

## 第 5 章

## 文本生成

基於轉換器的語言模型最不可思議的特徵之一是它們能夠生成與人類編寫的文本幾乎沒有區別的文本。一個著名的例子是 OpenAI 的 GPT-2，當給出提示時：<sup>1</sup>

在一個令人震驚的發現中，科學家發現了一群獨角獸生活在安第斯山脈一個以前未開發的偏遠山谷中。更令研究人員驚訝的是，獨角獸能說一口流利的英語。

能夠生成一篇關於會說話的獨角獸的引人注目的新聞文章：

這位科學家以他們獨特的角命名了這個種群，奧維德的獨角獸。這些四隻角的銀白色獨角獸以前不為科學界所知。現在，在將近兩個世紀之後，引發這種奇怪現象的謎團終於解開了。拉巴斯大學的進化生物學家 Jorge Pérez 博士和幾位同伴在探索安第斯山脈時發現了一個小山谷，沒有其他動物或人類。佩雷斯注意到山谷中有一個似乎是天然噴泉的東西，周圍環繞著兩座岩石和銀色雪峰。

佩雷斯和其他人隨後冒險深入山谷。“當我們到達一座山峰的頂部時，水看起來是藍色的，頂部有一些水晶，”佩雷斯說。看到獨角獸群，佩雷斯和他的朋友們都大吃一驚。從空中就可以看到這些生物，而不必移動太多就可以看到它們。它們離得如此之近，以至於可以觸摸到它們的角。在檢查這些奇怪的生物時，科學家們發現這些生物也會說一些相當普通的英語……

---

<sup>1</sup>這個例子來自 OpenAI 關於 GPT-2 的博文。

這個例子之所以如此引人注目，是因為它是在沒有任何明確監督的情況下生成的。通過簡單地學習預測數百萬網頁文本中的下一個單詞，GPT-2 及其更強大的後代（如 GPT-3）能夠獲得廣泛的技能和模式識別能力。這些技能和模式識別能力可以通過不同類型的輸入提示。圖 5-1 顯示了語言模型在預訓練期間有時如何暴露於任務序列，在這些任務序列中，它們需要僅根據上下文預測以下標記，例如加法、解讀單詞和翻譯。這使他們能夠在微調期間或（如果模型足夠大）在推理時有效地轉移這些知識。這些任務不是提前選擇的，而是自然發生在用於訓練十億參數語言模型的龐大語料庫中。

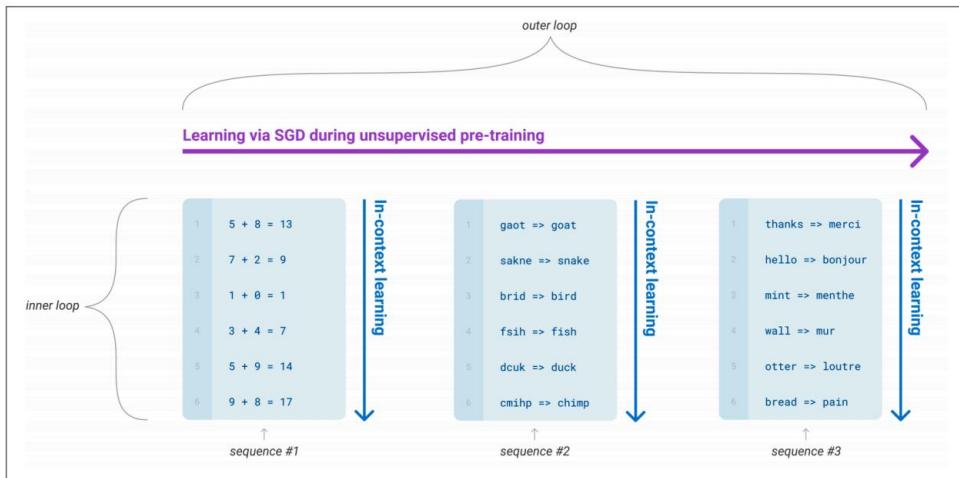


圖 5-1。在預訓練期間，語言模型暴露於可在推理期間調整的任務序列（由 Tom B. Brown 提供）

轉換器生成真實文本的能力導致了各種各樣的應用，比如 InferKit，用變壓器寫，人工智能地下城，和 Google 的 Meena 這樣的會話代理甚至可以講老生常談的笑話，如圖 5-2 所示！

2 然而，正如 Delip Rao 指出的那樣，米娜是否打算講老生常談的笑話是一個微妙的問題。

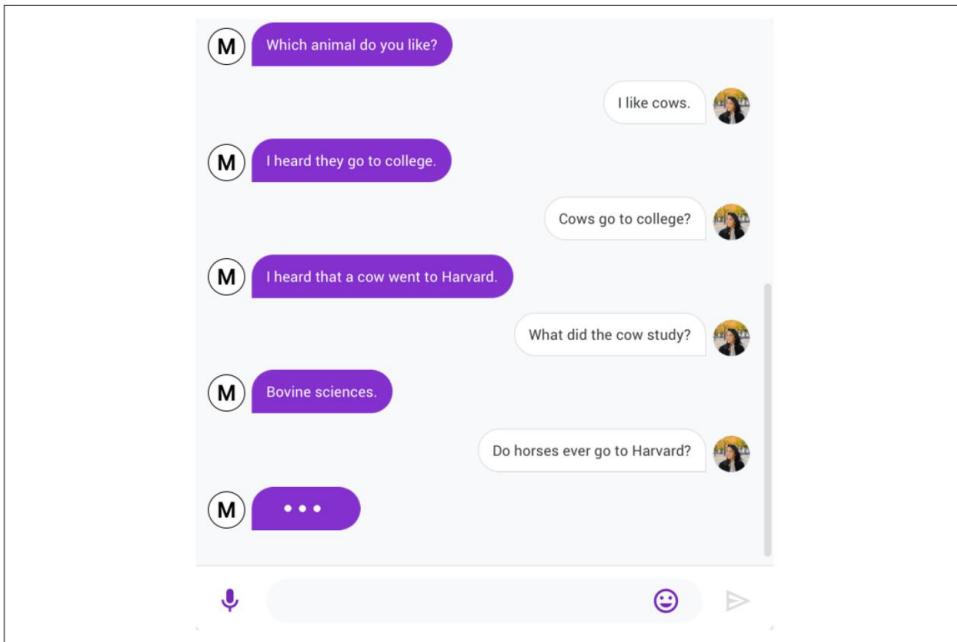


圖 5-2 左邊的 Meena 向右邊的人講一個老生常談的笑話（由 Daniel Adiwardana 和 Thang Luong 提供）

在本章中，我們將使用 GPT-2 來說明文本生成如何用於語言模型，並探索不同的解碼策略如何影響生成的文本。

## 生成連貫文本的挑戰

到目前為止，在本書中，我們一直專注於通過結合預訓練和監督微調來處理 NLP 任務。正如我們所見，對於序列或標記分類等任務特定的頭部，生成預測非常簡單，該模型產生一些 logits，我們要么取最大值來獲得預測的類別，要么應用 softmax 函數來獲得每個類別的預測概率。

相比之下，將模型的概率輸出轉換為文本需要一種解碼方法，這會帶來一些文本生成獨有的挑戰：

- 解碼是迭代完成的，因此與簡單地通過模型的正向傳遞傳遞一次輸入相比，涉及的計算要多得多。
- 生成文本的質量和多樣性取決於解碼方法和相關超參數的選擇。

要了解此解碼過程的工作原理，讓我們首先檢查 GPT-2 是如何預訓練並隨後應用於生成文本的。

與其他自回歸或因果語言模型一樣，GPT-2 被預訓練為在給定一些初始提示或上下文序列 =  $x_1, x_2, \dots, x_k$  由前後數個字元的連續條件來計算每個字元的條件概率是乘積實際的，因此通

( | )

$$y_1, \dots, y_t \mid \dots = \prod_{t=1}^n y_t \mid \dots, <\text{end}$$

在哪裡  $<\text{end}$  是序列  $y_1, \dots, y_{t-1}$  的簡寫符號。

正是從這些條件概率

中，我們得出的直覺是，自回歸語言建模相當於在給定句子中前面的詞的情況下預測每個詞；這正是前面等式右邊的概率所描述的。請注意，此預訓練目標與 BERT 有很大不同，後者利用過去和未來的上下文來預測掩碼標記。

到目前為止，您可能已經猜到我們如何調整下一個標記預測任務以生成任意長度的文本序列。如圖 5-3 所示，我們從“變形金剛是”這樣的提示開始，並使用模型來預測下一個標記。

一旦我們確定了下一個標記，我們將其附加到提示中，然後使用新的輸入序列生成另一個標記。我們這樣做，直到達到特殊的序列結束標記或預定義的最大長度。

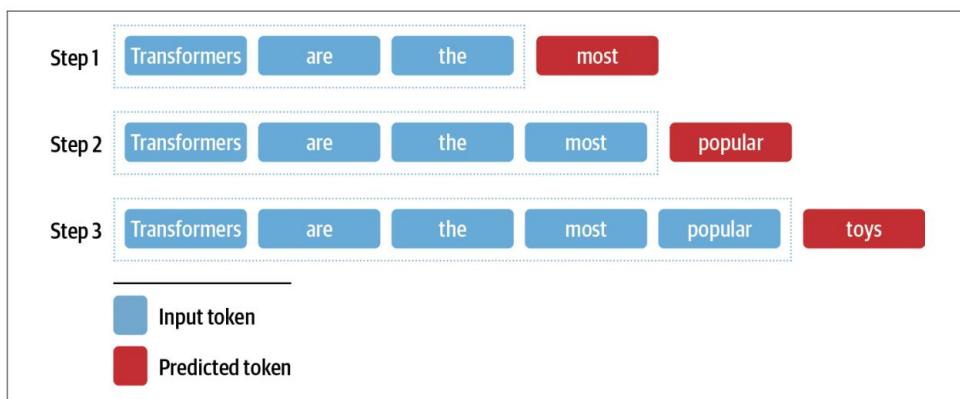


圖 5-3 通過在每一步向輸入添加一個新詞來從輸入序列生成文本



由於輸出序列以輸入提示的選擇為條件，因此這種文本生成通常稱為條件文本生成。

這個過程的核心是一種解碼方法，它確定在每個時間步選擇哪個標記。由於語言模型 head 在每個步驟中為詞彙表中的每個標記生成一個 logit  $z_{t,i}$ ，我們可以通過採用 softmax 獲得下可能的標記  $w_i$  的概率分佈：

$$P(y_t = w_i | y_{<t}, \cdot) = \text{softmax}(z_{t,i})$$

大多數解碼方法的目標是通過選擇一個這樣的序列來搜索最可能的整體序列：

$$\hat{y} = \text{argmax } P(\cdot | \cdot)$$

直接查找將涉及使用語言模型評估每個可能的序列。由於不存在可以在合理時間內完成此操作的算法，因此我們依賴於近似值。在本章中，我們將探索其中的一些近似值，並逐漸構建更智能、更複雜的算法，這些算法可用於生成高質量的文本。

## 貪心搜索解碼

從模型的連續輸出中獲取離散標記的最簡單解碼方法是在每個時間步貪婪地選擇概率最高的標記：

$$\hat{t} = \text{argmax } y_t \quad P(y_t | y_{<t}, \cdot)$$

要了解貪婪搜索的工作原理，讓我們首先加載帶有語言建模頭的 15 億參數版本的 GPT-2<sup>3</sup>

[從變壓器導入火炬](#)

導入 AutoTokenizer，AutoModelForCausalLM

```
device = "cuda" if torch.cuda.is_available() else "cpu"
model_name = "gpt2-xl"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModelForCausalLM.from_pretrained(model_name).to(device)
```

現在讓我們生成一些文本！儘管 Transformers 為 GPT-2 等自回歸模型提供了 generate() 函數，但我們將實現這種解碼方法

<sup>3</sup>如果您的機器內存不足，您可以通過替換 model\_name = 來加載較小的 GPT-2 版本“gpt-xl”與 model\_name = “gpt”。

我們自己看看引擎蓋下發生了什麼。為了熱身，我們將採用圖 5-3 中所示的相同迭代方法：我們將使用“Transformers are the”作為輸入提示並運行八個時間步的解碼。在每個時間步，我們為提示中的最後一個標記選擇模型的對數，並用 softmax 將它們包裝起來以獲得概率分佈。然後我們選擇概率最高的下一個標記，將其添加到輸入序列中，然後再次運行該過程。以下代碼完成這項工作，並在每個時間步存儲五個最可能的標記，以便我們可以可視化備選方案：

將熊貓導入為pd

```
input_txt = "變形金剛是" input_ids =
tokenizer(input_txt, return_tensors= pt )[ input_ids ].to(device) iterations = [] n_steps = 8
choices_per_step = 5
```

用torch.no\_grad()對於範圍

```
(n_steps):
    迭代= dict ()迭代[ “輸
        入” ]= tokenizer.decode (input_ids [0])輸出=模型 (input_ids =
        input_ids)
        # 選擇第一批和最後一個標記的 logits 並應用 softmax next_token_logits = output.logits[0, -1, :]
        next_token_probs = torch.softmax(next_token_logits, dim=-1) sorted_ids =
        torch.argsort(next_token_probs, dim=-1 ,降序=真)

        #在range(choices_per_step)中存儲具有最高概率的choice_idx
        標記： token_id = sorted_ids[choice_idx] token_prob =
        next_token_probs[token_id].cpu().numpy()
        token_choice = ( f {tokenizer.decode(token_id)} ({100 * token_prob:.2f}
        %)

    ) iteration[f Choice {choice_idx+1} ] = token_choice # 將預測的下一
    個標記附加到輸入input_ids = torch.cat([input_ids, sorted_ids[None, 0,
    None]], dim=-1) iterations.append(迭代)
```

pd.DataFrame (迭代)

	輸入	選擇1	選擇2	選擇	選擇	選擇5
0	變形金剛是	大多數 (8.53%)	只有 (4.96%)	最好 (4.65%)	變形金剛 (4.37%)	最終的 (2.16%)
1	變形金剛最多	受歡迎 (16.78%)	強大 (5.37%)	普通 (4.96%)	著名 (3.72%)	成功 (3.20%)
2	變形金剛最受歡迎	玩具 (10.63%)	玩具 (7.23%)	變形金剛 (6.60%)	的 (5.46%)	和 (3.76%)
3	變形金剛是最受歡迎的玩具	線 (34.38%)	在 (18.20%)	的 (11.71%)	品牌 (6.10%)	線 (2.69%)

	輸入	選擇1	選擇	選擇	選擇	選擇5
4變形金剛是最受歡迎的玩具系列	在 (46.28%)	的 (15.09%)	, (4.94%)	上 (4.40%)	曾經 (2.72%)	
5變形金剛是世界上最受歡迎的玩具系列	(65.99%)	歷史 (12.42%)	美國 (6.91%)	日本 (2.44%)	北方 (1.40%)	
6變形金剛是世界上最受歡迎的玩具系列	世界 (69.26%)	團結的 (4.55%)	歷史 (4.29%)	美國 (4.23%)	你 (2.30%)	
7變形金剛最世界流行的玩具系列	, (39.73%)	. (30.64%)	和 (9.87%)	與 (2.32%)	今天 (1.74%)	

通過這種簡單的方法，我們能夠生成“變形金剛是世界上最受歡迎的玩具系列”這句話。有趣的是，這表明 GPT-2 已經內化了一些關於變形金剛媒體專營權的知識，該專營權由兩家玩具公司（孩之寶和 Takara Tomy）創建。我們還可以在每個步驟中看到其他可能的延續，這顯示了文本生成的迭代性質。與序列分類等其他任務不同，在這些任務中，單個前向傳遞就足以生成預測，而對於文本生成，我們需要一次解碼一個輸出標記。

實現貪心搜索並不太難，但我們想要使用Transformers 的內置generate()函數來探索更複雜的解碼方法。為了重現我們的簡單示例，讓我們確保關閉採樣（默認情況下它是關閉的，除非您從狀態加載檢查點的模型的特定配置否則）並指定 max\_new\_tokens 作為新生成的令牌的數量：

```
input_ids = tokenizer(input_txt, return_tensors='pt')[input_ids].to(device)
output = model.generate(input_ids, max_new_tokens=n_steps, do_sample=False)
print(tokenizer.decode(output[0]))
```

變形金剛是世界上最受歡迎的玩具系列，

現在讓我們嘗試一些更有趣的事情：我們可以從 OpenAI 重現獨角獸的故事嗎？正如我們之前所做的那樣，我們將使用分詞器對提示進行編碼，並且我們將為max\_length指定一個更大的值以生成更長的文本序列：

```
max_length = 128
input_txt = "在一個令人震驚的發現中，科學家發現一群獨角獸生活在安第斯山脈—\n個以前未開發過的偏遠山谷中。更讓研究人員感到驚訝的是，獨角獸說話完美英語。\\n\\n
```

```
input_ids = tokenizer(input_txt, return_tensors='pt')[input_ids].to(device)
output_greedy = model.generate(input_ids, max_length=max_length, do_sample=False)
print(tokenizer.decode(output_greedy[0]))
```

在一個令人震驚的發現中，科學家發現了一群獨角獸生活在安第斯山脈一個以前未開發的偏遠山谷中。更令研究人員驚訝的是，獨角獸能說一口流利的英語。

來自加州大學戴維斯分校和科羅拉多大學博爾德分校的研究人員正在對安第斯雲霧林進行研究，這裡是稀有云霧林樹種的家園。

研究人員驚訝地發現獨角獸能夠相互交流，甚至與人類交流。

研究人員驚訝地發現獨角獸能夠

好吧，前幾句話與 OpenAI 示例完全不同，並且有趣地涉及不同大學的發現！我們還可以看到貪婪搜索解碼的主要缺點之一：它往往會產生重複的輸出序列，這在新聞文章中肯定是不受歡迎的。這是貪婪搜索算法的一個常見問題，它可能無法為您提供最優解；在解碼的上下文中，他們可能會錯過總體概率較高的單詞序列，因為高概率單詞恰好在低概率單詞之前。

幸運的是，我們可以做得更好。讓我們研究一種流行的方法，稱為波束搜索解碼。



儘管貪婪搜索解碼很少用於需要多樣性的文本生成任務，但它對於生成短序列很有用，例如首選確定性和事實正確輸出的算術。對於這些任務，您可以通過提供條件來調節 GPT-2 一些格式為 “ $5 + 8 \Rightarrow 13$   $\backslash n 7 + 2 \Rightarrow 9 \backslash n 1 + 0 \Rightarrow$ ” 的行分隔示例作為輸入提示。

## 波束搜索解碼

beam search 不是在每一步解碼概率最高的 token，而是跟蹤前 b 個最有可能的下一個 token，其中 b 指的是 beam 或部分假設的數量。通過考慮現有集合的所有可能的下一個標記擴展並選擇 b 個最可能的擴展來選擇下一組波束。重複該過程，直到我們達到最大長度或 EOS

---

4 NS Keskar 等人，“CTRL：用於可控生成的條件轉換器語言模型”，(2019)。

token，並且最可能的序列是通過根據它們的對數概率對  $b$  個波束進行排序來選擇的。圖 5-4 顯示了集束搜索的示意。

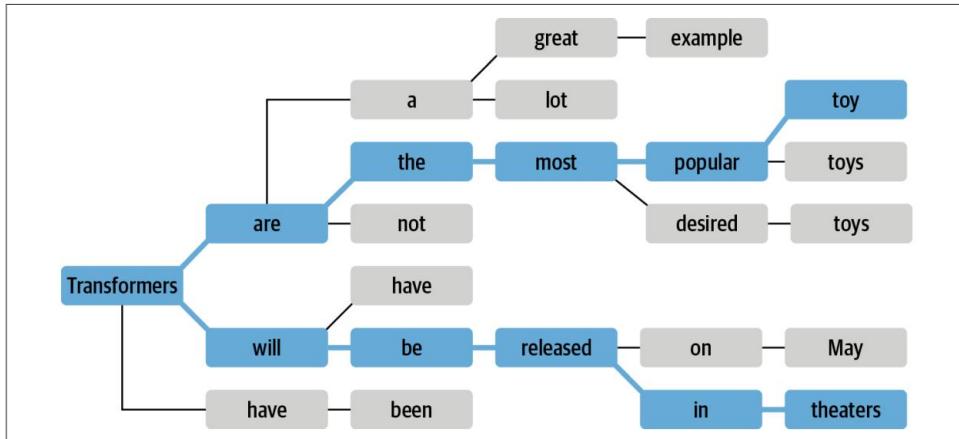


圖 5-4 帶有兩個波束的波束搜索

為什麼我們使用對數概率而不是概率本身對序列進行評分？計算序列  $P(y_1, y_2, \dots, y_t)$  的總體概率涉及計算條件概率  $P(y_t | y_{<t})$  的乘積，因為每個條件概率通常是  $[0, 1]$  範圍內的一個小數，取它們的乘積可以導致很容易下溢的總體概率。這意味著計算機無法再精確地表示計算結果。例如，假設我們有一個包含  $t = 1024$  個標記的序列，並且慷慨地假設每個標記的概率為 0.5。這個序列的總體概率是一個非常小的數字：

$$0.5^{1024}$$

$$5.562684646268003e-309$$

當我們遇到下溢時，這會導致數值不穩定。我們可以通過計算一個相關項，即對數概率來避免這種情況。如果我們將對數應用於聯合概率和條件概率，那麼在對數乘積規則的幫助下，我們得到：

$$\text{日誌 } P(y_1, \dots, y_t) \mid \quad ) = \sum_{t=1}^{\text{否}} \log P(y_t | y_{<t}, \quad )$$

換句話說，我們之前看到的概率乘積變成了對數概率之和，這不太可能遇到數值不穩定。例如，計算與前面相同的示例的對數概率給出：

將numpy導入為np

總和 ([np.log (0.5) ] \* 1024)

-709.7827128933695

這是一個我們可以輕鬆處理的數字，這種方法仍然適用於更小的數字。由於我們只想比較相對概率，我們可以直接用對數概率來做。

我們來計算比較一下貪心和集束搜索生成的文本的對數概率，看集束搜索是否能提高整體概率。由於 Transformers 模型返回給定輸入標記的下一個標記的非標準化 logits，因此我們首先需要對 logits 進行標準化，以便為序列中的每個標記創建整個詞彙表的概率分佈。然後我們只需要選擇序列中出現的標記概率。下面的函數實現了這些步驟：

將torch.nn.functional導入為F

```
def log_probs_from_logits(logits, labels): logp =
    F.log_softmax(logits, dim=-1) logp_label =
    torch.gather(logp, 2, labels.unsqueeze(2)).squeeze(-1)返回logp_label
```

這為我們提供了單個標記的對數概率，因此要獲得序列的總對數概率，我們只需要對每個標記的對數概率求和：

```
def sequence_logprob(model, labels, input_len=0): with
    torch.no_grad(): output = model(labels) log_p =
        log_probs_from_logits( output.logits[:, :-1, :], 標籤[:, 1:])
        seq_log_prob = torch.sum(log_p[:, input_len:])返回
        seq_log_prob.cpu().numpy()
```

請注意，我們忽略了輸入序列的對數概率，因為它們不是由模型生成的。我們還可以看到對齊 logits 和標籤很重要；由於模型預測下一個標記，我們沒有得到第一個標籤的 logit，並且我們不需要最後一個 logit，因為我們沒有它的基本事實標記。

讓我們使用這些函數首先計算貪婪解碼器在 OpenAI 提示符下的序列對數概率：

```
logp = sequence_logprob(model, output_greedy, input_len=len(input_ids[0]))
print(tokenizer.decode(output_greedy[0])) print(f' \nlog-prob: {logp:.2f} )
```

在一個令人震驚的發現中，科學家發現了一群獨角獸生活在安第斯山脈一個以前未開發的偏遠山谷中。更令研究人員驚訝的是，獨角獸能說一口流利的英語。

來自加州大學戴維斯分校和科羅拉多大學博爾德分校的研究人員正在對安第斯雲霧林進行研究，這裡是稀有云霧林樹種的家園。

研究人員驚訝地發現獨角獸能夠相互交流，甚至與人類交流。

研究人員驚訝地發現獨角獸能夠

對數概率：-87.43

現在讓我們將其與使用波束搜索生成的序列進行比較。要使用generate()函數激活束搜索，我們只需要使用num\_beams參數指定束的數量。我們選擇的光束越多，可能獲得的結果就越好；然而，生成過程變得更慢，因為我們為每個光束生成並行序列：

```
output_beam = model.generate(input_ids, max_length=max_length, num_beams=5, do_sample=False)
logp = sequence_logprob(model, output_beam,
input_len=len(input_ids[0]))打印(tokenizer.decode(output_beam[0])) print(f' \nlog-prob: {logp:.2f} )
```

在一個令人震驚的發現中，科學家發現了一群獨角獸生活在安第斯山脈一個以前未開發的偏遠山谷中。更令研究人員驚訝的是，獨角獸能說一口流利的英語。

獨角獸的發現是由來自加利福尼亞大學聖克魯斯分校和國家地理學會的一組科學家發現的。

科學家們在對安第斯山脈進行研究時，發現了一群獨角獸生活在安第斯山脈一個以前未開發過的偏遠山谷中。更令研究人員驚訝的是，獨角獸能說一口流利的英語。

對數概率：-55.23

我們可以看到，與使用簡單的貪心解碼相比，我們使用波束搜索獲得了更好的對數概率（越高越好）。但是，我們可以看到集束搜索也存在重複文本的問題。解決此問題的一種方法是使用no\_repeat\_ngram\_size參數施加 n-gram 懲罰，該參數跟蹤哪些 n-gram 已被看到，並將下一個標記概率設置為零，如果它會產生以前看到的

n-gram：

```

output_beam = model.generate(input_ids, max_length=max_length, num_beams=5,
                             do_sample=False, no_repeat_ngram_size=2)
logp = sequence_logprob(model, output_beam, input_len=len(input_ids[0]))
print(tokenizer.decode(output_beam[0])) print(f' \nlog-prob: {logp:.2f} ')

```

在一個令人震驚的發現中，科學家發現了一群獨角獸生活在安第斯山脈一個以前未開發的偏遠山谷中。更令研究人員驚訝的是，獨角獸能說一口流利的英語。

這一發現是由來自加利福尼亞大學聖克魯茲分校和國家地理學會的一組科學家做出的。

根據一份新聞稿，科學家們遇到牛群時正在對該地區進行調查。他們驚訝地發現自己能夠用英語與動物交談，儘管他們以前從未親眼見過獨角獸。研究人員是

對數概率： -93.12

這還不錯！我們已經設法阻止了重複，我們可以看到，儘管得分較低，但文本仍然連貫。帶有 n-gram 懲罰的集束搜索是在關注高概率標記（使用集束搜索）和減少重複（使用 n-gram 懲罰）之間找到權衡的好方法，它通常用於摘要或事實正確性很重要的機器翻譯。當事實正確性不如生成輸出的多樣性重要時，例如在開放域聊天或故事生成中，減少重複同時提高多樣性的另一種替代方法是使用抽樣。讓我們通過檢查一些最常見的採樣方法來完善我們對文本生成的探索。

## 取樣方法

最簡單的抽樣方法是在每個時間步從模型輸出在整個詞彙表上的概率分佈中隨機抽樣：

$$P(y_t = w_i | \quad \quad \quad ) = \text{softmax } z(t, i) = \frac{\exp z(t, i)}{\sum_{j=1}^{|V|} \exp z(t, j)}$$

其中  $V$  表示詞彙表的基數。我們可以通過添加一個溫度參數  $T$  來輕鬆控制輸出的多樣性，該參數在採用 softmax 之前重新調整 logits：

$$P(y_t = w_i | \text{context}) = \frac{\exp(z_{t,i} / T)}{\sum_{j=1}^{|V|} \exp(z_{t,j} / T)}$$

通過調整  $T$ ，我們可以控制概率分佈的形狀。當  $T \ll 1$  時，分佈在原點附近達到峰值，稀有標記被抑制。另一方面，當  $T \gg 1$  時，分佈變平，每個標記的可能性都相等。溫度對令牌概率的影響如圖 5-5 所示。

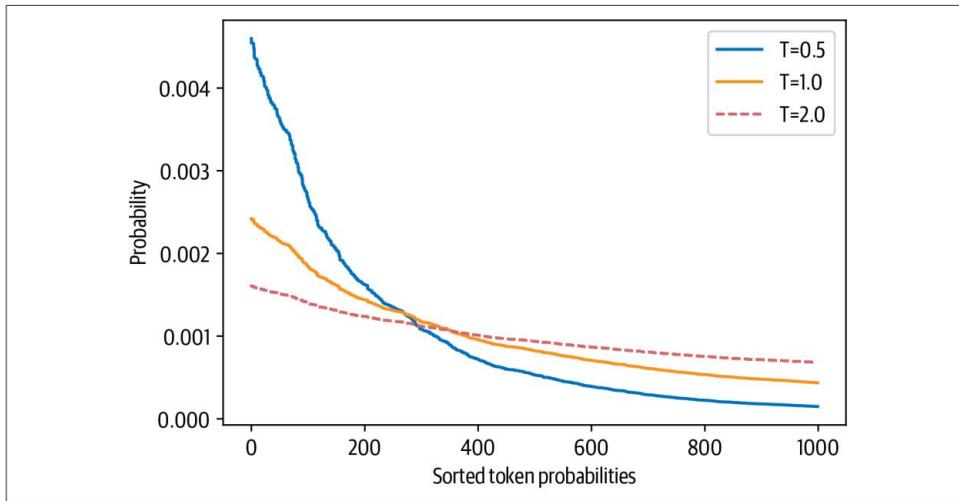


圖 5-5。三個選定溫度的隨機生成的標記概率分佈

要了解我們如何使用溫度來影響生成的文本，讓我們使用  $T=2$  通過在 `generate()` 函數中設置溫度參數（我們將在下一節解釋 `top_k` 參數的含義）：

```
output_temp = model.generate(input_ids, max_length=max_length, do_sample=True,
                             temperature=2.0, top_k=0)
print(tokenizer.decode(output_temp[0]))
```

在一個令人震驚的發現中，科學家發現了一群獨角獸生活在安第斯山脈一個以前未開發的偏遠山谷中。更令研究人員驚訝的是，獨角獸能說一口流利的英語。

趁車站是主角收到彭加拉懷舊花絮Regarding

5如果您了解一些物理知識，您可能會發現它與玻爾茲曼分佈驚人地相似。

Jenny loclonju AgreementCON 非理性的 rite Continent seaf A jer Turner  
 Dorbecue WILL Pumpkin mere Thatvernuildagain YoAniamond disse \*  
 Runewitingkusstemprop}};b zo coachinginventorymodules deflation press  
 Vaticanpres Wrestling chargesThingsctureddong Ty physician PET KimBi66 graz Oz at aff da temporou MD6  
 輻射儀

我們可以清楚地看到高溫產生了大部分亂碼 ;通過強調稀有標記 ,我們讓模型創建了奇怪的語法和相當多的自造詞 !讓我們看看如果我們降低溫度會發生什麼 :

```
output_temp = model.generate(input_ids, max_length=max_length, do_sample=True,
                             temperature=0.5, top_k=0)
print(tokenizer.decode(output_temp[0]))
```

在一個令人震驚的發現中 ,科學家發現了一群獨角獸生活在安第斯山脈一個以前未開發的偏遠山谷中 。更令研究人員驚訝的是 ,獨角獸能說一口流利的英語 。

科學家們正在尋找讓動物們又笑又哭的神秘聲音的來源 。

獨角獸住在安第斯山脈的一個偏遠山谷裡

阿根廷布宜諾斯艾利斯大學的研究員路易斯古茲曼說 :“當我們第一次聽到動物的叫聲時 ,我們以為是獅子或老虎 。”

但當

這明顯更加連貫 ,甚至引用了另一所被認為是這一發現的大學的話 !我們可以從溫度中吸取的主要教訓是 ,它允許我們控制樣本的質量 ,但始終需要在一致性 (低溫)和多樣性 (高溫)之間進行權衡 ,必須根據使用情況進行調整手頭的情況 。

調整連貫性和多樣性之間權衡的另一種方法是截斷詞彙的分佈 。這使我們能夠隨溫度自由調整多樣性 ,但在更有限的範圍內排除在上下文中太奇怪的詞 (即低概率詞) 。有兩種主要方法可以做到這一點 :top-k 和 nucleus (或 top-p)採樣 。讓我們來看看 。

## Top-k 和核採樣

Top-k 和 nucleus (top-p) 採樣是使用溫度的兩種流行替代方法或擴展方法 。在這兩種情況下 ,基本思想都是限制我們可以在每個時間步採樣的可能標記的數量 。要了解這是如何工作的 ,讓我們首先想像一下

如圖 5-6 所示，模型輸出在  $T = 1$  時的累積概率分佈。

讓我們梳理一下這些圖，因為它們包含很多信息。在上圖中，我們可以看到令牌概率的直方圖。它在  $10^{-8}$  附近有一個峰值，在  $10^{-4}$  附近有一個較小的第二個峰值，然後急劇下降，只有少數標記出現，概率在  $10^{-2}$  和  $10^{-1}$  之間。查看此圖，我們可以看到以最高概率（ $10^{-1}$  處的孤立條）挑選令牌的概率為十分之一。

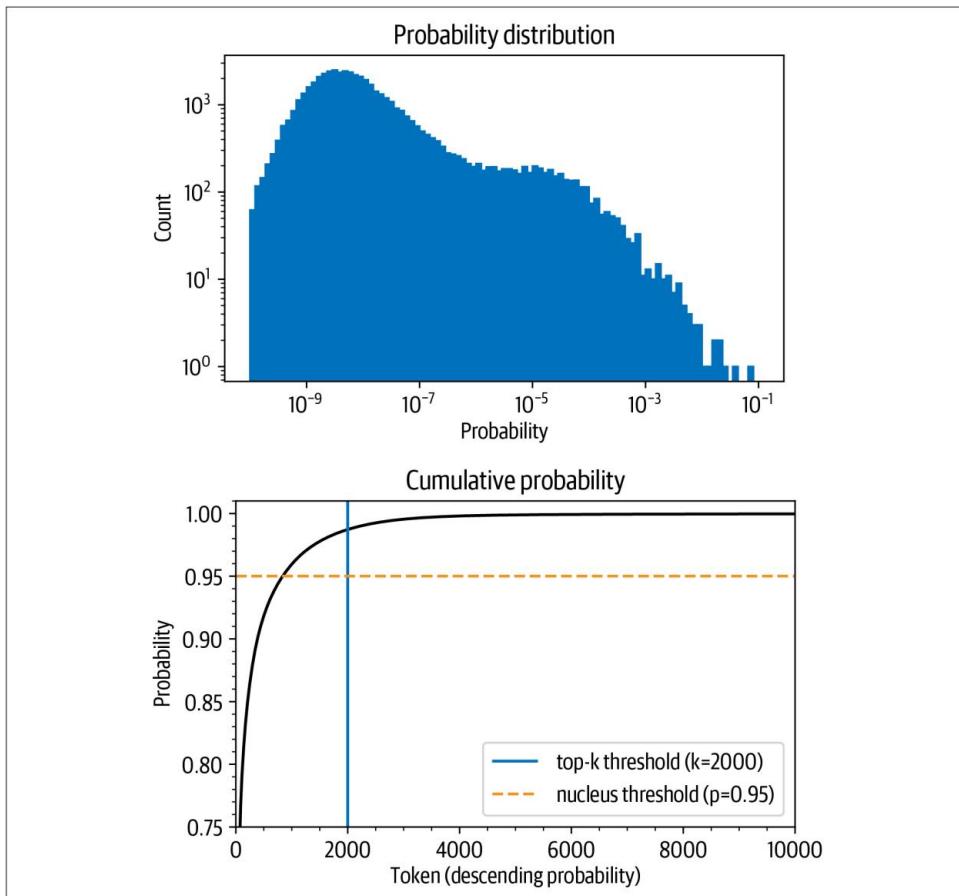


圖 5-6。下一個標記預測的概率分佈（上）和下降標記概率的累積分佈（下）

在下圖中，我們按降序概率對標記進行了排序，併計算了前 10,000 個標記的累計和（GPT-2 的詞彙表中總共有 50,257 個標記）。曲線表示選擇任何一個的概率

