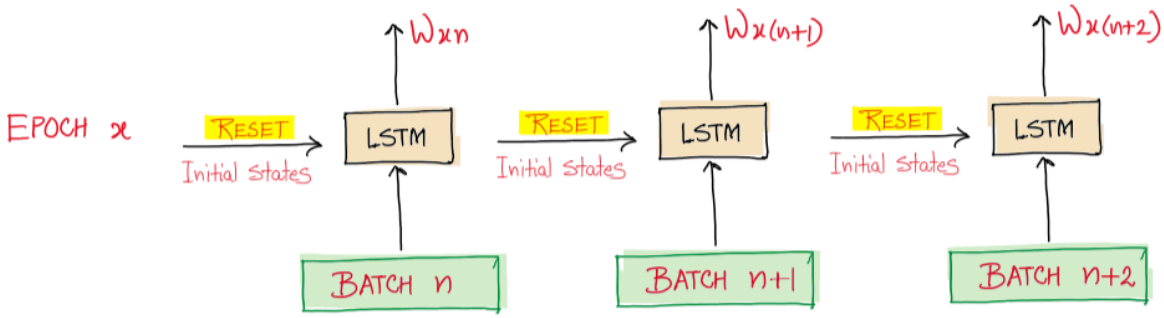
## Exercise 1

### What's the difference between Stateful RNN vs. Stateless RNN? What are their pros and cons?

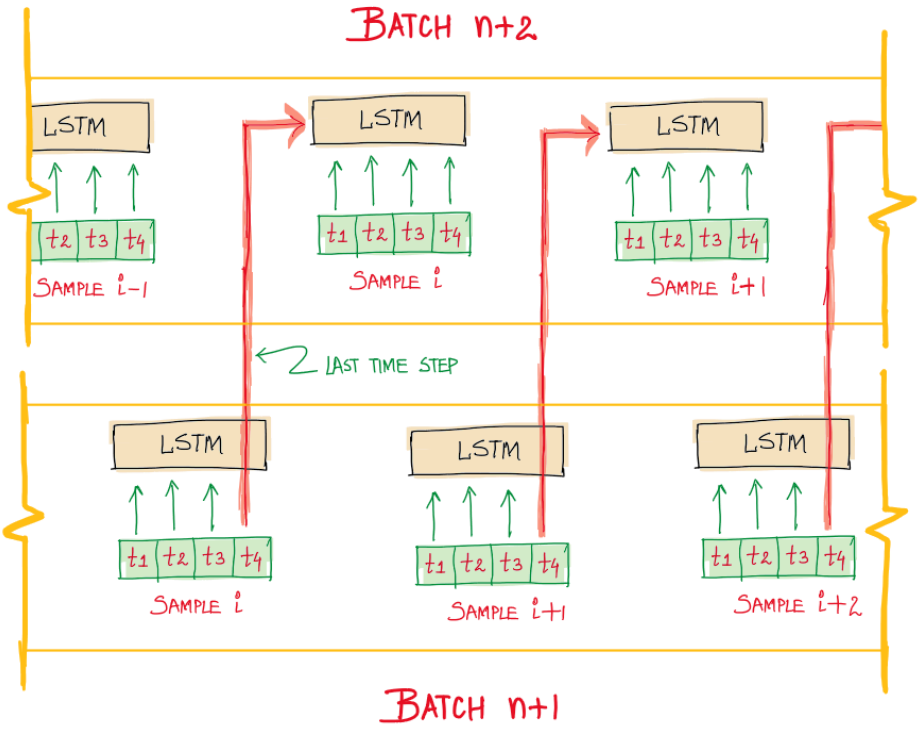
To find the difference between these two, we first need to explore their architectures. Sequential modeling algorithms are divided into two categories: **stateful** and **stateless** RNNs, depending on the architecture used during training. In stateless RNNs, each batch created for training is independent, with no mutual relationship between batches.

