|  | **Министерство науки и высшего образования Российской Федерации**  **Федеральное государственное бюджетное образовательное учреждение**  **высшего образования**  **«Московский государственный технический университет**  **имени Н.Э. Баумана**  **(национальный исследовательский университет)»**  **(МГТУ им. Н.Э. Баумана)** |
| --- | --- |

ФАКУЛЬТЕТ Информатики и систем управления

КАФЕДРА Теоретической информатики и компьютерных технологий

**Домашнее задание № 6**

**Рекуррентные нейронные сети (RNN, GRU, LSTM).**

**Классификация текстов**

**ПО КУРСУ:**

***«Теория искусственных нейронных сетей»***

Студент *Караник А. А.*

Преподаватель *Каганов Ю. Т.*

*Москва, 2024 г.*

**ОГЛАВЛЕНИЕ**

[Цель работы 3](#_Toc185550460)

[Постановка задачи 3](#_Toc185550461)

[Практическая реализация 3](#_Toc185550462)

[Результаты 9](#_Toc185550463)

[Вывод 11](#_Toc185550464)

# Цель работы

1. Изучение архитектур рекуррентных нейронных сетей: RNN, GRU, LSTM.

2. Реализовать нейронные сети на основе данных архитектур с использованием фреймворка PyTorch и обучить модели для задачи классификации отзывов на основе базы данных IMDB.

# Постановка задачи

1. Требуется обучить рекуррентные нейронные сети RNN, GRU, LSTM на основе базы данных отзывов IMDB с использованием различных оптимизаторов(SGD, AdaDelta, NAG, Adam).

2. Необходимо также сравнить полученные результаты и выделить лучшие модели.

# Практическая реализация

Исходный текст программы на языке программирования Python:

import math

import torch

import torch.nn as nn

import torch.nn.functional as F

from torch.utils.data import DataLoader

from torchtext.datasets import IMDB

from torchtext.data.utils import get\_tokenizer

from torchtext.vocab import build\_vocab\_from\_iterator

import time

import matplotlib.pyplot as plt

embedding\_size = 64

h\_size = 8

max\_length = 200

number\_of\_layers = 25

batch\_size = 1000

epochs = 5

device = torch.device("cuda" if torch.cuda.is\_available() else "cpu")

def prepare(text):

    tokens = tokenizer(text)

    token\_ids = vocab(tokens)

    if len(token\_ids) > max\_length:

        token\_ids = token\_ids[:max\_length]

    else:

        token\_ids = token\_ids + [0]\*(max\_length - len(token\_ids))

    return torch.tensor(token\_ids, dtype=torch.long)

def yield\_tokens(data\_iter):

    for \_, line in data\_iter:

        yield tokenizer(line)

def collate\_fn(batch):

    texts, labels = [], []

    for label, text in batch:

        texts.append(prepare(text))

        labels.append(1 if label == 'pos' else 0)

    return torch.stack(texts), torch.tensor(labels, dtype=torch.float)

train\_iter, test\_iter = IMDB(split=('train', 'test'))

tokenizer = get\_tokenizer('basic\_english')

vocab = build\_vocab\_from\_iterator(yield\_tokens(train\_iter), specials=["<unk>"])

vocab.set\_default\_index(vocab["<unk>"])

train\_loader = DataLoader(list(train\_iter), batch\_size=batch\_size, shuffle=True, collate\_fn=collate\_fn)

test\_loader = DataLoader(list(test\_iter), batch\_size=batch\_size, shuffle=False, collate\_fn=collate\_fn)

class RNNModel(nn.Module):

    def \_\_init\_\_(self, vocab\_size, embedding\_size, h\_size, output\_size=1, number\_of\_layers=25):

        super(RNNModel, self).\_\_init\_\_()

        self.h\_size = h\_size

        self.number\_of\_layers = number\_of\_layers

        self.embedding = nn.Embedding(vocab\_size, embedding\_size)

