

第九章

處理很少甚至沒有標籤

每個數據科學家心中都有一個根深蒂固的問題，通常是他們在開始新項目時首先要問的問題：是否有標記數據？

通常情況下，答案是“否”或“有一點”，隨後客戶期望您的團隊的精美機器學習模型仍應表現良好。由於在非常小的數據集上訓練模型通常不會產生好的結果，因此一個明顯的解決方案是註釋更多數據。然而，這需要時間並且可能非常昂貴，特別是如果每個註釋都需要領域專業知識來驗證。

幸運的是，有幾種方法非常適合處理很少甚至沒有標籤的情況！您可能已經熟悉其中的一些，例如零樣本學習或少樣本學習，正如 GPT-3 僅用幾十個示例即可執行各種任務的令人印象深刻的能力所證明的那樣。

一般來說，性能最佳的方法取決於任務、可用數據量以及標記數據的比例。**圖 9-1** 中顯示的決策樹可以幫助指導我們完成選擇最合適方法的過程。

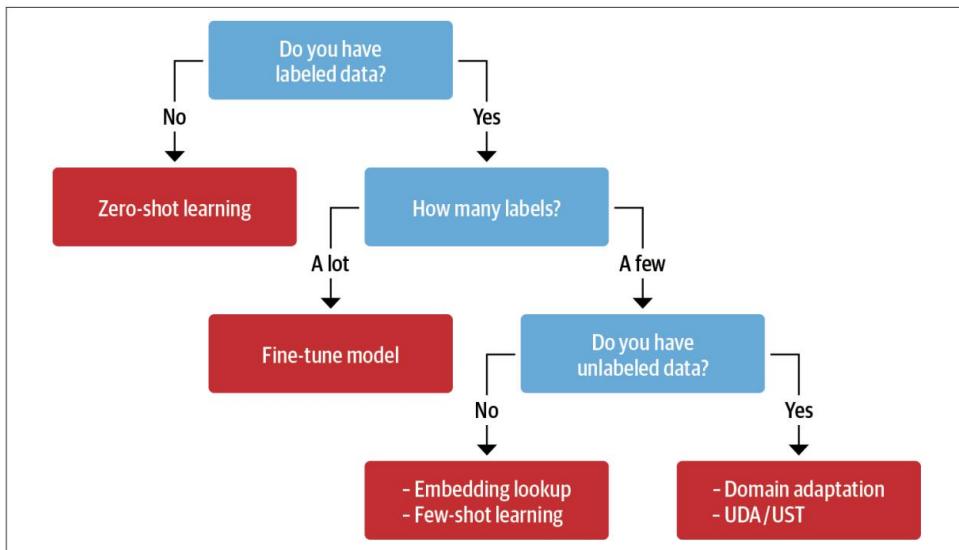


圖 9-1 在沒有大量標記數據的情況下可用於提高模型性能的幾種技術

讓我們逐步瀏覽這個決策樹：

1. 你有標註數據嗎？

即使是少數標記樣本也會對哪種方法最有效產生影響。如果您根本沒有標記數據，則可以從零樣本學習方法開始，該方法通常會設置一個強大的基線。

2. 有多少個標籤？

如果標記數據可用，決定因素是多少。如果您有大量可用的訓練數據，您可以使用第 2 章中討論的標準微調方法。

3. 你有未標記的數據嗎？

如果您只有少數標記樣本，那麼如果您可以訪問大量未標記數據，那麼它會大有幫助。如果你可以訪問未標記的數據，你可以在訓練分類器之前使用它來微調域上的語言模型，或者你可以使用更複雜的方法，例如無監督數據增強（UDA）或不確定性感知自training（UST）。¹如果您沒有任何未標記的數據可用，則您沒有註釋更多的選項。

¹ Q. Xie 等人，“用於一致性訓練的無監督數據增強”，(2019); S. Mukherjee 和 AH Awadallah，“小樣文本分類的不確定性感知自我訓練”，(2020)。

數據。在這種情況下，您可以使用少樣本學習或使用來自預訓練語言模型的嵌入來執行最近鄰搜索的查找。

在本章中，我們將通過解決許多使用問題跟蹤器（如Jira）的支持團隊面臨的一個常見問題來完成這個決策樹或GitHub幫助他們的用戶：根據問題的描述用元數據標記問題。這些標籤可能定義問題類型、導致問題的產品或負責處理所報告問題的團隊。自動化此過程會對生產力產生重大影響，並使支持團隊能夠專注於幫助他們的用戶。

作為運行示例，我們將使用與流行的開源項目相關的 GitHub 問題：變形金剛！現在讓我們看看這些問題中包含哪些信息，如何構建任務以及如何獲取數據。



本章介紹的方法適用於文本分類，但其他技術（如數據擴充）對於處理更複雜的任務（如命名實體識別、問答或摘要）可能是必需的。

構建 GitHub 問題標記器

如果您導航到“問題”選項卡在 Transformers 存儲庫中，您會發現如圖 9-2 所示的問題，其中包含標題、描述和一組表徵問題的標記或標籤。這提出了一種構建監督學習任務的自然方法：給定問題的標題和描述，預測一個或多個標籤。由於可以為每個問題分配可變數量的標籤，這意味著我們正在處理多標籤文本分類問題。這通常比我們在第 2 章中遇到的多類問題更具挑戰性，在第 2 章中，每條推文僅分配給一種情緒。

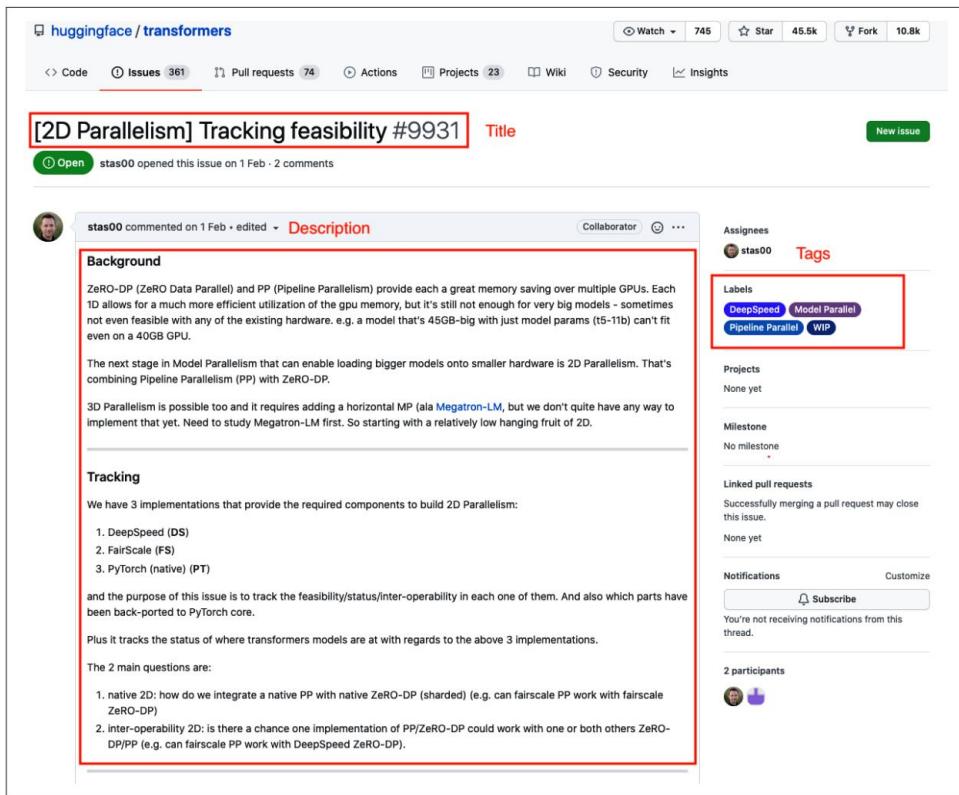


圖 9-2。Transformers 存儲庫中的一個典型 GitHub 問題

現在我們已經看到了 GitHub 問題的樣子，讓我們看看如何下載它們來創建我們的數據集。

獲取數據要獲取存儲

庫的所有問題，我們將使用 GitHub REST API 投票的問題端點。這個端點返回一個 JSON 對象列表，每個對像都包含大量關於手頭問題的字段，包括它的狀態（打開或關閉）、打開問題的人，以及我們在中看到的標題、正文和標籤圖 9-2。

由於獲取所有問題需要一段時間，因此我們在本書的 GitHub 存儲庫中包含了一個 `github-issues transformers.jsonl` 文件，以及您可以用來自下它們的 `fetch_issues()` 函數。



GitHub REST API 將拉取請求視為問題，因此我們的數據集包含兩者的混合。為簡單起見，我們將為這兩種類型的問題開發分類器，儘管在實踐中您可能會考慮構建兩個單獨的分類器以對模型的性能進行更細粒度的控制。

現在我們知道如何獲取數據，讓我們來看看如何清理數據。

準備數據

下載完所有問題後，我們可以使用 Pandas 加載它們：

將熊貓導入為pd

```
dataset_url = https://git.io/nlp-with-transformers df_issues =
pd.read_json(dataset_url, lines=True) print(f DataFrame 形狀：
{df_issues.shape} )
```

數據框形狀 :(9930, 26)

我們的數據集中有將近 10,000 個問題，通過查看一行我們可以看到從 GitHub API 檢索的信息包含許多字段，例如

URL、ID、日期、用戶、標題、正文和標籤：

```
cols=[ url , id , title , user , labels , state , created_at , body ] df_issues.loc[2,
cols].to_frame()
```

	網址
網址	https://api.github.com/repos/huggingface/trans...
ID	849529761
標題	[DeepSpeed] ZeRO 第 3 階段集成 : 獲得...
用戶	{ 登錄 : stas00 , id :10676103 , node_id : ... }
標籤	[{ id :2659267025 , node_id : MDU6TGFiZWwyNj...]
狀態	打開
created_at	2021-04-02 23:40:42 **[這
身體	個還沒活，準備重新...

labels列是我們感興趣的東西，每一行都包含一個 JSON 對象列表，其中包含關於每個標籤的元數據：

```
[  
 {  
   id :2659267025,  
   node_id : MDU6TGFiZWwyNjU5MjY3MDI1 ,  
   url : https://api.github.com/repos/huggingface... ,  
   name : DeepSpeed , color : 4D34F7 ,
```

```

    "默認" :假, "描
述" :""
}
]

```

出於我們的目的，我們只對每個標籤對象的名稱字段感興趣，所以讓我們只用標籤名稱覆蓋標籤列：

```

df_issues[ 標籤 ]=(df_issues[ 標籤 ]
                    .apply(lambda x:[meta[ name ] for meta in x]))
df_issues[[ 標籤 ]].head()

```

	標籤
0	
1	
2	[深遠]
3	
4	

現在標籤列中的每一行都是 GitHub 標籤的列表，因此我們可以計算每一行的長度以找到每個問題的標籤數量：

```
df_issues[ "標籤 " ].apply(lambda x:len(x)).value_counts().to_frame().T
```

	0	=	*	*	45		
標籤	6440	3057	305	100	25	3	

這表明大多數問題都有零個或一個標籤，只有一個以上的問題要少得多。接下來讓我們看一下數據集中出現頻率最高的前 10 個標籤。

在 Pandas 中，我們可以通過“分解”標籤列來做到這一點，使列表中的每個標籤成為一行，然後簡單地計算每個標籤的出現次數：

```

df_counts=df_issues[ labels ].explode().value_counts() print(f" 標籤數 :
{len(df_counts)} ")
# 顯示top-8標籤類別df_counts.to_frame().head(8).T

```

標籤數量 :65

	wontfix	模型卡	核：代幣化	新模型	核：造型	把招工廣告	先好問題	用法
標籤	2284	649	106	98	64	52	50	46

我們可以看到數據集中有 65 個唯一標籤，並且類別非常不平衡，wontfix 和 model card 是最常見的標籤。為了使分類問題更易於處理，我們將專注於為子集構建標記器

的標籤。例如，一些標籤，如Good First Issue或Help Wanted，可能很難從問題的描述中預測，而其他標籤，如模型卡，可以用一個簡單的規則進行分類，該規則檢測模型卡何時添加到擁抱臉集線器。

以下代碼針對我們將使用的標籤子集過濾數據集，同時對名稱進行標準化以使其更易於閱讀：

```
label_map = { "核心·標記化" : "標記化" ,
              "新模型" : "新模型" ,
              《核心·建模》 : 《模型訓練》 ,
              "用法" : "用法" ,
              核心·管道 : 管道 ,
              TensorFlow : tensorflow 或 tf ,
              PyTorch : 火炬 ,
              "例子" : "例子" ,
              "文檔" : "文檔" }

def filter_labels(x):返回
    [label_map[label] for label in x if label in label_map]

df_issues[ "標籤" ] = df_issues[ "標籤" ].apply(filter_labels) all_labels =
list(label_map.values())
```

現在讓我們看看新標籤的分佈：

```
df_counts = df_issues[ labels ].explode().value_counts()
df_counts.to_frame().T
```

	代幣化新模式		模型訓練	使用管道	tensorflow 或 tf		pytorch	文檔示例	
標籤106		98	64	46	42	41	37	28	24

在本章的後面，我們會發現將未標記的問題作為一個單獨的訓練集來處理是很有用的，所以讓我們創建一個新列來指示問題是否未標記：

```
df_issues[ split ]= 未標記 mask =
df_issues[ labels ].apply(lambda x: len(x))>0 df_issues.loc[mask,
split ]= labeled df_issues[ split ].value_counts().to_frame()
```

	分裂
未標記的9489	
標記的	441

現在讓我們看一個例子：

```
對於[ title , body , labels ]中的列：print(f {column}:
    {df_issues[column].iloc[26][:500]}\n )
```

title: 添加新的 CANINE 模型

身體： # ★ 新機型追加

型號說明

Google 最近提出了一個新的 **C** 字符 **A** 架構，具有 **N**o 標記化 ***I***n **N**eural **E**ncoders 架構
(CANINE)。不僅標題令人興奮：

流水線 NLP 系統在很大程度上已被端到端神經建模所取代，但幾乎所有常用模型仍然需要明確的標記化步驟。雖然最近基於數據派生的子詞詞典的標記化方法比手動 en 更不脆弱。

標籤 [新模型]

在這個例子中提出了一個新的模型架構，所以新的模型標籤是有意義的。我們還可以看到標題包含對我們的分類器有用的信息，所以讓我們將它與正文字段中的問題描述連接起來：

```
df_issues[ text ]=(df_issues .apply(lambda
    x:x[ title ]+ '\n\n' +x[ body ],axis=1))
```

在我們查看其餘數據之前，讓我們檢查數據中的任何重複項並使用 drop_duplicates ()方法刪除它們：

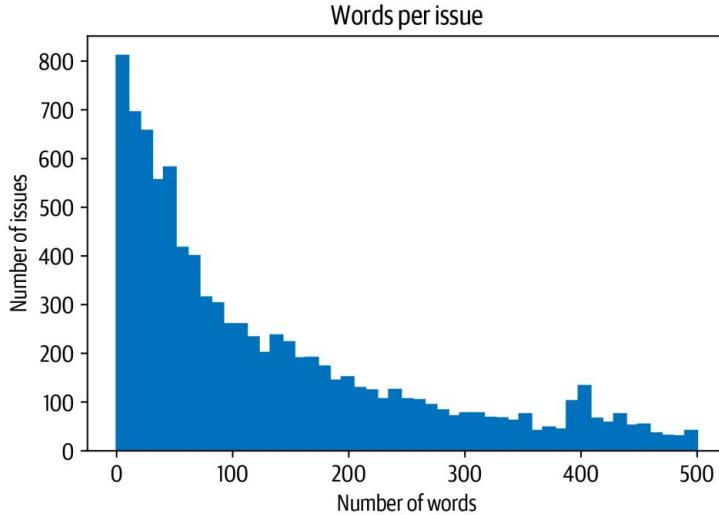
```
len_before = len(df_issues) df_issues
= df_issues.drop_duplicates(subset= text ) print(f 刪除了 {(len_before-
len(df_issues))/len_before:.2%} 重複項。 )
```

刪除了 1.88% 的重複項。

我們可以看到我們的數據集中有一些重複的問題，但它們只佔很小的比例。正如我們在其他章節中所做的那樣，快速查看文本中的單詞數量也是一個好主意，看看當我們截斷每個模型的上下文大小時是否會丟失很多信息：

將numpy導入為np將
matplotlib.pyplot導入為plt

```
(df_issues[ text ].str.split().apply(len)
    .hist(bins=np.linspace(0,500,50),grid=False,edgecolor= C0 ))
plt.title( 每期字數 ) plt.xlabel( 字
數 ) plt.ylabel( 期數 ) plt.show()
```



該分佈具有許多文本數據集的長尾特徵。大多數文本都很短，但也有超過 500 字的問題。一些很長的問題很常見，尤其是當錯誤消息和代碼片段與它們一起發佈時。鑑於大多數 transformer 模型的上下文大小為 512 個標記或更大，截斷少數長問題不太可能影響整體性能。現在我們已經探索並清理了我們的數據集，最後要做的是定義我們的訓練和驗證集來對我們的分類器進行基準測試。

讓我們來看看如何做到這一點。

創建訓練集創建訓練集和

驗證集對於多標籤問題有點棘手，因為不能保證所有標籤的平衡。但是，它可以近似，我們可以使用 [Scikit-multilearn 庫](#)，這是專門為此目的而設立的。我們需要做的第一件事是將我們的標籤集（如pytorch和標記化）轉換為模型可以處理的格式。在這裡，我們可以使用 Scikit-learn 的多標籤二值化器類，它採用標籤名稱列表並創建一個向量，其中缺少的標籤為零，現有標籤為零。我們可以通過在 all_labels 上安裝 Multi Label Binarizer 來測試這一點，以學習從標籤名稱到 ID 的映射，如下所示：

從 [sklearn.preprocessing](#) 導入 MultiLabelBinarizer

```
mlb = MultiLabelBinarizer()
mlb.fit([all_labels])
mlb.transform([[ tokenization , new model ],[ pytorch ]])
```

```
陣列 ([[0, 0, 0, 1, 0, 0, 0, 1, 0],
[0, 0, 0, 0, 1, 0, 0, 0]])
```

在這個簡單的例子中，我們可以看到第一行有兩個對應於標記化和新模型標籤，而第二行只有一個與pytorch 的匹配。

要創建拆分，我們可以使用Scikit-multilearn 中的iterative_train_test_split()函數，它會迭代創建訓練/測試拆分以實現標籤平衡。我們將其包裝在一個可以應用於DataFrame 的函數中。由於該函數需要一個二維特徵矩陣，因此我們需要在進行拆分之前為可能的索引添加一個維度：

從skmultilearn.model_selection導入iterative_train_test_split

```
def balanced_split(df, test_size=0.5): ind =
    np.expand_dims(np.arange(len(df)), axis=1) labels =
    mlb.transform(df[ labels ]) ind_train, _, ind_test, _ =
    iterative_train_test_split(ind,標籤,
                                test_size)返回
    df.iloc[ind_train[:, 0]], df.iloc[ind_test[:, 0]]
```

借助balanced_split()函數，我們可以將數據拆分為監督和非監督數據集，然後為監督部分創建平衡的訓練、驗證和測試集：

從sklearn.model_selection導入train_test_split

```
df_clean = df_issues[[ text , labels , split ]].reset_index(drop=True).copy() df_unsup =
df_clean.loc[df_clean[ split ]== unlabeled ,[ 文本 , labels ]] df_sup = df_clean.loc[df_clean[ split ] ==
labeled ,[ text , labels ]]

np.random.seed(0)
df_train, df_tmp = balanced_split(df_sup, test_size=0.5) df_valid, df_test =
balanced_split(df_tmp, test_size=0.5)
```

最後，讓我們創建一個包含所有拆分的DatasetDict，以便我們可以輕鬆標記數據集並與 Trainer 集成。在這裡，我們將使用漂亮的from_pandas()方法直接從相應的 Pandas DataFrame 加載每個拆分：

從數據集導入數據集， DatasetDict

```
ds = DatasetDict({ train :
    Dataset.from_pandas(df_train.reset_index(drop=True)), 有效 :
    Dataset.from_pandas(df_valid.reset_index(drop=True)),  test :
    Dataset.from_pandas(df_test.reset_index(drop=True)),  unsup :
    Dataset.from_pandas(df_unsup.reset_index(drop=True))})
```

這看起來不錯，所以最後要做的是創建一些訓練切片，以便我們可以評估每個分類器的性能作為訓練集大小的函數。

創建訓練切片數據集具有我

們要在本章中研究的兩個特徵：稀疏標記數據和多標籤分類。訓練集僅包含 220 個可供訓練的示例，即使使用遷移學習，這也無疑是一個挑戰。為了深入了解本章中的每種方法在標記數據很少的情況下如何執行，我們還將創建樣本更少的訓練數據切片。然後我們可以根據性能繪製樣本數量並研究各種制度。我們將從每個標籤只有 8 個樣本開始，並使用 `iterative_train_test_split()` 函數逐步構建直到切片覆蓋整個訓練集：

```
np.random.seed(0)
all_indices = np.expand_dims(list(range(len(ds['train']))), axis=1)
indices_pool =
all_indices labels = mlb.transform(ds['train'][“標籤”])
train_samples = [8, 16, 32, 64,
128]
train_slices, last_k = [], 0
```

```
對於 i, k in enumerate(train_samples) :
    # 拆分填充下一個拆分大小所需的樣本 indices_pool, labels, new_slice, _ =
    iterative_train_test_split(indices_pool, labels, (train_slices +追加(new_slice))否則 i == 0:
        train_slices.append (np.concatenate ((train_slices [-1], new_slice)))
```

#添加完整數據集作為最後一個切片

請注意，這種迭代方法只能將樣本近似地拆分為所需的大小，因為在給定的拆分大小下並不總是能夠找到平衡的拆分：

```
print( 目標拆分大小： )
print(train_samples) print( 實際拆分
大小： ) print([len(x) for x in
train_slices])
```

目標拆分大小 :[8、16、32、
64、128、223]
實際拆分大小 :[10, 19, 36,
68, 134, 223]

我們將使用指定的拆分大小作為以下繪圖的標籤。太好了，我們終於將數據集準備好進行訓練拆分了！接下來讓我們看一下如何訓練一個強大的基線模型！

實現樸素貝葉斯線

每當你開始一個新的 NLP 項目時，實施一組強大的基線總是一個好主意。這有兩個主要原因：

1. 基於正則表達式、手工規則或非常簡單的模型的基線可能已經很好地解決了問題。在這些情況下，沒有理由拿出像變壓器這樣的大砲，它們在生產環境中的部署和維護通常更加複雜。
2. 基線可在您探索更複雜的模型時提供快速檢查。例如，假設您訓練 BERT-large 並在驗證集上獲得 80% 的準確率。您可能會將其作為硬數據集註銷並收工。但是，如果您知道像邏輯回歸這樣的簡單分類器可以達到 95% 的準確率呢？這會引起您的懷疑並促使您調試您的模型。

因此，讓我們通過訓練基線模型來開始我們的分析。對於文本分類，一個很好的基線是樸素貝葉斯分類器，因為它非常簡單，訓練速度快，並且對輸入的擾動相當穩健。樸素貝葉斯的 Scikit-learn 實現不支持開箱即用的多標籤分類，但幸運的是，我們可以再次使用 Scikit-multilearn 庫將問題轉化為一對一的分類任務，我們為 L 訓練 L 個二元分類器標籤。首先，讓我們使用多標籤二值化器在我們的訓練集中創建一個新的label_ids列。我們可以使用map()函數一次性完成所有處理：

```
def prepare_labels (批處理) :
    batch[ 'label_ids' ] = mlb.transform(batch[ 'labels' ])返回批次
```

```
ds = ds.map(prepare_labels, batched=True)
```

為了衡量我們的分類器的性能，我們將使用微觀和宏觀F1分數，其中前者跟蹤頻繁標籤上的性能，後者跟蹤所有標籤上的性能，而不考慮頻率。由於我們將在不同大小的訓練拆分中評估每個模型，因此讓我們創建一個帶有列表的defaultdict來存儲每個拆分的分數：

從集合中導入defaultdict

```
macro_scores, micro_scores = defaultdict(列表), defaultdict(列表)
```

現在我們終於準備好訓練我們的基線了！下面是用於訓練模型並在增加訓練集大小的情況下評估我們的分類器的代碼：

```
從sklearn.naive_bayes導入MultinomialNB從sklearn.metrics導入classification_report從skmultilearn.problem_transform導入BinaryRelevance從sklearn.feature_extraction.text導入CountVectorizer
```

```

對於train_slices中的train_slice：
# 獲取訓練切片和測試數據ds_train_sample =
ds[ train ].select(train_slice) y_train =
np.array(ds_train_sample[ label_ids ]) y_test =
np.array(ds[ test ][ label_ids ])
# 使用簡單的計數向量器將我們的文本編碼為標記計數count_vect = CountVectorizer()

X_train_counts = count_vect.fit_transform(ds_train_sample[ 文本 ])
X_test_counts = count_vect.transform(ds[ test ][ text ])
# 創建並訓練我們的模型！分類器 =
BinaryRelevance(分類器=MultinomialNB()) classifier.fit(X_train_counts,
y_train)
# 生成預測並評估y_pred_test =
classifier.predict(X_test_counts) clf_report = classification_report(
    y_test, y_pred_test, target_names=mlb.classes_, zero_division=0, output_dict=True)

# 存儲指標
macro_scores[ Naive Bayes ].append(clf_report[ macro avg ][ f1-score ])
micro_scores[ Naive Bayes ].append(clf_report[ micro avg ][ f1-score ])

```

這段代碼中包含很多內容，所以讓我們對其進行解壓。首先，我們得到訓練切片並對標籤進行編碼。然後我們使用計數向量器通過簡單地創建一個詞彙量大小的向量來對文本進行編碼，其中每個條目對應於一個標記在文本中出現的頻率。這被稱為詞袋方法，因為所有關於詞序的信息都丟失了。然後我們訓練分類器並使用測試集上的預測通過分類報告獲得微觀和宏觀F1分數。

使用以下輔助函數，我們可以繪製該實驗的結果：

將matplotlib.pyplot導入為plt

```

def plot_metrics(micro_scores, macro_scores, sample_sizes, current_model):圖,(ax0, ax1)=
    plt.subplots(1, 2, figsize=(10, 4), sharey=True)

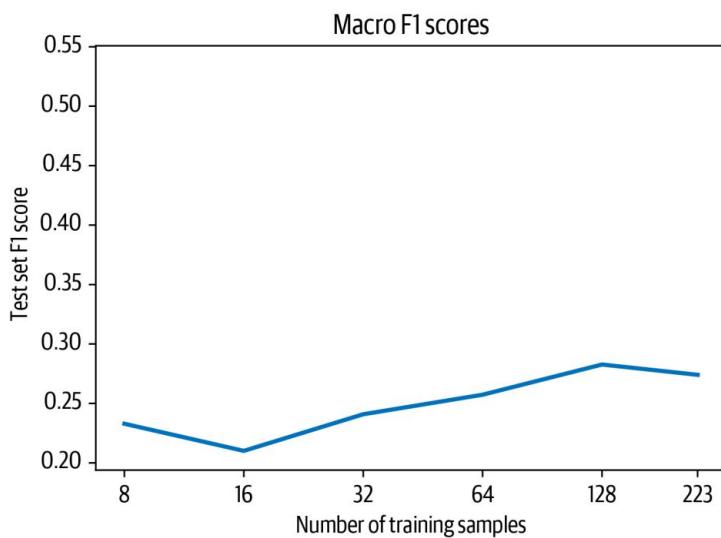
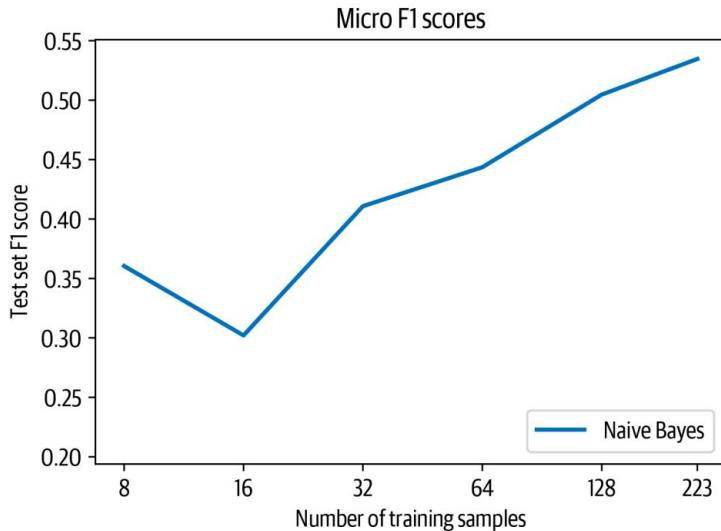
    在micro_scores.keys()中運行：如果運行 ==
        current_model :
            ax0.plot(sample_sizes, micro_scores[run], label=run, linewidth=2) ax1.plot(sample_sizes,
                macro_scores[run], label=run, linewidth=2)其他: ax0.plot(sample_sizes, micro_scores[run],
                label=run, linestyle= 虛線 ) ax1.plot(sample_sizes, macro_scores[run], label=run,
                linestyle= 虛線 )

            ax0.set_title( Micro F1 scores )
            ax1.set_title( Macro F1 scores )
            ax0.set_ylabel( Test set F1 score )
            ax0.legend(loc= lower right ) for ax in [ax0,
            ax1]:

```

```
ax.set_xlabel( 訓練樣本數 ) ax.set_xscale( log )
ax.set_xticks(sample_sizes) ax.set_xticklabels(sample_sizes)
ax.minorticks_off() plt.tight_layout() plt.show()
```

```
plot_metrics (micro_scores, macro_scores, train_samples, “樸素貝葉斯”)
```



請注意，我們在對數刻度上繪製了樣本數。從圖中我們可以看出，隨著訓練樣本數量的增加，微觀和宏觀F1分數都有所提高。由於要訓練的樣本太少，結果也有輕微的噪音，因為每個切片可能有不同的類別分佈。儘管如此，這裡重要的是趨勢，所以現在讓我們看看這些結果與基於轉換器的方法相比如何！

使用無標籤數據

我們要考慮的第一種技術是零樣本分類，它適用於根本沒有標記數據的設置。這在行業中非常普遍，並且可能會發生，因為沒有帶標籤的歷史數據，或者因為獲取數據的標籤很困難。我們將在本節中稍微作弊，因為我們仍將使用測試數據來衡量性能，但我們不會使用任何數據來訓練模型（否則將很難與以下方法進行比較）。

零樣本分類的目標是利用預訓練模型，而無需對特定任務的語料庫進行任何額外的微調。為了更好地了解這是如何工作的，請回想一下，像 BERT 這樣的語言模型經過預訓練，可以預測成千上萬本書和大型維基百科轉儲中文本中的掩碼標記。為了成功預測缺失的標記，模型需要了解上下文中的主題。我們可以嘗試通過提供如下句子來欺騙模型為我們對文檔進行分類：

“本節的主題是 [MASK]。”

然後模型應該對文檔的主題給出合理的建議，因為這是數據集中出現的自然文本。²

讓我們用下面的玩具問題進一步說明這一點：假設你有兩個孩子，其中一個喜歡有汽車的電影，而另一個更喜歡有動物的電影。不幸的是，他們已經看過所有你知道的，所以你想構建一個函數來告訴你一部新電影是關於什麼主題的。自然地，您求助於變形金剛來完成這項任務。首先要嘗試的是在填充掩碼管道中加載 BERT-base，它使用掩碼語言模型來預測掩碼標記的內容：

[從變壓器導入管道](#)

```
pipe = pipeline( fill-mask ,model= bert-base-uncased )
```

²我們感謝喬·戴維森向我們建議這種方法。

接下來，讓我們構建一個小電影描述，並用一個掩碼詞添加一個提示。提示的目標是引導模型幫助我們進行分類。

填充掩碼管道返回最有可能填充掩碼點的標記：

```
movie_desc = 電影 madagascar \的主要角色是獅子、斑馬、長頸鹿和河馬。 prompt
= 電影是關於 [MASK] 的。
```

```
output = pipe(movie_desc + prompt)用於輸出中的
元素：
print(f 代幣{element[ token_str ]}:{element[ score ]:.3f}% )
象徵性動物 :0.103% 象徵性獅子：
象徵性鳥類 :象徵性愛： 0.066%
               0.025%
               0.015%
代幣狩獵 :0.013%
```

顯然，該模型僅預測與動物相關的標記。我們也可以扭轉這個局面，我們可以查詢管道以獲得一些給定標記的概率，而不是獲取最有可能的標記。對於這個任務，我們可能會選擇汽車和動物，這樣我們就可以將它們作為目標傳遞給管道：

```
output = pipe(movie_desc + prompt, targets=[ animals , cars ])對於輸出中的元素：
print(f 代幣{element[ token_str ]}:{element[ score ]:.3f}% )
代幣動物 :0.103% 代幣汽車：
               0.001%
```

不出所料，代幣汽車的預測概率遠小於動物。讓我們看看這是否也適用於更接近汽車的描述：

```
movie_desc = 在電影《變形金剛》中，外星人可以變形為各種各樣的交
通工具。
output = pipe(movie_desc + prompt, targets=[ animals , cars ])對於輸出中的元素：
print(f 代幣{element[ token_str ]}:{element[ score ]:.3f}% )
代幣汽車 :0.139% 代幣動物：
               0.006%
```

確實如此！這只是一個簡單的例子，如果我們想確保它運行良好，我們應該對其進行徹底的測試，但它說明了本章中討論的許多方法的關鍵思想：找到一種方法，使預訓練模型無需訓練即可適應另一項任務- 正在處理它。在這種情況下，我們設置了一個帶有掩碼的提示，這樣我們就可以直接使用掩碼語言模型進行分類。讓我們看看我們是否可以通過調整一個模型來做得更好，該模型已經針對更接近文本分類的任務進行了微調：自然語言推理 (NLI)。

使用掩碼語言模型進行分類是一個很好的技巧，但我們可以通過使用在更接近分類的任務上訓練過的模型來做得更好。

有一個稱為文本蘊涵的巧妙代理任務符合要求。在文本蘊含中，模型需要確定兩個文本段落是否可能相互遵循或相互矛盾。模型通常經過訓練以檢測與多類型 NLI 語料庫 (MNLI) 或跨語言 NLI 語料庫 (XNLI) 等數據集的蘊涵和矛盾。³ 這些數據集中的每個樣本都由三部分組成：前提、假設和標籤，可以是蘊涵、中性或矛盾之一。當假設文本在前提下必然為真時，賦予蘊涵標籤。

當假設在前提下必然為假或不恰當時，使用矛盾標籤。如果這些情況都不適用，則分配中性標籤。有關每個的示例，請參見表 9-1。

表 9-1。 MNLI 數據集中的三個類

前提	假設	標籤
他最喜歡的顏色是藍色。	重金屬音樂。	中性的
她覺得這個笑話很好笑。她覺得這個笑話一點都不好笑。矛盾房子是最近蓋的。房子是新的。		
蘊涵		

現在，事實證明我們可以劫持在 MNLI 數據集上訓練的模型來構建分類器，而根本不需要任何標籤！關鍵思想是將我們希望分類的文本作為前提，然後將假設表述為：

“這個例子是關於{label}的。”

我們在其中插入標籤的類名。然後，蘊涵分數告訴我們該前提與該主題相關的可能性有多大，我們可以按順序對任意數量的類運行它。這種方法的缺點是我們需要為每個類執行前向傳遞，這使得它的效率低於標準分類器。

另一個有點棘手的方面是標籤名稱的選擇會對準確性產生很大影響，選擇具有語義意義的標籤通常是最好的方法。例如，如果標籤只是 1 類，則模型沒有暗示這可能意味著什麼以及這是否構成矛盾或蘊涵。

 Transformers 內置了一個用於零樣本分類的 MNLI 模型。我們可以 ini- 通過管道對其進行初始化，如下所示：

³ A. Williams、N. Nangia 和 SR Bowman，“通過推理理解句子的廣泛覆蓋挑戰語料庫”，(2018); A. Conneau 等人，“XNLI：評估跨語言句子表示”，(2018)。

從變壓器導入管道

```
pipe = pipeline( 零射擊分類 ,device=0)
```

設置device=0確保模型在 GPU 而不是默認 CPU 上運行以加速推理。要對文本進行分類，我們只需將其與標籤名稱一起傳遞給管道。此外，我們可以設置multi_label=True以確保返回所有分數，而不僅僅是單標籤分類的最大值：

```
示例= ds[ train ][0]
print(f Labels: {sample[ labels ]} )output
= pipe(sample[ text ],all_labels,multi_label=True) print(output[ sequence
][:400]) print( \n預測: )
```

對於標籤，zip中的分數（輸出[“標籤”]，輸出[“分數”]）：

```
print(f {label},{score:.2f} )
```

標籤 [新模型]

添加新的 CANINE 模型

```
# ★ 新機型追加
```

型號說明

Google 最近提出了一個新的 **C** 字符 **A** 架構，具有 **N**o 標記化 **I**n **N**eural **E**ncoders 架構 (CANINE)。不僅標題令人興奮：

