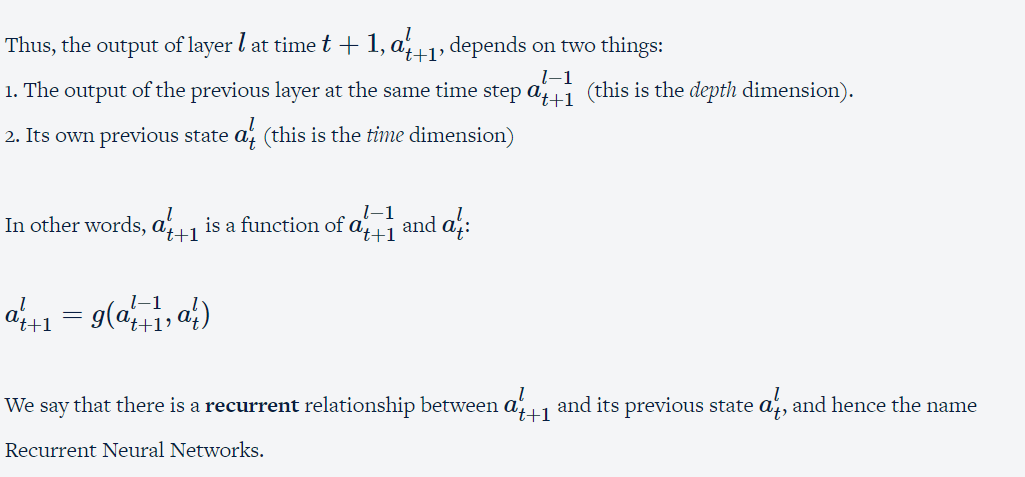
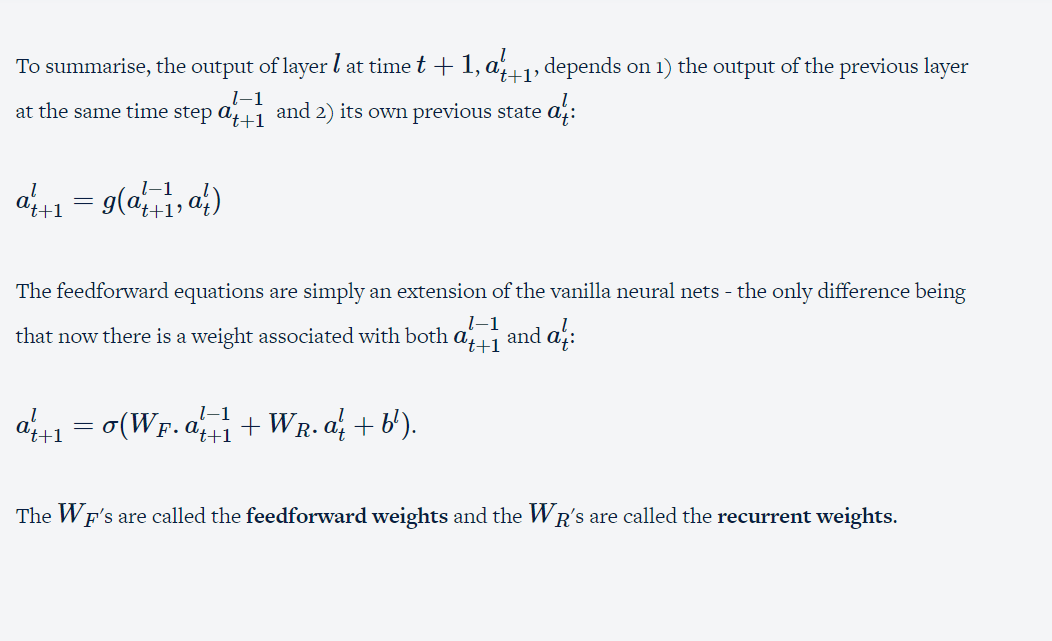
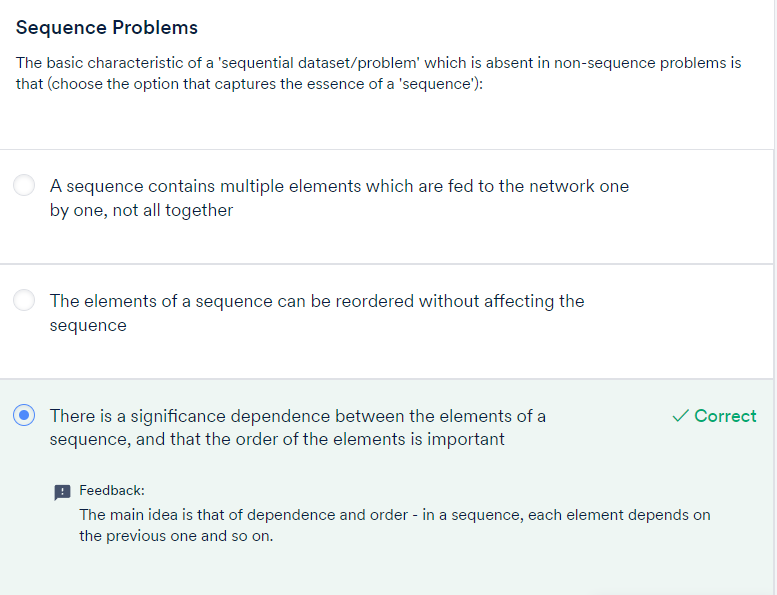
Welcome to the third module of this course. In this module, you’ll learn **Recurrent Neural Networks** or **RNNs**. RNNs are specially designed to work with **sequential data,**i.e. data where there is a natural notion of a 'sequence' such as text (sequences of words, sentences etc.), videos (sequences of images), speech etc. RNNs have been able to produce state-of-the-art results in fields such as natural language processing, computer vision, and time series analysis.

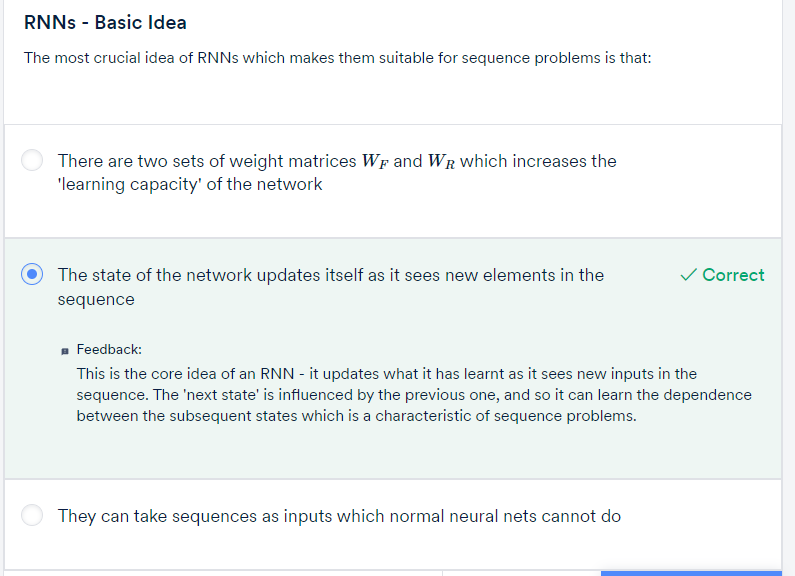
One particular domain RNNs have revolutionised is **natural language processing**. RNNs have given, and continue to give, state-of-the-art results in areas such as machine translation, sentiment analysis, question answering systems, speech recognition, text summarization, text generation, conversational agents, handwriting analysis and numerous other areas. In computer vision, RNNs are being used in tandem with CNNs in applications such as image and video processing.

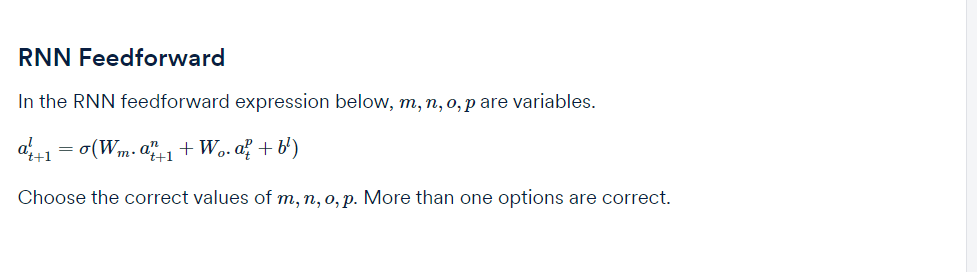
# Sequences:

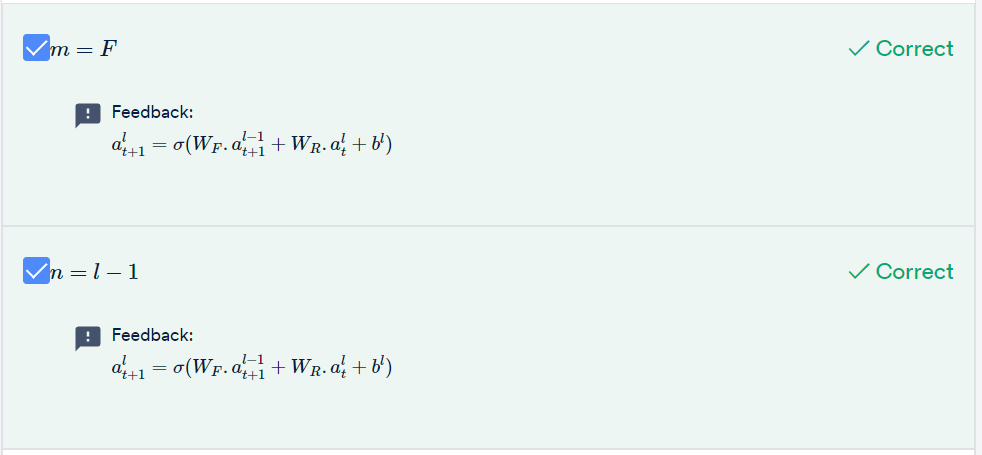


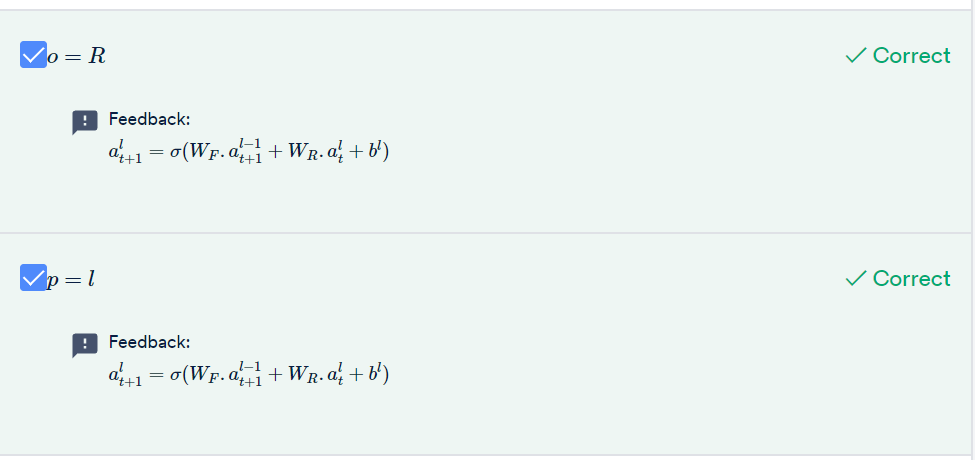


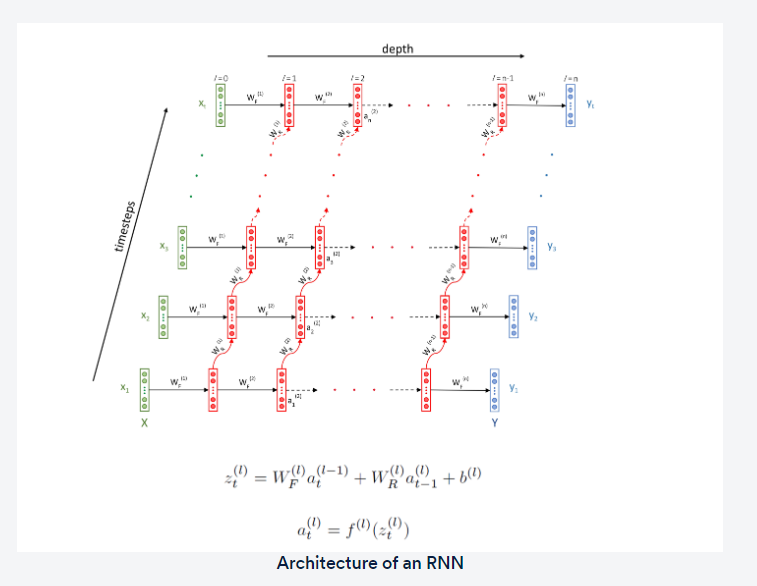


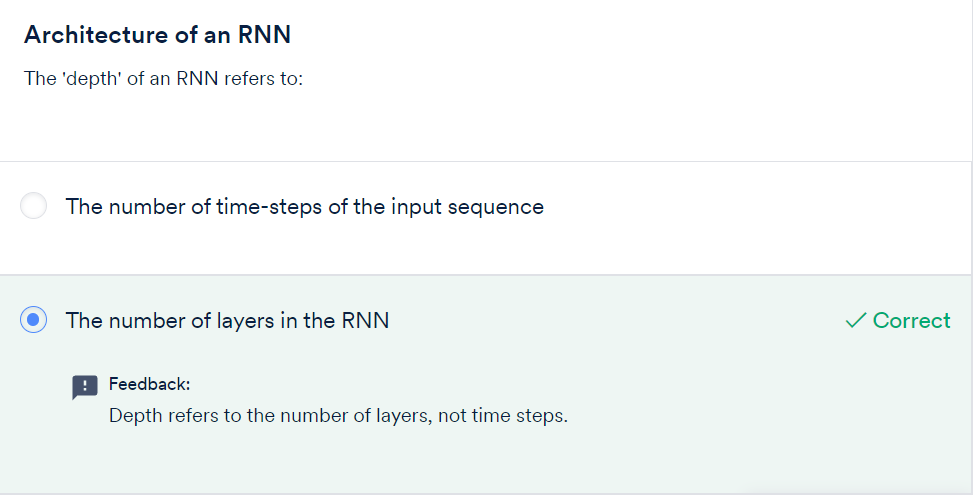




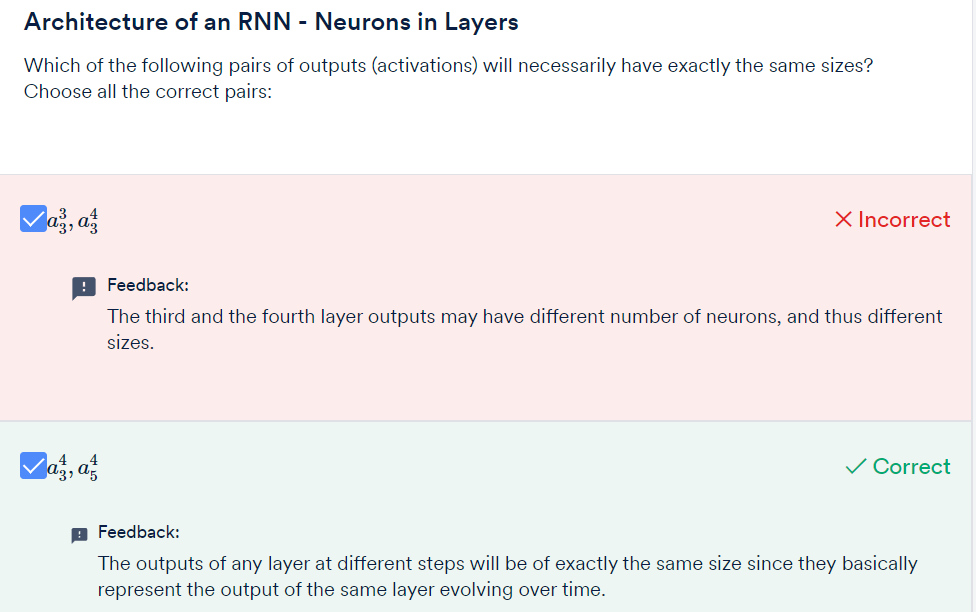


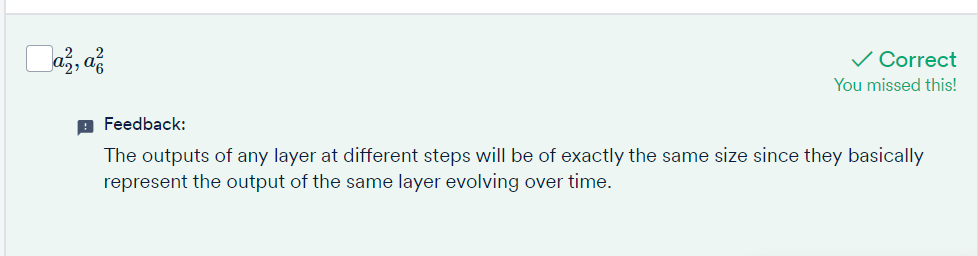


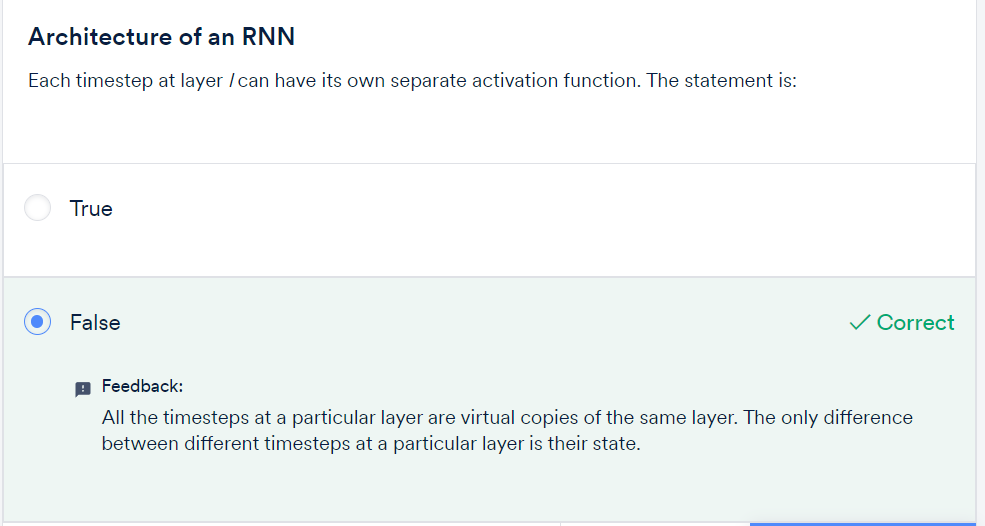


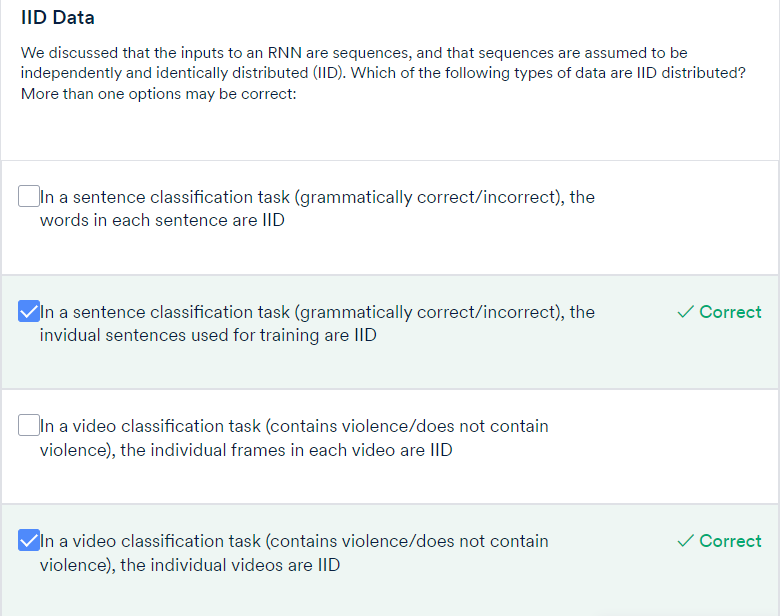


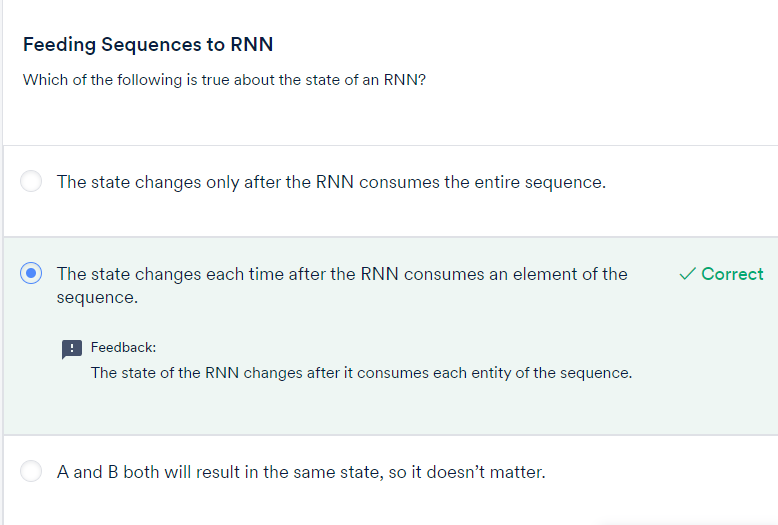


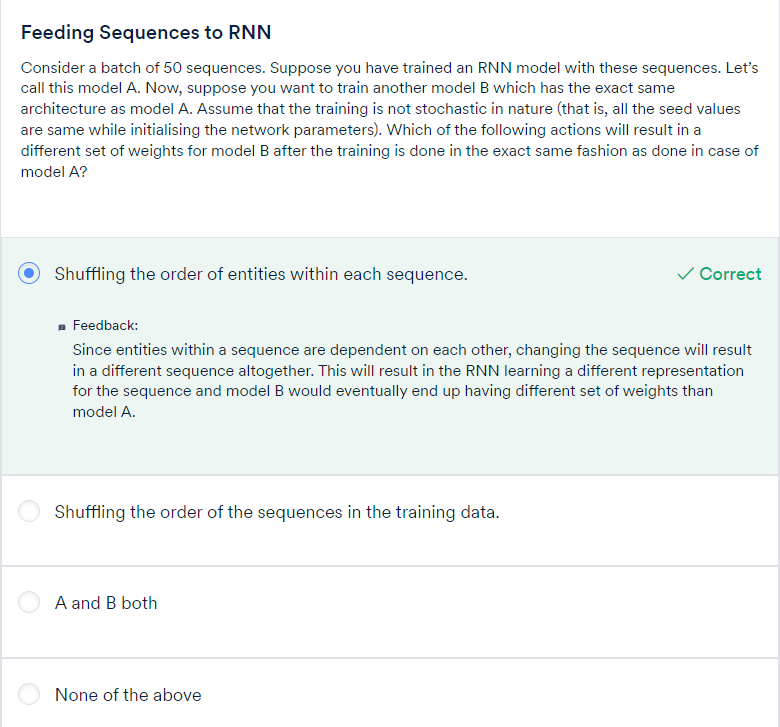


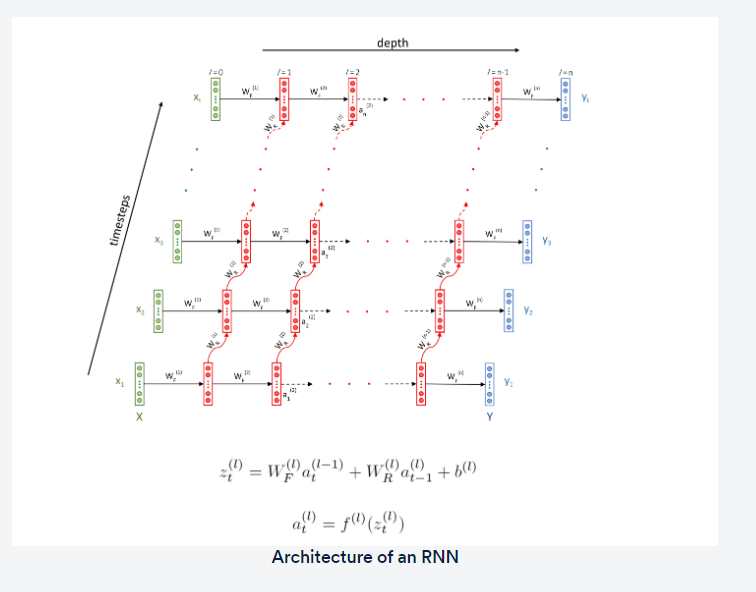












The WF's are the **feedforward weights** which propagate information from one layer to another (left to right).

The WR's are the **recurrent weights** which propagate the information across the time dimension.

Each layer has a bias bl also.



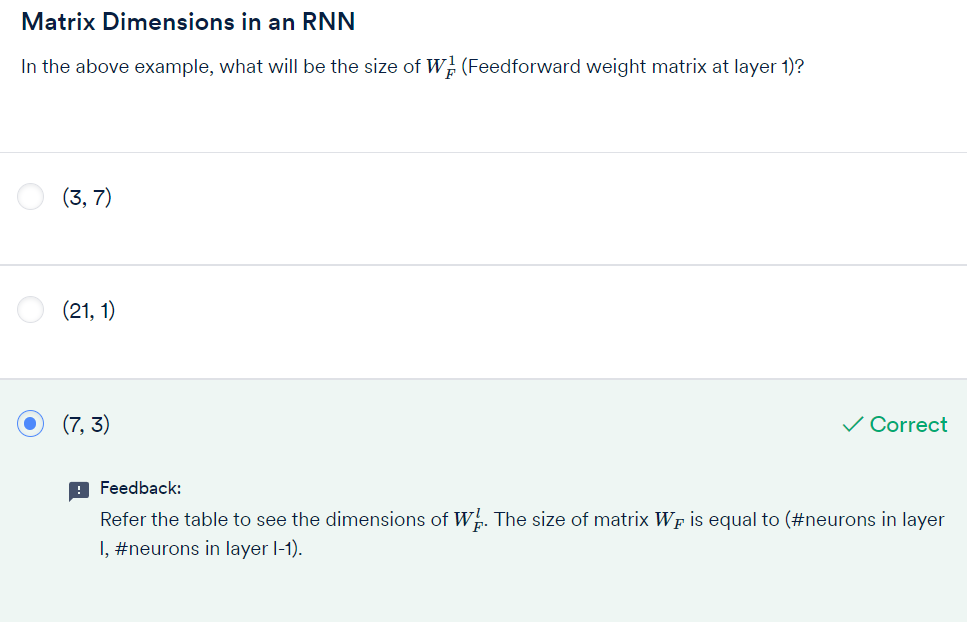


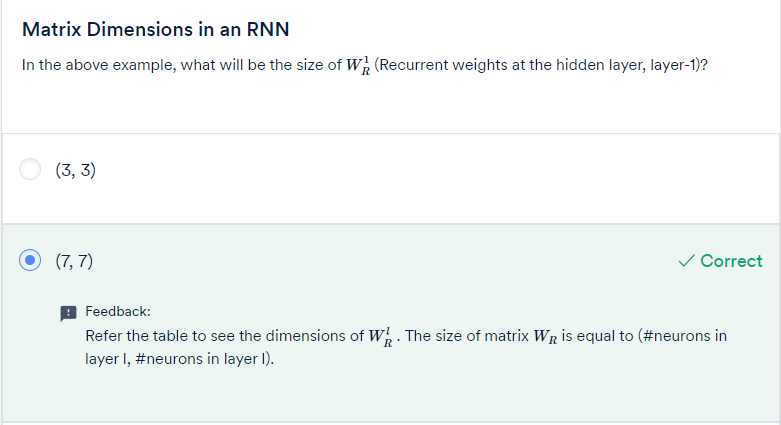
You know that the**recurrent weights** WR connect the same layer to its different states across different time steps. For example, the recurrent weights of layer-3 W3R connects the outputs of the third layer from one time step to the next: a31 to a32, a32 to a33 and so on.

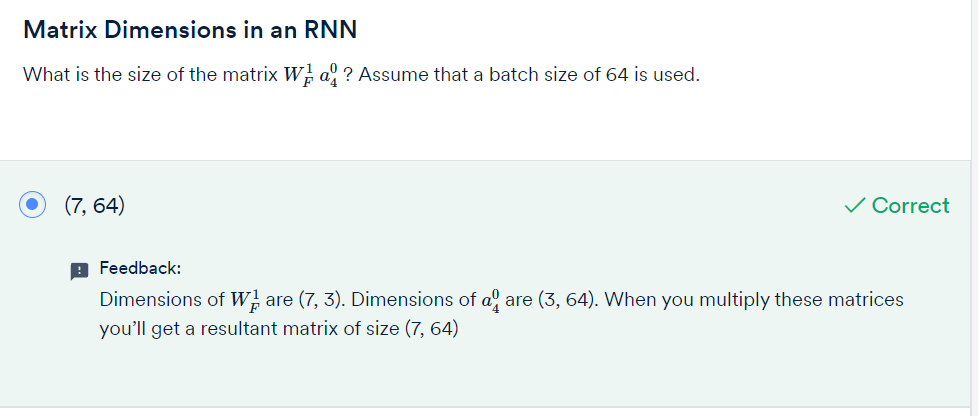
Notice that each of these outputs **have the same size** (it is the same layer, so the number of neurons at each time step is the same, and hence the output size is the same). Thus, all WR's are **square matrices**.

