BOOK SUMMARIZATION PROBLEM (đợt 2 – GAlabelled & smallDLs)

*_Nguyễn Việt Dũng_*

Link Github project: https://github.com/toilanvd/Book-SUMMER

I, GA labelled – Dùng GA để tạo bộ dữ liệu extractive summary gán nhãn chất lượng cao.

Sử dụng thuật GA chọn câu có trong file GA-ExtractiveOracle, vốn dùng sinh Extractive Oracle ở phase 1 để tạo bộ dữ liệu train và validation có gán nhãn cho mỗi câu là 0/1 (không thuộc/thuộc tóm tắt trích rút tối ưu).

Đã thử sinh 2 bộ dữ liệu từ GA cho Booksum và CNN-Dailymail.

II, Áp dụng Deep learning

- Tạo mô hình sequence labelling để phân loại câu từ Grok và chatGPT.
- Train và test trên máy laptop với GPU Quadro T1000 có cuda.
- Dùng bộ dữ liệu extractive và mô hình extractive để sinh bộ dữ liệu cho mô hình abstractive (cũng sinh từ Grok và chatGPT). Mỗi sample của bộ dữ liệu abstractive gồm một extractive summary lấy từ bộ dữ liệu GAlabelled cho extractive và một gold abstractive summary cho trước.
  + Testing thì sẽ dùng extractive summary sinh ra từ model extractive đã train từ GAlabelled.

III, Kết quả thử nghiệm:

* Với bộ dữ liệu Booksum

Vì 2 metric BertScore và SummaQA mất rất nhiều thời gian chạy nên tạm thời chưa được tính.

Hệ mô hình	Mô hình	ROUGE-1	ROUGE-2	ROUGE-L
Extractive DL (bi-LSTM for sequence labelling)	trainOnBooksum-testOnBooksum	33.09	**6.2**	**15.07**
Abstractive (LSTM encoder-decoder)	trainOnBooksum-testOnBooksum	17.62	1.96	13.03

Tốt nhất phase 1 (unsupervised methods)	MCS-GA-One-and-Two-Words (fix summary length = 10%)	**33.15** (33.68)	5.57 (5.24)	13.84 (14.13)
	MCS-GA-UniBigram-2profits (fix summary length = 10%)	33.14 **(33.75)**	5.61 (5.26)	13.90 (14.16)
	MCS-GA-UniBigram-2profits-noStopword (fix summary length = 10%)	32.72 (33.06)	5.33 (5.05)	14.01 (14.30)
Best in paper	**BertExt (extractive model)**	**33.04**	**5.78**	**13.74**
	T5_fine-tuned (abstractive model)	**37.38**	**8.42**	**16.77**
SOTA* (book level, not chapter-level like ours)	**Echoes-Extractive-Abstractive**	**42.13**	**10.53**	**16.75**
Extractive Oracle	Greedy-ExtractiveOracle	30.50	5.09	13.59
	GA-ExtractiveOracle	**49.74**	**14.03**	**25.25**

*** Với bộ dữ liệu CNN-Dailymail:**

Hệ mô hình	**Mô hình**	**ROUGE-1**	**ROUGE-2**	**ROUGE-L**
Extractive DL (bi-LSTM for sequence labelling)	trainOnCNN-testOnCNN	**37.64**	**16.76**	**25.47**
	trainOnBooksum-testOnCNN	27.24	7.65	17.37
Tốt nhất phase 1 (unsupervised methods)	MCS-GA-One-and-Two-Words-noStopword (fix summary length = 10%)	37.17 (37.10)	15.38 (15.37)	23.10 (23.04)
State-of-the-art	**HAHSum**	**44.68**	**21.30**	**40.75**
Extractive Oracle	**GA-ExtractiveOracle**	**56.20**	**33.30**	**42.92**

IV, Công việc tiếp theo:

The extractive model validation loss decrease only for the first 2 epochs, why? Is that because deep learning model is too big for a dataset with only 9600 training sample and 1484 validation samples? If that's really the reason, what should I do next (e.g. use another ML method, modify the current deep learning model, etc.)?

