DIFFERENT APPROACHES FOR SENTIMENT ANALYSIS ON THE IMDB REVIEW DATASET

BARON SCHITKA

INTRODUCTION

- **Overview:** My project focuses on sentiment analysis using deep learning techniques applied to the IMDB movie reviews dataset.
- **Objective:** Develop models to analyze sentiments (positive/negative) expressed in movie reviews.
- **Learning Problem:** A comparative study of different machine learning approaches. I will be comparing RNN, LSTM and CNN on a common dataset. That dataset is an IMDB review dataset.

DATASET

- **Source:** IMDB Movie Reviews Dataset
- **Size:** 25000
- **Split:** Train and test sets

```
# Load dataset
train_iter, test_iter = torchtext.datasets.IMDB(
    root='/home/jovyan/public/datasets/IMDB/',
    split=('train', 'test')
)
```

APPROACHES

- **Models:** Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM), Convolutional Neural Network (CNN)
- **Key Differences:**
  - **RNN:** Simple architecture with a recurrent layer.
  - **LSTM:** Utilizes Long Short-Term Memory cells for improved memory handling.
  - **CNN:** Convolutional layers capture local patterns and relationships.

PREPROCESSING

- Tokenization: Basic English tokenizer used.
- Vocabulary: Minimum frequency set at 5, with special tokens like <unk>, <s>, <eos>.
- Sequence Length: Padded sequences to a maximum length of 250.

```python
# Data loading and preprocessing
def load_dataframe(iterator):
    data = list(iter(iterator))
    df = pd.DataFrame(data, columns=['sentiment', 'review'])
    df['sentiment'] = df['sentiment'] - 1
    return df

tokenizer = get_tokenizer('basic_english')

def iterate_tokens(df):
    for review in tqdm(df['review']):
        yield tokenizer(review)

df_train = load_dataframe(train_iter)

vocab = build_vocab_from_iterator(
    iterate_tokens(df_train),
    min_freq=5,
    specials=['<unk>', '<s>', '<eos>']
)

vocab.set_default_index(0)

sequences = [torch.tensor(vocab.lookup_indices(tokenizer(review)), dtype=torch.int64) for review in df_train['review']]
padded_sequences = pad_sequence(sequences, batch_first=True)[:, :250]
sentiments = torch.tensor(df_train['sentiment'], dtype=torch.int64)

dataset = TensorDataset(padded_sequences, sentiments)
(train_dataset, val_dataset) = random_split(dataset, (0.7, 0.3))

batch_size = 32
train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=batch_size)
val_dataloader = DataLoader(val_dataset, shuffle=True, batch_size=batch_size)
```

