ATME COLLEGE OF ENGINEERING

13th KM Stone, Bannur Road, Mysore - 560 028

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

LECTURE NOTES

COURSE: Deep Learning

COURSE CODE: BCS714A

SEMESTER: VII

COURSE COORDINATOR: Mrs. Harshitha M K

(ACADEMIC YEAR 2026-27)

MODULE 4

Sequence Modelling: Recurrent and Recursive Nets: Unfolding Computational Graphs, Recurrent Neural Networks, Bidirectional RNNs, Encoder-Decoder Sequence-to-Sequence Architectures, Deep Recurrent Networks, Recursive Neural Networks. Long short-term memory.

Sequence Modeling: Recurrent and Recursive Nets:

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 $x^{(1)}$ ... $x^{(t)}$. Just as convolutional networks can readily scale to images with large width and height, and some convolutional networks can process images of variable size, recurrent networks can scale to much longer sequences than would be practical for networks without sequence-based specialization. Most recurrent networks can also process sequences of variable length.

The Recurrent Neural Network:

- To transition from multi-layer networks to recurrent networks, we leverage a concept from the 1980s in machine learning and statistical Modeling: parameter sharing across different parts of a model.
- This approach allows the model to handle inputs of varying forms (in this case, different sequence lengths) and to generalize effectively.
- 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.
- For simplicity, we describe RNNs as processing a sequence of vectors $x^{(t)}$, where the time step index t ranges from 1 to τ.
- In practice, however, recurrent networks typically operate on minibatches of sequences, where each sequence in the batch may have a different length τ. To keep the notation simple, we omit the minibatch indices.
- Additionally, the time step index t does not always represent the actual passage of time; it often simply indicates the position within the sequence.
- RNNs can also be applied in two dimensions for spatial data, such as images. Even when working with time-based data, the network can include connections that move backward in time, as long as the entire sequence is available before being processed by the network.
- The next section extends the idea of a computational graph to include cycles. These cycles represent the influence of the present value of a variable on its own value at a future time step.
- Such computational graphs allow us to define recurrent neural networks. We then describe many ways to construct, train, and use recurrent neural networks.

10.1 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:
  - ✦ $s^{(t)} = f(s^{(t-1)}; \theta)$, $s^{(t)}$ 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 τ, the graph can be unfolded by applying the definition τ-1 times.
- E.g. for $\tau = 3$ time steps we get $s^{(3)} = f(s^{(2)}; \theta) = f(f(s^{(1)}; \theta); \theta)$
- 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 $s^{(t)} = f(s^{(t-1)}; \theta)$ and $s^{(3)} = f(f(s^{(1)}; \theta); \theta)$ is illustrated (fig.1) as an unfolded computational graph.

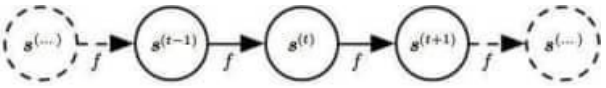

Fig 1. Illustration of an unfolded computational graph.

- 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 θ 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.

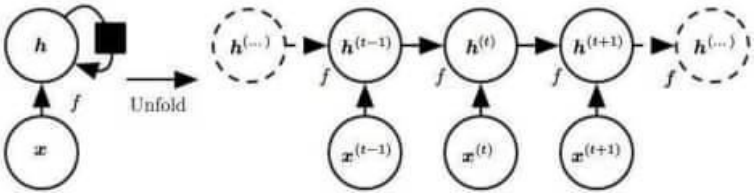

Fig.2 Hidden state h representation.

- 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, it 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, $(x^{(t)}, x^{(t-1)}, x^{(t-2)}, \ldots, x^{(2)}, x^{(1)})$, 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 greater 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, it 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 not be 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 $h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$ 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.