> 流水線 NLP 系統在很大程度上已被端到端神經建模所取代，但幾乎所有常用模型仍需要顯式標記

預測 :新模型、

0.98 tensorflow 或 tf、

0.37 示例、0.34 用法、0.30

pytorch、0.25 文檔、0.25 模型訓

練、0.24 標記化、0.17 管道、0.16



由於我們使用的是子詞分詞器，我們甚至可以將代碼傳遞給模型！標記化可能不是很有效，因為只有一小部分零樣本管道的預訓練數據集由代碼片段組成，但由於代碼也由很多自然詞組成，這不是一個大問題。此外，代碼塊可能包含重要信息，例如框架 (PyTorch 或 TensorFlow)。

我們可以看到該模型非常有信心該文本是關於一個新模型的，但它對其他標籤也產生了相對較高的分數。零鏡頭分類的一個重要方面是我們正在操作的領域。我們在這裡處理的文本非常技術性，主要與編碼有關，這使得它們與 MNLI 數據集中的原始文本分佈有很大不同。因此，這對模型來說是一項具有挑戰性的任務也就不足為奇了；對於某些領域，它可能比其他領域更有效，具體取決於它們與訓練數據的接近程度。

讓我們編寫一個函數，通過零樣本管道提供單個示例，然後通過運行map()將其擴展到整個驗證集：

```
def zero_shot_pipeline (示例) :
    output = pipe(example[ text ], all_labels, multi_label=True)
    example[ predicted_labels ] = output[ labels ]示例[ scores ] = output[ scores ]
    返回示例
```

```
ds_zero_shot = ds[ 有效 ].map(zero_shot_pipeline)
```

現在我們有了分數，下一步是確定應為每個示例分配哪組標籤。我們可以嘗試幾個選項：

- 定義一個閾值並選擇所有高於閾值的標籤。
- 選擇得分最高的前k 個標籤。

為了幫助我們確定哪種方法最好，讓我們編寫一個get_preds()函數，它應用其中一種方法來檢索預測：

```
def get_preds (示例 ,閾值=無 , topk=無) :
    preds = []如果
    閾值：
        對於標籤， zip中的分數 (示例 [ predicted_labels ] ,示例 [ scores ]) ：
            如果分數>=閾值：
                preds.append (標籤)
    elif topk: for i
        in range(topk):
            preds.append(example[ predicted_labels ][i])
    別的：
        raise ValueError( Set either `threshold` or `topk` . )return
    { pred_label_ids :list(np.squeeze(mlb.transform([preds])))}
```

接下來，讓我們編寫第二個函數get_clf_report()，它從具有預測標籤的數據集中返回 Scikit-learn 分類報告：

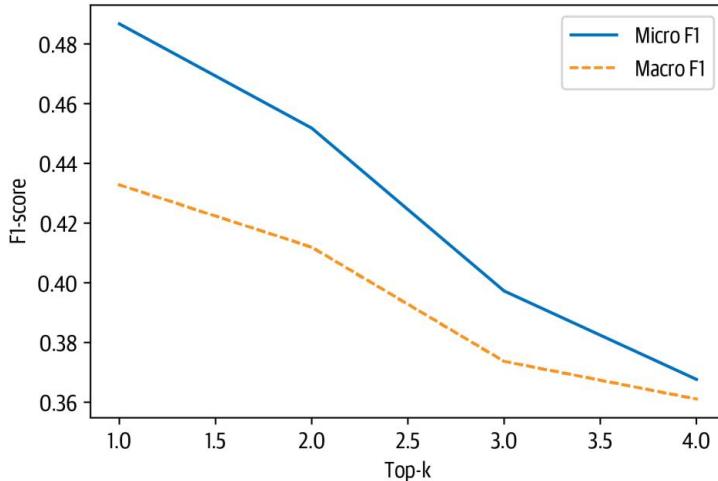
```
def get_clf_report(ds):y_true =
    np.array(ds[ label_ids ]) y_pred =
    np.array(ds[ pred_label_ids ])返回分類報告(
        y_true, y_pred, target_names=mlb.classes_, zero_division=0, output_dict=True)
```

有了這兩個函數，讓我們從 top-k 方法開始，通過增加幾個值的 k，然後繪製驗證中的微觀和宏觀 F1 分數
放：

```
macros, micros = [], []
topks = [1, 2, 3, 4]
for topk in topks:
    ds_zero_shot = ds_zero_shot.map(get_preds, batched=False, fn_kwarg={topk : topk})

    clf_report = get_clf_report(ds_zero_shot)
    micros.append(clf_report['micro avg']['f1-score'])
    macros.append(clf_report['macro avg']['f1-score'])

plt.plot(topks, micros, label='Micro F1')
plt.plot(topks, macros, label='Macro F1')
plt.xlabel('Top-k')
plt.ylabel('F1-score')
plt.legend(loc='best')
plt.show()
```



從圖中我們可以看出，最好的結果是通過選擇每個示例得分最高的標籤（前 1）獲得的。考慮到我們數據集中的大多數示例只有一個標籤，這也許並不令人驚訝。現在讓我們將其與設置閾值進行比較，這樣我們就可以為每個示例預測多個標籤：

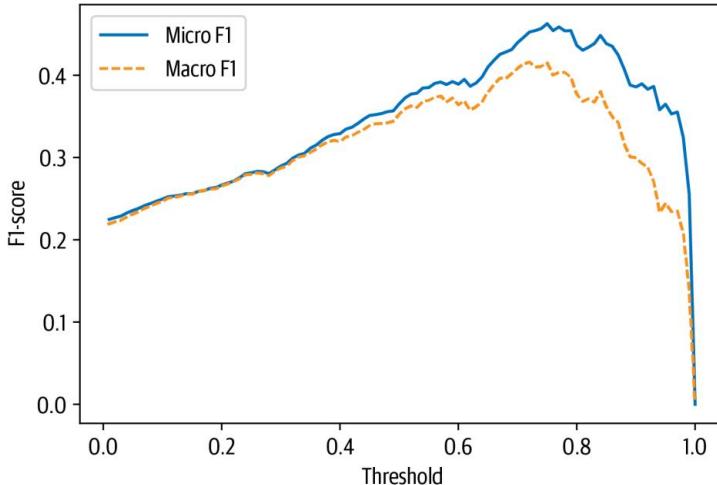
```
macros, micros = [], []
thresholds = np.linspace(0.01, 1, 100)
for thresholds in thresholds:
    ds_zero_shot = ds_zero_shot.map(get_preds,
                                    fn_kwarg={'閾值': thresholds})
    clf_report = get_clf_report(ds_zero_shot)
```

```

micros.append(clf_report[ micro avg ][ f1-score ])
macros.append(clf_report[ macro avg ][ f1-score ])

plt.plot(thresholds, micros, label= Micro F1 )
plt.plot(thresholds, macros, label= Macro F1 )
plt.xlabel( Threshold ) plt.ylabel( F1-score ) plt.圖例 (loc
= “最佳” ) plt.show ()

```



```

best_t, best_micro = 閾值[np.argmax(micros)], np.max(micros) print(f Best threshold
(micro): {best_t} with F1-score {best_micro:.2f}. ) best_t, best_macro = 閾值[np.argmax(macros)],
np.max(macros) print(f Best threshold (macro): {best_t} with F1-score {best_macro:.2f}. )

```

最佳閾值（微觀）: 0.75，F1 分數為 0.46。
最佳閾值（宏觀）: 0.72，F1 分數為 0.42。

這種方法比 top-1 結果稍差，但我們可以在這張圖中清楚地看到準確率/召回率的權衡。如果我們將閾值設置得太低，那麼預測就會太多，從而導致精度不高。如果我們將閾值設置得太高，那麼我們幾乎不會做出任何預測，從而產生較低的召回率。從圖中我們可以看出，0.8 左右的閾值是兩者之間的最佳點

二。

由於 top-1 方法表現最好，讓我們用它來比較零樣本分類和測試集上的樸素貝葉斯：

```

ds_zero_shot = ds[ test ].map(zero_shot_pipeline) ds_zero_shot
= ds_zero_shot.map(get_preds, fn_kwarg={ topk : 1}) clf_report =
get_clf_report(ds_zero_shot) 用於train_slices中的train_slice :

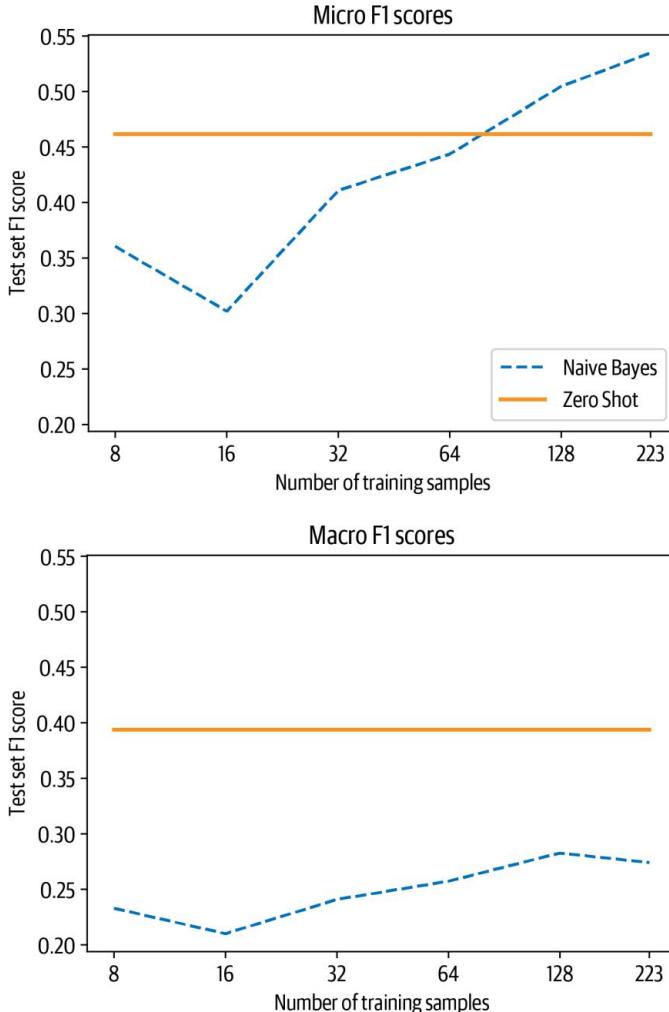
```

```

macro_scores[ Zero Shot ].append(clf_report[ macro avg ][ f1-score ])
micro_scores[ Zero Shot ].append(clf_report[ micro avg ][ f1-score ])

plot_metrics(micro_scores, macro_scores, train_samples, Zero Shot )

```



將零樣本管道與基線進行比較，我們觀察到兩件事：

1. 如果我們的標記樣本少於 50 個，零樣本流水線的性能會輕鬆優於基線。
2. 即使超過 50 個樣本，零樣本流水線的性能在同時考慮微觀和宏觀F1分數時也更勝一籌。微F1分數的結果告訴我們，基線在頻繁類別上表現良好，而

零樣本管道在這些方面表現出色，因為它不需要任何示例來學習。



您可能會注意到本節中的一個輕微悖論：儘管我們談論不處理標籤，但我們仍然使用驗證集和測試集。我們使用它們來展示不同的技術，並使它們之間的結果具有可比性。即使在真實的用例中，收集一些帶標籤的示例來運行一些快速評估也是有意義的。重要的一點是我們沒有根據數據調整模型的參數；相反，我們只是調整了一些超參數。

如果您發現很難在自己的數據集上獲得好的結果，可以採取以下措施來改進零樣本管道：

- 管道的工作方式使其對標籤名稱非常敏感。如果名稱沒有多大意義或不易與文本聯繫起來，管道的性能可能會很差。要么嘗試使用不同的名稱，要么同時使用多個名稱並在一個額外的步驟中聚合它們。
 - 您可以改進的另一件事是假設的形式。默認情況下它是 `hypothesis = This is example is about {}`，但您可以將任何其他文本傳遞到管道。
- 根據用例，這可能會提高性能。

現在讓我們轉向我們有一些可用於訓練模型的標記示例的機制。

使用幾個標籤

在大多數 NLP 項目中，您至少可以訪問一些帶標籤的示例。標籤可能直接來自客戶或跨公司團隊，或者您可能決定坐下來自己註釋幾個示例。即使對於之前的方法，我們也需要一些帶標籤的示例來評估零樣本方法的效果。

在本節中，我們將了解如何最好地利用我們擁有的少數珍貴的帶標籤示例。讓我們從一種稱為數據增強的技術開始，它可以幫助我們成倍增加我們擁有的少量標記數據。

數據增強 提高文本分類器

在小型數據集上的性能的一種簡單但有效的方法是應用數據增強技術從現有的訓練示例中生成新的訓練示例。這是計算機視覺中的一種常見策略，其中隨機擾動圖像而不改變數據的含義（例如，稍微

旋轉的貓仍然是貓）。對於文本，數據擴充有點棘手，因為擾亂單詞或字符可以完全改變含義。比如“大象比老鼠重嗎？”這兩個問題和“老鼠比大象重嗎？”相差一個單詞交換，但有相反的答案。然而，如果文本由多個句子組成（就像我們的 GitHub issues 那樣），那麼這些類型的轉換引入的噪音通常不會影響標籤。在實踐中，有兩種常用的數據增強技術：

回譯 使用源語言的

文本，使用機器翻譯將其翻譯成一種或多種目標語言，然後再將其翻譯回源語言。反向翻譯往往最適用於資源豐富的語言或不包含太多特定領域單詞的語料庫。

Token perturbations

給定訓練集中的文本，隨機選擇並執行簡單的轉換，如隨機同義詞替換、單詞插入、交換或刪除。⁴

表 9-2 中顯示了這些轉換的示例。有關 NLP 的其他數據增強技術的詳細列表，我們建議閱讀 Amit Chaudhary 的博客文章 “[NLP 中數據增強的可視化調查](#)”。

表 9-2。不同類型的文本數據增強技術

強化	句子
沒有任何	就算你打敗我威震天，別人也會起來打敗你的暴政
同義詞替換	就算你殺了我威震天，其他人也能打敗你的暴政
隨機插入	即使你打敗了我威震天，其他人類也會奮起反抗你的暴政
隨機交換	你就算打敗我威震天，別人也會反抗你的
隨機刪除	就算你我威震天，別人打敗暴政
back translation (德語)	就算你打敗了我，別人也會奮起反抗你的暴政

您可以使用[M2M100](#)等機器翻譯模型實現反向翻譯，而像[NLpAug](#)這樣的庫和[文本攻擊](#)為令牌擾動提供各種方法。在本節中，我們將重點介紹同義詞替換的使用，因為它易於實現並且可以理解數據擴充背後的主要思想。

⁴ J. Wei 和 K. Zou， “[EDA :用於提高文本分類性能的簡單數據增強技術-化任務](#)”， (2019)。

我們將使用NLpAug 的ContextualWordEmbsAug增強器來利用 DistilBERT 的上下文詞嵌入來進行同義詞替換。讓我們從一個簡單的例子開始：

```
從變形金剛導入set_seed導入
nlpAug.augmenter.word作為naw

set_seed(3)
aug = naw.ContextualWordEmbsAug(model_path= "distilbert-base-uncased" ,
    設備= "cpu" ·動作= "替代品" )

text = 變形金剛是最受歡迎的玩具 print(f 原始文本 :{text} )
print(f 增強文本 :{aug.augment(text)} )
```

原文 :變形金剛是最受歡迎的玩具
 增強文字 :變形金剛是最受歡迎的玩具

在這裡，我們可以看到“are”這個詞是如何被一個撇號代替的，以生成一個新的綜合訓練示例。我們可以將這種擴充包裝在一個簡單的函數中，如下所示：

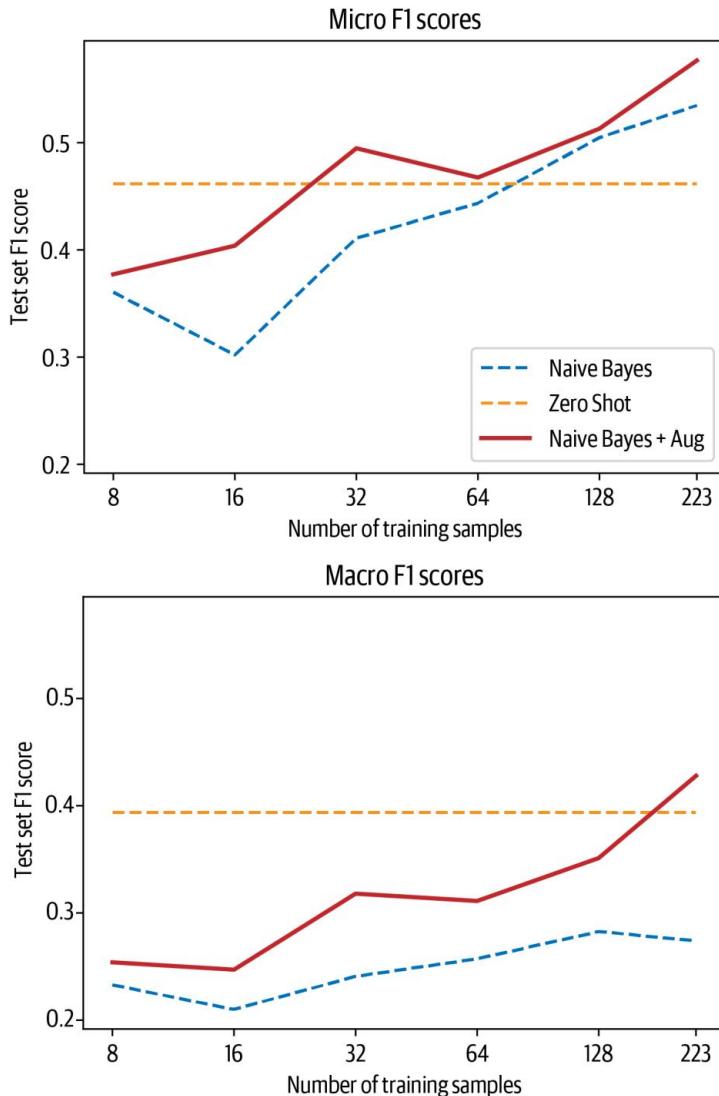
```
def augment_text (批次, transformations_per_example=1) :
    text_aug, label_ids = [], []
    for text,
        labels in zip(batch[ text ], batch[ label_ids ]):
        text_aug += [text]
        label_ids += [labels] for in
        range(transformations_per_example): text_aug +=
            [aug.augment(text)] label_ids += [labels] return
    { text : text_aug, label_ids : label_ids}
```

現在，當我們將此函數傳遞給map()方法時，我們可以使用transformations_per_example參數生成任意數量的新示例。我們可以在我們的代碼中使用這個函數來訓練樸素貝葉斯分類器，只需在我們選擇切片後添加一行：

```
ds_train_sample = ds_train_sample.map(augment_text, batched=True,
    remove_columns=ds_train_sample.column_names).shuffle(種子=42)
```

包括此並重新運行分析會生成此處顯示的圖：

```
plot_metrics(micro_scores, macro_scores, train_samples, "樸素貝葉斯 + Aug" )
```



從圖中，我們可以看到，少量的數據增強將樸素貝葉斯分類器的F1分數提高了大約5個點，並且一旦我們有大約170個訓練樣本，它就超過了宏觀分數的零樣本管道。現在讓我們看一下基於使用大型語言模型嵌入的方法。

使用嵌入作為查找表大型語言

模型（例如 GPT-3）已被證明在解決數據有限的任務方面表現出色。原因是這些模型學習了有用的文本表示，這些表示在許多維度上對信息進行編碼，例如情感、主題、文本結構等。出於這個原因，大型語言模型的嵌入可用於開發語義搜索引擎，查找相似的文檔或評論，甚至對文本進行分類。

在本節中，我們將創建一個以[OpenAI API 分類端點為模型的文本分類器](#)。這個想法遵循三個步驟：

1. 使用語言模型嵌入所有帶標籤的文本。
2. 對存儲的嵌入執行最近鄰搜索。
3. 聚合最近鄰的標籤以獲得預測。

該過程如圖 9-3 所示，它顯示了標記數據如何嵌入模型並與標籤一起存儲。當需要對新文本進行分類時，它也會被嵌入，並根據最近鄰的標籤給出標籤。

校準要搜索的鄰居數量很重要，因為太少可能會產生噪音，太多可能混入相鄰組。

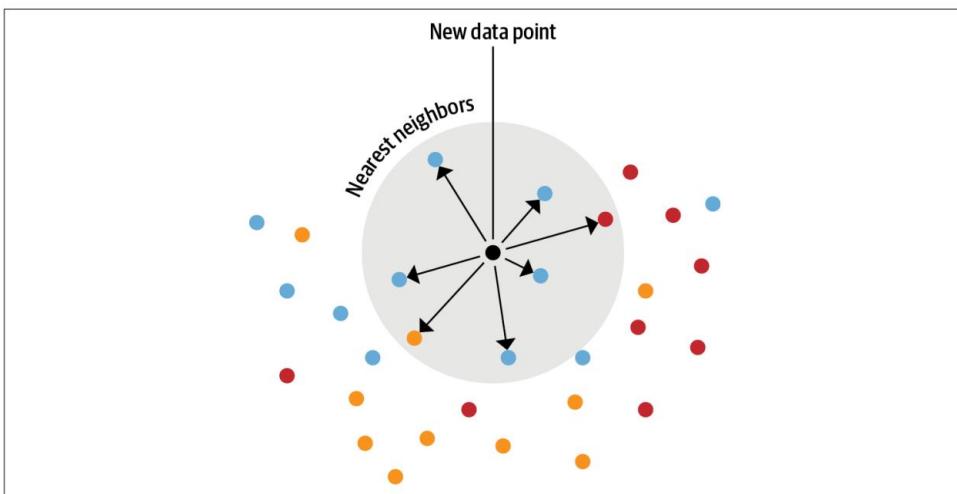


圖 9-3。最近鄰嵌入查找的圖示

這種方法的美妙之處在於，無需對模型進行微調即可利用少數可用的標記數據點。相反，使這種方法起作用的主要決定是選擇一個合適的模型，該模型最好在與您的數據集相似的域上進行預訓練。

由於 GPT-3 只能通過 OpenAI API 獲得，我們將使用 GPT-2 來測試該技術。具體來說，我們將使用在 Python 代碼上訓練的 GPT-2 變體，它有望捕獲我們 GitHub 問題中包含的一些上下文。

讓我們編寫一個輔助函數，它接受一個文本列表並使用該模型為每個文本創建一個單向量表示。我們必須處理的一個問題是，像 GPT-2 這樣的轉換器模型實際上將為每個標記返回一個嵌入向量。

例如，給定句子 “I take my dog for a walk”，我們可以預期有多個嵌入向量，每個標記一個。但我們真正想要的是整個句子的單個嵌入向量（或我們應用程序中的 GitHub issue）。為了解決這個問題，我們可以使用一種稱為池化的技術。最簡單的池化方法之一是對標記嵌入進行平均，這稱為均值池化。對於均值池化，我們唯一需要注意的是我們不在平均值中包含填充標記，因此我們可以使用注意力掩碼來處理它。

為了解這是如何工作的，讓我們加載一個 GPT-2 分詞器和模型，定義均值池化操作，並將整個過程包裝在一個簡單的 `embed_text()` 函數中：

```
從變壓器導入火炬
導入AutoTokenizer, AutoModel

model_ckpt = "miguelvictor/python-gpt2-large" tokenizer
= AutoTokenizer.from_pretrained(model_ckpt) 模型 =
AutoModel.from_pretrained(model_ckpt)

def mean_pooling(model_output, attention_mask): # 提取令牌嵌
    入token_embeddings = model_output[0]

    # 計算注意力掩碼input_mask_expanded
    =
    (attention_mask .unsqueeze(-1) .expand(token_embeddings.size()) .float())

    # 對嵌入求和，但忽略屏蔽標記sum_embeddings =
    torch.sum(token_embeddings * input_mask_expanded, 1) sum_mask =
    torch.clamp(input_mask_expanded.sum(1), min=1e-9)
    # 將平均值作為單個向量返回sum_embeddings /
    sum_mask

def embed_text (示例) :
    inputs = tokenizer(examples[ text ], padding=True, truncation=True,
                      max_length=128, return_tensors= pt ) with
    torch.no_grad(): model_output = model(**inputs) pooled_embeds =
    mean_pooling(model_output, inputs[ attention_mask ]) return
    { embedding : pooled_embeds.cpu().numpy()}
```

現在我們可以獲得每個拆分的嵌入。請注意，GPT 樣式模型沒有填充令牌，因此我們需要添加一個才能獲得嵌入。

以前面代碼中實現的批處理方式。為此，我們將只回收字符串結尾的標記：

```
tokenizer.pad_token = tokenizer.eos_token
embs_train =
= ds[ train ].map(embed_text, batched=True, batch_size=16) embs_valid =
ds[ valid ].map(embed_text, batched=True, batch_size=16) embs_test =
ds[ test ].map(embed_text, batched=True, batch_size=16)
```

現在我們有了所有的嵌入，我們需要建立一個系統來搜索它們。我們可以編寫一個函數來計算我們將查詢的新文本嵌入與訓練集中現有嵌入之間的餘弦相似度。或者，我們可以使用稱為 FAISS 索引的內置數據集結構。5我們已經在第 7 章中遇到過 FAISS。您可以將其視為嵌入的搜索引擎，稍後我們將仔細研究它的工作原理。我們可以使用數據集的現有字段通過 `add_faiss_index()` 索引，或者我們可以使用 `add_faiss_index_from_external_arrays()` 將新嵌入加載到數據集中。

讓我們使用前一個函數將我們的訓練嵌入添加到數據集中，如下所示：

```
embs_train.add_faiss_index( 嵌入 )
```

這創建了一個新的 FAISS 索引，稱為嵌入。我們現在可以通過調用函數 `get_nearest_examples()` 來執行最近鄰查找。它返回最近的鄰居以及每個鄰居的匹配分數。我們需要指定查詢嵌入以及要檢索的最近鄰居的數量。讓我們試一試，看看最接近示例的文檔：

```
i, k = 0, 3 # 選擇第一個查詢和3個最近的鄰居rn, nl = '\r\n\r\n\r\n' , \n # 用於去除文本
中的換行符以緊湊顯示

query = np.array(embs_valid[i][ embedding ], dtype=np.float32) 分數, samples =
embs_train.get_nearest_examples( embedding , query, k=k)

print(f QUERY LABELS: {embs_valid[i][ labels ]} )
print(f QUERY TEXT:\n{embs_valid[i][ text ][:200].replace(rn, nl)} [...] \n ) print( = *50) print(f 檢
索到的文檔: ) for score, label, text in zip(scores, samples[ labels ], samples[ text ]):

    print( = *50)
    print(f TEXT:\n{text[:200].replace(rn, nl)} [...] ) print(f SCORE:
{score:.2f} ) 打印 ( f “標籤 :{標籤}” )
```

```
查詢標籤 :[ 新模型 ]
查詢文本 :
在 T5 中實現高效的自我關注
```

5 J. Johnson、M. Douze 和 H. Jégou， “使用 GPU 進行十億級相似性搜索”，(2017)。

★ 新模型追加

我和我的隊友 (包括@ice-americano)想使用高效的自註意力方法 ,例如 Linformer 、Performer 和 [...]

=====

檢索到的文件 :

=====

文本 :

添加 Linformer 模型

Linformer 新模型添加到庫中模型地址 ###

Linear Complexity 論文於 6 月 9 日

在 ArXiv 上發表 :<https://arxiv.org/abs/2006.04768> La [...]

分數 :54.92 標籤 :

[新模型]

文本 :

添加 FAVOR+ / 表演者關注

★ FAVOR+ / 表演者註意力加法

是否有任何計劃將這個新的注意力近似塊添加到 Transformers 庫中 ?

模型描述 n [...]

分數 :57.90

標籤 :[新模型]

文本 :

實施 DeLight :非常深和輕量級的變形金剛

★ 新模型追加

模型描述 DeLight ,提供與帶有

符號 [...]

得分 :60.12

標籤 :[新模型]

好的 !這正是我們所希望的 :我們通過嵌入查找得到的三個檢索到的文檔都有相同的標籤 ,我們已經可以從標題中看出它們非常相似 。查詢和檢索到的文檔圍繞著添加新的高效轉換器模型 。然而 ,問題仍然存在 ,k 的最佳值是多少 ?同樣 ,我們應該如何聚合檢索到的文檔的標籤 ?例如 ,我們是否應該檢索三個文檔並分配至少出現兩次的所有標籤 ?或者我們應該選擇 20 次並使用至少出現 5 次的所有標籤 ?讓我們系統地研究一下 :我們將嘗試 k 的多個值 ,然後使用輔助函數改變標籤分配的閾值 $m < k$ 。

我們將記錄每個設置的宏觀和微觀表現 ,以便我們稍後可以決定哪個運行表現最好 。而不是遍歷驗證集中的每個樣本

我們可以使用函數`get_nearest_examples_batch()`，它接受一批查詢：

```
def get_sample_preds (樣本 米) :
    返回(np.sum(sample[ label_ids ], axis=0) >= m).astype(int)

def find_best_k_m(ds_train, valid_queries, valid_labels, max_k=17):
    max_k = min(len(ds_train), max_k) perf_micro =
    = np.zeros((max_k, max_k)) perf_macro =
    np.zeros((max_k, max_k)) for k in range(1, max_k):

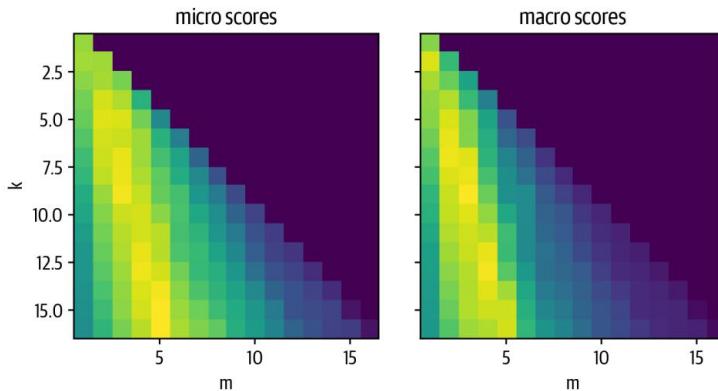
        對於m in range(1, k + 1): _ , samples
            = ds_train.get_nearest_examples_batch( embedding ,
                有效查詢 , k=k)
        y_pred = np.array([get_sample_preds(s, m) for s in samples]) clf_report =
        classification_report(valid_labels, y_pred,
            target_names=mlb.classes_, zero_division=0, output_dict=True)
        perf_micro[k, m] = clf_report[ micro avg ][ f1-score ] perf_macro[k,
        m] = clf_report[ macro avg ][ f1-score ] 返回perf_micro, perf_macro
```

讓我們檢查所有訓練樣本的最佳值是多少，並可視化所有 k 和 m 配置的分數：

```
valid_labels = np.array(embs_valid[ label_ids ]) valid_queries
= np.array(embs_valid[ embedding ], dtype=np.float32) perf_micro, perf_macro =
find_best_k_m(embs_train, valid_queries, valid_labels)

圖, (ax0, ax1) = plt.subplots(1, 2, figsize=(10, 3.5), sharey=True) ax0.imshow(perf_micro)
ax1.imshow(perf_macro)

ax0.set_title( micro scores )
ax0.set_ylabel( k ) ax1.set_title( macro
scores ) for ax in [ax0, ax1]:
    ax.set_xlim([0.5, 17 - 0.5]) ax.set_ylim(
    ([17 - 0.5, 0.5])) ax.set_xlabel( m )
plt.show()
```



從圖中我們可以看出存在一種模式。對於給定的 k 選擇太大的 m 會產生次優結果。選擇大約 $m/k = 1/3$ 的比率時可獲得最佳性能。讓我們看看哪個 k 和 m 給出了最好的整體結果：

```
k, m = np.unravel_index(perf_micro.argmax(), perf_micro.shape)
print(f'最佳k: {k},\n最佳m: {m} )
```

最佳 k:15 ,最佳 m:5

當我們選擇 $k = 15$ 和 $m = 5$ 時，或者換句話說，當我們檢索 15 個最近的鄰居然後分配出現至少 5 次的標籤時，性能最好。既然我們有了找到嵌入查找的最佳值的好方法，我們就可以玩與朴素貝葉斯分類器相同的遊戲，在其中我們遍歷訓練集的切片並評估性能。在我們對數據集進行切片之前，我們需要刪除索引，因為我們不能像數據集那樣對 FAISS 索引進行切片。其餘循環保持完全相同，增加了使用驗證集來獲得最佳 k 和 m 值：

```
embs_train.drop_index( embedding )
test_labels = np.array(embs_test[ label_ids ]) test_queries
= np.array(embs_test[ embedding ], dtype=np.float32)
```

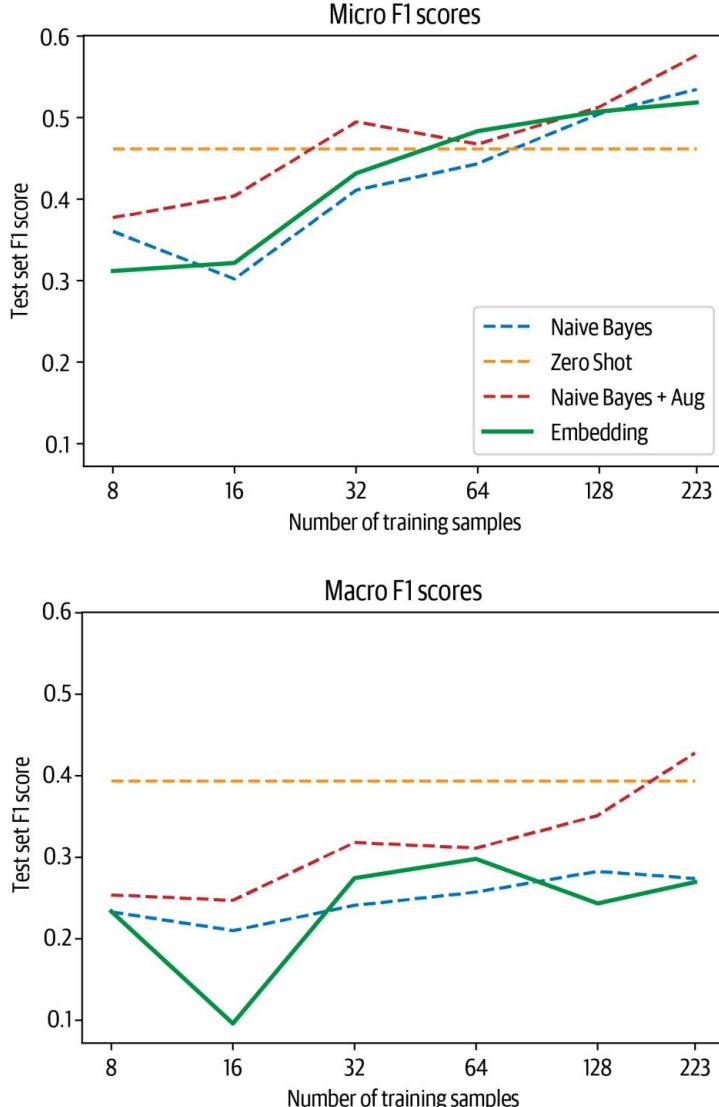
對於train_slices中的train_slice：

```
# 從訓練切片創建 Faiss 索引embs_train_tmp =
embs_train.select(train_slice)
embs_train_tmp.add_faiss_index( embedding )
# 使用驗證集獲取最佳 k \ m 值perf_micro, _ =
find_best_k_m(embs_train_tmp, test_labels, k, m, np.unravel_index(perf_micro.argmax(), perf_micro.shape))

# 獲取測試集_, samples =
embs_train_tmp.get_nearest_examples_batch( embedding , test_queries, k=int(k))
y_pred =
np.array([get_sample_preds(s, m) for s in samples])
# 評估預測
```

```
clf_report = classification_report (測試標籤， y_pred ,
                                    target_names=mlb.classes_, zero_division=0, output_dict=True)
macro_scores[ "Embedding" ].append(clf_report[ "macro avg" ][ "f1-score" ])
micro_scores[ "Embedding" ].append(clf_report[ "微平均" ][ "f1-score" ])

plot_metrics (micro_scores, macro_scores, train_samples, "嵌入")
```



嵌入查找在微觀分數上與以前的方法相比具有競爭力，同時只有兩個“可學習”參數 k 和 m ，但在宏觀分數上表現稍差。

對這些結果持保留態度；哪種方法效果最好很大程度上取決於領域。零樣本管道的訓練數據與我們正在使用它的 GitHub issues 數據集有很大不同，後者包含許多模型可能以前沒有遇到過的代碼。對於更常見的任務，例如評論的情感分析，管道可能會工作得更好。同樣，嵌入的質量取決於模型和訓練它的數據。我們嘗試了六個模型，例如 sentence-transformers/stsb-roberta-large，經過訓練可以提供高質量的句子嵌入，以及 microsoft/codebert-base 和 dbernsohn/roberta-python，它們經過代碼訓練和文檔。對於這個特定的用例，在 Python 代碼上訓練的 GPT-2 效果最好。

由於除了替換模型檢查點名稱以測試另一個模型外，您實際上不需要更改代碼中的任何內容，因此一旦設置了評估管道，您就可以快速嘗試幾個模型。

現在讓我們將這種簡單的嵌入技巧與根據我們擁有的有限數據簡單地微調轉換器進行比較。

使用 FAISS 進行高效相似性搜索我們在第
7章 中首次遇到 FAISS，我們使用它通過 DPR 嵌入檢索文檔。在這裡，我們將簡要說明 FAISS 庫的工作原理以及它為何是 ML 工具箱中的強大工具。

