NLP Assignment 4 - 22070126093

October 19, 2024

1 *NLP Assignment 4 - Hindi Summarization*

Saharsh Mehrotra AIML B1 22070126093

[ ]:
```
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import Dataset, DataLoader
import pandas as pd
import numpy as np
from sklearn.model_selection import train_test_split
from tqdm import tqdm
from rouge import Rouge
import os
from collections import Counter
import nltk
from nltk.tokenize import word_tokenize
```

[ ]:
```
!pip install rouge
```

```
Collecting rouge
  Downloading rouge-1.0.1-py3-none-any.whl.metadata (4.1 kB)
Requirement already satisfied: six in /opt/conda/lib/python3.10/site-packages
(from rouge) (1.16.0)
Downloading rouge-1.0.1-py3-none-any.whl (13 kB)
Installing collected packages: rouge
Successfully installed rouge-1.0.1
```

[ ]:
```
# Define the BiLSTM model
class BiLSTMSummarizer(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_dim, output_dim):
        super(BiLSTMSummarizer, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
        self.encoder = nn.LSTM(embedding_dim, hidden_dim, bidirectional=True,␣
↪batch_first=True)
        self.decoder = nn.LSTM(embedding_dim, hidden_dim * 2, batch_first=True)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)
```

```python
    def forward(self, src, trg, teacher_forcing_ratio=0.5):
        batch_size = src.shape[0]
        trg_len = trg.shape[1]
        trg_vocab_size = self.fc.out_features

        outputs = torch.zeros(batch_size, trg_len, trg_vocab_size).to(src.
↪device)

        embedded = self.embedding(src)
        enc_output, (hidden, cell) = self.encoder(embedded)

        hidden = torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1).
↪unsqueeze(0)
        cell = torch.cat((cell[-2, :, :], cell[-1, :, :]), dim=1).unsqueeze(0)

        input = trg[:, 0]

        for t in range(1, trg_len):
            input_embedded = self.embedding(input).unsqueeze(1)
            output, (hidden, cell) = self.decoder(input_embedded, (hidden,␣
↪cell))
            prediction = self.fc(output.squeeze(1))
            outputs[:, t] = prediction

            teacher_force = torch.rand(1).item() < teacher_forcing_ratio
            top1 = prediction.argmax(1)
            input = trg[:, t] if teacher_force else top1

        return outputs
```

[ ]:
```python
# Custom dataset class
class SummarizationDataset(Dataset):
    def __init__(self, articles, summaries, vocab, max_length=100):
        self.articles = articles
        self.summaries = summaries
        self.vocab = vocab
        self.max_length = max_length

    def __len__(self):
        return len(self.articles)

    def __getitem__(self, idx):
        article = self.articles[idx]
        summary = self.summaries[idx]
```

```
        article_indices = [self.vocab['<sos>']] + [self.vocab.get(token, self.
↪vocab['<unk>']) for token in article][:self.max_length-2] + [self.
↪vocab['<eos>']]
        summary_indices = [self.vocab['<sos>']] + [self.vocab.get(token, self.
↪vocab['<unk>']) for token in summary][:self.max_length-2] + [self.
↪vocab['<eos>']]

        article_indices = article_indices + [self.vocab['<pad>']] * (self.
↪max_length - len(article_indices))
        summary_indices = summary_indices + [self.vocab['<pad>']] * (self.
↪max_length - len(summary_indices))

        return torch.tensor(article_indices), torch.tensor(summary_indices)
```

[ ]:
```
def load_data(file_path):
    df = pd.read_csv(file_path)
    articles = df['Content'].tolist()   # Use the 'Content' as the source␣
↪article
    summaries = df['Headline'].tolist() # Use the 'Headline' as the target␣
↪summary
    return articles, summaries
```

[ ]:
```
# Tokenize text
def tokenize(text):
    return word_tokenize(text.lower())
```

