遞迴神經網路與變形器

作業一

學號：313831025

姓名：俞博云

一、 題目敘述

在本次作業中，首先需要對文本數據進行預處理，接著利用 LSTM 模型完成文本分類任務，以區分 AI 生成與人類生成的文本。目標是盡可能提高測試數據集上的準確率。

二、 數據預處理

本作業選用 AI-vs-Human-text 資料集，大約包含 50 萬筆資料。首先需要對資料集進行預處理成使模型可以理解的數值形式，主要分為**數據讀取、建立詞彙表、拆分數據集、封裝自定義數據集以及建立 DataLoaders：**

2.1 數據讀取

數據集共約有 50 萬筆資料，其中包含兩種資料，為”text”及”generated”，如表 2.1 所示，”text”為內文，而”generated”包含”0.0”及”1.0”，表示為 AI 及人類。

表 2.1、數據集資料

text	generated
Cars. Cars have been around since they became famous in the 1900s, when Henry Ford created and built...	0.0
A Sustainable Urban Vision In an increasingly urbanized world, the concept of car-free cities is ...	1.0

首先對於數據資料集(CSV 格式)使用 Pandas 讀取，並且針對上述兩個標籤分別存取，詳細程式碼如圖 2.1 所示。

```
86          # Read the dataset from the CSV file.
87          df = pd.read_csv(self.csv_path)
88          texts = df[self.text_col].tolist()
89          labels = df[self.label_col].tolist()
```

圖 2.1、讀取資料程式碼

2.2 建立詞彙表

接著對於”text”標籤的資料建立詞彙表，將每個單詞映射到一個唯一的數字 ID，具體流程如下：

首先，初始化一個計數器；然後對每條文本進行分詞，並累計每個單詞的出現次數。接著，遍歷計數器中的所有單詞，為每個單詞分配一個數字 ID。為了處理未知單詞與序列填充，預留了 0 和 1，分別對應特殊符號“<unk>”（未知詞）和“<pad>”（填充符號）。程式碼如圖 2.2 所示。

```
73      def _build_vocab(self, texts):
74          # Build a vocabulary from the texts using a Counter to count word frequencies.
75          counter = Counter()
76          for text in texts:
77              counter.update(tokenizer(text))
78          # Create vocabulary mapping starting from index 2
79          # (reserve index 0 for <unk> and index 1 for <pad>)
80          vocab = {word: i+2 for i, (word, _) in enumerate(counter.items())}
81          vocab['<unk>'] = 0
82          vocab['<pad>'] = 1
83          return vocab
```

圖 2.2、建立詞彙表程式碼

2.3 拆分數據集

將數據集拆分成訓練資料集 、驗證資料集以及測試資料集，比例分別為 70%、15%及 15%。具體來說，首先設定 random_state 參數為 42，確保每次拆分為相同數據，再將原始數據集隨機劃分成 70%的訓練集和 30%的驗證+測試集；接著，將驗證+測試集再次按 1:1 的比例拆分，分別獲得 15%的驗證集與 15%的測試集。程式碼如圖 2.3 所示。

```
94          # Split dataset into training and combined validation+test sets.
95          texts_train, texts_val_test, labels_train, labels_val_test = train_test_split(
96              texts, labels, test_size=self.test_ratio+self.val_ratio, random_state=self.random_state)
97
98          # Split combined set into validation and test sets.
99          texts_val, texts_test, labels_val, labels_test = train_test_split(
100             texts_val_test, labels_val_test, test_size=self.test_ratio/(self.test_ratio+self.val_ratio), random_state=self.random_state)
```

