

FINETUNING TRANSFORMERS AND LLMS

✓ `import pandas as pd`

- **What it is:** `pandas` is a powerful library for data manipulation and analysis.
 - **Usage here:** You use `pandas` to read and handle Excel or CSV files, and to convert them into DataFrames for easier manipulation and later conversion to Hugging Face `Dataset`.
-

✓ `import matplotlib.pyplot as plt`

- **What it is:** A plotting library from `matplotlib` for creating static, animated, and interactive visualizations.
 - **Usage here:** Used for plotting graphs like histograms or bar charts, e.g., visualizing class distributions or token lengths.
-

✓ `import numpy as np`

- **What it is:** The core numerical computing library in Python.
 - **Usage here:** Used for numerical operations like computing predictions (`np.argmax`) and handling arrays of data.
-

✓ `import torch`

- **What it is:** The main library for building and training deep learning models using PyTorch.
 - **Usage here:** Used to move the model to GPU (`torch.device("cuda")`) and manage PyTorch-based training.
-

✓ `import time`

- **What it is:** A built-in Python module for time-related functions.
 - **Usage here:** Used to measure training or execution time with `time.time()`.
-

✓ `from sklearn.model_selection import train_test_split`

- **What it is:** A function from Scikit-learn that splits datasets into training, test, and validation sets.
- **Usage here:** Used to divide your dataset into training, validation, and test splits with stratified sampling.

```
✓ from sklearn.metrics import classification_report, accuracy_score, f1_score
```

- **What they are:**
 - `classification_report` : Outputs a detailed report with precision, recall, F1-score for each class.
 - `accuracy_score` : Computes the overall accuracy.
 - `f1_score` : Computes the F1-score (harmonic mean of precision and recall).
 - **Usage here:** Used to evaluate the model's classification performance.
-

```
✓ from transformers import ...
```

From the Hugging Face `transformers` library:

- ♦ `AutoTokenizer`
 - **What:** Automatically loads the correct tokenizer associated with a model.
 - **Usage:** Tokenizes input text so it can be fed into the transformer model.
- ♦ `AutoModelForSequenceClassification`
 - **What:** Loads a pre-trained transformer model for text classification tasks (e.g., sentiment analysis, fake news).
 - **Usage:** Fine-tuned to classify text as either "Real" or "Fake".
- ♦ `AutoConfig`
 - **What:** Loads and modifies model configuration (e.g., number of labels, label mappings).
 - **Usage:** Used to specify `label2id` and `id2label` mappings for classification.

- `TrainingArguments`

- **What:** Contains all hyperparameters and options for training.
- **Usage:** Defines training settings like batch size, learning rate, number of epochs, evaluation strategy, etc.

- `Trainer`

- **What:** High-level Hugging Face class for training/evaluating transformer models.
- **Usage:** Handles the training loop, evaluation, and metrics automatically.

- `pipeline`

- **What:** A wrapper for easy inference using trained models.
 - **Usage:** Used to run text classification (or other tasks) on new inputs with a single line.
-

- ✓ `from datasets import Dataset, DatasetDict`

- **What it is:** From Hugging Face `datasets` library.
- **Usage:**
 - `Dataset` : Represents a dataset in Hugging Face format.
 - `DatasetDict` : Organizes datasets into `train` , `test` , and `validation` splits.
- **Used to:** Convert pandas DataFrames into a format compatible with Hugging Face training tools.

- ✓ `import evaluate`

- **What it is:** Hugging Face's library for loading standard evaluation metrics.
- **Usage:** Here, it's used to compute metrics like `accuracy` (via `evaluate.load("accuracy")`).

✓ 1. Library-by-Library Explanation

🔧 Standard Python Libraries

- `warnings` : Used to suppress or display warnings.
- `requests` , `BytesIO` : Used for downloading images from the web and converting them into a file-like object.
- `PIL.Image` : Python Imaging Library for opening and displaying image files.
- `numpy` : Core package for numerical computation, used for evaluating predictions.
- `torch` : PyTorch library, used here for device handling (CPU/GPU).

📦 Hugging Face Libraries

- `datasets.load_dataset` : Loads Hugging Face datasets like `"rajistics/indian_food_images"`.
- `transformers.AutoImageProcessor` : Prepares image processors compatible with vision transformer (ViT) models.
- `transformers.AutoModelForImageClassification` : Loads a pre-trained ViT model adapted for image classification tasks.
- `transformers.TrainingArguments` : Defines hyperparameters and configuration for training.
- `transformers.Trainer` : High-level wrapper for training/evaluating models.
- `transformers.pipeline` : Creates a ready-to-use image classification pipeline.

Evaluation

- `evaluate.load('accuracy')` : Loads the evaluation metric for classification (accuracy here).

Image Transforms

- `torchvision.transforms` : Used to transform image data before feeding it into the model:
 - `RandomResizedCrop` : Randomly resizes and crops image.
 - `ToTensor` : Converts image to tensor.
 - `Normalize` : Standardizes image pixel values based on model's expected format.

