Hugging Face Transformers Inference API

COMP4901Y

Binhang Yuan

Transformers Are Eating Deep Learning
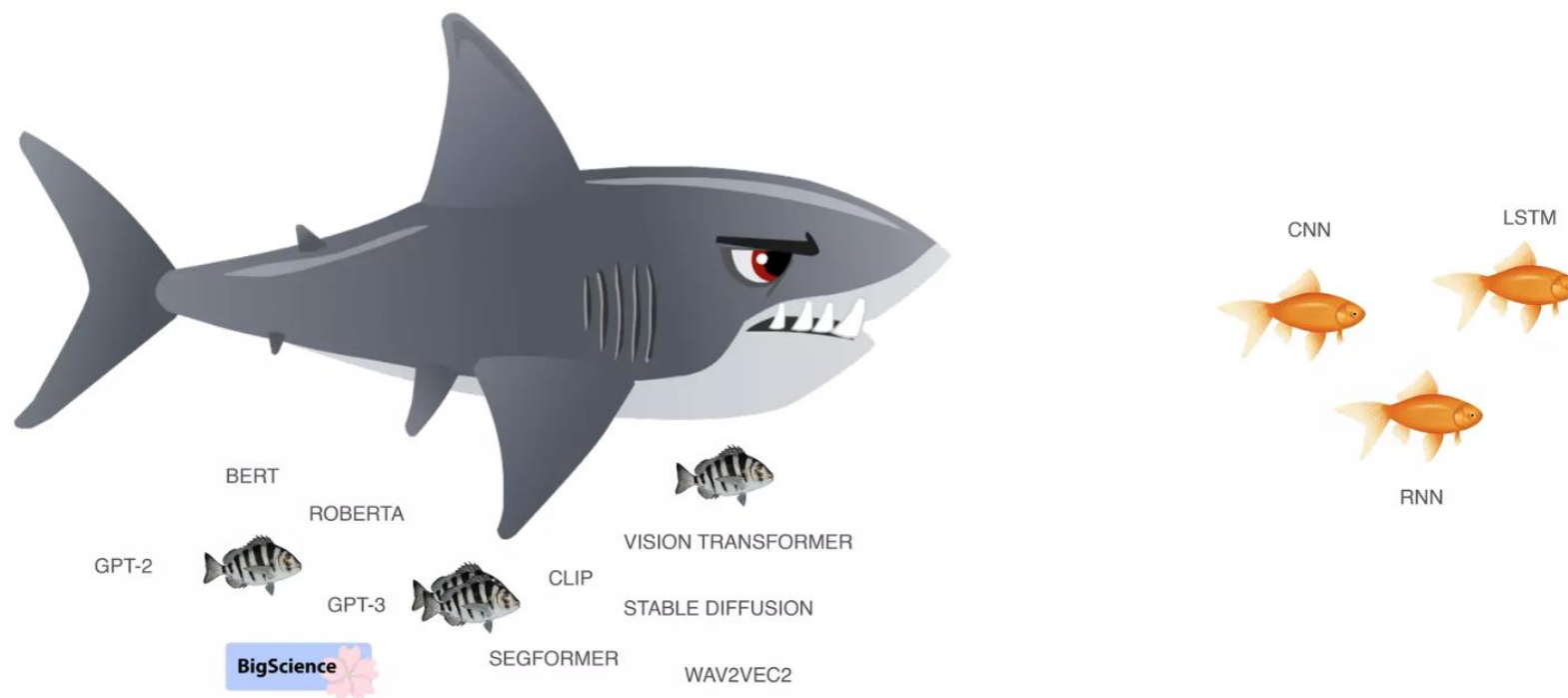
