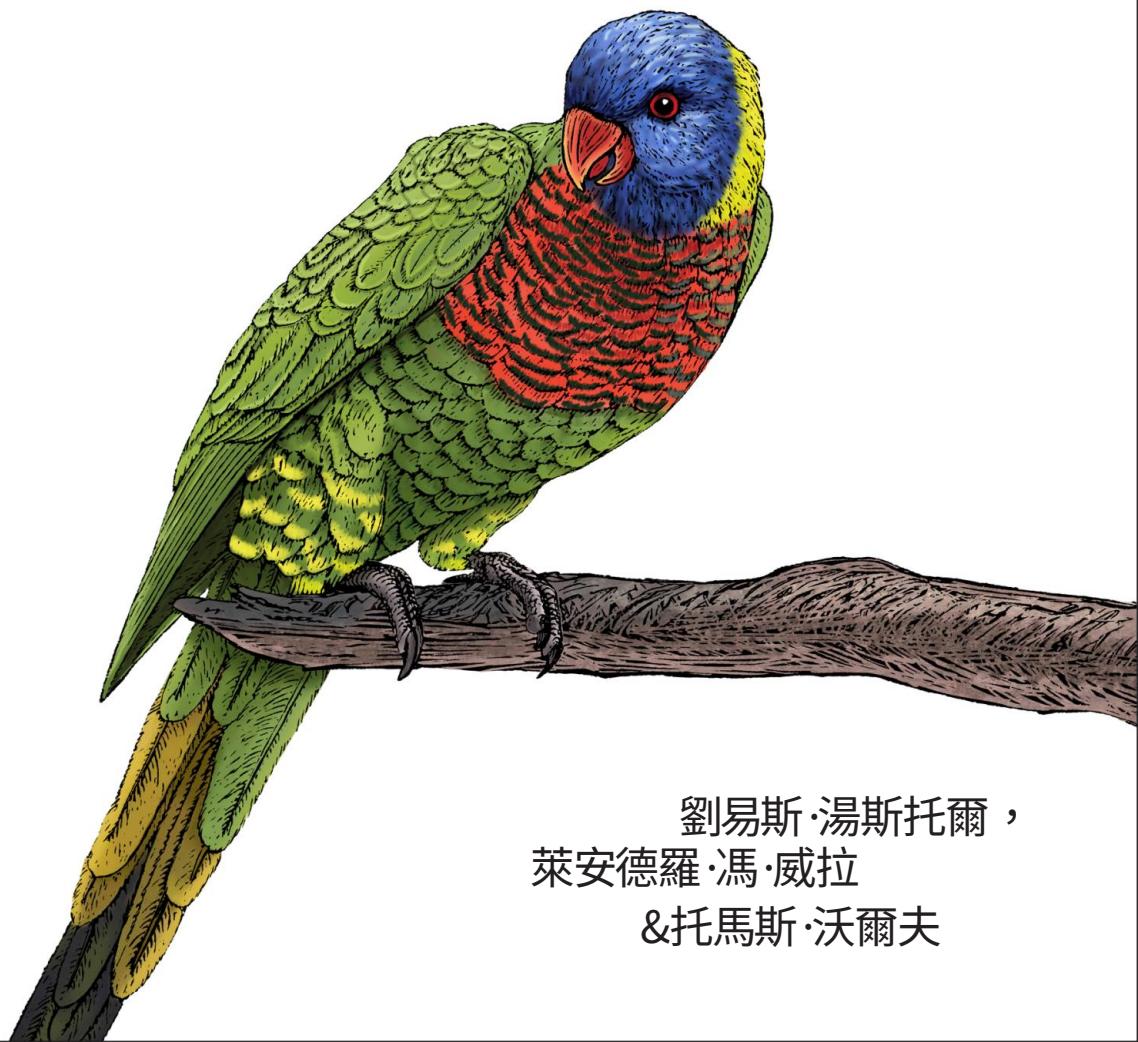


O'REILLY®

自然語言 處理與 變形金剛

用擁抱的臉構建語言應用程
序



劉易斯·湯斯托爾，
萊安德羅·馮·威拉
&托馬斯·沃爾夫



Natural Language Processing with Transformers

Since their introduction in 2017, transformers have quickly become the dominant architecture for achieving state-of-the-art results on a variety of natural language processing tasks. If you're a data scientist or coder, this practical book shows you how to train and scale these large models using Hugging Face Transformers, a Python-based deep learning library.

Transformers have been used to write realistic news stories, improve Google Search queries, and even create chatbots that tell corny jokes. In this guide, authors Lewis Tunstall, Leandro von Werra, and Thomas Wolf, among the creators of Hugging Face Transformers, use a hands-on approach to teach you how transformers work and how to integrate them in your applications. You'll quickly learn a variety of tasks they can help you solve.

- Build, debug, and optimize transformer models for core NLP tasks, such as text classification, named entity recognition, and question answering
- Learn how transformers can be used for cross-lingual transfer learning
- Apply transformers in real-world scenarios where labeled data is scarce
- Make transformer models efficient for deployment using techniques such as distillation, pruning, and quantization
- Train transformers from scratch and learn how to scale to multiple GPUs and distributed environments

"The preeminent book for the preeminent transformers library—a model of clarity!"

—Jeremy Howard
Cofounder of fast.ai and professor at University of Queensland

"A wonderfully clear and incisive guide to modern NLP's most essential library. Recommended!"

—Christopher Manning
Thomas M. Siebel Professor in Machine Learning, Stanford University

Lewis Tunstall is a machine learning engineer at Hugging Face. His current work focuses on developing tools for the NLP community and teaching people to use them effectively.

Leandro von Werra is a machine learning engineer in the open source team at Hugging Face, where he primarily works on code generation models and community outreach.

Thomas Wolf is chief science officer and cofounder of Hugging Face. His team is on a mission to catalyze and democratize AI research.

MACHINE LEARNING

US \$59.99 CAN \$79.99
ISBN: 978-1-098-10324-8
 5 5 9 9 9
9 781098 103248

Twitter: @oreillymedia
[linkedin.com/company/oreilly-media](https://www.linkedin.com/company/oreilly-media)
[youtube.com/oreillymedia](https://www.youtube.com/oreillymedia)

用 Transformers 進行自然語言處理的讚譽

預訓練的 transformer 語言模型席捲了 NLP 世界，而 Transformers 等庫使它們更易於使用。誰能比上述庫的創建者更好地教您如何利用 NLP 的最新突破？Natural Language

Processing with Transformers 是一部傑作，反映了其作者在工程和研究方面深厚的主題專業知識。這是一本難得的書，既提供了深刻的見解，又以通俗易懂的方式巧妙地將研究進展與現實世界的應用相結合。本書全面介紹了當前 NLP 中最重要的方法和應用，從多語言到高效模型，從問答到文本生成。每一章都基於豐富的代碼示例提供了細微的概述，突出了最佳實踐和實際考慮因素，並使您能夠將以研究為中心的模型用於有影響力的現實世界。無論您是 NLP 新手還是老手，本書都將提高您的理解力並加快您對最先進模型的開發和部署。

Sebastian Ruder ,谷歌 DeepMind

Transformers 改變了我們進行 NLP 的方式，而 Hugging Face 開創了我們在產品和研究中使用 Transformers 的方式。來自 Hugging Face 的 Lewis Tunstall、Leandro von Werra 和

Thomas Wolf 合著了一本及時的書，為這個關鍵主題提供了方便和實用的介紹。該書提供了變壓器力學的堅實概念基礎、變壓器動物園之旅、變壓器的應用以及培訓和將變壓器投入生產的實際問題。

閱讀了本書的章節後，憑藉其內容的深度和清晰的介紹，我相信這將成為任何有興趣學習轉換器（尤其是自然語言處理）的人的第一資源。

Delip Rao，《使用 PyTorch 進行自然語言處理和深度學習》一書的作者

複雜化簡單。這是一本關於 NLP、Transformer 以及圍繞它們不斷發展的生態系統 Hugging Face 的稀有而珍貴的書。無論這些對您來說仍然是流行語，還是您已經完全掌握了這一切，作者都將以幽默、科學嚴謹和大量代碼示例帶您進入最酷技術的最深層秘密。從“現成的預訓練”到“從頭開始定制”模型，從性能到缺失標籤問題，作者幾乎解決了 ML 工程師在現實生活中遇到的每一個難題，並提供了最先進的解決方案，使這本書注定要在未來幾年內決定該領域的標準。

埃森哲數據科學和機器學習副經理 Luca Perrozzi 博士

使用 Transformer 進行 自然語言處理

用擁抱的臉構建語言應用程序

Lewis Tunstall、Leandro von Werra 和 omas Wolf
Aurélien Géron 的前言

北京 波士頓 法納姆 塞瓦斯托波爾 東京

O'REILLY®

Lewis Tunstall、Leandro von Werra 和
Thomas Wolf 使用 Transformers 進行自然語言處理

版權所有 © 2022 Lewis Tunstall、Leandro von Werra 和 Thomas Wolf。版權所有。

在美利堅合眾國印刷。

由 O'Reilly Media, Inc. 出版 · 1005 Gravenstein Highway North, Sebastopol, CA 95472。

購買 O'Reilly 書籍可用於教育、商業或促銷用途。大多數書籍還提供在線版本(<http://oreilly.com>)。如需更多信息，請聯繫我們的企業/機構銷售部：800-998-9938 或 corporate@oreilly.com。

收購編輯：Rebecca Novack

索引器：Potomac Indexing, LLC

開發編輯：Melissa Potter

室內設計師：David Futato

製作編輯：Katherine Tozer

封面設計師：Karen Montgomery

文案編輯：Rachel Head

插畫家：Christa Lanz

校對：Kim Cofer

2022 年 2 月： 第一版

第一版的修訂歷史

2022-01-26 :首次發布

請參閱<http://oreilly.com/catalog/errata.csp?isbn=9781098103248>發布詳情。

O'Reilly 徽標是 O'Reilly Media, Inc. 的註冊商標。Natural Language Processing with Transformers 封面圖片和相關商業外觀是 O'Reilly Media, Inc. 的商標。

本作品中表達的觀點是作者的觀點，不代表出版商的觀點。

儘管出版商和作者已盡善意努力確保本作品中包含的信息和說明準確無誤，但出版商和作者不對錯誤或遺漏承擔任何責任，包括但不限於對因使用或對這項工作的依賴。使用本作品中包含的信息和說明的風險由您自行承擔。如果本作品包含或描述的任何代碼樣本或其他技術受開源許可或他人知識產權的約束，您有責任確保您對它們的使用符合此類許可和/或權利。

978-1-098-10324-8

目錄

前言。	習
前言。	十五
1.你好變形金剛。	18
編碼器-解碼器框架		2個
注意力機制		4個
NLP 中的遷移學習		6
擁抱面變形金剛 :彌合差距		9
變壓器應用之旅		10
文本分類		10
命名實體識別		11
問答		12
總結		13
翻譯		13
文本生成		14
抱臉生態系統		15
擁抱的臉集線器		16
擁抱面部標記器		17
擁抱面數據集		18
抱臉加速		18
變壓器的主要挑戰		19
結論		20
2.文本分類。	21
數據集		22
擁抱面孔數據集初探		23
從數據集到數據框		26

查看類別分佈 我們的推文有多長？	27
	28
從文本到標記	29
字符標記化	29
詞標記化	31
子詞分詞	33
標記整個數據集	35
訓練文本分類器	36
作為特徵提取器的變形金剛	38
微調變壓器	45
結論	54
 3. 變壓器剖析。	57
變壓器架構	57
編碼器	60
自註意力	61
前饋層	70
添加層歸一化	71
位置嵌入	73
添加分類頭	75
解碼器	76
認識變形金剛	78
變形金剛生命之樹	78
編碼器分支	79
解碼器分支	82
編碼器-解碼器分支	83
結論	84
 4. 多語言命名實體識別。	87
數據集	88
多語言變形金剛	92
仔細觀察代幣化	93
分詞器管道	94
SentencePiece 分詞器	95
用於命名實體識別的轉換器	96
Transformers 模型類剖析	98
身體和頭部	98
為令牌分類創建自定義模型	99
加載自定義模型	101
為 NER 標記文本	103
績效衡量	105
微調 XLM-RoBERTa	106

錯誤分析跨語言遷移零樣本遷移何時有意義？	108 115 116
一次微調多種語言與模型小部件的交互結論	118 121 121
5.文本生成。	123
生成連貫文本的挑戰 貪婪搜索解碼 波束搜索解碼 採樣方法	125
Top-k 和核採樣 哪種解碼方法最好？	127
	130
	134
	136
	140
結論	140
6.總結。	141
CNN/DailyMail 數據集	141
文本摘要管道	143
總結基線	143
GPT-2	144
T5	144
捷運	145
飛馬座	145
比較不同的摘要	146
衡量生成文本的質量	148
藍隊	148
胭脂	152
在 CNN/DailyMail 數據集上評估 PEGASUS	154
訓練摘要模型	157
在 SAMSum 上評估 PEGASUS	158
微調 PEGASUS	158
生成對話摘要	162
結論	163
7.問答。	165
建立基於評論的 QA 系統 數據集 從文本中提取答案 使用 Haystack 構建 QA 管道 改進我們的 QA 管道 評估檢索器	166 167 173 181 189 189

評估讀者	196
領域適應	199
評估整個 QA 流程	203
超越提取式 QA	205
結論	207
8.提高變壓器的生產效率。	209 210
意圖檢測作為案例研究	212
創建性能基準	217
通過知識蒸餾縮小模型	217
用於微調的知識蒸餾	220
預訓練的知識蒸餾	220
創建知識蒸餾培訓師	220
選擇一個好的學生初始化	222
使用 Optuna 尋找好的超參數	226
對我們的蒸餾模型進行基準測試	229
通過量化使模型更快	230
對我們的量化模型進行基準測試	236
使用 ONNX 和 ONNX 運行時優化推理	237
通過權重修剪使模型更稀疏	243
深度神經網絡中的稀疏性	244
權重修剪方法	244
結論	248
9.處理很少甚至沒有標籤。	249
構建 GitHub 問題標記器	251
獲取數據	252
準備數據	253
創建訓練集	257
創建訓練切片	259
實現樸素貝葉斯線	260
使用無標籤數據	263
使用幾個標籤	271
數據擴充	271
使用嵌入作為查找表	275
微調香草變壓器	284
帶提示的情境和小樣本學習	288
利用未標記的數據	289
微調語言模型	289
微調分類器	293
高級方法	295
結論	297

10.從頭開始訓練變形金剛。	299
大型數據集及其查找位置	300
構建大規模語料庫的挑戰	300
構建自定義代碼數據集	303
使用大型數據集	306
將數據集添加到 Hugging Face Hub	309
構建分詞器	310
分詞器模型	312
測量分詞器性能	312
Python 的分詞器	313
訓練分詞器	318
在集線器上保存自定義分詞器	322
從頭開始訓練模型	323
預訓練目標的故事	323
初始化模型	325
實現數據加載器	326
定義訓練循環	330
訓練跑	337
結果與分析	338
結論	343
11.未來方向。	345
Scaling Transformers	345
Scaling Laws 挑戰與	347
Scaling 注意！	348
稀疏注意力線性化注	351
意力	353
超越文本視覺表多模式	354
轉換器語音到文本視	355
覺和文本從這裡到哪	359
裡？	361
指數	361
	364
	370
指數。	371

前言

當你閱讀這些文字時，奇蹟正在發生：當它們在你的大腦皮層中導航時，這一頁上的波浪線正在轉化為文字、概念和情感。我從 2021 年 11 月開始的想法現在已經成功地侵入了你的大腦。如果他們設法引起你的注意並在這個嚴酷和競爭激烈的環境中生存足夠長的時間，當你與他人分享這些想法時，他們可能有機會再次繁殖。多虧了語言，思想已經成為空氣傳播的高度傳染性腦病菌，而且還沒有疫苗問世。

幸運的是，大多數腦部細菌都是無害的，¹還有一些非常有用。事實上，人類的大腦胚芽構成了我們最寶貴的兩項財富：知識和文化。就像沒有健康的腸道細菌我們無法正常消化一樣，沒有健康的腦細菌我們也無法正常思考。你的大部分想法實際上並不是你的：它們在感染你之前在許多其他大腦中出現、成長和進化。因此，如果我們想要製造智能機器，我們也需要找到感染它們的方法。

好消息是，在過去的幾年裡，另一個奇蹟正在發生：深度學習的幾項突破催生了強大的語言模型。

既然你在讀這本書，你可能已經看到了這些語言模型的一些驚人的演示，比如 GPT-3，給出一個簡短的提示，比如“青蛙遇到鱷魚”可以寫出一個完整的故事。儘管還不完全是莎士比亞，但有時很難相信這些文本是由人工神經網絡編寫的。事實上，GitHub 的 Copilot 系統正在幫助我寫下這些行：你永遠不會知道我真正寫了多少。

這場革命遠遠超出了文本生成。它涵蓋了自然語言處理 (NLP) 的整個領域，從文本分類到摘要、翻譯、問答、聊天機器人、自然語言理解 (NLU)，以及

¹有關大腦衛生技巧，請參閱 CGP Grey 關於模因的精彩視頻。

更多的。哪裡有語言、語音或文本，哪裡就有 NLP 的應用程序。您已經可以通過手機詢問明天的天氣，或者與虛擬服務台助理聊天以解決問題，或者從似乎真正理解您的查詢的搜索引擎中獲得有意義的結果。但這項技術太新了，最好的可能還在後頭。

與大多數科學進步一樣，NLP 最近的這場革命依賴於數百名無名英雄的辛勤工作。但其成功的三個關鍵因素確實站得住腳

出去：

5 年前

- transformer 是一種神經網絡架構，於2017年在一篇名為 “Attention Is All You Need”的開創性論文中提出，由一組谷歌研究人員發布。在短短幾年內，它席捲了整個領域。粉碎了以前通常基於遞歸神經網絡 (RNN) 的架構。Transformer 架構在捕獲長數據序列中的模式和處理龐大的數據集方面非常出色。以至於它的用途現在已經遠遠超出了 NLP，例如圖像處理任務。
- 在大多數項目中，您無法訪問龐大的數據集來從頭開始訓練模型。幸運的是，通常可以下載在通用數據集上預訓練的模型：然後您需要做的就是在您自己的（小得多的）數據集上對其進行微調。自 2010 年代初以來，預訓練一直是圖像處理的主流，但在 NLP 中，它僅限於無上下文詞嵌入（即單個詞的密集向量表示）。例如，“熊”這個詞在“泰迪熊”和“熊”中具有相同的預訓練嵌入。然後，在 2018 年，幾篇論文提出了可以針對各種 NLP 任務進行預訓練和微調的成熟語言模型；這完全改變了遊戲規則。

哈哈！！

- Hugging Face 等模型中心也改變了遊戲規則。在早期，預訓練的模型隨處可見，所以不容易找到你需要的東西。墨菲定律保證 PyTorch 用戶只能找到 TensorFlow 模型，反之亦然。當你找到一個模型時，弄清楚如何微調它並不總是那麼容易。這就是 Hugging Face 的 Transformers 庫的用武之地：它是開源的，它同時支持 TensorFlow 和 PyTorch，並且可以輕鬆地從 Hugging Face Hub 下載最先進的預訓練模型，為您的需求配置它任務，在你的數據集上對其進行微調，並對其進行評估。該庫的使用量增長迅速：在 2021 年第四季度，超過 5000 個組織使用它，每月使用 pip 安裝超過 400 萬次。此外，圖書館及其生態系統正在擴展到 NLP 之外：圖像處理模型也可用。您還可以從 Hub 下載大量數據集來訓練或評估您的模型。

© Hugging Face,
PyTorch > TensorFlow

那麼你還能要求什麼呢？嗯，這本書！它是由 Hugging Face 的開源開發人員編寫的，包括 Transformers 庫的創建者！而且它

顯示：您將在這些頁面中找到的信息的廣度和深度是驚人的。它涵蓋了從 Transformer 架構本身到 Transformers 庫以及圍繞它的整個生態系統的所有內容。我特別欣賞動手實踐的方法：您可以在 Jupyter notebooks 中跟進，所有代碼示例都直截了當且易於理解。作者在訓練非常大的 Transformer 模型方面擁有豐富的經驗，並且他們提供了豐富的提示和技巧來讓一切都有效地工作。最後但同樣重要的是，他們的寫作風格直接而生動：讀起來像小說。

Jupyter Lab, VS-code, + Copilot

簡而言之，我非常喜歡這本書，我相信你也會喜歡。任何對構建具有最先進語言處理功能的產品感興趣的人都需要閱讀它。它充滿了所有正確的大腦細菌！

Aurélien Géron
2021 年 11 月，新西蘭奧克蘭

前言

自 2017 年推出以來，Transformer 已成為學術界和工業界處理各種自然語言處理 (NLP) 任務的事實標準。不知不覺中，您今天可能與一個轉換器進行了交互：Google 現在使用 BERT 通過更好地理解用戶的搜索查詢來增強其搜索引擎。同樣，來自 OpenAI 的 GPT 模型系列因其生成類人文本和圖像的能力而多次成為主流媒體的頭條新聞。¹這些轉換器現在為 GitHub 的 Copilot、如圖 P-1 所示，它可以將評論轉換為源代碼，自動為你創建一個神經網絡！

那麼幾乎在一夜之間改變了這個領域的變形金剛是什麼呢？與許多偉大的科學突破一樣，它是當時研究界流行的多種思想的綜合，例如注意力、遷移學習和擴大神經網絡。

但是，無論它多麼有用，要在行業中獲得牽引力，任何奇特的新方法都需要工具來使其易於使用。變形金剛圖書館其周圍的生態系統通過讓從業者更容易使用、訓練和共享模型來響應這一號召。這極大地加速了 Transformer 的採用，該庫現在被 5000 多個組織使用。在本書中，我們將指導您如何針對實際應用訓練和優化這些模型。

¹ NLP 研究人員傾向於用芝麻街中的角色來命名他們的創作。我們將解釋所有這些第 1 章中的首字母縮略詞的含義。

```

1 # Create a convolutional neural network to classify MNIST images in PyTorch.
2 class ConvNet(nn.Module):
3     def __init__(self):
4         super(ConvNet, self).__init__()
5         self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
6         self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
7         self.conv2_drop = nn.Dropout2d()
8         self.fc1 = nn.Linear(320, 50)
9         self.fc2 = nn.Linear(50, 10)
10
11     def forward(self, x):
12         x = F.relu(F.max_pool2d(self.conv1(x), 2))
13         x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
14         x = x.view(-1, 320)
15         x = F.relu(self.fc1(x))
16         x = F.dropout(x, training=self.training)
17         x = self.fc2(x)
18         return F.log_softmax(x, dim=1)

```

圖 P-1 ◦來自 GitHub Copilot 的示例，其中給出了任務的簡要描述，應用程序為整個班級提供了建議（班級後的所有內容都是自動生成的）

這本書是給誰的？

本書是為數據科學家和機器學習工程師編寫的，他們可能聽說過最近涉及 Transformer 的突破，但缺乏深入的指南來幫助他們將這些模型適應自己的用例。這本書並不是機器學習的介紹，我們假設您熟悉 Python 編程並且對 PyTorch 等深度學習框架有基本的了解和張量流。我們還假設您有一些在 GPU 上訓練模型的實踐經驗。儘管本書重點介紹了 Transformers 的 PyTorch API，但第 2 章向您展示瞭如何將所有示例轉換為 TensorFlow。



以下資源為本書涵蓋的主題提供了良好的基礎。我們假設您的技術知識大致處於他們的水平：

- 使用 Scikit-Learn 和 TensorFlow 進行機器學習實踐，作者 :Aurélien Géron (O'Reilly)
- 程序員使用 fastai 和 PyTorch 進行深度學習，作者 :Jeremy Howard 和 Sylvain 古格 (O'Reilly)

- 使用 PyTorch 進行自然語言處理，作者 Delip Rao 和 Brian McMahan
(奧萊利)
- **擁抱臉課程**，由 Hugging Face 的開源團隊提供

你會學到什麼

本書的目標是使您能夠構建自己的語言應用程序。為此，它側重於實際用例，並僅在必要時深入研究理論。本書的風格是動手實踐，我們強烈建議您自己運行代碼示例進行實驗。

本書涵蓋了 Transformer 在 NLP 中的所有主要應用，每一章（除少數例外）專門針對一項任務，並結合了實際用例和數據集。每章還介紹了一些額外的概念。以下是我們將涵蓋的任務和主題的高級概述：

- **第1章**，你好變形金剛，介紹了變形金剛並將它們置於上下文中。它還介紹了 Hugging Face 生態系統。
- **第2章**，文本分類，側重於情感分析的任務（一個 common 文本分類問題）並介紹了 Trainer API。
- **第3章**，Transformer 剖析，更深入地探討 Transformer 架構，為後續章節做好準備。
- **第4章**，多語言命名實體識別，側重於識別多語言文本中的實體的任務（標記分類問題）。
- **第5章**，文本生成，探討了轉換器模型生成文本的能力。
- 吃文本，並介紹解碼策略和指標。
- **第6章**，總結，深入探討複雜的序列到序列任務
文本摘要並探索用於此任務的指標。
- **第7章**，問答，著重於構建基於評論的問題
問答系統，並介紹了使用 Haystack 進行檢索。
- **第8章**，使 Transformer 高效生產，重點介紹模型性能。我們將研究意圖檢測任務（一種序列分類問題）並探索知識蒸餾、量化和修剪等技術。
- **第9章**，處理很少或沒有標籤，研究在沒有大量標記數據的情況下提高模型性能的方法。我們將構建一個 GitHub 問題標記器並探索零樣本分類和數據擴充等技術。

第 10 章，從頭開始訓練 Transformer，向您展示如何從頭開始構建和訓練用於自動完成 Python 源代碼的模型。我們將研究數據集流和大規模訓練，並構建我們自己的分詞器。**第 11 章**，未來方向，探索變壓器面臨的挑戰以及該領域研究的一些令人興奮的新方向。

💡 Transformers 為使用和訓練 Transformer 模型提供了多個抽象層。我們將從易於使用的管道開始，這些管道允許我們通過模型傳遞文本示例，並僅用幾行代碼就可以研究預測。然後，我們將繼續介紹分詞器、模型類和 Trainer API，它們允許我們為自己的用例訓練模型。稍後，我們將向您展示如何用 Accelerate 庫替換 Trainer，這使我們能夠完全控制訓練循環，並允許我們完全從頭開始訓練大型變壓器！



雖然每一章大部分都是獨立的，但任務的難度在後面的章節中增加了。因此，我們建議先從第 1 章和第 2 章開始，然後再進入最感興趣的主題。

除了 🤖 變形金剛和 🤖 Accelerate，我們還將廣泛使用 Datasets，它與其他庫無縫集成。Datasets 提供與 Pandas 類似的數據處理功能，但其設計初衷是為了處理大型數據集和機器學習。

有了這些工具，您就擁有了應對幾乎所有 NLP 挑戰所需的一切！

軟件和硬件要求

由於本書的實踐方法，我們強烈建議您在閱讀每一章時運行代碼示例。由於我們正在處理變形金剛，因此您需要使用配備 NVIDIA GPU 的計算機來訓練這些模型。幸運的是，您可以使用幾個免費的在線選項，包括：

· 谷歌實驗室 · Kaggle 筆記

本 · Paperspace 漸變筆記

本

要運行這些示例，您需要遵循我們在本書的 GitHub 存儲庫中提供的安裝指南。您可以在 <https://github.com/nlp-with-transformers/notebooks> 找到本指南和代碼示例。



我們使用具有 16GB 內存的 NVIDIA Tesla P100 GPU 開發了大部分章節。一些免費平台提供的 GPU 內存較少，因此您可能需要在訓練模型時減小批量大小。

本書中使用的約定

本書使用以下排版約定：

斜體

表示新術語、URL、電子郵件地址、文件名和文件擴展名。

恆定寬度用於程

序列表，以及在段落中引用程序元素，例如變量或函數名稱、數據庫、數據類型、環境變量、語句和關鍵字。

等寬粗體顯示應由用

戶按字面輸入的命令或其他文本。

等寬斜體顯示應替換為

用戶提供的值或由上下文確定的值的文本。



此元素表示提示或建議。



該元素表示一般註釋。



該元素表示警告或警報。

使用代碼示例

補充材料（代碼示例、練習等）可從<https://github.com/nlp-with-transformers/notebooks>下載。

如果您在使用代碼示例時遇到技術問題或問題，請發送電子郵件至bookquestions@oreilly.com。

本書旨在幫助您完成工作。一般來說，如果本書提供了示例代碼，您可以在您的程序和文檔中使用它。你不

除非您要複制大部分代碼，否則需要聯繫我們以獲得許可。例如，編寫一個使用本書中幾段代碼的程序不需要許可。銷售或分發 O'Reilly 圖書中的示例需要獲得許可。通過引用本書和引用示例代碼來回答問題不需要許可。將本書中的大量示例代碼合併到您的產品文檔中確實需要獲得許可。

我們讚賞但通常不需要署名。署名通常包括書名、作者、出版商和 ISBN。例如：Lewis Tunstall、Leandro von Werra 和 Thomas Wolf (O'Reilly) 的“使用 Transformers 進行自然語言處理”。版權所有 2022 Lewis Tunstall、Leandro von Werra 和 Thomas Wolf，978-1-098-10324-8。”

如果您覺得您對代碼示例的使用不屬於合理使用或上述許可範圍，請隨時通過 permissions@oreilly.com 與我們聯繫。

O'Reilly 在線學習

O'REILLY[®] 40 多年來，O'Reilly Media 提供了技術和業務培訓、知識和洞察力來幫助公司取得成功。

我們獨特的專家和創新者網絡通過書籍、文章和我們的在線學習平台分享他們的知識和專長。O'Reilly 的在線學習平台讓您可以按需訪問實時培訓課程、深度學習路徑、交互式編碼環境，以及來自 O'Reilly 和 200 多家其他出版商的大量文本和視頻。如需更多信息，請訪問 <http://oreilly.com>。

如何聯繫我們

請將有關本書的評論和問題發送給出版商：

奧萊利媒體公司
1005 Gravenstein Highway North
Sebastopol, CA 95472 800-998-9938
(美國或加拿大境內)707-829-0515 (國際或本地)
707-829-0104 (傳真)

我們有本書的網頁，其中列出了勘誤表、示例和任何其他信息。您可以通過 <https://oreil.ly/nlp-with-transformers> 訪問此頁面。

電子郵件bookquestions@oreilly.com對本書發表評論或提出技術問題。

有關我們的書籍和課程的新聞和信息，請訪問<http://oreilly.com>。

在 Facebook 上找到我們：<http://facebook.com/oreilly>

在 Twitter 上關注我們：<http://twitter.com/oreillymedia>

在 YouTube 上觀看我們：<http://youtube.com/oreillymedia>

致謝

如果沒有許多人的幫助，就不可能寫出一本關於機器學習中發展最快的領域之一的書。我們感謝優秀的 O'Reilly 團隊，尤其是 Melissa Potter、Rebecca Novack 和 Katherine Tozer 的支持和建議。這本書還得益於出色的審稿人，他們花了無數時間為我們提供了寶貴的反饋。我們特別感謝 Luca Perozzi、Hamel Husain、Shabie Iqbal、Umberto Lupo、Malte Pietsch、Timo Möller 和 Aurélien Géron 的詳細審閱。我們感謝 Branden Chan 在 [deepset](#) 感謝他幫助擴展 Haystack 庫以支持第 7 章中的用例。

本書精美的插圖歸功於神奇的 [克里斯塔 Lanz](#)，謝謝你讓這本書變得特別。我們也有幸得到整個擁抱臉團隊的支持。非常感謝 Quentin Lhoest 回答了無數關於數據集的問題，感謝 Lysandre Debut 在與 Hugging Face Hub 相關的所有方面提供的幫助，感謝 Sylvain Gugger 在 Accelerate 方面的幫助，感謝 Jöe Davison 在第 9 章中關於零拍攝學習。我們也感謝 Sidd Karamcheti 和整個 [Mistral](#) 團隊為 GPT-2 添加穩定性調整以使第 10 章成為可能。這本書完全是用 Jupyter Notebooks 編寫的，我們感謝 Jeremy Howard 和 Sylvain Gugger 創造了像 [fastdoc](#) 這樣令人愉快的工具這使這成為可能。

劉易斯

感謝 Sofia，感謝你一直以來的支持和鼓勵。沒有這兩者，本書就不會存在。經過長時間的寫作，我們終於可以再次享受週末了！

萊昂德羅

感謝 Janine，在這漫長的一年裡，有許多深夜和忙碌的周末，感謝您的耐心和鼓勵支持。

托馬斯

我要首先感謝 Lewis 和 Leandro 提出這本書的想法，並大力推動以如此精美且易於理解的格式製作它。我還要感謝所有 Hugging Face 團隊相信 AI 的使命是社區的努力，感謝整個 NLP/AI 社區與我們一起構建和使用我們在本書中描述的庫和研究。

比我們建造的東西更重要的是，我們所走的旅程才是真正重要的，我們有幸與今天成千上萬的社區成員和讀者一起走過這條路。衷心感謝大家。

第1章

你好變形金剛

2017 年 ,谷歌的研究人員發表了一篇論文 ,提出了一種用於序列建模的新型神經網絡架構 。
1這種架構被稱為 Transformer ,在機器翻譯任務上的性能優於遞歸神經網絡 (RNN) ,無論
是在翻譯質量還是訓練成本方面 .

