KAZAKH-BRITISH TECHNICAL UNIVERSITY

DSSC5101 Natural language processing and pattern recognition methods

Educational Program 7M06107 Computer science and Data analytics

Midterm

Building an NLP-Based Sentiment Analysis and Text Generation System Using Deep Learning

24MD0525

Nitaliyeva Shakhidana

Date: 04.03.2025

ALMATY, 2025 year

1. Introduction

Objective of the Project: Overview of sentiment analysis and text generation. Discuss the significance of word embeddings, RNNs, LSTMs, and GRUs in NLP.

Purpose: To build and compare different models and evaluate their performance in sentiment classification and text generation tasks.

Outline of Report: Describe the structure of the report.

2. Data Preprocessing and Word Embeddings

2.1. Text Preprocessing: Discuss how the text was tokenized, lemmatized, and filtered using **NLTK** or **spaCy**.

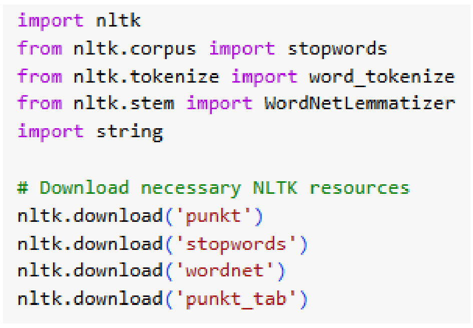
Tokenization: Both NLTK and space tokenize the text similarly, splitting it into words, punctuation, and symbols. However, space handles tokenization as part of its processing pipeline, which can be more sophisticated and context-aware.

Lemmatization: NLTK uses a simple lemmatizer (WordNetLemmatizer) that requires some manual setup (such as POS tagging) for accurate results. Words like "visited" and "offered" remain unchanged. spacey performs lemmatization based on its pre-trained model, which is more context-sensitive. It correctly lemmatized "visited" to "visit" and "her" to "she".

Filtering: **NLTK** removes common stopwords using a predefined list, but it can miss some context-sensitive stopwords. **spaCy** is more sophisticated in identifying stopwords, and its built-in stopword list is more comprehensive, removing words like "is", "the", and "on", punctuation, and non-alphanumeric characters.

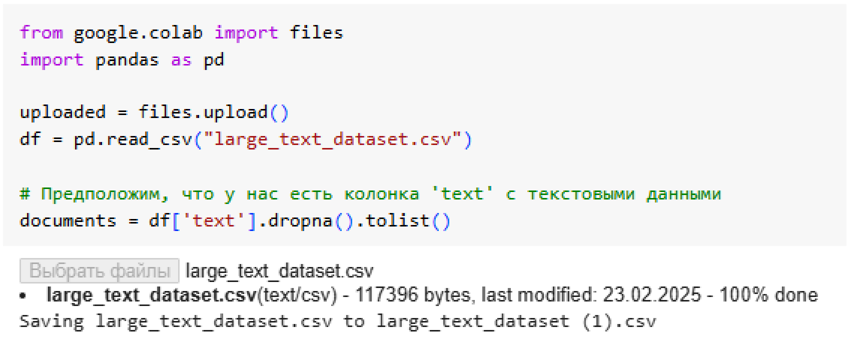
I filtered by using NLTK for text processing.

Downloading the necessary libraries.



```
import nltk
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize
from nltk.stem import WordNetLemmatizer
import string

# Download necessary NLTK resources
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('wordnet')
nltk.download('punkt_tab')
```

I uploaded the dataset manually. A dataset in CSV format.

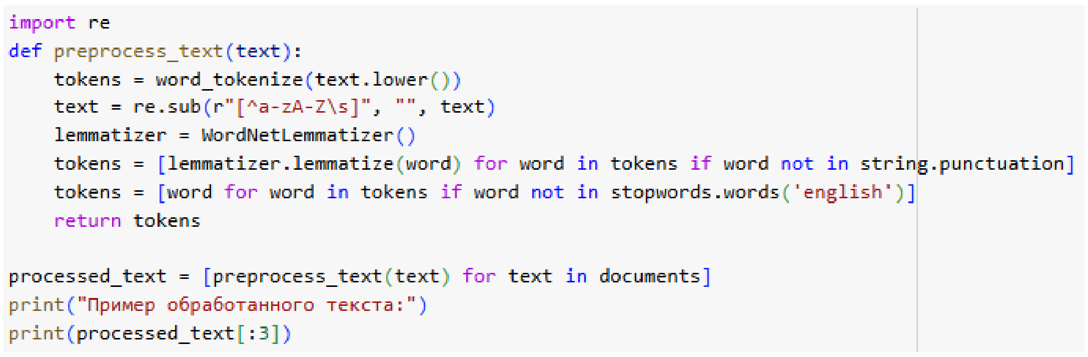

```
from google.colab import files
import pandas as pd

uploaded = files.upload()
df = pd.read_csv("large_text_dataset.csv")

# Предположим, что у нас есть колонка 'text' с текстовыми данными
documents = df['text'].dropna().tolist()
```

Выбрать файлы large_text_dataset.csv

- **large_text_dataset.csv**(text/csv) - 117396 bytes, last modified: 23.02.2025 - 100% done

```
Saving large_text_dataset.csv to large_text_dataset (1).csv
```

Next, we process the text, removing all characters and stopwords.


```
import re
def preprocess_text(text):
    tokens = word_tokenize(text.lower())
    text = re.sub(r"[^a-zA-Z\s]", "", text)
    lemmatizer = WordNetLemmatizer()
    tokens = [lemmatizer.lemmatize(word) for word in tokens if word not in string.punctuation]
    tokens = [word for word in tokens if word not in stopwords.words('english')]
    return tokens

processed_text = [preprocess_text(text) for text in documents]
print("Пример обработанного текста:")
print(processed_text[:3])
```

In this fragment, we process text by using NLTK and output the first 3 sentences from the dataset.

2.2. Word Embeddings: Describe the different word embeddings used (Word2Vec, GloVe, FastText) and how you evaluated them.

Word2Vec — allows you to convert words from a corpus of texts into vectors of numbers in such a way that words with similar semantic meanings have similar vector representations in a multidimensional space. This makes Word2Vec a powerful tool for natural language processing (NLP) tasks such as tonality analysis, machine translation, automatic summarization, and many others.

The two main architectures of the Word2Vec model

CBOW (Continuous Bag of Words): This approach predicts the current word based on the context around it. For example, for the phrase "blue sky above", the CBOW model will try to predict the word "sky" based on the contextual words "blue", "over", "head". CBOW processes large amounts of data quickly, but is less efficient for rare words.

Skip-Gram: In this approach, on the contrary, the current word is used to predict words in its context. For the same example, the Skip-Grant model will try to predict the words "blue", "above", "head" based on the word "sky". Skip-Gram processes data more slowly, but it works better with rare words and less frequent contexts.

Glove is based on matrix factorization technique on word context matrix. It first constructs a large matrix of (words x context) co-occurrence information ie, for each word, you count how frequently we see those words in some context in a large corpus.

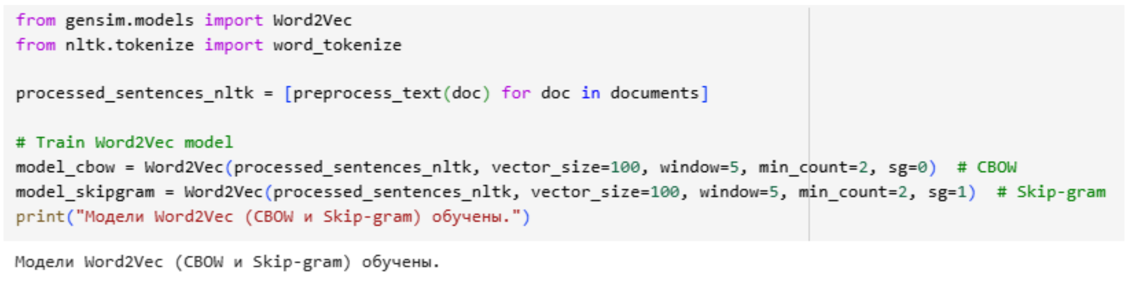
In NLP, global matrix factorization is the process of using matrix factorization form linear algebra to reduce large term frequency matrices. These matrices usually represent the occurrences or the absence of words in the document.

FastText is a vector representation technique developed by facebook AI research. As its name suggests its fast and efficient method to perform same task and because of the nature of its training method, it ends up learning morphological details as well.

FastText is unique because it can derive word vectors for unknown words or out of vocabulary words — this is because by taking morphological characteristics of words into account, it can create the word vector for an unknown word.

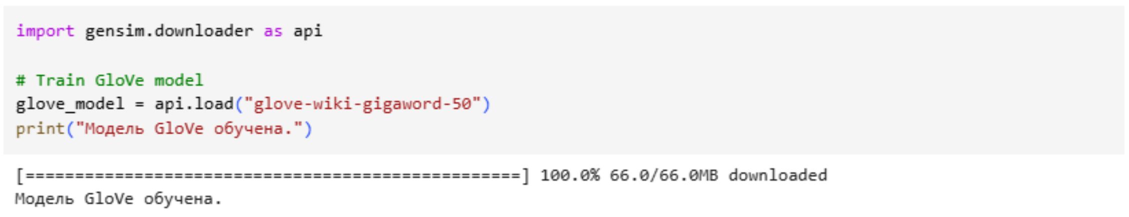
FastText works well with rare words. So even if a word wasn't seen during training, it can be broken down into n-grams to get its embeddings.

For our course work, we evaluated three different methods of embedding words. To do this, download the necessary libraries, create a models and train them.