        W\_xh\_layers = [nn.Parameter(torch.Tensor(embedding\_size, h\_size))]

        W\_hh\_layers = [nn.Parameter(torch.Tensor(h\_size, h\_size))]

        b\_h\_layers = [nn.Parameter(torch.Tensor(h\_size))]

        for \_ in range(number\_of\_layers - 1):

            W\_xh\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            W\_hh\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            b\_h\_layers.append(nn.Parameter(torch.Tensor(h\_size)))

        self.W\_xh\_layers = nn.ParameterList(W\_xh\_layers)

        self.W\_hh\_layers = nn.ParameterList(W\_hh\_layers)

        self.b\_h\_layers = nn.ParameterList(b\_h\_layers)

        self.W\_hy = nn.Parameter(torch.Tensor(h\_size, output\_size))

        self.b\_y = nn.Parameter(torch.Tensor(output\_size))

        self.reset\_parameters()

    def reset\_parameters(self):

        for i in range(self.number\_of\_layers):

            nn.init.kaiming\_uniform\_(self.W\_xh\_layers[i], a=math.sqrt(5))

            nn.init.kaiming\_uniform\_(self.W\_hh\_layers[i], a=math.sqrt(5))

            nn.init.zeros\_(self.b\_h\_layers[i])

        nn.init.kaiming\_uniform\_(self.W\_hy, a=math.sqrt(5))

        nn.init.zeros\_(self.b\_y)

    def forward(self, x, h\_0=None):

        batch\_size, seq\_len = x.size()

        if h\_0 is None:

            h\_0 = torch.zeros(self.number\_of\_layers, batch\_size, self.h\_size, device=x.device)

        embedded = self.embedding(x)

        h\_t\_layers = [h\_0[i] for i in range(self.number\_of\_layers)]

        for t in range(seq\_len):

            input\_t = embedded[:, t, :]

            for i in range(self.number\_of\_layers):

                h\_t\_layers[i] = torch.tanh(input\_t @ self.W\_xh\_layers[i] + h\_t\_layers[i] @ self.W\_hh\_layers[i] + self.b\_h\_layers[i])

                input\_t = h\_t\_layers[i]

        logits = input\_t @ self.W\_hy + self.b\_y

        h\_last = torch.stack(h\_t\_layers, dim=0)

        return logits.squeeze(1), h\_last

class LSTMModel(nn.Module):

    def \_\_init\_\_(self, vocab\_size, embedding\_size, h\_size, output\_size=1, number\_of\_layers=25):

        super(LSTMModel, self).\_\_init\_\_()

        self.h\_size = h\_size

        self.number\_of\_layers = number\_of\_layers

        self.embedding = nn.Embedding(vocab\_size, embedding\_size)

        W\_xf\_layers = [nn.Parameter(torch.Tensor(embedding\_size, h\_size))]

        W\_hf\_layers = [nn.Parameter(torch.Tensor(h\_size, h\_size))]

        b\_f\_layers = [nn.Parameter(torch.Tensor(h\_size))]

        W\_xi\_layers = [nn.Parameter(torch.Tensor(embedding\_size, h\_size))]

        W\_hi\_layers = [nn.Parameter(torch.Tensor(h\_size, h\_size))]

        b\_i\_layers = [nn.Parameter(torch.Tensor(h\_size))]

        W\_xC\_layers = [nn.Parameter(torch.Tensor(embedding\_size, h\_size))]

        W\_hC\_layers = [nn.Parameter(torch.Tensor(h\_size, h\_size))]

        b\_C\_layers = [nn.Parameter(torch.Tensor(h\_size))]

        W\_xo\_layers = [nn.Parameter(torch.Tensor(embedding\_size, h\_size))]

        W\_ho\_layers = [nn.Parameter(torch.Tensor(h\_size, h\_size))]

        b\_o\_layers = [nn.Parameter(torch.Tensor(h\_size))]

        for \_ in range(number\_of\_layers - 1):

