ ]: chunks[0].metadata.orig_elements
```

```
[7]: chunks[0].metadata.orig_elements[0].to_dict()
```

```
[7]: {'type': 'UncategorizedText',
      'element_id': '8089b0b0-9217-46cf-8d41-cb455ec7164f',
      'text': '3',
      'metadata': {'coordinates': {'points': ((45.388888888888886,
          594.2222222222224),
         (45.388888888888886, 622.0000000000002),
         (100.94444444444446, 622.0000000000002),
         (100.94444444444446, 594.2222222222224)),
        'system': 'PixelSpace',
        'layout_width': 1700,
        'layout_height': 2200},
       'last_modified': '2024-12-10T22:44:55',
       'filetype': 'PPM',
       'languages': ['eng'],
       'page_number': 1}}
```

```
[ ]: chunks[1].to_dict()
```

```
[ ]: elements = chunks [3].metadata.orig_elements
     chunk_images = [el for el in elements if 'Image' in str(type(el))]
     chunk_images[0].to_dict()
```

2.2 Separte the Text, Tables and Images

```
[11]: # separate tables from texts
      tables = []
      texts = []

      for chunk in chunks:
          if "Table" in str(type(chunk)):
              tables.append(chunk)

          if "CompositeElement" in str(type((chunk))):
              texts.append(chunk)
```

```
[ ]: tables[0].to_dict()
```

```
[13]: tables[0].metadata.text_as_html
```

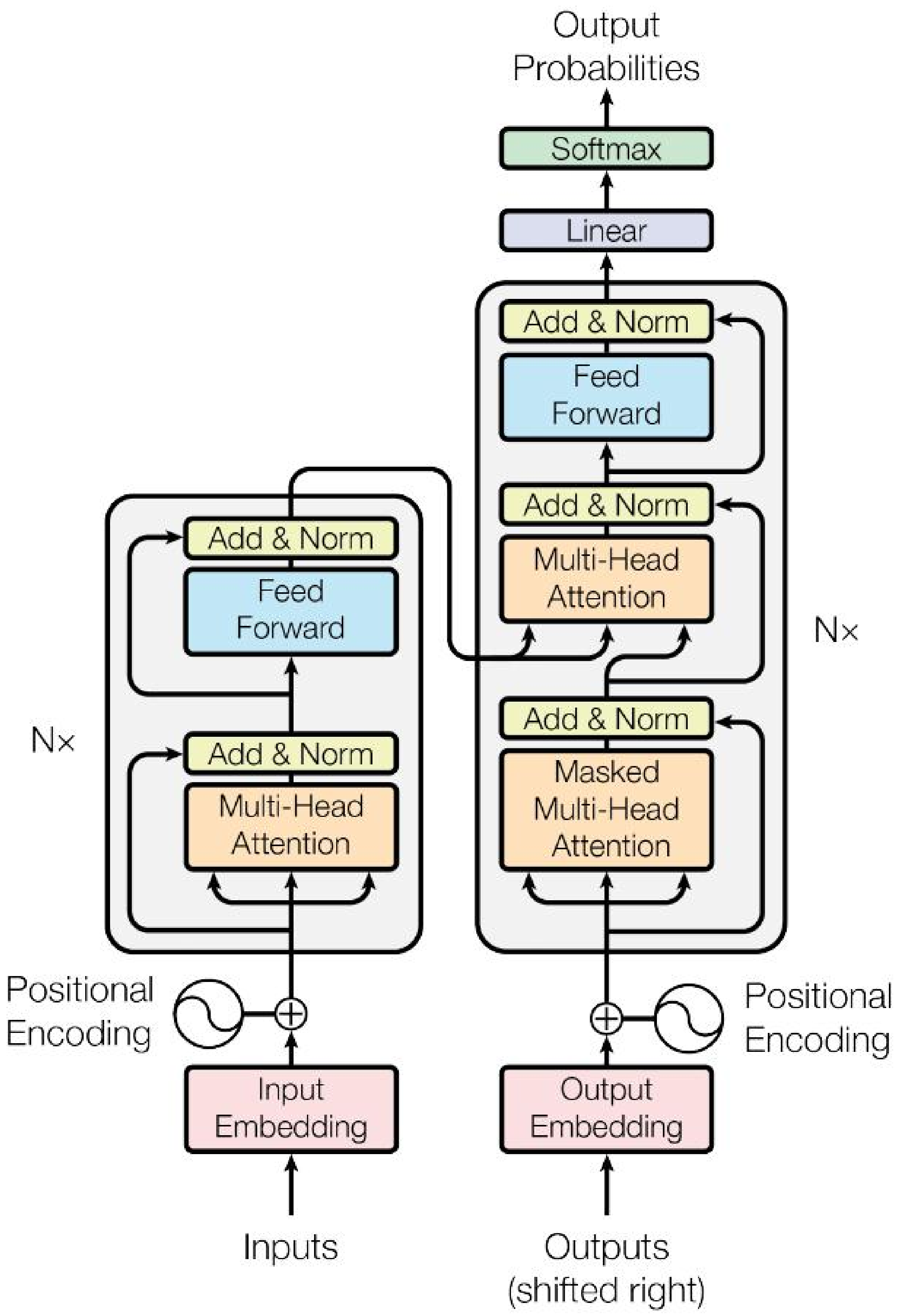
```
[13]: '<table><tr><td>Layer Type</td><td>Complexity per Layer</td><td>Sequential
      Operations</td><td>Maximum Path Length</td></tr><tr><td>Self-
      Attention</td><td>O(n? -
      d)</td><td>O(1)</td><td>O(1)</td></tr><tr><td>Recurrent</td><td>O(n- d?)</td><td
```