我們習慣於在維基百科等龐大數據集或使用谷歌等搜索引擎的網絡上執行快速文本查詢。當我們從文本轉移到嵌入時，我們希望保持這種性能；然而，用於加速文本查詢的方法不適用於嵌入。

為了加速文本搜索，我們通常創建一個倒排索引，將術語映射到文檔。倒排索引就像書末的索引：每個單詞都映射到它出現的頁面（或者在我們的例子中是文檔）。當我們稍後運行查詢時，我們可以快速查找在哪些文檔中出現搜索詞。這適用於單詞等離散對象，但不適用於矢量等連續對象。每個文檔可能都有一個唯一的向量，因此索引永遠不會與新向量匹配。我們需要尋找接近或相似的匹配，而不是尋找完全匹配。

當我們想要在數據庫中找到與查詢向量最相似的向量時，理論上我們需要將查詢向量與數據庫中的 n 個向量中的每一個進行比較。
對於像本章中那樣的小型數據庫，這沒有問題，但是如果我們

將其擴展到數千甚至數百萬個條目，我們需要等待一段時間才能處理每個查詢。

FAISS 通過幾個技巧解決了這個問題。主要思想是對數據集進行分區。如果我們只需要將查詢向量與數據庫的一個子集進行比較，我們就可以顯著加快這個過程。但是，如果我們只是對數據集進行隨機分區，我們如何決定搜索哪個分區，以及我們找到最相似條目的保證是什麼？顯然，必須有更好的解決方案：將 k-means 聚類應用於數據集！這通過相似性將嵌入聚類到組中。此外，對於每個組，我們得到一個質心向量，它是該組所有成員的平均值（圖 9-4）。

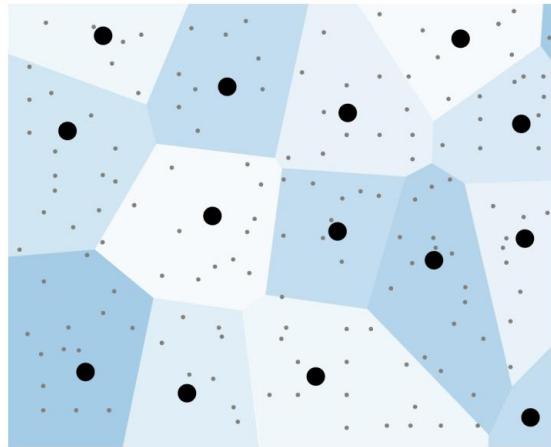


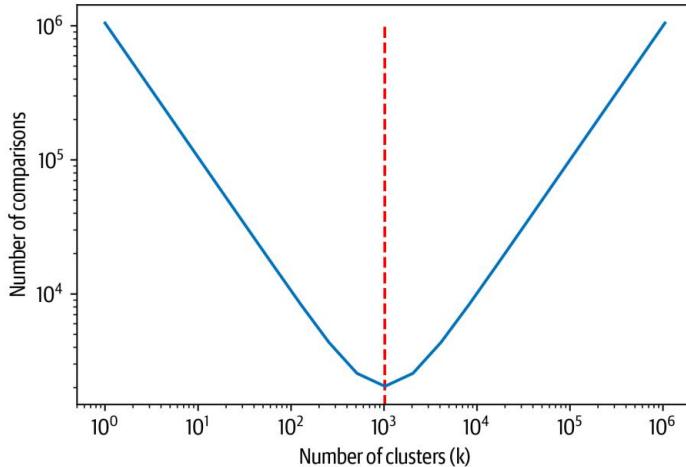
圖 9-4。FAISS 索引的結構：灰色點代表加入索引的數據點，粗黑點是通過 k-means 聚類找到的聚類中心，彩色區域代表屬於聚類中心的區域

給定這樣的分組，在 n 個向量中搜索要容易得多：我們首先在 k 個質心中搜索與我們的查詢最相似的質心 (k 個比較)，

然後我們在組內搜索（要比較的 kn 個元素）。這將比較次數從 n 減少到 $k + \frac{n}{k}$ 。所以問題是， k 的最佳選擇是什麼？

如果太小，每組還有很多樣本需要我們在第二步中進行比較；如果 k 太大，我們需要搜索的質心很多。

通過尋找函數的最小值 $f_k = k + \text{find } k = n$ 。事實上，我們關於 k ，我們可以用下圖將其可視化，其中 $n = 220$ 。



在圖中，您可以看到比較次數是聚類數的函數。我們正在尋找這個函數的最小值，我們需要做最少的比較。我們可以看到最小值正是我們期望看到的 = $210 = 1,024$ 。

它，在 $\sqrt{20}$

除了通過分區加速查詢之外，FAISS 還允許您利用 GPU 進一步加速。如果內存成為一個問題，還有幾個選項可以使用高級量化方案壓縮矢量。如果你想為你的項目使用 FAISS，存儲庫有一個簡單的[指南](#)供您為您的用例選擇正確的方法。

使用 FAISS 的最大項目之一是 Facebook 創建的 CCMatrix 語料庫。作者使用多語言嵌入來查找不同語言的平行句子。這個龐大的語料庫隨後被用來訓練**M2M100**，一個大型機器翻譯模型，能夠直接在 100 種語言中的任何一種之間進行翻譯。

微調 Vanilla Transformer 如果

我們可以訪問標記數據，我們也可以嘗試做一件顯而易見的事情：簡單地微調一個預訓練的 Transformer 模型。在本節中，我們將使用標準 BERT 檢查點作為起點。稍後，我們將看到微調語言模型對性能的影響。



對於許多應用程序，從預訓練的類 BERT 模型開始是個好主意。然而，如果你的語料庫的領域與預訓練語料庫（通常是維基百科）有很大不同，你應該探索 Hugging Face Hub 上可用的許多模型。很可能有人已經在您的域上預訓練了模型！

讓我們從加載預訓練的分詞器開始，對我們的數據集進行分詞，並刪除訓練和評估不需要的列：

```
從變壓器導入火炬
導入 (AutoTokenizer, AutoConfig,
      AutoModelForSequenceClassification)

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

def tokenize(batch): return
    tokenizer(batch[ text ], truncation=True, max_length=128) ds_enc = ds.map(tokenize,
batched=True) ds_enc = ds_enc.remove_columns([ labels , text ])
```

多標籤損失函數期望標籤為浮點類型，因為它還允許類概率而不是離散標籤。因此，我們需要更改列label_ids 的類型。由於按元素更改列的格式不能很好地適應 Arrow 的類型化格式，我們將採取一些解決方法。首先，我們創建一個帶有標籤的新列。該列的格式是從第一個元素推斷出來的。然後我們刪除原來的列並重命名新的一列來代替原來的列：

```
ds_enc.set_format( torch )
ds_enc = ds_enc.map(lambda x: { label_ids_f :x[ label_ids ].to(torch.float)},
remove_columns=[ label_ids ]) ds_enc = ds_enc.rename_column
( label_ids_f , label_ids )
```

由於訓練數據的大小有限，我們很可能會很快過擬合，因此我們設置
load_best_model_at_end=True並根據微觀選擇最佳模型
F1 分數：

```
從變形金剛進口培訓師，TrainingArguments

training_args_fine_tune = TrainingArguments(
    output_dir= ./results , num_train_epochs=20, learning_rate=3e-5,
    lr_scheduler_type= constant , per_device_train_batch_size=4,
    per_device_eval_batch_size=32, weight_decay=0.0, evaluation_strategy= epoch ,
    save_strategy= epoch , logging_strategy= "epoch" , load_best_model_at_end=True ,
    metric_for_best_model= micro f1 , save_total_limit=1, log_level= error )
```

我們需要F1分數來選擇最佳模型，因此我們需要確保在評估期間計算它。因為模型返回對數，我們首先需要用一個sigmoid函數對預測進行歸一化，然後用一個簡單的閾值將它們二值化。然後我們從分類報告中返回我們感興趣的分數：

```
來自scipy.special import expit作為sigmoid

def compute_metrics (預測) :
    y_true = pred.label_ids y_pred =
    = sigmoid(pred.predictions) y_pred =
    (y_pred>0.5).astype(float)

    clf_dict = classification_report(y_true, y_pred, target_names=all_labels,
                                      zero_division=0, output_dict=True) return
    { micro f1 :clf_dict[ micro avg ][ f1-score ], macro f1 :clf_dict[ macro avg ]
      [ f1-score ] }
```

現在我們準備好隆隆聲了！對於每個訓練集切片，我們從頭開始訓練一個分類器，在訓練循環結束時加載最佳模型，並將結果存儲在測試集上：

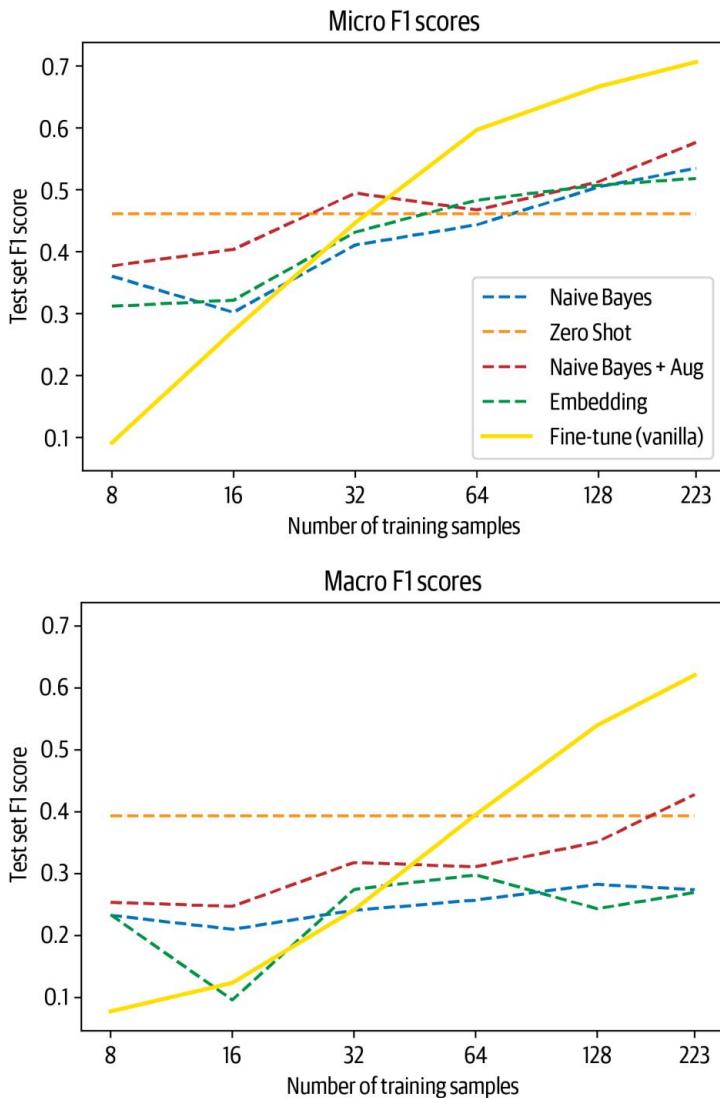
```
config = AutoConfig.from_pretrained(model_ckpt)
config.num_labels = len(all_labels) config.problem_type =
multi_label_classification

對於train_slices中的train_slice：
    模型= AutoModelForSequenceClassification.from_pretrained(model_ckpt, config=config)

    培訓師=培訓師（模型=模
        型， tokenizer=tokenizer，
        args=training_args_fine_tune，
        compute_metrics=compute_metrics，
        train_dataset=ds_enc[ “train” ].select (train_slice) ，
        eval_dataset=ds_enc[ “valid” ]，）

    trainer.train() pred
    =trainer.predict(ds_enc[ test ]) metrics =
    compute_metrics(pred) macro_scores[ Fine-tune
    (vanilla) ].append(metrics[ macro f1 ]) micro_scores[ 精細-tune
    (vanilla) ].append(metrics[ micro f1 ])

    plot_metrics (macro_scores, micro_scores, train_samples, “微調（香草）” )
```



首先，我們看到，當我們可以訪問大約 64 個示例時，只需對數據集上的普通 BERT 模型進行微調即可獲得具有競爭力的結果。我們還看到，在此之前行為有點不穩定，這也是由於在小樣本上訓練模型，其中一些標籤可能不平衡。在我們使用數據集的未標記部分之前，讓我們快速看一下另一種在少鏡頭域中使用語言模型的有前途的方法。

帶提示的上下文和小樣本學習我們在本章

前面看到，我們可以使用像 BERT 或 GPT-2 這樣的語言模型，並通過使用提示和解析模型的標記預測來使其適應監督任務。這不同於添加特定任務頭並為任務調整模型參數的經典方法。有利的一面是，這種方法不需要任何訓練數據，但不利的一面是，如果我們可以訪問標記數據，我們似乎無法利用它。我們有時可以利用一個中間地帶，稱為上下文學習或少樣本學習。

為了說明這個概念，考慮一個英語到法語的翻譯任務。在零鏡頭範式中，我們將構建一個提示，如下所示：

```
提示= “” “”
將英語翻譯成法語 :謝謝 =>
```

這有望促使模型預測單詞“merci”的標記。我們已經在第 6 章中看到使用 GPT-2 進行摘要時，將“TL;DR”添加到文本中會促使模型生成摘要，而無需明確訓練來執行此操作。GPT-3 論文的一個有趣發現是大型語言模型能夠有效地從提示中提供的示例中學習。因此，之前的翻譯示例可以增加幾個英語到德語的示例，這將使模型表現得更好在這個任務上。⁶

此外，作者發現模型縮放得越大，它們就越能更好地使用上下文中的示例，從而顯著提高性能。

儘管 GPT-3 大小的模型在生產中使用具有挑戰性，但這是一個令人興奮的新興研究領域，人們已經構建了很酷的應用程序，例如自然語言 shell，其中命令以自然語言輸入並由 GPT-3 解析為外殼命令。

使用標記數據的另一種方法是創建提示和所需預測的示例，並繼續在這些示例上訓練語言模型。

一種稱為 ADAPET 的新穎方法使用了這種方法，並在各種任務上擊敗了 GPT-3，⁷ 並使用生成的提示調整模型。Hugging Face 研究人員最近的工作表明，與微調自定義頭部相比，這種方法的數據效率更高。⁸

⁶ T. Brown 等人，“語言模型是小概率學習者”，(2020)。

⁷ D. Tam 等人，“改進和簡化模式開發培訓”，(2021)。

⁸ T. Le Scao 和 AM Rush，“一個提示值多少個數據點？”，(2021)。

在本節中，我們簡要介紹了充分利用我們擁有的少數標記示例的各種方法。很多時候，除了帶標籤的示例之外，我們還可以訪問大量未標記的數據；在下一節中，我們將討論如何充分利用它。

利用未標記的數據

儘管能夠訪問大量高質量標記數據是訓練分類器的最佳情況，但這並不意味著未標記數據毫無價值。想一想我們使用的大多數模型的預訓練：即使它們是根據來自互聯網的大部分不相關的數據進行訓練的，我們也可以利用預訓練的權重來處理各種文本上的其他任務。這就是 NLP 中遷移學習的核心思想。自然地，如果下游任務與預訓練文本具有相似的文本結構，遷移效果會更好，因此如果我們可以使預訓練任務更接近下游目標，我們就有可能改進遷移。

讓我們根據我們的具體用例考慮一下：BERT 在 BookCorpus 和英文維基百科上進行了預訓練，包含代碼和 GitHub 問題的文本在這些數據集中絕對是一個小眾市場。例如，如果我們從頭開始預訓練 BERT，我們可以通過爬取 GitHub 上的所有問題來完成。然而，這會很昂貴，而且 BERT 學到的很多關於語言的方面對於 GitHub issues 仍然有效。那麼，在從頭開始重新訓練和僅使用模型進行分類之間是否存在中間地帶？有，它被稱為領域適應（我們在第 7 章的問答中也看到了）。無需從頭開始重新訓練語言模型，我們可以繼續使用我們領域的數據對其進行訓練。在這一步中，我們使用經典語言模型目標來預測掩碼詞，這意味著我們不需要任何標記數據。之後，我們可以將調整後的模型作為分類器加載並對其進行微調，從而利用未標記的數據。

域適應的美妙之處在於，與標記數據相比，未標記數據通常是大量可用的。此外，經過調整的模型可以在許多用例中重複使用。想像一下，您想要構建一個電子郵件分類器並對所有歷史電子郵件應用域適應。您稍後可以使用相同的模型進行命名實體識別或其他分類任務（如情感分析），因為該方法與下游任務無關。

現在讓我們看看微調預訓練語言模型需要採取的步驟。

微調語言模型在本節中，我們將在數據集

的未標記部分上使用掩碼語言建模來微調預訓練的 BERT 模型。為此，我們只需要兩個新的

概念：標記數據和特殊數據整理器時的額外步驟。讓我們從標記化開始。

除了文本中的普通標記之外，標記器還向序列添加特殊標記，例如用於分類和下一句預測的[CLS]和[SEP]標記。當我們進行屏蔽語言建模時，我們希望確保我們不會訓練模型來預測這些標記。

出於這個原因，我們將它們從損失中屏蔽掉，並且我們可以在標記化時通過設置

`return_special_tokens_mask=True` 獲得一個掩碼。讓我們使用該設置重新標記文本：

```
def tokenize(batch): return
    tokenizer(batch[ "text" ], truncation=True,
              最大長度=128, return_special_tokens_mask=True)

ds_mlm = ds.map(tokenize, batched=True) ds_mlm
= ds_mlm.remove_columns([ "labels" , "text" , "label_ids" ])
```

從屏蔽語言建模開始缺少的是屏蔽輸入序列中的標記並在輸出中包含目標標記的機制。我們解決這個問題的一種方法是設置一個函數來屏蔽隨機標記並為這些序列創建標籤。但這會使數據集的大小加倍，因為我們還將目標序列存儲在數據集中，這意味著我們將在每個時期使用相同的序列掩碼。

一個更優雅的解決方案是使用數據整理器。請記住，數據收集器是在數據集和模型調用之間架起橋樑的函數。從數據集中採樣一個批次，數據整理器準備批次中的元素以將它們提供給模型。在我們遇到的最簡單的情況下，它只是將每個元素的張量連接成一個張量。在我們的例子中，我們可以使用它來動態地生成掩碼和標籤。這樣我們就不需要存儲標籤，每次採樣時我們都會得到新的面具。此任務的數據整理器稱為DataCollatorForLanguageModeling。我們使用模型的分詞器和我們希望通過mlm_probability參數屏蔽的分詞對其進行初始化。我們將使用此整理器來屏蔽 15% 的標記，這遵循 BERT 論文中的過程：

從變壓器導入DataCollatorForLanguageModeling，`set_seed`

```
data_collator = DataCollatorForLanguageModeling(tokenizer=tokenizer,
                                                mlm_probability=0.15)
```

讓我們快速瀏覽一下正在運行的數據整理器，看看它實際上做了什麼。為了在 DataFrame 中快速顯示結果，我們將分詞器和數據整理器的返回格式切換為 NumPy：

```
set_seed(3)
data_collator.return_tensors = np inputs =
tokenizer( 變形金剛真棒！ , return_tensors= np )
```

```

outputs = data_collator([{"input_ids": inputs["input_ids"][0]}])

pd.DataFrame({
    "原始代幣": tokenizer.convert_ids_to_tokens(inputs["input_ids"][0]),
    "屏蔽標記": tokenizer.convert_ids_to_tokens(outputs["input_ids"][0]),
    "原始input_ids": original_input_ids,
    "屏蔽的input_ids": masked_input_ids,
    "Labels": outputs["labels"][0]
}).T

```

	0	1	2	3	4	5
原始代幣	[CLS]	變形金剛太棒了！				[九月]
蒙面代幣	[CLS]	變形金剛很棒	[MASK]	[SEP]		
原始 input_ids 101		19081	2024	12476	999	102
屏蔽的 input_ids 101		19081	2024	12476	103	102
標籤	-100	-100	-100	-100	999	-100

我們看到感嘆號對應的token已經換成了掩碼token。此外，數據整理器返回一個標籤數組，原始標記為 -100，屏蔽標記為標記 ID。正如我們之前所見，包含 -100 的條目在計算損失時會被忽略。讓我們將數據整理器的格式切換回 PyTorch：

```
data_collator.return_tensors = "pt"
```

有了分詞器和數據整理器，我們就可以微調屏蔽語言模型了。我們像往常一樣設置 TrainingArguments 和 Trainer：

從 [變形金剛導入 AutoModelForMaskedLM](#)

```
training_args = TrainingArguments(
    output_dir=f'{model_ckpt}-issues-128',
    per_device_train_batch_size=32,
    logging_strategy='epoch',
    evaluation_strategy='epoch',
    save_strategy='no',
    num_train_epochs=16,
    push_to_hub=True,
    log_level='error',
    report_to='none'
)
```

```
培訓師=培訓師（模型
    =AutoModelForMaskedLM.from_pretrained("bert-base-uncased"),
    tokenizer=tokenizer,
    args=training_args,
    data_collator=data_collator,
    train_dataset=ds_mlm['unsup'],
    eval_dataset=ds_mlm['train'])
```

培訓師.train()

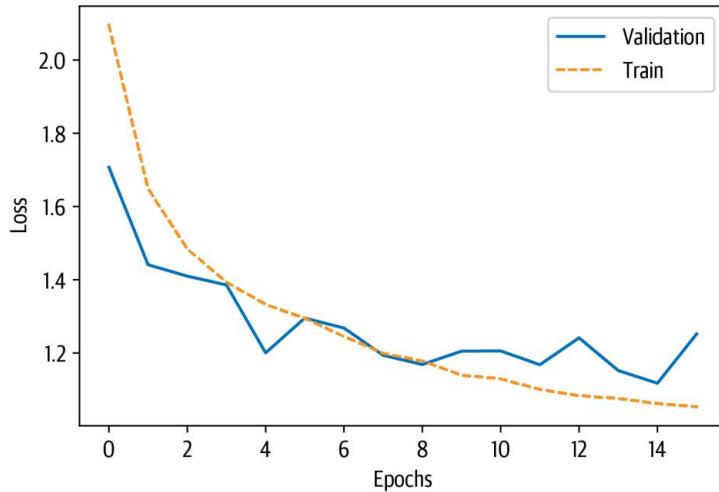
trainer.push_to_hub('訓練完成！')

我們可以訪問訓練器的日誌歷史記錄來查看模型的訓練和驗證損失。所有日誌都作為字典列表存儲在 trainer.state.log_history 中，我們可以輕鬆地將其加載到 Pandas DataFrame 中。由於訓練和驗證損失是在不同的步驟中記錄的，因此數據框中存在缺失值。出於這個原因，我們在繪製指標之前刪除了缺失值：

```
df_log = pd.DataFrame(trainer.state.log_history)

(df_log.dropna(subset=[ eval_loss ]).reset_index()[ eval_loss ]
 .plot(label= 驗證 ))
df_log.dropna(subset=[ loss ]).reset_index()[ loss ].plot(label= Train )

plt.xlabel( Epochs )
plt.ylabel( Loss )
plt.legend(loc= upper right )
plt.show()
```



似乎訓練和驗證損失都大大減少了。因此，讓我們檢查一下，當我們基於該模型微調分類器時，是否也能看到改進。

微調分類器

現在我們將重複微調過程，但略有不同的是我們加載了自己的自定義檢查點：

```
model_ckpt=f'{model_ckpt}-issues-128' config=
AutoConfig.from_pretrained(model_ckpt) config.num_labels =
len(all_labels) config.problem_type = 多標籤分類
```

對於train_slices中的train_slice：

```
模型= AutoModelForSequenceClassification.from_pretrained(model_ckpt, config=config)

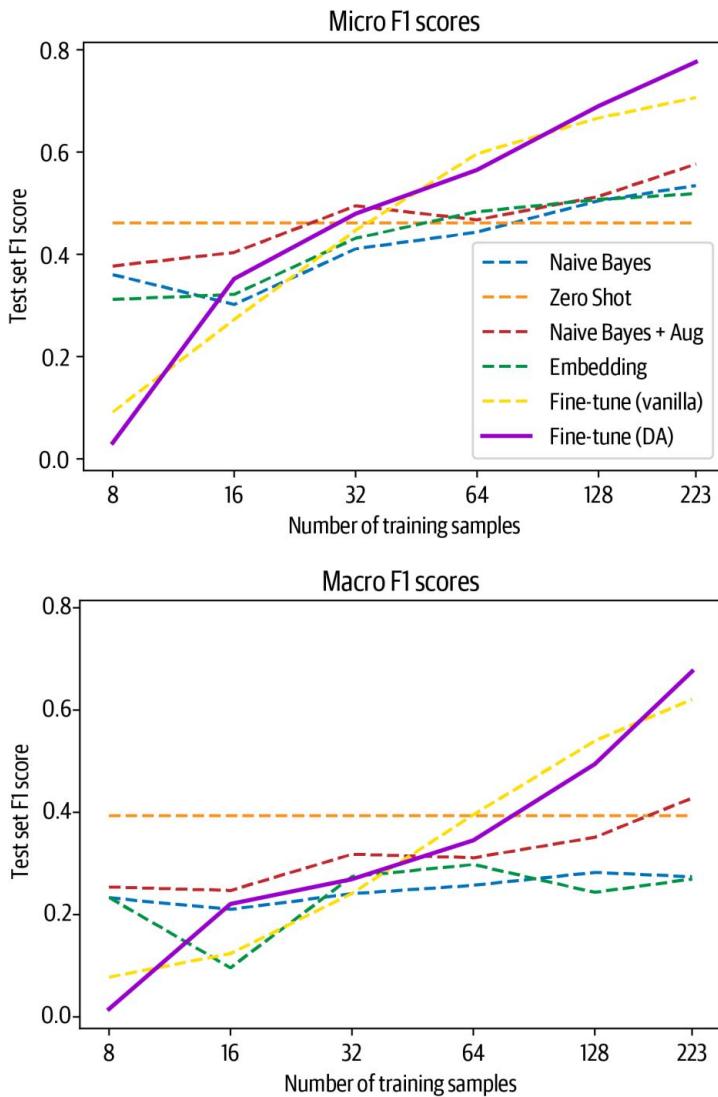
培訓師=培訓師（模型=模
型，
tokenizer=tokenizer，
args=training_args_fine_tune，
compute_metrics=compute_metrics，
train_dataset=ds_enc[“train”].select (train_slice) ，
eval_dataset=ds_enc[“有效”]，

)

trainer.train() pred
=trainer.predict(ds_enc[ test ])指標=
compute_metrics(pred)
# DA指標適配macro_scores[ Fine-tune
(DA) ].append(metrics[ macro f1 ]) micro_scores[ Fine-tune
(DA) ].append(metrics[ micro f1 ])
```

將結果與基於 vanilla BERT 的微調結果進行比較，我們發現我們獲得了優勢，尤其是在低數據領域。在有更多標記數據可用的情況下，我們也獲得了幾個百分點：

```
plot_metrics(micro_scores, macro_scores, train_samples, 微調 (DA) )
```



這突出表明，域適應可以使用未標記的數據和很少的努力來略微提升模型的性能。自然地，您擁有的未標記數據越多和標記數據越少，您使用此方法獲得的影響就越大。在結束本章之前，我們將向您展示更多利用未標記數據的技巧。

高級方法

在調整分類頭之前微調語言模型是一種提高性能的簡單而可靠的方法。然而，有一些複雜的方法可以進一步利用未標記的數據。我們在這裡總結了其中的一些方法，如果您需要更高的性能，這應該是一個很好的起點。

無監督數據擴充無監督數據擴充

充 (UDA) 背後的關鍵思想是模型的預測對於未標記的示例和稍微扭曲的示例應該是一致的。這種扭曲是通過標準數據增強策略引入的，例如令牌替換和反向翻譯。然後通過最小化原始示例和扭曲示例的預測之間的 KL 散度來強制執行一致性。這個過程如圖 9-5 所示，其中通過使用來自未標記示例的附加項來增加交叉熵損失來合併一致性要求。這意味著人們使用標準的監督方法在標記數據上訓練模型，但限制模型對未標記數據做出一致的預測。

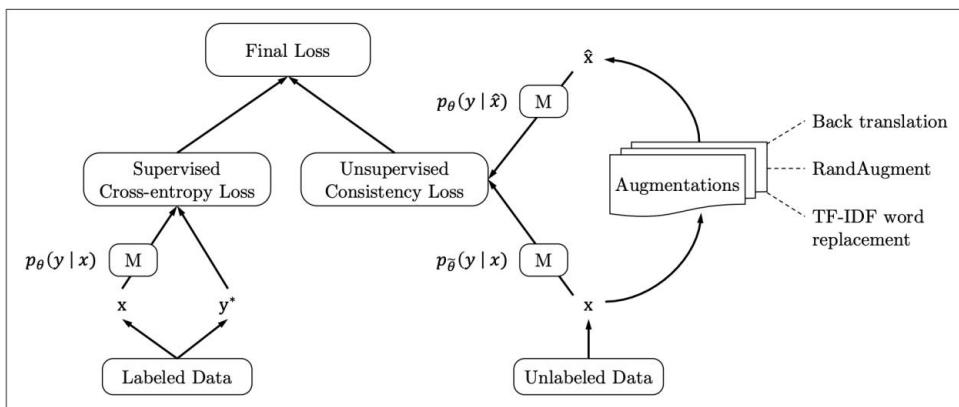


圖 9-5 使用 UDA 訓練模型 M (感謝 Qizhe Xie)

這種方法的性能令人印象深刻：使用少量標記示例，使用 UDA 訓練的 BERT 模型獲得與使用數千個示例訓練的模型相似的性能。缺點是你需要一個數據增強管道，而且訓練需要更長的時間，因為你需要多次前向傳遞來生成未標記和增強示例的預測分佈。

不確定性感知自我訓練另一種

利用未標記數據的有前途的方法是不確定性感知自我訓練 (UST)。這裡的想法是在標記數據上訓練教師模型，然後

使用該模型在未標記的數據上創建偽標籤。然後學生在偽標記數據上進行訓練，訓練後成為下一次迭代的老師。

這種方法的一個有趣方面是如何生成偽標籤：為了獲得模型預測的不確定性度量，在打開 dropout 的情況下，將相同的輸入多次輸入模型。然後，預測中的方差代表模型在特定樣本上的確定性。有了這種不確定性度量，然後使用一種稱為貝葉斯主動學習分歧 (BALD) 的方法對偽標籤進行採樣。完整的訓練流水線如圖 9-6 所示。

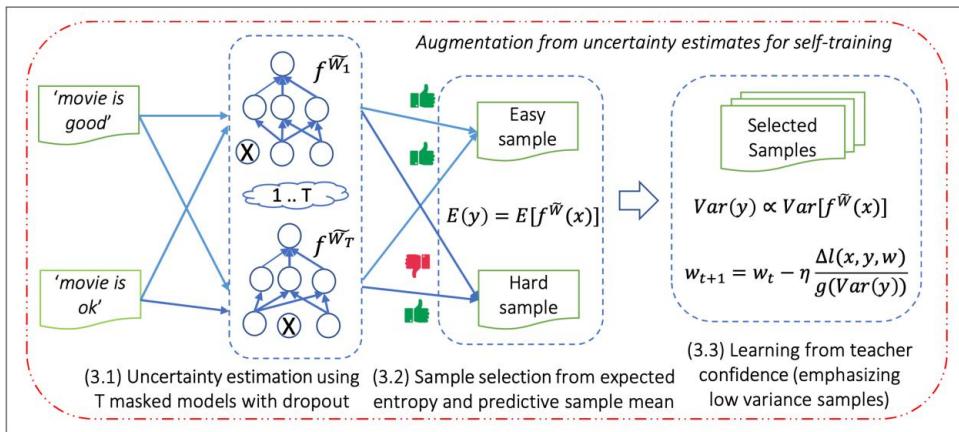


圖 9-6。UST 方法由生成偽標籤的教師和隨後接受這些標籤訓練的學生組成；學生接受培訓後成為老師並重複該步驟（由 Subhabrata Mukherjee 提供）⁹

通過這種迭代方案，教師在創建偽標籤方面不斷取得進步，從而提高了模型的性能。最終，這種方法得到了在具有數千個樣本的完整訓練數據上訓練的模型的百分之幾，甚至在幾個數據集上擊敗了 UDA。

現在我們已經看到了一些高級方法，讓我們退後一步，總結一下我們在本章中學到的內容。

⁹ S. Mukherjee 和 AH Awadallah， “小樣文本分類的不確定性感知自我訓練”，
(2020)。

結論

在本章中，我們已經看到，即使我們只有幾個標籤，甚至沒有標籤，也並非所有的希望都破滅了。我們可以利用已經在其他任務上預訓練的模型，例如 BERT 語言模型或在 Python 代碼上訓練的 GPT-2，對 GitHub 問題分類的新任務進行預測。此外，在使用普通分類頭訓練模型時，我們可以使用領域適應來獲得額外的提升。

所提出的哪種方法在特定用例上最有效取決於多個方面：您有多少標記數據、噪聲有多大、數據與預訓練語料庫的接近程度等等。要找出最有效的方法，建立評估管道然後快速迭代是個好主意。Transformers 靈活的 API 允許您快速加載少量模型並進行比較，而無需更改任何代碼。Hugging Face Hub 上有超過 10,000 個模型，過去可能有人處理過類似的問題，您可以在此基
 础上進行構建。

超出本書範圍的一個方面是更複雜的方法（如 UDA 或 UST）與獲取更多數據之間的權衡。要評估您的方法，至少在早期構建驗證和測試集是有意義的。在此過程中的每一步，您還可以收集更多標記數據。通常註釋幾百個示例需要幾個小時或幾天的工作，並且有許多工具可以幫助您完成這項工作。根據你想要實現的目標，投入一些時間來創建一個小型的、高質量的數據集是有意義的，而不是設計一個非常複雜的方法來彌補它的不足。使用我們在本章中介紹的方法，您可以確保從寶貴的標記數據中獲得最大價值。

在這裡，我們冒險進入了低數據領域，發現即使只有一百個例子，Transformer 模型仍然很強大。在下一章中，我們將研究完全相反的情況：當我們擁有數百 GB 的數據和大量計算時，我們將看看我們能做些什麼。我們將從頭開始訓練一個大型轉換器模型，為我們自動完成代碼。

第 10 章

從零開始訓練變形金剛

在本書的開頭段落中，我們提到了一個名為 GitHub Copilot 的複雜應用程序，它使用類似 GPT 的轉換器來執行代碼自動完成，這一功能在使用新語言或框架進行編程或學習編碼時特別有用，或者自动生成樣板代碼。

為此目的使用 AI 模型的其他產品包括 TabNine 和 風箏。後來，在 第5 章 中，我們仔細研究瞭如何使用 GPT 模型生成高質量文本。在本章中，我們將結束這個循環並構建我們自己的類 GPT 模型來生成 Python 源代碼！我們將生成的模型稱為 CodeParrot。

到目前為止，我們主要研究數據受限的應用程序，其中標記的訓練數據量有限。在這些情況下，遷移學習幫助我們構建了性能模型。我們在 第9章 中將遷移學習發揮到了極致，我們幾乎沒有使用任何訓練數據。

在本章中，我們將轉向另一個極端，看看當我們淹沒在我們可能想要的所有數據中時我們能做些什麼。我們將探索預訓練步驟本身，並學習如何從頭開始訓練變壓器。在解決這個問題的過程中，我們將研究一些我們尚未考慮的培訓方面，例如：

- 收集和處理一個非常大的數據集 · 為我們的數據集創建自定義分詞器
- 在多個 GPU 上大規模訓練模型

為了有效地訓練具有數十億參數的大型模型，我們需要用於分佈式訓練的特殊工具。雖然 Transformers 的 Trainer 支持分佈式訓練，但我們將藉此機會展示一個強大的 PyTorch 庫

稱為 **加速**。我們最終將觸及當今使用的一些最大的 NLP 模型，但首先，我們需要找到一個足夠大的數據集。



與本書其他代碼（可以在單個 GPU 上使用 Jupyter notebook 運行）不同，本章中的訓練代碼旨在作為腳本在多個 GPU 上運行。如果您想訓練自己的 CodeParrot 版本，我們建議運行 Transformers 存儲庫中提供的腳本。



大型數據集及其查找位置

在許多領域中，您實際上可能手頭有大量數據，從法律文件到生物醫學數據集再到編程代碼庫。在大多數情況下，這些數據集是未標記的，而且它們的大小意味著它們通常只能通過使用啟發式方法或使用在收集過程中存儲的附帶元數據來標記。

然而，一個非常大的語料庫即使在未標記或僅啟發式標記時也是有用的。我們在第 9 章中看到了一個這樣的例子，我們使用數據集的未標記部分來微調語言模型以適應領域。當可用數據有限時，這種方法通常會產生性能提升。從頭開始訓練而不是微調現有模型的決定主要取決於微調語料庫的大小以及可用的預訓練模型與語料庫之間的領域差異。

使用預訓練模型會強制您使用模型相應的分詞器，但使用在來自另一個領域的語料庫上訓練的分詞器通常不是最優的。例如，在法律文件、其他語言甚至完全不同的序列（如音符或 DNA 序列）上使用 GPT 的預訓練分詞器將導致分詞效果不佳（我們將在稍後看到）。

隨著您可以訪問的訓練數據量越來越接近用於預訓練的數據量，因此考慮從頭開始訓練模型和分詞器變得很有趣，前提是提供必要的計算資源。在我們進一步討論不同的預訓練目標之前，我們首先需要構建一個適合預訓練的大型語料庫。構建這樣一個語料庫會帶來一系列挑戰，我們將在下一節中探討這些挑戰。

構建大規模語料庫的挑戰預訓練後模型的質量很大

程度上反映了預訓練語料庫的質量。特別是，該模型將繼承預訓練語料庫中的任何缺陷。

因此，在我們嘗試創建我們自己的一個之前，最好了解一些

與為預訓練構建大型語料庫相關的常見問題和挑戰。

隨著數據集變得越來越大，您可以完全控制（或至少準確了解）其中內容的可能性會降低。一個非常大的數據集很可能不會由專門的創建者組裝而成，他們一次製作一個示例，同時了解並了解完整的管道和機器學習模型將應用於的任務。相反，通過收集作為其他活動的副作用而生成的數據，以自動或半自動的方式創建一個非常大的數據集的可能性要大得多。例如，它可能包含公司存儲的所有文檔（例如，合同、採購訂單等）、用戶活動日誌或從互聯網收集的數據。

大規模數據集大多是在高度自動化的情況下創建的，這一事實會帶來幾個重要的後果。一個重要的考慮因素是，對它們的內容和創建方式的控制有限，因此增加了在有偏見和低質量數據上訓練模型的風險。

最近對分別用於訓練 BERT 和 T5 的著名大型數據集（如 BookCorpus 和 C4）的調查發現（除其他外）¹：

- 很大一部分 C4 語料庫是機器翻譯的，而不是人工翻譯的。
- 由於 C4 中的停用詞過濾，非裔美國人英語的不同擦除導致此類內容的代表性不足。
通常很難在大型文本語料庫中找到包含（通常過多）性或其他露骨內容與完全刪除所有性或性別提及之間的中間地帶。一個令人驚訝的結果是，一個相當常見的詞，如“性”（它可以具有中性和明確的含義）對於在 C4 上訓練的分詞器來說是完全未知的，因為這個詞在語料庫中完全不存在。
- 在 BookCorpus 中有很多侵犯版權的事件，可能在其他大型數據集中也是如此。²
- BookCorpus 中的體裁偏向“浪漫”小說。

這些發現可能與在這些語料庫上訓練的模型的下游使用不兼容。例如，浪漫的強烈過度表現

1 Y. Zhu 等人，“對齊書籍和電影：通過看電影和閱讀書籍實現故事般的視覺解釋”，(2015); J. Dodge 等人，“記錄英國 Colossal Clean Crawled Corpus”，(2021)。

2 J. Bandy 和 N. Vincent，“解決機器學習研究中的文檔債務問題：BookCorpus 的回顧性數據表”，(2021)。

如果該模型旨在用作浪漫小說寫作工具或用於構建遊戲，則 BookCorpus 中的小說可能是可以接受的。

讓我們通過比較 GPT 和 GPT-2 的文本生成來說明模型被數據扭曲的概念。GPT 主要在 BookCorpus 上訓練的，而 GPT-2 是在網頁、博客和從 Reddit 鏈接的新聞文章上訓練的。我們將在同一提示下比較兩個模型的相似大小版本，因此主要區別在於預訓練數據集，我們將使用文本生成管道來研究模型輸出：

從變壓器導入管道，set_seed

```
generation_gpt = pipeline('text-generation', model='openai-gpt') generation_gpt2 = pipeline('text-generation', model='gpt2')
```

接下來，讓我們創建一個簡單的函數來計算每個模型中的參數數量：

```
def model_size(model): return sum(t.numel() for t in model.parameters())

print(f'GPT尺寸 :{model_size(generation_gpt.model)/1000**2:.1f}M 個參數 ') print(f'GPT2尺寸 :{model_size(generation_gpt2.model)/1000**2:.1f} M參數 ')

GPT 大小 :116.5M 參數 GPT2 大小 :124.4M
```

原始 GPT 模型與最小的 GPT-2 模型大小差不多。現在我們可以從每個模型生成三個不同的完成，每個都有相同的輸入提示：

```
def enum_pipeline_ouputs(管道, 提示, num_return_sequences):
    out = pipe(prompt, num_return_sequences=num_return_sequences,
               clean_up_tokenization_spaces=True) 返回 \n .join(f' {i+1}. +
    s[ generated_text ] for i, s in enumerate(out))

prompt = '\n當他們回來時 print(' GPT
completions:\n + enum_pipeline_ouputs(generation_gpt, prompt, 3)) print( ) print(' GPT-2 completions:
\n + enum_pipeline_ouputs(generation_gpt2, prompt, 3))
```

GPT完成 :1。

他們回來的時候。我們需要我們
能得到的一切，傑森說，一旦他們在沒有人阻止的情況下安頓在卡車後部。”離開這裏後，要找什麼就看我們了。現在

2.