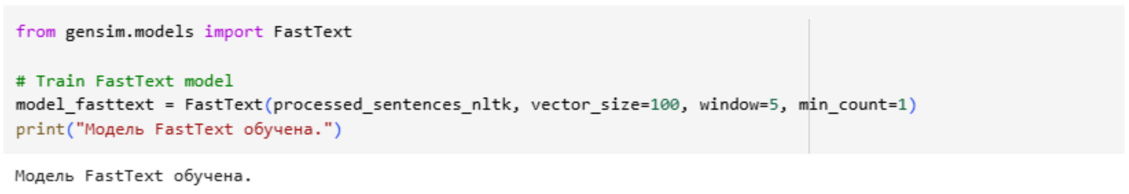
```
from gensim.models import Word2Vec
from nltk.tokenize import word_tokenize

processed_sentences_nltk = [preprocess_text(doc) for doc in documents]

# Train Word2Vec model
model_cbow = Word2Vec(processed_sentences_nltk, vector_size=100, window=5, min_count=2, sg=0)  # CBOW
model_skipgram = Word2Vec(processed_sentences_nltk, vector_size=100, window=5, min_count=2, sg=1)  # Skip-gram
print("Модели Word2Vec (CBOW и Skip-gram) обучены.")
```

```
Модели Word2Vec (CBOW и Skip-gram) обучены.
```



```
import gensim.downloader as api

# Train GloVe model
glove_model = api.load("glove-wiki-gigaword-50")
print("Модель GloVe обучена.")
```

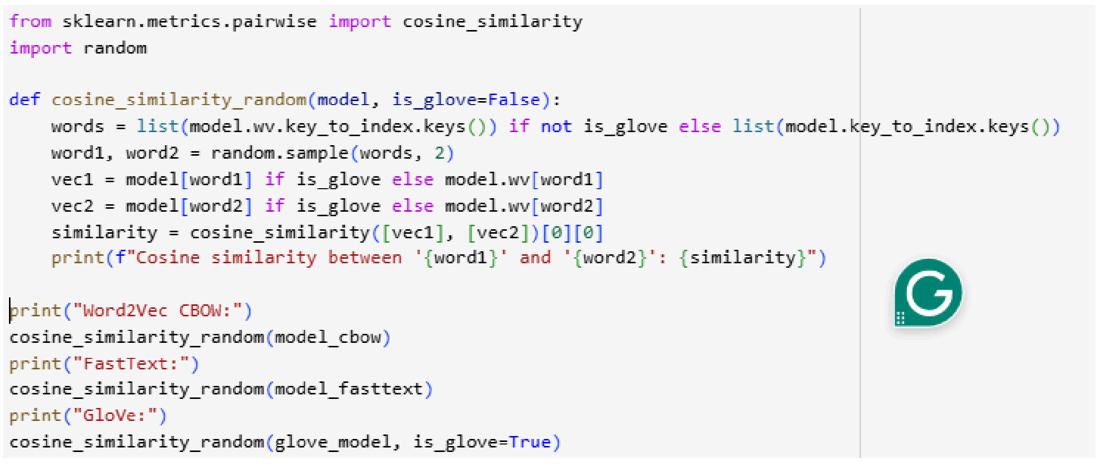
```
[==================================================] 100.0% 66.0/66.0MB downloaded
Модель GloVe обучена.
```



```
from gensim.models import FastText

# Train FastText model
model_fasttext = FastText(processed_sentences_nltk, vector_size=100, window=5, min_count=1)
print("Модель FastText обучена.")
```

```
Модель FastText обучена.
```

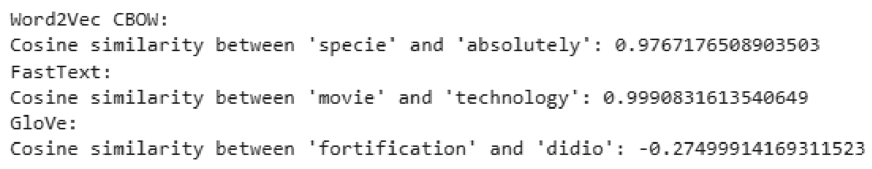
Next, we calculate the cosine similarity between random words from the vector space of the Word2Vec CBOW, FastText, and GloVe models.



```python
from sklearn.metrics.pairwise import cosine_similarity
import random

def cosine_similarity_random(model, is_glove=False):
    words = list(model.wv.key_to_index.keys()) if not is_glove else list(model.key_to_index.keys())
    word1, word2 = random.sample(words, 2)
    vec1 = model[word1] if is_glove else model.wv[word1]
    vec2 = model[word2] if is_glove else model.wv[word2]
    similarity = cosine_similarity([vec1], [vec2])[0][0]
    print(f"Cosine similarity between '{word1}' and '{word2}': {similarity}")

print("Word2Vec CBOW:")
cosine_similarity_random(model_cbow)
print("FastText:")
cosine_similarity_random(model_fasttext)
print("GloVe:")
cosine_similarity_random(glove_model, is_glove=True)
```

Close to 1- the words are very similar.

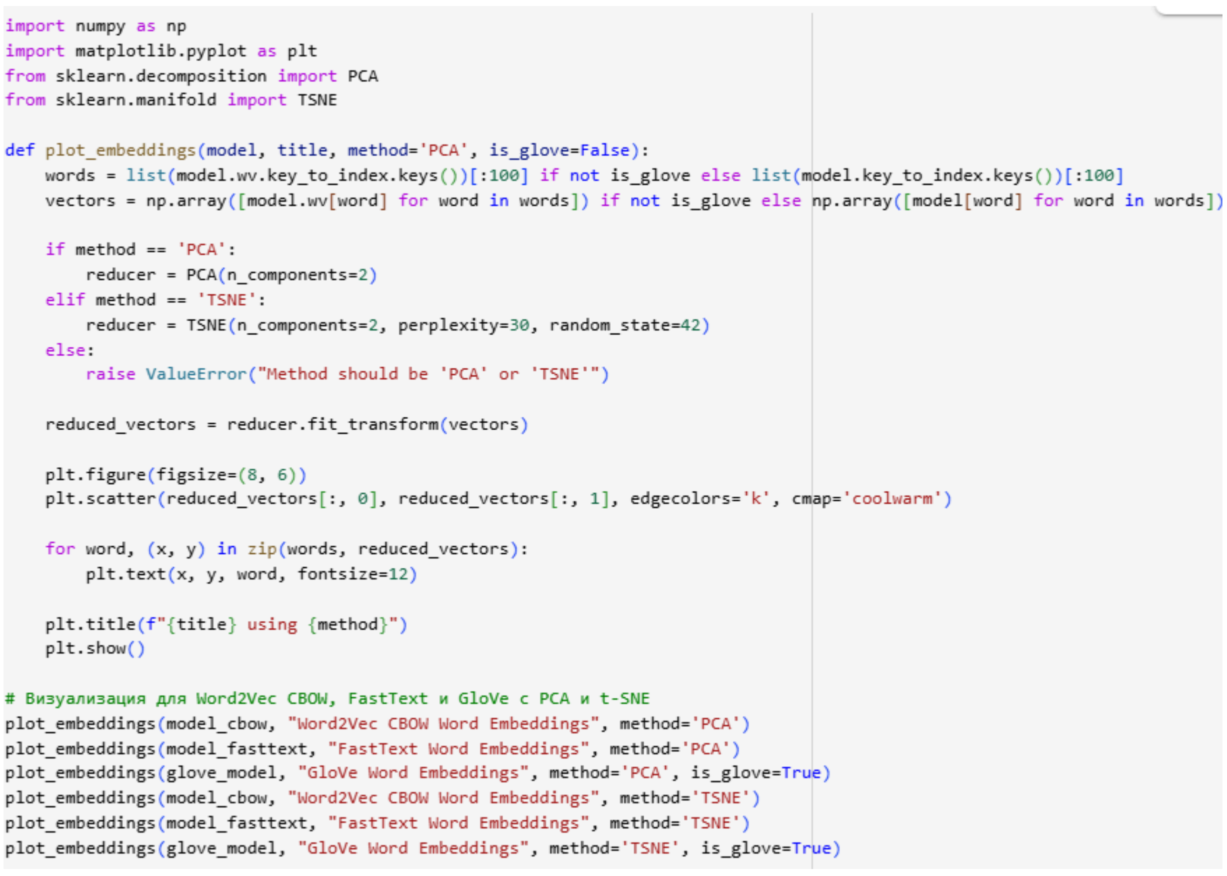
Close to 0 - the words are not related.



```
Word2Vec CBOW:
Cosine similarity between 'specie' and 'absolutely': 0.9767176508903503
FastText:
Cosine similarity between 'movie' and 'technology': 0.9990831613540649
GloVe:
Cosine similarity between 'fortification' and 'didio': -0.27499914169311523
```

2.3. Code Snippets: Include relevant code for preprocessing and embedding generation.

We are writing code to visualize the graph. We take the first 100 words from the model's dictionary. For visualization, we compress vectors from 100D to 2D so that we can draw on a graph.



```python
import numpy as np
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

def plot_embeddings(model, title, method='PCA', is_glove=False):
    words = list(model.wv.key_to_index.keys())[:100] if not is_glove else list(model.key_to_index.keys())[:100]
    vectors = np.array([model.wv[word] for word in words]) if not is_glove else np.array([model[word] for word in words])

    if method == 'PCA':
        reducer = PCA(n_components=2)
    elif method == 'TSNE':
        reducer = TSNE(n_components=2, perplexity=30, random_state=42)
    else:
        raise ValueError("Method should be 'PCA' or 'TSNE'")

    reduced_vectors = reducer.fit_transform(vectors)

    plt.figure(figsize=(8, 6))
    plt.scatter(reduced_vectors[:, 0], reduced_vectors[:, 1], edgecolors='k', cmap='coolwarm')

    for word, (x, y) in zip(words, reduced_vectors):
        plt.text(x, y, word, fontsize=12)

    plt.title(f"{title} using {method}")
    plt.show()

# Визуализация для Word2Vec CBOW, FastText и GloVe с PCA и t-SNE
plot_embeddings(model_cbow, "Word2Vec CBOW Word Embeddings", method='PCA')
plot_embeddings(model_fasttext, "FastText Word Embeddings", method='PCA')
plot_embeddings(glove_model, "GloVe Word Embeddings", method='PCA', is_glove=True)
plot_embeddings(model_cbow, "Word2Vec CBOW Word Embeddings", method='TSNE')
plot_embeddings(model_fasttext, "FastText Word Embeddings", method='TSNE')
plot_embeddings(glove_model, "GloVe Word Embeddings", method='TSNE', is_glove=True)
```

