Open in Colab

10. Milestone Project 3: Time series forecasting in TensorFlow (BitPredict)

The goal of this notebook is to get you familiar with working with time series data.

We're going to be building a series of models in an attempt to predict the price of Bitcoin.

Welcome to Milestone Project 3, BitPredict !

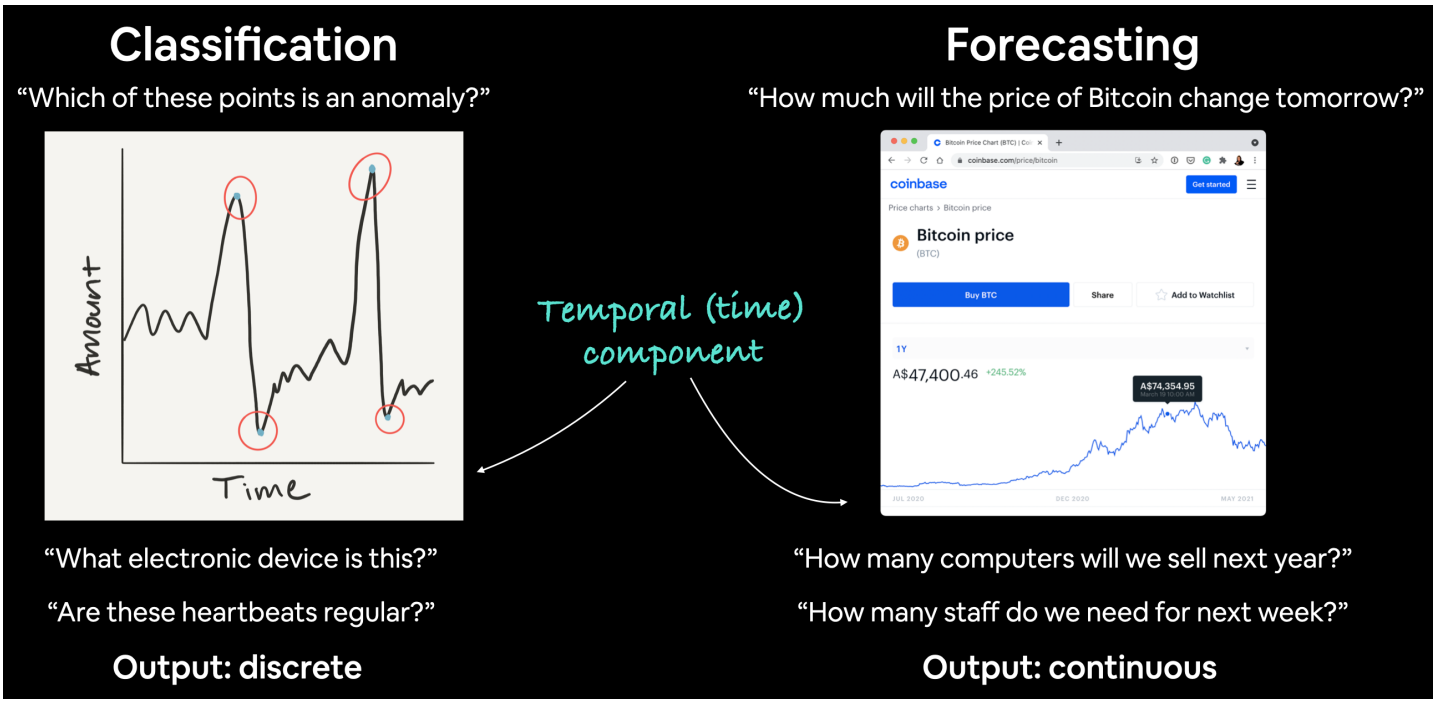
> **Note:** ⚠ This is not financial advice, as you'll see time series forecasting for stock market prices is actually quite terrible.

What is a time series problem?

Time series problems deal with data over time.

Such as, the number of staff members in a company over 10-years, sales of computers for the past 5-years, electricity usage for the past 50-years.

The timeline can be short (seconds/minutes) or long (years/decades). And the problems you might investigate using can usually be broken down into two categories.



Problem Type	Examples	Output
Classification	Anomaly detection, time series identification (where did this time series come from?)	Discrete (a label)
Forecasting	Predicting stock market prices, forecasting future demand for a product, stocking inventory requirements	Continuous (a number)

In both cases above, a supervised learning approach is often used. Meaning, you'd have some example data and a label assosciated with that data.

For example, in forecasting the price of Bitcoin, your data could be the historical price of Bitcoin for the past month and the label could be today's price (the label can't be tomorrow's price because that's what we'd want to predict).

Can you guess what kind of problem BitPredict is?

What we're going to cover

Are you ready?

We've got a lot to go through.

- **Get time series data (the historical price of Bitcoin)**
  - **Load in time series data using pandas/Python's CSV module**
- **Format data for a time series problem**
  - **Creating training and test sets (the wrong way)**
  - **Creating training and test sets (the right way)**
  - **Visualizing time series data**
  - **Turning time series data into a supervised learning problem (windowing)**
  - **Preparing univariate and multivariate (more than one variable) data**
- **Evaluating a time series forecasting model**
- **Setting up a series of deep learning modelling experiments**
  - **Dense (fully-connected) networks**
  - **Sequence models (LSTM and 1D CNN)**
  - **Ensembling (combining multiple models together)**
  - **Multivariate models**
  - **Replicating the N-BEATS algorithm using TensorFlow layer subclassing**
- **Creating a modelling checkpoint to save the best performing model during training**
- **Making predictions (forecasts) with a time series model**
- **Creating prediction intervals for time series model forecasts**
- **Discussing two different types of uncertainty in machine learning (data uncertainty and model uncertainty)**
- **Demonstrating why forecasting in an open system is BS (the turkey problem)**

How you can use this notebook

You can read through the descriptions and the code (it should all run), but there's a better option.

Write all of the code yourself.

Yes. I'm serious. Create a new notebook, and rewrite each line by yourself. Investigate it, see if you can break it, why does it break?

You don't have to write the text descriptions but writing the code yourself is a great way to get hands-on experience.

Don't worry if you make mistakes, we all do. The way to get better and make less mistakes is to write more code.

> **Resource:** Get all of the materials you need for this notebook on the course GitHub.

Check for GPU

In order for our deep learning models to run as fast as possible, we'll need access to a GPU.

In Google Colab, you can set this up by going to Runtime -> Change runtime type -> Hardware accelerator -> GPU.

After selecting GPU, you may have to restart the runtime.

In [ ]:

```
# Check for GPU
!nvidia-smi -L
```

```
GPU 0: Tesla K80 (UUID: GPU-c7456639-4229-1150-8316-e4197bf2c93e)
```

Get data

To build a time series forecasting model, the first thing we're going to need is data.

And since we're trying to predict the price of Bitcoin, we'll need Bitcoin data.

Specifically, we're going to get the prices of Bitcoin from 01 October 2013 to 18 May 2021.

Why these dates?

Because 01 October 2013 is when our data source (Coindesk) started recording the price of Bitcoin and 18 May 2021 is when this notebook was created.

If you're going through this notebook at a later date, you'll be able to use what you learn to predict on later dates of Bitcoin, you'll just have to adjust the data source.

> **Resource:** To get the Bitcoin historical data, I went to the Coindesk page for Bitcoin prices, clicked on "all" and then clicked on "Export data" and selected "CSV".

You can find the data we're going to use on GitHub.

In [ ]:

```
# Download Bitcoin historical data from GitHub
# Note: you'll need to select "Raw" to download the data in the correct format
!wget https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv
```

```
--2021-09-27 03:40:22--  https://raw.githubusercontent.com/mrdbourke/tensorflow-deep-learning/main/extras/BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv
Resolving raw.githubusercontent.com (raw.githubusercontent.com)... 185.199.108.133, 185.199.109.133, 185.199.110.133, ...
Connecting to raw.githubusercontent.com (raw.githubusercontent.com)|185.199.108.133|:443... connected.
HTTP request sent, awaiting response... 200 OK
Length: 178509 (174K) [text/plain]
Saving to: 'BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv'

BTC_USD_2013-10-01_ 100%[===================>] 174.33K  --.-KB/s    in 0.01s

2021-09-27 03:40:22 (16.5 MB/s) - 'BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv' saved [178509/178509]
```

Importing time series data with pandas

Now we've got some data to work with, let's import it using pandas so we can visualize it.

Because our data is in CSV (comma separated values) format (a very common data format for time series), we'll use the pandas `read_csv()` function.

And because our data has a date component, we'll tell pandas to parse the dates using the `parse_dates` parameter passing it the name our of the date column ("Date").

In [ ]:

```
# Import with pandas
import pandas as pd
# Parse dates and set date column to index
df = pd.read_csv("/content/BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv",
                 parse_dates=["Date"],
                 index_col=["Date"]) # parse the date column (tell pandas column 1 is a datetime)
df.head()
```

Out[ ]:

Date	Currency	Closing Price (USD)	24h Open (USD)	24h High (USD)	24h Low (USD)
2013-10-01	BTC	123.65499	124.30466	124.75166	122.56349
2013-10-02	BTC	125.45500	123.65499	125.75850	123.63383
2013-10-03	BTC	108.58483	125.45500	125.66566	83.32833
2013-10-04	BTC	118.67466	108.58483	118.67500	107.05816
2013-10-05	BTC	121.33866	118.67466	121.93633	118.00566

Looking good! Let's get some more info.

In [ ]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
DatetimeIndex: 2787 entries, 2013-10-01 to 2021-05-18
Data columns (total 5 columns):
 #   Column               Non-Null Count  Dtype  
---  ------               --------------  -----  
 0   Currency             2787 non-null   object 
 1   Closing Price (USD)  2787 non-null   float64
 2   24h Open (USD)       2787 non-null   float64
 3   24h High (USD)       2787 non-null   float64
 4   24h Low (USD)        2787 non-null   float64
dtypes: float64(4), object(1)
memory usage: 130.6+ KB
```

Because we told pandas to parse the date column and set it as the index, its not in the list of columns.

You can also see there isn't many samples.

In [ ]:

```
# How many samples do we have?
len(df)
```

Out[ ]:

```
2787
```

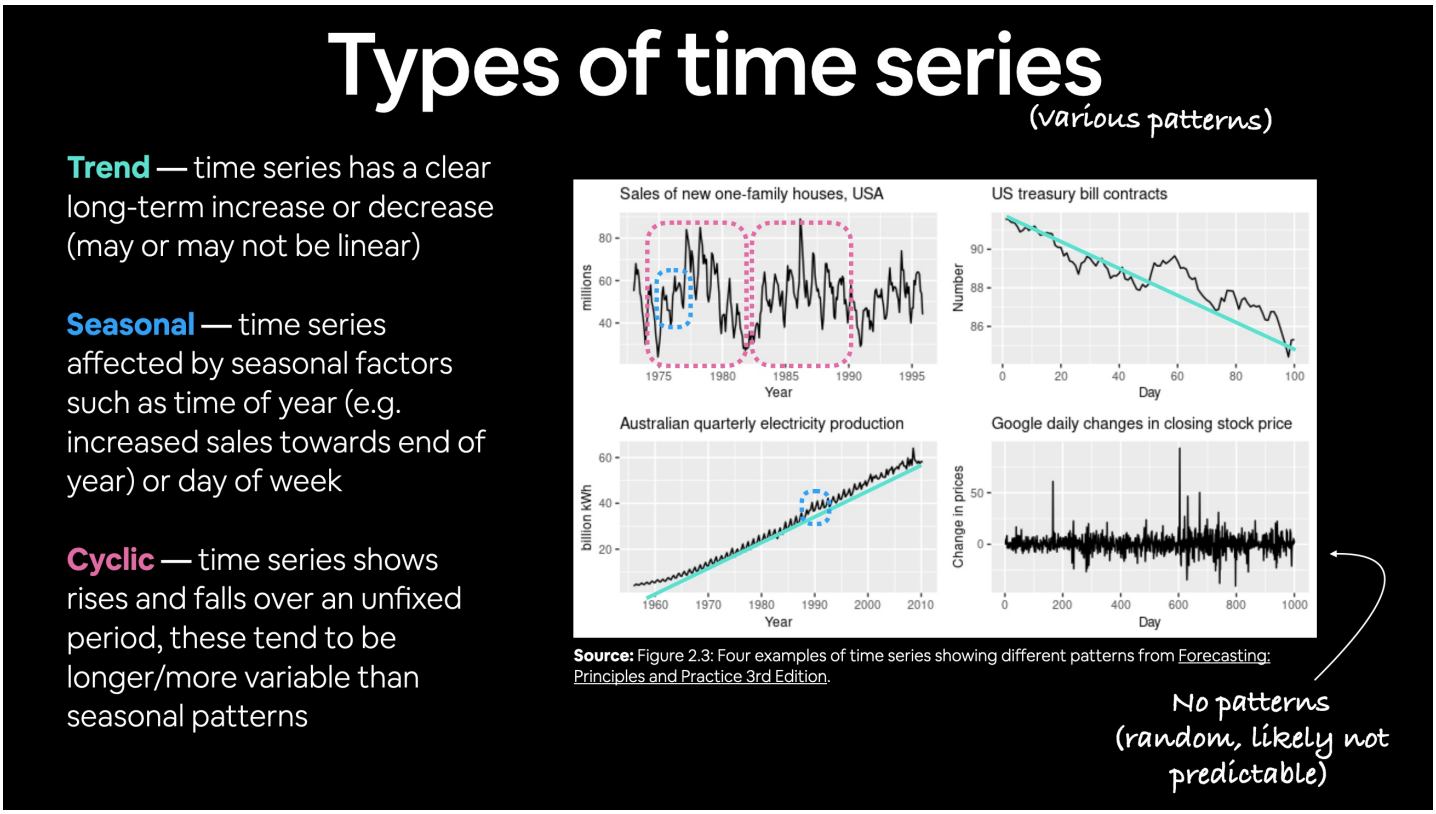
We've collected the historical price of Bitcoin for the past ~8 years but there's only 2787 total samples.

This is something you'll run into with time series data problems. Often, the number of samples isn't as large as other kinds of data.

For example, collecting one sample at different time frames results in:

1 sample per timeframe	Number of samples per year
Second	31,536,000
Hour	8,760
Day	365
Week	52
Month	12

> **Note:** The frequency at which a time series value is collected is often referred to as **seasonality**. This is usually mesaured in number of samples per year. For example, collecting the price of Bitcoin once per day would result in a time series with a seasonality of 365. Time series data collected with different seasonality values often exhibit seasonal patterns (e.g. electricity demand behing higher in Summer months for air conditioning than Winter months). For more on different time series patterns, see Forecasting: Principles and Practice Chapter 2.3.



Example of different kinds of patterns you'll see in time series data. Notice the bottom right time series (Google stock price changes) has little to no patterns, making it difficult to predict. See Forecasting: Principles and Practice Chapter 2.3 for full graphic.

Deep learning algorithms usually flourish with lots of data, in the range of thousands to millions of samples.

In our case, we've got the daily prices of Bitcoin, a max of 365 samples per year.

But that doesn't we can't try them with our data.

To simplify, let's remove some of the columns from our data so we're only left with a date index and the closing price.

In [ ]:

```
# Only want closing price for each day
bitcoin_prices = pd.DataFrame(df["Closing Price (USD)"]).rename(columns={"Closing Price (USD)": "Price"})
bitcoin_prices.head()
```

Out[ ]:

	Price
Date	
2013-10-01	123.65499
2013-10-02	125.45500
2013-10-03	108.58483
2013-10-04	118.67466
2013-10-05	121.33866

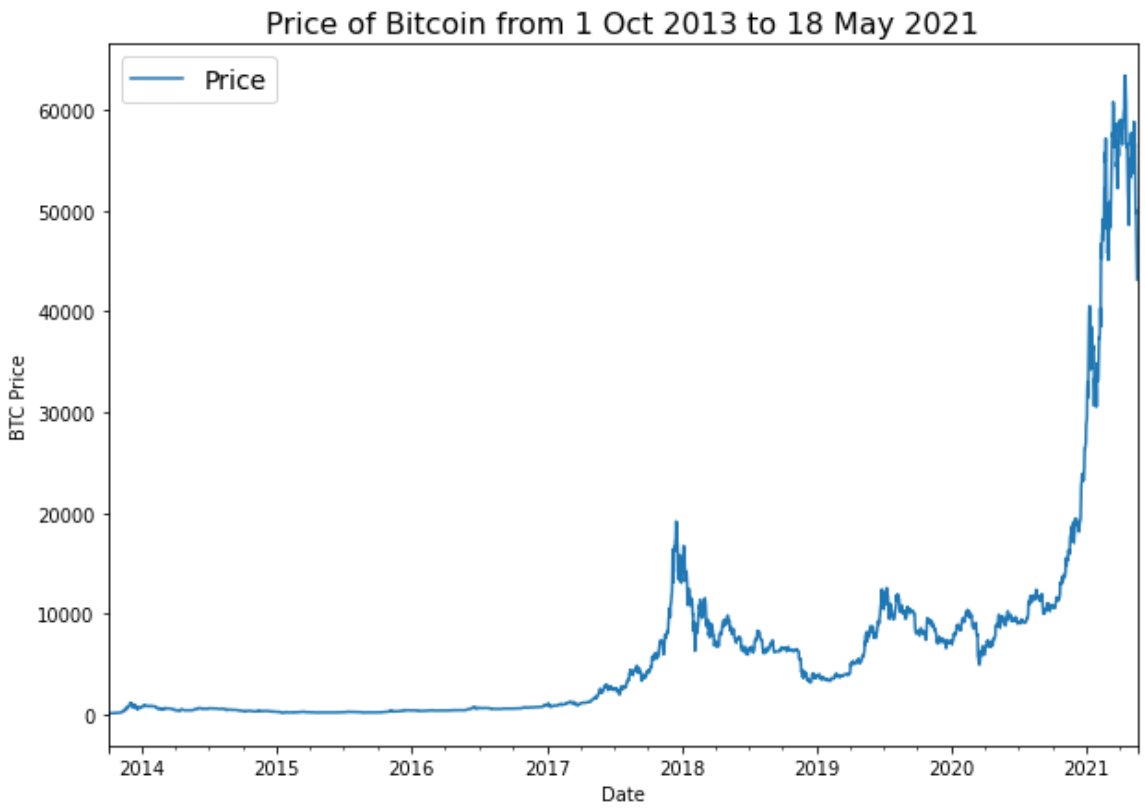
Much better!

But that's only five days worth of Bitcoin prices, let's plot everything we've got.

In [ ]:

```
import matplotlib.pyplot as plt
bitcoin_prices.plot(figsize=(10, 7))
plt.ylabel("BTC Price")
```

```
plt.title("Price of Bitcoin from 1 Oct 2013 to 18 May 2021", fontsize=16)
plt.legend(fontsize=14);
```



Woah, looks like it would've been a good idea to buy Bitcoin back in 2014.

Importing time series data with Python's CSV module

If your time series data comes in CSV form you don't necessarily have to use pandas.

You can use Python's in-built `csv` module. And if you're working with dates, you might also want to use Python's `datetime`.

Let's see how we can replicate the plot we created before except this time using Python's `csv` and `datetime` modules.

> **Resource:** For a great guide on using Python's `csv` module, check out Real Python's tutorial on Reading and Writing CSV files in Python.

In [ ]:

```
# Importing and formatting historical Bitcoin data with Python
import csv
from datetime import datetime

timesteps = []
btc_price = []
with open("/content/BTC_USD_2013-10-01_2021-05-18-CoinDesk.csv", "r") as f:
  csv_reader = csv.reader(f, delimiter=",") # read in the target CSV
  next(csv_reader) # skip first line (this gets rid of the column titles)
  for line in csv_reader:
    timesteps.append(datetime.strptime(line[1], "%Y-%m-%d")) # get the dates as dates (no
t strings), strptime = string parse time
    btc_price.append(float(line[2])) # get the closing price as float

# View first 10 of each
timesteps[:10], btc_price[:10]
```

Out[ ]:

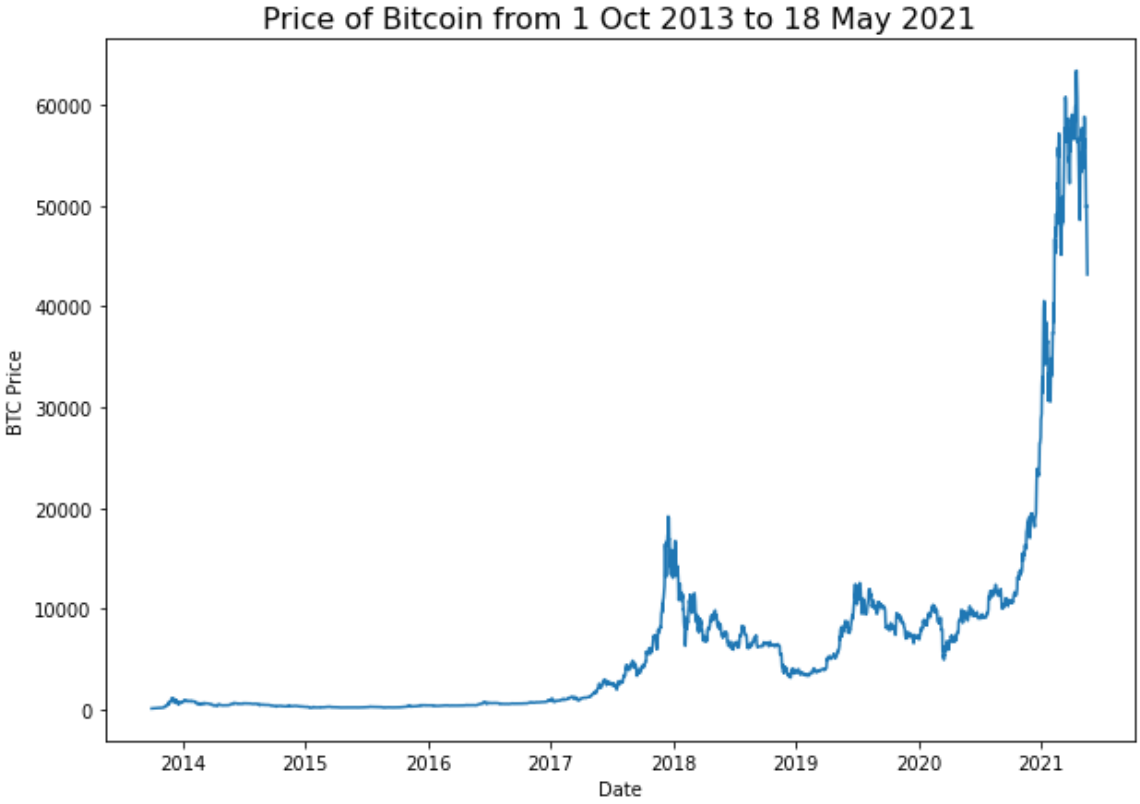
```
([datetime.datetime(2013, 10, 1, 0, 0),
```

```
  datetime.datetime(2013, 10, 2, 0, 0),
  datetime.datetime(2013, 10, 3, 0, 0),
  datetime.datetime(2013, 10, 4, 0, 0),
  datetime.datetime(2013, 10, 5, 0, 0),
  datetime.datetime(2013, 10, 6, 0, 0),
  datetime.datetime(2013, 10, 7, 0, 0),
  datetime.datetime(2013, 10, 8, 0, 0),
  datetime.datetime(2013, 10, 9, 0, 0),
  datetime.datetime(2013, 10, 10, 0, 0)],
 [123.65499,
  125.455,
  108.58483,
  118.67466,
  121.33866,
  120.65533,
  121.795,
  123.033,
  124.049,
  125.96116])
```

Beautiful! Now, let's see how things look.

In [ ]:

```
# Plot from CSV
import matplotlib.pyplot as plt
import numpy as np
plt.figure(figsize=(10, 7))
plt.plot(timesteps, btc_price)
plt.title("Price of Bitcoin from 1 Oct 2013 to 18 May 2021", fontsize=16)
plt.xlabel("Date")
plt.ylabel("BTC Price");
```



Ho ho! Would you look at that! Just like the pandas plot. And because we formatted the `timesteps` to be `datetime` objects, `matplotlib` displays a fantastic looking date axis.

Format Data Part 1: Creatining train and test sets for time series data

Alrighty. What's next?

If you guessed preparing our data for a model, you'd be right.

What's the most important first step for preparing any machine learning dataset?

Scaling?

No...

Removing outliers?

No...

How about creating train and test splits?

Yes!

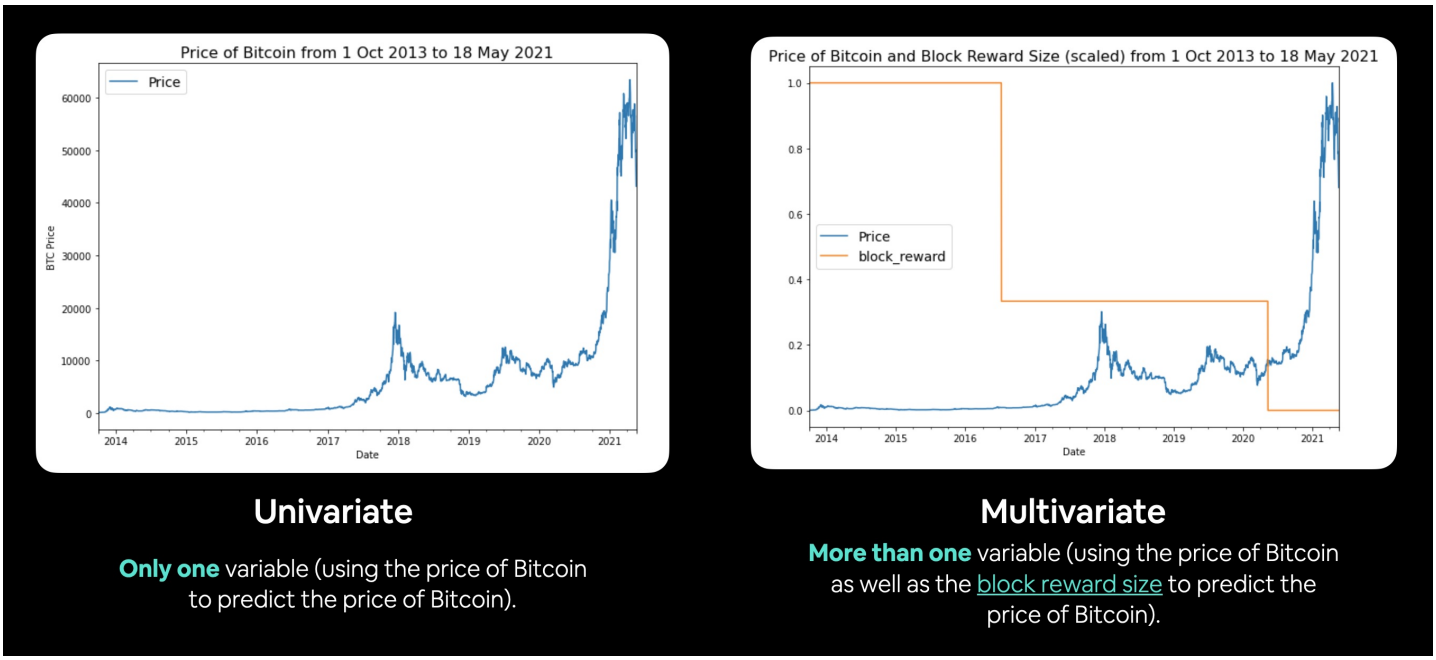
Usually, you could create a train and test split using a function like Scikit-Learn's outstanding `train_test_split()` but as we'll see in a moment, this doesn't really cut it for time series data.

But before we do create splits, it's worth talking about what *kind* of data we have.

In time series problems, you'll either have univariate or multivariate data.

Can you guess what our data is?

- **Univariate** time series data deals with *one* variable, for example, using the price of Bitcoin to predict the price of Bitcoin.
- **Multivariate** time series data deals with *more than one* variable, for example, predicting electricity demand using the day of week, time of year and number of houses in a region.



Example of univariate and multivariate time series data. Univariate involves using the target to predict the target. Multivariate inolves using the target as well as another time series to predict the target.

Create train & test sets for time series (the wrong way)

Okay, we've figured out we're dealing with a univariate time series, so we only have to make a split on one variable (for multivariate time series, you will have to split multiple variables).

How about we first see the *wrong way* for splitting time series data?

Let's turn our DataFrame index and column into NumPy arrays.

In [ ]:

```
# Get bitcoin date array
timesteps = bitcoin_prices.index.to_numpy()
prices = bitcoin_prices["Price"].to_numpy()
```

```
timesteps[:10], prices[:10]
```

Out[ ]:

```
(array(['2013-10-01T00:00:00.000000000', '2013-10-02T00:00:00.000000000',
        '2013-10-03T00:00:00.000000000', '2013-10-04T00:00:00.000000000',
        '2013-10-05T00:00:00.000000000', '2013-10-06T00:00:00.000000000',
        '2013-10-07T00:00:00.000000000', '2013-10-08T00:00:00.000000000',
        '2013-10-09T00:00:00.000000000', '2013-10-10T00:00:00.000000000'],
       dtype='datetime64[ns]'),
 array([123.65499, 125.455  , 108.58483, 118.67466, 121.33866, 120.65533,
        121.795  , 123.033  , 124.049  , 125.96116]))
```

And now we'll use the ever faithful `train_test_split` from Scikit-Learn to create our train and test sets.

In [ ]:

```
# Wrong way to make train/test sets for time series
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(timesteps, # dates
                                                    prices, # prices
                                                    test_size=0.2,
                                                    random_state=42)
X_train.shape, X_test.shape, y_train.shape, y_test.shape
```

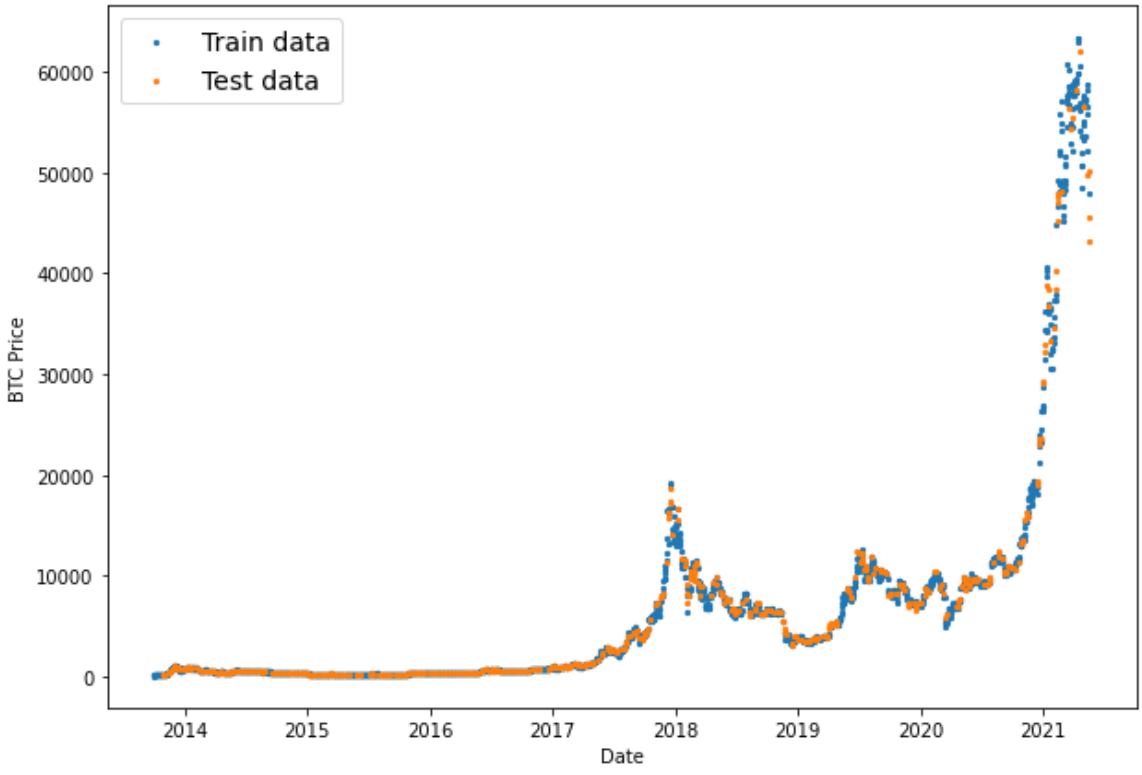
Out[ ]:

```
((2229,), (558,), (2229,), (558,))
```

Looks like the splits worked well, but let's not trust numbers on a page, let's visualize, visualize, visualize!

In [ ]:

```
# Let's plot wrong train and test splits
plt.figure(figsize=(10, 7))
plt.scatter(X_train, y_train, s=5, label="Train data")
plt.scatter(X_test, y_test, s=5, label="Test data")
plt.xlabel("Date")
plt.ylabel("BTC Price")
plt.legend(fontsize=14)
plt.show();
```



Hmmm... what's wrong with this plot?

Well, let's remind ourselves of what we're trying to do.

We're trying to use the historical price of Bitcoin to predict future prices of Bitcoin.

With this in mind, our seen data (training set) is what?

Prices of Bitcoin in the past.

And our unseen data (test set) is?

Prices of Bitcoin in the future.

Does the plot above reflect this?

No.

Our test data is scattered all throughout the training data.

This kind of random split is okay for datasets without a time component (such as images or passages of text for classification problems) but for time series, we've got to take the time factor into account.

To fix this, we've got to split our data in a way that reflects what we're actually trying to do.

We need to split our historical Bitcoin data to have a dataset that reflects the past (train set) and a dataset that reflects the future (test set).

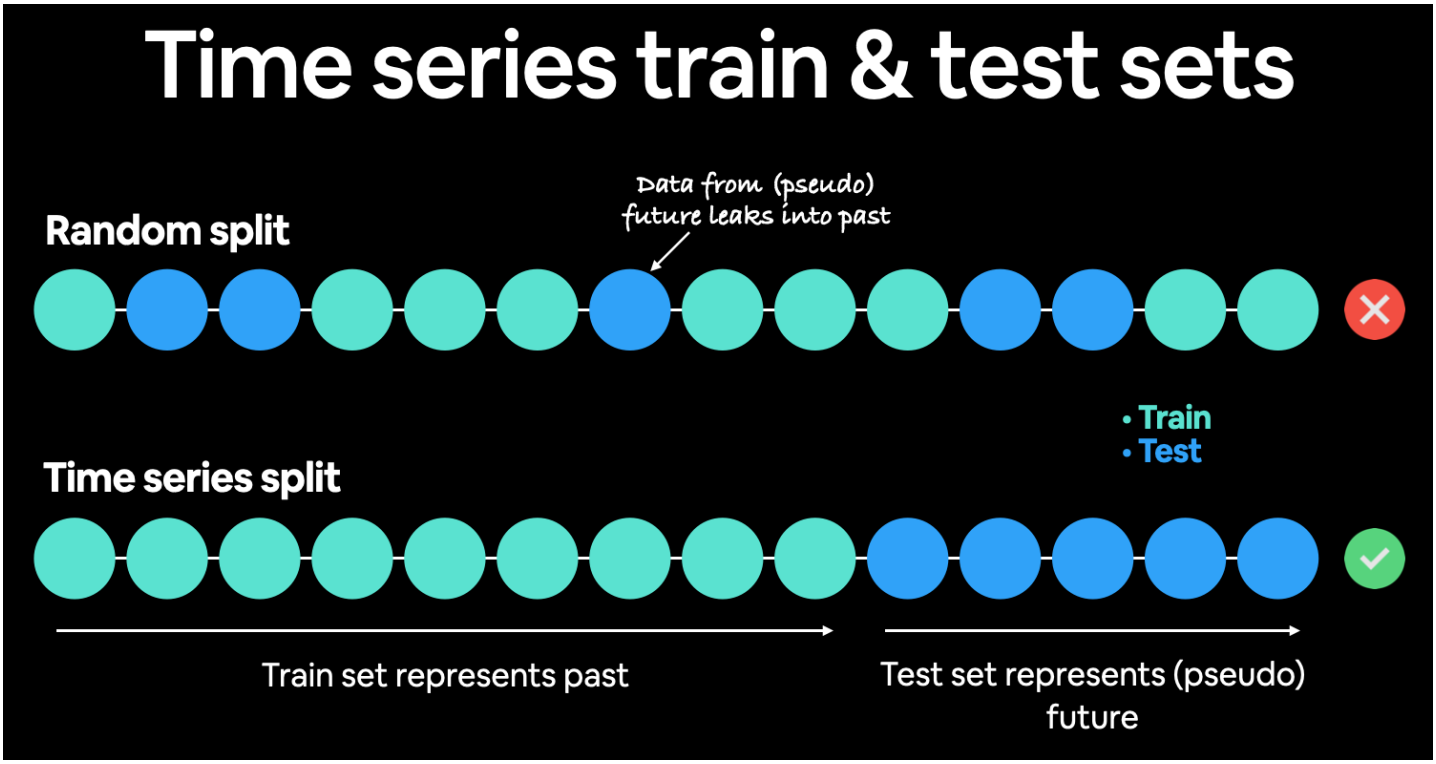
Create train & test sets for time series (the right way)

Of course, there's no way we can actually access data from the future.

But we can engineer our test set to be in the future with respect to the training set.

To do this, we can create an abitrary point in time to split our data.

Everything before the point in time can be considered the training set and everything after the point in time can be considered the test set.



Demonstration of time series split. Rather than a traditionaly random train/test split, it's best to split the time series data sequentially. Meaning, the test data should be data from the future when compared to the training data.

In [ ]:

```
# Create train and test splits the right way for time series data
```

```
split_size = int(0.8 * len(prices)) # 80% train, 20% test

# Create train data splits (everything before the split)
X_train, y_train = timesteps[:split_size], prices[:split_size]

# Create test data splits (everything after the split)
X_test, y_test = timesteps[split_size:], prices[split_size:]

len(X_train), len(X_test), len(y_train), len(y_test)
```

Out[ ]:

```
(2229, 558, 2229, 558)
```

Okay, looks like our custom made splits are the same lengths as the splits we made with `train_test_split`.

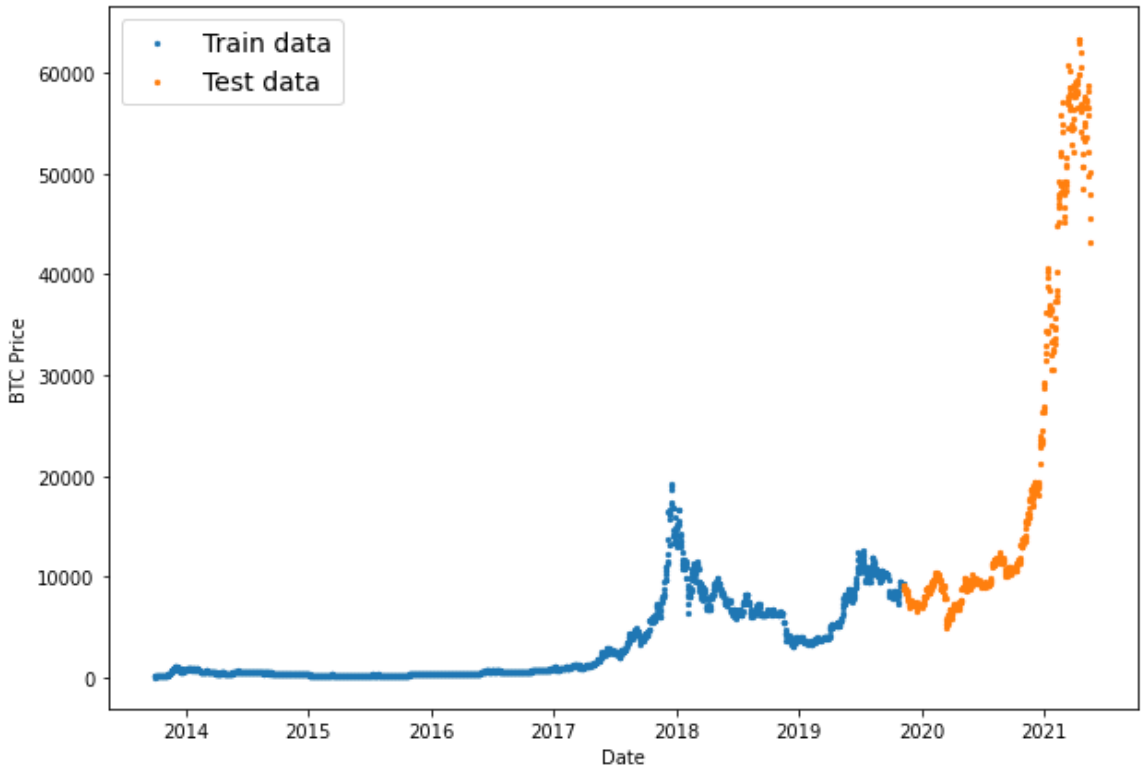
But again, these are numbers on a page.

And you know how the saying goes, trust one eye more than two ears.

Let's visualize.

In [ ]:

```
# Plot correctly made splits
plt.figure(figsize=(10, 7))
plt.scatter(X_train, y_train, s=5, label="Train data")
plt.scatter(X_test, y_test, s=5, label="Test data")
plt.xlabel("Date")
plt.ylabel("BTC Price")
plt.legend(fontsize=14)
plt.show();
```



That looks much better!

Do you see what's happened here?

We're going to be using the training set (past) to train a model to try and predict values on the test set (future).

Because the test set is an *artificial* future, we can guage how our model might perform on *actual* future data.

> **Note:** The amount of data you reserve for your test set not set in stone. You could have 80/20, 90/10, 95/5 splits or in some cases, you might not even have enough data to split into train and test sets (see the resource below). The point is to remember the test set is a pseudofuture and not

> **the actual future, it is only meant to give you an indication of how the models you're building are performing.**
>
> **Resource: Working with time series data can be tricky compared to other kinds of data. And there are a few pitfalls to watch out for, such as how much data to use for a test set. The article *3 facts about time series forecasting that surprise experienced machine learning practitioners* talks about different things to watch out for when working with time series data, I'd recommend reading it.**

Create a plotting function

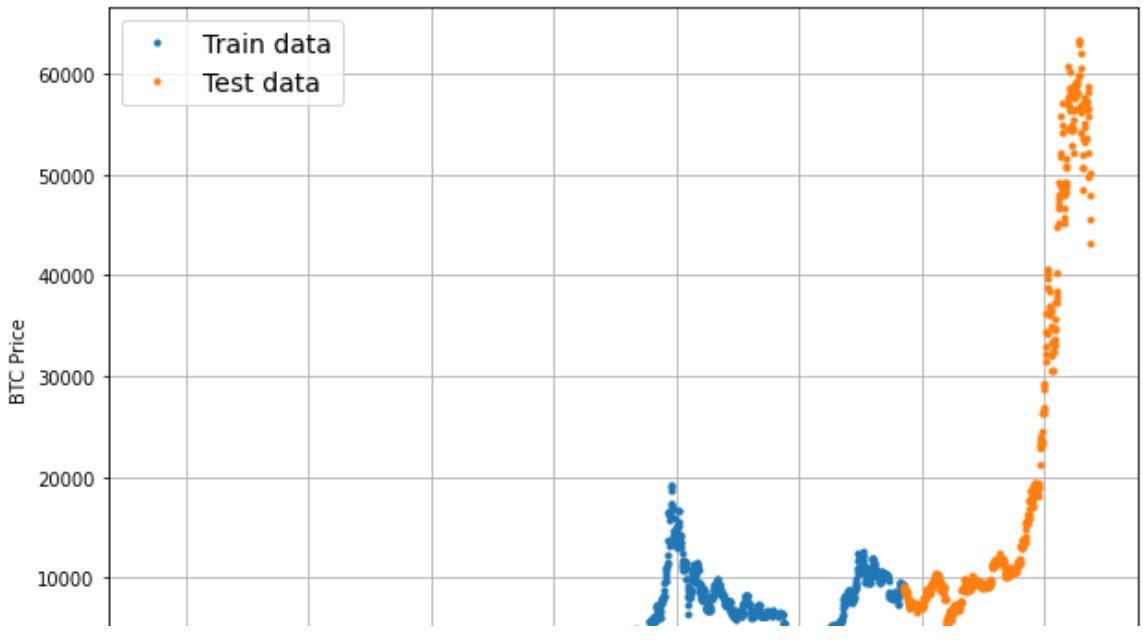
Rather than retyping `matplotlib` commands to continuously plot data, let's make a plotting function we can reuse later.

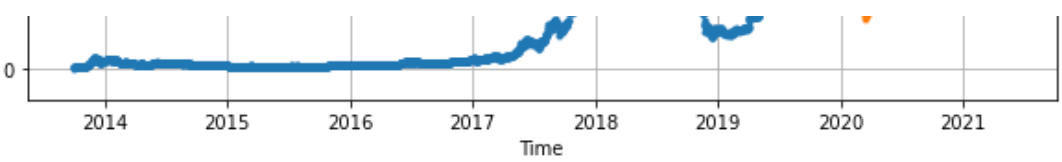
In [ ]:

```
# Create a function to plot time series data
def plot_time_series(timesteps, values, format='.', start=0, end=None, label=None):
  """
  Plots a timesteps (a series of points in time) against values (a series of values acros
s timesteps).