與此同時 ,一種名為 ULMFiT 的有效遷移學習方法表明 ,在非常龐大且多樣化的語料庫上訓
練長短期記憶 (LSTM) 網絡可以生成最先進的文本分類器 ,且標籤數據很少 。2

這些進步是當今最著名的兩種轉換器的催化劑 :生成式預訓練轉換器 (GPT)3和轉換器雙向
編碼器表示 (BERT) 4通過將轉換器架構與無監督學習相結合 ,這些模型消除了需要從頭開
始訓練特定於任務的架構 ,並以顯著的優勢打破了 NLP 中的幾乎所有基準 。自 GPT 和
BERT 發布以來 ,出現了一大堆 Transformer 模型 ;圖 1-1 顯示了最突出條目的時間軸 。

1 A. Vaswani 等 , “Attention Is All You Need” , (2017) °這個標題太吸引人了 ,以至於不少於 50 篇後續論文在
他們的標題中包含了 “你需要的一切” !

2 J. Howard 和 S. Ruder , “文本分類的通用語言模型微調” , (2018) 。

3 A. Radford 等人 , “通過生成式預訓練改善語言理解” , (2018) 。

4 J. Devlin 等人 , “BERT :用於語言理解的深度雙向轉換器的預訓練” ,
(2018) 。

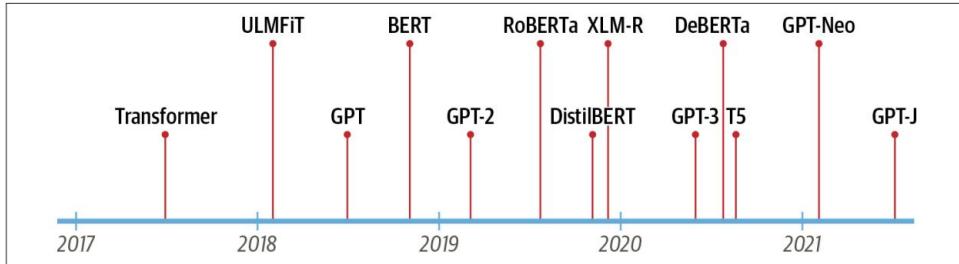


圖 1-1 變形金剛時間線

但我們已經超前了。要了解變形金剛的新穎之處，我們首先需要解釋一下：

- 編碼器-解碼器框架
- 注意機制
- 遷移學習

在本章中，我們將介紹 transformer 普遍存在的核心概念，瀏覽它們擅長的一些任務，並以工具和庫的 Hugging Face 生態系統作為結尾。

讓我們從探索編碼器-解碼器框架和轉換器興起之前的架構開始。

編碼器-解碼器框架

在 Transformer 出現之前，循環架構（例如 LSTM）是 NLP 中最先進的。這些體系結構在網絡連接中包含一個反饋迴路，允許信息從一個步驟傳播到另一個步驟，使它們非常適合對文本等順序數據進行建模。如圖 1-2 左側所示，RNN 接收一些輸入（可以是單詞或字符），將其饋送到網絡中，並輸出稱為隱藏狀態的向量。同時，模型通過反饋循環將一些信息反饋給自己，然後可以在下一步中使用這些信息。如果我們如圖 1-2 右側所示“展開”循環，則可以更清楚地看到這一點：RNN 將每一步的狀態信息傳遞給序列中的下一個操作。這允許 RNN 跟踪來自先前步驟的信息，並將其用於其輸出預測。

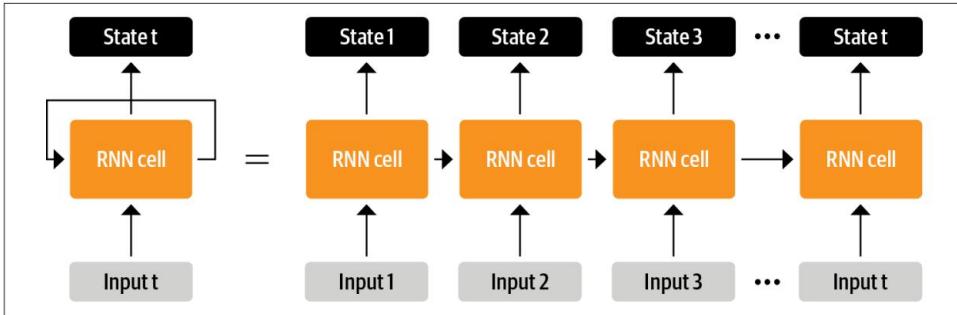


圖 1-2。及時展開 RNN

這些架構曾經（並將繼續）廣泛用於 NLP 任務、語音處理和時間序列。您可以在 Andrej Karpathy 的博客文章 [“遞歸神經網絡的不合理有效性”](#) 中找到對它們功能的精彩闡述。

RNN 發揮重要作用的一個領域是機器翻譯系統的開發，其目標是將一種語言的一系列單詞映射到另一種語言。此類任務通常使用編碼器-解碼器或序列到序列架構來解決，⁵這非常適合輸入和輸出均為任意長度序列的情況。編碼器的工作是將來自輸入序列的信息編碼成數字表示，通常稱為最後隱藏狀態。然後將該狀態傳遞給解碼器，解碼器生成輸出序列。

通常，編碼器和解碼器組件可以是任何一種可以對序列建模的神經網絡架構。圖 1-3 中的一對 RNN 說明了這一點，其中英文句子 “Transformers are great!” 被編碼為一個隱藏狀態向量，然後被解碼以產生德語翻譯 “Transformer sind grossartig！” 輸入詞按順序通過編碼器輸入，輸出詞從上到下一次生成一個。

⁵ I. Sutskever、O. Vinyals 和 QV Le， “[使用神經網絡進行序列到序列學習](#)”， (2014)。

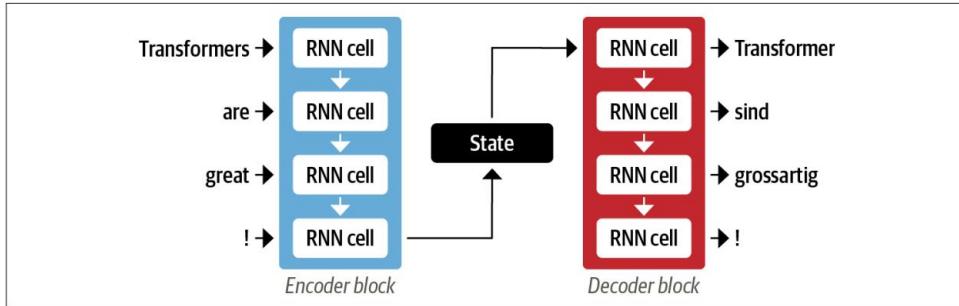


圖 1-3。具有一對 RNN 的編碼器-解碼器架構（通常，循環層比此處顯示的要多得多）

儘管簡潔優雅，但這種架構的一個弱點是解碼器的最終隱藏狀態造成了信息瓶頸：它必須代表整個輸入序列的含義，因為這是解碼器在生成輸出時可以訪問的所有內容。這對於長序列尤其具有挑戰性，因為在將所有內容壓縮為單個固定表示的過程中，序列開頭的信息可能會丟失。

幸運的是，有一種方法可以讓解碼器訪問編碼器的所有隱藏狀態，從而擺脫這個瓶頸。其通用機制稱為⁶，它是許多現代神經網絡架構中的關鍵組件。*attention*，了解 RNN 的主要構建塊 *attention* 讓我們來發展起來的。這將使我們更好地理解 Transformer 架構的

注意力機制

注意背後的主要思想是，編碼器不是為輸入序列生成單個隱藏狀態，而是在解碼器可以訪問的每個步驟輸出一個隱藏狀態。但是，同時使用所有狀態會給解碼器帶來巨大的輸入，因此需要某種機制來確定使用哪些狀態的優先級。這就是注意力的來源：它讓解碼器在每個解碼時間步為每個編碼器狀態分配不同的權重或“注意力”。這個過程如圖 1-4 所示，其中顯示了注意力在預測輸出序列中的第三個標記中的作用。

⁶ D. Bahdanau、K. Cho 和 Y. Bengio，“通過聯合學習對齊和翻譯進行神經機器翻譯”晚的”，(2014)。

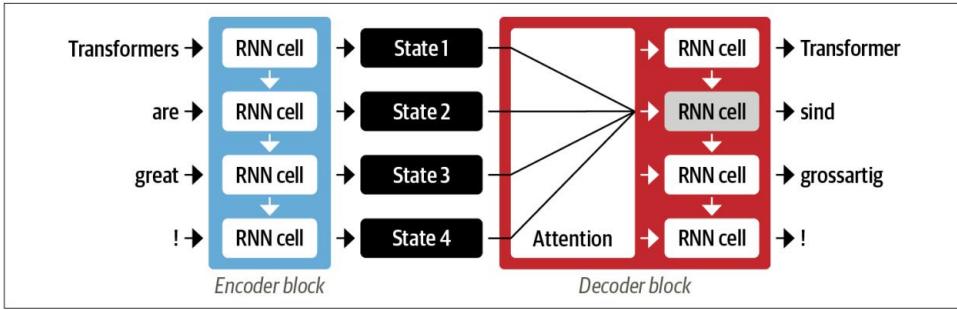


圖 1-4。具有一對注意機制的編碼器-解碼器架構

循環神經網絡

通過關注每個時間步最相關的輸入標記，這些基於注意力的模型能夠學習生成的翻譯中的單詞與源句子中的單詞之間的重要對齊。例如，[圖 1-5](#)可視化了英語到法語翻譯模型的注意力權重，其中每個像素表示一個權重。該圖顯示了解碼器如何能夠正確對齊單詞“zone”和“Area”，這兩個詞在兩種語言中的排序不同。

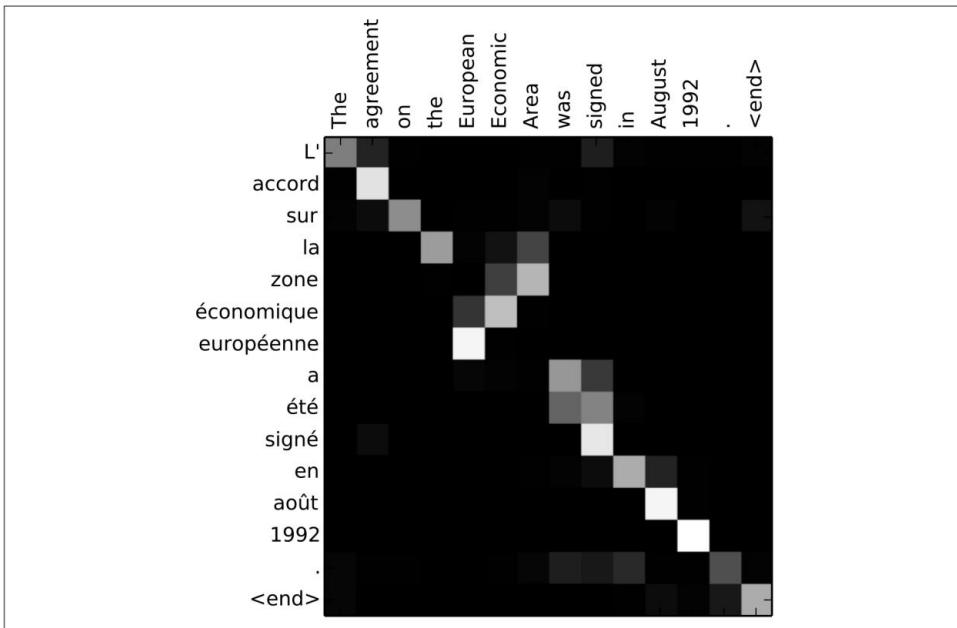


圖 1-5。RNN 編碼器-解碼器對齊英語單詞和生成的法語翻譯（由 Dzmitry Bahdanau 提供）

儘管注意力能夠產生更好的翻譯，但使用循環模型作為編碼器和解碼器仍然存在一個主要缺點：計算本質上是順序的，不能在輸入中並行化。

順序。

通過 Transformer，引入了一種新的建模範式：完全放棄重複，而是完全依賴一種稱為自註意力的特殊注意力形式。我們將在第 3 章中更詳細地介紹自註意力，但其基本思想是允許注意力對神經網絡同一層中的所有狀態進行操作。如圖 1-6 所示，編碼器和解碼器都有自己的自我注意機制，其輸出被饋送到前饋神經網絡（FF NN）。這種架構的訓練速度比循環模型快得多，並為 NLP 的許多近期突破鋪平了道路。

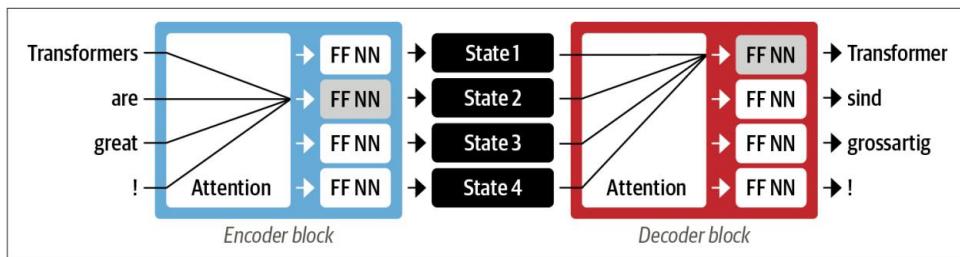


圖 1-6 原始 Transformer 的編碼器-解碼器架構

在最初的 Transformer 論文中，翻譯模型是在各種語言的大量句對語料庫上從頭開始訓練的。然而，在 NLP 的許多實際應用中，我們無法訪問大量帶標籤的文本數據來訓練我們的模型。缺少最後一塊來啟動變壓器革命：遷移學習。

NLP 中的遷移學習

現在計算機視覺中的常見做法是使用遷移學習在一項任務上訓練像 ResNet 這樣的捲積神經網絡，然後在新任務上對其進行調整或微調。這允許網絡利用從原始任務中學到的知識。在架構上，這涉及將模型分成身體和頭部，其中頭部是特定於任務的網絡。在訓練期間，身體的權重學習源域的廣泛特徵，這些權重用於為新任務初始化新模型。⁷ 與傳統的監督學習相比，這種方法通常會產生高質量的模型，可以接受更多培訓。

⁷ 權重是神經網絡的可學習參數。

有效地處理各種下游任務，並且標記數據少得多。圖 1-7 顯示了這兩種方法的比較。

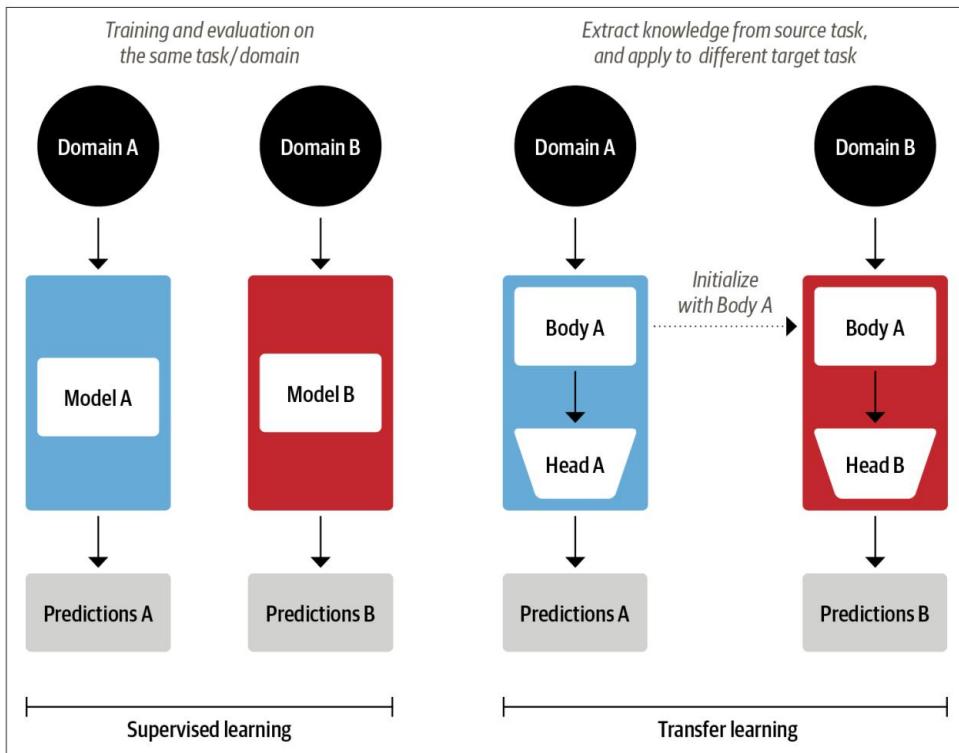


圖 1-7 傳統監督學習（左）與遷移學習（右）對比

在計算機視覺中，模型首先在 [ImageNet](#) 等大規模數據集上進行訓練，其中包含數百萬張圖片。這個過程稱為預訓練，其主要目的是教會模型圖像的基本特徵，例如邊緣或顏色。

然後可以在下游任務中對這些預訓練模型進行微調，例如使用相對較少的標記示例（通常每類數百個）對花卉物種進行分類。經過微調的模型通常比在相同數量的標記數據上從頭開始訓練的監督模型具有更高的準確性。

儘管遷移學習成為計算機視覺的標準方法，但多年來人們並不清楚 NLP 的類似預訓練過程是什麼。因此，NLP 應用程序通常需要大量標記數據才能實現高性能。即便如此，該性能也無法與視覺領域所取得的成績相提並論。

2017 年和 2018 年，多個研究小組提出了新方法，最終使遷移學習在 NLP 中發揮作用。它始於 OpenAI 研究人員的洞察力，他們通過使用從無監督預訓練中提取的特徵在情感分類任務上獲得了強大的性能。⁸隨後是 ULMFiT，它引入了一個通用框架來為各種任務調整預訓練的 LSTM 模型。⁹

如圖 1-8 所示，ULMFiT 包括三個主要步驟：

預訓練 最初

的訓練目標非常簡單：根據前面的詞預測下一個詞。此任務稱為語言建模。這種方法的優點在於不需要標記數據，並且可以利用來自維基百科等來源的大量可用文本。¹⁰

領域適應 一旦語言模型

型在大規模語料庫上進行了預訓練，下一步就是將其適應領域內語料庫（例如，從維基百科到電影評論的 IMDb 語料庫，如圖 1-8 所示）。這個階段仍然使用語言建模，但現在模型必須預測目標語料庫中的下一個單詞。

微調 在這一

步中，語言模型使用目標任務的分類層進行微調（例如，對圖 1-8 中的電影評論的情感進行分類）。

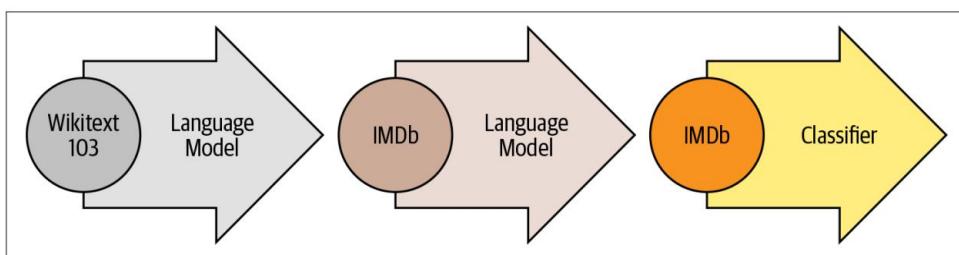


圖 1-8。 ULMFiT 過程（由 Jeremy Howard 提供）

通過在 NLP 中引入一個可行的預訓練和遷移學習框架，ULMFiT 提供了讓 transformers 騰飛的缺失部分。2018 年發布了兩款將自注意力與遷移學習相結合的 Transformer：

⁸ A. Radford、R. Jozefowicz 和 I. Sutskever， “學習生成評論和發現情緒”，
(2017)。

⁹ 此時的一項相關工作是 ELMo（來自語言模型的嵌入），它展示了如何預訓練
LSTM 可以為下游任務生成高質量的詞嵌入。

¹⁰ 與世界上大多數語言相比，英語更是如此，因為獲得大量數字化文本語料庫可能很困難。尋找彌合這一差距的方法是 NLP 研究和行動主義的一個活躍領域。

GPT

僅使用 Transformer 架構的解碼器部分，以及與 ULMFiT 相同的語言建模方法。 GPT 在 BookCorpus 11 上進行了預訓練，其中包含 7,000 本未出版的書籍，涉及各種類型，包括冒險、奇幻和浪漫。

BERT

使用 Transformer 架構的編碼器部分，以及一種特殊形式的語言建模，稱為屏蔽語言建模。掩蔽語言建模的目標是預測文本中隨機掩蔽的單詞。例如，給出這樣的句子“我看著我的[MASK]並看到[MASK]遲到了。”該模型需要預測由 [MASK] 表示的掩碼詞的最可能候選者。 BERT 在 BookCorpus 和英文維基百科上進行了預訓練。

GPT 和 BERT 在各種 NLP 基準測試中樹立了新的技術水平，並開創了變形金剛時代。

然而，隨著不同的研究實驗室在不兼容的框架（PyTorch 或 TensorFlow）中發布他們的模型，NLP 從業者將這些模型移植到他們自己的應用程序並不總是那麼容易。隨著變形金剛的上映，逐步構建了跨越 50 多個架構的統一 API。這個庫促進了對 transformer 研究的爆炸式增長，並迅速滲透到 NLP 從業者中，使得將這些模型輕鬆集成到當今許多現實生活中的應用程序變得容易。我們來看一下！

擁抱面變形金剛：彌合差距

將新穎的機器學習架構應用於新任務可能是一項複雜的工作，通常涉及以下步驟：

1. 在代碼中實現模型架構，通常基於 PyTorch 或張量流。
2. 從服務器加載預訓練權重（如果可用）。
3. 預處理輸入，將它們傳遞給模型，並應用一些特定於任務的後處理。
4. 實現數據加載器並定義損失函數和優化器來訓練模型。

11 Y. Zhu 等人，“將書籍和電影結合起來：通過看電影和閱讀書籍來實現故事般的視覺解釋”，(2015)。

這些步驟中的每一個都需要為每個模型和任務定製邏輯。傳統上（但並非總是如此！），當研究小組發表新文章時，他們還會發布代碼和模型權重。但是，此代碼很少標準化，並且通常需要數天的工程設計才能適應新的用例。

這就是變形金剛來拯救 NLP 從業者的地方！它為各種變壓器模型以及使這些模型適應新用例的代碼和工具提供了標準化接口。該庫目前支持三大深度學習框架（PyTorch、TensorFlow 和 JAX），並允許您在它們之間輕鬆切換。此外，它還提供特定於任務的頭部，因此您可以輕鬆地在文本分類、命名實體識別和問答等下游任務上微調轉換器。這將實踐者訓練和測試少數模型所需的時間從一周減少到一個下午！

您將在下一節中親眼看到這一點，在該節中，我們將展示僅需幾行代碼，就可以應用 Transformer 來處理您可能在野外遇到的一些最常見的 NLP 應用程序。

變壓器應用之旅

每個 NLP 任務都以一段文本開始，例如以下關於某個在線訂單的虛構客戶反饋：

```
text =     親愛的亞馬遜，上週我從你們在德國的在線商店訂購了一個擎天柱可動人偶。不幸的是，當我打開包裹時，我驚恐地發現我收到的是威震天可動人偶！作為霸天虎的終生敵人，我希望你能理解我的困境。為了解決這個問題，我要求用威震天換取我訂購的擎天柱人偶。隨函附上我關於此次購買的記錄副本。我希望收到你的來信很快。真誠的，大黃蜂。
```

根據您的應用程序，您正在處理的文本可能是法律合同、產品描述或其他完全不同的東西。對於客戶反饋，您可能想知道反饋是正面的還是負面的。

這個任務稱為情感分析，是我們將在第2章探討的更廣泛的文本分類主題的一部分。現在，讓我們看看如何使用 Transformers 從我們的文本片段中提取情感。



文本分類

正如我們將在後面的章節中看到的，Transformers 有一個分層的 API，允許您在不同的抽象級別與庫進行交互。在本章中，我們將從管道開始，它抽像出將原始文本轉換為來自微調模型的一組預測所需的所有步驟。

在 Transformers 中，我們通過調用 pipeline() 函數並提供我們感興趣的任務名稱來實例化管道：

[從變壓器導入管道](#)

分類器-管道（“文本分類”）

第一次運行此代碼時，您會看到一些進度條出現，因為管道會自動從Hugging Face Hub 下載模型權重。

第二次實例化管道時，庫會注意到您已經下載了權重，並將改用緩存版本。默認情況下，文本分類管道使用專為情感分析設計的模型，但它也支持多類和多標籤分類。

現在我們有了管道，讓我們生成一些預測！每個管道將文本字符串（或字符串列表）作為輸入並返回預測列表。每個預測都是一個 Python 字典，因此我們可以使用 Pandas 將它們很好地顯示為

數據框：

將熊貓導入為pd

輸出=分類器（文本） pd.DataFrame
(輸出)

	標籤	分數
0	負	0.901546

在這種情況下，模型非常確信文本具有負面情緒，考慮到我們正在處理來自憤怒客戶的投訴，這是有道理的！請注意，對於情緒分析任務，管道僅返回POSITIVE或NEGATIVE標籤之一，因為另一個可以通過計算1-score 來推斷。

現在讓我們看一下另一個常見任務，識別文本中的命名實體。

命名實體識別預測客戶反饋的情緒

是很好的第一步，但您通常想知道反饋是否與特定項目或服務有關。在 NLP 中，像產品、地點和人這樣的真實世界對像被稱為命名實體，從文本中提取它們被稱為命名實體識別 (NER)。我們可以通過加載相應的管道並將我們的客戶評論反饋給它來應用 NER：

```
ner_tagger = pipeline( ner , aggregation_strategy= simple )輸出=
ner_tagger(text) pd.DataFrame(outputs)
```

	entity_group	分數	單詞	開始	結束
ORG	0.879010	亞馬遜		=	11
1雜項	0.990859	擎天柱 36			49

	entity_group	分數	單詞	開始	結束
LOC	0.999755 德國			90	97
3雜項	0.556569 兆			208	212
4元	0.590256 ##創			212	216
5組織	0.669692 欺騙			253	259
6雜項	0.498350 ##圖標			259	264
7雜項	0.775361 威震天			350	358
8雜項	0.987854 霸天柱 367 380				
9元	0.812096 大黃蜂			502	511

您可以看到管道檢測到所有實體，並且還為每個實體分配了一個類別，例如ORG（組織）、LOC（位置）或PER（人）。這裡我們使用aggregation_strategy參數根據模型的預測對單詞進行分組。例如，實體“Optimus Prime”由兩個詞組成，但被分配了一個類別：MISC（雜項）。分數告訴我們模型對其識別的實體的信心程度。我們可以看到它是最不可靠的。

關於“霸天虎”和“威震天”的首次出現，它未能將這兩者歸為一個整體。



看到上表中單詞列中那些奇怪的哈希符號(#)了嗎？這些由模型的標記器生成，它將單詞拆分為稱為標記的原子單元。您將在第2章中了解有關標記化的所有知識。

提取文本中的所有命名實體很好，但有時我們想提出更有針對性的問題。這是我們可以使用問答的地方。

問答在問答中，我們為模型

提供一段稱為上下文的文本，以及一個我們想要提取其答案的問題。然後模型返回與答案對應的文本範圍。讓我們看看當我們詢問有關客戶反饋的具體問題時得到的結果：

```
reader = pipeline('question-answering') question
= 客戶想要什麼？ 輸出=讀者（問題=問題，上下文=文本）
pd.DataFrame ([輸出])
```

	分數	開始	結束	答案
0	0.631291 335 358	一次交換		威震天

我們可以看到，除了答案之外，管道還返回了開始和結束整數，這些整數對應於找到答案範圍的字符串索引（就像 NER 標記一樣）。我們將在第 7 章中研究多種類型的問答，但這種特殊類型稱為提取式問答，因為答案是直接從文本中提取的。

使用這種方法，您可以從客戶的反饋中快速閱讀和提取相關信息。但是，如果您收到堆積如山的冗長抱怨，而您又沒有時間閱讀所有內容怎麼辦？讓我們看看摘要模型是否可以提供幫助！

總結

文本摘要的目標是將長文本作為輸入並生成包含所有相關事實的簡短版本。這是一項比之前的任務複雜得多的任務，因為它需要模型生成連貫的文本。在現在應該是熟悉的模式中，我們可以按如下方式實例化摘要管道：

```
summarizer = pipeline( summarization ) 輸出=
summarizer(text, max_length=45, clean_up_tokenization_spaces=True) print(outputs[0]
[ summary_text ])
```

大黃蜂從您在德國的在線商店訂購了擎天柱可動人偶。不幸的是，當我打開包裹時，我驚恐地發現我收到的是威震天的可動人偶。

這個總結還不錯！雖然部分原文被複製，但該模型能夠抓住問題的本質，並正確識別出“Bumblebee”（出現在最後）是投訴的作者。在此示例中，您還可以看到我們向管道傳遞了一些關鍵字參數，例如max_length和clean_up_tokenization_spaces；這些允許我們在運行時調整輸出。

但是，當您收到使用您不懂的語言的反饋時會發生什麼？
你可以使用谷歌翻譯，或者你可以使用你自己的轉換器來為你翻譯！

翻譯

與摘要一樣，翻譯是一項輸出由生成的文本組成的任務。
讓我們使用翻譯管道將英語文本翻譯成德語：

```
translator = pipeline( translation_en_to_de ,
model= Helsinki-NLP/opus-mt-en-de ) 輸出=
translator(text, clean_up_tokenization_spaces=True, min_length=100) print(outputs[0]
[ translation_text ])
```

請在亞馬遜購買，購買 Optimus Prime 可動人偶在德國最好的 Ihrem 網上商店購買。Leider, als ich das Paket öffnete, entdeckte ich zu meinem Entsetzen, dass ich stattdessen eine 可動人偶

Megatron geschickt worden 戰爭！ Als lebenslanger Feind der Decepticons, Ich hoffe, Sie können mein Dilemma verstehen。 Um das Problem zu lösen ,Ich fordere einen Austausch von Megatron für die Optimus Prime Figur habe ich bestellt。

Anbei sind Kopien meiner Aufzeichnungen über diesen Kauf。 Ich erwarte ,禿頭 von Ihnen zu hören。 Aufrichtig ,大黃蜂。

同樣，該模型產生了一個非常好的翻譯，它正確地使用了德語的正式代詞，比如“*Ihrem*”和“*Sie*”。在這裡，我們還展示了如何覆蓋管道中的默認模型以選擇最適合您的應用程序的模型。您可以在 Hugging Face Hub 上找到數千種語言對的模型。在我們退後一步並查看整個 Hugging Face 生態系統之前，讓我們檢查最後一個應用程序。

文本生成

假設您希望能夠通過訪問自動完成功能來更快地回答客戶反饋。使用文本生成模型，您可以按如下方式執行此操作：

```
generator = pipeline( text-generation ) response =
    親愛的大黃蜂 ,很遺憾得知您的訂單混淆了。 prompt = text + \n\n客服回復 :\n + response outputs =
generator(prompt, max_length=200) print(outputs[0][ generated_text ])
```

親愛的亞馬遜，上週我從你們在德國的在線商店訂購了一個擎天柱可動人偶。不幸的是，當我打開包裹時，我驚恐地發現我收到的是威震天的可動人偶！作為霸天虎的終生敵人，希望您能理解我的困境。為了解決這個問題，我要求用威震天換取我訂購的擎天柱人偶。

隨函附上我關於此次購買的記錄副本。我希望很快收到你的來信。真誠的，大黃蜂。

客服回復：親愛的大黃蜂，很抱歉得知您
的訂單弄錯了。訂單完全貼錯標籤，這在我們的在線商店中很常見，但我對此表示感謝，因為根據我從本網站和前一天的
客戶服務中了解到，您的訂單在我們看來是錯誤的，我們是在解決這個問題的過程中。我們可以向您保證，您的訂單

好吧，也許我們不想用這個完成來讓 Bumblebee 平靜下來，但你明白了總體思路。

既然您已經看到了 transformer 模型的幾個很酷的應用，您可能想知道訓練在哪裡進行。我們在本章中使用的所有模型都是公開可用的，並且已經針對手頭的任務進行了微調。然而，一般來說，您會希望根據自己的數據微調模型，在接下來的章節中您將學習如何做到這一點。

但訓練模型只是任何 NLP 項目的一小部分。能夠高效地處理數據、與同事分享結果並使您的工作可重現也是關鍵組成部分。幸運的是，Transformers 周圍環繞著一個由有用工具組成的大型生態系統，這些工具支持大部分現代機器學習工作流程。讓我們來看看。

抱臉生態系統

從 Transformers 開始的東西已經迅速發展成為一個由許多庫和工具組成的完整生態系統，可以加速你的 NLP 和機器學習項目。Hugging Face 生態系統主要由兩部分組成：庫家族和 Hub，如圖1-9 所示。庫提供代碼，而 Hub 提供預訓練模型權重、數據集、評估指標腳本等。在本節中，我們將簡要介紹各種組件。我們將跳過變形金剛，因為我們已經討論過它，並且在本書的整個過程中我們會看到更多。

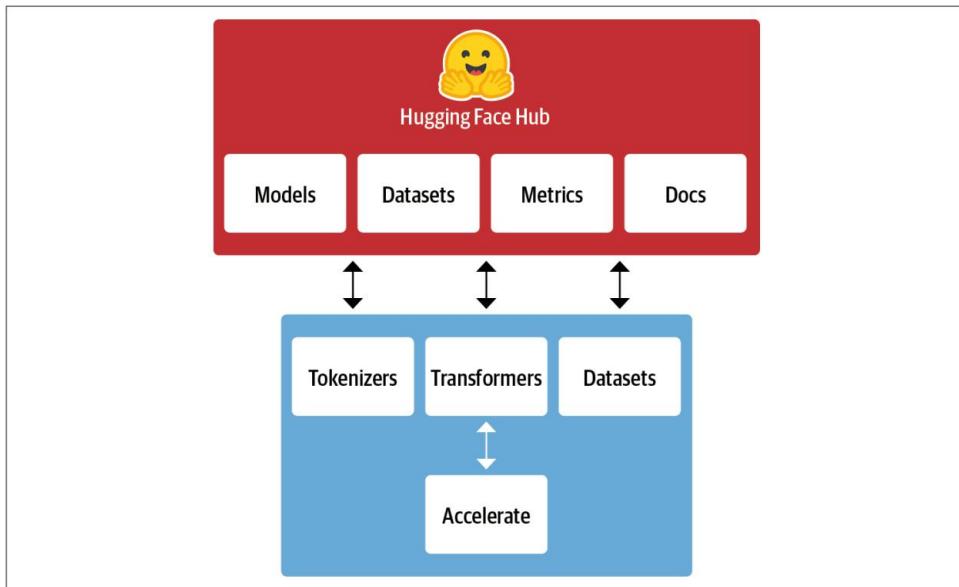


圖 1-9 抱臉生態系統概述

Hugging Face Hub

如前所述，遷移學習是推動 transformer 成功的關鍵因素之一，因為它可以為新任務重用預訓練模型。

因此，能夠快速加載預訓練模型並使用它們進行實驗至關重要。

Hugging Face Hub 擁有 20,000 多個免費模型。如圖 1-10 所示，有針對任務、框架、數據集等的過濾器，旨在幫助您瀏覽 Hub 并快速找到有前途的候選人。正如我們在管道中看到的那樣，在您的代碼中加載一個有前途的模型實際上只需要一行代碼。這使得嘗試各種模型變得簡單，並允許您專注於項目的特定領域部分。

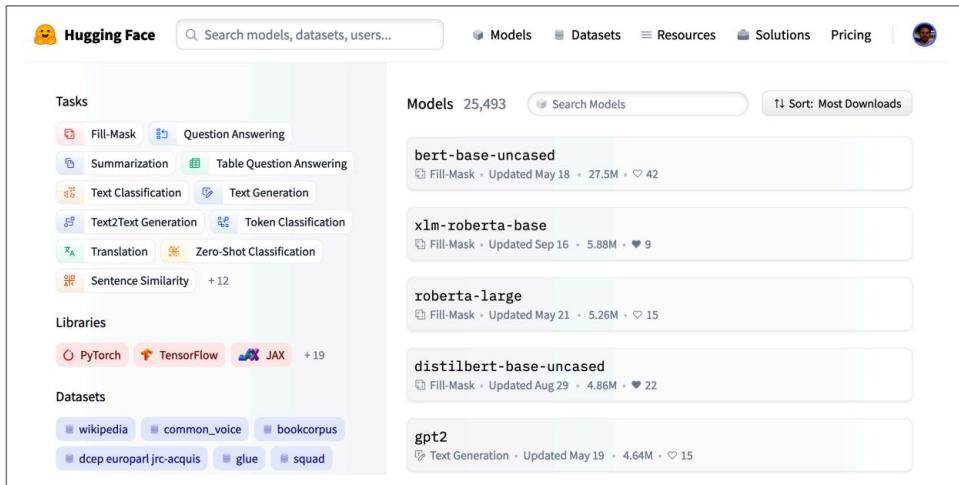


圖 1-10。Hugging Face Hub 的模型頁面，左側顯示過濾器，右側顯示模型列表

除了模型權重之外，該中心還託管用於計算指標的數據集和腳本，使您可以重現已發布的結果或為您的應用程序利用其他數據。

該中心還提供模型和數據集卡來記錄模型和數據集的內容，並幫助您就它們是否適合您做出明智的決定。Hub 最酷的功能之一是您可以直接通過各種特定於任務的交互式小部件來試用任何模型，如圖 1-11 所示。

BERT base model (uncased)

Pretrained model on English language using a masked language modeling (MLM) objective. It was introduced in [this paper](#) and first released in [this repository](#). This model is uncased: it does not make a difference between english and English.

Disclaimer: The team releasing BERT did not write a model card for this model so this model card has been written by the Hugging Face team.

Model description

BERT is a transformers model pretrained on a large corpus of English data in a self-supervised fashion. This means it was pretrained on the raw texts only, with no humans labelling them in any way (which is

Hosted inference API

NEW Select AutoNLP in the "Train" menu to fine-tune this model automatically.

