遞迴神經網路與變形器

作業三

學號：313831025

姓名：俞博云

一、 題目敘述

本作業旨在實作一套命名實體辨識（Named Entity Recognition, NER）系統，透過自然語言處理技術對輸入文本進行序列標註，標記出具特定語意的詞彙。使用提供的 DNTRI 資料集（train.txt、valid.txt、test.txt）訓練並驗證模型，最終預測測試集中的實體標籤。模型建議以 BERT 為基礎，可進一步整合 LSTM、BiLSTM 或 CRF 層以提升辨識效果。

二、 數據預處理

本作業選用 DNRTI 數據集，資料集包含三個部分用於訓練、驗證及測試階段，每一筆資料以「詞語+標籤」的格式儲存，並以空行區隔句子。首先需將原始文本資料依句子切分並擷取出詞語與對應標籤，接著進行斷詞與標籤編碼，將文字轉換為模型可接受的向量表示。為了配合 BERT 模型的輸入格式，資料需經過 tokenizer 處理，並添加適當的 attention mask 及 token type ids，此外需要對特殊標籤也須進一步轉為數值索引。以下分別進行介紹：

2.1 數據讀取

數據集的資料中，每一筆資料以「詞語+標籤」的格式儲存，並以空行區隔句子，如表 2.1 所示。這些資料均以.txt 純文字檔形式提供，其中文字檔內容為逐行標註的格式，每行表示一個詞與其對應的命名實體標籤，使用空格進行分隔。

表 2.1、數據集資料

詞語	標籤
The	O
admin@338	B-HackOrg
has	O
largely	O
targeted	O
organizations	O

首先對於數據資料集(txt 格式)使用 open 及 readlines() 讀取，並且使用 rstrip('\n')去除每行結尾的換行符號，並將每句話的詞語與對應標籤分別累積到 tokens_list 和 ner_tags_list 中。每一筆句子資料最後會以空格串接的形式儲存，方便後續模型訓練使用。這樣的處理流程可確保資料格式一致，利於批次輸入模型。

```python
def read_ner_txt(path):
    with open(path, 'r') as f:
        text = f.readlines()
        text = [i.rstrip('\n') for i in text]
    tokens_list = []
    ner_tags_list = []
    temp_vocab = ''
    temp_label = ''

    for item in text:
        if item == '' or item == '.':  # Empty line or period indicates a new segment
            if temp_vocab:  # Only append if temp_vocab is not empty
                tokens_list.append(temp_vocab)
                ner_tags_list.append(temp_label)
                temp_vocab = ''
                temp_label = ''
        else:
            parts = item.split(' ')
            if len(parts) == 2:  # Ensure the line has both token and label
                vocab, label = parts
                temp_vocab += ' ' + vocab
                temp_label += ' ' + label

    # Append the last segment if it exists
    if temp_vocab:
        tokens_list.append(temp_vocab)
        ner_tags_list.append(temp_label)

    return tokens_list, ner_tags_list
```

```python
# Read the training data
train_tokens, train_ner_tags = read_ner_txt(train_data_path)

# Read the validation data
val_tokens, val_ner_tags = read_ner_txt(val_data_path)

# Read the test data
test_tokens, test_ner_tags = read_ner_txt(test_data_path)
```

圖 2.1、讀取資料程式碼

為了建立標籤對應的編碼表，接著從訓練資料中提取出所有出現過的命名實體標籤，程式碼如圖 2.2 所示。

```python
all_tags = []
for tags in train_ner_tags:
    all_tags.extend(tags.strip().split())

tag_name = sorted(set(all_tags))
```

圖 2.2、提取標籤程式碼

2.2轉換成結構化的資料形式

將句子層級的 tokens_list 與 ner_tags_list 轉換為結構化的 pandas DataFrame，欄位為 tokens 與 ner_tags。此函數同時進行基本預處理，包括移除空值與空字串，並將每筆資料中的字串依空格分割為詞與標籤的列表。這樣的處理能確保資料格式正確，利於後續訓練中轉為張量。

```python
def list_to_dataframe(tokens_list, ner_tags_list):
    df = pd.DataFrame(zip(tokens_list,ner_tags_list), columns = ['tokens','ner_tags'])
    assert all(i for i in ['tokens','ner_tags'] if i in df.columns)

    # pre-processing
    df = df.dropna()
    df = df.drop(df[df['tokens']==''].index.values, )
    df = df.reset_index(drop=True)
    df['tokens'] = df['tokens'].map(lambda x: x.strip().split(' '))
    df['ner_tags'] = df['ner_tags'].map(lambda x: x.strip().split(' '))
    # df.head()
    return df
```