Outputs:

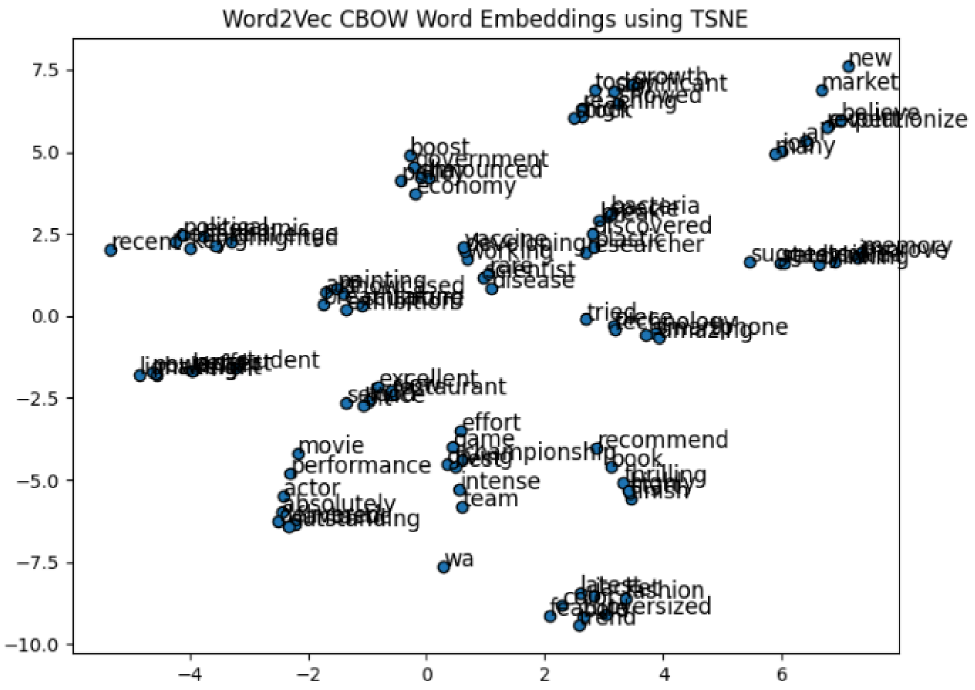


Figure 1: t-SNE visualization of Word2Vec embeddings

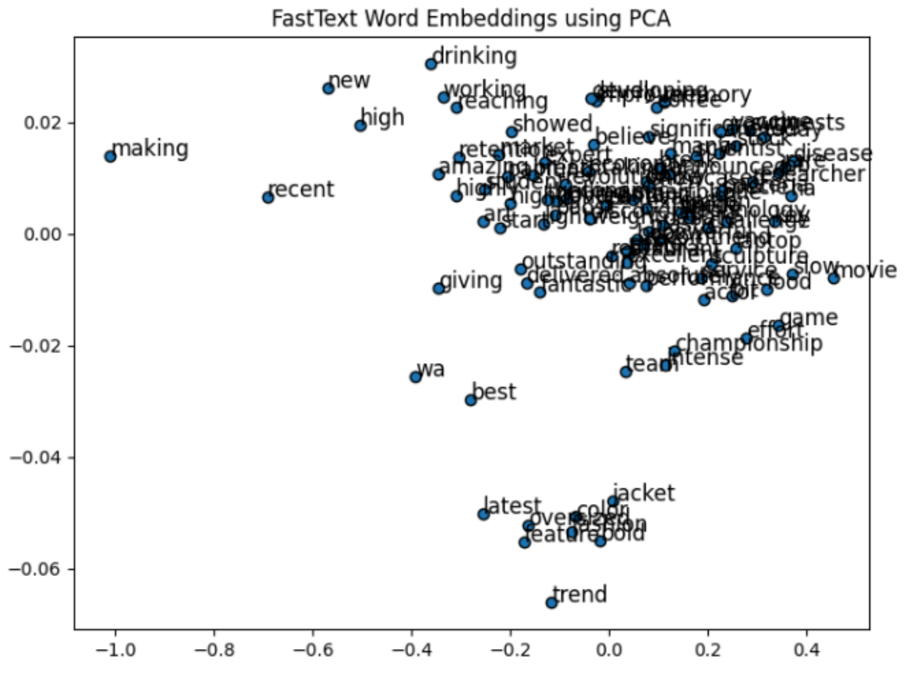


Figure 2: PCA visualization of FastText embeddings

PCA (Principal Component Analysis) is a dimensionality reduction method that linearly projects data into a space with a smaller dimension. It preserves the global data structure well, but it does not always reflect complex nonlinear dependencies.

t-SNE (t-Distributed Stochastic Neighbor Embedding) is a nonlinear dimensionality reduction method that preserves the local data structure, making it useful for clustering. However, tSNE is more difficult to set up and requires more calculations.

3. Recurrent Neural Networks for Sentiment Analysis

3.1. Describe the architecture of the simple RNN and its application to sentiment analysis.

Recurrent Neural Networks (RNNs) are specifically designed to handle sequential data, making them a powerful tool in various applications such as natural language processing and time series prediction. Unlike traditional feedforward neural networks, RNNs have connections that loop back on themselves, allowing them to maintain a form of memory about previous inputs. This unique architecture enables RNNs to process sequences of varying lengths, which is essential for tasks where context and order matter.

Simple RNN Architecture

The architecture of a simple RNN consists of an input layer, a hidden layer, and an output layer. The hidden layer is where the recurrent connections occur, allowing the network to retain information from previous time steps. The basic formula governing the hidden state at time step t is:

[h_t = f(W_h h_{t-1} + W_x x_t + b)]

Where:

(h_t) is the hidden state at time t.

(W_h) is the weight matrix for the hidden state.

(W_x) is the weight matrix for the input.

(x_t) is the input at time t.

(b) is the bias term.

(f) is a non-linear activation function, typically a tanh or ReLU.

Applications of Simple RNNs

Despite their limitations, simple RNNs are still used in various applications, including:

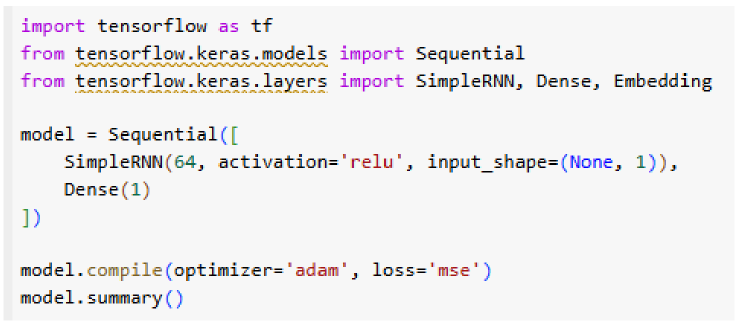
Language Modeling: Predicting the next word in a sentence based on the previous words.

Time Series Prediction: Forecasting future values based on past observations.

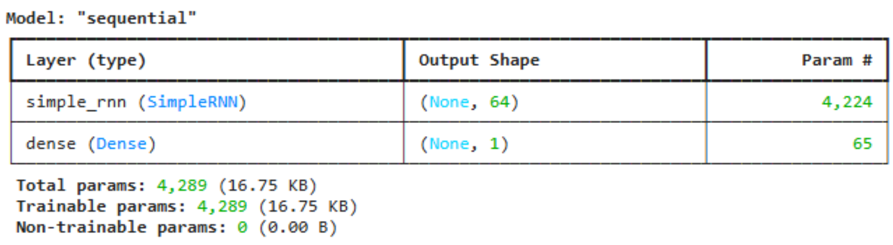
Sequence Classification: Classifying sequences of data, such as sentiment analysis in text.

3.2. Training and Results: Discuss the training process, learning curves (accuracy and loss), and the vanishing gradient problem.

Here we are using TensorFlow for creating RNN architecture.

