INFO 4000
Informatics III

Data Science Specialization - Advanced

Transfer Learning continued & Transformers

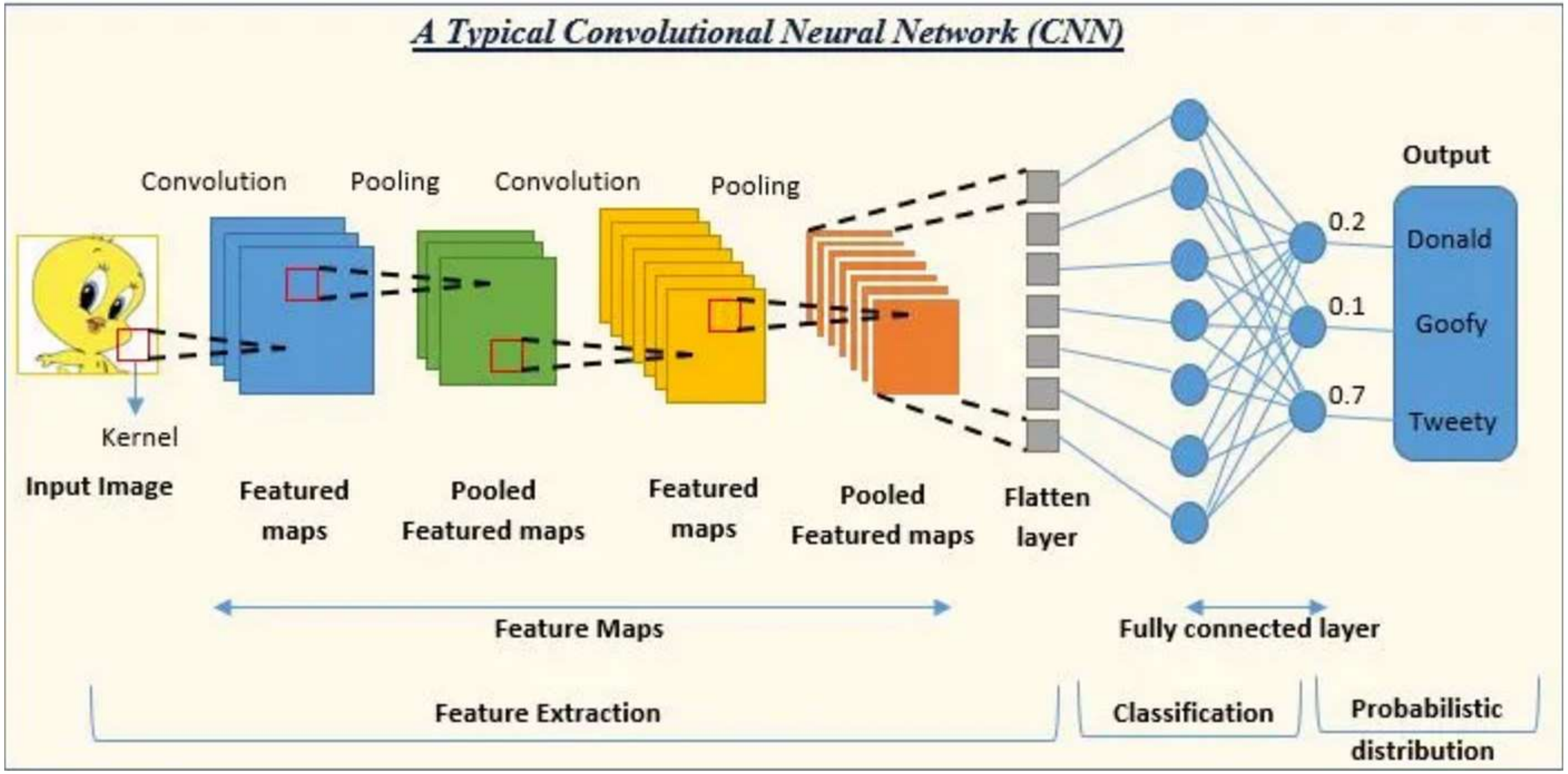
Course Instructor

Jagannath Rao

raoj@uga.edu

Recap

What is transfer learning?



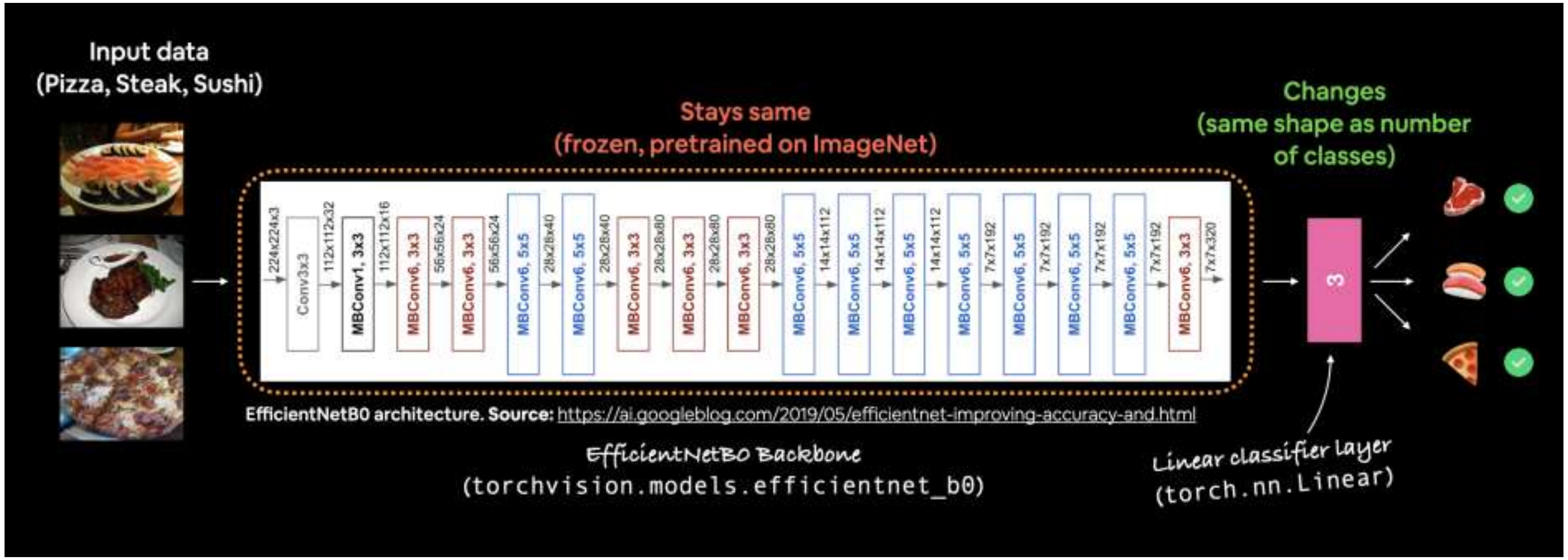
2 ways to use pretrained models

- **ConvNet as fixed feature extractor**: Here, we will freeze the weights for all of the network except that of the final fully connected layer. This last fully connected layer is replaced with a new one with random weights and only this layer is trained.

- **Finetuning the convnet**: Instead of random initialization, we initialize the network with a pretrained network, like the one that is trained on “Imagenet” 1000 dataset. Rest of the training looks as usual.

Setting up a pretrained model

- freeze some base layers of a pretrained model (typically the features section) and
- then adjust the output layers (also called head/classifier layers) to suit your needs.
- In the picture below, we freeze all the layers/parameters in the features section of this efficientnet_bo model. Freeze means the weights are not trainable OR as we know it “require_grad = False”.
- When using a pretrained model, it's important that **your custom data going into the model is prepared in the same way as the original training data that went into the model**.



Transformers

Before diving in – libraries to install in your virtual environment

- pip install transformers
- pip install accelerate
- pip install evaluate
- pip install datasets

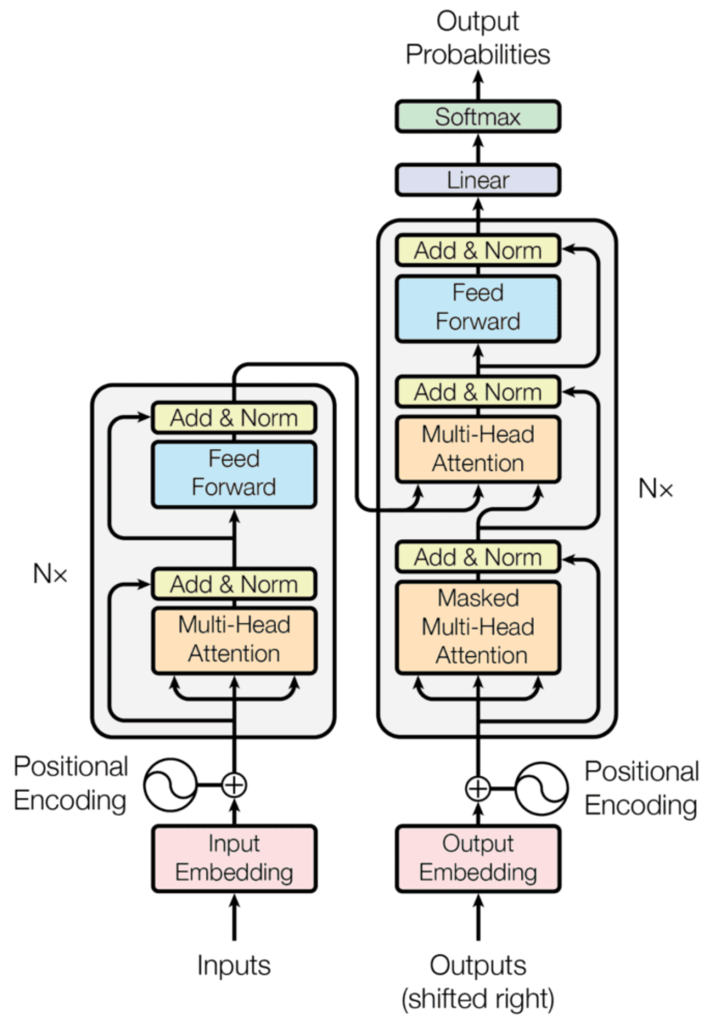
This should suffice but sometimes you may be prompted to install an additional package depending on how your virtual environment has been built.

What is a Transformer?

- A transformer model is a neural network that learns context and thus, meaning, by tracking relationships in sequential data like the words in this sentence.
- First described in a 2017 paper from Google called “Attention is all you need”, transformers are among the newest and one of the most powerful classes of models invented to date.
- Any application using sequential text, image or video data is a candidate for transformer models.
- Transformers are in many cases replacing convolutional and recurrent neural networks (CNNs and RNNs), the most popular types of deep learning models just a few years ago.
- RNNs / LSTMs work in a sequential manner (words coming in time steps) and hence do not lend themselves to parallel processing.
- Transformers on the other hand, process the words in parallel (as they develop an understanding for the context of the text) and hence can-do parallel processing. Makes them faster to train and they function in near real time.
- Also, as a result, there are no issues like vanishing gradients.

Transformer architecture

- The Transformer architecture follows an **encoder-decoder** structure but does not rely on recurrence and convolutions to generate an output.
- The task of the encoder, on the left half of the Transformer architecture, is to map an input sequence to a sequence of continuous representations, which is then fed into a decoder.
- The decoder, on the right half of the architecture, receives the output of the encoder together with the decoder output at the previous time step to generate an output sequence.