```
>O(n)</td><td>O(n)</td></tr><tr><td>Convolutional</td><td>O(k-n-
d?)</td><td>O(1)</td><td>O(logy(n))</td></tr><tr><td>Self-Attention
(restricted)</td><td>O(r-n-d)</td><td>ol)</td><td>O(n/r)</td></tr></table>'
```

[14]:
```
# Get the images from the CompositeElement objects
def get_images_base64(chunks):
    images_b64 = []
    for chunk in chunks:
        if "CompositeElement" in str(type(chunk)):
            chunk_els = chunk.metadata.orig_elements
            for el in chunk_els:
                if "Image" in str(type(el)):
                    images_b64.append(el.metadata.image_base64)
    return images_b64

images = get_images_base64(chunks)
```

2.3 Display Image

[15]:
```
import base64
from IPython.display import Image, display

def display_base64_image(base64_code):
    # Decode the base64 string to binary
    image_data = base64.b64decode(base64_code)
    # Display the image
    display(Image(data=image_data))

display_base64_image(images[0])
```


Output Probabilities
Softmax
Linear
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Add & Norm
Masked Multi-Head Attention
Nx
Add & Norm
Feed Forward
Add & Norm
Multi-Head Attention
Nx
Positional Encoding
Positional Encoding
Input Embedding
Output Embedding
Inputs
Outputs (shifted right)

3 Summary of the Data

3.1 Text and Table Summaries

[16]:
```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_ollama.llms import OllamaLLM

# Prompt
prompt_text = """
You are an assistant tasked with summarizing tables and text.
Give a concise summary of the table or text.

Respond only with the summary, no additionnal comment.
Do not start your message by saying "Here is a summary" or anything like that.
Just give the summary as it is.

Table or text chunk: {element}

"""
prompt = ChatPromptTemplate.from_template(prompt_text)

# Summary chain
model = OllamaLLM(temperature=0.5, model="llama3.1:8b")
summarize_chain = {"element": lambda x: x} | prompt | model | StrOutputParser()
```

[17]:
```
# Summarize text
text_summaries = summarize_chain.batch(texts, {"max_concurrency": 3})

