Introducing RNNs

The Sun rises in the ____

If we were asked to predict the blank term in the preceding sentence, we would probably say east. Why would we predict that the word east would be the right word here? Because we read the whole sentence, understood the context, and predicted that the word east would be an appropriate word to complete the sentence.

If we use a feedforward neural network to predict the blank, it would not predict the right word. This is due to the fact that in feedforward networks, each input is independent of other input and they make predictions based only on the current input, and they don't remember previous input.

Thus, the input to the network will just be the word preceding the blank, which is the word *the*. With this word alone as an input, our network cannot predict the correct word, because it doesn't know the context of the sentence, which means that it doesn't know the previous set of words to understand the context of the sentence and to predict an appropriate next word.

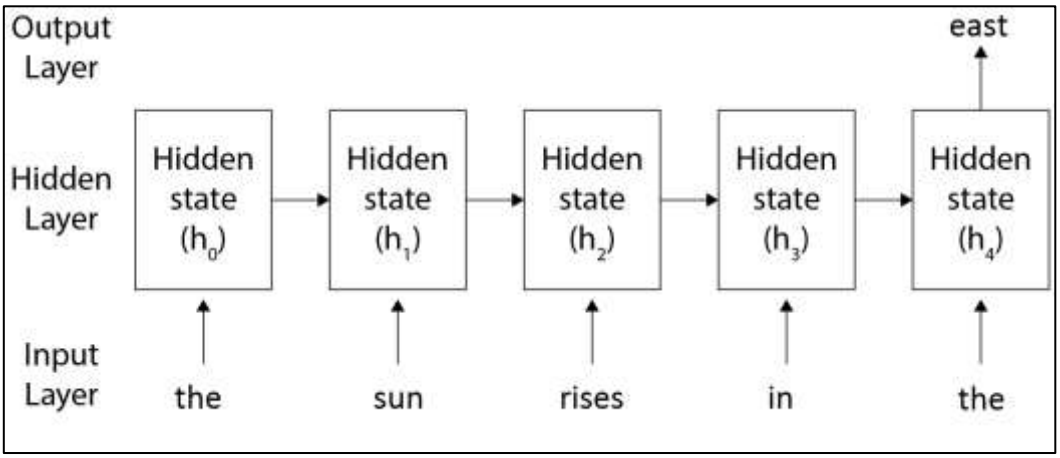
Here is where we use RNNs. They predict output not only based on the current input, but also on the previous hidden state. Why do they have to predict the output based on the current input and the previous hidden state? Why can't they just use the current input and the previous input?

This is because the previous input will only store information about the previous word, while the previous hidden state will capture the contextual information about all the words in the sentence that the network has seen so far. Basically, the previous hidden state acts like a memory and it captures the context of the sentence. With this context and the current input, we can predict the relevant word.

For instance, let's take the same sentence, *The sun rises in the* ____. As shown in the following figure, we first pass the word *the* as an input, and then we pass the next word, *sun*, as input; but along with this, we also pass the previous hidden state, h_0. So, every time we pass the input word, we also pass a previous hidden state as an input.

In the final step, we pass the word *the* and also the previous hidden state h3, which captures the contextual information about the sequence of words that the network has seen so far. Thus, h3 acts as the memory and stores information

about all the previous words that the network has seen. With h3 and the current input word (*the*), we can predict the relevant next word:

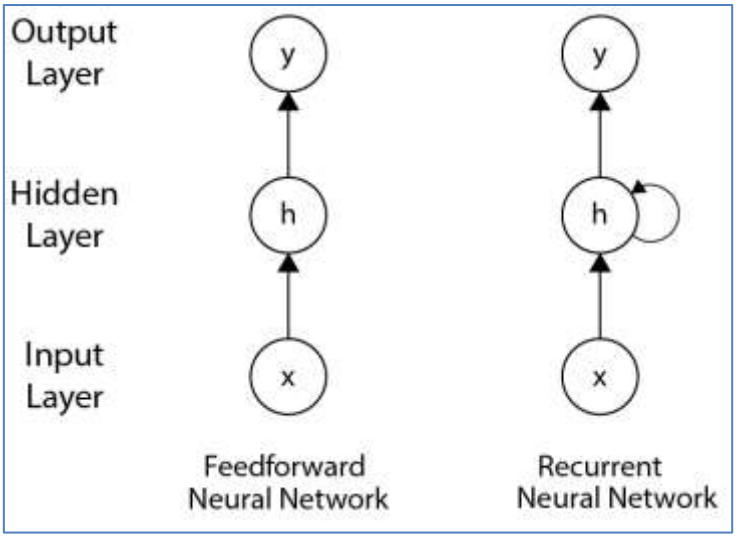


In a nutshell, an RNN uses the previous hidden state as memory which captures and stores the contextual information (input) that the network has seen so far.

RNNs are widely applied for use cases that involve sequential data, such as time series, text, audio, speech, video, weather, and much more. They have been greatly used in various **natural language processing** (**NLP**) tasks, such as **language translation, sentiment analysis, text generation**, and so on.

The difference between feedforward networks and RNNs

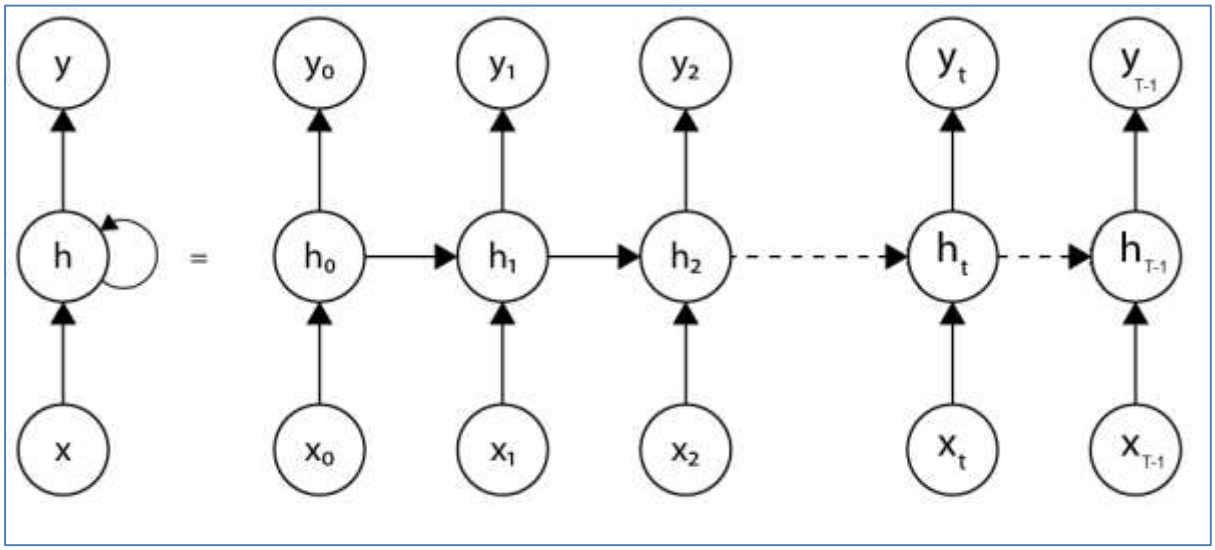
A comparison between an RNN and a feedforward network is shown in the following diagram:



As you can observe in the preceding diagram, the RNN contains a looped connection in the hidden layer, which implies that we use the previous hidden state along with the input to predict the output.

Still confused? Let's look at the following unrolled version of an RNN. But wait; what is the unrolled version of an RNN?

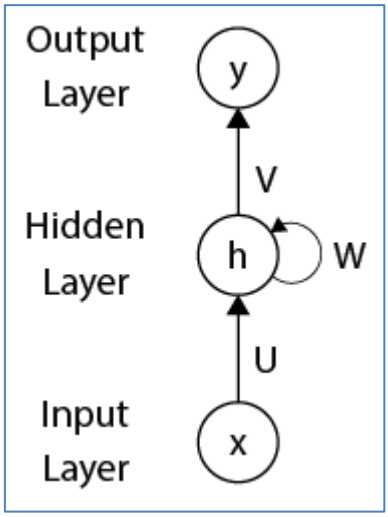
It means that we roll out the network for a complete sequence. Let's suppose that we have an input sentence with **T** words; then, we will have **0** to **T-1** layers, one for each word, as shown in the following figure:



As you can see in the preceding figure, at the time step **t = 1**, the output **y1** is predicted based on the current input **x1** and the previous hidden state **h0**. Similarly, at time step **t=2**, **y2** is predicted using the current input **x2** and the previous hidden state **h1**. This is how an RNN works; it takes the current input and the previous hidden state to predict the output.