Hugging Face is an AI company and an open-source platform that provides tools, models, and datasets for natural language processing (NLP) and machine learning (ML). It is widely known for hosting **pre-trained models** (like transformers) and offering libraries such as:

- **Transformers** – A library for state-of-the-art NLP models (e.g., BERT, GPT, T5).
- **Datasets** – A collection of ML datasets optimized for performance.
- **Tokenizers** – Efficient tokenization tools for NLP applications.
- **Diffusers** – A library for generative AI models (e.g., Stable Diffusion).
- **Hugging Face Hub** – A cloud-based repository for sharing models, datasets, and ML applications.

Uses of Hugging Face:

1. **Pre-trained Models** – Access and fine-tune large language models for NLP tasks.
2. **Model Deployment** – Easily deploy ML models via APIs.
3. **Data Processing** – Work with large datasets efficiently.
4. **Collaboration** – Share and manage ML projects with a community-driven approach.

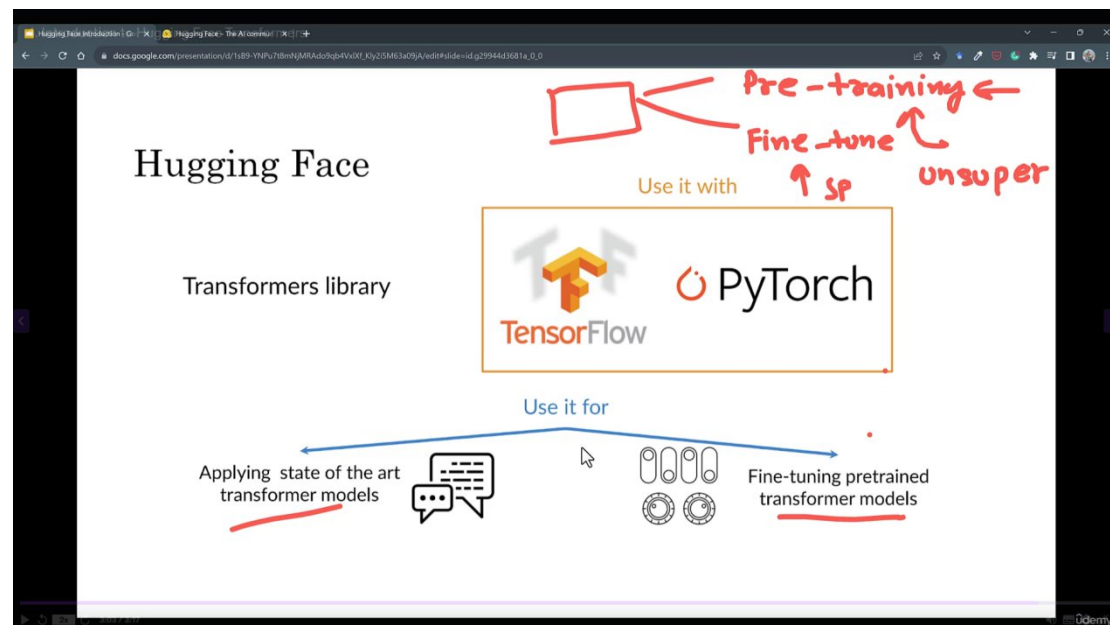
Since you're working with Small LLMs, TinyML, and embedded AI, Hugging Face can be useful for obtaining **lightweight models**, fine-tuning them for edge devices, and using quantization techniques (like ONNX or TFLite conversion) to optimize them for Raspberry Pi or MCUs.

Pretraining:

If you want to utilise the domain knowledge then, we can use this.
Un-supervised training.

Fine-tune tasks:

These are needed to get the application specific tasks out of it.
Supervised learning.



In the context of **transformers** and machine learning, **downstream tasks** refer to specific real-world applications or tasks where a pre-trained model is fine-tuned or applied. These tasks utilize the knowledge learned during **pre-training** (typically on large datasets) and adapt it for a particular objective.

Examples of Downstream Tasks in Transformers:

1. Natural Language Processing (NLP):

- **Text Classification:** Sentiment analysis, spam detection
- **Named Entity Recognition (NER):** Extracting names, locations, dates
- **Question Answering:** Finding answers from a given passage (e.g., SQuAD)
- **Text Summarization:** Generating short summaries from long texts
- **Machine Translation:** Translating text between languages
- **Text Generation:** Creating new text (e.g., chatbots, code generation)

2. Computer Vision (Vision Transformers - ViTs):

- **Image Classification:** Identifying objects in images
- **Object Detection:** Detecting and localizing objects
- **Image Captioning:** Generating textual descriptions of images

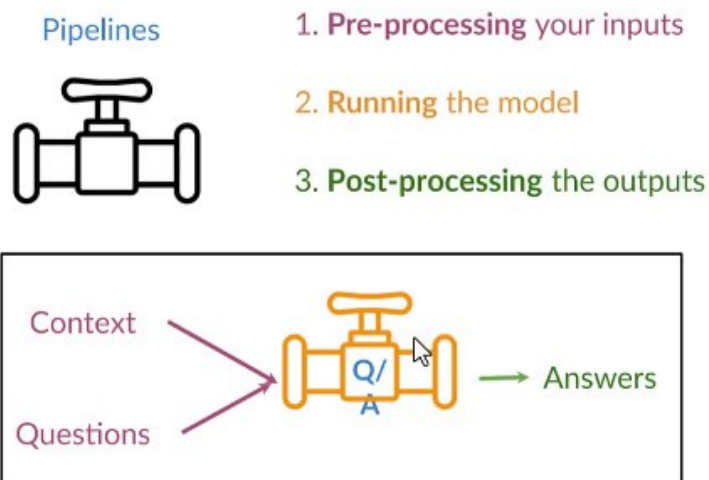
3. Audio Processing:

- **Speech Recognition:** Converting speech to text
- **Speaker Identification:** Identifying speakers from voice
- **Audio Classification:** Detecting sounds like alarms or gunshots

Since you're working with **audio transcription, TinyML, and Small LLMs**, your downstream tasks could include **speech-to-text conversion, anomaly detection (gunshots, unusual sounds), and real-time audio processing on embedded devices.**

Huggingface pipeline

Hugging Face: Using Transformers



The preprocessing can happen within the pipeline. Can also happen at outside of the pipeline.

