RNN Introduction

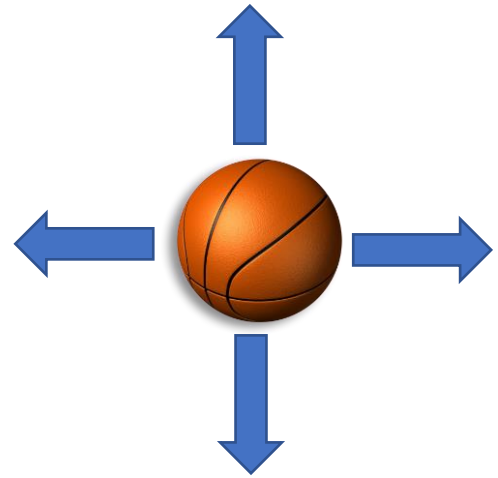
RNN Introduction

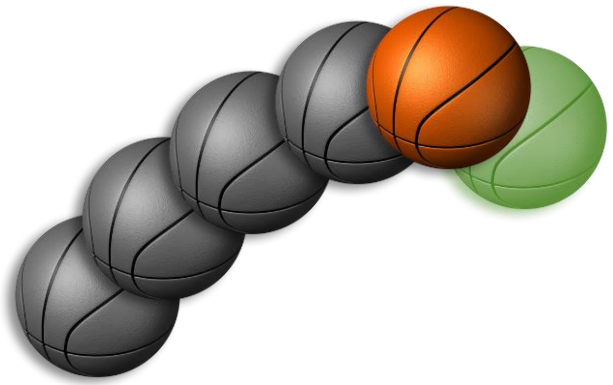
Sequential data

Imagine a ball in the air. We can see one frame of the ball, and we are asked to predict where it will be in 0.5 seconds. This task is quite challenging because we have no information about the ball's velocity.

Now, consider if we were given images of the ball from the last 2 seconds. Predicting the ball's position in the future becomes much easier, and we can achieve this with a high degree of accuracy.

This example illustrates a fundamental concept of sequential data. The position of the ball at each time step forms a sequence. Predicting the next element in the sequence is very difficult if we only have the previous element. However, having access to the entire sequence simplifies the task significantly.



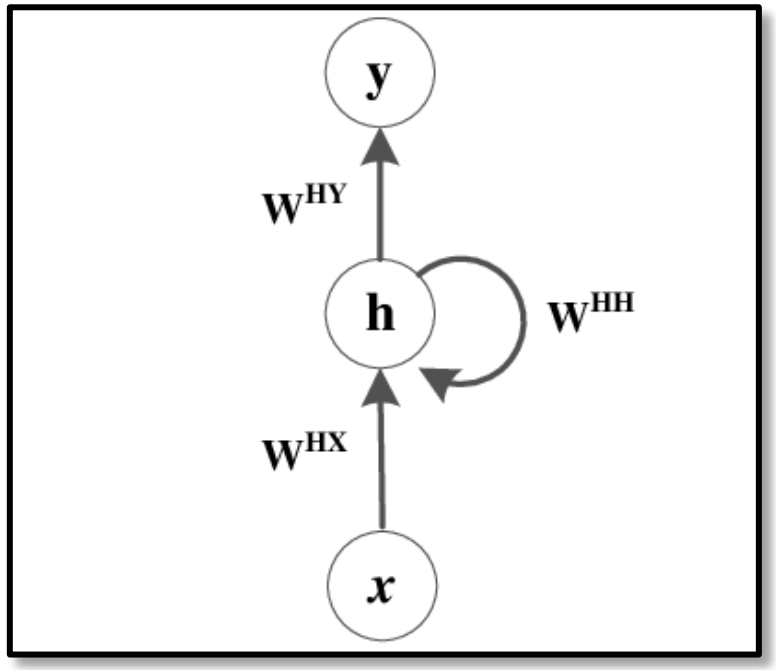


Why do we need RNNs?

When dealing with sequential data, capturing the dependencies and relationships between the elements of the sequence is crucial for making accurate predictions. Simple artificial neural networks (ANNs) are unable to capture these relationships effectively.