Forward propagation in RNNs

Let's look at how an RNN uses forward propagation to predict the output; but before we jump right in, let's get familiar with the notations:



The preceding figure illustrates the following:

- U represents the input to hidden layer weight matrix
- W represents the hidden to hidden layer weight matrix
- V represents the hidden to output layer weight matrix

The hidden state h at a time step t can be computed as follows:

$$h_t = \tanh(Ux_t + Wh_{t-1})$$

That is, *hidden state at a time step, t = tanh ([input to hidden layer weight x input] + [hidden to hidden layer weight x previous hidden state]).*

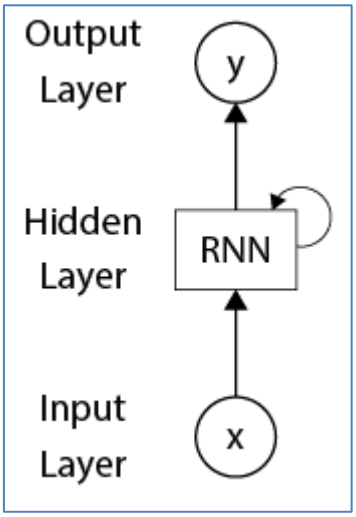
The output at a time step t can be computed as follows:

$$\hat{y}_t = \text{softmax}(Vh_t)$$

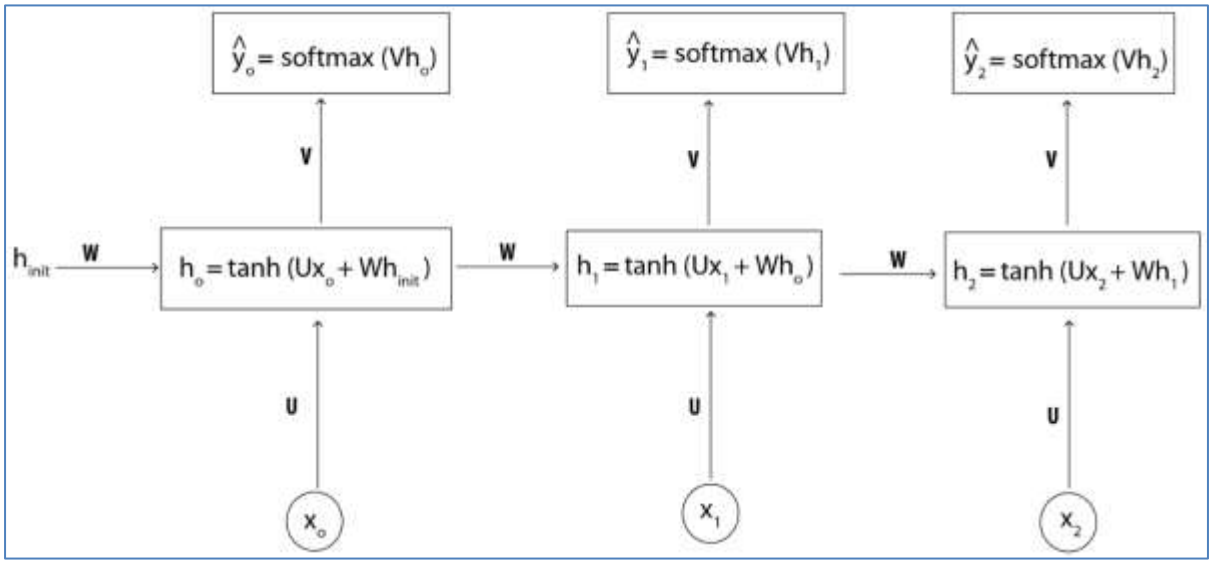
That is, *output at a time step, t = softmax (hidden to output layer weight* x *hidden state at a time t).*

We can also represent RNNs as shown in the following figure. As you can see, the hidden layer is represented by an RNN block, which implies that

our network is an RNN, and previous hidden states are used in predicting the output:



The following diagram shows how forward propagation works in an unrolled version of an RNN:



We initialize the initial hidden state h_{init} with random values. As you can see in the preceding figure, the output, $\hat{y}_0$, is predicted based on the current input, x_0 and the previous hidden state, which is an initial hidden state, h_{init}, using the following formula:

$$h_0 = \tanh(Ux_0 + Wh_{init})$$

$$\hat{y}_0 = \text{softmax}(Vh_0)$$

Similarly, look at how the output, $\hat{y}_1$, is computed. It takes the current Input, x_1, and the previous hidden state, h_0:

$$\boxed{h_1 = \tanh(Ux_1 + Wh_0)}$$
$$\boxed{\hat{y}_1 = \text{softmax}(Vh_1)}$$

Thus, in forward propagation to predict the output, RNN uses the current input and the previous hidden state.

Back propagating through time

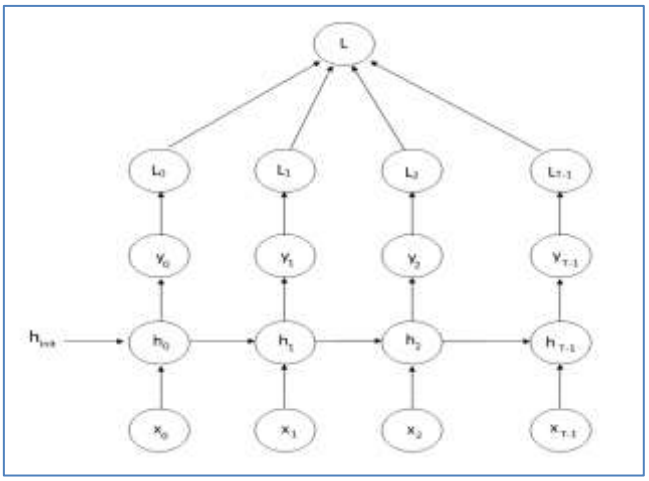
We just learned how forward propagation works in RNNs and how it predicts the output. Now, we compute the loss, L, at each time step, t, to determine how well the RNN has predicted the output. We use the **cross-entropy loss** as our loss function. The loss L at a time step t can be given as follows:

$$\boxed{L_t = -y_t \log(\hat{y}_t)}$$

Here, y_t is the actual output, and $\hat{y}_t$ is the predicted output at a time step t. The final loss is a sum of the loss at all the time steps. Suppose that we have $T-1$ layers; then, the final loss can be given as follows:

$$\boxed{L = \sum_{j=0}^{T-1} L_j}$$

As shown in the following figure, the final loss is obtained by the sum of loss at all the time steps:



We computed the loss, now our goal is to minimize the loss. How can we minimize the loss? We can minimize the loss by finding the optimal weights of the RNN. As we learned, we have three weights in RNNs: input to hidden, U, hidden to hidden, W, and hidden to output, V.

We need to find optimal values for all of these three weights to minimize the loss. We can use gradient descent algorithm to find the optimal weights. We begin by calculating the gradients of the loss function with respect to all the weights; then, we update the weights according to the weight update rule as follows:

$$V = V - \alpha \frac{\partial L}{\partial V}$$

$$W = W - \alpha \frac{\partial L}{\partial W}$$

$$U = U - \alpha \frac{\partial L}{\partial U}$$

Vanishing and exploding gradients problem

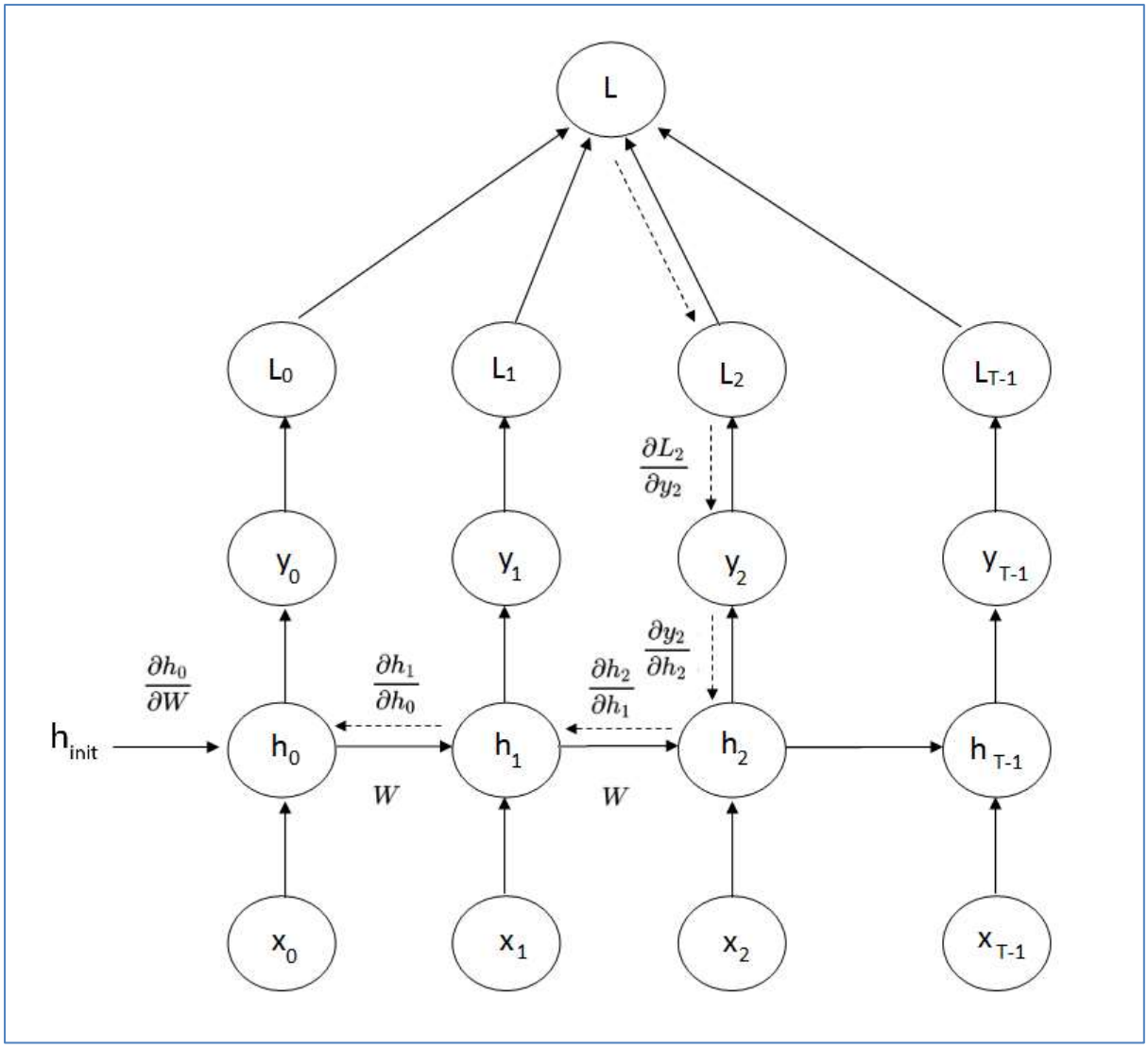
While computing the derivatives of loss with respect to W and U, we saw that we have to traverse all the way back to the first hidden state, as each hidden state at a time t is dependent on its previous hidden state at a time $t-1$.

