|  |
| --- |
| МИНОБРНАУКИ РОССИИ |
| Федеральное государственное бюджетное образовательное учреждение высшего образования **«МИРЭА − Российский технологический университет»**  **РТУ МИРЭА** |

**Институт информационных технологий (ИИТ)**

**Кафедра прикладной математики (ПМ)**

**ОТЧЕТ ПО ПРАКТИЧЕСКОЙ РАБОТЕ**

по дисциплине «Технологии и инструментарий машинного обучения»

**Практическое занятие № 5**

|  |  |  |  |
| --- | --- | --- | --- |
| Студент группы ИНБО-01-17 | ИНБО-06-22, Морозов Артемий Русланович | (подпись) | |
| Преподаватель | Трушин Степан Михайлович, старший преподаватель | (подпись) | |
| Отчет представлен | «10» мая 2025г. | |  | |

Москва 2025 г.

содержание

[практическая работа 5 3](#_Toc197794482)

[Тема 3](#_Toc197794483)

[Цель 3](#_Toc197794484)

[Задание 3](#_Toc197794485)

[Ход работы 5](#_Toc197794486)

[Вывод 10](#_Toc197794487)

[Листинг кода 10](#_Toc197794488)

[Ссылка на Google colab 18](#_Toc197794489)

практическая работа 5

Тема

Автокодировщики и рекуррентные нейронные сети

Цель

Освоить принципы работы автокодировщиков для сжатия и восстановления данных, а также изучить применение рекуррентных нейронных сетей (RNN, LSTM, GRU) для обработки временных последовательностей и текстов.

Задание

**Часть 1: Автокодировщик (1 пара)**

1. Подготовка данных:

* Загрузите датасет изображений (например, MNIST или Fashion MNIST).
* Выполните нормализацию данных.

1. Создание автокодировщика:

* Постройте сеть с симметричной архитектурой:
* Входной слой, сжимающий изображение до вектора.
* Скрытые слои, выполняющие кодирование данных.
* Декодирующие слои, восстанавливающие изображение из вектора.

1. Обучение автокодировщика:

* Обучите модель на обучающей выборке.
* Оцените качество восстановления изображений на тестовой выборке.

1. Визуализация:

* Постройте графики исходных и восстановленных изображений.
* Визуализируйте скрытые представления данных

1. Эксперимент:

* Изучите влияние уменьшения размера скрытого представления (вектора) на качество восстановления.

**Часть 2: Рекуррентные нейронные сети (1 пара)**

1. Подготовка данных:

* Используйте набор последовательных данных, например, текстовый датасет (предсказание следующего слова) или временной ряд (предсказание следующего значения).:

1. Реализация рекуррентной сети:

* Постройте базовую рекуррентную сеть с одним рекуррентным слоем (RNN).
* Обучите её на предоставленных данных.

1. Расширение модели:

* Замените базовую RNN на LSTM и GRU.
* Сравните их поведение и точность на тестовой выборке.

1. Визуализация:

* Графики ошибки и точности для каждой архитектуры (RNN, LSTM, GRU).
* Визуализация предсказаний на временных рядах или текстах.

Ход работы

Набор данных MNIST (Modified National Institute of Standards and Technology) содержит 70 000 черно-белых изображений рукописных цифр от 0 до 9 размером 28×28 пикселей. Обучающая выборка включает 60 000 изображений, а тестовая — 10 000. Каждая цифра представлена примерно одинаковым количеством примеров.

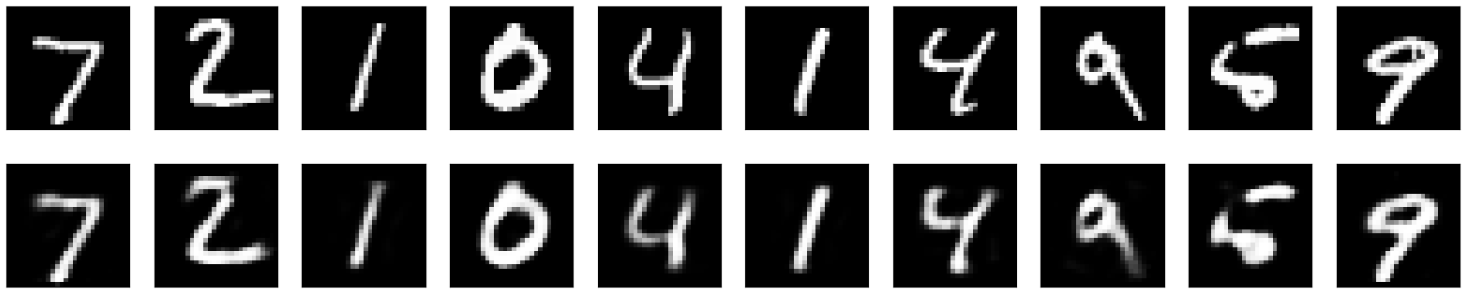
Автокодировщик — это нейронная сеть, которая учится сжимать входные данные в компактное представление (кодирование), а затем восстанавливать их обратно (декодирование) с минимальными потерями. Основные применения автокодировщиков включают уменьшение размерности данных, шумоподавление, генерацию новых данных и выделение ключевых признаков для задач машинного обучения, таких как кластеризация или визуализация сложных данных.

