AI Intro Basics #3

Large Language Models, Transformers, RAG

Roland Potthast, Stefanie Hollborn and Jan Keller

EUMETNET

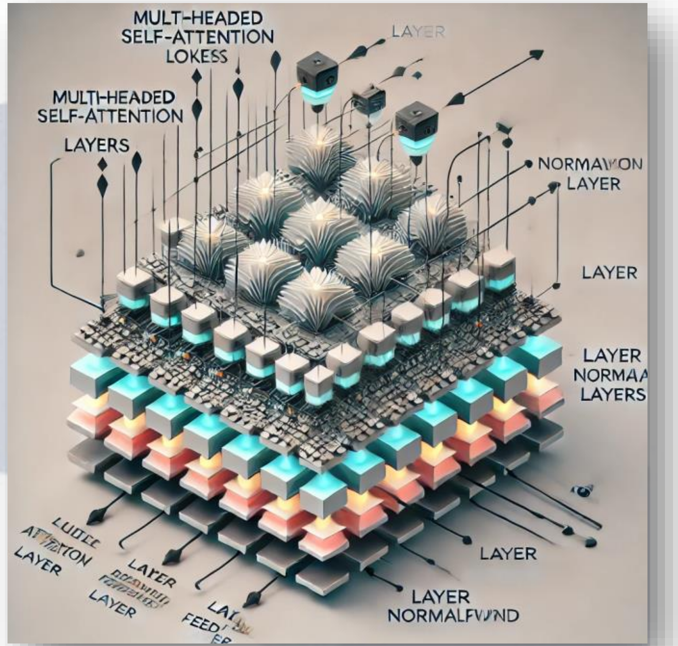
E-AI Basic Tutorials

1. Tutorial E-AI Basics 1: September 16, 2024, 11-12 CEST
2. Intro, Environment, First Example
3. 1.1 Basic Ideas of AI Techniques (20') [SH]
4. 1.2 Work Environment (20') [RP]
5. 1.3 First Example for AI - hands-on (20') [JK]

6. Tutorial E-AI Basics 2: September 18, 2024, 13-14 CEST
7. Dynamics, Downscaling, Data Assimilation Examples
8. 2.1 Dynamic Prediction by a Graph NN (20') [RP]
9. 2.2 Data Recovery/Denoising via Encoder-Decoder (20') [SH]
10. 2.3 AI for Data Assimilation (20') [JK]

11. Tutorial E-AI Basics 3: September 23, 2024, 11-12 CEST
12. LLM Use, Transformer Example, RAG
13. 3.1 Intro to LLM Use and APIs (20') [RP]
14. 3.2 Transformer for Language and Images (20') [JK]
15. 3.3 LLM Retrieval Augmented Generation (RAG) (20') [SH]

3.1



AI generated Images,
© Roland Potthast 2024

Large Language Models (LLMs)

- LLMs are models based on deep learning techniques, specifically neural networks, to process and understand patterns in the text.
- LLMs are trained on massive datasets that include books, websites, and other text sources.
- Through training, LLMs learn how words, sentences, and paragraphs relate to each other, enabling them to understand and generate coherent text.
- After training, LLMs can generate responses, summarize information, and assist with various language tasks based on user input.

Large Language Models, a Snapshot

#	Model Name	Comment
1	GPT-4	Last version from OpenAI, advanced text generation capabilities.
2	GPT-3.5	Widely used earlier version with strong performance.
3	BERT	Bidirectional model for natural language understanding tasks.
4	DistilBERT	Smaller, faster version of BERT, maintaining good performance.
5	T5	Text-to-text model, flexible for various NLP tasks.
6	RoBERTa	Optimized version of BERT, better performance on benchmarks.
7	XLNet	Autoregressive model that considers word sequence permutations.
8	ALBERT	Lightweight version of BERT with reduced size and comparable performance.
9	Megatron-Turing NLG	Massive model with strong text generation capabilities.
10	Claude	Focused on safety and alignment in language generation.
11	LLaMA	Research-focused family of models with strong performance.
12	OPT	General-purpose language models developed by Meta.
13	BLOOM	Multilingual model developed through collaborative research.
14	Cohere Command R	Designed for retrieval-augmented generation tasks.
15	ChatGPT	Conversational agent tuned for dialogue, based on GPT-3.5 and GPT-4.

Using LLMs for various tasks becomes a fantastic new tool!

The LLM can help you to access information

give me a short python code accessing open sources to plot the temperatures in Frankfurt over the past 30 days

Here's a Python script that uses the `requests` library to fetch temperature data from the Open-Meteo API, which provides open access to historical weather data, and plots it using `matplotlib`.

Step 1: Install Required Libraries

bash　Code kopieren

```
pip install requests matplotlib
```

The LLM generates code and explains or documents code!

