LLM Application Development

Arin Ghazarian
Chapman University

Hugging Face Library

Hugging Face

- A Premier Member of the PyTorch Foundation, an organization that supports the open-source PyTorch framework and ecosystem.
- The most popular framework to work with LLMs and Transformers model

Hugging Face Hub

- The Hugging Face Hub is a central repository via which you can discover, use, and contribute new state-of-the-art models and datasets. It hosts a wide variety of models, with more than 10,000 publicly available.
- The storage backend is Git
- When you upload a new model on the Hub, it automatically deploys a hosted Inference API for that model. Anyone in the community is welcome to test it directly on the model’s page using custom inputs and appropriate widgets.
- Sharing and using any public model on the Hub is completely free
- If you wish to share models privately in your organization then you need a paid plan form Hugging Face

Hugging Face Hub

- The huggingface_hub library is a library for interacting with the Hugging Face Hub, which is a collections of git-based repositories (models, datasets or Spaces).
  - Git-based approach: is led by the Repository class. This method uses a wrapper around the git command with additional functions specifically designed to interact with the Hub.
  - HTTP-based approach: involves making HTTP requests using the HfApi client.
    \>huggingface-cli repo create my-model -- type model

Hugging Face Datasets

- In addition to models; the Hugging Face hub also has datasets in different languages.
- https://huggingface.co/datasets

This command downloads and caches the dataset, by default in ~/.cache/huggingface/datasets. you can customize your cache folder by setting the HF_HOME environment variable.

```
from datasets import load_dataset

raw_datasets = load_dataset("glue", "mrpc")
```

Hugging Face Datasets

- To load a remote or local dataset:

```
url = "https://github.com/crux82/squad-it/raw/master/"

data_files = {

    "train": url + "SQuAD_it-train.json.gz",

    "test": url + "SQuAD_it-test.json.gz",

}

squad_it_dataset = load_dataset("json", data_files=data_files, field="data")
```

Hugging Face Datasets

- Datasets provides functionality for sampling, filtering, and mapping:

```
drug_sample = drug_dataset["train"].shuffle(seed=40).select(range(100))


def lowercase_condition(example):
    return {"condition": example["condition"].lower()}
drug_dataset.map(lowercase_condition)
drug_dataset = drug_dataset.filter(lambda x: x["condition"] is not None)
```

Hugging Face Datasets

- Hugging Face stores the data using the Apache Arrow library
- Using Dataset.set_format() function, you can change the output format of the dataset from Apache Arrow to your desired format such as Pandas. The formatting is done in place:

```
drug_dataset.set_format("pandas")

drug_dataset["train"][:3]
```

Hugging Face Datasets

- Hugging Face Datasets has two functionality which helps handling large input files:
  - **Memory-mapped files**
    - Provides a mapping between RAM and filesystem storage that allows the library to access and operate on elements of the dataset without needing to fully load it into memory.
    - Relies on Apache Arrow memory format and pyarrow library for these capabilities
    - Datasets treats each dataset as a memory-mapped file by default
  - **Streaming**
    - Instead of Dataset , it returns an IterableDataset

```
pubmed_dataset_streamed = load_dataset(on", data_files=data_files, split="train", streaming=True)



next(iter(pubmed_dataset_streamed))



tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")

tokenized_dataset = pubmed_dataset_streamed.map(lambda x: tokenizer(x["text"]))

next(iter(tokenized_dataset))
```

Datasets

```
# Skip the first 1,000 examples and include the rest in the training set
train_dataset = shuffled_dataset.skip(1000)
# Take the first 1,000 examples for the validation set
validation_dataset = shuffled_dataset.take(1000)


#combining two datasets
combined_dataset = interleave_datasets([dataset1, dataset2])
```

Datasets

- You can create and load your own datasets:

```
your_dataset_object.push_to_hub("your_dataset_name")
```

- Similar to models, you should create a dataset card and add tags to make it searchable

