Finetuning LayoutLMv3 for Named Entity Recognition on the SROIE Dataset

In the previous notebook, we preprocessed the SROIE dataset to be ready to be used by the LayoutLMv3 model. In this notebook we will finetune the LayoutLMv3 model on the SROIE dataset for Named Entity Recognition (NER).

To do so, we will add a token classification head on top of the LayoutLMv3 model. The head will be initialized with random weights and then trained on the SROIE dataset. At the end of the training, the model will be able to predict whether each token in the scanned receipt stands for the following entities: `company`, `date`, `address`, `total`, or `other`.

```
%%capture
! pip install transformers
! pip install datasets
! pip install evaluate
! pip install seqeval
import warnings
warnings.filterwarnings("ignore")
import os
os.environ["TOKENIZERS_PARALLELISM"] = "false"
```

Loading the Dataset

Let's start by loading the preprocessed SROIE dataset. They are stored in the `data` folder in the `train` and `test` subfolders. For more information about the preprocessing, please refer to the `Preprocessing.ipynb` notebook.

```
from datasets import load_from_disk

train_dataset = load_from_disk("data/train")
test_dataset = load_from_disk("data/test")
```

Loading the Processor

We will also need to load the LayoutLMv3 processor.

```
from transformers import AutoProcessor

model_id = 'microsoft/layoutlmv3-base'
processor = AutoProcessor.from_pretrained(model_id)
```

Loading the Model

We will use the `AutoModelForTokenClassification` class from the `transformers` library to load the LayoutLMv3 model with a token classification head. The model head will be initialized with random weights.

```
from transformers import AutoModelForTokenClassification

labels_list = ['O', 'B-COMPANY', 'B-DATE', 'B-ADDRESS', 'B-TOTAL']
ids2labels = {k: v for k, v in enumerate(labels_list)}
labels2ids = {v: k for k, v in enumerate(labels_list)}

model = AutoModelForTokenClassification.from_pretrained(model_id,
label2id=labels2ids, id2label=ids2labels)
```

```
Some weights of LayoutLMv3ForTokenClassification were not initialized
from the model checkpoint at microsoft/layoutlmv3-base and are newly
initialized: ['classifier.bias', 'classifier.weight']
You should probably TRAIN this model on a down-stream task to be able
to use it for predictions and inference.
```

Computing Metrics

We will use the `seqeval` library to compute the precision, recall, and F1 score of the model on the test set.

```
import numpy as np
import evaluate

seqeval = evaluate.load("seqeval")

def compute_metrics(p):
  predictions, labels = p
  predictions = np.argmax(predictions, axis=2)
  true_predictions = [
    [labels_list[p] for (p, l) in zip(prediction, label) if l != -100]
    for prediction, label in zip(predictions, labels)
  ]
  true_labels = [
    [labels_list[l] for (p, l) in zip(prediction, label) if l != -100]
    for prediction, label in zip(predictions, labels)
  ]
  results = seqeval.compute(predictions=true_predictions,
references=true_labels)
  return {
      "precision": results["overall_precision"],
      "recall": results["overall_recall"],
      "f1": results["overall_f1"],
```

```
        "accuracy": results["overall_accuracy"],
    }
```

Training

First we need to define the training arguments. I will use the following arguments:

- `output_dir='checkpoints'`: The directory where the model checkpoints and evaluation results will be saved.
- `num_train_epochs=3`: The number of epochs to train the model.
- `eval_steps=100`: Evaluate the model every 100 steps.
- `load_best_model_at_end=True`: Load the best model at the end of training.
- `metric_for_best_model='f1'`: Use the F1 score to select the best model.
- `per_device_train_batch_size=4`: The batch size for training. We decreased it from the default value of 8 to avoid running out of memory.
- `per_device_eval_batch_size=4`: The batch size for evaluation. We decreased it from the default value of 8 to avoid running out of memory.
- `save_strategy='steps'`: Save the model checkpoints every 100 steps.
- `eval_strategy='steps'`: Evaluate the model every 100 steps.

```
from transformers import TrainingArguments

training_args = TrainingArguments(output_dir="checkpoints",
                                  num_train_epochs=3,
                                  eval_steps=100,
                                  load_best_model_at_end=True,
                                  metric_for_best_model="f1",
                                  per_device_train_batch_size=4,
                                  per_device_eval_batch_size=4,
                                  save_strategy="steps",
                                  eval_strategy="steps")
```

