**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ  
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра вычислительной техники**

**ОТЧЁТ  
по дисциплине «Нейронные сети»**

| Студент \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ |  |
| --- | --- |
| Преподаватель \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ | Гатауллин Р.И. |

**Лабораторная работа № 2  
Тема: «Рекуррентные нейронные сети»**

Санкт-Петербург

2024

**Оглавление**

[Цель работы 3](#_1fob9te)

[Задачи работы 3](#_3znysh7)

[Набор данных 4](#_2et92p0)

[Краткая теоретическая информация 4](#_tyjcwt)

[Процесс выполнения работы 6](#)

[Вывод 16](#)

[Код программы 17](#_h6sqr2kyabjp)

# **Цель работы**

Эта работа нацелена на изучение рекуррентных нейронных сетей, приобретение практических навыков в их использовании, понимание архитектуры данных сетей, методов их обучения и применение их в различных задачах.

# **Задачи работы**

В ходе данной работы необходимо выполнить следующие задачи:

* Изучить основные принципы работы рекуррентных нейронных сетей.
* Изучить архитектуры рекуррентных нейронных сетей, такие как RNN, LSTM, GRU и их различия.
* Составить набор данных для обучения и тестирования модели.
* Разработать и обучить модель рекуррентной нейронной сети.
* Проанализировать результаты обучения и определить точность модели.
* Провести эксперименты, варьируя параметры модели, и оценить их влияние на качество классификации.
* Сравнить результаты работы различных архитектур рекуррентных нейронных сетей.

# 

# **Набор данных**

<https://www.kaggle.com/datasets/bulentsiyah/for-simple-exercises-time-series-forecasting/data?select=BeerWineLiquor.csv> – набор данных о продаже пивной продукции. Два столбца содержащие дату в году и кол-во проданного товара. Данный набор данных приспособлен специально для учебных целей. На нем удобно тестировать свои архитектуры нейронных сетей, так как есть несколько зависимостей стоимости продукции от времени (года, месяца, дня), предугадывая цену на продукт в разное время года, разный месяц/день/год.

# 

# **Краткая теоретическая информация**

**Рекуррентные нейронные сети** - это класс многослойных нейронных сетей, которые используются для анализа последовательностей данных. Эти сети могут обрабатывать входные данные, которые поступают в виде последовательности, такие как временные ряды, естественные языки и аудиозаписи. Рекуррентные нейронные сети имеют уникальную способность запоминать информацию из предыдущих шагов и использовать ее для принятия решений на текущем шаге.

**Нормализация данных** – это процесс приведения данных к определенному диапазону значений, чтобы упростить их обработку и улучшить производительность сети. MinMaxScaler нормализатор изменяет масштаб значений числового столбца в наборе данных. Он делает это, преобразуя значения от минимального до максимального значения, так что они находятся в диапазоне от 0 до 1. Это может помочь улучшить производительность модели, поскольку позволяет ей лучше различать разные значения.

**TimeseriesGenerator** - это класс в библиотеке sklearn, который используется для генерации временных рядов. Он позволяет создавать различные типы временных рядов, а также позволяет настраивать параметры генерации, такие как длина ряда, частота дискретизации и количество компонент.

**LSTM** (Long Short-Term Memory) - это вид рекуррентных нейронных сетей, разработанный для работы с последовательными данными. Он был разработан для решения проблемы исчезающего градиента, которая возникает при обучении традиционных рекуррентных нейронных сетей на долгосрочных зависимостях. LSTM использует специальные блоки, называемые ячейками памяти, которые позволяют сохранять информацию на протяжении длительного времени. Это делает их особенно полезными для задач, связанных с обработкой естественного языка, такими как машинный перевод, распознавание речи и ответы на вопросы.

**GRU** (Gated Recurrent Unit) - это другой тип рекуррентной нейронной сети, который был предложен в качестве альтернативы LSTM. GRU также использует механизм gating для управления потоком информации, но он проще, чем LSTM, и может быть более эффективным для некоторых задач. GRU используются в тех же областях, что и LSTM, включая обработку естественного языка и прогнозирование временных рядов.

**RMSE** (Root Mean Squared Error) – мера того, насколько точно модель предсказывает будущие значения временной последовательности. Он рассчитывается как квадратный корень из среднего квадратичного отклонения между предсказанными значениями и фактическими значениями.

В идеале RMSE должна быть как можно меньше. Это означает, что модель предсказывает будущие значения с высокой точностью.

# **Процесс выполнения работы**

**Подготовка**

Так как изначальный датасет не содержит в себе разделение на наборы обучения, необходимо разделить данные на обучающий и тестовый наборы, в соотношении ~80:20 (%). (В исходном файле - 325 записей)

Для дальнейшего обучения нужно подготовить исходные данные путем нормализации данных:

**scaler = MinMaxScaler()**

**scaler.fit(train)**

**scaled\_train = scaler.transform(train)**

**scaled\_test = scaler.transform(test)**

Так как в данном датасете рассматривается зависимость продаж от даты в году, укажем число важных входных признаков:

**n\_input = 12**

**n\_features = 1**

**generator = TimeseriesGenerator(scaled\_train, scaled\_train, length=n\_input, batch\_size=1)**

В данном случае 12 – кол-во месяцев в году, 1 – одна зависимая переменная – стоимость продукта. TimeseriesGenerator используется для генерации синтетических данных, которые нужны для обучения и тестирования моделей прогнозирования временных рядов. Это позволяет оценить производительность модели на различных типах данных и определить ее ограничения.