The training process of a **stateless RNN** is as follows:

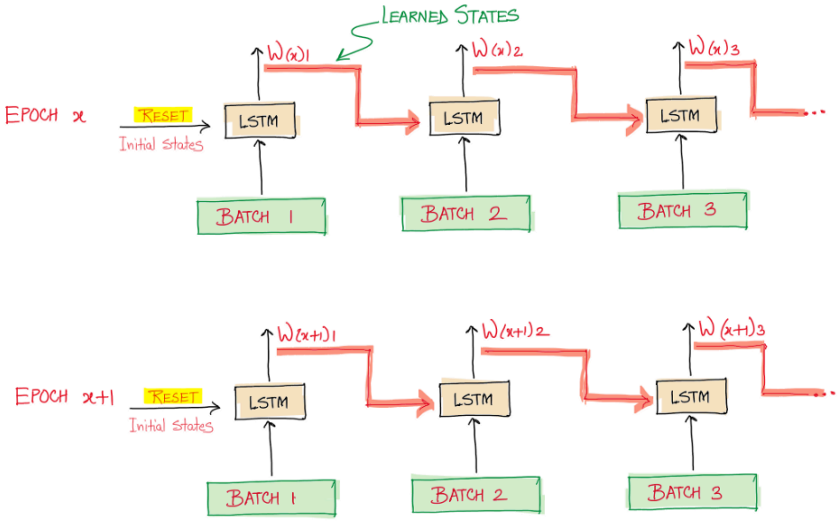



As shown in the diagram above, each time a new batch is processed, the initial states are reset to zero, meaning no previously learned states are used. This forces the model to forget what it learned from previous batches.

On the other hand, the training process of a **stateful RNN** is as follows:




Here, the cells and hidden states of this model are initialized for each batch using the states learned from the previous batch, enabling the model to learn dependencies between batches. However, at the start of each epoch, the states are reset. If we take a closer look at the model, we see the following diagram:



In this example, the state of sample located at index i, xix_ixi, is used to calculate the next sample xi+1x_{i+1}xi+1 in the next batch. More specifically, the last state for each sample in index i of one batch is used as the initial state for the same sample in the next batch. In the diagram, the sequence length for each sample is 4 time steps, and the model's state values at time step t = 4 are used as initial values for the next batch.

Thus, the difference between the two becomes clear. The distinction lies in how the states of the model (related to each batch) are initialized as we move from one batch to the next. Note that this should not be confused with the parameters/weights, which are propagated throughout the entire training process— the goal of training. In **stateless RNNs**, at the end of each batch, "the network's state is reset," while in **stateful RNNs**, the network's state is preserved across batches, though it must be manually reset at the end of each epoch.

In stateless RNNs, the model updates parameters in the first batch and then initializes hidden and cell states (usually all zeros) for the second batch. In contrast, stateful RNNs use the hidden and cell states outputted from the first batch as the initial states for the second batch. A stateless RNN requires you to structure your data in a specific way, leading to higher performance, while a stateful RNN can handle different time steps but incurs a significant performance penalty.

When comparing the pros and cons of these two models, we find that **stateless RNNs** generally offer better performance. However, **stateful RNNs** must be trained one epoch at a time, and their internal state must be manually reset using `reset_state()` after each epoch. A benefit of **stateful RNNs** is that they often require smaller networks, resulting in shorter training times. On the other hand, **stateless RNNs** take longer to train, need more memory, and are more sensitive to the random initialization of weights. Additionally, using techniques like dropout is harder in stateless RNNs.

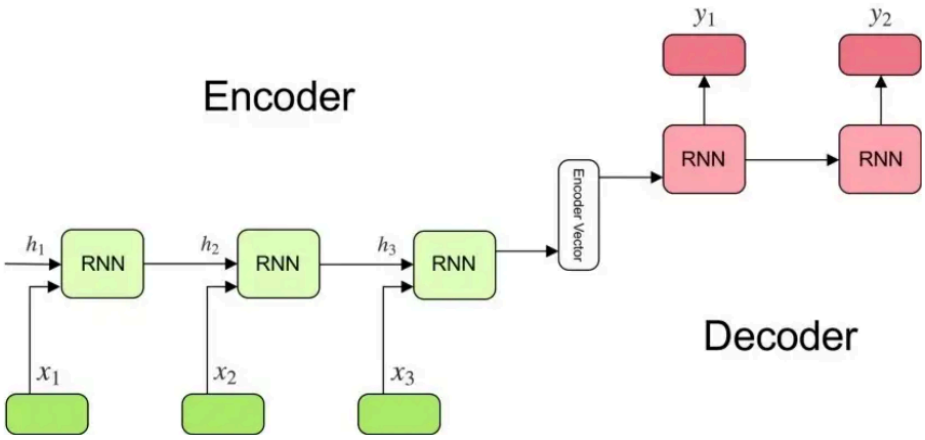
In general, when there is a connection between two sequences in two batches (e.g., stock prices), it is better to use a **stateful RNN**. Otherwise (e.g., when a sequence represents a complete sentence), a **stateless RNN** is more appropriate. Most people use **stateless RNNs** in practice because, when using stateful RNNs, the network must handle indefinitely long sequences during the generation phase, which can be challenging to manage.

