Machine Learning Applications

Lecture 5: Recurrent Neural Networks

Lecture Notes

M.Sc. in Financial Markets

ACADEMIC YEAR 2021-2022

Dr. Dominic O'Kane

Version: January 2022

Overview

- Motivation for Recurrent Neural Networks
- The main types of Recurrent Neural Networks
  - Simple RNNs
  - Layer of RNNs
  - Multiple Layers of RNNs
  - LSTMs
  - GRUs
- Examples of RNNs
  - Predicting a noisy sine wave
  - Trading example

Motivation for RNNs

- We've used neural networks to learn from labelled training sets
- But in many cases, the order of the data matters
- Word ordering matters - it affects our understanding of the sentence and as we process the meaning (output) that affects our understanding of what follows – it becomes an input
- The aim of Recurrent Neural Networks is to capture order-dependent information
- The hope is that it will improve the accuracy of our predictions
- What sort of problems are RNNs used for ?

RNNs are the Natural NN's to handle Sequences

- The iterative nature of RNNs makes them appropriate for capturing information in **sequences**
- They have been used in a range of applications:
  - Financial Time series analysis
  - Speech recognition
  - Language modelling
  - Translation
  - Image captioning
- They don't just have to be time sequences
- Some information that is not temporal, but order dependent can benefit e.g., an image can have pixels and their order is important

What is wrong with ordinary NNs ?

- We could pass in the sequence to the input layer of a NN

[5, 7, 9, 12, 18, 28]

- But how does the NN know that input 9 is before input 12 ?
- It can't know this as the input neurons to a NN are all treated in the same way – there is no spatial awareness
- It is not possible to use the architecture of a standard feedforward NN to tell the NN that the information is ordered
- And ordering is important – for example we may want the NN to predict the next term in this sequence !

[7 , 9 , 12, 18, 28, 40]

What else is wrong with ordinary NNs ?

- And what if the length of the series is 100 numbers long ?
- This may be too much for our NN to handle
- We also need something that is more economical
- Somehow we need the NN to have a "memory"
- It should remember recent events more than distant events
- How can we redesign the architecture of a NN to do this ?

Sequences vs Features

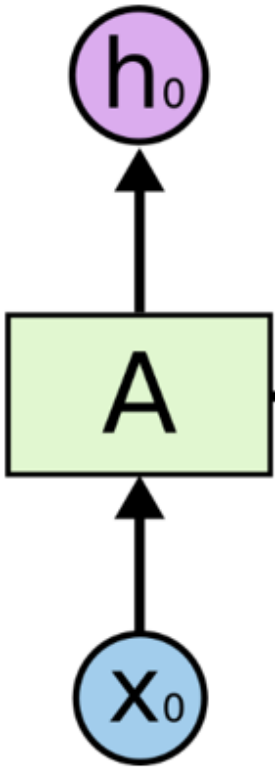
- Our training data comes in the form of sequences
- Unlike previous supervised training models, we don't have data with features and labels
- The labels are in the sequence
  - For example, we have a time series of stock prices
  - We use 10 stock prices to predict the 11^{th}
  - We have a rolling window of 10 stock prices

Simple RNN

The Feedforward Neuron

- So far our NNs are feedforward - activation flows in one direction
- An input X_0 –a number or a vector - passes through weights, is aggregated and passed via an activation function to give h_0

The Feedforward Neuron

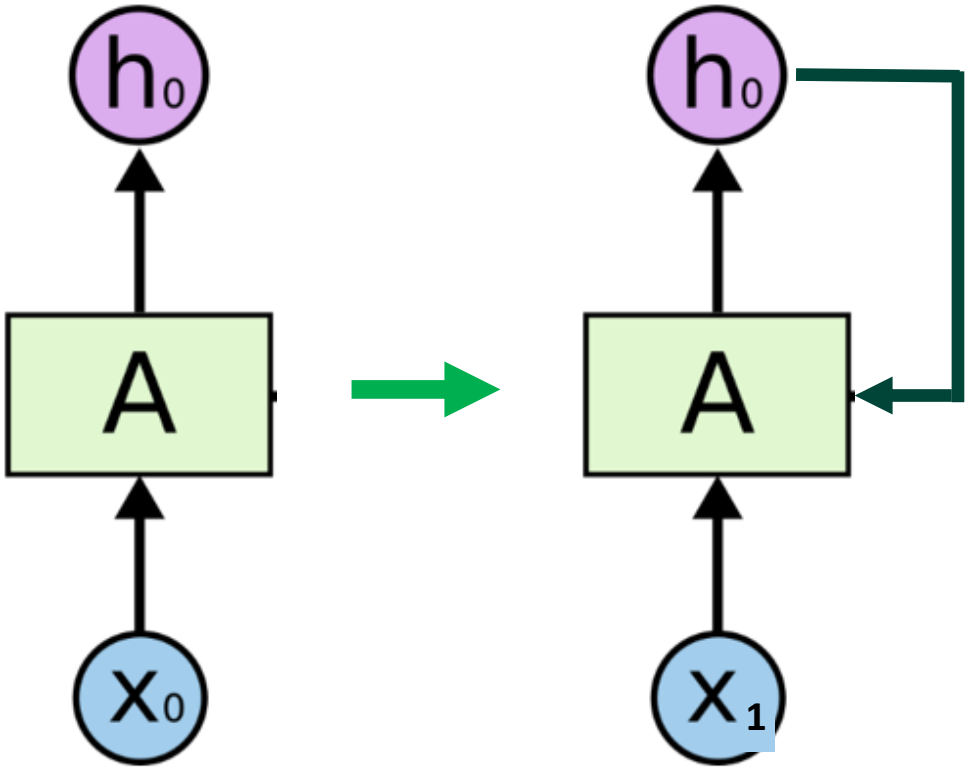


Initial input

X_0 combines with a weight vector and the aggregate z is passed into activation function A to output h_0

The Recurrent Neuron has an Output going Back

- Input X_0 passes through a set of weights, is aggregated and passed via an activation function to give h_0 which then goes back



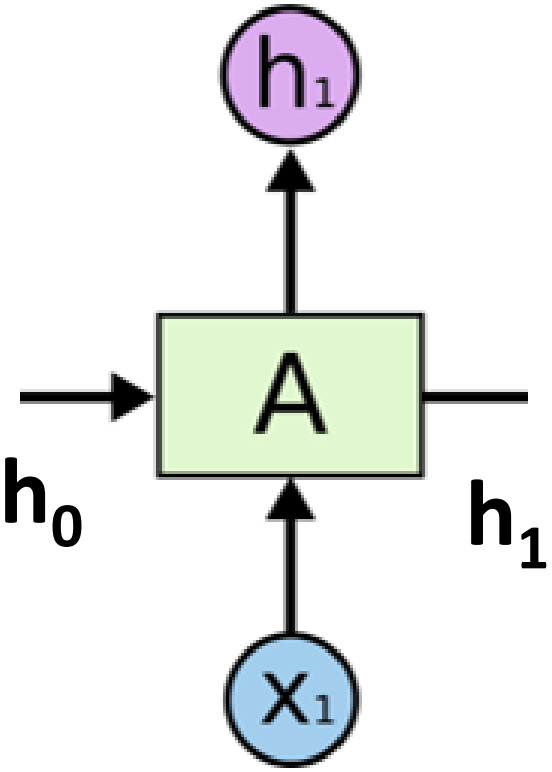
Step 1 **Step 2**

h_0 is the next output calculated by multiplying X_0 by a weight and adding a bias and it then goes back in along with the next input X_1

Inside the Single RNN cell after Each Time Step

- Any time after time zero there are **two inputs** to an RNN cell
  - First is x_t - the next sequence member
  - The other is h_{t-1}
  - This is the hidden state of the RNN cell
  - This was set on the previous iteration
- There is one output:
  - This is the value of h_t
  - It is calculated using weights w_x and w_h

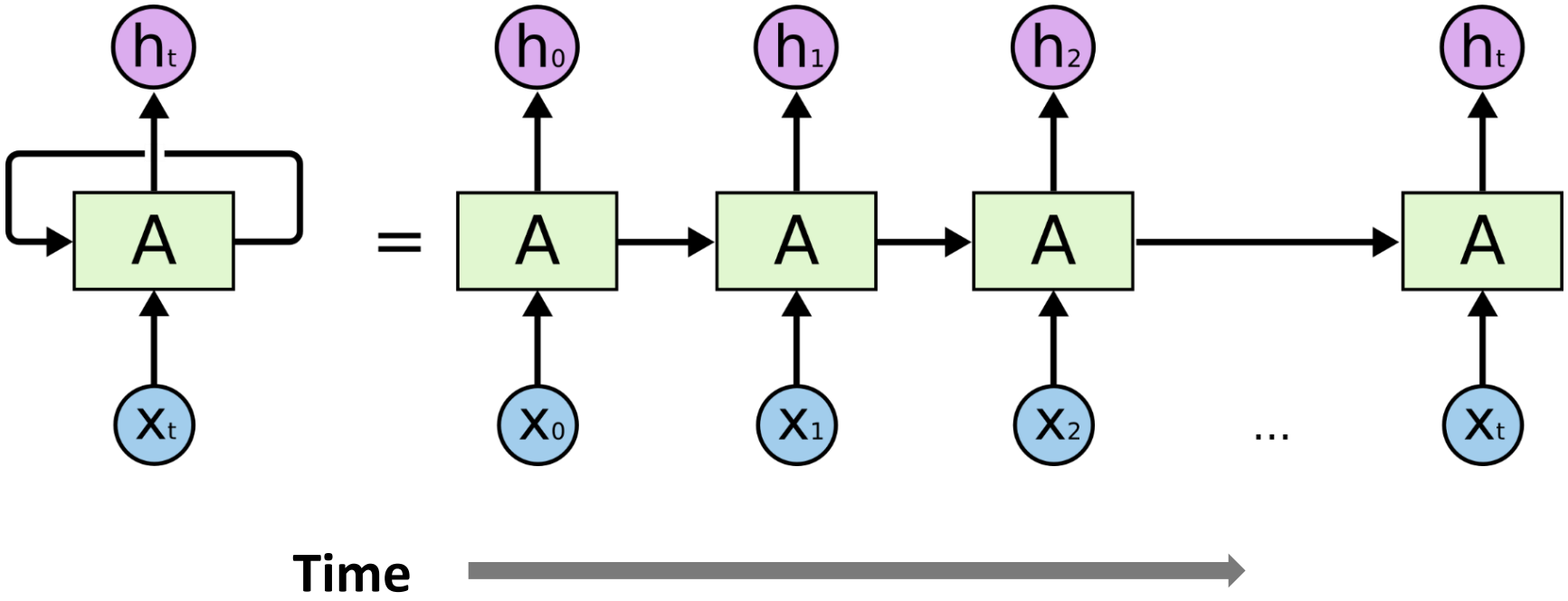
$$h_t = f\left(w_x x_t + w_h h_{t-1} + b\right)$$



