Assignment 8 — Solutions

Hugging Face Transformers

Part B — Transformers & LLMs · 14 questions with answers

Q1. What are the advantages and limitations of the Hugging Face transformers library?

Answer

Advantages:

- Access to **hundreds of thousands of pre-trained models** on the Hub with one line (`from_pretrained`).
- Unified API (`AutoModel`, `AutoTokenizer`) across architectures. (transformers v5+ is **PyTorch-first**; TensorFlow/JAX support was dropped – use transformers 4.x if you need TensorFlow.)
- `pipeline()` for quick inference; `Trainer` for fine-tuning; integrates with datasets, evaluate, PEFT, TRL, accelerate, bitsandbytes.
- Large community, documentation, model cards; easy sharing (`push_to_hub`).

Limitations:

- Large models need big GPUs; default inference is not the fastest for production serving (use vLLM/TGI/ONNX).
- High-level abstractions can hide details and make custom architectures harder.
- Frequent version changes / breaking updates; heavy dependencies.
- Model quality and licences vary – you must check each model card.

Q2. What tasks are supported by transformers pipelines?

Answer

NLP: text-classification (alias sentiment-analysis), token-classification (alias ner), text-generation, fill-mask, zero-shot-classification, feature-extraction (embeddings), table-question-answering.

Vision: image-classification, object-detection, image-segmentation, depth-estimation, zero-shot-image-classification, zero-shot-object-detection, mask-generation, video-classification.

Audio: automatic-speech-recognition, audio-classification, text-to-audio (alias text-to-speech), zero-shot-audio-classification.

Multimodal: image-text-to-text, document-question-answering, any-to-any.

Version note: transformers **4.x** also had summarization, translation, text2text-generation and question-answering pipelines. **v5 removed them** – use text-generation with an LLM, or load the model with AutoModelForSeq2SeqLM / AutoModelForQuestionAnswering directly. Check `transformers.pipelines.SUPPORTED_TASKS` for your installed version.

Q3. Summarize a paragraph using pipeline with the 'google-t5/t5-base' model. [CODE]

Answer

transformers 4.x – summarization pipeline:

```
from transformers import pipeline

summarizer = pipeline("summarization", model="google-t5/t5-base")
text = ("Transformers have become the dominant architecture in natural language "
        "processing. Unlike recurrent networks, they process all tokens in parallel "
        "using self-attention, which allows them to capture long-range dependencies "
        "and scale to billions of parameters. This has enabled large language models "
        "such as GPT and Llama that power modern chatbots and coding assistants.")
result = summarizer(text, max_length=50, min_length=15, do_sample=False)
print(result[0]["summary_text"])
```

(T5 is a text-to-text model; the pipeline adds the "summarize: " prefix automatically.)

transformers v5+ (summarization pipeline removed) – same result with the model directly:

```
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM

tok = AutoTokenizer.from_pretrained("google-t5/t5-base")
model = AutoModelForSeq2SeqLM.from_pretrained("google-t5/t5-base")
inputs = tok("summarize: " + text, return_tensors="pt", truncation=True, max_length=512)
ids = model.generate(**inputs, max_new_tokens=50, min_length=15, num_beams=4)
print(tok.decode(ids[0], skip_special_tokens=True))
```

Q4. Do sentiment analysis using a transformers pipeline. [CODE]

Answer

```
from transformers import pipeline

sentiment = pipeline("sentiment-analysis")   # default: distilbert-base-uncased-finetuned-sst-2-
    english
print(sentiment(["I love this phone, the battery lasts forever!",
                 "Worst customer service I have ever experienced."]))
# [{'label': 'POSITIVE', 'score': 0.9998}, {'label': 'NEGATIVE', 'score': 0.9997}]

# Specific / multilingual model
clf = pipeline("sentiment-analysis",
               model="cardiffnlp/twitter-roberta-base-sentiment-latest")
print(clf("The movie was okay, nothing special"))
```

Q5. Classify these headlines into ["entertainment", "sports", "politics", "science"] using a zero-shot classification pipeline:

- **Oscar-winning actor to play the lead role in an upcoming crime-drama film**
- **Australia beat India in the final of the Cricket World Cup by six wickets**
- **Election commission announces seven-phase voting for the general election**
- **NASA astronaut returns to Earth after a 6-month stay on the ISS [CODE]**

