TRANSFORMERS

Transformer Architecture

The **Transformer** is a deep learning model introduced for sequence-to-sequence tasks such as machine translation, but it has since become the backbone of modern NLP models like BERT and GPT. Its key innovation is the use of **self-attention** instead of recurrent or convolutional layers, making it highly parallelizable and effective for long sequences.

High-Level Structure

The Transformer consists of two main components:

1. **Encoder** – processes the input sequence and generates contextualized representations.
2. **Decoder** – generates the output sequence step by step using encoder outputs and its own previous predictions.

Encoder

The encoder is a stack of identical layers (commonly 6 in the original model). Each encoder layer has two main sub-layers:

1. **Multi-Head Self-Attention**
    - Allows each word (token) in the input to attend to every other word.
    - Captures relationships between tokens regardless of distance.
    - "Multi-head" means the model uses multiple attention mechanisms in parallel, then combines them.
2. **Feed Forward Network (FFN)**
    - A fully connected layer applied to each token independently.
    - Adds non-linearity and expands representational capacity.

Additional elements in the encoder:

- **Residual connections** around each sub-layer (helps gradients flow).
- **Layer normalization** after each sub-layer.
- **Positional encoding** added to inputs so the model knows token order.

Decoder

The decoder is also a stack of identical layers (commonly 6). Each layer has three main sub-layers:

1. **Masked Multi-Head Self-Attention**
    - Similar to encoder's self-attention, but masking prevents attending to future tokens (to preserve autoregressive generation).
    - Ensures predictions for position *t* depend only on tokens before *t*.
2. **Encoder-Decoder Attention**
    - Attends over the encoder's output representations.
    - Helps the decoder focus on relevant parts of the input sequence when generating output.
3. **Feed Forward Network (FFN)**
    - Same as in the encoder.

Additional elements in the decoder:

- Residual connections, layer normalization, and positional encoding are also used here.

Input and Output Flow

1. Input tokens are embedded and positional encodings are added.
2. The encoder processes these through stacked layers, producing context-aware embeddings.
3. The decoder takes target sequence tokens (shifted right, with masking) plus encoder outputs.
4. After passing through stacked decoder layers, the final outputs go through a linear layer and softmax to predict the next token.

Key Advantages of Transformers

- **Parallelization** – Unlike RNNs, all tokens are processed simultaneously.
- **Long-range dependencies** – Self-attention directly connects any two tokens.
- **Scalability** – Works extremely well when scaled up with large data and parameters.

This structure—**encoder, decoder, and attention mechanisms**—is what makes the Transformer the foundation of today's large language models.

Natural Language Processing Sequence

The process begins with the raw input sequence, such as a sentence.

The input text is split into smaller units called tokens.

Each token is converted into a dense vector representation that captures its semantic meaning.

Positional information is added to the embeddings to help the model understand the order of tokens.

INPUT SEQUENCE → TOKENIZATION → EMBEDDING LAYER → POSITIONAL ENCODING → ENCODER LAYERS → CONTEXTUALIZED REPRESENTATION → DECODER LAYERS

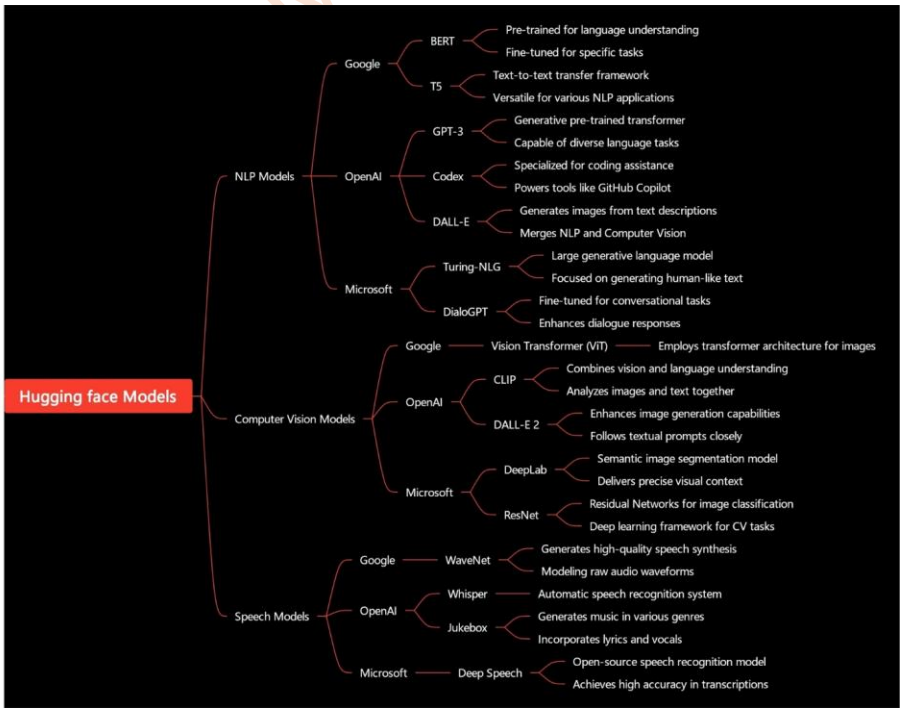
The encoded input passes through multiple identical encoder layers, each consisting of a Self-Attention Mechanism and a Feed-Forward Neural Network.