            W\_xf\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            W\_hf\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            b\_f\_layers.append(nn.Parameter(torch.Tensor(h\_size)))

            W\_xi\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            W\_hi\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            b\_i\_layers.append(nn.Parameter(torch.Tensor(h\_size)))

            W\_xC\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            W\_hC\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            b\_C\_layers.append(nn.Parameter(torch.Tensor(h\_size)))

            W\_xo\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            W\_ho\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            b\_o\_layers.append(nn.Parameter(torch.Tensor(h\_size)))

        self.W\_xf\_layers = nn.ParameterList(W\_xf\_layers)

        self.W\_hf\_layers = nn.ParameterList(W\_hf\_layers)

        self.b\_f\_layers = nn.ParameterList(b\_f\_layers)

        self.W\_xi\_layers = nn.ParameterList(W\_xi\_layers)

        self.W\_hi\_layers = nn.ParameterList(W\_hi\_layers)

        self.b\_i\_layers = nn.ParameterList(b\_i\_layers)

        self.W\_xC\_layers = nn.ParameterList(W\_xC\_layers)

        self.W\_hC\_layers = nn.ParameterList(W\_hC\_layers)

        self.b\_C\_layers = nn.ParameterList(b\_C\_layers)

        self.W\_xo\_layers = nn.ParameterList(W\_xo\_layers)

        self.W\_ho\_layers = nn.ParameterList(W\_ho\_layers)

        self.b\_o\_layers = nn.ParameterList(b\_o\_layers)

        self.W\_hy = nn.Parameter(torch.Tensor(h\_size, output\_size))

        self.b\_y = nn.Parameter(torch.Tensor(output\_size))

        self.reset\_parameters()

    def reset\_parameters(self):

        for i in range(self.number\_of\_layers):

            nn.init.kaiming\_uniform\_(self.W\_xf\_layers[i], a=math.sqrt(5))

            nn.init.kaiming\_uniform\_(self.W\_hf\_layers[i], a=math.sqrt(5))

            nn.init.zeros\_(self.b\_f\_layers[i])

            nn.init.kaiming\_uniform\_(self.W\_xi\_layers[i], a=math.sqrt(5))

            nn.init.kaiming\_uniform\_(self.W\_hi\_layers[i], a=math.sqrt(5))

            nn.init.zeros\_(self.b\_i\_layers[i])

            nn.init.kaiming\_uniform\_(self.W\_xC\_layers[i], a=math.sqrt(5))

            nn.init.kaiming\_uniform\_(self.W\_hC\_layers[i], a=math.sqrt(5))

            nn.init.zeros\_(self.b\_C\_layers[i])

            nn.init.kaiming\_uniform\_(self.W\_xo\_layers[i], a=math.sqrt(5))

            nn.init.kaiming\_uniform\_(self.W\_ho\_layers[i], a=math.sqrt(5))

            nn.init.zeros\_(self.b\_o\_layers[i])

        nn.init.kaiming\_uniform\_(self.W\_hy, a=math.sqrt(5))

        nn.init.zeros\_(self.b\_y)

    def forward(self, x, h\_0=None, c\_0=None):

        batch\_size, seq\_len = x.size()

        if h\_0 is None:

            h\_0 = torch.zeros(self.number\_of\_layers, batch\_size, self.h\_size, device=x.device)

        if c\_0 is None:

            c\_0 = torch.zeros(self.number\_of\_layers, batch\_size, self.h\_size, device=x.device)

        embedded = self.embedding(x)

        h\_t\_layers = [h\_0[i] for i in range(self.number\_of\_layers)]

        c\_t\_layers = [c\_0[i] for i in range(self.number\_of\_layers)]

        for t in range(seq\_len):

            input\_t = embedded[:, t, :]

            for i in range(self.number\_of\_layers):

                f\_t\_layer = torch.sigmoid(input\_t @ self.W\_xf\_layers[i] + h\_t\_layers[i] @ self.W\_hf\_layers[i] + self.b\_f\_layers[i])

                i\_t\_layer = torch.sigmoid(input\_t @ self.W\_xi\_layers[i] + h\_t\_layers[i] @ self.W\_hi\_layers[i] + self.b\_i\_layers[i])

