UNIT - IV Recurrent and Recursive Nets

INTRODUCTION :

- Recurrent neural networks or RNNs (Rumelhart et al. 1986a ,) are a family of neural networks for processing sequential data. Much as a convolutional network is a neural network that is specialized for processing a grid of values X such as an image, a recurrent neural network is a neural network that is specialized for processing a sequence of values x(1),...,x(t).
- Most recurrent networks can also process sequences of variable length.
- To go from multi-layer networks to recurrent networks, we need to take advantage of one of the early ideas found in machine learning and statistical models of the 1980s: sharing parameters across different parts of a model.
- Without shared parameters for each time step, the model would struggle to generalize to sequence lengths it hasn't encountered during training or to utilize statistical patterns across different sequence lengths and time positions.
- Parameter sharing is crucial, especially when specific information can appear at various points within a sequence.
- For instance, consider the sentences, "I went to Nepal in 2009" and "In 2009, I went to Nepal." If a machine learning model is tasked with identifying the year the narrator visited Nepal, it should recognize "2009" as the relevant information, regardless of whether it appears as the sixth word or the second word.
- If we used a feedforward network designed for fixed-length sentences, it would require separate parameters for each input feature, forcing it to learn the rules of the language independently for every position in the sentence.
- In contrast, a recurrent neural network (RNN) shares the same weights across multiple time steps, enabling it to generalize patterns regardless of their position within the sequence.
- A similar concept is the use of convolution over a one-dimensional temporal sequence, which forms the foundation of time-delay neural networks (Lang and Hinton, 1988; Waibel et al., 1989; Lang et al., 1990).
- Convolution enables a network to share parameters across time but remains relatively shallow. Each output in a convolutional network is determined by a small neighbourhood of inputs, with parameter sharing achieved through the repeated application of the same convolution kernel at every time step.

- Recurrent networks, on the other hand, share parameters differently. In these networks, each output depends on the preceding outputs, with the same update rule applied consistently across time steps.
- This recurrent mechanism results in parameter sharing across a much deeper computational graph, enabling the network to capture long-term dependencies in the data.
- **Why Recurrent Neural Networks?**
  RNN were created because there were a few issues in the feed-forward neural network:
  - Cannot handle sequential data
  - Considers only the current input
  - Cannot memorize previous inputs
- **Applications of Recurrent Neural Networks**
  - Image Captioning: RNNs are used to caption an image by analysing the activities present.
  - Time Series Prediction: Any time series problem, like predicting the prices of stocks in a particular month, can be solved using an RNN.
  - Natural Language Processing: Text mining and Sentiment analysis can be carried out using an RNN for Natural Language Processing (NLP).
  - Machine Translation: Given an input in one language, RNNs can be used to translate the input into different languages as output.
- **Advantages of Recurrent Neural Network**
  - Ability to Handle Variable-Length Sequences: RNNs are designed to handle input sequences of variable length, which makes them well-suited for tasks such as speech recognition.
  - Memory of Past Inputs :RNNs have a memory of past inputs, which allows them to capture information about the context of the input sequence.
  - Parameter Sharing: RNNs share the same set of parameters across all time steps, which reduce the number of parameters that need to be learned and can lead to better generalization.
  - Flexibility: RNNs can be adapted to a wide range of tasks and input types, including text, speech, and image sequences.
- **Disadvantages of Recurrent Neural Network**
  - Vanishing and Exploding Gradients: RNNs can suffer from the problem of vanishing or exploding gradients, which can make it difficult to train the network effectively.
  - Computational Complexity: RNNs can be computationally expensive to train, especially when dealing with long sequences.

- Lack of Parallelism: RNNs are inherently sequential, which makes it difficult to parallelize the computation. This can limit the speed and scalability of the network.
- Difficulty in Interpreting the Output: The output of an RNN can be difficult to interpret, especially when dealing with complex inputs such as natural language or audio.

Unfolding Computational Graphs:

- A computational graph provides a structured representation of computations, capturing the relationships between inputs, parameters, outputs, and loss. Here, we focus on the concept of unfolding recursive or recurrent computations into a computational graph with a repetitive structure, often representing a sequence of events.
- Unfolding this graph creates a chain-like structure, facilitating parameter sharing across the depth of the network.
- A Computational Graph is a way to formalize the structure of a set of computations such as mapping inputs and parameters to outputs and loss.
- We can unfold a recursive or recurrent computation into a computational graph that has a repetitive structure, corresponding to a chain of events. Unfolding this graph results in sharing of parameters across a deep network structure.
- Example of unfolding a recurrent equation:
  - ➔ consider the classical form of a dynamical system:
  - ➔ s (t) = f(s (t-1) ; 0), s (t) is called the state of the system.
- Equation is recurrent because the definition of s at time t refers back to the same definition at time t-1.
- For a finite no. of time steps t, the graph can be unfolded by applying the definition t-1 times.
- E.g. for t= 3 time steps we get s(3)= f (s(2); 0)= f (f (s(1); 0); 0)
- Unfolding equation by repeatedly applying the definition in this way has yielded expression without recurrence.
- s(1) is ground state and s(2) computed by applying f.
- Such an expression can be represented by a traditional acyclic computational graph.
- Unfolded dynamical system:
- The classical dynamical system described by s(t) =f (s(t-1); 0) and s(3)=f (f (s(1); 0); 0) is illustrated (fig.1) as an unfolded computational graph.

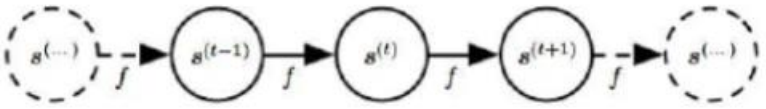


Fig 1. Illustration of an unfolded computational graph.

- Each node represents state at some time t. Function f maps state at time t to the state at t+1. The same parameters (the same value of 0 used to parameterize f) are used for all time steps.
- As another example, let us consider a dynamical system driven by an external signal x(t),

$$s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta),$$

- where we see that the state now contains information about the whole past sequence.
- Recurrent neural networks can be built in many different ways. Much as almost any function can be considered a feedforward neural network, essentially any function involving recurrence can be considered a recurrent neural network.
- To indicate that the state is the hidden units of the network, now let us rewrite the above equation using the variable h to represent the state:

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta),$$

- A recurrent network with no outputs is illustrated in figure2. This recurrent network just processes information from the input x by incorporating it into the state h that is passed forward through time. (Left)Circuit diagram.
- The black square indicates a delay of a single time step. (Right)The same network seen as an unfolded computational graph, where each node is now associated with one particular time instance.

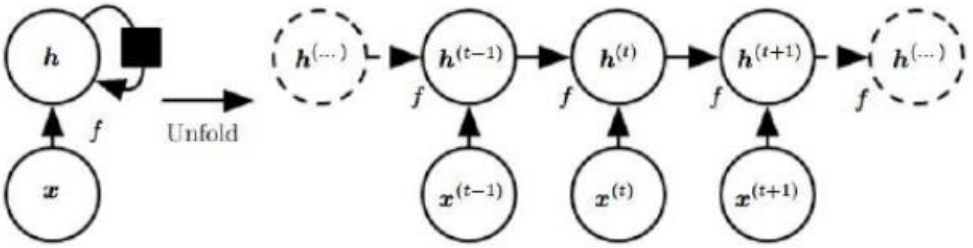


Fig.2 Hidden state h representation.

- As illustrated in figure 2, typical RNNs will add extra architectural features such as output layers that read information out of the state h to make predictions.
- When a recurrent network is trained for tasks that involve predicting the future from the past,typically learns to use h(t) as a condensed summary of the task-relevant aspects of the input sequence up to time t. This summary is inherently lossy because it compresses a sequence of arbitrary length sequence(x(t), x(t-1), x(t-2), . .. ,x(2) ,x(1)), into a fixed-length vector, h(t).The degree of detail retained in this summary depends on the training objective; some parts of the sequence may be preserved with more precision than others.
- For instance, in statistical language modeling, where the goal is to predict the next word based on previous words, the RNN does not need to store all information from the input sequence. Instead,retains just enough information to predict the continuation of the sentence effectively.
- When RNN is required to perform a task of predicting the future from the past, network typically learns to use h(t) as a lossy summary of the task-relevant aspects of the past sequence of inputs upto t.
- The summary is in general lossy since it maps a sequence of arbitrary length (x(t), x(t-1), .. ,X(2),x(1)) to a fixed length vector h(t)
- Information Contained in Summary: Depending on criterion, summary keeps some aspects of past sequence more precisely than other aspects.
- Examples:
- RNN used in statistical language modeling, typically to predict next word from past words. It may notbe necessary to store all information upto time t but only enough information to predict rest of sentence.
- Most demanding situation: we ask h(t) to be rich enough to allow one to approximately recover the input sequence as in autoencoders.
- Unfolding: from circuit diagram to computational graph:
- Equation h(t) =f (h(t-1), x(t); 0) can be written in two different ways: circuit diagram or an unfolded computational graph. Unfolding is the operation that maps a circuit to a computational graph with repeated pieces. The unfolded graph has a size dependent on the sequence length.

We can represent the unfolded recurrence after t steps with a function $g^{(t)}$: $h^{(t)} = g^{(t)}(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(2)}, x^{(1)}) = f(h^{(t-1)}, x^{(t)}; \theta)$ The function $g^{(t)}$ takes the whole past sequence $(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(2)}, x^{(1)})$ as input and produces the current state, but the unfolded recurrent structure allows us to factorize $g^{(t)}$ into repeated application of a function f.

- The unfolding process thus introduces two major advantages:
    i. Regardless of the sequence length, the learned model always has the same input size, because it is specified in terms of transition from one state to another state, rather than specified in terms of a variable-length history of states.
    ii. It is possible to use the same transition function f with the same parameters at every time step.
- These two factors enable the learning of a single model f that works across all time steps and sequence lengths, rather than requiring separate models g(t) for each time step. By sharing parameters, the model can generalize to sequence lengths not seen during training and can be trained with significantly fewer examples compared to models without parameter sharing.
- Both the recurrent graph and the unrolled graph have their uses. The recurrent graph is succinct. The unfolded graph provides an explicit description of which computations to perform. The unfolded graph also helps to illustrate the idea of information flow forward in time (computing outputs and losses) and backward in time (computing gradients) by explicitly showing the path along which this information flows.

