Transformers

The essentials of Hugging Face Transformers, from one-call pipelines to tokenization, text generation, and fine-tuning with Trainer.

Install & Load from the Hub

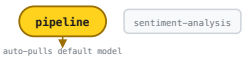
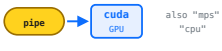
Install transformers, then pull any of 1M+ models by name.

PURPOSE	COMMAND	RESULT
Install transformers + torch.	`pip install "transformers[torch]"`	
Check the installed version.	`import transformers;transformers.__version__`	
Log in to the Hub (private/push).	`hf auth login`	
Load a model by Hub id.	`AutoModel.from_pretrained("bert-base-uncased")`	
Load weights in their stored dtype.	`AutoModel.from_pretrained(name,dtype="auto")`	
Spread a big model across devices.	`AutoModelForCausalLM.from_pretrained(name, device_map="auto")`	

from_pretrained("org/model") downloads and caches once; dtype="auto" loads stored precision, device_map="auto" shards big models

pipeline(): a task in 3 lines

One call: name a task, get a ready-to-run model.

PURPOSE	COMMAND	RESULT
Build a task pipeline.	`pipe = pipeline(task="sentiment-analysis")`	
Run it on text.	`pipe("I love this!")`	
Pick a specific model.	`pipeline("text-generation", model="Qwen/Qwen3-0.6B")`	
Choose the device.	`pipeline(task, model=name, device="cuda")`	
Batch a list of inputs.	`pipe(["text a", "text b", "text c"], batch_size=8)`	
Generation kwargs pass through.	`pipe(prompt, max_new_tokens=50, do_sample=True)`	

pipeline(task=...) loads a default model; set model= and device= to choose, pass a list with batch_size=, and extra kwargs flow to the model

Tokenize

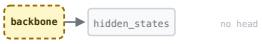
The tokenizer turns text into integer ids the model reads.

PURPOSE	COMMAND	RESULT
Load the matching tokenizer.	`tok = AutoTokenizer.from_pretrained(name)`	
Encode to tensors.	`enc =tok("Hello world",return_tensors="pt")`	
See the integer ids.	`enc["input_ids"]`	
Read the attention mask.	`enc["attention_mask"]`	
Decode ids back to text.	`tok.decode(enc["input_ids"][0])`	
Apply a chat template.	`tok.apply_chat_template(messages, add_generation_prompt=True, return_tensors="pt")`	

load AutoTokenizer from the same checkpoint as the model; tok(text, return_tensors="pt") gives input_ids + attention_mask, tok.decode goes back

AutoModel & AutoTokenizer

Auto-classes infer the right architecture from the name.

PURPOSE	COMMAND	RESULT
Raw backbone (hidden states only).	`AutoModel.from_pretrained(name)`	
Causal LM head (text generation).	`AutoModelForCausalLM.from_pretrained(name)`	
Sequence classification head.	`AutoModelForSequenceClassification .from_pretrained(name,num_labels=2)`	
Token classification (NER).	`AutoModelForTokenClassification .from_pretrained(name,num_labels=9)`	
Pair tokenizer to model.	`tok = AutoTokenizer.from_pretrained(name)`	
Save both locally.	`model.save_pretrained("out/")` `tok.save_pretrained("out/")`	

AutoModel is the bare backbone; AutoModelFor* snaps on a task head; always pair AutoTokenizer on the same checkpoint

Batch, Pad & Truncate

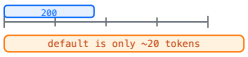
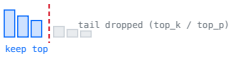
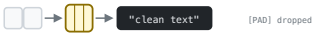
Encode many texts at once into one rectangular tensor.

PURPOSE	COMMAND	RESULT
Encode a list of texts.	`tok(["short","a longer one"], padding=True)`	
Pad to a fixed length.	`tok(texts, padding="max_length", max_length=128)`	
Truncate long inputs.	`tok(texts, truncation=True, max_length=512)`	
Mask hides the padding.	`enc["attention_mask"]`	
Pad on the left for generation.	`AutoTokenizer.from_pretrained(name, padding_side="left")`	
Dynamic padding per batch.	`DataCollatorWithPadding(tokenizer=tok)`	

padding=True pads to the longest row; the attention_mask makes padding safe; pad left for decoder LLMs, collate dynamically to save compute

Generate Text

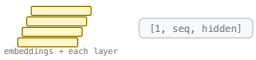
model.generate() decodes new tokens; dials control style.

PURPOSE	COMMAND	RESULT
Greedy: always pick the top token.	`model.generate(**enc, max_new_tokens=50)`	
Always set the output length.	`model.generate(**enc, max_new_tokens=200)`	
Sample for creative text.	`model.generate(**enc, do_sample=True, temperature=0.8)`	
Trim the candidate pool.	`model.generate(**enc, do_sample=True, top_k=50, top_p=0.95)`	
Curb repetition.	`model.generate(**enc, repetition_penalty=1.2)`	
Decode the new tokens to text.	`tok.batch_decode(out, skip_special_tokens=True)`	

generate() is greedy by default; always set max_new_tokens, then do_sample + temperature/top_k/top_p shape the output

Forward Pass & Logits

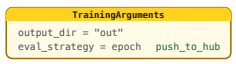
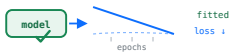
Call the model directly to read raw logits and hidden states.

PURPOSE	COMMAND	RESULT
Run a forward pass (no grad).	`with torch.no_grad():` `out = model(**enc)`	
Read the logits.	`out.logits`	
Turn logits into probabilities.	`out.logits.softmax(dim=-1)`	
Take the predicted class.	`out.logits.argmax(dim=-1)`	
Ask for hidden states.	`model(**enc,` `output_hidden_states=True).hidden_states`	
Pooled sentence embedding.	`out.last_hidden_state[:, 0]`	

model(**enc) returns .logits (pre-softmax); softmax for probabilities, argmax for the class, output_hidden_states=True for per-layer states

Fine-tune with Trainer & push

Trainer runs the loop; push_to_hub() shares the result.

PURPOSE	COMMAND	RESULT
Configure the run.	`args = TrainingArguments(output_dir="out",` `eval_strategy="epoch", push_to_hub=True)`	
Assemble the trainer.	`trainer = Trainer(model, args,` `train_dataset, eval_dataset,processing_class=tok)`	
Score during training.	`Trainer(..., compute_metrics=compute_metrics)`	
Train the model.	`trainer.train()`	
Evaluate on the held-out set.	`trainer.evaluate()`	
Push weights + tokenizer to Hub.	`trainer.push_to_hub()`	

Trainer wraps the training loop: configure with TrainingArguments, pass the tokenizer as processing_class=, then train(), evaluate(), push_to_hub()