MODEL ARCHITECTURES

1. **RNN:**
    1. Embedding Layer
    2. Single RNN Layer
    3. Linear Output Layer
2. **LSTM:**
    1. Embedding Layer
    2. Single LSTM Layer
    3. Linear Output Layer
3. **CNN:**
    1. Embedding Layer
    2. Multiple Convolutional Layers
    3. Linear Output Layer

```python
# Model definition with RNN
class MySequenceClassifierRNN(LightningModule):
    def __init__(self, vocab_size, dim_emb, dim_state):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, dim_emb)
        self.rnn = nn.RNN(input_size=dim_emb, hidden_size=dim_state, num_layers=1, batch_first=True)
        self.output = nn.Linear(dim_state, 2)
        self.accuracy = Accuracy(task='multiclass', num_classes=2)

    def forward(self, sequence_batch):
        emb = self.embedding(sequence_batch)
        _, h_n = self.rnn(emb)
        output = self.output(h_n)
        return output.squeeze(0)

    def loss(self, outputs, targets):
        return F.cross_entropy(outputs, targets)

    def training_step(self, batch, batch_index):
        inputs, targets = batch
        outputs = self.forward(inputs)
        loss = self.loss(outputs, targets)
        self.accuracy(outputs, targets)
        self.log('acc', self.accuracy, prog_bar=True)
        self.log('loss', loss)
        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters())

    def validation_step(self, batch, batch_index):
        inputs, targets = batch
        outputs = self.forward(inputs)
        loss = self.loss(outputs, targets)
        val_acc = self.accuracy(outputs, targets)
        self.log('val_acc', val_acc, prog_bar=True)
        return {"val_loss": loss, "val_acc": val_acc}
```

```python
# Model definition with LSTM
class MySequenceClassifierLSTM(LightningModule):
    def __init__(self, vocab_size, dim_emb, dim_state):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, dim_emb)
        self.rnn = nn.LSTM(input_size=dim_emb, hidden_size=dim_state, num_layers=1, batch_first=True)
        self.output = nn.Linear(dim_state, 2)
        self.accuracy = Accuracy(task='multiclass', num_classes=2)

    def forward(self, sequence_batch):
        emb = self.embedding(sequence_batch)
        _, (h_n, _) = self.rnn(emb)
        output = self.output(h_n[-1])
        return output.squeeze(0)

    def loss(self, outputs, targets):
        return F.cross_entropy(outputs, targets)

    def training_step(self, batch, batch_index):
        inputs, targets = batch
        outputs = self.forward(inputs)
        loss = self.loss(outputs, targets)
        self.accuracy(outputs, targets)
        self.log('acc', self.accuracy, prog_bar=True)
        self.log('loss', loss)
        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters())

    def validation_step(self, batch, batch_index):
        inputs, targets = batch
        outputs = self.forward(inputs)
        loss = self.loss(outputs, targets)
        val_acc = self.accuracy(outputs, targets)
        self.log('val_acc', val_acc, prog_bar=True)
        return {"val_loss": loss, "val_acc": val_acc}
```

```python
# Model definition with CNN
class MySequenceClassifierCNN(LightningModule):
    def __init__(self, vocab_size, dim_emb, num_filters, filter_sizes, dim_state):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, dim_emb)
        self.convs = nn.ModuleList([
            nn.Conv1d(in_channels=dim_emb, out_channels=num_filters, kernel_size=fs)
            for fs in filter_sizes
        ])
        self.fc = nn.Linear(len(filter_sizes) * num_filters, dim_state)
        self.output = nn.Linear(dim_state, 2)
        self.accuracy = Accuracy(task='multiclass', num_classes=2)

    def forward(self, sequence_batch):
        emb = self.embedding(sequence_batch)
        emb = emb.permute(0, 2, 1)  # Adjust for 1D convolution
        conv_outs = [F.relu(conv(emb)) for conv in self.convs]
        pooled_outs = [F.max_pool1d(conv_out, conv_out.shape[2]).squeeze(2) for conv_out in conv_outs]
        cat_out = torch.cat(pooled_outs, dim=1)
        fc_out = F.relu(self.fc(cat_out))
        output = self.output(fc_out)
        return output.squeeze(0)

    def loss(self, outputs, targets):
        return F.cross_entropy(outputs, targets)

    def training_step(self, batch, batch_index):
        inputs, targets = batch
        outputs = self.forward(inputs)
        loss = self.loss(outputs, targets)
        self.accuracy(outputs, targets)
        self.log('acc', self.accuracy, prog_bar=True)
        self.log('loss', loss)
        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters())

    def validation_step(self, batch, batch_index):
        inputs, targets = batch
        outputs = self.forward(inputs)
        loss = self.loss(outputs, targets)
        val_acc = self.accuracy(outputs, targets)
        self.log('val_acc', val_acc, prog_bar=True)
        return {"val_loss": loss, "val_acc": val_acc}
```

TRAINING

- **Optimizer:** Adam optimizer
- **Loss Function:** Cross-Entropy Loss
- **Metrics:** Training and validation accuracy tracked

```python
# Training RNN
logger_rnn = CSVLogger('./lightning_logs/rnn_logs/', name='rnn_logs')
trainer_rnn = Trainer(max_epochs=10, logger=logger_rnn)
model_rnn = MySequenceClassifierRNN(vocab_size=len(vocab), dim_emb=32, dim_state=64)
trainer_rnn.fit(model_rnn, train_dataloader, val_dataloader)

# Training LSTM
logger_lstm = CSVLogger('./lightning_logs/lstm_logs/', name='lstm_logs')
trainer_lstm = Trainer(max_epochs=10, logger=logger_lstm)
model_lstm = MySequenceClassifierLSTM(vocab_size=len(vocab), dim_emb=32, dim_state=64)
trainer_lstm.fit(model_lstm, train_dataloader, val_dataloader)

# Training CNN
logger_cnn = CSVLogger('./lightning_logs/cnn_logs/', name='cnn_logs')
trainer_cnn = Trainer(max_epochs=10, logger=logger_cnn)
model_cnn = MySequenceClassifierCNN(vocab_size=len(vocab), dim_emb=32, num_filters=64, filter_sizes=[3, 4, 5], dim_state=64)
trainer_cnn.fit(model_cnn, train_dataloader, val_dataloader)
```