                C\_t\_layer = torch.tanh(input\_t @ self.W\_xC\_layers[i] + h\_t\_layers[i] @ self.W\_hC\_layers[i] + self.b\_C\_layers[i])

                o\_t\_layer = torch.sigmoid(input\_t @ self.W\_xo\_layers[i] + h\_t\_layers[i] @ self.W\_ho\_layers[i] + self.b\_o\_layers[i])

                c\_t\_layers[i] = c\_t\_layers[i] \* f\_t\_layer + i\_t\_layer \* C\_t\_layer

                h\_t\_layers[i] = o\_t\_layer \* torch.tanh(c\_t\_layers[i])

                input\_t = h\_t\_layers[i]

        logits = input\_t @ self.W\_hy + self.b\_y

        h\_last = torch.stack(h\_t\_layers, dim=0)

        return logits.squeeze(1), h\_last

class GRUModel(nn.Module):

    def \_\_init\_\_(self, vocab\_size, embedding\_size, h\_size, output\_size=1, number\_of\_layers=25):

        super(GRUModel, self).\_\_init\_\_()

        self.h\_size = h\_size

        self.number\_of\_layers = number\_of\_layers

        self.embedding = nn.Embedding(vocab\_size, embedding\_size)

        W\_xz\_layers = [nn.Parameter(torch.Tensor(embedding\_size, h\_size))]

        W\_hz\_layers = [nn.Parameter(torch.Tensor(h\_size, h\_size))]

        b\_z\_layers = [nn.Parameter(torch.Tensor(h\_size))]

        W\_xr\_layers = [nn.Parameter(torch.Tensor(embedding\_size, h\_size))]

        W\_hr\_layers = [nn.Parameter(torch.Tensor(h\_size, h\_size))]

        b\_r\_layers = [nn.Parameter(torch.Tensor(h\_size))]

        W\_xH\_layers = [nn.Parameter(torch.Tensor(embedding\_size, h\_size))]

        W\_rH\_layers = [nn.Parameter(torch.Tensor(h\_size, h\_size))]

        b\_H\_layers = [nn.Parameter(torch.Tensor(h\_size))]

        for \_ in range(number\_of\_layers - 1):

            W\_xz\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            W\_hz\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            b\_z\_layers.append(nn.Parameter(torch.Tensor(h\_size)))

            W\_xr\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            W\_hr\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            b\_r\_layers.append(nn.Parameter(torch.Tensor(h\_size)))

            W\_xH\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            W\_rH\_layers.append(nn.Parameter(torch.Tensor(h\_size, h\_size)))

            b\_H\_layers.append(nn.Parameter(torch.Tensor(h\_size)))

        self.W\_xz\_layers = nn.ParameterList(W\_xz\_layers)

        self.W\_hz\_layers = nn.ParameterList(W\_hz\_layers)

        self.b\_z\_layers = nn.ParameterList(b\_z\_layers)

        self.W\_xr\_layers = nn.ParameterList(W\_xr\_layers)

        self.W\_hr\_layers = nn.ParameterList(W\_hr\_layers)

        self.b\_r\_layers = nn.ParameterList(b\_r\_layers)

        self.W\_xH\_layers = nn.ParameterList(W\_xH\_layers)

        self.W\_rH\_layers = nn.ParameterList(W\_rH\_layers)

        self.b\_H\_layers = nn.ParameterList(b\_H\_layers)

        self.W\_hy = nn.Parameter(torch.Tensor(h\_size, output\_size))

        self.b\_y = nn.Parameter(torch.Tensor(output\_size))

        self.reset\_parameters()

    def reset\_parameters(self):

        for i in range(self.number\_of\_layers):

