



```
In [ ]: import kagglehub

# Download latest version
path = kagglehub.dataset_download("lakshmi25npathi/imdb-dataset-of-50k-movie-r

print("Path to dataset files:", path)
```

Using Colab cache for faster access to the 'imdb-dataset-of-50k-movie-reviews' dataset.

Path to dataset files: /kaggle/input/imdb-dataset-of-50k-movie-reviews

```
In [ ]: !ls /kaggle/input/imdb-dataset-of-50k-movie-reviews

'IMDB Dataset.csv'
```

```
In [ ]: import torch
import torch.nn as nn
import pandas as pd
import numpy as np
import re
from torch.utils.data import Dataset, DataLoader
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from collections import Counter
import matplotlib.pyplot as plt

# -----
# 1⬢ Device
# -----
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
print("Using device:", device)

# -----
# 2⬢ Load IMDB dataset
# -----
df = pd.read_csv('/kaggle/input/imdb-dataset-of-50k-movie-reviews/IMDB Dataset

# Encode labels
le = LabelEncoder()
df['sentiment'] = le.fit_transform(df['sentiment'])

# -----
# 3⬢ Tokenize and Vocabulary
# -----
def tokenize(text):
    return re.findall(r'\b\w+\b', text.lower())

df['tokens'] = df['review'].apply(tokenize)

all_tokens = [token for tokens in df['tokens'] for token in tokens]
vocab = {word: idx + 2 for idx, (word, _) in enumerate(Counter(all_tokens).ite
vocab['<PAD>'] = 0
vocab['<UNK>'] = 1
```

```

def encode(tokens):
    return [vocab.get(token, 1) for token in tokens]

df['encoded'] = df['tokens'].apply(encode)

# -----
# 4 ♦ Padding
# -----
max_len = 300
def pad_sequence(seq):
    return seq[:max_len] + [0]*(max_len - len(seq)) if len(seq) < max_len else seq

df['padded'] = df['encoded'].apply(pad_sequence)

# -----
# 5 ♦ Train/Test split
# -----
X = np.array(df['padded'].tolist())
y = np.array(df['sentiment'].tolist())

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

X_train = torch.tensor(X_train, dtype=torch.long).to(device)
X_test = torch.tensor(X_test, dtype=torch.long).to(device)
y_train = torch.tensor(y_train, dtype=torch.float32).to(device)
y_test = torch.tensor(y_test, dtype=torch.float32).to(device)

train_data = torch.utils.data.TensorDataset(X_train, y_train)
test_data = torch.utils.data.TensorDataset(X_test, y_test)

train_loader = DataLoader(train_data, batch_size=128, shuffle=True)
test_loader = DataLoader(test_data, batch_size=128)

# -----
# 6 ♦ LSTM Model
# -----
class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim=200, hidden_dim=256, output_dim=1, num_layers=1, batch_first=True,
                  bidirectional=False, dropout=0.5):
        super(LSTMClassifier, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, num_layers=num_layers, batch_first=batch_first,
                             bidirectional=bidirectional, dropout=dropout)
        self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)
        _, (hidden, _) = self.lstm(x)
        if self.lstm.bidirectional:
            hidden = torch.cat((hidden[-2], hidden[-1]), dim=1)
        else:
            hidden = hidden[-1]
        out = self.fc(hidden)
        return self.sigmoid(out)