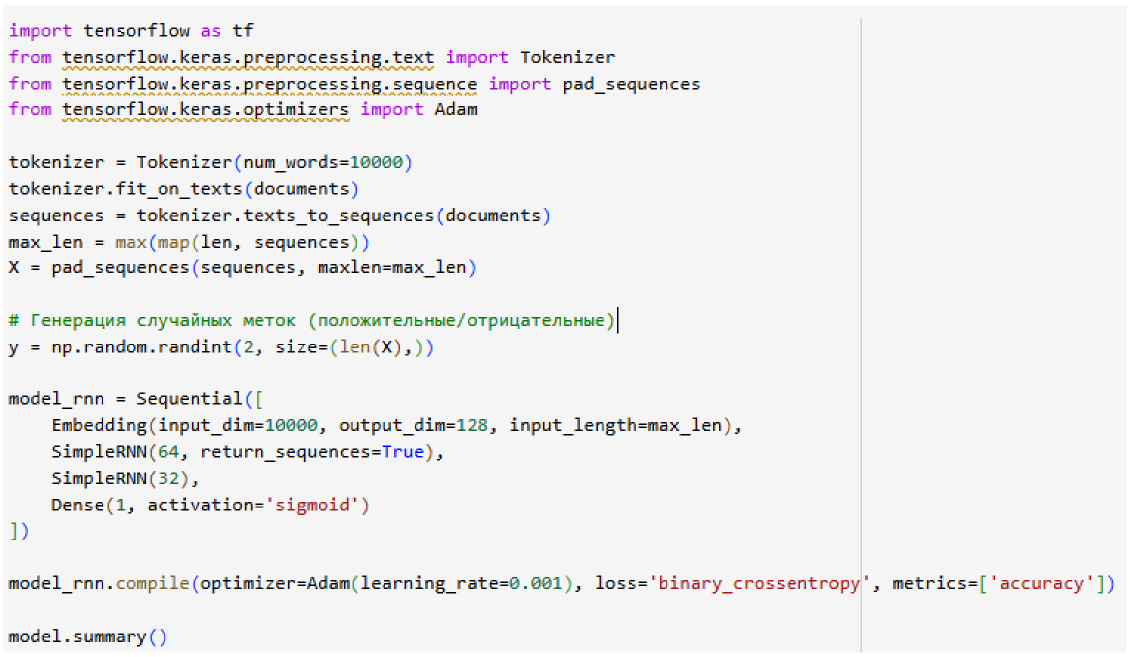


```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense, Embedding

model = Sequential([
    SimpleRNN(64, activation='relu', input_shape=(None, 1)),
    Dense(1)
])

model.compile(optimizer='adam', loss='mse')
model.summary()
```

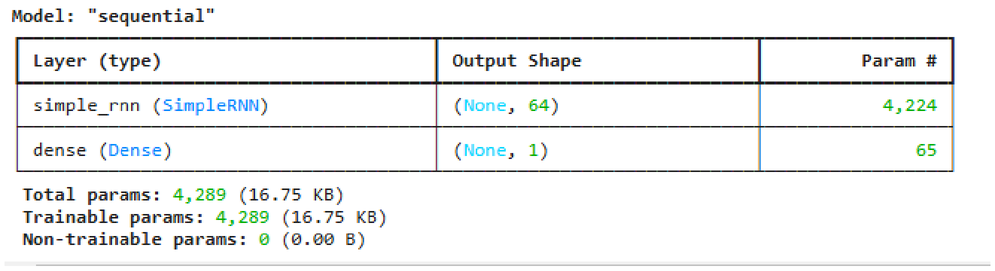


```
Model: "sequential"
┌──────────────────────────────┬─────────────────────────┬──────────────┐
│ Layer (type)                 │ Output Shape            │      Param # │
├──────────────────────────────┼─────────────────────────┼──────────────┤
│ simple_rnn (SimpleRNN)       │ (None, 64)              │        4,224 │
├──────────────────────────────┼─────────────────────────┼──────────────┤
│ dense (Dense)                │ (None, 1)               │           65 │
└──────────────────────────────┴─────────────────────────┴──────────────┘
 Total params: 4,289 (16.75 KB)
 Trainable params: 4,289 (16.75 KB)
 Non-trainable params: 0 (0.00 B)
```

Since the dataset used is on IMDb reviews, we randomly assign 0\1(positive/negative reviews)

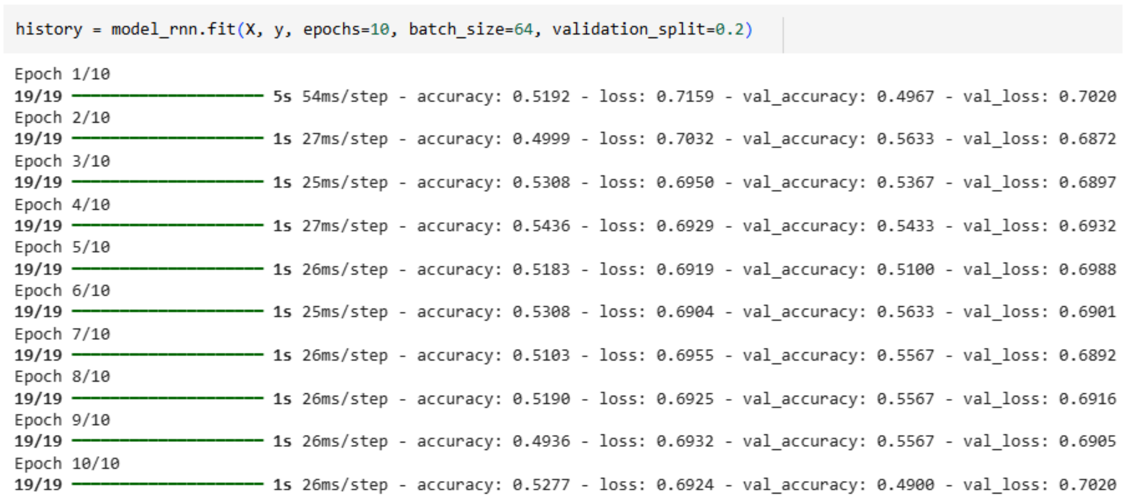


```python
import tensorflow as tf
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow.keras.optimizers import Adam

tokenizer = Tokenizer(num_words=10000)
tokenizer.fit_on_texts(documents)
sequences = tokenizer.texts_to_sequences(documents)
max_len = max(map(len, sequences))
X = pad_sequences(sequences, maxlen=max_len)

# Генерация случайных меток (положительные/отрицательные)
y = np.random.randint(2, size=(len(X),))

model_rnn = Sequential([
    Embedding(input_dim=10000, output_dim=128, input_length=max_len),
    SimpleRNN(64, return_sequences=True),
    SimpleRNN(32),
    Dense(1, activation='sigmoid')
])

model_rnn.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy', metrics=['accuracy'])

model.summary()
```



```
Model: "sequential"
┌──────────────────────────────┬─────────────────────────┬──────────────┐
│ Layer (type)                 │ Output Shape            │      Param # │
├──────────────────────────────┼─────────────────────────┼──────────────┤
│ simple_rnn (SimpleRNN)       │ (None, 64)              │        4,224 │
├──────────────────────────────┼─────────────────────────┼──────────────┤
│ dense (Dense)                │ (None, 1)               │           65 │
└──────────────────────────────┴─────────────────────────┴──────────────┘
 Total params: 4,289 (16.75 KB)
 Trainable params: 4,289 (16.75 KB)
 Non-trainable params: 0 (0.00 B)
```

Then train the model for 10 epochs and track its performance accuracy and loss.



```
history = model_rnn.fit(X, y, epochs=10, batch_size=64, validation_split=0.2)
```

```
Epoch 1/10
19/19 ━━━━━━━━━━━━━━━━━━━━ 5s 54ms/step - accuracy: 0.5192 - loss: 0.7159 - val_accuracy: 0.4967 - val_loss: 0.7020
Epoch 2/10
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.4999 - loss: 0.7032 - val_accuracy: 0.5633 - val_loss: 0.6872
Epoch 3/10
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.5308 - loss: 0.6950 - val_accuracy: 0.5367 - val_loss: 0.6897
Epoch 4/10
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 27ms/step - accuracy: 0.5436 - loss: 0.6929 - val_accuracy: 0.5433 - val_loss: 0.6932
Epoch 5/10
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5183 - loss: 0.6919 - val_accuracy: 0.5100 - val_loss: 0.6988
Epoch 6/10
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 25ms/step - accuracy: 0.5308 - loss: 0.6904 - val_accuracy: 0.5633 - val_loss: 0.6901
Epoch 7/10
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5103 - loss: 0.6955 - val_accuracy: 0.5567 - val_loss: 0.6892
Epoch 8/10
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5190 - loss: 0.6925 - val_accuracy: 0.5567 - val_loss: 0.6916
Epoch 9/10
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.4936 - loss: 0.6932 - val_accuracy: 0.5567 - val_loss: 0.6905
Epoch 10/10
19/19 ━━━━━━━━━━━━━━━━━━━━ 1s 26ms/step - accuracy: 0.5277 - loss: 0.6924 - val_accuracy: 0.4900 - val_loss: 0.7020
```

3.3. Code Snippets: Include code for RNN training and gradient plotting.

From the graph you can see that as the accuracy increases, the loss decreases.

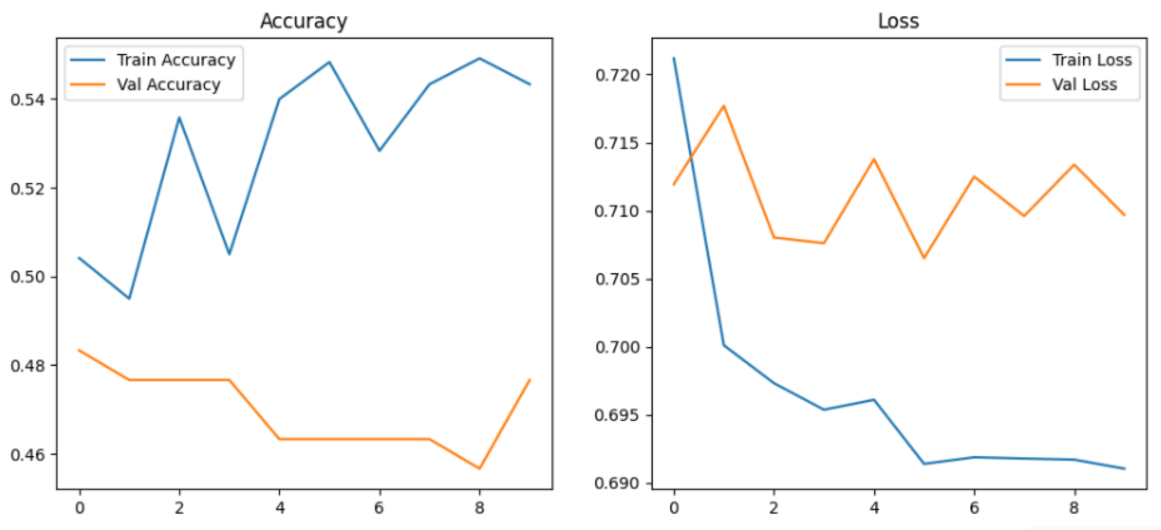


Figure 3: RNN training accuracy vs. loss curve

Writing code to visualize the gradient

```
gradients = []
for epoch in range(5):
    with tf.GradientTape() as tape:
        y_pred = model_rnn(X, training=True)
        # Reshape y_pred to match the shape of y
        y_pred = tf.reshape(y_pred, [-1]) # Reshape to (1500,)
        loss = tf.keras.losses.binary_crossentropy(y, y_pred)
    grads = tape.gradient(loss, model_rnn.trainable_variables)
    gradients.append(np.mean([tf.reduce_mean(tf.abs(g)).numpy() for g in grads]))

plt.plot(range(5), gradients)
plt.xlabel("Epoch")
plt.ylabel("Gradient Magnitude")
plt.title("Vanishing Gradient Problem")
plt.show()
```