  Parameters
  ---------
  timesteps : array of timesteps
  values : array of values across time
  format : style of plot, default "."
  start : where to start the plot (setting a value will index from start of timesteps & v
alues)
  end : where to end the plot (setting a value will index from end of timesteps & values)
  label : label to show on plot of values
  """
  # Plot the series
  plt.plot(timesteps[start:end], values[start:end], format, label=label)
  plt.xlabel("Time")
  plt.ylabel("BTC Price")
  if label:
    plt.legend(fontsize=14) # make label bigger
  plt.grid(True)
```

In [ ]:

```
# Try out our plotting function
plt.figure(figsize=(10, 7))
plot_time_series(timesteps=X_train, values=y_train, label="Train data")
plot_time_series(timesteps=X_test, values=y_test, label="Test data")
```





Looking good!

Time for some modelling experiments.

Modelling Experiments

We can build almost any kind of model for our problem as long as the data inputs and outputs are formatted correctly.

However, just because we *can* build *almost any* kind of model, doesn't mean it'll perform well/should be used in a production setting.

We'll see what this means as we build and evaluate models throughout.

Before we discuss what modelling experiments we're going to run, there are two terms you should be familiar with, **horizon** and **window**.

- **horizon** = number of timesteps to predict into future
- **window** = number of timesteps from past used to predict **horizon**

For example, if we wanted to predict the price of Bitcoin for tomorrow (1 day in the future) using the previous week's worth of Bitcoin prices (7 days in the past), the horizon would be 1 and the window would be 7.

Now, how about those modelling experiments?

Model Number	Model Type	Horizon size	Window size	Extra data
0	Naïve model (baseline)	NA	NA	NA
1	Dense model	1	7	NA
2	Same as 1	1	30	NA
3	Same as 1	7	30	NA
4	Conv1D	1	7	NA
5	LSTM	1	7	NA
6	Same as 1 (but with multivariate data)	1	7	Block reward size
7	N-BEATs Algorithm	1	7	NA
8	Ensemble (multiple models optimized on different loss functions)	1	7	NA
9	Future prediction model (model to predict future values)	1	7	NA
10	Same as 1 (but with turkey data introduced)	1	7	NA

> **Note:** To reiterate, as you can see, we can build many types of models for the data we're working with. But that doesn't mean that they'll perform well. Deep learning is a powerful technique but it doesn't always work. And as always, start with a simple model first and then add complexity as needed.

Model 0: Naïve forecast (baseline)

As usual, let's start with a baseline.

One of the most common baseline models for time series forecasting, the naïve model (also called the naïve forecast), requires no training at all.

That's because all the naïve model does is use the previous timestep value to predict the next timestep value.

The formula looks like this:

$$\hat{y}_t = y_{t-1}$$

In English:

> **The prediction at timestep `t` (y-hat) is equal to the value at timestep `t-1` (the previous timestep).**

Sound simple?

Maybe not.

In an open system (like a stock market or crypto market), you'll often find beating the naïve forecast with *any* kind of model is quite hard.

> **Note: For the sake of this notebook, an open system is a system where inputs and outputs can freely flow, such as a market (stock or crypto). Where as, a closed system the inputs and outputs are contained within the system (like a poker game with your buddies, you know the buy in and you know how much the winner can get). Time series forecasting in open systems is generally quite poor.**

In [ ]:

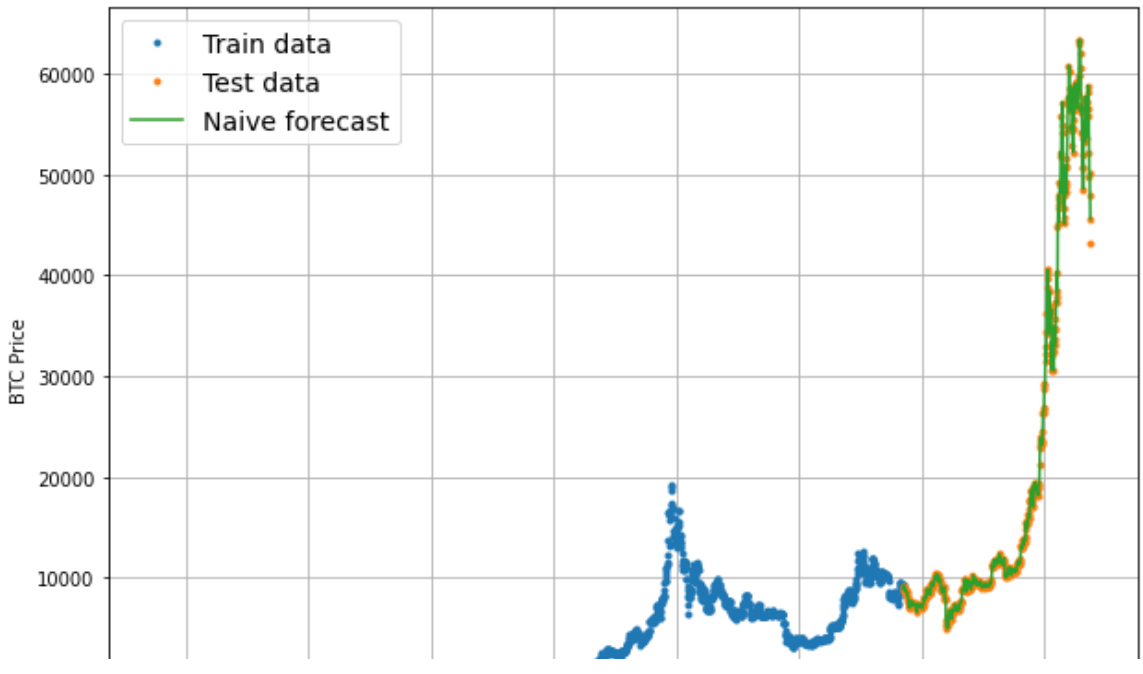
```
# Create a naïve forecast
naive_forecast = y_test[:-1] # Naïve forecast equals every value excluding the last valu
e
naive_forecast[:10], naive_forecast[-10:] # View frist 10 and last 10
```

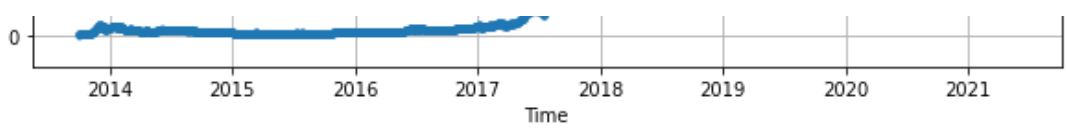
Out[ ]:

```
(array([9226.48582088, 8794.35864452, 8798.04205463, 9081.18687849,
        8711.53433917, 8760.89271814, 8749.52059102, 8656.97092235,
        8500.64355816, 8469.2608989 ]),
 array([57107.12067189, 58788.20967893, 58102.19142623, 55715.54665129,
        56573.5554719 , 52147.82118698, 49764.1320816 , 50032.69313676,
        47885.62525472, 45604.61575361]))
```

In [ ]:

```
# Plot naive forecast
plt.figure(figsize=(10, 7))
plot_time_series(timesteps=X_train, values=y_train, label="Train data")
plot_time_series(timesteps=X_test, values=y_test, label="Test data")
plot_time_series(timesteps=X_test[1:], values=naive_forecast, format="-", label="Naive f
orecast");
```





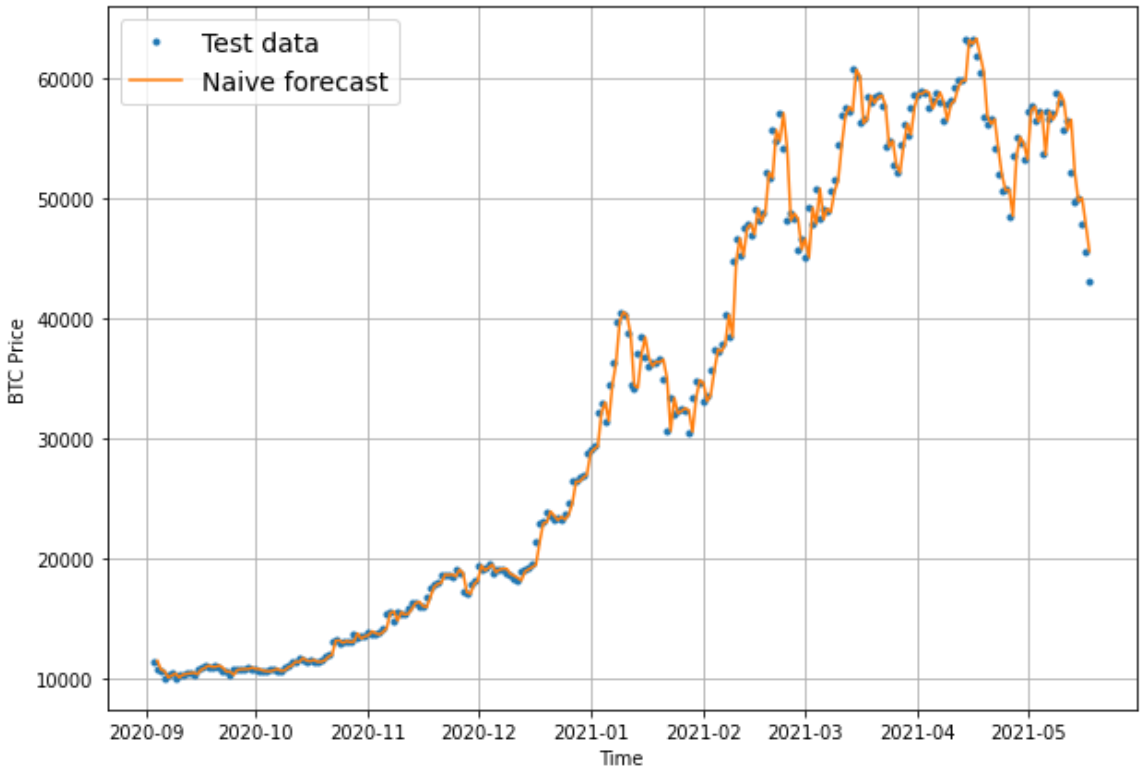
The naive forecast looks like it's following the data well.

Let's zoom in to take a better look.

We can do so by creating an offset value and passing it to the `start` parameter of our `plot_time_series()` function.

In [ ]:

```
plt.figure(figsize=(10, 7))
offset = 300 # offset the values by 300 timesteps
plot_time_series(timesteps=X_test, values=y_test, start=offset, label="Test data")
plot_time_series(timesteps=X_test[1:], values=naive_forecast, format="-", start=offset,
label="Naive forecast");
```



When we zoom in we see the naïve forecast comes slightly after the test data. This makes sense because the naive forecast uses the previous timestep value to predict the next timestep value.

Forecast made. Time to evaluate it.

Evaluating a time series model

Time series forecasting often involves predicting a number (in our case, the price of Bitcoin).

And what kind of problem is predicting a number?

Ten points if you said regression.

With this known, we can use regression evaluation metrics to evaluate our time series forecasts.

The main thing we will be evaluating is: how do our model's predictions (`y_pred`) compare against the actual values (`y_true` or *ground truth values*)?

> **Resource:** We're going to be using several metrics to evaluate our different model's time series forecast accuracy. Many of them are sourced and explained mathematically and conceptually in Forecasting: Principles and Practice chapter 5.8, I'd recommend reading through here for a more

> in-depth overview of what we're going to practice.

For all of the following metrics, **lower is better** (for example an MAE of 0 is better than an MAE 100).

Scale-dependent errors

These are metrics which can be used to compare time series values and forecasts that are on the same scale.

For example, Bitcoin historical prices in USD veresus Bitcoin forecast values in USD.

Metric	Details	Code
MAE (mean absolute error)	Easy to interpret (a forecast is X amount different from actual amount). Forecast methods which minimises the MAE will lead to forecasts of the median.	`tf.keras.metrics.mean_absolute_error()`
RMSE (root mean square error)	Forecasts which minimise the RMSE lead to forecasts of the mean.	`tf.sqrt( tf.keras.metrics.mean_square_error() )`

Percentage errors

Percentage errors do not have units, this means they can be used to compare forecasts across different datasets.

Metric	Details	Code
MAPE (mean absolute percentage error)	Most commonly used percentage error. May explode (not work) if `y=0`.	`tf.keras.metrics.mean_absolute_percentage_error()`
sMAPE (symmetric mean absolute percentage error)	Recommended not to be used by Forecasting: Principles and Practice, though it is used in forecasting competitions.	Custom implementation

Scaled errors

Scaled errors are an alternative to percentage errors when comparing forecast performance across different time series.

Metric	Details	Code
MASE (mean absolute scaled error).	MASE equals one for the naive forecast (or very close to one). A forecast which performs better than the naïve should get <1 MASE.	See sktime's `mase_loss()`

> **Question:** There are so many metrics... which one should I pay most attention to? It's going to depend on your problem. However, since its ease of interpretation (you can explain it in a sentence to your grandma), MAE is often a very good place to start.

Since we're going to be evaluing a lot of models, let's write a function to help us calculate evaluation metrics on their forecasts.

First we'll need TensorFlow.

In [ ]:

```
# Let's get TensorFlow!
import tensorflow as tf
```

And since TensorFlow doesn't have a ready made version of MASE (mean aboslute scaled error), how about we create our own?

We'll take inspiration from sktime's (Scikit-Learn for time series) `MeanAbsoluteScaledError` class which calculates the MASE.

In [ ]:

```
# MASE implemented courtesy of sktime - https://github.com/alan-turing-institute/sktime/blob/ee7a06843a44f4aaec7582d847e36073a9ab0566/sktime/performance_metrics/forecasting/_functions.py#L16
def mean_absolute_scaled_error(y_true, y_pred):
  """
  Implement MASE (assuming no seasonality of data).
  """
  mae = tf.reduce_mean(tf.abs(y_true - y_pred))

  # Find MAE of naive forecast (no seasonality)
  mae_naive_no_season = tf.reduce_mean(tf.abs(y_true[1:] - y_true[:-1])) # our seasonality is 1 day (hence the shifting of 1 day)

  return mae / mae_naive_no_season
```

You'll notice the version of MASE above doesn't take in the training values like sktime's `mae_loss()`. In our case, we're comparing the MAE of our predictions on the test to the MAE of the naïve forecast on the test set.

In practice, if we've created the function correctly, the naïve model should achieve an MASE of 1 (or very close to 1). Any model worse than the naïve forecast will achieve an MASE of >1 and any model better than the naïve forecast will achieve an MASE of <1.

Let's put each of our different evaluation metrics together into a function.

In [ ]:

```
def evaluate_preds(y_true, y_pred):
  # Make sure float32 (for metric calculations)
  y_true = tf.cast(y_true, dtype=tf.float32)
  y_pred = tf.cast(y_pred, dtype=tf.float32)

  # Calculate various metrics
  mae = tf.keras.metrics.mean_absolute_error(y_true, y_pred)
  mse = tf.keras.metrics.mean_squared_error(y_true, y_pred) # puts and emphasis on outliers (all errors get squared)
  rmse = tf.sqrt(mse)
  mape = tf.keras.metrics.mean_absolute_percentage_error(y_true, y_pred)
  mase = mean_absolute_scaled_error(y_true, y_pred)

  return {"mae": mae.numpy(),
          "mse": mse.numpy(),
          "rmse": rmse.numpy(),
          "mape": mape.numpy(),
          "mase": mase.numpy()}
```

Looking good! How about we test our function on the naive forecast?

In [ ]:

```
naive_results = evaluate_preds(y_true=y_test[1:],
                               y_pred=naive_forecast)
naive_results
```

Out[ ]:

```
{'mae': 567.9802,
 'mape': 2.516525,
 'mase': 0.99957,
 'mse': 1147547.0,
 'rmse': 1071.2362}
```

Alright, looks like we've got some baselines to beat.

Taking a look at the naïve forecast's MAE, it seems on average each forecast is ~$567 different than the actual Bitcoin price.

How does this compare to the average price of Bitcoin in the test dataset?

In [ ]:

```
# Find average price of Bitcoin in test dataset
tf.reduce_mean(y_test).numpy()
```

Out[ ]:

```
20056.632963737226
```

Okay, looking at these two values is starting to give us an idea of how our model is performing:

- **The average price of Bitcoin in the test dataset is: $20,056 (note: average may not be the best measure here, since the highest price is over 3x this value and the lowest price is over 4x lower)**
- **Each prediction in naive forecast is on average off by: $567**

Is this enough to say it's a good model?

That's up your own interpretation. Personally, I'd prefer a model which was closer to the mark.

How about we try and build one?

Other kinds of time series forecasting models which can be used for baselines and actual forecasts

Since we've got a naïve forecast baseline to work with, it's time we start building models to try and beat it.

And because this course is focused on TensorFlow and deep learning, we're going to be using TensorFlow to build deep learning models to try and improve on our naïve forecasting results.

That being said, there are many other kinds of models you may want to look into for building baselines/performing forecasts.

Some of them may even beat our best performing models in this notebook, however, I'll leave trying them out for extra-curriculum.

Model/Library Name	Resource
Moving average	https://machinelearningmastery.com/moving-average-smoothing-for-time-series-forecasting-python/
ARIMA (Autoregression Integrated Moving Average)	https://machinelearningmastery.com/arima-for-time-series-forecasting-with-python/
sktime (Scikit-Learn for time series)	https://github.com/alan-turing-institute/sktime
TensorFlow Decision Forests (random forest, gradient boosting trees)	https://www.tensorflow.org/decision_forests
Facebook Kats (purpose-built forecasting and time series analysis library by Facebook)	https://github.com/facebookresearch/Kats
LinkedIn Greykite (flexible, intuitive and fast forecasts)	https://github.com/linkedin/greykite

Format Data Part 2: Windowing dataset

Surely we'd be ready to start building models by now?

We're so close! Only one more step (really two) to go.

We've got to window our time series.

Why do we window?

Windowing is a method to turn a time series dataset into supervised learning problem.

In other words, we want to use windows of the past to predict the future.

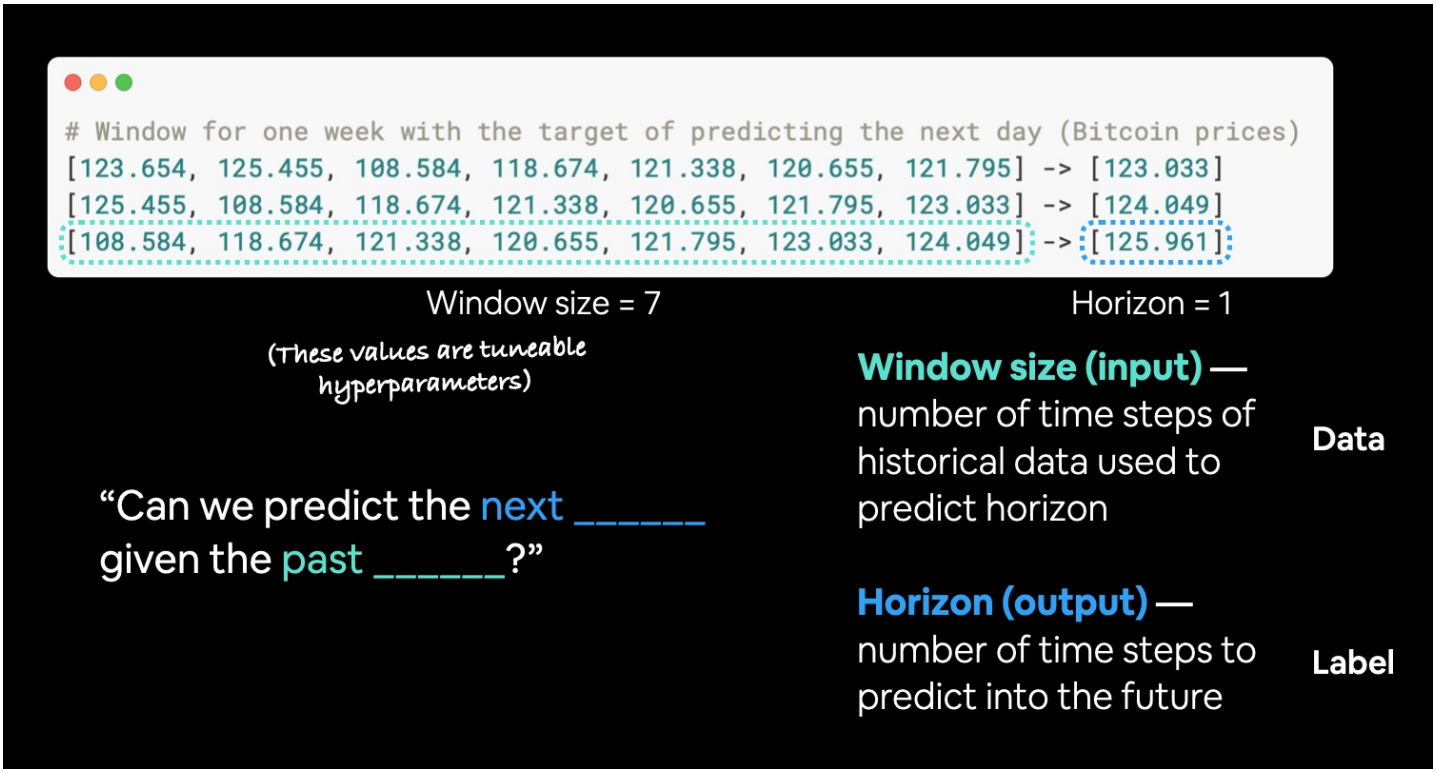
For example for a univariate time series, windowing for one week (`window=7`) to predict the next single value (`horizon=1`) might look like:

(`horizon=1`) might look like:

```
Window for one week (univariate time series)

[0, 1, 2, 3, 4, 5, 6] -> [7]
[1, 2, 3, 4, 5, 6, 7] -> [8]
[2, 3, 4, 5, 6, 7, 8] -> [9]
```

Or for the price of Bitcoin, it'd look like:

```
Window for one week with the target of predicting the next day (Bitcoin prices)

[123.654, 125.455, 108.584, 118.674, 121.338, 120.655, 121.795] -> [123.033]
[125.455, 108.584, 118.674, 121.338, 120.655, 121.795, 123.033] -> [124.049]
[108.584, 118.674, 121.338, 120.655, 121.795, 123.033, 124.049] -> [125.961]
```



Example of windows and horizons for Bitcoin data. Windowing can be used to turn time series data into a supervised learning problem.

Let's build some functions which take in a univariate time series and turn it into windows and horizons of specified sizes.

We'll start with the default horizon size of 1 and a window size of 7 (these aren't necessarily the best values to use, I've just picked them).

In [ ]:

```
HORIZON = 1 # predict 1 step at a time
WINDOW_SIZE = 7 # use a week worth of timesteps to predict the horizon
```

Now we'll write a function to take in an array and turn it into a window and horizon.

In [ ]:

```
# Create function to label windowed data
def get_labelled_windows(x, horizon=1):
  """
  Creates labels for windowed dataset.

  E.g. if horizon=1 (default)
  Input: [1, 2, 3, 4, 5, 6] -> Output: ([1, 2, 3, 4, 5], [6])
  """
  return x[:, :-horizon], x[:, -horizon:]
```

In [ ]:

```
# Test out the window labelling function
test_window, test_label = get_labelled_windows(tf.expand_dims(tf.range(8)+1, axis=0), ho
rizon=HORIZON)
print(f"Window: {tf.squeeze(test_window).numpy()} -> Label: {tf.squeeze(test_label).numpy
()}")
```

```
Window: [1 2 3 4 5 6 7] -> Label: 8
```

Oh yeah, that's what I'm talking about!

Now we need a way to make windows for an entire time series.

We could do this with Python for loops, however, for large time series, that'd be quite slow.

To speed things up, we'll leverage NumPy's array indexing.

Let's write a function which:

1. **Creates a window step of specific window size, for example:** `[[0, 1, 2, 3, 4, 5, 6, 7]]`
2. **Uses NumPy indexing to create a 2D of multiple window steps, for example:**

```
[[0, 1, 2, 3, 4, 5, 6, 7],
[1, 2, 3, 4, 5, 6, 7, 8],
[2, 3, 4, 5, 6, 7, 8, 9]]
```

3. **Uses the 2D array of multuple window steps to index on a target series**
4. **Uses the `get_labelled_windows()` function we created above to turn the window steps into windows with a specified horizon**

> **Resource:** The function created below has been adapted from Syafiq Kamarul Azman's article *Fast and Robust Sliding Window Vectorization with NumPy*.

In [ ]:

```
# Create function to view NumPy arrays as windows
def make_windows(x, window_size=7, horizon=1):
  """
  Turns a 1D array into a 2D array of sequential windows of window_size.
  """
  # 1. Create a window of specific window_size (add the horizon on the end for later labe
lling)
  window_step = np.expand_dims(np.arange(window_size+horizon), axis=0)
  # print(f"Window step:\n {window_step}")

  # 2. Create a 2D array of multiple window steps (minus 1 to account for 0 indexing)
  window_indexes = window_step + np.expand_dims(np.arange(len(x)-(window_size+horizon-1)
), axis=0).T # create 2D array of windows of size window_size
  # print(f"Window indexes:\n {window_indexes[:3], window_indexes[-3:], window_indexes.sh
ape}")

  # 3. Index on the target array (time series) with 2D array of multiple window steps
  windowed_array = x[window_indexes]

  # 4. Get the labelled windows
  windows, labels = get_labelled_windows(windowed_array, horizon=horizon)

  return windows, labels
```

Phew! A few steps there... let's see how it goes.

In [ ]:

```
full_windows, full_labels = make_windows(prices, window_size=WINDOW_SIZE, horizon=HORIZO
N)
len(full_windows), len(full_labels)
```

Out[ ]:

```
(2780, 2780)
```

Of course we have to visualize, visualize, visualize!

In [ ]:

```
# View the first 3 windows/labels
for i in range(3):
  print(f"Window: {full_windows[i]} -> Label: {full_labels[i]}")
```

```
Window: [123.65499 125.455   108.58483 118.67466 121.33866 120.65533 121.795  ] -> Label:
[123.033]
Window: [125.455   108.58483 118.67466 121.33866 120.65533 121.795   123.033  ] -> Label:
[124.049]
Window: [108.58483 118.67466 121.33866 120.65533 121.795   123.033   124.049  ] -> Label:
[125.96116]
```

In [ ]:

```
# View the last 3 windows/labels
for i in range(3):
  print(f"Window: {full_windows[i-3]} -> Label: {full_labels[i-3]}")
```

```
Window: [58788.20967893 58102.19142623 55715.54665129 56573.5554719
 52147.82118698 49764.1320816  50032.69313676] -> Label: [47885.62525472]
Window: [58102.19142623 55715.54665129 56573.5554719  52147.82118698
 49764.1320816  50032.69313676 47885.62525472] -> Label: [45604.61575361]
Window: [55715.54665129 56573.5554719  52147.82118698 49764.1320816
 50032.69313676 47885.62525472 45604.61575361] -> Label: [43144.47129086]
```

> **Note:** You can find a function which achieves similar results to the ones we implemented above at `tf.keras.preprocessing.timeseries_dataset_from_array()`. Just like ours, it takes in an array and returns a windowed dataset. It has the benefit of returning data in the form of a tf.data.Dataset instance (we'll see how to do this with our own data later).

Turning windows into training and test sets

Look how good those windows look! Almost like the stain glass windows on the Sistine Chapel, well, maybe not that good but still.

Time to turn our windows into training and test splits.

We could've windowed our existing training and test splits, however, with the nature of windowing (windowing often requires an offset at some point in the data), it usually works better to window the data first, then split it into training and test sets.

Let's write a function which takes in full sets of windows and their labels and splits them into train and test splits.

In [ ]:

```
# Make the train/test splits
def make_train_test_splits(windows, labels, test_split=0.2):
  """
  Splits matching pairs of windows and labels into train and test splits.
  """
  split_size = int(len(windows) * (1-test_split)) # this will default to 80% train/20% te
st
  train_windows = windows[:split_size]
  train_labels = labels[:split_size]
  test_windows = windows[split_size:]
  test_labels = labels[split_size:]
  return train_windows, test_windows, train_labels, test_labels
```

Look at that amazing function, lets test it.

In [ ]:

```
train_windows, test_windows, train_labels, test_labels = make_train_test_splits(full_windows, full_labels)
len(train_windows), len(test_windows), len(train_labels), len(test_labels)
```

Out[ ]:

```
(2224, 556, 2224, 556)
```

Notice the default split of 80% training data and 20% testing data (this split can be adjusted if needed).

How do the first 5 samples of the training windows and labels looks?

In [ ]:

```
train_windows[:5], train_labels[:5]
```

Out[ ]:

```
(array([[123.65499, 125.455  , 108.58483, 118.67466, 121.33866, 120.65533,
         121.795  ],
        [125.455  , 108.58483, 118.67466, 121.33866, 120.65533, 121.795  ,
         123.033  ],
        [108.58483, 118.67466, 121.33866, 120.65533, 121.795  , 123.033  ,
         124.049  ],
        [118.67466, 121.33866, 120.65533, 121.795  , 123.033  , 124.049  ,
         125.96116],
        [121.33866, 120.65533, 121.795  , 123.033  , 124.049  , 125.96116,
         125.27966]]), array([[123.033  ],
        [124.049  ],
        [125.96116],
        [125.27966],
        [125.9275 ]]))
```

In [ ]:

```
# Check to see if same (accounting for horizon and window size)
np.array_equal(np.squeeze(train_labels[:-HORIZON-1]), y_train[WINDOW_SIZE:])
```

Out[ ]:

```
True
```

Make a modelling checkpoint

We're so close to building models. So so so close.

Because our model's performance will fluctuate from experiment to experiment, we'll want to make sure we're comparing apples to apples.

What I mean by this is in order for a fair comparison, we want to compare each model's best performance against each model's best performance.

For example, if `model_1` performed incredibly well on epoch 55 but its performance fell off toward epoch 100, we want the version of the model from epoch 55 to compare to other models rather than the version of the model from epoch 100.

And the same goes for each of our other models: compare the best against the best.

To take of this, we'll implement a `ModelCheckpoint` callback.

The `ModelCheckpoint` callback will monitor our model's performance during training and save the best model to file by setting `save_best_only=True`.

That way when evaluating our model we could restore its best performing configuration from file.

> 🔑 **Note:** Because of the size of the dataset (smaller than usual), you'll notice our modelling experiment results fluctuate quite a bit during training (hence the implementation of the `ModelCheckpoint` callback to save the best model).

Because we're going to be running multiple experiments, it makes sense to keep track of them by saving models to file under different names.

To do this, we'll write a small function to create a `ModelCheckpoint` callback which saves a model to specified filename.

In [ ]:

```
import os

# Create a function to implement a ModelCheckpoint callback with a specific filename
def create_model_checkpoint(model_name, save_path="model_experiments"):
  return tf.keras.callbacks.ModelCheckpoint(filepath=os.path.join(save_path, model_name)
, # create filepath to save model
                                            verbose=0, # only output a limited amount o
f text
                                            save_best_only=True) # save only the best mo
del to file
```

Model 1: Dense model (window = 7, horizon = 1)

Finally!

Time to build one of our models.

If you think we've been through a fair bit of preprocessing before getting here, you're right.

Often, preparing data for a model is one of the largest parts of any machine learning project.

And once you've got a good model in place, you'll probably notice far more improvements from manipulating the data (e.g. collecting more, improving the quality) than manipulating the model.

We're going to start by keeping it simple, `model_1` will have:

- A single dense layer with 128 hidden units and ReLU (rectified linear unit) activation
- An output layer with linear activation (or no activation)
- Adam optimizer and MAE loss function
- Batch size of 128
- 100 epochs

Why these values?

I picked them out of experimentation.

A batch size of 32 works pretty well too and we could always train for less epochs but since the model runs so fast (you'll see in a second, it's because the number of samples we have isn't massive) we might as well train for more.

> 🔑 **Note:** As always, many of the values for machine learning problems are experimental. A reminder that the values you can set yourself in a machine learning algorithm (the hidden units, the batch size, horizon size, window size) are called **hyperparameters**. And experimenting to find the best values for hyperparameters is called **hyperparameter tuning**. Where as parameters learned by a model itself (patterns in the data, formally called weights & biases) are referred to as **parameters**.

Let's import TensorFlow and build our first deep learning model for time series.

In [ ]:

```
import tensorflow as tf
```

```
from tensorflow.keras import layers

# Set random seed for as reproducible results as possible
tf.random.set_seed(42)

# Construct model
model_1 = tf.keras.Sequential([
  layers.Dense(128, activation="relu"),
  layers.Dense(HORIZON, activation="linear") # linear activation is the same as having no activation
], name="model_1_dense") # give the model a name so we can save it

# Compile model
model_1.compile(loss="mae",
                optimizer=tf.keras.optimizers.Adam(),
                metrics=["mae"]) # we don't necessarily need this when the loss function is already MAE

# Fit model
model_1.fit(x=train_windows, # train windows of 7 timesteps of Bitcoin prices
            y=train_labels, # horizon value of 1 (using the previous 7 timesteps to predict next day)
            epochs=100,
            verbose=1,
            batch_size=128,
            validation_data=(test_windows, test_labels),
            callbacks=[create_model_checkpoint(model_name=model_1.name)]) # create Model Checkpoint callback to save best model
```

```
Epoch 1/100
18/18 [==============================] - 3s 12ms/step - loss: 780.3455 - mae: 780.3455 - val_loss: 2279.6526 - val_mae: 2279.6526
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 2/100
18/18 [==============================] - 0s 4ms/step - loss: 247.6756 - mae: 247.6756 - val_loss: 1005.9991 - val_mae: 1005.9991
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 3/100
18/18 [==============================] - 0s 4ms/step - loss: 188.4116 - mae: 188.4116 - val_loss: 923.2862 - val_mae: 923.2862
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 4/100
18/18 [==============================] - 0s 4ms/step - loss: 169.4340 - mae: 169.4340 - val_loss: 900.5872 - val_mae: 900.5872
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 5/100
18/18 [==============================] - 0s 4ms/step - loss: 165.0895 - mae: 165.0895 - val_loss: 895.2238 - val_mae: 895.2238
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 6/100
18/18 [==============================] - 0s 4ms/step - loss: 158.5210 - mae: 158.5210 - val_loss: 855.1982 - val_mae: 855.1982
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 7/100
18/18 [==============================] - 0s 5ms/step - loss: 151.3566 - mae: 151.3566 - val_loss: 840.9168 - val_mae: 840.9168
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 8/100
18/18 [==============================] - 0s 5ms/step - loss: 145.2560 - mae: 145.2560 - val_loss: 803.5956 - val_mae: 803.5956
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 9/100
18/18 [==============================] - 0s 4ms/step - loss: 144.3546 - mae: 144.3546 - val_loss: 799.5455 - val_mae: 799.5455
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 10/100
18/18 [==============================] - 0s 4ms/step - loss: 141.2943 - mae: 141.2943 - val_loss: 763.5010 - val_mae: 763.5010
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 11/100
18/18 [==============================] - 0s 4ms/step - loss: 135.6595 - mae: 135.6595 - val_loss: 771.3357 - val_mae: 771.3357
```

```
Epoch 12/100
18/18 [==============================] - 0s 5ms/step - loss: 134.1700 - mae: 134.1700 - v
al_loss: 782.8079 - val_mae: 782.8079
Epoch 13/100
18/18 [==============================] - 0s 4ms/step - loss: 134.6015 - mae: 134.6015 - v
al_loss: 784.4449 - val_mae: 784.4449
Epoch 14/100
18/18 [==============================] - 0s 4ms/step - loss: 130.6127 - mae: 130.6127 - v
al_loss: 751.3234 - val_mae: 751.3234
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 15/100
18/18 [==============================] - 0s 5ms/step - loss: 128.8347 - mae: 128.8347 - v
al_loss: 696.5757 - val_mae: 696.5757
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 16/100
18/18 [==============================] - 0s 5ms/step - loss: 124.7739 - mae: 124.7739 - v
al_loss: 702.4698 - val_mae: 702.4698
Epoch 17/100
18/18 [==============================] - 0s 4ms/step - loss: 123.4474 - mae: 123.4474 - v
al_loss: 704.9241 - val_mae: 704.9241
Epoch 18/100
18/18 [==============================] - 0s 5ms/step - loss: 122.2105 - mae: 122.2105 - v
al_loss: 667.9725 - val_mae: 667.9725
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 19/100
18/18 [==============================] - 0s 4ms/step - loss: 121.7263 - mae: 121.7263 - v
al_loss: 718.8797 - val_mae: 718.8797
Epoch 20/100
18/18 [==============================] - 0s 4ms/step - loss: 119.2420 - mae: 119.2420 - v
al_loss: 657.0667 - val_mae: 657.0667
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 21/100
18/18 [==============================] - 0s 4ms/step - loss: 121.2275 - mae: 121.2275 - v
al_loss: 637.0330 - val_mae: 637.0330
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 22/100
18/18 [==============================] - 0s 4ms/step - loss: 119.9544 - mae: 119.9544 - v
al_loss: 671.2490 - val_mae: 671.2490
Epoch 23/100
18/18 [==============================] - 0s 5ms/step - loss: 121.9248 - mae: 121.9248 - v
al_loss: 633.3593 - val_mae: 633.3593
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 24/100
18/18 [==============================] - 0s 5ms/step - loss: 116.3666 - mae: 116.3666 - v
al_loss: 624.4852 - val_mae: 624.4852
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 25/100
18/18 [==============================] - 0s 4ms/step - loss: 114.6816 - mae: 114.6816 - v
al_loss: 619.7571 - val_mae: 619.7571
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 26/100
18/18 [==============================] - 0s 4ms/step - loss: 116.4455 - mae: 116.4455 - v
al_loss: 615.6364 - val_mae: 615.6364
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 27/100
18/18 [==============================] - 0s 5ms/step - loss: 116.5868 - mae: 116.5868 - v
al_loss: 615.9631 - val_mae: 615.9631
Epoch 28/100
18/18 [==============================] - 0s 4ms/step - loss: 113.4691 - mae: 113.4691 - v
al_loss: 608.0920 - val_mae: 608.0920
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 29/100
18/18 [==============================] - 0s 5ms/step - loss: 113.7598 - mae: 113.7598 - v
al_loss: 621.9306 - val_mae: 621.9306
Epoch 30/100
18/18 [==============================] - 0s 4ms/step - loss: 116.8613 - mae: 116.8613 - v
al_loss: 604.4056 - val_mae: 604.4056
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 31/100
18/18 [==============================] - 0s 4ms/step - loss: 111.9375 - mae: 111.9375 - v
al_loss: 609.3882 - val_mae: 609.3882
Epoch 32/100
```