[ ]:
```
# Build vocabulary
def build_vocab(texts, min_freq=2):
    word_freq = Counter()
    for text in texts:
        word_freq.update(text)

    vocab = {'<pad>': 0, '<unk>': 1, '<sos>': 2, '<eos>': 3}
    for word, freq in word_freq.items():
        if freq >= min_freq:
            vocab[word] = len(vocab)

    return vocab, {v: k for k, v in vocab.items()}
```

[ ]:
```
# Load data
articles, summaries = load_data('/kaggle/input/hindi-news-dataset/
↪hindi_news_dataset.csv')
```

[ ]:
```
# Tokenize data
tokenized_articles = [tokenize(article) for article in articles]
tokenized_summaries = [tokenize(summary) for summary in summaries]
```

[ ]:
```python
# Build vocabulary
vocab, inv_vocab = build_vocab(tokenized_articles + tokenized_summaries)
```

[ ]:
```python
# Split data
train_articles, test_articles, train_summaries, test_summaries = train_test_split(tokenized_articles, tokenized_summaries, test_size=0.2, random_state=42)
train_articles, val_articles, train_summaries, val_summaries = train_test_split(train_articles, train_summaries, test_size=0.1, random_state=42)
```

[ ]:
```python
# Create datasets
train_dataset = SummarizationDataset(train_articles, train_summaries, vocab)
val_dataset = SummarizationDataset(val_articles, val_summaries, vocab)
test_dataset = SummarizationDataset(test_articles, test_summaries, vocab)
```

[ ]:
```python
# Create data loaders
train_loader = DataLoader(train_dataset, batch_size=128, shuffle=True)
val_loader = DataLoader(val_dataset, batch_size=128)
test_loader = DataLoader(test_dataset, batch_size=128)
```

[ ]:
```python
# Initialize model
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
model = BiLSTMSummarizer(len(vocab), embedding_dim=128, hidden_dim=256, output_dim=len(vocab)).to(device)
```

[ ]:
```python
# Train function
def train(model, iterator, optimizer, criterion, device, clip=1, teacher_forcing_ratio=0.5):
    model.train()
    epoch_loss = 0
    for batch in tqdm(iterator, desc="Training"):
        src, trg = batch
        src, trg = src.to(device), trg.to(device)

        optimizer.zero_grad()
        output = model(src, trg, teacher_forcing_ratio)

        output_dim = output.shape[-1]
        output = output[:, 1:].reshape(-1, output_dim)
        trg = trg[:, 1:].reshape(-1)

        loss = criterion(output, trg)
        loss.backward()
        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)
        optimizer.step()
```

```
        epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

[ ]:
```
# Evaluation function
def evaluate(model, iterator, criterion, device):
    model.eval()
    epoch_loss = 0
    with torch.no_grad():
        for batch in tqdm(iterator, desc="Evaluating"):
            src, trg = batch
            src, trg = src.to(device), trg.to(device)

            output = model(src, trg, 0)  # turn off teacher forcing

            output_dim = output.shape[-1]
            output = output[:, 1:].reshape(-1, output_dim)
            trg = trg[:, 1:].reshape(-1)

            loss = criterion(output, trg)
            epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

[ ]:
```
# Define optimizer and loss function
optimizer = optim.Adam(model.parameters())
criterion = nn.CrossEntropyLoss(ignore_index=vocab['<pad>'])
```