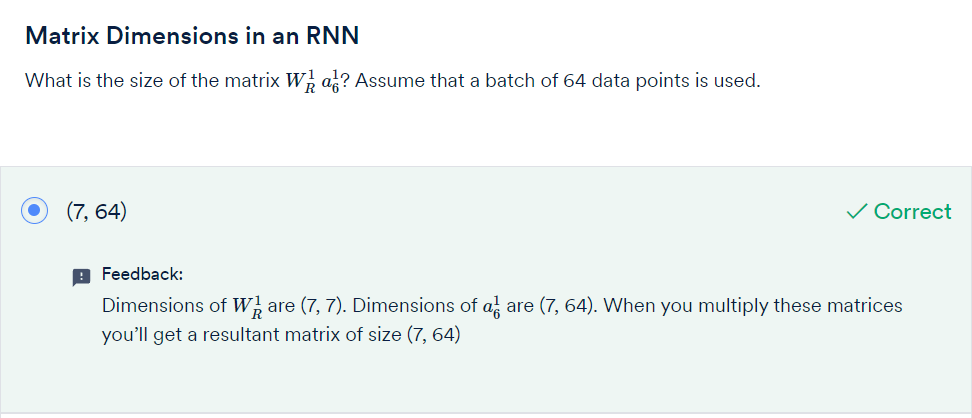
The size of the output vector alt+1 (for any l and t) is the same as that of alt. In other words, the recurrent operation does not change the size of the output.

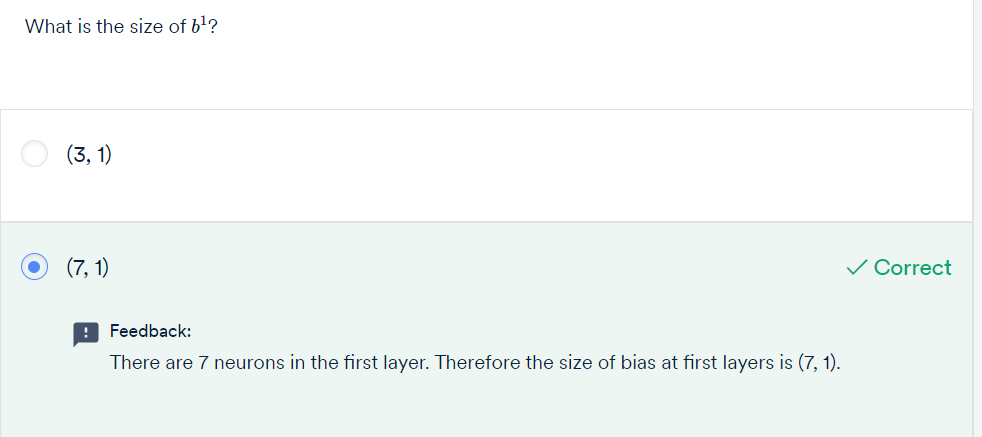
The **biases** of each layer bl, as usual, have the **size equal to the number of neurons** in that layer.