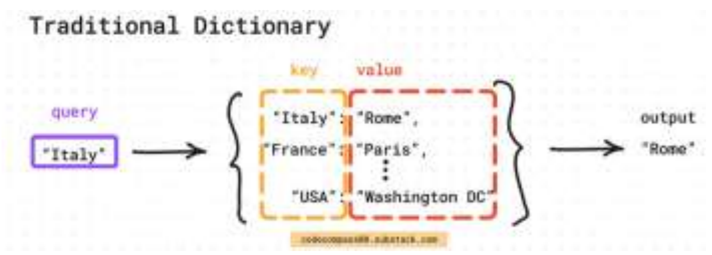
“Attention Mechanism” – Intuitive Understanding

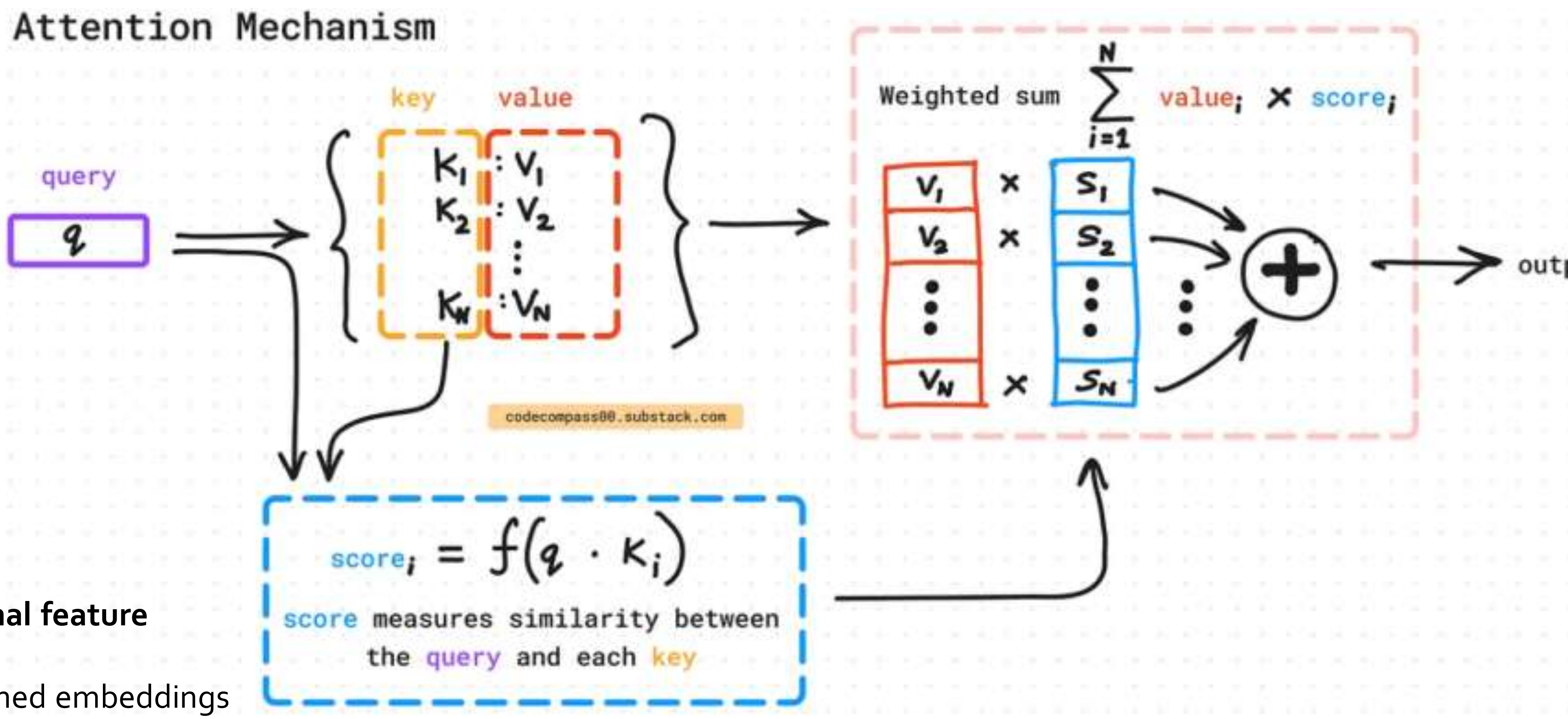
- While understanding the meaning of the word "bank" in a sentence, your brain refers to other parts of the sentence or paragraph to determine whether "bank" means a financial institution or the side of a river.
- This act of referring and determining the importance of previous words is analogous to the attention mechanism.
- In the attention mechanism, every word in a sentence can focus on every other word, including itself.
- This focus is quantified by weights (often called attention scores). A higher weight means the word is paying more "attention" to another specific word.
- For example, for the sentence "He put money in the bank," when trying to understand the context of the word "bank," the model might assign higher attention scores to "money" and "put" to infer that it's about a financial institution and not the river's side.
- Using these attention scores, the model computes a weighted combination of all input words, generating a new representation for each word. This new representation carries the context.
- For the word "bank" in our example, its new representation will carry more context from "money" and "put" due to the higher attention weights.

“Multi-Head Attention” Mechanism

- In Transformer models, this attention mechanism is not performed just once. It's done multiple times in parallel, each with different learned parameters, allowing the model to focus on different aspects of the input.
- These parallel operations are called "heads" in multi-head attention.
- For instance, one "head" might focus on syntactic aspects (the structure of the sentence) while another might focus on semantic aspects (the meaning of the words).
- Combining these multiple perspectives allows the model to generate richer context.
- Attention allows deep learning models to dynamically focus on different parts of the input data, making them particularly powerful for tasks where context from various parts of the input is crucial, like in NLP.
- The Transformer architecture utilizes this mechanism to great effect, leading to state-of-the-art performance in many tasks.
- Transformer needs a way to consider the position of words in a sequence. Positional encoding is a method to give the model information about the position of words in a sequence.
- Once the positional information is added, the modified embeddings are passed through the attention mechanism.
- Now, when the model calculates attention scores, it has access to both word meaning and position, allowing it to generate context-aware representations of words based on their content and order in the sequence.

‘attention’ mechanism math





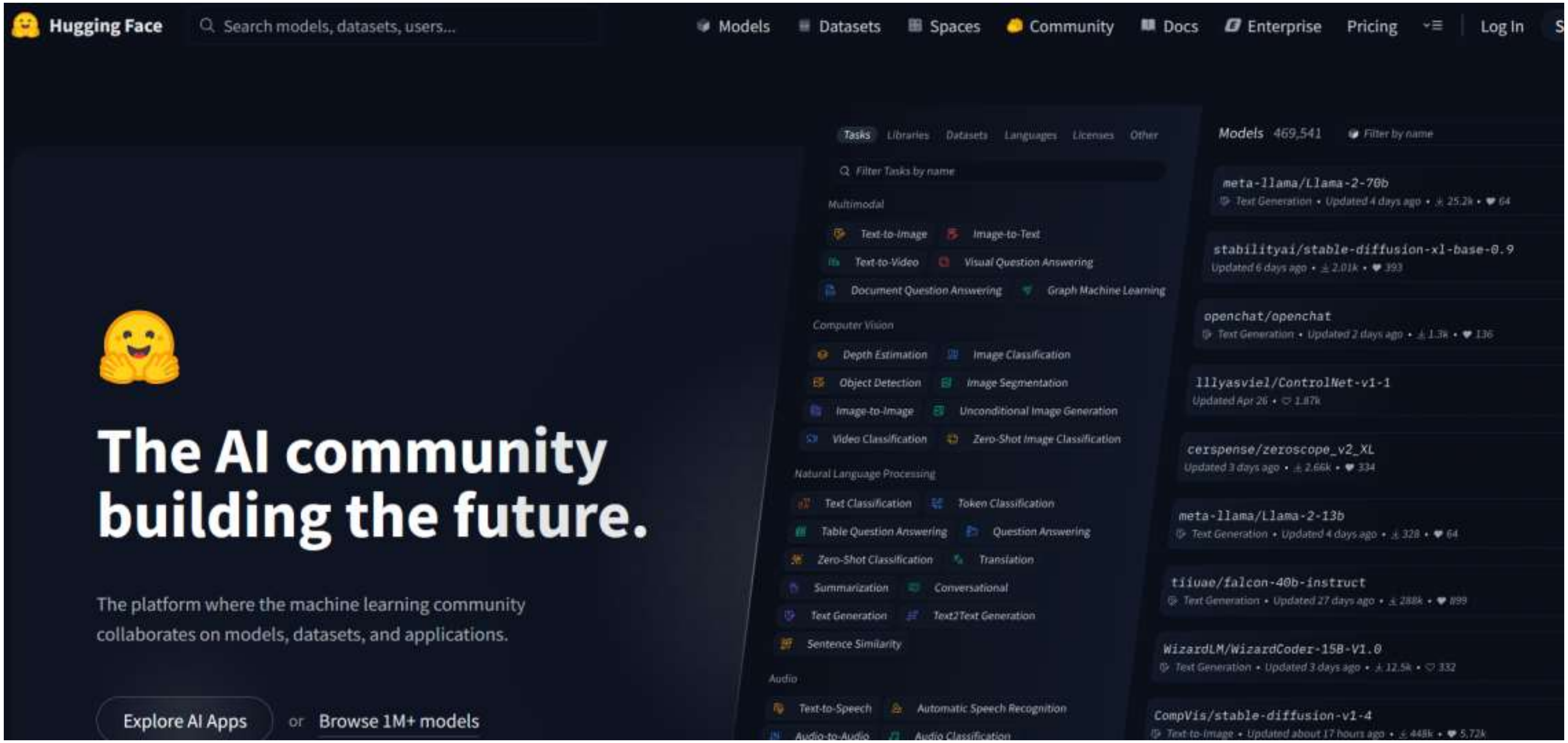
Attention Mechanism
query
q
key
value
$K_1 : V_1$
$K_2 : V_2$
$K_N : V_N$
codecompass00.substack.com
Weighted sum $\sum_{i=1}^{N}$ value$_i$ × score$_i$
V_1 × S_1
V_2 × S_2
V_N × S_N
outp
score$_i$ = $f(q \cdot K_i)$
score measures similarity between the query and each key

- Here, the **query**, **keys**, and **values** are **D-dimensional feature vectors**.
- These can be outputs of previous layers or the learned embeddings of the encoded tokens.
- The matrices contain learnable parameters that are optimized during training.

Coded Example to understand the “attention” mechanism

HuggingFace (HF): Transformer pretrained models

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



The AI community building the future.

The platform where the machine learning community collaborates on models, datasets, and applications.

meta-llama/Llama-2-70b
Text Generation • Updated 4 days ago • 25.2k • 64

stabilityai/stable-diffusion-xl-base-0.9
Updated 6 days ago • 2.01k • 393

openchat/openchat
Text Generation • Updated 2 days ago • 1.3k • 136

lllyasviel/ControlNet-v1-1
Updated Apr 26 • 1.87k

cerspense/zeroscope_v2_XL
Updated 3 days ago • 2.66k • 334

meta-llama/Llama-2-13b
Text Generation • Updated 4 days ago • 328 • 64

tiiuae/falcon-40b-instruct
Text Generation • Updated 27 days ago • 288k • 899

WizardLM/WizardCoder-15B-V1.0
Text Generation • Updated 3 days ago • 12.5k • 332

CompVis/stable-diffusion-v1-4
Text-to-Image • Updated about 17 hours ago • 448k • 5.72k

“HuggingFace” models and functions

- HuggingFace is a large open-source community that builds tools to enable users to build, train, and deploy machine learning models based on open-source code and technologies.