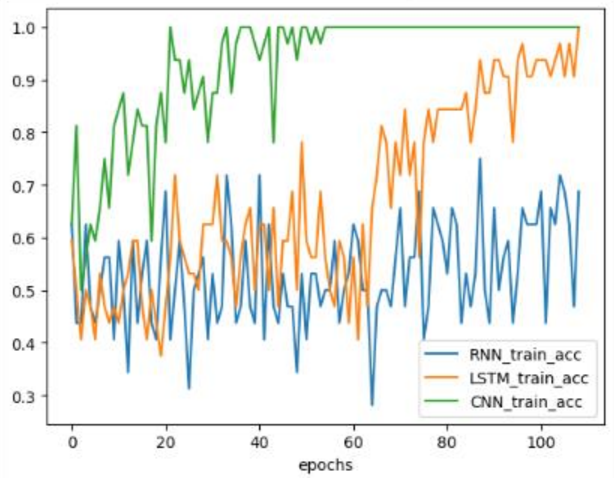
TRAINING RESULTS

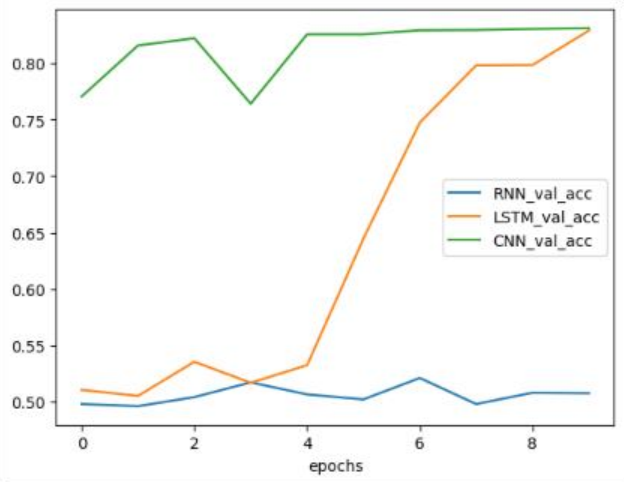
- **Graphs:** Display training and validation accuracy over epochs for RNN, LSTM, and CNN.
- **Comparison:** Highlight differences in performance.

```python
# Comparing RNN, LSTM, and CNN
metrics_rnn_path = './lightning_logs/rnn_logs/rnn_logs/version_1/metrics.csv'
metrics_lstm_path = './lightning_logs/lstm_logs/lstm_logs/version_1/metrics.csv'
metrics_cnn_path = './lightning_logs/cnn_logs/cnn_logs/version_1/metrics.csv'

metrics_rnn = pd.read_csv(metrics_rnn_path)
metrics_lstm = pd.read_csv(metrics_lstm_path)
metrics_cnn = pd.read_csv(metrics_cnn_path)

# Extracting the correct column names for training accuracy
train_acc_cols_rnn = [col for col in metrics_rnn.columns if 'acc' in col]
train_acc_cols_lstm = [col for col in metrics_lstm.columns if 'acc' in col]
train_acc_cols_cnn = [col for col in metrics_cnn.columns if 'acc' in col]

if train_acc_cols_rnn:
    train_acc_col_rnn = train_acc_cols_rnn[0]
    train_acc_rnn = metrics_rnn[train_acc_col_rnn].dropna().reset_index(drop=True).to_frame()
    train_acc_rnn.index.name = 'epochs'
    train_acc_rnn.columns = ['RNN_train_acc']

    if train_acc_cols_lstm:
        train_acc_col_lstm = train_acc_cols_lstm[0]
        train_acc_lstm = metrics_lstm[train_acc_col_lstm].dropna().reset_index(drop=True).to_frame()
        train_acc_lstm.index.name = 'epochs'
        train_acc_lstm.columns = ['LSTM_train_acc']

        if train_acc_cols_cnn:
            train_acc_col_cnn = train_acc_cols_cnn[0]
            train_acc_cnn = metrics_cnn[train_acc_col_cnn].dropna().reset_index(drop=True).to_frame()
            train_acc_cnn.index.name = 'epochs'
            train_acc_cnn.columns = ['CNN_train_acc']

            # Continue with plotting training accuracy
            acc_train_all = train_acc_rnn.merge(train_acc_lstm, left_index=True, right_index=True)
            acc_train_all = acc_train_all.merge(train_acc_cnn, left_index=True, right_index=True)
            acc_train_all.plot.line()
            acc_train_all
        else:
            print("No column with 'acc' found in CNN metrics file.")
    else:
        print("No column with 'acc' found in LSTM metrics file.")
else:
    print("No column with 'acc' found in RNN metrics file.")
```