Downloads last month
27,529,242

Fill-Mask Mask token: [MASK]

I looked at my [MASK] and saw I was la Compute

Computation time on cpu: cached

watch	0.228
phone	0.099
face	0.059
hands	0.053
hand	0.035

[JSON Output](#) Maximize

圖 1-11 來自 Hugging Face Hub 的示例模型卡，推理小部件，它允許您與模型交互，顯示在右側

讓我們繼續我們的 Tokenizers 之旅。



火炬和張量流還提供自己的集線器，如果特定模型或數據集在 Hugging Face Hub 上不可用，則值得一試。

擁抱面部分詞器我們在本章中看

到的每個管道示例背後都有一個分詞步驟，它將原始文本拆分為更小的片段，稱為分詞。我們將在第 2 章中詳細了解它是如何工作的，但現在只需了解標記可以是單詞、單詞的一部分，或者只是標點符號等字符就足夠了。Transformer 模型是根據這些標記的數字表示進行訓練的，因此正確執行此步驟對於整個 NLP 項目非常重要！

😊 分詞器提供了許多標記化策略，並且由於其 Rust 後端，它在文本標記化方面非常快。¹² 它還負責所有預處理和後處理步驟，例如規範化輸入和將模型輸出轉換為所需格式。使用 Tokenizers，我們可以像使用 Transformer 加載預訓練模型權重一樣加載分詞器。



¹² 生鏽是一種高性能的編程語言。

我們需要一個數據集和指標來訓練和評估模型，所以讓我們來看看
 負責該方面的數據集。

擁抱面部數據集加載、處理和

存儲數據集可能是一個繁瑣的過程，尤其是當數據集變得太大而無法容納筆記本電腦的 RAM 時。此外，您通常需要實現各種腳本來下載數據並將其轉換為標準格式。

 **數據集**通過為可以在集線器上找到的數千個數據集提供標準接口來簡化此過程。它還提供智能緩存（因此您不必在每次運行代碼時都重新進行預處理）並通過利用稱為內存映射的特殊機制避免 RAM 限制，該機制將文件內容存儲在虛擬內存中並啟用多個更有效地修改文件的過程。該庫還可以與 Pandas 和 NumPy 等流行框架進行互操作，因此您不必離開您最喜歡的數據整理工具的舒適環境。

然而，如果您不能可靠地衡量性能，擁有一個好的數據集和強大的模型是毫無價值的。不幸的是，經典的 NLP 指標有許多不同的實現，這些實現可能略有不同並導致具有欺騙性的結果。通過為許多指標提供腳本，Datasets 有助於提高實驗的可重現性和結果的可信度。



借助 Transformers、Tokenizers 和 Datasets 庫，我們擁有了訓練我們自己的轉換器模型所需的一切！然而，正如我們將在第 10 章中看到的那樣，在某些情況下我們需要對訓練循環進行細粒度控制。這就是生態系統的最後一個庫發揮作用的地方：Accelerate。



抱臉加速如果您曾經不得不在

PyTorch 中編寫自己的訓練腳本，那麼在嘗試將筆記本電腦上運行的代碼移植到組織集群上運行的代碼時，您可能會遇到一些麻煩。**加速**為你的正常訓練循環添加一個抽象層，負責處理訓練基礎設施所需的所有自定義邏輯。通過在必要時簡化基礎架構的更改，這從字面上加速了您的工作流程。

以上總結了Hugging Face開源生態系統的核心組成部分。但在結束本章之前，讓我們看一下在現實世界中嘗試部署轉換器時遇到的一些常見挑戰。

變壓器的主要挑戰

在本章中，我們大致了解了可以使用 Transformer 模型處理的各種 NLP 任務。閱讀媒體頭條，有時聽起來他們的能力是無限的。然而，儘管它們很有用，但變壓器遠非靈丹妙藥。以下是與它們相關的一些挑戰，我們將在整本書中探討這些挑戰：

語言

NLP 研究以英語為主。其他語言有多種模型，但很難找到稀有或低資源語言的預訓練模型。在第 4 章中，我們將探索多語言轉換器及其執行零樣本跨語言遷移的能力。

數據可用性 儘

管我們可以使用遷移學習來顯著減少模型所需的標記訓練數據量，但與人類執行任務所需的數據量相比仍然很多。第 9 章的主題是處理幾乎沒有標記數據的場景。

處理長文檔

Self-attention 在段落長的文本上效果非常好，但是當我們轉向像整個文檔這樣的較長文本時，它會變得非常昂貴。第 11 章討論了緩解這種情況的方法。

不透明

度與其他深度學習模型一樣，Transformer 在很大程度上是不透明的。很難或不可能解開模型做出特定預測的“原因”。當部署這些模型來做出關鍵決策時，這是一個特別艱鉅的挑戰。我們將在第 2 章和第 4 章探索一些方法來探測變壓器模型的錯誤。

Bias Transformer 模型主要使用來自互聯網的文本數據進行預訓練。

這會將數據中存在的所有偏差都烙印到模型中。確保這些既不是種族主義、性別歧視，也不是更糟的是一項具有挑戰性的任務。我們將在第 10 章中更詳細地討論其中一些問題。

儘管令人生畏，但其中許多挑戰是可以克服的。除了在提到的特定章節中，我們將在接下來的幾乎每一章中觸及這些主題。

結論

希望到現在為止，您已經興奮地學習如何開始訓練這些多功能模型並將其集成到您自己的應用程序中！您在本章中已經看到，只需幾行代碼，您就可以使用最先進的模型進行分類、命名實體識別、問答、翻譯和摘要，但這實際上只是“技巧”冰山。”

在接下來的章節中，您將學習如何使轉換器適應廣泛的用例，例如構建文本分類器或用於生產的輕量級模型，甚至從頭開始訓練語言模型。我們將採取動手實踐的方法，這意味著對於涵蓋的每個概念，都會有相應的代碼，您可以在 Google Colab 或您自己的 GPU 機器上運行這些代碼。

現在我們已經掌握了轉換器背後的基本概念，是時候開始我們的第一個應用程序了：文本分類。這就是下一章的主題！

第2章

文本分類

文本分類是 NLP 中最常見的任務之一 ;它可用於廣泛的應用程序 ,例如將客戶反饋標記為類別或根據他們的語言路由支持票 。很可能你的電子郵件程序的垃圾郵件過濾器正在使用文本分類來保護你的收件箱免受大量不需要的垃圾郵件的侵害 !

另一種常見的文本分類類型是情感分析 ,它 (如我們在第 1 章中所見) 旨在識別給定文本的極性 。例如 ,像特斯拉這樣的公司可能會分析圖 2-1 中的 Twitter 帖子 ,以確定人們是否喜歡它的新車頂。



圖 2-1 分析 Twitter 內容可以從客戶那裡得到有用的反饋 (由 Aditya Veluri 提供)

現在假設你是一名數據科學家，需要構建一個系統來自動識別人們在 Twitter 上表達的關於你公司產品的情緒狀態，例如“憤怒”或“喜悅”。在本章中，我們將使用 BERT 的一個變體 DistilBERT 來解決這個任務。這個模型的主要優點是它實現了與 BERT 相當的性能，同時體積更小，效率更高。這使我們能夠在幾分鐘內訓練一個分類器，如果您想訓練更大的 BERT 模型，您只需更改預訓練模型的檢查點即可。檢查點對應於加載到給定轉換器架構中的一組權重。

這也將是我們第一次接觸來自 Hugging Tokenizers 和 Transformers 的三個核心庫。如圖 2-2 中的微調模型。因此，本著 Optimus Prime² 的精神生讓我們學習數據集所要的轉換步驟並能訓練出好的

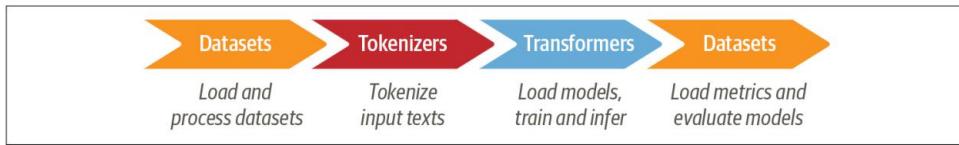


圖 2-2 使用 Tokenizers 和 Transformers 庫訓練轉換器模型的典型管道

數據集，

數據集

為了構建我們的情緒檢測器，我們將使用一篇文章中的一個很好的數據集³，該文章探討了英語 Twitter 消息中情緒是如何表達的。³ 與大多數僅涉及“積極”和“消極”極性的情緒分析數據集不同，該數據集包含六個基本情緒：憤怒、厭惡、恐懼、喜悅、悲傷和驚訝。給定一條推文，我們的任務是訓練一個模型，將其分類為其中一種情緒。

1 V. Sanh 等人，“DistilBERT：BERT 的精簡版：更小、更快、更便宜、更輕”，（2019）。

2 Optimus Prime 是廣受歡迎的兒童變形金剛系列中機器人種族的領導者（以及那些心地年輕的人！）。

3 E. Saravia 等人，“CARER：情緒識別的情境化情感表徵”，2018 年自然語言處理經驗方法會議論文集（2018 年 10 月至 11 月）：3687-3697，<http://dx.doi.org/10.18653/v1/D18-1404>。

抱臉數據集初探我們將使用數據集從抱臉中心下載

數據。我們可以使用list_datasets()函數來查看 Hub 上有哪些數據集可用：

[從數據集導入list_datasets](#)

```
all_datasets = list_datasets() print(f There
there are {len(all_datasets)} datasets currently available on the Hub ) print(f 前10個是 :{all_datasets[:10]} )
```

目前 Hub 上有 1753 個數據集

前 10 個是 :[acronym_identification , ade_corpus_v2 , adversarial_qa , aeslc ,
afrikaans_ner_corpus , ag_news , ai2_arc , air_dialogue , ajgt_twitter_ar ,
allegro_reviews]

我們看到每個數據集都有一個名稱，所以讓我們使用load_dataset()函數加載情緒數據集：

[從數據集導入load_dataset](#)

情緒= load_dataset (“情緒”)

如果我們看一下我們的情緒對象：

情緒

```
DatasetDict({ train:
    數據集({ features:
        [ text , label ], num_rows:
        16000
    })
    驗證 :數據集 ({特徵 :[ 文
        本 , 標籤 ] ,num_rows :2000
    })
    测试 :數據集 ({特
        徵 :[ 文本 , 標籤 ] ,num_rows :
        2000
    })
})
```

我們看到它類似於 Python 字典，每個鍵對應不同的拆分。我們可以使用通常的字典語法來訪問單個拆分：

train_ds =情緒[“火車”] train_ds

```
數據集
  ({ features:[ text , label ],
  num_rows:16000
})
```

它返回Dataset類的一個實例。 Dataset對像是 Datasets 中的核心數據結構之一，我們將在本書的整個過程中探索它的許多特性。對於初學者來說，它的行為就像一個普通的 Python 數組或列表，所以我們可以查詢它的長度：

```
len(train_ds)
```

```
16000
```

或通過其索引訪問單個示例：

```
train_ds[0]
```

```
{ label :0, text : 我沒有感到羞辱 }
```

這裡我們看到單行被表示為一個字典，其中的鍵對應於列名：

```
train_ds.column_names
```

```
[ 文本 , 標籤 ]
```

價值觀是推文和情感。這反映了 Datasets 基於Apache Arrow 的事實，它定義了一種類型化的柱狀格式，它比原生 Python 的內存效率更高。我們可以通過訪問Dataset對象的features屬性來查看底層使用了哪些數據類型：

```
打印 (train_ds.features)
```

```
{ text :Value(dtype= string ,id=None), label :ClassLabel(num_classes=6,names=[ 悲傷 ,  
喜悅 , 愛 , 憤怒 , 恐懼 , 驚喜 ],names_file=None,id=None)}
```

在這種情況下，文本列的數據類型是字符串，而標籤列是一個特殊的ClassLabel對象，它包含有關類名及其到整數的映射的信息。我們還可以使用切片訪問多行：

```
打印 (train_ds[ :5])
```

```
{ text :[ 我沒有感到羞辱 , 我可以從感到如此絕望到如此該死的充滿希望只是因為身邊有一個關心和清醒的人身  
邊 , 我抓緊時間發帖我覺得貪婪錯了 , “我一直對壁爐懷舊，我知道它還在酒店裡”，“我很不高興” ] , “標  
籤” :[0, 0, 3, 2, 3]}
```

請注意，在這種情況下，字典值現在是列表而不是單個元素。我們還可以通過名稱獲取完整的列：

```
打印 (train_ds[ “文本” ][:5])
```

```
[ 我沒有感到羞辱 , 我可以從感到如此絕望到如此該死的充滿希望只是因為在一個關心和清醒的人身  
邊 , 我抓緊時間發帖我覺得貪婪錯了 , 我一直覺得對壁爐懷舊，我會知道它仍然在財產上”，“我感到不高  
興” ]
```

現在我們已經了解瞭如何通過檢查推文的內容來加載和檢查數 據。 😊 數據集，讓我們做一些

如果我的數據集不在集線器上怎麼辦？

我們將使用 Hugging Face Hub 下載本書中大多數示例的數據集。但在許多情況下，您會發現自己處理的數據要么存儲在筆記本電腦上，要么存儲在組織中的遠程服務器上。

Datasets 提供了幾個加載腳本來處理本地和遠程數據集。表 2-1 中顯示了最常見數據格式的示例。

表 2-1。如何加載各種格式的數據集

數據格式加載腳本示例

CSV文件	格式文件	load_dataset(csv ,data_files= my_file.csv)
文本	文本	load_dataset(文本 ,data_files= my_file.txt)
JSON	JSON	load_dataset(json ,data_files= my_file.jsonl)

如您所見，對於每種數據格式，我們只需要將相關的加載腳本傳遞給 `load_dataset()` 函數，以及指定一個或多個文件的路徑或 URL 的 `data_files` 參數。例如，情感數據集的源文件實際上託管在 Dropbox 上，因此加載數據集的另一種方法是先下載其中一個拆分：

```
dataset_url = https://www.dropbox.com/s/1pzkadrvffbqw6o/train.txt !wget {dataset_url}
```

如果你想知道為什麼有一個 `!` 前面的 shell 命令中的字符，那是因為我們正在 Jupyter 筆記本中運行這些命令。如果您想在終端中下載和解壓縮數據集，只需刪除前綴即可。現在，如果我們查看 `train.txt` 文件的第一行：

```
!head -n 1火車.txt
```

我沒有感到羞辱；悲傷

我們可以看到這裡沒有列標題，每條推文和情感都用分號分隔。儘管如此，這與 CSV 文件非常相似，因此我們可以通過使用 csv 腳本並將 `data_files` 參數指向 `train.txt` 文件來在本地加載數據集：

```
emotions_local = load_dataset( csv ,data_files= train.txt ,sep= ; ,  
                               name=[“文本”，“標籤”])
```

這裡我們還指定了分隔符的類型和列的名稱。一種更簡單的方法是將 `data_files` 參數指向 URL 本身：

```
dataset_url = https://www.dropbox.com/s/1pkadrffbw6o/train.txt?dl=1 emotions_remote
= load_dataset( csv ,data_files=dataset_url,sep= ; ,  
名稱=[“文本”, “標籤”])
```

它將自動為您下載和緩存數據集。如您所見，load_dataset()函數非常通用。我們建議查看數據集文檔以獲得完整的概述。



從數據集到數據框

儘管 Datasets 提供了許多低級功能來對我們的數據進行切片和切塊，但將 Dataset 對象轉換為 Pandas DataFrame 通常很方便，因此我們可以訪問高級 API 以實現數據可視化。為了啟用轉換，Datasets 提供了一個set_format()方法，允許我們更改 Dataset 的輸出格式。請注意，這不會更改基礎數據格式（這是一個箭頭表），如果需要，您可以稍後切換到另一種格式：

[將熊貓導入為pd](#)

```
情緒.set_format(type= pandas ) df =
emotions[ train ][:] df.head()
```

	文本	標籤
0我沒有感到羞辱		0
1我可以從感到如此絕望到如此該死... 0 2我抓緊時間發帖我		
覺得貪婪錯了 3 3我一直對更換感到懷舊... 4我感到不高興		
		28
		28

如您所見，列標題已被保留，前幾行與我們之前的數據視圖相匹配。但是，標籤表示為整數，因此讓我們使用標籤功能的int2str()方法在我們的DataFrame中創建一個具有相應標籤名稱的新列：

```
def label_int2str(row): return
    emotions[ train ].features[ label ].int2str(row)

df[ label_name ] = df[ label ].apply(label_int2str) df.head()
```

	文本	標籤	標籤名稱
0我沒有感到羞辱	0		悲傷
1我可以從感到如此絕望到如此該死... 0			悲傷

	文本	標籤	標名稱
2我抓緊時間發帖 ,我覺得貪心錯了 3			憤怒
3我一直懷念更換... 4我感到不高興	28		愛
	28		憤怒

在深入構建分類器之前 ,讓我們仔細看看數據集 。正如 Andrej Karpathy 在他著名的博客文章 “訓練神經網絡的秘訣”中指出的那樣 ,成為 “數據中的一員”是訓練優秀模型的必要步驟 !

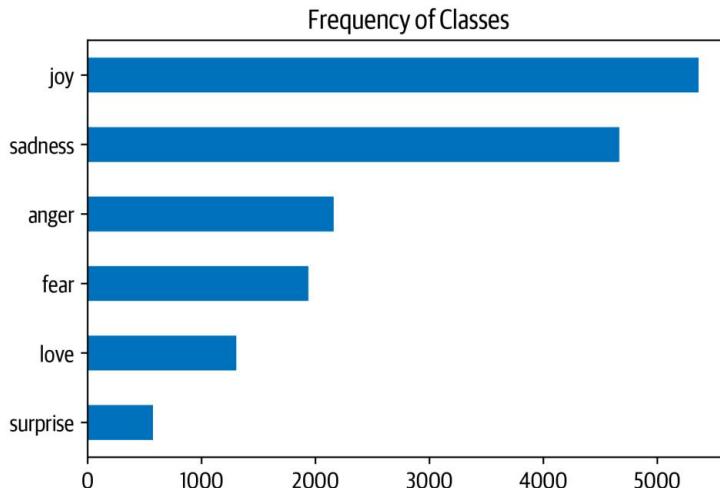
查看類分佈無論何時處理文本分類問題 ,檢

查樣本在類中的分佈是一個好主意 。具有偏斜類分佈的數據集在訓練損失和評估指標方面可能需要與平衡數據集不同的處理方式 。

借助 Pandas 和 Matplotlib ,我們可以快速可視化類別分佈 ,如下所示 :

將matplotlib.pyplot導入為plt

```
df[ label_name ].value_counts(ascending=True).plot.barh() plt.title( '類別  
頻率' ) plt.show()
```



在這種情況下 ,我們可以看到數據集嚴重不平衡 ;快樂和悲傷類出現頻率很高 ,而愛和驚喜類出現的頻率大約是它的 5-10 倍 。

有幾種方法可以處理不平衡數據 ,包括 :

- 對少數類進行隨機過採樣。 · 隨機對多數類進行欠採樣。 · 從代表性不足的類別中收集更多標記數據。

在本章中，為了簡單起見，我們將使用原始的、不平衡的類頻率。如果您想了解有關這些採樣技術的更多信息，我們建議您查看[Imbalanced-learn 庫](#)。只需確保在創建訓練/測試拆分之前不應用採樣方法，否則它們之間會出現大量洩漏！

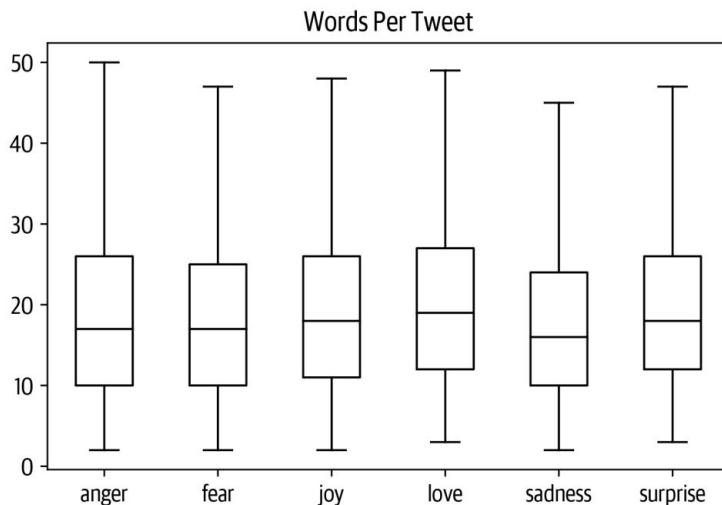
現在我們已經了解了這些類，讓我們來看看推文本身。

我們的推文有多長？

Transformer 模型具有最大輸入序列長度，稱為最大上下文大小。對於使用 DistilBERT 的應用程序，最大上下文大小為 512 個標記，相當於幾段文本。正如我們將在下一節中看到的，標記是一段原子文本；現在，我們將標記視為單個單詞。

通過查看每條推文的單詞分佈，我們可以粗略估計每種情緒的推文長度：

```
df[“每條推文的字數”] = df[“文本”].str.split().apply(len)
df.boxplot(“每條推文的字數”, by=“label_name”, grid=False,
           showfliers=False, color=“黑色”)
plt.suptitle(“”)
plt.xlabel(“”)
plt.show()
```



從圖中我們可以看出，對於每種情緒，大多數推文的長度都在 15 個單詞左右，最長的推文遠低於 DistilBERT 的最大上下文大小。比模型的上下文大小長的文本需要被截斷，如果截斷的文本包含關鍵信息，這可能會導致性能下降；在這種情況下，這似乎不是問題。

現在讓我們弄清楚如何將這些原始文本轉換為適合變形金剛的格式！當我們這樣做的時候，

讓我們也重置數據集的輸出格式

因為我們不再需要DataFrame格式了：

`情緒.reset_format()`

從文本到標記

像 DistilBERT 這樣的 Transformer 模型不能接收原始字符串作為輸入；相反，他們假設文本已被標記化並編碼為數字向量。標記化是將字符串分解為模型中使用的原子單位的步驟。可以採用多種標記化策略，並且通常從語料庫中學習將單詞最佳拆分為子單元。在查看用於 DistilBERT 的分詞器之前，讓我們考慮兩種極端情況：字符和單詞分詞。

字符標記化

最簡單的標記化方案是將每個字符單獨提供給模型。

在 Python 中，str 對象實際上是引擎蓋下的數組，這使我們能夠僅用一行代碼快速實現字符級標記化：

```
text = "標記文本是 NLP 的核心任務。 tokenized_text =  
list(text)打印(tokenized_text)
```

```
[ T , o , k , e , n , i , z , i , n , g , , t , e , x , t , , ,  
i , s , , a , , c , o , r , e , , t , a , s , k , , o ,  
f , , N , L , P , . ]
```

這是一個好的開始，但我們還沒有完成。我們的模型期望將每個字符轉換為整數，這個過程有時稱為數值化。一種簡單的方法是用唯一的整數對每個唯一的標記（在本例中是字符）進行編碼：

```
token2idx = {ch: idx for idx, ch in enumerate(sorted(set(tokenized_text)))} print(token2idx)  
  
{ ' ':0, '.':1, 'L':2, 'N':3, 'P':4, 'T':5, 'a':6, 'c':7, 'e':8, 'f':9, 'g':10, 'i':11,  
'k':12, 'n':13, 'o':14, 'r':15, 's':16, 't':17, 'x':18, 'z':19}
```

這為我們提供了從詞彙表中的每個字符到唯一整數的映射。

我們現在可以使用token2idx將標記化文本轉換為整數列表：

```
input_ids = [token2idx[token] for token in tokenized_text] print(input_ids)
```

```
[5, 14, 12, 8, 13, 11, 19, 11, 13, 10, 0, 17, 8, 18, 17, 0, 11, 16, 0, 6, 0, 7, 14, 15, 8, 0, 17, 6, 16, 12, 0, 14, 9, 0, 3, 2, 4, 1]
```

每個令牌現在都已映射到一個唯一的數字標識符（因此名稱為input_ids）。最後一步是將input_ids轉換為one-hot向量的二維張量。

單熱向量在機器學習中經常用於對分類數據進行編碼，這些數據可以是有序的也可以是名義的。例如，假設我們想要對變形金剛電視劇中的角色名稱進行編碼。一種方法是將每個名稱映射到一個唯一的ID，如下所示：

```
categorical_df = pd.DataFrame({ '名稱' :[ '大黃蜂', '擎天柱', '威震天' ], '標籤 ID' :[0,1,2]}) categorical_df
```

	姓名	標籤編號
0	大黃蜂	0
1	擎天柱	1
2	威震天	2

這種方法的問題在於它在名稱之間創建了一個虛構的順序，而神經網絡非常擅長學習這些類型的關係。

因此，我們可以為每個類別創建一個新列，並在類別為真時分配1，否則分配0。在Pandas中，這可以使用get_dummies()函數實現，如下所示：

```
pd.get_dummies(categorical_df[ '名稱' ])
```

	大黃蜂	威震天	擎天柱
0	1	0	0
1	0	1	0
2	0	0	1

這個DataFrame的行是單熱向量，它有一個“熱”條目，其他地方都是1和0。現在，看看我們的input_ids，我們有一個類似的問題：元素創建一個序數標度。這意味著添加或減去兩個ID是無意義的操作，因為結果是代表另一個隨機令牌的新ID。

另一方面，添加兩個單熱編碼的結果很容易解釋：“熱”的兩個條目表示相應的標記同時出現。

我們可以通過將input_ids轉換為張量並應用one_hot()函數來在PyTorch中創建單熱編碼，如下所示：

導入torch導入
torch.nn.functional作為F

```
input_ids = torch.tensor(input_ids)
one_hot_encodings = F.one_hot(input_ids, num_classes=len(token2idx)) one_hot_encodings.shape
```

火炬.Size([38, 20])

對於 38 個輸入標記中的每一個，我們現在都有一個 20 維的單熱向量，因為我們的詞彙表包含 20 個獨特的字符。



始終在one_hot()函數中設置num_classes很重要，否則 one-hot 向量最終可能會比詞彙表的長度短（並且需要手動用零填充）。在TensorFlow 中，等效函數是tf.one_hot()，其中深度參數扮演num_classes 的角色。

通過檢查第一個向量，我們可以驗證 1 出現在 `input_ids[0]` 指示的位置：

```
print(f Token: {tokenized_text[0]} )  
print(f Tensor index: {input_ids[0]} ) print(f One-  
hot: {one_hot_encodings[0]} )  
  
代幣 : T  
張量指數 : 5  
  
一熱 : 張量 ([0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

從我們的簡單示例中，我們可以看到字符級標記化會忽略文本中的任何結構，並將整個字符串視為字符流。雖然這有助於處理拼寫錯誤和生僻詞，但主要缺點是需要從數據中學習單詞等語言結構。這需要大量的計算、內存和數據。出於這個原因，字符標記化在實踐中很少使用。相反，在標記化步驟中保留了一些文本結構。

詞標記化是實現這一目標的一種直接方法，所以讓我們來看看它是如何工作的。

詞標記化

我們可以將其拆分為單詞並將每個單詞映射為一個整數，而不是將文本拆分為字符。從一開始就使用單詞可以使模型跳過從字符學習單詞的步驟，從而降低訓練的複雜性過程。

一類簡單的單詞分詞器使用空格來分詞文本。我們可以通過直接在原始文本上應用 Python 的 `split()`函數來做到這一點（就像我們測量推文長度一樣）：

```
tokenized_text = text.split()打印
(tokenized_text)

[ Tokenizing , text , is , a , core , task , of , NLP. ]
```

從這裡我們可以採取與字符分詞器相同的步驟將每個單詞映射到一個 ID。然而，我們已經可以看到這種標記化方案的一個潛在問題：沒有考慮標點符號，所以 NLP 被視為單個令牌。鑑於單詞可能包含偏角、變位或拼寫錯誤，詞彙量很容易增長到數百萬！



一些分詞器有額外的標點符號規則。還可以應用詞幹提取或詞形還原，將單詞標準化為它們的詞幹（例如，“great”、“greater”和“greatest”都變成“great”）。代價是丢失文本中的一些信息。

詞彙量大是一個問題，因為它要求神經網絡具有大量參數。為了說明這一點，假設我們有 100 萬個唯一單詞，並希望在神經網絡的第一層將 100 萬維輸入向量壓縮為 1000 維向量。這是大多數 NLP 架構中的標準步驟，第一層的權重矩陣將包含 $100 \times 1000 = 10$ 億個權重。這已經可以與最大的 GPT-2 模型相媲美，總共有大約 15 億個參數！

自然地，我們希望避免浪費我們的模型參數，因為模型的訓練成本很高，而且較大的模型更難維護。一種常見的方法是通過考慮語料庫中 100,000 個最常見的詞來限制詞彙量並丟棄稀有詞。不屬於詞彙表的詞被歸類為“未知”並映射到共享的 UNK 標記。這意味著我們在單詞標記化過程中丟失了一些潛在的重要信息，因為模型沒有關於與 UNK 相關的單詞的信息。

如果在保留所有輸入信息和部分輸入結構的字符和單詞標記化之間存在妥協，那不是很好嗎？有：子詞標記化。

⁴ GPT-2 是 GPT 的繼任者，它以其令人印象深刻的生成能力吸引了公眾的注意
現實的文字。我們將在第 6 章詳細探討 GPT-2。

子詞分詞

子詞標記化背後的基本思想是結合字符和單詞標記化的最佳方面。一方面，我們希望將稀有詞拆分成更小的單元，讓模型能夠處理複雜的詞和拼寫錯誤。另一方面，我們希望將頻繁出現的單詞保留為唯一實體，以便我們可以將輸入的長度保持在可管理的範圍內。子詞標記化（以及單詞標記化）的主要區別特徵是它是使用統計規則和算法的組合從預訓練語料庫中學習的。

有幾種 NLP 中常用的子詞分詞算法，但讓我們從BERT 和 DistilBERT 分詞器使用的 WordPiece 5 開始。

了解 WordPiece 工作原理的最簡單方法是實際查看它。Transformers 提供了一個方便的 AutoTokenizer 類，它允許您快速加載與預訓練模型關聯的分詞器。我們只需調用它的 from_pretrained() 方法，提供 Hub 上模型的 ID 或本地文件路徑。讓我們從為 DistilBERT 加載分詞器開始：

從變形金剛導入AutoTokenizer

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

AutoTokenizer 類屬於一組更大的“**自動**”類。其工作是從檢查點的名稱中自動檢索模型的配置、預訓練權重或詞彙表。這允許您在模型之間快速切換，但如果希望手動加載特定類，您也可以這樣做。例如，我們可以按如下方式加載 DistilBERT 分詞器：

從變形金剛導入DistilBertTokenizer

```
distilbert_tokenizer = DistilBertTokenizer.from_pretrained(model_ckpt)
```



當你第一次運行AutoTokenizer.from_pretrained()方法時，你會看到一個進度條，顯示預訓練分詞器的哪些參數是從 Hugging Face Hub 加載的。當您第二次運行代碼時，它會從緩存中加載分詞器，通常位於 `~/.cache/huggingface`。

讓我們通過簡單的“文本分詞是 NLP 的核心任務”來檢查這個分詞器是如何工作的。示例文本：

5 M. Schuster 和 K. Nakajima，“日語和韓語語音搜索”，2012 年 IEEE 聲學、語音和信號處理國際會議 (2012)：5149–5152。<https://doi.org/10.1109/ICASSP.2012.6289079>。

```
encoded_text = tokenizer(text)打印
(encoded_text)

{ input_ids :[101, 19204, 6026, 3793, 2003, 1037, 4563, 4708, 1997, 17953, 2361, 1012, 102],
attention_mask :[1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
```

與字符標記化一樣，我們可以看到單詞已映射到input_ids字段中的唯一整數。我們將在下一節討論attention_mask字段的作用。現在我們有了input_ids，我們可以使用標記器的convert_ids_to_tokens()方法將它們轉換回標記：

```
tokens = tokenizer.convert_ids_to_tokens(encoded_text.input_ids)打印(令牌)
```

```
[ [CLS] , token , ##izing , text , is , a , core , task , of , nl , ##p ,
。 , [九月] ]
```

我們可以在這裡觀察到三件事。首先，一些特殊的[CLS]和[SEP]標記已添加到序列的開頭和結尾。這些標記因模型而異，但它們的主要作用是指示序列的開始和結束。其次，每個標記都被小寫了，這是這個特定檢查點的一個特徵。

最後，我們可以看到“tokenizing”和“NLP”被拆分為兩個token，這是有道理的，因為它們不是常用詞。##izing和##p中的##前綴表示前面的字符串不是空格；當您將標記轉換回字符串時，任何帶有此前綴的標記都應與前一個標記合併。AutoTokenizer類有一個convert_tokens_to_string()方法可以做到這一點，所以讓我們將它應用到我們的令牌：

```
打印 (tokenizer.convert_tokens_to_string (令牌) )
```

[CLS] 標記文本是nlp的核心任務。[九月]

AutoTokenizer類還有幾個提供有關分詞器信息的屬性。例如，我們可以檢查詞彙表的大小：

```
tokenizer.vocab_size
```

30522

以及相應模型的最大上下文大小：

```
tokenizer.model_max_length
```

512

另一個需要了解的有趣屬性是模型在其前向傳遞中期望的字段名稱：

```
tokenizer.model_input_names
```

```
[ input_ids , attention_mask ]
```

現在我們對單個字符串的標記化過程有了基本的了解，讓我們看看如何對整個數據集進行標記化！



使用預訓練模型時，確保使用與訓練模型相同的分詞器非常重要。

從模型的角度來看，切換分詞器就像打亂詞彙表。如果你周圍的每個人都開始隨機交換“貓”之類的單詞，你也很難理解發生了什麼！

標記整個數據集為了標記整個語料庫，我

們將使用DatasetDict對象的map()方法。

在本書中我們會多次遇到這種方法，因為它提供了一種將處理函數應用於數據集中每個元素的便捷方法。我們很快就會看到，`map()`方法也可用於創建新的行和列。

首先，我們需要一個處理函數來標記我們的示例：

```
def tokenize(batch): return  
    tokenizer(batch[ text ], padding=True, truncation=True)
```

此函數將分詞器應用於一批示例；`padding=True`會將示例用零填充到一批中最長的示例的大小，而`truncation=True`會將示例截斷為模型的最大上下文大小。要查看`tokenize()`的實際效果，讓我們從訓練集中傳遞一批兩個示例：

打印 (標記化 (情緒[“火車”][:2]))

在這裡我們可以看到填充的結果：`input_ids`的第一個元素比第二個元素短，因此向該元素添加了零以使其長度相同。

這些零在詞彙表中有對應的[PAD]標記，特殊標記集還包括我們之前遇到的[CLS]和[SEP]標記：

特殊令牌	[PAD]	[UNK]	[CLS]	[SEP]	[MASK]
特殊令牌 ID 0		100	101	102	103

另請注意，除了將編碼後的推文作為input_id返回之外，分詞器還返回一個attention_mask數組列表。這是因為我們不希望模型被額外的填充標記混淆：注意掩碼允許模型忽略輸入的填充部分。[圖2-3](#)直觀地解釋瞭如何填充輸入ID和注意掩碼。

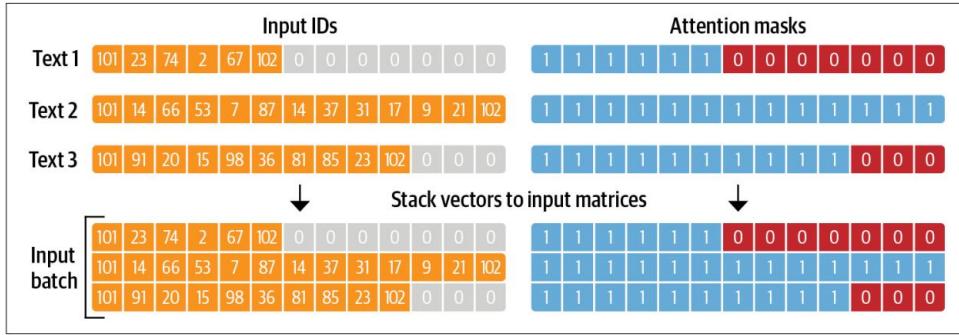


圖 2-3 °對於每個批次，輸入序列被填充到批次中的最大序列長度；注意掩碼在模型中用於忽略輸入張量的填充區域

一旦我們定義了一個處理函數，我們就可以用一行代碼將它應用於語料庫中的所有拆分：

```
emotions_encoded = emotions.map (標記化， batched=True， batch_size=None)
```

默認情況下，`map()`方法對語料庫中的每個示例單獨運行，因此設置`batched=True`將分批編碼推文。因為我們設置了`batch_size=None`，所以我們的`tokenize()`函數將作為單個批次應用於整個數據集。這確保了輸入張量和注意力掩碼具有全局相同的形狀，我們可以看到該操作為數據集添加了新的`input_ids`和`attention_mask`列：