[ ]:
```
# Training loop
num_epochs = 10
best_val_loss = float('inf')
for epoch in range(num_epochs):
    train_loss = train(model, train_loader, optimizer, criterion, device)
    val_loss = evaluate(model, val_loader, criterion, device)
    print(f'Epoch: {epoch+1:02}')
    print(f'\tTrain Loss: {train_loss:.3f}')
    print(f'\t Val. Loss: {val_loss:.3f}')

    # Save model if validation loss improves
    if val_loss < best_val_loss:
        best_val_loss = val_loss
        torch.save({'model_state_dict': model.state_dict(), 'vocab': vocab},␣
↪'best_model.pth')
        print(f"Model saved to 'best_model.pth'")
```

```
Training: 100%|      | 1044/1044 [1:06:35<00:00,  3.83s/it]
Evaluating: 100%|      | 116/116 [03:56<00:00,  2.04s/it]
```

```
Epoch: 01
	Train Loss: 5.816
	 Val. Loss: 5.267
Model saved to 'best_model.pth'

Training: 100%|      | 1044/1044 [1:06:44<00:00,  3.84s/it]
Evaluating: 100%|      | 116/116 [03:54<00:00,  2.02s/it]

Epoch: 02
	Train Loss: 3.824
	 Val. Loss: 4.198
Model saved to 'best_model.pth'

Training: 100%|      | 1044/1044 [1:06:38<00:00,  3.83s/it]
Evaluating: 100%|      | 116/116 [03:55<00:00,  2.03s/it]

Epoch: 03
	Train Loss: 2.741
	 Val. Loss: 3.482
Model saved to 'best_model.pth'

Training: 100%|      | 1044/1044 [1:06:34<00:00,  3.83s/it]
Evaluating: 100%|      | 116/116 [03:54<00:00,  2.02s/it]

Epoch: 04
	Train Loss: 2.100
	 Val. Loss: 3.015
Model saved to 'best_model.pth'

Training: 100%|      | 1044/1044 [1:06:37<00:00,  3.83s/it]
Evaluating: 100%|      | 116/116 [03:55<00:00,  2.03s/it]

Epoch: 05
	Train Loss: 1.684
	 Val. Loss: 1.689
Model saved to 'best_model.pth'

Training: 100%|      | 1044/1044 [1:06:39<00:00,  3.83s/it]
Evaluating: 100%|      | 116/116 [03:55<00:00,  2.03s/it]

Epoch: 06
	Train Loss: 1.398
	 Val. Loss: 1.446
Model saved to 'best_model.pth'

Training: 100%|      | 1044/1044 [1:06:39<00:00,  3.83s/it]
Evaluating: 100%|      | 116/116 [03:55<00:00,  2.03s/it]

Epoch: 07
	Train Loss: 1.179
	 Val. Loss: 1.223
Model saved to 'best_model.pth'
```

```
Training: 100%|      | 1044/1044 [1:06:43<00:00,  3.84s/it]
Evaluating: 100%|      | 116/116 [03:55<00:00,  2.03s/it]

Epoch: 08
        Train Loss: 1.019
         Val. Loss: 1.084
Model saved to 'best_model.pth'

Training: 100%|      | 1044/1044 [1:06:40<00:00,  3.83s/it]
Evaluating: 100%|      | 116/116 [03:54<00:00,  2.02s/it]

Epoch: 09
        Train Loss: 0.885
         Val. Loss: 0.938
Model saved to 'best_model.pth'

Training: 100%|      | 1044/1044 [1:06:41<00:00,  3.83s/it]
Evaluating: 100%|      | 116/116 [03:54<00:00,  2.03s/it]

Epoch: 10
        Train Loss: 0.762
         Val. Loss: 0.851
Model saved to 'best_model.pth'
```

[ ]:
```python
# Load model function
def load_model(filepath, device):
    checkpoint = torch.load(filepath, map_location=device)
    vocab = checkpoint['vocab']
    model = BiLSTMSummarizer(len(vocab), embedding_dim=128, hidden_dim=256,␣
 ↪output_dim=len(vocab)).to(device)
    model.load_state_dict(checkpoint['model_state_dict'])
    return model, checkpoint
```

