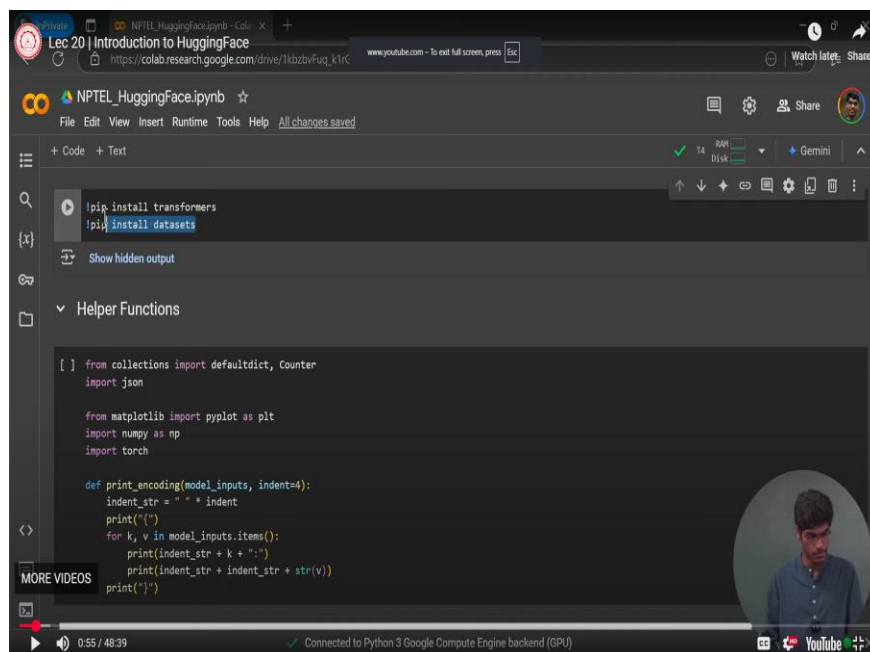


Introduction to Large Language Models (LLMs)
Prof. Tanmoy Chakraborty, Prof. Soumen Chakraborti
Department of Computer Science & Engineering
Indian Institute of Technology, Delhi
Lecture 20
Introduction to HuggingFace

Hi, everyone. In this tutorial, we'll be discussing kind of introduction to HuggingFace library. So HuggingFace is a very useful library when we work with basically the transformer-based model, mostly the transformer-based model. All these models, open-source models, data sets are available on something called HuggingFace Hub. So we will see how we can use this HuggingFace library to load the models, to fine-tune them using some toy examples, how to load the data sets and how to use it for inference. So let's start.



```
!pip install transformers
!pip install datasets

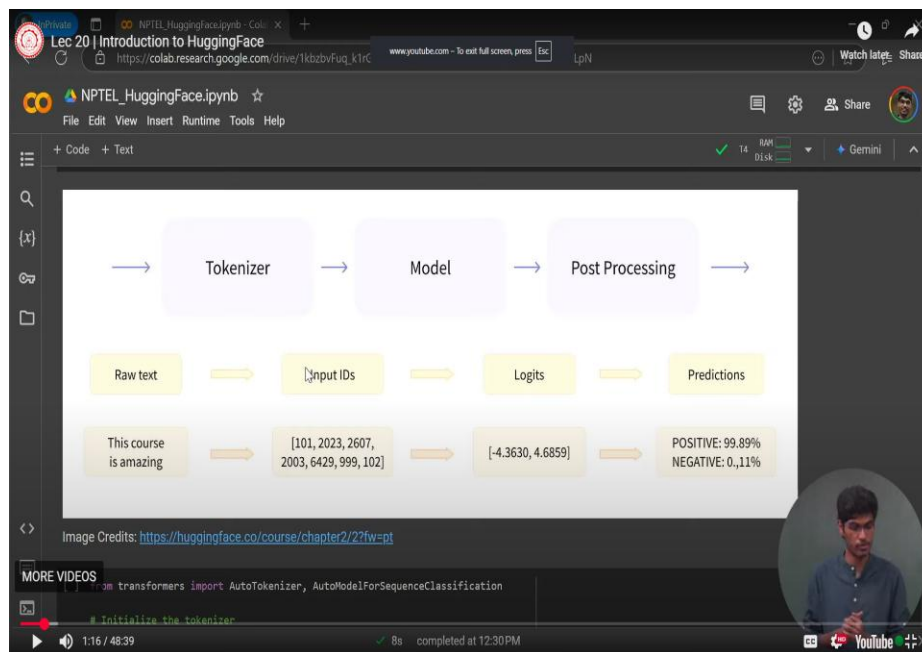
from collections import defaultdict, Counter
import json

from matplotlib import pyplot as plt
import numpy as np
import torch

def print_encoding(model_inputs, indent=4):
    indent_str = " " * indent
    print("(")
    for k, v in model_inputs.items():
        print(indent_str + k + ":")
        print(indent_str + indent_str + str(v))
    print(")
```

So here... That package in HuggingFace, which deals with transformer-based models, is called Transformers.

And there's a package called Datasets, which deals with all the open-source datasets. So we need to first install them.



Let's first look at the whole pipeline, the whole flow of how we process an input. So, first we have this raw text. This course is amazing.

And then we have a tokenization algorithm which breaks the text into some tokens and map it to some numbers right. So these numbers represent a token right. So this code is amazing suppose we apply some tokenization algorithm say like byte pen code or something or a sentence piece tokenization something like that and it maps it to a sequence of tokens and tokens are represented as numbers right. So these tokens numbers are basically kind of dictionary mapping only, nothing else. So 101, say, this tells this is a token, and its number is 101, something like that.

So we have then a list of input IDs. And then these are passed on to the model. So when the model gets the input IDs, which is a list of token numbers, What it does is it goes to its embedding matrix, does a lookup, so the embedding dimension of the token is already stored in the embedding matrix of the pre-trained model. If it is not pre-trained, if we are looking to train it from scratch, then initialization is some random initialization or some

informed initialization like Xavier initialization or something like that. And then when you train the model, these embeddings are also updated.

Embeddings of the token are also updated. So for now, let us assume that the model is pre-trained. So once we pass the input IDs, the model maps the token IDs to token embeddings. and then the token embeddings are passed on to the model position encoding is added in case of transformers and whatever the model architecture is there whether it's an encoder decoder model it's a decoder only model whether it's an encoder only model it is processed accordingly. We have already seen in our lecture in I guess one or two weeks ago that how do you implement transformers from scratch using pytorch right, there we saw how within the model how a transformer layer is implemented right.

we saw every component of it, multi-reduction positional encoding, layer normalization, encoder block, decoder block, we saw all that. So now we are putting all this together into a single model right and there are simple variations within the model Like, say, GPT-2 doesn't use rotational positional encoding, or ROPE. Whereas LLaMA uses rope, Mistral uses sliding window attention. So there are all these slight variations in the architecture and the types of attention. They used llamas like group query attention.

So all these variations are present. But let us assume that within the model, we have a kind of function. A module that takes the input IDs maps them to the embedding. Add any equations and encodings that are defined for that model. processes it, and the final output is logits.

Logits over the vocabulary tokens. So suppose there are 32,000 tokens in the model's vocabulary. So, the output will be a set—a series of logits over these 32,000 tokens. And then it is passed through the softmax function. The softmax maps these 32,000 logits into a probability distribution over the 32,000 tokens.

So it will say, "Okay, so token I has a probability of 0.3." Token this has a probability of 0.2, and just like that, there is a probability distribution. Over the entire vocabulary set, and then you can do some post-processing.

So if you are ready to start, we will take the example of the. Sentiment classification task. Try to perform sentiment classification using an encoder-only model like BERT. We'll use Robota here. So, Robota, just think of it as a variation of BERT.

The architecture is the same as that of BERT. We have already discussed that BERT is trained using masked language modeling. And so, what BERT does is give you a representation of an input sentence. And then you can train some linear layers on top of BERT. To perform any classification task or anything you want with the sentence representation.

```
from transformers import AutoTokenizer, AutoModelForSequenceClassification

# Initialize the tokenizer
tokenizer = AutoTokenizer.from_pretrained("bert-base-uncased")
# Initialize the model
model = AutoModelForSequenceClassification.from_pretrained("bert-base-uncased")
```

/usr/local/lib/python3.10/dist-packages/huggingface_hub/utils/_auth.py:94: UserWarning: The secret 'HF_TOKEN' does not exist in your Colab secrets. To authenticate with the Hugging Face Hub, create a token in your settings tab (https://huggingface.co/settings/tokens), set it as secret in your Goo. You will be able to reuse this secret in all of your notebooks. Please note that authentication is recommended but still optional to access public models or datasets. warnings.warn(
tokenizer_config.json: 100% 256/256 [00:00<00:00, 7.33kB/s]
tokenizer_config.json: 100% 687/687 [00:00<00:00, 27.5kB/s]
Warning: you are connected to a GPU runtime, but not utilizing it. Change to a standard runtime X
6:12 / 48:39

This is the flow, so once you have the sentence representation, You can use it to make some predictions or something like that, right? So let us now load the model and tokenizer properly. So, there is something in the transformer library called AutoTokenizer. So what the auto tokenizer does is, when you give the model path, right? We'll see where the model's path comes from. You give the model the path; it automatically has some kind of dictionary defined. So it automatically looks into what kind of tokenizer is trained for that model, right? So during pre-training, there is a particular tokenizer organization algorithm.

So when you're doing inference, you need to use the same organizational algorithm. So the auto tokenizer handles this. You can also explicitly say that, for now, we're using, say,

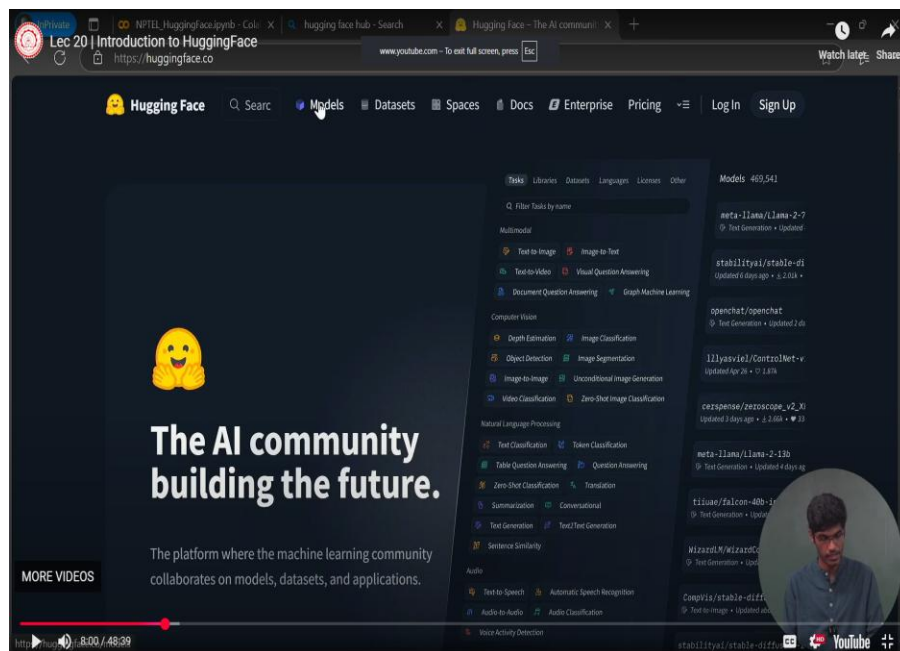
Robata. So you can explicitly define which tokenizer you need to use. But the AutoTokenizer makes our job easy.

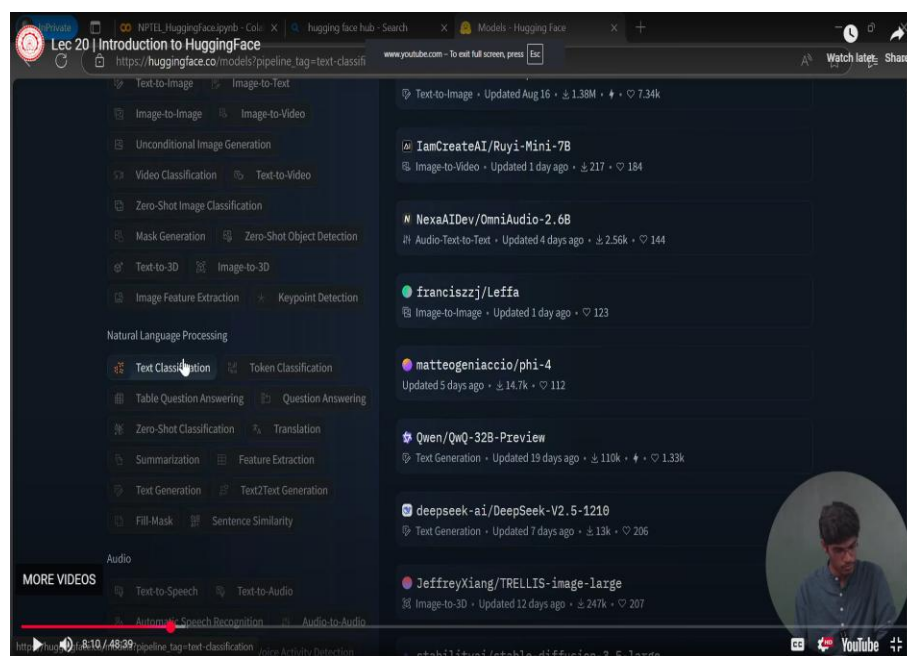
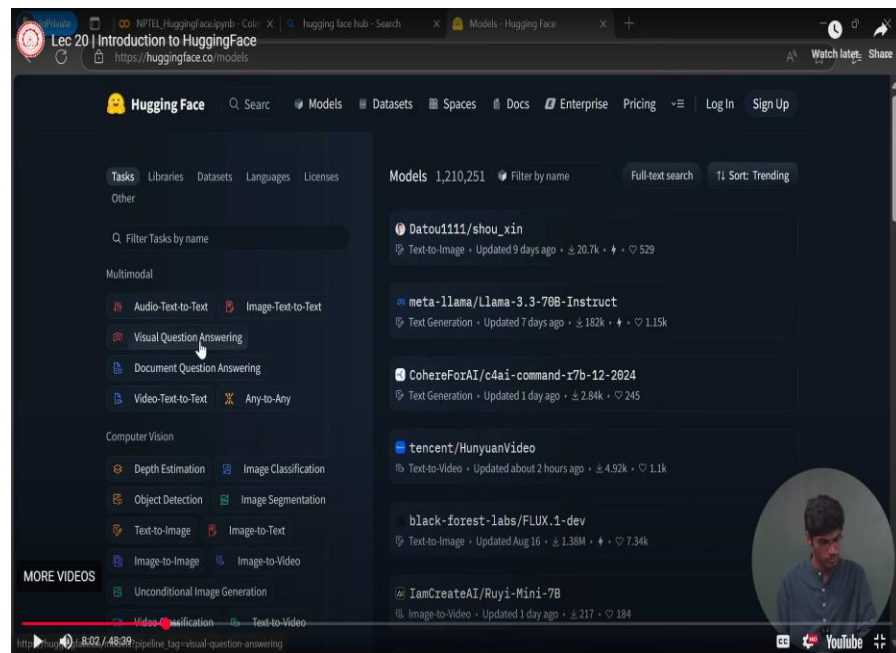
And we have the Auto Model, which, again, takes in the path. and loads the corresponding model in that path. So the auto model, again, does our job, at least here. You could have explicitly defined it. And here you write what the tasks are.

So the form is that you write the auto model. And then you write for the task you are trying to complete. It can be sequence classification. It can be a mathematical language model. It can be causal language modeling.

So it can be an automobile model for causal elements. It can be an auto model for a math scale. It can be an automobile model for sequence classification. So it's the auto model for the task you have, right? And you initialize the tokenizer as the auto tokenizer from the pre-trained model. So "from pre-trained" is a method defined in the AutoTokenizer class.

So dot from pre-trained models. And then you pass the path to the model. So the question is, where do you get the path to this model?



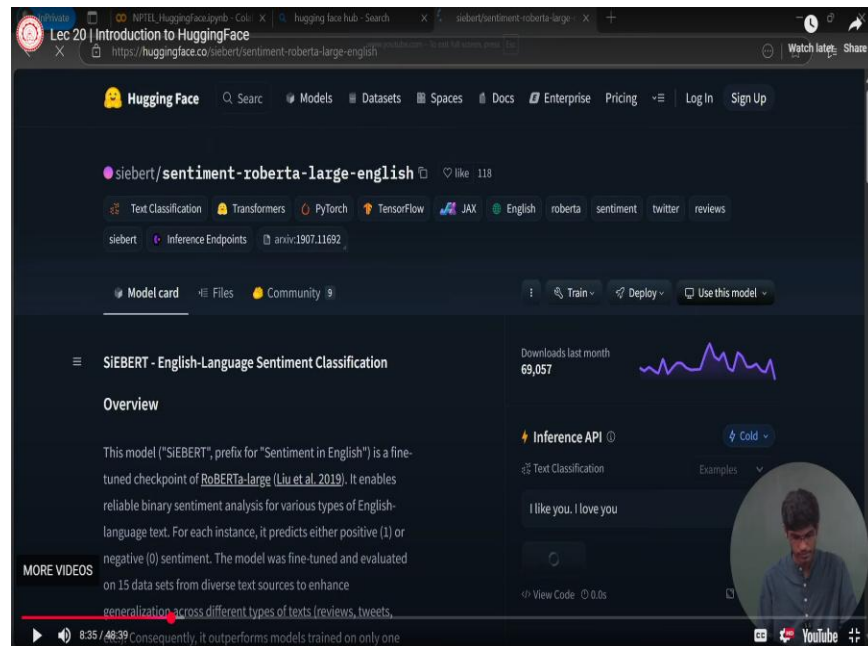


So normally, on the web, if you search for Hugging Face hubs, You will have this Hugging Face Hub. So there you can find a list of all the models. So, if you go to Models, you can see the task.

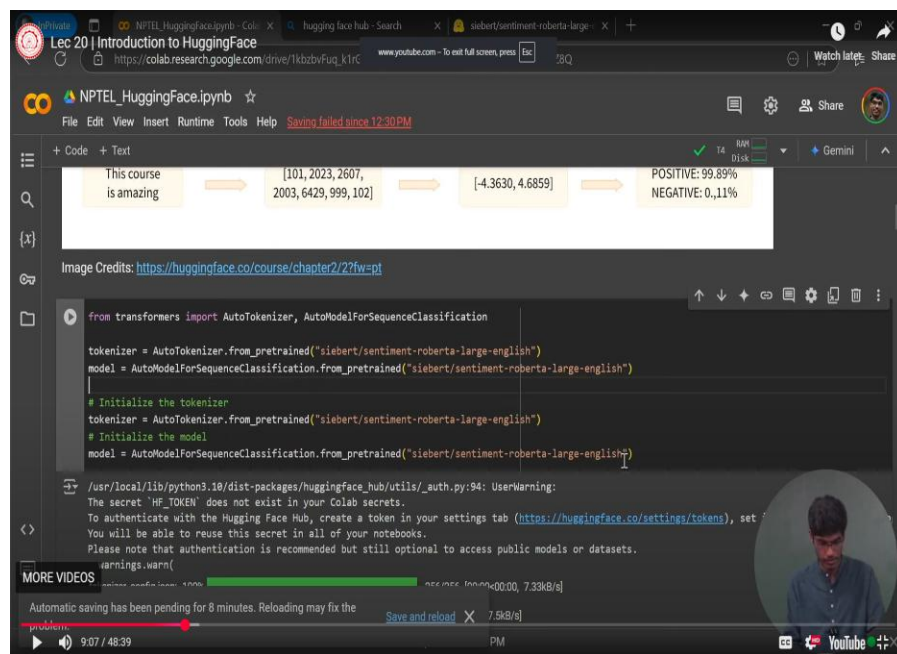
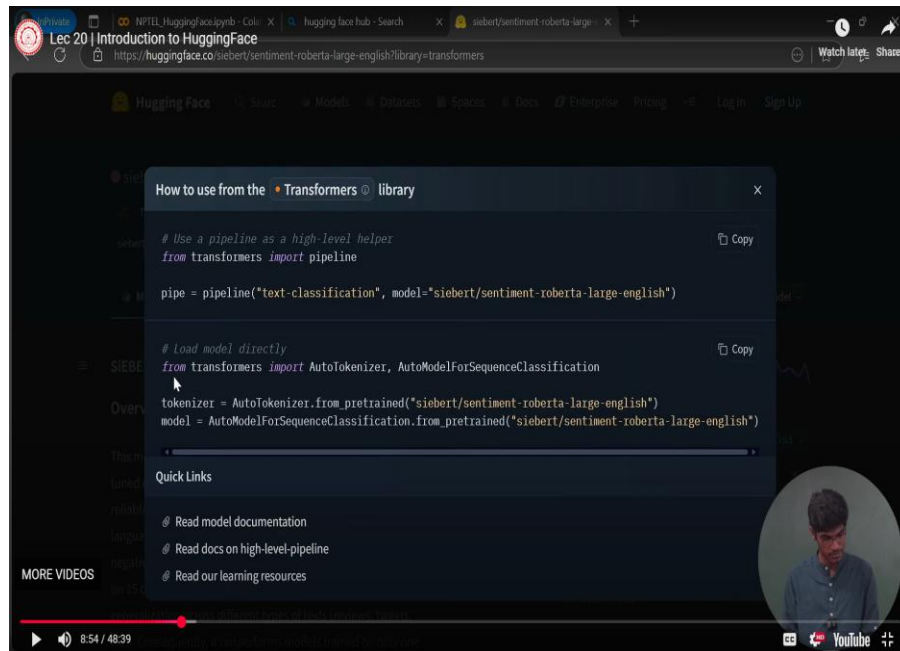
You can filter by task; you can see the datasets. You can see the models and other things like that. Suppose you do want to perform a natural language processing task of text

classification. Then, if you go to the models, you have all these models. And we will be using a model that is trained on Robata.