```
打印 (情緒編碼[“火車”].column_names)
[ attention_mask , input_ids , label , text ]
```



在後面的章節中，我們將看到如何使用數據整理器動態填充每批中的張量。全局填充將在下一節派上用場，我們從整個語料庫中提取特徵矩陣。

訓練文本分類器

正如第 1 章中所討論的，像 DistilBERT 這樣的模型經過預訓練可以預測文本序列中的掩碼詞。但是，我們不能直接使用這些語言模型進行文本分類；我們需要稍微修改一下。為了理解需要進行哪些修改，讓我們看一下基於編碼器的模型（例如 DistilBERT）的架構，如圖 2-4 所示。

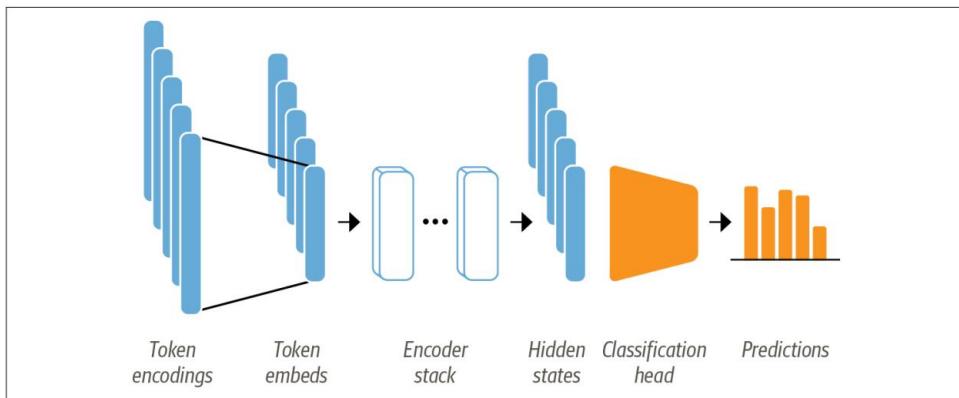


圖 2-4。使用基於編碼器的轉換器進行序列分類的架構；它由模型的預訓練主體和自定義分類頭組成

首先，文本被標記化並表示為稱為標記編碼的單熱向量。

tokenizer 詞彙表的大小決定了 token 編碼的維度，它通常由 20k-200k 個唯一的 token 組成。接下來，這些標記編碼被轉換為標記嵌入，它們是存在於低維空間中的向量。然後，令牌嵌入通過編碼器塊層傳遞，為每個輸入令牌生成隱藏狀態。對於語言建模的預訓練目標，⁶每個隱藏狀態都被饋送到預測屏蔽輸入標記的層。

對於分類任務，我們將語言建模層替換為分類層。



實際上，PyTorch 跳過了為標記編碼創建單熱向量的步驟，因為將矩陣與單熱向量相乘與從矩陣中選擇列相同。這可以通過直接從矩陣中獲取帶有令牌 ID 的列來完成。我們將在第 3 章使用 `nn.Embedding` 類時看到這一點。

我們有兩種選擇可以在我們的 Twitter 數據集上訓練這樣的模型：

特徵提取

我們使用隱藏狀態作為特徵，只在它們上訓練分類器，而不修改預訓練模型。

⁶在 DistilBERT 的情況下，它正在猜測屏蔽的標記。

微調我們端到
端地訓練整個模型，這也會更新預訓練模型的參數。

在以下部分中，我們將探討 DistilBERT 的兩個選項並檢查它們的權衡。

作為特徵提取器的變形金剛

使用轉換器作為特徵提取器非常簡單。如圖 2-5 所示，我們在訓練期間凍結身體的權重，並將隱藏狀態用作分類器的特徵。這種方法的優點是我們可以快速訓練一個小的或淺的模型。這樣的模型可以是神經分類層或不依賴梯度的方法，例如隨機森林。如果 GPU 不可用，此方法特別方便，因為隱藏狀態只需要預計算一次。

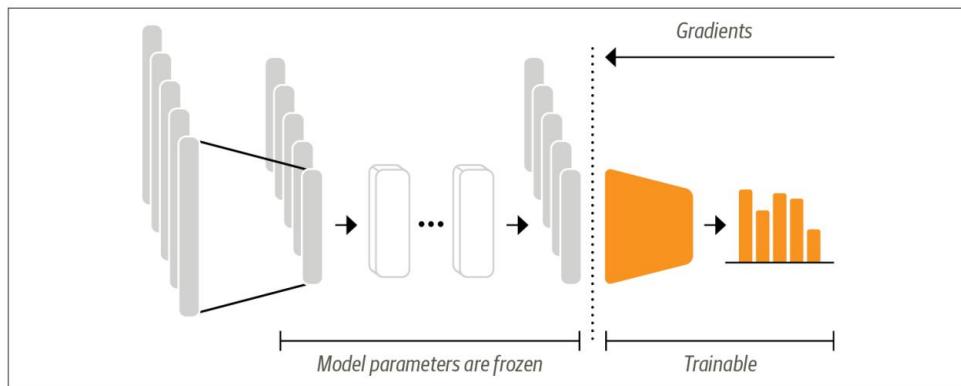


圖 2-5 在基於特徵的方法中，DistilBERT 模型被凍結，只為分類器提供特徵

使用預訓練模型

我們將使用來自 Transformers 的另一個方便的自動類，稱為AutoModel。

與AutoTokenizer類類似，AutoModel有一個from_pretrained()方法來加載預訓練模型的權重。讓我們使用此方法加載 DistilBERT 檢查點：

從變壓器導入AutoModel

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

在這裡，我們使用 PyTorch 檢查 GPU 是否可用，然後將 PyTorch nn.Module.to()方法鏈接到模型加載器。這確保了

如果我們有 GPU ,該模型將在 GPU 上運行 。否則 ,模型將在 CPU 上運行 ,這可能會慢得多 。

AutoModel類將標記編碼轉換為嵌入 ,然後通過編碼器堆棧提供它們以返回隱藏狀態 。讓我們來看看如何從我們的語料庫中提取這些狀態 。

框架之間的互操作性儘管本書中的代碼大部分是用

PyTorch 編寫的 ,但 Transformers 提供了與 TensorFlow 和 JAX 的緊密互操作性 。這意味著您只需更改幾行代碼即可在您最喜歡的深度學習框架中加載預訓練模型 !例如 ,我們可以使用 TFAutoModel類在 TensorFlow 中加載 DistilBERT ,如下所示 :

[從變壓器導入TFAutoModel](#)

```
tf_model = TFAutoModel.from_pretrained(model_ckpt)
```

當模型僅在一個框架中發布 ,但您想在另一個框架中使用時 ,這種互操作性特別有用 。例如 ,**XLM-RoBERTa 模型**我們將在**第 4 章**遇到的那個只有 PyTorch 權重 ,所以如果你嘗試像我們之前那樣在 TensorFlow 中加載它 :

```
tf_xlmr = TFAutoModel.from_pretrained( 'xlm-roberta-base' )
```

你會得到一個錯誤 。在這些情況下 ,您可以為TfAutoModel.from_pretrained()函數指定一個 from_pt=True參數 ,該庫將自動下載並為您轉換 PyTorch 權重 :

```
tf_xlmr = TFAutoModel.from_pretrained( 'xlm-roberta-base' ,from_pt=True)
```

如您所見 ,在 Transformers 中切換框架非常簡單 !在大多數情況下 ,您只需向類添加 “TF”前綴 ,即可獲得等效的 TensorFlow 2.0 類 。當我們使用PyTorch 的縮寫 “pt”字符串 (例如 ,在下一節中)時 ,只需將其替換為 TensorFlow 的縮寫 “tf” 。

提取最後的隱藏狀態為了熱身 ,

讓我們檢索單個字符串的最後隱藏狀態 。我們需要做的第一件事是對字符串進行編碼並將標記轉換為 PyTorch 張量 。這可以通過向分詞器提供return_tensors= pt 參數來完成 ,如下所示 :

```
text = "this is a test" inputs
= tokenizer(text, return_tensors= "pt" ) print(f" 輸入張量形
狀 :{inputs[ 'input_ids' ].size()} ")
輸入張量形狀 :torch.Size([1, 6])
```

正如我們所見，生成的張量具有 [batch_size, n_tokens] 的形狀。現在我們已經將編碼作為張量，最後一步是將它們放置在與模型相同的設備上並按如下方式傳遞輸入：

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

 BaseModelOutput(last_hidden_state=張量([-0.1565, -0.1862, 0.0528, ..., -0.1188, 0.0662, 0.5470],
 [-0.3575, -0.6484, -0.0618, ..., -0.3040, 0.3508, 0.5221], [-0.2772, -0.4459, 0.1818, ...,
 -0.0948, -0.0076, 0.9958], [-0.2841, -0.3917, 0.3753, ..., -0.2151, -0.1173, 1.0526],
 [0.2661, -0.5094, -0.3180, ..., -0.4203, 0.0144, -0.2149], [0.9441, 0.0112, -0.4714, ...,
 0.1439, -0.7288, -0.1619]]),
 device= cuda:0 ), hidden_states=None, attentions=None)
```

這裡我們使用了 `torch.no_grad()` 上下文管理器來禁用梯度的自動計算。這對於推理很有用，因為它減少了計算的內存佔用。根據模型配置，輸出可以包含多個對象，例如隱藏狀態、損失或註意力，它們排列在類似於 Python 中的命名元組的類中。在我们的示例中，模型輸出是 `BaseModelOutput` 的一個實例，我們可以簡單地通過名稱訪問它的屬性。當前模型只返回一個屬性，即最後一個隱藏狀態，所以讓我們檢查一下它的形狀：

```
輸出.last_hidden_state.size()
火炬.Size([1, 6, 768])
```

查看隱藏狀態張量，我們看到它的形狀為 [batch_size, n_tokens, hidden_dim]。換句話說，為 6 個輸入標記中的每一個返回一個 768 維向量。對於分類任務，通常的做法是僅使用與 [CLS] 標記相關的隱藏狀態作為輸入特徵。由於這個標記出現在每個序列的開頭，我們可以通過簡單地索引到 `outputs.last_hidden_state` 來提取它，如下所示：

```
輸出.last_hidden_state[:,0].size()
火炬.Size([1, 768])
```

現在我們知道如何獲取單個字符串的最後隱藏狀態；讓我們通過創建一個新的 `hidden_state` 列來存儲所有這些向量，對整個數據集做同樣的事情。正如我們對分詞器所做的那樣，我們將使用 `DatasetDict` 的 `map()` 方法一次提取所有隱藏狀態。我們需要做的第一件事是將前面的步驟包裝在一個處理函數中：

```
def extract_hidden_states (批處理) :
    # 在 GPU 上放置模型輸入 inputs =
    {k:v.to(device) for k,v in batch.items()}
```

```

    如果k在tokenizer.model_input_names 中}
#使用torch.no_grad()提取最後的隱藏狀態
態： last_hidden_state =
    model(**inputs).last_hidden_state # [CLS] 令牌的返回向量return
{ hidden_state :last_hidden_state[:,0].cpu().numpy() }

```

此函數與我們之前的邏輯之間的唯一區別是最後一步，我們將最終隱藏狀態作為 NumPy 數組放回 CPU。當我們使用批處理輸入時，`map()`方法要求處理函數返回 Python 或 NumPy 對象。

由於我們的模型需要張量作為輸入，接下來要做的是將`input_ids`和`attention_mask`列轉換為“torch”格式，如下所示：

```
emotions_encoded.set_format( 手電筒 ,
    columns=[ input_ids , attention_mask , label ])
```

然後我們可以繼續並一次提取所有拆分的隱藏狀態：

```
emotions_hidden = emotions_encoded.map (extract_hidden_states, batched=True)
```

請注意，在這種情況下我們沒有設置`batch_size=None`，這意味著使用默認的`batch_size=1000`。正如預期的那樣，應用`extract_hidden_states()`函數為我們的數據集添加了一個新的`hidden_state`列：

```
情緒隱藏[ “火車” ].column_names
[ attention_mask , hidden_state , input_ids , label , text ]
```

現在我們有了與每條推文關聯的隱藏狀態，下一步是在它們上訓練分類器。為此，我們需要一個特徵矩陣讓我們來看看。

創建一個特徵矩陣預處

理後的數據集現在包含了我們在其上訓練分類器所需的所有信息。我們將使用隱藏狀態作為輸入特徵，使用標籤作為目標。我們可以輕鬆地以眾所周知的 Scikit-learn 格式創建相應的數組，如下所示：

將`numpy`導入為`np`

```
X_train = np.array(emotions_hidden[ train ][ hidden_state ])
X_valid = np.array(emotions_hidden[ validation ][ hidden_state ])y_train =
np.array(emotions_hidden[ train ][ label ])y_valid =
np.array(emotions_hidden[ validation ][ 標籤"])
X_train.shape, X_valid.shape
((16000, 768), (2000, 768))
```

在我們訓練隱藏狀態的模型之前，最好進行快速檢查以確保它們提供了我們想要的情緒的有用表示

分類。在下一節中，我們將看到可視化功能如何提供一種快速的方法來實現這一目標。

可視化訓練集由於可視化

768 維的隱藏狀態至少可以說是棘手的，我們將使用強大的 UMAP 算法將向量向下投影到 2D。⁷因為當特徵被縮放到位於 [0,1] 區間，我們將首先應用 MinMaxScaler，然後使用umap-learn庫中的 UMAP 實現來減少隱藏狀態：

```
從umap導入UMAP從
sklearn.preprocessing導入MinMaxScaler

# 將特徵縮放到 [0,1] 範圍X_scaled =
MinMaxScaler().fit_transform(X_train)
# 初始化和擬合 UMAP mapper =
UMAP(n_components=2, metric= cosine ).fit(X_scaled)
# 創建一個二維嵌入的 DataFrame df_emb =
pd.DataFrame(mapper.embedding_, columns=[ X , Y ]) df_emb[ label ] =
y_train df_emb.head()
```

	x		標籤
0	4.358075	6.140816	0
1	-3.134567	5.329446	0
2	5.152230	2.732643	3
3	-2.519018	8.067250	2
4	-3.364520	8.356613	3

結果是一個包含相同數量訓練樣本的數組，但只有 2 個特徵，而不是我們開始時的 768 個。⁷讓我們進一步研究壓縮數據並分別繪製每個類別的點密度：

```
圖， axes = plt.subplots(2, 3, figsize=(7,5)) axes = axes.flatten()
cmaps = [ Greys , Blues , Oranges , Reds ,
Purples , Greens ] labels = emotions[ train ].features[ label ].names
```

```
對於我，枚舉 (zip (標籤， cmaps) 中的 (標籤， cmap)：df_emb_sub =
df_emb.query (f “label == {i}” ) axes [i].hexbin (df_emb_sub
[ “X” ]， df_emb_sub [ Y ]， cmap=cmap， gridsize=20， linewidths=(0,))
```

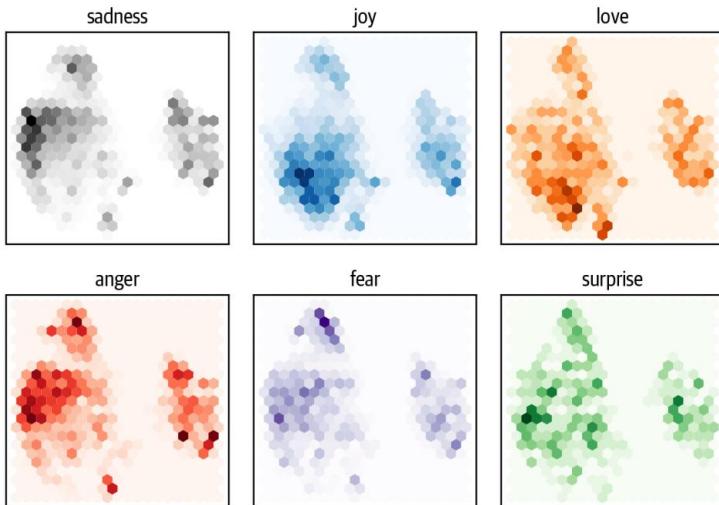
⁷ L. McInnes & J. Healy 和 J. Melville，“UMAP：用於降維的均勻流形近似和投影”，(2018)。

```

        軸[i].set_title(標籤)軸
        [i].set_xticks([]),軸[i].set_yticks([])

plt.tight_layout() plt.顯
示()

```



這些只是在低維空間上的投影。僅僅因為某些類別重疊並不意味著它們在原始空間中不可分離。相反，如果它們在投影空間中是可分離的，則它們在原始空間中也是可分離的。

從這個圖中我們可以看到一些清晰的模式：悲傷、憤怒和恐懼等負面情緒都佔據相似的區域，但分佈略有不同。另一方面，快樂和愛與負面情緒很好地分開，也共享一個相似的空間。最後，驚喜散落一地。儘管我們可能希望有一些分離，但這絕不能保證，因為模型沒有經過訓練來了解這些情緒之間的區別。它只是通過猜測文本中的掩碼詞來隱含地學習它們。

現在我們已經對數據集的特徵有了一些了解，最後讓我們在它上面訓練一個模型吧！

訓練一個簡單的分類器

我們已經看到，情緒之間的隱藏狀態有些不同，儘管其中一些沒有明顯的界限。讓我們使用這些隱藏狀態通過 Scikit-learn 訓練邏輯回歸模型。訓練這樣一個簡單的模型很快並且不需要 GPU：

從sklearn.linear_model導入LogisticRegression

```
# 我們增加 `max_iter` 以保證收斂lr_clf = LogisticRegression(max_iter=3000)
lr_clf.fit(X_train, y_train) lr_clf.score(X_valid, y_valid)
```

0.633

從準確性上看，我們的模型似乎只比隨機模型好一點。但由於我們處理的是不平衡的多類數據集，它實際上要好得多。我們可以通過將模型與簡單的基線進行比較來檢查我們的模型是否有用。在 Scikit-learn 中，有一個DummyClassifier 可用於構建具有簡單啟發式的分類器，例如始終選擇多數類或始終繪製隨機類。在這種情況下，性能最好的啟發式算法總是選擇最頻繁出現的類，其準確率約為 35%：

從sklearn.dummy導入DummyClassifier

```
dummy_clf = DummyClassifier(strategy= most_frequent )
dummy_clf.fit(X_train, y_train) dummy_clf.score(X_valid, y_valid)
```

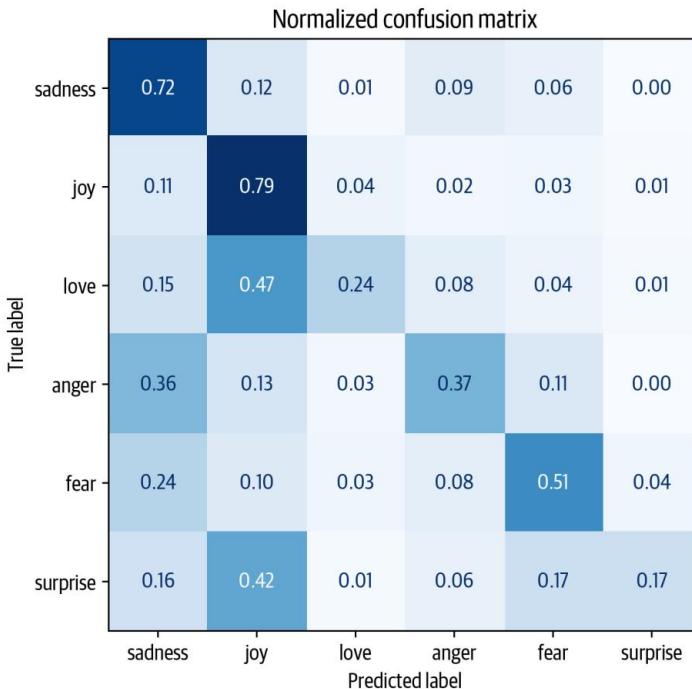
0.352

因此，我們帶有 DistilBERT 嵌入的簡單分類器明顯優於我們的基線。我們可以通過查看分類器的混淆矩陣進一步研究模型的性能，它告訴我們真實標籤和預測標籤之間的關係：

從sklearn.metrics導入ConfusionMatrixDisplay, confusion_matrix

```
def plot_confusion_matrix (y_preds, y_true ,標籤) :
    cm = confusion_matrix(y_true, y_preds, normalize= true )圖, ax =
    plt.subplots(figsize=(6, 6)) disp = ConfusionMatrixDisplay(confusion_matrix=cm,
    display_labels=labels) disp.plot(cmap= Blues ,values_format= .2f ,ax=ax, colorbar=False)
    plt.title( 歸一化混淆矩陣 ) plt.show()
```

```
y_preds = lr_clf.predict(X_valid)
plot_confusion_matrix(y_preds, y_valid, labels)
```



我們可以看到憤怒和恐懼最常與悲傷混淆，這與我們在可視化嵌入時所做的觀察一致。此外，愛和驚喜經常被誤認為是快樂。

在下一節中，我們將探索微調方法，這會帶來更好的分類性能。但是，請務必注意，這樣做需要更多的計算資源，例如 GPU，而您的組織可能沒有這些資源。在這種情況下，基於特徵的方法可以很好地折衷進行傳統機器學習和深度學習。

微調變壓器現在讓我們探討端

到端微調變壓器需要什麼。使用微調方法，我們不使用隱藏狀態作為固定特徵，而是如圖 2-6 所示訓練它們。這就要求分類頭是可微的，這就是為什麼這種方法通常使用神經網絡進行分類。

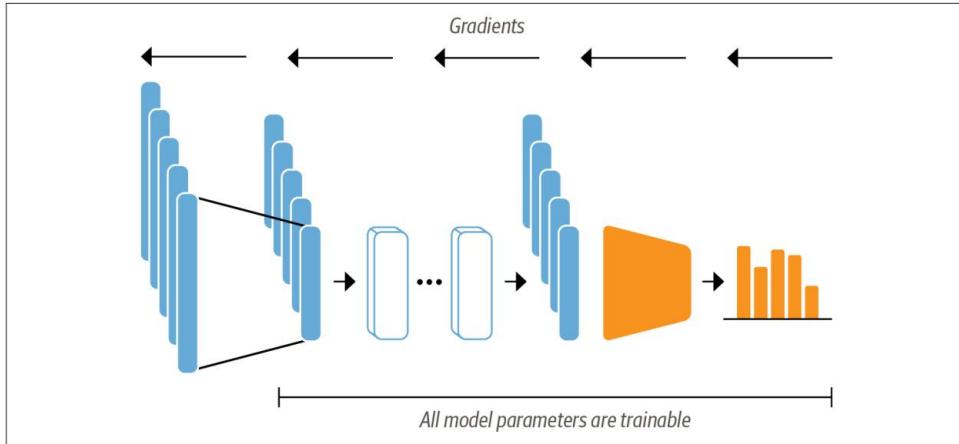


圖 2-6。當使用微調方法時，整個 DistilBERT 模型與分類頭一起訓練

訓練用作分類模型輸入的隱藏狀態將幫助我們避免處理可能不太適合分類任務的數據的問題。相反，初始隱藏狀態會在訓練期間進行調整以減少模型損失，從而提高其性能。

我們將使用Transformers 的Trainer API 来簡化訓練循環。
讓我們看看我們需要的成分來設置一個！

加載預訓練模型我們首先

需要的是預訓練 DistilBERT 模型，就像我們在基於特徵的方法中使用的模型一樣。唯一的細微修改是我們使用AutoModelFor SequenceClassification模型而不是AutoModel。不同之處在於AutoModelForSequenceClassification模型在預訓練模型輸出之上有一個分類頭，可以很容易地使用基礎模型進行訓練。我們只需要指定模型必須預測多少個標籤（在我們的例子中是六個），因為這決定了分類頭的輸出數量：

從變壓器導入AutoModelForSequenceClassification

```
num_labels = 6模型
= (AutoModelForSequenceClassification.from_pretrained(model_ckpt,
    num_labels=num_labels).to(device))
```

您將看到一條警告，指出模型的某些部分是隨機初始化的。這是正常的，因為分類頭還沒有被訓練。下一步是定義我們將用於在微調期間評估模型性能的指標。

定義性能指標為了在訓練期間監控指標，我們需要為Trainer 定義一個compute_metrics()函數。此函數接收一個EvalPrediction 對象（它是一個具有預測和label_ids屬性的命名元組）並且需要返回一個字典，該字典將每個指標的名稱映射到它的值。對於我們的應用程序，我們將計算F1分數和模型的準確性，如下所示：

從sklearn.metrics導入accuracy_score, f1_score

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

準備好數據集和指標後，在定義Trainer類之前，我們只需要處理最後兩件事：

1. 登錄我們在抱臉中心的賬號。這將使我們能夠將經過微調的模型推送到我們在 Hub 上的帳戶並與社區共享。
2. 定義訓練運行的所有超參數。

我們將在下一節中處理這些步驟。

訓練模型

如果您在 Jupyter notebook 中運行此代碼，則可以使用以下輔助函數登錄 Hub：

從huggingface_hub導入notebook_login

```
notebook_login()
```

這將顯示一個小部件，您可以在其中輸入用戶名和密碼，或具有寫入權限的訪問令牌。您可以在 Hub 文檔中找到有關如何創建訪問令牌的詳細信息。如果你在終端中工作，你可以通過運行以下命令來登錄：

\$ huggingface-cli 登錄

要定義訓練參數，我們使用TrainingArguments類。這個類存儲了大量信息，讓您可以對訓練和評估進行細粒度控制。要指定的最重要的參數是output_dir，這是存儲所有來自訓練的工件的地方。這是TrainingArguments的一個例子：

從變形金剛進口培訓師，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="錯誤")
```

在這裡，我們還設置了批量大小、學習率和 epoch 數，並指定在訓練運行結束時加載最佳模型。有了這個最終成分，我們就可以使用 Trainer 實例化和微調我們的模型：

從變壓器導入培訓師

```
培訓師=培訓師（模型=模型，args=training_args，
compute_metrics=compute_metrics，
train_dataset=emotions_encoded[ "train" ]，
eval_dataset=emotions_encoded[ "validation" ]，
tokenizer=tokenizer）
培訓師.train();
```

Epoch	Training Loss	Validation 損失精度 F		
0	0.840900	0.327445	0.896500	0.892285
200	0.255000	0.220472	0.922500	0.922550

查看日誌，我們可以看到我們的模型在驗證集上的F1分數約為 92%。這是對基於特徵的方法的重大改進！

我們可以通過計算混淆矩陣來更詳細地了解訓練指標。為了可視化混淆矩陣，我們首先需要獲得對驗證集的預測。Trainer類的predict ()方法返回幾個我們可以用於評估的有用對象：

```
preds_output = trainer.predict(emotions_encoded[ "驗證" ])
```

predict()方法的輸出是一個PredictionOutput對象，其中包含預測數組和label_id，以及我們傳遞給訓練器的指標。例如，可以按如下方式訪問驗證集上的指標：

```
preds_output.metrics
```

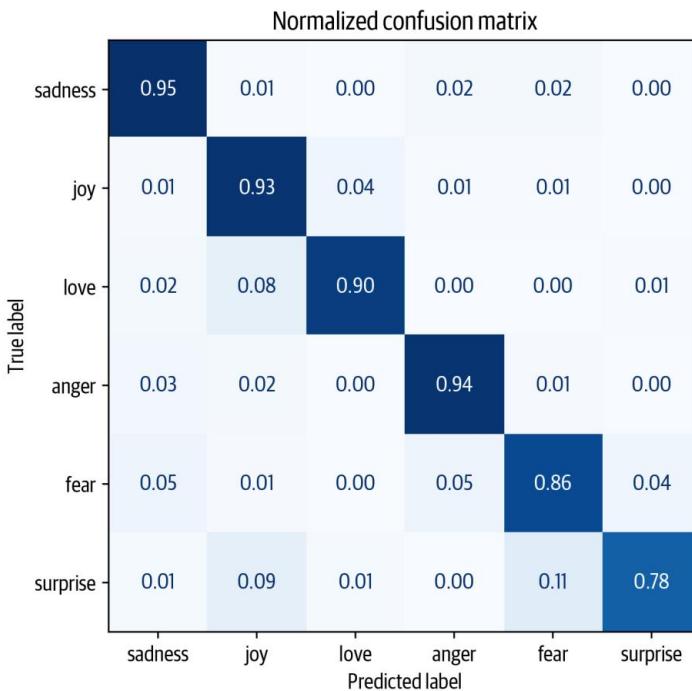
```
{ test_loss :  
  0.22047173976898193, test_accuracy :  
  0.9225, test_f1 :  
  0.9225500751072866, test_runtime :  
  1.6357, test_samples_per_second :  
  1222.725, test_steps_per_second : 19.564}
```

它還包含每個類別的原始預測。我們可以使用 `np.argmax()` 貪婪地解碼預測。這會產生預測標籤，並且與基於特徵的方法中 Scikit-learn 模型返回的標籤具有相同的格式：

```
y_preds = np.argmax(preds_output.predictions, axis=1)
```

有了預測，我們可以再次繪製混淆矩陣：

```
plot_confusion_matrix (y_preds, y_valid, 標籤)
```



這更接近於理想的對角線混淆矩陣。愛情範疇仍然經常與快樂相混淆，這似乎很自然。驚訝也經常被誤認為是喜悅，或者與恐懼相混淆。總體而言，該模型的性能似乎相當不錯，但在我們結束之前，讓我們更深入地了解一下我們的模型可能會犯的錯誤類型。

使用 Keras 微調如果您使用

TensorFlow，也可以使用 Keras API 微調您的模型。與 PyTorch API 的主要區別在於沒有 Trainer 類，因為 Keras 模型已經提供了內置的 fit() 方法。為了解其工作原理，我們首先將 DistilBERT 作為 TensorFlow 模型加載：

從變壓器導入 TFAutoModelForSequenceClassification

```
tf_model =  
    (TFAutoModelForSequenceClassification.from_pretrained(model_ckpt, num_labels=num_labels))
```

接下來，我們會將數據集轉換為 tf.data.Dataset 格式。因為我們已經填充了標記化的輸入，所以我們可以通過將 to_tf_dataset() 方法應用於 emotions_encoded 來輕鬆地進行此轉換：

```
# 要轉換為 TensorFlow 張量的列名 tokenizer_columns =  
    tokenizer.model_input_names  
  
tf_train_dataset = emotions_encoded[ train ].to_tf_dataset(  
    columns=tokenizer_columns, label_cols=[ label ], shuffle=True,  
    batch_size=batch_size) tf_eval_dataset = emotions_encoded[ 驗證 ].to_tf_dataset(  
  
    columns=tokenizer_columns, label_cols=[ label ], shuffle=False,  
    batch_size=batch_size)
```

在這裡，我們還打亂了訓練集，並為其和驗證集定義了批量大小。最後要做的是編譯和訓練模型：

將 tensorflow 導入為 tf

```
tf_model.compile(優化  
    器=tf.keras.optimizers.Adam(learning_rate=5e-5),  
    loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),  
    metrics=tf.metrics.SparseCategoricalAccuracy())  
  
tf_model.fit(tf_train_dataset, validation_data=tf_eval_dataset, epochs=2)
```

錯誤分析

在繼續之前，我們應該進一步調查一下我們模型的預測。一種簡單而強大的技術是根據模型損失對驗證樣本進行排序。

當我們在前向傳播過程中傳遞標籤時，損失會自動計算並返回。這是一個返回損失和預測標籤的函數：

從 torch.nn.functional 導入 cross_entropy

```
def forward_pass_with_label(batch): # 將所有輸  
    入張量放在與模型相同的設備上 inputs = {k:v.to(device) for k,v in batch.items()}
```

```
如果k在tokenizer.model_input_names 中}
```

```
用torch.no_grad(): output =
    model(**inputs) pred_label =
        torch.argmax(output.logits, axis=-1) loss = cross_entropy(output.logits,
            batch[ label ].to(device),
                減少= “無” )
    # 將輸出放在 CPU 上以與其他數據集列兼容return { loss : loss.cpu().numpy(),
        predicted_label : pred_label.cpu().numpy()}
```

再次使用map()方法，我們可以應用此函數來獲取所有樣本的損失：

```
# 將我們的數據集轉換回 PyTorch 張量
emotions_encoded.set_format( torch ,
                                columns=[ input_ids , attention_mask , label ])
# 計算損失值
emotions_encoded[ validation ]=emotions_encoded[ validation ].map( forward_pass_with_label,
    batched=True, batch_size=16)
```

最後，我們創建一個包含文本、損失和預測/真實標籤的DataFrame：

```
emotions_encoded.set_format( pandas ) cols
= [ text , label , predicted_label , loss ] df_test=
emotions_encoded[ 驗證 ][:] df_test[ 標籤 ]=
df_test[ label ].apply(label_int2str) df_test[ predicted_label ]=
(df_test[ predicted_label ].apply(label_int2str))
```

我們現在可以輕鬆地按升序或降序對按損失編碼的emotions_encoded進行排序。本練習的目標是檢測以下其中一項：

錯誤的標籤

每個向數據添加標籤的過程都可能存在缺陷。註釋者可能會犯錯誤或不同意，而從其他特徵推斷出的標籤可能是錯誤的。

如果自動註釋數據很容易，那麼我們就不需要模型來做這件事。因此，有一些錯誤標記的示例是正常的。通過這種方法，我們可以快速找到並糾正它們。

數據集的怪癖 現實世

界的數據集總是有點亂。在處理文本時，輸入中的特殊字符或字符串會對模型的預測產生很大影響。檢查模型最弱的預測可以幫助識別此類特徵，清理數據或註入類似示例可以使模型更加穩健。

我們先來看看損失最高的數據樣本：

```
df_test.sort_values( 損失 , ascending=False).head(10)
```

文本	標籤	預測標籤損失	
我覺得他被我自稱支持生命的配角所掩蓋，並在不知道這些信息的情況下投票給佩里小女孩把她們置於危險之中，以便在經濟上支持與他親近的人甜甜圈送給俄亥俄州哥倫布市聖家天主教堂的信徒。我作為美國公司一切錯誤的代表，覺得把他送到華盛頓是一個荒謬的想法。不知道如此深入的自我吸收讓我感到不舒服。	愛	悲傷	5.704531
喜悅	悲傷	5.484461	
	喜悅	悲傷	5.434768
	愛	悲傷	5.257482
	驚喜悲傷		4.827708
	喜悅	害怕	4.713047
我要去參加幾個假期聚會，我迫不及待地感到超級尷尬。我錯了，以為我可能是同性戀。我想我們自然會感到孤獨，即使是對你說不友好的話的人也可能會被錯過。我懶惰，我的角色屬於自鳴得意和或 blas 的人和他們的陪襯人感覺自鳴得意和或 blas 的人帶來的不便	喜悅	悲傷	4.704955
	害怕	悲傷	4.656096
	憤怒	悲傷	4.593202
	喜悅	害怕	4.311287

我們可以清楚地看到模型錯誤地預測了一些標籤。另一方面，似乎有很多示例沒有明確的類別，可能被錯誤標記或完全需要一個新類別。特別是，快樂似乎好幾次被貼錯了標籤。有了這些信息，我們可以改進數據集，這通常可以帶來與擁有更多數據或更大模型一樣大（或更多）的性能提升！

當查看損失最低的樣本時，我們觀察到該模型在預測悲傷類別時似乎最有信心。深度學習模型非常擅長尋找和利用捷徑來進行預測。出於這個原因，在時間查看模型最有信心的示例也是值得的，這樣我們就可以確信模型不會不正確地利用文本的某些特徵。因此，讓我們也看看損失最小的預測：

```
df_test.sort_values( 損失 , ascending=True).head(10)
```

文本	標籤	預測標籤損失
我覺得試著告訴我我忘恩負義告訴我我基本上是世界上最糟糕的女兒姐姐	悲傷悲傷	0.017331
我有點鬆了口氣，但同時我感到沮喪	悲傷悲傷	0.017392
我對此感到非常忘恩負義，但我期待著夏天、溫暖和明亮的夜晚，我記得有一天，當我們正在研究一首詩時，我感到很沮喪，一個節一個節地逐節地剖析它，我覺得自己像個忘恩負義的混蛋。我離開會議的感覺不僅僅是有點沮喪，我感到有點沮喪	悲傷悲傷	0.017400
	悲傷悲傷	0.017461
	悲傷悲傷	0.017485
	悲傷悲傷	0.017670
	悲傷悲傷	0.017685
我覺得我應該因為我的輕浮而被打破	悲傷悲傷	0.017888
我是出於純粹的意圖開始寫這個博客的，我必須承認最近開始感到有點沮喪，因為我知道似乎沒有人在讀它。我覺得現在希望懷孕結束真是忘恩負義	悲傷悲傷	0.017899
	悲傷悲傷	0.017913

我們現在知道快樂有時會被錯誤標記，並且該模型最有信心預測悲傷標籤。有了這些信息，我們可以對我們的數據集進行有針對性的改進，並密切關注模型似乎非常有信心的類別。

提供經過訓練的模型之前的最後一步是將其保存以備後用。Transformers 允許我們通過幾個步驟完成此操作，我們將在下一節中向您展示。

保存和共享模型NLP 社區從

共享預訓練和微調模型中受益匪淺，每個人都可以通過 Hugging Face Hub 與他人共享他們的模型。任何社區生成的模型都可以從 Hub 下載，就像我們下載 DistilBERT 模型一樣。使用 Trainer API，保存和共享模型很簡單：

```
trainer.push_to_hub(commit_message= "訓練完成！")
```

我們還可以使用微調模型對新推文進行預測。由於我們已將模型推送到 Hub，我們現在可以將它與pipeline()函數一起使用，就像我們在第 1 章中所做的那樣。首先，讓我們加載管道：

從變壓器導入管道

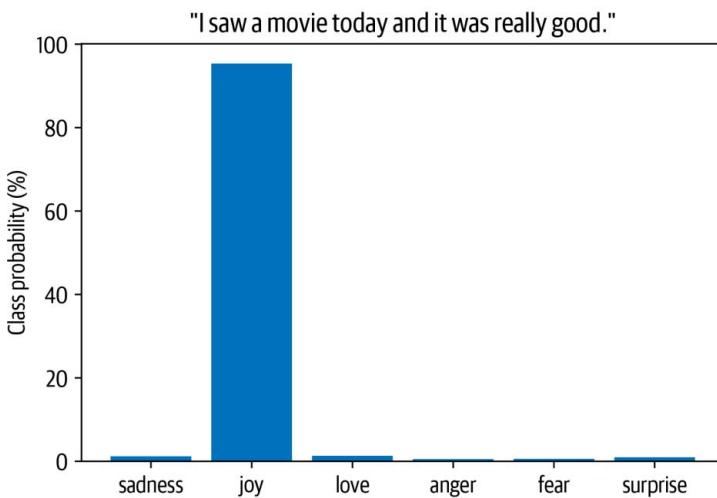
```
# 將 `transformersbook` 更改為您的 Hub 用戶名model_id =
transformersbook/distilbert-base-uncased-finetuned-emotion classifier = pipeline( text-
classification ,model=model_id)
```

然後讓我們使用示例推文測試管道：