QA

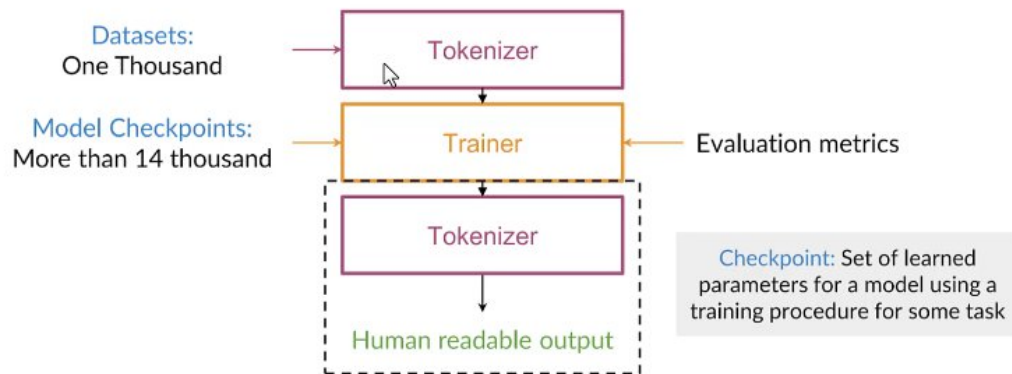
Give both the context and the question, finally you will be getting the answer.

Sentiment Analysis

Give input to the model, finally you will get to know what is the sentiment of the sentence.

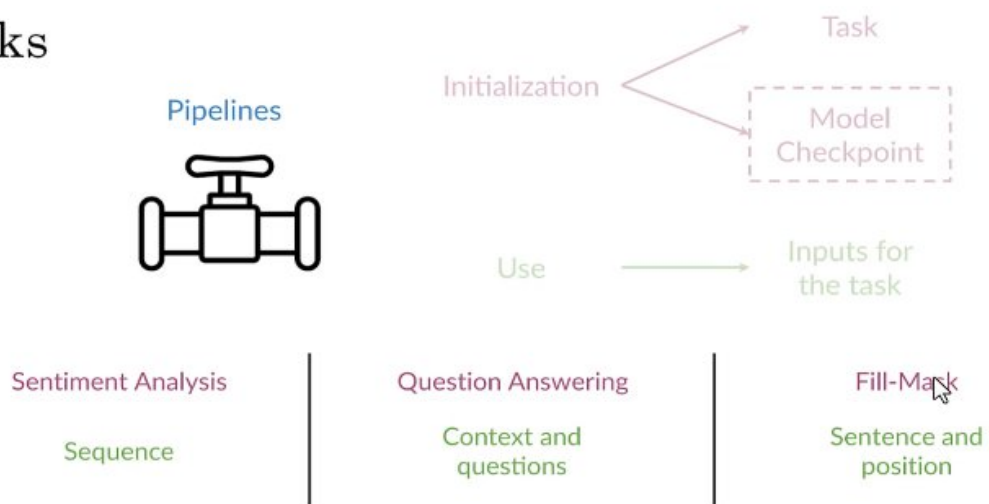
Pipelines comes to existence: Only after fine tuning of the models, inside the pipeline automatically above things will happen.

Hugging Face: Fine-Tuning Transformers



Once the QA or sentiment analysis scenarios are passed on to the transformer, that will be tokenized.
Model trainer will load the model.
Finally the human readable output is generated.

Tasks



Pipeline

What is pipeline in huggingface transformer

In **Hugging Face Transformers**, a **pipeline** is a high-level API that simplifies the process of applying pre-trained models to downstream tasks like text classification, translation, summarization, and more. It abstracts away the complexity of loading models, tokenizing inputs, and processing outputs, allowing you to perform tasks with just a few lines of code.

Key Features of pipeline :

- **Easy to Use:** No need to manually load models or preprocess data.
- **Supports Multiple Tasks:** Handles text, vision, and audio tasks.
- **Automatic Model Selection:** Downloads the appropriate model if not specified.


When you call a **pipeline** in Hugging Face Transformers, a sequence of operations occurs under the hood to process your input data using a **pre-trained transformer model**. Let's break it down step by step.

What Happens Inside a Transformer Pipeline?

1. Model Selection & Download (if not cached)

- When you call `pipeline("task_name")`, Hugging Face **automatically loads the best available model** for the task (or downloads it if it's not already cached).
- Example: If you call `pipeline("text-classification")`, it loads a model like `distilbert-base-uncased-finetuned-sst-2-english`.