他們回來的時候。他的目光掃過
她的身體。他也給她穿上了她在旅途中穿的借來的衣服。我以為把你留在那裡會更容易。“

像這樣的女人

3.

當他們回到屋子時，她正和小男孩坐在一起。

“別害怕，”他告訴她。她緩緩點頭，睜大眼睛。她是如此迷失在她發現的一切中湯姆知道她的錯誤

GPT-2 完成 :1。

當他們回來時，我們吃了一頓豐盛的晚餐，其他人去看看他們對她的看法。我做了一個小時，他們對此很滿意。

2.

當他們回到這個島上時，又發生了一場屠殺，但他不禁為這個無助的犧牲者而被遺棄而死，而他們那天卻失敗了。所以真的非常非常感激。 3.

早上他們回到我們家時，我問她是否確定。她說：“沒有。”那天早上兩個孩子都走了。我以為他們又回到了好朋友的狀態。

什麼時候結束

通過對兩個模型的少量輸出進行採樣，我們已經可以看到 GPT 生成中明顯的“浪漫”偏差，這通常會想像一個女人和一個男人之間的浪漫互動對話。另一方面，GPT-2 是在與 Reddit 文章鏈接和來自 Reddit 文章的網絡文本上進行訓練的，並且在其生成中大多采用中性的“他們”，其中包含“類似博客”或與冒險相關的元素。

一般來說，任何在數據集上訓練的模型都會反映語言偏見以及訓練數據中人口和事件的過度或不足。對於與模型交互的目標受眾，模型行為中的這些偏差很重要；如需一些有用的指南，我們建議您參閱 Google 的一篇論文，該論文提供了數據集開發框架。 3

這個簡短的介紹應該讓您了解在創建大型文本語料庫時面臨的困難挑戰。考慮到這些，現在讓我們來看看如何創建我們自己的數據集！

構建自定義代碼數據集為了稍微簡化任務，

我們將專注於僅為 Python 編程語言構建代碼生成模型。 4我們首先需要的是一個由 Python 源代碼組成的大型預訓練語料庫。幸運的是，有一個每個軟件工程師都知道的天然資源：GitHub！著名的代碼共享網站擁有數 TB 的代碼存儲庫，這些代碼存儲庫可公開訪問，並且可以根據各自的許可證下載和使用。在撰寫本書時，GitHub

3 B. Hutchinson 等人，“[機器學習數據集的責任：來自軟件工程和基礎設施的實踐](#)”，(2020)。

4相比之下，GitHub Copilot 支持十幾種編程語言。

擁有超過 2000 萬個代碼存儲庫。其中許多是用戶為學習、未來的副項目或測試目的而創建的小型或測試存儲庫。

可以通過兩種主要方式訪問 GitHub 存儲庫：

- 通過GitHub REST API，就像我們在第 9 章下載所有的 Transformers 存儲庫的 GitHub 問題 · 通過Google BigQuery等公共數據集清單

由於 REST API 有速率限制，並且我們需要大量數據用於預訓練語料庫，因此我們將使用 Google BigQuery 提取所有 Python 存儲庫。bigquery-public-data.github_repos.contents 表包含所有小於 10 MB 的 ASCII 文件的副本。根據 GitHub 的許可 API 的規定，項目還需要是開源的才能被包含在內。



Google BigQuery 數據集不包含星級或下游使用信息。對於這些屬性，我們可以使用 GitHub REST API 或類似 Libraries.io 的服務監控開源包。事實上，GitHub 的一個團隊最近發布了一個名為 CodeSearchNet 的數據集，使用來自 Libraries.io 的信息過濾至少一項下游任務中使用的存儲庫。

讓我們看看使用 Google BigQuery 創建我們的代碼數據集需要什麼。

使用 Google BigQuery 創建數據集我們

將從 Google BigQuery 上的快照中提取 GitHub 公共存儲庫中的所有 Python 文件開始。為了可重現性，以防將來免費使用 BigQuery 的政策發生變化，我們還將在 Hugging Face Hub 上共享此數據集。導出這些文件的步驟改編自轉碼器實現並如下⁵：

1. 創建一個 Google Cloud 帳戶（免費試用就足夠了）。
2. 在您的帳戶下創建一個 Google BigQuery 項目。
3. 在這個項目中，創建一個數據集。
4. 在此數據集中，創建一個用於存儲 SQL 請求結果的表。
5. 在github_repos上準備並運行以下 SQL 查詢（要保存查詢結果，選擇 More > Query Options，選中“Set a destination table for query results”框，並指定表名）：

⁵ M.-A. Lachaux 等人，“編程語言的無監督翻譯”，(2020)。

[選擇](#)

```
f.repo_name, f.path, c.copies, c.size, c.content, l.license FROM `bigquery-public-data.github_repos.files` AS f JOIN
```

```
`bigquery-public-data.github_repos.contents` AS c ON f.id = c.id
```

[JOIN](#)

```
`bigquery-public-data.github_repos.licenses` AS l ON f.repo_name = l.repo_name WHERE NOT c.binary AND ((f.path LIKE '%.py'))
```

AND (尺寸在1024和1048575之間))

此命令處理大約 2.6 TB 的數據以提取 2680 萬個文件。結果是一個包含大約 50 GB 壓縮 JSON 文件的數據集，每個文件都包含 Python 文件的源代碼。我們過濾以刪除空文件和小文件，例如 `__init__.py` 不包含太多有用信息的文件。我們還過濾掉了大於 1 MB 的文件，並下載了所有文件的許可證，以便以後可以根據許可證過濾訓練數據。

接下來，我們將結果下載到我們的本地機器。如果您在家中嘗試此操作，請確保您有良好的可用帶寬和至少 50 GB 的可用磁盤空間。將結果表發送到本地計算機的最簡單方法是遵循以下兩步過程：

1. 將結果導出到 Google Cloud：

A. 在 Google Cloud Storage (GCS) 中創建存儲桶和文件夾。b. 通過選擇

Export > Export to GCS 將您的表導出到這個桶，帶有
JSON 和 gzip 壓縮的導出格式。

2. 要將存儲桶下載到您的計算機，請使用 gsutil 庫：

A. 使用 `pip install gsutil` 安裝 gsutil。

b. 使用您的 Google 帳戶配置 gsutil：`gsutil config`。C. 在您的機器上複製您的存儲桶：

```
$ gsutil -m -o
"GSUtil:parallel_process_count=1" cp -r gs://<name_of_bucket>
```

或者，您可以使用以下命令直接從 Hugging Face Hub 下載數據集：

```
$ git clone https://huggingface.co/datasets/transformersbook/codeparrot
```

是否過濾噪音？

任何人都可以創建 GitHub 存儲庫，因此項目的質量各不相同。關於我們希望系統在現實環境中如何執行，需要做出一些有意識的選擇。在訓練數據集中加入一些噪音將使我們的系統在推理時對噪音輸入更加穩健，但也會使其預測更加隨機。根據預期用途和整個系統集成，您可以選擇更多或更少的噪音數據並添加前後過濾操作。

出於本章的教育目的和保持數據準備代碼簡潔，我們不會根據星級或用途進行過濾，而只會抓取 GitHub BigQuery 數據集中的所有 Python 文件。然而，數據準備是至關重要的一步，您應該確保盡可能多地清理數據集。在我們的案例中，需要考慮的幾件事是是否要平衡數據集中的編程語言；過濾低質量數據（例如，通過 GitHub stars 或來自其他 repos 的引用）；刪除重複的代碼示例；考慮版權信息；調查文檔、註釋或文檔字符串中使用的語言；並刪除個人識別信息，例如密碼或密鑰。

使用 50 GB 的數據集可能具有挑戰性；它需要足夠的磁盤空間，並且必須小心不要用完 RAM。在下一節中，我們將了解數據集如何幫助處理在小型機器上處理大型數據集的這些限制。

使用大型數據集加載非常大的數據

集通常是一項具有挑戰性的任務，尤其是當數據大於機器的 RAM 時。對於大規模的預訓練數據集，這是一種非常常見的情況。在我們的示例中，我們有 50 GB 的壓縮數據和大約 200 GB 的未壓縮數據，這些數據很難提取並加載到標準尺寸的筆記本電腦或台式機的 RAM 內存中。

值得慶幸的是，Datasets 的設計是為了克服這個問題，它有兩個特定的功能，可以讓您擺脫 RAM 和硬盤空間的限制：內存映射和流。

內存映射為了

克服 RAM 限制，Datasets 使用一種默認激活的零複製和零開銷內存映射機制。基本上，每個數據集都緩存在驅動器上的一個文件中，該文件直接反映了 RAM 內存中的內容。

Datasets 不是在 RAM 中加載數據集，而是打開一個只讀指針指向這個

文件並將其用作 RAM 的替代品，基本上是將硬盤驅動器用作 RAM 內存的直接擴展。

到目前為止，我們主要使用數據集來訪問 Hugging Face Hub 上的遠程數據集。在這裡，我們將直接加載我們在本地存儲在 codeparrot 存儲庫中的 50 GB 壓縮 JSON 文件。由於 JSON 文件是壓縮的，我們首先需要解壓它們，Datasets 會幫我們解壓。小心，因為這需要大約 180 GB 的可用磁盤空間！但是，它幾乎不使用 RAM。通過在數據集的下載配置中設置 `delete_extracted=True`，我們可以確保盡快刪除所有不再需要的文件：

從數據集導入 `load_dataset`，`DownloadConfig`

```
download_config = DownloadConfig(delete_extracted=True)數據集=
load_dataset( ./codeparrot ,split= train ,
download_config=download_config)
```

在底層，Datasets 通過將所有壓縮的 JSON 文件加載到一個優化的緩存文件中來提取和讀取它們。讓我們看看這個數據集一旦加載有多大：

導入 `psutil`

```
print(f Number of python files code in dataset :{len(dataset)} ) ds_size =
sum(os.stat([ filename ])st_size for f in dataset.cache_files) # os.stat.st_size以字節表示，所以我們轉換成GB print(f Dataset size (cache file) :{ds_size / 2**30:.2f} GB )

# Process.memory_info 以字節表示，所以我們轉換為 MB print(f RAM used:
{psutil.Process(os.getpid()).memory_info().rss >> 20} MB )
```

數據集中的 python 文件代碼數：18695559 數據集大小（緩存文件）：
183.68 GB 使用的 RAM 內存：4924 MB

正如我們所見，數據集比我們典型的 RAM 內存大得多，但我們仍然可以加載和訪問它，而且我們實際上使用的內存量非常有限。

您可能想知道這是否會使我們的訓練受 I/O 限制。在實踐中，與模型處理計算相比，NLP 數據的加載量通常非常輕，因此這很少成為問題。此外，零複製/零開銷格式在底層使用 Apache Arrow，這使得訪問任何元素都非常高效。

根據硬盤驅動器的速度和批處理大小，對數據集的迭代通常可以以十分之幾 GB/s 到幾 GB/s 的速率完成。這很好，但是如果不能釋放足夠的磁盤空間來在本地存儲完整的數據集怎麼辦？

每個人都知道當您收到磁盤已滿警告並且需要通過查找要刪除的隱藏文件來痛苦地嘗試回收幾 GB 時的無助感。幸運的是，如果您使用的流功能，則不需要在本地存儲完整的數據集。

 數據集！

流式傳輸

一些數據集（達到 1 TB 或更多）即使在標準硬盤上也很難容納。在這種情況下，擴展您正在使用的服務器的另一種方法是流式傳輸數據集。對於許多可以逐行讀取的壓縮或未壓縮文件格式的數據集，這也是可能的，例如 JSON 行、CSV 或文本（原始或 zip、gzip 或 zstandard 壓縮）。讓我們直接從壓縮的 JSON 文件加載我們的數據集，而不是從它們創建緩存文件：

```
streamed_dataset = load_dataset('codeparrot', split='train', streaming=True)
```

正如您將看到的，加載數據集是即時的！在流模式下，壓縮的 JSON 文件將被打開並即時讀取。我們的數據集現在是一個 Iterable Dataset 對象。這意味著我們不能訪問它的隨機元素，比如 `streamed_dataset[1264]`，但我們需要按順序讀取它，例如使用 `next(iter(streamed_dataset))`。仍然可以使用 `shuffle()` 之類的方法，但這些方法將通過獲取示例緩衝區並在此緩衝區內洗牌（緩衝區的大小可調）來運行。當多個文件作為原始文件提供時（比如我們這裡的 184 個文件），`shuffle()` 也會隨機化迭代文件的順序。

流式數據集的樣本與非流式數據集的樣本相同，正如我們所見：

```
迭代器 = iter(流式數據集)
```

```
打印(數據集[0] == 下一個(迭代器)) 打印(數據集[1] == 下  
一個(迭代器))
```

```
真的  
真的
```

使用流式數據集的主要好處是加載此數據集不會在驅動器上創建緩存文件或需要任何（重要的）RAM 內存。當請求一批新的例子時，原始原始文件被提取並即時讀取，並且只有那批被加載到內存中。這將我們數據集的內存佔用從 180 GB 減少到 50 GB。但我們可以更進一步：我們可以引用集線器上的數據集，而不是指向本地數據集，然後直接下載樣本，而無需在本地下載原始文件：

```
remote_dataset = load_dataset('transformersbook/codeparrot', split='train', streaming=True)
```

這個數據集的行為與前一個完全一樣，但在幕後動態下載示例。有了這樣的設置，我們就可以在（幾乎）任意小的服務器上使用任意大的數據集。讓我們將帶有訓練和驗證拆分的數據集推送到 Hugging Face Hub 並通過流訪問它。

將數據集添加到 Hugging Face Hub

將我們的數據集推送到 Hugging Face Hub 將使我們能夠：

- 從我們的培訓服務器輕鬆訪問它。 · 查看
- 流式數據集如何與來自中心的數據集無縫協作。 · 與社區分享，包括您，親愛的讀者！

要上傳數據集，我們首先需要通過在終端中運行以下命令並提供相關憑據來登錄我們的 Hugging Face 帳戶：

```
$ huggingface-cli 登錄
```

這相當於我們在前面章節中使用的notebook_login()輔助函數。完成後，我們可以直接在 Hub 上創建一個新數據集並上傳壓縮的 JSON 文件。為了簡化事情，我們將創建兩個存儲庫：一個用於訓練拆分，一個用於驗證拆分。我們可以通過運行CLI 的repo create命令來完成此操作，如下所示：

```
$ huggingface-cli repo create --type dataset --organization transformersbook\ codeparrot-train $ huggingface-cli repo create --type dataset --organization transformersbook\ codeparrot-有效
```

在這裡，我們指定存儲庫應該是一個數據集（與用於存儲權重的模型存儲庫相反），以及我們希望將存儲庫存儲在其下的組織。如果您在個人帳戶下運行此代碼，則可以省略--organization標誌。接下來，我們需要將這些空存儲庫克隆到我們的本地機器，將 JSON 文件複製到它們，並將更改推送到 Hub。我們將從我們擁有的 184 個 JSON 文件中取出最後一個壓縮的 JSON 文件作為驗證文件（即，我們數據集的大約 0.5%）。執行這些命令將存儲庫從集線器克隆到本地計算機：

```
$ git 克隆 https://huggingface.co/datasets/transformersbook/codeparrot-train $ git 克隆 https://huggingface.co/datasets/transformersbook/codeparrot-valid
```

接下來，複製除最後一個 GitHub 文件之外的所有文件作為訓練集：

```
$ cd codeparrot-train $ cp ../
codeparrot*.json.gz $ rm ./
file-000000000183.json.gz
```

然後提交文件並將它們推送到 Hub：

```
$ 混帳添加。$ git commit -m 添加數據集文件 $ git push
```

現在，對驗證集重複該過程：

```
$ cd ../codeparrot-valid $ cp ../
codeparrot/file-000000000183.json.gz 。 $ mv ./
file-000000000183.json.gz ./file-000000000183_validation.json.gz $ git 添加。$ git commit -m "添
加數據集文件" $ git push
```

混帳添加。由於計算了所有文件的哈希值，因此這一步可能需要幾分鐘。上傳所有文件也需要一些時間。由於這將使我們能夠在本章後面使用流，但是，這並沒有浪費時間，而且這一步將使我們在其餘實驗中進行得更快。請注意，我們在驗證文件名中添加了一個_validation後綴。這將使我們能夠稍後將其作為驗證拆分加載。

就是這樣！我們的數據集的兩個拆分以及完整的數據集現在位於以下 URL 的 Hugging Face Hub 上：

- <https://huggingface.co/datasets/transformersbook/codeparrot>
- <https://huggingface.co/datasets/transformersbook/codeparrot-train>
- <https://huggingface.co/datasets/transformersbook/codeparrot-valid>



最好添加 README 卡片來解釋數據集是如何創建的，並提供盡可能多的有用信息。一個記錄良好的數據集更有可能對其他人有用，對你未來的自己也是如此。您可以閱讀[數據集自述文件指南](#)有關如何編寫好的數據集文檔的詳細說明。您也可以稍後使用💡 Web 編輯器直接在 Hub 上修改您的 README 卡。

構建分詞器

現在我們已經收集並加載了我們的大型數據集，讓我們看看如何有效地處理數據以提供給我們的模型。在前面的章節中，我們使用了伴隨我們使用的模型的分詞器。這是有道理的，因為這些模型是使用通過分詞器中定義的特定預處理管道傳遞的數據進行預訓練的。使用預訓練模型時，務必堅持為預訓練選擇的相同預處理設計選擇。否則，模型可能會輸入分佈模式或未知標記。

然而，當我們訓練一個新模型時，使用為另一個數據集準備的分詞器可能不是最優的。以下是我們在使用現有分詞器時可能遇到的問題類型的一些示例：

- T5 分詞器在C4上訓練我們之前遇到的語料庫，但是使用了廣泛的停用詞過濾步驟來創建它。因此，T5 分詞器從未見過諸如 “sex”之類的常見英文單詞。
- CamemBERT 標記器也在非常大的文本語料庫上進行了訓練，但只包含法語文本（OSCAR 的法語子集語料庫）。因此，它不知道諸如 “being”之類的常見英語單詞。

我們可以在實踐中輕鬆測試每個分詞器的這些功能：

從變形金剛導入AutoTokenizer

```
def tok_list(分詞器,字符串) :
    input_ids = tokenizer(string, add_special_tokens=False)[ input_ids ] 返回
    [tokenizer.decode(tok) for tok in input_ids]

tokenizer_T5 = AutoTokenizer.from_pretrained( t5-base )
tokenizer_camembert = AutoTokenizer.from_pretrained( camembert-base )

打印 ( “性別”的 f T5 標記 :{tok_list(tokenizer_T5, “性別”)} ) 打印
(f CamemBERT “存在”的標記 :{tok_list(tokenizer_camembert, “存在”)} )

“性別”的 T5 標記 :[ , s , ex ]
“存在”的 CamemBERT 標記 :[ be , ing ]
```

在許多情況下，將如此短且常用的單詞拆分為子部分效率不高，因為這會增加模型的輸入序列長度（上下文有限）。因此，了解用於訓練分詞器的數據集的域和預處理非常重要。分詞器和模型可以對數據集中的偏差進行編碼，這些偏差會影響模型的下游行為。

因此，要為我們的數據集創建最佳分詞器，我們需要自己訓練一個。
讓我們看看如何做到這一點。



訓練模型涉及從一組給定的權重開始，並使用來自設計目標上的誤差信號的反向傳播來最小化模型的損失，並為模型找到一組最佳權重以執行訓練目標定義的任務。另一方面，訓練分詞器不涉及反向傳播或權重。這是一種創建從文本字符串到模型可以攝取的整數列表的最佳映射的方法。在今天的分詞器中，最佳的字符串到整數轉換涉及一個由原子字符串列表和關聯方法組成的詞彙表，該方法用於將文本字符串轉換、規範化、剪切或映射到具有該詞彙表的索引列表。這個索引列表就是我們神經網絡的輸入。

分詞器模型

正如您在第 4 章中看到的，分詞器是一個由四個步驟組成的處理管道：規範化、預分詞器、分詞器模型和後處理。分詞器管道中可以根據數據進行訓練的部分是分詞器模型。正如我們在第 2 章中討論的那樣，可以使用多種子詞標記化算法，例如 BPE、WordPiece 和 Unigram。

BPE 從基本單位列表（單個字符）開始，通過逐步創建新標記的過程創建詞彙表，新標記通過合併最頻繁同時出現的基本單位並將它們添加到詞彙表中。重複此過程，直到達到預定義的詞彙表大小。

Unigram 從另一端開始，通過使用語料庫中的所有詞和潛在的子詞初始化其基本詞彙表。然後它逐漸刪除或拆分不太有用的標記以獲得越來越小的詞彙表，直到達到目標詞彙表大小。WordPiece 是 Unigram 的前身，其官方實現從未被谷歌開源。

這些不同算法對下游性能的影響因任務而異，總體而言，很難確定一種算法是否明顯優於其他算法。BPE 和 Unigram 在大多數情況下都具有合理的性能，但讓我們來看看評估時需要考慮的一些方面。

測量分詞器的性能
分詞器的最優性和
性能在實踐中很難衡量。一些可能的指標包括：

- Subword fertility，它計算每個產生的子詞的平均數量
分詞詞
- 連續詞的比例，指的是語料庫中被分詞成至少兩個子分詞的詞的比例
- 覆蓋率指標，例如標記化語料庫中未知詞或很少使用的標記的比例

此外，通常會估計對拼寫錯誤或噪聲的魯棒性，以及此類域外示例的模型性能，因為這在很大程度上取決於標記化過程。

這些度量給出了一組關於分詞器性能的不同觀點，但它們往往忽略了分詞器與模型的交互。例如，可以通過在詞彙表中包含所有可能的詞來最小化子詞的生育率，但這將為模型產生非常大的詞彙表。

最後，通常通過使用模型的下游性能作為最終指標來最好地估計各種標記化方法的性能。例如，早期 BPE 方法的良好性能通過使用這些標記器和詞彙而不是基於字符或單詞的標記化訓練的模型顯示機器翻譯任務的改進性能得到證明。

讓我們看看如何構建我們自己的針對 Python 代碼優化的分詞器。

Python 的分詞器我們需

要一個自定義分詞器來處理我們的用例：分詞 Python 代碼。pretokenization 的問題值得對編程語言進行一些討論。如果我們拆分空格並刪除它們，我們將丟失所有縮進信息，這在 Python 中對於程序的語義很重要（想想while循環，或if-then-else語句）。另一方面，換行符沒有意義，可以添加或刪除而不影響語義。類似地，拆分標點符號，如下劃線，用於從多個子部分組成單個變量名稱，可能不如在自然語言中那麼有意義。

因此，使用自然語言預分詞器對代碼進行分詞似乎可能不是最優的。

讓我們看看 Hub 上提供的集合中是否有任何可能對我們有用的分詞器。我們想要一個保留空格的分詞器，因此一個好的候選者可能是像 GPT-2 中的那樣的字節級分詞器。讓我們加載這個分詞器並探索它的分詞屬性：

[從變形金剛導入AutoTokenizer](#)

```
python_code = r'''def say_hello():
    print(Hello, World!)'''

# 打印它
say_hello()

tokenizer = AutoTokenizer.from_pretrained('gpt2')打印
(tokenizer(python_code).tokens())

[ def , say , _ , hello , (): , , , , , print , ( ,
Hello , , , World , ! , ) , # , Print , it , , , say , _ ,
hello , () , C ]
```



Python 有一個內置的 tokenize 模塊，可以將 Python 代碼字符串拆分成有意義的單元（代碼操作、註釋、縮進和縮進等）。使用這種方法的一個問題是這種預分詞器是基於 Python 的，因此通常速度很慢並且受到 Python 全局解釋器鎖 (GIL) 的限制。另一方面，Transformers 庫中的大多數分詞器都是由 Tokenizers 庫提供的，並且是用 Rust 編碼的。



Rust 分詞器的訓練和使用速度要快很多個數量級，因此我們可能希望在我們語料庫的規模下使用它們。



這是一個非常奇怪的輸出，所以讓我們嘗試通過運行分詞器管道的各個子模塊來了解這裡發生了什麼。首先讓我們看看這個分詞器應用了什麼規範化：

[打印 \(tokenizer.backend_tokenizer.normalizer\)](#)

沒有任何

正如我們所見，GPT-2 分詞器不使用規範化。它直接在原始 Unicode 輸入上工作，無需任何規範化步驟。現在讓我們看一下預標記化：

[打印 \(tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str \(python_code\)\)](#)

```
[def (0, 3), say (3, 7), _, (7, 8),你好 (8, 13), (): (13, 16), (16, 20), print (20, 26), (_, (26, 28)), (你好 (28, 33), (, (33, 34)), (World (34, 40), (!) (40, 43)), (# (43, 45)), (Print (45, 51)), (it (51, 54)), (, (54, 55)), (, (55, 56)), (說 (56, 59)), (_ (59, 60)), (你好 (60, 65)), ( (65, 67)), (, (67, 68))]
```

所有這些符號是什麼，標記旁邊的數字是什麼？

讓我們解釋一下，看看我們是否能更好地理解這個分詞器是如何工作的。

讓我們從數字開始。Tokenizers 有一個非常有用的功能，用於在字符串和標記之間切換，稱為偏移跟蹤。輸入字符串上的所有操作都會被跟蹤，這樣就可以準確地知道標記化後的標記對應於輸入字符串的哪一部分。這些數字只是表示每個標記在原始字符串中的來源；例如，第一行中的單詞“hello”對應於原始字符串中的字符 8 到 13。如果在規範化步驟中刪除了一些字符，我們仍然能夠將每個標記與原始字符串中的相應部分相關聯。

標記化文本的另一個奇怪特徵是外觀古怪的字符，例如和。字節級意味著這個分詞器處理字節而不是 Unicode 字符。每個 Unicode 字符由 1 到 4 個字節組成，具體取決於字符。字節的好處在於，雖然 Unicode 字母表中有 143,859 個 Unicode 字符，但字節字母表中只有 256 個元素，

您可以將每個 Unicode 字符表示為這些字節的序列。如果我們處理字節，那麼我們可以將所有由 UTF-8 世界組成的字符串表示為這個包含 256 個值的字母表中的較長字符串。也就是說，我們可以擁有一個僅使用 256 個單詞的字母表的模型，並且能夠處理任何 Unicode 字符串。讓我們看看一些字符的字節表示是什麼樣子的：

```
a, e = u  a , u  €
byte = ord(a.encode(  utf-8 ))
print(f  `{a}`編碼為` {a.encode(  utf-8 )}` 帶有一個字節 :{byte}  ) byte = [ord(chr(i)) for i in
e.encode(  utf-8 )] print(f  `{e}`被編碼為` {e.encode(  utf-8 )}` 三個字節 :{byte}  )
```

`a` 被編碼為具有單個字節的 `b` a :97 `€` 被編碼為具有三個字節
的 `b` \xe2\x82\xac :[226, 130, 172]

此時你可能想知道：為什麼要在字節級別上工作？回想一下我們在第 2 章中關於字符和單詞標記之間權衡的討論。我們可以決定從 143,859 個 Unicode 字符構建我們的詞彙表，但我們也想在我們的詞彙表中包含單詞，即 Unicode 字符的組合，所以這個（已經非常大的）大小只是總詞彙量。這將使我們模型的嵌入層非常大，因為它為每個詞彙標記包含一個向量。

在另一個極端，如果我們只使用 256 字節的值作為我們的詞彙表，輸入序列將被分割成許多小塊（每個字節構成 Unicode 字符），因此我們的模型將不得不處理長輸入並花費大量計算能力從單獨的字節重建 Unicode 字符，然後從這些字符重建單詞。有關此開銷的詳細研究，請參閱 ByT5 模型發布隨附的論文。⁶ 中間解決方案是通過使用最常見的字節組合擴展 256 個單詞的詞彙表來構建中等規模的詞彙表。這是 BPE 算法採用的方法。這個想法是通過迭代合併詞彙表中最常同時出現的標記對來創建新的詞彙標記，從而逐步構建一個預定義大小的詞彙表。例如，如果 t 和 h 非常頻繁地一起出現，就像在英語中一樣，我們將在詞彙表中添加一個標記 th 來對這對標記進行建模，而不是將它們分開。t 和 h 標記保留在詞彙表中，以標記它們不一起出現的實例。從基本單元的基本詞彙表開始，我們可以有效地對任何字符串進行建模。

⁶ L. Xue 等人，“ByT5：使用預訓練的字節到字節模型邁向無令牌的未來”，(2021)。



注意不要將“Byte-Pair Encoding”中的“byte”與“byte-level”中的“byte”混淆。字節對編碼這個名稱來自 Philip Gage 於 1994 年提出的一種數據壓縮技術，最初是對字節進行操作。⁷ 與這個名稱可能表明的不同，NLP 中的標準 BPE 算法通常對 Unicode 字符串而不是字節進行操作（儘管有是一種專門針對字節工作的新型 BPE，稱為字節級 BPE）。如果我們以字節為單位讀取我們的 Unicode 字符串，我們就可以重用一個簡單的 BPE 子詞拆分算法。

在 NLP 中使用典型的 BPE 算法時只有一個問題。這些算法旨在使用乾淨的 Unicode 字符串作為輸入，而不是字節，並期望輸入中的常規 ASCII 字符，沒有空格或控製字符。但是在前 256 個字節對應的 Unicode 字符中，有很多控製字符（換行符、製表符、轉義符、換行符和其他不可打印的字符）。為了克服這個問題，GPT-2 標記器首先將所有 256 個輸入字節映射到標準 BPE 算法可以輕鬆消化的 Unicode 字符串。也就是說，我們將 256 個基本值映射到 Unicode 字符串，這些字符串都對應於標準可打印 Unicode 字符。

這些 Unicode 字符每個都用 1 個字節或更多字節編碼並不是很重要；重要的是我們最後有 256 個單一值，形成我們的基本詞彙表，並且這 256 個值由我們的 BPE 算法正確處理。

讓我們看一下使用 GPT-2 分詞器進行這種映射的一些示例。我們可以訪問整個映射如下：

從`transformers.models.gpt2.tokenization_gpt2`導入`bytes_to_unicode`

```
byte_to_unicode_map = bytes_to_unicode()
unicode_to_byte_map = dict((v, k) for k, v in byte_to_unicode_map.items())
base_vocab = list(unicode_to_byte_map.keys())

print(f'我們的基本詞彙表的大小 :{len(base_vocab)}')
print(f'第一個元素 :`{base_vocab[0]}`')
print(f'最後一個元素 :`{base_vocab[-1]}`')
```

我們的基本詞彙表的大小 :256 第一個元素 :`!`，
最後一個元素 :`ÿ`

我們可以在表 10-1 中查看字節的一些常見值和關聯的映射 Unicode 字符。

⁷ P. Gage，“一種新的數據壓縮算法”，C 用戶雜誌第 12 期，第 1 期。2 (1994): 23–38, <https://dx.doi.org/10.14569/IJACSA.2012.030803>。

表 10-1。BPE 中的字符映射示例

描述	字符字節		映射字節
常規字符	`a` 和 `?`	97 和 63	`a` 和 `?`
不可打印的控制字符 (回車)`U+000D` 13			``
空格		32	` `
牢不可破的空格	`\xa0`	160	` `
一個換行符	`\n`	10	` `

我們本可以使用更明確的轉換，例如將換行符映射到 NEWLINE 字符串，但 BPE 算法通常設計用於處理字符。出於這個原因，為每個字節字符保留一個 Unicode 字符更容易使用開箱即用的 BPE 算法來處理。現在我們已經了解了 Unicode 編碼的黑魔法，我們可以更好地理解我們的標記化轉換：

```
打印 (tokenizer.backend_tokenizer.pre_tokenizer.pre_tokenize_str (python_code) )

[([ def , (0, 3)), ( say , (3, 7)), ( _ , (7, 8)), ( 你好 , (8, 13)), ( ():, ,(13, 16)), ( , ,(16, 20)), ( print , (20, 26)), ( ( , (26, 28)), ( 你好 , (28, 33)), ( , ,(33, 34)), ( World , (34, 40)), ( ! ) , (40, 43)), ( # , (43, 45)), ( Print , (45, 51)), ( it , (51, 54)), ( , ,(54, 55)), ( , ,(55, 56)), ( 說 , (56, 59)), ( _ , (59, 60)), ( 你好 , (60, 65)), ( () , (65, 67)), ( , ,(67, 68))]
```