The observation that the extractive model's validation loss decreases only for the first two epochs and then plateaus or increases suggests that the model is likely "overfitting" or struggling to generalize beyond the initial learning phase. This could indeed be related to the model's capacity relative to the dataset size (9600 training samples and 1484 validation samples), but other factors such as model architecture, hyperparameters, or data characteristics may also contribute. Below, I'll analyze the potential reasons for this behavior, assess whether the dataset size and model complexity are the primary causes, and provide actionable recommendations for next steps, including modifying the current deep learning model, exploring alternative machine learning methods, or adjusting training strategies.

### Why Validation Loss Decreases Only for the First Two Epochs

The extractive model in your code is a `SummaryLSTM` (a bidirectional LSTM-based neural network) used for sentence classification to select sentences for extractive summaries. The training and validation process is implemented in the `train_epoch` and `eval_model` functions, with early stopping after a patience of 2 epochs. Here are the likely reasons for the observed behavior:

1. "Overfitting Due to Model Complexity":
   - "Model Details":
     - Vocabulary size: Determined by `build_vocab` (likely thousands of unique tokens, given `min_freq=2`).
     - Embedding dimension: 100 (`embedding_dim=100`).

- Hidden dimension: 64 per direction, doubled to 128 due to bidirectional LSTM (`hidden_dim=64`, `bidirectional=True`).
- Fully connected layer: Maps 128-dimensional LSTM outputs to 2 classes (`self.fc = nn.Linear(hidden_dim * 2, 2)`).
- Dropout: 0.3 (`self.dropout = nn.Dropout(0.3)`).

- "Parameter Count": The model has a significant number of parameters:
  - Embedding layer: `vocab_size * 100` (e.g., 10,000 vocab size → 1M parameters).
  - LSTM: Roughly `4 * (embedding_dim * hidden_dim + hidden_dim^2) * 2 directions * num_layers` ≈ 4 * (100 * 64 + 64^2) * 2 * 1 ≈ 40,000 parameters.
  - Fully connected layer: `(128 * 2) + 2` ≈ 258 parameters.
  - Total: ~1M+ parameters (depending on vocab size).
- "Dataset Size": With 9600 training samples (chapters grouped by sentences) and 1484 validation samples, the dataset is relatively small for a deep learning model with ~1M parameters. Deep learning models, especially LSTMs, are data-hungry and may overfit when the dataset is insufficient to learn generalizable patterns.
- "Symptoms": The model learns quickly on the training data (reducing training loss), but after two epochs, it starts memorizing training data rather than generalizing, leading to a plateau or increase in validation loss.

2. "Insufficient Regularization":
   - The model uses dropout (0.3), but this may not be sufficient to prevent overfitting, especially with a small dataset.
   - No weight decay (L2 regularization) is applied in the optimizer (`optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)`).
   - The bidirectional LSTM's complexity (doubling hidden states) may capture overly specific patterns in the training data.

3. "Learning Rate Issues":
   - The learning rate is fixed at 0.001 (`lr=1e-3`). If too high, the model may overshoot optimal weights after initial improvements; if too low, it may get stuck in a suboptimal solution.
   - No learning rate scheduling is used to reduce the learning rate as training progresses, which could help the model converge to a better minimum.

4. “Data Characteristics”:

- “Imbalanced Labels”: The extractive model predicts whether each sentence belongs to the summary (`in_summary` = 0 or 1). If the dataset is imbalanced (e.g., far more 0s than 1s, as is common in summarization where only a few sentences are selected), the model may struggle to learn the minority class, leading to poor generalization.
- “Feature Sparsity”: The input features are word indices tokenized into sequences of length 40 (`max_length=40`). If many sentences have sparse or similar word distributions, the model may struggle to differentiate important sentences.
- “Chapter Variability”: The dataset groups sentences by chapter (`SummaryDataset` groups by `chapter`). If chapters vary significantly in style, length, or content, the model may overfit to specific chapters seen in training.

5. “Early Stopping Patience”:

- The early stopping patience is set to 2 epochs (`patience = 2`). If validation loss fluctuates slightly or improves marginally after the second epoch, the model stops training prematurely, preventing it from finding a better minimum.