```

```

        return self.sigmoid(out)

model = LSTMClassifier(vocab_size=len(vocab)).to(device)

# -----
# 7💎 Training
# -----
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patie

epochs = 6
train_losses, test_losses = [], []

for epoch in range(epochs):
    model.train()
    total_loss = 0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model(inputs).squeeze()
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    train_loss = total_loss / len(train_loader)

    model.eval()
    total_loss = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model(inputs).squeeze()
            loss = criterion(outputs, labels)
            total_loss += loss.item()
    test_loss = total_loss / len(test_loader)

    train_losses.append(train_loss)
    test_losses.append(test_loss)
    scheduler.step(test_loss)

    print(f"Epoch {epoch+1}/{epochs} | Train Loss: {train_loss:.4f} | Test Los

# -----
# 8💎 Loss Curves
# -----
plt.plot(train_losses, label='Train Loss')
plt.plot(test_losses, label='Test Loss')
plt.legend()
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Train/Test Loss')
plt.show()

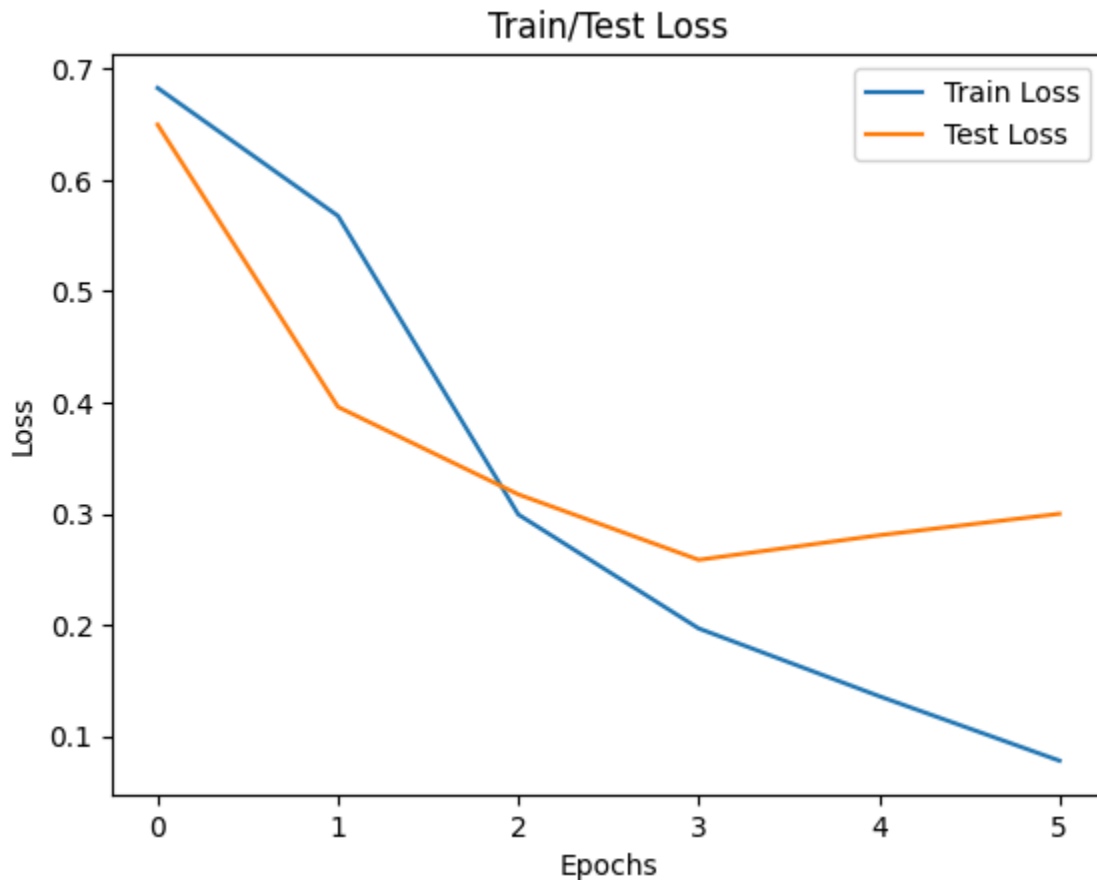
# -----

```

```
# 9 Evaluate Accuracy
# -----
model.eval()
correct, total = 0, 0
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model(inputs).squeeze()
        predicted = (outputs > 0.5).float()
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
accuracy = 100 * correct / total
print(f"Test Accuracy: {accuracy:.2f}%")
```

Using device: cuda

Epoch 1/6	Train Loss: 0.6827	Test Loss: 0.6498
Epoch 2/6	Train Loss: 0.5677	Test Loss: 0.3960
Epoch 3/6	Train Loss: 0.2993	Test Loss: 0.3174
Epoch 4/6	Train Loss: 0.1969	Test Loss: 0.2586
Epoch 5/6	Train Loss: 0.1360	Test Loss: 0.2807
Epoch 6/6	Train Loss: 0.0780	Test Loss: 0.3000