The final output from the encoder layers is a contextualized representation of the input sequence.

The contextualized representation is fed into multiple identical decoder layers, which generate the output sequence iteratively.

HUGGING FACE

Hugging Face serves as a one-stop repository for state-of-the-art models across domains like NLP, computer vision, and speech processing.



HUGGING FACE TRANSFOMERS

1. Introduction to Hugging Face

1.1 Company Overview

Hugging Face was founded in 2016 by Clément Delangue, Julien Chaumond, and Thomas Wolf. Initially, it focused on building an AI-powered chatbot, but quickly pivoted to open-source AI tools, recognizing the broader value in the underlying NLP models. Today, Hugging Face is a central hub for machine learning, hosting a vast repository of models, datasets, and tools for NLP, computer vision, audio, and multimodal AI. The company is valued at over $4.5 billion as of 2023 and is often referred to as the "GitHub of machine learning".

1.2 Mission and Role in the AI/ML Ecosystem

Hugging Face's mission is to democratize good machine learning and maximize its positive impact across industries and society. It champions open-source principles, transparency, and community-driven innovation, making state-of-the-art AI tools accessible to everyone.

Key Roles:

- **Open-Source Leadership**: Hosts over a million repositories for models, datasets, and demos.
- **Transformers Library**: Industry standard for implementing and experimenting with cutting-edge models.
- **Expanding Modalities**: Supports NLP, vision, audio, and multimodal AI.
- **Enterprise Tools**: Offers Inference Endpoints, AutoTrain, Spaces, and Private Hubs.
- **Education & Ethics**: Provides extensive documentation, tutorials, and prioritizes ethical AI development.

1.3 Importance of Transformers and Hugging Face

Transformers have revolutionized NLP and are now foundational in computer vision, audio, and generative AI. Hugging Face's Transformers library and Model Hub have made these models accessible, reusable, and easy to fine-tune, accelerating research and real-world applications.

2. The Hugging Face Website – Detailed Walkthrough

2.1 Models Hub

- **What it is**: The Models Hub is a repository of over 1.7 million open-source machine learning models for NLP, vision, audio, and multimodal tasks.

- **Organization**: Models are organized by task (e.g., text generation, sentiment analysis), framework (PyTorch, TensorFlow, JAX), language, and other tags.
- **Model Cards**: Each model has a card with documentation, usage, intended applications, limitations, and performance metrics.
- **Search Filters**: Filter by task, language, framework, license, and more.
- **Download & Usage**: Models can be loaded programmatically:

```
from transformers import AutoModel
model = AutoModel.from_pretrained('bert-base-uncased')
```

- **Popular Categories**: Text classification, text generation, translation, summarization, image classification, speech recognition.

2.2 Datasets Hub

- **Available Datasets**: Over 400,000 datasets in more than 8,000 languages.
- **Dataset Cards**: Each dataset includes documentation, usage instructions, and a Dataset Viewer for in- browser exploration.
- **Formats & Licensing**: Supports CSV, JSON, text, images, audio, and more. Licensing is clearly indicated for each dataset.
- **Usage Example**:

```
from datasets import load_dataset
dataset = load_dataset("imdb")
```

- **Privacy Controls**: Datasets can be public or private, with organization- level access.

2.3 Spaces

- **What they are**: Spaces are interactive demo applications that showcase ML models in the browser.
- **How to Create/Host/Deploy**:
- Choose Gradio or Streamlit as the SDK.
- Add `app.py` and `requirements.txt`.
- Push to the Space; deployment is automatic.
- **Showcase**: Spaces are used for text- to- image generators, audio transcription apps, chatbots, and more.

