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.

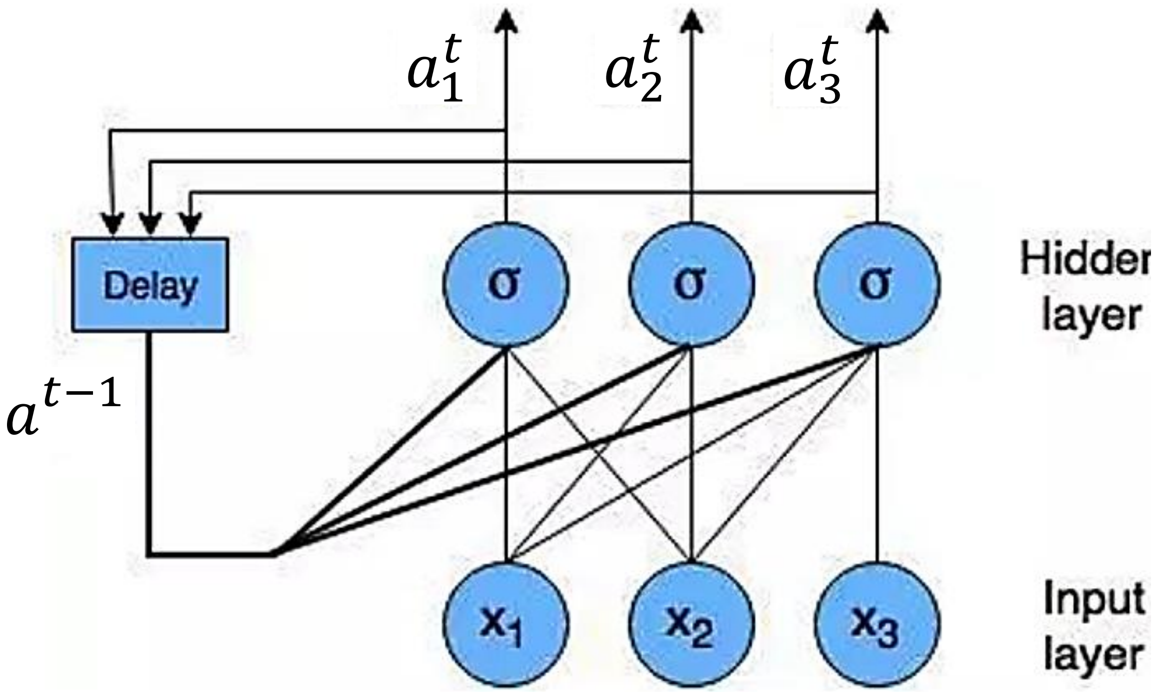
https://heartbeat.fritz.ai/a-beginners-guide-to-implementing-long-short-term-memory-networks-lstm-eb7a2ff09a27

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



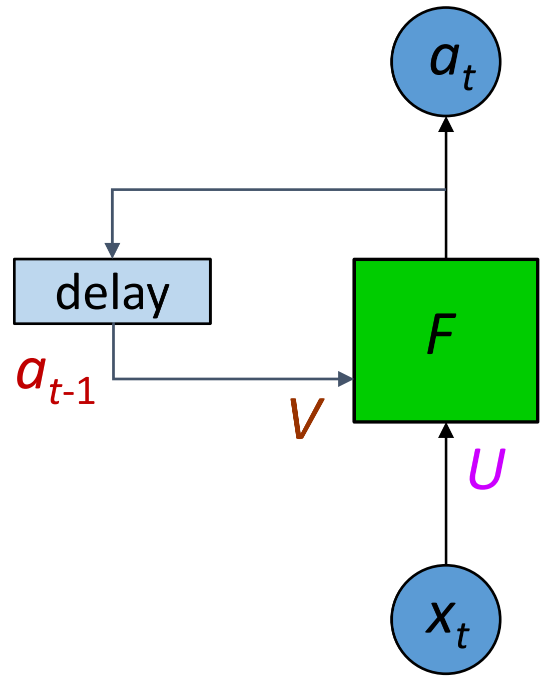
The output of the hidden layer is *fed back* into the same hidden layer

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

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 function

weight matrix for input

current input

weight matrix for recurrent output

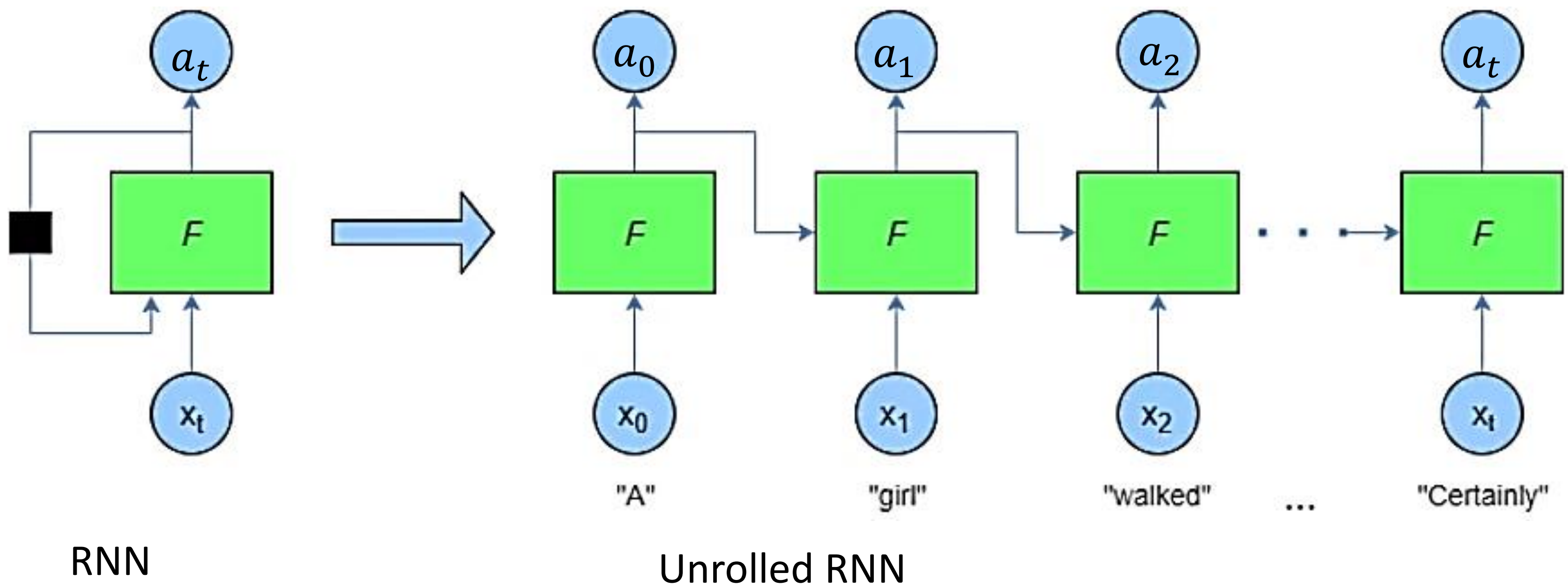
recurrent output

Example

"A girl walked into a bar, and she said: 'Can I have a drink please?'. The bartender said 'Certainly {?}"

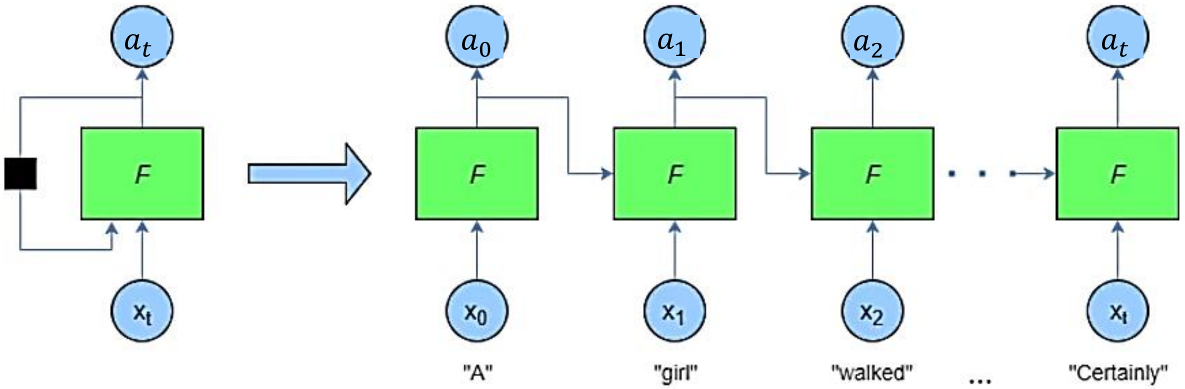
{?} can be "miss", "ma'am", …
"sir", "Mister", … also could fit

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.



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

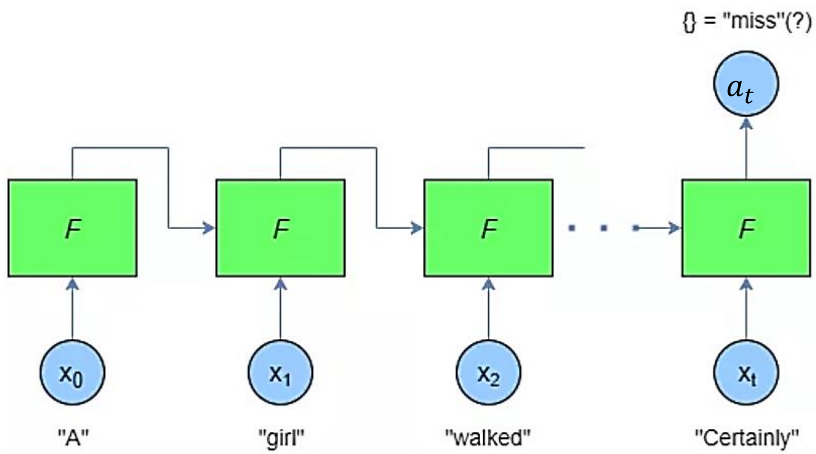
https://adventuresinmachinelearning.com/recurrent-neural-networks-lstm-tutorial-tensorflow/



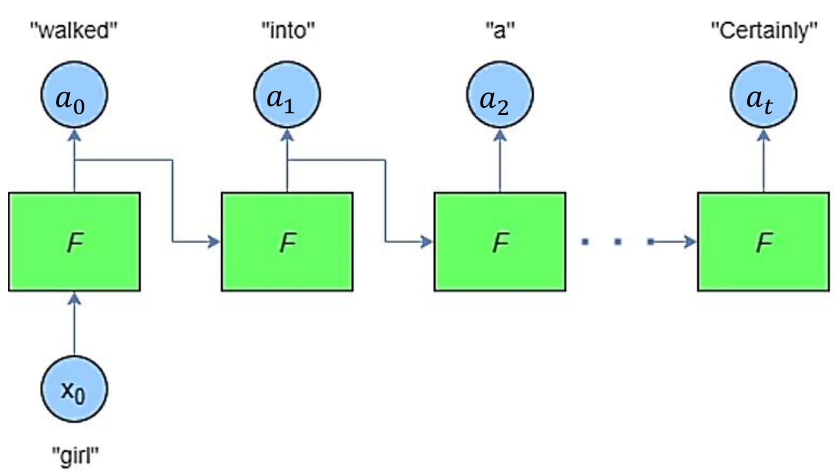
many-to-many model

inputs: “A girl walked into a bar…”

outputs (predicted): h_0 to h_t.



many-to-one model



one-to-many model

com/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**

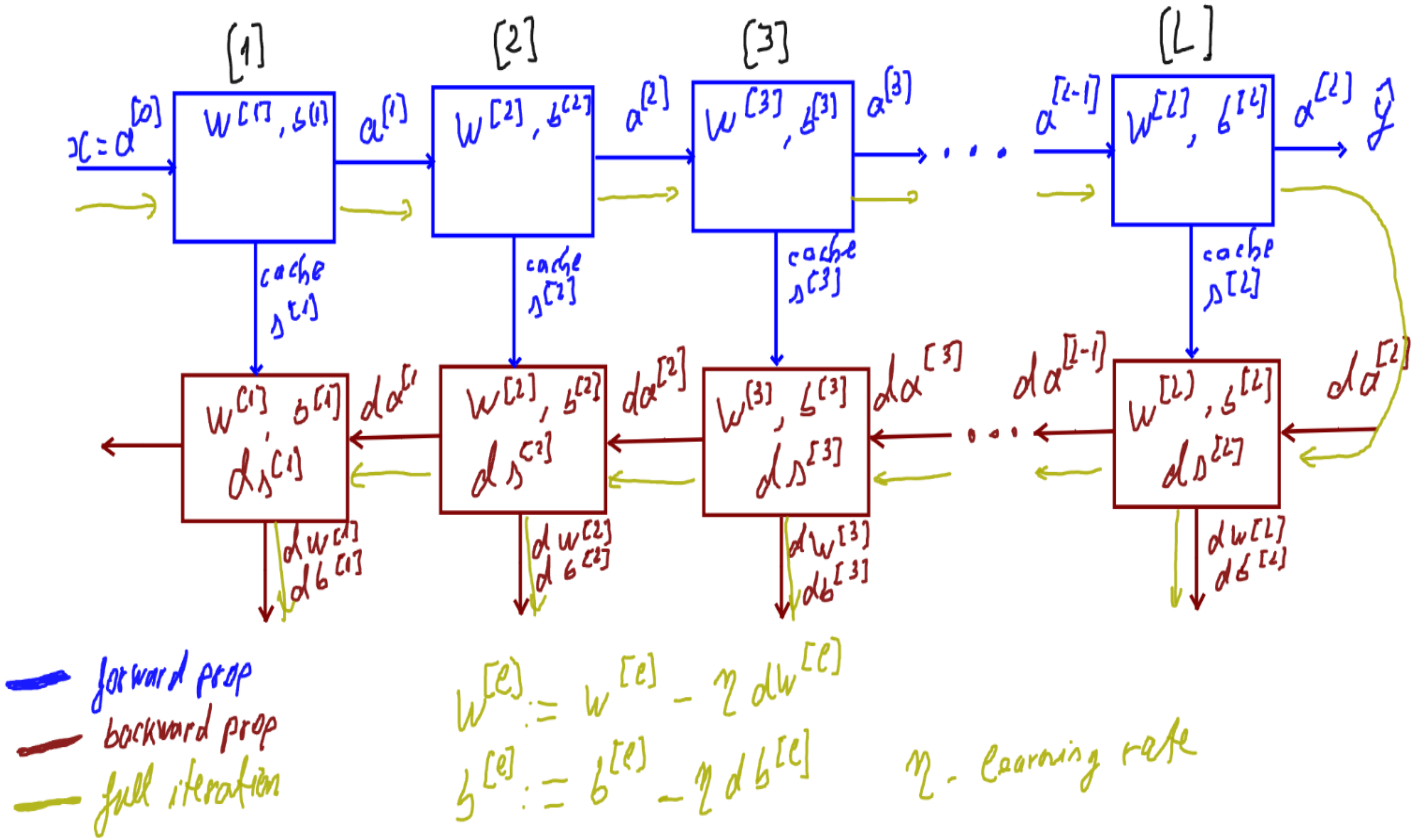
- **vanishing gradient / exploding gradient**

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
- **vanishing** down to nothing

Forward and backward propagation for a DNN



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.

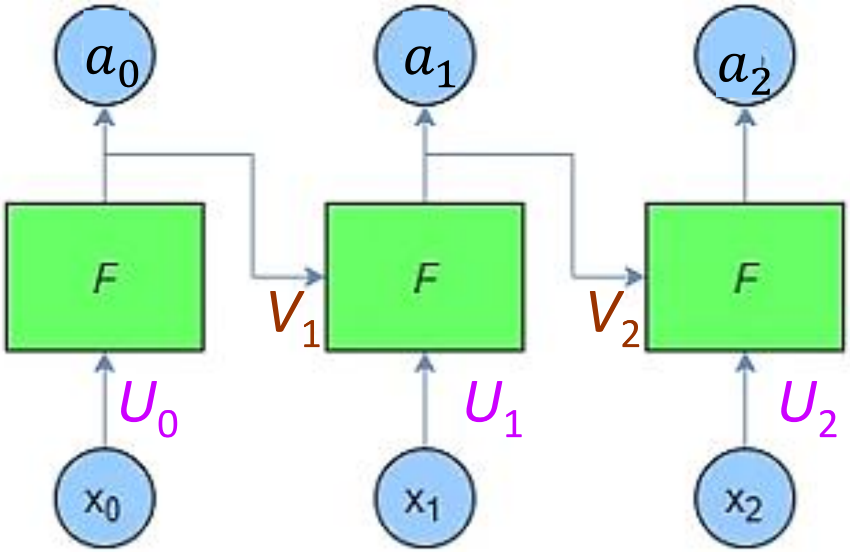
The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1.0.

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



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

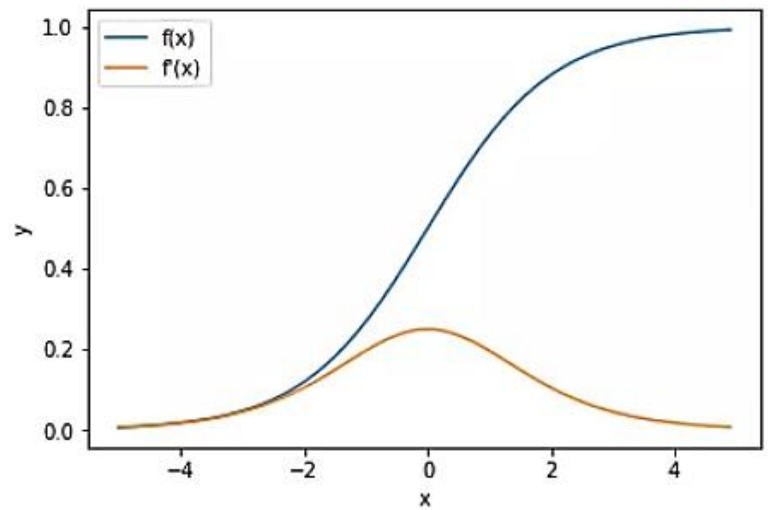
For back-propagation we compute the gradients of the activation function

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

- LSTM - 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 will help **reduce the gradient problem** sufficiently to make recurrent neural networks more useful for long-term memory tasks i.e. text sequence predictions.

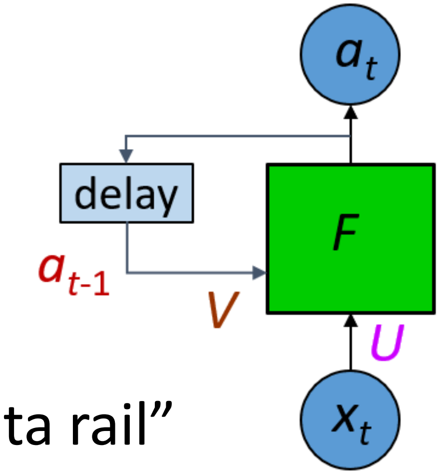
The way it does so is by creating **an internal memory state** which is simply *added* to the processed input, which greatly reduces the multiplicative effect of small gradients.

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.**

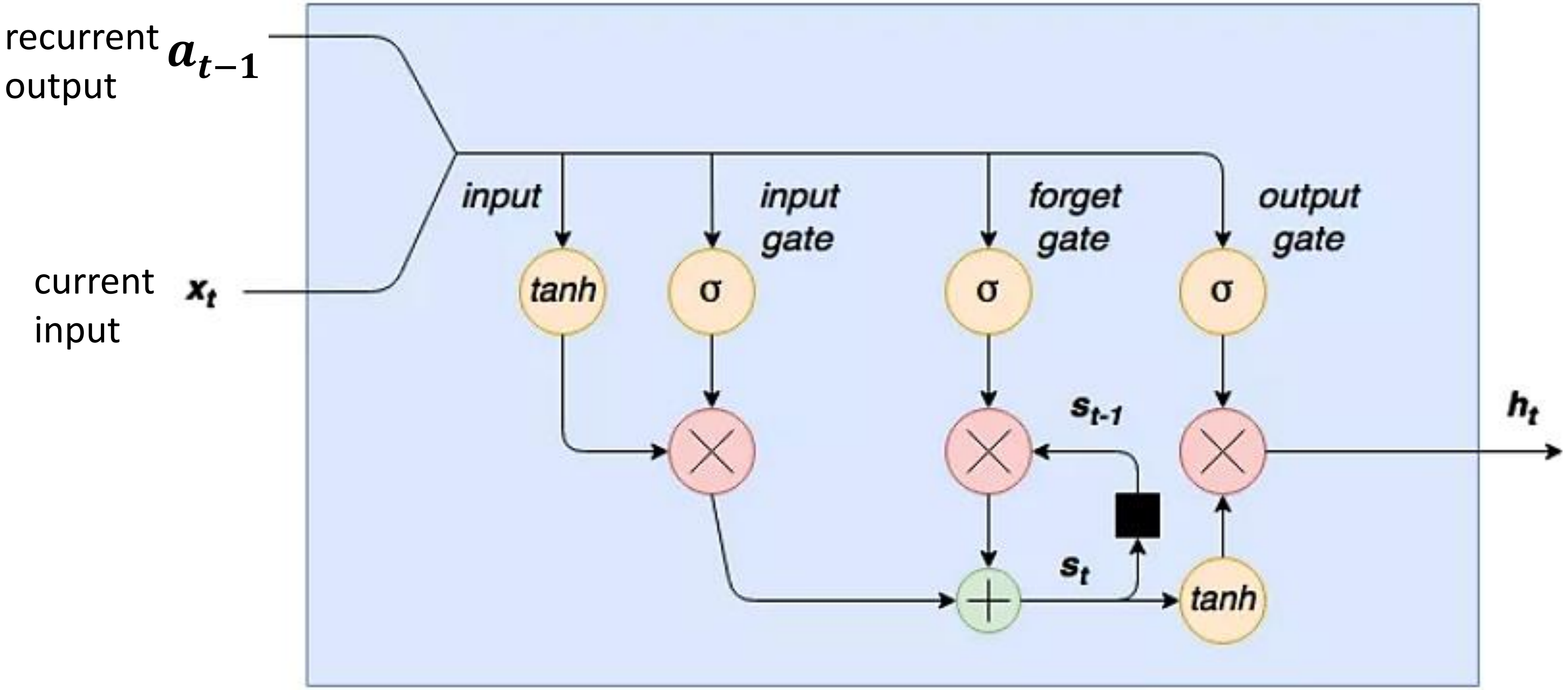
Two other gates, the ***input gate*** and ***output gate***, are also featured in LSTM cells.

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

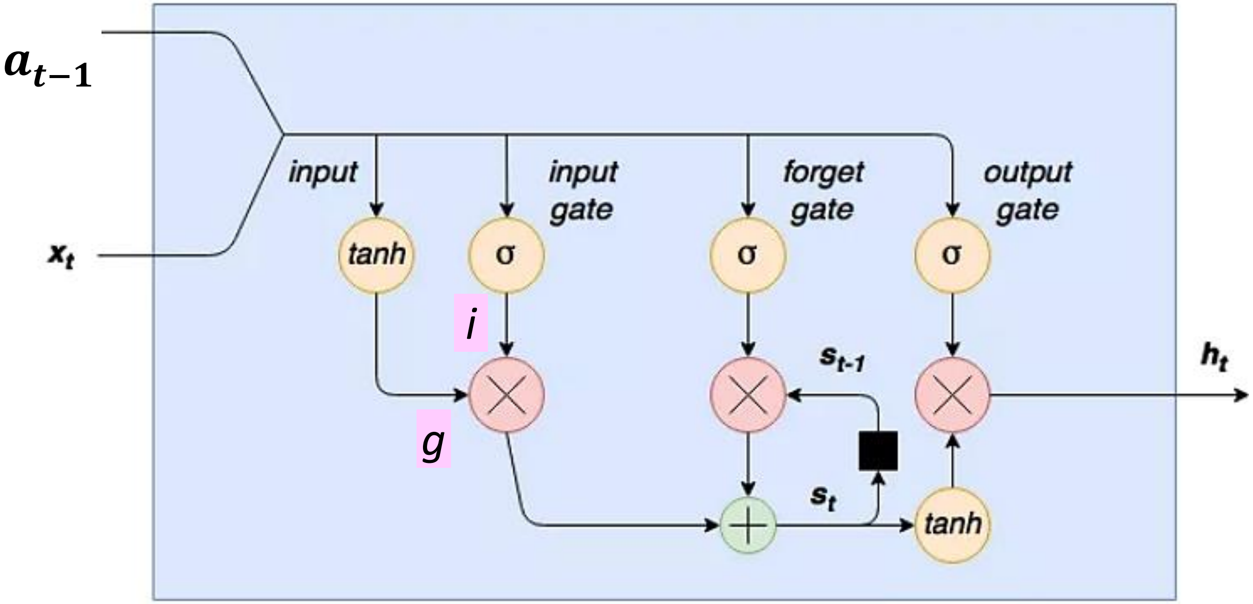
LSTM cell structure

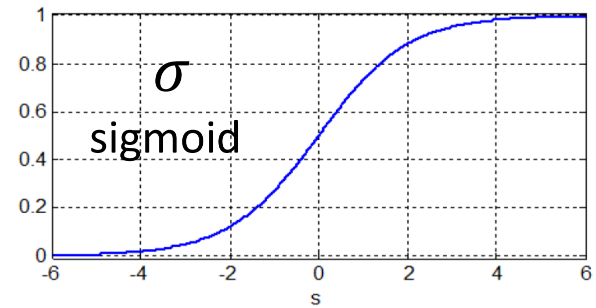


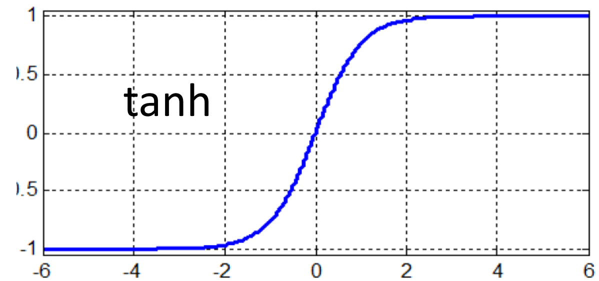
x_t and a_{t-1} concatenated together enters the top "data rail"



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







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

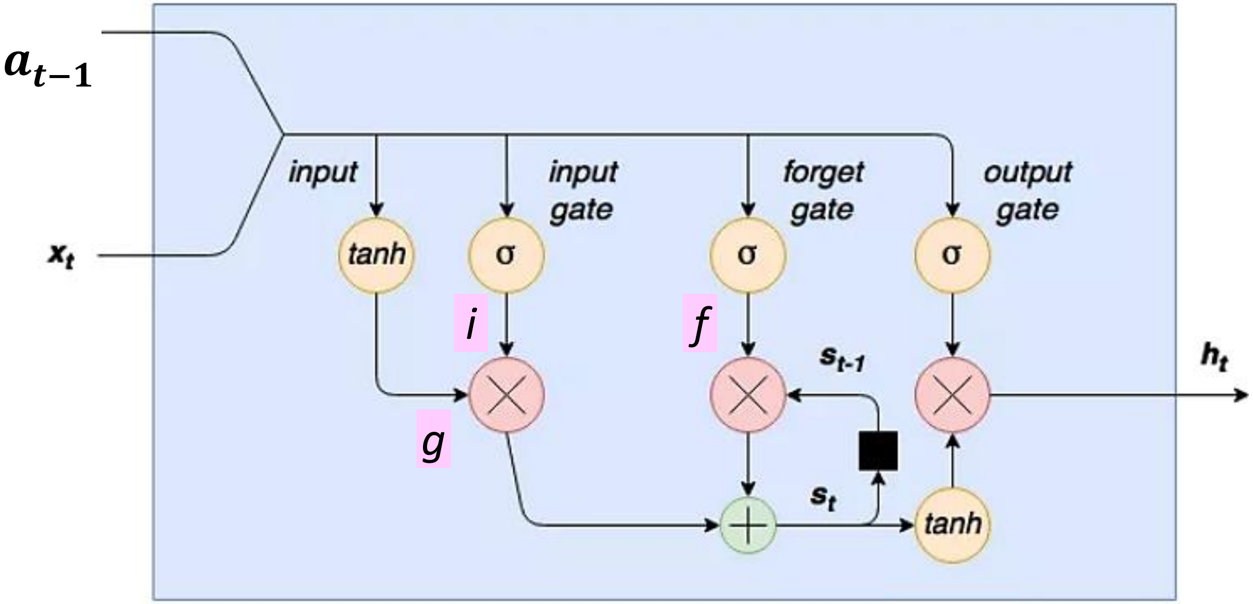
$$i = \sigma(b^i + x_t U^i + a_{t-1} V^i)$$

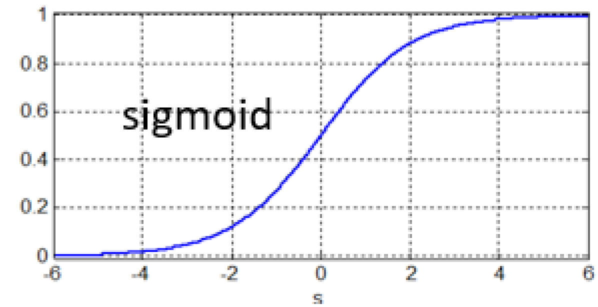
U - weight matrix for input
V - weight matrix for recurrent output

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

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





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)
- forgotten (i.e. forget gate output close to 0).

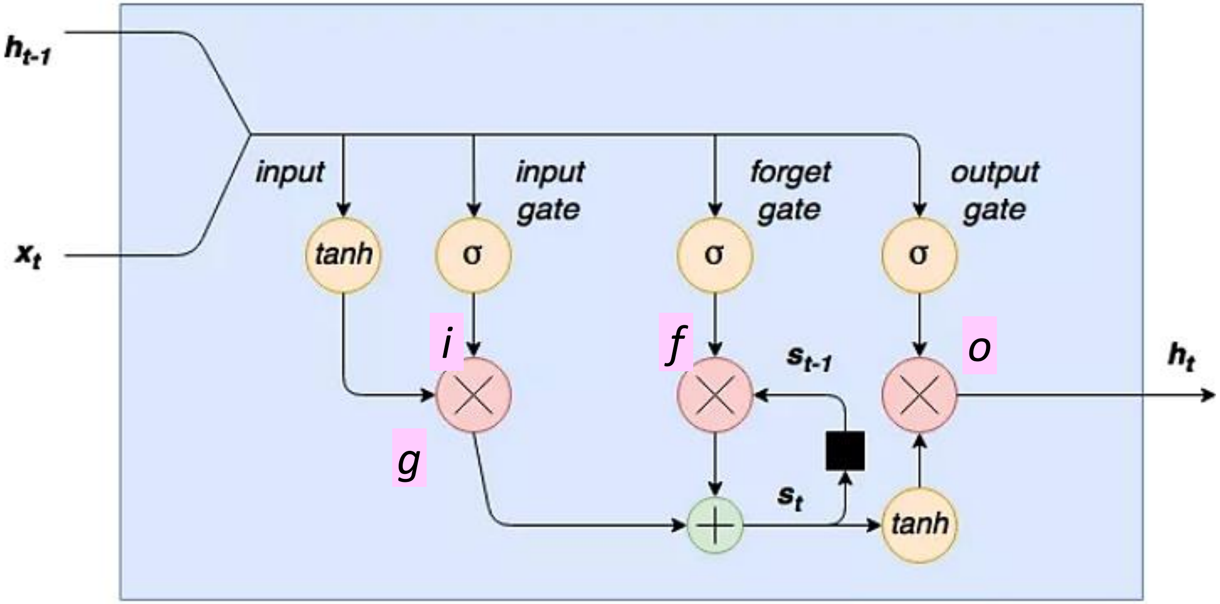
self-reccurent

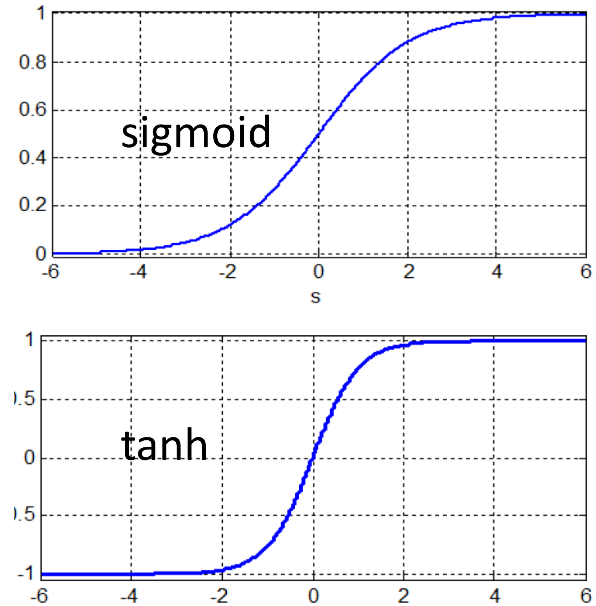
$$f = \sigma(b^f + x_t U^f + a_{t-1} V^f) \qquad s_t = s_{t-1} \circ f + g \circ i$$

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.

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





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).

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

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

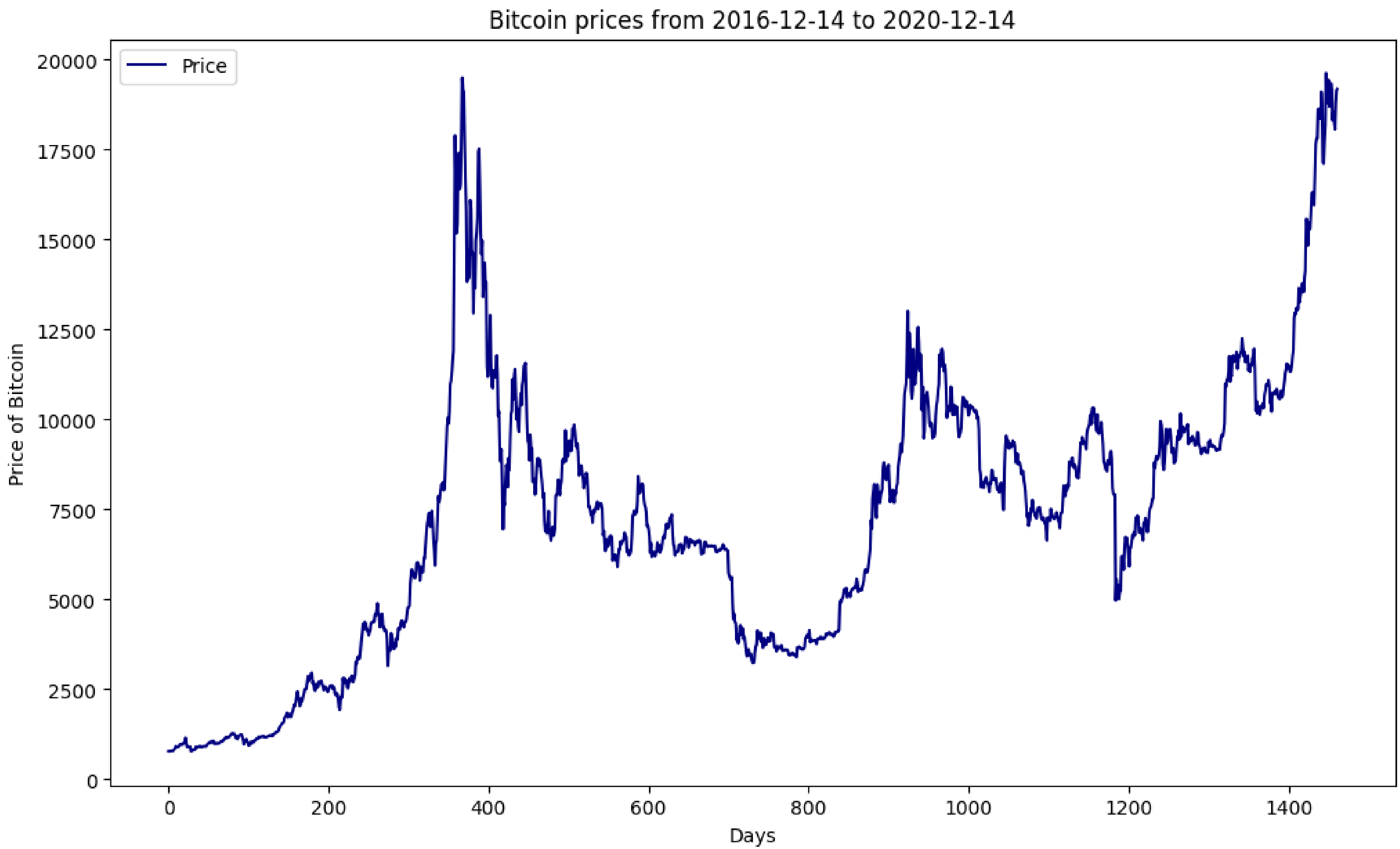
.csv

	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:

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

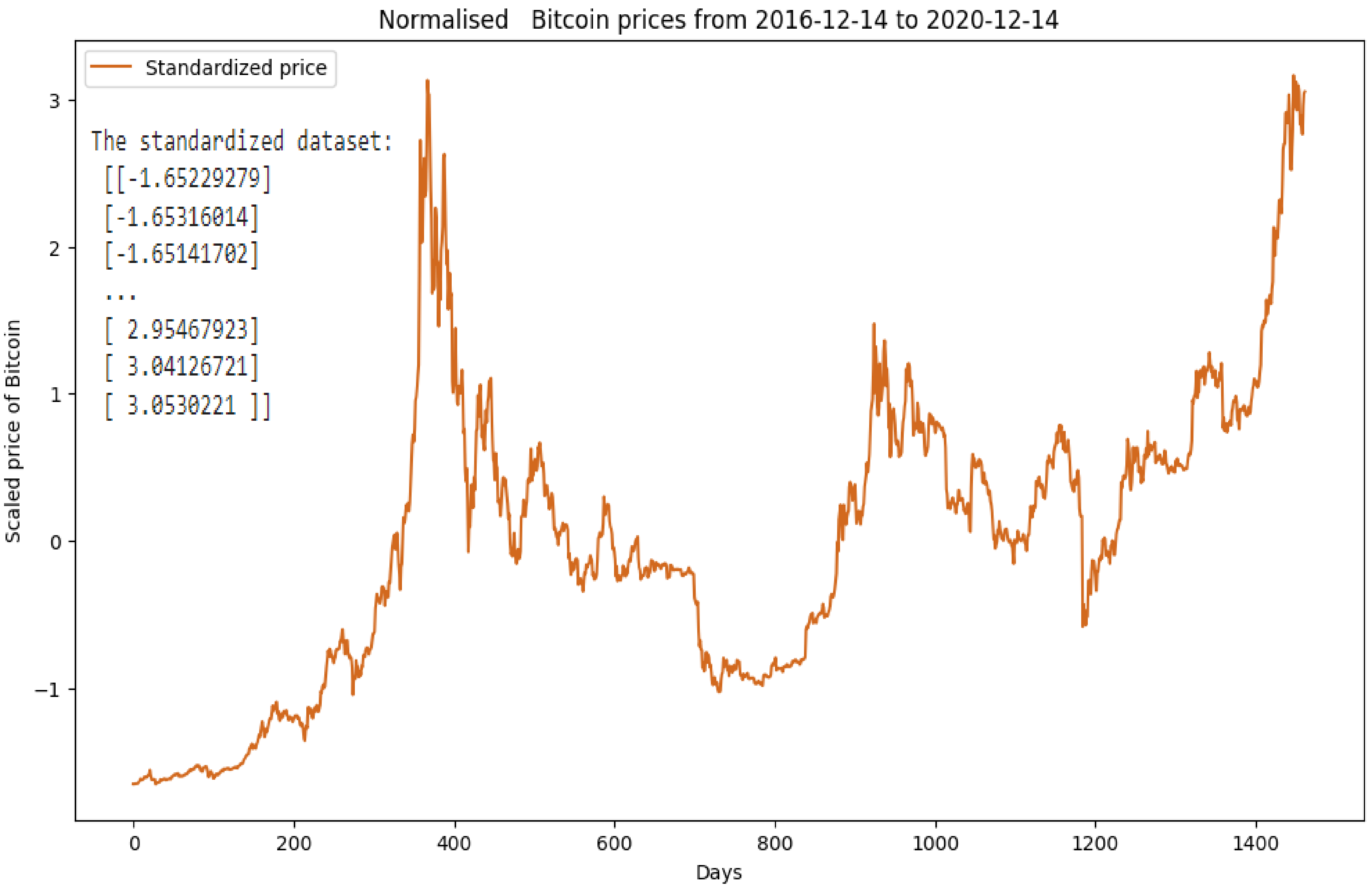
u is the mean of the training samples
s is the standard deviation of the training samples.

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

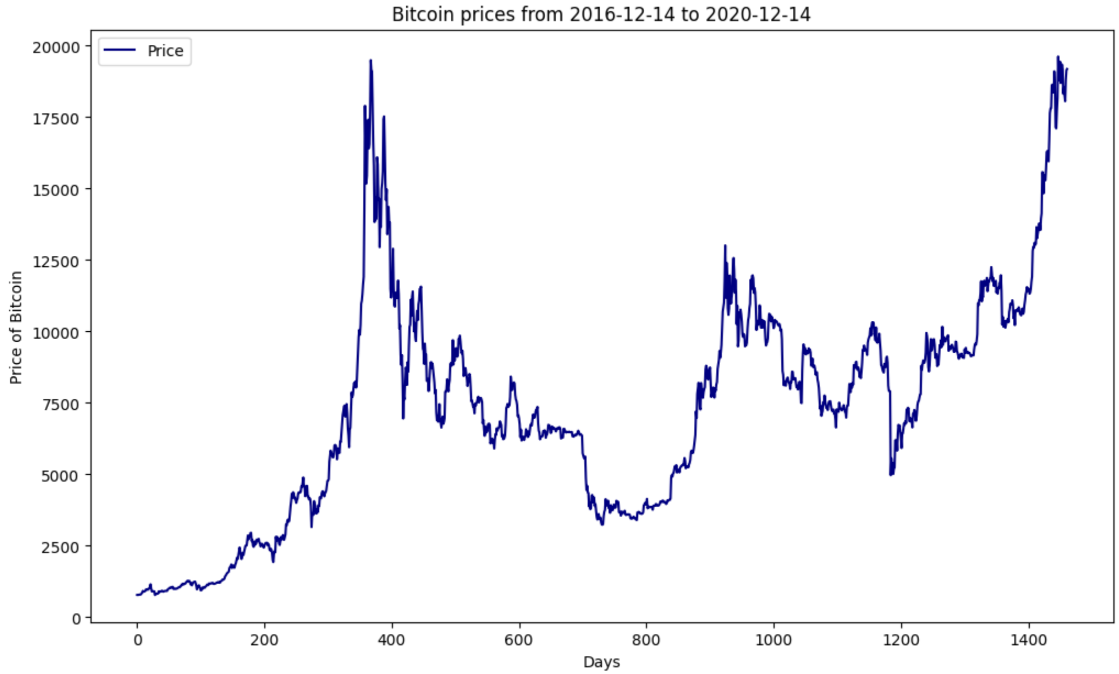
Mean and standard deviation are then stored to be used on later data using transform.

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).

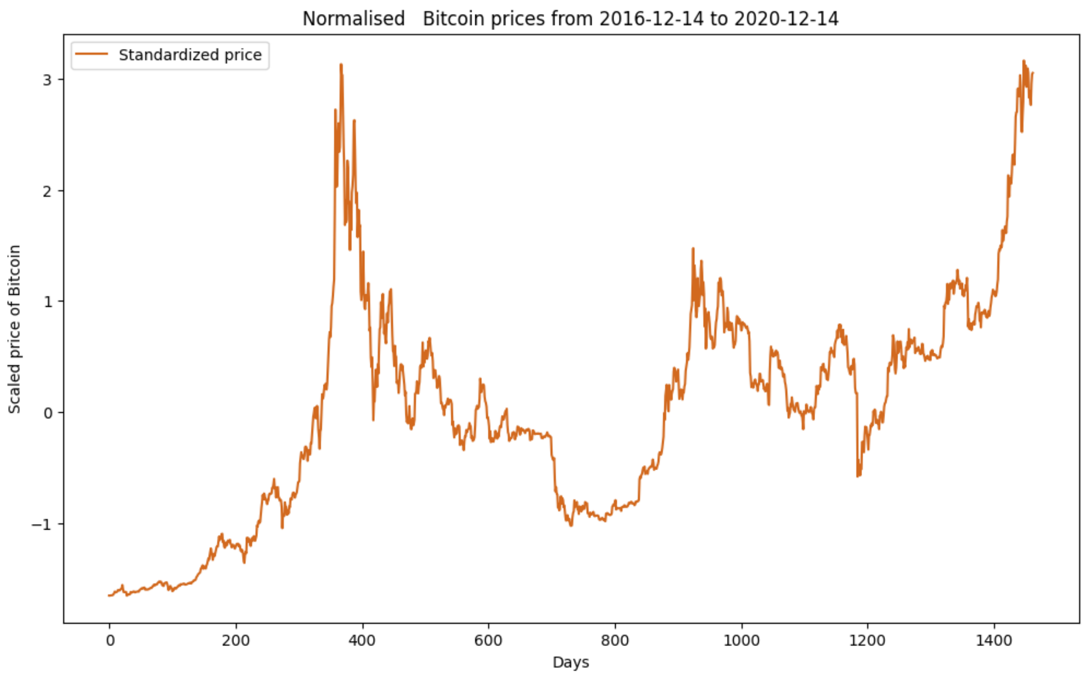
Standardized data



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



max: 19625.835938   min: 777.757019   mean: 7245.143068168262



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

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.

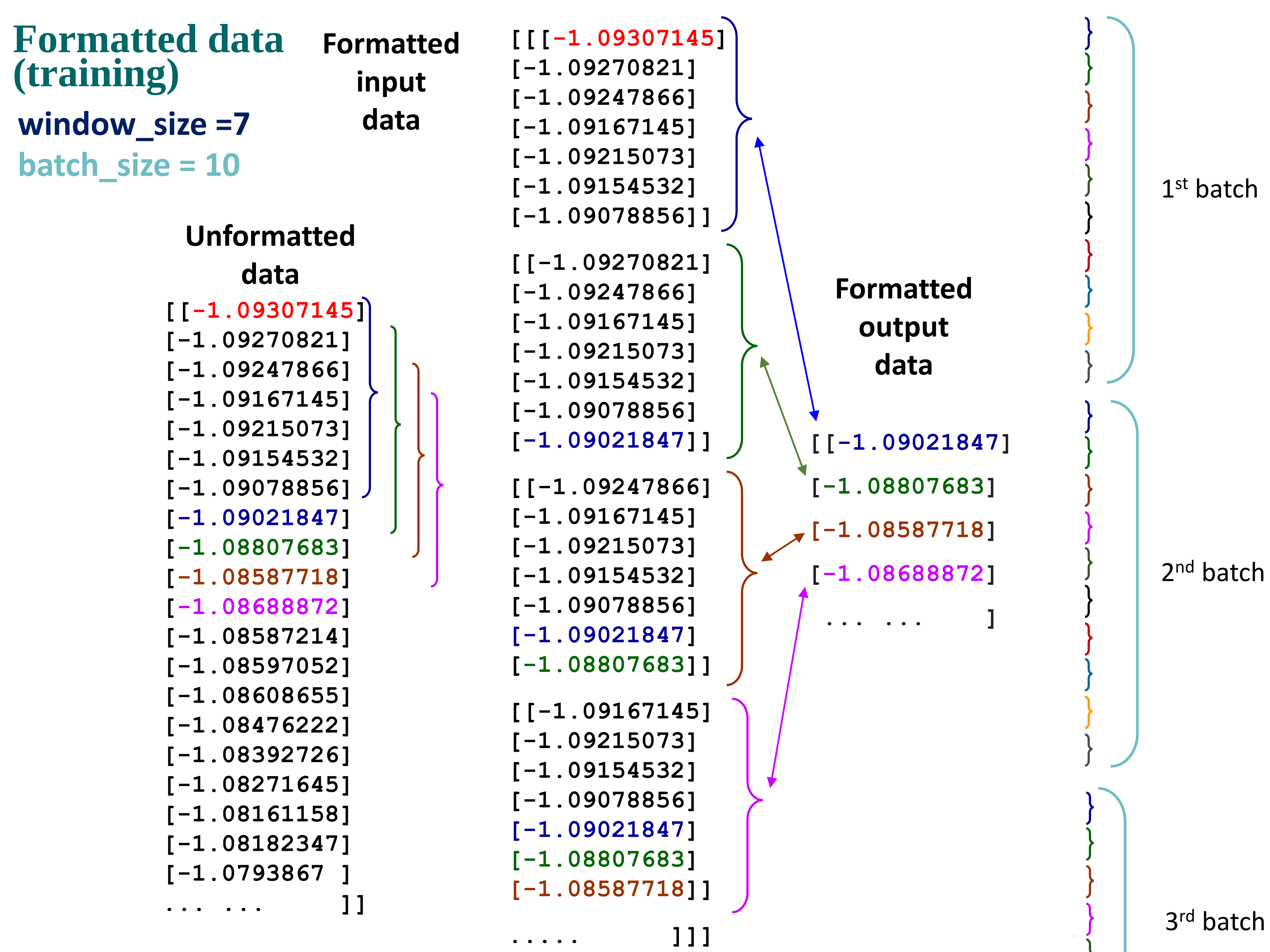
window_size = 7; The number of previous days we consider to predict the bitcoin price for our case.

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).

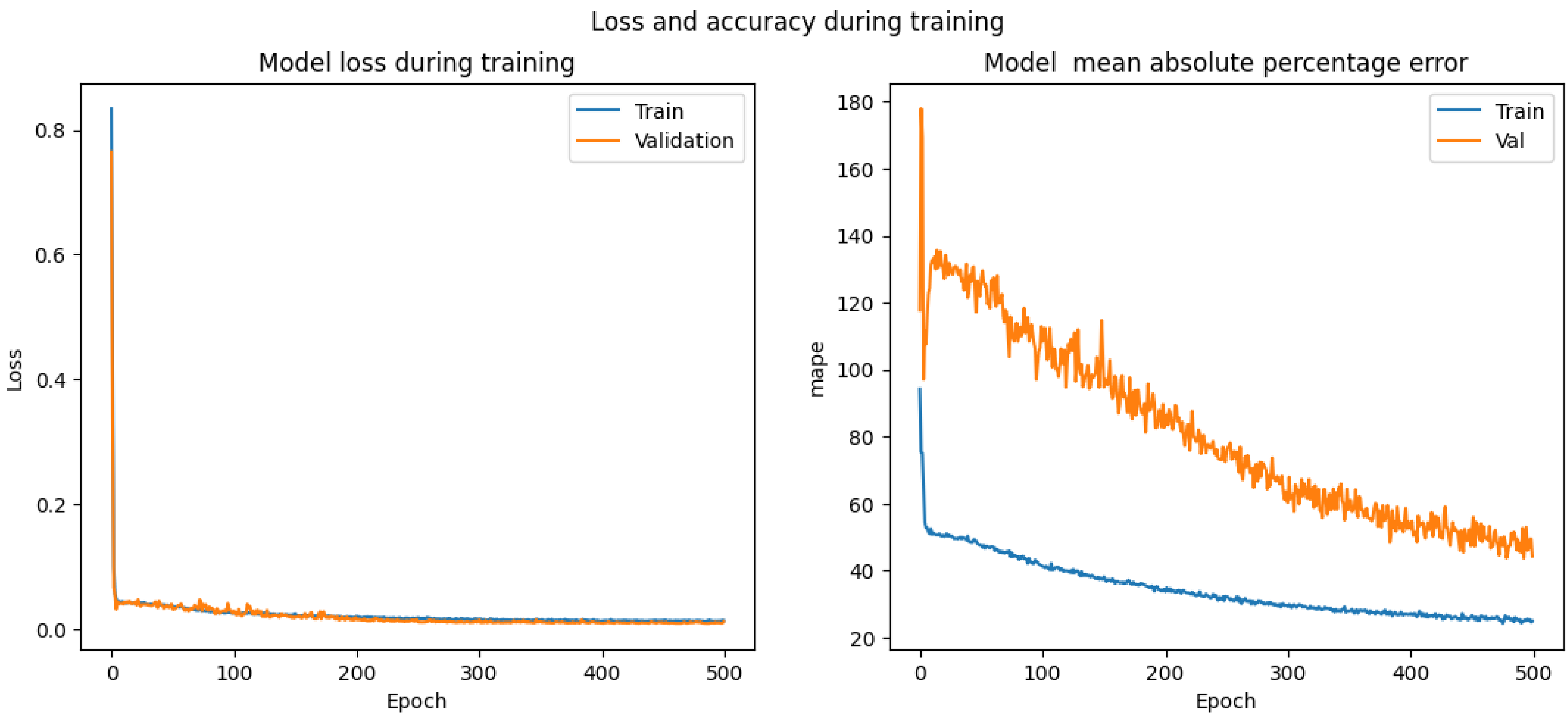
learning_rate = 0.00005

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

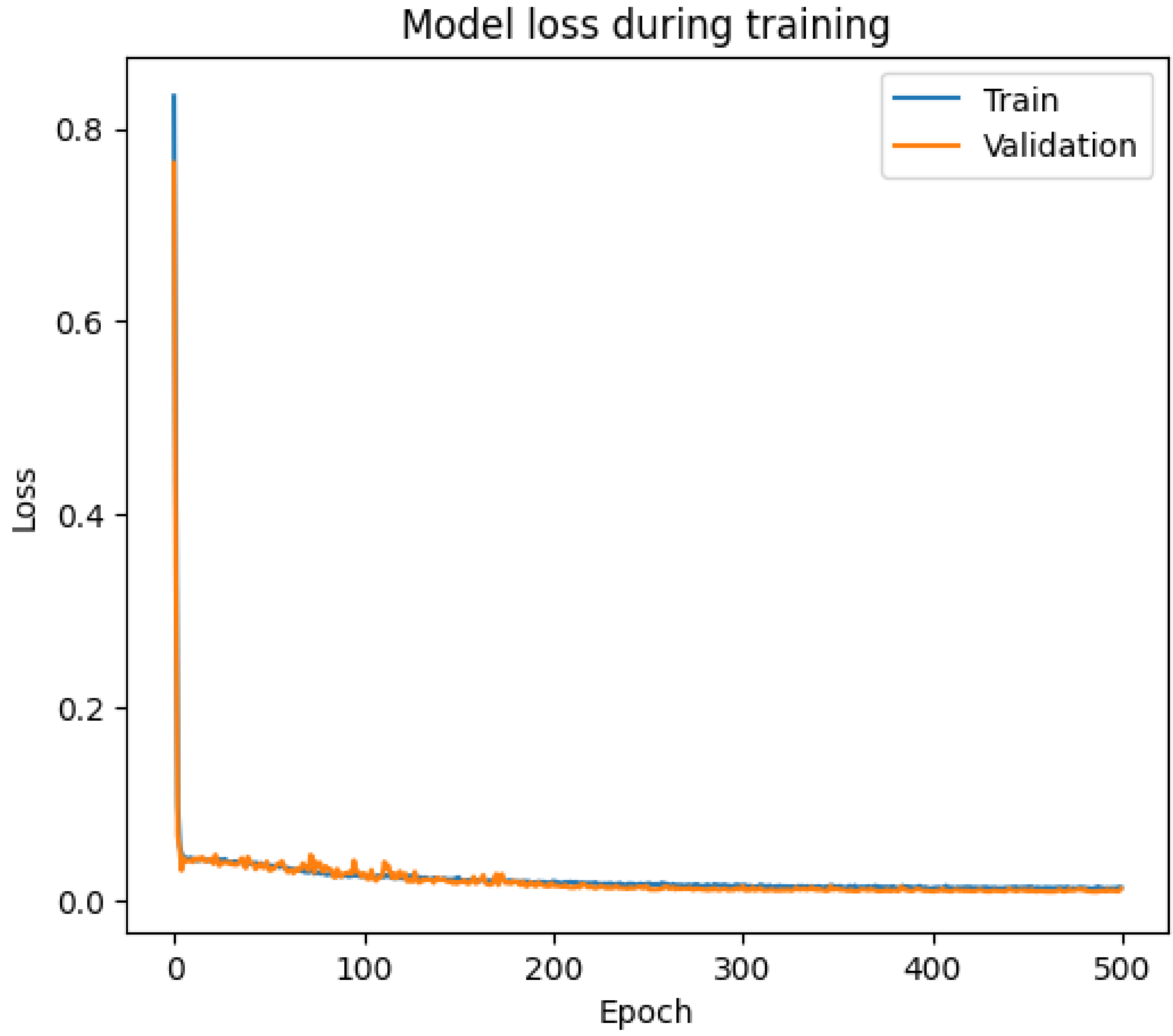


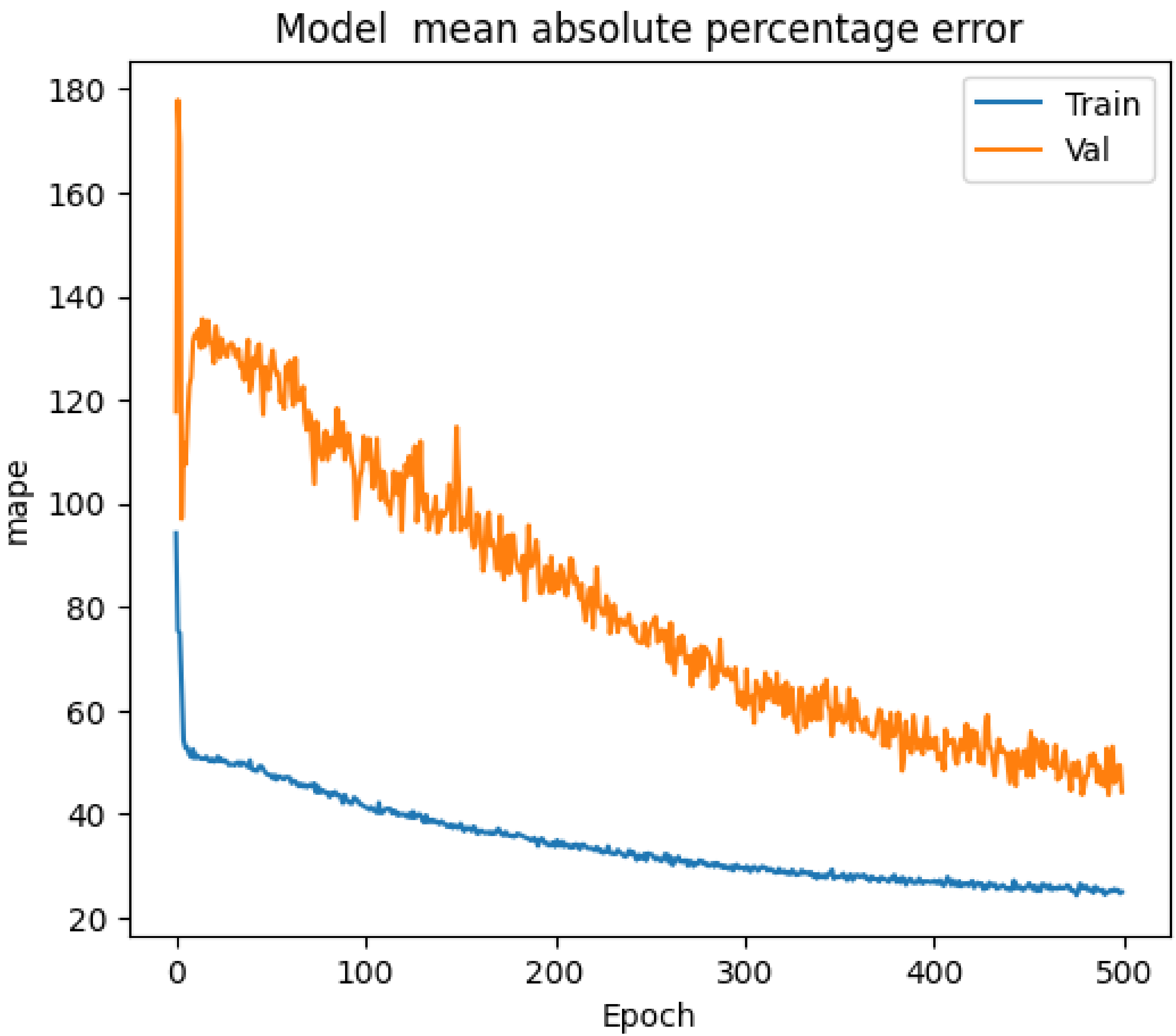
Formatted data (training)
window_size =7
batch_size = 10
Formatted input data
Unformatted data
[[-1.09307145]
[-1.09270821]
[-1.09247866]
[-1.09167145]
[-1.09215073]
[-1.09154532]
[-1.09078856]
[-1.09021847]
[-1.08807683]
[-1.08587718]
[-1.08688872]
[-1.08587214]
[-1.08597052]
[-1.08608655]
[-1.08476222]
[-1.08392726]
[-1.08271645]
[-1.08161158]
[-1.08182347]
[-1.0793867]
... ...]]
[[[-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]
... ...]
1st batch
2nd batch
3rd batch

Training the RNN

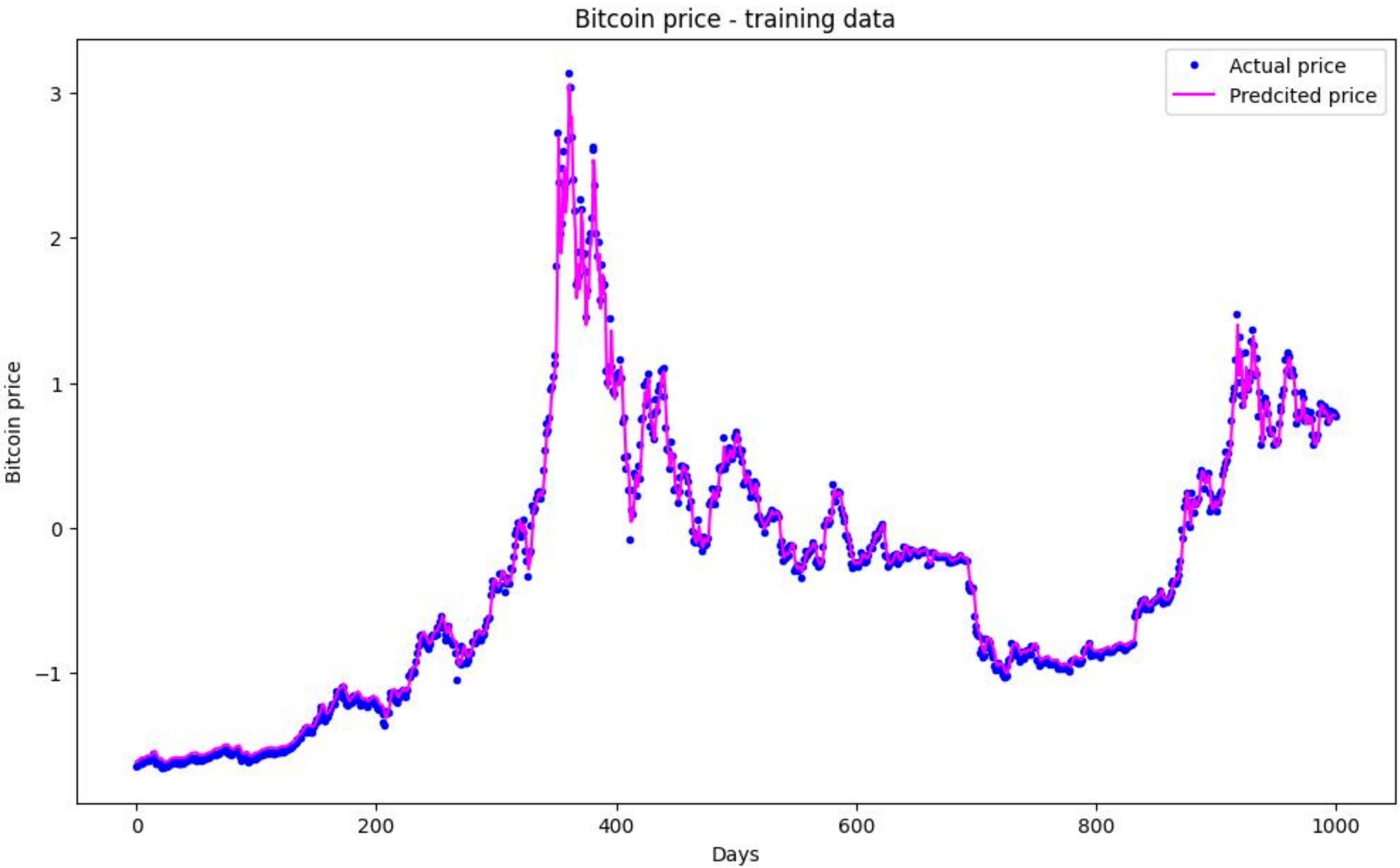



```
15/15 - 0s - loss: 0.0111 - mape: 44.2792 - 85ms/epoch - 6ms/step
Accuracy in the test data:  44.279170989990234
```


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

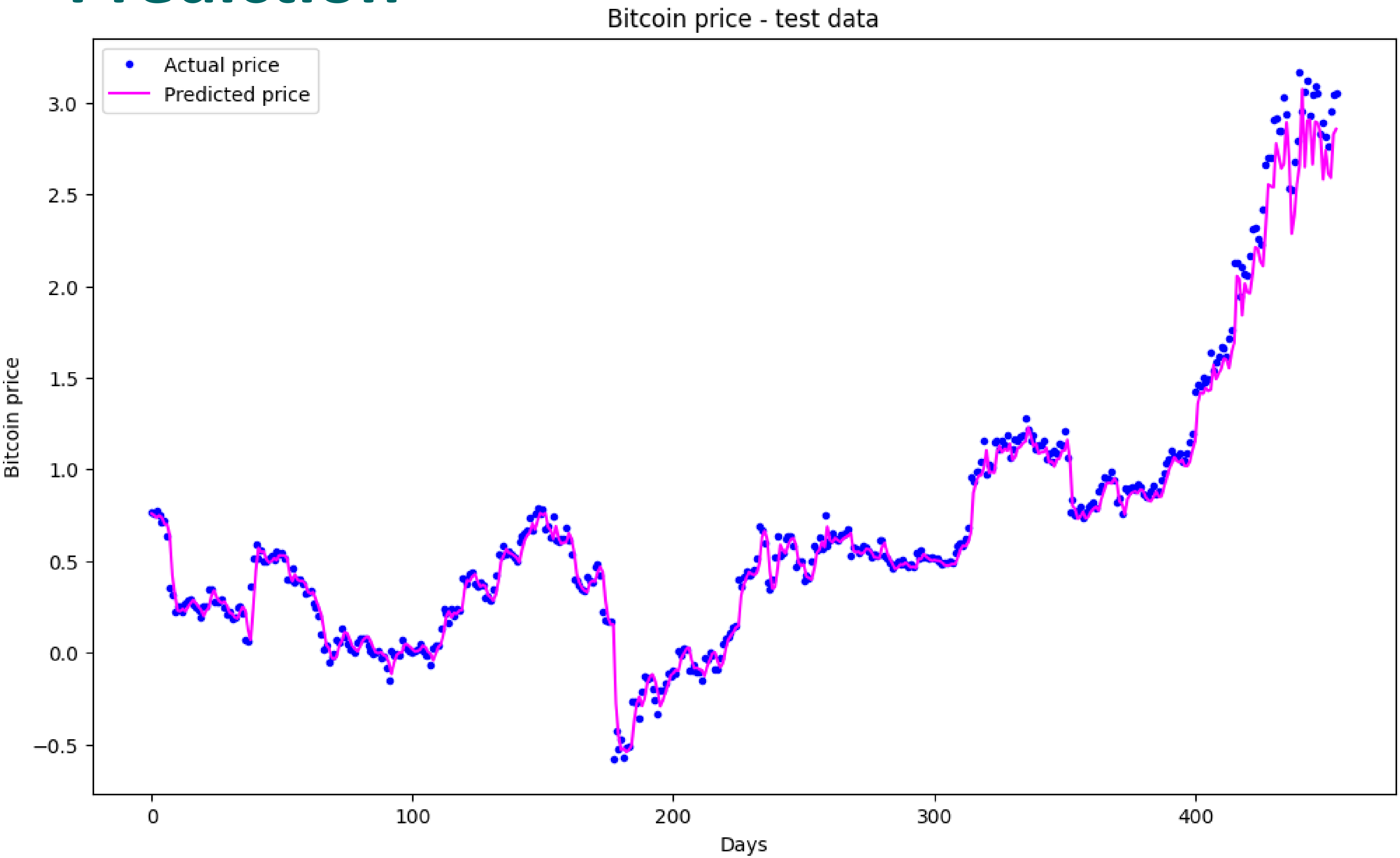

Model mean absolute percentage error
Train
Val
mape
Epoch
180
160
140
120
100
80
60
40
20
0
100
200
300
400
500

Prediction



Bitcoin price - training data
Actual price
Predcited price
Bitcoin price
Days
3
2
1
0
−1
0
200
400
600
800
1000

Prediction


Bitcoin price - test data
Actual price
Predicted price
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
Bitcoin price
0
100
200
300
400
Days

Using the Notebook file

This is a link to the application notebook:

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