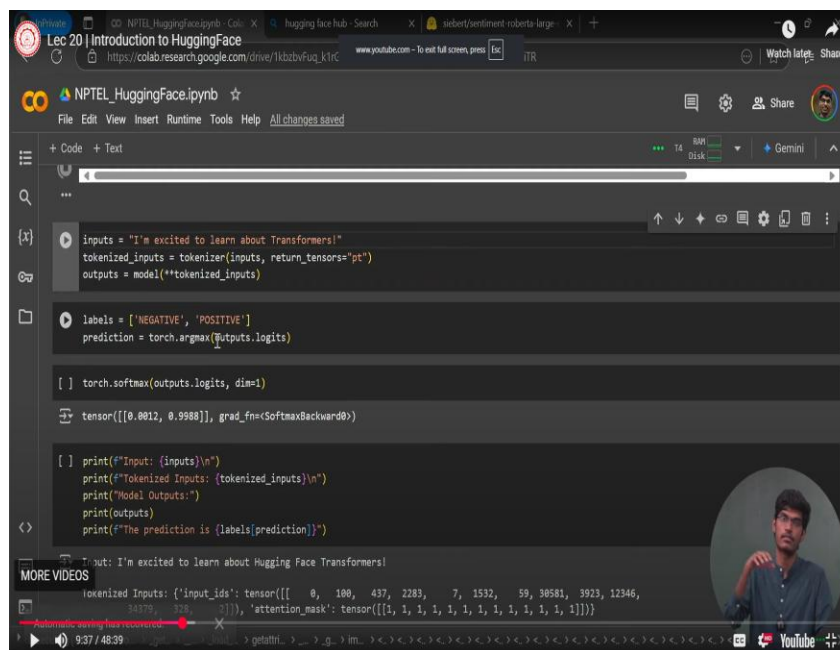
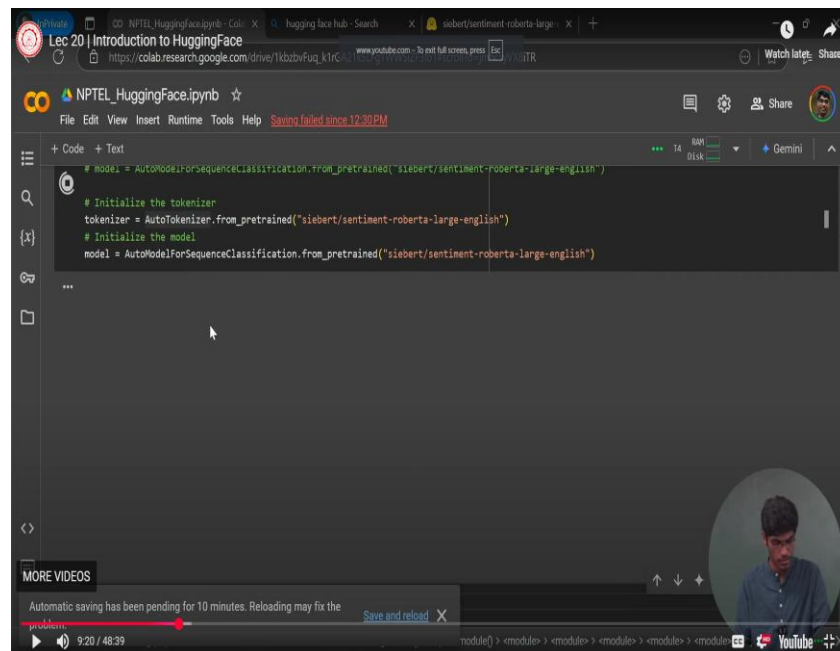
So you can search here.



You can search for Roberta's sentiment. Roberta speaks English fluently. So we'll be using this model. So this model has been trained on the sentiment analysis tasks. Roberta is basically the same architecture as BERT, and it is trained on English.



So, how can you use this model? Just go use this model feature in transformer libraries. And there you can see that, okay? So this is what you need to do: you're going to tokenize. So, this is what you need to do: it says hugging face. So just copy that and then paste it. This is what we have done here, right? So that's how you can search for a model on Hub and use it in your code.



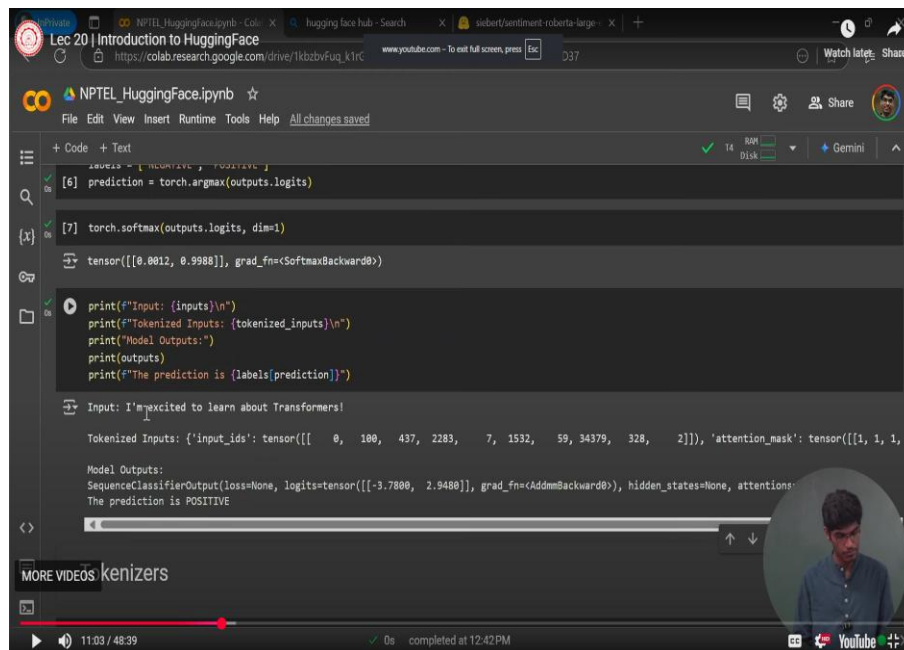
So running it will just load some binary files that correspond to the models. So, it takes some time. So let it load all the files, and let us proceed. So, how does it, as I said, what are the steps? Let us think step by step now, shall we? So, the first thing is to tokenize.

So suppose you have an input sentence. I am excited to learn about transformers. So what you do is pass the input to the tokenizer, right? And this returned tensor is equal to PT. What does it do? It says that, okay, return the output in the form of PyTorch tensors. If you

don't write it, it doesn't return as tensors; it returns as a normal list. And then output to get the outputs you pass to the model.

The unpacked list of tokenized input: you'll see this in more detail. And then the prediction is basically whatever logic you use. You take the maximum of that and map it to the label "negative" or "positive," right? So, what is this? OK, what do you think? No, the dot is not defined. Maybe I'm not wrong about this.

Now it should be fine. So dot was not important. So now it should be fine. So now, if we do a softmax on the outputs, you get this logic. So it says, "OK, I'm excited to learn about Transformers." If you look at its sentiment, there is a 0.0012% probability. That is negative with a 0.9988% probability that it's positive. So that's what is done.



```
Lec 20 | Introduction to HuggingFace
https://colab.research.google.com/drive/1kbzvFuq_k1rC

NPTEL_HuggingFace.ipynb
File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text
[6] prediction = torch.argmax(outputs.logits)
[7] torch.softmax(outputs.logits, dim=1)
tensor([[0.0012, 0.9988]], grad_fn=<SoftmaxBackward0>)

print(f"Input: {inputs}\n")
print(f"Tokenized Inputs: {tokenized_inputs}\n")
print("Model Outputs:")
print(outputs)
print(f"The prediction is {labels[prediction]}")

Input: I'm excited to learn about Transformers!

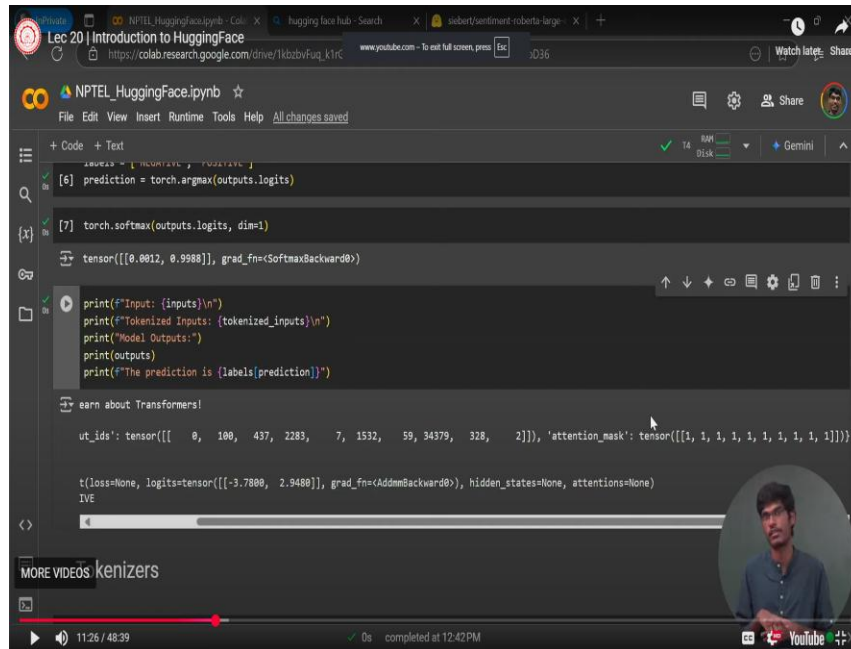
Tokenized Inputs: {'input_ids': tensor([[ 0, 100, 437, 2283, 7, 1532, 59, 34379, 328, 2]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])}

Model Outputs:
SequenceClassifierOutput(loss=None, logits=tensor([[ -3.7800, 2.9480]]), grad_fn=<AddmmBackward0>), hidden_states=None, attentions=None)

The prediction is POSITIVE
```

MORE VIDEOS: kenizers

11:03 / 48:39 0s completed at 12:42 PM YouTube



```
prediction = torch.argmax(outputs.logits)

torch.softmax(outputs.logits, dim=1)

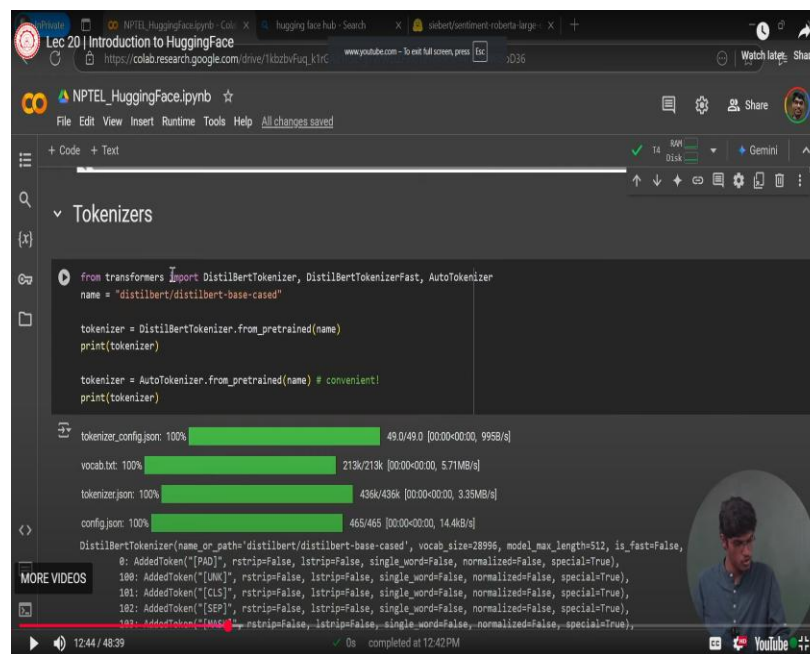
tensor([[0.0012, 0.9988]], grad_fn=<SoftmaxBackward0>)

print(f"Input: {inputs}\n")
print(f"Tokenized Inputs: {tokenized_inputs}\n")
print("Model Outputs:")
print(outputs)
print(f"The prediction is {labels[prediction]}")

learn about Transformers!

ut_ids': tensor([[ 0, 100, 437, 2283, 7, 1532, 59, 34379, 328, 2]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1]])

t(loss=None, logits=tensor([[ -3.7880, 2.9488]], grad_fn=<AddmmBackward0>), hidden_states=None, attentions=None)
IVE
```



```
from transformers import DistilBertTokenizer, DistilBertTokenizerFast, AutoTokenizer
name = "distilbert/distilbert-base-cased"

tokenizer = DistilBertTokenizer.from_pretrained(name)
print(tokenizer)

tokenizer = AutoTokenizer.from_pretrained(name) # convenient!
print(tokenizer)

tokenizer.config.json: 100% 49.0/49.0 [00:00<00:00, 995B/s]
vocab.txt: 100% 213k/213k [00:00<00:00, 5.71MB/s]
tokenizer.json: 100% 436k/436k [00:00<00:00, 3.35MB/s]
config.json: 100% 465/465 [00:00<00:00, 14.4kB/s]

DistilBertTokenizer(name_or_path='distilbert/distilbert-base-cased', vocab_size=28995, model_max_length=512, is_fast=False,
0: AddedToken("[PAD]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
180: AddedToken("[UNK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
181: AddedToken("[CLS]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
182: AddedToken("[SEP]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
183: AddedToken("[MASK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
```

So let us look at it step by step. So this is the input. The input was, "I am excited to learn about transformers." So, when I pass it to the tokenizer, the tokenized output is like that. 0, 100, 437.

So, this is a list of token numbers, as I said. And the attention mask, if you remember, We discussed in our "Transformers from Scratch" tutorial that the attention mask, There are two kinds of things that it handles. So when it's on the encoder side, it handles whether you

have a padding token or not. If you have a padding token, the attention mask will be zero in that position. If you have a future, such as for causal language modeling, When you are using the decoder on the decoder side, the attention mask will inform you whether to mask the future tokens or not right. So for the future tokens, all future tokens for a current token will be zeroed out. So when you apply softmax, it is subtracted first within the implementation. We mix zero to negative infinity in those attention scores. And then applying softmax makes it zero attention in the attention matrix, basically, right? So this is the tokenized outputs. And then our model outputs are produced when you pass them to the model.

So, what is the output? The logit over whatever distribution of apps. So, this is a model for sequence classification. So we already have a two-dimensional output linear layer. So it outputs a two-dimensional vector of two-dimensional logits, right? So it outputs a tensor that is two-dimensional because it's a sentiment robot, right? And then the prediction is the arc max of that; the logic is right. So now we will basically divide this lecture or tutorial into two or three parts.

In the first part, we will look at tokenizers, right?



```
from transformers import DistilBertTokenizer, DistilBertTokenizerFast, AutoTokenizer
name = "distilbert/distilbert-base-cased"

tokenizer = DistilBertTokenizer.from_pretrained(name)
print(tokenizer)

tokenizer = AutoTokenizer.from_pretrained(name) # convenient!
print(tokenizer)
```

tokenizer_config.json: 100% 49.0/49.0 [00:00<00:00, 995B/s]

vocab.txt: 100% 213k/213k [00:00<00:00, 5.71MB/s]

tokenizer.json: 100% 436k/436k [00:00<00:00, 3.35MB/s]

config.json: 100% 465/465 [00:00<00:00, 14.4kB/s]

```
DistilBertTokenizer(name_or_path='distilbert/distilbert-base-cased', vocab_size=28996, model_max_length=512, is_fast=False, padding_side='right', truncation_side='left',
 0: AddedToken("[PAD]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
100: AddedToken("[UNK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
101: AddedToken("[CLS]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
102: AddedToken("[SEP]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
103: AddedToken("[MASK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
)

DistilBertTokenizerFast(name_or_path='distilbert/distilbert-base-cased', vocab_size=28996, model_max_length=512, is_fast=True, padding_side='right', truncation_side='left',
 0: AddedToken("[PAD]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
100: AddedToken("[UNK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
101: AddedToken("[CLS]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
102: AddedToken("[SEP]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
103: AddedToken("[MASK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
)
```

```
tokenizer = DistilBertTokenizerFast(name_or_path='distilbert/distilbert-base-cased', vocab_size=28996, model_max_length=512, is_fast=True, padding_side='right', truncation_side='left',
 0: AddedToken("[PAD]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
100: AddedToken("[UNK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
101: AddedToken("[CLS]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
102: AddedToken("[SEP]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
103: AddedToken("[MASK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
)

# Call the tokenizer
input_str = "NPTEL is great!"
tokenized_inputs = tokenizer(input_str)

print(tokenized_inputs)
```

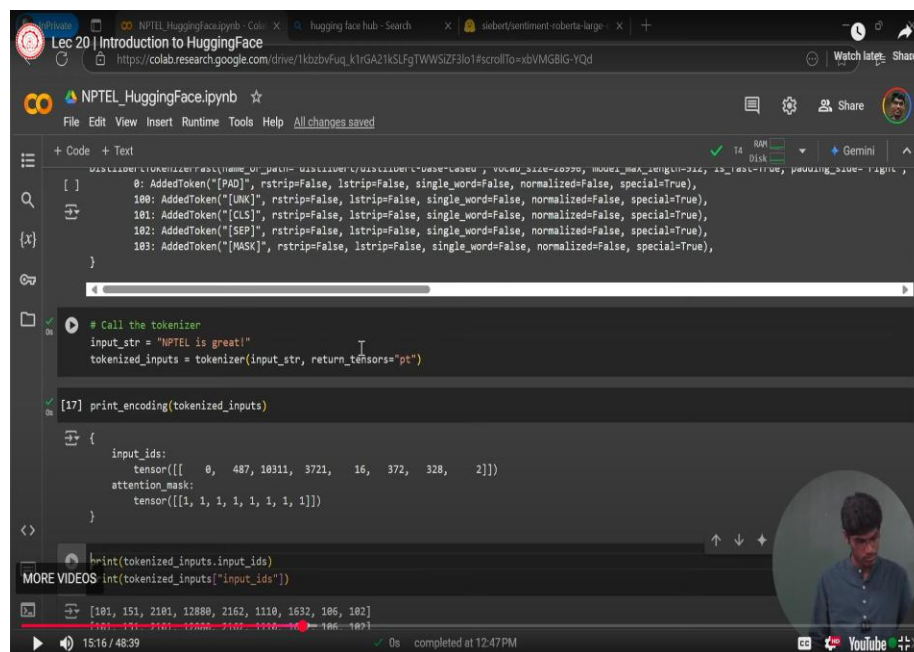
So, tokenizer, like I said, you can either import AutoTokenizer and play with it. Or you can explicitly say, "Okay, this model I know uses the DistilBERT tokenizer." So here we are using the DistilBERT model. So, DistilBERT is basically the same architecture as BERT. It's trained with knowledge distillation from a larger BERT model, isn't it? So think of it as a simple BERT model.

