Word Embeddings and Recurrent Neural Networks for Sentiment Analysis Student's

Askerbay Arman
Student ID: 24MD0515
Date: 15.04.2025

1 Introduction

Natural Language Processing (NLP) is a field of artificial intelligence that enables computers to understand, interpret, and process human language. This report presents basic methods of text representation and sentiment analysis using neural networks. In the course of the work, various word vectorization techniques (One-hot, TF-IDF[1], Word2Vec[2], GloVe[3], FastText[4]) as well as text classification using RNN, LSTM, and GRU are implemented and compared. The goal is to compare the effectiveness of the models and highlight the advantages of each design.

2 Implementation and Code Snippets

Our goal is to implement and visualize various vectorization methods: One-hot, TF-IDF[1], Word2Vec[2], GloVe[3], FastText[4].

2.1 Exercise 1

```
sentences = ["I love NLP", "NLP is fun", "I love machine learning"]
def one_hot_encode(sentence):
    encoding = []
    for word in sentence.split():
        vec = [0] * len(words)
        vec[word2idx[word]] = 1
        encoding.append(vec)
    return encoding
```

The word2idx dictionary is used, in which each word is assigned its unique index.

```
vectorizer = TfidfVectorizer()
X = vectorizer.fit_transform(sentences)
df = pd.DataFrame(X.toarray(), columns=vectorizer.
   get_feature_names_out())
df.iloc[0].plot(kind='bar', title="TF-IDF for Sentence 1")
plt.show()
```

The TfidfVectorizer object is needed to tokenize (break into words), build a word frequency matrix, calculate TF-IDF values for each word in each document[1].

2.2 Exercise 2

```
cbow_model = Word2Vec(sentences=tokenized, vector_size=50, window=2,
    min_count=1, sg=0)
similarity = cbow_model.wv.similarity('nlp', 'love')
```

```
sg_model = Word2Vec(sentences=tokenized, vector_size=50, window=2,
   min_count=1, sg=1)
similarity = sg_model.wv.similarity('nlp', 'love')
```

This code shows training Word2Vec[2] with two architectures (CBOW and Skip-gram). How to compare the resulting vectors of two words. This is useful for assessing the semantic similarity of words in the trained space.

```
def plot_embeddings(model):
    words = list(model.wv.key_to_index.keys())
    X = model.wv[words]
    pca = PCA(n_components=2)
    result = pca.fit_transform(X)
```

A PCA (Principal Component Analysis)[4] object is created for dimensionality reduction. We reduce the dimensionality of the embeddings from 100 $\rightarrow$ 2 so that they can be displayed on a 2D plot.

2.3 Exercise 3

```
def find_similar(word, top_n=5):
    word_vec = np.array(glove[word]).reshape(1, -1)
    similarities = {w: cosine_similarity([vec], word_vec)[0][0] for
       w, vec in glove.items()}
    similar = sorted(similarities.items(), key=lambda item: item[1],
        reverse=True)[1:top_n+1]
```

The cosine similarity in figure 6 between the vector of a given word word_vec and all other vectors in the glove[3] dictionary is calculated.

```
fasttext_model = FastText(sentences=tokenized, vector_size=50,
   window=2, min_count=1)

similar_words = fasttext_model.wv.most_similar("nlp")
```

Unlike Word2Vec[2], FastText[4] breaks words into substrings (e.g. 'nlp' $\rightarrow$ ['¡nl', 'nlp', 'lp¿']) and trains them, so it can handle new, unfamiliar words if parts of them have already appeared in the data.