Recurrent Neural Networks:

- Recurrent neural networks (RNNs) are different from other neural networks with their unique capabilities:
    - Internal Memory: This is the key feature of RNNs. It allows them to remember past inputs and use that context when processing new information.
    - Sequential Data Processing: Because of their memory, RNNs are exceptional at handling sequential data where the order of elements matters. This makes them ideal for speech recognition, machine translation, natural language processing (NLP) and text generation.
    - Contextual Understanding: RNNs can analyze the current input in relation to what they've "seen" before. This contextual understanding is crucial for tasks where meaning depends on prior information.
    - Dynamic Processing: RNNs can continuously update their internal memory as they process new data, allowing them to adapt to changing patterns within a sequence.
- Some examples of important design patterns for recurrent neural networks include the following:

- Recurrent networks that produce an output at each time step and have recurrent connections between hidden units, illustrated in figure 10.3 .
- Recurrent networks that produce an output at each time step and have recurrent connections only from the output at one time step to the hidden units at the next time step, illustrated in figure 10.4
- Recurrent networks with recurrent connections between hidden units, that 10.5 . read an entire sequence and then produce a single output,

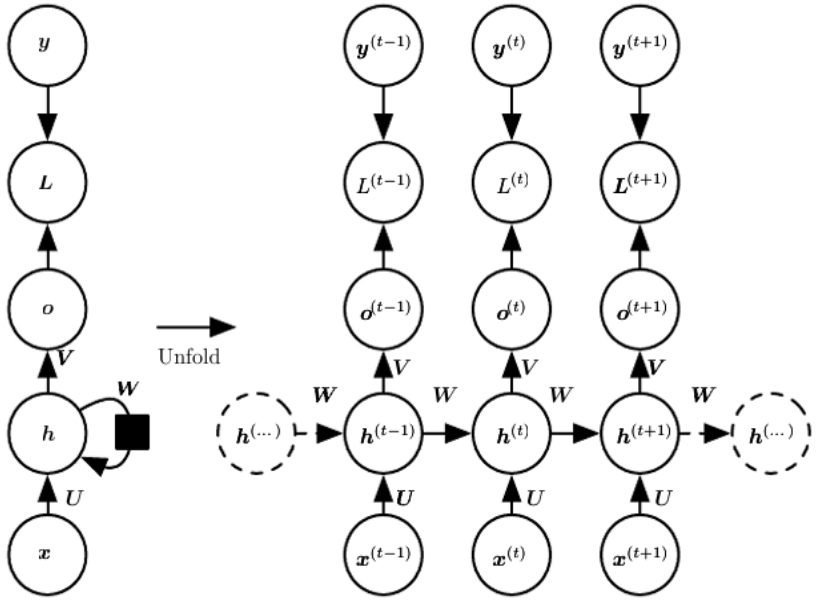


Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of $\boldsymbol{x}$ values to a corresponding sequence of output $\boldsymbol{o}$ values. A loss L measures how far each $\boldsymbol{o}$ is from the corresponding training target $\boldsymbol{y}$. When using softmax outputs, we assume $\boldsymbol{o}$ is the unnormalized log probabilities. The loss L internally computes $\hat{\boldsymbol{y}} = \text{softmax}(\boldsymbol{o})$ and compares this to the target $\boldsymbol{y}$. The RNN has input to hidden connections parametrized by a weight matrix $\boldsymbol{U}$, hidden-to-hidden recurrent connections parametrized by a weight matrix $\boldsymbol{W}$, and hidden-to-output connections parametrized by a weight matrix $\boldsymbol{V}$. Equation 10.8 defines forward propagation in this model. *(Left)*The RNN and its loss drawn with recurrent connections. *(Right)*The same seen as an time-unfolded computational graph, where each node is now associated with one particular time instance.

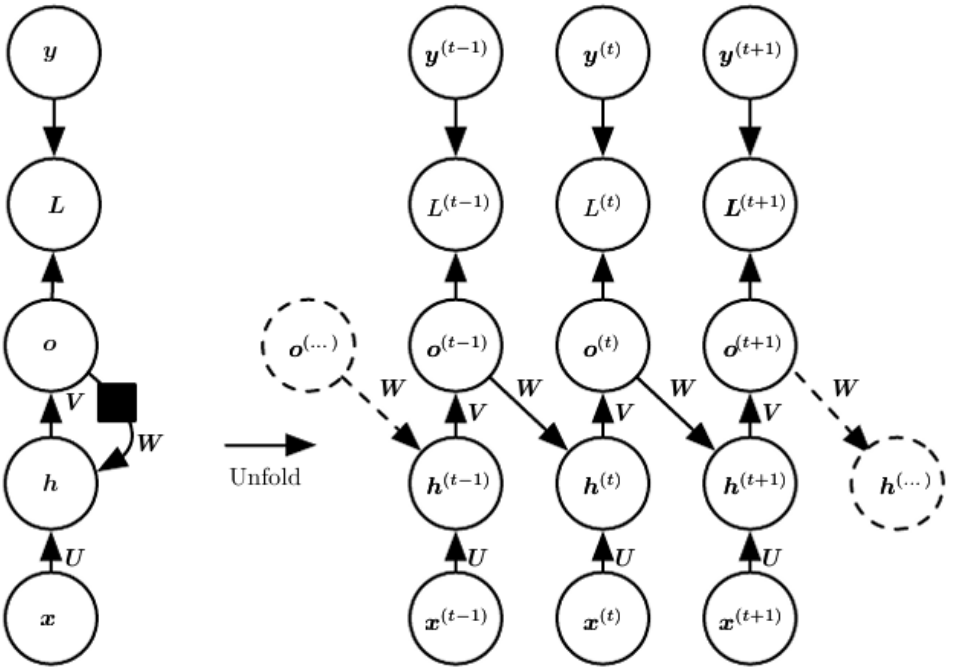


Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer.

- The forward propagation equations for the RNN are as follows:
- Forward propagation begins with a specification of the initial state. Then, for each time step from t=1 to t= t, we apply the following update equations:

$$\begin{aligned} \boldsymbol{a}^{(t)} &= \boldsymbol{b} + \boldsymbol{W}\boldsymbol{h}^{(t-1)} + \boldsymbol{U}\boldsymbol{x}^{(t)} \\ \boldsymbol{h}^{(t)} &= \tanh(\boldsymbol{a}^{(t)}) \\ \boldsymbol{o}^{(t)} &= \boldsymbol{c} + \boldsymbol{V}\boldsymbol{h}^{(t)} \\ \hat{\boldsymbol{y}}^{(t)} &= \mathrm{softmax}(\boldsymbol{o}^{(t)}) \end{aligned}$$

  Where the parameters are the bias vectors b and c along with the weight matrices U, V and W, respectively for input-to-hidden, hidden-to-output and hidden-to hidden connections.

- **Teacher Forcing and Networks with Output Recurrence:**
  - The advantage of eliminating hidden-to-hidden recurrence is that, for any loss function based on comparing the prediction at time t to the training target at time t, all the time steps are decoupled.
  - There is no need to compute the output for the previous time step first, because the training set provides the ideal value of that output.

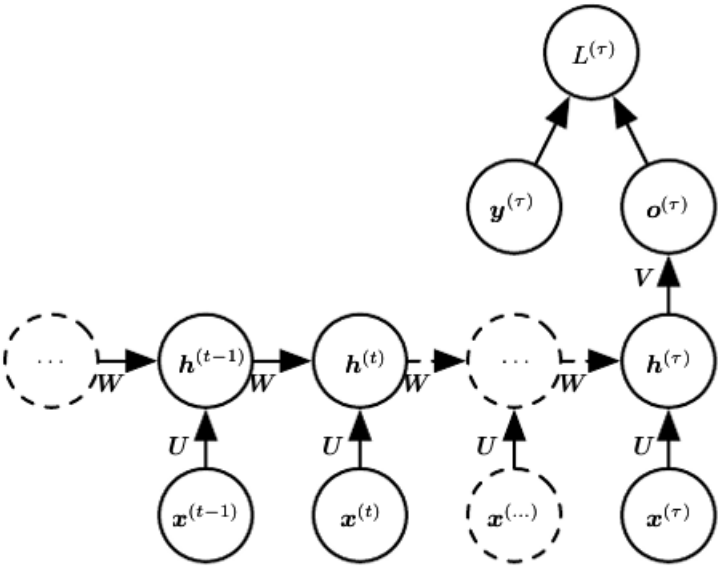


Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence.

- Models that have recurrent connections from their outputs leading back into the model may be trained with teacher forcing.
- The conditional maximum likelihood criterion is :

$$\log p\left(\boldsymbol{y}^{(1)}, \boldsymbol{y}^{(2)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right)$$
$$= \log p\left(\boldsymbol{y}^{(2)} \mid \boldsymbol{y}^{(1)}, \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right) + \log p\left(\boldsymbol{y}^{(1)} \mid \boldsymbol{x}^{(1)}, \boldsymbol{x}^{(2)}\right)$$

In this example, we see that at time t = 2, the model is trained to maximize the conditional probability of y(2) given both the x sequence so far and the previous y value from the training set.

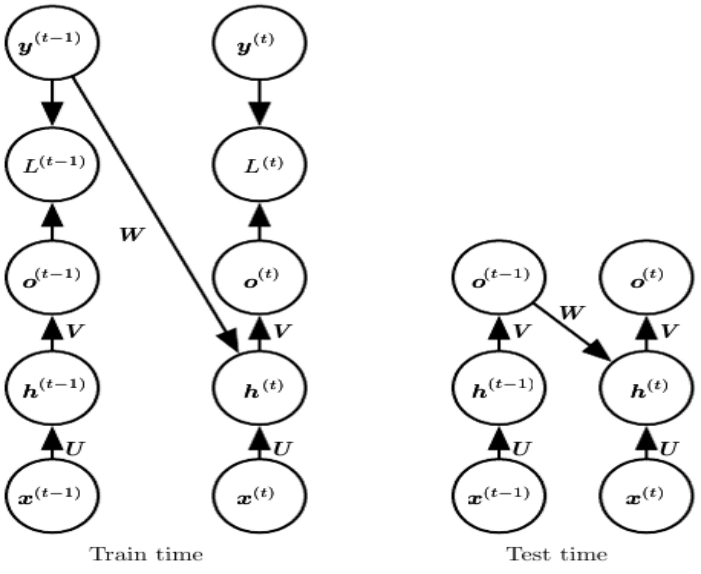


Figure10.6: Illustration of teacher forcing.