Get Python Code from your LLM

Step 2: Python Code

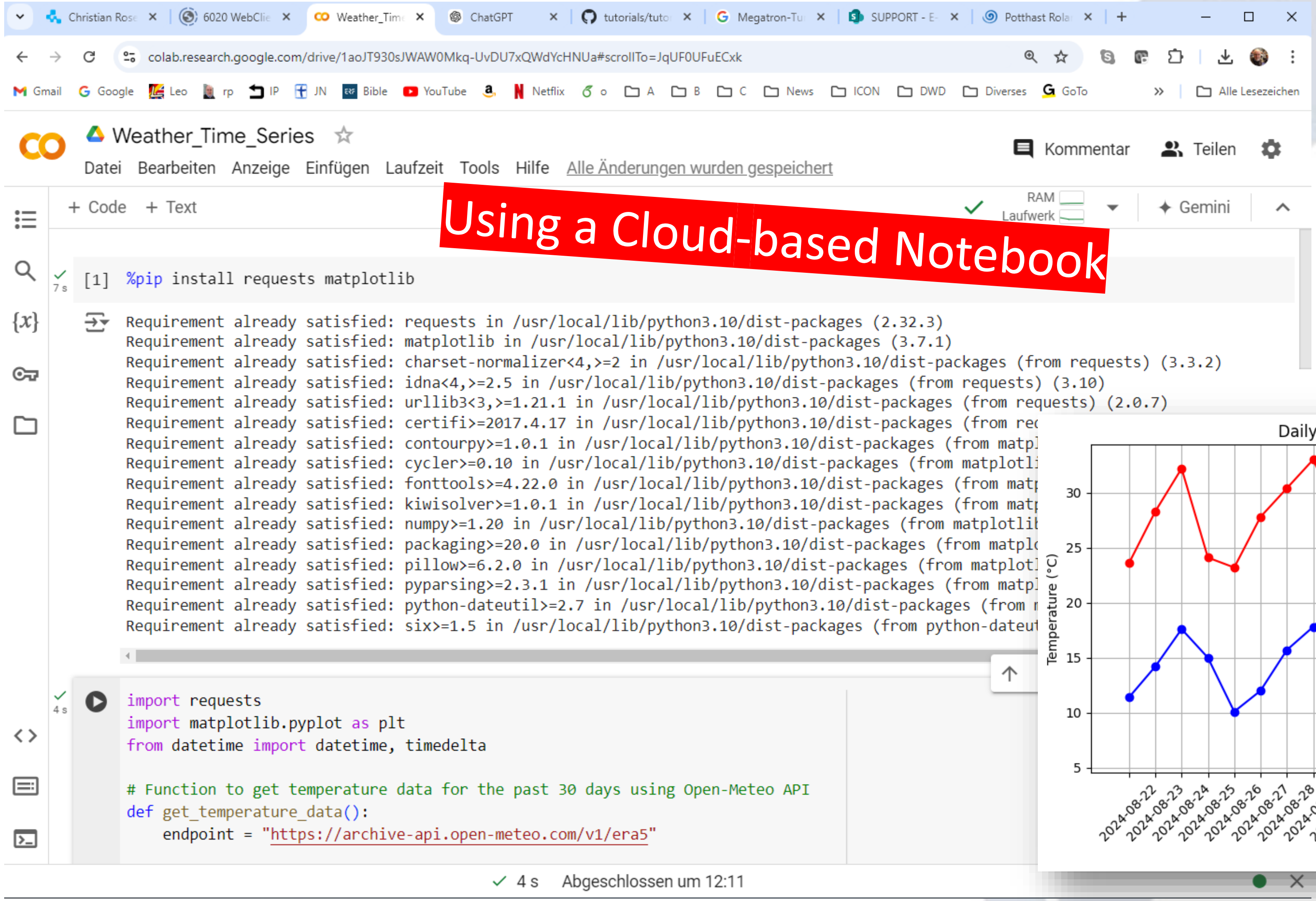
python Code kopieren

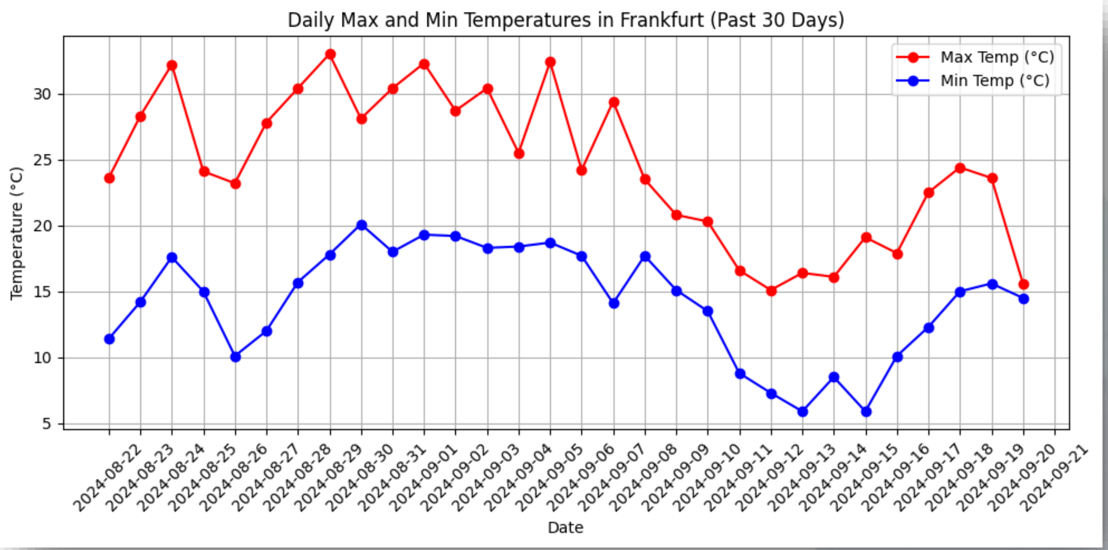
```python
import requests
import matplotlib.pyplot as plt
from datetime import datetime, timedelta

# Function to get temperature data for the past 30 days using Open-Meteo API
def get_temperature_data():
    endpoint = "https://archive-api.open-meteo.com/v1/era5"

    # Set the latitude and longitude for Frankfurt
    params = {
        "latitude": 50.1109,  # Latitude for Frankfurt
        "longitude": 8.6821,  # Longitude for Frankfurt
        "start_date": (datetime.now() - timedelta(days=30)).strftime("%Y-%m-%d"),
        "end_date": datetime.now().strftime("%Y-%m-%d"),
        "temperature_unit": "celsius",
        "daily": "temperature_2m_max,temperature_2m_min",
        "timezone": "Europe/Berlin"
    }

    # Fetch the data
    response = requests.get(endpoint, params=params)
    data = response.json()
```

```python
    # Extract relevant data
    dates = data['daily']['time']
    temp_max = data['daily']['temperature_2m_max']
    temp_min = data['daily']['temperature_2m_min']

    return dates, temp_max, temp_min

# Plotting function
def plot_temperatures(dates, temp_max, temp_min):
    plt.figure(figsize=(10, 5))

    # Plot max and min temperatures
    plt.plot(dates, temp_max, label="Max Temp (°C)", color="red", marker='o')
    plt.plot(dates, temp_min, label="Min Temp (°C)", color="blue", marker='o')

    # Formatting the plot
    plt.title("Daily Max and Min Temperatures in Frankfurt (Past 30 Days)")
    plt.xlabel("Date")
    plt.ylabel("Temperature (°C)")
    plt.xticks(rotation=45)
    plt.grid(True)
    plt.legend()

    # Show plot
    plt.tight_layout()
    plt.show()

# Fetch the temperature data
dates, temp_max, temp_min = get_temperature_data()

# Plot the data
plot_temperatures(dates, temp_max, temp_min)
```

The LLM generates code and explains or documents code!

Code can be executed directly on the Cloud or within the LLM environment

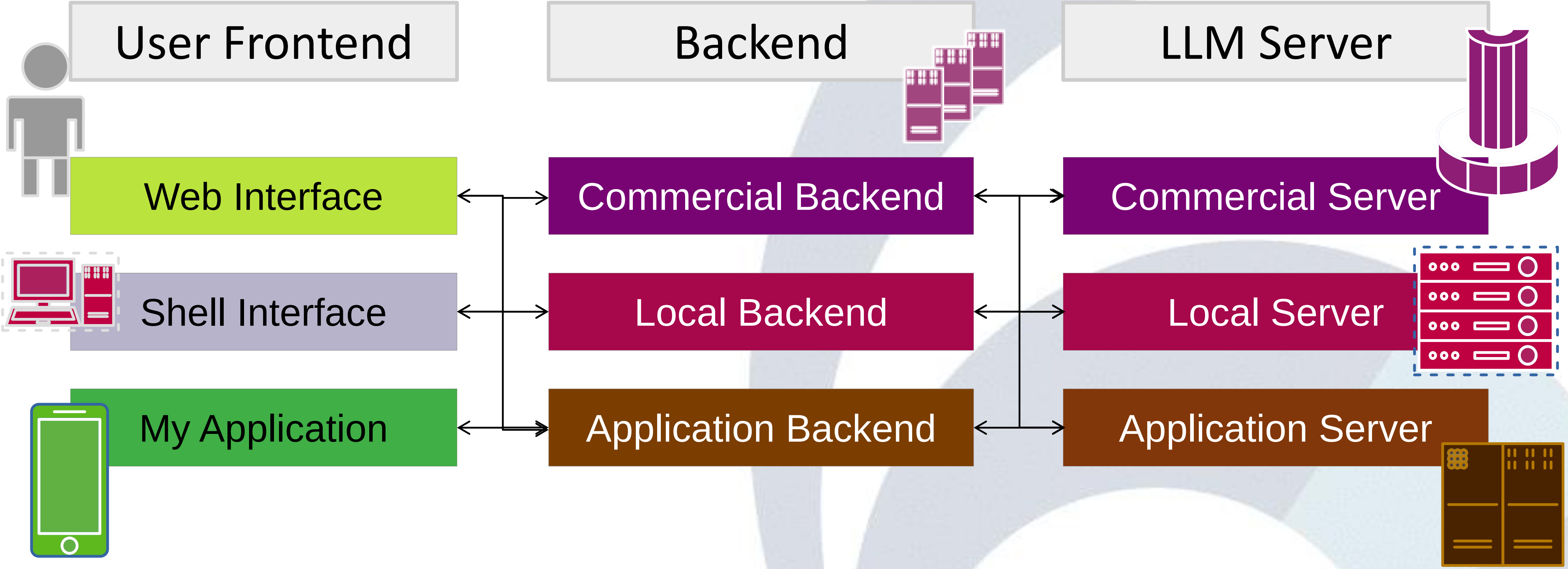




Within 2 minutes we get the result

LLM/GPT Usage and Access

Modular Options



Backend API – how it works

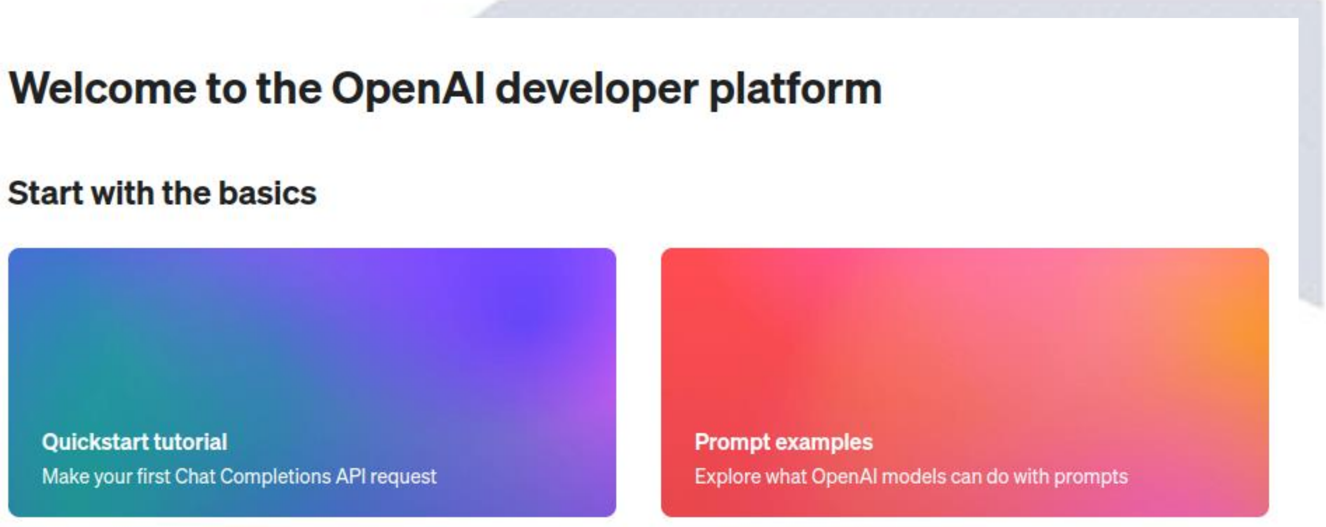
- **Request**: Your application sends an HTTP request (usually a POST request) to the API endpoint. The request typically includes the input text or prompt you want the LLM to process.
- **Processing**: The backend API tokenizes the input text, prepares it for the LLM, and sends the tokens to the model.
- **Model Inference**: The LLM processes the tokens, generates predictions, and produces an output (usually text).
- **Response**: The API receives the LLM's output, converts it back to human-readable text, and sends it as an HTTP response to your application.