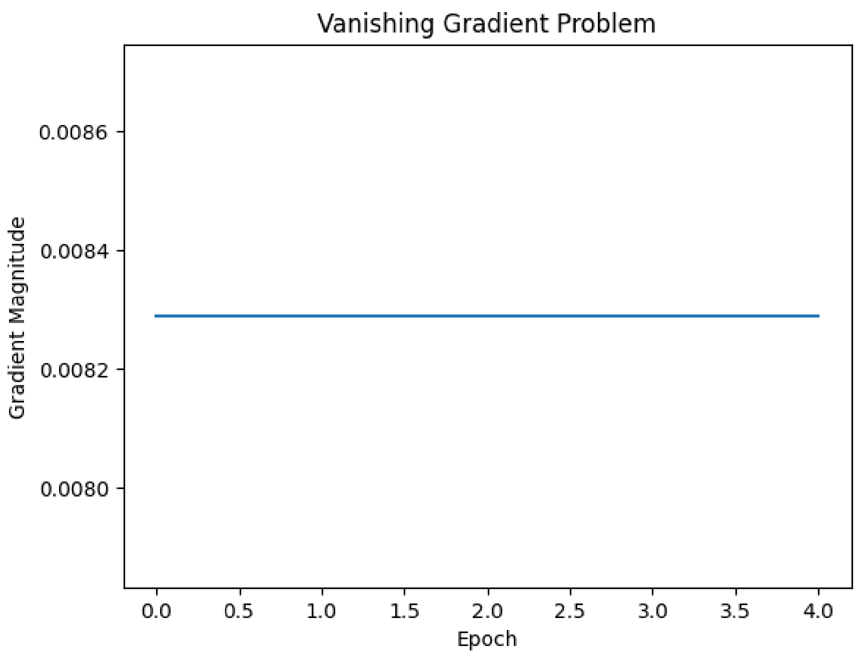
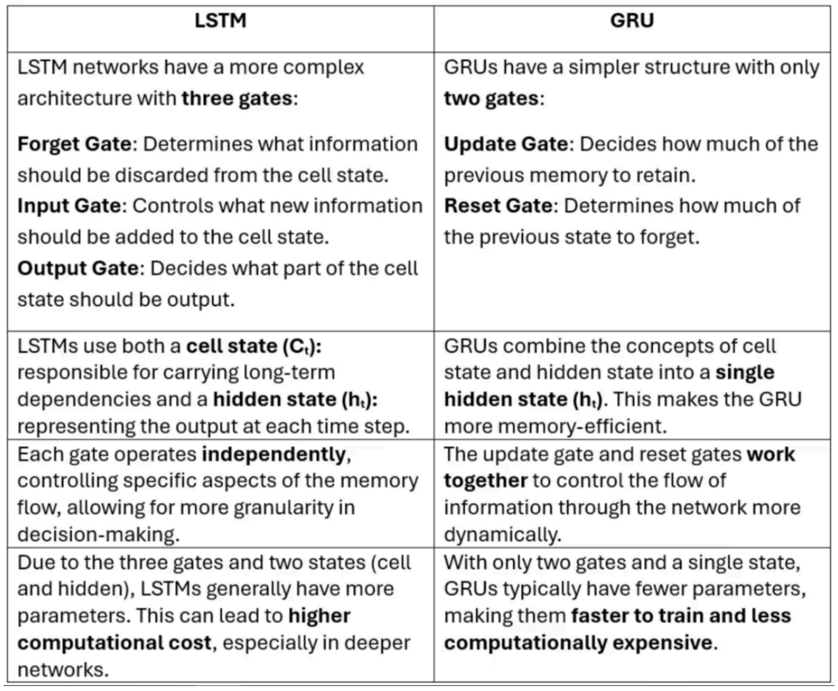


Figure 4: Gradient values over epochs (vanishing gradient problem)

4. LSTM vs. GRU for Text Classification

4.1. Comparison: Discuss the differences between **LSTM** and **GRU** and why both are used for text classification tasks.

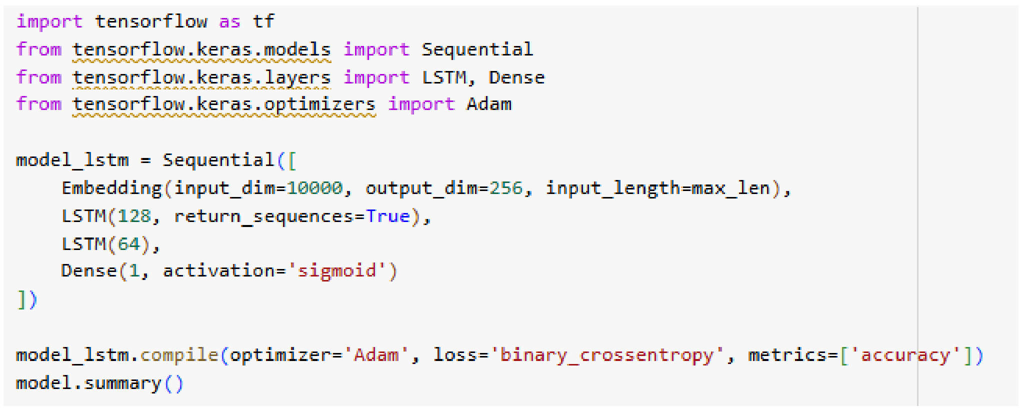
Both LSTM and GRU are designed to address the vanishing gradient problem by enabling the models to retain information over longer sequences. While LSTMs achieve this by having separate gates for different operations (input, forget, and output), GRUs simplify this by combining the two states into a single hidden state, and the two gates (update and reset) allow for dynamic memory control.



LSTM	GRU
LSTM networks have a more complex architecture with **three gates**: **Forget Gate**: Determines what information should be discarded from the cell state. **Input Gate**: Controls what new information should be added to the cell state. **Output Gate**: Decides what part of the cell state should be output.	GRUs have a simpler structure with only **two gates**: **Update Gate**: Decides how much of the previous memory to retain. **Reset Gate**: Determines how much of the previous state to forget.
LSTMs use both a **cell state (C_t):** responsible for carrying long-term dependencies and a **hidden state (h_t):** representing the output at each time step.	GRUs combine the concepts of cell state and hidden state into a **single hidden state (h_t)**. This makes the GRU more memory-efficient.
Each gate operates **independently**, controlling specific aspects of the memory flow, allowing for more granularity in decision-making.	The update gate and reset gates **work together** to control the flow of information through the network more dynamically.
Due to the three gates and two states (cell and hidden), LSTMs generally have more parameters. This can lead to **higher computational cost**, especially in deeper networks.	With only two gates and a single state, GRUs typically have fewer parameters, making them **faster to train and less computationally expensive**.

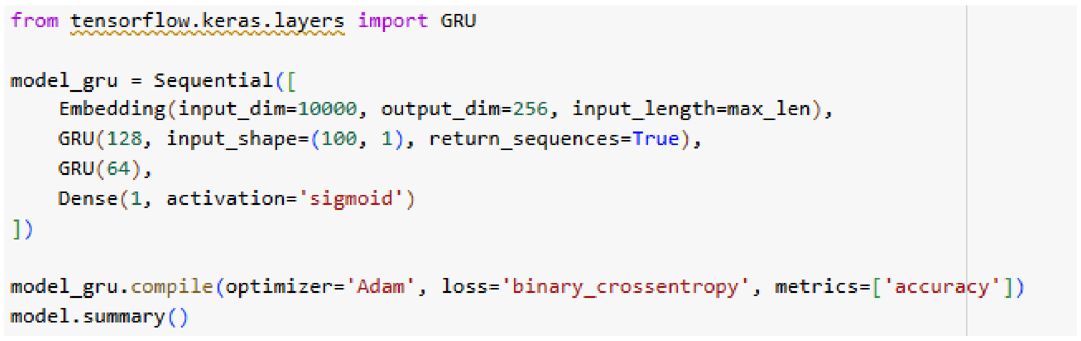
4.2. Training Results: Present the training and validation accuracy and loss comparison for both models.