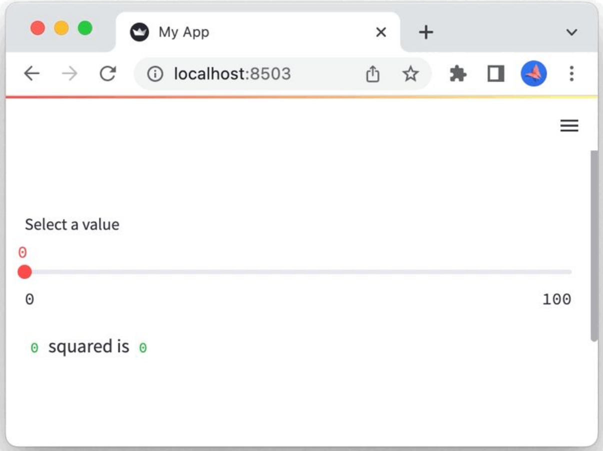
Hugging Face Spaces

- Hugging Face Spaces is a platform that allows users to create and host machine learning (ML) demo applications on their profile or organization's profile. Spaces can be used for a variety of purposes, including:
  a. To create your ML portfolio
  b. Showcase your projects at conferences or to stakeholders
  c. Work collaboratively with other people in the ML ecosystem
- You can use different libraries to quickly create a UI for your ML app:
  - Streamlit
  - Gradio
  - Static

Streamlit

- Streamlit is an open-source Python framework for data scientists and AI/ML engineers to deliver dynamic data apps with only a few lines of code:

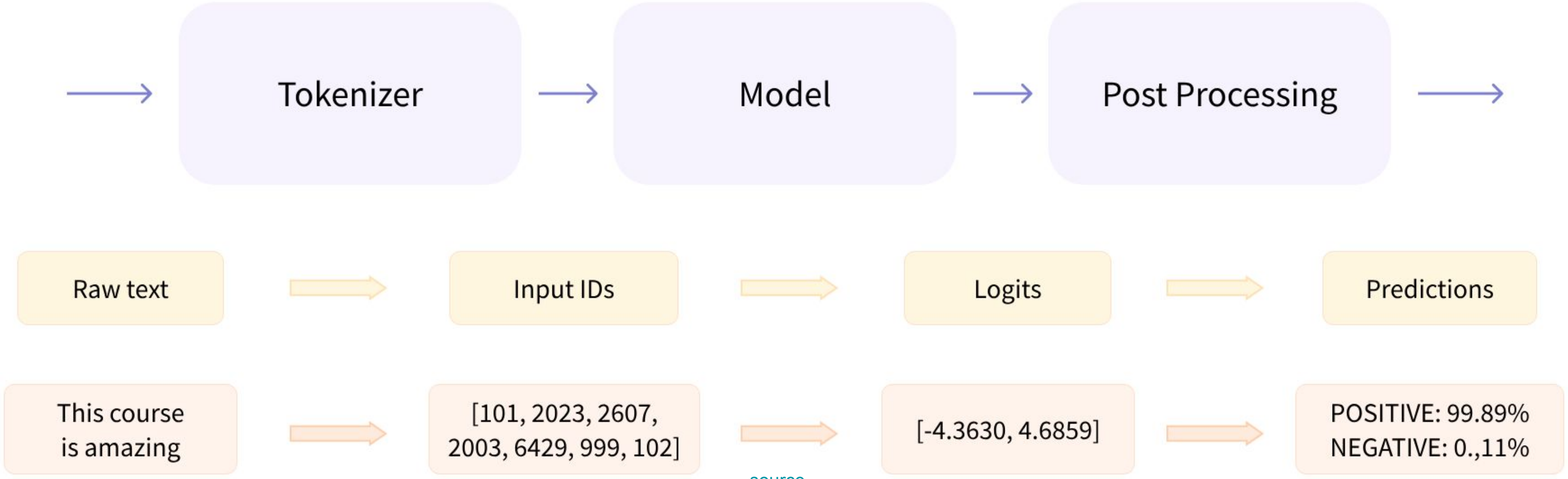
```
import streamlit as st

x = st.slider("Select a value")

t.write(x, "squared is", x * x)
```