# Summarize tables
tables_html = [table.metadata.text_as_html for table in tables]
table_summaries = summarize_chain.batch(tables_html, {"max_concurrency": 3})
```

[ ]:
```
text_summaries
```

[ ]:
```
table_summaries
```

3.2 Image Summaries

[20]:
```
from langchain_openai import OpenAI


from langchain_openai import ChatOpenAI

prompt_template = """Describe the image in detail. For context,
                  the image is part of a research paper explaining the␣
  ↪transformers
                  architecture. Be specific about graphs, such as bar plots."""
messages = [
    (
        "user",
        [
            {"type": "text", "text": prompt_template},
            {
                "type": "image_url",
                "image_url": {"url": "data:image/jpeg;base64,{image}"},
            },
        ],
    )
]

prompt = ChatPromptTemplate.from_messages(messages)

chain = prompt | ChatOpenAI(model="gpt-4o-mini") | StrOutputParser()


image_summaries = chain.batch(images)
```

[ ]:
```
image_summaries
```

3.3 Vetor Storage

3.4 Creation of vector Store

[22]:
```
import uuid
from langchain_chroma import Chroma
from langchain.storage import InMemoryStore
from langchain_core.documents import Document
from langchain_openai import OpenAIEmbeddings
from langchain.retrievers.multi_vector import MultiVectorRetriever

# The vectorstore to use to index the child chunks
vectorstore = Chroma(collection_name="multi_modal_rag",␣
  ↪embedding_function=OpenAIEmbeddings())

# The storage layer for the parent documents
store = InMemoryStore()
id_key = "doc_id"

# The retriever (empty to start)
retriever = MultiVectorRetriever(
    vectorstore=vectorstore,
    docstore=store,
    id_key=id_key,
)
```

3.5 Load the summaries and link the to the original data

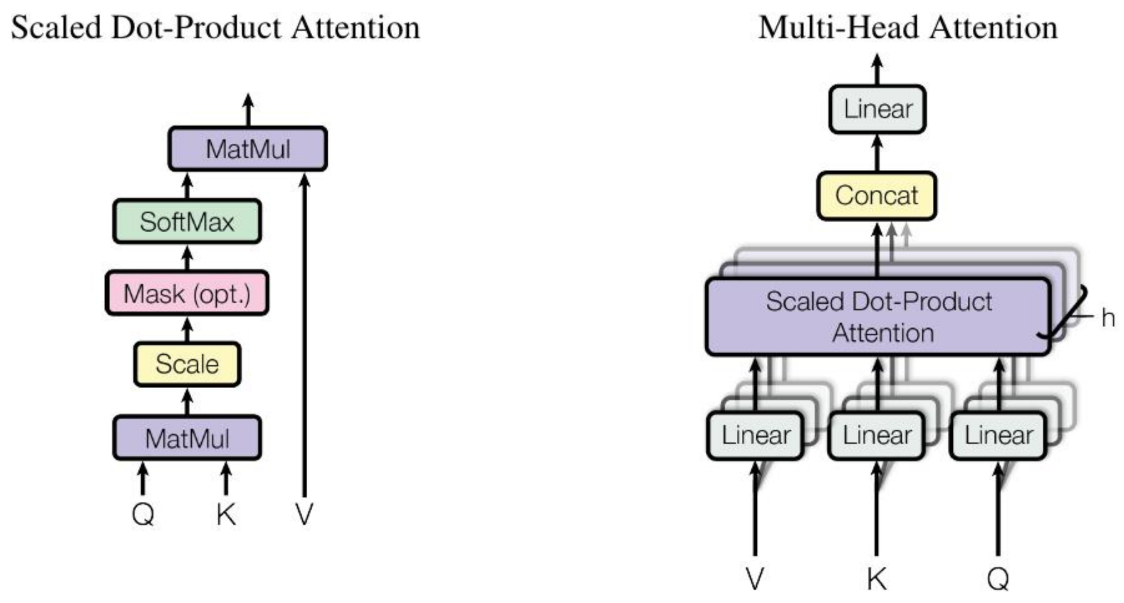
[23]:
```
# Add texts
doc_ids = [str(uuid.uuid4()) for _ in texts]
summary_texts = [
    Document(page_content=summary, metadata={id_key: doc_ids[i]}) for i,␣
  ↪summary in enumerate(text_summaries)
]
retriever.vectorstore.add_documents(summary_texts)
retriever.docstore.mset(list(zip(doc_ids, texts)))

