ENCO, COMPUTER ENGINEERING

ENCS5141, INTELLIGENT SYSTEMS LAB

Case Study #2

Prepared by: Yafa Naji 1200708

Instructor: Dr. Mohammad Jubran

Assistant: Eng. Hanan Awawdeh

Section: #1

Date of Submission: 6.Jan.2025

BIRZEIT

Jan – 2025

Abstract:

In this study, a deep learning model was developed to analyze handwritten character data using a Convolutional Neural Network (CNN) and a BERT model with LSTM. A dataset containing images of characters was loaded and processed, where data augmentation techniques were applied to enhance the model's performance. The data was divided into training, validation, and test sets, and the AdamW optimization algorithm was utilized along with early stopping techniques to prevent overfitting. A hyperparameter tuning process was executed to select the best settings for the model, resulting in improved accuracy. Additionally, the model was evaluated using a textual dataset, where BERT was employed for feature extraction, followed by an LSTM layer to enhance performance. Performance metrics such as accuracy, recall, and F1 score were calculated.

Contents

List of Figures

List of Tables

1. Introduction:

1.1. Motivation:

This project attempts to solve two major challenges: character identification through using CNN with pre trained ResNet18 model and text classification with the use of BERT and LSTM models using class weights modification to address class imbalance problem. The aim is to improve the performance metrics of the model by as well as model accuracy using advanced techniques like text classification fine tuning of neural network and alteration of class weight.

1.2. Background:

The first step involves handwritten character recognition with the use of EMNIST database alongside data augmentation strategies which include color variations, rotation and flipping. Once again, a ResNet18 pretrained neural network is used so as to enhance training speed and accuracy. In the second module, which deals with text classification, both BERT and LSTM are employed. The BERT model is a type of Transformer architecture model, it has been used for dealing with text data by using trained models that automatically change words to numerical values. Moreover, LSTM model is used to take care of temporal aspects of the data with the aim of enhancing the ability of the model to classify text. This enhancement is achieved by making changes to the class weights of the two classes to cater for the imbalance in the SUBJ and OBJ classes.

1.3. Objective:

The main aim of these tasks is to train high-performance classifiers using state-of-the-art deep-learning approaches. Each task has its corresponding objectives, which are:

- How to classify handwritten characters using custom CNN model and pre-trained ResNet18 model?
- How can we improve text classification accuracy by leveraging both BERT and LSTM models?

2. Procedure and Discussion:

2.1. Task 1: Data Extraction and Preprocessing

2.1.1. Dataset Preparation

The thing that I implemented was a custom character dataset class, named CharacterDataset, tailored to load and preprocess the characters dataset. The dataset consisted of images represented by their pixel values and a label for each image. I executed pixel values to check that they were in [0, 1] range and filled in the gaps for missing and infinite values. Also, I added a mapping file for the labels to their corresponding ASCII characters.

2.1.2. Data Splitting and Loading

I first employed the train_test_split library to create three subsets. 70% of the dataset was allocated for training, 15% for validation and 15% testing. After this, I created DataLoaders to also assist in loading the data in batches during the training phase.

2.1.3. Model Architecture

For recognition of the characters, I built the custom architecture of convolutional neural network (CNN). It comprised three convolutional layers equipped with batch normalization and max pooling layers, fully connected layers, and dropout layers to reduce the risks of overfitting.

2.1.4. Pre-Trained Model Integration

I modified the first convolutional layer of ResNet18 as per the requirement of the dataset and created a new output layer that corresponds to the number of classes in the dataset.

2.1.5. Training Setup

I set the loss function as Cross-Entropy Loss, best used for multi-class problems, whereas for the model, I employed the AdamW optimizer with a lower learning rate to increase the stability. Apart from this, I also made use of gradient clipping so that exploding gradients were kept in check during training.

2.1.6. Early Stopping Implementation

I incorporated an early stopping procedure to keep track of validation loss in order to prevent overfitting. If the validation loss kept on stagnating for multiple epochs, training was stopped so retraining would not be required.

2.1.7. Training and Validation

The model was trained on the training data and its performance was validated using the conventional metrics of model loss and accuracy. It is used an early stopping algorithm to check performance during training. The loss and accuracy values were logged for each epoch of train/validation.

2.1.8. Model Testing

After applying the early stopping method, the model was then tested on the new dataset, to see final performance. The loss and accuracy were computed, and the weighted accuracy, recall, and F1s for all classes.

Results and Analysis:

This Table 2.1-1 presents the best performance metrics achieved for various combinations of learning rates, batch sizes, and the number of layers during the training process. The metrics include training loss, training accuracy, validation loss, and validation accuracy.

