МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное

образовательное учреждение высшего образования

«Самарский национальный исследовательский университет

имени академика С.П. Королева»

Институт информатики и кибернетики

Факультет информатики

Кафедра программных систем

ОТЧЁТ

по лабораторной работе № 2

«Прогнозирование временных рядов»

по курсу «Нейронные сети»

Выполнил:

Портнов К.А. Хорина В.И.

гр. 6132-020402D

Проверила:

Жданова А.Н.

Самара 2022

Оглавление

[Математическая постановка задачи 4](#_Toc122611821)

[Архитектура сети, параметры, методы обучения 5](#_Toc122611822)

[Деление данных на выборки 6](#_Toc122611823)

[Окно данных 7](#_Toc122611824)

[Создание датасета 9](#_Toc122611825)

[Одноступенчатые модели 9](#_Toc122611826)

[Базовый уровень 10](#_Toc122611827)

[Линейная модель 10](#_Toc122611828)

[Плотная модель 11](#_Toc122611829)

[Многоплотная модель 11](#_Toc122611830)

[Сверточная модель 12](#_Toc122611831)

[Реккурентная модель 12](#_Toc122611832)

[Модели с несколькими выходами 12](#_Toc122611833)

[Базовая 13](#_Toc122611834)

[Плотная 13](#_Toc122611835)

[RNN 13](#_Toc122611836)

[Остаточная RNN 13](#_Toc122611837)

[Многошаговые модели 14](#_Toc122611838)

[Базовые показатели 15](#_Toc122611839)

[Однозарядные модели 15](#_Toc122611840)

[Линейная модель 16](#_Toc122611841)

[Плотная модель 16](#_Toc122611842)

[Сверточная 16](#_Toc122611843)

[Реккурентная 16](#_Toc122611844)

[Авторегрессионная модель 16](#_Toc122611845)

[Вычислительные эксперименты 18](#_Toc122611846)

[Одноступенчатые модели 18](#_Toc122611847)

[Базовый уровень 18](#_Toc122611848)

[Линейчатая модель 18](#_Toc122611849)

[Плотная модель 19](#_Toc122611850)

[Многоплотная модель 19](#_Toc122611851)

[Сверточная модель 20](#_Toc122611852)

[Реккурентная 20](#_Toc122611853)

[Многошаговые модели 21](#_Toc122611854)

[Базовая(линия) 21](#_Toc122611855)

[Базовый(повторение) 22](#_Toc122611856)

[Линейная 22](#_Toc122611857)

[Плотная 23](#_Toc122611858)

[Сверточная 23](#_Toc122611859)

[Реккурентная 24](#_Toc122611860)

[Авторегрессионная модель 24](#_Toc122611861)

[Результаты работы 25](#_Toc122611862)

[Для одношаговых 25](#_Toc122611863)

[Для многошаговых. 27](#_Toc122611864)

[Заключение 27](#_Toc122611865)

[Ссылки 28](#_Toc122611866)

[Листинг программы 28](#_Toc122611867)

# Математическая постановка задачи

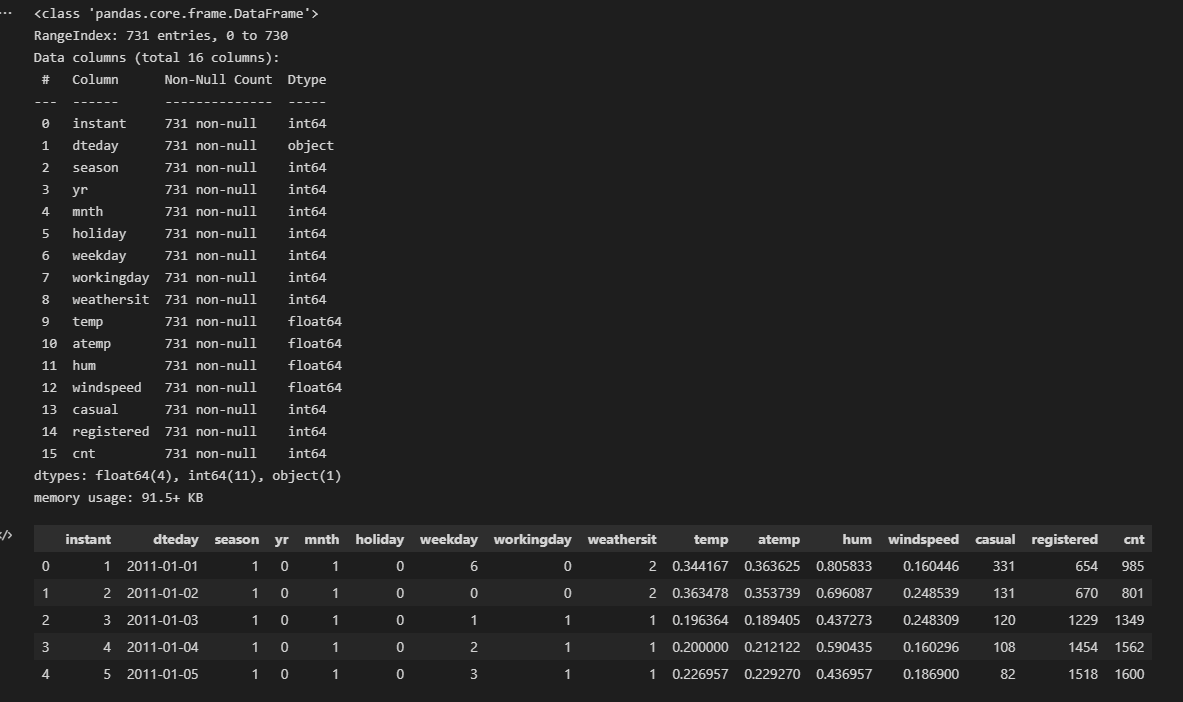
Набор данных содержит 731 строку данных взятых в период с 20011-01-01 по 2012-12-31. Набор данных основан на показателях по аренде велосипедов.

1. instant: индекс;
2. dteday: дата;
3. season: сезон (1:весна, 2:лето, 3:осень, 4:зима);
4. yr: год (0: 2011, 1:2012);
5. mnth: месяц ( 1 to 12);
6. hr: час (0 to 23);
7. holiday: выходной день (1 да, 0 нет);
8. weekday: день недели;
9. workingday: если день не является ни выходным, ни праздничным днем, равен 1, иначе равен 0;
10. погода:
    1. 1: ясно, небольшая облачность, переменная облачность, переменная облачность
    2. 2: Туман + Облачно, Туман + Разорванные облака, Туман + Немного облаков, Туман
    3. 3: Слабый снег, Слабый дождь + Гроза + Рассеянные облака, Слабый дождь + Рассеянные облака
    4. 4: Сильный дождь + Ледяные поддоны + Гроза + Туман, Снег + Туман
11. temp: Нормализованная температура в градусах Цельсия. Значения делятся на 41 (макс.);
12. atemp: Нормализовалось ощущение температуры в градусах Цельсия. Значения делятся на 50 (макс.);
13. hum: Нормированная влажность. Значения делятся на 100 (макс.);
14. windspeed: Нормированная скорость ветра. Значения делятся на 67 (макс.);
15. casual: количество случайных пользователей;
16. registered: количество зарегистрированных пользователей;
17. cnt: общее количество арендованных велосипедов, включая обычные и зарегистрированные.

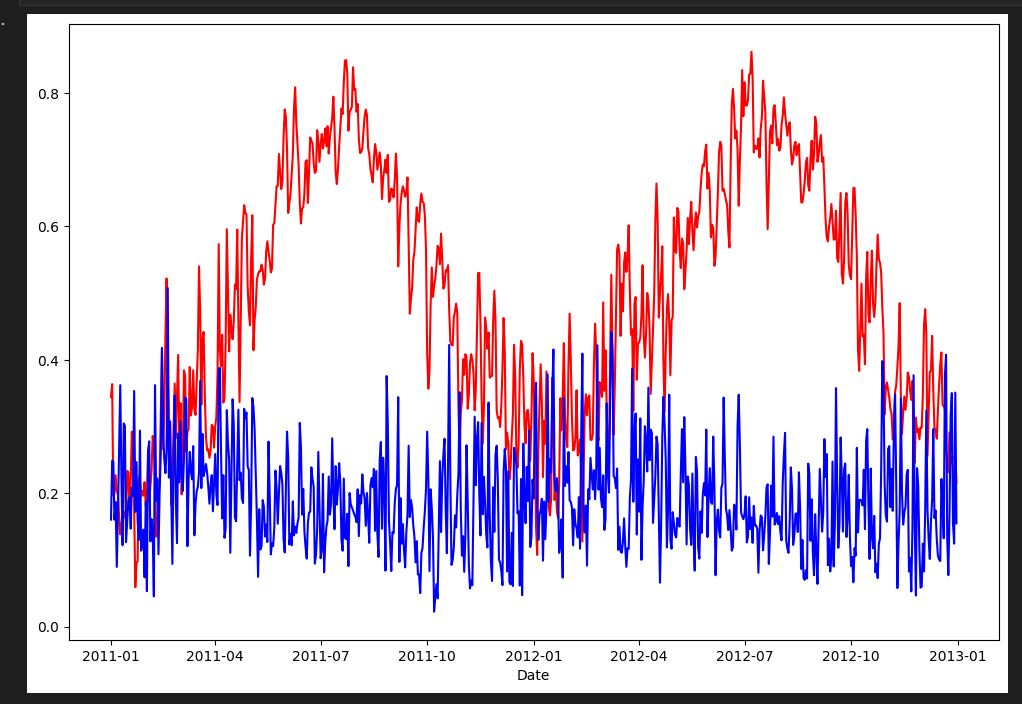
# Архитектура сети, параметры, методы обучения

Для выполнения работы использовался фреймворк TensorFlow, а также вспомогательные библиотеки Python. С помощью данного фреймворка были построены модели для прогнозирования временных рядов.

Загрузим эти данные с диска и прочитаем их при помощи библиотеки pandas (файл формата \*csv).



Выведем некоторые функции с течением времени



Деление данных на выборки

Будем использовать разделение (70%, 20%, 10%) для обучающих, проверочных и тестовых наборов. Обратите внимание, что данные не перемешиваются случайным образом перед разделением. Это по двум причинам:

1. Это гарантирует, что разделение данных на окна последовательных выборок по-прежнему возможно.
2. Это гарантирует, что результаты проверки/тестирования будут более реалистичными, поскольку они оцениваются на основе данных, собранных после обучения модели

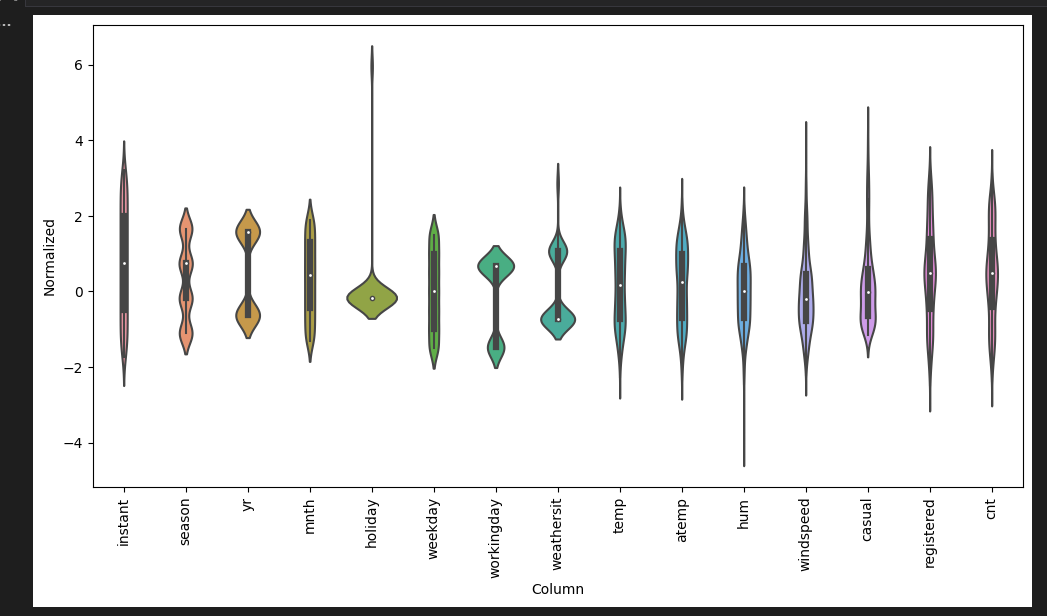
Нормализация данных

Перед обучением нейронной сети важно масштабировать функции. Нормализация – это распространенный способ масштабирования: вычесть среднее значение и разделить на стандартное отклонение каждого признака.