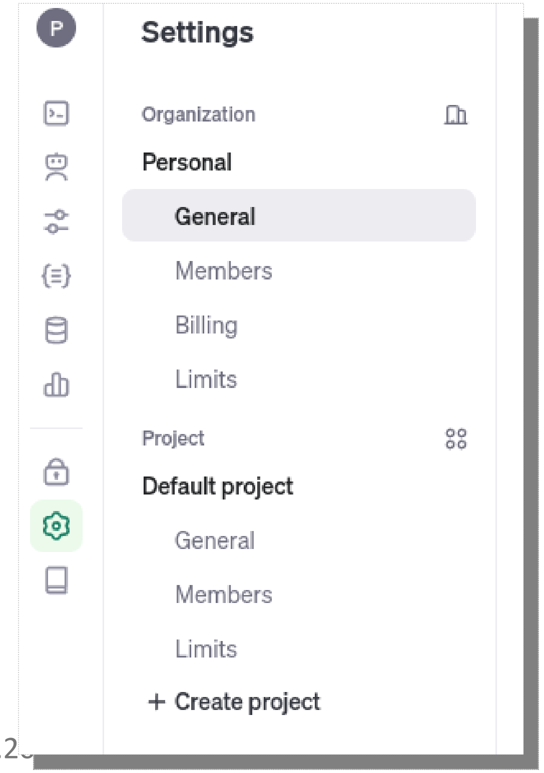
Commercial Backend Example

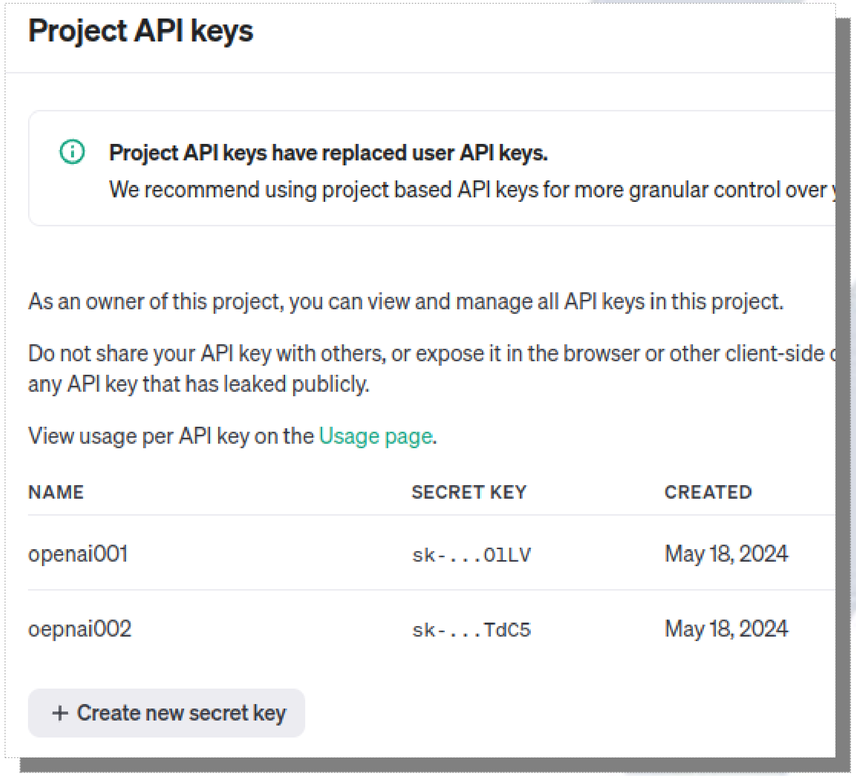
1. Sign in at openai.com

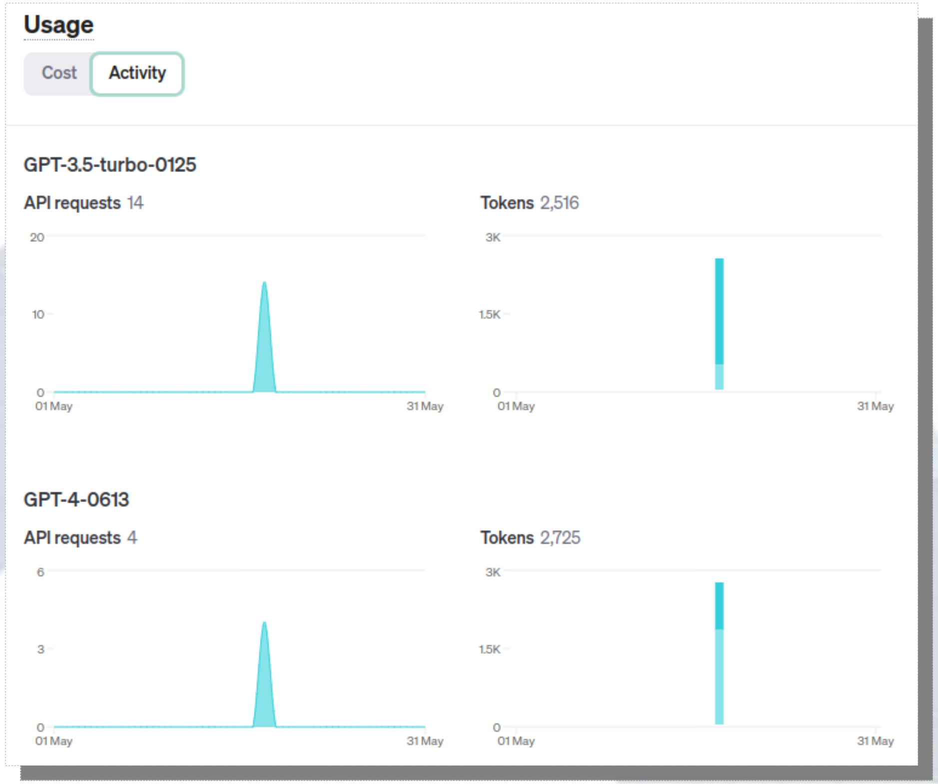
2. Provide Payment Methods (Credit card)

3. generate an openai_api_key









Commercial Backend Example

4. Put your OPENAI_API_KEY into the .env file

.env_example — Kate

```
# Once you add your API key below, make sure to not share it with anyone!
# The API key should remain private.

OPENAI_API_KEY=sk-proj-Titbv53s690xx2dc7aba74ddC5
```

5. pip install python-dotenv

6. Now the key is available in your python notebook by load_dotenv()

```python
# to load the openai_api_key we load the .env environment
from dotenv import load_dotenv
load_dotenv()
```

6. Now you can connect to ChatGPT 3.5 or 4 easily by the openai package.

```python
import os as os
from openai import OpenAI

client = OpenAI(
  api_key=os.environ.get("OPENAI_API_KEY"),
)
```

Commercial Backend Example

```python
completion = client.chat.completions.create(
  model="gpt-3.5-turbo",
  messages=[
    {"role": "user", "content": "Tell me about the ICON weather model!"}
  ])

print(completion.choices[0].message.content)
```

Here the Request

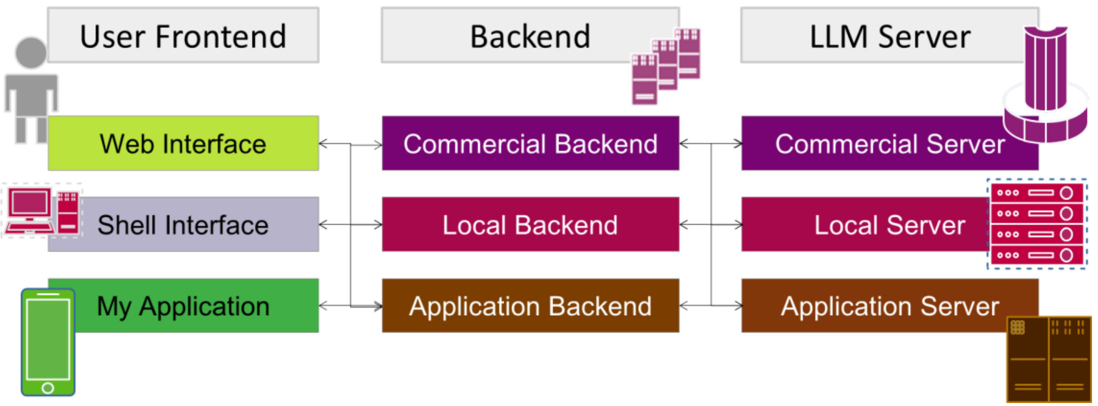
Answer in the completion structure

```
The ICON weather model is a numerical weather prediction model developed by the German Meteorolog
ical Service (Deutscher Wetterdienst). It is a high-resolution global model that provides forecas
ts for various weather parameters such as temperature, precipitation, wind, and pressure.

The ICON model uses a combination of atmospheric physical equations, data assimilation technique
s, and observations from satellites and ground-based weather stations to generate forecast data.
It is capable of producing forecasts with a horizontal resolution of up to 13 kilometers, making
it one of the highest-resolution global models available.

The ICON model is used by meteorologists and weather forecasters around the world to provide accu
rate and timely weather forecasts for various applications, including aviation, agriculture, and
disaster preparedness. It is known for its reliable predictions and ability to capture complex we
ather patterns and phenomena.

Overall, the ICON weather model plays a crucial role in improving our understanding of the atmosp
here and providing valuable information for decision-making in various sectors that are impacted
by weather conditions.
```

LLM in my Institution

LLM on your own linux machine or linux server

- Installing my own **open source LLM** server
  - Full Privacy
  - Full Control
  - Pretrained



Large language model

Llama 2: open source, free for research and commercial use

We're unlocking the power of these large language models. Our latest version of Llama – Llama 2 – is now accessible to individuals, creators, researchers, and businesses so they can experiment, innovate, and scale their ideas responsibly.