To address this limitation, we need to introduce a new architecture that utilizes memory to perform this task efficiently. Recurrent Neural Networks (RNNs) were specifically designed to tackle this problem.

These neural networks possess a unique ability to store the relationships between different elements in a sequence of data. This capability makes RNNs the ideal choice for problems involving sequential data.



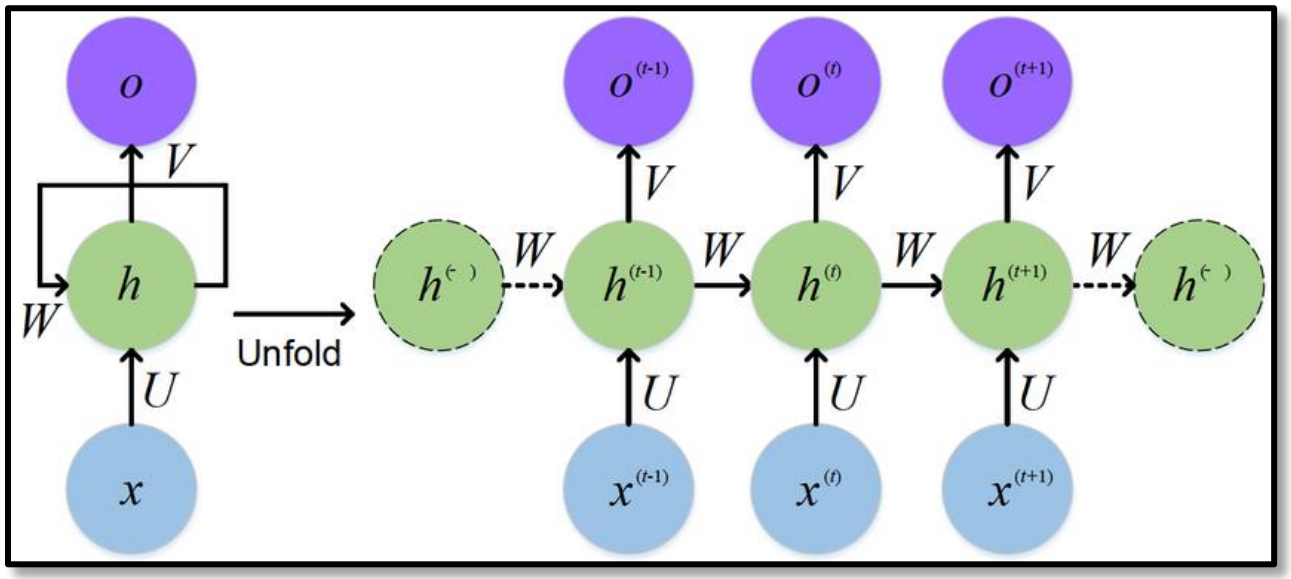
Source: www.researchgate.net

Standard illustration of RNNs

There are two common ways to visualize an RNN: the standard diagram and the unfolded diagram.

For now, let's consider the standard diagram. As we discussed earlier, elements in a sequence have relationships with each other, and it is crucial to capture these dependencies. We will utilize hidden states to accomplish this. When the first input is fed into the model, it makes a prediction based on that input and also saves some features of that input in its hidden states. Then, when the next element of the sequence is given to the model, it predicts the new output based on the new input and the hidden state from the previous element.

This process continues for each element in the sequence, with the model updating its hidden states and making predictions based on the current input and the previous hidden state.