- HuggingFace makes it easy to share tools, models, model weights, and datasets, between other practitioners, via its toolkit.

- HuggingFace provides state-of-the-art models for different tasks. It has a vast number of pre-trained models for different tasks.

- HuggingFace is famous for its contribution to the NLP domain. The NLP tasks are:
    - Text classification
    - Text generation
    - Translation
    - Summarization
    - Fill-mask
    - Question-Answering
    - Zero-shot classification
    - Sentence similarity

- The computer vision tasks are as follows:
    - Image classification
    - Image segmentation
    - Object detection

“HuggingFace” models and functions.. contd

- The audio tasks are as follows:
    - Speech recognition
    - Text-to-speech
    - Automatic Speech recognition
    - Audio classification
    - Sound / event detection
- HuggingFace, the Transformers library, allows us to use these models in a way that abstracts unnecessary details.
- The Datasets library by HuggingFace provides us the facility to load their datasets (1000’s), as well as our own datasets.

NLP with pre-trained transformer models

Review of NLP and its tasks

The aim of NLP tasks is not only to understand single words individually, but to be able to understand the context of those words.

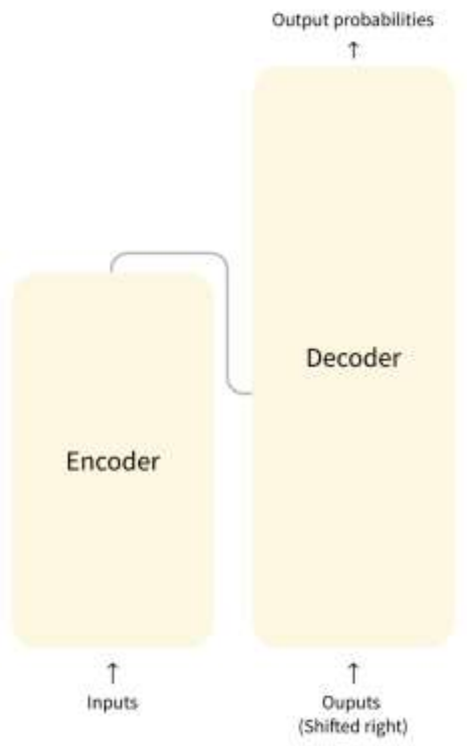
- **Classifying whole sentences**: Getting the sentiment of a review, detecting if an email is spam, determining if a sentence is grammatically correct or whether two sentences are logically related or not
- **Classifying each word in a sentence**: Identifying the grammatical components of a sentence (noun, verb, adjective), or the named entities (person, location, organization)
- **Generating text content**: Completing a prompt with auto-generated text, filling in the blanks in a text with masked words
- **Extracting an answer from a text**: Given a question and a context, extracting the answer to the question based on the information provided in the context
- **Generating a new sentence from an input text**: Translating a text into another language, summarizing a text
- NLP also tackles complex challenges in speech recognition and computer vision:
  - Generating a transcript of an audio sample
  - A description of an image.
- Transformer models are used to solve these kinds of NLP tasks.

Transformer architecture in the NLP context

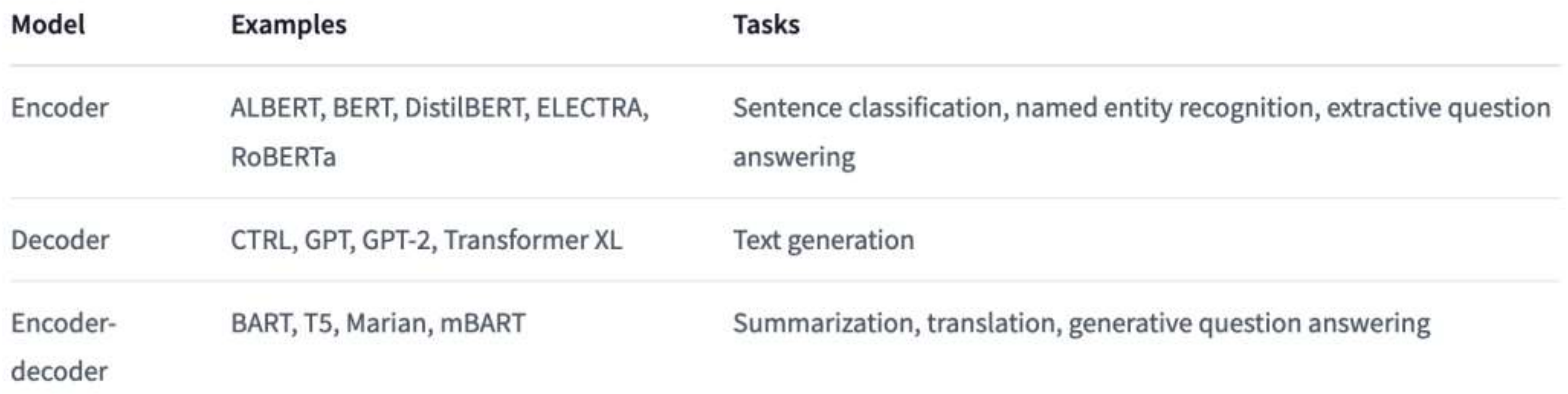
- **Encoder (left)**:
  - The encoder receives an input and builds a representation of it (its features).
  - This means that the model is optimized to acquire understanding from the input.
- **Decoder (right)**:
  - The decoder uses the encoder's representation (features) along with other inputs to generate a target sequence.
  - This means that the model is optimized for generating outputs.

Each of these parts can be used independently, depending on the task:

- **Encoder-only models**: Good for tasks that require understanding of the input, such as sentence classification and named entity recognition.
- **Decoder-only models**: Good for generative tasks such as text generation.
- **Encoder-decoder models** or **sequence-to-sequence models**: Good for generative tasks that require an input, such as translation or summarization.



Some of the available models



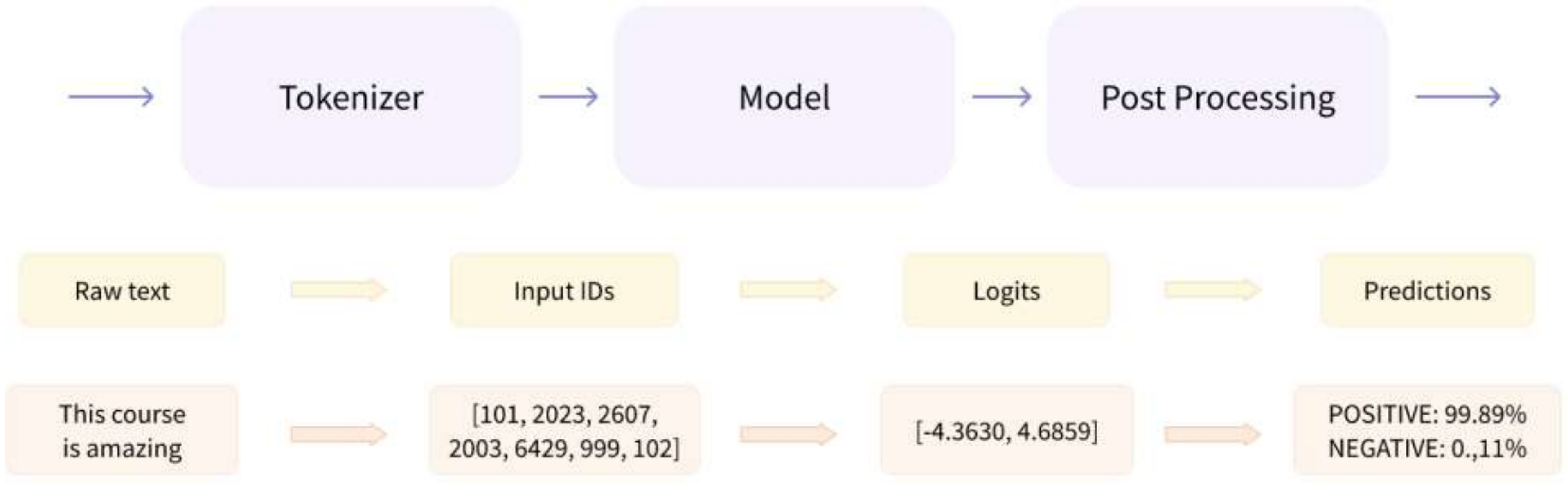
Model	Examples	Tasks
Encoder	ALBERT, BERT, DistilBERT, ELECTRA, RoBERTa	Sentence classification, named entity recognition, extractive question answering
Decoder	CTRL, GPT, GPT-2, Transformer XL	Text generation
Encoder-decoder	BART, T5, Marian, mBART	Summarization, translation, generative question answering

Template for using the HF models

Terminology - Architecture, checkpoints and model

- **Architecture**: This is the skeleton of the model — the definition of each layer and each operation that happens within the model.
- **Checkpoints**: These are the weights that will be loaded in a given architecture.
- **Model**: This is an umbrella term that isn't as precise as "architecture" or "checkpoint": it can mean both.
- As an example:
  - "BERT" is an architecture
  - "bert-base-cased", a set of weights trained by the Google team for the first release of BERT, is a checkpoint.
  - One can say, "the BERT model" and "the bert-base-cased model."
- **Caution:** Depending on how these models have been trained (often on large corpus of text scraped from the web), they can have intrinsic, gender, race or other biases. When we us these pretrained models for fine-tuning, the biases do not go away.

The process in building NLP apps with pre-trained models



The pre-trained model

- There are models for the different types of problems you want to solve.
- As n example, for text classification you may import the generic – *“AutoModelForSequenceClassification”*
- The specific *“AutoModelFor…”* class you use will depend on the task you're performing and there are a few.
- However, this import statement only gives you the genre of the task you are solving but you still have to associate it with a specific pretrained model that is available for that genre.
- As an example, you may want to use a “Bert” model or a “GPT-2” model for the LLM task and so on.
- The statement will look like this example:
  - *# Load the pre-trained model (using BERT for sequence classification)*
  - *model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased")*

Tokenizer

- A ***'tokenizer'*** does the splitting the input into words, sub words, or symbols (like punctuation) that are called *tokens* Mapping each token to an integer.