我們可以識別換行符，我們現在知道換行符映射到 `，空格映射到 `。我們還看到：

- 保留空格，特別是連續的空格（例如，`中的三個空格`）。 · 連續的空格被視為一個單詞。
- 單詞前的每個空格都附加到並被視為子詞的一部分。

quent 詞（例如，在 say 中）。

現在讓我們試驗一下 BPE 模型。正如我們所提到的，它負責將單詞拆分為子單元，直到所有子單元都屬於預定義的詞彙表。

我們的 GPT-2 分詞器的詞彙表包含 50,257 個單詞：

- 具有 256 個字節值的基本詞彙表 · 通過反複合並最常見的同時出現的標記創建了 50,000 個額外的標記
- 添加到詞彙表中以表示文檔邊界的特殊字符

我們可以通過查看 tokenizer 的 length 屬性輕鬆檢查：

```
print(f 詞彙表的大小 :{len(tokenizer) })
```

詞彙量 :50257

在我們的輸入代碼上運行完整的管道會得到以下輸出：

打印 (分詞器 (python_code) .tokens ())

```
[ def , say , _ , hello , () , , , , , print , ( ,  
Hello , , World , ! , ) , # , Print , it , , , say , _ ,  
hello , () , C ]
```

正如我們所見，BPE 分詞器保留了大部分單詞，但會將縮進的多個空格拆分為幾個連續的空格。發生這種情況是因為此分詞器並未專門針對代碼進行訓練，而是主要針對連續空格很少見的文本進行訓練。因此，BPE 模型不在詞彙表中包含用於縮進的特定標記。在這種情況下，分詞器模型不太適合數據集的域。正如我們之前討論的，解決方案是在目標語料庫上重新訓練分詞器。讓我們開始吧！

訓練分詞器讓我們在語料庫的

一部分上重新訓練字節級 BPE 分詞器，以獲得更好地適應 Python 代碼的詞彙表。重新訓練 Transformers 提供的分詞器很簡單。我們只需要：

- 指定我們的目標詞彙量。 · 準備一個迭代器來提供要處理的輸入字符串列表，以訓練分詞器。
er的模型。
 - 調用train_new_from_iterator()方法。

與通常期望從訓練語料庫中記住大量特定細節的深度學習模型不同，分詞器實際上只是接受了提取主要統計數據的訓練。簡而言之，分詞器只是經過訓練，知道哪些字母組合在我們的語料庫中出現頻率最高。

因此，您不一定需要在非常大的語料庫上訓練分詞器；語料庫只需要代表您的領域，並且足夠大，以便分詞器提取具有統計意義的度量。但是根據詞彙量和語料庫中的確切文本，分詞器最終可能會存儲意想不到的詞。例如，在查看 GPT-2 分詞器詞彙表中最長的單詞時，我們可以看到這一點：

這些標記看起來像可能在論壇上使用的分隔線。這是有道理的，因為 GPT-2 是在以 Reddit 為中心的語料庫上訓練的。現在讓我們看一下最後添加到詞彙表中的單詞，也就是出現頻率最低的單詞：

```
tokens = sorted(tokenizer.vocab.items(), key=lambda x: x[1], reverse=True)
print([f'{tokenizer.convert_tokens_to_string(t)} 對於, _ 在令牌[:12]]);
```

[<|endoftext> , 凝視 , 線人 , 對撞機 , 回歸 , 被淘汰 , 放大 , 比較 , ... , (/ , 委員會 , 殺手]

第一個標記`<|endoftext|>`是用於指定文本序列結尾的特殊標記，是在構建 BPE 詞彙表之後添加的。對於這些標記中的每一個，我們的模型都必須學習相關的詞嵌入，我們可能不希望嵌入矩陣包含太多嘈雜的詞。還要注意一些非常特定於時間和空間的世界知識（例如，像Hitman和Commission這樣的專有名詞）是如何在我們的建模方法中以非常低的水平嵌入的，這些詞被授予單獨的標記，並在詞彙表中關聯向量。BPE 標記器創建此類特定標記也可能表明目標詞彙量太大或語料庫包含特殊標記。

讓我們在我們的語料庫上訓練一個新的分詞器並檢查它學到的詞彙。由於我們只需要一個能合理代表我們的數據集統計數據的語料庫，讓我們從我們的語料庫中選擇大約 1-2 GB 的數據，或大約 100,000 個文檔：

從tqdm.auto導入tqdm

```
length = 100000
dataset_name = "transformersbook/codeparrot-train" dataset =
load_dataset(dataset_name, split= "train" , streaming=True) iter_dataset = iter(數據集)

def batch_iterator(batch_size=10):
    為了 _ 在 tqdm(range(0, length, batch_size)) 中 : yield
        [next(iter_dataset)[ content ] for _ 在範圍內 (batch_size) ]

new_tokenizer = tokenizer.train_new_from_iterator(batch_iterator(), vocab_size=12500,
```

讓我們研究一下 BPE 算法創建的第一個和最後一個詞，看看我們的詞彙表有多相關。我們跳過 256 字節的標記並查看其後添加的第一個標記：

```

tokens = sorted(new_tokenizer.vocab.items(), key=lambda x: x[1], reverse=False)打印
([f {tokenizer.convert_tokens_to_string(t)} for t, in tokens[257:280]]); -
[ , , , se , in , , or , , re , on , te ,
, \n st , de , \n , th , le , = , if , self ,
我 , al ]

```

在這裡我們可以看到各種標準級別的縮進和空白標記，以及簡短的常見 Python 關鍵字，如 self、or 和 in。這是一個好兆頭，表明我們的 BPE 算法正在按預期工作。現在讓我們看看最後的話：

```

print([f {new_tokenizer.convert_tokens_to_string(t)} for t, in tokens[-12:]]);

[ capt , embedded , regarding , Bundle , 355 , recv , dmp , vault ,
Mongo , 可能 , 實施 , 匹配 ]

```

這裡還是有一些比較常用的詞，比如 recv，以及可能來自評論的一些更嘈雜的詞。

我們還可以標記我們的簡單 Python 代碼示例，以查看我們的標記器在一個簡單示例上的行為方式：

```

打印 (new_tokenizer (python_code) .tokens ())
[ def , s , ay , _ , hello , (): , , print , ( , Hello , , ,
Wor , ld , ! ) , # , Print , it , , s , ay , _ , hello ,
() , C ]

```

儘管它們不是代碼關鍵字，但看到像 World 這樣的常見英語單詞或被我們的分詞器拆分還是有點煩人，因為我們希望它們在語料庫中出現得相當頻繁。讓我們檢查一下是否所有的 Python 保留關鍵字都在詞彙表中：

導入關鍵詞

```

print(f There in totally {len(keyword.kwlist)} Python keywords. ) for keyw in keyword.kwlist:
if keyw not in new_tokenizer.vocab: print(f No, keyword ` {keyw}` is not 在詞彙表中 )

```

總共有 35 個 Python 關鍵字。

不，關鍵字 `await` 不在詞彙表中 不，關鍵字 `finally` 不在詞彙表
中 不，關鍵字 `nonlocal` 不在詞彙表中

似乎幾個非常頻繁的關鍵字，比如 finally，也不在詞彙表中。讓我們嘗試使用更大的數據集樣本構建更大的詞彙表。

例如，我們可以構建一個包含 32,768 個單詞的詞彙表（對於一些高效的 GPU/TPU 計算，8 的倍數更好）並在兩倍大的語料庫上訓練分詞器：

```

length = 200000
new_tokenizer_larger = tokenizer.train_new_from_iterator(batch_iterator(),
vocab_size=32768, initial_alphabet=base_vocab)

```

我們不希望在添加更多文檔時最頻繁的標記發生太大變化，但讓我們看看最後的標記：

```
tokens = sorted(new_tokenizer_larger.vocab.items(), key=lambda x: x[1],
    反向=假)
print([f {tokenizer.convert_tokens_to_string(t)} 對於t,
        _ 在令牌[-12:]]);

[lineEdit , spik , BC , pective , OTA , theus , FLUSH , executils ,
 00000002 , DIVISION , CursorPosition , InfoBar ]
```

簡短的檢查沒有顯示這裡有任何常規的編程關鍵字，這是很有可能的。讓我們嘗試使用新的更大的分詞器來分詞我們的示例代碼示例：

```
打印 (new_tokenizer_larger (python_code) °tokens () )

[ def , say , _ , hello , (): , , print , ( , Hello , , ,
World , ! ) , # , Print , it , , say , _ , hello , () ,
]
```

這裡縮進也很方便地保留在詞彙表中，我們看到像Hello、World和say這樣的常見英語單詞也作為單個標記包含在內。這似乎更符合我們對模型在下游任務中可能看到的數據的預期。讓我們像之前一樣研究常見的 Python 關鍵字：

```
對於keyword.kwlist中的keyw：
如果keyw不在new_tokenizer_larger.vocab 中：print(f 不，
    關鍵字 `{keyw}` 不在詞彙表中 )

不，關鍵字 “nonlocal”不在詞彙表中
```

我們仍然缺少nonlocal關鍵字，但它在實踐中也很少使用，因為它使語法更加複雜。將其排除在詞彙之外似乎是合理的。

在這次手動檢查之後，我們更大的分詞器似乎很適合我們的任務。但正如我們之前提到的，客觀地評估分詞器的性能是一項具有挑戰性的任務，而不測量模型的性能。我們將繼續這個並訓練一個模型，看看它在實踐中的效果如何。



通過比較標記化代碼示例的序列長度，您可以輕鬆驗證新標記器的效率大約是標準 GPT-2 標記器的兩倍。我們的標記生成器使用大約一半的標記來編碼文本，這免費為我們提供了兩倍的有效模型上下文。

當我們在大小為 1,024 的上下文窗口上使用新分詞器訓練新模型時，它等效於在大小為 2,048 的上下文窗口中使用舊分詞器訓練同一模型，具有速度更快和內存效率更高的優勢。

在集線器上保存自定義分詞器現在我們

的分詞器已經過訓練，我們應該保存它。保存它並能夠在以後從任何地方訪問它的最簡單方法是將它推送到 Hugging Face Hub。當我們使用單獨的訓練服務器時，這將在以後特別有用。

要創建私有模型存儲庫並將我們的分詞器作為第一個文件保存在其中，我們可以直接使用分詞器的push_to_hub()方法。由於我們已經使用huggingface-cli 登錄驗證了我們的帳戶，我們可以簡單地推送標記器，如下所示：

```
model_ckpt = "codeparrot" org
= "transformersbook"
new_tokenizer_larger.push_to_hub(model_ckpt, organization=org)
```

如果您不想推送给組織，您可以簡單地省略組織參數。這將在您的名稱空間中創建一個名為codeparrot 的存儲庫，然後任何人都可以通過運行來加載它：

```
reloaded_tokenizer = AutoTokenizer.from_pretrained(org + / + model_ckpt)打印
(reloaded_tokenizer(python_code).tokens())
[ def , say , _ , hello , (): , , , print , ( , Hello , , ,
World , ! ) , # , Print , it , , , say , _ , hello , () ,
]
```

從 Hub 加載的分詞器的行為與我們剛才看到的完全一樣。我們還可以調查它的文件和Hub 上保存的詞彙表。為了可重複性，讓我們也保存較小的分詞器：

```
new_tokenizer.push_to_hub(model_ckpt+ -small-vocabulary ,organization=org)
```

這是對為特定用例構建分詞器的深入研究。接下來，我們將最終創建一個新模型並從頭開始訓練它。

從頭開始訓練模型

這是您可能一直在等待的部分：模型訓練。在本節中，我們將決定哪種架構最適合該任務，初始化一個沒有預訓練權重的新模型，設置一個自定義數據加載類，並創建一個可擴展的訓練循環。在決賽中，我們將分別訓練具有 1.11 億和 15 億參數的小型和大型 GPT-2 模型！但是，讓我們不要超越自己。

首先，我們需要確定哪種架構最適合代碼自動完成。



在本節中，我們將實施一個比平常更長的腳本來在分佈式基礎設施上訓練模型。因此，您不應單獨運行每個代碼片段，而應下載 Transformers 存儲庫中提供的腳本。按照隨附的說明在您的硬件上使用 Accelerate 執行腳本。



預訓練目標的故事現在我們可以訪問大規

模的預訓練語料庫和高效的分詞器，我們可以開始考慮如何預訓練 transformer 模型。有了由如圖 10-1 所示的代碼片段組成的如此龐大的代碼庫，我們可以處理多項任務。我們選擇哪一個會影響我們對預訓練目標的選擇。讓我們看一下三個常見任務。

Example from corpus

```
def add_numbers(a,b):
    "add two numbers"
    return a+b
```

圖 10-1。可以在我們的數據集中找到的 Python 函數示例

因果語言建模文本數據的

一項自然任務是為模型提供代碼示例的開頭，並要求它生成可能的補全。這是一個自我監督的訓練目標，我們可以在其中使用沒有註釋的數據集。這應該敲響警鐘：這是我們在第 5 章中遇到的因果語言建模任務。直接相關的下游任務是代碼自動完成，因此我們肯定會將此模型列入候選名單。僅解碼器架構（例如 GPT 模型系列）通常最適合此任務，如圖 10-2 所示。

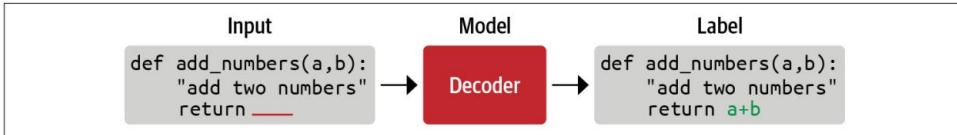


圖 10-2 在因果語言建模中，未來的標記被屏蔽，模型必須預測它們；通常使用 GPT 等解碼器模型來完成此類任務

掩碼語言建模一個相關但略

有不同的任務是為模型提供帶有噪聲的代碼樣本，例如用隨機或掩碼詞替換的代碼指令，並要求它重建原始的干淨樣本，如圖 10-3。這也是一個自我監督的訓練目標，通常稱為屏蔽語言建模或去噪目標。很難考慮與去噪直接相關的下游任務，但去噪通常是一個很好的預訓練任務，可以為後來的下游任務學習一般表示。我們在前幾章中使用的許多模型（如 BERT 和 XLM-RoBERTa）都是以這種方式進行預訓練的。因此，在大型語料庫上訓練掩碼語言模型可以與在下游任務上使用有限數量的標記示例微調模型相結合。

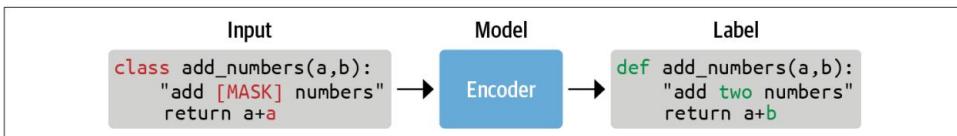


圖 10-3 在屏蔽語言建模中，一些輸入標記被屏蔽或替換，模型的任務是預測原始標記；這是 Transformer 模型編碼器分支的底層架構

Sequence-to-sequence

training 另一個任務是使用正則表達式之類的啟發式方法將註釋或文檔字符串與代碼分開，並構建可用作帶註釋數據集的（代碼、註釋）對的大規模數據集。然後，訓練任務是一個監督訓練目標，其中一個類別（代碼或評論）用作模型的輸入，另一個類別（評論或代碼）用作標籤。這是使用（輸入、標籤）對進行監督學習的情況，如圖 10-4 中突出顯示的那樣。有了一個大的、乾淨的、多樣化的數據集以及一個有足夠容量的模型，我們可以嘗試訓練一個學習在代碼中轉錄註釋的模型，反之亦然。與此監督訓練任務直接相關的下游任務是從代碼生成文檔或從文檔生成代碼，具體取決於我們如何設置輸入/輸出。在此設置中，一個序列被轉換為另一個序列，這是 T5、BART 和 PEGASUS 等編碼器-解碼器架構大放異彩的地方。

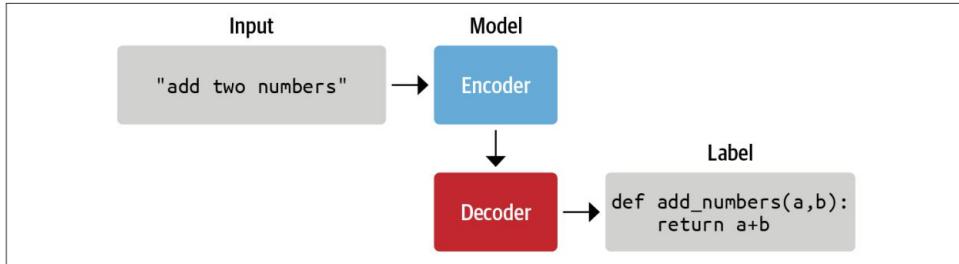


圖 10-4 將編碼器-解碼器架構用於序列到序列任務，其中使用啟發式方法將輸入拆分為註釋/代碼對。模型將一個元素作為輸入並需要生成另一個元素

由於我們要構建代碼自動完成模型，因此我們將選擇第一個目標並為任務選擇 GPT 架構。因此，讓我們初始化一個新的 GPT-2 模型！

初始化模型

這是本書中我們第一次不使用 `from_pretrained()` 方法加載模型而是初始化新模型。然而，我們將加載 `gpt2-xl` 的配置，以便我們使用相同的超參數，並且只為新的分詞器調整詞彙表大小。然後，我們使用 `from_config()` 方法使用此配置初始化一個新模型：

從變壓器導入 `AutoConfig`、`AutoModelForCausalLM`、`AutoTokenizer`

```
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
config = AutoConfig.from_pretrained("gpt2-xl", vocab_size=len(tokenizer))
model = AutoModelForCausalLM.from_config(config)
```

讓我們檢查模型實際有多大：

```
print(f" GPT-2 (xl) 尺寸 :{model_size(model)/1000**2:.1f}M 參數 ")
GPT-2 (xl) 大小 :1529.6M 參數
```

這是 1.5B 參數模型！這是一個很大的容量，但我們也有一個大數據集。一般來說，只要數據集相當大，大型語言模型的訓練效率就會更高。讓我們將新初始化的模型保存在 `models/` 文件夾中並將其推送到 Hub：

```
model.save_pretrained("models/" + model_ckpt, push_to_hub=True,
                      organization=组织)
```

考慮到檢查點的大小 (> 5 GB) ,將模型推送到 Hub 可能需要幾分鐘時間。由於此模型非常大 ,我們還將創建一個較小的版本 ,我們可以對其進行訓練以確保在擴大規模之前一切正常。我們將以標準 GPT-2 尺寸為基礎：

```
tokenizer = AutoTokenizer.from_pretrained(model_ckpt) config_small
= AutoConfig.from_pretrained( gpt2 ,vocab_size=len(tokenizer))模型_small =
AutoModelForCausalLM.from_config(config_small)

print(f GPT-2尺寸: {model_size(model_small)/1000**2:.1f}M 參數 )

GPT-2 大小 :111.0M 參數
```

讓我們將它也保存到 Hub 中 ,以便於共享和重用：

```
model_small.save_pretrained( models/ + model_ckpt + -small ,push_to_hub=True, organization=org)
```

現在我們有兩個可以訓練的模型 ,我們需要確保我們可以在訓練期間有效地為它們提供輸入數據。

實現數據加載器為了能夠以最大效

率進行訓練 ,我們需要為我們的模型提供填充其上下文的序列。例如 ,如果我們模型的上下文長度是 1,024 個標記 ,我們總是希望在訓練期間提供 1,024 個標記序列。但是我們的一些代碼示例可能比 1,024 個標記更短或更長。為了將具有 sequence_length 的完整序列的批次提供給我們的模型 ,我們應該刪除最後一個不完整的序列或填充它。然而 ,這會使我們的訓練效率稍低 ,並迫使我們處理填充和屏蔽填充的標記標籤。我們更多的是計算而不是數據約束 ,所以我們將在這裡採用簡單有效的方法。我們可以使用一個小技巧來確保我們不會丟失太多尾隨片段 :我們可以標記幾個示例 ,然後將它們連接起來 ,用特殊的序列結束標記分隔 ,以獲得一個很長的序列。最後 ,我們將這個序列分成大小相等的塊 ,如圖10-5 所示。使用這種方法 ,我們最終最多丟失一小部分數據。

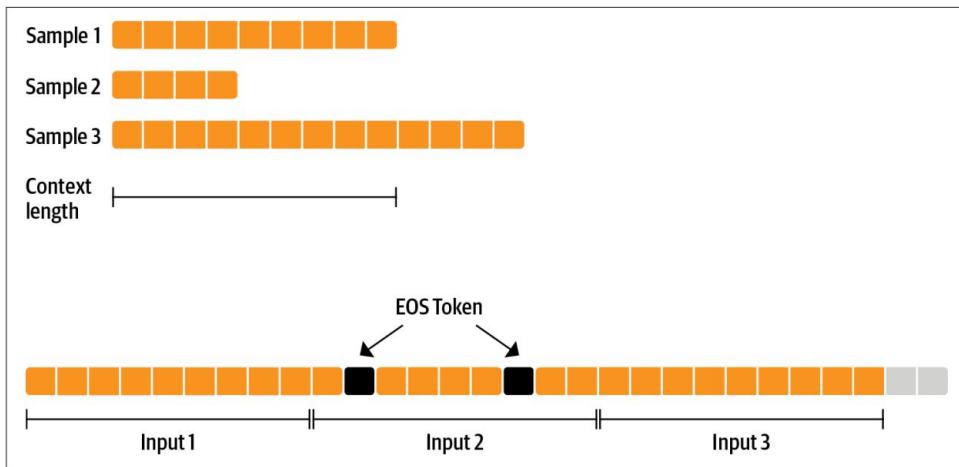


圖 10-5 通過在分塊之前將幾個標記化示例與 EOS 標記連接起來，為因果語言建模準備不同長度的序列

例如，我們可以通過將輸入字符串字符長度定義為以下方式來確保在標記化示例中有大約一百個完整序列：

```
input_characters = number_of_sequences * sequence_length * characters_per_token
```

在哪裡：

- `input_characters`是輸入到我們的字符串中的字符數分詞器。
- `number_of_sequences`是我們想要的（截斷的）序列數來自我們的分詞器，（例如，100）。
- `sequence_length`是令牌返回的每個序列的令牌數。化器，（例如，1,024）。
- `characters_per_token`是每個輸出標記的平均字符數我們首先需要估計。

如果我們輸入一個包含`input_characters`個字符的字符串，我們將得到平均`number_of_sequences`個輸出序列，並且我們可以很容易地計算出通過刪除最後一個序列我們丟失了多少輸入數據。如果`number_of_sequences=100`這意味著我們堆疊大約 100 個序列並且最多丟失最後一個元素，這可能太短或太長。這相當於最多丟失我們數據集的 1%。

同時，這種方法通過切斷大部分文件結尾來確保我們不會引入偏差。

讓我們首先估計數據集中每個標記的平均字符長度：

```
示例，total_characters，total_tokens = 500, 0, 0 dataset =
load_dataset( transformersbook/codeparrot-train ,split= train ,streaming=True)
```

為了 `_`,
`tqdm(zip(range(examples), iter(dataset)), total=len(examples))` 中的示例：
`total_characters += len(example[content]) total_tokens += len(tokenizer(example[content]).tokens())`

`characters_per_token = total_characters / total_tokens`

打印 (`characters_per_token`)

3.6233025034779565

有了它，我們就擁有了創建自己的`IterableDataset`（這是 PyTorch 提供的輔助類）所需的一切，以便為模型準備恆定長度的輸入。我們只需要從`IterableDataset`繼承並設置`__iter__()`函數，該函數使用我們剛剛走過的邏輯產生下一個元素：

從`torch.utils.data`

導入`torch`導入`IterableDataset`

類`ConstantLengthDataset (IterableDataset) :`

```
def __init__(self, tokenizer, dataset, seq_length=1024, num_of_sequences=1024,
            chars_per_token=3.6): self.tokenizer = tokenizer
            self.concat_token_id = tokenizer.eos_token_id self.dataset = dataset
            self.seq_length = seq_length self.input_characters = seq_length *
            chars_per_token * num_of_sequences
```

```
def __iter__(self): iterator =
    iter(self.dataset) more_examples =
    True while more_examples: buffer,
    buffer_len = [], 0 while True:
```

如果`buffer_len >= self.input_characters: m=f 緩衝區
 已滿 :{buffer_len}>={self.input_characters:.0f} print(m) break`

嘗試：
`m=f 填充緩衝區 :{buffer_len}<={self.input_characters:.0f} print(m)`
`buffer.append(next(iterator)[content]) buffer_len += len(buffer[-1])` 除了
`StopIteration: iterator = iter(self.dataset)`

`all_token_ids = []`

```

tokenized_inputs = self.tokenizer(buffer, truncation=False) for tokenized_input in
tokenized_inputs[ input_ids ]: for tokenized_input in tokenized_inputs:
    all_token_ids.extend(tokenized_input + [self.concat_token_id])

for i in range(0, len(all_token_ids), self.seq_length): input_ids = all_token_ids[i : i + self.seq_length] if len(input_ids) == self.seq_length: yield
    torch.tensor(input_ids)

```

`__iter__()`函數構建一個字符串緩衝區，直到它包含足夠的字符。緩衝區中的所有元素都被標記化並與 EOS 標記連接，然後 `all_token_ids` 中的長序列被分塊成 `seq_length` 大小的切片。通常，我們需要注意掩碼來堆疊不同長度的填充序列，並確保在訓練期間忽略填充。我們通過只提供相同（最大）長度的序列來處理這個問題，所以我們不需要這裡的掩碼，只返回 `input_ids`。讓我們測試我們的可迭代數據集：

```

shuffled_dataset = dataset.shuffle(buffer_size=100)
constant_length_dataset = ConstantLengthDataset(tokenizer, shuffled_dataset,
                                                num_of_sequences=10)
dataset_iterator = iter(constant_length_dataset)

lengths = [len(b) for b in zip(range(5), dataset_iterator)] print(f' 序列長度 {lengths} ')

```

填充緩衝區 :0<36864
 填充緩衝區 :3311<36864
 填充緩衝區 :9590<36864
 填充緩衝區 :22177<36864
 填充緩衝區 :25530<36864
 填充緩衝區 :31098<36864
 填充緩衝區 :32232<36864
 填充緩衝區 :33867<36864
 緩衝區已滿 :41172>=36864
 序列長度 :[1024, 1024, 1024, 1024, 1024]

很好，這按預期工作，我們為模型獲得了恆定長度的輸入。現在我們有了可靠的模型數據源，是時候構建實際的訓練循環了。



請注意，我們在創建 `ConstantLengthDataset` 之前打亂了原始數據集。由於這是一個可迭代的數據集，我們不能在一開始就打亂整個數據集。相反，我們設置了一個大小為 `buffer_size` 的緩衝區，並在從數據集中獲取元素之前打亂該緩衝區中的元素。

我們現在定義訓練循環

擁有一個訓練循環的所有元素。訓練我們自己的語言模型的一個明顯限制是我們將使用的 GPU 的內存限制。即使在現代顯卡上，您也無法在合理的時間內以 GPT-2 規模訓練模型。

在本教程中，我們將實現數據並行性，這將幫助我們利用多個 GPU 進行訓練。幸運的是，我們可以使用 Accelerate 使我們的代碼可擴展。

Accelerate 庫旨在簡化分佈式訓練以及更改訓練的底層硬件。我們也可以使用 Trainer 進行分佈式訓練，但 Accelerate 使我們能夠完全控制訓練循環，這就是我們想要在這裡探索的內容。



Accelerate 提供了一個簡單的 API，使訓練腳本以混合精度和任何類型的分佈式設置（單個 GPU、多個 GPU 和 TPU）運行。然後，相同的代碼可以在您的本地機器上無縫運行以進行調試，或者在您強大的訓練集群上無縫運行以進行最終訓練。您只需對本機 PyTorch 訓練循環進行少量更改：

```
導入火炬導入
torch.nn.functional as F from datasets
import load_dataset + from accelerate import
Accelerator

- 設備 = cpu + 加速
器 = 加速器()

- 模型 = torch.nn.Transformer().to(設備) + 模型 =
torch.nn.Transformer()
優化器 = torch.optim.Adam(model.parameters())
dataset =
load_dataset( my_dataset ) data =
torch.utils.data.DataLoader(dataset, shuffle=True) + model, optimizer, data =
accelerator.prepare(model, 優化器, 數據)

model.train() for
epoch in range(10) : 對於源, 數據
    中的目標：
    - source = source.to(device) targets
    = targets.to(device) optimizer.zero_grad()
    output = model(source) loss =
        F.cross_entropy(output, targets)
    loss.backward() 加速器.backward(loss) 優化器步()
    +

```

更改的核心部分是對 prepare() 的調用，它確保模型、優化器和數據加載器都已準備好並分佈在基礎設施上。

PyTorch 訓練循環的這些微小變化使您能夠輕鬆地跨不同基礎設施擴展訓練。考慮到這一點，讓我們開始建立我們的培訓

編寫腳本並定義一些輔助函數。首先，我們設置用於訓練的超參數並將它們包裝在命名空間中以便於訪問：

從argparse導入命名空間

```
# 註釋參數對應小模型config = { train_batch_size : 2, #12
    valid_batch_size : 2, #12 weight_decay : 0.1, shuffle_buffer :
    1000, learning_rate : 2e-4, #5e-4 "lr_scheduler_type" :
    "餘弦", "num_warmup_steps" : 750, #2000
    "gradient_accumulation_steps" : 16, #1
    "max_train_steps" : 50000, #150000
    "max_eval_steps" : -1, "seq_length" : 1024, "種
    子" : 1, save_checkpoint_steps : 50000} #15000
```

args =命名空間 (**配置)

接下來，我們為訓練設置日誌記錄。由於我們是從頭開始訓練模型，因此訓練運行需要一段時間並且需要昂貴的基礎設施。因此，我們希望確保所有相關信息都已存儲且易於訪問。

setup_logging ()方法設置了三個級別的日誌記錄：使用標準的 Python Logger，**張量板**，和**權重與偏差**。根據您的偏好和用例，您可以在此處添加或刪除日誌記錄框架：

從torch.utils.tensorboard導入SummaryWriter導入日誌導入wandb

```
def setup_logging(project_name): logger =
    logging.getLogger(__name__)
    logging.basicConfig(format= "%(asctime)s - %
        (levelname)s - %(name)s - %(message)s", datefmt= "%m/%d/%Y %H:%M:%S", 級別
        =logging.INFO, handlers=[ logging.FileHandler(f log/
        debug_{accelerator.process_index}.log ), logging.StreamHandler() ]) if
        accelerator.is_main_process: # 我們只想設置一次日誌記錄

    wandb.init(project=project_name, config=args) run_name =
    wandb.run.name tb_writer = SummaryWriter()
    tb_writer.add_hparams(vars(args), { 0 : 0})
    logger.setLevel(logging.INFO)數據
    集.utils.logging.set_verbosity_debug()
    transformers.utils.logging.set_verbosity_info() else: tb_writer
    = None

    run_name =
    logger.setLevel(記錄.ERROR)
```

```
datasets.utils.logging.set_verbosity_error()
transformers.utils.logging.set_verbosity_error()返回記錄器、
tb_writer、run_name
```

每個 worker 都有一個唯一的accelerator.process_index，我們將其與文件處理程序一起使用，將每個 worker 的日誌寫入一個單獨的文件。我們還使用了accelerator.is_main_process屬性，該屬性僅適用於 main worker。

我們確保不會多次初始化TensorBoard和Weights & Biases記錄器，並降低其他worker的日誌記錄級別。我們返回自動生成的唯一wandb.run.name，稍後我們將使用它來命名Hub上的實驗分支。

我們還將定義一個函數來記錄TensorBoard和Weights & Biases的指標。我們在這裡再次使用accelerator.is_main_process來確保我們只記錄一次指標，而不是為每個工作人員記錄指標：

```
def log_metrics(step, metrics):
    logger.info(f Step {step}: {metrics} ) if
    accelerator.is_main_process: wandb.log(metrics)
        [tb_writer.add_scalar(k, v, step) for k, v in
         metrics.items()]
```

接下來，讓我們編寫一個函數，使用我們全新的ConstantLengthDataset類為訓練集和驗證集創建數據加載器：

```
從torch.utils.data.dataloader導入DataLoader

def create_dataloaders (數據集名稱) :
    train_data = load_dataset(dataset_name+ -train ,split= train ,
        流媒體=True)
    train_data = train_data.shuffle(buffer_size=args.shuffle_buffer, seed=args.seed)

    valid_data = load_dataset(dataset_name+ -valid ,split= validation ,streaming=True)

    train_dataset = ConstantLengthDataset(tokenizer, train_data,
        seq_length=args.seq_length)
    valid_dataset = ConstantLengthDataset(tokenizer, valid_data, seq_length=args.seq_length)

    train_dataloader=DataLoader(train_dataset, batch_size=args.train_batch_size)
    eval_dataloader=DataLoader(valid_dataset, batch_size=args.valid_batch_size)返回train_dataloader,
    eval_dataloader
```

最後，我們將數據集包裝在DataLoader中，它也處理批處理。

👉 Accelerate 將負責將批次分發給每個工人。

我們需要實施的另一個方面是優化。我們將在主循環中設置優化器和學習率計劃，但我們在這裡定義了一個輔助函數來區分應該接收權重衰減的參數。一般來說，偏差和LayerNorm權重不受權重衰減的影響：

```

def get_grouped_params(model, no_decay=[ bias , LayerNorm.weight ]): params_with_wd,
    params_without_wd = [], [] for n, p in model.named_parameters():

        如果有 (nd in n for nd in no_decay) :
            params_without_wd.append (p)否則：
            params_with_wd.append (p)

    返回[{ params :params_with_wd, weight_decay :args.weight_decay},
          { params :params_without_wd, weight_decay :0.0}]

```

最後，我們想不時地在驗證集上評估模型，所以讓我們添加一個評估函數，我們可以調用它來計算評估集上的損失和困惑：

```

def evaluate():
    model.eval()
    losses = [] for
    step, batch in enumerate(eval_dataloader):
        使用torch.no_grad(): outputs
            = model(batch, labels=batch) loss =
            outputs.loss.repeat(args.valid_batch_size)
            losses.append(accelerator.gather(loss))如果args.max_eval_steps >
            0且step >= args.max_eval_steps: break loss = torch.mean(torch.cat(losses))嘗試： perplexity
            = torch.exp(loss)除了OverflowError : perplexity = torch.tensor(float( inf ))
    返回loss.item(), perplexity.item()

```

困惑度衡量模型的輸出概率分佈預測目標標記的程度。因此，較低的困惑對應於更好的性能。

請注意，我們可以通過對從模型輸出中獲得的交叉熵損失求冪來計算困惑度。尤其是在損失仍然很高的訓練開始時，計算困惑度時可能會出現數值溢出。

我們捕獲了這個錯誤，並在這些情況下將困惑度設置為無窮大。

在我們將它們全部放在訓練腳本中之前，我們還要使用一個額外的函數。正如您現在所知，Hugging Face Hub 在後台使用 Git 來存儲和版本化模型和數據集。使用huggingface_hub庫中的Repository類，您可以以編程方式訪問存儲庫並拉取、分支、提交或推送。我們將在我們的腳本中使用它在訓練期間不斷地將模型檢查點推送到 Hub。

現在我們已經準備好所有這些輔助函數，可以開始編寫訓練腳本的核心部分了：

```

set_seed (args.seed)

# 加速器accelerator
= Accelerator()

```

```

samples_per_step = accelerator.state.num_processes * args.train_batch_size

# 日誌記錄
器，tb_writer，run_name=setup_logging(project_name.split( / )[1])
logger.info(accelerator.state)

#如果accelerator.is_main_process:
加载模型和分詞器：
    hf_repo = Repository( ./ , clone_from=project_name, revision=run_name)模型=
AutoModelForCausalLM.from_pretrained( ./ , gradient_checkpointing=True) tokenizer =
AutoTokenizer.from_pretrained( ./ )

# 加載數據集和數據加載器
train_dataloader, eval_dataloader = create_dataloaders(dataset_name)

#準備優化器和學習率調度器_ _

def get_lr():返回優
化器.param_groups[0][ lr ]

#使用我們的`accelerator`準備一切（args的順序並不重要）

# 訓練模型
model.train()
completed_steps = 0 for
step, batch in enumerate(train_dataloader, start=1): loss = model(batch,
    labels=batch).loss log_metrics(step, { lr :get_lr(), samples 步驟
    *樣本每步 ,
        steps :completed_steps, loss/train :loss.item()}) loss = loss /
args.gradient_accumulation_steps accelerator.backward(loss) if step %
args.gradient_accumulation_steps == 0: optimizer.step() lr_scheduler.step () optimizer.zero_grad()
completed_steps += 1

if step % args.save_checkpoint_steps == 0: logger.info( 評
估和保存模型檢查點 ) eval_loss, perplexity = evaluate() log_metrics(step,
{ loss/eval :eval_loss, perplexity :perplexity})加速器...-----_
-----.max_train_steps:

```

休息

```
# 評估並保存最後一個檢查點logger.info( Evaluating and
saving model after training ) eval_loss, perplexity = evaluate() log_metrics(step,
{ loss/eval : eval_loss, perplexity : perplexity}) accelerator.wait_for_everyone
() unwrapped_model = accelerator.unwrap_model(model) if accelerator.is_main_process:
unwrapped_model.save_pretrained( ./ ) hf_repo.push_to_hub(commit_message=f 最終模型 )
```