            nn.init.kaiming\_uniform\_(self.W\_xz\_layers[i], a=math.sqrt(5))

            nn.init.kaiming\_uniform\_(self.W\_hz\_layers[i], a=math.sqrt(5))

            nn.init.zeros\_(self.b\_z\_layers[i])

            nn.init.kaiming\_uniform\_(self.W\_xr\_layers[i], a=math.sqrt(5))

            nn.init.kaiming\_uniform\_(self.W\_hr\_layers[i], a=math.sqrt(5))

            nn.init.zeros\_(self.b\_r\_layers[i])

            nn.init.kaiming\_uniform\_(self.W\_xH\_layers[i], a=math.sqrt(5))

            nn.init.kaiming\_uniform\_(self.W\_rH\_layers[i], a=math.sqrt(5))

            nn.init.zeros\_(self.b\_H\_layers[i])

        nn.init.kaiming\_uniform\_(self.W\_hy, a=math.sqrt(5))

        nn.init.zeros\_(self.b\_y)

    def forward(self, x, h\_0=None):

        batch\_size, seq\_len = x.size()

        if h\_0 is None:

            h\_0 = torch.zeros(self.number\_of\_layers, batch\_size, self.h\_size, device=x.device)

        embedded = self.embedding(x)

        h\_t\_layers = [h\_0[i] for i in range(self.number\_of\_layers)]

        for t in range(seq\_len):

            input\_t = embedded[:, t, :]

            for i in range(self.number\_of\_layers):

                z\_t\_layer = torch.sigmoid(input\_t @ self.W\_xz\_layers[i] + h\_t\_layers[i] @ self.W\_hz\_layers[i] + self.b\_z\_layers[i])

                r\_t\_layer = torch.sigmoid(input\_t @ self.W\_xr\_layers[i] + h\_t\_layers[i] @ self.W\_hr\_layers[i] + self.b\_r\_layers[i])

                H\_t\_layer = torch.tanh(input\_t @ self.W\_xH\_layers[i] + (h\_t\_layers[i] \* r\_t\_layer) @ self.W\_rH\_layers[i] + self.b\_H\_layers[i])

                h\_t\_layers[i] = h\_t\_layers[i] \* (1 - z\_t\_layer) + z\_t\_layer \* H\_t\_layer

                input\_t = h\_t\_layers[i]

        logits = input\_t @ self.W\_hy + self.b\_y

        h\_last = torch.stack(h\_t\_layers, dim=0)

        return logits.squeeze(1), h\_last

def train\_model(model, train\_loader, optimizer, criterion):

    model.train()

    total\_loss = 0

    total\_correct = 0

    total\_count = 0

    for texts, labels in train\_loader:

        texts, labels = texts.to(device), labels.to(device)

        optimizer.zero\_grad()

        logits, \_ = model(texts)

        loss = criterion(logits, labels)

        loss.backward()

        optimizer.step()

        total\_loss += loss.item() \* texts.size(0)

        preds = (torch.sigmoid(logits) > 0.5).long()

        correct = (preds == labels.long()).sum().item()

        total\_correct += correct

        total\_count += texts.size(0)

    average\_loss = total\_loss / total\_count

    accuracy = total\_correct / total\_count

    return average\_loss, accuracy

def evaluate\_model(model, test\_loader, criterion):

    model.eval()

    total\_loss = 0

    total\_correct = 0

    total\_count = 0

    with torch.no\_grad():

        for texts, labels in test\_loader:

            texts, labels = texts.to(device), labels.to(device)

            logits, \_ = model(texts)

            loss = criterion(logits, labels)

            total\_loss += loss.item() \* texts.size(0)

            preds = (torch.sigmoid(logits) > 0.5).long()

            correct = (preds == labels.long()).sum().item()

            total\_correct += correct

            total\_count += texts.size(0)

    average\_loss = total\_loss / total\_count

    accuracy = total\_correct / total\_count

    return average\_loss, accuracy

vocab\_size = len(vocab)

model = RNNModel(vocab\_size, embedding\_size, h\_size, number\_of\_layers=number\_of\_layers).to(device)

criterion = nn.BCEWithLogitsLoss()

optimizer = torch.optim.Adam(model.parameters(), lr=1e-7)