前面的標記。例如，大約有 96% 的機會以最高概率從 1,000 個代幣中選出任何一個。我們看到概率迅速上升到 90% 以上，但僅在幾千個令牌後才飽和到接近 100%。該圖顯示，有 100 分之一的機會不選擇甚至不在前 2,000 名中的任何代幣。

儘管這些數字乍一看可能很小，但它们變得很重要，因為我們在生成文本時對每個標記採樣一次。因此，即使只有 100 分之一或 1,000 分之一的機會，如果我們進行數百次採樣，也很有可能在某個時候選擇不太可能的標記，並且在採樣時選擇此類標記會嚴重影響生成文本的質量。出於這個原因，我們通常希望避免使用這些不太可能的令牌。這就是 top-k 和 top-p 採樣發揮作用的地方。

top-k 抽樣背後的想法是通過僅從概率最高的 k 個標記中抽樣來避免低概率選擇。這對分佈的長尾進行了固定切割，並確保我們僅從可能的選擇中進行抽樣。回到圖 5-6，top-k 採樣相當於定義一條垂直線並從左邊的標記中採樣。同樣，generate()函數提供了一種使用 top\_k 參數實現此目的的簡單方法：

```
output_topk = model.generate(input_ids, max_length=max_length, do_sample=True,
                             top_k=50)
print(tokenizer.decode(output_topk[0]))
```

在一個令人震驚的發現中，科學家發現了一群獨角獸生活在安第斯山脈一個以前未開發的偏遠山谷中。更令研究人員驚訝的是，獨角獸能說一口流利的英語。

野生獨角獸在與阿根廷接壤的卡哈馬卡地區的安第斯山脈漫遊（圖片來源：Alamy/Ecole Nationale Supérieure d'Histoire Naturelle）

研究人員在山谷中遇到了大約 50 種動物。他們在那個地方如此偏遠和孤立的地區生活了將近一千年，以至於

這可以說是迄今為止我們生成的最人性化的文本。但是我們如何選擇 k？k 的值是手動選擇的，並且對於序列中的每個選擇都是相同的，與實際輸出分佈無關。我們可以通過查看一些文本質量指標來找到一個合適的 k 值，我們將在下一章探討這些指標。但固定的截止值可能不是很令人滿意。

另一種方法是使用動態截止。對於 nucleus 或 top-p 採樣，我們不是選擇固定的截止值，而是設置何時截止的條件。這種情況是在選擇中達到一定的概率質量時。假設我們設置了那個值

到 95%。然後我們按概率降序排列所有標記，並從列表頂部一個接一個地添加標記，直到所選標記的概率總和為 95%。回到圖 5-6，`p` 的值在概率累積和圖上定義了一條水平線，我們僅從該線下方的標記中採樣。根據輸出分佈，這可能只是一個（很可能）令牌或一百個（更有可能）令牌。此時，您可能不會對`generate()`函數還提供一個用於激活 top-p 採樣的參數感到驚訝。讓我們試試看：

```
output_topp = model.generate(input_ids, max_length=max_length, do_sample=True, top_p=0.90)
print(tokenizer.decode(output_topp[0]))
```

在一個令人震驚的發現中，科學家發現了一群獨角獸生活在安第斯山脈一個以前未開發的偏遠山谷中。更令研究人員驚訝的是，獨角獸能說一口流利的英語。

科學家們研究了這些動物的 DNA，得出的結論是，這群人是大約 5 萬年前生活在阿根廷的史前獸群的後代。

據科學分析，最早遷徙到南美洲的人類是在最後一個冰河時代結束後，從南非和澳大利亞遷徙到安第斯山脈的。

自遷徙以來，這些動物一直在適應

Top-p 抽樣也產生了一個連貫的故事，這一次有一個關於從澳大利亞到南美洲的移民的新轉折。您甚至可以結合兩種採樣方法來獲得兩全其美的效果。設置`top_k=50`和`top_p=0.9`對應於從最多 50 個 token 的池中選擇概率質量為 90% 的 token 的規則。



我們還可以在使用採樣時應用波束搜索。我們可以對它們進行採樣並以相同的方式構建光束，而不是貪婪地選擇下一批候選令牌。

## 哪種解碼方法最好？

不幸的是，沒有普遍的“最佳”解碼方法。哪種方法最好取決於您為其生成文本的任務的性質。如果你想讓你的模型執行精確的任務，比如算術或提供特定問題的答案，那麼你應該降低溫度或使用確定性方法，比如貪婪搜索結合集束搜索來保證得到最有可能的答案。如果你想讓模型生成更長的文本，甚至有點創意，那麼你應該切換到採樣方法並提高溫度，或者混合使用 top-k 和核採樣。

## 結論

在本章中，我們研究了文本生成，這是一個與我們之前遇到的 NLU 任務截然不同的任務。生成文本需要每個生成的標記至少一次正向傳遞，如果我們使用波束搜索，則需要更多。這使得文本生成對計算要求很高，並且需要合適的基礎設施來大規模運行文本生成模型。此外，將模型的輸出概率轉換為離散標記的良好解碼策略可以提高文本質量。

尋找最佳解碼策略需要進行一些實驗並對生成的文本進行主觀評估。

然而，在實踐中，我們不想僅僅根據直覺做出這些決定！與其他 NLP 任務一樣，我們應該選擇一個反映我們要解決的問題的模型性能指標。不出所料，選擇範圍很廣，我們將在下一章中遇到最常見的選擇，我們將了解如何訓練和評估文本摘要模型。或者，如果您迫不及待地想學習如何從頭開始訓練 GPT 類型的模型，您可以直接跳到第10章，我們將在其中收集大量代碼，然後在其上訓練自回歸語言模型。

## 第 6 章

### 總結

有時，您可能需要總結一份文件，無論是研究文章、財務收益報告還是一系列電子郵件。如果你仔細想想，這需要一系列的能力，比如理解長篇文章、對內容進行推理，以及產生流利的文本，其中包含原始文檔的主要主題。此外，準確總結新聞文章與總結法律合同有很大不同，因此能夠做到這一點需要復雜程度的領域泛化。由於這些原因，文本摘要對於包括轉換器在內的神經語言模型來說是一項艱鉅的任務。儘管存在這些挑戰，文本摘要為領域專家提供了顯著加快工作流程的前景，並被企業用來濃縮內部知識、總結合同、自動生成社交媒體發布的內容等等。

為了幫助您理解所涉及的挑戰，本章將探討我們如何利用預訓練的轉換器來總結文檔。摘要是一種經典的序列到序列 (seq2seq) 任務，具有輸入文本和目標文本。正如我們在第 1 章中看到的，這就是編碼器-解碼器轉換器的優勢所在。

在本章中，我們將構建自己的編碼器-解碼器模型，將幾個人之間的對話濃縮成一個清晰的摘要。但在開始之前，讓我們先看一下用於摘要的規範數據集之一：CNN/DailyMail 語料庫。

### CNN/DailyMail 數據集

CNN/DailyMail 數據集包含大約 300,000 對新聞文章及其相應的摘要，這些摘要由 CNN 和 DailyMail 附加在其文章中的要點組成。數據集的一個重要方面是

摘要是抽象的而不是提取的，這意味著它們由新句子而不是簡單的摘錄組成。該數據集在Hub上可用；我們將使用版本3.0.0，這是一個為匯總而設置的非匿名版本。我們可以使用與第4章中看到的拆分類似的方式，使用版本關鍵字來選擇版本。因此，讓我們深入研究一下：

從數據集導入load\_dataset

```
dataset = load_dataset('cnn_dailymail', version='3.0.0')
print(f'Features: {dataset['train'].column_names}')
```

特點：[文章，亮點，id]

數據集包含三列：article，其中包含新聞文章、帶有摘要的亮點，以及用於唯一標識每篇文章的id。我們來看一篇文章的節選：

```
樣本=數據集['火車'][1]打印(f
文章（摘錄500個字符，總長度:{len(sample['article'])}): ) print(sample['article'][:500])
print(f'\nSummary (length: {len(樣本["亮點"])}): ) 打印(樣本["亮點"])
```

文章（節選500字，總長度：3192）：

(CNN) 尤塞恩·博爾特(Usain Bolt)在周日的世錦賽上圓滿收官，他在莫斯科奪得了他的第三枚金牌，他在男子4x100米接力賽中幫助牙買加隊取得了勝利。世界上跑得最快的人超越了美國對手賈斯汀·加特林，內斯塔·卡特，凱馬爾·貝利-科爾、尼克爾·阿什米德和博爾特組成的牙買加四人組以37.36秒的成績獲勝。美國以37.56秒獲得第二

在英國因錯誤交接被取消資格後，加拿大獲得銅牌。26歲的博爾特有n

Summary (length: 180): Usain Bolt 奪得世錦賽第三金。  
主播牙買加以4x100米接力獲勝。  
牙買加博爾特在女子4x100米接力錦標賽中獲得第八枚金牌。

我們看到，與目標摘要相比，文章可能會很長；在這種特殊情況下，差異是17倍。長文章對大多數轉換器模型構成了挑戰，因為上下文大小通常限制在1,000個標記左右，相當於幾段文本。處理此摘要的標準但粗略的方法是簡單地截斷超出模型上下文大小的文本。

顯然，文本末尾的摘要可能包含重要信息，但現在我們需要忍受模型架構的這種限制。

# 文本摘要管道

讓我們先定性地查看前面示例的輸出，看看一些最流行的用於摘要的 transformer 模型是如何執行的。儘管我們將探索的模型架構具有不同的最大輸入大小，但讓我們將輸入文本限制為 2,000 個字符，以使所有模型具有相同的輸入，從而使輸出更具可比性：

```
sample_text = dataset[ train ][1][ article ][:2000]
# 我們將在字典中收集每個模型的生成摘要 summaries = {}
```

摘要中的慣例是用換行符分隔摘要句子。

我們可以在每個句點後添加一個換行符，但這種簡單的啟發式方法對於“US”或“UN”這樣的字符串會失敗出現在縮寫中：

```
從nltk.tokenize
導入nltk導入sent_tokenize

nltk.download( punkt )

string = 美國是一個國家。聯合國是一個組織。 sent_tokenize(字符串)

[ 美國是一個國家。， 聯合國是一個組織。 ]
```



在接下來的部分中，我們將加載幾個大型模型。如果內存不足，您可以用較小的檢查點（例如，“gpt”、“t5-small”）替換大型模型，或者跳過本節並跳至第 154 頁的“在 CNN/DailyMail 數據集上評估 PEGASUS”。

## 總結基線

總結新聞文章的一個常見基準是簡單地使用文章的前三個句子。借助 NLTK 的句子分詞器，我們可以輕鬆實現這樣的基線：

```
def three_sentence_summary(text): return
    \n .join(sent_tokenize(text)[:3])

摘要[ “基線” ] = three_sentence_summary(sample_text)
```

## GPT-2

我們已經在第 5 章中看到 GPT-2 如何根據提示生成文本。

該模型的一個令人驚訝的特性是，我們還可以通過簡單地在輸入文本的末尾附加“TL;DR”來使用它來生成摘要。表達“TL;DR”（太長；沒讀）經常在 Reddit 等平台上使用，表示一篇長文章的簡短版本。我們將通過使用 Transformers 中的 pipeline()函數重新創建原始論文的過程來開始我們的摘要實驗。我們創建一個文本生成管道並加載大型 GPT-2 模型：



從變壓器導入管道，set\_seed

```
set_seed(42)
pipe = pipeline('text-generation', model='gpt2-xl') gpt2_query
= sample_text + '\nTL;DR:\n' pipe_out = pipe(gpt2_query,
max_length=512, clean_up_tokenization_spaces=True)摘要[ 'gpt2' ] = '\n'.join(
sent_tokenize(pipe_out[0][ 'generated_text' ][len(gpt2_query):]))
```

在這裡，我們只是通過切掉輸入查詢來存儲生成文本的摘要，並將結果保存在 Python 字典中以供以後比較。

## T5

接下來讓我們試試 T5 變壓器。正如我們在第 3 章中看到的，該模型的開發人員對 NLP 中的遷移學習進行了全面研究，發現他們可以通過將所有任務制定為文本到文本任務來創建通用的轉換器架構。T5 檢查點在無監督數據（重建掩碼詞）和監督數據的混合數據上進行訓練，用於多項任務，包括摘要。因此，這些檢查點可以直接用於執行摘要，而無需使用預訓練期間使用的相同提示進行微調。在此框架中，模型用於總結文檔的輸入格式是“summarize: <ARTICLE>”，而對於翻譯，它看起來像“將英語翻譯成德語 <TEXT>”。如圖 6-1 所示，這使得 T5 用途極為廣泛，可以讓您使用一個模型解決許多任務。

我們可以直接加載 T5 以使用 pipeline()函數進行匯總，該函數還負責將輸入格式化為文本到文本的格式，因此我們不需要在它們前面加上“summarize”：

```
pipe = pipeline('summarization', model='t5-large') pipe_out =
pipe(sample_text)摘要[ 't5' ] = '\n'.join(sent_tokenize(pipe_out[0]
[ 'summary_text' ]))
```

---

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

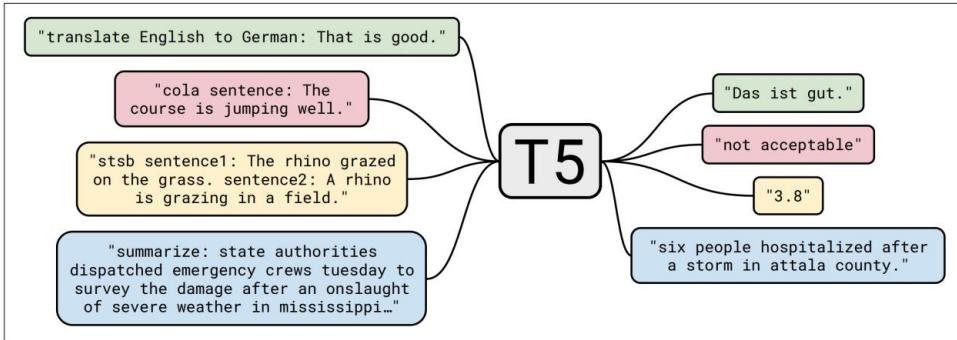


圖 6-1。T5 的文本到文本框架圖（由 Colin Raffel 提供）；除了翻譯和摘要外，還顯示了 CoLA（語言可接受性）和 STSB（語義相似性）任務

## 捷運

BART 還使用編碼器-解碼器架構，並經過訓練可以重建 cor-

輸入中斷。它結合了 BERT 和 GPT-2.2 的預訓練方案。我們將使用facebook/bart-large-cnn檢查點，它在 CNN/DailyMail 數據集上進行了專門微調：

```
pipe = pipeline( summarization ,model= facebook/bart-large-cnn ) pipe_out =
pipe(sample_text)摘要[ bart ] = \n .join(sent_tokenize(pipe_out[0]
[ summary_text ]))
```

## 飛馬座

與 BART 一樣，PEGASUS 也是一種編碼器-解碼器轉換器。<sup>3</sup>如圖6-2所示，其預訓練目標是預測多句文本中的掩碼句子。作者認為，預訓練目標越接近下游任務，它就越有效。為了找到比一般語言建模更接近摘要的預訓練目標，他們在一個非常大的語料庫中自動識別包含其周圍段落大部分內容的句子（使用摘要評估指標作為內容重疊的啟發式方法）並預訓練 PEGASUS 模型來重建這些句子，從而獲得最先進的文本摘要模型。

<sup>2</sup> M. Lewis 等人，“BART：用於自然語言生成的去噪序列到序列預訓練，翻譯和理解”，（2019）。

<sup>3</sup> J. Zhang 等人，“PEGASUS：使用提取的空缺句進行抽象摘要的預訓練”，（2019）。

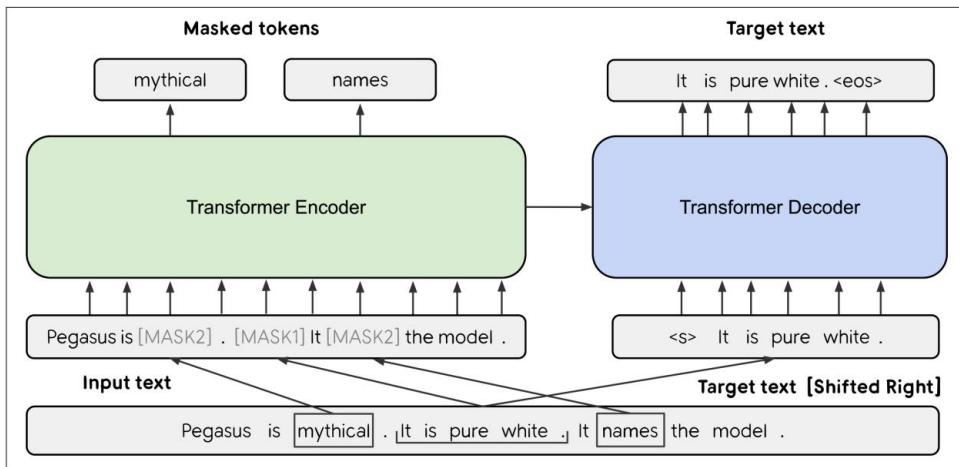


圖 6-2。PEGASUS 架構圖 (由 Jingqing Zhang 等人提供)

這個模型有一個特殊的換行標記，這就是我們不需要sent\_tokenize()函數的原因：

```
pipe = pipeline('summarization', model='google/pegasus-cnn_dailymail') pipe_out = pipe(sample_text)摘要[pegasus] = pipe_out[0][summary_text].replace('.<n>', '.\n')
```

## 比較不同的摘要

現在我們已經用四種不同的模型生成了摘要，讓我們比較一下結果。請記住，一個模型根本沒有在數據集上進行過訓練（GPT-2），一個模型已經針對此任務進行了微調（T5），並且兩個模型專門針對此任務進行了微調（BART 和 PEGASUS）。讓我們看一下這些模型生成的摘要：

```
print('GROUND TRUTH')
print(數據集[train][1][highlights]) print()
```

對於摘要中的model\_name：  
打印(model\_name.upper()) 打印(摘要[model\_name]) 打印("")

真相 Usain Bolt 贏  
得了世界錦標賽的第三枚金牌。  
主播牙賈加以4x100米接力獲勝。  
牙賈加博爾特在女子4x100米接力錦標賽中獲得第八枚金牌。

基線

(CNN) 尤塞恩·博爾特 (Usain Bolt) 在周日的世錦賽上圓滿收官，他在莫斯科奪得了他的第三枚金牌，他在男子 4x100 米接力賽中幫助牙買加隊取得了勝利。

世界上跑得最快的人超越了美國對手賈斯汀·加特林，內斯塔·卡特、凱馬爾·貝利·科爾、尼克爾·阿什米德和博爾特組成的牙買加四人組以 37.36 秒的成績獲勝。

美國隊以 37.56 秒獲得第二名，加拿大隊在英國隊因交接錯誤被取消資格後獲得銅牌。

#### GPT2

內斯塔，世界上跑得最快的人。  
加特林，有史以來最成功的奧運選手。  
凱馬爾，牙買加傳奇人物。  
雪莉·安，有史以來跑得最快的女人。  
博爾特，世界上最偉大的運動員。  
撐桿跳這項團隊運動

#### T5

博爾特在男子 4x100 米接力賽中贏得了他的第三枚世錦賽金牌。26 歲的牙買加隊在俄羅斯首都的比賽中取得了勝利

他現在已經在錦標賽中獲得了八枚金牌，追平了紀錄

#### 捷運

烏塞恩博爾特在莫斯科贏得了他的第三枚世錦賽金牌。  
博爾特帶領牙買加隊在男子 4x100 米接力賽中獲勝。  
這位 26 歲的選手現在已經在世界錦標賽上獲得了 8 枚金牌。  
牙買加女子也在接力賽中擊敗法國奪得金牌。

#### 飛馬座

博爾特贏得世錦賽的第三枚金牌。  
幫助牙買加隊在男子 4x100 米接力賽中獲勝。  
博爾特在世錦賽上的第八枚金牌。  
牙買加還贏得女子 4x100 米接力。

通過查看模型輸出，我們注意到的第一件事是 GPT-2 生成的摘要與其他摘要完全不同。它不是對文本進行總結，而是對人物進行總結。GPT-2 模型通常會“產生幻覺”或捏造事實，因為它沒有經過明確的訓練來生成真實的摘要。例如，在撰寫本文時，內斯塔還不是世界上跑得最快的人，而是排在第九位。

將其他三個模型摘要與基本事實進行比較，我們發現存在顯著的重疊，PEGASUS 的輸出具有最驚人的相似性。

現在我們已經檢查了幾個模型，讓我們試著決定在生產環境中使用哪一個。所有四個模型似乎都提供了定性合理的結果，我們可以生成更多示例來幫助我們做出決定。但是，這不是確定最佳模型的系統方法！理想情況下，我們會定義一個

metric，在一些基準數據集上對所有模型進行測量，然後選擇性能最佳的模型。但是如何定義文本生成的度量標準呢？我們所見的標準指標，如準確性、召回率和精確率，並不容易應用於此任務。對於人類編寫的每個“黃金標準”摘要，許多其他帶有同義詞、釋義或稍微不同的表述事實方式的摘要也同樣可以接受。

在下一節中，我們將了解一些為衡量生成文本的質量而開發的常用指標。

## 衡量生成文本的質量

良好的評估指標很重要，因為我們使用它們來衡量模型的性能，不僅在我們訓練它們時，而且在後來的生產中。如果我們的指標不好，我們可能對模型退化視而不見。如果它們與業務目標不一致，我們可能無法創造任何價值。

衡量文本生成任務的性能並不像標準分類任務（例如情感分析或命名實體識別）那麼容易。以翻譯為例；給出一個像“我愛狗！”這樣的句子用英文翻譯成西班牙文可能有多種有效的可能性，比如“¡Me encantan los perros!”或者“¡Me gustan los perros!”簡單地檢查與參考翻譯的精確匹配並不是最優的；即使是人類在這樣的指標上也會表現不佳，因為我們寫的文本彼此之間略有不同（甚至與我們自己不同，這取決於一天或一年的時間！）。幸運的是，還有其他選擇。

用於評估生成的文本的兩個最常見的指標是 BLEU 和 ROUGE。讓我們來看看它們是如何定義的。

### 藍隊

BLEU 的想法很簡單<sup>4</sup>：我們不是查看生成的文本中有多少標記與參考文本標記完全對齊，而是查看單詞或 n-gram。BLEU 是一種基於精度的度量，這意味著當我們比較兩個文本時，我們計算參考中出現的生成中的單詞數，並將其除以參考的長度。

但是，這種原始精度存在問題。假設生成的文本一遍又一遍地重複同一個詞，並且這個詞也出現在參考文獻中。如果重複的次數與參考文本的長度一樣多，那麼我們得到

---

<sup>4</sup> K. Papineni 等人，“BLEU：機器翻譯的自動評估方法”，計算語言學協會第 40 屆年會論文集（2002 年 7 月）：311–318，<http://dx.doi.org/10.3115/1073083.1073135>。

完美的精度 !出於這個原因 ,BLEU 論文的作者引入了一個細微的修改 :一個詞只計算它在參考文獻中出現的次數 。為了說明這一點 ,假設我們有參考文本 “the cat is on the mat”和生成的文本 “the the the the the the” 。

從這個簡單的例子中 ,我們可以計算精度值如下 :

$$\text{香草} = \frac{2}{6}$$

$$pmod = \frac{2}{6}$$

我們可以看到簡單的修正產生了一個更合理的值 。現在讓我們通過不僅查看單個單詞 ,還查看 n-gram 來擴展它 。

假設我們有一個生成的句子 snt ,我們想將其與參考句子 snt' 進行比較 。我們提取所有可能的 n 次 n-grams 並進行計算以獲得精度 pn :

$$pn = \frac{\sum_{n\text{-gram}} \text{sntCount} \cdot \min(\text{snt}' \cdot \text{n-gram}, pn)}{\sum_{n\text{-gram}} \text{snt}' \cdot \text{n-gram}}$$

為了避免獎勵重複生成 ,分子中的計數被剪掉了 。這意味著 n-gram 的出現次數上限為它在參考句子中出現的次數 。另請注意 ,在這個等式中 ,句子的定義不是很嚴格 ,如果生成的文本跨越多個句子 ,您會將其視為一個句子 。

一般來說 ,我們要評估的測試集中有多個樣本 ,因此我們需要通過對語料庫 C 中的所有樣本求和來稍微擴展等式 :

$$pn = \frac{\sum_{snt \in C} \sum_{n\text{-gram}} \text{sntCount} \cdot \min(\text{snt}' \cdot \text{n-gram}, pn)}{\sum_{snt \in C} \sum_{n\text{-gram}} \text{snt}' \cdot \text{n-gram}}$$

我們快到了 。由於我們不考慮召回率 ,因此與較長的句子相比 ,所有生成的短而精確的序列都有好處 。因此 ,精度得分有利於短代 。為了補償 BLEU 的作者引入了一個額外的術語 ,即簡潔懲罰 :

$$BR = \min(1, e^{(1 - \ell_{ref}^{\ell_{gen}})^2})$$

通過取最小值 ,我們確保這個懲罰永遠不會超過 1 ,並且當生成的文本 lgen 的長度小於參考文本 lref 時 ,指數項呈指數級變小 。此時您可能會問 ,為什麼我們不也使用 F1 分數之類的東西來考慮召回率 ?答案是通常在翻譯數據集中有多個參考句子 ,而不是只有一個 ,所以如果我們也測量召回率 ,我們會激勵使用所有參考中的所有單詞的翻譯 。因此 ,最好在翻譯中尋找高精度並確保翻譯和參考具有相似的長度 。

最後 ,我們可以將所有內容放在一起並得出 BLEU 分數的等式 :

$$\text{BLEU-N} = \text{BR} \times \left( \prod_{n=1}^N p_n \right)^{1/N}$$

最後一項是修正精度的幾何平均值 ,直到 n-gram N 。在實踐中 ,通常報告 BLEU-4 分數 。但是 ,您可能已經看到該指標有很多局限性 ;例如 ,它沒有考慮同義詞 ,並且推導中的許多步驟看起來像是臨時的且相當脆弱的啟發式方法 。您可以在 Rachel Tatman 的博客文章 “[Evaluating Text Output in NLP: BLEU at Your Own Risk](#)” 中找到對 BLEU 缺陷的精彩闡述 。

總的來說 ,文本生成領域仍在尋找更好的評估指標 ,而尋找克服 BLEU 等指標限制的方法是一個活躍的研究領域 。BLEU 指標的另一個弱點是它期望文本已經被標記化 。如果不使用完全相同的文本標記化方法 ,這可能會導致不同的結果 。SacreBLEU 指標通過內部化標記化步驟解決了這個問題 ;因此 ,它是基準測試的首選指標 。

我們現在已經研究了一些理論 ,但我們真正想做的是計算一些生成文本的分數 。這是否意味著我們需要在 Python 中實現所有這些邏輯 ?不用擔心 ,Datasets 還提供指標 !加載指標就像加載數據集一樣工作 : 😊

[從數據集導入 load\\_metric](#)

```
bleu_metric = load_metric('sacrebleu')
```

bleu\_metric 對像是 Metric 類的一個實例 ,其工作方式類似於聚合器 :您可以使用 add() 添加單個實例或通過 add\_batch() 添加整個批次 。一旦您添加了所有需要評估的樣本 ,您就可以調用 compute() 併計算指標 。這將返回一個包含多個值的字典 ,例如每個 n-gram 的精度、長度懲罰以及最終的 BLEU 分數 。讓我們看一下之前的例子 :

將pandas導入為pd將numpy  
導入為np

```
bleu_metric.add(prediction= "the the the the the", reference=[ "貓在墊子上"])
結果= bleu_metric.compute(smooth_method= "floor", smooth_value=0)
results[ "precisions" ] = [np.round(p, 2) for p in results[ "precisions" ]]
pd.DataFrame.from_dict(results,方向= "索引",列= [ "價值" ])
```

	價值
分數	0.0
計數	[2, 0, 0, 0]
總數	[6, 5, 4, 3]
精度[33.33, 0.0, 0.0, 0.0]	
血壓	1.0
系統長度	4
參考長度	4



如果有多個參考翻譯，BLEU 分數也有效。這就是引用作為列表傳遞的原因。為了使 n-gram 中零計數的度量更平滑，BLEU 集成了修改精度計算的方法。一種方法是給分子加上一個常數。這樣，丟失的 n-gram 不會導致分數自動歸零。為了解釋這些值，我們通過設置smooth\_value=0 來關閉它。

我們可以看到 1-gram 的精度確實是 2/6，而 2/3/4-gram 的精度都是 0。（有關單個指標的更多信息，如計數和 bp，請參閱 SacreBLEU 存儲庫。）這意味著幾何平均值為零，因此 BLEU 分數也為零。讓我們看另一個預測幾乎正確的例子：

```
bleu_metric.add(prediction= "貓在墊子上", reference=[ "貓在墊子上"])
結果= bleu_metric.compute(smooth_method= "floor", smooth_value=0)
results[ "precisions" ] = [np.round(p, 2) for p in results[ "precisions" ]]
pd.DataFrame.from_dict(results,方向= "索引",列= [ "價值" ])
```

	價值
分數	57.893007
計數	[5, 3, 2, 1]
總數	[5, 4, 3, 2]
精度[100.0, 75.0, 66.67, 50.0]	

	價值
血壓	0.818731
sys_len	•••
參考長度	•••

我們觀察到精度分數要好得多。預測中的 1-grams 都匹配，只有在精度分數中我們才能看到有問題。對於 4-gram 只有兩個候選，[ the , cat , is , on ] 和 [ cat , is , on , mat ]，最後一個一個不匹配，因此精度為 0.5。

BLEU 分數廣泛用於評估文本，尤其是在機器翻譯中，因為精確翻譯通常比包含所有可能和適當單詞的翻譯更受青睞。

還有其他應用程序，例如匯總，情況有所不同。

在那裡，我們想要生成文本中的所有重要信息，因此我們支持高召回率。這是通常使用 ROUGE 分數的地方。

## 胭脂

ROUGE 分數是專門為摘要等應用開發的，在這些應用中，高召回率比精度更重要。該方法與 BLEU 分數非常相似，因為我們查看不同的 n-gram 並比較它們在生成的文本和參考中的出現文本。不同之處在於，對於 ROUGE，我們檢查參考文本中有多少 n-gram 也出現在生成的文本中。對於 BLEU，我們查看了生成文本中有多少 n-gram 出現在參考中，因此我們可以重用精確度公式，並稍作修改，計算分子中生成文本中參考 n-gram 的（未剪切）出現次數：

$$\text{胭脂-N} = \frac{\sum_{\text{snt}} C \sum_{\text{n-gram}} \text{snt}' \text{Countmatch}(\text{n-gram})}{\sum_{\text{snt}} C \sum_{\text{n-gram}} \text{snt}' \text{計數}(\text{n-gram})}$$

這是 ROUGE 的最初提案。隨後，研究人員發現完全去除精度會產生強烈的負面影響。回到沒有剪裁計數的 BLEU 公式，我們也可以測量精度，然後我們可以結合調和平均數中的精度和召回率 ROUGE 分數以獲得 F1 分數。該分數是當今 ROUGE 普遍報告的指標。

---

5元。Lin，“ROUGE：自動評估摘要包”，Text Summarization Branches Out（2004年7月），<https://aclanthology.org/W04-1013.pdf>。

ROUGE 中有一個單獨的分數來衡量最長公共子串 (LCS) ,稱為 ROUGE-L 。可以為任何一對字符串計算 LCS 。例如，“abab”和“abc”的 LCS 是“ab”，它的長度是 2 。如果我們想比較兩個樣本之間的這個值，我們需要以某種方式對其進行歸一化，否則更長的文本將是處於優勢。為了實現這一點，ROUGE 的發明者提出了一種類似 F-score 的方案，其中 LCS 使用參考和生成文本的長度進行歸一化，然後將兩個歸一化分數混合在一起：

$$\text{RLCS} = \frac{\text{滿海戰鬥艦 } X, Y}{\text{米}}$$

$$\text{可編程邏輯控制器} = \frac{\text{滿海戰鬥艦 } X, Y}{n}$$

$$\text{後勤保障計劃} = \frac{(1 + \beta)^{\text{RLCSPLCS}}}{\text{RLCS} + \beta \text{PLCS}}, \text{ 其中 } \beta = \text{PLCS}/\text{RLCS}$$

這樣，LCS 分數就會被適當地歸一化，並且可以跨樣本進行比較。

在 Datasets 實現中，計算了 ROUGE 的兩種變體：一種計算每個句子的分數並對摘要進行平均 (ROUGE-L)，另一種直接在整個摘要上計算 (ROUGE-Lsum)。

我們可以按如下方式加載指標：

```
rouge_metric = load_metric('rouge')
```

我們已經用 GPT-2 和其他模型生成了一組摘要，現在我們有了一個指標來系統地比較這些摘要。讓我們將 ROUGE 分數應用於模型生成的所有摘要：

```
reference = dataset['train'][1]['highlights']
records = []
rouge_names = ['rouge1', 'rouge2', 'rougeL',
               'rougeLsum']
```

對於摘要中的model\_name：

```
rouge_metric.add(prediction=summaries[model_name], reference=reference)
score = rouge_metric.compute()
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names)
records.append(rouge_dict)
pd.DataFrame.from_records(records, index=summaries.keys())
```

	胭脂1	胭脂2	胭脂L	rougeLsum
基線	0.303571	0.090909	0.214286	0.232143
gpt2	0.187500	0.000000	0.125000	0.187500

	胭脂1	胭脂2	胭脂L	rougeLsum
t5	0.486486	0.222222	0.378378	0.486486
巴特	0.582278	0.207792	0.455696	0.506329
飛馬座	0.866667	0.655172	0.800000	0.833333



Datasets 庫中的 ROUGE 指標還計算置信區間（默認情況下，第 5 個和第 95 個百分位數）。平均值存儲在屬性mid中，可以使用low和high檢索間隔。

這些結果顯然不是很可靠，因為我們只查看了一個樣本，但我們可以比較該樣本的摘要質量。該表證實了我們的觀察，即在我們考慮的模型中，GPT-2 表現最差。這並不奇怪，因為它是該組中唯一沒有明確訓練進行總結的模型。然而，令人驚訝的是，與具有大約十億個參數的 transformer 模型相比，簡單的前三句話基線並沒有太差！PEGASUS 和 BART 是總體上最好的模型（ROUGE 分數越高越好），但 T5 在 ROUGE-1 和 LCS 分數上略好。

這些結果將 T5 和 PEGASUS 列為最佳模型，但同樣應謹慎對待這些結果，因為我們僅在單個示例中評估模型。

查看 PEGASUS 論文中的結果，我們預計 PEGASUS 在 CNN/DailyMail 數據集上的表現優於 T5。

讓我們看看我們是否可以用 PEGASUS 重現這些結果。

## 在 CNN/DailyMail 數據集上評估 PEGASUS

我們現在已經準備好正確評估模型的所有部分：我們有一個帶有來自 CNN/DailyMail 的測試集的數據集，我們有一個帶有 ROUGE 的度量，我們有一個摘要模型。我們只需要將各個部分放在一起。讓我們首先評估三句基線的性能：

```
def evaluate_summaries_baseline(dataset, metric,
                                 column_text= article ,
                                 column_summary= highlights ):
    summaries = [three_sentence_summary(text) for text in dataset[column_text]]
    metric.add_batch(predictions=summaries,
                      references=dataset[column_summary]) score
    = metric.compute()返回分數
```