🤗 Hugging Face
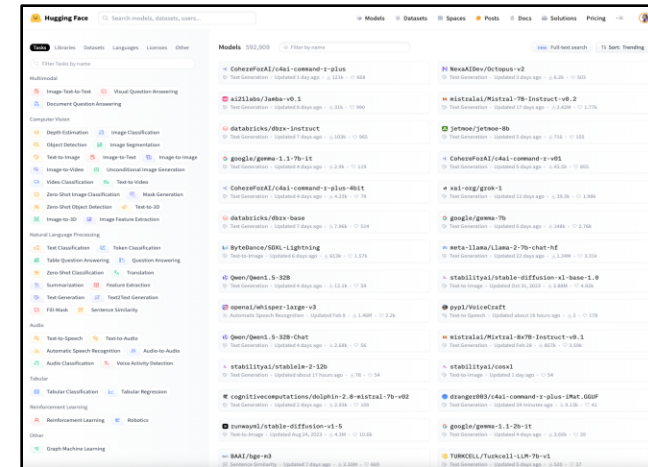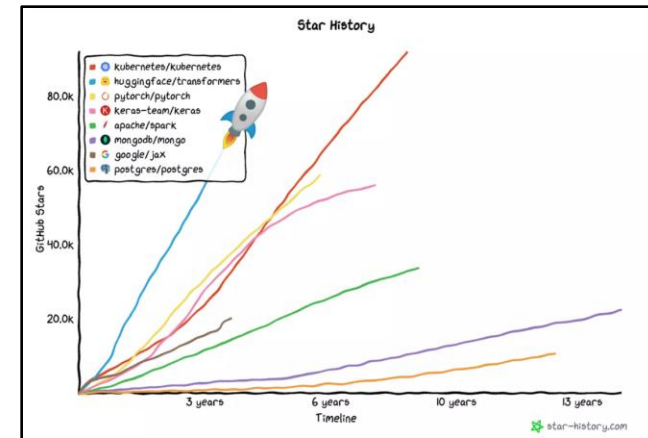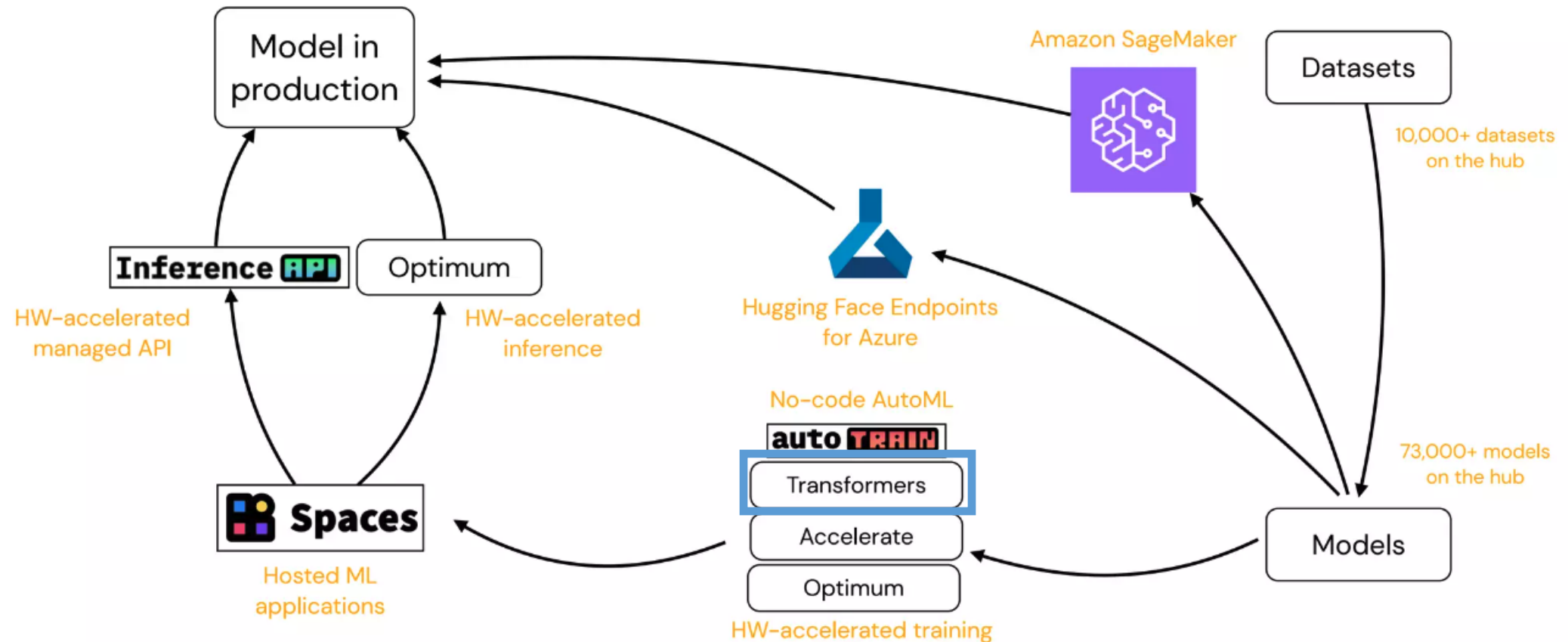
Hugging Face