```
custom_tweet = "我今天看了一部電影，真的很棒。"
preds = 分類器(custom_tweet,
return_all_scores=True)
```

最後，我們可以在條形圖中繪製每個類別的概率。顯然，該模型估計最有可能的類別是快樂，考慮到推文，這似乎是合理的：

```
preds_df = pd.DataFrame(preds[0])
plt.bar(labels, 100 * preds_df['score'], color='C0')
plt.title(f'{custom_tweet} 類別概率 (%)')
plt.show()
```



結論

恭喜，您現在知道如何訓練 Transformer 模型來對推文中的情緒進行分類！我們已經看到了兩種基於特徵和微調的互補方法，並研究了它們的優缺點。

然而，這只是使用 Transformer 模型構建真實世界應用程序的第一步，我們還有很多內容需要涵蓋。以下是您在 NLP 旅程中可能遇到的挑戰列表：

我的老闆昨天想讓我的模型投入生產！

在大多數應用程序中，您的模型不只是坐在積灰的地方。您要確保它正在為預測服務！將模型推送到 Hub 時，會自動創建一個可以使用 HTTP 請求調用的推理端點。我們建議查看[Inference API 的文檔](#)如果你想了解更多。

我的用戶想要更快的預測！

我們已經看到了解決這個問題的一種方法：使用 DistilBERT。在第 8 章中，我們將深入探討知識蒸餾（創建 DistilBERT 的過程）以及其他加速 transformer 模型的技巧。

你的模型也可以做 X 嗎？

正如我們在本章中提到的，變形金剛的用途極其廣泛。在本書的其餘部分，我們將探索一系列任務，如問答和命名實體識別，所有這些都使用相同的基本架構。

我的文字都不是英文的！

事實證明，轉換器也有多種語言版本，我們將在第 4 章中使用它們同時處理多種語言。

我沒有任何標籤！

如果可用的標記數據非常少，則可能無法進行微調。在第 9 章中，我們將探討一些處理這種情況的技術。

現在我們已經了解了訓練和共享轉換器所涉及的內容，在下一章中我們將從頭開始探索實現我們自己的轉換器模型。

第3章

變壓器解剖

在第 2 章中，我們了解了微調和評估變壓器所需的條件。現在讓我們來看看它們在幕後是如何工作的。在本章中，我們將探索變壓器模型的主要構建塊以及如何使用 PyTorch 實現它們。

我們還將提供有關如何在 TensorFlow 中執行相同操作的指導。我們將首先專注於構建注意力機制，然後添加使轉換器編碼器工作所需的點點滴滴。我們還將簡要了解架構差異

編碼器和解碼器模塊之間的引用。到本章結束時，您將能夠自己實現一個簡單的轉換器模型！

雖然對 Transformer 架構的深入技術理解通常不是使用 Transformers 和為您的用例微調模型所必需的，但它有助於理解導航 Transformer 的局限性以及在新領域中使用它們。

本章還介紹了 Transformer 的分類法，以幫助您了解近年來出現的眾多模型。在深入研究代碼之前，讓我們首先概述啟動變壓器革命的原始架構。

變壓器架構

正如我們在第 1 章中看到的，最初的 Transformer 基於編碼器-解碼器架構，該架構廣泛用於機器翻譯等任務，將一系列單詞從一種語言翻譯成另一種語言。該架構由兩個組件組成：

編碼器

將輸入的標記序列轉換為嵌入向量序列，通常稱為隱藏狀態或上下文

解碼器

使用編碼器的隱藏狀態迭代生成令牌輸出序列，一次一個令牌

如圖 3-1 所示，編碼器和解碼器本身由幾個構建塊組成。

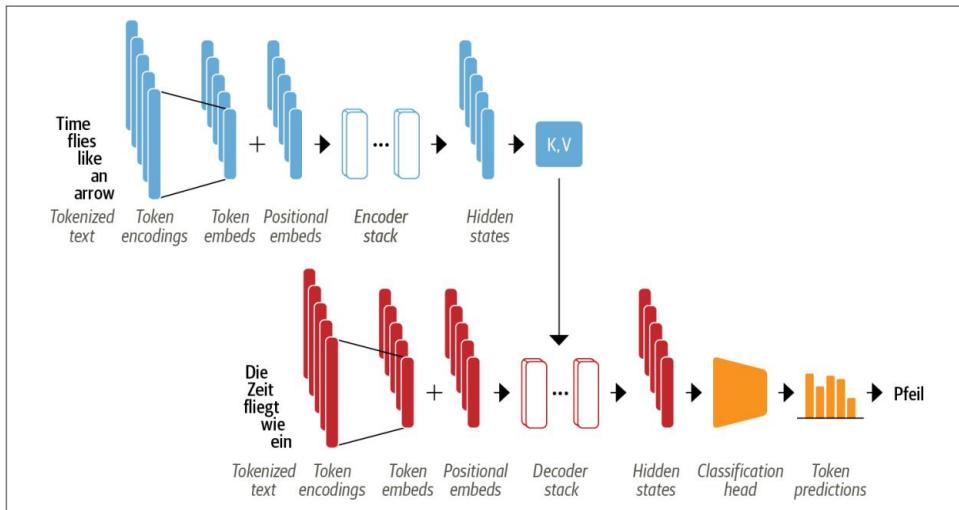


圖 3-1。transformer的Encoder-decoder架構，上半部分是encoder，下半部分是decoder

稍後我們將詳細查看每個組件，但我們已經可以在圖 3-1 中看到一些表徵 Transformer 架構的東西：

- 使用我們在第 2 章中遇到的技術將輸入文本標記化並轉換為標記嵌入。由於注意力機制不知道標記的相對位置，我們需要一種方法將有關標記位置的一些信息注入到輸入以模擬文本的順序性質。因此，令牌嵌入與包含每個令牌的位置信息的位置嵌入相結合。
- 編碼器由一堆編碼器層或“塊”組成，類似於計算機視覺中堆疊的捲積層。解碼器也是如此，它有自己的解碼器層堆棧。
- 編碼器的輸出被饋送到每個解碼器層，然後解碼器為序列中最有可能的下一個標記生成預測。然後將此步驟的輸出反饋到解碼器以生成下一個令牌，依此類推，直到達到特殊的序列結束 (EOS) 令牌。在圖 3-1 的示例中，假設解碼器已經預測了“Die”和“Zeit”。現在它

將這兩個作為輸入以及所有編碼器的輸出來預測下一個標記“fliegt”。在下一步中，解碼器將“fliegt”作為附加輸入。我們重複這個過程，直到解碼器預測到 EOS 令牌或我們達到最大長度。

Transformer 架構最初是為機器翻譯等序列到序列的任務而設計的，但編碼器和解碼器塊很快就被改編為獨立模型。儘管有數百種不同的變壓器模型，但其中大多數屬於以下三種類型之一：

Encoder-only

這些模型將輸入的文本序列轉換為豐富的數字表示，非常適合文本分類或命名實體識別等任務。

BERT 及其變體，如 RoBERTa 和 DistilBERT，屬於此類架構。在此架構中為給定標記計算的表示取決於左側（標記前）和右側（標記後）上下文。這通常稱為雙向注意。

僅解碼器給定

“謝謝你吃午飯，我有一個……”這樣的文本提示，這些模型將通過迭代預測最可能的下一個單詞來自動完成序列。

GPT 模型家族屬於此類。在此體系結構中為給定標記計算的表示僅取決於左上下文。這通常稱為因果或自回歸注意。

編解碼器

這些用於建模從一個文本序列到另一個文本序列的複雜映射；它們適用於機器翻譯和摘要任務。除了我們已經看到的結合了編碼器和解碼器的 Transformer 架構之外，BART 和 T5 模型也屬於此類。



實際上，僅解碼器架構與僅編碼器架構的應用程序之間的區別有點模糊。例如，像 GPT 系列中的那些僅解碼器的模型可以為翻譯等任務做好準備，這些任務通常被認為是序列到序列的任務。同樣，像 BERT 這樣的純編碼器模型可以應用於通常與編碼器-解碼器或純解碼器模型相關聯的摘要任務。¹

現在您已經對 Transformer 架構有了較高的了解，讓我們仔細看看編碼器的內部工作原理。

¹ Y. Liu 和 M. Lapata，“[使用預訓練編碼器進行文本摘要](#)”，(2019)。

編碼器

正如我們之前看到的，transformer 的編碼器由許多彼此相鄰堆疊的編碼器層組成。如圖 3-2 所示，每個編碼器層都接收到一系列嵌入，並將它們饋送到以下子層：

- 多頭自註意層 · 應用於每個輸入
- 嵌入的全連接前饋層

每個編碼器層的輸出嵌入與輸入具有相同的大小，我們很快就會看到編碼器堆棧的主要作用是“更新”輸入嵌入以生成對序列中的某些上下文信息進行編碼的表示。例如，如果單詞“keynote”或“phone”接近它，“apple”將被更新為更“company-like”而不是“fruit-like”。

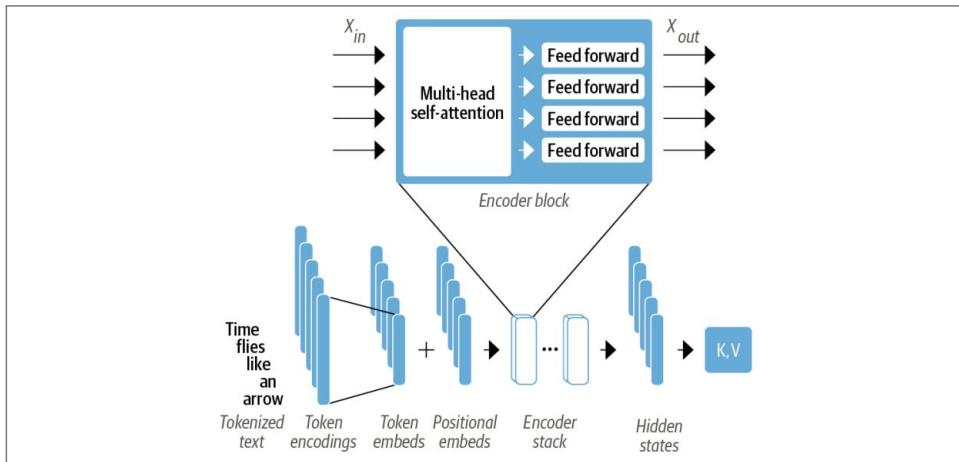


圖 3-2 放大編碼器層

這些子層中的每一個還使用跳躍連接和層歸一化，這是有效訓練深度神經網絡的標準技巧。但要真正了解是什麼讓變壓器工作，我們必須更深入。讓我們從最重要的構建塊開始：自註意力層。

自註意力

正如我們在第 1 章中討論的那樣，注意力是一種允許神經網絡為序列中的每個元素分配不同數量的權重或“注意力”的機制。

對於文本序列，元素是標記嵌入，就像我們在第 2 章中遇到的那樣，其中每個標記都映射到某個固定維度的向量。例如，在 BERT 中，每個標記都表示為 768 維向量。self-attention 的“self”部分指的是這些權重是針對同一集合中的所有隱藏狀態計算的。例如，編碼器的所有隱藏狀態。相比之下，與遞歸模型相關的注意機制涉及在給定的解碼時間步計算每個編碼器隱藏狀態與解碼器隱藏狀態的相關性。

self-attention 背後的主要思想是，我們可以使用整個序列來計算每個嵌入的加權平均值，而不是對每個標記使用固定的嵌入。另一種表述方式是說，給定一系列標記自註意力會產生一系列新嵌入 x'_1, \dots, x'_n ，其中每個 x'_i 是所有嵌入的線性組合 x_j ：

$$\text{習} = \sum_{j=1}^n w_{ji} x_j$$

這些係數被稱為 **權重** 或 **權重**，儘得慮當您看到 w “蒼蠅” 鏡像詞時會想標記麼。你可能會想到煩人的昆蟲，但如果你有更多的上下文，比如“time flies like arrow”，那麼你會意識到“flies”指的是動詞。類似地，我們可以通過以不同比例組合所有標記嵌入來為“蒼蠅”創建一個包含此上下文的表示，也許通過為“時間”和“箭頭”的標記嵌入分配更大的權重 w_{ji} 。以這種方式生成的嵌入稱為上下文化嵌入，早於 ELMo 等語言模型中轉換器的發明。² 圖 3-3 顯示了該過程的圖表，其中我們說明瞭如何根據上下文，兩個可以通過自註意力生成“蒼蠅”的不同表示。

² ME Peters 等人，“深度語境化詞表徵”，(2017)。

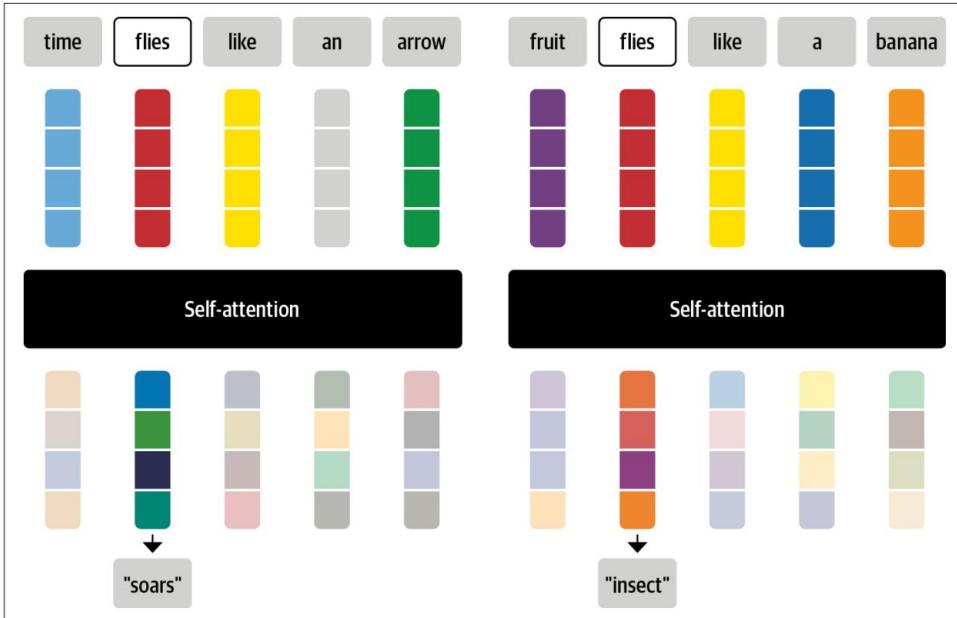


圖 3-3 顯示自註意力如何將原始標記嵌入（上）更新為上下文嵌入（下）以創建包含整個序列信息的表示的圖表

現在讓我們看看如何計算注意力權重。

縮放點積注意力有多種方

法可以實現自註意力層，但最常見的方法是縮放點積注意力，來自介紹 Transformer 架構的論文。3 實現此機制需要四個主要步驟：

1. 將每個標記嵌入投影到三個向量中，稱為查詢、鍵和值。
2. 計算注意力分數。我們使用相似度函數確定查詢和關鍵向量彼此相關的程度。顧名思義，縮放點積注意力的相似函數是點積，使用嵌入的矩陣乘法有效計算。相似的查詢和鍵會有很大的點積，而那些沒有太多共同點的查詢和鍵幾乎沒有重疊。這一步的輸出稱為注意力分數，對於具有 n 個輸入標記的序列，有一個相應的 $n \times n$ 注意力分數矩陣。

3 A. Vaswani 等， “Attention Is All You Need” ，(2017)。

3. 計算注意力權重。點積通常可以產生任意大的數字，這會破壞訓練過程的穩定性。為了解決這個問題，首先將注意力分數乘以一個比例因子以歸一化它們的方差，然後使用 softmax 歸一化以確保所有列值總和為 1。結果-

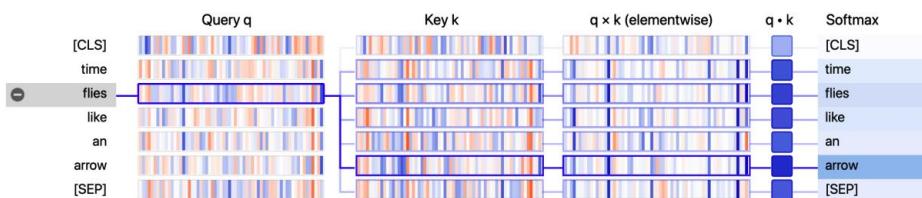
ing $n \times n$ 矩陣現在包含所有注意力權重 w_{ji} 。

4. 更新代幣嵌入。計算出注意力權重後，我們將它們乘以值向量 v_1, \dots, v_n 以獲得嵌入的更新表示 $x_i = \sum j w_{ji} v_j$ 。

我們可以用一個名為 BertViz for Jupyter 的漂亮庫來可視化注意力權重是如何計算的。這個庫提供了幾個函數，可以用來可視化 transformer 模型中註意力的不同方面。為了可視化注意力權重，我們可以使用 neuron_view 模塊，它跟蹤權重的計算以顯示查詢和關鍵向量如何組合以產生最終權重。由於 BertViz 需要利用模型的注意力層，我們將使用 BertViz 的模型類實例化我們的 BERT 檢查點，然後使用 show() 函數為特定的編碼器層和注意力頭生成交互式可視化。請注意，您需要單擊左側的“+”以激活注意力可視化：

從變壓器導入 AutoTokenizer 從
`bertviz.transformers_neuron_view` 導入 BertModel 從 `bertviz.neuron_view` 導入顯示

```
model_ckpt = bert-base-uncased
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
model = BertModel.from_pretrained(model_ckpt)
text = "光陰似箭"
顯示(模型, "bert", tokenizer, 文本, display_mode = "light", layer = 0, head = 8)
```



從可視化圖中，我們可以看到查詢和關鍵向量的值表示為垂直條帶，其中每個條帶的強度對應於大小。連接線根據標記之間的注意力進行加權，我們可以看到“蒼蠅”的查詢向量與“箭頭”的鍵向量具有最強的重疊。

揭開查詢、鍵和值的神秘面紗

詢、鍵和值向量的概念在您第一次遇到它們時可能看起來有點神秘。它們的名字受到信息檢索系統的啟發，但我們可以用一個簡單的類比來激發它們的含義。想像一下，你在超市購買晚餐所需的所有食材。您有這道菜的食譜，每一種所需的配料都可以看作是一個查詢。當您掃描貨架時，您會查看標籤（鍵）並檢查它們是否與您列表中的成分相匹配（相似度函數）。如果匹配，則從貨架上取下物品（價值）。

在這個類比中，對於每個與成分匹配的標籤，你只能得到一件雜貨。Self-attention是一個更抽象和“平滑”的版本：超市中的每個標籤與成分的匹配程度取決於每個鍵與查詢的匹配程度。因此，如果您的清單包括一打雞蛋，那麼您最終可能會得到 10 個雞蛋、一個煎蛋捲和一個雞翅。

讓我們通過實現計算縮放點積注意力的操作圖來更詳細地了解這個過程，如圖3-4 所示。

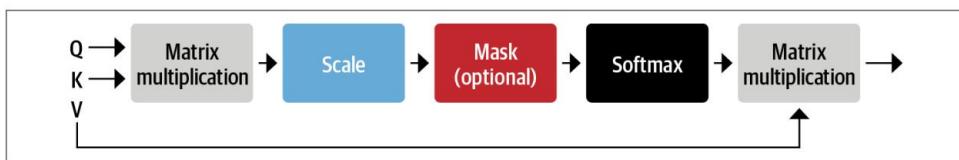


圖 3-4 °縮放點積注意力的操作

我們將在本章中使用 PyTorch 來實現 Transformer 架構，但 TensorFlow 中的步驟是類似的。我們在表 3-1 中提供了兩個框架中最重要功能之間的映射。

表 3-1 °本章使用的 PyTorch 和 TensorFlow (Keras) 類和方法

火炬	張量流（凱拉斯）	創建/實施
nn.線性	keras.layers.Dense	密集的神經網絡層
nn.模塊	keras.layers.圖層	模型的構建塊
nn.Dropout	keras.layers.Dropout	丟棄層
nn.LayerNorm	keras.layers.LayerNormalization	層歸一化一個嵌入層
nn.Embedding	keras.layers.嵌入	
格魯	keras.activations.gelu	高斯誤差線性單元激活函數
nn.bmm	tf.matmul	批量矩陣乘法
模型.前向	模型.調用	模型的前向傳播

我們需要做的第一件事是標記文本，所以讓我們使用我們的標記器來提取輸入 ID：

```
inputs = tokenizer(text, return_tensors= "pt", add_special_tokens=False)輸入.input_ids
```

```
張量([[ 2051, 10029, 2066, 2019, 8612]])
```

正如我們在第 2 章中看到的，句子中的每個標記都已映射到標記器詞彙表中的唯一 ID。為了簡單起見，我們還通過設置 `add_special_tokens=False` 排除了 [CLS] 和 [SEP] 標記。接下來，我們需要創建一些密集的嵌入。在這種情況下，密集意味著嵌入中的每個條目都包含一個非零值。相比之下，我們在第 2 章中看到的 one-hot 編碼是稀疏的，因為除了一個之外的所有條目都是零。在 PyTorch 中，我們可以通過使用 `torch.nn.Embedding` 層來完成此操作，該層充當每個輸入 ID 的查找表：

從 [火炬導入nn](#) 從 [變壓器導入](#)
AutoConfig

```
config = AutoConfig.from_pretrained(model_ckpt) token_emb =  
nn.Embedding(config.vocab_size, config.hidden_size)令牌_emb
```

```
嵌入(30522, 768)
```

在這裡，我們使用了 `AutoConfig` 類來加載與 `bert-base-uncased` 檢查點關聯的 `config.json` 文件。在 `Transformers` 中，每個檢查點都分配了 `vocab_size` 和 `hidden_size`。這兩項指定了我們在模型參數中向我們展示了每個輸入 ID 將映射到存儲在 `nn.Embedding` 中的 30,522 個嵌入向量之一，每個向量的大小為 768。

`AutoConfig` 類還存儲其他元數據，例如用於格式化模型預測的標籤名稱。

請注意，此時的標記嵌入與其上下文無關。這意味著同音異義詞（拼寫相同但含義不同的單詞），如前面示例中的“蒼蠅”，具有相同的表示形式。隨後的注意層的作用是混合這些標記嵌入，以消除歧義並告知每個標記的表示與其上下文的內容。

現在我們有了查找表，我們可以通過輸入輸入 ID 來生成嵌入：

```
inputs_embeds = token_emb(inputs.input_ids)  
inputs_embeds.size()  
torch.Size([1, 5, 768])
```

這給了我們一個形狀為 `[batch_size, seq_len, hidden_dim]` 的張量，就像我們在第 2 章中看到的那樣。我們將推遲位置編碼，所以下一步是

創建查詢、鍵和值向量，並使用點積作為相似度函數計算注意力分數：

從數學導入sqrt導入火炬

```
query = key = value = inputs_embeds dim_k =
key.size(-1) scores = torch.bmm(query,
key.transpose(1,2)) / sqrt(dim_k) 分數.size()
```

火炬.Size([1, 5, 5])

這為批次中的每個樣本創建了一個 5×5 的注意力分數矩陣。稍後我們將看到查詢、鍵和值向量是通過將獨立的權重矩陣 WQ 、 K 、 V 應用於嵌入來生成的，但現在為了簡單起見，我們將它們縮放到單一樣本。我們在計算過程中會遇到該步驟的進一步細節，這可能導致我們將在接下來應用的 softmax 飽和。



`torch.bmm()` 函數執行批量矩陣-矩陣乘積，簡化了注意力分數的計算，其中查詢和關鍵向量的形狀為 $[batch_size, seq_len, hidden_dim]$ 。如果我們忽略批量維度，我們可以通過簡單地將關鍵張量轉置為 $[hidden_dim, seq_len]$ 形狀然後使用矩陣積來收集每個查詢和關鍵向量之間的點積來計算 $[seq_len, seq_len]$ 矩陣。由於我們想獨立地對批次中的所有序列執行此操作，因此我們使用 `torch.bmm()`，它採用兩批矩陣並將第一批中的每個矩陣與第二批中的相應矩陣相乘。

現在讓我們應用 softmax：

將 `torch.nn.functional` 導入為 F

```
weights = F.softmax(scores, dim=-1)
weights.sum(dim=-1)
```

張量 $[[1., 1., 1., 1., 1.]]$, grad_fn=<SumBackward1>

最後一步是將注意力權重乘以值：

```
attn_outputs = torch.bmm (權重 * 值)
attn_outputs.shape
torch.Size([1, 5, 768])
```

就是這樣，我們已經完成了實現自我關注的簡化形式的所有步驟！請注意，整個過程只是兩個矩陣乘法和一個 softmax，因此您可以將“自註意力”視為一種奇特的平均形式。

讓我們將這些步驟包裝到一個我們可以稍後使用的函數中：

```
def scaled_dot_product_attention(query, key, value): dim_k =
    query.size(-1) scores = torch.bmm(query, key.transpose(1, 2)) /
    sqrt(dim_k) 權重= F.softmax(scores, dim =-1) 返回torch.bmm(權重, 值)
```

我們具有相等查詢和關鍵向量的注意力機制將為上下文中的相同單詞分配非常大的分數，特別是當前單詞本身：查詢與自身的點積始終為 1。但在實踐中，上下文中的互補詞比相同的詞更能說明一個詞的含義。例如，“flies”的含義通過結合“time”和“arrow”的信息比通過另一個提到的“flies”更好地定義。我們如何促進這種行為？

讓我們通過使用三種不同的線性投影將我們的初始標記向量投影到三個不同的空間，讓模型為標記的查詢、鍵和值創建一組不同的向量。

多頭注意力

在我們的簡單示例中，我們僅“按原樣”使用嵌入來計算注意力分數和權重，但這遠非全部。實際上，自註意力層對每個嵌入應用三個獨立的線性變換以生成查詢、鍵和值向量。這些轉換投影了嵌入，每個投影都帶有自己的一組可學習參數，這使得自註意力層可以專注於序列的不同語義方面。

事實證明，擁有多組線性投影也是有益的，每組代表一個所謂的注意力頭。生成的多頭注意力層如圖 3-5 所示。但是為什麼我們需要不止一個注意力頭呢？原因是一個頭部的 softmax 傾向於主要關注相似性的一個方面。

擁有多個頭可以讓模型同時關注多個方面。例如，一個腦袋可以專注於主謂互動，而另一個腦袋則可以找到附近的形容詞。顯然我們並沒有將這些關係手工製作到模型中，它們完全是從數據中學習到的。如果您熟悉計算機視覺模型，您可能會發現它與卷積神經網絡中的過濾器有相似之處，其中一個過濾器可以負責檢測人臉，另一個過濾器可以在圖像中找到汽車的車輪。

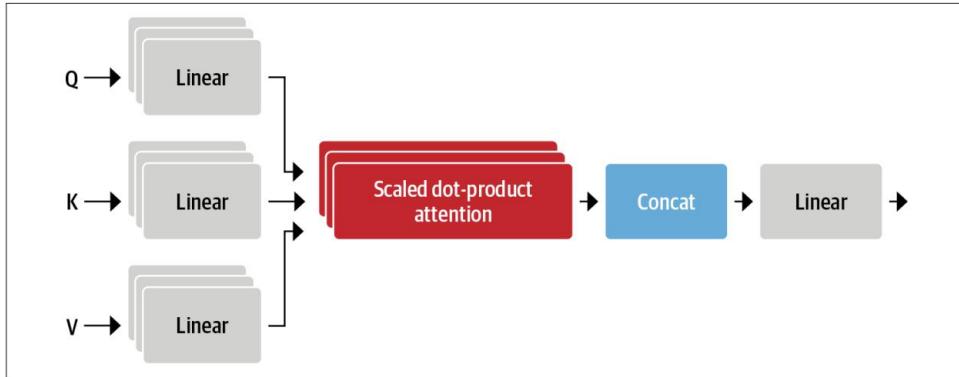


圖 3-5 多頭注意力

讓我們首先編寫一個注意力頭來實現這一層：

```
類AttentionHead(nn.Module): def __init__(self,
    embed_dim, head_dim):
    super().__init__()
    self.q = nn.Linear(embed_dim, head_dim)
    self.k = nn.Linear(embed_dim, head_dim)
    self.v = nn.Linear(embed_dim, head_dim)

    def forward(self, hidden_state):
        attn_outputs
        = scaled_dot_product_attention(
            self.q(hidden_state), self.k(hidden_state), self.v(hidden_state))
        return attn_outputs
```

這裡我們初始化了三個獨立的線性層，它們將矩陣乘法應用於嵌入向量以產生形狀為 [batch_size, seq_len, head_dim] 的張量，其中head_dim是我們要投影到的維度數。

儘管head_dim不必小於標記的嵌入維數(embed_dim)，但實際上它被選擇為embed_dim的倍數，以便每個頭的計算是恆定的。比如BERT有12個attention head，那麼每個head的維度就是 $768/12 = 64$ 。

現在我們有了一個注意力頭，我們可以連接每個注意力頭的輸出來實現完整的多頭注意力層：

```
類MultiHeadAttention(nn.Module): def __init__(self,
    config):
    super().__init__()
    self.embed_dim =
    config.hidden_size
    num_heads =
    config.num_attention_heads
    head_dim =
    embed_dim // num_heads
    self.heads =
    nn.ModuleList([AttentionHead(embed_dim, head_dim)
        對於
        在範圍內 (num_heads) ]
    )
    self.output_linear = nn.Linear(embed_dim, embed_dim)
```

```
def轉發（自我， hidden_state）：
```

```
x = torch.cat([h(hidden_state) for h in self.heads], dim=-1) x = self.output_linear(x)
返回x
```

請注意，來自註意力頭的串聯輸出也通過最終線性層饋送，以產生形狀為[batch_size, seq_len, hidden_dim]的輸出張量，適用於下游的前饋網絡。為了確認，讓我們看看多頭注意力層是否產生了我們輸入的預期形狀。

在初始化MultiHeadAttention模塊時，我們傳遞了之前從預訓練 BERT 模型加載的配置。這確保我們使用與 BERT 相同的設置：

```
multihead_attn = MultiHeadAttention(配置) attn_output
= multihead_attn(inputs_embeds) attn_output.size()
```

```
torch.Size([1, 5, 768])
```

有用！為了結束關於注意力的這一部分，讓我們再次使用 BertViz 來可視化“蒼蠅”一詞的兩種不同用法的注意力。在這裡，我們可以使用BertViz 的 head_view()函數來計算預訓練檢查點的注意力並指示句子邊界所在的位置：

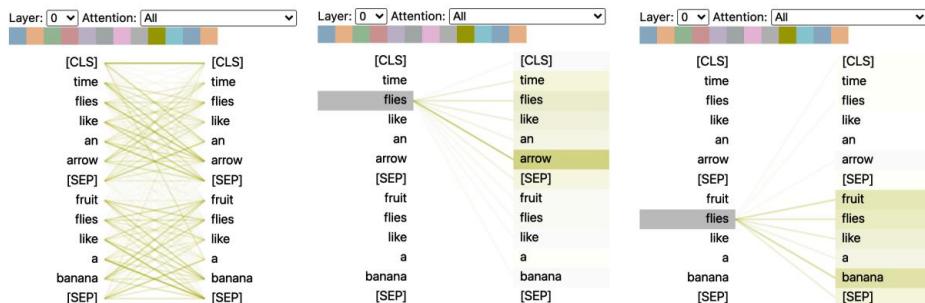
```
從bertviz導入head_view從變形金剛導入
AutoModel
```

```
模型= AutoModel.from_pretrained(model_ckpt, output_attentions=True)
```

```
sentence_a = 光陰似箭 sentence_b = 果蠅如香
蕉
```

```
viz_inputs = tokenizer(sentence_a, sentence_b, return_tensors= pt )注意=模型
(**viz_inputs).attentions sentence_b_start = (viz_inputs.token_type_ids == 0).sum(dim=1)
tokens = tokenizer.convert_ids_to_tokens(viz_inputs.input_ids [0])
```

```
head_view（注意，標記， sentence_b_start， heads=[8]）
```



此可視化將注意力權重顯示為連接其嵌入正在更新的令牌（左）與正在關注的每個單詞（右）的線。

線條的強度表示注意力權重的強度，深色線條表示接近 1 的值，淺色線條表示接近 0 的值。

在此示例中，輸入由兩個句子組成，[CLS]和[SEP]標記是我們在第 2 章中遇到的 BERT 分詞器中的特殊標記。從可視化中我們可以看到的一件事是注意力權重在屬於同一個句子的單詞，這表明 BERT 可以判斷它應該注意同一個句子中的單詞。然而，對於“flies”這個詞，我們可以看到 BERT 在第一句中將“arrow”識別為重要，在第二句中將“fruit”和“banana”識別為重要。這些注意力權重使模型能夠根據“蒼蠅”出現的上下文來區分“蒼蠅”是作為動詞還是名詞使用！

現在我們已經介紹了注意力，讓我們來看看實現編碼器層缺失的部分：位置前饋網絡。

前饋層編碼器和解碼器中的前

饋子層只是一個簡單的兩層全連接神經網絡，但有一個轉折點：它不是將整個嵌入序列作為單個向量處理，而是獨立處理每個嵌入。因此，該層通常被稱為位置前饋層。您可能還會看到它被稱為核大小為 1 的一維卷積，通常被具有計算機視覺背景的人稱為（例如，OpenAI GPT 代碼庫使用此命名法）。文獻中的經驗法則是第一層的隱藏大小是嵌入大小的四倍，最常用的是 GELU 激活函數。這是假設發生大部分容量和記憶的地方，也是在擴展模型時最常擴展的部分。我們可以將其實現為一個簡單的 nn.Module，如下所示：

```
類前饋 (nn.Module) :
    def __init__(self, config):
        super().__init__()
        self.linear_1 = nn.Linear(config.hidden_size, config.intermediate_size)
        self.linear_2 = nn.Linear(config.intermediate_size, config.hidden_size)
        self.gelu = nn.GELU()
        self.dropout = nn.Dropout(config.hidden_dropout_prob)

    def forward(self, x):
        x = self.linear_1(x)
        x = self.gelu(x)
        x = self.linear_2(x)
        x = self.dropout(x)
        return x
```

請注意，前饋層（如`nn.Linear`）通常應用於形狀為`(batch_size, input_dim)`的張量，它獨立地作用於批量維度的每個元素。這實際上對於除最後一個維度之外的任何維度都是正確的，所以當我們傳遞一個形狀為`(batch_size, seq_len, hidden_dim)`的張量時，該層將獨立應用於`batch`和`sequence`的所有標記嵌入，這正是我們想要的。讓我們通過傳遞注意力輸出來測試它：

```
feed_forward = FeedForward(配置) ff_outputs
= feed_forward(attn_outputs) ff_outputs.size()
```