圖 2.3、結構化函式程式碼

並分別將訓練、驗證及測試資料均轉為此型態，如圖 2.4 所示，而輸出結果如圖 2.5 所示。

```python
df = list_to_dataframe(train_tokens, train_ner_tags)
valid_df = list_to_dataframe(val_tokens, val_ner_tags)
test_df = list_to_dataframe(test_tokens, test_ner_tags)
```

圖 2.4、資料轉換型態程式碼

```
                                                tokens  \
0     [The, admin@338, has, largely, targeted, organ...
1     [The, admin@338, started, targeting, Hong, Kon...
2     [Multiple, China-based, cyber, threat, groups,...
```

```
                                                ner_tags
0     [O, B-HackOrg, O, O, O, O, O, O, B-Idus, O, B-...
1     [O, B-HackOrg, O, O, B-Area, I-Area, B-Org, I-...
2     [O, B-Area, B-HackOrg, I-HackOrg, I-HackOrg, O...
```

圖 2.5、轉換結果

2.3轉換訓練用 Dataset 格式

為配合 Hugging Face 訓練流程，我們實作 df_to_dataset 函數，將預處理後的 DataFrame（包含詞語與對應標籤）轉換為 datasets.Dataset 物件。此函數首先將標籤字串轉換為整數編碼（透過 tags.str2int），再與原始的詞語列表一起構成字典格式，並透過 Dataset.from_dict() 生成標準資料格式，程式碼如圖 2.6 所示。

```
tags = ClassLabel(num_classes=len(tag_name), names=tag_name)

dataset_structure = {"ner_tags":Sequence(tags),
                     'tokens': Sequence(feature=Value(dtype='string'))}
```

```
def df_to_dataset(df, columns=['ner_tags', 'tokens']):
  assert set(['ner_tags', 'tokens']).issubset(df.columns)

  ner_tags = df['ner_tags'].map(tags.str2int).values.tolist()
  tokens = df['tokens'].values.tolist()

  assert isinstance(tokens[0], list)
  assert isinstance(ner_tags[0], list)
  d = {'ner_tags':ner_tags, 'tokens':tokens}
  # create dataset
  dataset = Dataset.from_dict(mapping=d,
                              features=Features(dataset_structure),)
  return dataset

train_dataset = df_to_dataset(df)
valid_dataset = df_to_dataset(valid_df)
test_dataset = df_to_dataset(test_df)
```

圖 2.6、建立 dataset 程式碼

最後再將三個 dataset 集合成字典格式，程式碼如圖 2.7 所示，資料集資訊如圖 2.8 所示。

```
dataset = DatasetDict({
    'train': train_dataset,
    'test': test_dataset,
    'valid': valid_dataset})
```

圖 2.7、字典格式程式碼

```
DatasetDict({
    train: Dataset({
        features: ['ner_tags', 'tokens'],
        num_rows: 5251
    })
    test: Dataset({
        features: ['ner_tags', 'tokens'],
        num_rows: 664
    })
    valid: Dataset({
        features: ['ner_tags', 'tokens'],
        num_rows: 662
    })
})
```