For instance, the gradient of loss L_2 with respect to W is given as:

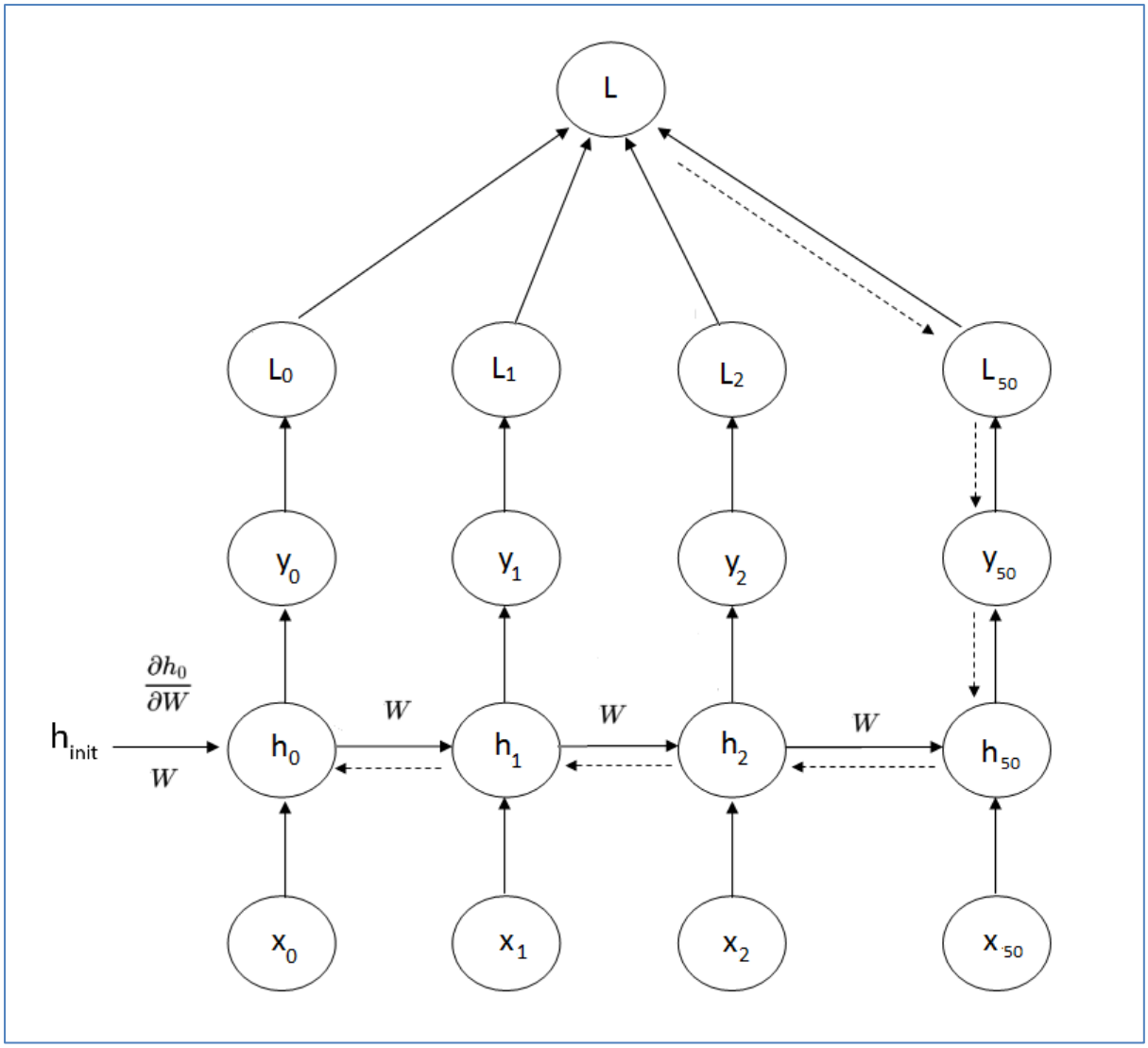
$$\frac{\partial L_2}{\partial W} = \frac{\partial L_2}{\partial y_2} \frac{\partial y_2}{\partial h_2} \frac{\partial h_2}{\partial W}$$

If you look at the term $\frac{\partial h_2}{\partial W}$ from the preceding equation, we can't calculate the derivative of h_2 with respect to W directly. As we know, $h_2 = tanh(Ux_2 + Wh_1)$ is a function that is dependent on h_1 and w. So, we need to calculate the derivative with respect to h_1, as well. Even $h_1 = tanh(Ux_2 + Wh_0)$ is a function that is dependent on h_0 and W. Thus, we need to calculate the derivative with respect to h_0, as well.

As shown in the following figure, to compute the derivative of L_2, we need to go all the way back to the initial hidden state h_0, as each hidden state is dependent on its previous hidden state:



So, to compute any loss L_j, we need to traverse all the way back to the initial hidden state h_0, as each hidden state is dependent on its previous hidden state. Suppose that we have a deep recurrent network with 50 layers. To compute the loss L_{50}, we need to traverse all the way back to h_0, as shown in the following figure:



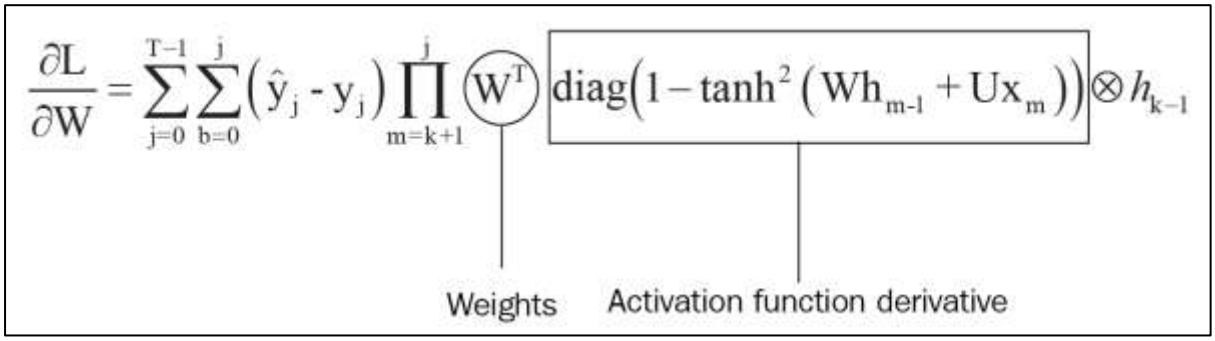
So, what is the problem here, exactly? While back propagating towards the initial hidden state, we lose information, and the RNN will not back propagate perfectly.

Remember $h_t = \tanh(Ux_t + Wh_{t-1})$? Every time we move backward, we compute the derivative of h_t. A derivative of tanh is bounded to 1. We know that any two values between 0 and 1, when multiplied with each other will give us a smaller number. We usually initialize the weights of the network to a small number. Thus, when we multiply the derivatives and weights while back propagating, we are essentially multiplying smaller numbers.

So, when we multiply smaller numbers at every step while moving backward, our gradient becomes infinitesimally small and leads to a

number that the computer can't handle; this is called the **vanishing gradient problem.**

Recall the equation of gradient of the loss with respect to the W that we saw in the *Gradients with respect to hidden to hidden layer weights, W* section:



As you can observe, we are multiplying the weights and derivative of the tanh function at every time step. Repeated multiplication of these two leads to a small number and causes the vanishing gradients problem.

The vanishing gradients problem occurs not only in RNN but also in other deep networks where we use sigmoid or tanh as the activation function. So, to overcome this, we can use **ReLU** as an activation function instead of tanh.

However, we have a variant of the RNN called the **long short-term memory** (**LSTM**) network, which can solve the vanishing gradient problem effectively.

