RNN

Recurrent Neural Network

Artificial neural network that are able to recognize and predict **sequences of data** such as text, genomes, handwriting, spoken word, or numerical time series data.

They have **loops** that allow a consistent flow of information and can work on sequences of arbitrary lengths.

Make use of internal state (**memory**) to process a sequence of inputs.

https://heartbeat.fritz.ai/a-beginners-guide-to-implementing-long-short-term-memory-networks-lstm-eb7a2ff09a27

RNN utilization

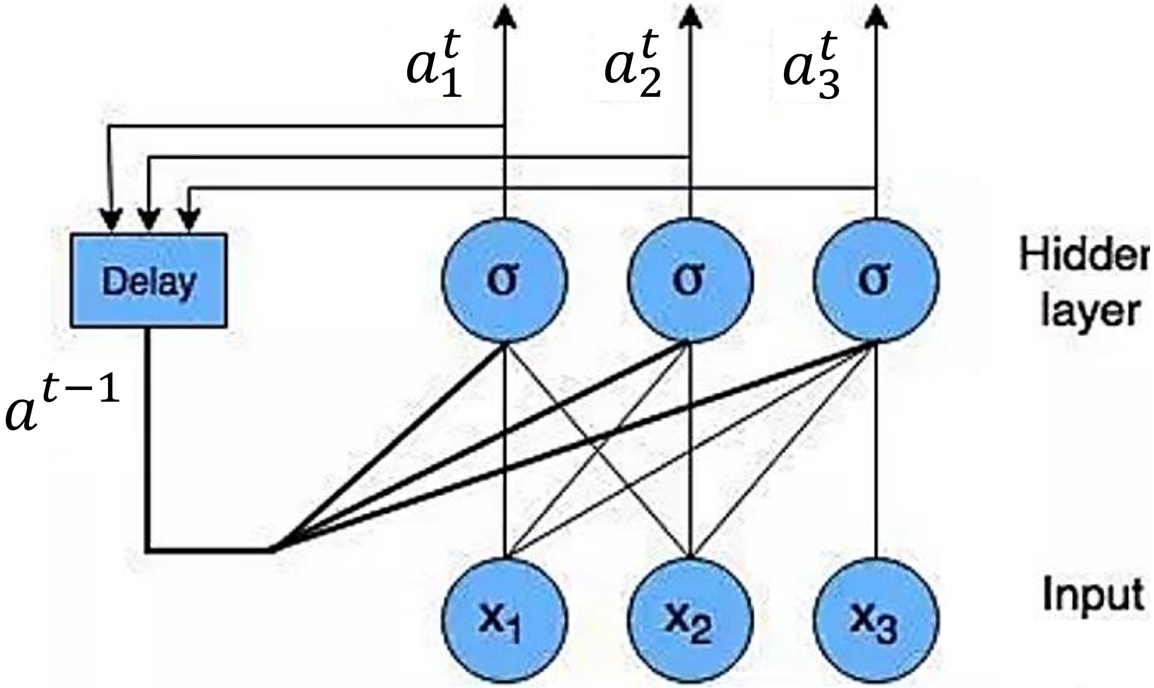
Processing sequential data, where the order of elements matters (e.g., time series, text, language).

RNNs are used to solve several problems:

- Language translation and modeling
- Speech recognition
- Image captioning
- **Time series data such as stock prices** (tell when to buy or sell)
- Automatic (autonomous?) driving systems to anticipate car trajectories; help avoid accidents.

https://heartbeat.fritz.ai/a-beginners-guide-to-implementing-long-short-term-memory-networks-lstm-eb7a2ff09a27

RNN structure



The output of the hidden layer is ***fed back*** into the same hidden layer

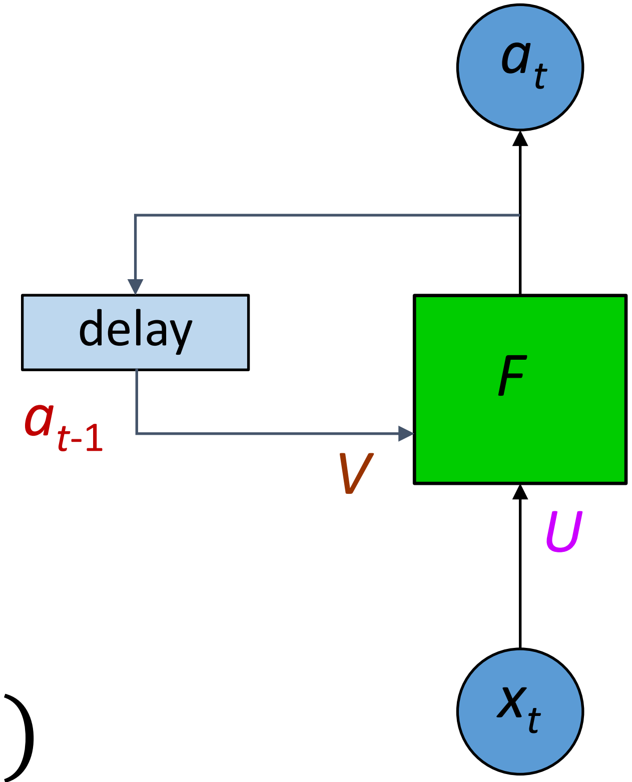
We can model *time* or sequence-dependent data (time series)

t – time moment

The weights of the connections between time steps are *shared* i.e. there isn't a different set of weights for each time step.

https://adventuresinmachinelearning.com/recurrent-neural-networks-lstm-tutorial-tensorflow/

RNN structure



$$a_t = F\,(\,U\,x_t + V\,a_{t-1})$$

current output (activation)

activation function

weight matrix for input

current input

weight matrix for recurrent output

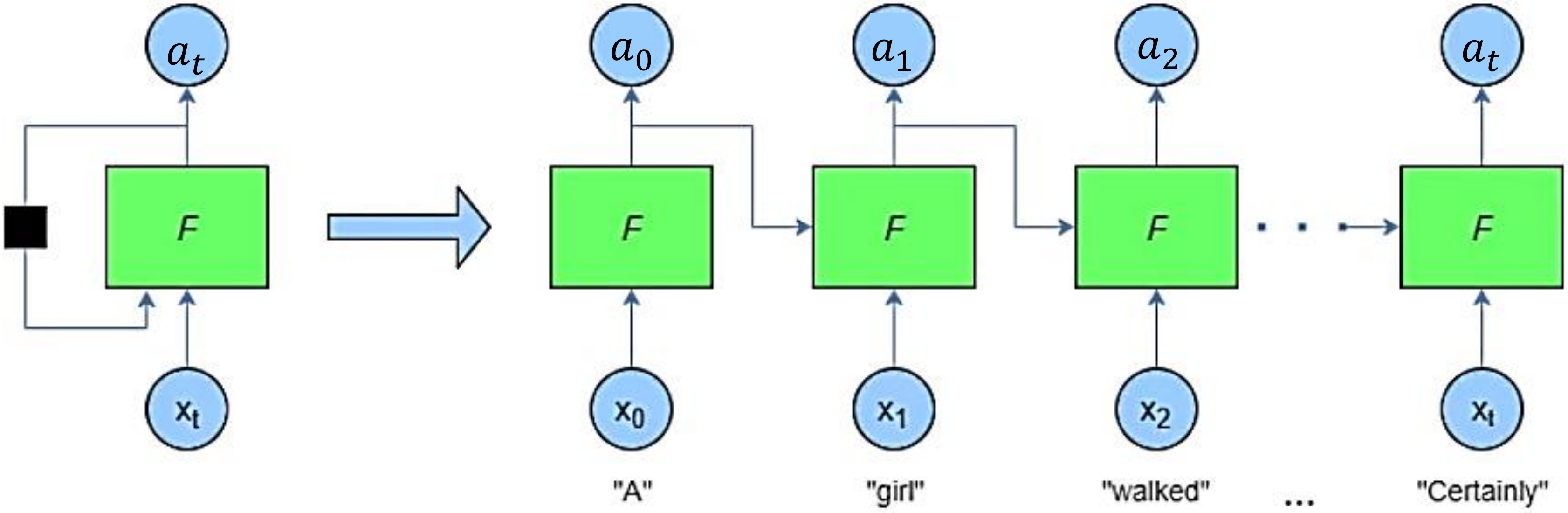
recurrent output

Example

“A **girl** walked into a bar, and **she** said: ‘Can I have a drink please?’. The bartender said ‘Certainly {?}”

{?} can be “miss”, “ma’am”, …
“sir”, “Mister”, … also could fit

To get the correct gender of the noun, the neural network needs to **recall** that two previous words designating the likely gender (i.e., “**girl**” and “**she**”) were used.



RNN

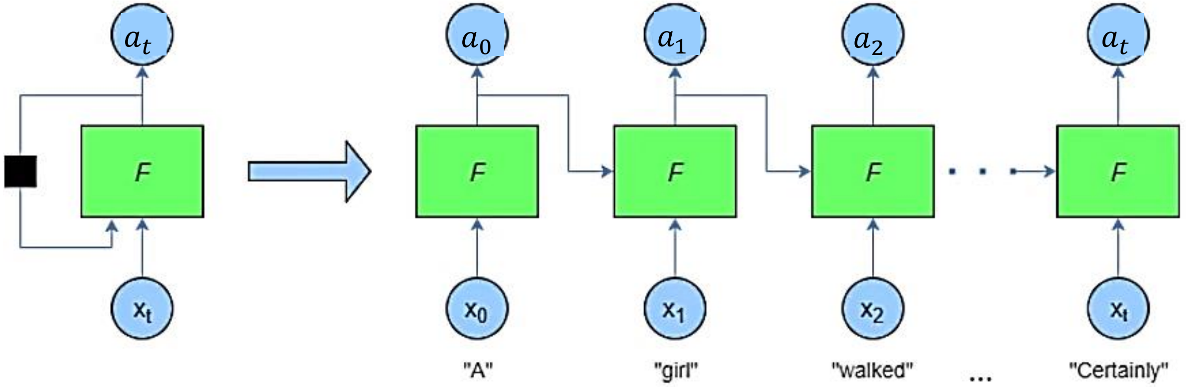
Unrolled RNN

Serial-to- parallel conversion of data sequence to supply a stream of data to the RNN

https://adventuresinmachinelearning.com/recurrent-neural-networks-lstm-tutorial-tensorflow/

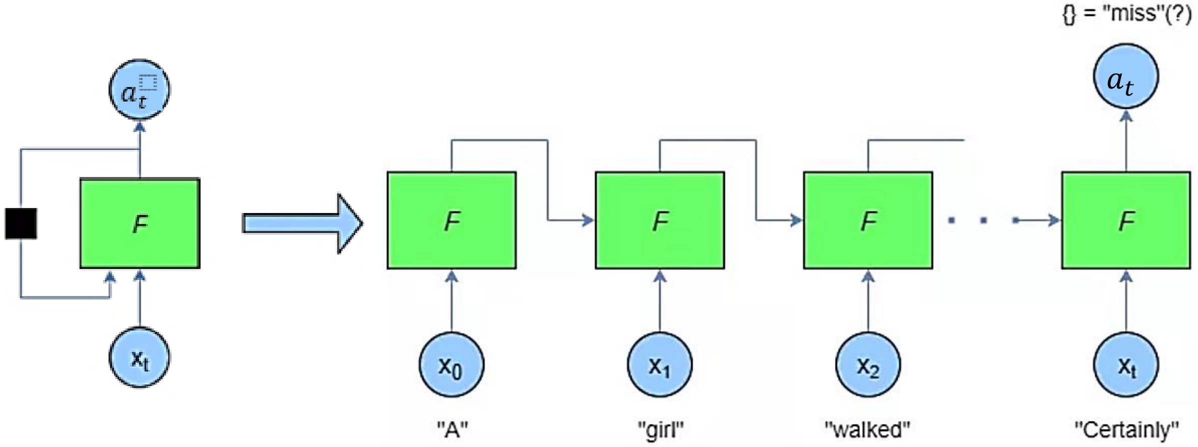
input-to-activation model

many-to-many model

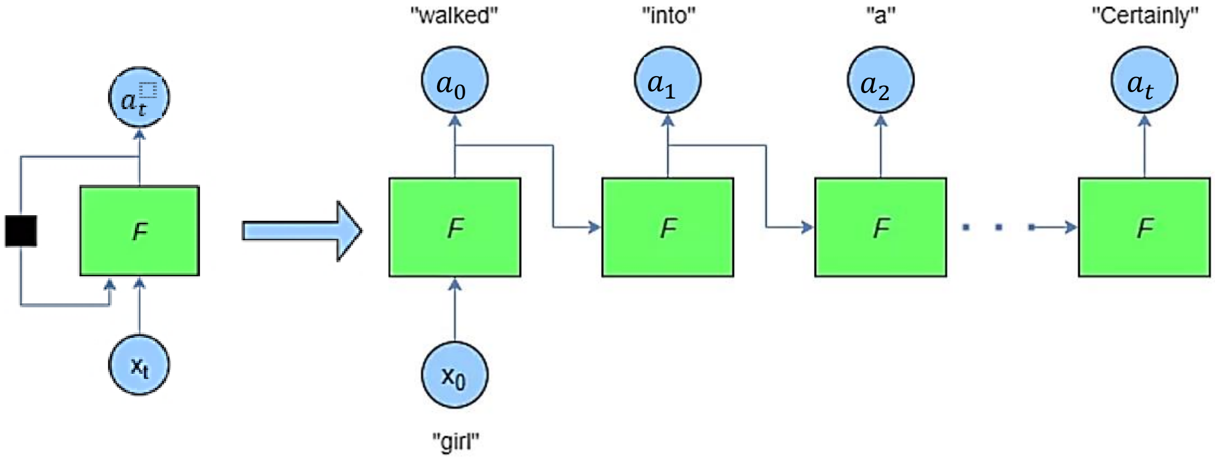


inputs: “A girl walked into a bar…”

outputs (predicted): h_0 to h_t.



many-to-one model



one-to-many model

:om/recurrent-neural-networks-lstm-tutorial-tensorflow/

Basic RNN - critical analyses

For RNN, ideally, we would want to have long memories (many time steps), so the network can connect data relationships at significant distances in time.

An RNN with long memory could make real progress in understanding how language and narrative work, how stock market events are correlated, etc.

But

RNNs present a **major setback**

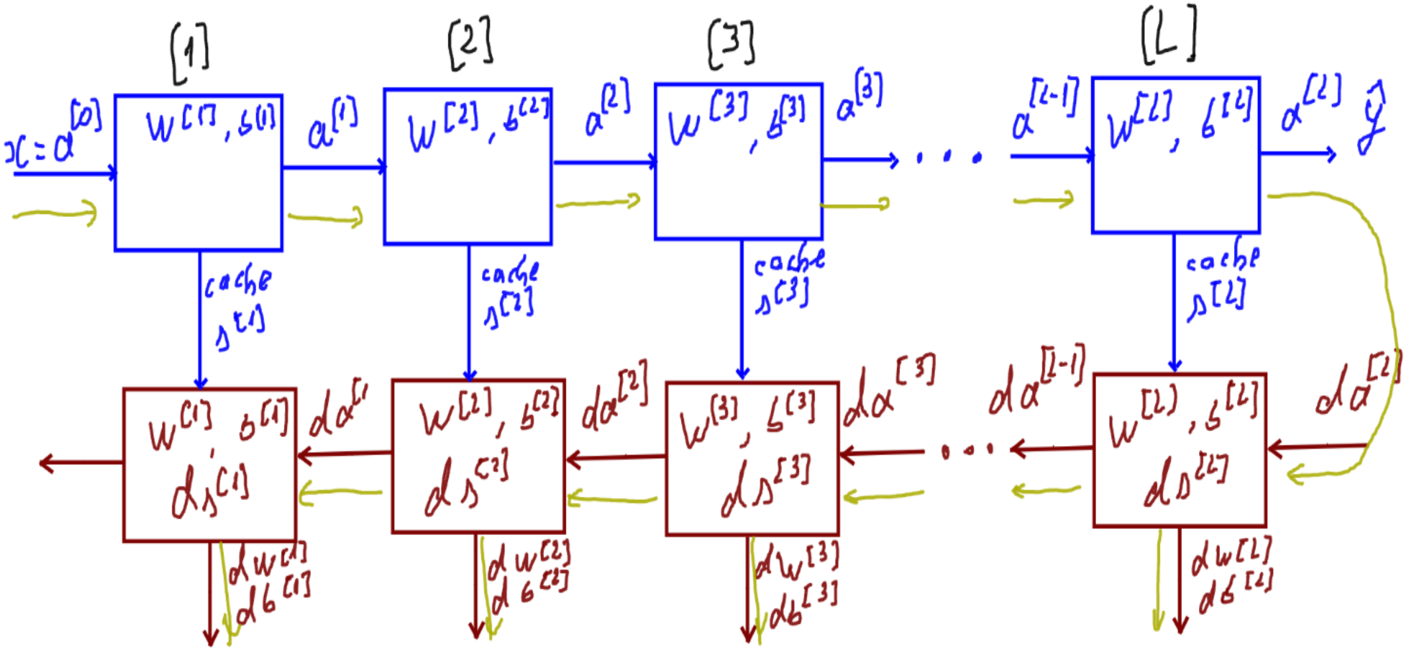
- ○ **vanishing gradient / exploding gradient**

They have difficulties in learning long-range dependencies (relationship between entities that are several steps apart).

The more time steps we have, the more chance we have of back-propagation error gradients:

- **accumulating** and exploding (for values > 1)
- **vanishing** down to nothing (for values < 1)

Forward and backward propagation for a DNN



▬ forward prop

▬ backward prop

▬ full iteration

$$w^{[l]} := w^{[l]} - \eta\, dw^{[l]}$$

$$b^{[l]} := b^{[l]} - \eta\, db^{[l]}$$

η - learning rate

For each layer $W := w - \eta\, dw$ $\quad dw = \frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial s} \cdot \frac{\partial s}{\partial w} = \frac{\partial \mathcal{L}}{\partial y} \cdot \frac{\partial \hat{y}}{\partial s} \cdot \frac{\partial s}{w}$

For multiple layer – multiplications accumulate for all layers

Basic RNN - critical analyses – cont.

In deep networks or recurrent neural networks, **error gradients can accumulate** during an update and result in very large gradients.

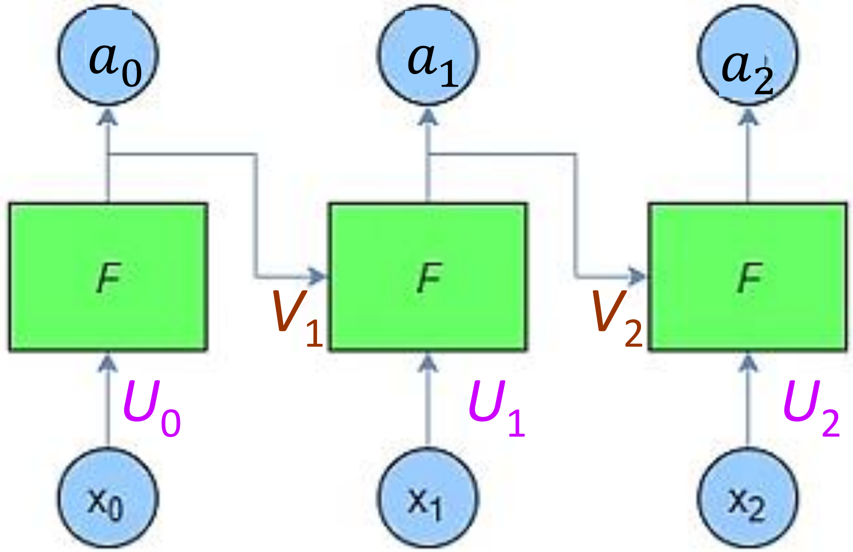
The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1.0.

These in turn result in large updates to the network weights, and in turn, an unstable network.
At an extreme, the values of weights can become so large as to overflow and result in NaN values.

When *n* hidden layers use an activation that give small gradients (below unity, like the sigmoid function), *n* small derivatives are multiplied together. Thus, the **error gradient decreases exponentially** as we propagate down to the initial layers.

A small gradient means that the weights and biases of the initial layers will not be updated effectively with each training session. Since these initial layers are often crucial to recognizing the core elements of the input data, it can lead to **overall inaccuracy** of the whole network.

Basic RNN - critical analyses



$$a_2 = F\left(U_2 x_2 + V_2 \cdot \left(F\left(U_1 x_1 + V_1 \cdot \left(F(U_0 x_0)\right)\right)\right)\right)$$

For back-propagation we compute the gradients of the activation function

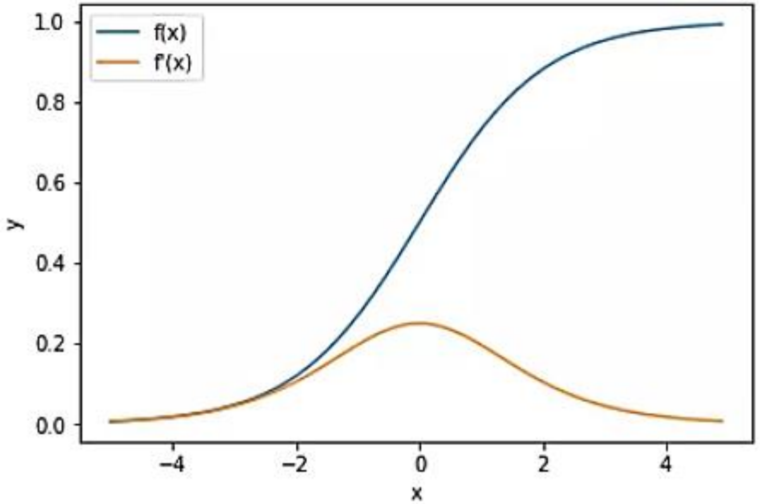
The problem with the sigmoid-type activation function occurs when the input values are such that the output is close to either 0 or 1:

- the gradient is very small

Multiplying many sigmoid gradients: → 0

Vanishing gradients

Solution: LSTM neural network



LSTM network Long Short-Term Memory

To reduce the vanishing/exploding gradient problem, **reduce the multiplication** of gradients.

The **LSTM** cell is a specifically designed unit of logic that help **reduce the gradient problem** sufficiently to make recurrent neural networks more useful for **long-term memory tasks** i.e. text sequence predictions.

The way it does so is by creating **an internal memory state** which is simply *added* to the processed input, which greatly reduces the multiplicative effect of small gradients.

The **time dependence and effects of previous inputs** are controlled by an interesting concept called a ***forget gate***, which determines which states are **remembered or forgotten.**

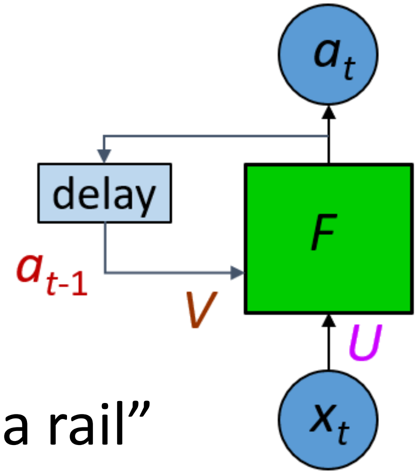
- selectively remember or forget information over time

Two other gates, the ***input gate*** and ***output gate***, are also featured in LSTM cells.

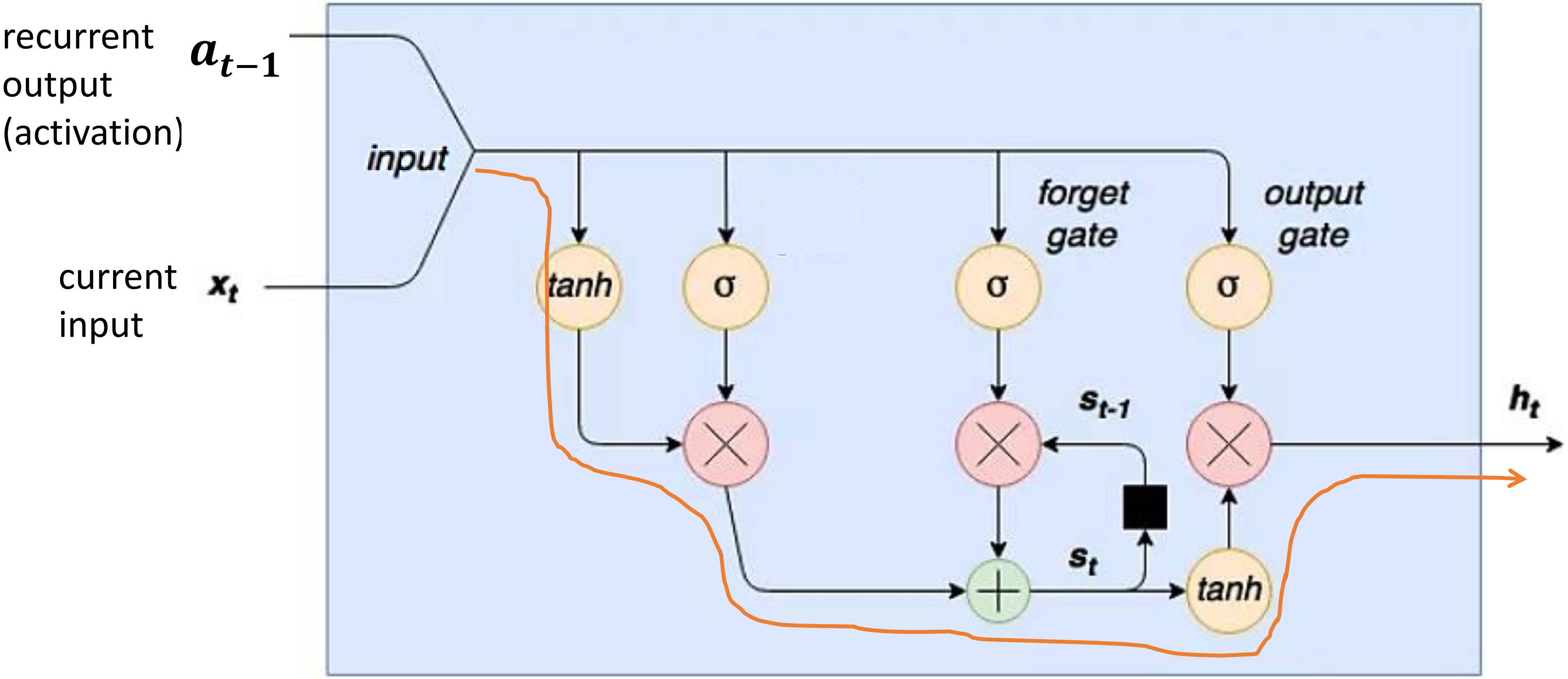
LSTM excels at capturing long-range dependencies in sequences.

https://adventuresinmachinelearning.com/recurrent-neural-networks-lstm-tutorial-tensorflow/

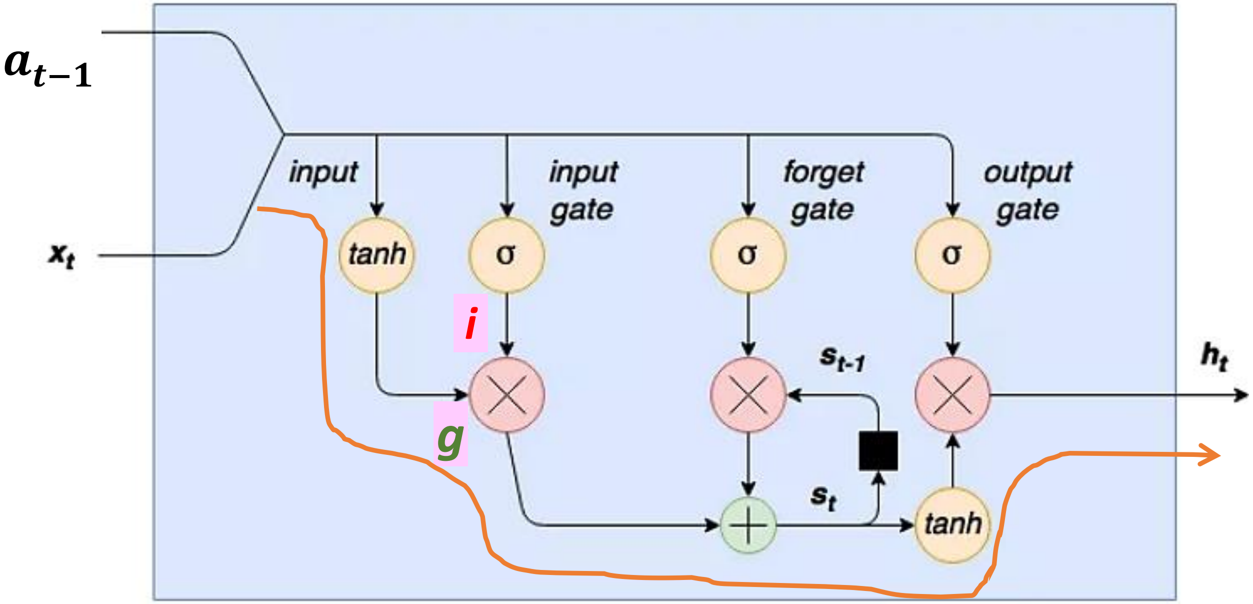
LSTM cell structure

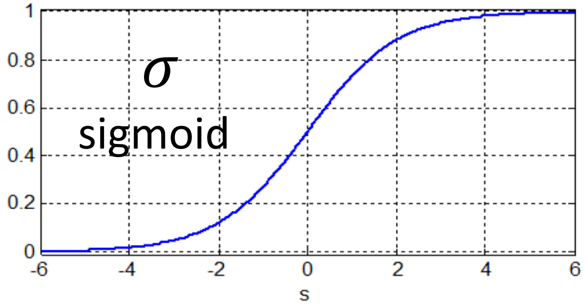


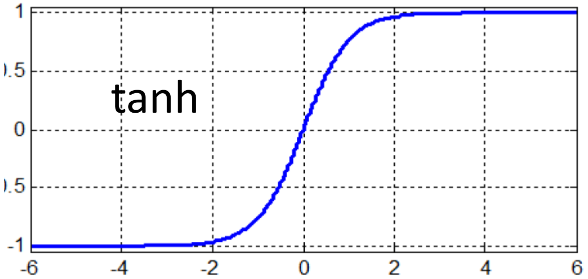
x_t and a_{t-1} concatenated together enters the top “data rail”



https://adventuresinmachinelearning.com/recurrent-neural-networks-lstm-tutorial-tensorflow/







g - computed input

i - switch

U - weight matrix for input

V - weight matrix for recurrent output

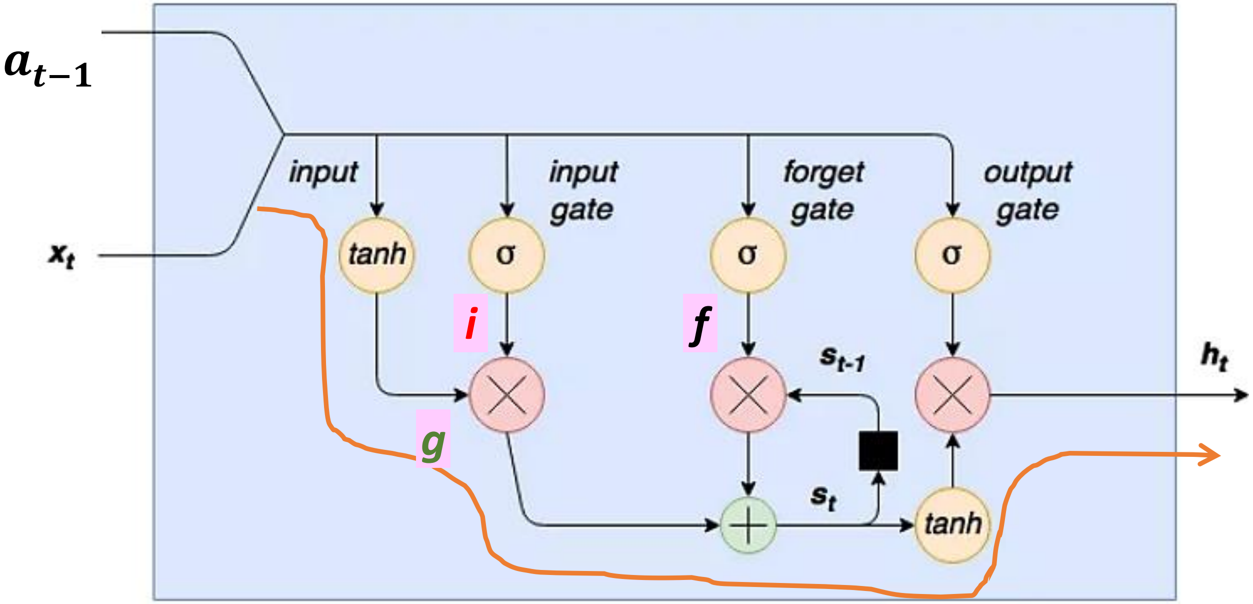
$$g = tanh(b^g + x_t U^g + a_{t-1} V^g)$$

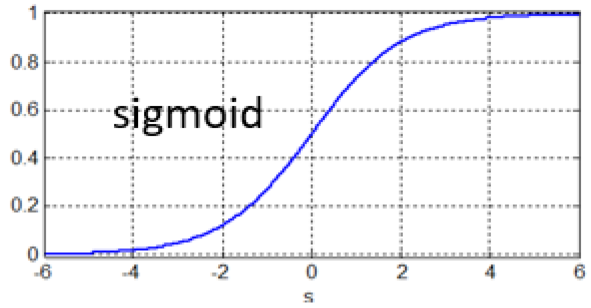
$$i = \sigma(b^i + x_t U^i + a_{t-1} V^i)$$

The value of *i* is "learned" during the training by its b^i, U^i, V^i

The **input gate** acts as a **filter** determining which inputs (through *g*) are switched on and off (*i* – between 0 and 1)

g and *i* - multiplied element-wise ($g \circ i$) giving the output of the input stage





self-reccurent

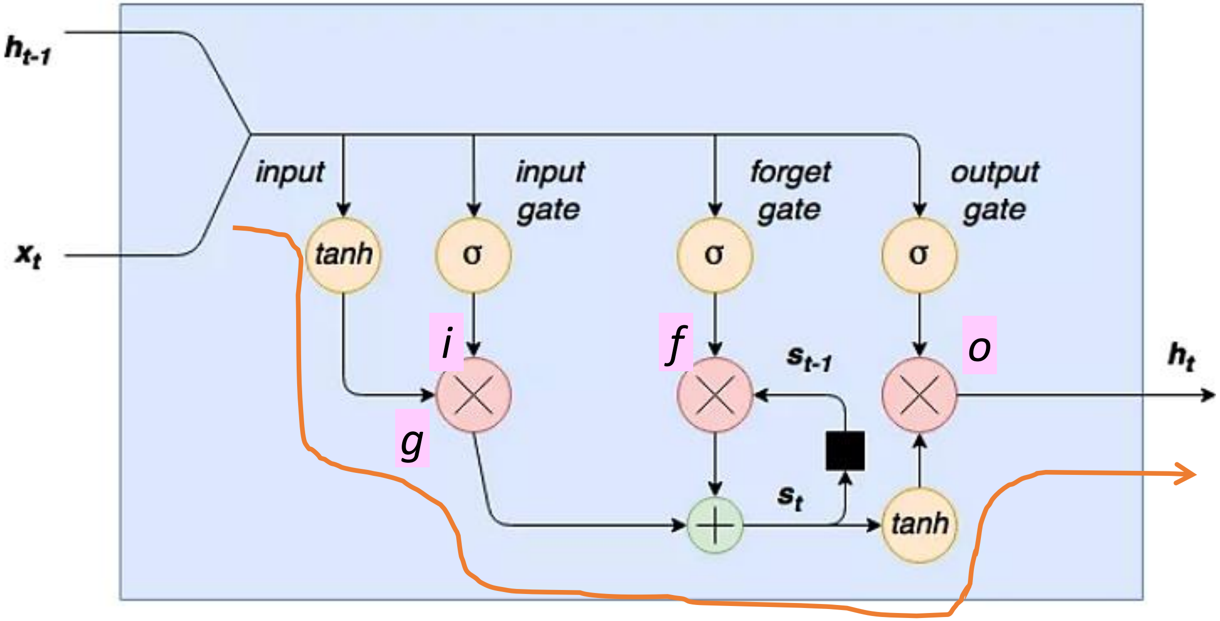
Forget gate is a sigmoid activated set of nodes which is element-wise multiplied by $s_{t\text{-}1}$ to determine which **previous states** should be

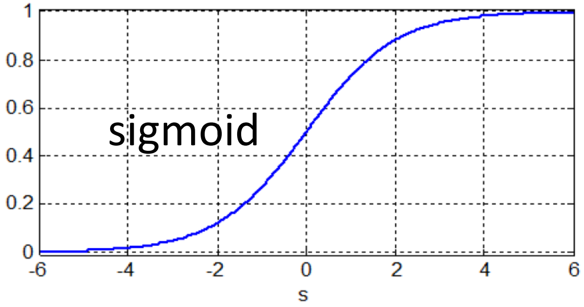
- remembered (i.e. forget gate output close to 1, f → 1), $\boldsymbol{s_{t\text{-}1}}$ is remembered (add to $\boldsymbol{s_t}$)
- forgotten (i.e. forget gate output close to 0, f → 0), $\boldsymbol{s_{t\text{-}1}}$ is forgotten (no add to $\boldsymbol{s_t}$)

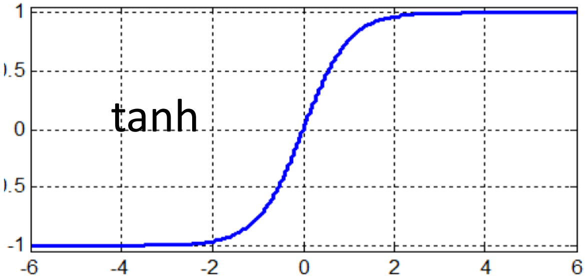
$$f = \sigma(b^f + x_t U^f + a_{t-1} V^f) \qquad s_t = s_{t-1} \circ f + g \circ i$$

The forget-gate: "filtered" state is **simply added to the input, rather than multiplied by it**, or mixed with it via weights and a sigmoid activation function as occurs in a standard recurrent neural network.

This is important to **reduce the issue of vanishing gradients**.







The output gate has two components

- *tanh* squashing function
- output sigmoid gating function.

The output sigmoid gating function determine which values of the state are output from the cell (values of the output gate close to 1, *o*=1).

$$o = \sigma(b^o + x_t U^o + a_{t-1} V^o) \qquad h_t = tanh(s_t) \circ o$$

The **LSTM cell** is very flexible, with gating functions controlling

✓ what is taken as input,

✓ what is “remembered” in the internal state variable,

✓ what is output from the LSTM cell.

https://adventuresinmachinelearning.com/recurrent-neural-networks-lstm-tutorial-tensorflow/

Case study

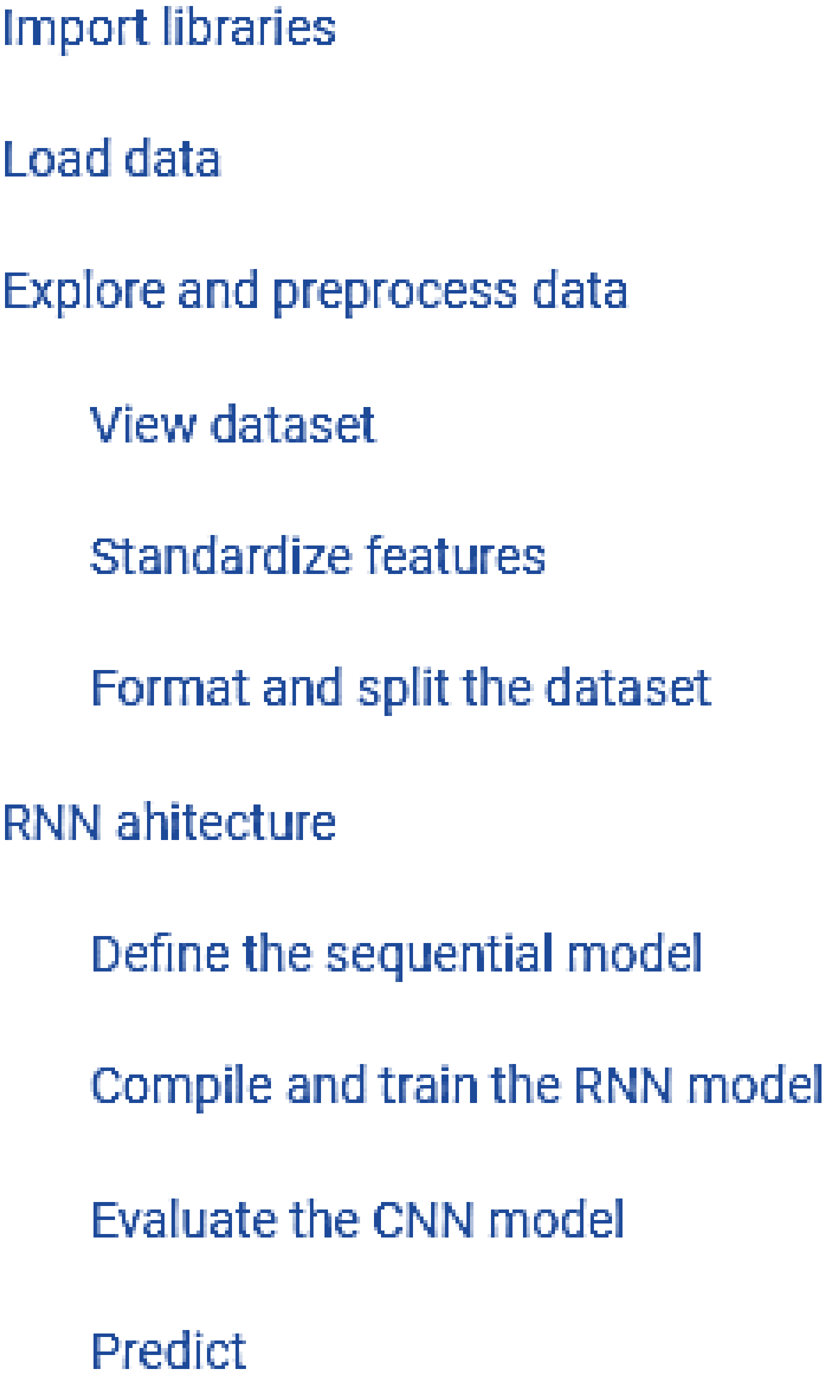
- Implement a time series analysis using a RNN (LSTM) to predict the prices of Bitcoin using historical data from CryptoDataDownload

Python, TensorFlow

Colaboratory

Application flowchart

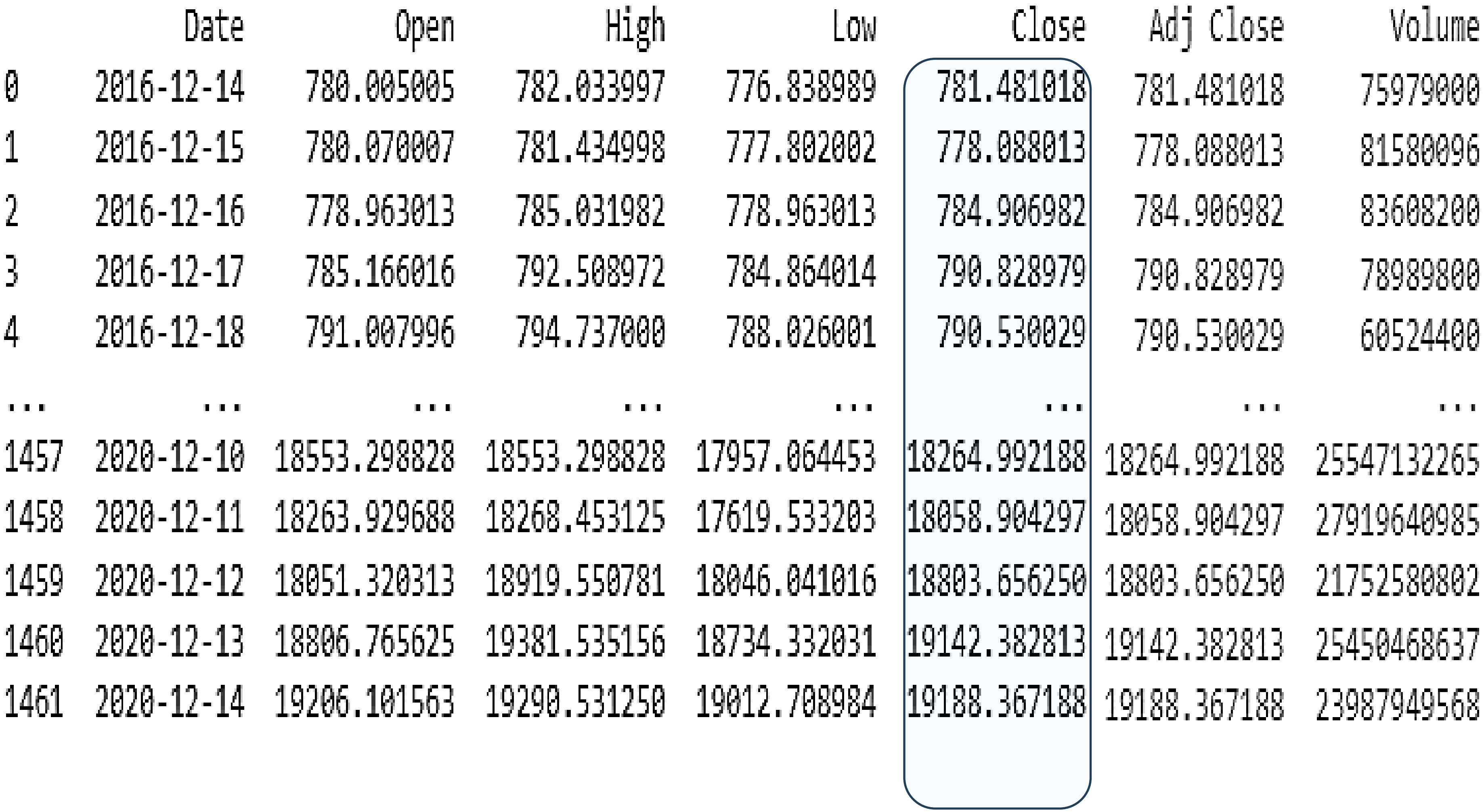
Uses TensorFlow



- Import libraries
- Load data
- Explore and preprocess data
    - View dataset
    - Standardize features
    - Format and split the dataset
- RNN ahitecture
    - Define the sequential model
    - Compile and train the RNN model
    - Evaluate the CNN model
    - Predict

Original data

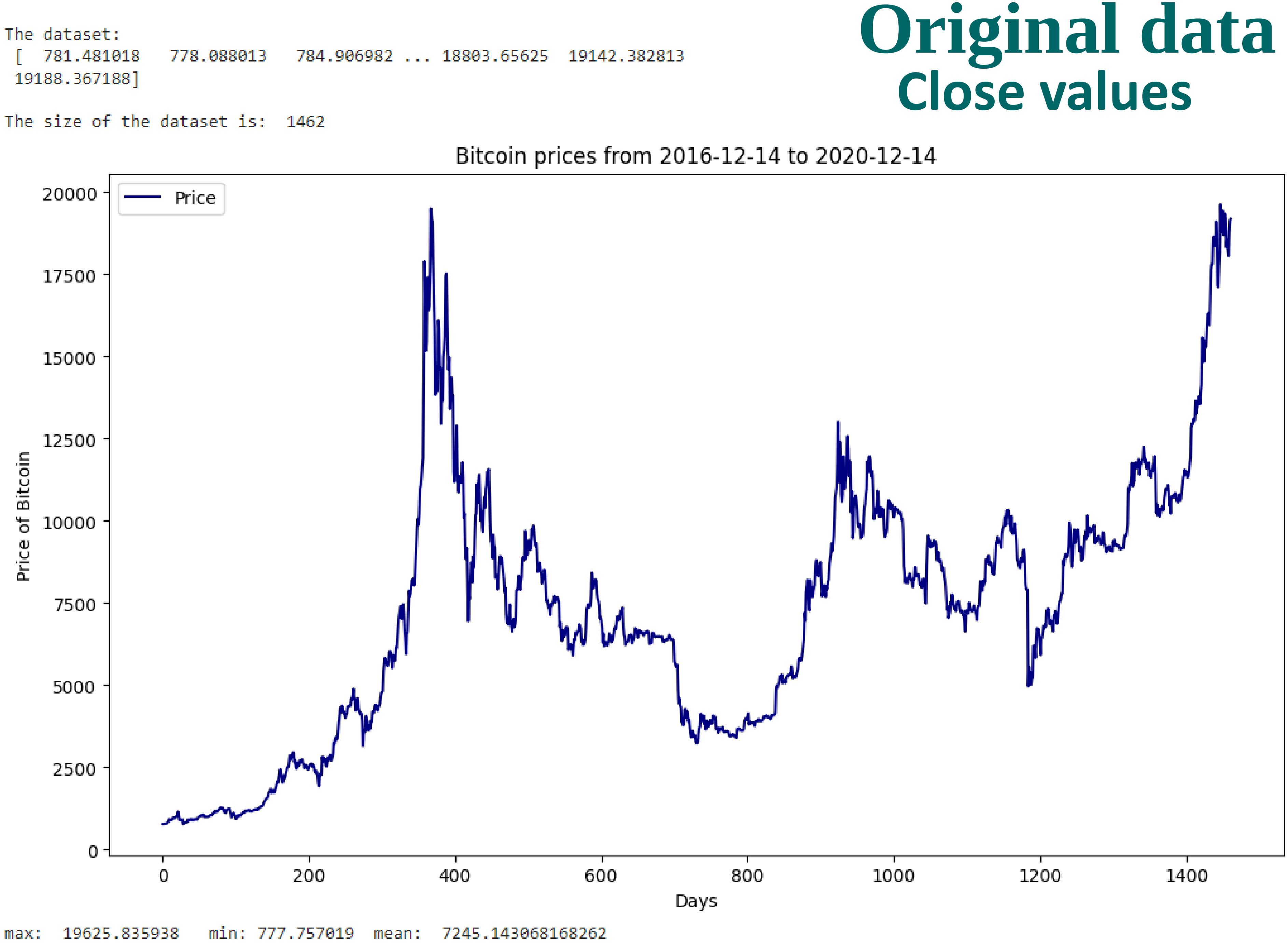
. csv



```
            Date          Open          High           Low         Close     Adj Close       Volume
0     2016-12-14    780.005005    782.033997    776.838989    781.481018    781.481018     75979000
1     2016-12-15    780.070007    781.434998    777.802002    778.088013    778.088013     81580096
2     2016-12-16    778.963013    785.031982    778.963013    784.906982    784.906982     83608200
3     2016-12-17    785.166016    792.508972    784.864014    790.828979    790.828979     78989800
4     2016-12-18    791.007996    794.737000    788.026001    790.530029    790.530029     60524400
...          ...           ...           ...           ...           ...           ...          ...
1457  2020-12-10  18553.298828  18553.298828  17957.064453  18264.992188  18264.992188  25547132265
1458  2020-12-11  18263.929688  18268.453125  17619.533203  18058.904297  18058.904297  27919640985
1459  2020-12-12  18051.320313  18919.550781  18046.041016  18803.656250  18803.656250  21752580802
1460  2020-12-13  18806.765625  19381.535156  18734.332031  19142.382813  19142.382813  25450468637
1461  2020-12-14  19206.101563  19290.531250  19012.708984  19188.367188  19188.367188  23987949568
```

Original data

Close values



```
The dataset:
 [  781.481018   778.088013   784.906982 ... 18803.65625  19142.382813
 19188.367188]

The size of the dataset is:  1462
```

```
max:  19625.835938   min: 777.757019   mean:   7245.143068168262
```

Standardize features - normalization

Standardize features by removing the mean and scaling to unit variance. The standard score of a sample x is calculated as:

$$z = (x - u) / s$$

u is the mean of the training samples
s is the standard deviation of the training samples.

Centering and scaling happen independently on each feature by computing the relevant statistics on the samples in the training set.

Mean and standard deviation are then stored to be used on later data using transform.

Standardization of a dataset is a common requirement for many machine learning estimators: they might behave badly if the individual features do not more or less look like standard normally distributed data (e.g. Gaussian with 0 mean and unit variance).

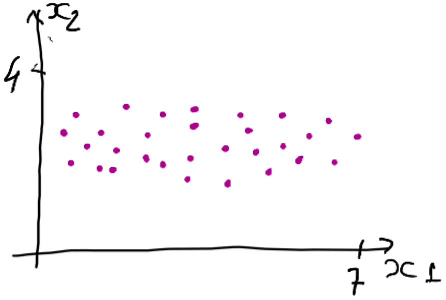
Standard Scaler

$$x\_scaled = \frac{x - x\_mean}{standard\_deviation}$$

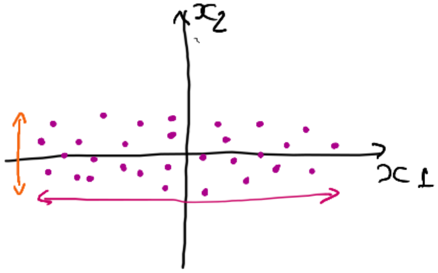
Applied separately on each data feature

Use the same μ, σ to normalize all data sets

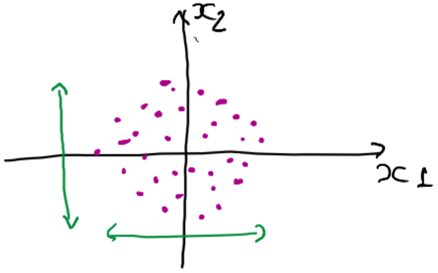
- ✓ **Training**
- ✓ **Validation**
- ✓ **Test**



Initial dataset



Subtract mean (zero out the mean)



Normalize the variance

$x = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}$

$\mu = \frac{1}{m}\sum_{i=1}^{m} x^{(i)}$; $\mu = \begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}$

$x := x - \mu$

μ – mean

$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m} \left(x^{(i)}\right)^2$

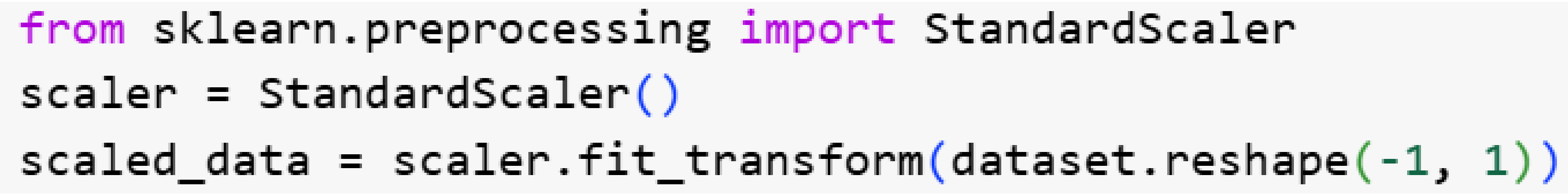
σ^2 – variance

$x := \frac{x}{\sigma}$

σ – standard deviation $\sigma = \begin{bmatrix} \sigma_1 \\ \sigma_2 \end{bmatrix}$

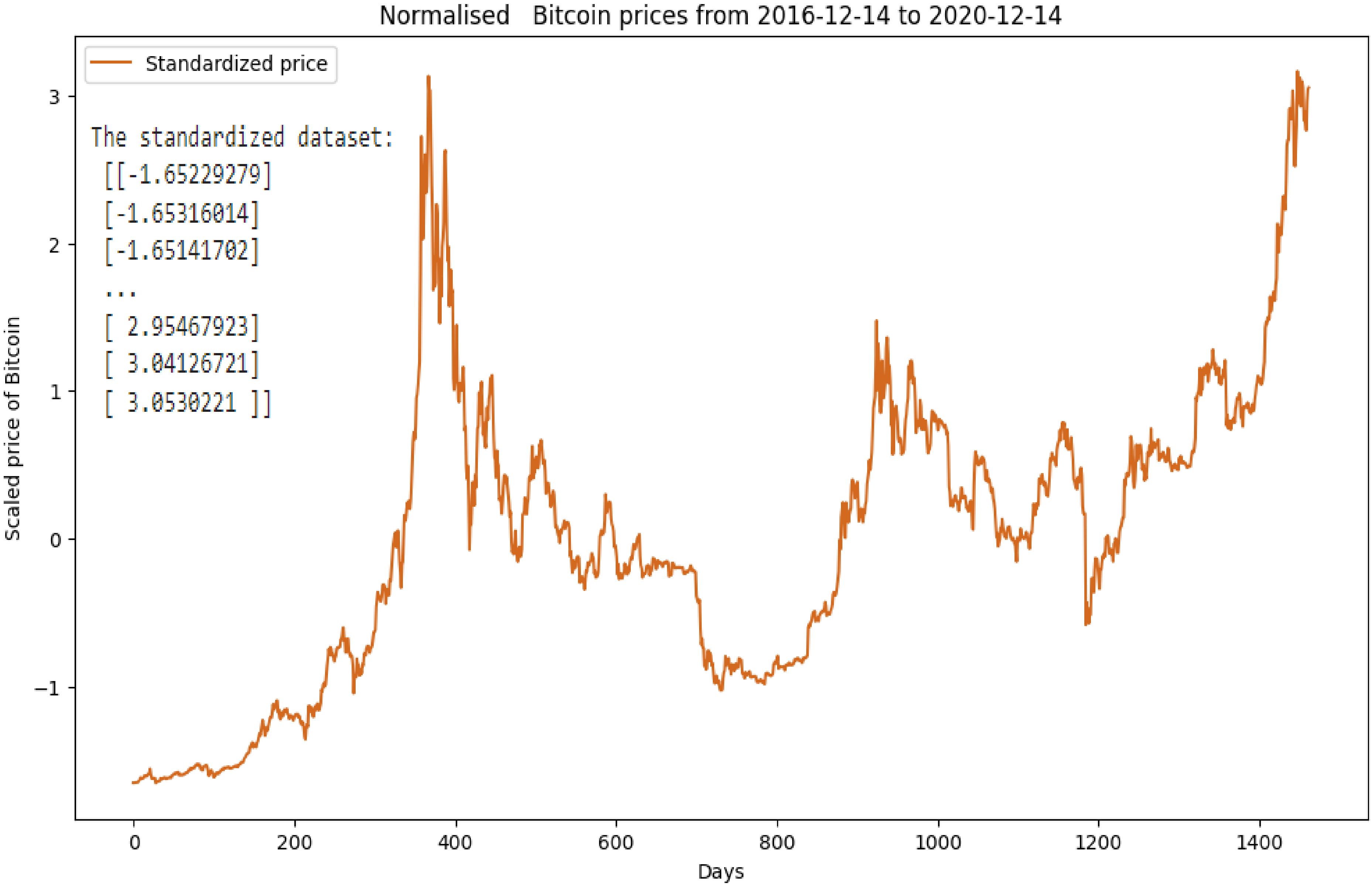
Standardizes features by removing the mean and scaling to unit variance.

1. **Centering:** The mean of the feature is subtracted from each feature value (x). This shifts the distribution of the feature so that its mean becomes 0.
2. **Scaling:** Each centered feature value is then divided by the standard deviation. This scales the distribution so that its variance becomes 1.



```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
scaled_data = scaler.fit_transform(dataset.reshape(-1, 1))
```

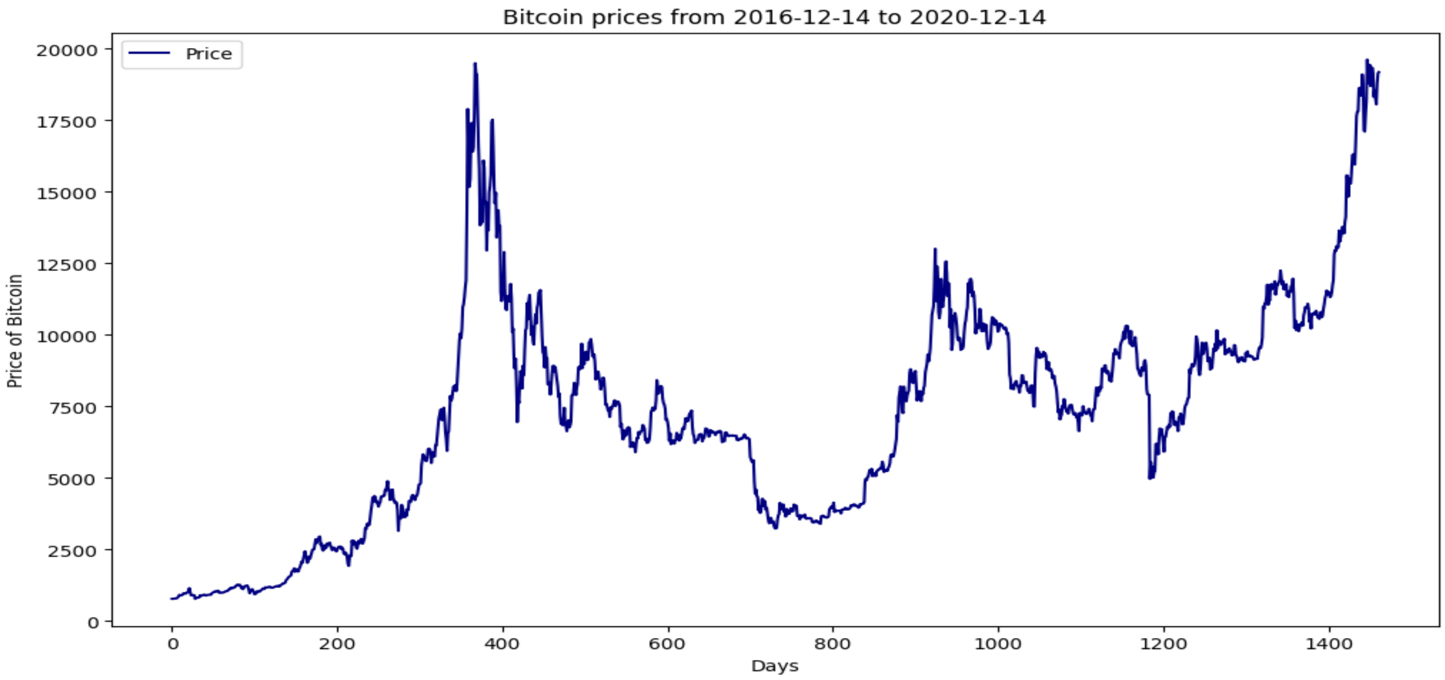
Standardized data



```
The standardized dataset:
 [[-1.65229279]
 [-1.65316014]
 [-1.65141702]
 ...
 [ 2.95467923]
 [ 3.04126721]
 [ 3.0530221 ]]
```

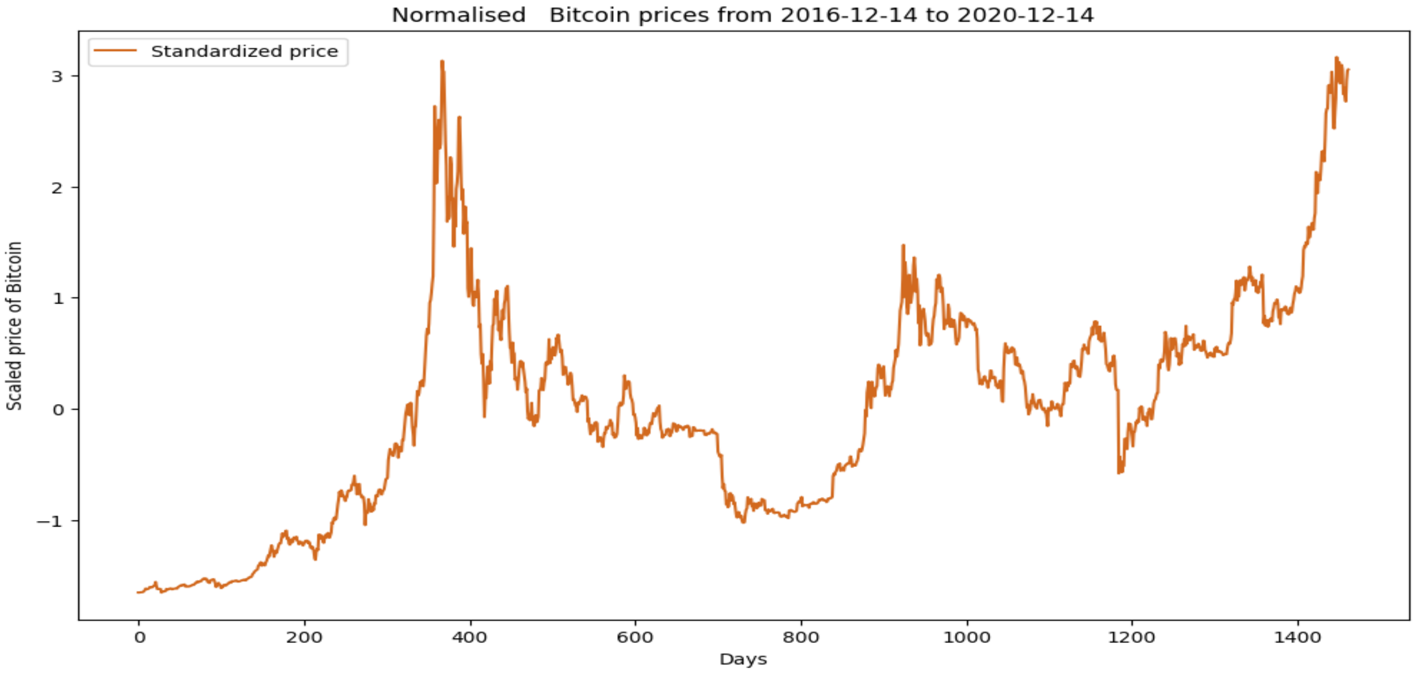
```
max:  [3.16485135]   min: [-1.65324475]  mean:  7.776117491218607e-17
```

Original data



```
max:   19625.835938    min: 777.757019   mean:   7245.143068168262
```

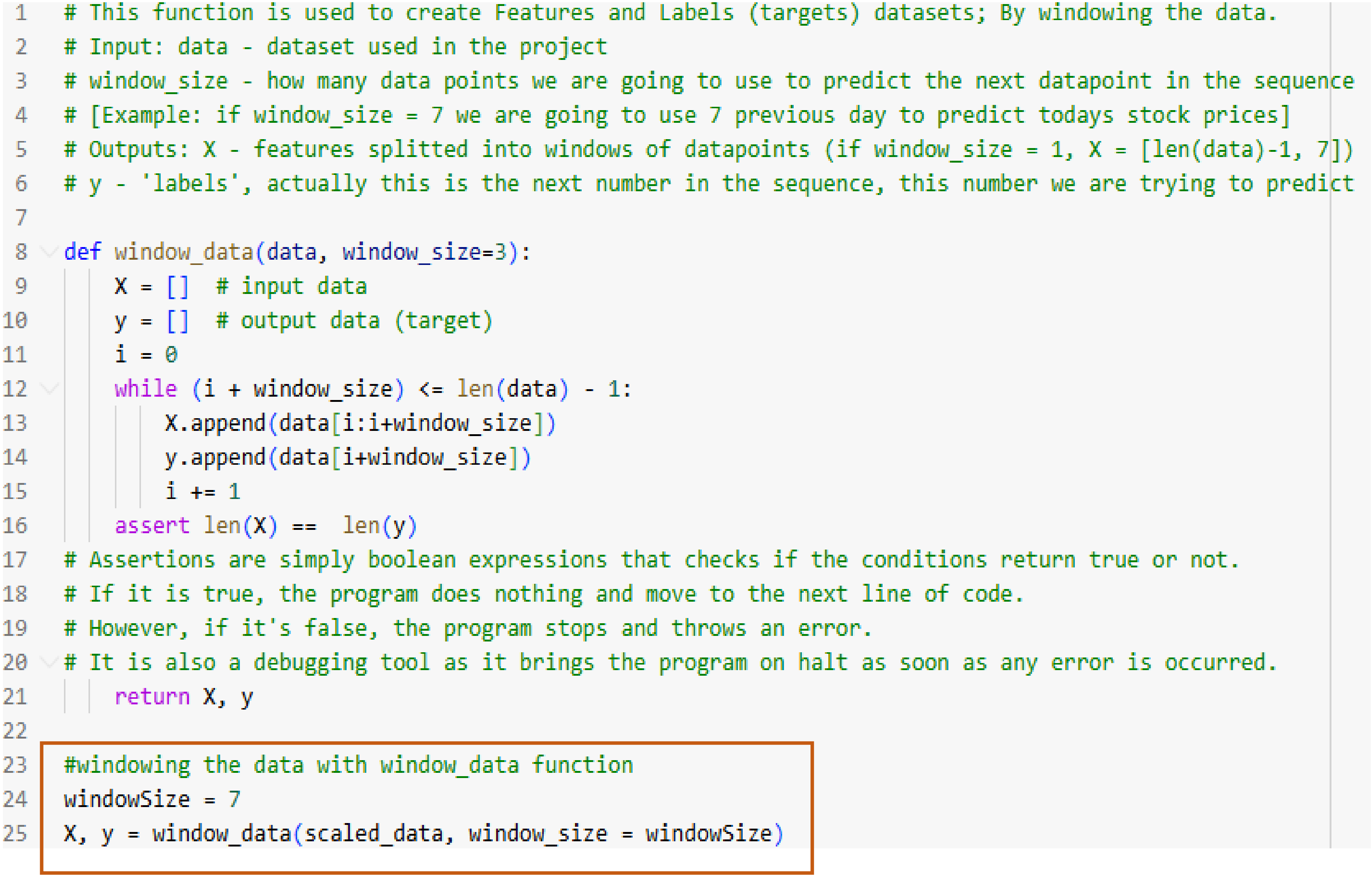
Standardized data



```
max:   [3.16485135]    min: [-1.65324475]   mean:   7.776117491218607e-17
```

Data formatting

window_size = 7; The number of previous days we consider to predict the bitcoin price for our case.

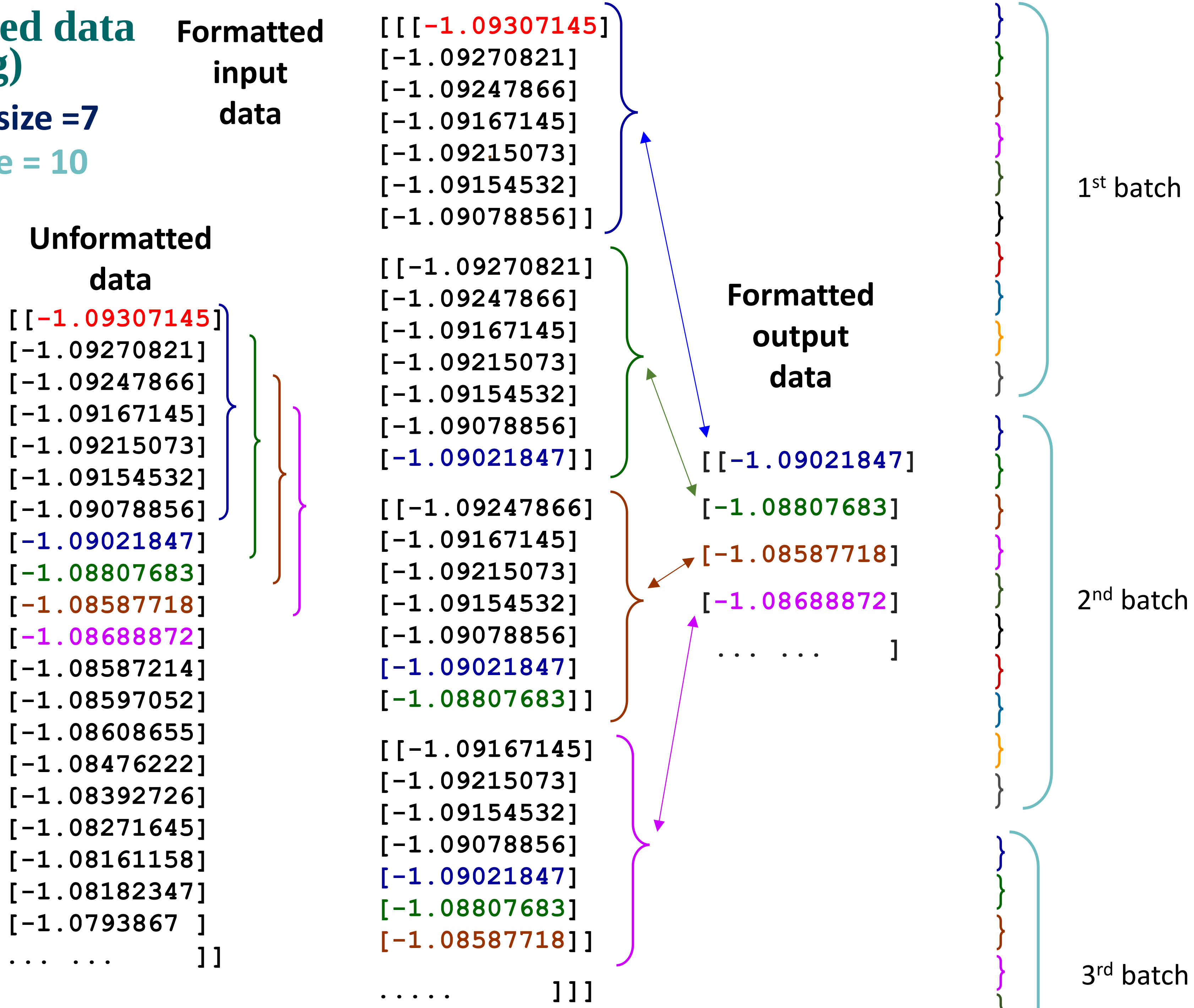


```python
# This function is used to create Features and Labels (targets) datasets; By windowing the data.
# Input: data - dataset used in the project
# window_size - how many data points we are going to use to predict the next datapoint in the sequence
# [Example: if window_size = 7 we are going to use 7 previous day to predict todays stock prices]
# Outputs: X - features splitted into windows of datapoints (if window_size = 1, X = [len(data)-1, 7])
# y - 'labels', actually this is the next number in the sequence, this number we are trying to predict

def window_data(data, window_size=3):
    X = []  # input data
    y = []  # output data (target)
    i = 0
    while (i + window_size) <= len(data) - 1:
        X.append(data[i:i+window_size])
        y.append(data[i+window_size])
        i += 1
    assert len(X) ==  len(y)
# Assertions are simply boolean expressions that checks if the conditions return true or not.
# If it is true, the program does nothing and move to the next line of code.
# However, if it's false, the program stops and throws an error.
# It is also a debugging tool as it brings the program on halt as soon as any error is occurred.
    return X, y

#windowing the data with window_data function
windowSize = 7
X, y = window_data(scaled_data, window_size = windowSize)
```

Formatted data (training)

window_size =7

batch_size = 10



Unformatted data

```
[[-1.09307145]
[-1.09270821]
[-1.09247866]
[-1.09167145]
[-1.09215073]
[-1.09154532]
[-1.09078856]
[-1.09021847]
[-1.08807683]
[-1.08587718]
[-1.08688872]
[-1.08587214]
[-1.08597052]
[-1.08608655]
[-1.08476222]
[-1.08392726]
[-1.08271645]
[-1.08161158]
[-1.08182347]
[-1.0793867 ]
...  ...      ]]
```

Formatted input data

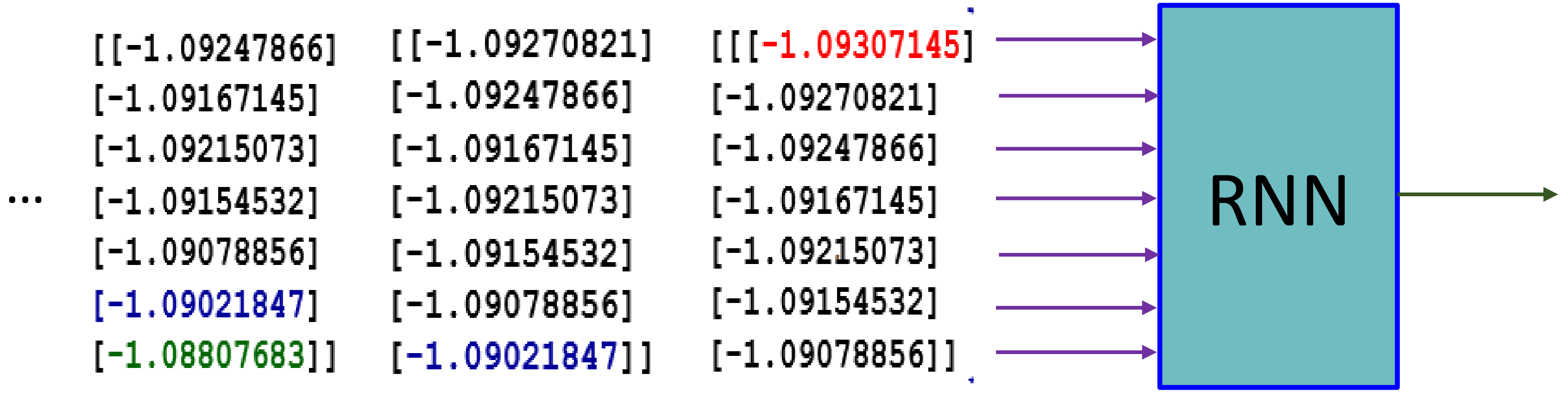
```
[[[-1.09307145]
[-1.09270821]
[-1.09247866]
[-1.09167145]
[-1.09215073]
[-1.09154532]
[-1.09078856]]

[[-1.09270821]
[-1.09247866]
[-1.09167145]
[-1.09215073]
[-1.09154532]
[-1.09078856]
[-1.09021847]]

[[-1.09247866]
[-1.09167145]
[-1.09215073]
[-1.09154532]
[-1.09078856]
[-1.09021847]
[-1.08807683]]

[[-1.09167145]
[-1.09215073]
[-1.09154532]
[-1.09078856]
[-1.09021847]
[-1.08807683]
[-1.08587718]]

.....         ]]]
```

Formatted output data

```
[[-1.09021847]
[-1.08807683]
[-1.08587718]
[-1.08688872]
 ...  ...     ]
```

Data flow

```
[[-1.09307145]
 [-1.09270821]
 [-1.09247866]
 [-1.09167145]
 [-1.09215073]
 [-1.09154532]
 [-1.09078856]
 [-1.09021847]
 [-1.08807683]
 [-1.08587718]
 [-1.08688872]
 [-1.08587214]
 [-1.08597052]
 [-1.08608655]
 [-1.08476222]
 [-1.08392726]
 [-1.08271645]
 [-1.08161158]
 [-1.08182347]
 [-1.0793867 ]
 ... ...      ]]
```



...

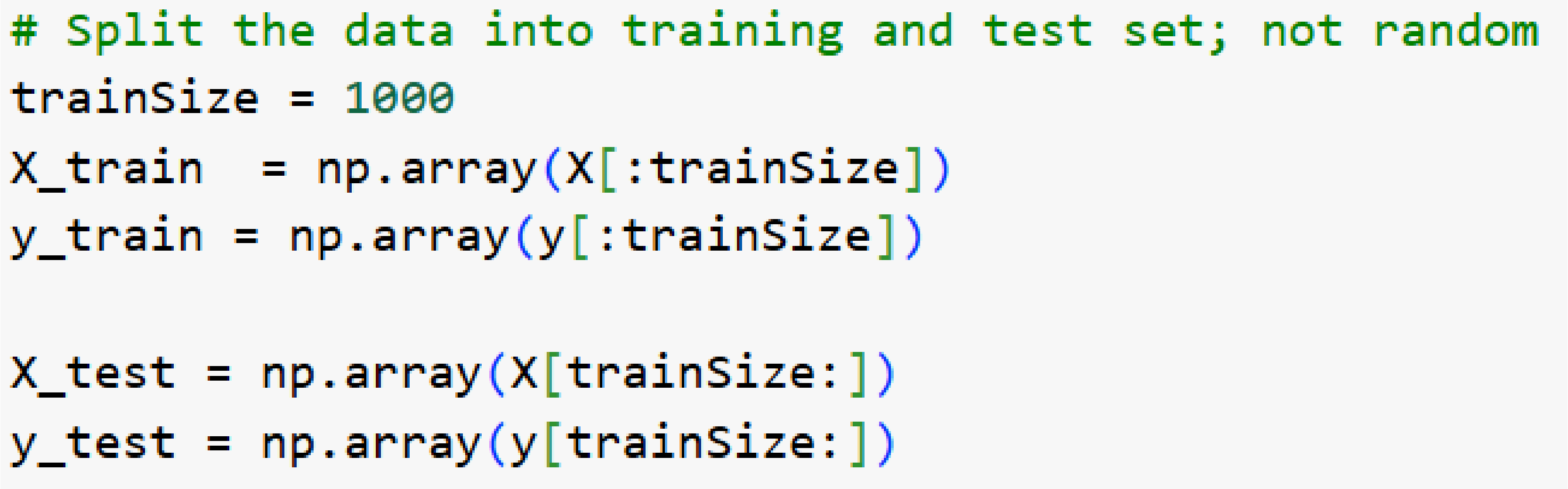
```
[[-1.09247866]
[-1.09167145]
[-1.09215073]
[-1.09154532]
[-1.09078856]
[-1.09021847]
[-1.08807683]]
```

```
[[-1.09270821]
[-1.09247866]
[-1.09167145]
[-1.09215073]
[-1.09154532]
[-1.09078856]
[-1.09021847]]
```

```
[[[-1.09307145]
[-1.09270821]
[-1.09247866]
[-1.09167145]
[-1.09215073]
[-1.09154532]
[-1.09078856]]
```

Splitting the dataset

Processing sequential data, where **the order of elements matters**



```
# Split the data into training and test set; not random
trainSize = 1000
X_train  = np.array(X[:trainSize])
y_train = np.array(y[:trainSize])

X_test = np.array(X[trainSize:])
y_test = np.array(y[trainSize:])
```

X_train size: (1000, 7, 1)

y_train size: (1000, 1)

X_test size: (455, 7, 1)
y_test size: (455, 1)

Defining the network

Hyperparameters

Hyperparameters explain higher-level structural information about the RNN model.

batch_size = 64; This is the number of windows of data we are passing at once.

window_size = 7; The number of previous days we consider to predict the bitcoin price for our case.

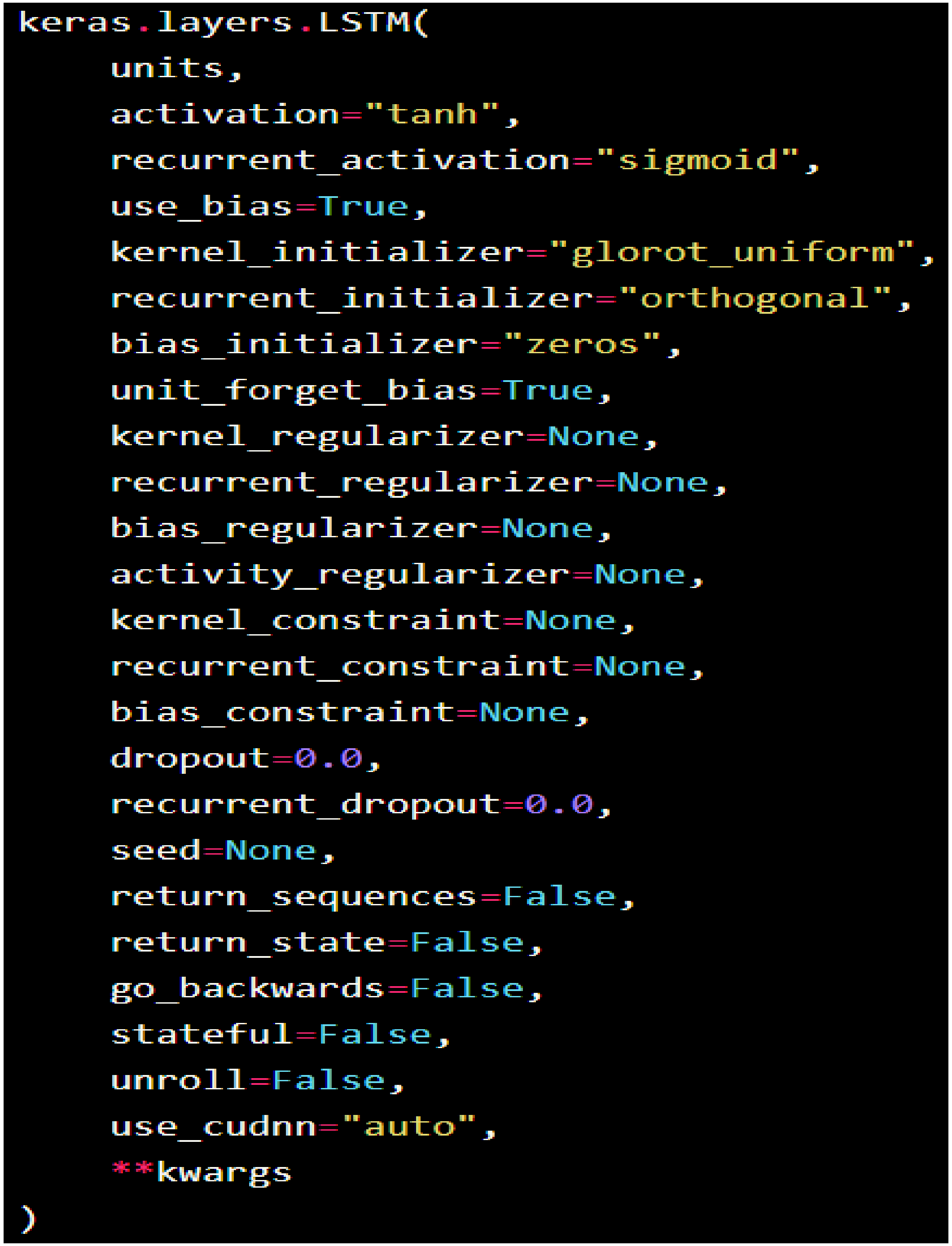
hidden_layers = 3; (LSTM units: 256, 512, 512)

clip_margin = 4; This is to prevent exploding the gradient (to clip gradients below/ above this margin).

learning_rate = 0.00005

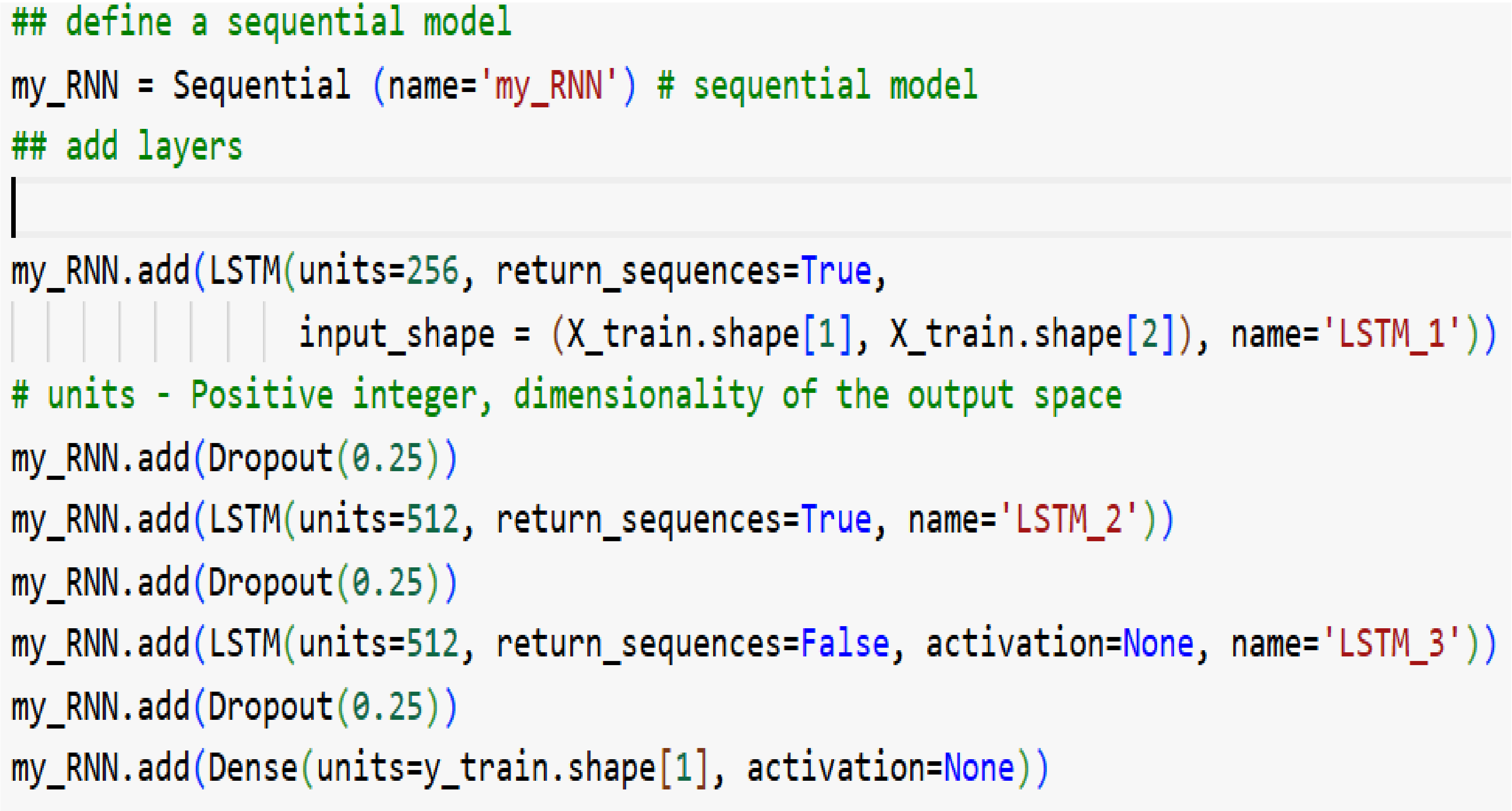
epochs = 500; This is the number of iterations (forward and back propagation) our model needs to make.

LSTM layer



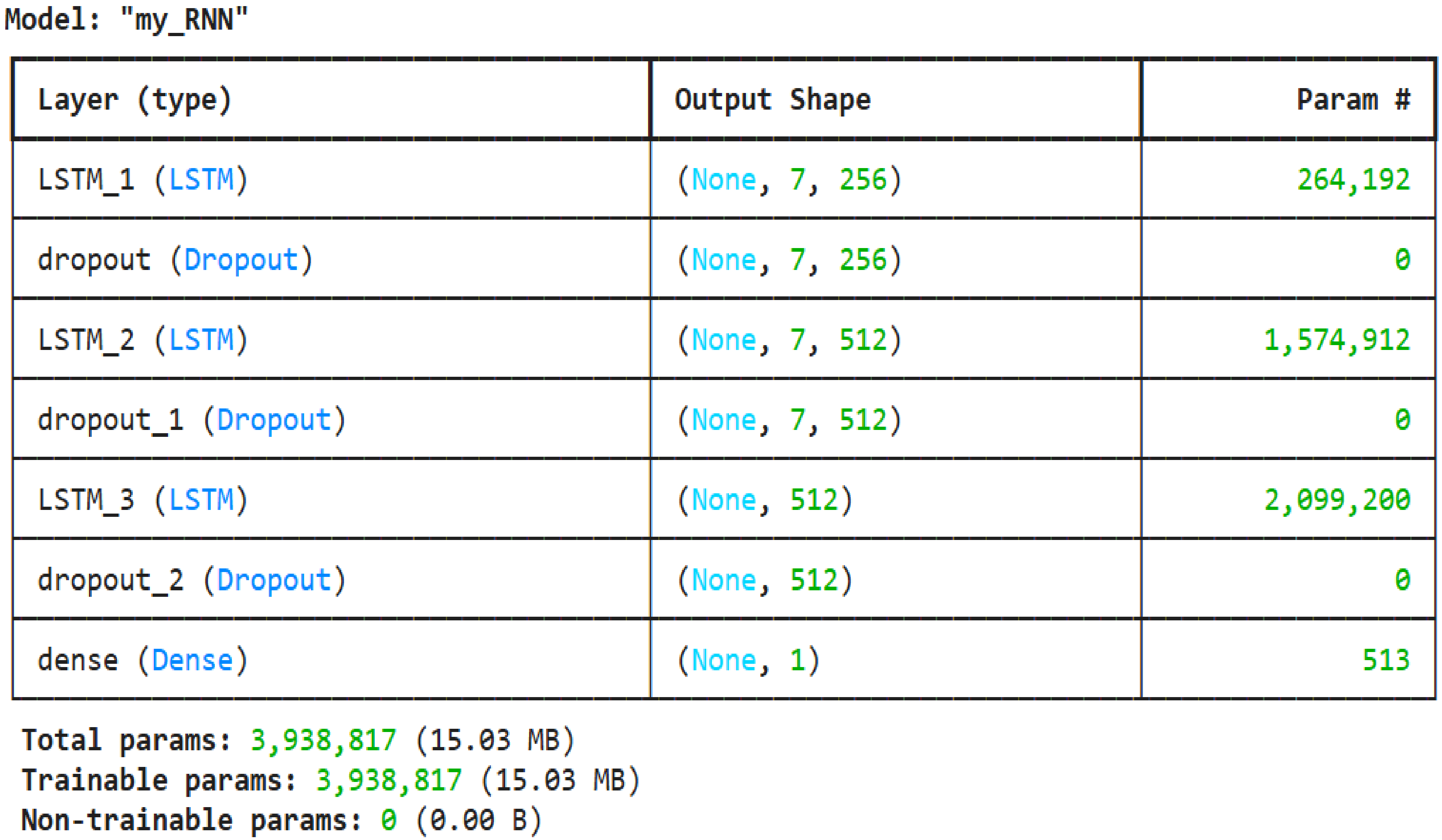
```
keras.layers.LSTM(
    units,
    activation="tanh",
    recurrent_activation="sigmoid",
    use_bias=True,
    kernel_initializer="glorot_uniform",
    recurrent_initializer="orthogonal",
    bias_initializer="zeros",
    unit_forget_bias=True,
    kernel_regularizer=None,
    recurrent_regularizer=None,
    bias_regularizer=None,
    activity_regularizer=None,
    kernel_constraint=None,
    recurrent_constraint=None,
    bias_constraint=None,
    dropout=0.0,
    recurrent_dropout=0.0,
    seed=None,
    return_sequences=False,
    return_state=False,
    go_backwards=False,
    stateful=False,
    unroll=False,
    use_cudnn="auto",
    **kwargs
)
```

Define the RNN model



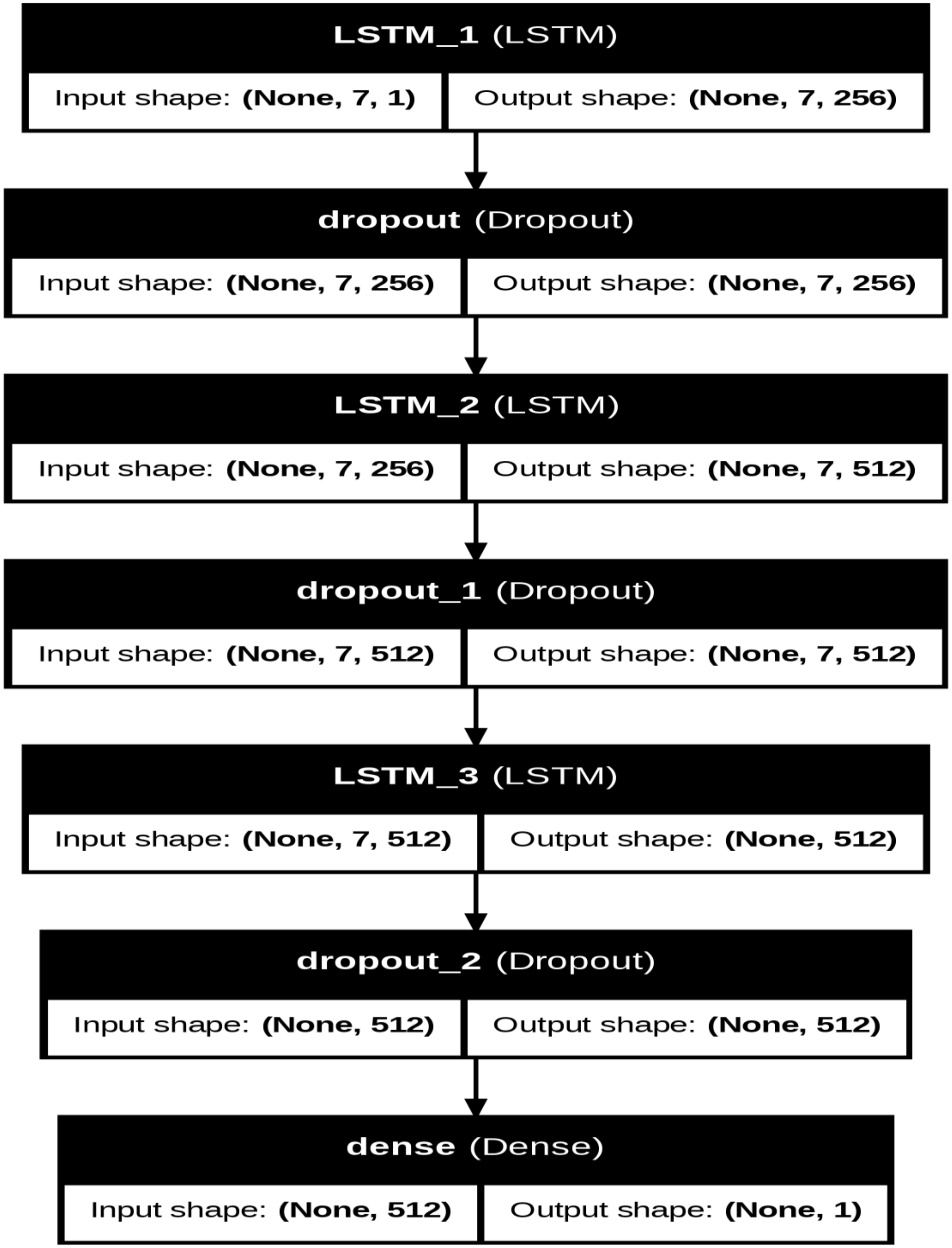
```
## define a sequential model
my_RNN = Sequential (name='my_RNN') # sequential model
## add layers

my_RNN.add(LSTM(units=256, return_sequences=True,
                input_shape = (X_train.shape[1], X_train.shape[2]), name='LSTM_1'))
# units - Positive integer, dimensionality of the output space
my_RNN.add(Dropout(0.25))
my_RNN.add(LSTM(units=512, return_sequences=True, name='LSTM_2'))
my_RNN.add(Dropout(0.25))
my_RNN.add(LSTM(units=512, return_sequences=False, activation=None, name='LSTM_3'))
my_RNN.add(Dropout(0.25))
my_RNN.add(Dense(units=y_train.shape[1], activation=None))
```

RNN model structure



```
Model: "my_RNN"
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━┓
┃ Layer (type)                  ┃ Output Shape           ┃       Param # ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━┩
│ LSTM_1 (LSTM)                 │ (None, 7, 256)         │       264,192 │
├───────────────────────────────┼────────────────────────┼───────────────┤
│ dropout (Dropout)             │ (None, 7, 256)         │             0 │
├───────────────────────────────┼────────────────────────┼───────────────┤
│ LSTM_2 (LSTM)                 │ (None, 7, 512)         │     1,574,912 │
├───────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_1 (Dropout)           │ (None, 7, 512)         │             0 │
├───────────────────────────────┼────────────────────────┼───────────────┤
│ LSTM_3 (LSTM)                 │ (None, 512)            │     2,099,200 │
├───────────────────────────────┼────────────────────────┼───────────────┤
│ dropout_2 (Dropout)           │ (None, 512)            │             0 │
├───────────────────────────────┼────────────────────────┼───────────────┤
│ dense (Dense)                 │ (None, 1)              │           513 │
└───────────────────────────────┴────────────────────────┴───────────────┘
 Total params: 3,938,817 (15.03 MB)
 Trainable params: 3,938,817 (15.03 MB)
 Non-trainable params: 0 (0.00 B)
```

RNN model structure



LSTM_1 (LSTM)

Input shape: **(None, 7, 1)** | Output shape: **(None, 7, 256)**

dropout (Dropout)

Input shape: **(None, 7, 256)** | Output shape: **(None, 7, 256)**

LSTM_2 (LSTM)

Input shape: **(None, 7, 256)** | Output shape: **(None, 7, 512)**

dropout_1 (Dropout)

Input shape: **(None, 7, 512)** | Output shape: **(None, 7, 512)**

LSTM_3 (LSTM)

Input shape: **(None, 7, 512)** | Output shape: **(None, 512)**

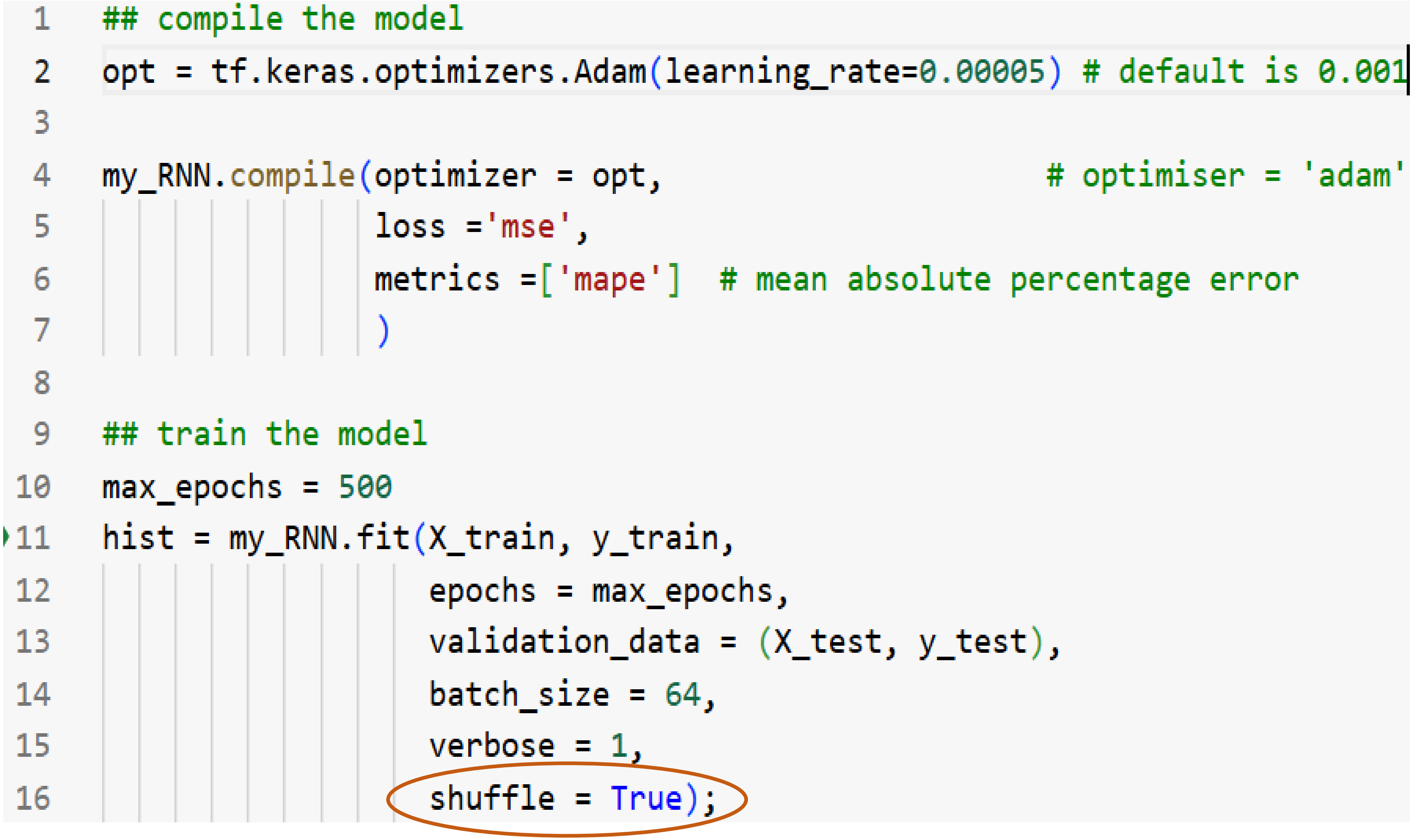
dropout_2 (Dropout)

Input shape: **(None, 512)** | Output shape: **(None, 512)**

dense (Dense)

Input shape: **(None, 512)** | Output shape: **(None, 1)**

Configure (compile) and train the model

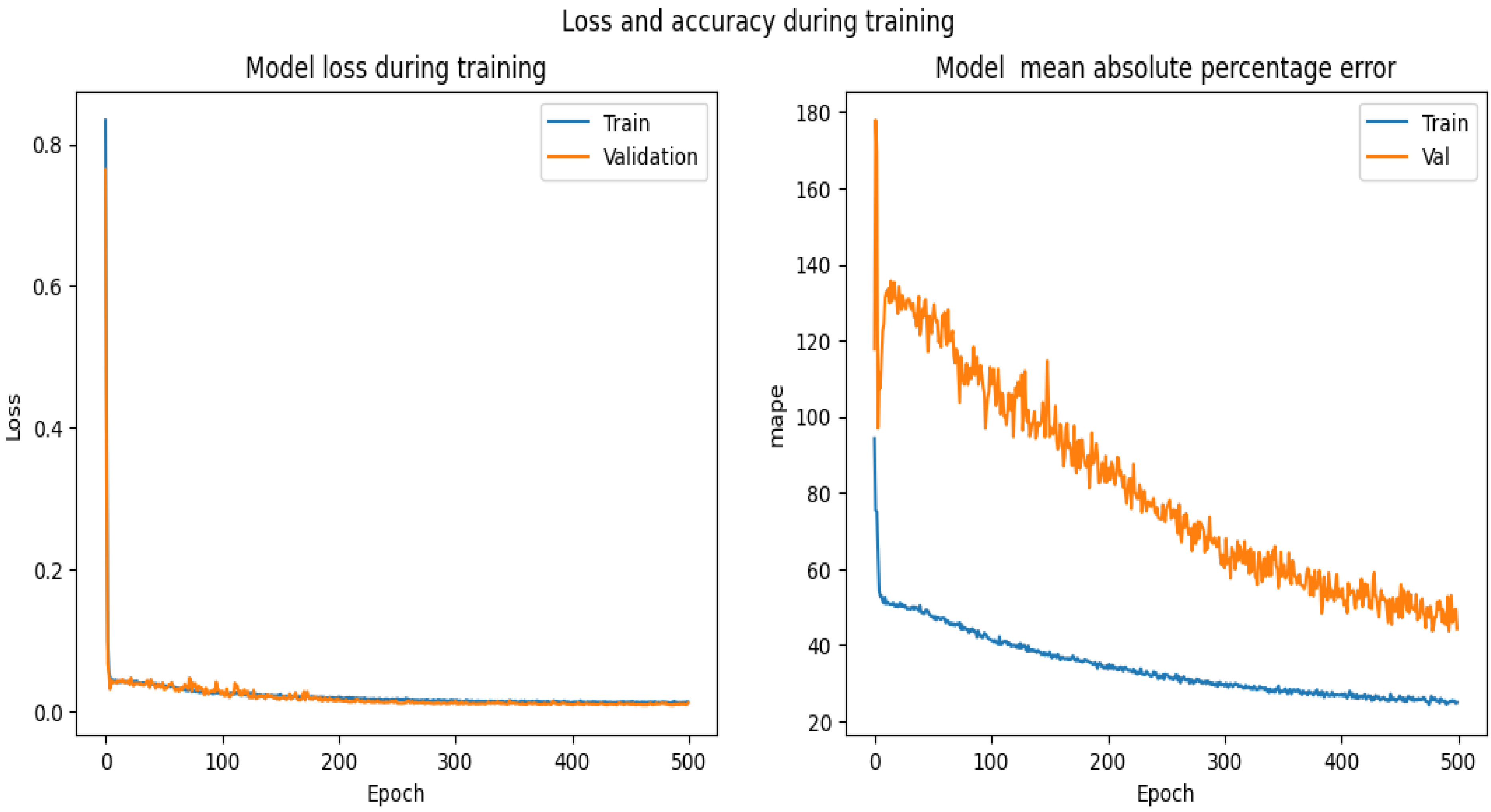


```
## compile the model
opt = tf.keras.optimizers.Adam(learning_rate=0.00005) # default is 0.001

my_RNN.compile(optimizer = opt,                     # optimiser = 'adam'
               loss ='mse',
               metrics =['mape']  # mean absolute percentage error
               )

## train the model
max_epochs = 500
hist = my_RNN.fit(X_train, y_train,
                  epochs = max_epochs,
                  validation_data = (X_test, y_test),
                  batch_size = 64,
                  verbose = 1,
                  shuffle = True);
```

shuffle

- **Data Order:** During training, your model sees your training data in batches. By default (shuffle=True), model.fit() will randomly shuffle the order of your training data before each epoch (a full pass through the training data).
- **Why it's important:** Shuffling helps prevent your model from learning patterns that are specific to the order of your data. This can lead to better generalization and performance on unseen data.
- **When to set shuffle=False:** You might set shuffle=False in very specific situations, like when the order of your data is crucial (e.g., time-series data where the order represents a sequence of events) and you don't want it to be randomized.

Training evolution 500 epochs



```
15/15 - 0s - loss: 0.0111 - mape: 44.2792 - 85ms/epoch - 6ms/step
Accuracy in the test data:  44.279170989990234
```

Elapsed time: 4696.791035413742 seconds = **78.30 min** = **1h 18 min**   **no GPU**

Elapsed time: 279.7024142742157 seconds = **4.66 min T4 GPU**   **16.8x**

MSE vs MAPE

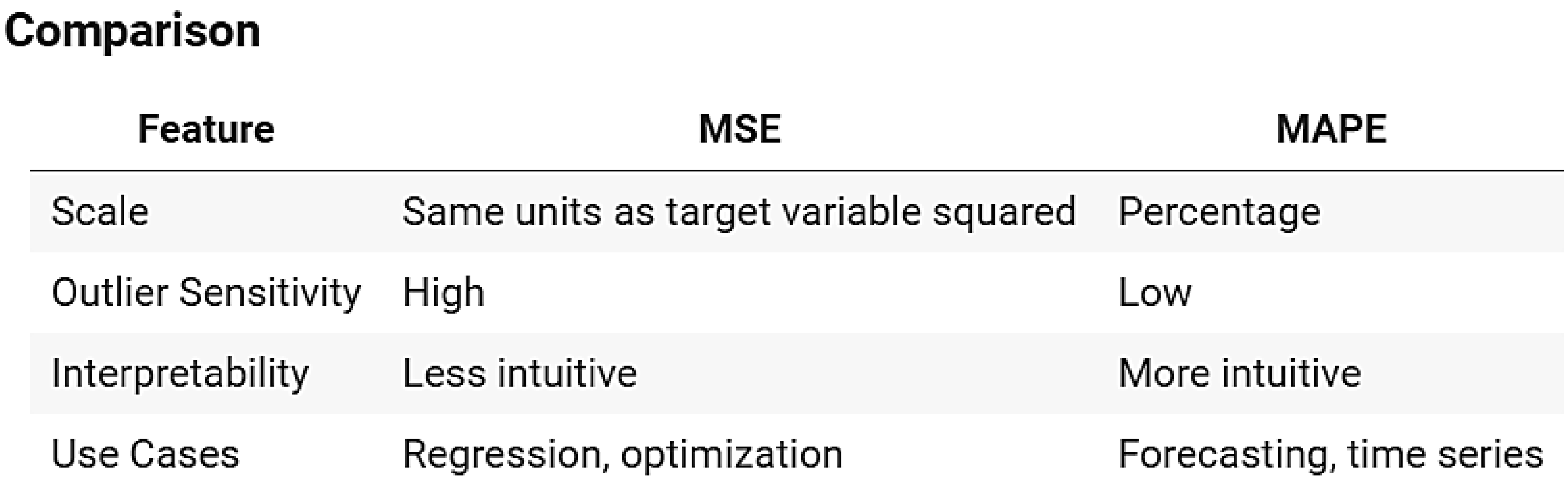
MSE (Mean Squared Error)

- **Definition:** MSE measures the average squared difference between the predicted and actual values.
- **Formula:** $MSE = (1/m) * \Sigma(\text{actual} - \text{predicted})^2$
- **Characteristics:**
    - It gives **higher weight to larger errors** due to the squaring.
    - It is sensitive to outliers.
    - It is in the same units as the target variable squared.
- **Usefulness:** MSE is widely used and is differentiable, which is important for **optimization algorithms.**

MAPE (Mean Absolute Percentage Error)

- **Definition:** MAPE measures the average absolute percentage difference between the predicted and actual values.
- **Formula:** $MAPE = (1/m) * \Sigma(|\text{actual} - \text{predicted}| / |\text{actual}|) * 100$
- **Characteristics:**
    - It is **expressed as a percentage, making it easy to interpret**.
    - It is less sensitive to outliers compared to MSE.
    - It is not defined when actual values are zero.
- **Usefulness:** MAPE is useful when the relative error is more important than the absolute error. It is often **used in forecasting and time series analysis**.

MSE vs MAPE



Comparison

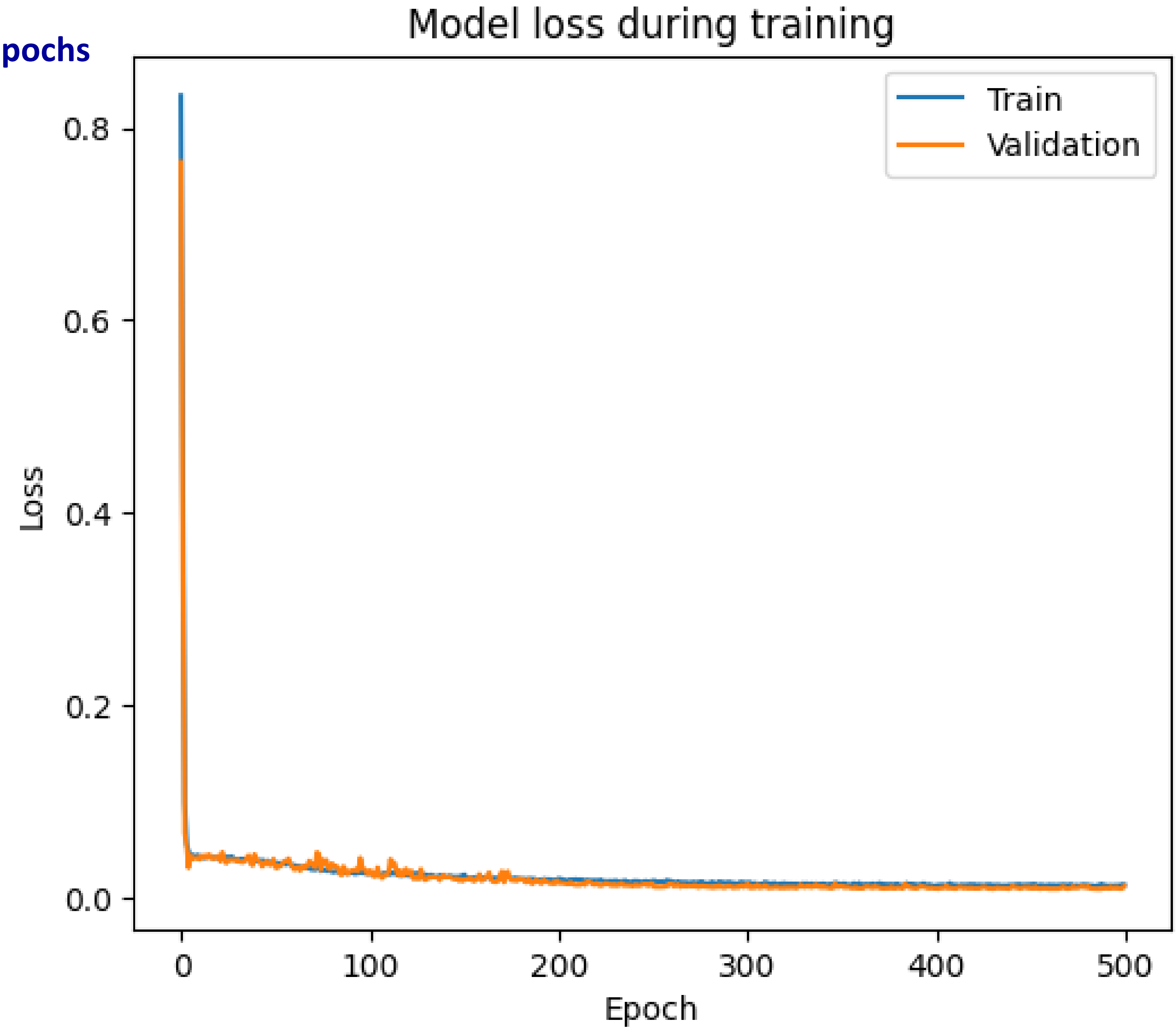
Feature	MSE	MAPE
Scale	Same units as target variable squared	Percentage
Outlier Sensitivity	High	Low
Interpretability	Less intuitive	More intuitive
Use Cases	Regression, optimization	Forecasting, time series

Choosing between MSE and MAPE

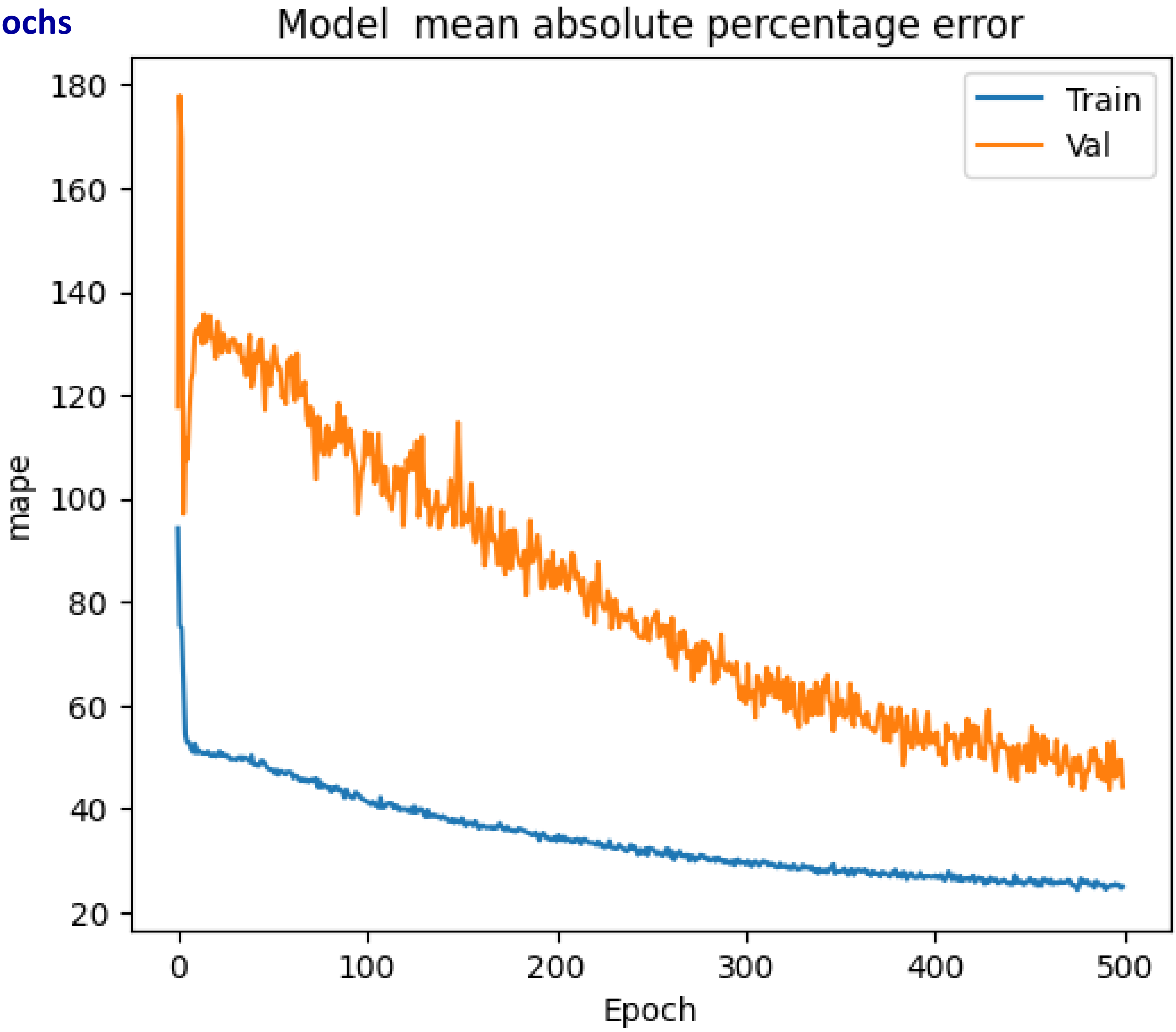
The choice between MSE and MAPE depends on your specific needs and the nature of your data.

- ➢ If you want to penalize larger errors more and your data has no zero values, MSE might be a good choice.
- ➢ If you prefer a more interpretable metric that is less sensitive to outliers, MAPE might be more suitable.

500 epochs

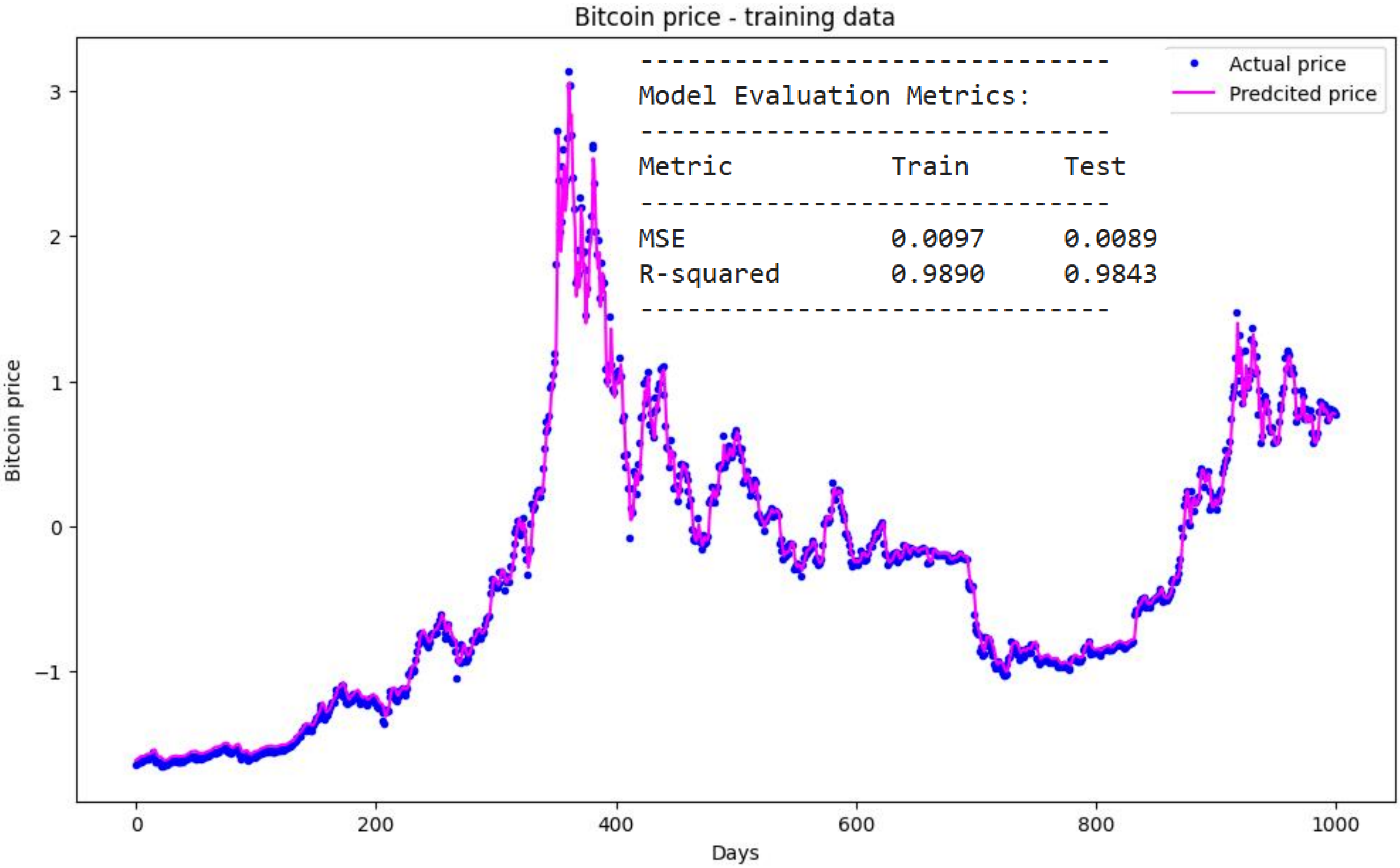

Model loss during training
Train
Validation
Loss
0.8
0.6
0.4
0.2
0.0
Epoch
0
100
200
300
400
500

500 epochs


Model mean absolute percentage error
mape
180
160
140
120
100
80
60
40
20
Epoch
0
100
200
300
400
500
Train
Val

Prediction

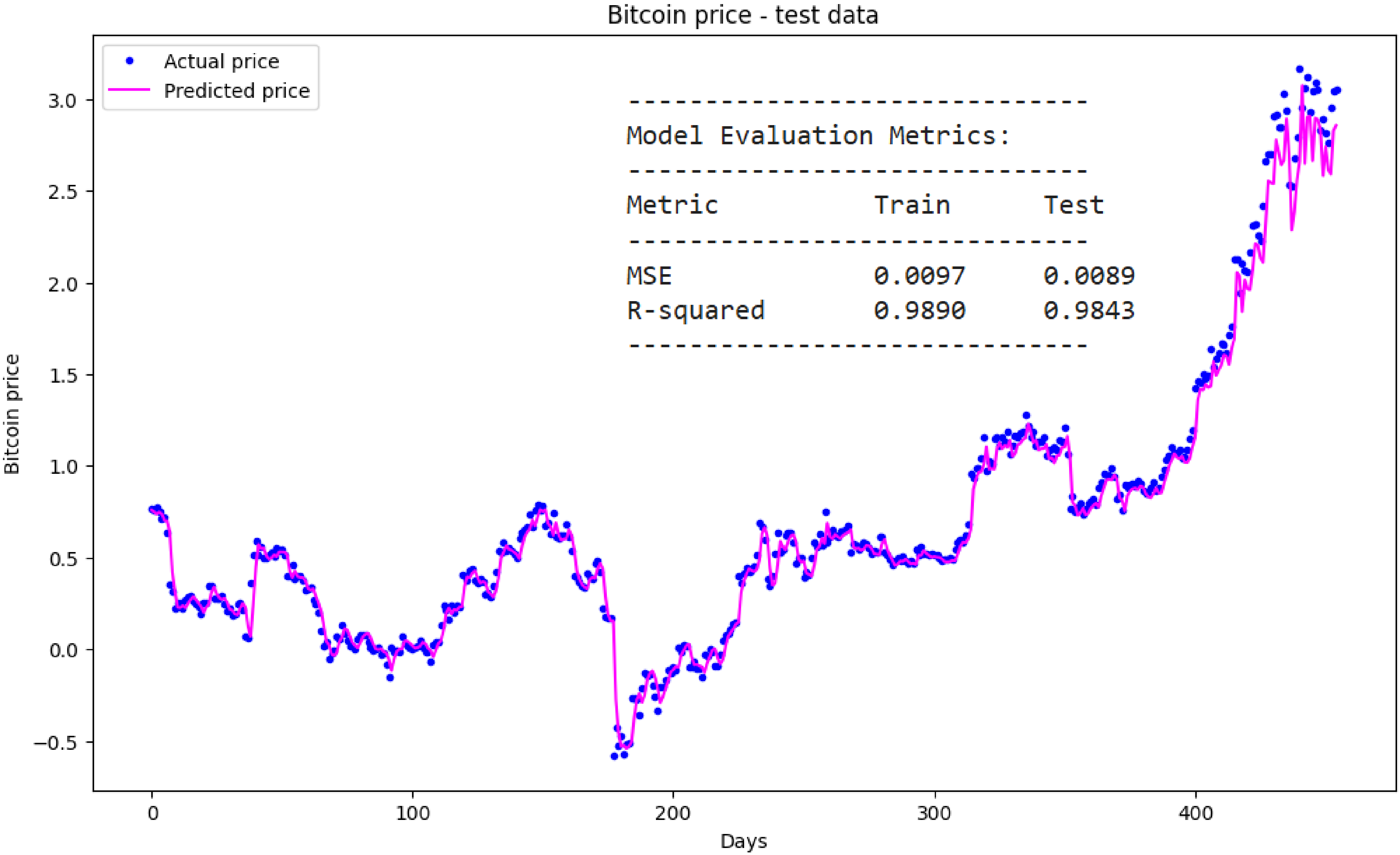
500 epochs



```
--------------------------------
Model Evaluation Metrics:
--------------------------------
Metric              Train      Test
--------------------------------
MSE                 0.0097     0.0089
R-squared           0.9890     0.9843
--------------------------------
```

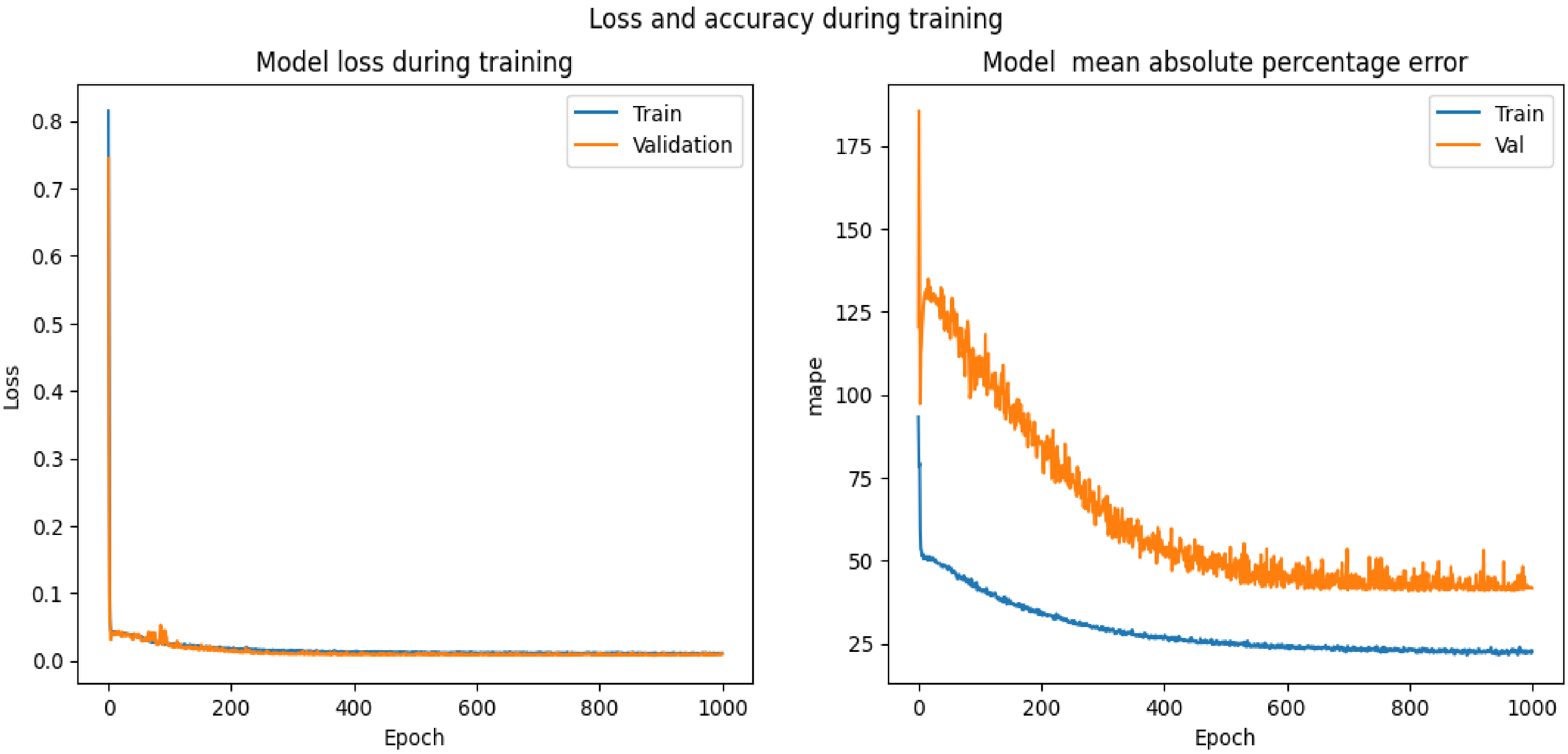
Prediction

500 epochs



```
--------------------------------
Model Evaluation Metrics:
--------------------------------
Metric              Train       Test
--------------------------------
MSE                 0.0097      0.0089
R-squared           0.9890      0.9843
--------------------------------
```

Train longer, 1000 epochs

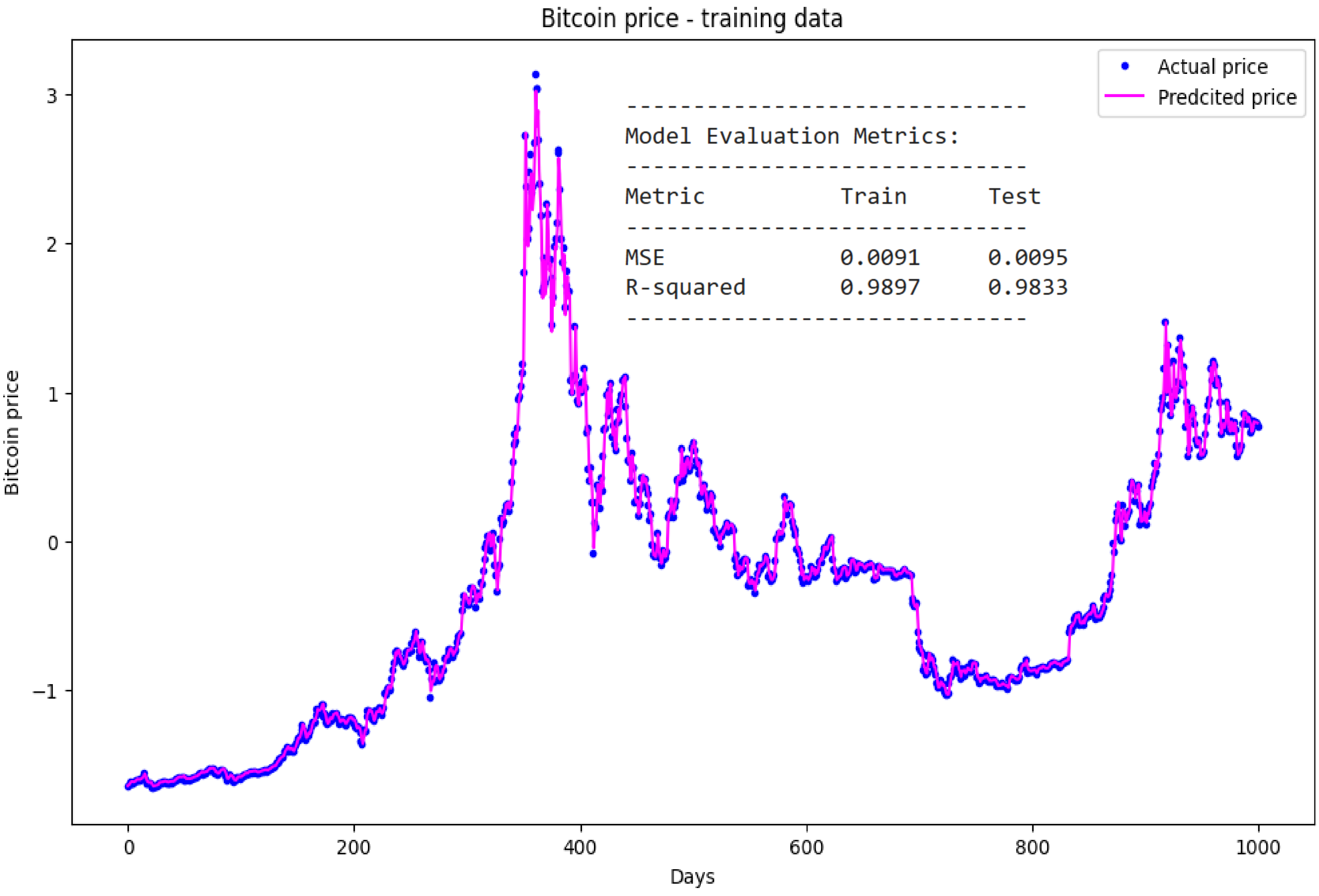


```
15/15 - 0s - 6ms/step - loss: 0.0095 - mape: 41.6279
Accuracy in the test data: 41.62788391113281
```

Elapsed time: 581.6502554416656 seconds = **9.67 min T4 GPU**

Prediction

1000 epochs

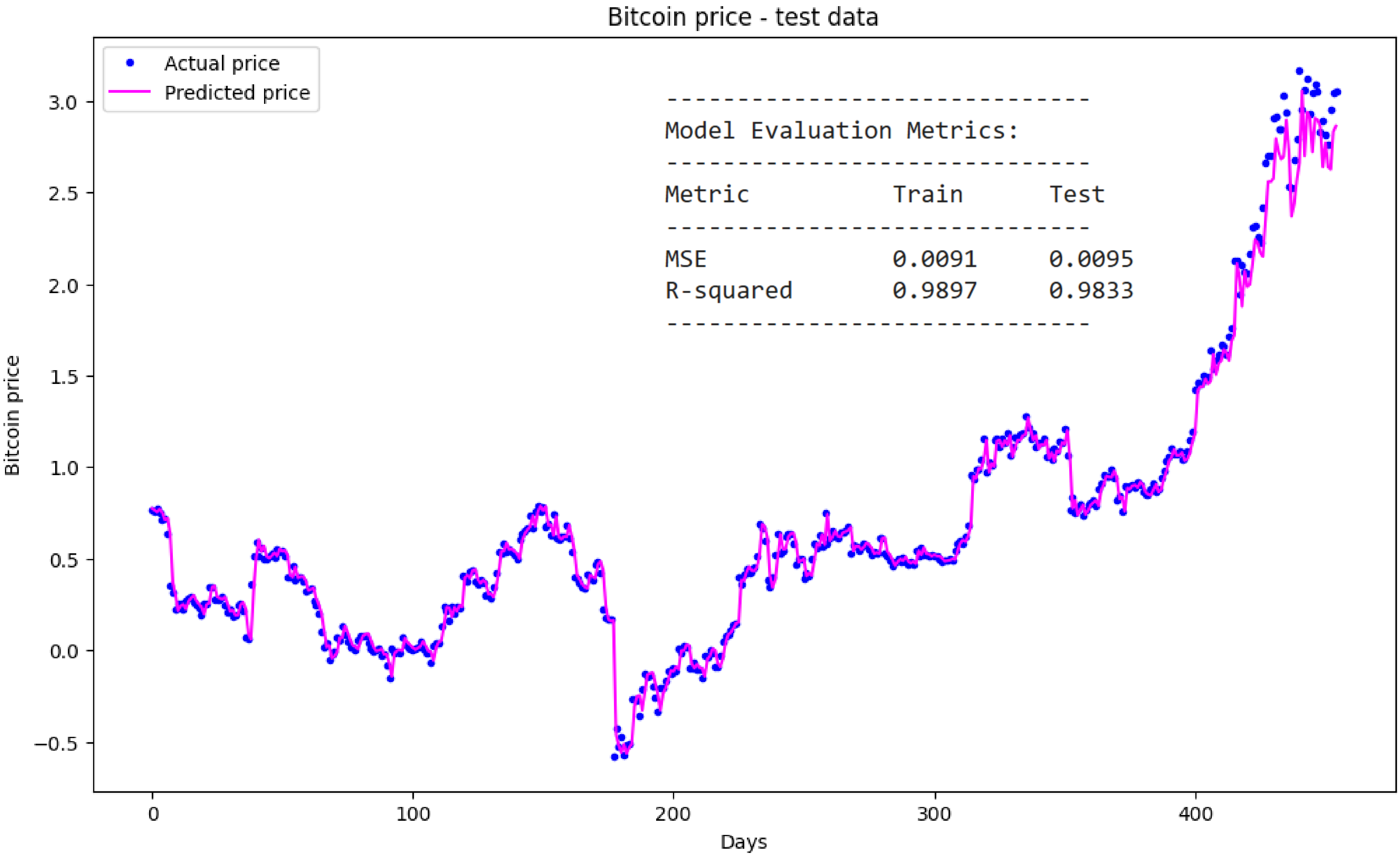


Bitcoin price - training data

```
--------------------------------
Model Evaluation Metrics:
--------------------------------
Metric           Train      Test
--------------------------------
MSE              0.0091     0.0095
R-squared        0.9897     0.9833
--------------------------------
```