Рекуррентные нейронные сети (RNN) — это класс нейронных сетей, предназначенных для обработки последовательных данных, где важна зависимость между элементами последовательности, например, временные ряды, тексты или речь. В отличие от обычных нейронных сетей, RNN имеют внутреннюю память, которая позволяет сохранять информацию о предыдущих состояниях и использовать её для обработки последующих элементов. Однако классические RNN страдают от проблем исчезающих или взрывающихся градиентов, что затрудняет их обучение на длинных последовательностях.

Для решения этих проблем были разработаны более сложные архитектуры, такие как LSTM (Long Short-Term Memory) и GRU (Gated Recurrent Unit). LSTM использует механизм ворот (input, forget и output gates), чтобы контролировать поток информации и эффективно запоминать долгосрочные зависимости, в то время как GRU упрощает эту структуру, объединяя некоторые ворота, что делает её менее вычислительно затратной, но при этом сохраняющей способность к обучению на длинных последовательностях. Обе архитектуры широко применяются в задачах машинного перевода, генерации текста, прогнозирования временных рядов и других областях, где важна контекстная информация.

В рассмотренном далее примере с автокодировщиком для MNIST архитектура сети состояла из симметричных кодирующей и декодирующей частей. На вход подавались растянутые в вектор изображения размером 784 (28×28 пикселей). Кодирующая часть содержала один скрытый слой с 32 нейронами и активацией ReLU, который сжимал данные до скрытого представления размером 32. Декодер симметрично восстанавливал исходную размерность через слой из 784 нейронов с сигмоидной активацией, обеспечивающей выходные значения в диапазоне [0, 1].

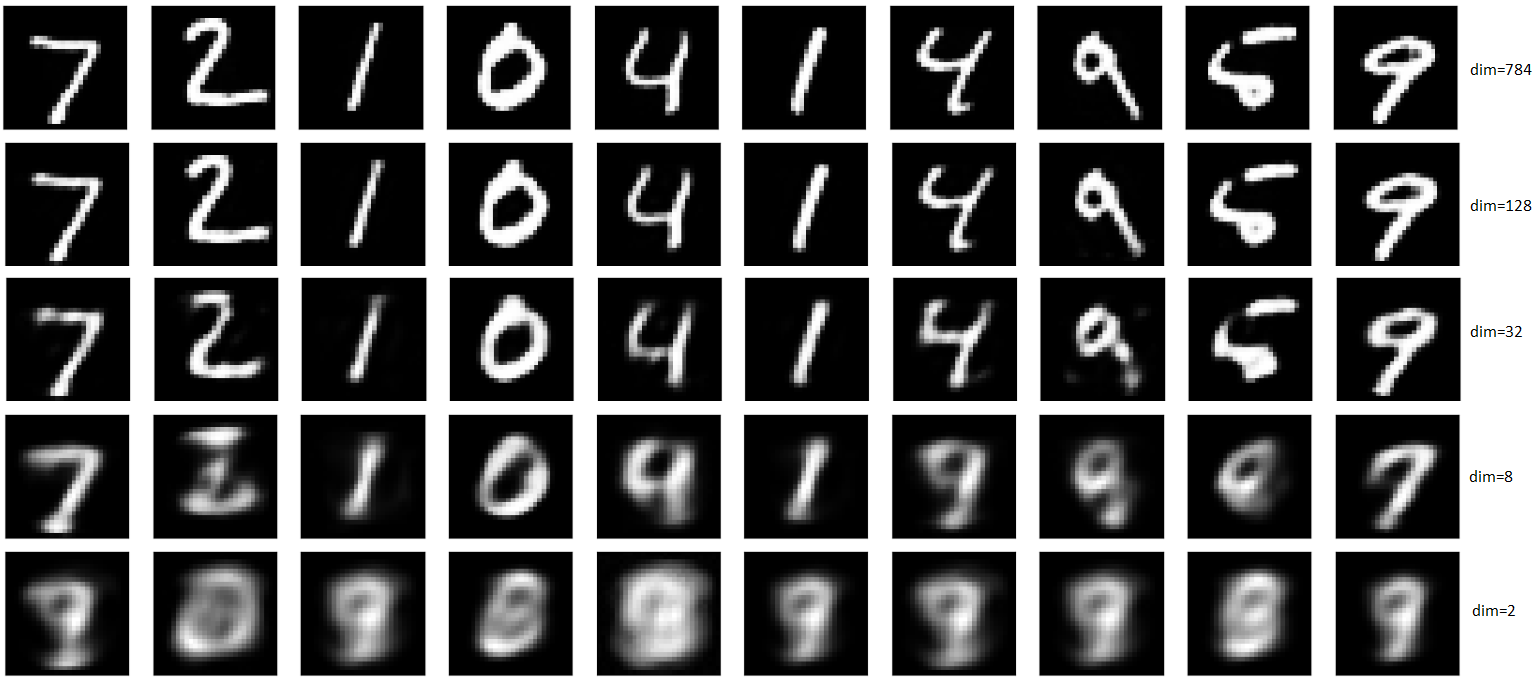
Проанализируем исходные и полученные через декодер данные с размером 32.