- **Computing the Gradient in a Recurrent Neural Network :**
  - Computing the gradient through a recurrent neural network is straightforward. One simply applies the generalized back-propagation algorithm to the unrolled computational graph.
  - The nodes of our computational graph include the parameters U, V, W, b and c as well as the sequence of nodes indexed by t for x()t ,h()t ,o()t and L().
  - We start the recursion with the nodes immediately preceding the final loss.

$$\frac{\partial L}{\partial L^{(t)}} = 1.$$

  - In this derivation we assume that the outputs o()t are used as the argument to the softmax function to obtain the vector ˆy of probabilities over the outpu t.
  - The gradient ∇o( )t L on the outputs at time step t, for all i, t, is as follows:

$$(\nabla_{\boldsymbol{o}^{(t)}} L)_i = \frac{\partial L}{\partial o_i^{(t)}} = \frac{\partial L}{\partial L^{(t)}} \frac{\partial L^{(t)}}{\partial o_i^{(t)}} = \hat{y}_i^{(t)} - \mathbf{1}_{i,y^{(t)}}.$$

  - We work our way backwards, starting from the end of the sequence. At the final time step ,τ h( )τ only has o( )τ as a descendent, so its gradient is simple:

$$\nabla_{\boldsymbol{h}^{(\tau)}} L = \boldsymbol{V}^\top \nabla_{\boldsymbol{o}^{(\tau)}} L.$$

We can then iterate backwards in time to back-propagate gradients through time, from $t = \tau - 1$ down to $t = 1$, noting that $\boldsymbol{h}^{(t)}$ (for $t < \tau$) has as descendents both $\boldsymbol{o}^{(t)}$ and $\boldsymbol{h}^{(t+1)}$. Its gradient is thus given by

$$\nabla_{\boldsymbol{h}^{(t)}} L = \left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \boldsymbol{h}^{(t)}}\right)^\top (\nabla_{\boldsymbol{h}^{(t+1)}} L) + \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}}\right)^\top (\nabla_{\boldsymbol{o}^{(t)}} L) \tag{10.20}$$

$$= \boldsymbol{W}^\top (\nabla_{\boldsymbol{h}^{(t+1)}} L) \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right) + \boldsymbol{V}^\top (\nabla_{\boldsymbol{o}^{(t)}} L) \tag{10.21}$$

where $\operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t+1)}\right)^2\right)$ indicates the diagonal matrix containing the elements $1 - (h_i^{(t+1)})^2$. This is the Jacobian of the hyperbolic tangent associated with the hidden unit i at time $t + 1$.

Using this notation, the gradient on the remaining parameters is given by:

$$\nabla_{\boldsymbol{c}} L = \sum_t \left(\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{c}} \right)^\top \nabla_{\boldsymbol{o}^{(t)}} L = \sum_t \nabla_{\boldsymbol{o}^{(t)}} L \tag{10.22}$$

$$\nabla_{\boldsymbol{b}} L = \sum_t \left(\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{b}^{(t)}} \right)^\top \nabla_{\boldsymbol{h}^{(t)}} L = \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \nabla_{\boldsymbol{h}^{(t)}} L \tag{10.23}$$

$$\nabla_{\boldsymbol{V}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial o_i^{(t)}} \right) \nabla_{\boldsymbol{V}} o_i^{(t)} = \sum_t \left(\nabla_{\boldsymbol{o}^{(t)}} L\right) \boldsymbol{h}^{(t)\top} \tag{10.24}$$

$$\nabla_{\boldsymbol{W}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{W}^{(t)}} h_i^{(t)} \tag{10.25}$$

$$= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{h}^{(t-1)\top} \tag{10.26}$$

$$\nabla_{\boldsymbol{U}} L = \sum_t \sum_i \left(\frac{\partial L}{\partial h_i^{(t)}} \right) \nabla_{\boldsymbol{U}^{(t)}} h_i^{(t)} \tag{10.27}$$

$$= \sum_t \operatorname{diag}\left(1 - \left(\boldsymbol{h}^{(t)}\right)^2\right) \left(\nabla_{\boldsymbol{h}^{(t)}} L\right) \boldsymbol{x}^{(t)\top} \tag{10.28}$$

We do not need to compute the gradient with respect to $\boldsymbol{x}^{(t)}$ for training because it does not have any parameters as ancestors in the computational graph defining the loss.

- **Recurrent Networks as Directed Graphical Models:**
  - One way to interpret an RNN as a graphical model is to view the RNN as defining a graphical model whose structure is the complete graph, able to represent direct dependencies between any pair of y values with the complete graph structure is shown in figure 10.7
  - graph interpretation of the RNN is based on ignoring the hidden units h(t) by marginalizing them out of the model.
  - It is more interesting to consider the graphical model structure of RNNs that results from regarding the hidden units h(t) hidden units in the graphical model reveals that the RNN provides a very efficient parametrization of the joint distribution over the observations.
  - The edges in a graphical model indicate which variables depend directly on other variables.

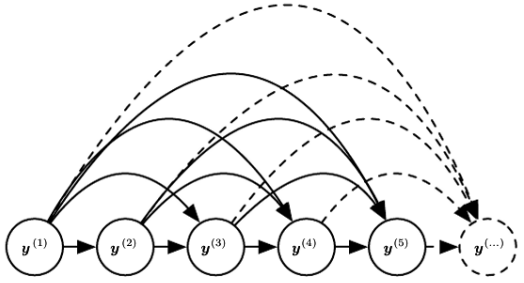


Figure10.7: Fully connected graphical model for a sequence y(1),y(2),...,y()t ,...: every past observation y()i may influence the conditional distribution of some y()t (for t>i), given the previous values.

- RNNs obtain the same full connectivity but efficient parametrization , as illustrated in figure .10.8. is ,

$$L = \sum_t L^{(t)}$$

Where ,

$$L^{(t)} = -\log P(\mathrm{y}^{(t)} = y^{(t)} \mid y^{(t-1)}, y^{(t-2)}, \ldots, y^{(1)}).$$

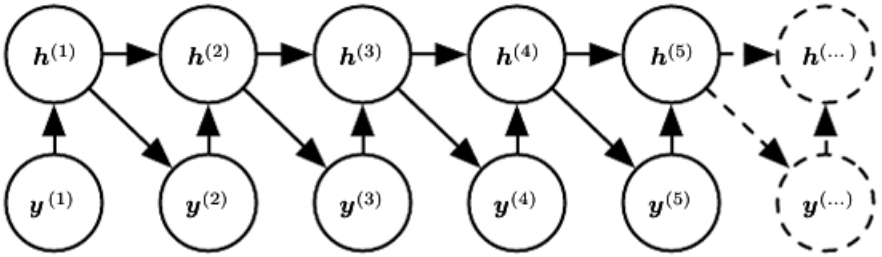


Figure10.8: Introducing the state variable in the graphical model of the RNN.

- **Modeling Sequences Conditioned on Context with RNNs**
  - Modeling Sequences Conditioned on Context with RNNs focuses on how recurrent neural networks (RNNs) can generate or predict sequential data while taking additional contextual information into account. This approach is critical in tasks such as conditional language modeling, video captioning, or time-series prediction conditioned on external variables.
  - Some common ways of providing an extra input to an RNN are:
    1. as an extra input at each time step, or
    2. as the initial state h(0), or
    3. both.

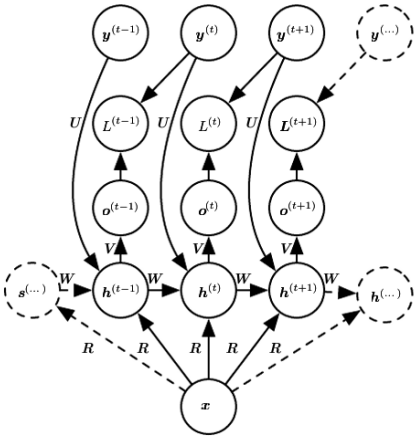


Figure10.9: An RNN that maps a fixed-length vector x into a distribution over sequences Y.

- The RNN described in equation 10.8 corresponds to a conditional distribution P(y(1),...,y( )τ |x(1),...,x( )τ) that makes a conditional independence assumption that this distribution factorizes as

$$\prod_t P(\boldsymbol{y}^{(t)} \mid \boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(t)}).$$

- To remove the conditional independence assumption, we can add connections from the output at time t to the hidden unit at time t+1, as shown in figure.10.10.

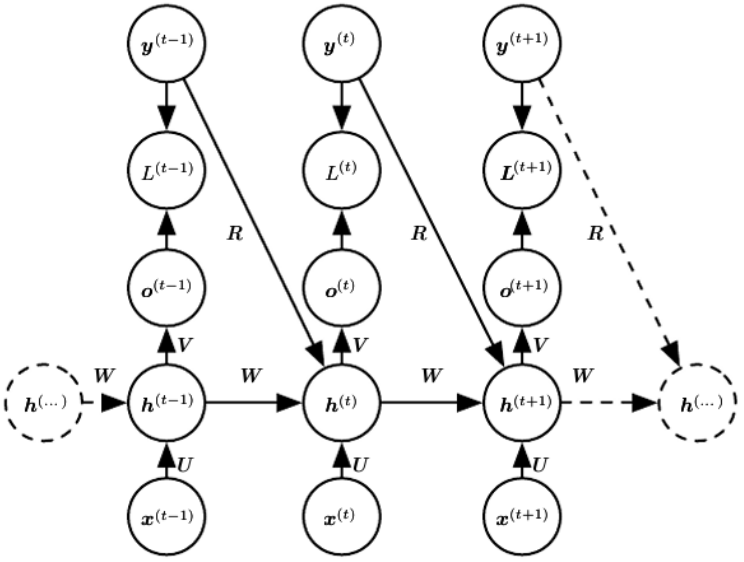


Figure10.10: A conditional recurrent neural network mapping a variable-length sequence of x values into a distribution over sequences of y values of the same length.

- This kind of model representing a distribution over a sequence given another sequence still has one restriction, which is that the length of both sequences must be the same.