# Add tables
table_ids = [str(uuid.uuid4()) for _ in tables]
summary_tables = [
    Document(page_content=summary, metadata={id_key: table_ids[i]}) for i,␣
  ↪summary in enumerate(table_summaries)
]
retriever.vectorstore.add_documents(summary_tables)
retriever.docstore.mset(list(zip(table_ids, tables)))

# Add image summaries
img_ids = [str(uuid.uuid4()) for _ in images]
summary_img = [
    Document(page_content=summary, metadata={id_key: img_ids[i]}) for i,␣
  ↪summary in enumerate(image_summaries)
]
retriever.vectorstore.add_documents(summary_img)
retriever.docstore.mset(list(zip(img_ids, images)))
```

3.6 Retrivel Analysis

[34]:
```
chunks = retriever.invoke(
    "what is multihead attention"
)
```

[ ]:
```
chunks
```

[ ]:
```
display_base64_image(chunks[1])
```



[ ]:
```
for chunk in chunks:
    print(chunk)
```

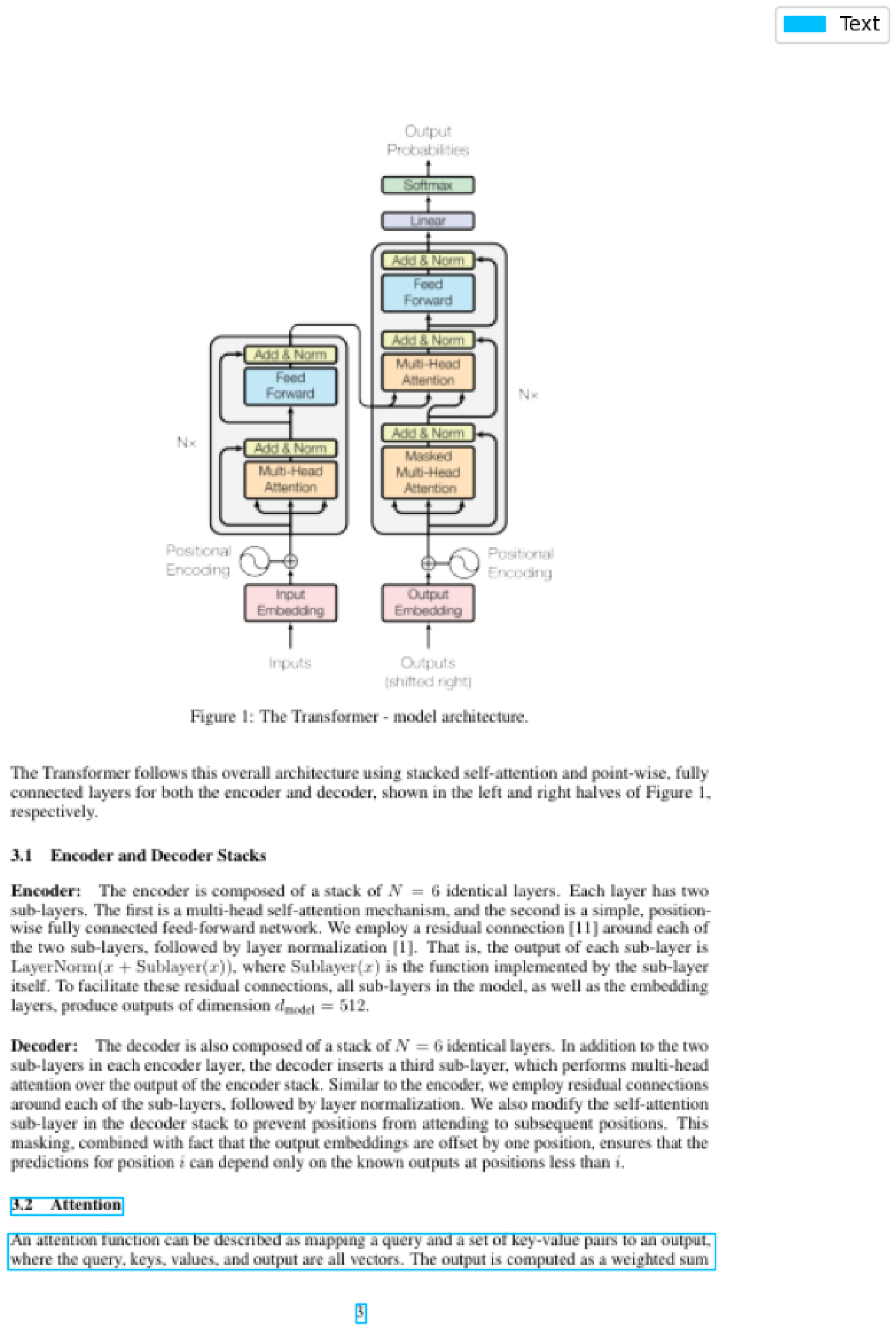
[39]:
```
for i, chunk in enumerate(chunks):
    if "CompositeElement" in str(type(chunk)):
        print("\n\nChunk", i)
        for doc in chunk.metadata.orig_elements:
            print(doc.to_dict()["type"], doc.metadata.page_number)
```