So B-E-R-T BERT. So it's an encoder-only model, right? Masked language modeling pre-training objectives. So connect with whatever you have learned in the previous weeks. So this part is encoded only by the model. So you can either use the DistilBERT tokenizer, or you can just tell the tokens. I auto-tokenize and just pass it, right? So you can see that there are some tokens there.

Special tokens like pad token, unk token, and unk tokens handle unknown words in the vocabulary. So, if there is a token that is unknown, then that's how it's handled. We'll see the classifier. During the BERT lecture, you probably saw what the classification token does. The classification token is a token added in front of the input sentence.

So it captures the representation of the entire sentence. So we can use the representation of this classification token. Only to classify the sentence or perform any kind of classification tasks on the sentence. Separated tokens separate into sentences. So it's generally at the end of the sentence. And the mask is basically masking the forward tokens if you do a generation task.

Things like that.

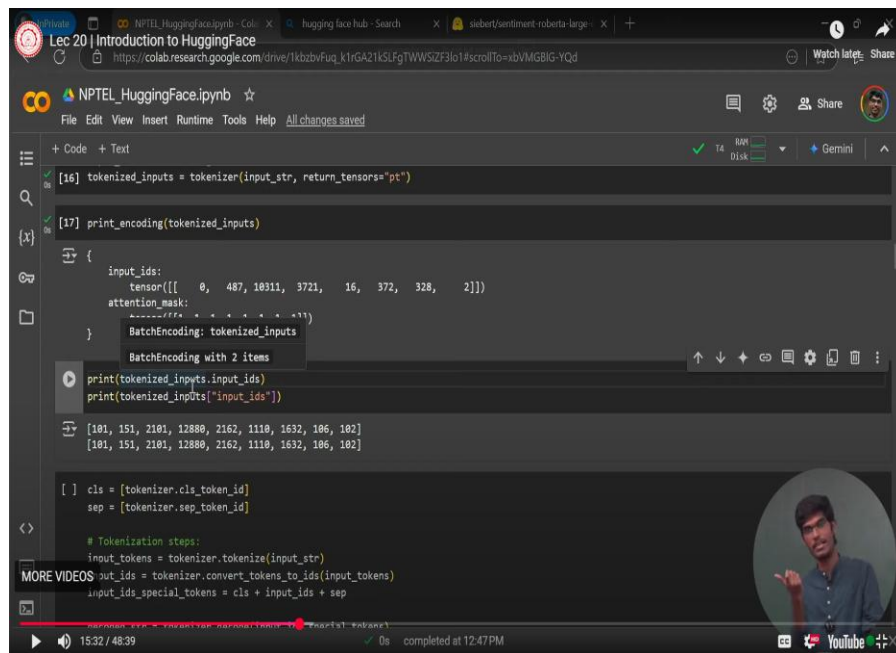


```
DistilBertTokenizerFast({'tokens': ['[PAD]', '[UNK]', '[CLS]', '[SEP]', '[MASK]'], 'vocab_size': 4096, 'model_max_length': 512, 'do_lower_case': True, 'padding_side': 'right', 'truncation_side': 'left'})
0: AddedToken("[PAD]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
100: AddedToken("[UNK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
101: AddedToken("[CLS]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
102: AddedToken("[SEP]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
103: AddedToken("[MASK]", rstrip=False, lstrip=False, single_word=False, normalized=False, special=True),
}

Call the tokenizer
input_str = "NPTEL is great!"
tokenized_inputs = tokenizer(input_str, return_tensors="pt")

[17] print_encoding(tokenized_inputs)
{
  input_ids:
    tensor([[ 0, 487, 10311, 3721, 16, 372, 328, 2]])
  attention_mask:
    tensor([[1, 1, 1, 1, 1, 1, 1, 1]])
}

b=int(tokenized_inputs.input_ids)
int(tokenized_inputs["input_ids"])
[101, 151, 2181, 12880, 2162, 1110, 1632, 186, 182]
100 101 102 103 104 105 106 107
15:16 / 48:39 completed at 12:47PM
```

```
[16] tokenized_inputs = tokenizer(input_str, return_tensors="pt")

[17] print(tokenized_inputs)

{
  input_ids:
    tensor([[ 0, 487, 18311, 3721, 16, 372, 328, 2]])
  attention_mask:
    tensor([[1, 1, 1, 1, 1, 1, 1, 1]])
}

BatchEncoding: tokenized_inputs
BatchEncoding with 2 items

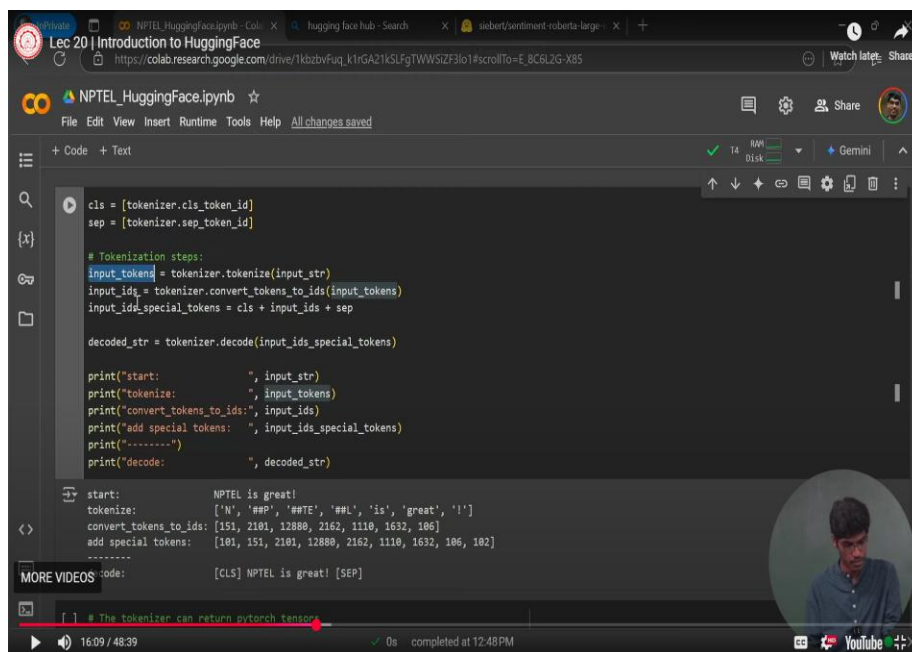
print(tokenized_inputs.input_ids)
print(tokenized_inputs.attention_mask)

[[101, 151, 2101, 12880, 2162, 1110, 1632, 106, 102]
 [101, 151, 2101, 12880, 2162, 1110, 1632, 106, 102]]

[ ] cls = [tokenizer.cls_token_id]
sep = [tokenizer.sep_token_id]

# Tokenization steps:
input_tokens = tokenizer.tokenize(input_str)
input_ids = tokenizer.convert_tokens_to_ids(input_tokens)
input_ids_special_tokens = cls + input_ids + sep

# The tokenizer can return pytorch tensors
```



```
cls = [tokenizer.cls_token_id]
sep = [tokenizer.sep_token_id]

# Tokenization steps:
input_tokens = tokenizer.tokenize(input_str)
input_ids = tokenizer.convert_tokens_to_ids(input_tokens)
input_ids_special_tokens = cls + input_ids + sep

decoded_str = tokenizer.decode(input_ids_special_tokens)

print("start: ", input_str)
print("tokenize: ", input_tokens)
print("convert_tokens_to_ids: ", input_ids)
print("add special tokens: ", input_ids_special_tokens)
print("-----")
print("decode: ", decoded_str)

start: NPTEL is great!
tokenize: ['N', '##P', '##TE', '##L', 'is', 'great', '!']
convert_tokens_to_ids: [151, 2101, 12880, 2162, 1110, 1632, 106]
add special tokens: [101, 151, 2101, 12880, 2162, 1110, 1632, 106, 102]
-----
decode: [CLS] NPTEL is great! [SEP]
```

So now suppose we have the input string, and PTL is great, right? So I pass the input token and the raw input sentences to the tokenizer. So during tokenization, if we say the return tensor is equal to pt, What it does is look; now it makes sense if we remove it. Right, if we remove this. It was just a list.

And if we add return tensors, what it does is convert them. into PyTorch tensors. So, that's the difference. So the attention mask tells you, okay, you need to attend to all tokens. There

are no special tokens. So whether you use tokenized inputs or input IDs, Or these tokenized inputs are basically a dictionary, aren't they? So tokenized, when you pass the input string to the tokenizer, It runs a dictionary.

One key of the dictionary is the input IDs, and another key is the attention mask. So you can access the input IDs just like you access any value in the dictionary. And then you can define what the classification token ID is. The classification token ID is used for the classification token, as I mentioned.

It is used to perform any kind of classification task. So, the classification token contains this whole sentence representation. And a separate token separates two sentences, right? So first, the tokenizer does is tokenizes this into a list of numbers. Token numbers. And then we have the input IDs, right? So we are seeing what the step is.

And then what the tokenizer does is that you have NPTEL, which is great. It first tokenizes it into "NPTEL is great"; that kind of tokenizing is breaking. Then what it does is add, in the case of the BERT tokenizer, What it does is add a CLS token to the front. And a separate token at the end, right? So we can see this.

So this is a step.

```
start:      NPTEL is great!
tokenize:   ['N', 'T', 'EL', 'is', 'great', '!']
convert_tokens_to_ids: [487, 10311, 3721, 16, 372, 328]
add special tokens:   [0, 487, 10311, 3721, 16, 372, 328, 2]
-----
decode:     <s>NPTEL is great!</s>
```

Lec 20 | Introduction to HuggingFace

NPTEL_HuggingFace.ipynb

```
input_tokens = tokenizer.tokenize(input_str)
input_ids = tokenizer.convert_tokens_to_ids(input_tokens)
input_ids_special_tokens = cls + input_ids + sep

decoded_str = tokenizer.decode(input_ids_special_tokens)

print("start:      ", input_str)
print("tokenize:     ", input_tokens)
print("convert_tokens_to_ids:", input_ids)
print("add special tokens: ", input_ids_special_tokens)
print("-----")
print("decode:        ", decoded_str)
```

start: NPTEL is great!

tokenize: ['N', 'PT', 'EL', 'dis', 'great', '!']

convert_tokens_to_ids: [487, 10311, 3721, 16, 372, 328]

add special tokens: [0, 487, 10311, 3721, 16, 372, 328, 2]

decode: <s>NPTEL is great!</s>

tokenizer.decode([10311])

'N'

The tokenizer can return pytorch tensors

model_inputs = tokenizer("Hugging Face Transformers is great!", return_tensors="pt")

17:28 / 48:39

Lec 20 | Introduction to HuggingFace

NPTEL_HuggingFace.ipynb

```
input_tokens = tokenizer.tokenize(input_str)
input_ids = tokenizer.convert_tokens_to_ids(input_tokens)
input_ids_special_tokens = cls + input_ids + sep

decoded_str = tokenizer.decode(input_ids_special_tokens)

print("start:      ", input_str)
print("tokenize:     ", input_tokens)
print("convert_tokens_to_ids:", input_ids)
print("add special tokens: ", input_ids_special_tokens)
print("-----")
print("decode:        ", decoded_str)
```

start: NPTEL is great!

tokenize: ['N', 'PT', 'EL', 'dis', 'great', '!']

convert_tokens_to_ids: [487, 10311, 3721, 16, 372, 328]

add special tokens: [0, 487, 10311, 3721, 16, 372, 328, 2]

decode: <s>NPTEL is great!</s>

tokenizer.decode([328])

!'

The tokenizer can return pytorch tensors

model_inputs = tokenizer("Hugging Face Transformers is great!", return_tensors="pt")

17:39 / 48:39

NPTEL_HuggingFace.ipynb

```
[19] input_tokens = tokenizer.tokenize(input_str)
input_ids = tokenizer.convert_tokens_to_ids(input_tokens)
input_ids_special_tokens = cls + input_ids + sep

decoded_str = tokenizer.decode(input_ids_special_tokens)

print("start:      ", input_str)
print("tokenize:     ", input_tokens)
print("convert_tokens_to_ids:", input_ids)
print("add special tokens:  ", input_ids_special_tokens)
print("-----")
print("decode:        ", decoded_str)
```

start: NPTEL is great!
tokenize: ['N', 'PT', 'EL', 'dis', 'great', '!']
convert_tokens_to_ids: [487, 18311, 3721, 16, 372, 328]
add special tokens: [0, 487, 18311, 3721, 16, 372, 328, 2]

decode: <s>NPTEL is great!</s>

len(tokenizer.get_vocab())

58265

The tokenizer can return pytorch tensors

```
model_inputs = tokenizer("Hugging Face Transformers is great!", return_tensors="pt")
```

completed at 12:50 PM

Lec 20 | Introduction to HuggingFace

NPTEL_HuggingFace.ipynb

```
print("start:      ", input_str)
print("tokenize:     ", input_tokens)
print("convert_tokens_to_ids:", input_ids)
print("add special tokens:  ", input_ids_special_tokens)
print("-----")
print("decode:        ", decoded_str)
```

start: NPTEL is great!
tokenize: ['N', 'PT', 'EL', 'dis', 'great', '!']
convert_tokens_to_ids: [487, 18311, 3721, 16, 372, 328]
add special tokens: [0, 487, 18311, 3721, 16, 372, 328, 2]

decode: <s>NPTEL is great!</s>

[23] tokenizer.decode([0])

'<s>'

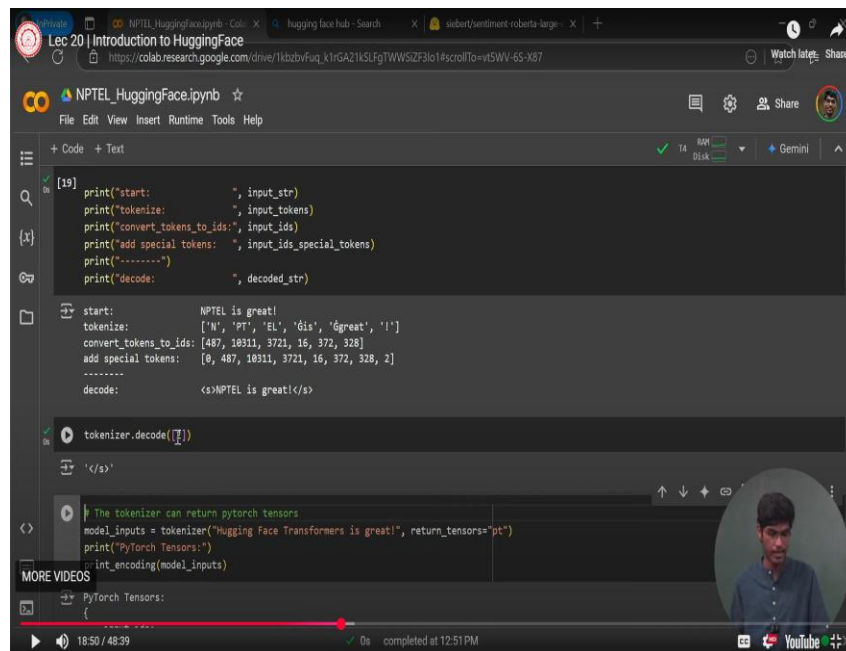
The tokenizer can return pytorch tensors

```
model_inputs = tokenizer("Hugging Face Transformers is great!", return_tensors="pt")
print("PyTorch Tensors:")
int_encoding(model_inputs)
```

PyTorch Tensors:

18:38 / 48:39

completed at 12:51 PM



```
[19] print("start:         ", input_str)
      print("tokenize:      ", input_tokens)
      print("convert tokens to ids:", input_ids)
      print("add special tokens: ", input_ids_special_tokens)
      print("-----")
      print("decode:         ", decoded_str)

start:         NPTEL is great!
tokenize:      ['N', 'P', 'T', 'E', 'L', 'Ġis', 'Ġgreat', '!']
convert tokens to ids: [487, 18311, 3721, 16, 372, 328]
add special tokens:  [0, 487, 18311, 3721, 16, 372, 328, 2]
-----
decode:         <S>NPTEL is great!</S>
```

```
tokenizer.decode([1])

'</S>'
```

```
The tokenizer can return pytorch tensors
model_inputs = tokenizer("Hugging Face Transformers is great!", return_tensors="pt")
print("PyTorch Tensors:")
int_encoding(model_inputs)
```

MORE VIDEOS

PyTorch Tensors:

18:50 / 48:39

So the first, this is a start sentence, NPTEL is great. When you tokenize it, it tokenizes into N-P-T-E-L, which is great, isn't it? So these are some special tokens that are handled. And then it is converted to these tokens that are going to be identified. So, N is 487. So you can easily check, can you? You can easily check that tokenizer.decode, and you can pass the input ID, right? You can pass on the input; I'd say 487, right? This is a 487.

So it should give n, right? 487 corresponds to n. Similarly, if you pass on, say, 328, right? So what should 328 give? It is the exclamation mark! So you can just see it; look, it's the exclamation mark! So just like that, in fact, you can check what the size is of the vocabulary of a tokenizer right. You can just get the vocabulary and take the length of the program. Then you can see, okay, there are 50,265 tokens in the DistilBERT tokenizer, right? So the DistilBERT tokenizer can handle 50,265 tokens. So that's how you can see what the number of tokens is in a particular tokenizer.

How many known tokens are there? So, these are the steps of the tokenizer. So once you have converted the tokens to IDs, What the tokenizer does is add a zero. This zero corresponds to, again, you can see that over here, as I said, tokenizer.decode. What is a zero? Zero is basically our classification token.

So the classification token or the start of centers; you can also see it.

At the start of centers. This acts as our classification token. And this... 2 is our separated token. So that's the kind of thing that ends sentences. So that these two are added to the beginning of the classifier tokens, towards the end, that is the separated token. And that is the final tokenized sentence.

So if we have already discussed that, you should return tensor 0 to pt. The input IDs and the attention mask, instead of a list, are returned as a Torch tensor is correct.

Lec 20 | Introduction to HuggingFace

https://colab.research.google.com/drive/1kbzbvFuq_k1rC

NPTEL_HuggingFace.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
[ ] tensor([[ 101, 20164, 10932, 10289, 25267, 1110, 1632, 106, 102]])
attention_mask:
tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1]])
```

You can pass multiple strings into the tokenizer and pad them as you need

```
model_inputs = tokenizer(["NPTEL is great!",
                        "Learning Transformers is fun.",
                        "We will learn about RLHF next week."],
                        return_tensors="pt",
                        padding=True,
                        truncation=True)

print(f"Pad token: {tokenizer.pad_token} | Pad token id: {tokenizer.pad_token_id}")
print("Padding:")
print_encoding(model_inputs)
```

Pad token: [PAD] | Pad token id: 0

Padding:

```
{
  input_ids:
    tensor([[ 101, 151, 2101, 12880, 2162, 1110, 1632, 106, 102, 0,
              0, 0],
            [ 101, 9681, 25267, 1110, 4106, 119, 102, 0, 0, 0, 0, 0],
            [ 101, 1284, 1209, 3858, 1164, 155, 2162, 13561, 1397, 1989,
              119, 102]])
  attention_mask:
    tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
            [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
}
```

MORE VIDEOS

19:21 / 48:39

0s completed at 12:51 PM

YouTube

Lec 20 | Introduction to HuggingFace

https://colab.research.google.com/drive/1kbzbvFuq_k1rGA21k5UgTWW5Zf3lo1#scrollTo=H3bA2peoD3

NPTEL_HuggingFace.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
# You can pass multiple strings into the tokenizer and pad them as you need
model_inputs = tokenizer(["NPTEL is great!",
                        "Learning Transformers is fun.",
                        "We will learn about RLHF next week."],
                        return_tensors="pt",
                        padding=True,
                        truncation=True)

print(f"Pad token: {tokenizer.pad_token} | Pad token id: {tokenizer.pad_token_id}")
print("Padding:")
print_encoding(model_inputs)
```

Pad token: [PAD] | Pad token id: 0

Padding:

```
{
  input_ids:
    tensor([[ 101, 151, 2101, 12880, 2162, 1110, 1632, 106, 102, 0,
              0, 0],
            [ 101, 9681, 25267, 1110, 4106, 119, 102, 0, 0, 0, 0, 0],
            [ 101, 1284, 1209, 3858, 1164, 155, 2162, 13561, 1397, 1989,
              119, 102]])
  attention_mask:
    tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
            [1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
}
```

MORE VIDEOS

Play (k)

20:37 / 48:39

0s completed at 12:51 PM

YouTube

The screenshot shows a Jupyter Notebook titled "NPTEL_HuggingFace.ipynb" with the following code and output:

```
Learning transformers is fun. ,
    "We will learn about RLHF next week.",
],
return_tensors="pt",
padding=True,
truncation=True)

print("Pad token: <pad> | Pad token id: (tokenizer.pad_token_id)")
print("Padding:")
print_encoding(model_inputs)
```

Output:

```
Pad token: <pad> | Pad token id: 1
Padding:
{
  input_ids:
    tensor([[ 0, 487, 10311, 3721, 16, 372, 328, 2, 1, 1],
            [ 0, 42489, 34379, 16, 1531, 4, 2, 1, 1, 1],
            [ 0, 170, 40, 1532, 59, 28483, 25894, 220, 186, 4,
              2]])
  attention_mask:
    tensor([[1, 1, 1, 1, 1, 1, 1, 1, 0, 0],
            [1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
}
```

The video player at the bottom shows a progress bar at 20:42 / 48:39.