這是一個相當大的代碼塊，但請記住，這是在分佈式基礎設施上訓練奇特的大型語言模型所需的全部代碼。讓我們稍微解構腳本並突出顯示最重要的部分：

模型保存我們

從模型存儲庫中運行腳本，並在開始時檢查一個新分支，該分支以我們從 Weights & Biases 獲得的run_name命名。稍後，我們在每個檢查點提交模型並將其推送到 Hub。使用該設置，每個實驗都在一個新分支上，每個提交代表一個模型檢查點。請注意，我們需要調用wait_for_everyone()和unwrap_model()以確保模型在存儲時正確同步。

優化 對於模型

優化，我們在線性預熱期後使用帶有餘弦學習率計劃的AdamW。對於超參數，我們嚴格遵循 GPT-3 論文中描述的類似大小模型的參數。⁸

評估

每次保存時，我們都會在評估集上評估模型，即每次save_checkpoint_steps和訓練後。除了驗證損失，我們還記錄了驗證困惑。

梯度積累和檢查點

所需的批量大小不適合 GPU 的內存，即使我們在最新的 GPU 上運行也是如此。因此，我們實現了梯度累積，它通過多次反向傳遞收集梯度，並在累積足夠的梯度後進行優化。在第 6 章中，我們看到瞭如何使用Trainer 來做到這一點。對於大型模型，即使是單個批處理也不太適合單個 GPU。使用稱為梯度檢查點的方法，我們可以交換一些內存佔用

⁸ T. Brown 等人，“語言模型是小概率學習者”，（2020）。

大約 20% 的訓練減速。⁹這使我們能夠在單個 GPU 中擬合大型模型。

一個可能仍然有點模糊的方面是在多個 GPU 上訓練模型意味著什麼。根據模型的大小和數據量，有多種方法可以以分佈式方式訓練模型。Accelerate 使用的方法稱為 DataDistributedParallelism (DDP)。⁹這種方法的主要優點是它允許您使用不適合任何單個 GPU 的更大批量大小更快地訓練模型。該過程如圖 10-6 所示。

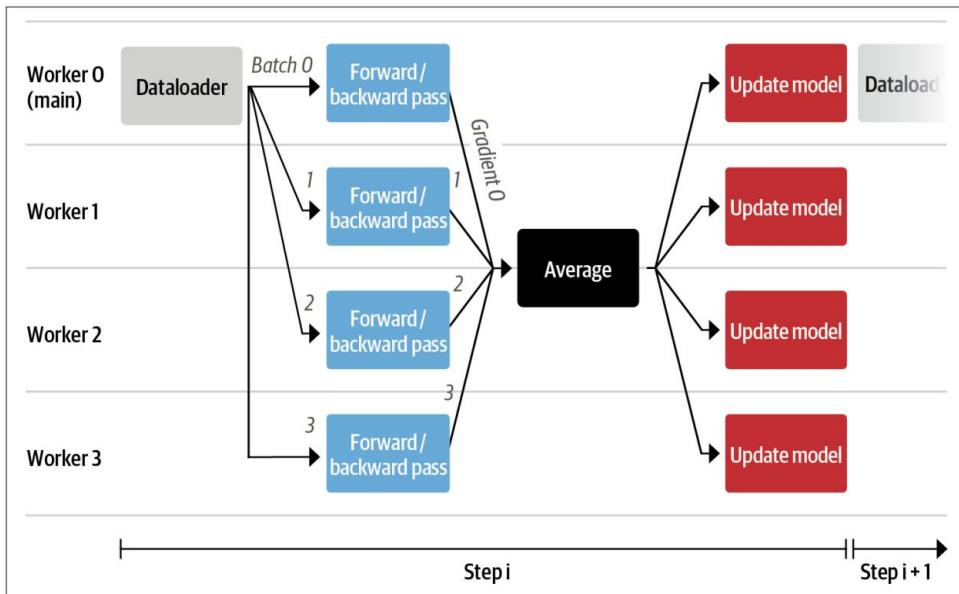


圖 10-6 ◦四個 GPU 的 DDP 中的處理步驟圖示

讓我們逐步了解管道：

1. 每個worker由一個GPU組成。在 Accelerate 中，有一個在主進程上運行的數據加載器，它準備批量數據並將它們發送給所有工作人員。
2. 每個 GPU 接收一批數據並使用模型的本地副本計算損失和來自前向和反向傳遞的相應累積梯度。
3. 來自每個節點的梯度使用歸約模式進行平均，平均老化的梯度被發送回每個工人。

⁹您可以在 OpenAI 的[發布帖子](#)中閱讀有關梯度檢查點的更多信息。

4. 使用優化器分別在每個節點上應用梯度。

儘管這看起來像是多餘的工作，但它避免了在節點之間傳輸大型模型的副本。我們需要至少更新一次模型，如果沒有這種方法，其他節點都需要等待，直到它們收到更新版本。

5. 一旦所有模型都更新了，我們就重新開始，主要工作人員準備新的批次。

這種簡單的模式使我們能夠通過擴展到可用 GPU 的數量來極快地訓練大型模型，而無需太多額外的邏輯。然而，有時這還不夠。例如，如果模型不適合單個 GPU，您可能需要更複雜的並行策略。現在我們已經有了訓練所需的所有部分，是時候開始工作了！正如您將在下一節中看到的，這非常簡單。

訓練運行我們將訓練腳

本保存在一個名為 codeparrot_training.py 的文件中，以便我們可以在我們的訓練服務器上執行它。為了讓生活更輕鬆，我們將把它與包含所有必需的 Python 依賴項的 requirements.txt 文件一起添加到 Hub 上的模型存儲庫。請記住，Hub 上的模型本質上是 Git 存儲庫，因此我們只需克隆存儲庫，添加我們想要的任何文件，然後將它們推送回 Hub。在訓練服務器上，我們可以使用以下幾個命令啟動訓練：

```
$ git clone https://huggingface.co/transformersbook/codeparrot $ cd codeparrot $  
pip install -r requirements.txt $ wandb login $ accelerate config $ accelerate launch  
codeparrot_training.py
```

就是這樣，我們的模型現在正在訓練！請注意，wandb 登錄將提示您使用 Weights & Biases 進行身份驗證以進行日誌記錄。accelerate config 命令將指導您完成基礎設施的設置；您可以在表 10-2 中看到用於此實驗的設置。我們使用 **a2-megagpu-16g** 實例對於所有實驗，這是一個工作站，有 16 個 A100 GPU，每個 40 GB 內存。

表 10-2。用於訓練 CodeParrot 模型的配置

環境	價值
計算環境？多GPU 有多少台機器？	
1 深度？	不
多少道工序？16 使用	
FP16？	是的

對於小型和大型模型，在該基礎設施上使用這些設置運行訓練腳本分別需要大約 24 小時和 7 天。如果您訓練自己的自定義模型，請確保您的代碼在較小的基礎架構上順利運行，以確保昂貴的長期運行也能順利進行。完整訓練運行成功完成後，您可以使用以下命令將集線器上的實驗分支合併回主分支：

```
$ git checkout main $ git
merge <RUN_NAME> $ git push
```

當然，RUN_NAME 應該是您要合併的集線器上的實驗分支的名稱。現在我們有了一個經過訓練的模型，讓我們看看如何調查它的性能。

結果與分析

在焦急地監視日誌一周後，你可能會看到如圖 10-7 所示的損失和困惑曲線。訓練損失和驗證困惑持續下降，損失曲線在對數對數尺度上看起來幾乎是線性的。我們還看到，儘管整體訓練時間更長，但大型模型在處理的標記方面收斂得更快。

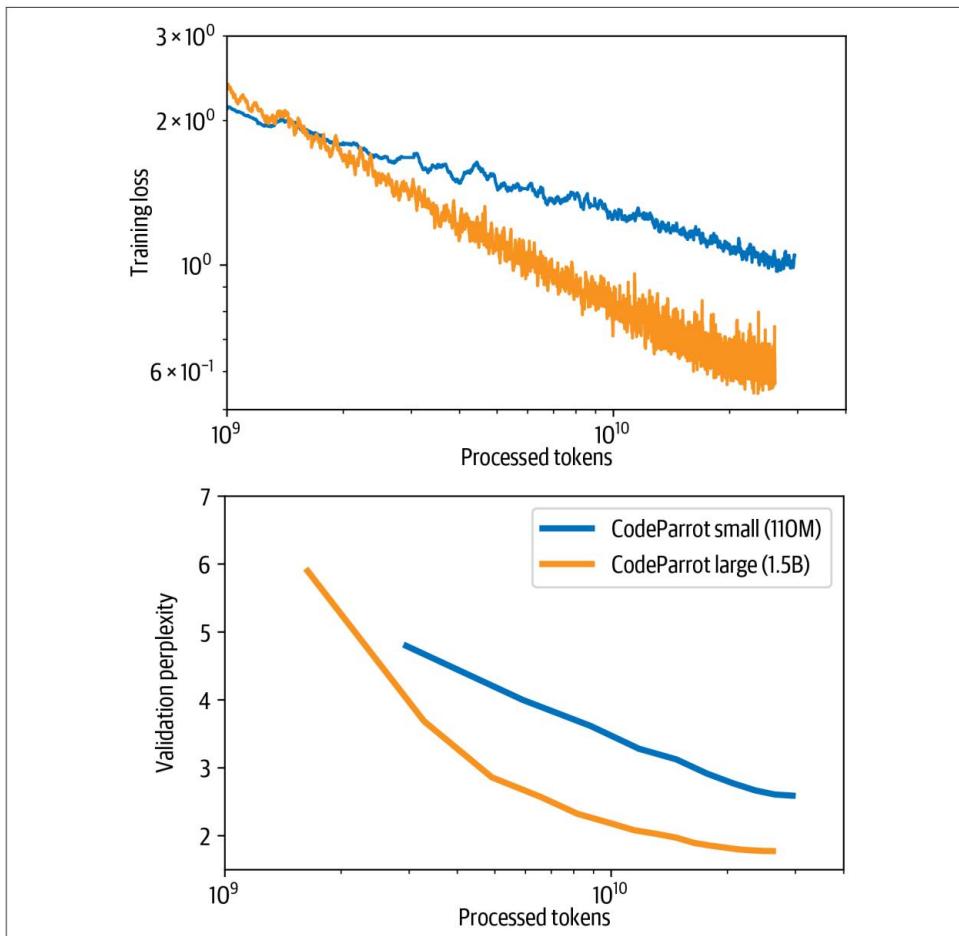


圖 10-7 °訓練損失和驗證困惑作為小型和大型 CodeParrot 模型的已處理令牌的函數

那麼我們可以用剛從 GPU 烤箱中取出的新鮮出爐的語言模型做什麼呢？好吧，我們可以用它來為我們編寫一些代碼。我們可以進行兩種類型的分析：定性和定量。在前者中，我們查看具體示例並嘗試更好地理解模型在哪些情況下成功以及在哪些情況下失敗。在後一種情況下，我們在大量測試用例上以統計方式評估模型的性能。在本節中，我們將探討如何使用我們的模型。首先，我們將看幾個例子，然後我們將簡要討論如何系統地、更穩健地評估模型。首先，讓我們將小模型包裝在一個管道中，並用它來繼續一些代碼輸入：

從變壓器導入管道，`set_seed`

```
model_ckpt=  transformersbook/codeparrot-small  generation =
pipeline( text-generation ,model=model_ckpt,device=0)
```

現在我們可以使用生成管道從給定的提示中生成候選補全。默認情況下，管道將生成代碼直到達到預定義的最大長度，並且輸出可能包含多個函數或類。因此，為了保持輸出簡潔，我們將實現一個first_block()函數，它使用正則表達式來提取函數或類的第一次出現。下面的complete_code ()函數應用此邏輯打印出 CodeParrot 生成的補全：

```
從變形金剛導入
set_seed __

def first_block (字符串) :
    返回re.split( \nclass\nndef\n#\n@\nprint\nif ,string)[0].rstrip()

def complete_code(pipe, prompt, max_length=64, num_completions=4, seed=1): set_seed(seed) gen_kwargs =
    { temperature :0.4, top_p :0.95, top_k :0, num_beams :1, "do_sample":真,}

code_gens =生成 (提示, num_return_sequences=num_completions, max_length=max_length,
                  **gen_kwargs)
code_strings =[] for
code_gen in code_gens:
    generated_code =first_block(code_gen[ generated_text ][len(prompt):]) code_strings.append(generated_code)

print(( \n + = *80 + \n ).join(code_strings))
```

讓我們從一個簡單的例子開始，讓模型為我們編寫一個計算矩形面積的函數：

```
prompt=      def area_of_rectangle(a: float, b: float):      返回矩形的面
        積。
            complete_code(generation, prompt)
```

```
返回 math.sqrt(a * b)
=====
```

```
返回一個      * b / 2.0
=====
```

```
返回一個      *乙
=====
```

```
返回一個      *乙/乙
```

看起來不錯！雖然不是所有的世代都是正確的，但正確的解決方案就在那裡。現在，該模型還能解決更複雜的從 HTML 字符串中提取 URL 的任務嗎？讓我們來看看：

```

prompt =     def get_urls_from_html(html):      獲
            取 HTML 字串中的所有嵌入 URL。
complete_code(generation, prompt)

如果不是 html :返
    回 []
    返回 [re.findall 中 url 的 url(r <a href= ([^/]+/[^\ ]+?) > ,html)]
=====
=====

    返回 [re.findall 中 url 的 url(r <a href= (.*) ,html) if url]

=====
=====

    返回 [re.findall(r <a href= (/.* ) ,html) 中的 url 的 url]
=====

    返回 re.findall(r <a href= (.*) class= url [^>]*> ,html)

```

雖然在第二次嘗試中沒有完全正確，但其他三代都是正確的。我們可以在Hugging Face首頁測試功能：

導入請求

```

def get_urls_from_html(html): return [url
    for url in re.findall(r <a href= (.*) ,html) if url]

print( | .join(get_urls_from_html(requests.get( https://hf.co/ ).text)))

https://github.com/huggingface/transformers | /艾倫奈 | /臉書 | /小行星隊 | /谷歌 | /亞馬遜 | /語音大
腦 | /微軟 | /語法 | /模型 | /推理API | /distilbert-base-uncased | /dbmdz/bert-large-cased-finetuned-conll03-english
| https://huggingface.co/transformers | https://arxiv.org/abs/1811.06031 | https://arxiv.org/abs/1803.10631 |
https://transformer.huggingface.co/ | /核心 | https://medium.com/huggingface/distilbert-8cf3380435b5

```

我們可以看到所有以https開頭的URL都是外部頁面，而其他的是主網站的子頁面。這正是我們想要的。最後，讓我們加載大型模型，看看我們是否可以使用它來將函數從純Python轉換為NumPy：

```

model_ckpt = transformersbook/codeparrot generation
= pipeline( text-generation ,model=model_ckpt, device=0)

prompt =     # 原生 python 中的函數 :def mean(a): return
sum(a)/len(a)

# 使用 numpy 的相同函數 :import numpy as
np def mean(a):
complete_code(generation, prompt,
max_length=64)

```

將 `pad_token_id` 設置為 `eos_token_id` :0 用於開放式生成。

返回 np.mean(a)

返回 np.mean(a)

返回 np.mean(a)

返回 np.mean(a)

成功了 !讓我們看看是否也可以使用 CodeParrot 模型來幫助我們構建 Scikit-learn 模型：

```
prompt =      X = np.random.randn(100, 100) y =
np.random.randint(0, 1, 100)
```

用 20 個估計器擬合隨機森林分類器

```
complete_code (生成 :提示 , max_length=96)
```

將 `pad_token_id` 設置為 `eos_token_id` :0 用於開放式生成。

```
reg = DummyRegressor()
```

```
森林 = RandomForestClassifier(n_estimators=20)
```

```
forest.fit(X, y)
```

```
clf = ExtraTreesClassifier(n_estimators=100, max_features= sqrt ) clf.fit(X, y)
```

```
clf = RandomForestClassifier(n_estimators=20, n_jobs=n_jobs, random_state=1) clf.fit(X, y)
```

```
clf = RandomForestClassifier(n_estimators=20) clf.fit(X, y)
```

儘管在第二次嘗試中它試圖訓練一個額外的樹分類器 ,它生成了我們在其他情況下要求的內容。

在第 5 章中 ,我們探索了一些衡量生成文本質量的指標。

其中包括經常用於此目的的 BLEU 分數。雖然這個指標通常有局限性 ,但它特別不適合我們的用例。

BLEU 分數衡量參考文本和生成文本之間 n-gram 的重疊。在編寫代碼時 ,我們在變量方面有很大的自由度

和類，一個程序的成功與否並不取決於命名方案，只要保持一致即可。然而，BLEU 分數會懲罰偏離參考命名的一代，這實際上幾乎是不可能預測的（即使對於人類編碼員也是如此）。

在軟件開發中，有更好、更可靠的方法來衡量代碼的質量，例如單元測試。這就是所有 OpenAI Codex 模型的評估方式：通過一組單元測試為編碼任務運行幾代代碼，併計算通過測試的幾代代碼。¹⁰ 為了進行適當的性能衡量，我們應該應用相同的評估對我們的模型進行養生，但這超出了本章的範圍。您可以在[模型隨附的博客文章](#)中找到有關 CodeParrot 如何在 HumanEval 基準測試中執行的詳細信息。

結論

讓我們退後一步，思考一下我們在本章中取得的成就。我們著手為 Python 創建代碼自動完成功能。首先，我們構建了一個適合預訓練大型語言模型的自定義大型數據集。然後我們創建了一個自定義分詞器，它能夠使用該數據集高效地編碼 Python 代碼。最後，在 Accelerate 的幫助下，我們將所有內容放在一起並編寫了一個訓練腳本，以在多 GPU 基礎設施上從頭開始訓練 GPT-2 模型的小型和大型版本，代碼不到 200 行。調查模型輸出，我們看到它可以生成合理的代碼延續，並且我們討論瞭如何系統地評估模型。

您現在不僅知道如何微調 Hub 上的許多預訓練模型中的任何一個，還知道如何在您有足夠的數據和可用計算資源時從頭開始預訓練自定義模型。您現在已準備好使用轉換器處理幾乎所有 NLP 用例。所以問題是：下一步去哪裡？在下一章和最後一章中，我們將了解該領域目前的發展方向，以及 NLP 轉換器模型之外的令人興奮的新應用和領域可以解決的問題。

¹⁰ M. Chen 等人，“[評估在代碼上訓練的大型語言模型](#)”，(2021)。

第十一章

未來發展方向

在本書中，我們探索了 Transformer 在廣泛的 NLP 任務中的強大功能。在最後一章中，我們將轉變視角，審視這些模型當前面臨的一些挑戰，以及試圖克服這些挑戰的研究趨勢。在第一部分中，我們探討了在模型和語料庫大小方面擴大變換器的主題。然後我們將注意力轉向已經提出的各種技術，這些技術可以使自註意力機制更有效。最後，我們探索了新興的、令人興奮的多模式轉換器領域，它可以對跨文本、圖像和音頻等多個領域的輸入進行建模。

縮放變壓器

2019 年，研究員 Richard Sutton 寫了一篇題為 “苦澀”的挑釁性文章
課”他在其中辯稱：

從 70 年的 AI 研究中可以讀到的最大教訓是，利用計算的通用方法最終是最有效的，而且差距很大……為了尋求在短期內產生影響的改進，研究人員尋求利用他們對該領域的人類知識，但從長遠來看，唯一重要的是利用計算。這兩者不必相互背道而馳，但在實踐中它們傾向於……。人類知識方法往往會使方法複雜化，使它們不太適合利用利用計算的一般方法。

這篇文章提供了幾個歷史例子，例如下國際象棋或圍棋，在這些例子中，在人工智能系統中編碼人類知識的方法最終被增加的計算所超越。Sutton 稱這是 AI 研究領域的 “慘痛教訓”：

我們必須吸取慘痛的教訓，即從長遠來看，我們認為自己的想法是行不通的……。應該從慘痛的教訓中吸取的一件事是通用方法的強大功能，即使可用計算變得非常大，這些方法也會隨著計算的增加而繼續擴展。以這種方式似乎可以任意擴展的兩種方法是搜索和學習。

現在有跡象表明類似的教訓正在變形金剛中發揮作用；雖然許多早期的 BERT 和 GPT 後代專注於調整架構或預訓練目標，但 2021 年中期表現最好的模型，如 GPT-3，基本上是原始模型的基本放大版本，沒有太多架構修改。在圖 11-1 中，您可以看到自 2017 年發布原始 Transformer 架構以來最大模型的開發時間線，這表明模型大小在短短幾年內增加了四個數量級以上！

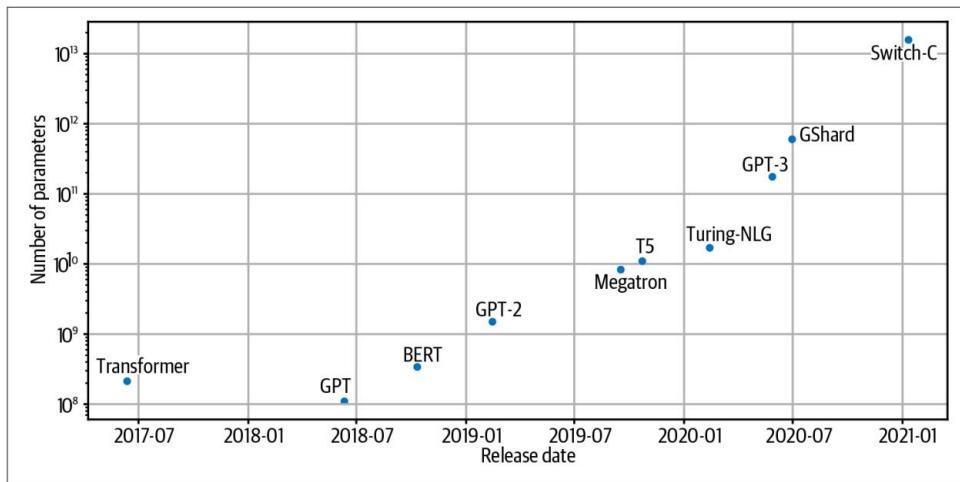


圖 11-1 突出的 Transformer 架構隨時間推移的參數計數

這種顯著增長的動機是經驗證據表明，大型語言模型在下游任務中表現更好，並且在 10 到 1000 億參數範圍內出現了零樣本和少樣本學習等有趣的能力。然而，參數的數量並不是影響模型性能的唯一因素；計算和訓練數據的數量也必須同步縮放以訓練這些怪物。鑑於像 GPT-3 這樣的大型語言模型估計要花費 460 萬美元為了進行訓練，顯然希望能夠提前估計模型的性能。有點令人驚訝的是，語言模型的性能似乎服從幕律與模型大小和其他因素的關係，這些因素被編入一組比例定律。¹讓我們來看看這個令人興奮的研究領域。

1 J. Kaplan 等人，“神經語言模型的縮放法則”，（2020）。

縮放定律縮

放定律允許人們通過研究語言模型在不同的計算預算 C 、數據集大小 D 和模型大小 N 下的行為，對語言模型的“越大越好”範式進行經驗量化。

2基本思想是繪製交叉熵損失 L 對這三個因素的依賴關係，並確定是否存在關係。

對於 GPT 系列中的自回歸模型，生成的損失曲線如圖 11-2 所示，其中每條藍色曲線代表單個模型的訓練運行。

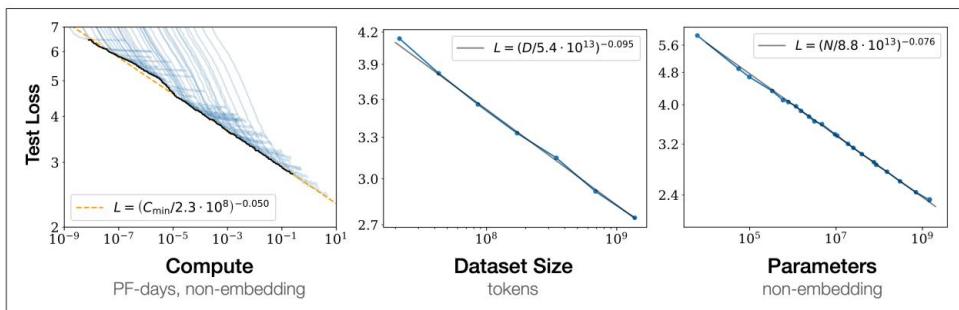


圖 11-2 ◦測試損失的幕律縮放與計算預算（左）、數據集大小（中）和模型大小（右）（由 Jared Kaplan 提供）

從這些損失曲線我們可以得出一些結論：

性能與規模的關係

儘管許多 NLP 研究人員專注於架構調整或超參數優化（如調整層數或註意力頭）以提高一組固定數據集的性能，但縮放定律的含義是通往更好模型的更有效途徑是專注於串聯增加 N 、 C 和 D 。

平滑幕律 測試損失 L

與 N 、 C 和 D 中的每一個都具有跨越幾個數量級的幕律關係（幕律關係在對數對數尺度上是線性的）。

對於 $X = N \cdot C \cdot D$ ，我們可以將這些幕律關係表示為 $L \propto 1/X^\alpha$ ，其中 α 是一個比例指數，由圖所示損耗曲線的擬合確定。典型值在 0.05–0.095 范圍內，和圖 11-2 中的一個吸引力。這些幕律的一個重要特徵是可以推斷損失曲線的早期部分，以預測如果訓練時間更長，損失的近似值是多少。

3 T. Henighan 等人，“自回歸生成模型的縮放法則”，(2020)。

2 數據集大小以標記的數量衡量，而模型大小不包括參數嵌入層。

3 T. Henighan 等人，“自回歸生成模型的縮放法則”，(2020)。

樣本效率 大型模

型能夠以較少的訓練步驟達到與較小模型相同的性能。這可以通過比較損失曲線在一定數量的訓練步驟中穩定的區域來看出，這表明與簡單地擴大模型相比，性能回報遞減。

有點令人驚訝的是，其他模式也觀察到了比例定律，例如圖像、視頻和數學問題解決，如圖 11-3 所示。

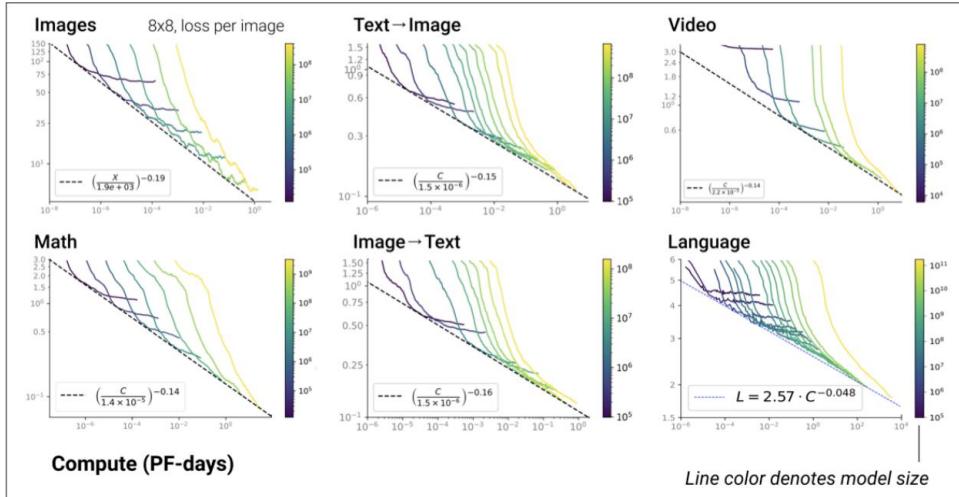


圖 11-3 ◦測試損失的幕律縮放與各種模式的計算預算（由 Tom Henighan 提供）

目前尚不清楚幕律縮放是否是 Transformer 語言模型的普遍屬性。目前，我們可以使用比例定律作為一種工具來推斷大型、昂貴的模型，而無需明確地訓練它們。然而，擴展並不像聽起來那麼容易。現在讓我們看看在繪製這個邊界時突然出現的一些挑戰。

擴展的挑戰雖然擴展在理論上聽

起來很簡單（“只需添加更多層！”），但在實踐中存在許多困難。以下是您在擴展語言模型時可能遇到的一些最大挑戰：

基礎設施供應

和管理可能跨越數百或數千個節點並具有盡可能多的 GPU 的基礎設施並不適合膽小的人。所需數量的節點是否可用？節點之間的通信是一個瓶子嗎？

脖子？解決這些問題需要的技能與大多數數據科學團隊的技能截然不同，通常需要熟悉運行大規模分佈式實驗的專業工程師參與。

成本

大多數 ML 從業者都經歷過半夜驚醒，一身冷汗的感覺，想起他們忘記關閉雲端那個花哨的 GPU。這種感覺在進行大規模實驗時會加劇，而且大多數公司無力負擔最大規模訓練模型所需的團隊和資源。訓練一個 GPT-3 大小的模型可能要花費數百萬美元，這不是許多公司都可以買到的那種零花錢。⁴

數據集管理

模型的好壞取決於它所訓練的數據。訓練大型模型需要大型、高質量的數據集。當使用數 TB 的文本數據時，確保數據集包含高質量文本變得更加困難，甚至預處理也變得具有挑戰性。此外，需要確保有一種方法可以控制這些語言模型在大規模網絡文本語料庫上訓練時可能獲得的性別歧視和種族主義等偏見。另一種類型的考慮圍繞可以嵌入大型文本數據集中的訓練數據和個人信息的許可問題。

模型評估 一旦模型

被訓練好，挑戰就不會停止。再次評估下游任務的模型需要時間和資源。此外，即使您確信自己創建了一個乾淨的數據集，您也需要探索模型中有偏見和有毒的世代。這些步驟需要時間，需要徹底執行，以盡量減少以後產生不利影響的風險。

部署 最後，為

大型語言模型提供服務也構成了重大挑戰。在第 8 章中，我們研究了一些方法，例如蒸餾、修剪和量化，以幫助解決這些問題。但是，如果您從一個大小為數百 GB 的模型開始，這可能還不夠。託管服務，例如 OpenAI API 或者 Hugging Face 的加速推理 API 旨在幫助不能或不想應對這些部署挑戰的公司。

⁴然而，最近提出了一種分佈式深度學習框架，使較小的團隊能夠以協作方式匯集他們的計算資源和預訓練模型。參見 M. Diskin 等人，“開放協作中的分佈式深度學習”，(2021)。

這絕不是一個詳盡無遺的清單，但它應該讓您了解將語言模型擴展到更大規模時需要考慮的各種因素和挑戰。雖然這些努力中的大部分都集中在一些擁有資源和專業知識來突破界限的機構周圍，但目前有兩個社區主導的項目旨在公開生產和探索大型語言模型：

BigScience

這是一個為期一年的研究研討會，從 2021 年持續到 2022 年，重點是大型語言模型。該研討會旨在促進圍繞這些模型的研究問題（能力、局限性、潛在改進、偏見、倫理、環境影響、在一般人工智能/認知研究領域中的作用）以及圍繞創建和研究的挑戰進行討論和反思。出於研究目的以及在研究界之間共享此類模型和數據集。協作任務涉及創建、共享和評估大型多語言數據集和大型語言模型。為這些協作任務分配了異常大的計算預算（數千個 GPU 上的數百萬個 GPU 小時）。如果成功，該研討會將在未來再次舉辦，重點是涉及一組更新的或不同的協作任務。如果你想加入這項工作，你可以在 [該項目的網站上找到更多信息](#)。

EleutherAI

這是一個由志願研究人員、工程師和開發人員組成的去中心化集體，專注於 AI 對齊、擴展和開源 AI 研究。它的目標之一是訓練和開源 GPT-3 大小的模型，該組織已經發布了一些令人印象深刻的模型，如 [GPT-Neo](#) 和 [GPT-J](#)，這是一個 60 億參數模型，是目前在零樣本性能方面表現最好的公開可用 transformer。您可以在 EleutherAI 的網站上找到更多信息。

現在我們已經探索瞭如何跨計算、模型大小和數據集大小擴展轉換器，讓我們研究另一個活躍的研究領域：使自註意力更有效。

請注意！

我們在整本書中都看到，自註意力機制在 Transformer 的架構中起著核心作用；畢竟，最初的 Transformer 論文叫做“Attention Is All You Need”！然而，self-attention 存在一個關鍵挑戰：由於權重是通過對序列中所有標記的成對比較生成的，因此在嘗試處理長文檔或將轉換器應用於語音處理等領域時，這一層成為計算瓶頸計算機視覺。在時間和內存複雜度方面，self-attention 層

Transformer 架構天真地像 n 序列一樣縮放。 $\mathcal{O}(n^2)$, 其中 n 是

因此，最近對變壓器的大部分研究都集中在提高自註意力的效率上。研究方向大致集中在圖 11-4 中。

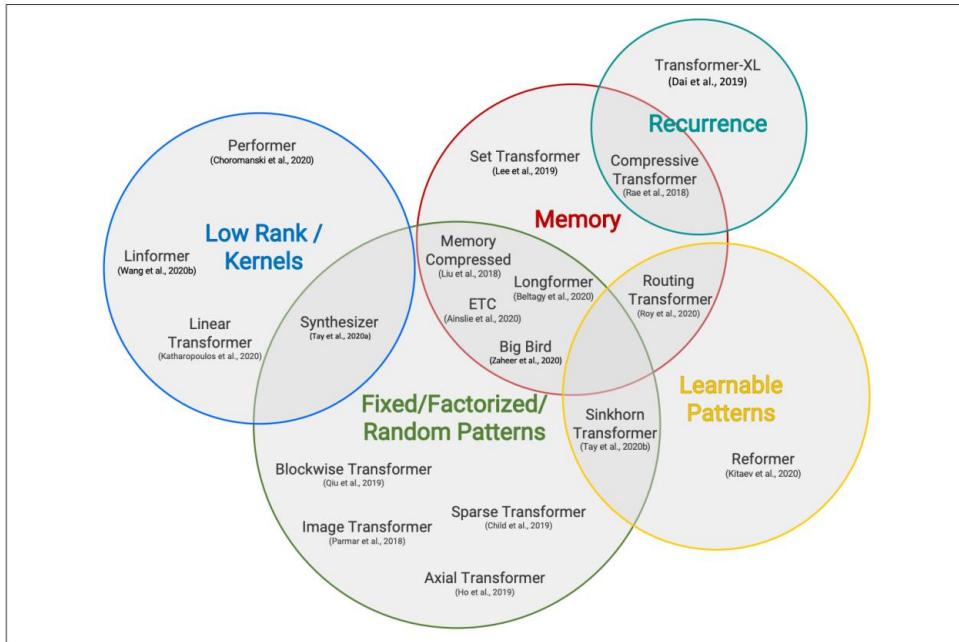


圖 11-4 提高注意力效率的研究方向總結（由 Yi Tay 等人提供）⁶

一種常見的模式是通過將稀疏性引入註意力機製或將核應用於注意力矩陣來提高注意力的效率。讓我們快速瀏覽一些最流行的提高自註意力的方法，從稀疏性開始。

稀疏注意力

減少在自我關注層中執行的計算數量的一種方法是簡單地限制生成的查詢密鑰對的數量

⁶ 雖然自註意力的標準實現有 $\mathcal{O}(n^2)$ 時間和內存複雜度，谷歌研究人員最近的一篇論文表明通過對操作進行簡單的重新排序，可以將內存複雜度降低到 $O \log n$ 。（ ）

6 Yi Tay 等人，“高效變壓器：一項調查”，(2020)。

根據一些預定義的模式。文獻中探索了許多稀疏模式，但大多數可以分解為圖 11-5 所示的少數“原子”模式。

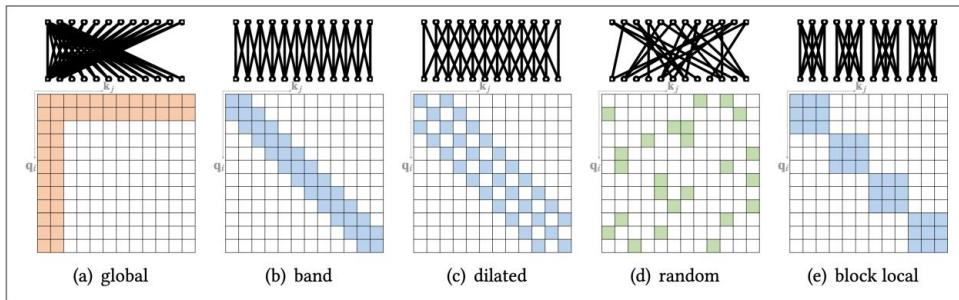


圖 11-5。自注意力的常見原子稀疏注意力模式：彩色方塊表示計算了注意力分數，而空白方塊表示分數被丟棄（由 Tianyang Lin 提供）

我們可以將這些模式描述如下：⁷

全球關注

在允許關注所有其他標記的序列中定義一些特殊標記

波段關注

計算對角帶上的注意力

注意力分散

通過使用帶間隙的擴大窗口跳過一些查詢鍵對

隨機關注

為每個查詢隨機抽取幾個鍵來計算注意力分數

阻止本地關注

將序列分成塊並將注意力限制在這些塊內

在實踐中，大多數具有稀疏注意力的變換器模型使用圖 11-5 所示的原子稀疏模式的混合來生成最終的注意力矩陣。如圖 11-6 所示，像 Longformer 這樣的模型混合使用全局注意力和波段注意力，而 BigBird 為混音添加隨機注意力。將稀疏性引入注意力矩陣使這些模型能夠處理更長的序列；在 Longformer 和 BigBird 的情況下，最大序列長度為 4,096 個標記，比 BERT 大 8 倍！