- <u>Hugging Face Transformers</u>:
  - Provide APIs and tools to easily download and train state-of-the-art pre-trained models.
- <u>Hugging Face Hub</u>:
  - The Github of machine learning models.
- <u>Hugging Face Optimum</u>:
  - Provides a set of performance optimization tools for deployment on various hardware.
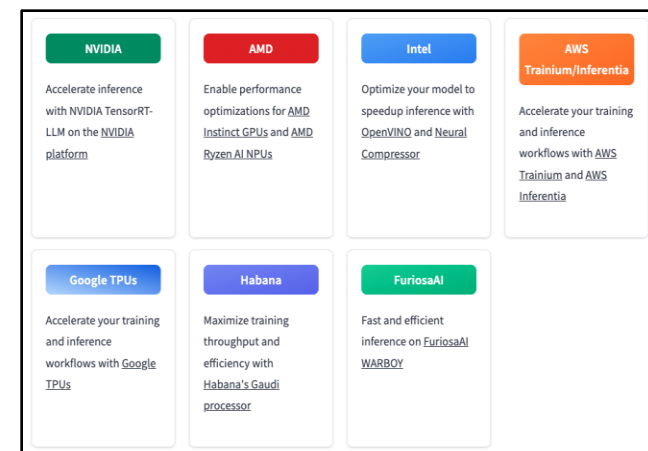
Hugging Face Ecosystem
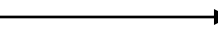
Hugging Face Transformers

Transformers Overview

- Hugging Face Transformers provides APIs and tools to download and train state-of-the-art pre-trained models easily.

- Transformers support frameworks of PyTorch, TensorFlow, and JAX for the implementation of those pre-trained models.

- Users can use a different framework at each stage of a model's life:
  - Train a model in three lines of code in one framework;
  - Load it for inference in another;
  - Models can also be exported to a format like ONNX and TorchScript for deployment in production environments.

Pipelines

- The pipelines are the easiest way to use models for inference.
- These pipelines are objects that abstract most of the complex code from the library, offering a simple API dedicated to <u>a list of tasks</u>.
- Two categories of pipeline abstractions:
  - The **pipeline()** which is the most powerful object encapsulating all other pipelines.
  - *Task-specific pipelines* are available for audio, computer vision, natural language processing, and multimodal tasks.

Audio
   Audio classification
   Automatic speech recognition
Computer vision
   Image classification
   Object detection
   Image segmentation
   Depth estimation
Natural language processing
   Text classification
   Token classification
   Question answering
   Summarization
   Translation
   Language modeling
Multimodal
   Document question answering

List of Tasks in Pipelines

Task	Description	Modality	Pipeline identifier
Text classification	assign a label to a given sequence of text	NLP	pipeline(task="sentiment-analysis")
Text generation	generate text given a prompt	NLP	pipeline(task="text-generation")
Summarization	generate a summary of a sequence of text or document	NLP	pipeline(task="summarization")
Image classification	assign a label to an image	Computer vision	pipeline(task="image-classification")
Image segmentation	assign a label to each individual pixel of an image (supports semantic, panoptic, and instance segmentation)	Computer vision	pipeline(task="image-segmentation")
Object detection	predict the bounding boxes and classes of objects in an image	Computer vision	pipeline(task="object-detection")
Audio classification	assign a label to some audio data	Audio	pipeline(task="audio-classification")
Automatic speech recognition	transcribe speech into text	Audio	pipeline(task="automatic-speech-recognition")
Visual question answering	answer a question about the image, given an image and a question	Multimodal	pipeline(task="vqa")
Document question answering	answer a question about the document, given a document and a question	Multimodal	pipeline(task="document-question-answering")
Image captioning	generate a caption for a given image	Multimodal	pipeline(task="image-to-text")

