INFO 4000
Informatics III

Data Science Specialization - Advanced

Transformers – ViT and LLMs

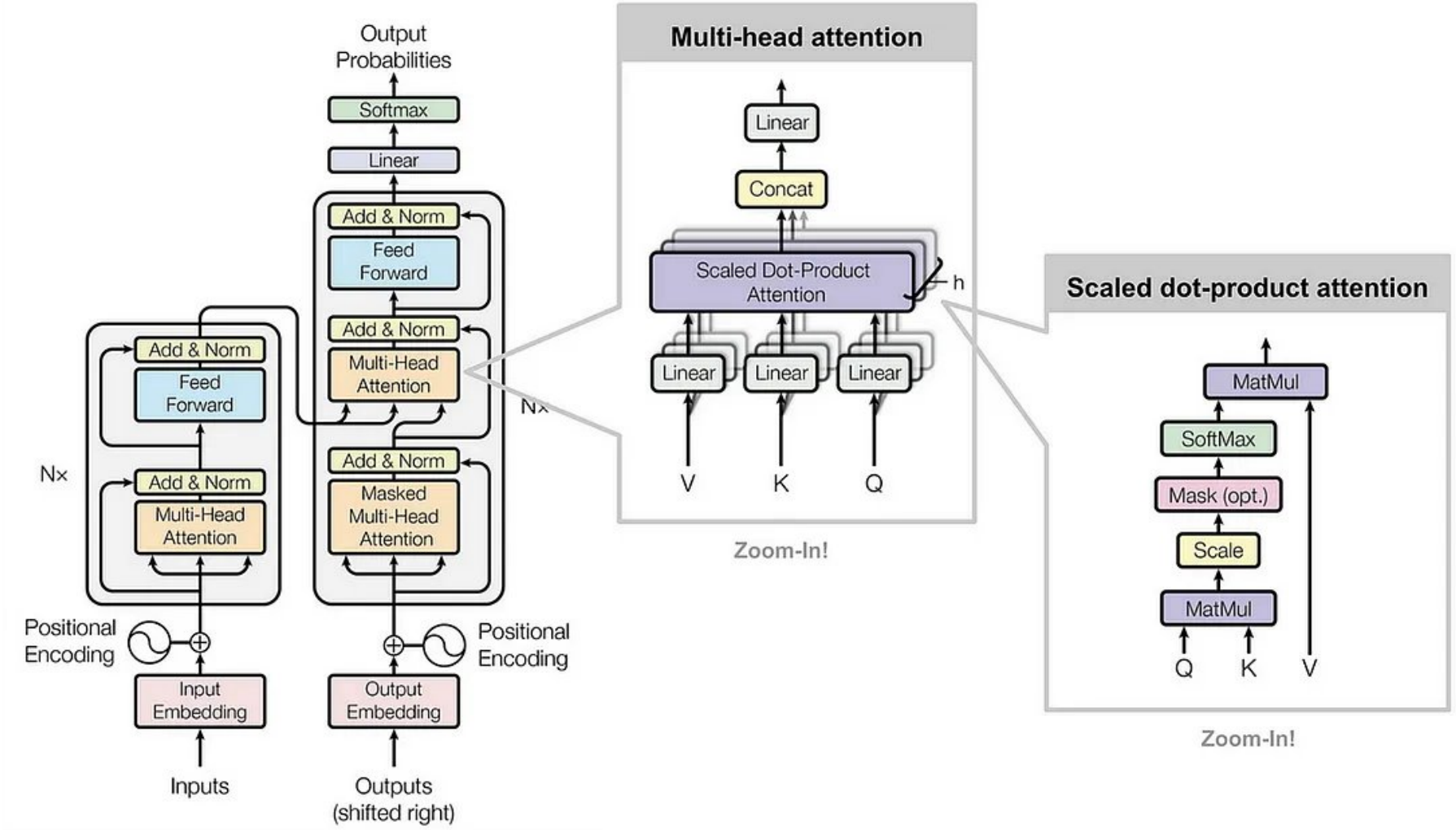
Course Instructor

Jagannath Rao

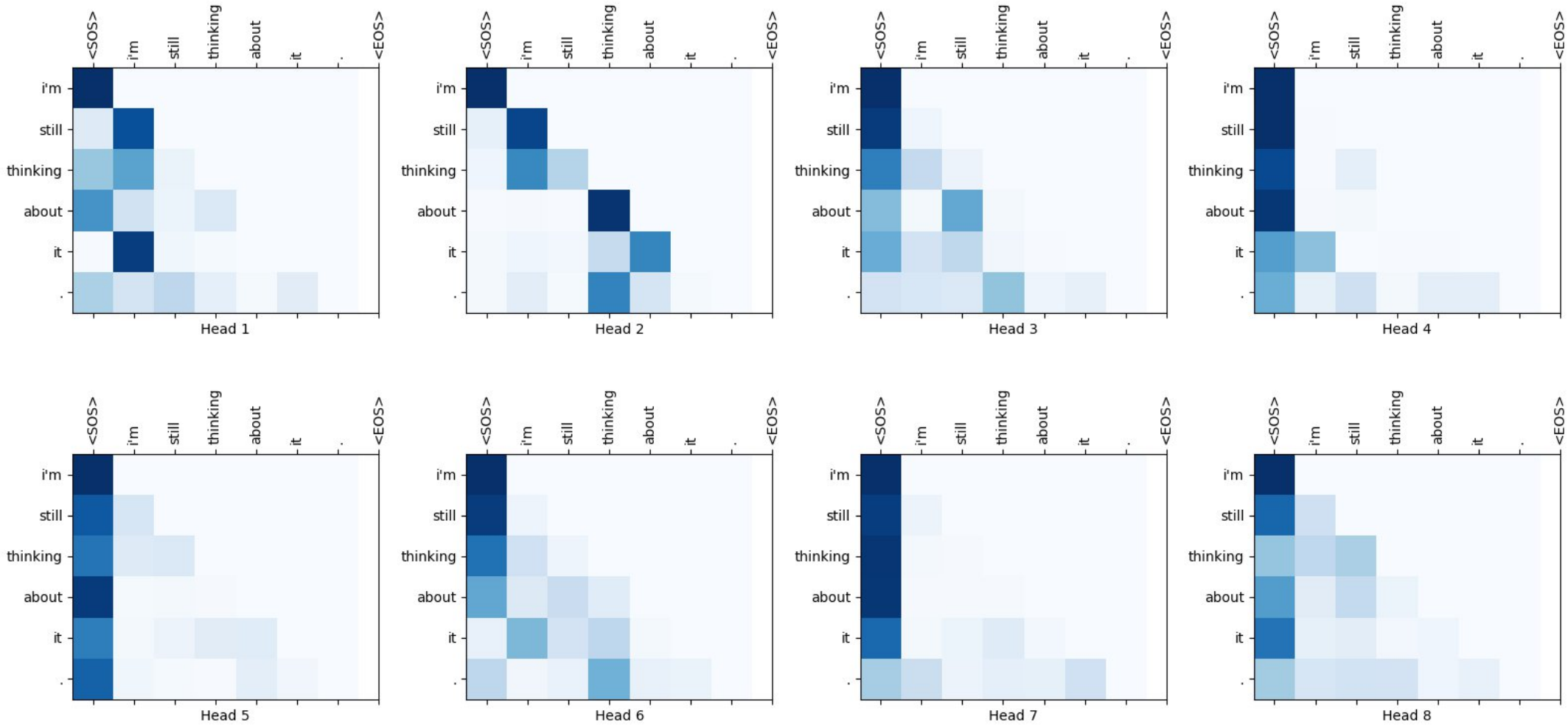
raoj@uga.edu

Recap

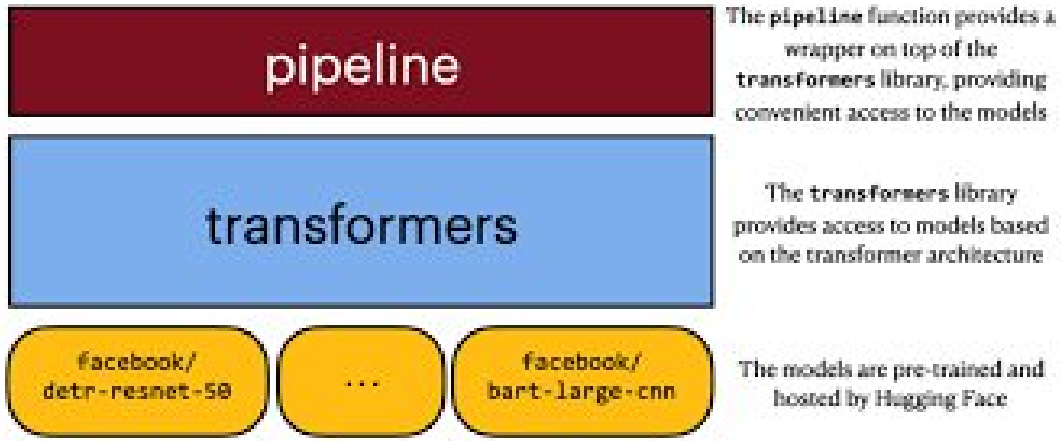
The transformer



Attention


<SOS>
i'm
still
thinking
about
it
.
<EOS>
Head 1
Head 2
Head 3
Head 4
Head 5
Head 6
Head 7
Head 8