圖 2.8、字典格式轉換結果

2.4轉換資料格式並對應標籤

為了實作 SecBert 結合 LSTM 以及 CRF，因此資料格式不能採用 2.2 節至 2.3 節所述，因此將資料先透過 process_data 函數，將原始的 tokens_list 與 tags_list 字串格式資料轉換為「list of list」形式，即每個句子為一個子列表，方便進行編碼與訓練。接著，我們從訓練標籤中提取所有出現過的實體類別，並建立 label2id 與 id2label 的對映關係，用以支援模型訓練中的標籤編碼與預測結果的解碼。

```python
# === Preprocess to list of list ===
def process_data(tokens_list, tags_list):
    tokens = [s.strip().split() for s in tokens_list]
    tags = [s.strip().split() for s in tags_list]
    return tokens, tags

train_tokens, train_labels = process_data(train_tokens, train_ner_tags)
valid_tokens, valid_labels = process_data(val_tokens, val_ner_tags)
test_tokens, test_labels   = process_data(test_tokens, test_ner_tags)

# === create label map ===
unique_tags = sorted(set(tag for seq in train_labels for tag in seq))
label2id = {t: i for i, t in enumerate(unique_tags)}
id2label = {i: t for t, i in label2id.items()}
```

圖 2.9、資料轉換與對應標籤程式碼

資料轉換程式碼如圖 2.9 所示，而結果如圖 2.10 所示。

```
(['The',
  'admin@338',
  'has',
  'largely',
  'targeted',
  'organizations',
  'involved',
  'in',
  'financial',
  ',',
```

```
{'B-Area': 0,
 'B-Exp': 1,
 'B-Features': 2,
 'B-HackOrg': 3,
 'B-Idus': 4,
 'B-OffAct': 5,
 'B-Org': 6,
```

圖 2.10、資料轉換與對應標籤結果示意圖

最後與 2.3 節結尾步驟相同，將資料存成字典格式，程式碼如圖 2.11 所示。

```python
dataset = DatasetDict({
    "train": Dataset.from_dict({"tokens": train_tokens, "ner_tags": train_labels}),
    "valid": Dataset.from_dict({"tokens": valid_tokens, "ner_tags": valid_labels}),
    "test":  Dataset.from_dict({"tokens": test_tokens,  "ner_tags": test_labels}),
})
```

圖 2.11、資料轉換字典程式碼

三、 建立模型及訓練細節

在本次作業中，嘗試了基於 distilbert-base-uncased 以及 SecBert+LSTM+CRF 的模型架構，分別以相同資料集進行訓練並比較兩者結果差異。首先定義模型及 Tokenizer，並對資料進行 Tokenize，再進行訓練，分別介紹如下：

3.1 模型架構

以兩個小節分別介紹 distilbert-base-uncased 以及 SecBert+LSTM+CRF 的模型架構，並且先將設備設定為 GPU，如圖 3.1 所示。

```
#check if gpu is present
device = torch.device('cuda') if torch.cuda.is_available() else torch.device('cpu')
```

圖 3.1、設備指定程式碼

3.1.1 Distilbert-Base-Uncased 架構

此模型是 BERT 基礎模型的精簡版本，DistilBERT 是一個 Transformer 模型，比 BERT 更小、更快，它以自我監督的方式在同一語料庫上進行預訓練，使用 BERT 基礎模型作為老師。這意味著它僅對原始文字進行了預訓練，沒有人工以任何方式對其進行標記，而 Uncased 意思為均為小寫字母輸出入。