Hugging Face Transformers Module

- Provides a wide variety of pre-trained models and architectures like BERT, GPT-2, T5, and others
- Supports multiple languages and tasks like text classification, question-answering, text generation, translation, and more.
- Install the transformers library via pip:

```
pip install transformers
```


Tokenizer
Model
Post Processing
Raw text
Input IDs
Logits
Predictions
This course is amazing
[101, 2023, 2607, 2003, 6429, 999, 102]
[-4.3630, 4.6859]
POSITIVE: 99.89%
NEGATIVE: 0.,11%

source

Tokenizers

- Convert the text inputs into numbers
  - Splitting the input into tokens: words, subwords, or symbols (like punctuation)
  - Mapping each token to an integer
  - Adding additional inputs that may be useful to the model

```
from transformers import AutoTokenizer

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

tokenizer = AutoTokenizer.from_pretrained(checkpoint)
```

Tokenizers

- You can use Transformers with different ML framework as a backend: PyTorch or TensorFlow, or Flax for some models.
- However, Transformer models only accept *tensors* as input
- To specify the type of tensors we want to get back (PyTorch, TensorFlow, or plain NumPy), we use the `return_tensors` argument:

```
raw_inputs = [
    "I always wanted to have an Ice cream.",
    "Hugging face goes well with an Ice cream",]
inputs = tokenizer(raw_inputs, padding=True, truncation=True, return_tensors="pt")
print(inputs)
```

Tokenizers: return_tensors

```
# Returns PyTorch tensors
model_inputs = tokenizer(sequences, padding=True, return_tensors="pt")

# Returns TensorFlow tensors
model_inputs = tokenizer(sequences, padding=True, return_tensors="tf")

# Returns NumPy arrays
model_inputs = tokenizer(sequences, padding=True, return_tensors="np")
```

Decoding the Output

```
from transformers import AutoTokenizer


tokenizer = AutoTokenizer.from_pretrained("bert-base-cased")

sequence = "Using a Transformer network is simple"

tokens = tokenizer.tokenize(sequence)

print(tokens)


ids = tokenizer.convert_tokens_to_ids(tokens)


decoded_string = tokenizer.decode([7993, 170, 11303, 1200, 2443, 1110, 3014])

print(decoded_string)
```

We can also use BertTokenizer directly, the AutoTokenizer class will pick the proper tokenizer class based on the checkpoint name

from transformers import BertTokenizer

tokenizer = BertTokenizer.from_pretrained("bert-base-cased")

Padding

- The following list of lists cannot be converted to a tensor because the sequences have different lengths.

```
batched_ids = [
    [300, 400, 600],
    [300, 200]
]
```

- To address this issue, we'll use padding to ensure our tensors have a rectangular shape.

Padding

- Padding ensures all sentences have the same length by adding a special word called the padding token to those that are shorter:

```
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)

sequence1_ids = [[200, 200, 200]]

sequence2_ids = [[200, 200]]

batched_ids = [

    [200, 200, 200],

    [200, 200, tokenizer.pad_token_id],

]

print(model(torch.tensor(sequence1_ids)).logits)

print(model(torch.tensor(sequence2_ids)).logits)

print(model(torch.tensor(batched_ids)).logits)
```

Padding

```
# Will pad the sequences up to the maximum sequence length
model_inputs = tokenizer(sequences, padding="longest")


# Will pad the sequences up to the model max length
# (512 for BERT or DistilBERT)
model_inputs = tokenizer(sequences, padding="max_length")


# Will pad the sequences up to the specified max length
model_inputs = tokenizer(sequences, padding="max_length", max_length=8)
```

Truncate

```
sequences = ["I've been waiting for a HuggingFace course my whole life.", "So have I!"]

# Will truncate the sequences that are longer than the model max length
# (512 for BERT or DistilBERT)
model_inputs = tokenizer(sequences, truncation=True)

# Will truncate the sequences that are longer than the specified max length
model_inputs = tokenizer(sequences, max_length=8, truncation=True)
```

Transformers Models

- Is used to download a pretrained model

```
from transformers import AutoModel

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"
model = AutoModel.from_pretrained(checkpoint)
```