圖 2.3、拆分數據程式碼

2.4 封裝自定義數據集

這個步驟封裝了數據讀取與數值化的過程，使得每次從數據集中取出的數據都已經轉換成了模型可直接使用的數字張量格式。將訓練資料集、驗證資料集以及測試資料集分別定義為一個類別資料以便 DataLoader 讀取。程式碼如圖 2.5 及圖 2.6 所示，而圖 2.4 為兩個用於文本處理的函數，tokenizer 函數文本作為輸入，並利用空格將文本拆分成多個單詞，從而返回一個單詞列表。Numericalize 函數則是先將文本分詞，然後依據提供的詞彙表將每個單詞轉換為對應的數字 ID。如果某個單詞不在詞彙表中，則使用詞彙表中預設的未知詞”<unk>”的數字 ID 來替代。

```
7     def tokenizer(text):
8         # Split the text into words based on whitespace
9         return text.split()
10
11    def numericalize(text, vocab):
12        # Convert text into a list of numbers according to the given vocabulary.
13        # If a word is not in the vocabulary, the '<unk>' token index is used.
14        return [vocab.get(word, vocab['<unk>']) for word in tokenizer(text)]
```

圖 2.4、工具 Fuction 程式碼

```
27  class TextDataset(Dataset):
28      def __init__(self, texts, labels, vocab):
29          # Initialize dataset with texts, labels and vocabulary
30          self.texts = texts
31          self.labels = labels
32          self.vocab = vocab
33
34      def __len__(self):
35          # Return the total number of samples in the dataset
36          return len(self.texts)
37
38      def __getitem__(self, idx):
39          # Retrieve the numericalized text and corresponding label for a given index
40          text = numericalize(self.texts[idx], self.vocab)
41          label = self.labels[idx]
42          return torch.tensor(text), torch.tensor(label, dtype=torch.float)
```

圖 2.5、封裝自定義數據集程式碼

```
102           # Create TextDataset objects for training, validation, and test sets.
103           train_dataset = TextDataset(texts_train, labels_train, self.vocab)
104           val_dataset = TextDataset(texts_val, labels_val, self.vocab)
105           test_dataset = TextDataset(texts_test, labels_test, self.vocab)
```

圖 2.6、創建封裝自定義數據集程式碼

2.5 建立 DataLoaders

這個步驟用於為訓練、驗證與測試數據集建立 DataLoader，從而在訓練或評估模型時可以自動生成批次數據。

對於訓練數據集，DataLoader 從上述自定義訓練資料集中按照 batch_size 提取數據，並且使用自定義的 collate 函數如圖 2.7 所示，根據詞彙表中"<pad>"的索引對批次內不同長度的文本序列進行填充，確保所有數據具有相同的維度。並且在每個 epoch 都會隨機打亂數據順序。

```python
def pad_collate_fn(batch, pad_idx):
    # Custom collate function to pad sequences in a batch to the same length.
    texts, labels = zip(*batch)
    text_lens = [len(text) for text in texts]
    max_len = max(text_lens)
    # Create a tensor filled with pad index values.
    padded_texts = torch.full((len(texts), max_len), pad_idx, dtype=torch.long)
    for i, text in enumerate(texts):
        padded_texts[i, :text_lens[i]] = text
    return padded_texts, torch.tensor(labels, dtype=torch.float)
```

圖 2.7、創建封裝自定義數據集程式碼

對於驗證和測試數據均不會打亂數據，以便在調參和最終模型評估時保持結果的穩定性，詳細程式碼如圖 2.8 所示。

```python
        # Create DataLoaders for each split.
        self.train_loader = DataLoader(
            train_dataset,
            batch_size=self.batch_size_train,
            shuffle=True,
            collate_fn=lambda batch: pad_collate_fn(batch, self.vocab['<pad>'])
        )
        self.val_loader = DataLoader(
            val_dataset,
            batch_size=self.batch_size_val,
            shuffle=False,
            collate_fn=lambda batch: pad_collate_fn(batch, self.vocab['<pad>'])
        )
        self.test_loader = DataLoader(
            test_dataset,
            batch_size=self.batch_size_test,
            shuffle=False,
            collate_fn=lambda batch: pad_collate_fn(batch, self.vocab['<pad>'])
        )
```

圖 2.8、建立 DataLoaders 程式碼

三、 LSTM 模型架構與訓練細節

在本次作業中，嘗試了多種基於 LSTM 的文本分類模型，並在架構上進行了一些變化，以探索不同結構的影響。下面分為 LSTM 模型架構及模型超參數兩個部分進行說明：