Table 2.1-1: ResNet18 Model Training and validation Results

Learning Rate	Batch Size	Num Layers	Best Train Loss	Best Train Accuracy	Best Val Loss	Best Val Accuracy
0.001	32	1	0.6263	79.12%	0.4506	84.79%
0.001	32	2	0.4588	84.09%	0.3438	87.41%
0.001	64	1	0.5233	81.93%	0.4181	85.63%
0.001	64	2	0.4090	85.52%	0.3374	88.26%
0.001	128	1	0.4831	83.17%	0.4125	86.00%
0.001	128	2	0.3839	86.29%	0.3342	88.10%
0.0005	32	1	0.5340	82.02%	0.4273	85.42%
0.0005	32	2	0.4136	85.46%	0.3488	87.87%
0.0005	64	1	0.5068	82.56%	0.4204	85.43%
0.0005	64	2	0.3973	85.90%	0.3309	88.33%
0.0005	128	1	0.4742	83.61%	0.4132	85.95%
0.0005	128	2	0.3868	86.20%	0.3371	87.68%
0.00008	32	1	0.4556	84.49%	0.4040	86.34%
0.00008	32	2	0.3789	86.76%	0.3329	87.98%
0.00008	64	1	0.4808	84.00%	0.4202	85.55%
0.00008	64	2	0.3863	86.68%	0.3450	87.80%
0.00008	128	1	0.5060	83.36%	0.4308	85.31%
0.00008	128	2	0.4147	85.86%	0.3576	87.55%

Best Performance: The best performance was achieved with the hyperparameters lr=0.0005, batch_size=64, and num_layers=2, where the validation accuracy reached 88.33% with a validation loss of 0.3309.

Effect of Number of Layers: Increasing the number of layers from 1 to 2 generally led to improved performance, indicating that a more complex model can learn better features from the data.

Effect of Batch Size: A batch size of 64 had a positive impact on performance compared to batch sizes of 32 and 128, suggesting that this size may provide a good balance between training stability and convergence speed.

Total Training Time: 307.42 seconds

Evaluating the model on test data:

Table 2.1-2: Test Results of ResNet18

Metric	Value
Test Accuracy	88.43%
Test Loss	0.3485
Precision	0.8869
Recall	0.8843
F1-score	0.8828

The performance of the model looks promising, according to the results, as it has a test accuracy of 88.43%, signifying its capacity of classifying the data accurately. The precision, recall, and F1-score values illustrate a fair degree of concurrence between these metrics which means that the model is doing well in always classifying the classes in the right way without much underclassifying some classes or overestimating some classes.

Limitations Faced During Data Extraction, Preprocessing, and Model Training Phases:

- Hyperparameter Tuning: The tuning process was time-consuming and computationally expensive, requiring multiple training runs.
- Computational Resources: Limited GPU memory restricted batch sizes, affected training speed and model performance.

Convolutional Neural Network (CNN):

- **Strengths**:

Effective at capturing spatial hierarchies in images.

Data augmentation improved robustness and generalization.

- **Weaknesses**:

Sensitive to noise and variations in input data.

Resource-intensive training.

2.2. Task 2: Subjectivity Classification

2.2.1. Loading and Preparing the Data:

Using pandas, I imported data from train_en.tsv, removed the unnecessary solved_conflict column, and mapped text labels to numbers (SUBJ to 0, OBJ to 1) using a dictionary. Sentences and labels were stored in separate variables for easier handling.

2.2.2. Splitting the Data into Training and Validation Sets:

I divided the data into three sets: 70% for training, 15% for validation, and 15% for testing. I used the train_test_split function from the sklearn library to acco

2.2.3. Computing Class Weights:

In order to combat class imbalance within the data, class weights were computed and class turbulence was accounted for during the split. The weight of the OBJ class had to be increased, therefore, 1.2, an adjustment factor, was used to rescale the weights. The class weights were subsequently converted into tensor for efficient GPU and CPU training.

2.2.4. Setting Up the Tokenizer:

To convert text into numerical formats necessary for BERT understanding, I used the BertTokenizer that came with the pre-compiled BERT model bert-base-uncased. Additionally, the maximum length of the tokenized text was limited to 128 words.

2.2.5. Creating the Dataset and DataLoader:

I created a TextDataset class inheriting from torch.utils.data.Dataset and used BertTokenizer for sentence tokenization, saving tokenized sentences with labels. DataLoaders for training and validation were built using WeightedRandomSampler to ensure balanced sampling.

2.2.6. Designing the BERT-LSTM Model:

I designed a BERT_LSTM_Classifier model that combines the pre-trained BERT model with an LSTM (Long Short-Term Memory) layer. The outputs from BERT were passed through a bidirectional LSTM to capture context, followed by a Fully Connected layer to generate the final classification output. I added a Dropout layer to prevent overfitting.