2.4 Exercise 4

```
tokenizer = get_tokenizer('basic_english')

def yield_tokens(data_iter):
```

```
    for label, text in data_iter:
        yield tokenizer(text)

train_iter, _ = IMDB(split=('train', 'test'))
vocab = build_vocab_from_iterator(yield_tokens(train_iter), specials
   =["<unk>"])
vocab.set_default_index(vocab["<unk>"])

data = ["hello", "world"]
chars = sorted(list(set("".join(data))))
char2idx = {ch: i for i, ch in enumerate(chars)}
idx2char = {i: ch for ch, i in char2idx.items()}

def one_hot_seq(seq):
    return torch.eye(len(chars))[torch.tensor([char2idx[c] for c in
       seq])]

class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super().__init__()
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        out, hn = self.rnn(x)
        out = self.fc(out)
        return out

input_size = len(chars)
hidden_size = 8
output_size = len(chars)
model = SimpleRNN(input_size, hidden_size, output_size)
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.01)

grad_norms = []

for epoch in range(100):
    total_loss = 0
    model.zero_grad()
```

```
    for word in data:
        x = one_hot_seq(word[:-1]).unsqueeze(0)
        y = torch.tensor([char2idx[c] for c in word[1:]])

        output = model(x).squeeze(0)
        loss = loss_fn(output, y)
        loss.backward()

        total_grad_norm = sum(p.grad.norm().item() for p in model.
            parameters() if p.grad is not None)
        grad_norms.append(total_grad_norm / len(list(model.
            parameters())))

        optimizer.step()
```

Trains an RNN to predict the next letter in a word (character by character). Uses one-hot representations. Stores the average gradient norm so that you can later plot it and see if there is a problem with vanishing gradients.

2.5 Exercise 5

```
train_iter, test_iter = IMDB(split=('train', 'test'))
vocab = build_vocab_from_iterator(yield_tokens(train_iter), specials
    =["<unk>"])
vocab.set_default_index(vocab["<unk>"])

def collate_batch(batch):
    label_map = {'pos': 1, 'neg': 0}
    text_list, label_list = [], []
    for label, text in batch:
        tokens = tokenizer(text)
        ids = torch.tensor(vocab(tokens), dtype=torch.long)
        text_list.append(ids)
        label_list.append(torch.tensor(label if isinstance(label,
            int) else label_map[label], dtype=torch.long))
    text_list = pad_sequence(text_list, batch_first=True)
    return text_list, torch.tensor(label_list)

train_iter, test_iter = IMDB(split=('train', 'test'))
train_loader = DataLoader(list(train_iter)[:1000], batch_size=32,
    collate_fn=collate_batch)
```

```
class RNNClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_size,
       output_size):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
        self.rnn = nn.RNN(embed_dim, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.embed(x)
        _, h = self.rnn(x)
        return self.fc(h.squeeze(0))

# Training setup
model = RNNClassifier(len(vocab), 64, 128, 2)
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

This code is an implementation of an RNN classifier for sentiment analysis on the IMDB dataset. The model will classify reviews as positive (pos) or negative (neg).

2.6 Exercise 6

```
def collate_batch(batch):
    label_map = {'pos': 1, 'neg': 0}
    text_list, label_list = [], []
    for label, text in batch:
        tokens = tokenizer(text)
        ids = torch.tensor(vocab(tokens), dtype=torch.long)
        text_list.append(ids)
        if isinstance(label, str):
            label = label_map[label]
        label_list.append(torch.tensor(label, dtype=torch.long))

    text_list = pad_sequence(text_list, batch_first=True)
    return text_list, torch.tensor(label_list)

train_iter, _ = IMDB(split=('train', 'test'))
train_loader = DataLoader(list(train_iter)[:1000], batch_size=32,
   collate_fn=collate_batch)

class RNNBase(nn.Module):
```

```
    def __init__(self, vocab_size, embed_dim, hidden_size,
       output_size, model_type='LSTM'):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
        if model_type == 'LSTM':
            self.rnn = nn.LSTM(embed_dim, hidden_size, batch_first=
               True)
        else:
            self.rnn = nn.GRU(embed_dim, hidden_size, batch_first=
               True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = self.embed(x)
        _, h = self.rnn(x)
        if isinstance(h, tuple):
            h = h[0]
        return self.fc(h.squeeze(0))

def train_model(model_type='LSTM'):
    model = RNNBase(len(vocab), 64, 128, 2, model_type)
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    loss_fn = nn.CrossEntropyLoss()

    losses = []
    accuracies = []

    start = time.time()
    for epoch in range(5):
        total_loss = 0
        correct = 0
        total = 0
        for x_batch, y_batch in train_loader:
            optimizer.zero_grad()
            output = model(x_batch)
            loss = loss_fn(output, y_batch)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
            preds = torch.argmax(output, dim=1)
```