- Adding additional inputs that may be useful to the model (like start or end of sentences)

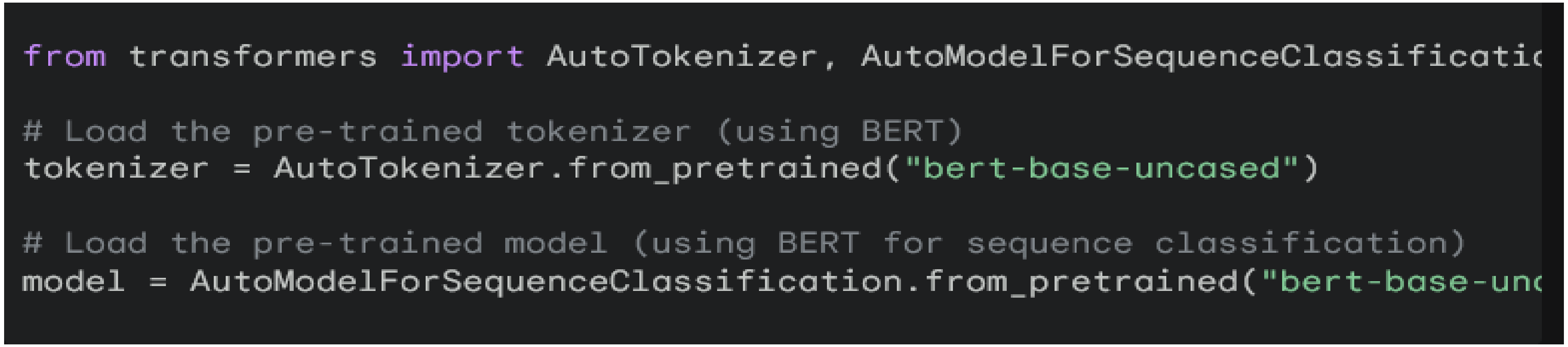
- All this preprocessing needs to be done in exactly the same way as when the model was pretrained:

  - so, we first need to download ***AutoTokenizer*** class (*from transformers import AutoTokenizer*) and

  - Specifically pick the preprocessing method used for the pre-trained model by associating it with it.

  - Using the name of our model, it will automatically fetch the data associated with the model's tokenizer method

The starter template for any NLP task



```
from transformers import AutoTokenizer, AutoModelForSequenceClassificatic

# Load the pre-trained tokenizer (using BERT)
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")

# Load the pre-trained model (using BERT for sequence classification)
model = AutoModelForSequenceClassification.from_pretrained("bert-base-unc
```

- *"bert-base-uncased"* for both the tokenizer and the model.
- We can replace *"bert-base-uncased"* with other model names depending on your specific task and requirements.
- The model's name is what is referred to as “checkpoint”
- Refer to the Hugging Face model hub for a comprehensive list of available pre-trained models and their corresponding names.

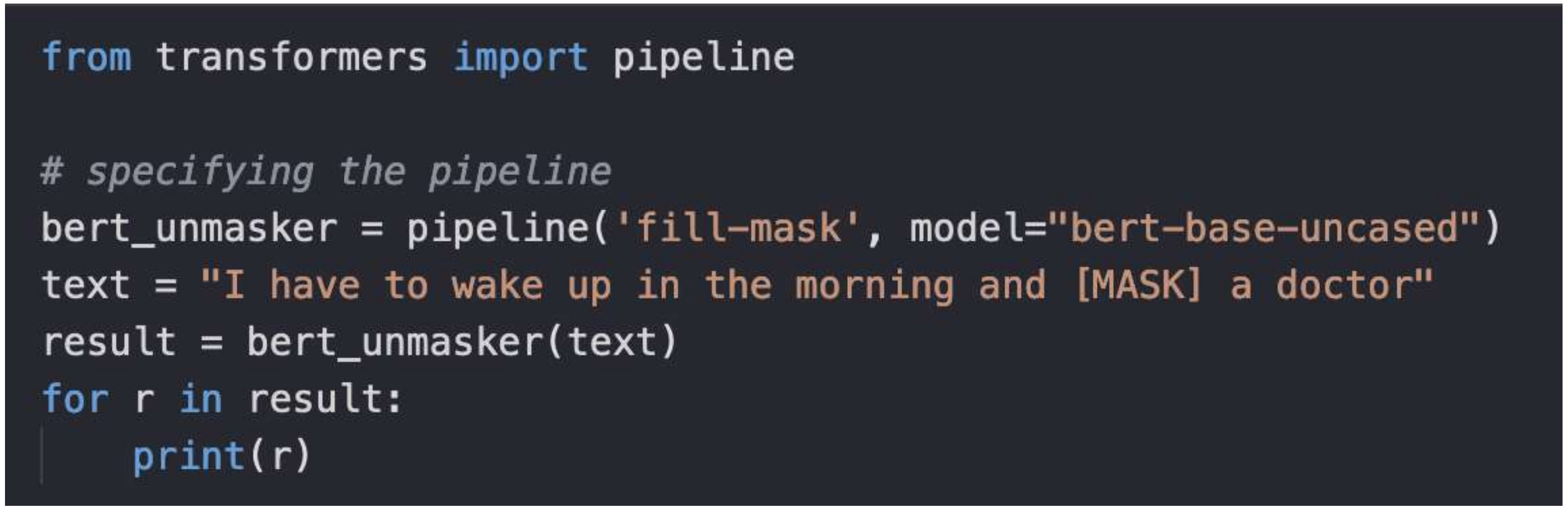
Inferencing:
The HF Pipeline Class

Pipeline() for inferencing / predicting

- The very useful module in the Transformers library is the Pipeline() Class.
- **It is and can be used only for predicting or inferencing function using a pre-trained /trained model.**
- It connects a model with its necessary preprocessing and postprocessing steps, allowing us to directly input any text and get an intelligible answer .
- There are three main steps involved when you pass some text to a pipeline:
    - The text is preprocessed into a format the model can understand.
    - The preprocessed inputs are passed to the model.
    - The predictions of the model are post-processed, so you can make sense of them.
- Some of the available pipelines are (Examples):
    - feature-extraction (get the vector representation of a text)
    - fill-mask
    - NER (named entity recognition)
    - question-answering
    - sentiment-analysis
    - summarization
    - text-generation
    - translation
    - zero-shot-classification

Example of use of the Pipeline function

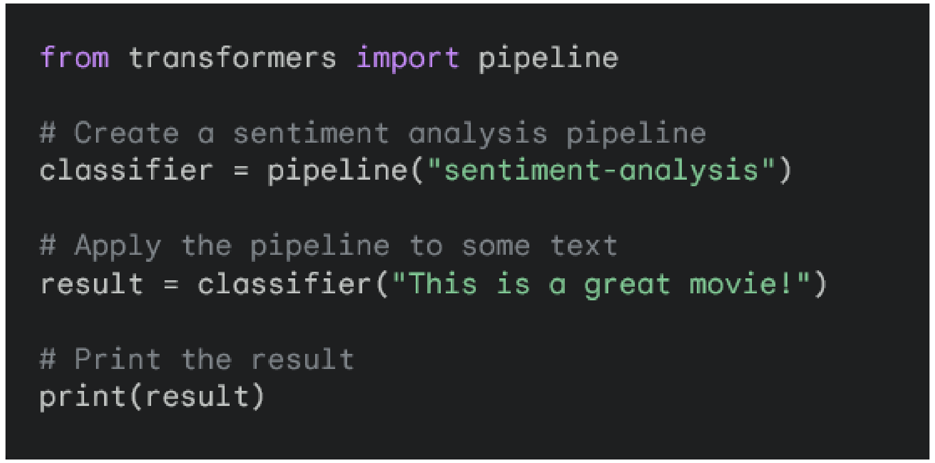
Transformers library to use a pre-trained model to generate predictions for a missing word



```python
from transformers import pipeline

# specifying the pipeline
bert_unmasker = pipeline('fill-mask', model="bert-base-uncased")
text = "I have to wake up in the morning and [MASK] a doctor"
result = bert_unmasker(text)
for r in result:
    print(r)
```

Salient points to remember about the Pipeline Class

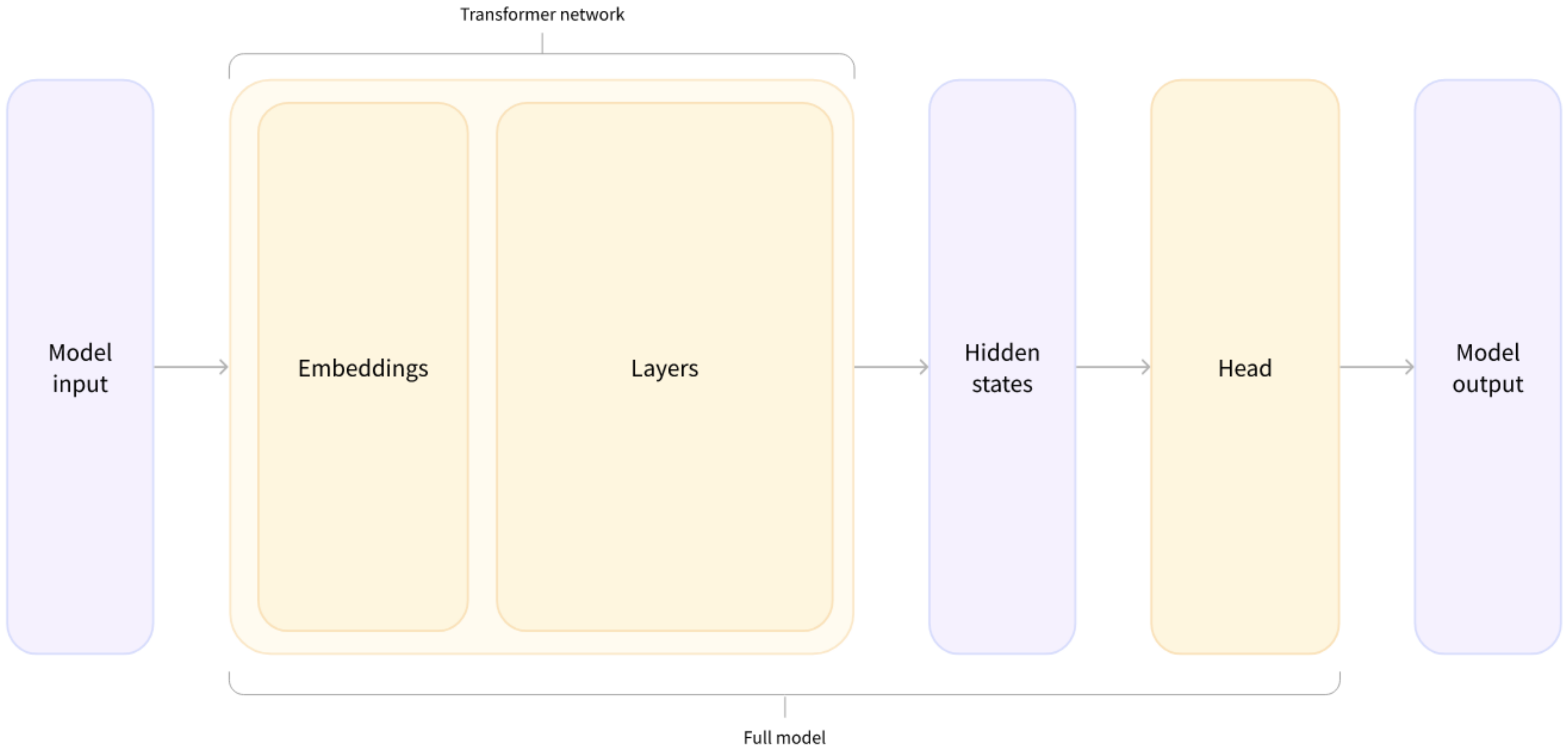
- pipeline class is a powerful and convenient tool for quickly applying pre-trained models to various natural language processing (NLP) tasks.
- The pipeline class abstracts away much of the complexity involved in loading models, tokenizing text, and generating predictions.
- Hugging Face offers a wide range of pipelines for various NLP tasks.
- The pipeline class seamlessly integrates with Hugging Face's vast library of pre-trained models, allowing you to leverage state-of-the-art models without the need for extensive training.
- While convenient for quick experimentation, pipelines also offer options for customization, including specifying the model, tokenizer, and other parameters to tailor the behavior to your specific needs.
- Hugging Face pipeline class is primarily designed for inference (applying a pre-trained model to new data) and not for training.



```python
from transformers import pipeline

# Create a sentiment analysis pipeline
classifier = pipeline("sentiment-analysis")

# Apply the pipeline to some text
result = classifier("This is a great movie!")

# Print the result
print(result)
```