Similarly, when we initialize the weights of the network to a very large number, the gradients will become very large at every step. While back propagating, we multiply a large number together at every time step, and it leads to infinity. This is called the **exploding gradient problem.**

Different types of RNN architectures:

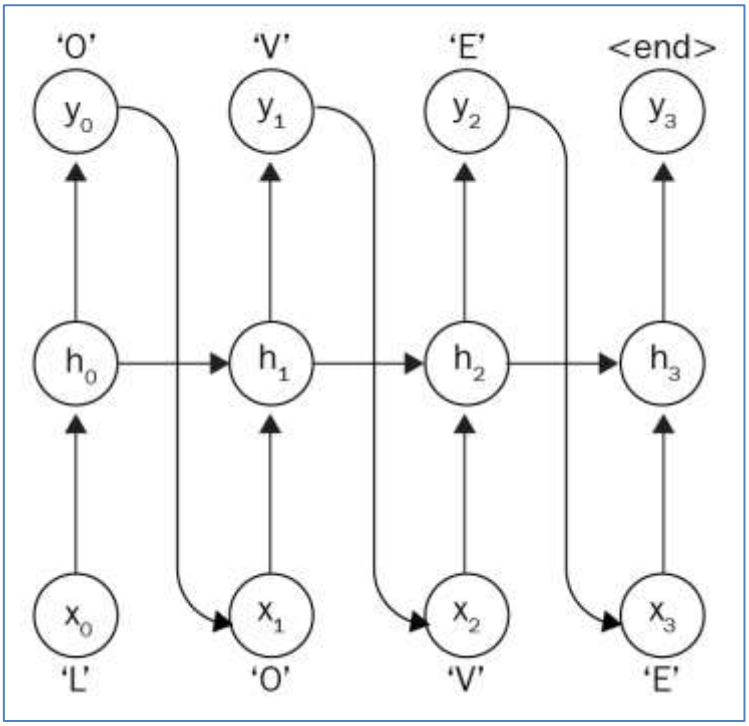
Different type of RNN architectures are based on numbers of input and output.

One-to-one architecture:

In a **one-to-one architecture**, a single input is mapped to a single output, and the output from the time step *t* is fed as an input to the next time step. We have already seen this architecture in the last section for generating songs using RNNs.

For instance, for a text generation task, we take the output generated from a current time step and feed it as the input to the next time step to generate the next word. This architecture is also widely used in **stock market predictions.**

The following figure shows the one-to-one RNN architecture. As you can see, output predicted at the time step *t* is sent as the input to the next time step:



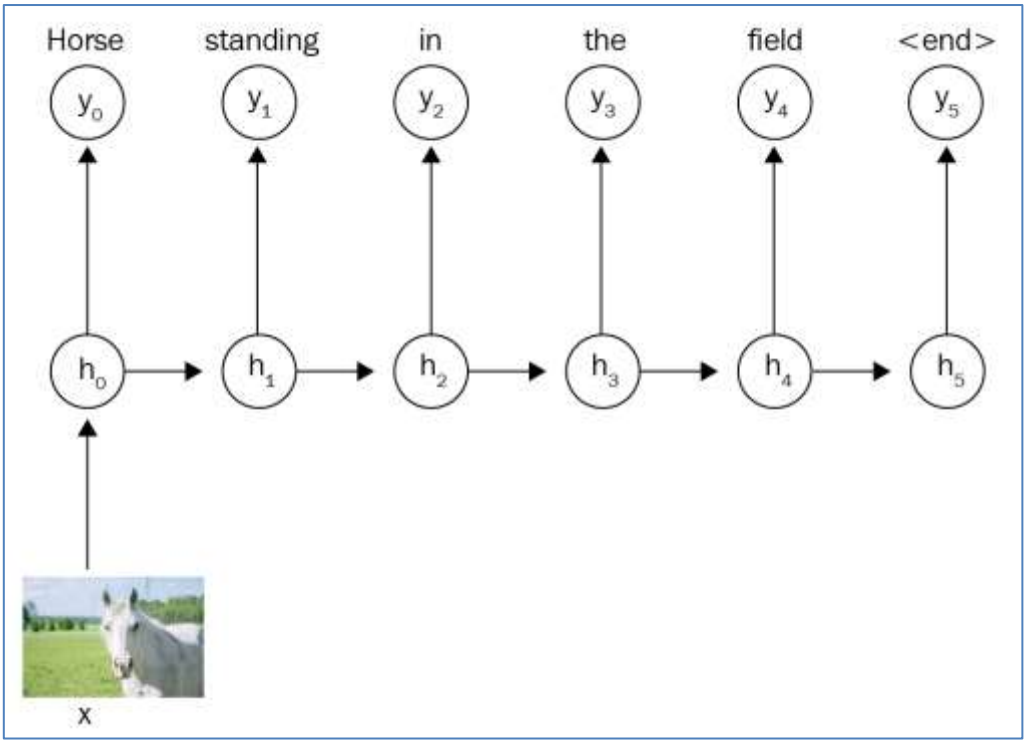
One-to-many architecture:

In a **one-to-many architecture**, a single input is mapped to multiple hidden states and multiple output values, which means RNN takes a single input and maps it to an output sequence. Although we have a single input value, we share the hidden states across time steps to predict the output. Unlike the previous

one-to-one architecture, here, we only share the previous hidden states across time steps and not the previous outputs.

One such application of this architecture is **image caption generation**. We pass a single image as an input, and the output is the sequence of words constituting a caption of the image.

As shown in the following figure, a single image is passed as an input to the RNN, and at the first time step, t_0, the word *Horse* is predicted; on the next time step, t_1, the previous hidden state h_0 is used to predict the next word which is *standing*. Similarly, it continues for a sequence of steps and predicts the next word until the caption is generated:

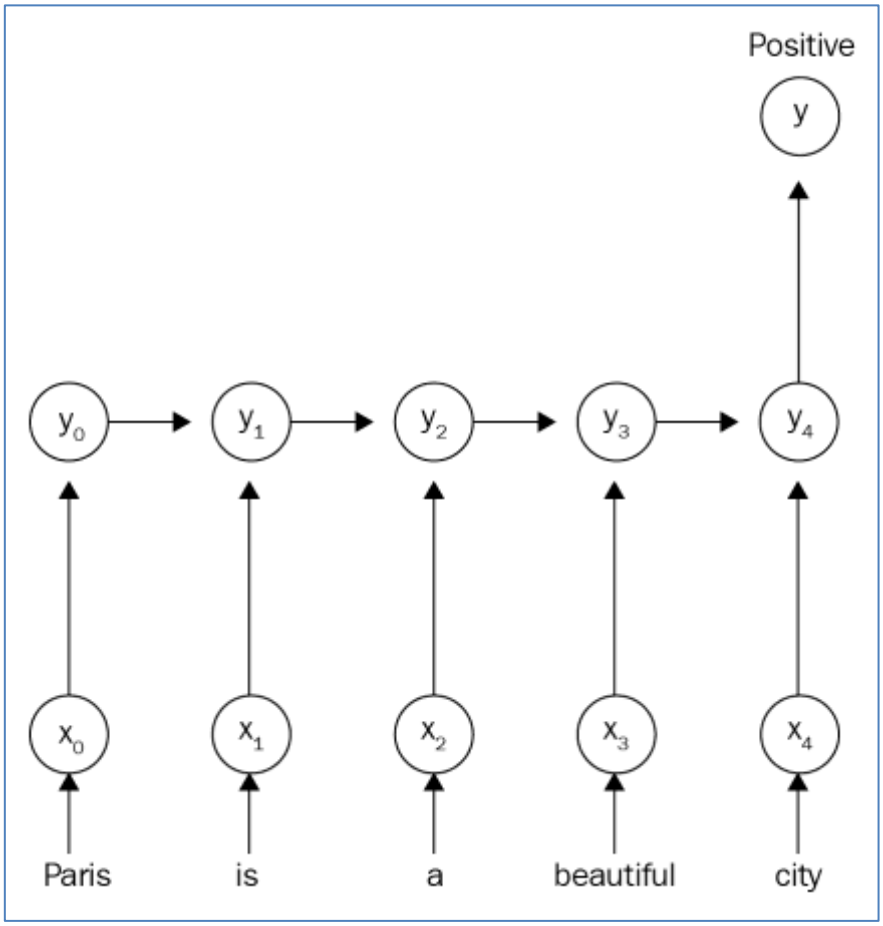


Many-to-one architecture:

A **many-to-one architecture,** as the name suggests, takes a sequence of input and maps it to a single output value. One such popular example of a many-to-one architecture is **sentiment classification**. A sentence is a sequence of words, so on each time step, we pass each word as input and predict the output at the final time step.

Let's suppose that we have a sentence: *Paris is a beautiful city*. As shown in the following figure, at each time step, a single word is passed as an

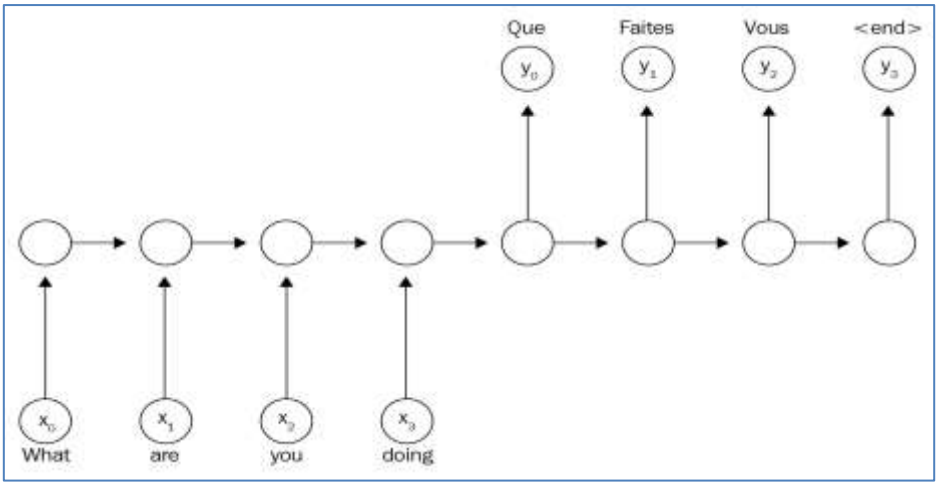
input, along with the previous hidden state; and, at the final time step, it predicts the sentiment of the sentence:



Many-to-many architecture:

In **many-to-many architectures**, we map a sequence of input of arbitrary length to a sequence of output of arbitrary length. This architecture has been used in various applications. Some of the popular applications of many-to-many architectures include language translation, conversational bots, and audio generation.

Let's suppose that we are converting a sentence from English to French. Consider our input sentence: *What are you doing?* It would be mapped to, *Que faites vous* as shown in the following figure:


Que
Faites
Vous
<end>
What
are
you
doing

Improvements to the RNN

- The drawback of a **recurrent neural network** (**RNN**) is that it will not retain information for a long time in memory. We know that an RNN stores sequences of information in its hidden state but when the input sequence is too long, it cannot retain all the information in its memory due to the vanishing gradient problem.
- To combat this, we introduce a variant of RNN called a **long short-term memory** (**LSTM**) cell, which resolves the vanishing gradient problem by using a special structure called a **gate**. Gates keep the information in memory as long as it is required. They learn what information to keep and what information to discard from the memory.

LSTM to the rescue:

- While back propagating an RNN, we discovered a problem called **vanishing gradients**. Due to the vanishing gradient problem, we cannot train the network properly, and this causes the RNN to not retain long sequences in the memory. To understand what we mean by this, let's consider a small sentence:

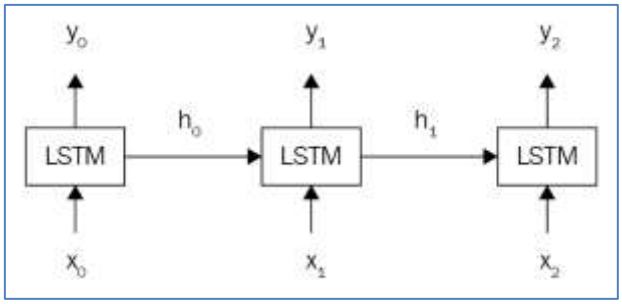
  The sky is __.

- An RNN can easily predict the blank as *blue* based on the information it has seen, but it cannot cover the long-term dependencies. What does that mean? Let's consider the following sentence to understand the problem better:

  Archie lived in China for 13 years. He loves listening to good music. He is a fan of comics. He is fluent in ____.

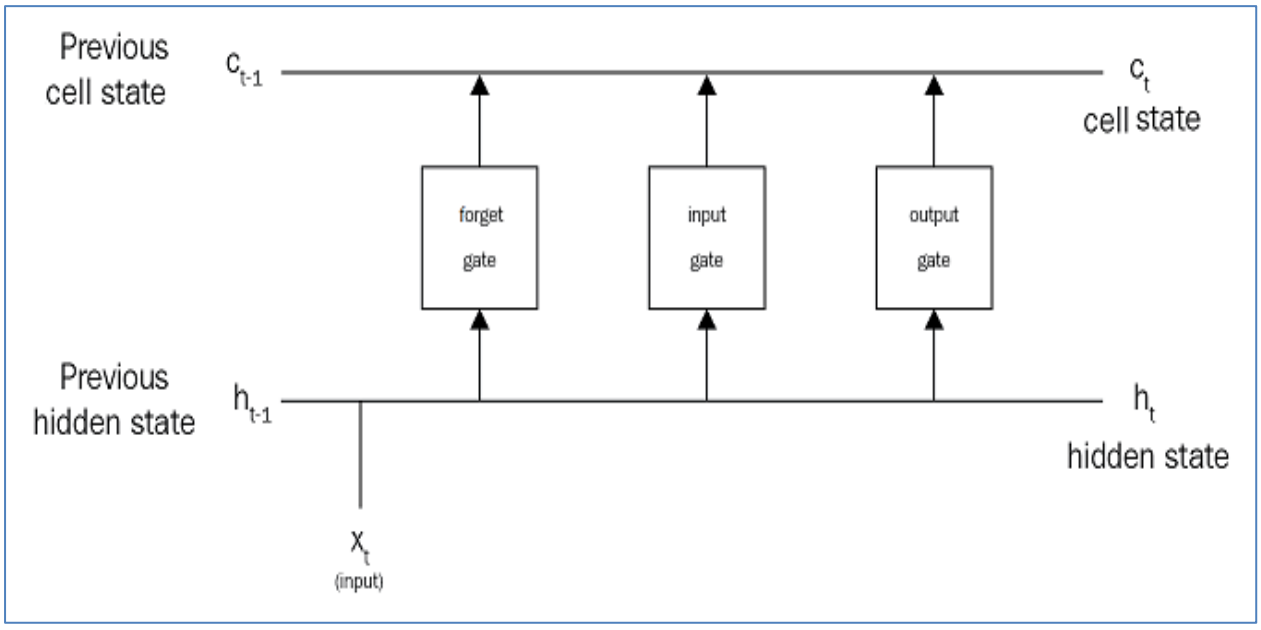
- Now, if we were asked to predict the missing word in the preceding sentence, we would predict it as *Chinese*, but how did we predict that? We simply remembered the previous sentences and understood that Archie lived for 13 years in China. This led us to the conclusion that Archie might be fluent in Chinese.
- An RNN, on the other hand, cannot retain all of this information in its memory to say that Archie is fluent in Chinese. Due to the vanishing gradient problem, it cannot recollect/remember information for a long time in its memory. That is, when the input sequence is long, the RNN memory (hidden state) cannot hold all the information. To alleviate this, we use an LSTM cell.

- LSTM is a variant of an RNN that resolves the vanishing gradient problem and retains information in the memory as long as it is required. Basically, RNN cells are replaced with LSTM cells in the hidden units, as shown in the following diagram:



Understanding the LSTM cell:

- What makes LSTM cells so special? How do LSTM cells achieve long-term dependency? How does it know what information to keep and what information to discard from the memory?
- This is all achieved by special structures called **gates**. As shown in the following diagram, a typical LSTM cell consists of three special gates called the input gate, output gate, and forget gate:



- These three gates are responsible for deciding what information to add, output, and forget from the memory. With these gates, an LSTM cell effectively keeps information in the memory only as long as required.

- In an RNN cell, we used the hidden state, h_t, for two purposes: one for storing the information and the other for making predictions. Unlike RNN, in the LSTM cell we break the hidden states into two states, called the **cell state** and the **hidden state**:

  - The cell state is also called internal memory and is where all the information will be stored
  - The hidden state is used for computing the output, that is, for making predictions

Both the cell state and hidden state are shared across every time step. Now, we will deep dive into the LSTM cell and see exactly how these gates are used and how the hidden state predicts the output.

Forget gate:

- The forget gate, f_t, is responsible for deciding what information should be removed from the cell state (memory). Consider the following sentence:

  Harry is a good singer. He lives in New York. Zayn is also a good singer.

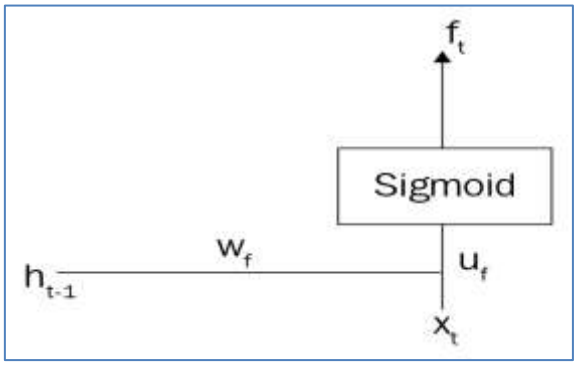
- As soon as we start talking about Zayn, the network will understand that the subject has been changed from Harry to Zayn, and the information about Harry is no longer required. Now, the forget gate will remove/forget information about Harry from the cell state.
- The forget gate is controlled by a sigmoid function. At time step t, we pass input x_t, and the previous hidden state, h_{t-1}, to the forget gate. It returns 0 if the particular information from the cell state should be removed and returns 1 if the information should not be removed. The forget gate, f, at a time step, t, is expressed as follows:

$$f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$$

  Here, the following applies:

  - U_f is the input-to-hidden layer weights of the forget gate
  - W_f is the hidden-to-hidden layer weights of the forget gate
  - b_f is the bias of the forget gate

The following diagram shows the forget gate. As you can see, input x_t is multiplied with U_f and the previous hidden state, h_{t-1}, is multiplied with W_f, then both will be added together and sent to the sigmoid function, which returns f_t, as follows:



Input gate:

- The input gate is responsible for deciding what information should be stored in the cell state. Let's consider the same example:

  Harry is a good singer. He lives in New York. Zayn is also a good singer.

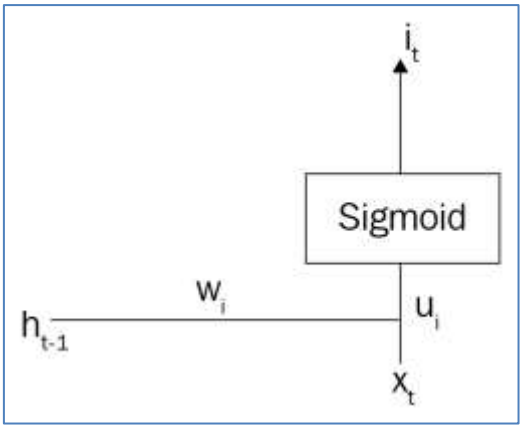
- After the forget gate removes information from the cell state, the input gate decides what information it has to keep in the memory. Here, since the information about Harry is removed from the cell state by the forget gate, the input gate decides to update the cell state with information about *Zayn*.
- Similar to the forget gate, the input gate is controlled by a sigmoid function that returns output in the range of 0 to 1. If it returns `1`, then the particular information will be stored/updated to the cell state, and if it returns `0`, we will not store the information to the cell state. The input gate i at time step t is expressed as follows:

$$i_t = \sigma(U_i x_t + W_i h_{t-1} + b_i)$$

  Here, the following applies:

  - U_i is the input-to-hidden weights of the input gate
  - W_i is the hidden-to-hidden weights of the input gate
  - b_i is the bias of the input gate

  The following diagram shows the input gate:



Output gate:

- We will have a lot of information in the cell state (memory). The output gate is responsible for deciding what information should be taken from the cell state to give as an output. Consider the following sentence:

  Zayn's debut album was a huge success. Congrats ____.

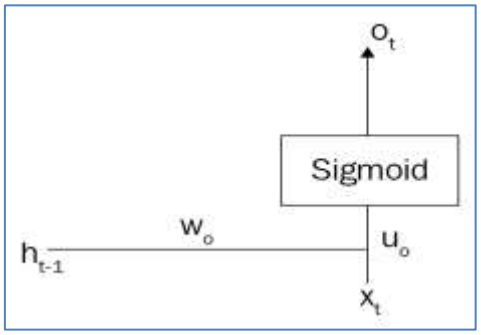
- The output gate will look up all the information in the cell state and select the correct information to fill the blank. Here, `congrats` is an adjective that is used to describe a noun. So, the output gate will predict *Zayn* (noun) to fill the blank. Similar to other gates, it is also controlled by a sigmoid function. The output gate o at time step t is expressed as follows:

$$o_t = \sigma(U_o x_t + W_o h_{t-1} + b_o)$$

Here, the following applies:

- U_o is the input-to-hidden weights of the output gate
- W_o is the hidden-to-hidden weights of the output gate
- b_o is the bias of the output gate

The output gate is shown in the following diagram:



Updating the cell state:

- We just learned how all three gates work in an LSTM network, but the question is, how can we actually update the cell state by adding relevant new information and deleting information that is not required from the cell state with the help of the gates?
- First, we will see how to add new relevant information to the cell state. To hold all the new information that can be added to the cell state (memory), we create a new vector called g_t. It is called a **candidate state** or **internal state vector**.
- Unlike gates that are regulated by the sigmoid function, the candidate state is regulated by the tanh function, but why? The sigmoid function returns values in the range of `0` to `1`, that is, it is always positive. We need to allow the values of g_t to be either positive or negative. So, we use the tanh function, which returns values in the range of `-1` to `+1`.

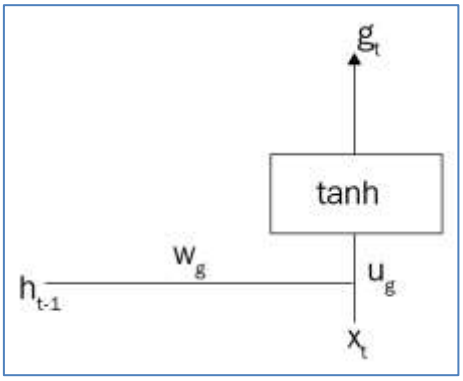
  The candidate state, g, at time t is expressed as follows:

$$g_t = tanh(U_g x_t + W_g h_{t-1} + b_g)$$

  Here, the following applies:

  - U_g is the input-to-hidden weights of the candidate state
  - W_g is the hidden-to-hidden weights of the candidate state
  - $b_g$is the bias of the candidate state

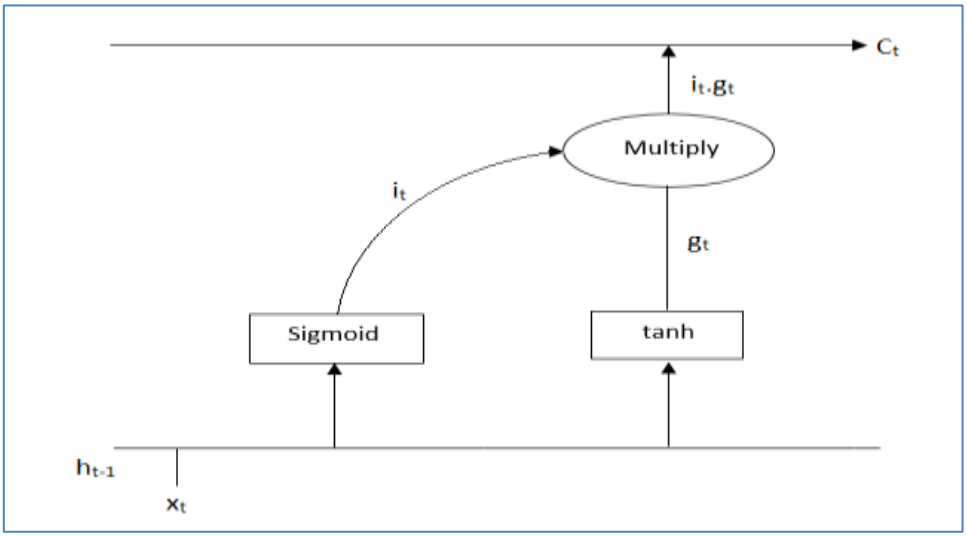
Thus, the candidate state holds all the new information that can be added to the cell state (memory). The following diagram shows the candidate state:



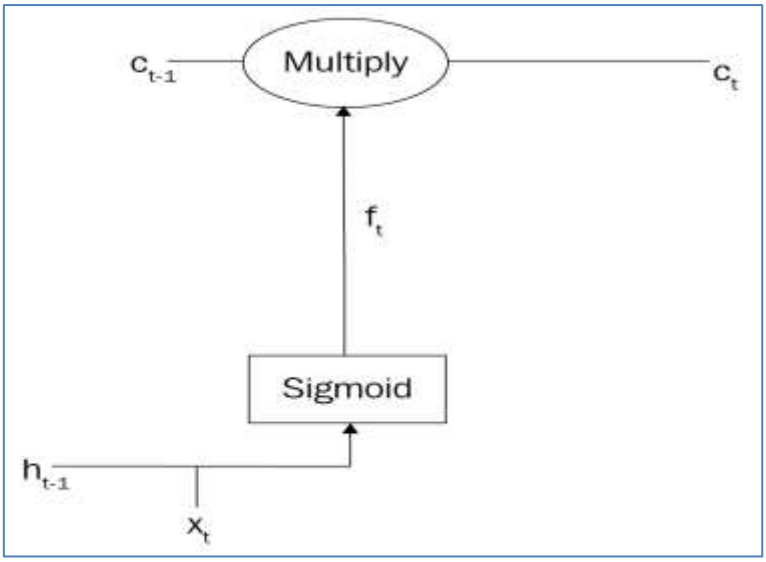
- How do we decide whether the information in the candidate state is relevant? How do we decide whether to add new information or not from the candidate state to the cell state? We learned that the input

gate is responsible for deciding whether to add new information or not, so if we multiply g_t and i_t, we only get relevant information that should be added to the memory.

- That is, we know the input gate returns 0 if the information is not required and 1 if the information is required. Say $i_t = 0$, then multiplying g_t and i_t gives us 0, which means the information in g_t is not required and we don't want to update the cell state with g_t. When $i_t = 1$, then multiplying g_t and i_t gives us g_t, which implies we can update the information in g_t to the cell state.
- Adding the new information to the cell state with input gate i_t and candidate state g_t is shown in the following diagram:



- Now, we will see how to remove information from the previous cell state that is no longer required.
- We learned that the forget gate is used for removing information that is not required in the cell state. So, if we multiply the previous cell state, c_{t-1}, and forget gate, f_t, then we retain only relevant information in the cell state.
- Say $f_t = 0$, then multiplying c_{t-1} and f_t gives us 0, which means the information in cell state, c_{t-1}, is not required and should be removed (forgotten). When $f_t = 1$, then multiplying c_{t-1} and f_t gives c_{t-1}, which implies that information in the previous cell state is required and should not be removed.
- Removing information from the previous cell state, c_{t-1}, with the forget gate, f_t, is shown in the following diagram:



- Thus, in a nutshell, we update our cell state by multiplying g_t and i_t to add new information, and multiplying c_{t-1} ,and f_t to remove information. We can express the cell state equation as follows:

$$\boxed{c_t = f_t c_{t-1} + i_t g_t}$$

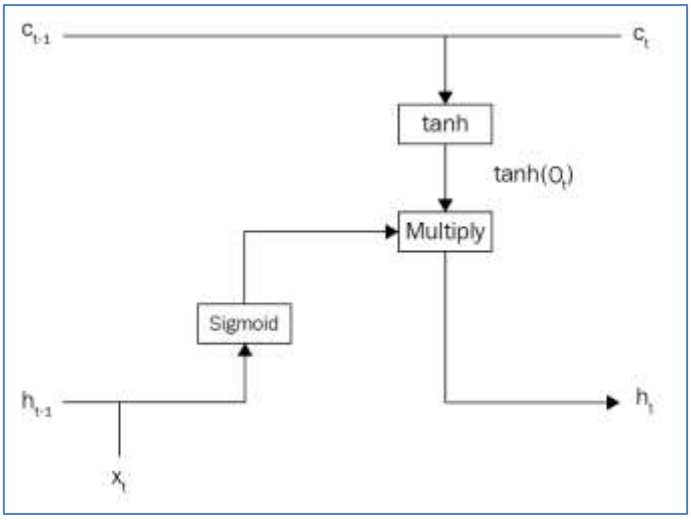
Updating hidden state:

- We just learned how the information in the cell state will be updated. Now, we will see how the information in the hidden state will be updated. We learned that the hidden state, h_t, is used for computing the output, but how can we compute the output?
- We know that the output gate is responsible for deciding what information should be taken from the cell state to give as output. Thus, multiplying o_t and tanh (to squash between -1 and +1) of cell state, $\tanh(c_t)$, gives us the output.

  Thus, the hidden state, h_t, is expressed as follows:

$$\boxed{h_t = o_t tanh(c_t)}$$

- The following diagram shows how the hidden state, h_t, is computed by multiplying o_t and $\tanh(c_t)$:



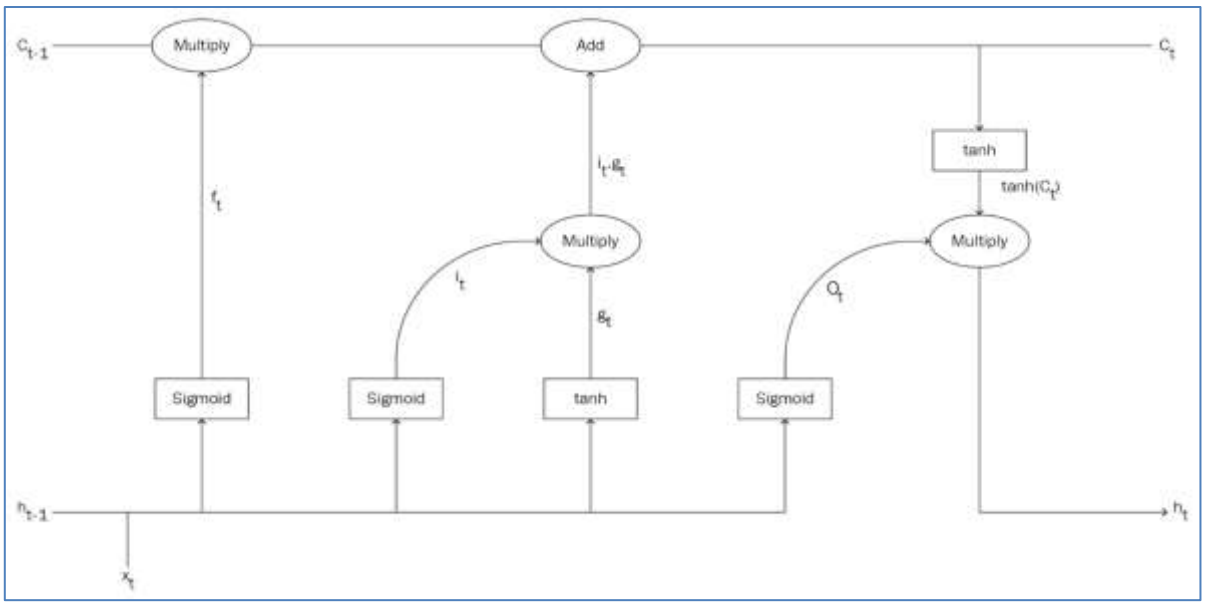
Finally, once we have the hidden state value, we can apply the softmax function and compute $\hat{y}_t$ as follows:

$$\hat{y}_t = softmax(Vh_t)$$

Here, V is the hidden-to-output layer weights.

Forward propagation in LSTM:

- Putting it all together, the final LSTM cell with all the operations is shown in the following diagram. Cell state and hidden states are shared across time steps, meaning that the LSTM computes the cell state, c_t, and hidden state, h_t, at time step t, and sends it to the next time step:



The complete forward propagation steps in the LSTM cell can be given as follows:

1. **Input gate**: $i_t = \sigma(U_i x_t + W_f h_{t-1} + b_i)$
2. **Forget gate**: $f_t = \sigma(U_f x_t + W_f h_{t-1} + b_f)$
3. **Output gate**: $o_t = \sigma(U_o x_t + W_o h_{t-1} + b_o)$
4. **Candidate state**: $g_t = tanh(U_g x_t + W_g h_{t-1} + b_g)$
5. **Cell state**: $c_t = f_t c_{t-1} + i_t g_t$
6. **Hidden state**: $h_t = o_t tanh(c_t)$
7. **Output**: $\hat{y}_t = softmax(V h_t)$

Backpropagation in LSTM

- We compute the loss at each time step to determine how well our LSTM model is predicting the output. Say we use cross-entropy as a loss function, then the loss, L, at time step t is given by the following equation:

$$\boxed{L_t = -y_t \log(\hat{y}_t)}$$

- Here, y_t is the actual output and $\hat{y}_t$ is the predicted output at time step t. Our final loss is the sum of loss at all time steps, and can be given as follows:

$$\boxed{L = \sum_{j=0}^{T} L_j}$$

- We minimize the loss using gradient descent. We find the derivative of loss with respect to all of the weights used in the network and find the optimal weights to minimize the loss:
  - We have four inputs-to-hidden layer weights, U_i, U_f, U_o, U_g, which are the input-to-hidden layer weights of the input gate, forget gate, output gate, and candidate state, respectively
  - We have four hidden-to-hidden layer weights, W_i, W_f, W_o, W_g, which implies hidden-to-hidden layer weights of input gate, forget gate, output gate, and candidate state, respectively
  - We have one hidden-to-output layer weight, V

We find the optimal values for all these weights through gradient descent and update the weights according to the weight update rule. The weight update rule is given by the following equation:

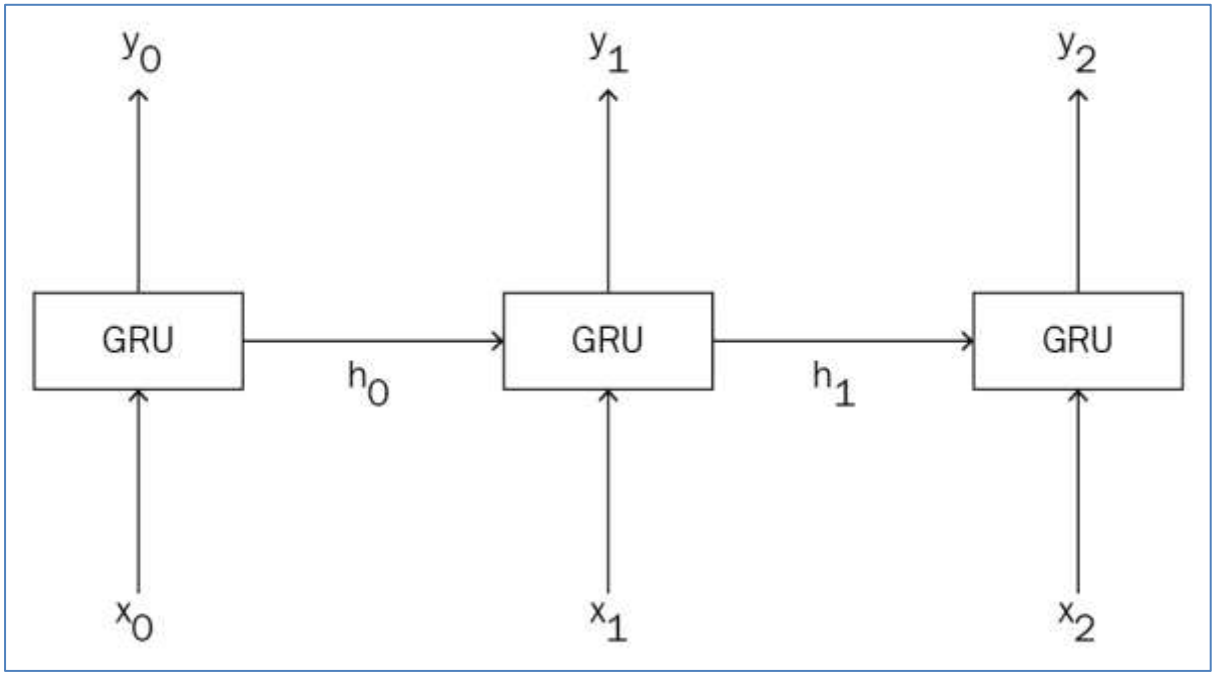
$$\text{weight} = \text{weight} - \alpha \frac{\partial Loss}{\text{weight}}$$

Gated recurrent units

So far, we have learned about how the LSTM cell uses different gates and how it solves the vanishing gradient problem of the RNN. But, as you may have noticed, the LSTM cell has too many parameters due to the presence of many gates and states.

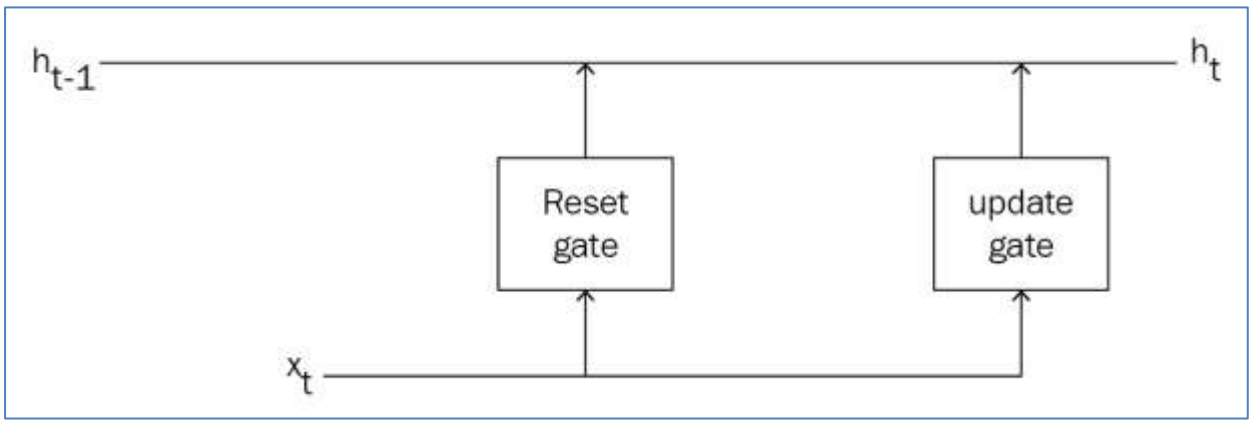
Thus, while back propagating the LSTM network, we need to update a lot of parameters in every iteration. This increases our training time. So, we introduce the **Gated Recurrent Units** (**GRU**) cell, which acts as a simplified version of the LSTM cell. Unlike the LSTM cell, the GRU cell has only two gates and one hidden state.

An RNN with a GRU cell is shown in the following diagram:



Understanding the GRU cell

As shown in the following diagram, a GRU cell has only two gates, called the update gate and the reset gate, and one hidden state:



Let's delve deeper and see how these gates are used and how the hidden state is computed.

Update gate

The update gate helps to decide what information from the previous time step, h_{t-1}, can be taken forward to the next time step, h_t. It is basically a combination of an input gate and a forget gate, which we learned about in LSTM cells. Similar to the gates about the LSTM cell, the update gate is also regulated by the sigmoid function.

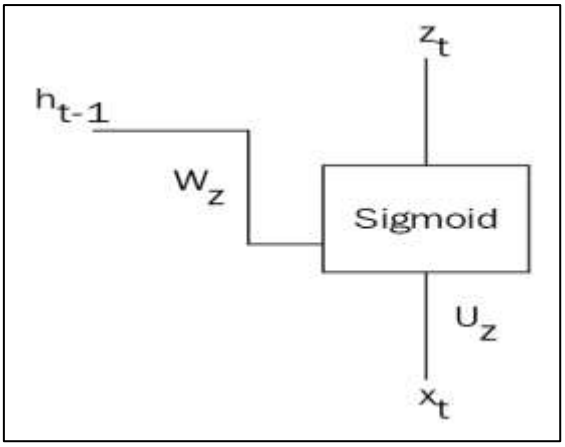
The update gate, z, at time step t is expressed as follows:

$$z_t = \sigma(U_z x_t + W_z h_{t-1} + b_z)$$

Here, the following applies:

- U_z is the input-to-hidden weights of the update gate
- W_z is the hidden-to-hidden weights of the update gate
- b_z is the bias of the update gate

The following diagram shows the update gate. As you can see, input x_t is multiplied with U_z, and the previous hidden state, h_{t-1}, o and 1:



Reset gate

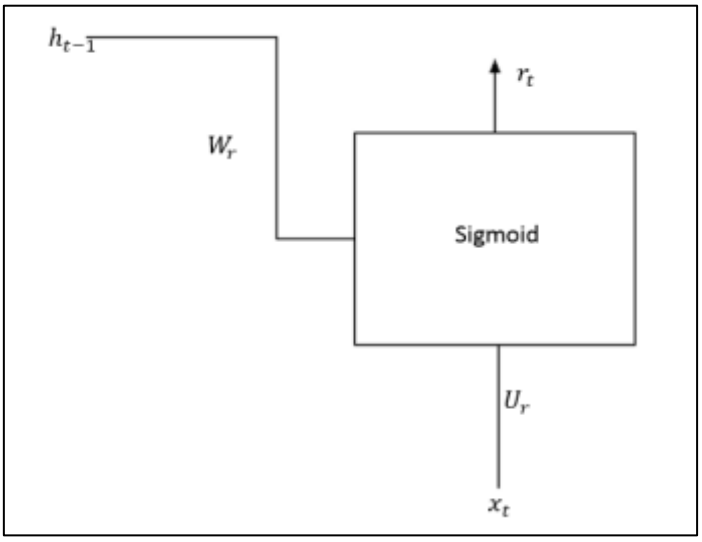
The reset gate helps to decide how to add the new information to the memory, that is, how much of the past information it can forget. The reset gate, r ,at time step t is expressed as follows:

$$r_t = \sigma(U_r x_t + W_r h_{t-1} + b_r)$$

Here, the following applies:

- U_r is the input-to-hidden weights of the reset gate
- W_r is the hidden-to-hidden weights of the reset gate
- b_r is the bias of the reset gate

The reset gate is shown in the following diagram:



Updating hidden state

We just learned how the update and reset gates work, but how do these gates help in updating the hidden state? That is, how do you add new information to the hidden state and how do you remove unwanted information from the hidden state with the help of the reset and update gates?

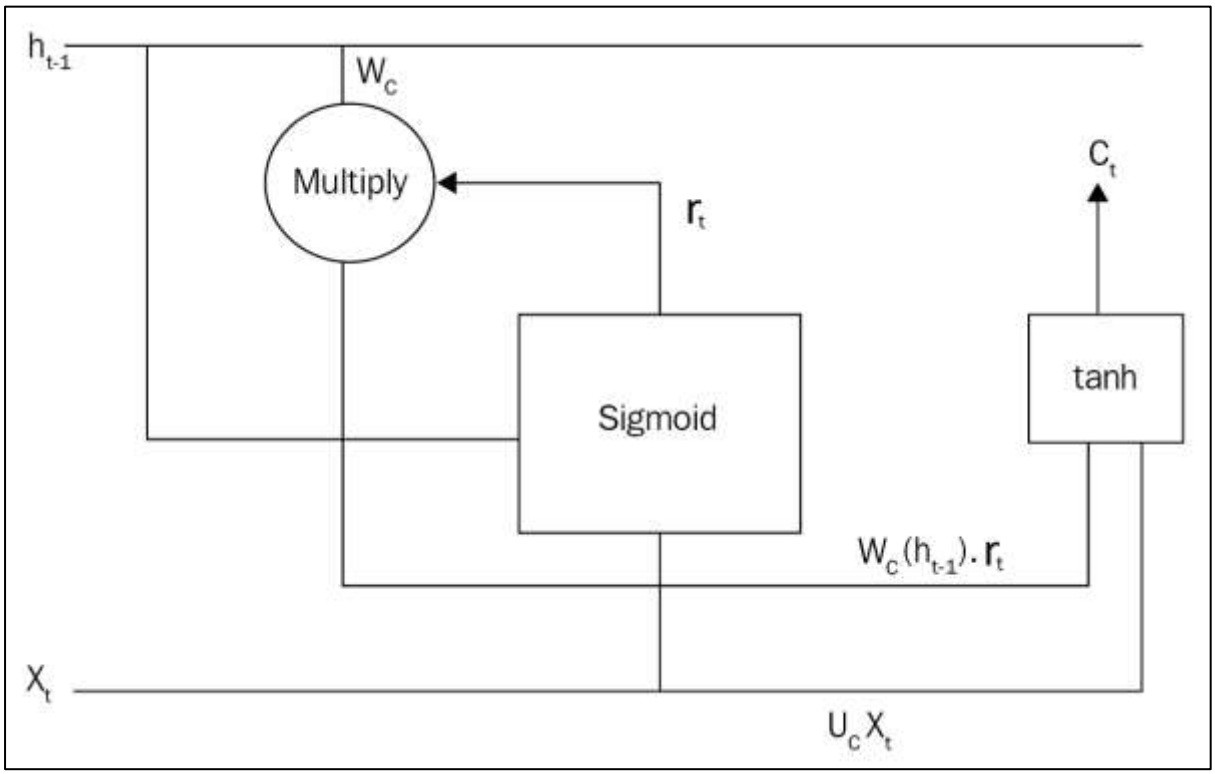
First, we will see how to add new information to the hidden state.

We create a new state called the **content state**, c_t, for holding the information. We know that the reset gate is used to remove information that is not required. So, using the reset gate, we create a content state, c_t, that holds only the required information.

The content state, c, at time step t is expressed as follows:

$$c_t = tanh(U_c x_t + W_c(h_{t-1}) \cdot r_t)$$

The following diagram shows how the content state is created with the reset gate:



Now we will see how to remove information from the hidden state.

We learned that the update gate, z_t, helps to decide what information from the previous time step, h_{t-1}, can be taken forward to the next time step, h_t. Multiplying z_t and h_{t-1} gives us only the relevant information from the previous step. Instead of having a new gate, we just a take a complement of z_t, that is $(1 - z_t)$, and multiply them with c_t.

The hidden state is then updated as follows:

$$h_t = (1 - z_t) \cdot c_t + z_t \cdot h_{t-1}$$

Once the hidden state is computed, we can apply the softmax function and compute the output as follows:

$$\hat{y}_t = softmax(V h_t)$$