```
Chunk 0
Title 4
NarrativeText 4
NarrativeText 4
UncategorizedText 4
NarrativeText 5
NarrativeText 5
Formula 5
NarrativeText 5
NarrativeText 5
Title 5
NarrativeText 5
ListItem 5
ListItem 5
ListItem 5
```

```
Chunk 2
ListItem 12
ListItem 12
ListItem 12
ListItem 12
ListItem 12
Footer 12
Title 13
Image 13
FigureCaption 13
Header 13
Image 14
NarrativeText 14
UncategorizedText 14
Image 15
Image 15
FigureCaption 15
Header 15


Chunk 3
Title 3
NarrativeText 3
Footer 3
Image 4
Image 4
NarrativeText 4
NarrativeText 4
Title 4
NarrativeText 4
NarrativeText 4
Formula 4
NarrativeText 4
NarrativeText 4
```

[40]:
```
import fitz
import matplotlib.patches as pataches
import matplotlib.pyplot as plt
from PIL import Image

def plot_pdf_with_boxes(pdf_page, segments):
    pix = pdf_page.get_pixmap()
    pil_image = Image.frombytes('RGB', [pix.width, pix.height], pix.samples)

    fig, ax = plt.subplots(1, figsize=(10, 10))
    ax.imshow(pil_image)
    categorites = set()
    category_to_color = {
        'Title': 'orchid',
        'Image':'forestgreen',
        'Table':'tomato',
    }

    for segment in segments:
        points = segment['coordinates']['points']
        layout_width = segment["coordinates"]['layout_width']
        layout_height = segment['coordinates']['layout_height']
        scaled_points = [
            (x * pix.width / layout_width, y * pix.height / layout_height)
            for x, y in points
        ]
        box_color = category_to_color.get(segment['category'], 'deepskyblue')
        categorites.add(segment['category'])
        rect = pataches.Polygon(
            scaled_points, linewidth=1, edgecolor=box_color, facecolor='none'
        )
        ax.add_patch(rect)

    #Legend
    legend_handles = [pataches.Patch(color='deepskyblue', label='Text')]
    for category in ['Title', 'Image', 'Table']:
        if category in categorites:
            legend_handles.append(
                pataches.Patch(color=category_to_color[category],␣
 ↪label=category)
            )
    ax.axis('off')
    ax.legend(handles=legend_handles, loc='upper right')
    plt.tight_layout()
    plt.show()

def render_page(doc_list: list, page_number: int, print_text=True) -> None:
    pdf_page = fitz.open(file_path).load_page(page_number - 1)
    page_docs = [
        doc for doc in doc_list if doc.metadata.get('page_number') ==␣
 ↪page_number
    ]
    segments = [doc.metadata for doc in page_docs]
    plot_pdf_with_boxes(pdf_page=pdf_page, segments=segments)
    if print_text:
        for doc in page_docs:
            print(f'{doc.page_content}\n')
```

[43]:
```python
from langchain_core.documents import Document
def extract_page_numbers_from_chunk(chunk):
    elements = chunk.metadata.orig_elements