Среднее значение и стандартное отклонение следует вычислять только с использованием обучающих данных, чтобы модели не имели доступа к значениям в проверочных и тестовых наборах.

Также можно утверждать, что модель не должна иметь доступа к будущим значениям в тренировочном наборе во время обучения и что эта нормализация должна выполняться с использованием скользящих средних.

Взглянем на распределение функций.



Окно данных

Модели будут делать набор прогнозов на основе окна последовательных выборок из данных.

Основные особенности окон ввода:

* Ширина (количество временных шагов) окон ввода и метки.
* Смещение времени между ними.
* Какие функции используются в качестве входных данных, меток или того и другого.

Создаются различные модели (включая линейные модели, модели DNN, CNN и RNN) и используются они для обеих целей:

* Прогнозы с одним выходом и несколькими выходами.
* Прогнозы с одним и несколькими временными шагами.

В этом разделе основное внимание уделяется реализации окна данных, чтобы его можно было повторно использовать для всех этих моделей.

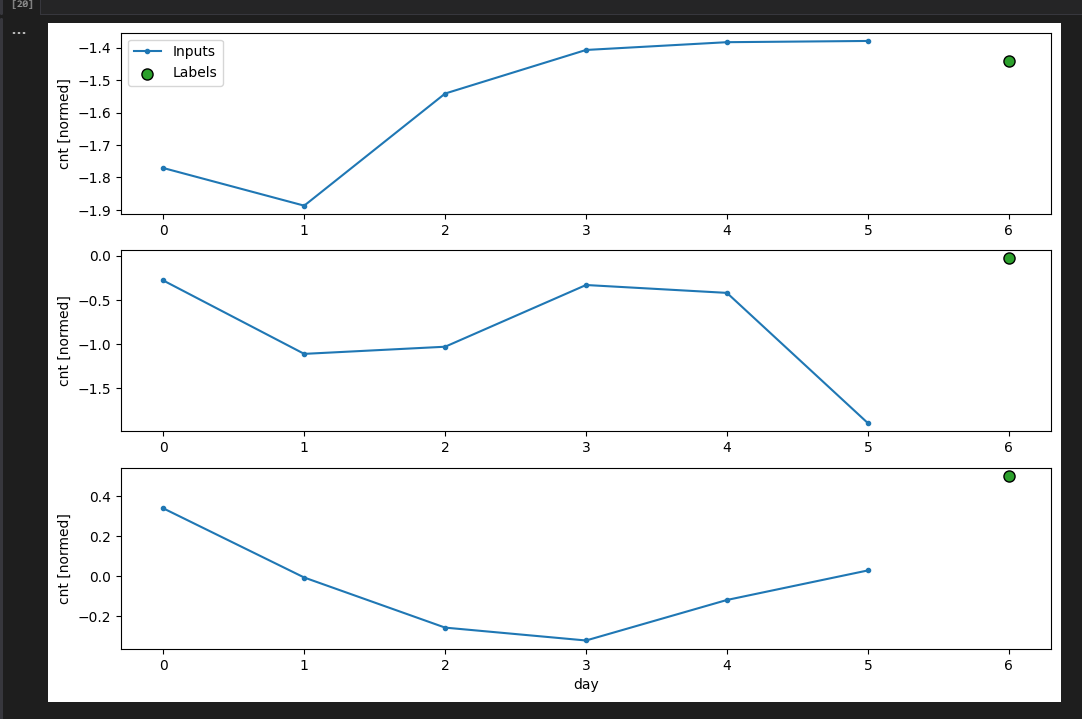
В зависимости от задачи и типа модели может потребоваться создание различных окон данных.

Начнем с создания класса WindowGenerator. Метод \_\_init\_\_ включает всю необходимую логику для индексов ввода и меток.

Он также принимает обучающие, оценочные и тестовые кадры данных в качестве входных данных. Позже они будут преобразованы в tf.data.Dataset окон.

Учитывая список последовательных входных данных, метод split\_window преобразует их в окно входных данных и окно меток.

Этот график выравнивает входные данные, метки и (более поздние) прогнозы на основе времени, к которому относится элемент:

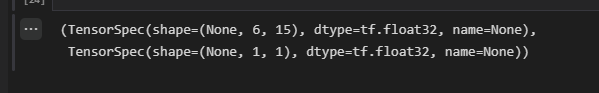


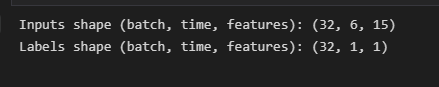
Создание датасета

Наконец, этот метод make\_dataset возьмет временной ряд DataFrame и преобразует его в tf.data.Dataset из пар (input\_window, label\_window) с помощью функции tf.keras.utils.timeseries\_dataset\_from\_array

Объект WindowGenerator содержит обучающие, проверочные и тестовые данные.

Свойство Dataset.element\_spec сообщает нам структуру, типы данных и формы элементов набора данных. Итерация по Dataset дает конкретные партии.





Одноступенчатые модели

Самая простая модель, которую вы можете построить на такого рода данных, – это модель, которая предсказывает значение одной функции – 1 временной шаг (один день) в будущее, основываясь только на текущих условиях.

Итак, начнем с построения моделей для прогнозирования значения Open (цены открытия) на один день вперед.

Настроим объект WindowGenerator для создания этих одношаговых пар (input, label). Объект window создает tf.data.Dataset из обучающих, проверочных и тестовых наборов, что позволяет легко перебирать пакеты данных.

### Базовый уровень

Перед созданием обучаемой модели было бы неплохо иметь базовый уровень производительности в качестве точки для сравнения с более поздними более сложными моделями.

Эта первая задача состоит в том, чтобы предсказать цену открытия на один день вперед, учитывая текущее значение всех признаков. Текущие значения включают текущую цену.

Итак, начнем с модели, которая просто возвращает текущую цену в качестве прогноза, прогнозируя «без изменений». Это разумная базовая линия, поскольку цена изменяется медленно.

В качестве потерь используем tf.losses.MeanSquaredError. Вычисляет среднее квадратическое значение ошибок между метками и прогнозами.

В качестве метрик tf.metrics.MeanAbsoluteError. Вычисляет среднюю абсолютную ошибку между метками и прогнозами.

Модель делает прогнозы на один день вперед на основе одного входного временного шага.

Метки, сдвинуты вправо на один день.

### Линейная модель

Самая простая обучаемая модель, которую вы можно применить к этой задаче, – это вставить линейное преобразование между входом и выходом. В этом случае результат временного шага зависит только от этого шага:

Слой tf.keras.layers.Dense без набора activation является линейной моделью. Слой преобразует только последнюю ось данных из (batch, time, inputs) в (batch, time, units); он применяется независимо к каждому элементу по осям batch и time.

В данной модели используется лишь 1 слой Dense.

Так как мы будем обучать множество моделей, то упакуем процедуру обучения в функцию def.

В качестве потерь используем tf.losses.MeanSquaredError. Вычисляет среднее квадратическое значение ошибок между метками и прогнозами.

В качестве метрик tf.optimizers.Adam. Вычисляет среднюю абсолютную ошибку между метками и прогнозами.

В функции активации используем tf.losses.MeanSquaredError. Вычисляет среднее квадратическое значение ошибок между метками и прогнозами.

Также в качестве ранней остановки используем функцию tf.keras.callbacks.EarlyStopping, чтобы прекратите обучение, когда отслеживаемый показатель перестал улучшаться.

Количество эпох составляет 40.

### Плотная модель

В данной модели используются 3 слоя полносвязных слоёв Dense. Первые 2 из которых состоят из 64 нейронов. В качестве функции активации используется – нелинейная функция и (Relu). А также выходной слой Dense как в линейной функции.

Обучим модель при помощи, ранее упакованной в функцию def процедуры обучения.

### Многоплотная модель

Baseline, linear и dense модели обрабатывали каждый временной шаг независимо. Здесь модель будет принимать несколько временных шагов в качестве входных данных для получения одного вывода.

В данной модели Flatten используется в качестве первого слоя и выравнивает входные данные модели затем используются 3 слоя полносвязных слоёв Dense. Первые 2 из которых состоят из 32 нейронов. В качестве функции активации используется – нелинейная функция и (Relu). А также выходной слой Dense как в линейной функции. На выходе используется слой Reshape который преобразует данные в заданную форму.

Обучим модель при помощи, ранее упакованной в функцию def процедуры обучения.

### Сверточная модель

Слой свертки ( tf.keras.layers.Conv1D ) также использует несколько временных шагов в качестве входных данных для каждого прогноза.

Представлена та же модель, что и в multi\_step\_dense, переписанная с помощью свертки.

tf.keras.layers.Flatten и первый tf.keras.layers.Dense заменяются tf.keras.layers.Conv1D. В нем задается размер свертки равный 3.

tf.keras.layers.Reshape больше не нужен, так как свертка сохраняет ось времени в своих выходных данных.

Обучим модель при помощи, ранее упакованной в функцию def процедуры обучения.

### Реккурентная модель

Рекуррентная нейронная сеть (RNN) — это тип нейронной сети, хорошо подходящий для данных временных рядов. RNN обрабатывают временной ряд шаг за шагом, сохраняя внутреннее состояние от шага к шагу.

Важным аргументом конструктора для всех слоев Keras RNN, таких как tf.keras.layers.LSTM , является аргумент return\_sequences . Этот параметр может настроить слой.

В данной модели LSTM используется в качестве первого слоя и запоминает данные. Он состоит из 32 нейронов, а также параметр return\_sequences, который находится в состоянии True, слой возвращает результат для каждого входа. А также выходной слой Dense как в линейной функции.

С return\_sequences=True модель можно обучать на данных за 24 дня за раз.

## Модели с несколькими выходами

До сих пор все модели предсказывали одну выходную характеристику T (degC) для одного временного шага.

Все эти модели можно преобразовать для прогнозирования нескольких функций, просто изменив количество единиц в выходном слое и настроив окна обучения, чтобы включить все функции в labels ( example\_labels ).

### Базовая

Используем ту же базовую модель ( Baseline ), но на этот раз с повторением всех функций вместо выбора конкретного label\_index.

### Плотная

Используем ту же Dense модель ( Dense ), но на этот раз с повторением всех функций вместо выбора конкретного label\_index.

### RNN

Используем ту же RNN модель ( LSTM ), но на этот раз с повторением всех функций вместо выбора конкретного label\_index.

### Остаточная RNN

Baseline модель, использованная ранее, использовала тот факт, что последовательность не сильно меняется от одного временного шага к другому. Каждая модель, обученная в этом руководстве до сих пор, была случайным образом инициализирована, а затем должна была узнать, что результат представляет собой небольшое изменение по сравнению с предыдущим временным шагом.

В анализе временных рядов обычно строят модели, которые вместо предсказания следующего значения предсказывают, как значение изменится на следующем временном шаге. Точно так же остаточные сети — или ResNets — в глубоком обучении относятся к архитектурам, в которых каждый уровень добавляется к накапливающемуся результату модели.

tf.initializers.zeros , чтобы гарантировать, что начальные прогнозируемые изменения будут небольшими и не пересилят остаточное соединение. Здесь нет проблем с нарушением симметрии для градиентов, поскольку zeros используются только на последнем слое.

## Многошаговые модели

Как модели с одним выходом, так и модели с несколькими выходами в предыдущих разделах делали прогнозы с одним временным шагом, на один день вперед.

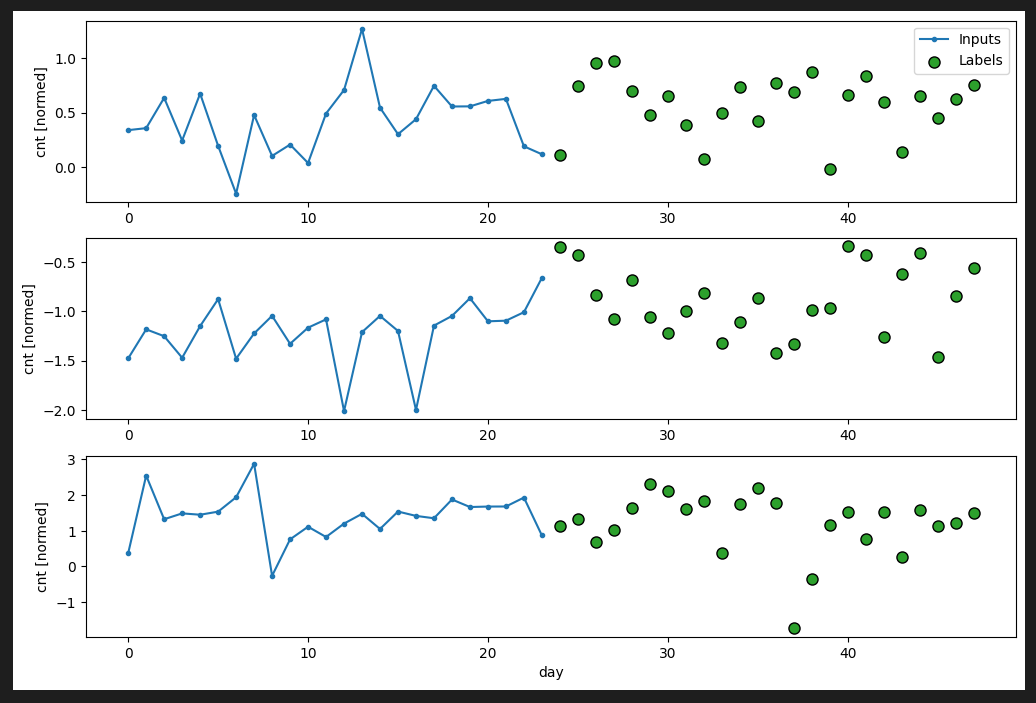
В многоэтапном прогнозировании модель должна научиться прогнозировать диапазон будущих значений. Таким образом, в отличие от одноступенчатой модели, в которой предсказывается только одна точка будущего, многоступенчатая модель предсказывает последовательность будущих значений.

Есть два грубых подхода к этому:

1. Прогнозы одиночного выстрела, когда весь временной ряд прогнозируется сразу.
2. Прогнозы авторегрессии, при которых модель делает только одношаговые прогнозы, а ее выходные данные возвращаются в качестве входных данных.

Все модели будут предсказывать все функции на всех выходных временных шагах.

Для многоступенчатой модели обучающие данные снова состоят из ежедневных выборок. Однако здесь модели научатся предсказывать будущее на 24 дня, учитывая 24 дня прошлого.



### Базовые показатели

#### Базовый(линейный)

Простая базовая линия для этой задачи – повторить последний временной шаг ввода для необходимого количества выходных временных шагов.

#### Базовый(повторение)

Поскольку эта задача состоит в том, чтобы предсказать 24 дня в будущем, учитывая 24 дня в прошлом, другой простой подход состоит в том, чтобы повторить предыдущий день, предполагая, что завтра будет похоже.

## Однозарядные модели

Одним из высокоуровневых подходов к этой проблеме является использование «однократной» модели, в которой модель делает прогноз всей последовательности за один шаг.

Это может быть эффективно реализовано как tf.keras.layers.Dense с выходными единицами OUT\_STEPS\*features . Модель просто должна преобразовать этот вывод в требуемый (OUTPUT\_STEPS, features).

### Линейная модель

Простая линейная модель, основанная на последнем входном временном шаге, работает лучше любой базовой линии, но у нее недостаточно мощности. Модель должна прогнозировать временные шаги OUTPUT\_STEPS на основе одного входного временного шага с линейной проекцией. Он может зафиксировать только низкоразмерный фрагмент поведения, вероятно, основанный в основном на времени суток и времени года.

### Плотная модель

Добавление tf.keras.layers.Dense между входом и выходом делает линейную модель более мощной, но по-прежнему основано только на одном временном шаге ввода. В модели меняется количество нейронов.

### Сверточная

Сверточная модель делает прогнозы на основе истории фиксированной ширины, что может привести к более высокой производительности, чем плотная модель, поскольку она может видеть, как все меняется с течением времени. В модели меняется количество нейронов.

### Реккурентная

Рекуррентная модель может научиться использовать длинную историю входных данных, если она имеет отношение к прогнозам, которые делает модель. Здесь модель будет накапливать внутреннее состояние за 24 дня, прежде чем сделать один прогноз на следующие 24 дня.

В этом однократном формате LSTM нужно выводить только на последнем временном шаге, поэтому установим return\_sequences=False в tf.keras.layers.LSTM.

## Авторегрессионная модель

Все вышеперечисленные модели предсказывают всю выходную последовательность за один шаг.

В некоторых случаях для модели может быть полезно разложить этот прогноз на отдельные временные шаги. Затем выходные данные каждой модели могут быть возвращены в себя на каждом шаге, и прогнозы могут быть сделаны в зависимости от предыдущего, как в классическом Генерация последовательностей с рекуррентными нейронными сетями. Одним из явных преимуществ модели этого типа является то, что ее можно настроить для получения выходных данных различной длины.

Модель будет иметь ту же базовую форму, что и одноэтапные модели LSTM из более ранних версий: слой tf.keras.layers.LSTM , за которым следует слой tf.keras.layers.Dense , который преобразует выходные данные слоя LSTM в прогнозы модели.

tf.keras.layers.LSTM — это tf.keras.layers.LSTMCell , обернутый в tf.keras.layers.RNN более высокого уровня, который управляет для вас результатами состояния и последовательности.

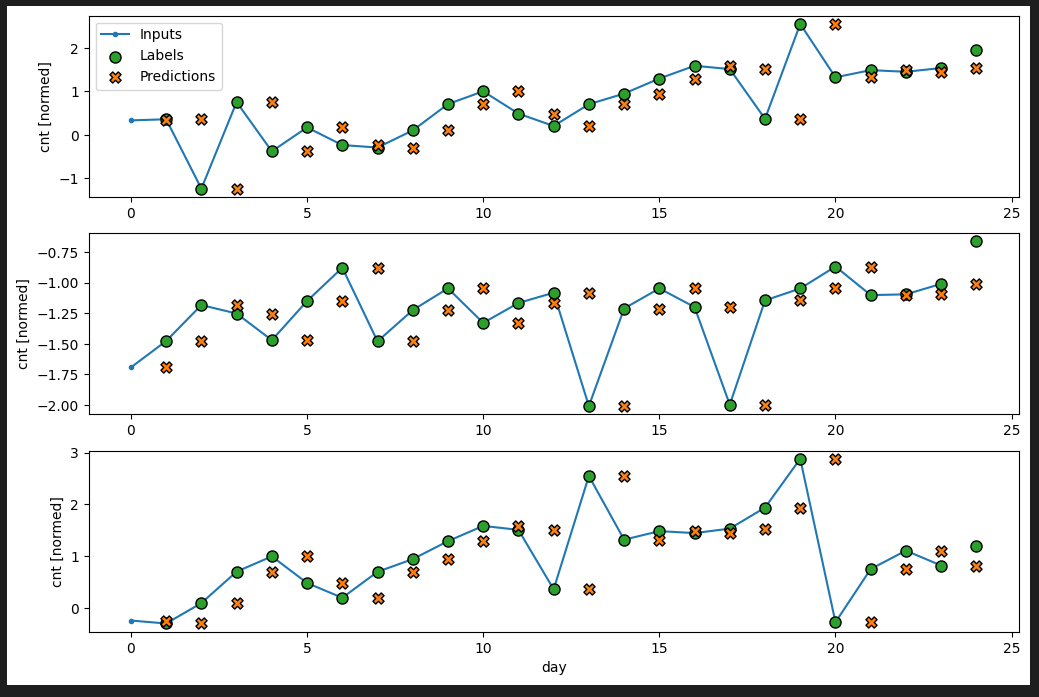
В этом случае модель должна вручную управлять входными данными для каждого шага, поэтому она использует tf.keras.layers.LSTMCell непосредственно для интерфейса нижнего уровня с одним временным шагом.

Первый метод, в котором нуждается эта модель, — это метод warmup для инициализации ее внутреннего состояния на основе входных данных. После обучения это состояние будет захватывать соответствующие части истории ввода. Это эквивалентно одноэтапной модели LSTM из более раннего.

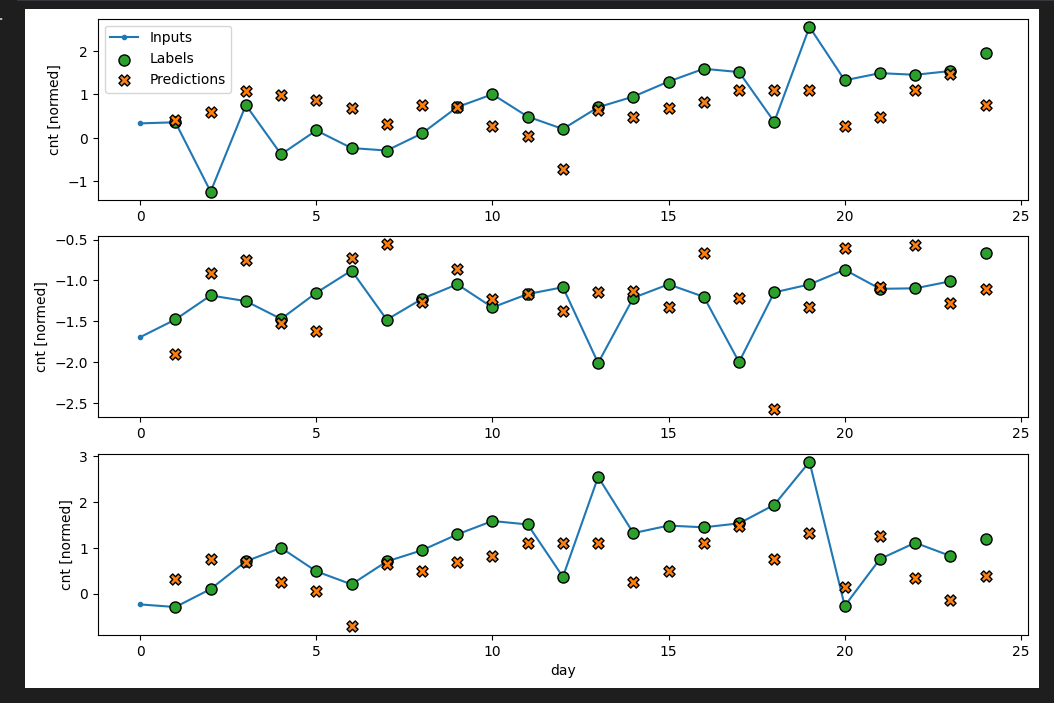
# Вычислительные эксперименты

## Одноступенчатые модели

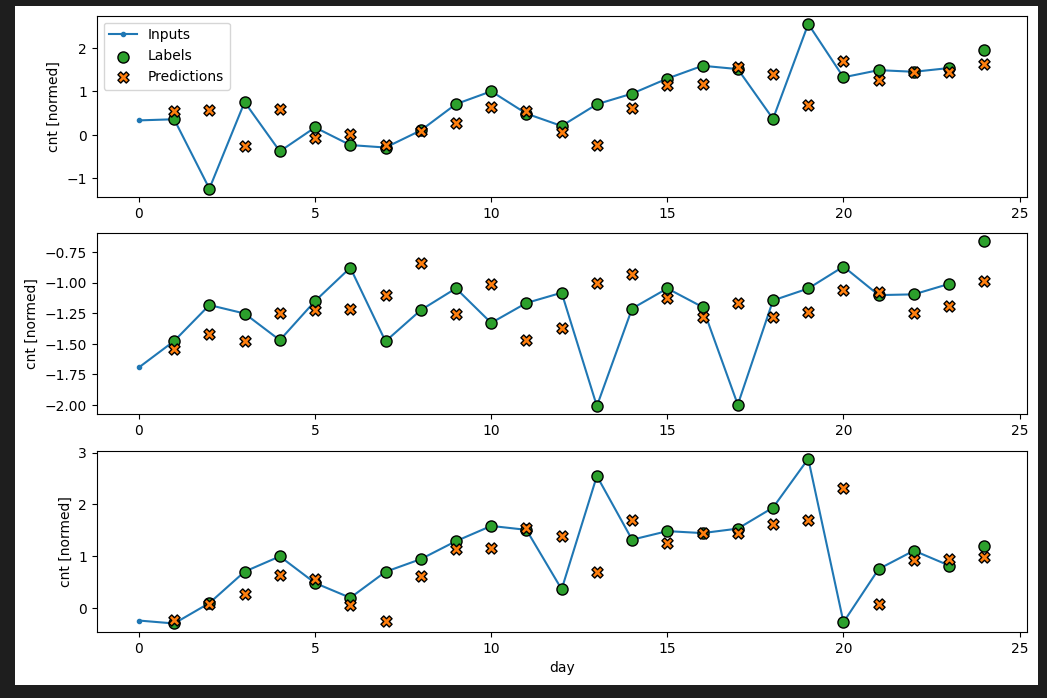
### Базовый уровень



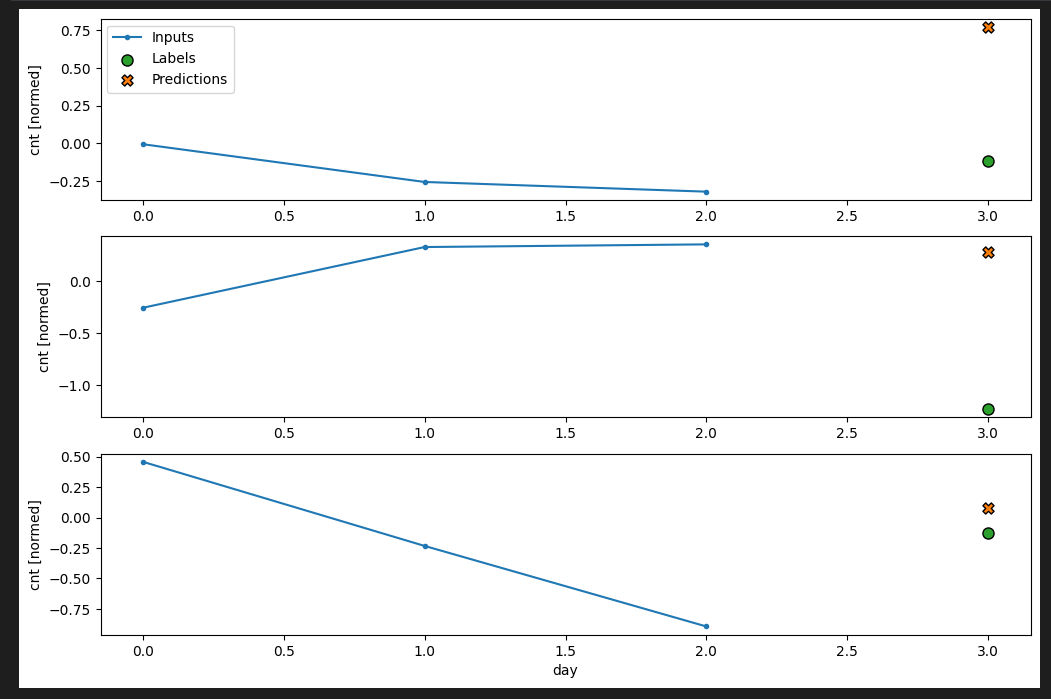
### Линейчатая модель



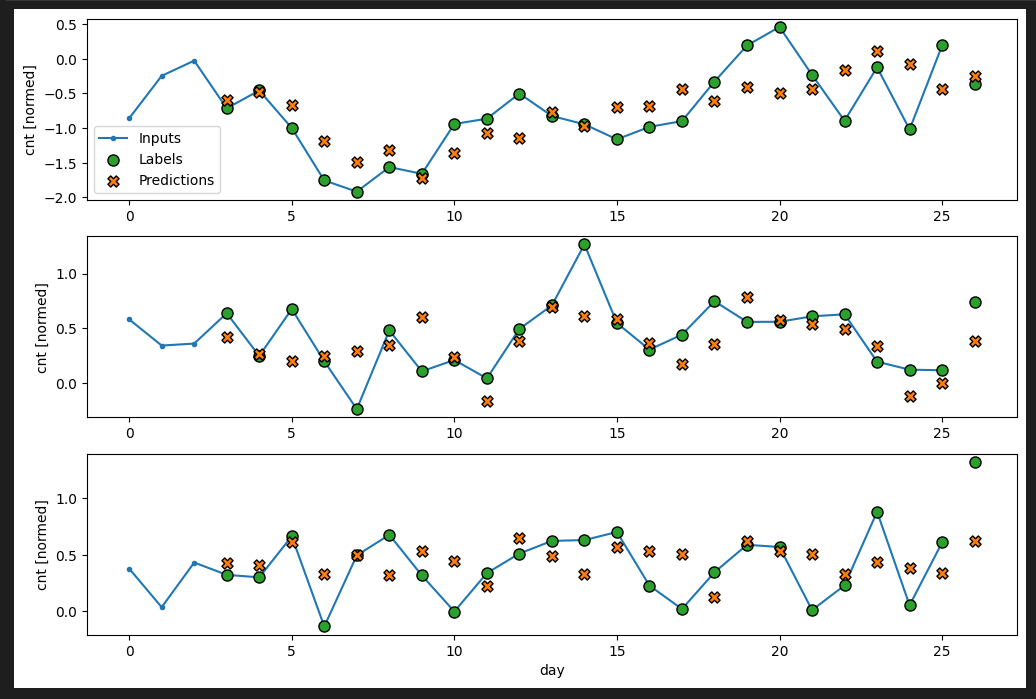
### Плотная модель



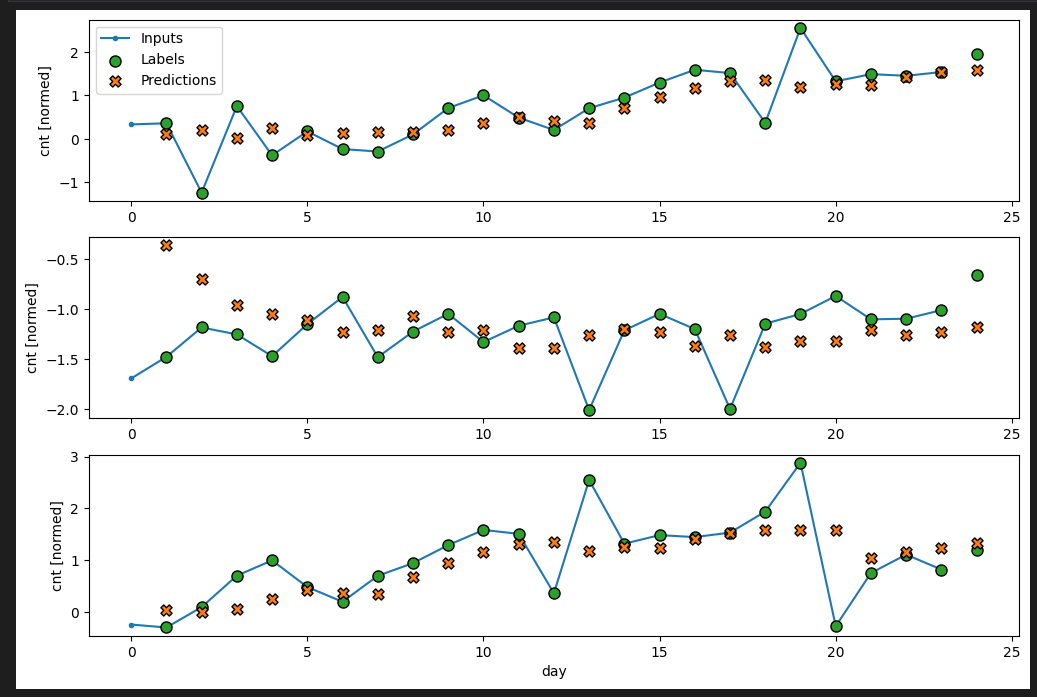
### Многоплотная модель



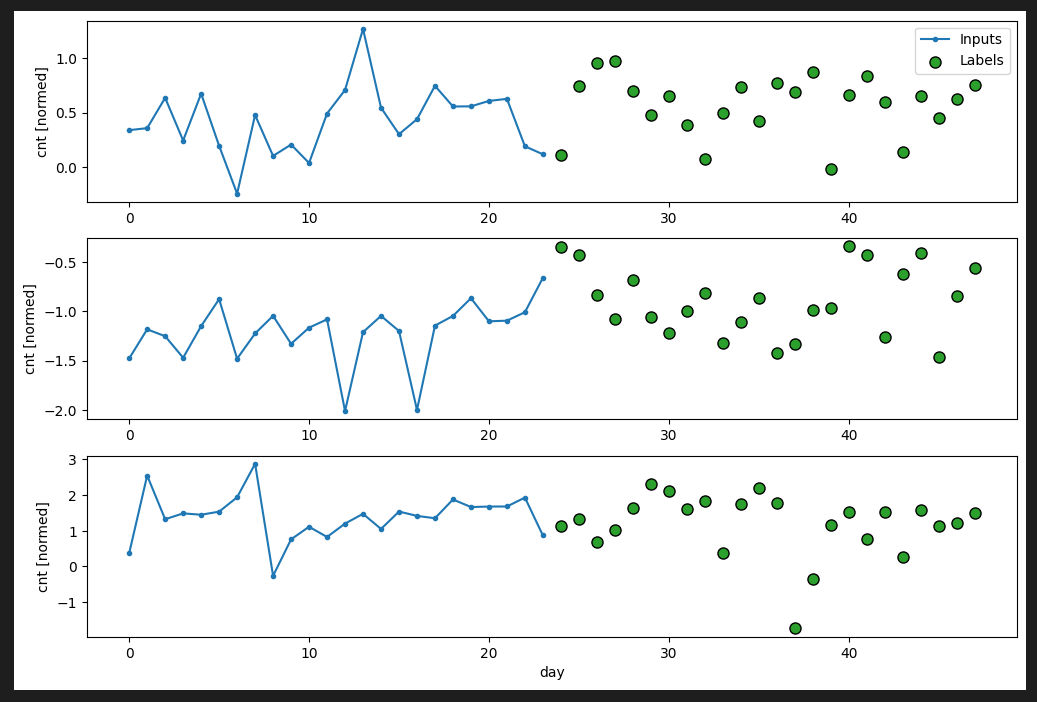
### Сверточная модель



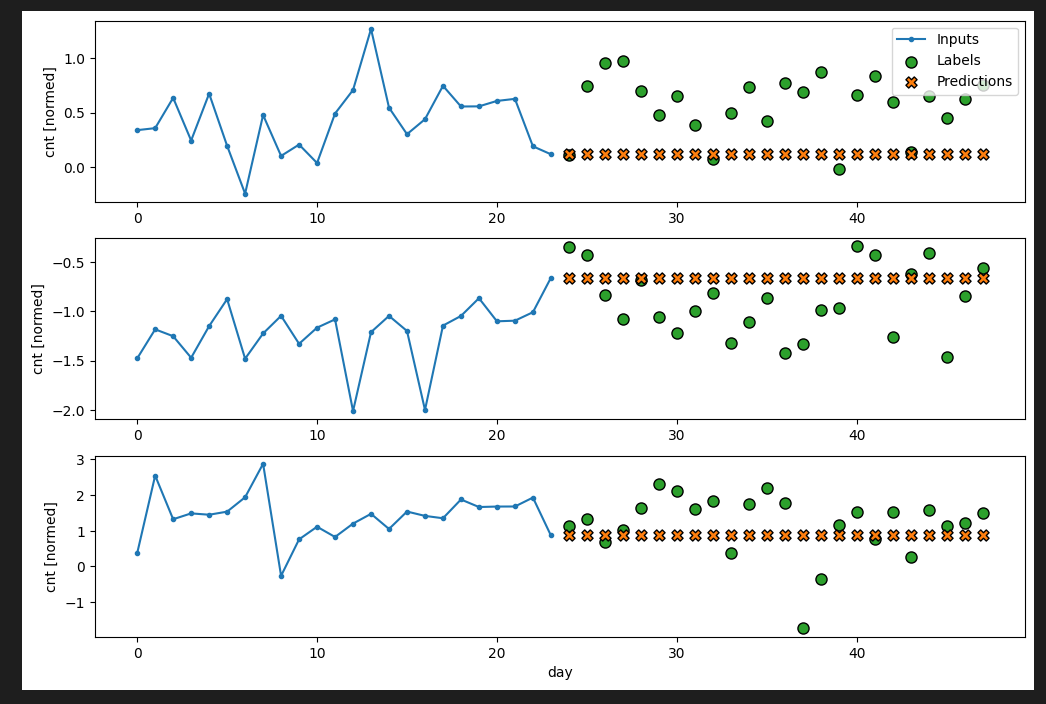
### Реккурентная



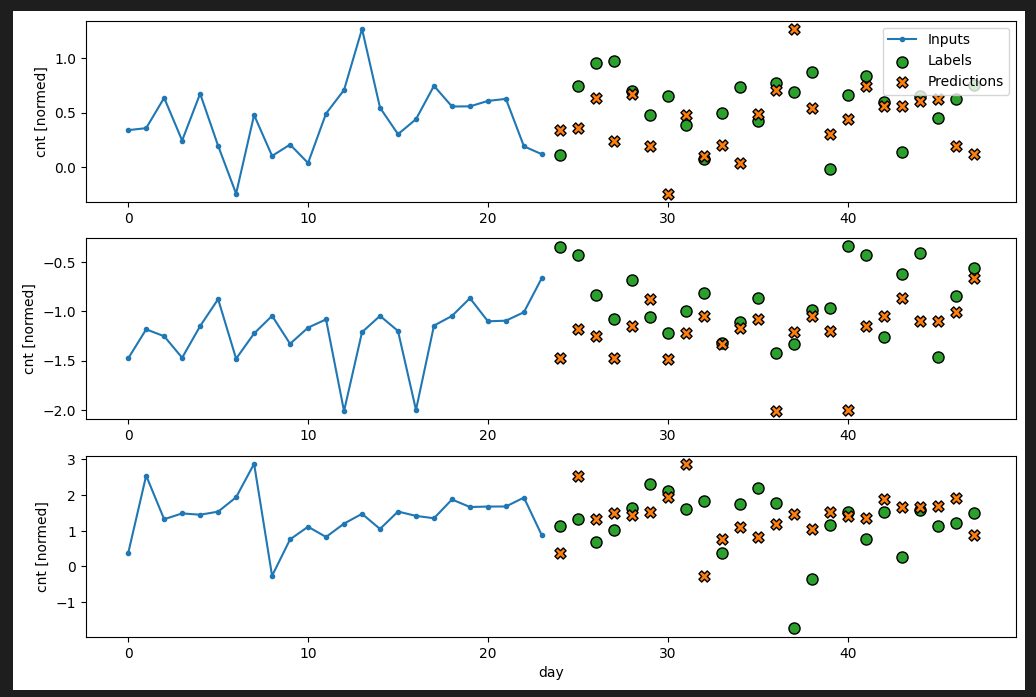
## Многошаговые модели



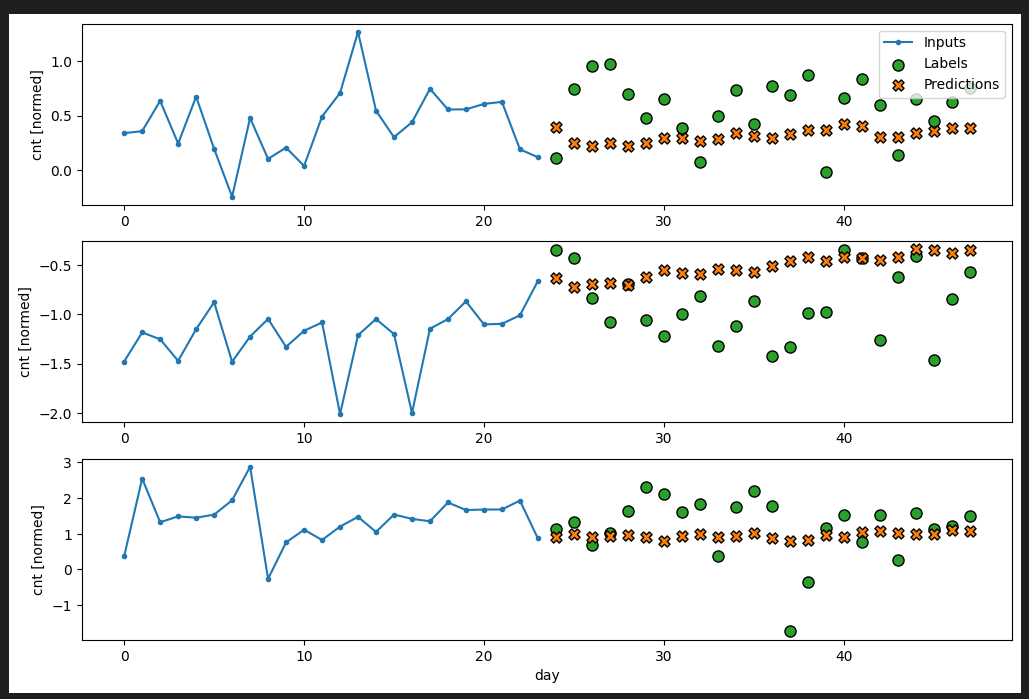
### Базовая(линия)



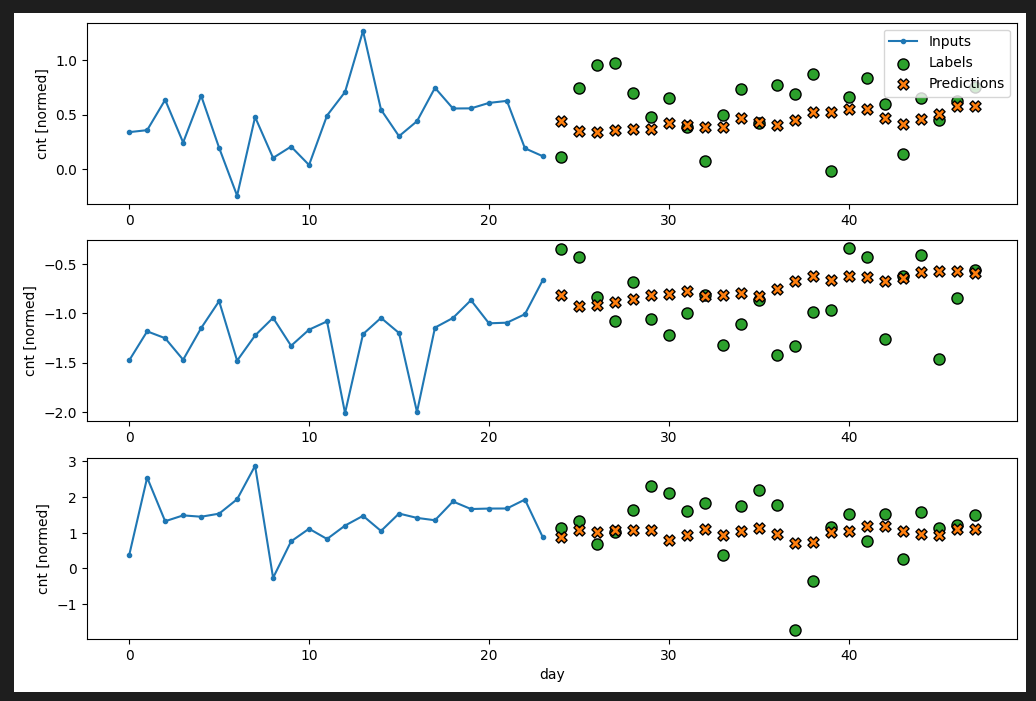
### Базовый(повторение)



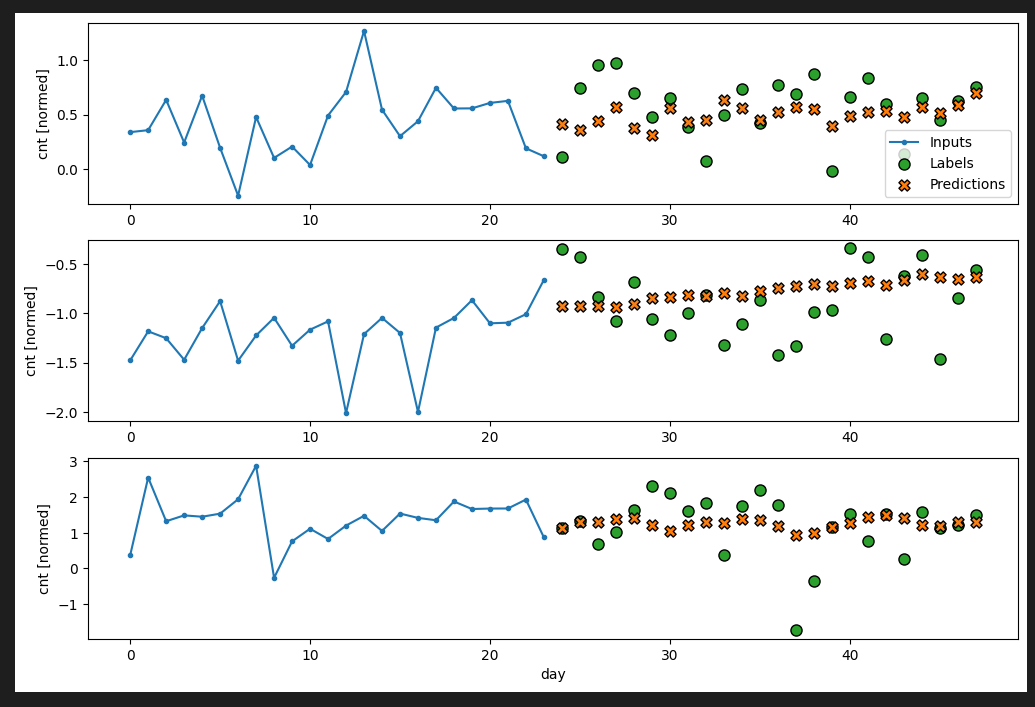
### Линейная



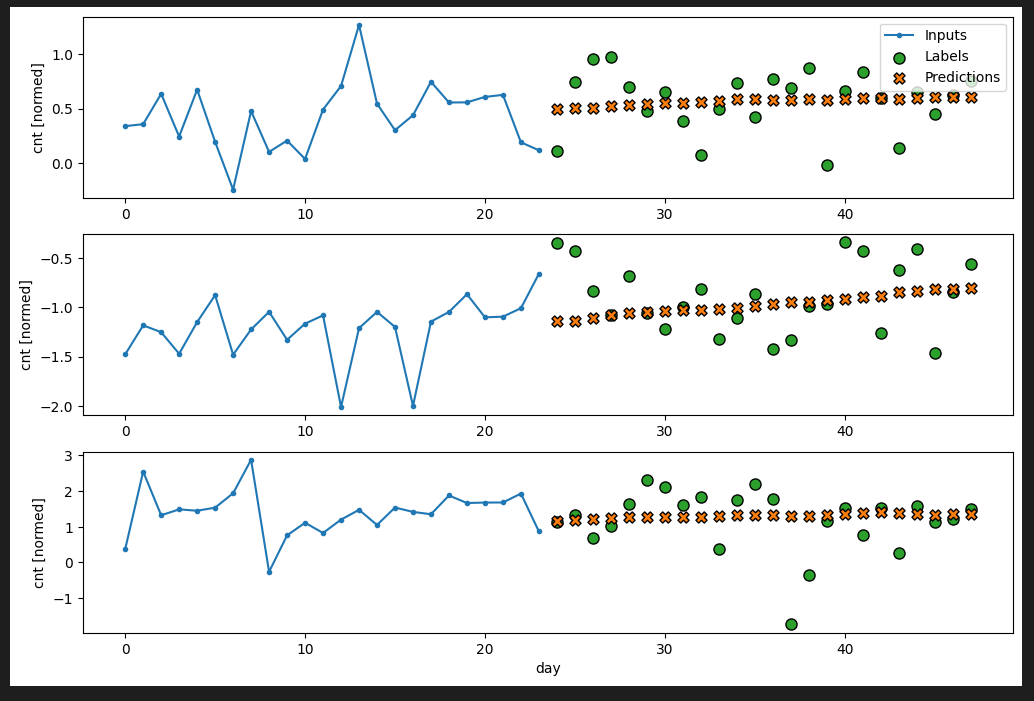
### Плотная



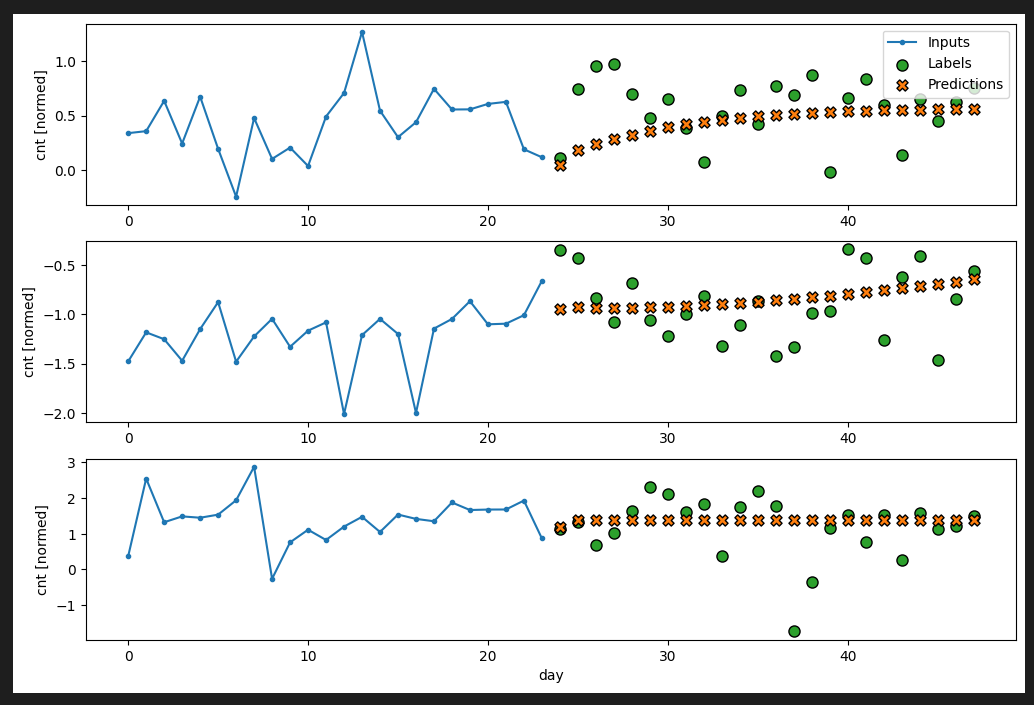
### Сверточная



### Реккурентная



### Авторегрессионная модель

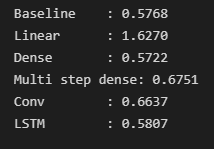


# Результаты работы

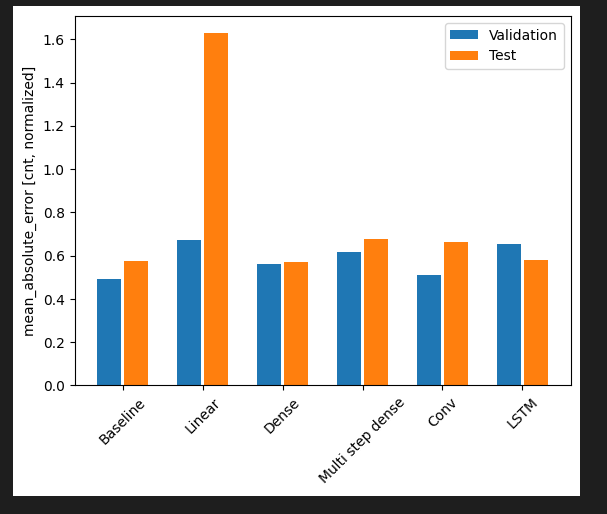
Таким образом были разработаны модели для одношаговых и многошаговых моделей нейронных сетей.

## Для одношаговых

Представим величину ошибки всех моделей на графике и проанализируем их

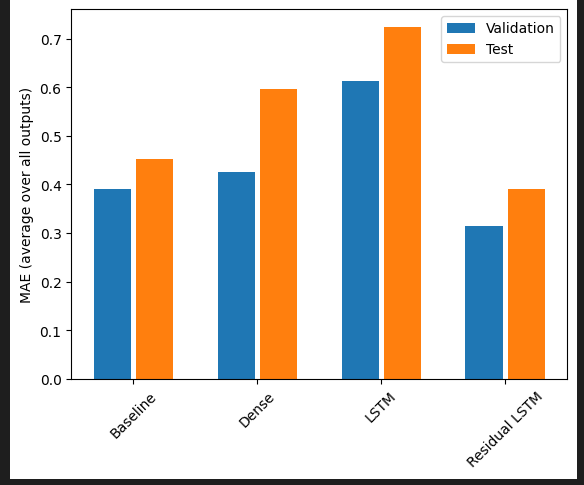


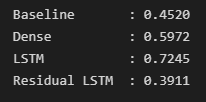
Самой выигрышной является базовая модель. Выигрыш, достигнутый при переходе от плотной модели к многоплотной и сверточной, составляет всего несколько процентов (если вообще есть). LSTM модель показала себя не так плохо, а линейная модель показала себя явно хуже остальных.



Из графика видно, что линейная модель показала себя лучше на проверочной выборке, но очень плохо на тестовом сете. В остальных случаях

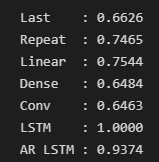
Для моделей с несколькими выходами величина ошибки и график выглядя следующим образом.

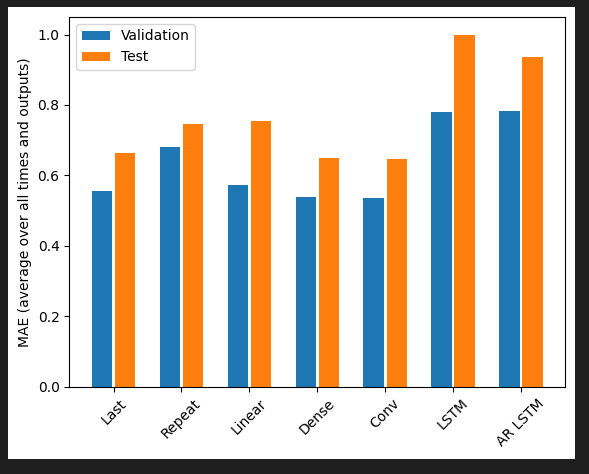




Самой выигрышной является базовая модель и Residual LTSM, а LSTM модель показала себя явно хуже, как Dense модель.

## Для многошаговых.





# Заключение

В ходе выполнения лабораторной работы были изучены основные принципы работы с обучением нейронных сетей. Обучены различные нейронные сети на наборе из 731 данных.

Проведены эксперименты по сравнению различных моделей нейронных сетей.

Явно наблюдается увеличение отдачи в зависимости от сложности модели.

Метрики для моделей с несколькими выходами в первой половине этого руководства показывают среднюю производительность по всем выходным функциям. Эти характеристики аналогичны, но также усреднены по временным шагам вывода.

В результате было выявлено, что одношаговые модели ведут себя гораздо лучше, чем многошаговые модели на данном наборе данных.

# Ссылки

Датасет: <http://archive.ics.uci.edu/ml/datasets/bike+sharing+dataset>

GitHub: <https://github.com/PKkDev/nn_lab2>

# Листинг программы

# Setup

# импортируем всё, что понадобится

import os

import datetime

import IPython

import IPython.display

import matplotlib as mpl

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

import seaborn as sns

import tensorflow as tf

# импортируем датасет

df = pd.read\_csv('day.csv')

#df.head()

#df = df.loc[:,['dteday','workingday','weathersit','temp','windspeed','cnt']]

df.info()

df.head()

df = df.set\_index('dteday')

df.index = pd.to\_datetime(df.index,unit='ns')

print(df.index)

# temp vs windspeed #

fig, ax1 = plt.subplots(figsize = (12, 8))

ax1.plot(df.index, df['temp'], 'r-', label='temp')

ax1.plot(df.index, df['windspeed'], 'b-', label='windspeed')

ax1.set\_xlabel('Date')

plt.show()

# cnt #

fig, ax1 = plt.subplots(figsize = (12, 8))

ax1.plot(df.index, df['cnt'], 'r-', label='cnt')

ax1.set\_xlabel('Date')

plt.show()

# статистика набора данных

df.describe().transpose()

## Split the data

column\_indices = {name: i for i, name in enumerate(df.columns)}

n = len(df)

train\_df = df[0:int(n\*0.7)]

val\_df = df[int(n\*0.7):int(n\*0.9)]

test\_df = df[int(n\*0.9):]

num\_features = df.shape[1]

## Normalize the data

train\_mean = train\_df.mean()

train\_std = train\_df.std()

train\_df = (train\_df - train\_mean) / train\_std

val\_df = (val\_df - train\_mean) / train\_std

test\_df = (test\_df - train\_mean) / train\_std

df\_std = (df - train\_mean) / train\_std

df\_std = df\_std.melt(var\_name='Column', value\_name='Normalized')

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

ax = sns.violinplot(x='Column', y='Normalized', data=df\_std)

\_ = ax.set\_xticklabels(df.keys(), rotation=90)

# Data windowing

class WindowGenerator():

def \_\_init\_\_(self, input\_width, label\_width, shift,

train\_df=train\_df, val\_df=val\_df, test\_df=test\_df,

label\_columns=None):

# Store the raw data.

self.train\_df = train\_df

self.val\_df = val\_df

self.test\_df = test\_df

# Work out the label column indices.

self.label\_columns = label\_columns

if label\_columns is not None:

self.label\_columns\_indices = {name: i for i, name in

enumerate(label\_columns)}

self.column\_indices = {name: i for i, name in

enumerate(train\_df.columns)}

# Work out the window parameters.

self.input\_width = input\_width

self.label\_width = label\_width

self.shift = shift

self.total\_window\_size = input\_width + shift

self.input\_slice = slice(0, input\_width)

self.input\_indices = np.arange(self.total\_window\_size)[self.input\_slice]

self.label\_start = self.total\_window\_size - self.label\_width

self.labels\_slice = slice(self.label\_start, None)

self.label\_indices = np.arange(self.total\_window\_size)[self.labels\_slice]

def \_\_repr\_\_(self):

return '\n'.join([

f'Total window size: {self.total\_window\_size}',

f'Input indices: {self.input\_indices}',

f'Label indices: {self.label\_indices}',

f'Label column name(s): {self.label\_columns}'])

w1 = WindowGenerator(input\_width=24, label\_width=1, shift=24,

label\_columns=['cnt'])

w1

w2 = WindowGenerator(input\_width=6, label\_width=1, shift=1,

label\_columns=['cnt'])

w2

## Split

def split\_window(self, features):

inputs = features[:, self.input\_slice, :]

labels = features[:, self.labels\_slice, :]

if self.label\_columns is not None:

labels = tf.stack(

[labels[:, :, self.column\_indices[name]] for name in self.label\_columns],

axis=-1)

# Slicing doesn't preserve static shape information, so set the shapes

# manually. This way the `tf.data.Datasets` are easier to inspect.

inputs.set\_shape([None, self.input\_width, None])

labels.set\_shape([None, self.label\_width, None])

return inputs, labels

WindowGenerator.split\_window = split\_window

# Стек три куска, длина всего окна.

example\_window = tf.stack([np.array(train\_df[:w2.total\_window\_size]),

np.array(train\_df[100:100+w2.total\_window\_size]),

np.array(train\_df[200:200+w2.total\_window\_size])])

example\_inputs, example\_labels = w2.split\_window(example\_window)

print('All shapes are: (batch, time, features)')

print(f'Window shape: {example\_window.shape}')

print(f'Inputs shape: {example\_inputs.shape}')

print(f'Labels shape: {example\_labels.shape}')

## Plot

w2.example = example\_inputs, example\_labels

def plot(self, model=None, plot\_col='cnt', max\_subplots=3):

inputs, labels = self.example

plt.figure(figsize=(12, 8))

plot\_col\_index = self.column\_indices[plot\_col]

max\_n = min(max\_subplots, len(inputs))

for n in range(max\_n):

plt.subplot(max\_n, 1, n+1)

plt.ylabel(f'{plot\_col} [normed]')

plt.plot(self.input\_indices, inputs[n, :, plot\_col\_index],

label='Inputs', marker='.', zorder=-10)

if self.label\_columns:

label\_col\_index = self.label\_columns\_indices.get(plot\_col, None)

else:

label\_col\_index = plot\_col\_index

if label\_col\_index is None:

continue

plt.scatter(self.label\_indices, labels[n, :, label\_col\_index],

edgecolors='k', label='Labels', c='#2ca02c', s=64)

if model is not None:

predictions = model(inputs)

plt.scatter(self.label\_indices, predictions[n, :, label\_col\_index],

marker='X', edgecolors='k', label='Predictions',

c='#ff7f0e', s=64)

if n == 0:

plt.legend()

plt.xlabel('day')

WindowGenerator.plot = plot

w2.plot()

# пример для другого столбца

w2.plot(plot\_col='hum')

## Create tf.data.Datasets

def make\_dataset(self, data):

data = np.array(data, dtype=np.float32)

ds = tf.keras.utils.timeseries\_dataset\_from\_array(

data=data,

targets=None,

sequence\_length=self.total\_window\_size,

sequence\_stride=1,

shuffle=True,

batch\_size=32,)

ds = ds.map(self.split\_window)

return ds

WindowGenerator.make\_dataset = make\_dataset

# свойство для доступа к обучающим данным

@property

def train(self):

return self.make\_dataset(self.train\_df)

# свойство для доступа к проверочным данным

@property

def val(self):

return self.make\_dataset(self.val\_df)

# свойство для доступа к тестовым данным

@property

def test(self):

return self.make\_dataset(self.test\_df)

# стандартный пакет примеров для быстрого доступа и построения графиков

@property

def example(self):

"""Get and cache an example batch of `inputs, labels` for plotting."""

result = getattr(self, '\_example', None)

if result is None:

# No example batch was found, so get one from the `.train` dataset

result = next(iter(self.train))

# And cache it for next time

self.\_example = result

return result

WindowGenerator.train = train

WindowGenerator.val = val

WindowGenerator.test = test

WindowGenerator.example = example

# Каждый элемент представляет собой пару (входы, метка).

w2.train.element\_spec

# Итерация по Dataset дает конкретные партии

for example\_inputs, example\_labels in w2.train.take(1):

print(f'Inputs shape (batch, time, features): {example\_inputs.shape}')

print(f'Labels shape (batch, time, features): {example\_labels.shape}')

# Single step models

single\_step\_window = WindowGenerator(

input\_width=1, label\_width=1, shift=1,

label\_columns=['cnt'])

single\_step\_window

for example\_inputs, example\_labels in single\_step\_window.train.take(1):

print(f'Inputs shape (batch, time, features): {example\_inputs.shape}')

print(f'Labels shape (batch, time, features): {example\_labels.shape}')

## Baseline

class Baseline(tf.keras.Model):

def \_\_init\_\_(self, label\_index=None):

super().\_\_init\_\_()

self.label\_index = label\_index

def call(self, inputs):

if self.label\_index is None:

return inputs

result = inputs[:, :, self.label\_index]

return result[:, :, tf.newaxis]

baseline = Baseline(label\_index=column\_indices['cnt'])

baseline.compile(loss=tf.losses.MeanSquaredError(),

metrics=[tf.metrics.MeanAbsoluteError()])

val\_performance = {}

performance = {}

val\_performance['Baseline'] = baseline.evaluate(single\_step\_window.val)

performance['Baseline'] = baseline.evaluate(single\_step\_window.test, verbose=0)

# генерирует окно в 24 дня последовательных входных данных и меток за раз

wide\_window = WindowGenerator(

input\_width=24, label\_width=24, shift=1,

label\_columns=['cnt'])

wide\_window

print('Input shape:', wide\_window.example[0].shape)

print('Output shape:', baseline(wide\_window.example[0]).shape)

wide\_window.plot(baseline)

## Linear model

linear = tf.keras.Sequential([

tf.keras.layers.Dense(units=1)

])

print('Input shape:', single\_step\_window.example[0].shape)

print('Output shape:', linear(single\_step\_window.example[0]).shape)

# обучается множество моделей, поэтому упакуйте процедуру обучения в функцию

MAX\_EPOCHS = 20

def compile\_and\_fit(model, window, patience=2):

early\_stopping = tf.keras.callbacks.EarlyStopping(monitor='val\_loss',

patience=patience,

mode='min')

model.compile(loss=tf.losses.MeanSquaredError(),

optimizer=tf.optimizers.Adam(),

metrics=[tf.metrics.MeanAbsoluteError()])

history = model.fit(window.train, epochs=MAX\_EPOCHS,

validation\_data=window.val,

callbacks=[early\_stopping])

return history

# Обучение модели и оценка ее производительность

history = compile\_and\_fit(linear, single\_step\_window)

val\_performance['Linear'] = linear.evaluate(single\_step\_window.val)

performance['Linear'] = linear.evaluate(single\_step\_window.test, verbose=0)

print('Input shape:', wide\_window.example[0].shape)

print('Output shape:', baseline(wide\_window.example[0]).shape)

wide\_window.plot(linear)

plt.bar(x = range(len(train\_df.columns)),

height=linear.layers[0].kernel[:,0].numpy())

axis = plt.gca()

axis.set\_xticks(range(len(train\_df.columns)))

\_ = axis.set\_xticklabels(train\_df.columns, rotation=90)

## Dense

dense = tf.keras.Sequential([

tf.keras.layers.Dense(units=64, activation='relu'),

tf.keras.layers.Dense(units=64, activation='relu'),

tf.keras.layers.Dense(units=1)

])

history = compile\_and\_fit(dense, single\_step\_window)

val\_performance['Dense'] = dense.evaluate(single\_step\_window.val)

performance['Dense'] = dense.evaluate(single\_step\_window.test, verbose=0)

print('Input shape:', wide\_window.example[0].shape)

print('Output shape:', dense(wide\_window.example[0]).shape)

wide\_window.plot(dense)

## Multi-step dense

CONV\_WIDTH = 3

conv\_window = WindowGenerator(

input\_width=CONV\_WIDTH,

label\_width=1,

shift=1,

label\_columns=['cnt'])

conv\_window

conv\_window.plot()

plt.title("Given 3 hours of inputs, predict 1 hour into the future.")

multi\_step\_dense = tf.keras.Sequential([

# Shape: (time, features) => (time\*features)

tf.keras.layers.Flatten(),

tf.keras.layers.Dense(units=32, activation='relu'),

tf.keras.layers.Dense(units=32, activation='relu'),

tf.keras.layers.Dense(units=1),

# Add back the time dimension.

# Shape: (outputs) => (1, outputs)

tf.keras.layers.Reshape([1, -1]),

])

print('Input shape:', conv\_window.example[0].shape)

print('Output shape:', multi\_step\_dense(conv\_window.example[0]).shape)

history = compile\_and\_fit(multi\_step\_dense, conv\_window)

IPython.display.clear\_output()

val\_performance['Multi step dense'] = multi\_step\_dense.evaluate(conv\_window.val)

performance['Multi step dense'] = multi\_step\_dense.evaluate(conv\_window.test, verbose=0)

conv\_window.plot(multi\_step\_dense)

## Convolution neural network (CNN)

conv\_model = tf.keras.Sequential([

tf.keras.layers.Conv1D(filters=32,

kernel\_size=(CONV\_WIDTH,),

activation='relu'),

tf.keras.layers.Dense(units=32, activation='relu'),

tf.keras.layers.Dense(units=1),

])

print("Conv model on `conv\_window`")

print('Input shape:', conv\_window.example[0].shape)

print('Output shape:', conv\_model(conv\_window.example[0]).shape)

history = compile\_and\_fit(conv\_model, conv\_window)

IPython.display.clear\_output()

val\_performance['Conv'] = conv\_model.evaluate(conv\_window.val)

performance['Conv'] = conv\_model.evaluate(conv\_window.test, verbose=0)

# вывод короче ввода

print("Wide window")

print('Input shape:', wide\_window.example[0].shape)

print('Labels shape:', wide\_window.example[1].shape)

print('Output shape:', conv\_model(wide\_window.example[0]).shape)

LABEL\_WIDTH = 24

INPUT\_WIDTH = LABEL\_WIDTH + (CONV\_WIDTH - 1)

wide\_conv\_window = WindowGenerator(

input\_width=INPUT\_WIDTH,

label\_width=LABEL\_WIDTH,

shift=1,

label\_columns=['cnt'])

wide\_conv\_window

print("Wide conv window")

print('Input shape:', wide\_conv\_window.example[0].shape)

print('Labels shape:', wide\_conv\_window.example[1].shape)

print('Output shape:', conv\_model(wide\_conv\_window.example[0]).shape)

wide\_conv\_window.plot(conv\_model)

## Recurrent neural network (RNN)

lstm\_model = tf.keras.models.Sequential([

# Shape [batch, time, features] => [batch, time, lstm\_units]

tf.keras.layers.LSTM(32, return\_sequences=True),

# Shape => [batch, time, features]

tf.keras.layers.Dense(units=1)

])

print('Input shape:', wide\_window.example[0].shape)

print('Output shape:', lstm\_model(wide\_window.example[0]).shape)

history = compile\_and\_fit(lstm\_model, wide\_window)

IPython.display.clear\_output()

val\_performance['LSTM'] = lstm\_model.evaluate(wide\_window.val)

performance['LSTM'] = lstm\_model.evaluate(wide\_window.test, verbose=0)

wide\_window.plot(lstm\_model)

## Performance

x = np.arange(len(performance))

width = 0.3

metric\_name = 'mean\_absolute\_error'

metric\_index = lstm\_model.metrics\_names.index('mean\_absolute\_error')

val\_mae = [v[metric\_index] for v in val\_performance.values()]

test\_mae = [v[metric\_index] for v in performance.values()]

plt.ylabel('mean\_absolute\_error [cnt, normalized]')

plt.bar(x - 0.17, val\_mae, width, label='Validation')

plt.bar(x + 0.17, test\_mae, width, label='Test')

plt.xticks(ticks=x, labels=performance.keys(),

rotation=45)

\_ = plt.legend()

for name, value in performance.items():

print(f'{name:12s}: {value[1]:0.4f}')

# Multi-output models

single\_step\_window = WindowGenerator(

# `WindowGenerator` returns all features as labels if you

# don't set the `label\_columns` argument.

input\_width=1, label\_width=1, shift=1)

wide\_window = WindowGenerator(

input\_width=24, label\_width=24, shift=1)

# ось features меток теперь имеет ту же глубину, что и входные данные, а не 1

for example\_inputs, example\_labels in wide\_window.train.take(1):

print(f'Inputs shape (batch, time, features): {example\_inputs.shape}')

print(f'Labels shape (batch, time, features): {example\_labels.shape}')

### Baseline

baseline = Baseline()

baseline.compile(loss=tf.losses.MeanSquaredError(),

metrics=[tf.metrics.MeanAbsoluteError()])

val\_performance = {}

performance = {}

val\_performance['Baseline'] = baseline.evaluate(wide\_window.val)

performance['Baseline'] = baseline.evaluate(wide\_window.test, verbose=0)

### Dense

dense = tf.keras.Sequential([

tf.keras.layers.Dense(units=64, activation='relu'),

tf.keras.layers.Dense(units=64, activation='relu'),

tf.keras.layers.Dense(units=num\_features)

])

history = compile\_and\_fit(dense, single\_step\_window)

IPython.display.clear\_output()

val\_performance['Dense'] = dense.evaluate(single\_step\_window.val)

performance['Dense'] = dense.evaluate(single\_step\_window.test, verbose=0)

### RNN

%%time

wide\_window = WindowGenerator(

input\_width=24, label\_width=24, shift=1)

lstm\_model = tf.keras.models.Sequential([

# Shape [batch, time, features] => [batch, time, lstm\_units]

tf.keras.layers.LSTM(32, return\_sequences=True),

# Shape => [batch, time, features]

tf.keras.layers.Dense(units=num\_features)

])

history = compile\_and\_fit(lstm\_model, wide\_window)

IPython.display.clear\_output()

val\_performance['LSTM'] = lstm\_model.evaluate( wide\_window.val)

performance['LSTM'] = lstm\_model.evaluate( wide\_window.test, verbose=0)

print()

### Advanced: Residual connections

class ResidualWrapper(tf.keras.Model):

def \_\_init\_\_(self, model):

super().\_\_init\_\_()

self.model = model

def call(self, inputs, \*args, \*\*kwargs):

delta = self.model(inputs, \*args, \*\*kwargs)

# The prediction for each time step is the input

# from the previous time step plus the delta

# calculated by the model.

return inputs + delta

%%time

residual\_lstm = ResidualWrapper(

tf.keras.Sequential([

tf.keras.layers.LSTM(32, return\_sequences=True),

tf.keras.layers.Dense(

num\_features,

# The predicted deltas should start small.

# Therefore, initialize the output layer with zeros.

kernel\_initializer=tf.initializers.zeros())

]))

history = compile\_and\_fit(residual\_lstm, wide\_window)

IPython.display.clear\_output()

val\_performance['Residual LSTM'] = residual\_lstm.evaluate(wide\_window.val)

performance['Residual LSTM'] = residual\_lstm.evaluate(wide\_window.test, verbose=0)

print()

### Performance-2

x = np.arange(len(performance))

width = 0.3

metric\_name = 'mean\_absolute\_error'

metric\_index = lstm\_model.metrics\_names.index('mean\_absolute\_error')

val\_mae = [v[metric\_index] for v in val\_performance.values()]

test\_mae = [v[metric\_index] for v in performance.values()]

plt.bar(x - 0.17, val\_mae, width, label='Validation')

plt.bar(x + 0.17, test\_mae, width, label='Test')

plt.xticks(ticks=x, labels=performance.keys(),

rotation=45)

plt.ylabel('MAE (average over all outputs)')

\_ = plt.legend()

for name, value in performance.items():

print(f'{name:15s}: {value[1]:0.4f}')

# Multi-step models

OUT\_STEPS = 24

multi\_window = WindowGenerator(input\_width=24,

label\_width=OUT\_STEPS,

shift=OUT\_STEPS)

multi\_window.plot()

multi\_window

## Baselines

class MultiStepLastBaseline(tf.keras.Model):

def call(self, inputs):

return tf.tile(inputs[:, -1:, :], [1, OUT\_STEPS, 1])

last\_baseline = MultiStepLastBaseline()

last\_baseline.compile(loss=tf.losses.MeanSquaredError(),

metrics=[tf.metrics.MeanAbsoluteError()])

multi\_val\_performance = {}

multi\_performance = {}

multi\_val\_performance['Last'] = last\_baseline.evaluate(multi\_window.val)

multi\_performance['Last'] = last\_baseline.evaluate(multi\_window.test, verbose=0)

multi\_window.plot(last\_baseline)

class RepeatBaseline(tf.keras.Model):

def call(self, inputs):

return inputs

repeat\_baseline = RepeatBaseline()

repeat\_baseline.compile(loss=tf.losses.MeanSquaredError(),

metrics=[tf.metrics.MeanAbsoluteError()])

multi\_val\_performance['Repeat'] = repeat\_baseline.evaluate(multi\_window.val)

multi\_performance['Repeat'] = repeat\_baseline.evaluate(multi\_window.test, verbose=0)

multi\_window.plot(repeat\_baseline)

## Single-shot models

### Linear

multi\_linear\_model = tf.keras.Sequential([

# Take the last time-step.

# Shape [batch, time, features] => [batch, 1, features]

tf.keras.layers.Lambda(lambda x: x[:, -1:, :]),

# Shape => [batch, 1, out\_steps\*features]

tf.keras.layers.Dense(OUT\_STEPS\*num\_features,

kernel\_initializer=tf.initializers.zeros()),

# Shape => [batch, out\_steps, features]

tf.keras.layers.Reshape([OUT\_STEPS, num\_features])

])

history = compile\_and\_fit(multi\_linear\_model, multi\_window)

IPython.display.clear\_output()

multi\_val\_performance['Linear'] = multi\_linear\_model.evaluate(multi\_window.val)

multi\_performance['Linear'] = multi\_linear\_model.evaluate(multi\_window.test, verbose=0)

multi\_window.plot(multi\_linear\_model)

### Dense-2

multi\_dense\_model = tf.keras.Sequential([

# Take the last time step.

# Shape [batch, time, features] => [batch, 1, features]

tf.keras.layers.Lambda(lambda x: x[:, -1:, :]),

# Shape => [batch, 1, dense\_units]

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

# Shape => [batch, out\_steps\*features]

tf.keras.layers.Dense(OUT\_STEPS\*num\_features,

kernel\_initializer=tf.initializers.zeros()),

# Shape => [batch, out\_steps, features]

tf.keras.layers.Reshape([OUT\_STEPS, num\_features])

])

history = compile\_and\_fit(multi\_dense\_model, multi\_window)

IPython.display.clear\_output()

multi\_val\_performance['Dense'] = multi\_dense\_model.evaluate(multi\_window.val)

multi\_performance['Dense'] = multi\_dense\_model.evaluate(multi\_window.test, verbose=0)

multi\_window.plot(multi\_dense\_model)

### CNN

CONV\_WIDTH = 3

multi\_conv\_model = tf.keras.Sequential([

# Shape [batch, time, features] => [batch, CONV\_WIDTH, features]

tf.keras.layers.Lambda(lambda x: x[:, -CONV\_WIDTH:, :]),

# Shape => [batch, 1, conv\_units]

tf.keras.layers.Conv1D(256, activation='relu', kernel\_size=(CONV\_WIDTH)),

# Shape => [batch, 1, out\_steps\*features]

tf.keras.layers.Dense(OUT\_STEPS\*num\_features,

kernel\_initializer=tf.initializers.zeros()),

# Shape => [batch, out\_steps, features]

tf.keras.layers.Reshape([OUT\_STEPS, num\_features])

])

history = compile\_and\_fit(multi\_conv\_model, multi\_window)

IPython.display.clear\_output()

multi\_val\_performance['Conv'] = multi\_conv\_model.evaluate(multi\_window.val)

multi\_performance['Conv'] = multi\_conv\_model.evaluate(multi\_window.test, verbose=0)

multi\_window.plot(multi\_conv\_model)

### RNN-2

multi\_lstm\_model = tf.keras.Sequential([

# Shape [batch, time, features] => [batch, lstm\_units].

# Adding more `lstm\_units` just overfits more quickly.

tf.keras.layers.LSTM(32, return\_sequences=False),

# Shape => [batch, out\_steps\*features].

tf.keras.layers.Dense(OUT\_STEPS\*num\_features,

kernel\_initializer=tf.initializers.zeros()),

# Shape => [batch, out\_steps, features].

tf.keras.layers.Reshape([OUT\_STEPS, num\_features])

])

history = compile\_and\_fit(multi\_lstm\_model, multi\_window)

IPython.display.clear\_output()

multi\_val\_performance['LSTM'] = multi\_lstm\_model.evaluate(multi\_window.val)

multi\_performance['LSTM'] = multi\_lstm\_model.evaluate(multi\_window.test, verbose=0)

multi\_window.plot(multi\_lstm\_model)

## Advanced: Autoregressive model

### RNN-3

class FeedBack(tf.keras.Model):

def \_\_init\_\_(self, units, out\_steps):

super().\_\_init\_\_()

self.out\_steps = out\_steps

self.units = units

self.lstm\_cell = tf.keras.layers.LSTMCell(units)

# Also wrap the LSTMCell in an RNN to simplify the `warmup` method.

self.lstm\_rnn = tf.keras.layers.RNN(self.lstm\_cell, return\_state=True)

self.dense = tf.keras.layers.Dense(num\_features)

feedback\_model = FeedBack(units=32, out\_steps=OUT\_STEPS)

def warmup(self, inputs):

# inputs.shape => (batch, time, features)

# x.shape => (batch, lstm\_units)

x, \*state = self.lstm\_rnn(inputs)

# predictions.shape => (batch, features)

prediction = self.dense(x)

return prediction, state

FeedBack.warmup = warmup

prediction, state = feedback\_model.warmup(multi\_window.example[0])

prediction.shape

def call(self, inputs, training=None):

# Use a TensorArray to capture dynamically unrolled outputs.

predictions = []

# Initialize the LSTM state.

prediction, state = self.warmup(inputs)

# Insert the first prediction.

predictions.append(prediction)

# Run the rest of the prediction steps.

for n in range(1, self.out\_steps):

# Use the last prediction as input.

x = prediction

# Execute one lstm step.

x, state = self.lstm\_cell(x, states=state,

training=training)

# Convert the lstm output to a prediction.

prediction = self.dense(x)

# Add the prediction to the output.

predictions.append(prediction)

# predictions.shape => (time, batch, features)

predictions = tf.stack(predictions)

# predictions.shape => (batch, time, features)

predictions = tf.transpose(predictions, [1, 0, 2])

return predictions

FeedBack.call = call

print('Output shape (batch, time, features): ', feedback\_model(multi\_window.example[0]).shape)

history = compile\_and\_fit(feedback\_model, multi\_window)

IPython.display.clear\_output()

multi\_val\_performance['AR LSTM'] = feedback\_model.evaluate(multi\_window.val)

multi\_performance['AR LSTM'] = feedback\_model.evaluate(multi\_window.test, verbose=0)

multi\_window.plot(feedback\_model)

## Performance-3

x = np.arange(len(multi\_performance))

width = 0.3

metric\_name = 'mean\_absolute\_error'

metric\_index = lstm\_model.metrics\_names.index('mean\_absolute\_error')

val\_mae = [v[metric\_index] for v in multi\_val\_performance.values()]

test\_mae = [v[metric\_index] for v in multi\_performance.values()]

plt.bar(x - 0.17, val\_mae, width, label='Validation')

plt.bar(x + 0.17, test\_mae, width, label='Test')

plt.xticks(ticks=x, labels=multi\_performance.keys(),

rotation=45)

plt.ylabel(f'MAE (average over all times and outputs)')

\_ = plt.legend()

for name, value in multi\_performance.items():

print(f'{name:8s}: {value[1]:0.4f}')