- Here f is the activation function – usually a **tanh**
- Weights learn to the importance of past versus the present

A Simple RNN – A cell iterates through time

- An RNN cell iterates the inputs through time – we draw this as



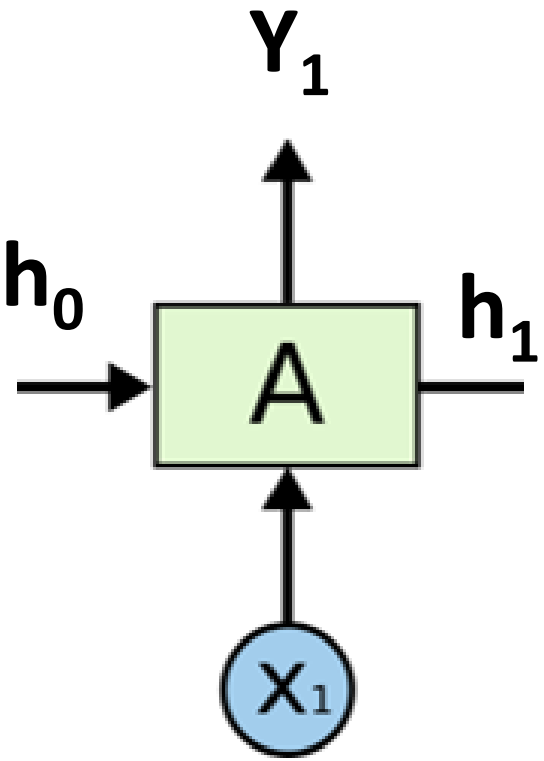
- This is the **same cell being self-iterated** or "unrolled"
- At the end it has generated a sequence h_0, h_1, …, h_t

Variations on the Single RNN cell

- In some RNN cells the output is not the same as the value of h - h is an "internal" state
- In this case the output is called y_{t+1}
- It is a function of h_{t+1} and weight w_y
- Often it passes through another function
- In this case we would have

$$h_t = f\left(w_x x_t + w_h h_{t-1} + b_h\right)$$

$$y_t = g\left(w_y h_t + b_y\right)$$



- For simplicity I will only consider an RNN with

$$y_t = h_t$$

The Single Layer RNN

- We can generalise the RNN architecture to a layer of n cells
- Each takes in a sequence of data X and outputs a sequence Y
- The same input value **x** is now a vector **X** applied to all of the cells
- The number of rows in each **W** is the number of cells
- The output of a layer is given by repeatedly unrolling using

$$\mathbf{y}_t = f\left(\mathbf{W}_x^T \cdot \mathbf{X}_t \cdot + \mathbf{W}_y^T \cdot \mathbf{h}_{t-1} + \mathbf{b}\right)$$

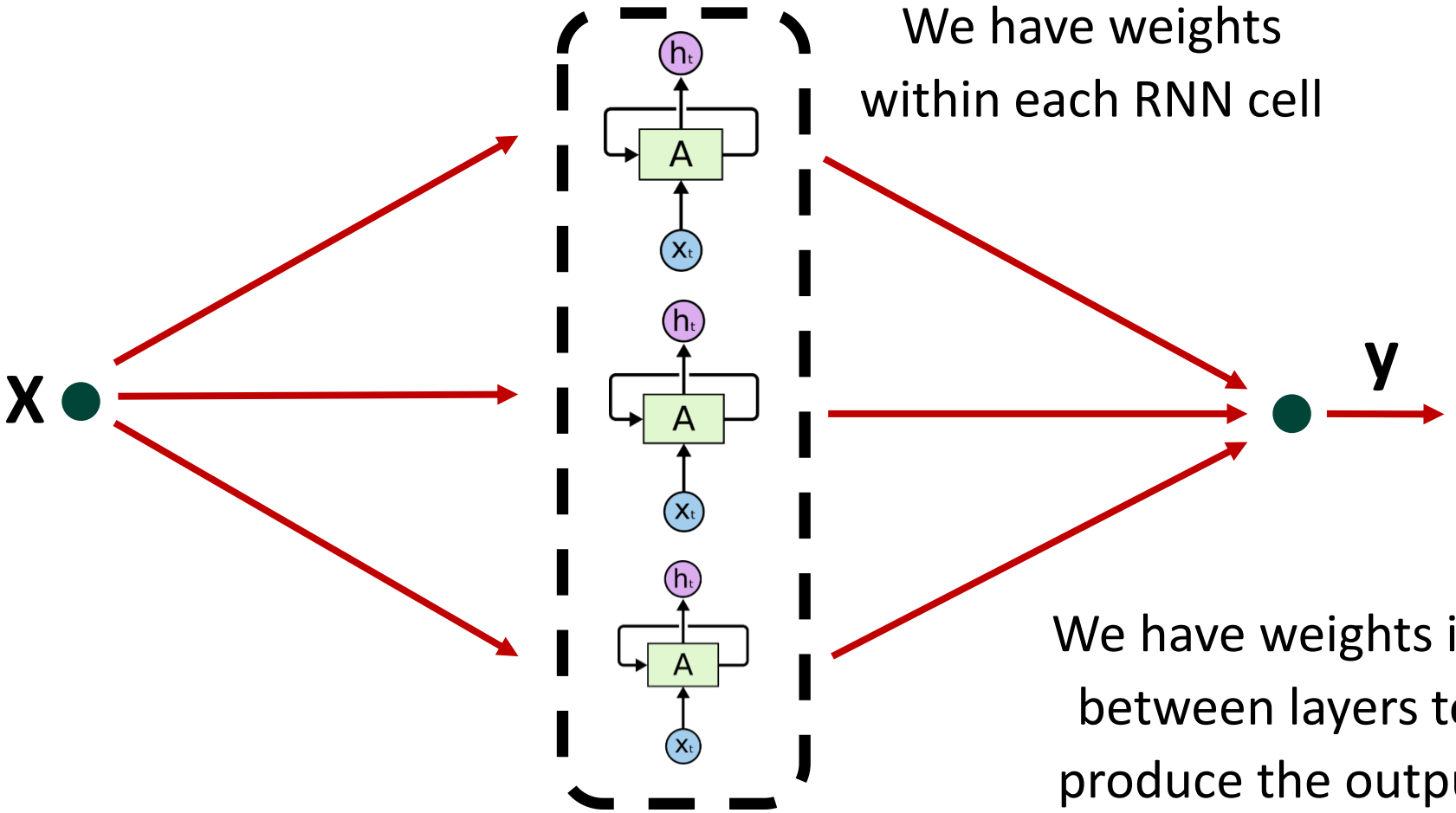
- At the end we have a sequence **y** output by the layer

Updating Mini-Batches in the Single Layer RNN

- We can even add a **mini-batch** dimension to X
- And hence this adds a dimension to the output sequence Y
- This means that each iteration of the Back propagation algorithm is over a subset of the training data
- We can do this operation in parallel on many data samples
- If it uses a GPU, this can be very efficient

A Layer of RNNs

- X is the input - The RNN layer unrolls – Y is the weighted output



The Multiple Layer RNN

- When we have multiple layers of RNN cells then we let each layer process the sequence and then output the resulting sequence
- This is then passed on to the next hidden RNN layer
- You might just want to output a single value to the output layer
- In TensorFlow you use the flag

return_sequences = True

to enforce this

Implementing our own RNN Layer in Python

- Consider the underlying Python (untested pseudo) code for an **RNNLayer** class with n RNN cells in the layer

```
class RNNLayer:

    def __init__(self, n, wx, wh):
        self.h = np.zeros(n)
        self.wx = wx
        self.wh = wh

    def step(self, x):
        # update the hidden state using a tanh activation function
        self.h = np.tanh(np.dot(self.wh, self.h) + np.dot(self.wx, x))
        # compute the output vector
        return self.h
```

Unrolling the RNN Layer over T time steps

- Consider unrolling the RNNLayer
- We have input vector x, and two weight vectors W_{hh} and W_{xh}
- We iterate the T time steps

```
h1 = rnn.step(x1)
h2 = rnn.step(x2)
…
hT = rnn.step(xT)
```

- Because the hidden state is held internally and used in each step to compute the next value of h, we have a "memory" cell
- It remembers the sequence and is aware that the input series X has a specific order
- This architecture is fundamentally different from feedforward NN

Training an RNN

- Training RNNs is simple
- Just think of each iteration of the RNN layer as a new layer
- We can therefore use Backpropagation

Batches and Time Series Data

- When we want to predict a time series

[5, 7, 9, 12, 18, 28]

- We need to split this and pass in the input sequence

[5, 7, 9, 12, 18]

- And then pass in the label for this which is [28]
- We will then train the RNN to predict this next value
- In practice we only train the RNN on a fixed length sequence
- Suppose we have a sequence length of 3 – our training data is

[5, 7, 9] -> [12]

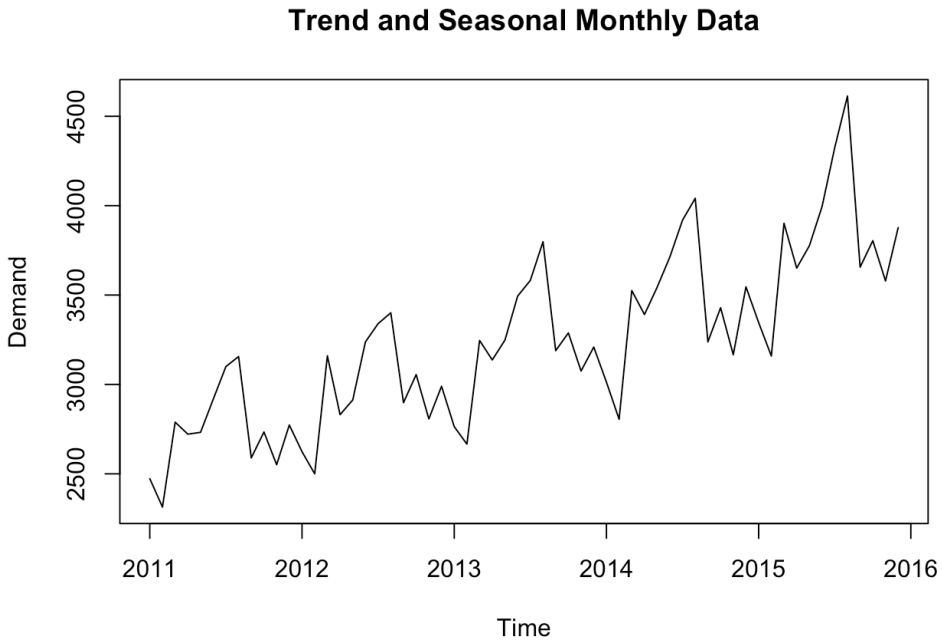
[7, 9, 12] -> [18]

[9, 12, 18] -> [28]

- How long should the sequence length be ?