**Обучение на 3-х разных архитектурах**

Архитектуры RNN:

* Архитектура №1

В данной архитектуре используются следующие методы обучения: **SimpleRNN** – слой рекуррентной нейронной сети, работа с последовательностями данных, **Tanh** – это функция активации, которая возвращает их гиперболический тангенс, **Dense(32)** – используется для преобразования входных данных в выходные данные желаемой размерности, **RELU** – функция активации (до 0 – 0, после 0 – линейна), **Dense(1)** – используется для классификации и регрессии, позволяет сети изучать сложные функции и зависимости между входными данными и целевым значением, **Adam** – оптимизатор.

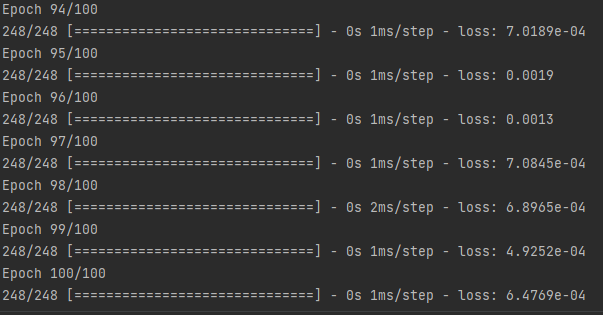
На данной архитектуре нейронной сети получаются следующие результаты:

Рис. 1 – Обучение нейронной сети на архитектуре №1

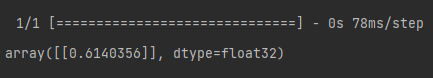


Рис. 2 – Точность работы нейронной сети на тестовом наборе

архитектуры №1

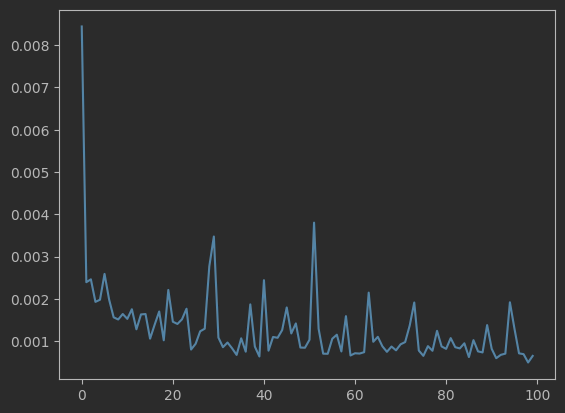


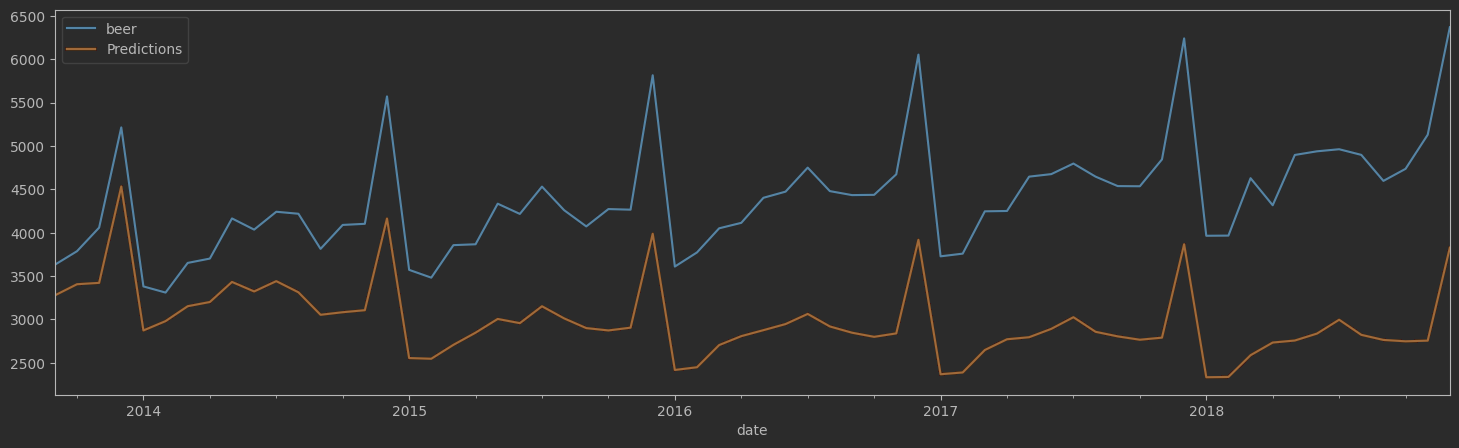
Рис. 3 – Зависимость потерь от кол-ва эпох на тренировочном наборе.

Рис. 4 – Зависимость продаж от времени на тренировочном и тестовом наборах.

Из данных графиков можно сделать вывод, что на данной архитектуре модель обучается недостаточно точно, множественные скачки в графике при обучении обусловлены малым числом слоев и неточной настройкой гипер параметров модели. Хоть при увеличении кол-ва эпох процент потерь падает, но все равно множественные скачки не дают модели возможности нормально обучиться. Значение RMSE – 732.93, достаточно велико, однако оправдано малой точностью модели.