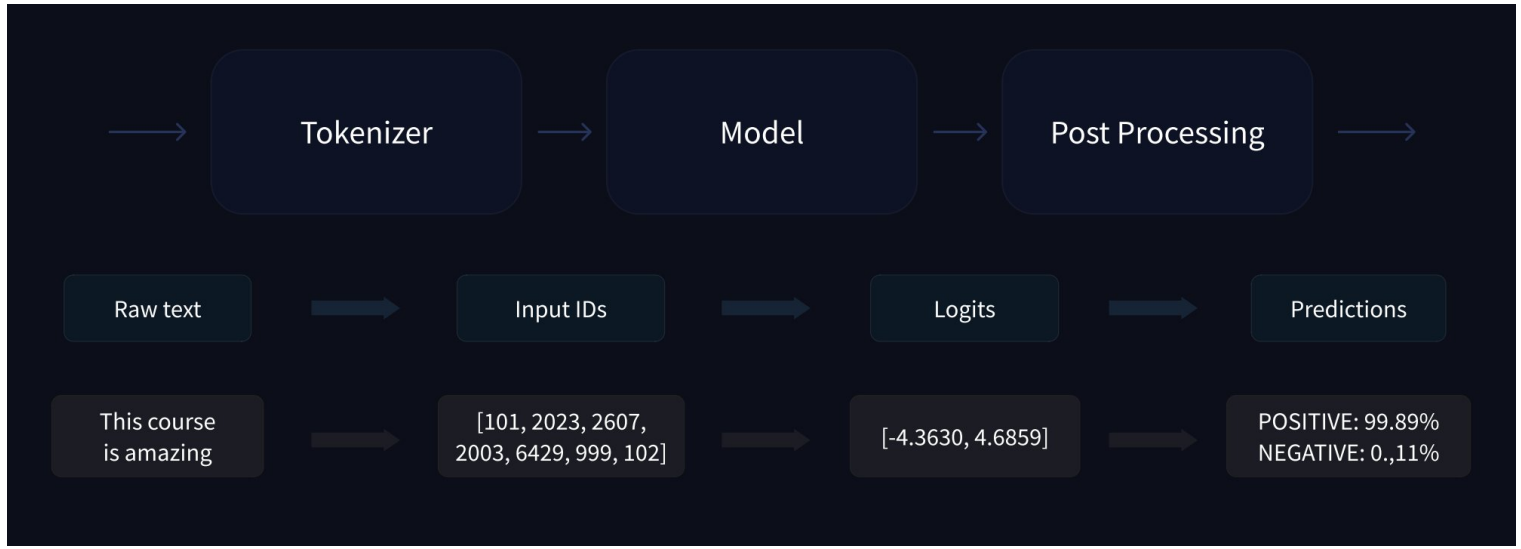
Pipeline class and the ease of use



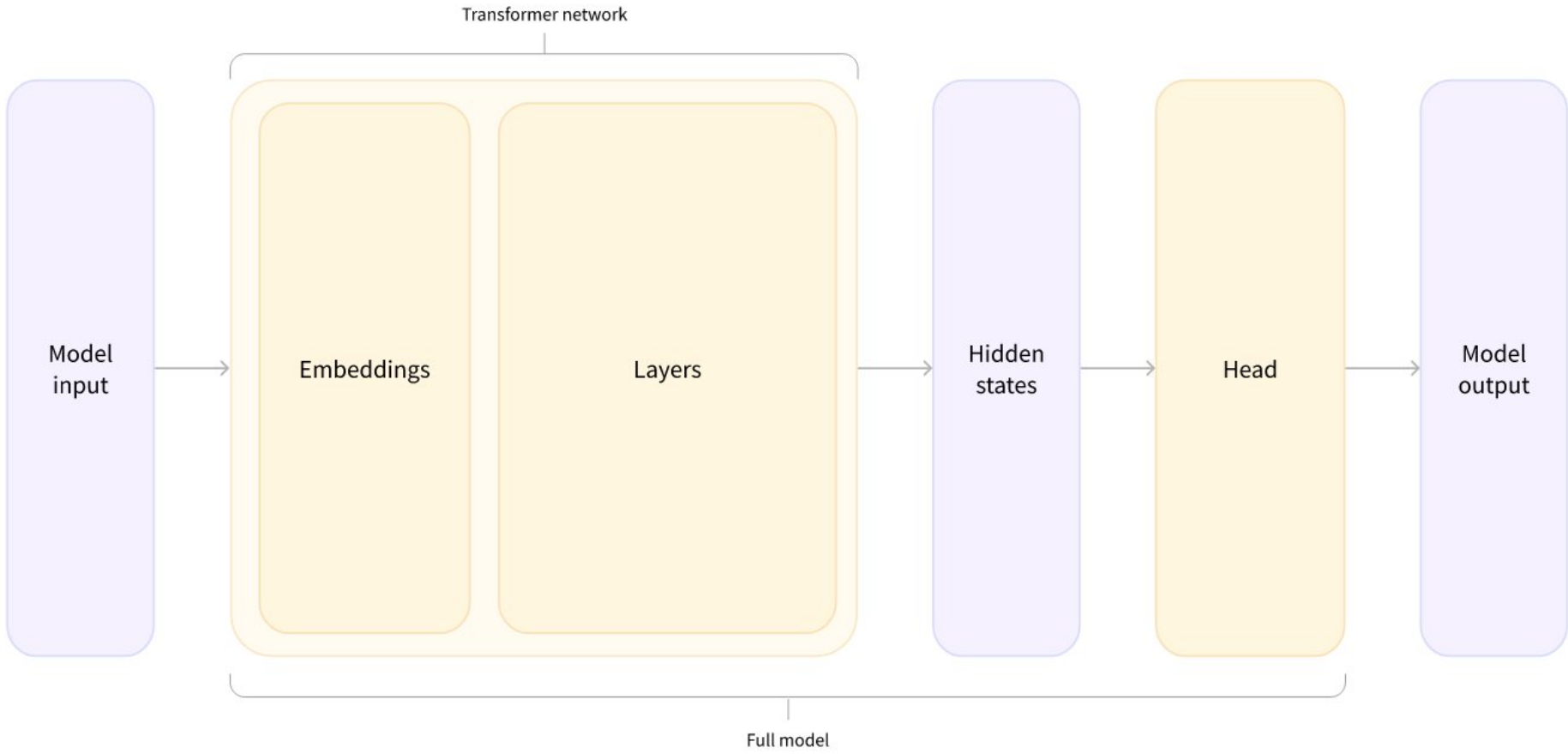
The `pipeline` function provides a wrapper on top of the `transformers` library, providing convenient access to the models

The `transformers` library provides access to models based on the transformer architecture