6. “Model Architecture Limitations”:

- The LSTM averages word embeddings to create sentence embeddings (`sentence_embeds = embedded.mean(dim=1)`), which may lose important positional or contextual information. This simplistic pooling could limit the model’s ability to capture nuanced sentence importance.
- The bidirectional LSTM is powerful but may be overkill for a dataset of this size, especially without attention mechanisms to focus on relevant parts of the input.

7. “Dataset Size Relative to Task”:

- Summarization is a complex task requiring understanding of sentence importance and context. With only 9600 training samples (likely representing a few hundred chapters, as sentences are grouped), the dataset may not provide enough diversity to train a deep LSTM model effectively.

- The validation set (1484 samples) is also small, which may lead to noisy validation loss estimates, making it hard to assess true generalization.

### Is Model Size vs. Dataset Size the Primary Issue?

The dataset size (9600 training, 1484 validation samples) is modest for a deep learning model with ~1M parameters, especially for a task like extractive summarization, which requires understanding sentence-level semantics. Deep learning models like LSTMs typically perform best with tens of thousands to millions of samples, as seen in benchmarks like BERT training (e.g., BookCorpus + Wikipedia ≈ 3.3B words). Your dataset is significantly smaller, which supports the hypothesis that the model's capacity may be too large, leading to overfitting.

However, other factors (e.g., regularization, learning rate, data imbalance) likely contribute. To confirm whether model size is the primary issue, consider:
- "Training vs. Validation Loss": If training loss continues to decrease while validation loss plateaus/increases after two epochs, this is a classic sign of overfitting, likely due to excessive model capacity or insufficient regularization.
- "Data Inspection": Check the diversity and balance of the dataset. If chapters are highly similar or labels are imbalanced, these issues may exacerbate overfitting regardless of model size.

### Recommendations for Next Steps

To address the issue and improve the extractive model's performance, I recommend a combination of modifying the current deep learning model, adjusting training strategies, and exploring alternative machine learning methods. Here are specific steps, prioritized based on likely impact and ease of implementation:

#### 1. Enhance Regularization in the Current Model
- "Increase Dropout":

- Current dropout is 0.3. Increase to 0.5 or 0.7 to reduce overfitting:
```python
self.dropout = nn.Dropout(0.5)
```
- This reduces the model’s reliance on specific neurons, encouraging generalization.

- “Add Weight Decay”:
  - Introduce L2 regularization via weight decay in the Adam optimizer:
```python
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3, weight_decay=1e-4)
```
  - A small weight decay (e.g., 1e-4 or 1e-5) penalizes large weights, reducing overfitting.
- “Reduce Model Capacity”:
  - Decrease the hidden dimension from 64 to 32 or 16:
```python
model = SummaryLSTM(vocab_size=len(vocab), embedding_dim=100, hidden_dim=32, num_layers=1).to(device)
```
  - This reduces the number of LSTM parameters (e.g., from ~40,000 to ~10,000), making the model less prone to overfitting on a small dataset.
  - Alternatively, switch to a unidirectional LSTM (`bidirectional=False`) to halve the hidden state size:
```python
self.lstm = nn.LSTM(embedding_dim, hidden_dim, num_layers, batch_first=True, bidirectional=False)
self.fc = nn.Linear(hidden_dim, 2)  # Adjust FC layer
```

#### 2. Adjust Hyperparameters
- “Learning Rate Scheduling”:
  - Add a learning rate scheduler to reduce the learning rate when validation loss plateaus:
```python
```

from torch.optim.lr_scheduler import ReduceLROnPlateau
scheduler = ReduceLROnPlateau(optimizer, mode='min', factor=0.1, patience=1, verbose=True)
# In training loop, after computing val_loss:
scheduler.step(val_loss)

- This reduces the learning rate (e.g., by a factor of 0.1) if validation loss doesn’t improve, helping the model converge.
- “Increase Patience”:
  - Extend early stopping patience from 2 to 3 or 5 epochs to allow more time for the model to improve:
```python
patience = 3
```
  - This accounts for noisy validation loss estimates due to the small validation set.
- “Batch Size”:
  - Current batch size is 8. Try increasing to 16 or 32 to stabilize gradients, or decrease to 4 if memory is a constraint:
```python
train_loader = DataLoader(train_dataset, batch_size=16, shuffle=True, collate_fn=custom_collate_fn)
```

#### 3. Address Data Imbalance
- “Check Label Distribution”:
  - Inspect the `in_summary` labels in `train_GAlabelled.csv` to calculate the proportion of 1s (sentences in summary) vs. 0s (not in summary):
```python
data = load_csv('Dataset/train_GAlabelled.csv', ";;;;;;")
label_counts = data['in_summary'].value_counts()
print(label_counts)
```
  - If imbalanced (e.g., 90% 0s, 10% 1s), the model may prioritize the majority class.