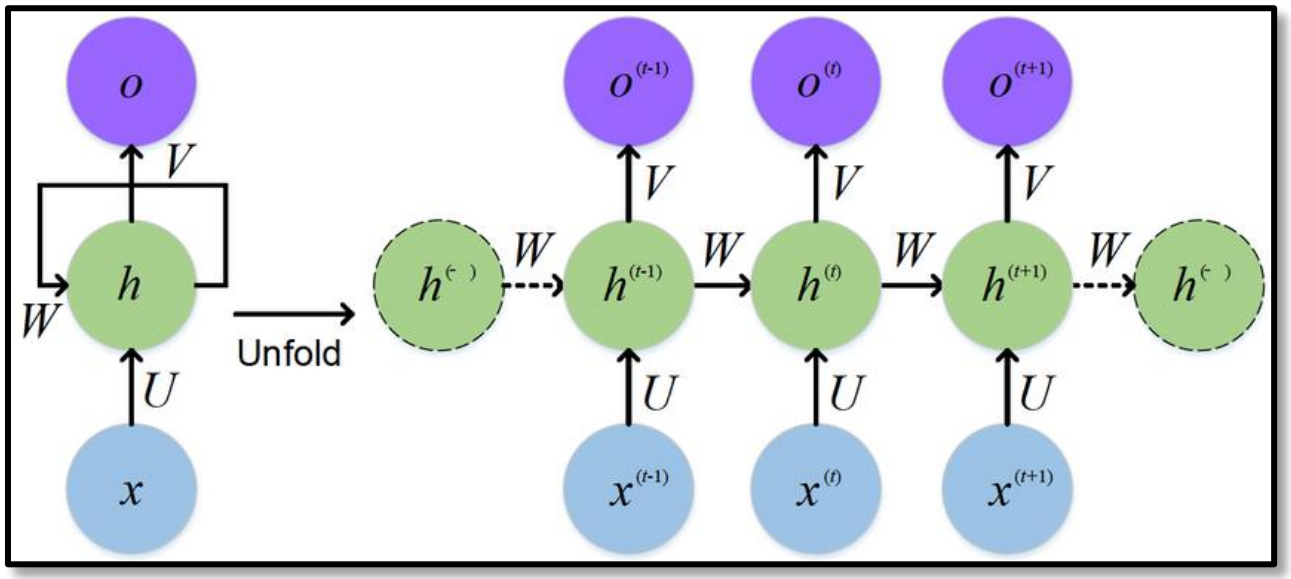


Forward feed in the unfolded illustration

We now understand what the standard illustration of the RNN means. Next, we will use our example to better comprehend the unfolded illustration of the model.

Let's say we have captured the position of the ball along all three axes (x, y, z) every 0.25 seconds. We will use the last 5 elements in the sequence to predict where the ball will be in the next 0.25 seconds.

Each of our inputs (X) is a vector of three numbers corresponding to the three axes. In total, we have 5 of these inputs. The data from the first time step generates the first output and the first hidden state. Then, the data from the second time step, along with the first hidden state, generates the second output and the second hidden state. This process continues until the fifth output is generated by the fourth hidden state and the fifth input. Finally, we can add a fully connected layer to predict the next position of the ball.

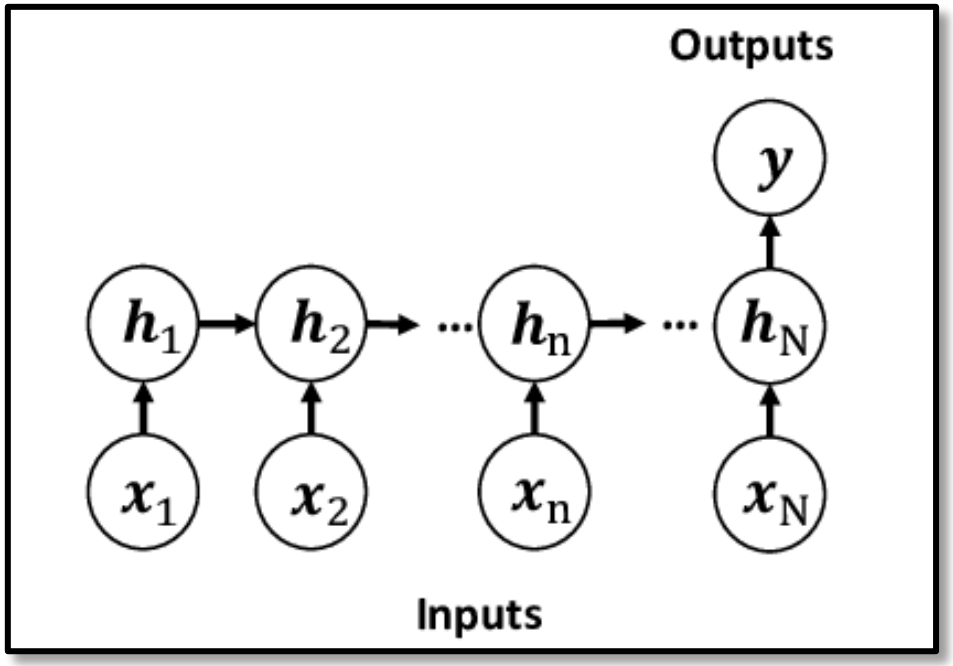


Many to One RNN

In the previous illustration, we observed the many-to-many architecture. This means that we have more than one set of inputs and generate more than one output as well.