The Sequence Length

- The **sequence length** should be determined by the data series
- It should be long enough to identify any patterns in the data



- If the data is seasonal – i.e. repeats annually, **then the sequence length should be longer than a year**
- You can make this a hyper-parameter

Predicting One Period into the Future

- We start by predicting one time step into the future
- Suppose our time series is

[5, 7, 9, 12, 18, 28]

- Let us use a sequence length of 3 periods to **predict one period ahead**
- Our RNN predicts

[5, 7, 9] -> [13]

- Then we predict the next period

[7, 9, 12] -> [19]

- Going out again one more period

[9, 12, 18] -> [27]

- We train the RNN network to close the prediction error

Predicting Multiple Periods into the Future

- We may decide to predict multiple periods into the future

[5, 7, 9, 12, 18, 28]

- Let's use a sequence length of 3 to predict 3 time periods ahead

[5, 7, 9] -> [13]

- Then we predict the next period using the previous prediction

[7, 9, 13] -> [20]

- Once again the prediction is not correct – made worse by the fact that we **are predicting based on a prediction!**
- Going out again one more period

[9, 13, 20] -> [30]

- We have predicted three periods ahead – a tougher task!
- We train the network to close the prediction error

Batch Sizes

- When supplying data to the network we need to organise the sequence data in batches
- **Backpropagation** is done for one batch at a time
- We need an input sequence and an output label for each training example in the batch
- Suppose our data series is **[5, 7, 9, 12, 18, 28, 40, 63]**
- A sequence length of 3 and a batch size of 4 would give the following as a single batch

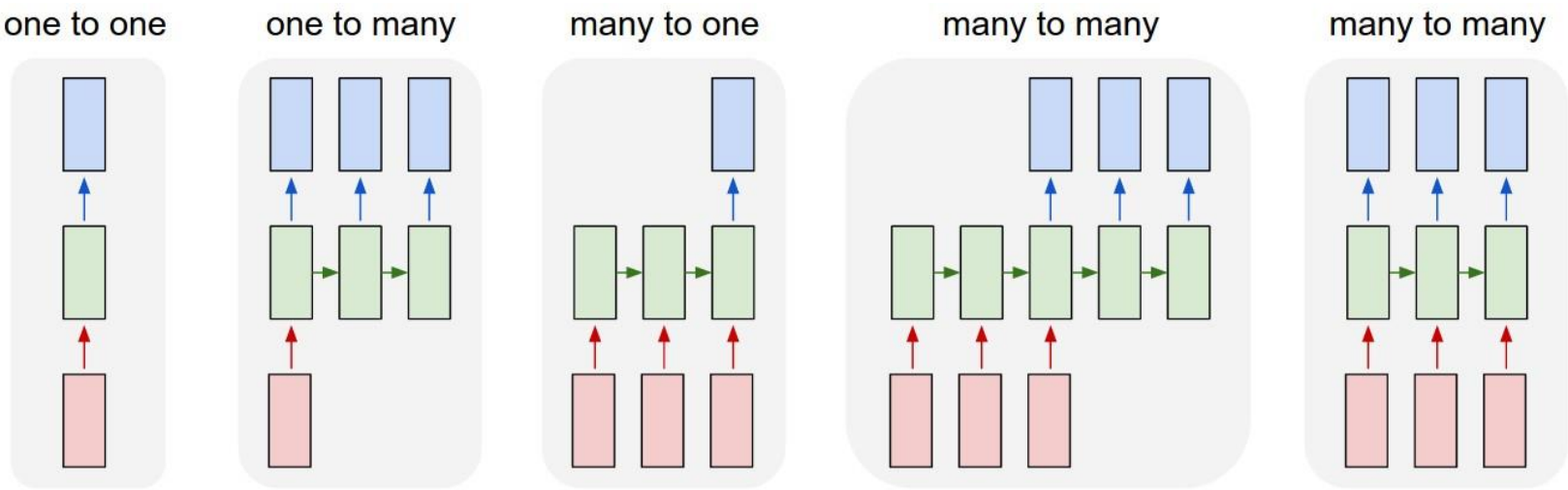
[5, 7, 9], [12]

[7, 9, 12], [18]

[9, 12, 18], [28]

[12, 18, 28], [40]

The RNN can take several Possible Structures



- Sequence to Sequence
  - Pass in a time series and get the shifted time series
- Sequence to Value or Vector
  - Pass in a series of words from sentence - get sentiment score
- Vector to Sequence
  - Pass in an image matrix or vector and get back a sequence

Case Study: Predicting a Noisy Sine Wave

Generating a Noisy Sine Wave

- I want to generate a single mini-batch of **batch_size** examples
- **n_steps** is the length of the sequence of each batch
- The NN will want an input matrix of size

n_batches x batch_size x n_steps

```
def generate_time_series(batch_size, n_steps):

            … see on next slide

    return …
```

- This function will return one batch
- So it returns a matrix of size **1 x batch_size x n_steps**

How to Generate a Noisy Sine Wave

```
def generate_time_series(batch_size, n_steps):
    freq1, freq2, offsets1, offsets2 = np.random.rand(4, batch_size, 1)
    time = np.linspace(0, 1, n_steps)
    series = 0.5 * np.sin((time - offsets1) * (freq1 * 10 + 10))  #   wave 1
    series += 0.2 * np.sin((time - offsets2) * (freq2 * 20 + 20)) # + wave 2
    series += 0.1 * (np.random.rand(batch_size, n_steps) - 0.5)   # + noise
    return series[..., np.newaxis].astype(np.float32)
```

- First generate 4 uniform randoms for frequencies and offsets
- Generate time vector with n_steps out to time 1 year
- Compute 2 sine functions using the different frequencies and offsets which we add using different weights
- We add on a noise using random uniforms zeroed at 0
- We add a dimension and convert the resulting series to float32

Calling the Noisy Sine Wave Generator

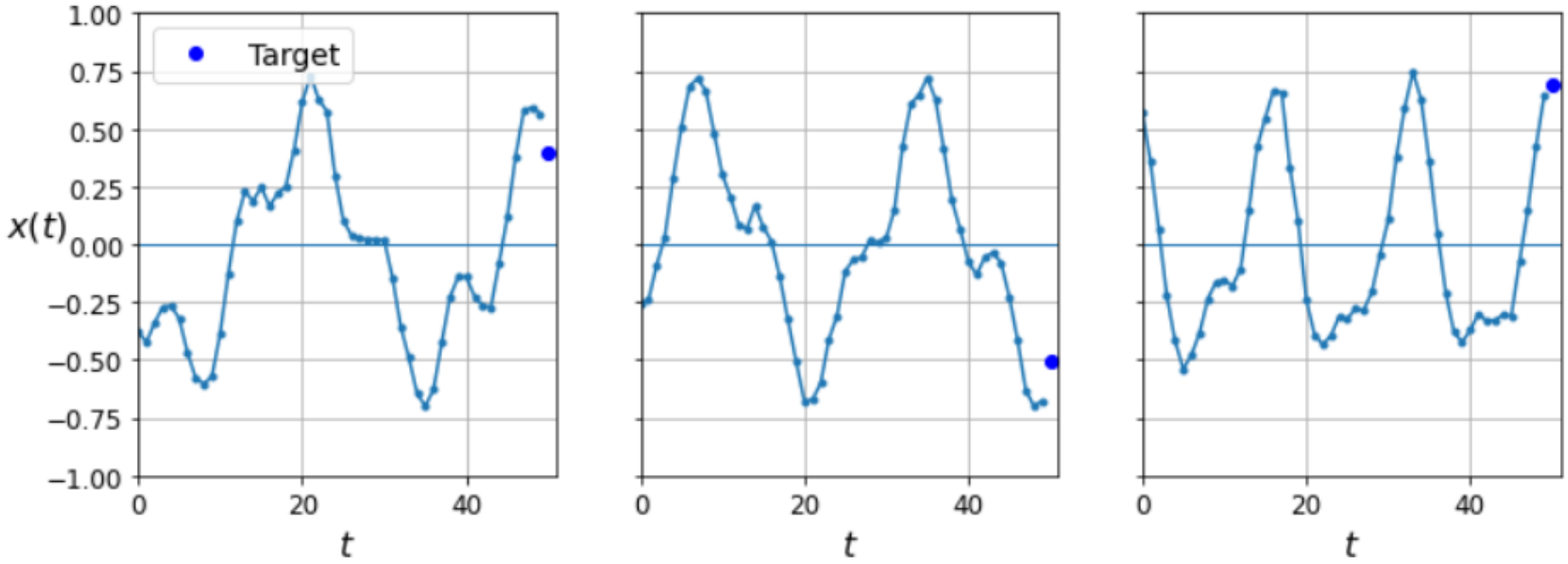
- We call the noisy sine wave single batch generator as follows

```
n_steps = 50
batch_size  = 10000
series = generate_time_series(batch_size, n_steps + 1)
X_train, y_train = series[:7000, :n_steps], series[:7000, -1]
X_valid, y_valid = series[7000:9000, :n_steps], series[7000:9000, -1]
X_test, y_test = series[9000:, :n_steps], series[9000:, -1]
```

- I generate **one** batch of 10K samples, each of 51 time-steps
- X_train has the first 50 elements of the sequence
- Y_train is the **final element** of the sequence – to be predicted
- I then split this into a training, validation and a test set
- The training, validation and test sets have 7K, 2K and 1K samples

The Time Series and the Prediction

- I am going to predict a noisy Sine Wave – see 3 examples below
- The blue dot is the target to be predicted by the RNN



- There is periodicity at different frequencies
- There is randomness
- Not an easy task !