現在我們將該函數應用於數據的一個子集。由於 CNN/DailyMail 數據集的測試部分包含大約 10,000 個樣本，因此為所有這些文章生成摘要需要花費大量時間。回想一下第 5 章，每個生成的令牌

需要向前通過模型；因此，為每個樣本生成 100 個標記將需要 100 萬次前向傳播。如果我們使用波束搜索，這個數字將乘以波束的數量。為了保持計算速度相對較快，我們將對測試集進行二次抽樣，然後對 1,000 個樣本進行評估。這應該會給我們一個更穩定的分數估計，同時在 PEGASUS 模型的單個 GPU 上完成不到一小時：

```
test_sampled = 數據集[“測試”].shuffle(seed=42).select(range(1000))

分數= evaluate_summaries_baseline(test_sampled, rouge_metric) rouge_dict = dict((rn,
score[rn].mid.fmeasure) for rn in rouge_names) pd.DataFrame.from_dict(rouge_dict,
orient= index ,columns=[ baseline ]).T
```

	胭脂1	胭脂2	胭脂L	rougeLsum
基線	0.396061	0.173995	0.245815	0.361158

分數大多比前面的例子差，但仍然比 GPT-2 取得的分數好！現在讓我們實現相同的評估函數來評估 PEGASUS 模型：

從tqdm導入tqdm導入火炬

```
device = cuda 如果torch.cuda.is_available()否則 cpu

def chunks(list_of_elements, batch_size):    從
    list_of_elements 中產生連續的批量大小的塊。    for i in range(0, len(list_of_elements),
    batch_size):                                yield list_of_elements[i : i + batch_size]

def evaluate_summaries_pegasus(dataset, metric, model, tokenizer, batch_size=16,
                                 device=device, column_text= article ,
                                 column_summary= highlights ):
    article_batches = list(chunks(dataset[column_text],
batch_size)) target_batches = 列表(塊 (數據集[column_summary]， batch_size) )

    對於tqdm中的article_batch、target_batch ( zip
        (article_batches, target_batches) ， total=len (article_batches) ) :

        inputs = tokenizer(article_batch, max_length=1024, truncation=True, padding= max_length ,
                           return_tensors= pt )

        摘要= model.generate(input_ids=inputs[ input_ids ].to(device),
                            attention_mask=inputs[ attention_mask ].to(device),
                            length_penalty=0.8, num_beams=8, max_length=128)

        decoded_summaries = [tokenizer.decode(s, skip_special_tokens=True,
                                             clean_up_tokenization_spaces=True)
    對於摘要中的s ]
```

```

    decoded_summaries = [d.replace( <n> , ) for d in decoded_summaries]
    metric.add_batch(predictions=decoded_summaries, references=target_batch)

score = metric.compute()  
返回分  
數

```

讓我們稍微解壓一下這個評估代碼。首先，我們將數據集拆分為可以同時處理的較小批次。然後對於每個批次，我們將輸入文章標記化並將它們提供給generate()函數以使用集束搜索生成摘要。我們使用與本文中提出的相同的生成參數。長度懲罰的新參數確保模型不會生成太長的序列。最後，我們對生成的文本進行解碼，替換<n>標記，並將解碼後的文本添加到對度量的引用中。最後，我們計算並返回 ROUGE 分數。現在讓我們使用用於 seq2seq 生成任務的 AutoModelFor Seq2SeqLM 類再次加載模型，並對其進行評估：

[從變壓器導入AutoModelForSeq2SeqLM， AutoTokenizer](#)

```

model_ckpt = google/pegasus-cnn_dailymail tokenizer
= AutoTokenizer.from_pretrained(model_ckpt) model =
AutoModelForSeq2SeqLM.from_pretrained(model_ckpt).to(device)分數=
evaluate_summaries_pegasus(test_sampled, rouge_metric,
                             模型'分詞器， batch_size=8) rouge_dict =
dict((rn, score[rn].mid.fmeasure) for rn in rouge_names) pd.DataFrame(rouge_dict,
index=[ pegasus ])

```

	胭脂1	胭脂2	rougeL	rougeLsum
pegasus	0.434381	0.210883	0.307195	0.373231

這些數字與公佈的結果非常接近。這裡要注意的一件事是，損失和每個標記的準確性在某種程度上與 ROUGE 分數脫鉤。損失與解碼策略無關，而 ROUGE 分數是強耦合的。

由於 ROUGE 和 BLEU 與人類判斷的相關性高於損失或準確性，因此我們應該關注它們並在構建文本生成模型時仔細探索和選擇解碼策略。然而，這些指標遠非完美，人們也應該始終考慮人類的判斷。

現在我們已經配備了評估功能，是時候訓練我們自己的模型進行總結了。

## 訓練摘要模型

我們已經研究了很多關於文本摘要和評估的細節，所以讓我們用它來訓練自定義文本摘要模型！對於我們的應用程序，我們將使用SAMSum 數據集由 Samsung 開發，由一系列對話和簡短摘要組成。在企業環境中，這些對話可能代表客戶與支持中心之間的交互，因此生成準確的摘要有助於改善客戶服務並檢測客戶請求中的常見模式。讓我們加載它並看一個例子：

```
dataset_samsum = load_dataset('samsum')
split_lengths = [len(dataset_samsum[split]) for split in dataset_samsum]

打印(f“分割長度 :{split_lengths}” )打印(f“特徵：
{dataset_samsum[‘train’].column_names}” )打印(“\n對話：” )打印
(dataset_samsum[“測試”][0][“對話”])打印(“\n總結:” )打印
(dataset_samsum[“測試”][0][“總結”])
```

拆分長度 :[14732 819 818]  
特徵 :[ id , 對話 , 摘要 ]

對話：漢娜：  
嘿，你有貝蒂的電話號碼嗎？  
阿曼達：讓我檢查一下 漢娜：  
<file\_gif> 阿曼達：抱歉，找  
不到。  
阿曼達：問問拉里 阿曼達：  
上次我們一起去公園的時候他給她打過電話 漢娜：我不太了解他 漢娜：<file\_gif> 阿曼達：別  
害羞，他很好 漢娜：如果你這麼說..

漢娜：我寧願你給他發短信 阿曼達：給他發短信 漢  
娜：呃……好吧 漢娜：再見 阿曼達：再見

簡介：漢娜  
需要貝蒂的電話號碼，但阿曼達沒有。她需要聯繫拉里。

對話看起來就像您通過 SMS 或 WhatsApp 聊天時所期望的那樣，包括表情符號和 GIF 占位符。對話字段包含全文和摘要對話。在 CNN/DailyMail 數據集上微調的模型可以處理這個問題嗎？讓我們找出答案！

## 在 SAMSum 上評估 PEGASUS

首先，我們將使用 PEGASUS 運行相同的匯總管道，以查看輸出結果。我們可以重用我們用於 CNN/DailyMail 摘要生成的代碼：

```
pipe_out = pipe(dataset_samsum[ test ][0][ dialogue ])
print( Summary: ) print(pipe_out[0][ summary_text ].replace( .<n> ,
    \n ))
```

Summary:

Amanda: Ask Larry Amanda: 上次我們一起去公園的時候他給她打了電話。

漢娜：我寧願你給他發短信。

阿曼達：給他發短信

我們可以看到該模型主要嘗試通過從對話中提取關鍵句子來進行總結。這在 CNN/DailyMail 數據集上的效果可能相對較好，但 SAMSum 中的摘要更加抽象。讓我們通過在測試集上運行完整的 ROUGE 評估來確認這一點：

```
score = evaluate_summaries_pegasus(dataset_samsum[ test ], rouge_metric=模型,
                                    tokenizer, column_text= 對話 ,
                                    column_summary= 摘要 , batch_size=8)

rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names) pd.DataFrame(rouge_dict,
index=[ pegasus ])
```

	胭脂1	胭脂2	胭脂L rougeLsum
飛馬座	0.296168	0.087803	0.229604 0.229514

嗯，結果不是很好，但這並不意外，因為我們已經遠離 CNN/DailyMail 數據分發。然而，在訓練之前設置評估管道有兩個優點：我們可以直接用指標衡量訓練的成功，並且我們有一個很好的基線。在我們的數據集上微調模型應該會立即改善 ROUGE 指標，如果不是這種情況，我們就會知道我們的訓練循環有問題。

## 微調 PEGASUS

在我們處理訓練數據之前，讓我們快速看一下輸入和輸出的長度分佈：

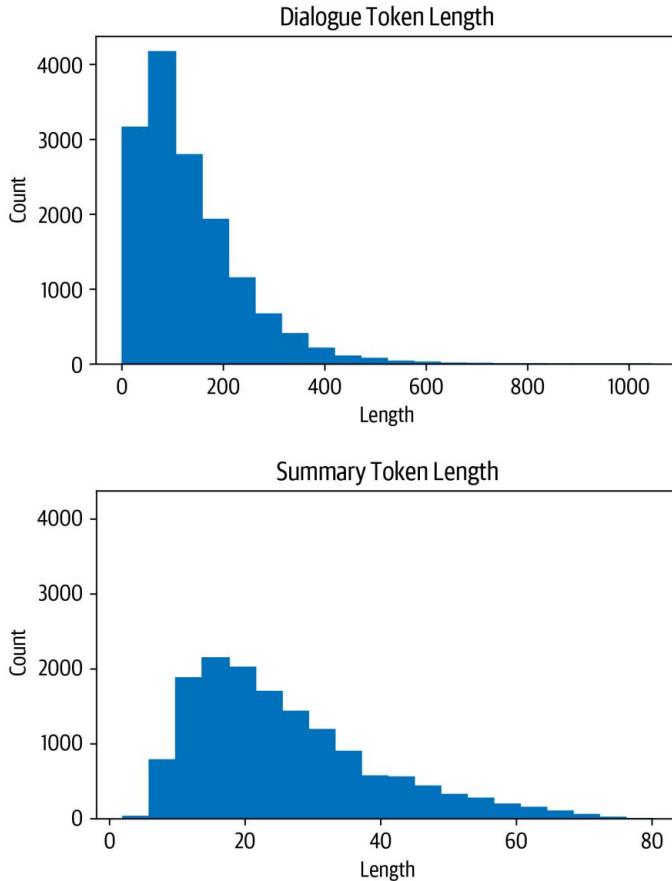
```
d_len = [len(tokenizer.encode(s)) for s in dataset_samsum[ train ][ dialogue ]]
s_len = [len(tokenizer.encode(s))
for s in dataset_samsum[ train ][ "概括" ]]

圖， axes = plt.subplots(1, 2, figsize=(10, 3.5), sharey=True) axes[0].hist(d_len, bins=20,
color= C0 , edgecolor= C0 )軸[0].set_title( Dialogue Token Length )軸
[0].set_xlabel( 長度 )
```

```

axes[0].set_ylabel( Count )
axes[1].hist(s_len, bins=20, color= C0 , edgecolor= C0 )
axes[1].set_title( Summary Token Length )軸[1].set_xlabel( 長度 )
plt.tight_layout() plt.show()

```



我們看到大多數對話比 CNN/DailyMail 文章短得多，每個對話有 100-200 個標記。同樣，摘要要短得多，大約有 20-40 個標記（一條推文的平均長度）。

在為 Trainer 構建數據整理器時，讓我們牢記這些觀察結果。

首先，我們需要標記數據集。現在，我們將對話和摘要的最大長度分別設置為 1024 和 128：

```

def convert_examples_to_features (example_batch) :
    input_encodings = tokenizer(example_batch[ 對話 ], max_length=1024,截斷=真)

```

```

使用tokenizer.as_target_tokenizer() :
    target_encodings = tokenizer(example_batch[ "summary" ], max_length=128,
                                  截斷=真)

    返回[ input_ids :input_encodings[ "input_ids" ],
           attention_mask :input_encodings[ "attention_mask" ], labels :
           target_encodings[ "input_ids" ]]

dataset_samsum_pt = dataset_samsum.map(convert_examples_to_features,
                                         批處理=真)
columns = [ input_ids , labels , attention_mask ]
dataset_samsum_pt.set_format(type= torch , columns=columns)

```

使用標記化步驟的一個新事物是tokenizer.as\_target\_tokenizer()上下文。一些模型在解碼器輸入中需要特殊標記，因此區分編碼器和解碼器輸入的標記化很重要。在with語句（稱為上下文管理器）中，分詞器知道它正在為解碼器進行分詞，並可以相應地處理序列。

現在，我們需要創建數據整理器。就在批處理通過模型之前，在訓練器中調用此函數。在大多數情況下，我們可以使用默認的收集器，它從批處理中收集所有張量並簡單地堆疊它們。對於摘要任務，我們不僅需要堆疊輸入，還需要在解碼器端準備目標。PEGASUS 是一種編碼器-解碼器轉換器，因此具有經典的 seq2seq 架構。在 seq2seq 設置中，一種常見的方法是在解碼器中應用“teacher forcing”。通過這種策略，解碼器接收輸入令牌（就像在 GPT-2 等解碼器模型中一樣），除了編碼器輸出之外，這些輸入令牌還包含移動 1 的標籤；因此，在對下一個標記進行預測時，解碼器將 ground truth 移動 1 作為輸入，如下表所示：

	解碼器輸入	標籤
步		
0	[軟墊]	變形金剛
1	[PAD、變形金剛]	是
2	[PAD、變形金剛、都是]	驚人的
3	【PAD、變形金剛、都、棒棒噠】	為了
4	[PAD, 變形金剛, are, awesome, for]	文本
5	【PAD、變形金剛、are、awesome、for、text】	總結

我們將其移動一位，以便解碼器只能看到以前的地圖真值標籤，而看不到當前或未來的標籤。僅移位就足夠了，因為解碼器已經屏蔽了自我注意，屏蔽了當前和未來的所有輸入。

因此，當我們準備批次時，我們通過將標籤向右移動一位來設置解碼器輸入。之後，我們通過將它們設置為 -100 來確保標籤中的填充標記被損失函數忽略。不過，我們實際上不必手動執行此操作，因為 DataCollatorForSeq2Seq 來拯救並為我們處理所有這些步驟：

[從變壓器導入DataCollatorForSeq2Seq](#)

```
seq2seq_data_collator = DataCollatorForSeq2Seq(分詞器, 模型=模型)
```

然後，像往常一樣，我們設置一個用於訓練的 TrainingArguments：

[從變形金剛導入TrainingArguments, Trainer](#)

```
training_args = TrainingArguments(
    output_dir= pegasus-samsum , num_train_epochs=1, warmup_steps=500,
    per_device_train_batch_size=1, per_device_eval_batch_size=1, weight_decay=0.01,
    logging_steps=10, push_to_hub=True, evaluation_strategy= steps , eval_steps=500,
    save_steps=1e6, gradient_accumulation_steps=16)
```

與之前設置不同的一件事是新參數 gradient\_accumulation\_steps。由於模型很大，我們不得不將批量大小設置為 1。但是，批量大小太小會影響收斂。為了解決這個問題，我們可以使用一種稱為梯度累積的巧妙技術。顧名思義，我們不是一次計算整個批次的梯度，而是生成更小的批次並聚合梯度。當我們聚合了足夠的梯度時，我們運行優化步驟。當然，這比一次完成要慢一點，但它為我們節省了大量 GPU 內存。

現在讓我們確保我們已登錄到 Hugging Face，以便我們可以在訓練後將模型推送到 Hub：

[從huggingface\\_hub導入notebook\\_login](#)

```
notebook_login()
```

我們現在擁有使用模型、分詞器、訓練參數和數據整理器以及訓練和評估集初始化訓練器所需的一切：

```
培訓師=培訓師(模型=模型, args=training_args,
                  tokenizer=分詞器, data_collator=seq2seq_data_collator,
                  train_dataset=dataset_samsun_pt[ train ],
                  eval_dataset=dataset_samsun_pt[ validation ])
```

我們準備好訓練了。訓練完成後，我們可以直接在測試集上運行評估函數，看看模型的表現如何：

```
trainer.train() score
=
evaluate_summaries_pegasus(dataset_samsun[ test ], rouge_metric, trainer.model,
                           tokenizer, batch_size=2, column_text= 對話 , column_summary= 摘要 )
```

```
rouge_dict = dict((rn, score[rn].mid.fmeasure) for rn in rouge_names) pd.DataFrame(rouge_dict,
index=[f' pegasus '])
```

	胭脂1	胭脂2	rougeL	rougeLsum
pegasus	0.427614	0.200571	0.340648	0.340738

我們看到 ROUGE 分數比沒有微調的模型有很大提高，因此即使以前的模型也接受了摘要訓練，但它並不能很好地適應新領域。讓我們將模型推送到 Hub：

```
trainer.push_to_hub( 訓練完成！ )
```

在下一節中，我們將使用該模型為我們生成一些摘要。



您還可以將生成作為訓練循環的一部分進行評估：使用名為Seq2Seq Training Arguments的TrainingArguments擴展並指定predict\_with\_generate=True，將其傳遞給名為Seq2SeqTrainer的專用訓練器，然後它使用generate()函數而不是模型的前向傳遞來創建評估預測。試一試！

## 生成對話摘要查看損失和 ROUGE 分

數，該模型似乎比僅在 CNN/DailyMail 上訓練的原始模型有了顯著改進。讓我們看看對測試集中的樣本生成的摘要是什麼樣子的：

```
gen_kwargs = { length_penalty : 0.8, num_beams : 8, max_length : 128} sample_text =
dataset_samsum[ test ][0][ dialogue ] reference = dataset_samsum[ test ][0][
summary ] pipe = pipeline( summarization ,model= transformersbook/pegasus-samsum )
```

```
print( Dialogue: )
print(sample_text)
print( \nReference Summary: )
print(reference) print( \nModel
Summary: ) print(pipe(sample_text,
**gen_kwargs)[0][ summary_text ])
```

對話：漢娜：  
嘿，你有貝蒂的電話號碼嗎？  
阿曼達：讓我檢查一下 漢娜：  
<file\_gif> 阿曼達：抱歉，找  
不到。  
阿曼達：問問拉里 阿曼達：  
上次我們一起去公園的時候他給她打過電話 漢娜：我不太了解他 漢娜 <file\_gif>

阿曼達 :別害羞 ,他人很好 漢娜 :如果你這麼說...

漢娜 :我寧願你給他發短信 阿曼達 :給他發短信 漢  
娜 :呃……好吧 漢娜 :再見 阿曼達 :再見

參考摘要 :漢娜需要貝蒂的  
電話號碼 ,但阿曼達沒有 。她需要聯繫拉里。

模特概要 :阿曼達找  
不到貝蒂的電話號碼 。上次他們一起在公園時 ,拉里給貝蒂打了電話 。漢娜希望阿曼達給拉里發短信 ,而不是給貝蒂打電話 。

這看起來更像是參考摘要 。該模型似乎已經學會了將對話合成為摘要 ,而不僅僅是  
提取段落 。現在 ,終極測試 :模型在自定義輸入上的表現如何 ?

```
custom_dialogue =      \
Thom :大家好 ,你們聽說過變形金剛嗎 ?  
劉易斯 :是的 ,我最近用過它們 !  
Leandro :的確 ,Hugging Face 有一個很棒的圖書館 。  
湯姆 :我知道 ,我幫助建造了它 ; )  
劉易斯 :酷 ,也許我們應該寫一本關於它的書 。你怎麼認為 ?  
Leandro :好主意 ,這能有多難 ? !  
湯姆 :我在 !  
劉易斯 :太棒了 ,讓我們一起來吧 !
```

打印 (管道 (自定義對話 , \*\*gen\_kwarg)[0][“summary\_text”])

Thom 和 Lewis 和 Leandro 打算寫一本關於變形金剛的書 。Thom 通過 Hugging Face 幫助建立了一個圖書館 。他們  
將一起做 。

生成的自定義對話摘要很有意義 。它很好地總結了所有參與討論的人都想一起寫這  
本書 ,而不是簡單地提取單個句子 。例如 ,它將第三行和第四行合成為一個邏輯組  
合 。

## 結論

與其他可以定義為分類任務的任務 (如情感分析、命名實體識別或問答)相比 ,文本  
摘要提出了一些獨特的挑戰 。諸如準確性之類的常規指標並不能反映生成文本的質  
量 。正如我們所見 ,BLEU 和 ROUGE 指標可以更好地評估生成的文本 ;然而 ,人的  
判斷仍然是最好的衡量標準 。

使用摘要模型時的一個常見問題是我們如何摘要文本長於模型上下文長度的文檔 。

不幸的是，沒有解決這個問題的單一策略，迄今為止這仍然是一個開放和活躍的研究問題。例如，OpenAI 最近的工作展示了如何通過遞歸地將摘要應用於長文檔並在循環中使用人工反饋來擴展摘要。<sup>6</sup>

在下一章中，我們將研究問答，即根據文本段落提供問題答案的任務。與摘要相比，對於此任務，存在處理長文檔或多文檔的好策略，我們將向您展示如何將問答擴展到數千個文檔。

---

<sup>6</sup> J. Wu 等人，“利用人類反饋遞歸總結書籍”，(2021)。

## 第7章

## 問答

無論您是研究人員、分析師還是數據科學家，在某些時候您都可能需要在文檔的海洋中跋涉才能找到您正在尋找的信息。更糟糕的是，Google 和 Bing 不斷提醒您存在更好的搜索方式！例如，如果我們搜索“居里夫人甚麼時候獲得她的第一個諾貝爾獎？”在 Google 上，我們立即得到正確答案“1903”，如圖7-1所示。

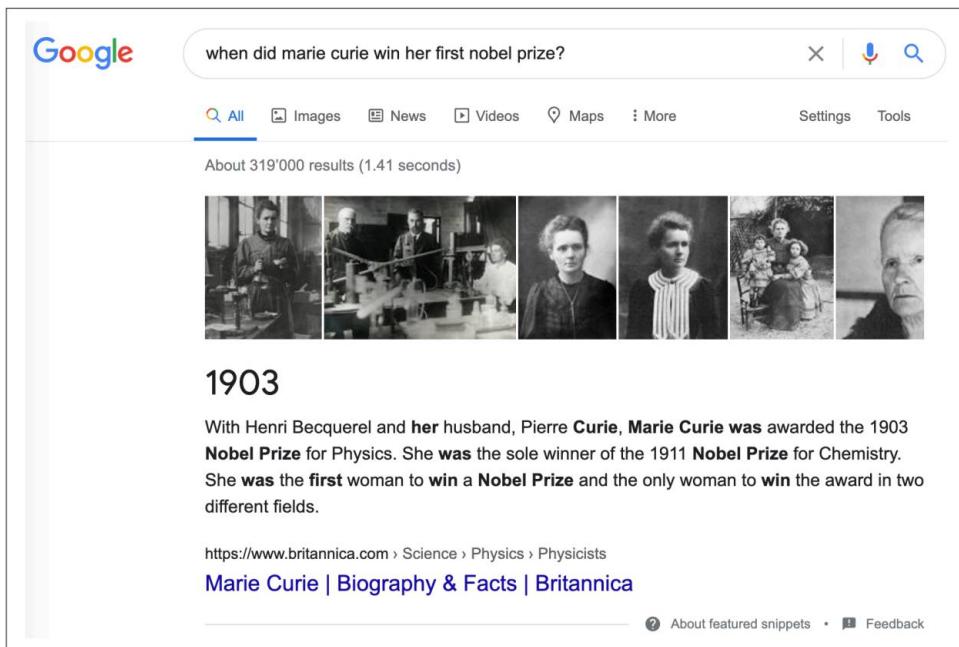


圖 7-1。Google 搜索查詢和相應的答案片段

在這個例子中，谷歌首先檢索了大約 319,000 個與查詢相關的文檔，然後執行了一個額外的處理步驟來提取帶有相應段落和網頁的答案片段。不難看出為什麼這些答案片段很有用。例如，如果我們搜索一個更棘手的問題，例如“哪種吉他調音最好？”Google 不提供答案，我們必須點擊搜索引擎返回的網頁之一才能自己找到它。<sup>1</sup>

這項技術背後的一般方法稱為問答 (QA)。

QA 有很多種，但最常見的是提取式 QA，它涉及的問題的答案可以識別為文檔中的一段文本，其中文檔可能是網頁、法律合同或新聞文章。首先檢索相關文檔，然後從中提取答案的兩階段過程也是許多現代 QA 系統的基礎，包括語義搜索引擎、智能助手和自動信息提取器。在本章中，我們將應用此過程來解決電子商務網站面臨的一個常見問題：幫助消費者回答特定查詢以評估產品。我們將看到客戶評論可以用作 QA 的豐富且具有挑戰性的信息來源，並且在此過程中，我們將了解轉換器如何充當強大的閱讀理解模型，可以從文本中提取含義。讓我們從充實用例開始。



本章重點介紹提取式 QA，但其他形式的 QA 可能更適合您的用例。例如，社區 QA 涉及收集用戶在 [Stack Overflow](#) 等論壇上生成的問答對，然後使用語義相似性搜索找到與新問題最接近的匹配答案。還有長篇 QA，旨在為開放式問題生成複雜的段落長度答案，例如“天空為什麼是藍色的？”值得注意的是，還可以對表格和 [TAPAS](#) 等轉換器模型進行 QA，甚至可以執行聚合以產生最終答案！

## 建立基於審查的質量保證系統

如果您曾經在線購買過產品，您可能會依賴客戶評論來幫助您做出決定。這些評論通常可以幫助回答具體問題，例如“這把吉他有背帶嗎？”或者“我可以在晚上使用這台相機嗎？”僅從產品描述可能很難回答。然而，受歡迎的產品可能有成百上千條評論，因此很難找到相關的評論。一種替代方法是在社區 QA 上發布您的問題。

---

<sup>1</sup>儘管在這種特殊情況下，每個人都同意 Drop C 是最好的吉他調音。

亞馬遜等網站提供的平台，但通常需要幾天時間才能得到答案（如果你能得到答案的話）。如果我們能立即得到答案，那不是很好嗎？就像圖 7-1 中的 Google 示例一樣？讓我們看看我們是否可以使用變形金剛來做到這一點！

## 數據集

為了構建我們的 QA 系統，我們將使用 SubjQA 數據集<sup>2</sup>，它包含 10,000 多條關於六個領域的產品和服務的英文客戶評論：Trip-Advisor、餐廳、電影、書籍、電子產品和雜貨。如圖 7-2 所示，每個評論都與一個問題相關聯，該問題可以使用評論中的一個或多個句子來回答。<sup>3</sup>

**Product:** Nokia Lumia 521 RM-917 8GB



**Query:** Why is the camera of poor quality?

**Review:** Item like the picture, fast deliver 3 days well packed, good quality for the price. The camera is decent (as phone cameras go), There is no flash though...

圖 7-2 °關於產品的問題和相應的評論（答案跨度有下劃線）

該數據集的有趣之處在於，大多數問題和答案都是主觀的；也就是說，它們取決於用戶的個人體驗。圖 7-2 中的示例顯示了為什麼此功能使任務可能比

---

2 J. Bjerva 等人，“SubjQA：主觀性和評論理解數據集”，(2020)。

3 我們很快就會看到，還有一些無法回答的問題旨在生成更強大的模型。

尋找事實問題的答案，例如“英國的貨幣是什麼？”首先，查詢是關於“質量差”的，這是主觀的，取決於用戶對質量的定義。其次，查詢的重要部分根本沒有出現在評論中，這意味著它無法通過關鍵字搜索或解釋輸入問題等快捷方式來回答。這些特性使 SubjQA 成為一個現實的數據集，可以作為我們基於評論的 QA 模型的基準，因為如圖 7-2 所示的用戶生成的內容類似於我們在野外可能遇到的內容。



QA 系統通常根據它們在響應查詢時可以訪問的數據域進行分類。封閉域 QA 處理有關狹窄主題（例如，單個產品類別）的問題，而開放域 QA 處理幾乎所有內容（例如，亞馬遜的整個產品目錄）的問題。通常，封閉域 QA 涉及搜索的文檔少於開放域案例。

首先，讓我們從 Hugging Face Hub 下載數據集。正如我們在第 4 章中所做的那樣，我們可以使用 `get_dataset_config_names()` 函數來找出哪些子集可用：

[從數據集導入 `get\_dataset\_config\_names`](#)

```
域 = get_dataset_config_names( subjqa ) 域
```

```
[ 書籍 , 電子產品 , 雜貨店 , 電影 , 餐館 , tripadvisor ]
```

對於我們的用例，我們將專注於為電子領域構建 QA 系統。要下載電子子集，我們只需將此值傳遞給 `load_dataset()` 函數的名稱參數：

[從數據集導入 `load\_dataset`](#)

```
subjqa = load_dataset( subjqa , name= electronics )
```

與 Hub 上的其他問答數據集一樣，SubjQA 將每個問題的答案存儲為嵌套字典。例如，如果我們檢查答案列中的其中一行：

```
print(subjqa[ "火車" ][ "答案" ][1])
{
    text :[ Bass weak as expected , Bass as expected, even with EQ adjusted up ],
    answer_start : [1302, 1302],
    answer_subj_level : [1, 1],
    ans_subj_score : [0.5083333253860474, 0.5083333253860474],
    is_ans_subjective : [真, 真]
}
```

我們可以看到答案存儲在文本字段中，而起始字符索引在 `answer_start` 中提供。為了更輕鬆地探索數據集，我們將展平

使用flatten()方法將這些嵌套列轉換為 Pandas  
數據框如下：

將熊貓導入為pd

```
dfs = {split: dset.to_pandas() for split, dset in subjqa.flatten().items()}
```

對於拆分，dfs.items()中的df：

```
print(f' {split} 中的問題數 :{df[ "id" ].nunique()} )
```

訓練題數 :1295

試題數 :358

驗證問題數 :255

請注意，數據集相對較小，總共只有 1,908 個示例。這模擬了一個真實世界的場景，因為讓領域專家標記提取的 QA 數據集是勞動密集型和昂貴的。例如，用於法律合同提取 QA 的 CUAD 數據集估計價值 200 萬美元，以說明註釋其 13,000 個示例所需的法律專業知識。<sup>4</sup> SubjQA 數據集中有很多列，但最有趣的列用於構建我們的質量保證系統的那些如表 7-1 所示。

表 7-1。SubjQA 數據集中的列名及其描述

列名	描述
標題	與每個產品關聯的亞馬遜標準識別號 (ASIN)
問題	問題
answers.answer_text	註釋器標記的評論中的文本範圍
answers.answer_start	答案範圍的起始字符索引
語境	客戶評價

讓我們關注這些列，並查看一些訓練示例。我們可以使用sample()方法來選擇一個隨機樣本：

```
qa_cols=[ "title" , "question" , "answers.text" ,
          "answers.answer_start" , "context" ]
sample_df=dfs[ "train" ][qa_cols].sample(2, random_state=7) sample_df
```

---

<sup>4</sup> D. Hendrycks 等人，“CUAD：用於法律合同審查的專家註釋 NLP 數據集”，(2021)。

標題	問題	answers.text	answers.answer_start	上下文
B005DKZTMG	做 鍵盤輕 巧？	[這個鍵盤很緊 湊]	[215]	我真的很喜歡這個鍵盤。我給它 4 星，因為它沒有 CAPS LOCK 鍵，所以我永遠不知道我的大寫字母是否打開。但就價格而言，它確實是一款無線鍵盤。我的手很大而且這個鍵盤很緊湊，但我沒有什麼可抱怨的。
B00AAIPT76	電池怎麼樣？	[]	[]	在我購買的第一個備用 gopro 電池無法充電後，我買了這個。我對這類產品抱有非常現實的期望，我對充電時間和電池壽命的驚人故事持懷疑態度，但我確實希望電池至少能充電幾週，並且充電器能像充電器一樣工作。在這一點上我沒有失望。我是河豚，發現 gopro 很快就耗盡了電量，所以這次購買解決了這個問題。電池充電，在短途旅行中，額外的兩個電池就足夠了，在長途旅行中，我可以使用我的朋友 JOOS Orange 為它們充電。我剛買了一個新的 xtreme powerpak，希望能夠用它充電，所以我會不會再沒電了。

從這些例子中，我們可以做出一些觀察。首先，問題的語法不正確，這在電子商務網站的常見問題解答部分很常見。其次，空的answers.text條目表示“無法回答”的問題，其答案無法在評論中找到。最後，我們可以使用起始索引和答案跨度的長度來切出評論中與答案對應的文本跨度：

```
start_idx = sample_df[ answers.answer_start ].iloc[0][0] end_idx =
start_idx + len(sample_df[ answers.text ].iloc[0][0]) sample_df[ context ].iloc[ 0 ]
[start_idx:end_idx]
```

這個鍵盤很緊湊

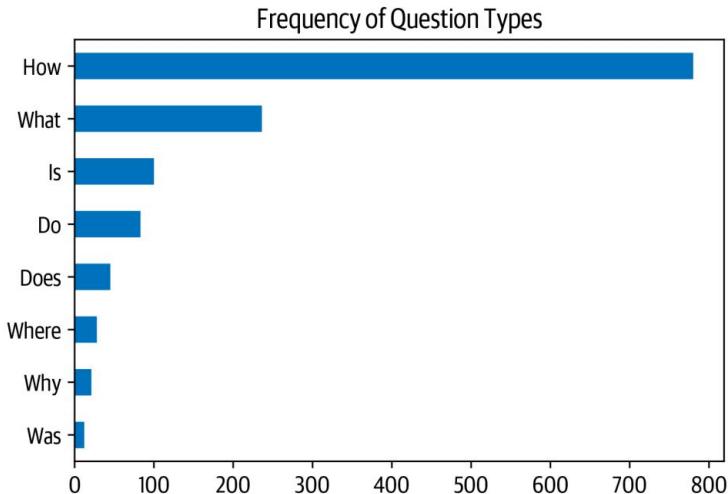
接下來，我們通過統計幾個常見起始詞開頭的問題，來感受下訓練集中有哪些類型的問題：

```
counts = {}
question_types = [ What , How , Is , Does , Do , Was , Where , Why ]
```

對於question\_types中的q：

```
counts[q] = df[ train ][ question ].str.startswith(q).value_counts()[True]
```

```
pd.Series(counts).sort_values().plot.barh() plt.title( 題型  
頻率 ) plt.show()
```



我們可以看到以“How”、“What”和“Is”開頭的問題是最常見的，所以讓我們看一些例子：

```
對於[ How , What , Is ]中的問題類型 :對於
(dfs[ train ]
[dfs[ train ].question.str.startswith(question_type)]中的問題.sample(n=3 ,
random_state=42)[ 問題 ]) :打印(問題)
```

相機怎麼樣？  
你覺得控制如何？  
充電器有多快？  
什麼是方向？  
袋子的結構質量如何？  
您對產品的印象如何？  
這是縮放的工作原理嗎？  
聲音清晰嗎？  
是無線鍵盤嗎？

斯坦福問答數據集SubjQA 的 (question, review, [answer sentences]) 格式常用於抽取式 QA 數據集，並在斯坦福問答數據集 (SQuAD) 中首創。<sup>5</sup> 這是一個著名的數據集，它是通常用於測試機器閱讀一段文本並回答相關問題的能力。該數據集是通過從維基百科中抽取數百篇英文文章創建的，將每篇文章分成幾個段落，然後要求眾多人員生成一組問題

<sup>5</sup> P. Rajpurkar 等人，“SQuAD：機器理解文本的 100,000 多個問題”，(2016)。

以及每一段的答案。在 SQuAD 的第一個版本中，每個問題的答案都保證存在於相應的段落中。但沒過多久，序列模型就開始比人類更好地提取包含答案的正確文本範圍。為了使任務更加困難，SQuAD 2.0 是通過使用一組與給定段落相關但不能僅從文本中回答的對抗性問題來擴充 SQuAD 1.1 創建的。<sup>6</sup>本書寫作時的最新技術水平如下所示在圖 7-3 中，自 2019 年以來的大多數模型都超過了人類的表現。