The screenshot shows the same Jupyter Notebook with updated code and output:

```
print("Pad token: (tokenizer.pad_token) | Pad token id: (tokenizer.pad_token_id)")
print("Padding:")
print_encoding(model_inputs)

Pad token: <pad> | Pad token id: 1
Padding:
{
  input_ids:
    tensor([[ 0, 487, 10311, 3721, 16, 372, 328, 2, 1, 1],
            [ 0, 42489, 34379, 16, 1531, 4, 2, 1, 1, 1],
            [ 0, 170, 40, 1532, 59, 28483, 25894, 220, 186, 4,
              2]])
  attention_mask:
    tensor([[1, 1, 1, 1, 1, 1, 1, 1, 0, 0],
            [1, 1, 1, 1, 1, 1, 1, 0, 0, 0],
            [1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])
}

# You can also decode a whole batch at once:
print("Batch Decode:")
print(tokenizer.batch_decode(model_inputs.input_ids))
print()
print("Batch Decode: (no special characters)")
print(tokenizer.batch_decode(model_inputs.input_ids, skip_special_tokens=True))
```

The video player at the bottom shows a progress bar at 21:59 / 48:39.

In fact, you can pass a batch of inputs to the tokenizer. So instead of passing a single sentence, you can pass three sentences.

So that's how training works. There are batches of sentences passed together. So that is how we exploit the parallel processing power of GPUs. We pass inputs in batches.

So you can pass three sentences. NPTEL is great. Learning transformers is fun. We will learn about RLHF next week. So there are three sentences, and we set the padding equal to true. But what padding truly does is what we already discussed in the Transformers tutorial. On implementing it using Python.

So padding, what it does is if the number of the three sentences. Suppose, in this case, there are three sentences. So one sentence is shorter than the other. So the smaller sentence will be padded PadTokenIDs, which we will set here to 0, are defined. So the PadTokenID is 0, for example. So then it will be padded with 0s at the end to ensure That all sentences in the batch have the same length is incorrect. And what we do with truncation is define a maximum length. Any sentence after tokenization is longer than the maximum length.

The number of tokens is greater than the maximum length. It truncates to the maximum length. So that's what padding and tokenization do. So just see that the pad token ID is set to 0. And if we run this, we see the pad token is a pad. And the part token ID is set to one over here, okay? So here the pad token is set to one; it's not zero; it's one.

That's so fine; it can be anything: it can be zero; it can be one. It can be two, it can be three thousand, anything. But it is a tokenizer-dependent kind of thing, isn't it? So after padding, what happens is, you see, this was our sentence. NPTEL is great, isn't it? There is this initial CLS, these two separators, and this one is padded.

So, the longest sentence is the last sentence. We will learn about RLHF next week. So there is no padding here. There are no single ID tokens. And just look at the attention mask now! So, in the first sentence, there are three "ones" toward the end. And what the attention mask does is place zeros in place of the padded tokens.

So it tells the model, "Okay, these are the spatial tokens." Unnecessary. You don't need to pay attention to this while processing. So this reduces noise during training and unnecessary computations as well for using these tokens. So these are after tokenization. This is the tokenization of the batch. And you can batch decode. So you can just pass on whatever output the tokenizer gives as a matrix. Whether the rows correspond to each input example.

```
# You can also decode a whole batch at once:
print("Batch Decode:")
print(tokenizer.batch_decode(model_inputs.input_ids))
print()
print("Batch Decode: (no special characters)")
print(tokenizer.batch_decode(model_inputs.input_ids, skip_special_tokens=True))
```

Batch Decode:
['[CLS] NPTEL is great! [SEP] [PAD] [PAD] [PAD]', '[CLS] Learning Transformers is fun. [SEP] [PAD] [PAD] [PAD] [PAD] [PAD]', '[CLS] We will learn abo

Batch Decode: (no special characters)
['NPTEL is great!', 'Learning Transformers is fun.', 'We will learn about RLHF next week.']

Models

For specific tasks, we need "heads" that need to be trained if we're doing sequence classification, question answering, or some other task.

MORE VIDEOS: Face automatically sets up the architecture we need when we specify the model class. For example, we are doing sentiment analysis so we are going to use `DistilBertForSequenceClassification`. If we were going to continue training DistilBERT on its masked-language modelling training objective, we would use `DistilBertForMaskedLM` and if we just wanted the model's representations, maybe for our own

And when you batch-decode, this is what you get. So CLS. So, if you batch decode, okay, this is what you get.

This is the start of the sentence. Okay, this is basically C.L.S. NPTEL is great. End of the sentence; basically, a separator. And then three pads. Three pad tokens, right? Similarly, learning transformations is fun.

Is there a CLS? There is a separator. There are four pad tokens. And similarly, there is no padding in the last sentence. Because it's the maximum sentence length. So when you do simple batch decoding, you have all of these special tokens shown in your output. But if you just pass on an argument, skip special tokens equal to true. During the decoding and then during output, it skips all those pad tokens. CLS tokens, but it skips and just gives the raw text, right? So this is tokenization. So this is the first part, right?

Lec 20 | Introduction to HuggingFace

NPTEL_HuggingFace.ipynb

```
[5] print("{}")
    for k, v in model_inputs.items():
        print(indent_str + k + ":")
        print(indent_str + indent_str + str(v))
    print("{}")
```

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%

23:19 / 48:39

Lec 20 | Introduction to HuggingFace

PART 02
MODELS

@HUGGINGFACE/TRANSFORMERS

23:22 / 48:39

Lec 20 | Introduction to HuggingFace

NPTEL_HuggingFace.ipynb

modeling training objective, we would use `DistilBertForMaskedLM`, and if we just wanted the model's representations, maybe for our own downstream task, we could just use `DistilBertModel`.

Here's a diagram of a model recreated from one found here: <https://huggingface.co/course/chapter2/2?fw=pt>.

Full model

23:52 / 48:29

completed at 12:54 PM

Lec 20 | Introduction to HuggingFace

NPTEL_HuggingFace.ipynb

Full model

Here are some examples.

```
*Model
*ForMaskedLM
*ForSequenceClassification
*ForTokenClassification
*ForQuestionAnswering
*ForMultipleChoice
...
```

where * can be `AutoModel` or a specific pretrained model (e.g. `DistilBert`)

There are three types of models:

- Encoders (e.g. BERT)
- Decoders (e.g. GPT2)
- Encoder-Decoder models (e.g. T5)

Note that not all models are compatible with all model architectures, for example `DistilBert` is not compatible with the `Seq2Seq` models because it only consists of an encoder.

25:04 / 48:39

completed at 12:54 PM

So if we go back to our image, we have already dealt now that How do you tokenize the raw text into input IDs? The second part is passing it to the model.

Now we'll tackle the models. So, as I said, it's generally automatic. What is the name of the module that handles it? It is an `AutoModel` model. What is the name of the class? You can say "`AutoModel`" for the task name.

So, the auto model for sequence classification. Auto model for causal elements. Like that. Or you can explicitly say that instead of the auto model, we are using DistilBERT. So you are saying, okay, it is DistilBERT for sequence classification. It's a DistilBERT for masters.

We can say it like that. So this diagram illustrates what the model architecture looks like. You have a model's inputs. You have the embedding layer here. And then you have all these layers.

So, in the case of an encoder-only model, there are n stacked encoder layers. And then you have the hidden states. which comes up to the layer, and there is a head. So, a head is basically a kind of feedforward layer or a linear layer. Feedforward network MLP or a single linear layer that just maps.

From the output vocabulary dimension to the end. Say you are doing classification on two classes. So it maps from, say, 1,024 to 2. Output the dimension to the number of classes. And then there's the final model output.

So that's how this whole model architecture looks. And, as I said, it is generally a star. Star can be an auto model. So, star for the task name, star for the master lab, star for sequence classification. In the case of a star, you can just keep the auto model. Or you can keep the explicit names of the models, like DistilBERT and GPT-2, right? Like T5 and other similar items. So, yeah.

Lec 20 | Introduction to HuggingFace

https://colab.research.google.com/drive/1kbzbvfuq_k1rC

NPTEL_HuggingFace.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

Encoder-Decoder models (e.g. TS)

Note that not all models are compatible with all model architectures, for example DistilBERT is not compatible with the Seq2Seq models because it only consists of an encoder.

```
from transformers import AutoModelForSequenceClassification, DistilBertForSequenceClassification, DistilBertModel
print('Loading base model')
base_model = AutoModelForSequenceClassification.from_pretrained('distilbert-base-cased')
print("(class) DistilBertForSequenceClassification checkpoint")
model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-cased', num_labels=2)
model = AutoModelForSequenceClassification.from_pretrained('distilbert-base-cased', num_labels=2)
```

Loading base model

config.json: 100% 465/465 [00:00<00:00, 14.9KB/s]

model.safetensors: 100% 263M/263M [00:01<00:00, 220MB/s]

Loading classification model from base model's checkpoint

Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-cased and are newly initialized from random values. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-cased and are newly initialized from random values. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

MORE VIDEOS

[] model_inputs = tokenizer(input_str, return_tensors="pt")

25:16 / 48:39

completed at 12:54 PM

Lec 20 | Introduction to HuggingFace

https://colab.research.google.com/drive/1kbzbvfuq_k1rGA21KSLFgTWW5ZF3lo1#scrollTo=ieABaShrB42z

NPTEL_HuggingFace.ipynb

File Edit View Insert Runtime Tools Help

+ Code + Text

```
[27] model = AutoModelForSequenceClassification.from_pretrained('distilbert-base-cased', num_labels=2)
```

Loading base model

config.json: 100% 465/465 [00:00<00:00, 27.9KB/s]

model.safetensors: 100% 263M/263M [00:03<00:00, 226MB/s]

Loading classification model from base model's checkpoint

Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-cased and are newly initialized from random values. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-cased and are newly initialized from random values. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

```
model_inputs = tokenizer(input_str, return_tensors="pt")
model_outputs = model(input_ids=model_inputs.input_ids, attention_mask=model_inputs.attention_mask)
[ ] model_outputs = model(**model_inputs)
print(model_inputs)
int()
int(model_outputs)
print()
print("Distribution over labels: ", torch.softmax(model_outputs.logits, dim=-1))
```

MORE VIDEOS

26:12 / 48:39

completed at 12:58 PM

So, how do you load the model? I already said you just need to define that class, didn't I?

You will create an instance of this class like an auto model for signal classification.

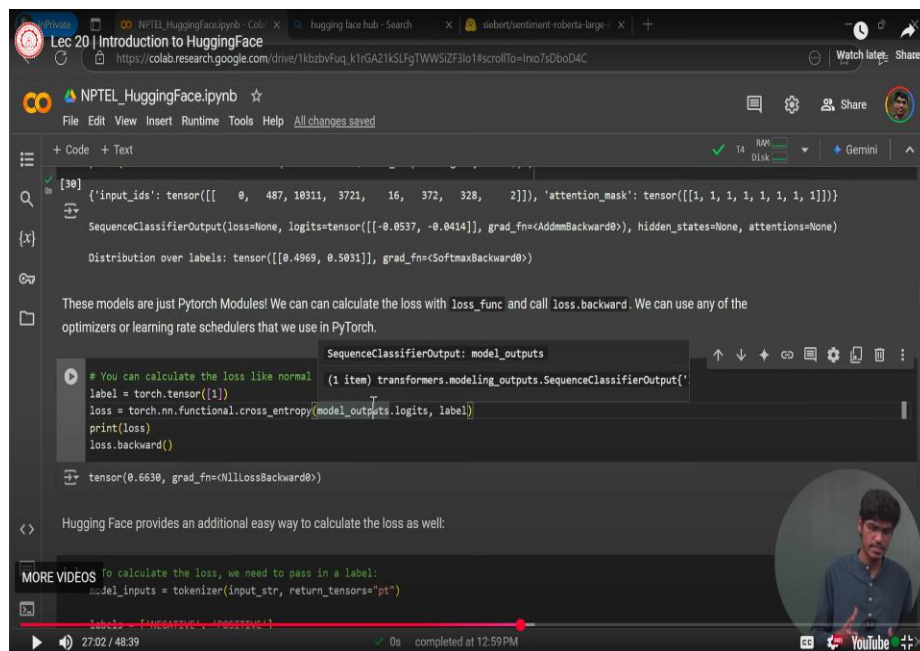
And then there is a method called dot from pre-trend, and you pass it along the path. How do you get to the path? If you have stored it in your local checkpoint, you have already

saved it. In your local environment, you just provide the local path. If you don't have it stored, you can get it from the Hugging Face Hub, right? And when you are doing sequence classification for these models, You also need to tell what the dimensions of the head are. How many levels are we expecting as output? So, this is how you load the model.

And then, what are the inputs to the model? The input is after tokenization. That is the input. What is the output of the model? You just passed. So, what should you pass to the model? You should pass the input IDs to the model that were obtained after tokenization. And the attention mask indicates which tokens to ignore during attention computation. You can either tell it explicitly that the input ID is able to model the inputs.

Or input the attention mass of this, or you can just do a. Simple star model input. So what the star model does is model the inputs in the dictionary. So, star star basically unpacks the dictionary, and the corresponding arguments are passed on. are mapped accordingly right.

So you can just print the model inputs and the model outputs. And just to check, this is the distribution of the labels. Okay.



```
[30]: {'input_ids': tensor([[ 0, 487, 10311, 3721, 16, 372, 328, 2]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1]])}
SequenceClassifierOutput(loss=None, logits=tensor([[ -0.0537, -0.0414]]), grad_fn=<AddmmBackward0>, hidden_states=None, attentions=None)
Distribution over labels: tensor([[0.4969, 0.5031]]), grad_fn=<SoftmaxBackward0>)

These models are just Pytorch Modules! We can calculate the loss with loss_func and call loss.backward. We can use any of the optimizers or learning rate schedulers that we use in PyTorch.

SequenceClassifierOutput: model_outputs
(1 item) transformers.modeling_outputs.SequenceClassifierOutput{'

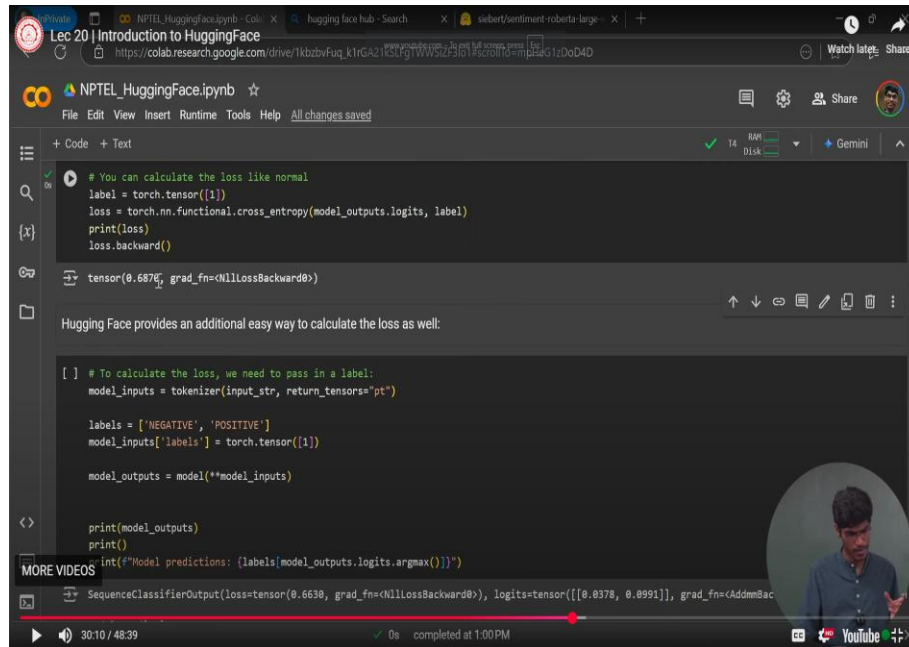
# You can calculate the loss like normal
label = torch.tensor([1])
loss = torch.nn.functional.cross_entropy(model_outputs.logits, label)
print(loss)
loss.backward()

tensor(0.6630, grad_fn=<NllLossBackward0>)

Hugging Face provides an additional easy way to calculate the loss as well:

MORE VIDEOS to calculate the loss, we need to pass in a label:
model_inputs = tokenizer(input_str, return_tensors="pt")
labels = torch.tensor([1])

27:02 / 48:39 completed at 12:59 PM
```

The screenshot shows a Jupyter Notebook interface with the title "Lec 20 | Introduction to HuggingFace". The code in the notebook is as follows:

```
# You can calculate the loss like normal
label = torch.tensor([1])
loss = torch.nn.functional.cross_entropy(model_outputs.logits, label)
print(loss)
loss.backward()

tensor(0.6676, grad_fn=<NllLossBackward0>)
```

Below the code, there is a text box that says "Hugging Face provides an additional easy way to calculate the loss as well:" followed by another code block:

```
[ ] # To calculate the loss, we need to pass in a label:
model_inputs = tokenizer(input_str, return_tensors="pt")

labels = ['NEGATIVE', 'POSITIVE']
model_inputs['labels'] = torch.tensor([1])

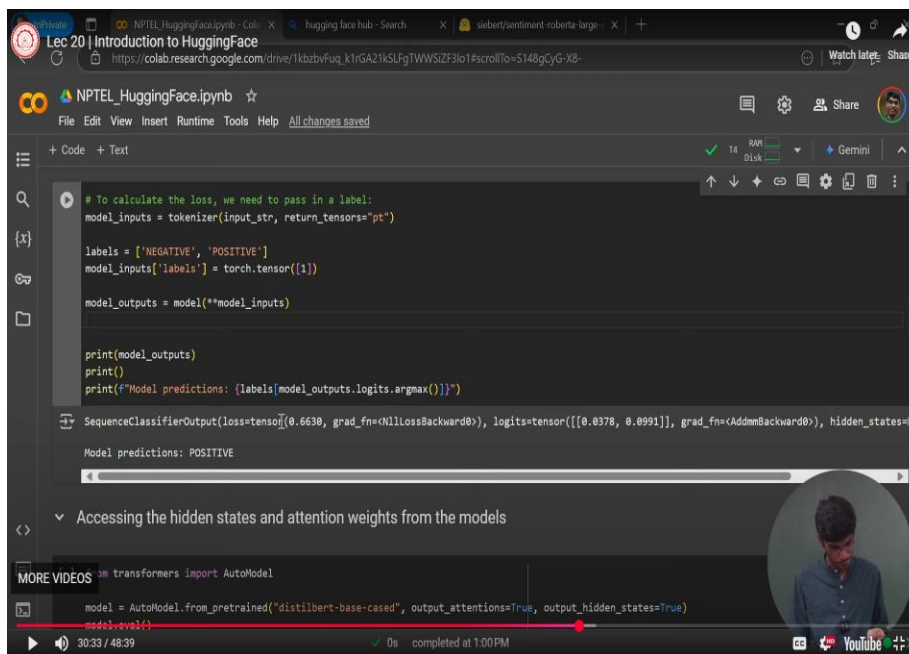
model_outputs = model(**model_inputs)

print(model_outputs)
print()
print(f"Model predictions: {labels[model_outputs.logits.argmax()]}")
```

The output of the second code block is:

```
SequenceClassifierOutput(loss=tensor(0.6638, grad_fn=<NllLossBackward0>), logits=tensor([[0.8378, 0.8991]], grad_fn=<AddmmBackward0>), hidden_states=)
```

The video player at the bottom shows a progress bar at 30:10 / 48:39 and a thumbnail of a person speaking.