Download the model

Ollama

Ollama

LLM on your own linux machine
or linux server

Get up and running with large language models.

Run Llama 3, Phi 3, Mistral, Gemma, and other models. Customize and create your own.

Ollama is a framework to

- install
- use and
- fine-tune

large language models locally

You can install it by typing

> pip install Ollama

In your local virtual environment.

Ollama list of LLM models available

Ollama supports a list of models available on ollama.com/library

Here are some example models that can be downloaded:

Model	Parameters	Size	Download
Llama 3	8B	4.7GB	`ollama run llama3`
Llama 3	70B	40GB	`ollama run llama3:70b`
Phi-3	3.8B	2.3GB	`ollama run phi3`
Mistral	7B	4.1GB	`ollama run mistral`
Neural Chat	7B	4.1GB	`ollama run neural-chat`
Starling	7B	4.1GB	`ollama run starling-lm`
Code Llama	7B	3.8GB	`ollama run codellama`
Llama 2 Uncensored	7B	3.8GB	`ollama run llama2-uncensored`
LLaVA	7B	4.5GB	`ollama run llava`
Gemma	2B	1.4GB	`ollama run gemma:2b`
Gemma	7B	4.8GB	`ollama run gemma:7b`
Solar	10.7B	6.1GB	`ollama run solar`

LLM on your own linux machine or linux server

Now use Ollama in your python notebook by importing Ollama

Local access to your LLM

In [1]:
```python
import ollama
```

In [2]:
```python
response = ollama.chat(model='mistral',messages=[{'role': 'user', 'content':
                    'tell me a joke involving mathematics'}])
print(response['message']['content'])
```

Out [2]:
```
 Why was the equal sign so humble?

Because it knew it wasn't less than or greater than anyone else! (I know, math jokes
```

In [3]:
```python
response = ollama.chat(model='llama2',
            messages=[{'role': 'user', 'content': 'tell me a joke involving meteorology'}])
print(response['message']['content'])
```

Out [3]:
```
Why did the meteorologist break up with his girlfriend?

She was always clouding his judgment!
```

Arguments for Private and Open-Source LLM Solutions

Criteria	Commercial LLM API	Open Source Pre-Trained LLM
Expertise	Vendor provides expertise and support	Requires in-house expertise for deployment and tuning
R&D Budget	Initial costs (subscription fees); long-term may rise with usage	Low initial cost; long-term costs for infrastructure and maintenance
Time to Market	Faster implementation (ready-to-use solutions)	Slower setup; fine-tuning and deployment take time
Control over Model Quality	Limited control; dependent on vendor updates	Full control; can customize and optimize as needed
Data Privacy	Potential data sharing concerns with vendor	Full control; data remains in-house
Inference Speed	Generally optimized for speed by provider	Speed varies; may need optimization for production use
Cost Efficiency at Scale	Higher long-term costs with increased usage	More cost-effective at scale; mostly fixed costs

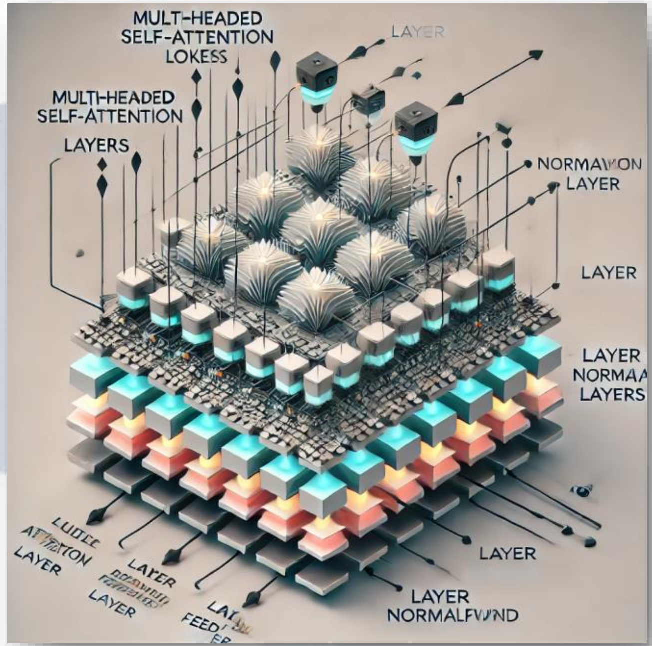
E-AI Basic Tutorials

1. Tutorial E-AI Basics 1: September 16, 2024, 11-12 CEST
2. Intro, Environment, First Example
3. 1.1 Basic Ideas of AI Techniques (20') [SH]
4. 1.2 Work Environment (20') [RP]
5. 1.3 First Example for AI - hands-on (20') [JK]

6. Tutorial E-AI Basics 2: September 18, 2024, 13-14 CEST
7. Dynamics, Downscaling, Data Assimilation Examples
8. 2.1 Dynamic Prediction by a Graph NN (20') [RP]
9. 2.2 Data Recovery/Denoising via Encoder-Decoder (20') [SH]
10. 2.3 AI for Data Assimilation (20') [JK]

11. Tutorial E-AI Basics 3: September 23, 2024, 11-12 CEST
12. LLM Use, Transformer Example, RAG
13. 3.1 Intro to LLM Use and APIs (20') [RP]
14. 3.2 Transformer for Language and Images (20') [JK]
15. 3.3 LLM Retrieval Augmented Generation (RAG) (20') [SH]

3.2

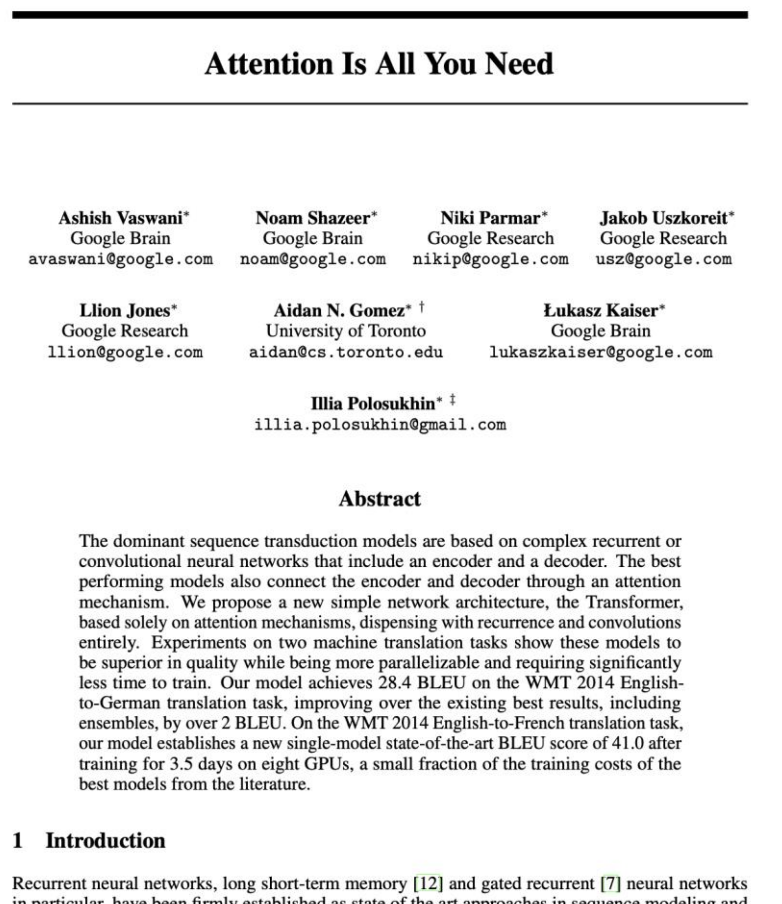


AI generated Images,
© Roland Potthast 2024

Transformer architectures

- Transformers process all tokens in a sequence simultaneously, which significantly speeds up training and inference.
- Transformers leverage the self-attention mechanism to capture relationships between (distant) tokens.
- Transformers can be easily scaled by adjusting the number of layers and attention heads, allowing for models to grow in capacity as needed.

Vaswani et al., 2017



Attention Is All You Need

Ashish Vaswani* Google Brain avaswani@google.com

Noam Shazeer* Google Brain noam@google.com

Niki Parmar* Google Research nikip@google.com

Jakob Uszkoreit* Google Research usz@google.com

Llion Jones* Google Research llion@google.com

Aidan N. Gomez* † University of Toronto aidan@cs.toronto.edu

Łukasz Kaiser* Google Brain lukaszkaiser@google.com

Illia Polosukhin* ‡ illia.polosukhin@gmail.com

Abstract

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely. Experiments on two machine translation tasks show these models to be superior in quality while being more parallelizable and requiring significantly less time to train. Our model achieves 28.4 BLEU on the WMT 2014 English-to-German translation task, improving over the existing best results, including ensembles, by over 2 BLEU. On the WMT 2014 English-to-French translation task, our model establishes a new single-model state-of-the-art BLEU score of 41.0 after training for 3.5 days on eight GPUs, a small fraction of the training costs of the best models from the literature.

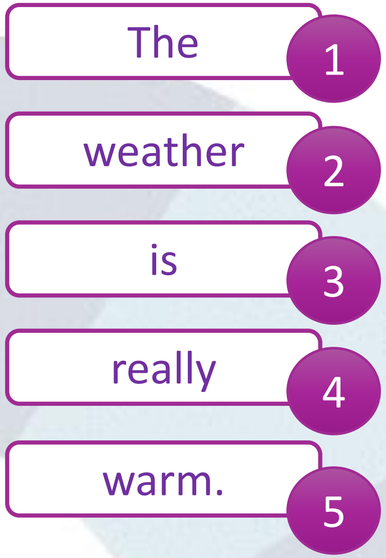
1 Introduction

Recurrent neural networks, long short-term memory [12] and gated recurrent [7] neural networks

Attention Mechanism

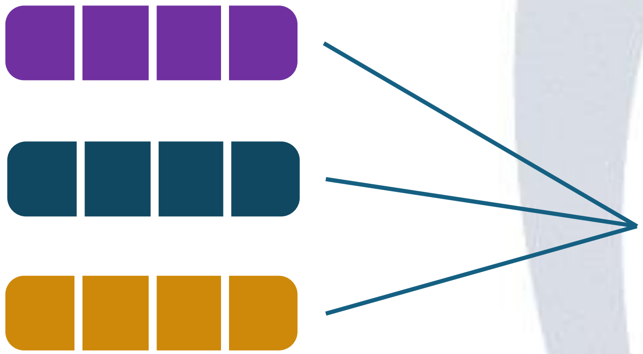
- The attention mechanism allows the model to focus on specific parts of the input sequence when producing each output. It dynamically weighs the importance of different tokens.

- First, tokenize the input: “The weather is really warm.” and create embeddings



Attention Mechanism

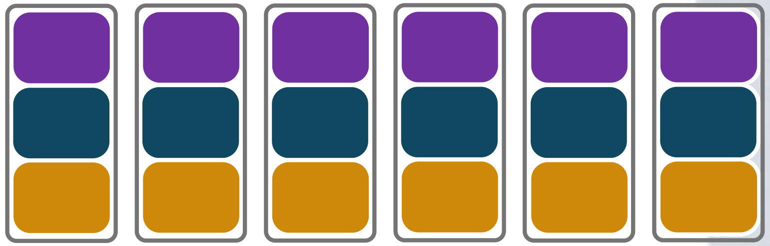
- Query, Key, Value:
  - Queries Q — Represent the current token needing information.
  - Keys K — Represent all tokens in the sequence.
  - Values V — The actual information associated with each token.



Vectors and Matrices (embeddings)

Attention Mechanism

- Attention Scores are calculated using the dot product of Q and K. Higher scores indicate more relevant tokens.
- The scores are then normalized using softmax to create a probability distribution, and the values are summed accordingly.



Multi-head attention

Self-Attention Mechanism

- Specific type of attention to create contextual embeddings
- Each token considers other tokens in the sequence, allowing to capture relationships regardless of their distance in the text.
- Effective for long sequences, as each token can reference all others in a single computation step
- Generate a new representation for each token based on its attention to all other tokens.

Text generation example

- Generate the data to train the model on

```python
# Example Dataset
sentences = [
    "The sky is clear, and the sun is shining brightly.",
    "Tomorrow's forecast predicts a chance of thunderstorms.",
    "The temperature is expected to drop below freezing tonight.",
    "The weather is perfect for a day at the beach.",
    "Strong winds are causing power outages across the region.",
    "A hurricane is approaching the coastline, and residents are advised
    "There is a severe weather warning in effect until midnight.",
    "The sunset painted the sky with hues of orange and pink.",
    "The heatwave has broken temperature records this year.",
    "It's a cloudy day with a chance of light showers in the afternoon."
    "The weather has been unpredictable lately, changing from sunny to r
    "The spring blossoms are early this year due to mild weather.",
    "People are enjoying outdoor concerts as the nights get warmer.",
    "A warm breeze carried the scent of blooming flowers through the air
    "A heat advisory has been issued for the upcoming days.",
    "The local weather station reported record high temperatures today."
    "A cool breeze is a welcome relief from the afternoon sun.",
    "Unexpected weather changes have become a common theme this year.",
    "The windchill factor makes it feel much colder outside.",
]
```

```python
# Build vocabulary mapping words to IDs
def build_vocab(sentences):
    vocab = {"<pad>": 0, "<unk>": 1}
    index = 2
    for sentence in sentences:
        for word in sentence.lower().split():
            if word not in vocab:
                vocab[word] = index
                index += 1
    return vocab

vocab = build_vocab(sentences)
vocab_size = len(vocab)
padding_idx = vocab["<pad>"]
```

Text generation example

- Tokenize the input and set up the data set

```python
# Tokenization function
def tokenize_sentence(sentence, vocab):
    return [vocab.get(word.lower(), vocab["<unk>"]) for word in sentence.split()]

# Padding function
def pad_sequence(seq, max_len, pad_value=0):
    return seq + [pad_value] * (max_len - len(seq)) if len(seq) < max_len else seq[:max_len]

# Dataset class
class TextDataset(Dataset):
    def __init__(self, sentences, vocab, max_len):
        self.max_len = max_len
        self.vocab = vocab
        self.data = [tokenize_sentence(sentence, vocab) for sentence in sentences]

    def __len__(self):
        return len(self.data)

    def __getitem__(self, idx):
        seq = self.data[idx]
        x = seq[:-1]  # Input sequence
        y = seq[1:]   # Target sequence (shifted by one)
        x_padded = pad_sequence(x, self.max_len)
        y_padded = pad_sequence(y, self.max_len)
        return torch.tensor(x_padded, dtype=torch.long), torch.tensor(y_padded, dtype=torch.long)
```

```python
# Dataset and DataLoader
dataset = TextDataset(sentences, vocab, max_len)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
```

EUMETNET

Text generation example

- The transformer model

```
class TransformerModel(nn.Module):
    def __init__(self, vocab_size, d_model, nhead, num_layers, dim_feedforward, max_len, padding_idx):
        super(TransformerModel, self).__init__()
        self.embedding = nn.Embedding(vocab_size, d_model, padding_idx=padding_idx)
        self.pos_encoder = PositionalEncoding(d_model, max_len)
        encoder_layer = nn.TransformerEncoderLayer(d_model, nhead, dim_feedforward)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers)
        self.fc_out = nn.Linear(d_model, vocab_size)
        self.d_model = d_model

    def forward(self, src):
        src_mask = self.generate_square_subsequent_mask(src.size(1)).to(src.device)
        src_pad_mask = (src == padding_idx).to(src.device)
        src = self.embedding(src) * math.sqrt(self.d_model)
        src = self.pos_encoder(src)
        output = self.transformer_encoder(src.transpose(0, 1), mask=src_mask, src_key_padding_mask=src_pad_mask)
        output = self.fc_out(output)
        return output.transpose(0, 1)

    def generate_square_subsequent_mask(self, sz):
        mask = torch.triu(torch.ones(sz, sz) * float('-inf'), diagonal=1)
        return mask
```

Attention!