* Архитектура №2

В данной архитектуре используются следующие методы обучения: **SimpleRNN** – слой рекуррентной нейронной сети, работа с последовательностями данных, **Tanh** – это функция активации, которая возвращает их гиперболический тангенс, **Dense**(64) – используется для преобразования входных данных в выходные данные желаемой размерности, **RELU** – функция активации (до 0 – 0, после 0 – линейна), **Dense**(64) – используется для преобразования входных данных в выходные данные желаемой размерности, **Dense**(32) – дополнительный слой сжатия с функцией активации **RELU**, **Dense**(1) – используется для классификации и регрессии, позволяет сети изучать сложные функции и зависимости между входными данными и целевым значением, **rmsprop** – оптимизатор.(**rmsprop** использует корень из среднего квадрата градиентов для обновления весов, в то время как **Adam** использует адаптивную оценку моментов)

На данной архитектуре нейронной сети получаются следующие результаты:

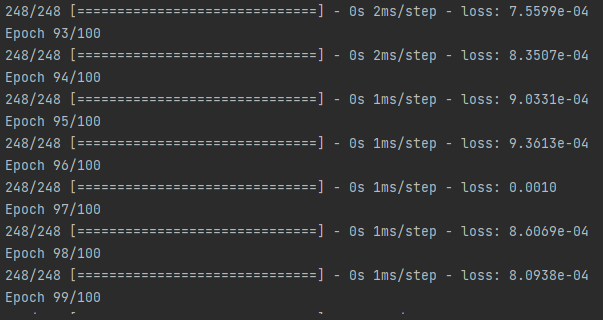


Рис. 5 – Обучение нейронной сети на архитектуре №2

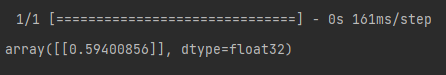


Рис. 6 – Точность работы нейронной сети на тестовом наборе

архитектуры №2

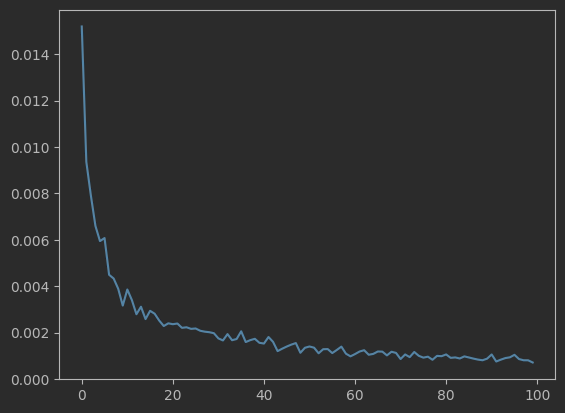


Рис. 7 – Зависимость потерь от кол-ва эпох на тренировочном наборе.

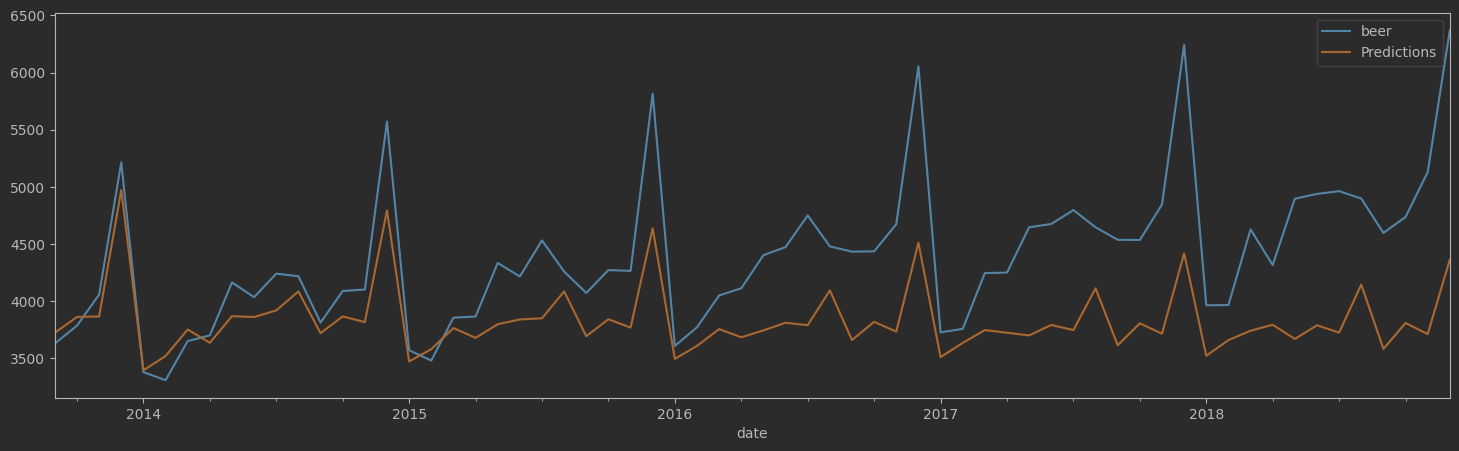


Рис. 8 – Зависимость продаж от времени на тренировочном и тестовом наборах.

Благодаря добавленному слою **Dense(64)** и еще одной функции активации **RELU**, данная модель значительно улучшила свои показатели. Исходя из данных на графиках можно сделать вывод, что теперь модель более точно предполагает нужные значения, однако добавив еще пару слоев, можно добится более гладких результатов 2-го графика. Значение **RMSE** – 343.41, все еще большое, но уже лучше чем на 1-ой архитектуре.

* Архитектура №3

Для получения более гладких результатов, в данную архитектуру был добавлен еще один слой **SimpleRNN** – слой рекуррентной нейронной сети с функцией активации **RELU**. **Adam –** оптимизатор. Это позволило более точно выделять нужные признаки.