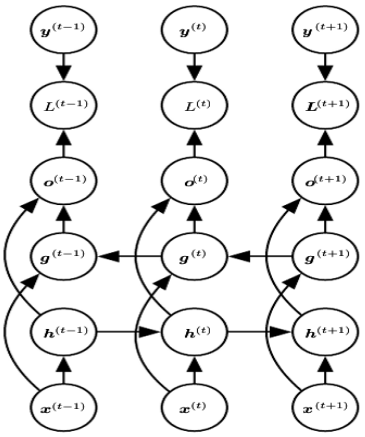
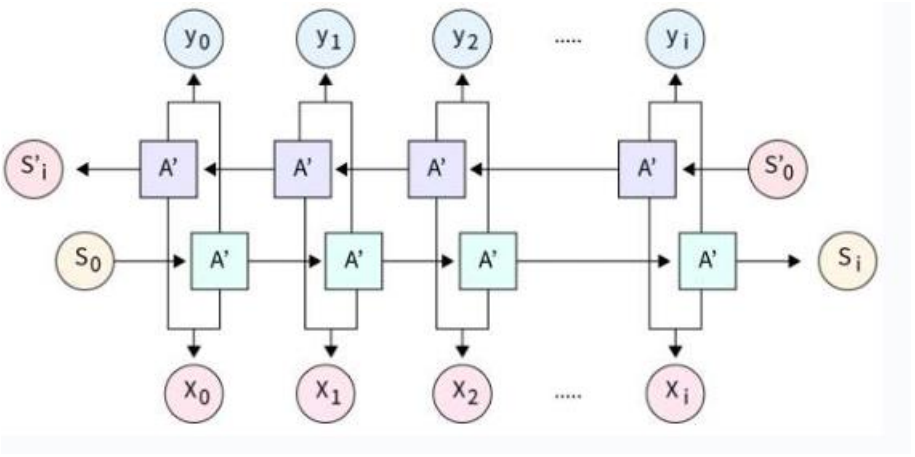


Figure10.11: Computation of a typical bidirectional recurrent neural network.

Bidirectional RNNs:

- A bi-directional recurrent neural network (Bi-RNN) is a type of recurrent neural network (RNN) that processes input data in both forward and backward directions.
- The goal of a Bi RNN is to capture the contextual dependencies in the input data by processing it in both directions, which can be useful in a variety of natural language processing (NLP) tasks.
- This means that the network has two separate RNNs:
  - One that processes the input sequence from left to right
  - Another one that processes the input sequence from right to left.

These two RNNs are typically referred to as the forward and backward RNNs, respectively.



During the forward pass of the RNN, the forward RNN processes the input sequence in the usual way by taking the input at each time step and using it to update the hidden state. The updated hidden state is then used to predict the output at that time step.

- This allows the RNN to capture information from the input sequence that may be relevant to the output prediction, but the same could be lost in a traditional RNN that only processes the input sequence in one direction.
- bidirectional RNNs can help improve the performance of a model on a variety of sequence-based tasks.
- This can be useful for tasks such as language processing, where understanding the context of a word or phrase can be important for making accurate predictions.
- Back-propagation through time (BPTT) is a widely used algorithm for training recurrent neural networks (RNNs). It is a variant of the back-

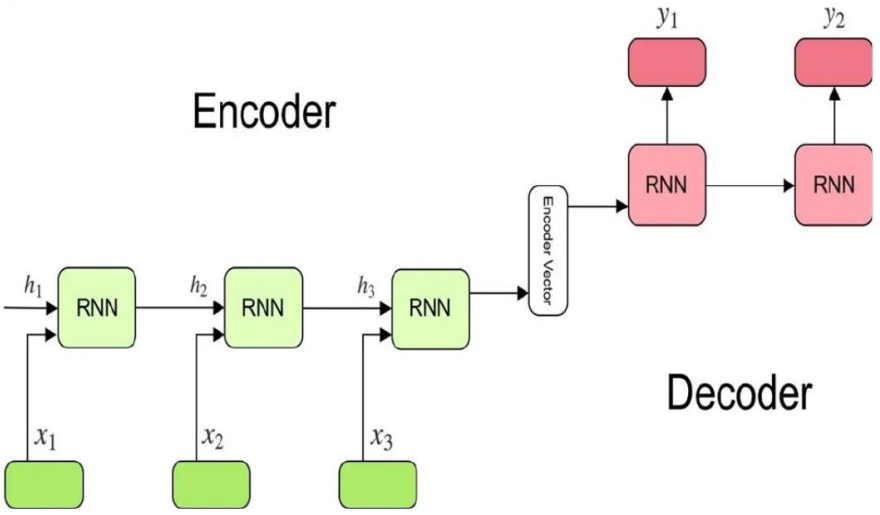
propagation algorithm specifically designed to handle the temporal nature of RNNs, where the output at each time step depends on the inputs and outputs at previous time steps.

- In the case of a bidirectional RNN, BPTT involves two separate Back-propagation passes: one for the forward RNN and one for the backward RNN.
- During the forward pass, the forward RNN processes the input sequence in the usual way and makes predictions for the output sequence.
- During the backward pass, the backward RNN processes the input sequence in reverse order and makes predictions for the output sequence.
- Advantages:
    - Improved performance on tasks that involve processing sequential data. Because bidirectional RNNs can consider information from both past and future time steps when making predictions, they can outperform traditional RNNs on tasks such as natural language processing, time series forecasting, and audio processing.
- Disadvantages:
    - Increased computational complexity. Because bidirectional RNNs have two separate RNNs (one for the forward pass and one for the backward pass), they can require more computational resources to train and evaluate than traditional RNNs.
    - This can make them more difficult to implement and less efficient in terms of runtime performance.

Encoder-Decoder Sequence-to-Sequence Architectures:

- A sequence to sequence model lies behind numerous systems which you face on a daily basis. For instance, seq2seq model powers applications like Google Translate, voice-enabled devices and online chatbots.
- **Definition of the Sequence to Sequence Model:**
  Introduced for the first time in 2014 by Google, a sequence to sequence model aims to map a fixed-length input with a fixed-length output where the length of the input and output may differ.
  For example, translating "What are you doing today?" from English to Chinese has input of 5 words and output of 7 symbols(今天你在做什麼?).Clearly, we can't use a regular LSTM network to map each word from the English sentence to the Chinese sentence.

- This is why the sequence to sequence model is used to address problems like that one.
- In order to fully understand the model's underlying logic, we will go over the below illustration of



The model consists of 3 partsff encoder, intermediate (encoder) vector and decoder.

- **Encoder**
  - A stack of several recurrent units (LSTM or GRU cells for better performance) where each accepts a single element of the input sequence, collects information for that element and propagates it forward.
  - In question-answering problem, the input sequence is a collection of all words from the question. Each word is represented as x_i where i is the order of that word.
  - The hidden states h_i are computed using the formula of

$$h_t = f(W^{(hh)} h_{t-1} + W^{(hx)} x_t)$$

This simple formula represents the result of an ordinary recurrent neural network. As you can see, we just apply the appropriate weights to the previous hidden state h_(t-1) and the input vector x_t.

- **Encoder Vector**
  - This is the final hidden state produced from the encoder part of the model. It is calculated using the formula above.

    - This vector aims to encapsulate the information for all input elements in order to help the decoder make accurate predictions.
    - It acts as the initial hidden state of the decoder part of the model.
- **Decoder**
    - A stack of several recurrent units where each predicts an output y_t at a time step t.
    - Each recurrent unit accepts a hidden state from the previous unit and produces and output as well as its own hidden state.
    - In the question-answering problem, the output sequence is a collection of all words from the answer. Each word is represented as y _i where i is the order of that word.
    - Any hidden state h i is computed using the formula of

$$h_t = f(W^{(hh)} h_{t-1})$$

      As you can see, we are just using the previous hidden state to compute the next one.
    - The output y_t at time step t is computed using the formula of

$$y_t = softmax(W^S h_t)$$

      We calculate the outputs using the hidden state at the current time step together with the respective weight W(S). Softmax is used to create a probability vector which will help us determine the final output (e.g. wordin the question-answering problem).
- The encoder-decoder architecture is a type of RNN design that processes an input sequence of variable length and generates an output sequence of potentially different length. This is particularly useful in tasks like:
    - Machine Translation: Translating a sentence from one language to another.
    - Speech Recognition: Converting speech audio to text.
    - Question Answering: Mapping a question to a textual answer.

Deep Recurrent Networks:

- A deep RNN is simply an RNN with multiple hidden layers stacked on top of each other. This stacking allows the network to learn more complex patterns and representations from the data. Each layer in a RNN can capture different levels of abstraction, making it more powerful than a single-layer RNN.
- Key Idea is that they process one step of a sequence at a time (like reading one word or one frame (video). Information flows not only through the sequence (via time) but also vertically through the layers to capture deeper relationships.
- Example: Imagine analysing a video: A simple RNN might understand how objects in a single frame related to the previous one. A deep RNN could understand more abstract things, like the movement of a person or the mood of the scene.
- Architecture of a Deep RNN:
  Here's a visual representation of a deep RNN architecture:

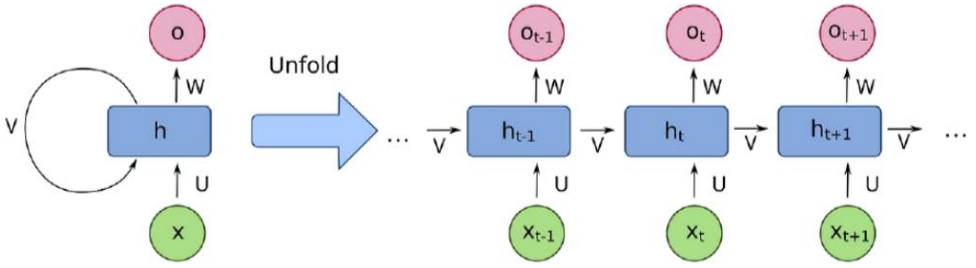


Fig 4. Architecture of a deep neural network.

In the above figure 4, the deep RNN has three hidden layers. The arrows indicate the flow of informatic through the network over time.
Let us now try to understand how a recurrent neural network can be made deep in many ways.

a. The hidden recurrent state can be broken down into groups organized hierarchically.
b. Deeper computation (e.g., an MLP) can be introduced in the input-to-hidden, hidden-to-hidden and hidden-to-output parts. This may lengthen the shortest path linking different time steps.
c. The path-lengthening effect can be mitigated by introducing skip connections.