Pipelines API

```
f transformers.pipeline                                                    <source>

( task: str = None, model: Union = None, config: Union = None, tokenizer: Union = None,
feature_extractor: Union = None, image_processor: Union = None, framework: Optional = None, revision:
Optional = None, use_fast: bool = True, token: Union = None, device: Union = None, device_map = None,
torch_dtype = None, trust_remote_code: Optional = None, model_kwargs: Dict = None, pipeline_class:
Optional = None, **kwargs ) → Pipeline
```

https://huggingface.co/docs/transformers/v4.39.3/en/main_classes/pipelines#transformers.pipeline

- Pipelines are made of:
  - A **tokenizer** in charge of mapping raw textual input to token.
  - A **model** to make predictions from the inputs.
  - **Some (optional) post processing** for enhancing model's output.

Pipeline Text Classification Example

- Text Classification Example:

Code	Output

``python
from transformers import pipeline
 	
pipe = pipeline(task="text-classification",
                model="distilbert/distilbert-base-uncased-finetuned-sst-2-english")
 	
result1 = pipe("The food in HKUST is terrible")
print(result1)
 	
result2 = pipe("This course is a awesome")
print(result2)
``
	`[{'label': 'NEGATIVE', 'score': 0.9986492991447449}]`
	
	`[{'label': 'POSITIVE', 'score': 0.9998718500137329}]`

Pipeline Translation Example

- Translation Example:

Code	Output
``from transformers import pipeline

text = "translate English to French: Hugging Face is a community-based open-source platform for machine learning."
translator = pipeline(task="translation",
model="google/flan-t5-small")

text_in_chinese = translator(text)
print(text_in_chinese)``` | ```[{'translation_text': "Hugging Face est un platform d'open-source pour l'apprentissage de machine."}]``` |

Pipeline Summarization Example

- Summarization Example:

Code	Output
``from transformers import pipeline

summarizer = pipeline(task="summarization",
model="google/flan-t5-small", max_new_tokens=9)

text = "Summarization creates a shorter version of a text
from a longer one while trying to preserve most of the
meaning of the original document. "
result = summarizer(text)
print(result)``` | `[{'summary_text': 'Summarization creates a shorter version'}]` |

Pipeline Language Modeling Example

- Language modeling Example:

Code	Output
``from transformers import pipeline

generator = pipeline(task="text-generation",
model="facebook/opt-350m")

prompt_text = "Which university is best in Europe?"
result = generator(prompt_text)
print(result)``` | ```[{'generated_text': 'Which university is best in Europe?\n\nThe University of Oxford is the best university in Europe.'}]``` |

Model API

- **Model** is the base class implement the common methods for loading/saving a model:
    - From a local file or directory or;
    - From a pretrained model configuration provided by the library (downloaded from HuggingFace's AWS S3 repository).
- Model class implements a methods which are common among all the models to, e.g.:
    - Resize the input token embeddings when new tokens are added to the vocabulary;
    - Prune the attention heads of the model.
- PyTorch: **PreTrainedModel**;
- TensorFlow: **TFPreTrainedModel**;
- Flax: **FlaxPreTrainedModel**.

Model API

class transformers.**PreTrainedModel** 🔗   <source>

```
( config: PretrainedConfig, *inputs, **kwargs )
```

```
( repo_id: str, use_temp_dir: Optional = None, commit_message: Optional = None, private: Optional =
None, token: Union = None, max_shard_size: Union = '5GB', create_pr: bool = False, safe_serialization:
bool = True, revision: str = None, commit_description: str = None, tags: Optional = None,
**deprecated_kwargs )
```

Parameters

- **repo_id** (str) — The name of the repository you want to push your model to. It should contain your organization name when pushing to a given organization.

- **use_temp_dir** (bool, *optional*) — Whether or not to use a temporary directory to store the files saved before they are pushed to the Hub. Will default to True if there is no directory named like repo_id, False otherwise.

- **commit_message** (str, *optional*) — Message to commit while pushing. Will default to "Upload model".

- **private** (bool, *optional*) — Whether or not the repository created should be private.

- **token** (bool or str, *optional*) — The token to use as HTTP bearer authorization for remote files. If True, will use the token generated when running huggingface-cli login (stored in ~/.huggingface). Will default to True if repo_url is not specified.

