Таблица 1 – Описание столбцов набора данных

Название столбца (признака)	Описание
data_day	День торга, дата в формате гггг-мм-дд
open_usd	Цена одного биткоина на момент открытия торгов, доллары США
close_usd	Цена одного биткоина на момент закрытия торгов, доллары США
high_usd	Наибольшая цена одного биткоина в день торга, доллары США
low_usd	Наименьшая цена одного биткоина в день торга, доллары США
volume	Объём торгов, кол-во акций

Набор данных собран пользователем платформы Kaggle – системы организации конкурсов по исследованию данных – Ахмедом Адамом, и доступен по ссылке – https://www.kaggle.com/ahmedadam415/digital-currency-time-series/.

На рисунке 1 представлены графики значений признаков open_usd, high_usd, close_usd на протяжении некоторого времени.

В настоящей работе необходимо предсказать цену акции на момент закрытия торгов (close_usd), учитывая текущее значение признаков data_day, open_usd, high_usd, low_usd, volume.

3 Архитектура нейронной сети

Искусственная нейронная сеть (нейросеть, сеть, ИНС) – это способ собрать нейроны в сеть так, чтобы она решала определённую задачу, например, задачу классификации.

Нейроны собираются по слоям. Есть входной слой, куда подаётся входной сигнал, есть выходной слой, откуда снимается результат работы нейросети, и между ними есть скрытые слои. Если скрытых слоёв больше, чем

один, нейросеть считается глубокой [1], если один слой, то сеть неглубокая.

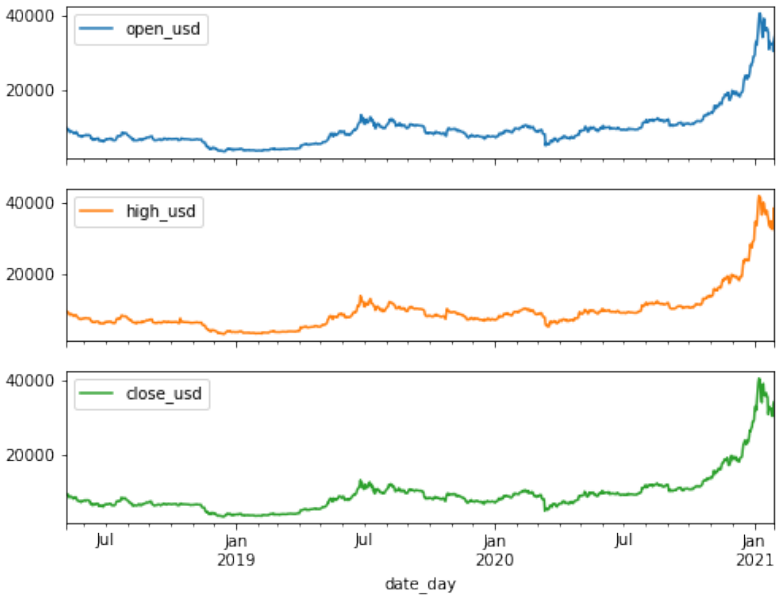


Рисунок 1 – Графики значений признаков на протяжении некоторого времени (начало)

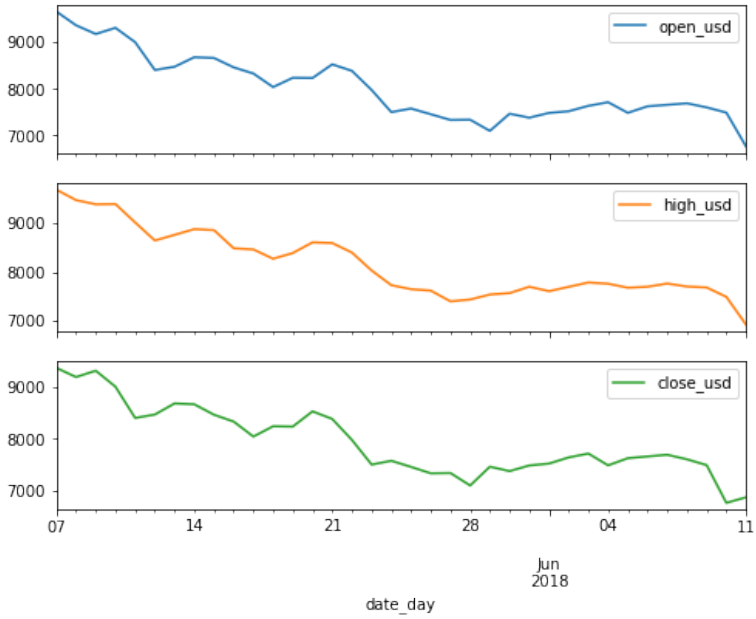


Рисунок 1 – Графики значений признаков на протяжении некоторого времени (окончание)

Для прогнозирования цены акций на момент закрытия торгов (далее, цена акции) были построены несколько одно- и многошаговых моделей. Далее, рассмотрим архитектуры этих моделей. Для оценки моделей набор данных был разделён на три части в соотношении 70%, 20%, 10%: набор для обучения, для валидации, для тестирования

4 Одношаговые модели

Одношаговая модель – модель, которая предсказывает значение одного признака на 1 шаг в будущем, основываясь только на текущих значениях признаков. На рисунке 3 представлена схема одношаговой модели.

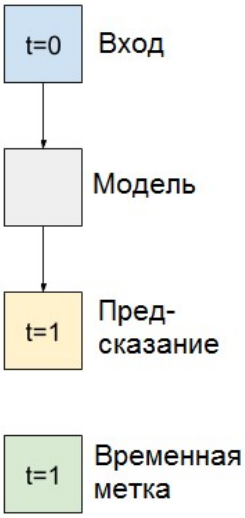


Рисунок 3 – Схема одношаговой модели

На входной слой модели подаются признаки (или один признак), на выходном слое модель выдаёт предсказания значений признаков (или значение одного признака) только для следующей временной метки (на 1 шаг в будущем).

4.1 Линия отсчёта

Перед построением обучаемой модели хорошо бы иметь некоторую линию отсчёта (baseline) в качестве примера для сравнения с более сложными моделями.

Создадим модель, которая возвращает в качестве прогноза цену акции за текущий момент времени, иными словами модель предсказывает никаких изменений. На рисунке 4 представлена схема линии отсчёта.