I predict we will do the following:

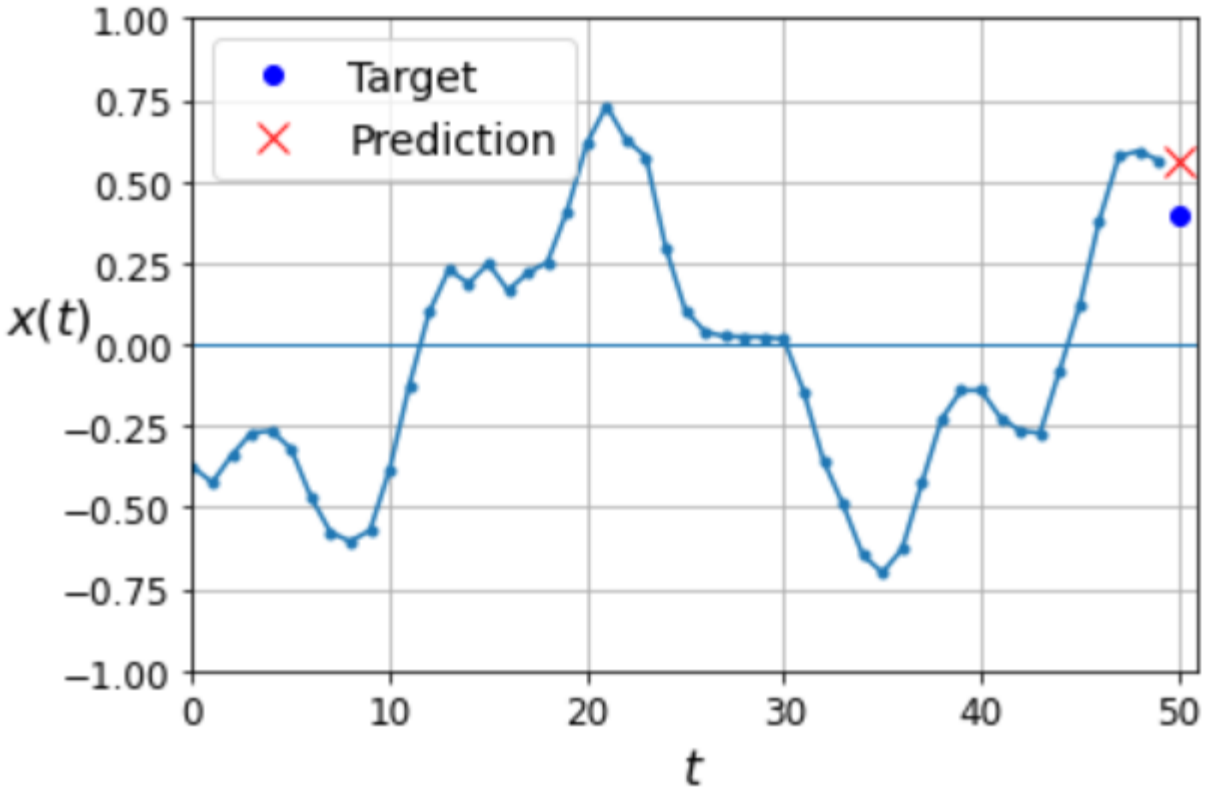
- Prediction of the next value
  - Use the previous value
  - Using linear deep learning feedforward ANN
  - Use a simple RNN to predict a sequence one value at a time
- Predict the entire sequence in one pass using an RNN
- Do this using an LSTM and then a GRU

First Approach – Predict using the Previous Value

- Calculate the mean squared error predicting using last value of x

```
from tensorflow import keras
y_pred = X_valid[:, -1]
np.mean(keras.losses.mean_squared_error(y_valid, y_pred))
```

```
0.020211367
```



Basic Feedforward NN as a Baseline

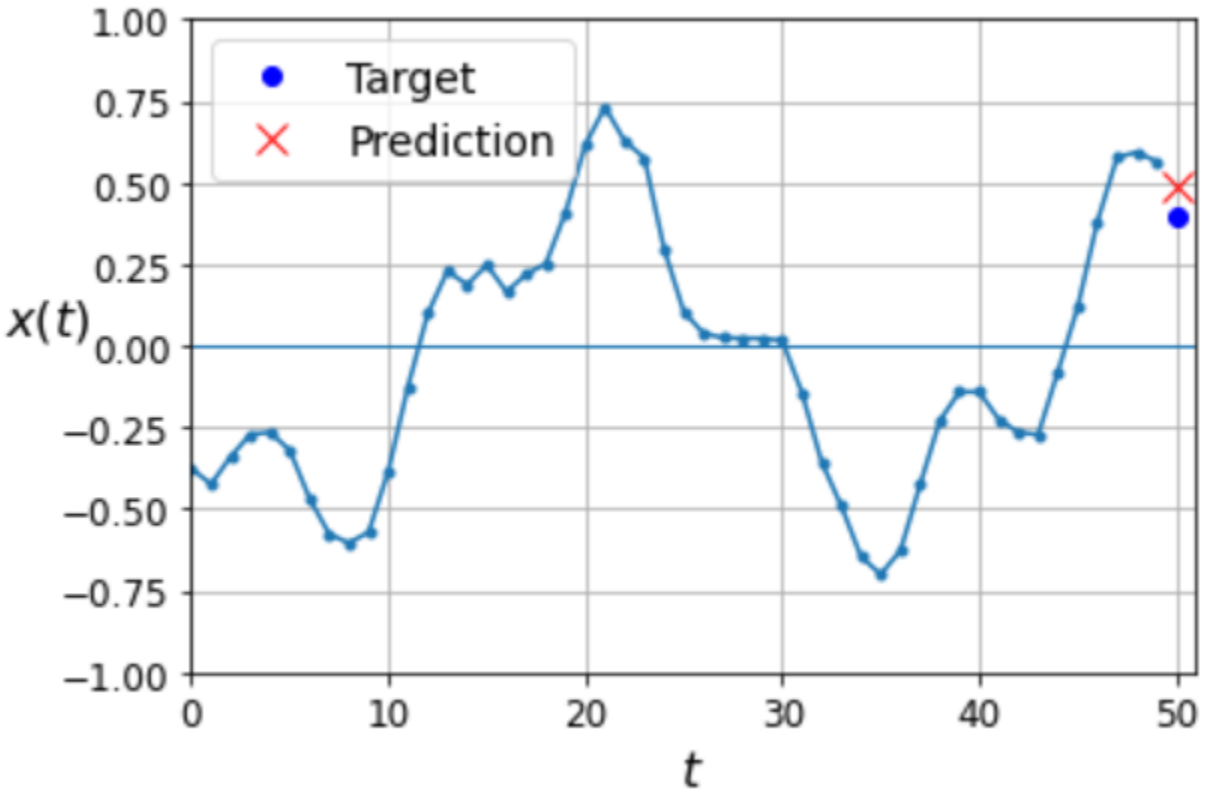
- We would like a reference NN to give us a sense of what is good
- We build a simple ANN with an input layer of 50 neurons
- This takes in the 50 historical prices
- I add in two layers of 50 neurons and a linear output neuron
- As it's a regression, I minimise the mean squared error

```
np.random.seed(42)
tf.random.set_seed(42)
model = keras.models.Sequential([
    keras.layers.Flatten(input_shape=[50, 1]),
    keras.layers.Dense(50),
    keras.layers.Dense(50),
    keras.layers.Dense(1)
])
model.summary()
model.compile(loss="mse", optimizer="adam")
```

- This model has 5,151 parameters

Simple Fully Connected NN

- The Fully Connected NN trains quickly
- After 20 epochs the prediction loss is 0.00306
- The prediction error for this example looks OK
- It fails to anticipate the steepness of the drop in the time series



What does this NN know about a sequence ?

- A feedforward NN cannot know that **X** is an ordered sequence
- What is sees is an input set of 50 features
- We know that these consists of the function values **x**
- It always sees x[0] at input 1 and … x[49] at input 50
- It can "learn" that the prediction is usually close to x[49]
- It also learns about the different shapes of this curve
- As the shape keeps changing, this is much harder than deterministic function fitting
- Making the architecture "understand" about sequences and order may give the ANN an extra edge
- **In an RNN the idea is fundamentally different – we have 1 feature which is a sequence of length 50**

We repeat this with a Deep RNN

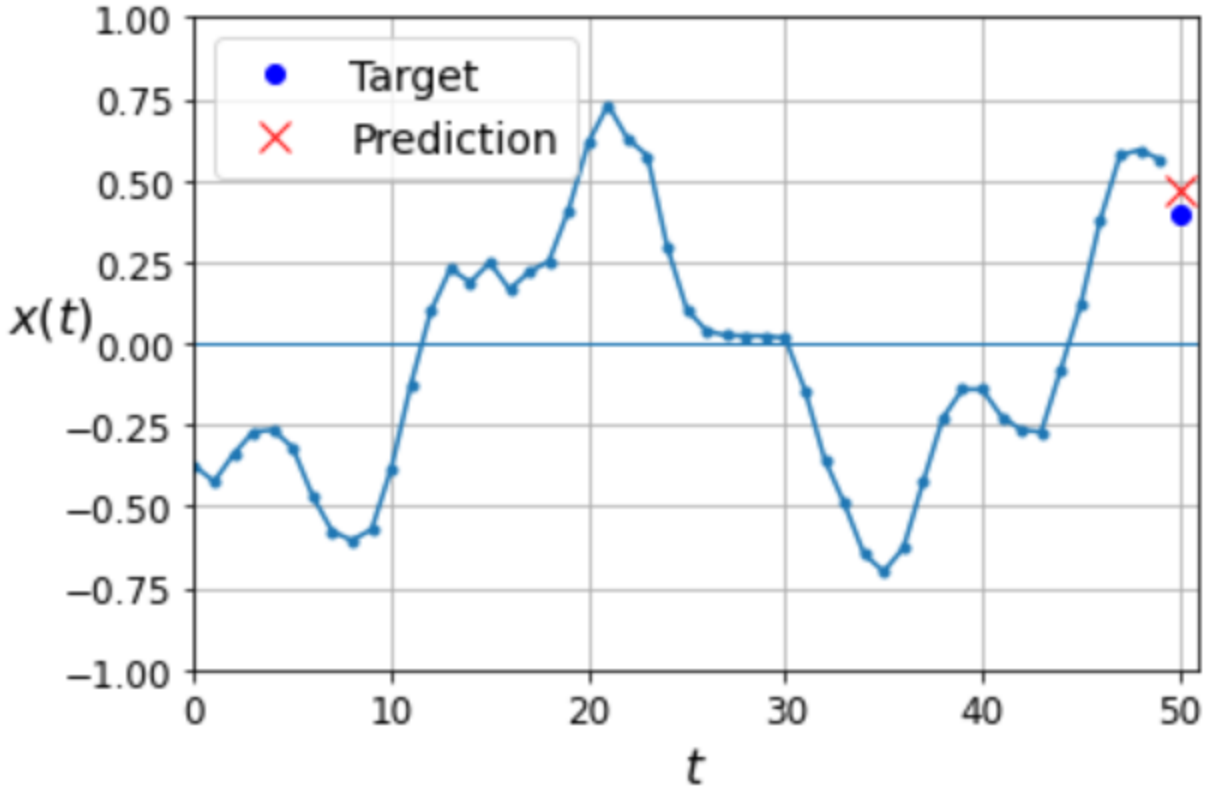
- We build a simple RNN with 2 hidden layers and 20 cells
- The input shape is a 2D array with an undefined number of rows and one column