Basic NLP workflow when using Transformer pre-trained models

Visual representation

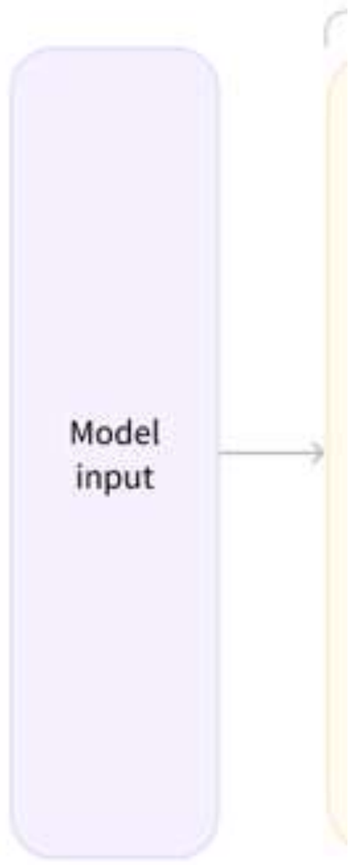


Preparing the inputs to the model

```python
from transformers import AutoTokenizer

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

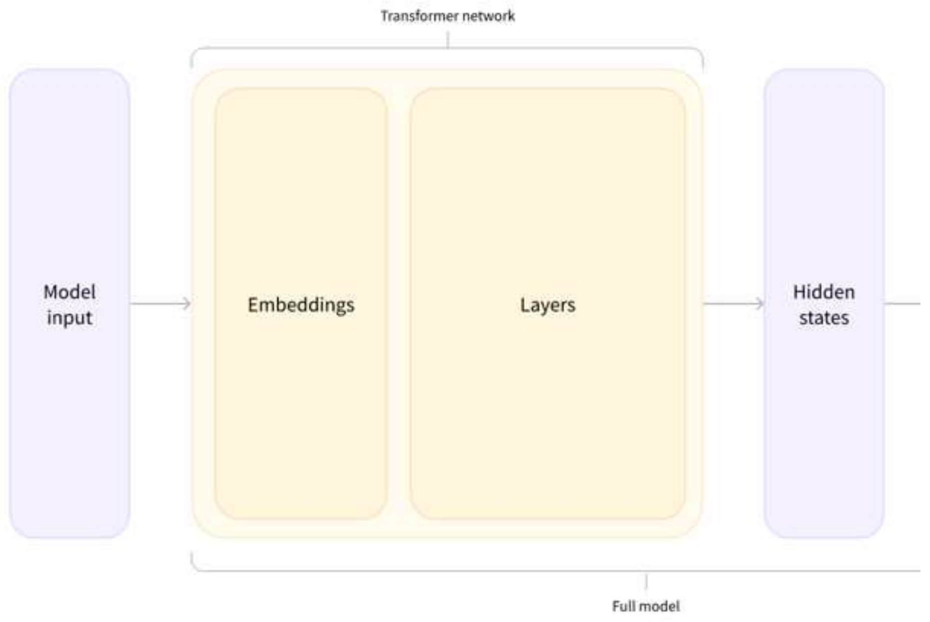
```python
raw_inputs = [
    "I've been waiting for a HuggingFace course my whole life.",
    "I hate this so much!",
]
inputs = tokenizer(raw_inputs, padding=True, truncation=True, return_tensors="pt")
print(inputs)
```

```
{
    'input_ids': tensor([
        [  101,  1045,  1005,  2310,  2042,  3403,  2005,  1037, 17662, 12172, 2607,  2026,  2
        [  101,  1045,  5223,  2023,  2061,  2172,   999,   102,     0,     0,    0,     0,
    ]),
    'attention_mask': tensor([
        [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1],
        [1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0]
    ])
}
```



Feed the model

- We can now feed it to the model.
- We know by now that the Transformers provides an *AutoModel* class which also has a *from_pretrained()* method.
- This architecture contains only the base Transformer module:
  - given some inputs, it outputs what we'll call ***hidden states***, also known as ***features***.
- For each model input, we'll retrieve a high-dimensional vector representing the ***contextual understanding of that input by the Transformer model***.
- While these hidden states can be useful on their own, they're usually inputs to another part of the model, known as the ***head***.
- The vector output by the Transformer module is usually large. It generally has three dimensions:
  - Batch size: The number of sequences processed at a time (2 in our example).
  - Sequence length: The length of the numerical representation of the sequence.
  - Hidden size: The vector dimension of each model input.



To the Transformer ‘Head’

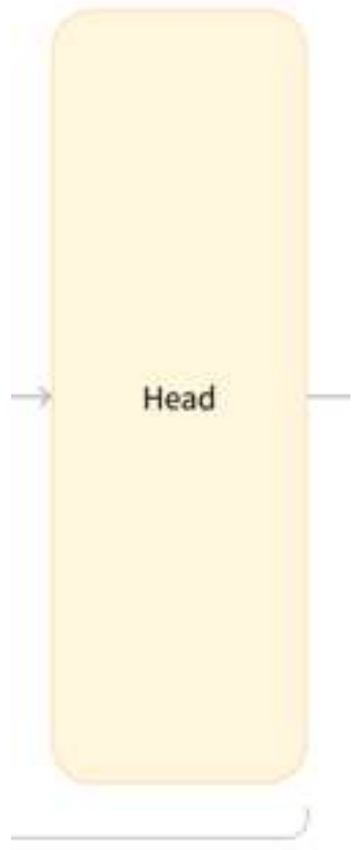
1. The output of the Transformer is sent directly to the model head to be processed.

2. In the context of transformers, the "head" typically refers to a ***self-attention*** head within the ***multi-head attention mechanism***.

3. The core of a transformer model is its self-attention mechanism. This allows the model to weigh the importance of different words in a sequence when processing a particular word.

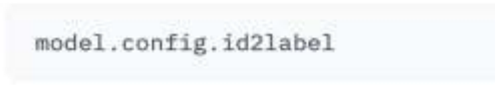
4. These attention scores determine how much each word should "attend" to other words when forming its representation.

5. The outputs from all heads are then combined to produce a final representation for each word that incorporates information from different perspectives.

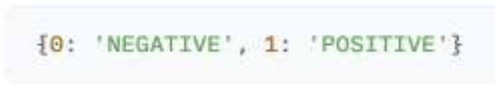


Model Output and post-processing

- The values we get as output from our model don't necessarily make sense by themselves.
- They are not probabilities but ***'logits"*** and hence not normalized.
- To be converted to probabilities, they need to go through a SoftMax layer. Then we would have meaningful scores.
- To get the labels corresponding to each position, we can inspect the id2label attribute of the model config.

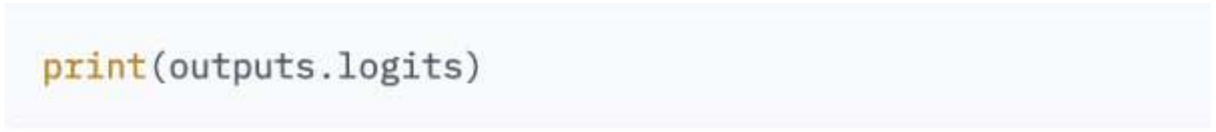


```
model.config.id2label
```

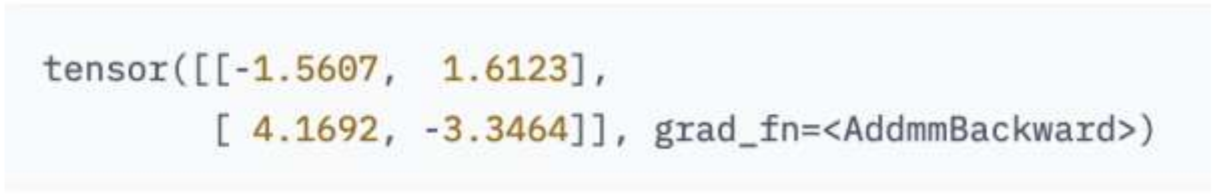


```
{0: 'NEGATIVE', 1: 'POSITIVE'}
```

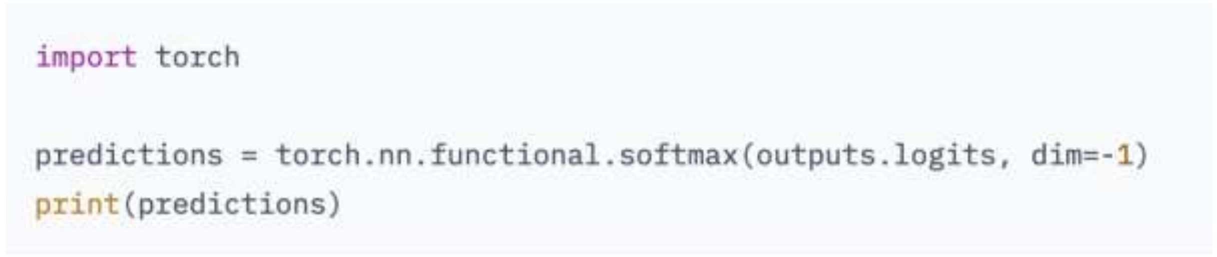
- We have successfully reproduced the three steps of the pipeline: preprocessing with tokenizers, passing the inputs through the model, and postprocessing.



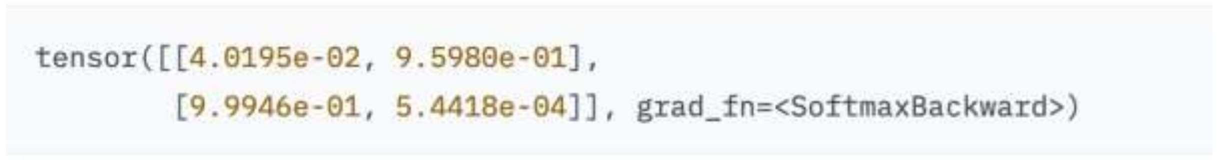
```
print(outputs.logits)
```