Такая линия отсчёта имеет смысл, так как цена акции меняется медленно изо дня в день, но эта модель будет работать плохо, если делать прогноз через несколько дней.

Создадим линию отсчёта и предскажем с помощью неё нормированную

цену акции на момент закрытия торгов за последующие 24 дня. На рисунке 5 представлен график предсказаний линии отсчёта.

Рисунок 4 – Схема линии отсчёта

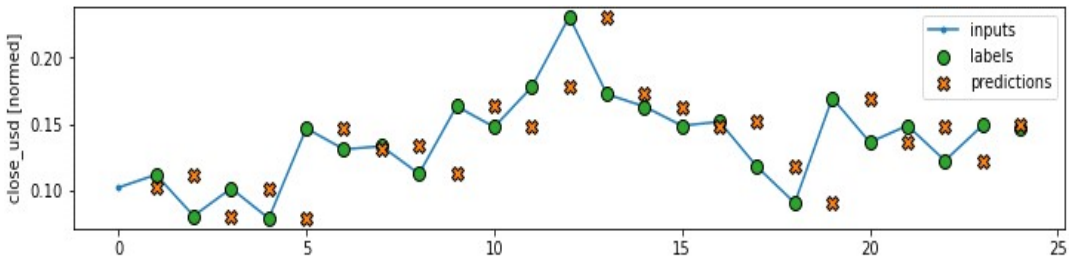


Рисунок 5 – График предсказаний линии отсчёта

Синими точками (соединены синей линией) отмечены входные значения цены акции. Модель получает все признаки, но на данном графике представлена только цена акции.

Зелёные кружки показывают целевое значение прогноза. Эти точки отображаются во время прогнозирования, а не во время ввода. Именно поэтому диапазон кружков смещён на 1 шаг относительно входных данных.

Крестики – это предсказания модели для каждого выходного временного шага. Если бы модель предсказывала идеально, предсказания попадали бы прямо на зелёные кружки.

4.2 Линейная модель

Самая простая обучаемая модель, которую возможно применить к задаче предсказания цены акции, это вставить линейное преобразование между входом и выходом. В этом случае значение на временном шаге зависит только от этого шага.

Схема линейной модели соответствует схеме одношаговой модели

(рисунок 3). Как и в случае линии отсчёта, на каждом временном шаге предсказание не зависит от других предсказаний.

На рисунке 6 представлен график предсказаний линейной модели. Линейная модель предсказывает хуже, чем линия отсчёта.

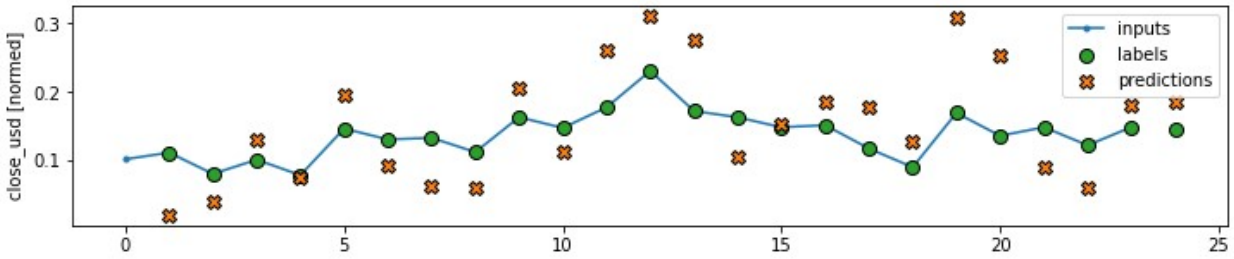


Рисунок 6 – График предсказаний линейной модели

4.3 Многошаговая полносвязанная модель

Модель с одним временным шагом не имеет контекста для текущих значений своих входных данных. Она не может видеть, как входные функции меняются с течением времени. Для решения этой проблемы модели необходим доступ к нескольким временным шагам при составлении прогнозов.

Линия отсчёта и линейная модель обрабатывали каждый временной шаг независимо. Многошаговая модель будет принимать несколько временных шагов в качестве входных данных для получения одного вывода.

Создадим многошаговую полносвязанную модель и предскажем с помощью неё нормированную цену акций на момент закрытия торгов на следующий день, на вход модели подадим значения за три дня. На рисунке 7 представлен график предсказаний многошаговой полносвязанной модели.

4.4 Свёрточная нейронная сеть

Создадим свёрточную нейронную сеть (СНС). Эта нейросеть состоит из двух слоёв:

- свёрточный слой с размером фильтра 32 и размером ядра 3;
- слой из 32-ух полносвязанных нейронов.

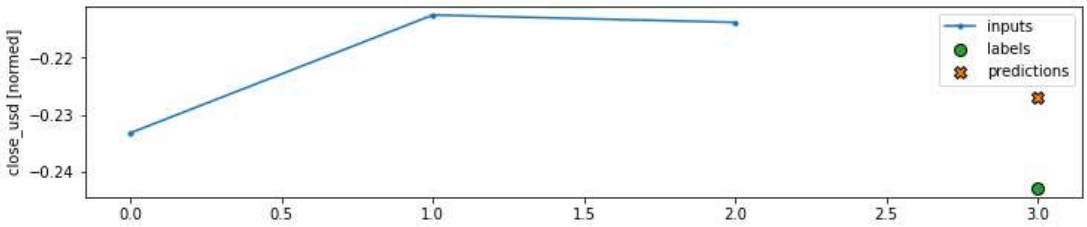


Рисунок 7 – График предсказаний многошаговой полносвязанной модели

Предсказание этой модели точнее линейной: погрешность ~0.01.

Разница между этой моделью и многошаговой полносвязанной моделью заключается в том, что свёрточная модель может выполняться на входах любой длины. Свёрточный слой наносится на скользящее окно входных данных. На рисунке 6 представлена схема свёрточной нейронной сети.