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(1)
])
```

- The output is a simple neuron – we're only interested in 1 value
- It is trained over the 7K series in the training set
- Set **return_sequences = True** to pass time series to next layer
- This model has 1,281 parameters – fewer than the feedforward

Deep RNN Performance

- After 20 epochs the prediction loss is 0.0026 – it's even better!
- The prediction error for this example looks much better too



- We did this with fewer parameters

Predicting Several Periods Ahead

- Up until now we only look one period ahead
- Now we want to see if we can use the memory of the RNN to predict several periods ahead
- We can always take the models we have trained and ask them to do it. The algorithm would be:
  1. We predict the next value
  2. We add the next value onto the series
  3. We use the model to compute the next value
  4. Go to 2 until we have gone far enough forward
- This might work but it might not be as good as actually training it to do this specifically

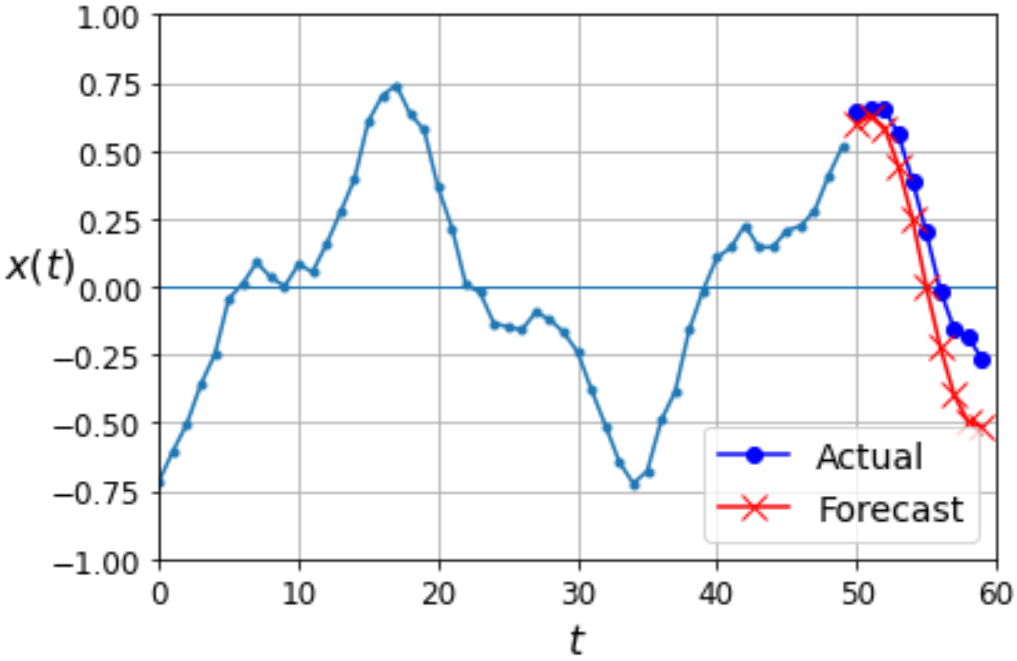
One-Period Model Predicting Multiple Periods

- See how well our simple deep RNN predicts multiple periods
- First we generate a series **with 10 extra values on the end**
- We loop over 10 time steps and use the existing one period model to predict the next value
- We then add the prediction to the time series and use that for the next period

```
series = generate_time_series(1, n_steps + 10)
X_new, Y_new = series[:, :n_steps], series[:, n_steps:]
X = X_new
for step_ahead in range(10):
    y_pred_one = model.predict(X[:, step_ahead:])[:, np.newaxis, :]
    X = np.concatenate([X, y_pred_one], axis=1)

Y_pred = X[:, n_steps:]
```

One Period Model – Multi-Period Prediction Results



- The result looks OK – the downward gradient is monotonic
- It overshoots a bit and then slows down correctly
- The error increases with time
- **Expected as we only trained it to learn one time step !!**

We generate the Data with 10 Extra Points

- This is the same as before, but we go out 10 extra points

```
np.random.seed(42)

n_steps = 50
series = generate_time_series(10000, n_steps + 10)
X_train, Y_train = series[:7000, :n_steps], series[:7000, -10:, 0]
X_valid, Y_valid = series[7000:9000, :n_steps], series[7000:9000, -10:, 0]
X_test, Y_test = series[9000:, :n_steps], series[9000:, -10:, 0]
```

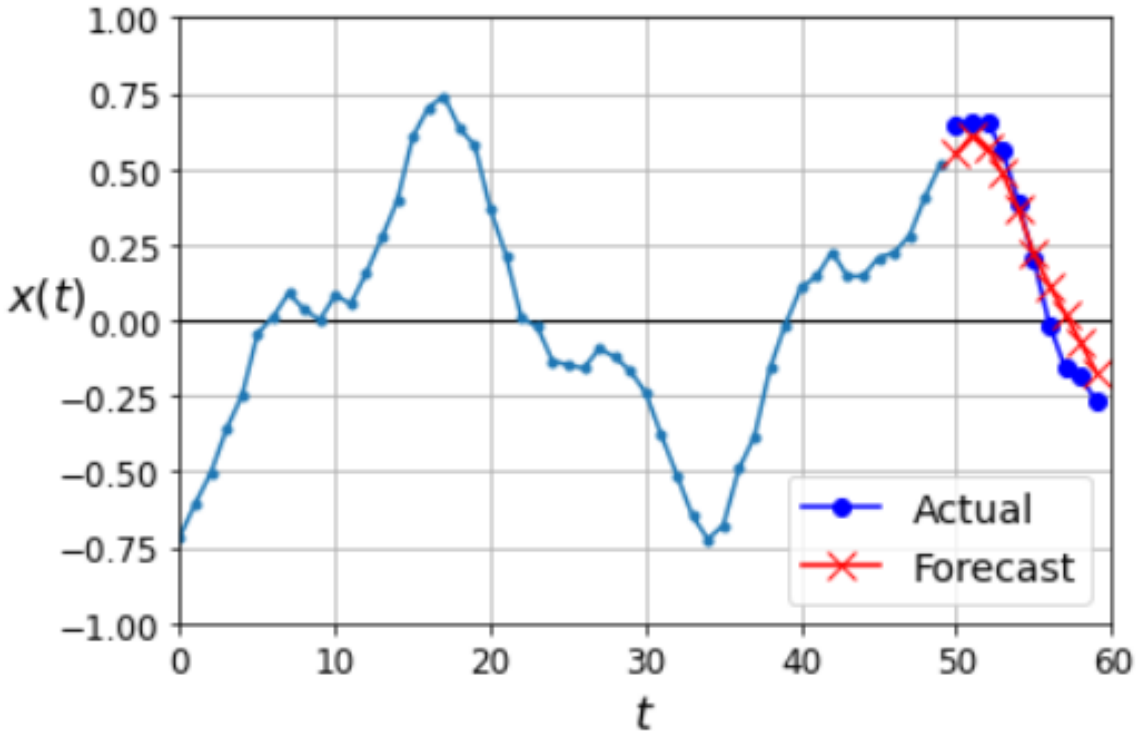
- We build a deep RNN with 10 outputs

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True,
                           input_shape=[None, 1]),
    keras.layers.SimpleRNN(20),
    keras.layers.Dense(10)
])
```

- The model is trained to match the 10 neurons in the output layer to the next 10 sequences values

Multi-Period Prediction Results

- The quality of the prediction looks better than the single-period model – it has the turning point and a similar slope and the error does not noticeably diverge with the number of periods



- But this is still not yet a sequence-to-sequence model

Sequence to Sequence Model

- Want a model that predicts next 10 time periods at each period
  - It forecasts periods 1-10 at period 0
  - …
  - It forecasts periods 50-59 at period 49
- The input X data is unchanged

```
n_steps = 50
series = generate_time_series(10000, n_steps + 10)
X_train = series[:7000, :n_steps]
X_valid = series[7000:9000, :n_steps]
X_test = series[9000:, :n_steps]
```

Generating the Label Data

- The Y data is generated as the following 10 periods

```
Y = np.empty((10000, n_steps, 10))
for step_ahead in range(1, 10 + 1):
    Y[..., step_ahead - 1] = series[..., step_ahead:step_ahead + n_steps, 0]
Y_train = Y[:7000]
Y_valid = Y[7000:9000]
Y_test = Y[9000:]
```

- To hold the training data, we create an array for the 10K time series, each with 50 vectors (for each time period) with each of length 10 (the prediction horizon)
- We loop over the prediction horizon (out to 10 time-periods)
- At each step_ahead we assign the corresponding series of values which start at the next time period and end 50 periods later

The Sequence-Sequence Deep RNN

- NN must output a sequence so set **return_sequences = True**
- The output layer needs to handle a sequence of values with a Dense layer for each – use a special Keras **TimeDistributed** layer

```
model = keras.models.Sequential([
    keras.layers.SimpleRNN(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.SimpleRNN(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])

model.summary()
```

```
Model: "sequential_14"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 simple_rnn_27 (SimpleRNN)   (None, None, 20)          440

 simple_rnn_28 (SimpleRNN)   (None, None, 20)          820

 time_distributed_2 (TimeDis  (None, None, 10)         210
 tributed)
```

Time Distributed Layer

- We have an RNN outputting the entire sequence of the sequence h after it has iterated through all the inputs it has received
- The output of each layer is the entire sequence – when **return_sequence = False**, only the final value is returned
- We then want to pass each element of the series through a Dense layer before outputting a single value
- In the previous example we see that it has 210 parameters
- There are 20 RNN neurons inputting to this layer
- Each is passing to 10 output neurons
- Hence the number of parameters is (20 + 1) x 10 = 210
- The same weights are applied to each element of the sequence - **TimeDistributed** is simply adding a loop over the time series

Evaluation Cost Function

- In the model the cost function used to perform back propagation is the mean squared error
- It looks across the sequence of predictions and the sequence of known values and calculates their MSE
- However, for prediction purposes, the value that interests us most is the error of the 10-step prediction
- We therefore define a custom metric that is reported during training **even if it is not the objective function**