```
tensor([[-1.5607,  1.6123],
        [ 4.1692, -3.3464]], grad_fn=<AddmmBackward>)
```



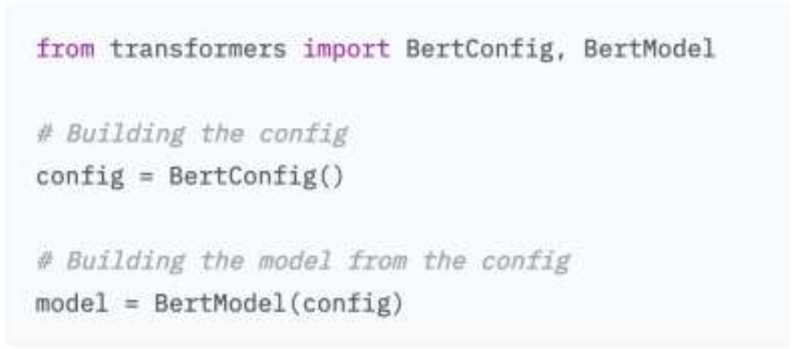
```
import torch

predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
print(predictions)
```



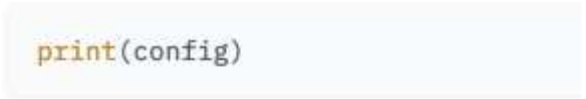
```
tensor([[4.0195e-02, 9.5980e-01],
        [9.9946e-01, 5.4418e-04]], grad_fn=<SoftmaxBackward>)
```

Model

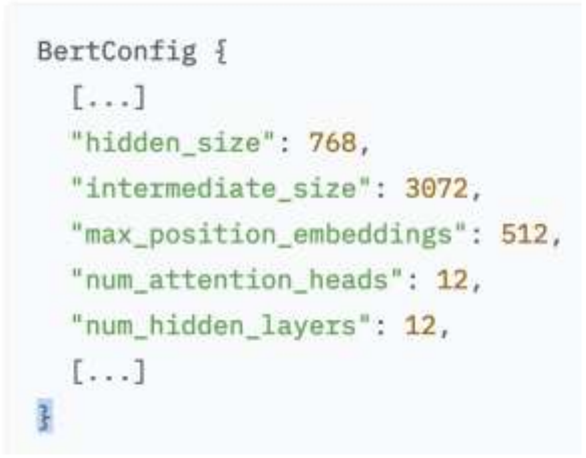
- The AutoModel class, which is handy when you want to instantiate any model from a checkpoint.
- The AutoModel class and all of its relatives are actually simple wrappers over the wide variety of models available in the library.
- It’s a clever wrapper as it can automatically guess the appropriate model architecture for your checkpoint and then instantiates a model with this architecture.
- If you know the type of model you want to use, you can use the class that defines its architecture directly.
- To initialize, for example, “Bert” model, we need to load a configuration object.
- The configuration contains many attributes that are used to build the model.
- However, creating a model from the default configuration initializes it with random values and is no use for predictions (needs training).



```python
from transformers import BertConfig, BertModel

# Building the config
config = BertConfig()

# Building the model from the config
model = BertModel(config)
```



```python
print(config)
```

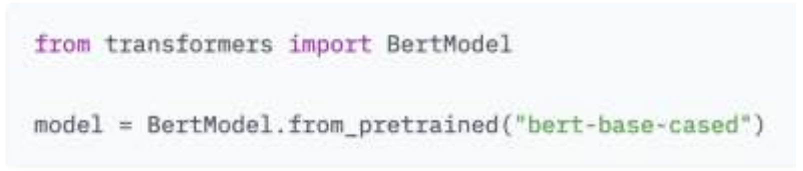


```
BertConfig {
  [...]
  "hidden_size": 768,
  "intermediate_size": 3072,
  "max_position_embeddings": 512,
  "num_attention_heads": 12,
  "num_hidden_layers": 12,
  [...]
}
```

Loading and Saving a Model

Loading a model

- Loading a Transformer model that is already trained is simple — we can do this using the *from_pretrained()* method.
- We could replace" BertModel" with the equivalent "AutoModel class".
- Is good do this as this produces checkpoint-agnostic code; if your code works for one checkpoint, it should work seamlessly with another.
- This applies even if the architecture is different, if the checkpoint was trained for a similar task (for example, a sentiment analysis task).
- In the code sample on the right, we didn't use "BertConfig", and instead loaded a pretrained model via the "bert-base-cased" identifier.
- This is a model checkpoint that was trained by the authors of BERT themselves.
- It can be used directly for inference on the tasks it was trained on, and it can also be fine-tuned on a new task.



```python
from transformers import BertModel

model = BertModel.from_pretrained("bert-base-cased")
```

Saving a model

- Saving a model is as easy as loading one — we use the “save_pretrained()” method.

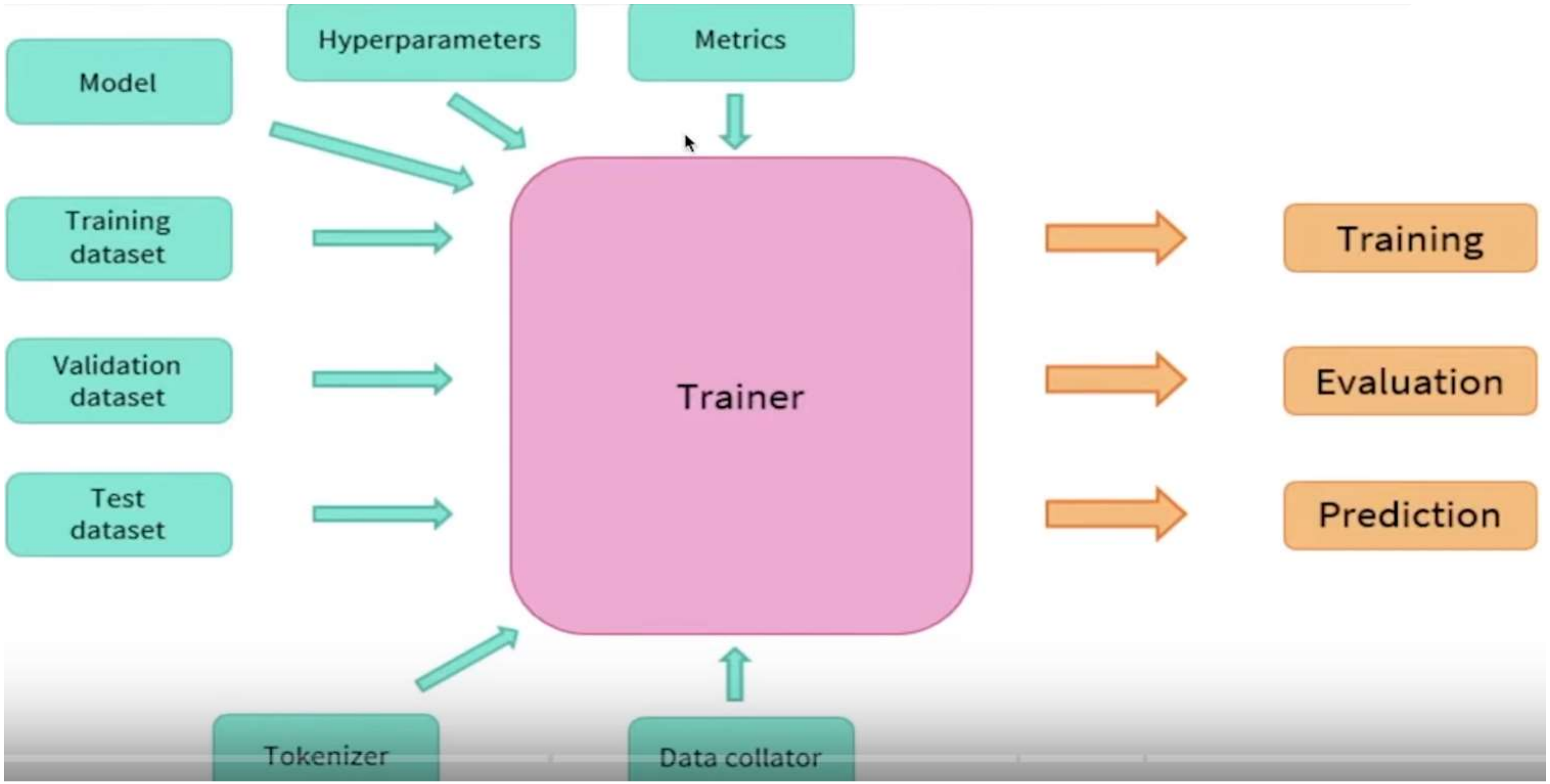
```
model.save_pretrained("directory_on_my_computer")
```

- This saves two files to your disk: ***“config.json”*** and **“pytorch_model.bin”.**
- *config.json* file holds the attributes necessary to build the model architecture.
- The *pytorch_model.bin* file is known as the *state dictionary*; it contains all your model’s weights.
- The two files go hand in hand; the configuration is necessary to know your model’s architecture, while the model weights are your model’s parameters.

Let us understand these details with examples

Fine-Tuning:
HF way using the “trainer()”

Trainer()


Model
Hyperparameters
Metrics
Training dataset
Validation dataset
Test dataset
Trainer
Training
Evaluation
Prediction
Tokenizer
Data collator

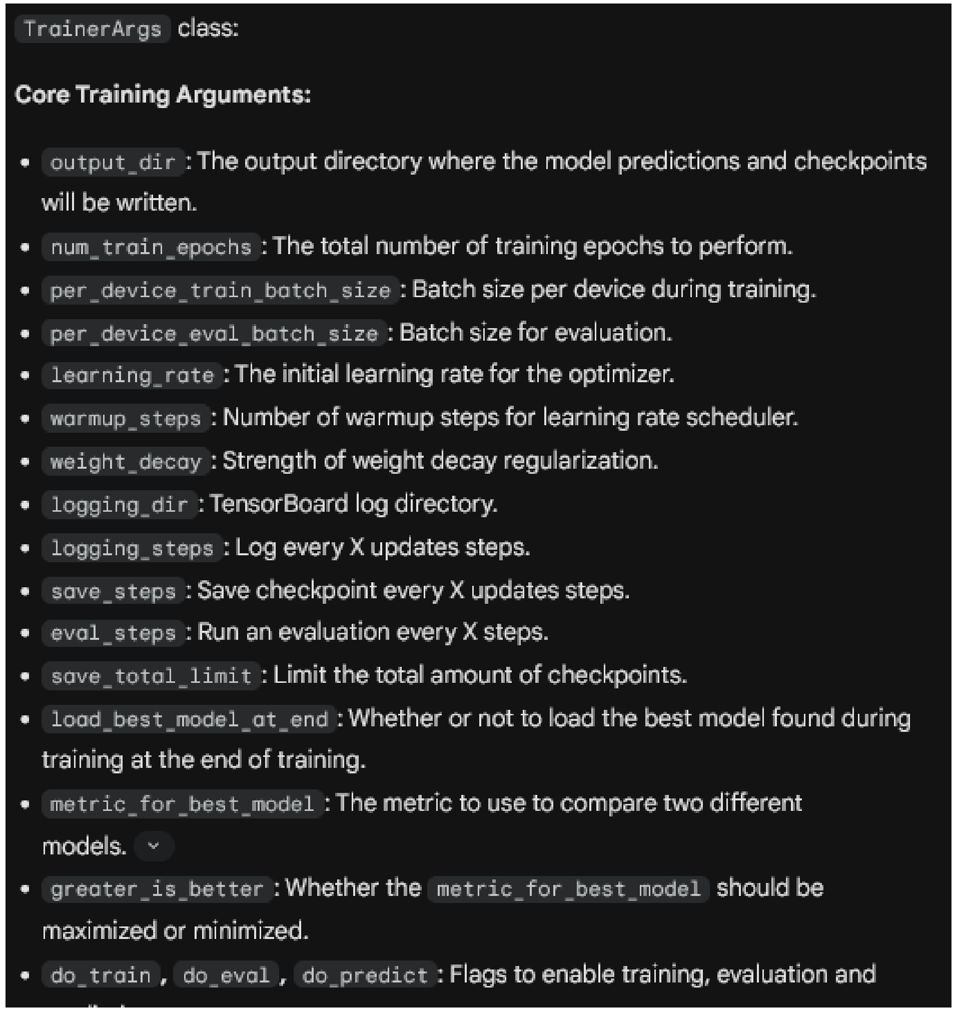
Key characteristics of the Trainer Class

- The Hugging Face Trainer class is a high-level API designed to streamline the training and fine-tuning of models, particularly those in the transformer's library.
- Instead of manually writing a training loop in PyTorch, the Trainer class handles the heavy lifting, allowing users to focus on the model, data, and hyperparameters.
- The Trainer class abstracts away the complexities of a standard training loop, including the forward pass, backward pass, and weight updates. It's a "feature-complete" solution for many common use cases.
- Integration with “TrainingArguments”: The Trainer works in conjunction with the TrainingArguments class, where you can define and customize all your training hyperparameters, such as learning rate, batch size, number of epochs, and output directory.
- Automatic Features: The class automates essential aspects of the training process, including:
  - Evaluation and Logging: It provides methods to evaluate model performance and log training progress, making it easy to monitor metrics like loss and accuracy.
  - Checkpointing: The Trainer can automatically save model checkpoints, which is crucial for resuming training from a specific point or recovering from errors.
- Fine-Tuning with Hugging Face Trainer This video demonstrates how to fine-tune a model using the Hugging Face Trainer class.

Trainer Arguments

- Before instantiating your Trainer, we create a TrainingArguments to access all the points of customization during training.

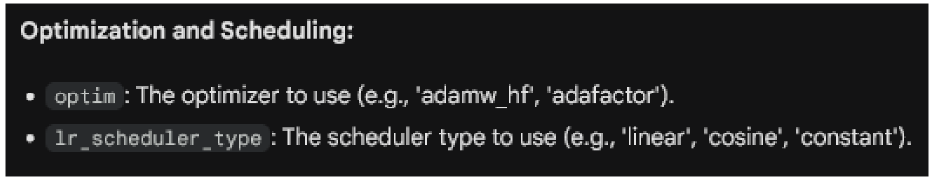
- These are the most used but there can be more.



`TrainerArgs` class:

Core Training Arguments:

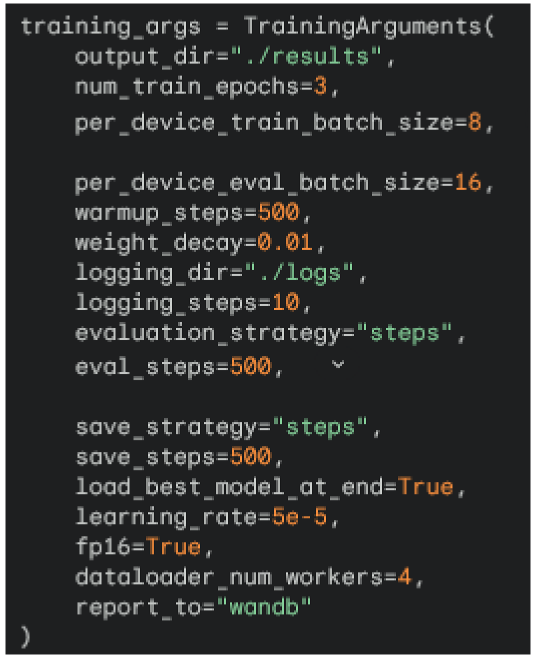
- `output_dir`: The output directory where the model predictions and checkpoints will be written.
- `num_train_epochs`: The total number of training epochs to perform.
- `per_device_train_batch_size`: Batch size per device during training.
- `per_device_eval_batch_size`: Batch size for evaluation.
- `learning_rate`: The initial learning rate for the optimizer.
- `warmup_steps`: Number of warmup steps for learning rate scheduler.
- `weight_decay`: Strength of weight decay regularization.
- `logging_dir`: TensorBoard log directory.
- `logging_steps`: Log every X updates steps.
- `save_steps`: Save checkpoint every X updates steps.
- `eval_steps`: Run an evaluation every X steps.
- `save_total_limit`: Limit the total amount of checkpoints.
- `load_best_model_at_end`: Whether or not to load the best model found during training at the end of training.
- `metric_for_best_model`: The metric to use to compare two different models.
- `greater_is_better`: Whether the `metric_for_best_model` should be maximized or minimized.
- `do_train`, `do_eval`, `do_predict`: Flags to enable training, evaluation and



Optimization and Scheduling:

- `optim`: The optimizer to use (e.g., 'adamw_hf', 'adafactor').
- `lr_scheduler_type`: The scheduler type to use (e.g., 'linear', 'cosine', 'constant').

The code

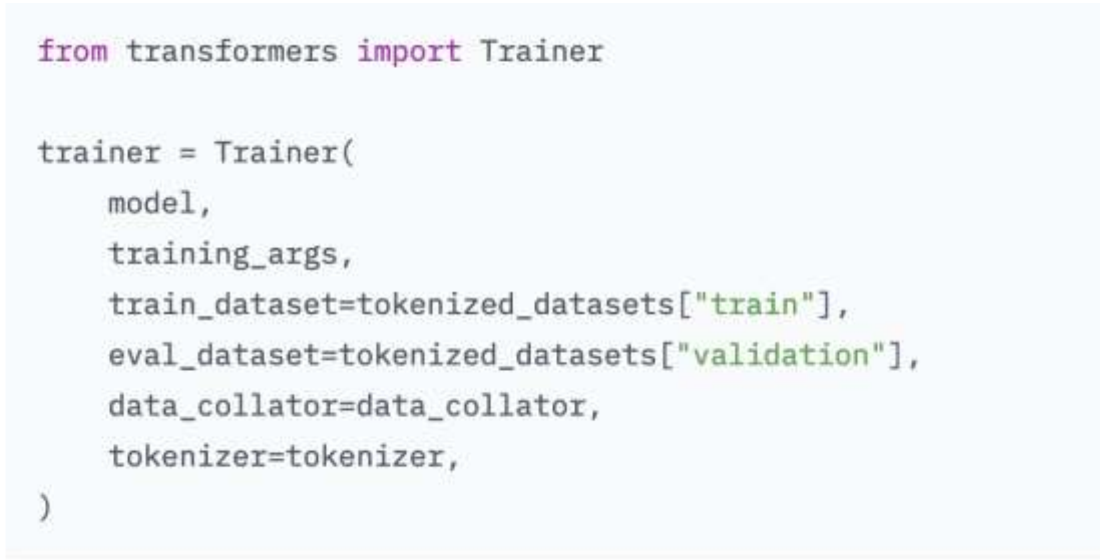