На данной архитектуре нейронной сети получаются следующие результаты:

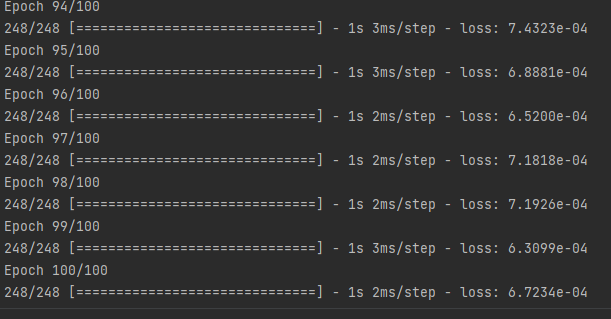


Рис. 9 – Обучение нейронной сети на архитектуре №3

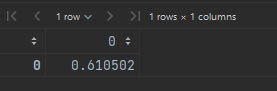


Рис. 10 – Точность работы нейронной сети на тестовом наборе

архитектуры №3

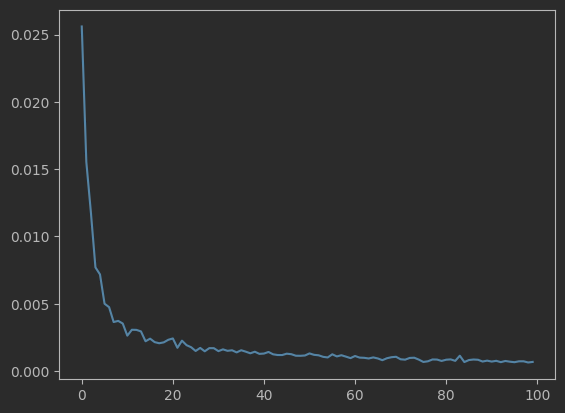


Рис. 11 – Зависимость потерь от кол-ва эпох на тренировочном наборе.

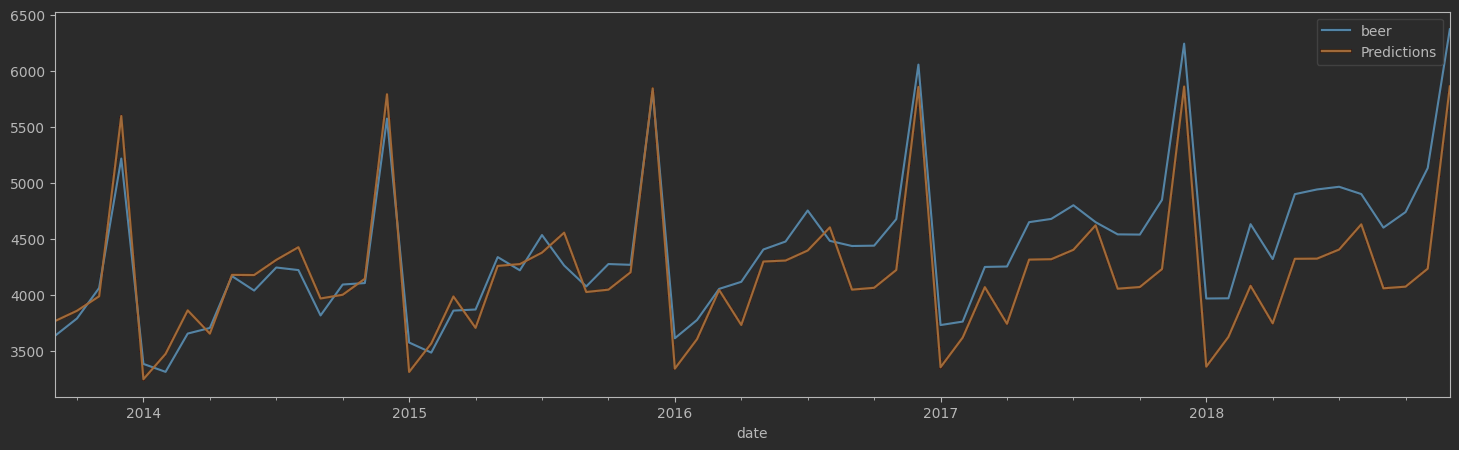


Рис. 12 – Зависимость продаж от времени на тренировочном и тестовом наборах.

Благодаря добавлению еще одного слоя архитектура показала лучшие результаты. На данной архитектуре оптимизатор Adam – показал результаты лучше чем другие, однако это не гарантирует, то что он всегда выгоднее. Точность работы на данной архитектуре хоть и не превышает 70%, но тестовый набор проходит достаточно успешно. Для увеличения точности могут использоваться добавочные слои и функции, а также перебор разных гипер параметров модели. Значение RMSE – 205.1, указывает на то что, модель обучена хорошо, не идеально, но при этом нет переобучения.

**Использование LSTM и GRU**

* Архитектура **LSTM**

На данной архитектуре нейронной сети получаются следующие результаты:

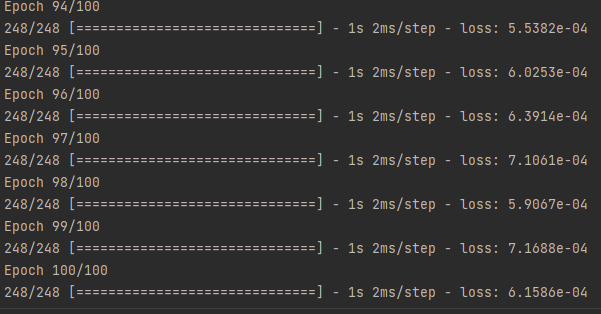


Рис. 13 – Обучение нейронной сети на архитектуре LSTM

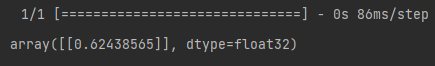


Рис. 14 – Точность работы нейронной сети на тестовом наборе

архитектуры LSTM

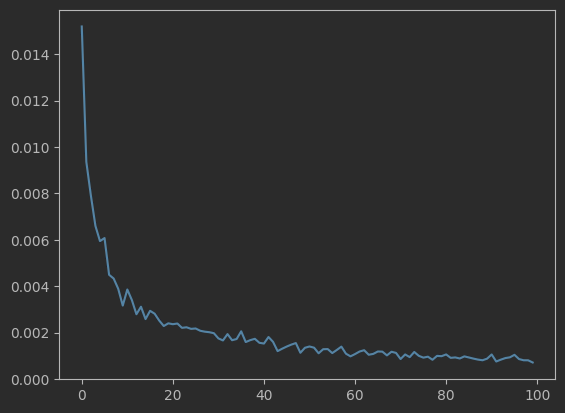


Рис. 15 – Зависимость потерь от кол-ва эпох на тренировочном наборе.

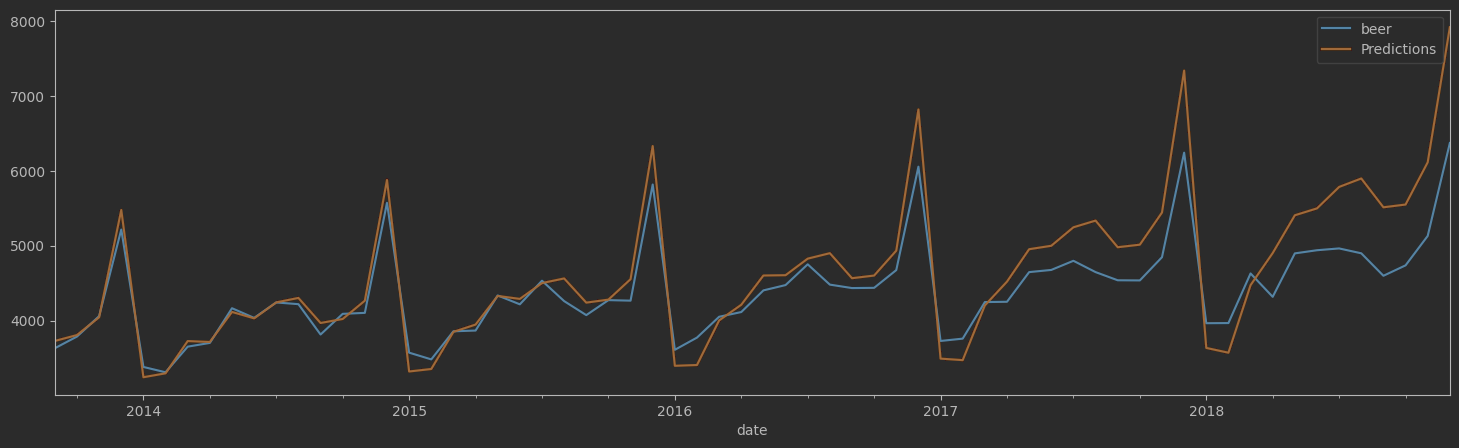


Рис. 16 – Зависимость продаж от времени на тренировочном и тестовом наборах.

* Архитектура **GRU**

На данной архитектуре нейронной сети получаются следующие результаты:

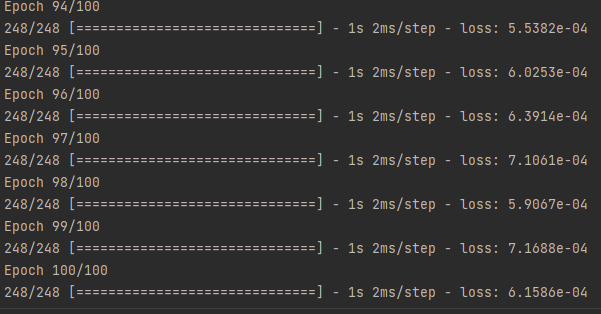


Рис. 17 – Обучение нейронной сети на архитектуре GRU

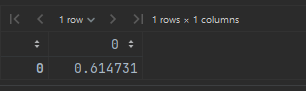


Рис. 18 – Точность работы нейронной сети на тестовом наборе

архитектуры GRU

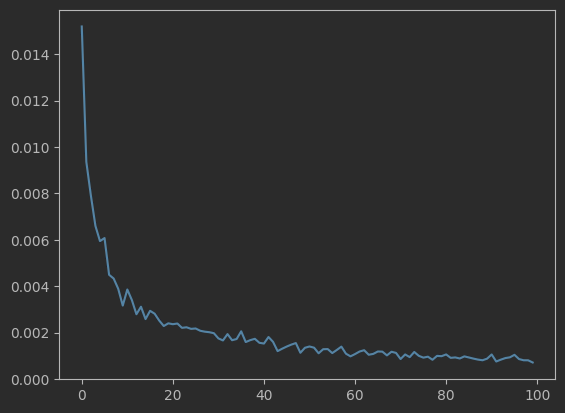


Рис. 19 – Зависимость потерь от кол-ва эпох на тренировочном наборе.