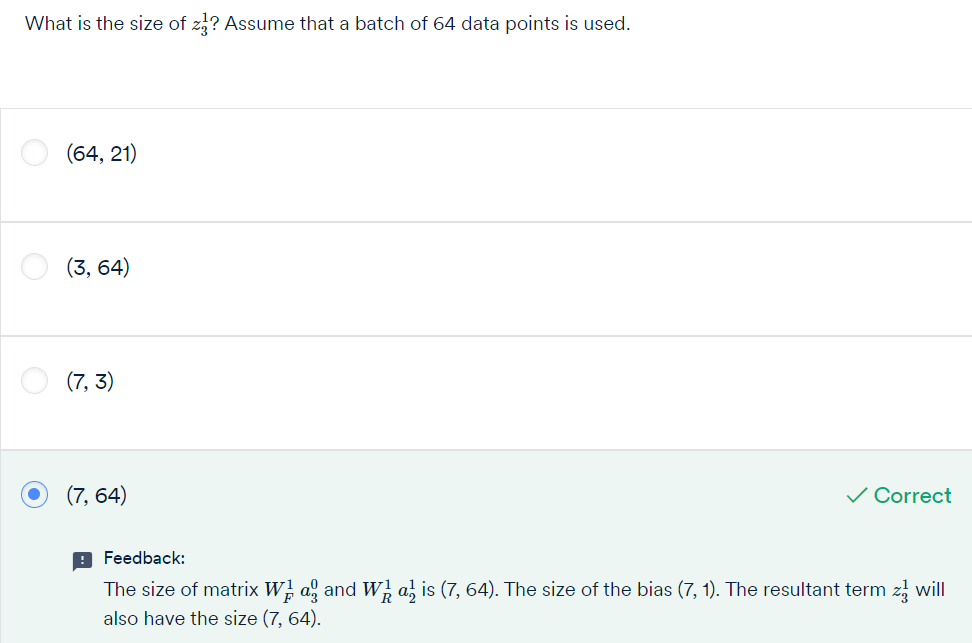


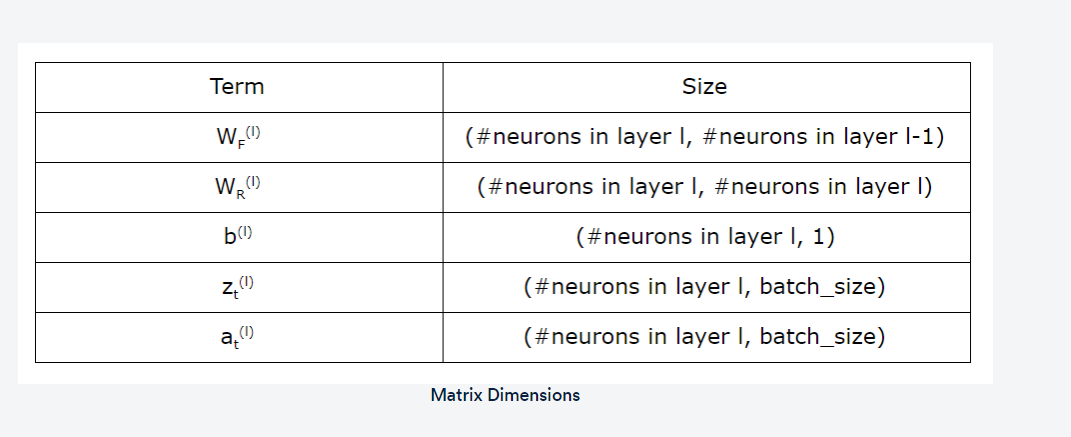


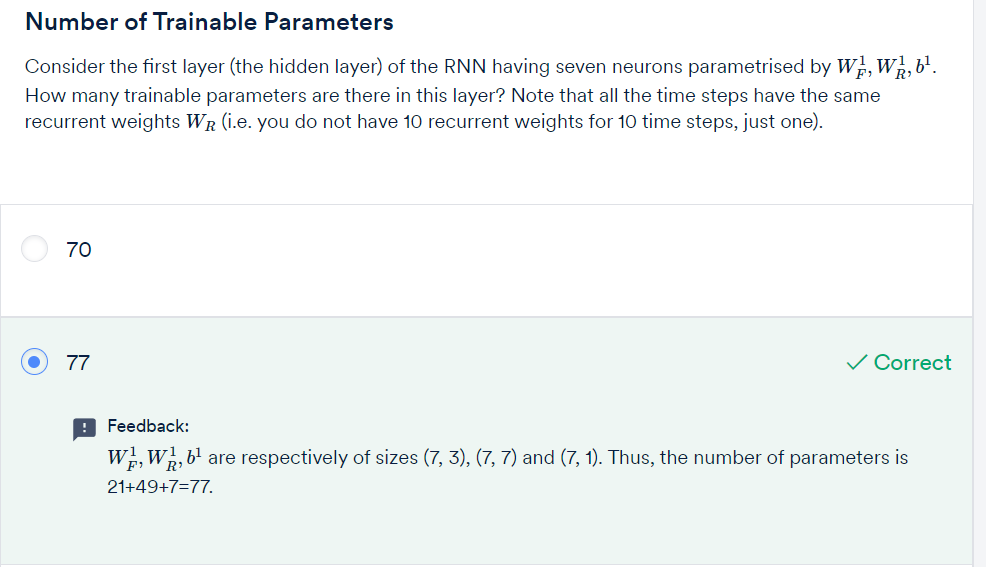


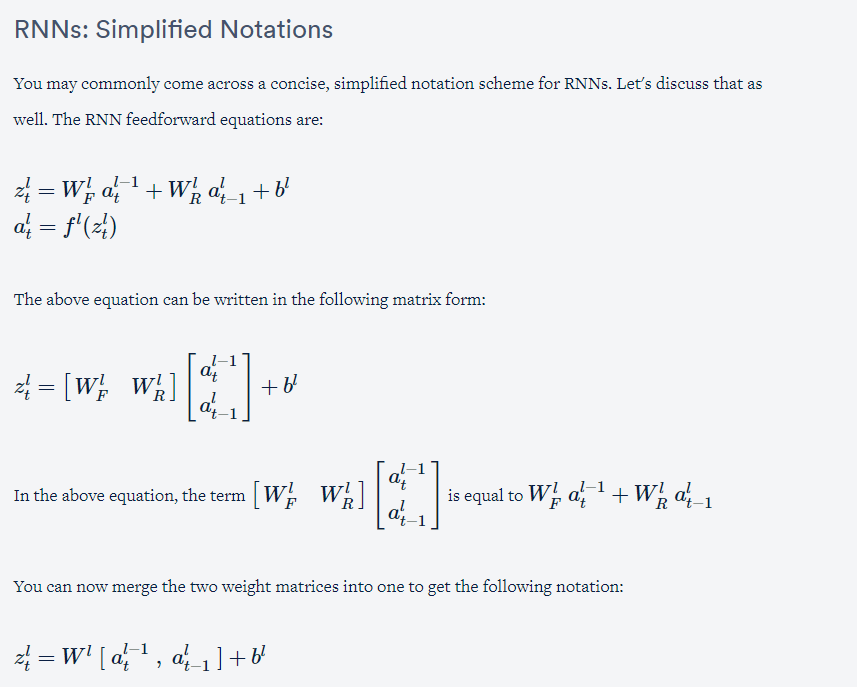


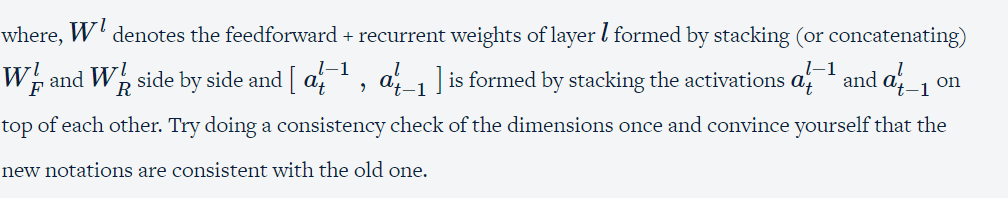


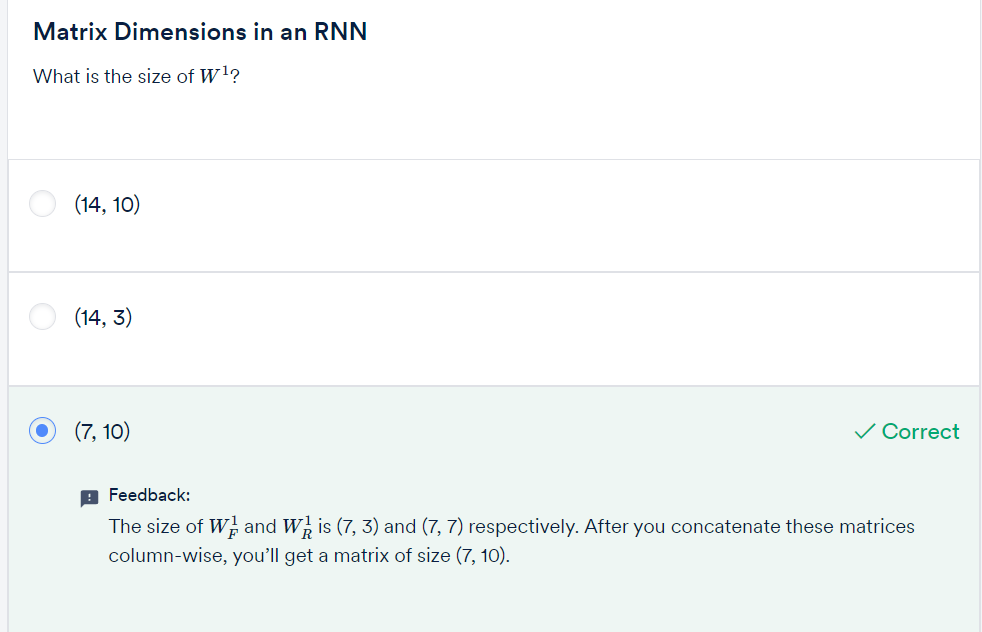


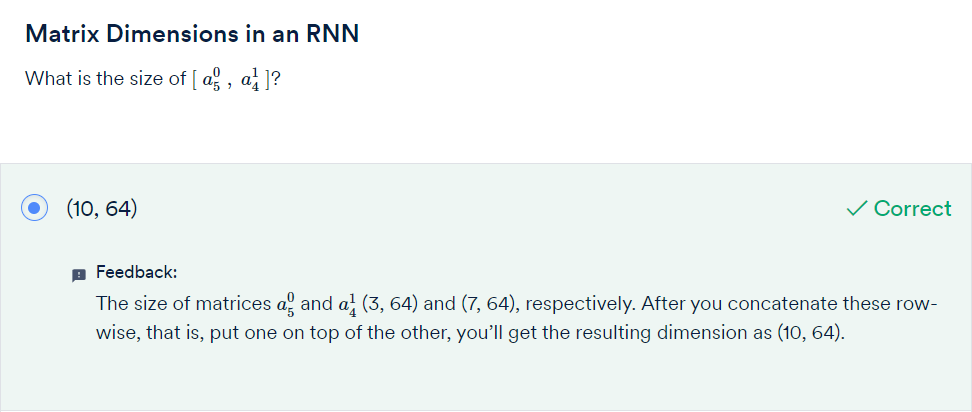




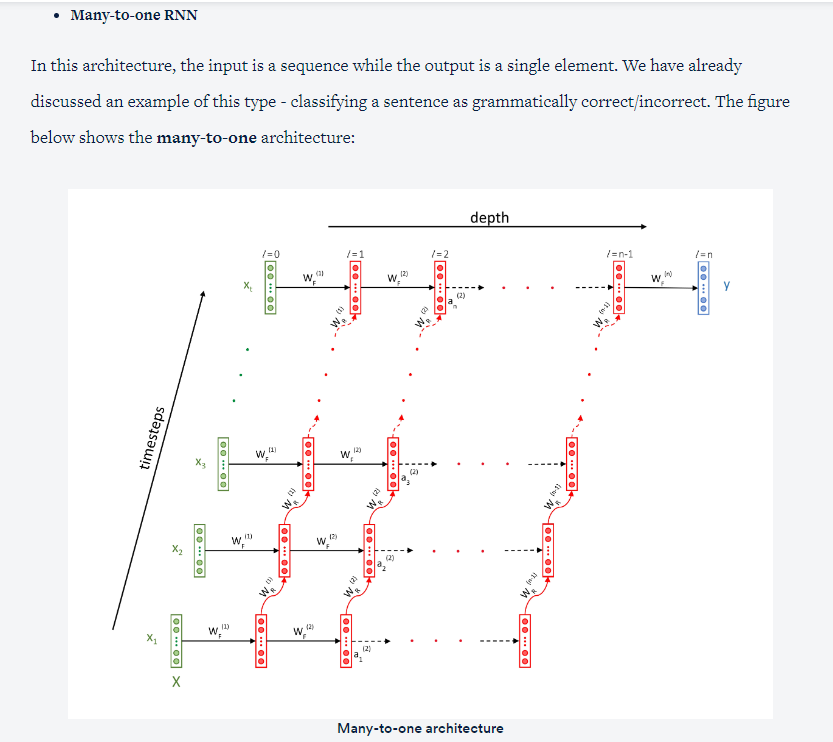






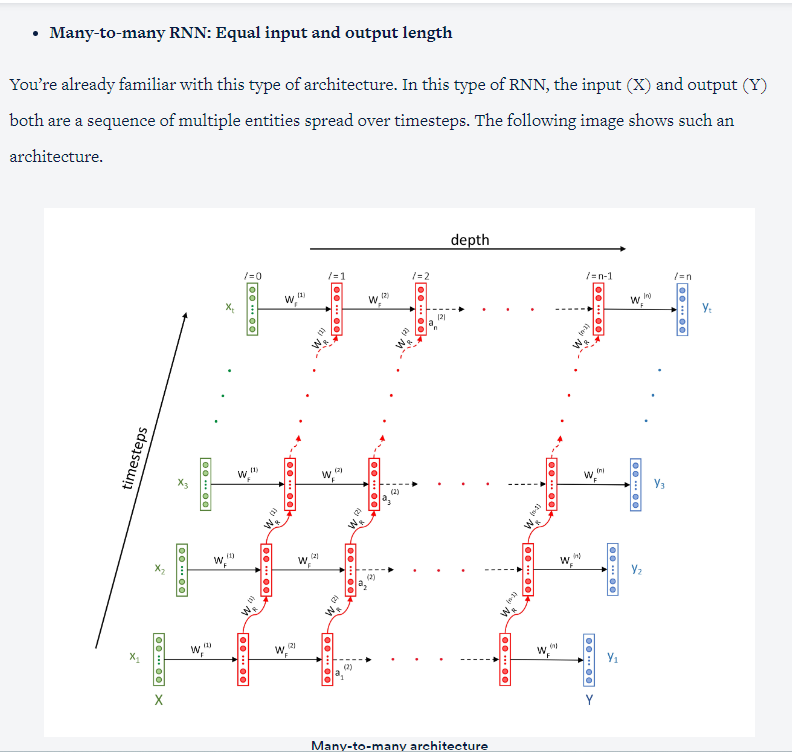


# Types of RNNs – 1



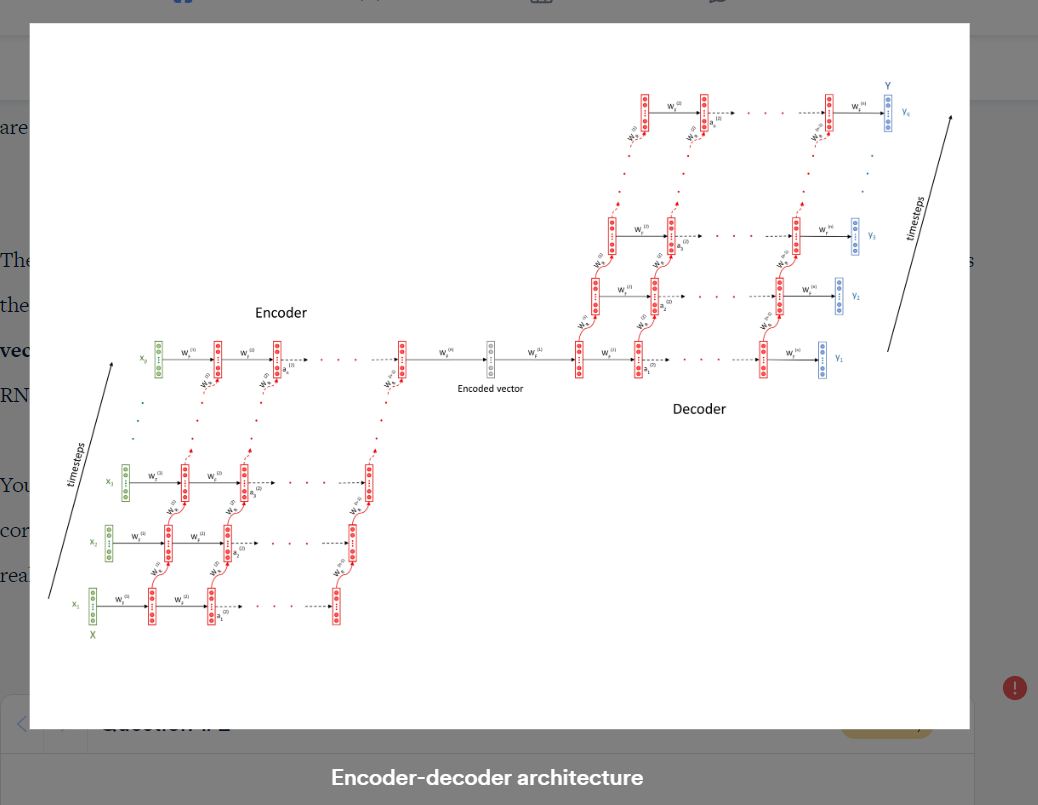
Some other examples of many-to-one problems are:

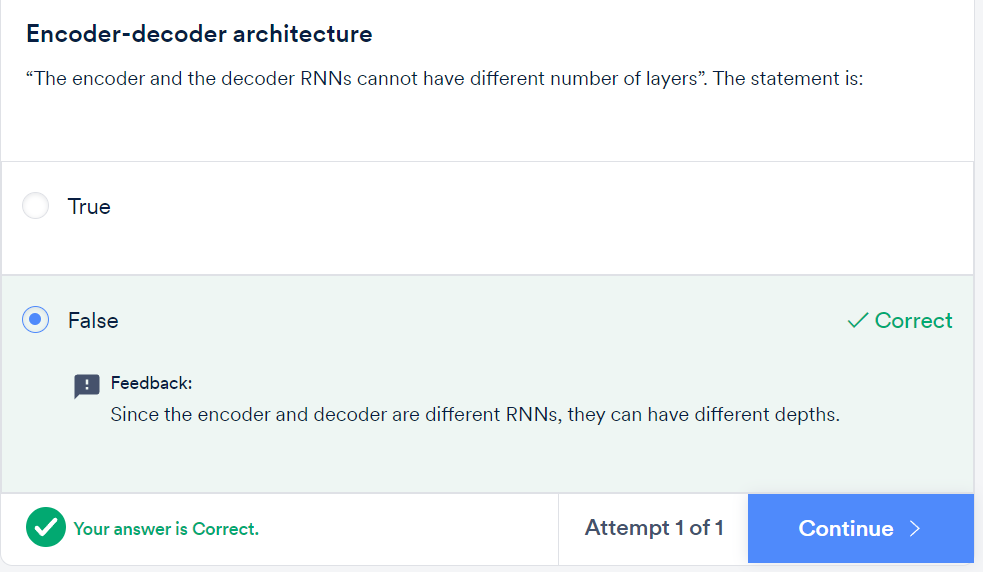
* Predicting the sentiment score of a text (between -1 to 1). For e.g., you can train an RNN to assign sentiment scores to customer reviews etc. Note that this can be framed as either a**regression problem** (where the output is a continuous number) or a**classification problem** (e.g. when the sentiment is positive/neutral/negative)
* Classifying videos into categories. For example, say you want to classify YouTube videos into two categories 'contains violent content / does not contain violence'. The output can be a single softmax neuron which predicts the probability that a video is violent.

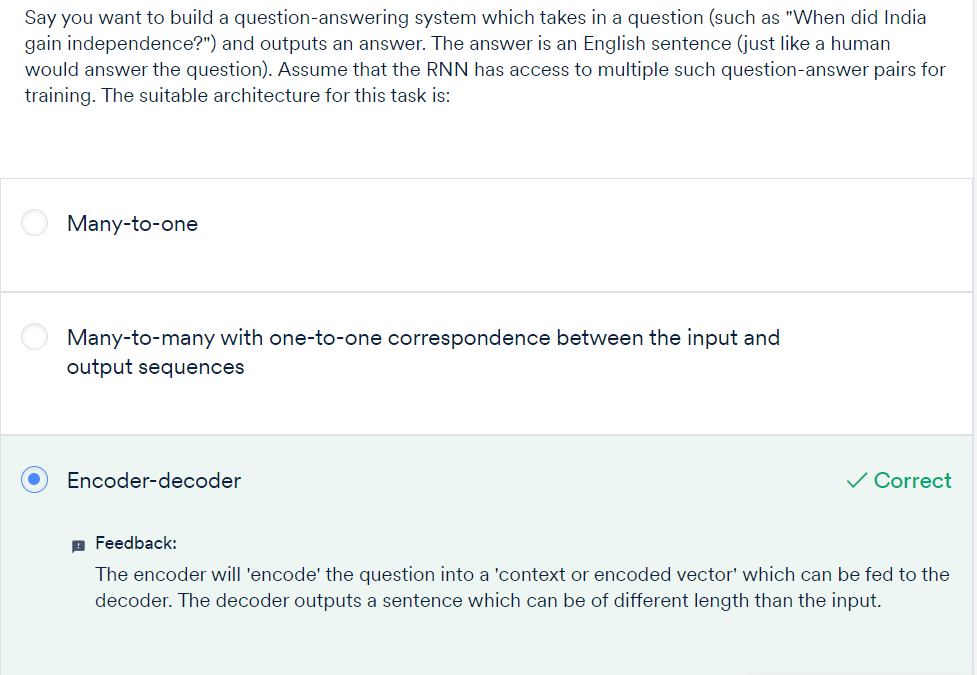


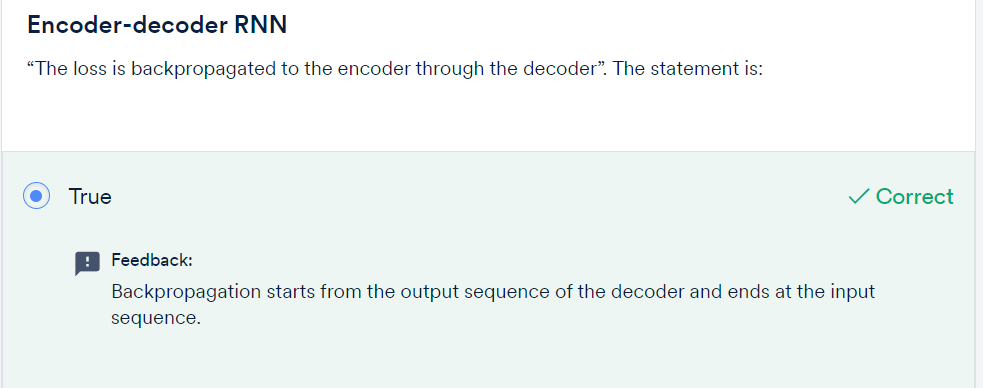
* **Many-to-many RNN: Unequal input and output lengths**

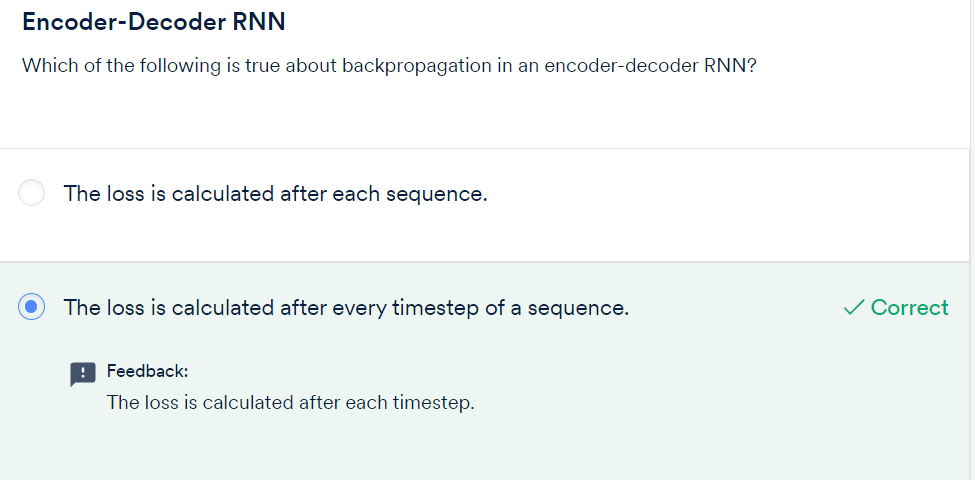
In the previous many-to-many example of POS tagging, we had assumed that the lengths of the input and output sequences are equal. However, this is not always the case. There are many problems where the **lengths of the input and output sequences are different**. For example, consider the task of **machine translation** - the length of a Hindi sentence can be different from the corresponding English sentence.