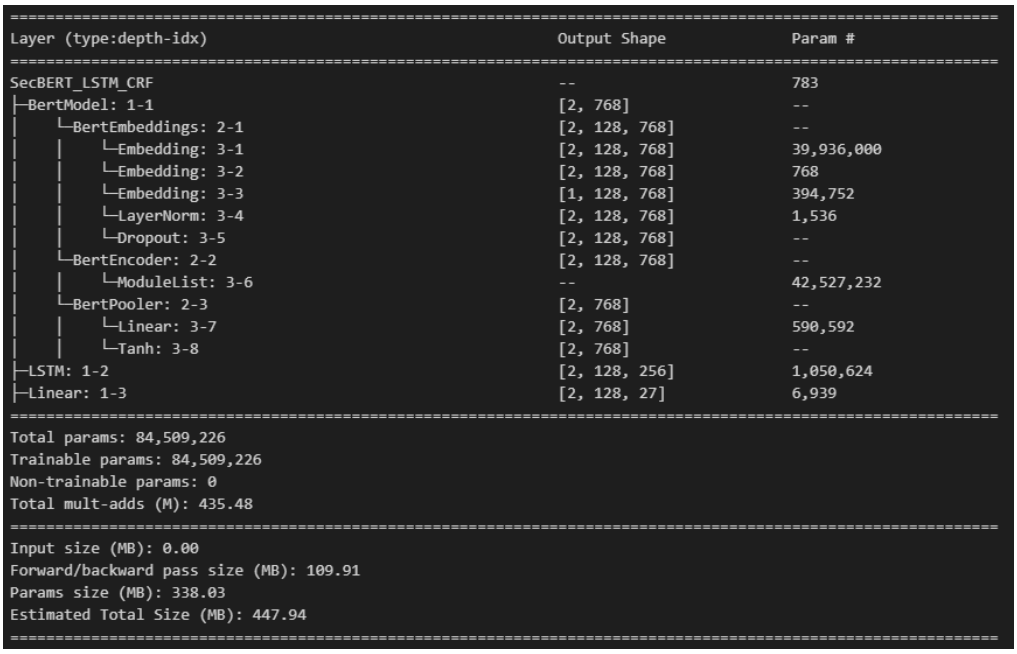
```
tokenizer = AutoTokenizer.from_pretrained("distilbert-base-uncased")
data_collator = DataCollatorForTokenClassification(tokenizer)

model = AutoModelForTokenClassification.from_pretrained("distilbert-base-uncased", num_labels=len(label_names))
model.to(device)
```

圖 3.2、定義 distilbert-base-uncased 程式碼

在模型初始化階段，程式碼如圖 3.2 所示，從 Hugging Face 上下載 distilbert-base-uncased 作為預訓練語言模型，並使用 Hugging Face 提供的 AutoTokenizer 進行分詞處理。為了支援命名實體辨識任務，我們透過 AutoModelForTokenClassification 建立分類模型，並指定標籤數量為實體類別的總數 num_labels。此外，為配合批次訓練，使用 DataCollatorForTokenClassification 搭配 tokenizer 自動補齊與對齊標籤。最後，將模型移至 GPU 以加速訓練流程。

3.1.2 SecBert+LSTM+CRF 架構

圖 3.3 顯示的是結合 SecBERT、LSTM 與 CRF 的命名實體識別（Named Entity Recognition, NER）模型架構，本模型主要由三個模組所組成，分別負責語意特徵提取、序列依賴建模與標籤推論。



```
==========================================================================================================
Layer (type:depth-idx)                                  Output Shape              Param #
==========================================================================================================
SecBERT_LSTM_CRF                                        --                        783
├─BertModel: 1-1                                        [2, 768]                  --
│    └─BertEmbeddings: 2-1                              [2, 128, 768]             --
│    │    └─Embedding: 3-1                              [2, 128, 768]             39,936,000
│    │    └─Embedding: 3-2                              [2, 128, 768]             768
│    │    └─Embedding: 3-3                              [1, 128, 768]             394,752
│    │    └─LayerNorm: 3-4                              [2, 128, 768]             1,536
│    │    └─Dropout: 3-5                                [2, 128, 768]             --
│    └─BertEncoder: 2-2                                 [2, 128, 768]             --
│    │    └─ModuleList: 3-6                             --                        42,527,232
│    └─BertPooler: 2-3                                  [2, 768]                  --
│    │    └─Linear: 3-7                                 [2, 768]                  590,592
│    │    └─Tanh: 3-8                                   [2, 768]                  --
├─LSTM: 1-2                                             [2, 128, 256]             1,050,624
├─Linear: 1-3                                           [2, 128, 27]              6,939
==========================================================================================================
Total params: 84,509,226
Trainable params: 84,509,226
Non-trainable params: 0
Total mult-adds (M): 435.48
==========================================================================================================
Input size (MB): 0.00
Forward/backward pass size (MB): 109.91
Params size (MB): 338.03
Estimated Total Size (MB): 447.94
==========================================================================================================
```