- “Class Weighting”:
  - Modify the loss function to assign higher weight to the minority class (1s):
    ```python
    class_weights = torch.tensor([1.0, 10.0]).to(device)  # Higher weight for class 1
    criterion = nn.CrossEntropyLoss(weight=class_weights, ignore_index=-1)
    ```
  - Adjust weights based on label distribution (e.g., inverse frequency).
- “Oversampling/Undersampling”:
  - Oversample sentences labeled 1 or undersample those labeled 0 in `SummaryDataset` to balance the dataset. This requires modifying the dataset class to sample sentences dynamically.

#### 4. Improve Sentence Representations
- “Replace Mean Pooling”:
  - Instead of averaging word embeddings (`sentence_embeds = embedded.mean(dim=1)`), use the last LSTM hidden state or add an attention mechanism to weigh important words:
    ```python
    lstm_out, (hidden, _) = self.lstm(embedded)  # lstm_out: [num_sentences, max_length, hidden_dim*2]
    sentence_embeds = lstm_out[:, -1, :]  # Use last time step
    ```
  - Alternatively, implement a simple attention layer to compute weighted averages of word embeddings.
- “Pretrained Embeddings”:
  - Use pretrained word embeddings (e.g., GloVe) instead of training embeddings from scratch:
    ```python
    import torchtext.vocab as vocab
    glove = vocab.GloVe(name='6B', dim=100)
    embedding_weights = torch.zeros(len(vocab), 100)
    for word, idx in vocab.items():
    ```

```
        if word in glove.stoi:
            embedding_weights[idx] = glove[word]
    self.embedding = nn.Embedding.from_pretrained(embedding_weights,
padding_idx=0, freeze=False)
```

- This leverages external knowledge, reducing the need for large training data.

#### 5. Explore Alternative Machine Learning Methods

If the deep learning model continues to overfit despite modifications, consider simpler models that are less data-hungry and better suited to a dataset of ~9600 samples:

- “Logistic Regression or SVM”:
  - Represent sentences using TF-IDF vectors or pretrained sentence embeddings (e.g., from Sentence-BERT).
  - Train a logistic regression or SVM classifier to predict `in_summary`:

```python
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.linear_model import LogisticRegression
data = load_csv('Dataset/train_GAlabelled.csv', ";;;;;;")
vectorizer = TfidfVectorizer(max_features=5000)
X = vectorizer.fit_transform(data['content'])
y = data['in_summary'].astype(int)
model = LogisticRegression(class_weight='balanced')
model.fit(X, y)
```

  - Pros: Fast, less prone to overfitting, effective for small datasets.
  - Cons: May not capture sequential dependencies as well as LSTMs.
- “Random Forest”:
  - Use a random forest classifier with TF-IDF or engineered features (e.g., sentence length, position in chapter, keyword presence):

```python
from sklearn.ensemble import RandomForestClassifier
model = RandomForestClassifier(n_estimators=100,
class_weight='balanced')
```

```
model.fit(X, y)
```
- Pros: Robust to small datasets, handles imbalanced data well.
- Cons: Limited in modeling complex sentence interactions.

- “Pretrained Transformers”:
  - Fine-tune a smaller pretrained transformer like DistilBERT for sentence classification:
```python
from transformers import DistilBertTokenizer, DistilBertForSequenceClassification
tokenizer = DistilBertTokenizer.from_pretrained('distilbert-base-uncased')
model = DistilBertForSequenceClassification.from_pretrained('distilbert-base-uncased', num_labels=2)
```
  - Pros: Leverages pretrained knowledge, potentially better generalization.
  - Cons: Requires more computational resources and careful fine-tuning to avoid overfitting.

