NLP with Transformers

Chaper 1&2

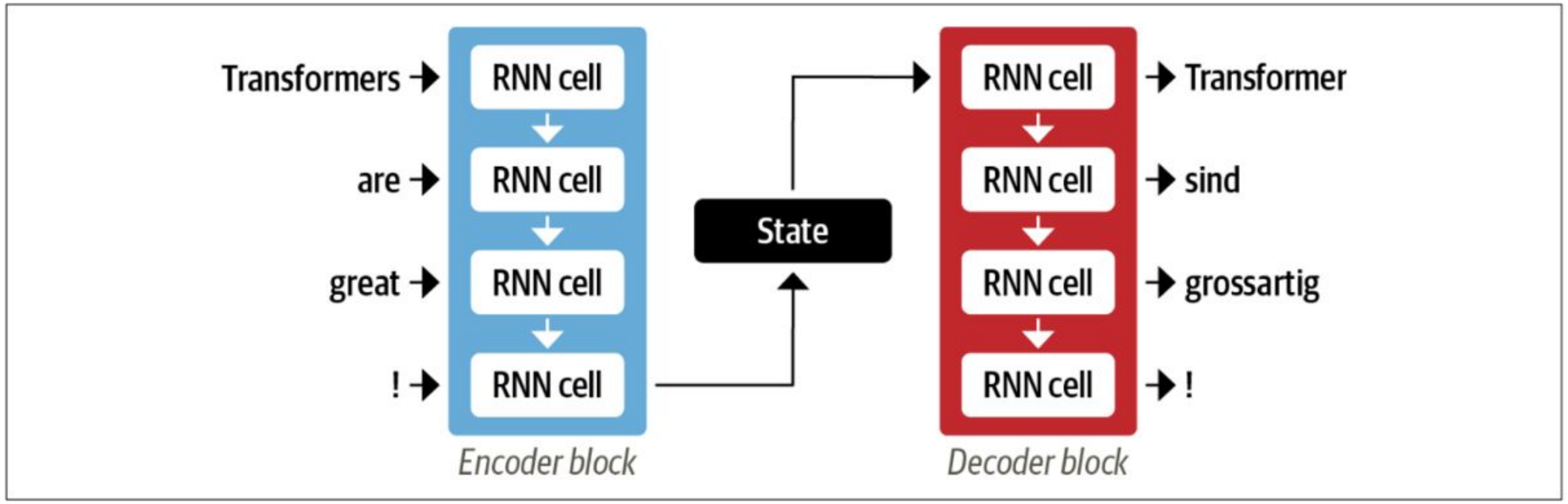
목차

Core Concepts of Transformer.

(1) The Change of Encoder-Decoder FrameWork

 (1) The Change of Encoder-Decoder FrameWork

An encoder-decoder architecture with a pair of RNNs



💡 **The Encoder encodes** the information **from the input sequence**
into a numerical representation that is often **called the last hidden state.**
This state is then passed to the decoder,
which generates the output sequence.

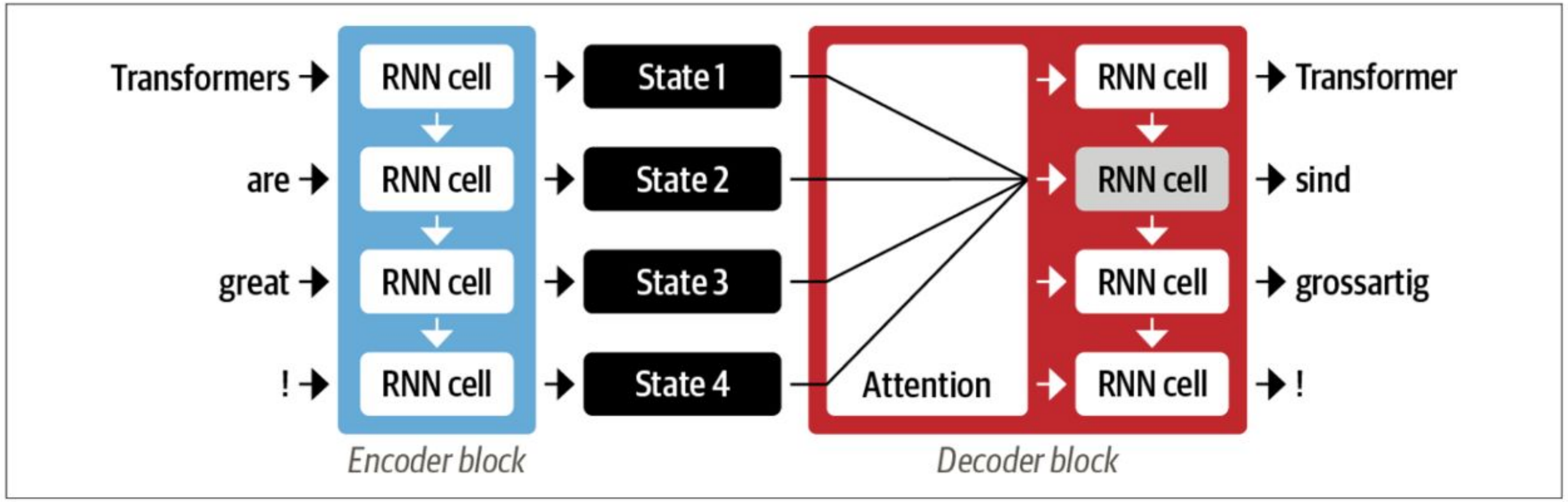
Core Concepts of Transformer. (1) The Change of Encoder-Decoder FrameWork