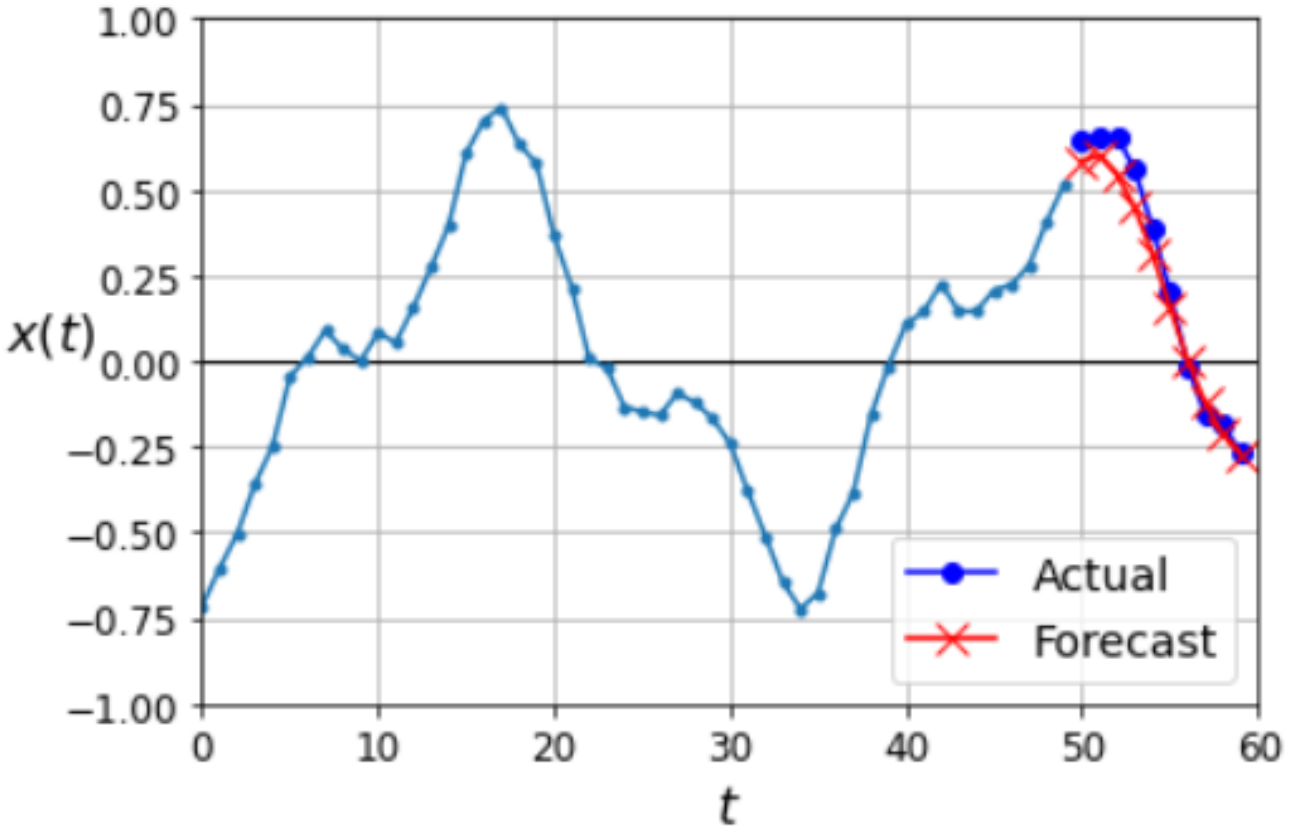
```
model.compile(loss="mse", optimizer=keras.optimizers.Adam(learning_rate=0.01),
              metrics=[last_time_step_mse])
```

- It is defined as

```
def last_time_step_mse(Y_true, Y_pred):
    return keras.metrics.mean_squared_error(Y_true[:, -1], Y_pred[:, -1])
```

Sequence to Sequence Results

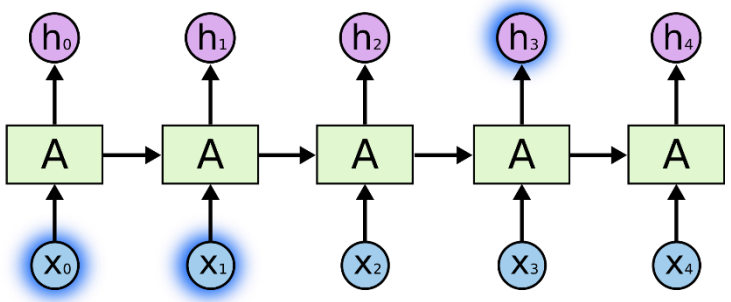
- The resulting prediction is as follows
- Looks good ! The final period MSE was found to be 0.0085



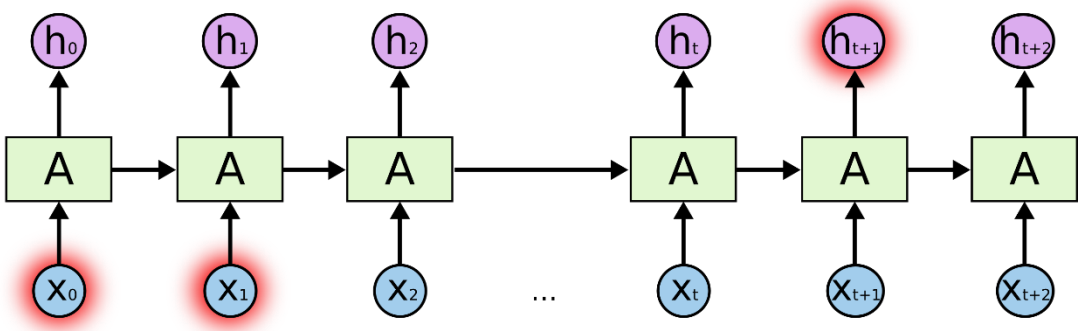
Long Short Term Memory Cells (LSTMs)

Long-Term Dependencies

- It is important for the history of the sequence to be remembered
- For example, if we are predicting words, then a word earlier in the same sentence may help us to predict the next word



- But if the word was 2-3 sentences ago then RNNs find it difficult to retain that information



Some problems with RNNs

- The vanishing gradients problems means that many iterations of the RNN will make it more likely that the neurons saturate
- This makes training hard when using backpropagation
- This limits the length of the memory
- A related issue is simply that early memories are overwritten by more recent memories
- This is where the LSTM comes in

Why we need LSTMs

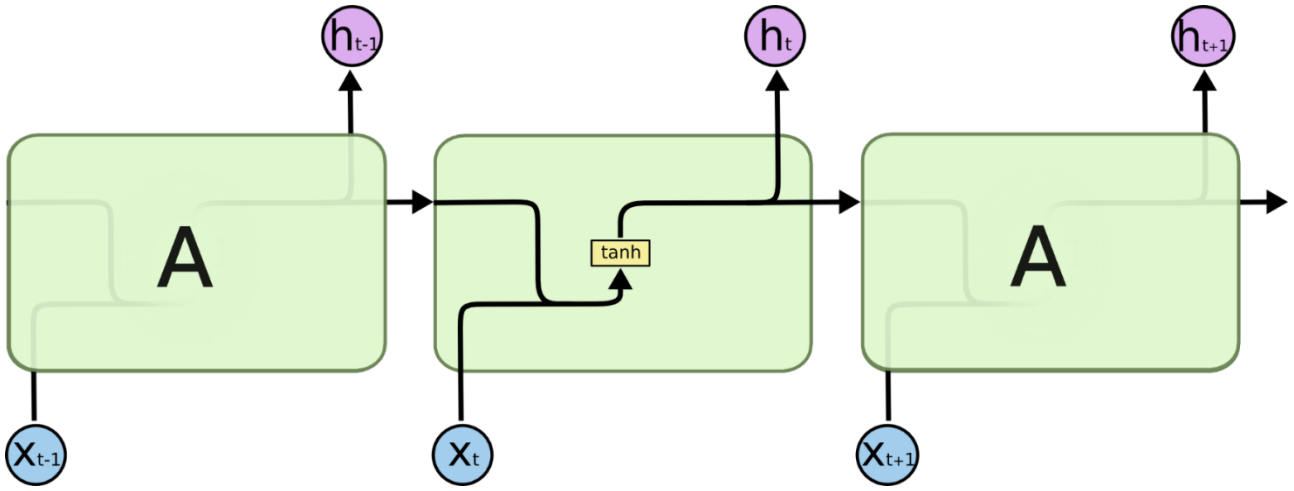
- If we have a long sequence and need a long chain of RNNs, we start to get problems with back propagation
- For sigmoidal activations we get derivatives tending to zero
- The accumulation of the chain rule of many products of small derivatives means that we end up with vanishing gradients
- We can also end up with exploding gradients – if the gradient is large and we multiply many of them, the result can explode
- Both problems prevent the NN from learning
- We can overcome this problem using
  - ReLu Activation functions
  - Gradient clipping
  - Long Short-Term Memory units (LSTMs)

Long Short-Term Memory (LSTM)

- An LSTM is an artificial recurrent neural network (RNN)
- As the name suggests, it has a long and a short-term memory
- LSTMs are good at making predictions based on lengthy time-series, e.g., market prices, sequences of words or game actions
- LSTMs are quite complex to understand as they are composed of a cell, an input gate, an output gate and a forget gate
- They were developed to solve the vanishing gradients problem
- LSTMs were used in the core of the NN developed by DeepMind's AlphaStar to play Starcraft II in late 2018, beating a top pro.

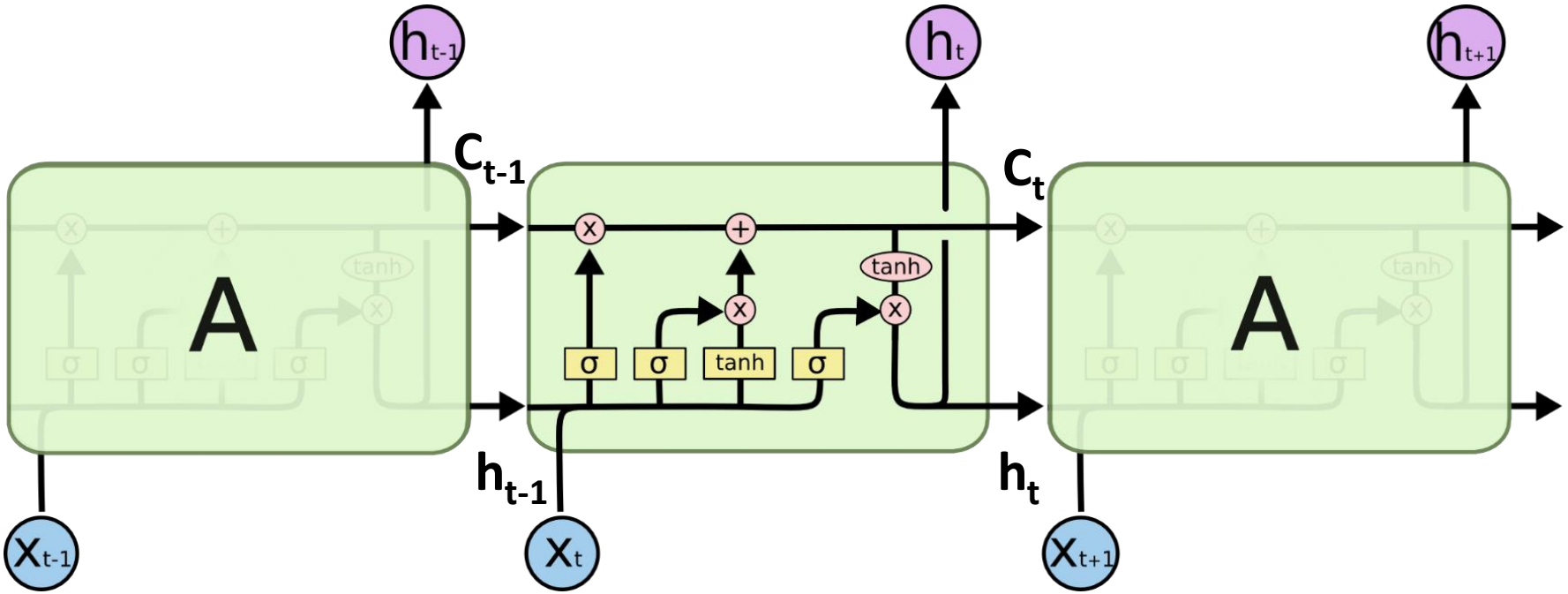
The LSTM Chain Solves this Problem

- From outside the LSTM looks just the same as an RNN chain
- As before an input x enters an output h appears