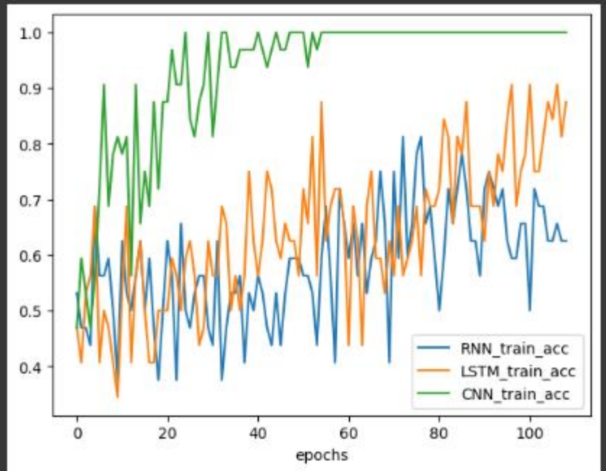
```python
# Validation accuracy plots
val_acc_cols_rnn = [col for col in metrics_rnn.columns if 'val_acc' in col]
val_acc_cols_lstm = [col for col in metrics_lstm.columns if 'val_acc' in col]
val_acc_cols_cnn = [col for col in metrics_cnn.columns if 'val_acc' in col]

# Plotting validation accuracy for RNN
if val_acc_cols_rnn:
    val_acc_col_rnn = val_acc_cols_rnn[0]
    val_acc_rnn = metrics_rnn[val_acc_col_rnn].dropna().reset_index(drop=True).to_frame()
    val_acc_rnn.index.name = 'epochs'
    val_acc_rnn.columns = ['RNN_val_acc']
    val_acc_all = val_acc_rnn
else:
    print("No column with 'val_acc' found in RNN metrics file.")

# Plotting validation accuracy for LSTM
if val_acc_cols_lstm:
    val_acc_col_lstm = val_acc_cols_lstm[0]
    val_acc_lstm = metrics_lstm[val_acc_col_lstm].dropna().reset_index(drop=True).to_frame()
    val_acc_lstm.index.name = 'epochs'
    val_acc_lstm.columns = ['LSTM_val_acc']
    val_acc_all = val_acc_all.merge(val_acc_lstm, left_index=True, right_index=True)
else:
    print("No column with 'val_acc' found in LSTM metrics file.")

# Plotting validation accuracy for CNN
if val_acc_cols_cnn:
    val_acc_col_cnn = val_acc_cols_cnn[0]
    val_acc_cnn = metrics_cnn[val_acc_col_cnn].dropna().reset_index(drop=True).to_frame()
    val_acc_cnn.index.name = 'epochs'
    val_acc_cnn.columns = ['CNN_val_acc']
    val_acc_all = val_acc_all.merge(val_acc_cnn, left_index=True, right_index=True)
else:
    print("No column with 'val_acc' found in CNN metrics file.")

# Plotting validation accuracy for all models
val_acc_all.plot.line()
val_acc_all
```

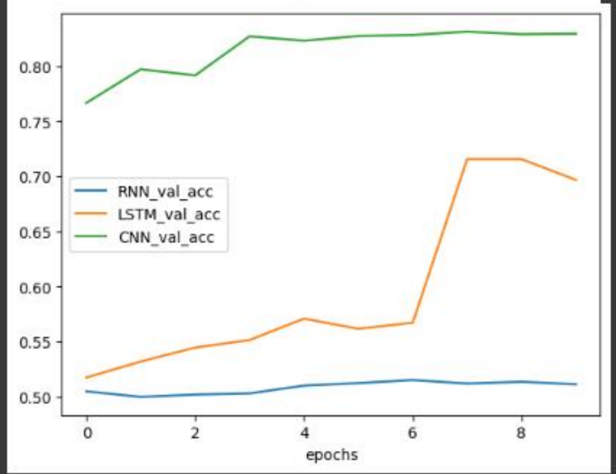
epochs			
0	0.498000	0.510533	0.770400
1	0.496133	0.505200	0.815600
2	0.504133	0.535467	0.822000
3	0.517200	0.516800	0.764000
4	0.506533	0.532400	0.825467
5	0.502133	0.644533	0.825467
6	0.521067	0.747333	0.829067
7	0.498000	0.798000	0.829333
8	0.508000	0.798267	0.830267
9	0.507600	0.828800	0.830933



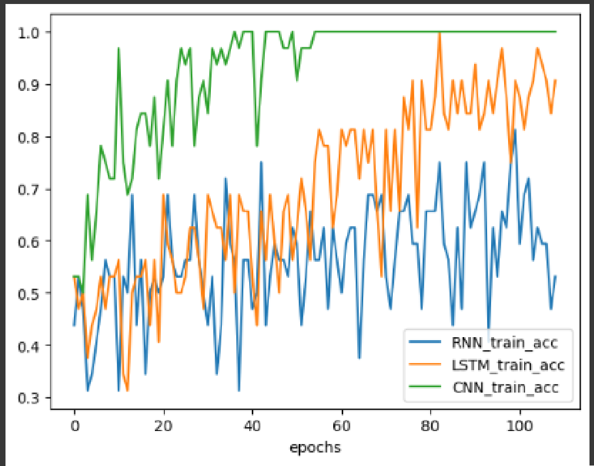


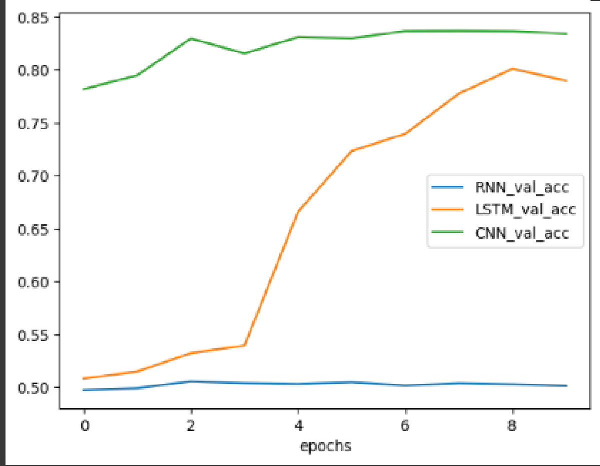
epochs			
0	0.504667	0.517200	0.766533
1	0.499600	0.531733	0.797067
2	0.501733	0.544400	0.791467
3	0.502800	0.551200	0.826933
4	0.509867	0.570667	0.822933
5	0.512133	0.561467	0.827200
6	0.514933	0.566933	0.828133
7	0.511733	0.715467	0.831200
8	0.513333	0.715467	0.828800
9	0.511067	0.696533	0.829333





epochs			
0	0.497067	0.508000	0.781733
1	0.498800	0.514667	0.794800
2	0.505333	0.532000	0.829600
3	0.503600	0.539200	0.815467
4	0.502800	0.665733	0.830933
5	0.504533	0.723200	0.829733
6	0.501333	0.739467	0.836667
7	0.503467	0.777467	0.836933
8	0.502533	0.800933	0.836533
9	0.501067	0.789733	0.834133





RESULTS

- RNN pales in comparison to LSTM and CNN. CNN seems to be better most often, but LSTM can sometimes reach it.
- LSTM can vary in accuracy in comparison to CNN.

SENTIMENT ANALYSIS EXAMPLE - CNN

- **Example 1:** "It's bad"
  - **Result:** Negative
- **Example 2:** "This movie is great"
  - **Result:** Positive

SENTIMENT ANALYSIS EXAMPLE - LSTM

- **Example 1:** "This movie is great"
  - **Result:** Positive
- **Example 2:** "This movie is bad"
  - **Result:** Negative
- **Example 3:** “Love it"
  - **Result:** Positive
- **Example 4:** “Hate it"
  - **Result:** Negative

Here is an example of the CNN model being used for sentiment analysis.

```
[ ] # Sentiment analysis on a single input using the best model (CNN)
    def sentiment_analysis(input_text, model):
        # Tokenize and preprocess the input text
        input_tokens = list(tokenizer(input_text))
        input_indices = [vocab[token] for token in input_tokens]

        # Ensure minimum sequence length for the CNN model
        filter_sizes = model.convs[0].kernel_size  # Extract filter sizes from the model
        min_seq_length = max(max(filter_sizes), 5)  # Use the maximum filter size as a reference
        input_indices.extend([vocab['<pad>']] * (min_seq_length - len(input_indices)))

        input_tensor = torch.tensor(input_indices, dtype=torch.int64).unsqueeze(0)

        # Perform sentiment analysis using the model
        model.eval()
        with torch.no_grad():
            output = model(input_tensor)

        # Get the predicted sentiment (positive/negative)
        _, predicted_class = torch.max(output, dim=-1)
        sentiment = "positive" if predicted_class.item() == 1 else "negative"

        return sentiment

    # Perform sentiment analysis and print the result
    predicted_sentiment = sentiment_analysis("It's bad", model_cnn)
    print(f"The sentiment of the review 'It's bad' is {predicted_sentiment}.")

    predicted_sentiment = sentiment_analysis("This movie is great", model_cnn)
    print(f"The sentiment of the review 'This movie is great' is {predicted_sentiment}.")
```

```
The sentiment of the review 'It's bad' is negative.
The sentiment of the review 'This movie is great' is positive.
```

Here is an example of the LSTM model being used for sentiment analysis.

```
# Sentiment analysis on a single input using the LSTM model
def sentiment_analysis_lstm(input_text, model):
    # Tokenize and preprocess the input text
    input_tokens = list(tokenizer(input_text))
    input_indices = [vocab[token] for token in input_tokens]

    # Ensure minimum sequence length for the LSTM model
    min_seq_length = 5
    input_indices.extend([vocab['<pad>']] * (min_seq_length - len(input_indices)))

    input_tensor = torch.tensor(input_indices, dtype=torch.int64).unsqueeze(0)

    # Perform sentiment analysis using the LSTM model
    model.eval()
    with torch.no_grad():
        output = model(input_tensor)

    # Get the predicted sentiment (positive/negative)
    _, predicted_class = torch.max(output, dim=-1)
    sentiment = "positive" if predicted_class.item() == 1 else "negative"

    return sentiment

# Perform sentiment analysis using the LSTM model and print the result
predicted_sentiment_lstm = sentiment_analysis_lstm("This movie is great", model_lstm)
print(f"The sentiment of the review 'This movie is great' is {predicted_sentiment_lstm}.")

predicted_sentiment_lstm = sentiment_analysis_lstm("This movie is bad", model_lstm)
print(f"The sentiment of the review 'This movie is bad' is {predicted_sentiment_lstm}.")


predicted_sentiment_lstm = sentiment_analysis_lstm("Love it", model_lstm)
print(f"The sentiment of the review 'Love it' is {predicted_sentiment_lstm}.")

predicted_sentiment_lstm = sentiment_analysis_lstm("Hate it", model_lstm)
print(f"The sentiment of the review 'Hate it' is {predicted_sentiment_lstm}.")
```

```
The sentiment of the review 'This movie is great' is positive.
The sentiment of the review 'This movie is bad' is negative.
The sentiment of the review 'Love it' is positive.
The sentiment of the review 'Hate it' is negative.
```

MODEL DEPLOYMENT

- **Saved Models:** CNN and LSTM models are saved for deployment.
- **Vocabulary:** Saved for tokenization during deployment.

For the deployment and creation of a sentiment analysis application the vocabulary and models are saved and downloaded to be used in the application.

```
[ ] # Save the vocabulary
    torch.save(vocab, '/content/vocab.pth')
```

```
[ ] # Save models
    torch.save(model_cnn.state_dict(), '/content/model_cnn.pth')
    torch.save(model_lstm.state_dict(), '/content/model_lstm.pth')
```

```
[ ] # Download the vocabulary
    from google.colab import files

    files.download('/content/vocab.pth')
```

```
[ ] # Download the models
    from google.colab import files

    files.download('/content/model_cnn.pth')
    files.download('/content/model_lstm.pth')
```

APPLICATION DEVELOPMENT

- **Technologies Used:** PyTorch, TorchText, PyTorch Lightning, Python, Tkinter.
- *I integrated my pretrained models into a GUI-based application using Tkinter.*
- *The application takes user input, analyzes sentiment using both CNN and LSTM models, and displays the results.*

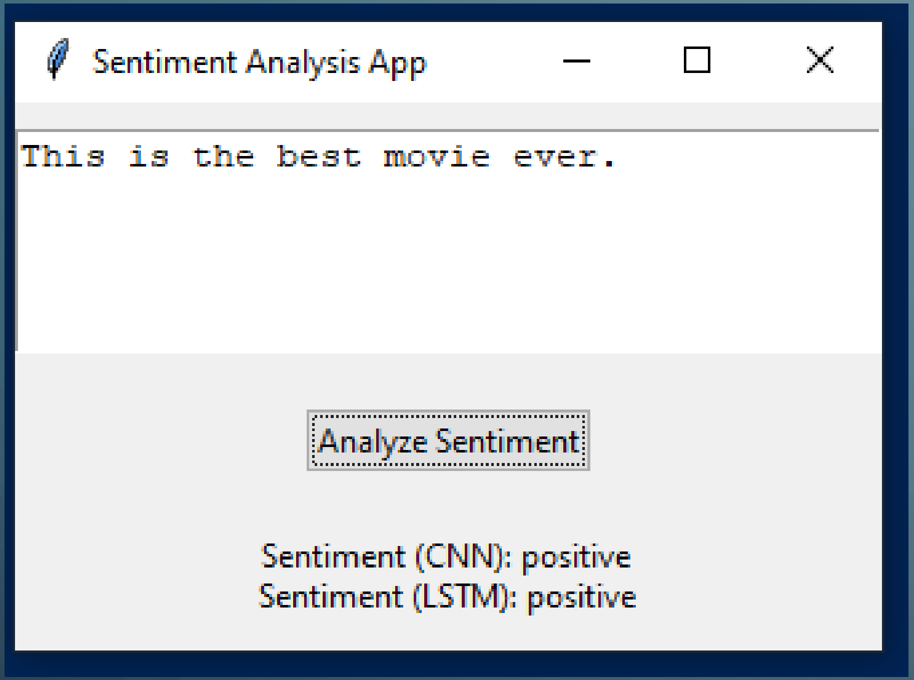
```python
import torch
from torchtext.data.utils import get_tokenizer
from torch import nn
import torch.nn.functional as F
from torchmetrics import Accuracy
from pytorch_lightning import LightningModule
import tkinter as tk
from tkinter import ttk


class MySequenceClassifierCNN(LightningModule):
    def __init__(self, vocab_size, dim_emb, num_filters, filter_sizes, dim_state):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, dim_emb)
        self.convs = nn.ModuleList([
            nn.Conv1d(in_channels=dim_emb,
                      out_channels=num_filters, kernel_size=fs)
            for fs in filter_sizes
        ])
        self.fc = nn.Linear(len(filter_sizes) * num_filters, dim_state)
        self.output = nn.Linear(dim_state, 2)
        self.accuracy = Accuracy(task='multiclass', num_classes=2)

    def forward(self, sequence_batch):
        emb = self.embedding(sequence_batch)
        emb = emb.permute(0, 2, 1)  # Adjust for 1D convolution
        conv_outs = [F.relu(conv(emb)) for conv in self.convs]
        pooled_outs = [F.max_pool1d(conv_out, conv_out.shape[2]).squeeze(
            2) for conv_out in conv_outs]
        cat_out = torch.cat(pooled_outs, dim=1)
        fc_out = F.relu(self.fc(cat_out))
        output = self.output(fc_out)
        return output.squeeze(0)

    def loss(self, outputs, targets):
        return F.cross_entropy(outputs, targets)

    def training_step(self, batch, batch_index):
        inputs, targets = batch
        outputs = self.forward(inputs)
        loss = self.loss(outputs, targets)
        self.accuracy(outputs, targets)
        self.log('acc', self.accuracy, prog_bar=True)
        self.log('loss', loss)
        return loss
```