2.4 Docs & Tutorials

- **Structure**: Documentation is organized by topic—Hub, Models, Datasets, Spaces, API/library docs, security, and best practices.
- **Transformers Library Guides**: Cover installation, usage, fine- tuning, and advanced features.
- **Key Tutorials**: Sentiment analysis, text generation, translation, summarization, image classification, and more.

2.5 Community

- **Forums & Discussions**: Engage with other users, ask questions, and share knowledge.
- **Contributions**: Submit models, datasets, and code improvements.
- **Leaderboards**: Track top- performing models on various benchmarks.

2.6 Other Tools & APIs

- **Inference API**: Deploy models for real- time inference.
- **Tokenizers**: Fast, customizable tokenization library.
- **Evaluate**: Library for model evaluation metrics.
- **Accelerate**: Simplifies distributed and mixed- precision training.

3. Hugging Face Transformers Library

3.1 Installation and Setup

Install via pip:

```
pip install transformers
```

For datasets and evaluation:

```
pip install datasets evaluate accelerate
```

Test installation:

```
from transformers import pipeline
print(pipeline('sentiment-analysis')(‘hugging face is the best’))
```


3.2 Core Concepts

Tokenizers

- **Purpose**: Convert raw text into tokens and numerical IDs for model input.
- **Types**: BPE, WordPiece, Unigram, SentencePiece.
- **Special Tokens**: `[CLS]`, `[SEP]`, `[PAD]`, etc.
- **AutoTokenizer**: Automatically loads the correct tokenizer for any model.

Models

- **Definition**: Neural networks trained for specific tasks (classification, generation, etc.).
- **AutoModel**: Loads the correct model architecture for a given checkpoint.

Pipelines

- **High- level abstraction**: For common tasks like sentiment analysis, text generation, etc.

- **Example**:

```
from transformers import pipeline
classifier = pipeline('sentiment-analysis')
print(classifier("Hugging Face is awesome!"))
```

3.3 AutoTokenizer & AutoModel

What Are Auto Classes?

Auto classes (AutoTokenizer, AutoModel, etc.) automatically select and load the correct implementation for a given model checkpoint, abstracting away the details of which architecture or tokenizer is needed.

Example: Loading Tokenizer and Model

```
from transformers import AutoTokenizer, AutoModel

checkpoint = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModel.from_pretrained(checkpoint)
```

Tokenizing and Running Inference

```
text = "Hugging Face makes AI easy."
inputs = tokenizer(text, return_tensors="pt")
outputs = model(**inputs)
print(outputs.last_hidden_state.shape)
```


3.4 Architecture Overview

Encoder- Only

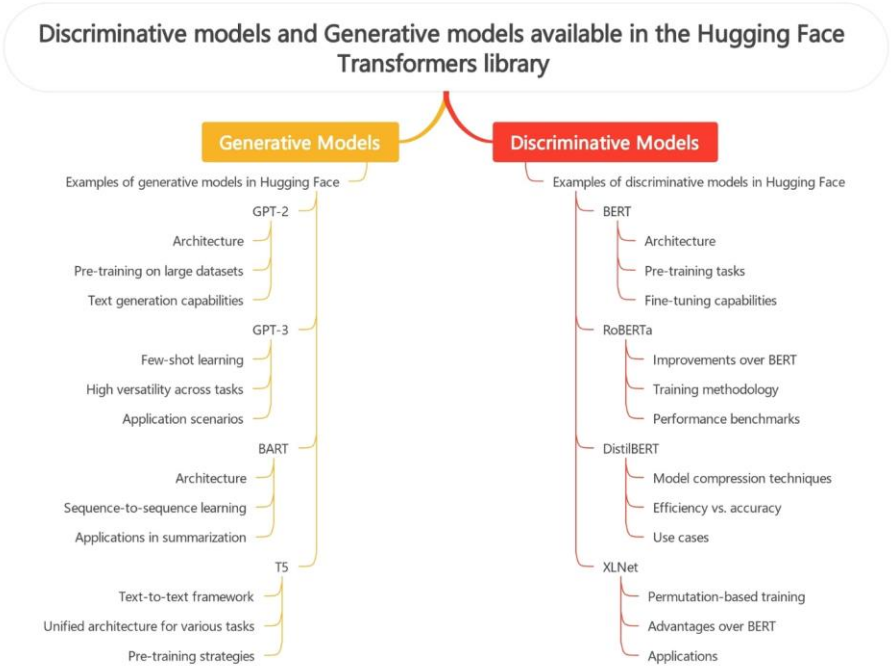
- **Examples**: BERT, RoBERTa, DistilBERT.
- **Use**: Text classification, NER, extractive QA.
- **Mechanism**: Bidirectional self- attention.