```
training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,
    per_device_train_batch_size=8,

    per_device_eval_batch_size=16,
    warmup_steps=500,
    weight_decay=0.01,
    logging_dir="./logs",
    logging_steps=10,
    evaluation_strategy="steps",
    eval_steps=500,

    save_strategy="steps",
    save_steps=500,
    load_best_model_at_end=True,
    learning_rate=5e-5,
    fp16=True,
    dataloader_num_workers=4,
    report_to="wandb"
)
```

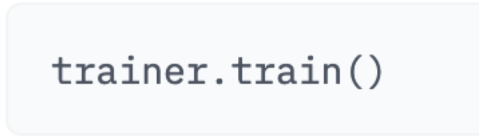
Trainer

- Once we have our model, we can define a Trainer by passing it all the objects constructed up to now :
  - the ***model***,
  - the ***training_args***,
  - the ***training and validation datasets***,
  - our ***data_collator***, and
  - our ***tokenizer***.
  - our ***metrics function***
- To Fine Tune: call the ***trainer***.

When the DataCollator is defined and passed to the Hugging Face Trainer, its primary role is to prepare and format batches of data before they are fed to the model. It takes a list of pre-processed examples from the dataset and "collates" them into a single batch of tensors. This is a crucial step because deep learning models require uniform-sized inputs to be processed efficiently on hardware like GPUs.



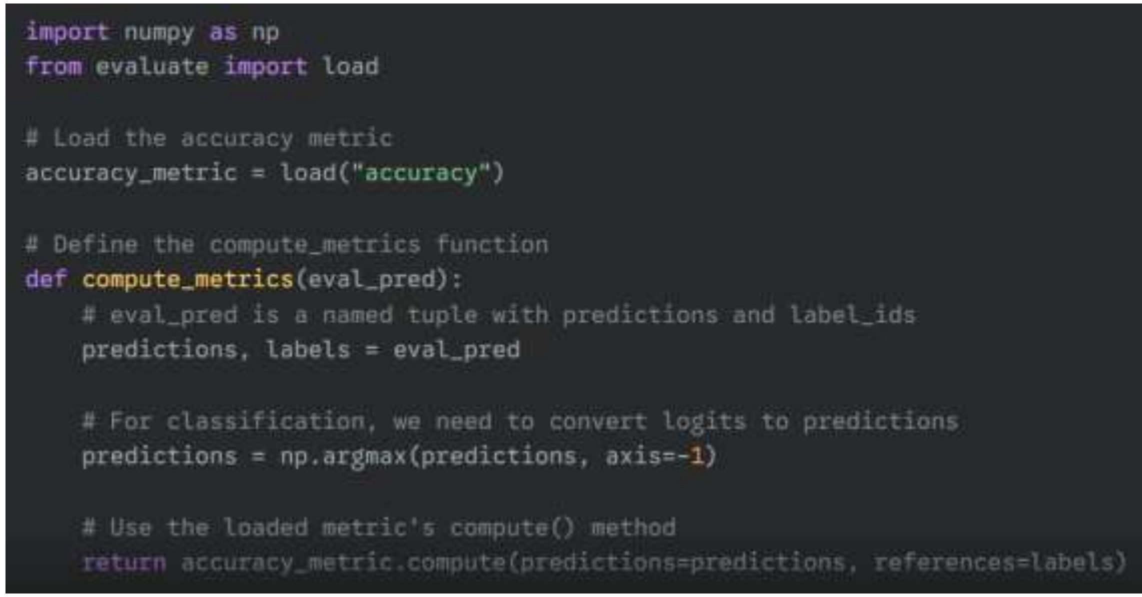
```python
from transformers import Trainer