    page_numbers = set()
    for element in elements:
        page_numbers. add (element.metadata.page_number)
    return page_numbers


def display_chunk_pages (chunk):
    page_numbers = extract_page_numbers_from_chunk(chunk)
    docs = []
    for element in chunk.metadata.orig_elements:
        metadata = element.metadata.to_dict()
        if "Table" in str(type (element)):
            metadata ["category"] = "Table"
        elif "Image" in str(type(element) ):
            metadata ["category"] = "Image"
        else:
            metadata ["category"] = "Text"
        metadata ["page_number"] = int (element.metadata.page_number)

        docs. append (Document( page_content=element.text, metadata=metadata))

    for page_number in page_numbers:
        render_page(docs, page_number, False)


extract_page_numbers_from_chunk(chunks[3])
display_chunk_pages(chunks[3])
```



Text

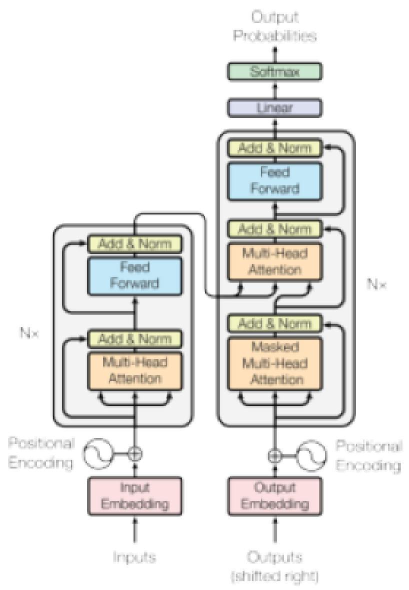


Figure 1: The Transformer - model architecture.

The Transformer follows this overall architecture using stacked self-attention and point-wise, fully connected layers for both the encoder and decoder, shown in the left and right halves of Figure 1, respectively.

3.1 Encoder and Decoder Stacks

Encoder: The encoder is composed of a stack of $N = 6$ identical layers. Each layer has two sub-layers. The first is a multi-head self-attention mechanism, and the second is a simple, position-wise fully connected feed-forward network. We employ a residual connection [11] around each of the two sub-layers, followed by layer normalization [1]. That is, the output of each sub-layer is $\mathrm{LayerNorm}(x + \mathrm{Sublayer}(x))$, where $\mathrm{Sublayer}(x)$ is the function implemented by the sub-layer itself. To facilitate these residual connections, all sub-layers in the model, as well as the embedding layers, produce outputs of dimension $d_{\text{model}} = 512$.

Decoder: The decoder is also composed of a stack of $N = 6$ identical layers. In addition to the two sub-layers in each encoder layer, the decoder inserts a third sub-layer, which performs multi-head attention over the output of the encoder stack. Similar to the encoder, we employ residual connections around each of the sub-layers, followed by layer normalization. We also modify the self-attention sub-layer in the decoder stack to prevent positions from attending to subsequent positions. This masking, combined with fact that the output embeddings are offset by one position, ensures that the predictions for position i can depend only on the known outputs at positions less than i.

3.2 Attention

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum

3

🟦 Text
🟩 Image

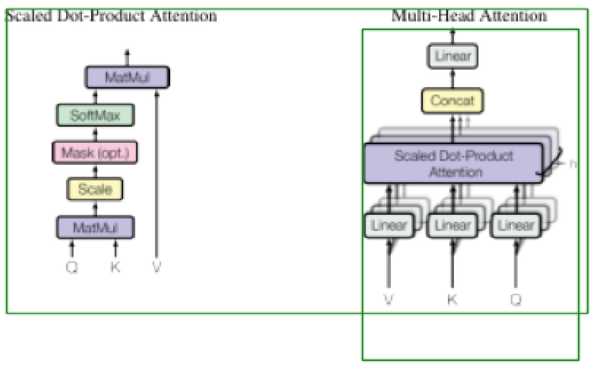


Figure 2: (left) Scaled Dot-Product Attention. (right) Multi-Head Attention consists of several attention layers running in parallel.

of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

3.2.1 Scaled Dot-Product Attention

We call our particular attention "Scaled Dot-Product Attention" (Figure 2). The input consists of queries and keys of dimension d_k, and values of dimension d_v. We compute the dot products of the query with all keys, divide each by $\sqrt{d_k}$, and apply a softmax function to obtain the weights on the values.

In practice, we compute the attention function on a set of queries simultaneously, packed together into a matrix Q. The keys and values are also packed together into matrices K and V. We compute the matrix of outputs as:

$$\mathrm{Attention}(Q, K, V) = \mathrm{softmax}(\frac{QK^T}{\sqrt{d_k}})V \qquad (1)$$

The two most commonly used attention functions are additive attention [2], and dot-product (multiplicative) attention. Dot-product attention is identical to our algorithm, except for the scaling factor of $\frac{1}{\sqrt{d_k}}$. Additive attention computes the compatibility function using a feed-forward network with a single hidden layer. While the two are similar in theoretical complexity, dot-product attention is much faster and more space-efficient in practice, since it can be implemented using highly optimized matrix multiplication code.

While for small values of d_k the two mechanisms perform similarly, additive attention outperforms dot product attention without scaling for larger values of d_k [3]. We suspect that for large values of d_k, the dot products grow large in magnitude, pushing the softmax function into regions where it has extremely small gradients [4]. To counteract this effect, we scale the dot products by $\frac{1}{\sqrt{d_k}}$.

3.2.2 Multi-Head Attention

Instead of performing a single attention function with d_{model}-dimensional keys, values and queries, we found it beneficial to linearly project the queries, keys and values h times with different, learned linear projections to d_k, d_k and d_v dimensions, respectively. On each of these projected versions of queries, keys and values we then perform the attention function in parallel, yielding d_v-dimensional

[4]To illustrate why the dot products get large, assume that the components of q and k are independent random variables with mean 0 and variance 1. Then their dot product, $q \cdot k = \sum_{i=1}^{d_k} q_i k_i$, has mean 0 and variance d_k.

4 RAG Pipeline

[44]:
```python
from langchain_core.runnables import RunnablePassthrough, RunnableLambda
from langchain_core.messages import SystemMessage, HumanMessage
from langchain_openai import ChatOpenAI
from base64 import b64decode