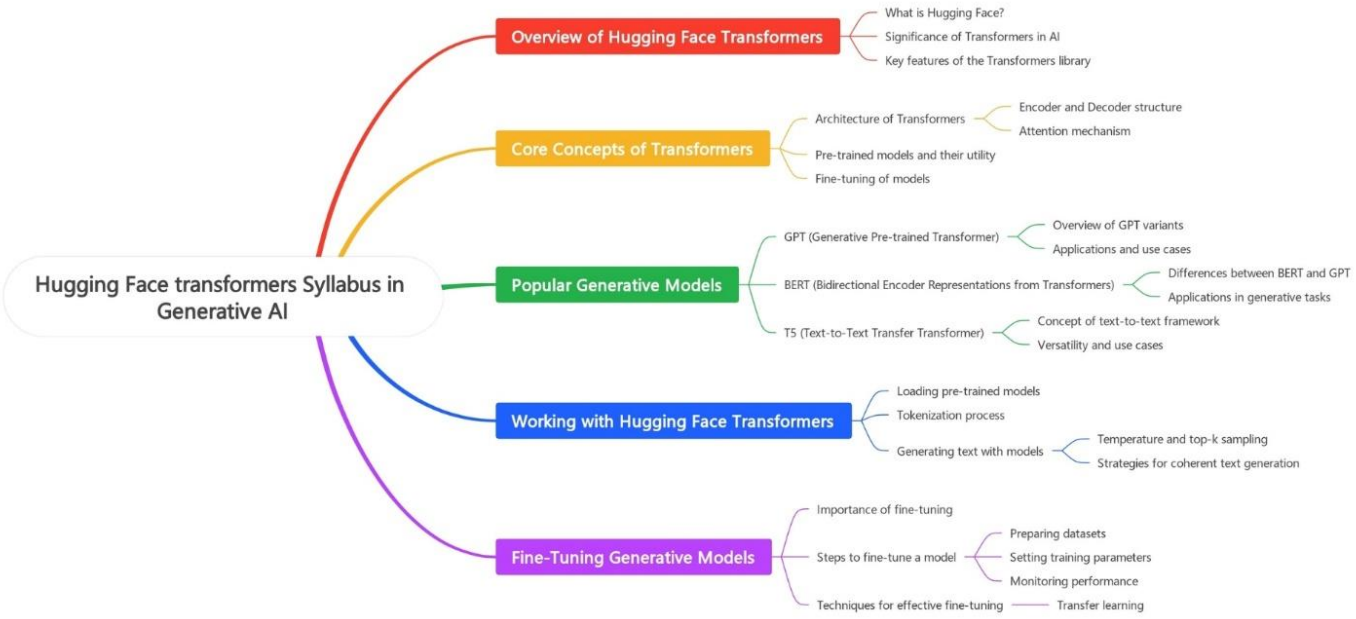
Decoder- Only

- **Examples**: GPT- 2, GPT- 3, LLaMA.
- **Use**: Text generation, chatbots, code generation.
- **Mechanism**: Unidirectional (causal) self- attention.

Encoder- Decoder

- **Examples**: T5, BART, MarianMT.
- **Use**: Translation, summarization, generative QA.
- **Mechanism**: Encoder processes input, decoder generates output with cross- attention.


Discriminative models and Generative models available in the Hugging Face Transformers library
Generative Models
Examples of generative models in Hugging Face
GPT-2
Architecture
Pre-training on large datasets
Text generation capabilities
GPT-3
Few-shot learning
High versatility across tasks
Application scenarios
BART
Architecture
Sequence-to-sequence learning
Applications in summarization
T5
Text-to-text framework
Unified architecture for various tasks
Pre-training strategies
Discriminative Models
Examples of discriminative models in Hugging Face
BERT
Architecture
Pre-training tasks
Fine-tuning capabilities
RoBERTa
Improvements over BERT
Training methodology
Performance benchmarks
DistilBERT
Model compression techniques
Efficiency vs. accuracy
Use cases
XLNet
Permutation-based training
Advantages over BERT
Applications