The screenshot shows a Jupyter Notebook interface with the title "Lec 20 | Introduction to HuggingFace". The code in the notebook is as follows:

```
# To calculate the loss, we need to pass in a label:
model_inputs = tokenizer(input_str, return_tensors="pt")

labels = ['NEGATIVE', 'POSITIVE']
model_inputs['labels'] = torch.tensor([1])

model_outputs = model(**model_inputs)

print(model_outputs)
print()
print(f"Model predictions: {labels[model_outputs.logits.argmax()]}")
```

The output of the code is:

```
SequenceClassifierOutput(loss=tensor(0.6638, grad_fn=<NllLossBackward0>), logits=tensor([[0.8378, 0.8991]], grad_fn=<AddmmBackward0>), hidden_states=)
```

Below the output, there is a text box that says "Model predictions: POSITIVE".

Below the text box, there is a code block that says:

```
Accessing the hidden states and attention weights from the models
```

The video player at the bottom shows a progress bar at 30:33 / 48:39 and a thumbnail of a person speaking.

So this hugging phase transformer model works very well; this is basically in the back end. This is PyTorch, right? Everything is written in PyTorch. So this works very well with PyTorch modules.

These are basically just PyTorch modules renamed in some classes. So they have inherited the nn.Module class and written their own classes. Just like we did for transformers when we implemented them in Scratch.

In a previous tutorial. So Hugging Face has also done that. So this works very easily with PyTorch training loops. So we already know how to train a model using PyTorch. So you have an actual target level defined. You have a defined loss, right? So here, suppose we use cross-entropy loss. So, the cross-entropy loss is between the model output logits and the labels, right? And then you get the loss, and you do the backward pass.

So what does it do? So, how do you train a model? So, you have a model defined right in Python. What do you do if you can have a defined model? You define a loss function; okay, I want to do cross-entropy losses. Between the model outputs and logits, you define the optimizer, right? Optimizer setup and you pass it on. Specify the parameters: what is the learning rate of the optimizer's steps? If you want to have weight decay or not. And you pass the model parameters to the optimizer.

Then what does the training loop do? You first initiate the model, right? You take an input batch and pass it to the model. The model gives some output, doesn't it? You calculate the loss between the model output and the gold output. which you already have as levels.

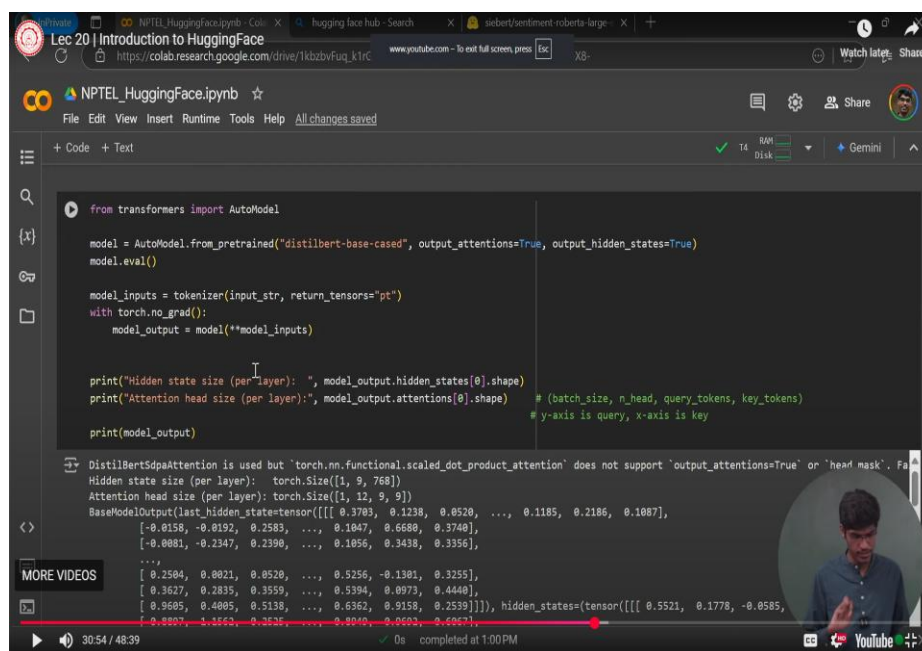
Then you do `optimizer.zero_grad`, so it ensures That the gradient in all tensors is set to zero. Then you do `optimizer.step`. So what does `optimizer.step()` do? It performs a single step of updating. So before the `optimizer.step()`, you need to calculate the gradient. So for that, you do `loss.backward`. So first you have the optimizer with zero gradients, then we do `loss.backward`, right? So loss, the optimizer of zero grad, sets the gradient of each tensor to zero.

Then, when you do `loss.backward`, what it does is take the loss. And perform a single backpropagation. So the backpropagation now populates the gradient field, right? So every tensor has this gradient field, right? It is defined as a gradient function, and a gradient is stored in each tensor. The tensor is basically an object. So this gradient for each tensor is stored after backpropagation.

And then, when you do `optimizer.step()`, it updates the trainable weights as Whatever the equation of the optimizer is. It can be simple gradient descent or stochastic gradient descent. So it can simply be the parameter minus η ; η is the learning rate into gradient of that parameter. It can be simple like that, or it can be more complicated in terms of

momentum. Weight decay and similar concepts. So this is how it works. So you can just do `loss.backward()` to perform a single step of backpropagation. And you can also print the loss using this cross-entropy.

In fact, when you model a single forward pass, The model-hugging phase automatically computes the loss. So if you pass this model input to the model, it outputs what you see. I already have this loss computed; you can see it, right? So it actually makes our job during the hugging phase much easier. By implementing a class for these PyTorch modules, which already provide the loss. as a part of the model output object right.



The screenshot shows a Jupyter Notebook titled "NPTEL_HuggingFace.ipynb" with a code cell containing the following Python code:

```
from transformers import AutoModel

model = AutoModel.from_pretrained("distilbert-base-cased", output_attentions=True, output_hidden_states=True)
model.eval()

model_inputs = tokenizer(input_str, return_tensors="pt")
with torch.no_grad():
    model_output = model(**model_inputs)

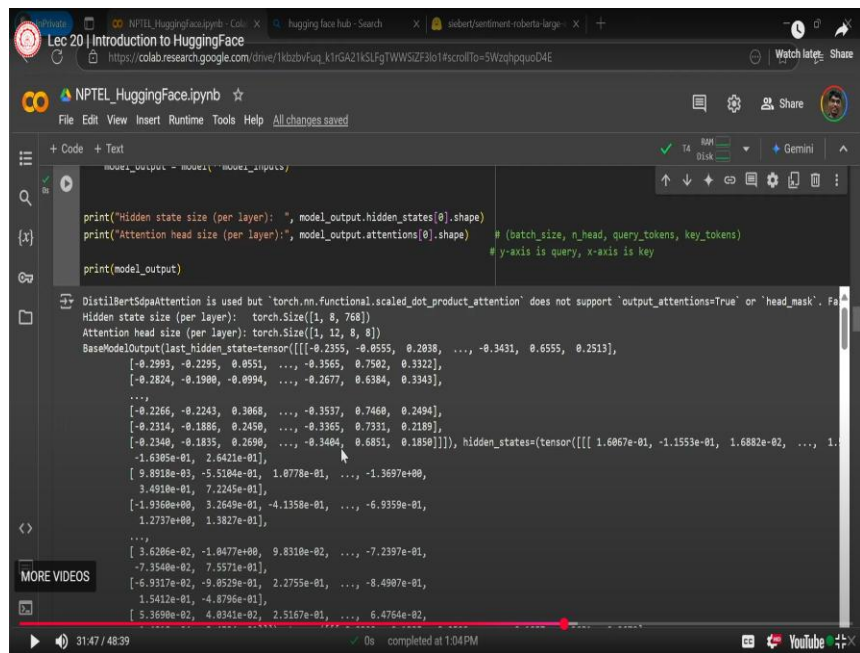
print("Hidden state size (per layer): ", model_output.hidden_states[0].shape)
print("Attention head size (per layer):", model_output.attentions[0].shape) # (batch_size, n_head, query_tokens, key_tokens)
# y-axis is query, x-axis is key

print(model_output)
```

The output of the code cell is as follows:

```
DistilBertSdpaAttention is used but 'torch.nn.functional.scaled_dot_product_attention' does not support 'output_attentions=True' or 'head_mask'. Falling back to the manual impl...
Hidden state size (per layer): torch.Size([1, 9, 768])
Attention head size (per layer): torch.Size([1, 12, 9, 9])
BaseModelOutput(last_hidden_state=tensor([[[[ 0.3783, 0.1238, 0.0520, ..., 0.1185, 0.2186, 0.1087],
        [-0.0158, -0.0192, 0.2583, ..., 0.1047, 0.6680, 0.3740],
        [-0.0081, -0.2347, 0.2390, ..., 0.1056, 0.3438, 0.3356],
        ...,
        [ 0.2504, 0.0021, 0.0520, ..., 0.5256, -0.1301, 0.3255],
        [ 0.3627, 0.2835, 0.3859, ..., 0.5394, 0.0973, 0.4640],
        [ 0.9605, 0.4805, 0.5138, ..., 0.6362, 0.9158, 0.2539]]]], hidden_states=(tensor([[[[ 0.5521, 0.1778, -0.0585,
        [-0.0007, -0.2662, -0.2038, ..., -0.0040, -0.0603, -0.6061],
        ...,
        [ 0.2504, 0.0021, 0.0520, ..., 0.5256, -0.1301, 0.3255],
        [ 0.3627, 0.2835, 0.3859, ..., 0.5394, 0.0973, 0.4640],
        [ 0.9605, 0.4805, 0.5138, ..., 0.6362, 0.9158, 0.2539]]]]], hidden_states=)
```

The video player interface at the bottom shows the video is 30:54 / 48:39 and was completed at 1:00 PM.

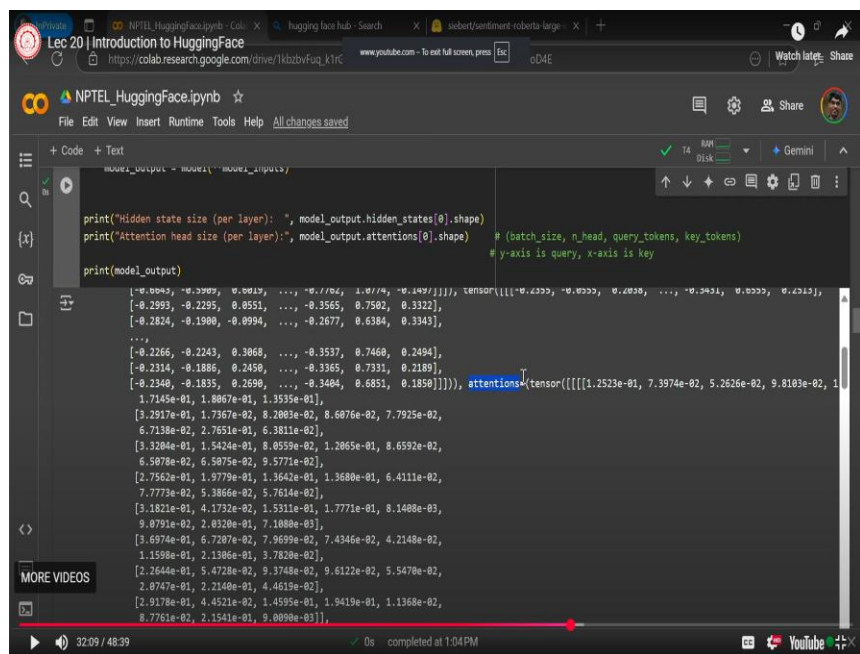


```
model_output = model(**model_inputs)

print("Hidden state size (per layer): ", model_output.hidden_states[0].shape)
print("Attention head size (per layer):", model_output.attentions[0].shape) # (batch_size, n_head, query_tokens, key_tokens)
# y-axis is query, x-axis is key

print(model_output)

DistilBertSdpaAttention is used but 'torch.nn.functional.scaled_dot_product_attention' does not support 'output_attentions=True' or 'head_mask'. Fa
Hidden state size (per layer): torch.Size([1, 8, 768])
Attention head size (per layer): torch.Size([1, 12, 8])
BaseModelOutput(last_hidden_state=tensor([[[[-0.2355, -0.8555, 0.2838, ..., -0.3431, 0.6555, 0.2513],
[-0.2993, -0.2295, 0.8551, ..., -0.3565, 0.7582, 0.3322],
[-0.2824, -0.1980, -0.8994, ..., -0.2677, 0.6384, 0.3343],
...,
[-0.2266, -0.2243, 0.3868, ..., -0.3537, 0.7468, 0.2494],
[-0.2314, -0.1886, 0.2458, ..., -0.3365, 0.7331, 0.2189],
[-0.2340, -0.1835, 0.2698, ..., -0.3484, 0.6851, 0.1850]]]], hidden_states=(tensor([[[[ 1.6867e-01, -1.1553e-01, 1.6882e-02, ..., 1.
-1.6385e-01, 2.6421e-01],
[ 9.8918e-03, -5.5104e-01, 1.0778e-01, ..., -1.3697e+00,
3.4918e-01, 7.2245e-01],
[-1.9368e+00, 3.2649e-01, -4.1358e-01, ..., -6.9359e-01,
1.2737e+00, 1.3827e-01],
...,
[ 3.6286e-02, -1.0477e+00, 9.8310e-02, ..., -7.2397e-01,
-7.3540e-02, 7.5571e-01],
[-6.9317e-02, -9.0529e-01, 2.2755e-01, ..., -8.4987e-01,
1.5412e-01, -4.8796e-01],
[ 5.3698e-02, 4.0341e-02, 2.5167e-01, ..., 6.4764e-02,
```



```
model_output = model(**model_inputs)

print("Hidden state size (per layer): ", model_output.hidden_states[0].shape)
print("Attention head size (per layer):", model_output.attentions[0].shape) # (batch_size, n_head, query_tokens, key_tokens)
# y-axis is query, x-axis is key

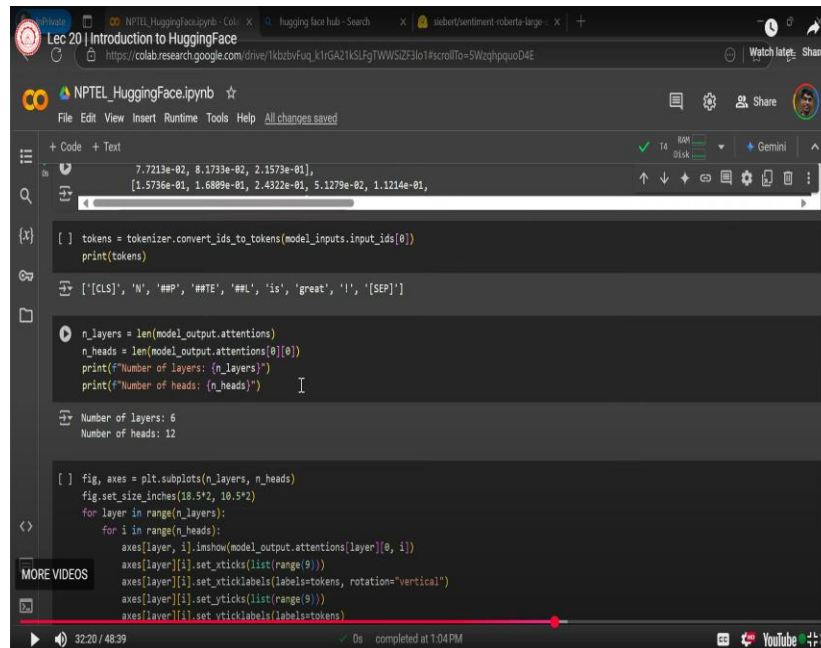
print(model_output)

[-0.0043, -0.2993, 0.8551, ..., -0.3565, 0.7582, 0.3322],
[-0.2993, -0.2295, 0.8551, ..., -0.3565, 0.7582, 0.3322],
[-0.2824, -0.1980, -0.8994, ..., -0.2677, 0.6384, 0.3343],
...,
[-0.2266, -0.2243, 0.3868, ..., -0.3537, 0.7468, 0.2494],
[-0.2314, -0.1886, 0.2458, ..., -0.3365, 0.7331, 0.2189],
[-0.2340, -0.1835, 0.2698, ..., -0.3484, 0.6851, 0.1850]]]], attentions=tensor([[[[ 1.2523e-01, 7.3974e-02, 5.2626e-02, 9.8183e-02, 1.
1.7145e-01, 1.8867e-01, 1.3535e-01],
[ 3.2917e-01, 1.7367e-02, 8.2083e-02, 8.6876e-02, 7.7925e-02,
6.7138e-02, 2.7651e-01, 6.3811e-02],
[ 3.3284e-01, 1.5424e-01, 8.8559e-02, 1.2065e-01, 8.6592e-02,
6.5078e-02, 6.5075e-02, 9.5771e-02],
[ 2.7562e-01, 1.9779e-01, 1.3642e-01, 1.3688e-01, 6.4111e-02,
7.7773e-02, 5.3866e-02, 5.7614e-02],
[ 3.1821e-01, 4.1732e-02, 1.5311e-01, 1.7771e-01, 8.1488e-03,
9.0791e-02, 2.8320e-01, 7.1888e-03],
[ 3.6974e-01, 6.7207e-02, 7.9699e-02, 7.4346e-02, 4.2148e-02,
1.1598e-01, 2.1386e-01, 3.7828e-02],
[ 2.2644e-01, 5.4728e-02, 9.3748e-02, 9.6122e-02, 5.5478e-02,
2.0747e-01, 2.2140e-01, 4.4619e-02],
[ 2.9178e-01, 4.4521e-02, 1.4595e-01, 1.9419e-01, 1.1368e-02,
8.7761e-02, 2.1541e-01, 9.8098e-03]]]]],
```

Now, one thing that is very good about this transformer module from Hugging Face is that you can access like every inner component of the model. So you have this model. So setting this model eval sets it into a non-training mode. So it sets the request grad equal to false, right? So now there is no gradient computation. So that's what `torch.no_grad` does, as well. So when you pass in the model, push the model, and you can access the hidden state. So if you print the model output, right? This is the model output; print it right.

What you see is that I can maybe show you a little bit. This is a big model output, and if I go on top of it, I am right. Let's look at this. The hidden states per layer are 1, 8, 7, 6, and 8. Attention, is head size right? And just look at this. It's the output of the base model. What do you have? You have all the hidden states that are computed for each layer.

And also, we have the attention computed for each head, right? So you already have that in the model output.