Creating LSTM model



```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import LSTM, Dense
from tensorflow.keras.optimizers import Adam

model_lstm = Sequential([
    Embedding(input_dim=10000, output_dim=256, input_length=max_len),
    LSTM(128, return_sequences=True),
    LSTM(64),
    Dense(1, activation='sigmoid')
])

model_lstm.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

Creating GRU model



```
from tensorflow.keras.layers import GRU

model_gru = Sequential([
    Embedding(input_dim=10000, output_dim=256, input_length=max_len),
    GRU(128, input_shape=(100, 1), return_sequences=True),
    GRU(64),
    Dense(1, activation='sigmoid')
])

model_gru.compile(optimizer='Adam', loss='binary_crossentropy', metrics=['accuracy'])
model.summary()
```

Training the LSTM model

```
history_lstm = model_lstm.fit(X, y, epochs=10, batch_size=32, validation_split=0.2)
```

```
Epoch 1/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 6s 163ms/step - accuracy: 0.5177 - loss: 0.6933 - val_accuracy: 0.4667 - val_loss: 0.6991
Epoch 2/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 3s 84ms/step - accuracy: 0.5442 - loss: 0.6876 - val_accuracy: 0.5000 - val_loss: 0.6949
Epoch 3/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 3s 73ms/step - accuracy: 0.5441 - loss: 0.6892 - val_accuracy: 0.4733 - val_loss: 0.6977
Epoch 4/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 3s 72ms/step - accuracy: 0.5267 - loss: 0.6899 - val_accuracy: 0.4733 - val_loss: 0.7007
Epoch 5/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 4s 97ms/step - accuracy: 0.5423 - loss: 0.6858 - val_accuracy: 0.5267 - val_loss: 0.6980
Epoch 6/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 4s 69ms/step - accuracy: 0.5301 - loss: 0.6889 - val_accuracy: 0.4733 - val_loss: 0.6989
Epoch 7/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 5s 73ms/step - accuracy: 0.5287 - loss: 0.6905 - val_accuracy: 0.5133 - val_loss: 0.6963
Epoch 8/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 4s 95ms/step - accuracy: 0.5517 - loss: 0.6863 - val_accuracy: 0.4733 - val_loss: 0.6979
Epoch 9/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 4s 66ms/step - accuracy: 0.5125 - loss: 0.6921 - val_accuracy: 0.5000 - val_loss: 0.7008
Epoch 10/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 3s 69ms/step - accuracy: 0.5418 - loss: 0.6877 - val_accuracy: 0.4867 - val_loss: 0.6968
```

Training the GRU model

```
history_gru = model_gru.fit(X, y, epochs=10, batch_size=32, validation_split=0.2)
```

```
Epoch 1/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 9s 79ms/step - accuracy: 0.5230 - loss: 0.6932 - val_accuracy: 0.4667 - val_loss: 0.6951
Epoch 2/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 5s 84ms/step - accuracy: 0.5267 - loss: 0.6891 - val_accuracy: 0.4733 - val_loss: 0.6980
Epoch 3/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 3s 75ms/step - accuracy: 0.5573 - loss: 0.6874 - val_accuracy: 0.4733 - val_loss: 0.6968
Epoch 4/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 2s 63ms/step - accuracy: 0.5500 - loss: 0.6880 - val_accuracy: 0.4733 - val_loss: 0.7011
Epoch 5/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 3s 64ms/step - accuracy: 0.5486 - loss: 0.6857 - val_accuracy: 0.4733 - val_loss: 0.6967
Epoch 6/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 3s 63ms/step - accuracy: 0.5246 - loss: 0.6857 - val_accuracy: 0.4733 - val_loss: 0.6977
Epoch 7/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 4s 95ms/step - accuracy: 0.5513 - loss: 0.6844 - val_accuracy: 0.5133 - val_loss: 0.6952
Epoch 8/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 4s 63ms/step - accuracy: 0.5508 - loss: 0.6877 - val_accuracy: 0.4733 - val_loss: 0.6964
Epoch 9/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 3s 65ms/step - accuracy: 0.5655 - loss: 0.6861 - val_accuracy: 0.5133 - val_loss: 0.6969
Epoch 10/10
38/38 ━━━━━━━━━━━━━━━━━━━━ 2s 64ms/step - accuracy: 0.5430 - loss: 0.6834 - val_accuracy: 0.5267 - val_loss: 0.6970
```

4.3. Code Snippets: Include the code for training and evaluating both LSTM and GRU models.

We output a visualization for a graphical comparison of accuracy for LSTM and GRU models

```
plt.figure(figsize=(10, 5))
plt.plot(history_lstm.history['val_accuracy'], label='LSTM Accuracy')
plt.plot(history_gru.history['val_accuracy'], label='GRU Accuracy')
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend()
plt.title("LSTM vs GRU Performance Comparison")
plt.show()
```

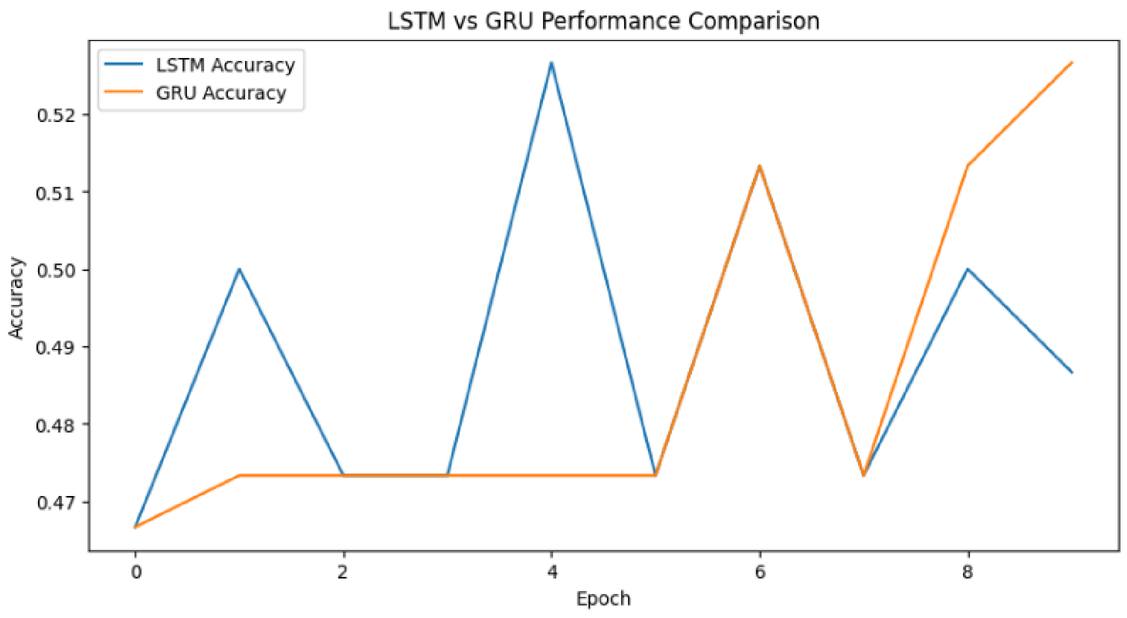


Figure 5: LSTM vs. GRU accuracy comparison

We output a visualization for a graphical comparison of loss for LSTM and GRU models