An encoder-decoder architecture with a pair of RNNs

😟 One weakness of this architecture is that the final
hidden state of the encoder creates an ***information bottleneck***
→ ***The decoder only accesses the last hidden state of the encoder.***
→ ***If the input sequence is long, it is more vulnerable*** *because there is a possibility that information at the beginning may be lost.*

🔑 Fortunately, there is a way out of this bottleneck by **allowing the decoder to have access to all of the encoder's hidden states.**
How? → Attention!

An encoder-decoder architecture with an attention mechanism



💡 - Instead of producing a single hidden state for the input sequence, **the encoder outputs a hidden state at each step that the decoder can access.**
- **the decoder assign a different amount of weight, or "attention," to each of the encoder states at every decoding time step.**

Core Concepts of Transformer. (1) The Change of Encoder-Decoder FrameWork

An encoder-decoder architecture with an attention mechanism

😟 There was still a major shortcoming with using recurrent models for the encoder and decoder
→ Due to the nature of RNN Architecture, **the calculation of hidden states** is performed sequentially and **cannot be parallelized** across the entire input sequence

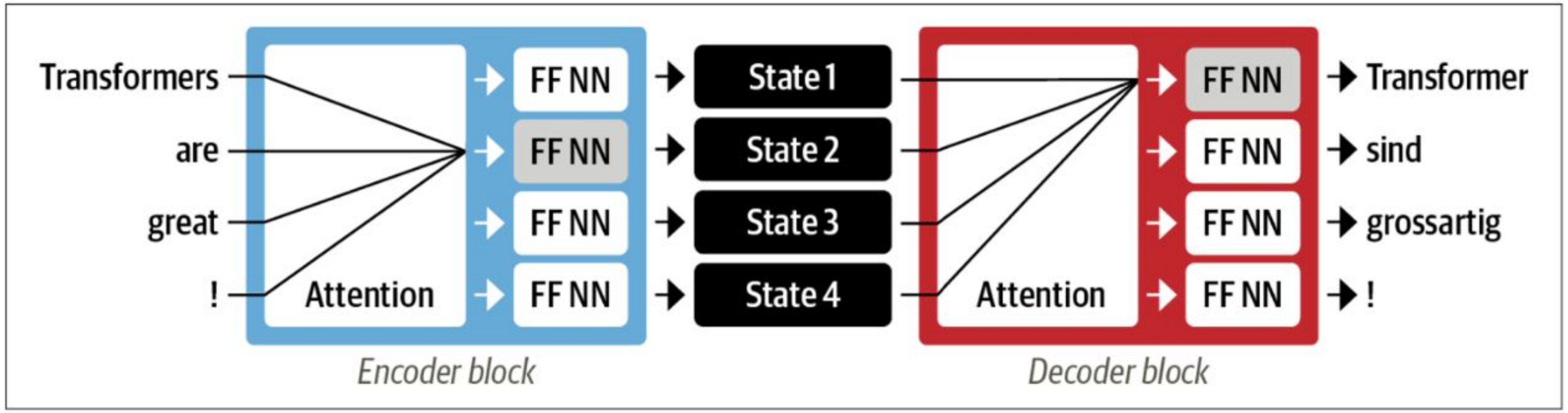
🔑 **Attention is All you Need!**

- dispense with recurrence altogether
- instead rely entirely on a special form of attention called *self-attention*

This architecture can be trained much faster than recurrent models and paved the way for many of the recent breakthroughs in NLP.