2. Tokenizer Processing (Preprocessing Step)

- The input text (or image/audio) is **converted into numerical tokens** using a **tokenizer**.
- For text, words are tokenized into subwords or word pieces (like "running" → `["run", "##ning"]`).
- Tokenized text is converted into **input IDs** (numbers representing words in the model's vocabulary).

- Special tokens like `[CLS]` (classification start) or `[SEP]` (sentence separator) are added if required.

3. Model Inference (Processing Step)

- The tokenized input is fed into the transformer model.
- Inside the model:
 - **Embedding Layer**: Converts tokens into high-dimensional vector representations.
 - **Self-Attention Mechanism**: Captures relationships between words, even if they are far apart in the sentence.
 - **Feedforward Layers**: Process the transformed representations through deep neural networks.
 - **Final Layer (Task-Specific Head)**: Generates output specific to the task (e.g., classification scores, generated text, etc.).

4. Post-processing (Decoding Output)

- The model outputs **logits (raw scores)**, which are converted to human-readable results.
- Example:
 - In **text classification**, logits are converted into probabilities (using softmax) and assigned labels like `"positive"`, `"negative"`, etc.
 - In **text generation**, the model generates token IDs that are **decoded back into words**.

- In **speech-to-text**, audio waveforms are converted into text.

Example: What Happens in a Sentiment Analysis Pipeline?

```
pythonCopy Edit  
  
from transformers import pipeline  
  
classifier = pipeline("text-classification")  
  
result = classifier("I love this product!")  
print(result)
```

Under the Hood Processing:

1. **Model Selection** → Loads `distilbert-base-uncased-finetuned-sst-2-english` (a fine-tuned DistilBERT model).
2. **Tokenization** → Converts `"I love this product!"` into numerical token IDs.
3. **Model Processing** → The transformer model computes attention and predicts sentiment.
4. **Post-processing** → Converts model logits into a probability score and assigns a label:

```
jsonCopy Edit  
  
[{"label": "POSITIVE", "score": 0.9998}]
```

Key Takeaways for Your Work:

Since you're working with **TinyML, Small LLMs, and Embedded AI**, you should focus on:

- **Optimizing pipelines for lightweight models** (e.g., using quantization, pruning).
- **Using pipelines for speech-to-text (ASR)** to transcribe medical data.
- **Deploying these models on Raspberry Pi** by converting them to TensorFlow Lite (TFLite) or ONNX.

Under the Hood:

1. Model Selection & Loading:

- The pipeline **automatically selects and loads the best pre-trained model** for the task (or downloads it if not cached).
- Example: For `pipeline("text-classification")`, it loads a fine-tuned BERT or DistilBERT model.

2. Tokenization & Embeddings:

- The input **text is tokenized** (split into smaller units like words/subwords).
- Tokens are converted into **numerical IDs** that the model can understand.
- These token IDs are then mapped to **vector embeddings**, representing their meaning in a multi-dimensional space.

3. Self-Attention & Feedforward Processing:

- **Self-Attention Mechanism:**
 - This helps the model **understand the relationships between words**, even if they are far apart.
- **Feedforward Layers:**
 - The self-attention outputs are passed through multiple **dense (fully connected) layers** to further refine the understanding of the input.

4. Final Task-Specific Layer:

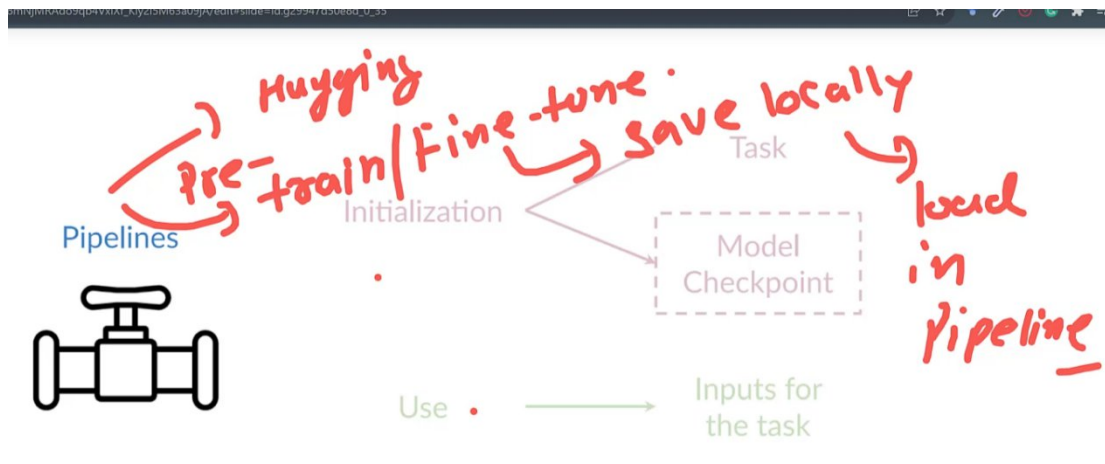
- The model applies a final layer **tailored for the specific task** (e.g., classification, text generation, etc.).
- Example: In sentiment analysis, a softmax layer produces probabilities for labels like **positive/negative**.

5. Post-Processing & Human-Readable Output:

- The raw model output (logits) is processed into a **human-readable format**.
- Example:
 - In text classification, probabilities are converted into labels (`"positive"`, `"negative"`).
 - In text generation, token IDs are **decoded back into words**.