Model Architectures in Hugging Face

- This AutoModel architecture contains only the base Transformer module
  - Given some inputs, it outputs the hidden states (or as features): a high-dimensional vector representing the contextual understanding of that input by the Transformer model

```
outputs = model(**inputs)

print(outputs.last_hidden_state.shape)
```

Sequence length: in Hugging Face refers to the number of characters or tokens in a language model sequence. Most Transformer models can handle sequences of up to 1024 or 512 tokens. the number of tokens that are processed by the transformer together. Each token could be a word embedding

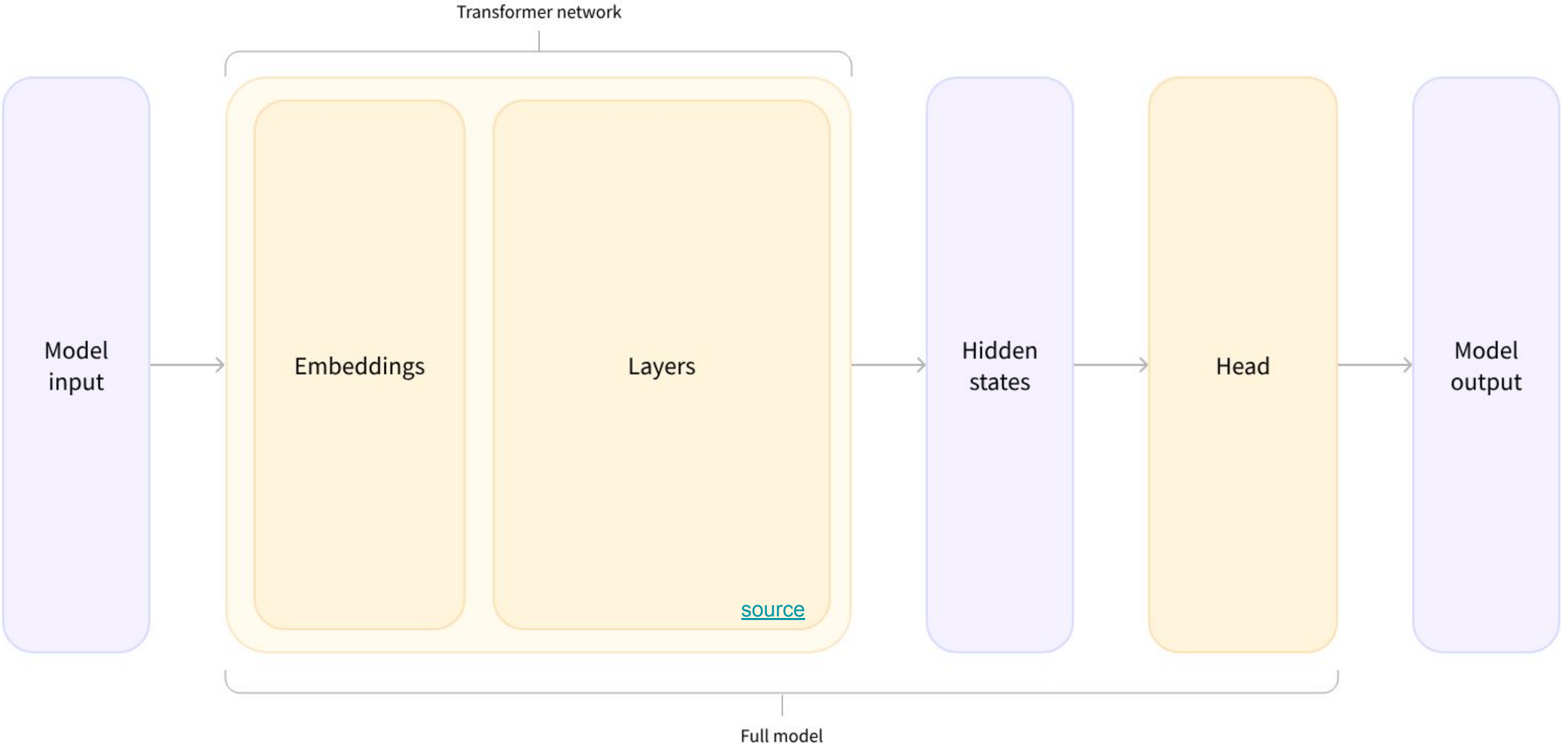
Output: torch.Size([2, 16, 768])

Hidden size: The vector dimension of each model input.

Batch size: The number of sequences processed at a time (2 in our example).

Transformers Architecture

- The model heads take the high-dimensional vector of hidden states as input and project them onto a different



Model

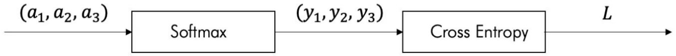
```
from transformers import AutoModelForSequenceClassification

checkpoint = "distilbert-base-uncased-finetuned-sst-2-english"

model = AutoModelForSequenceClassification.from_pretrained(checkpoint)

outputs = model(**inputs)

print(outputs.logits.shape)
```

Output: torch.Size([2, 2])

Note: Since we have just two sentences and two labels, the result we get from our model is of shape 2 x 2.

Note: The output.logits are not probabilities but logits (unnormalized scores). We need to apply softmax to convert them to probabilities. Transformer models output logits because the loss function used for training typically combines the final activation function, like SoftMax, with the actual loss function, such as cross-entropy:

```
predictions = torch.nn.functional.softmax(outputs.logits, dim=-1)
```



$$
\begin{aligned}
& L(a_1, a_2, a_3) \\
& = -t_1 \log y_1 - t_2 \log y_2 - t_3 \log y_3 \\
& = -t_1 \log \frac{e^{a_1}}{e^{a_1}+e^{a_2}+e^{a_3}} - t_2 \log \frac{e^{a_2}}{e^{a_1}+e^{a_2}+e^{a_3}} - t_3 \log \frac{e^{a_3}}{e^{a_1}+e^{a_2}+e^{a_3}} \\
& = -t_1 \log e^{a_1} - t_2 \log e^{a_2} - t_3 \log e^{a_3} + (t_1 + t_2 + t_3) \log(e^{a_1} + e^{a_2} + e^{a_3}) \\
& = -t_1 a_1 - t_2 a_2 - t_3 a_3 - t_3 a_3 + \log(e^{a_1} + e^{a_2} + e^{a_3})
\end{aligned}
$$

source

Model

- There are many different architectures available in Transformers library
  a. Model (retrieve the hidden states)
  b. ForCausalLM
  c. ForMaskedLM
  d. ForMultipleChoice
  e. ForQuestionAnswering
  f. ForSequenceClassification
  g. ForTokenClassification
  h. etc.

Pipelines

- Pipeline groups together three steps: preprocessing, passing the inputs through the model, and postprocessing, and is an easy way to use AI model functionalities in a single line of code

```
import torch

from transformers import pipeline



speech_recognizer = pipeline("automatic-speech-recognition", model="facebook/wav2vec2-base-960h")
```

Pipelines

```
from transformers import pipeline


classifier = pipeline("zero-shot-classification")
classifier(
    "The new movie was awesome!",
    candidate_labels=["education", "politics", "business","Entertainment"],
```

Pipelines

```
from transformers import pipeline

translator = pipeline("translation", model="Helsinki-NLP/opus-mt-fr-en")

translator("C'est mon ami")
```

Pipelines

- Connects a model with its necessary preprocessing and postprocessing steps, allowing us to directly input any text and get an intelligible answer
- You can use the pipeline() out-of-the-box for many tasks across different modalities, some of which are shown in the table

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")

Sharing Models and Datasets on the Hub

- There are three ways to go about creating new model repositories:
  a. push_to_hub API
  b. huggingface_hub Python library
  c. Web interface

push_to_hub API

- Insert to hub as you are training the model automatically:

```
from transformers import TrainingArguments

training_args = TrainingArguments("bert-finetuned-mrpc", save_strategy="epoch", push_to_hub=True)
```

- You can also call push_to_hub method on tokenizer, model and other objects:

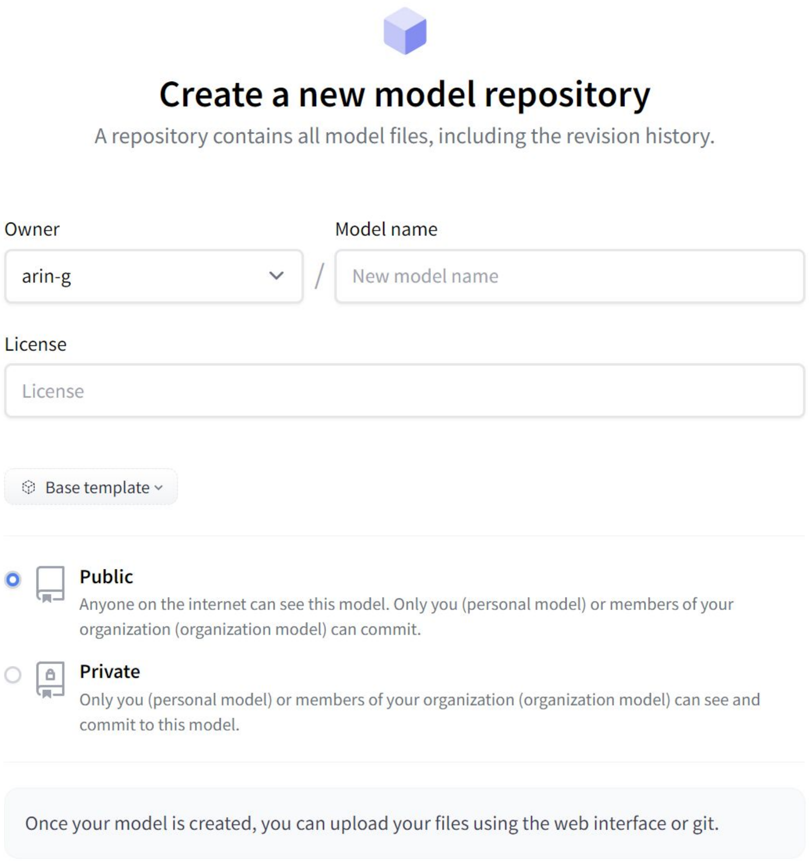
```
model.push_to_hub("dummy-model")

tokenizer.push_to_hub("dummy-model", organization="your_name_space")
```

huggingface_hub Python library

```
from huggingface_hub import create_repo

create_repo("dummy-model", organization="huggingface")
```

Web Interface

- You can easily create repositories and add files using a web GUI
- You can upload model files using huggingface_hub or through git commands.