**Рисунок 1 — Цифры автокодировщика с размером 32**

На Рисунке 1 сверху изображены исходные данные, а снизу результат работы модели. Автокодировщик со скрытым слоем размером 32 успешно справился с восстановлением исходных цифр MNIST, небольшое размытие практически незаметно.

Далее выполним сравнение моделей с различными размерами скрытого представления. Результаты сравнения работы автокодировщика с разными размерами скрытого слоя (от 784 до 2 нейронов) показывают, как степень сжатия влияет на качество восстановления цифр MNIST.



**Рисунок 2 — Сравнение автокодировщика с разными размерами представлений**

На Рисунке 2 при максимальном размере скрытого слоя (dim-784, эквивалентном исходным данным) восстановление происходит идеально, так как сжатие фактически отсутствует. С уменьшением размера скрытого слоя до 128 и 32 нейронов качество остаётся высоким — все цифры восстанавливаются корректно, что подтверждает эффективность автокодировщика для MNIST при умеренном сжатии. Однако при сильном сжатии (dim-8 и особенно dim-2) начинают проявляться ошибки: в строке dim-8 цифра "5" превращается в "9", а в строке dim-2 ошибки становятся невозможным определить цифру, что свидетельствует о потере ключевых признаков из-за чрезмерного уменьшения размерности.

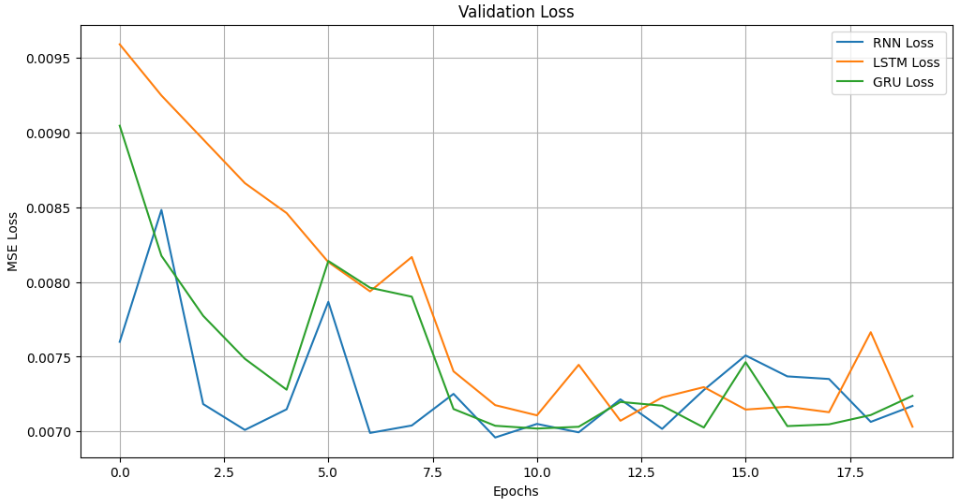
Таким образом, оптимальный баланс между степенью сжатия и качеством восстановления для MNIST достигается при размере скрытого слоя около 32 нейронов, тогда как экстремальное сжатие (2 нейрона) приводит к существенному искажению данных.

Рекуррентные нейронные сети (RNN) представляют собой класс нейросетевых архитектур, специально разработанных для обработки последовательных данных, где каждый элемент зависит от предыдущих. Основная особенность RNN — наличие скрытого состояния, которое передает информацию между шагами последовательности, позволяя сети учитывать контекст. Однако классические RNN страдают от проблемы исчезающих градиентов, что затрудняет обучение на длинных последовательностях.

Для решения этой проблемы были разработаны LSTM (Long Short-Term Memory) сети, которые вводят механизм ворот (input, forget и output gates), позволяющий явно контролировать поток информации. Входные ворота решают, какую новую информацию запомнить, forget gate — что забыть из предыдущего состояния, а output gate — какое количество информации передать на следующий шаг. Эта архитектура эффективно сохраняет долгосрочные зависимости, но требует больше вычислительных ресурсов.