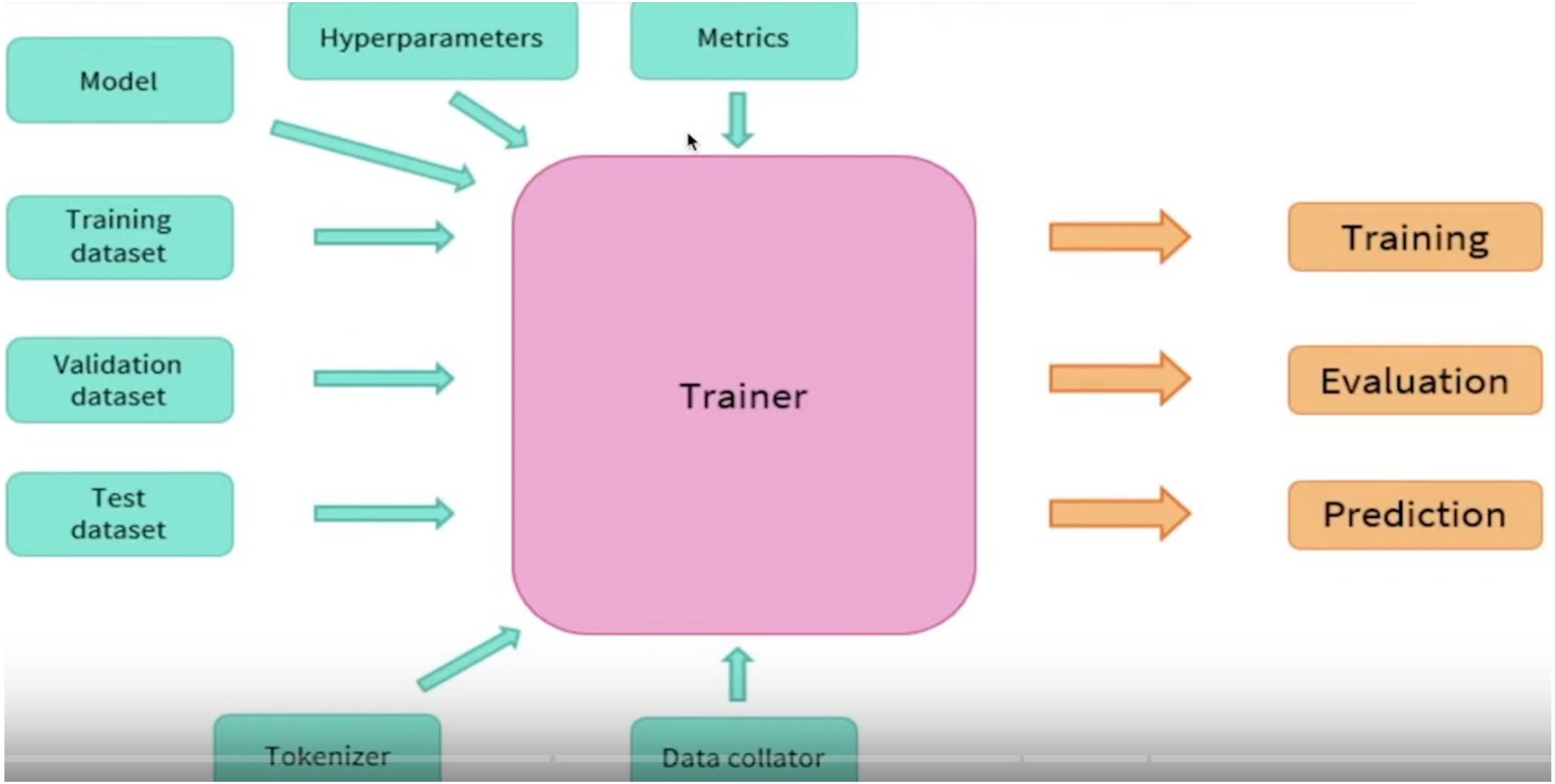
The models are pre-trained and hosted by Hugging Face



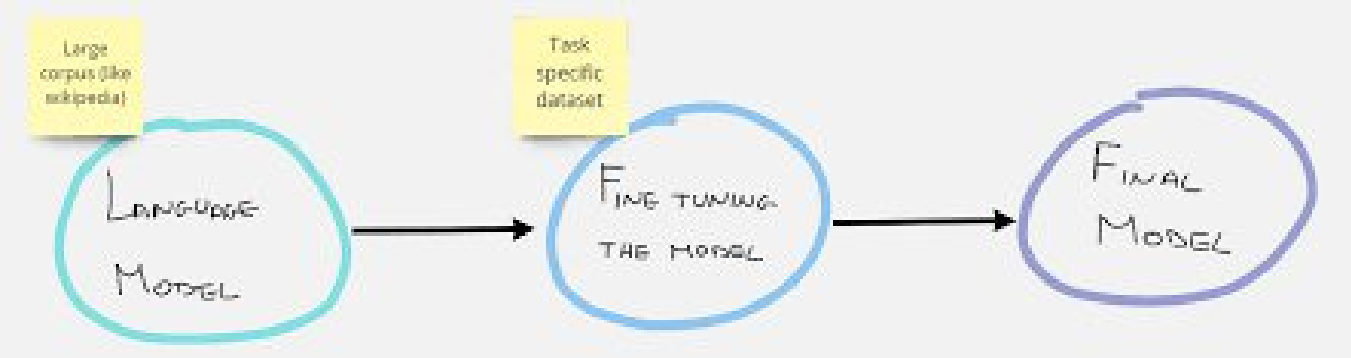
Visual representation

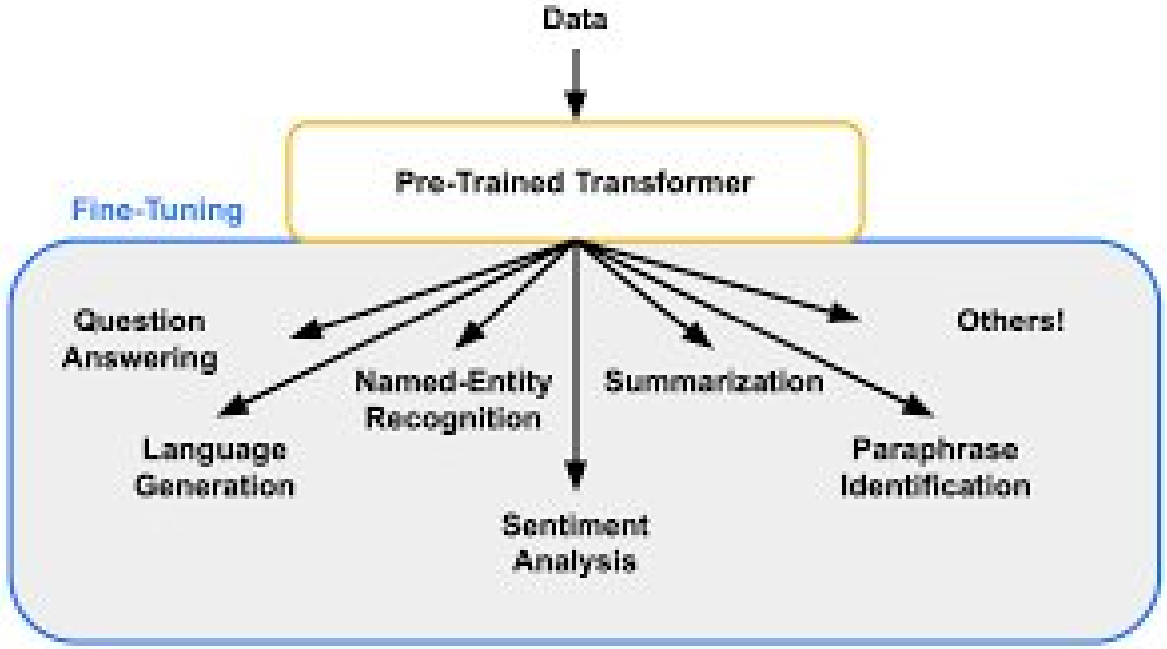


HuggingFace Trainer() – Built on PyTorch



Fine tuning a Language model

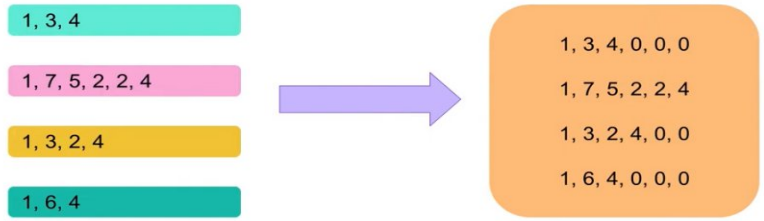




What do Data Collators do?