Text generation example

- Training loop

```
# Hyperparameters
max_len = 15
batch_size = 2
d_model = 64
nhead = 2
num_layers = 2
dim_feedforward = 128
num_epochs = 200

# Dataset and DataLoader
dataset = TextDataset(sentences, vocab, max_len)
dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

# Initialize model, criterion, and optimizer
model = TransformerModel(vocab_size, d_model, nhead, num_layers, dim_feedforward, max_len, padding_idx)
criterion = nn.CrossEntropyLoss(ignore_index=padding_idx)
optimizer = optim.Adam(model.parameters(), lr=0.0005)
```

```
# Training loop
for epoch in range(1, num_epochs + 1):
    model.train()
    total_loss = 0
    for x_batch, y_batch in dataloader:
        optimizer.zero_grad()
        output = model(x_batch)
        output = output.reshape(-1, vocab_size)
        y_batch = y_batch.view(-1)
        loss = criterion(output, y_batch)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    avg_loss = total_loss / len(dataloader)
    if (epoch%5==0):
        print(f"Epoch [{epoch}/{num_epochs}], Loss: {avg_loss:.4f}")
```

```
Epoch [5/200], Loss: 4.2328
Epoch [10/200], Loss: 3.4469
Epoch [15/200], Loss: 2.7120
Epoch [20/200], Loss: 2.0709
Epoch [25/200], Loss: 1.5228
Epoch [30/200], Loss: 1.1387
```

```
Epoch [125/200], Loss: 0.1806
Epoch [130/200], Loss: 0.1879
Epoch [135/200], Loss: 0.2028
Epoch [140/200], Loss: 0.1772
Epoch [145/200], Loss: 0.1941
Epoch [150/200], Loss: 0.1680
Epoch [155/200], Loss: 0.1955
Epoch [160/200], Loss: 0.1798
Epoch [165/200], Loss: 0.1839
Epoch [170/200], Loss: 0.1731
Epoch [175/200], Loss: 0.1766
Epoch [180/200], Loss: 0.1682
Epoch [185/200], Loss: 0.1688
Epoch [190/200], Loss: 0.1836
Epoch [195/200], Loss: 0.1751
Epoch [200/200], Loss: 0.1599
```

EUMETNET

Text generation example

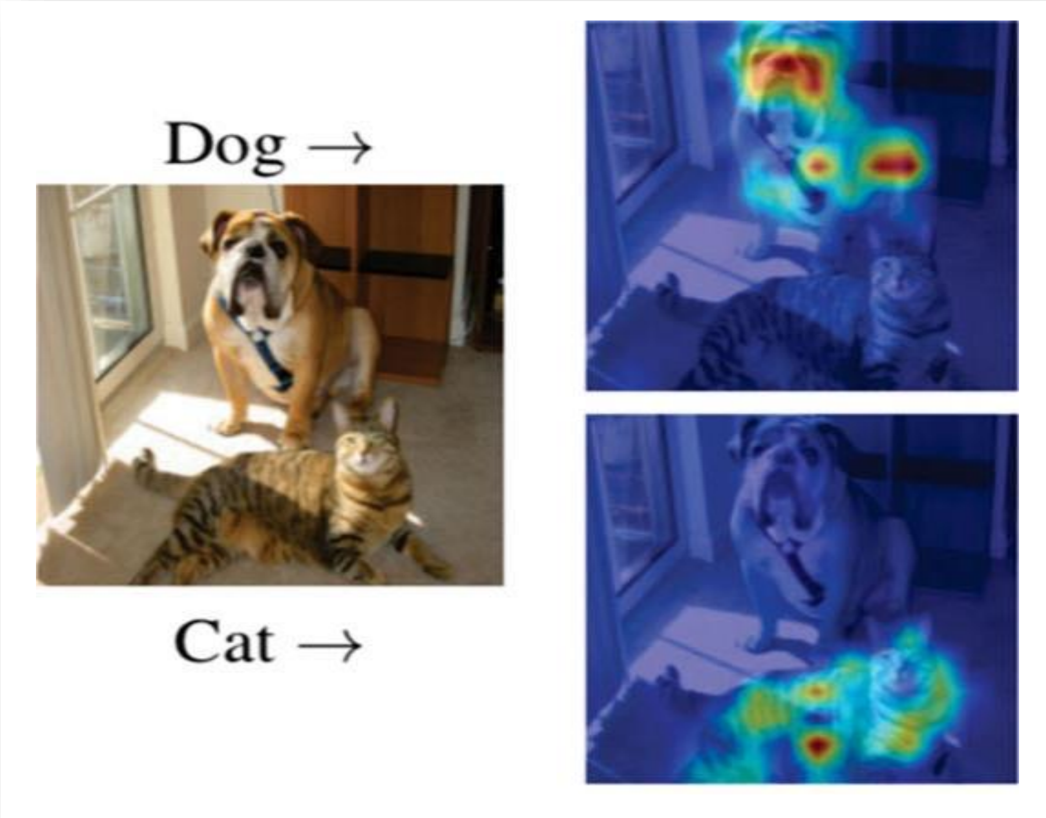
- Inference

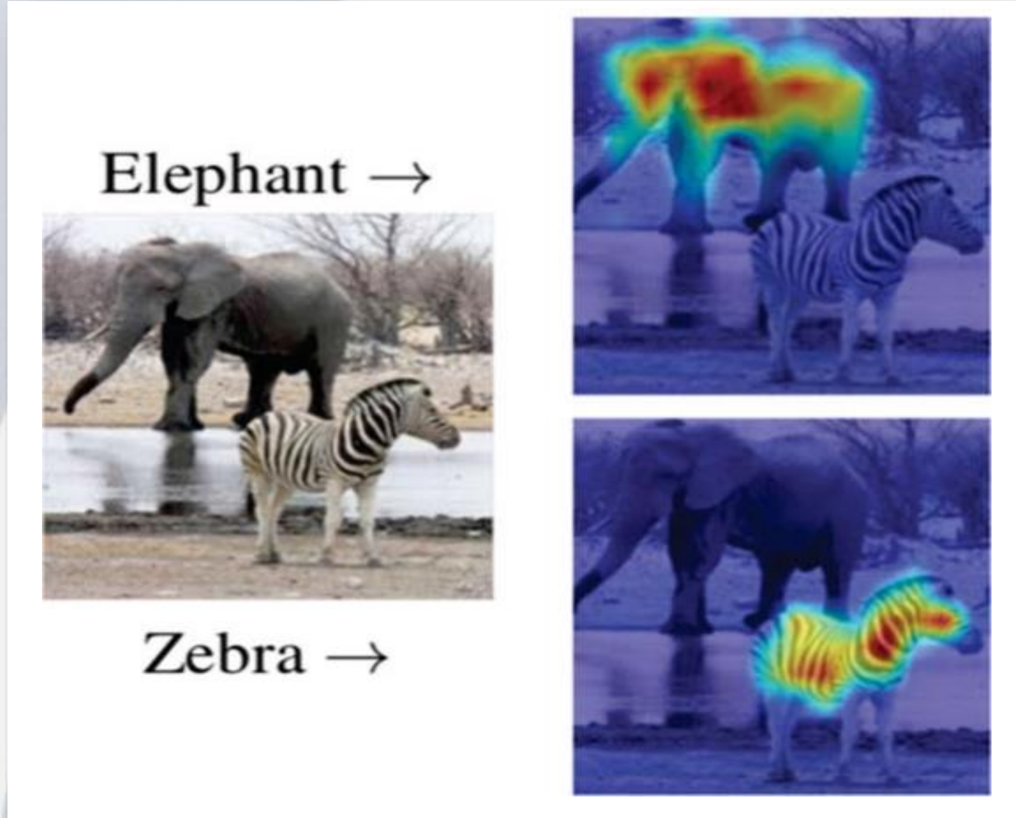
```
# Text generation function
def generate_text(model, vocab, start_text, max_len):
    model.eval()
    words = start_text.lower().split()
    input_ids = [vocab.get(word, vocab["<unk>"]) for word in words]
    generated = words.copy()
    generated[0]=generated[0].capitalize()
    input_seq = torch.tensor([pad_sequence(input_ids, max_len)], dtype=torch.long)
    with torch.no_grad():
        for _ in range(max_len - len(input_ids)):
            output = model(input_seq)
            next_token_logits = output[0, len(generated) - 1, :]
            next_token_id = torch.argmax(next_token_logits).item()
            next_word = [word for word, idx in vocab.items() if idx == next_token_id][0]
            generated.append(next_word)
            input_seq[0, len(generated) - 1] = next_token_id
            if next_token_id == vocab["<pad>"] or next_token_id == vocab["<unk>"] or any([s in next_word for s in {'.', '!', '?'}]):
                break
    return ' '.join(generated)

# Generate text
start_text = "The weather"
words=start_text.lower().split()
generated_text = generate_text(model, vocab, start_text, max_len)
print("\nGenerated Text:")
print(generated_text+"\n")
```

```
Generated Text:
The weather is perfect for a day at the beach.
```

Transformers for images

• Visualization of attention as a heatmap





Adaption von Xu et al. (2022)