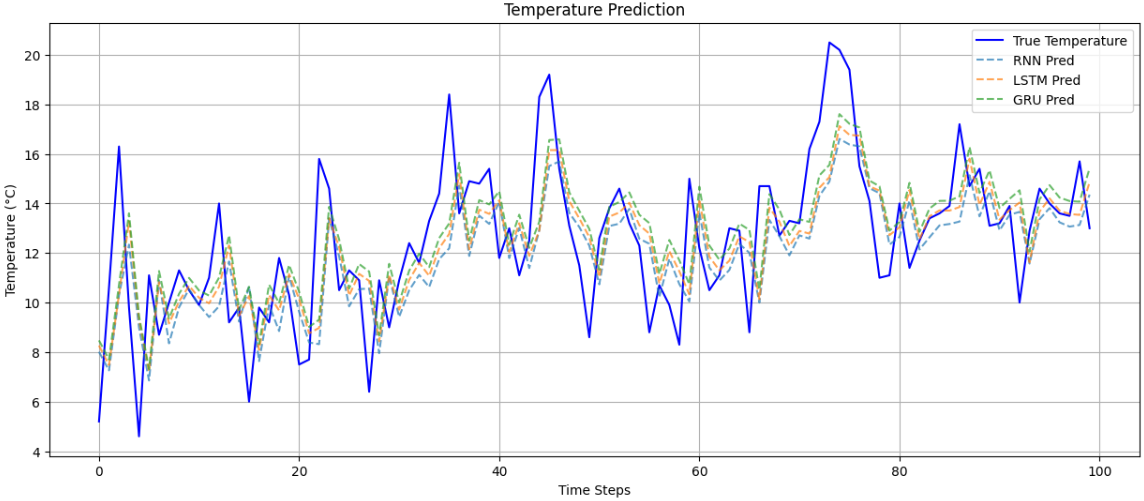
GRU (Gated Recurrent Unit) — это упрощенная версия LSTM, объединяющая forget и input gates в единый механизм "update gate" и исключающая output gate в пользу полного состояния. GRU сохраняет способность к обучению на длинных последовательностях, но работает быстрее благодаря меньшему количеству параметров.

Выполним обучение моделей и сравним их ошибки на тестовых данных.



**Рисунок 3 — График потерь рекуррентных моделей**

На Рисунке 3 все модели достигают сходимости, но с разной динамикой: классическая RNN показывает более резкие скачки, что объясняется её склонностью к проблеме исчезающих градиентов. LSTM и GRU демонстрируют близкие результаты, но GRU стабильнее LSTM после 5-й эпохи, благодаря более эффективному механизму обновления состояния при меньших вычислительных затратах.



**Рисунок 4 — Визуализация предсказаний в сравнении с правильными значениями**

На Рисунке 4 у всех трех моделей видны несущественные отклонения от исходных значений, при этом между собой предсказания моделей практически не отличаются, что показывает стабильный результат их работы.

Таким образом, рекуррентные сети (RNN, LSTM, GRU) оптимальны для последовательностей: RNN подходит для коротких зависимостей, LSTM — для сложных долгосрочных (тексты, временные ряды), а GRU предлагает баланс точности и скорости. Для задач с высокой частотой обновления (речь, транзакции) предпочтительна GRU, тогда как LSTM лучше справляется с контекстно-зависимыми данными (машинный перевод). Выбор архитектуры определяется компромиссом между точностью, скоростью обучения и ресурсами.

Автокодировщики эффективны для задач сжатия данных, шумоподавления и выделения признаков, демонстрируя лучшие результаты при умеренной размерности скрытого слоя (например, 32 нейрона для MNIST). Для простых данных достаточно линейной архитектуры, тогда как для сложных (например, изображений CIFAR-10) требуются глубокие или вариационные автокодировщики.

Вывод

Я освоил принципы работы автокодировщиков для сжатия и восстановления данных, а также изучил применение рекуррентных нейронных сетей (RNN, LSTM, GRU) для обработки временных последовательностей и текстов.

Листинг кода

*Листинг 1 – Загрузка датасета*

import numpy as np

import matplotlib.pyplot as plt

from tensorflow.keras.datasets import mnist

from tensorflow.keras.models import Model

from tensorflow.keras.layers import Input, Dense

from tensorflow.keras.optimizers import Adam

#Загрузка данных MNIST

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

#Нормализация данных

x\_train = x\_train.astype('float32') / 255.

x\_test = x\_test.astype('float32') / 255.