1. They put together a list of samples into a mini training batch
2. At the very least they pad or truncate input samples to make sure all samples in a mini batch are of same length.
3. Typical mini-batch sizes range from 16 to 256 samples, depending on the model size, data, and hardware constraints
4. Data collators are **task-specific**

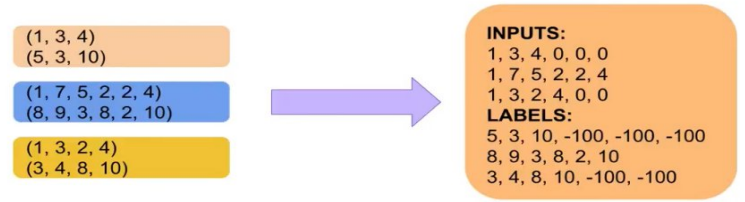
```
data_collator = DataCollatorWithPadding(
  tokenizer=tokenizer
)
```



Does basic padding of batches of data

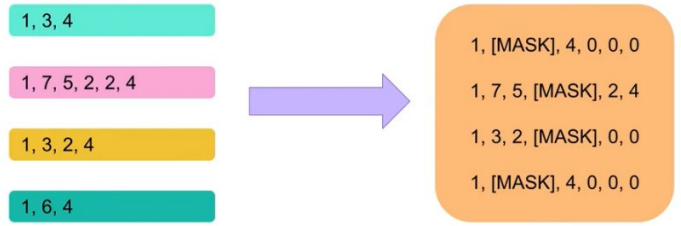
```
data_collator = DataCollatorForTokenClassification(
            tokenizer=tokenizer
)

data_collator = DataCollatorForSeq2Seq(
  tokenizer=tokenizer,
  model=model
)
```

Used when labels are also sequences of different lengths




```
data_collator = DataCollatorForLanguageModeling(
            tokenizer=tokenizer,
            mlm=True,
            mlm_probability=0.15
)
```

When masking is involved



Process as we did in the example

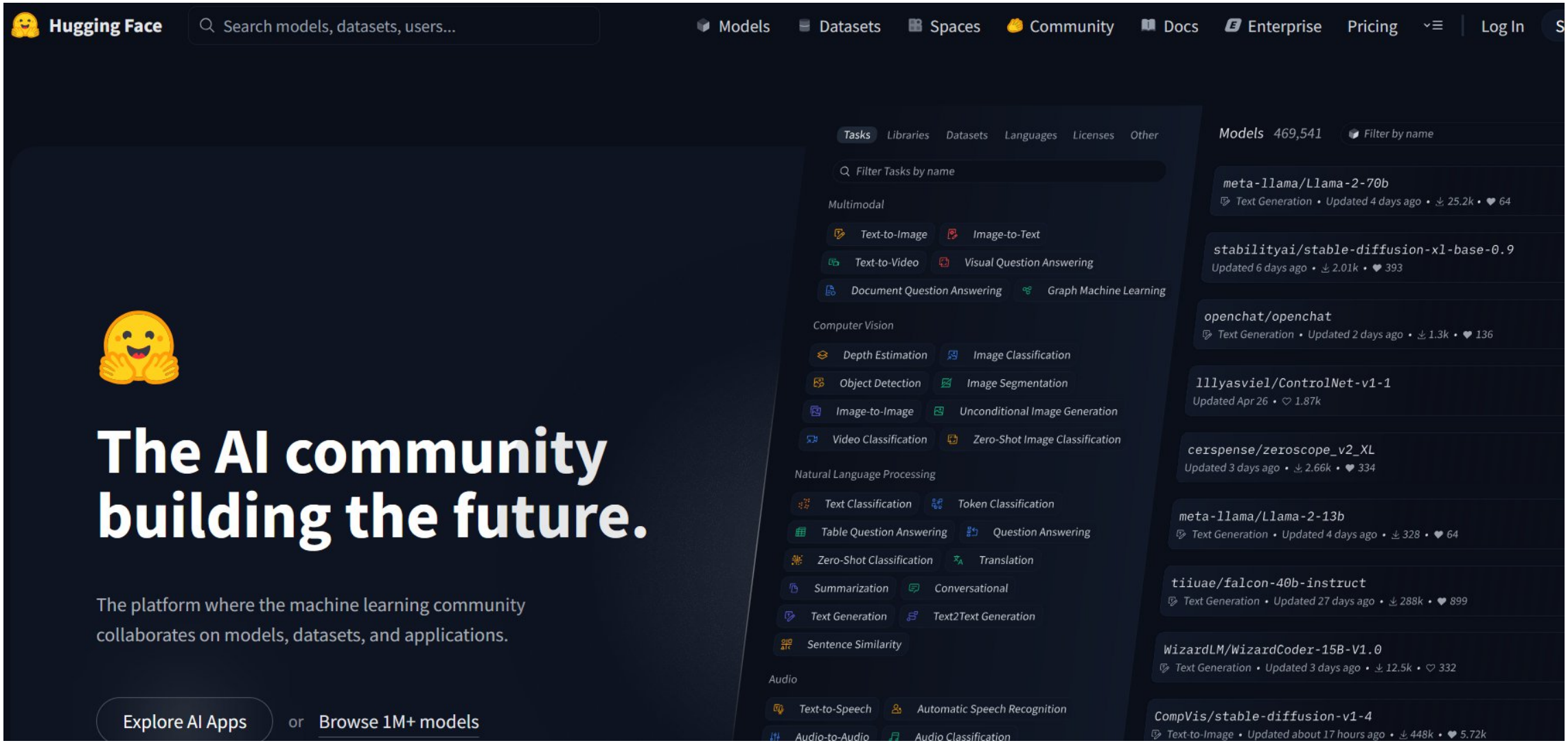
```python
from datasets import load_dataset
from transformers import AutoTokenizer, DataCollatorWithPadding

raw_datasets = load_dataset("glue", "mrpc")
checkpoint = "bert-base-cased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)

def tokenize_function(examples):
    return tokenizer(examples["sentence1"], examples["sentence2"], truncation=True)

tokenized_datasets = raw_datasets.map(tokenize_function, batched=True)
data_collator = DataCollatorWithPadding(tokenizer)

from transformers import AutoModelForSequenceClassification

model = AutoModelForSequenceClassification.from_pretrained(checkpoint, num_labels=2)

from transformers import TrainingArguments

training_args = TrainingArguments(
    "test-trainer",
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=5,
    learning_rate=2e-5,
    weight_decay=0.01,
)
```

```python
from transformers import Trainer

trainer = Trainer(
    model,
    training_args,
    train_dataset=tokenized_datasets["train"],
    eval_dataset=tokenized_datasets["validation"],
    data_collator=data_collator,
    tokenizer=tokenizer,
)
trainer.train()
```

```python
predictions = trainer.predict(tokenized_datasets["validation"])
print(predictions.predictions.shape, predictions.label_ids.shape)
```

You can now build the prediction function to use the model

Hugging Face is a great platform for getting all sorts of pre-trained models



Important point to note for using some models from HF

- It’s important to note a key detail about its "free" status:
  - There are some models which are ‘gated’ and need an acceptance of license agreement and Access token.
  - Example, you must **accept a license agreement** from Google to download and use their model weights.
  - This is a standard requirement for many open-source models from major companies.
- For example, to access Gemma 2B on Hugging Face:
  - **Create a Hugging Face Account:** If you don't already have one, sign up for a free account on the Hugging Face website.
  - **Navigate to the Model Page:** Go to the Gemma 2B model page, which is google/gemma-2b.
  - **Accept the License:** You'll see a prompt asking you to review and agree to Google's usage license. Click the button to agree. This grants you permission to use the model's files.
  - Once you've done these steps, do the next steps – see next slide OR watch this video https://www.youtube.com/watch?v=USgyt6sHJeg

Generate and use Access Token

When you open any Gemma model card in HuggingFace, you'll be prompted to acknowledge the license. By clicking on it, you provide your consent and agree to the terms a conditions of Gemma.