```
torch.Size([1, 5, 768])
```

我們現在擁有創建完全成熟的轉換器編碼器層的所有要素！

剩下要做的唯一決定是在哪裡放置跳躍連接和層歸一化。讓我們來看看這如何影響模型架構。

添加層歸一化如前所述，Transformer

架構使用層歸一化和跳過連接。前者將批次中的每個輸入歸一化為零均值和單位方差。Skip connections 將一個張量傳遞到模型的下一層而不進行處理，並將其添加到已處理的張量中。當涉及到在轉換器的編碼器或解碼器層中放置層歸一化時，文獻中採用了兩種主要選擇：

Post layer normalization

這是Transformer論文中使用的安排；它將層歸一化放置在跳過連接之間。這種安排很難從頭開始訓練，因為梯度可能會發散。為此，你會經常看到一個稱為學習率預熱的概念，即在訓練過程中學習率從一個小值逐漸增加到某個最大值。

Pre layer normalization

這是文獻中最常見的安排；它將層歸一化置於跳過連接的範圍內。這在訓練期間往往更加穩定，並且通常不需要任何學習率預熱。

圖 3-6 說明了這兩種安排之間的區別。

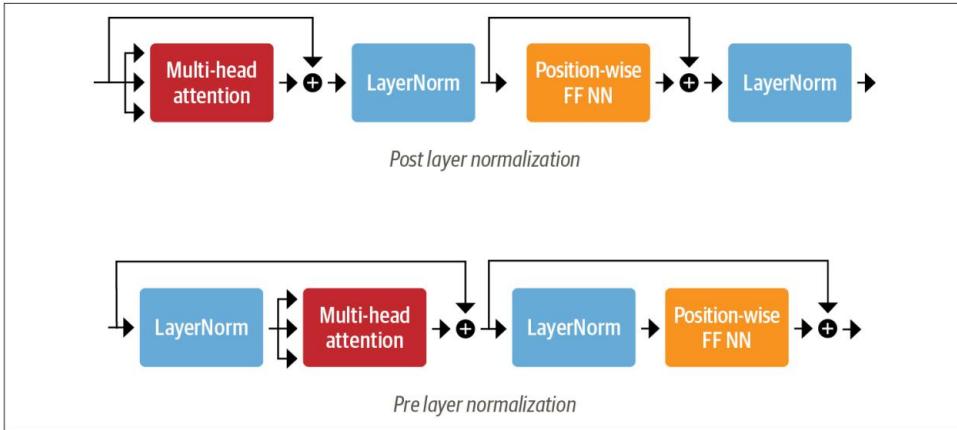


圖 3-6。Transformer 編碼器層中層歸一化的不同安排

我們將使用第二種安排，因此我們可以簡單地將我們的構建塊組合在一起，如下所示：

```
類TransformerEncoderLayer(nn.Module):
    def __init__(self, config):
        super().__init__()
        self.layer_norm_1 = nn.LayerNorm(config.hidden_size)
        self.layer_norm_2 = nn.LayerNorm(config.hidden_size)
        self.attention = MultiHeadAttention(配置)
        self.feed_forward = FeedForward(配置)

    def forward(self, x): # 應用層規範化，然後將輸入複製到查詢、鍵、值hidden_state = self.layer_norm_1(x)
        # 通過跳過連接應用注意力x = x + self.attention(hidden_state)
        # 應用帶跳躍連接的前饋層x = x + self.feed_forward(self.layer_norm_2(x))
        return x
```

現在讓我們用我們的輸入嵌入來測試這個：

```
encoder_layer = TransformerEncoderLayer(config)
inputs_embeds.shape, encoder_layer(inputs_embeds).大小()
(手電筒尺寸([1, 5, 768]), 手電筒尺寸([1, 5, 768]))
```

我們現在已經從頭開始實現了我們的第一個轉換器編碼器層！然而，我們設置編碼器層的方式有一個警告：它們完全是

對標記的位置不變。由於多頭注意力層實際上是一個奇特的加權和，因此有關標記位置的信息丟失了。⁴幸運的是，有一個簡單的技巧可以使用位置嵌入來合併位置信息。讓我們來看看。

Positional Embeddings位置

嵌入基於一個簡單但非常有效的想法：使用排列在向量中的位置相關的值模式來增加標記嵌入。

如果模式是每個位置的特徵，則每個堆棧中的注意力頭和前饋層都可以學習將位置信息合併到它們的轉換中。

有幾種方法可以實現這一點，最流行的方法之一是使用可學習模式，尤其是當預訓練數據集足夠大時。

這與令牌嵌入的工作方式完全相同，但使用位置索引而不是令牌 ID 作為輸入。通過這種方法，可以在預訓練期間學習一種有效的編碼標記位置的方法。

讓我們創建一個自定義嵌入模塊，該模塊結合了將 input_ids 投影到密集隱藏狀態的標記嵌入層以及對 position_ids 執行相同操作的位置嵌入。生成的嵌入只是兩個嵌入的總和：

```
嵌入類 (nn.Module) :
    def __init__(self, config):
        super().__init__()
        self.token_embeddings = nn.Embedding(config.vocab_size,
                                             config.hidden_size)
        self.position_embeddings = nn.Embedding(config.max_position_embeddings, config.hidden_size)
        self.layer_norm =
            nn.LayerNorm(config.hidden_size, eps=1e-12)
        self.dropout = nn.Dropout()

    def forward(self, input_ids): # 為輸入序
        # 創建位置 ID seq_length = input_ids.size(1)
        position_ids =
            torch.arange(seq_length, dtype=torch.long).unsqueeze(0)

        # 創建令牌和位置嵌入 token_embeddings =
        self.token_embeddings(input_ids) position_embeddings =
        self.position_embeddings(position_ids)
        # 組合標記和位置嵌入 embeddings = token_embeddings
        + position_embeddings embeddings = self.layer_norm(embeddings)
```

⁴用更高級的術語來說，自注意力層和前饋層被稱為置換等變。如果輸入被置換，則該層的相應輸出將以完全相同的方式置換。

```

embeddings = self.dropout(embeddings)返回嵌入

embedding_layer = Embeddings(配置)
embedding_layer(inputs.input_ids).size()

torch.Size([1, 5, 768])

```

我們看到嵌入層現在為每個標記創建了一個單一的、密集的嵌入。

雖然可學習的位置嵌入易於實現和廣泛使用，但還有一些替代方案：

絕對位置表示 Transformer 模

型可以使用由調製正弦和余弦信號組成的靜態模式來對標記的位置進行編碼。當沒有大量可用數據時，這種方法特別有效。

相對位置表示 儘管絕對位置很

重要，但可以爭辯說，在計算嵌入時，周圍的標記是最重要的。相對位置表示遵循該直覺並對令牌之間的相對位置進行編碼。這不能通過在開始時引入一個新的相對嵌入層來設置，因為每個標記的相對嵌入都會根據我們關注它的序列的位置而變化。取而代之的是，注意機製本身被修改為額外的條款，這些條款考慮了令牌之間的相對位置。DeBERTa 等模型使用此類表示。5現在讓我們通過構建將嵌入與編碼器層相結合的完整變換器編碼器將所有這些放在一起：

```

類TransformerEncoder(nn.Module): def
    __init__(self, config): super().__init__()
        self.embeddings = Embeddings(config)
        self.layers =
            nn.ModuleList([TransformerEncoderLayer(config) for i in range(config.
                num_hidden_layers)])

```

```

def forward(self, x): x =
    self.embeddings(x) for layer
    in self.layers: x = layer(x) return x

```

讓我們檢查一下編碼器的輸出形狀：

5通過結合絕對和相對位置表示的思想，旋轉位置嵌入在許多任務上取得了出色的效果。GPT-Neo 是具有旋轉位置嵌入的模型示例。

```
encoder = TransformerEncoder(config)編碼器
(inputs.input_ids).size()

torch.Size([1, 5, 768])
```

我們可以看到我們為批次中的每個標記獲得了一個隱藏狀態。這種輸出格式使體系結構非常靈活，我們可以輕鬆地將其應用於各種應用程序，例如預測掩碼語言建模中丟失的標記或預測問答中答案的開始和結束位置。在下一節中，我們將看到如何構建一個類似於第 2 章中使用的分類器。

Adding a Classification Head

Transformer 模型通常分為任務無關的主體和任務特定的頭部。我們將在第 4 章查看 Transformers 的設計模式時再次遇到這種模式。到目前為止，我們構建的是正文，因此如果我們希望構建文本分類器，我們需要將分類頭附加到該正文。我們對每個標記都有一個隱藏狀態，但我們只需要做出一個預測。有幾種選擇可以解決這個問題。傳統上，此類模型中的第一個標記用於預測，我們可以附加一個 dropout 和一個線性層來進行分類預測。以下類擴展了現有的序列分類編碼器：

```
類TransformerForSequenceClassification (nn.Module) :
    def __init__(self, config):
        super().__init__()
        self.encoder
        = TransformerEncoder(config)
        self.dropout =
        nn.Dropout(config.hidden_dropout_prob)
        self.classifier =
        nn.Linear(config.hidden_size, config.num_labels)

    def forward(self, x):
        x =
        self.encoder(x)[:, 0, :]
        # 選擇 [CLS] 標記的隱藏狀態
        x = self.dropout(x)
        x = self.classifier(x)
        return x
```

在初始化模型之前，我們需要定義我們想要預測的類別數量：

```
config.num_labels = 3
encoder_classifier = TransformerForSequenceClassification(config)編碼器_分類器
(inputs.input_ids).size()

torch.Size([1, 3])
```

這正是我們一直在尋找的。對於批處理中的每個示例，我們得到輸出中每個類的非標準化 logits。這對應於我們在第 2 章中用於檢測推文中的情緒的 BERT 模型。

我們對編碼器的分析以及我們如何將其與任務特定的頭結合起來就結束了。現在讓我們把注意力（雙關語！）轉移到解碼器上。

解碼器

如圖 3-7 所示，解碼器和編碼器之間的主要區別在於解碼器有兩個注意力子層：

蒙面多頭自註意層

確保我們在每個時間步生成的標記僅基於過去的輸出和正在預測的當前標記。否則，解碼器可能會在訓練期間通過簡單地複制目標翻譯來作弊；屏蔽輸入確保任務不是微不足道的。

編碼器-解碼器注意層對編碼器堆

樣的輸出鍵和值向量執行多頭注意，解碼器的中間表示作為查詢。⁶ 這樣，編碼器-解碼器注意層學習如何將來自兩個標記的標記相關聯不同的序列，例如兩種不同的語言。解碼器可以訪問每個塊中的編碼器鍵和值。

讓我們看一下我們需要進行的修改以在我們的自註意層中包含掩碼，並將編碼器-解碼器注意層的實現留作作業問題。masked self-attention 的技巧是引入一個掩碼矩陣，其下對角線為 1，上方為 0：

```
seq_len = inputs.input_ids.size(-1)
mask = torch.tril(torch.ones(seq_len, seq_len)).unsqueeze(0).mask[0]
```

```
張量([[1., 0., 0., 0., 0.], [1., 1., 0., 0., 0.],
       [1., 1., 1., 0., 0.], [1., 1., 1.,
       1., 0.], [1., 1., 1., 1., 1.]])
```

在這裡，我們使用 PyTorch 的 `tril()`函數來創建下三角矩陣。

一旦我們有了這個掩碼矩陣，我們就可以通過使用 `Tensor.masked_fill()` 將所有零替換為負無窮大來防止每個注意力頭偷看未來的標記：

```
scores.masked_fill(mask == 0, -float('inf'))
```

⁶請注意，與自註意層不同，編碼器-解碼器注意力中的鍵和查詢向量可以有不同的長度。這是因為編碼器和解碼器輸入通常會涉及不同長度的序列。因此，這一層的注意力分數矩陣是矩形的，而不是正方形的。

```
張量 ([[[26.8082, -inf, -inf, -inf, -0.6981, 26.9043, -inf, -inf, [-2.3190, 1.2928, 27.8710, 0, 170.5898, 807, 27.5488, [0.5275, 2.0493, -0.4869, 1.6100, 29.0893]]], grad_fn=<MaskedFillBackward0>)
```

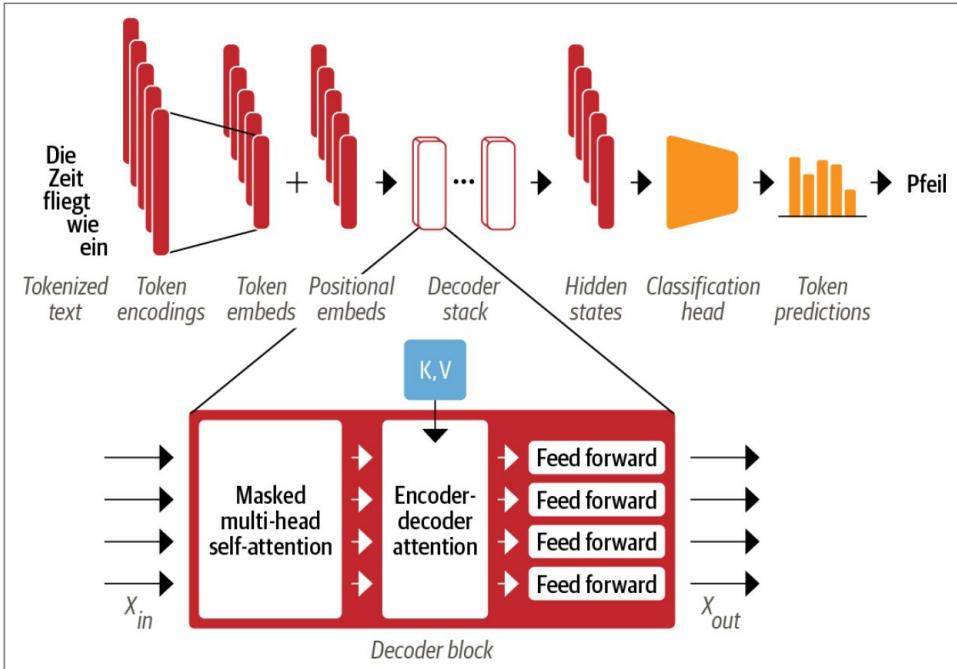


圖 3-7 放大到變壓器解碼器層

通過將上限值設置為負無窮大，我們保證一旦我們對分數進行 softmax 計算時注意力權重全部為零，因為 $e^{-\infty} = 0$ （回想一下 softmax 計算歸一化指數）。我們可以很容易地包含這種屏蔽行為，只需對我們在本章前面實現的縮放點積注意力函數做一個小改動：

```
def scaled_dot_product_attention(query, key, value, mask=None): dim_k = query.size(-1)
    scores = torch.bmm(query, key.transpose(1, 2)) / sqrt(dim_k)如果掩碼不是None :
        scores = scores.masked_fill(mask == 0, float( -inf )) weights =
        F.softmax(scores, dim=-1)返回weights.bmm(value)
```

從這裡開始構建解碼器層是一件簡單的事情；我們向讀者指出 minGPT 的出色實施由 Andrej Karpathy 提供詳細信息。

我們在這裡為您提供了大量技術信息，但現在您應該對 Transformer 架構的每一部分如何工作有了很好的理解。在我們繼續為比文本分類更高級的任務構建模型之前，讓我們回顧一下不同的轉換器模型的概況以及它們之間的相互關係來結束本章。

揭開編碼器-解碼器注意力的神秘面紗

讓我們看看能否揭開編碼器-解碼器注意力的神秘面紗。

想像一下你（解碼器）正在課堂上參加考試。你的任務是根據前面的單詞（解碼器輸入）預測下一個單詞，這聽起來很簡單但非常難（自己嘗試並預測本書段落中的下一個單詞）。

幸運的是，你的鄰居（編碼器）有全文。不幸的是，他們是一名外國交換生，文本是用他們的母語寫的。狡猾的學生，你想辦法作弊。你畫了一幅小卡通來說明你已有的文本（查詢），然後把它交給你的鄰居。他們試圖找出哪一段與該描述相匹配（關鍵），畫一幅漫畫來描述該段落後面的單詞（值），然後將其傳回給你。有了這個系統，您就可以通過考試。

認識變形金剛

正如您在本章中看到的，Transformer 模型主要有三種架構：編碼器、解碼器和編碼器-解碼器。早期 transformer 模型的初步成功引發了模型開發的寒武紀大爆炸，因為研究人員在不同大小和性質的各種數據集上構建模型，使用新的預訓練目標，並調整架構以進一步提高性能。

儘管模型庫仍在快速增長，但仍可分為這三類。

在本節中，我們將簡要概述每個類中最重要的轉換器模型。讓我們首先看一下 transformer 家族樹。

變形金剛生命之樹

隨著時間的推移，三種主要架構中的每一種都經歷了自己的演變。圖 3-8 對此進行了說明，其中顯示了一些最突出的模型及其後代。

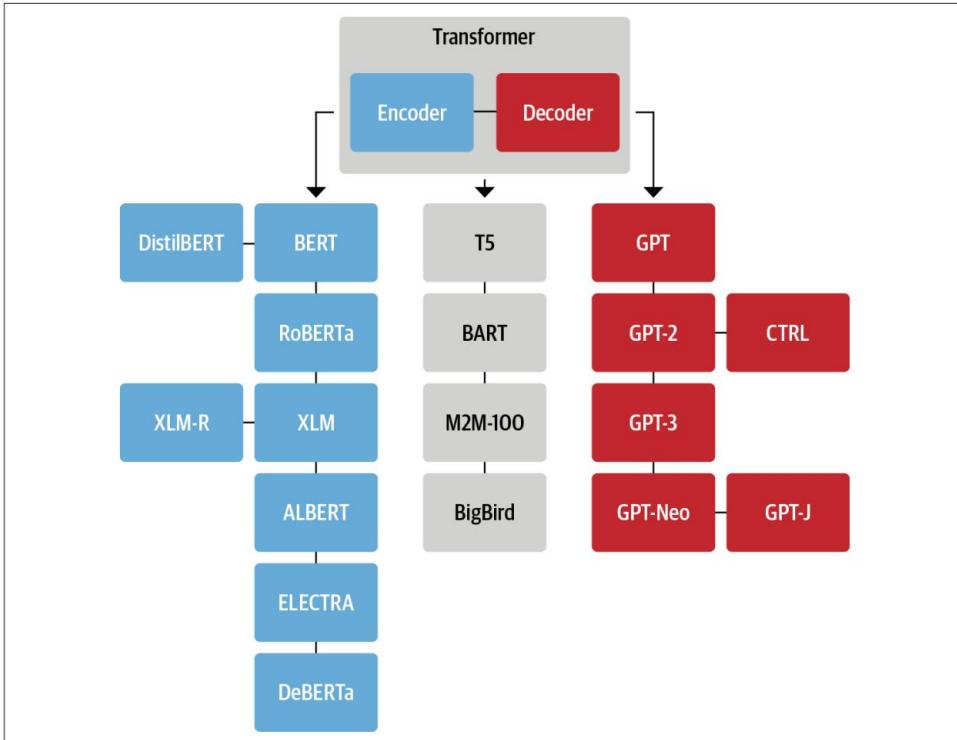


圖 3-8 一些最著名的變壓器架構的概述

Transformers 中包含超過 50 種不同的架構，這個家譜絕不提供所有現有架構的完整概述：它只是強調了幾個架構里程碑。我們已經在本章中深入介紹了原始的 Transformer 架構，所以讓我們仔細看看一些關鍵的後代，從編碼器分支開始。

編碼器分支

第一個基於 Transformer 架構的僅編碼器模型是 BERT。在它發佈時，它在流行的 GLUE 基準測試中優於所有最先進的模型，⁷ 該基準測試跨多個不同難度的任務測量自然語言理解 (NLU)。隨後，對 BERT 的預訓練目標和架構進行了調整，以進一步提高性能。

僅編碼器模型仍然主導著 NLU 任務（例如文本）的研究和行業

⁷ A. Wang 等人，“GLUE：自然語言理解的多任務基準和分析平台-

荷蘭國際集團” (2018)。

分類、命名實體識別和問答。讓我們簡單了解一下 BERT 模型及其變體：

BERT

BERT 預訓練有兩個目標：預測文本中的掩碼標記和確定一個文本段落是否可能跟在另一個文本段落之後。⁸ 前一個任務稱為掩碼語言建模 (MLM)，後者稱為下一句預測 (NSP)。

DistilBERT

儘管 BERT 提供了很好的結果，但它的規模可能使其難以部署在需要低延遲的環境中。通過在預訓練期間使用一種稱為知識蒸餾的技術，DistilBERT 實現了 BERT 97% 的性能，同時使用的內存減少了 40%，速度提高了 60%。⁹ 您可以在第 8 章中找到有關知識蒸餾的更多詳細信息。

RoBERTa

BERT 發布後的一項研究表明，可以通過修改預訓練方案來進一步提高其性能。RoBERTa 的訓練時間更長，批次更大，訓練數據更多，它放棄了 NSP 任務。¹⁰ 與原始 BERT 模型相比，這些變化共同顯著提高了其性能。

XLM

在跨語言語言模型 (XLM)¹¹ 的工作中探索了構建多語言模型的幾個預訓練目標，包括來自類 GPT 模型的自回歸語言建模和來自 BERT 的 MLM。此外，XLM 預訓練論文的作者介紹了翻譯語言建模 (TLM)，它是 MLM 對多語言輸入的擴展。通過對這些預訓練任務進行試驗，他們在多個多語言 NLU 基準測試以及翻譯任務上取得了最先進的結果。

XLM-羅伯塔

繼 XLM 和 RoBERTa 的工作之後，XLM-RoBERTa 或 XLM-R 模型通過大規模升級訓練數據，使多語言預訓練更進一步。¹² 使用 Common Crawl 語料庫，它的開發人員創建了一個包含 2.5 TB 文本的數據集；然後他們用 MLM 訓練了一個編碼器。

8 J. Devlin 等人，“BERT：用於語言理解的深度雙向轉換器的預訓練”，
(2018)。

9 V. Sanh 等人，“DistilBERT：BERT 的精簡版：更小、更快、更便宜、更輕”，(2019)。

10 Y. Liu 等人，“RoBERTa：一種穩健優化的 BERT 預訓練方法”，(2019)。

11 G. Lample 和 A. Conneau，“跨語言語言模型預訓練”，(2019)。

12 A. Conneau 等人，“大規模無監督跨語言表示學習”，(2019)。

數據集。由於數據集僅包含沒有平行文本（即翻譯）的數據，因此放棄了 XLM 的 TLM 目標。這種方法大大擊敗了 XLM 和多語言 BERT 變體，尤其是在低資源語言上。

ALBERT

ALBERT 模型引入了三個變化，使編碼器架構更高效。¹³首先，它將令牌嵌入維度與隱藏維度解耦，從而允許嵌入維度較小，從而節省參數，尤其是當詞彙量變大時。其次，所有層共享相同的參數，這進一步減少了有效參數的數量。最後，NSP 目標被句子順序預測所取代：模型需要預測兩個連續句子的順序是否被交換，而不是預測它們是否屬於一起。這些變化使得用更少的參數訓練更大的模型成為可能，並在 NLU 任務上達到卓越的性能。

伊萊克特拉

標準 MLM 預訓練目標的一個限制是，在每個訓練步驟中，只有屏蔽標記的表示會被更新，而其他輸入標記則不會。為了解決這個問題，ELECTRA 使用了兩種模型的方法：¹⁴第一個模型（通常很小）的工作方式類似於標準的屏蔽語言模型並預測屏蔽標記。第二個模型稱為鑑別器，然後負責預測第一個模型輸出中的哪些標記最初被屏蔽了。因此，判別器需要對每個 token 進行二元分類，從而使訓練效率提高 30 倍。對於下游任務，鑑別器會像標準 BERT 模型一樣進行微調。

德貝塔

DeBERTa 模型引入了兩個架構變化。¹⁵首先，每個標記都表示為兩個向量：一個表示內容，另一個表示相對位置。通過將標記的內容從它們的相對位置中分離出來，自註意力層可以更好地模擬附近標記對的依賴性。另一方面，單詞的絕對位置也很重要，尤其是對於解碼而言。

為此，在令牌解碼頭的 softmax 層之前添加了絕對位置嵌入。DeBERTa 是第一個模型（作為一個整體）

¹³ Z. Lan 等人，“ALBERT：用於語言表示自監督學習的 Lite BERT”，（2019）。

¹⁴ K. Clark 等人，“ELECTRA：預訓練文本編碼器作為鑑別器而不是生成器”，（2020）。

¹⁵ P. He 等人，“DeBERTa：解碼增強型 BERT 與分離注意力”，（2020）。

在 SuperGLUE 基準測試 16 上擊敗了人類基線，這是一個更難的 GLUE 版本，由幾個用於衡量 NLU 性能的子任務組成。

現在我們已經強調了一些主要的純編碼器架構，讓我們來看看純解碼器模型。

解碼器分支

OpenAI 在很大程度上引領了 Transformer 解碼器模型的進步。這些模型非常擅長預測序列中的下一個單詞，因此主要用於文本生成任務（更多細節參見第 5 章）。通過使用更大的數據集並將語言模型擴展到越來越大的規模，推動了他們的進步。讓我們來看看這些引人入勝的生成模型的演變：

GPT

GPT 的引入結合了 NLP 中的兩個關鍵思想：17新穎高效的 Transformer 解碼器架構和遷移學習。在該設置中，模型通過根據先前的單詞預測下一個單詞來進行預訓練。

該模型在 BookCorpus 上進行了訓練，並在分類等下游任務上取得了很好的效果。

GPT-2

受簡單且可擴展的預訓練方法成功的啟發，對原始模型和訓練集進行了升級以生成 GPT-2。18 該模型能夠生成長序列的連貫文本。出於對可能的誤用的擔憂，該模型以分階段的方式發布，首先發布較小的模型，然後發布完整的模型。

像

GPT-2 這樣的 CTRL 模型可以繼續輸入序列（也稱為提示）。然而，用戶幾乎無法控制生成序列的樣式。Conditional Transformer Language (CTRL) 模型通過在序列的開頭添加“控制標記”來解決此問題。19這些允許控制生成文本的樣式，從而實現多樣化生成。

16 A. Wang 等人，“SuperGLUE：通用語言理解系統的粘性基準”，
(2019)。

17 A. Radford 等人，“通過生成式預訓練改善語言理解”，開放人工智能 (2018)。

18 A. Radford 等人，“語言模型是無監督的多任務學習者”，開放人工智能 (2019)。

19 NS Keskar 等人，“CTRL：用於可控發電的條件轉換器語言模型”，(2019)。

GPT-3

在將 GPT 成功擴展到 GPT-2 之後，對不同規模的語言模型行為的全面分析表明，存在簡單的幕律來控制計算、數據集大小、模型大小和性能之間的關係²⁰。受這些見解的啟發，GPT-2 被放大了 100 倍以產生具有 1750 億個參數的 GPT-3。²¹

除了能夠生成令人印象深刻的逼真文本段落外，該模型還展示了少樣本學習能力：通過一些新任務的示例，例如將文本轉換為代碼，該模型能夠在新示例上完成任務。OpenAI 並沒有開源這個模型，而是通過 OpenAI API 提供了一個接口。

GPT-Neo/GPT-J-6B

GPT-Neo 和 GPT-J-6B 是由 EleutherAI 訓練的類 GPT 模型，一群旨在重新創建和發布 GPT-3 比例模型的研究人員。²² 目前的模型是完整的 1750 億參數模型的較小變體，具有 1.3、2.7 和 60 億參數，並且與 OpenAI 提供的更小的 GPT-3 模型。

變形金剛生命之樹的最後一個分支是編碼器-解碼器模型。讓我們來看看。

編碼器-解碼器分支

儘管使用單個編碼器或解碼器堆棧構建模型已變得很普遍，但 Transformer 架構的多種編碼器-解碼器變體在 NLU 和 NLG 領域都有新穎的應用：

T5

T5 模型通過將所有 NLU 和 NLG 任務轉換為文本到文本任務來統一它們。²³ 所有任務都被構造為序列到序列任務，其中採用編碼器-解碼器架構是很自然的。例如，對於文本分類問題，這意味著文本用作編碼器輸入，解碼器必須將標籤生成為普通文本而不是類。我們將在第 6 章中更詳細地討論這一點。T5 架構使用原始的 Transformer 架構。使用大型爬取的 C4 數據集，通過翻譯所有的

20 J. Kaplan 等人，“神經語言模型的縮放法則”，(2020)。

21 T. Brown 等人，“語言模型是小概率學習者”，(2020)。

22 S. Black 等人，“GPT-Neo：使用 Mesh-TensorFlow 進行大規模自回歸語言建模”，(2021); B. Wang 和 A. Komatsuzaki，“GPT-J-6B：一個 60 億參數自回歸語言模型”，(2021)。

23 C. Raffel 等人，“使用統一的文本到文本轉換器探索遷移學習的局限性”，(2019)。

他們到文本到文本的任務。具有 110 億個參數的最大模型在多個基準測試中產生了最先進的結果。

BART

BART 在編碼器-解碼器架構中結合了 BERT 和 GPT 的預訓練過程。²⁴輸入序列經歷了幾種可能的轉換之一，從簡單的掩碼到句子排列、標記刪除和文檔旋轉。這些修改後的輸入通過編碼器傳遞，解碼器必須重建原始文本。這使得模型更加靈活，因為它可以用於 NLU 和 NLG 任務，並且它在兩者上都實現了最先進的性能。

M2M-100 通

常為一種語言對和翻譯方向建立翻譯模型。當然，這不會擴展到多種語言，此外，語言對之間可能存在共享知識，可用於稀有語言之間的翻譯。M2M-100 是第一個可以在 100 種語言中的任何一種之間進行翻譯的翻譯模型。²⁵這允許在稀有和代表性不足的語言之間進行高質量的翻譯。該模型使用前綴標記（類似於特殊的[CLS]標記）來指示源語言和目標語言。

BigBird

由於注意機制的二次內存需求，Transformer 模型的一個主要限制是最大上下文大小。BigBird 通過使用線性擴展的稀疏注意力形式解決了這個問題。²⁶這允許將上下文從大多數 BERT 模型中的 512 個令牌大幅擴展到 BigBird 中的 4,096 個。這在需要保留長依賴性的情況下特別有用，例如在文本摘要中。



我們在本節中看到的所有模型的預訓練檢查點都可以在 [Hugging Face Hub](#) 上找到並且可以根據您使用 Transformers 的用例進行微調，如前一章所述。

結論

在本章中，我們從 Transformer 架構的核心開始，深入探討了自註意力，隨後我們添加了所有必要的部分來構建一個

²⁴ M. Lewis 等人，“[BART :用於自然語言生成的去噪序列到序列預訓練，翻譯和理解](#)”，(2019)。

²⁵ A. Fan 等人，“[超越以英語為中心的多語言機器翻譯](#)”，(2020)。

²⁶ M. Zaheer 等人，“[大鳥 :更長序列的變形金剛](#)”，(2020)。

變壓器編碼器模型。我們為標記和位置信息添加了嵌入層，我們構建了一個前饋層來補充注意力頭，最後我們在模型主體中添加了一個分類頭來進行預測。我們還了解了 Transformer 架構的解碼器端，並以最重要的模型架構的概述結束本章。

現在您已經更好地理解了基本原理，讓我們超越簡單的分類並構建多語言命名實體識別模型。

第 4 章

多語言命名實體識別

到目前為止，在本書中我們已經應用轉換器來解決英語語料庫的 NLP 任務。但是當您的文檔是用希臘語、斯瓦希里語或克林貢語編寫的時，您會怎麼做？一種方法是在 Hugging Face Hub 中搜索合適的預訓練語言模型，並根據手頭的任務對其進行微調。然而，這些預訓練模型往往只存在於“高資源”語言，如德語、俄語或普通話，其中有大量的網絡文本可用於預訓練。當你的語料庫是多語言時，另一個常見的挑戰出現了：在生產中維護多個單語言模型對你或你的工程團隊來說沒有任何樂趣。

幸運的是，有一類多語言轉換器可以解決這個問題。與 BERT 一樣，這些模型使用掩碼語言建模作為預訓練目標，但它們是在超過 100 種語言的文本上聯合訓練的。通過對跨多種語言的龐大語料庫進行預訓練，這些多語言轉換器可實現零樣本跨語言遷移。這意味著在一種語言上微調的模型可以應用於其他語言而無需任何進一步的訓練！這也使得這些模型非常適合“語碼轉換”，即說話者在一次對話的上下文中在兩種或多種語言或方言之間交替。

在本章中，我們將探討如何對稱為 XLM-RoBERTa（在第3 章中介紹）¹的單個轉換器模型進行微調，以跨多種語言執行命名實體識別（用於識別我們在第1章組翻譯位置等實體）。這些實體可用於各種應用，例如從公司文檔中獲得洞察力、提高搜索引擎的質量，或者簡單地從語料庫構建結構化數據庫。

¹ A. Conneau 等人，“[大規模無監督跨語言表徵學習](#)”，(2019)。

對於本章，我們假設我們要為位於瑞士的客戶執行 NER，那裡有四種國家語言（英語通常作為它們之間的橋樑）。讓我們首先為這個問題獲取一個合適的多語言語料庫。



零樣本遷移或零樣本學習通常是指在一組標籤上訓練模型然後在另一組標籤上對其進行評估的任務。在 transformer 的上下文中，零樣本學習也可能指的是像 GPT-3 這樣的語言模型在甚至沒有微調的下游任務上進行評估的情況。

數據集

在本章中，我們將使用稱為 WikiANN 或 PAN-X 的多語言編碼器跨語言傳輸評估 (XTREME) 基準的一個子集。²該數據集包含多種語言的維基百科文章，包括瑞士最常用的四種語言：德語 (62.9%)、法語 (22.9%)、意大利語 (8.4%) 和英語 (5.9%)。每篇文章都以“從內到外”(IOB2) 格式標註 LOC (位置)、PER (人) 和 ORG (組織) 標籤。在這種格式中，B-前綴表示實體的開始，屬於同一實體的連續令牌被賦予 I-前綴。O 標籤表示令牌不屬於任何實體。例如下面的句子：

Jeff Dean 是加利福尼亞谷歌的一名計算機科學家，
將以 IOB2 格式進行標記，如表 4-1 所示。

表 4-1。用命名實體註釋的序列示例

代幣	Je		Dean	是加利福尼亞谷歌的一名計算機科學家				
標籤	B-PER	I-PER	OOO		歐	O B-ORG	O B-地方	

要在 XTREME 中加載 PAN-X 子集之一，我們需要知道要傳遞給 `load_dataset()` 函數的數據集配置。無論何時處理具有多個域的數據集，都可以使用 `get_dataset_config_names()` 函數找出哪些子集可用：

² J. Hu 等人，“XTREME：用於評估跨語言泛化的大規模多語言多任務基準”，(2020); X. Pan 等人，“282 種語言的跨語言名稱標記和鏈接”，計算語言學協會第 55 屆年會論文集 1 (2017 年 7 月) :1946–1958 ,<http://dx.doi.org/10.18653/v1/P17-1178>。

從數據集導入get_dataset_config_names

```
xtreme_subsets = get_dataset_config_names( xtreme )
print(f XTREME 有 {len(xtreme_subsets)} 配置 )
```

XTREME有183種配置

哇 ,那是很多配置 !讓我們通過查找以 “PAN”開頭的配置來縮小搜索範圍 :

```
panx_subsets = [s for s in xtreme_subsets if s.startswith( PAN )] pnx_subsets[:3]
```

```
[ PAN-X.af , PAN-X.ar , PAN-X.bg ]
```

好的 ,我們似乎已經確定了 PAN-X 子集的語法 :每個子集都有一個兩個字母的後綴 ,似乎是 ISO 639-1 語言代碼 。這意味著要加載德語語料庫 ,我們將de代碼傳遞給load_dataset()的 name參數 ,如下所示 :

從數據集導入load_dataset

```
load_dataset( xtreme ,name= PAN-X.de )
```

為了製作真實的瑞士語料庫 ,我們將根據口語比例從 PAN-X 中抽取德語(de)、法語(fr)、意大利語(it)和英語(en) 語料庫 。

這將造成語言不平衡 ,這在現實世界的數據集中非常普遍 ,由於缺乏精通該語言的領域專家 ,因此獲取少數民族語言的標記示例可能會很昂貴 。這個不平衡的數據集將模擬處理多語言應用程序時的常見情況 ,我們將看到如何構建適用於所有語言的模型 。

為了跟蹤每種語言 ,讓我們創建一個 Python defaultdict ,它將語言代碼存儲為鍵 ,將類型為 DatasetDict 的 PAN-X 語料庫存儲為值 :

從集合導入defaultdict從數據集導入DatasetDict

```
langs = [ de , fr , it , en ]fracs
= [0.629, 0.229, 0.084, 0.059]
# 如果鍵不存在則返回 DatasetDict pnx_ch =
defaultdict(DatasetDict)
```

對於 lang , zip 中的 frac (langs , fracs) :

```
# 加載單語語料庫ds =
load_dataset( xtreme ,name=f PAN-X.{lang} )
# 根據語音比例對每個拆分進行隨機播放和下採樣for split in ds: pnx_ch[lang][split] =
(ds[split].shuffle(seed=0).select(range(int(frac * ds[split].num_rows) )) )
```

在這裡，我們使用了shuffle()方法來確保我們不會意外地偏向我們的數據集拆分，而select()允許我們根據fracs中的值對每個語料庫進行下採樣。讓我們通過訪問Dataset.num_rows屬性看看訓練集中每種語言有多少示例：

將熊貓導入為pd

```
pd.DataFrame({lang: [panx_ch[lang][train].num_rows] for lang in langs},
              index=[ 訓練樣例數 ])
```

	德	fr	它	恩
訓練樣例數	12580	4580	1680	1180

按照設計，我們的德語示例比所有其他語言的示例總和還要多，因此我們將使用它作為起點，從中執行到法語、意大利語和英語的零樣本跨語言遷移。讓我們檢查德國語料庫中的一個例子：