#Преобразование в одномерные векторы

x\_train = x\_train.reshape((len(x\_train), 784))

x\_test = x\_test.reshape((len(x\_test), 784))

print(f"Train data shape: {x\_train.shape}")

print(f"Test data shape: {x\_test.shape}")

*Листинг 2 – Создание и обучение авктокодировщика*

encoding\_dim = 32

#Входной слой

input\_img = Input(shape=(784,))

#Кодирующая часть

encoded = Dense(encoding\_dim, activation='relu')(input\_img)

#Декодирующая часть

decoded = Dense(784, activation='sigmoid')(encoded)

autoencoder = Model(input\_img, decoded)

autoencoder.compile(optimizer=Adam(learning\_rate=0.001),

loss='binary\_crossentropy')

autoencoder.summary()

#Обучение модели

history = autoencoder.fit(x\_train, x\_train,

epochs=20,

batch\_size=64,

shuffle=True,

validation\_data=(x\_test, x\_test))

decoded\_imgs = autoencoder.predict(x\_test)

scaler = StandardScaler()

X\_scaled = scaler.fit\_transform(X)

*Листинг 3 – Создание модели с Bath нормализацией*

def create\_bn\_model():

model = models.Sequential([

layers.Flatten(input\_shape=(32, 32, 3)),

layers.Dense(512),

layers.BatchNormalization(),

layers.Activation('relu'),

layers.Dense(256),

layers.BatchNormalization(),

layers.Activation('relu'),

layers.Dense(128),

layers.BatchNormalization(),

layers.Activation('relu'),

layers.Dense(10, activation='softmax')

])

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

return model

#Построение модели

bn\_model = create\_bn\_model()

bn\_history = bn\_model.fit(train\_images, train\_labels,

epochs=10, batch\_size=256,

validation\_data=(test\_images, test\_labels),

verbose=1)

model.compile(optimizer='adam', # Оптимизатор Adam

loss='categorical\_crossentropy', # Функция потерь: кросс-энтропия

metrics=['accuracy'])

history = model.fit(x\_train, y\_train,

epochs=10, # Количество эпох

batch\_size=128, # Размер батча

validation\_data=(x\_test, y\_test)) # Тестовые данные для валидации

test\_loss, test\_acc = model.evaluate(x\_test, y\_test)

print(f'Точность на тестовых данных: {test\_acc:.4f}')

*Листинг 4 – Создание модели с Layer нормализацией*

def create\_ln\_model():

model = models.Sequential([

layers.Flatten(input\_shape=(32, 32, 3)),

layers.Dense(512),

layers.LayerNormalization(),

layers.Activation('relu'),

layers.Dense(256),

layers.LayerNormalization(),

layers.Activation('relu'),

layers.Dense(128),

layers.LayerNormalization(),

layers.Activation('relu'),

layers.Dense(10, activation='softmax')

])

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

return model

#Построение модели

ln\_model = create\_ln\_model()

ln\_history = ln\_model.fit(train\_images, train\_labels,

epochs=10, batch\_size=256,

validation\_data=(test\_images, test\_labels),

verbose=1)

#Обучение и оценка моделей

histories = []

for params in params\_combinations:

print(f"\nTraining model with: {params}")

model = create\_model(\*\*params)

history = model.fit(x\_train, y\_train,

epochs=6,

batch\_size=64,

validation\_data=(x\_test, y\_test),

verbose=1)

histories.append((params, history))print(f"Среднеквадратичная ошибка: {mean\_squared\_error(y\_test, y\_pred):.4f}")

coef = pd.Series(lasso.coef\_, index=X.columns)

print("Коэффициенты модели с Lasso:")

print(coef)

coef.plot(kind='bar', figsize=(10, 6))

plt.title("Коэффициенты модели с Lasso-регуляризацией")

plt.show()

*Листинг 5 – Создание модели с Dropout*

def create\_dropout\_model():

model = models.Sequential([

layers.Flatten(input\_shape=(32, 32, 3)),

layers.Dense(512, activation='relu'),

layers.Dropout(0.5),

layers.Dense(256, activation='relu'),

layers.Dropout(0.5),

layers.Dense(128, activation='relu'),

layers.Dropout(0.5),

layers.Dense(10, activation='softmax')

])

model.compile(optimizer='adam',

loss='categorical\_crossentropy',

metrics=['accuracy'])

return model

#Построение модели

dropout\_model = create\_dropout\_model()

dropout\_history = dropout\_model.fit(train\_images, train\_labels,

epochs=10, batch\_size=256,

validation\_data=(test\_images, test\_labels),

verbose=1)

*Листинг 6 – Визуализация результатов*

def plot\_history(histories, labels):

plt.figure(figsize=(12, 5))

#График точности

plt.subplot(1, 2, 1)

for history, label in zip(histories, labels):

plt.plot(history.history['accuracy'], label=f'{label} train')

plt.plot(history.history['val\_accuracy'], '--', label=f'{label} test')

plt.title('Model accuracy')

plt.ylabel('Accuracy')

plt.xlabel('Epoch')

plt.legend()