```
plt.figure(figsize=(10, 5))
plt.plot(history_lstm.history['val_loss'], label='LSTM Accuracy')
plt.plot(history_gru.history['val_loss'], label='GRU Accuracy')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend()
plt.title("LSTM vs GRU Performance Comparison")
plt.show()
```

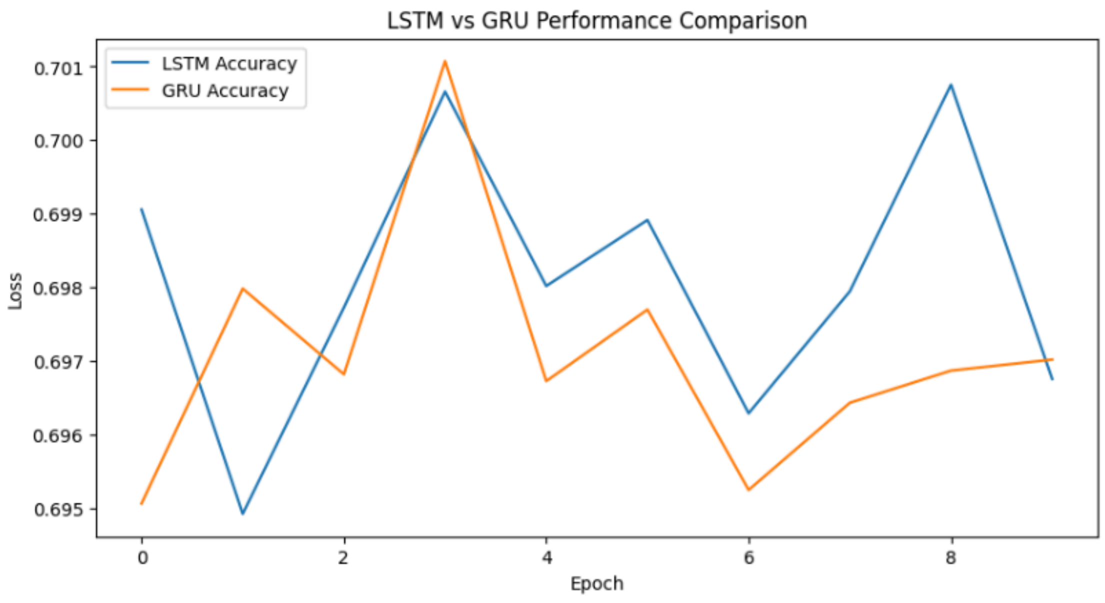


Figure 6: LSTM vs. GRU loss comparison

5. Text Generation with LSTM

5.1. Describe the LSTM architecture used for text generation

The architecture of an LSTM can be visualized as a series of repeating “blocks” or “cells”, each of which contains a set of interconnected nodes. Here’s a high-level overview of the architecture:

1. **Input**: At each time step, the LSTM takes in an input vector, x_t, which represents the current observation or token in the sequence.
2. **Hidden State**: The LSTM maintains a hidden state vector, h_t, which represents the current “memory” of the network. The hidden state is initialized to a vector of zeros at the beginning of the sequence.
3. **Cell State**: The LSTM also maintains a cell state vector, c_t, which is responsible for storing long-term information over the course of the sequence. The cell state is initialized to a vector of zeros at the beginning of the sequence.
4. **Gates**: The LSTM uses three types of gates to control the flow of information through the network:

a). Forget Gate: This gate takes in the previous hidden state, h_{t-1}, and the. current input, x_t, and outputs a vector of values between 0 and 1 that represent how much of the previous cell state to “forget” and how much to retain. This gate allows the LSTM to selectively “erase” or “remember” information from the previous time step.

b). Input Gate: This gate takes in the previous hidden state, h_{t-1}, and the current input, x_t, and outputs a vector of values between 0 and 1 that represent how much of the current input to add to the cell state. This gate allows the LSTM to selectively “add” or “discard” new information to the cell state.

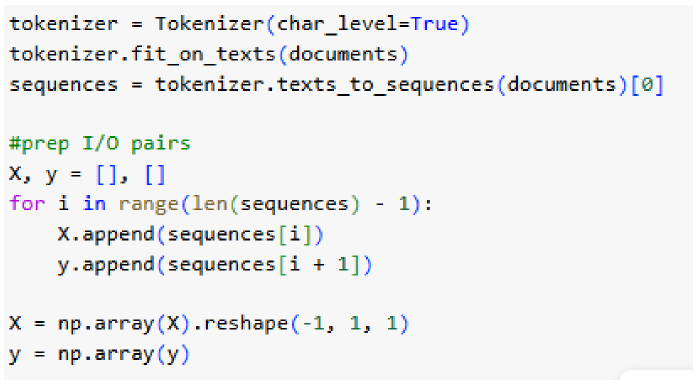
c). Output Gate: This gate takes in the previous hidden state, h_{t-1}, and the current input, x_t, and the current cell state, c_t, and outputs a vector of values between 0 and 1 that represent how much of the current cell state to output as the current hidden state, h_t. This gate allows the LSTM to selectively “focus” or “ignore” certain parts of the cell state when computing the output.

5. **Output**: At each time step, the LSTM outputs a vector, y_t, that represents the network’s prediction or encoding of the current input.

The combination of the cell state, hidden state, and gates allows the LSTM to selectively “remember” or “forget” information over time, making it well-suited for tasks that require modeling long-term dependencies or sequences.

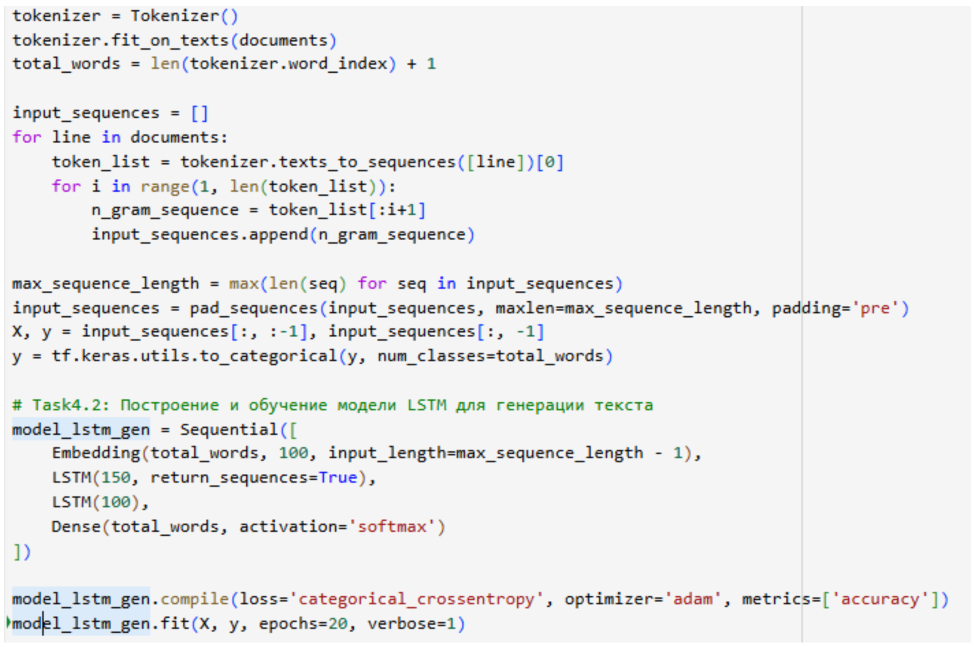
5.2. **Generated Text**: Present examples of the text generated by the LSTM model based on different seed texts.

Tokenize the corpus and prepare input-output sequences



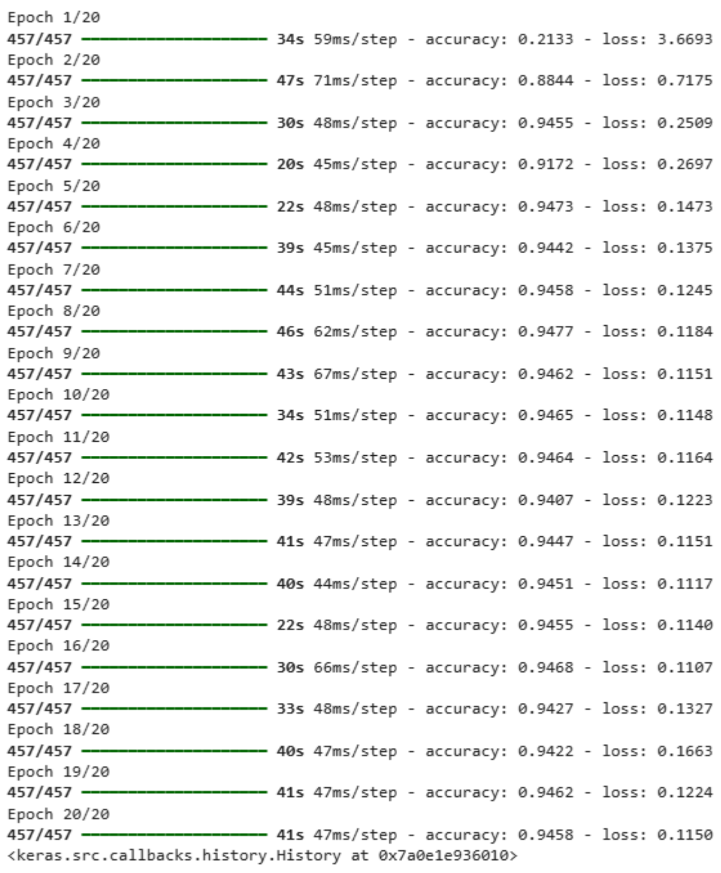
```
tokenizer = Tokenizer(char_level=True)
tokenizer.fit_on_texts(documents)
sequences = tokenizer.texts_to_sequences(documents)[0]

#prep I/O pairs
X, y = [], []
for i in range(len(sequences) - 1):
    X.append(sequences[i])
    y.append(sequences[i + 1])

X = np.array(X).reshape(-1, 1, 1)
y = np.array(y)
```

Here we prepare the data for LSTM text generation and train it on the dataset.



```
tokenizer = Tokenizer()
tokenizer.fit_on_texts(documents)
total_words = len(tokenizer.word_index) + 1

input_sequences = []
for line in documents:
    token_list = tokenizer.texts_to_sequences([line])[0]
    for i in range(1, len(token_list)):
        n_gram_sequence = token_list[:i+1]
        input_sequences.append(n_gram_sequence)

max_sequence_length = max(len(seq) for seq in input_sequences)
input_sequences = pad_sequences(input_sequences, maxlen=max_sequence_length, padding='pre')
X, y = input_sequences[:, :-1], input_sequences[:, -1]
y = tf.keras.utils.to_categorical(y, num_classes=total_words)

# Task4.2: Построение и обучение модели LSTM для генерации текста
model_lstm_gen = Sequential([
    Embedding(total_words, 100, input_length=max_sequence_length - 1),
    LSTM(150, return_sequences=True),
    LSTM(100),
    Dense(total_words, activation='softmax')
])

model_lstm_gen.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
model_lstm_gen.fit(X, y, epochs=20, verbose=1)
```

The training was successful using the example of 20

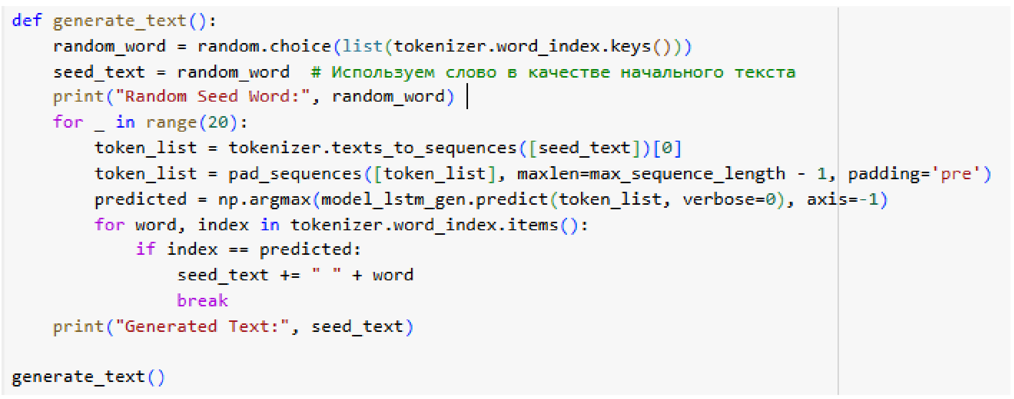