(1) The Change of Encoder-Decoder FrameWork

Encoder-decoder architecture of the original Transformer



💡 The basic idea is to **allow attention to operate on all the states in the *same* layer** of the neural net-work.
The encoder and the decoder have their own self-attention mechanisms, whose outputs are fed to feed-forward neural networks.

Core Concepts of Transformer.

(2) Transfer Learning in NLP

😟 Although transfer learning became the standard approach in computer vision, for many years
it was not clear what the analogous pretraining process was for NLP.

New approaches that finally made transfer learning work for NLP

🔑 "ULMFiT" introduced **a general framework to adapt pretrained LSTM models for various tasks**. The elegance of this approach lies in the fact that **no labeled data is required.**

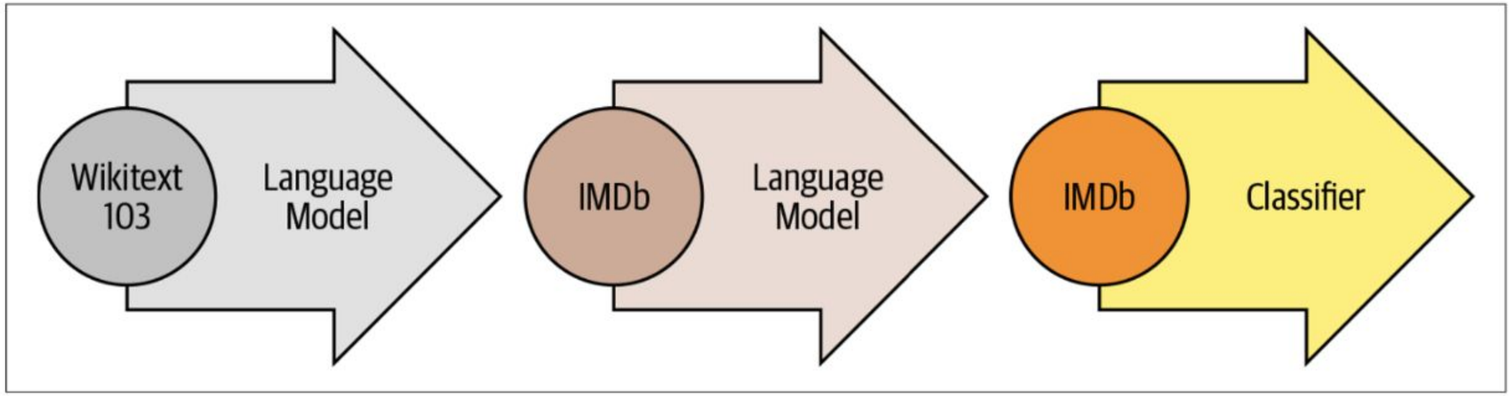


Figure 1-8. The ULMFiT process (courtesy of Jeremy Howard)

Core Concepts of Transformer. (2) Transfer Learning in NLP

New approaches that finally made transfer learning work for NLP

💡 **Three Steps**

Pretraining

- Pretrain the language model on a large-scale corpus.

Domain adaptation

- Adapt the language model to the domain(target) corpus.

Fine-tuning

- Fine-tune the language model with a classification layer for the target task (e.g., classifying the sentiment of movie reviews).

💡 By introducing a viable framework for **pretraining and transfer learning in NLP, ULMFiT provided** the missing piece to make **transformers take off**

GPT & BERT

GPT

Uses only the decoder part of the Transformer architecture, and the same language modeling approach as ULMFiT.

BERT

Uses the encoder part of the Transformer architecture, and a special form of language modeling called *masked language modeling*.

> 💡 GPT and BERT set a new state of the art across a variety of NLP benchmarks and ushered in the age of transformers.

Why use Hugging Face 🤗?