Steps to Create a New Access Token with Read Permissions:

1. Navigate to Your Profile:
    - Click on your profile icon located at the top right corner of the screen.
2. Access Settings:
    - From the dropdown menu, select Settings to open your account settings page.
3. Go to Access Tokens:
    - In the settings menu, find and click on Access Tokens. This section allows you to manage your personal access tokens.
4. Create a New Token:
    - Click on the Create New Token button to generate a new access token.
5. Set Permissions:
    - In the permissions section, ensure that Repositories permissions are configured with Read access. This is important for accessing the repositories you need.
6. Generate Token:
    - Once you've configured the necessary permissions, click on Generate Token. Make sure to copy the token and store it securely, as it will only be displayed once.

After that, follow the lines of code below in your notebook:

```
from huggingface_hub import login
login(access_token)
```

Important point to note for using some models from HF

- It’s important to note a key detail about its "free" status:
  - There are some models which are ‘gated’ and need an acceptance of license agreement and Access token.
  - Example, you must **accept a license agreement** from Google to download and use their model weights.
  - This is a standard requirement for many open-source models from major companies.
- For example, to access Gemma 2B on Hugging Face:
  - **Create a Hugging Face Account:** If you don't already have one, sign up for a free account on the Hugging Face website.
  - **Navigate to the Model Page:** Go to the Gemma 2B model page, which is google/gemma-2b.
  - **Accept the License:** You'll see a prompt asking you to review and agree to Google's usage license. Click the button to agree. This grants you permission to use the model's files.
  - Once you've done these steps, do the next steps – see next slide OR watch this video https://www.youtube.com/watch?v=USgyt6sHJeg

Get API key for free GenAI model APIs from google

- It’s important to note a key detail about its "free" status:
  - There are some models which are ‘gated’ and need an acceptance of license agreement and Access token.
  - Example, you must **accept a license agreement** from Google to download and use their model weights.
  - This is a standard requirement for many open-source models from major companies.
- For example, to access Gemma 2B on Hugging Face:
  - **Create a Hugging Face Account:** If you don't already have one, sign up for a free account on the Hugging Face website.
  - **Navigate to the Model Page:** Go to the Gemma 2B model page, which is google/gemma-2b.
  - **Accept the License:** You'll see a prompt asking you to review and agree to Google's usage license. Click the button to agree. This grants you permission to use the model's files.
  - Once you've done these steps, do the next steps – see next slide OR watch this video https://www.youtube.com/watch?v=USgyt6sHJeg

Important point to note for using some models from HF

To access a free API for Google's generative AI models, use

- Google AI Studio to generate an API key for the Gemini API. This API has a free tier for developers to test and build applications.

How to get a free API key?

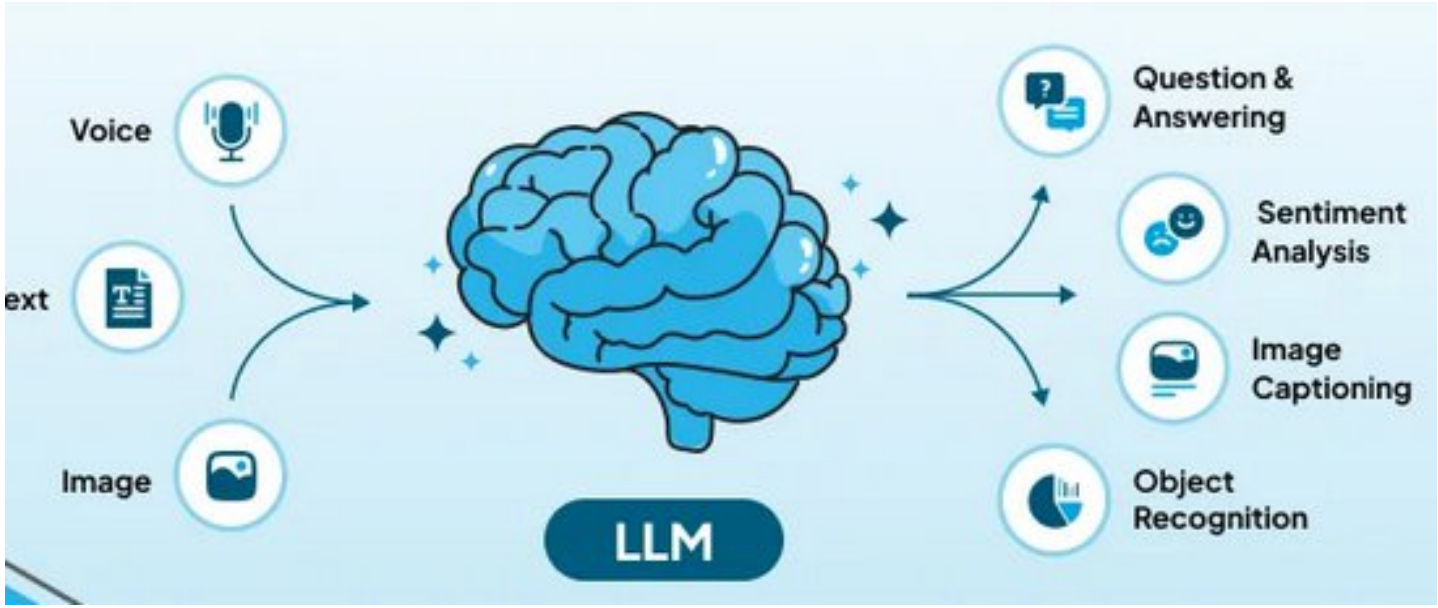
- Go to the Google AI for Developers website: Navigate to ai.google.dev/aistudio.
- Sign in with a Google account: A personal Google account is required. It is free to use for all users with a Google account.
- Go to the "Get API Key" section: Once logged in, click "Get an API key".
- Create your key: Follow the on-screen instructions to create a new Gemini API key. This key authenticates requests to the API.

What the free tier provides

- Access to Gemini models: The Gemini API allows integrating Google's latest generative AI models, including Gemini 1.5 Flash, into projects.
- Generate code, text, and more: The API enables building applications that can generate code, write text, and process multimodal inputs. There are daily rate limits and you can check that on google AI Studi website.
- Test and prototype: The free tier is for prototyping and testing, with daily rate limits sufficient for development.

Building a Q&A Chatbot

Application building with LLMs



What is a chatbot?

Definition of a chatbot:

- A chatbot is a computer program designed to simulate human conversation through text or voice interactions.
- It uses natural language processing and machine learning techniques to understand user inputs and generate appropriate responses, often in real-time.
- A Q&A chatbot is a specialized type of conversational AI program designed to understand and respond to user queries by providing relevant answers from a knowledge base.
- Q&A chatbots are particularly useful for information retrieval tasks and can be employed in various domains such as customer support, educational platforms, or general knowledge applications.

Key characteristics of a Q&A chatbot include:

- Focused on answering specific questions rather than engaging in open-ended conversation
- Typically operates on a defined knowledge base or dataset
- Aims to provide concise, accurate answers rather than generating creative or contextual responses

Methods of building a chatbot

Using pre-trained models:

- Utilize text to text (seq2seq) generation models from HF or google directly
- Fine-tune these models on your specific Q&A dataset if required.

Pipeline approach:

- Use Hugging Face's pipeline for text-generation, which simplifies the process.

Custom model architecture:

- Design a custom model using Hugging Face's transformers library.
- Train it from scratch.

Retrieval-based approach:

- Small systems: **Retrieval-based approach for building Q&A chatbots.**

- More complex and large knowledge base systems: Retrieval Augmented Generation (RAG)
RAG uses a retrieval component to find relevant information from a knowledge base.

- Generative Q&A: It uses a language model to generate responses based on corpus it is trained on and the input query.

Q & A models which need a context for retrieval of answers

- QA models are designed to **find answers within a given context** rather than generate answers from their internal "knowledge.“ (They are extractive).
- So, the basis is:
  - You have access to a knowledge base or document during inference.
  - You can use a retrieval mechanism to first fetch the relevant context (like a paragraph from a manual) before answering the question.
  - To retrieve the context, you can use embeddings and find similarity between input and knowledge base embeddings.

RAG - Context based retrieval systems

1. Retrieval-Augmented Generation (RAG) is a technique that connects LLMs with external knowledge sources to improve the quality of AI-generated responses.