RNN\_losses, RNN\_accuracy = [], []

test\_loss, test\_accuracy = evaluate\_model(model, test\_loader, criterion)

RNN\_losses.append(test\_loss)

RNN\_accuracy.append(test\_accuracy)

print(f"First Test Loss: {test\_loss:.4f}; First Test Accuracy: {test\_accuracy:.4f}")

for epoch in range(1, epochs + 1):

    start\_time = time.time()

    train\_loss, train\_accuracy = train\_model(model, train\_loader, optimizer, criterion)

    test\_loss, test\_accuracy = evaluate\_model(model, test\_loader, criterion)

    RNN\_losses.append(test\_loss)

    RNN\_accuracy.append(test\_accuracy)

    elapsed\_time = time.time() - start\_time

    print(f"Epoch: {epoch} \n"

          f"Elapsed Time: {elapsed\_time:.2f}s \n"

          f"Train Loss: {train\_loss:.4f}; Train Accuracy: {train\_accuracy:.4f} \n"

          f"Test Loss: {test\_loss:.4f}; Test Accuracy: {test\_accuracy:.4f} \n")

vocab\_size = len(vocab)

LSTMmodel = LSTMModel(vocab\_size, embedding\_size, h\_size, number\_of\_layers=number\_of\_layers).to(device)

criterion = nn.BCEWithLogitsLoss()

optimizer = torch.optim.Adam(LSTMmodel.parameters(), lr=1e-10)

LSTM\_losses, LSTM\_accuracy = [], []

test\_loss, test\_accuracy = evaluate\_model(LSTMmodel, test\_loader, criterion)

LSTM\_losses.append(test\_loss)

LSTM\_accuracy.append(test\_accuracy)

print(f"First Test Loss: {test\_loss:.4f}; First Test Accuracy: {test\_accuracy:.4f}")

for epoch in range(1, epochs + 1):

    start\_time = time.time()

    train\_loss, train\_accuracy = train\_model(LSTMmodel, train\_loader, optimizer, criterion)

    test\_loss, test\_accuracy = evaluate\_model(LSTMmodel, test\_loader, criterion)

    LSTM\_losses.append(test\_loss)

    LSTM\_accuracy.append(test\_accuracy)

    elapsed\_time = time.time() - start\_time

    print(f"Epoch: {epoch} \n"

          f"Elapsed Time: {elapsed\_time:.2f}s \n"

          f"Train Loss: {train\_loss:.4f}; Train Accuracy: {train\_accuracy:.4f} \n"

          f"Test Loss: {test\_loss:.4f}; Test Accuracy: {test\_accuracy:.4f} \n")

vocab\_size = len(vocab)

GRUmodel = GRUModel(vocab\_size, embedding\_size, h\_size, number\_of\_layers=number\_of\_layers).to(device)

criterion = nn.BCEWithLogitsLoss()

optimizer = torch.optim.Adam(GRUmodel.parameters(), lr=1e-9)

GRU\_losses, GRU\_accuracy = [], []

test\_loss, test\_accuracy = evaluate\_model(GRUmodel, test\_loader, criterion)

GRU\_losses.append(test\_loss)

GRU\_accuracy.append(test\_accuracy)

print(f"First Test Loss: {test\_loss:.4f}; First Test Accuracy: {test\_accuracy:.4f}")

for epoch in range(1, epochs + 1):

    start\_time = time.time()

    train\_loss, train\_accuracy = train\_model(GRUmodel, train\_loader, optimizer, criterion)

    test\_loss, test\_accuracy = evaluate\_model(GRUmodel, test\_loader, criterion)

    GRU\_losses.append(test\_loss)