#График потерь

plt.subplot(1, 2, 2)

for history, label in zip(histories, labels):

plt.plot(history.history['loss'], label=f'{label} train')

plt.plot(history.history['val\_loss'], '--', label=f'{label} test')

plt.title('Model loss')

plt.ylabel('Loss')

plt.xlabel('Epoch')

plt.legend()

plt.tight\_layout()

plt.show()

#Сравнение всех моделей

histories = [baseline\_history, bn\_history, ln\_history, dropout\_history]

labels = ['Baseline', 'BatchNorm', 'LayerNorm', 'Dropout']

plot\_history(histories, labels)

#Анализ переобучения

final\_train\_acc = [h.history['accuracy'][-1] for h in histories]

final\_val\_acc = [h.history['val\_accuracy'][-1] for h in histories]

overfitting\_gap = [train - val for train, val in zip(final\_train\_acc, final\_val\_acc)]

print("\nПереобучение (разница между train и test accuracy):")

for label, gap in zip(labels, overfitting\_gap):

print(f"{label}: {gap:.4f}")

#Анализ скорости сходимости (сколько эпох до стабилизации)

def epochs\_to\_converge(history, metric='val\_accuracy', window=5, threshold=0.001):

values = history.history[metric]

for i in range(len(values)-window):

if max(values[i:i+window]) - min(values[i:i+window]) < threshold:

return i

return len(values)

print("\nСкорость сходимости (эпохи до стабилизации val\_accuracy):")

for label, history in zip(labels, histories):

print(f"{label}: {epochs\_to\_converge(history)} эпох") # Визуализация для лучшей модели (64 фильтра, 3x3, ReLU)

best\_model = create\_model(filters=64, kernel\_size=(3,3), activation='relu')

best\_model.fit(x\_train, y\_train, epochs=20, verbose=0)

plt.figure(figsize=(15, 10))

for i, (img, label) in enumerate(zip(sample\_images, sample\_labels)):

# Подготовка изображения

img\_array = np.expand\_dims(img, axis=0)

# Получаем heatmap

heatmap = grad\_cam(best\_model, img\_array)

# Масштабируем heatmap к размеру изображения

heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))

heatmap = np.uint8(255 \* heatmap)

heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP\_JET)

# Наложение heatmap на оригинальное изображение

superimposed\_img = heatmap \* 0.4 + img \* 255 \* 0.6

superimposed\_img = np.clip(superimposed\_img, 0, 255).astype('uint8')

# Визуализация

plt.subplot(5, 2, 2\*i+1)

plt.imshow(img)

plt.title(f"True: {label}")

plt.axis('off')

plt.subplot(5, 2, 2\*i+2)

plt.imshow(superimposed\_img)

plt.title("Grad-CAM")

plt.axis('off')

plt.tight\_layout()

plt.show()

#SHAP анализ

explainer = shap.TreeExplainer(xgb\_model)

shap\_values = explainer.shap\_values(X\_test)

shap.summary\_plot(shap\_values, X\_test, plot\_type="bar")

shap.dependence\_plot("Feature 0", shap\_values, X\_test)

*Листинг 7 – Эксперименты с ResNet, VGG, MobileNet*

import torchvision.models as models

import torch.nn as nn

from torchvision import models

import torch.nn as nn

def get\_models(num\_classes=10):

#ResNet18

resnet = models.resnet18(pretrained=False)

resnet.fc = nn.Linear(resnet.fc.in\_features, num\_classes)

#VGG11

vgg = models.vgg11(pretrained=False)

vgg.classifier[6] = nn.Linear(vgg.classifier[6].in\_features, num\_classes)

#MobileNet

mobile = models.mobilenet\_v3\_small(pretrained=False)

mobile.classifier[3] = nn.Linear(mobile.classifier[3].in\_features, num\_classes)

return {'ResNet18': resnet, 'VGG11': vgg, 'MobileNetV3-Small': mobile}

models\_dict = get\_models()

import torch.optim as optim

from torch.utils.tensorboard import SummaryWriter

import time

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

#Гиперпараметры

lr = 0.001

num\_epochs = 20

#Функция тренировки одной модели

def train\_and\_eval(model, name):

model.to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=lr)

writer = SummaryWriter(log\_dir=f'runs/{name}')

history = {'train\_loss': [], 'train\_acc': [], 'test\_acc': []}

start\_time = time.time()

for epoch in range(1, num\_epochs+1):

#Тренировочная фаза

model.train()

correct = total = running\_loss = 0

for inputs, labels in train\_loader:

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

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()\*inputs.size(0)

\_, preds = torch.max(outputs, 1)

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

total += labels.size(0)

train\_loss = running\_loss/total

train\_acc = correct/total

history['train\_loss'].append(train\_loss)

history['train\_acc'].append(train\_acc)

#Тестовая фаза

model.eval()

correct = total = 0

with torch.no\_grad():

for inputs, labels in test\_loader:

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

outputs = model(inputs)

\_, preds = torch.max(outputs, 1)

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

total += labels.size(0)

test\_acc = correct/total

history['test\_acc'].append(test\_acc)

writer.add\_scalar('Loss/train', train\_loss, epoch)

writer.add\_scalar('Accuracy/train', train\_acc, epoch)

writer.add\_scalar('Accuracy/test', test\_acc, epoch)

print(f"{name} Epoch {epoch}/{num\_epochs} — loss: {train\_loss:.4f}, train\_acc: {train\_acc:.4f}, test\_acc: {test\_acc:.4f}")

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

writer.close()

return history, total\_time

#Запуск обучения для всех моделей

histories = {}

times = {}

for name, model in models\_dict.items():

hist, t = train\_and\_eval(model, name)

histories[name] = hist

times[name] = t

#Сравнение всех моделей

histories = [baseline\_history, bn\_history, ln\_history, dropout\_history]

labels = ['Baseline', 'BatchNorm', 'LayerNorm', 'Dropout']

plot\_history(histories, labels)

#Анализ переобучения

final\_train\_acc = [h.history['accuracy'][-1] for h in histories]

final\_val\_acc = [h.history['val\_accuracy'][-1] for h in histories]

overfitting\_gap = [train - val for train, val in zip(final\_train\_acc, final\_val\_acc)]

print("\nПереобучение (разница между train и test accuracy):")

for label, gap in zip(labels, overfitting\_gap):

print(f"{label}: {gap:.4f}")

#Анализ скорости сходимости (сколько эпох до стабилизации)

def epochs\_to\_converge(history, metric='val\_accuracy', window=5, threshold=0.001):

values = history.history[metric]

for i in range(len(values)-window):

if max(values[i:i+window]) - min(values[i:i+window]) < threshold:

return i

return len(values)

print("\nСкорость сходимости (эпохи до стабилизации val\_accuracy):")

for label, history in zip(labels, histories):

print(f"{label}: {epochs\_to\_converge(history)} эпох") # Визуализация для лучшей модели (64 фильтра, 3x3, ReLU)

best\_model = create\_model(filters=64, kernel\_size=(3,3), activation='relu')

best\_model.fit(x\_train, y\_train, epochs=20, verbose=0)

plt.figure(figsize=(15, 10))

for i, (img, label) in enumerate(zip(sample\_images, sample\_labels)):

# Подготовка изображения

img\_array = np.expand\_dims(img, axis=0)

# Получаем heatmap

heatmap = grad\_cam(best\_model, img\_array)

# Масштабируем heatmap к размеру изображения

heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))

heatmap = np.uint8(255 \* heatmap)

heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP\_JET)

# Наложение heatmap на оригинальное изображение

superimposed\_img = heatmap \* 0.4 + img \* 255 \* 0.6

superimposed\_img = np.clip(superimposed\_img, 0, 255).astype('uint8')

# Визуализация

plt.subplot(5, 2, 2\*i+1)

plt.imshow(img)

plt.title(f"True: {label}")

plt.axis('off')

plt.subplot(5, 2, 2\*i+2)

plt.imshow(superimposed\_img)

plt.title("Grad-CAM")

plt.axis('off')

plt.tight\_layout()

plt.show()

#SHAP анализ

explainer = shap.TreeExplainer(xgb\_model)

shap\_values = explainer.shap\_values(X\_test)

shap.summary\_plot(shap\_values, X\_test, plot\_type="bar")

shap.dependence\_plot("Feature 0", shap\_values, X\_test)

*Листинг 8 – Визуализация результатов*

# Графики ошибок

plt.figure(figsize=(12, 6))

plt.plot(history\_rnn.history['val\_loss'], label='RNN Loss')

plt.plot(history\_lstm.history['val\_loss'], label='LSTM Loss')

plt.plot(history\_gru.history['val\_loss'], label='GRU Loss')

plt.title('Validation Loss')

plt.xlabel('Epochs')

plt.ylabel('MSE Loss')

plt.legend()

plt.grid()

plt.show()

# Визуализация предсказаний (последние 100 точек)

plt.figure(figsize=(15, 6))

plt.plot(y\_test\_orig[-100:], label='True Temperature', color='blue')

plt.plot(y\_pred\_rnn\_orig[-100:], label='RNN Pred', linestyle='--', alpha=0.7)

plt.plot(y\_pred\_lstm\_orig[-100:], label='LSTM Pred', linestyle='--', alpha=0.7)

plt.plot(y\_pred\_gru\_orig[-100:], label='GRU Pred', linestyle='--', alpha=0.7)

plt.title('Temperature Prediction')

plt.xlabel('Time Steps')

plt.ylabel('Temperature (°C)')

plt.legend()

plt.grid()

plt.show()

def train\_and\_eval(model, name):

model.to(device)

criterion = nn.CrossEntropyLoss()

optimizer = optim.Adam(model.parameters(), lr=lr)

writer = SummaryWriter(log\_dir=f'runs/{name}')

history = {'train\_loss': [], 'train\_acc': [], 'test\_acc': []}

start\_time = time.time()

for epoch in range(1, num\_epochs+1):