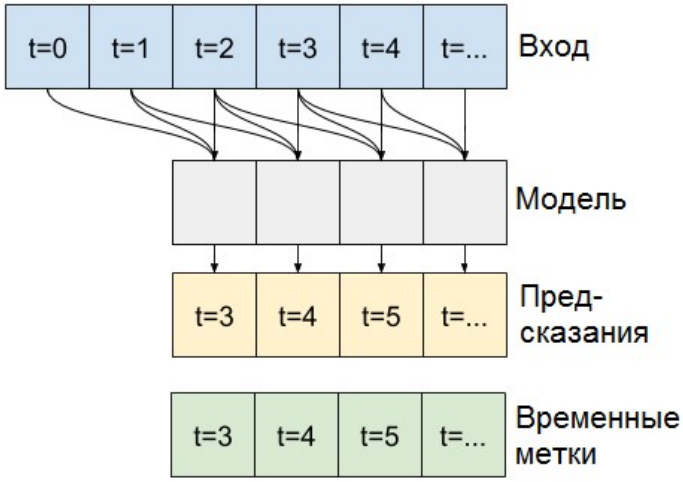


Рисунок 8 – Схема свёрточной нейронной сети

На рисунке 9 представлен график предсказаний свёрточной нейронной сети. Обратите внимание на 3 входных временных шага перед первым предсказанием: каждое предсказание здесь основано на 3 предыдущих временных шагах.

4.5 Рекуррентная нейронная сеть

Рекуррентные нейронные сети (РНС) – это класс нейронных сетей, который эффективен для моделирования данных последовательности, таких как временные ряды или естественный язык.

РНС использует цикл for для перебора временных шагов последовательности, сохраняя при этом внутреннее состояние, которое

кодирует информацию о временных шагах, которые она видела.

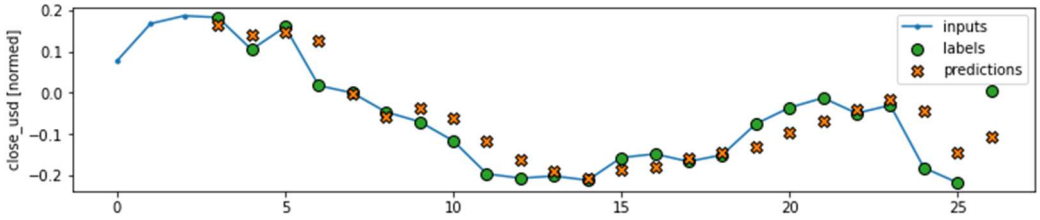


Рисунок 9 – График предсказаний свёрточной нейронной сети

Обучение модели ведётся на нескольких шагах одновременно. На рисунке 10 представлена схема РНС.

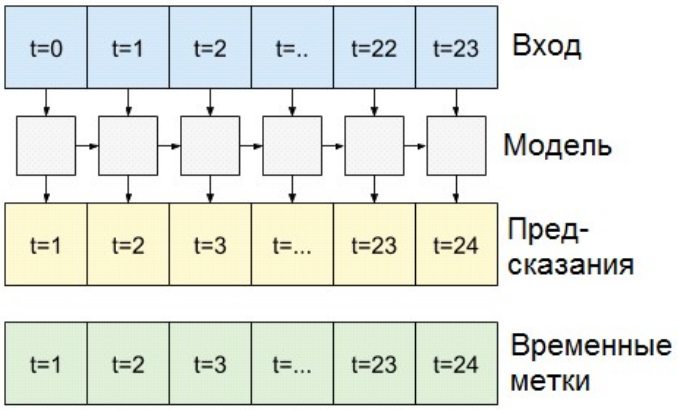
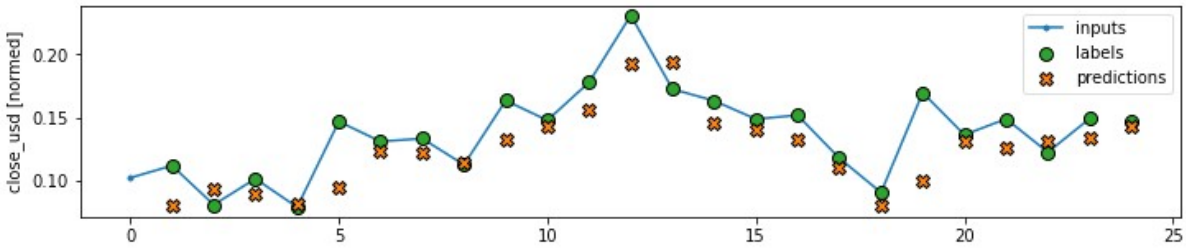


Рисунок 10 – Схема рекуррентной нейронной сети

На рисунке 11 представлен график предсказаний РНС.



Рисунке 11 – График предсказаний рекуррентной нейронной сети

4.6 Сравнение

На рисунке 12 изображена гистограмма, представляющая значение средней абсолютной ошибки (САО) для каждой модели. Линия отсчёта отмечена как baseline, линейная модель – linear, многошаговая полносвязанная модель – multi-step, СНС – conv, РНС – lstm. Многошаговая полносвязанная модель показывает лучший результат.

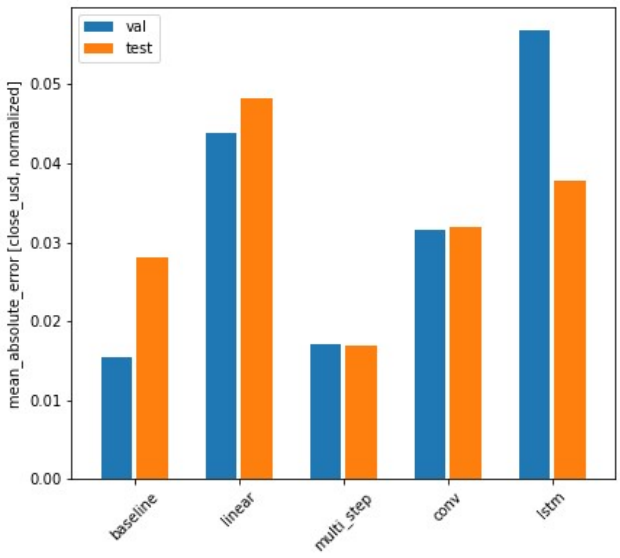


Рисунок 12 – Гистограмма средней абсолютной ошибки каждой модели

5 Многошаговые модели

При многошаговом прогнозировании модель должна научиться предсказывать диапазон будущих значений. Таким образом, в отличие от одношаговой модели, где предсказывается только одна будущая точка, многошаговая модель предсказывает последовательность будущих значений.

Существуют два подхода многошагового предсказания:

– разовые прогнозы – весь временной ряд предсказывается сразу;

– авторегрессионные прогнозы – модель предсказывает один шаг, а её выходные данные возвращаются в качестве входных данных.