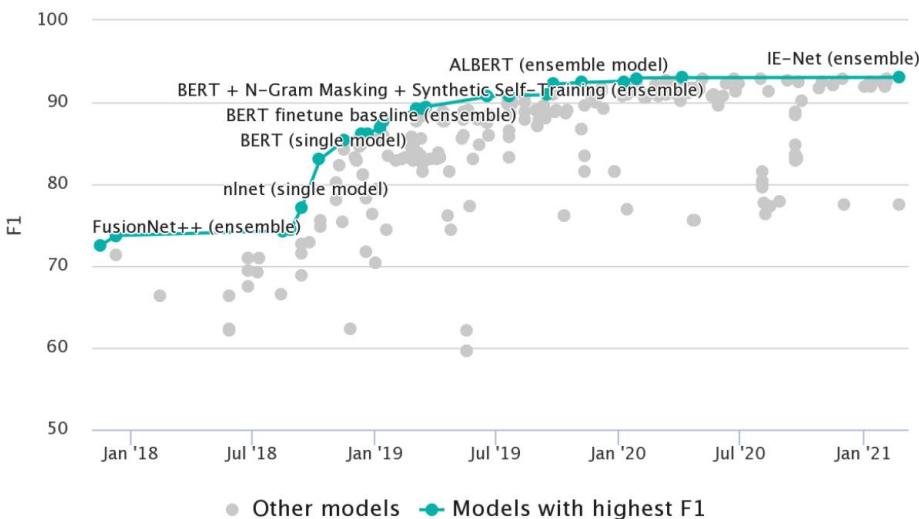


圖 7-3。SQuAD 2.0 基準測試的進展（圖片來自 Papers with Code）

然而，這種超人的表現似乎並不能反映出真正的閱讀理解能力，因為“無法回答”的問題的答案通常可以通過文章中的模式（如反義詞）來識別。為了解決這些問題，谷歌發布了自然問題（NQ）數據集，<sup>7</sup>它涉及從谷歌搜索用戶那裡獲得的事實調查問題。NQ 中的答案比 SQuAD 中的答案長得多，並且提出了更具挑戰性的基準。

現在我們已經稍微探索了我們的數據集，讓我們深入了解轉換器如何從文本中提取答案。

<sup>6</sup> P. Rajpurkar、R. Jia 和 P. Liang，“你知道你不知道什麼：SQuAD 無法回答的問題”，（2018）。

<sup>7</sup> T. Kwiatkowski 等人，“自然問題：問答研究的基準”，計算語言學協會彙刊 7（2019 年 3 月）：452-466。[http://dx.doi.org/10.1162/tacl\\_a\\_00276](http://dx.doi.org/10.1162/tacl_a_00276)。

## 從文本中提取答案我們的

QA 系統需要做的第一件事是找到一種方法，將潛在答案識別為客戶評論中的一段文本。例如，如果我們有這樣的問題“它防水嗎？”評論段落是“This watch is waterproof at 30m depth”，那麼模型應該輸出“waterproof at 30m”。為此，我們需要了解如何：

- 構建監督學習問題。
- 為QA任務標記和編碼文本。
- 處理超過模型最大上下文大小的長段落。

讓我們先來看看如何界定問題。

### 跨度分類從文本

中提取答案的最常見方法是將問題定義為跨度分類任務，其中答案跨度的開始和結束標記充當模型需要預測的標籤。這個過程如圖 7-4 所示。

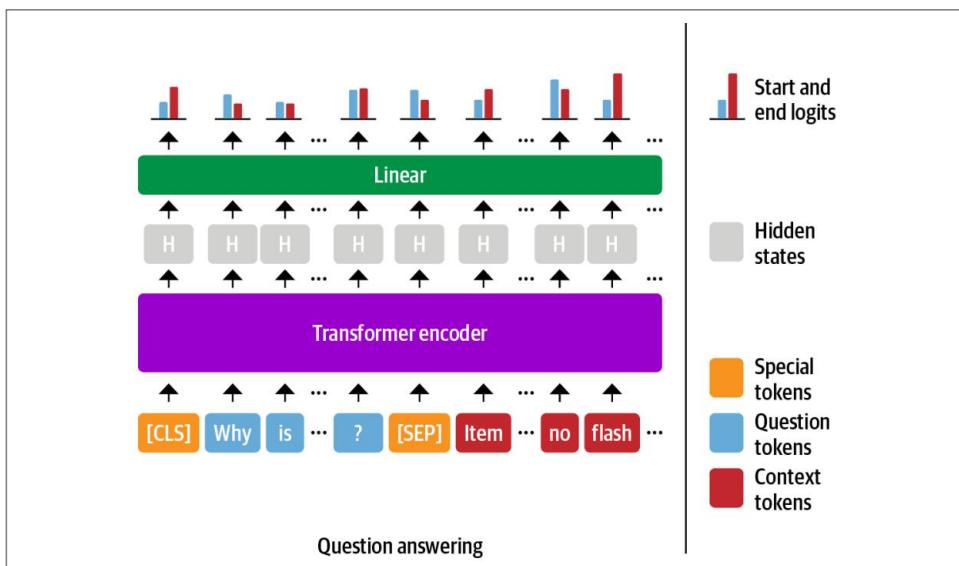


圖 7-4。QA 任務的跨度分類頭

由於我們的訓練集相對較小，只有 1,295 個示例，因此一個好的策略是從已經在 SQuAD 等大型 QA 數據集上微調過的語言模型開始。總的來說，這些模型具有很強的閱讀理解能力，可以作為構建更準確系統的良好基線。

這與前幾章中採用的方法有些不同，我們在前幾章中

通常從預訓練模型開始，然後自己微調特定任務的頭部。例如，在第 2 章中，我們必須微調分類頭，因為類的數量與手頭的數據集相關聯。對於提取式 QA，我們實際上可以從微調模型開始，因為標籤的結構在數據集中保持不變。

您可以通過導航到 Hugging Face Hub 找到提取 QA 模型的列表並在“模型”選項卡上搜索“squad”（圖 7-5）。

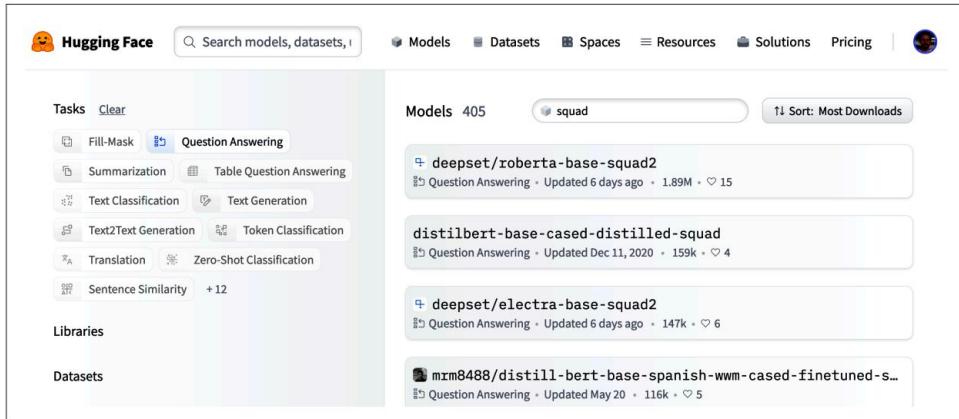


圖 7-5。Hugging Face Hub 上的精選 QA 模型

如您所見，在撰寫本文時，有超過 350 種 QA 模型可供選擇。那麼您應該選擇哪一種呢？一般來說，答案取決於各種因素，比如你的語料庫是單語還是多語，以及在生產環境中運行模型的限制。表 7-2 列出了一些模型，它們為構建提供了良好的基礎。

表 7-2 在 SQuAD 2.0 上微調的基線變壓器模型

變壓器	描述	參數個數	F1 - SQuAD 2.0 得分
迷你LM	BERT-base 的精煉版本保留了 99% 的性能，同時速度是其兩倍 RoBERTa 模型具有比其 BERT 對應模型更好的性能，並且可以使用單	66M	79.5
羅伯特基地	個 GPU 在大多數 QA 數據集上進行微調 最先進的性能在 SQuAD 2.0 上，但計算量大且難以部署具有強大零樣本性能的 100 種語言的多語	125M	83.0
艾伯特-XXL	言模型	235M	88.1
XLM-RoBERTa 大號		570M	83.8

出於本章的目的，我們將使用經過微調的 MiniLM 模型，因為它可以快速訓練並允許我們快速迭代我們將要探索的技術。<sup>8</sup> 像往常一樣，我們首先需要的是一個標記器來對我們的文本進行編碼，所以讓我們來看看它如何用於 QA 任務。

### 為 QA 標記文本

為了對我們的文本進行編碼，我們將從Hugging Face加載 MiniLM 模型檢查點中心照常：

從變形金剛導入AutoTokenizer

```
model_ckpt = "deepset/minilm-uncased-squad2"
tokenizer = AutoTokenizer.from_pretrained(model_ckpt)
```

要查看運行中的模型，讓我們首先嘗試從一小段文本中提取答案。在提取式 QA 任務中，輸入以 (question, context) 對的形式提供，因此我們將它們都傳遞給分詞器，如下所示：

```
question = "這能容納多少音樂？" context = 一個  
MP3 大約是 1 MB/分鐘，所以大約 6000 小時取決於\文件大小。 inputs = tokenizer(question, context,  
return_tensors= pt )
```

這裡我們返回了 PyTorch Tensor 對象，因為我們需要它們來運行模型的前向傳遞。如果我們將標記化的輸入視為表格：

input_ids	101	2129	2172	2189	2064	2023	...	5834	2006	5371	2946	1012	102										
token_type_ids	0	0	0	0	0	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
attention_mask	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

我們可以看到熟悉的input\_ids和attention\_mask張量，而token\_type\_ids張量表示輸入的哪一部分對應於問題和上下文（0 表示問題標記，1 表示上下文標記）。<sup>9</sup>

要了解分詞器如何格式化 QA 任務的輸入，讓我們解碼input\_ids張量：

```
打印 (tokenizer.decode (輸入[ "input_ids" ][0]))
```

---

<sup>8</sup> W. Wang 等人，“MINILM：用於預訓練任務不可知壓縮的深度自我注意蒸餾變形金剛”，（2020）。

<sup>9</sup> 請注意，token\_type\_ids並非存在於所有轉換器模型中。對於類似 BERT 的模型（例如 MiniLM），token\_type\_ids 也在預訓練期間用於合併下一句預測任務。

[CLS] 這能容納多少音樂？ [SEP] 一個 mp3 大約是 1 mb/分鐘，所以大約 6000 小時取決於文件大小。 [九月]

我們看到對於每個 QA 示例，輸入採用以下格式：

[CLS] 問題標記 [SEP] 上下文標記 [SEP]

其中第一個[SEP]令牌的位置由 token\_type\_ids 確定。

現在我們的文本已標記化，我們只需要用 QA 頭實例化模型並通過前向傳遞運行輸入：

[從變壓器導入火炬](#)

[導入AutoModelForQuestionAnswering](#)

```
模型= AutoModelForQuestionAnswering.from_pretrained(model_ckpt)
```

```
用torch.no_grad(): outputs =
model(**inputs) print(outputs)
```

```
QuestionAnsweringModelOutput(loss=None, start_logits=tensor([[ -0.9862, -4.7750,
-5.4025, -5.2378, -5.2863, -5.5117, -4.9819, -6.1880, -0.9862, 0.2596,
-0.2144, -1.7136, 3.7806, 4.8561, -1.0546, -3.9097, -1.7374, -8.4, 594, -4, 4, 5924 5.0390, -0.2018,
-3.0193, -4.8549, -2.3107, -3.5110, -3.5713, -0.9862]]), end_logits=tensor([[ -0.9623, -5.4733, -5.0326,
-5.1639, -5.4278, -5.5151, -5.1749, -4.6233, -0.9623, -3.7855, -0.8715, -3.7745, -3.0161, -1.1780,
0.1758, -2.7365, 4.8934, 0.3046, -3.1761, -3.2762, 0.8937, -03 -5.4, 5, 6662, 0.8937, -5.4, 5, 6666 3.2531,
-0.0914, 1.6211, -0.9623]]), hidden_states=無,
```

(注意=無)

在這裡我們可以看到我們得到了一個QuestionAnsweringModelOutput對像作為 QA 負責人的輸出。[如圖 7-4](#)所示，QA 頭對應於一個線性層，它從編碼器中獲取隱藏狀態併計算開始和結束跨度的對數。<sup>10</sup>這意味著我們將 QA 視為令牌分類的一種形式，類似於類似於我們在第 4 章中遇到的命名實體識別。要將輸出轉換為答案範圍，我們首先需要獲取開始和結束標記的對數：

```
start_logits = outputs.start_logits end_logits =
outputs.end_logits
```

如果我們將這些 logits 的形狀與輸入 ID 進行比較：

```
print(f Input IDs shape: {inputs.input_ids.size()} ) print(f Start logits
shape: {start_logits.size()} ) print(f End logits shape:
{end_logits.size()} )
```

---

<sup>10</sup>有關如何提取這些隱藏狀態的詳細信息，請參閱第 2 章。

輸入 ID 形狀 :`torch.Size([1, 28])`  
 開始 logits 形狀 :`torch.Size([1, 28])`  
 結束邏輯形狀 :`torch.Size([1, 28])`

我們看到有兩個 logits (開始和結束)與每個輸入標記相關聯。如圖 7-6 所示，較大的正對數對應於開始和結束標記的更可能候選者。在這個例子中，我們可以看到模型將最高的起始標記 logits 分配給數字“1”和“6000”，這是有道理的，因為我們的問題是詢問一個數量。同樣，我們看到具有最高 logits 的結束標記是“分鐘”和“小時”。

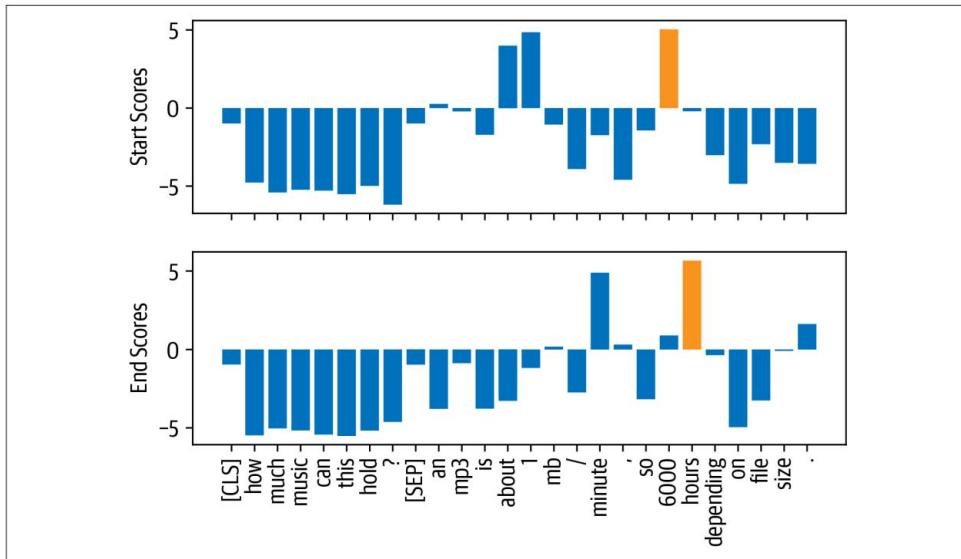


圖 7-6。開始和結束標記的預測 logits；得分最高的標記為橙色

為了得到最終答案，我們可以計算開始和結束標記對數的 `argmax`，然後從輸入中切分跨度。以下代碼執行這些步驟並對結果進行解碼，以便我們打印結果文本：

進口手電筒

```
start_idx = torch.argmax(start_logits) end_idx =
torch.argmax(end_logits) + 1 answer_span =
inputs[ input_ids ][0][start_idx:end_idx] answer =
tokenizer.decode(answer_span) print(f' 問題:{問題} ) 打印(f' 答案:{answer} )
```

問題：這能容納多少音樂？  
 答：6000小時

太好了，成功了！在 Transformers 中，所有這些預處理和後處理步驟都方便地包裝在專用管道中。我們可以通過傳遞我們的分詞器和微調模型來實例化管道，如下所示：

#### 從變壓器導入管道

```
pipe = pipeline('question-answering', model=model, tokenizer=tokenizer)管道
(question=question, context=context, topk=3)

[{"score": 0.26516005396842957,
 "start": 38, "end": 48,
 "answer": "6000 hours"},
 {"score": 0.2208300083875656,
 "start": 16, "end": 48,
 "answer": "1 MB/minute, so about
 6000 hours"}, {"score": 0.10253632068634033,
 "start": 16, "end": 27,
 "answer": "1 MB/minute"}]
```

除了答案之外，管道還在分數字段中返回模型的概率估計（通過對 logits 進行 softmax 獲得）。當我們想在一個上下文中比較多個答案時，這很方便。我們還展示了我們可以通過指定 topk 參數讓模型預測多個答案。有時，可能會有無法回答的問題，例如 SubjQA 中空的 answers.answer\_start 示例。在這些情況下，模型將為 [CLS] 標記分配一個高的開始和結束分數，並且管道將此輸出映射到一個空字符串：

```
pipe(question="為什麼沒有數據？", context=context,
      handle_impossible_answer=True)

{"分數": 0.9068416357040405, "開始": 0, "結束": 0, "答案": ""}
```



在我們的簡單示例中，我們通過獲取相應 logits 的 argmax 來獲得開始和結束索引。然而，這種啟發式方法可以通過選擇屬於問題而不是上下文的標記來產生超出範圍的答案。在實踐中，流水線會根據各種約束計算開始和結束索引的最佳組合，例如在範圍內，要求開始索引在結束索引之前，等等。

### 處理長篇文章閱讀理解模型

型面臨的一個微妙之處是，上下文中包含的標記通常多於模型的最大序列長度（通常最多為幾百個標記）。如圖 7-7 所示，SubjQA 訓練集的相當一部分包含問題-上下文對，這些對不適合 MiniLM 的 512 個標記的上下文大小。

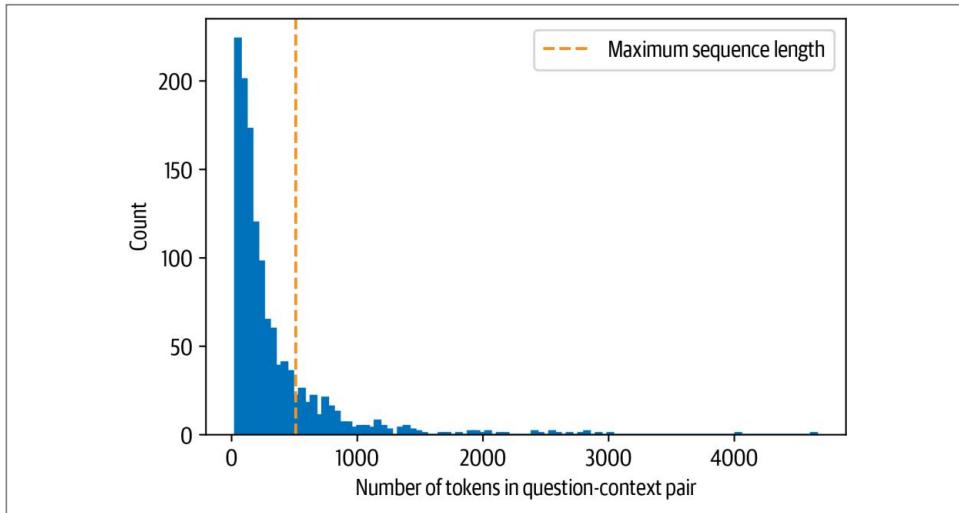


圖 7-7。SubjQA 訓練集中每個問題-上下文對的標記分佈

對於文本分類等其他任務，我們假設[CLS]標記的嵌入中包含足夠的信息以生成準確的預測，從而簡單地截斷長文本。然而，對於 QA，這種策略是有問題的，因為問題的答案可能位於上下文的末尾附近，因此會被截斷刪除。如圖 7-8 所示，處理這個問題的標準方法是在輸入上應用一個滑動窗口，其中每個窗口都包含一段適合模型上下文的標記。

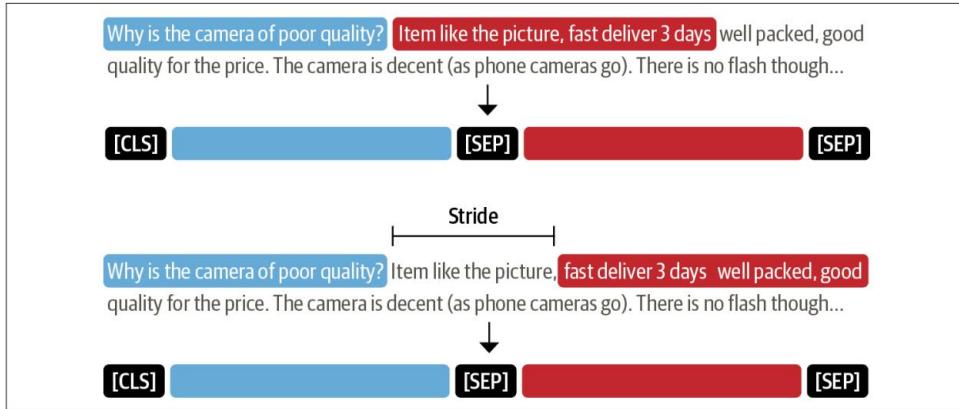


圖 7-8 ◦滑動窗口如何為長文檔創建多個問題-上下文對 第一個欄對應於問題 ,而第二個欄是每個窗口中捕獲的上下文

在 Transformers 中 ,我們可以在 tokenizer 中設置 return\_overflowing\_tokens=True 以啟用滑動窗口 。滑動窗口的大小由 max\_seq\_length 參數控制 ,stride 的大小由 doc\_stride 控制 。

讓我們從訓練集中獲取第一個示例並定義一個小窗口來說明它是如何工作的：

```
example = dfs[ train ].iloc[0][[ question , context ]].tokenized_example
= tokenizer(example[ question ],example[ context ],return_overflowing_tokens=True,
max_length=100,步幅=25)
```

在這種情況下 ,我們現在得到一個 input\_ids 列表 ,每個窗口一個 。讓我們檢查每個窗口中的標記數量：

```
對於 idx 枚舉中的窗口 (tokenized_example[ “input_ids” ]) : print (f “Window #{idx} has {len (window)} tokens” )
```

窗口 #0 有 100 個令牌  
窗口 #1 有 88 個標記

最後 ,我們可以通過解碼輸入來查看兩個窗口重疊的位置：

```
對於 tokenized_example[ input_ids ] 中的窗口 :
print(f {tokenizer.decode(window)} \n )
```

[CLS]低音如何 ? [SEP] 我過去有過 koss 耳機 ,pro 4aa 和 qz-99 ◦koss portapro 是便攜式的 ,低音響應很好 。與我的 android 手機配合使用效果很好 ,可以捲起來放入我的機車夾克或電腦包中 ,不會嘎吱作響 。它們非常輕 ,即使在聽完音樂後也不會感到沉重或壓在你的耳朵上一整天 。聲音是 [SEP]

[CLS]低音如何 ? [SEP] 甚至不要感到沉重或壓在耳朵上

在和他們一起聽了一整天的音樂之後。聲音比任何耳塞都要好，而且幾乎和 pro 4aa 一樣好。他們是露天接近。只需 32 美元，您就不會出錯。[九月]  
”耳機，因此您無法將低音與密封類型相匹配，但它

現在我們對 QA 模型如何從文本中提取答案有了一些直覺，讓我們看看構建端到端 QA 管道所需的其他組件。

### 使用 Haystack 構建 QA 管道在我們簡單的答案

提取示例中，我們向模型提供了問題和上下文。然而，實際上我們系統的用戶只會提供有關產品的問題，因此我們需要某種方式從語料庫中的所有評論中選擇相關段落。一種方法是將給定產品的所有評論連接在一起，並將它們作為單個長上下文提供給模型。

雖然簡單，但這種方法的缺點是上下文可能變得非常長，從而給我們的用戶查詢帶來不可接受的延遲。

例如，假設平均每個產品有 30 條評論，每條評論的處理時間為 100 毫秒。如果我們需要處理所有評論以獲得答案，這將導致每個用戶查詢的平均延遲為 3 秒。對於電子商務網站來說太長了！

為了解決這個問題，現代 QA 系統通常基於檢索器-閱讀器架構，它有兩個主要組件：

#### 獵犬

負責為給定查詢檢索相關文檔。獵犬通常被歸類為稀疏或密集。稀疏檢索器使用詞頻將每個文檔和查詢表示為一個稀疏向量。<sup>11</sup>然後通過計算向量的內積來確定查詢和文檔的相關性。另一方面，密集檢索器使用像轉換器這樣的編碼器將查詢和文檔表示為上下文嵌入（這是密集向量）。這些嵌入編碼語義，並允許密集檢索器通過理解查詢的內容來提高搜索準確性。

#### Reader

負責從檢索器提供的文檔中提取答案。閱讀器通常是一個閱讀理解模型，儘管在本章末尾我們將看到可以生成自由形式的模型示例

答案。

---

<sup>11</sup>如果向量的大部分元素為零，則該向量是稀疏的。

如圖 7-9 所示，還可以有其他組件對檢索器獲取的文檔或閱讀器提取的答案應用後處理。例如，檢索到的文檔可能需要重新排序，以消除可能使讀者感到困惑的嘈雜或不相關的文檔。同樣，當正確答案來自長文檔中的不同段落時，通常需要對讀者的答案進行後處理。

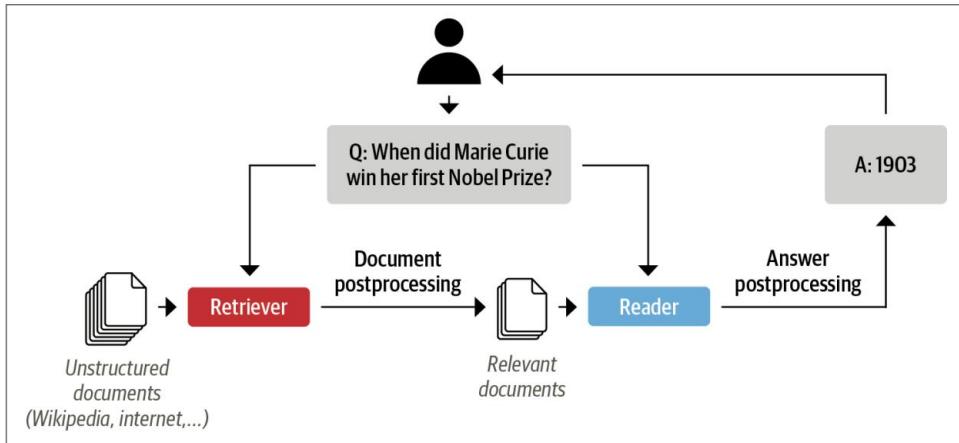


圖 7-9 現代 QA 系統的檢索器閱讀器架構

為了構建我們的 QA 系統，我們將使用 **Haystack 庫**由 deepset 開發，一家專注於 NLP 的德國公司。Haystack 基於 retriever-reader 架構，抽象了構建這些系統所涉及的大部分複雜性，並與 Transformers 緊密集成。



除了檢索器和閱讀器之外，在使用 Haystack 構建 QA 管道時還涉及兩個組件：

#### 文檔存儲

一個面向文檔的數據庫，用於存儲在查詢時提供給檢索器的文檔和元數據

#### 管道

結合 QA 系統的所有組件以啟用自定義查詢流程、合併來自多個檢索器的文檔等

在本節中，我們將了解如何使用這些組件快速構建原型 QA 管道。稍後，我們將研究如何提高其性能。



本章是使用 Haystack 庫的 0.9.0 版編寫的。在 [0.10.0 版本中](#)，重新設計了管道和評估 API，以便更輕鬆地檢查檢索器或讀取器是否正在影響性能。要查看本章的代碼在使用新 API 時的樣子，請查看[GitHub 存儲庫](#)。

## 初始化文檔存儲在 Haystack

中，有多種文檔存儲可供選擇，每個存儲都可以與一組專用的檢索器配對。這在 [表 7-3 中進行了說明](#)，其中顯示了每個可用文檔存儲的稀疏（TF-IDF、BM25）和密集（嵌入、DPR）檢索器的兼容性。我們將在本章後面解釋所有這些縮寫詞的含義。

表 7-3。 Haystack 檢索器和文檔存儲的兼容性

	內存中 Elasticsearch FAISS Milvus			
特遠離-IDF	是的	是的	不	不
BM25	不	是的	不	不
嵌入是		是的	是的	是的
向量共形	是的	是的	是的	是的

由於我們將在本章中探索稀疏和密集檢索器，因此我們將使用 ElasticsearchDocumentStore，它與這兩種檢索器類型兼容。Elasticsearch 是一種搜索引擎，能夠處理多種數據類型，包括文本、數字、地理空間、結構化和非結構化數據。它能夠存儲大量數據並使用全文搜索功能對其進行快速過濾，這使其特別適合開發 QA 系統。它還具有成為基礎架構分析行業標準的優勢，因此您的公司很有可能已經擁有一個可以使用的集群。

要初始化文檔存儲，我們首先需要下載並安裝 Elasticsearch。

按照 Elasticsearch 的[指南](#)，<sup>12</sup>我們可以使用 wget 獲取 Linux 的最新版本，並使用 tar shell 命令將其解壓縮：

```
url =      https://artifacts.elastic.co/downloads/elasticsearch/elasticsearch-7.9.2-
linux-x86_64.tar.gz      !wget -nc -q {url} !tar -xzf elasticsearch-7.9.2-linux-
x86_64.tar.gz
```

接下來我們需要啟動 Elasticsearch 服務器。由於我們在 Jupyter 筆記本中運行本書中的所有代碼，因此我們需要使用 Python 的 Popen()函數來生成

---

<sup>12</sup>該指南還提供了 macOS 和 Windows 的安裝說明。

一個新的過程。當我們這樣做的時候，讓我們也使用chown shell 命令在後台運行子進程：

[從子進程導入](#)

[os導入Popen、PIPE、STDOUT](#)

```
# 將 Elasticsearch 作為後台進程運行!chown -R daemon:daemon
elasticsearch-7.9.2 es_server =
Popen(args=[ elasticsearch-7.9.2/bin/elasticsearch ], stdout=PIPE, stderr=STDOUT,
       preexec_fn=lambda : os.setuid(1))
# 等待 Elasticsearch 啟動!sleep 30
```

在Popen()函數中， args指定我們希望執行的程序，而stdout=PIPE為標準輸出創建一個新管道，而stderr=STDOUT在同一個管道中收集錯誤。 preexec\_fn參數指定了我們希望使用的子進程的ID。默認情況下，Elasticsearch 在本地端口 9200 上運行，因此我們可以通過向本地主機發送 HTTP 請求來測試連接：

```
!curl -X GET  localhost:9200/_漂亮
{
    "name" : "96938eee37cd" ,
    "cluster_name" : "docker-cluster" ,
    "cluster_uuid" : "ABGDdvbbRWmMb9Umz79HbA" ,
    "version" : { "number" : "7.9.2" , "build_flavor" :
        "default" , "build_type" : "docker" , "build_hash" :
        "d34da0ea4a966c4e49417f2da2f244e3e97b4e6e" ,
        "build_date" : "2020-09-23T00:45:33.626720Z" ,
        "build_snapshot" : false , "lucene_version" : "8.6.2" ,
        "minimillibity_version_wire" : 6.8.0 ,
        "minimum_index_compatibility_version" : 6.0.0-beta1 }, "tagline" :
    You Know, for Search
}
```

現在我們的 Elasticsearch 服務器已啟動並運行，接下來要做的是實例化文檔存儲：

[從haystack.document\\_store.elasticsearch導入ElasticsearchDocumentStore](#)

```
# 返回文檔嵌入以供以後與密集檢索器一起使用document_store =
ElasticsearchDocumentStore(return_embedding=True)
```

默認情況下，ElasticsearchDocumentStore在 Elasticsearch 上創建兩個索引：一個稱為文檔（您猜對了）用於存儲文檔，另一個稱為標籤用於存儲帶註釋的答案範圍。現在，我們只填充文檔索引

SubjQA 評論和 Haystack 的文檔存儲需要一個包含文本和元鍵的字典列表，如下所示：

```
{
    text : <the-context> ,
    meta :{ field_01 :
        <additional-metadata> , field_02 :
        <additional-metadata> ,
        ...
    }
}
```

meta中的字段可用於在檢索期間應用過濾器。出於我們的目的，我們將包含SubjQA 的item\_id 和q\_review\_id列，以便我們可以按產品和問題 ID 以及相應的訓練拆分進行過濾。然後我們可以循環遍歷每個DataFrame中的示例，並使用write\_documents()方法將它們添加到索引中，如下所示：

```
對於拆分，dfs.items()中的df：
# 排除重複評論docs = [{ text :
    row[ context ], meta :{ item_id :
        row[ title ], question_id :row[ id ],
        "分裂" :分裂}}
    對於_， df.drop_duplicates(subset= context ).iterrows()中的行
    document_store.write_documents (文檔 索引= “文檔” )

print(f 加載 {document_store.get_document_count()} 個文檔 )
已加載 1615 個文件
```

太好了，我們已將所有評論加載到索引中！要搜索索引，我們需要一個檢索器，所以讓我們看看如何為 Elasticsearch 初始化一個檢索器。

## 初始化檢索器

Elasticsearch 文檔存儲可以與任何 Haystack 檢索器配對，因此讓我們從使用基於 BM25（“Best Match 25”的縮寫）的稀疏檢索器開始。

BM25 是經典詞頻-逆文檔頻率 (TF-IDF) 算法的改進版本，將問題和上下文表示為可以在 Elasticsearch 上高效搜索的稀疏向量。BM25 分數衡量有多少匹配文本與搜索查詢有關，並通過快速飽和 TF 值和標準化文檔長度來改進 TF-IDF，從而使短文檔比長文檔更受青睞。13

---

13有關使用 TF-IDF 和 BM25 對文檔評分的深入解釋，請參閱 Speech and Lan- 的第 23 章  
量表處理，第 3 版，作者 D. Jurafsky 和 JH Martin (Prentice Hall)。

在 Haystack 中， ElasticsearchRetriever默認使用 BM25 檢索器，因此讓我們通過指定我們希望搜索的文檔存儲來初始化此類：

從haystack.retriever.sparse導入ElasticsearchRetriever

```
es_retriever = ElasticsearchRetriever(document_store=document_store)
```

接下來，讓我們看一下對訓練集中單個電子產品的簡單查詢。

對於像我們這樣的基於評論的 QA 系統，將查詢限制為單個項目很重要，否則檢索器將獲取與用戶查詢無關的產品評論。例如，詢問“相機質量好嗎？”沒有產品過濾器可能會返回有關手機的評論，而此時用戶可能會詢問特定的筆記本電腦相機。我們數據集中的 ASIN 值本身有點神秘，但我們可以使用亞馬遜 ASIN 等在線工具破譯它們或者只需將 item\_id 的值附加到 www.amazon.com/dp/ URL。

以下項目 ID 對應於亞馬遜的一款 Fire 平板電腦，因此讓我們使用檢索器的 retrieve() 方法來詢問它是否適合閱讀：

```
item_id = B0074BW614
query = "適合閱讀嗎？"
retrieved_docs =
es_retriever.retrieve(query=query, top_k=3,
    filters=[{"item_id": item_id}, {"split": ["train"]}]}
```