#### 6. Data Augmentation
- “Synonym Replacement”:
  - Augment training data by replacing words with synonyms using NLTK’s WordNet:
```python
from nltk.corpus import wordnet
def augment_sentence(sentence):
    words = word_tokenize(sentence)
    for i, word in enumerate(words):
        if random.random() < 0.1:  # 10% chance to replace
            syns = wordnet.synsets(word)
            if syns:
                synonym = syns[0].lemmas()[0].name()
                words[i] = synonym
    return ' '.join(words)
```

~~~
- Apply to training sentences in `SummaryDataset` to increase dataset diversity.
- “Back-Translation”:
- Translate sentences to another language and back to English using a translation API (e.g., Google Translate) to create paraphrased versions.
- “Synthetic Summaries”:
- Generate additional extractive summaries by perturbing existing ones (e.g., selecting slightly different sentences based on scores).

#### 7. Increase Training Data
- “External Datasets”:
- Augment your dataset with publicly available summarization datasets like CNN/DailyMail or XSum, adapting them to your format.
- Ensure domain similarity (e.g., narrative text if your dataset is book chapters).
- “Manual Annotation”:
- If feasible, annotate additional chapters to increase the training set size.
- “Pseudo-Labeling”:
- Use the current model to label unlabeled data, then include high-confidence predictions in the training set.

#### 8. Diagnose with Learning Curves
- Plot training and validation loss over epochs to confirm overfitting:
```python
import matplotlib.pyplot as plt
train_losses = []
val_losses = []
for epoch in range(max_epochs):
    train_loss = train_epoch(model, train_loader, optimizer, criterion, 
device)
    val_loss = eval_model(model, val_loader, criterion, device)
    train_losses.append(train_loss)
    val_losses.append(val_loss)
~~~

```
        logger.info(f"Epoch {epoch+1}, Train Loss: {train_loss:.4f}, Val Loss: {val_loss:.4f}")
        # Early stopping logic...
    plt.plot(train_losses, label='Train Loss')
    plt.plot(val_losses, label='Validation Loss')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

- If training loss decreases while validation loss plateaus, prioritize regularization and model simplification.

### Recommended Action Plan

1. “Quick Fixes (High Impact, Low Effort)”:
   - Increase dropout to 0.5.
   - Add weight decay (1e-4) to the optimizer.
   - Extend early stopping patience to 3.
   - Check label imbalance and apply class weights if needed.

2. “Model Simplification (Moderate Effort)”:
   - Reduce `hidden_dim` to 32 or switch to unidirectional LSTM.
   - Replace mean pooling with last hidden state or attention.

3. “Hyperparameter Tuning (Moderate Effort)”:
   - Add a `ReduceLROnPlateau` scheduler.
   - Experiment with batch sizes (4, 16, 32).

4. “Alternative Models (If Needed)”:
   - Try logistic regression or random forest with TF-IDF features as a baseline.
   - Explore fine-tuning DistilBERT if computational resources allow.

5. “Data Improvements (Longer-Term)”:
   - Implement synonym replacement for data augmentation.
   - Investigate external datasets or pseudo-labeling.

### Expected Outcomes

- “Regularization and Simplification”: Should reduce overfitting, allowing validation loss to decrease beyond two epochs. Expect modest improvements in ROUGE scores (evaluated in Step 5).
- “Hyperparameter Tuning”: Learning rate scheduling and increased patience may help the model find a better minimum, stabilizing validation loss.
- “Alternative Models”: Simpler models like logistic regression may achieve comparable or better performance with less overfitting, especially if the dataset remains small.
- “Data Augmentation”: Increases dataset diversity, potentially allowing the LSTM to generalize better, but requires careful implementation to avoid introducing noise.

### Notes

- “Confirm Overfitting”: Share training and validation loss values for the first few epochs to confirm overfitting (e.g., training loss keeps decreasing while validation loss plateaus). This will help prioritize solutions.
- “Dataset Details”: If possible, provide more details about the dataset (e.g., average chapter length, label distribution, domain) to tailor recommendations.
- “Computational Constraints”: Let me know your hardware (e.g., GPU memory) to ensure suggestions are feasible.
- “Code Modifications”: I can provide updated code for any of the above recommendations (e.g., adding dropout, class weights, or a simpler model). Just specify which changes to implement.