5.1 Линии отсчёта

Для создания многошаговой линия отсчёта достаточно повторить последний шаг времени ввода для требуемого количества временных шагов вывода. Схема этой модели и график её предсказаний представлены на рисунках 13 и 14.

Поскольку задача многошаговой модели заключается в том, чтобы предсказать n шагов, учитывая n последних шагов, существует другой подход определения линии отсчёта, он состоит в том, чтобы повторить предыдущие n шагов, предполагая, что следующие будут такими же. Схема этой линии отсчёта и график её предсказаний представлены на рисунках 15 и 16.

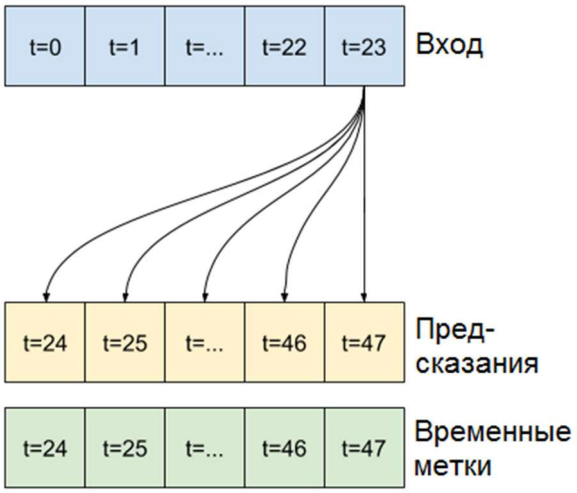


Рисунок 13 – Схема линии отсчёта (повторение одного шага)

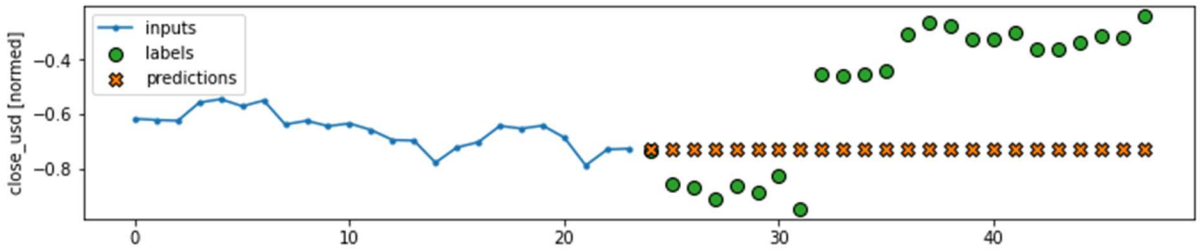


Рисунок 14 – График предсказаний линии отсчёта (повторение одного шага)

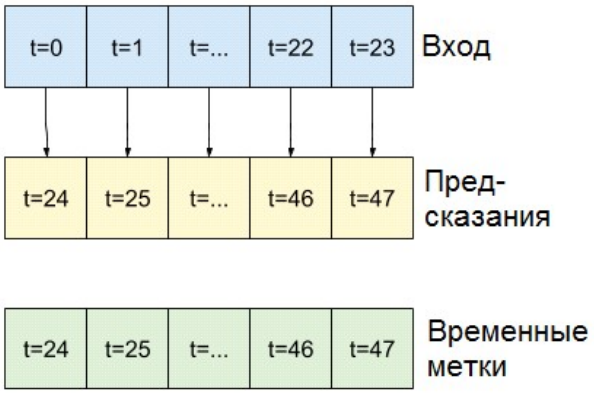


Рисунок 15 – Схема линии отсчёта (повторение последовательности шагов)

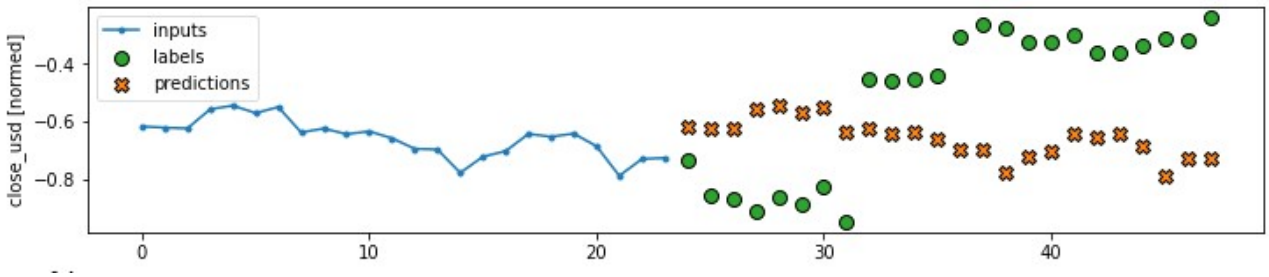


Рисунок 16 – График предсказаний линии отсчёта (повторение последовательности шагов)

5.2 Линейная и полносвязанная модели

Линейная модель и её полносвязанная модификация должна предсказывать n временных шагов с одного входного временного шага с линейной проекцией. На рисунке 17 представлена схема модели, предсказывающей n временных шагов из одного входного шага.

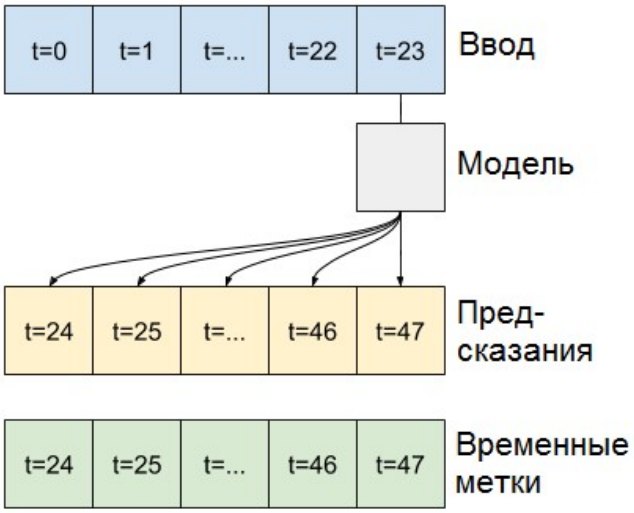


Рисунок 17 – схема модели, предсказывающей n временных шагов из одного входного шага.

На рисунках 18 и 19 представлены графики предсказаний линейной модели и её полносвязанной модификации (добавлен полносвязанный слой из 512 нейронов). Разница между этими двумя предсказаниями практически отсутствует.

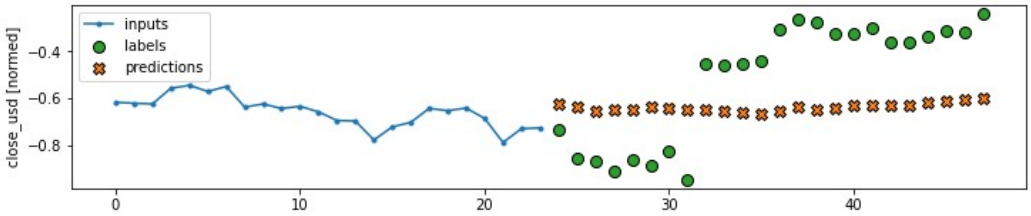


Рисунок 18 – График предсказаний линейной модели

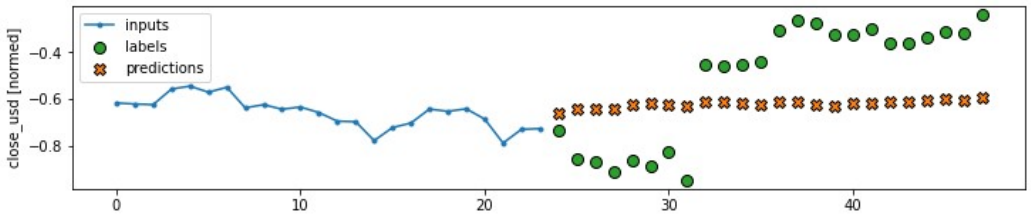


Рисунок 19 – График предсказаний полносвязанной модели

5.3 Свёрточная нейронная сеть

Свёрточная модель делает прогнозы на основе последовательности значений определённой ширины, что может привести к лучшей производительности, чем плотная модель, поскольку модель может видеть, как значение признака меняется с течением времени. На рисунке 20 представлена схема СНС, на рисунке 21 – её график предсказаний.

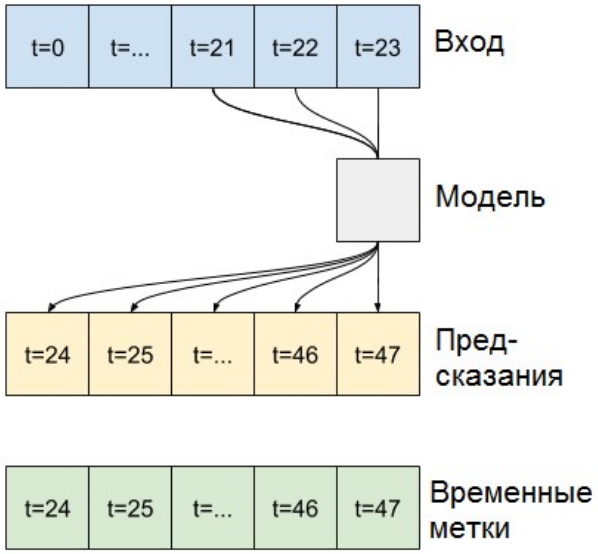


Рисунок 20 – Схема многошаговой свёрточной нейронной сети

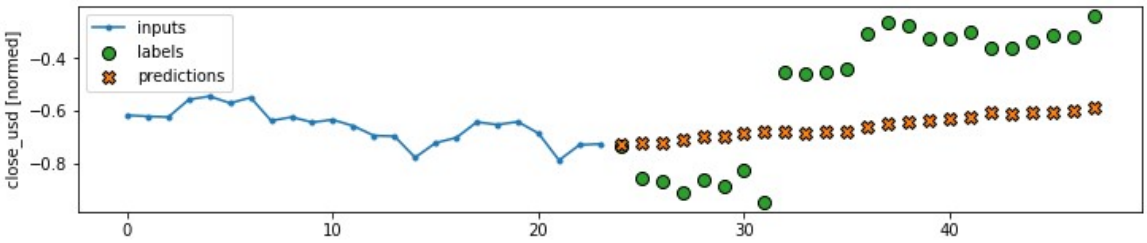


Рисунок 21 – График предсказаний многошаговой свёрточной нейронной сети

5.4 Рекуррентная нейронная сеть

Рекуррентная модель использует длинную последовательность входных данных. Модель накапливает внутреннее состояние в течение n временных шагов, прежде чем сделать один прогноз на следующие n шага. На рисунке 22 представлена схема РНС, на рисунке 23 – график предсказаний.

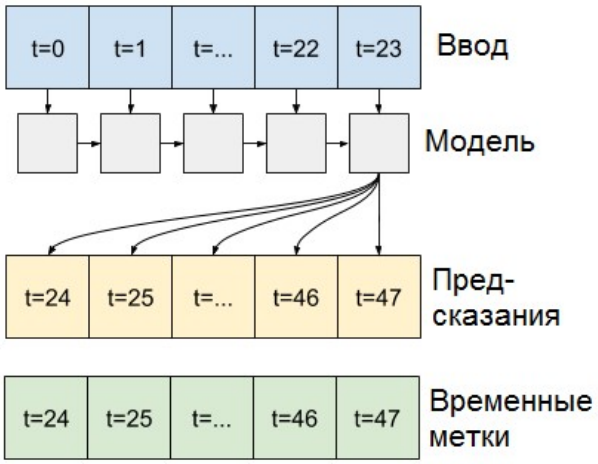


Рисунок 22 – Схема многошаговой рекуррентной нейронной сети

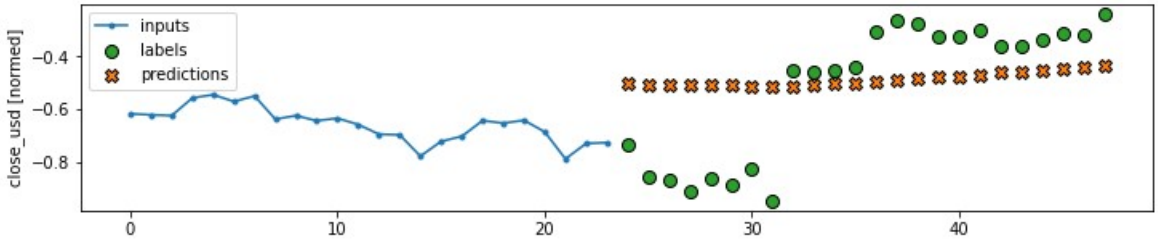


Рисунок 23 – График предсказаний многошаговой рекуррентной нейронной сети

5.5 Авторегрессионная рекуррентная нейронная сеть

В авторегрессионной модели выходные данные могут быть возвращены в саму себя на каждом шаге, и прогнозы могут быть сделаны с учётом предыдущих временных шагов.

Построим авторегрессионную РНС: слой длинной краткосрочной памяти (слой ДКП), за которым следует полносвязанный слой, которые преобразует вывод слоя ДКП в предсказания.

На рисунке 24 представлена схема авторегрессионной РНС, на рисунке 25 – её график предсказаний.

5.6 Сравнение

На рисунке 10 изображена гистограмма, представляющая значение средней абсолютной ошибки ранее описанных многошаговых модели. Линии отсчёта отмечены как last и repeat, линейная модель – linear,

многошаговая полносвязанная модель – dense, СНС – conv, РНС – lstm, авторегрессионная РНС – autoregr.

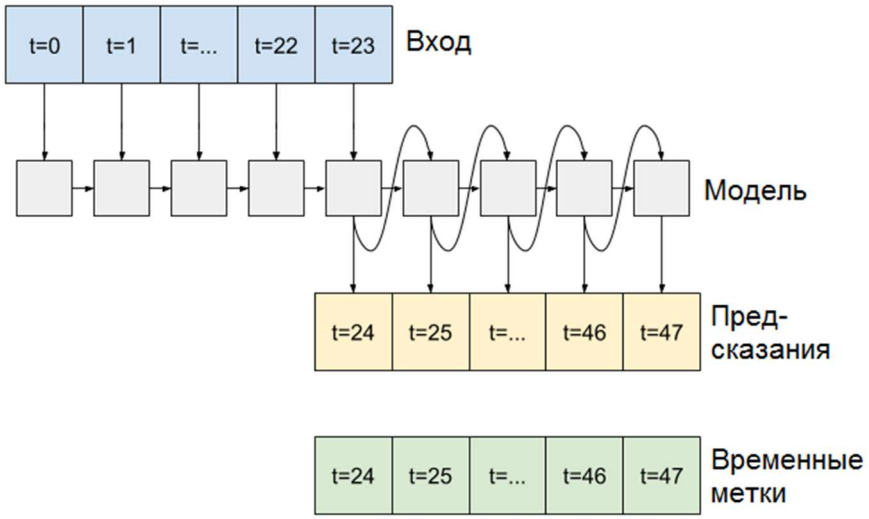


Рисунок 24 – Схема авторегрессионной рекуррентной нейронной сети

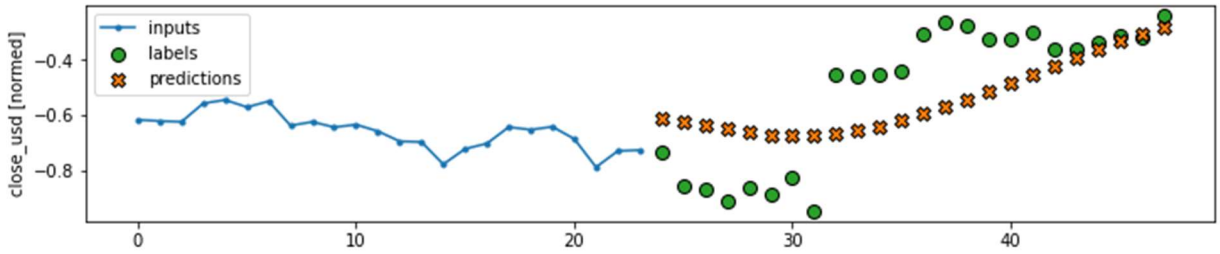


Рисунок 25 – График предсказаний авторегрессионной рекуррентной нейронной сети

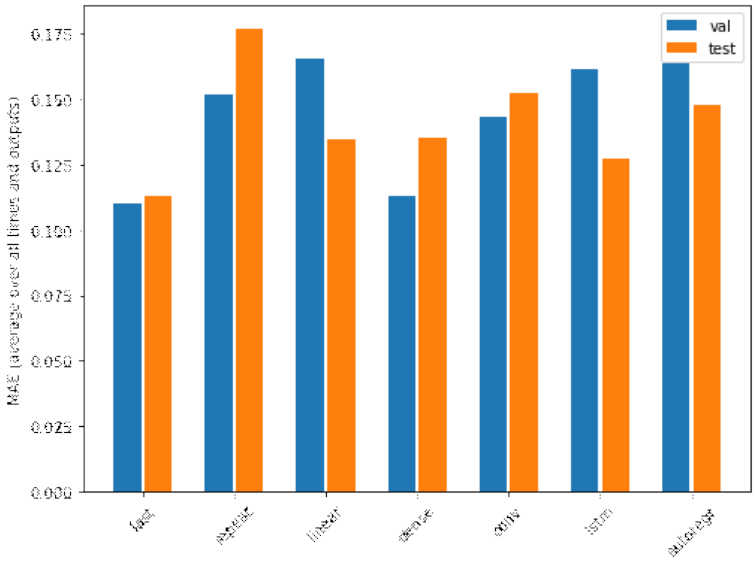


Рисунок 26 – Гистограмма средней абсолютной ошибки каждой модели

6 Вывод

Для данного набора данных любая из многошаговых моделей предсказывает хуже любой одношаговой (рисунки 12 и 26):

– среди одношаговых наибольшая САО не превышает 0.06, а наименьшая – 0.015;

– среди многошаговых наибольшая САО – 0.175, наименьшая – 0.113.

Лучше всех справляется одношаговая полносвязанная модель, её САО составляет 0.0168 на тестовом наборе – это лучше результата линии отсчёта (0.0281), предсказание которой заключалось в повторении значения предыдущего шага.

Исходный код работы на языке Python представлен в приложении А, по следующей ссылке доступен её юпитер-блокнот (jupyter notebook) – https://github.com/algorithm-ssau/image-caption-generator/tree/main/lab2-stock-forecasting.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1 Лысяк Александр Сергеевич, Рябко Борис Яковлевич Методы прогнозирования временных рядов с большим алфавитом на основе универсальной меры и деревьев принятия решений // ЖВТ. 2014. №2. URL: https://cyberleninka.ru/article/n/metody-prognozirovaniya-vremennyh-ryadov-s-bolshim-alfavitom-na-osnove-universalnoy-mery-i-dereviev-prinyatiya-resheniy (дата обращения: 20.05.2021)

Приложение А

Исходный код

```
import IPython.display
import matplotlib as mpl
import matplotlib.pyplot as plt

import numpy as np
import pandas as pd
import tensorflow as tf

mpl.rcParams['figure.figsize'] = (8, 6)
mpl.rcParams['axes.grid'] = False

df = pd.read_csv(r"D:\YandexDisk\datasets\daily-bitcoin-stock-2018-2021.csv").rename( columns={'Unnamed:
0':'date_day'}).rename(str.lower, axis='columns')
date_days = pd.to_datetime(df.pop('date_day'), format='%Y.%m.%d')

my_col_name = 'close_usd'
my_time_name = 'days'
my_plot_cols = ['open_usd', 'high_usd', 'close_usd']

my_width = 24
my_shift = 1

my_conv_width = 3
wide_label_width = 24
wide_input_width = wide_label_width + (my_conv_width - 1)

my_out_steps = 24
OUT_STEPS = 24
CONV_WIDTH = 3

df.head()

plot_features = df[my_plot_cols]
plot_features.index = date_days
hello = plot_features.plot(subplots=True)

plot_features = df[my_plot_cols][-36:]
plot_features.index = date_days[-36:]
_ = plot_features.plot(subplots=True)

column_indices = {name: i for i, name in enumerate(df.columns)}
column_indices

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):]

val_performance = {}
test_performance = {}

val_performance2 = {}
test_performance2 = {}

multi_val_performance = {}
multi_test_performance = {}
```

```
num_features = df.shape[1]

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

class WindowGenerator:
  def __init__(self, input_width, label_width, shift,
               train_df, val_df, test_df,
               label_columns=None):

    self.train_df = train_df
    self.val_df = val_df
    self.test_df = test_df

    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)}

    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 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)

    inputs.set_shape([None, self.input_width, None])
    labels.set_shape([None, self.label_width, None])

    return inputs, labels

  def make_dataset(self, data):
    data = np.array(data, dtype=np.float32)
    ds = tf.keras.preprocessing.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

@property
def train(self):
    return self.make_dataset(self.train_df)

@property
def val(self):
    return self.make_dataset(self.val_df)

@property
def test(self):
    return self.make_dataset(self.test_df)

@property
def example(self):
    result = getattr(self, '_example', None)
    if result is None:
        result = next(iter(self.train))
        self._example = result

    return result

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}'])

def plot(self, model=None, plot_col='unnamed', plot_time='unnamed', max_subplots=3):
    inputs, labels = self.example
    plot_col_index = self.column_indices[plot_col]

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

w1 = WindowGenerator(input_width=24, label_width=1, shift=24,
                     train_df=train_df, val_df=val_df, test_df=test_df,
                     label_columns=[my_col_name])
w1

w2 = WindowGenerator(input_width=6, label_width=1, shift=1,
                     train_df=train_df, val_df=val_df, test_df=test_df,
                     label_columns=[my_col_name])
w2

example_features = tf.stack([np.array(train_df[:w2.total_window_size]),
                             np.array(train_df[50:50+w2.total_window_size]),
                             np.array(train_df[100:100+w2.total_window_size])])

example_inputs, example_labels = w2.split_window(example_features)

print('all shapes are: (batch, time, features)')
print(f'window shape: {example_features.shape}')
print(f'inputs shape: {example_inputs.shape}')
print(f'labels shape: {example_labels.shape}')

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_window = WindowGenerator(
  input_width=1, label_width=1, shift=1,
  label_columns=[my_col_name],
  train_df=train_df, val_df=val_df, test_df=test_df)
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}')

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[my_col_name])
baseline.compile(loss=tf.losses.MeanSquaredError(),
                 metrics=[tf.metrics.MeanAbsoluteError()])

val_performance['baseline'] = baseline.evaluate(single_step_window.val)
test_performance['baseline'] = baseline.evaluate(single_step_window.test)

wide_window = WindowGenerator(
  input_width=my_width, label_width=my_width, shift=my_shift,
  label_columns=[my_col_name],
  train_df=train_df, val_df=val_df, test_df=test_df)
wide_window
```

```
print('input shape:', wide_window.example[0].shape)
print('output shape:', baseline(wide_window.example[0]).shape)

wide_window.plot(baseline, plot_col=my_col_name, plot_time=my_time_name)

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)

def compile_and_fit(model, window, patience=2, epochs=20):
    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=epochs,
                        validation_data=window.val,
                        callbacks=[early_stopping])

    return history

%%time
history = compile_and_fit(linear, single_step_window)

val_performance['linear'] = linear.evaluate(single_step_window.val)
test_performance['linear'] = linear.evaluate(single_step_window.test)

print('input shape:', wide_window.example[0].shape)
print('output shape:', baseline(wide_window.example[0]).shape)

wide_window.plot(linear, plot_col=my_col_name, plot_time=my_time_name)

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 = 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)
])

%%time
history = compile_and_fit(dense, single_step_window)

val_performance['dense'] = dense.evaluate(single_step_window.val)
test_performance['dense'] = dense.evaluate(single_step_window.test)

conv_window = WindowGenerator(
    input_width=my_conv_width, label_width=1, shift=my_shift,
    label_columns=[my_col_name],
    train_df=train_df, val_df=val_df, test_df=test_df)
conv_window

conv_window.plot(plot_col=my_col_name, plot_time=my_time_name)
```

```
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)

%%time
history = compile_and_fit(multi_step_dense, conv_window)

IPython.display.clear_output()
val_performance['multi_step'] = multi_step_dense.evaluate(conv_window.val)
test_performance['multi_step'] = multi_step_dense.evaluate(conv_window.test)

conv_window.plot(multi_step_dense, plot_col=my_col_name, plot_time=my_time_name)

print('input shape:', wide_window.example[0].shape)
try:
    print('output shape:', multi_step_dense(wide_window.example[0]).shape)
except Exception as e:
    print(f'\n{type(e).__name__}:{e}')

conv_model = tf.keras.Sequential([
    tf.keras.layers.Conv1D(filters=32,
                           kernel_size=(my_conv_width,),
                           activation='relu'),
    tf.keras.layers.Dense(units=32, activation='relu'),
    tf.keras.layers.Dense(units=1),
])

print('input shape:', conv_window.example[0].shape)
print('output shape:', conv_model(conv_window.example[0]).shape)

%%time
history = compile_and_fit(conv_model, conv_window)

IPython.display.clear_output()
val_performance['conv'] = conv_model.evaluate(conv_window.val)
test_performance['conv'] = conv_model.evaluate(conv_window.test)

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)

wide_conv_window = WindowGenerator(
    input_width=wide_input_width,
    label_width=wide_label_width,
    shift=my_shift,
    label_columns=[my_col_name],
    train_df=train_df, val_df=val_df, test_df=test_df)
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, plot_col=my_col_name, plot_time=my_time_name)

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)

%%time
history = compile_and_fit(lstm_model, wide_window)

IPython.display.clear_output()
val_performance['lstm'] = lstm_model.evaluate(wide_window.val)
test_performance['lstm'] = lstm_model.evaluate(wide_window.test)

wide_window.plot(lstm_model, plot_col=my_col_name, plot_time=my_time_name)

x = np.arange(len(test_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 test_performance.values()]

plt.ylabel(f'mean_absolute_error [{my_col_name}, normalized]')
plt.bar(x - 0.17, val_mae, width, label='val')
plt.bar(x + 0.17, test_mae, width, label='test')
plt.xticks(ticks=x, labels=test_performance.keys(), rotation=45)
_ = plt.legend()

for name, value in test_performance.items():
    print(f'{name:12s}: {value[1]:0.4f}')

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,
    train_df=train_df, val_df=val_df, test_df=test_df)

wide_window = WindowGenerator(
    input_width=24, label_width=24, shift=1,
    train_df=train_df, val_df=val_df, test_df=test_df)

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.compile(loss=tf.losses.MeanSquaredError(),
                 metrics=[tf.metrics.MeanAbsoluteError()])

val_performance2['baseline'] = baseline.evaluate(wide_window.val)
test_performance2['baseline'] = baseline.evaluate(wide_window.test)

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_performance2['dense'] = dense.evaluate(single_step_window.val)
test_performance2['dense'] = dense.evaluate(single_step_window.test)

%%time
wide_window = WindowGenerator(
    input_width=24, label_width=24, shift=1,
    train_df=train_df, val_df=val_df, test_df=test_df)

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_performance2['lstm'] = lstm_model.evaluate( wide_window.val)
test_performance2['lstm'] = lstm_model.evaluate( wide_window.test)

x = np.arange(len(test_performance2))
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_performance2.values()]
test_mae = [v[metric_index] for v in test_performance2.values()]

plt.bar(x - 0.17, val_mae, width, label='val')
plt.bar(x + 0.17, test_mae, width, label='test')
plt.xticks(ticks=x, labels=test_performance2.keys(),
           rotation=45)
plt.ylabel('MAE (average over all outputs)')
_ = plt.legend()

for name, value in test_performance2.items():
    print(f'{name:15s}: {value[1]:0.4f}')

multi_window = WindowGenerator(input_width=24, label_width=my_out_steps, shift=my_out_steps,
                               train_df=train_df, val_df=val_df, test_df=test_df)
multi_window.plot(plot_col=my_col_name, plot_time=my_time_name)
multi_window

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['last'] = last_baseline.evaluate(multi_window.val)
multi_test_performance['last'] = last_baseline.evaluate(multi_window.test)

multi_window.plot(last_baseline, plot_col=my_col_name, plot_time=my_time_name)
```

```
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_test_performance['repeat'] = repeat_baseline.evaluate(multi_window.test)

multi_window.plot(repeat_baseline, plot_col=my_col_name, plot_time=my_time_name)

multi_linear_model = tf.keras.Sequential([
  # 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])
])

%%time
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_test_performance['linear'] = multi_linear_model.evaluate(multi_window.test)

multi_window.plot(multi_linear_model, plot_col=my_col_name, plot_time=my_time_name)

multi_dense_model = tf.keras.Sequential([
  # 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])
])

%%time
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_test_performance['dense'] = multi_dense_model.evaluate(multi_window.test, verbose=0)

multi_window.plot(multi_dense_model, plot_col=my_col_name, plot_time=my_time_name)

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])
])
```

```
%%time
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_test_performance['conv'] = multi_conv_model.evaluate(multi_window.test)

multi_window.plot(multi_conv_model, plot_col=my_col_name, plot_time=my_time_name)

multi_lstm_model = tf.keras.Sequential([
  # shape [batch, time, features] => [batch, lstm_units]
  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])
])

%%time
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_test_performance['lstm'] = multi_lstm_model.evaluate(multi_window.test)

multi_window.plot(multi_lstm_model, plot_col=my_col_name, plot_time=my_time_name)

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)
    self.lstm_rnn = tf.keras.layers.RNN(self.lstm_cell, return_state=True)
    self.dense = tf.keras.layers.Dense(num_features)

  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

  def call(self, inputs, training=None):
    predictions = []
    prediction, state = self.warmup(inputs)

    predictions.append(prediction)

    for n in range(1, self.out_steps):
      x = prediction
      x, state = self.lstm_cell(x, states=state,
                                training=training)

      # Convert the lstm output to a prediction.
      prediction = self.dense(x)
      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_model = Feedback(units=32, out_steps=OUT_STEPS)

prediction, state = feedback_model.warmup(multi_window.example[0])
prediction.shape

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['autoregr'] = feedback_model.evaluate(multi_window.val)
multi_test_performance['autoregr'] = feedback_model.evaluate(multi_window.test)

multi_window.plot(feedback_model, plot_col=my_col_name, plot_time=my_time_name)

x = np.arange(len(multi_test_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_test_performance.values()]

plt.bar(x - 0.17, val_mae, width, label='val')
plt.bar(x + 0.17, test_mae, width, label='test')
plt.xticks(ticks=x, labels=multi_test_performance.keys(),
           rotation=45)
plt.ylabel(f'MAE (average over all times and outputs)')
_ = plt.legend()
```