在這裡，我們指定了使用 top\_k 參數返回多少文檔，並對包含在我們文檔的元字段中的 item\_id 和 split 鍵應用過濾器。 retrieved\_docs 的每個元素都是一個 Haystack Document 對象，用於表示文檔並包括檢索器的查詢分數以及其他元數據。讓我們看一下檢索到的文檔之一：

[打印（檢索文檔\[0\]）](#)

```
{ "text": "這是給自己的禮物。我使用 kindle 已有 4 年，這是我的第三次使用。我從沒想過我會想要生火，因為我主要用它來讀書。我決定試一試，因為我旅行時會帶上筆記本電腦、手機和 iPod classic。我喜歡我的 iPod，但在飛機上用它看電影可能會很困難，因為它太小了。筆記本電腦的電池壽命不如 Kindle，所以 Fire 為我結合了我需要這三者做的事情。到目前為止一切順利。" , "score": 6.243799, "probability": 0.6857824513476455, "question": None, "meta": { "item_id": B0074BW614, "question_id": 868e311275e26dbafe5af70774a300f3, "split": "火車" }, "embed": null, "id": 252e83e25d52df7311d597dc89eef9f6 }
```

除了文檔的文本之外，我們還可以看到 Elasticsearch 計算的與查詢的相關性的分數（分數越高表示匹配越好）。在幕後，Elasticsearch 依賴於 Lucene 用於索引和搜索，因此默認情況下它使用 Lucene 的實用評分功能。您可以在 Elasticsearch 文檔中找到評分函數背後的細節，但簡而言之，它首先通過應用布爾測試（文檔是否與查詢匹配？）來過濾候選文檔。

然後應用基於將文檔和查詢表示為向量的相似性度量。

現在我們有了檢索相關文檔的方法，接下來我們需要的是從中提取答案的方法。這是讀者進來的地方，所以讓我們來看看我們如何在 Haystack 中加載我們的 MiniLM 模型。

## 初始化閱讀器

在 Haystack 中，有兩種類型的閱讀器可以用來從給定的上下文中提取答案：

### 農場閱讀器

基於 deepset 的 **FARM 框架** 用於微調和部署轉換器。兼容使用 Transformers 訓練的模型，可以直接從 Hugging Face Hub 加載模型。

### 變形金剛閱讀器

基於 Transformers 的 QA 管道。僅適用於運行推論。

儘管兩個讀者以相同的方式處理模型的權重，但在將預測轉換為答案的方式上存在一些差異：

- 在  Transformers 中，QA 管道在每個段落中使用 softmax 對開始和結束 logit 進行歸一化。這意味著只有比較從同一篇文章中提取的答案之間的答案分數才有意義，其中概率總和為 1。例如，一篇文章的答案分數為 0.9 不一定比另一篇文章的答案分數為 0.8 好。在 FARM 中，logits 沒有標準化，因此可以更容易地比較段落間的答案。
- TransformersReader 有時會兩次預測相同的答案，但得分不同。如果答案位於兩個重疊的窗口中，這可能會在較長的上下文中發生。在 FARM 中，這些重複項已被刪除。

由於我們將在本章後面對閱讀器進行微調，因此我們將使用 FARMReader。與變形金剛一樣，要加載模型，我們只需要在 Hugging Face Hub 上指定 MiniLM 檢查點以及一些特定於 QA 的參數：

從 `haystack.reader.farm` 導入 FARMReader

```
model_ckpt = 'deepset/minilm-uncased-squad2'
max_seq_length = 384
doc_stride = 128
reader = FARMReader(
    model_name_or_path=model_ckpt,
    progress_bar=False,
    max_seq_len=max_seq_length,
    doc_stride=doc_stride,
    return_no_answer=True)
```



也可以直接在 Transformers 中微調閱讀理解模型，然後將其加載到Transformers的詳細信息中。請參閱 [推理當有關閱讀教程](#) 微調步驟。

在 FARMReader 中，滑動窗口的行為由我們在分詞器中看到的相同 max\_seq\_length 和 doc\_stride 參數控制。這裡我們使用了 MiniLM 論文中的值。為了確認，現在讓我們用之前的簡單示例來測試讀者：

```
打印 (reader.predict_on_texts (問題=問題, 文本=[上下文], top_k=1) )  
{ query : 這能放多少音樂？, no_ans_gap : 12.648084878921509, answers :[{ answer :  
    6000 小時, score : 10.69961929321289, probability : 0.3988136053085327, context : 一個  
MP3 大約是 1 MB/分鐘，所以大約 6000 小時取決於文件大小。, offset_start : 38, offset_end : 48,  
offset_start_in_doc : 38, offset_end_in_doc : 48, document_id : e344757014e804eff50faa3ecf1c9c75 }]}  
太棒了，閱讀器似乎按預期工作。所以接下來，讓我們使用 Haystack 的管道之一將所有組件連接在一起。
```

將它們放在一起

Haystack 提供了一個流水線抽象，它允許我們將檢索器、讀取器和其他組件組合在一起作為一個圖，可以很容易地為每個用例定制。還有類似於 Transformers 中的預定義管道，但專門用於 QA 系統。在我們的例子中，我們對提取答案感興趣，因此我們將使用 ExtractiveQAPipeline，它以單個檢索器-閱讀器對作為其參數：

[從 haystack.pipeline 導入 ExtractiveQAPipeline](#)

```
pipe = ExtractiveQAPipeline (閱讀器, es_retriever)
```

每個 Pipeline 都有一個 run() 方法來指定查詢流應該如何執行。對於 ExtractiveQAPipeline，我們只需要傳遞查詢、使用 top\_k\_retriever 檢索的文檔數量，以及使用 top\_k\_reader 從這些文檔中提取的答案數量。在我們的例子中，我們還需要在項目 ID 上指定一個過濾器，這可以像我們之前對檢索器所做的那樣使用 filters 參數來完成。讓我們再次使用我們關於 Amazon Fire 平板電腦的問題來運行一個簡單的示例，但這次返回提取的答案：

```
n_answers = 3  
preds = pipe.run(query=query, top_k_retriever=3, top_k_reader=n_answers, filters=[{ item_id :  
    [item_id], split : [ train ] }])  
print(f 問題: {preds[ query ]} \n )
```

```
對於範圍內的idx (n_answers) :
print(f Answer {idx+1}: {preds[ answers ][idx][ answer ]} ) print(f 評論
片段：...{preds[ answers ][idx][ 上下文 ]}... ) 打印( \n\n )
```

問 讀書好不好？

答1 :我主要用它看書是我的第三個。我從沒想過我會想要一把火！Review snippet: ... 主要用來讀書。我決定在旅行時嘗試火，因為我帶著我的 la...

答案 2 :與 Kindle 相比更大的屏幕讓閱讀更輕鬆我喜歡這種顏色，這是我從未想過的……

答案 3 :在沒有燈光的情況下看書非常好 評論片段：...對她上癮了！我們的兒子喜歡它，它非常適合在沒有燈光的情況下看書。驚人的聲音，但我建議好的耳機...

太棒了，我們現在有一個用於亞馬遜產品評論的端到端 QA 系統！這是一個好的開始，但請注意，第二個和第三個答案更接近問題的實際要求。為了做得更好，我們需要一些指標來量化檢索器和讀取器的性能。我們接下來會看一下。

## 改進我們的質量檢查流程

儘管最近關於 QA 的大部分研究都集中在改進閱讀理解模型上，但在實踐中，如果檢索器一開始就找不到相關文檔，那麼你的閱讀器有多好都沒有關係！特別是，檢索器為整個 QA 系統的性能設置了上限，因此確保它做得很好很重要。考慮到這一點，讓我們首先介紹一些評估檢索器的常用指標，以便我們可以比較稀疏表示和密集表示的性能。

### 評估檢索器評估檢索器的一個

常用指標是召回率，它衡量檢索到的所有相關文檔的比例。在這種情況下，“相關”僅表示答案是否出現在一段文本中，因此給定一組問題，我們可以通過計算答案在返回的前 k 個文檔中出現的次數來計算召回率獵犬。

在 Haystack 中，有兩種評估檢索器的方法：

- 使用檢索器的內置eval()方法。這可用於開放域和封閉域 QA，但不適用於像 SubjQA 這樣的數據集，其中每個文檔與單個產品配對，我們需要為每個查詢按產品 ID 進行過濾。
- 構建一個將檢索器與EvalRetriever類組合在一起的自定義管道。

這可以實現自定義指標和查詢流。



召回率的一個補充指標是平均準確率 (mAP)，它獎勵能夠將正確答案放在文檔排名中較高位置的檢索器。

由於我們需要評估每個產品的召回率，然後匯總所有產品，我們將選擇第二種方法。Pipeline圖中的每個節點都代表一個類，該類接受一些輸入並通過run()方法產生一些輸出：

```
類PipelineNode: def
    __init__(self):
        self.outgoing_edges = 1

    def run(self, **kwargs) :
        ...
        return (output, "outgoing_edge_name")
```

這裡的kwargs對應於圖中前一個節點的輸出，它在run()方法中被操作以返回下一個節點的輸出元組，以及傳出邊的名稱。唯一的其他要求是包含一個outgoing\_edges屬性，該屬性指示節點的輸出數量（在大多數情況下outgoing\_edges=1，除非您在管道中有分支根據某些標準路由輸入）。

在我們的例子中，我們需要一個節點來評估檢索器，因此我們將使用EvalRetriever類，其run()方法跟蹤哪些文檔的答案與基本事實相匹配。使用此類，我們可以通過在代表檢索器本身的節點之後添加評估節點來構建管道圖：

```
從haystack.pipeline導入管道從haystack.eval導入
EvalDocuments
```

```
類EvalRetrieverPipeline: def
    __init__(self, retriever): self.retriever =
        retriever
        self.eval_retriever =
            EvalDocuments()
        pipe = Pipeline()
        pipe.add_node(component=self.retriever,
                      name='ESRetriever',
```

```
    輸入=[“查詢”])
pipe.add_node(component=self.eval_retriever, name=  EvalRetriever ,
              inputs=[ ESRetriever ])
self.pipeline =管道
```

管道=EvalRetrieverPipeline(es\_retriever)

請注意，每個節點都有一個名稱和一個輸入列表。在大多數情況下，每個節點都有一個出邊，所以我們只需要在輸入中包含前一個節點的名稱。

現在我們有了評估管道，我們需要傳遞一些查詢及其相應的答案。為此，我們會將答案添加到文檔存儲中的專用標籤索引中。Haystack 提供了一個Label對象，它以標準化的方式表示答案範圍及其元數據。為了填充標籤索引，我們將首先創建一個標籤對象列表，方法是遍歷測試集中的每個問題並提取匹配的答案和其他元數據：

#### 從haystack導入標籤

```
labels = [] for i,
row in df[“test”].iterrows(): # Retriever 中用於過
    濾的元數據meta = { item_id :row[ title ], question_id :
        row[ id ] }
    # 用答案填充問題的標籤if len(row[ answers.text ]) > 0:
        for answer in row[ answers.text ]: label =
            Label(question=row[ question ], answer=answer, id
                =i, origin=row[ id ], meta=meta,
                is_correct_answer=True, is_correct_document=True, no_answer=False)標籤.附加(標籤)
```

# 為沒有其他答案的問題填充標籤：

```
label =
Label( question=row[ question ], answer= , id=i, origin=row[ id ],
meta=meta, is_correct_answer=True, is_correct_document=True, no_answer=True)
標籤.append (標籤)
```

如果我們查看其中一個標籤：

```
打印(標籤[0])
{
    id : e28f5e62-85e8-41b2-8a34-fbff63b7a466 , created_at :None, updated_at :None,
    question : 這些耳機的音調平衡是多少? , answer : 三十年來我一直是耳機狂熱者 , is_correct_answer :
True, is_correct_document :True, origin : d0781d13200014aa25860e44da9d5ea7 , document_id :
None, offset_start_in_doc :None, no_answer :False, model_id :無, meta :
{ item_id : B00001WRSJ , question_id : d0781d13200014aa25860e44da9d5ea7 }}
```

我們可以看到問答對，以及包含唯一問題 ID 的原始字段，因此我們可以按問題過濾文檔存儲。我們還將產品 ID 添加到元字段中，以便我們可以按產品過濾標籤。現在我們有了標籤，我們可以將它們寫入Elasticsearch 上的標籤索引，如下所示：

```
document_store.write_labels(labels, index= label ) print(f      加
    載 {document_store.get_label_count(index= label ))}\n問答對      )
```

加載了 358 個問答對

接下來，我們需要在我們的問題 ID 和可以傳遞給管道的相應答案之間建立映射。要獲取所有標籤，我們可以使用文檔存儲中的get\_all\_labels\_aggregated()方法，該方法將聚合與唯一 ID 關聯的所有問答對。此方法返回MultiLabel對象的列表，但在我們的例子中，由於我們按問題 ID 進行過濾，因此我們只獲得一個元素。我們可以建立一個聚合標籤列表，如下所示：

```
labels_agg = document_store.get_all_labels_aggregated(index= label ,
    open_domain=True, aggregate_by_meta=[ item_id ]

)
打印 (len (labels_agg) )
330
```

通過查看其中一個標籤，我們可以看到與給定問題相關的所有答案都聚合在一個multiple\_answers 字段中：

```
打印 (labels_agg [109])
{
    question : 風扇是如何工作的？ , multiple_answers :[ 風扇真的非常好 , 風扇本身不是很大聲。有一個可調節的刻度盤來改變風扇速度 ], is_correct_answer :True, is_correct_document :True, origin :
    5a9b7616541f700f103d21f8ad41bc4b , multiple_document_ids :[None, None],
    multiple_offset_start_in_docs :[None, None], no_answer :False, model_id :無, 元 :
    { item_id : B002MU1ZRS }}
```

我們現在擁有評估檢索器的所有要素，所以讓我們定義一個函數，將與每個產品相關聯的每個問答對提供給評估管道，並在我們的管道對像中跟蹤正確的檢索：

```
def run_pipeline (管道, top_k_retriever=10, top_k_reader=4) :
    對於labels_agg 中的：
        -
        pipeline.pipeline.run(query=l.question,
            top_k_retriever=top_k_retriever,
            top_k_reader=top_k_reader,
            top_k_eval_documents=top_k_retriever, labels=l,
            filters={ item_id : [l.meta[ item_id ]], 拆分 :[ 測試 ]})
```

```
run_pipeline(pipe, top_k_retriever=3) print(f Recall@3:
```

```
{pipe.eval_retriever.recall:.2f} )
```

```
召回@3 :0.95
```

太好了，它有效！請注意，我們為 `top_k_retriever` 選擇了一個特定值來指定要檢索的文檔數。通常，增加此參數會提高召回率，但代價是向讀者提供更多文檔並減慢端到端管道。為了指導我們決定選擇哪個值，我們將創建一個函數，該函數循環多個 `k` 值併計算每個 `k` 在整個測試集中的召回率：

```
def evaluate_retriever(retriever, topk_values = [1,3,5,10,20]): topk_results = {}
```

對於 `topk_values` 中的 `topk`：

```
# 創建管道p =
EvalRetrieverPipeline(retriever)
# 遍歷測試集中的每個問答對 run_pipeline(p, top_k_retriever=topk)
```

```
# 獲取指標
topk_results[topk] = { "召回" : p.eval_retriever.recall}
```

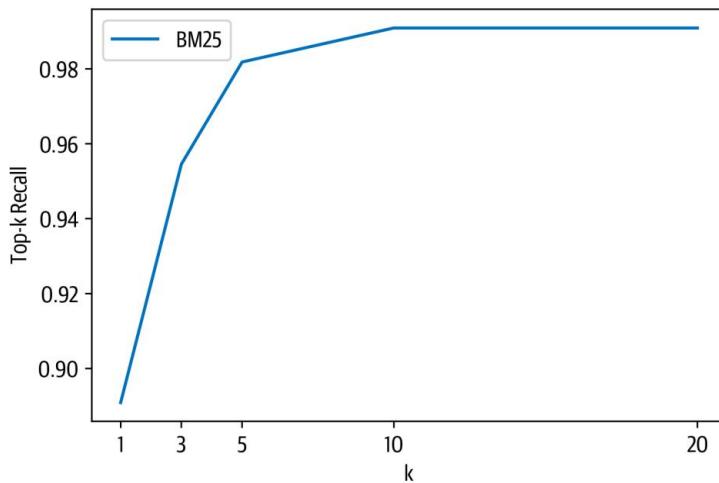
```
返回pd.DataFrame.from_dict(topk_results, orient= index )
```

```
es_topk_df = evaluate_retriever(es_retriever)
```

如果我們繪製結果，我們可以看到隨著 `k` 的增加，召回率如何提高：

```
def plot_retriever_eval(dfs, retriever_names):
    fig, ax = plt.subplots() for df, zip中
    的retriever_name (dfs, retriever_names): df.plot(y= recall , ax=ax,
        label=retriever_name) plt.xticks(df.index) plt.ylabel ( Top-k Recall )
    plt.xlabel( k ) plt.show()
```

```
plot_retriever_eval([es_topk_df], [ BM25 ])
```



從圖中，我們可以看到在  $k = 5$  附近有一個拐點，從  $k = 10$  開始，我們得到了幾乎完美的召回率。現在讓我們看一下使用密集向量技術檢索文檔。

### 密集段落檢索我們已經

看到，當我們的稀疏檢索器返回  $k = 10$  個文檔時，我們得到了幾乎完美的召回率，但是我們可以在較小的  $k$  值時做得更好嗎？這樣做的好處是我們可以將更少的文檔傳遞給讀者，從而減少 QA 管道的整體延遲。像 BM25 這樣的稀疏檢索器的一個眾所周知的限制是，如果用戶查詢包含與評論不完全匹配的術語，它們可能無法捕獲相關文檔。一個有前途的替代方案是使用密集嵌入來表示問題和文檔，目前最先進的是一種稱為密集通道檢索 (DPR) 的架構。<sup>14</sup> DPR 背後的主要思想是使用兩個 BERT 模型作為編碼器問題和段落。如圖 7-10 所示，這些編碼器將輸入文本映射為 [CLS] 標記的  $d$  維向量表示。

---

<sup>14</sup> V. Karpukhin 等人，“用於開放域問答的密集段落檢索”，(2020)。

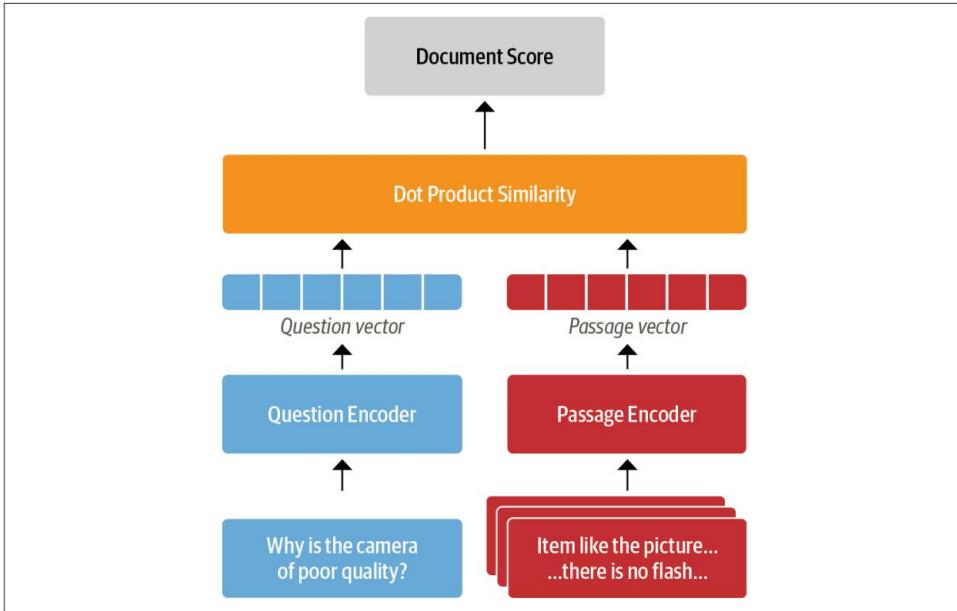


圖 7-10。DPR 用於計算文檔和查詢相關性的雙編碼器架構

在 Haystack 中，我們可以用與為 BM25 所做的類似的方式為 DPR 初始化一個檢索器。除了指定文檔存儲之外，我們還需要為問題和段落選擇 BERT 編碼器。這些編碼器通過向它們提供相關（正面）段落和不相關（負面）段落的問題進行訓練，目標是了解相關問題-段落對具有更高的相似性。對於我們的用例，我們將使用在 NQ 語料庫上經過微調的編碼器。

方式：

從 `haystack.retriever.dense` 導入 `DensePassageRetriever`

```
dpr_retriever = DensePassageRetriever(document_store=document_store,
    query_embedding_model= facebook/dpr-question_encoder-single-nq-base ,
    passage_embedding_model= facebook/dpr-ctx_encoder-single-nq-base , embed_title=False)
```

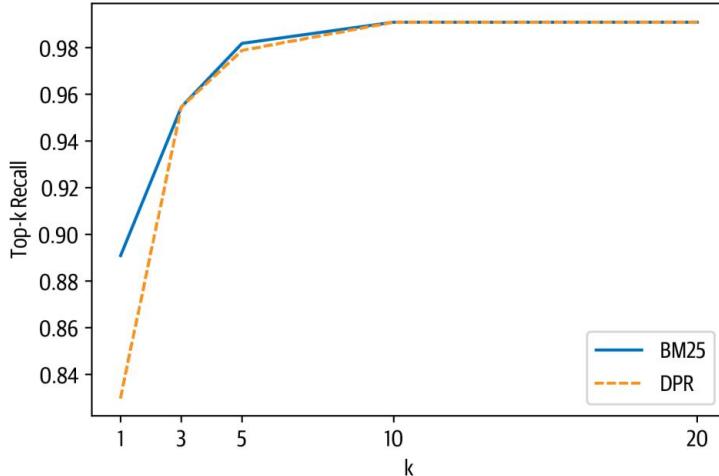
此處我們還設置了 `embed_title=False`，因為連接文檔的標題（即 `item_id`）不會提供任何附加信息，因為我們按產品進行過濾。

一旦我們初始化了密集檢索器，下一步就是迭代 Elasticsearch 索引中的所有索引文檔，並應用編碼器來更新嵌入表示。這可以按如下方式完成：

```
document_store.update_embeddings (retriever=dpr_retriever)
```

我們現在準備好了 !我們可以用與 BM25 相同的方式評估密集檢索器並比較 top-k 召回率：

```
dpr_topk_df = evaluate_retriever(dpr_retriever)
plot_retriever_eval([es_topk_df, dpr_topk_df], [ "BM25" , "DPR" ])
```



在這裡我們可以看到 DPR 沒有提供比 BM25 更高的召回率並且在  $k = 3$  附近飽和。



使用 Facebook 的[FAISS 庫](#)可以加快執行嵌入的相似性搜索作為文檔存儲。同樣，可以通過對目標域進行微調來提高 DPR 檢索器的性能。如果您想了解如何微調 DPR，請查看 [Haystack 教程](#)。

現在我們已經探索了對獵犬的評價，讓我們轉向對讀者的評價。

## 評估讀者

在提取式 QA 中，有兩個主要指標用於評估讀者：

完全匹配 (EM)

如果預測答案和真實答案中的字符完全匹配，則給出  $EM = 1$  的二元度量，否則給出  $EM = 0$ 。如果沒有預期的答案，如果模型預測任何文本，它就會得到  $EM = 0$ 。

F1

-score 測量精度和召回率的調和平均值。

讓我們通過從 FARM 導入一些輔助函數並將它們應用到一個簡單示例來了解這些指標如何工作：

從farm.evaluation.squad\_evaluation導入compute\_f1, compute\_exact

```
pred = about 6000 hours  label
= 6000 hours  print(f EM:
{compute_exact(label, pred)} ) print(f F1:
{compute_f1(label, pred)} )
```

新興市場 :0
F1 :0.8

在幕後，這些函數首先通過刪除標點符號、修復空格和轉換為小寫來規範化預測和標籤。然後將規範化的字符串標記為詞袋，最後在標記級別計算指標。從這個簡單的例子中，我們可以看出 EM 是一個比 F1 分數嚴格得多的指標：將單個標記添加到預測中得到的 EM 為零。另一方面，F1 分數可能無法捕捉到真正錯誤的答案。例如，如果我們預測的答案範圍是“大約 6000 美元”，那麼我們得到：

```
pred = 大約 6000 美元  print(f EM:
{compute_exact(label, pred)} ) print(f F1:
{compute_f1(label, pred)} )
```

新興市
場 :0 F1 :0.4

因此，僅依賴 F1 分數會產生誤導，跟蹤這兩個指標是平衡低估 (EM) 和高估 (F1 分數) 模型性能之間權衡的好策略。

現在一般來說，每個問題都有多個有效答案，因此會為評估集中的每個問題-答案對計算這些指標，並從所有可能的答案中選擇最佳分數。然後通過對每個問答對的各個分數進行平均來獲得模型的總體 EM 和 F1 分數。

為了評估閱讀器，我們將創建一個包含兩個節點的新管道：一個閱讀器節點和一個用於評估閱讀器的節點。我們將使用EvalReader類，它從閱讀器中獲取預測併計算相應的 EM 和 F1 分數。為了與 SQuAD 評估進行比較，我們將使用存儲在 EvalAnswers 中的 top\_1\_em 和 top\_1\_f1 指標為每個查詢獲取最佳答案：

從haystack.eval導入EvalAnswers

```
def evaluate_reader(reader):
    score_keys = [ top_1_em , top_1_f1 ]
    eval_reader = EvalAnswers(skip_incorrect_retrieval=False) pipe = Pipeline()
    pipe.add_node(component=reader, name= QAReader , inputs=[ 檢索 ])
    pipe.add_node(component=eval_reader, name= EvalReader , inputs=[ QAReader ])

    對於labels_agg中的l : doc =
        document_store.query(l.question,
                               filters={ question_id :[l.origin]})=
        _ pipe.run(query=l.question, documents=doc, labels=l)

    返回{k:v for k,v in eval_reader._dict_.items() if k in score_keys}

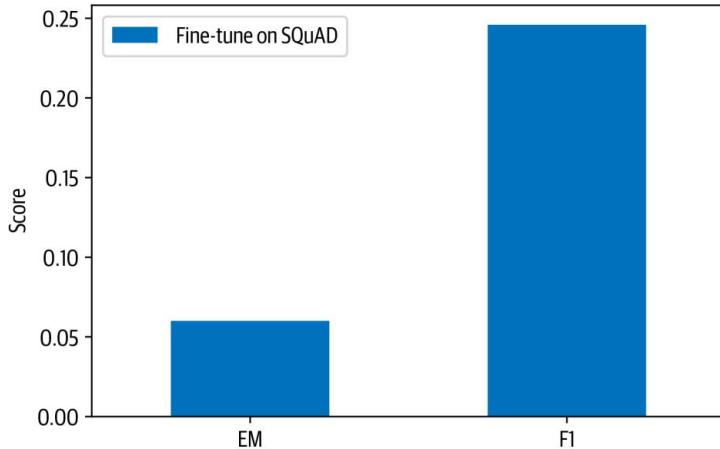
reader_eval={}
reader_eval[ 對 SQuAD 進行微調 ]= evaluate_reader(reader)
```

請注意，我們指定了skip\_incorrect\_retrieval=False。這是為了確保檢索器始終將上下文傳遞給讀取器（如在 SQuAD 評估中）。

現在我們已經通過閱讀器運行了每個問題，讓我們打印分數：

```
def plot_reader_eval(reader_eval): fig, ax=
    plt.subplots() df =
    pd.DataFrame.from_dict(reader_eval)
    df.plot(kind= bar ,ylabel= Score ,rot=0, ax=ax)
    ax.set_xticklabels([ EM , F1 ]) plt.legend(loc= upper left )
    plt.show()

plot_reader_eval (reader_eval)
```



好吧，微調模型在 SubjQA 上的表現似乎比在 SQuAD 2.0 上的表現要差得多，其中 MiniLM 的 EM 和 F1 分數分別為 76.1 和 79.5。性能下降的一個原因是客戶評論與生成 SQuAD 2.0 數據集的維基百科文章有很大不同，而且他們使用的語言通常是非正式的。另一個因素可能是我們數據集固有的主觀性，其中問題和答案都與維基百科中包含的事實信息不同。讓我們看看如何在數據集上微調模型以獲得更好的域自適應結果。

### 領域適應儘管在 SQuAD

上微調的模型通常可以很好地泛化到其他領域，但我們已經看到，對於 SubjQA，我們模型的 EM 和 F1 分數比 SQuAD 差得多。在其他提取 QA 數據集中也觀察到這種泛化失敗，並被理解為 transformer 模型特別擅長過度擬合 SQuAD 的證據。<sup>15</sup>改進閱讀器的最直接方法是在 SubjQA 上進一步微調我們的 MiniLM 模型訓練集。FARMReader 有一個專為此目的而設計的 train() 方法，它希望數據採用 SQuAD JSON 格式，其中每個項目的所有問答對都分組在一起，如圖 7-11 所示。

---

<sup>15</sup> D. Yogatama 等人，“學習和評估一般語言智能”，(2019)。

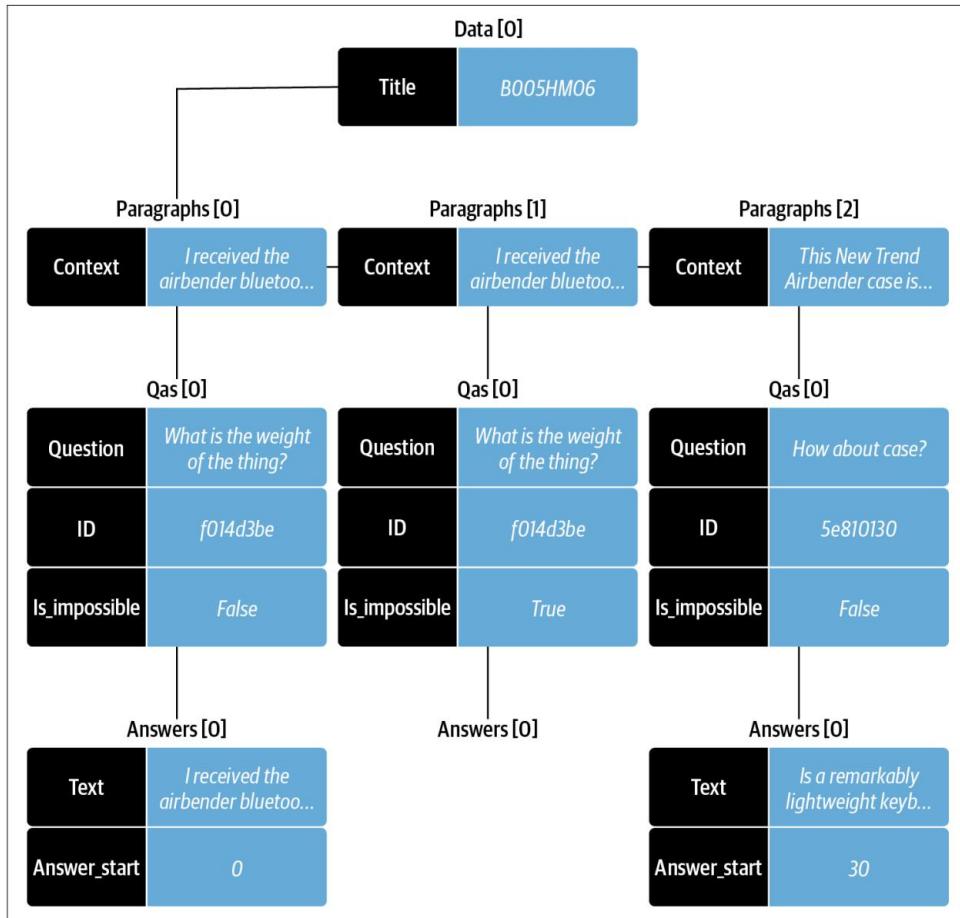


圖 7-11。SQuAD JSON 格式的可視化

這是一種相當複雜的數據格式，因此我們需要一些函數和一些 Pandas 魔法來幫助我們進行轉換。我們需要做的第一件事是實現一個函數，該函數可以創建與每個產品 ID 關聯的段落數組。該數組中的每個元素都包含一個上下文（即評論）和一個問答對的qas數組。這是一個構建段落數組的函數：

```
def create_paragraphs(df):
    paragraphs = []
    id2context = dict(zip(df['review_id'], df['context']))
    for review_id, review in id2context.items():
        qas = []

        # 過濾關於特定上下文的所有問答對

        # 構建 qas 數組
```

```

for qid, question in id2question.items(): # 過濾單個問題ID
    question_df = df.query(f id == {qid}).to_dict(orient= list ) ans_start_idxs = question_df[ answers.answer_start ][0].tolist() ans_text = question_df[ answers.text ][0].tolist()

    # 填寫可回答的問題if len(ans_start_idxs):
        answers = [
            { text :text, answer_start :answer_start} for text,
            answer_start in zip(ans_text,ans_start_idxs)]
        is_impossible = False else:
        answers = [] is_impossible = True
        # 添加問答對到 qas
        qas.append({ question :
        question, id :qid,
        "is_impossible" : is_impossible, "答案" :answers})
# 為段落添加上下文和問答對 paragraphs.append ({ qas :qas, context :
review}) return paragraphs

```

現在，當我們應用到與單個產品 ID 關聯的DataFrame的行時，我們得到 SQuAD 格式：

```

product = dfs[ train ].query( title == B00001P4ZH )
create_paragraphs(產品)

[ { qas :[{ question : 低音怎麼樣？ ,
    id : 2543d296da9766d8d17d040ecc781699 ,
    is_impossible :True, answers :[]}],
context : 我有 Koss 耳機..... , id :
d476830bf9282e2b9033e2bb44bbb995 ,
is_impossible :False, answers :[{ text : Bass
is weak as expected , answer_start :1302},{ text : 低音如預期的那樣弱，即使調高了 EQ ,
answer_start :1302}]},
context : 對於所有沒有嘗試過的人... },{ qas :[{ question :
低音怎麼樣？ ,
    id : 455575557886d6dfeea5aa19577e5de4 ,
    is_impossible :False, answers :[{ text : 唯一
的缺點是低音 ,
    answer_start :650}]},
context : 我有很多低於 100 美元的耳機..... }]

```

最後一步是將此函數應用於每個拆分的DataFrame中的每個產品 ID。以下convert\_to\_squad()函數執行此操作並將結果存儲在 electronics-{split}.json 文件中：

```

導入json

def convert_to_squad(dfs) :用於分
割， df in dfs.items():

```

```

subjqa_data = {}
# 為每個產品 ID 創建 `paragraphs` groups =
(df.groupby( title ).apply(create_paragraphs)
 .to_frame(name= 段落 ).reset_index())
subjqa_data[ data ] = groups.to_dict(orient= records )
# 將結果保存到磁盤with
open(f electronics-{split}.json , w+ ,encoding= utf-8 ) as f: json.dump(subjqa_data,f)

convert_to_squad (dfs)