Correction in Your Last Line:

The model **does NOT return vector embeddings** as the final output. Instead, it **returns a structured human-readable result** (like labels, text, or translated output).



Hugging face - download the retrained, or build your own and save it locally and use it for the purpose.

Huggingface-Checkpoints

In Hugging Face, **checkpoints** refer to saved states of machine learning models at different training stages. These checkpoints store model weights, configurations, and sometimes optimizers, allowing you to resume training or use the model for inference without retraining from scratch.

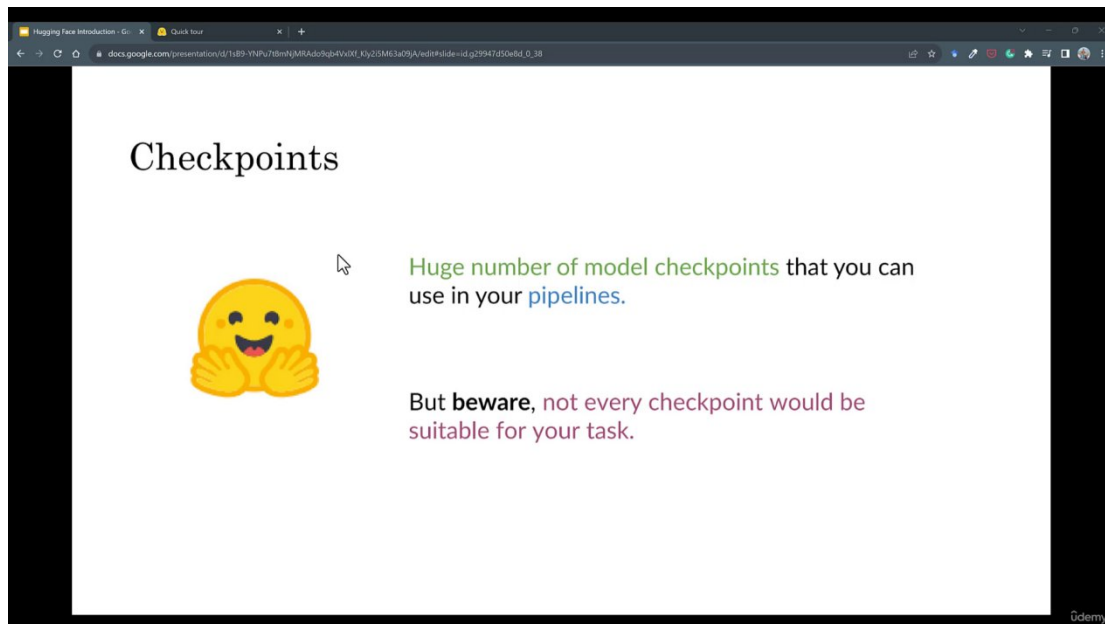
Types of Hugging Face Checkpoints:

1. **Pretrained Model Checkpoints** – Models trained on large datasets and shared on [Hugging Face Model Hub](#), e.g., `bert-base-uncased` or `facebook/opt-1.3b`.
2. **Fine-tuned Model Checkpoints** – Custom models fine-tuned on specific tasks like text classification, summarization, etc.
3. **Intermediate Training Checkpoints** – Checkpoints saved during training, enabling resumption if interrupted.

Where Are They Stored?

- Local directory (e.g., `./checkpoint-500/`)
- Hugging Face Hub (if uploaded)
- Includes files like:
 - `pytorch_model.bin` (model weights)
 - `config.json` (model configuration)
 - `tokenizer.json` (tokenizer details)





referring the checkpoint
username/checkpoint

What Are Hugging Face Auto Classes?

Hugging Face **Auto Classes** (`AutoModel`, `AutoTokenizer`, etc.) are **generic, high-level classes** that automatically load the appropriate **pretrained model, tokenizer, or configuration** for a given task.

Instead of manually specifying a model type (like `BertModel` or `T5Model`), you can use `AutoModel` to **automatically detect and load the correct model** based on a given checkpoint.

Why Use Auto Classes?

- ✓ **Simplifies Model Loading** – No need to remember specific model classes (BERT, GPT, T5, etc.).
- ✓ **Supports Any Model** – Works with any model on Hugging Face Hub.
- ✓ **Easy Model Switching** – Just change the checkpoint, and the correct model is loaded.