- **max_shard_size** (int or str, *optional*, defaults to "5GB") — Only applicable for models. The maximum size for a checkpoint before being sharded. Checkpoints shard will then be each of size lower than this size. If expressed as a string, needs to be digits followed by a unit (like "5MB"). We default it to "5GB" so that users can easily load models on free-tier Google Colab instances without any CPU OOM issues.

- **create_pr** (bool, *optional*, defaults to False) — Whether or not to create a PR with the uploaded files or directly commit.

- **safe_serialization** (bool, *optional*, defaults to True) — Whether or not to convert the model weights in safetensors format for safer serialization.

- **revision** (str, *optional*) — Branch to push the uploaded files to.

- **commit_description** (str, *optional*) — The description of the commit that will be created

- **tags** (List[str], *optional*) — List of tags to push on the Hub.

Tokenizer API

- A tokenizer is in charge of preparing the inputs for a model. The library contains tokenizers for all the models.

- The base class e.g., **PreTrainedTokenizer** implement the common methods for encoding string inputs as model inputs and instantiating/saving tokenizers either from a local file or directory or from a pretrained tokenizer provided by the library (downloaded from HuggingFace's AWS S3 repository).

- Tokenizer implements the main methods for using all the tokenizers:
  - Tokenizing (splitting strings in sub-word token strings), converting tokens strings to ids and back, and encoding/decoding (i.e., tokenizing and converting to integers).
  - Adding new tokens to the vocabulary in a way that is independent of the underlying structure (BPE, SentencePiece…).
  - Managing special tokens (like mask, beginning-of-sentence, etc.): adding them, assigning them to attributes in the tokenizer for easy access and making sure they are not split during tokenization.

Tokenizer API

class transformers.**PreTrainedTokenizer** 🔗   <source>

(**kwargs)

Class attributes (overridden by derived classes)

- **vocab_files_names** (`Dict[str, str]`) — A dictionary with, as keys, the `__init__` keyword name of each vocabulary file required by the model, and as associated values, the filename for saving the associated file (string).

- **pretrained_vocab_files_map** (`Dict[str, Dict[str, str]]`) — A dictionary of dictionaries, with the high-level keys being the `__init__` keyword name of each vocabulary file required by the model, the low-level being the `short-cut-names` of the pretrained models with, as associated values, the `url` to the associated pretrained vocabulary file.

- **max_model_input_sizes** (`Dict[str, Optional[int]]`) — A dictionary with, as keys, the `short-cut-names` of the pretrained models, and as associated values, the maximum length of the sequence inputs of this model, or `None` if the model has no maximum input size.

- **pretrained_init_configuration** (`Dict[str, Dict[str, Any]]`) — A dictionary with, as keys, the `short-cut-names` of the pretrained models, and as associated values, a dictionary of specific arguments to pass to the `__init__` method of the tokenizer class for this pretrained model when loading the tokenizer with the from_pretrained() method.

- **model_input_names** (`List[str]`) — A list of inputs expected in the forward pass of the model.

- **padding_side** (`str`) — The default value for the side on which the model should have padding applied. Should be `'right'` or `'left'`.

- **truncation_side** (`str`) — The default value for the side on which the model should have truncation applied. Should be `'right'` or `'left'`.

Generation API

- Each framework has a generate method for text generation implemented in their respective **GenerationMixin** class:
  - PyTorch **generate**() is implemented in **GenerationMixin**.
  - TensorFlow **generate**() is implemented in **TFGenerationMixin**.
  - Flax/JAX **generate**() is implemented in **FlaxGenerationMixin**.
- Regardless of the framework of choice, one can parameterize the generate method with a **GenerationConfig** class instance.

Customize Text Generation

- Some of the commonly adjusted parameters include:
  - **max_new_tokens**: the maximum number of tokens to generate. In other words, the size of the output sequence, not including the tokens in the prompt. Or you can implement some **StoppingCriteria**.
  - **num_beams**: by specifying a number of beams higher than 1, you are effectively switching from greedy search to beam search. This strategy evaluates several hypotheses at each time step and eventually chooses the hypothesis that has the overall highest probability for the entire sequence. This has the advantage of identifying high-probability sequences that start with a lower probability initial tokens and would've been ignored by the greedy search.
  - **do_sample**: if set to True, this parameter enables decoding strategies such as multinomial sampling, beam-search multinomial sampling, Top-K sampling and Top-p sampling. All these strategies select the next token from the probability distribution over the entire vocabulary with various strategy-specific adjustments.
  - **num_return_sequences**: the number of sequence candidates to return for each input. This option is only available for the decoding strategies that support multiple sequence candidates, e.g. variations of beam search and sampling. Decoding strategies like greedy search and contrastive search return a single output sequence.