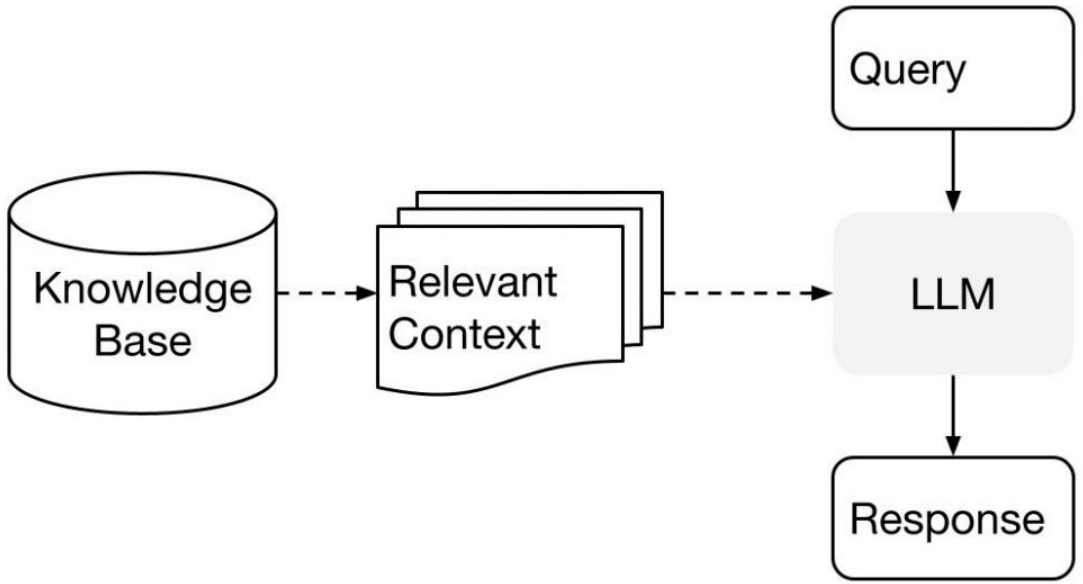
2. **Retrieval:** The system queries a connected vector store built from your structured and unstructured data — including websites, PDFs, documents, and more. These sources are converted into embeddings, allowing the system to fetch the most relevant context based on the user's query.

3. **Generation:** It passes both the query and the retrieved content to the language model, which uses that combined context to generate a response.

4. Unlike pre-trained models that rely on outdated data, RAG pulls live content from your documents, knowledge base, or website.

5. RAG systems adapt quickly to any domain — from legal and healthcare to ecommerce and education. You don't need domain-specific models. Just plug in your knowledge base and go live.

6. In summary: The embedding-based retrieval system acts as a librarian finding the right books, but the LLM is the expert author who reads those books and writes a new, informative article based on their contents



How to search in the knowledge base for context?

1. Any vectorizer takes an input sentence (in this case, a question) and converts it into a **fixed-length vector embedding**.
2. This is the vector representation of the question.
3. It's a numerical vector that captures the meaning of the question in a way that allows us to compare it to other sentences or texts (like the knowledge base).
4. **Similarity Computation:** The vector for the question is compared to the vector representations of all the entries in the knowledge base (which should also be embedded using the same model). This comparison uses cosine similarity.
5. **Finding the Closest Match:** The code identifies which entry in the knowledge base has the highest similarity to the question (i.e., which context is most likely to contain the answer).
6. **Retrieving the Closest Context:** Based on the highest similarity score, the most relevant context from the knowledge base is selected, which can then be used by the QA model to extract the final answer.

Embedding of data – Vector databases and Term-Document Matrix (TDM)

1. Embeddings are numerical representations of data, such as text, images, audio, or other complex information.
2. They are typically high-dimensional vectors, meaning they consist of a large array of numbers.
3. The key idea behind embeddings is that similar items in the original data space will have their corresponding embeddings located close to each other in the multi-dimensional vector space.
4. A **vector database** is a specialized database designed to efficiently store, index, and query these high-dimensional vector embeddings.
5. A **TDM** is a fundamental data structure in text mining and natural language processing (NLP). It is a mathematical matrix that describes the frequency of terms (words) that occur in a collection of documents (a corpus).
   - A TDM serves as a proxy for a document embedding because each row of the matrix represents a document as a numerical vector
   - In the assignment you will use TDM as the basis for embeddings and as you have learned already this is generated using the TF-IDF vectorizer.

Q & A models which are generative in nature

1. If you want the model to generate answers without providing a context, you'll need to fine-tune a text-generation model (like GPT-2, T5, or BART) instead of a BERT-QA model.

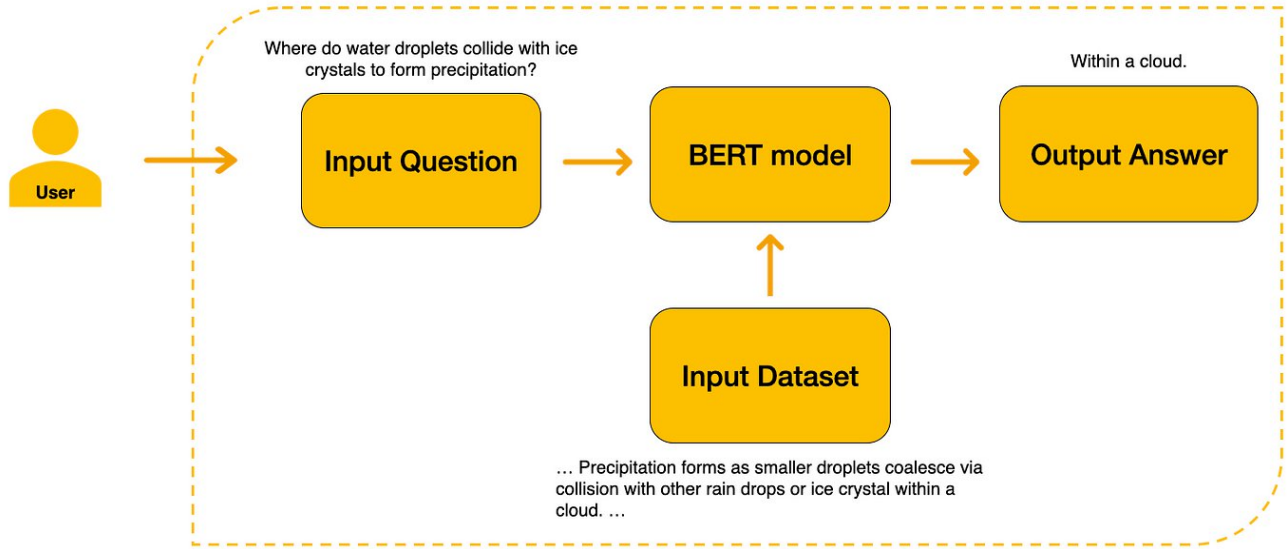
2. A **T5 or GPT2** model can be fine tuned on a corpus of domain specific text.

3. Prepare a Q&A Dataset where each entry consists of a question as input and the corresponding answer as output. The model will learn to map questions to answers during fine-tuning.

4. Fine-Tune the Model on this dataset. During fine-tuning, the model will learn to generate answers based on the questions.

5. **Inference:** Once fine-tuned, you can ask questions without providing any external context, and the model will generate answers based on its fine-tuned knowledge.

6. You don't need to provide a context at inference time. The model will generate the answers based on the knowledge it has learned during fine-tuning.