```
            correct += (preds == y_batch).sum().item()
            total += y_batch.size(0)

        losses.append(total_loss / len(train_loader))
        accuracies.append(correct / total)
        print(f"[{model_type}] Epoch {epoch+1}: Loss = {losses
            [-1]:.4f}, Accuracy = {accuracies[-1]:.4f}")
    duration = time.time() - start
    return losses, accuracies, duration

print("\nTraining LSTM...")
lstm_losses, lstm_accuracies, lstm_time = train_model('LSTM')

print("\nTraining GRU...")
gru_losses, gru_accuracies, gru_time = train_model('GRU')
```

This code implements and compares two recurrent neural networks, LSTM and GRU, for the task of classifying reviews from the IMDB dataset into positive and negative.

2.7 Exercise 7

```
text = (
    "To be, or not to be, that is the question:\n"
    "Whether 'tis nobler in the mind to suffer\n"
    "The slings and arrows of outrageous fortune,\n"
    "Or to take arms against a sea of troubles\n"
    "And by opposing end them."
)

chars = sorted(set(text))
char2idx = {ch: idx for idx, ch in enumerate(chars)}
idx2char = {idx: ch for ch, idx in char2idx.items()}
vocab_size = len(chars)

def encode_text(txt):
    return [char2idx[ch] for ch in txt]

def decode_text(indices):
    return ''.join([idx2char[idx] for idx in indices])

encoded = encode_text(text)
seq_len = 40
```

```
def get_batches(encoded, batch_size):
    inputs, targets = [], []
    for i in range(len(encoded) - seq_len):
        x_seq = encoded[i:i+seq_len]
        y_seq = encoded[i+1:i+seq_len+1]
        inputs.append(torch.tensor(x_seq))
        targets.append(torch.tensor(y_seq))
    dataset = list(zip(inputs, targets))
    return DataLoader(dataset, batch_size=batch_size, shuffle=True)

class CharLSTM(nn.Module):
    def __init__(self, vocab_size, embed_dim=64, hidden_dim=128):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, vocab_size)

    def forward(self, x, hidden=None):
        x = self.embed(x)
        out, hidden = self.lstm(x, hidden)
        out = self.fc(out)
        return out, hidden

model = CharLSTM(vocab_size)
loss_fn = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.003)

train_loader = get_batches(encoded, batch_size=16)
losses = []

for epoch in range(15):
    total_loss = 0
    for x, y in train_loader:
        optimizer.zero_grad()
        out, _ = model(x)
        out = out.view(-1, vocab_size)
        y = y.view(-1)
        loss = loss_fn(out, y)
        loss.backward()
        optimizer.step()
```

```
        total_loss += loss.item()
    losses.append(total_loss / len(train_loader))
    print(f"Epoch {epoch + 1}, Loss: {losses[-1]:.4f}")

plt.plot(losses)
plt.title("Training Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.show()

def generate_text(model, start_text, length=200):
    model.eval()
    input_seq = torch.tensor([char2idx[ch] for ch in start_text],
       dtype=torch.long).unsqueeze(0)
    hidden = None
    generated = list(start_text)

    with torch.no_grad():
        for _ in range(length):
            out, hidden = model(input_seq, hidden)
            last_logits = out[:, -1, :]
            probs = torch.softmax(last_logits, dim=-1).squeeze()
            char_idx = torch.multinomial(probs, 1).item()
            generated.append(idx2char[char_idx])
            input_seq = torch.tensor([[char_idx]], dtype=torch.long)
    return ''.join(generated)
```

This code is an LSTM-based text generator that is trained on a passage from Shakespeare's Hamlet. It first learns to predict the next character, and then uses that to generate new text in the style of the original. The model learns to write text similar to Shakespeare. It predicts character by character, not words. LSTM "remembers" the context so that the predictions are logical. Generation is done character by character, taking into account all the previous text.

2.8 Exercise 8

```
label_map = {'pos': 1, 'neg': 0}

def collate_batch(batch):
    text_list, label_list = [], []
    for label, text in batch:
        if isinstance(label, int):
            label_id = label
```