```
element = pnx_ch[ de ][ train ][0] for key,
value in element.items(): print(f {key}: {value} )

langs: [ de , de ]
ner_tags: [0,0,0,0,5,6,0,0,5,5,6,0] 標記:[ 2.000 , Einwohnern , an , der , Danziger ,
Bucht , in , der , polnischen , Woiwodschaft , Pommern , . ]
```

與我們之前遇到的Dataset對像一樣，我們示例中的鍵對應於Arrow表的列名，而值表示每列中的條目。特別是，我們看到ner_tags列對應於每個實體到類ID的映射。這對人眼來說有點神秘，所以讓我們用熟悉的LOC、PER和ORG標籤創建一個新列。為此，首先要注意的是我們的Dataset對像有一個features屬性，它指定了與每一列相關聯的底層數據類型：

對於鍵，值在panx_ch[de][train].features.items(): print(f {key}: {value})

```
標記 :序列 (特徵=值 (dtype= string , id=None) ,長度=-1 ,id=None) ner_tags :序列 (特徵=ClassLabel
(num_classes=7 ,names=[ O , B-每 ,
I-PER , B-ORG , I-ORG , B-LOC , I-LOC ], names_file=None, id=None), length=-1,
id=None) langs :序列 (特徵=值 (dtype= string , id=None) ,長度=-1 ,id=None)
```

Sequence類指定該字段包含一個特徵列表，在ner_tags的情況下對應於一個ClassLabel特徵列表。讓我們從訓練集中挑選出這個特徵，如下所示：

```
tags = pannx_ch[ de ][ train ].features[ ner_tags ].feature打印 (標籤)
```

```
ClassLabel(num_classes=7, names=[ O , B-PER , I-PER , B-ORG , I-ORG ,
B-LOC , I-LOC ], names_file=None, id=None)
```

我們可以使用我們在第 2 章中遇到的 ClassLabel.int2str() 方法在我們的訓練集中創建一個新列，其中包含每個標籤的類名。我們將使用 map() 方法返回一個字典，其鍵對應於新的列名

以及作為類名列表的值：

```
def create_tag_names (批處理) :
    返回[ ner_tags_str : [tags.int2str(idx) for idx in batch[ ner_tags ]]]}

panx_de = pannx_ch[ de ].map(create_tag_names)
```

現在我們有了人類可讀格式的標籤，讓我們看看訓練集中第一個示例的標記和標籤如何對齊：

```
de_example = pannx_de[ train ][0]
pd.DataFrame([de_example[ tokens ], de_example[ ner_tags_str ]], [ Tokens ,
Tags ])
```

	0	*	2 3 4	*	*	6 7	*	*	10	11
標籤	代幣	2.000	Einwohnern	an der	Danziger	Bucht	in der	polnischen	Woiwodschaft	Pommern。
	歐	歐	O O	B-LOC	I-LOC	OO	集團	集團	I-LOC	歐

LOC 標籤的存在是有道理的，因為句子 “2,000 Einwohnern an der Danziger Bucht in der polnischen Woiwodschaft Pommern” 在英語中的意思是 “2,000 inhabitants at the Polish voivodeship of Pomerania”，而 Gdansk Bay 是一個海灣在波羅的海，而 “voivodeship” 對應於波蘭的一個州。

為了快速檢查我們在標籤中沒有任何異常的不平衡，讓我們計算每個實體在每個拆分中的頻率：

從集合導入計數器

```
split2freqs = defaultdict(Counter) for split,
dataset in pannx_de.items(): for row in
dataset[ ner_tags_str ]: for tag in row: if
tag.startswith( B ): tag_type =
tag.split( - )[1] split2freqs[split]
[tag_type] += 1

pd.DataFrame.from_dict(split2freqs, orient= index )
```

	組織位置	
驗證2683 31	72 2893	

	組織位置	
測試	2573 3180 3071	
火車	5366 6186 5810	

這看起來不錯 每個拆分的PER、LOC和ORG頻率分佈大致相同，因此驗證集和測試集應該可以很好地衡量我們的NER標記器的泛化能力。接下來，讓我們看看一些流行的多語言轉換器，以及如何調整它們來處理我們的NER任務。

多語言變形金剛

多語言轉換器涉及與單語言轉換器相似的架構和訓練過程，除了用於預訓練的語料庫包含多種語言的文檔。這種方法的一個顯著特點是，儘管沒有收到明確的信息來區分語言，但由此產生的語言表示能夠很好地跨語言泛化，用於各種下游任務。在某些情況下，這種執行跨語言遷移的能力可以產生與單語言模型相媲美的結果，從而避免了為每種語言訓練一個模型的需要！

為了衡量NER跨語言遷移的進展，CoNLL-2002和CoNLL-2003數據集通常用作英語、荷蘭語、西班牙語和德語的基準。該基準由與PAN-X相同的LOC、PER和ORG類別註釋的新聞文章組成，但它包含一個額外的MISC標籤，用於不屬於前三組的雜項實體。多語言轉換器模型通常以三種不同的方式進行評估：

恩

微調英語訓練數據，然後評估每種語言的測試放。

每個

微調和評估單語測試數據以衡量每種語言的性能。

全部

對所有訓練數據進行微調，以對每種語言的測試集進行評估。

我們將對我們的NER任務採用類似的評估策略，但首先我們需要選擇一個模型進行評估。第一個多語言轉換器是mBERT，它使用與BERT相同的架構和預訓練目標，但將來自多種語言的維基百科文章添加到預訓練語料庫中。從那時起，mBERT就被XLM-RoBERTa（或簡稱XLM-R）所取代，因此這就是我們將在本章中考慮的模型。

正如我們在第 3 章中看到的，XLM-R 僅使用 MLM 作為 100 種語言的預訓練目標，但與它的前輩相比，它的預訓練語料庫規模巨大：維基百科轉儲每種語言和 2.5 TB 的 Common Crawl 數據來自網絡。這個語料庫比早期模型中使用的語料庫大幾個數量級，並且為緬甸語和斯瓦希里語等低資源語言提供了顯著的信號提升，其中只有少量維基百科文章

存在。

模型名稱中的 RoBERTa 部分指的是預訓練方法與單語 RoBERTa 模型相同。RoBERTa 的開發人員在 BERT 的幾個方面進行了改進，特別是通過完全刪除下一個句子預測任務。³ XLM-R 還刪除了 XLM 中使用的語言嵌入，並使用 SentencePiece 直接標記原始文本。⁴ 除了其多語言特性外，一個值得注意的 XLM-R 和 RoBERTa 之間的區別在於各自詞彙表的大小：250,000 個標記對 55,000 個！

XLM-R 是多語言 NLU 任務的絕佳選擇。在下一節中，我們將探討它如何有效地跨多種語言進行標記化。

仔細觀察代幣化

XLM-R 沒有使用 WordPiece 分詞器，而是使用了一個名為 SentencePiece 的分詞器，它在所有一百種語言的原始文本上進行了訓練。為了了解 SentencePiece 與 WordPiece 的比較，讓我們以通常的方式使用 Transformers 加載 BERT 和 XLM-R 分詞器：



從變形金剛導入 AutoTokenizer

```
bert_model_name = "bert-base-cased"
xlmr_model_name = "xlm-roberta-base"
bert_tokenizer = AutoTokenizer.from_pretrained(bert_model_name)
xlmr_tokenizer = AutoTokenizer.from_pretrained(xlmr_model_name)
```

通過對一小段文本進行編碼，我們還可以檢索每個模型在預訓練期間使用的特殊標記：

```
text = Jack Sparrow 喜歡紐約！ bert_tokens =
bert_tokenizer(text).tokens() xlmr_tokens =
xlmr_tokenizer(text).tokens()
```

BERT [CLS] 傑克	溫泉	##rrow 愛紐約			!	[SEP] 無
---------------	----	------------	--	--	---	---------

3 Y. Liu 等人，“RoBERTa：一種穩健優化的 BERT 預訓練方法”，(2019)。

4 T. Kudo 和 J. Richardson，“SentencePiece：一種用於神經文本處理的簡單且與語言無關的子詞標記器和去標記器”，(2018)。

XLM-R	< s >	傑克斯	帕排		愛		紐約	！		</ s >
-------	-------	-----	----	--	---	--	----	---	--	--------

在這裡我們看到 XLM-R 使用 `<s>` 和 `</s>` 來表示序列的開始和結束，而不是 BERT 用於句子分類任務的 `[CLS]` 和 `[SEP]` 標記。這些標記是在標記化的最後階段添加的，我們將在接下來看到。

Tokenizer Pipeline 到目前為止，

我們已經將標記化視為將字符串轉換為整數的單個操作，我們可以將其傳遞給模型。這並不完全準確，如果仔細觀察就會發現它實際上是一個完整的處理流水線，通常由四個步驟組成，如圖 4-1 所示。



圖 4-1。令牌化管道中的步驟

讓我們仔細看看每個處理步驟，並用例句 “Jack Sparrow loves New York！” 來說明它們的效果：

normalization

此步驟對應於您應用於原始字符串以使其“更乾淨”的一組操作。常見操作包括去除空格和刪除重音字符。**Unicode規範化**是另一種常見的規範化操作，被許多分詞器應用來處理這樣一個事實，即通常存在多種方式來書寫相同的字符。這會使“相同”字符串的兩個版本（即具有相同的抽象字符序列）看起來不同；NFC、NFD、NFKC 和 NFKD 等 Unicode 規範化方案取代了使用標準格式書寫相同字符的各種方式。規範化的另一個例子是小寫。如果期望模型只接受和使用小寫字符，則可以使用此技術來減少所需的詞彙量。標準化後，我們的示例字符串看起來像“jack sparrow loves new york!”。

pretokenization

此步驟將文本拆分為更小的對象，這些對像給出了訓練結束時標記的上限。考慮這一點的一個好方法是，預分詞器會將你的文本拆分為“單詞”，而你的最終標記將是這些單詞的一部分。對於允許這樣做的語言（英語、德語和許多印歐語言），字符串通常可以根據空格和標點符號拆分為單詞。例如，這一步可能會改變我們的 [“jack”、“sparrow”、“loves”、“new”、“york”、“!”]。這些詞更容易拆分成

在管道的下一步中使用字節對編碼 (BPE) 或 Unigram 算法的子詞。然而，拆分成“單詞”並不總是一個微不足道的確定性操作，甚至不是一個有意義的操作。例如，在中文、日文或韓文等語言中，以印歐語詞等語義單位對符號進行分組可能是具有多個同等有效組的非確定性操作。在這種情況下，最好不要對文本進行預標記，而是使用特定於語言的庫進行預標記。

分詞器模型一旦

輸入文本被規範化和預分詞，分詞器就會對單詞應用子詞拆分模型。這是管道的一部分，需要在你的語料庫上進行訓練（或者如果你使用預訓練的分詞器，則已經訓練過）。模型的作用是將詞拆分成子詞以減少詞彙表的大小，並儘量減少詞彙外標記的數量。存在幾種子詞標記化算法，包括 BPE、Unigram 和 WordPiece。例如，在應用分詞器模型後，我們的運行示例可能看起來像[jack, spa, rrow, loves, new, york, !]。請注意，此時我們不再有字符串列表，而是整數列表（輸入 ID）；為了使示例更加清晰，我們保留了單詞但去掉了引號以指示轉換。

後處理 這是標

記化管道的最後一步，其中可以對標記列表應用一些額外的轉換。例如，在標記索引的輸入序列的開頭或結尾添加特殊標記。例如，BERT 風格的分詞器會添加分類和分隔符：[CLS, jack, spa, rrow, loves, new, york, !, SEP]。然後可以將該序列（回想一下，這將是一個整數序列，而不是您在此處看到的標記）提供給模型。

回到我們對 XLM-R 和 BERT 的比較，我們現在了解到 SentencePiece 在後處理步驟中添加了<s>和<\s>而不是[CLS]和[SEP]（作為慣例，我們將繼續在圖形插圖中使用[CLS]和[SEP]）。讓我們回到 SentencePiece 分詞器，看看它有什麼特別之處。

SentencePiece 分詞器

SentencePiece 分詞器基於一種稱為 Unigram 的子詞分段，並將每個輸入文本編碼為 Unicode 字符序列。最後一個功能對於多語言語料庫特別有用，因為它允許 SentencePiece 不關心重音、標點符號以及許多語言（如日語）沒有空白字符的事實。SentencePiece 的另一個特點是空格被分配了 Unicode 符號 U+2581，或字符，也稱為下四分之一塊字符。這使 SentencePiece 能夠去標記一個

沒有歧義且不依賴於特定語言的預分詞器的序列。例如，在上一節的示例中，我們可以看到 WordPiece 丟失了“York”和“!”之間沒有空格的信息。相比之下，SentencePiece 保留了標記化文本中的空格，因此我們可以毫無歧義地轉換回原始文本：

```
.join(xlmr_tokens).replace(u'\\u2581', ' ')
<s> Jack Sparrow 喜歡紐約 !</s>
```

現在我們了解了 SentencePiece 的工作原理，讓我們看看如何以適合 NER 的形式對我們的簡單示例進行編碼。要做的第一件事是加載帶有標記分類頭的預訓練模型。但不是直接從變形金剛加載這個頭部，而是我們自己構建它！深入了解變形金剛



API，我們只需幾步就可以做到這一點。

用於命名實體識別的轉換器

在第 2 章中，我們看到對於文本分類，BERT 使用特殊的[CLS]標記來表示整個文本序列。然後，該表示通過全連接層或密集層輸出所有離散標籤值的分佈，如圖4-2 所示。

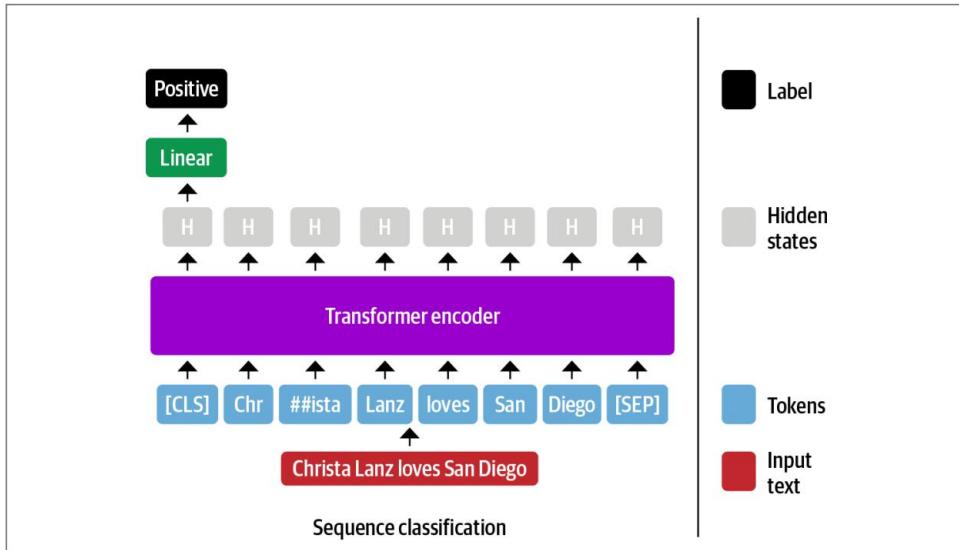


圖 4-2 微調基於編碼器的轉換器以進行序列分類

BERT 和其他僅編碼器的轉換器對 NER 採取了類似的方法，不同之處在於每個單獨的輸入令牌的表示被饋送到相同的全連接層以輸出令牌的實體。出於這個原因，NER 通常被定義為標記分類任務。該過程類似於圖 4-3 中的圖表。

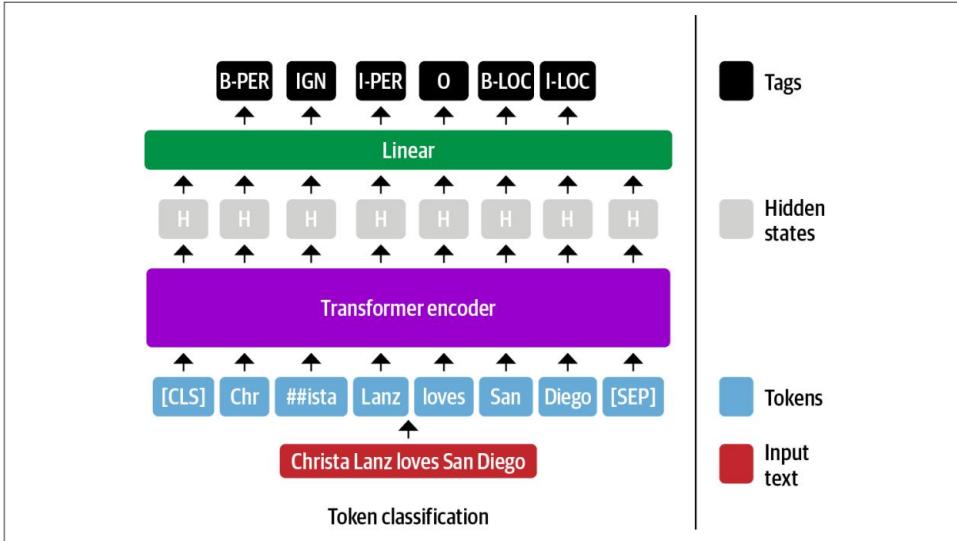


圖 4-3 ◦微調基於編碼器的轉換器以進行命名實體識別

到目前為止一切順利，但我們應該如何處理標記分類任務中的子詞？

例如，圖 4-3 中的名字 “Christa” 被標記為子詞 “Chr” 和 “##ista”，那麼應該為哪個（哪些）分配 B-PER 標籤？

在 BERT 論文 5 中，作者將此標籤分配給第一個子詞（在我們的示例中為 “Chr”）並忽略了以下子詞（“##ista”）。這是我們將在此處採用的約定，我們將用 IGN 指示忽略的子詞。我們稍後可以在後處理步驟中輕鬆地將第一個子詞的預測標籤傳播到後續子詞。我們也可以選擇通過為其分配 B-LOC 標籤的副本來包含 “##ista” 子詞的表示，但這違反了 IOB2 格式。

幸運的是，我們在 BERT 中看到的所有架構方面都適用於 XLM-R，因為它的架構基於與 BERT 相同的 RoBERTa！接下來我們將看到 Transformers 如何通過微小的修改支持許多其他任務。

5 J. Devlin 等人，“BERT：用於語言理解的深度雙向轉換器的預訓練”，(2018)。

Transformers 模型類剖析

 Transformers 是圍繞每個架構和任務的專用類組織的。

與不同任務關聯的模型類根據<Model Name>For<Task>約定命名，或者在使用AutoModel 類時根據 AutoModelFor<Task>命名。

但是，這種方法有其局限性，為了激發對 Transformers API 的深入研究，請考慮以下場景。

 假設你有一個很好的想法，可以用 transformer 模型解決你腦海中縈繞已久的 NLP 問題。因此，您安排與老闆會面，並通過精心製作的 PowerPoint 演示文稿宣傳，如果您最終能解決問題，就可以增加部門的收入。對你豐富多彩的演講和關於利潤的談論印象深刻，你的老闆慷慨地同意給你一周的時間來建立一個概念驗證。對結果感到滿意，您立即開始工作。你啟動你的 GPU 並打開一個筆記本。您執行from transformers import BertForTaskXY（請注意，TaskXY是您想要解決的假想任務），當可怕的紅色填滿您的屏幕時，顏色從您的臉上消失：ImportError: cannot import name BertForTaskXY。哦不，您的用例沒有 BERT 模型！如果您必須自己實施整個模型，您如何在一週內完成項目？！你應該從哪裡開始？

不要恐慌！ Transformers 旨在使您能夠輕鬆地為您的特定用例擴展現有模型。您可以從預訓練模型加載權重，並且可以訪問特定於任務的輔助函數。這使您能夠以極少的開銷為特定目標構建自定義模型。在本節中，我們將了解如何實現我們自己的自定義模型。

身體和頭部

使 Transformer 如此通用的主要概念是將架構分為主體和頭部（正如我們在第 1 章中看到的）。我們已經看到，當我們從預訓練任務切換到下游任務時，我們需要將模型的最後一層替換為適合該任務的一層。最後一層稱為模型頭，這是特定於任務的部分。模型的其餘部分稱為主體；它包括與任務無關的令牌嵌入和轉換器層。這種結構也反映在 Transformers 代碼中：模型的主體在BertModel或GPT2Model等返回最後一層隱藏狀態的類中實現。BertForMaskedLM 或BertForSequence 分類等任務特定模型使用基礎模型並在隱藏狀態之上添加必要的頭部，如圖4-4 所示。

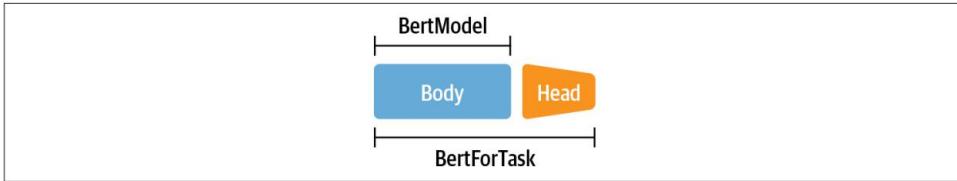


圖 4-4。BertModel 類只包含模型的主體，而 Bert For<Task> 類將主體與給定任務的專用頭部結合在一起。

正如我們接下來將看到的，這種身體和頭部的分離使我們能夠為任何任務構建一個自定義頭部，並將其安裝在預訓練模型之上。

為令牌分類創建自定義模型讓我們完成為 XLM R 構建自定義令

牌分類頭的練習。由於 XLM-R 使用與 RoBERTa 相同的模型架構，我們將使用 RoBERTa 作為基本模型，但增加了特定於 XLM-R。請注意，這是一項教育練習，旨在向您展示如何為您自己的任務構建自定義模型。

對於令牌分類，`XLMRobertaForTokenClassification` 類已經存在，您可以從 `Transformers` 導入該類。如果需要，可以跳到下一節並直接使用該節。

首先，我們需要一個數據結構來代表我們的 XLM-R NER 標註器。作為第一個猜測，我們需要一個配置對象來初始化模型和一個 `forward()` 函數來生成輸出。讓我們繼續構建用於標記分類的 XLM-R 類：

```

從變壓器導入torch.nn作為nn
從transformers.modeling_outputs導入XLMRobertaConfig
從transformers.models.roberta.modeling_roberta導入TokenClassifierOutput從
transformers.models.roberta.modeling_roberta導入RobertaModel導入RobertaPreTrainedModel
  
```

`XLMRobertaForTokenClassification` 類 (`RobertaPreTrainedModel`) :

配置類 = `XLMRobertaConfig`

```

def __init__(self, config):
    super().__init__(config)
    self.num_labels = config.num_labels # 加載模型
    主體self.roberta = RobertaModel(config,
        add_pooling_layer=False)
    # 設置token分類頭self.dropout =
    nn.Dropout(config.hidden_dropout_prob) self.classifier =
    nn.Linear(config.hidden_size, config.num_labels)
    # 加載並初始化權重self.init_weights()
  
```

```

def forward(self, input_ids=None, attention_mask=None, token_type_ids=None,
  
```

```

    標籤=無， **kwargs) :
# 使用模型主體獲取編碼器表示outputs = self.roberta(input_ids,
attention_mask=attention_mask, token_type_ids=token_type_ids, **kwargs)

# 將分類器應用於編碼器表示sequence_output =
self.dropout(outputs[0]) logits = self.classifier(sequence_output)

# 計算損失
loss = None如果
標籤不是None :
    loss_fct = nn.CrossEntropyLoss() loss =
    loss_fct(logits.view(-1, self.num_labels), labels.view(-1))
# 返回模型輸出對象return
TokenClassifierOutput(loss=loss, logits=logits,
hidden_states=outputs.hidden_states,
attentions=outputs.attentions)

```

config_class確保在我們初始化新模型時使用標準的 XLM-R 設置。如果要更改默認參數，可以通過覆蓋配置中的默認設置來實現。使用super()方法，我們調用RobertaPreTrainedModel類的初始化函數。這個抽像類處理預訓練權重的初始化或加載。然後我們加載我們的模型主體，即RobertaModel，並使用我們自己的分類頭對其進行擴展，該分類頭由一個 dropout 和一個標準前饋層組成。請注意，我們設置add_pooling_layer=False以確保返回所有隱藏狀態，而不僅僅是與[CLS]令牌關聯的隱藏狀態。最後，我們通過調用繼承自RobertaPreTrainedModel 的init_weights()方法來初始化所有權重，該方法將為模型主體加載預訓練權重並隨機初始化我們的標記分類頭的權重。

剩下要做的唯一一件事就是使用forward()方法定義模型在前向傳遞中應該做什麼。在正向傳遞過程中，數據首先通過模型主體饋送。有許多輸入變量，但我們現在唯一需要的是input_ids和attention_mask。隱藏狀態是模型主體輸出的一部分，然後通過丟棄層和分類層提供。如果我們在前向傳遞中也提供標籤，我們可以直接計算損失。如果有註意力掩碼，我們需要做更多的工作以確保我們只計算未掩碼標記的損失。最後，我們將所有輸出包裝在一個TokenClassifierOutput 對像中，該對象允許我們訪問前面章節中熟悉的命名元組中的元素。

通過只實現一個簡單類的兩個函數，我們可以構建自己的自定義轉換器模型。由於我們繼承自PreTrainedModel，我們可以立即訪問所有有用的 Transformer 實用程序，例如from_pretrained()！讓我們看看如何將預訓練的權重加載到我們的自定義模型中。

加載自定義模型現在我們準備加

載我們的令牌分類模型。我們需要提供模型名稱之外的一些額外信息，包括我們將用於標記每個實體的標籤以及每個標籤與 ID 的映射，反之亦然。所有這些信息都可以從我們的tags變量中導出，作為一個ClassLabel對象，它有一個names屬性，我們可以使用它來導出映射：

```
index2tag = {idx : idx的標籤 ,枚舉中的標籤 (tags.names) } tag2index = {tag : idx的
idx ,枚舉中的標籤 (tags.names) }
```

我們將在第3章中遇到的AutoConfig對像中存儲這些映射和tags.num_classes屬性。將關鍵字參數傳遞給from_pretrained()方法會覆蓋默認值：

從變壓器導入AutoConfig

```
xlmr_config = AutoConfig.from_pretrained(xlmr_model_name,
                                         num_labels=tags.num_classes,
                                         id2label=index2tag, label2id=tag2index)
```

AutoConfig類包含模型架構的藍圖。當我們使用AutoModel.from_pretrained(model_ckpt)加載模型時，與該模型關聯的配置文件會自動下載。但是，如果我們想要修改類的數量或標籤名稱等內容，那麼我們可以先使用我們想要自定義的參數加載配置。

現在，我們可以像往常一樣使用帶有附加配置參數的from_pretrained()函數加載模型權重。請注意，我們沒有在自定義模型類中實現加載預訓練權重；我們通過從RobertaPreTrainedModel繼承來免費獲得這個：

進口手電筒

```
device = torch.device( cuda if torch.cuda.is_available() else cpu ) xlmr_model =
(XLMRobertaForTokenClassification
.from_pretrained(xlmr_model_name, config=xlmr_config).to(device))
```

為了快速檢查我們是否正確初始化了分詞器和模型，讓我們測試對已知實體的小序列的預測：

```
input_ids = xlmr_tokenizer.encode(text, return_tensors= pt )
pd.DataFrame([xlmr_tokens, input_ids[0].numpy()], index=[ 令牌 , 輸入 ID ])
```

	0	1	2	3	4	5 6	7	8 9	
Tokens <s>	Jack Spar	行			愛紐約！				</s>
輸入 ID 0		21763	37456	15555	5161	7 2356	5758	38 2	

正如您在此處看到的，開始`<S>`和結束`</S>`標記分別被賦予 ID 0 和 2。

最後，我們需要將輸入傳遞給模型並通過使用 `argmax` 來提取預測，以獲得每個標記最可能的類別：

```
outputs = xlmr_model(input_ids.to(device)).logits predictions =
torch.argmax(outputs, dim=-1) print(f Number of tokens in sequence:
{len(xlmr_tokens)} ) print(f 形狀輸出 :{outputs.shape} )
```

序列中的標記數：10 輸出形狀：`torch.Size([1, 10, 7])`

在這裡我們看到 `logits` 的形狀為`[batch_size, num_tokens, num_tags]`，每個標記在七個可能的 NER 標籤中被賦予一個 `logit`。通過枚舉序列，我們可以快速查看預訓練模型的預測結果：

```
preds = [tags.names[p] for p in predictions[0].cpu().numpy()] pd.DataFrame([xlmr_tokens,
preds], index=[ Tokens , Tags ])
```

	0	1	2	3	4	5	6	7	8	9
Tokens	<S> Jack	Spar	行		愛		紐約！			</S>
標籤	歐	I-LOC	B-LOC	B-LOC	O		I-LOC	O	歐	I-LOC

毫不奇怪，我們的具有隨機權重的令牌分類層還有很多不足之處；讓我們對一些標記數據進行微調以使其更好！在這樣做之前，讓我們將前面的步驟包裝到一個輔助函數中以備後用：

```
def tag_text(text, tags, model, tokenizer): # 獲取帶有特殊字符的標記
    tokens = tokenizer(text).tokens()

    # 將序列編碼成 IDs input_ids = xlmr_tokenizer(text,
    return_tensors= pt ).input_ids.to(device)
    # 將預測作為分佈在 7 個可能的類中outputs = model(inputs)[0]

    # 採用 argmax 來獲得每個標記最有可能的類別predictions =
    torch.argmax(outputs, dim=2)
    # 轉換為DataFrame
    preds = [tags.names[p] for p in predictions[0].cpu().numpy()] return pd.DataFrame([tokens,
    preds], index=[ Tokens , Tags ])
```

在我們可以訓練模型之前，我們還需要標記輸入並準備標籤。我們接下來會這樣做。

為 NER 標記文本

現在我們已經確定分詞器和模型可以對單個示例進行編碼，下一步是對整個數據集進行分詞，以便我們可以將其傳遞給 XLM-R 模型進行微調。正如我們在第 2 章中看到的，Datasets 提供了一種使用 `map()` 操作來標記 Dataset 對象的快速方法。 這個與 `map()` 一樣，我們首先需要定義一個

函數（示例：`Dict[str, List]`）`> Dict[str, List]`

其中 `examples` 相當於數據集的一部分，例如 `panx_de[train][10]`。

由於 XLM-R 分詞器返回模型輸入的輸入 ID，我們只需要使用注意力掩碼和標籤 ID 來擴充此信息，這些標籤 ID 編碼有關與每個 NER 標籤相關聯的令牌的信息。

按照 Transformers 文檔中採用的方法，讓我們首先將單詞和標籤收集為普通列表，看看它如何與我們的單個德語示例一起使用：

```
單詞,標籤 = de_example[ tokens ], de_example[ ner_tags ]
```

接下來，我們標記每個單詞並使用 `is_split_into_words` 參數告訴標記器我們的輸入序列已經被拆分成單詞：

```
tokenized_input = xlmr_tokenizer(de_example[ tokens ], is_split_into_words=True)
tokens = xlmr_tokenizer.convert_ids_to_tokens(tokenized_input[ input_ids ]) pd.DataFrame([tokens], index=[令牌])
```

代幣<s>	0	1	2	3	4	5	6	...	18	19	20	21	22	23	24	
2.000 Einwohner	n	a	n	d	...	s	c	h	a	f	m	m	r	e	n	.</s>

在這個例子中，我們可以看到分詞器將 “Einwohner” 拆分為兩個子詞，“Einwohner” 和 “n”。由於我們遵循只有 “Einwohner” 應該與 B-LOC 標籤相關聯的約定，我們需要一種方法來屏蔽第一個子詞之後的子詞表示。幸運的是，`tokenized_input` 是一個包含 `word_ids()` 函數的類，可以幫助我們實現這一點：

```
word_ids = tokenized_input.word_ids()
pd.DataFrame([tokens, word_ids], index=[Tokens, Word IDs])
```

代幣<s>	0	1	2	3	4	5	6	...	18	19	20	21	22	23	24	
2.000 Einwohner	n	a	n	d	...	s	c	h	a	f	m	m	r	e	n	.</s>
單詞	無	0			1	2			...	9		10	10	10	11	無
身份證																

在這裡我們可以看到word_ids已經將每個子詞映射到單詞序列中的相應索引，因此第一個子詞“2.000”被分配了索引0，而“Einwohner”和“n”被分配了索引1（因為

“Einwohnern”是單詞中的第二個單詞）。我們還可以看到像<S>和<\S>這樣的特殊標記被映射到None。讓我們將-100設置為這些特殊標記的標籤以及我們希望在訓練期間屏蔽的子詞：

```
previous_word_idx = None
= []

對於word_ids中的word_idx：
如果word_idx為None或word_idx == previous_word_idx:
    label_ids.append(-100) elif word_idx != previous_word_idx:

    label_ids.append(標籤[word_idx])
    previous_word_idx = word_idx

labels = [index2tag[l] if l != -100 else IGN for l in label_ids] index = [
    Tokens , Word
IDs , Label IDs , Labels ]  

pd.DataFrame([tokens, word_ids, label_ids, labels], index=index)
```

	0	=	20	21	41	51	... 19	20	21	22 23	24
代幣<S>	2.000	Einwohner	n			一個	der ... Po	mmer n			.
字 ID 無 0			=	=	20	51	... 10	10	10	11 11	沒有任何
標籤 ID -100 0		0	-100 0	0	0	... 6	-100	-100 0 -100 -100			
標籤	IGN	歐	歐	點火開關	歐	... I-LOC	點火	點亮點點點			



為什麼我們選擇-100作為ID來屏蔽子詞表示？原因是在PyTorch中，交叉熵損失類torch.nn.CrossEntropyLoss有一個名為ignore_index的屬性，其值為-100。該索引在訓練期間被忽略，因此我們可以使用它來忽略與連續子詞相關聯的標記。

就是這樣！我們可以清楚地看到標籤ID如何與標記對齊，因此讓我們通過定義一個包裝所有邏輯的函數將其擴展到整個數據集：

```
def tokenize_and_align_labels (示例) :
    tokenized_inputs = xlmr_tokenizer (示例[“令牌”]，截斷=真，is_split_into_words=真)

    labels = [] for
    idx, label in enumerate(examples[“ner_tags”]):
        word_ids = tokenized_inputs.word_ids(batch_index=idx) previous_word_idx
        = None label_ids = [] for word_idx in word_ids:
```

如果`word_idx`為`None`或`word_idx == previous_word_idx`: `label_ids.append(-100)`否則 :

```
label_ids.append(標籤[word_idx]) previous_word_idx
= word_idx labels.append(label_ids)

tokenized_inputs[ labels ] =標籤返回tokenized_inputs
```

我們現在擁有了對每個拆分進行編碼所需的所有成分 ,所以讓我們編寫一個可以迭代的函數 :

```
def encode_pnx_dataset (語料庫) :
    返回corpus.map(tokenize_and_align_labels,batched=True,remove_columns=[ langs ,
        ner_tags , tokens ])
```

通過將此函數應用於DatasetDict對象 ,我們在每個拆分中獲得一個編碼的Dataset對象 。讓我們用它來編碼我們的德語語料庫 :

```
panx_de_encoded = encode_pnx_dataset(pnx_ch[ de ])
```

現在我們有了一個模型和一個數據集 ,我們需要定義一個性能指標 。

績效衡量

評估 NER 模型類似於評估文本分類模型 ,通常會報告精度、召回率和F1分數的結果 。唯一的微妙之處在於 ,實體的所有單詞都需要被正確預測 ,才能將預測算作正確 。幸運的是 ,有一個漂亮的庫叫做seqeval專為此類任務而設計 。例如 ,給定一些佔位符 NER 標籤和模型預測 ,我們可以通過 seqeval 的classification_report()函數計算指標 :

從seqeval.metrics導入分類報告

```
y_true=[[ O , O , O , B-MISC , I-MISC , I-MISC , O ],
        [ B-PER , I-PER , O ]]y_pred
=[[ O , O , B-MISC , I-MISC , I-MISC , I-MISC , O ],
        [ "B-PER" 、 "I-PER" 、 "O" ]]打印
(分類報告 (y_true , y_pred )
```

	精確	召回	f1-score	支持
雜項	0.00	0.00	0.00	2
每	1.00	1.00	1.00	2
微平均	0.50	0.50	0.50	2
宏觀平均加權平	0.50	0.50	0.50	2
均	0.50	0.50	0.50	2

正如我們所見 ,seqeval 期望預測和標籤作為列表的列表 ,每個列表對應於我們的驗證或測試集中的單個示例 。整合這些

訓練期間的指標，我們需要一個函數來獲取模型的輸出並將它們轉換為 seqeval 期望的列表。以下通過確保我們忽略與後續子詞關聯的標籤 ID 來解決問題：

將numpy導入為np