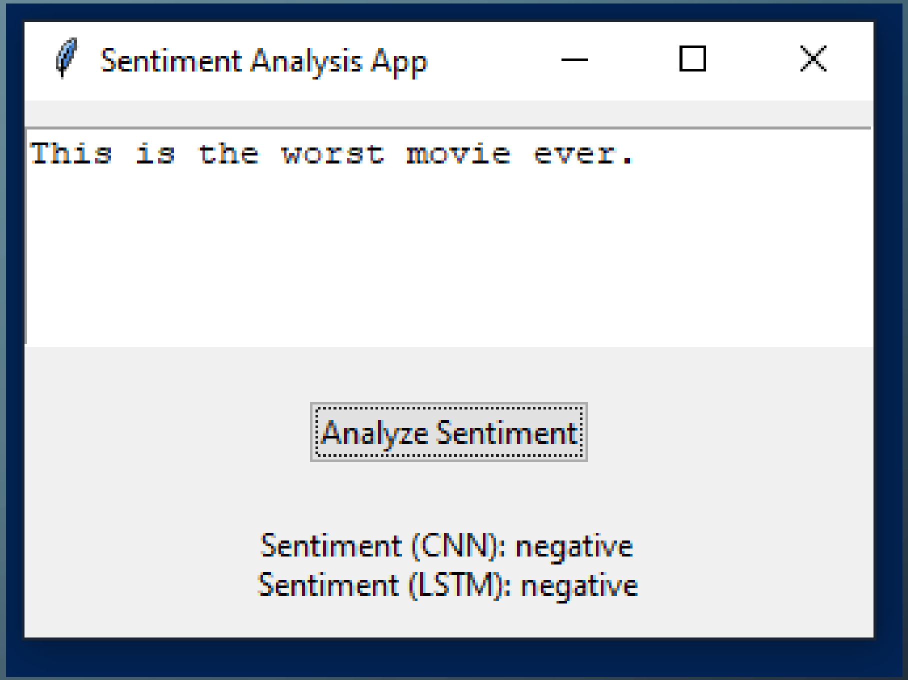
```python
    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters())

    def validation_step(self, batch, batch_index):
        inputs, targets = batch
        outputs = self.forward(inputs)
        loss = self.loss(outputs, targets)
        val_acc = self.accuracy(outputs, targets)
        self.log('val_acc', val_acc, prog_bar=True)
        return {"val_loss": loss, "val_acc": val_acc}


class MySequenceClassifierLSTM(LightningModule):
    def __init__(self, vocab_size, dim_emb, dim_state):
        super().__init__()
        self.embedding = nn.Embedding(vocab_size, dim_emb)
        self.rnn = nn.LSTM(
            input_size=dim_emb, hidden_size=dim_state, num_layers=1, batch_first=True)
        self.output = nn.Linear(dim_state, 2)
        self.accuracy = Accuracy(task='multiclass', num_classes=2)

    def forward(self, sequence_batch):
        emb = self.embedding(sequence_batch)
        _, (h_n, _) = self.rnn(emb)
        output = self.output(h_n[-1])
        return output.squeeze(0)

    def loss(self, outputs, targets):
        return F.cross_entropy(outputs, targets)

    def training_step(self, batch, batch_index):
        inputs, targets = batch
        outputs = self.forward(inputs)
        loss = self.loss(outputs, targets)
        self.accuracy(outputs, targets)
        self.log('acc', self.accuracy, prog_bar=True)
        self.log('loss', loss)
        return loss

    def configure_optimizers(self):
        return torch.optim.Adam(self.parameters())
```

```python
    def validation_step(self, batch, batch_index):
        inputs, targets = batch
        outputs = self.forward(inputs)
        loss = self.loss(outputs, targets)
        val_acc = self.accuracy(outputs, targets)
        self.log('val_acc', val_acc, prog_bar=True)
        return {"val_loss": loss, "val_acc": val_acc}


# Load vocabulary
vocab = torch.load('vocab.pth')

# Load pretrained models
model_cnn = MySequenceClassifierCNN(vocab_size=len(
    vocab), dim_emb=32, num_filters=64, filter_sizes=[3, 4, 5], dim_state=64)
model_cnn.load_state_dict(torch.load('model_cnn.pth'))
model_cnn.eval()

model_lstm = MySequenceClassifierLSTM(
    vocab_size=len(vocab), dim_emb=32, dim_state=64)
model_lstm.load_state_dict(torch.load('model_lstm_better.pth'))
model_lstm.eval()

# Tokenizer
tokenizer = get_tokenizer('basic_english')


def sentiment_analysis_cnn(input_text, model):
    input_tokens = list(tokenizer(input_text))
    input_indices = [vocab[token] for token in input_tokens]

    # Ensure minimum sequence length for the CNN model
    filter_sizes = model.convs[0].kernel_size
    min_seq_length = max(max(filter_sizes), 5)
    input_indices.extend([vocab['<pad>']] *
                         (min_seq_length - len(input_indices)))

    input_tensor = torch.tensor(input_indices, dtype=torch.int64).unsqueeze(0)

    with torch.no_grad():
        output = model(input_tensor)

    _, predicted_class = torch.max(output, dim=-1)
    sentiment = "positive" if predicted_class.item() == 1 else "negative"

    return sentiment
```

```python
def sentiment_analysis_lstm(input_text, model):
    input_tokens = list(tokenizer(input_text))
    input_indices = [vocab[token] for token in input_tokens]

    # Ensure minimum sequence length for the LSTM model
    min_seq_length = 5
    input_indices.extend([vocab['<pad>']] *
                         (min_seq_length - len(input_indices)))

    input_tensor = torch.tensor(input_indices, dtype=torch.int64).unsqueeze(0)

    with torch.no_grad():
        output = model(input_tensor)

    _, predicted_class = torch.max(output, dim=-1)
    sentiment = "positive" if predicted_class.item() == 1 else "negative"

    return sentiment


def analyze_sentiment():
    user_input = text_entry.get("1.0", tk.END).strip()

    if user_input.lower() == 'exit':
        root.destroy()
        return

    sentiment_cnn = sentiment_analysis_cnn(user_input, model_cnn)
    sentiment_lstm = sentiment_analysis_lstm(user_input, model_lstm)

    result_label.config(
        text=f"Sentiment (CNN): {sentiment_cnn}\nSentiment (LSTM): {sentiment_lstm}")
```

```python
# GUI setup
root = tk.Tk()
root.title("Sentiment Analysis App")

# Text entry
text_entry = tk.Text(root, wrap=tk.WORD, width=40, height=5)
text_entry.pack(pady=10)

# Analyze button
analyze_button = ttk.Button(
    root, text="Analyze Sentiment", command=analyze_sentiment)
analyze_button.pack(pady=10)

# Result label
result_label = tk.Label(root, text="")
result_label.pack(pady=10)

# Run the GUI
root.mainloop()
```


Sentiment Analysis App
This is the best movie ever.
Analyze Sentiment
Sentiment (CNN): positive
Sentiment (LSTM): positive


Sentiment Analysis App
This is the worst movie ever.
Analyze Sentiment
Sentiment (CNN): negative
Sentiment (LSTM): negative

CONCLUSION

- **Summary:** Developed and compared RNN, LSTM, and CNN models for sentiment analysis.
- **Achievements:** Achieved competitive accuracy on IMDB movie reviews.