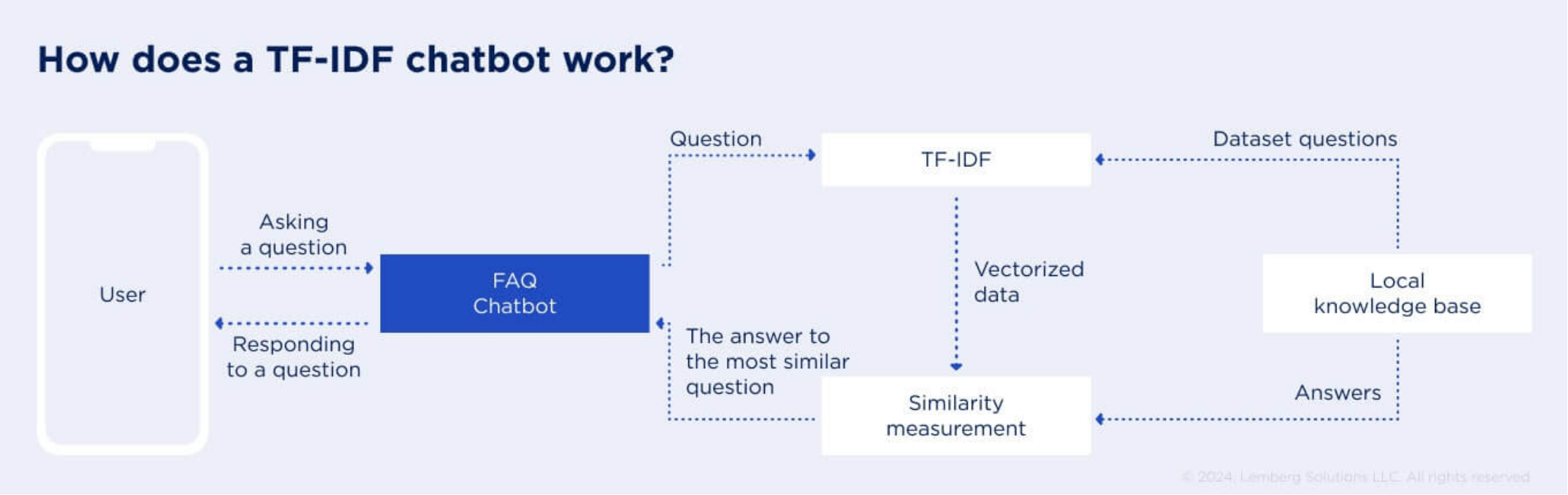


MP1

Two-part mini project

1. **Part 1:** Develop an NLP application, finetuning, to diagnose the disease based on the symptoms - dataset provided
   - Build a separate prediction function using the pipeline class
   - New inputs are provided
   - Use a HF model of your choice for the given type of application use the trainer class.
2. **Part 2:** Develop a context retrieval-based Physics Q&A chatbot which will answer your physics questions.
   - A manual of physics concepts are provided as a Json file, and it has “context’ as key and a physics concept as value.
   - Use a suitable pretrained LLM model from HF or google directly and make sure it is text to text generation (don’t pick a model which is too large as it may take a long time to download and may not have enough memory). A possible model to us - "google/flan-t5-small"
   - You shall use the TDM developed using TF-IDF as your embeddings.
   - The LLM along with the context / knowledge based shall run as an API on the localhost and
   - Q&A frontend will be developed as a streamlit app, and it should talk to the backend for getting answers and displaying

General schematic to guide you

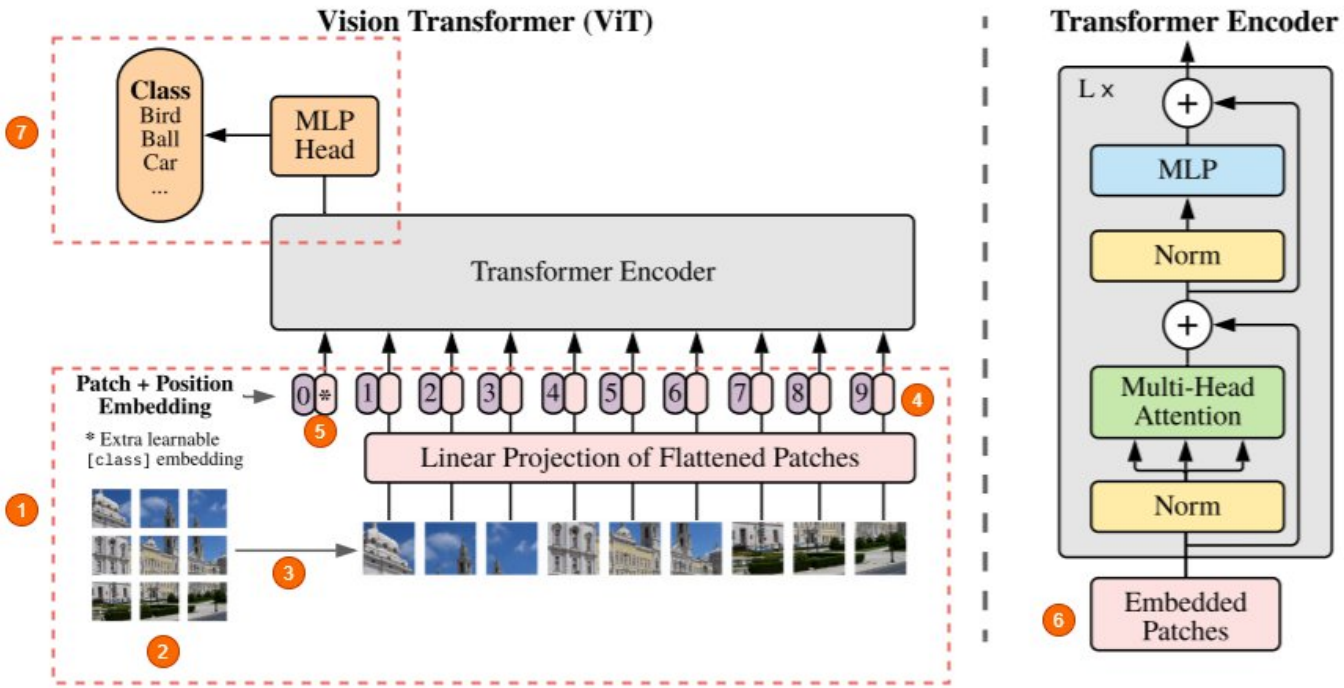


ViT – Vision Transformers

What is a Vision Transformer?

- A **Vision Transformer (ViT)** is a type of neural network that can be used for image classification and other computer vision tasks.
- It was first published in another seminal Google Brain paper called ***"An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale"***
- Vision Transformer, is a model that applies the Transformer architecture, originally designed for natural language processing tasks, to computer vision tasks.
- ViT is an "encoder" only model and does not use the "decoder".
- The fundamental technique in ViT is to treat an image as a sequence of fixed-size patches, much like how a sentence is treated as a sequence of words or tokens in NLP and then process these patches using the Transformer's mechanisms.
- When trained on enough data, ViTs have achieved competitive, and in many cases superior, performance compared to traditional convolutional neural networks (CNNs) on a range of vision tasks.
- The easiest comparison to doing NLP with transformers is
  - The image is considered as sentence.
  - If we breakdown an image into patches, the patches would be the tokens of that image.

Vision Transformer Architecture



(1) We are only using the Encoder part of the transformer, but the difference is in how they are feeding the images into the network.

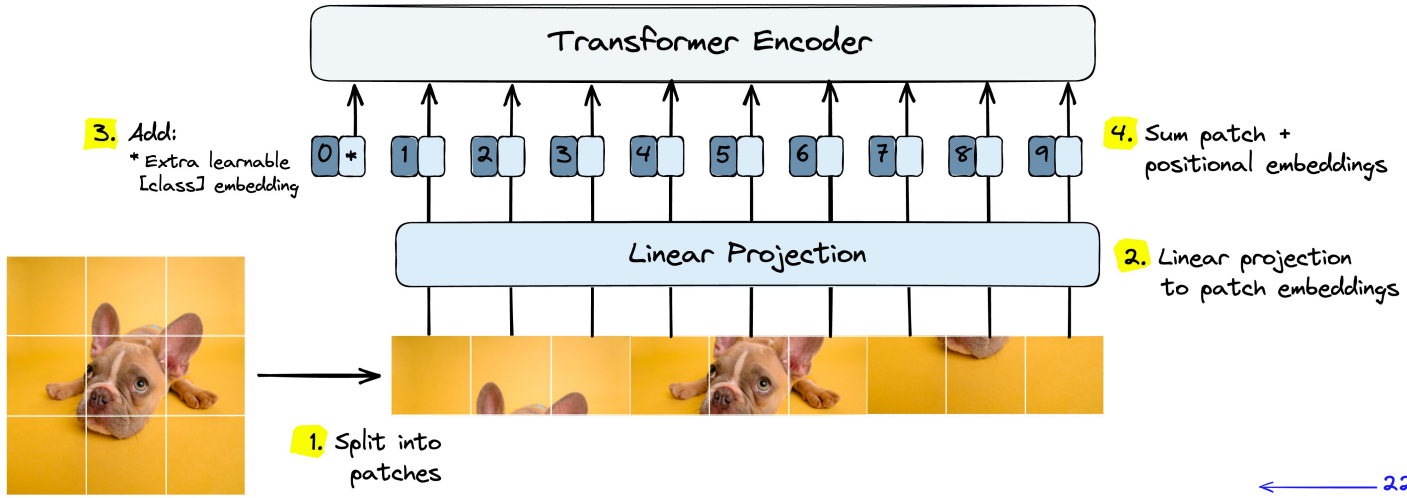
Note: From here everything is just the same as a standard transformer

(7) With the only difference being, instead of a decoder the output from the encoder is passed directly into a Feed Forward Neural Network to obtain the classification output.