圖 3.3、SecBert+LSTM+CRF 架構

首先，SecBERT 作為 Tokenizer，將每個輸入詞轉換為語意向量，模型採用 Hugging Face 上的"jackaduma/SecBERT"。接著，這些向量會輸入至 LSTM 層，以捕捉序列中詞與詞之間的依賴關係。最後，透過條件隨機場（CRF）層，考量標籤間的轉移關係，輸出最佳的標註序列，模型總參數量為 84,509,226 而總大小為 447(MB)，程式碼如圖 3.4 所示。

```python
class SecBERT_LSTM_CRF(nn.Module):
    def __init__(self, model_name, hidden_dim, tagset_size, bidirectional=False):
        super().__init__()
        self.bert = AutoModel.from_pretrained(model_name)
        lstm_output_dim = hidden_dim * 2 if bidirectional else hidden_dim
        self.lstm = nn.LSTM(self.bert.config.hidden_size, hidden_dim, num_layers=1,
                            batch_first=True, bidirectional=bidirectional)
        self.fc = nn.Linear(lstm_output_dim, tagset_size)
        self.crf = CRF(tagset_size, batch_first=True)

    def forward(self, input_ids, attention_mask, labels=None):
        x = self.bert(input_ids=input_ids, attention_mask=attention_mask).last_hidden_state
        x, _ = self.lstm(x)
        emissions = self.fc(x)
        if labels is not None:
            loss = -self.crf(emissions, labels, mask=attention_mask.bool(), reduction='mean')
            return loss
        else:
            return self.crf.decode(emissions, mask=attention_mask.bool())
```

圖 3.4、SecBert+LSTM+CRF 架構程式碼

3.2 資料集 Tokenize 處理

由於用 pytorch 訓練 SecBert+LSTM+CRF 模型與 DistilBERT 處理資料方式不同，因此以下分為兩種分詞方式進行說明。

3.2.1 DistilBERT 分詞處理

首先為將原始詞語資料轉換為模型可接受的輸入格式，定義 tokenize_function，透過 Hugging Face 的 tokenizer 對 tokens 欄位中的每個詞句進行分詞處理。考量到輸入為已拆分的詞語列表，因此設定 is_split_into_words=True，並使用 padding="max_length" 及 truncation=True 確保每筆輸入長度一致。接著，利用 dataset.map() 批次處理整個資料集，生成 tokenized_datasets_ 作為訓練輸入，程式碼如圖 3.5 所示。

```
def tokenize_function(examples):
    return tokenizer(examples["tokens"], padding="max_length",
                     truncation=True, is_split_into_words=True)

tokenized_datasets_ = dataset.map(tokenize_function, batched=True)
```

圖 3.5、生成 tokenized datasets

```
#Get the values for input_ids, attention_mask, adjusted labels
def tokenize_adjust_labels(all_samples_per_split):
  tokenized_samples = tokenizer.batch_encode_plus(all_samples_per_split["tokens"],
                is_split_into_words=True, truncation=True)
  # print(tokenized_samples['input_ids'][:2])
  # tokenizer(string, padding=True, truncation=True)
  # assert False
  print(len(tokenized_samples["input_ids"]))
  print(tokenized_samples.word_ids(batch_index=2))
  total_adjusted_labels = []

  for k in range(0, len(tokenized_samples["input_ids"])):
    prev_wid = -1
    word_ids_list = tokenized_samples.word_ids(batch_index=k)
    existing_label_ids = all_samples_per_split["ner_tags"][k]
    i = -1
    adjusted_label_ids = []
    # print(existing_label_ids)
    # print(adjusted_label_ids)
    # assert False
    for word_idx in word_ids_list:
      # Special tokens have a word id that is None. We set the label to -100 so they are automatically
      # ignored in the loss function.
      if(word_idx is None):
        adjusted_label_ids.append(-100)
      elif(word_idx!=prev_wid):
        i = i + 1
        adjusted_label_ids.append(existing_label_ids[i])
        prev_wid = word_idx
      else:
        label_name = label_names[existing_label_ids[i]]
        adjusted_label_ids.append(existing_label_ids[i])

    total_adjusted_labels.append(adjusted_label_ids)

  #add adjusted labels to the tokenized samples
  tokenized_samples["labels"] = total_adjusted_labels
  return tokenized_samples

tokenized_dataset = dataset.map(tokenize_adjust_labels,
                batched=True,
                remove_columns=list(dataset["train"].features.keys()))
```