```
        else:
            label_id = label_map[label]
        tokens = tokenizer(text)
        ids = torch.tensor(vocab(tokens), dtype=torch.long)
        text_list.append(ids)
        label_list.append(torch.tensor(label_id, dtype=torch.long))
    text_list = pad_sequence(text_list, batch_first=True)
    label_list = torch.tensor(label_list)
    return text_list, label_list

train_iter, test_iter = IMDB(split=('train', 'test'))
train_loader = DataLoader(list(train_iter)[:1000], batch_size=32,
   collate_fn=collate_batch)
test_loader = DataLoader(list(test_iter)[:500], batch_size=32,
   collate_fn=collate_batch)

class LSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, output_dim
       ):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, batch_first=True)
        self.fc = nn.Linear(hidden_dim, output_dim)

    def forward(self, x):
        embedded = self.embed(x)
        _, (hidden, _) = self.lstm(embedded)
        return self.fc(hidden[-1])

class BiLSTMClassifier(nn.Module):
    def __init__(self, vocab_size, embed_dim, hidden_dim, output_dim
       ):
        super().__init__()
        self.embed = nn.Embedding(vocab_size, embed_dim)
        self.lstm = nn.LSTM(embed_dim, hidden_dim, batch_first=True,
            bidirectional=True)
        self.fc = nn.Linear(hidden_dim * 2, output_dim)

    def forward(self, x):
        embedded = self.embed(x)
        _, (hidden, _) = self.lstm(embedded)
```

```
        h = torch.cat((hidden[-2], hidden[-1]), dim=1)
        return self.fc(h)

def train_model(model, train_loader, epochs=5):
    loss_fn = nn.CrossEntropyLoss()
    optimizer = optim.Adam(model.parameters(), lr=0.001)
    train_losses, train_accuracies = [], []

    for epoch in range(epochs):
        total_loss = 0
        correct, total = 0, 0
        for x_batch, y_batch in train_loader:
            optimizer.zero_grad()
            output = model(x_batch)
            loss = loss_fn(output, y_batch)
            loss.backward()
            optimizer.step()
            total_loss += loss.item()
            preds = torch.argmax(output, dim=1)
            correct += (preds == y_batch).sum().item()
            total += y_batch.size(0)
        acc = correct / total
        train_losses.append(total_loss / len(train_loader))
        train_accuracies.append(acc)
        print(f"Epoch {epoch+1}: Loss={train_losses[-1]:.4f},
           Accuracy={acc:.4f}")

    return train_losses, train_accuracies

print("Training Standard LSTM")
lstm_model = LSTMClassifier(len(vocab), 64, 128, 2)
lstm_losses, lstm_accs = train_model(lstm_model, train_loader)

print("\nTraining Bidirectional LSTM")
bilstm_model = BiLSTMClassifier(len(vocab), 64, 128, 2)
bilstm_losses, bilstm_accs = train_model(bilstm_model, train_loader)
```

This code implements and compares two models for classifying text based on reviews from the IMDB dataset: LSTMClassifier is a regular unidirectional LSTM BiLSTMClassifier is a bidirectional LSTM (BiLSTM) that reads text from both left to right and right to left

Both models are trained to determine whether a review is positive (pos) or negative (neg).

Compares the performance of regular LSTM and bidirectional BiLSTM on the IMDB review classification task

Trains and visualizes the quality at each epoch

BiLSTM usually shows better accuracy because it takes into account the context from both sides

3 Results and Discussion

3.1 One-hot

One-hot express simple, but do not reflect the semantic closeness of words. For example, the words ”good” and ”cute” belong to completely different vectors, although they have a good meaning[5].

That is why NLP often prefers to use Word2Vec[2], GloVe[3], FastText - they draw dense contours with semantic information[4].

```
Sentence: I love NLP
One-hot: [[0, 0, 0, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0]]

Sentence: NLP is fun
One-hot: [[0, 0, 0, 0, 1, 0, 0], [0, 0, 0, 0, 0, 1, 0], [0, 0, 1, 0, 0, 0, 0]]