In some other RNN structures, only the last RNN cell in the layer generates an output, while the other RNN cells only produce their next hidden state using the previous hidden state and their input.

These structures are referred to as many-to-one networks, as they have many inputs but generate only one output. We will discuss the various applications of these different types of RNNs later.



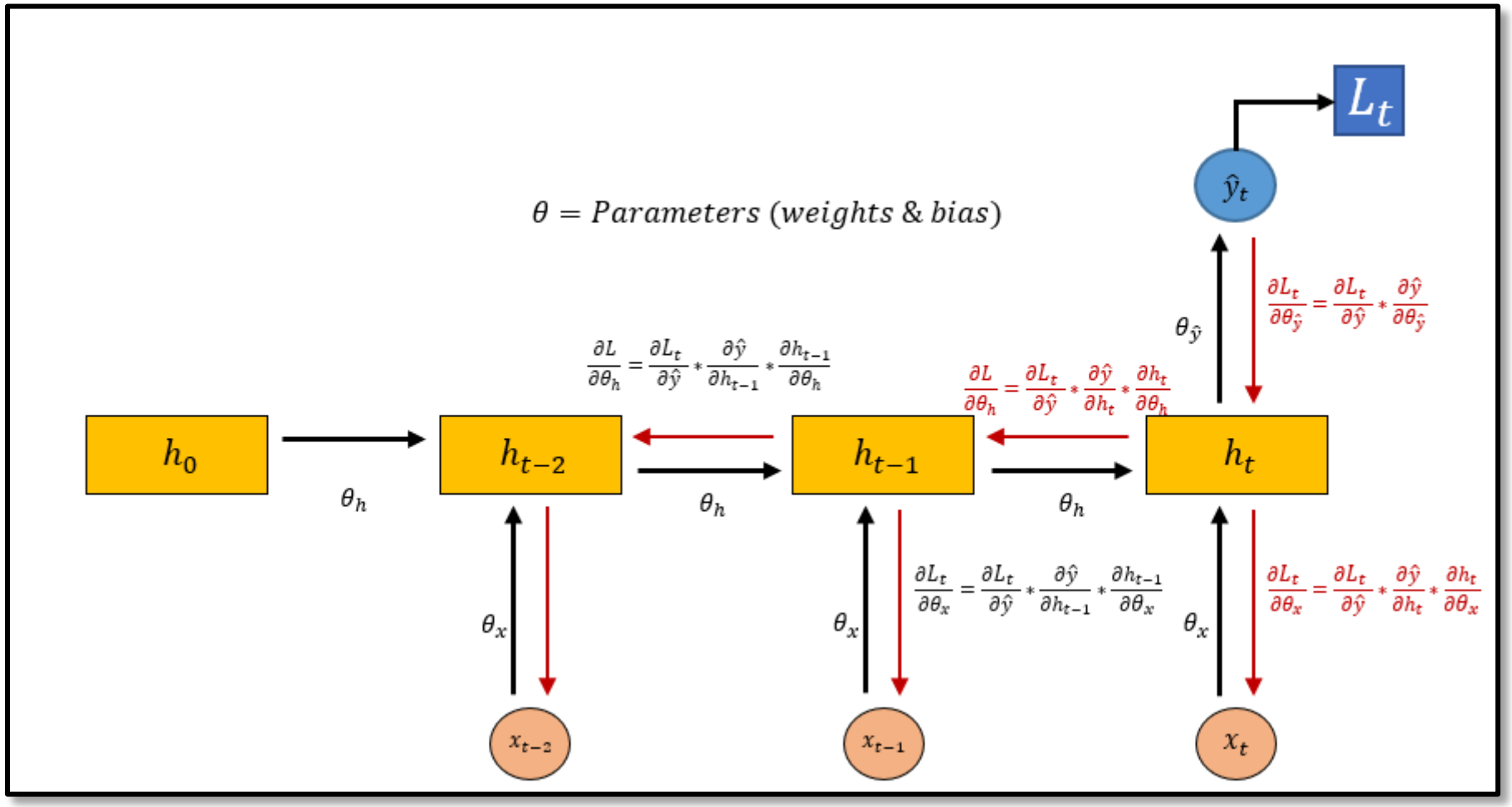
Source: www.researchgate.net

Backpropagation in RNNs

We have already seen how data is fed into RNN structures and how these models generate their outputs.
Now, we will discuss how backpropagation is performed in RNNs.
In RNNs, the output of the next neuron depends on its previous hidden state, which is generated by the current input and the previous hidden state. This dependency means that we cannot calculate the derivatives with respect to the weights of the cell number n without first calculating the derivatives for the cell number n+1.