```
Epoch 1/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 34s 59ms/step - accuracy: 0.2133 - loss: 3.6693
Epoch 2/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 47s 71ms/step - accuracy: 0.8844 - loss: 0.7175
Epoch 3/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 30s 48ms/step - accuracy: 0.9455 - loss: 0.2509
Epoch 4/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 20s 45ms/step - accuracy: 0.9172 - loss: 0.2697
Epoch 5/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 22s 48ms/step - accuracy: 0.9473 - loss: 0.1473
Epoch 6/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 39s 45ms/step - accuracy: 0.9442 - loss: 0.1375
Epoch 7/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 44s 51ms/step - accuracy: 0.9458 - loss: 0.1245
Epoch 8/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 46s 62ms/step - accuracy: 0.9477 - loss: 0.1184
Epoch 9/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 43s 67ms/step - accuracy: 0.9462 - loss: 0.1151
Epoch 10/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 34s 51ms/step - accuracy: 0.9465 - loss: 0.1148
Epoch 11/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 42s 53ms/step - accuracy: 0.9464 - loss: 0.1164
Epoch 12/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 39s 48ms/step - accuracy: 0.9407 - loss: 0.1223
Epoch 13/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 41s 47ms/step - accuracy: 0.9447 - loss: 0.1151
Epoch 14/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 40s 44ms/step - accuracy: 0.9451 - loss: 0.1117
Epoch 15/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 22s 48ms/step - accuracy: 0.9455 - loss: 0.1140
Epoch 16/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 30s 66ms/step - accuracy: 0.9468 - loss: 0.1107
Epoch 17/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 33s 48ms/step - accuracy: 0.9427 - loss: 0.1327
Epoch 18/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 40s 47ms/step - accuracy: 0.9422 - loss: 0.1663
Epoch 19/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 41s 47ms/step - accuracy: 0.9462 - loss: 0.1224
Epoch 20/20
457/457 ━━━━━━━━━━━━━━━━━━━━ 41s 47ms/step - accuracy: 0.9458 - loss: 0.1150
<keras.src.callbacks.history.History at 0x7a0e1e936010>
```

5.3. Code Snippets: Include the code for text generation.

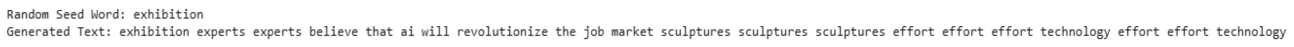
The dataset based on Shakespeare's poems took a long time to generate and the data flew off. Because of this, I took my previously used dataset.

We use any random word to generate it.



```python
def generate_text():
    random_word = random.choice(list(tokenizer.word_index.keys()))
    seed_text = random_word  # Используем слово в качестве начального текста
    print("Random Seed Word:", random_word)
    for _ in range(20):
        token_list = tokenizer.texts_to_sequences([seed_text])[0]
        token_list = pad_sequences([token_list], maxlen=max_sequence_length - 1, padding='pre')
        predicted = np.argmax(model_lstm_gen.predict(token_list, verbose=0), axis=-1)
        for word, index in tokenizer.word_index.items():
            if index == predicted:
                seed_text += " " + word
                break
    print("Generated Text:", seed_text)

generate_text()
```

Output:



```
Random Seed Word: exhibition
Generated Text: exhibition experts experts believe that ai will revolutionize the job market sculptures sculptures sculptures effort effort effort technology effort effort technology
```

Figure 8: Generated text from a different seed text

6. **Bidirectional LSTM for Improved Performance**

6.1. Discuss the modification made to the LSTM model by using a Bidirectional LSTM layer.

A **Bidirectional LSTM (BiLSTM)** is a modification of the standard Long Short-Term Memory (LSTM) network that improves the model's ability to capture dependencies in sequential data. This modification significantly enhances performance, especially in natural language processing (NLP) tasks, where contextual understanding of both past and future words is essential.

Modifications Introduced by Bidirectional LSTM

1.Dual LSTM Layers:

Instead of a single LSTM layer processing the sequence in a forward direction (left to right), a second LSTM layer processes the same sequence in the reverse direction (right to left).

These two layers run in parallel and independently.

2.Concatenation of Outputs:

The outputs of the forward and backward LSTM layers are concatenated at each time step, creating a richer representation that considers both past and future context.

Mathematically, if the forward LSTM produces an output ht→h_t^{\rightarrow}ht→ and the backward LSTM produces ht←h_t^{\leftarrow}ht←, the final output is: ht=[ht→;ht←]h_t = [h_t^{\rightarrow}; h_t^{\leftarrow}]ht=[ht→;ht←]

This improves the model’s ability to understand dependencies in a sequence.

3.Enhanced Context Awareness:

A standard LSTM can only rely on previous time steps (past information).

A BiLSTM integrates future context by processing data from both directions.

This is particularly beneficial for text-related tasks like machine translation, speech recognition, and named entity recognition (NER).

4.Increased Computation and Memory Usage:

Since BiLSTM effectively doubles the number of parameters (due to the two LSTM layers), it requires more computation and memory.

However, this trade-off is often justified by the performance improvement.

6.2. Comparison: Present the performance improvement, if any, over the standard LSTM model.

Implementing a Bidirectional LSTM layer in the existing LSTM-based model.

```python
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Embedding, LSTM, GRU, Bidirectional, Dense

def build_model(cell_type="LSTM", bidirectional=False):
    model = Sequential([
        Embedding(input_dim=num_words, output_dim=32, input_length=max_length),
        Bidirectional(LSTM(32)) if bidirectional else (LSTM(32) if cell_type == "LSTM" else GRU(32)),
        Dense(1, activation='sigmoid')
    ])
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model

# Создание моделей
bi_lstm_model = build_model("LSTM", bidirectional=True)
```

We are training a model for plotting.

```python
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split

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

def train_and_plot(model, name):
    # Changed x_train, y_train, x_test to X_train, y_train, X_test
    history = model.fit(X_train, y_train, epochs=10, batch_size=64, validation_data=(X_test, y_test))
    plt.plot(history.history['accuracy'], label=f'{name} Train Accuracy')
    return history

# Обучение моделей
plt.figure(figsize=(10, 5))
history_lstm = train_and_plot(lstm_model, "LSTM")
bi_lstm_history = train_and_plot(bi_lstm_model, "Bidirectional LSTM")
```

6.3. Code Snippets: Include the code for the Bidirectional LSTM model.

We build a visualization to compare the two models.

```python
plt.legend()
plt.title("Comparison LSTM и Bidirectional LSTM")
plt.show()
```

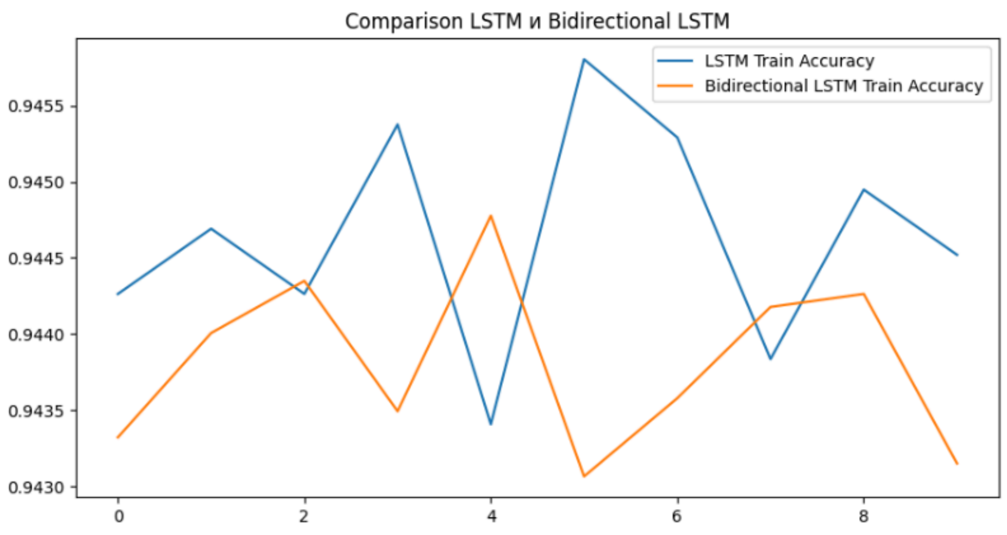


Figure 9: Comparison of Bidirectional LSTM and LSTM model performance

7. Conclusion

Throughout our research on NLP models, we have explored various methods of text representation and generation, including word embedding (Word2Vec, GloVe, FastText) and advanced deep learning architectures such as RNN, LSTM, GRU, and Bidirectional LSTM. Based on our laboratory work, several aspects have been identified:

Word2Vec effectively captures semantic connections, but does not use vocabulary processing (OOV).

GloVe provides meaningful global representations of matches, but requires careful configuration of the matrix factorization parameters.

FastText surpasses them by including information about subwords, which makes it more resistant to rare words and morphological variations.

Standard RNNs face long-term dependencies due to vanishing gradient issues.

LSTM eliminates this problem by using closed mechanisms, resulting in improved performance when performing sequential tasks.

GRU offers a simplified alternative to LSTM with comparable results that are often achieved faster.

Bidirectional LSTM improves context understanding by processing input data both forward and backward, which proves effective in sentiment analysis and sequence labeling tasks.

Learning LSTM for text generation requires careful tuning of sequence length and temperature to balance consistency and variety of output.

Retraining remains a difficult task, requiring the use of regularization methods such as dropout.

In the future, to improve the work, you can increase the speed of work, adjust the alignment parameters, and work on accuracy. For improved architectures Implement transformer-based models (for example, BERT, GPT) to compare their effectiveness with traditional architectures. Explore hybrid models combining CNN and RNN to improve contextual representation. For greater efficiency, transfer learning can be used to use pre-trained implementations for faster convergence.

By integrating these improvements, we can increase the efficiency and scalability of NLP models for word processing tasks.

8. References

1. https://medium.com/analytics-vidhya/word-embeddings-in-nlp-word2vec-glove-fasttext-24d4d4286a73
2. https://habr.com/ru/articles/801807/
3. https://www.restack.io/p/simple-recurrent-neural-network-answer-nlp-task-design-cat-ai
4. https://medium.com/@sudeshnasen/understanding-the-differences-between-lstm-and-gru-23d09abdd998
5. Understanding LSTM: Architecture, Pros and Cons, and Implementation | by Anishnama | Medium