Customize Text Generation: Beam Search

- A variant of breadth-first search (BFS) to expand the search space of greedy search.
  - Greedy search: in every generation step, keep the token with the highest probability.

- Beam search: keep top-k (**num_beams**=k) tokens for generation.
  - For the first decode step, the k words with the highest current target value (e.g., conditional probability) are selected as the first word of the candidate output sequence.
  - For each subsequent decode step, based on the output sequence of the previous step, select k largest target value (e.g., conditional probability) in all combinations as the candidate output sequence by the current step.
  - It always keeps k candidates and picks the best of the candidates at the end.

Gready Search Example

- Three tokens: {I, L, U}

	Step 1	Step 2	Step 3
I	**0.6**	0.2	0.05
L	0.3	**0.7**	0.05
U	0.1	0.1	**0.9**

Beam Search Example

- Three tokens: {I, L, U}
- k = 2

Sampling

- **Sampling**: instead of determinstic selecting the largest tokens, we use a random number generator to sample tokens following the distribution computed by the LM.

- **Top-k sampling**: only the k (e.g., k = 6) most likely next words are filtered to be sampled.

- **Top-p sampling**: chooses from the smallest possible set of words whose cumulative probability exceeds the probability p (e.g., p = 0.92).
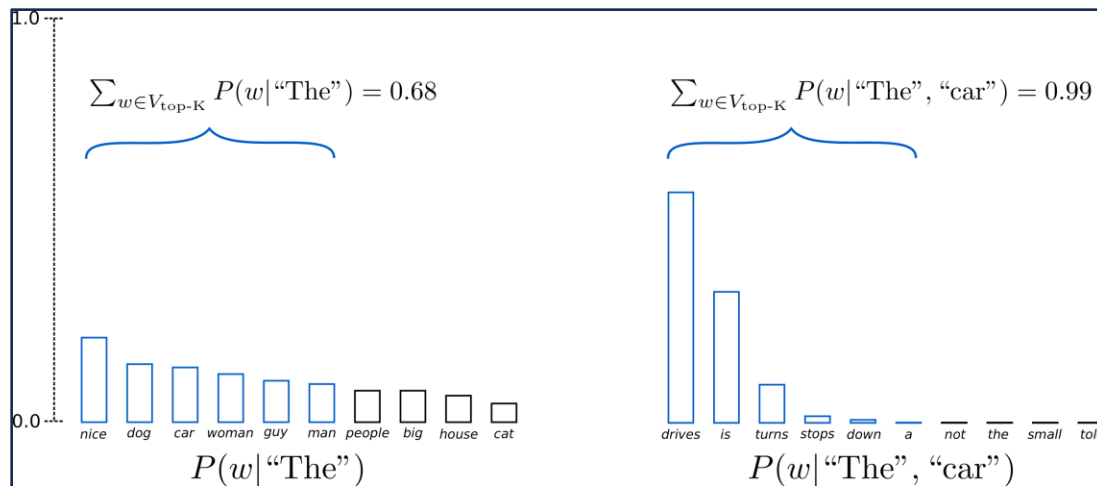
Top-k sampling