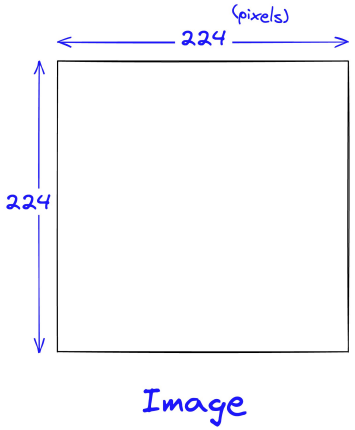
Vision Transformer structure

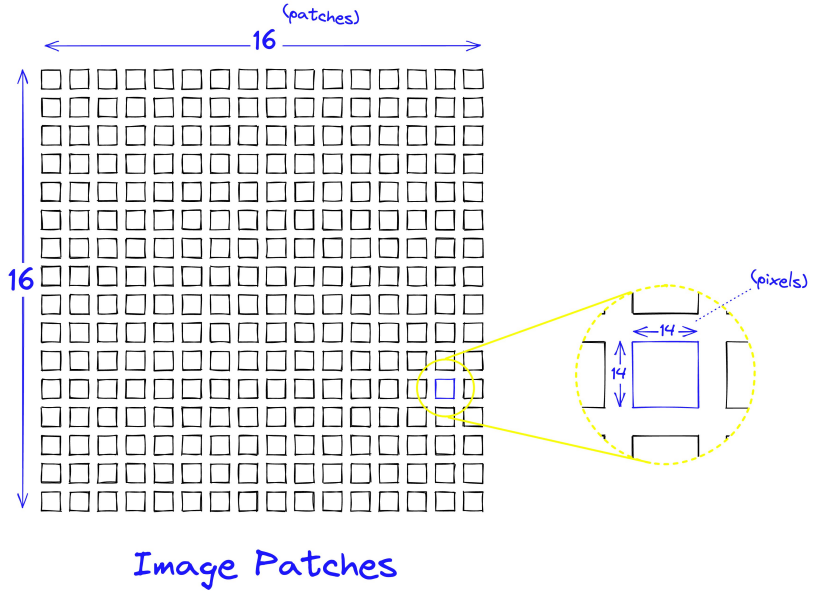
- **Image Patching**:
  - Divide the input image into small fixed-size patches (e.g., 16x16 pixels per patch).
  - Flatten each patch to a 1D vector.
  - These flattened vectors serve as the input "tokens" for the Transformer, and this is done by an “image_processor” similar to a “tokenizer”.
- **Embedding the Patches**:
  - Project (or embed) each flattened patch into a specified dimension using a linear transformation. This is analogous to word embeddings in NLP.
  - This gives the model the ability to learn richer representations for each patch.
- **Positional Encoding**:
  - Transformer do not have an inherent notion of the order of input data, positional encodings (like in NLP Transformers) are added to the patch embeddings.
  - This ensures that the model knows the relative positions of patches in the image.
- **Transformer Encoder**:
  - Feed the sequence of embedded patches (with positional encodings) into a stack of Transformer encoder layers.
  - The Transformer layers allow each patch to attend to all other patches, making the representation of each patch context-aware.
- **Classification Head**:
  - After passing through the Transformer layers, the representation corresponding to the beginning of the sequence, is given..
  - This representation is passed through a feed-forward neural network to produce the final classification output.

Image patching – What is it?



- Just like in NLP we use the model's tokenizer, or ViT models we use the model's "image_processor()".
- The image_processor ensures that your custom images are correctly resized, normalized, and converted into the specific tensor format that the pre-trained model was originally trained on and expects as input.
- Without this step, the model's performance would be severely impacted because its learned weights are tuned to a very specific input data distribution.





Embedding after patching

- The Vision Transformer's first layer, acts as a linear projection that takes each 2D image patch and converts it into a 1D, high-dimensional vector.
- The embeddings are a direct product of the model's architecture and the knowledge it acquired during its initial, extensive pre-training.
- When you load a model using" from_pretrained", like google/vit-base-patch16-224, you are downloading and initializing the model's entire set of weights and biases that were learned from a vast dataset, such as ImageNet-21k.
- The embedding layer's weights are part of this set, meaning they are already optimized to create meaningful embeddings for a wide range of images.
- In essence, if we extract the output of the last hidden layer of the pretrained model, we can look at those embeddings. It is usually of a size of 768.
- So now comes the big question – if we extract the embeddings of the image of a dog and that of a cat, they must be different or dissimilar and therefore if we do a k-means clustering, we should see 2 clusters.
- Therefore, the embeddings you extract are not just random numbers; they are a direct output of the model's learned ability to represent visual information in a high-dimensional vector space. In this space, the model places embeddings that are semantically or visually similar closer together.
- Patches of the Same Object Across Different Images: If you have multiple images of cats, the embeddings for patches of cat fur, eyes, or ears would likely be more similar to each embeddings of the same features in another cat image than they would be to patches of grass or sky in the same image.

Let us check out if everything we discussed about embeddings is true

What does self-attention in ViT do?

- After the patches are converted into embedding vectors
  - the positional encodings are added, so the model can understand the spatial relationships of the patches,
  - the resulting sequence of vectors is then fed into a stack of Transformer Encoder layers,
  - here the multi-head self-attention mechanism operates.
- Within each of the stacked Transformer encoder layers,
  - the self-attention mechanism calculates a set of **"attention scores" that describe the relationship between every pair of patches in the image**.
  - This is done by comparing their vector embeddings to determine how similar or relevant they are to each other, based on their semantic and contextual meaning.
  - For example, a patch containing a dog's nose might have a high attention score (or alignment score) with a patch of its eye or ear, demonstrating the model's understanding that these pieces belong to the same object.
  - This mechanism is what allows Vision Transformers to capture long-range dependencies and global context across the entire image, distinguishing them from traditional convolutional neural networks that are limited to local receptive fields.
  - The attention scores in ViT's are not exclusively about spatial similarity; rather, they are a sophisticated blend of both spatial and semantic similarity.

Finaly – the classification layer

- **The Classification Token ():**
  - During the patching and embedding process, an additional `197$^{th}$ `, learnable token is prepended to the sequence.
  - This token acts as a placeholder that gathers information from all the other patches as the sequence moves through the encoder layers.
  - Its final output vector is used as a summary representation of the entire image for the classification head to make a prediction.
- The classifier's job is straightforward:
  - It takes the 768-dimensional vector of the final token as its input.
  - It projects this vector into an output vector whose size is equal to the number of classes in the task (e.g., 1,000 for ImageNet, 10 for CIFAR-10, or 2 for a cat vs. dog classifier).
  - A final activation function, typically SoftMax, is applied to this output to produce a set of class probabilities.

Example:
ViT with pretrained models

How to train ViT on custom dataset?

1. Install Vision Transformer dependencies.

2. Load and preprocess your dataset. Ensure that the images are resized to match the input size expected by the Vision Transformer (e.g., 224x224 for ViT models with 224 input size).

3. Normalize the images using the same mean and standard deviation used during the pre-training of the ViT model. Create data loaders for your training and validation datasets.

4. Choose a pre-trained model and load it.

5. To perform feature extraction, freeze the transformer layers so that their weights do not update during training. This way, you use the pre-trained features and only train the final classifier layers.

6. Replace the final classification layer with a new one that matches the number of classes in your specific task.

7. Since only the classification head is trainable, set up the optimizer to update only the weights of this layer.

8. Use a loss function suitable for your task, such as CrossEntropyLoss for classification tasks.

9. Train and Evaluate the Vision Transformer on a test image.

10. (Optional) Fine-Tune the Entire Model: After feature extraction, if needed, unfreeze some or all of the transformer layers and fine-tune the entire model on your dataset.

End of Lesson