3.1 LSTM 模型架構

3.1.1 LSTMClassifier

此為基本的 LSTM 架構，分為三個部分，如圖 3.1 所示，分別為**嵌入層 (Embedding Layer)、LSTM 層以及全連接層(Fully Connected Layer)**。

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
LSTMClassifier                           [64, 1]                   --
├─Embedding: 1-1                         [64, 100, 100]            54,483,000
├─LSTM: 1-2                              [64, 100, 256]            892,928
├─Linear: 1-3                            [64, 1]                   257
==========================================================================================
Total params: 55,376,185
Trainable params: 55,376,185
Non-trainable params: 0
Total mult-adds (G): 9.20
```

圖 3.1、LSTMClassifier 架構

程式碼如圖 3.2 所示，建立完三層基本架構後，模型會先檢查輸入序列長度是否為零，如果是會返回全零的張量以免計算出錯，再將輸入的單詞索引轉換為對應的嵌入向量，接著將向量序列傳入 LSTM 層最後透過全連接層映射成一個 Logit 值作為二分類任務的輸出值。

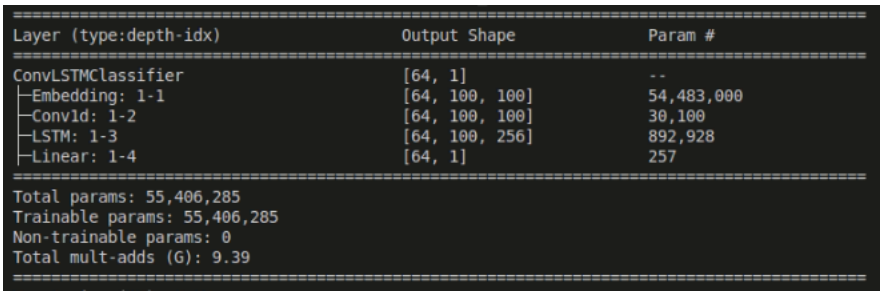
```python
class LSTMClassifier(nn.Module):
    """
    # This LSTMClassifier uses an embedding layer followed by a multi-layer LSTM for text classification.
    """
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, padding_idx, num_layers=1):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=padding_idx)
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True, num_layers=num_layers)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        # Check if sequence length is zero and return zeros if so.
        if text.size(1) == 0:
            return torch.zeros(text.size(0), self.fc.out_features, device=text.device)
        # Convert input word indices to embeddings
        embedded = self.embedding(text)
        # Process embeddings through LSTM
        _, (hidden, _) = self.lstm(embedded)
        # Use the last hidden state for classification
        return self.fc(hidden[-1])
```

圖 3.2、LSTMClassifier 架構程式碼

3.1.2 ConvLSTMClassifier

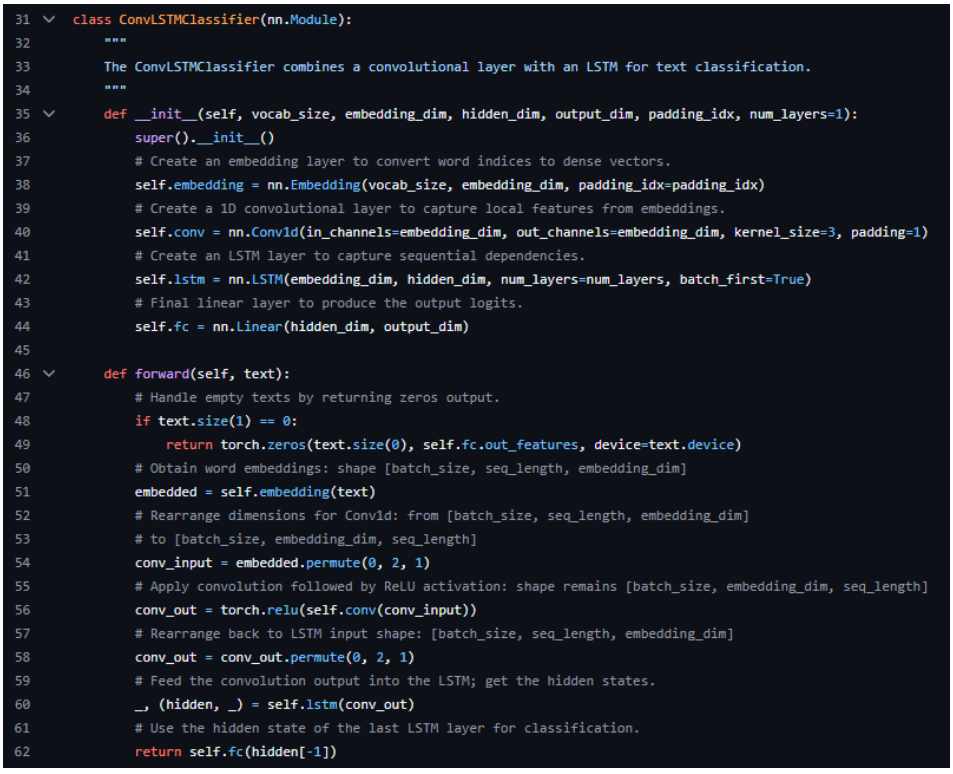
此為基本的 LSTM 架構，分為四個部分，如圖 3.3 所示，分別為**嵌入層 (Embedding Layer)、一維卷積層、LSTM 層以及全連接層 (Fully Connected Layer)**。



```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
ConvLSTMClassifier                       [64, 1]                   --
├─Embedding: 1-1                         [64, 100, 100]            54,483,000
├─Conv1d: 1-2                            [64, 100, 100]            30,100
├─LSTM: 1-3                              [64, 100, 256]            892,928
├─Linear: 1-4                            [64, 1]                   257
==========================================================================================
Total params: 55,406,285
Trainable params: 55,406,285
Non-trainable params: 0
Total mult-adds (G): 9.39
==========================================================================================
```

圖 3.3、ConvLSTMClassifier 架構

程式碼如圖 3.4 所示，建立完四層的模型架構後，模型會與 3.1.1 節基本架構做相同動作，不同的是此結構結合了卷積神經網絡和長短期記憶網絡的優點，建立了一個一維卷積層，用來從嵌入向量中提取局部特徵。卷積層的 kernel_size 設為 3 並使用 padding 保持序列長度不變，依次進行嵌入、卷積、ReLU 激活、並且重新轉換回 LSTM 所需的形狀，最後通過 LSTM 層獲取隱藏狀態並用全連接層進行分類輸出。



```python
class ConvLSTMClassifier(nn.Module):
    """
    The ConvLSTMClassifier combines a convolutional layer with an LSTM for text classification.
    """
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, padding_idx, num_layers=1):
        super().__init__()
        # Create an embedding layer to convert word indices to dense vectors.
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=padding_idx)
        # Create a 1D convolutional layer to capture local features from embeddings.
        self.conv = nn.Conv1d(in_channels=embedding_dim, out_channels=embedding_dim, kernel_size=3, padding=1)
        # Create an LSTM layer to capture sequential dependencies.
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers=num_layers, batch_first=True)
        # Final linear layer to produce the output logits.
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, text):
        # Handle empty texts by returning zeros output.
        if text.size(1) == 0:
            return torch.zeros(text.size(0), self.fc.out_features, device=text.device)
        # Obtain word embeddings: shape [batch_size, seq_length, embedding_dim]
        embedded = self.embedding(text)
        # Rearrange dimensions for Conv1d: from [batch_size, seq_length, embedding_dim]
        # to [batch_size, embedding_dim, seq_length]
        conv_input = embedded.permute(0, 2, 1)
        # Apply convolution followed by ReLU activation: shape remains [batch_size, embedding_dim, seq_length]
        conv_out = torch.relu(self.conv(conv_input))
        # Rearrange back to LSTM input shape: [batch_size, seq_length, embedding_dim]
        conv_out = conv_out.permute(0, 2, 1)
        # Feed the convolution output into the LSTM; get the hidden states.
        _, (hidden, _) = self.lstm(conv_out)
        # Use the hidden state of the last LSTM layer for classification.
        return self.fc(hidden[-1])
```

圖 3.4、ConvLSTMClassifie 架構程式碼

3.1.3 BiStackedLSTMClassifier

此架構與基本的 LSTM 架構相同，分為三個部分，如圖 3.5 所示，**分別為嵌入層(Embedding Layer)、LSTM 層以及全連接層(Fully Connected Layer)**，不同的是設定了雙向堆疊使模型同時捕捉了正向和反向的上下文訊息。

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
BiStackedLSTMClassifier                  [64, 1]                   --
├─Embedding: 1-1                         [64, 100, 100]            54,483,000
├─LSTM: 1-2                              [64, 100, 512]            2,310,144
├─Linear: 1-3                            [64, 1]                   513
==========================================================================================
Total params: 56,793,657
Trainable params: 56,793,657
Non-trainable params: 0
Total mult-adds (G): 18.27
==========================================================================================
```