    GRU\_accuracy.append(test\_accuracy)

    elapsed\_time = time.time() - start\_time

    print(f"Epoch: {epoch} \n"

          f"Elapsed Time: {elapsed\_time:.2f}s \n"

          f"Train Loss: {train\_loss:.4f}; Train Accuracy: {train\_accuracy:.4f} \n"

          f"Test Loss: {test\_loss:.4f}; Test Accuracy: {test\_accuracy:.4f} \n")

horizontal\_axis = range(epochs + 1)

plt.plot(horizontal\_axis, RNN\_losses, label="RNN")

plt.plot(horizontal\_axis, LSTM\_losses, label="LSTM")

plt.plot(horizontal\_axis, GRU\_losses, label="GRU")

plt.title("Графики зависимости значений функции потерь от выбранной архитектуры")

plt.xlabel("Epoch")

plt.ylabel("Loss")

plt.grid(True)

plt.legend()

plt.show()

plt.plot(horizontal\_axis, RNN\_accuracy, label="RNN")

plt.plot(horizontal\_axis, LSTM\_accuracy, label="LSTM")

plt.plot(horizontal\_axis, GRU\_accuracy, label="GRU")

plt.title("Графики зависимости точности от выбранной архитектуры")

plt.xlabel("Epoch")

plt.ylabel("Accuracy")

plt.grid(True)

plt.legend()

plt.show()