def parse_docs(docs):
    """Split base64-encoded images and texts"""
    b64 = []
    text = []
    for doc in docs:
        try:
            b64decode(doc)
            b64.append(doc)
        except Exception as e:
            text.append(doc)
    return {"images": b64, "texts": text}


def build_prompt(kwargs):

    docs_by_type = kwargs["context"]
    user_question = kwargs["question"]

    context_text = ""
    if len(docs_by_type["texts"]) > 0:
        for text_element in docs_by_type["texts"]:
            context_text += text_element.text

    # construct prompt with context (including images)
    prompt_template = f"""
    Answer the question based only on the following context, which can include␣
  ↪text, tables, and the below image.
    Context: {context_text}
    Question: {user_question}
    """

    prompt_content = [{"type": "text", "text": prompt_template}]

    if len(docs_by_type["images"]) > 0:
        for image in docs_by_type["images"]:
            prompt_content.append(
                {
                    "type": "image_url",
                    "image_url": {"url": f"data:image/jpeg;base64,{image}"},
                }
            )

    return ChatPromptTemplate.from_messages(
        [
            HumanMessage(content=prompt_content),
        ]
    )


chain = (
    {
        "context": retriever | RunnableLambda(parse_docs),
        "question": RunnablePassthrough(),
    }
    | RunnableLambda(build_prompt)
    | ChatOpenAI(model="gpt-4o-mini")
    | StrOutputParser()
)