Top-p sampling

24
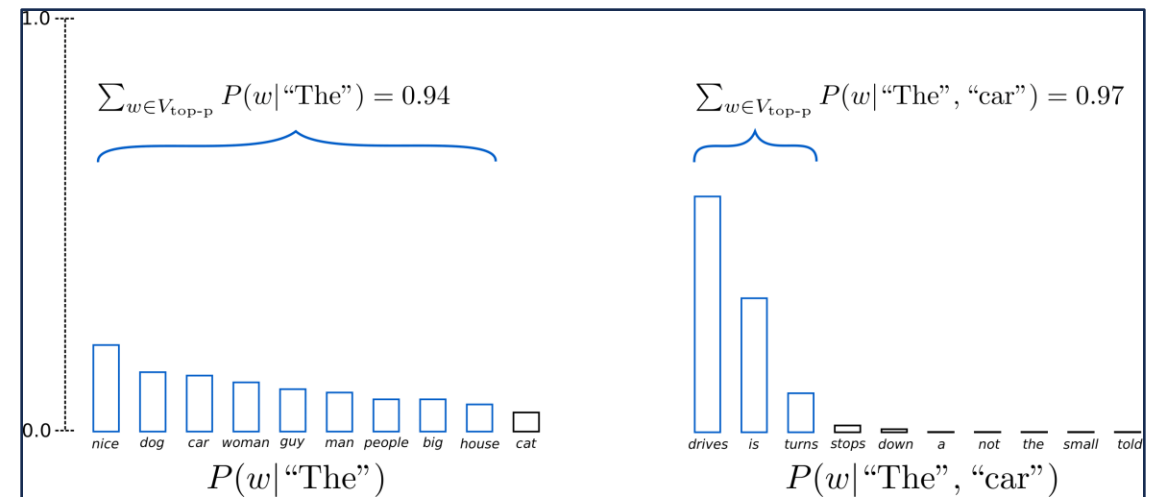
An Inference Example

Define the Generate Function

Code

```python
import argparse
import torch
from transformers import AutoTokenizer,AutoModelForCausalLM
from transformers.models.opt.modeling_opt import *


def generate(task_info, device, model, tokenizer):
    contexts = task_info["prompt_seqs"]
    inputs = tokenizer(contexts, return_tensors="pt").to(device)
    print(f"start_ids: length ({inputs.input_ids.shape[0]}) ids: {inputs.input_ids}")
    input_length = inputs.input_ids.shape[1]

    outputs = model.generate(
        **inputs, do_sample=True, top_p=task_info['top_p'],
        temperature=1.0, top_k=1,
        max_new_tokens=task_info["output_len"],
        return_dict_in_generate=True,
        output_scores=True,  # return logit score
        output_hidden_states=False,  # return embeddings
    )
    print(f"[INFO] raw output: {outputs.keys()} {len(outputs)}, {outputs[0].shape},
({outputs[1][0].shape},{outputs[1][1].shape}) {len(outputs[2])}")
    token = outputs.sequences[0, input_length:]  # exclude context input from the output
    print(f"[INFO] raw token: {token}")
    output = tokenizer.decode(token)
    print(f"[INFO] \n[Context]\n{contexts}\n\n[Output]\n{output}\n")
```

Test the Model

Code

```python
def test_model(args):
    print(f"<test_model> initialization start")
    device = torch.device(args.get('device', 'cpu'))
    tokenizer = AutoTokenizer.from_pretrained(args['hf_model_name'])
    model = AutoModelForCausalLM.from_pretrained(args['hf_model_name'])
    model = model.to(device)
    torch.manual_seed(0)
    task_info = {
        "seed": 0,
        "prompt_seqs": None,
        "output_len": 16,
        "beam_width": 1,
        "top_k": 50,
        "top_p": 1,
        "beam_search_diversity_rate": 0,
        "len_penalty": 0,
        "repetition_penalty": 1.0,
        "stop": args.get("stop", []),
        "logprobs": 5,
    }
    print(f"<test_model> initialization done")

    if args["interactive"]:
        while True:
            prompt_data = input("Please enter the prompt input:\n")
            task_info["prompt_seqs"] = prompt_data
            generate(task_info, device, model, tokenizer)
    else:
        generate(task_info, device, model, tokenizer)
```

The Main Entrance

Code

```python
if __name__ == "__main__":
    parser = argparse.ArgumentParser()
    parser.add_argument('--hf_model_name', type=str, default='facebook/opt-350m',
                        help='hugging face model name (used to load config).')
    args = parser.parse_args()
    test_model(args={
        "hf_model_name": args.hf_model_name,
        "interactive": True,
        "device": "cpu",
        "dtype": torch.float32,
    })
```

References

- https://www.youtube.com/watch?v=fckyXntHy1s&t=1070s
- https://huggingface.co/docs/transformers/index
- https://huggingface.co/docs/transformers/generation_strategies
- https://huggingface.co/blog/how-to-generate