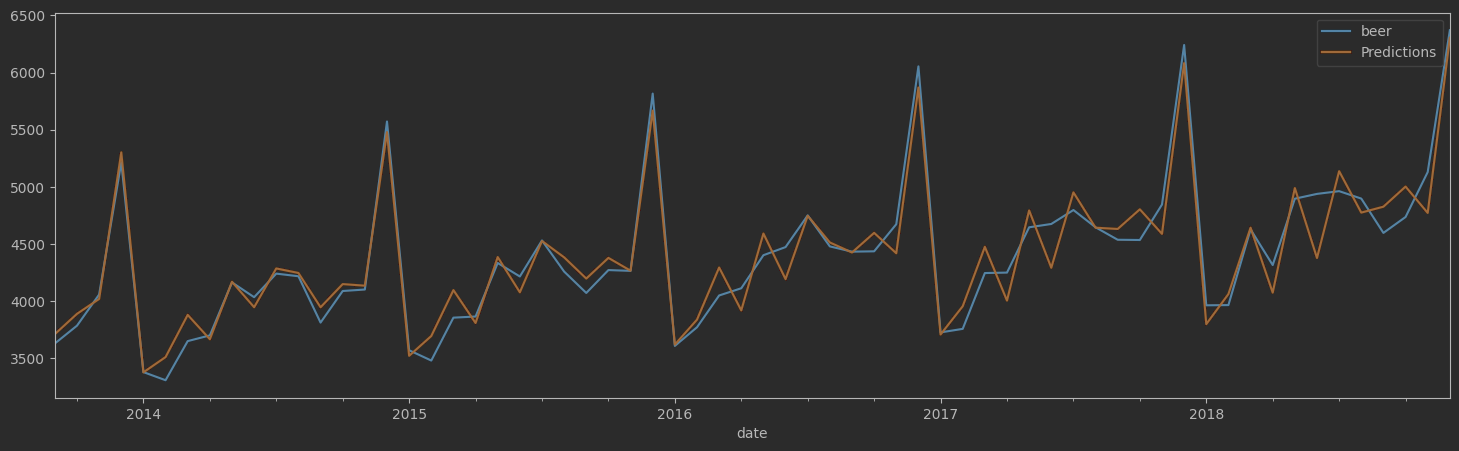


Рис. 20 – Зависимость продаж от времени на тренировочном и тестовом наборах.

Данные архитектурные показали результаты лучше чем “самодельные”. Это обусловлено тем, что данные модели имеют более сложную структуру слоев, при этом данный подход лишен проблемы исчезающего градиента, присутствующей в *SimpleRNN*. По графикам можно точно сказать, что данные архитектуры намного лучше работают на тестовом наборе, выделяя более явные признаки сильнее. Точность менее 70% обусловлена малой выборкой данных, поэтому для данного датасета, можно использовать как архитектуры разработанные в начале лабораторной работы, так и готовые варианты, однако при работы с более большими наборами данных, *LSTM* или *GRU* покажут более точный результат. Показатели RMSE – в 173 и 185 единиц говорят о хорошей обученности модели и отсутствии переобучения.

# **Вывод**

В рамках проведенной лабораторной работы были разработаны и протестированы три собственные архитектуры рекуррентных нейронных сетей, а также применены готовые решения обучения ***LSTM*** и ***GRU***. Было отмечено, что изменение гипер параметров и добавление нескольких слоев, могут привести к улучшению точности модели.

Самостоятельно разработанные архитектуры постепенно улучшали свои показатели, так как с каждой новой архитектурой, они получали некоторые добавочные элементы, улучшающие процесс обучения модели.

На данных архитектурах удалось добиться 62% точности работы на тестовом наборе данных. Для данного датасета, данный процент вполне можно считать приемлемым, учитывая малую выборку данных и “самодельные” архитектуры.

Среди полностью самостоятельно созданных архитектур лучшие результаты продемонстрировала третья архитектура: она показала наилучшие метрики качества. Исходя из этого, если же перед пользователем стоит выбор между тремя данными архитектурами, рекомендуется остановить свой выбор именно на третьей архитектуре.

Однако уже готовые и протестированные модели *LSTM* или *GRU* будут выдавать лучшие результаты в ходе трансферного обучения. Так модель на основе *LSTM* продемонстрировала лучшие показатели эффективности по сравнению с моделями сделанными в ходе лабораторной работы.