Test Accuracy: 89.77%

```
In [ ]: # 10 Prediction
def predict_sentiment(review, model, vocab, max_len, device):
    model.eval()
    tokens = tokenize(review)
    encoded = encode(tokens)
```

```

        padded = pad_sequence(encoded)
        input_tensor = torch.tensor(padded, dtype=torch.long).unsqueeze(0).to(device)
        with torch.no_grad():
            output = model(input_tensor).squeeze()
            prediction = (output > 0.5).item()
        return "Positive" if prediction == 1 else "Negative", output.item()

# Example prediction
sample_review = "This movie was absolutely amazing! I loved every part of it."
sentiment, probability = predict_sentiment(sample_review, model, vocab, max_length)
print(f"Review: \"{sample_review}\"")
print(f"Predicted Sentiment: {sentiment} (Probability: {probability:.4f})")

sample_review_2 = "This movie was terrible. I hated it."
sentiment_2, probability_2 = predict_sentiment(sample_review_2, model, vocab, max_length)
print(f"Review: \"{sample_review_2}\"")
print(f"Predicted Sentiment: {sentiment_2} (Probability: {probability_2:.4f})")

```

```

Review: "This movie was absolutely amazing! I loved every part of it."
Predicted Sentiment: Positive (Probability: 0.9903)
Review: "This movie was terrible. I hated it."
Predicted Sentiment: Negative (Probability: 0.0013)

```

```
In [ ]: model.eval()
```

```

Out[ ]: LSTMClassifier(
  (embedding): Embedding(101946, 200, padding_idx=0)
  (lstm): LSTM(200, 256, num_layers=2, batch_first=True, dropout=0.3, bidirectional=True)
  (fc): Linear(in_features=512, out_features=1, bias=True)
  (sigmoid): Sigmoid()
)

```

RNN Model

```

In [ ]: from torch.utils.data import DataLoader, TensorDataset
        from sklearn.model_selection import train_test_split
        from sklearn.preprocessing import LabelEncoder
        from collections import Counter
        import matplotlib.pyplot as plt

        le = LabelEncoder()
        df['sentiment'] = le.fit_transform(df['sentiment'])

```

```

Epoch 1/6 | Train Loss: 0.6958 | Test Loss: 0.6941
Epoch 2/6 | Train Loss: 0.6946 | Test Loss: 0.6957
Epoch 3/6 | Train Loss: 0.6965 | Test Loss: 0.6980
Epoch 4/6 | Train Loss: 0.6959 | Test Loss: 0.6976
Epoch 5/6 | Train Loss: 0.6933 | Test Loss: 0.6933
Epoch 6/6 | Train Loss: 0.6930 | Test Loss: 0.6934
RNN Test Accuracy: 51.31%

```

```

In [ ]: def tokenize(text):
        return re.findall(r'\b\w+\b', text.lower())

df['tokens'] = df['review'].apply(tokenize)
all_tokens = [token for tokens in df['tokens'] for token in tokens]
vocab_rnn = {word: idx + 2 for idx, (word, _) in enumerate(Counter(all_tokens))}
vocab_rnn['<PAD>'] = 0
vocab_rnn['<UNK>'] = 1

def encode(tokens):
    return [vocab_rnn.get(token, 1) for token in tokens]

df['encoded'] = df['tokens'].apply(encode)

# -----
# Padding and sequence length reduction
# -----
max_len = 200 # smaller sequence for speed
def pad_sequence(seq):
    return seq[:max_len] + [0]*(max_len - len(seq)) if len(seq) < max_len else seq

df['padded'] = df['encoded'].apply(pad_sequence)

X = np.array(df['padded'].tolist())
y = np.array(df['sentiment'].tolist())

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

X_train = torch.tensor(X_train, dtype=torch.long).to(device)
X_test = torch.tensor(X_test, dtype=torch.long).to(device)
y_train = torch.tensor(y_train, dtype=torch.float32).to(device)
y_test = torch.tensor(y_test, dtype=torch.float32).to(device)

train_loader = DataLoader(TensorDataset(X_train, y_train), batch_size=256, shuffle=True)
test_loader = DataLoader(TensorDataset(X_test, y_test), batch_size=256, shuffle=False)

# -----
# Efficient Vanilla RNN
# -----
class EfficientVanillaRNN(nn.Module):
    def __init__(self, vocab_size, embed_dim=200, hidden_dim=256, output_dim=1, num_layers=1,
                 batch_first=True, nonlinearity='tanh', dropout=0.5,
                 bidirectional=False):
        super(EfficientVanillaRNN, self).__init__()
        self.embedding = nn.Embedding(vocab_size, embed_dim, padding_idx=0)
        self.rnn = nn.RNN(embed_dim, hidden_dim, num_layers=num_layers,
                           batch_first=batch_first, nonlinearity=nonlinearity, dropout=dropout,
                           bidirectional=bidirectional)
        self.fc = nn.Linear(hidden_dim * 2 if bidirectional else hidden_dim, output_dim)
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        x = self.embedding(x)
        _, hidden = self.rnn(x)
        if self.rnn.bidirectional:
            hidden = torch.cat((hidden[-1, :, :], hidden[0, :, :]), dim=-1)
        else:
            hidden = hidden[-1, :, :]
        out = self.fc(hidden)
        return self.sigmoid(out)