#Тренировочная фаза

model.train()

correct = total = running\_loss = 0

for inputs, labels in train\_loader:

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

optimizer.zero\_grad()

outputs = model(inputs)

loss = criterion(outputs, labels)

loss.backward()

optimizer.step()

running\_loss += loss.item()\*inputs.size(0)

\_, preds = torch.max(outputs, 1)

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

total += labels.size(0)

train\_loss = running\_loss/total

train\_acc = correct/total

history['train\_loss'].append(train\_loss)

history['train\_acc'].append(train\_acc)

#Тестовая фаза

model.eval()

correct = total = 0

with torch.no\_grad():

for inputs, labels in test\_loader:

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

outputs = model(inputs)

\_, preds = torch.max(outputs, 1)

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

total += labels.size(0)

test\_acc = correct/total

history['test\_acc'].append(test\_acc)

writer.add\_scalar('Loss/train', train\_loss, epoch)

writer.add\_scalar('Accuracy/train', train\_acc, epoch)

writer.add\_scalar('Accuracy/test', test\_acc, epoch)

print(f"{name} Epoch {epoch}/{num\_epochs} — loss: {train\_loss:.4f}, train\_acc: {train\_acc:.4f}, test\_acc: {test\_acc:.4f}")

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

writer.close()

return history, total\_time

#Запуск обучения для всех моделей

histories = {}

times = {}

for name, model in models\_dict.items():

hist, t = train\_and\_eval(model, name)

histories[name] = hist

times[name] = t

#Сравнение всех моделей

histories = [baseline\_history, bn\_history, ln\_history, dropout\_history]

labels = ['Baseline', 'BatchNorm', 'LayerNorm', 'Dropout']

plot\_history(histories, labels)

#Анализ переобучения

final\_train\_acc = [h.history['accuracy'][-1] for h in histories]

final\_val\_acc = [h.history['val\_accuracy'][-1] for h in histories]

overfitting\_gap = [train - val for train, val in zip(final\_train\_acc, final\_val\_acc)]

print("\nПереобучение (разница между train и test accuracy):")

for label, gap in zip(labels, overfitting\_gap):

print(f"{label}: {gap:.4f}")

#Анализ скорости сходимости (сколько эпох до стабилизации)

def epochs\_to\_converge(history, metric='val\_accuracy', window=5, threshold=0.001):

values = history.history[metric]

for i in range(len(values)-window):

if max(values[i:i+window]) - min(values[i:i+window]) < threshold:

return i

return len(values)

print("\nСкорость сходимости (эпохи до стабилизации val\_accuracy):")

for label, history in zip(labels, histories):

print(f"{label}: {epochs\_to\_converge(history)} эпох") # Визуализация для лучшей модели (64 фильтра, 3x3, ReLU)

best\_model = create\_model(filters=64, kernel\_size=(3,3), activation='relu')

best\_model.fit(x\_train, y\_train, epochs=20, verbose=0)

plt.figure(figsize=(15, 10))

for i, (img, label) in enumerate(zip(sample\_images, sample\_labels)):

# Подготовка изображения

img\_array = np.expand\_dims(img, axis=0)

# Получаем heatmap

heatmap = grad\_cam(best\_model, img\_array)

# Масштабируем heatmap к размеру изображения

heatmap = cv2.resize(heatmap, (img.shape[1], img.shape[0]))

heatmap = np.uint8(255 \* heatmap)

heatmap = cv2.applyColorMap(heatmap, cv2.COLORMAP\_JET)

# Наложение heatmap на оригинальное изображение

superimposed\_img = heatmap \* 0.4 + img \* 255 \* 0.6

superimposed\_img = np.clip(superimposed\_img, 0, 255).astype('uint8')

# Визуализация

plt.subplot(5, 2, 2\*i+1)

plt.imshow(img)

plt.title(f"True: {label}")

plt.axis('off')

plt.subplot(5, 2, 2\*i+2)

plt.imshow(superimposed\_img)

plt.title("Grad-CAM")

plt.axis('off')

plt.tight\_layout()

plt.show()

#SHAP анализ

explainer = shap.TreeExplainer(xgb\_model)

shap\_values = explainer.shap\_values(X\_test)

shap.summary\_plot(shap\_values, X\_test, plot\_type="bar")

shap.dependence\_plot("Feature 0", shap\_values, X\_test)

Ссылка на Google colab

https://colab.research.google.com/drive/1nzF7T0ibMcKb1B9npsS\_-noRjsUmRfjV?usp=sharing