# **Код программы**

**lab2\_1.ipynb**

**#%%**

**import pandas as pd**

**import numpy as np**

**import matplotlib.pyplot as plt**

**from statsmodels.tsa.seasonal import seasonal\_decompose**

**from sklearn.preprocessing import MinMaxScaler**

**Alco = pd.read\_csv('BeerWineLiquor.csv', index\_col='date', parse\_dates=True)**

**Alco.index.freq='MS'**

**#%%**

**train = Alco.iloc[:260]**

**test = Alco.iloc[260:]**

**#%%**

**scaler = MinMaxScaler()**

**#%%**

**scaler.fit(train)**

**scaled\_train = scaler.transform(train)**

**scaled\_test = scaler.transform(test)**

**#%%**

**from keras.preprocessing.sequence import TimeseriesGenerator**

**#%%**

**n\_input = 12**

**n\_features = 1**

**generator = TimeseriesGenerator(scaled\_train, scaled\_train, length=n\_input, batch\_size=1)**

**#%%**

**from keras.models import Sequential**

**from keras.layers import Dense**

**from keras.layers import SimpleRNN**

**model\_rnn\_1 = Sequential()**

**model\_rnn\_1.add(SimpleRNN(64, activation='tanh', input\_shape=(n\_input, n\_features)))**

**model\_rnn\_1.add(Dense(32, activation='relu'))**

**model\_rnn\_1.add(Dense(1))**

**model\_rnn\_1.compile(optimizer='adam', loss='mse')**

**model\_rnn\_2 = Sequential()**

**model\_rnn\_2.add(SimpleRNN(128, activation='tanh', input\_shape=(n\_input, n\_features)))**

**model\_rnn\_2.add(Dense(64, activation='relu'))**

**model\_rnn\_2.add(Dense(32, activation='relu'))**

**model\_rnn\_2.add(Dense(1, activation='linear'))**

**model\_rnn\_2.compile(optimizer='rmsprop', loss='mse')**

**model\_rnn\_3 = Sequential()**

**model\_rnn\_3.add(SimpleRNN(256, activation='relu', input\_shape=(n\_input, n\_features), return\_sequences=True))**

**model\_rnn\_3.add(SimpleRNN(128, activation='relu'))**

**model\_rnn\_3.add(Dense(64, activation='relu'))**

**model\_rnn\_3.add(Dense(32, activation='relu'))**

**model\_rnn\_3.add(Dense(1))**

**model\_rnn\_3.compile(optimizer='Adam', loss='mse')**

**#%%**

**model\_rnn\_2.summary()**

**#%%**

**model\_rnn\_3.summary()**

**#%%**

**model\_rnn\_1.fit(generator,epochs=100)**

**#%%**

**model\_rnn\_2.fit(generator,epochs=100)**

**#%%**

**model\_rnn\_3.fit(generator,epochs=100)**

**#%%**

**loss\_per\_epoch = model\_rnn\_1.history.history['loss']**

**plt.plot(range(len(loss\_per\_epoch)),loss\_per\_epoch)**

**#%%**

**loss\_per\_epoch = model\_rnn\_2.history.history['loss']**

**plt.plot(range(len(loss\_per\_epoch)),loss\_per\_epoch)**

**#%%**

**loss\_per\_epoch = model\_rnn\_3.history.history['loss']**

**plt.plot(range(len(loss\_per\_epoch)),loss\_per\_epoch)**

**#%%**

**last\_train\_batch = scaled\_train[-n\_input:]**

**last\_train\_batch = last\_train\_batch.reshape((1, n\_input, n\_features))**

**model\_rnn\_1.predict(last\_train\_batch)**

**#%%**

**test\_predictions = []**

**first\_eval\_batch = scaled\_train[-n\_input:]**

**current\_batch = first\_eval\_batch.reshape((1, n\_input, n\_features))**

**for i in range(len(test)):**

**# get the prediction value for the first batch**

**current\_pred = model\_rnn\_1.predict(current\_batch)[0]**

**# append the prediction into the array**

**test\_predictions.append(current\_pred)**

**# use the prediction to update the batch and remove the first value**

**current\_batch = np.append(current\_batch[:,1:,:],[[current\_pred]],axis=1)**

**#%%**

**true\_predictions = scaler.inverse\_transform(test\_predictions)**

**test['Predictions'] = true\_predictions**

**#%%**

**test.plot(figsize=(18,5))**

**#%%**

**from sklearn.metrics import mean\_squared\_error**

**from math import sqrt**

**rmse=sqrt(mean\_squared\_error(test['beer'],test['Predictions']))**

**print(rmse)**

**#%%**

**last\_train\_batch = scaled\_train[-n\_input:]**

**last\_train\_batch = last\_train\_batch.reshape((1, n\_input, n\_features))**

**model\_rnn\_2.predict(last\_train\_batch)**

**#%%**

**test\_predictions = []**

**first\_eval\_batch = scaled\_train[-n\_input:]**

**current\_batch = first\_eval\_batch.reshape((1, n\_input, n\_features))**

**for i in range(len(test)):**

**# get the prediction value for the first batch**

**current\_pred = model\_rnn\_2.predict(current\_batch)[0]**

**# append the prediction into the array**

**test\_predictions.append(current\_pred)**

**# use the prediction to update the batch and remove the first value**

**current\_batch = np.append(current\_batch[:,1:,:],[[current\_pred]],axis=1)**

**#%%**

**true\_predictions = scaler.inverse\_transform(test\_predictions)**

**test['Predictions'] = true\_predictions**

**#%%**

**test.plot(figsize=(18,5))**

**#%%**

**from sklearn.metrics import mean\_squared\_error**

**from math import sqrt**

**rmse=sqrt(mean\_squared\_error(test['beer'],test['Predictions']))**

**print(rmse)**

**#%%**

**last\_train\_batch = scaled\_train[-n\_input:]**

**last\_train\_batch = last\_train\_batch.reshape((1, n\_input, n\_features))**

**model\_rnn\_3.predict(last\_train\_batch)**

**#%%**

**test\_predictions = []**

**first\_eval\_batch = scaled\_train[-n\_input:]**

**current\_batch = first\_eval\_batch.reshape((1, n\_input, n\_features))**

**for i in range(len(test)):**

**# get the prediction value for the first batch**

**current\_pred = model\_rnn\_3.predict(current\_batch)[0]**

**# append the prediction into the array**

**test\_predictions.append(current\_pred)**

**# use the prediction to update the batch and remove the first value**

**current\_batch = np.append(current\_batch[:,1:,:],[[current\_pred]],axis=1)**

**#%%**

**true\_predictions = scaler.inverse\_transform(test\_predictions)**

**test['Predictions'] = true\_predictions**

**#%%**

**test.plot(figsize=(18,5))**

**#%%**

**from sklearn.metrics import mean\_squared\_error**

**from math import sqrt**

**rmse=sqrt(mean\_squared\_error(test['beer'],test['Predictions']))**

**print(rmse)**

**#%%**

**lab2\_LSTM**

**#%%**

**import pandas as pd**

**import numpy as np**

**import matplotlib.pyplot as plt**

**#%%**

**Alco = pd.read\_csv('BeerWineLiquor.csv', index\_col='date', parse\_dates=True)**

**#%%**

**Alco.index.freq='MS'**

**#%%**

**from statsmodels.tsa.seasonal import seasonal\_decompose**

**#%%**

**train = Alco.iloc[:260]**

**test = Alco.iloc[260:]**

**#%%**

**from sklearn.preprocessing import MinMaxScaler**

**scaler = MinMaxScaler()**

**#%%**

**scaler.fit(train)**

**scaled\_train = scaler.transform(train)**

**scaled\_test = scaler.transform(test)**

**#%%**

**from keras.preprocessing.sequence import TimeseriesGenerator**

**#%%**

**n\_input = 12**

**n\_features = 1**

**generator = TimeseriesGenerator(scaled\_train, scaled\_train, length=n\_input, batch\_size=1)**

**#%%**

**from keras.models import Sequential**

**from keras.layers import Dense**

**from keras.layers import LSTM**

**model = Sequential()**

**model.add(LSTM(64, activation='relu', input\_shape=(n\_input, n\_features)))**

**model.add(Dense(32, activation='relu'))**

**model.add(Dense(16, activation='relu'))**

**model.add(Dense(1))**

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

**#%%**

**model.summary()**

**#%%**

**model.fit(generator,epochs=100)**

**#%%**

**loss\_per\_epoch = model.history.history['loss']**

**plt.plot(range(len(loss\_per\_epoch)),loss\_per\_epoch)**

**#%%**

**last\_train\_batch = scaled\_train[-n\_input:]**

**last\_train\_batch = last\_train\_batch.reshape((1, n\_input, n\_features))**

**model.predict(last\_train\_batch)**

**#%%**

**scaled\_test[0]**

**#%%**

**test\_predictions = []**

**first\_eval\_batch = scaled\_train[-n\_input:]**

**current\_batch = first\_eval\_batch.reshape((1, n\_input, n\_features))**

**for i in range(len(test)):**

**current\_pred = model.predict(current\_batch)[0]**

**test\_predictions.append(current\_pred)**

**current\_batch = np.append(current\_batch[:,1:,:],[[current\_pred]],axis=1)**

**#%%**

**test.head()**

**#%%**

**true\_predictions = scaler.inverse\_transform(test\_predictions)**

**test['Predictions'] = true\_predictions**

**#%%**

**test.plot(figsize=(18,5))**

**#%%**

**from sklearn.metrics import mean\_squared\_error**

**from math import sqrt**

**rmse=sqrt(mean\_squared\_error(test['beer'],test['Predictions']))**

**print(rmse)**

**#%%**

**lab2\_GRU**

**#%%**

**import pandas as pd**

**import numpy as np**

**import matplotlib.pyplot as plt**

**#%%**

**Alco = pd.read\_csv('BeerWineLiquor.csv', index\_col='date', parse\_dates=True)**

**#%%**

**Alco.index.freq='MS'**

**#%%**

**Alco.plot(figsize=(12,6))**

**#%%**

**from statsmodels.tsa.seasonal import seasonal\_decompose**

**#%%**

**train = Alco.iloc[:260]**

**test = Alco.iloc[260:]**

**#%%**

**from sklearn.preprocessing import MinMaxScaler**

**scaler = MinMaxScaler()**

**scaler.fit(train)**

**scaled\_train = scaler.transform(train)**

**scaled\_test = scaler.transform(test)**

**from keras.preprocessing.sequence import TimeseriesGenerator**

**#%%**

**n\_input = 12**

**n\_features = 1**

**generator = TimeseriesGenerator(scaled\_train, scaled\_train, length=n\_input, batch\_size=1)**

**#%%**

**from keras.models import Sequential**

**from keras.layers import Dense**

**from keras.layers import GRU**

**model = Sequential()**

**model.add(GRU(100, activation='relu', input\_shape=(n\_input, n\_features)))**

**model.add(Dense(1))**

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

**#%%**

**model.fit(generator,epochs=100)**

**#%%**

**loss\_per\_epoch = model.history.history['loss']**

**plt.plot(range(len(loss\_per\_epoch)),loss\_per\_epoch)**

**#%%**

**last\_train\_batch = scaled\_train[-12:]**

**last\_train\_batch = last\_train\_batch.reshape((1, n\_input, n\_features))**

**model.predict(last\_train\_batch)**

**#%%**

**scaled\_test[0]**

**#%%**

**test\_predictions = []**

**first\_eval\_batch = scaled\_train[-n\_input:]**

**current\_batch = first\_eval\_batch.reshape((1, n\_input, n\_features))**

**for i in range(len(test)):**

**current\_pred = model.predict(current\_batch)[0]**

**test\_predictions.append(current\_pred)**

**current\_batch = np.append(current\_batch[:,1:,:],[[current\_pred]],axis=1)**

**#%%**

**true\_predictions = scaler.inverse\_transform(test\_predictions)**

**test['Predictions'] = true\_predictions**

**#%%**

**test.plot(figsize=(18,5))**

**#%%**

**from sklearn.metrics import mean\_squared\_error**

**from math import sqrt**

**rmse = sqrt(mean\_squared\_error(test['beer'], test['Predictions']))**

**print(rmse)**

**#%%**