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



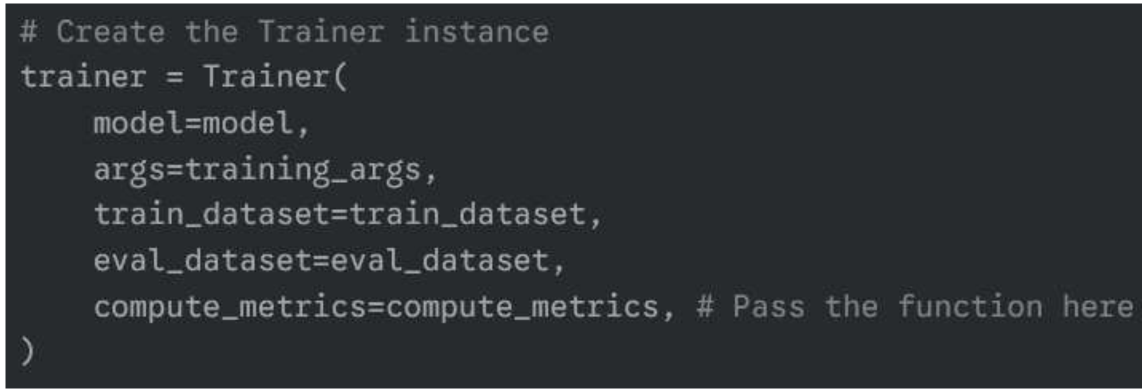
```python
trainer.train()
```

Computing Metrics with the trainer class

- cCeate and add the compute_metrics function to the Hugging Face Trainer, you'll follow these key steps: the ***model***,
  - define the function,
  - load the necessary metrics, and then
  - pass your function to the Trainer instance. :
- The function must adhere to a specific signature: it takes an EvalPrediction object as input and returns a dictionary.
- The EvalPrediction object is a named tuple with two main attributes:
  - predictions: A NumPy array or a tensor containing the model's predictions (often raw logits)
  - label_ids: A NumPy array or a tensor containing the ground-truth labels.
- The keys of the dictionary it returns are the metric names (strings), and the values are the computed metric scores (floats).



```python
import numpy as np
from evaluate import load

# Load the accuracy metric
accuracy_metric = load("accuracy")

# Define the compute_metrics function
def compute_metrics(eval_pred):
    # eval_pred is a named tuple with predictions and label_ids
    predictions, labels = eval_pred

    # For classification, we need to convert logits to predictions
    predictions = np.argmax(predictions, axis=-1)

    # Use the loaded metric's compute() method
    return accuracy_metric.compute(predictions=predictions, references=labels)
```



```python
# Create the Trainer instance
trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=eval_dataset,
    compute_metrics=compute_metrics, # Pass the function here
)
```

Fine-Tuning a pretrained model for a custom dataset

Finetuning or Feature extraction modes

- We have seen how to use the different pre-trained models for inferencing using the Pipeline class
- We have seen how the workflow of inputs to outputs looks like
- However, one main purpose we have often is:
  - to take a pretrained model and tune it with out custom dataset to build our own application.
  - To save it and reuse it
- To do this there are two ways:
  - Using the HuggingFace ***'trainer()'*** function
  - Or, the normal PyTorch way
- Let us now look at examples of both

Summary of FineTuning methods

- We learned about models and tokenizers, and now you know how to fine-tune them for your own data.
- Learned how to load and preprocess datasets, including using dynamic padding and collators.
- Implemented your own fine-tuning and evaluation of a model Implemented a lower-level training loop.

ViT – Vision Transformers

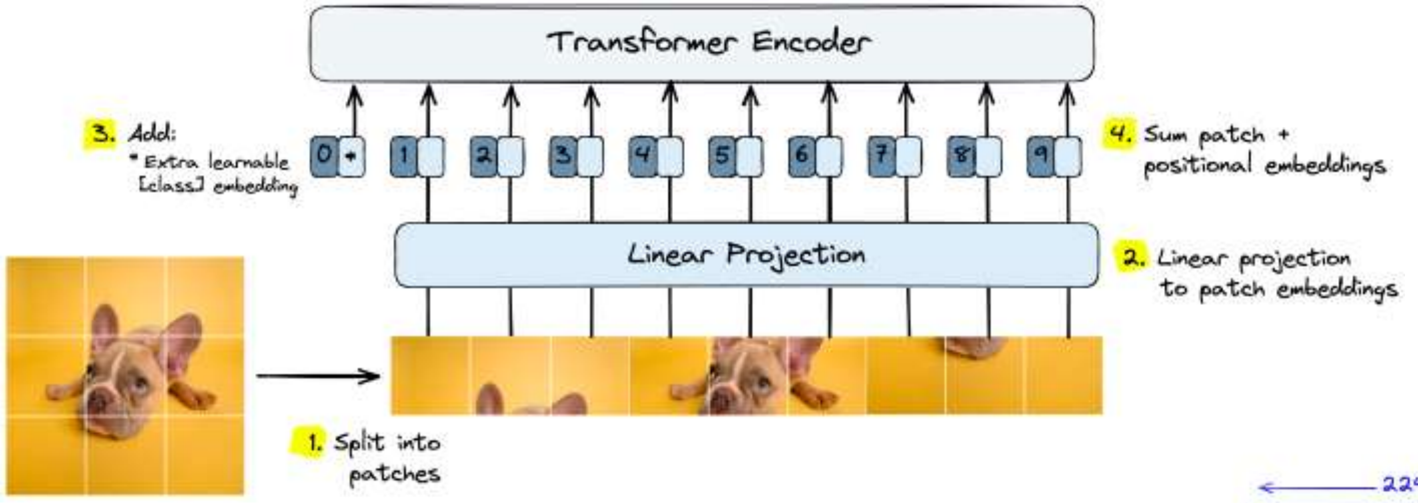
What is a Vision Transformer?

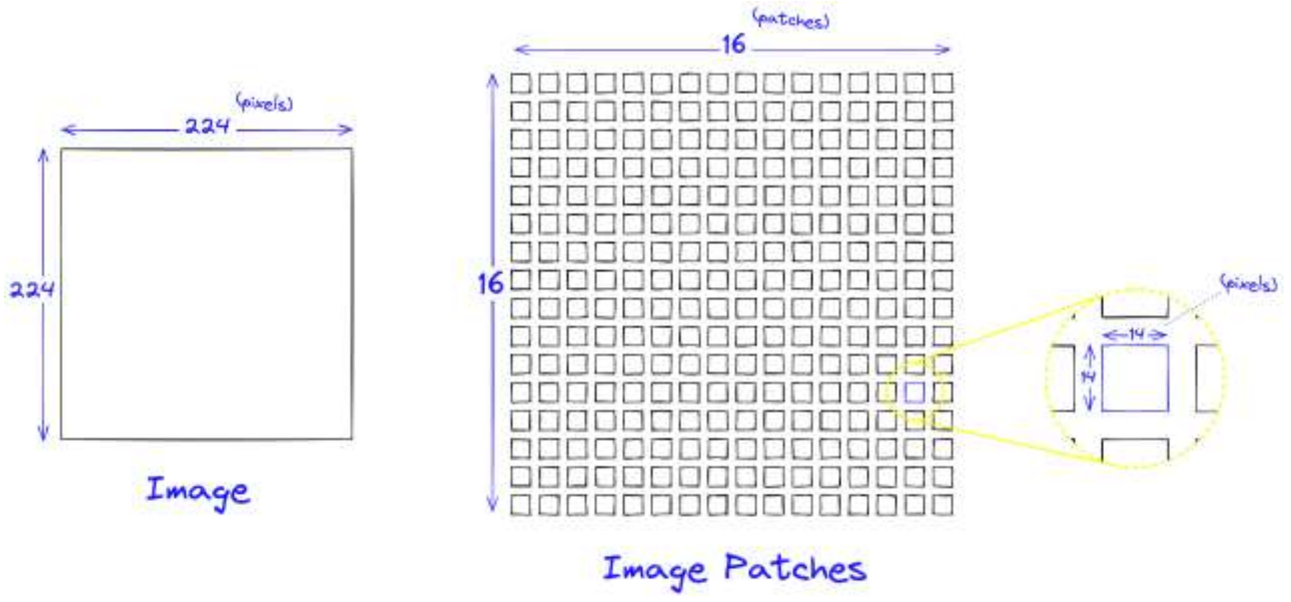
- A Vision Transformer (ViT) is **a type of neural network that can be used for image classification and other computer vision tasks**.
- Vision Transformer, is a model that applies the Transformer architecture, originally designed for natural language processing tasks, to computer vision tasks.
- The 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.
- **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.
- **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.

What is a Vision Transformer? … contd

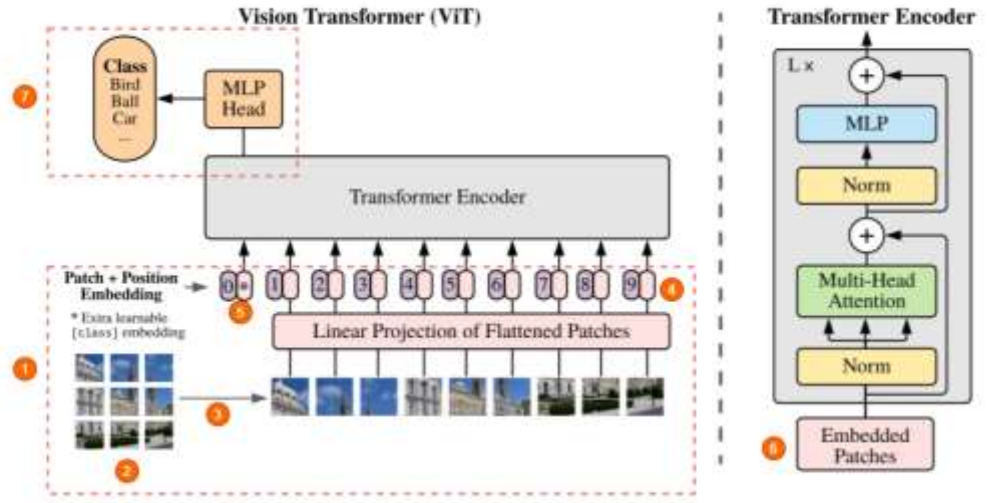
- **Positional Encoding**:
  - Transformer do not have an inherent notion of the order of input data, positional encodings (like in NLP Transformers) are added to the patch embeddings.
  - This ensures that the model knows the relative positions of patches in the image.
- **Transformer Encoder**:
  - 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.
- **Training**:
  - The model can be trained end-to-end using standard backpropagation.
  - Initially, when the ViT model was introduced, it benefited heavily from pre-training on large datasets and then fine-tuning on smaller, task-specific datasets.
- The appeal of the Vision Transformer lies in its ability to scale with data and compute.
- 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.

Image patching


Transformer Encoder
3. Add:
* Extra learnable [class] embedding
0 *
1
2
3
4
5
6
7
8
9
4. Sum patch + positional embeddings
Linear Projection
2. Linear projection to patch embeddings
1. Split into patches



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.

(2) We are breaking down the image into fixed size patches. So, one of these patches can be of dimension 16x16 or 32x32 as proposed in the paper. More patches means more simpler it is to train these networks as the patches themselves get smaller. Hence, we have that in the title - "An Image is worth 16x16 words".

(3) The patches are then unrolled (flattened) and sent for further processing into the network.

(4) Unlike NNs here the model has no idea whatsoever about the position of the samples in the sequence, here each sample is a patch from the input image. So, the image is fed **along with a positional embedding vector** and into the encoder. One thing to note here is the positional embeddings are also learnable, so you don't actually feed hard-coded vectors w.r.t to their positions.

(5) There is also a special token at the start just like BERT.

(6) Each image patch is first unrolled (flattened) into a big vector and gets multiplied with an embedding matrix which is also learnable, creating embedded patches. And these embedded patches are combined with the positional embedding vector and that gets fed into the Transformer.

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.

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.

Exercise 3

Transfer Learning on Exotic animals Dataset

1. Use the VGG16 pretrained model and train on the Exotic dataset.
2. Do two versions – Feature Extraction and Fine-Tuning.
3. For the finetuning version, the choice of how many layers and which ones you want to not freeze, is yours.
4. Build the training and testing loops by choosing all the required hyper-parameters like loss function, optimizer, et, etc.
5. Save the model – Look at PyTorch documentation / examples .
6. Do it for 2 epochs and print the results of loss and accuracy.
7. Download the image of any one of the animals from the internet and write a predict function to predict the outcome using the built model – preprocessing would be required.
8. Submit 2 separate Jupyter notebooks.
9. Exercise is due by Sep 3rd midnight.

End of Lesson