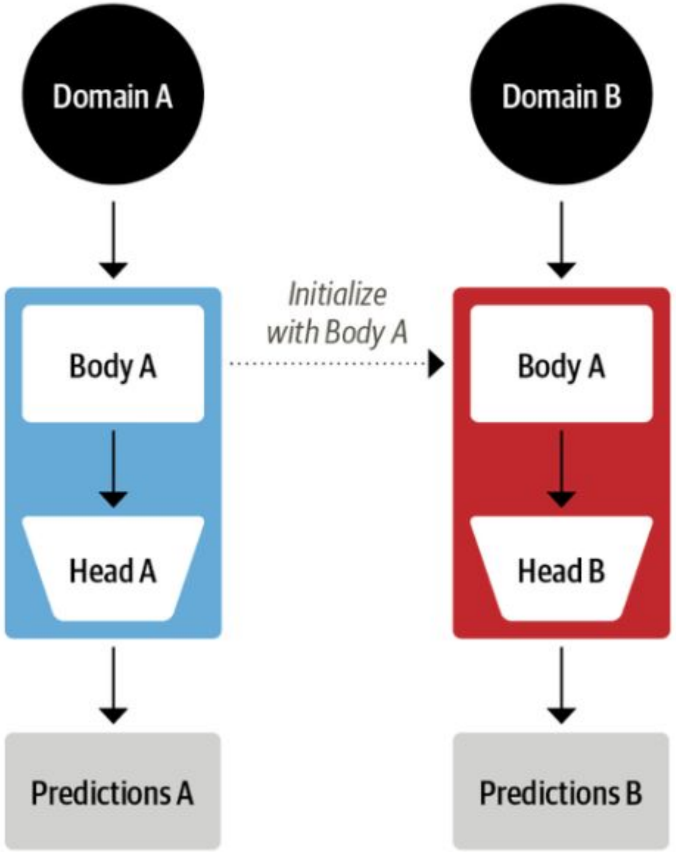
Why use Hugging Face 🤗?

😟 Applying a novel machine learning architecture to a new task can be a complex undertaking, and usually involves many steps.

💡 🤗 Transformers **provides a standardized interface to a wide range of transformer models** as well as code and tools to adapt these models to new use cases.

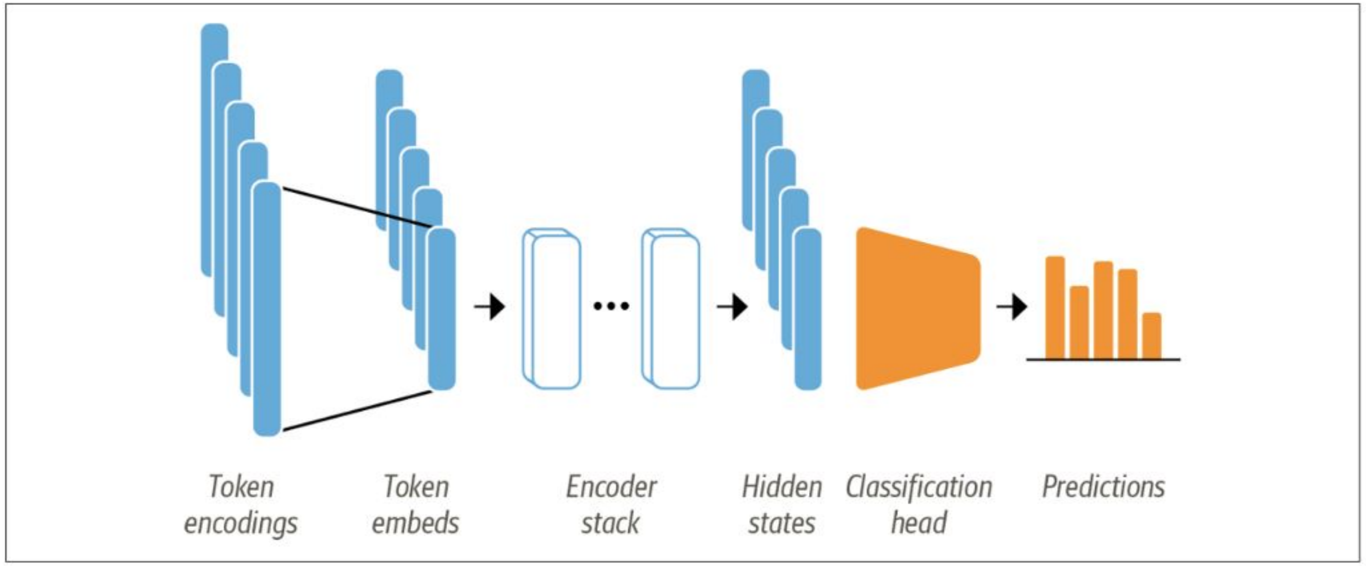
The library currently **supports three major deep learning frameworks** (PyTorch, TensorFlow, and JAX).