```

```

        hidden = torch.cat((hidden[-2], hidden[-1]), dim=1)
    else:
        hidden = hidden[-1]
    out = self.fc(hidden)
    return self.sigmoid(out)

model_rnn = EfficientVanillaRNN(vocab_size=len(vocab_rnn)).to(device)

# -----
# Training
# -----
criterion = nn.BCELoss()
optimizer = torch.optim.Adam(model_rnn.parameters(), lr=0.001)
scheduler = torch.optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min', patie

epochs = 5
train_losses, test_losses = [], []

for epoch in range(epochs):
    model_rnn.train()
    total_loss = 0
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model_rnn(inputs).squeeze()
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()
        total_loss += loss.item()
    train_loss = total_loss / len(train_loader)

    model_rnn.eval()
    total_loss = 0
    with torch.no_grad():
        for inputs, labels in test_loader:
            outputs = model_rnn(inputs).squeeze()
            loss = criterion(outputs, labels)
            total_loss += loss.item()
    test_loss = total_loss / len(test_loader)

    train_losses.append(train_loss)
    test_losses.append(test_loss)
    scheduler.step(test_loss)

    print(f"Epoch {epoch+1}/{epochs} | Train Loss: {train_loss:.4f} | Test Los

# -----
# Accuracy
# -----
model_rnn.eval()
correct, total = 0, 0
with torch.no_grad():
    for inputs, labels in test_loader:
        outputs = model_rnn(inputs).squeeze()

```

```

        predicted = (outputs > 0.5).float()
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    accuracy_rnn = 100 * correct / total
    print(f" RNN Test Accuracy: {accuracy_rnn:.2f}%")

```

```

Epoch 1/5 | Train Loss: 0.6851 | Test Loss: 0.6567
Epoch 2/5 | Train Loss: 0.6454 | Test Loss: 0.6347
Epoch 3/5 | Train Loss: 0.6036 | Test Loss: 0.6925
Epoch 4/5 | Train Loss: 0.6691 | Test Loss: 0.6940
Epoch 5/5 | Train Loss: 0.6084 | Test Loss: 0.6734
RNN Test Accuracy: 61.15%

```

```

In [ ]: import torch
import re

# Function to preprocess and tokenize a new review
def preprocess_review(review, vocab, max_len=200):
    tokens = re.findall(r'\b\w+\b', review.lower())
    encoded = [vocab.get(token, 1) for token in tokens] # 1 = <UNK>
    if len(encoded) < max_len:
        encoded += [0] * (max_len - len(encoded)) # pad
    else:
        encoded = encoded[:max_len]
    return torch.tensor(encoded, dtype=torch.long).unsqueeze(0).to(device) #

# Prediction function
def predict_sentiment(model, review, vocab):
    model.eval()
    with torch.no_grad():
        input_tensor = preprocess_review(review, vocab)
        output = model(input_tensor).squeeze()
        prediction = 1 if output.item() > 0.5 else 0
    return "Positive" if prediction == 1 else "Negative"

# -----
# Example Usage
# -----
sample_review1 = "The movie was amazing, I really enjoyed it!"
sample_review2 = "It was a terrible movie, waste of time."

print("Review 1 Prediction:", predict_sentiment(model_rnn, sample_review1, vocab))
print("Review 2 Prediction:", predict_sentiment(model_rnn, sample_review2, vocab))