2.2.7. Training the Model with Various Hyperparameters:

Learning_rates, batch_sizes and num_layers_list are several hyperparameters that I set. The model was trained on all these sets in a combinatorial fashion and for a number of epochs while loss and accuracy were recorded after each epoch after updating the parameters with AdamW.

2.2.8. Implementing Early Stopping:

I have supplied several stopping criteria to prevent the continuation of training if after a certain number of epochs (here it was set to 3) the validation performance did not get better than before. This step was used to reduce the overfitting problem as well as save time on training.

2.2.9. Evaluating Performance After Training:

After training, I calculated several evaluation metrics such as precision, recall, and F1-score using classification_report from sklearn. These metrics were printed along with a detailed report on the model's performance on the validation set.

2.2.10. Measuring Training Time and Memory Usage:

The overall training length was calculated by the use of time.time() alongside the recording of the maximum memory consumption (MB) with torch.cuda.max_memory_allocated.

Results and Analysis:

The Table 2.2-1 presents the results of training a model with different configurations, The training accuracy reached a whopping 98.62% while the validation accuracy stood at 85.48% when a batch size of 64 and learning rate of 0.00001 were used. It was found that Precision, Recall and F1-Score was kept in the range of 0.79 and 0.84, making the metrics plausible as there was average resource usage, and the algorithm trained between 102 to 205 seconds.

Table 2.2-1: Training Performance Across Different Configurations

LR	Batch Size	#Layers	Best Train Accuracy	Best Val Accuracy	Precision	Recall	F1	Training Time (s)	(MB)
0.000005	32	1	0.9725	0.8226	0.8242	0.8226	0.8232	166.22	4216.75
0.000005	32	2	0.9071	0.8145	0.7921	0.79032	0.7910	146.37	4327.41
0.000005	64	1	0.9483	0.8468	0.8378	0.83871	0.8373	172.07	4327.41
0.000005	64	2	0.9535	0.8387	0.8139	0.8145	0.8141	175.90	4341.66
0.00001	32	1	0.9604	0.8306	0.8132	0.8145	0.81341	114.93	4341.66
0.00001	32	2	0.9432	0.8145	0.8152	0.8145	0.8148	102.68	4341.66
0.00001	64	1	0.9862	0.83064	0.8327	0.8306	0.8268	186.46	4341.66
0.00001	64	2	0.9569	0.8548	0.84019	0.83871	0.83923	117.45	4341.66
0.000008	32	1	0.9845	0.83870	0.8378	0.83871	0.8374	143.25	4341.66
0.000008	32	2	0.9690	0.83870	0.8364	0.8145	0.8166	146.16	4341.66
0.000008	64	1	0.9518	0.8306	0.8229	0.8145	0.81615	114.93	4341.66
0.000008	64	2	0.9810	0.8467	0.81394	0.8145	0.8141	205.69	4341.66

Evaluating the model on test data:

The evaluation results indicate that the model performed well overall on the classification task. The overall classification accuracy reached approximately 75%, suggesting that the model was able to correctly classify new samples in most cases. Additionally, the precision, recall, and F1-score values were relatively close, indicating a good balance in the model's performance.

```
Test Accuracy: 0.7490, Precision: 0.7520, Recall: 0.7490, F1: 0.7490
              precision    recall  f1-score   support

        SUBJ       0.78      0.72      0.75       127
         OBJ       0.72      0.78      0.75       116

    accuracy                           0.75       243
   macro avg       0.75      0.75      0.75       243
weighted avg       0.75      0.75      0.75       243
```

Figure 2.2-1: Performance evaluation of the model on the SUBJ and OBJ classes

Limitations Faced:

- Class Imbalance: The unequal distribution of classes (SUBJ vs. OBJ) could have biased the model, although class weights were used to mitigate this issue.
- Computational Resources: Training the BERT-LSTM model was resource-intensive, which potentially led to longer training times and memory constraints.

Strengths and Weaknesses of the BERT-LSTM Classifier:

- **Strengths**:

Contextual Understanding: BERT effectively captured sentence semantics.

Sequential Processing: LSTM enhanced the model's ability to understand the flow of information.

- **Weaknesses**:

Resource Intensive: The model required significant computational power and memory.

Complexity: It necessitated longer training times and careful hyperparameter tuning.

Conclusion

In conclusion, the developed deep learning models effectively analyzed both handwritten character and textual data. The Convolutional Neural Network (CNN) achieved a validation accuracy of 88.33% and a test accuracy of 88.43%, benefiting from data augmentation and hyperparameter tuning. The BERT-LSTM model reached a best validation accuracy of 85.48% and a test accuracy of 75%, utilizing pre-trained embeddings. Early stopping and the AdamW optimization algorithm helped prevent overfitting. Evaluation metrics showed precision and recall scores around 0.839 and 0.838, respectively, indicating strong performance. These results underscore the potential of combining advanced neural network architectures for effective data analysis, paving the way for future research in similar areas.