- However, internally the LSTM is far more complicated
- It seeks to maintain an internal short and a long term memory
- The long term memory carries information from before
- Gates (multiplying values by a number 0-1) control what is kept

Inside an LSTM



- At time step t, an LSTM cell has 1 inputs, 1 output and 2 states
  - **Input:** X_t
  - **Output**: h_t
  - **States:** LT memory C_{t-1}, ST memory h_{t-1}
- Gates control how much of the LT memory is kept/forgotten

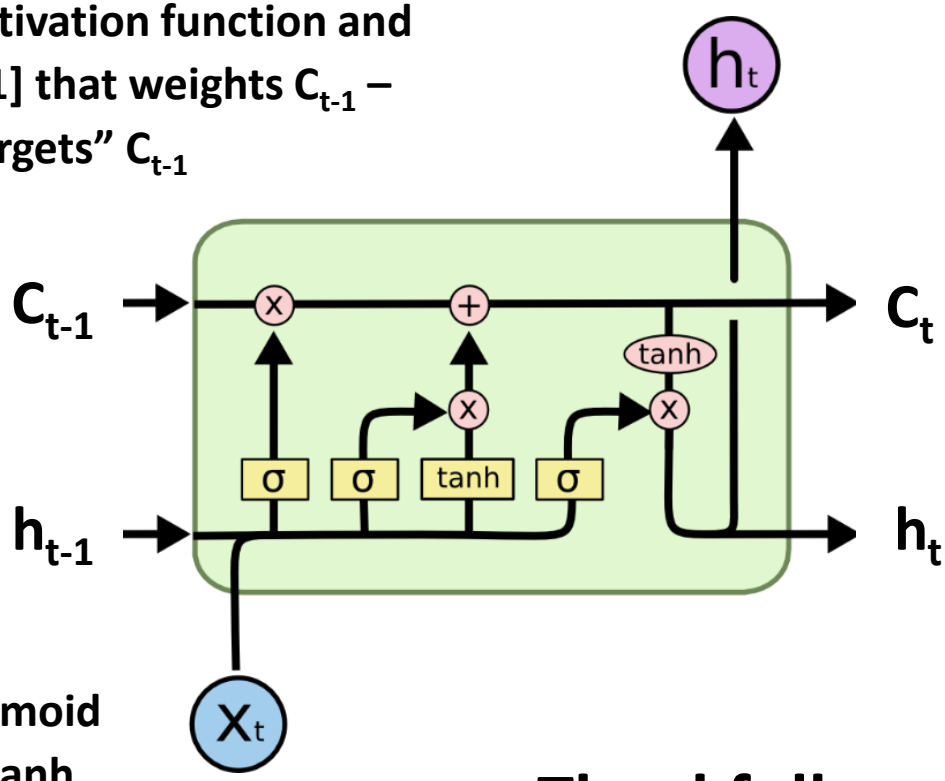
The LSTM Mechanics

Step 1: FORGET GATE

h_{t-1} and x_t combine via weights, pass through a sigmoid activation function and output a number [0,1] that weights C_{t-1} – a value close to 0 "forgets" C_{t-1}

Step 3: OUTPUT GATE

h_{t-1} goes through a sigmoid (with weights) and multiplies the new LT state value after passing through a tanh (via weights) and this is the new ST output (hidden) state h_t



Step 2: UPDATE GATE

h_{t-1} goes through a sigmoid (with weights) and a tanh (via weights) and they multiply and update the new LT cell state C_t

Thankfully we don't have to code this up ourselves – it's all done in TensorFlow ;-)

We Repeat Sine Wave but with an LSTM

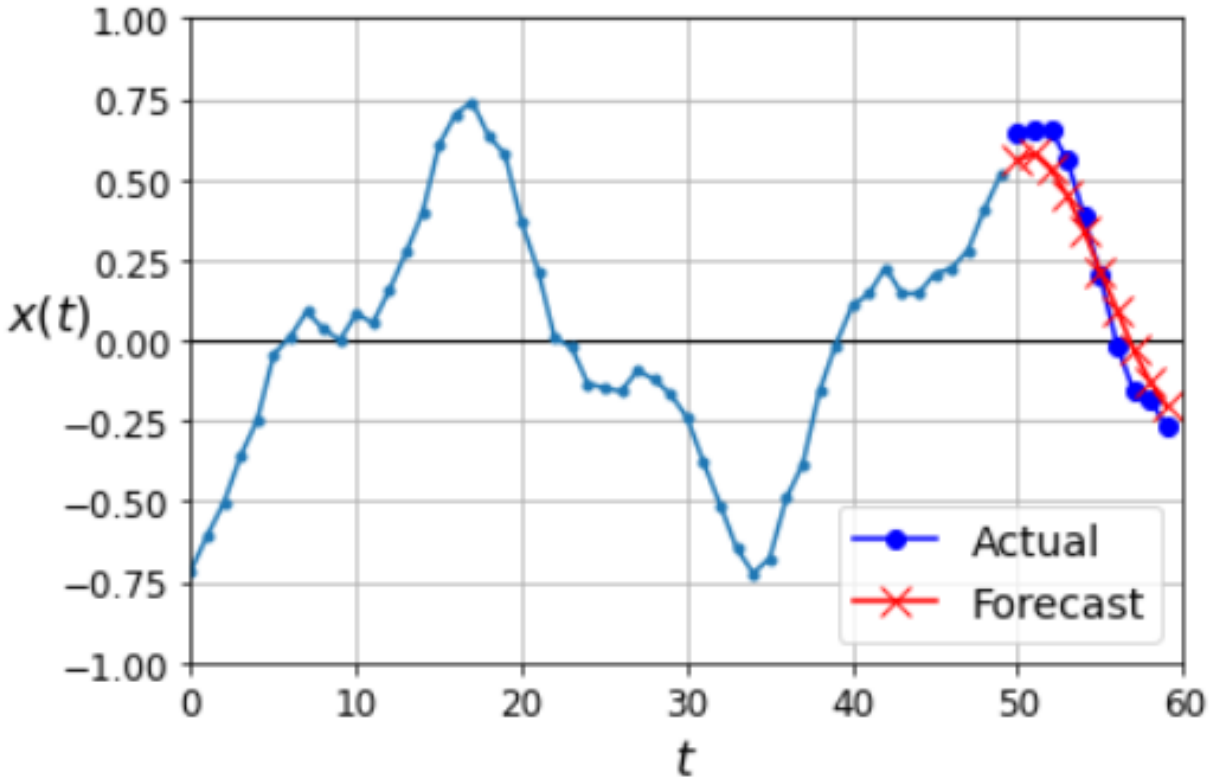
- We build a deep LSTM network
- You should be very familiar with the syntax now !

```
model = keras.models.Sequential([
    keras.layers.LSTM(20, return_sequences=True, input_shape=[None, 1]),
    keras.layers.LSTM(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

- It has 20 LSTM cells in 2 hidden layers
- Once again we are doing sequence to sequence so each layer has return_sequences = True
- The output layer is a TimeDistributed layer wrapping a Dense layer with 10 outputs

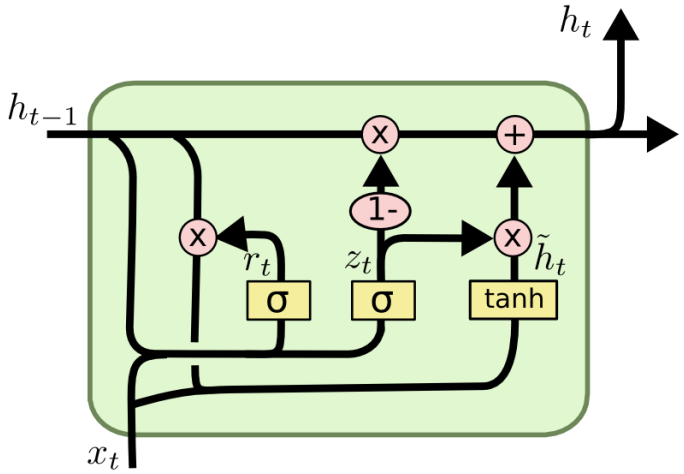
Deep LSTM

- The Deep LSTM does well and has almost identical results to the Deep Sequence to Sequence RNN
- Its final period MSE is 0.0086



The Architecture of the GRU

- Introduced by Cho et al (2014)



$$z_t = \sigma\left(W_z \cdot [h_{t-1}, x_t]\right)$$

$$r_t = \sigma\left(W_r \cdot [h_{t-1}, x_t]\right)$$

$$\tilde{h}_t = \tanh\left(W \cdot [r_t * h_{t-1}, x_t]\right)$$

$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$

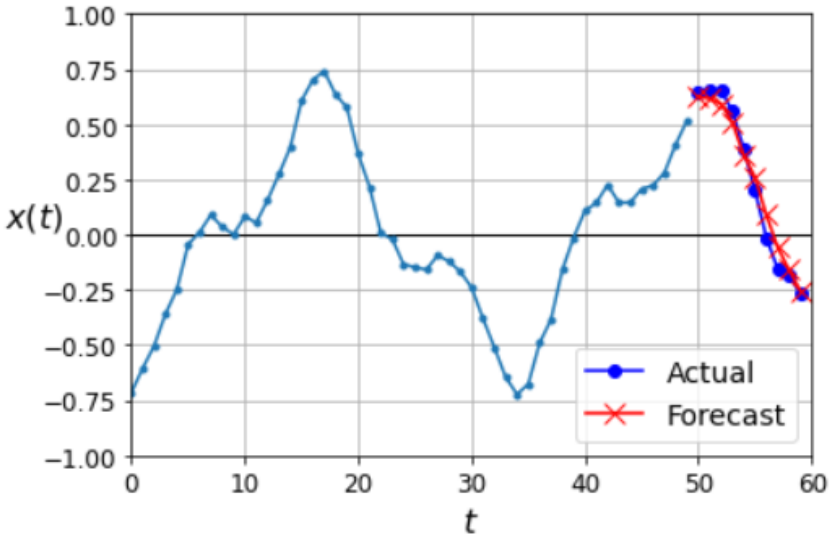
- Combines the input and forget gates into a single update gate
- This makes it simpler than the LSTM
- Growing in popularity although results are similar

The Special Layer

- At the end of my notebook, I showed a new layer
- It is a **Convolution Layer**, and you will learn about it next week