```

```

Review 1 Prediction: Positive
Review 2 Prediction: Negative

```

```

In [ ]: plt.figure(figsize=(8,5))

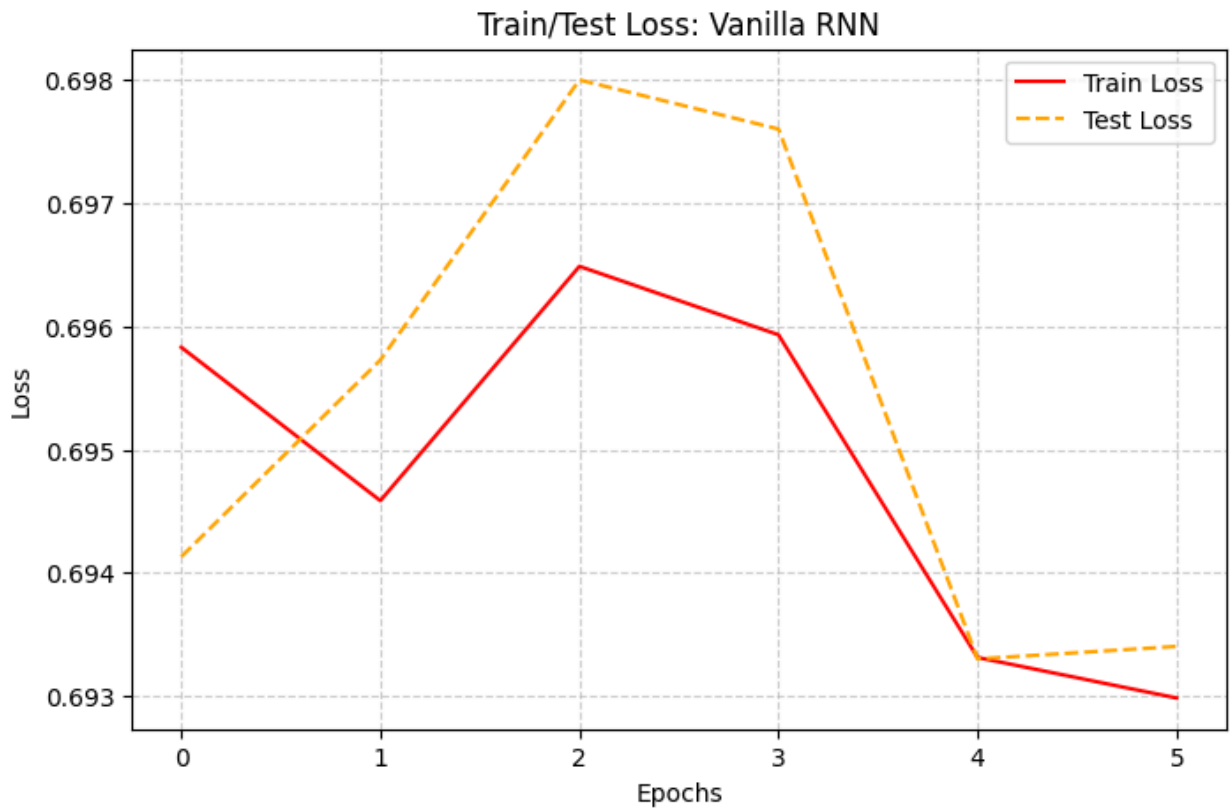
# Plot train and test loss
plt.plot(train_losses_rnn, label='Train Loss', color='red', linestyle='-')
plt.plot(test_losses_rnn, label='Test Loss', color='orange', linestyle='--')

plt.xlabel('Epochs')

```



```
plt.ylabel('Loss')
plt.title('Train/Test Loss: Vanilla RNN')
plt.legend()
plt.grid(True, linestyle='--', alpha=0.6)
plt.show()
```



```
In [ ]: import matplotlib.pyplot as plt

# Assuming you already have:
# LSTM: train_losses, test_losses, accuracy
# Vanilla RNN: train_losses_rnn (or train_losses), test_losses_rnn (or test_lo

# -----
# 1 Loss Comparison
# -----
plt.figure(figsize=(10,6))
plt.plot(train_losses, label='LSTM Train Loss', color='blue', linestyle='-')
plt.plot(test_losses, label='LSTM Test Loss', color='blue', linestyle='--')
plt.plot(train_losses_rnn, label='Vanilla RNN Train Loss', color='red', linestyle='-')
plt.plot(test_losses_rnn, label='Vanilla RNN Test Loss', color='red', linestyle='--')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.title('Train/Test Loss: LSTM vs Vanilla RNN')
plt.legend()
plt.grid(True)
plt.show()

# -----
```

```

# 2 Accuracy Comparison
# -----
plt.figure(figsize=(6,4))
models = ['LSTM', 'Vanilla RNN']
accuracies = [accuracy, accuracy_rnn]
colors = ['blue', 'red']

plt.bar(models, accuracies, color=colors)
for i, acc in enumerate(accuracies):
    plt.text(i, acc + 0.5, f"{acc:.2f}%", ha='center', fontsize=12)
plt.ylabel('Accuracy (%)')
plt.title('Test Accuracy: LSTM vs Vanilla RNN')
plt.ylim(0, 100)
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.show()

```