Fine Tuning using Hugging Face

- We can choose one dataset and one model from the hub and fine tune it

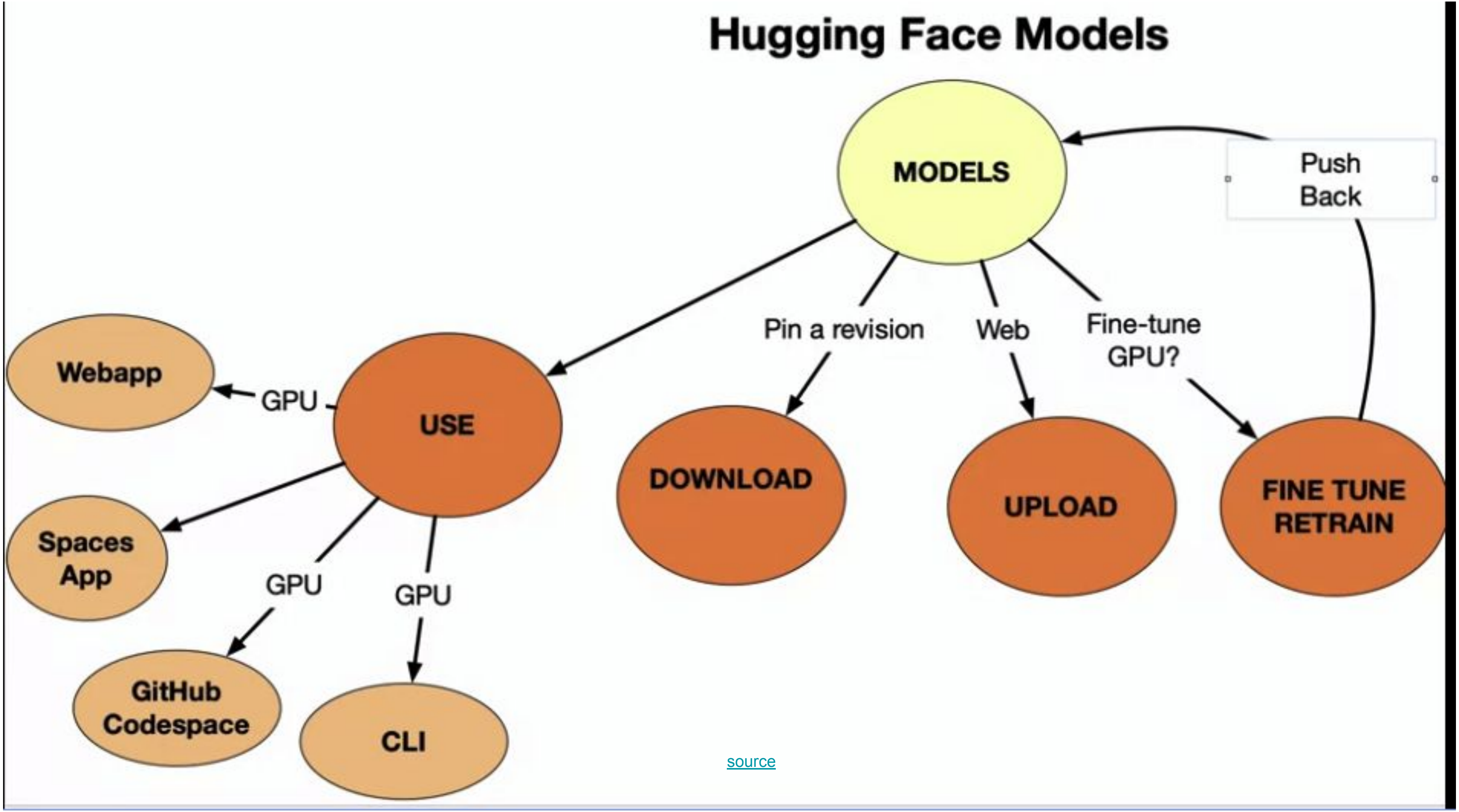
```
import torch
from transformers import AdamW, AutoTokenizer, AutoModelForSequenceClassification
# Same as before
checkpoint = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSequenceClassification.from_pretrained(checkpoint)
sequences = [
    "I've been waiting for a HuggingFace course my whole life.",
    "This course is amazing!",
]
batch = tokenizer(sequences, padding=True, trun cation=True, return_tensors="pt")
# This is new
batch["labels"] = torch.tensor([1, 1])
optimizer = AdamW(model.parameters())
loss = model(**batch).loss
loss.backward()
optimizer.step()
```

Model Cards

- Do not forget to create a model card for your repo!
- The model card usually starts with a brief explanation of what the model is for, followed the following sections:
    - Model description
    - Intended uses & limitations
    - How to use
    - Limitations and bias
    - Training data
    - Training procedure
    - Evaluation results
- Example: GPT2

Demo: Fine Tuning

- https://huggingface.co/learn/nlp-course/chapter3/3?fw=pt
- https://huggingface.co/learn/nlp-course/chapter3/4?fw=pt


Hugging Face Models
MODELS
Push
Back
USE
Pin a revision
Web
Fine-tune
GPU?
Webapp
GPU
DOWNLOAD
UPLOAD
FINE TUNE
RETRAIN
Spaces
App
GPU
GPU
GitHub
Codespace
CLI

source

Demo: Hugging Face

- http://localhost:8888/notebooks/jupyter-notebooks/chapman-generative-AI/hugging-faces.ipynb

Online Resources

- https://huggingface.co/learn/nlp-course/chapter1/1
- https://huggingface.co/docs/transformers/en/quicktour
- https://github.com/TirendazAcademy/Hugging-Face-Tutorials