In addition, it provides task-specific heads so **you can easily fine-tune transformers** on downstream tasks.



Text Classification
 with Pretrained DistilBERT Using HuggingFace.

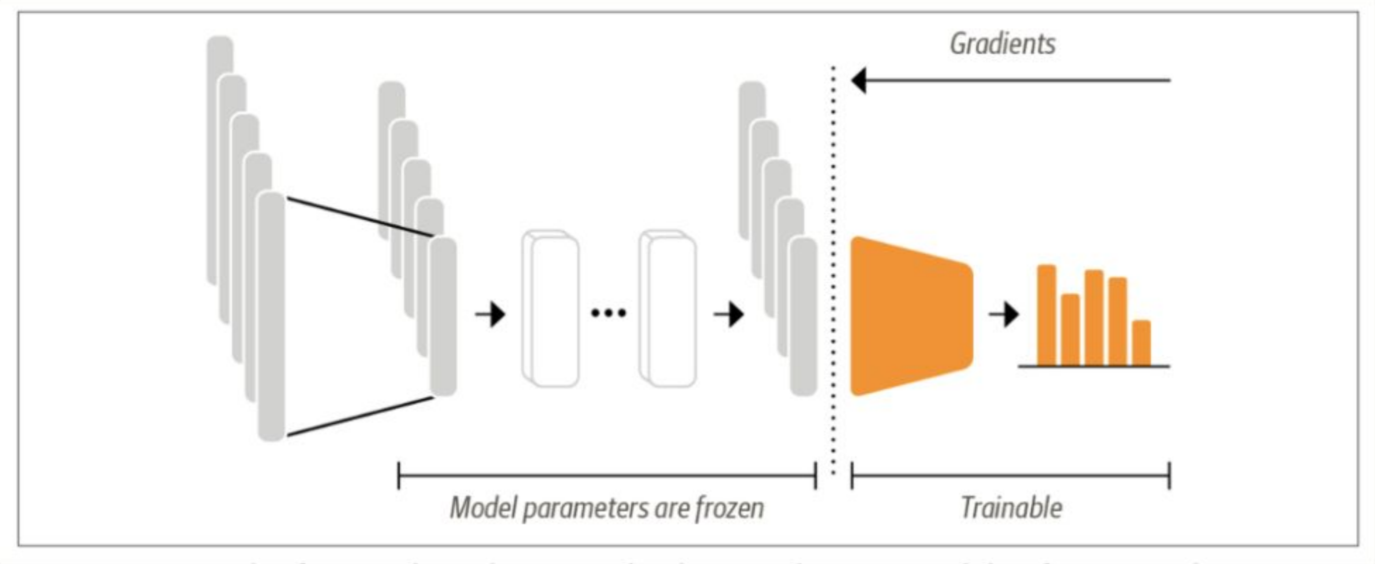
Architecture used for sequence classification with an encoder-based transformer



💡 It consists of (1) **the model's pretrained body** combined with (2) **a custom classification head**.

1. ***Feature extraction***

We **use the hidden states as features and just train a classifier on them**, without modifying the pretrained model.



Step 1 Preparing the Model Input

Text Tokenization & Token Encoding.

💡 **WordPiece is used as the BERT and DistilBERT tokenizer**

```
from transformers import AutoTokenizer

model_ckpt = "distilbert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
text = "Tokenizing text is a core task of NLP"
encoded_text = tokenizer(text, return_tensors="pt")
tokenized_text = tokenizer.convert_ids_to_tokens(encoded_text.input_ids[0])
```