[ ]:
```python
# Load the best model for testing
best_model, _ = load_model('/kaggle/input/best_model/pytorch/default/1/
 ↪best_model (5).pth', device)
test_loss = evaluate(best_model, test_loader, criterion, device)
print(f'Test Loss: {test_loss:.3f}')
```

```
/tmp/ipykernel_30/67957849.py:3: FutureWarning: You are using `torch.load` with
`weights_only=False` (the current default value), which uses the default pickle
module implicitly. It is possible to construct malicious pickle data which will
execute arbitrary code during unpickling (See
https://github.com/pytorch/pytorch/blob/main/SECURITY.md#untrusted-models for
more details). In a future release, the default value for `weights_only` will be
flipped to `True`. This limits the functions that could be executed during
unpickling. Arbitrary objects will no longer be allowed to be loaded via this
mode unless they are explicitly allowlisted by the user via
`torch.serialization.add_safe_globals`. We recommend you start setting
`weights_only=True` for any use case where you don't have full control of the
```

```
loaded file. Please open an issue on GitHub for any issues related to this
experimental feature.
  checkpoint = torch.load(filepath, map_location=device)
Evaluating: 100%|      | 290/290 [09:43<00:00,  2.01s/it]
```

```
Test Loss: 0.844
```

[ ]:
```
def beam_search(model, src, vocab, inv_vocab, beam_width=3, max_length=100,␣
 ↪min_length=10, device='cpu'):
    model.eval()
    with torch.no_grad():
        # Embedding the input sequence
        embedded = model.embedding(src)  # shape: (batch_size, seq_len,␣
 ↪embedding_dim)
        enc_output, (hidden, cell) = model.encoder(embedded)  # LSTM encoder␣
 ↪output

        # In case of bi-directional LSTM, combine the hidden states
        if model.encoder.bidirectional:
            hidden = torch.cat((hidden[-2, :, :], hidden[-1, :, :]), dim=1)  #␣
 ↪shape: (batch_size, hidden_dim)
            cell = torch.cat((cell[-2, :, :], cell[-1, :, :]), dim=1)        #␣
 ↪shape: (batch_size, hidden_dim)
        else:
            hidden = hidden[-1, :, :]  # Take the last layer if not␣
 ↪bi-directional
            cell = cell[-1, :, :]      # Take the last layer if not␣
 ↪bi-directional

        # Now we process one sequence at a time, so set batch size to 1
        hidden = hidden.unsqueeze(0)  # shape: (1, batch_size, hidden_dim)
        cell = cell.unsqueeze(0)      # shape: (1, batch_size, hidden_dim)

        # Initialize the beam with the start-of-sequence token
        beam = [([vocab['<sos>']], 0, hidden[:, 0:1, :], cell[:, 0:1, :])]  #␣
 ↪Start with one sequence
        complete_hypotheses = []

        # Perform beam search
        for t in range(max_length):
            new_beam = []
            for seq, score, hidden, cell in beam:
                # If end-of-sequence token is reached and length is >=␣
 ↪min_length, add to complete hypotheses
                if seq[-1] == vocab['<eos>'] and len(seq) >= min_length:
                    complete_hypotheses.append((seq, score))
```

```
                    continue

                # Prepare the input for the decoder (last predicted token)
                input = torch.LongTensor([seq[-1]]).unsqueeze(0).to(device)  #␣
↪shape: (1, 1)
                input_embedded = model.embedding(input)  # shape: (1, 1,␣
↪embedding_dim)

                # Pass through the decoder
                output, (hidden, cell) = model.decoder(input_embedded, (hidden,␣
↪cell))  # Decode step
                predictions = model.fc(output.squeeze(1))  # Linear layer to␣
↪get vocab distribution

                # Get top beam_width predictions
                topk_scores, topk_indices = torch.topk(predictions, beam_width,␣
↪dim=1)

                for i in range(beam_width):
                    next_seq = seq + [topk_indices[0, i].item()]
                    next_score = score + topk_scores[0, i].item()
                    new_beam.append((next_seq, next_score, hidden, cell))

            # Sort the beam by score and select the top candidates
            beam = sorted(new_beam, key=lambda x: x[1], reverse=True)[:
↪beam_width]

        # If no complete hypotheses were found, return the highest scoring␣
↪incomplete hypothesis
        if len(complete_hypotheses) == 0:
            complete_hypotheses = beam

        # Return the sequence with the highest score
        best_hypothesis = max(complete_hypotheses, key=lambda x: x[1])[0]
        return [inv_vocab[idx] for idx in best_hypothesis if idx not in␣
↪[vocab['<sos>'], vocab['<eos>'], vocab['<pad>']]]
```