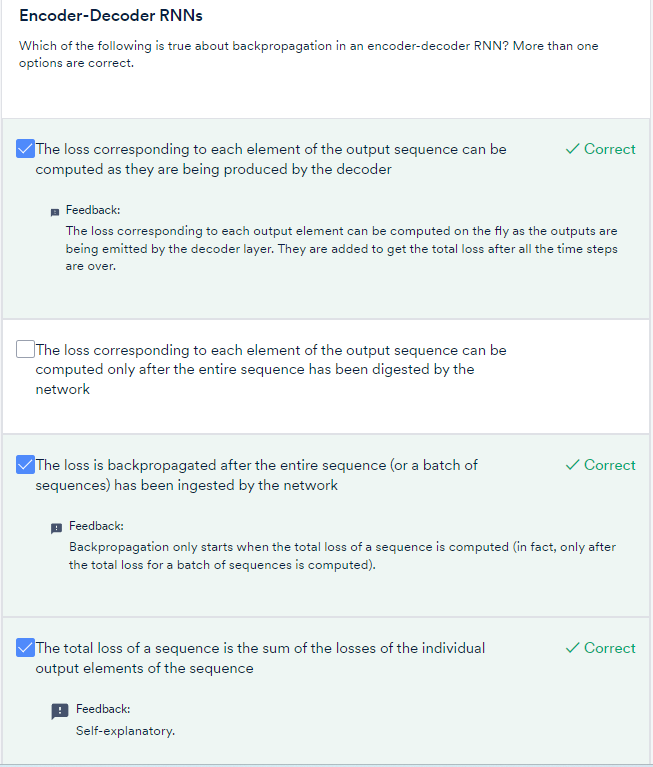


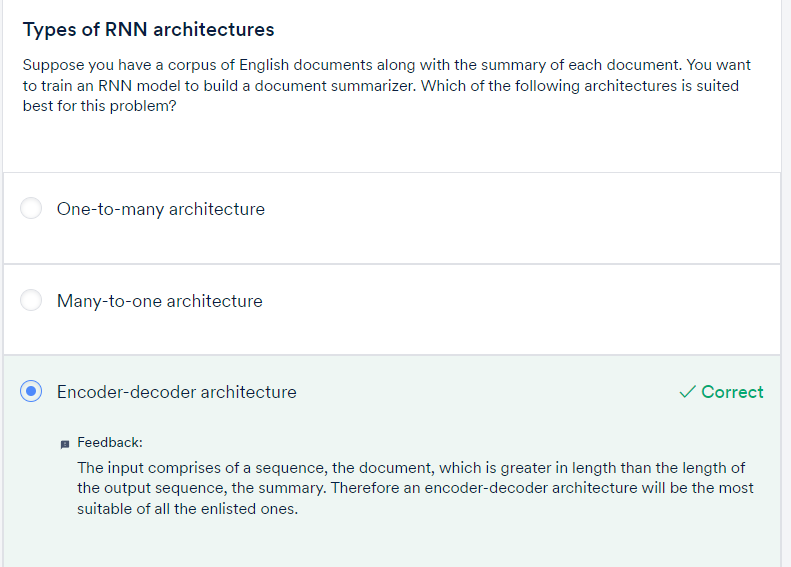


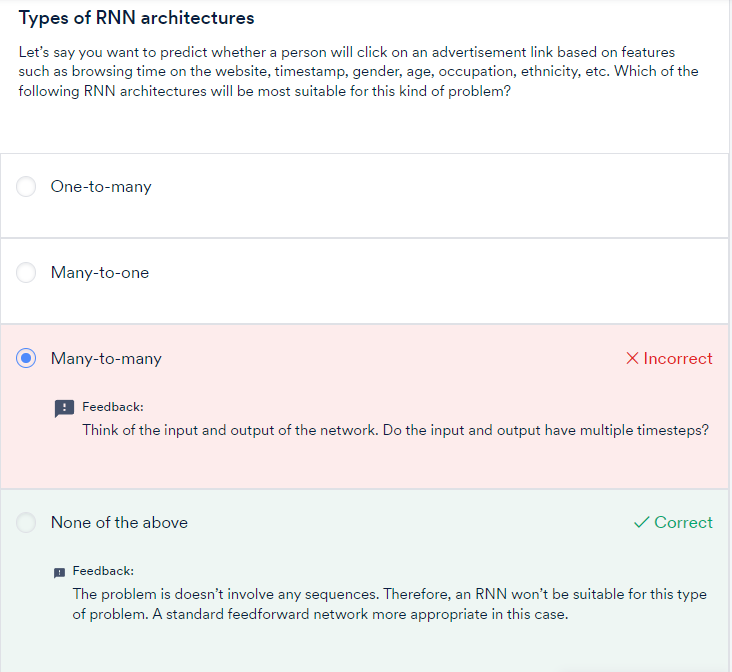


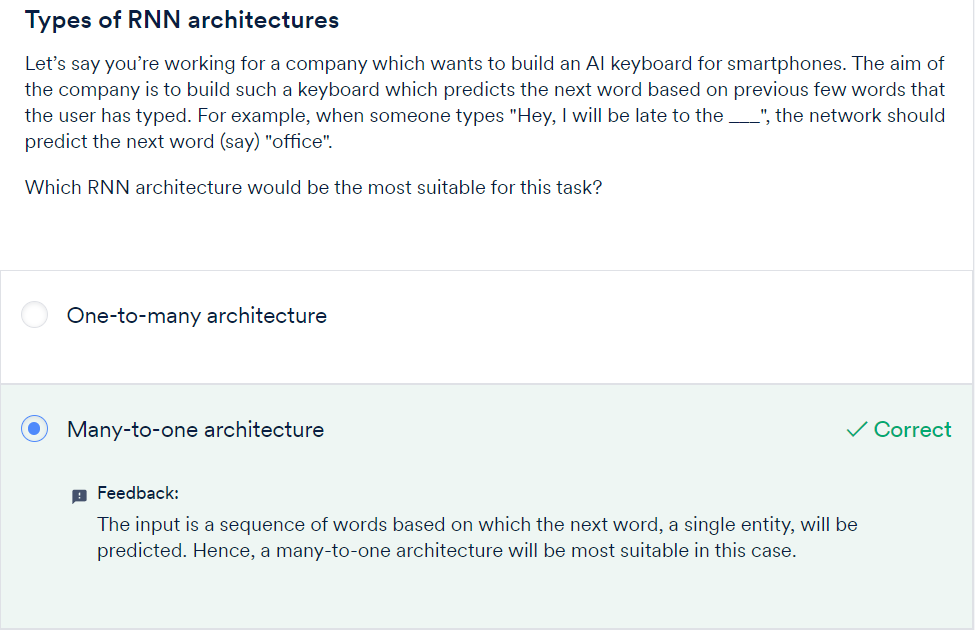


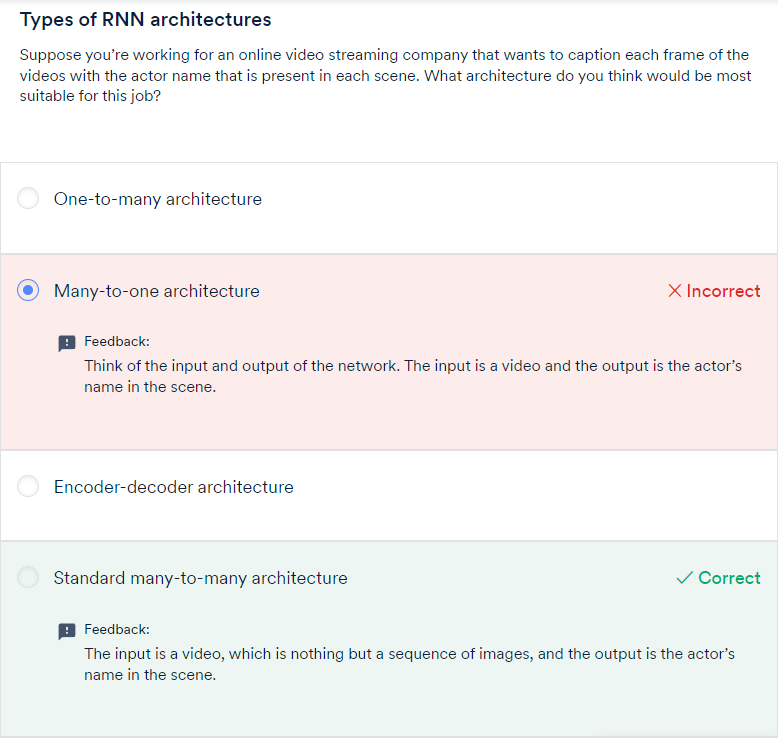












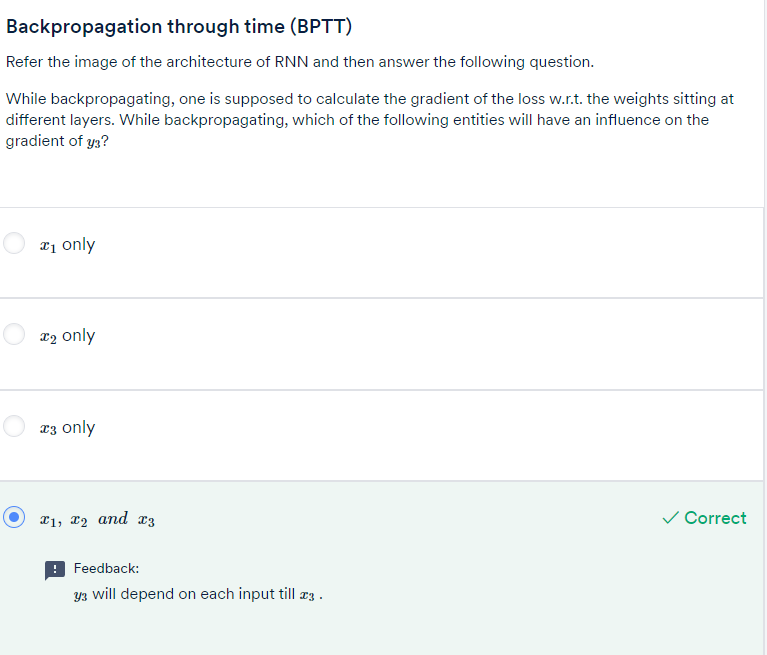
To summarise, these are the four main types of RNN architectures that you can use to build different kinds of applications:

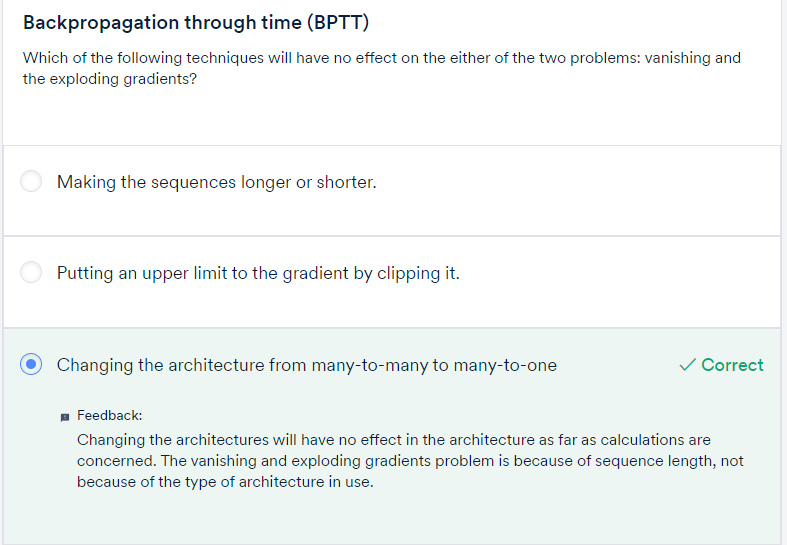
* Many-to-one
* Many-to-many (same input-output lengths)
* Many-to-many, or encoder-decoder (different input-output lengths)
* One-to-many (generative models)

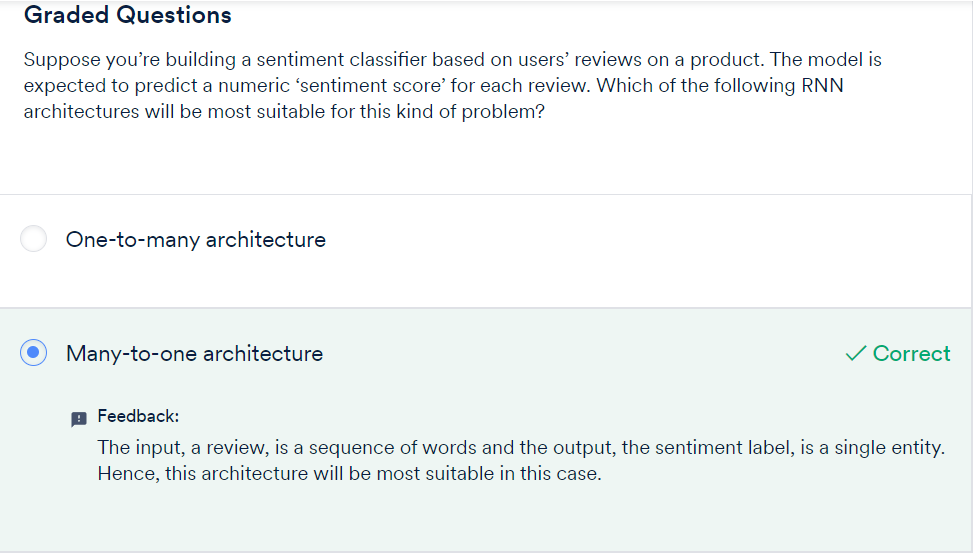
You could still use some workarounds to solve the problem of exploding gradients. You can impose an upper limit to the gradient while training, commonly known as **gradient clipping**. By controlling the maximum value of a gradient, you could do away with the problem of exploding gradients.

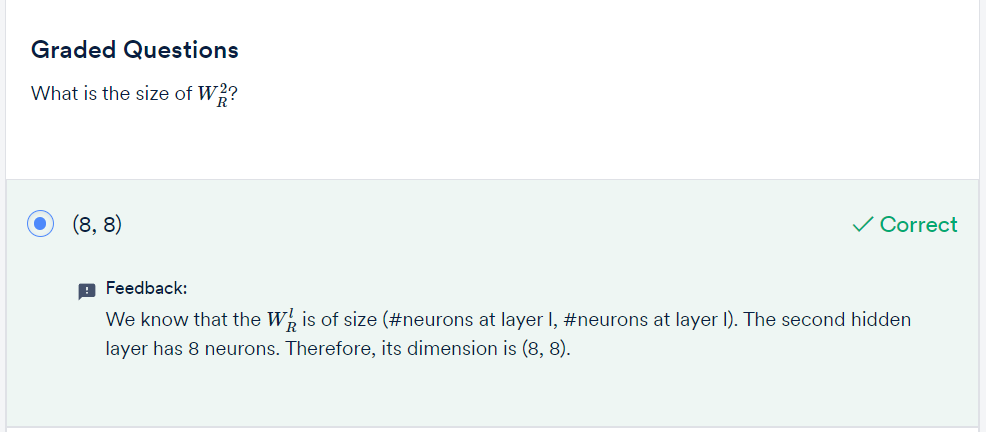
But the problem of vanishing gradients is a more serious one. The vanishing gradient problem is so rampant and serious in the case of RNNs that it renders RNNs useless in practical applications. One way to get rid of this problem is to use short sequences instead of long sequences. But this is more of a compromise than a solution - it restricts the applications where RNNs can be used.

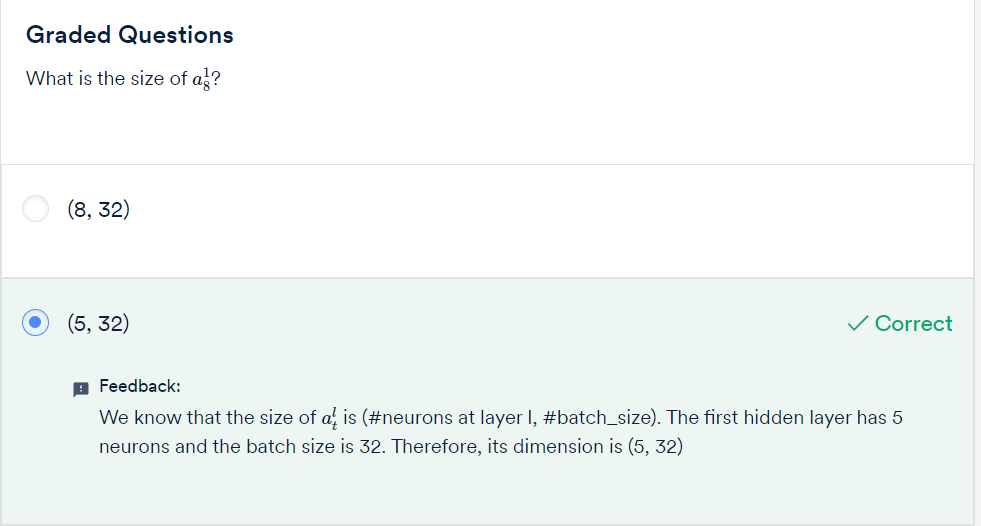
To get rid of the vanishing gradient problem, researchers have been tinkering around with the RNN architecture for a long while. The most notable and popular modifications are the **long short-term memory units** (**LSTMs**) and the **gated recurrent units**(**GRUs**).