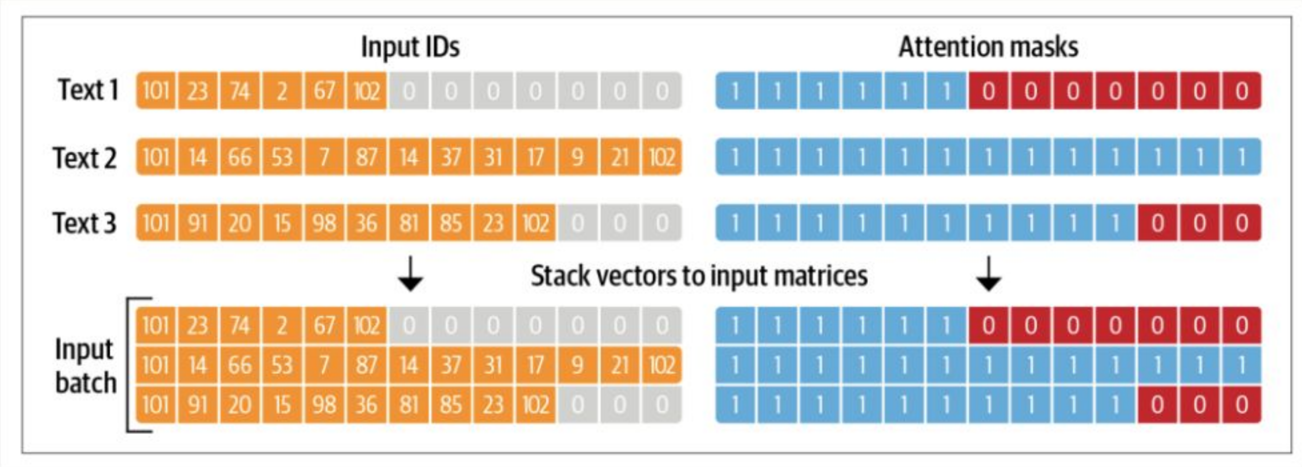
```
original text:
 Tokenizing text is a core task of NLP

tokenized text:
 ['[CLS]', 'token', '##izing', 'text', 'is', 'a', 'core', 'task', 'of', 'nl', '##p', '[SEP]']

encoded_text:
 {'input_ids': tensor([[  101, 19204,  6026,  3793,  2003,  1037,  4563,  4708,  1997, 17953,
          2361,   102]]), 'attention_mask': tensor([[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]])}
```

Why does the Encoded Tokens have attention masks?

❓ → input_ids are padded by the max sequence length of the model.
→ the attention mask is needed to make the model ignore the padded parts of the input.



PitFall

⚠️ When using pretrained models, it is *really* important to make sure that you **use the same tokenizer that the model was trained with.**

Step2 Extracting the last hidden states

```
import torch
from transformers import AutoModel
model_ckpt = "distilbert-base-uncased"
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = AutoModel.from_pretrained(model_ckpt).to(device)

text = "Tokenizing text is a core task of NLP"
inputs = tokenizer(text, return_tensors="pt")

inputs = {k:v.to(device) for k,v in inputs.items()}
with torch.no_grad():
    outputs = model(**inputs)

CLS = outputs.last_hidden_state[:,0]
```

```
Input tensor shape([batch_size, n_tokens]): torch.Size([1, 12])
```

```
last hidden state shape([batch_size, n_tokens, hidden_dim]): torch.Size([1, 12, 768])
```

```
[CLS] tokens shape([batch_size, hidden_dim]): torch.Size([1, 768])
```