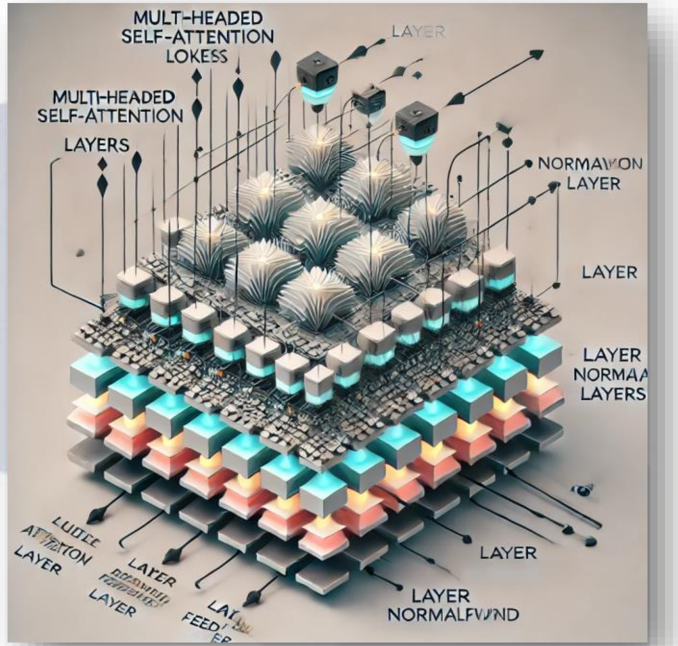
E-AI Basic Tutorials

1. Tutorial E-AI Basics 1: September 16, 2024, 11-12 CEST
2. Intro, Environment, First Example
3. 1.1 Basic Ideas of AI Techniques (20') [SH]
4. 1.2 Work Environment (20') [RP]
5. 1.3 First Example for AI - hands-on (20') [JK]

6. Tutorial E-AI Basics 2: September 18, 2024, 13-14 CEST
7. Dynamics, Downscaling, Data Assimilation Examples
8. 2.1 Dynamic Prediction by a Graph NN (20') [RP]
9. 2.2 Data Recovery/Denoising via Encoder-Decoder (20') [SH]
10. 2.3 AI for Data Assimilation (20') [JK]

11. Tutorial E-AI Basics 3: September 23, 2024, 11-12 CEST
12. LLM Use, Transformer Example, RAG
13. 3.1 Intro to LLM Use and APIs (20') [RP]
14. 3.2 Transformer for Language and Images (20') [JK]
15. 3.3 LLM Retrieval Augmented Generation (RAG) (20') [SH]

3.3



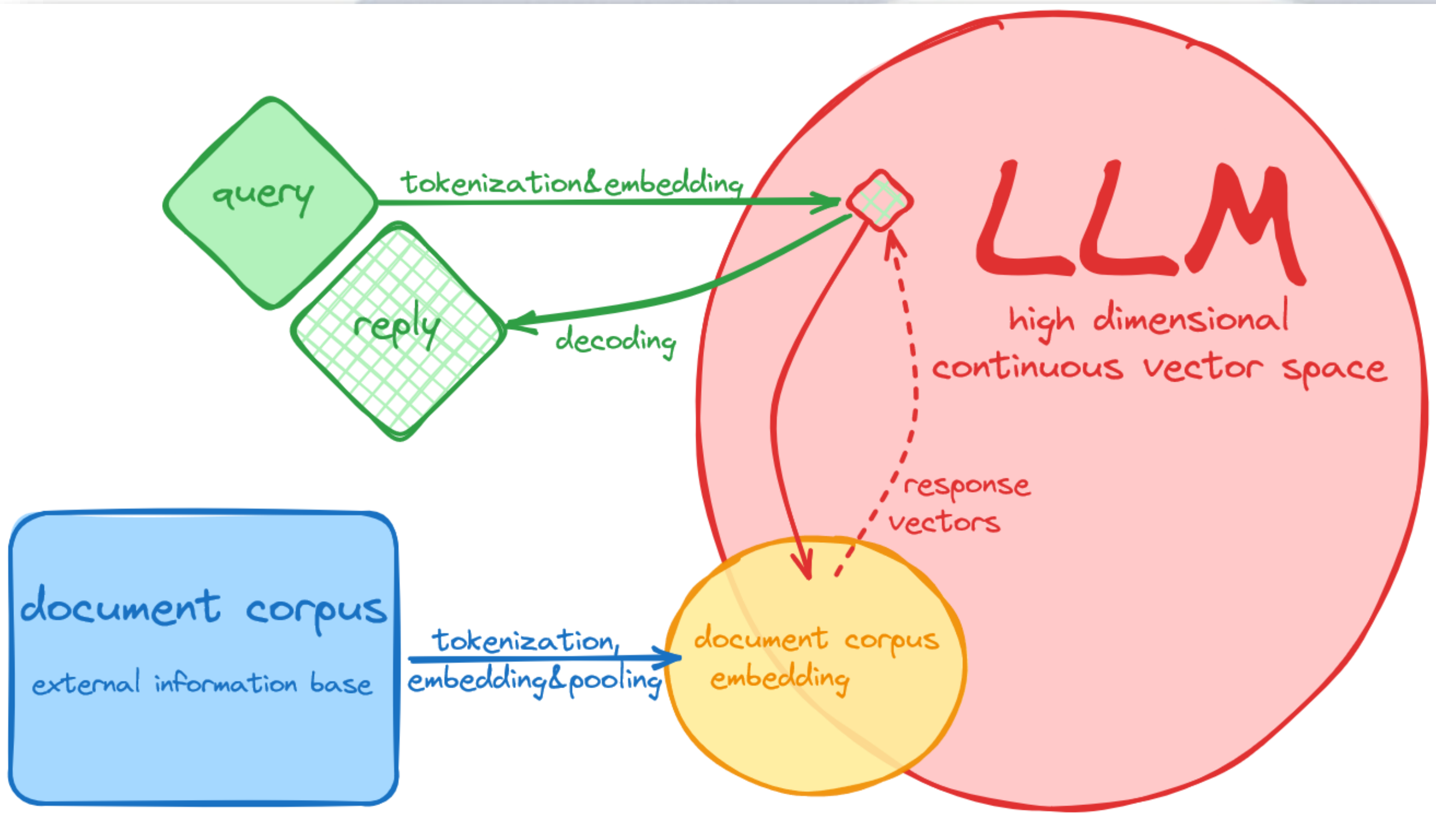
AI generated Images,
© Roland Potthast 2024

Retrieval-Augmented Generation

Retrieval-Augmented Generation (RAG) is an approach that combines information retrieval from **external knowledge sources** with a **language model** to generate more informed and accurate responses by **retrieving** relevant documents or data before generating an answer.

Core Components for RAG are

- **LLM**: The pre-trained language model (e.g., BERT, GPT) that generates the responses. It has been trained on a broad, general corpus during training.
- **Corpus**: The external, domain-specific information provided to the RAG system to improve retrieval and enhance the LLM's output. The LLM uses this corpus at inference time to "look up" relevant information rather than relying solely on its pre-trained knowledge.



Load the Language Model and Tokenizer

You will need the following libraries installed in your environment:

transformers: For handling tokenization and model loading.

torch: For handling tensors and leveraging the pre-trained model.

faiss: For building a vector search engine.

Here, we use DistilBERT, a pre-trained language model with tokenizer from Hugging Face.

tokenizer: Tokenizes the input text into numerical format.

language model: Outputs embeddings for the tokenized input, representing the text in a high-dimensional space.

```
[ ]:
import numpy as np
import faiss
import torch
from transformers import AutoTokenizer, AutoModel

# Step 1: Load the LLM
model_name = "distilbert-base-uncased"  # You can use any compatible model
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)
```

A **tokenizer** transforms raw input text (e.g., sentences or documents) into **tokens** (smaller units of text) that can be understood and processed by a language model. These tokens are usually numerical representations that the model can use as inputs.

Embedding refers to the process of converting tokens (usually token IDs) into **dense, continuous vectors** that represent the semantic meaning of the text. The embedding is a learned representation that captures the relationships between tokens in a high-dimensional space.

Prepare Documents for the Vector Database

Next, define some example **documents** that will be stored in the vector database. Each document will later be **encoded** into vectors for efficient similarity search. These documents represent a small collection, but in a real-world scenario, you'd use a much larger document set.

```
# Step 2: Prepare some documents for the vector database
documents = [
    "The cat sat on the mat.",
    "The dog chased the ball.",
    "Birds fly in the sky.",
    "Fish swim in the ocean.",
    "Tables have four legs."
]
```

The **vector base** is a specialized database that stores document embeddings and enables efficient similarity search between query and document vectors. It uses distance metrics to retrieve the most relevant documents, which are then passed to the language model to generate a response. By using vector-based search, RAG systems can provide more accurate and contextually relevant answers.

Encode Documents into Vectors

We create a function to **encode** the documents into vectors using the pre-trained language model. We use average pooling over the token embeddings to generate a single fixed-size vector for each document. This function takes in a list of documents, **tokenizes** them, and then passes them through the language model to get the **embeddings** (vector representations).
Pooling refers to the process of combining the individual token embeddings from a sequence (e.g., a sentence or document) into a single vector that represents the entire sequence.

```
# Step 3: Encode documents into vectors
def encode_documents(documents):
    inputs = tokenizer(documents, padding=True, truncation=True, return_tensors="pt")
    with torch.no_grad():
        embeddings = model(**inputs).last_hidden_state.mean(dim=1)  # Average pooling
    return embeddings.numpy()

# Create the vector database
document_vectors = encode_documents(documents)
dim = document_vectors.shape[1]
```

- **Tokenization** breaks text into tokens, and each token is mapped to an **embedding**.
- **Pooling** reduces token-level embeddings into a single fixed-size vector to represent the entire document or query. Different types of pooling (mean, max, CLS) are used depending on the task and model.