Hugging Face transformers Syllabus in Generative AI
Overview of Hugging Face Transformers
What is Hugging Face?
Significance of Transformers in AI
Key features of the Transformers library
Core Concepts of Transformers
Architecture of Transformers
Encoder and Decoder structure
Attention mechanism
Pre-trained models and their utility
Fine-tuning of models
Popular Generative Models
GPT (Generative Pre-trained Transformer)
Overview of GPT variants
Applications and use cases
BERT (Bidirectional Encoder Representations from Transformers)
Differences between BERT and GPT
Applications in generative tasks
T5 (Text-to-Text Transfer Transformer)
Concept of text-to-text framework
Versatility and use cases
Working with Hugging Face Transformers
Loading pre-trained models
Tokenization process
Generating text with models
Temperature and top-k sampling
Strategies for coherent text generation
Fine-Tuning Generative Models
Importance of fine-tuning
Steps to fine-tune a model
Preparing datasets
Setting training parameters
Monitoring performance
Techniques for effective fine-tuning
Transfer learning

3.5 Pre-trained vs Fine- Tuned Models

What is a Pre- trained Model?

A model trained on large, general- purpose datasets to learn broad patterns (e.g., BERT trained on Wikipedia). Not specialized for any specific task.

What is a Fine- tuned Model?

A pre- trained model further trained on a smaller, labeled, task- specific dataset (e.g., BERT fine- tuned for sentiment analysis). This adapts the model for a particular application.

Why Fine- tuning is Necessary

- **Task Adaptation**: Pre- trained models are generalists; fine- tuning makes them specialists.
- **Domain- Specific Improvements**: Fine- tuning on medical, legal, or other domain data improves relevance and accuracy.

How to Load, Fine- tune, and Use Pre- trained Models

Loading a Pre- trained Model:

```
from transformers import AutoModelForSequenceClassification
model = AutoModelForSequenceClassification.from_pretrained("bert-base-
uncased")
```

Fine- tuning Example:

```
from transformers import Trainer, TrainingArguments

training_args = TrainingArguments(
    output_dir="./results",
    num_train_epochs=3,
    per_device_train_batch_size=16,
    evaluation_strategy="epoch",
)

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=val_dataset,
)
trainer.train()
```

3.6 Sequential Code Examples

Using a Pipeline for Sentiment Analysis

```
from transformers import pipeline
sentiment_analyzer = pipeline("sentiment-analysis")
result = sentiment_analyzer("I love using Hugging Face Transformers!")
print(result)
```

Using AutoTokenizer and AutoModel

```
from transformers import AutoTokenizer, AutoModel
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
model = AutoModel.from_pretrained("bert-base-uncased")

inputs = tokenizer("Transformers are amazing!", return_tensors="pt")
outputs = model(**inputs)
print(outputs.last_hidden_state)
```

Fine- tuning a Pre- trained Model

See the fine- tuning example above for sentiment analysis.

4. Popular & Best Models on Hugging Face

4.1 NLP Models

- **BERT**: Encoder- only; text classification, NER, QA.
- **RoBERTa**: Improved BERT; classification, fill- mask, NER.
- **T5**: Encoder- decoder; summarization, translation, QA.
- **BART**: Encoder- decoder; summarization, text generation.
- **DistilBERT**: Lightweight BERT; mobile/edge deployments.
- **Bloom, LLaMA**: Large- scale generative models; text generation, chat.

4.2 Vision Models

- **ViT (Vision Transformer)**: Image classification, feature extraction.
- **CLIP**: Multimodal; zero- shot image classification, retrieval.
- **DETR**: Object detection, segmentation.
- **Stable Diffusion**: Text- to- image generation, inpainting.

4.3 Audio Models

- **Wav2Vec2**: Speech recognition.
- **Whisper**: Multilingual speech transcription.

4.4 Special- Purpose Models

- **Falcon, Mistral, Phi, Gemma**: Large language models for generative tasks, instruction following, and research.

4.5 Use Cases and Model Selection

- **Text Classification**: BERT, RoBERTa, DistilBERT.
- **Text Generation**: GPT- 2, LLaMA, Bloom.
- **Summarization/Translation**: T5, BART, MarianMT.
- **Image Classification**: ViT, CLIP.
- **Text- to- Image**: Stable Diffusion.
- **Speech Recognition**: Wav2Vec2, Whisper.