```
model = keras.models.Sequential([
    keras.layers.Conv1D(filters=20, kernel_size=4, strides=2,
                        padding="valid", input_shape=[None,1]),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.GRU(20, return_sequences=True),
    keras.layers.TimeDistributed(keras.layers.Dense(10))
])
```

- It works well !



RNNs versus Time Series Models

- Question is if a traditional time series model would do better
- A 2020 paper (Muncharaz) tested this using the S&P 500 found

Abstract

In the financial literature, there is great interest in the prediction of stock prices. Stock prediction is necessary for the creation of different investment strategies, both speculative and hedging ones. The application of neural networks has involved a change in the creation of predictive models. In this paper, we analyze the capacity of recurrent neural networks, in particular the long short-term recurrent neural network (LSTM) as opposed to classic time series models such as the Exponential Smooth Time Series (ETS) and the Arima model (ARIMA). These models have been estimated for 284 stocks from the S&P 500 stock market index, comparing the MAE obtained from their predictions. The results obtained confirm a significant reduction in prediction errors when LSTM is applied. These results are consistent with other similar studies applied to stocks included in other stock market indices, as well as other financial assets such as exchange rates.

- https://hal.archives-ouvertes.fr/hal-03149342/document
- I think this is possible – ML goes beyond the simple linear relationships of traditional linear regression based models
- You will only know if you do careful cross-validation

NLP with RNNs

Use of RNNs in NLP

- I have shown how RNNs are used to predict sequences
- But there are other sorts of sequences that RNNs can model
- Language is a sequence of words - a sequence of characters
- Hence RNN in the field of NLP is a common approach
- Andrew Karpathy who works at Tesla wrote a very famous blog in which he showed how to use RNN to predict Shakespeare
- He even made it write Shakespeare !
- I will take you through this if I have time.

Test Case: Predicting Seasonal Data

Predict Clothing and Clothing Accessory Stores

- We predict a seasonal time series from FRED

We Load the Data from FRED and Remove 2020

- The data can be easily imported

```
import pandas_datareader.data as web
df = web.DataReader("MRTSSM448USN", 'fred')
```

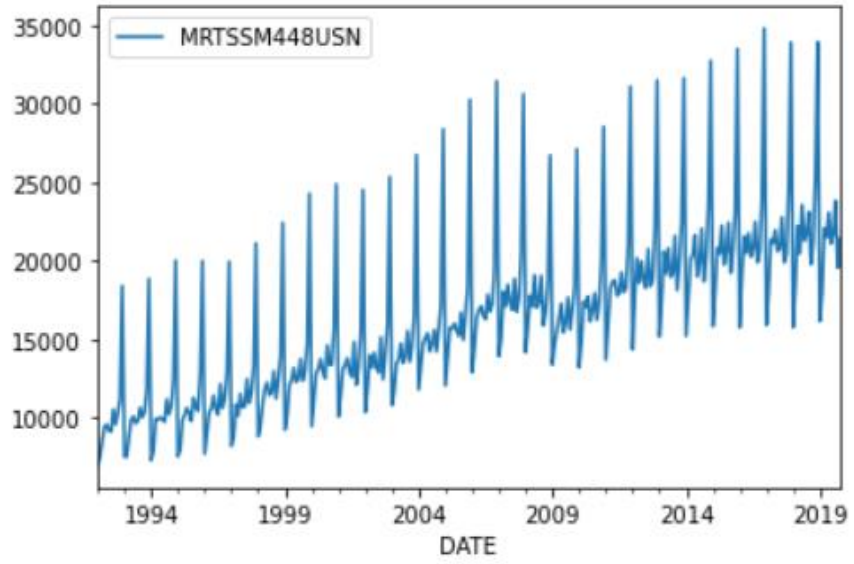
```
df.head(5)
```

DATE	MRTSSM448USN
2017-03-01	21332
2017-04-01	21143
2017-05-01	21945
2017-06-01	20618
2017-07-01	20795

```
df = df[0:334]
```

Let's look at the data series we want to predict

```
df.plot();
```



- It has monthly sales values starting in March 2017
- We will cut off 2020 – predicting Covid is too complicated !

Preparing the Data

- We use the last 18 months as test data
- We also do a min-max scaler – we calculate the scaling on the training data and apply it to both training and test data

```
n_train = df.shape[0] - 18
```

```
df_train = df[0:n_train]
df_test  = df[n_train:]
```

```
len(df_train), len(df_test)
```

```
(39, 18)
```

```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
scaler.fit(df_train)
```

```
MinMaxScaler()
```

```
df_train_scaled = scaler.transform(df_train)
df_test_scaled = scaler.transform(df_test)
```

Time Series Generator

- To manage the time series and batches, I use the Keras TimeseriesGenerator object
- Look in the Keras documentation to see how this works
- I also set a batch length of 12 – fine as it's less than the test set of 18 months

```
from tensorflow.keras.preprocessing.sequence import TimeseriesGenerator
```

```
batch_length = 12 # Must be less than the size of the test set
batch_size = 1 # Number of batches in each training iteration

generator = TimeseriesGenerator(df_train_scaled,
                                df_train_scaled,
                                length=batch_length,
                                batch_size=batch_size)
```

Build an LSTM Model

- The architecture is very familiar by now – one neuron in the output layer

```
n_features = 1 # it's a time series
```

```
model = Sequential()
model.add(LSTM(100, activation="relu", input_shape=(batch_length, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

```
model.summary()
```

```
Model: "sequential_8"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 lstm_7 (LSTM)               (None, 100)               40800

 dense_7 (Dense)             (None, 1)                 101

=================================================================
Total params: 40,901
Trainable params: 40,901
Non-trainable params: 0
```

Early Stopping

- We would like the training to stop if it is not improving
- We can do this using an EarlyStopping callback
- This stops if a defined metric does not improve

```
from tensorflow.keras.callbacks import EarlyStopping
early_stop = EarlyStopping(monitor="val_loss", patience = 2)
```

- Here the training stops if the validation loss does not improve after 2 epochs of training, and we apply it as follows

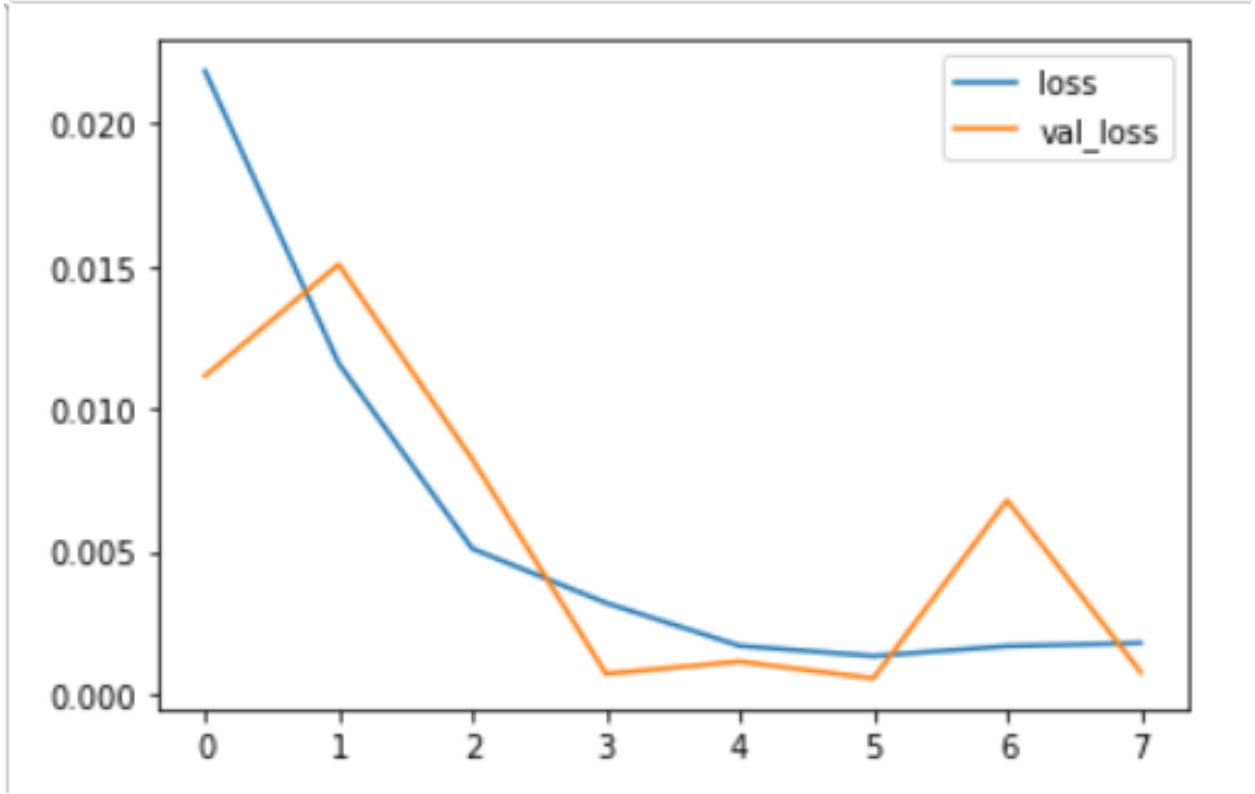
```
model.fit(generator, epochs=20,
          validation_data = validation_generator,
          callbacks = [early_stop])
```

```
Epoch 1/20
304/304 [==============================] - 2s 3ms/step - loss: 0.0249 - val_loss: 0.0130
Epoch 2/20
304/304 [==============================] - 1s 2ms/step - loss: 0.0167 - val_loss: 0.0049
Epoch 3/20
304/304 [==============================] - 1s 2ms/step - loss: 0.0081 - val_loss: 0.0027
```

Training Results

```
losses = pd.DataFrame(model.history.history)
```

```
losses.plot();
```



Prediction Results

- Loop over the test set of 18 months
- Predict the next value
- Add it to the test predictions vector
- Update the next batch with this next prediction and drop first

```
test_predictions = []
first_eval_batch = df_train_scaled[-batch_length:]
current_batch = first_eval_batch.reshape((1, batch_length, n_features))

for i in range(len(df_test)):

    # predict one time step ahead
    current_pred = model.predict(current_batch)[0]

    # store the prediction
    test_predictions.append(current_pred)

    # update batch to now include prediction and drop first value
    current_batch = np.append(current_batch[:,1:,:],[[current_pred]],axis=1)
```

Looks Good

- The prediction looks good over the 18-month period

```
df_test.plot();
```