The screenshot shows a Jupyter Notebook titled "NPTEL_HuggingFace.ipynb" in a Google Colab environment. The code in the notebook is as follows:

```
7.7213e-02, 8.1733e-02, 2.1573e-01,
[1.5736e-01, 1.6889e-01, 2.4322e-01, 5.1279e-02, 1.1214e-01,

[] tokens = tokenizer.convert_ids_to_tokens(model_inputs.input_ids[0])
print(tokens)

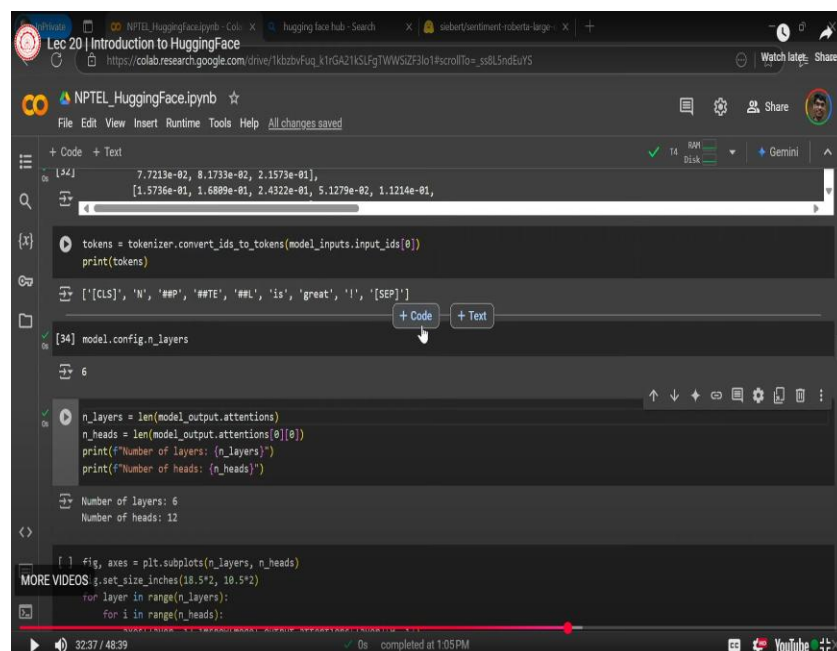
['[CLS]', 'N', '##P', '##TE', '##L', 'is', 'great', 'I', '[SEP]']

n_layers = len(model_output.attentions)
n_heads = len(model_output.attentions[0][0])
print(f"Number of layers: {n_layers}")
print(f"Number of heads: {n_heads}")

Number of layers: 6
Number of heads: 12

[] fig, axes = plt.subplots(n_layers, n_heads)
fig.set_size_inches(18.5*2, 10.5*2)
for layer in range(n_layers):
    for i in range(n_heads):
        axes[layer, i].imshow(model_output.attentions[layer][0, i])
        axes[layer, i].set_yticks(list(range(9)))
        axes[layer, i].set_xticklabels(labels=tokens, rotation="vertical")
        axes[layer, i].set_yticks(list(range(9)))
        axes[layer, i].set_xticklabels(labels=tokens)
```

The notebook interface includes a file explorer on the left, a toolbar with options like "Code" and "Text", and a status bar at the bottom indicating the execution time (32:20 / 48:39) and completion status (completed at 1:04 PM).



This screenshot shows the same Jupyter Notebook as the previous one, but with a cursor pointing to the output of the `model.config.n_layers` line. The code is identical to the previous screenshot:

```
7.7213e-02, 8.1733e-02, 2.1573e-01,
[1.5736e-01, 1.6889e-01, 2.4322e-01, 5.1279e-02, 1.1214e-01,

[] tokens = tokenizer.convert_ids_to_tokens(model_inputs.input_ids[0])
print(tokens)

['[CLS]', 'N', '##P', '##TE', '##L', 'is', 'great', 'I', '[SEP]']

[34] model.config.n_layers

6

n_layers = len(model_output.attentions)
n_heads = len(model_output.attentions[0][0])
print(f"Number of layers: {n_layers}")
print(f"Number of heads: {n_heads}")

Number of layers: 6
Number of heads: 12

[] fig, axes = plt.subplots(n_layers, n_heads)
fig.set_size_inches(18.5*2, 10.5*2)
for layer in range(n_layers):
    for i in range(n_heads):
```

The notebook interface is the same, but the status bar at the bottom indicates a slightly different execution time (32:27 / 48:39) and completion status (completed at 1:05 PM).

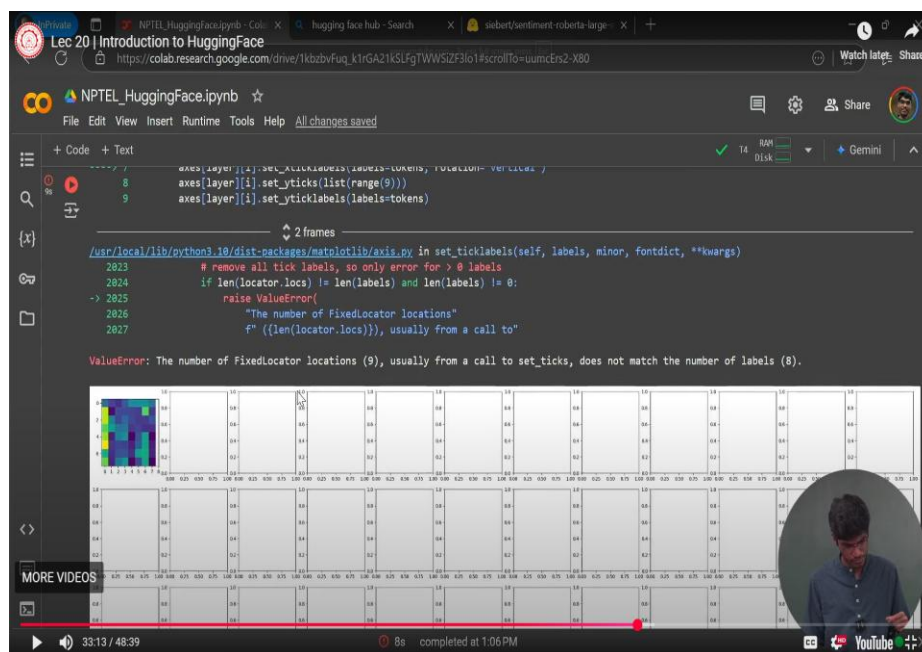
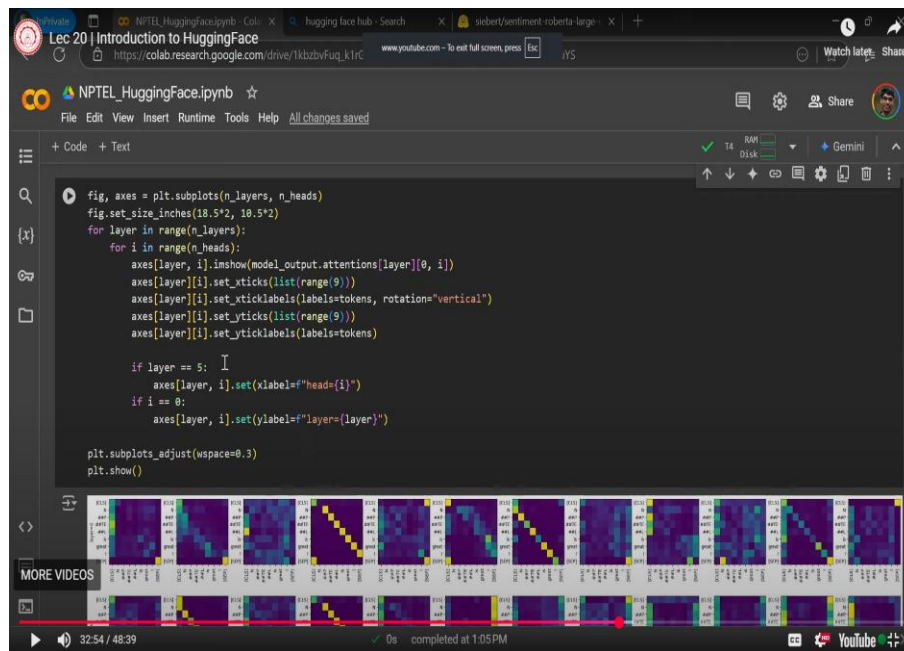
The screenshot shows a Google Colab notebook with the following content:

```
7.7213e-02, 8.1733e-02, 2.1579e-01],  
[1.5736e-01, 1.6809e-01, 2.4322e-01, 5.1279e-02, 1.1214e-01,  
[ ] tokens = tokenizer.convert_ids_to_tokens(model_inputs.input_ids[0])  
print(tokens)  
['[CLS]', 'N', '##P', '##TE', '##L', 'is', 'great', '!', '[SEP]']  
model.config.n_heads  
12  
n_layers = len(model_output.attentions)  
n_heads = len(model_output.attentions[0][0])  
print(f"Number of layers: {n_layers}")  
print(f"Number of heads: {n_heads}")  
Number of layers: 6  
Number of heads: 12  
fig, axes = plt.subplots(n_layers, n_heads)  
g.set_size_inches(18.5*2, 10.5*2)  
for layer in range(n_layers):  
    for i in range(n_heads):
```

The notebook interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with icons for code, text, and search, and a status bar at the bottom indicating the video is at 32:44 / 48:39 and completed at 1:05 PM.

Right, so these are the tokens you already said yes to so, in fact, you can see just the number of layers. And the number of heads in the model is. So an easier way to do that is to go to `model.config`. You can say that the DistilBERT model has six layers. So, six encoder blocks, right? And if you do N heads.

It says twelve. So each layer has 12 attention heads, and in fact, you can access. These heads of each layer print the attention matrices. Which is computed by each head when you perform a forward pass.



So maybe you can look at the code; it's very simple. We just take the Attention matrix, which is computed by each head.

And just plot it using a heat map by plotting it correctly. The number of this will give you the number of layers in the X. and the number of heads at the number of layers on the y-axis. The number of heads on the X-axis is fine.

Lec 20 | Introduction to HuggingFace

NPTEL_HuggingFace.ipynb

File Edit View Insert Runtime Tools Help All changes saved

Step-1: Loading the dataset

```
from datasets import load_dataset, DatasetDict

dataset_name = "takala/financial_phrasebank"

dataset = load_dataset(dataset_name, 'sentences_allagree')
```

README.md: 100% 8.88k/8.88k [00:00<00:00, 462kB/s]

financial_phrasebank.py: 100% 6.04k/6.04k [00:00<00:00, 291kB/s]

The repository for takala/financial_phrasebank contains custom code which must be executed to correctly load the dataset. You can inspect the repository in the future by passing the argument "trust_remote_code=True".

Do you wish to run the custom code? [y/N] y

FinancialPhraseBank-v1.0.zip: 100% 682k/682k [00:00<00:00, 24.5MB/s]

Generating train split: 100% 2264/2264 [00:00<00:00, 19526.44 examples/s]

MORE VIDEOS

dataset

33:22 / 48:39 8s completed at 1:06 PM

Watch later Share

Lec 20 | Introduction to HuggingFace

www.youtube.com - To exit full screen, press Esc

Watch later Share

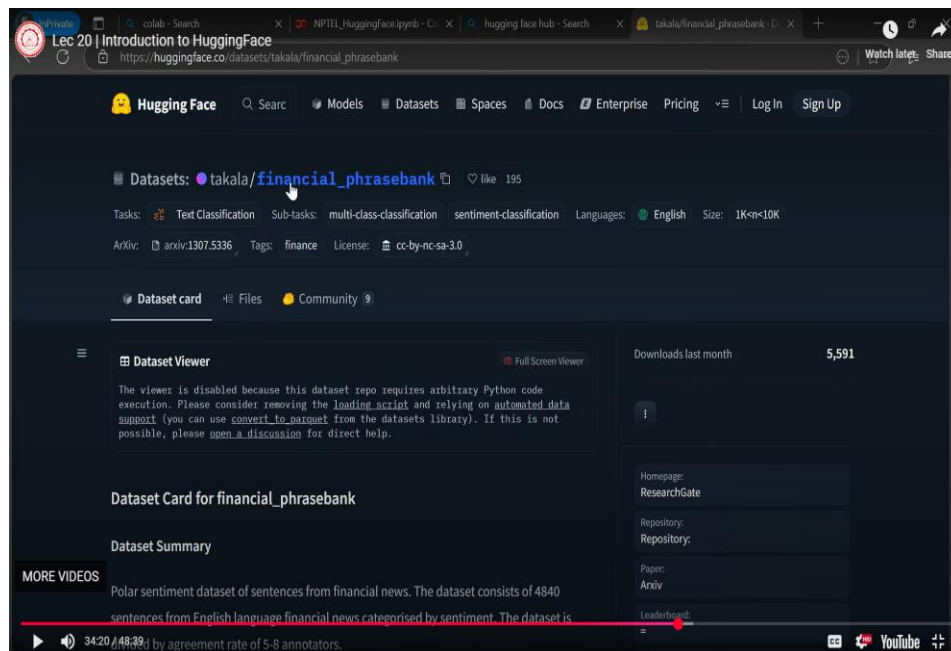
PART 03 FINETUNING

@HUGGINGFACE/TRANSFORMERS/TRAINING

MORE VIDEOS

33:27 / 48:39

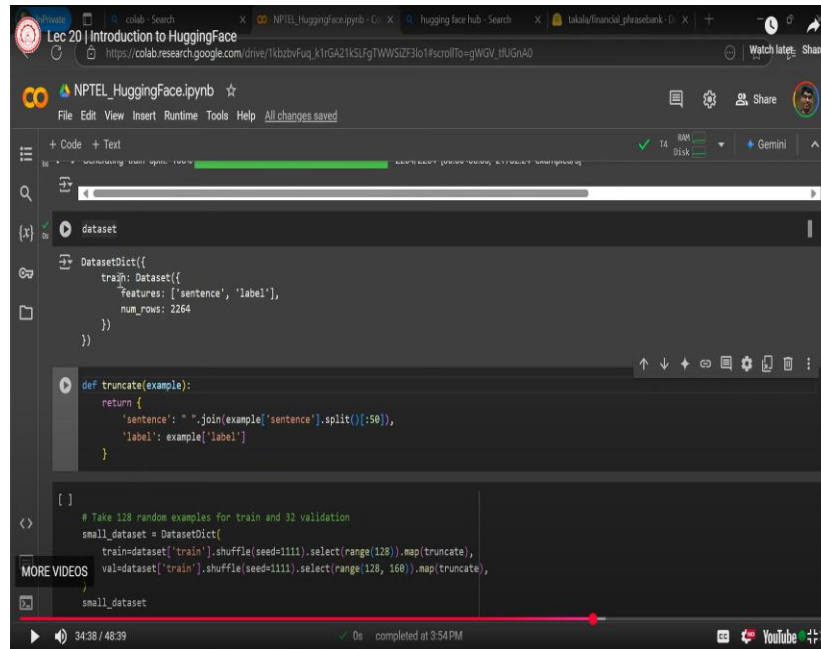
YouTube



So now, we will just look at a simple task of fine-tuning. Until now, we have seen how to load a model and how to use tokenizers.

And how do models process the inputs, and how can you access the internals? Like the attention and hidden states in a model. Now we'll see how you can fine-tune a model on a particular dataset. For this purpose, we are using a dataset called Financial Phrase Bank, right? So you can also find this dataset on the Hugging Face Hub, right? If you go to the Hugging Face Hub, you will find datasets. And just search for the financial phrase "Bank," right? Financial phrase: Bank.

So this data set that we are using, right? So we are using this dataset. So the way to load the dataset is just to use the `load_dataset` function.



```
dataset
DatasetDict({
  train: Dataset({
    features: ['sentence', 'label'],
    num_rows: 2264
  })
})

def truncate(example):
    return {
        'sentence': " ".join(example['sentence'].split()[0:50]),
        'label': example['label']
    }

[ ]
# Take 128 random examples for train and 32 validation
small_dataset = DatasetDict({
    train=dataset['train'].shuffle(seed=1111).select(range(128)).map(truncate),
    val=dataset['train'].shuffle(seed=1111).select(range(128, 160)).map(truncate),
})
small_dataset
```

After loading, if we print what a dataset is, So, the dataset is basically an instance of the class DatasetDict. And it says that it is a train split that you are using. Each example in the dataset has two features: a sentence and a label.

There are basically two, six, and four examples within a dataset. We define a truncation function that basically restricts the size of each sentence. To a maximum of 50 tokens. We take 128 random examples from 2,264 and use them as a trendset. And some 32 examples are used for the validation split. So we split the dataset into two parts: training and validation.

So this is a very standard practice in ML, isn't it?

```
num_rows: 32
}}
}}

small_dataset['train'][:10]

{'sentence': ['Ahlstrom 's share is quoted on the NASDAQ OMX Helsinki .',
'Viking will pay EUR 130 million for the new ship , which will be completed in January 2008 .',
'' This vessel order will help Aspo secure the long-term competitiveness of its fleet , both in terms of technology and pricing .',
'The above mentioned shareholders will suggest that a monthly salary of EUR 1,400 would be paid for the Board members outside the company .',
'Consolidated pretax profit decreased by 69.2 % to EUR 41.8 mn from EUR 133.1 mn in 2007 .',
'Niklas Skogster has been employed by the ABB Group in various positions concerning the development of operations .',
'In 2009 , Lee & Man had a combined annual production capacity of close to 4.5 million tonnes of paper and 300,000 tonnes of pulp .',
'The new name of the Sanoma Division will be Sanoma News .',
'He wore a black beanie-type cap and a black jacket .',
'The combined capital of these funds is expected to be EUR 100mn-150mn .'],
'label': [1, 1, 2, 1, 0, 1, 1, 1, 1, 1]}

[] # Prepare the dataset - this tokenizes the dataset in batches of 16 examples.
small_tokenized_dataset = small_dataset.map(
    lambda example: tokenizer(example['sentence'], padding=True, truncation=True), # https://huggingface.co/docs/transformers
    batched=True,
    batch_size=16

small_tokenized_dataset = small_tokenized_dataset.remove_columns(["sentence"])
small_tokenized_dataset = small_tokenized_dataset.remove_columns(["label"])

35:16 / 48:39 completed at 3:54 PM
```

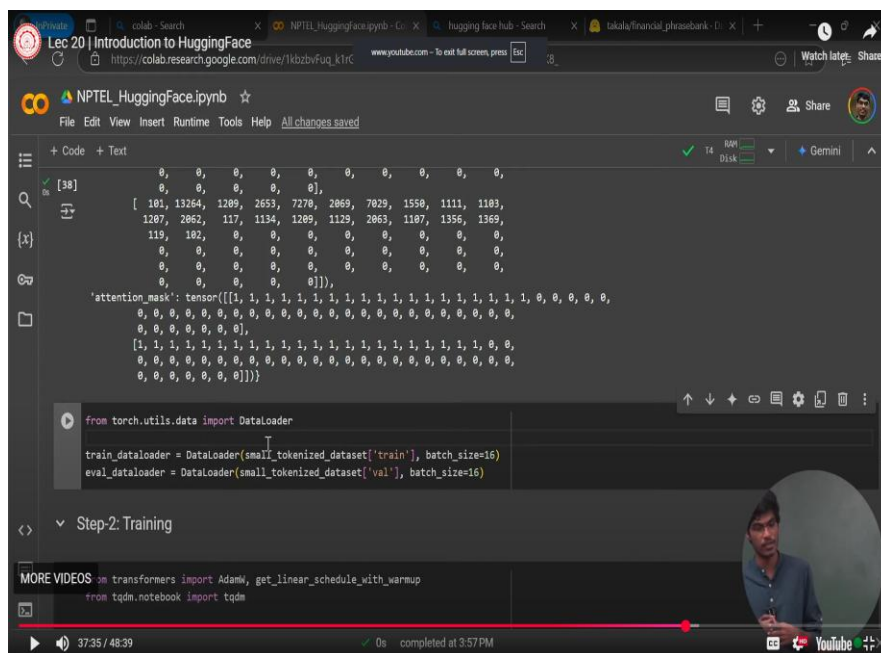
And like this, we can see what the first ten sentences look like. So if we run all of this, and then this. Okay, so these are the first ten sentences. So an example sentence is supposed to be the combined capital of these funds. It is expected to be between 100 million and 150 million euros.

So this is a kind of financial stress. And this one level indicates that it's neutral; two indicates that it's positive. And zero indicates it's negative, right? So, this is how the dataset now looks. Sentences are followed by their levels, right?

function is each example, right? So we pass the sentence, a component of this example, through the tokenizer. With padding and truncation, we use a batch size of 16. So when we map the data set using this defined lambda function, It is first tokenized and then batched into 16 sentence. Or are 16 tokenized sentences put in each batch, right? Then we remove some columns and format the dataset.

so that now the dataset after removing the, so what do we do? We removed the sentence column because we don't need the raw sentences. Now that we already have the tokenized input IDs, don't we? And we rename some columns, and the format is set to touch. So all are like down by touch tensors, right? So now, how does the dataset look? So now the dataset looks like we have the labels in that form PyTorch tensors. We have the input IDs. So there are 16 input IDs in each batch. In this 16-item list of input IDs in the batch. So, if we look at the first two examples, this is one sentence of input ID. zeros are the padded tokens, and another, and this attention marks tells you.

So this is how we have the dataset now.



The screenshot shows a Google Colab notebook titled "NPTEL_HuggingFace.ipynb". The interface includes a menu bar (File, Edit, View, Insert, Runtime, Tools, Help), a toolbar with icons for file operations, and a status bar at the bottom showing "37/35 / 48:39" and "completed at 3:57 PM".

The notebook content is divided into two main sections:

- Code Section:** Contains a Python code cell with the following code:

```
from torch.utils.data import DataLoader

train_dataloader = DataLoader(smaller_tokenized_dataset['train'], batch_size=16)
eval_dataloader = DataLoader(smaller_tokenized_dataset['val'], batch_size=16)
```
- Output Section:** Displays the output of the code execution, showing a list of tokenized sentences (input IDs) and an attention mask tensor. The output is formatted as a JSON object with keys for the input IDs and the attention mask.

The output shows a list of tokenized sentences (input IDs) and an attention mask tensor. The input IDs are represented as a list of lists, where each inner list represents a sentence. The attention mask is a tensor of shape (16, 16), where each row represents the attention weights for a specific sentence in the batch.

Lec 20 | Introduction to HuggingFace

NPTEL_HuggingFace.ipynb

```
from transformers import AdamW, get_linear_schedule_with_warmup
from tqdm.notebook import tqdm

model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-cased', num_labels=3)

num_epochs = 1
num_training_steps = len(train_dataloader)
optimizer = AdamW(model.parameters(), lr=5e-5, weight_decay=0.01)
lr_scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=num_training_steps)

best_val_loss = float("inf")
progress_bar = tqdm(range(num_training_steps))
for epoch in range(num_epochs):
    # training
    model.train()
    for batch_i, batch in enumerate(train_dataloader):
        # batch = ([text1, text2], [0, 1])

        output = model(**batch)

        optimizer.zero_grad()
        output.loss.backward()
```

37:48 / 48:39

Lec 20 | Introduction to HuggingFace

NPTEL_HuggingFace.ipynb

```
optimizer = AdamW(model.parameters(), lr=5e-5, weight_decay=0.01)
lr_scheduler = get_linear_schedule_with_warmup(optimizer, num_warmup_steps=0, num_training_steps=num_training_steps)

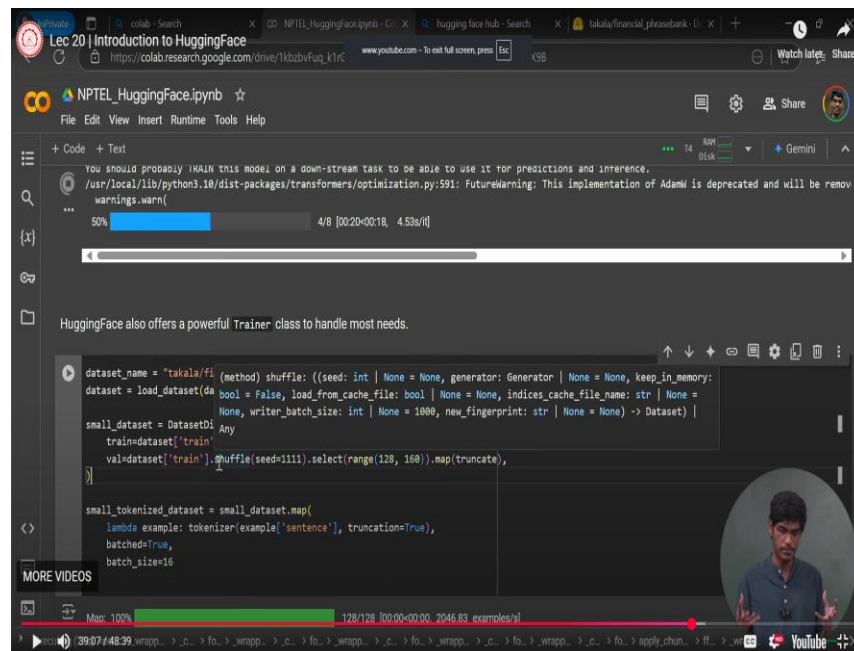
best_val_loss = float("inf")
progress_bar = tqdm(range(num_training_steps))
for epoch in range(num_epochs):
    # training
    model.train()
    for batch_i, batch in enumerate(train_dataloader):
        # batch = ([text1, text2], [0, 1])

        output = model(**batch)

        optimizer.zero_grad()
        output.loss.backward()
        optimizer.step()
        lr_scheduler.step()
        progress_bar.update(1)

    # validation
    model.eval()
    for batch_i, batch in enumerate(eval_dataloader):
        with torch.no_grad():
            output = model(**batch)
            loss += output.loss
```

38:11 / 48:39

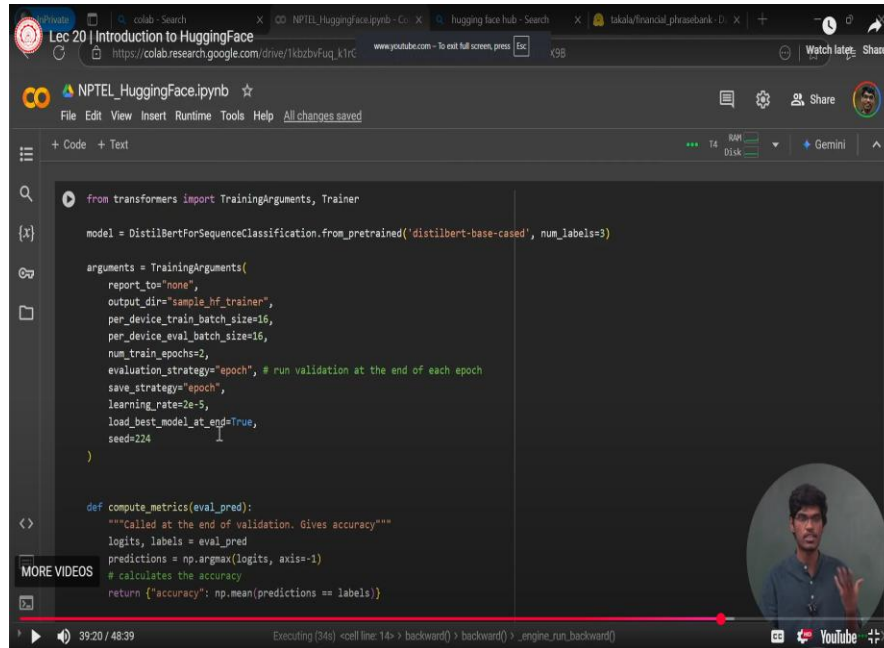


And there is something called a data loader in Torch that you already know. which basically fits these batches of data into the model. while training right. Now, how do you train it? We already discussed this in the Python tutorial.

In this tutorial, I will tell you what to do. We define an optimizer; you define a learning scheduler. And for each epoch, what you do is train the model. that is now you set the request grad equal to true for all parameters And for the batches, you first pass the batch to the model. Initialize the gradient of the tensors to zero and then perform backpropagation, right? Optimize the step that updates the weights using the gradient computed by backpropagation. And then learning about a scheduler if there is a word decal, stuffing things like that.

And then you can evaluate it. So this is how a standard PyTorch training loop works, right? You can pause and maybe see this code again. But this is a standard PyTorch training loop. You must be very familiar with that by now, right? Okay. So that's how you train the model using PyTorch. But then Hugging Face provides something more powerful. It provides something called a trainer class, doesn't it? So, the trainer class will be enough to handle most of this fine-tuning training work, right? So, for the trainer class, what you need is to load the dataset again.

And in this small data set, you just have that trend test split. and similarly we define this lambda function where this tokenized right. So, after tokenization, you need to specify your training arguments, right?



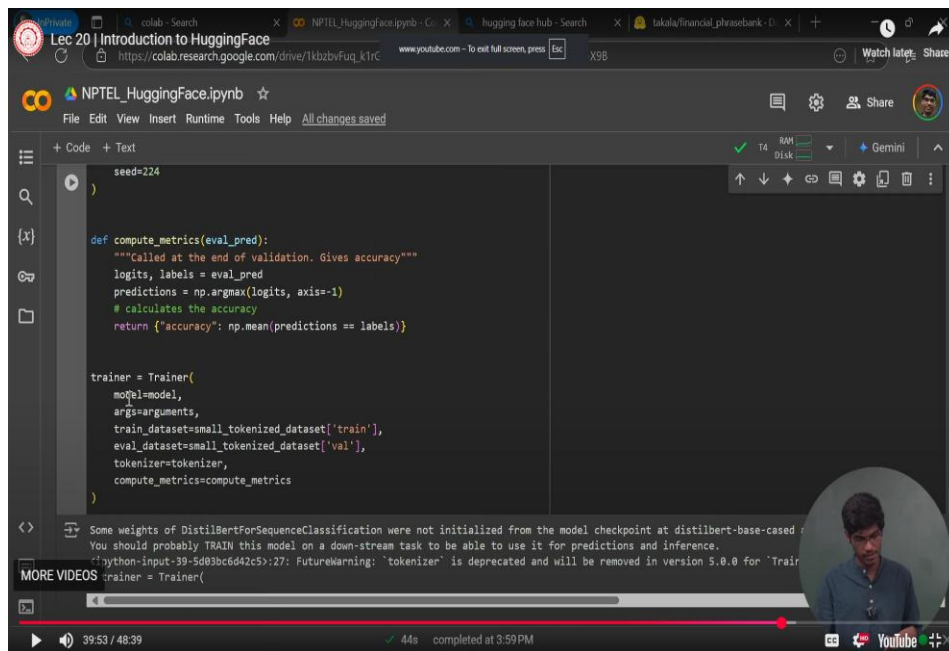
```
from transformers import TrainingArguments, Trainer

model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-cased', num_labels=3)

arguments = TrainingArguments(
    report_to="none",
    output_dir="sample_hf_trainer",
    per_device_train_batch_size=16,
    per_device_eval_batch_size=16,
    num_train_epochs=2,
    evaluation_strategy="epoch", # run validation at the end of each epoch
    save_strategy="epoch",
    learning_rate=2e-5,
    load_best_model_at_end=True,
    seed=224
)

def compute_metrics(eval_pred):
    """Called at the end of validation. Gives accuracy"""
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    # calculates the accuracy
    return {"accuracy": np.mean(predictions == labels)}
```

So for using the trainer class of a hundred, So, what do you need to specify for the training arguments? You need to specify the batch size. So, per device, we are using one GPU. There can be eight GPUs, right? So that's what this per-device feature does. And then you don't need to specify the number of training epochs. The evaluation strategy basically calculates the validation score for each epoch, doesn't it? And then how do you say, "What is the learning rate?" and things like that?



```
seed=224

def compute_metrics(eval_pred):
    """Called at the end of validation. Gives accuracy"""
    logits, labels = eval_pred
    predictions = np.argmax(logits, axis=-1)
    # calculates the accuracy
    return {"accuracy": np.mean(predictions == labels)}

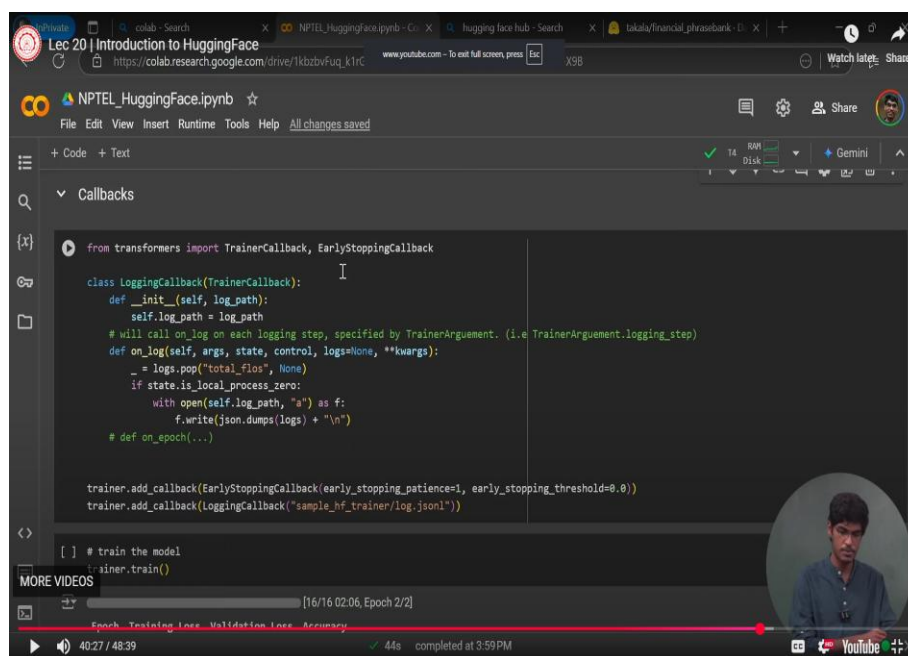
trainer = Trainer(
    model=model,
    args=arguments,
    train_dataset=small_tokenized_dataset['train'],
    eval_dataset=small_tokenized_dataset['val'],
    tokenizer=tokenizer,
    compute_metrics=compute_metrics
)
```

Some weights of DistilBertForSequenceClassification were not initialized from the model checkpoint at distilbert-base-cased. You should probably TRAIN this model on a down-stream task to be able to use it for predictions and inference.

python-input-39-5d83bc6d42c5>:27: FutureWarning: 'tokenizer' is deprecated and will be removed in version 5.0.0 for 'Trainer'.

MORE VIDEOS

39:53 / 48:39 44s completed at 3:59 PM



```
from transformers import TrainerCallback, EarlyStoppingCallback

class LoggingCallback(TrainerCallback):
    def __init__(self, log_path):
        self.log_path = log_path
        # will call on_log on each logging step, specified by TrainerArgument. (i.e TrainerArgument.logging_step)
    def on_log(self, args, state, control, logs=None, **kwargs):
        _ = logs.pop("total_flos", None)
        if state.is_local_process_zero:
            with open(self.log_path, "a") as f:
                f.write(json.dumps(logs) + "\n")
        # def on_epoch(...)

trainer.add_callback(EarlyStoppingCallback(early_stopping_patience=1, early_stopping_threshold=0.0))
trainer.add_callback(LoggingCallback("sample_tf_trainer/log.jsonl"))

[ ] # train the model
trainer.train()
```

MORE VIDEOS

16/16 02:06, Epoch 2/2

Epoch Training Loss Validation Loss Accuracy

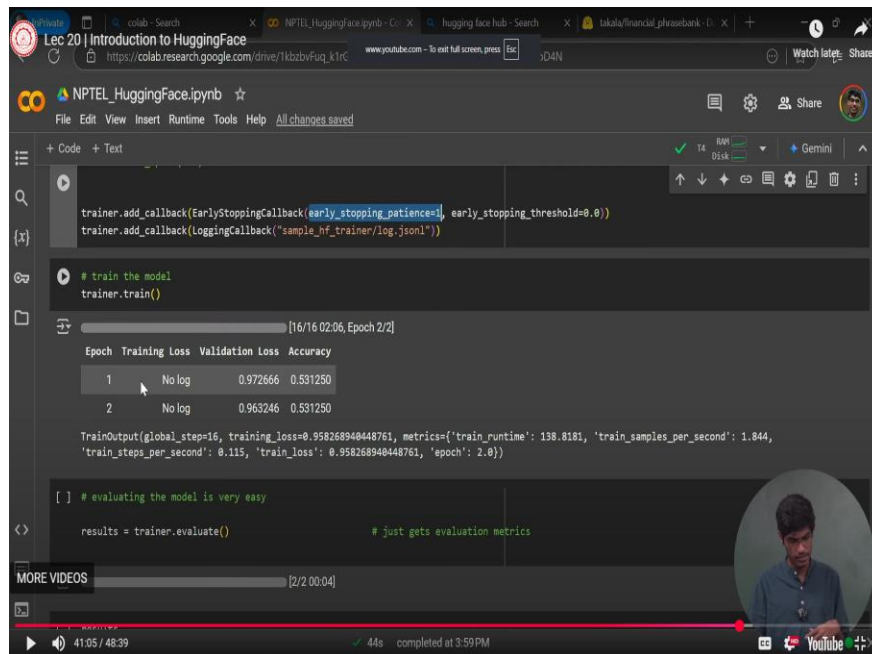
40:27 / 48:39 44s completed at 3:59 PM

So in the trainer class, you pass the model and all of these training arguments.

You have defined right, and then. You basically pass the tokenizer and the compute matrix that you want to use. So I won't run this because it takes a little bit of time, right? So I won't run this here. You can just pause the code and the video. Type the code, and then run it yourself, right? So this is what the trainer module does.

Now you can also specify some callbacks, such as checks on the validation accuracy. And maybe stop your training early—things like that. You can add that using `trainer.addCallback`, like the early-stopping callback. Patience is equal to one.

Because you can wait for one epoch, you may see if the validation accuracy is decreasing. or not And if it continues to decrease for two epochs, you can stop. and there is some threshold. So you can maybe read the ad callback documentation a bit. if you want to use this early stopping functionalities



The screenshot shows a Jupyter Notebook titled "NPTEL_HuggingFace.ipynb" with the following code and output:

```
trainer.add_callback(EarlyStoppingCallback(early_stopping_patience=1, early_stopping_threshold=0.0))
trainer.add_callback(LoggingCallback("sample_hf_trainer/log.jsonl"))

# train the model
trainer.train()
```

The output shows a progress bar at [16/16 02:06, Epoch 2/2] and a table of training metrics:

Epoch	Training Loss	Validation Loss	Accuracy
1	No log	0.972666	0.531250
2	No log	0.963246	0.531250

Below the table, the output shows the training output for epoch 2:

```
TrainOutput(global_step=16, training_loss=0.958268940448761, metrics={'train_runtime': 138.8181, 'train_samples_per_second': 1.844, 'train_steps_per_second': 0.115, 'train_loss': 0.958268940448761, 'epoch': 2.0})
```

The notebook also includes a comment "# evaluating the model is very easy" and the code `results = trainer.evaluate()` with a note "# just gets evaluation metrics".

And then, if you call `trainer.train`, I've already done that. In this way, you are informed of the validation loss and the corresponding accuracy. And the evaluation is also quite simple. You just do `trainer.evaluate`, and... It will automatically evaluate the validation data. If you want to make predictions on some data, you just do `trainer.predict()`. And pass the test set.

So, in the evaluation metric here, in the trainer function, If you remember, in the trainer argument and in the evaluation strategy, You already specified the epoch, and here, compute the metrics. You already specified the metric by which you want to evaluate. So the evaluation and prediction are also made quite simple using the Trainer module. I

strongly suggest that you go to the Hugging Face documentation for the Hugging Face Trainer.

It's very simple. You just searched for the Hugging Face Trainer, right? And you will be directed to this training module. So you go through this documentation once to familiarize yourself with all the parameters. It takes in, and how do you use that correctly? It will be really helpful. You can pause the video and run this yourself on your Colab notebook, right? So after the prediction, you get the results and things, right? Now, before we end, all these things we did until now, right? So, in this way, you can have a test example.

You can simply load the fine-tuned model that you saved during training. So, the checkpointing is the thing. So here, if you see again, we wrote that it is a load-based model. at end is going to be true and save the strategy will be poke.

So we are saving this. The checkpoints were saved. So, after eight epochs, we saved the checkpoint. So you can load that this way from pre-trend. Then specify the local path, as I said. If you have a local path, you can specify it. Then pass the input string to the tokenizer, and you can predict and see.