```

現在我們有了正確格式的拆分，讓我們通過指定訓練和開發拆分的位置以及保存微調模型的位置來微調我們的閱讀器：

```

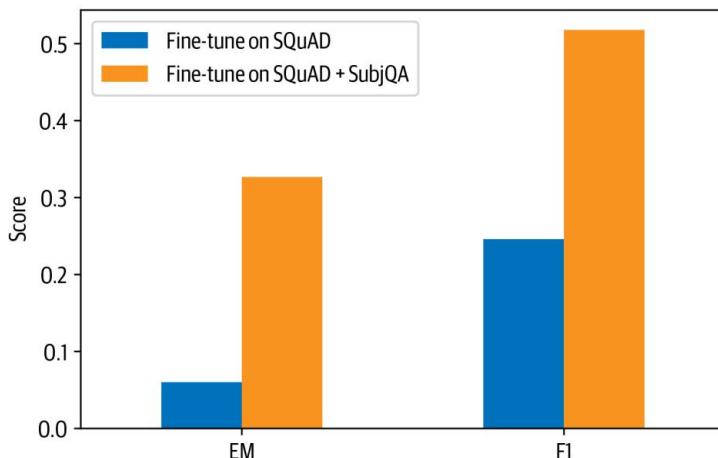
train_filename = electronics-train.json dev_filename
= electronics-validation.json

reader.train(data_dir= . ,use_gpu=True, n_epochs=1, batch_size=16,
            train_filename=train_filename, dev_filename=dev_filename)

```

閱讀器經過微調後，現在讓我們將其在測試集上的性能與我們的基線模型進行比較：

```
reader_eval[ “微調SQuAD + SubjQA” ] = evaluate_reader(reader) plot_reader_eval(reader_eval)
```



哇，領域適應使我們的 EM 得分提高了六倍，並且 F1 得分增加了一倍多！此時，您可能想知道為什麼我們不直接在 SubjQA 訓練集上微調預訓練語言模型。一個原因是我們在 SubjQA 中只有 1,295 個訓練示例，而 SQuAD 有超過 100,000 個，因此我們可能會遇到過度擬合的挑戰。儘管如此，讓我們來看看樸素的微調會產生什麼。為了公平比較，我們將使用相同的語言模型。

用於微調我們在 SQuAD 上的基線。和以前一樣，我們將使用 FARMReader 加載模型：

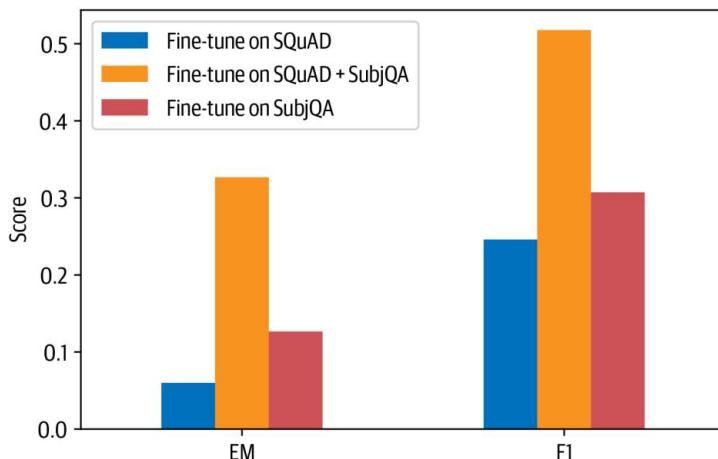
```
minilm_ckpt = "microsoft/MiniLM-L12-H384-uncased"
minilm_reader = FARMReader(model_name_or_path=minilm_ckpt, progress_bar=False,
                           max_seq_len=max_seq_length, doc_stride=doc_stride,
                           return_no_answer=True)
```

接下來，我們對一個 epoch 進行微調：

```
minilm_reader.train(data_dir='.', use_gpu=True, n_epochs=1, batch_size=16,
                     train_filename=train_filename, dev_filename=dev_filename)
```

並包括對測試集的評估：

```
reader_eval[["SubjQA微調"]] = evaluate_reader(minilm_reader) plot_reader_eval(reader_eval)
```



我們可以看到，直接在 SubjQA 上微調語言模型會導致比在 SQuAD 和 SubjQA 上微調更差的性能。



在處理小型數據集時，最佳做法是在評估轉換器時使用交叉驗證，因為它們很容易過度擬合。您可以在 **FARM** 存儲庫中找到有關如何使用 SQuAD 格式的數據集執行交叉驗證的示例。

評估整個 QA 流水線現在我們已經了解瞭如何單獨

評估讀取器和檢索器組件，讓我們將它們結合在一起來衡量流水線的整體性能。為此，我們需要使用讀取器及其節點的節點來擴充我們的檢索器管道

評估。我們已經看到我們在  $k = 10$  時獲得了幾乎完美的召回率，因此我們可以修正這個值並評估這對閱讀器性能的影響（因為與 SQuAD 式評估相比，它現在每個查詢都會接收多個上下文）：

```
# 初始化檢索器管道pipe =
EvalRetrieverPipeline(es_retriever)
# 為閱讀器添加節點eval_reader
= EvalAnswers()
pipe.pipeline.add_node(component=reader, name= QAReader ,
    inputs=[ EvalRetriever ])
pipe.pipeline.add_node(component=eval_reader, name= EvalReader ,
    inputs=[ QAReader ])
# 評價！
run_pipeline (管道)
# 從閱讀器中提取指標reader_eval[ QA
Pipeline (top-1) ]={ k:v for k,v in eval_reader.__dict__.items() if k in [ top_1_em ,
    top_1_f1 ]}
```

然後我們可以比較模型的前 1 個 EM 和 F1 分數，以預測圖 7-12 中檢索器返回的文檔中的答案。

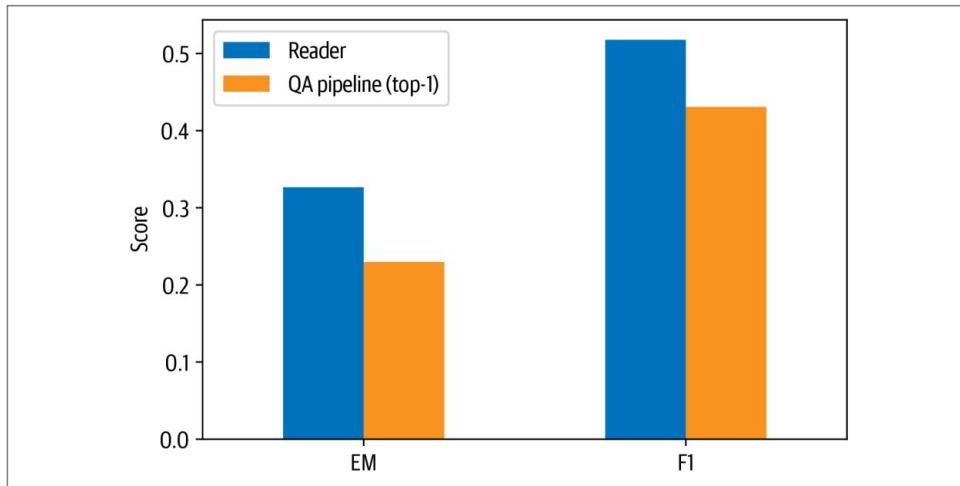


圖 7-12 閱讀器的 EM 和 F1 分數與整個 QA 流水線的比較

從這個圖中我們可以看到獵犬對整體表現的影響。

特別是，與匹配問題上下文對相比，整體性能有所下降，就像在 SQuAD 式評估中所做的那樣。這可以通過增加允許讀者預測的可能答案的數量來避免。

到目前為止，我們只從上下文中提取了答案範圍，但一般來說，答案的點點滴滴可能散佈在整個文檔中。

我們希望我們的模型將這些片段綜合成一個連貫的答案。讓我們看看如何使用生成式 QA 來成功完成此任務。

## 超越提取式 QA

將答案提取為文檔中文本跨度的一種有趣替代方法是使用預訓練語言模型生成它們。這種方法通常被稱為抽像或生成 QA，並且有可能產生措辭更好的答案，從而綜合多個段落的證據。雖然不如提取式 QA 成熟，但這是一個快速發展的研究領域，所以當你閱讀本文時，這些方法很可能會在行業中得到廣泛採用！在本節中，我們將簡要介紹當前最先進的技術：檢索增強生成 (RAG)。<sup>16</sup>

RAG 擴展了我們在本章中看到的經典檢索器-讀取器架構，將讀取器替換為生成器並使用 DPR 作為檢索器。生成器是一個預訓練的序列到序列轉換器，如 T5 或 BART，它從 DPR 接收文檔的潛在向量，然後根據查詢和這些文檔迭代生成答案。由於 DPR 和生成器是可微分的，因此整個過程可以端到端地進行微調，如圖 7-13 所示。

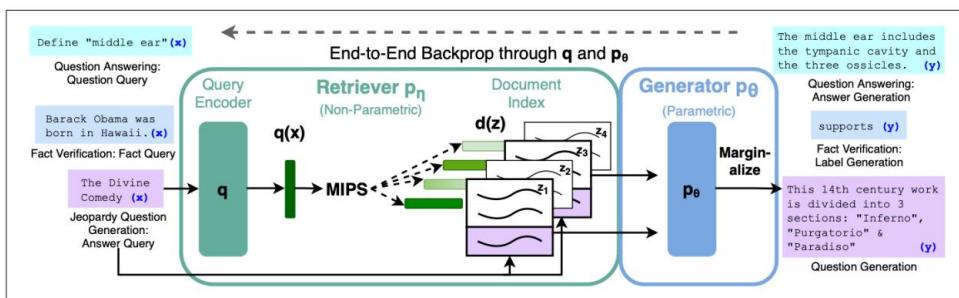


圖 7-13。用於端到端微調檢索器和生成器的 RAG 架構（由 Ethan Perez 提供）

為了展示 RAG 的運行，我們將使用之前的 DPRRetriever，所以我們只需要實例化一個生成器。RAG 型號有兩種可供選擇：

### RAG-Sequence

使用相同的檢索文檔生成完整答案。特別地，來自檢索器的前  $k$  個文檔被饋送到生成器，生成器為每個文檔生成一個輸出序列，並將結果邊緣化以獲得最佳答案。

16 P. Lewis 等人，“用於知識密集型 NLP 任務的檢索增強生成”，(2020)。

## RAG-代幣

可以使用不同的文檔來生成答案中的每個標記。這允許生成器從多個文檔中合成證據。

由於 RAG-Token 模型往往比 RAG-Sequence 模型表現更好，我們將使用在 NQ 上微調的令牌模型作為我們的生成器。在 Haystack 中實例化生成器類似於實例化讀取器，但我們沒有為上下文上的滑動窗口指定 max\_seq\_length 和 doc\_stride 參數，而是指定了控制文本生成的超參數：

從 [haystack.generator.transformers](#) 導入 RAGenerator

```
generator = RAGenerator(model_name_or_path= "facebook/rag-token-nq" ,  
                        embed_title=False, num_beams=5)
```

這裡 num\_beams 指定波束搜索中使用的波束數量（文本生成在第 5 章 中詳細介紹）。正如我們對 DPR 檢索器所做的那樣，我們沒有嵌入文檔標題，因為我們的語料庫始終按產品 ID 進行過濾。

接下來要做的是使用 Haystack 將檢索器和生成器連接在一起生成 QAP 流水線：

從 [haystack.pipeline](#) 導入 GenerativeQAPipeline

```
pipe = GenerativeQAPipeline (生成器=生成器, 檢索器=dpr_retriever)
```



在 RAG 中，查詢編碼器和生成器都是端到端訓練的，而上下文編碼器是凍結的。在 Haystack 中，GenerativeQAPipeline 使用來自 RAGenerator 的查詢編碼器和來自 DensePassageRetriever 的上下文編碼器。

現在讓我們通過輸入之前關於 Amazon Fire 平板電腦的一些查詢來給 RAG 一個旋轉。為了簡化查詢，我們將編寫一個簡單的函數來接受查詢並打印出最佳答案：

```
def generate_answers(query, top_k_generator=3): preds =  
    pipe.run(query=query, top_k_generator=top_k_generator, top_k_retriever=5,  
            filters=[ item_id : [ B0074BW614 ]])  
    print(f Question: {preds[ query ]}\n ) for idx  
    in range(top_k_generator): print(f Answer {idx+1}:  
        {preds[ answers ][idx][ answer ]} )
```

好的，現在我們準備好對其進行測試：

生成答案（查詢）

問：讀書好不好？

答1：畫面絕對漂亮

答案2 :屏幕絕對漂亮  
答案 3 :Kindle fire

這個結果對於一個答案來說並不算太糟糕，但它確實表明問題的主觀性質使生成器感到困惑。讓我們嘗試一些更實際的東西：

generate\_answers( **主要缺點是什麼？** )

問：主要缺點是什麼？

答案一：價格  
答案2：不支持閃光燈  
答案3：費用

這更明智！為了獲得更好的結果，我們可以在 SubjQA 上端到端地微調 RAG；我們將把它留作練習，但如果您有興趣探索它，**Transformers 存儲庫**中有腳本幫助您入門。



## 結論

好吧，那是一次旋風般的 QA 之旅，您可能還有很多問題需要回答（雙關語！）。在本章中，我們討論了兩種 QA 方法（提取和生成）並研究了兩種不同的檢索算法（BM25 和 DPR）。在此過程中，我們看到域適應可以是一種簡單的技術，可以顯著提高我們的 QA 系統的性能，並且我們研究了一些用於評估此類系統的最常見指標。儘管我們關注的是封閉域 QA（即電子產品的單一域），但本章中的技術可以很容易地推廣到開放域案例；我們推薦閱讀 Cloudera 優秀的 **Fast Forward QA 系列**查看涉及的內容。

在野外部署 QA 系統可能是一件棘手的事情，而且我們的經驗是，價值的很大一部分來自於首先為最終用戶提供有用的搜索功能，然後是提取組件。在這方面，讀者可以以新穎的方式使用，而不是回答按需用戶查詢。例如，**Grid Dynamics**的研究人員能夠使用他們的閱讀器自動提取客戶目錄中每種產品的一組優缺點。他們還表明，閱讀器可用於通過創建諸如“哪種相機？”之類的查詢來以零鏡頭方式提取命名實體。鑑於其初期和微妙的故障模式，我們建議僅在其他兩種方法都用盡後才探索生成式 QA。這種用於解決 QA 問題的“需求層次結構”**如圖 7-14 所示**。

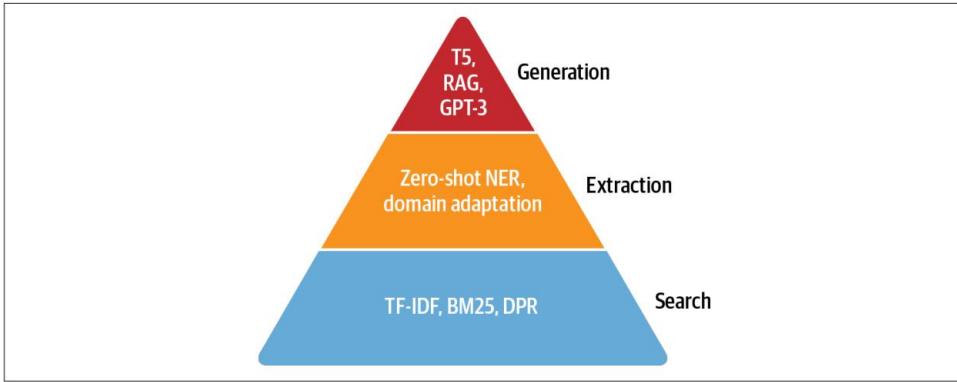


圖 7-14 質量保證需求層次

展望未來，一個令人興奮的研究領域是多模式 QA，它涉及對文本、表格和圖像等多種模式的 QA。正如 MultiModalQA 基準測試中所描述的那樣，<sup>17</sup>這樣的系統可以讓用戶回答複雜的問題，這些問題整合了不同模態的信息，比如“那幅用兩個手指觸摸的名畫是什麼時候完成的？”另一個具有實際業務應用的領域是知識圖的 QA，其中圖的節點對應於現實世界的實體，它們的關係由邊定義。通過將類事實編碼為（主語、謂語、賓語）三元組，可以使用該圖來回答有關缺失元素的問題。有關將轉換器與知識圖相結合的示例，請參閱 [Haystack 教程](#)。一個更有前途的方向是自動問題生成，作為一種使用未標記數據或數據增強進行某種形式的無監督/弱監督訓練的方法。最近的兩個例子包括關於跨語言設置的可能已回答問題（PAQ）基準和綜合數據增強的論文。<sup>18</sup>

在本章中，我們已經看到，為了成功地將 QA 模型用於真實世界的用例，我們需要應用一些技巧，例如實施快速檢索管道以近乎實時地進行預測。儘管如此，將 QA 模型應用於少量預選文檔在生產硬件上可能需要幾秒鐘的時間。雖然這聽起來可能並不多，但想像一下，如果您必須等待幾秒鐘才能獲得 Google 搜索結果，您的體驗會有多麼不同。幾秒鐘的等待時間可以決定您的 transformer 驅動的應用程序的命運。在下一章中，我們將了解一些進一步加速模型預測的方法。

<sup>17</sup> A. Talmor 等人， “MultiModalQA：基於文本、表格和圖像的複雜問答”， (2021)。

<sup>18</sup> P. Lewis 等人， “PAQ：6500 萬個可能提出的問題以及您可以用它們做什麼”， (2021); A. Riabi 等人， “零樣本跨語言問答的綜合數據增強”， (2020)。

## 第 8 章

# 使變壓器在生產中高效

在前面的章節中，您已經了解瞭如何微調 transformer 以在廣泛的任務中產生出色的結果。然而，在許多情況下，準確度（或者你正在優化的任何指標）是不夠的；如果最先進的模型太慢或太大而無法滿足應用程序的業務需求，那麼它就不是很有用。一個明顯的替代方案是訓練一個更快、更緊湊的模型，但模型容量的減少往往伴隨著性能的下降。

那麼，當您需要一個快速、緊湊但高度準確的模型時，您可以做什麼呢？

在本章中，我們將探索四種可用於加速預測並減少 transformer 模型內存佔用的互補技術：知識蒸餾、量化、修剪和使用開放神經網絡交換 (ONNX) 格式的圖形優化和 ONNX 運行時 (ORT)。我們還將看到如何將其中一些技術結合起來以產生顯著的性能提升。例如，這是 Roblox 工程團隊在他們的文章 “[我們如何擴展 Bert 以在 CPU 上處理 1+十億個每日請求](#)” 中採用的方法，如圖 8-1 所示，他們發現結合知識蒸餾和量化使他們能夠將 BERT 分類器的延遲和吞吐量提高 30 倍以上！

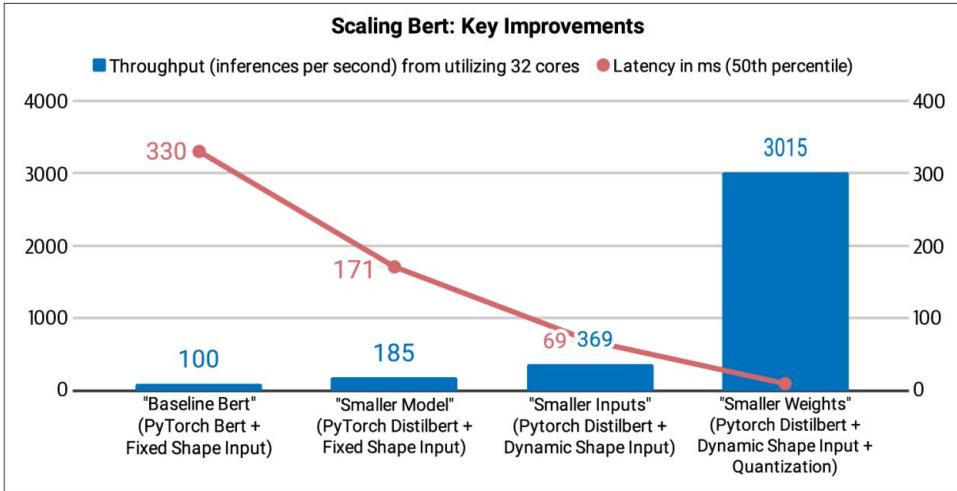


圖 8-1。Roblox 如何通過知識蒸餾、動態填充和權重量化擴展 BERT（照片由 Roblox 員工 Quoc N. Le 和 Kip Kaeler 提供）

為了說明與每種技術相關的好處和權衡，我們將使用意圖檢測作為案例研究；這是基於文本的助手的重要組成部分，低延遲對於保持實時對話至關重要。

在此過程中，您將學習如何創建自定義訓練器、執行高效的超參數搜索，並了解如何使用 Transformers 實施前沿研究。讓我們開始吧！



## 意圖檢測作為案例研究

假設我們正在嘗試為我們公司的呼叫中心構建一個基於文本的助手，以便客戶可以查詢他們的帳戶餘額或進行預訂，而無需與人工座席交談。為了理解客戶的目標，我們的助手需要能夠將各種自然語言文本分類為一組預定義的動作或意圖。例如，客戶可能會發送如下關於即將到來的旅行的消息：

嘿，我想從 11 月 1 日到 11 月 15 日在巴黎租一輛車，我需要一輛 15 座客車

我們的意圖分類器可以自動將其歸類為租車意圖，然後觸發操作和響應。為了在生產環境中保持穩健，我們的分類器還需要能夠處理範圍外的查詢，在這種情況下，客戶提出的查詢不屬於任何預定義的意圖，系統應該產生一個後備響應。例如，在圖 8-2 所示的第二種情況中，客戶提出了有關體育的問題（超出範圍），文本助手錯誤地將其歸類為已知的範圍內意圖之一併返回

發薪日回應。在第三種情況下，文本助手經過培訓可以檢測超出範圍的查詢（通常標記為單獨的類）並告知客戶它可以回答有關哪些主題的問題。



圖 8-2。人類（右）和基於文本的助手（左）之間的三種個人理財交流（由 Stefan Larson 等人提供）

作為基線，我們對基於 BERT 的模型進行了微調。該模型在 CLINC150 數據集上的準確率達到了 94% 左右。該數據集包括 22,500 個範圍內查詢，涉及 150 個意圖和 10 個領域，例如銀行業和旅遊，以及包括屬於 oos 意圖類的 1,200 個超出範圍的查詢。在實踐中，我們也會收集自己的內部數據集，但使用公共數據是快速迭代和生成初步結果的好方法。

首先，讓我們從 Hugging Face Hub 下載經過微調的模型，並將其包裝在用於文本分類的管道中：

[從變壓器導入管道](#)

```
bert_ckpt = transformersbook/bert-base-uncased-finetuned-clinc pipe =
pipeline( 文本分類 , model=bert_ckpt)
```

現在我們有了一個管道，我們可以通過一個查詢來從模型中獲取預測的意圖和置信度分數：

---

<sup>1</sup> S. Larson 等人，“意圖分類和範圍外預測的評估數據集”，（2019）。

```
query =     嘿 ·我想在 11 月 1 日至 11 月 15 日期間租一輛車
巴黎和我需要一輛 15 座客車      管道 (查詢)
```

```
[[ label : car_rental , score :0.549003541469574]]
```

太好了，`car_rental`意圖是有道理的。現在讓我們看看創建一個基準，我們可以用它來評估我們的基線模型的性能。

## 創建性能基準

與其他機器學習模型一樣，在生產環境中部署 Transformer 涉及在多個約束之間進行權衡，最常見的是<sup>2</sup>

模型性能我們的模

型在反映生產數據的精心設計的測試集上的表現如何？當犯錯的成本很高（並且最好在循環中有人來減輕），或者當我們需要對數百萬個示例進行推理並且對模型指標進行小的改進可以轉化為巨大的收益時，這一點尤其重要。總計的。

延遲 我們的

模型能以多快的速度提供預測？我們通常關心處理大量流量的實時環境中的延遲，例如 Stack Overflow 如何需要一個分類器來快速檢測網站上不受歡迎的評論。

內存 我

們如何部署需要千兆字節磁盤存儲和 RAM 的十億參數模型，如 GPT-2 或 T5？內存在移動或邊緣設備中起著特別重要的作用，在這些設備中，模型必須在無法訪問強大的雲服務器的情況下生成預測。

未能解決這些限制可能會對您的應用程序的用戶體驗產生負面影響。更常見的是，運行可能只需要處理少量請求的昂貴雲服務器可能會導致成本激增。為了探索如何使用各種壓縮技術優化這些約束中的每一個，讓我們首先創建一個簡單的基準來測量給定管道和測試集的每個數量。下面的類給出了我們需要的框架：

```
類PerformanceBenchmark: def
    __init__(self, pipeline, dataset, optim_type= BERT baseline ): self.pipeline = pipeline
```

---

<sup>2</sup>正如 Emmanuel Ameisen 在構建機器學習驅動的應用程序 (O'Reilly) 中所述，業務或產品指標是最重要的考慮因素。畢竟，如果您的模型不能解決您的企業關心的問題，那麼您的模型再準確也無所謂。在本章中，我們假設您已經定義了對您的應用程序重要的指標，並專注於優化模型指標。

```

    self.dataset = 數據集 self.optim_type
    = optim_type

def compute_accuracy(自我) :
    # 我們稍後會定義這個
    經過

def compute_size(自身) :
    # 我們稍後會定義這個
    經過

def time_pipeline(自我) :
    # 我們將在稍後定義這個通道

def run_benchmark(self): metrics =
    {} metrics[self.optim_type] =
    self.compute_size() metrics[self.optim_type].update(self.time_pipeline())
    metrics[self.optim_type].update(self.compute_accuracy())回報指標

```

我們定義了一個optim\_type參數來跟蹤我們將在本章中介紹的不同優化技術。我們將使用run\_benchmark()方法收集字典中的所有指標，鍵由optim\_type給出。

現在讓我們通過計算測試集上的模型精度來為這個類的骨架添加一些內容。首先，我們需要一些數據進行測試，因此讓我們下載用於微調我們的基線模型的 CLINC150 數據集。我們可以使用 Datasets 從 Hub 獲取數據集，如下所示：



從數據集導入load\_dataset

```
clinc = load_dataset('clinc_oos', 'plus')
```

這裡，plus配置是指包含超出範圍的訓練示例的子集。CLINC150 數據集中的每個示例都包含文本列中的查詢及其對應的意圖。我們將使用測試集來對我們的模型進行基準測試，所以讓我們看一下數據集的一個示例：

樣本= clinc['測試'][42]樣本

```
{ intent : 133, text : '從我的支票賬戶轉 100 美元到儲蓄賬戶' }
```

意圖以 ID 的形式提供，但我們可以通過訪問數據集的features屬性輕鬆獲取到字符串的映射（反之亦然）：

```
intents = clinc['test'].features['intent']
intents.int2str(sample['intent'])
```

轉移

現在我們對CLINC150數據集中的內容有了基本的了解，下面我們來實現PerformanceBenchmark的compute\_accuracy()方法。由於數據集在意圖類別之間是平衡的，因此我們將使用準確性作為我們的指標。我們可以使用數據集加載此指標，如下所示：



[從數據集導入load\\_metric](#)

```
accuracy_score = load_metric('accuracy')
```

準確性指標期望預測和參考（即地面實況標籤）為整數。我們可以使用管道從文本字段中提取預測，然後使用intents對象的str2int()方法將每個預測映射到其對應的ID。以下代碼在返回數據集的準確性之前收集列表中的所有預測和標籤。讓我們也將它添加到我們的PerformanceBenchmark類中：

```
def compute_accuracy(self):
    # 這會覆蓋 PerformanceBenchmark.compute_accuracy() 方法
    preds, labels = [], []
    for example in self.dataset:
        pred = self.pipeline(example['text'])[0]['label']
        label = example['intent']
        preds.append(intents.str2int(pred))
        labels.append(label)
    accuracy = accuracy_score.compute(predictions=preds, references=labels)
    print(f'測試集的準確性 - {accuracy:.3f} 返回精度')
    return accuracy
```

`PerformanceBenchmark.compute_accuracy = compute_accuracy`

接下來，讓我們通過使用 PyTorch 中的`torch.save()`函數將模型序列化到磁盤來計算模型的大小。在底層，`torch.save()`使用 Python 的 pickle 模塊，可用於保存從模型到張量再到普通 Python 對象的任何內容。在 PyTorch 中，推薦的保存模型的方法是使用它的`state_dict`，它是一個 Python 字典，將模型中的每一層映射到它的可學習參數（即權重和偏差）。讓我們看看我們的基線模型的`state_dict`中存儲了什麼：

`list(pipe.model.state_dict().items())[42]`

```
(bert.encoder.layer.2.attention.self.value.weight, 張量
([[-1.0526e-02, -3.2215e-02, 2.2097e-02, ..., -6.0953e-03, 4.6521e-03, 2.9844e-02],
 [-1.4964e-02, -1.0915e-02, 5.2396e-04, ..., 3.2047e-05, -2.6890e-02,
 -2.1943e-02], [-2.9640e-02, -3.7842e-03, -1.2582e-02, ..., -1.0917e-02,
 3.1152e-02, -9.7786e-03], ...,
 [-1.5116e-02, -3.3226e-02, 4.2063e-02, ..., -5.2652e-03, 1.1093e-02, 2.9703e-03],
```

```
[ -3.6809e-02, 5.6848e-02, -2.6544e-02, ..., -4.0114e-02, 6.7487e-03, 1.0511e-03],
 [-2.4961e-02, 1.4747e-03, -5.4271e-02, ..., 2.0004e-02, 2.3981e-02,
 -4.2880e-02]]))
```

我們可以清楚地看到每個鍵/值對對應於 BERT 中的特定層和張量。所以如果我們保存我們的模型：

```
torch.save(pipe.model.state_dict(), model.pt )
```

然後我們可以使用Python 的pathlib模塊中的Path.stat()函數來獲取有關底層文件的信息。特別是Path( model.pt )。st\_size將為我們提供以字節為單位的模型大小。讓我們將所有這些放在compute\_size()函數中並將其添加到PerformanceBenchmark：

[從路徑庫導入火炬](#)

導入路徑

```
def compute_size (自身) :
    這會覆蓋 PerformanceBenchmark.compute_size() 方法      state_dict =
    self.pipeline.model.state_dict() tmp_path = Path( model.pt ) torch.save(state_dict,
    tmp_path)

    # 以兆字節為單位計算大小size_mb =
    Path(tmp_path).stat().st_size / (1024 * 1024)
    # 刪除臨時文件tmp_path.unlink()
    print(f 模型大小(MB) -
    {size_mb:.2f} ) return { size_mb :size_mb}
```

```
PerformanceBenchmark.compute_size = compute_size
```

最後讓我們實現time\_pipeline()函數，以便我們可以計算每個查詢的平均延遲時間。對於此應用程序，延遲是指將文本查詢提供給管道並從模型返回預測意圖所需的時間。在引擎蓋下，管道還標記文本，但這比生成預測快大約一千倍，因此對整體延遲的貢獻可以忽略不計。測量代碼片段執行時間的一種簡單方法是使用Python時間模塊中的perf\_counter()函數。此函數比time.time()函數具有更好的時間分辨率，非常適合獲得精確的結果。

我們可以使用perf\_counter()通過傳遞測試查詢併計算開始和結束之間的時間差（以毫秒為單位）來為管道計時：

[從時間導入perf\\_counter](#)

```
為了 _ 在範圍內 (3) :
    start_time = perf_counter ()
    _ =管道 (查詢)
```

```

latency = perf_counter() - start_time print(f 延遲
(ms) - {1000 * latency:.3f} )

延遲 (毫秒) - 85.367 延遲 (毫
秒) - 85.241 延遲 (毫秒) -
87.275

```

這些結果顯示出相當大的延遲分佈，表明每次運行代碼時，對通過管道的單次傳遞進行計時可能會產生截然不同的結果。因此，我們將收集多次運行的延遲，然後使用生成的分佈來計算均值和標準差，這將使我們了解值的分佈。下面的代碼完成了我們所需要的，包括一個在執行實際定時運行之前預熱 CPU 的階段：

```

將numpy導入為np

def time_pipeline(self, query= 我的帳戶的密碼是多少？ ):      這會覆蓋
    PerformanceBenchmark.time_pipeline() 方法      latencies = []

#範圍內的預
熱(10):=self.pipeline(query)

# 範圍內的定時
運行 (100) :
    start_time = perf_counter() =
    _ self.pipeline(query) latency =
    perf_counter() - start_time latencies.append(latency)

# 計算運行統計數據time_avg_ms
= 1000 * np.mean(latencies) time_std_ms = 1000 *
np.std(latencies) print(f 平均延遲 (ms) -
{time_avg_ms:.2f} +/- {time_std_ms:.2f} )返回[ time_avg_ms :time_avg_ms,  time_std_ms :
time_std_ms]

PerformanceBenchmark.time_pipeline =時間流水線

```

為了簡單起見，我們將使用相同的查詢值來對我們所有的模型進行基準測試。一般來說，延遲將取決於查詢長度，一個好的做法是使用它們在生產環境中可能遇到的查詢來對模型進行基準測試。

現在我們的PerformanceBenchmark類已經完成，讓我們試一試吧！讓我們從對 BERT 基線進行基準測試開始。對於基線模型，我們只需要傳遞管道和我們希望對其執行基準測試的數據集。我們將在perf\_metrics字典中收集結果以跟蹤每個模型的性能：

```

pb = PerformanceBenchmark(pipe, clinc[ 測試 ]) perf_metrics
= pb.run_benchmark()

模型大小 (MB) - 418.16 平均延遲
(ms) - 54.20 +/- 1.91 測試集準確度 - 0.867

```

現在我們有了一個參考點，讓我們看看我們的第一個壓縮技術：知識蒸餾。



平均延遲值將根據您運行的硬件類型而有所不同。例如，您通常可以通過在 GPU 上運行推理來獲得更好的性能，因為它支持批處理。就本章而言，重要的是模型之間延遲的相對差異。一旦我們確定了性能最佳的模型，我們就可以探索不同的後端以在需要時減少絕對延遲。

## 通過知識蒸餾縮小模型

知識蒸餾是一種通用方法，用於訓練較小的學生模型來模仿較慢、較大但表現較好的教師的行為。最初於 2006 年在集成模型的背景下引入<sup>3</sup>，後來在 2015 年的一篇著名論文中得到推廣，該論文將該方法推廣到深度神經網絡並將其應用於圖像分類和自動語音識別。<sup>4</sup>

鑑於預訓練語言模型的趨勢是參數數量不斷增加（在撰寫本文時最大的參數數量超過一萬億），<sup>5</sup>知識蒸餾也成為壓縮這些龐大模型並使其更適合於構建實際應用。

用於微調的知識蒸餾那麼，在訓練過程中，知識

實際上是如何“蒸餾”或從教師轉移到學生的呢？對於像微調這樣的監督任務，主要思想是用來自教師的“軟概率”分佈來增加地面實況標籤，為學生提供補充信息來學習。例如，如果我們的基於 BERT 的分類器將高概率分配給多個意圖，那麼這可能表明這些意圖在特徵空間中彼此靠近。通過訓練學生模仿這些概率，目標是提煉出一些教師學到的“暗知識”<sup>6</sup>，即僅從標籤中無法獲得的知識。

<sup>3</sup> C. Buciluă 等人，“模型壓縮”，第 12 屆 ACM SIGKDD 知識發現和數據挖掘國際會議論文集（2006 年 8 月）：535–541。<https://doi.org/10.1145/1150402.1150464>。

<sup>4</sup> G. Hinton、O. Vinyals 和 J. Dean，“在神經網絡中提煉知識”，(2015)。

<sup>5</sup> W. Fedus、B. Zoph 和 N. Shazeer，“開關變壓器：使用簡單的方法擴展到萬億參數模型和有效的稀疏性”，(2021)。