Build the FAISS Index

We use FAISS, a library for efficient similarity search, to build the vector search engine.
The IndexFlatL2 method builds an index that uses L2 distance (Euclidean distance) for similarity search. This is the simplest type of FAISS index, where all vectors are stored and compared using brute-force search. This is effective only for small datasets but scales poorly with large datasets.

FAISS provides several types of indexes, each optimized for different use cases.

```
# Step 4: Build the FAISS index
index = faiss.IndexFlatL2(dim)  # Using L2 distance
index.add(document_vectors)  # Add document vectors to the index
```

The **FAISS index** is a core component of the **FAISS** (Facebook AI Similarity Search) library, designed to enable fast and efficient similarity search and clustering of high-dimensional vectors. In the context RAG, FAISS is used to efficiently find the nearest neighbors (most similar vectors) in a large corpus of vector representations (document embeddings).

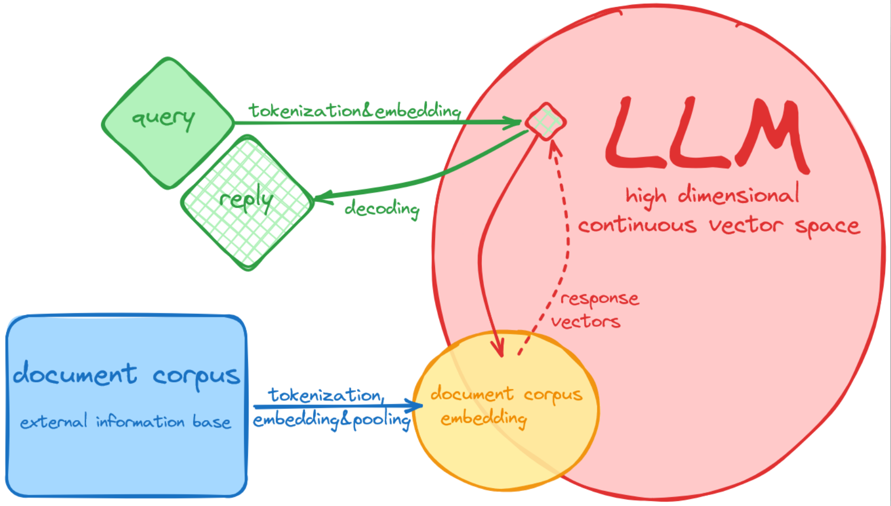
Define the RAG Function

In this step, we implement the core function of the RAG system, which involves:

1. **Encoding** the query into a vector.
2. **Searching** for the most similar document vectors using the FAISS index.
3. **Returning** the relevant document(s) as the "retrieved" portion of RAG.
4. **Generating** a response (here, we just return the retrieved documents).

In a more advanced setup, instead of just concatenating the documents, you could pass the relevant documents back into a model like GPT to generate a more complex answer. Here, for simplicity, we just concatenate the top retrieved document.

```python
# Step 5: Define a function for RAG
def retrieve_and_generate(query):
    # Encode the query
    query_vector = encode_documents([query])

    # Retrieve top-k similar documents
    k = 1  # Number of top results to retrieve
    D, I = index.search(query_vector, k)  # D: distances, I: indices

    # Get the relevant documents
    relevant_docs = [documents[i] for i in I[0]]

    # Simple "generation" (for demonstration, just concatenate)
    response = " ".join(relevant_docs)
    return response
```



Use the RAG System

Finally, let's test the system with a sample query.

```
# Step 6: Use the RAG system
query = "What do animals do?"
response = retrieve_and_generate(query)
print("Response:", response)
```

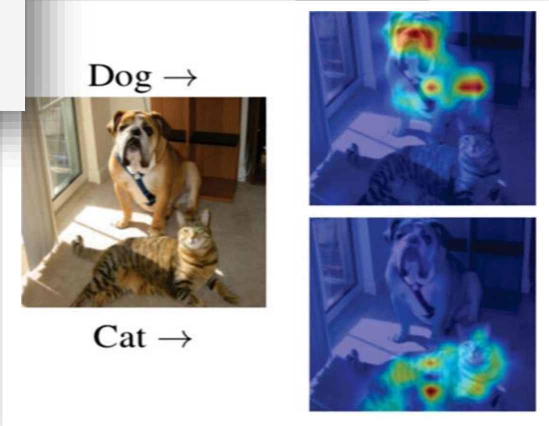
```
Response: Fish swim in the ocean.
```

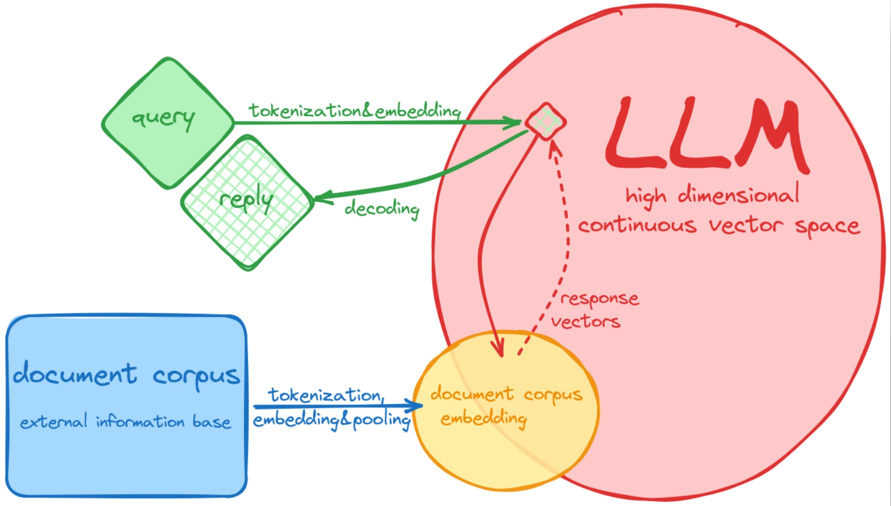
```
[ ]:
```

```
query = "What do you know about barking "
response = retrieve_and_generate(query)
print("Response:", response)
```

```
Response: The dog chased the ball.
```

```
documents = [
    "The cat sat on the mat.",
    "The dog chased the ball.",
    "Birds fly in the sky.",
    "Fish swim in the ocean.",
    "Tables have four legs."
]
```





E-AI Basic Tutorials

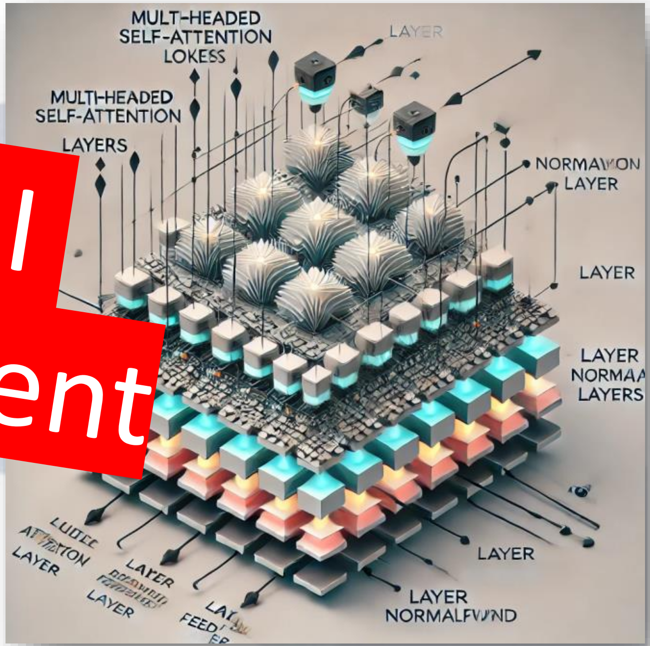
1. Tutorial E-AI Basics 1: September 16, 2024, 11-12 CEST
2. Intro, Environment, First Example
3. 1.1 Basic Ideas of AI Techniques (20') [SH]
4. 1.2 Work Environment (20') [RP]
5. 1.3 First Example for AI - hands-on (20') [JK]

6. Tutorial E-AI Basics 2: September 18, 2024, 13-14 CEST
7. Dynamics, Downscaling, Data Assimilation Examples
8. 2.1 Dynamic Prediction by a Graph NN (20') [RP]
9. 2.2 Data Recovery/Denoising via Encoder-Decoder (20') [SH]
10. 2.3 AI for Data Assimilation (20') [JK]

11. Tutorial E-AI Basics 3: September 23, 2024, 11-12 CEST
12. LLM Use, Transformer Example, RAG
13. 3.1 Intro to LLM Use and APIs (20') [RP]
14. 3.2 Transformer for Language and Images (20') [JK]
15. 3.3 LLM Retrieval Augmented Generation (RAG) (20') [SH]

Many Thanks

Enjoy AI Development



AI generated Images,
© Roland Potthast 2024