# Результаты

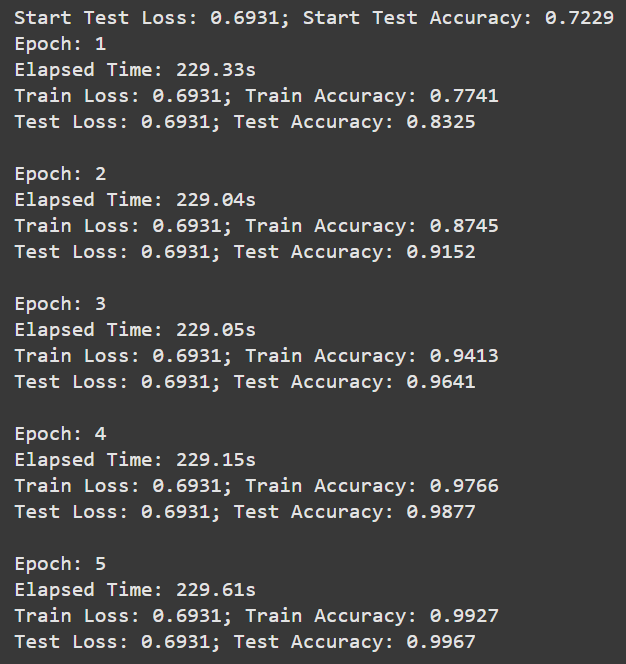


Рисунок – RNN



Рисунок – LSTM

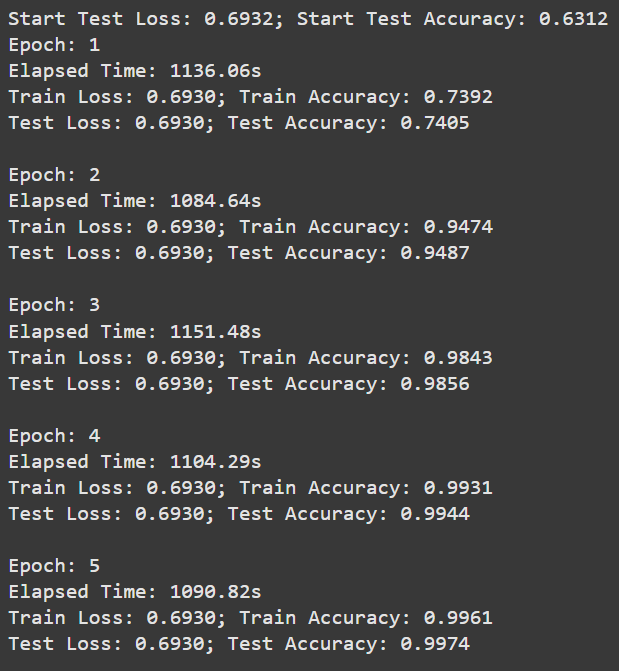


Рисунок – GRU

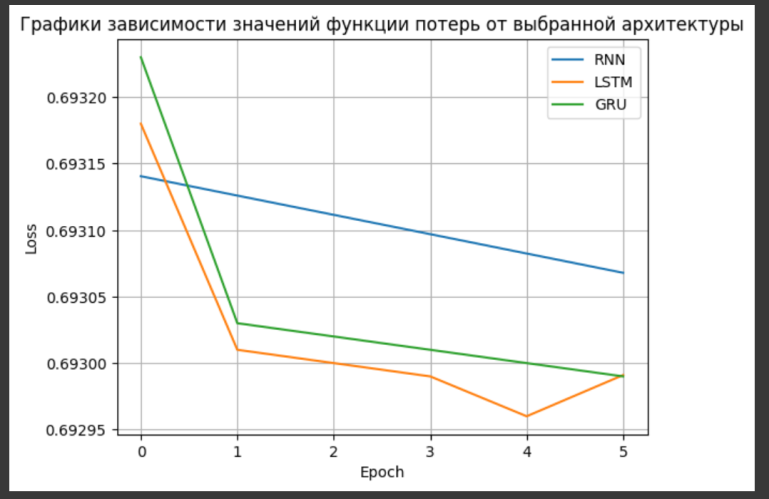


Рисунок - Графики зависимости значений функции потерь от выбранной архитектуры

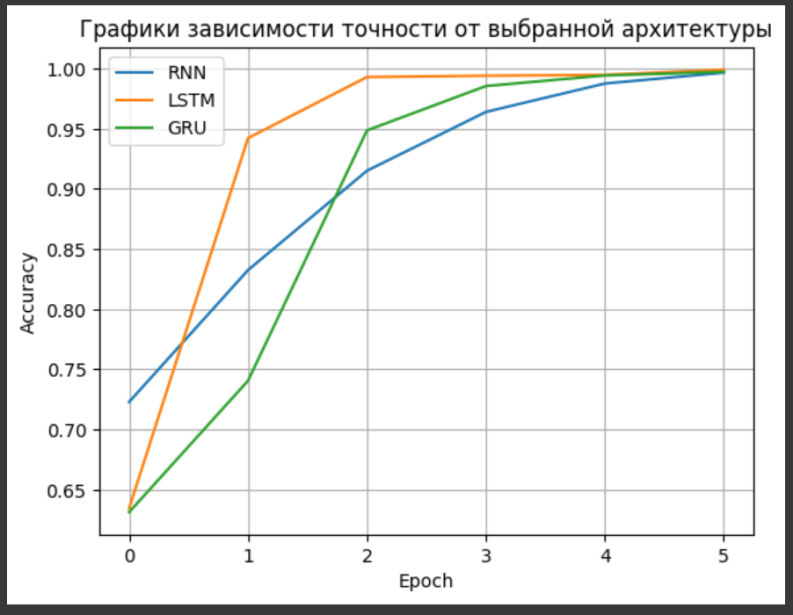


Рисунок - Графики зависимости точности от выбранной архитектуры

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Архитектура | Кол-во эпох | Скорость обучения | Верность |
| 1 | RNN | 5 | 1e-7 | 0.9967 |
| 2 | LSTM | 2 | 1e-10 | 0.9992 |
| 3 | GRU | 3 | 1e-9 | 0.9974 |

# Вывод

В ходе выполнения данной работы было установлено, что наилучшие результаты для поставленной задачи продемонстрировала архитектура LSTM, тогда как простейшая рекуррентная нейронная сеть (RNN) оказалась наименее эффективной по сравнению с другими архитектурами.