Now we can initialize the `Trainer` class and start the training.

```
from transformers import Trainer, default_data_collator

trainer = Trainer(
    model=model,
    args=training_args,
    train_dataset=train_dataset,
    eval_dataset=test_dataset,
    tokenizer=processor,
    data_collator=default_data_collator,
    compute_metrics=compute_metrics,
)

trainer.train()
```

```
{"model_id":"33b5aeb91f094fabbad02fc3c6c2f785","version_major":2,"version_minor":0}

{"model_id":"f43dd54a2d6747e38ddf5b494509004f","version_major":2,"version_minor":0}

{'eval_loss': 0.0747908353805542, 'eval_precision': 0.8391608391608392, 'eval_recall': 0.8120300751879699, 'eval_f1': 0.8253725640045854, 'eval_accuracy': 0.9754089539388722, 'eval_runtime': 18.1638, 'eval_samples_per_second': 18.994, 'eval_steps_per_second': 4.79, 'epoch': 0.64}

{"model_id":"239ae106242e41c4a1463a23ce6b5422","version_major":2,"version_minor":0}

{'eval_loss': 0.05888143181800842, 'eval_precision': 0.8512035010940919, 'eval_recall': 0.8774436090225564, 'eval_f1': 0.8641243983709737, 'eval_accuracy': 0.9807361170899699, 'eval_runtime': 18.3645, 'eval_samples_per_second': 18.786, 'eval_steps_per_second': 4.737, 'epoch': 1.27}

{"model_id":"e81ddf62c0d14994b5db8cf0ba26a0c4","version_major":2,"version_minor":0}

{'eval_loss': 0.049223482608795166, 'eval_precision': 0.8889716840536512, 'eval_recall': 0.8969924812030076, 'eval_f1': 0.8929640718562875, 'eval_accuracy': 0.9846642272922944, 'eval_runtime': 18.7048, 'eval_samples_per_second': 18.444, 'eval_steps_per_second': 4.651, 'epoch': 1.91}

{"model_id":"240a206e1ed84343847c851976ad981d","version_major":2,"version_minor":0}

{'eval_loss': 0.05197935178875923, 'eval_precision': 0.891449814126394, 'eval_recall': 0.9015037593984963, 'eval_f1': 0.8964485981308411, 'eval_accuracy': 0.9851485148514851, 'eval_runtime': 18.4515, 'eval_samples_per_second': 18.698, 'eval_steps_per_second': 4.715, 'epoch': 2.55}
{'train_runtime': 289.5086, 'train_samples_per_second': 6.487, 'train_steps_per_second': 1.627, 'train_loss': 0.07087611139706493, 'epoch': 3.0}

TrainOutput(global_step=471, training_loss=0.07087611139706493, metrics={'train_runtime': 289.5086, 'train_samples_per_second': 6.487, 'train_steps_per_second': 1.627, 'total_flos': 495041961535488.0, 'train_loss': 0.07087611139706493, 'epoch': 3.0})
```

On a typical run, the model should reach a F1 score of around 0.9 on the test set.

```
trainer.evaluate()
```

{"model_id":"7c4755ffa9f84671ab077c0224233078","version_major":2,"version_minor":0}


```
{'eval_loss': 0.04994415119290352,
 'eval_precision': 0.8912071535022354,
 'eval_recall': 0.8992481203007519,
 'eval_f1': 0.8952095808383232,
 'eval_accuracy': 0.9850408953938872,
 'eval_runtime': 18.4562,
 'eval_samples_per_second': 18.693,
 'eval_steps_per_second': 4.714,
 'epoch': 3.0}
```