Answer

```
from transformers import pipeline

classifier = pipeline("zero-shot-classification", model="facebook/bart-large-mnli")
labels = ["entertainment", "sports", "politics", "science"]
headlines = [
    "Oscar-winning actor to play the lead role in an upcoming crime-drama film",
    "Australia beat India in the final of the Cricket World Cup by six wickets",
    "Election commission announces seven-phase voting for the general election",
    "NASA astronaut returns to Earth after a 6-month stay on the ISS",
]
for h in headlines:
    r = classifier(h, candidate_labels=labels)
    print(f"{r['labels'][0]:13} ({r['scores'][0]:.2f})  {h}")
# entertainment, sports, politics, science
```

How it works: the NLI model treats each label as a hypothesis ("This text is about sports.") and scores entailment – no task-specific training needed. An LLM with a prompt is another option.

Q6. Initialize the tokenizer for distilbert-base-uncased and encode a sentence: [CODE]

```
tokenizer = AutoTokenizer.from_pretrained(model_name)
encoded = tokenizer("this is a outstanding, product love it.")
print(encoded)
# {'input_ids': [101, 2023, 2003, 1037, 5151, ...,102], 'attention_mask': [1, 1, 1, ..., 1]}
```

(a) What are input_ids and attention_mask in this output? (b) How do you return PyTorch tensors instead?

Answer

(a)

- **input_ids** – the token IDs of the sentence from the tokenizer's vocabulary, including special tokens: 101 = [CLS] at the start, 102 = [SEP] at the end. These are what the model's embedding layer looks up.
- **attention_mask** – 1 for real tokens and 0 for padding tokens, telling the model which tokens to attend to. Here all are 1 because there is no padding.

(b) Pass `return_tensors="pt"` (PyTorch); "tf" for TensorFlow, "np" for NumPy:

```
from transformers import AutoTokenizer
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
encoded = tokenizer("this is a outstanding, product love it.",
                    padding=True, truncation=True, return_tensors="pt")
print(encoded["input_ids"].shape)   # torch.Size([1, 11])
```

Q7. Convert the sentence "This is an outstanding product. I love it." into input IDs using the BERT tokenizer. [CODE]

Answer

```
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("bert-base-uncased")

sentence = "This is an outstanding product. I love it."
print(tok.tokenize(sentence))
# ['this', 'is', 'an', 'outstanding', 'product', '.', 'i', 'love', 'it', '.']
ids = tok.encode(sentence)          # adds [CLS] and [SEP]
print(ids)
# [101, 2023, 2003, 2019, 5151, 4031, 1012, 1045, 2293, 2009, 1012, 102]
print(tok.convert_tokens_to_ids(tok.tokenize(sentence)))   # without special tokens
```

Q8. Convert the input IDs [0, 100, 101, 102, 103] back into tokens using the BERT tokenizer. What do you notice? [CODE]

Answer

```
from transformers import AutoTokenizer
tok = AutoTokenizer.from_pretrained("bert-base-uncased")
print(tok.convert_ids_to_tokens([0, 100, 101, 102, 103]))
# ['[PAD]', '[UNK]', '[CLS]', '[SEP]', '[MASK]']
print(tok.decode([0, 100, 101, 102, 103]))
```

Observation: these IDs are all **special tokens** – in BERT's vocabulary, 0 = [PAD], 100 = [UNK], 101 = [CLS], 102 = [SEP], 103 = [MASK]. (Note: in RoBERTa/GPT tokenizers the same IDs map to completely different tokens – IDs are only meaningful for a specific tokenizer.)

Q9. How do you download any model from the Hugging Face Hub in PyTorch and in TensorFlow (AutoModel vs TFAutoModel)?

Answer

```
model_name = "bert-base-uncased"

# PyTorch
from transformers import AutoTokenizer, AutoModel
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)

# TensorFlow (only transformers 4.x - TF support was removed in v5)
from transformers import TFAutoModel
tf_model = TFAutoModel.from_pretrained(model_name)

# Task-specific heads
from transformers import AutoModelForSequenceClassification
clf = AutoModelForSequenceClassification.from_pretrained(model_name, num_labels=2)

# Save locally / download all files
model.save_pretrained("./my_bert"); tokenizer.save_pretrained("./my_bert")
from huggingface_hub import snapshot_download
snapshot_download(repo_id=model_name, local_dir="./bert_files")
```

Models are cached in ~/.cache/huggingface. For gated models (e.g. Llama), log in with `huggingface-cli login` first. (transformers v5 removed TensorFlow classes such as TFAutoModel; install transformers<5 for TensorFlow, otherwise use PyTorch.)