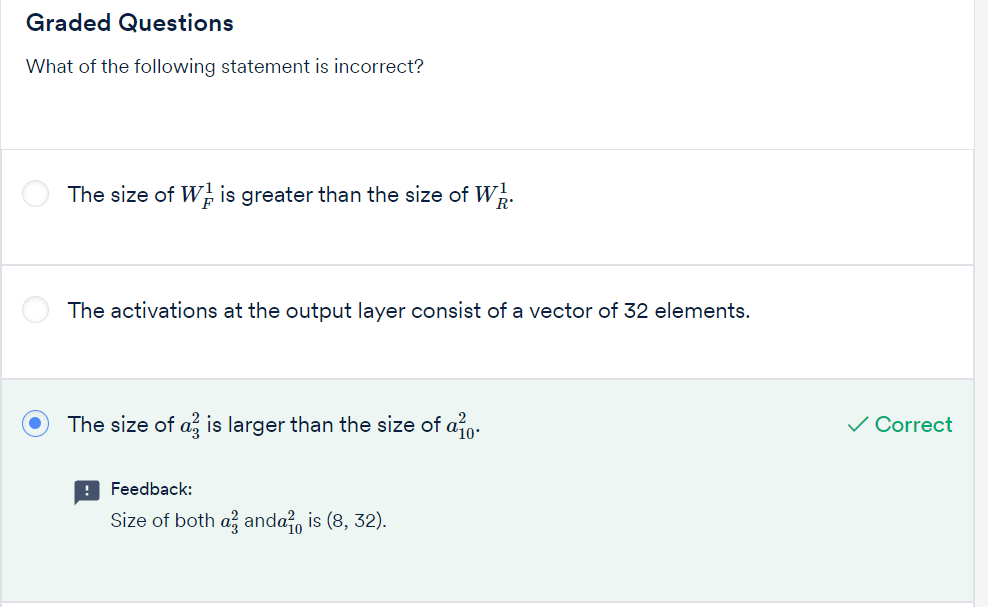


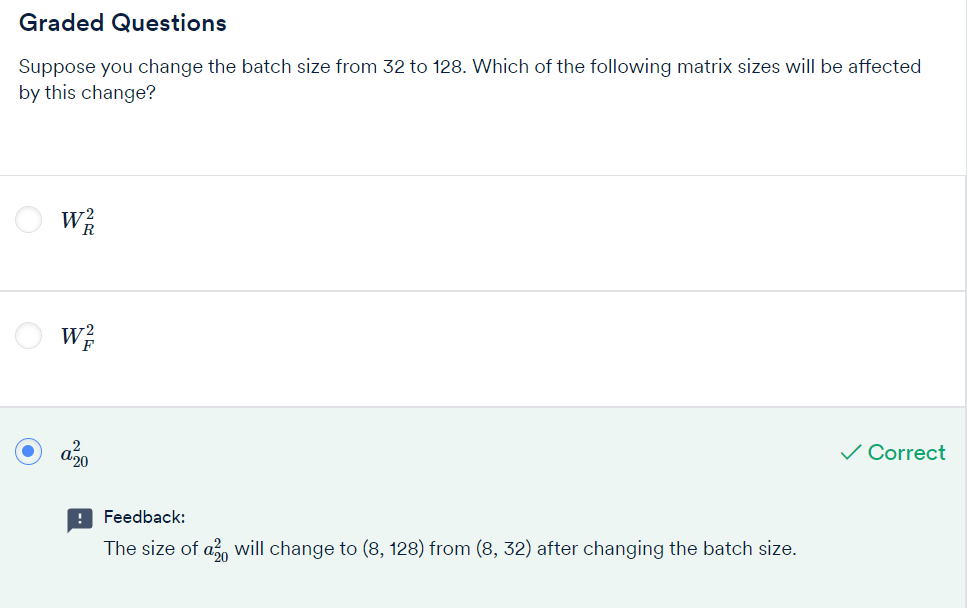












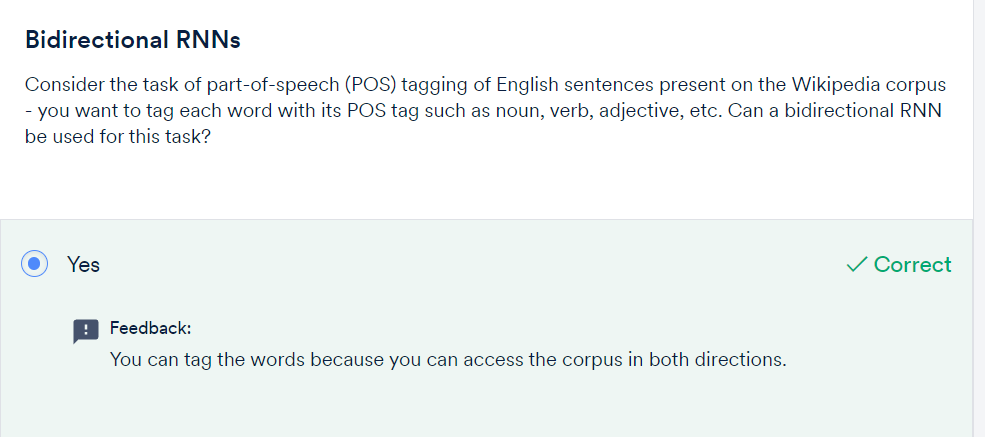
For example, when you want to **assign a sentiment score** to a piece of text (say a customer review), the network can see the entire review text before assigning them a score. On the other hand, in a task such as **predicting the next word** given previous few typed words, the network does not have access to the words in the future time steps while predicting the next word.

These **two types of tasks** are called **offline and online** **sequence processing**respectively.

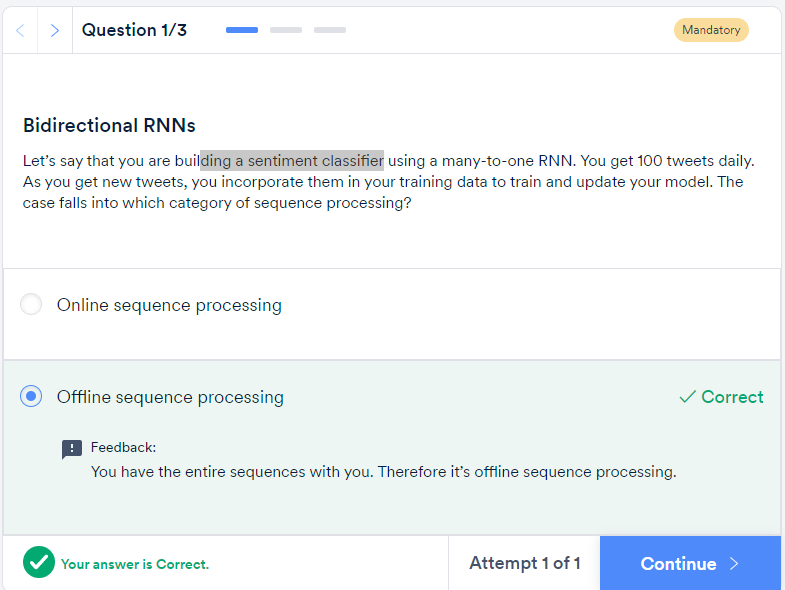
Thus, there are two types of sequences:

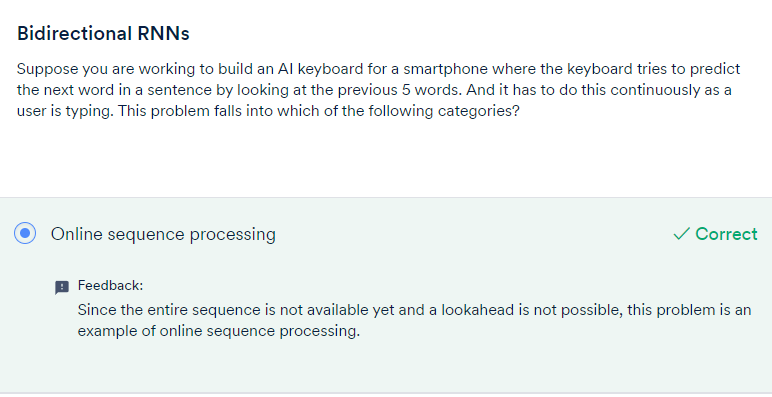
1. **Online sequence**: Here, you don’t have access to the entire sequence before you start processing it. The network has to make predictions as it sees each input coming in.
2. **Offline sequence**: The entire sequence is available before you start processing it.

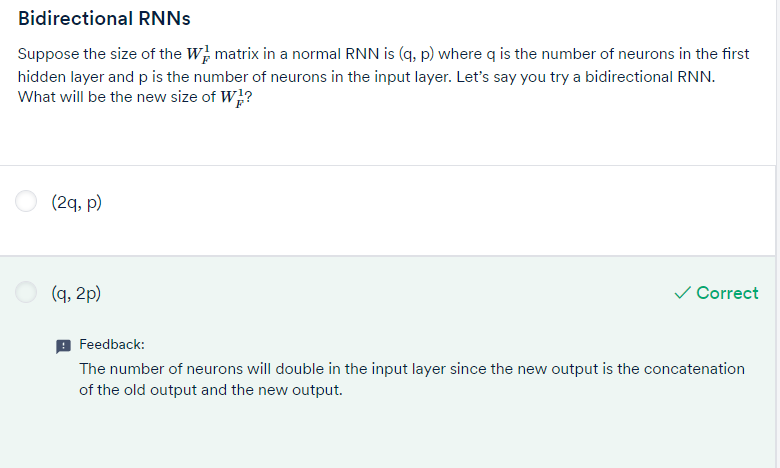
A bidirectional RNN can only be applied to **offline sequences**.



By using bidirectional RNNs, it is almost certain that you’ll get better results. However, bidirectional RNNs take almost double the time to train since the number of parameters of the network increase. Therefore, you have a tradeoff between training time and performance. The decision to use a bidirectional RNN depends on the computing resources that you have and the performance you are aiming for.







To solve the vanishing gradients problem, many attempts have been made to tweak the vanilla RNNs such that the gradients don’t die when sequences get long. The most popular and successful of these attempts has been the **long, short-term memory network**, or the **LSTM**. LSTMs have proven to be so effective that they have almost replaced vanilla RNNs.

The main drastic improvement that LSTMs have brought is because of a novel change in the **structure of a neuron** itself. In the case of LSTMs, the neurons are called cells, and an **LSTM cell**is different from a normal neuron in many ways.

Thus, one of the fundamental differences between an RNN and an LSTM is that an LSTM has an**explicit memory** **unit** which stores information relevant for learning some task. In the standard RNN, the only way the network remembers past information is through updating the hidden states over time, but it does not have an explicit memory to store information.

On the other hand, in LSTMs, the memory units retain pieces of information even when the sequences get really long.

The **gating mechanisms**allow modifying the state in certain ways. You’ll learn about how exactly the gating mechanisms work in the next section, though the main idea is that gating mechanisms **regulate the information** that the network stores (and passes on to the next layer) or forgets.

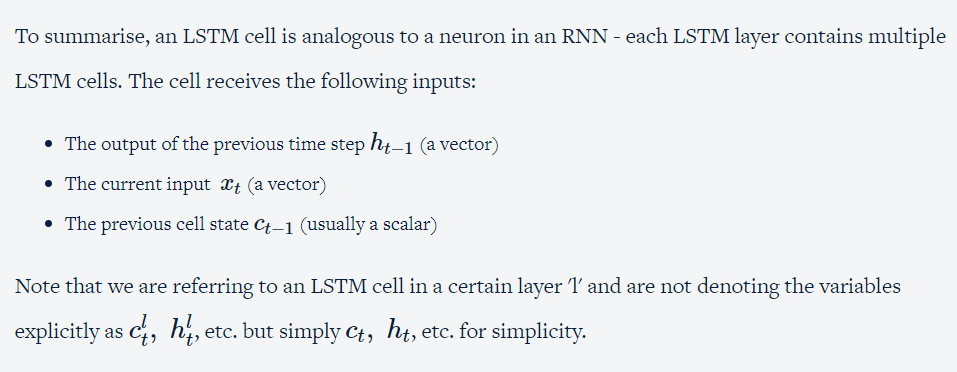
LSTM features

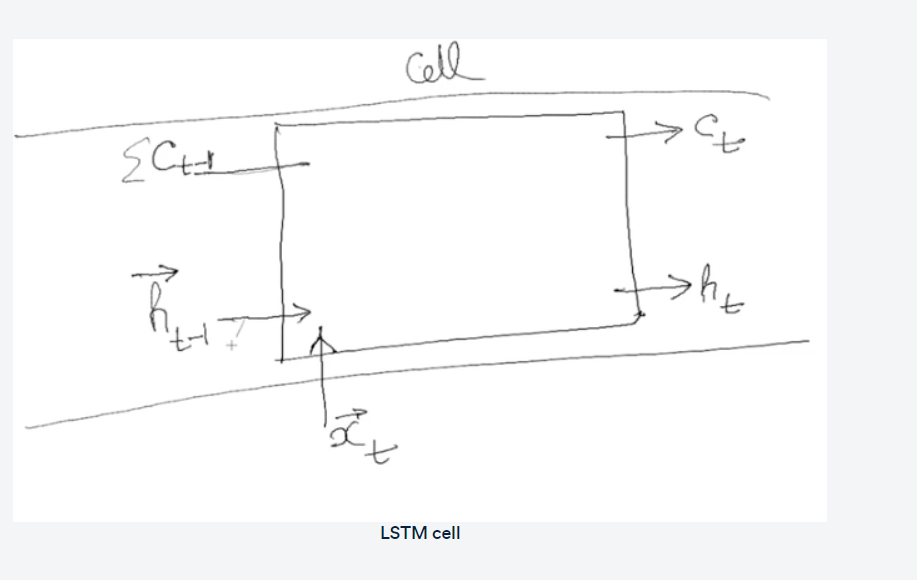
* 1. Cell state (memory)
  2. Gating mechanism – filter the information/ replace the info
  3. Get rid of vanishing Gradient – Constant Error Carrousel

The structure of an LSTM cell allows an LSTM network to have a smooth and uninterrupted flow of gradients while backpropagating. This flow is also called the **constant error carousel**. This third characteristic more or less a result of the first two characteristics and is the reason LSTMs are able to solve the problem of vanishing and exploding gradients.