<sup>6</sup> Geoff Hinton 在一次演講中創造了這個詞指軟化概率揭示隱藏的觀察結果。老師的書房知識。

從數學上講，它的工作方式如下。假設我們將輸入序列  $x$  提供給教師以生成一個對數向量  $\mathbf{z} = [z_1, \dots, z_N]$ 。我們可以通過應用 softmax 函數將這些 logits<sup>7</sup>轉換為概率：

$$\frac{\exp z_i(x)}{\sum_j \exp z_j(x)}$$

然而，這並不是我們想要的，因為在許多情況下，老師會給一個班級分配一個高概率，而所有其他班級的概率接近於零。當發生這種情況時，教師不會提供超出真實標籤之外的更多信息，因此我們在應用 softmax 之前通過使用溫度超參數  $T$  縮放 logits 來“軟化”概率：<sup>7</sup>

$$p_i(x) = \frac{\exp z_i(x)/T}{\sum_j \exp z_j(x)}$$

如圖 8-3 所示，更高的  $T$  值會在類上產生更柔和的概率分佈，並揭示更多關於教師為每個訓練示例學習的決策邊界的信息。當  $T = 1$  時，我們恢復原始的 softmax 分佈。

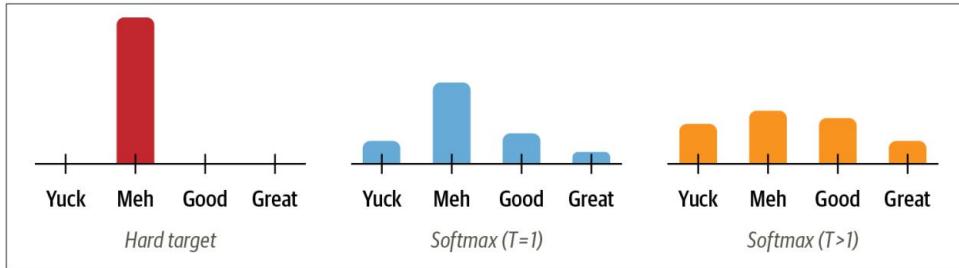


圖 8-3。one-hot 編碼的硬標籤（左）softmax 概率（中）和軟化類概率（右）的比較

由於學生也產生自己的軟化概率  $q_i(x)$ ，我們可以使用 Kullback-Leibler (吉隆坡) divergence 衡量兩個概率分佈之間的差異：

$$DKL(p||q) = \sum_i p_i \log \frac{p_i}{q_i}$$

<sup>7</sup>我們在第 5 章的文本生成上下文中也遇到了溫度。

有了 KL 散度，我們可以計算出當我們近似教師與學生的概率分佈時損失了多少。這使我們能夠定義知識蒸餾損失：

$$L_{KD} = T \cdot D_{KL}$$

其中  $T^{-\frac{2}{T}}$  是一個歸一化因子，用於解釋以下事實：

軟標籤產生的梯度按  $1/T$  損失縮放，然後是具有通常的交叉熵。

對於分類任務，學生熵損失的蒸餾損失的加權平均值

ground truth標籤的 LCE：

$$L_{Student} = \alpha L_{CE} + 1 - e^{-L_{KD}}$$

其中  $\alpha$  是控制每個損失的相對強度的超參數。整個過程的示意圖如圖 8-4 所示；在推理時將溫度設置為 1 以恢復標準的 softmax 概率。

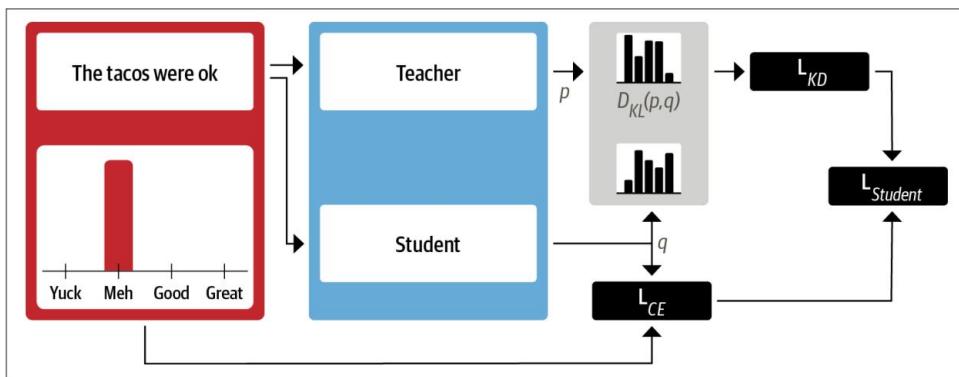


圖 8-4 知識蒸餾過程

## 用於預訓練的知識蒸餾知識蒸餾也可以在預訓練

練過程中使用，以創建一個通用的學生，隨後可以對下游任務進行微調。在這種情況下，教師是像 BERT 這樣的預訓練語言模型，它將有關掩蔽語言建模的知識轉移給學生。例如，在 DistilBERT 論文 8 中，掩碼語言建模損失  $L_{mlm}$  增加了來自知識蒸餾的項和余弦嵌入損失  $L_{cos} = 1 - \cos hs, ht$  以對齊教師和學生之間隱藏狀態向量的方向：

( )

$$L_{DistilBERT} = \alpha L_{mlm} + \beta L_{KD} + \gamma L_{cos}$$

由於我們已經有了一個微調的基於 BERT 的模型，讓我們看看如何使用知識蒸餾來微調一個更小更快的模型。為此，我們需要一種方法來使用 LKD 項來增加交叉熵損失。幸運的是，我們可以通過創建自己的訓練器來做到這一點！

## 創建知識蒸餾培訓師

為了實現知識蒸餾，我們需要向 Trainer 基類添加一些東西：

- 新的超參數  $\alpha$  和  $T$ ，它們控制蒸餾損失的相對權重以及標籤的概率分佈應該被平滑多少
- 微調的教師模型，在我們的案例中是基於 BERT 的
- 結合了交叉熵損失和知識的新損失函數  
蒸餾損失

添加新的超參數非常簡單，因為我們只需要繼承 TrainingArguments 將它們作為新屬性包括在內：

[從變形金剛導入TrainingArguments](#)

```
類DistillationTrainingArguments (TrainingArguments) :
    def __init__(self, *args, alpha=0.5, temperature=2.0, **kwargs):
        super().__init__(*args, **kwargs)
        self.alpha = alpha
        self.temperature = 減度
```

---

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

對於訓練器本身，我們需要一個新的損失函數。實現這一點的方法是將Trainer子類化並重寫compute\_loss()方法以包含知識蒸餾損失項LKD：

```
import torch.nn as nn
import torch.nn.functional as F from
transformers import Trainer

類DistillationTrainer(Trainer): def __init__(self,
    *args, teacher_model=None, **kwargs): super().__init__(*args, **kwargs)
    self.teacher_model = teacher_model

def compute_loss(self, model, inputs, return_outputs=False):
    outputs_stu = 模型 (**輸入)
    # 從學生 loss_ce = outputs_stu.loss 中提取交叉熵損失和logits _ _ _

    # 軟化概率併計算蒸餾損失 loss_fct = nn.KLDivLoss(reduction= batchmean )
    loss_kd = self.args.temperature ** 2 * loss_fct(F.log_softmax(logits_stu /
        self.args.temperature, dim=-1), F.softmax(logits_tea /
        self.args.temperature, dim=-1))

    # 返回加權學生損失 loss = self.args.alpha *
    loss_ce + (1. - self.args.alpha) * loss_kd return (loss, outputs_stu) if return_outputs else loss
```

讓我們稍微解壓這段代碼。當我們實例化DistillationTrainer時，我們將teacher\_model參數傳遞給已經針對我們的任務進行微調的教師。

接下來，在compute\_loss()方法中，我們從學生和教師中提取logits，根據溫度對它們進行縮放，然後在將它們傳遞給PyTorch的nn.KLDivLoss()函數以計算KL散度之前使用softmax對其進行歸一化。nn.KLDivLoss()的一個怪癖是它期望對數概率形式的輸入和正常概率的標籤。這就是為什麼我們使用F.log\_softmax()函數來標準化學生的logits，而教師的logits使用標準softmax轉換為概率。nn.KLDivLoss()中的reduction=batchmean參數指定我們對批量維度的損失進行平均。



您還可以使用Transformers庫的Keras API執行知識蒸餾。為此，您需要實現一個自定義Distiller類，該類覆蓋tf.keras.Model()的train\_step()、test\_step()和compile()方法。請參閱[Keras文檔](#)有關如何執行此操作的示例。

選擇一個好的學生初始化既然我們有了自

定義訓練器，您可能會遇到的第一個問題是我們應該為學生選擇哪種預訓練語言模型？一般來說，我們應該為學生選擇一個較小的模型，以減少延遲和內存佔用。文獻中的一個很好的經驗法則是，當教師和學生屬於同一模型類型時，知識蒸餾效果最好。

9一個可能的原因是不同的模型類型，比如 BERT 和 RoBERTa，可能有不同的輸出嵌入空間，這阻礙了學生模仿老師的能力。在我們的案例研究中，教師是 BERT，因此 DistilBERT 是初始化學生的自然候選者，因為它的參數減少了 40%，並且已被證明在下游任務中取得了很好的結果。

首先，我們需要對查詢進行分詞和編碼，因此讓我們從 DistilBERT 實例化分詞器並創建一個簡單的 tokenize\_text() 函數來處理預處理：

從 [變形金剛導入 AutoTokenizer](#)

```
student_ckpt = "distilbert-base-uncased"
student_tokenizer = AutoTokenizer.from_pretrained(student_ckpt)

def tokenize_text(batch): 返回
    student_tokenizer(batch[ "text" ], truncation=True)

clinc_enc = clinc.map(tokenize_text, batched=True, remove_columns=[ "text" ])
clinc_enc = clinc_enc.rename_column( "intent" , "labels" )
```

在這裡，我們刪除了文本列，因為我們不再需要它，我們還將意圖列重命名為標籤，以便訓練器可以自動檢測到它。<sup>10</sup>

現在我們已經處理了我們的文本，接下來我們需要做的是為我們的 DistillationTrainer 定義超參數和 compute\_metrics() 函數。我們還將把我們所有的模型推送到 Hugging Face Hub，所以讓我們先登錄我們的帳戶：

從 [huggingface\\_hub導入 notebook\\_login](#)

```
notebook_login()
```

---

<sup>9</sup> Y. Kim 和 H. Awadalla，“FastFormers：用於自然語言下的高效轉換器模型”  
常設”，（2020）。

<sup>10</sup> 默認情況下，Trainer 在對分類任務進行微調時會查找名為標籤的列。您還可以通過指定 TrainingArguments 的 label\_names 參數來覆蓋此行為。

接下來，我們將定義訓練期間要跟蹤的指標。正如我們在性能基準測試中所做的那樣，我們將使用準確性作為主要指標。這意味著我們可以在我們將包含在 DistillationTrainer 中的 compute\_metrics()函數中重用我們的accuracy\_score()函數：

```
def compute_metrics (預測) :
    預測 .標籤= pred 預測= np.argmax (預
    測 .軸 = 1)返回accuracy_score.compute (預測 = 預測 ,參考 =
    標籤)
```

在這個函數中，來自序列建模頭的預測以 logits 的形式出現，因此我們使用 np.argmax ()函數來找到最有信心的類別預測並將其與基本事實標籤進行比較。

接下來我們需要定義訓練參數。為了熱身，我們將設置  $\alpha = 1$  以查看 DistilBERT 在沒有來自教師的任何信號的情況下的表現如何。<sup>11</sup>然後我們將微調模型推送到一個名為 distilbert-base-uncased finetuned-clinc 的新存儲庫，因此我們只需要在 DistillationTrainingArguments 的 output\_dir 參數中指定：

```
batch_size = 48

finetuned_ckpt = distilbert-base-uncased-finetuned-clinc student_training_args
= DistillationTrainingArguments(
    output_dir=finetuned_ckpt, evaluation_strategy = epoch ,
    num_train_epochs=5, learning_rate=2e-5, per_device_train_batch_size=batch_size,
    per_device_eval_batch_size=batch_size, alpha=1, weight_decay=0.01,
    push_to_hub=True)
```

我們還調整了一些默認的超參數值，例如迭代次數、權重衰減和學習率。接下來要做的是初始化學生模型。由於我們將與訓練器進行多次運行，因此我們將創建一個 student\_init() 函數來在每次新運行時初始化模型。當我們將此函數傳遞給 DistillationTrainer 時，這將確保我們每次調用 train() 方法時都會初始化一個新模型。

我們需要做的另一件事是為學生模型提供每個意圖和標籤 ID 之間的映射。這些映射可以從我們在管道中下載的 BERT 基礎模型中獲得：

```
id2label = pipe.model.config.id2label label2id =
pipe.model.config.label2id
```

---

<sup>11</sup>這種微調通用的、提煉的語言模型的方法有時被稱為“任務不可知論”蒸餾。

有了這些映射，我們現在可以使用我們在第3章和第4章中遇到的AutoConfig類帽子創建自定義模型配置。讓我們使用它為我們的學生創建一個包含標籤映射信息的配置：

#### [從變壓器導入AutoConfig](#)

```
num_labels = intents.num_classes
student_config = (AutoConfig
    .from_pretrained(student_ckpt, num_labels=num_labels,
        id2label=id2label, label2id=label2id))
```

在這裡，我們還指定了我們的模型應該期望的類數。然後我們可以將此配置提供給AutoModelForSequenceClassification類的from\_pretrained()函數，如下所示：

#### [從變壓器導入火炬](#)

#### 導入AutoModelForSequenceClassification

```
device = torch.device(  cuda  if torch.cuda.is_available() else  cpu  )

def student_init():返回
    (AutoModelForSequenceClassification
        .from_pretrained(student_ckpt, config=student_config.to(device)))
```

我們現在擁有蒸餾訓練器所需的所有成分，所以讓我們加載訓練器並進行微調：

```
teacher_ckpt =  transformersbook/bert-base-uncased-finetuned-clinc  teacher_model =
(AutoModelForSequenceClassification
    .from_pretrained(teacher_ckpt, num_labels=num_labels) .to(device))

distilbert_trainer = DistillationTrainer (model_init=student_init ,
    teacher_model=teacher_model , args=student_training_args ,
    train_dataset=clinc_enc[  train  ] , eval_dataset=clinc_enc[  validation  ] ,
    compute_metrics=compute_metrics , tokenizer=student_tokenizer)

distilbert_trainer.train()
```

Epoch	Training Loss	Validation 損失精度	
1	4.2923	3.289337	0.742258
2	2.6307	1.883680	0.828065
3	1.5483	1.158315	0.896774
4	1.0153	0.861815	0.909355
5	0.7958	0.777289	0.917419

與基於 BERT 的教師所達到的 94% 相比，驗證集上 92% 的準確率看起來相當不錯。現在我們已經對 DistilBERT 進行了微調，讓我們將模型推送到 Hub，以便我們以後可以重用它：

```
distilbert_trainer.push_to_hub( 訓練完成！ )
```

現在我們的模型安全地存儲在 Hub 上，我們可以立即在管道中使用它來進行性能基準測試：

```
finetuned_ckpt = transformersbook/distilbert-base-uncased-finetuned-clinc pipe = pipeline( text-classification ,model=finetuned_ckpt)
```

然後我們可以將這個管道傳遞給我們的PerformanceBenchmark類來計算與這個模型相關的指標：

```
optim_type = DistilBERT pb =
PerformanceBenchmark(pipe, clinc[ test ], optim_type=optim_type)
perf_metrics.update(pb.run_benchmark())
```

模型大小 (MB) - 255.89 平均延遲  
(ms) - 27.53 +/- 0.60 測試集準確度 - 0.858

為了將這些結果與我們的基線進行比較，讓我們創建一個精度與延遲的散點圖，每個點的半徑對應於磁盤上模型的大小。以下函數執行我們需要的操作，並將當前優化類型標記為虛線圓圈，以幫助與之前的結果進行比較：

將熊貓導入為pd

```
def plot_metrics (perf_metrics , current_optim_type) :
    df = pd.DataFrame.from_dict(perf_metrics, orient= index )

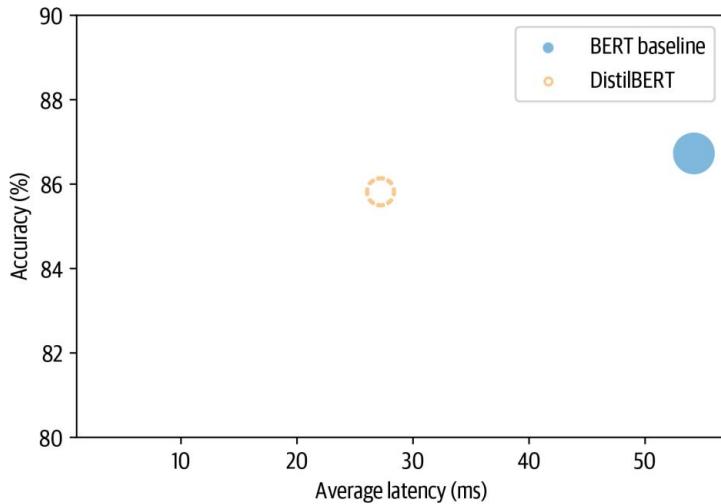
    對於df.index中的idx :
        df_opt = df.loc[idx]
        # 在當前優化類型周圍添加虛線圓圈if idx == current_optim_type:

            plt.scatter(df_opt[ time_avg_ms ],df_opt[ 準確度 ] * 100,
                        alpha=0.5, s=df_opt[ size_mb ],label=idx, marker= $ \u25CC$ )
        否則：
            plt.scatter(df_opt[ time_avg_ms ],df_opt[ accuracy ] * 100,
                        s=df_opt[ size_mb ],label=idx, alpha=0.5)

    legend = plt.legend(bbox_to_anchor=(1,1))用於
    legend.legendHandles中的句柄： handle.set_sizes([20])

    plt.ylim(80,90)
    # 使用最慢的模型定義x軸範圍xlim = int(perf_metrics[ BERT baseline ])
    [ time_avg_ms ] + 3) plt.xlim(1, xlim) plt.ylabel( Accuracy (%) ) plt.xlabel( 平均
    延遲 (ms) ) plt.show()

plot_metrics (perf_metrics , optim_type)
```



從圖中我們可以看出，通過使用較小的模型，我們成功地顯著降低了平均延遲。所有這一切都是代價是準確率降低 1% 多一點！讓我們看看我們是否可以通過包括教師的蒸餾損失並為  $\alpha$  和  $T$  找到好的值來縮小最後一個差距。

使用 Optuna 尋找好的超參數為了找到  $\alpha$  和  $T$  的好的值，我

們可以在 2D 參數空間上進行網格搜索。但更好的選擇是使用 Optuna，<sup>12</sup>這是專為此類任務設計的優化框架。Optuna 根據通過多次試驗優化的目標函數來尋找問題的“香蕉函數”：

$$f(x, y) = (x - 1)^2 + 100(y - x^2)^2$$

這是一個著名的優化框架測試用例。如圖 8-5 所示，該函數的名稱來自於彎曲的等高線，並且在處有一個全局最小值。找到谷是一個簡單的優化問題，但收斂到  $x, y = 1, 1$  全局最小值卻不是。

<sup>12</sup> T. Akiba 等人，“Optuna：下一代超參數優化框架”，（2019）。

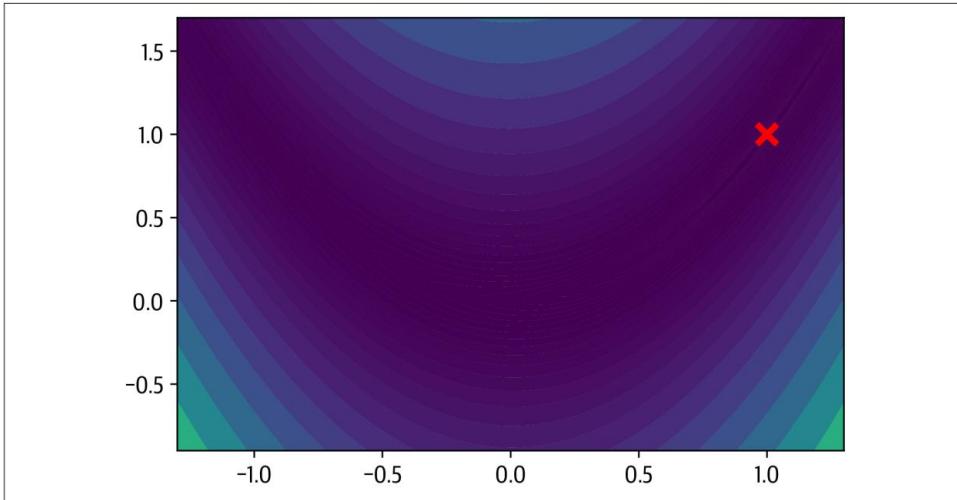


圖 8-5 兩個變量的 Rosenbrock 函數圖

在 Optuna 中，我們可以通過定義一個返回  $f_x, f_y$  值的 `objective()` 函數來找到  $f_x, f_y$  的最小值：

```
def objective(trial): x =
    trial.suggest_float('x', -2, 2) y =
    trial.suggest_float('y', -2, 2) return (1 - x) ** 2
    + 100 * (y - x ** 2) ** 2
```

`trial.suggest_float` 對象指定要從中統一採樣的參數範圍；Optuna 還分別為整數和分類參數提供 `suggest_int` 和 `suggest_categorical`。Optuna 收集多個試驗作為一項研究，因此要創建一個試驗，我們只需將 `objective()` 函數傳遞給 `study.optimize()`，如下所示：

進口 `optuna`

```
study = optuna.create_study()
study.optimize(目標, n_trials=1000)
```

一旦研究完成，我們就可以找到最佳參數如下：

```
研究.best_params
{ 'x': 1.003024865971437, 'y': 1.00315167589307}
```

我們看到，經過一千次試驗，Optuna 已經設法找到合理接近全局最小值的  $x$  和  $y$  值。為了在 Transformers 中使用 Optuna，我們使用類似的邏輯，首先定義我們希望優化<sup>10</sup>的超參數空間。除了  $\alpha$  和  $T$  之外，我們還將包括訓練時期的數量，如下所示：

```
def hp_space(trial): return
    { num_train_epochs : trial.suggest_int( num_train_epochs , 5, 10), alpha :
        trial.suggest_float( alpha , 0, 1), temperature : trial.suggest_int( “溫度” , 2, 20)}
```

使用Trainer運行超參數搜索非常簡單 ;我們只需要指定要運行的試驗次數和優化方向 。因為我們想要最好的精度 ,所以我們在訓練器的hyperparameter\_search()方法中指定方向= 最大化 ,並傳遞超參數搜索空間 ,如下所示 :

```
best_run = distilbert_trainer.hyperparameter_search(
    n_trials=20, direction= 最大化 , hp_space=hp_space)
```

hyperparameter\_search ()方法返回一個BestRun對象 ,其中包含最大化的目標值 (默認情況下 ,所有指標的總和)及其用於該運行的超參數 :

打印 (最佳運行)

```
BestRun(run_id= 1 , objective=0.927741935483871,
hyperparameters={ num_train_epochs :10, alpha :0.12468168730193585, temperature :
7})
```

這個α值告訴我們大部分訓練信號來自知識蒸餾項 。讓我們用這些值更新我們的訓練參數並運行最終訓練 :

對於best\_run.hyperparameters.items()中的k,v :

```
setattr(student_training_args, k, v)
```

```
# 定義一個新的存儲庫來存儲我們的蒸餾模型distilled_ckpt = distilbert-base-
uncased-distilled-clinc student_training_args.output_dir = distilled_ckpt
```

```
# 創建一個具有最佳參數的新訓練器distil_trainer =
DistillationTrainer(model_init=student_init, teacher_model=teacher_model,
args=student_training_args, train_dataset=clinc_enc[ train ],
eval_dataset=clinc_enc[ validation ], compute_metrics=compute_metrics,
tokenizer=student_tokenizer)

distil_trainer.train();
```

Epoch	Training Loss	Validation 損失精度	
1	0.9031	0.574540	0.736452
2	0.4481	0.285621	0.874839
3	0.2528	0.179766	0.918710
4	0.1760	0.139828	0.929355
5	0.1416	0.121053	0.934839
6	0.1243	0.111640	0.934839
7	0.1133	0.106174	0.937742
8	0.1075	0.103526	0.938710
9	0.1039	0.101432	0.938065
10	0.1018	0.100493	0.939355

值得注意的是，我們已經能夠訓練學生以匹配教師的準確性，儘管它的參數數量幾乎減少了一半！讓我們將模型推送到 Hub 以供將來使用：

```
distil_trainer.push_to_hub( 訓練完成 )
```

## 對我們的蒸餾模型進行基準測試

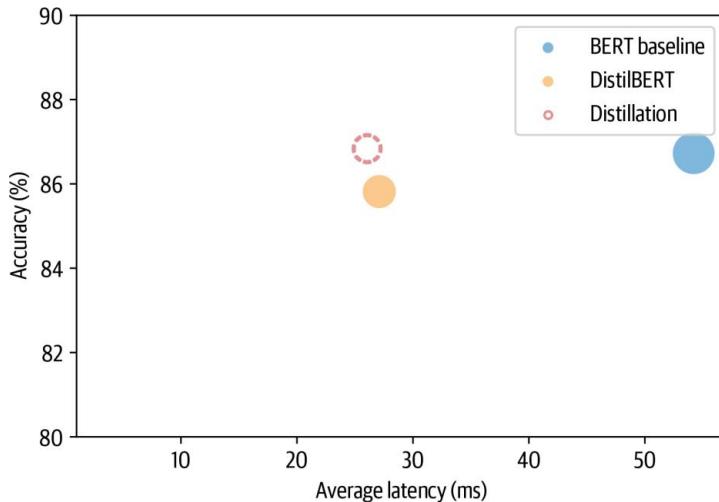
現在我們有了一個準確的學生，讓我們創建一個管道並重做我們的基準測試，看看我們在測試集上的表現如何：

```
distilled_ckpt = transformersbook/distilbert-base-uncased-distilled-clinc pipe = pipeline( 文本分類，model=distilled_ckpt) optim_type = 蒸餾 pb = PerformanceBenchmark(pipe, clinc[ test ] , optim_type=optim_type) perf_metrics.update(pb.run_benchmark())
```

模型大小 (MB) - 255.89 平均延遲  
(ms) - 25.96 +/- 1.63 測試集準確度 - 0.868

為了將這些結果放在上下文中，我們還使用plot\_metrics()函數將它們可視化：

```
plot_metrics (perf_metrics, optim_type)
```



正如預期的那樣，與 DistilBERT 基準相比，模型大小和延遲基本保持不變，但準確性有所提高，甚至超過了教師的表現！解釋這個令人驚訝的結果的一種方法是，教師可能沒有像學生那樣系統地進行微調。這很好，但我們實際上可以使用一種稱為量化的技術進一步壓縮我們的蒸餾模型。這是下一節的主題。

## 通過量化使模型更快

我們現在已經看到，通過知識蒸餾，我們可以通過將信息從教師轉移到較小的學生來減少運行推理的計算和內存成本。量化採用不同的方法；它並沒有減少計算次數，而是通過使用低精度數據類型（如 8 位整數（INT8）而不是通常的 32 位浮點數

（FP32））來表示權重和激活，從而提高了計算效率。減少位數意味著生成的模型需要更少的內存存儲，並且可以使用整數運算更快地執行矩陣乘法等操作。值得注意的是，這些性能提升可以在精度幾乎沒有損失的情況下實現！

浮點數和定點數入門今天的大多數轉換器都使用浮點數（通常是 FP32 或 FP16 和 FP32 的混合）進行預訓練和微調，因為它們提供了適應非常不同的範圍所需的精度權重、激活和梯度。像 FP32 這樣的浮點數表示一個 32 位的序列，這些位按符號、指數和尾數分組。符號決定數字是正數還是負數，而有效數位對應於有效數字的數量，使用某些固定基數中的指數進行縮放（通常二進制為 2 或十進制為 10）。

例如，數字 137.035 可以通過以下算法表示為十進制浮點數：

$$137.035 = (-1) \cdot 0.37035 \times 10^2 \times 1$$

其中 1.37035 是尾數，2 是以 10 為底的指數。通過指數，我們可以表示範圍廣泛的實數，小數點或二進制小數點可以放在相對於有效數字的任何位置（因此得名“浮動”-觀點）。

然而，一旦模型經過訓練，我們只需要前向傳播就可以運行推理，因此我們可以降低數據類型的精度，而不會對準確性產生太大影響。對於神經網絡，通常對低精度數據類型使用定點格式，其中實數表示為 B 位整數，對於相同類型的所有變量，這些整數按公因數縮放。例如，137.035 可以表示為按 1/1,000 縮放的整數 137,035。我們可以通過調整比例因子來控制定點數的範圍和精度。

量化背後的基本思想是，我們可以“離散化”每個張量中的浮點值  $f$ ，方法是映射它們的範圍  $[f_{\min}, f_{\max}]$  轉換成較小的  $[q_{\min}, q_{\max}]$  定點數  $q$ ，並線性分佈其間的所有值。

在數學上，此映射由以下等式描述：

$$f = \left( \frac{F_{\max} - f}{q_{\max} - q_{\min}} \right) (q - z) + s(q - z)$$

其中比例因子  $s$  為正浮點數，常量  $z$  與  $q$  同類型，稱為零點，因為它對應浮點值  $f=0$  的量化值。注意映射需要是彷射的，這樣我們

當我們對定點數進行反量化時得到浮點數。13圖 8-6 顯示了轉換的圖示。

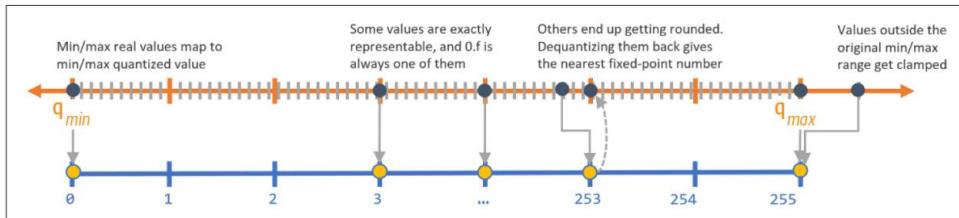
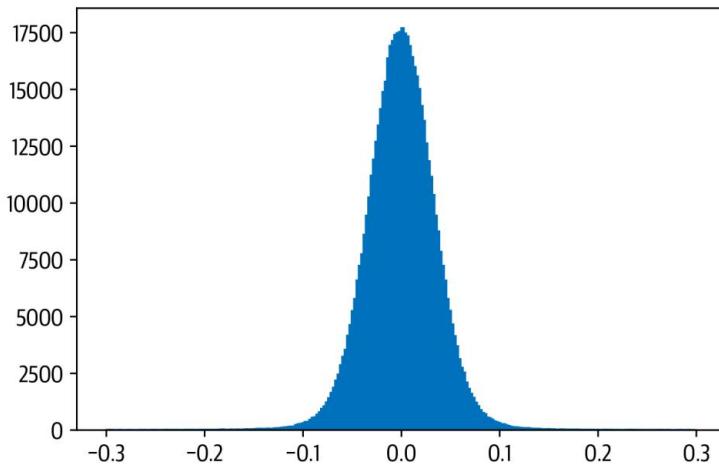


圖 8-6 將浮點數量化為無符號 8 位整數（由 Manas Sahni 提供）

現在，transformer（以及更普遍的深度神經網絡）成為量化的主要候選者的主要原因之一是權重和激活傾向於在相對較小的範圍內取值。這意味著我們不必將可能的 FP32 數字的整個範圍壓縮到 INT8 表示的  $2 = 256$  數字中。為了看到這一點，讓我們從我們的蒸餾模型中挑選出一個注意力權重矩陣並繪製值的頻率分佈：

將 `matplotlib.pyplot` 導入為 plt

```
state_dict = pipe.model.state_dict()
weights = state_dict[ distilbert.transformer.layer.0.attention.out_lin.weight ]
plt.hist(weights.flatten().numpy(), bins=250, range=(-0.3,0.3), edgecolor='C0')
plt.show()
```



13 仿射映射只是您在  $a$  的線性層中熟悉的  $y = Ax + b$  映射的奇特名稱  
神經網絡。

正如我們所見，權重值分佈在小範圍  $[-0.1, 0.1]$  大約為零。現在，假設我們想將這個張量量化為一個帶符號的 8 位整數。

在那種情況下，我們整數的可能值範圍是  $[q_{\max}, q_{\min}] = [-128, 127]$ 。

零點與 FP32 的零點重合，比例因子按上式計算：

```
zero_point = 0
scale = (weights.max() - weights.min()) / (127 - (-128))
```

要獲得量化張量，我們只需反轉映射  $q = f/S + Z$ ，將它們四捨五入到最接近的整數，並使用Tensor.char()將結果表示為 torch.int8數據類型功能：

```
(weights / scale + zero_point).clamp(-128, 127).round().char()
```

```
張量([[-5, -8, 0, ..., -6, -4, 8], [8, 3, 1, ..., -4, 7, 0], [-9, -6, 5, ...,
1, 5, -3],
```

```
..., [6, 0, 6, -1]0[12, 2, 12, ..., 12, -7, -13], [-13, -1,
-10, ..., 8, 2, -2]], dtype=火炬.int8)
```

太好了，我們剛剛量化了第一個張量！在 PyTorch 中，我們可以通過使用 quantize\_per\_tensor() 函數和量化數據類型 torch.qint來簡化轉換，該數據類型針對整數算術運算進行了優化：

從火炬導入quantize\_per\_tensor

```
dtype = torch.qint8
quantized_weights = quantize_per_tensor(weights, scale, zero_point, dtype)量化權重.int_repr()
```

```
張量([[-5, -8, 0, ..., -6, -4, 8], [8, 3, 1, ..., -4, 7, 0], [-9, -6, 5, ...,
1, 5, -3],
```

```
..., [6, 0, 6, -1]0[12, 2, 12, ..., 12, -7, -13], [-13, -1,
-10, ..., 8, 2, -2]], dtype=火炬.int8)
```

圖 8-7 中的圖非常清楚地顯示了僅精確映射某些權重值並對其餘權重值進行四捨五入所引起的離散化。

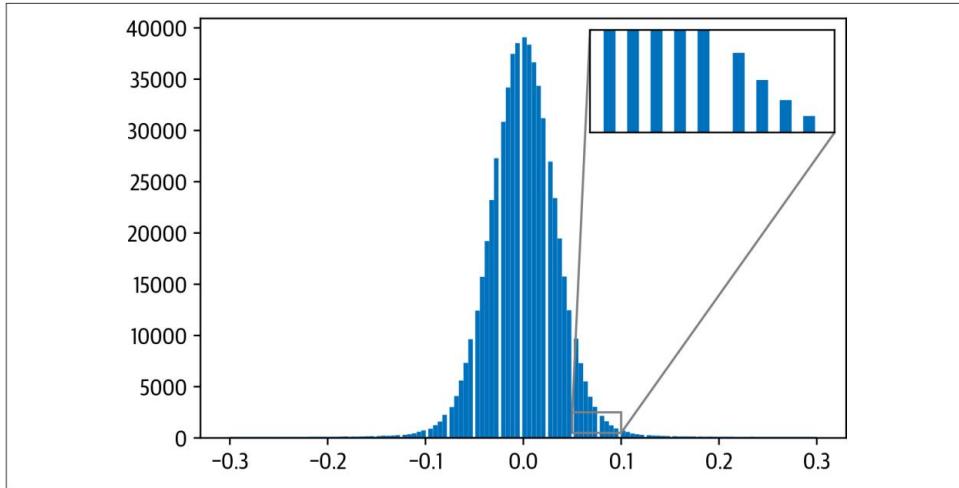


圖 8-7 量化對變壓器權重的影響

為了完善我們的小分析，讓我們比較一下計算兩個權重張量與 FP32 和 INT8 值的乘積需要多長時間。對於 FP32 張量，我們可以使用 PyTorch 的漂亮的@運算符將它們相乘：

```
%%%timeit
權重@權重
每個循環 393 µs ± 3.84 µs (7 次運行的平均值 ± 標準偏差，每次 1000 次循環)
```

