# Invoice Text Processing with LayoutLM and ONNX

## Table of Contents

* [Introduction](#introduction)
* [Approach](#approach)
* [Model Architecture](#model-architecture)
* [Training Process](#training-process)
* [Evaluation Metrics](#evaluation-metrics)
* [Optimization Techniques](#optimization-techniques)
* [Deployment Steps](#deployment-steps)
* [Performance on Client Desktop](#performance-on-client-desktop)
* [Conclusion](#conclusion)

## Introduction

This report documents the process of using the LayoutLM model for invoice text processing, including the approach, model architecture, training process, evaluation metrics, optimization techniques, deployment steps, and performance evaluation on the client desktop.

## Approach

The project leverages the LayoutLM model, which is specifically designed for token classification tasks involving structured documents such as invoices. The approach includes:

1. Preprocessing the invoice text data.
2. Tokenizing the text using the LayoutLM tokenizer.
3. Training the LayoutLM model on the tokenized data.
4. Converting the trained model to ONNX format for optimized inference.
5. Running the model for inference on invoice text.

## Model Architecture

The LayoutLM model is based on the Transformer architecture, which is suitable for capturing the contextual relationships within the text. The key components of the LayoutLM model include:

* **Embeddings**: Includes word embeddings, position embeddings, and segment embeddings.
* **Encoder**: Comprises multiple layers of self-attention mechanisms and feed-forward neural networks.
* **Classifier**: A linear layer that maps the encoded representations to the label space.

## Training Process

### Dataset

The training data consists of annotated invoice texts. Each token in the text is labeled with the appropriate category (e.g., invoice number, date, amount).

### Preprocessing

1. **Tokenization**: The invoice text is tokenized using the LayoutLM tokenizer.
2. **Label Encoding**: Labels are encoded to match the tokenized input.

### Model Training

The model is fine-tuned on the tokenized and labeled dataset using the following configuration:

python

Copy code

from transformers import LayoutLMForTokenClassification, LayoutLMTokenizer, Trainer, TrainingArguments

model\_name = "microsoft/layoutlm-base-uncased"

model = LayoutLMForTokenClassification.from\_pretrained(model\_name, num\_labels=2)

tokenizer = LayoutLMTokenizer.from\_pretrained(model\_name)

training\_args = TrainingArguments(

output\_dir='./results',

evaluation\_strategy="epoch",

learning\_rate=5e-5,

per\_device\_train\_batch\_size=4,

per\_device\_eval\_batch\_size=4,

num\_train\_epochs=3,

weight\_decay=0.01,

)

trainer = Trainer(

model=model,

args=training\_args,

train\_dataset=train\_dataset,

eval\_dataset=eval\_dataset,

)

trainer.train()

### Evaluation Metrics

The model's performance is evaluated using standard metrics for classification tasks:

* **Accuracy**: Proportion of correctly predicted labels.
* **Precision**: Ratio of true positive predictions to the total predicted positives.
* **Recall**: Ratio of true positive predictions to the total actual positives.
* **F1 Score**: Harmonic mean of precision and recall.

## Optimization Techniques

To optimize the model for better performance:

1. **Learning Rate Scheduling**: Adjust the learning rate during training to improve convergence.
2. **Weight Decay**: Regularization technique to prevent overfitting.
3. **Batch Size Adjustment**: Experimenting with different batch sizes to find the optimal configuration.

## Deployment Steps

### Model Conversion to ONNX

Convert the trained model to ONNX format for optimized inference:

python

Copy code

import torch

dummy\_input = {

'input\_ids': torch.ones(1, 512, dtype=torch.int64),

'attention\_mask': torch.ones(1, 512, dtype=torch.int64),

}

torch.onnx.export(model, (dummy\_input['input\_ids'], dummy\_input['attention\_mask']), "layoutlm.onnx", input\_names=['input\_ids', 'attention\_mask'], output\_names=['output'])

### Running the Model

To run the model for inference:

python

Copy code

import onnxruntime as ort

import numpy as np

def preprocess\_text(text, tokenizer, max\_length=512):

encoded\_input = tokenizer(text, padding='max\_length', truncation=True, max\_length=max\_length, return\_tensors='np')

return {

'input\_ids': encoded\_input['input\_ids'],

'attention\_mask': encoded\_input['attention\_mask']

}

def run\_model(text, tokenizer, onnx\_session):

inputs = preprocess\_text(text, tokenizer)

input\_feed = {

'input\_ids': inputs['input\_ids'],

'attention\_mask': inputs['attention\_mask']

}

outputs = onnx\_session.run(None, input\_feed)

return outputs

# Load the tokenizer and ONNX model

tokenizer = LayoutLMTokenizer.from\_pretrained(model\_name)

onnx\_session = ort.InferenceSession("layoutlm.onnx")

# Example usage

invoice\_text = "Sample invoice text for testing"

outputs = run\_model(invoice\_text, tokenizer, onnx\_session)

print(outputs)

## Performance on Client Desktop

### Environment

* **Hardware**: Specify the client desktop hardware specifications (CPU, GPU, RAM).
* **Software**: List the software and libraries used (Python version, ONNX runtime version, etc.).

### Inference Speed

Measure and report the inference time for processing a batch of invoices.

### Accuracy

Evaluate the model's accuracy on a sample set of invoices.

## Conclusion

The project successfully demonstrates the use of LayoutLM for invoice text processing. The model, when converted to ONNX format, provides efficient and fast inference suitable for deployment on client desktops. Further optimization and fine-tuning can improve the model's performance.