Q10. Look at this code and output:

```
model = AutoModel.from_pretrained(model_name)
output = model(**inputs)
output.last_hidden_state.shape
# torch.Size([3, 14, 768])
```

What does each number (3, 14, 768) represent?

Answer

`last_hidden_state` has shape **(batch_size, sequence_length, hidden_size)**:

- **3** – batch size: 3 sentences were passed in.
- **14** – sequence length: each sentence has 14 tokens after tokenization **and padding** (including [CLS] and [SEP]).
- **768** – hidden size: each token is represented by a 768-dimensional contextual embedding (BERT-base / DistilBERT).

So it contains one 768-d vector for every token of every sentence. For a sentence embedding, take the [CLS] vector (`[:, 0, :]`) or mean-pool over tokens using the attention mask.

Q11. Explain the Trainer and TrainingArguments classes. ★

Answer

- **TrainingArguments** – a configuration object with all training hyperparameters and settings: output_dir, learning_rate, num_train_epochs, per_device_train_batch_size, weight_decay, eval_strategy, save_strategy, logging_steps, fp16/bf16, gradient_accumulation_steps, load_best_model_at_end, report_to, etc.
- **Trainer** – a high-level class that runs the full training loop for you: batching, forward/backward pass, optimizer (AdamW) and LR scheduler, mixed precision, multi-GPU/distributed training, evaluation, checkpointing, logging and early stopping (callbacks).

```
from transformers import Trainer, TrainingArguments
args = TrainingArguments(output_dir="out", learning_rate=2e-5, num_train_epochs=3,
                         per_device_train_batch_size=16, eval_strategy="epoch",
                         save_strategy="epoch", load_best_model_at_end=True)
trainer = Trainer(model=model, args=args, train_dataset=train_ds,
                  eval_dataset=val_ds, compute_metrics=compute_metrics)
trainer.train(); trainer.evaluate()
```

Q12. What are autoregressive models? ★

Answer

Autoregressive models generate a sequence **one token at a time, where each new token is predicted from all previously generated tokens**: P(x) = Π P(x_t | x_1 … x_(t−1)). The generated token is appended to the input and the process repeats until an end token or max length.

Examples: GPT family, Llama, Mistral (decoder-only LLMs); the decoder of T5/BART is also autoregressive. They use causal masking during training. Contrast: BERT (masked/non-autoregressive) cannot generate text naturally.

Q13. In output = model.generate(**inputs, num_beams=5, no_repeat_ngram_size=2, max_new_tokens=50) explain **inputs, num_beams, no_repeat_ngram_size and max_new_tokens.

Answer

- ****inputs** – unpacks the tokenizer output dictionary (input_ids, attention_mask) as keyword arguments to `generate()`.
- **num_beams=5** – **beam search** with 5 beams: keeps the 5 most probable partial sequences at each step and returns the best complete one (better quality than greedy, slower).
- **no_repeat_ngram_size=2** – forbids any 2-gram (pair of consecutive tokens) from appearing twice in the output → reduces repetition.
- **max_new_tokens=50** – generates at most 50 new tokens (not counting the prompt).

Q14. Compute BLEU, METEOR, ROUGE and perplexity using the Hugging Face evaluate library. [CODE]

Answer

```
import evaluate

preds = ["the cat sat on the mat"]
refs  = ["the cat is sitting on the mat"]

bleu = evaluate.load("bleu")
print(bleu.compute(predictions=preds, references=[[r] for r in refs]))

rouge = evaluate.load("rouge")
print(rouge.compute(predictions=preds, references=refs))   # rouge1, rouge2, rougeL

meteor = evaluate.load("meteor")
print(meteor.compute(predictions=preds, references=refs))

perplexity = evaluate.load("perplexity", module_type="metric")
print(perplexity.compute(predictions=["The weather is nice today."], model_id="gpt2"))
# lower perplexity = the model finds the text more natural
```

BLEU → translation (n-gram precision); ROUGE → summarization (n-gram recall); METEOR → translation with synonyms/stemming; Perplexity → language-model fluency.