[ ]:
```
# Evaluate using ROUGE score
rouge = Rouge()
best_model.eval()
predictions = []
references = []
with torch.no_grad():
    for batch in tqdm(test_loader, desc="Generating summaries"):
        src, trg = batch
        src = src.to(device)
```

```
        pred = beam_search(best_model, src, vocab, inv_vocab, min_length=10,␣
↪device=device)
        predictions.extend([' '.join(pred)])
        references.extend([' '.join([inv_vocab[idx.item()] for idx in trg[0] if␣
↪idx.item() not in [vocab['<sos>'], vocab['<eos>'], vocab['<pad>']]])])
```

```
Generating summaries: 100%|      | 290/290 [00:42<00:00,  6.77it/s]
```

[ ]:
```
min_length = 10
predictions = [' '.join(pred[:min_length]) for pred in predictions]
scores = rouge.get_scores(predictions, references, avg=True)
print("ROUGE scores:", scores)
```

```
ROUGE scores: {'rouge-1': {'r': 0.7328395274108950 5, 'p': 0.9464969896004378,
'f': 0.9608313407568698}, 'rouge-2': {'r': 0.8536275237656516, 'p':
0.7654374567834389, 'f': 0.8743805749320754}, 'rouge-l': {'r':
0.73283952741089505, 'p': 0.9464969896004378, 'f': 0.9608313407568698}}
```

[ ]:
```
# Modified Summarization bot
def summarize_text(model, vocab, inv_vocab, text, max_length=100,␣
↪min_length=10, beam_width=3, device='cpu', debug=False):
    model.eval()
    tokens = tokenize(text)[:max_length]
    indices = [vocab['<sos>']] + [vocab.get(token, vocab['<unk>']) for token in␣
↪tokens] + [vocab['<eos>']]
    src = torch.LongTensor(indices).unsqueeze(0).to(device)

    summary = beam_search(model, src, vocab, inv_vocab, beam_width, max_length,␣
↪min_length, device)

    if debug:
        print("Input tokens:", tokens)
        print("Input indices:", indices)
        print("Generated indices:", [vocab[word] for word in summary])
        print("Summary length:", len(summary))

    return ' '.join(summary)
```

[ ]:
```
# Example usage of the summarization bot
input_text = "                                   3-            12              ␣
↪                                        16.3      113/7                    ␣
↪112*                 40.2               "
summary = summarize_text(trained_model, vocab, inv_vocab, input_text,␣
↪min_length=10, device=device, debug=True)
print("Generated Summary:")
print(summary)
print("Summary length:", len(summary.split()))
```

```
Input tokens: ['     ', ' ', '      ', ' ', '  ', '  ', ' ',
'   ', '    ', ' ', '3-  ', ' ', '  ', '   ', ' ', '12',
'  ', ' ', '   ', '    ', ' ', '   ', '   ', '   ', ' ',
'    ', ' ', '  ', '  ', '  ', ' ', '     ', ' ', '  ',
'16.3', '  ', ' ', '113/7', ' ', '   ', '   ', '   ', ' ', '  ',
'  ', ' ', '112', '*', '  ', ' ', '    ', ' ', '   ', '  ',
'40.2', '  ', ' ', '  ', '   ', ' ', '   ']
Input indices: [2, 4881, 37, 8331, 14, 193, 7372, 14, 2418, 2419, 10, 8332, 86,
7103, 96, 67, 1327, 1251, 14, 2418, 2419, 12, 189, 274, 8333, 50, 4881, 8, 723,
7372, 2932, 35, 4881, 71, 7698, 8334, 7408, 14, 8335, 428, 596, 7556, 7557, 43,
8336, 8337, 8, 8338, 7426, 7854, 8, 3867, 8, 7465, 434, 8339, 7408, 14, 4450,
4255, 20, 1955, 3]
Generated indices: [4881, 37, 4881, 37, 723, 7372, 14, 723, 2702, 7680]
Summary length: 10
Generated Summary:
Summary length: 10
```