Sentence: I love machine learning
One-hot: [[0, 0, 0, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0, 0], [0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 1]]
```

Figure 1: Visualization of One-hot for all sentences

3.2 TF-IDF

TF-IDF Table:

	fun	is	learning	love	machine	nlp
0	0.000000	0.000000	0.000000	0.707107	0.000000	0.707107
1	0.622766	0.622766	0.000000	0.000000	0.000000	0.473630
2	0.000000	0.000000	0.622766	0.473630	0.622766	0.000000

Figure 2: TF-IDF Table for all sentences

At figure 2 TF-IDF[1] shows which words distinguish each document.

For example, if document 0 contains ”love” and ”nlp”, the model can infer that the document is more about positive feelings and NLP.

TF-IDF is a basic technique for analyzing text before training a model, especially in logistic regression, SVM or decision trees[1].

This visualization at figure 3 helps you understand which words are key for the document.

Feature selection when training models. Removing noise words. Interpreting the model if you are using logistic regression, SVM, etc.

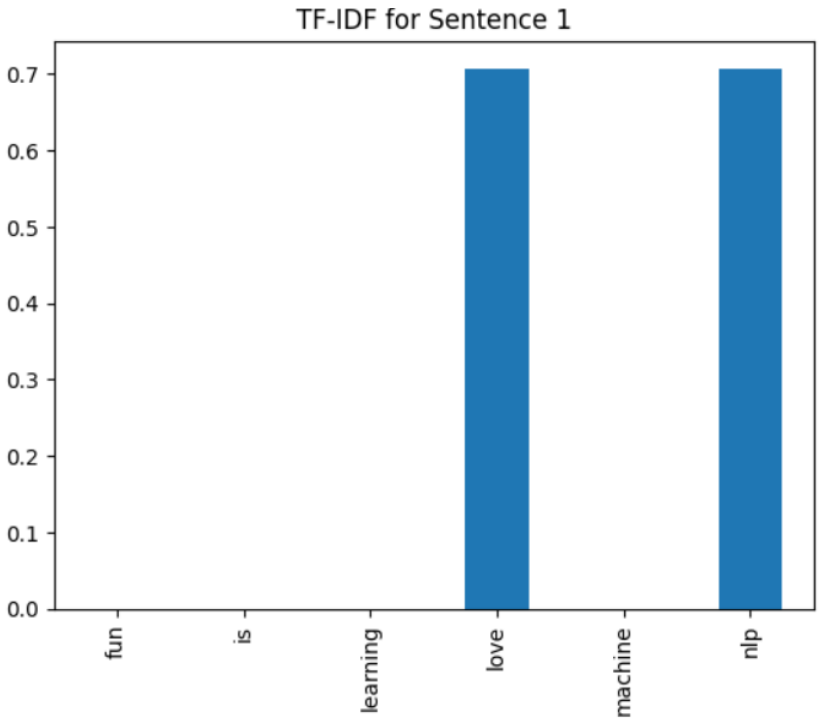

Figure 3: Visualization of TF-IDF for 1st Sentence

3.3 Word2Vec