To summarise, the LSTM is characterised by the following three main properties:

* The cells have an explicit 'memory'
* The gating mechanisms
* Constant error carousel

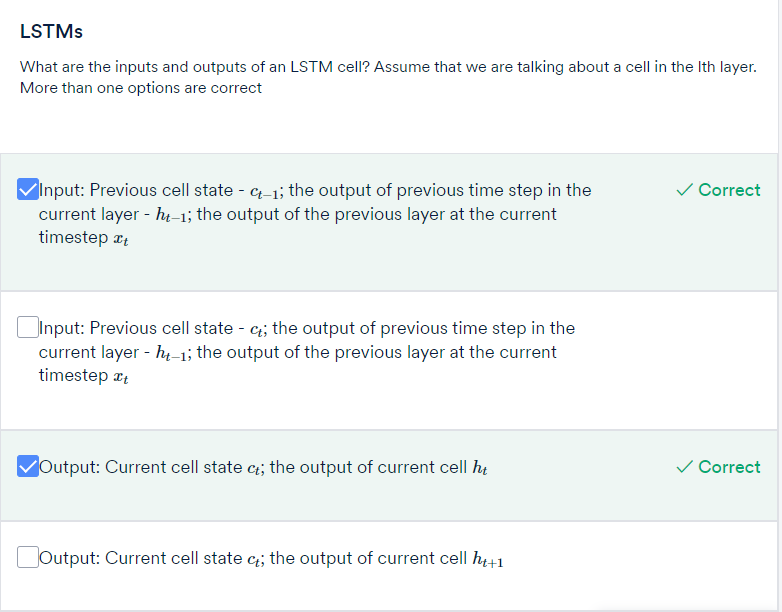


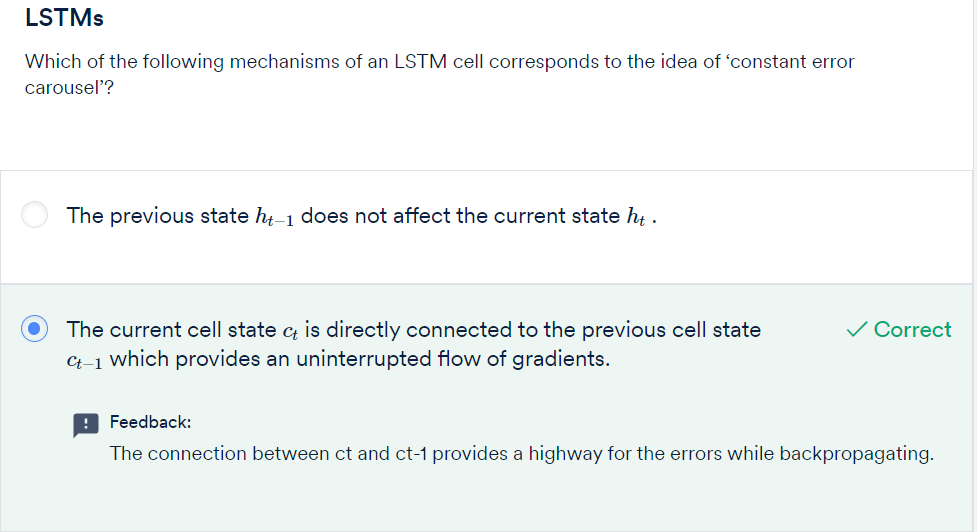


The cell produces two outputs:

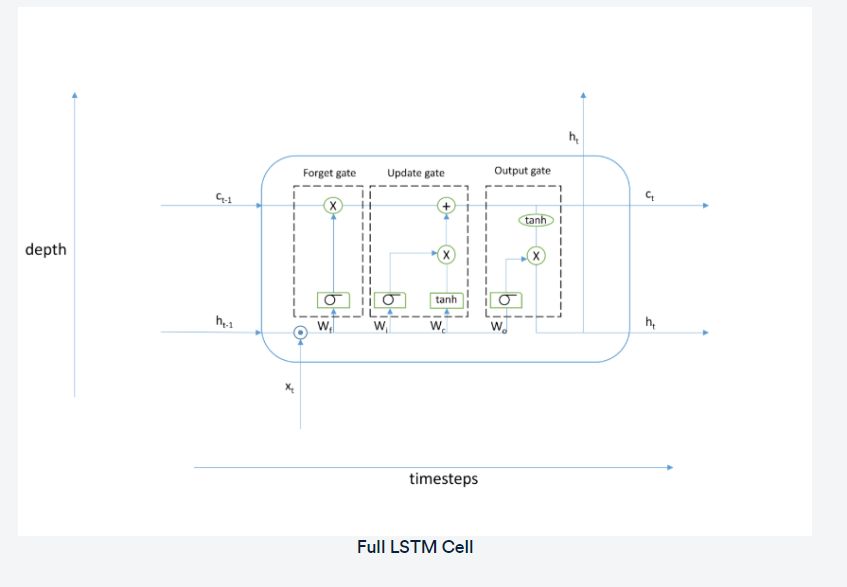
* The current cell state ct (a scalar)
* The current state output ht (a scalar)

Each cell in the LSTM layer will produce an output ht which will then be combined to form a vector and fed to the next LSTM layer.





You saw the structure of the LSTM cell. You also saw the three gating mechanisms - the **forget gate**, the **update gate**and the **output gate.**

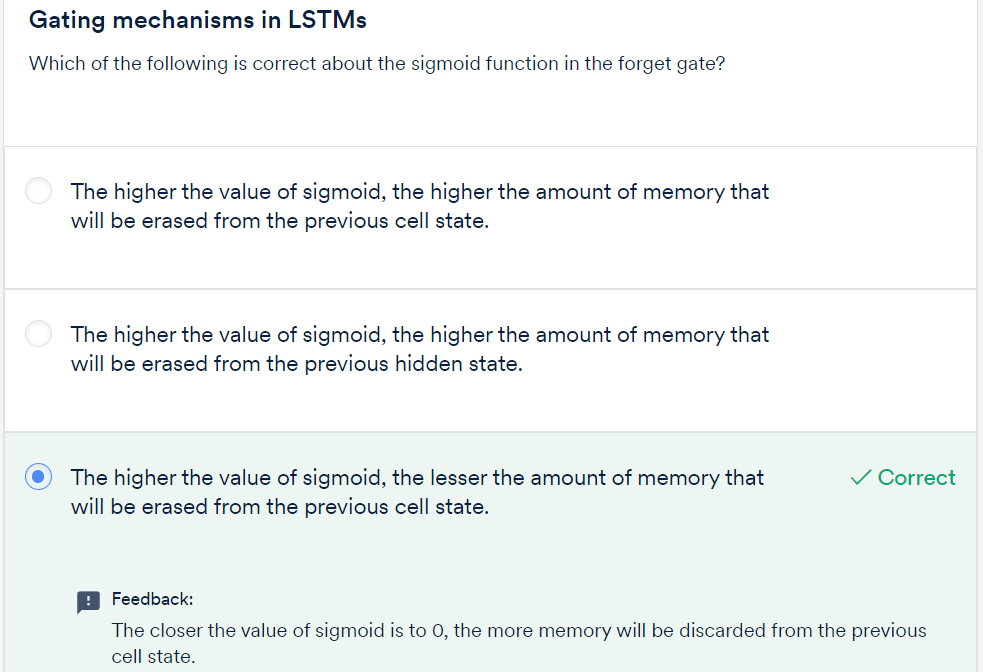


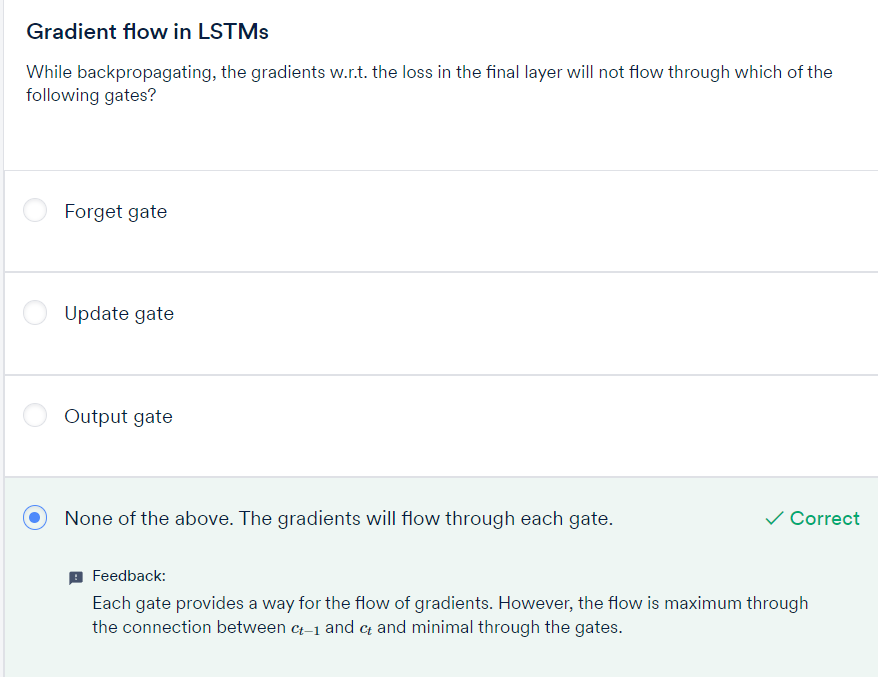
Let’s understand the intuition of each gate with a specific example. Let’s say you’re working on a **video tagging problem** where you need to tag the action that takes place in each frame of the video. Let’s look at the function of each gate in the context of this problem:

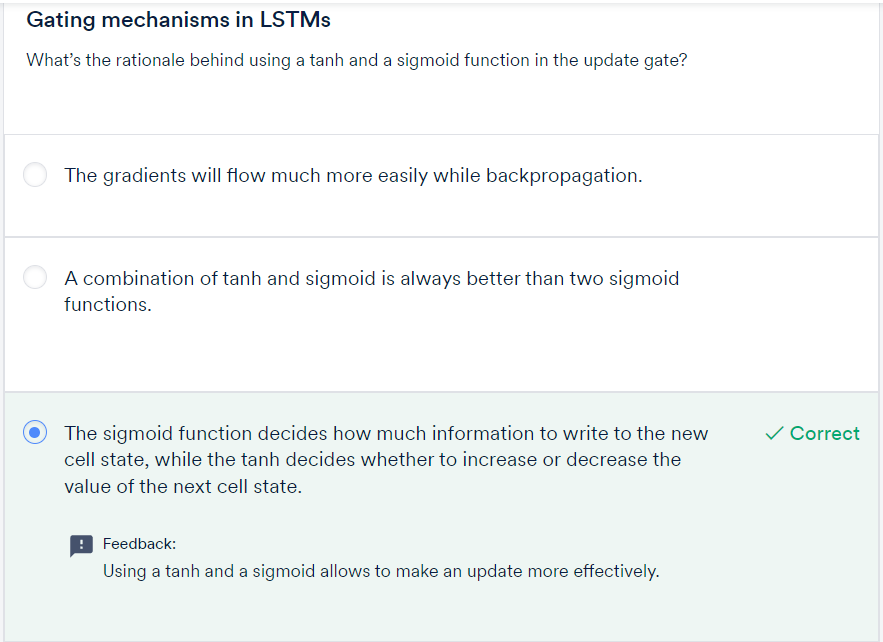
* **Forget gate**: This gate controls how much information needs to be discarded from the previous cell state (ct−1) depending on the new input. In the video tagging problem, whenever a new action takes place, this gate needs to decide how much information to retain from the previous frames. If the same action is happening over and over again, then very less information should be discarded. When the action changes, the forget gate 'forgets' a lot of information.
* **Update gate**: This gate makes an **update to the previous the cell state** by writing a new piece of information to it. In the video tagging problem, when the action changes, this gate will update the cell state with information relevant to the new action. In case the action is the same as the previous frame, negligible information will be written to the cell state. If the scene changes drastically, the update will be drastic too.

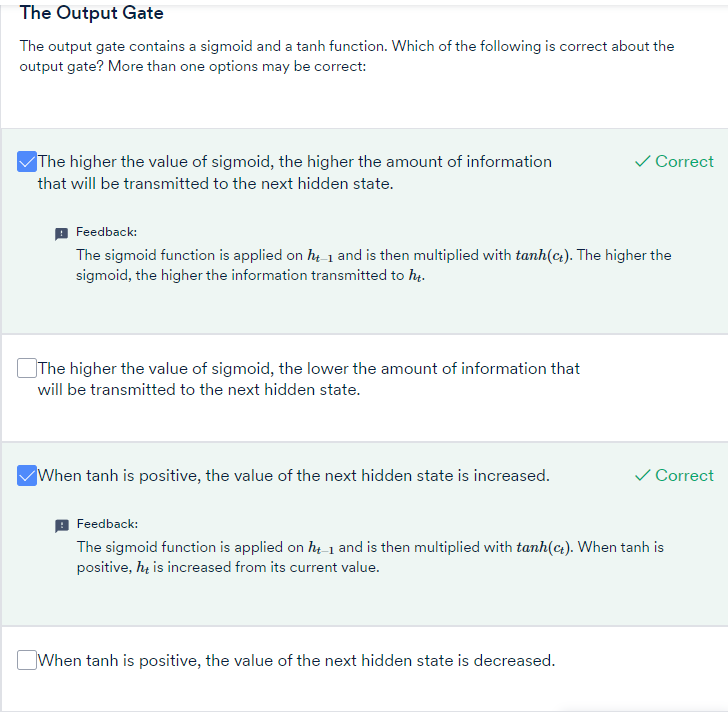
The new cell state ct is the cumulative result of the information discarded from ct−1 by the forget gate and the new information freshly updated to ct−1 by the update gate.

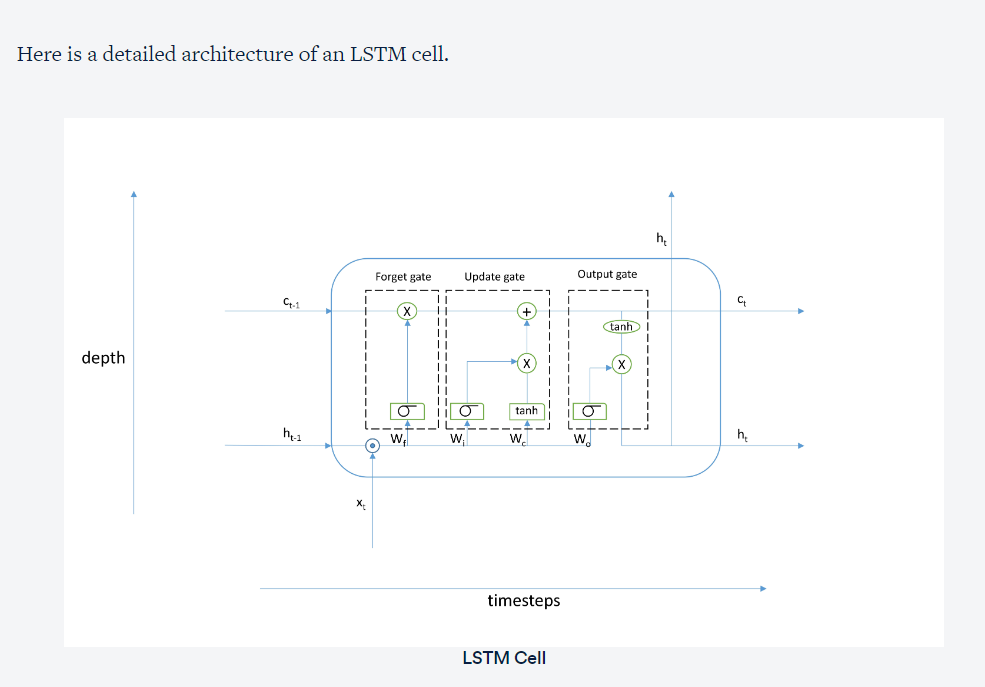
* **Output gate**: This gate controls how much information needs to be passed on to the next LSTM layer based on the current cell state.