```
18/18 [==============================] - 0s 4ms/step - loss: 112.4175 - mae: 112.4175 - v
al_loss: 603.0588 - val_mae: 603.0588
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 33/100
18/18 [==============================] - 0s 4ms/step - loss: 112.6697 - mae: 112.6697 - v
al_loss: 645.6975 - val_mae: 645.6975
Epoch 34/100
18/18 [==============================] - 0s 4ms/step - loss: 111.9867 - mae: 111.9867 - v
al_loss: 604.7632 - val_mae: 604.7632
Epoch 35/100
18/18 [==============================] - 0s 4ms/step - loss: 110.9451 - mae: 110.9451 - v
al_loss: 593.4648 - val_mae: 593.4648
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 36/100
18/18 [==============================] - 0s 5ms/step - loss: 114.4816 - mae: 114.4816 - v
al_loss: 608.0073 - val_mae: 608.0073
Epoch 37/100
18/18 [==============================] - 0s 4ms/step - loss: 110.2017 - mae: 110.2017 - v
al_loss: 597.2309 - val_mae: 597.2309
Epoch 38/100
18/18 [==============================] - 0s 5ms/step - loss: 112.2372 - mae: 112.2372 - v
al_loss: 637.9797 - val_mae: 637.9797
Epoch 39/100
18/18 [==============================] - 0s 4ms/step - loss: 115.1289 - mae: 115.1289 - v
al_loss: 587.4679 - val_mae: 587.4679
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 40/100
18/18 [==============================] - 0s 5ms/step - loss: 110.0854 - mae: 110.0854 - v
al_loss: 592.7117 - val_mae: 592.7117
Epoch 41/100
18/18 [==============================] - 0s 4ms/step - loss: 110.6343 - mae: 110.6343 - v
al_loss: 593.8997 - val_mae: 593.8997
Epoch 42/100
18/18 [==============================] - 0s 4ms/step - loss: 113.5762 - mae: 113.5762 - v
al_loss: 636.3674 - val_mae: 636.3674
Epoch 43/100
18/18 [==============================] - 0s 5ms/step - loss: 116.2286 - mae: 116.2286 - v
al_loss: 662.9264 - val_mae: 662.9264
Epoch 44/100
18/18 [==============================] - 0s 4ms/step - loss: 120.0192 - mae: 120.0192 - v
al_loss: 635.6360 - val_mae: 635.6360
Epoch 45/100
18/18 [==============================] - 0s 4ms/step - loss: 110.9675 - mae: 110.9675 - v
al_loss: 601.9926 - val_mae: 601.9926
Epoch 46/100
18/18 [==============================] - 0s 4ms/step - loss: 111.6012 - mae: 111.6012 - v
al_loss: 593.3531 - val_mae: 593.3531
Epoch 47/100
18/18 [==============================] - 0s 6ms/step - loss: 109.6161 - mae: 109.6161 - v
al_loss: 637.0014 - val_mae: 637.0014
Epoch 48/100
18/18 [==============================] - 0s 5ms/step - loss: 109.1368 - mae: 109.1368 - v
al_loss: 598.4199 - val_mae: 598.4199
Epoch 49/100
18/18 [==============================] - 0s 5ms/step - loss: 112.4355 - mae: 112.4355 - v
al_loss: 579.7040 - val_mae: 579.7040
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 50/100
18/18 [==============================] - 0s 4ms/step - loss: 110.2108 - mae: 110.2108 - v
al_loss: 639.2326 - val_mae: 639.2326
Epoch 51/100
18/18 [==============================] - 0s 5ms/step - loss: 111.0958 - mae: 111.0958 - v
al_loss: 597.3575 - val_mae: 597.3575
Epoch 52/100
18/18 [==============================] - 0s 4ms/step - loss: 110.7351 - mae: 110.7351 - v
al_loss: 580.7227 - val_mae: 580.7227
Epoch 53/100
18/18 [==============================] - 0s 5ms/step - loss: 111.1785 - mae: 111.1785 - v
al_loss: 648.3588 - val_mae: 648.3588
Epoch 54/100
18/18 [==============================] - 0s 4ms/step - loss: 114.0832 - mae: 114.0832 - v
al_loss: 593.2007 - val_mae: 593.2007
```

```
Epoch 55/100
18/18 [==============================] - 0s 4ms/step - loss: 110.4910 - mae: 110.4910 - v
al_loss: 579.5065 - val_mae: 579.5065
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 56/100
18/18 [==============================] - 0s 5ms/step - loss: 108.0489 - mae: 108.0489 - v
al_loss: 807.3851 - val_mae: 807.3851
Epoch 57/100
18/18 [==============================] - 0s 4ms/step - loss: 125.0614 - mae: 125.0614 - v
al_loss: 674.1654 - val_mae: 674.1654
Epoch 58/100
18/18 [==============================] - 0s 4ms/step - loss: 115.4340 - mae: 115.4340 - v
al_loss: 582.2698 - val_mae: 582.2698
Epoch 59/100
18/18 [==============================] - 0s 5ms/step - loss: 110.0881 - mae: 110.0881 - v
al_loss: 606.7637 - val_mae: 606.7637
Epoch 60/100
18/18 [==============================] - 0s 4ms/step - loss: 108.7156 - mae: 108.7156 - v
al_loss: 602.3102 - val_mae: 602.3102
Epoch 61/100
18/18 [==============================] - 0s 4ms/step - loss: 108.1525 - mae: 108.1525 - v
al_loss: 573.9990 - val_mae: 573.9990
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 62/100
18/18 [==============================] - 0s 4ms/step - loss: 107.3727 - mae: 107.3727 - v
al_loss: 581.7012 - val_mae: 581.7012
Epoch 63/100
18/18 [==============================] - 0s 4ms/step - loss: 110.7667 - mae: 110.7667 - v
al_loss: 637.5252 - val_mae: 637.5252
Epoch 64/100
18/18 [==============================] - 0s 4ms/step - loss: 110.1539 - mae: 110.1539 - v
al_loss: 586.6601 - val_mae: 586.6601
Epoch 65/100
18/18 [==============================] - 0s 5ms/step - loss: 108.2325 - mae: 108.2325 - v
al_loss: 573.5620 - val_mae: 573.5620
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 66/100
18/18 [==============================] - 0s 5ms/step - loss: 108.6825 - mae: 108.6825 - v
al_loss: 572.2206 - val_mae: 572.2206
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 67/100
18/18 [==============================] - 0s 4ms/step - loss: 106.6371 - mae: 106.6371 - v
al_loss: 646.6349 - val_mae: 646.6349
Epoch 68/100
18/18 [==============================] - 0s 4ms/step - loss: 114.1603 - mae: 114.1603 - v
al_loss: 681.8561 - val_mae: 681.8561
Epoch 69/100
18/18 [==============================] - 0s 5ms/step - loss: 124.5514 - mae: 124.5514 - v
al_loss: 655.9885 - val_mae: 655.9885
Epoch 70/100
18/18 [==============================] - 0s 4ms/step - loss: 125.0235 - mae: 125.0235 - v
al_loss: 601.0032 - val_mae: 601.0032
Epoch 71/100
18/18 [==============================] - 0s 6ms/step - loss: 110.3652 - mae: 110.3652 - v
al_loss: 595.3962 - val_mae: 595.3962
Epoch 72/100
18/18 [==============================] - 0s 5ms/step - loss: 107.9285 - mae: 107.9285 - v
al_loss: 573.7085 - val_mae: 573.7085
Epoch 73/100
18/18 [==============================] - 0s 4ms/step - loss: 109.5085 - mae: 109.5085 - v
al_loss: 580.4180 - val_mae: 580.4180
Epoch 74/100
18/18 [==============================] - 0s 4ms/step - loss: 108.7380 - mae: 108.7380 - v
al_loss: 576.1211 - val_mae: 576.1211
Epoch 75/100
18/18 [==============================] - 0s 5ms/step - loss: 107.9404 - mae: 107.9404 - v
al_loss: 591.1477 - val_mae: 591.1477
Epoch 76/100
18/18 [==============================] - 0s 5ms/step - loss: 109.4232 - mae: 109.4232 - v
al_loss: 597.8605 - val_mae: 597.8605
Epoch 77/100
18/18 [==============================] - 0s 4ms/step - loss: 107.5879 - mae: 107.5879 - v
```

```
al_loss: 571.9299 - val_mae: 571.9299
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 78/100
18/18 [==============================] - 0s 4ms/step - loss: 108.1598 - mae: 108.1598 - v
al_loss: 575.2383 - val_mae: 575.2383
Epoch 79/100
18/18 [==============================] - 0s 4ms/step - loss: 107.9175 - mae: 107.9175 - v
al_loss: 617.3071 - val_mae: 617.3071
Epoch 80/100
18/18 [==============================] - 0s 4ms/step - loss: 108.9510 - mae: 108.9510 - v
al_loss: 583.4847 - val_mae: 583.4847
Epoch 81/100
18/18 [==============================] - 0s 5ms/step - loss: 106.0505 - mae: 106.0505 - v
al_loss: 570.0802 - val_mae: 570.0802
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 82/100
18/18 [==============================] - 0s 4ms/step - loss: 115.6827 - mae: 115.6827 - v
al_loss: 575.7382 - val_mae: 575.7382
Epoch 83/100
18/18 [==============================] - 0s 5ms/step - loss: 110.9379 - mae: 110.9379 - v
al_loss: 659.6570 - val_mae: 659.6570
Epoch 84/100
18/18 [==============================] - 0s 5ms/step - loss: 111.4836 - mae: 111.4836 - v
al_loss: 570.1959 - val_mae: 570.1959
Epoch 85/100
18/18 [==============================] - 0s 4ms/step - loss: 107.5948 - mae: 107.5948 - v
al_loss: 601.5945 - val_mae: 601.5945
Epoch 86/100
18/18 [==============================] - 0s 4ms/step - loss: 108.9426 - mae: 108.9426 - v
al_loss: 592.8107 - val_mae: 592.8107
Epoch 87/100
18/18 [==============================] - 0s 5ms/step - loss: 105.7717 - mae: 105.7717 - v
al_loss: 603.6169 - val_mae: 603.6169
Epoch 88/100
18/18 [==============================] - 0s 5ms/step - loss: 107.9217 - mae: 107.9217 - v
al_loss: 569.0500 - val_mae: 569.0500
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 89/100
18/18 [==============================] - 0s 5ms/step - loss: 106.0344 - mae: 106.0344 - v
al_loss: 568.9512 - val_mae: 568.9512
INFO:tensorflow:Assets written to: model_experiments/model_1_dense/assets
Epoch 90/100
18/18 [==============================] - 0s 5ms/step - loss: 105.4977 - mae: 105.4977 - v
al_loss: 581.7681 - val_mae: 581.7681
Epoch 91/100
18/18 [==============================] - 0s 5ms/step - loss: 108.8468 - mae: 108.8468 - v
al_loss: 573.6023 - val_mae: 573.6023
Epoch 92/100
18/18 [==============================] - 0s 5ms/step - loss: 110.8884 - mae: 110.8884 - v
al_loss: 576.8247 - val_mae: 576.8247
Epoch 93/100
18/18 [==============================] - 0s 5ms/step - loss: 113.8781 - mae: 113.8781 - v
al_loss: 608.3018 - val_mae: 608.3018
Epoch 94/100
18/18 [==============================] - 0s 5ms/step - loss: 110.5763 - mae: 110.5763 - v
al_loss: 601.6047 - val_mae: 601.6047
Epoch 95/100
18/18 [==============================] - 0s 4ms/step - loss: 106.5906 - mae: 106.5906 - v
al_loss: 570.3652 - val_mae: 570.3652
Epoch 96/100
18/18 [==============================] - 0s 4ms/step - loss: 116.9515 - mae: 116.9515 - v
al_loss: 615.2581 - val_mae: 615.2581
Epoch 97/100
18/18 [==============================] - 0s 5ms/step - loss: 108.0739 - mae: 108.0739 - v
al_loss: 580.3073 - val_mae: 580.3073
Epoch 98/100
18/18 [==============================] - 0s 4ms/step - loss: 108.7102 - mae: 108.7102 - v
al_loss: 586.6512 - val_mae: 586.6512
Epoch 99/100
18/18 [==============================] - 0s 5ms/step - loss: 109.0488 - mae: 109.0488 - v
al_loss: 570.0629 - val_mae: 570.0629
Epoch 100/100
```

```
18/18 [==============================] - 0s 4ms/step - loss: 106.1845 - mae: 106.1845 - v
al_loss: 585.9763 - val_mae: 585.9763
```

Out[ ]:

```
<keras.callbacks.History at 0x7fdcf48bd110>
```

Because of the small size of our data (less than 3000 total samples), the model trains very fast.

Let's evaluate it.

In [ ]:

```
# Evaluate model on test data
model_1.evaluate(test_windows, test_labels)
```

```
18/18 [==============================] - 0s 2ms/step - loss: 585.9762 - mae: 585.9762
```

Out[ ]:

```
[585.9761962890625, 585.9761962890625]
```

You'll notice the model achieves the same `val_loss` (in this case, this is MAE) as the last epoch.

But if we load in the version of `model_1` which was saved to file using the `ModelCheckpoint` callback, we should see an improvement in results.

In [ ]:

```
# Load in saved best performing model_1 and evaluate on test data
model_1 = tf.keras.models.load_model("model_experiments/model_1_dense")
model_1.evaluate(test_windows, test_labels)
```

```
18/18 [==============================] - 0s 3ms/step - loss: 568.9512 - mae: 568.9512
```

Out[ ]:

```
[568.951171875, 568.951171875]
```

Much better! Due to the fluctuating performance of the model during training, loading back in the best performing model see's a sizeable improvement in MAE.

Making forecasts with a model (on the test dataset)

We've trained a model and evaluated the it on the test data, but the project we're working on is called BitPredict so how do you think we could use our model to make predictions?

Since we're going to be running more modelling experiments, let's write a function which:

1. **Takes in a trained model (just like `model_1`)**
2. **Takes in some input data (just like the data the model was trained on)**
3. **Passes the input data to the model's `predict()` method**
4. **Returns the predictions**

In [ ]:

```
def make_preds(model, input_data):
  """
  Uses model to make predictions on input_data.

  Parameters
  ----------
  model: trained model
  input_data: windowed input data (same kind of data model was trained on)

  Returns model predictions on input_data.
  """
```

```
  forecast = model.predict(input_data)
  return tf.squeeze(forecast) # return 1D array of predictions
```

Nice!

Now let's use our `make_preds()` and see how it goes.

In [ ]:

```
# Make predictions using model_1 on the test dataset and view the results
model_1_preds = make_preds(model_1, test_windows)
len(model_1_preds), model_1_preds[:10]
```

Out[ ]:

```
(556, <tf.Tensor: shape=(10,), dtype=float32, numpy=
 array([8861.711, 8769.886, 9015.71 , 8795.517, 8723.809, 8730.11 ,
        8691.95 , 8502.054, 8460.961, 8516.547], dtype=float32)>)
```

> **Note: With these outputs, our model isn't *forecasting* yet. It's only making predictions on the test dataset. Forecasting would involve a model making predictions into the future, however, the test dataset is only a pseudofuture.**

Excellent! Now we've got some prediction values, let's use the `evaluate_preds()` we created before to compare them to the ground truth.

In [ ]:

```
# Evaluate preds
model_1_results = evaluate_preds(y_true=tf.squeeze(test_labels), # reduce to right shape
                                 y_pred=model_1_preds)
model_1_results
```

Out[ ]:

```
{'mae': 568.95123,
 'mape': 2.5448983,
 'mase': 0.9994897,
 'mse': 1171744.0,
 'rmse': 1082.4713}
```

How did our model go? Did it beat the naïve forecast?

In [ ]:

```
naive_results
```

Out[ ]:

```
{'mae': 567.9802,
 'mape': 2.516525,
 'mase': 0.99957,
 'mse': 1147547.0,
 'rmse': 1071.2362}
```

It looks like our naïve model beats our first deep model on nearly every metric.

That goes to show the power of the naïve model and the reason for having a baseline for any machine learning project.

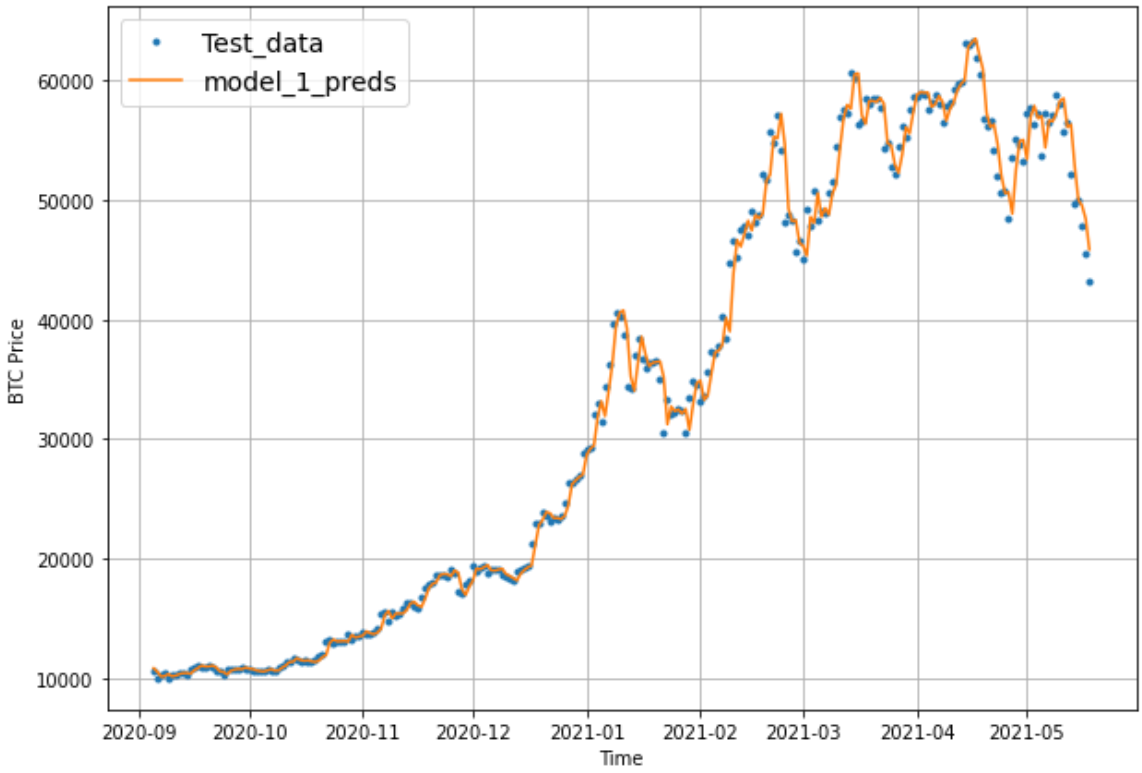
And of course, no evaluation would be finished without visualizing the results.

Let's use the `plot_time_series()` function to plot `model_1_preds` against the test data.

In [ ]:

```
offset = 300
```

```
plt.figure(figsize=(10, 7))
# Account for the test_window offset and index into test_labels to ensure correct plottin
g
plot_time_series(timesteps=X_test[-len(test_windows):], values=test_labels[:, 0], start=
offset, label="Test_data")
plot_time_series(timesteps=X_test[-len(test_windows):], values=model_1_preds, start=offse
t, format="-", label="model_1_preds")
```



What's wrong with these predictions?

As mentioned before, they're on the test dataset. So they're not actual forecasts.

With our current model setup, how do you think we'd make forecasts for the future?

Have a think about it for now, we'll cover this later on.

Model 2: Dense (window = 30, horizon = 1)

A naïve model is currently beating our handcrafted deep learning model.

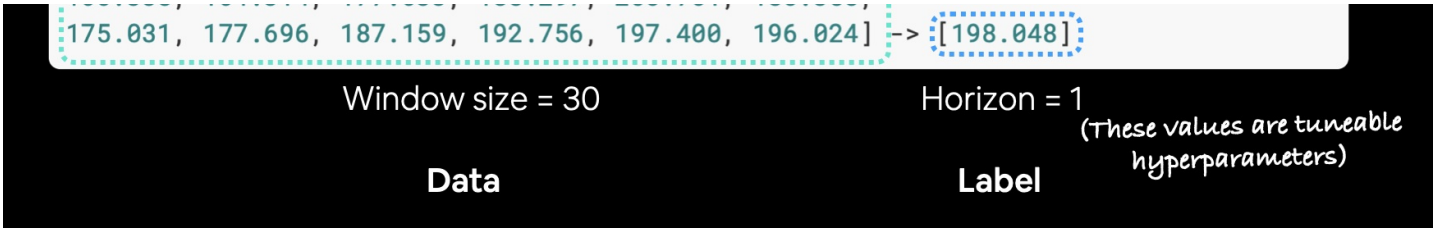
We can't let this happen.

Let's continue our modelling experiments.

We'll keep the previous model architecture but use a window size of 30.

In other words, we'll use the previous 30 days of Bitcoin prices to try and predict the next day price.

```
# Window for one month with the target of predicting the next day (Bitcoin prices)
[123.654, 125.455, 108.584, 118.674, 121.338, 120.655,
121.795, 123.033, 124.049, 125.961, 125.279, 125.927,
126.383, 135.241, 133.203, 142.763, 137.923, 142.951,
152.551, 160.338, 164.314, 177.633, 188.297, 200.701,
180.355, 175.031, 177.696, 187.159, 192.756, 197.400] -> [196.024]

[125.455, 108.584, 118.674, 121.338, 120.655, 121.795,
123.033, 124.049, 125.961, 125.279, 125.927, 126.383,
135.241, 133.203, 142.763, 137.923, 142.951, 152.551,
160.338, 164.314, 177.633, 188.297, 200.701, 180.355,
```



Example of Bitcoin prices windowed for 30 days to predict a horizon of 1.

> **Note: Recall from before, the window size (how many timesteps to use to fuel a forecast) and the horizon (how many timesteps to predict into the future) are hyperparameters. This means you can tune them to try and find values will result in better performance.**

We'll start our second modelling experiment by preparing datasets using the functions we created earlier.

In [ ]:

```
HORIZON = 1 # predict one step at a time
WINDOW_SIZE = 30 # use 30 timesteps in the past
```

In [ ]:

```
# Make windowed data with appropriate horizon and window sizes
full_windows, full_labels = make_windows(prices, window_size=WINDOW_SIZE, horizon=HORIZO
N)
len(full_windows), len(full_labels)
```

Out[ ]:

```
(2757, 2757)
```

In [ ]:

```
# Make train and testing windows
train_windows, test_windows, train_labels, test_labels = make_train_test_splits(windows=
full_windows, labels=full_labels)
len(train_windows), len(test_windows), len(train_labels), len(test_labels)
```

Out[ ]:

```
(2205, 552, 2205, 552)
```

Data prepared!

Now let's construct `model_2`, a model with the same architecture as `model_1` as well as the same training routine.

In [ ]:

```
tf.random.set_seed(42)

# Create model (same model as model 1 but data input will be different)
model_2 = tf.keras.Sequential([
  layers.Dense(128, activation="relu"),
  layers.Dense(HORIZON) # need to predict horizon number of steps into the future
], name="model_2_dense")

model_2.compile(loss="mae",
                optimizer=tf.keras.optimizers.Adam())

model_2.fit(train_windows,
            train_labels,
            epochs=100,
            batch_size=128,
            verbose=0,
            validation_data=(test_windows, test_labels),
            callbacks=[create_model_checkpoint(model_name=model_2.name)])
```

```
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
```

```
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_2_dense/assets
```

Out[ ]:

```
<keras.callbacks.History at 0x7fdcf1094290>
```

Once again, training goes nice and fast.

Let's evaluate our model's performance.

In [ ]:

```python
# Evaluate model 2 preds
model_2.evaluate(test_windows, test_labels)
```

```
18/18 [==============================] - 0s 2ms/step - loss: 608.9615
```

Out[ ]:

```
608.9614868164062
```

Hmmm... is that the best it did?

How about we try loading in the best performing `model_2` which was saved to file thanks to our `ModelCheckpoint` callback.

In [ ]:

```python
# Load in best performing model
model_2 = tf.keras.models.load_model("model_experiments/model_2_dense/")
model_2.evaluate(test_windows, test_labels)
```

```
18/18 [==============================] - 0s 2ms/step - loss: 608.9615
```

Out[ ]:

```
608.9614868164062
```

Excellent! Loading back in the best performing model see's a performance boost.

But let's not stop there, let's make some predictions with `model_2` and then evaluate them just as we did before.

In [ ]:

```python
# Get forecast predictions
```

```
model_2_preds = make_preds(model_2,
                           input_data=test_windows)
```

In [ ]:

```
# Evaluate results for model 2 predictions
model_2_results = evaluate_preds(y_true=tf.squeeze(test_labels), # remove 1 dimension of test labels
                                 y_pred=model_2_preds)
model_2_results
```

Out[ ]:

```
{'mae': 608.9615,
 'mape': 2.7693386,
 'mase': 1.0644706,
 'mse': 1281438.8,
 'rmse': 1132.0065}
```

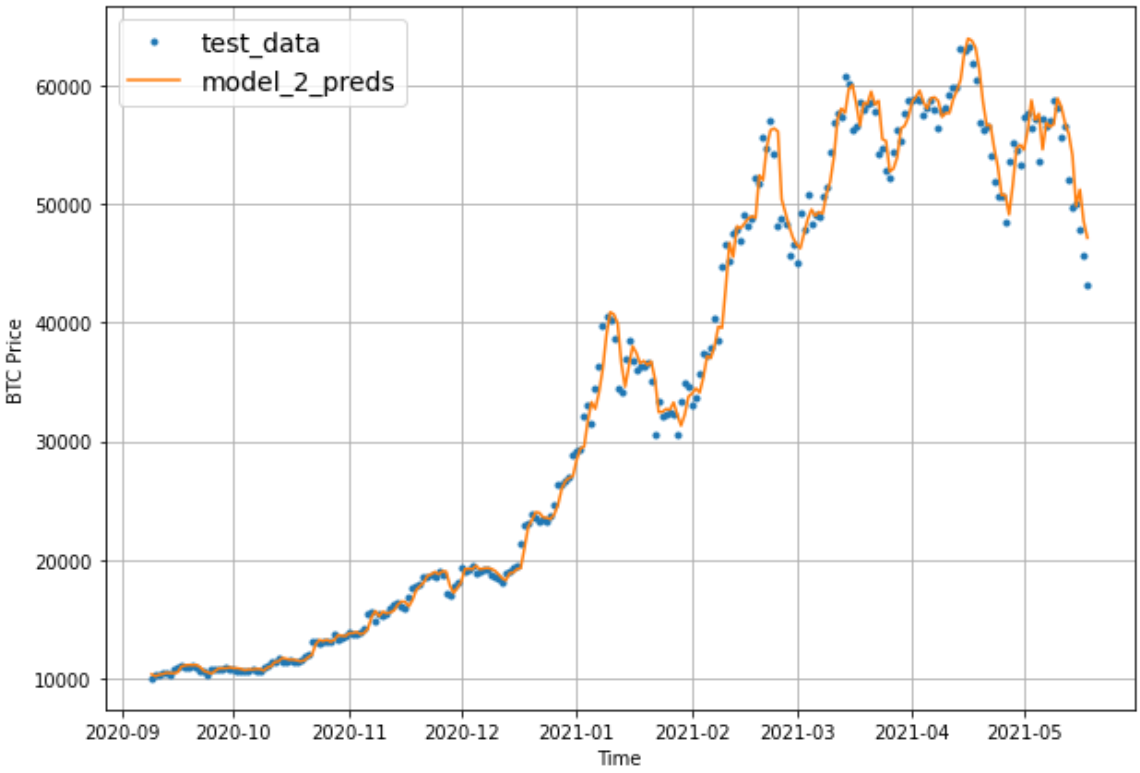
It looks like `model_2` performs worse than the naïve model as well as `model_1`!

Does this mean a smaller window size is better? (I'll leave this as a challenge you can experiment with)

How do the predictions look?

In [ ]:

```
offset = 300
plt.figure(figsize=(10, 7))
# Account for the test_window offset
plot_time_series(timesteps=X_test[-len(test_windows):], values=test_labels[:, 0], start=offset, label="test_data")
plot_time_series(timesteps=X_test[-len(test_windows):], values=model_2_preds, start=offset, format="-", label="model_2_preds")
```



Model 3: Dense (window = 30, horizon = 7)

Let's try and predict 7 days ahead given the previous 30 days.

First, we'll update the `HORIZON` and `WINDOW_SIZE` variables and create windowed data.

In [ ]:

```
```

```
HORIZON = 7
WINDOW_SIZE = 30

full_windows, full_labels = make_windows(prices, window_size=WINDOW_SIZE, horizon=HORIZO
N)
len(full_windows), len(full_labels)
```

Out[ ]:

```
(2751, 2751)
```

And we'll split the full dataset windows into training and test sets.

In [ ]:

```
train_windows, test_windows, train_labels, test_labels = make_train_test_splits(windows=
full_windows, labels=full_labels, test_split=0.2)
len(train_windows), len(test_windows), len(train_labels), len(test_labels)
```

Out[ ]:

```
(2200, 551, 2200, 551)
```

Now let's build, compile, fit and evaluate a model.

In [ ]:

```
tf.random.set_seed(42)

# Create model (same as model_1 except with different data input size)
model_3 = tf.keras.Sequential([
  layers.Dense(128, activation="relu"),
  layers.Dense(HORIZON)
], name="model_3_dense")

model_3.compile(loss="mae",
                optimizer=tf.keras.optimizers.Adam())

model_3.fit(train_windows,
            train_labels,
            batch_size=128,
            epochs=100,
            verbose=0,
            validation_data=(test_windows, test_labels),
            callbacks=[create_model_checkpoint(model_name=model_3.name)])
```

```
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
INFO:tensorflow:Assets written to: model_experiments/model_3_dense/assets
```

Out[ ]:

```
<keras.callbacks.History at 0x7fdcf1d36350>
```

In [ ]:

```
# How did our model with a larger window size and horizon go?
model_3.evaluate(test_windows, test_labels)
```

```
18/18 [==============================] - 0s 2ms/step - loss: 1321.5201
```

Out[ ]:

```
1321.5201416015625
```

To compare apples to apples (best performing model to best performing model), we've got to load in the best version of `model_3`.

In [ ]:

```
# Load in best version of model_3 and evaluate
model_3 = tf.keras.models.load_model("model_experiments/model_3_dense/")
model_3.evaluate(test_windows, test_labels)
```

```
18/18 [==============================] - 0s 2ms/step - loss: 1237.5063
```

Out[ ]:

```
1237.50634765625
```

In this case, the error will be higher because we're predicting 7 steps at a time.

This makes sense though because the further you try and predict, the larger your error will be (think of trying to predict the weather 7 days in advance).

Let's make predictions with our model using the `make_preds()` function and evaluate them using the `evaluate_preds()` function.

In [ ]:

```
# The predictions are going to be 7 steps at a time (this is the HORIZON size)
model_3_preds = make_preds(model_3,
                           input_data=test_windows)
model_3_preds[:5]
```

Out[ ]:

```
<tf.Tensor: shape=(5, 7), dtype=float32, numpy=
array([[9004.693 , 9048.1   , 9425.088 , 9258.258 , 9495.798 , 9558.451 ,
        9357.354 ],
       [8735.507 , 8840.304 , 9247.793 , 8885.6   , 9097.188 , 9174.328 ,
        9156.819 ],
       [8672.508 , 8782.388 , 9123.8545, 8770.37  , 9007.13  , 9003.87  ,
        9042.724 ],
       [8874.398 , 8784.737 , 9043.901 , 8943.051 , 9033.479 , 9176.488 ,
        9039.676 ],
       [8825.891 , 8777.4375, 8926.779 , 8870.178 , 9213.232 , 9268.156 ,
        8942.485 ]], dtype=float32)>
```

In [ ]:

```
# Calculate model_3 results - these are going to be multi-dimensional because
# we're trying to predict more than one step at a time.
model_3_results = evaluate_preds(y_true=tf.squeeze(test_labels),
                                 y_pred=model_3_preds)
model_3_results
```

Out[ ]:

```
{'mae': array([ 513.60516 ,  355.08356 ,  327.17035 ,  358.50977 ,  420.53207 ,
         537.8539  ,  545.6606  ,  485.92307 ,  584.4969  ,  687.3814  ,
         836.22675 ,  755.1571  ,  731.4959  ,  775.3398  ,  567.9548  ,
         266.80865 ,  188.80217 ,  188.1077  ,  253.09521 ,  301.4336  ,
         151.10742 ,  196.81424 ,  191.46184 ,  231.65067 ,  143.6114  ,
         122.59033 ,  132.78844 ,  190.8116  ,  179.1598  ,  228.25949 ,
```

314.44016 , 379.093 , 278.3254 , 295.34604 , 299.38525 ,
248.64963 , 299.75635 , 259.6937 , 180.30559 , 206.72887 ,
374.62906 , 144.85129 , 142.33607 , 131.1158 , 93.94057 ,
54.825542, 73.7943 , 103.59989 , 121.3337 , 168.67209 ,
183.90945 , 152.25314 , 186.57137 , 146.91309 , 240.42955 ,
351.00668 , 540.9516 , 549.15674 , 521.2422 , 526.8553 ,
453.36313 , 257.98166 , 277.29492 , 301.82465 , 455.71756 ,
458.96002 , 503.44427 , 522.3119 , 223.07687 , 250.09473 ,
297.14468 , 400.56976 , 495.79785 , 364.33664 , 283.3654 ,
325.59457 , 238.21178 , 318.9777 , 460.77246 , 651.0755 ,
835.88074 , 669.9654 , 319.5452 , 261.99496 , 142.39217 ,
136.62834 , 154.75252 , 221.32631 , 290.50446 , 503.8846 ,
414.2602 , 434.35727 , 377.98926 , 251.7899 , 204.28418 ,
388.22684 , 360.65945 , 493.80902 , 614.86035 , 754.7017 ,
533.7708 , 378.98898 , 280.49484 , 339.48062 , 413.12875 ,
452.87933 , 550.53906 , 634.57214 , 935.57227 , 931.6003 ,
881.2804 , 426.40094 , 179.45885 , 121.66225 , 160.43806 ,
372.07037 , 341.85776 , 476.52475 , 618.3239 , 1038.8976 ,
1569.5022 , 2157.1196 , 1987.6074 , 2158.6108 , 2303.5603 ,
2662.9421 , 1405.5017 , 728.30145 , 351.70047 , 322.03168 ,
493.94742 , 435.48492 , 565.55615 , 350.1165 , 289.0203 ,
251.21645 , 409.05566 , 342.2103 , 320.83594 , 330.88596 ,
357.8457 , 335.9481 , 206.1031 , 544.3837 , 700.0093 ,
468.91965 , 404.4963 , 172.80176 , 308.33664 , 210.47566 ,
318.95444 , 486.1319 , 428.87982 , 533.91095 , 433.7813 ,
396.89447 , 138.40346 , 189.96617 , 170.39133 , 181.54387 ,
282.8902 , 264.3889 , 250.62172 , 240.33713 , 276.9541 ,
326.0306 , 489.61572 , 686.2451 , 526.9798 , 603.0119 ,
825.38275 , 871.04694 , 990.06903 , 1090.0593 , 560.2842 ,
310.5219 , 371.39035 , 348.45996 , 355.73465 , 429.56473 ,
581.2839 , 550.5218 , 635.4312 , 913.40375 , 840.71844 ,
305.03488 , 493.93414 , 751.3177 , 410.63434 , 220.62459 ,
282.25473 , 291.85352 , 422.50293 , 458.65375 , 637.2345 ,
647.82367 , 417.24442 , 220.23717 , 246.72168 , 200.22935 ,
455.14926 , 719.6058 , 696.3037 , 485.09836 , 294.9534 ,
170.52371 , 211.82463 , 270.56488 , 189.89355 , 171.21721 ,
366.4471 , 231.96303 , 318.78726 , 273.79352 , 358.55582 ,
412.22797 , 512.31573 , 185.43848 , 196.38113 , 200.00586 ,
224.58678 , 213.41965 , 186.23926 , 113.37096 , 172.23117 ,
168.89885 , 236.09584 , 307.59683 , 328.93903 , 566.5961 ,
285.04282 , 300.4495 , 125.45201 , 168.94322 , 137.25754 ,
143.50404 , 145.27776 , 107.340126, 77.16044 , 131.87096 ,
134.01953 , 167.45871 , 137.92188 , 148.91281 , 204.95467 ,
157.89732 , 196.95201 , 167.93861 , 156.45578 , 188.6808 ,
161.44113 , 90.997765, 136.84096 , 198.51799 , 230.13881 ,
294.7839 , 594.24286 , 699.64856 , 815.137 , 905.33887 ,
1127.2999 , 1342.6952 , 1317.0686 , 590.1699 , 296.34543 ,
243.4368 , 256.3987 , 222.28488 , 323.8387 , 82.51339 ,
120.14453 , 249.14857 , 205.73674 , 243.45451 , 250.64857 ,
287.27567 , 224.3803 , 266.55972 , 221.50935 , 218.49539 ,
272.42242 , 279.5964 , 252.58775 , 381.56473 , 417.97028 ,
624.5562 , 368.0364 , 327.25473 , 263.4396 , 349.95242 ,
398.62485 , 297.07465 , 147.04924 , 164.54367 , 313.36246 ,
477.7486 , 675.042 , 897.269 , 1094.7098 , 1460.177 ,
1398.5858 , 952.72375 , 645.9594 , 166.17912 , 144.66867 ,
189.1822 , 304.81375 , 435.10742 , 449.64426 , 425.244 ,
441.93637 , 407.29605 , 252.4036 , 248.36928 , 336.17062 ,
482.8111 , 437.53043 , 533.0855 , 346.98047 , 127.68722 ,
110.208565, 301.75223 , 195.50697 , 174.65248 , 238.4707 ,
302.3711 , 313.51703 , 310.1289 , 200.29701 , 172.47209 ,
140.2302 , 252.48479 , 289.32407 , 343.62222 , 504.11816 ,
635.1339 , 602.131 , 519.0612 , 214.90291 , 195.91197 ,
265.03683 , 198.9008 , 345.51227 , 517.22235 , 631.02997 ,
988.4249 , 1174.2136 , 1196.2849 , 1253.93 , 526.06573 ,
210.31306 , 215.27205 , 169.86008 , 283.78543 , 269.76117 ,
228.63477 , 186.64719 , 410.5434 , 601.2659 , 618.12164 ,
768.7538 , 1158.5449 , 1232.8798 , 1254.429 , 423.02524 ,
390.03613 , 367.23926 , 209.55803 , 530.2231 , 821.38293 ,
812.37195 , 741.19464 , 953.48285 , 1258.9928 , 1844.9332 ,
1605.2852 , 1112.7673 , 594.18945 , 549.7676 , 632.5438 ,
829.2656 , 1103.2213 , 1130.1024 , 1033.3527 , 878.5851 ,
595.4096 , 1115.6515 , 1371.1725 , 1385.25 , 387.52484 ,

```
        303.74945 ,  495.12305 ,  719.5804  ,  648.9032  ,  766.1624  ,
        683.23157 ,  596.4121  ,  523.09235 ,  577.34015 , 1337.6241  ,
       2454.597   , 2759.2363  , 3135.8757  , 3407.3806  , 3602.6362  ,
       2527.1584  , 1158.1016  ,  679.84375 ,  748.5586  , 1073.4896  ,
       1217.7305  , 1459.5664  , 2502.7336  , 3075.8264  , 3090.2869  ,
       2564.758   , 2660.5317  , 2928.9348  , 3690.5566  , 3525.1433  ,
       4183.6704  , 5144.0693  , 4384.6533  , 4164.9614  , 4431.0967  ,
       3335.121   , 3017.6858  , 2589.5403  , 4103.6313  , 5582.089   ,
       5108.456   , 1382.9648  ,  953.5435  , 1081.6842  , 2483.1992  ,
       2992.056   , 3127.0942  , 2972.2717  , 3054.6477  , 3283.7107  ,
       3617.3652  , 1170.2556  , 1074.1886  , 1059.3181  , 1044.5385  ,
       1060.6202  , 2115.7234  , 3569.8901  , 3474.3647  , 2448.0173  ,
       2641.4202  , 3188.6016  , 4899.913   , 5516.293   , 4635.4883  ,
       4677.036   , 5732.8804  , 5866.7344  , 8083.341   , 5049.8237  ,
       1476.2316  , 1873.3314  , 1219.1033  , 2324.0698  , 2467.0044  ,
       3005.0864  , 2805.0486  , 3688.2556  , 2962.7131  , 3664.236   ,
       5618.6836  , 6925.0913  , 9751.091   , 8790.822   , 5057.624   ,
       2971.6863  , 1715.495   , 1125.7689  , 2201.432   , 3969.5005  ,
       2988.5112  , 2911.9336  , 2796.869   , 3937.8755  , 5449.5913  ,
       6395.02    , 6148.2524  , 5437.706   , 4291.8267  , 2362.3962  ,
       1657.3444  , 1426.63    , 1647.7142  , 2730.3962  , 1813.716   ,
       2061.149   , 3095.72    , 4312.8096  , 5500.7227  , 5258.029   ,
       5152.029   , 1331.4045  , 1634.4017  , 2493.4336  , 3957.9548  ,
       4499.2153  , 2215.8533  , 2587.4136  , 1622.4392  ,  591.8778  ,
       1297.4688  , 1284.5931  ,  915.22656 ,  940.4565  , 1084.6562  ,
        875.89954 , 1041.3527  , 2299.2512  , 3152.6038  , 3518.3923  ,
       2531.5212  , 2682.5737  , 3094.2947  , 3829.3286  , 5550.609   ,
       7159.343   , 8820.526   , 8587.338   , 6777.814   , 5193.2866  ,
       4793.332   , 2605.2517  , 1668.3544  , 2510.7244  , 4105.6865  ,
       6278.3906  , 4109.759   , 1711.1211  , 2261.9878  , 2384.754   ,
       1246.6602  , 1672.8494  , 1103.1322  , 1743.3594  , 1608.452   ,
       1933.8995  , 2742.3455  , 3945.3633  , 5765.9414  , 6955.8384  ,
       8107.707   ], dtype=float32),
'mape': array([ 5.8601456 ,  4.087344  ,  3.7758112 ,  4.187648  ,  4.9781313 ,
        6.4483633 ,  6.614449  ,  6.0616755 ,  7.547588  ,  9.095916  ,
       11.300213  , 10.396466  , 10.109091  , 10.697323  ,  7.8628383 ,
        3.6872823 ,  2.585636  ,  2.505     ,  3.3504546 ,  4.013595  ,
        2.0036254 ,  2.6397336 ,  2.6087892 ,  3.160931  ,  1.9612147 ,
        1.6679796 ,  1.7952247 ,  2.5901198 ,  2.436256  ,  3.1327312 ,
        4.3595414 ,  5.2856445 ,  3.9403565 ,  4.3142104 ,  4.3733163 ,
        3.6376827 ,  4.362974  ,  3.7757344 ,  2.6192985 ,  2.9007592 ,
        5.1501136 ,  2.014742  ,  1.9706616 ,  1.8104095 ,  1.3007166 ,
        0.75351447,  1.0192169 ,  1.4282547 ,  1.6741827 ,  2.3589606 ,
        2.563099  ,  2.1327207 ,  2.5974777 ,  2.017332  ,  3.1387668 ,
        4.502132  ,  6.959625  ,  6.9602156 ,  6.5331526 ,  6.5757833 ,
        5.610295  ,  3.0642097 ,  3.2474015 ,  3.4847279 ,  5.222716  ,
        5.229475  ,  5.727943  ,  5.9273834 ,  2.5284538 ,  2.8663971 ,
        3.4309018 ,  4.6874056 ,  5.841506  ,  4.30981   ,  3.3552744 ,
        3.8162014 ,  2.7363582 ,  3.4623704 ,  4.971694  ,  7.0260944 ,
        8.974986  ,  7.167823  ,  3.401335  ,  2.7761257 ,  1.522323  ,
        1.4535459 ,  1.6354691 ,  2.252781  ,  2.9475648 ,  5.073901  ,
        4.1032834 ,  4.295298  ,  3.7052956 ,  2.4620814 ,  2.013232  ,
        3.8428285 ,  3.6306674 ,  5.0162754 ,  6.2794676 ,  7.757465  ,
        5.495671  ,  3.8907118 ,  2.8546908 ,  3.531113  ,  4.464891  ,
        5.0162473 ,  6.1257253 ,  7.1294317 , 10.711446  , 10.66408   ,
       10.087247  ,  4.893642  ,  2.071729  ,  1.362841  ,  1.8011061 ,
        4.219873  ,  4.055696  ,  5.744903  ,  7.492872  , 15.032968  ,
       24.38846   , 35.42812   , 34.702423  , 39.86109   , 42.83657   ,
       49.6046    , 26.081123  , 13.669959  ,  6.360867  ,  5.4986367 ,
        7.9925113 ,  6.8575478 ,  8.82061   ,  5.311197  ,  4.5279207 ,
        3.870413  ,  6.2283797 ,  5.2470355 ,  5.1045485 ,  5.329424  ,
        5.7789664 ,  5.358487  ,  3.2211263 ,  8.118441  , 10.231978  ,
        6.726861  ,  5.757309  ,  2.4041245 ,  4.2864537 ,  2.977692  ,
        4.454852  ,  6.9261975 ,  6.1590724 ,  7.7638254 ,  6.308201  ,
        5.7371497 ,  1.9719449 ,  2.7191157 ,  2.3998728 ,  2.568591  ,
        3.953004  ,  3.6584775 ,  3.453558  ,  3.301869  ,  3.7942226 ,
        4.310324  ,  6.4319835 ,  8.567106  ,  6.293662  ,  7.0238814 ,
        9.469167  ,  9.915066  , 11.217036  , 12.28066   ,  6.209284  ,
        3.2802734 ,  3.8508573 ,  3.6124747 ,  3.7172396 ,  4.6127787 ,
        6.2338514 ,  6.037696  ,  7.1359625 , 10.132258  ,  9.337389  ,
        3.379447  ,  5.190343  ,  7.822919  ,  4.2644997 ,  2.328503  ,
        3.0320923 ,  3.122556  ,  4.5721846 ,  5.013454  ,  7.0593185 ,
```

```
 7.180876  ,  4.6288185 ,  2.4294462 ,  2.6893413 ,  2.1346936 ,
 4.6703887 ,  7.429678  ,  7.177754  ,  4.9274387 ,  2.969763  ,
 1.7036035 ,  2.1020818 ,  2.7900844 ,  1.9541025 ,  1.7562655 ,
 3.8170404 ,  2.4557195 ,  3.335657  ,  2.8912652 ,  3.8006802 ,
 4.3685193 ,  5.4371476 ,  1.96905   ,  2.086854  ,  2.1329875 ,
 2.4014482 ,  2.2829742 ,  1.9878384 ,  1.2127163 ,  1.8306142 ,
 1.8156711 ,  2.5207567 ,  3.2923858 ,  3.5837193 ,  6.1877694 ,
 3.1183949 ,  3.2884538 ,  1.3740205 ,  1.8472785 ,  1.5017135 ,
 1.5671413 ,  1.5889224 ,  1.1630745 ,  0.83710796,  1.4223018 ,
 1.4407477 ,  1.8030001 ,  1.4797007 ,  1.599079  ,  2.2175813 ,
 1.7130904 ,  2.1422603 ,  1.8232663 ,  1.7015574 ,  2.0563016 ,
 1.7596645 ,  0.98229814,  1.4560648 ,  2.1070251 ,  2.4114208 ,
 3.0499995 ,  5.7738442 ,  6.618569  ,  7.495159  ,  8.227665  ,
10.186895  , 11.928129  , 11.676125  ,  5.199727  ,  2.5929737 ,
 2.1249924 ,  2.2335236 ,  1.9233094 ,  2.846623  ,  0.7076666 ,
 1.0350616 ,  2.132228  ,  1.778991  ,  2.1161108 ,  2.1635604 ,
 2.4792838 ,  1.9270526 ,  2.2551262 ,  1.8457135 ,  1.8112589 ,
 2.262672  ,  2.3515143 ,  2.1173332 ,  3.2111585 ,  3.5736024 ,
 5.3715696 ,  3.18694   ,  2.858538  ,  2.295805  ,  3.0580947 ,
 3.4806054 ,  2.5900245 ,  1.2625037 ,  1.4050376 ,  2.7622259 ,
 4.3049536 ,  6.2830725 ,  8.513568  , 10.517086  , 14.161078  ,
13.640608  ,  9.320237  ,  6.3080745 ,  1.6300542 ,  1.3983166 ,
 1.8191801 ,  2.8875985 ,  4.068832  ,  4.1736484 ,  3.92286   ,
 4.0472727 ,  3.7387483 ,  2.3069727 ,  2.3063238 ,  3.1941137 ,
 4.5895944 ,  4.157942  ,  5.0558047 ,  3.3005743 ,  1.2102847 ,
 1.037109  ,  2.8088598 ,  1.8151615 ,  1.6353209 ,  2.2353542 ,
 2.8412042 ,  2.952003  ,  2.9187465 ,  1.8913074 ,  1.6178632 ,
 1.2982845 ,  2.3039308 ,  2.6034784 ,  3.017652  ,  4.444261  ,
 5.5989385 ,  5.26695   ,  4.535601  ,  1.8671715 ,  1.7055075 ,
 2.3196962 ,  1.7318225 ,  2.7897227 ,  4.062603  ,  4.8944497 ,
 7.6816416 ,  9.107798  ,  9.207181  ,  9.502263  ,  3.9754786 ,
 1.5744586 ,  1.5997065 ,  1.242373  ,  2.0770247 ,  1.974714  ,
 1.6764196 ,  1.3502706 ,  2.7858412 ,  3.9915662 ,  4.0835853 ,
 5.016299  ,  7.6388373 ,  8.078356  ,  8.1091385 ,  2.6612654 ,
 2.4994845 ,  2.2963924 ,  1.304031  ,  3.2585588 ,  4.9145164 ,
 4.7377896 ,  4.253874  ,  5.353144  ,  6.9800787 , 10.170477  ,
 8.772201  ,  6.0037975 ,  3.184015  ,  3.0090022 ,  3.637018  ,
 4.7740936 ,  6.289234  ,  6.4191875 ,  5.904088  ,  4.9963098 ,
 3.1906037 ,  5.8562336 ,  7.18103   ,  7.211652  ,  2.0219958 ,
 1.5912417 ,  2.624319  ,  3.8534112 ,  3.5140653 ,  4.1405296 ,
 3.6843975 ,  3.215591  ,  2.8228219 ,  2.8898468 ,  6.2849083 ,
11.458009  , 12.342688  , 13.764669  , 14.824157  , 15.566204  ,
10.829556  ,  4.9278235 ,  2.8345613 ,  3.0226748 ,  4.311801  ,
 4.7200484 ,  5.516797  ,  9.303045  , 11.212711  , 11.037065  ,
 8.631595  ,  8.57773   ,  9.389048  , 11.497772  , 10.463535  ,
11.683612  , 13.823776  , 11.273857  , 10.805961  , 11.400717  ,
 8.4966135 ,  7.8893795 ,  7.169477  , 11.423848  , 15.540573  ,
14.24638   ,  3.8430922 ,  2.545029  ,  2.926236  ,  7.288314  ,
 8.999365  ,  9.586316  ,  9.271836  ,  9.423251  , 10.242945  ,
11.396603  ,  3.6557918 ,  3.3454742 ,  3.2290876 ,  3.159098  ,
 3.1852577 ,  6.1412444 , 10.163153  ,  9.673946  ,  6.657339  ,
 6.975828  ,  8.377453  , 12.244584  , 13.00238   , 10.547294  ,
10.269914  , 12.4054    , 12.598526  , 17.187769  , 10.61679   ,
 3.0717697 ,  3.7662785 ,  2.3915377 ,  4.4263554 ,  4.6217527 ,
 5.4785805 ,  5.1152287 ,  6.826434  ,  5.61515   ,  7.0909567 ,
11.599258  , 14.51015   , 20.678543  , 18.670784  , 10.813841  ,
 6.395925  ,  3.68131   ,  2.323965  ,  4.4593787 ,  8.030951  ,
 5.9532943 ,  5.6428704 ,  5.1379447 ,  7.1625338 ,  9.831415  ,
11.209798  , 10.556711  ,  9.350725  ,  7.3507624 ,  3.9944    ,
 2.8233469 ,  2.3929296 ,  2.8540497 ,  4.729086  ,  3.236382  ,
 3.6810744 ,  5.6685753 ,  8.005048  , 10.247115  ,  9.747856  ,
 9.5460415 ,  2.4771707 ,  2.9854116 ,  4.3493814 ,  6.8806467 ,
 7.758685  ,  3.8011405 ,  4.4251776 ,  2.7610898 ,  1.0093751 ,
 2.2422    ,  2.2289147 ,  1.58688   ,  1.6239213 ,  1.8814766 ,
 1.5162641 ,  1.7166423 ,  3.7594385 ,  5.0978    ,  5.664717  ,
 4.0754285 ,  4.3486743 ,  5.1419683 ,  6.618967  ,  9.823991  ,
12.98137   , 16.388018  , 16.260675  , 13.249782  , 10.180592  ,
 9.383685  ,  5.1218452 ,  3.2893803 ,  4.634893  ,  7.4068375 ,
11.284324  ,  7.287351  ,  3.0196326 ,  4.006139  ,  4.172643  ,
 2.233759  ,  3.0054173 ,  1.9702865 ,  3.1210938 ,  2.7774746 ,
 3.4924886 ,  5.1419373 ,  7.6572022 , 11.470654  , 14.311543  ,
17.28017   ], dtype=float32),
```

```
'mase': 2.2020736,
'mse': array([3.15562156e+05, 1.69165547e+05, 1.44131812e+05, 1.76002922e+05,
       2.63519750e+05, 3.91517188e+05, 3.99524688e+05, 3.44422312e+05,
       4.93892156e+05, 6.88785625e+05, 9.18703562e+05, 8.00988125e+05,
       6.35508625e+05, 6.45535125e+05, 3.57740000e+05, 1.03007117e+05,
       5.33132578e+04, 4.65829805e+04, 1.10349188e+05, 1.11647695e+05,
       4.35359297e+04, 5.45387773e+04, 4.55849375e+04, 7.64886406e+04,
       4.03074453e+04, 2.71072188e+04, 1.90268887e+04, 4.55625000e+04,
       4.25345820e+04, 6.63246172e+04, 1.16741711e+05, 1.69440984e+05,
       1.10486977e+05, 1.59707406e+05, 1.61631141e+05, 1.23861828e+05,
       1.50757328e+05, 1.31196297e+05, 5.50281289e+04, 6.28155391e+04,
       1.77474391e+05, 2.68843926e+04, 2.53033750e+04, 2.58044258e+04,
       1.34592480e+04, 3.88203076e+03, 9.22772559e+03, 1.22399551e+04,
       2.20868379e+04, 3.60825352e+04, 4.87962109e+04, 3.74652812e+04,
       4.10560898e+04, 4.17418750e+04, 1.10490836e+05, 2.00762031e+05,
       3.48149000e+05, 3.58060812e+05, 3.28375281e+05, 2.96505844e+05,
       2.33354141e+05, 1.54047203e+05, 1.43171328e+05, 1.62586469e+05,
       2.96452719e+05, 2.66880719e+05, 2.94189062e+05, 3.10067656e+05,
       6.27178750e+04, 8.63868672e+04, 9.69545391e+04, 2.01814359e+05,
       2.97053594e+05, 2.00674891e+05, 1.27305148e+05, 1.32509578e+05,
       8.46020391e+04, 1.74817453e+05, 2.95115844e+05, 4.75888719e+05,
       7.21894500e+05, 4.62711281e+05, 1.33154750e+05, 8.13640156e+04,
       3.43108672e+04, 3.45204180e+04, 5.36851641e+04, 7.42413281e+04,
       1.09339375e+05, 2.79682844e+05, 2.19119422e+05, 1.99934359e+05,
       1.65903391e+05, 8.16738047e+04, 4.82395195e+04, 1.65301000e+05,
       1.81699391e+05, 3.30645000e+05, 4.63625750e+05, 6.75044000e+05,
       3.34911500e+05, 1.69343750e+05, 1.03219234e+05, 1.48347609e+05,
       2.98684844e+05, 3.71952656e+05, 4.43306688e+05, 5.27904688e+05,
       1.12945275e+06, 1.00652938e+06, 8.03246875e+05, 2.21277969e+05,
       6.64886406e+04, 2.66182090e+04, 3.29688867e+04, 1.68663250e+05,
       1.76257219e+05, 2.99365562e+05, 4.31793438e+05, 1.91346712e+06,
       3.90353825e+06, 6.29502500e+06, 5.49042400e+06, 6.25676850e+06,
       6.28618000e+06, 7.23876400e+06, 2.08143525e+06, 6.66786500e+05,
       1.44639844e+05, 1.83413594e+05, 3.41026562e+05, 3.02069719e+05,
       3.71339750e+05, 2.02418219e+05, 1.36151141e+05, 8.24758203e+04,
       1.96006719e+05, 1.61595656e+05, 1.71401391e+05, 2.00606359e+05,
       2.37198891e+05, 1.58720172e+05, 6.36030352e+04, 3.38904031e+05,
       5.23524156e+05, 2.86212188e+05, 1.92326125e+05, 3.73765547e+04,
       1.31962000e+05, 9.33163047e+04, 1.18443672e+05, 3.00598719e+05,
       2.28996984e+05, 3.57954281e+05, 2.36219078e+05, 1.71530359e+05,
       2.68482539e+04, 4.79163125e+04, 4.00419297e+04, 5.41031875e+04,
       1.07641391e+05, 9.80020000e+04, 7.08350078e+04, 6.55985312e+04,
       8.53740000e+04, 1.33837578e+05, 2.72516312e+05, 6.54133500e+05,
       4.97212750e+05, 6.11190062e+05, 9.92907688e+05, 9.56692312e+05,
       1.09541412e+06, 1.19283125e+06, 3.75002219e+05, 2.22053797e+05,
       2.60620031e+05, 1.88109625e+05, 1.87927500e+05, 2.49058609e+05,
       3.79192656e+05, 4.07706781e+05, 6.24783062e+05, 9.62079375e+05,
       7.95344562e+05, 1.40557531e+05, 2.65227406e+05, 6.26269500e+05,
       2.05804234e+05, 6.53089570e+04, 1.33562969e+05, 1.14692070e+05,
       2.26913016e+05, 2.61226500e+05, 5.08516250e+05, 4.90744656e+05,
       2.38721828e+05, 6.85555703e+04, 7.30576641e+04, 5.36113750e+04,
       3.28223562e+05, 6.26447375e+05, 5.17735438e+05, 3.32536062e+05,
       1.72657266e+05, 8.28138125e+04, 1.43392656e+05, 8.83169844e+04,
       5.15846680e+04, 4.20798125e+04, 1.78049141e+05, 1.17351570e+05,
       1.16870422e+05, 1.02416836e+05, 1.67488781e+05, 2.05509953e+05,
       2.73221469e+05, 4.73881641e+04, 4.60577500e+04, 5.33725703e+04,
       7.87071094e+04, 7.06486719e+04, 5.23131758e+04, 2.50280723e+04,
       4.52024609e+04, 4.14732461e+04, 7.00107344e+04, 1.24132148e+05,
       1.34168141e+05, 3.38401562e+05, 9.75599844e+04, 1.10023000e+05,
       1.99194824e+04, 3.32738164e+04, 2.43711914e+04, 2.76617012e+04,
       2.70386660e+04, 1.58115801e+04, 1.16166348e+04, 2.74852148e+04,
       2.95266426e+04, 3.95874922e+04, 2.81846035e+04, 3.19155312e+04,
       5.60368477e+04, 3.98338398e+04, 5.02201602e+04, 3.19759512e+04,
       2.96142285e+04, 4.48333711e+04, 4.32265664e+04, 1.52425576e+04,
       2.86077324e+04, 4.23488125e+04, 7.25032578e+04, 1.21924688e+05,
       6.41490688e+05, 8.50461562e+05, 1.11135362e+06, 1.22003100e+06,
       1.54110162e+06, 2.03452138e+06, 1.76440575e+06, 4.22100781e+05,
       1.30513797e+05, 7.30581250e+04, 8.56110391e+04, 7.42401094e+04,
       1.49466766e+05, 1.25623496e+04, 2.13100293e+04, 9.81795703e+04,
       7.28018203e+04, 1.26064742e+05, 1.01337961e+05, 1.24553164e+05,
       7.13400000e+04, 8.41950469e+04, 6.25289375e+04, 8.18789453e+04,
       1.27303195e+05, 8.94661250e+04, 1.04100523e+05, 1.66147031e+05,
```

```
        2.32076422e+05, 4.41965062e+05, 1.75838281e+05, 1.76445328e+05,
        9.12169375e+04, 1.67975469e+05, 1.87876422e+05, 1.07636930e+05,
        3.26578105e+04, 5.35412461e+04, 1.38143531e+05, 3.28638219e+05,
        6.38893688e+05, 1.08279325e+06, 1.55802525e+06, 2.42136925e+06,
        2.08980088e+06, 9.83019312e+05, 4.27435500e+05, 5.84014961e+04,
        4.02855312e+04, 5.27582578e+04, 1.33694297e+05, 2.41687594e+05,
        2.54570500e+05, 2.18475578e+05, 2.39844719e+05, 1.72920359e+05,
        9.46134531e+04, 8.14810703e+04, 1.83791891e+05, 3.41631531e+05,
        2.67090906e+05, 3.54819906e+05, 1.69607641e+05, 2.63053145e+04,
        2.11339043e+04, 1.11686047e+05, 6.08852578e+04, 4.64014297e+04,
        6.97661953e+04, 1.16936102e+05, 1.44631031e+05, 1.14626898e+05,
        7.92795234e+04, 5.69158320e+04, 3.03999824e+04, 9.47752891e+04,
        1.19674977e+05, 2.15400000e+05, 3.15528031e+05, 4.17611781e+05,
        3.91754000e+05, 2.83129906e+05, 8.15816328e+04, 5.00839805e+04,
        1.01595930e+05, 6.17696953e+04, 3.16726438e+05, 5.76748000e+05,
        7.41103312e+05, 1.33218000e+06, 1.56516188e+06, 1.55553338e+06,
        1.68517900e+06, 3.21850031e+05, 6.30693242e+04, 7.78374766e+04,
        5.12543047e+04, 1.14444250e+05, 1.13520102e+05, 7.03014531e+04,
        4.71108398e+04, 4.52435781e+05, 7.79108188e+05, 7.32536312e+05,
        1.03909088e+06, 1.56208738e+06, 1.63956762e+06, 1.72789975e+06,
        4.00525438e+05, 2.34608641e+05, 1.76724859e+05, 6.78014609e+04,
        3.90382062e+05, 9.80141562e+05, 1.03537131e+06, 8.44669812e+05,
        1.38893975e+06, 1.98341138e+06, 3.62835425e+06, 2.66920750e+06,
        1.42117000e+06, 4.54602219e+05, 3.65967062e+05, 9.46589188e+05,
        1.45184738e+06, 1.85900112e+06, 1.89930475e+06, 1.82373112e+06,
        1.21286088e+06, 4.94641594e+05, 1.49650312e+06, 2.03091800e+06,
        2.25657350e+06, 2.11182750e+05, 1.21396250e+05, 3.04308844e+05,
        7.11903438e+05, 6.20312000e+05, 8.05386562e+05, 6.26902875e+05,
        5.66389875e+05, 4.27517438e+05, 7.83491562e+05, 3.63025775e+06,
        8.18037650e+06, 1.07676250e+07, 1.26563580e+07, 1.33500370e+07,
        1.33957390e+07, 6.52511100e+06, 1.60696775e+06, 8.56758062e+05,
        1.15616712e+06, 1.69269000e+06, 2.17122950e+06, 3.34218200e+06,
        7.84994850e+06, 1.08378930e+07, 1.05726990e+07, 1.01602540e+07,
        1.14205900e+07, 1.16179750e+07, 1.73510000e+07, 1.79301800e+07,
        2.70626460e+07, 3.72665640e+07, 2.97687340e+07, 2.14580120e+07,
        2.44066140e+07, 1.56006580e+07, 1.03771360e+07, 9.93532800e+06,
        2.18889460e+07, 3.45001800e+07, 2.72495080e+07, 2.51545225e+06,
        2.11410550e+06, 1.65101288e+06, 8.78065000e+06, 1.42263970e+07,
        1.50277600e+07, 1.42628750e+07, 1.11520980e+07, 1.31385740e+07,
        1.44270080e+07, 2.10807200e+06, 2.37278650e+06, 2.06521262e+06,
        1.51271400e+06, 1.63591925e+06, 6.01038950e+06, 1.48181850e+07,
        1.51462730e+07, 8.88826900e+06, 1.03656340e+07, 1.24236510e+07,
        3.12300180e+07, 4.44122480e+07, 3.35616920e+07, 3.37026000e+07,
        4.33024720e+07, 4.04647440e+07, 6.68120760e+07, 2.68142400e+07,
        3.37228800e+06, 6.18199650e+06, 3.33604150e+06, 9.76003100e+06,
        9.50538200e+06, 1.48300340e+07, 1.08088990e+07, 1.52812380e+07,
        1.21298770e+07, 1.47110990e+07, 4.46196920e+07, 6.19225800e+07,
        1.05739048e+08, 8.06071760e+07, 3.00356620e+07, 1.30779130e+07,
        4.55988750e+06, 1.67929975e+06, 6.99432300e+06, 1.63487150e+07,
        1.21498980e+07, 1.21798540e+07, 1.68979740e+07, 2.39387020e+07,
        3.73065680e+07, 5.09610480e+07, 4.59523240e+07, 3.20003840e+07,
        2.06371540e+07, 8.14507050e+06, 4.14795750e+06, 4.23499100e+06,
        3.09784650e+06, 8.37917500e+06, 7.36108300e+06, 7.06094350e+06,
        1.57310800e+07, 2.78767160e+07, 4.07339880e+07, 3.28521600e+07,
        2.89016500e+07, 3.40228600e+06, 3.84223250e+06, 8.10806000e+06,
        1.74778820e+07, 2.13824600e+07, 5.77506950e+06, 7.30700800e+06,
        3.70581275e+06, 6.14524500e+05, 1.92765088e+06, 2.29573075e+06,
        1.09031975e+06, 1.19423488e+06, 2.01535425e+06, 1.38583700e+06,
        2.60915325e+06, 7.48127550e+06, 1.25728690e+07, 1.49188240e+07,
        7.64712100e+06, 9.50861100e+06, 1.16537410e+07, 2.15388360e+07,
        4.32744960e+07, 6.52254160e+07, 9.30125040e+07, 8.49970400e+07,
        5.87007080e+07, 3.43532560e+07, 2.82685780e+07, 9.30442400e+06,
        4.66932300e+06, 8.00189950e+06, 2.05975580e+07, 4.07234120e+07,
        1.93118840e+07, 4.04353375e+06, 5.92508700e+06, 7.37346950e+06,
        2.68488000e+06, 5.27761500e+06, 1.84823362e+06, 4.87232050e+06,
        4.87888400e+06, 5.77534400e+06, 1.21702240e+07, 2.70199400e+07,
        4.94704840e+07, 6.90300960e+07, 8.50554880e+07], dtype=float32),
 'rmse': array([  561.7492  ,   411.2974  ,   379.64694 ,   419.52707 ,
          513.34174 ,   625.7134  ,   632.07965 ,   586.87506 ,
          702.7746  ,   829.9311  ,   958.4903  ,   894.97943 ,
          797.1879  ,   803.452   ,   598.1137  ,   320.94724 ,
          230.89664 ,   215.8309  ,   332.18848 ,   334.13724 ,
```

208.65266 , 233.53539 , 213.50632 , 276.5658 ,
200.76715 , 164.6427 , 137.93799 , 213.45375 ,
206.23912 , 257.53568 , 341.67487 , 411.6321 ,
332.3958 , 399.6341 , 402.03375 , 351.9401 ,
388.27478 , 362.2103 , 234.58073 , 250.63026 ,
421.2771 , 163.96461 , 159.07036 , 160.63757 ,
116.014 , 62.305946, 96.06105 , 110.63433 ,
148.61641 , 189.95403 , 220.89862 , 193.55951 ,
202.62302 , 204.30829 , 332.4016 , 448.06476 ,
590.04156 , 598.38184 , 573.0404 , 544.5235 ,
483.0674 , 392.4885 , 378.37988 , 403.22012 ,
544.47473 , 516.605 , 542.39197 , 556.8372 ,
250.43535 , 293.91644 , 311.3752 , 449.23752 ,
545.02625 , 447.9675 , 356.79846 , 364.01865 ,
290.8643 , 418.11176 , 543.24567 , 689.84686 ,
849.64374 , 680.2288 , 364.9038 , 285.2438 ,
185.23193 , 185.7967 , 231.7006 , 272.47263 ,
330.66504 , 528.85046 , 468.10193 , 447.1402 ,
407.3124 , 285.7863 , 219.63496 , 406.57227 ,
426.26212 , 575.0174 , 680.9007 , 821.6106 ,
578.7154 , 411.51398 , 321.27753 , 385.1592 ,
546.5207 , 609.8793 , 665.8128 , 726.57056 ,
1062.7572 , 1003.25934 , 896.2404 , 470.40195 ,
257.8539 , 163.15088 , 181.57336 , 410.6863 ,
419.83 , 547.14307 , 657.1099 , 1383.2812 ,
1975.7374 , 2508.9888 , 2343.1653 , 2501.3535 ,
2507.2256 , 2690.495 , 1442.718 , 816.56995 ,
380.3155 , 428.26813 , 583.9748 , 549.6087 ,
609.3766 , 449.90912 , 368.98663 , 287.18607 ,
442.72644 , 401.98962 , 414.00653 , 447.891 ,
487.03067 , 398.39697 , 252.1964 , 582.15466 ,
723.54974 , 534.98804 , 438.55002 , 193.33018 ,
363.26575 , 305.47717 , 344.15646 , 548.26886 ,
478.53625 , 598.29285 , 486.0237 , 414.16226 ,
163.85437 , 218.89795 , 200.1048 , 232.60092 ,
328.0875 , 313.0527 , 266.14847 , 256.1221 ,
292.1883 , 365.8382 , 522.03094 , 808.7852 ,
705.1331 , 781.7865 , 996.4475 , 978.1065 ,
1046.6204 , 1092.1682 , 612.37427 , 471.22586 ,
510.50952 , 433.7161 , 433.50607 , 499.05774 ,
615.78625 , 638.5192 , 790.4321 , 980.85645 ,
891.8209 , 374.91006 , 515.0024 , 791.3719 ,
453.65652 , 255.55615 , 365.4627 , 338.66217 ,
476.35388 , 511.10318 , 713.1032 , 700.5317 ,
488.5917 , 261.8312 , 270.2918 , 231.54132 ,
572.908 , 791.48425 , 719.5384 , 576.6594 ,
415.52045 , 287.7739 , 378.6722 , 297.18173 ,
227.1226 , 205.13364 , 421.95868 , 342.56616 ,
341.86316 , 320.02628 , 409.25394 , 453.33206 ,
522.70593 , 217.68823 , 214.61069 , 231.02502 ,
280.54785 , 265.7982 , 228.72073 , 158.20264 ,
212.60869 , 203.64981 , 264.5954 , 352.3239 ,
366.2897 , 581.72296 , 312.34595 , 331.69714 ,
141.1364 , 182.4111 , 156.11276 , 166.31807 ,
164.43439 , 125.7441 , 107.780495, 165.78665 ,
171.83319 , 198.96605 , 167.8827 , 178.64919 ,
236.72102 , 199.58418 , 224.09854 , 178.8182 ,
172.08786 , 211.73892 , 207.91 , 123.46075 ,
169.1382 , 205.78828 , 269.2643 , 349.1772 ,
800.93115 , 922.2048 , 1054.2076 , 1104.5502 ,
1241.4113 , 1426.3665 , 1328.3093 , 649.6928 ,
361.26694 , 270.29266 , 292.59366 , 272.4704 ,
386.6093 , 112.081894, 145.97955 , 313.3362 ,
269.8181 , 355.05597 , 318.33624 , 352.9209 ,
267.0955 , 290.16382 , 250.05786 , 286.145 ,
356.79572 , 299.1089 , 322.64612 , 407.6114 ,
481.7431 , 664.8045 , 419.33078 , 420.05396 ,
302.0214 , 409.84808 , 433.44717 , 328.0807 ,
180.71472 , 231.3898 , 371.67667 , 573.2698 ,
799.3082 , 1040.5736 , 1248.2089 , 1556.075 ,
1445.6144 , 991.4733 , 653.7855 , 241.664 ,
200.71255 , 229.69164 , 365.6423 , 491.61737 ,

```
 504.54974 ,   467.4137  ,   489.7394  ,   415.83694 ,
 307.593   ,   285.44888 ,   428.70956 ,   584.49255 ,
 516.80835 ,   595.6676  ,   411.8345  ,   162.18913 ,
 145.37505 ,   334.1946  ,   246.74936 ,   215.40991 ,
 264.1329  ,   341.95923 ,   380.3039  ,   338.56595 ,
 281.5662  ,   238.5704  ,   174.35591 ,   307.85596 ,
 345.94073 ,   464.11203 ,   561.7188  ,   646.22894 ,
 625.9026  ,   532.09955 ,   285.625   ,   223.7945  ,
 318.74115 ,   248.5351  ,   562.78455 ,   759.4393  ,
 860.8736  ,  1154.201   ,  1251.0643  ,  1247.2102  ,
1298.1444  ,   567.3183  ,   251.13608 ,   278.9937  ,
 226.39412 ,   338.29608 ,   336.92746 ,   265.14423 ,
 217.05034 ,   672.6335  ,   882.671   ,   855.88336 ,
1019.35803 ,  1249.835   ,  1280.4559  ,  1314.4961  ,
 632.8708  ,   484.36414 ,   420.38657 ,   260.38715 ,
 624.8056  ,   990.02106 ,  1017.5319  ,   919.0592  ,
1178.533   ,  1408.3364  ,  1904.8239  ,  1633.771   ,
1192.1284  ,   674.242   ,   604.9521  ,   972.9281  ,
1204.9263  ,  1363.4519  ,  1378.1527  ,  1350.4559  ,
1101.2997  ,   703.3076  ,  1223.3164  ,  1425.1029  ,
1502.1896  ,   459.54623 ,   348.41965 ,   551.64197 ,
 843.7437  ,   787.5989  ,   897.4333  ,   791.77203 ,
 752.5888  ,   653.84814 ,   885.1505  ,  1905.3235  ,
2860.1357  ,  3281.406   ,  3557.5776  ,  3653.7703  ,
3660.019   ,  2554.4297  ,  1267.6624  ,   925.6123  ,
1075.2522  ,  1301.0342  ,  1473.5092  ,  1828.1636  ,
2801.7761  ,  3292.0957  ,  3251.5688  ,  3187.5154  ,
3379.4363  ,  3408.5151  ,  4165.453   ,  4234.4043  ,
5202.1772  ,  6104.635   ,  5456.073   ,  4632.28    ,
4940.305   ,  3949.767   ,  3221.3562  ,  3152.0356  ,
4678.5625  ,  5873.6855  ,  5220.106   ,  1586.0177  ,
1453.9965  ,  1284.9175  ,  2963.2163  ,  3771.7898  ,
3876.5652  ,  3776.622   ,  3339.4758  ,  3624.7173  ,
3798.29    ,  1451.92    ,  1540.3851  ,  1437.0847  ,
1229.9243  ,  1279.0306  ,  2451.6096  ,  3849.44    ,
3891.8213  ,  2981.3203  ,  3219.5703  ,  3524.72    ,
5588.383   ,  6664.252   ,  5793.246   ,  5805.394   ,
6580.4614  ,  6361.1904  ,  8173.865   ,  5178.2466  ,
1836.379   ,  2486.362   ,  1826.4835  ,  3124.1047  ,
3083.0798  ,  3850.9783  ,  3287.689   ,  3909.1228  ,
3482.797   ,  3835.5054  ,  6679.7974  ,  7869.091   ,
10282.95   ,  8978.149   ,  5480.48    ,  3616.3398  ,
2135.3894  ,  1295.878   ,  2644.6782  ,  4043.3542  ,
3485.6704  ,  3489.9646  ,  4110.7144  ,  4892.719   ,
6107.91    ,  7138.7007  ,  6778.814   ,  5656.8877  ,
4542.8135  ,  2853.957   ,  2036.6534  ,  2057.9094  ,
1760.0701  ,  2894.6804  ,  2713.1316  ,  2657.2437  ,
3966.2427  ,  5279.841   ,  6382.3184  ,  5731.68    ,
5376.0254  ,  1844.5288  ,  1960.1613  ,  2847.4656  ,
4180.656   ,  4624.117   ,  2403.1375  ,  2703.1477  ,
1925.0488  ,   783.9162  ,  1388.3987  ,  1515.1669  ,
1044.1838  ,  1092.8105  ,  1419.6317  ,  1177.2158  ,
1615.2874  ,  2735.192   ,  3545.8242  ,  3862.4895  ,
2765.3428  ,  3083.6038  ,  3413.7578  ,  4640.995   ,
6578.3354  ,  8076.225   ,  9644.3     ,  9219.384   ,
7661.638   ,  5861.165   ,  5316.8203  ,  3050.3152  ,
2160.8616  ,  2828.763   ,  4538.453   ,  6381.4897  ,
4394.529   ,  2010.854   ,  2434.1501  ,  2715.4133  ,
1638.5604  ,  2297.3062  ,  1359.4976  ,  2207.3335  ,
2208.8198  ,  2403.1946  ,  3488.5847  ,  5198.071   ,
7033.5254  ,  8308.436   ,  9222.554   ], dtype=float32)}
```

Make our evaluation function work for larger horizons

You'll notice the outputs for `model_3_results` are multi-dimensional.

This is because the predictions are getting evaluated across the `HORIZON` timesteps (7 predictions at a time).

To fix this, let's adjust our `evaluate_preds()` function to work with multiple shapes of data.

In [ ]:

```
def evaluate_preds(y_true, y_pred):
  # Make sure float32 (for metric calculations)
  y_true = tf.cast(y_true, dtype=tf.float32)
  y_pred = tf.cast(y_pred, dtype=tf.float32)

  # Calculate various metrics
  mae = tf.keras.metrics.mean_absolute_error(y_true, y_pred)
  mse = tf.keras.metrics.mean_squared_error(y_true, y_pred)
  rmse = tf.sqrt(mse)
  mape = tf.keras.metrics.mean_absolute_percentage_error(y_true, y_pred)
  mase = mean_absolute_scaled_error(y_true, y_pred)

  # Account for different sized metrics (for longer horizons, reduce to single number)
  if mae.ndim > 0: # if mae isn't already a scalar, reduce it to one by aggregating tenso
rs to mean
    mae = tf.reduce_mean(mae)
    mse = tf.reduce_mean(mse)
    rmse = tf.reduce_mean(rmse)
    mape = tf.reduce_mean(mape)
    mase = tf.reduce_mean(mase)

  return {"mae": mae.numpy(),
          "mse": mse.numpy(),
          "rmse": rmse.numpy(),
          "mape": mape.numpy(),
          "mase": mase.numpy()}
```

Now we've updated `evaluate_preds()` to work with multiple shapes, how does it look?

In [ ]:

```
# Get model_3 results aggregated to single values
model_3_results = evaluate_preds(y_true=tf.squeeze(test_labels),
                                 y_pred=model_3_preds)
model_3_results
```

Out[ ]:

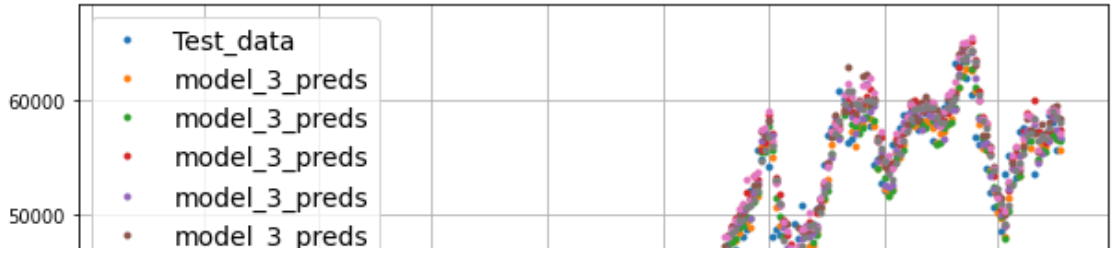
```
{'mae': 1237.5063,
 'mape': 5.5588784,
 'mase': 2.2020736,
 'mse': 5405198.5,
 'rmse': 1425.7477}
```

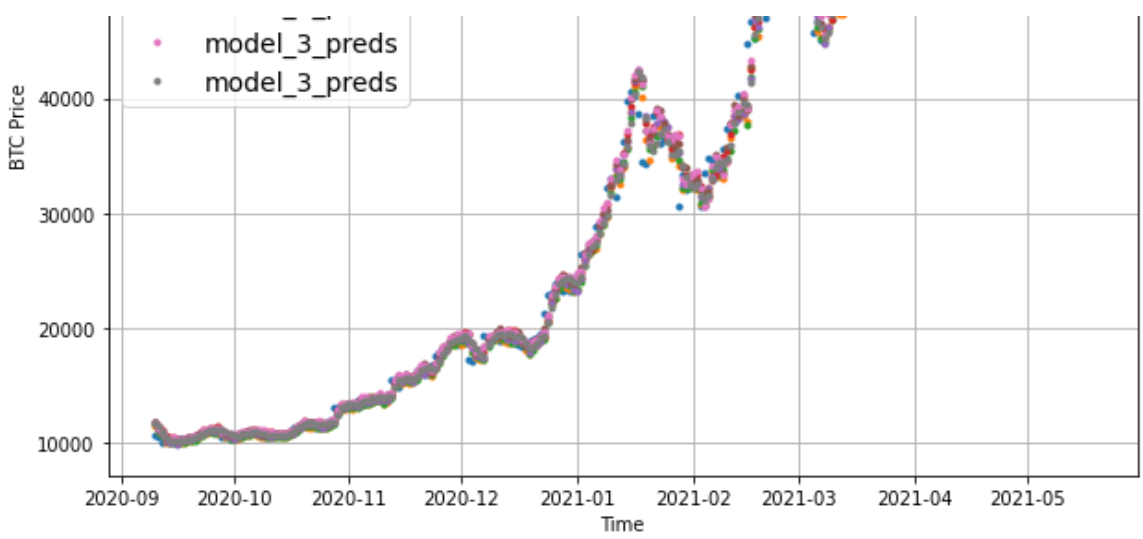
Time to visualize.

If our prediction evaluation metrics were mutli-dimensional, how do you think the predictions will look like if we plot them?

In [ ]:

```
offset = 300
plt.figure(figsize=(10, 7))
plot_time_series(timesteps=X_test[-len(test_windows):], values=test_labels[:, 0], start=
offset, label="Test_data")
# Checking the shape of model_3_preds results in [n_test_samples, HORIZON] (this will scr
ew up the plot)
plot_time_series(timesteps=X_test[-len(test_windows):], values=model_3_preds, start=offse
t, label="model_3_preds")
```





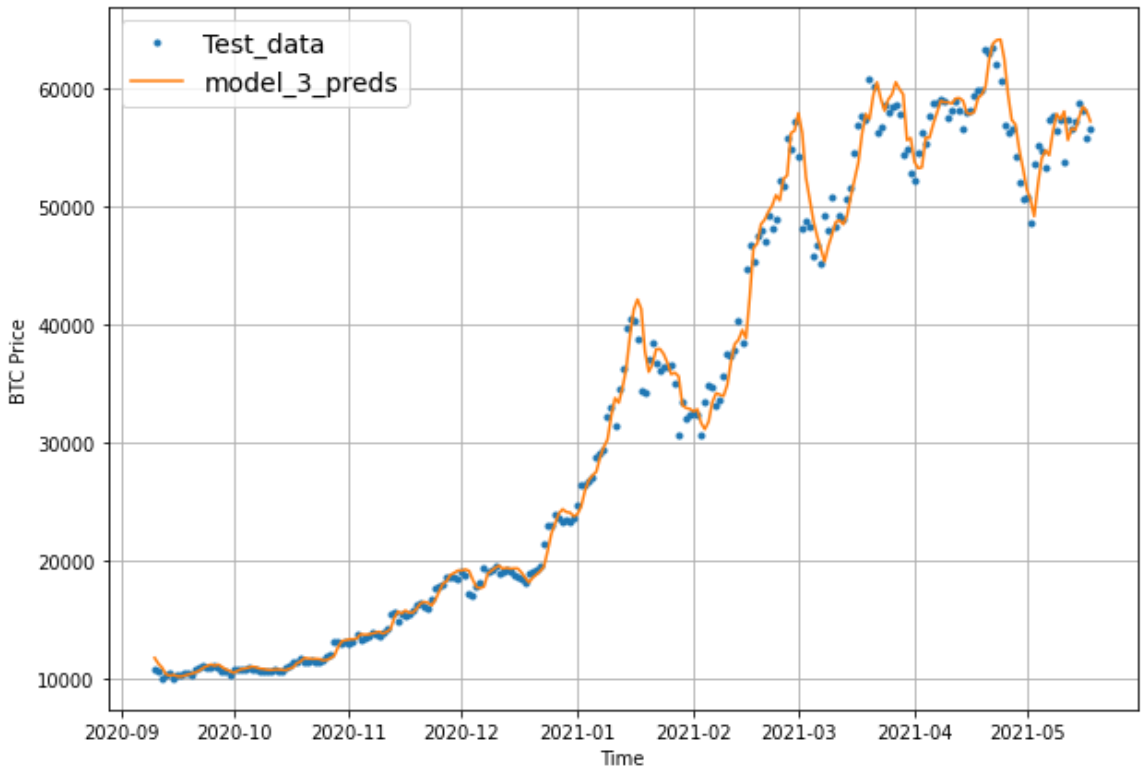
When we try to plot our multi-horizon predicts, we get a funky looking plot.

Again, we can fix this by aggregating our model's predictions.

> **Note: Aggregating the predictions (e.g. reducing a 7-day horizon to one value such as the mean) loses information from the original prediction. As in, the model predictions were trained to be made for 7-days but by reducing them to one, we gain the ability to plot them visually but we lose the extra information contained across multiple days.**

In [ ]:

```
offset = 300
plt.figure(figsize=(10, 7))
# Plot model_3_preds by aggregating them (note: this condenses information so the preds w
ill look fruther ahead than the test data)
plot_time_series(timesteps=X_test[-len(test_windows):],
                 values=test_labels[:, 0],
                 start=offset,
                 label="Test_data")
plot_time_series(timesteps=X_test[-len(test_windows):],
                 values=tf.reduce_mean(model_3_preds, axis=1),
                 format="-",
                 start=offset,
                 label="model_3_preds")
```



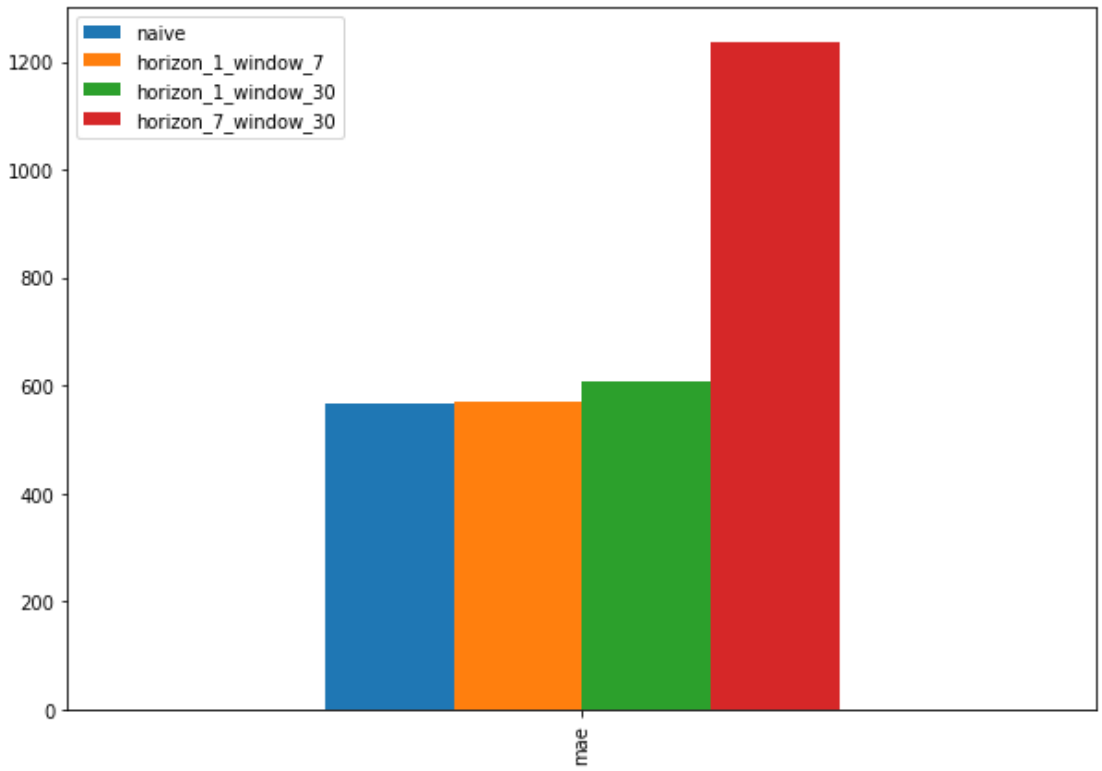
Which of our models is performing best so far?

So far, we've trained 3 models which use the same architecture but use different data inputs.

Let's compare them with the naïve model to see which model is performing the best so far.

In [ ]:

```
pd.DataFrame({"naive": naive_results["mae"],
              "horizon_1_window_7": model_1_results["mae"],
              "horizon_1_window_30": model_2_results["mae"],
              "horizon_7_window_30": model_3_results["mae"]}, index=["mae"]).plot(figsiz
e=(10, 7), kind="bar");
```



Woah, our naïve model is performing best (it's very hard to beat a naïve model in open systems) but the dense model with a horizon of 1 and a window size of 7 looks to be performing cloest.

Because of this, let's use `HORIZON=1` and `WINDOW_SIZE=7` for our next series of modelling experiments (in other words, we'll use the previous week of Bitcoin prices to try and predict the next day).

> **Note:** You might be wondering, why are the naïve results so good? One of the reasons could be due the presence of **autocorrelation** in the data. If a time series has **autocorrelation** it means the value at `t+1` (the next timestep) is typically close to the value at `t` (the current timestep). In other words, today's value is probably pretty close to yesterday's value. Of course, this isn't always the case but when it is, a naïve model will often get fairly good results.
>
> **Resource:** For more on how autocorrelation influences a model's predictions, see the article *How (not) to use Machine Learning for time series forecasting: Avoiding the pitfalls* by Vegard Flovik

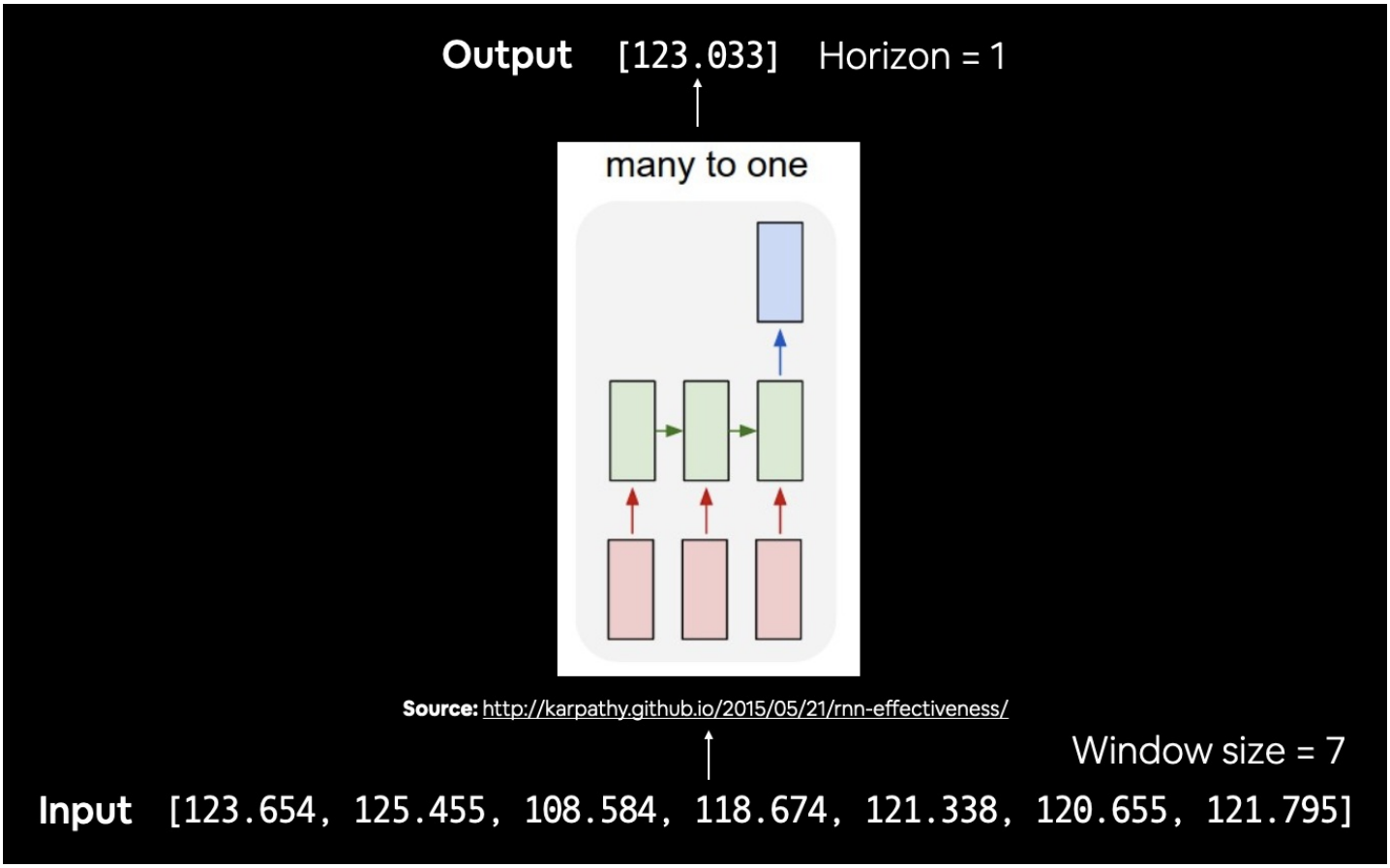
Model 4: Conv1D

Onto the next modelling experiment!

This time, we'll be using a Conv1D model. Because as we saw in the sequence modelling notebook, Conv1D models can be used for seq2seq (sequence to sequence) problems.

In our case, the input sequence is the previous 7 days of Bitcoin price data and the output is the next day (in

In our case, the input sequence is the previous 7 days of Bitcoin price data and the output is the next day (in seq2seq terms this is called a many to one problem).



Framing Bitcoin forecasting in seq2seq (sequence to sequence) terms. Using a window size of 7 and a horizon of one results in a many to one problem. Using a window size of >1 and a horizon of >1 results in a many to many problem. The diagram comes from Andrei Karpathy's [The Unreasonable Effectiveness of Recurrent Neural Networks](#).

Before we build a Conv1D model, let's recreate our datasets.

In [ ]:

```
HORIZON = 1 # predict next day
WINDOW_SIZE = 7 # use previous week worth of data
```

In [ ]:

```
# Create windowed dataset
full_windows, full_labels = make_windows(prices, window_size=WINDOW_SIZE, horizon=HORIZO
N)
len(full_windows), len(full_labels)
```

Out[ ]:

```
(2780, 2780)
```

In [ ]:

```
# Create train/test splits
train_windows, test_windows, train_labels, test_labels = make_train_test_splits(full_win
dows, full_labels)
len(train_windows), len(test_windows), len(train_labels), len(test_labels)
```

Out[ ]:

```
(2224, 556, 2224, 556)
```

Data windowed!

Now, since we're going to be using [Conv1D layers](#), we need to make sure our input shapes are correct.

The Conv1D layer in TensorFlow takes an input of: `(batch_size, timesteps, input_dim)`.

In our case, the `batch_size` (by default this is 32 but we can change it) is handled for us but the other values will be:

- **`timesteps = WINDOW_SIZE` - the `timesteps` is also often referred to as `features`, our features are the previous `WINDOW_SIZE` values of Bitcoin**
- **`input_dim = HORIZON` - our model views `WINDOW_SIZE` (one week) worth of data at a time to predict `HORIZON` (one day)**

Right now, our data has the `timesteps` dimension ready but we'll have to adjust it to have the `input_dim` dimension.

In [ ]:

```
# Check data sample shapes
train_windows[0].shape # returns (WINDOW_SIZE, )
```

Out[ ]:

```
(7,)
```

To fix this, we could adjust the shape of all of our `train_windows` or we could use a `tf.keras.layers.Lamdba` (called a Lambda layer) to do this for us in our model.

The Lambda layer wraps a function into a layer which can be used with a model.

Let's try it out.

In [ ]:

```
# Before we pass our data to the Conv1D layer, we have to reshape it in order to make sur
e it works
x = tf.constant(train_windows[0])
expand_dims_layer = layers.Lambda(lambda x: tf.expand_dims(x, axis=1)) # add an extra di
mension for timesteps
print(f"Original shape: {x.shape}") # (WINDOW_SIZE)
print(f"Expanded shape: {expand_dims_layer(x).shape}") # (WINDOW_SIZE, input_dim)
print(f"Original values with expanded shape:\n {expand_dims_layer(x)}")
```

```
Original shape: (7,)
Expanded shape: (7, 1)
Original values with expanded shape:
 [[123.65499]
 [125.455  ]
 [108.58483]
 [118.67466]
 [121.33866]
 [120.65533]
 [121.795  ]]
```

Looking good!

Now we've got a Lambda layer, let's build, compile, fit and evaluate a Conv1D model on our data.

> **Note: If you run the model below without the Lambda layer, you'll get an input shape error (one of the most common errors when building neural networks).**

In [ ]:

```
tf.random.set_seed(42)

# Create model
model_4 = tf.keras.Sequential([
  # Create Lambda layer to reshape inputs, without this layer, the model will error
  layers.Lambda(lambda x: tf.expand_dims(x, axis=1)), # resize the inputs to adjust for
window size / Conv1D 3D input requirements
```

```
  layers.Conv1D(filters=128, kernel_size=5, padding="causal", activation="relu"),
  layers.Dense(HORIZON)
], name="model_4_conv1D")

# Compile model
model_4.compile(loss="mae",
                optimizer=tf.keras.optimizers.Adam())

# Fit model
model_4.fit(train_windows,
            train_labels,
            batch_size=128,
            epochs=100,
            verbose=0,
            validation_data=(test_windows, test_labels),
            callbacks=[create_model_checkpoint(model_name=model_4.name)])
```

```
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
INFO:tensorflow:Assets written to: model_experiments/model_4_conv1D/assets
```

Out[ ]:

```
<keras.callbacks.History at 0x7fdcf1dfba50>
```

What does the Lambda layer look like in a summary?

In [ ]:

```
model_4.summary()
```

```
Model: "model_4_conv1D"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
lambda_1 (Lambda)            (None, 1, 7)              0         
```

```
_________________________________________________________________
conv1d (Conv1D)              (None, 1, 128)            4608
_________________________________________________________________
dense_6 (Dense)              (None, 1, 1)              129
=================================================================
Total params: 4,737
Trainable params: 4,737
Non-trainable params: 0
_________________________________________________________________
```

The Lambda layer appears the same as any other regular layer.

Time to evaluate the Conv1D model.

In [ ]:

```
# Load in best performing Conv1D model and evaluate it on the test data
model_4 = tf.keras.models.load_model("model_experiments/model_4_conv1D")
model_4.evaluate(test_windows, test_labels)
```

```
18/18 [==============================] - 0s 3ms/step - loss: 570.8283
```

Out[ ]:

```
570.8283081054688
```

In [ ]:

```
# Make predictions
model_4_preds = make_preds(model_4, test_windows)
model_4_preds[:10]
```

Out[ ]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([8851.464, 8754.471, 8983.928, 8759.672, 8703.627, 8708.295,
       8661.667, 8494.839, 8435.317, 8492.115], dtype=float32)>
```

In [ ]:

```
# Evaluate predictions
model_4_results = evaluate_preds(y_true=tf.squeeze(test_labels),
                                 y_pred=model_4_preds)
model_4_results
```

Out[ ]:

```
{'mae': 570.8283,
 'mape': 2.5593357,
 'mase': 1.0027872,
 'mse': 1176671.1,
 'rmse': 1084.7448}
```

Model 5: RNN (LSTM)

As you might've guessed, we can also use a recurrent neural network to model our sequential time series data.

> **Resource:** For more on the different types of recurrent neural networks you can use for sequence problems, see the Recurrent Neural Networks section of notebook 08.

Let's reuse the same data we used for the Conv1D model, except this time we'll create an LSTM-cell powered RNN to model our Bitcoin data.

Once again, one of the most important steps for the LSTM model will be getting our data into the right shape.

The `tf.keras.layers.LSTM()` layer takes a tensor with `[batch, timesteps, feature]` dimensions.

As mentioned earlier, the `batch` dimension gets taken care of for us but our data is currently only has the `feature` dimension (`WINDOW_SIZE`).

To fix this, just like we did with the `Conv1D` model, we can use a `tf.keras.layers.Lambda()` layer to adjust the shape of our input tensors to the LSTM layer.

In [ ]:

```
tf.random.set_seed(42)

# Let's build an LSTM model with the Functional API
inputs = layers.Input(shape=(WINDOW_SIZE))
x = layers.Lambda(lambda x: tf.expand_dims(x, axis=1))(inputs) # expand input dimension
to be compatible with LSTM
# print(x.shape)
# x = layers.LSTM(128, activation="relu", return_sequences=True)(x) # this layer will err
or if the inputs are not the right shape
x = layers.LSTM(128, activation="relu")(x) # using the tanh loss function results in a ma
ssive error
# print(x.shape)
# Add another optional dense layer (you could add more of these to see if they improve mo
del performance)
# x = layers.Dense(32, activation="relu")(x)
output = layers.Dense(HORIZON)(x)
model_5 = tf.keras.Model(inputs=inputs, outputs=output, name="model_5_lstm")

# Compile model
model_5.compile(loss="mae",
                optimizer=tf.keras.optimizers.Adam())

# Seems when saving the model several warnings are appearing: https://github.com/tensorfl
ow/tensorflow/issues/47554
model_5.fit(train_windows,
            train_labels,
            epochs=100,
            verbose=0,
            batch_size=128,
            validation_data=(test_windows, test_labels),
            callbacks=[create_model_checkpoint(model_name=model_5.name)])
```

```
WARNING:tensorflow:Layer lstm will not use cuDNN kernels since it doesn't meet the criter
ia. It will use a generic GPU kernel as fallback when running on GPU.
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
```

```
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
INFO:tensorflow:Assets written to: model_experiments/model_5_lstm/assets
```

Out[ ]:

```
<keras.callbacks.History at 0x7fdcf17b5190>
```

In [ ]:

```
# Load in best version of model 5 and evaluate on the test data
model_5 = tf.keras.models.load_model("model_experiments/model_5_lstm/")
model_5.evaluate(test_windows, test_labels)
```

```
WARNING:tensorflow:Layer lstm will not use cuDNN kernels since it doesn't meet the criter
ia. It will use a generic GPU kernel as fallback when running on GPU.
18/18 [==============================] - 0s 2ms/step - loss: 596.6447
```

Out[ ]:

```
596.6446533203125
```

Now we've got the best performing LSTM model loaded in, let's make predictions with it and evaluate them.

In [ ]:

```
# Make predictions with our LSTM model
model_5_preds = make_preds(model_5, test_windows)
model_5_preds[:10]
```

Out[ ]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([8991.225, 8823.2  , 9009.359, 8847.859, 8742.254, 8788.655,
       8744.746, 8552.568, 8514.823, 8542.873], dtype=float32)>
```

In [ ]:

```
# Evaluate model 5 preds
model_5_results = evaluate_preds(y_true=tf.squeeze(test_labels),
                                 y_pred=model_5_preds)
model_5_results
```

Out[ ]:

```
{'mae': 596.64465,
 'mape': 2.6838453,
 'mase': 1.0481395,
 'mse': 1273486.9,
 'rmse': 1128.4888}
```

Hmmm... it seems even with an LSTM-powered RNN we weren't able to beat our naïve models results.

Perhaps adding another variable will help?

> **Note:** I'm putting this here again as a reminder that because neural networks are such powerful algorithms, they can be used for almost any problem, however, that doesn't mean they'll achieve performant or usable results. You're probably starting to clue onto this now.

Make a multivariate time series

So far all of our models have barely kept up with the naïve forecast.

And so far all of them have been trained on a single variable (also called univariate time series): the historical price of Bitcoin.

If predicting the price of Bitcoin using the price of Bitcoin hasn't worked out very well, maybe giving our model more information may help.

More information is a vague term because we could actually feed almost anything to our model(s) and they would still try to find patterns.

For example, we could use the historical price of Bitcoin as well as anyone with the name Daniel Bourke Tweeted on that day to predict the future price of Bitcoin.

But would this help?

Porbably not.

What would be better is if we passed our model something related to Bitcoin (again, this is quite vauge, since in an open system like a market, you could argue everything is related).

This will be different for almost every time series you work on but in our case, we could try to see if the Bitcoin block reward size adds any predictive power to our model(s).

What is the Bitcoin block reward size?

The Bitcoin block reward size is the number of Bitcoin someone receives from mining a Bitcoin block.

At its inception, the Bitcoin block reward size was 50.

But every four years or so, the Bitcoin block reward halves.

For example, the block reward size went from 50 (starting January 2009) to 25 on November 28 2012.

Let's encode this information into our time series data and see if it helps a model's performance.

> **Note:** Adding an extra feature to our dataset such as the Bitcoin block reward size will take our data from **univariate** (only the historical price of Bitcoin) to **multivariate** (the price of Bitcoin as well as the block reward size).

In [ ]:

```
# Let's make a multivariate time series
bitcoin_prices.head()
```

Out[ ]:

Date	Price
2013-10-01	123.65499
2013-10-02	125.45500
2013-10-03	108.58483
2013-10-04	118.67466
2013-10-05	121.33866

Alright, time to add another feature column, the block reward size.

First, we'll need to create variables for the different block reward sizes as well as the dates they came into play.

The following block rewards and dates were sourced from cmcmarkets.com.

Block Reward	Start Date
50	3 January 2009 (2009-01-03)

25	28 November 2012
Block Reward	**Start Date**
12.5	9 July 2016
6.25	11 May 2020
3.125	TBA (expected 2024)
1.5625	TBA (expected 2028)

> 🔑 **Note:** Since our Bitcoin historical data starts from 01 October 2013, none of the timesteps in our multivariate time series will have a block reward of 50.

In [ ]:

```
# Block reward values
block_reward_1 = 50 # 3 January 2009 (2009-01-03) - this block reward isn't in our datase
t (it starts from 01 October 2013)
block_reward_2 = 25 # 28 November 2012
block_reward_3 = 12.5 # 9 July 2016
block_reward_4 = 6.25 # 11 May 2020

# Block reward dates (datetime form of the above date stamps)
block_reward_2_datetime = np.datetime64("2012-11-28")
block_reward_3_datetime = np.datetime64("2016-07-09")
block_reward_4_datetime = np.datetime64("2020-05-11")
```

We're going to get the days (indexes) for different block reward values.

This is important because if we're going to use multiple variables for our time series, they have to the same frequency as our original variable. For example, if our Bitcoin prices are daily, we need the block reward values to be daily as well.

> 🔑 **Note:** For using multiple variables, make sure they're the same frequency as each other. If your variables aren't at the same frequency (e.g. Bitcoin prices are daily but block rewards are weekly), you may need to transform them in a way that they can be used with your model.

In [ ]:

```
# Get date indexes for when to add in different block dates
block_reward_2_days = (block_reward_3_datetime - bitcoin_prices.index[0]).days
block_reward_3_days = (block_reward_4_datetime - bitcoin_prices.index[0]).days
block_reward_2_days, block_reward_3_days
```

Out[ ]:

```
(1012, 2414)
```

Now we can add another feature to our dataset `block_reward` (this gets lower over time so it may lead to increasing prices of Bitcoin).

In [ ]:

```
# Add block_reward column
bitcoin_prices_block = bitcoin_prices.copy()
bitcoin_prices_block["block_reward"] = None

# Set values of block_reward column (it's the last column hence -1 indexing on iloc)
bitcoin_prices_block.iloc[:block_reward_2_days, -1] = block_reward_2
bitcoin_prices_block.iloc[block_reward_2_days:block_reward_3_days, -1] = block_reward_3
bitcoin_prices_block.iloc[block_reward_3_days:, -1] = block_reward_4
bitcoin_prices_block.head()
```

Out[ ]:

	Price	block_reward

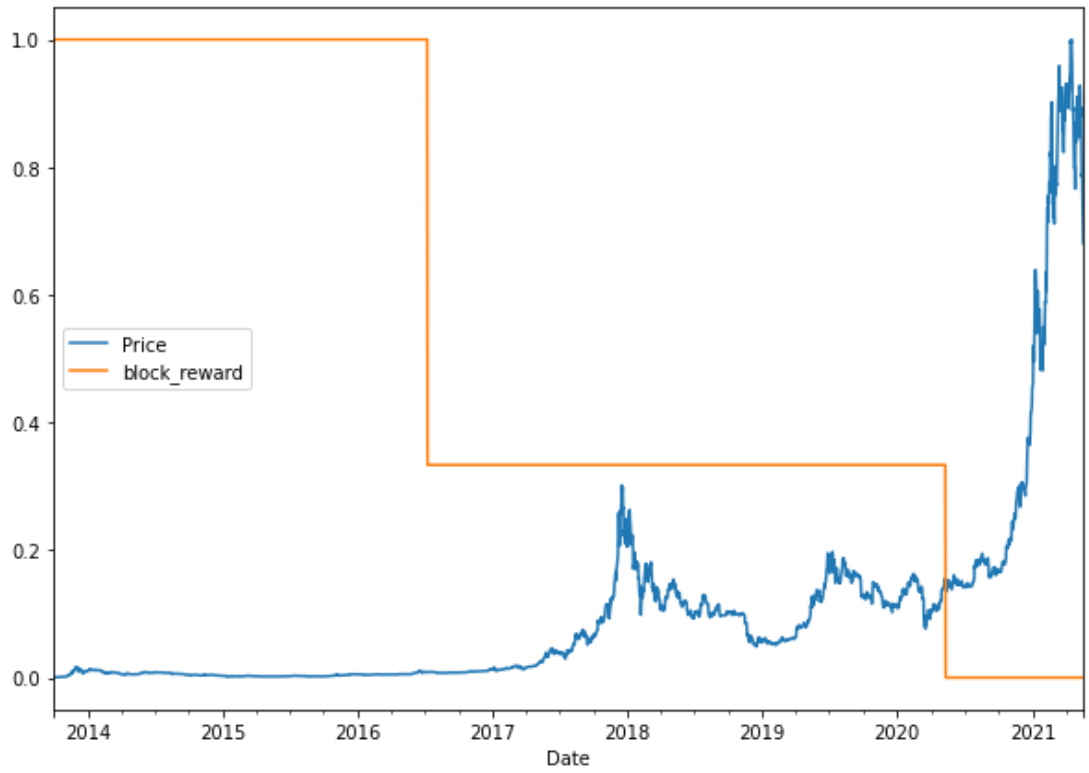
Date	Price	block_reward
2013-10-01	123.65499	25
Date		
2013-10-02	125.45500	25
2013-10-03	108.58483	25
2013-10-04	118.67466	25
2013-10-05	121.33866	25

Woohoo! We've officially added another variable to our time series data.

Let's see what it looks like.

In [ ]:

```
# Plot the block reward/price over time
# Note: Because of the different scales of our values we'll scale them to be between 0 an
d 1.
from sklearn.preprocessing import minmax_scale
scaled_price_block_df = pd.DataFrame(minmax_scale(bitcoin_prices_block[["Price", "block_
reward"]]), # we need to scale the data first
                                     columns=bitcoin_prices_block.columns,
                                     index=bitcoin_prices_block.index)
scaled_price_block_df.plot(figsize=(10, 7));
```



When we scale the block reward and the Bitcoin price, we can see the price goes up as the block reward goes down, perhaps this information will be helpful to our model's performance.

Making a windowed dataset with pandas

Previously, we used some custom made functions to window our univariate time series.

However, since we've just added another variable to our dataset, these functions won't work.

Not to worry though. Since our data is in a pandas DataFrame, we can leverage the `pandas.DataFrame.shift()` method to create a windowed multivariate time series.

The `shift()` method offsets an index by a specified number of periods.

Let's see it in action.

In [ ]:

```
# Setup dataset hyperparameters
HORIZON = 1
WINDOW_SIZE = 7
```

In [ ]:

```
# Make a copy of the Bitcoin historical data with block reward feature
bitcoin_prices_windowed = bitcoin_prices_block.copy()

# Add windowed columns
for i in range(WINDOW_SIZE): # Shift values for each step in WINDOW_SIZE
  bitcoin_prices_windowed[f"Price+{i+1}"] = bitcoin_prices_windowed["Price"].shift(perio
ds=i+1)
bitcoin_prices_windowed.head(10)
```

Out[ ]:

Date	Price	block_reward	Price+1	Price+2	Price+3	Price+4	Price+5	Price+6	Price+7
2013-10-01	123.65499	25	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2013-10-02	125.45500	25	123.65499	NaN	NaN	NaN	NaN	NaN	NaN
2013-10-03	108.58483	25	125.45500	123.65499	NaN	NaN	NaN	NaN	NaN
2013-10-04	118.67466	25	108.58483	125.45500	123.65499	NaN	NaN	NaN	NaN
2013-10-05	121.33866	25	118.67466	108.58483	125.45500	123.65499	NaN	NaN	NaN
2013-10-06	120.65533	25	121.33866	118.67466	108.58483	125.45500	123.65499	NaN	NaN
2013-10-07	121.79500	25	120.65533	121.33866	118.67466	108.58483	125.45500	123.65499	NaN
2013-10-08	123.03300	25	121.79500	120.65533	121.33866	118.67466	108.58483	125.45500	123.65499
2013-10-09	124.04900	25	123.03300	121.79500	120.65533	121.33866	118.67466	108.58483	125.45500
2013-10-10	125.96116	25	124.04900	123.03300	121.79500	120.65533	121.33866	118.67466	108.58483

Now that we've got a windowed dataset, let's separate features (`X`) from labels (`y`).

Remember in our windowed dataset, we're trying to use the previous `WINDOW_SIZE` steps to predict `HORIZON` steps.

```
Window for a week (7) to predict a horizon of 1 (multivariate time series)
WINDOW_SIZE & block_reward -> HORIZON

[0, 1, 2, 3, 4, 5, 6, block_reward] -> [7]
[1, 2, 3, 4, 5, 6, 7, block_reward] -> [8]
[2, 3, 4, 5, 6, 7, 8, block_reward] -> [9]
```

We'll also remove the `NaN` values using pandas `dropna()` method, this equivalent to starting our windowing function at `sample 0 (the first sample) + WINDOW_SIZE`.

In [ ]:

```
# Let's create X & y, remove the NaN's and convert to float32 to prevent TensorFlow error
s
X = bitcoin_prices_windowed.dropna().drop("Price", axis=1).astype(np.float32)
y = bitcoin_prices_windowed.dropna()["Price"].astype(np.float32)
X.head()
```

Out[ ]:

Date	block_reward	Price+1	Price+2	Price+3	Price+4	Price+5	Price+6	Price+7
2013-10-08	25.0	121.794998	120.655327	121.338661	118.674660	108.584831	125.455002	123.654991

Date	block_reward	Price+1	Price+2	Price+3	Price+4	Price+5	Price+6	Price+7
2013-10-09	25.0	123.032997	121.794998	120.655327	121.338661	118.674660	108.584831	125.455002
2013-10-10	25.0	124.049004	123.032997	121.794998	120.655327	121.338661	118.674660	108.584831
2013-10-11	25.0	125.961159	124.049004	123.032997	121.794998	120.655327	121.338661	118.674660
2013-10-12	25.0	125.279663	125.961159	124.049004	123.032997	121.794998	120.655327	121.338661

In [ ]:

```
# View labels
y.head()
```

Out[ ]:

```
Date
2013-10-08    123.032997
2013-10-09    124.049004
2013-10-10    125.961159
2013-10-11    125.279663
2013-10-12    125.927498
Name: Price, dtype: float32
```

What a good looking dataset, let's split it into train and test sets using an 80/20 split just as we've done before.

In [ ]:

```
# Make train and test sets
split_size = int(len(X) * 0.8)
X_train, y_train = X[:split_size], y[:split_size]
X_test, y_test = X[split_size:], y[split_size:]
len(X_train), len(y_train), len(X_test), len(y_test)
```

Out[ ]:

```
(2224, 2224, 556, 556)
```

Training and test multivariate time series datasets made! Time to build a model.

Model 6: Dense (multivariate time series)

To keep things simple, let's the `model_1` architecture and use it to train and make predictions on our multivariate time series data.

By replicating the `model_1` architecture we'll be able to see whether or not adding the block reward feature improves or detracts from model performance.

In [ ]:

```
tf.random.set_seed(42)

# Make multivariate time series model
model_6 = tf.keras.Sequential([
  layers.Dense(128, activation="relu"),
  # layers.Dense(128, activation="relu"), # adding an extra layer here should lead to bea
ting the naive model
  layers.Dense(HORIZON)
], name="model_6_dense_multivariate")

# Compile
model_6.compile(loss="mae",
                optimizer=tf.keras.optimizers.Adam())

# Fit
model_6.fit(X_train, y_train,
            epochs=100,
            batch_size=128,
            verbose=0, # only print 1 line per epoch
            validation_data=(X_test, y_test),
```

```
            callbacks=[create_model_checkpoint(model_name=model_6.name)])
```

```
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
INFO:tensorflow:Assets written to: model_experiments/model_6_dense_multivariate/assets
```

Out[ ]:

```
<keras.callbacks.History at 0x7fdceed05590>
```

Multivariate model fit!

You might've noticed that the model inferred the input shape of our data automatically (the data now has an extra feature). Often this will be the case, however, if you're running into shape issues, you can always explicitly define the input shape using `input_shape` parameter of the first layer in a model.

Time to evaluate our multivariate model.

In [ ]:

```
# Make sure best model is loaded and evaluate
model_6 = tf.keras.models.load_model("model_experiments/model_6_dense_multivariate")
model_6.evaluate(X_test, y_test)
```

```
18/18 [==============================] - 0s 2ms/step - loss: 567.5873
```

Out[ ]:

```
567.5873413085938
```

In [ ]:

```
# Make predictions on multivariate data
model_6_preds = tf.squeeze(model_6.predict(X_test))
model_6_preds[:10]
```

Out[ ]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([8836.276, 8763.8  , 9040.486, 8741.225, 8719.326, 8765.071,
       8661.102, 8496.891, 8463.231, 8521.585], dtype=float32)>
```

In [ ]:

```
# Evaluate preds
model_6_results = evaluate_preds(y_true=y_test,
                                 y_pred=model_6_preds)
model_6_results
```

Out[ ]:

```
{'mae': 567.5874,
 'mape': 2.541387,
 'mase': 0.99709386,
 'mse': 1161688.4,
 'rmse': 1077.8165}
```

Hmmm... how do these results compare to `model_1` (same window size and horizon but without the block reward feature)?

In [ ]:

```
model_1_results
```

Out[ ]:

```
{'mae': 568.95123,
 'mape': 2.5448983,
 'mase': 0.9994897,
 'mse': 1171744.0,
 'rmse': 1082.4713}
```

It looks like the adding in the block reward may have helped our model slightly.

But there a few more things we could try.

> **Resource:** For different ideas on how to improve a neural network model (from a model perspective), refer to the *Improving a model* section in notebook 02.
>
> **Exercise(s):**
>
> 1. Try adding an extra `tf.keras.layers.Dense()` layer with 128 hidden units to `model_6`, how does this effect model performance?
> 2. Is there a better way to create this model? As in, should the `block_reward` feature be bundled in with the Bitcoin historical price feature? Perhaps you could test whether building a multi-input model (e.g. one model input for Bitcoin price history and one model input for `block_reward`) works better? See *Model 4: Hybrid embedding* section of notebook 09 for an idea on how to create a multi-input model.

Model 7: N-BEATS algorithm

Time to step things up a notch.

So far we've tried a bunch of smaller models, models with only a couple of layers.

But one of the best ways to improve a model's performance is to increase the number of layers in it.

That's exactly what the N-BEATS (Neural Basis Expansion Analysis for Interpretable Time Series Forecasting) algorithm does.

The N-BEATS algorithm focuses on univariate time series problems and achieved state-of-the-art performance in the winner of the M4 competition (a forecasting competition).

For our next modelling experiment we're going to be replicating the generic architecture of the N-BEATS algorithm (see section 3.3 of the N-BEATS paper).

We're not going to go through all of the details in the paper, instead we're going to focus on:

1. **Replicating the model architecture in Figure 1 of the N-BEATS paper**

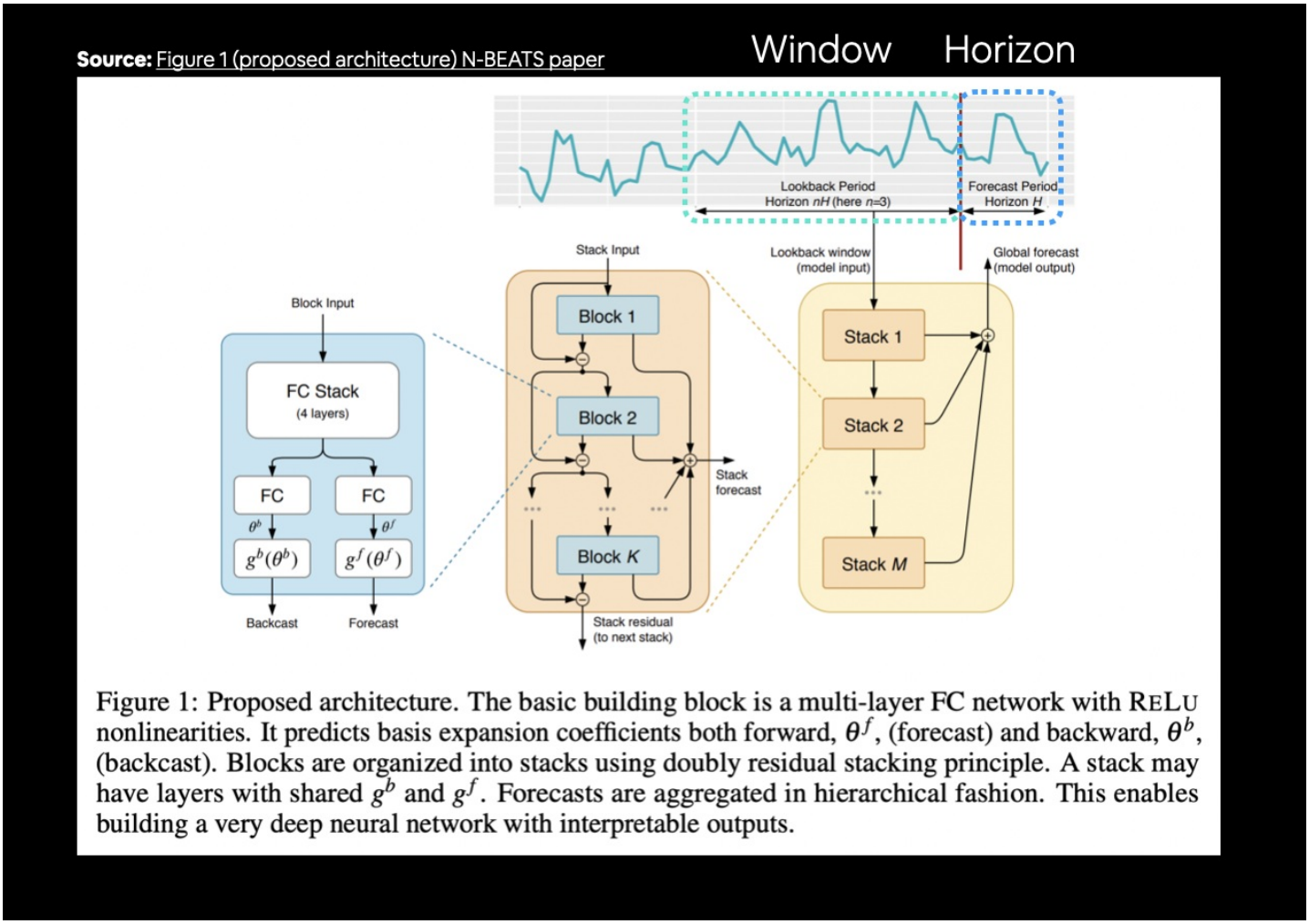


Figure 1: Proposed architecture. The basic building block is a multi-layer FC network with RELU nonlinearities. It predicts basis expansion coefficients both forward, θ^f, (forecast) and backward, θ^b, (backcast). Blocks are organized into stacks using doubly residual stacking principle. A stack may have layers with shared g^b and g^f. Forecasts are aggregated in hierarchical fashion. This enables building a very deep neural network with interpretable outputs.

N-BEATS algorithm we're going to replicate with TensorFlow with window (input) and horizon (output) annotations.

1. **Using the same hyperparameters as the paper which can be found in Appendix D of the N-BEATS paper**

Doing this will give us an opportunity to practice:

- **Creating a custom layer for the `NBeatsBlock` by subclassing `tf.keras.layers.Layer`**
  - **Creating a custom layer is helpful for when TensorFlow doesn't already have an existing implementation of a layer or if you'd like to make a layer configuration repeat a number of times (e.g. like a stack of N-BEATS blocks)**
- **Implementing a custom architecture using the Functional API**
- **Finding a paper related to our problem and seeing how it goes**

> **Note:** **As you'll see in the paper, the authors state "N-BEATS is implemented and trained in TensorFlow", that's what we'll be doing too!**

Building and testing an N-BEATS block layer

Let's start by building an N-BEATS block layer, we'll write the code first and then discuss what's going on.

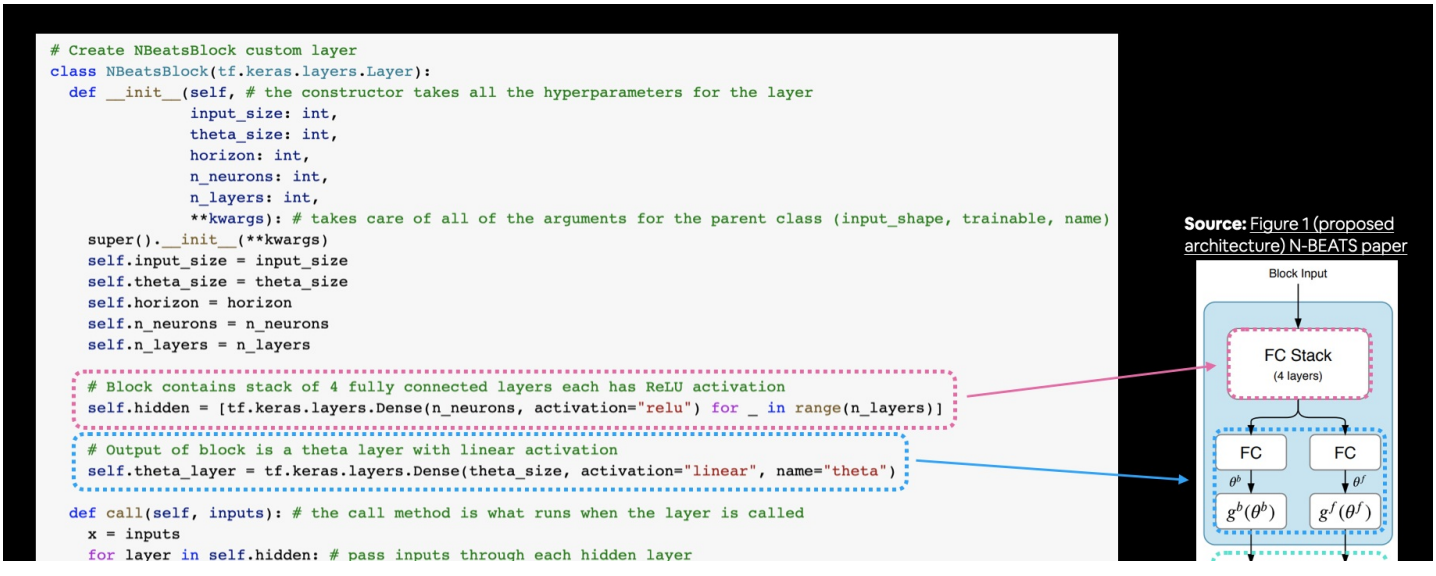
In [ ]:

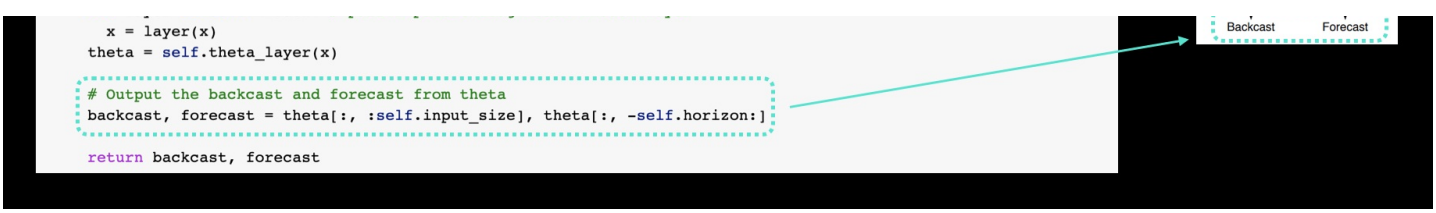
```
# Create NBeatsBlock custom layer
class NBeatsBlock(tf.keras.layers.Layer):
  def __init__(self, # the constructor takes all the hyperparameters for the layer
```

```
                 input_size: int,
                 theta_size: int,
                 horizon: int,
                 n_neurons: int,
                 n_layers: int,
                 **kwargs): # the **kwargs argument takes care of all of the arguments for
the parent class (input_shape, trainable, name)
    super().__init__(**kwargs)
    self.input_size = input_size
    self.theta_size = theta_size
    self.horizon = horizon
    self.n_neurons = n_neurons
    self.n_layers = n_layers

    # Block contains stack of 4 fully connected layers each has ReLU activation
    self.hidden = [tf.keras.layers.Dense(n_neurons, activation="relu") for _ in range(n_
layers)]
    # Output of block is a theta layer with linear activation
    self.theta_layer = tf.keras.layers.Dense(theta_size, activation="linear", name="thet
a")

  def call(self, inputs): # the call method is what runs when the layer is called
    x = inputs
    for layer in self.hidden: # pass inputs through each hidden layer
      x = layer(x)
    theta = self.theta_layer(x)
    # Output the backcast and forecast from theta
    backcast, forecast = theta[:, :self.input_size], theta[:, -self.horizon:]
    return backcast, forecast
```

Setting up the `NBeatsBlock` custom layer we see:

- **The class inherits from `tf.keras.layers.Layer` (this gives it all of the methods assosciated with `tf.keras.layers.Layer`)**
- **The constructor (`def __init__(...)`) takes all of the layer hyperparameters as well as the `**kwargs` argument**
  - **The `**kwargs` argument takes care of all of the hyperparameters which aren't mentioned in the constructor such as, `input_shape`, `trainable` and `name`**
- **In the constructor, the block architecture layers are created:**
  - **The hidden layers are created as a stack of fully connected with `n_nuerons` hidden units layers with ReLU activation**
  - **The theta layer uses `theta_size` hidden units as well as linear activation**
- **The `call()` method is what is run when the layer is called:**
  - **It first passes the inputs (the historical Bitcoin data) through each of the hidden layers (a stack of fully connected layers with ReLU activation)**
  - **After the inputs have been through each of the fully connected layers, they get passed through the theta layer where the backcast (backwards predictions, shape: `input_size`) and forecast (forward predictions, shape: `horizon`) are returned**





Using TensorFlow layer subclassing to replicate the N-BEATS basic block. See section 3.1 of N-BEATS paper for details.

Let's see our block replica in action by together by creating a toy version of `NBeatsBlock`.

> **Resource: Much of the creation of the time series materials (the ones you're going through now), including replicating the N-BEATS algorithm were streamed live on Twitch. If you'd like to see replays of how the algorithm was replicated, check out the *Time series research and TensorFlow course material creation playlist* on the Daniel Bourke arXiv YouTube channel.**

In [ ]:

```
# Set up dummy NBeatsBlock layer to represent inputs and outputs
dummy_nbeats_block_layer = NBeatsBlock(input_size=WINDOW_SIZE,
                                       theta_size=WINDOW_SIZE+HORIZON, # backcast + fore
cast
                                       horizon=HORIZON,
                                       n_neurons=128,
                                       n_layers=4)
```

In [ ]:

```
# Create dummy inputs (have to be same size as input_size)
dummy_inputs = tf.expand_dims(tf.range(WINDOW_SIZE) + 1, axis=0) # input shape to the mo
del has to reflect Dense layer input requirements (ndim=2)
dummy_inputs
```

Out[ ]:

```
<tf.Tensor: shape=(1, 7), dtype=int32, numpy=array([[1, 2, 3, 4, 5, 6, 7]], dtype=int32)>
```

In [ ]:

```
# Pass dummy inputs to dummy NBeatsBlock layer
backcast, forecast = dummy_nbeats_block_layer(dummy_inputs)
# These are the activation outputs of the theta layer (they'll be random due to no traini
ng of the model)
print(f"Backcast: {tf.squeeze(backcast.numpy())}")
print(f"Forecast: {tf.squeeze(forecast.numpy())}")
```

```
Backcast: [ 0.19014978  0.83798355 -0.32870018  0.25159916 -0.47540277 -0.77836645
 -0.5299447 ]
Forecast: -0.7554212808609009
```

Preparing data for the N-BEATS algorithm using `tf.data`

We've got the basic building block for the N-BEATS architecture ready to go.

But before we use it to replicate the entire N-BEATS generic architecture, let's create some data.

This time, because we're going to be using a larger model architecture, to ensure our model training runs as fast as possible, we'll setup our datasets using the `tf.data` API.

And because the N-BEATS algorithm is focused on univariate time series, we'll start by making training and test windowed datasets of Bitcoin prices (just as we've done above).

In [ ]:

```
HORIZON = 1 # how far to predict forward
WINDOW_SIZE = 7 # how far to lookback
```

In [ ]:

```
# Create NBEATS data inputs (NBEATS works with univariate time series)
bitcoin_prices.head()
```

Out[ ]:

	Price
Date	
2013-10-01	123.65499
2013-10-02	125.45500
2013-10-03	108.58483
2013-10-04	118.67466
2013-10-05	121.33866

In [ ]:

```
# Add windowed columns
bitcoin_prices_nbeats = bitcoin_prices.copy()
for i in range(WINDOW_SIZE):
  bitcoin_prices_nbeats[f"Price+{i+1}"] = bitcoin_prices_nbeats["Price"].shift(periods=i
+1)
bitcoin_prices_nbeats.dropna().head()
```

Out[ ]:

	Price	Price+1	Price+2	Price+3	Price+4	Price+5	Price+6	Price+7
Date								
2013-10-08	123.03300	121.79500	120.65533	121.33866	118.67466	108.58483	125.45500	123.65499
2013-10-09	124.04900	123.03300	121.79500	120.65533	121.33866	118.67466	108.58483	125.45500
2013-10-10	125.96116	124.04900	123.03300	121.79500	120.65533	121.33866	118.67466	108.58483
2013-10-11	125.27966	125.96116	124.04900	123.03300	121.79500	120.65533	121.33866	118.67466
2013-10-12	125.92750	125.27966	125.96116	124.04900	123.03300	121.79500	120.65533	121.33866

In [ ]:

```
# Make features and labels
X = bitcoin_prices_nbeats.dropna().drop("Price", axis=1)
y = bitcoin_prices_nbeats.dropna()["Price"]

# Make train and test sets
split_size = int(len(X) * 0.8)
X_train, y_train = X[:split_size], y[:split_size]
X_test, y_test = X[split_size:], y[split_size:]
len(X_train), len(y_train), len(X_test), len(y_test)
```

Out[ ]:

```
(2224, 2224, 556, 556)
```

Train and test sets ready to go!

Now let's convert them into TensorFlow `tf.data.Dataset`'s to ensure they run as fast as possible whilst training.

We'll do this by:

1. **Turning the arrays in tensor Datasets using `tf.data.Dataset.from_tensor_slices()`**
    - **Note: `from_tensor_slices()` works best when your data fits in memory, for extremely large datasets, you'll want to look into using the `TFRecord` format**
2. **Combine the labels and features tensors into a Dataset using `tf.data.Dataset.zip()`**

2. Combine the labels and features tensors into a Dataset using `tf.data.Dataset.zip()`
3. **Batch and prefetch the Datasets using `batch()` and `prefetch()`**

    - **Batching and prefetching ensures the loading time from CPU (preparing data) to GPU (computing on data) is as small as possible**

> 🔑 **Resource: For more on building highly performant TensorFlow data pipelines, I'd recommend reading through the *Better performance with the tf.data API* guide.**

In [ ]:

```
# 1. Turn train and test arrays into tensor Datasets
train_features_dataset = tf.data.Dataset.from_tensor_slices(X_train)
train_labels_dataset = tf.data.Dataset.from_tensor_slices(y_train)

test_features_dataset = tf.data.Dataset.from_tensor_slices(X_test)
test_labels_dataset = tf.data.Dataset.from_tensor_slices(y_test)

# 2. Combine features & labels
train_dataset = tf.data.Dataset.zip((train_features_dataset, train_labels_dataset))
test_dataset = tf.data.Dataset.zip((test_features_dataset, test_labels_dataset))

# 3. Batch and prefetch for optimal performance
BATCH_SIZE = 1024 # taken from Appendix D in N-BEATS paper
train_dataset = train_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)
test_dataset = test_dataset.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

train_dataset, test_dataset
```

Out[ ]:

```
(<PrefetchDataset shapes: ((None, 7), (None,)), types: (tf.float64, tf.float64)>,
 <PrefetchDataset shapes: ((None, 7), (None,)), types: (tf.float64, tf.float64)>)
```

Data prepared! Notice the input shape for the features `(None, 7)`, the `None` leaves space for the batch size where as the `7` represents the `WINDOW_SIZE`.

Time to get create the N-BEATS architecture.

Setting up hyperparameters for N-BEATS algorithm

Ho ho, would you look at that! Datasets ready, model building block ready, what'd you say we put things together?

Good idea.

Okay.

Let's go.

To begin, we'll create variables for each of the hyperparameters we'll be using for our N-BEATS replica.

> 🔑 **Resource: The following hyperparameters are taken from Figure 1 and Table 18/Appendix D of the N-BEATS paper.**

Table 18: Settings of hyperparameters across subsets of M4, M3, TOURISM datasets.

	M4						M3				TOURISM		
	Yly	Qly	Mly	Wly	Dly	Hly	Yly	Qly	Mly	Other	Yly	Qly	Mly
Parameter	N-BEATS-I												
L_H	1.5	1.5	1.5	10	10	10	20	5	5	20	20	10	20
Iterations	15K	15K	15K	5K	5K	5K	50	6K	6K	250	30	500	300
Losses	sMAPE/MAPE/MASE						sMAPE/MAPE/MASE				MAPE		

S-width	2048
S-blocks	3
S-block-layers	4
T-width	256
T-degree	2
T-blocks	3
T-block-layers	4
Sharing	STACK LEVEL
Lookback period	$2H, 3H, 4H, 5H, 6H, 7H$
Batch	1024

Parameter	N-BEATS-G												
L_H	1.5	1.5	1.5	10	10	10	20	20	20	10	5	10	20
Iterations	15K	15K	15K	5K	5K	5K	20	250	10K	250	30	100	100
Losses	sMAPE/MAPE/MASE						sMAPE/MAPE/MASE				MAPE		
Width	512												
Blocks	1												
Block-layers	4												
Stacks	30												
Sharing	NO												
Lookback period	$2H, 3H, 4H, 5H, 6H, 7H$												
Batch	1024												

Table 18 from [N-BEATS paper](#) describing the hyperparameters used for the different variants of N-BEATS. We're using N-BEATS-G which stands for the generic version of N-BEATS.

> **Note: If you see variables in a machine learning example in all caps, such as "`N_EPOCHS = 100`", these variables are often hyperparameters which are used through the example. You'll usually see them instantiated towards the start of an experiment and then used throughout.**

In [ ]:

```
# Values from N-BEATS paper Figure 1 and Table 18/Appendix D
N_EPOCHS = 5000 # called "Iterations" in Table 18
N_NEURONS = 512 # called "Width" in Table 18
N_LAYERS = 4
N_STACKS = 30

INPUT_SIZE = WINDOW_SIZE * HORIZON # called "Lookback" in Table 18
THETA_SIZE = INPUT_SIZE + HORIZON

INPUT_SIZE, THETA_SIZE
```

Out[ ]:

```
(7, 8)
```

Getting ready for residual connections

Beautiful! Hyperparameters ready, now before we create the N-BEATS model, there are two layers to go through which play a large roll in the architecture.

They're what make N-BEATS double residual stacking (section 3.2 of the [N-BEATS paper](#)) possible:

- `tf.keras.layers.subtract(inputs)` **- subtracts list of input tensors from each other**
- `tf.keras.layers.add(inputs)` **- adds list of input tensors to each other**

Let's try them out.

In [ ]:

```
# Make tensors
tensor_1 = tf.range(10) + 10
tensor_2 = tf.range(10)

# Subtract
```

```
subtracted = layers.subtract([tensor_1, tensor_2])

# Add
added = layers.add([tensor_1, tensor_2])

print(f"Input tensors: {tensor_1.numpy()} & {tensor_2.numpy()}")
print(f"Subtracted: {subtracted.numpy()}")
print(f"Added: {added.numpy()}")
```

```
Input tensors: [10 11 12 13 14 15 16 17 18 19] & [0 1 2 3 4 5 6 7 8 9]
Subtracted: [10 10 10 10 10 10 10 10 10 10]
Added: [10 12 14 16 18 20 22 24 26 28]
```

Both of these layer functions are straight-forward, subtract or add together their inputs.

And as mentioned before, they're what powers N-BEATS double residual stacking.

The power of residual stacking or residual connections was revealed in Deep Residual Learning for Image Recognition where the authors were able to build a deeper but less complex neural network (this is what introduced the popular ResNet architecture) than previous attempts.

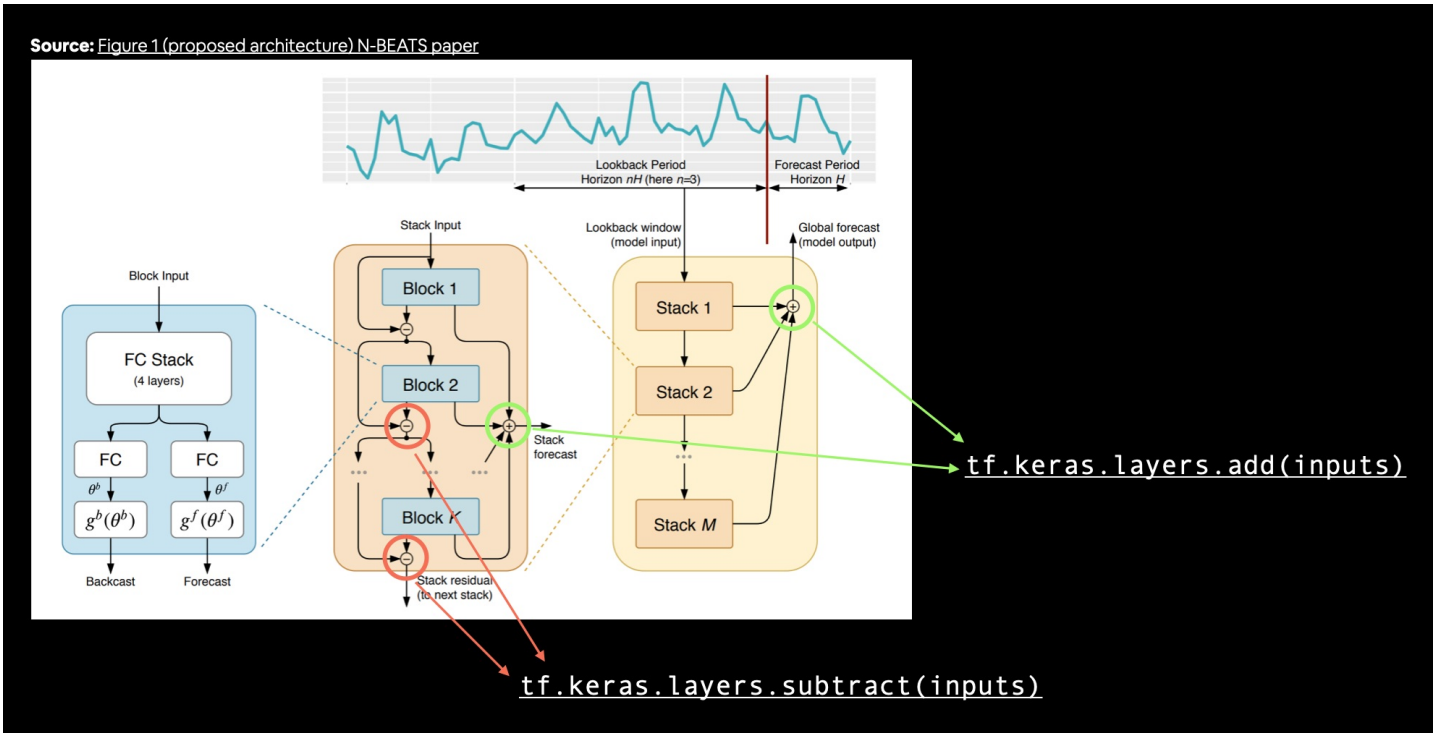
This deeper neural network led to state of the art results on the ImageNet challenge in 2015 and different versions of residual connections have been present in deep learning ever since.

> **What is a residual connection?**

A residual connection (also called skip connections) involves a deeper neural network layer receiving the outputs as well as the inputs of a shallower neural network layer.

In the case of N-BEATS, the architecture uses residual connections which:

- **Subtract the backcast outputs from a previous block from the backcast inputs to the current block**
- **Add the forecast outputs from all blocks together in a stack**



Annotated version of Figure 1 from the N-BEATS paper highlighting the double residual stacking (section 3.2) of the architecture. Backcast residuals of each block are subtracted from each other and used as the input to the next block where as the forecasts of each block are added together to become the stack forecast.

> **What are the benefits of residual connections?**

In practice, residual connections have been beneficial for training deeper models (N-BEATS reaches ~150 layers, also see "These approaches provide clear advantages in improving the trainability of deep architectures" in section 3.2 of the N-BEATS paper).

It's thought that they help avoid the problem of vanishing gradients (patterns learned by a neural network not being passed through to deeper layers).

Building, compiling and fitting the N-BEATS algorithm

Okay, we've finally got all of the pieces of the puzzle ready for building and training the N-BEATS algorithm.

We'll do so by going through the following:

1. **Setup an instance of the N-BEATS block layer using `NBeatsBlock` (this'll be the initial block used for the network, the rest will be created as part of stacks)**
2. **Create an input layer for the N-BEATS stack (we'll be using the Keras Functional API for this)**
3. **Make the initial backcast and forecasts for the model with the layer created in (1)**
4. **Use a for loop to create stacks of block layers**
5. **Use the NBeatsBlock class within the for loop created in (4) to create blocks which return backcasts and block-level forecasts**
6. **Create the double residual stacking using subtract and add layers**
7. **Put the model inputs and outputs together using `tf.keras.Model()`**
8. **Compile the model with MAE loss (the paper uses multiple losses but we'll use MAE to keep it inline with our other models) and Adam optimizer with default settings as per section 5.2 of N-BEATS paper)**
9. **Fit the N-BEATS model for 5000 epochs and since it's fitting for so many epochs, we'll use a couple of callbacks:**
   - **`tf.keras.callbacks.EarlyStopping()` - stop the model from training if it doesn't improve validation loss for 200 epochs and restore the best performing weights using `restore_best_weights=True` (this'll prevent the model from training for loooonggggggg period of time without improvement)**
   - **`tf.keras.callbacks.ReduceLROnPlateau()` - if the model's validation loss doesn't improve for 100 epochs, reduce the learning rate by 10x to try and help it make incremental improvements (the smaller the learning rate, the smaller updates a model tries to make)**

Woah. A bunch of steps. But I'm sure you're up to it.

Let's do it!

In [ ]:

```
%%time

tf.random.set_seed(42)

# 1. Setup N-BEATS Block layer
nbeats_block_layer = NBeatsBlock(input_size=INPUT_SIZE,
                                 theta_size=THETA_SIZE,
                                 horizon=HORIZON,
                                 n_neurons=N_NEURONS,
                                 n_layers=N_LAYERS,
                                 name="InitialBlock")

# 2. Create input to stacks
stack_input = layers.Input(shape=(INPUT_SIZE), name="stack_input")

# 3. Create initial backcast and forecast input (backwards predictions are referred to as residuals in the paper)
backcast, forecast = nbeats_block_layer(stack_input)
# Add in subtraction residual link, thank you to: https://github.com/mrdbourke/tensorflow-deep-learning/discussions/174
residuals = layers.subtract([stack_input, backcast], name=f"subtract_00")

# 4. Create stacks of blocks
for i, _ in enumerate(range(N_STACKS-1)): # first stack is already creted in (3)

  # 5. Use the NBeatsBlock to calculate the backcast as well as block forecast
  backcast, block_forecast = NBeatsBlock(
      input_size=INPUT_SIZE,
      theta_size=THETA_SIZE,
```

```
      horizon=HORIZON,
      n_neurons=N_NEURONS,
      n_layers=N_LAYERS,
      name=f"NBeatsBlock_{i}"
  )(residuals) # pass it in residuals (the backcast)

  # 6. Create the double residual stacking
  residuals = layers.subtract([residuals, backcast], name=f"subtract_{i}")
  forecast = layers.add([forecast, block_forecast], name=f"add_{i}")

# 7. Put the stack model together
model_7 = tf.keras.Model(inputs=stack_input,
                         outputs=forecast,
                         name="model_7_N-BEATS")

# 8. Compile with MAE loss and Adam optimizer
model_7.compile(loss="mae",
                optimizer=tf.keras.optimizers.Adam(0.001),
                metrics=["mae", "mse"])

# 9. Fit the model with EarlyStopping and ReduceLROnPlateau callbacks
model_7.fit(train_dataset,
            epochs=N_EPOCHS,
            validation_data=test_dataset,
            verbose=0, # prevent large amounts of training outputs
            # callbacks=[create_model_checkpoint(model_name=stack_model.name)] # saving
model every epoch consumes far too much time
            callbacks=[tf.keras.callbacks.EarlyStopping(monitor="val_loss", patience=200
, restore_best_weights=True),
                       tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss", patience=
100, verbose=1)])
```

```
Epoch 00328: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00428: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
CPU times: user 1min 23s, sys: 4.58 s, total: 1min 28s
Wall time: 3min 44s
```

And would you look at that! N-BEATS algorithm fit to our Bitcoin historical data.

How did it perform?

In [ ]:

```
# Evaluate N-BEATS model on the test dataset
model_7.evaluate(test_dataset)
```

```
1/1 [==============================] - 0s 46ms/step - loss: 585.4998 - mae: 585.4998 - ms
e: 1179491.5000
```

Out[ ]:

```
[585.4998168945312, 585.4998168945312, 1179491.5]
```

In [ ]:

```
# Make predictions with N-BEATS model
model_7_preds = make_preds(model_7, test_dataset)
model_7_preds[:10]
```

Out[ ]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([8908.059, 8854.672, 8990.933, 8759.821, 8819.711, 8774.012,
       8604.187, 8547.038, 8495.928, 8489.514], dtype=float32)>
```

In [ ]:

```
# Evaluate N-BEATS model predictions
model_7_results = evaluate_preds(y_true=y_test,
                                 y_pred=model_7_preds)
model_7_results
```

Out[ ]:

```
{'mae': 585.4998,
 'mape': 2.7445195,
 'mase': 1.028561,
 'mse': 1179491.5,
 'rmse': 1086.044}
```

Woah... even with all of those special layers and hand-crafted network, it looks like the N-BEATS model doesn't perform as well as `model_1` or the original naive forecast.

This goes to show the power of smaller networks as well as the fact not all larger models are better suited for a certain type of data.

Plotting the N-BEATS architecture we've created

You know what would be cool?

If we could plot the N-BEATS model we've crafted.

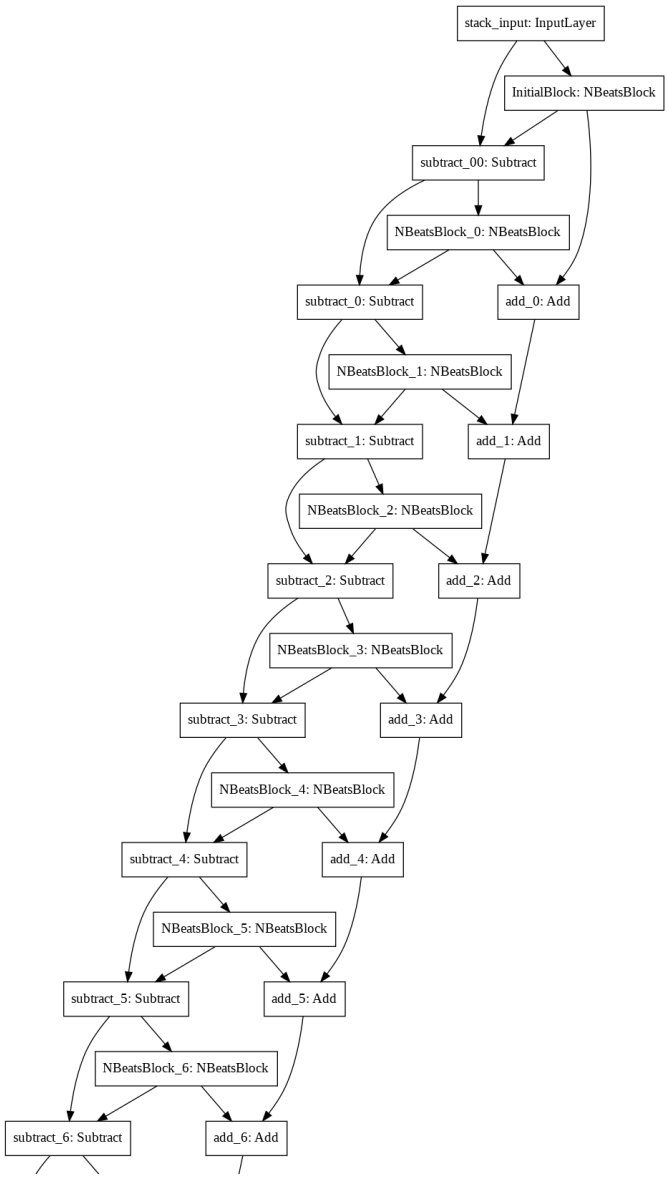
Well it turns out we can using `tensorflow.keras.utils.plot_model()`.

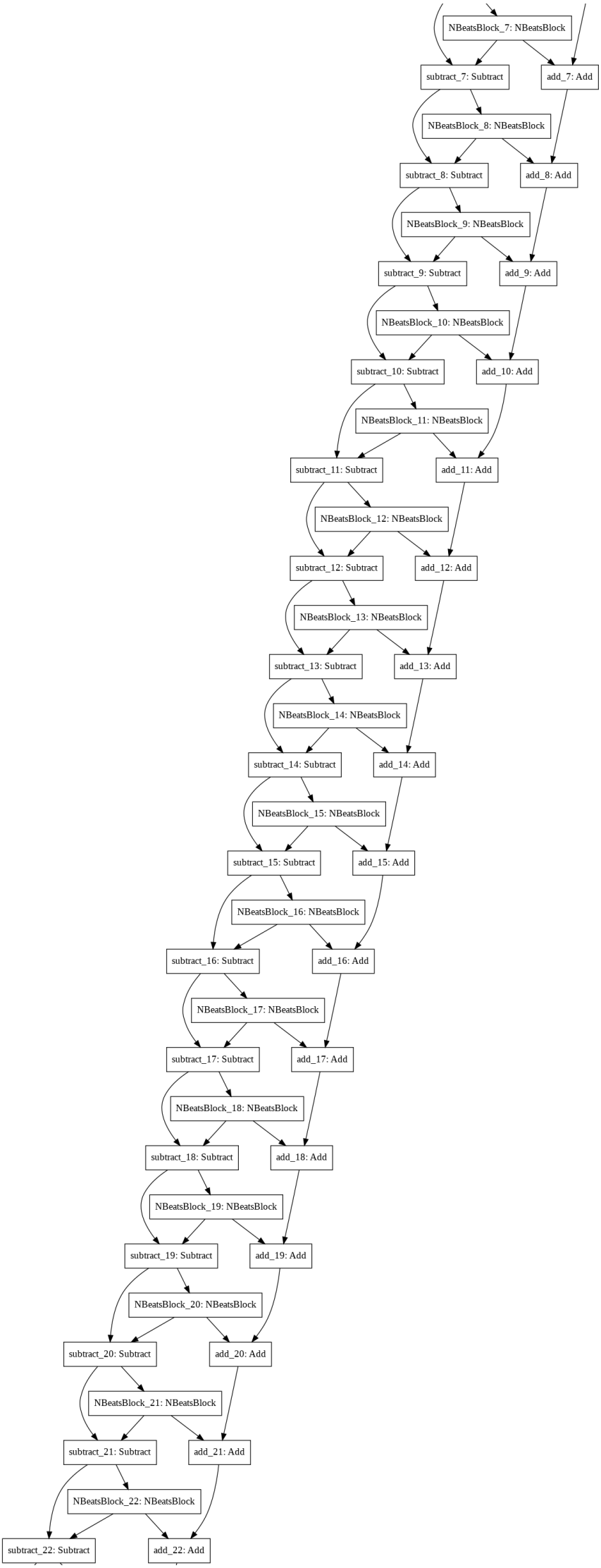
Let's see what it looks like.

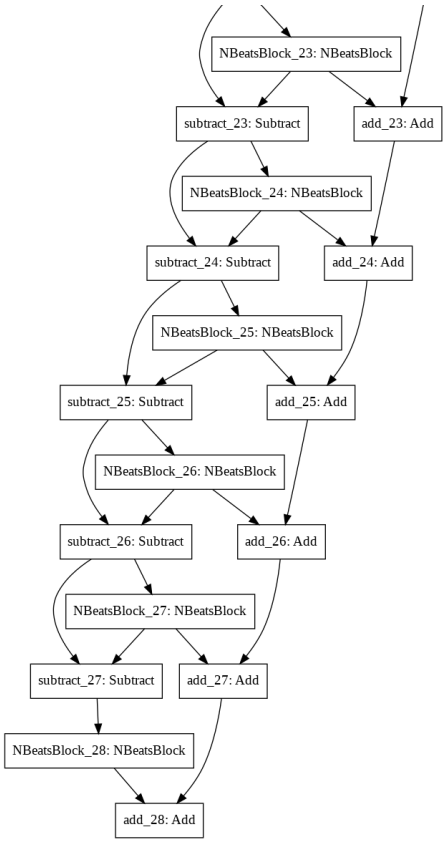
In [ ]:

```
# Plot the N-BEATS model and inspect the architecture
from tensorflow.keras.utils import plot_model
plot_model(model_7)
```

Out[ ]:

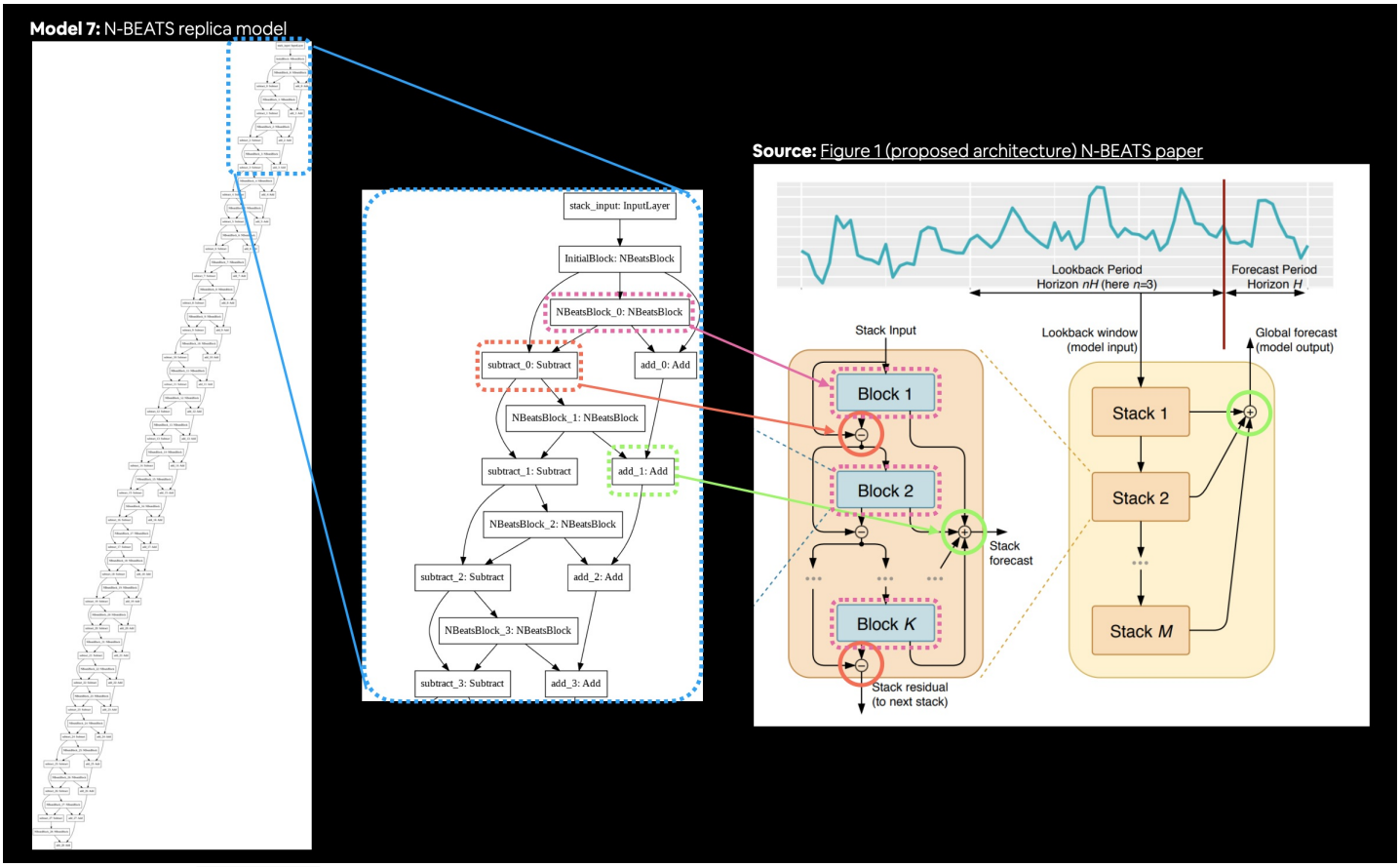



NBeatsBlock_7: NBeatsBlock
subtract_7: Subtract
add_7: Add
NBeatsBlock_8: NBeatsBlock
subtract_8: Subtract
add_8: Add
NBeatsBlock_9: NBeatsBlock
subtract_9: Subtract
add_9: Add
NBeatsBlock_10: NBeatsBlock
subtract_10: Subtract
add_10: Add
NBeatsBlock_11: NBeatsBlock
subtract_11: Subtract
add_11: Add
NBeatsBlock_12: NBeatsBlock
subtract_12: Subtract
add_12: Add
NBeatsBlock_13: NBeatsBlock
subtract_13: Subtract
add_13: Add
NBeatsBlock_14: NBeatsBlock
subtract_14: Subtract
add_14: Add
NBeatsBlock_15: NBeatsBlock
subtract_15: Subtract
add_15: Add
NBeatsBlock_16: NBeatsBlock
subtract_16: Subtract
add_16: Add
NBeatsBlock_17: NBeatsBlock
subtract_17: Subtract
add_17: Add
NBeatsBlock_18: NBeatsBlock
subtract_18: Subtract
add_18: Add
NBeatsBlock_19: NBeatsBlock
subtract_19: Subtract
add_19: Add
NBeatsBlock_20: NBeatsBlock
subtract_20: Subtract
add_20: Add
NBeatsBlock_21: NBeatsBlock
subtract_21: Subtract
add_21: Add
NBeatsBlock_22: NBeatsBlock
subtract_22: Subtract
add_22: Add



Now that is one good looking model!

It even looks similar to the model shown in Figure 1 of the N-BEATS paper.



Comparison of `model_7` (N-BEATS replica model make with Keras Functional API) versus actual N-BEATS architecture diagram.

Looks like our Functional API usage did the trick!

> **Note:** Our N-BEATS model replicates the N-BEATS **generic architecture**, the training setups are largely the same, except for the N-BEATS paper used an ensemble of models to make predictions (multiple different loss functions and multiple different lookback windows), see Table 18 of the N-BEATS paper for more. An extension could be to setup this kind of training regime and see if it

> improves performance.

How about we try and save our version of the N-BEATS model?

In [ ]:

```
# This will error out unless a "get_config()" method is implemented - this could be extra curriculum
model_7.save(model_7.name)
```

```
WARNING:absl:Found untraced functions such as theta_layer_call_and_return_conditional_losses, theta_layer_call_fn, theta_layer_call_and_return_conditional_losses, theta_layer_call_fn, theta_layer_call_and_return_conditional_losses while saving (showing 5 of 750). These functions will not be directly callable after loading.
```

```
INFO:tensorflow:Assets written to: model_7_N-BEATS/assets
```

```
INFO:tensorflow:Assets written to: model_7_N-BEATS/assets
/usr/local/lib/python3.7/dist-packages/keras/utils/generic_utils.py:497: CustomMaskWarning: Custom mask layers require a config and must override get_config. When loading, the custom mask layer must be passed to the custom_objects argument.
  category=CustomMaskWarning)
```

You'll notice a warning appears telling us to fully save our model correctly we need to implement a `get_config()` method in our custom layer class.

> 🔑 **Resource:** If you would like to save and load the N-BEATS model or any other custom or subclassed layer/model configuration, you should overwrite the `get_config()` and optionally `from_config()` methods. See the TensorFlow Custom Objects documentation for more.

Model 8: Creating an ensemble (stacking different models together)

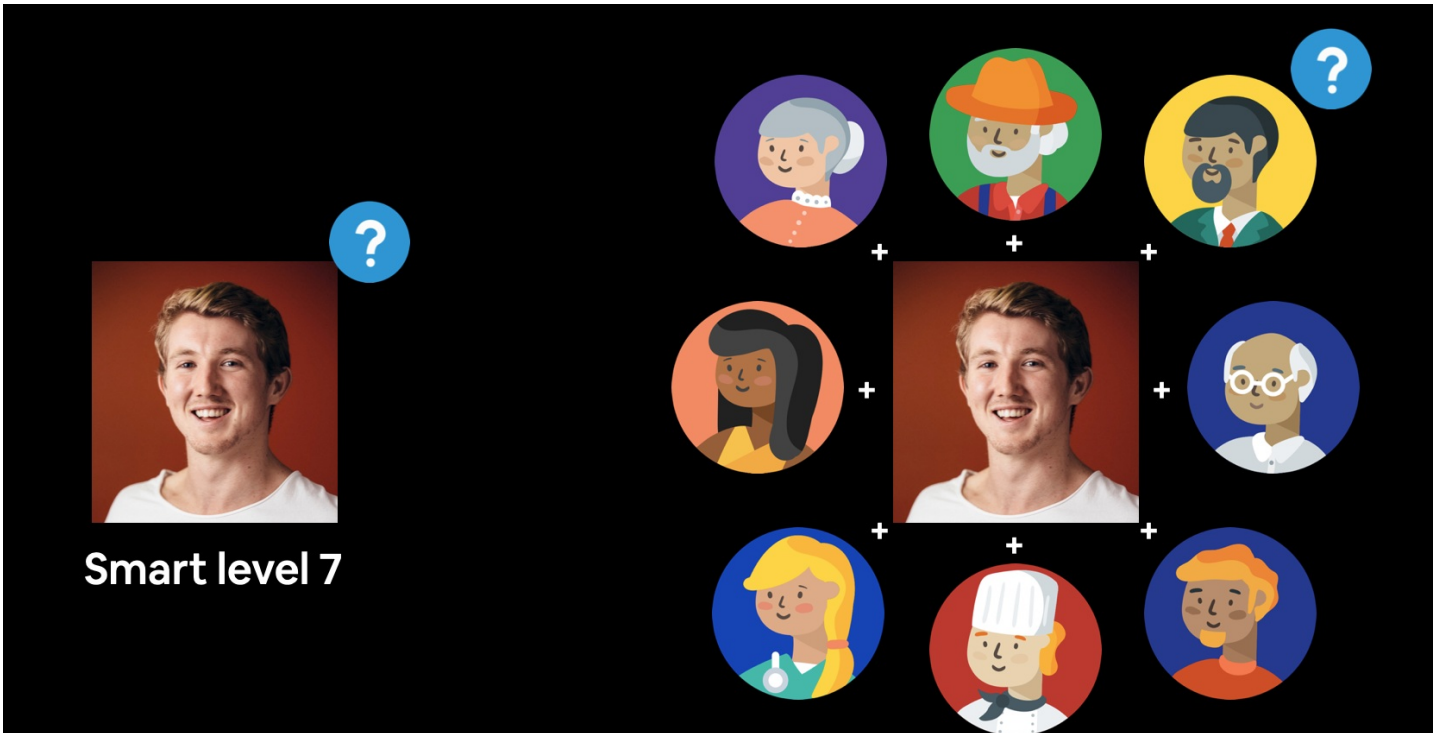
After all that effort, the N-BEATS algorithm's performance was underwhelming.

But again, this is part of the parcel of machine learning. Not everything will work.

That's when we refer back to the motto: experiment, experiment, experiment.

Our next experiment is creating an ensemble of models.

An ensemble involves training and combining multiple different models on the same problem. Ensemble models are often the types of models you'll see winning data science competitions on websites like Kaggle.



Example of the power of ensembling. One Daniel model makes a decision with a smart level of 7 but when a Daniel model teams up with multiple different people, together (ensembled) they make a decision with a smart level of 10. The key here is combining the decision power of people with different backgrounds, if you combined multiple Daniel models, you'd end up with an average smart level of 7. Note: smart level is not an actual measurement of decision making, it is for demonstration purposes only.

For example, in the N-BEATS paper, they trained an ensemble of models (180 in total, see section 3.4) to achieve the results they did using a combination of:

- **Different loss functions (sMAPE, MASE and MAPE)**
- **Different window sizes (2 x horizon, 3 x horizon, 4 x horizon...)**

The benefit of ensembling models is you get the "decision of the crowd effect". Rather than relying on a single model's predictions, you can take the average or median of many different models .

The keyword being: different.

It wouldn't make sense to train the same model 10 times on the same data and then average the predictions.

Fortunately, due to their random initialization, even deep learning models with the same architecture can produce different results.

What I mean by this is each time you create a deep learning model, it starts with random patterns (weights & biases) and then it adjusts these random patterns to better suit the dataset it's being trained on.

However, the process it adjusts these patterns is often a form of guided randomness as well (the SGD optimizer stands for stochastic or random gradient descent).

To create our ensemble models we're going to be using a combination of:

- **Different loss functions (MAE, MSE, MAPE)**
- **Randomly initialized models**

Essentially, we'll be creating a suite of different models all attempting to model the same data.

And hopefully the combined predictive power of each model is better than a single model on its own.

Let's find out!

We'll start by creating a function to produce a list of different models trained with different loss functions. Each layer in the ensemble models will be initialized with a random normal (Gaussian) distribution using He normal initialization, this'll help estimating the prediction intervals later on.

> **Note: In your machine leanring experiments, you may have already dealt with examples of ensemble models. Algorithms such as the random forest model are a form of ensemble, it uses a number of randomly created decision trees where each individual tree may perform poorly but when combined gives great results.**

Constructing and fitting an ensemble of models (using different loss functions)

In [ ]:

```
def get_ensemble_models(horizon=HORIZON,
                        train_data=train_dataset,
                        test_data=test_dataset,
                        num_iter=10,
                        num_epochs=100,
                        loss_fns=["mae", "mse", "mape"]):
  """
  Returns a list of num_iter models each trained on MAE, MSE and MAPE loss.
```

```
  For example, if num_iter=10, a list of 30 trained models will be returned:
  10 * len(["mae", "mse", "mape"]).
  """
  # Make empty list for trained ensemble models
  ensemble_models = []

  # Create num_iter number of models per loss function
  for i in range(num_iter):
    # Build and fit a new model with a different loss function
    for loss_function in loss_fns:
      print(f"Optimizing model by reducing: {loss_function} for {num_epochs} epochs, mode
l number: {i}")

      # Construct a simple model (similar to model_1)
      model = tf.keras.Sequential([
        # Initialize layers with normal (Gaussian) distribution so we can use the models
for prediction
        # interval estimation later: https://www.tensorflow.org/api_docs/python/tf/keras/
initializers/HeNormal
        layers.Dense(128, kernel_initializer="he_normal", activation="relu"),
        layers.Dense(128, kernel_initializer="he_normal", activation="relu"),
        layers.Dense(HORIZON)
      ])

      # Compile simple model with current loss function
      model.compile(loss=loss_function,
                    optimizer=tf.keras.optimizers.Adam(),
                    metrics=["mae", "mse"])

      # Fit model
      model.fit(train_data,
                epochs=num_epochs,
                verbose=0,
                validation_data=test_data,
                # Add callbacks to prevent training from going/stalling for too long
                callbacks=[tf.keras.callbacks.EarlyStopping(monitor="val_loss",
                                                            patience=200,
                                                            restore_best_weights=True),
                           tf.keras.callbacks.ReduceLROnPlateau(monitor="val_loss",
                                                                patience=100,
                                                                verbose=1)])

      # Append fitted model to list of ensemble models
      ensemble_models.append(model)

  return ensemble_models # return list of trained models
```

Ensemble model creator function created!

Let's try it out by running `num_iter=5` runs for 1000 epochs. This will result in 15 total models (5 for each different loss function).

Of course, these numbers could be tweaked to create more models trained for longer.

> **Note: With ensembles, you'll generally find more total models means better performance. However, this comes with the tradeoff of having to train more models (longer training time) and make predictions with more models (longer prediction time).**

In [ ]:

```
%%time
# Get list of trained ensemble models
ensemble_models = get_ensemble_models(num_iter=5,
                                      num_epochs=1000)
```

```
Optimizing model by reducing: mae for 1000 epochs, model number: 0

Epoch 00794: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
```

```
Epoch 00928: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
Optimizing model by reducing: mse for 1000 epochs, model number: 0

Epoch 00591: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00707: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.

Epoch 00807: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.
Optimizing model by reducing: mape for 1000 epochs, model number: 0

Epoch 00165: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00282: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.

Epoch 00382: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.
Optimizing model by reducing: mae for 1000 epochs, model number: 1
Optimizing model by reducing: mse for 1000 epochs, model number: 1

Epoch 00409: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00509: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
Optimizing model by reducing: mape for 1000 epochs, model number: 1

Epoch 00185: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00726: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.

Epoch 00826: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.
Optimizing model by reducing: mae for 1000 epochs, model number: 2
Optimizing model by reducing: mse for 1000 epochs, model number: 2
Optimizing model by reducing: mape for 1000 epochs, model number: 2

Epoch 00241: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00341: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
Optimizing model by reducing: mae for 1000 epochs, model number: 3

Epoch 00572: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.
Optimizing model by reducing: mse for 1000 epochs, model number: 3

Epoch 00304: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00607: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.

Epoch 00707: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.
Optimizing model by reducing: mape for 1000 epochs, model number: 3

Epoch 00301: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00401: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
Optimizing model by reducing: mae for 1000 epochs, model number: 4
Optimizing model by reducing: mse for 1000 epochs, model number: 4

Epoch 00640: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00740: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.
Optimizing model by reducing: mape for 1000 epochs, model number: 4

Epoch 00132: ReduceLROnPlateau reducing learning rate to 0.00010000000474974513.

Epoch 00609: ReduceLROnPlateau reducing learning rate to 1.0000000474974514e-05.

Epoch 00709: ReduceLROnPlateau reducing learning rate to 1.0000000656873453e-06.
CPU times: user 6min 24s, sys: 38.6 s, total: 7min 3s
Wall time: 7min 58s
```

Look at all of those models!

How about we now write a function to use the list of trained ensemble models to make predictions and then return a list of predictions (one set of predictions per model)?

Making predictions with an ensemble model

In [ ]:

```
# Create a function which uses a list of trained models to make and return a list of pred
ictions
def make_ensemble_preds(ensemble_models, data):
  ensemble_preds = []
  for model in ensemble_models:
    preds = model.predict(data) # make predictions with current ensemble model
    ensemble_preds.append(preds)
  return tf.constant(tf.squeeze(ensemble_preds))
```

In [ ]:

```
# Create a list of ensemble predictions
ensemble_preds = make_ensemble_preds(ensemble_models=ensemble_models,
                                     data=test_dataset)
ensemble_preds
```

```
WARNING:tensorflow:5 out of the last 22 calls to <function Model.make_predict_function.<l
ocals>.predict_function at 0x7fdcef255d40> triggered tf.function retracing. Tracing is ex
pensive and the excessive number of tracings could be due to (1) creating @tf.function re
peatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects
instead of tensors. For (1), please define your @tf.function outside of the loop. For (2)
, @tf.function has experimental_relax_shapes=True option that relaxes argument shapes tha
t can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/gu
ide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/func
tion for  more details.
```

```
WARNING:tensorflow:5 out of the last 22 calls to <function Model.make_predict_function.<l
ocals>.predict_function at 0x7fdcef255d40> triggered tf.function retracing. Tracing is ex
pensive and the excessive number of tracings could be due to (1) creating @tf.function re
peatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects
instead of tensors. For (1), please define your @tf.function outside of the loop. For (2)
, @tf.function has experimental_relax_shapes=True option that relaxes argument shapes tha
t can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/gu
ide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/func
tion for  more details.
```

```
WARNING:tensorflow:6 out of the last 23 calls to <function Model.make_predict_function.<l
ocals>.predict_function at 0x7fdc77fed9e0> triggered tf.function retracing. Tracing is ex
pensive and the excessive number of tracings could be due to (1) creating @tf.function re
peatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects
instead of tensors. For (1), please define your @tf.function outside of the loop. For (2)
, @tf.function has experimental_relax_shapes=True option that relaxes argument shapes tha
t can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/gu
ide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/func
tion for  more details.
```

```
WARNING:tensorflow:6 out of the last 23 calls to <function Model.make_predict_function.<l
ocals>.predict_function at 0x7fdc77fed9e0> triggered tf.function retracing. Tracing is ex
pensive and the excessive number of tracings could be due to (1) creating @tf.function re
peatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects
instead of tensors. For (1), please define your @tf.function outside of the loop. For (2)
, @tf.function has experimental_relax_shapes=True option that relaxes argument shapes tha
t can avoid unnecessary retracing. For (3), please refer to https://www.tensorflow.org/gu
ide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/func
tion for  more details.
```

Out[ ]:

```
<tf.Tensor: shape=(15, 556), dtype=float32, numpy=
array([[ 8805.756,  8773.019,  9028.609, ..., 50112.656, 49132.555,
         46455.695],
       [ 8764.092,  8740.744,  9051.838, ..., 49355.098, 48502.336,
         45333.934],
       [ 8732.57 ,  8719.407,  9093.386, ..., 49921.9  , 47992.15 ,
         45316.45 ],
       ...,
       [ 8938.421,  8773.84 ,  9045.577, ..., 49488.133, 49741.4  ,
         46536.25 ],
```

```
       [ 8724.761,  8805.311,  9094.972, ..., 49553.086, 48492.86 ,
        45084.266],
       [ 8823.311,  8768.297,  9047.492, ..., 49759.902, 48090.945,
        45874.336]], dtype=float32)>
```

Now we've got a set of ensemble predictions, we can evaluate them against the ground truth values.

However, since we've trained 15 models, there's going to be 15 sets of predictions. Rather than comparing every set of predictions to the ground truth, let's take the median (you could also take the mean too but [the median is usually more robust than the mean](#)).

In [ ]:

```
# Evaluate ensemble model(s) predictions
ensemble_results = evaluate_preds(y_true=y_test,
                                  y_pred=np.median(ensemble_preds, axis=0)) # take the m
edian across all ensemble predictions
ensemble_results
```

Out[ ]:

```
{'mae': 567.4423,
 'mape': 2.5843322,
 'mase': 0.996839,
 'mse': 1144512.9,
 'rmse': 1069.8191}
```

Nice! Looks like the ensemble model is the best performing model on the MAE metric so far.

Plotting the prediction intervals (uncertainty estimates) of our ensemble

Right now all of our model's (prior to the ensemble model) are predicting single points.

Meaning, given a set of `WINDOW_SIZE=7` values, the model will predict `HORIZION=1`.

But what might be more helpful than a single value?

Perhaps a range of values?

For example, if a model is predicting the price of Bitcoin to be 50,000USD tomorrow, would it be helpful to know it's predicting the 50,000USD because it's predicting the price to be between 48,000 and 52,000USD? (note: "$" has been omitted from the previous sentence due to formatting issues)

Knowing the range of values a model is predicting may help you make better decisions for your forecasts.

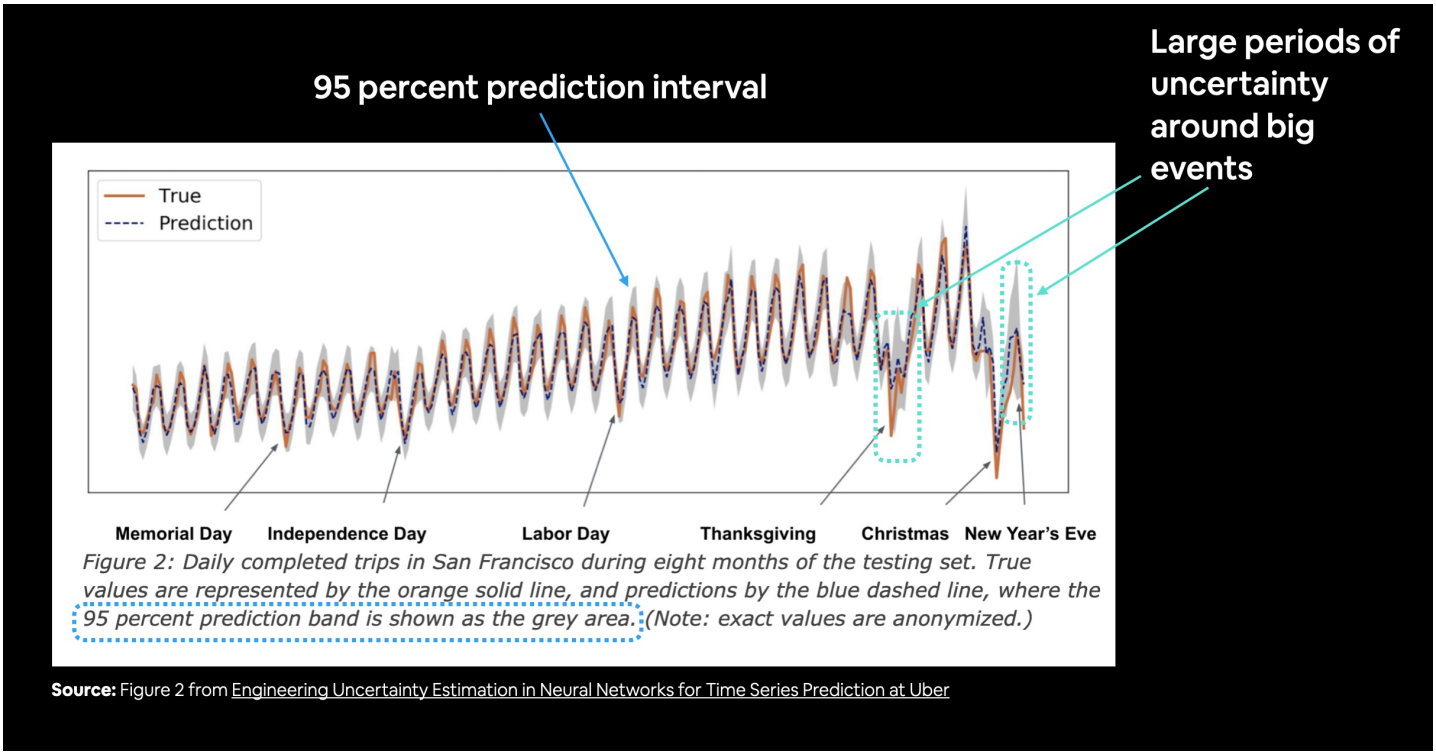
You'd know that although the model is predicting 50,000USD (a point prediction, or single value in time), the value could actually be within the range 48,000USD to 52,000USD (of course, the value could also be *outside* of this range as well, but we'll get to that later).

These kind of prediction ranges are called prediction intervals or uncertainty estimates. And they're often as important as the forecast itself.

Why?

Because point predictions are almost always going to be wrong. So having a range of values can help with decision making.

> **Resource(s):**
>
> - **The steps we're about to take have been inspired by the Machine Learning Mastery blog post *Prediction Intervals for Deep Learning Neural Networks*. Check out the post for more options to measure uncertainty with neural networks.**
> - **For an example of uncertainty estimates being used in the wild, I'd also refer to Uber's *Engineering Uncertainty Estimation in Neural Networks for Time Series Prediction at Uber* blog post.**



Example of how uncertainty estimates and predictions intervals can give an understanding of where point predictions (a single number) may not include all of useful information you'd like to know. For example, your model's point prediction for Uber trips on New Years Eve might be 100 (a made up number) but really, the prediction intervals are between 55 and 153 (both made up for the example). In this case, preparing 100 rides might end up being 53 short (it could even be more, like the point prediction, the prediction intervals are also estimates). The image comes from Uber's blog post on uncertainty estimation in neural networks .

One way of getting the 95% condfidnece prediction intervals for a deep learning model is the bootstrap method:

1. **Take the predictions from a number of randomly initialized models (we've got this thanks to our ensemble model)**
2. **Measure the standard deviation of the predictions**
3. **Multiply standard deviation by 1.96 (assuming the distribution is Gaussian, 95% of observations fall within 1.96 standard deviations of the mean, this is why we initialized our neural networks with a normal distribution)**
4. **To get the prediction interval upper and lower bounds, add and subtract the value obtained in (3) to the mean/median of the predictions made in (1)**

In [ ]:

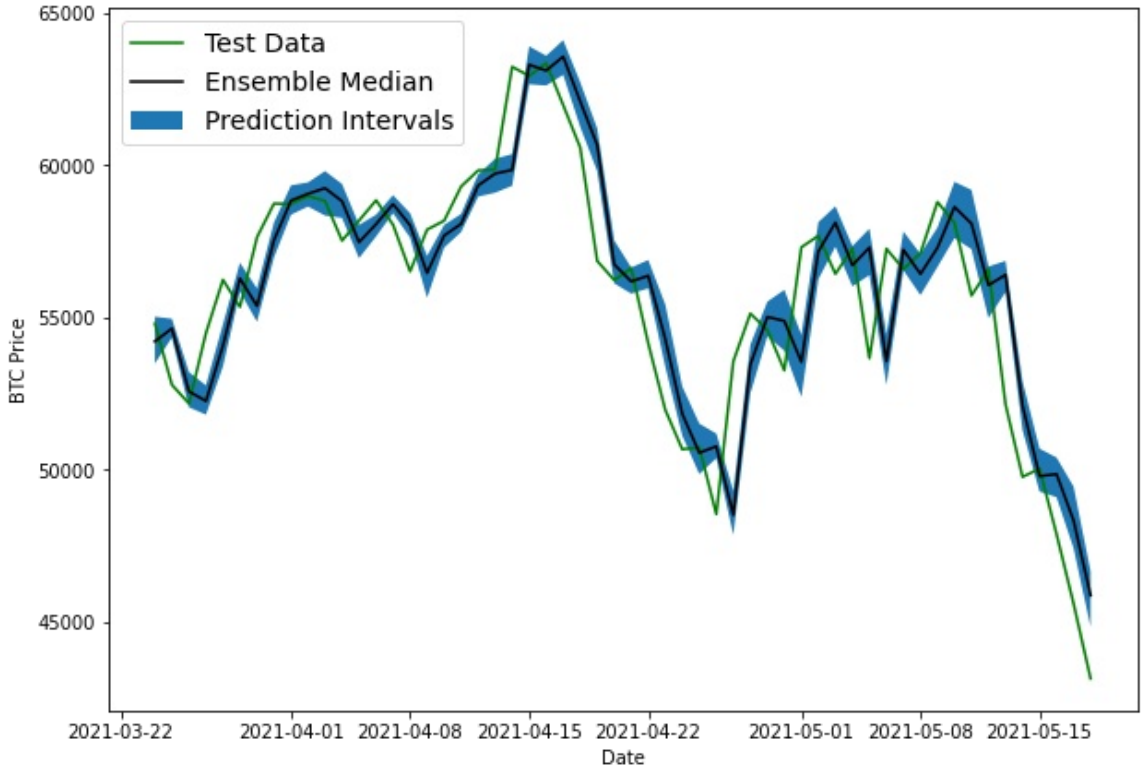
```
# Find upper and lower bounds of ensemble predictions
def get_upper_lower(preds): # 1. Take the predictions of multiple randomly initialized de
ep learning neural networks

  # 2. Measure the standard deviation of the predictions
  std = tf.math.reduce_std(preds, axis=0)

  # 3. Multiply the standard deviation by 1.96
  interval = 1.96 * std # https://en.wikipedia.org/wiki/1.96

  # 4. Get the prediction interval upper and lower bounds
  preds_mean = tf.reduce_mean(preds, axis=0)
  lower, upper = preds_mean - interval, preds_mean + interval
  return lower, upper

# Get the upper and lower bounds of the 95%
lower, upper = get_upper_lower(preds=ensemble_preds)
```

Wonderful, now we've got the upper and lower bounds for the the 95% prediction interval, let's plot them against our ensemble model's predictions.

To do so, we can use our plotting function as well as the `matplotlib.pyplot.fill_between()` method to shade in the space between the upper and lower bounds.

In [ ]:

```
# Get the median values of our ensemble preds
ensemble_median = np.median(ensemble_preds, axis=0)

# Plot the median of our ensemble preds along with the prediction intervals (where the pr
edictions fall between)
offset=500
plt.figure(figsize=(10, 7))
plt.plot(X_test.index[offset:], y_test[offset:], "g", label="Test Data")
plt.plot(X_test.index[offset:], ensemble_median[offset:], "k-", label="Ensemble Median")
plt.xlabel("Date")
plt.ylabel("BTC Price")
plt.fill_between(X_test.index[offset:],
                 (lower)[offset:],
                 (upper)[offset:], label="Prediction Intervals")
plt.legend(loc="upper left", fontsize=14);
```



We've just plotted:

- **The test data (the ground truth Bitcoin prices)**
- **The median of the ensemble predictions**
- **The 95% prediction intervals (assuming the data is Gaussian/normal, the model is saying that 95% of the time, predicted value should fall between this range)**

What can you tell about the ensemble model from the plot above?

It looks like the ensemble predictions are lagging slightly behind the actual data.

And the prediction intervals are fairly low throughout.

The combination of lagging predictions as well as low prediction intervals indicates that our ensemble model may be overfitting the data, meaning it's basically replicating what a naïve model would do and just predicting the previous timestep value for the next value.

This would explain why previous attempts to beat the naïve forecast have been futile.

We can test this hypothesis of overfitting by creating a model to make predictions into the future and seeing what they look like.

> **Note:** Our prediction intervals assume that the data we're using come from a Gaussian/normal distribution (also called a bell curve), however, open systems rarely follow the Gaussian. We'll see

> this later on with the turkey problem . For further reading on this topic, I'd recommend reading *The Black Swan* by Nassim Nicholas Taleb, especially Part 2 and Chapter 15.

Aside: two types of uncertainty (coconut and subway)

Inheritly, you know you cannot predict the future.

That doesn't mean trying to isn't valuable.

For many things, future predictions are helpful. Such as knowing the bus you're trying to catch to the library leaves at 10:08am. The time 10:08am is a **point prediction**, if the bus left at a random time every day, how helpful would it be?

Just like saying the price of Bitcoin tomorrow will be 50,000USD is a point prediction.

However, as we've discussed knowing a **prediction interval** or **uncertainty estimate** can be as helpful or even more helpful than a point prediction itself.

Uncertainty estimates seek out to qualitatively and quantitatively answer the questions:

- What can my model know? (with perfect data, what's possible to learn?)
- What doesn't my model know? (what can a model never predict?)

There are two types of uncertainty in machine learning you should be aware of:

- **Aleatoric uncertainty** - this type of uncertainty cannot be reduced, it is also referred to as "data" or "subway" uncertainty.
  - Let's say your train is scheduled to arrive at 10:08am but very rarely does it arrive at *exactly* 10:08am. You know it's usually a minute or two either side and perhaps up to 10-minutes late if traffic is bad. Even with all the data you could imagine, this level of uncertainty is still going to be present (much of it being noise).
  - When we measured prediction intervals, we were measuring a form of subway uncertainty for Bitcoin price predictions (a little either side of the point prediction).
- **Epistemic uncertainty** - this type of uncertainty can be reduced, it is also referred to as "model" or "coconut" uncertainty, it is very hard to calculate.
  - The analogy for coconut uncertainty involves whether or not you'd get hit on the head by a coconut when going to a beach.
    - If you were at a beach with coconuts trees, as you could imagine, this would be very hard to calculate. How often does a coconut fall of a tree? Where are you standing?
    - But you could reduce this uncertainty to zero by going to a beach without coconuts (collect more data about your situation).
  - Model uncertainty can be reduced by collecting more data samples/building a model to capture different parameters about the data you're modelling.

The lines between these are blurred (one type of uncertainty can change forms into the other) and they can be confusing at first but are important to keep in mind for any kind of time series prediction.

If you ignore the uncertanties, are you really going to get a reliable prediction?

Perhaps another example might help.

Uncertainty in dating

Let's say you're going on a First Date Feedback Radio Show to help improve your dating skills.

Where you go on a blind first date with a girl (feel free to replace girl with your own preference) and the radio hosts record the date and then playback snippets of where you could've improved.

And now let's add a twist.

Last week your friend went on the same show. They told you about the girl they met and how the conversation went.

Because you're now a machine learning engineer, you decide to build a machine learning model to help you with first date conversations.

What levels of uncertainty do we have here?

From an **aleatory uncertainty** (data) point of view, no matter how many conversations of first dates you collect, the conversation you end up having will likely be different to the rest (the best conversations have no subject and appear random).

From an **epistemic uncertainty** (model) point of view, if the date is truly blind and both parties don't know who they're seeing until they meet in person, the epistemic uncertainty would be high. Because now you have no idea who the person you're going to meet is nor what you might talk about.

However, the level of epistemic uncertainty would be reduced if your friend told about the girl they went on a date with last week on the show and it turns out you're going on a date with the same girl.

But even though you know a little bit about the girl, your **aleatory uncertainty** (or subway uncertainty) is still high because you're not sure where the conversation will go.

If you're wondering where above scenario came from, it happened to me this morning. Good timing right?

Learning more on uncertainty

The field of quantifying uncertainty estimation in machine learning is a growing area of research.

If you'd like to learn more I'd recommend the following.

> **Resources:** Places to learn more about uncertainty in machine learning/forecasting:
>
> - MIT 6.S191: Evidential Deep Learning and Uncertainty
> - Uncertainty quantification on Wikipedia
> - *Why you should care about the Nate Silver vs. Nassim Taleb Twitter war* by Isaac Faber - a great insight into the role of uncertainty in the example of election prediction.
> - *3 facts about time series forecasting that surprise experienced machine learning practitioners* by Skander Hannachi - fantastic outline of some of the main mistakes people make when building forecasting models, especially forgetting about uncertainty estimates.
> - *Engineering Uncertainty Estimation in Neural Networks for Time Series Prediction at Uber* - a discussion on techniques Uber used to engineer uncertainty estimates into their time sereis neural networks.

Model 9: Train a model on the full historical data to make predictions into future

What would a forecasting model be worth if we didn't use it to predict into the future?

It's time we created a model which is able to make future predictions on the price of Bitcoin.

To make predictions into the future, we'll train a model on the full dataset and then get to make predictions to some future horizon.

Why use the full dataset?

Previously, we split our data into training and test sets to evaluate how our model did on pseudo-future data (the test set).

But since the goal of a forecasting model is to predict values into the actual-future, we won't be using a test set.

> **Note:** Forecasting models need to be retrained every time a forecast is made. Why? Because if Bitcoin prices are updated daily and you predict the price for tomorrow. Your model is only really valid for one day. When a new price comes out (e.g. the next day), you'll have to retrain your model to incorporate that new price to predict the next forecast.

Let's get some data ready

Let's get some data ready:

In [ ]:

```
bitcoin_prices_windowed.head()
```

Out[ ]:

Date	Price	block_reward	Price+1	Price+2	Price+3	Price+4	Price+5	Price+6	Price+7
2013-10-01	123.65499	25	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2013-10-02	125.45500	25	123.65499	NaN	NaN	NaN	NaN	NaN	NaN
2013-10-03	108.58483	25	125.45500	123.65499	NaN	NaN	NaN	NaN	NaN
2013-10-04	118.67466	25	108.58483	125.45500	123.65499	NaN	NaN	NaN	NaN
2013-10-05	121.33866	25	118.67466	108.58483	125.45500	123.65499	NaN	NaN	NaN

In [ ]:

```
# Train model on entire data to make prediction for the next day
X_all = bitcoin_prices_windowed.drop(["Price", "block_reward"], axis=1).dropna().to_nump
y() # only want prices, our future model can be a univariate model
y_all = bitcoin_prices_windowed.dropna()["Price"].to_numpy()
```

Windows and labels ready! Let's turn them into performance optimized TensorFlow Datasets by:

1. **Turning `X_all` and `y_all` into tensor Datasets using `tf.data.Dataset.from_tensor_slices()`**
2. **Combining the features and labels into a Dataset tuple using `tf.data.Dataset.zip()`**
3. **Batch and prefetch the data using `tf.data.Dataset.batch()` and `tf.data.Dataset.prefetch()` respectively**

In [ ]:

```
# 1. Turn X and y into tensor Datasets
features_dataset_all = tf.data.Dataset.from_tensor_slices(X_all)
labels_dataset_all = tf.data.Dataset.from_tensor_slices(y_all)

# 2. Combine features & labels
dataset_all = tf.data.Dataset.zip((features_dataset_all, labels_dataset_all))

# 3. Batch and prefetch for optimal performance
BATCH_SIZE = 1024 # taken from Appendix D in N-BEATS paper
dataset_all = dataset_all.batch(BATCH_SIZE).prefetch(tf.data.AUTOTUNE)

dataset_all
```

Out[ ]:

```
<PrefetchDataset shapes: ((None, 7), (None,)), types: (tf.float64, tf.float64)>
```

And now let's create a model similar to `model_1` except with an extra layer, we'll also fit it to the entire dataset for 100 epochs (feel free to play around with the number of epochs or callbacks here, you've got the skills to now).

In [ ]:

```
tf.random.set_seed(42)

# Create model (nice and simple, just to test)
model_9 = tf.keras.Sequential([
  layers.Dense(128, activation="relu"),
  layers.Dense(128, activation="relu"),
  layers.Dense(HORIZON)
])
```

```
# Compile
model_9.compile(loss=tf.keras.losses.mae,
                optimizer=tf.keras.optimizers.Adam())

# Fit model on all of the data to make future forecasts
model_9.fit(dataset_all,
            epochs=100,
            verbose=0) # don't print out anything, we've seen this all before
```

Out[ ]:

```
<keras.callbacks.History at 0x7fdc77ae0dd0>
```

Make predictions on the future

Let's predict the future and get rich!

Well... maybe not.

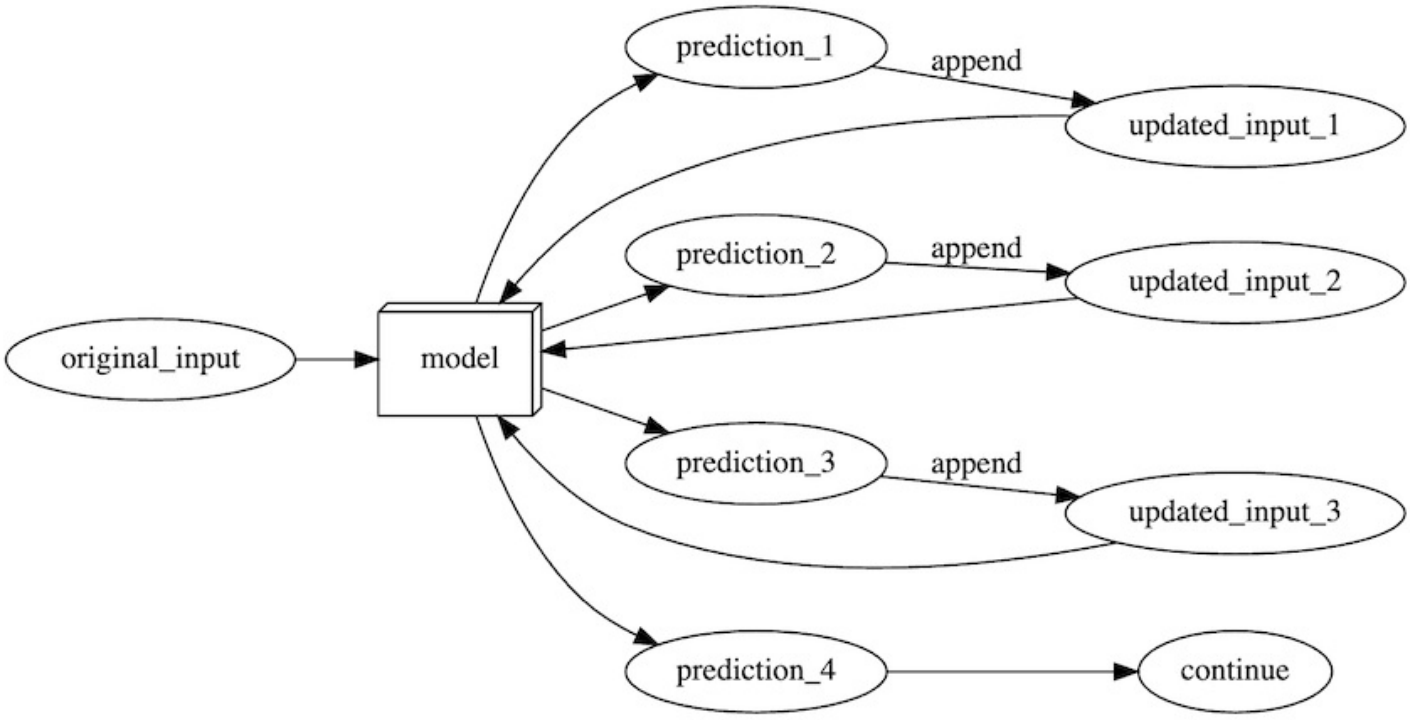
As you've seen so far, our machine learning models have performed quite poorly at predicting the price of Bitcoin (time series forecasting in open systems is typically a game of luck), often worse than the naive forecast.

That doesn't mean we can't use our models to *try* and predict into the future right?

To do so, let's start by defining a variable `INTO_FUTURE` which decides how many timesteps we'd like to predict into the future.

In [ ]:

```
# How many timesteps to predict into the future?
INTO_FUTURE = 14 # since our Bitcoin data is daily, this is for 14 days
```



Example flow chart representing the loop we're about to create for making forecasts. Not pictured: retraining a forecasting model every time a forecast is made & new data is acquired. For example, if you're predicting the price of Bitcoin daily, you'd want to retrain your model every day, since each day you're going to have a new data point to work with.

Alright, let's create a function which returns `INTO_FUTURE` forecasted values using a trained model.

To do so, we'll build the following steps:

1. **Function which takes as input:**
    - **a list of values (the Bitcoin historical data)**
    - **a trained model (such as `model_9`)**
    - **a window into the future to predict (our `INTO_FUTURE` variable)**

- **the window size a model was trained on (`WINDOW_SIZE`) - the model can only predict on the same kind of data it was trained on**

2. **Creates an empty list for future forecasts (this will be returned at the end of the function) and extracts the last `WINDOW_SIZE` values from the input values (predictions will start from the last `WINDOW_SIZE` values of the training data)**
3. **Loop `INTO_FUTURE` times making a prediction on `WINDOW_SIZE` datasets which update to remove the first the value and append the latest prediction**
    - **Eventually future predictions will be made using the model's own previous predictions as input**

In [ ]:

```
# 1. Create function to make predictions into the future
def make_future_forecast(values, model, into_future, window_size=WINDOW_SIZE) -> list:
  """
  Makes future forecasts into_future steps after values ends.

  Returns future forecasts as list of floats.
  """
  # 2. Make an empty list for future forecasts/prepare data to forecast on
  future_forecast = []
  last_window = values[-WINDOW_SIZE:] # only want preds from the last window (this will get updated)

  # 3. Make INTO_FUTURE number of predictions, altering the data which gets predicted on each time
  for _ in range(into_future):

    # Predict on last window then append it again, again, again (model starts to make forecasts on its own forecasts)
    future_pred = model.predict(tf.expand_dims(last_window, axis=0))
    print(f"Predicting on: \n {last_window} -> Prediction: {tf.squeeze(future_pred).numpy()}\n")

    # Append predictions to future_forecast
    future_forecast.append(tf.squeeze(future_pred).numpy())
    # print(future_forecast)

    # Update last window with new pred and get WINDOW_SIZE most recent preds (model was trained on WINDOW_SIZE windows)
    last_window = np.append(last_window, future_pred)[-WINDOW_SIZE:]

  return future_forecast
```

Nice! Time to bring BitPredict 🔮 to life and make future forecasts of the price of Bitcoin.

> **🛠 Exercise: In terms of a forecasting model, what might another approach to our `make_future_forecasts()` function? Recall, that for making forecasts, you need to retrain a model each time you want to generate a new prediction.**
>
> **So perhaps you could try to: make a prediction (one timestep into the future), retrain a model with this new prediction appended to the data, make a prediction, append the prediction, retrain a model... etc.**
>
> **As it is, the `make_future_forecasts()` function skips the retraining of a model part.**

In [ ]:

```
# Make forecasts into future of the price of Bitcoin
# Note: if you're reading this at a later date, you may already be in the future, so the forecasts
# we're making may not actually be forecasts, if that's the case, readjust the training data.
future_forecast = make_future_forecast(values=y_all,
                                       model=model_9,
```

```
                                        into_future=INTO_FUTURE,
                                        window_size=WINDOW_SIZE)
```

```
Predicting on:
 [56573.5554719  52147.82118698 49764.1320816  50032.69313676
 47885.62525472 45604.61575361 43144.47129086] -> Prediction: 55764.46484375

Predicting on:
 [52147.82118698 49764.1320816  50032.69313676 47885.62525472
 45604.61575361 43144.47129086 55764.46484375] -> Prediction: 50985.9453125

Predicting on:
 [49764.1320816  50032.69313676 47885.62525472 45604.61575361
 43144.47129086 55764.46484375 50985.9453125 ] -> Prediction: 48522.96484375

Predicting on:
 [50032.69313676 47885.62525472 45604.61575361 43144.47129086
 55764.46484375 50985.9453125  48522.96484375] -> Prediction: 48137.203125

Predicting on:
 [47885.62525472 45604.61575361 43144.47129086 55764.46484375
 50985.9453125  48522.96484375 48137.203125  ] -> Prediction: 47880.63671875

Predicting on:
 [45604.61575361 43144.47129086 55764.46484375 50985.9453125
 48522.96484375 48137.203125   47880.63671875] -> Prediction: 46879.71875

Predicting on:
 [43144.47129086 55764.46484375 50985.9453125  48522.96484375
 48137.203125   47880.63671875 46879.71875   ] -> Prediction: 48227.6015625

Predicting on:
 [55764.46484375 50985.9453125  48522.96484375 48137.203125
 47880.63671875 46879.71875    48227.6015625 ] -> Prediction: 53963.69140625

Predicting on:
 [50985.9453125  48522.96484375 48137.203125   47880.63671875
 46879.71875    48227.6015625  53963.69140625] -> Prediction: 49685.55859375

Predicting on:
 [48522.96484375 48137.203125   47880.63671875 46879.71875
 48227.6015625  53963.69140625 49685.55859375] -> Prediction: 47596.17578125

Predicting on:
 [48137.203125   47880.63671875 46879.71875    48227.6015625
 53963.69140625 49685.55859375 47596.17578125] -> Prediction: 48114.4296875

Predicting on:
 [47880.63671875 46879.71875    48227.6015625  53963.69140625
 49685.55859375 47596.17578125 48114.4296875 ] -> Prediction: 48808.0078125

Predicting on:
 [46879.71875    48227.6015625  53963.69140625 49685.55859375
 47596.17578125 48114.4296875  48808.0078125 ] -> Prediction: 48623.85546875

Predicting on:
 [48227.6015625  53963.69140625 49685.55859375 47596.17578125
 48114.4296875  48808.0078125  48623.85546875] -> Prediction: 50178.72265625
```

In [ ]:

```
future_forecast[:10]
```

Out[ ]:

```
[55764.465,
 50985.945,
 48522.965,
 48137.203,
 47880.637,
 46879.72,
 48227.6
```

```
 53963.69,
 49685.56,
 47596.176]
```

Plot future forecasts

This is so exciting! Forecasts made!

But right now, they're just numbers on a page.

Let's bring them to life by adhering to the data explorer's motto: visualize, visualize, visualize!

To plot our model's future forecasts against the historical data of Bitcoin, we're going to need a series of future dates (future dates from the final date of where our dataset ends).

How about we create a function to return a date range from some specified start date to a specified number of days into the future (`INTO_FUTURE`).

To do so, we'll use a combination of NumPy's `datetime64` datatype (our Bitcoin dates are already in this datatype) as well as NumPy's `timedelta64` method which helps to create date ranges.

In [ ]:

```
def get_future_dates(start_date, into_future, offset=1):
  """
  Returns array of datetime values from ranging from start_date to start_date+horizon.

  start_date: date to start range (np.datetime64)
  into_future: number of days to add onto start date for range (int)
  offset: number of days to offset start_date by (default 1)
  """
  start_date = start_date + np.timedelta64(offset, "D") # specify start date, "D" stands for day
  end_date = start_date + np.timedelta64(into_future, "D") # specify end date
  return np.arange(start_date, end_date, dtype="datetime64[D]") # return a date range between start date and end date
```

The start date of our forecasted dates will be the last date of our dataset.

In [ ]:

```
# Last timestep of timesteps (currently in np.datetime64 format)
last_timestep = bitcoin_prices.index[-1]
last_timestep
```

Out[ ]:

```
Timestamp('2021-05-18 00:00:00')
```

In [ ]:

```
# Get next two weeks of timesteps
next_time_steps = get_future_dates(start_date=last_timestep,
                                   into_future=INTO_FUTURE)
next_time_steps
```

Out[ ]:

```
array(['2021-05-19', '2021-05-20', '2021-05-21', '2021-05-22',
       '2021-05-23', '2021-05-24', '2021-05-25', '2021-05-26',
       '2021-05-27', '2021-05-28', '2021-05-29', '2021-05-30',
       '2021-05-31', '2021-06-01'], dtype='datetime64[D]')
```

Look at that! We've now got a list of dates we can use to visualize our future Bitcoin predictions.

But to make sure the lines of the plot connect (try not running the cell below and then plotting the data to see what I mean), let's insert the last timestep and Bitcoin price of our training data to the `next_time_steps` and `future_forecast` arrays.

In [ ]:

```
# Insert last timestep/final price so the graph doesn't look messed
next_time_steps = np.insert(next_time_steps, 0, last_timestep)
future_forecast = np.insert(future_forecast, 0, btc_price[-1])
next_time_steps, future_forecast
```

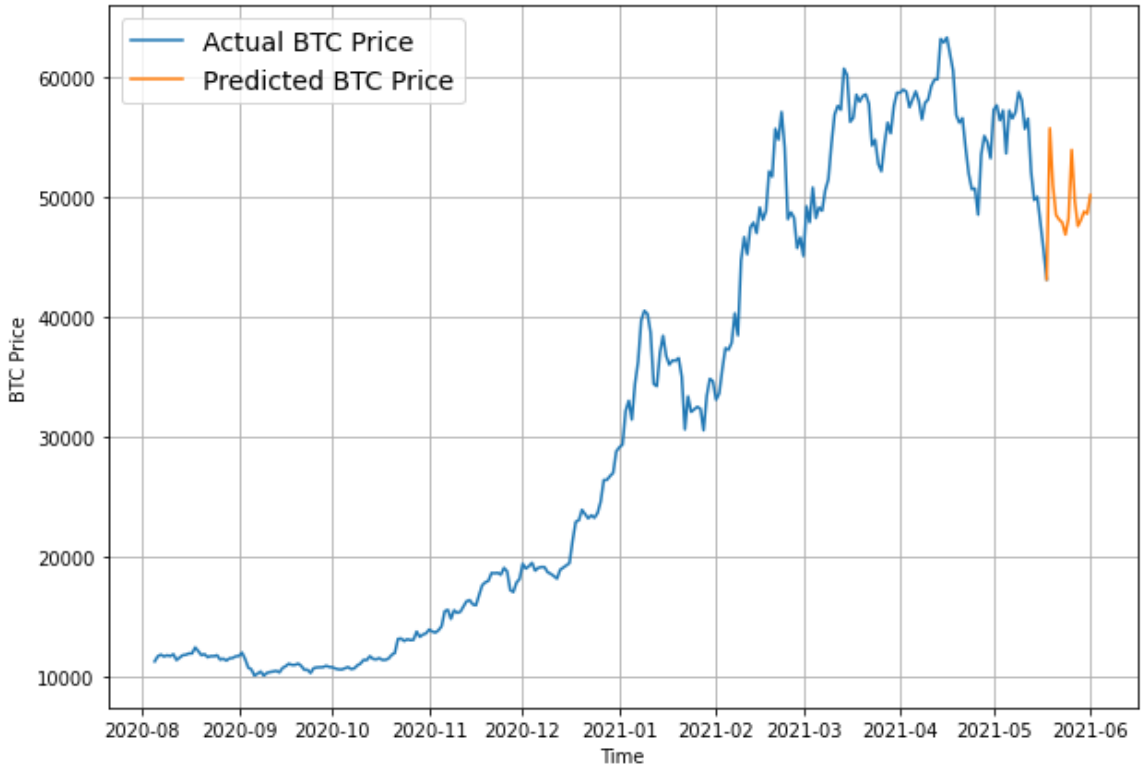
Out[ ]:

```
(array(['2021-05-18', '2021-05-19', '2021-05-20', '2021-05-21',
        '2021-05-22', '2021-05-23', '2021-05-24', '2021-05-25',
        '2021-05-26', '2021-05-27', '2021-05-28', '2021-05-29',
        '2021-05-30', '2021-05-31', '2021-06-01'], dtype='datetime64[D]'),
 array([43144.473, 55764.465, 50985.945, 48522.965, 48137.203, 47880.637,
        46879.72 , 48227.6  , 53963.69 , 49685.56 , 47596.176, 48114.43 ,
        48808.008, 48623.855, 50178.723], dtype=float32))
```

Time to plot!

In [ ]:

```
# Plot future price predictions of Bitcoin
plt.figure(figsize=(10, 7))
plot_time_series(bitcoin_prices.index, btc_price, start=2500, format="-", label="Actual BTC Price")
plot_time_series(next_time_steps, future_forecast, format="-", label="Predicted BTC Price")
```



Hmmm... how did our model go?

It looks like our predictions are starting to form a bit of a cyclic pattern (up and down in the same way).

Perhaps that's due to our model overfitting the training data and not generalizing well for future data. Also, as you could imagine, the further you predict into the future, the higher your chance for error (try seeing what happens when you predict 100 days into the future).

But of course, we can't measure these predictions as they are because after all, they're predictions into the actual-future (by the time you read this, the future might have already happened, if so, how did the model go?).

> **Note:** A reminder, the predictions we've made here are not financial advice. And by now, you should be well aware of just how poor machine learning models can be at forecasting values in an open system - anyone promising you a model which can "beat the market" is likely trying to scam

> you, oblivious to their errors or very lucky.

Model 10: Why forecasting is BS (the turkey problem)

When creating any kind of forecast, you must keep the turkey problem in mind.

The turkey problem is an analogy for when your observational data (your historical data) fails to capture a future event which is catostrophic and could lead you to ruin.

The story goes, a turkey lives a good life for 1000 days, being fed every day and taken care of by its owners until the evening before Thanksgiving.

Based on the turkey's observational data, it has no reason to believe things shouldn't keep going the way they are.

In other words, how could a turkey possibly predict that on day 1001, after 1000 consectutive good days, it was about to have a far from ideal day.

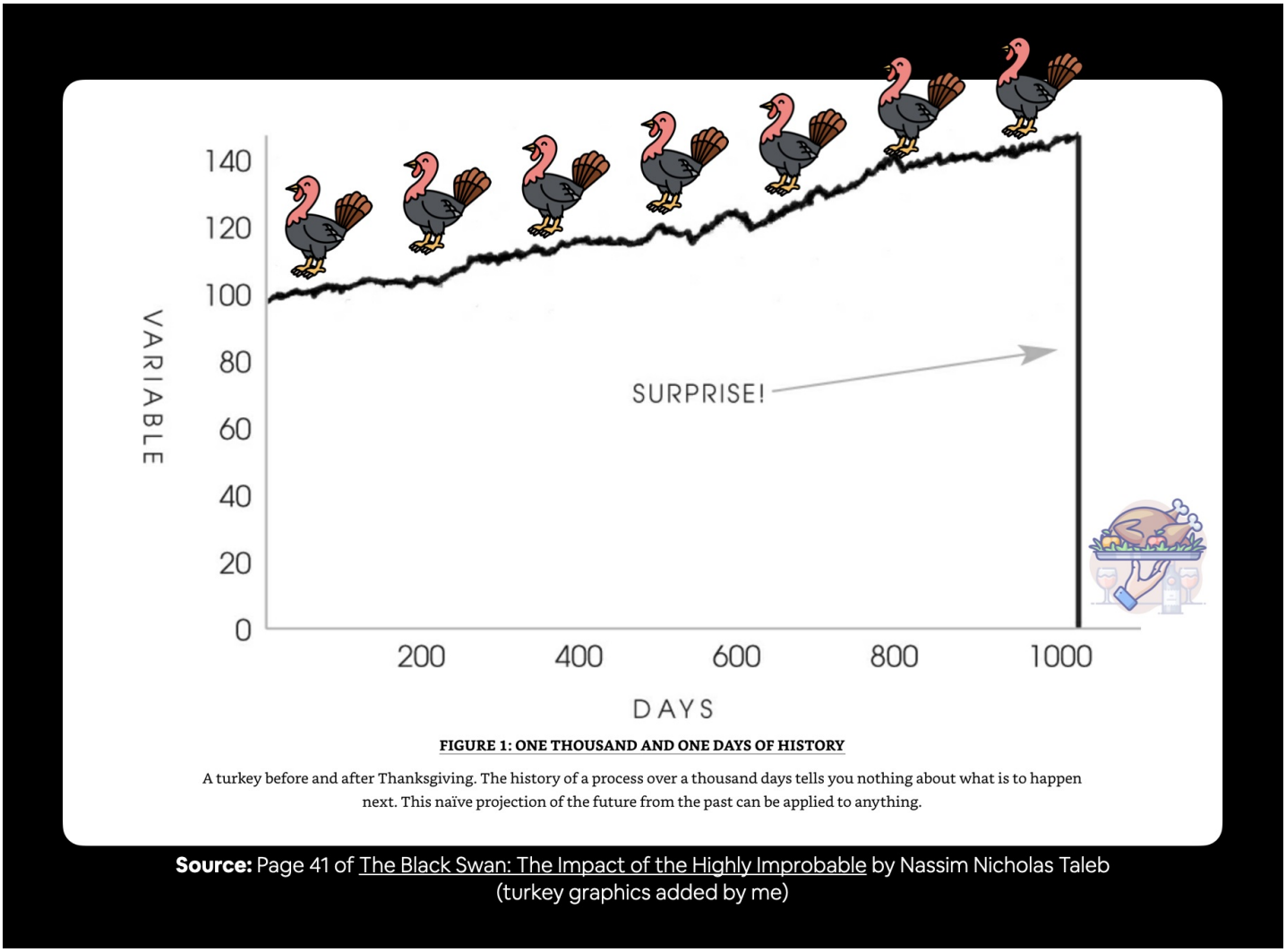


FIGURE 1: ONE THOUSAND AND ONE DAYS OF HISTORY

A turkey before and after Thanksgiving. The history of a process over a thousand days tells you nothing about what is to happen next. This naïve projection of the future from the past can be applied to anything.

Source: Page 41 of The Black Swan: The Impact of the Highly Improbable by Nassim Nicholas Taleb
(turkey graphics added by me)

Example of the turkey problem. A turkey might live 1000 good days and none of them would be a sign of what's to happen on day 1001. Similar with forecasting, your historical data may not have any indication of a change which is about to come. The graph image is from page 41 of The Black Swan by Nassim Taleb (I added in the turkey graphics).

How does this relate to predicting the price of Bitcoin (or the price of any stock or figure in an open market)?

You could have the historical data of Bitcoin for its entire existence and build a model which predicts it perfectly.

But then one day for some unknown and unpredictable reason, the price of Bitcoin plummets 100x in a single day.

Of course, this kind of scenario is unlikely.

But that doesn't take away from its significance.

Think about it in your own life, how many times have the most significant events happened seemingly out of the blue?

As in, you could go to a cafe and run into the love of your life, despite visiting the same cafe for 10-years straight and never running into this person before.

The same thing goes for predicting the price of Bitcoin, you could make money for 10-years straight and then lose it all in a single day.

It doesn't matter how many times you get paid, it matters the amount you get paid.

> 🔑 **Resource:** If you'd like to learn more about the turkey problem, I'd recommend the following:
>
> - Explaining both the XIV trade and why forecasting is BS by Nassim Taleb
> - *The Black Swan* by Nassim Taleb (epsecially Chapter 4 which outlines and discusses the turkey problem)

Let's get specific and see how the turkey problem effects us modelling the historical and future price of Bitcoin.

To do so, we're going to manufacture a highly unlikely data point into the historical price of Bitcoin, the price falling 100x in one day.

> 🔑 **Note:** A very unlikely and unpredictable event such as the price of Bitcoin falling 100x in a single day (note: the adjective "unlikely" is based on the historical price changes of Bitcoin) is also referred to a Black Swan event. A Black Swan event is an unknown unknown, you have no way of predicting whether or not it will happen but these kind of events often have a large impact.

In [ ]:

```
# Let's introduce a Turkey problem to our BTC data (price BTC falls 100x in one day)
btc_price_turkey = btc_price.copy()
btc_price_turkey[-1] = btc_price_turkey[-1] / 100
```

In [ ]:

```
# Manufacture an extra price on the end (to showcase the Turkey problem)
btc_price_turkey[-10:]
```

Out[ ]:

```
[58788.2096789273,
 58102.1914262342,
 55715.5466512869,
 56573.5554719043,
 52147.8211869823,
 49764.1320815975,
 50032.6931367648,
 47885.6252547166,
 45604.6157536131,
 431.44471290860304]
```

Notice the last value is 100x lower than what it actually was (remember, this is not a real data point, its only to illustrate the effects of the turkey problem).

Now we've got Bitcoin prices including a turkey problem data point, let's get the timesteps.

In [ ]:

```
# Get the timesteps for the turkey problem
btc_timesteps_turkey = np.array(bitcoin_prices.index)
btc_timesteps_turkey[-10:]
```

Out[ ]:

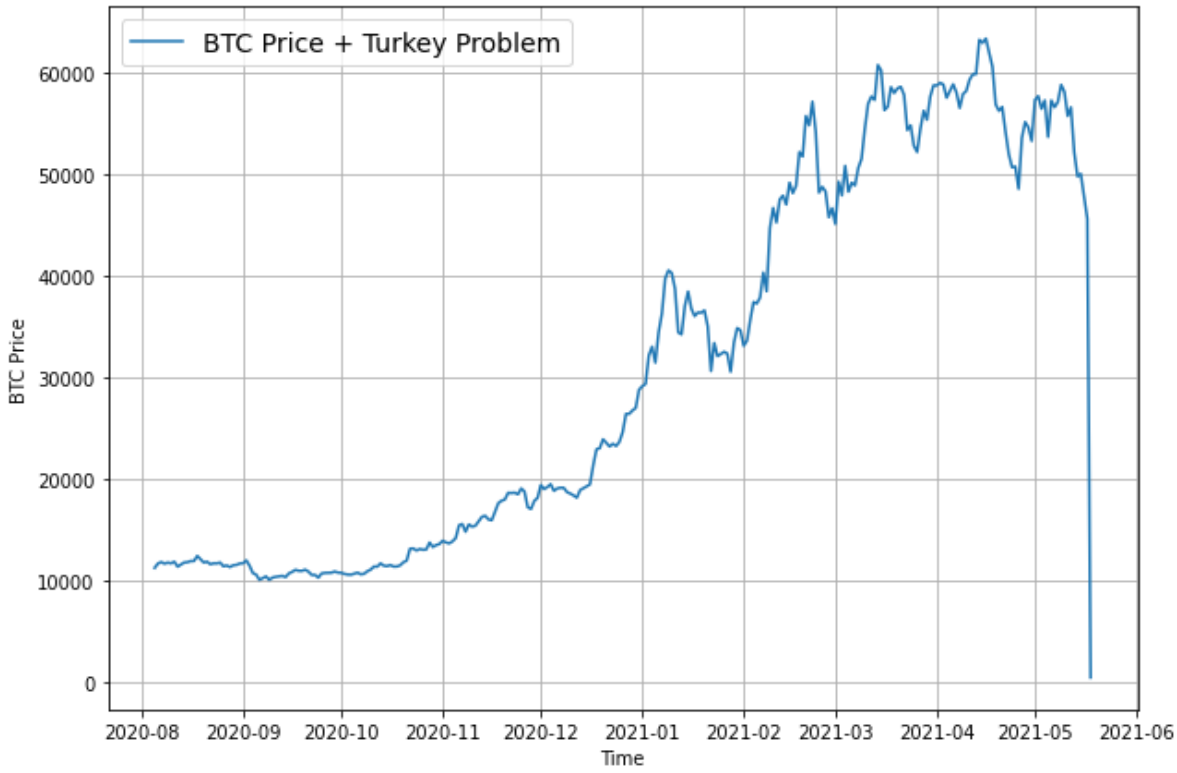
```
array(['2021-05-09T00:00:00.000000000', '2021-05-10T00:00:00.000000000',
       '2021-05-11T00:00:00.000000000', '2021-05-12T00:00:00.000000000',
```

```
       '2021-05-13T00:00:00.000000000', '2021-05-14T00:00:00.000000000',
       '2021-05-15T00:00:00.000000000', '2021-05-16T00:00:00.000000000',
       '2021-05-17T00:00:00.000000000', '2021-05-18T00:00:00.000000000'],
      dtype='datetime64[ns]')
```

Beautiful! Let's see our artificially created turkey problem Bitcoin data.

In [ ]:

```
plt.figure(figsize=(10, 7))
plot_time_series(timesteps=btc_timesteps_turkey,
                 values=btc_price_turkey,
                 format="-",
                 label="BTC Price + Turkey Problem",
                 start=2500)
```



How do you think building a model on this data will go?

Remember, all we've changed is a single data point out of our entire dataset.

Before we build a model, let's create some windowed datasets with our turkey data.

In [ ]:

```
# Create train and test sets for turkey problem data
full_windows, full_labels = make_windows(np.array(btc_price_turkey), window_size=WINDOW_
SIZE, horizon=HORIZON)
len(full_windows), len(full_labels)

X_train, X_test, y_train, y_test = make_train_test_splits(full_windows, full_labels)
len(X_train), len(X_test), len(y_train), len(y_test)
```

Out[ ]:

```
(2224, 556, 2224, 556)
```

Building a turkey model (model to predict on turkey data)

With our updated data, we only changed 1 value.

Let's see how it effects a model.

To keep things comparable to previous models, we'll create a `turkey_model` which is a clone of `model_1`

(same architecture, but different data).

That way, when we evaluate the `turkey_model` we can compare its results to `model_1_results` and see how much a single data point can influence a model's performance.

In [ ]:

```
# Clone model 1 architecture for turkey model and fit the turkey model on the turkey data
turkey_model = tf.keras.models.clone_model(model_1)
turkey_model._name = "Turkey_Model"
turkey_model.compile(loss="mae",
                     optimizer=tf.keras.optimizers.Adam())
turkey_model.fit(X_train, y_train,
                 epochs=100,
                 verbose=0,
                 validation_data=(X_test, y_test),
                 callbacks=[create_model_checkpoint(turkey_model.name)])
```

```
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
```

```
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
```

```
INFO:tensorflow:Assets written to: model_experiments/Turkey_Model/assets
```

Out[ ]:

```
<keras.callbacks.History at 0x7fdc7a4dd550>
```

In [ ]:

```
# Evaluate turkey model on test data
turkey_model.evaluate(X_test, y_test)
```

```
18/18 [==============================] - 0s 2ms/step - loss: 696.1285
```

Out[ ]:

```
696.1284790039062
```

In [ ]:

```
# Load best model and evaluate on test data
turkey_model = tf.keras.models.load_model("model_experiments/Turkey_Model/")
turkey_model.evaluate(X_test, y_test)
```

```
18/18 [==============================] - 0s 2ms/step - loss: 638.3047
```

Out[ ]:

```
638.3046875
```

Alright, now let's make some predictions with our model and evaluate them on the test data.

In [ ]:

```
# Make predictions with Turkey model
turkey_preds = make_preds(turkey_model, X_test)
turkey_preds[:10]
```

Out[ ]:

```
<tf.Tensor: shape=(10,), dtype=float32, numpy=
array([8858.391, 8803.98 , 9039.575, 8785.937, 8778.044, 8735.638,
       8684.118, 8558.659, 8461.373, 8542.206], dtype=float32)>
```

In [ ]:

```
# Evaluate turkey preds
turkey_results = evaluate_preds(y_true=y_test,
                                y_pred=turkey_preds)
turkey_results
```

Out[ ]:

```
{'mae': 17144.766,
 'mape': 121.58286,
 'mase': 26.53158,
 'mse': 615487800.0,
 'rmse': 23743.305}
```

And with just one value change, our error metrics go through the roof.

To make sure, let's remind ourselves of how `model_1` went on unmodified Bitcoin data (no turkey problem).

In [ ]:

```
model_1_results
```

Out[ ]:

```
{'mae': 568.95123,
 'mape': 2.5448983,
 'mase': 0.9994897,
```

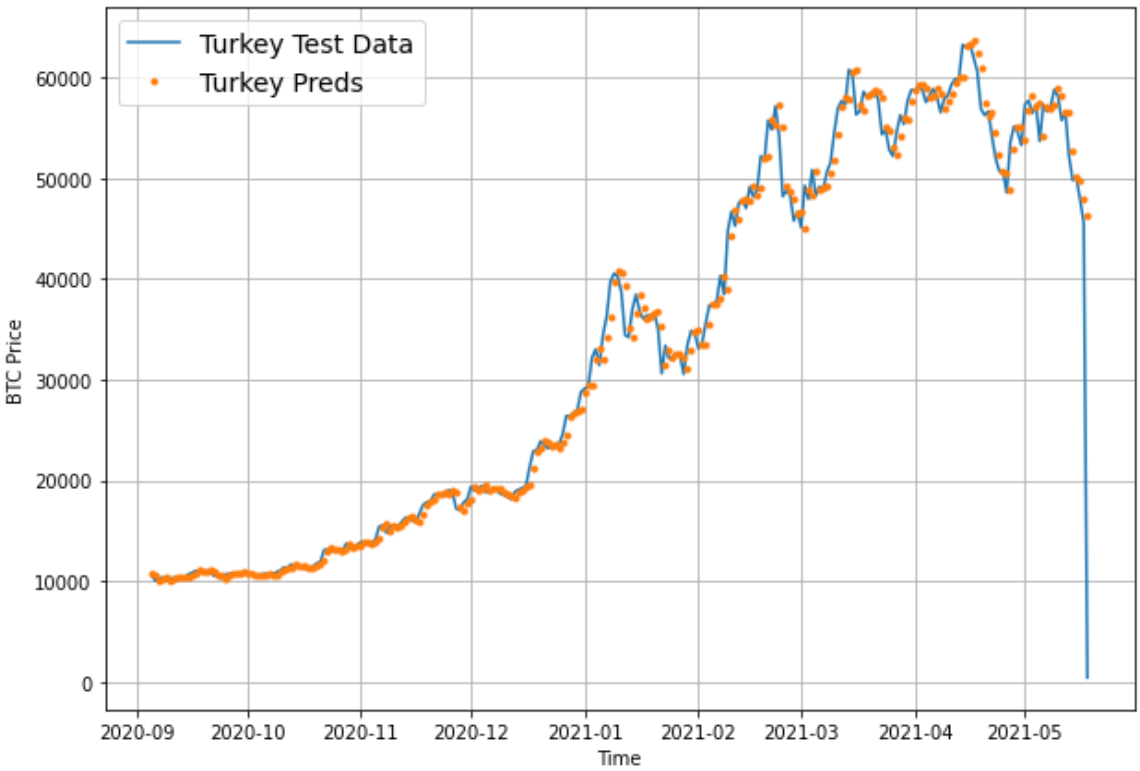
```
 'mse': 1171744.0,
 'rmse': 1082.4713}
```

By changing just one value, the `turkey_model` MAE increases almost 30x over `model_1`.

Finally, we'll visualize the turkey predictions over the test turkey data.

In [ ]:

```
plt.figure(figsize=(10, 7))
# plot_time_series(timesteps=btc_timesteps_turkey[:split_size], values=btc_price_turkey[:split_size], label="Train Data")
offset=300
plot_time_series(timesteps=btc_timesteps_turkey[-len(X_test):],
                 values=btc_price_turkey[-len(y_test):],
                 format="-",
                 label="Turkey Test Data", start=offset)
plot_time_series(timesteps=btc_timesteps_turkey[-len(X_test):],
                 values=turkey_preds,
                 label="Turkey Preds",
                 start=offset);
```



Why does this happen?

Why does our model fail to capture the turkey problem data point?

Think about it like this, just like a turkey who lives 1000 joyful days, based on observation alone has no reason to believe day 1001 won't be as joyful as the last, a model which has been trained on historical data of Bitcoin which has no single event where the price decreased by 100x in a day, has no reason to predict it will in the future.

A model cannot predict anything in the future outside of the distribution it was trained on.

In turn, highly unlikely price movements (based on historical movements), upward or downward will likely never be part of a forecast.

However, as we've seen, despite their unlikeliness, these events can have huuuuuuuuge impacts to the performance of our models.

> **Resource:** For a great article which discusses Black Swan events and how they often get ignored due to the assumption that historical events come from a certain distribution and that future events will come from the same distribution see *Black Swans, Normal Distributions and Supply*

> *Chain Risk* by Spend Matters.

Compare Models

We've trained a bunch of models.

And if anything, we've seen just how poorly machine learning and deep learning models are at forecasting the price of Bitcoin (or any kind of open market value).

To highlight this, let's compare the results of all of the modelling experiments we've performed so far.

In [ ]:

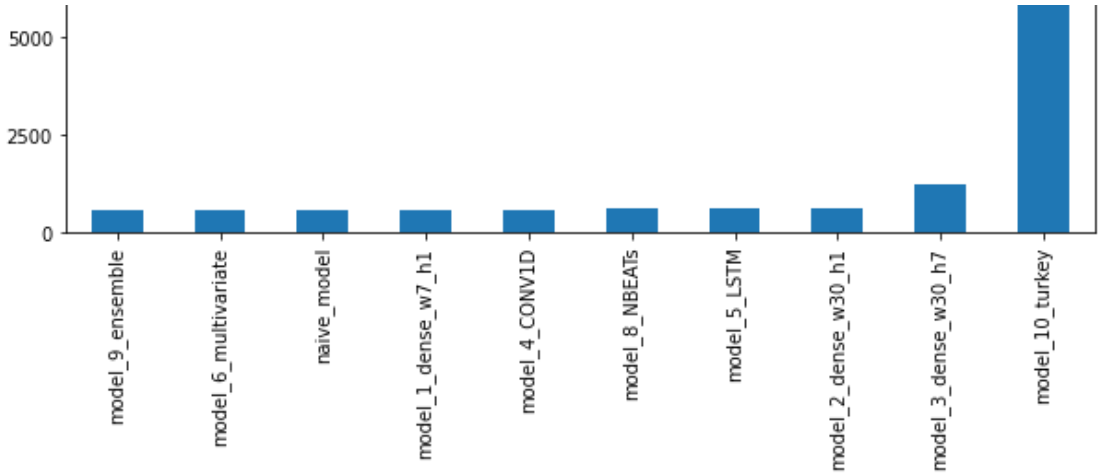
```
# Compare different model results (w = window, h = horizon, e.g. w=7 means a window size of 7)
model_results = pd.DataFrame({"naive_model": naive_results,
                              "model_1_dense_w7_h1": model_1_results,
                              "model_2_dense_w30_h1": model_2_results,
                              "model_3_dense_w30_h7": model_3_results,
                              "model_4_CONV1D": model_4_results,
                              "model_5_LSTM": model_5_results,
                              "model_6_multivariate": model_6_results,
                              "model_8_NBEATs": model_7_results,
                              "model_9_ensemble": ensemble_results,
                              "model_10_turkey": turkey_results}).T
model_results.head(10)
```

Out[ ]:

	mae	mse	rmse	mape	mase
naive_model	567.980225	1.147547e+06	1071.236206	2.516525	0.999570
model_1_dense_w7_h1	568.951233	1.171744e+06	1082.471313	2.544898	0.999490
model_2_dense_w30_h1	608.961487	1.281439e+06	1132.006470	2.769339	1.064471
model_3_dense_w30_h7	1237.506348	5.405198e+06	1425.747681	5.558878	2.202074
model_4_CONV1D	570.828308	1.176671e+06	1084.744751	2.559336	1.002787
model_5_LSTM	596.644653	1.273487e+06	1128.488770	2.683845	1.048139
model_6_multivariate	567.587402	1.161688e+06	1077.816528	2.541387	0.997094
model_8_NBEATs	585.499817	1.179492e+06	1086.043945	2.744519	1.028561
model_9_ensemble	567.442322	1.144513e+06	1069.819092	2.584332	0.996839
model_10_turkey	17144.765625	6.154878e+08	23743.304688	121.582863	26.531580

In [ ]:

```
# Sort model results by MAE and plot them
model_results[["mae"]].sort_values(by="mae").plot(figsize=(10, 7), kind="bar");
```



The majority of our deep learning models perform on par or only slightly better than the naive model. And for the turkey model, changing a single data point destroys its performance.

> **Note:** **Just because one type of model performs better here doesn't mean it'll perform the best elsewhere (and vice versa, just because one model performs poorly here, doesn't mean it'll perform poorly elsewhere).**

As I said at the start, this is not financial advice.

After what we've gone through, you'll now have some of the skills required to callout BS for any future tutorial or blog post or investment sales guide claiming to have model which is able to predict the futrue.

[Mark Saroufim's Tweet] sums this up nicely (stock market forecasting with a machine learning model is just as reliable as palm reading).

Beware the tutorials or trading courses which claim to use some kind of algorithm to beat the market (an open system), they're likely a scam or the creator is very lucky and hasn't yet come across a turkey problem.

Don't let these results get you down though, forecasting in a closed system (such as predicting the demand of electricity) often yields quite usable results.

If anything, this module teaches anti-knowledge. Knowing that forecasting methods usually *don't* perform well in open systems.

Plus, sometimes not knowing the future is a benefit. A known future is already the past.

🛠 Exercises

1. Does scaling the data help for univariate/multivariate data? (e.g. getting all of the values between 0 & 1)
    - Try doing this for a univariate model (e.g. `model_1`) and a multivariate model (e.g. `model_6`) and see if it effects model training or evaluation results.
2. Get the most up to date data on Bitcoin, train a model & see how it goes (our data goes up to May 18 2021).
    - You can download the Bitcoin historical data for free from coindesk.com/price/bitcoin and clicking "Export Data" -> "CSV".
3. For most of our models we used `WINDOW_SIZE=7`, but is there a better window size?
    - Setup a series of experiments to find whether or not there's a better window size.
    - For example, you might train 10 different models with `HORIZON=1` but with window sizes ranging from 2-12.
4. Create a windowed dataset just like the ones we used for `model_1` using `tf.keras.preprocessing.timeseries_dataset_from_array()` and retrain `model_1` using the recreated dataset.
5. For our multivariate modelling experiment, we added the Bitcoin block reward size as an extra feature to make our time series multivariate.
    - Are there any other features you think you could add?
    - If so, try it out, how do these affect the model?
6. Make prediction intervals for future forecasts. To do so, one way would be to train an ensemble model on all of the data, make future forecasts with it and calculate the prediction intervals of the ensemble just like we did for `model_8`.
7. For future predictions, try to make a prediction, retrain a model on the predictions, make a prediction, retrain a model, make a prediction, retrain a model, make a prediction (retrain a model each time a new prediction is made). Plot the results, how do they look compared to the future predictions where a model wasn't retrained for every forecast (`model_9`)?
8. Throughout this notebook, we've only tried algorithms we've handcrafted ourselves. But it's worth seeing how a purpose built forecasting algorithm goes.
    - Try out one of the extra algorithms listed in the modelling experiments part such as:
        - Facebook's Kats library - there are many models in here, remember the machine learning practioner's motto: experiment, experiment, experiment.
        - LinkedIn's Greykite library

📖 Extra-curriculum

We've only really scratched the surface with time series forecasting and time series modelling in general. But the good news is, you've got plenty of hands-on coding experience with it already.

If you'd like to dig deeper in to the world of time series, I'd recommend the following:

- Forecasting: Principles and Practice is an outstanding online textbook which discusses at length many of the most important concepts in time series forecasting. I'd especially recommend reading at least Chapter 1 in full.
    - I'd definitely recommend at least checking out chapter 1 as well as the chapter on forecasting accuracy measures.
- 📖 Introduction to machine learning and time series by Markus Loning goes through different time series problems and how to approach them. It focuses on using the `sktime` library (Scikit-Learn for time series), though the principles are applicable elsewhere.
- *Why you should care about the Nate Silver vs. Nassim Taleb Twitter war* by Isaac Faber is an outstanding discussion insight into the role of uncertainty in the example of election prediction.
- TensorFlow time series tutorial - A tutorial on using TensorFlow to forecast weather time series data with TensorFlow.
- 📖 *The Black Swan* by Nassim Nicholas Taleb - Nassim Taleb was a pit trader (a trader who trades on their own behalf) for 25 years, this book compiles many of the lessons he learned from first-hand experience. It changed my whole perspective on our ability to predict.
- *3 facts about time series forecasting that surprise experienced machine learning practitioners* by Skander

- 5 facts about time series forecasting that surprise experienced machine learning practitioners by Skanda Hannachi, Ph.D - time series data is different to other kinds of data, if you've worked on other kinds of machine learning problems before, getting into time series might require some adjustments, Hannachi outlines 3 of the most common.
- 🎓 World-class lectures by Jordan Kern, watching these will take you from 0 to 1 with time series problems:
  - Time Series Analysis - how to analyse time series data.
  - Time Series Modelling - different techniques for modelling time series data (many of which aren't deep learning).