對於量化張量，我們需要QFunctional包裝類，以便我們可以使用特殊的torch.qint8數據類型執行操作：

從[torch.nn.quantized](#)導入QFunctional

```
q_fn = QFunctional()
```

這個類支持各種基本運算，比如加法，在我們的例子中，我們可以按如下方式計算量化張量的乘法：

```
%%%timeit
q_fn.mul (量化權重，量化權重)
每個循環 23.3 µs ± 298 ns (7 次運行的平均值 ± 標準差，每次 10000 次循環)
```

與我們的 FP32 計算相比，使用 INT8 張量快了近 100 倍！通過使用專用後端來高效運行量化運算符可以獲得更大的收益。在撰寫本書時，PyTorch 支持：

- 支持 AVX2 或更高版本的 x86 CPU ·
- ARM CPU（通常存在於移動/嵌入式設備中）

由於 INT8 數字的位數比 FP32 數字少四倍，因此量化也將內存存儲要求降低了四分之一。在我們的簡單示例中，我們可以通過使用 `Tensor.storage()` 函數和 Python `sys` 模塊中的 `getsizeof()` 函數比較權重張量及其量化表親的底層存儲大小來驗證這一點：

#### [導入系統](#)

```
sys.getsizeof(weights.storage()) / sys.getsizeof(quantized_weights.storage())
3.999633833760527
```

對於全尺寸變換器，實際壓縮率取決於哪些層被量化（正如我們將在下一節中看到的那樣，通常只有線性層才會被量化）。

那麼量化有什麼問題呢？更改我們模型中所有計算的精度會在模型計算圖中的每個點引入小干擾，這會復合併影響模型的性能。量化模型的方法有多種，各有利弊。對於深度神經網絡，通常有三種主要的量化方法：

#### 動態量化當使用動態量

化時，訓練期間沒有任何變化，並且僅在推理期間執行自適應。與我們將討論的所有量化方法一樣，模型的權重在推理時間之前被轉換為 INT8。除了權重之外，模型的激活也被量化。這種方法是動態的，因為量化是動態發生的。

這意味著可以使用高度優化的 INT8 函數計算所有矩陣乘法。在這裡討論的所有量化方法中，動態量化是最簡單的一種。然而，通過動態量化，激活以浮點格式寫入和讀取到內存中。整數和浮點數之間的這種轉換可能是性能瓶頸。

#### 靜態量化不是動態計

算激活的量化，我們可以通過預先計算量化方案來避免轉換為浮點數。靜態量化通過在推理時間之前觀察代表性數據樣本的激活模式來實現這一點。計算並保存理想的量化方案。這使我們能夠跳過 INT8 和 FP32 值之間的轉換並加快計算速度。但是，它需要訪問良好的數據樣本並在管道中引入一個額外的步驟，因為我們現在需要在執行推理之前訓練和確定量化方案。還有一個方面靜態量化沒有解決：訓練和推理期間的精度之間的差異，這會導致模型指標（例如準確性）的性能下降。

## 量化感知訓練

通過對 FP32 值進行“假”量化，可以在訓練期間有效地模擬量化效果。FP32 值不是在訓練期間使用 INT8 值，而是四捨五入以模仿量化效果。這是在前向和後向傳遞過程中完成的，並且在靜態和動態量化的模型指標方面提高了性能。

使用 Transformer 進行推理的主要瓶頸是與這些模型中大量權重相關的計算和內存帶寬。出於這個原因，動態量化是目前 NLP 中基於轉換器的模型的最佳方法。在較小的計算機視覺模型中，限制因素是激活的內存帶寬，這就是為什麼通常使用靜態量化（或在性能下降太顯著的情況下進行量化感知訓練）的原因。

在 PyTorch 中實現動態量化非常簡單，只需一行代碼即可完成：

從`torch.quantization`導入`quantize_dynamic`

```
model_ckpt = transformersbook/distilbert-base-uncased-distilled-clinc tokenizer =
AutoTokenizer.from_pretrained(model_ckpt)模型= (AutoModelForSequenceClassification
.from_pretrained(model_ckpt).to( cpu ))  
  
model_quantized = quantize_dynamic (模型, {nn.Linear}, dtype=torch.qint8)
```

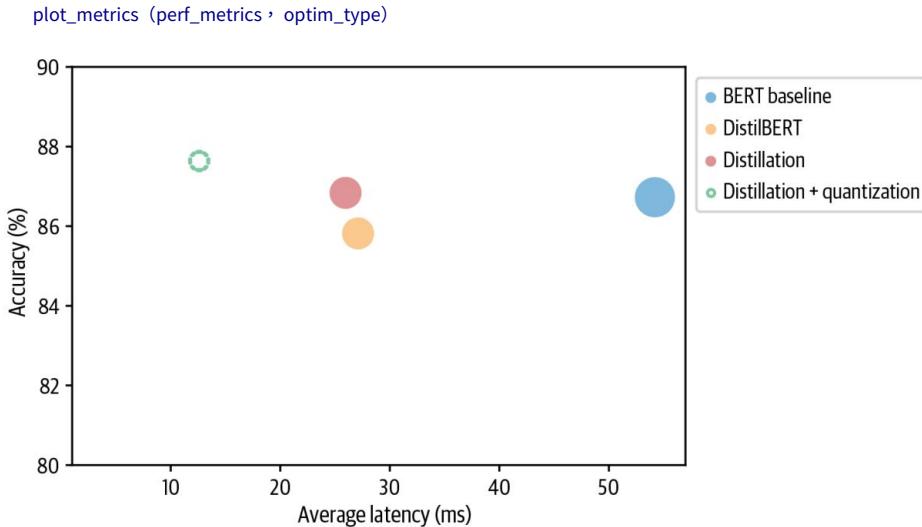
在這裡，我們將全精度模型傳遞給`quantize_dynamic()`並在該模型中指定我們要量化的一組 PyTorch 層類。`dtype`參數指定目標精度，可以是`fp16`或`qint8`。一個好的做法是選擇您可以容忍的評估指標的最低精度。在本章中，我們將使用`INT8`，我們很快就會發現它對我們模型的準確性影響很小。

## 對我們的量化模型進行基準測試

現在我們的模型已經量化，讓我們通過基準測試並可視化結果：

```
pipe = pipeline( text-classification , model=model_quantized, tokenizer=tokenizer)
          optim_type = 蒸餾+量化 pb = PerformanceBenchmark(pipe,
clinc[ test ], optim_type=optim_type) perf_metrics.update(pb.run_benchmark())
```

模型大小 (MB) - 132.40 平均延遲  
(ms) - 12.54 +/- 0.73 測試集準確度 - 0.876



很好，量化模型的大小幾乎是我們蒸餾模型的一半，甚至還獲得了輕微的精度提升！讓我們看看我們是否可以使用稱為 ONNX 運行時的強大框架將我們的優化推向極限。

## 使用 ONNX 和 ONNX 運行時優化推理

ONNX 是一種開放標準，它定義了一組通用運算符和一種通用文件格式，用於表示各種框架（包括 PyTorch 和 TensorFlow）中的深度學習模型。<sup>14</sup> 當模型導出為 ONNX 格式時，將使用這些運算符構建一個計算圖（通常稱為中間表示）來表示通過神經網絡的數據流。圖 8-8 顯示了 BERT-base 的此類圖的示例，其中每個節點接收一些輸入，應用 Add 或 Squeeze 等操作，然後將輸出饋送到下一組節點。

---

<sup>14</sup>有一個名為 ONNX-ML 的單獨標準，專為傳統機器學習模型（如隨機森林）和框架（如 Scikit-learn）而設計。

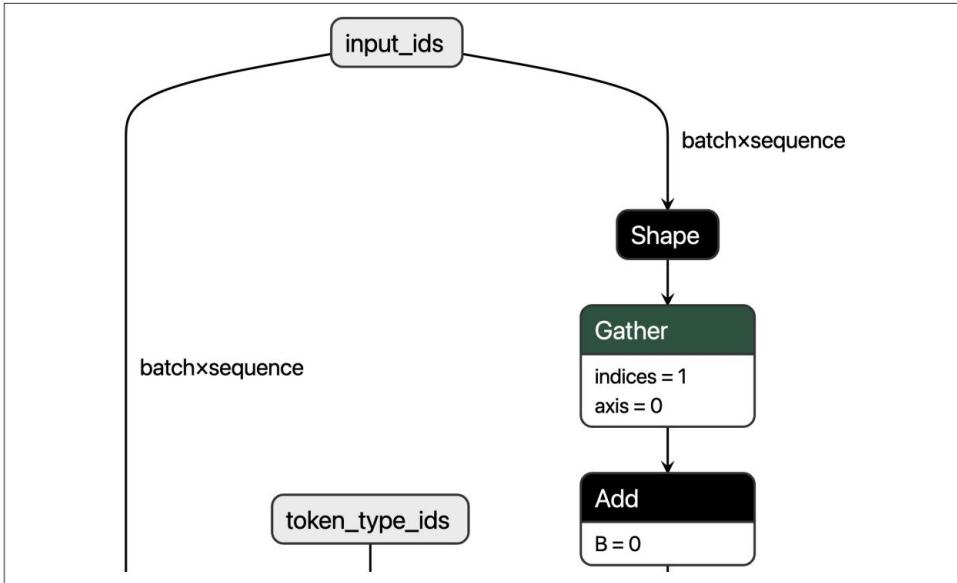


圖 8-8。BERT-base 的 ONNX 圖的一部分，在 Netron 中可視化

通過公開具有標準化運算符和數據類型的圖形，ONNX 可以輕鬆地在框架之間切換。例如，在 PyTorch 中訓練的模型可以導出為 ONNX 格式，然後導入到 TensorFlow 中（反之亦然）。

ONNX 真正閃耀的地方在於它與 [ONNX Runtime](#) 等專用加速器結合使用時，或簡稱 ORT。<sup>15</sup> ORT 提供了通過運算符融合和常量折疊等技術優化 ONNX 圖的工具，<sup>16</sup>並定義了執行提供程序的接口，使您可以在不同類型的硬件上運行模型。這是一個強大的抽象。[圖 8-9](#)顯示了 ONNX 和 ORT 生態系統的高層架構。

<sup>15</sup>其他流行的加速器包括 [NVIDIA 的 TensorRT](#) 和 [Apache TVM](#)。

<sup>16</sup>融合操作涉及將一個運算符（通常是激活函數）合併到另一個運算符中，以便它們可以一起執行。例如，假設我們要將激活函數  $f$  應用於矩陣乘積  $A \times B$ 。

通常乘積的結果需要在計算激活之前寫回 GPU 內存。算子融合允許  $as$  一步計算  $f A \times B$ 。常量折疊是指在編譯時而非運行時對常量表達式求值的過程。

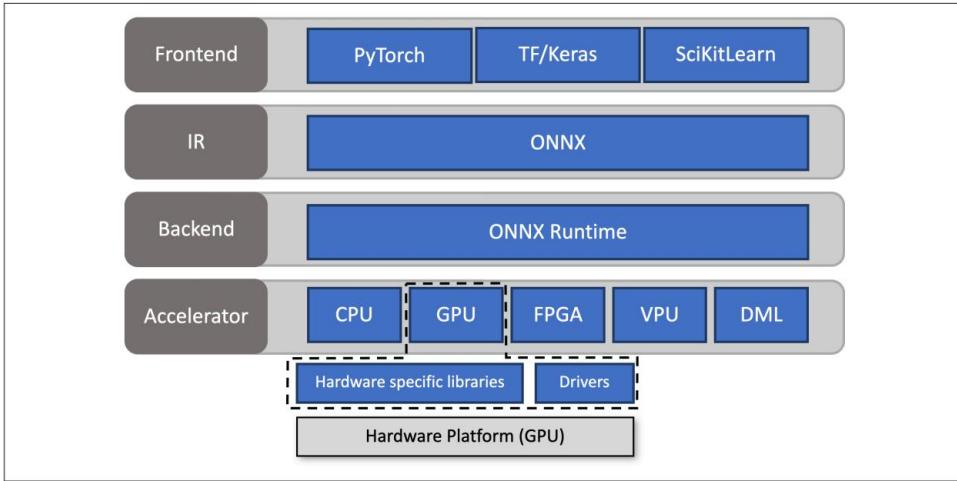


圖 8-9。ONNX 和 ONNX Runtime 生態系統的架構（由 ONNX Runtime 團隊提供）

要查看 ORT 的運行情況，我們需要做的第一件事是將我們的蒸餾模型轉換為 ONNX 格式。

Transformers 庫有一個名為convert\_graph\_to\_onnx.convert()內置函數，它通過以下步驟簡化了這個過程：

1. 將模型初始化為Pipeline。
2. 通過管道運行佔位符輸入，以便 ONNX 可以記錄計算圖。
3. 定義動態軸來處理動態序列長度。
4. 保存帶有網絡參數的圖形。

要使用這個功能，我們首先需要設置一些OpenMP的環境變量  
ONNX：

從psutil導入  
cpu\_count導入操作系統

```
os.environ[ 'OMP_NUM_THREADS' ] = f'{cpu_count()}'  
os.environ[ 'OMP_WAIT_POLICY' ] = 'ACTIVE'
```

OpenMP 是為開發高度並行化的應用程序而設計的 API。OMP\_NUM\_THREADS環境變量設置用於ONNX 運行時並行計算的線程數，而OMP\_WAIT\_POLICY=ACTIVE指定等待線程應該處於活動狀態（即，使用 CPU 處理器週期）。

接下來，讓我們將我們的蒸餾模型轉換為 ONNX 格式。這裡我們需要指定參數pipeline\_name= text-classification，因為convert()包裝了

在轉換期間在 Transformers pipeline()函數中建模。除了 model\_ckpt 之外，我們還通過分詞器來初始化管道：

從transformers.convert\_graph\_to\_onnx導入轉換

```
model_ckpt = transformersbook/distilbert-base-uncased-distilled-clinc onnx_model_path =
Path( onnx/model.onnx ) convert(framework= pt ,model=model_ckpt,tokenizer=tokenizer,
輸出=onnx_model_path, opset=12 , pipeline_name= 文本分類 )
```

ONNX 使用運算符集將不可變的運算符規範組合在一起，因此opset=12對應於特定版本的ONNX庫。

現在我們已經保存了模型，我們需要創建一個InferenceSession實例來為模型提供輸入：

```
從onnxruntime import (GraphOptimizationLevel, InferenceSession,
會話選項)

def create_model_for_provider(model_path, provider= CPUExecutionProvider ):
    options = SessionOptions()
    options.intra_op_num_threads = 1
    options.graph_optimization_level = GraphOptimizationLevel.ORT_ENABLE_ALL session =
    InferenceSession(str(model_path), options, providers=[provider]) session.disable_fallback()返回會話
```

```
onnx_model = create_model_for_provider(onnx_model_path)
```

現在，當我們調用onnx\_model.run() 時，我們可以從 ONNX 模型中獲取類 logits。讓我們用測試集中的一個例子來測試一下。由於convert()的輸出告訴我們 ONNX 只需要input\_ids和attention\_mask作為輸入，我們需要從示例中刪除標籤列：

```
inputs = clinc_enc[ test ][1:] del
inputs[ labels ] logits_onnx =
onnx_model.run(None, inputs)[0] logits_onnx.shape
```

```
(1, 151)
```

一旦我們有了 logits，我們就可以通過使用 argmax 輕鬆獲得預測標籤：

```
np.argmax (logits_onnx)
```

```
61
```

這確實與基本事實標籤一致：

```
clinc_enc[ “測試” ][0][ “標籤” ]
```

```
61
```

ONNX 模型與文本分類管道不兼容，因此我們將創建自己的類來模仿核心行為：

```
從scipy.special導入softmax

類OnnxPipeline: def
    __init__(self, model, tokenizer): self.model =
        model self.tokenizer = tokenizer

    def __call__(self, query):
        model_inputs = self.tokenizer(query, return_tensors= pt ) inputs_onnx =
        {k: v.cpu().detach().numpy() for k, v in model_inputs.items() }

        logits = self.model.run(None, inputs_onnx)[0][0, :] probs =
        softmax(logits) pred_idx = np.argmax(probs).item() 返回[ [ label :
        intents.int2str(pred_idx), 得分 : probs[pred_idx]]]
```

然後我們可以在我們的簡單查詢上測試它，看看我們是否恢復了car\_rental意圖：

```
pipe = OnnxPipeline(onnx_model, tokenizer)管道(查詢)
```

```
[{ label : car_rental , score :0.7848334}]
```

太好了，我們的管道按預期工作。下一步是為 ONNX 模型創建性能基準。在這裡，我們可以在我們所做的工作的基礎上。

PerformanceBenchmark類，只需覆蓋compute\_size()方法並保持compute\_accuracy()和time\_pipeline()方法不變。我們需要覆蓋compute\_size()方法的原因是我們不能依賴state\_dict和torch.save()來測量模型的大小，因為onnx\_model在技術上是一個ONNX InferenceSession對象，它不能訪問屬性PyTorch的nn.Module。在任何情況下，結果邏輯都很簡單，可以按如下方式實現：

```
類OnnxPerformanceBenchmark(PerformanceBenchmark): def
    __init__(self, *args, model_path, **kwargs): super().__init__(*args,
    **kwargs) self.model_path = model_path

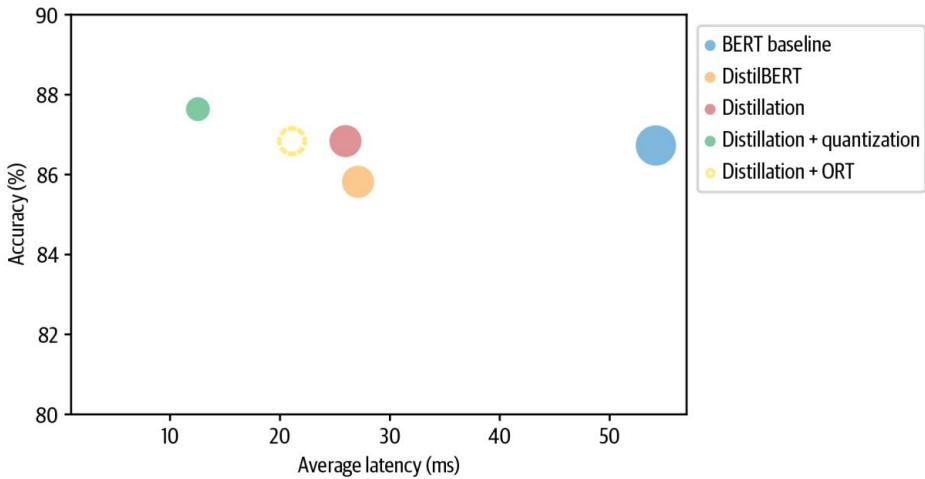
    def compute_size (自身) :
        size_mb = Path(self.model_path).stat().st_size / (1024 * 1024) print(f 模型大小 (MB)
        -{size_mb:.2f} ) return { size_mb :size_mb}
```

使用我們的新基準，讓我們看看我們的蒸餾模型在轉換為ONNX格式時的表現：

```
optim_type = Distillation + ORT pb =
OnnxPerformanceBenchmark(pipe, clin[ test ], optim_type, model_path= onnx/
model.onnx )
perf_metrics.update(pb.run_benchmark())
```

模型大小 (MB) - 255.88 平均延遲  
(ms) - 21.02 +/- 0.55 測試集準確度 - 0.868

plot\_metrics (perf\_metrics, optim\_type)



值得注意的是，轉換為 ONNX 格式並使用 ONNX Runtime 使我們的蒸餾模型（即圖中的“蒸餾”圓圈）延遲增加了！

讓我們看看我們是否可以通過將量化添加到混合。

與 PyTorch 類似，ORT 提供了三種量化模型的方法：動態、靜態和量化感知訓練。正如我們對 PyTorch 所做的那樣，我們將對我們的蒸餾模型應用動態量化。在 ORT 中，量化是通過 `quantize_dynamic()` 函數應用的，它需要一個到要量化的 ONNX 模型的路徑，一個要將量化模型保存到的目標路徑，以及要將權重降低到的數據類型：

```
從onnxruntime.quantization import quantize_dynamic, QuantType
```

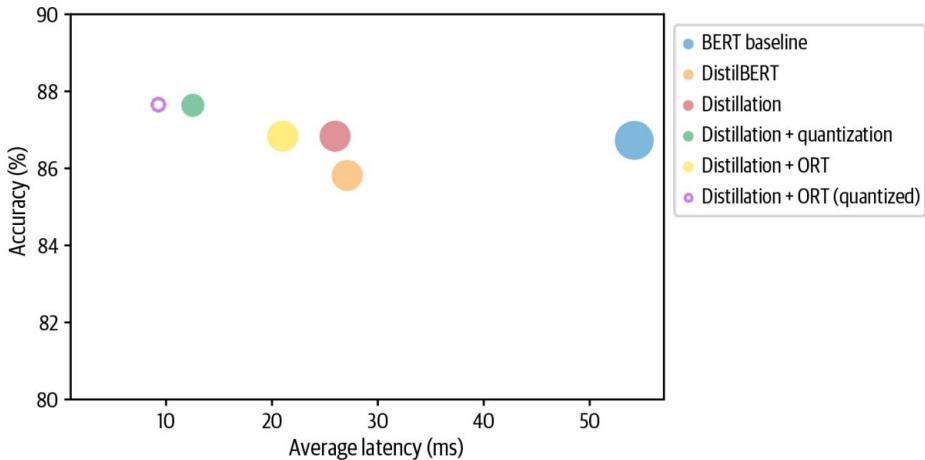
```
model_input = onnx/model.onnx
model_output = onnx/model.quant.onnx
quantize_dynamic(model_input, model_output, weight_type=QuantType.QInt8)
```

現在模型已經量化，讓我們通過我們的基準運行它：

```
onnx_quantized_model = create_model_for_provider(model_output) pipe =
OnnxPipeline(onnx_quantized_model, tokenizer) optim_type = 蒸餾 + ORT (量化)
pb = OnnxPerformanceBenchmark(pipe, clinic[ test ], optim_type,
model_path=model_output) perf_metrics.update(pb.run_benchmark ())
```

模型大小 (MB) - 64.20 平均延遲  
(ms) - 9.24 +/- 0.29 測試集準確度 - 0.877

plot\_metrics (perf\_metrics, optim\_type)



與從 PyTorch 量化（蒸餾 + 量化 blob）獲得的模型相比，ORT 量化將模型大小和延遲減少了約 30%。原因之一是 PyTorch 只優化了 nn.Linear 模塊，而 ONNX 也量化了嵌入層。從圖中我們還可以看到，與我們的 BERT 基線相比，將 ORT 量化應用於我們的蒸餾模型提供了近三倍的增益！

我們對加速變換器推理的技術的分析到此結束。我們已經看到，量化等方法通過降低表示的精度來減小模型大小。減小尺寸的另一種策略是完全去除一些重量。這種技術稱為權重修剪，它是下一節的重點。

## 通過權重修剪使模型更稀疏

到目前為止，我們已經看到知識蒸餾和權重量化在生成更快的推理模型方面非常有效，但在某些情況下，您可能還對模型的內存佔用有嚴格的限制。例如，如果我們的產品經理突然決定我們的文本助手需要部署在移動設備上，那麼我們就需要我們的意圖分類器佔用盡可能少的存儲空間。為了完善我們對壓縮方法的調查，讓我們來看看如何通過識別和刪除網絡中最不重要的權重來減少模型中的參數數量。

深度神經網絡中的稀疏性如圖 8-10 所

示，修剪背後的主要思想是在訓練期間逐漸移除權重連接（以及潛在的神經元），從而使模型逐漸變得稀疏。生成的修剪模型具有較少數量的非零參數，然後可以將其存儲在緊湊的稀疏矩陣格式中。修剪也可以與量化相結合以獲得進一步的壓縮。

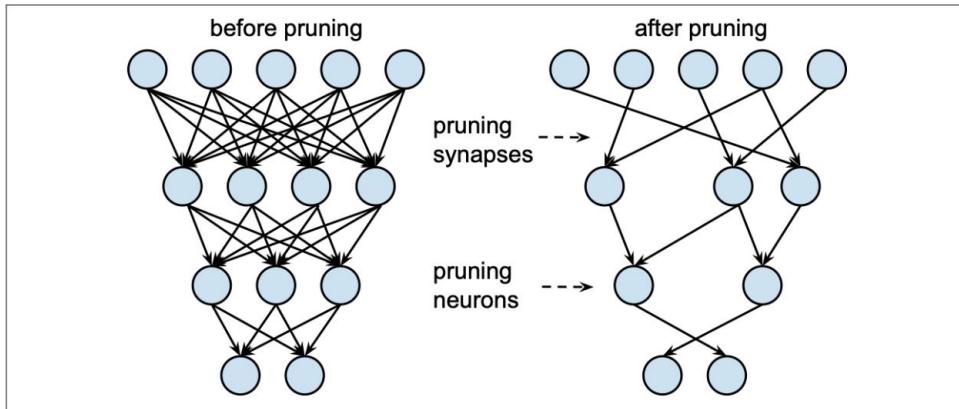


圖 8-10 剪枝前後的權重和神經元（宋涵提供）

## 權重修剪方法

從數學上講，大多數權重修剪方法的工作方式是計算重要性分數矩陣，然後按重要性選擇前  $k\%$  的權重：

$$\text{Topk}_{ij} = \begin{cases} 1 & \text{如果 } S_{ij} \text{ 在前 } k\% \\ 0 & \text{否} \end{cases}$$

實際上， $k$  作為一個新的超參數來控制模型中的稀疏度 即，權重為零的比例。較低的  $k$  值對應於較稀疏的矩陣。然後，根據這些分數，我們可以定義一個掩碼矩陣，在前向傳遞過程中用一些輸入  $x_i$  掩碼權重  $W_{ij}$ ，並有效地創建一個稀疏的激活網絡  $a_i$ ：

$$a_i = \sum_k \text{維克米克斯克}$$

正如半開玩笑的“Optimal Brain Surgeon”論文<sup>17</sup>中所討論的，每種修剪方法的核心是一組需要考慮的問題：

- 應該消除哪些權重？ · 應如何調整剩餘權重以獲得最佳性能？ · 這種網絡修剪如何以計算效率高的方式進行？

這些問題的答案告訴我們分數矩陣是如何計算的，所以讓我們從最早和最流行的修剪方法之一開始：量級修剪。

### Magnitude

pruning顧名思義，magnitude pruning根據權重 =  $W_{ij}$ 的大小計算得分，然後從 = Topk推導出掩碼。

( ) 在文獻中，通常以迭代方式應用幅度剪枝，首先訓練模型以了解哪些連接是重要的，然後剪掉最不重要的權重。<sup>18</sup>然後重新訓練稀疏模型並重複該過程，直到達到所需的稀疏度到達了。

這種方法的一個缺點是它對計算要求很高：在修剪的每一步，我們都需要訓練模型收斂。出於這個原因，通常最好逐漸將初始稀疏度s（通常為零）增加到最終稀疏度

經過一些步驟N：f後的值s

19

$$\text{小號} = \text{小號}_{\text{F}} + s \left( -\frac{\text{F}_i}{\text{F}} \right) \left( 1 - \frac{t - t_0}{N\Delta t} \right)^m \quad \text{堡壘} = \{ 0, 0 + \Delta t, \dots, + N\Delta t \}$$

這裡的想法是每 $\Delta t$ 步更新二元掩碼，以允許掩碼權重在訓練期間重新激活，並從修剪過程引起的任何潛在的精度損失中恢復。如圖<sup>8-11</sup>所示，三次因子意味著權重剪枝的速率在早期階段最高（當冗餘權重的數量很大時），然後逐漸變小。

---

17 B. Hassibi 和 D. Stork，“網絡修剪的二階導數：最佳腦外科醫生”，第五屆神經信息處理系統國際會議論文集（1992年11月）：164-171。<https://papers.nips.cc/paper/1992/hash/303ed4c69846ab36c2904d3ba8573050-Abstract.html>。

18 S. Han 等人，“學習有效神經網絡的權重和連接”，(2015)。

19 M. Zhu 和 S. Gupta，“修剪還是不修剪：探索修剪對模型壓縮的功效”，(2017)。

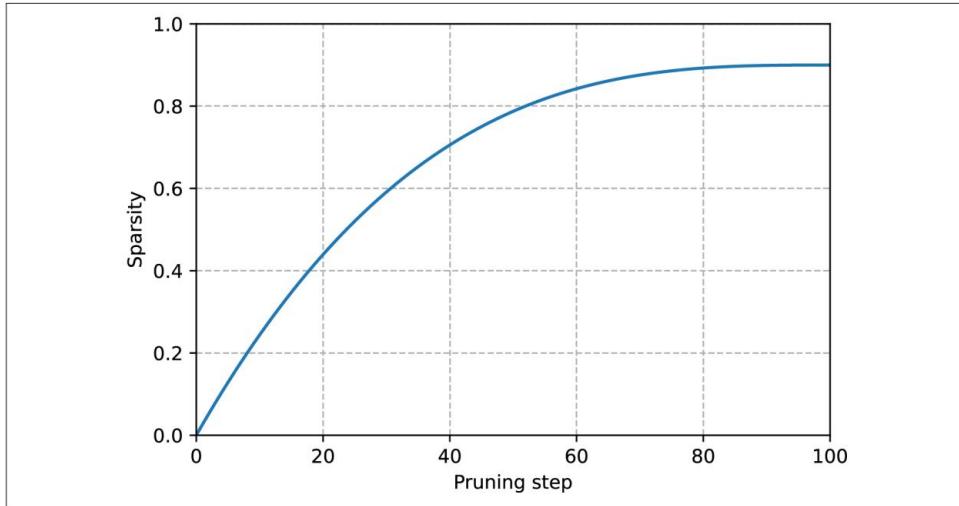


圖 8-11。用於修剪的三次稀疏調度器

幅度剪枝的一個問題是它實際上是為純監督學習而設計的，其中每個權重的重要性都與手頭的任務直接相關。

相比之下，在遷移學習中，權重的重要性主要由預訓練階段決定，因此幅度剪枝可以移除對微調任務很重要的連接。最近，Hugging Face 研究人員提出了一種稱為運動修剪的自適應方法，讓我們來看看。<sup>20</sup>

### 運動修剪運動修剪

背後的基本思想是在微調過程中逐漸去除權重，使模型逐漸變得稀疏。關鍵的新穎之處在於權重和分數都是在微調期間學習的。因此，運動修剪中的分數不是直接從權重中得出（如幅度剪枝），而是任意的，並且像任何其他神經網絡參數一樣通過梯度下降學習。這意味著在向後傳遞中，我們還跟蹤損失  $L$  相對於分數  $S_{ij}$  的梯度。

一旦學習了分數，就可以直接使用  $= \text{Topk}$  生成二進制掩碼。

( )<sup>21</sup>

運動修剪背後的直覺是，從零“移動”最多的權重是最重要的保留。換句話說，正

---

20 V. Sanh、T. Wolf 和 AM Rush，“運動修剪：通過微調實現的自適應稀疏性”，(2020)。

21 還有一個“軟”版本的運動修剪，不是選擇權重的前  $k\%$ ，而是使用一個定義二進制掩碼的全局閾值  $\tau := > \tau$  ( )。

權重在微調期間增加（負權重反之亦然），這相當於說分數隨著權重遠離零而增加。如圖 8-12 所示，此行為不同於幅度修剪，後者選擇離零最遠的權重作為最重要的權重。

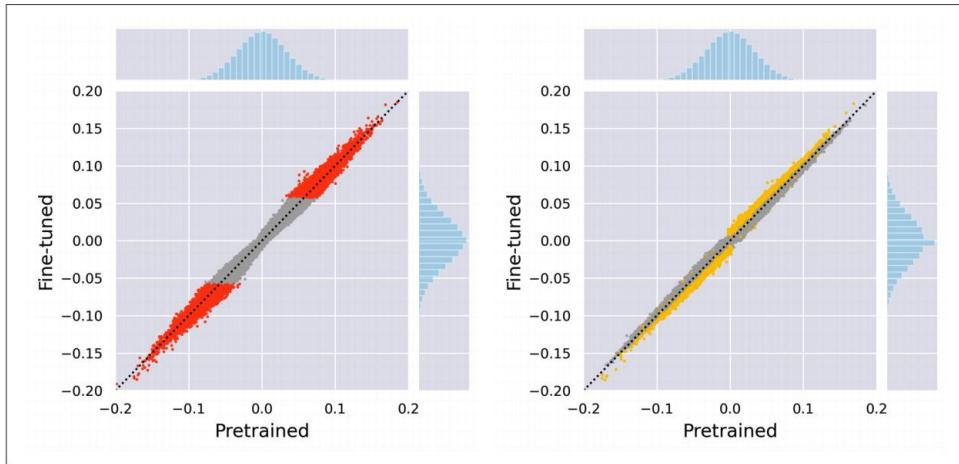


圖 8-12 幅度剪枝（左）和運動剪枝（右）去除權重的比較

兩種剪枝方法之間的這些差異在剩餘權重的分佈中也很明顯。如圖 8-13 所示，幅度剪枝產生兩個權重簇，而移動剪枝產生更平滑的分佈。

在本書寫作之時，變形金剛不支持開箱即用的剪枝方法。幸運的是，有一個漂亮的庫叫做 [神經網絡塊運動修剪](#)，它實現了其中的許多想法，如果內存限制是一個問題，我們建議您檢查一下。

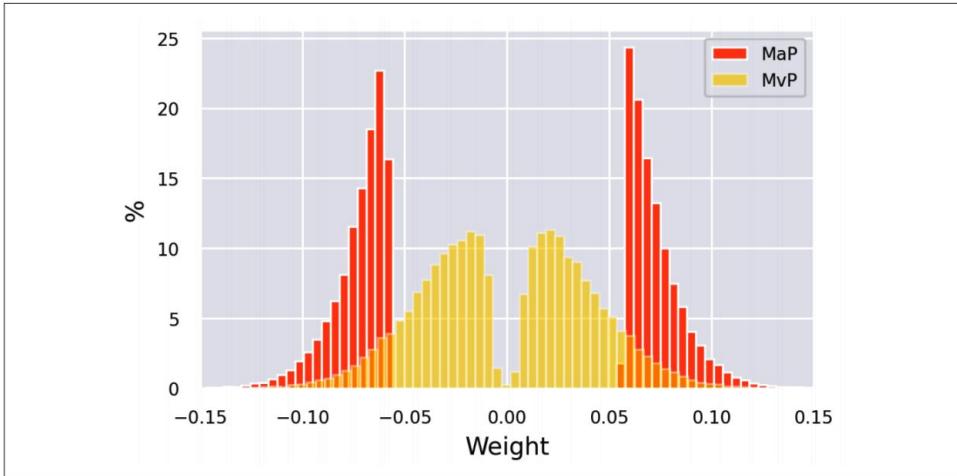


圖 8-13 幅度剪枝 (MaP)和移動剪枝 (MvP)的剩餘權重分佈

## 結論

我們已經看到，優化部署在生產環境中的轉換器涉及兩個維度的壓縮：延遲和內存佔用。從微調模型開始，我們通過 ORT 應用蒸餾、量化和優化，以顯著減少這兩種情況。特別是，我們發現 ORT 中的量化和轉換以最小的努力獲得了最大的收益。

儘管剪枝是減少 transformer 模型存儲大小的有效策略，但當前的硬件並未針對稀疏矩陣運算進行優化，這限制了該技術的實用性。然而，這是一個活躍的研究領域，到本書上架時，其中許多限制可能已經解決。

那麼從這裡到哪裡呢？本章中的所有技術都可以適用於其他任務，例如問答、命名實體識別或語言建模。如果您發現自己難以滿足延遲要求，或者您的模型耗盡了所有計算預算，我們建議您嘗試其中之一。

在下一章中，我們將不再關注性能優化，而是探索每位數據科學家最糟糕的噩夢：處理很少甚至沒有標籤。