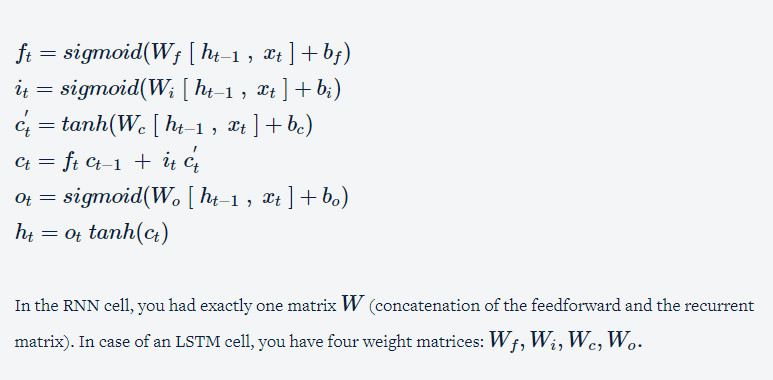


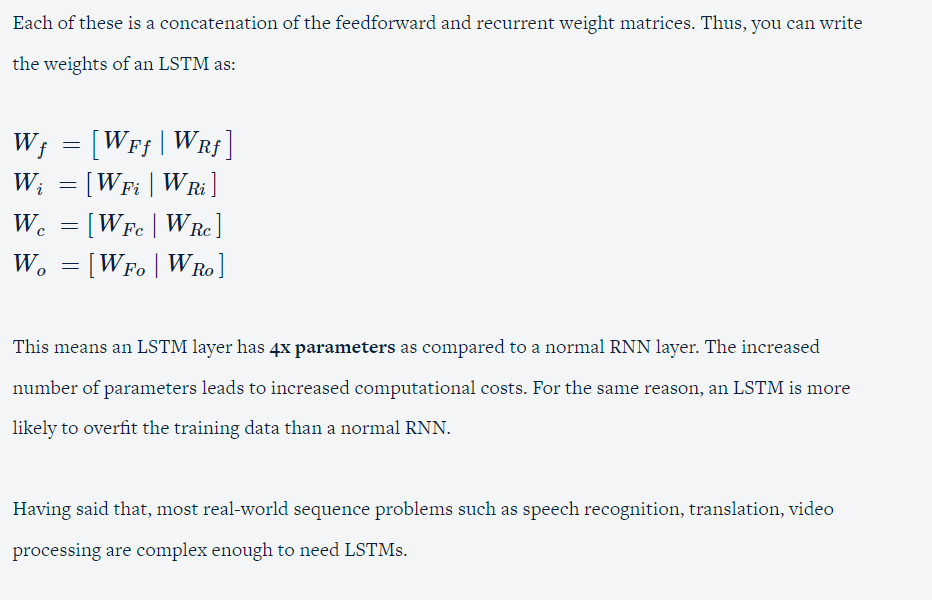


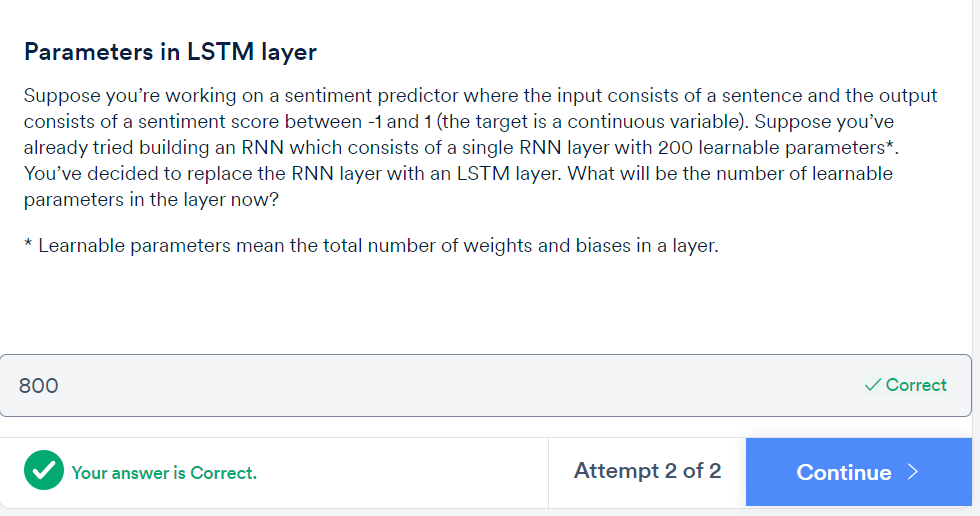




In feedforward, first the previous activations ht−1 and the current input xt get concatenated (shown by the dot operator). The concatenated vector goes into each of the three gates. The 'x' denotes element-wise multiplication while the '+' denotes element-wise addition between two vectors/matrices. Note that the output gate has another tanh function though it is not a gate (there are no weights involved in that operation, as shown in the figure).







Q&A

# Summary

In this session, you studied some commonly used variants of RNNs.

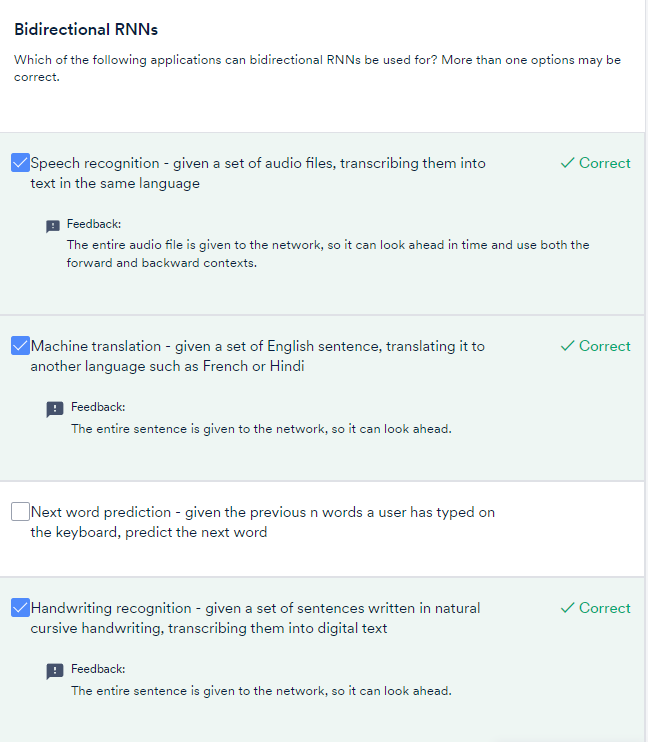
First, you studied **bidirectional RNNs**. You learnt that in case of offline sequences, you can reverse the order of a sequence. You can use bidirectional RNNs to feed sequences in regular as well as in the reverse order. This gives you better result in most cases.

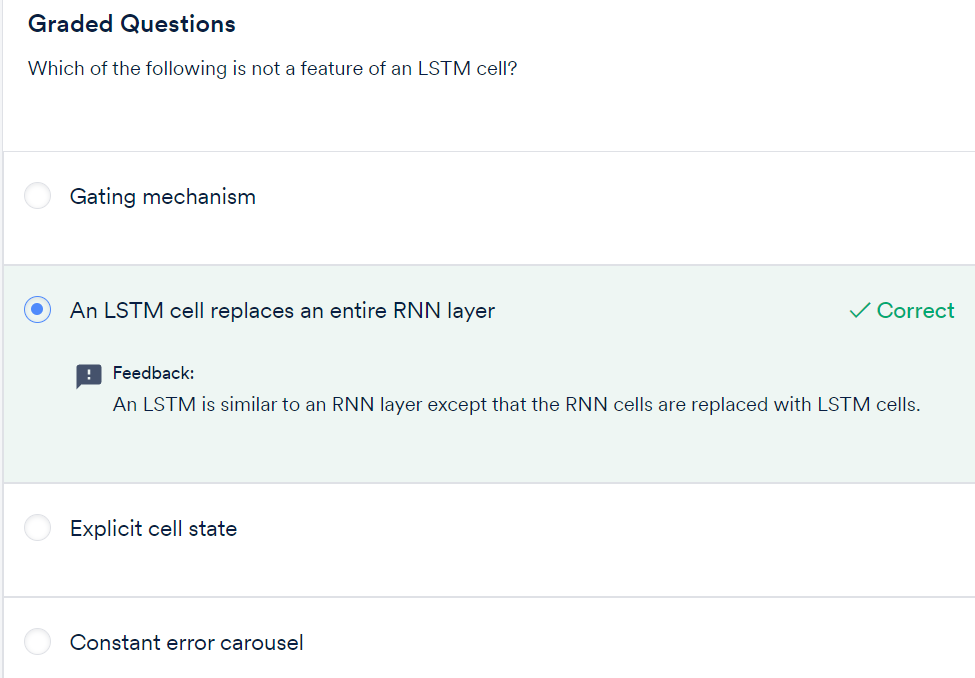
Then you learnt about **long short-term memory networks or LSTMs.** You learnt the three features of an LSTM cell:

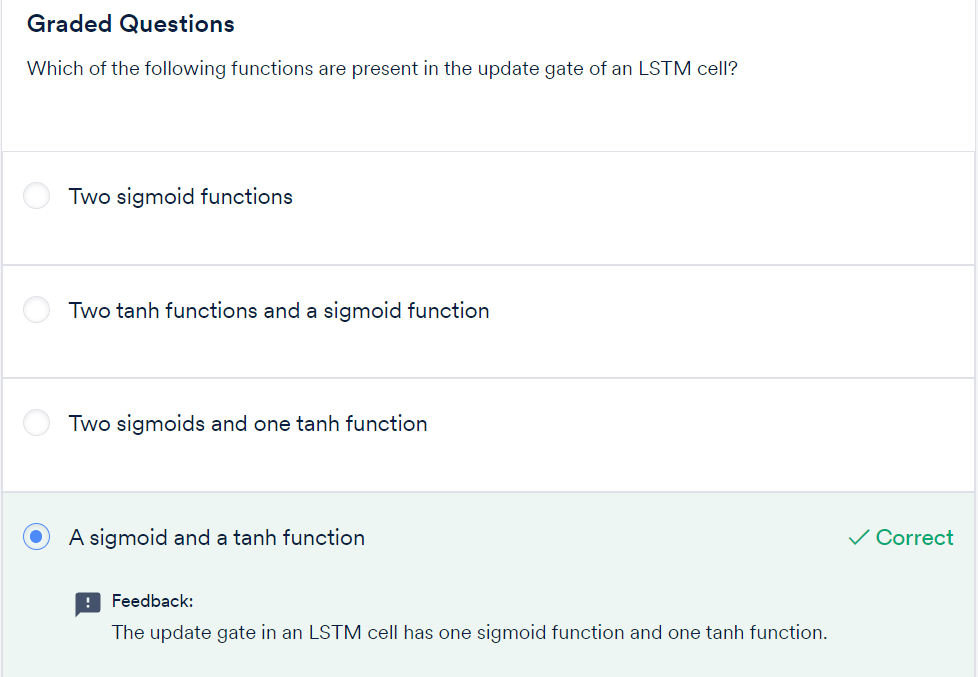
* Presence of an explicit memory
* Gating mechanisms
* Constant error carousel

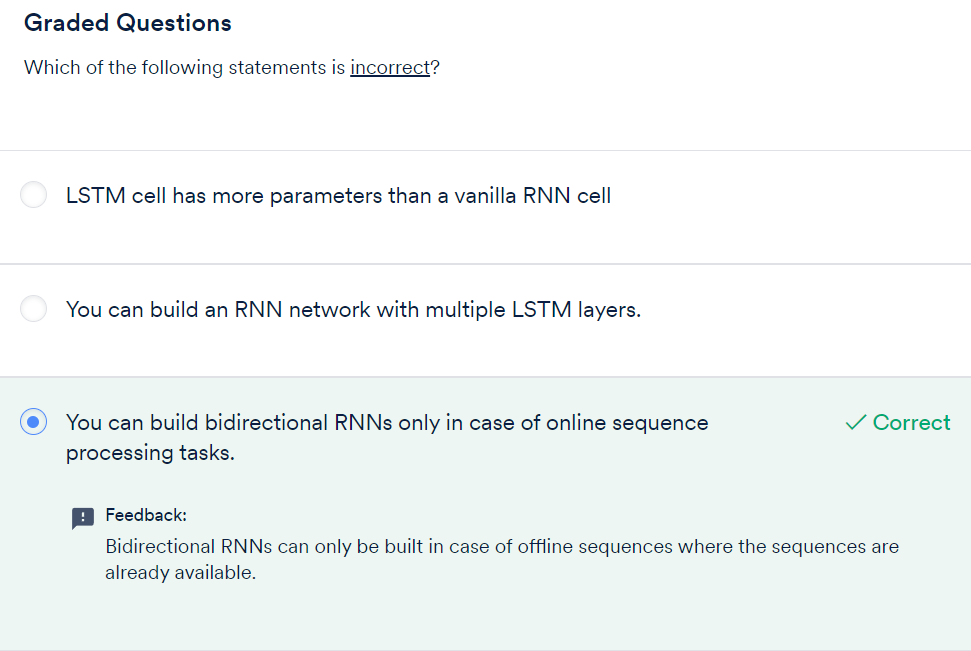
You also learnt the structure of an LSTM cell and its feed forward equations. The number of parameters in an LSTM layer are **4x** the number of parameters in a standard RNN.

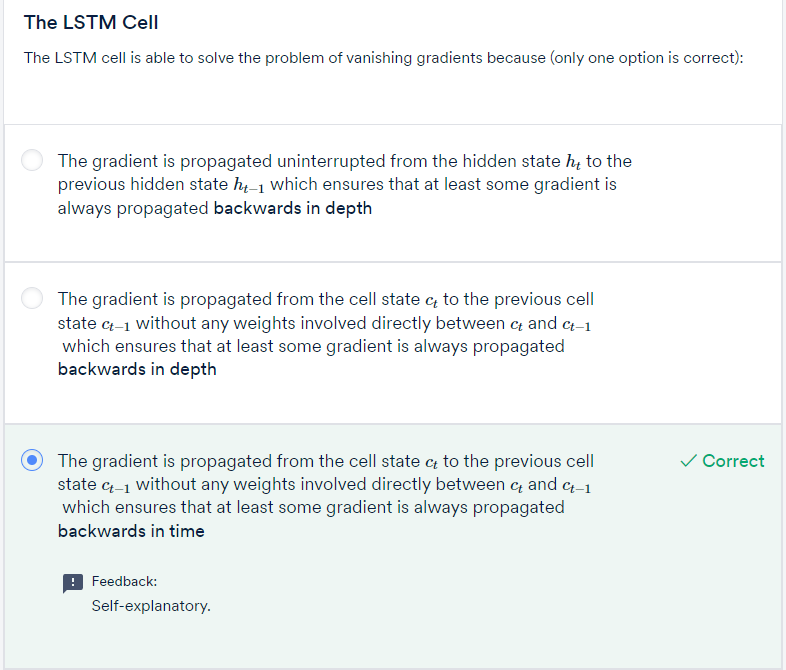
Finally, you briefly looked at some other variants of LSTM such as **GRUs** and LSTMs with peephole connections. The number of parameters in a GRU layer are **3x** the number of parameters in a standard RNN layer.











# BUILDING RNNs IN PYTHON: