50.039 Theory and Practice of Deep Learning

W6-S1 Times Series Data and Recurrent Neural Networks

Matthieu De Mari

About this week (Week 6)

1. What is a **time series dataset**?
2. How to **analyze a time series**, and what are **typical deep learning models** capable of doing that?
3. What is **history**, and why is it needed for time series predictions?
4. What is a **good history length**?
5. What is **memory** and how to represent it in a Neural Network model?
6. How to implement a first **vanilla Recurrent Neural Network** model?
7. Why is the **vanishing gradient problem** prominent in RNNs?

About this week (Week 6)

8. What is the **LSTM** model?
9. What is the **GRU** model?
10. What are **one-to-one**, **one-to-many**, **many-to-one** and **many-to-many** models?
11. What are some **typical application examples** of these?
12. What is a **Seq2Seq** model?
13. What are **encoder** and **decoder** architectures?
14. What is an **autoregressive RNN**?

Time series dataset

Definition (time series dataset):

A **time series dataset** is a **sequence of data points that are collected over time**, typically at regular intervals.

Time series data can be used to study how a particular variable changes over time, with many applications as

- Stock prices,
- Weather patterns,
- Etc.

Data points are usually ordered chronologically, with each data point representing the value of the variable at a specific point in time.

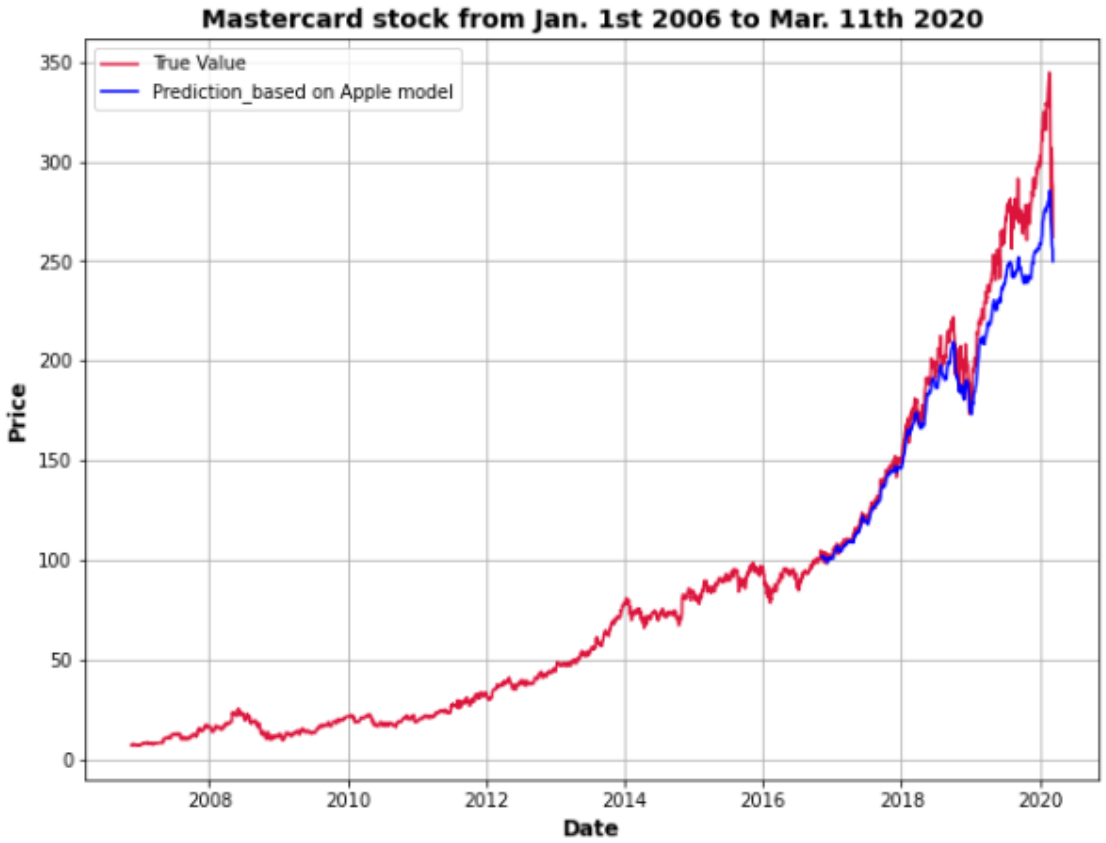


Figure 7. MAST stock price predictions using LSTM trained on AAPL

A mock dataset

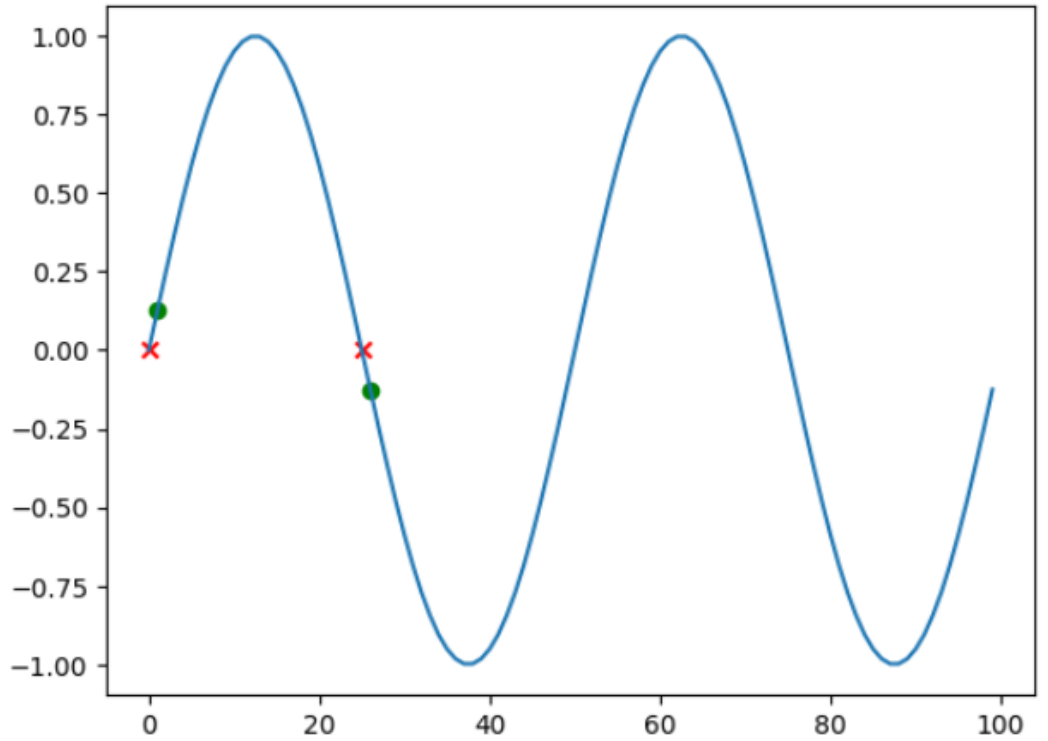
We will generate **a mock dataset for a time series** in Python.

- It will consist of a simple sinusoid curve, as shown, with values (t, s_t), and $t \in \{1, 2, \dots\}$.
- We will attempt to predict x_{t+1} using the previous value x_t.
- Out times series dataset D then consists of all the pairs $D = [(x_t, x_{t+1}) \mid t \in \{1, 2, \dots\}]$

```python
for batch in dataloader:
    inputs, outputs = batch
    print("Input for sample #0: ", inputs[0])
    print("Output for sample #0: ", outputs[0])
    print("Input for sample  #25: ", inputs[25])
    print("Output for sample  #25: ", outputs[25])
    break
```

```
Input for sample #0:  tensor([0.])
Output for sample #0:  tensor([0.1250])
Input for sample  #25:  tensor([0.])
Output for sample  #25:  tensor([-0.1250])
```

```python
# Show the first 100 points
plt.plot(datapoints[:100])
plt.scatter(0, inputs[0], c = 'r', marker = 'x')
plt.scatter(1, outputs[0], c = 'g')
plt.scatter(25, inputs[25], c = 'r', marker = 'x')
plt.scatter(26, outputs[25], c = 'g')
plt.show()
```



Typical methods for analysing a time series

Typical methods for analysing a time series include

1. **Descriptive statistics:** calculating basic summary statistics for the time series (mean, median, variance, etc.). These statistics can provide insights into the central tendency, variability, and distribution of the data.
2. **Visual inspection:** Plotting the time series data helps identify patterns, trends, and outliers in the data.

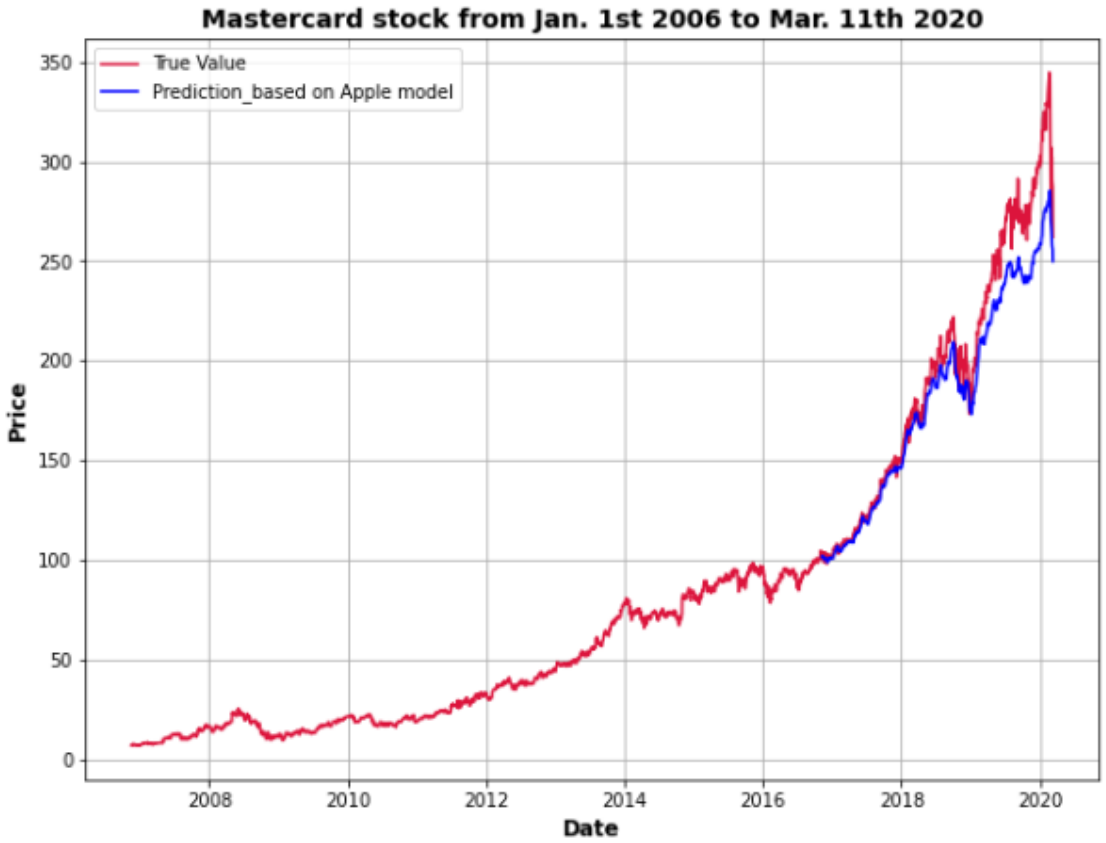


Figure 7. MAST stock price predictions using LSTM trained on AAPL

Typical methods for analysing a time series

3. **Decomposition:** Time series data often exhibit trends, seasonal patterns, and irregular fluctuations.

Decomposition is a method for separating these components and analysing them separately

Common methods for decomposition include additive and multiplicative decomposition.

(These are out-of-scope.)

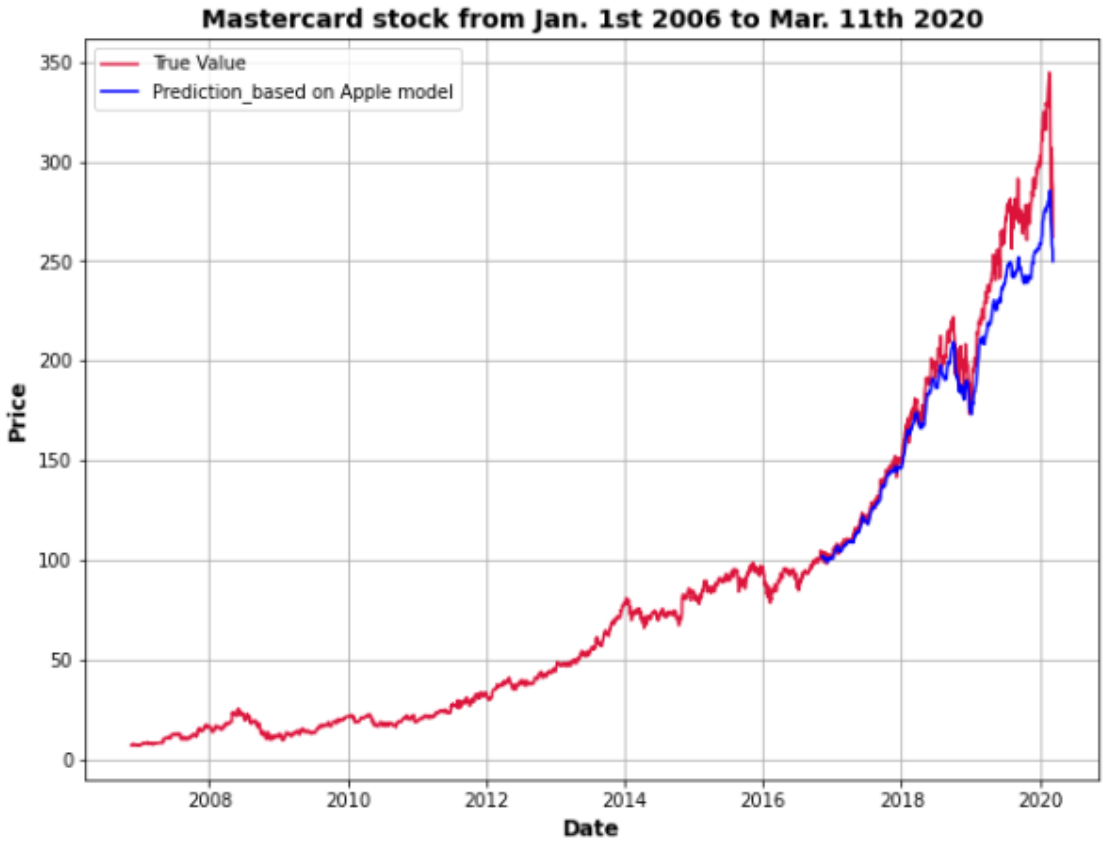


Figure 7. MAST stock price predictions using LSTM trained on AAPL

A quick word about decomposition

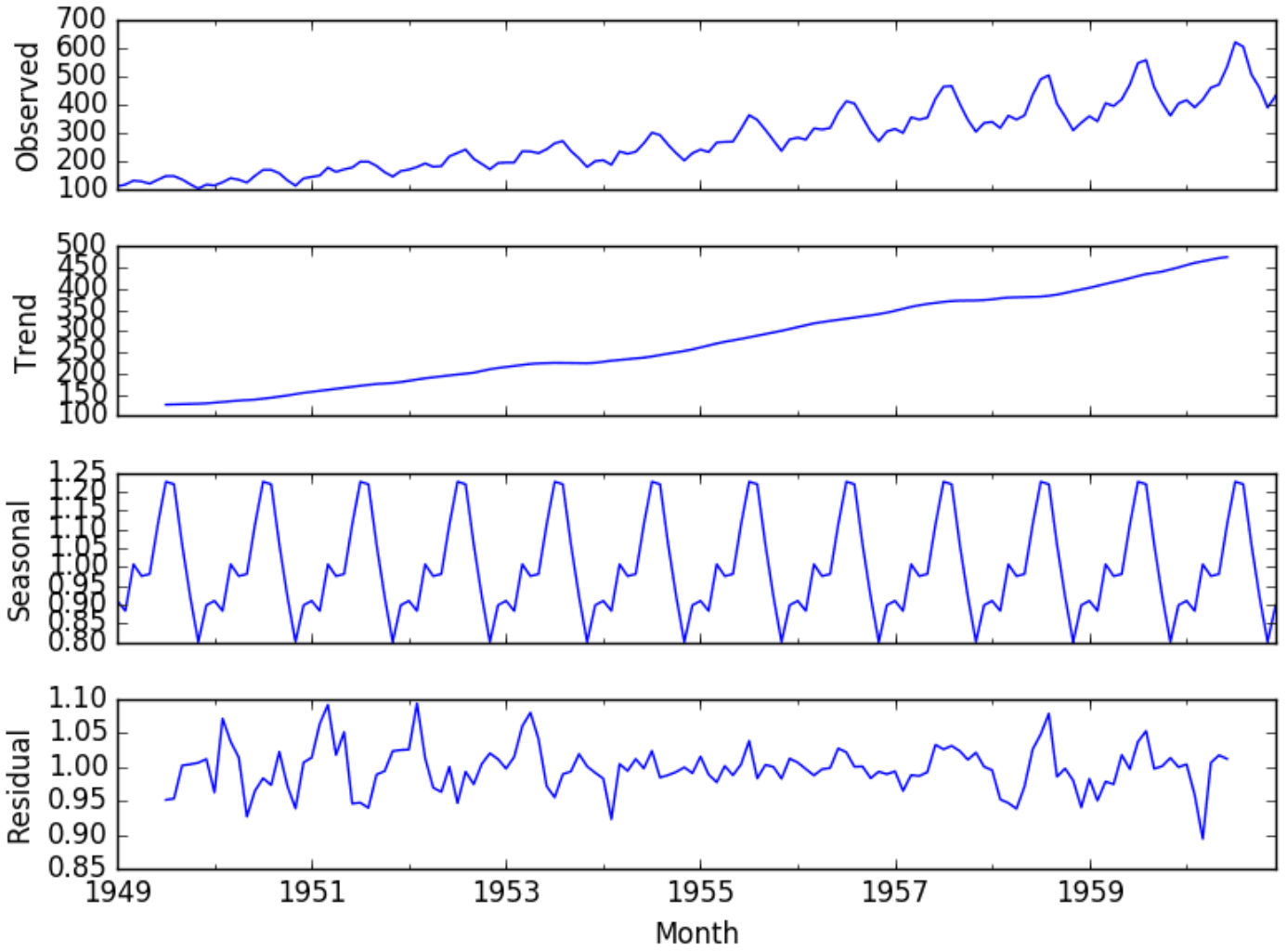
Decomposition separates the time series data into different components that capture distinct aspects of the data, expressing it as the sum of several underlying components, each with its own distinct pattern, e.g.:

1. **Trend:** This component represents the long-term changes or fluctuations in the data over time. It is typically modelled as a smooth curve that captures the overall direction of the time series.
2. **Seasonal:** This component captures the regular, periodic fluctuations in the data that occur over fixed periods of time, such as daily, weekly, or monthly. It is often modelled as a set of repeating patterns that correspond to the seasonal cycles.
3. **Residual:** This component represents the random or irregular fluctuations in the data that are not explained by the trend or seasonal components. It is often modelled as a series of random noise or white noise.

A quick word about decomposition

Decomposition separates a time series into **trend** (long-term changes), **seasonal** (periodic fluctuations), and **residual** (random fluctuations) components.

Trend is a smooth curve, **seasonal** is a set of repeating patterns, and **residual** is a series of random noise.



Typical methods for analysing a time series

4. **Time series models:** Statistical models used to describe the underlying processes that generate the time series data.

Time series models can be used for forecasting future values, identifying important predictors, and understanding the relationships among variables.

ML models: autoregressive integrated moving average (ARIMA) models, exponential smoothing models, state space models, etc.

(These are out-of-scope.)

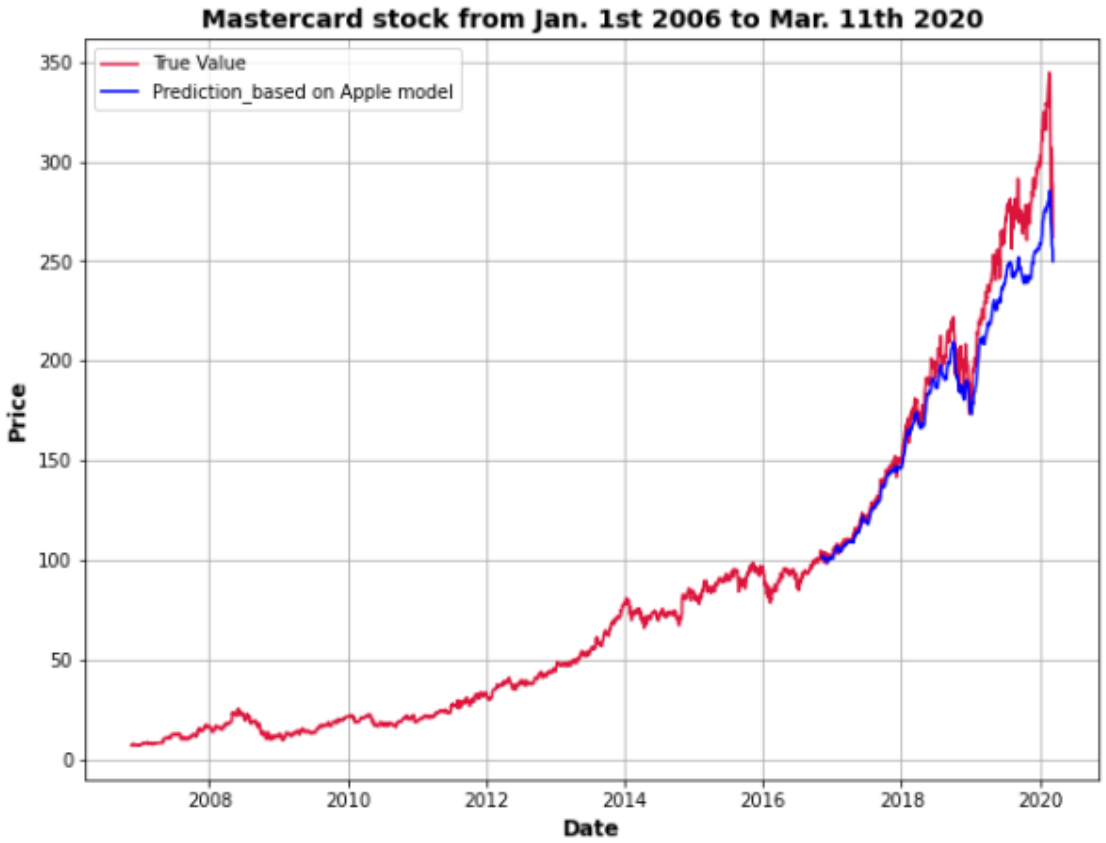


Figure 7. MAST stock price predictions using LSTM trained on AAPL

Typical methods for analysing a time series

4. **Time series models:** Statistical models used to describe the underlying processes that generate the time series data.

Time series models can be used for forecasting future values, identifying important predictors, and understanding the relationships among variables.

DL models: using neural networks as predictors, to predict the next value for a time series variable given a history as input.

(to be investigated!)

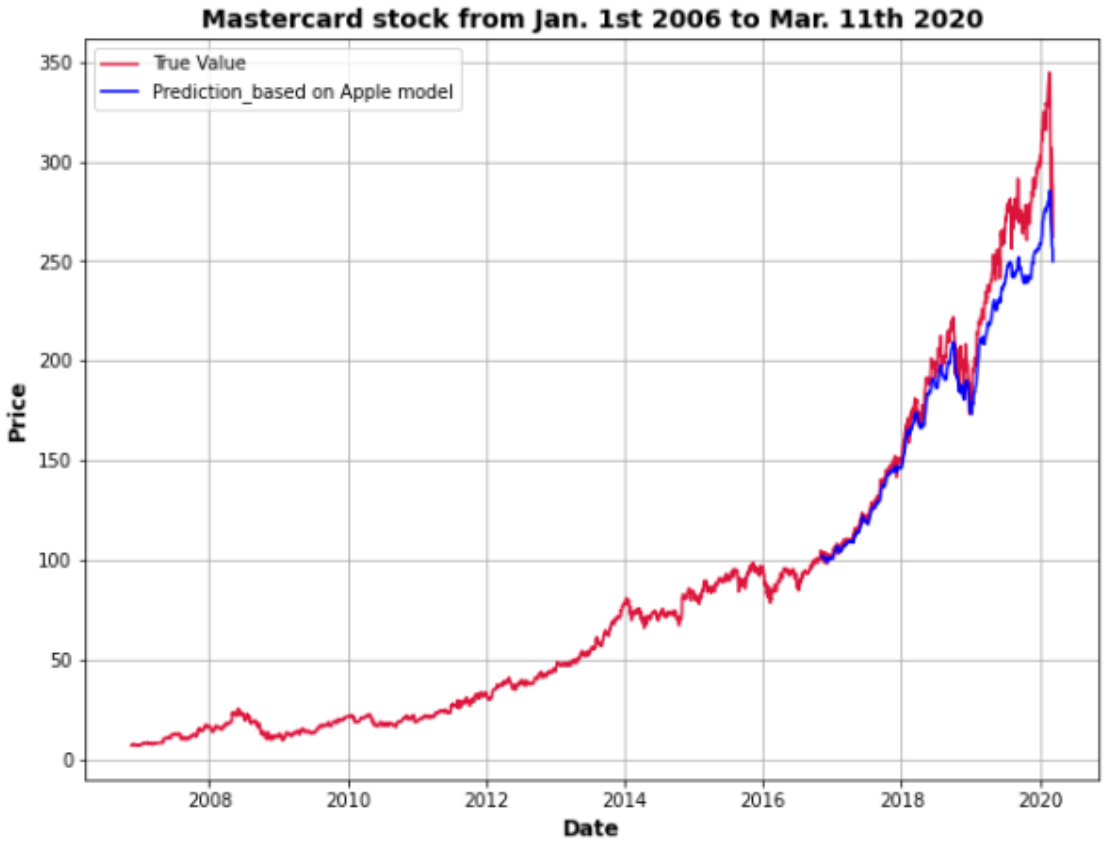


Figure 7. MAST stock price predictions using LSTM trained on AAPL

Problem with DL

Problem: Cannot use a single value x_t to make predictions on x_{t+1}.

We have a dataset D, where it is possible to find two values in time $t_1 \neq t_2$, such that:

$$\boldsymbol{x_{t_1} = x_{t_2}}$$

(here two values, with $t_1 = 0$ and $t_2 = 25$, displayed in red crosses),

And, yet have

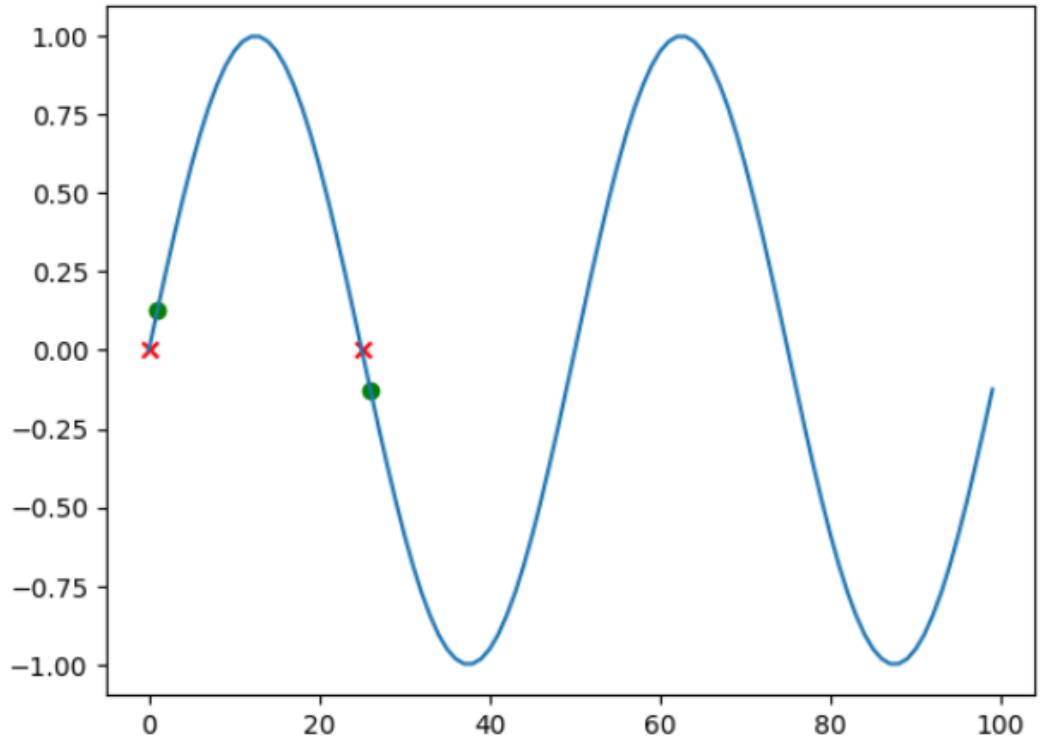
$$\boldsymbol{x_{t_1+1} \neq x_{t_2+1}}$$

(here two values, displayed in green).


```python
for batch in dataloader:
    inputs, outputs = batch
    print("Input for sample #0: ", inputs[0])
    print("Output for sample #0: ", outputs[0])
    print("Input for sample  #25: ", inputs[25])
    print("Output for sample  #25: ", outputs[25])
    break
```

```
Input for sample #0:  tensor([0.])
Output for sample #0:  tensor([0.1250])
Input for sample  #25:  tensor([0.])
Output for sample  #25:  tensor([-0.1250])
```

```python
# Show the first 100 points
plt.plot(datapoints[:100])
plt.scatter(0, inputs[0], c = 'r', marker = 'x')
plt.scatter(1, outputs[0], c = 'g')
plt.scatter(25, inputs[25], c = 'r', marker = 'x')
plt.scatter(26, outputs[25], c = 'g')
plt.show()
```



Problem with DL

Problem: Cannot use a single value x_t to make predictions on x_{t+1}.

We have a dataset D, where it is possible to find two values in time $t_1 \neq t_2$, such that:

$$\boldsymbol{x_{t_1} = x_{t_2}}$$

(here two values, with $t_1 = 0$ and $t_2 = 25$, displayed in red crosses),

And, yet have

$$\boldsymbol{x_{t_1+1} \neq x_{t_2+1}}$$

(here two values, displayed in green).

Problem (cont'd): Using a DNN which only takes x_t as input and attempts to predict x_{t+1} as output is then problematic.

- Our DNN is not smart enough to output two different values for the same given inputs $\boldsymbol{x_{t_1} = x_{t_2}}$.
- More importantly, the DNN does not know which "direction" the curve is currently going.

Using a simple DNN to predict

Let us try it, anyway: Let us try and define a simple DNN:

- 3 Linear Layers + ReLU,
- No ReLU on final layer,
- Number of inputs = 1,
- Number of outputs = 1,
- Hidden layers sizes: 32 and 8.

```
class DNN(torch.nn.Module):

    def __init__(self):
        super(DNN, self).__init__()
        self.layers = torch.nn.Sequential(torch.nn.Linear(1, 32),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(32, 8),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(8, 1))

    def forward(self, inputs):
        out = self.layers(inputs)
        return out
```

```
model = DNN().to(device)
print(model)
```

```
DNN(
  (layers): Sequential(
    (0): Linear(in_features=1, out_features=32, bias=True)
    (1): ReLU()
    (2): Linear(in_features=32, out_features=8, bias=True)
    (3): ReLU()
    (4): Linear(in_features=8, out_features=1, bias=True)
  )
)
```

Using a simple DNN to predict

Let us try it, anyway: Let us try and define a simple DNN:

- 3 Linear Layers + ReLU,
- No ReLU on final layer,
- Number of inputs = 1,
- Number of outputs = 1,
- Hidden layers sizes: 32 and 8.

Simple trainer function, and seems to train, but is it good?

```python
def train(model, dataloader, num_epochs, learning_rate):
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)

    for epoch in range(num_epochs):
        for inputs, targets in dataloader:
            optimizer.zero_grad()
            outputs = model(inputs.to(device))
            loss = criterion(outputs, targets.to(device))
            loss.backward()
            optimizer.step()

        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item():.4f}")
```

```python
# Train the model
model = DNN().to(device)
train(model, dataloader, num_epochs = 20, learning_rate = 0.1)
```

```
Epoch 1/20, Loss: 0.0383
Epoch 2/20, Loss: 0.0249
Epoch 3/20, Loss: 0.0102
Epoch 4/20, Loss: 0.0077
Epoch 5/20, Loss: 0.0078
Epoch 6/20, Loss: 0.0075
Epoch 7/20, Loss: 0.0072
Epoch 8/20, Loss: 0.0072
Epoch 9/20, Loss: 0.0073
Epoch 10/20, Loss: 0.0074
Epoch 11/20, Loss: 0.0074
Epoch 12/20, Loss: 0.0075
Epoch 13/20, Loss: 0.0075
Epoch 14/20, Loss: 0.0076
Epoch 15/20, Loss: 0.0076
Epoch 16/20, Loss: 0.0077
Epoch 17/20, Loss: 0.0077
Epoch 18/20, Loss: 0.0077
Epoch 19/20, Loss: 0.0078
Epoch 20/20, Loss: 0.0078
```

Using a simple DNN to predict

This low loss obtained after training does not mean much…

The model cannot learn to predict correctly: it is not smart enough to output two different values for the same given inputs $\boldsymbol{x_{t_1}} = \boldsymbol{x_{t_2}}$.

Also, it needs to know which "direction" the curve is currently going.

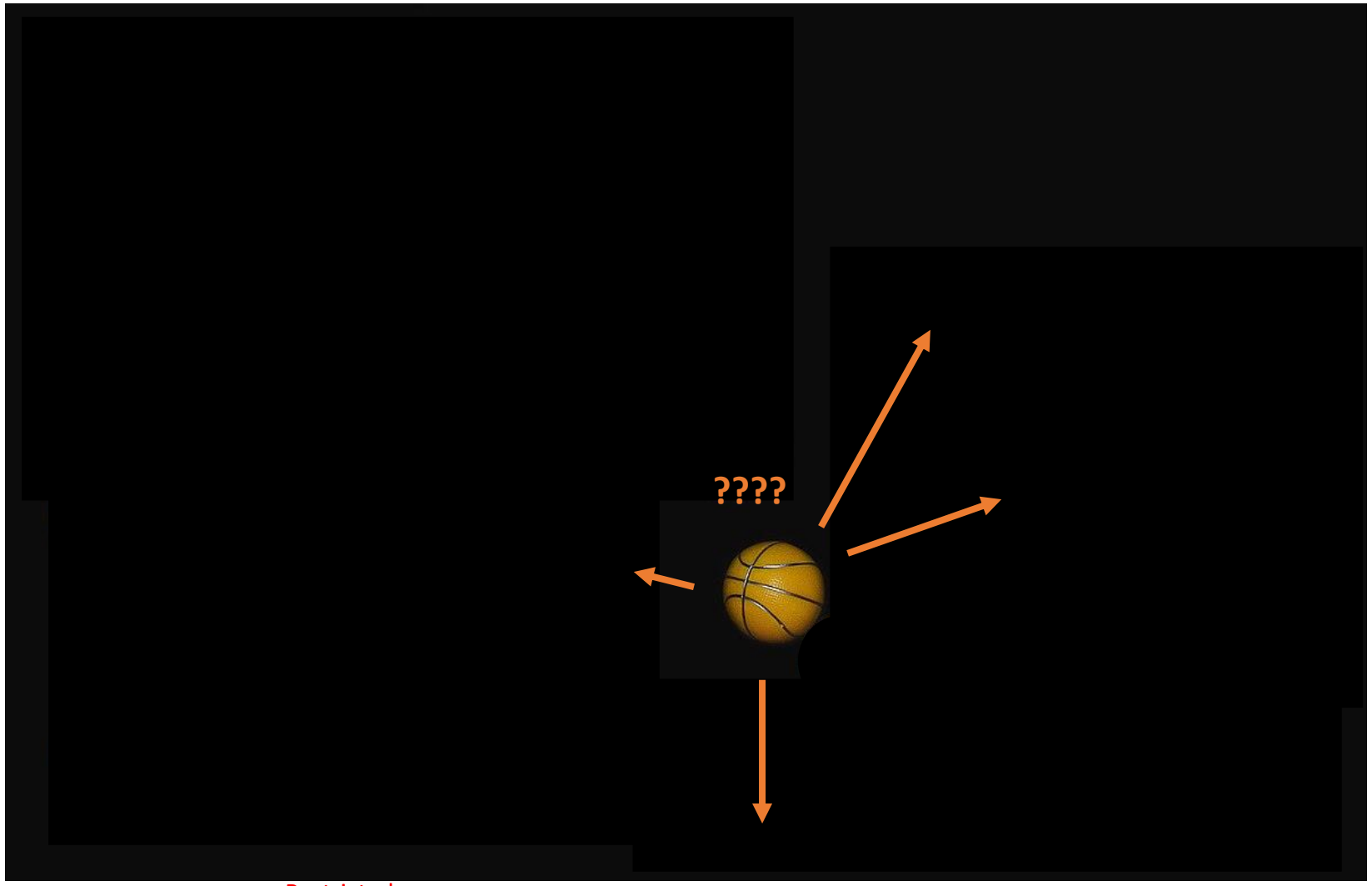
```
for batch in dataloader:
    inputs, outputs = batch
    print("Input for sample #0: ", inputs[0])
    print("Output for sample #0: ", outputs[0])
    print("Model prediction for sample #0:", model(torch.tensor([[inputs[0]], ]).to(device)))
    print("Input for sample  #25: ", inputs[25])
    print("Output for sample  #25: ", outputs[25])
    print("Model prediction for sample #0:", model(torch.tensor([[inputs[25]], ]).to(device)))
    break
```

```
Input for sample #0:  tensor([0.])
Output for sample #0:  tensor([0.1250])
Model prediction for sample #0: tensor([[0.0080]], device='cuda:0', grad_fn=<AddmmBackward0>)
Input for sample  #25:  tensor([0.])
Output for sample  #25:  tensor([-0.1250])
Model prediction for sample #0: tensor([[0.0080]], device='cuda:0', grad_fn=<AddmmBackward0>)
```

A visual example for the need of “direction”

Let us pretend that x_t is the position of the basketball at time t.

Looking at the picture on the right, can you guess what the next ball position x_{t+1} will be?

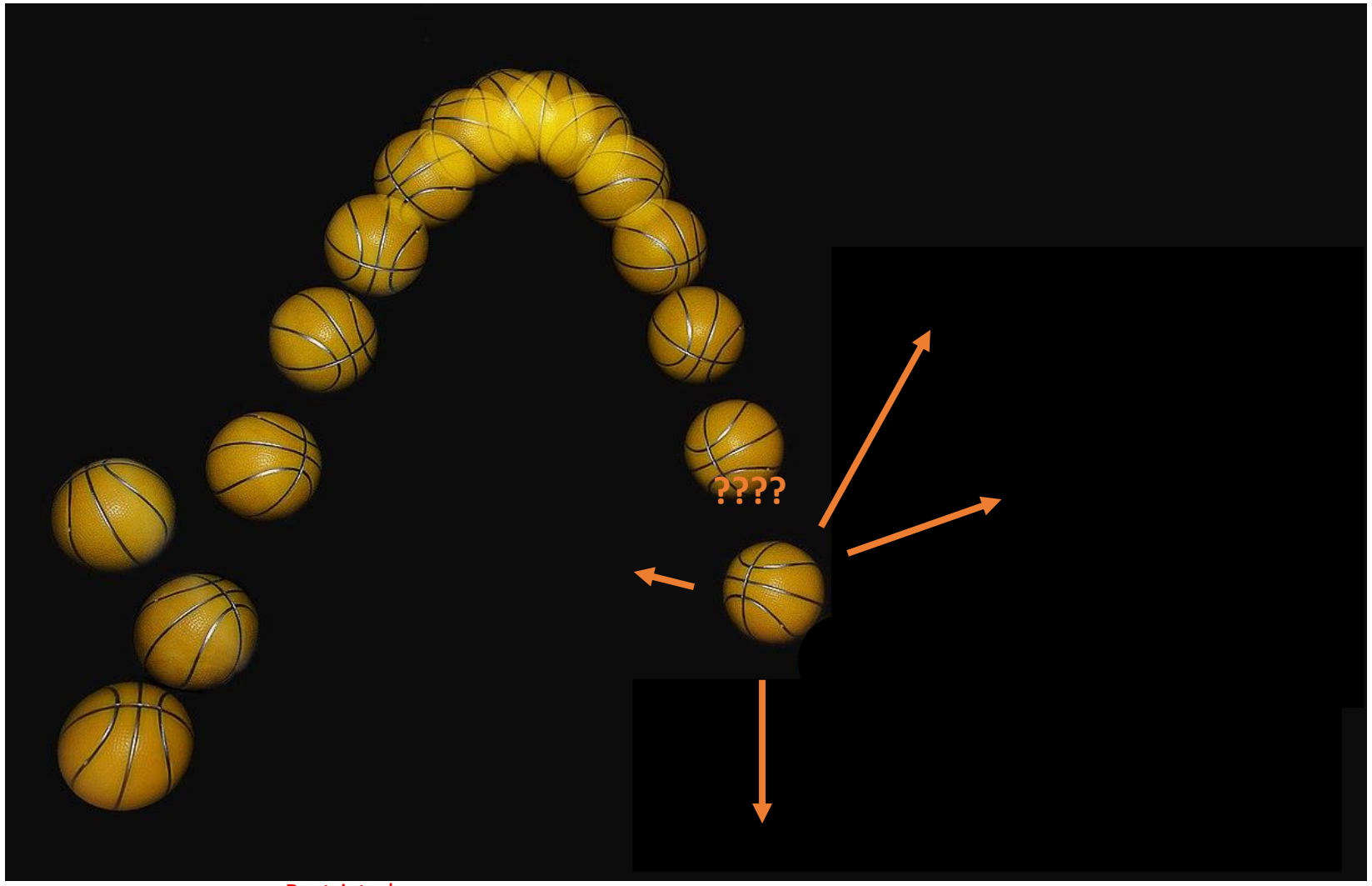


A visual example for the need of “direction”

Let us pretend that x_t is the position of the basketball at time t.

Looking at the picture on the right, can you guess what the next ball position x_{t+1} will be?

What if I give you a bit of history now?



A visual example for the need of “direction”

Definition (history):

History refers to the **past observations or values of the time series up to a certain point in time**.

It provides information about the patterns and trends of the data over time, which can be used to make predictions and identify important factors that influence the time series.

E.g., predicting a stock price, requires to know the history of the stock prices to understand its behaviour.

A visual example for the need of “direction”

As a consequence: when predicting x_{t+1}, what plays the role of inputs should be a subset of $(x_1, x_2, \ldots, x_t)$, consisting of all the observations from the beginning until the present time t.

For instance, could we predict the next position of the ball knowing only the last two frames, i.e. the last two positions of the ball?

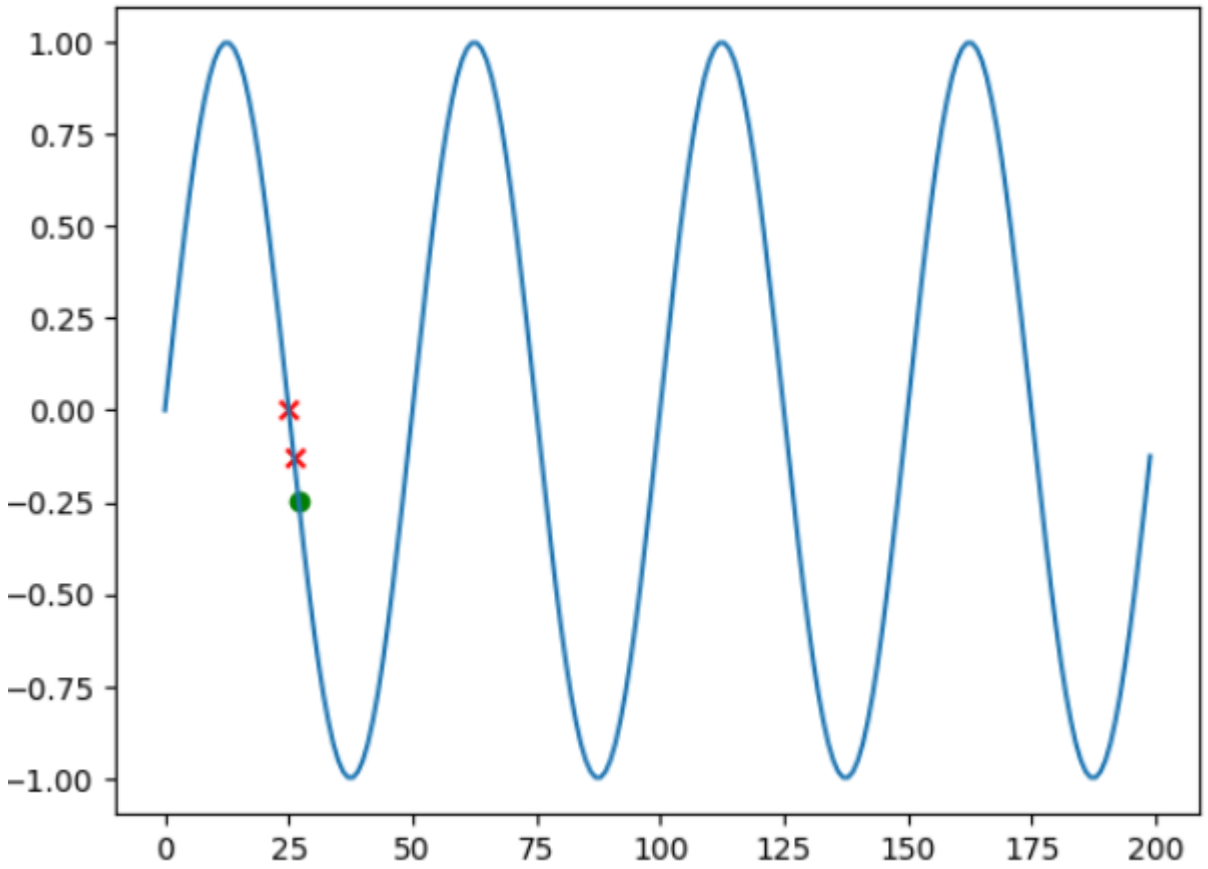
Lookback size

Definition (lookback size):

The **number of past observations or values of a time series that are used to make a prediction** is commonly referred to as the "**lookback**" or "**window**" **size**.

It is the length of the history that is used to inform the prediction of the next value in the time series.

In our example, the lookback size is simply 2.



Using a simple DNN to predict

Let us rewrite our previous DNN model, with some changes:

- 3 Linear Layers + ReLU,
- No ReLU on final layer,
- **Number of inputs = 2, being (x_{t-1}, x_t) this time,**
- Number of outputs = 1, x_{t+1},
- Hidden layers sizes: 32 and 8.

```python
class DNN_model_with_history(torch.nn.Module):
    def __init__(self, history_len):
        super(DNN_model_with_history, self).__init__()

        # Lag
        self.history_len = history_len
        # Define layers
        self.layers = torch.nn.Sequential(torch.nn.Linear(self.history_len, 32),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(32, 8),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(8, 1))

    def forward(self, inputs):
        out = self.layers(inputs)
        return out
```

```python
model = DNN_model_with_history(history_len = 2).to(device)
print(model.layers)
```

```
Sequential(
  (0): Linear(in_features=2, out_features=32, bias=True)
  (1): ReLU()
  (2): Linear(in_features=32, out_features=8, bias=True)
  (3): ReLU()
  (4): Linear(in_features=8, out_features=1, bias=True)
)
```

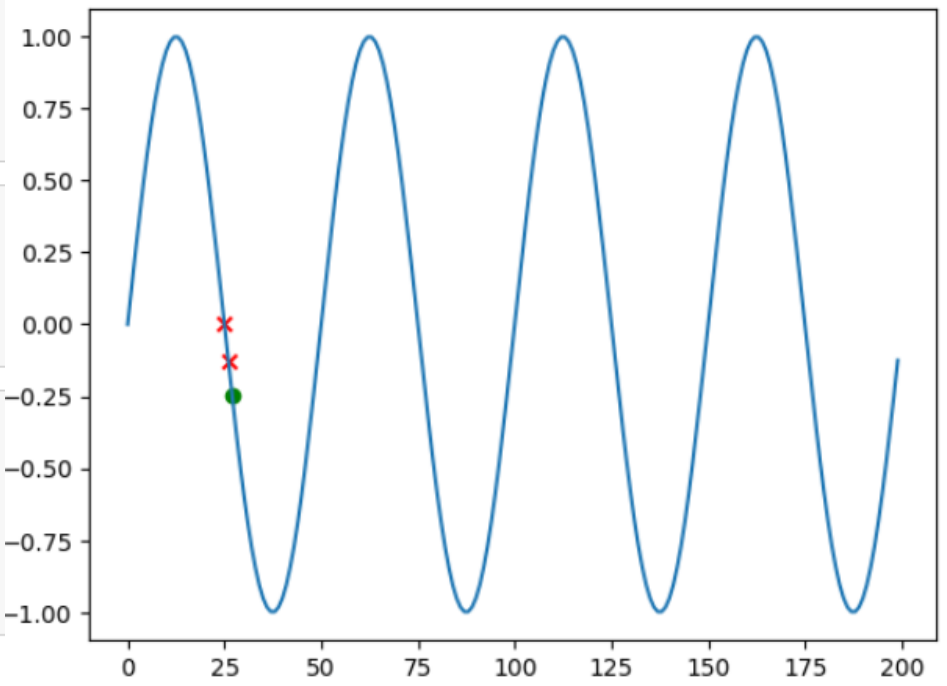
Reworking the Dataset to include history

```python
# Our custom Dataset object
class CustomSeriesDataset(Dataset):
    def __init__(self, datapoints, history_len = 2):
        self.history_len = history_len
        self.inputs_number = self.history_len
        self.outputs_number = 1
        self.datapoints = datapoints
        self.inputs = [datapoints[i:(i+self.inputs_number)] \
                       for i in range(len(datapoints) - self.inputs_number - 1)]
        self.outputs = [datapoints[(i+self.inputs_number):(i + self.inputs_number + self.outputs_number)] \
                        for i in range(len(datapoints) - self.inputs_number - 1)]
    def __len__(self):
        return len(self.outputs)
    def __getitem__(self, index):
        inputs = torch.tensor(self.inputs[index]).float()
        outputs = torch.tensor(self.outputs[index]).float().reshape(-1)
        return inputs, outputs
```

```python
# Create dataset and dataloader
np.random.seed(27)
datapoints = [np.sin(2*np.pi*i/50) for i in range(1000)]
dataset = CustomSeriesDataset(datapoints, history_len = 2)
dataloader = DataLoader(dataset, batch_size = 32, shuffle = True)
```

```python
# Fetch a datapoint
torch.manual_seed(15)
data = next(iter(dataloader))
inputs_test = data[0][20]
outputs_test = data[1][20]
print(inputs_test)
print(outputs_test)
```

```
tensor([0.3681, 0.4818])
tensor([0.5878])
```



Using a simple DNN to predict

Almost as before:

- 3 Linear Layers + ReLU,
- No ReLU on final layer,
- **Number of inputs = 2, being (x_{t-1}, x_t) this time,**
- Number of outputs = 1, x_{t+1},
- Hidden layers sizes: 32 and 8.

Same trainer function, and seems to train with lower loss values, but again, is it good?

```python
def train(model, dataloader, num_epochs, learning_rate, device):
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)

    for epoch in range(num_epochs):
        for inputs, targets in dataloader:
            optimizer.zero_grad()
            outputs = model(inputs.to(device))
            loss = criterion(outputs.to(device), targets.to(device))
            loss.backward()
            optimizer.step()

        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item():.4f}")
```

```python
# Train the model
model = DNN_model_with_lag(lag = 2).to(device)
train(model, dataloader, num_epochs = 20, learning_rate = 0.01, device = device)
```

```
Epoch 1/20, Loss: 4.6535
Epoch 2/20, Loss: 0.9900
Epoch 3/20, Loss: 0.2949
Epoch 4/20, Loss: 0.0384
Epoch 5/20, Loss: 0.0006
Epoch 6/20, Loss: 0.0007
Epoch 7/20, Loss: 0.0004
Epoch 8/20, Loss: 0.0053
Epoch 9/20, Loss: 0.0005
Epoch 10/20, Loss: 0.0155
Epoch 11/20, Loss: 0.0022
Epoch 12/20, Loss: 0.0013
Epoch 13/20, Loss: 0.0004
Epoch 14/20, Loss: 0.0001
Epoch 15/20, Loss: 0.0011
Epoch 16/20, Loss: 0.1153
Epoch 17/20, Loss: 0.0001
Epoch 18/20, Loss: 0.1178
Epoch 19/20, Loss: 0.0000
Epoch 20/20, Loss: 0.0059
```

Using a simple DNN to predict

This time the predictor works, as we obtain the good prediction values.

In the example below, we had $x_{t_1} = x_{t_2}$, but $x_{t_1-1} \neq x_{t_2-1}$.

This was good enough to help the DNN formulate two good different predictions x_{t_1+1} and x_{t_2+1}.

```
for batch in dataloader:
    inputs, outputs = batch
    print("Input for sample #24: ", inputs[24])
    print("Output for sample #24: ", outputs[24])
    print("Model prediction for sample #24:", model(inputs[24].to(device)))
    print("Input for sample  #49: ", inputs[49])
    print("Output for sample  #49: ", outputs[49])
    print("Model prediction for sample #49:", model(inputs[49].to(device)))
    break
```

```
Input for sample #24:  tensor([0.1250, 0.0000])
Output for sample #24:  tensor([-0.1253])
Model prediction for sample #24: tensor([-0.1281], device='cuda:0', grad_fn=<AddBackward0>)
Input for sample  #49:  tensor([-0.1250, -0.0000])
Output for sample  #49:  tensor([0.1253])
Model prediction for sample #49: tensor([0.1248], device='cuda:0', grad_fn=<AddBackward0>)
```

What is a good history length then?

In general: Having more datapoints in the history (i.e. a greater lookback length), will usually lead to a model that performs better.

Therefore, when attempting to predict x_{t+1}, at time t,

- We will be using x_t as input (normal),
- And the ideal history h_t would then consist of all the elements from $t' = 1$, to the present time i.e. $t' = t - 1$, that is $h_t = (x_1, x_2, \ldots, x_{t-1})$.

This could pose a problem as the history length will change over time, accounting for more and more data points being added to history over time.

The problem of varying length

History vectors will then have varying lengths.

For instance, when attempting to predict x_4, at time $t = 3$,

- You will be using x_3 as input (normal),
- And the history would then consist of all the elements from $t' = 1$, to the present time i.e. $t' = 4 - 1 = 2$, that is $h_3 = (x_1, x_2)$.

Whereas, when attempting to predict x_{11}, at time $t = 10$,

- You will be using x_{10} as input (normal),
- And the history would then consist of all the elements from $t' = 1$, to the present time i.e. $t' = 10 - 1 = 9$, that is $h_{10} = (x_1, x_2, \dots, x_9)$.

The problem of varying length

Problem: Variable history length cannot work with our current neural network because **it requires fixed-length input sequences**.

```
class DNN_model_with_history(torch.nn.Module):
    def __init__(self, history_len):
        super(DNN_model_with_history, self).__init__()

        # Lag
        self.history_len = history_len
        # Define layers
        self.layers = torch.nn.Sequential(torch.nn.Linear(self.history_len, 32),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(32, 8),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(8, 1))

    def forward(self, inputs):
        out = self.layers(inputs)
        return out
```

The problem of varying length

Problem: Variable history length cannot work with our current neural network because **it requires fixed-length input sequences**.

A quick fix would be to assume a fixed length l for the history vector h_t to be used at all times:

- We could **truncate** the history vectors that have a length greater than l, keeping the last elements only, i.e. $h_t = (x_{t-l}, \dots, x_{t-1})$.
- If too short, we could use **padding** for the history vectors that have a lower length than l, adding zeros to artificially increase their length. For instance, after padding, we could have $h_3 = (0, \dots, 0, x_1, x_2)$.

The problem of varying length

Problem: Variable history length cannot work with our current neural network because **it requires fixed-length input sequences**.

A quick fix would be to assume a fixed length l for the history vector h_t to be used at all times:

- We could **truncate** the history vectors that have a length greater than l, keeping the last elements only, i.e. $h_t = (x_{t-l}, \dots, x_{t-1})$.
- If too short, we could use **padding** for the history vectors that have a lower length than l, adding zeros to artificially increase their length. For instance, after padding, we could have $h_3 = (0, \dots, 0, x_1, x_2)$.

However, we could do much better than this.

The need for memory

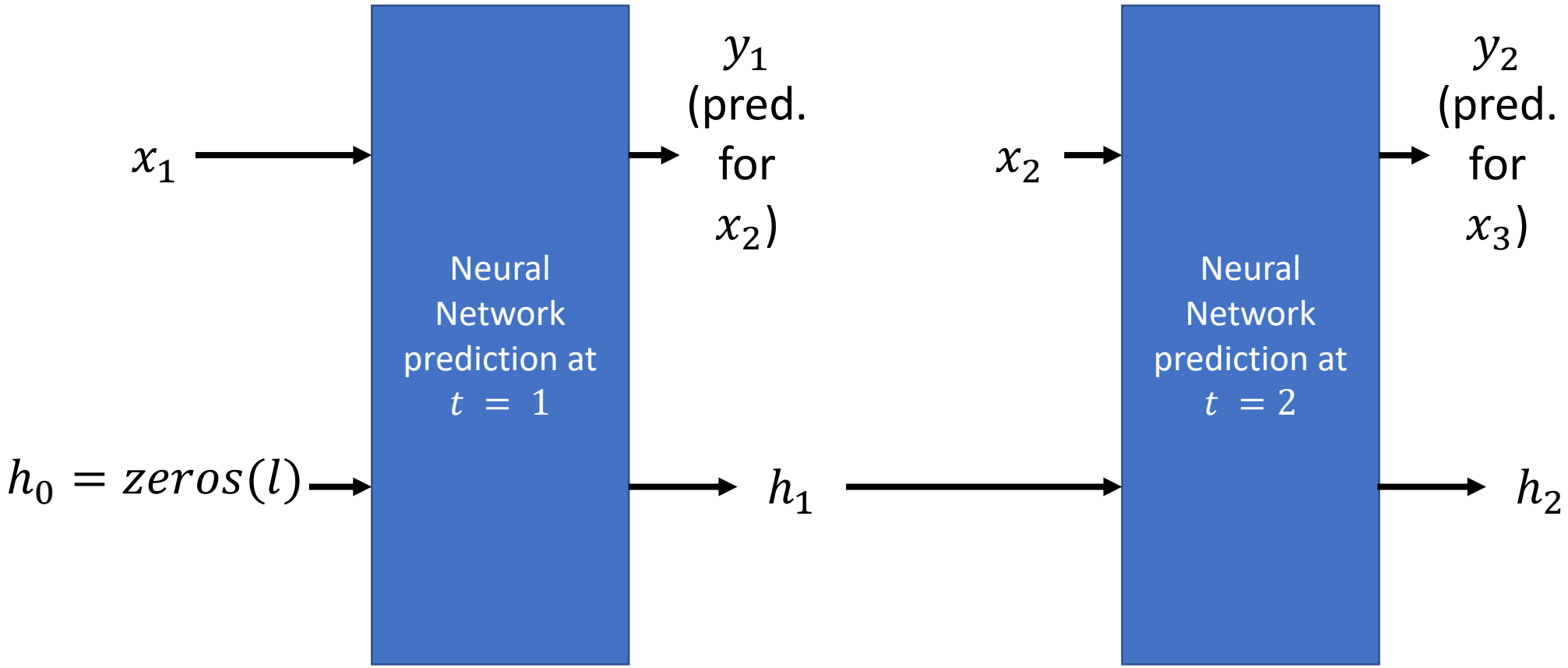
Definition (memory or hidden vector in RNNs):

In practice however, it is preferable to define a **memory (or hidden) vector** and leave it to the Neural Network to figure out what information should put in there.

On each time t, the Neural Network:

- Would receive an input which consists of **the observation x_t at time t**, and a **memory vector h_t computed as one of the Neural Network outputs at time $t-1$**.
- Would compute a **prediction y_t for what might be the value of x_{t+1}**, and an **updated memory vector h_{t+1},** hopefully keeping a memory of what has happened in the previous operations.

The need for memory



Etc.
(use for loop on time t)

Defining a Recurrent Neural Network (RNN)

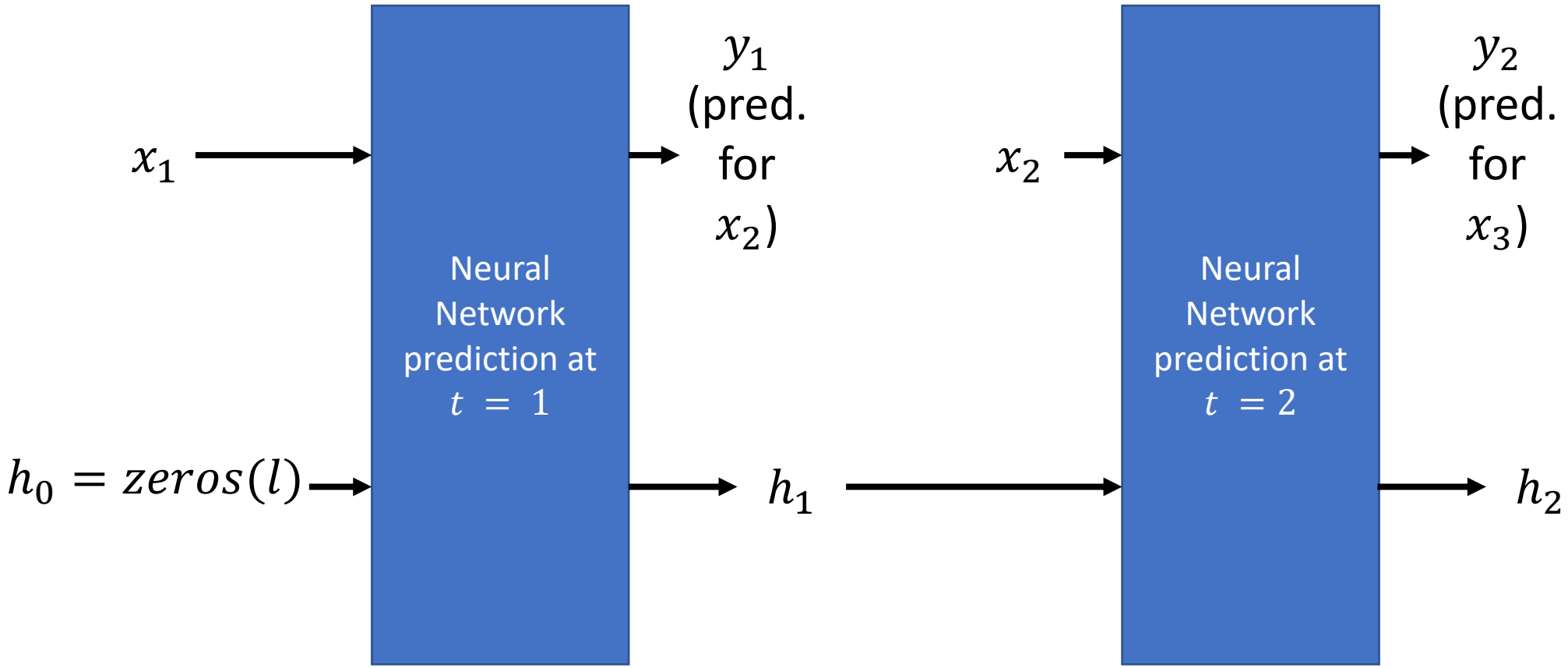
Definition (Recurrent Neural Network):

A **Recurrent Neural Network (or RNN)** is a neural network that operates on series and will:

- Receive an input which consists of **the observation x_t at time t**, and a **memory vector h_t computed as one of the Neural Network outputs at time $t-1$**.
- Compute a **prediction y_t for what should match the value of x_{t+1}**, and an **updated memory vector h_{t+1},** hopefully keeping a memory of what has happened in the previous operations.

The RNN is then used on all datapoints in the time series, using a for loop repeating the forward pass operation on all data points.