```
def align_predictions(predictions, label_ids): preds =
    np.argmax(predictions, axis=2) batch_size, seq_len =
    preds.shape[0], preds_list = [], []

    對於範圍內的batch_idx (batch_size) :
        example_labels, example_preds = [], [] for seq_idx
        in range(seq_len): # 忽略標籤 IDs = -100 if
            label_ids[batch_idx, seq_idx] != -100:
                example_labels.append(index2tag[label_ids[batch_idx]
                    [seq_idx]]) example_preds.append(index2tag[preds[batch_idx][seq_idx]])

        labels_list.append(example_labels)
        preds_list.append(example_preds)

    返回preds_list, labels_list
```

配備性能指標後，我們可以繼續實際訓練模型。

微調 XLM-RoBERTa

我們現在擁有微調模型的所有要素！我們的第一個策略是在 PAN-X 的德語子集上微調我們的基本模型，然後評估其在法語、意大利語和英語上的零樣本跨語言性能。像往常一樣，我們將使用 Transformers Trainer 來處理我們的訓練循環，所以首先我們需要定義



使用 TrainingArguments 類訓練屬性：

從變形金剛導入TrainingArguments

```
num_epochs = 3
batch_size = 24
logging_steps = len(panx_de_encoded['train']) // batch_size model_name =
f'{xlmr_model_name}-finetuned-panx-de' training_args = TrainingArguments(
    output_dir=model_name, log_level='error', num_train_epochs=num_epochs,
    per_device_train_batch_size=batch_size, per_device_eval_batch_size=batch_size,
    evaluation_strategy='epoch', save_steps=1e6, weight_decay=0.01, disable_tqdm=False,
    logging_steps=logging_steps, push_to_hub=True)
```

在這裡，我們在每個 epoch 結束時評估模型對驗證集的預測，調整權重衰減，並將 save_steps 設置為較大的數字以禁用檢查點，從而加快訓練速度。

這也是確保我們登錄到 Hugging Face Hub 的好方法（如果您在終端中工作，則可以改為執行命令`huggingface-cli login`）：

從`huggingface_hub`導入`notebook_login`

```
notebook_login()
```

我們還需要告訴Trainer如何在驗證集上計算指標，因此在這裡我們可以使用我們之前定義的`align_predictions()`函數以`seqeval`計算F1分數所需的格式提取預測和標籤：

從`seqeval.metrics`導入`f1_score`

```
def compute_metrics (eval_pred) :
    y_pred, y_true = align_predictions(eval_pred.predictions,
                                         eval_pred.label_ids)返回
    { f1 :f1_score(y_true, y_pred)}
```

最後一步是定義一個數據整理器，這樣我們就可以將每個輸入序列填充到批次中的最大序列長度。Transformers 為令牌分類提供了👉個專用的數據整理器，它將與輸入一起填充標籤：

從`變壓器`導入`DataCollatorForTokenClassification`

```
data_collator = DataCollatorForTokenClassification(xlmr_tokenizer)
```

填充標籤是必要的，因為與文本分類任務不同，標籤也是序列。這裡的一個重要細節是標籤序列用值 -100 填充，正如我們所見，PyTorch 損失函數忽略了該值。

我們將在本章中訓練多個模型，因此我們將通過創建`model_init()`方法來避免為每個Trainer初始化一個新模型。此方法加載未經訓練的模型並在`train()`調用開始時調用：

```
def model_init():返回
    (XLMRobertaForTokenClassification
     .from_pretrained(xlmr_model_name, config=xlmr_config).to(device))
```

我們現在可以將所有這些信息與編碼數據集一起傳遞給培訓師：

從`變壓器`導入`培訓師`

```
培訓師=培訓師 (model_init=model_init, args=training_args,
                  data_collator=data_collator, compute_metrics=compute_metrics,
                  train_dataset=panx_de_encoded[“訓練”], eval_dataset=panx_de_encoded[“驗證”],
                  tokenizer=xlmr_tokenizer)
```

然後按如下方式運行訓練循環並將最終模型推送到 Hub：

```
trainer.train() trainer.push_to_hub(commit_message= 訓練完成 ! )
```

Epoch	training Loss 訓練損失	F1	
1	0.2652	0.160244	0.822974
2	0.1314	0.137195	0.852747
3	0.0806	0.138774	0.864591

這些 F1 分數對於 NER 模型來說相當不錯。為了確認我們的模型是否按預期工作，讓我們在簡單示例的德語翻譯上對其進行測試：

```
text_de = Jeff Dean ist ein Informatiker bei Google in Kalifornien tag_text(text_de, tags,
trainer.model, xlmr_tokenizer)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	
Tokens <s>	傑德安				ist	ein	...	bei	Google	in	Kaliforni	en		</s>	
標籤	歐	B-PER	I-PER	I-PER	O		歐	...哦		B-組織	O	B-LOC		I-LOC	O

有用！但是我們永遠不應該基於單個示例對性能過於自信。相反，我們應該對模型的錯誤進行適當和徹底的調查。在下一節中，我們將探討如何為 NER 任務執行此操作。

錯誤分析

在我們深入研究 XLM-R 的多語言方面之前，讓我們花點時間調查一下我們模型的錯誤。正如我們在第 2 章中看到的，對模型進行徹底的錯誤分析是訓練和調試 Transformer（以及一般的機器學習模型）時最重要的方面之一。有幾種故障模式，看起來模型表現良好，但實際上它有一些嚴重的缺陷。訓練可能失敗的例子包括：

- 我們可能會不小心掩蓋了太多的標記，也掩蓋了我們的一些標籤以得到一個真正有前途的損失下降。
- `compute_metrics()`函數可能存在錯誤，會高估真實值表現。
- 我們可能將 NER 中的零類或 O 實體作為普通類包括在內，這將嚴重扭曲準確性和 F1 分數，因為它是多數類，而且幅度很大。

當模型的性能比預期差很多時，查看錯誤可以產生有用的見解並揭示僅通過查看代碼很難發現的錯誤。

即使模型表現良好並且代碼中沒有錯誤，錯誤分析仍然是了解模型優缺點的有用工具。這些都是

在生產中部署模型時，我們始終需要牢記的方面環境。

對於我們的分析，我們將再次使用我們可以使用的最強大的工具之一，即查看損失最高的驗證示例。我們可以重用我們在第 2 章中構建的用於分析序列分類模型的大部分函數，但我們現在將計算樣本序列中每個標記的損失。

讓我們定義一個可以應用於驗證集的方法：

從`torch.nn.functional`導入`cross_entropy`

```
def forward_pass_with_label (批處理) :
    # 將列表字典轉換為適合數據整理器的字典列表features = [dict(zip(batch, t)) for t in
    zip(*batch.values())]
    #填充輸入和標籤並將所有張量放在設備上 .to(device) with torch.no_grad():
    # 通過模型傳遞數據output = trainer.model(input_ids, attention_mask) #
    logit.size: [batch_size, sequence_length, classes]

    # 預測類軸上logit值最大的類predicted_label = torch.argmax(output.logits,
    axis=-1).cpu().numpy()
    # 在使用 view loss = cross_entropy(output.logits.view(-1, 7), labels.view(-1), reduction= none )
    展平批量維度後計算每個標記的損失

    # 展開批維度並轉換為 numpy 數組loss = loss.view(len(input_ids),
    -1).cpu().numpy()

    返回{ “損失” :損失， “預測標籤” :predicted_label}
```

我們現在可以使用`map()`將此函數應用於整個驗證集，並將所有數據加載到`DataFrame`中以供進一步分析：

```
valid_set = paxn_de_encoded[ “驗證” ] valid_set =
valid_set.map(forward_pass_with_label, batched=True, batch_size=32) df = valid_set.to_pandas()
```

標記和標籤仍然使用它們的 ID 進行編碼，因此讓我們將標記和標籤映射回字符串，以便更容易讀取結果。對於標籤為 -100 的填充標記，我們分配了一個特殊標籤IGN，以便稍後過濾它們。我們還通過將`loss`和`predicted_label`字段截斷為輸入的長度來去除所有填充：

```
index2tag[-100] = IGN
df[ input_tokens ] = df[ input_ids ].apply(
    lambda x: xlmr_tokenizer.convert_ids_to_tokens(x))
df[ predicted_label ] = df[ predicted_label ].apply(lambda x:
    [index2tag[i] for i in x])
```

```

df[“標籤”] = df[“標籤”].apply(
    lambda x: [index2tag[i] for i in x]) df[“loss”] =
df.apply(lambda x: x[“loss”][:len(x[“input_ids”])],
        axis=1) df[“預測標籤”] = df.apply(
    lambda x: x[“predicted_label”][:len(x[“input_ids”])], axis=1) df.head(1)

```

	attention_mask input_ids	標籤	損失	預測標籤	input_tokens
0	[1, 1, 1, 1, 1, 1] [0, 10699, 11, 15, 16104, 1388, 2]	[IGN, B 組織 \GN\ 我 組織 \我\組織 , I-ORG, IGN]	[0.0, 0.014679872, 0.0, 0.009469474, 0.010393422, 0.01293836, 0.0]	[我\組織 \B\組織 \我\組織 , 我\組織 \我\組織 \我\組織 \我 組織]	Ham, a, (, Unternehmen,, </s>]

每列包含每個樣本的標記、標籤、預測標籤等的列表。讓我們通過解壓縮這些列表來單獨查看令牌。pandas.Series.explode()函數允許我們通過為原始行列表中的每個元素創建一行來在一行中完成此操作。由於一行中的所有列表都具有相同的長度，我們可以對所有列並行執行此操作。我們還刪除了我們命名為IGN的填充令牌，因為它們的損失無論如何都是零。最後，我們將仍然是numpy.Array對象的損失轉換為標準浮點數：

```

df_tokens = df.apply(pd.Series.explode) df_tokens =
df_tokens.query(“labels != IGN”) df_tokens[“loss”] =
df_tokens[“loss”].astype(float).round(2) 標籤df_tokens.head(7)

```

	attention_mask input_ids	labels	loss	predicted_label	input_tokens	
-	10699	B-ORG	0.01	B-組織		火腿
-	15	I-ORG	0.01	I-組織		
-	16104	I-ORG	0.01	I-組織		(Unternehmen) WE
-	1388	I-ORG	0.01	I-組織		Luz
-	56530	歐	0.00	歐		
-	83982	B-ORG	0.34	B-組織		
-	10	I-ORG	0.45	I-組織		一個

有了這種形狀的數據，我們現在可以按輸入標記對其進行分組，並將每個標記的損失與計數、平均值和總和相加。最後，我們根據損失總和對聚合數據進行排序，看看哪些代幣在驗證集中累積的損失最多：

```

(
    df_tokens.groupby(“input_tokens”)
    [[“loss”]].agg([“count”, “mean”,  

        “sum”]).droplevel(level=0, axis=1) #去掉多級列

```

```
.sort_values(by= sum ,
ascending=False).reset_index().round(2).head(10)
```

```
.T  
)
```

	0	1	2	3	4	5	6	7	8	9
input_tokens der 在 von /					和 ()		一個	
數數	6066	1388	989	808	163	1171	246	246		2898 125
意思是	0.03	0.1	0.14	0.14	0.64	0.08	0.3	0.29	0.02	0.44
和	200.71	138.05	137.33	114.92	104.28	99.15	74.49	72.35	59.31	54.48

我們可以在這個列表中觀察到幾種模式：

- 空白標記的總損失最高，這並不奇怪，因為它也是列表中最常見的標記。但是，它的平均損失遠低於列表中的其他代幣。這意味著該模型不會費力對其進行分類。
- 像“in”、“von”、“der”和“und”這樣的詞出現得比較頻繁。它們經常與命名實體一起出現，有時是它們的一部分，這解釋了為什麼模型可能會混淆它們。
- 單詞開頭的括號、斜杠和大寫字母較少見，但有相對較高的平均損失。我們將進一步調查他們。

我們還可以對標籤 ID 進行分組並查看每個類別的損失：

```
(  
    df_tokens.groupby( labels )  
    [[ loss ]].agg([ count , mean ,  
        sum ]).droplevel(level=0,  
        axis=1).sort_values(by= mean ,  
        ascending=False).reset_index().round(2)  
  
.T  
)
```

	0	1	2	3	4	5	6
標籤 B-ORG I-LOC			組織機構	集團	B-PER I-PER O		
計數2683	1462	3820	3172	2893	4139	43648	
均值0.66	0.64	0.48	0.35	0.26	0.18	0.03	
總和1769.47	930.94	1850.39	1111.03	760.56	750.91	1354.46	

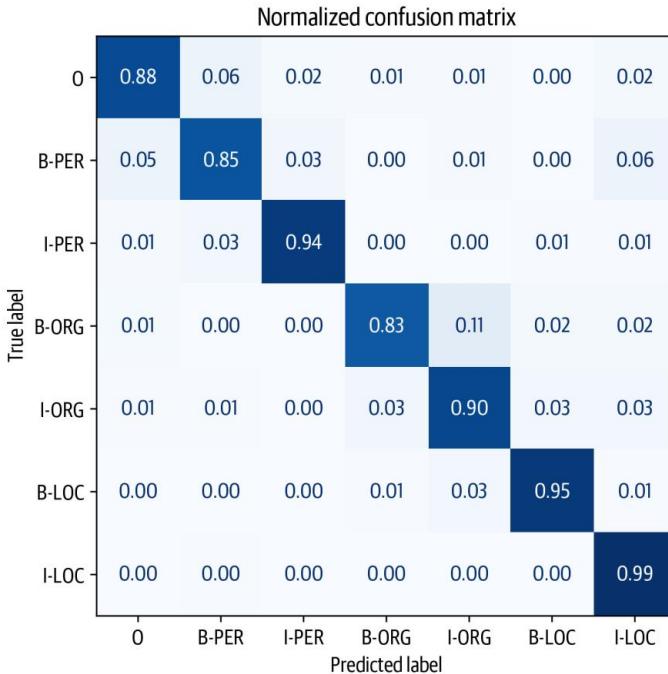
我們看到B-ORG的平均損失最高，這意味著確定組織的開始對我們的模型提出了挑戰。

我們可以通過繪製令牌分類的混淆矩陣來進一步分解它，我們可以看到組織的開頭經常與後續的I-ORG令牌混淆：

從sklearn.metrics導入ConfusionMatrixDisplay, confusion_matrix

```
def plot_confusion_matrix (y_preds , y_true ,標籤) :
    cm = confusion_matrix(y_true,y_preds, normalize= true )圖, ax =
    plt.subplots(figsize=(6, 6)) disp = ConfusionMatrixDisplay(confusion_matrix=cm,
    display_labels=labels) disp.plot(cmap= Blues , values_format= .2f ,ax=ax, colorbar=False)
    plt.title( 歸一化混淆矩陣 ) plt.show()
```

```
plot_confusion_matrix (df_tokens [ “標籤” ] , df_tokens [ “predicted_label” ] ,
標籤.名稱)
```



從圖中可以看出，我們的模型最容易混淆B-ORG和I-ORG實體。否則，它非常擅長對剩餘實體進行分類，混淆矩陣的近對角線性質清楚地表明了這一點。

現在我們已經檢查了令牌級別的錯誤，讓我們繼續看看具有高損失的序列。對於此計算，我們將重新訪問我們的“未分解”數據框並通過對每個令牌的損失求和來計算總損失。為此，讓我們首先編寫一個函數來幫助我們顯示帶有標籤和損失的標記序列：

```
def get_samples(df):
    for _, row in df.iterrows():
        preds, labels = [], []
        for i in range(len(row['attention_mask'])):
            if row['attention_mask'][i] != 0:
                labels.append(row['labels'][i])
                preds.append(row['predicted_label'][i])
                tokens.append(row['input_tokens'][i])
                losses.append(float(row['loss'][i]))
    df_tmp = pd.DataFrame({'tokens': tokens, 'labels': labels,
                           'preds': preds, 'tokens': tokens, 'losses': losses})
    df['total_loss'] = df['loss'].sum()
    df.sort_values(by='total_loss', ascending=False).head(3)
    return df_tmp
```

對於get_samples(df_tmp)中的樣本顯示（樣本）

	0	1	2	3	4	...	13	14	15	16	17
令牌	代	幣	.	巨	力		n	後	花	園	</S>
標籤	B-ORG	IGN	IGN	I-ORG	I-ORG	...		IGN	IGN	I-ORG	點火
預測值		歐	歐	歐	歐	...	I-ORG	I-ORG	I-ORG	I-ORG	O
損失	7.89	0.00	0.00	6.88	8.05		0.00	0.00		0.01	0.00

	0	1	2	3	4	...	13	14	15	16	17	18
代幣	代	幣	▼	K		...	k				阿拉	</S>
標籤	OOO			點亮	O	...		IGN	I-LOC	I-LOC	點火	點火
預測對象			B-ORG	OO	...	OO				歐	歐	歐
損失	0.00	0.00	3.59			0.00	0.00	...	0.00	7.86	7.78	0.00

	0	1	2	3	4	...	10	11	12	13	14	
代幣聯合國	多維積分	...中	非	共和國	</S>							
標籤	B-PER	I-PER	I-PER	IGN		I-PER	...	我-PER	我-PER	I-PER	I-PER	IGN
預測	B-ORG	組織機構	組織機構	組織機構		組織機構	組織機構	組織機構	組織機構	.
損失	6.46	5.59	5.51	0.00		5.11	...	4.77	5.32	5.10	4.87	0.00

很明顯，這些樣本的標籤有問題；例如，聯合國和中非共和國都被標記為一個人！同時，“8. Juli”在第一個例子中被標記為一個組織。事實證明，PAN-X 數據集的註釋是通過自動化過程生成的。

此類註釋通常被稱為“銀標準”（與人工生成註釋的“黃金標準”相反），並且在某些情況下自動方法無法生成合理的標籤也就不足為奇了。事實上，這種故障模式並不是自動方法所獨有的；即使人類仔細地註釋數據，當註釋者註意力不集中或者他們只是誤解了句子時，也可能會出現錯誤。

我們之前註意到的另一件事是括號和斜杠的損失相對較高。讓我們看幾個帶有左括號的序列示例：

```
df_tmp = df.loc[df['input_tokens'].apply(lambda x: u'\u2581' in x)].head(2)用於get_samples(df_tmp)
中的樣本：顯示（樣本）
```

	0	2倍	3倍	4倍	5倍
代幣火腿		(Unternehmen)			</s>
標籤B-ORG IGN		我-組織我-組織		I-ORG 點火	
preds B-ORG I-ORG I-ORG I-ORG				我-組織我-組織	
損失0.01	0.00	0.01	0.01	0.01	0.00

	0	2倍	3倍	4倍	5倍	6倍	7
代幣Kesk kül		A	(瑪娜)		</s>
標籤B-LOC IGN		IGN	I-LOC	I-LOC 點火		I-LOC 點火	
預測B-LOC I-LOC I-LOC I-LOC I-LOC				我-LOC	I-LOC	I-LOC	
損失0.02	0.00	0.00	0.01	0.01	0.00	0.01	0.00

通常我們不會將括號及其內容作為命名實體的一部分，但這似乎是自動提取註釋文檔的方式。在其他示例中，括號中包含地理說明。

雖然這確實也是一個位置，但我們可能希望將它與註釋中的原始位置斷開連接。該數據集由不同語言的維基百科文章組成，文章標題通常在括號中包含某種解釋。

例如，在第一個示例中，括號中的文字表示 Hama 是一家 “Unternehmen”，即英語中的公司。這些是我們推出模型時需要知道的重要細節，因為它們可能會對模型所屬的整個管道的下游性能產生影響。

通過相對簡單的分析，我們發現了模型和數據集中的一些弱點。在實際用例中，我們將迭代此步驟，清理

數據集，重新訓練模型，並分析新的錯誤，直到我們對性能感到滿意為止。

在這裡，我們分析了單一語言的錯誤，但我們也對跨語言的性能感興趣。在下一節中，我們將進行一些實驗，看看 XLM-R 中的跨語言遷移效果如何。

跨語言遷移

現在我們已經在德語上對 XLM-R 進行了微調，我們可以評估它通過 Trainer 的predict()方法遷移到其他語言的能力。由於我們計劃評估多種語言，因此讓我們創建一個簡單的函數來為我們執行此操作：

```
def get_f1_score(trainer, dataset):返回
    trainer.predict(dataset).metrics[ test_f1 ]
```

我們可以使用此函數來檢查測試集的性能並在字典中跟蹤我們的分數：

```
f1_scores = defaultdict(dict)
f1_scores[ de ][ de ] = get_f1_score(trainer, pnx_de_encoded[ test ]) print(f [de] 模
型在 [de] 數據集上的 F1 分數 :{f1_scores[ de ][ de ]:.3f} )
[de] 數據集上 [de] 模型的 F1 分數 :0.868
```

這些對於 NER 任務來說是非常好的結果。我們的指標在 85% 左右，我們可以看到該模型似乎在ORG實體上表現最差，可能是因為它們在訓練數據中最不常見，而且許多組織名稱在 XLM-R 中很少見詞彙。其他語言怎麼樣？為了熱身，讓我們看看我們的模型如何針對法語的德國票價進行微調：

```
text_fr = Jeff Dean est informaticien chez Google en Californie tag_text(text_fr, tags,
trainer.model, xlmr_tokenizer)
```

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
Tokens	< s >	傑德安			est	informatic	ien	chez	Google	en	Cali	for		
標籤	歌	B-PER	每	每	歌	歌	哦		B-ORG O		B-LOC		本地代碼	本地代碼

不錯！儘管兩種語言的名稱和組織相同，但該模型確實成功地正確標記了“Kalifornien”的法語翻譯。接下來，讓我們通過編寫一個對數據集進行編碼並生成分類報告的簡單函數來量化我們的德語模型在整個法語測試集上的表現：

```
def evaluate_lang_performance(lang, trainer):
    pnx_ds = encode_pnx_dataset(pnx_ch[lang]) 返回
    get_f1_score(trainer, pnx_ds[ test ])
```

```
f1_scores[ de ][ fr ]=evaluate_lang_performance( fr ,trainer) print(f [fr] 數據
集上 [de] 模型的 F1 分數 :{f1_scores[ de ][ fr ]:.3f} )
[fr] 數據集上 [de] 模型的 F1 分數 :0.714
```

儘管我們看到微平均指標下降了大約 15 個點，但請記住，我們的模型還沒有看到一個帶標籤的法語示例！一般來說，性能下降的大小與語言之間的“距離”有關。

儘管德語和法語被歸為印歐語系，但從技術上講，它們屬於不同的語系：分別是日耳曼語系和羅曼語系。

接下來，讓我們評估意大利語的表現。由於意大利語也是一種羅曼語，我們希望得到與法語相似的結果：

```
f1_scores[ de ][ it ]=evaluate_lang_performance( it ,trainer) print(f [de] 模
型在 [it] 數據集上的 F1 分數 :{f1_scores[ de ][ it ]:.3f} )
[de] 模型在 [it] 數據集上的 F1 分數 :0.692
```

事實上，我們的期望得到了F1分數的證實。最後，我們來看看日耳曼語系英語的表現：

```
f1_scores[ de ][ en ]=evaluate_lang_performance( en ,trainer) print(f [de]
模型在 [en] 數據集上的 F1 分數 :{f1_scores[ de ][ en ]:.3f} )
[de] 模型在 [en] 數據集上的 F1 分數 :0.589
```

令人驚訝的是，我們的模型在英語上表現最差，儘管我們可能直覺上認為德語與英語比法語更相似。在對德語進行微調並對法語和英語進行零樣本遷移後，接下來讓我們檢查一下何時直接對目標語言進行微調是有意義的。

零樣本傳輸何時有意義？

到目前為止，我們已經看到在德語語料庫上微調 XLM-R 產生了大約 85% 的F1分數，並且在沒有任何額外訓練的情況下，該模型能夠在我們語料庫中的其他語言上取得適度的性能。問題是，這些結果有多好？它們與在單語語料庫上微調的 XLM-R 模型相比如何？

在本節中，我們將通過在規模不斷增加的訓練集上微調 XLM-R 來探討法語語料庫的這個問題。通過這種方式跟蹤性能，我們可以確定零樣本跨語言遷移在哪一點上更勝一籌，這在實踐中有助於指導是否收集更多標記數據的決策。

為簡單起見，我們將保留來自德國語料庫微調運行的相同超參數，除了我們將調整Training Arguments的logging_steps參數以考慮不斷變化的訓練集大小。我們可以將這一切包裝在一個簡單的函數中，該函數接受一個對應於

單語語料庫，通過num_samples對其進行下採樣，並對該樣本微調 XLM-R 以返回最佳時期的指標：

```
def train_on_subset(數據集, num_samples): train_ds =
    dataset[ train ].shuffle(seed=42).select(range(num_samples)) valid_ds = dataset[ validation ]
    test_ds = dataset[ test ] training_args.logging_steps = len(train_ds) // batch_size

    培訓師=培訓師 (model_init=model_init, args=training_args,
        data_collator=data_collator, compute_metrics=compute_metrics,
        train_dataset=train_ds, eval_dataset=valid_ds, tokenizer=xlmr_tokenizer) trainer.train()如果
    training_args.push_to_hub:

        trainer.push_to_hub(commit_message= 訓練完成 ! )

    f1_score = get_f1_score(trainer, test_ds)返回
    pd.DataFrame.from_dict(
        { "num_samples" : [len (train_ds) ], "f1_score" : [f1_score]})
```

就像我們對德語語料庫進行微調一樣，我們還需要將法語語料庫編碼為輸入 ID、注意掩碼和標籤 ID：

```
panx_fr_encoded = encode_pnx_dataset(pnx_ch[ fr ])
```

接下來讓我們通過在包含 250 個示例的小型訓練集上運行來檢查我們的函數是否有效：

```
training_args.push_to_hub =錯誤的metrics_df
= train_on_subset(panx_fr_encoded, 250) metrics_df
```

	num_samples	f1_score
0	250	0.137329

我們可以看到，僅用 250 個示例，對法語的微調就大大低於德語的零樣本遷移。現在讓我們將訓練集大小增加到 500、1,000、2,000 和 4,000 個示例，以了解性能如何

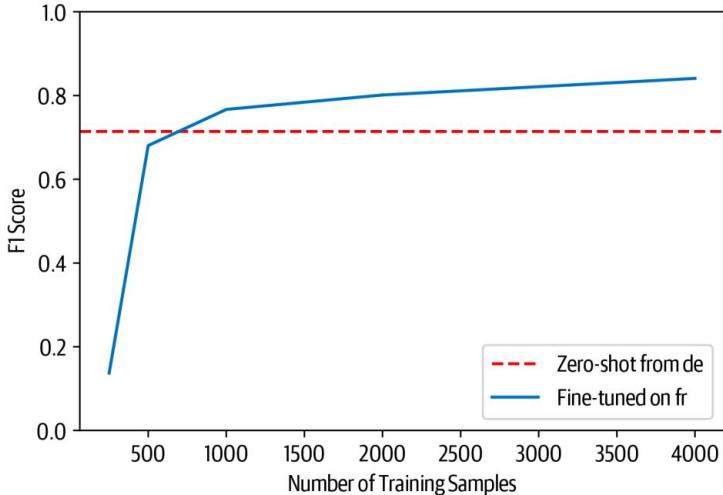
增加：

```
對於[500, 1000, 2000, 4000]中的樣本數： metrics_df =
    metrics_df.append(
        train_on_subset(panx_fr_encoded, num_samples), ignore_index=True)
```

我們可以通過繪製測試集上的F1分數作為增加訓練集大小的函數來比較法語樣本的微調與德語的零樣本跨語言遷移相比如何：

```
圖， ax = plt.subplots()
ax.axhline(f1_scores[ de ][ fr ], ls= -- , color= r )
metrics_df.set_index( num_samples ).plot(ax=斧頭)
```

```
plt.legend([ Zero-shot from de , Fine-tuned on fr ], loc= lower right ) plt.ylim((0, 1))
plt.xlabel( 訓練樣本數 ) plt.ylabel( F1分數 ) plt.show()
```



從圖中我們可以看出，在大約 750 個訓練示例之前，零樣本遷移仍然具有競爭力，之後對法語的微調達到了與對德語進行微調時獲得的性能水平相似的水平。儘管如此，這個結果是不容小覷的！根據我們的經驗，讓領域專家標記甚至數百個文檔可能成本很高，尤其是對於 NER，標記過程是細粒度且耗時的。

我們可以嘗試使用最後一種技術來評估多語言學習：同時對多種語言進行微調！讓我們看看如何做到這一點。

同時對多種語言進行微調到目前為止，我們已經看

到從德語到法語或意大利語的零樣本跨語言遷移會導致性能下降約 15 個百分點。緩解這種情況的一種方法是同時微調多種語言。為了看看我們可以獲得什麼類型的收益，讓我們首先使用Datasets 中的concatenate_datasets()函數將德語和法語語料庫連接在一起：



從數據集導入concatenate_datasets

```
def concatenate_splits(corpora):
    multi_corpus = DatasetDict()用於在語
    料庫 [0].keys()中拆分： multi_corpus[split]
    = concatenate_datasets(
```

```
[corpus[split] for corpus in corpora]).shuffle(seed=42)
返回multi_corpus

panx_de_fr_encoded = concatenate_splits([panx_de_encoded, pанx_fr_encoded])
```

對於訓練，我們將再次使用前面部分中的相同超參數，因此我們可以簡單地更新訓練器中的日誌記錄步驟、模型和數據集：

```
training_args.logging_steps = len(panx_de_fr_encoded[ "train" ]) // batch_size
training_args.push_to_hub = True
training_args.output_dir = "xlm-roberta-base-finetuned-panx-de-fr"
```

```
培訓師=培訓師 (model_init=model_init, args=training_args,
data_collator=data_collator, compute_metrics=compute_metrics,
tokenizer=xlmr_tokenizer, train_dataset=panx_de_fr_encoded[ "訓練" ],
eval_dataset=panx_de_fr_encoded[ "驗證" ])

trainer.train()
trainer.push_to_hub(commit_message= "訓練完成！")
```

讓我們看看模型在每種語言的測試集上的表現如何：

```
for lang in langs: f1 =
    evaluate_lang_performance(lang, trainer) print(f F1-score of
[de-fr] model on [{lang}] dataset: {f1:.3f} )
```

```
[de-fr] 模型在 [de] 數據集上的 F1 分數 :0.866 [de-fr] 模型在 [fr] 數據集
上的 F1 分數 :0.868 [de-fr] 模型在 [it] 數據集上的 F1 分數 : [de-fr] 模
型在 [en] 數據集上的 0.815 F1 分數 :0.677
```

它在 French split 上的表現比以前好得多，與德國測試集上的表現相當。有趣的是，它在意大利語和英語拆分上的表現也提高了大約 10 個百分點！因此，即使添加另一種語言的訓練數據也能提高模型在未見過的語言上的性能。

讓我們通過比較每種語言的微調性能與所有語料庫的多語言學習的性能來完善我們的分析。由於我們已經對德語語料庫進行了微調，我們可以使用train_on_subset()函數對其餘語言進行微調，其中num_samples等於訓練集中的示例數：

```
語料庫= [panx_de_encoded]

# 從迭代中排除德語for lang in langs[1:]:
training_args.output_dir=f "xlm-roberta-
base-finetuned-panx-{lang}"
# 微調單語語料庫ds_encoded =
encode_pанx_dataset(panx_ch[{lang}]) metrics =
train_on_subset(ds_encoded, ds_encoded[ "train" ].num_rows)
# 在普通字典中收集 F1 分數f1_scores[{lang}][{lang}]
= metrics[ "f1_score" ][0]
```

```
# 將單語語料庫添加到語料庫列表中以連接語料庫 .append(ds_encoded)
```

現在我們已經對每種語言的語料庫進行了微調，下一步是將所有拆分連接在一起以創建所有四種語言的多語言語料庫。與之前的德語和法語分析一樣，我們可以使用concatenate_splits()函數為我們在上一步生成的語料庫列表上執行此步驟：

```
corpora_encoded = concatenate_splits(語料庫)
```

現在我們有了多語言語料庫，我們用訓練器運行熟悉的步驟：

```
training_args.logging_steps = len(corpora_encoded[“train”]) // batch_size training_args.output_dir  
= “xlm-roberta-base-finetuned-panx-all”
```

```
培訓師=培訓師(model_init=model_init, args=training_args,  
data_collator=data_collator, compute_metrics=compute_metrics,  
tokenizer=xlmr_tokenizer, train_dataset=corpora_encoded[“訓練”],  
eval_dataset=corpora_encoded[“驗證”])
```

```
trainer.train()  
trainer.push_to_hub(commit_message=“訓練完成！”)
```

最後一步是生成訓練器對每種語言的測試集的預測。這將使我們深入了解多語言學習的實際效果。

我們將在我們的f1_scores字典中收集F1分數，然後創建一個DataFrame來總結我們多語言實驗的主要結果：

```
對於idx, lang in enumerate(langs) :  
    f1_scores[“all”][lang] = get_f1_score(trainer, 語料庫[idx][“test”])  
  
scores_data = {“de” : f1_scores[“de”],  
               “each” : {lang: f1_scores[lang][lang] for lang in langs}, “all” :  
               f1_scores[“all”]} f1_scores_df = pd.DataFrame(scores_data).T.round(4)  
f1_scores_df.rename_axis(index=“Fine-tune on”, columns=“Evaluated on”, inplace=True)
```

f1_scores_df

評價於德		fr	它	恩
微調開啟				
德	0.8677	0.7141	0.6923	0.5890
每個	0.8677	0.8505	0.8192	0.7068
全部	0.8682	0.8647	0.8575	0.7870

從這些結果中，我們可以得出一些一般性結論：

- 多語言學習可以顯著提高性能，尤其是當用於跨語言遷移的低資源語言屬於相似語言時

家庭。在我們的實驗中，我們可以看到德語、法語和意大利語在所有類別中都取得了相似的表現，這表明這些語言彼此之間的相似性高於英語。

- 作為一般策略，最好將注意力集中在語系內的跨語言遷移上，尤其是在處理不同文字（如日語）時。

與模型小部件交互

在本章中，我們將許多經過微調的模型推送到 Hub。雖然我們可以使用 pipeline() 函數在我們的本地機器上與它們交互，但 Hub 提供了非常適合這種工作流程的小部件。圖 4-5 中顯示了我們的 transformersbook/xlm-roberta-base-finetuned-pax-all 檢查點的示例。如您所見，它在識別德語文本的所有實體方面做得很好。

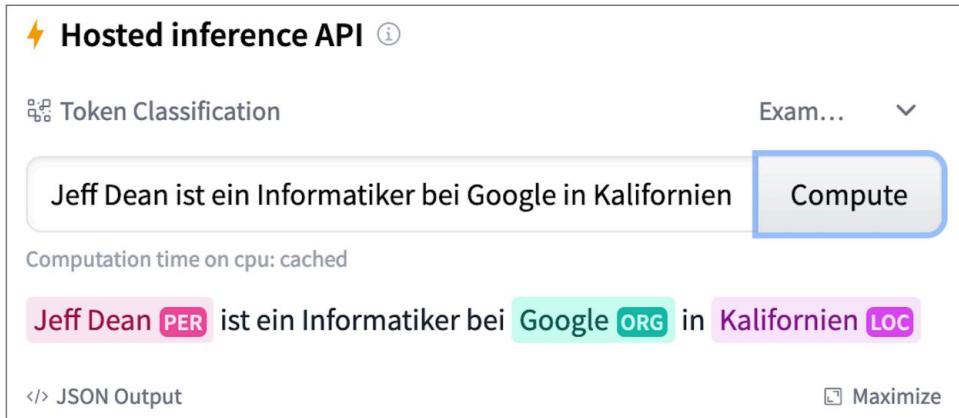


圖 4-5。Hugging Face Hub 上的小部件示例

結論

在本章中，我們了解瞭如何使用在 100 種語言上預訓練的單個轉換器 XLM-R 在多語言語料庫上處理 NLP 任務。雖然我們能夠證明，當只有少量標記示例可用於微調時，從德語到法語的跨語言遷移具有競爭力，但如果目標語言與目標語言明顯不同，通常不會出現這種良好的性能。其中一個基礎模型經過微調，或者不是預訓練期間使用的 100 種語言之一。像 MAD-X 這樣的最新提案正是為這些設計的。

低資源場景，並且由於 MAD-X 構建在 Transformers 之上，您可以輕鬆調整本章中的代碼以使用它¹⁶。

到目前為止，我們已經研究了兩個任務：序列分類和標記分類。

這些都屬於自然語言理解領域，其中文本被合成為預測。在下一章中，我們將首先了解文本生成，其中模型的輸入和輸出都是文本。

¹⁶ J. Pfeiffer 等人，“MAD-X：基於適配器的多任務跨語言傳輸框架”，(2020)。