⁷ T. Lin 等人，“變壓器調查”，(2021)。

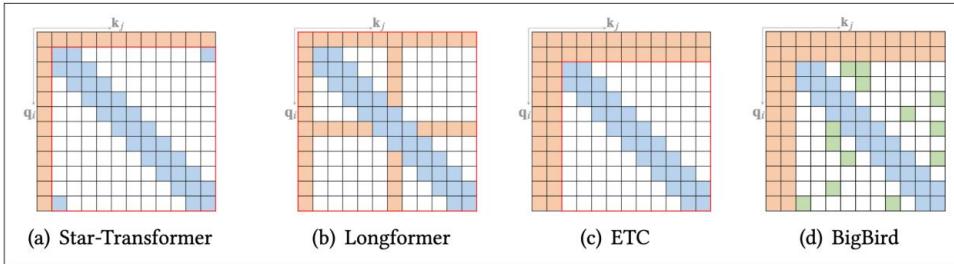


圖 11-6 ◦近期 Transformer 模型的稀疏注意力模式（由 Tianyang Lin 提供）



也可以以數據驅動的方式學習稀疏模式◦這種方法背後的基本思想是將標記聚類成塊◦例如◦改革者使用哈希函數將相似的標記聚集在一起◦

現在我們已經了解了稀疏性如何降低 self-attention 的複雜性◦讓我們來看看另一種流行的基於直接改變操作的方法◦

線性化注意力

另一種使自註意力更有效的方法是更改計算注意力分數時涉及的操作順序◦回想一下◦要計算查詢和鍵的自註意力分數◦我們需要一個相似度函數◦對於轉換器來說它只是一個簡單的點積◦然而◦對於一般的相似性函數 $\text{sim}(q_i, k_j)$ ◦我們可以將注意力輸出表示為以下等式◦

$$(\quad)$$

$$y_i = \sum_k \frac{\text{sim}(q_i, k_j)}{\sum_k \text{sim}(q_i, k_j)} v_j$$

線性化注意力機制背後的技巧是將相似度函數表示為將操作分解為兩部分的核函數◦

$$\text{sim}(q_j, k_j) = \phi_{kj}(\text{Qi})$$

其中通常是高維特徵圖◦由於 Q 獨立於 j 和 k ◦我們可以將其拉到總和下◦將注意力輸出寫成如下所示◦

$$y_{i-} = \frac{\phi(\sum_j K_j V_j)}{\phi(\sum_k K_k)}$$

通過先計算 $\sum_j K_j V_j$ 和 $\sum_k K_k$ ，我們可以有效地線性化空間和 attention 的時間複雜度！兩種方法之間的比較如圖 11-7 所示。實現線性化自註意力的流行模型包括 Linear Transformer 和 Performer。⁸

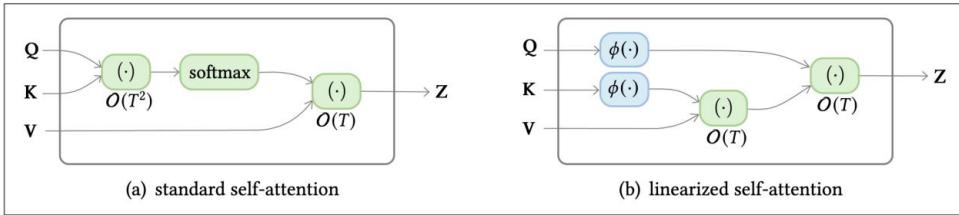


圖 11-7 標準自註意力和線性化自註意力之間的複雜性差異（林田洋提供）

在本節中，我們了解瞭如何擴展一般的 Transformer 架構和特定的注意力，以在廣泛的任務上實現更好的性能。在下一節中，我們將了解轉換器如何從 NLP 擴展到其他領域，例如音頻和計算機視覺。

超越文本

使用文本來訓練語言模型一直是 transformer 語言模型與遷移學習相結合的成功背後的驅動力。一方面，文本豐富，可以對大型模型進行自我監督訓練。另一方面，分類和問答等文本任務很常見，為它們制定有效的策略使我們能夠解決廣泛的現實世界問題。

但是，這種方法存在局限性，包括：

人為報告偏差 文本中

事件的頻率可能並不代表它們的真實頻率。⁹僅根據來自互聯網的文本進行訓練的模型可能會產生非常扭曲的世界圖像。

8 A. Katharopoulos 等人，“Transformers Are RNNs :Fast Autoregressive Transformers with Linear Attention”，(2020); K. Choromanski 等人，“重新思考表演者的注意力”，(2020)。

9 J. Gordon 和 B. Van Durme，“報告偏差和知識提取”，(2013)。

常識

常識是人類推理的基本品質，但很少被記錄下來。因此，在文本上訓練的語言模型可能知道關於世界的許多事實，但缺乏基本的常識推理。

事實

概率語言模型無法以可靠的方式存儲事實，並且可能會生成與事實不符的文本。同樣，此類模型可以檢測命名實體，但無法直接訪問有關它們的信息。

模態語言

模型無法連接到可以解決前面幾點的其他模態，例如音頻或視覺信號或表格數據。

因此，如果我們能夠解決模態限制，我們也有可能解決其他一些問題。最近在將 Transformer 推向新模式，甚至構建多模態模型方面取得了很大進展。在本節中，我們將重點介紹其中的一些進步。

想像

自從啟動深度學習革命以來，視覺一直是卷積神經網絡 (CNN) 的大本營。最近，Transformer 開始應用於該領域並實現與 CNN 相似或更好的效率。

讓我們看幾個例子。

iGPT

受成功的 GPT 系列文本模型的啟發，iGPT（圖像 GPT 的縮寫）將相同的方法應用於圖像。¹⁰ 通過將圖像視為像素序列，iGPT 使用 GPT 架構和自回歸預訓練目標來預測下一個像素值。在大型圖像數據集上進行預訓練使 iGPT 能夠“自動完成”部分圖像，如圖 11-8 所示。當將分類頭添加到模型時，它還可以在分類任務上取得高性能結果。

¹⁰ M. Chen 等人，“像素生成預訓練”，第 37 屆國際機器學習會議論文集 119 (2020) :1691–1703，<https://proceedings.mlr.press/v119/chen20s.html>。

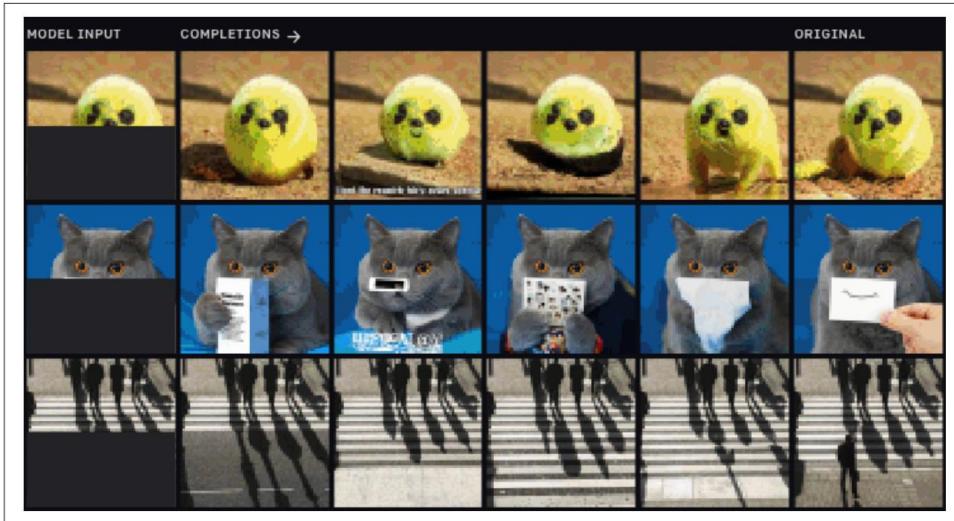


圖 11-8 ◦使用 iGPT 完成圖像的示例（由 Mark Chen 提供）

維特

我們看到 iGPT 嚴格遵循 GPT 風格的架構和預訓練過程。Vision Transformer (ViT)11 是一種 BERT 風格的視覺轉換器，如圖11-9 所示。首先，圖像被分割成更小的塊，每個塊都嵌入了線性投影。結果與 BERT 中的令牌嵌入非常相似，接下來的內容幾乎相同。補丁嵌入與位置嵌入相結合，然後通過普通的變壓器編碼器饋送。在預訓練期間，一些補丁被遮蓋或扭曲，目標是預測遮蓋補丁的平均顏色。

11 A. Dosovitskiy 等人，“一幅圖像值得 16×16 個單詞：用於大規模圖像識別的轉換器”，(2020)。

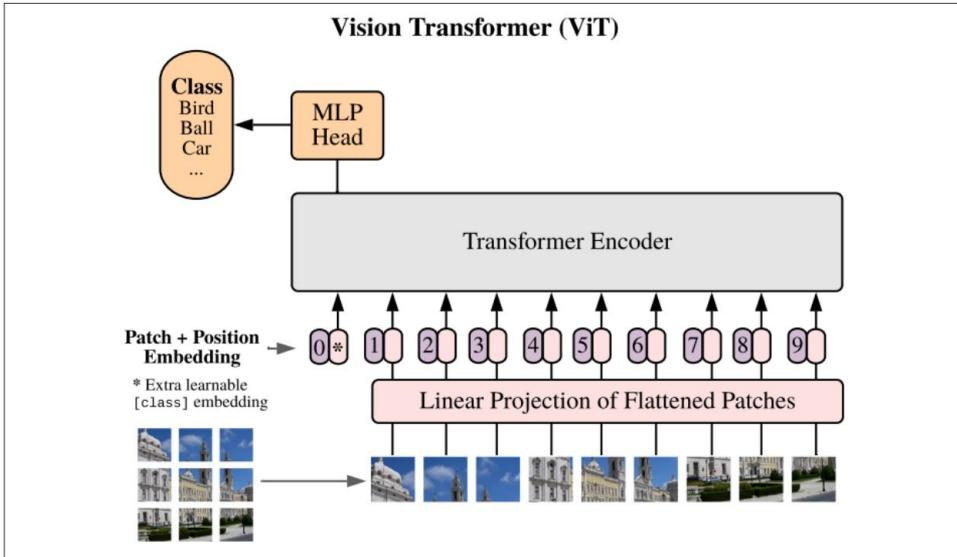


圖 11-9。ViT 架構（由 Alexey Dosovitskiy 等人提供）

雖然這種方法在標準 ImageNet 數據集上進行預訓練時並沒有產生更好的結果，但它在更大的數據集上的擴展性明顯優於 CNN。

ViT 集成在 `Transformers` 中，使用它與我們在本書中使用的 NLP 流水線非常相似。讓我們從加載一隻相當有名的狗的圖像開始：

```
從PIL import Image import
matplotlib.pyplot as plt

image = Image.open(  images/doge.jpg  )
plt.imshow(圖像) plt.axis(  關閉  ) plt.show()
```



要加载 ViT 模型，我们只需要指定图像分类管道，然后我们输入图像以提取预测类：

从变压器导入管道导入熊猫
作为pd

```
image_classifier = 管道（“图像分类”） preds = image_classifier（图  
像） preds_df = pd.DataFrame（preds） preds_df
```

	分数	标签
0	0.643599	爱斯基摩犬 · 哈士奇
1	0.207407	西伯利亚雪橇犬2
2	0.060160	野狗 · warrigal · warragal · Canis dingo
3	0.035359	Norwegian elkhound, elkhound
4	0.012927	malamute, malamute, 阿拉斯加雪橇犬

太好了，预测的类别似乎与图像相符！

图像模型的自然延伸是视频模型。除了空间维度之外，视频还具有时间维度。随着数据量变得越来越大，并且需要处理额外的维度，这使得任务更具挑战性。诸如 TimeSformer 之类的模型引入了空间和时间注意力机制来解决这两者。¹²将来，此类模型可以帮助构建用于视频序列分类或注释等广泛任务的工具。

¹² G. Bertasius · H. Wang 和 L. Torresani，“视频理解只需要时空注意力吗？”，
(2021)。

表

許多數據，例如公司內部的客戶數據，都存儲在結構化數據庫中，而不是原始文本。我們在第 7 章中看到，通過問答模型，我們可以用自然文本中的問題來查詢文本。如果我們可以對表格做同樣的事情，如圖11-10所示，那不是很好嗎？

Table				Example questions												
Rank	Name	No. of reigns	Combined days	#	Question				Answer				Example Type			
1	Lou Thesz	3	3,749	1	<i>Which wrestler had the most number of reigns?</i>				Ric Flair				Cell selection			
2	Ric Flair	8	3,103	2	<i>Average time as champion for top 2 wrestlers?</i>				AVG(3749,3103)=3426				Scalar answer			
3	Harley Race	7	1,799	3	<i>How many world champions are there with only one reign?</i>				COUNT(Dory Funk Jr., Gene Kiniski)=2				Ambiguous answer			
4	Dory Funk Jr.	1	1,563	4	<i>What is the number of reigns for Harley Race?</i>				7				Ambiguous answer			
5	Dan Severn	2	1,559	<i>Which of the following wrestlers were ranked in the bottom 3?</i>				{Dory Funk Jr., Dan Severn, Gene Kiniski}				Cell selection				
6	Gene Kiniski	1	1,131	<i>Out of these, who had more than one reign?</i>				Dan Severn				Cell selection				

圖 11-10。在桌子上回答問題（由 Jonathan Herzig 提供）

TAPAS (Table Parser 的縮寫)助您一臂之力！該模型通過將表格信息與查詢相結合，將 Transformer 架構應用於表格，如圖11-11 所示。

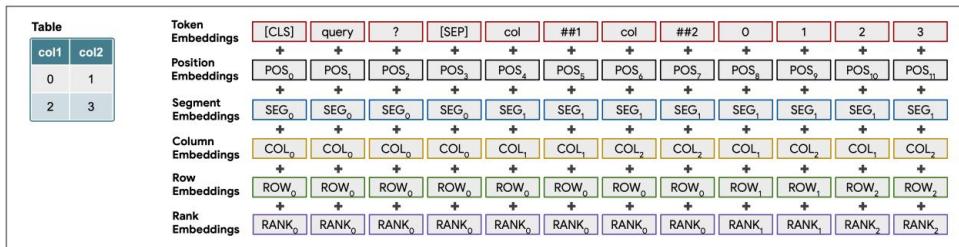


圖 11-11。TAPAS 的架構（由 Jonathan Herzig 提供）

讓我們看一個 TAPAS 在實踐中如何工作的例子。我們創建了本書目錄的虛構版本。它包含章節編號、章節名稱以及章節的起始頁和結束頁：

```
book_data =
[{"chapter": 0, "name": "簡介", "start_page": 1, "end_page": 11}, {"chapter": 1,
    "name": "文本分類", "start_page": 12, "end_page": 48}, {"chapter": 2, "name": "命名
    實體識別", "start_page": 49, "end_page": 73},
    {"chapter": 3, "name": "Question Answering", "start_page": 74},
```

13 J. Herzig 等人，“TAPAS：通過預訓練進行弱監督表解析”，(2020)。

```

        end_page :120},
{ chapter :4, name : 總結 , start_page :121, end_page :140},
{ chapter :5, name : 結論 , “起始頁” : 141, “結束頁” : 144}

]

```

我們還可以輕鬆地將每章的頁數添加到現有字段中。

為了更好地使用 TAPAS 模型，我們需要確保所有列都是str 類型：

```

table = pd.DataFrame(book_data)
table[ number_of_pages ]=table[ end_page ]-table[ start_page ]table =
table.astype(str)表

```

	章節名稱		start_page	end_page	number_of_pages
0 0	介紹	“	11	10	
1 1	文本分類	12	48歲	36	
2 2	命名實體識別 49		73	24	
3 3	問答	74	120	46	
4 4	總結	121	140	19	
5 5	結論	141	144	“	

到現在為止你應該知道演習了。我們首先加載表格問答管道：

```
table_qa = pipeline( table-question-answering )
```

然後通過一些查詢來提取答案：

```
table_qa = pipeline( table-question-answering )queries =
[ 第四章的主題是什麼？ , 總頁數是多少？ , 問答這一章從第
幾頁開始？ , “有多少章節超過20頁？” ]
```

```
preds = table_qa (表 ,查詢)
```

這些預測將表操作的類型與答案一起存儲在聚合器字段中。讓我們看看 TAPAS 在回答我們的問題時表現如何：

```
對於查詢， pred in zip(queries, preds): print(query) if
pred[ aggregator ]== NONE :
print( Predicted answer: + pred[ answer ])
else:
    “
print( 預測答案：print( = *50) “+ pred[ “答案” ])
```

第 4 章的主題是什麼？

預測答案 :總結

總頁數是多少？

預測答案 : $\text{SUM} > 10, 36, 24, 46, 19, 3$

關於問答的章節從哪一頁開始？

預測答案 :平均 > 74

有多少章超過 20 頁？

預測答案 : $\text{COUNT} > 1, 2, 3$

對於第一章，該模型準確預測了一個沒有聚集的細胞。如果我們查看表格，就會發現答案實際上是正確的。在下一個示例中，模型結合求和聚合器預測了包含頁數的所有單元格，這也是計算總頁數的正確方法。第三題的答案也是正確的；在這種情況下，平均聚合不是必需的，但它沒有什麼區別。最後，我們有一個稍微複雜一點的問題。要確定有多少章節超過 20 頁，我們首先需要找出哪些章節滿足該標準，然後對它們進行計數。看來 TAPAS 又一次做對了，正確判斷出第 1、2、3 章的頁數超過 20 頁，並且在 cells 中添加了 count aggregator。

我們提出的問題類型也可以通過一些簡單的 Pandas 命令來解決；但是，使用自然語言而不是 Python 代碼提問的能力允許更廣泛的受眾查詢數據以回答特定問題。

想像一下，業務分析師或經理手中有這樣的工具，他們能夠驗證自己對數據的假設！

多模式變形金剛

到目前為止，我們已經研究了將 Transformer 擴展到一種新的模式。TAPAS 可以說是多模式的，因為它結合了文本和表格，但表格也被視為文本。在本節中，我們將研究同時結合兩種模式的轉換器：音頻加文本和視覺加文本。

語音轉文本雖然能夠

使用文本與計算機交互是一個巨大的進步，但使用口頭語言是我們交流的一種更自然的方式。您可以在行業中看到這種趨勢，Siri 和 Alexa 等應用程序正在興起並變得越來越有用。此外，對於很大一部分人來說，寫作和閱讀比口語更具挑戰性。因此，能夠處理和理解音頻不僅方便，而且可以幫助許多人獲取更多信息。該領域的一項常見任務是自動語音識別 (ASR)，

它將口語轉換為文本，並使 Siri 等語音技術能夠回答諸如“今天天氣如何？”之類的問題。

wav2vec 2.0 模型系列是 ASR 的最新發展之一：它們將變換器層與 CNN 結合使用，如圖 11-12 所示。

¹⁴ 通過在預訓練期間利用未標記的數據，這些模型只需幾分鐘的標記數據即可獲得具有競爭力的結果。

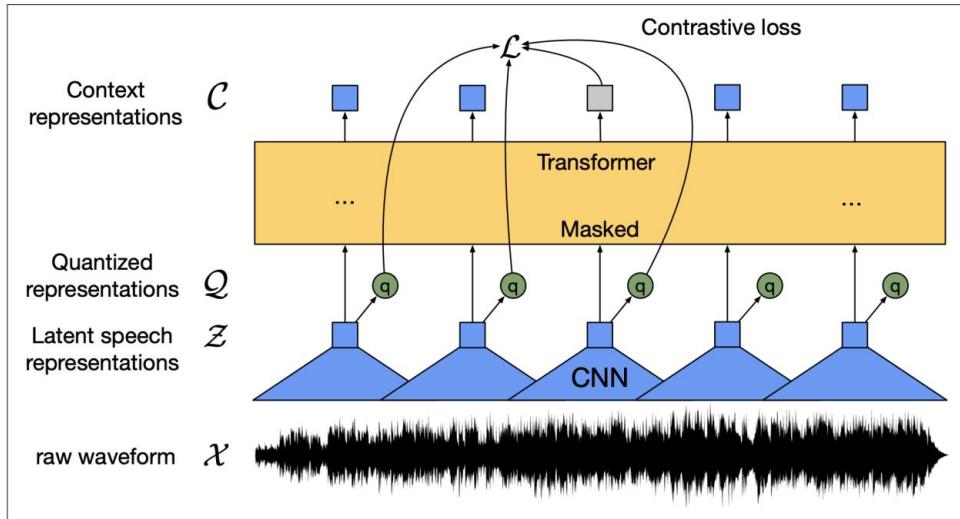


圖 11-12。wav2vec 2.0 的架構（由 Alexei Baevski 提供）

wav2vec 2.0 模型集成在 Transformers 中，你不會驚訝地發現加載和使用它們遵循的是我們在本書中看到的熟悉的步驟。讓我們加載一個經過 960 小時語音音頻訓練的預訓練模型：

asr = pipeline(自動語音識別)

為了將此模型應用於某些音頻文件，我們將使用 SUPERB 數據集的 ASR 子集，這是預訓練模型的同一數據集。由於數據集非常大，我們將只加載一個示例用於演示目的：

從數據集導入 load_dataset

```
ds = load_dataset('superb', 'asr', split='validation[:1]') print(ds[0])
```

```
{ chapter_id : 128104, speaker_id : 1272, file : '~/cache/huggingface/datasets/downloads/extracted/e4e70a454363bec1c1a8ce336139866a39442114d86a433
```

14 A. Baevski 等人，“[wav2vec 2.0 語音表示自監督學習框架](#)”，(2020)。

```
6014acd4b1ed55e55/LibriSpeech/dev-clean/1272/128104/1272-128104-0000.flac , id :  
1272-128104-0000 , text : 奎爾特先生是中產階級的使徒 ·我們是很高興迎接他的福音 }
```

在這裡我們可以看到文件列中的音頻以 FLAC 編碼格式存儲，而預期的轉錄由文本列給出。要將音頻轉換為浮點數組，我們可以使用**SoundFile 庫**使用map()讀取數據集中的每個文件：

將聲音文件導入為sf

```
def map_to_array(batch):  
    speech, _ = sf.read(batch[ file ])  
    batch[ speech ] = speech返回批  
  
ds = ds.map(map_to_array)
```

如果你使用的是 Jupyter Notebook，你可以使用以下IPython小部件輕鬆播放聲音文件：

從IPython.display導入音頻

```
顯示（音頻（ds[0][ 語音 ]，速率=16000））
```

最後，我們可以將輸入傳遞給管道並檢查預測：

```
pred = asr(ds[0][ 語音 ])打印(pred)
```

```
{ text : 奎爾特先生是中產階級的使徒，我們很高興  
歡迎他的福音 }
```

這個轉錄似乎是正確的。我們可以看到缺少一些標點符號，但這很難單獨從音頻中獲得，可以在後處理步驟中添加。

只需幾行代碼，我們就可以為自己構建一個最先進的語音轉文本應用程序！

為一種新語言構建模型仍然需要最少數量的標記數據，這可能很難獲得，特別是對於低資源語言。在 wav2vec 2.0 發布後不久，一篇描述名為 wav2vec-U 的方法的論文發表了。¹⁵在這項工作中，結合巧妙的聚類和 GAN 訓練，僅使用獨立的未標記語音和未標記的文本數據。圖 11-13 詳細顯示了這個過程。根本不需要對齊的語音和文本數據，這使得能夠為更廣泛的語言訓練高性能的語音到文本模型。

¹⁵ A. Baevski 等人，“無監督語音識別”，(2021)。

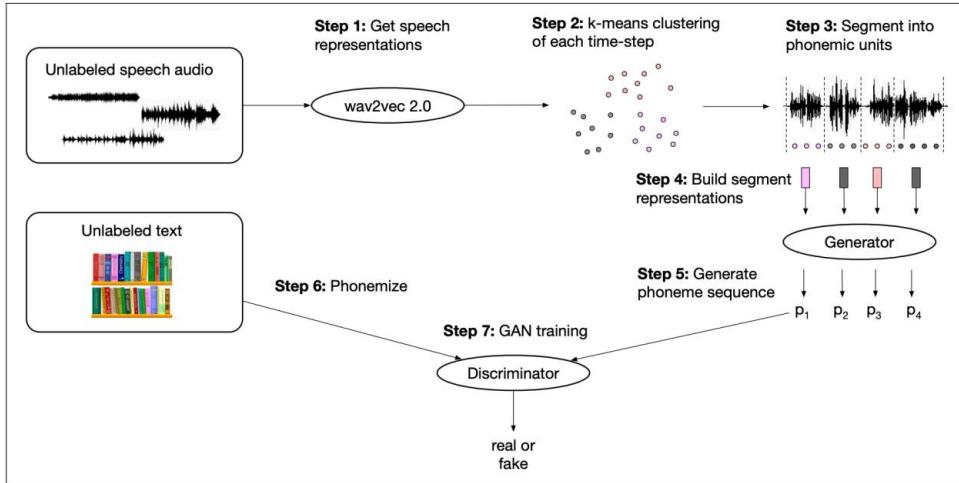


圖 11-13。 wav2vec-U 的訓練方案（由 Alexsei Baevski 提供）

太棒了，所以變形金剛現在可以“讀”文本和“聽”音頻。它們還能“看”嗎？答案是肯定的，這也是當前該領域的研究熱點之一。

視覺與文字

視覺和文本是另一種自然結合的模式，因為我們經常使用語言來交流和推理圖像和視頻的內容。除了視覺轉換器之外，在結合視覺和文本信息的方向上還有一些發展。在本節中，我們將查看四個結合視覺和文本的模型示例：VisualQA、LayoutLM、DALL-E 和 CLIP。

VQA

在第 7 章中，我們探討瞭如何使用轉換器模型來提取基於文本的問題的答案。這可以臨時完成以從文本或離線中提取信息，其中問答模型用於從一組文檔中提取結構化信息。已經有一些努力將這種方法擴展到視覺數據集，例如 VQA，¹⁶如圖 11-14 所示。

¹⁶ Y. Goyal 等人，“讓 VQA 中的 V 變得重要：提升圖像理解在視覺問題中的作用接聽”，(2016)。



圖 11-14 ◦來自 VQA 數據集的視覺問答任務示例（由 Yash Goyal 提供）

LXMERT 和 VisualBERT 等模型使用 ResNets 等視覺模型從圖片中提取特徵，然後使用變換編碼器將它們與自然問題結合起來並預測答案。 17

LayoutLM

分析掃描的商業文檔，如收據、發票或報告是另一個領域，在該領域中提取視覺和佈局信息可以成為識別感興趣的文本字段的有用方法。這裡是LayoutLM模型系列是當前最先進的。他們使用增強的 Transformer 架構，該架構接收三種模式作為輸入：文本、圖像和佈局。因此，如圖 11-15 所示，有與每種模態相關的嵌入層、空間感知的自我注意機制以及圖像和文本/圖像預訓練目標的混合以對齊不同的模態。通過對數百萬份掃描文檔進行預訓練，LayoutLM 模型能夠以類似於用於 NLP 的 BERT 的方式轉移到各種下游任務。

17 H. Tan 和 M. Bansal， “LXMERT：從變形金剛中學習跨模態編碼器表示”， (2019); LH Li 等人，“VisualBERT：視覺和語言的簡單而高效的基線”， (2019)。

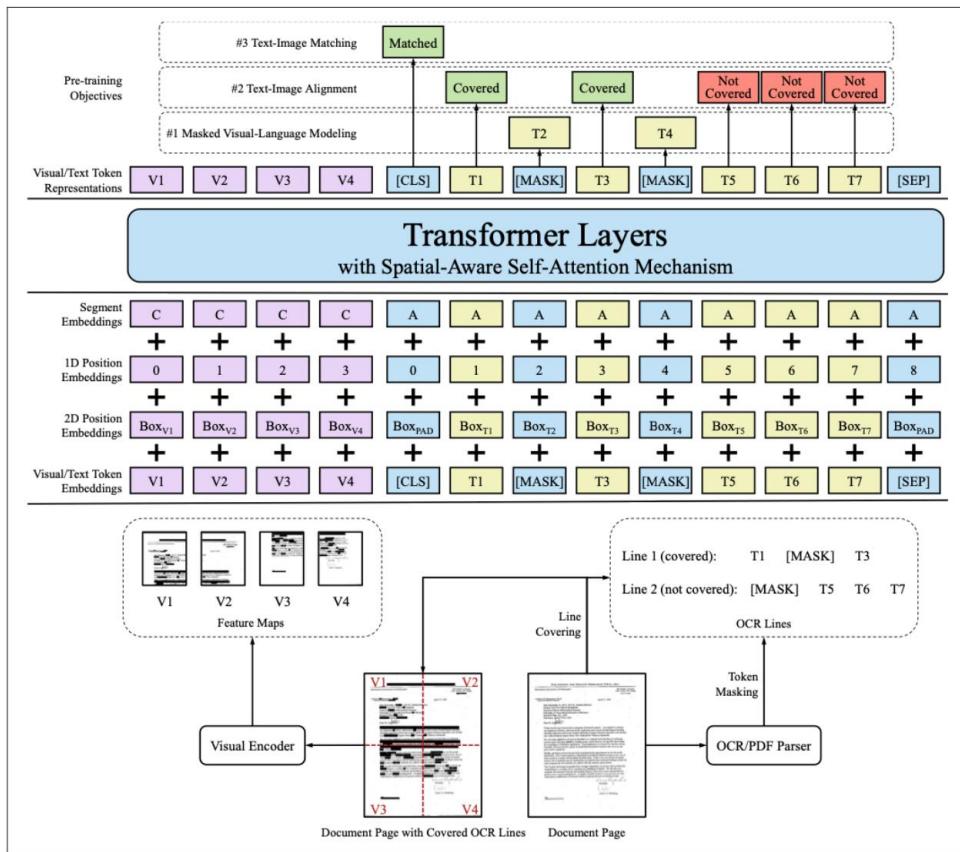


圖 11-15。LayoutLMv2 的模型架構和預訓練策略（由 Yang Xu 提供）

達爾·厄

一個結合視覺和文本進行生成任務的模型是 DALL-E.18，它使用 GPT 架構和自回歸建模從文本生成圖像。受 iGPT 的啟發，它將單詞和像素視為一個標記序列，因此能夠從文本提示中繼續生成圖像，如圖11-16 所示。

18 A. Ramesh 等人，“零樣本文本到圖像生成”，(2021)。



圖 11-16。DALL·E 的生成示例（由 Aditya Ramesh 提供）

夾子

最後，讓我們看一下 CLIP¹⁹，它也結合了文本和視覺，但專為監督任務而設計。它的創建者構建了一個包含 4 億個圖像/說明對的數據集，並使用對比學習來預訓練模型。CLIP 架構由文本和圖像編碼器（均為轉換器）組成，它們創建字幕和圖像的嵌入。對帶有說明文字的一批圖像進行採樣，對比目標是最大化相應對的嵌入的相似性（通過點積衡量），同時最小化其餘圖像的相似性，如圖 11-17 所示。

為了使用預訓練模型進行分類，可能的類別嵌入了文本編碼器，類似於我們使用零樣本流水線的方式。然後將所有類的嵌入與我們要分類的圖像嵌入進行比較，並選擇相似度最高的類。

19 A. Radford 等人，“從自然語言監督中學習可遷移視覺模型”，(2021)。

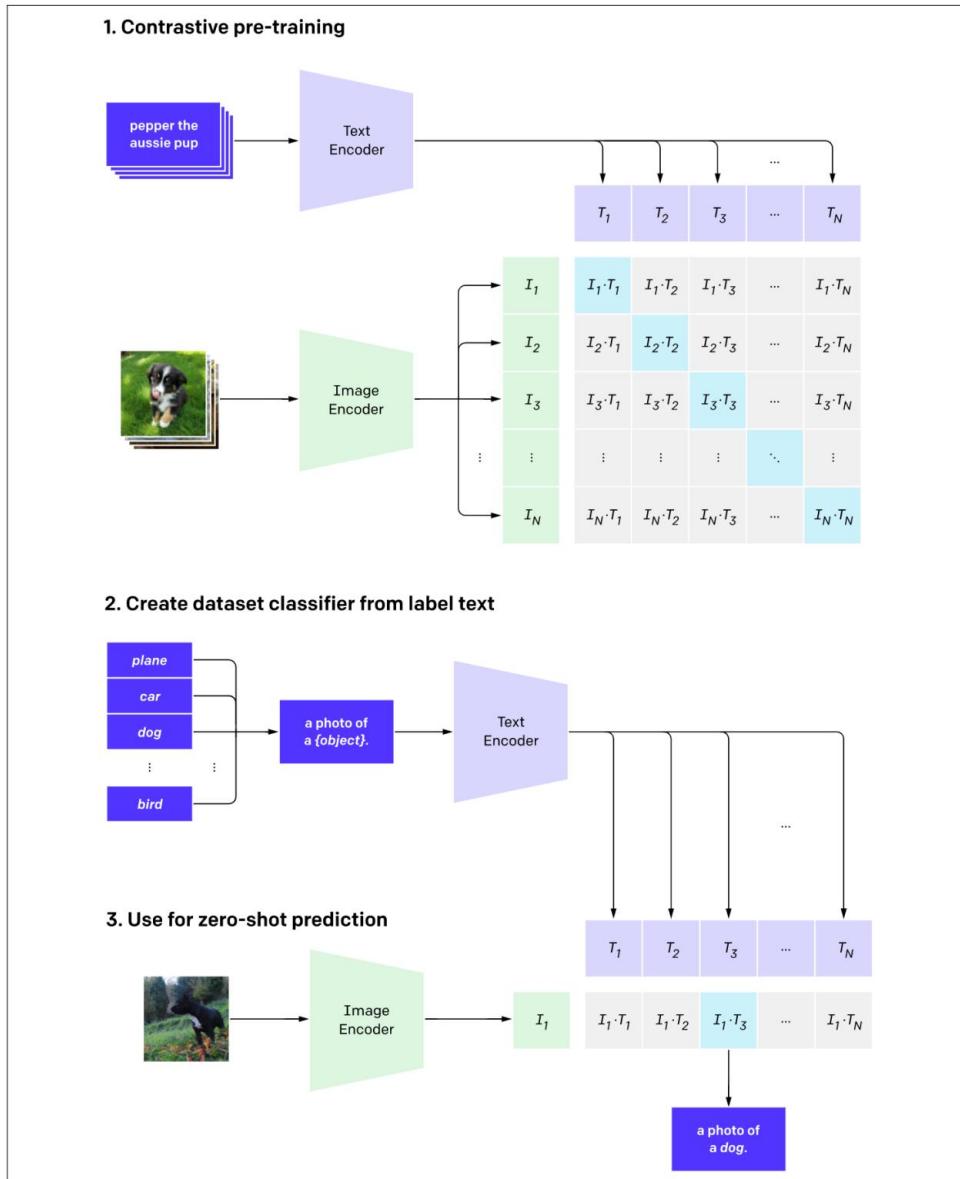


圖 11-17。CLIP 的架構（由 Alec Radford 提供）

CLIP 的零樣本圖像分類性能非常出色，與完全監督訓練的視覺模型相比具有競爭力，同時在新類別方面更加靈活。CLIP 也完全集成在 Transformers 中，所以我們可以嘗試一下。對於圖像到文本的任務，我們實例化了一個由特徵提取器和分詞器組成的處理器。特徵提取器的作用是將圖像轉換成

形式適合模型，而分詞器負責將模型的預測解碼為文本：

從變形金剛進口CLIPProcessor， CLIPModel

```
clip_ckpt = openai/clip-vit-base-patch32 模型=
CLIPModel.from_pretrained(clip_ckpt)處理器=
CLIPProcessor.from_pretrained(clip_ckpt)
```

然後我們需要一個合適的圖像來嘗試一下。還有什麼比擎天柱的照片更合適呢？

```
image = Image.open( images/optimusprime.jpg )
plt.imshow(圖像) plt.axis( 關閉 ) plt.show()
```



接下來，我們設置文本以與圖像進行比較並將其傳遞給模型：

進口手電筒

```
texts = [ “變形金剛的照片” 、 “機器人的照片” 、 “agi 的照片” ] inputs = processor(text=texts,
images=image, return_tensors= pt , padding=True) with torch.no_grad(): outputs = model(**inputs)
logits_per_image = outputs.logits_per_image probs = logits_per_image.softmax(dim=1)概率
```

張量 ([[0.9557 0.0413 0.0031]])

好吧，它幾乎得到了正確答案（當然是 AGI 的照片）。撇開玩笑不談，CLIP 允許我們通過文本定義類而不是在模型架構中硬編碼類，從而使圖像分類變得非常靈活。我們的多模態轉換器模型之旅到此結束，但我們希望我們已經激起了您的胃口。

從這裡到哪裡？

好吧，旅程到此結束；感謝您加入我們穿越變形金剛領域的旅程！在本書中，我們探索了轉換器如何處理廣泛的任務並獲得最先進的結果。在本章中，我們看到了當前一代的模型如何通過擴展被推向極限，以及它們如何擴展到新的領域和模式。