```
CBOW similarity (nlp, love): 0.011071973
Skip-gram similarity (nlp, love): 0.011071973
```

Figure 4: CBOW vs. Skip-gram

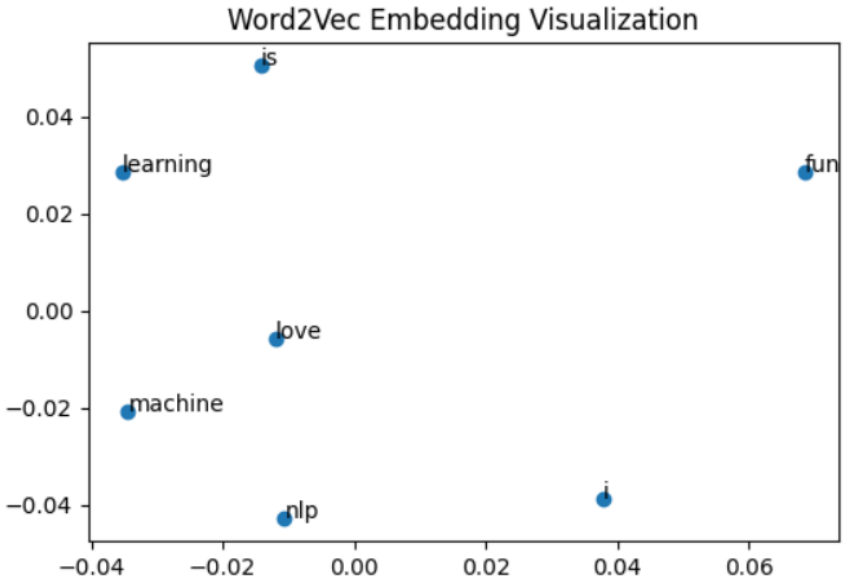

Figure 5: Word2Vec Embedding Visualization

The graph in figure 5 shows how Word2Vec[2] reflects semantic relationships between words: the closer the words are in the vector space, the more related they are in meaning. ”PHP” is far away because it belongs to a different subject area.

The cosine similarity in figure 6 between the vector of a given word word_vec and all other vectors in the glove[3] dictionary is calculated.

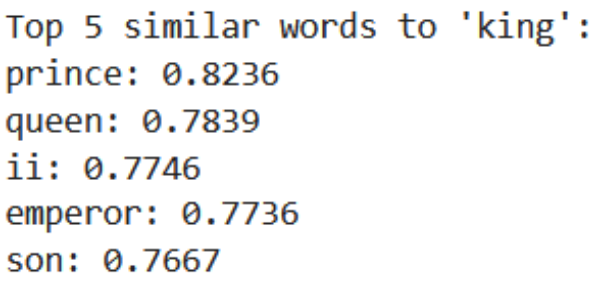

Figure 6: Top 5 similar words to ’king’

The model is trained on large texts and ”learns” to represent words as vectors, where semantically close words (for example, ”king” and ”queen”) are next to each other in vector space. The coefficients show how much their vectors coincide.

3.4 FastText

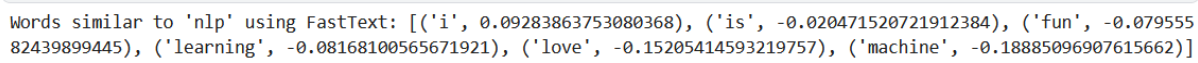

Figure 7: Words similar to ’nlp’ using FastText

This figure 7 shows the results of searching for semantically similar words to the term ’nlp’ (Natural Language Processing) using the FastText model[4]. However, unlike the classic Word2Vec[2], the results look unexpected: instead of thematically related words (e.g. ”text”, ”ai”, ”linguistics”), the model returned common words (”i”, ”is”, ”fun”) with very low similarity coefficients, including negative values.

3.5 SimpleRNN

This figure 8 shows a graph of the Average Gradient Norm during the training of a model (e.g., a neural network) over epochs.

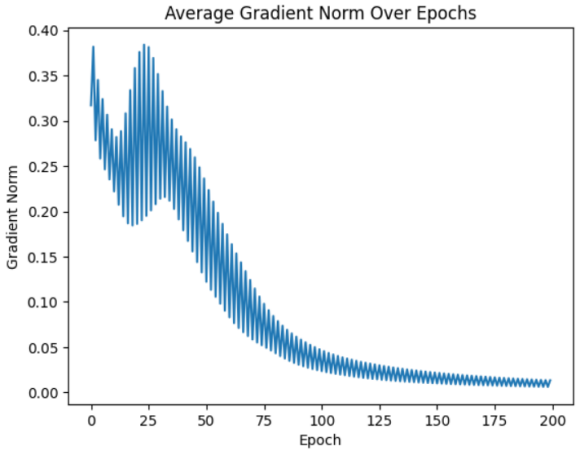

Figure 8: Average Gradient Norm over epochs

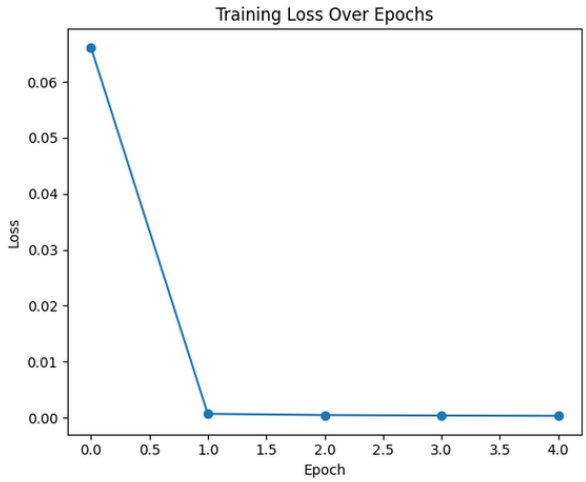