This illustrates a key difference from regular layers, where we can calculate the derivatives of all the weights of a layer simultaneously.
In RNNs, we must backpropagate through the layer in reverse order, moving backward through the sequence to update the weights effectively.
This process is known as Backpropagation Through Time (BPTT), as it involves unrolling the RNN into a deep feedforward network and applying backpropagation. Furthermore, BPTT allows us to capture long-term dependencies in the data, as the gradients are propagated back through the entire sequence.

Backpropagation in RNNs



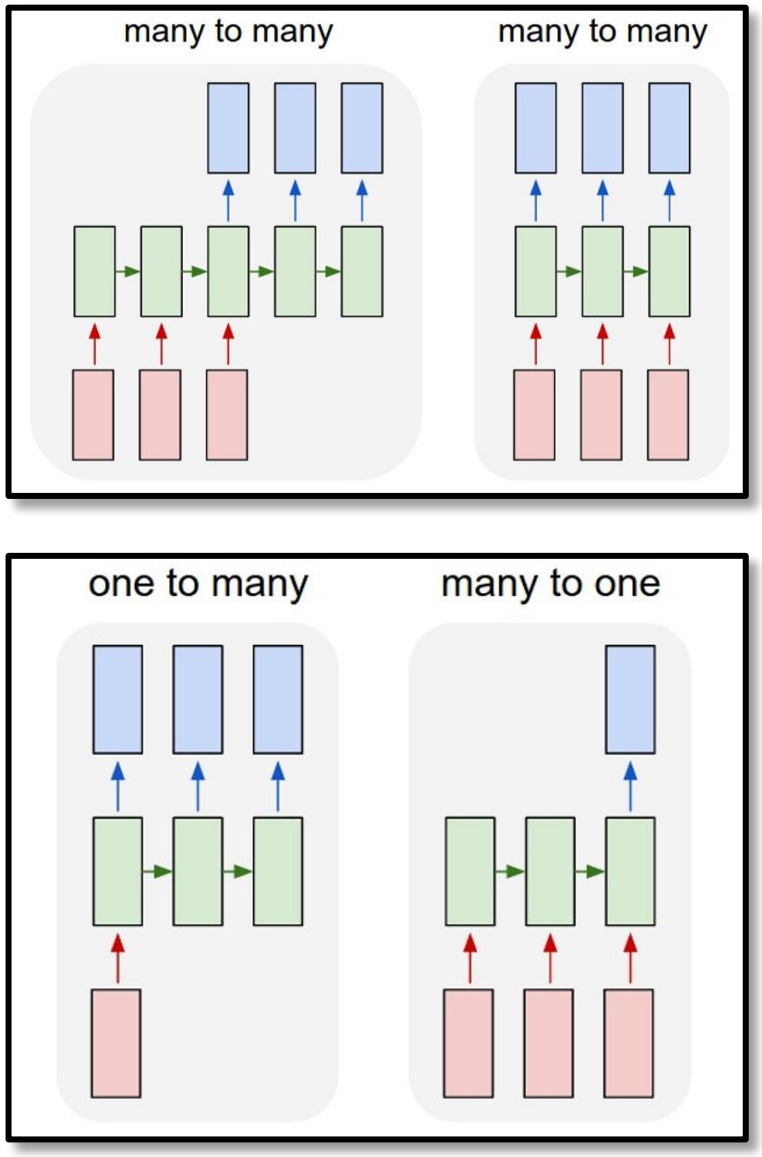
Source: www.towardsdatascience.com

Different sequence problems

Now that we understand what an RNN is, it is important to explore the different problems this architecture can address.

RNNs were specifically designed to solve sequence-related problems by capturing the relationships between elements in a sequence. There are various types of sequential problems we can tackle with RNNs.

Sometimes, we use sequential data to predict a numerical value. Other times, we apply sequential data for classification tasks. In some cases, we aim to generate a sequence of data from. Lastly, we can perform sequence-to-sequence prediction, where the input and output are both sequences.

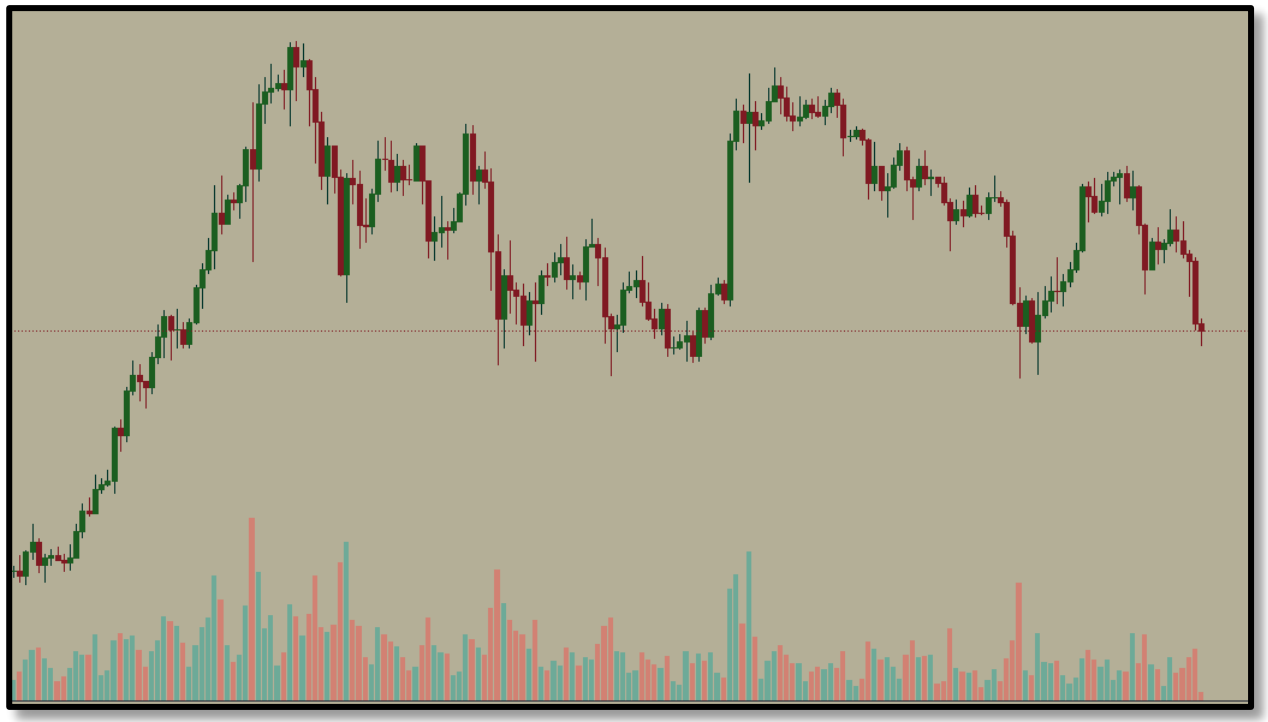


Sequence Numeric Prediction

In these problems we have a sequence of data and we want to output a number as our output. The first example in which we wanted to predict the next position of the ball using its last 5 positions relates to this class.

Another popular example of these kind of problems is stock prediction. In this example we have the open, high, low and close price of each day and we want to predict the tomorrow's closing price.

To tackle these regression problems using RNNs, we can use a many-to-one architecture. The RNN takes a sequence of inputs (e.g., past stock prices) and generates a single output (e.g., the predicted closing price for the next day).

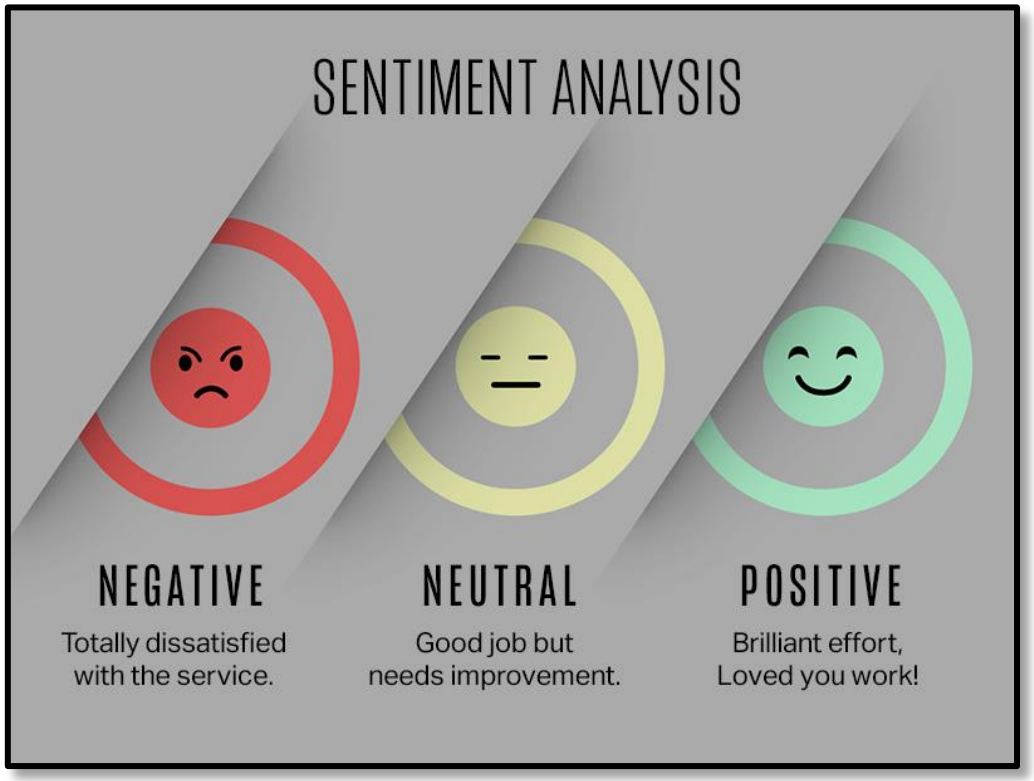


Sequence Classification

In classification problems, we have a sequence of data, and our goal is to classify them into predefined categories.

A popular example of this type of problem is sentiment analysis in text. In this task, we are given a text and aim to analyze its sentiment. For instance, we can classify the text as offensive or not offensive. Alternatively, we can categorize it as optimistic or pessimistic.

To tackle these classification problems using RNNs, we will need to add a layer with softmax activation on top of the network, just like in any other classification problem. The key difference lies in the RNN layer, where we capture the relationships between the words in the input sequence.



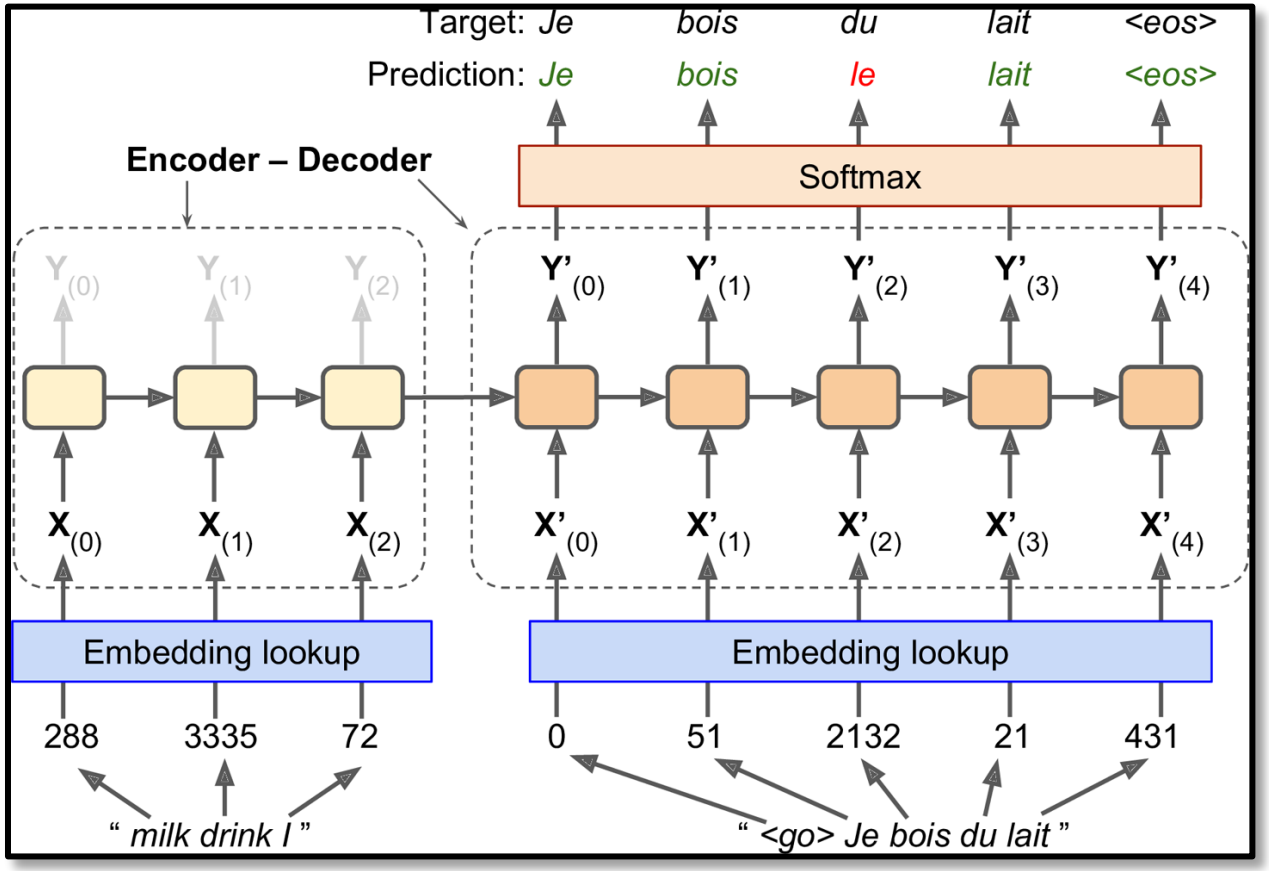
Source: www.imerit.net

Sequence to Sequence

In sequence-to-sequence problems, we have a sequence of data, and our goal is to generate another sequence based on it.

A popular example of this type of problem is text translation. For instance, let's say we are given a sequence of words in Turkish and we want to translate it into English. This means we will generate a sequence of words in English using the input text in Turkish.

To solve these types of problems, we can use a many-to-many RNN architecture. In this setup, the RNN processes the input sequence and generates an output sequence, allowing it to effectively capture the relationships and dependencies between the elements in both sequences.



Source: www.oreilly.com

Summary

In this presentation, we covered sequential data and the importance of capturing the relationships within these sequences.

Recurrent Neural Networks (RNNs) were introduced as a solution, utilizing hidden states to capture these relationships. We explored both the feedforward and backpropagation processes in these networks, noting that RNNs implement backpropagation through time.

Finally, we discussed various sequential problems that RNNs can address. In the upcoming lectures, we will demonstrate how to implement these networks in Python. We will also examine the limitations of RNNs and introduce alternative solutions.