如果你想強化你在本書中學到的概念和技能，這裡有一些關於從這裡開始的想法：

加入 Hugging Face 社區活動

Hugging Face 舉辦短期衝刺，重點是改進生態系統中的庫，這些活動是結識社區和體驗開源軟件開發的好方法。到目前為止，已經在向 Datasets 添加 600 多個數據集、微調各種語言的 300 多個 ASR 模型以及在 JAX/Flax 中實施數百個項目方面進行了衝刺。



構建您自己的項目 測試您

在機器學習方面的知識的一種非常有效的方法是構建一個項目來解決您感興趣的問題。你可以重新實現一篇 transformer 論文，或者將 transformers 應用到一個新的領域。

向 Transformers 貢獻模型

如果你正在尋找更高級的東西，那麼向 Transformers 貢獻一個新發布的架構是深入了解庫的具體細節的好方法。Transformers 文檔中有詳細的指南可幫助您入門。



關於你學到的東西的博客

將您所學的知識傳授給其他人是對您自身知識的有力考驗，從某種意義上說，這是我們編寫本書的驅動動機之一！有很多很棒的工具可以幫助您開始撰寫技術博客；我們推薦快頁因為您可以輕鬆地使用 Jupyter notebooks 來處理所有事情。

指數

- 絕
對位置表示，[74](#)抽象QA，[205](#)抽象摘要，[141](#)
- 縮放點積，[62](#)自，[6](#)，[350](#)稀疏，[351](#)
- Accelerate library
about，[18](#)作為
Hugging Face 生態系統的一部分，[15](#)訓練循環的變化，[330](#)與 Trainer 的比較，[330](#)基礎設施配置，[337](#)啟動訓練工作，[337](#)
- 《Attention Is All You Need》，[xii](#)
注意力頭，[67](#)個注意力機制，[4](#)個注意力分數，[62](#)個注意力權重，[61](#)個自動類，[38](#)個
- AutoConfig 已定義，[65](#)
`from_pretrained()`，[224](#)
覆蓋默認值，[101](#)、[224](#)、[325](#)
- AutoModel
about，[38](#)
`from_pretrained()`，[38](#)
`output_attentions`，[69](#)
TensorFlow 類，[39](#)
- AutoModelFor CausalLM
`from_config()`，[325](#)
`from_pretrained()`，[127](#)、[325](#)
`gradient_checkpointing`，[333](#)
- AutoModelForMaskedLM，[291](#)
- AutoModelForQuestionAnswering，[176](#)
- AutoModelForSeq2SeqLM，[156](#)
- AutoModelForSequenceClassification
about，[46](#) `from_pretrained()`，[46](#)
- TensorFlow 類，[50](#)自回歸注意力，[59](#)自回歸語言模型，[126](#)
- 自動分詞器
`add_special_tokens`，[311](#)
`as_target_tokenizer ()`，[159](#)

backend_tokenizer.normalizer, 314
 backend_tokenizer.pre_tokenizer, 314
 convert_ids_to_tokens(), 290
 convert_tokens_to_string(), 34 decode(), 105, 127, 175 from_pretrained(), 33 從緩存加載, 33 填充, 35, 161 push_to_hub(), 322
 return_special_tokens_mask, 290
 return_tensors, 154 截斷, 35 vocab_size, 34

B

反向翻譯, 272
 balanced_split() 函數, 258
 BALD (基於分歧的貝葉斯主動學習), 296 band attention, 352

BART模型, 84, 145 基線
 匯總, 143 波束搜索解碼, 130-134
 波束, 130

BERT模型, 1, 9, 79, 211, 217, 220, 224, 237, 260, 263, ---
 BertViz 庫, 63 偏差, 19, 301 雙向注意力, 59
 大鳥模型, 84, 352
 大查詢, 306
 大科學, 350
 BLEU 分數, 148-152 體 (神經網絡), 98
 玻爾茲曼分佈, 135
 BookCorpus 數據集, 9, 80, 301
 BPE (Byte-Pair Encoding), 94, 312, 316 字級, 314

C

C4 數據集, 83, 301, 310
 CamemBERT tokenizer, 310 因果注意力, 59 因果語言建模, 126, 323

CCMatrix 語料庫, 284 個
 字符分詞, 29
 Chaudhary, Amit, 272 類分佈, 27 分類頭, 75

分類器, 微調, 47, 293
 類標籤
 關於, 24
 int2str(), 26, 91
 個名字, 101
 str2int(), 214
 CLINC150 數據集, 213
 CLIP模型, 367 閉域
 QA, 168
 [CLS] token
 about, 34
 excluding from tokenizer, 65 作用於問答, 179 作用於文本分類, 37
 special token ID, 35

CNN (卷積神經網絡), 355
 CNN/DailyMail 數據集, 141, 154
 CodeParrot 模型, 299, 337, 342
 CodeSearchNet 數據集, 304
 Colab 筆記本, 18, 20
 Common Crawl 語料庫, 80, 93 常識限制、文本和、355 社區 QA、166 compile() 方法、221 compute() 函數、150 compute_accuracy() 方法、214, 241 compute_loss() 方法、221 compute_metrics() 函數、47, 108, 222 compute_size() 函數, 215, 241 concatenate_datasets() 函數, 118 條件文本生成, 126

CoNLL 數據集, 92
 個常量折疊, 238 個上下文, 12 個上下文管理器, 160 個上下文大小, 28, 34, 84, 321 上下文嵌入, 61 convert_graph_to_onnx.convert() 函數, 239 convert_ids_to_tokens() 方法, 34 convert_tokens_to_string() 方法, 34 語料庫, 9, 80, 92, 284, 300-303, 310 成本, 作為縮放的挑戰, 349 覆蓋指標, 312 交叉熵損失, 104, 219, 295, 347 cross-lingual transfer about, 115 fine-tuning multiple languages simultaneously, 118-120

零樣本遷移， 116
CSV 數據集， 25
CTRL 模型， 82
CUAD 數據集， 169 個自定義數據集， 25 個自定義模型， 99-102 ·101 截止值、 138

丁

DALL-E 模型， 366 數據增強， 271 可用性 ·作為變壓器的挑戰， 19 矛域， 168 交換格式， 26 數據整理器， 36 ·107 ·160 ·289 數據並行性， 330

DataFrame ·數據集對象來自， 26,109,258 數據加載器 ·實施， 326-329 數據集（對象）更改 輸出格式， 26 ·29 ·40 創建 FAISS 索引， 277 DataFrame 轉換為， 26 ·109 ·258 個特徵， 23 個 flatten()， 168 個用 map() 函數處理數據， 35 個， 51 個， 103-105 個 select()， 90 個 shuffle()， 90 個 數據集卡片， 16 個， 310 個 數據集添加到 Hugging Face Hub， 309 add_faiss_index() 函數， 277 add_faiss_index_from_external_arrays() 函數， 277 BookCorpus, 9, 80, 301 建築 自定義代碼， 303-306 C4 ·83 ·301 ·310 --- 臨床 150, 213 美國有線電視新聞網/每日郵報， 141, 154 代碼鵝鵠， 299, 337, 342 代碼搜索網， 304 常見爬行， 80 CoNLL， 92 使用 Google BigQuery 創建， 304 計算機文件， 25 CUAD， 169 策展 ·作為擴展的挑戰， 349

定制， 25
情感， 23 ·用於構建基於審查的 QA 系統， 167-172

用於多語言命名實體識別， 88-92 GitHub， 252-257 膠水， 23 圖像網， 7 JSON， 25 大， 300-310 加載各種格式， 25 265 _ 網管， 172 奧斯卡， 310 PAN-X, 88 怪癖， 51 薩姆森， 157 小隊， 23, 171 SubjQA， 167-172 棒極了， 362 SuperGLUE, 81, 83 文本， 25 整體分詞， 35

質量保證， 364 維基網絡， 88 歲 維基百科， 80 極限， 88

關於數據集庫， 18 作為 Hugging Face 生態系統的一部分， 16 檢查所有數據集配置， 88 檢查元數據， 23 在集線器上列出數據集， 23 從集線器加載數據集， 23 加載本地數據集， 25 從集線器加載指標， 150, 153 ·214 加載遠程數據集、 25

戴維森 ·喬， 263 DDP (DataDistributedParallelism)， 336 DeBERTa 模型、 81 解碼器、 58 ·76 解碼器分支、 82 解碼器層、 58 解碼器-only 模型、 59 解碼 beam search 解碼、 130-134 貪心搜索、 127-130 解碼方法、 125 ·140

深度神經網絡, 244 deepset, xxi, 182, 187 deployment, as a challenge of scaling, 349對話 (conversation), 141, 157對話摘要, 生成, 162 dilated attention, 352鑑別器, 81

DistilBERT 模型, 22、28、33、36、80文檔長度 作為轉換器的挑戰, 19與Haystack 檢索器的文檔存儲兼容性, 182定義, 182使用Elasticsearch 初始化, 183加載文檔, 185加載標籤, 191領域

adaptation, 8, 199-203, 289
of data, 168 domain
adaptation, 8, 199-203, 289點積, 62-67, 77, 353, 367 downsample, 90, 116

DPR (密集通道檢索), 194動態量化, 235

E

效率, 209-247關於, 209基準量化模型, 236創建性能基準, 212-217意圖檢測, 210知識蒸餾, 217-230使用ONNX/ONNX Runtime優化推理, 237-243量化, 230-236權重剪枝, 243-247 Elasticsearch, 183, 186ElasticsearchRetriever.eval() 方法, 190 ELECTRA 模型, 81 EleutherAI, 83, 350 ELMO 模型, 8, 61EM (精確匹配)分數, 196嵌入上下文, 61密集, 65distilBERT, 37位置, 73令牌, 57, 58用作查找表, 275-282

字, 十二, 8
情感數據集, 23
encoder about, 60
adding
classification heads, 75 adding
layer normalization, 71
defined, 57 feed-forward layer, 70 positional embeddings, 73
self-attention, 61-70 encoder branch, 79-82 encoder layers, 58 encoder-解碼器注意力, 76編碼器解碼器分支, 83編碼器 解碼器模型, 2、59僅編碼器模型, 59端到端, 37、45、181、189、193、205

英文維基百科數據集, 80
EOS (end-of-sequence) token, 58錯誤分析, 50-53, 108-115指數, 231從文本中提取答案, 173-181最後的隱藏狀態, 39

提取 QA, 13, 166提取摘要, 141

F

F.log_softmax() 函數, 221 F1-score(s), 48, 105, 120, 150, 196, 260, 285事實限制, 文本和, 355 FAISS 文檔存儲, 183, 196高效相似性搜索, 282索引, 添加到一個Dataset對象, 277庫, 196, 282 family tree, of transformers, 78
FARM library reader for question answering, 187 training models with, 199-203
FARMReader

about, 187
與pipeline()函數的比較, 187
加載模型, 187
predict_on_texts(), 188
train(), 199, 202
Fast Forward QA 系列, 207
fastdoc 庫, xxi

fastpages, 370
 特徵提取器, 38-45, 368特徵矩陣,
 創建, 41前饋層, 70小樣本學習, 288

FF NNs (前饋神經網絡), 6過濾噪聲, 306微
 調作為 ULMFIT 過程的一個步驟, 8 個分類器,
 47 293知識蒸餾, 217語言模型, 289-292多種
 語言同時, 118-120

PEGASUS, 158-162
 變形金剛, 45-54香草
 變形金剛, 284帶 Keras,
 50
 XLM-RoBERTa, 106-115
 fit()方法, 50定點數, 231
 flatten ()方法, 168浮點數,
 231 forward()函數, 99 100框
 架, 互操作性, 39 from_config()
 方法, 325 from_pandas() 方
 法, 258 from_pretrained() 方法, 33, 38, 101,
 224,

325
 融合操作, 238

G
 generate ()函數, 127 133 135 138 156生成
 QA, 205生成任務, 366

Geron、Aurelien、xvi
 getsizeof() 函數、235
 get_all_labels_aggregated() 方法、192
 get_dataset_config_names() 函數、88 168
 get_dummies() 函數、30 get_nearest_examples()
 函數、277 get_nearest_examples_batch () 函數、
 278 get_preds() 函數、267

GitHub
 構建問題標記器, 251-259
 許可證 API、304存
 儲庫、252 300網站、
 251
 GitHub 副駕駛, 299, 303
 GitHub REST API, 252, 304

全球關注, 352
 膠水數據集, 23, 79
 谷歌合作實驗室 (Colab), xviii, 20
 谷歌搜索, 166
 谷歌的米娜, 124
 GPT模型, 1 9 82 302 _
 GPT- 2型號, 82 123 129 144 146 276 302 、
 313 321 330 _
 GPT-3 模型, 83, 276, 346
 GPT-J 型號, 83, 350
 GPT - Neo模型, 83 350梯
 度積累, 161 335梯度檢查點, 335
 貪心搜索編碼, 127-130

Grid Dynamics, 207
 ground truth, 132, 147, 160, 190, 196, 214, 217, 223,
 240
 Gugger, Sylvain, xvi

H

硬件要求, xviii哈希符號 (#), 12

Haystack 圖書館構
 建 QA 管道使用, 181-189評估閱讀器,
 196評估檢索器, 189-196評估整個管道,
 203初始化文檔存儲, 183初始化管道, 188
 初始化閱讀器, 187初始化檢索器, 185檢
 索器-閱讀器架構, 181教程, 196 208個網
 站、182個頭 (神經網絡)、98 個
 head_view() 函數、69個隱藏狀態、2 37

欣頓、傑夫, 217
 “我們如何擴展 Bert 以每天為超過 10 億人提供服務
 對 CPU 的請求”, 209
 Howard, Jeremy, xvi
 the Hub (參見 Hugging Face Hub)
 擁抱的臉
 加速圖書館, 18 個社區
 活動, 370
 數據集庫, 18生態系
 統, 15
 分詞器庫, 17
 擁抱面數據集, 23

Hugging Face Hub
 關於， 16
 添加數據集， 309
 選擇問題答模型， 168
 列出數據集， 23
 登錄， 47
 保存自定義分詞器， 322
 保存模型， 53
 小部件， 121

抱臉變形金剛， 9 版（另見變形金剛）

The Hugging Face Course， xvi
 人類報告偏差限制，文本和， 354 個超參數，使用 Optuna 查找， 226 hyperparameter_search() 方法， 228

我iGPT 型號， 355
 ImageNet 數據集， 7
 不平衡學習庫， 28
 IMDb，
 8情境學習， 288
 推理 API， 54、349 推理
 小部件， 16
 InferKit, 124
 信息瓶頸， 4基礎設施，作為擴展的挑戰， 348
 初始化文檔存儲， 183
 模型， 325
 讀者， 187
 檢索器， 185
 init_weights() 方法， 100
 int2str() 方法， 26
 意圖檢測， 210
 中間表示， 237
 互操作性，框架之間， 39

ISO 639-1 語言代碼， 89
 問題選項卡， 251
 Issues Tagger, building, 251-259
 iter() 函數， 328
 iterative_train_test_split() 函數， 258, 259

傑
 JAX 庫， 10
 吉拉， 251
 JSON 數據集， 25
 Jupyter 筆記本， 47, 300, 363

鉀
 Kaggle 筆記本，十八
 Karpathy, 安德烈, 3, 27, 77
 Keras庫， 50、221
 核函數， 353
 鍵， 62
 鍵/值對， 215
 風箏， 299
 KL (Kullback-Leibler) 散度， 218
 關於知識蒸餾， 217
 對模型進行基準測試， 229
 選擇學生初始化， 222-226
 創建訓練器， 220
 使用 Optuna 尋找超參數， 226
 用於微調， 217
 用於預訓練， 220

L
 標籤， 249-296
 關於， 249
 構建 GitHub Issues 標註器， 251-259
 不正確， 51
 利用未標記數據， 289-296
 使用少數數據， 271-289
 使用無標記數據， 263-271

語言模型
 微調， 289-292
 語言，作為變換器的挑戰， 19
 最後隱藏狀態， 3
 39延遲，作為性能基準， 212
 層歸一化， 71
 LayoutLM 模型， 365
 LCS（最長公共子串）， 153
 學習率預熱， 71
 Libraries.io， 304
 線性化注意力， 353
 list_datasets() 函數， 23
 加載自定義數據集， 25
 自定義模型， 101
 預訓練模型， 46
 tokenizer， 33
 load_dataset() 函數下載配置， 306
 加載單個配置， 88, 168
 加載特定版本， 141
 流式傳輸， 308
 日誌概率， 131

羅吉特, 75, 102, 125, 127, 131, 134, 176-178, 187, 218, 221, 240, 286
 長格式質量檢查, 166
 Longformer 模型, 352
 表 · 使用嵌入作為, 275-282
 LSTM (長短期記憶) 網絡, 1
 盧塞恩, 186
 LXMERT 模型, 365

米

M2M100 型號, 84, 272, 284
 MAD-X庫, 121
 量級剪枝,
 245尾數, 231 mAP (平均精度), 190 map ()方法, 35、
 40、51、103、260、267、273掩碼矩陣, 76、244
 掩碼多頭自體-attention, 76
 矩陣, 66, 232, 244
 最大內容大小, 28
 mean pooling, 276

Meena (Google), 124

內存
 映射, 18, 306
 內存, 作為性能基準, 212
 指標

準確性, 47、163、214
 add()函數, 150
 add_batch()函數, 150
 BLEU, 148-152
 計算(), 150
 完全匹配, 196
 F1-score, 48, 105, 120, 150, 197, 260, 285
 log probability, 130
 mean average precision, 189

困惑, 333

精度, 105, 148

召回, 105, 150, 152, 189

胭脂, 152

SacreBLEU, 150

鐘 GPT 模型, 77

MiniLM 模型, 174

MLM (掩碼語言建模), 9, 80, 324

MNLI 數據集, 265 個

態限制 · 文本和, 355 個
 模型卡片, 16 個

型中心, xi
 模型權重, 16 個
 模型小部件 · 交互, 121 個

艾伯特, 81 歲, 174 歲
 巴特, 84, 145
 伯特, 1, 9, 79, 211, 217, 220, 224, 237, 260, 263

大鳥, 84, 352
 CamemBERT, 310
 剪輯, 367
 代碼鸚鵡, 299, 337, 342
 控制, 82

達爾-E, 366
 黛博塔, 81 歲
 DistilBERT, 22, 28, 33, 36, 80
 民主共和國, 194
 伊萊克特拉, 81 歲
 ELMO, 8, 61
 評估 · 作為縮放的挑戰, 349
 GPT, 1, 9, 82, 302
 GPT - 2、82、276、302、313、321、330
 GPT - 3、83、276、346
 GPT-J, 83, 350
 GPT-Neo, 83, 350
 iGPT, 355
 初始化, 325

佈局LM, 365
 長篇小說, 352
 長短期記憶網絡 —
 立信科技, 365
 M2M100, 84, 272, 284
 米娜, 124 分
 鐘 GPT, 77
 你 LM, 174
 樸素貝葉斯, 260-263
 PEGASUS, 145、154、158、158-162
 性能, 作為性能基準, 212

抹布, 205
 改革者, 353
 ResNet, 6, 365
 循環神經網絡, 2
 RoBERTa, 80, 174
 儲蓄, 53
 分享, 53

T5、83、144、310
 TAPAS, 166, 359
 訓練, 47種, 221

超低頻濾波器, 1, 8
 視覺伯特, 365
 維特, 356

Wav2Vec2 , 362
 XLM , 80
 XLM-RoBERTa , 39 , 80 , 93 , 106-115 , 174
 model_init()方法 , 107運動剪枝、 246多頭注意力、 67多標籤文本分類問題、 251多語言命名實體識別、 87-121關於、 87解剖Transformers 模型類 , 98主體 , 98創建用於標記分類的自定義模型 , 99-102跨語言傳輸 , 115-120數據集 , 88-92錯誤分析 , 108-115同時微調多種語言 , 118 -120微調XLM-RoBERTa , 106-115頭 , 98與模型小部件交互 , 121加載自定義模型 , 101-102多語言轉換器 , 92性能測量 , 105

SentencePiece 分詞器 , 95分詞器 , 93-96分詞器管道 , NER 的94分詞文本 , 103-105轉換器 , 96

XLM-RoBERTa , 93
 個零樣本遷移 , 116 個多語言轉換器 , 92 個多模態轉換器 , 361-364

N

個n-gram懲罰 , 133個n-gram , 152
 樸素基線 實施 , 260-263
 樸素貝葉斯分類器 , 260 個命名實體 , 11
 NER (命名實體識別)對齊預測 , 105
 作為轉換器應用程序 , 11 (另見多語言命名實體識別)任務 , 92 、 108 、 115標記文本 , 103-105轉換器 , 96神經網絡架構 , xii , 1, 4

神經網絡 Block Movement Pruning 庫 , 247下一個標記概率 , 133

NLI (自然語言推理) , 265-271
 NLP (自然語言處理) , 遷移學習 , 6-10

NlpAug 庫 , 272
 NLU (自然語言理解) , 79噪聲 , 過濾 , 306非局部關鍵字 , 321歸一化 , 71 , 94 notebook_login()函數 , 309

NQ數據集 , 172
 NSP (下一句預測) , 80核採樣 , 136-139數值化 , 29

O

objective() 函數 , 227偏移跟踪 , 314 one-hot 編碼 , 30 , 65 one-hot 向量 , 30, 37
 one_hot() 函數 , 30

ONNX-ML , 237

ONNX/ONNX 運行時 , 優化推論 , 237-243不透明度 , 作為變壓器的挑戰 , 19開源 , 251 , 304 , 312 、 350 、 370開放域 QA , 168

開放人工智能 , 8, 82, 123, 129, 276, 349

OpenMP , 239
 個運算符集 , 240 個
 “最佳腦外科醫生”論文 , 245
 Optuna 寻找超參數 , 226
 ORT (ONNX 運行時) , 242
 OSCAR 語料庫 , 310
 個範圍外查詢 , 210

P

PAN-X 數據集 , 88 , 114
 pandas.Series.explode() 函數 , 110
 Paperspace 梯度筆記本 , xvii部分假設 , 130 Path.stat () 函數 , 215 PEGASUS 模型 , 145在 CNN/DailyMail 數據集上進行評估 , 154評估在 SAMSum 上 , 158微調 , 158-162

性能創建基

準， 212-217定義指標， 47度
量， 105與規模的關係， 347

`perf_counter()` 函數、 215置換
等變、 72 `pip` 、 xii使用
`Haystack` 構建管道、 181-189
分詞器、 94 、 312

`Transformers` 庫， 10 個
`pipeline()` 函數
 `aggregation_strategy`，
 10 個定義， 10 個命名實體
 識別， 10 個問題回答， 12 個摘要， 13 個文本分類， 11 個文本生成， 14 個翻譯， 13 個使用來自 Hub 的模型， 13 個
`plot_metrics()` 函數， 229 池化， 276

`Popen()`函數， 183

`position-wise feed-forward layer`，
70 `positional embeddings`， 58 、 73
`post layer normalization`， 71
`postprocessing`， 95 `pre layer normalization`， 71
predict()方法， 48 、 115 `prepare()`函數， 330
`pretokenization`， 94 個預訓練模型，
38 個， 46 個關於預訓練， 7 個作為
ULMFiT 過程的一個步驟， 8 個知識蒸餾， 220 個目標， 323 個提示， 288 個
比例連續詞， 312 個偽標籤， 296
個`push_to_hub()`方法， 53， 322

Python 分詞器， 313-318
關於 PyTorch 庫， 10 類和方法， 64 集線器， 17 互操作性， 39

`tril()` 函數， 76 網站， xvi

問

QA（問答）， 165-207關於， 165 抽象， 205 作為轉換器應用程序， 12 使用 `Haystack` 構建管道， 181-189 構建基於審查的系統， 166-189 封閉域， 168 社區， 166 數據集， 167-172 領域適應， 199-203 評估讀者， 196-199 評估檢索器， 189-196 評估整個管道， 203 從文本中提取答案， 173-181 提取 、 13 、 166 生成， 205 改進管道， 189-199 中的長段落， 179 長格式， 166 開放域， 168

RAG (retrieval-augmented generation) ， 205 跨度分類任務， 173 SQuAD 數據集， 171 表 QA， 359 標記化文本， 175-178 質量， 生成文本， 148-154 量化， 230-235 基準模型， 236 動態， 235 量化感知訓練， 236 靜態， 235 策略， 235 `quantize_dynamic()` 函數， 236， 242 `quantize_per_tensor()` 函數， 233 查詢， 62 問答對， 166， 191， 197， 199 問題-上下文對， 179

R

小數點， 231 RAG (檢索增強生成) ， 205 RAG-序列模型， 205 RAG-Token 模型， 206 隨機注意力， 352

閱讀器作
為檢索器-閱讀器架構的組成部分，[181](#)評估，[196-199](#)初始化，[187](#)閱讀理解模型，[166](#)

README 卡，[310](#)
recall，[189](#) recv
keyword，[320](#)
Reformer 模型，[353](#) 個
相對位置表示，[74](#)
ResNet 模型、[6](#)-[365](#)
retrieve ()方法、[186](#)
retriever
作為 retriever-reader 架構的一個組件-確實，[181](#)
評估，[189-196](#) 初始化，[185](#) 檢索器閱讀器
架構，[181](#) 基於審查的 QA 系統 架構
建，[166-189](#)
RNN（循環神經網絡），[2](#)
RoBERTa 模型，[80](#), [174](#)
ROUGE 分數，[152](#)
run() 方法，[188](#)、[190](#)
run_benchmark() 方法，[213](#)
Rust 編程語言，[17](#), [314](#)

小號
SacreBLEU，[150](#) 採樣效率，[348](#) sample() 方法，[169](#) 採樣方法，[134-139](#)

SAMSum 數據集，[157](#)
三星，[157](#) 在
Hugging Face
Hub 上保存自定義分詞器，[322](#)
模型，[53](#)縮放點積注意力，[62-67](#)縮放定律，[347](#)縮放變換器，[345](#)挑戰，[348](#)線性化注意力，[353](#)縮放定律，[347](#)自註意力機制，[350](#)稀疏注意力，[351](#)

Scikit-learn 格式，[41](#)
Scikit 多學習庫，[257](#)

select() 方法，[90](#) self-attention，[6](#), [350](#) self-attention layer about，[61](#) multi-headed attention，[67-70](#) scaled dot-product attention，[62-67](#)
SentencePiece 分詞器，[93](#)、[95](#)情緒分析，[10](#) sent_tokenize() 函數，[146](#)
[SEP] token，[34](#), [35](#), [65](#), [70](#), [94](#), [95](#), [176](#), [180](#), [290](#)
seq2seq (sequence-to-sequence)，[3](#), [324](#) seqeval library，[105](#)
序列類，[90](#)個 setup_logging()方法，[331](#)個 set_format()方法，[26](#)個共享模型，[53](#)個 shuffle()方法，[90](#)個，[308](#)個符號，[231](#)個有效數字，[231](#)個銀標，[114](#)個相似函數，[62](#)個 skip connections，[71](#)個平滑羈律，[347](#) softmax，[62](#), [66](#), [77](#),[81](#), [125](#), [127](#), [134](#), [178](#), [187](#),[218](#), [221](#)軟件要求，[xviii](#)

SoundFile 庫，[363](#)跨度分類任務，[173](#)稀疏注意力，[縮放和，\[351\]\(#\)語音到文本，\[361-364\]\(#\)加速，\[284\]\(#\) split\(\) 函數，\[31\]\(#\)](#)

SQuAD (斯坦福問答數據套)，[23](#), [171](#), [198](#), [202](#)
Stack Overflow，[166](#), [212](#)
state of the art，[209](#)靜態量化，[235](#) str2int() 方法，[214](#)流數據集，[308](#)主觀性，[167](#)

SubjQA 數據集，[167-172](#)
sublayer，[60](#), [70](#), [76](#)
subword fertility，[312](#)
subword tokenization，[33](#)
summarization，[141-163](#)
about，[141](#) abstractive summaries，[141](#) as a transformer application，[13](#)
baseline，[143](#)

CNN/DailyMail 數據集， 141
 比較摘要， 146在 CNN/
 DailyMail 數據集上評估 PEGASUS， 154提
 取摘要， 141生成對話摘要， 162測量生
 成文本的質量， 148-154

文本摘要管道， 143-146訓練模型，
 157-163
 SUPERB 數據集， 362
 SuperGLUE 數據集, 81, 83
 薩頓 ,理查德， 345

T T5 模型, 83, 144, 310
 Table QA, 359 TabNine,
 299 TAPAS 模型, 166,
 359任務無關蒸餾, 223
 Tensor.masked_fill() 函數, 76
 Tensor.storage() 函數, 235
 TensorBoard, 331 TensorFlow
 about, 10類和方法， 64 個使用
 Keras API 的微調模型， 50 個中
 心， 17 個網站， xvi

張量
 批量矩陣-矩陣乘積， 66轉換為
 TensorFlow， 50創建單熱編碼， 30
 用掩碼填充元素， 76 的整數表示，
 233量化， 231在分詞器中返回， 39存
 儲大小， 235 test_step() 方法， 221

文本
 從中提取答案， 173-181超越，
 354-370用於 QA 的標記化， 175-178
 視覺和， 364-370文本分類， 21-55
 關於， 21作為轉換器應用程序， 10
 字符標記化， 29類分佈， 27

DataFrames， 26
 個數據集， 22-25
 數據集庫， 22-25 個微
 調轉換器， 45-54 個推文長度，
 28個子詞標記化， 33 個標記化整
 個數據集， 35 個訓練文本分類
 器， 36-45 個作為特徵提取器的轉
 換器， 38-45 個單詞標記化， 31
 個文本數據集， 25 text entailment, 265
 text generation, 123-140 about, 123 as a
 transformer application, 14 beam search
 decoding, 130-134 challenges with, 125
 choosing decoding methods, 140 greedy
 search encoding, 127 sampling methods,
 134-139 text summarization pipelines，
 143-146

TextAttack 庫， 272
 TF-IDF (詞頻-逆文檔
 頻率)算法， 185
 TimeSformer 模型， 358
 時間步， 4-61、76、127、130、134
 time_pipeline()函數， 215
 TLM (翻譯語言建模)， 80 個標記分類，創
 建自定義模型， 99-102 個標記嵌入， 58 個，
 61 個標記擾動， 272 個標記化， 29 個，
 93 個字符， 29 個整個數據集， 35 個子詞， 33
 個QA 文本， NER 的175-178 個文本， 103-105
 個單詞， 31 個分詞器模型， 95、312 個分詞器管
 道， 94 個分詞器， 12 個建築， 310-321個
 Python， 313-318測量性能， 312在
 Hugging Face Hub 上保存， 322 個訓練，
 318-321

Tokenizers 庫， 17
 個作為 Hugging
 Face 生態系統的一部分， 15 個自動類，
 38 個從 Hub 加載的標記器， 38 個標記化
 文本， 38 個top-k 採樣， 136-139 top-p
 採樣， 136-139 torch.bmm() 函數， 66
 torch.save() 函數， 214, 241 torch.tril() 函數，
 76 to_tf_dataset() 方法， 50 train() 方法， 199,
 223 Trainer 關於， xviii 計算自定義損失， 221 創
 建自定義Trainer， 221 定義指標， 47, 107, 223,
 286 微調模型， 48 生成預測， 48
 hyperparameter_search()， 228 知識蒸餾， 220
 記錄歷史， 291 model_init()， 107
 push_to_hub()， 53 使用數據整理器， 107,
 160, 290 個訓練模型， 47 個摘要模型，
 157-163 個文本分類器， 36 個分詞器，
 318-321 個訓練循環， 定義， 330-337 個訓練
 運行， 337 個訓練集， 42, 257 個訓練切片， 從
 頭開始創建， 259 個訓練轉換器， 299-343 關於，
 299 將數據集添加到 Hugging Face
 Hub， 309 構建自定義代碼數據集，
 303-306 構建分詞器， 310-321 cha 構建大
 規模語料庫的挑戰，

300-303

定義訓練循環， 330-337 實施
 Dataloader， 326-329 初始化模型，
 325 個大型數據集， 300-310 測量分詞
 器性能， 312 個預訓練目標， 323 個結
 果和分析， 338-343

在 Hugging Face 上保存自定義分詞器
 Hub，
 322 Python 分詞器， 313-318 分
 詞器模型， 312 訓練運行， 337 訓
 練分詞器， 318-321

關於

TrainingArguments，
 47 創建自定義 TrainingArguments， 220 梯度累
 積， 161 label_names， 222 save_steps， 106
 train_new_from_iterator() 方法， 318
 train_on_subset() 函數， 119 train_step() 方
 法， 221

TransCoder 模型， 304 邊

移學習與監督學習比較， 6
 計算機視覺， 6NLP， 6-10 權重剪枝， 243

transformer applications

about， 10 命名實體識
 別， 11 問答， 12 摘要， 13 文本分類，
 10 文本生成， 14 翻譯， 13

Transformer 架構， 57-84 關於，

1 · 57-59 添加分類頭， 75 添加層
 歸一化， 71 解碼器， 76 解碼器分
 支， 82 編碼器， 60-75 編碼器分
 支， 79-82 編碼器-解碼器注意力，
 76 編碼器-解碼器 branch， 83
 family tree， 78 feed-forward
 layer， 70 positional embeddings，
 73 self-attention， 61-70
 transformers about， xii 作為特徵
 提取器， 38-45

BERT， 9

efficiency of， 209-247
 fine-tuned on SQuAD， 173

- 微調，45-54用於命
- 名實體識別，96
- GPT, 9
- main challenges with, 19
- multilingual, 92 scaling
(see scaling transformers) training
(see training transformers)
- 關於 Transformers 庫，9個作為
 - Hugging Face 生態系統的一部分，15個自動類，33個微調模型，45-50個來自 Hub 的加載模型，38個來自 Hub 的加載標記器，33個管道，10-15個保存模型輪轂，53
- TransformersReader，187翻譯作為transformer應用，13
- ü
- UDA（無監督數據增強），250，295
- ULMFiT（通用語言模型精細調音），1, 8
- UMAP算法，42
- Unicode歸一化、94、314
- Unigram，312未標記數據利用，289-296高檔，82
- UST（不確定性感知自我訓練），250，295
- V
- 值，62香
- 草變壓器，微調，284
- 視覺，355-358, 364-370
- VisualBERT 模型，365可
- 視化訓練集，42
- ViT 模型，356
- VQA 數據集，364
- W
- Wav2Vec2 models, 362
- webtext, 87, 303, 349
- weight pruning about, 243 methods for, 244-247 sparsity in deep neural networks, 244 weighted average, 61
- 權重和偏差，331
- WikiANN 數據集，88詞
- 分詞，31
- WordPiece, 33, 93, 312
- word_ids() 函數, 103
- 使用 Transformer 寫入，124
- write_documents() 方法，185
- X
- XLM 型號，80
- XLM - RoBERTa型號，39、80、93、106-115、174
- XTREME 基準測試，88
- Z
- 零點、231零鏡頭分類、263-271零鏡頭跨語言遷移、87零鏡頭學習、88零鏡頭遷移、88、116

關於作者

Lewis Tunstall 是 Hugging Face 的機器學習工程師。他在 NLP、拓撲數據分析和時間序列領域為初創企業和企業構建了機器學習應用程序。Lewis 擁有理論物理學博士學位，並曾在澳大利亞、美國和瑞士擔任研究職位。他目前的工作重點是為 NLP 社區開發工具並教人們有效地使用它們。

Leandro von Werra 是 Hugging Face 開源團隊的一名機器學習工程師。他擁有多年的行業經驗，通過跨整個機器學習堆棧將 NLP 項目投入生產，並且是一個名為 TRL 的流行 Python 庫的創建者，該庫將轉換器與強化學習相結合。

Thomas Wolf 是 Hugging Face 的首席科學官兼聯合創始人。他的團隊肩負著促進和民主化 NLP 研究的使命。在共同創立 Hugging Face 之前，Thomas 獲得了物理學博士學位，後來又獲得了法學學位。他曾擔任物理研究員和歐洲專利代理人。

版權所有

Natural Language Processing with Transformers 封面上的鳥是椰子澳洲鸚鵡 (*Trichoglossus haematodus*)，它是長尾小鸚鵡和鸚鵡的親戚。它也被稱為綠枕澳洲鸚鵡，原產於大洋洲。

椰子澳洲鸚鵡的羽毛融入了它們豐富多彩的熱帶和亞熱帶環境。他們的綠色頸背與深藍色腦袋下方的黃色項圈相接，腦袋末端是橙紅色的喙。他們的眼睛是橙色的，胸前的羽毛是紅色的。椰子澳洲鸚鵡的尾巴是七種澳洲鸚鵡中最長的尖尾巴之一，從上面是綠色的，從下面是黃色的。這些鳥長 10 到 12 英寸，重 3.8 到 4.8 盎司。

椰子吸蜜鸚鵡有一個一夫一妻制的伴侶，一次產兩個啞光白蛋。它們在桉樹上築巢 80 多英尺高，在野外可存活 15 至 20 年。該物種遭受棲息地喪失和寵物貿易捕獲。O'Reilly 封面上的許多動物都瀕臨滅絕；所有這些對世界都很重要。

封面插圖由 Karen Montgomery 繪製，基於英國百科全書的黑白版畫。封面字體是 Gilroy Semibold 和 Guardian Sans。

文字字體為 Adobe Minion Pro；標題字體為 Adobe Myriad Condensed；代碼字體是 Dalton Maag 的 Ubuntu Mono。

O'REILLY®

向專家學習。
成為你自己。

圖書 | 直播在線課程

即時解答 | 虛擬事件

視頻 | 互動學習

從 oreilly.com 開始。