## Exercise 2

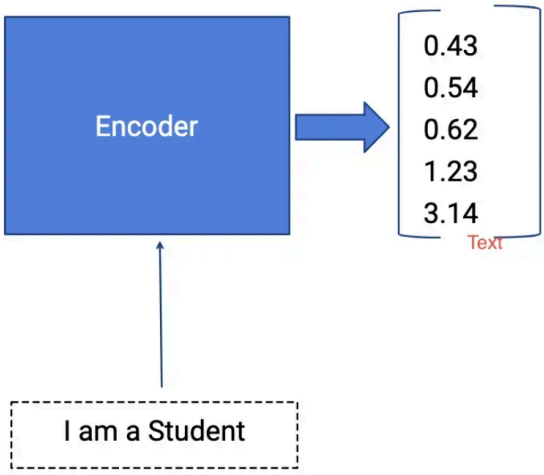
### Explain the main difference between Encoder-Decoder RNNs such as Seq2Seq model vs plain RNNs.

First, let's look at the **Encoder-Decoder RNN**. This model consists of two recurrent neural networks: one acting as an encoder and the other as a decoder. The encoder maps a source sequence of variable length to a fixed-length vector, and the decoder maps that vector representation to a target sequence of variable length. The encoder-decoder RNN is often used for predicting the next value in a sequence with real values or for outputting a class label for an input sequence. This can be set up as one-to-one or many-to-one (e.g., predicting the next step based on previous ones).

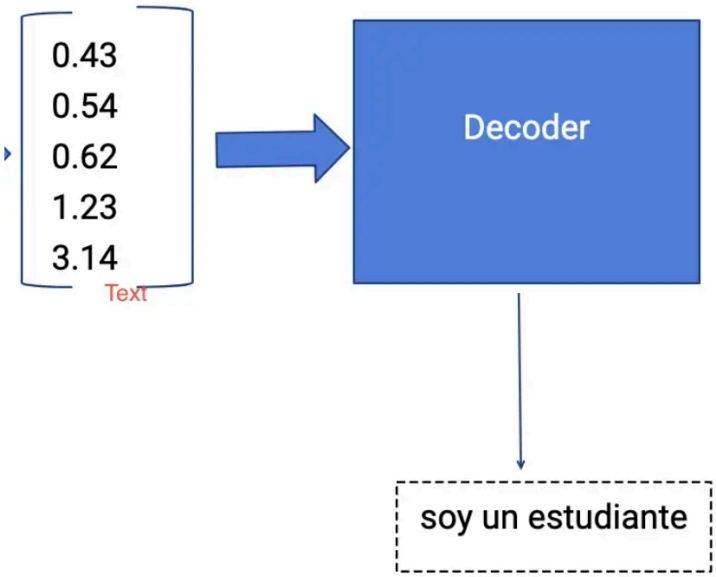
A more complex version of sequence prediction involves taking a sequence as input and requiring a sequence as output. This problem is known as **sequence-to-sequence** (Seq2Seq) prediction. To understand the structure, we can refer to the following diagram:




The encoder consists of several RNN cells arranged together. The RNN reads each input in sequence. For each time step (each input), the hidden state (hidden vector) $h_t$ is updated according to the current input $x_t$. After reading all the inputs, the final hidden state of the encoder represents the summary/context of the entire input sequence. For example, if the input sequence is "I am a student," the encoder will have four time steps/tokens. At each time step, the hidden state $h_t$ is updated using the previous hidden state and the current input.

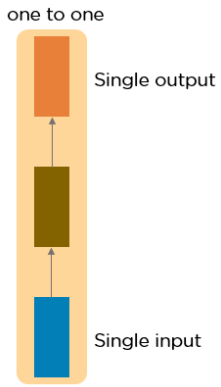



The decoder generates the output sequence by predicting the next output based on the hidden state. The input to the decoder is the final hidden state from the encoder. Each layer will have three inputs: the hidden vector from the previous layer $h_{t-1}$, the output from the previous layer $y_{t-1}$, and the main hidden vector h.