Figure 9: SimpleRNN training loss over epochs

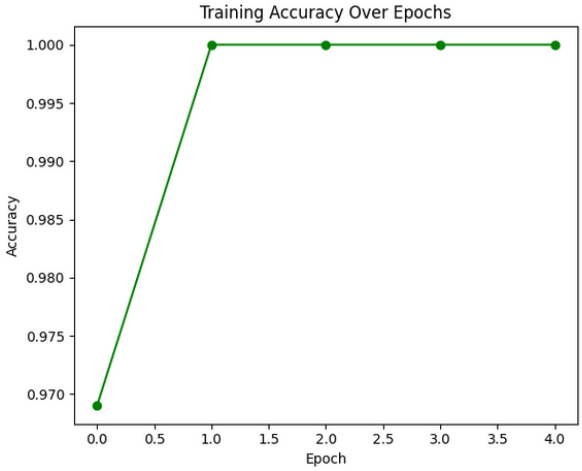

Figure 10: SimpleRNN training accuracy over epochs

Figures 9, 10 show two interconnected graphs showing the training process of a machine learning model (e.g. a neural network) over epochs.

In this experiment at table 1 and in figures 12, 11 GRU showed slightly better results in both

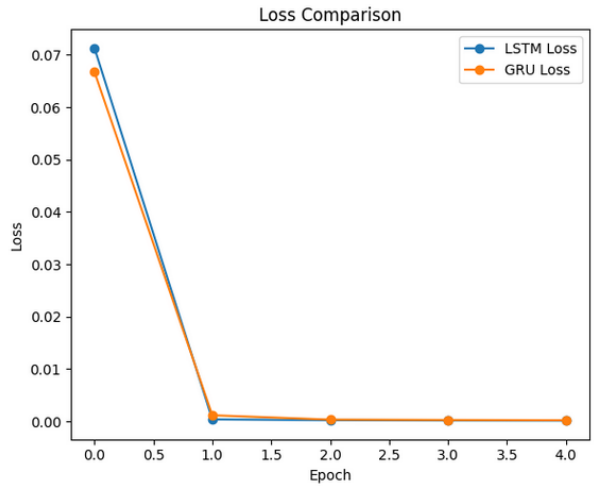

Figure 11: LSTM vs GRU loss

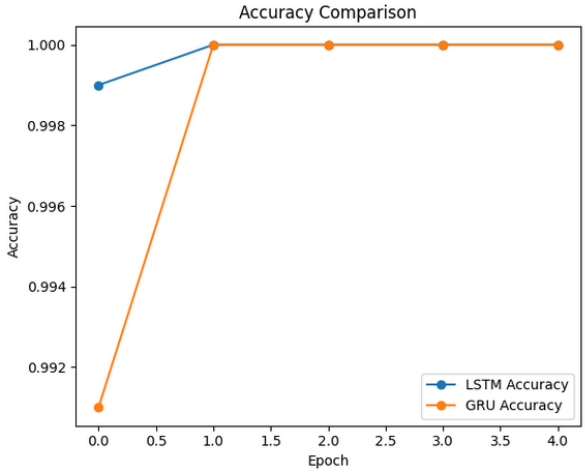

Figure 12: LSTM vs GRU accuracy

Loss Comparison	**Accuracy Comparison**
The vertical axis shows the loss values (from 0.00 to 0.07).	The vertical axis shows the accuracy (from 0.992 to 1.000).
The horizontal axis shows the training epochs (from 0.0 to 4.0).	The horizontal axis shows the same training epochs.
The two lines are compared: LSTM Loss and GRU Loss.	The LSTM Accuracy and GRU Accuracy are compared.
Both models show a decrease in loss with increasing number of epochs, with GRU achieving slightly lower loss values.	Both models show an increase in accuracy, with GRU demonstrating slightly higher accuracy in later epochs.

Table 1: Comparison of LSTM and GRU based on Loss and Accuracy

loss reduction and accuracy growth compared to LSTM. Both models are successfully trained, achieving high accuracy (99%) after 4 epochs.

Figure 13 shows the graph of losses (Training Loss) during the model training process. Features: Loss reduction dynamics - if the values decrease over time, the model is trained

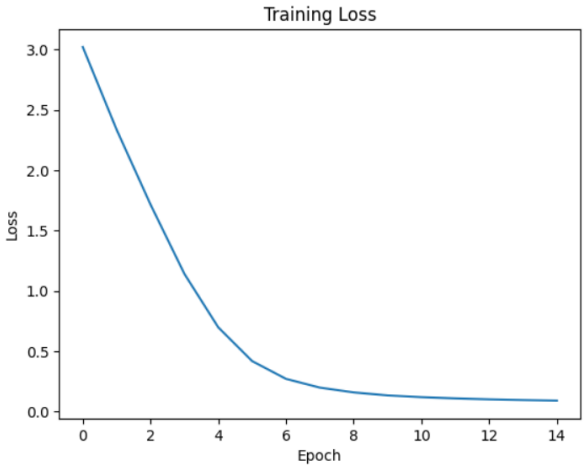

Figure 13: CharLSTM Training Loss

correctly; Sharp jumps are signs of instability; Plateaus are possible overfitting or need to change hyperparameters.