圖 3.6、對齊分詞後的標籤

在 BERT 模型中，Tokenizer 可能將單一詞語切成多個子詞，因此我們需調整每個 token 對應的標籤，使其能正確對齊。tokenize_adjust_labels 函數透過 tokenizer.batch_encode_plus 取得詞語的 word_ids，再依據分詞後的位置對應原始標籤。對於特殊符號與子詞延伸部分，統一標註為 -100，使其在計算損失時被忽略。最終，我們將調整後的標籤加入 tokenized 結果中，並透過 dataset.map 完成整體資料集的處理，程式碼如圖 3.6 所示。

因此最後 tokenized dataset 如圖 3.7 所示。

```
DatasetDict({
    train: Dataset({
        features: ['input_ids', 'attention_mask', 'labels'],
        num_rows: 5251
    })
    test: Dataset({
        features: ['input_ids', 'attention_mask', 'labels'],
        num_rows: 664
    })
    valid: Dataset({
        features: ['input_ids', 'attention_mask', 'labels'],
        num_rows: 662
    })
})
```

圖 3.7、tokenized dataset 格式

3.2.2 SecBert+LSTM+CRF 分詞處理

此原始標籤對齊處理函式方法如圖 3.8 所示，大致與 3.1.1 小節相同，取得詞語的 word_ids，再依據分詞後的位置對應原始標籤。對於特殊符號與子詞延伸部分，我們統一標註為 -100，只有因為資料輸入不同而重新定義函式。

```
def tokenize_and_align_labels(examples):
    tokenized = tokenizer(
        examples["tokens"],
        is_split_into_words=True,
        truncation=True,
        padding="max_length",
        max_length=128
    )
    aligned_labels = []
    for i, word_labels in enumerate(examples["ner_tags"]):
        word_ids = tokenized.word_ids(batch_index=i)
        label_ids = []
        previous_word_idx = None
        for word_idx in word_ids:
            if word_idx is None:
                label_ids.append(-100)
            elif word_idx != previous_word_idx:
                label_ids.append(label2id[word_labels[word_idx]])
            else:
                label_ids.append(-100)
            previous_word_idx = word_idx
        aligned_labels.append(label_ids)
    tokenized["labels"] = aligned_labels
    return tokenized
```

圖 3.8、tokenize_and_align_labels 函數程式碼

不同的是完成了資料轉換、標籤對齊、格式設定後，須建立訓練所需的DataLoader，如圖 3.9 所示。

```
dataset = DatasetDict({
    "train": Dataset.from_dict({"tokens": train_tokens, "ner_tags": train_labels}),
    "valid": Dataset.from_dict({"tokens": valid_tokens, "ner_tags": valid_labels}),
    "test": Dataset.from_dict({"tokens": test_tokens, "ner_tags": test_labels}),
})

dataset = dataset.map(tokenize_and_align_labels, batched=True)
dataset.set_format("torch", columns=["input_ids", "attention_mask", "labels"])

# === 建 dataloader ===
from torch.utils.data import DataLoader
train_loader = DataLoader(dataset["train"], batch_size=128, shuffle=True)
valid_loader = DataLoader(dataset["valid"], batch_size=128)
test_loader = DataLoader(dataset["test"], batch_size=1)
```

圖 3.9、建立 Dataloader 程式碼

以 tokenize_and_align_labels 函數進行分詞與標籤對齊處理。經過 set_format("torch") 設定後，資料即可轉為 PyTorch 所需的張量格式，包含 input_ids、attention_mask 及 labels。最後，透過 DataLoader 建立批次讀取器，支援模型訓練與驗證時的高效數據載入與隨機打亂。

3.3模型超參數設置

所有的超參數都是透過命令列參數來設定的，其預設值如表 3.1 所示。隱藏層維度(hidding dim)設為 256，LSTM 輸入層大小與隱藏層維度相同。此外，訓練和驗證的批次大小均設定為 128，以便於觀察每個樣本的預測結果，本次實驗中各模型的訓練 epoch 範圍均設為 20。這些參數共同決定了模型的結構、容量以及訓練時的數據處理方式，從而最終影響分類任務的效果。