Step3 Training a simple classifier with extracted features

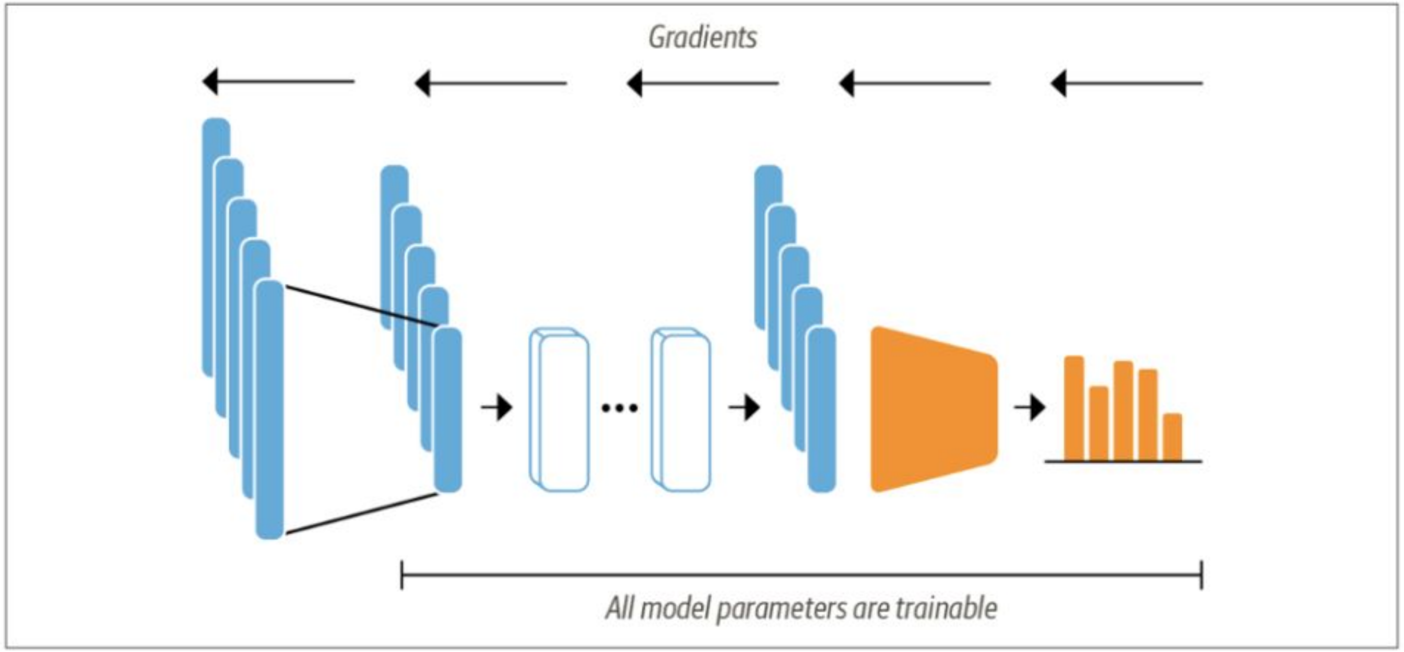
```
from sklearn.linear_model import LogisticRegression

# We increase `max_iter` to guarantee convergence
lr_clf = LogisticRegression(max_iter=3000)
lr_clf.fit(X_train, y_train)
lr_clf.score(X_valid, y_valid)
```

💡 Use the extracted hidden states to train a classification model(its parameters)

💡 2. ***Fine-tuning***

We **train the whole model end-to-end**, which also updates the parameters of the pretrained model.



Loading a pretrained model

```
from transformers import AutoModelForSequenceClassification
    num_labels = 6
    model = (AutoModelForSequenceClassification
             .from_pretrained(model_ckpt, num_labels=num_labels)
             .to(device))
```

AutoModelForSequenceClassification model **has a classification head on top of the pretrained model outputs**, which can be easily trained with the base model.
We **just need to specify how many labels the model has to predict**

Defining the performance metrics

```
from sklearn.metrics import accuracy_score, f1_score
def compute_metrics(pred):
  labels = pred.label_ids
  preds = pred.predictions.argmax(-1)
  f1 = f1_score(labels, preds, average="weighted") acc = accuracy_score(labels,
  return {"accuracy": acc, "f1": f1}
```

Training the model

```
from transformers import Trainer, TrainingArguments
batch_size = 64
logging_steps = len(emotions_encoded["train"]) // batch_size
model_name = f"{model_ckpt}-finetuned-emotion"
training_args = TrainingArguments(output_dir=model_name,
                                  num_train_epochs=2,
                                  learning_rate=2e-5,
                                  per_device_train_batch_size=batch_size,
                                  per_device_eval_batch_size=batch_size,
                                  weight_decay=0.01,
                                  evaluation_strategy="epoch",
                                  disable_tqdm=False,
                                  logging_steps=logging_steps,
                                  push_to_hub=True,
                                  log_level="error")
```

```
from transformers import Trainer
trainer = Trainer(model=model, args=training_args,
                  compute_metrics=compute_metrics,
                  train_dataset=emotions_encoded["train"],
                  eval_dataset=emotions_encoded["validation"],
                  tokenizer=tokenizer)
trainer.train();
```

💡 To define the **training parameters, we use the TrainingArguments class.**
we can **instantiate and fine-tune our model with the Trainer class**

Saving and sharing the model

💡 The NLP community benefits greatly from sharing pretrained and fine-tuned models, and everybody can share their models with others via the Hugging Face Hub.
Any community-generated model can be downloaded from the Hub.
With the Trainer API, saving and sharing a model is simple.

```
trainer.push_to_hub(commit_message="Training completed!")
```

Thankyo