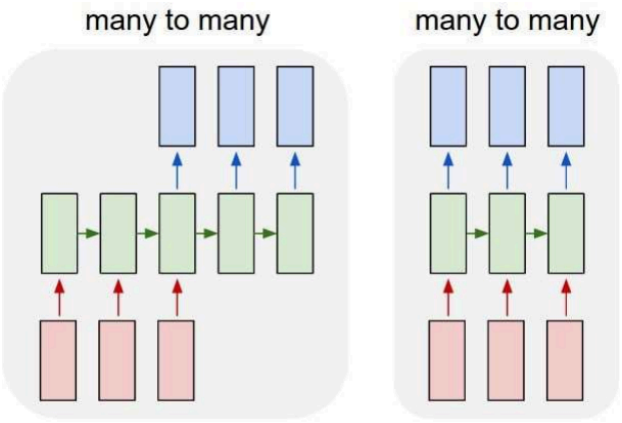


Now that we've explained this model, let's move on to the next one. We know how a plain RNN works. There are different types, such as one-to-one, one-to-many, many-to-one, and many-to-many.

- **One-to-one**: This is also known as the vanilla neural network and is used for machine learning problems where there is one input and one output.




- **One-to-many**: This RNN has one input and multiple outputs, such as image captioning.
- **Many-to-one**: This model takes a sequence of inputs and produces one output. Sentiment analysis is a good example, where a sentence is input, and it is classified as having positive or negative sentiment.
- **Many-to-many**: This model takes a sequence of inputs and produces a sequence of outputs, like machine translation.



Now that we understand the basic structure of these two models, let's discuss their applications. Generally, we prefer using the **Seq2Seq model** when we need to predict sequences or periodic signals using an RNN. A vanilla RNN works well but struggles to "remember" events from more than 20 steps back. If we want a machine translation system, using an encoder-decoder RNN is the better approach.

## Exercise 3

Suppose we want to build a gated RNN cell that sums its inputs over time. What should be the gating values be? To focus on the gating aspect, your design can change the activation function of the RNN cell (e.g., replace tanh by linear).

For the LSTM architecture, what should be the value of the input gate and the forget gate?

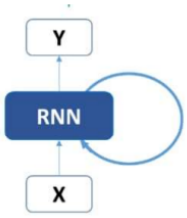
For the GRU architecture, what should be the value of the reset gate and the update gate?

Suppose we want to build a gated RNN cell that sums its inputs over time. The RNN formula looks like this:

$h_t = f(h_{t-1}, x_t)$

$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$

$y_t = W_{hy}h_t$




If we sum these inputs over time, we get the following formula:

$h_t = linear(h_{t-1} + x_t)$ $\quad if\ w_{hh} = w_{hx} = w_{hy} = 1 \quad h_t = h_{t-1} + x_t = y_t$

$if\ t = 0: \quad y_0 = x_0$

$if\ t = 1: \quad y_1 = h_0 + x_1 = x_0 + x_1$ $\quad y_{(t)} = h_{(t-1)} + x_t$

Now, for the **LSTM architecture**, we know that the activation function used is sigmoid, which is not suitable for summing inputs. Thus, we replace the activation function with a linear one. The gating values would be:

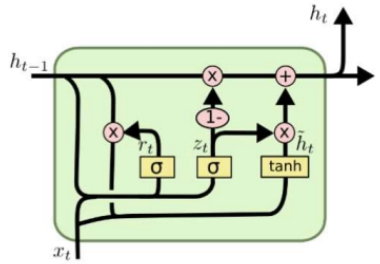
$f_{(t)} = 0$

$g_{(t)} = 1$

$w_c = 1$ , $b_c = 0$ => $C'_{(t)} = linear\left(h_{t-1} + x_{t)}\right) = h_{t-1} + x_t$

$C_t = C'_t = h_{(t-1)} + h_{(t)}$ => "Since we reached the same previous function, it indicates that the model is working correctly."

For the **GRU architecture**, if we replace the activation function as before, we get:




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

If we replace the activation function with an activation function of one, we will have:

$z_{(t)} = 1$

=> $h'_{(t)} = linear(W.[h_{(t-1)}, x_{(t)}])$

$W = 1$ => $h'_{(t)} = linear\left(h_{(t-1)} + x_{(t)}\right) + x_{(t)} = h_{(t-1)} + x_{(t)}$ , $h_{(t)} = h'_{(t)} = h_{(t-1)} + x_{(t)}$

Since the model has computed the formula correctly again, this confirms that the model design is sound.