Specifying the particular transformer

from transformer import t5-base

from transformer import t5-small

After the introduction of the Auto

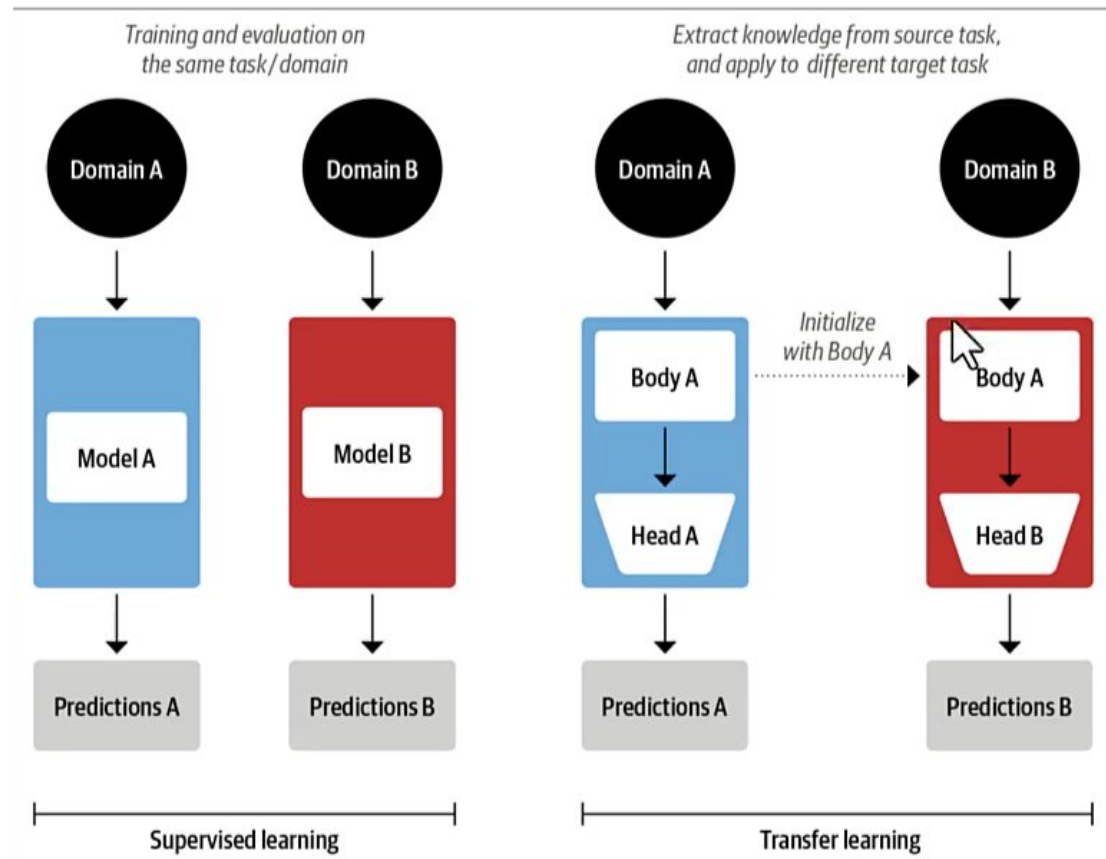
from transformer import AutoModel

```
model = AutoModel('t5-base')
```

```
model = AutoModel('t5-small')
```


Like the above many api calls are available, which will be used for benefits.

Hello Transformers



How the transfer learning will occur?

Supervised learning, we will make the model to learn based on the dataset we have got. So that will be done separately.

Transfer learning,

1. Model A:
 - a) Initialized with the random weights, made it learn from the dataset.
2. Model B:
 - a) Instead of initializing the weights randomly the weights from the model A is used and based on that training happens. Finally attaching the body of B.

Yes, you are mostly correct! Here's a clearer breakdown of what you described:

1. **Model A**

- Initialized with **random weights**
- Trained on a dataset

2. **Model B**

- Instead of random initialization, **Model A's trained weights** are used
- Training continues from these pretrained weights
- **A new output layer (body of B)** is attached (if needed) for a different task

What You're Describing:

- This is a form of **Transfer Learning** where Model A's knowledge is transferred to Model B.
- If Model A and Model B are the same architecture, B is just a **fine-tuned version of A**.
- If B has a different architecture, only part of A (like feature extractors) is reused.

Great question! Let me explain it clearly.

When **Model A and Model B have different architectures**, you **can't directly use all of Model A's weights** in Model B. Instead, you **reuse only certain parts** of Model A—usually the **feature extractor** (earlier layers)—and then modify or replace the later layers to match Model B's architecture.

What is a Feature Extractor?

In deep learning, the **early layers** of a model learn **low-level patterns** (e.g., edges, textures), while the **later layers** learn **high-level features** (e.g., object shapes, categories).

- Example: In an image model, **early layers detect edges, corners, textures**, while later layers detect **faces, objects, letters, etc.**

Example: Using a Pretrained CNN (like ResNet) for a New Task

1. **Model A (Pretrained ResNet)**

- Trained on ImageNet (classifies 1000 objects)
- We **keep only the feature extractor** (convolutional layers)

2. **Model B (Your Custom Model)**

- Instead of training from scratch, we **use ResNet's feature extractor**
- We **replace the last layers** (classifier) with a new one suited for our dataset

📌 Why do this?

- Saves time (less training required)
- Requires less data (the model has already learned useful patterns)
- Works well for tasks like **fine-tuning, object detection, segmentation, etc.**

🔧 Code Example (Using PyTorch)

python

```
import torch
import torchvision.models as models
import torch.nn as nn

# Load Pretrained Model A (ResNet)
model_A = models.resnet50(pretrained=True)

# Remove the classifier (FC layer) and keep feature extractor
feature_extractor = nn.Sequential(*list(model_A.children())[:-1])
```

Model B: Using Feature Extractor with a Custom Classifier

📄 Copy 🖋 Edit

```
class ModelB(nn.Module):
    def __init__(self, num_classes):
        super(ModelB, self).__init__()
        self.features = feature_extractor # Use ResNet's feature extractor
        self.classifier = nn.Linear(2048, num_classes) # New classifier

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x

# Instantiate Model B
model_B = ModelB(num_classes=10) # Example: Classifying into 10 categories
```

🤖 Summary

- ✅ If **Model B is different from A**, you **reuse the feature extractor from A** and add a new final layer.
- ✅ If **Model B is similar to A**, you can fine-tune the whole model instead.

Yes! In most deep learning models, we can think of **three main parts**:

1 Feature Extractor (Early & Mid Layers)

- Learns **low-level features** (edges, textures, patterns) in the early layers.
- Learns **mid-level features** (shapes, object parts) in deeper layers.
- Usually consists of **Convolutional layers (CNNs) or Transformer layers (in NLP models)**.
- Example: In ResNet, the convolutional backbone acts as a **feature extractor**.

2 Bottleneck or Intermediate Representation (Deep Layers)

- Transforms extracted features into **higher-level representations**.
- Helps in **dimensionality reduction** or **context understanding**.
- Examples:
 - CNN: **Pooling layers or Fully Connected (FC) layers**
 - Transformer: **Self-attention layers capturing relationships between tokens**
 - RNNs: **Hidden states capturing sequential patterns**
- **Sometimes this part is removed/replaced during transfer learning.**

3 Application-Specific Head (Final Layer)

- Custom layers depending on the **specific task** (classification, detection, etc.).
- Example:
 - **Image Classification** → Fully Connected Layer + Softmax
 - **Object Detection** → Bounding Box Regression + Classification
 - **NLP Sentiment Analysis** → Linear Layer + Sigmoid
 - **Speech Recognition** → Sequence-to-Sequence Decoder

Example in a CNN Model (ResNet)

Layer Type	Purpose
Feature Extractor	Convolutional Layers (detect edges, patterns)
Intermediate Representation	Pooling & Fully Connected layers (refine features)
Application-Specific Layer	Softmax for classification, Regression head for detection

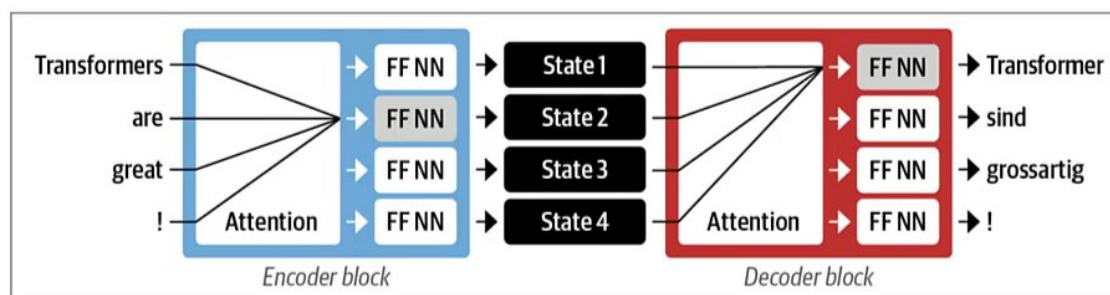
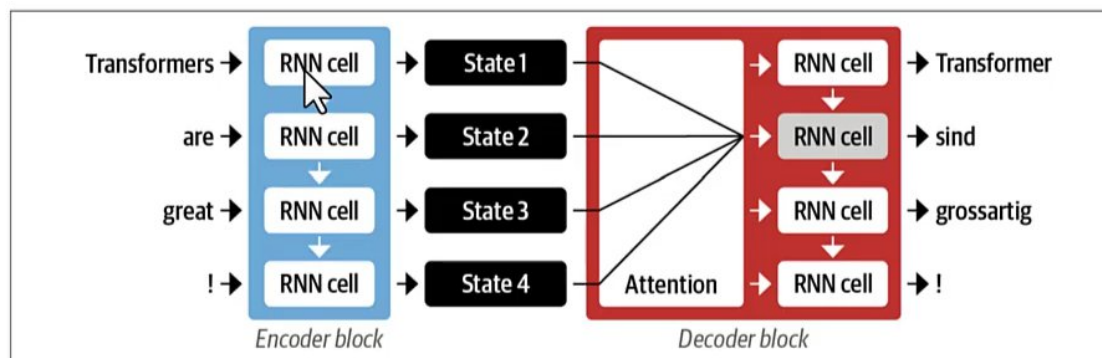
Example in NLP (Transformer-Based Model like BERT)

Layer Type	Purpose
Feature Extractor	Word Embeddings + Transformer Layers
Intermediate Representation	Self-Attention + Feed-Forward Layers
Application-Specific Layer	Classification Head (CLS Token) for sentiment analysis