```
Generated Text:
To be, or not to be, that is the question:
Whether 'tis nobler is the question:
Whether 'tis nobler in the mind to suffer
The slings and arrows of ourtune,
Or to take arms agais nd arrows of outrageous fortune,
Or to take arms against a sea of troubles
And by opposing end that is the question:
Whether 'tis no
```

Figure 14: LSTM vs GRU accuracy

Figure 14 shows the results of a model generating text based on a famous Shakespearean passage from Hamlet. The model produced errors and repetitions (for example, duplication of the phrase ”that is the question” and ”Whether ’tis nobler”).

Figures 15, 16 comparative analytics dashboard for two deep learning models, LSTM (unidirectional) and BiLSTM (bidirectional), during their training. Two curves in figure 15 showing how the loss decreases with each epoch. BiLSTM often shows faster loss reduction due to bidirectional context analysis. Accuracy growth curves in figure 16. BiLSTM generally achieves higher accuracy, especially in context-sensitive tasks (e.g. NLP).

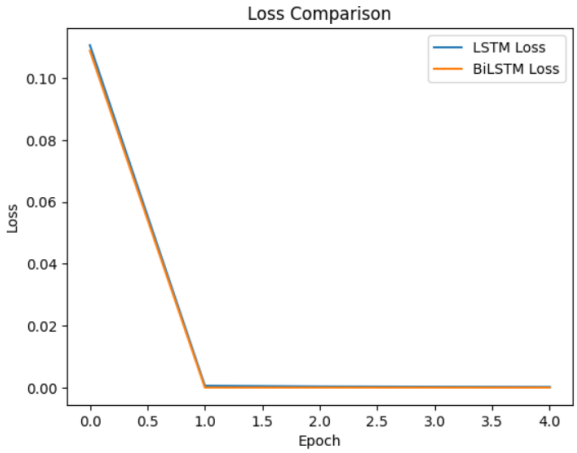

Figure 15: LSTM vs GRU accuracy

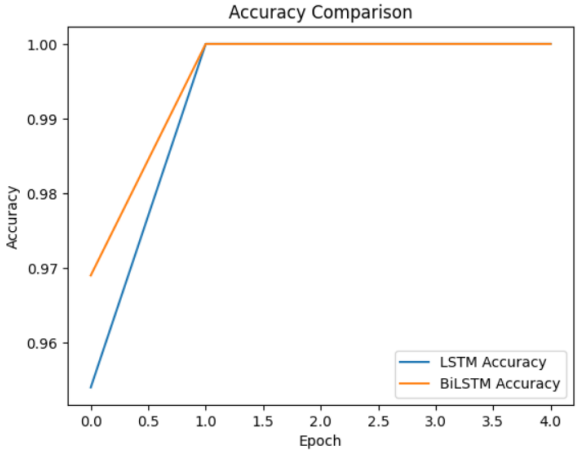

Figure 16: LSTM vs GRU accuracy

References

[1] Huilong Fan and Yongbin Qin. Research on text classification based on improved tf-idf algorithm. In *Proceedings of the 2018 International Conference on Network, Communication, Computer Engineering (NCCE 2018)*, pages 501–506. Atlantis Press, 2018/05.

[2] Xin Rong. word2vec parameter learning explained, 2016.

[3] Jeffrey Pennington, Richard Socher, and Christopher Manning. GloVe: Global vectors for word representation. In Alessandro Moschitti, Bo Pang, and Walter Daelemans, editors, *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, October 2014. Association for Computational Linguistics.

[4] Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. Bag of tricks for efficient text classification, 2016.

[5] Ekaterina Poslavskaya and Alexey Korolev. Encoding categorical data: Is there yet anything 'hotter' than one-hot encoding?, 2023.