We will also save the model and the processor to disk for later use.

```
trainer.save_model("model")
```

Inference

We will now create a class to load the model and the processor from disk and use them to make predictions on new receipts. The class will take an image as input, preprocess it, and then output the words, bounding boxes, and predicted entities.

```
import torch
from collections import Counter

class ReceiptReader:
  def __init__(self, path_to_model="model"):
    self.model = 
AutoModelForTokenClassification.from_pretrained(path_to_model)
    self.model.eval()
    self.processor = AutoProcessor.from_pretrained(path_to_model)


  def __call__(self, image):
    encodings = self.__get_encodings(image)
    words = self.__get_words(encodings)
    bboxes = encodings.bbox[0]
    logits = self.model(**encodings).logits
    predictions = torch.argmax(logits, dim=2)
    labeled_tokens = [self.model.config.id2label[t.item()] for t in 
predictions[0]]
    response_dict = self.__merge_tokens(words, bboxes, labeled_tokens)
    response_dict["bboxes"] = [self.__unnormalize_bbox(bbox, image) 
for bbox in response_dict["bboxes"]]
    return response_dict

  def __get_encodings(self, image):
    return self.processor(image, return_tensors="pt")
```

```
  def __get_words(self, encodings):
    words = [self.processor.tokenizer.decode(input_id) for input_id in
encodings.input_ids[0]]
    return words

  def __merge_tokens(self, words, bboxes, labels):
    new_words = []
    new_bboxes = []
    new_labels = []
    i = 0
    while i < len(words):
        token, bbox, label = words[i], bboxes[i], labels[i]
        j = i + 1
        while j < len(words) and self.__is_same_bbox(bbox, bboxes[j]):
            token += words[j]
            j += 1
        counter = Counter([labels[k] for k in range(i, j)])
        sorted_labels = sorted(counter, key=counter.get, reverse=True)
        if sorted_labels[0] == "O" and len(sorted_labels) > 1:
          label = sorted_labels[1]
        else:
          label = sorted_labels[0]
        new_words.append(token)
        new_bboxes.append(bbox)
        new_labels.append(label)
        i = j
    return {
        "words": new_words,
        "bboxes": new_bboxes,
        "labels": new_labels
    }

  def __is_same_bbox(self, bbox1, bbox2):
    for i in range(4):
        if abs(bbox1[i] - bbox2[i]) > 3:
            return False
    return True

  def __unnormalize_bbox(self, bbox, image):
    width, height = image.size
    return [bbox[0] * width / 1000, bbox[1] * height / 1000, bbox[2] *
width / 1000, bbox[3] * height / 1000]
```

Let's ensure the length of all the resulting lists are the same.

```
import PIL

image =
```

```
PIL.Image.open("data/unprocessed/SROIE2019/test/img/X51007339122.jpg")
.convert("RGB")
receipt_reader = ReceiptReader()
receipt_data = receipt_reader(image)
len(receipt_data["words"]), len(receipt_data["bboxes"]),
len(receipt_data["labels"])
```

```
(127, 127, 127)
```

Now we are going to use `ReceiptReader` class to create `ReceiptLabeler` class. This class will take an image as input and then visualize the bounding boxes and the predicted entities.

```
from PIL import ImageDraw
class ReceiptLabeler:
  def __init__(self, path_to_model="model"):
    self.receipt_reader = ReceiptReader(path_to_model)
    self.label_colors = {
        "O": "yellow",
        "B-COMPANY": "purple",
        "B-DATE": "green",
        "B-ADDRESS": "blue",
        "B-TOTAL": "red"
    }

  def __call__(self, image, include_others=False,
include_words=False):
    receipt_data = self.receipt_reader(image)

    labeled_image = image.copy()
    draw = ImageDraw.Draw(labeled_image)
    for word, bbox, label in zip(receipt_data['words'],
receipt_data["bboxes"], receipt_data["labels"]):
      if include_others or label != "O":
        draw.rectangle(bbox, outline=self.label_colors[label],
width=2)
        draw.text((bbox[0], bbox[1]-10), label,
fill=self.label_colors[label])
        if include_words:
          draw.text((bbox[0], bbox[3]), word,
fill=self.label_colors[label])

    return labeled_image

receipt_labeler = ReceiptLabeler()
labeled_image = receipt_labeler(image, include_words=True)
labeled_image
```

SANYO STATIONERY SHOP

NO. 31G&33G, JALAN SETIA INDAH X ,U13/X
40170 SETIA ALAM
Mobile /Whatsapps : +6012-918 7937
Tel: +603-3362 4137
GST ID No: 001531760640

TAX INVOICE

Owned By :
SANYU SUPPLY SDN BHD (1135772-K)

CASH SALES COUNTER

---

1. 2012-0029	RESTAURANT ORDER CHIT NCR 3.5"x6"		
3 X 2.9000	-	8.70	SR

Total Sales Inclusive GST @6%	8.70
Discount	0.00
Total	8.70
Round Adj	0.00
Final Total	8.70
CASH	20.00
CHANGE	11.30

GST Summary	Amount(RM)	Tax(RM)

We will also create `ReceiptInformationExtractor` class. This class will take an image as input and returns a dictionary with the extracted information. The class will select one of the labeled bounding boxes based on the stragety previously dicussed in the `Preprocessing.ipynb` notebook.

```
class ReceiptInformationExtractor:
  def __init__(self, path_to_model="model"):
    self.receipt_reader = ReceiptReader(path_to_model)

  def __call__(self, image):
    receipt_data = self.receipt_reader(image)
    response_dict = {
        "company": "",
        "date": "",
        "address": "",
        "total": ""
    }

    # Get the company having the largest bbox
    max_bbox = 0
    for word, bbox, label in zip(receipt_data['words'],
receipt_data["bboxes"], receipt_data["labels"]):
      if label == "B-COMPANY":
        bbox_size = (bbox[2] - bbox[0]) * (bbox[3] - bbox[1])
        if bbox_size > max_bbox:
          response_dict["company"] = word.strip()
          max_bbox = bbox_size

    # Get the address having the largest bbox
    max_bbox = 0
    for word, bbox, label in zip(receipt_data['words'],
receipt_data["bboxes"], receipt_data["labels"]):
      if label == "B-ADDRESS":
        bbox_size = (bbox[2] - bbox[0]) * (bbox[3] - bbox[1])
        if bbox_size > max_bbox:
          response_dict["address"] = word.strip()
          max_bbox = bbox_size

    # Get the topmost date
    min_y = float("inf")
    for word, bbox, label in zip(receipt_data['words'],
receipt_data["bboxes"], receipt_data["labels"]):
      if label == "B-DATE" and bbox[1] < min_y:
        response_dict["date"] = word.strip()
        min_y = bbox[1]

    # Get the bottommost total
    max_y = 0
    for word, bbox, label in zip(receipt_data['words'],
receipt_data["bboxes"], receipt_data["labels"]):
```

```
        if label == "B-TOTAL" and bbox[3] > max_y:
          response_dict["total"] = word.strip()
          max_y = bbox[3]

    return response_dict


receipt_info_extractor = ReceiptInformationExtractor()
receipt_info_extractor(image)
```

```
{'company': 'STATIONERY',
 'date': '14/10/2017',
 'address': '31G&33G,',
 'total': '8.70'}
```

Summary

In this notebook, we finetuned the LayoutLMv3 model on the SROIE dataset for Named Entity Recognition. We then created two classes to make predictions on new receipts: `ReceiptLabeler` and `ReceiptInformationExtractor`. The `ReceiptLabeler` class visualizes the bounding boxes and the predicted entities, while the `ReceiptInformationExtractor` class extracts the information from the receipt.