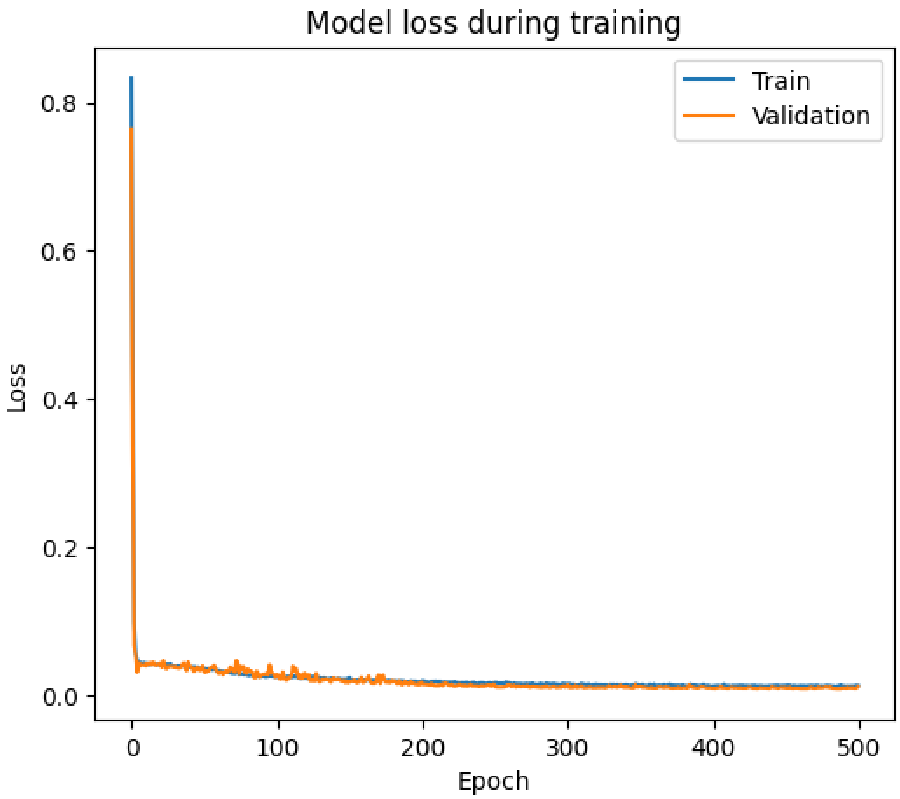
Prediction

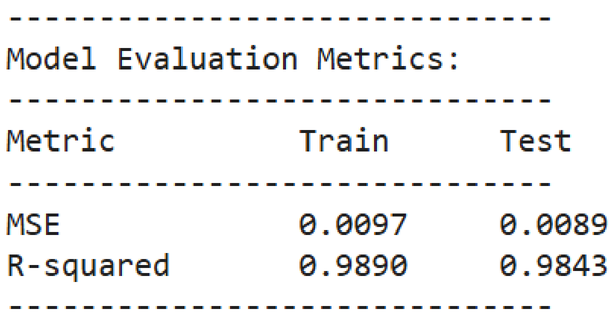
1000 epochs



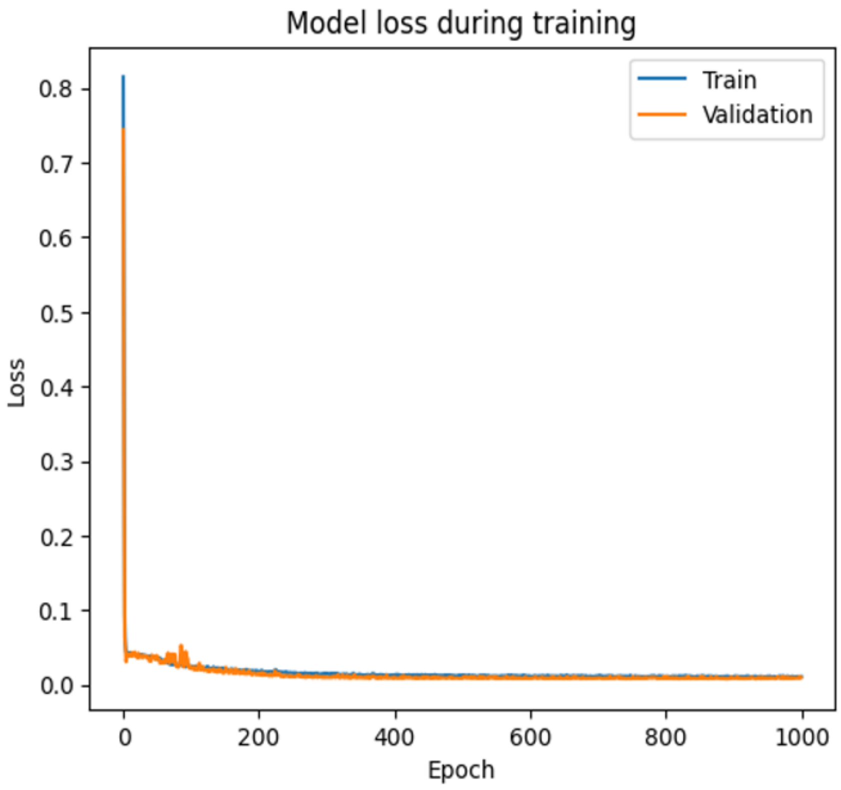
Bitcoin price - test data

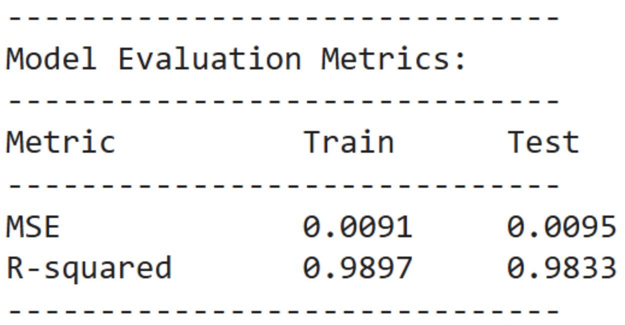
```
--------------------------------
Model Evaluation Metrics:
--------------------------------
Metric           Train      Test
--------------------------------
MSE              0.0091     0.0095
R-squared        0.9897     0.9833
--------------------------------
```



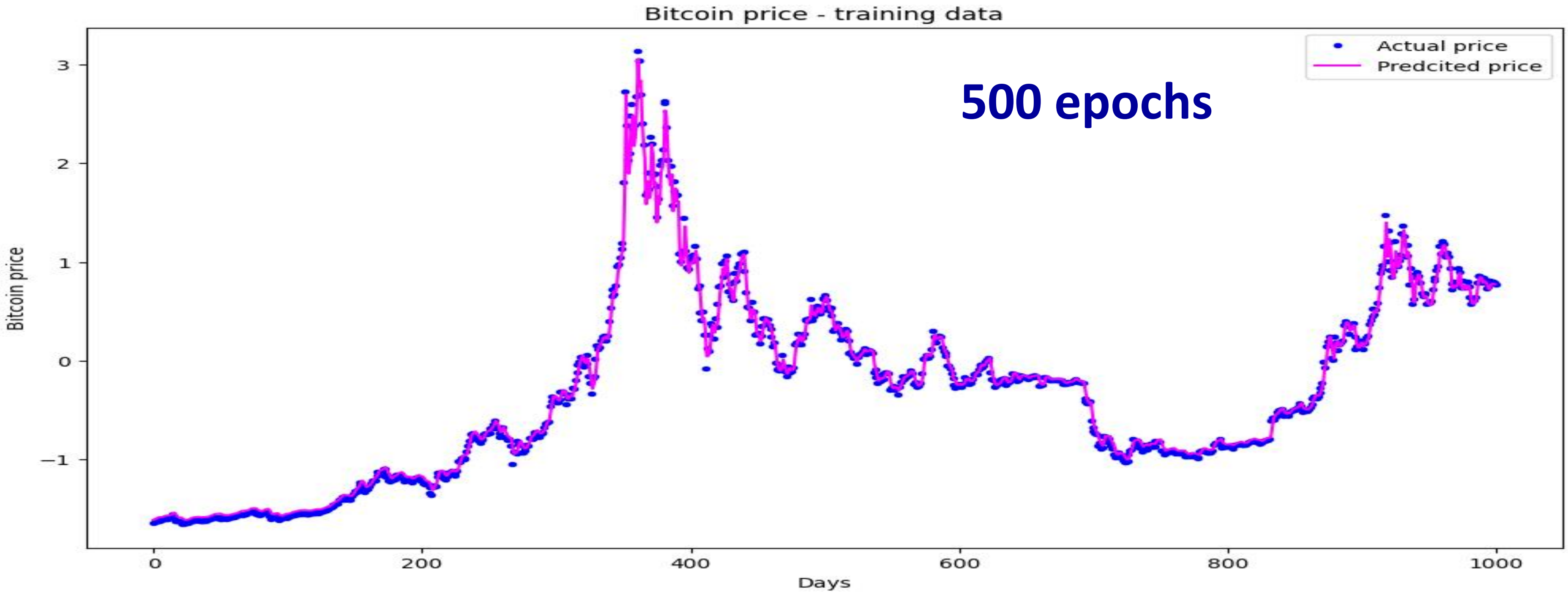


```
------------------------------
Model Evaluation Metrics:
------------------------------
Metric          Train     Test
------------------------------
MSE             0.0097    0.0089
R-squared       0.9890    0.9843
------------------------------
```

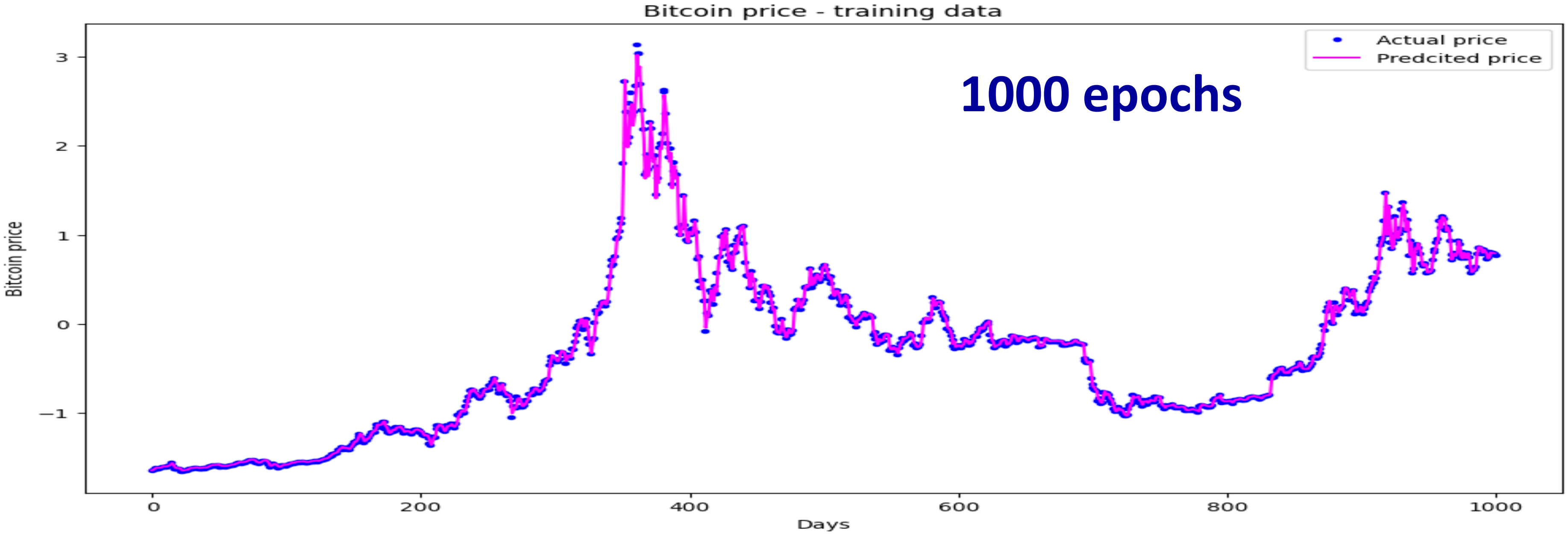




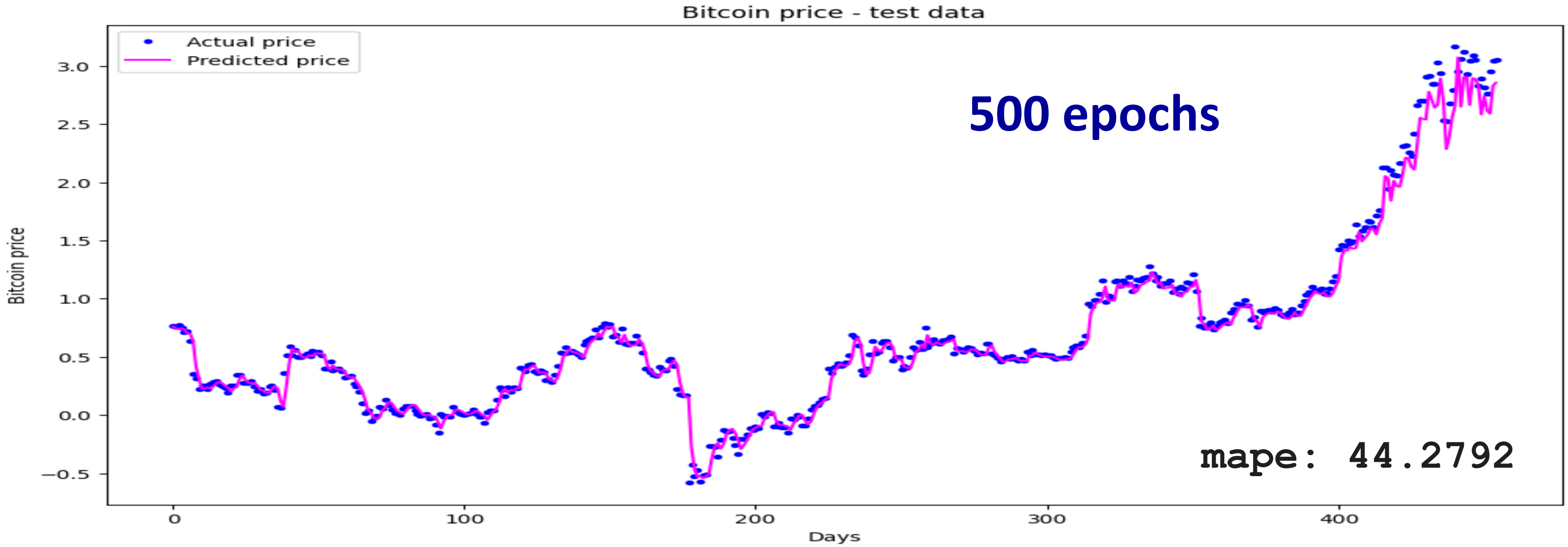
```
------------------------------
Model Evaluation Metrics:
------------------------------
Metric          Train     Test
------------------------------
MSE             0.0091    0.0095
R-squared       0.9897    0.9833
------------------------------
```



500 epochs

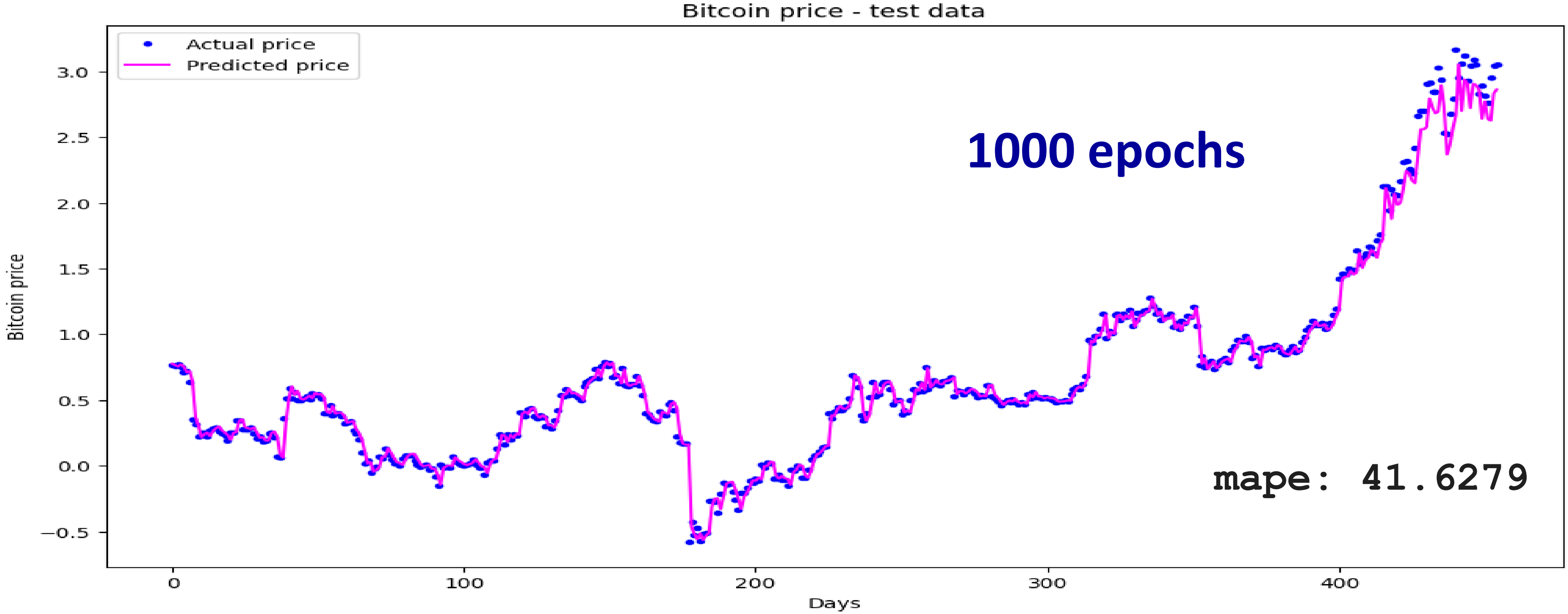


1000 epochs



500 epochs

mape: 44.2792



1000 epochs

mape: 41.6279

Using the Notebook file

This is a link to the application notebook:

https://colab.research.google.com/drive/1zqHQZYvbeQMRtAQCl9A_64cLBeoI92-A?usp=sharing