圖 3.5、BiStackedLSTMClassifier 架構

程式碼如圖 3.6 所示，程式運作如同基本架構，不同的是通過設定 bidirectional=True，建立了一個雙向 LSTM 層，該層能夠同時處理從正向和反向讀取的序列訊息。由於是雙向的，LSTM 輸出的隱藏狀態數量會翻倍，因此全連接層的輸入維度設定為原本的兩倍，以便將正向和反向的隱藏狀態合併，最後，這個 Concat 後的隱藏狀態傳入全連接層，生成最終的分類結果。

```python
class BiStackedLSTMClassifier(nn.Module):
    """
    BiStackedLSTMClassifier implements a bidirectional LSTM classifier.
    It applies an embedding layer followed by a bidirectional LSTM and a fully connected layer.
    """
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, padding_idx, num_layers=2):
        super().__init__()
        # Embedding layer to convert input token indices into dense vectors.
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=padding_idx)
        # Bidirectional LSTM layer with the specified number of layers.
        # batch_first=True means the input shape is [batch_size, sequence_length, embedding_dim].
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True, num_layers=num_layers, bidirectional=True)
        # Fully connected layer that maps the concatenated hidden states to the output dimension.
        self.fc = nn.Linear(hidden_dim * 2, output_dim)

    def forward(self, text):
        # If the input text has zero length, return a tensor of zeros with appropriate shape.
        if text.size(1) == 0:
            return torch.zeros(text.size(0), self.fc.out_features, device=text.device)
        # Convert input token indices into embeddings.
        embedded = self.embedding(text)
        # Pass embeddings through the bidirectional LSTM.
        # The output is ignored as we are only interested in the hidden states.
        _, (hidden, _) = self.lstm(embedded)
        # Concatenate the last forward and backward hidden states.
        hidden_cat = torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1)
        # Pass the concatenated hidden states through the fully connected layer to get logits.
        return self.fc(hidden_cat)
```

圖 3.6、BiStackedLSTMClassifier 架構程式碼

3.1.4 AttentionBiStackedLSTMClassifier

此架構結合了 3.1.3 小節的架構以及 Attention 機制，分為四個部分，如圖 3.7 所示，分別為**嵌入層(Embedding Layer)、LSTM 層、Attention 層以及全連接層(Fully Connected Layer)**。

```
==========================================================================================
Layer (type:depth-idx)                   Output Shape              Param #
==========================================================================================
AttenationBiStackedLSTMClassifier        [64, 1]                   --
├─Embedding: 1-1                         [64, 100, 100]            54,483,000
├─LSTM: 1-2                              [64, 100, 512]            2,310,144
├─Attention: 1-3                         [64, 512]                 512
│    └─Linear: 2-1                       [64, 1, 512]              262,656
│    └─Linear: 2-2                       [64, 100, 512]            262,656
├─Linear: 1-4                            [64, 1]                   513
==========================================================================================
Total params: 57,319,481
Trainable params: 57,319,481
Non-trainable params: 0
Total mult-adds (G): 18.31
==========================================================================================
```

圖 3.7、AttentionBiStackedLSTMClassifier 架構

```python
class Attention(nn.Module):
    """
    Attention module that computes a context vector.
    It uses the current hidden state as the query and the encoder outputs as keys to
    calculate compatibility scores. The scores undergo a softmax to produce attention
    weights, which are then used to compute a weighted sum of the encoder outputs.
    """
    def __init__(self, hidden_dim):
        super().__init__()
        # Linear layer to transform the query (hidden state)
        self.W = nn.Linear(hidden_dim, hidden_dim)
        # Linear layer to transform the encoder outputs (keys)
        self.U = nn.Linear(hidden_dim, hidden_dim)
        # Learnable parameter for computing the compatibility score
        self.v = nn.Parameter(torch.randn(hidden_dim))

    def forward(self, hidden, encoder_outputs):
        # hidden: (batch_size, hidden_dim)
        # encoder_outputs: (batch_size, seq_length, hidden_dim)

        # Expand hidden to (batch_size, 1, hidden_dim) to use it as the query for attention
        hidden = hidden.unsqueeze(1)

        # Compute intermediate scores by applying a tanh activation on the sum of
        # transformed hidden (query) and encoder outputs (keys); shape: (batch_size, seq_length, hidden_dim)
        score = torch.tanh(self.W(hidden) + self.U(encoder_outputs))

        # Compute raw attention scores by taking the dot product with the learnable vector v;
        # resulting shape: (batch_size, seq_length)
        attention_weights = torch.matmul(score, self.v)

        # Apply softmax to obtain a probability distribution over the sequence length
        attention_weights = F.softmax(attention_weights, dim=1)

        # Compute the context vector as the weighted sum of encoder outputs according to the attention weights;
        # resulting shape: (batch_size, hidden_dim)
        context_vector = torch.sum(attention_weights.unsqueeze(-1) * encoder_outputs, dim=1)
        return context_vector
```

圖 3.8、Attention 機制程式碼

建立一個 Attention 機制如圖 3.8 所示，目的是根據當前隱藏狀態與編碼器的輸出計算一個上下文向量，用以捕捉文本中最相關的部分。具體來說，定義了兩層線性層，一個做為 query 另一個用於 Encoder 的輸出，這當中再設定一個可學習的參數向量用於計算 query 與 key 間的分數。

首先將隱藏狀態與編碼器輸出相加。接著，對擴展後的隱藏狀態和編碼器輸出分別進行線性變換，並將它們相加後通過 tanh 激活函數得到一個中間分數，再將這個中間分數與向量 v 進行內積運算，產生每個時間的注意力分數，最後再進行 softmax 正規化，得到注意力權重，最後，利用這些權重對編碼器輸出進行加權求和，計算得到的上下文向量將作為最終的輸出。

```python
class AttenationBiStackedLSTMClassifier(nn.Module):
    """
    AttenationBiStackedLSTMClassifier implements a bidirectional LSTM classifier augmented with an attention mechanism.
    It utilizes an embedding layer, a bidirectional LSTM to capture both forward and backward context, and an attention module to combine the encoded sequence information.
    The resulting context vector is then passed through a fully connected layer to produce the final output for classification.
    """
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim, padding_idx, num_layers=2):
        super().__init__()
        # Initialize the embedding layer to convert input indices to dense vectors.
        self.embedding = nn.Embedding(vocab_size, embedding_dim, padding_idx=padding_idx)
        # Create a bidirectional LSTM layer.
        # Note: This generates outputs with dimension hidden_dim * 2 due to bidirectionality.
        self.lstm = nn.LSTM(embedding_dim, hidden_dim, batch_first=True, num_layers=num_layers, bidirectional=True)
        # Fully connected layer to map the context vector from attention to the desired output dimension.
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
        # Attention module that operates on the concatenated hidden states (from both directions).
        self.attention = Attention(hidden_dim * 2)

    def forward(self, text):
        # Handle empty input: if the sequence length is zero, return a tensor of zeros.
        if text.size(1) == 0:
            return torch.zeros(text.size(0), self.fc.out_features, device=text.device)
        # Obtain embeddings from the input text.
        embedded = self.embedding(text)

        # Process the embeddings through the bidirectional LSTM.
        # outputs: tensor of shape [batch_size, seq_length, hidden_dim*2]
        # hidden: tensor of shape [num_layers*2, batch_size, hidden_dim]
        outputs, (hidden, _) = self.lstm(embedded)

        # Concatenate the last hidden state from the forward and backward passes.
        # hidden[-2, :, :] corresponds to the forward LSTM of the last layer.
        # hidden[-1, :, :] corresponds to the backward LSTM of the last layer.
        hidden_cat = torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1)

        # Use the concatenated hidden state as query and the LSTM outputs as keys/values for attention.
        context = self.attention(hidden_cat, outputs)

        # Map the attention context vector to the output logits.
        return self.fc(context)
```

圖 3.9、AttenationBiStackedLSTMClassifier 架構程式碼

此模型架構程式碼如圖 3.9 所示，模型透過與 3.1.3 節的輸出作為 quary 和 key，再進入 Attention 加權求和後生成上下文向量，最終通過全連接層映射到輸出，得到分類結果。整體來說，這個模型通過結合雙向 LSTM 獲取全局上下文和注意力機制強化關鍵特徵，以強化整體性能。

3.2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。每個單詞的嵌入向量維度(Embedding dim)設為 100，隱藏層(Hidden dim)大小設為 256，這個數值決定了 LSTM 隱藏狀態的大小，並且，這些向量會經過設定的 Layer 層數(num layers)，最終模型在二分類任務中輸出一個標量。此外，訓練和驗證的批次大小均設定為 64，而測試批次大小為 1，以便於觀察每個樣本的預測結果，並且每個實驗均跑 50 個 Epoch。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
Embedding dim	100
Hidden dim	256
Output dim	1
Num layers	1~2
Train Batch size	64
Val Batch size	64
Test Batch size	1
Epoch	50

此外在訓練設定中，損失函數使用 Binary CrossEntropy Loss 結合 Sigmoid 和二元交叉熵計算，因此能夠直接處理模型輸出 logits，提升數值穩定性。在 Optimizer 優化器使用 Admin 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。

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四種模型進行六種實驗，分別對不同 LSTM 結構以及層數進行比較，詳細如表 4.1 所示。

表 4.1、模型實驗詳細內容表

Model_Name	Configuration
LSTM	LSTMClassifier (層數：1)
2LSTM	LSTMClassifier (層數：2)
CNN+LSTM	ConvLSTMClassifier (層數：1)
CNN+2LSTM	ConvLSTMClassifier (層數：2)
BiStackedLSTM	BiStackedLSTMClassifier (層數：2)
AttentionBiStackedLSTM	AttenationBiStackedLSTMClassifier (層數：2)

4.2 實驗結果

從圖 4.1 可看出，所有模型的訓練損失在前幾個 Epoch 都迅速下降，並逐漸趨近於零，說明這些模型能夠成功擬合訓練數據。整體而言，各模型在訓練階段的表現相當接近，收斂速度都很快，代表它們對此任務都有足夠的學習能力。

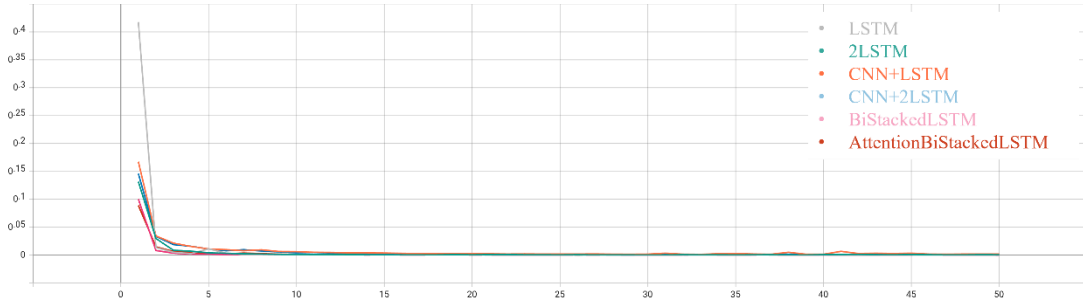


圖 4.1、Train Loss 圖

而從圖 4.2 可看出，隨著訓練 Epoch 的增加，各個模型在驗證集上的損失逐漸下降並進入穩定區域，但穩定度和下降速度存在一定差異。整體來看，模型最終都能將驗證損失維持在相對較低的水準，顯示其在學習過程中並未出現明顯的過擬合現象。但可以發現，BiStackedLSTM 與 AttentionBiStackedLSTM 在後期的曲線波動相對較小，代表它們在驗證階段更為穩定。

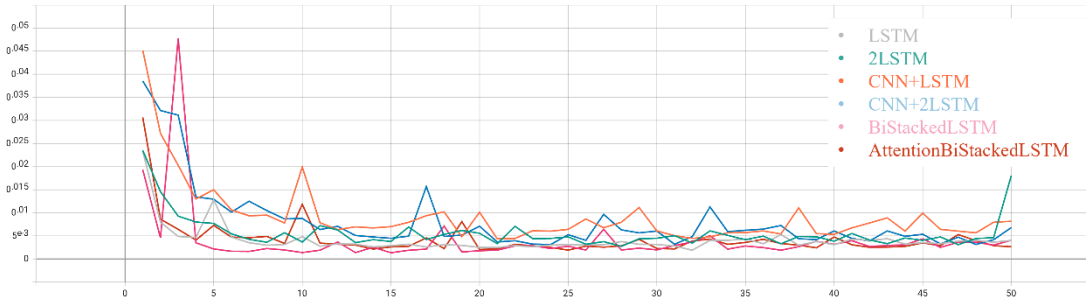


圖 4.2、Val Loss 圖

在驗證準確率上，如圖 4.3 所示，所有模型的準確率幾乎都在初期快速攀升，除了 BiStackedLSTM 以及 CNN+2LSTM，可能是因為模型在初期對數據進行快速調整導致出現短暫的不穩定，但隨著後續訓練，模型均達到了穩定且高水平的準確率並達到接近於 0.999，表明這些模型具有較好的學習能力。

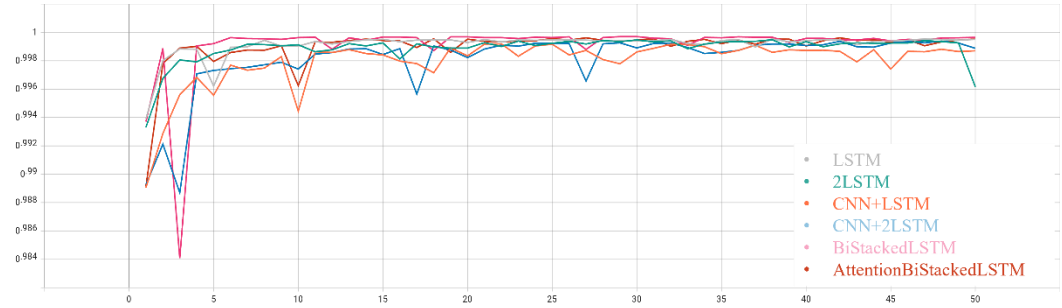


圖 4.3、Accuracy 圖

最後對測試資料集進行實驗，將表 4.1 六種模型對於測試資料集進行比較，實驗結果如表 4.2 所示。

表 4.2、實驗結果數據表

Model_Name	Accuracy
LSTM	0.9973
2LSTM	0.9960
CNN+LSTM	0.9813
CNN+2LSTM	0.9987
BiStackedLSTM	0.9994
AttentionBiStackedLSTM	0.9993

從表 4.2 中的結果可以看出，各種模型在測試集上都達到了相當高的準確率，其中 BiStackedLSTM 模型達到 0.9994 的成績，僅比 AttentionBiStackedLSTM 的 0.9993 高出 0.0001，而 CNN+2LSTM 也有 0.9987 的表現，顯示結合 CNN 與多

層 LSTM 同樣具備不錯的潛力。相較之下，單層 LSTM 或簡單的 CNN+LSTM 僅維持在 0.9813～0.9973 左右，可能的原因在於，單層 CNN 可能無法充分捕捉文本中的關鍵局部模式，或者與 LSTM 的特徵整合效果不理想，導致模型無法充分發揮兩者的優勢。從結果也可發現即使是較為基礎的模型也能取得相當不錯的結果。不過整體而言，結果顯示更複雜的模型結構在此分類任務上具有夠高的準確率，能更進一步逼近 100%。