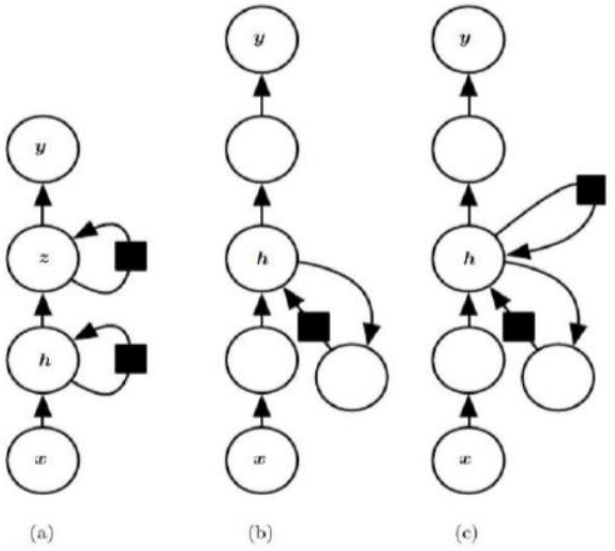


Fig 5. A recurrent neural network can be made deep in many ways.

(a) Hidden State Organized Hierarchically:

- As seen in figure 4(a), the hidden recurrent state is broken into groups or layers that are organized hierarchically.
- The state h is updated at each time step, and it interacts with another intermediate layer z before producing the final output y.
- Hierarchical organization means the recurrent hidden states (h) have additional layers of computational (z) between them, creating a deeper network.
- The hierarchical nature of hidden states aligns with the idea that the hidden recurrent state can broken into groups.

(b) Deeper Computation at Each Stage:

- The shows deeper computation introduced at key parts of the network:
  - Input-to-hidden: The input x undergoes additional transformations before updating the hidden state h.
  - Hidden-to-hidden: The hidden state h involves multiple layers or additional computation before transitioning to the next time step.
  - Hidden-to-output: The output layer y is generated after deeper computation from the hidden state.
- The additional processing lengthens the shortest path connecting information across time steps making the network deeper.

- The deeper computation mentioned in the paragraph is clearly represented here by the additional processing layers between x, h, and y.
- This "deepening" allows for more complex transformations and modelling of the data, but it also lengthens the path for gradients during backpropagation through time.

(c)Skip Connections to Mitigate Path Lengthening:

- In figure 4(c), skip connections are introduced:
- The input x has direct connections to later parts of the network (hidden states h and outputs).
- The hidden state h also has connections that bypass some layers to directly influence the output y.
- These skip connections shorten the gradient path by allowing information to flow directly between layers, mitigating the path-lengthening effect caused by deeper computation.
- The skip connections mentioned in the paragraph are explicitly shown here as shortcuts in the network.
- These connections address the problem of long paths, improving gradient flow and enabling the RNN to handle deeper architectures effectively.

Recursive Neural Networks:

- Recursive neural networks represent yet another generalization of recurrent networks, with a different kind of computational graph, which is structured as a deep tree, rather than the chain-like structure of RNNs. The typical computational graph for a recursive network is illustrated in figure 5.

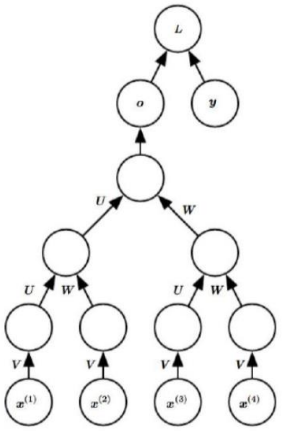


Figure 5. A typical graph of recursive neural network.

- Recursive networks were first proposed by Pollack (1990), with their potential application for reasoning highlighted by Bottou (2011). These networks have been effectively utilized for processing structured data as inputs to neural networks (Frasconi et al., 1997, 1998), as well as in natural language processing and computer vision.
- One clear advantage of recursive nets over recurrent nets is that for a sequence of the same length τ, the depth (measured as the number of compositions of nonlinear operations) can be drastically reduced from τ to $O(\log\tau)$, which might help deal with long-term dependencies.
- In certain domains, external methods can provide suitable tree structures. For instance, in natural language processing, the tree structure can be aligned with the parse tree of a sentence generated by a natural language parser. Ideally, the learner itself would infer and discover the most appropriate tree structure that a given input, as suggested by Bottou (2011).
- There are numerous possible variants of the recursive network concept. For instance, Frasconi et al. (1997)&(1998) associate the data with a tree structure where inputs and targets are linked to individual nodes of tree.
- The computations performed at each node are not limited to the conventional artificial neuron operation(an affine transformation followed by a monotonic nonlinearity). For example, Socher et al. (2013a) propose the use of tensor operations and bilinear forms, which have previously been shown to effective model relationships between concepts (Weston et al., 2010; Bordes et al., 2012) when the concepts represented as continuous vector embeddings.
- **Recurrent Neural Network Vs. Recursive Neural Networks**

Aspect	Recurrent Neural Networks (RNNs)	Recursive Neural Networks
Type	A class of neural networks used for processing sequential data.	A class of neural networks that operate on hierarchical data structures.
Structure	Usually have a chain-like structure where each node represents a time step.	Have a tree-like structure where each node can have a fixed number of children.
Data Representation	Represent temporal sequences, such as sentences, speech, or time series.	Represent hierarchical models, such as parse trees in NLP.

Application Areas	Mainly used in Natural Language Processing (NLP) and other sequential tasks like speech recognition and text generation.	Used in semantic analysis, syntactic parsing, and other tasks involving structured data.
Weight Sharing	Weights are shared across all time steps in the sequence, keeping dimensionality constant.	Weights are shared across all nodes in the tree, applying the same transformation recursively.
Computation Process	Processes data sequentially over time.	Processes data recursively over a hierarchical structure.
Efficiency	Efficient for sequential data but may suffer from vanishing gradient problems.	Generally more efficient than feed-forward networks for hierarchical data.
Relation Between Them	Recurrent networks are recurrent over time.	Recursive networks are a generalization of recurrent networks.

- **Benefits of RvNNs for Natural Language Processing:**
  - The two significant advantages of Recursive Neural Networks for Natural Language Processing are their structure and reduction in network depth.
  - As already explained, the tree structure of Recursive Neural Networks can manage hierarchical data like in parsing problems.
  - Another benefit of RvNN is that the trees can have a logarithmic height. When there are O(n) input words, a Recursive Neural Network can represent a binary tree with height O(log\ n).
  - This lessens the distance between the first and last input elements. Hence, the long-term dependency turns shorter and easier to grab.
- **Disadvantages of RvNNs for Natural Language Processing:**
  - The main disadvantage of recursive neural networks can be the tree structure. Using the tree structure indicates introducing a unique inductive bias to our model.
  - The bias corresponds to the assumption that the data follow a tree hierarchy structure.
  - But that is not the truth. Thus, the network may not be able to learn the existing patterns.
  - Another disadvantage of the Recursive Neural Network is that sentence parsing can be slow and ambiguous.
  - Interestingly, there can be many parse trees for a single sentence.

- Also, it is more time-consuming and labor-intensive to label the training data for recursive neural networks than to construct recurrent neural networks.
- Manually parsing a sentence into short components is more time-consuming and tedious than assigning a label to a sentence

The Challenge of Long-Term Dependencies:

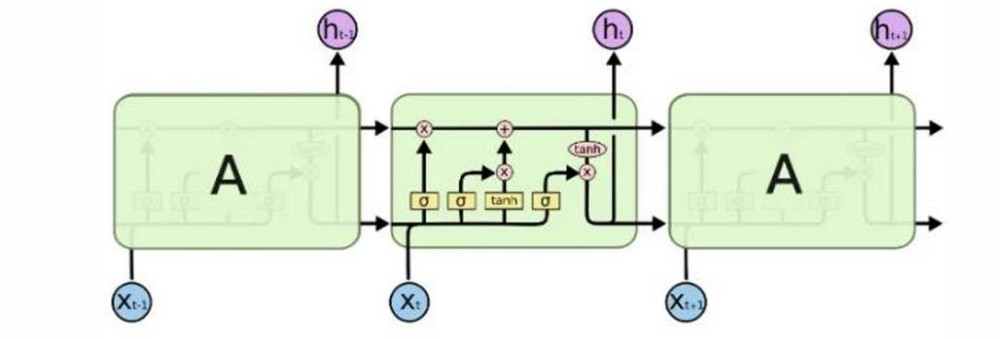
- **What are long-term dependencies ?**
  - Long-term dependencies are the situations where the output of an RNN depends on the input that occurred many time steps ago.
  - For instance, consider the sentence "The cat, which was very hungry, ate the mouse".
  - To understand the meaning of this sentence, you need to remember that the cat is the subject of the verb ate, even though they are separated by a long clause.
  - This is a long-term dependency, and it can affect the performance of an RNN that tries to generate or analyze such sentences.
- **Why are long-term dependencies?**
  - Recurrent neural networks (RNNs) are powerful machine learning models that can process sequential data, such as text, speech, or video.
  - However, they often struggle to capture long-term dependencies, which are the relationships between distant elements in the sequence.
- **Why are long-term dependencies hard to learn?**
  - The main reason why long-term dependencies are hard to learn is that RNNs suffer from the vanishing or exploding gradient problem.
  - This means that the gradient, which is the signal that tells the network how to update its weights, becomes either very small or very large as it propagates through the network.
  - When the gradient vanishes, the network cannot learn from the distant inputs, and when it explodes, the network becomes unstable and produces erratic outputs.
  - This problem is caused by the repeated multiplication of the same matrix, which represents the connections between the hidden units, at each time step.

- **How can you use gated units to handle long-term dependencies?**
  - Another way to handle long-term dependencies is to use gated units, which are special types of hidden units that can control the flow of information in the network.
  - The most popular gated units are the long short-term memory (LSTM) and the gated recurrent unit (GRU).
  - These units have internal mechanisms that allow them to remember or forget the previous inputs and outputs, depending on the current input and output.
  - This way, they can selectively access the relevant information from the and ignore the irrelevant information such as distant past.
- **How can you use attention mechanisms to handle long-term dependencies?**
  - Another way to handle long-term dependencies is to use attention mechanisms, which are modules that can learn to focus on the most important parts of the input or output sequence.
  - The most common attention mechanism is the self-attention, which computes the similarity between each element in the sequence and assigns a weight to each one.
  - Then, it uses these weights to create a context vector, which summarizes the information from the whole sequence.
  - This way, it can capture the relationships between the distant elements and enhance the representation of the sequence.
- **Challenges**
  - Neural network optimization face a difficulty when computational graphs become deep, e.g.,
    . Feedforward networks with many layers
    . RNNs that repeatedly apply the same operation at each time step of a long temporal sequence
  - Gradients propagated over many stages tend to either vanish (most of the time) or explode (damaging optimization)
  - The difficulty with long-term dependencies arise from exponentially smaller weights given to long-term interactions (involving multiplication of many Jacobians)
- **LSTM :The Solution To Long Term Dependencies**
  - Sometimes we just need to look at recent information to perform the present task.

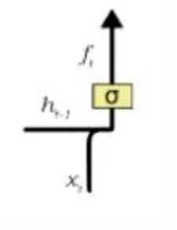
- For example, consider a language model trying to predict the last word in " the clouds are in the sky".
- Here it's easy to predict the next word as sky based on the previous words. But consider the sentence I grew up in France I speak fluent French.
- " Here it is not easy to predict that the language is French directly.
- It depends on previous input also.
- In such sentences it's entirely possible for the gap between the relevant information and the point where it is needed to become very large.
- In theory, RNN's are absolutely capable of handling such "long-term dependencies."
- A human could carefully pick parameters for them to solve toy problems of this form.
- Sadly, in practice, recurrent neural network don't seem to be able to learn them.
- This problem is called Vanishing gradient problem.
- The neural network updates the weight using the gradient descent algorithm.
- The gradients grow smaller when the network progress down to lower layers.
- The gradients will stay constant meaning there is no space for improvement.
- The model learns from a change in the gradient.
- This change affects the network's output.
- However, if the difference in the gradients is very small network will not learn anything and so no difference in the output.
- Therefore, a network facing a vanishing gradient problem cannot converge towards a good solution.

- **Long Short Term Memory networks**
  - Long Short Term Memory networks (LSTMs) is a special kind of recurrent neural network capable of learning long-term dependencies.
  - They were introduced by Hochreiter & Schmidhuber in 1997.
  - Remembering information for longer periods of time is their default behavior.

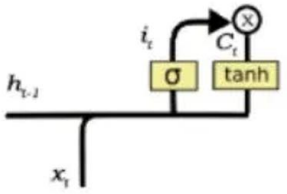
- The Long short-term memory (LSTM) is made up of a memory cell, an input gate, an output gate and a forget gate.
- The memory cell is responsible for remembering the previous state while the gates are responsible for controlling the amount of memory to be exposed.
- The memory cell is responsible for keeping track of the dependencies between the elements in the input sequence.
- The present input and the previous is passed to forget gate and the output of this forget gate is fed to the previous cell state.
- After that the output from the input gate is also fed to the previous cell state.
- By using this the output gate operates and will generate the output.



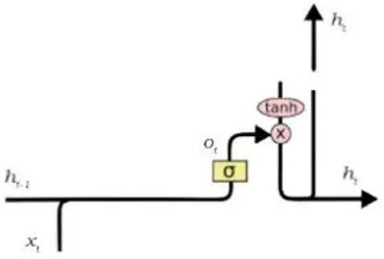
- **Forget Gate**:
  - There are some information from the previous cell state that is not needed for the present unit in a LSTM.
  - A forget gate is responsible for removing this information from the cell state.
  - The information that is no longer required for the LSTM to understand or the information that is of less importance is removed via multiplication of a filter.
  - This is required for optimizing the performance of the LSTM network.
  - In other words we can say that it determines how much of previous state is to be passed to the next state.
  - The gate has two inputs X t and h t-1. h t-1 is the output of the previous cell and x t is the input at that particular time step.
  - The given inputs are multiplied by the weight matrices and a bias is added. Following this, the sigmoid function(activation function) is applied to this value.



- **Input Gate**:
  - The process of adding new information takes place in input gate.
  - Here combination of x t and h t-1 is passed through sigmoid and tanh functions(activation functions) and added.
  - Creating a vector containing all the possible values that can be added (as perceived from h t-1 and x t) to the cell state.
  - This is done using the tanh function. By this step we ensure that only that information is added to the cell state that is important and is not redundant.



- **Output Gate**:
  - A vector is created after applying tanh function to the cell state.
  - Then making a filter using the values of h t-1 and x t, such that it can regulate the values that need to be output from the vector created above.
  - This filter again employs a sigmoid function.
  - Then both of them are multiplied to form output of that cell state.



Echo State Networks:

- Echo State Networks (ESNs) are a specific kind of recurrent neural network (RNN) designed to efficiently handle sequential data.
- In Python, ESNs are created as a reservoir computing framework, which includes a fixed, randomly initialized recurrent layer known as the “reservoir.”

- The key feature of ESNs is their ability to make the most of the echo-like dynamics of the reservoir, allowing them to effectively capture and replicate temporal patterns in sequential input.
- The way ESNs work is by linearly combining the input and reservoir states to generate the network's output.
- The reservoir weights remain fixed, this unique approach makes ESNs particularly useful in tasks where capturing temporal dependencies is critical, such as time-series prediction and signal processing.
- Implementing ESNs in Python, researchers and practitioners often turn to libraries like PyTorch or specialized reservoir computing frameworks.
- **WHAT IS ECHOSTATE NETWORKS?**
    - Echo State Networks (ESNs) in Python are a fascinating type of recurrent neural network (RNN) tailored for handling sequential data.
    - Imagine it as a three-part orchestra: there's the input layer, a reservoir filled with randomly initialized interconnected neurons, and the output layer.
    - The magic lies in the reservoir, where the weights are like a musical improvisation– fixed and randomly assigned.
    - This creates an "echo" effect, capturing the dynamics of the input signal.
    - During training, we tweak only the output layer, guiding it to map the reservoir's states to the desired output.
    - An Echo State Network (ESN) in Python is like a smart system that can predict what comes next in a sequence of data.
    - Imagine you have a list of numbers or values, like the temperature each day.
    - An ESN can learn from this data and then try to guess the temperature for the next day.

      1. Reservoir: It has a special part called a "reservoir," which is like a pool of interconnected neurons. These neurons work together to remember patterns in the data.
      2. Training: We show the ESN some of our data and let it learn. It doesn't learn everything but gets the hang of the patterns.
      3. Predicting: Now, when we give the ESN a new piece of data (like the past temperatures), it uses what it learned to make a guess about what comes next.

4. Output: The ESN gives us its prediction, and we can compare it to the real answer. If it's good, great! If not, we might need to tweak things a bit.

- **Applications of Echo-State Networks**
    1. Time Series Prediction: ESNs excel at predicting future values in a time series.
    2. Efficient Training: ESNs have a unique training approach.While the reservoir is randomly generated and fixed, only the output weights are trained.
    3. Nonlinear Mapping: The reservoir in an ESN introduces nonlinearity to the model.
    4. Robustness to Noise: ESNs are known for being robust to noise in the input data.
    5. Universal Approximator: Theoretically, ESNs are capable of approximating any dynamical system.
    6. Ease of Implementation: Implementing ESNs is often simpler compared to training traditional RNNs.
- **Concepts of Echo-State Networks**
    1. Reservoir Computing: Reservoir Computing is a framework that includes Echo State Networks. In ESNs, the reservoir is a dynamic memory of randomly initialized recurrent neurons that captures temporal patterns in sequential data.
    2. Reservoir: The reservoir is a fixed and randomly initialized collection of recurrent neurons in an ESN.
    3. Echo State Property: The Echo State Property is a key characteristic of the reservoir, indicating that its dynamics amplify and retain information from the input over time.
    4. Input Layer: The input layer of an ESN receives sequential input data. Each input corresponds to a time step in the sequence.
    5. Output Layer: The output layer produces the final result based on the combination of the reservoir states. It is typically a linear combination of the reservoir states, and only the output layer is trained.
    6. Training: During training, the ESN learns to map the reservoir states to the desired output.
    7. Fixed Weights: The weights in the reservoir are randomly initialized and remain fixed during training.

Leaky Units and Other Strategies for Multiple Time Scales:

- Models can handle long-term dependencies by operating at multiple time scales, where some parts handle fine-grained short-term details and others manage coarse long-term information.
- Common strategies include skip connections across time, leaky units, and removing fine-grained connections.
- Skip connections add direct links from past time steps to the present to capture long-term dependencies and reduce gradient vanishing. Introducing a time delay in recurrent connections helps gradients diminish more slowly, improving the learning of long-term relations.
- Leaky units with linear self-connections and weights near one allow the network to retain information for longer periods, acting like running averages.
- The α parameter in leaky units controls memory retention — when α is near one, past information is retained longer, and when α is near zero, it is quickly forgotten.
- Time constants in leaky units can either be fixed manually or learned automatically during training.
- Removing short-length connections forces units to operate on longer time scales, improving long-term dependency handling.
- Different groups of recurrent units can be designed to operate at different time scales or update frequencies, as proposed by Mozer (1992), El Hihi and Bengio (1996), and Koutnik et al. (2014).
- Additionally, having leaky units operating at different time scales helps the model balance short-term and long-term information more effectively.
- The linear self-connection approach in leaky units provides a smoother and more flexible way to access past information compared to fixed skip lengths.
- Organizing the RNN state at multiple time scales allows information to flow more easily over long distances, improving performance on tasks involving long-term dependencies.
- The use of both delayed and single-step connections allows the model to learn from recent as well as distant past information simultaneously.
- Leaky units have proven effective in architectures like **Echo State Networks**, enhancing memory retention and stability.

The Long Short-Term Memory and Other Gated RNNs:

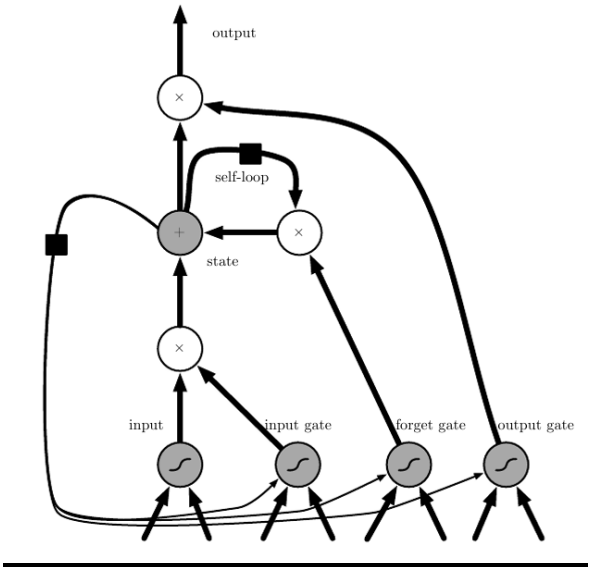


Figure : Block diagram of the LSTM recurrent network cell.

Long Short-Term Memory:

- LSTM excels in sequence prediction tasks, capturing long-term dependencies.
- Ideal for time series, machine translation, and speech recognition due to order dependence.
- **What is LSTM?**

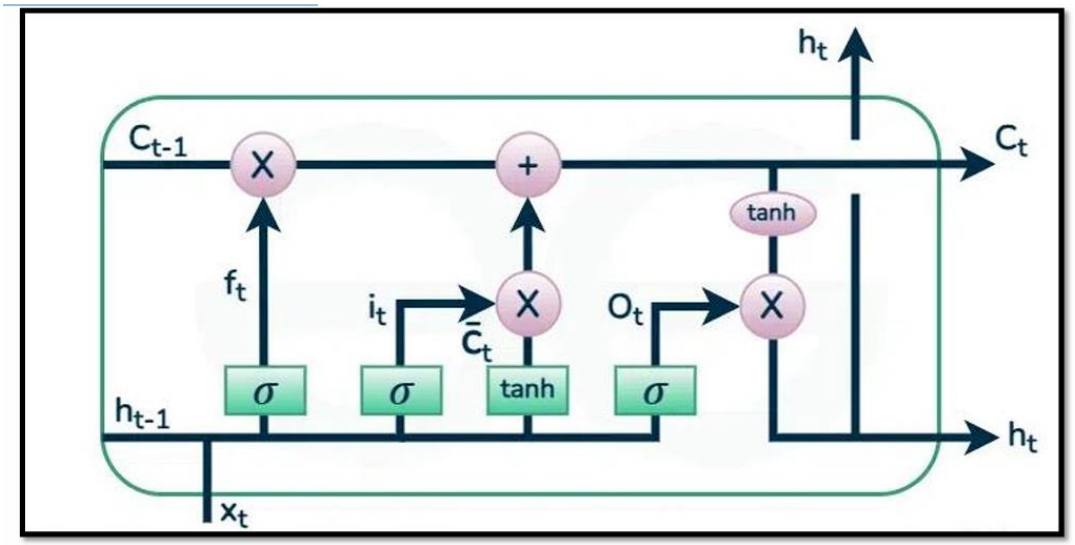
  Long Short-Term Memory is an improved version of recurrent neural network designed by Hochreiter & Schmidhuber.
- Atraditional RNN has a single hidden state that is passed through time, which can make it difficult for the network to learn long-term dependencies.
- LSTMs model address this problem by introducing a memory cell, which is a container that can hold information for an extended period.
- LSTM architectures are capable of learning long-term dependencies in sequential data, which makes them well-suited for tasks such as language translation, speech recognition, and time series forecasting.
- **LSTM Architecture:**
  - The LSTM architectures involves the memory cell which is controlled by three gates: the input gate, the forget gate, and the output gate.
  - These gates decide what information to add to, remove from, and output from the memory cell.

1. The input gate controls what information is added to the memory cell.
2. The forget gate controls what information is removed from the memory cell.
3. The output gate controls what information is output from the memory cell.

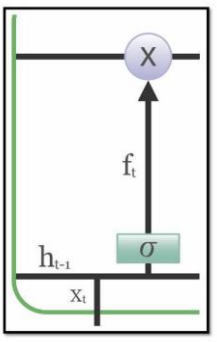
- This allows LSTM networks to selectively retain or discard information as it flows through the network, which allows them to learn long-term dependencies.
- The LSTM maintains a hidden state, which acts as the short-term memory of the network.
- The hidden state is updated based on the input, the previous hidden state, and the memory cell's current state.

- **Bidirectional LSTM Model:**
  - Bidirectional LSTM (Bi LSTM/ BLSTM) is recurrent neural network (RNN) that is able to process sequential data in both forward and backward directions.
  - This allows Bi LSTM to learn longer-range dependencies in sequential data than traditional LSTMs, which can only process sequential data in one direction.
  - Bi LSTMs are made up of two LSTM networks, one that processes the input sequence in the forward direction and one that processes the input sequence in the backward direction.

    1. The outputs of the two LSTM networks are then combined to produce the final output.
    2. Networks in LSTM architectures can be stacked to create deep architectures, enabling the learning of even more complex patterns and hierarchies in sequential data.
  - Each LSTM layer in a stacked configuration captures different levels of abstraction and temporal dependencies within the input data.
- **LSTM Working:** LSTM architecture has a chain structure that contains four neural networks and different memory blocks called cells.



- Information is retained by the cells and the memory manipulations are done by the three gates: Forget Gate, Input gate, Output gate.
- **Forget Gate:**
  - The information that is no longer useful in the cell state is removed with the forget gate.
  - Two inputs xt (input at the particular time) and ht-1 (previous cell output) are fed to the gate and multiplied with weight matrices followed by the addition of bias.
  - The resultant is passed through an activation function which gives a binary output.
  - If for a particular cell state the output is 0, the piece of information is forgotten and for output 1, the information is retained for future use.



  - The equation for the forget gate is:

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f)$$

where:

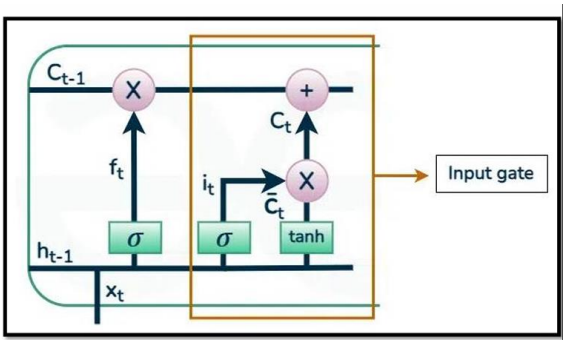
W_f represents the weight matrix associated with the forget gate.

[h_t-1, x_t] denotes the concatenation of the current input and the previous hidden state.

b_f is the bias with the forget gate.

σ is the sigmoid activation function.

- **Input gate:**
  - The addition of useful information to the cell state is done by the input gate.
  - First, the information is regulated using the sigmoid function and filter the values to be remembered similar to the forget gate using inputs ht-1 and xt. .
  - Then, a vector is created using tanh function that gives an output from-1 to +1, which contains all the possible values from ht-1 and xt.
  - At last, the values of the vector and the regulated values are multiplied to obtain the useful information.



  - The equation for the input gate is:

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i)$$

$$\hat{C}_t = tanh(W_c \cdot [h_{t-1}, x_t] + b_c)$$

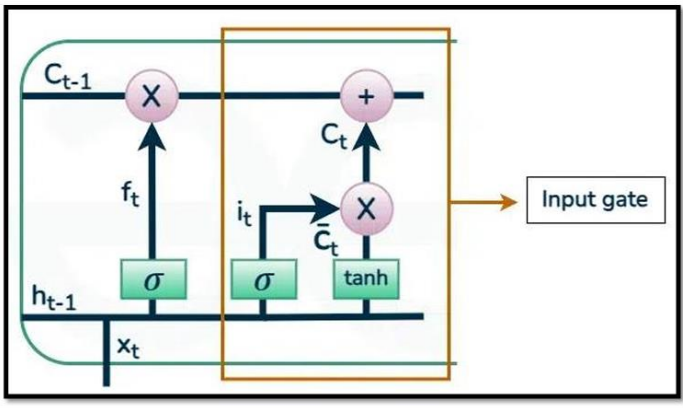
We multiply the previous state by ft, disregarding the information we had previously chosen to ignore. Next, we include it*Ct. This represents the updated candidate values, adjusted for the amount that we chose to update each state value.

$$C_t = f_t \odot C_{t-1} + i_t \odot \hat{C}_t$$

where

- ⊙ denotes element-wise multiplication
- tanh is tanh activation function

- **Output gate:**
  - **The task of extracting useful information from the current cell state to be presented as output is done by the output gate.**
  - **First, a vector is generated by applying tanh function on the cell.**
  - **Then, the information is regulated using the sigmoid function and filter by the values to be remembered using inputs ht−1 and xt .**
  - **At last, the values of the vector and the regulated values are multiplied to be sent as an output and input to the next cell.**



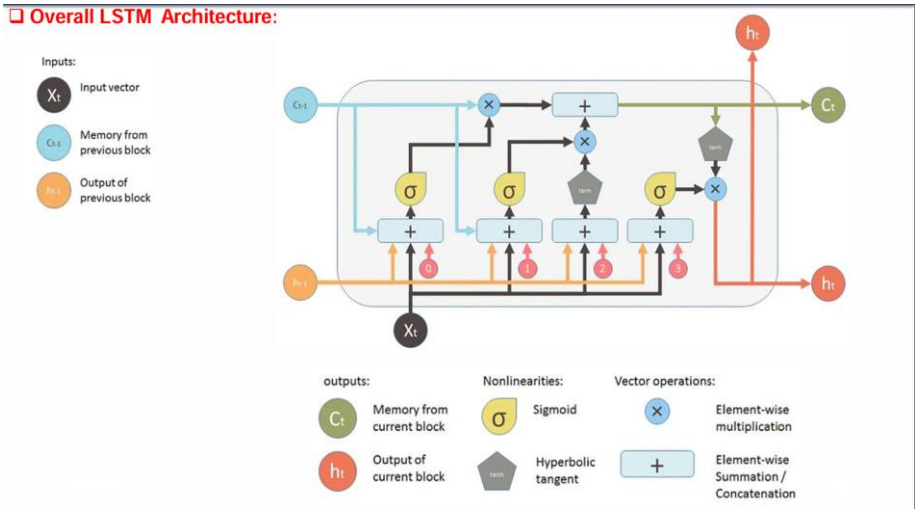
  - **The equation for the output gate is:**

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o)$$

- **LSTM Applications:**

  1. Language Modeling: LSTMs have been used for natural language processing tasks such as language modeling, machine translation, and text summarization.
  2. Speech Recognition: LSTMs have been used for speech recognition tasks such as transcribing speech to text and recognizing spoken commands.
  3. Time Series Forecasting: LSTMs have been used for time series forecasting tasks such as predicting stock prices, weather, and energy consumption.
  4. Anomaly Detection: LSTMs have been used for anomaly detection tasks such as detecting fraud and network intrusion.
  5. Recommender Systems: LSTMs have been used for recommendation tasks such as recommending movies, music, and books.

6. Video Analysis: LSTMs have been used for video analysis tasks such as object detection, activity recognition, and action classification.



Other Gated RNNs Gated Recurrent Unit :

- **What is Gated Recurrent Unit(GRU)?**
  - GRU stands for Gated Recurrent Unit, which is a type of recurrent neural network (RNN) architecture that is similar to LSTM (Long Short-Term Memory).
  - Like LSTM, GRU is designed to model sequential data by allowing information to be selectively remembered or forgotten over time.
  - However, GRU has a simpler architecture than LSTM, with fewer parameters, which can make it easier to train and more computationally efficient.
  - The main difference between GRU and LSTM is the way they handle the memory cell state.
  - In LSTM, the memory cell state is maintained separately from the hidden state and is updated using three gates: the input gate, output gate, and forget gate.
  - In GRU, the memory cell state is replaced with a “candidate activation vector,” which is updated using two gates: the reset gate and update gate.
  - The reset gate determines how much of the previous hidden state to forget, while the update gate determines how much of the candidate activation vector to incorporate into the new hidden state.

- Overall, GRU is a popular alternative to LSTM for modeling sequential data, especially in cases where computational resources are limited or where a simpler architecture is desired.

- **How GRUWorks ?**
  - Like other recurrent neural network architectures, GRU processes sequential data one element at a time, updating its hidden state based on the current input and the previous hidden state.
  - At each time step, the GRU computes a “candidate activation vector” that combines information from the input and the previous hidden state.
  - This candidate vector is then used to update the hidden state for the next time step.
  - The candidate activation vector is computed using two gates: the reset gate and the update gate.
  - The reset gate determines how much of the previous hidden state to forget, while the update gate determines how much of the candidate activation vector to incorporate into the new hidden state.
  - Here’s the math behind the GRU architecture:
    1. The reset gate r and update gate z are computed using the current input x and the previous hidden state h_t-1

       r_t = sigmoid(W_r * [h_t-1, x_t])

       z_t = sigmoid(W_z * [h_t-1, x_t])

       where W_r and W_z are weight matrices that are learned during training.
    2. The candidate activation vector h_t~ is computed using the current input x and a modified version of the previous hidden state that is "reset" by the reset gate:

       h_t~ = tanh(W_h * [r_t * h_t-1, x_t])

       where W_h is another weight matrix.
    3. The new hidden state h_t is computed by combining the candidate activation vector with the previous hidden state, weighted by the update gate:

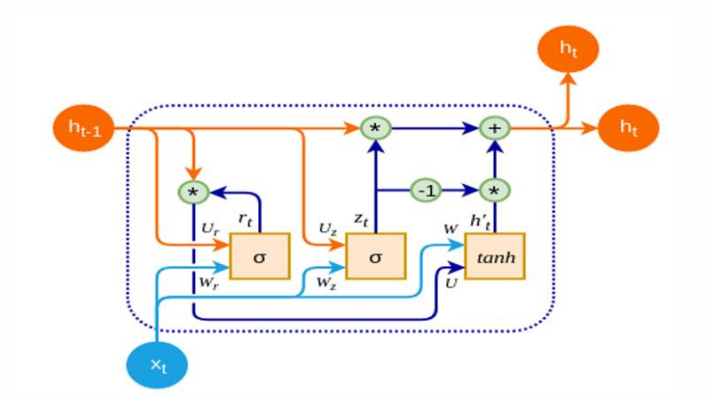
       h_t = (1 - z_t) * h_t-1 + z_t * h_t~

    4. Overall, the reset gate determines how much of the previous hidden state to remember or forget, while the

update gate determines how much of the candidate activation vector to incorporate into the new hidden state.

5. The result is a compact architecture that is able to selectively update its hidden state based on the input and previous hidden state, without the need for a separate memory cell state like in LSTM.

- **GRU Architecture**



1. Input layer: The input layer takes in sequential data, such as a sequence of words or a time series of values, and feeds it into the GRU.
2. Hidden layer: The hidden layer is where the recurrent computation occurs. At each time step, the hidden state is updated based on the current input and the previous hidden state. The hidden state is a vector of numbers that represents the network's "memory" of the previous inputs.
3. Reset gate: The reset gate determines how much of the previous hidden state to forget. It takes as input the previous hidden state and the current input, and produces a vector of numbers between 0 and 1 that controls the degree to which the previous hidden state is "reset" at the current time step.
4. Update gate: The update gate determines how much of the candidate activation vector to incorporate into the new hidden state. It takes as input the previous hidden state and the current input, and produces a vector of numbers between 0 and 1 that controls the degree to which the candidate activation vector is incorporated into the new hidden state.
5. Candidate activation vector: The candidate activation vector is a modified version of the previous hidden state that is "reset" by the reset gate and combined with the current input. It is computed using a tanh activation function that squashes its output between-1 and

6. Output layer: The output layer takes the final hidden state as input and produces the network's output. This could be a single number, a sequence of numbers, or a probability distribution over classes, depending on the task at hand.

Optimization for Long- Term Dependencies:

- Optimizing for long-term dependencies in models, especially in the context of sequence data (like time series or natural language), can be challenging.
- Here are some strategies that are commonly used to enhance the ability of models to capture these dependencies:
    1. Recurrent Neural Networks (RNNs)
        - LSTM and GRU: Use Long Short-Term Memory (LSTM) networks or Gated Recurrent Units (GRUs), which are designed to remember information for long periods and mitigate issues like vanishing gradients.
    2. Attention Mechanisms
        - Self-Attention: Incorporate self-attention mechanisms, as seen in Transformer models. These allow the model to weigh the importance of different parts of the input sequence, effectively capturing long-range dependencies.
        - Multi-Head Attention: Using multiple attention heads can help the model focus on different parts of the sequence simultaneously.
    3. Positional Encoding
        - In Transformer models, use positional encoding to maintain information about the position of elements in the sequence, which helps in understanding the order and context over long ranges.
    4. Hierarchical Models
        - Implement hierarchical structures that process data at different levels of granularity. For example, you can first capture local dependencies and then model global dependencies.
    5. Dilated Convolutions
        - Use dilated convolutions in convolutional neural networks (CNNs) to capture wider contexts without increasing the number of parameters significantly.

6. Memory-Augmented Networks
    - Consider models that utilize external memory (like Neural Turing Machines or Differentiable Neural Computers) to store and retrieve information over long periods.
7. Regularization Techniques
    - Employ regularization methods to prevent overfitting, which can make the model less sensitive to long-term dependencies.
8. Feature Engineering
    - Design features that explicitly capture long-term trends, such as moving averages or seasonal indicators in time series data.
9. Data Augmentation
    - Use techniques that artificially expand the training dataset to expose the model to more varied long term dependencies.

10. Gradient Clipping
    - Implement gradient clipping to manage exploding gradients, allowing the model to learn from longer sequences without losing important information.
11. Batch Normalization and Layer Normalization
    - Normalize activations to stabilize learning and allow the model to better capture dependencies.
12. Training Techniques
    - Use techniques like curriculum learning, where the model is first trained on simpler sequences before gradually increasing complexity.
13. Advanced Architectures
    - Explore architectures like Transformers with memory networks or other novel designs that inherently address long-range dependencies.

Explicit Memory:

- Intelligence requires both implicit and explicit knowledge, but neural networks mainly excel at learning implicit forms of knowledge such as recognizing patterns, while they struggle to memorize explicit facts.
- Neural networks need many repetitions of the same input to store precise information, which makes fact memorization inefficient.

- Graves et al. (2014b) suggested that this limitation arises because neural networks lack a working memory system similar to that of humans, which allows for holding and manipulating information intentionally.
- To address this, Weston et al. (2014) introduced memory networks that include external memory cells with an addressing mechanism for storing and retrieving information.
- Graves et al. (2014b) further developed the Neural Turing Machine (NTM), which learns to read and write arbitrary content in memory without explicit supervision, using a soft attention mechanism.
- This mechanism allows end-to-end training by assigning weighted coefficients to memory cells, enabling smooth gradient-based learning.
- Memory cells in such architectures often store vectors rather than scalars, allowing for more complex, content-based addressing similar to recalling a song from partial lyrics.
- Explicit memory models can propagate information and gradients over longer time durations, enabling them to learn tasks that traditional RNNs or LSTMs cannot.
- In some architectures, stochastic methods are used to read single memory cells based on probabilities, but training such models remains more challenging.
- The addressing mechanism in these memory systems is closely related to attention mechanisms, which were first introduced for tasks like handwriting generation and later popularized in machine translation by Bahdanau et al. (2015).

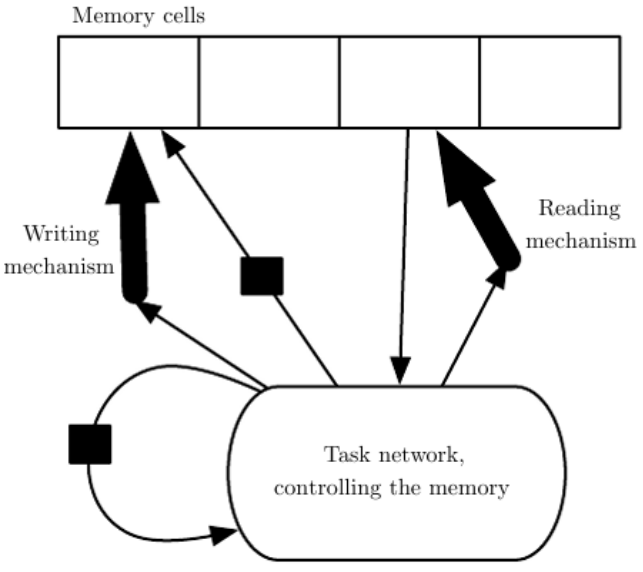


Figure : A schematic of an example of a network with an explicit memory, capturing some of the key design elements of the neural Turing machine.