5. Datasets and Data Handling

5.1 Finding and Loading Datasets

- **From the Hub**:

```
from datasets import load_dataset
dataset = load_dataset("imdb")
```

- **From Local Files**:

```
python dataset = load_dataset('csv', data_files='my_file.csv')
```


5.2 Data Processing with Datasets Library

- **Mapping Functions**:

```
python dataset = dataset.map(lambda x: {"length": len(x["text"])})
```

- **Filtering**:

```
python filtered = dataset.filter(lambda x: x['label'] == 1)
```

- **Tokenization**:

```
python from transformers import AutoTokenizer tokenizer =
AutoTokenizer.from_pretrained('bert-base-cased') tokenized_dataset =
dataset.map(lambda x: tokenizer(x['text']), batched=True)
```

- **Integration with ML Frameworks**:

```
python dataset.set_format(type="torch", columns=["input_ids", "labels"])
```

6. Spaces and Deployment

6.1 Creating and Hosting Apps with Gradio/Streamlit

- **Gradio Example**:

```
```python import gradio as gr

def greet(name): return f"Hello, {name}!"

demo = gr.Interface(fn=greet, inputs="text", outputs="text") demo.launch() ```
```

Save as `app.py`, add gradio to `requirements.txt`, and push to your Space.

- **Streamlit Example**:

```
```python import streamlit as st

st.title("Text Transformer") user_input = st.text_area("Enter text:", "Type here...") if st.button('Transform'): st.write(user_input.upper()) ```
```

Save as `app.py`, add streamlit to `requirements.txt`, and push to your Space.

6.2 Integrating Models into Production

- **Spaces**: Host interactive demos and share via public URLs.
- **Inference API**: Deploy models for real- time inference in production systems.

7. Practical Examples

7.1 Text Classification

```
from transformers import pipeline
classifier = pipeline("text-classification")
print(classifier("Hugging Face is transforming AI!"))
```

7.2 Text Generation

```
from transformers import pipeline
generator = pipeline("text-generation", model="gpt2")
print(generator("Once upon a time,", max_length=30))
```

7.3 Translation

```
from transformers import pipeline
translator = pipeline("translation_en_to_fr", model="t5-base")
print(translator("Hugging Face is amazing!"))
```

7.4 Summarization

```
from transformers import pipeline
summarizer = pipeline("summarization", model="facebook/bart-large-cnn")
print(summarizer("Long article text here..."))
```

7.5 Image Classification

```
from transformers import pipeline
image_classifier = pipeline("image-classification", model="google/vit-base-patch16-224")
print(image_classifier("path/to/image.jpg"))
```

7.6 Text- to- Image Generation

```
from diffusers import StableDiffusionPipeline
pipe = StableDiffusionPipeline.from_pretrained("CompVis/stable-diffusion-v1-4")
image = pipe("A futuristic cityscape at sunset").images[0]
image.show()
```


7.7 Audio Transcription

```
from transformers import pipeline
transcriber = pipeline("automatic-speech-recognition", model="openai/whisper-base")
print(transcriber("path/to/audio.wav"))
```

8. Contribution and Collaboration

8.1 Uploading Models and Datasets

- **Upload via Web UI or CLI**: Use the Hugging Face website or `huggingface_hub` Python package to upload models and datasets.
- **Model Cards and Dataset Cards**: Provide documentation, intended use, and limitations.

8.2 Contributing to Open- Source Libraries

- **Find Issues**: Look for “Good First Issue” or “Good Second Issue” labels.
- **Fork, Clone, and Branch**: Standard GitHub workflow.
- **Code, Test, and Submit PRs**: Follow contribution guidelines, run tests, and respond to feedback.
- **Community Support**: Engage in forums, answer questions, and help others.

9. Best Practices and Tips

9.1 Model Evaluation and Benchmarking

- **Iterative Evaluation**: Continuously assess models using both offline (curated datasets) and online (real- world) evaluation.
- **Metrics**: Use accuracy, completeness, conciseness, relevance, consistency, and safety metrics.
- **Human- in- the- Loop**: Combine automated and human evaluation for nuanced tasks.

9.2 Ethical Considerations

- **Non- Maleficence**: Avoid harm to individuals or groups.
- **Privacy**: Protect sensitive data.
- **Fairness**: Avoid bias and discrimination.
- **Transparency**: Document processes and decisions.
- **Regulatory Alignment**: Ensure compliance with legal and ethical standards.

Enhanced Content and Recent Updates

The original material above captures the core functions and offerings of Hugging Face. Below is additional context, updated information, and recent developments that enhance each section.

Updated Company Insights

- **Series- D Funding (August 2023)**: A Series- D round led by Salesforce, Google, Nvidia and others valued Hugging Face at **US $4.5 billion**. Funds were earmarked to expand hiring and invest in technology.
- **Continued Growth**: Hugging Face employs over 170 people and continues to grow as its platform becomes the **“GitHub of machine learning.”**

Expanded Mission & Tools

- **Ethical AI Advocacy**: Hugging Face promotes responsible AI through model documentation standards, reproducibility guidelines, and safety best practices.
- **HF Jobs (July 2025)**: Developers can run compute jobs (scripts or large- scale training) on Hugging Face infrastructure via CLI. Jobs are billed by the second and support CPU and GPU tasks.

- **Rewritten CLI**: The `huggingface-cli` was replaced with `hf <resource> <action>` for a more intuitive command structure.

New Model Releases & Innovation

- **Llama 4 Series (April 2025)**: Released by Meta, Llama 4 uses a mixture- of- experts architecture and includes:
- **Scout**: 109 billion total parameters with 17 billion active parameters and a 10 million- token context window.
- **Maverick**: 400 billion total parameters with 17 billion active parameters and a 1 million- token context window[5]. Both models are multimodal (text and images), multilingual, and trained on public, licensed and proprietary data[6].
- **GPT- OSS (August 2025)**: OpenAI's first open- weight family. The **gpt- oss- 120b** model (~117 B parameters) and **gpt- oss- 20b** model (~21 B parameters) use mixture- of- experts architecture and **4- bit MXFP4 quantization**, enabling the 120B model to run on a single H100 GPU and the 20B model on a 16 GB consumer GPU. They are released under the Apache 2.0 license[8].
- **Smolagents (Dec 2024)**: A minimalist library enabling language models to call external tools by writing actions in code. It supports agentic workflows requiring multi- step reasoning.

Parameter- Efficient Tuning & TRL

- **PEFT Library**: Hugging Face introduced the **PEFT** library in February 2023. PEFT fine- tunes only a small subset of parameters while freezing the rest, drastically reducing compute and storage requirements[11]. It helps avoid catastrophic forgetting and yields portable checkpoints of only a few megabytes.
- **TRL**: The **Transformer Reinforcement Learning** library provides tools for reinforcement- learning based fine- tuning and alignment. Recent tutorials show how to use **QLoRA** and **Spectrum** to fine- tune large models efficiently.

Improved Platform Features

- **Dataset Viewer JSON Support**: Added July 2025; the dataset viewer now displays nested JSON cells directly, making it easier to explore complex datasets.
- **Inference Providers**: Hugging Face's Inference Providers service now supports OpenAI- compatible APIs and allows specifying the provider name when calling models. This makes it easier to switch providers or deploy GPT- OSS models.

Expanded Best Practices & Ethics

- **Usage Policies**: Models like GPT- OSS are released with minimal usage policies that mandate lawful, ethical use and require avoiding harmful applications.
- **Evaluation & Feedback**: Combining automated metrics with human feedback helps identify hallucinations, bias and harmful outputs.

Additional Learning Resources

- **PEFT Blog**: Explains the motivation, methods (LoRA, Prefix Tuning, Prompt Tuning) and benefits of parameter- efficient fine- tuning.
- **Heavybit Guide**: Discusses full fine- tuning vs. PEFT. Full fine- tuning retrains *all* parameters and offers maximum control but is resource- intensive and carries risks like overfitting and catastrophic forgetting. PEFT tunes only a subset of parameters, reducing compute requirements and memory usage, though fully fine- tuned models may achieve slightly better performance.

Best Models Across Domains — Parameter and Task Summary

The table below summarises widely used models for text, vision, audio and video tasks. It lists the model type, parameter count and typical use cases. Parameter counts come from publicly available sources such as research papers, model documentation or blog posts.