表 3.1、超參數設定表

超參數名稱	數值
hidding dim	256
out dim	27
Learning rate	5e-5
Train Batch size	128
Val Batch size	128
Epoch	20

此外在訓練設定中，在 Optimizer 優化器使用 AdamW 優化器，其自適應學習率調整機制有助於加快收斂並提高訓練效率，這是目前在 Transformer 架構中最常見的選擇之一。AdamW 在 Adam 的基礎上加入了對權重衰減（weight decay）的正歸化處理，有助於減少過擬合現象，提升模型的泛化能力，並且學習率設為 5e-5。

四、 實驗介紹與結果

4.1 實驗介紹

本章節中對於第三章提到的兩種模型進行實驗，比較兩者模型的性能，詳細如表 4.1 所示。

表 4.1、模型實驗詳細內容表

Model_Name	Configuration
DistilBert	DtilBert 架構
SecBert+LSTM+CRF	SecBert 結合 LSTM 與 CRF 架構

4.2 實驗結果

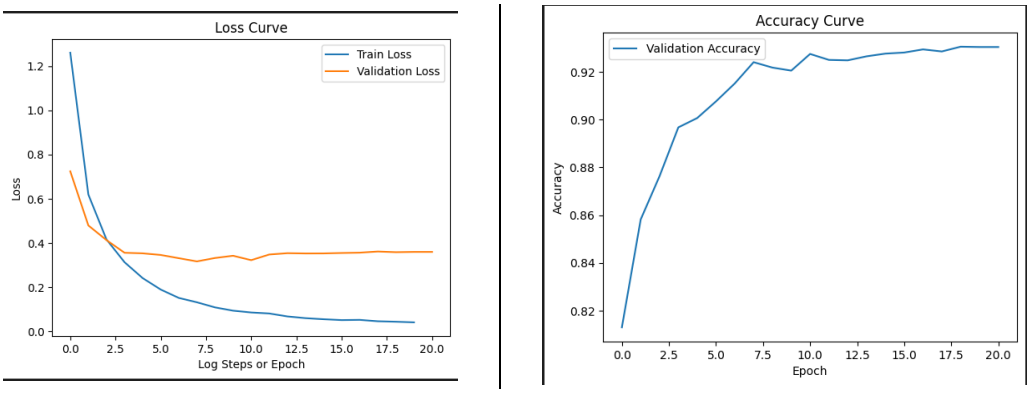


圖 4.1、DistilBert Train/Val loss 及 Val Accuracy 圖

採用 DistilBERT 作為命名實體辨識任務的預訓練語言模型，並進行微調訓練，圖 4.1 為 loss 以及 Accuracy 圖。在驗證集上的表現結果如圖 4.2 顯示，模型在第 20 個 epoch 時已達穩定收斂，其驗證損失為 0.359，代表模型在處理未見資料時仍具有良好的預測能力。從指標來看，模型的精確率為 82.69%，召回率為 85.42%，F1 分數為 84.04%，顯示模型在正確預測命名實體與完整覆蓋實體標註上具有良好平衡。而整體準確率達到 93.04%，反映模型整體辨識結果的正確性。

```
{'eval_loss': 0.3589390516281128,
 'eval_precision': 0.8269190325972661,
 'eval_recall': 0.8542255051053661,
 'eval_f1': 0.8403505022440693,
 'eval_accuracy': 0.9304489944159441,
 'eval_runtime': 0.6763,
 'eval_samples_per_second': 978.9,
 'eval_steps_per_second': 8.872,
 'epoch': 20.0}
```

圖 4.2、DistilBert 驗證集表現

```
'overall_precision': np.float64(0.8668357641090678),
'overall_recall': np.float64(0.8845987920621226),
'overall_f1': np.float64(0.8756272018789367),
'overall_accuracy': 0.949143198419626}
```