The need for memory



Etc.
(use for loop on time t)

```python
# Our custom Dataset object
class CustomSeriesDataset(Dataset):
    def __init__(self, datapoints):
        self.datapoints = datapoints
        self.inputs = [datapoints[i] for i in range(len(datapoints) - 1)]
        self.outputs = [datapoints[i + 1] for i in range(len(datapoints) - 1)]

    def __len__(self):
        return len(self.outputs)

    def __getitem__(self, index):
        inputs = torch.tensor(self.inputs[index]).float()
        outputs = torch.tensor(self.outputs[index]).float()
        return inputs, outputs
```

```python
# Create dataset and dataloader
dataset = CustomSeriesDataset(datapoints)
dataloader = DataLoader(dataset, batch_size = 1, shuffle = False)
```

```python
inputs, outputs = dataset[0]
print("Input for sample #0: ", inputs)
print("Output for sample #0: ", outputs)
inputs, outputs = dataset[25]
print("Input for sample  #25: ", inputs)
print("Output for sample  #25: ", outputs)
```

```
Input for sample #0:  tensor(0.)
Output for sample #0:  tensor(0.1250)
Input for sample  #25:  tensor(0.)
Output for sample  #25:  tensor(-0.1250)
```

Implementing a RNN

Let us rewrite our first RNN model, which resembles the previous DNN with some changes:

- 3 Linear Layers + ReLU,
- No ReLU on final layer,
- **Number of inputs = 2, being (x_t, h_t) this time,**
- **Number of outputs = 2, being (y_t, h_{t+1}),**
- Hidden layers sizes: 32 and 8.

```
class RNN(torch.nn.Module):

    def __init__(self):
        super(RNN, self).__init__()
        self.layers = torch.nn.Sequential(torch.nn.Linear(2, 32),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(32, 8),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(8, 2))

    def forward(self, inputs, hidden):
        combined = torch.tensor([inputs, hidden]).to(inputs.device)
        out = self.layers(combined)
        return out
```

Notice how the forward method expects two values, being x_t (inputs) and h_t (hidden). They will be combined before going through the NN layers.

Implementing a RNN

Our trainer function is almost the same as before, except that:

- We initialize a hidden tensor with zero value,
- Outputs are split into y_t (in variable out) and the new hidden vector h_{t+1} (in variable hidden),
- Loss function uses y_t (in variable out) and target.

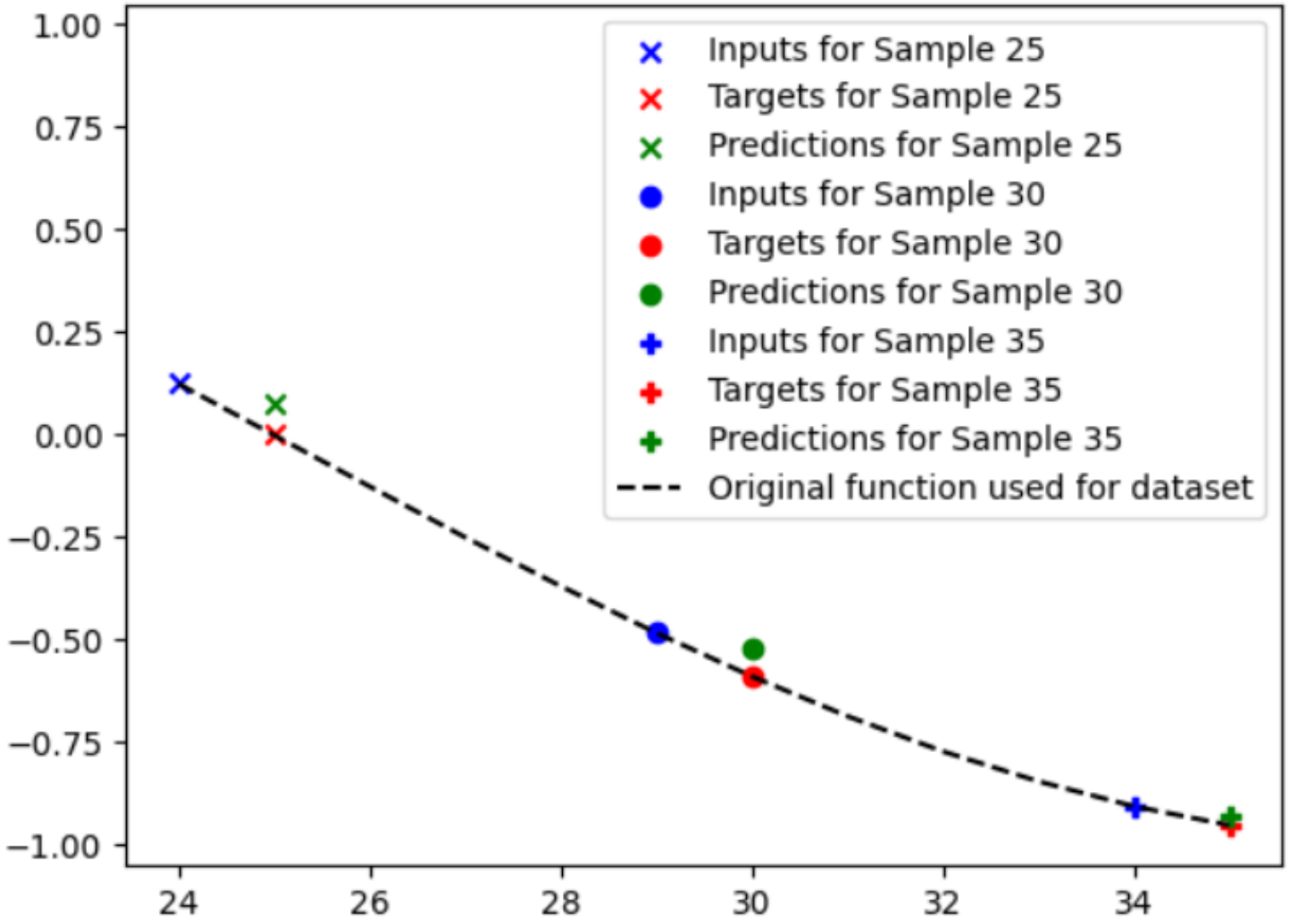
```
def train(model, dataloader, num_epochs, learning_rate, device):
    criterion = torch.nn.MSELoss()
    optimizer = torch.optim.Adam(model.parameters(), lr = learning_rate)
    hidden = torch.tensor([0]).to(device)
    for epoch in range(num_epochs):
        loss = 0
        for inputs, targets in dataloader:
            optimizer.zero_grad()
            outputs = model(inputs.to(device), hidden)
            out, hidden = outputs[0], outputs[1]
            loss += criterion(out.to(device), targets.to(device))
        loss /= len(dataloader)
        loss.backward()
        optimizer.step()

        print(f"Epoch {epoch + 1}/{num_epochs}, Loss: {loss.item():.4f}")
```

It trains!

```
# Train the model
model = RNN().to(device)
train(model, dataloader, num_epochs = 100, learning_rate = 0.1, device = device)
```

```
Epoch 1/100, Loss: 0.6229
Epoch 2/100, Loss: 0.3301
Epoch 3/100, Loss: 0.1675
Epoch 4/100, Loss: 0.1387
Epoch 5/100, Loss: 0.1081
Epoch 6/100, Loss: 0.1354
Epoch 7/100, Loss: 0.0655
Epoch 8/100, Loss: 0.1487
Epoch 9/100, Loss: 0.0623
Epoch 10/100, Loss: 0.0445
Epoch 11/100, Loss: 0.0718
Epoch 12/100, Loss: 0.0507
Epoch 13/100, Loss: 0.0573
Epoch 14/100, Loss: 0.0709
Epoch 15/100, Loss: 0.0554
Epoch 16/100, Loss: 0.0324
Epoch 17/100, Loss: 0.0331
Epoch 18/100, Loss: 0.0315
```

And it is not too bad at predicting!



Improving our RNN

Our first RNN model, uses

- 3 Linear Layers + ReLU,
- No ReLU on final layer,
- **Number of inputs = 2, being (x_t, h_t) this time.**
- **Number of outputs = 2, being (y_t, h_{t+1}).**
- Hidden layers sizes: 32 and 8.

```
class RNN(torch.nn.Module):

    def __init__(self):
        super(RNN, self).__init__()
        self.layers = torch.nn.Sequential(torch.nn.Linear(2, 32),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(32, 8),
                                          torch.nn.ReLU(),
                                          torch.nn.Linear(8, 2))

    def forward(self, inputs, hidden):
        combined = torch.tensor([inputs, hidden]).to(inputs.device)
        out = self.layers(combined)
        return out
```

Question: The memory vector h_t currently consists of only 1 element, but what if we increased the size of the memory vector h_t to have now $l \neq 1$ elements?

What is a good memory length then?

Question: How would we decide the size l of the memory vector h_t to use for our RNN?

Remember: We said earlier that *"In general: Having more datapoints in the history (i.e. a greater history length), will usually lead to a model that performs better."*

Same intuition: It would make sense that having a higher size l of the memory vector h_t would allow the model to remember more things and therefore lead to better performance.

What is a good memory length then?

In practice,

1. **Consider the time scale of the problem:** The appropriate lookback length typically depends on the **time scale of the problem** you are trying to solve.

For example, if you are trying to predict daily stock prices, you might use a lookback length of 30 days. If you are trying to predict hourly traffic patterns, you might use a lookback length of 24 hours.

(Typically, look at seasonality and trend of your data!)

→ **The more information you need in the memory, the larger the memory size of it should be.**

What is a good memory length then?

In practice,

2. **Balance model complexity with overfitting:** Finding the optimal balance between model complexity and overfitting is a key consideration in choosing a lookback length.

A larger lookback length will capture more of the historical patterns and trends in the data, but it can also increase the complexity of the model and lead to overfitting.

A smaller lookback length may be simpler and less prone to overfitting, but it may also miss important patterns and trends in the data.

→ **Tradeoff: memory length vs. underfitting/overfitting.**

Defining a Recurrent Neural Network (RNN)

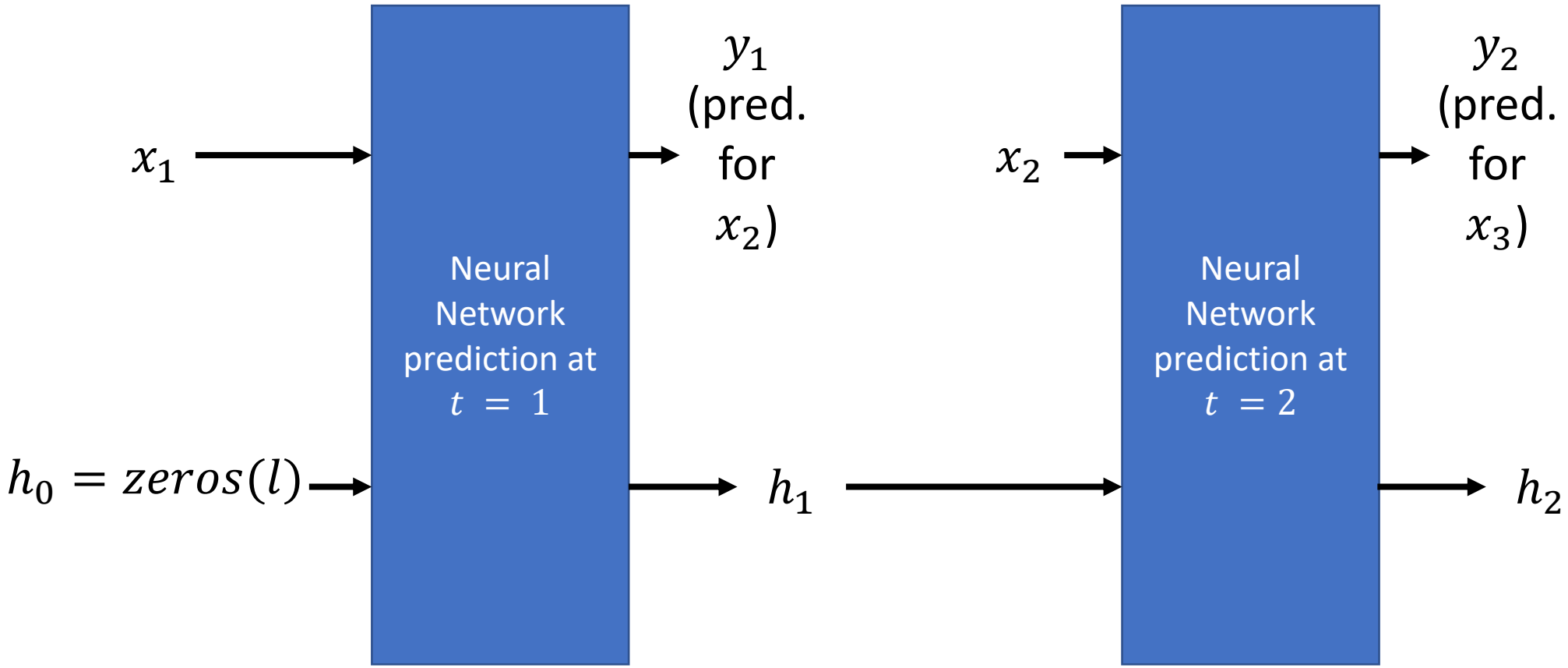
Definition (Recurrent Neural Network):

A **Recurrent Neural Network (or RNN)** is a neural network that operates on series and will:

- Receive an input which consists of **the observation x_t at time t**, and a **memory vector h_t computed as one of the Neural Network outputs at time $t-1$**.
- Compute a **prediction y_t for what should match the value of x_{t+1}**, and an **updated memory vector h_{t+1},** hopefully keeping a memory of what has happened in the previous operations.

The RNN is then used on all datapoints in the time series, using a for loop repeating the forward pass operation on all data points.

The need for memory



Etc.
(use for loop on time t)

Backpropagation through time

At the moment, we have a RNN, which, at each time t, will:

- Process each input x_t,
- Attempt to predict the value of x_{t+1},
- Keeps track of some memory, in a vector h_t, updated at each time t,
- Adjusts the parameters of the RNN layers at each time t.

This means that we perform **1 parameter update per sample** (as we would if we used **stochastic gradient descent** on a standard DNN).

We have however seen that using batches of **N data samples per parameter update** was better (a.k.a. as mini-batch gradient descent).

Backpropagation through time

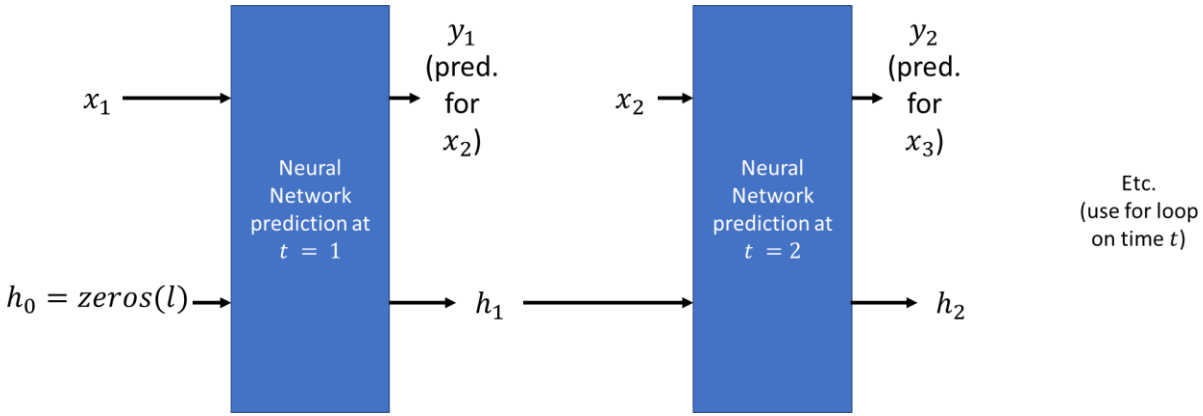
Definition (backpropagation through time):

In order to implement **a parameter update every N samples**, we need to implement a **backpropagation through time**.

- Unfold the RNN operations over time as shown,
- Run predictions on N consecutive samples and keep track of loss,
- Eventually perform one parameter update after N samples.

This basically allows to **teach the RNN how to update the memory vectors over time**.

Somewhat equivalent to a chain of successive layers, with gradient propagated through all the layers and time steps.

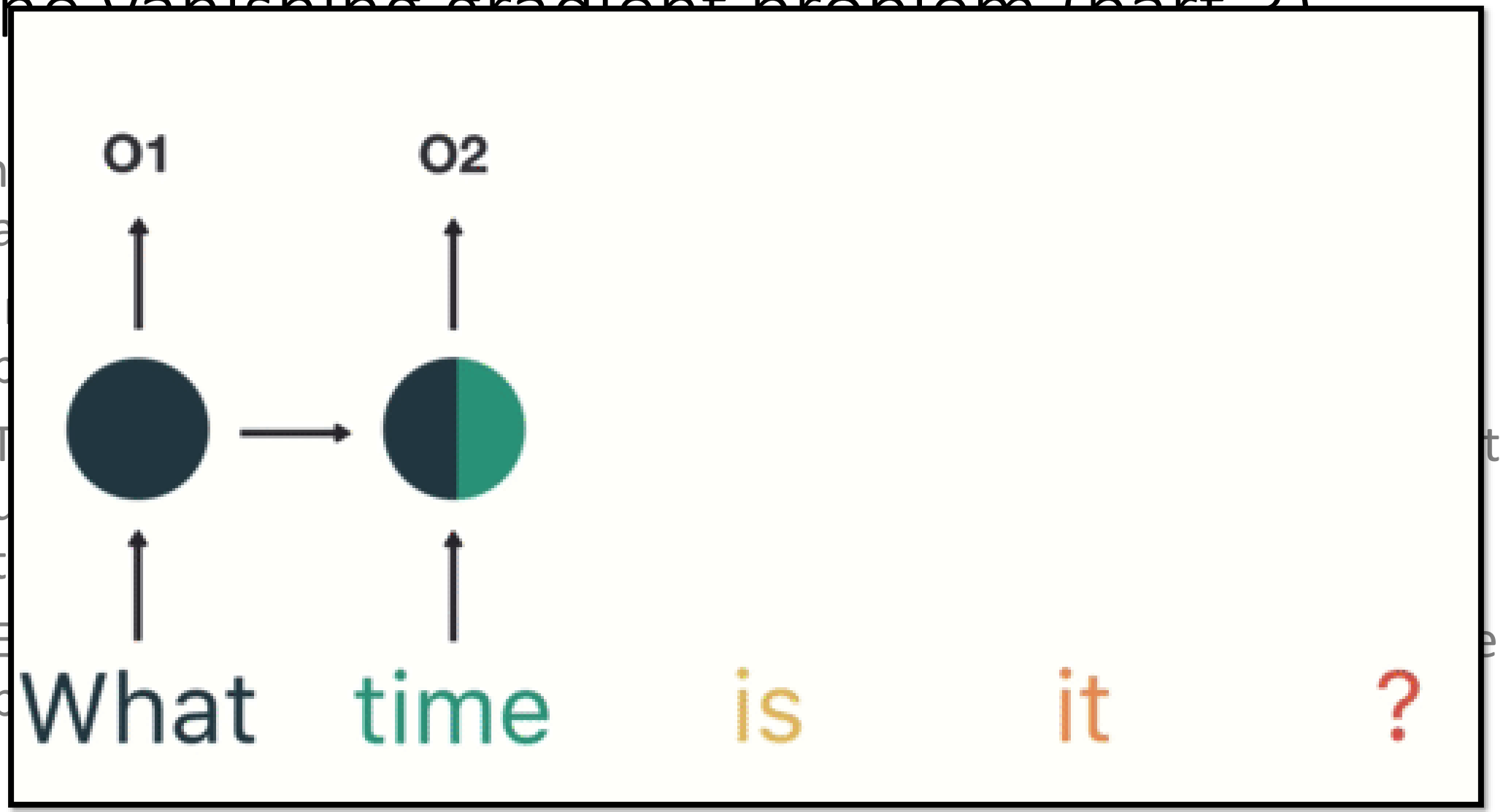


The vanishing gradient problem (part 3)

The **vanishing gradient problem** can often occur in RNNs during training.

- In RNNs, each hidden state h_{t+1} is updated via a function of the previous hidden state h_t and the input x_t.
- The memory vector will attempt to integrate “new” information, but ultimately will have to discard “old” information to make space for the “new” information.
- Eventually “old” information will slowly be lost over time, which iis unfortunate as it could have proven useful in some predictions…

The vanishing gradient problem (part 2)



The vanishing gradient problem (part 3)

The **vanishing gradient problem** can often occur in RNNs during training.

- In RNNs, each hidden state h_{t+1} is updated via a function of the previous hidden state h_t and the input x_t.
- During training, we used backpropagation over time, learning how to update the memory vectors over time for that problem/dataset.
- In general, our derivatives wrt. trainable parameters will be products of many other derivatives.
- **This backpropagation can result in very small gradients for the earlier time steps**, where the weights are then updated very slowly.

The vanishing gradient problem (part 3)

The **vanishing gradient problem** can often occur in RNNs during training.

- This can be particularly problematic for RNNs, since they are designed to handle time series data that can exhibit long-term dependencies.
- The vanishing gradient problem can be particularly severe in RNNs that use activation functions that saturate, such as the hyperbolic tangent (tanh) or sigmoid functions.
- Indeed, when the activation function saturates, the gradients become very small, making it difficult to propagate information through the network.

The vanishing gradient problem (part 3)

How to address the vanishing gradient problem in RNNs?

- Several techniques have been developed to address the vanishing gradient problem in RNNs, such as using alternative activation functions, initializing the weights of the network carefully, etc.
- **The "best" (?) solution, at the moment, suggests using specialized architectures like Long Short-Term Memory (LSTM) networks or Gated Recurrent Units (GRUs) to process memory.**
- These are designed to better handle long-term dependencies, and can help to mitigate the vanishing gradient problem.
- This eventually improves the performance of RNNs on time series. (To be discussed on the next lecture).