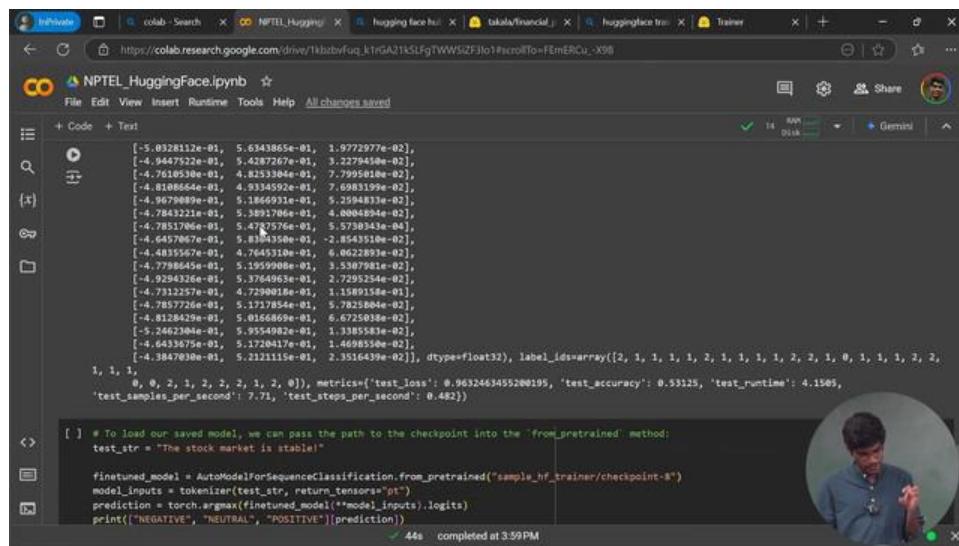
So, these all use the encoder-only models. Now, how do you use a decoder-only model? Say, you just want to generate stuff, right? Generate text. So, there, it's pretty much the same. There, you also need to specify the auto model. But now the task is just a causal LM, right? So, causal LM is basically the task of next-generation prediction. So you just load the auto model for the causal LM class. And there, from pre-training, you can specify any decoder-only model.

Suppose we use GPT2 here, or say we distill GPT2, right? And then how do you basically pass on the generated? So there is a function called `model.generate`. So, within the `model.generate`, what do you pass? You pass the tokenized prompt, unpacked tokenized prompt.

So first you suppose once upon a time is your prompt. So you tokenize the "once upon a prompt" phrase. So this GPT tokenizer is basically... Auto-tokenizer from a pre-trained GPT-2. Like before, very similar to what we did for the decoder-only model. The only

difference is that during model loading, you have to specify the auto model for the causal element, right? So you just do `model.generate()` and pass the tokenized input IDs. What is the maximum length, and how many maximum tokens do you want to generate? So I just restrict it to 50. And this `doSampleEqualToTrue` basically does the sampling. So it doesn't, if you do `doSampleEqualToFalse`, it does grid decoding. So, we have already discussed grid decoding, right? So here I do `doSampleEqualToTrue`, and `top` is equal to 0.9. So we have already discussed nuclear sampling and top sampling, haven't we? So they are specified. You can also specify the temperature parameter instead of top-p. You can specify a temperature equal to 0.3 instead of top-p. You can also use top-k sampling. Like top k is equal to something based on what you want it to be. So here I perform nucleus sampling, and there I basically generate 10 possible completions.

Once upon a time, right? So if I like "Once Upon a Time" for such great uncertainty, the truth of this matter or something that GPT-2 is generating. So this is basically a sentence-completion task where you give a prompt. And you expect the model to complete the prompt or generate something based on the prompts.



```

[[-5.0328112e-01, 5.6343865e-01, 1.9722977e-02],
 [-4.9447522e-01, 5.4287267e-01, 3.2279450e-02],
 [-4.7610530e-01, 4.8253304e-01, 7.7995010e-02],
 [-4.8108664e-01, 4.9334592e-01, 7.6983199e-02],
 [-4.9079089e-01, 5.1866931e-01, 5.2948832e-02],
 [-4.7843221e-01, 5.3891706e-01, 4.8004894e-02],
 [-4.7851706e-01, 5.4787576e-01, 5.5730343e-04],
 [-4.6457067e-01, 5.8304350e-01, -2.8543510e-02],
 [-4.4835567e-01, 4.7645310e-01, 6.0622893e-02],
 [-4.7798645e-01, 5.1959908e-01, 5.5307981e-02],
 [-4.9284326e-01, 5.3764963e-01, 2.7291254e-02],
 [-4.7312257e-01, 4.7290018e-01, 1.1589158e-01],
 [-4.7857726e-01, 5.1717054e-01, 5.7825804e-02],
 [-4.8128429e-01, 5.0166869e-01, 6.6725030e-02],
 [-5.2462304e-01, 5.9554082e-01, 1.3385583e-02],
 [-4.6433875e-01, 5.3728017e-01, 1.4688550e-02],
 [-4.3847030e-01, 5.2121115e-01, 2.3516439e-02]], dtype=float32), label_ids=array([2, 1, 1, 1, 1, 2, 1, 1, 1, 1, 2, 1, 0, 1, 1, 1, 2, 2,
 1, 1, 1, 0, 2, 1, 2, 2, 1, 2, 0]), metrics={'test_loss': 0.9632463455200195, 'test_accuracy': 0.53125, 'test_runtime': 4.1505,
'test_samples_per_second': 7.71, 'test_steps_per_second': 0.482})

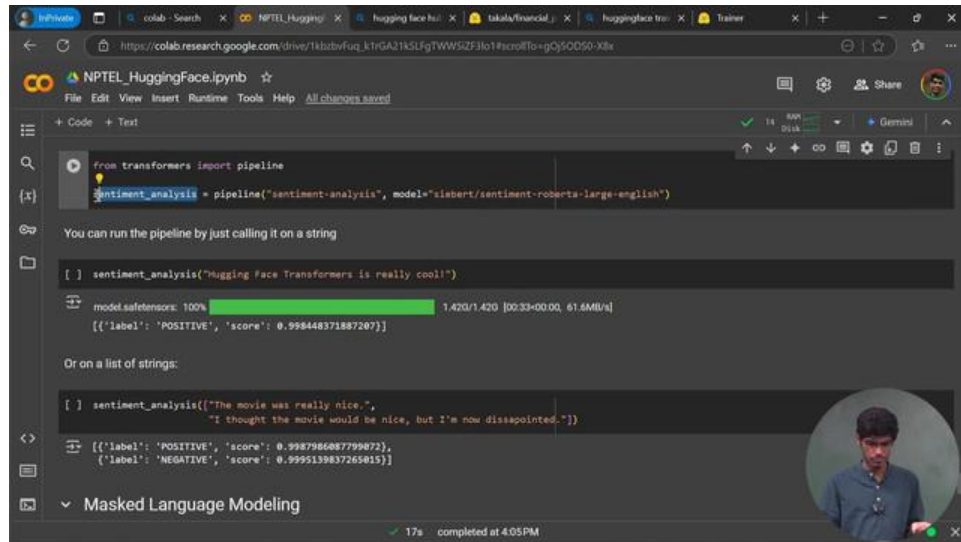
[ ] # To load our saved model, we can pass the path to the checkpoint into the 'from_pretrained' method:
test_str = "The stock market is stable"

finetuned_model = AutoModelForSequenceClassification.from_pretrained("sample_hf_trainer/checkpoint-8")
model_inputs = tokenizer(test_str, return_tensors="pt")
prediction = torch.argmax(finetuned_model(**model_inputs).logits)
print(["NEGATIVE", "NEUTRAL", "POSITIVE"][prediction])

```

Now, there is something called a pipeline in the pipeline's function. The pipeline class automates your entire pipeline for a particular task. For some popular tasks. Suppose we are dealing with the task of sentiment analysis. So, Hugging Face already has a full pipeline written for sentiment analysis. So, you just take the pipeline class; you call the pipeline class. You pass the argument, the task name, and the model, right? So, suppose the task

name here is sentiment analysis and the model is say, sentiment robot, right? So, you pass that into the pipeline, and then you can simply. To this object, sentiment analysis, which is an instance of the class Pipeline, Defined using the class name and the model, the sentence can be simply passed and get its predicted output.



The screenshot shows a Google Colab notebook interface. The top bar includes the Colab logo and the file name 'NPTEL_HuggingFace.ipynb'. The code editor displays the following Python code:

```
from transformers import pipeline
sentiment_analysis = pipeline("sentiment-analysis", model="distilbert/distilbert-base-uncased")
```

Below the code, there is a text prompt: "You can run the pipeline by just calling it on a string". The output shows the result of calling the pipeline on a string:

```
[ ] sentiment_analysis("Hugging Face Transformers is really cool!")
model.safetensors: 100% 1.42G/1.42G [00:33<00:00, 61.6MB/s]
[{'label': 'POSITIVE', 'score': 0.998448371887267}]
```

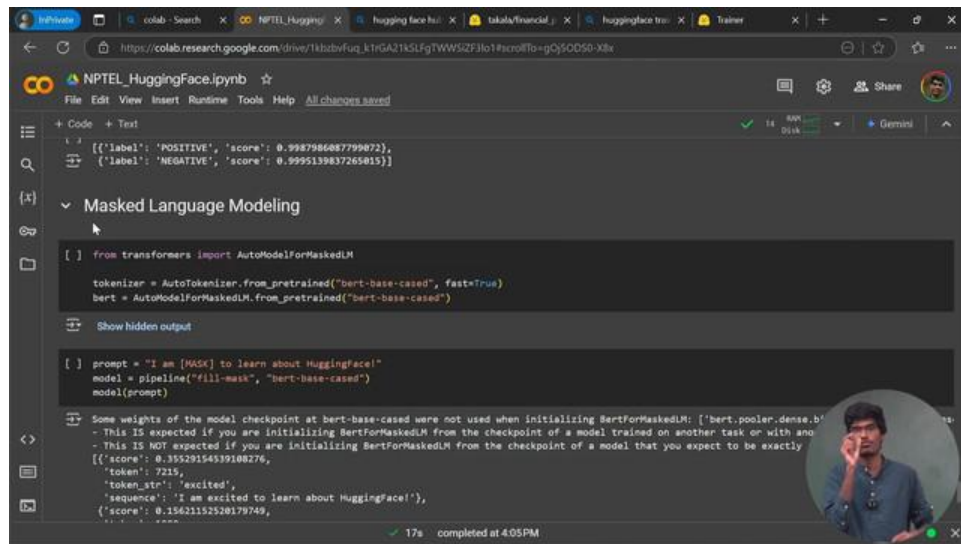
Below this, there is another text prompt: "Or on a list of strings:". The output shows the result of calling the pipeline on a list of strings:

```
[ ] sentiment_analysis(["The movie was really nice.",
                        "I thought the movie would be nice, but I'm now disappointed."])
[{'label': 'POSITIVE', 'score': 0.9987886887799672},
 {'label': 'NEGATIVE', 'score': 0.9995139837265813}]
```

The bottom of the notebook shows a section titled "Masked Language Modeling" with a progress bar indicating 17s completed at 4:05 PM.

So there are many pipelines defined for several popular tasks. But generally, when we do research, we define our own tasks. And things like that are not always helpful. But when you are doing some fast coding, you want to get your things done. There are already some very well-defined tasks you want to test. If you want to be really fast, then this pipeline function becomes very handy, right? And to conclude, I will also show you the masked language modeling. So for causal language modeling tasks, we were loading the auto model for causality.

For masked language modeling, what will it be? It's a very simple, very trivial auto model for masked LM, right? So they are also the tokenizer you call support; we're using BERT for support. So you call the BERT tokenizer and the auto model for masked LM from Pretend. You just give the BERT base case, whatever path you have.



```
[[{"label": "POSITIVE", "score": 0.9987986087799072}], [{"label": "NEGATIVE", "score": 0.9995139837265815}]]
```

Masked Language Modeling

```
[ ] from transformers import AutoTokenizer, AutoModelForMaskedLM

tokenizer = AutoTokenizer.from_pretrained("bert-base-cased", fast=True)
bert = AutoModelForMaskedLM.from_pretrained("bert-base-cased")

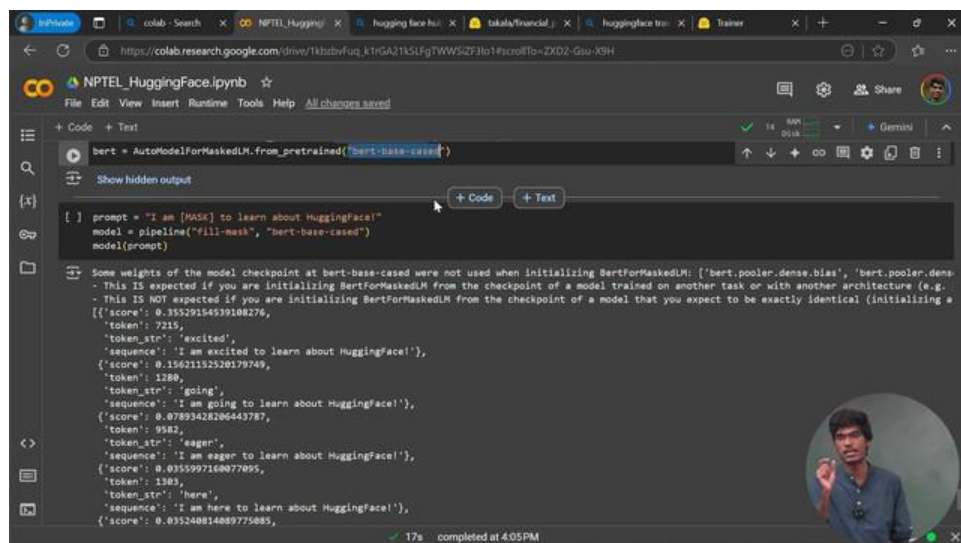
[ ] prompt = "I am [MASK] to learn about HuggingFace!"
model = pipeline("fill-mask", "bert-base-cased")
model(prompt)
```

Some weights of the model checkpoint at bert-base-cased were not used when initializing BertForMaskedLM: ['bert.pooler.dense.bias', 'bert.pooler.dense.weight', 'bert.pooler.dense.weight']. This is expected if you are initializing BertForMaskedLM from the checkpoint of a model trained on another task or with another architecture (e.g. a GPT-J model). This is NOT expected if you are initializing BertForMaskedLM from the checkpoint of a model that you expect to be exactly identical (initializing a BertForMaskedLM from the checkpoint of a BertForMaskedLM model).

```
[{"score": 0.35529154539108276, "token": 7215, "token_str": "excited", "sequence": "I am excited to learn about HuggingFace!"}, {"score": 0.15621152520179749, "token": 1280, "token_str": "going", "sequence": "I am going to learn about HuggingFace!"}, {"score": 0.07893428206443787, "token": 9582, "token_str": "eager", "sequence": "I am eager to learn about HuggingFace!"}, {"score": 0.0355997160077095, "token": 1303, "token_str": "here", "sequence": "I am here to learn about HuggingFace!"}, {"score": 0.03240814089775085, "token": 1280, "token_str": "going", "sequence": "I am going to learn about HuggingFace!"}]]
```

17s completed at 4:05 PM

And what is the masked language modeling objective? The masked language modeling objective is within your input sentences or prompts whatever you call it, there is a masked token. Now, the task of masked language modeling is to predict possible tokens in place of the masked token.



```
bert = AutoModelForMaskedLM.from_pretrained("bert-base-cased")
```

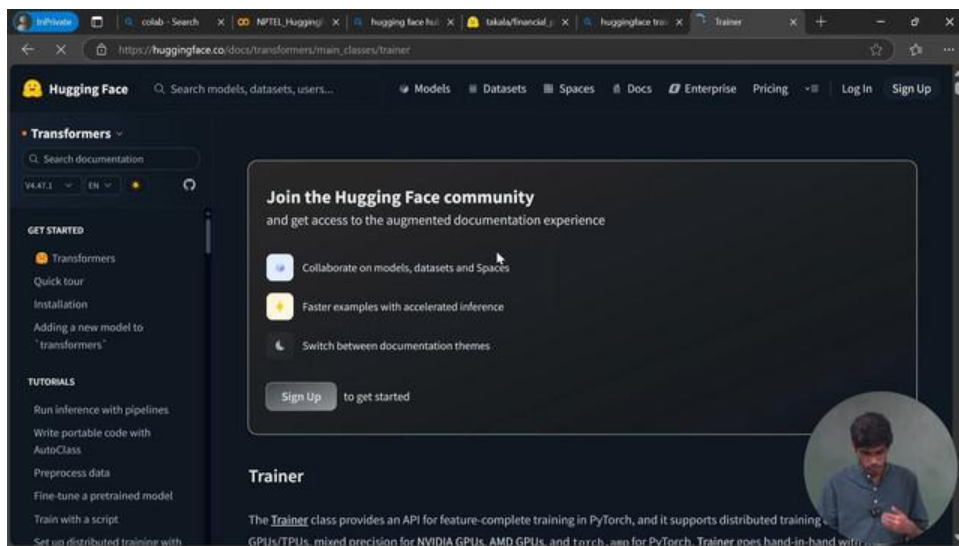
```
[ ] prompt = "I am [MASK] to learn about HuggingFace!"
model = pipeline("fill-mask", "bert-base-cased")
model(prompt)
```

Some weights of the model checkpoint at bert-base-cased were not used when initializing BertForMaskedLM: ['bert.pooler.dense.bias', 'bert.pooler.dense.weight', 'bert.pooler.dense.weight']. This is expected if you are initializing BertForMaskedLM from the checkpoint of a model trained on another task or with another architecture (e.g. a GPT-J model). This is NOT expected if you are initializing BertForMaskedLM from the checkpoint of a model that you expect to be exactly identical (initializing a BertForMaskedLM from the checkpoint of a BertForMaskedLM model).

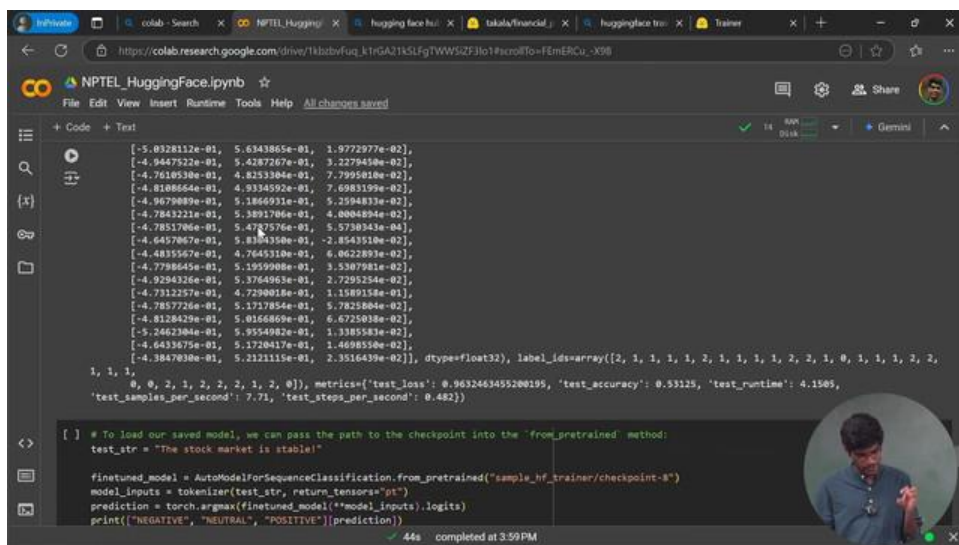
```
[{"score": 0.35529154539108276, "token": 7215, "token_str": "excited", "sequence": "I am excited to learn about HuggingFace!"}, {"score": 0.15621152520179749, "token": 1280, "token_str": "going", "sequence": "I am going to learn about HuggingFace!"}, {"score": 0.07893428206443787, "token": 9582, "token_str": "eager", "sequence": "I am eager to learn about HuggingFace!"}, {"score": 0.0355997160077095, "token": 1303, "token_str": "here", "sequence": "I am here to learn about HuggingFace!"}, {"score": 0.03240814089775085, "token": 1280, "token_str": "going", "sequence": "I am going to learn about HuggingFace!"}]]
```

17s completed at 4:05 PM

So basically, where there is a masked token, The masked language model will give you a probability distribution as output over the set of vocabulary tokens. So suppose the prompt is that I am asked to learn about Hugging Face. and the pipeline is fill mask. So fill mask is the task now, right? And you give the model the base case. And then, if you pass the prompt to the model, what does the model output? The model outputs a score that is a probability for each possible token.



So this was a brief tutorial on Hugging Face. To learn more, you should really go and check out their documentation. Read through the documentation and familiarize yourself with the different classes and methods in HuggingFace. So, thank you, and I hope you're enjoying the course.



So happy to learn!