圖 4.3、DistilBert 測試集表現

在測試集上的最終表現結果，DistilBERT 模型在未見資料上仍保持穩定且優異的表現，其整體精確率（Precision）為 86.68%，召回率（Recall）為 88.46%，F1 分數達到 87.56%，整體準確率則高達 94.91%。這些結果顯示，模型不僅能準確辨識命名實體，亦能有效涵蓋大部分實體標註，在不同類型命名實體的辨識上具備良好的一致性與穩定性。

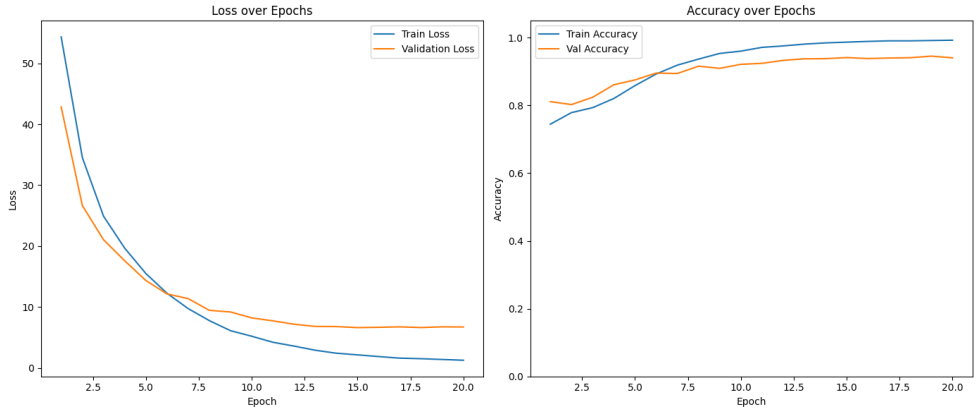


圖 4.4、SecBert+LSTM+CRF Train/Val loss 及 Val Accuracy 圖

SecBert+LSTM+CRF 模型於驗證集上表現出穩定學習趨勢。從 Loss 圖如圖 4.4 左圖可觀察到，訓練與驗證損失皆隨訓練進行而下降，顯示模型成功學習語意與標註模式，最終驗證損失降至 6.81。準確率如圖 4.4 右圖中顯示模型的訓練準確率逐步逼近 1.0，而驗證準確率則穩定上升至 0.941，代表模型具良好的泛化能力。

```
[Epoch 20] Val Loss: 6.7151 | Accuracy: 0.9406 | Precision: 0.7330 | Recall: 0.8130 | F1: 0.7709
```

圖 4.5、SecBert+LSTM+CRF 驗證集表現

在驗證指標方面，模型於驗證集達到 Precision 為 73.30%、Recall 為 81.30%、F1-score 為 77.09%，且 Accuracy 為 94.06%，顯示在不同類型實體的辨識上仍具良好涵蓋性。整體來看，該結合架構能有效捕捉上下文資訊並透過 CRF 增強序列標註一致性，是一個適合用於 NER 任務的結構設計。

```
Precision: 0.79913
Recall:    0.85976
F1 Score:  0.82834
Accuracy:  0.95421
```

圖 4.6、SecBert+LSTM+CRF 測試集表現

在測試階段，SecBERT-LSTM-CRF 模型展現出優異性能，驗證其在未見資料上的實用性與穩定性。具體來說，模型在測試集上達到 Precision 為 79.92%、Recall 為 85.98%，而整體 F1 Score 為 82.83%，展現出相當平衡且穩定的實體辨識能力。此外，Accuracy 高達 95.42%，顯示模型對大多數標註能做出正確預測，且錯誤率低。

綜合兩種模型表現，DistilBERT 在驗證集上擁有較高的準確率與 F1 分數，而 SecBERT-LSTM-CRF 雖然在驗證集準確率略低，但 SecBERT-LSTM-CRF 測試集達到 95.42%，略高於 DistilBERT 的 94.91%，反映其具備良好的整體預測正確性。綜合考量，DistilBERT 模型在訓練效率與準確性方面具優勢，適合資源受限的應用場景；而 SecBERT-LSTM-CRF 結合序列建模與標籤依賴性學習能力，在需強化語境理解與連續實體辨識的情境中表現出更穩定的實務潛力。