chain_with_sources = {
    "context": retriever | RunnableLambda(parse_docs),
    "question": RunnablePassthrough(),
} | RunnablePassthrough().assign(
    response=(
        RunnableLambda(build_prompt)
        | ChatOpenAI(model="gpt-4o-mini")
        | StrOutputParser()
    )
)
```

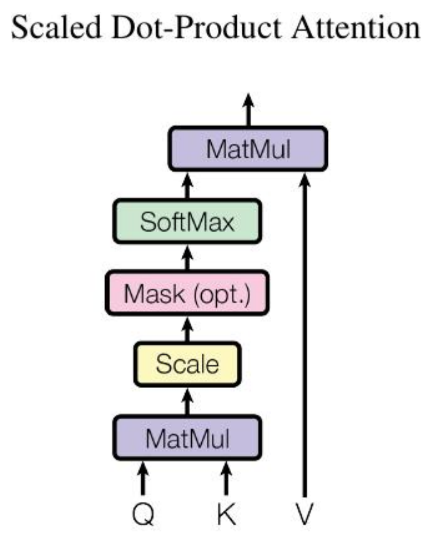
[45]:
```
response = chain.invoke(
    "What is the attention mechanism?"
)

print(response)
```

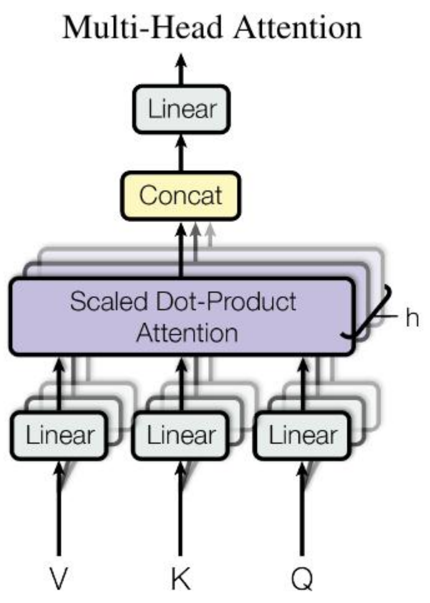
```
The attention mechanism, specifically the "Scaled Dot-Product Attention," is a
method that computes a weighted sum of values based on the compatibility of
queries with keys. The key steps involved are:

1. **Input Matrices**: It takes three matrices as input-queries (Q), keys (K),
and values (V), all of which are vectors that represent different aspects of the
input data.

2. **Dot Products**: The dot products of the queries and keys are computed,
scaled by the square root of the dimension of the keys (√dk).
```

```
3. **Softmax**: A softmax function is applied to the scaled dot products to
obtain the attention weights, which indicate the importance of each value based
on its corresponding key.

4. **Weighted Sum**: Finally, these weights are used to compute a weighted sum
of the values (V), resulting in the output of the attention mechanism.

This attention mechanism allows the model to focus on relevant parts of the
input sequence for each output element, enabling it to capture relationships and
dependencies effectively.

Additionally, the multi-head attention expands this concept by running several
attention functions in parallel to gather information from different
representation subspaces.
```

[47]:
```python
response = chain_with_sources.invoke(
    "What is multihead?"
)

print("Response:", response['response'])

#Context

# print("\n\nContext:")
# for text in response['context']['texts']:
#     print(text.text)
#     print("Page number: ", text.metadata.page_number)
#     print("\n" + "-"*50 + "\n")

for image in response['context']['images']:
    display_base64_image(image)
```

```
Response: Multi-head attention is a mechanism that extends the standard
attention function by using multiple attention heads in parallel. Instead of
computing a single set of attention scores, multi-head attention projects the
input queries, keys, and values into multiple lower-dimensional spaces. Each
attention head then performs the attention function independently, allowing the
model to capture different aspects of the input data simultaneously.

The outputs from all the attention heads are concatenated and linearly
transformed to produce the final output. This approach enables the model to
attend to information from different representation subspaces at various
positions, enhancing its ability to learn complex patterns within the data.
```


Scaled Dot-Product Attention
MatMul
SoftMax
Mask (opt.)
Scale
MatMul
Q
K
V


Multi-Head Attention
Linear
Concat
Scaled Dot-Product Attention
h
Linear
Linear
Linear
V
K
Q

5 THANK YOU!