Summary

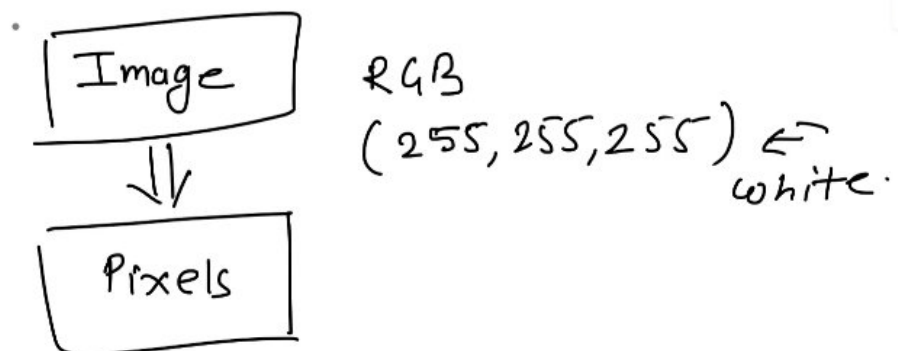
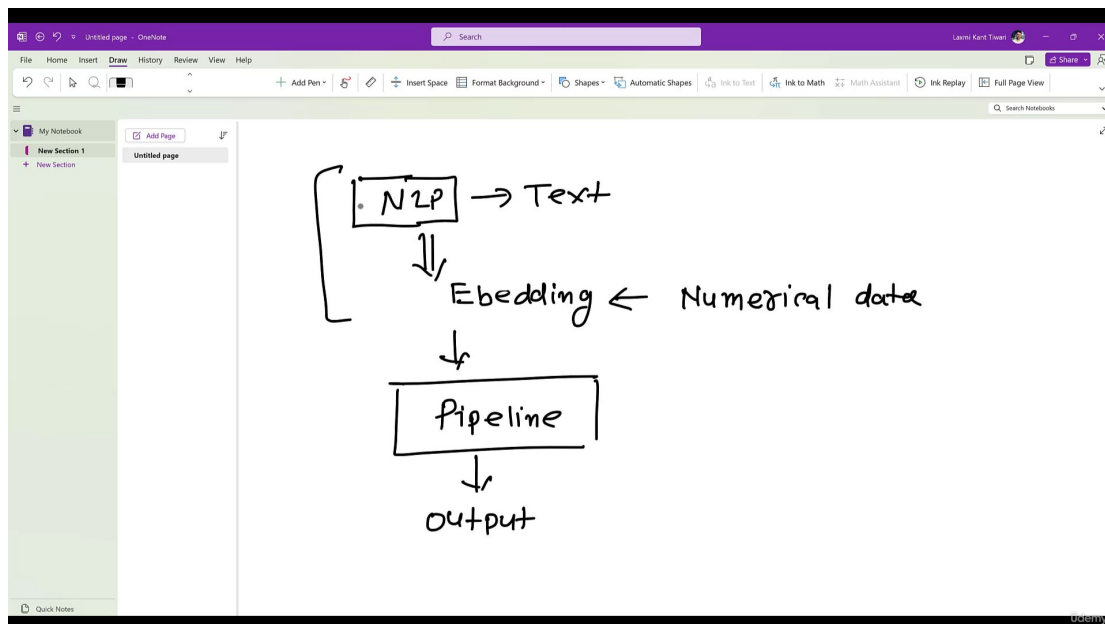
- ✓ **Feature Extractor** → Learns general patterns
- ✓ **Intermediate Representation** → Processes & refines information
- ✓ **Application-Specific Layer** → Final output layer for specific tasks

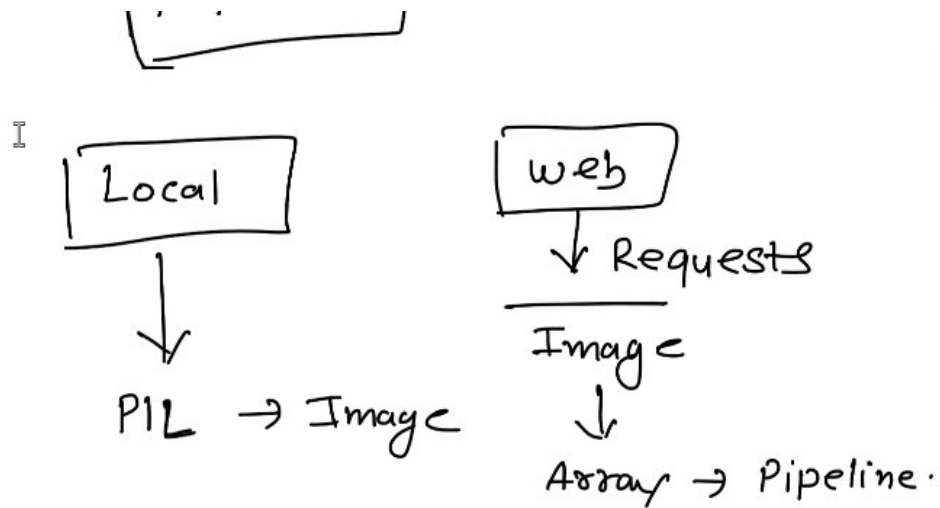
In **transfer learning**, we **keep the feature extractor**, maybe modify the **bottleneck**, and **replace the final layer** depending on the new task.



<https://huggingface.co/docs/transformers/v4.35.0/en/quicktour#pipeline>

- Text Classification
- Named Entity Recognition (NER)
- Question Answering
- Summarization
- Translation
- Text Generation
- Image Generation
- Audio Generation
- Video Generation
- Mixing of Audio and Video





There are transformer for classification, emotion, age detection, segmentation, everything the transformers do.

1. Text to speech, text to music, transformer.
2. General template.

```

from transformers import pipeline
"""
Prepre the data based on demand or application you are developing
"""
variable = pipeline("action_required", model_option)
variable(data)___this genertaes the output |
  
```

3.