Domain/Task	Model	Parameter Count	Typical Use Cases (examples)
NLP (Text)	**BERT (base)**	110 M parameters	Sentiment analysis, text classification, named entity recognition
	BERT (large)	340 M parameters	Higher- accuracy text classification and question answering
	T5- Base	~220 M parameters	Summarization, translation, question answering
	T5- Large	~770 M parameters	High- quality generation and summarization
	T5- 3B / T5- 11B	3 B / 11 B parameters	Large- scale generative tasks, multilingual translation
	Llama 4 (Scout)	109 B total; 17 B active parameters	Instruction following, code generation, multimodal chat
	GPT- OSS- 120B	~117 B parameters	Open- weight reasoning, tool calling, chatbots
Vision (Images)	**Vision Transformer (ViT- B/16)**	~86 M parameters	Image classification, feature extraction
	CLIP (ViT-B/16 variant)	150 M total (86.2 M vision, 63.1 M text)	Zero- shot image classification, cross- modal retrieval
	Stable Diffusion	~1.1 B parameters	Text- to- image generation,

Domain/Task	Model	Parameter Count	Typical Use Cases (examples)
		(approx., depending on version)	inpainting
Audio	**Wav2Vec2 (base)**	95 M parameters	English speech recognition
	Wav2Vec2 (large)	317 M parameters	High- accuracy speech recognition
	XLS- R (0.3B / 1B / 2B)	300 M, 1 B, 2 B parameters	Multilingual speech recognition across 128 languages
	Whisper (tiny / base / small)	39 M, 74 M, 244 M parameters	Low- resource or fast speech recognition; supports transcription and translation
	Whisper (medium / large)	769 M, 1.55 B parameters	Multilingual transcription with higher accuracy; translation
Video	**VideoMAE (large)**	304 M parameters	Video classification and action recognition

Full Fine-Tuning vs Parameter-Efficient Fine-Tuning (PEFT)

Full Fine- Tuning

Full fine- tuning means updating **all** parameters of a pre- trained model during training. It offers maximum control over model behaviour and can yield the best performance because the entire model adapts to the new task. However, this approach has notable drawbacks:

- **Resource Intensive**: Fine- tuning a model with hundreds of millions or billions of parameters demands **large compute and memory** resources. For example, fully fine- tuning a 405B- parameter model would require a powerful GPU cluster.
- **Large Checkpoints**: The fine- tuned model remains as large as the base model. Storing and deploying multiple task- specific checkpoints becomes expensive.
- **Risk of Overfitting and Catastrophic Forgetting**: When training on small domain datasets, the model can memorize training data and perform poorly on new data. Overfitting and catastrophic forgetting (loss of general knowledge) are common concerns.

Parameter- Efficient Fine- Tuning (PEFT)

PEFT methods fine- tune **only a small subset of parameters** while keeping the majority of the pre- trained weights **frozen**. These methods, such as **LoRA**, **Prefix**

Tuning, **Prompt Tuning** and **P-Tuning**, drastically reduce the number of trainable parameters. Advantages include:

- **Lower Compute and Memory Requirements**: PEFT demands far less memory and compute than full fine-tuning because only a few additional matrices are trained[11]. This allows developers to fine-tune large models on consumer GPUs and even in free Colab notebooks.
- **Smaller Checkpoints**: The trained parameters (adapters or low-rank matrices) are tiny—often just a few megabytes. They can be layered on top of the base model, enabling the same pre-trained model to serve multiple tasks simply by swapping adapter weights.
- **Mitigation of Catastrophic Forgetting**: Because the base model weights are frozen, the model retains its general knowledge while adapting to new tasks. PEFT methods often generalize better in low-data regimes.
- **Reduced Training Time**: With fewer trainable parameters, training runs faster, enabling quick iteration and experimentation.

However, PEFT comes with **trade-offs**:

- **Slightly Lower Peak Performance**: Fully fine-tuned models usually achieve marginally higher accuracy or lower loss than PEFT models because all weights are optimized.
- **Limited Task Control**: Since most parameters remain fixed, there are fewer degrees of freedom to dramatically alter model behaviour. For highly specialized or complex tasks, full fine-tuning may still be preferable.

Choosing Between Full Fine-Tuning and PEFT

Consider the following factors when deciding which method to use:

1. **Dataset Size and Quality**: Full fine-tuning requires large, high-quality datasets to avoid overfitting. If only a small dataset is available, PEFT is often safer and more efficient.
2. **Compute Resources**: Full fine-tuning demands powerful GPUs and significant storage, while PEFT can run on consumer hardware. Teams with limited hardware should favor PEFT.
3. **Performance Requirements**: If the highest possible performance is critical and resources permit, full fine-tuning may yield better results. Otherwise, PEFT offers a strong balance of performance and efficiency.
4. **Deployment and Portability**: PEFT's small checkpoints simplify deployment across devices and tasks.