10.2 Recurrent Neural Networks:

RNNs are specialized neural networks designed for sequence data. They process sequences step by step, sharing parameters across time to handle inputs of varying lengths. The key concept to be captured in here are:

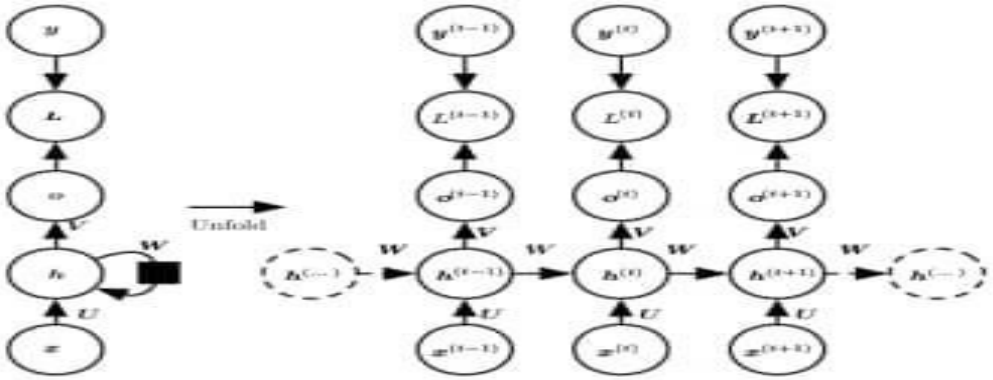

Figure 10.3: The computational graph to compute the training loss of a recurrent network that maps an input sequence of x values to a corresponding sequence of output o values. A loss L measures how far each o is from the corresponding training target y. When using softmax outputs, we assume o is the unnormalized log probabilities. The loss L internally computes $\hat{y} = \text{softmax}(o)$ and compares this to the target y. The RNN has input to hidden connections parametrized by a weight matrix U, hidden-to-hidden recurrent connections parametrized by a weight matrix W, and hidden-to-output connections parametrized by a weight matrix 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.

- It shows **how an RNN processes a sequence of inputs step by step**, updating its internal memory (hidden state) and producing outputs at each time step.

(Left) : Compact RNN Representation

- This is the *folded* view (simplified single-step view):

x → input vector

h → hidden state (memory)

o → output before activation

y → final output after activation (e.g., softmax)

L → loss function

- The RNN has three sets of weights:

U: connects input x to hidden state h

W: connects previous hidden state h(t-1) to current hidden state h(t)

V: connects hidden state h to output o

- The equation for one time step:

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

$$o_t = Vh_t$$

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

$$L_t = loss(\hat{y}_t, y_t)$$

- **(Right) : Unfolded RNN Through Time**

This is the *unrolled* view that shows the RNN across time steps $t-1, t, t+1$.

Each column represents **one time step** of the sequence.

- At each time step:

The **input** $x^{(t)}$is fed into the network.

The **hidden state** $h^{(t)}$depends on both the current input and the previous hidden state:

$$h^{(t)} = f(Ux^{(t)} + Wh^{(t-1)})$$

- The **output** $o^{(t)}$is computed as:

$$o^{(t)} = Vh^{(t)}$$

The **predicted output** $\hat{y}^{(t)} = softmax(o^{(t)})$

The **loss** $L^{(t)}$measures how far the predicted output is from the true target $y^{(t)}$.

When training:

- We **unfold** the network across time steps.
- Compute **loss at each time step**.
- Then apply **Backpropagation Through Time (BPTT)** to update the shared weights U, W, V.
- All time steps share the **same parameters**, which allows the network to **capture temporal dependencies** in sequential data (like text, speech, ECG signals, etc.).

Some examples of important design patterns for recurrent neural networks include the following:

1. Recurrent networks that produce an output at each time step and have recurrent connections between hidden units, illustrated **in figure 10.3**
2. 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**
3. Recurrent networks with recurrent connections between hidden units, that read an entire sequence and then produce a single output, illustrated in **figure 10.5**

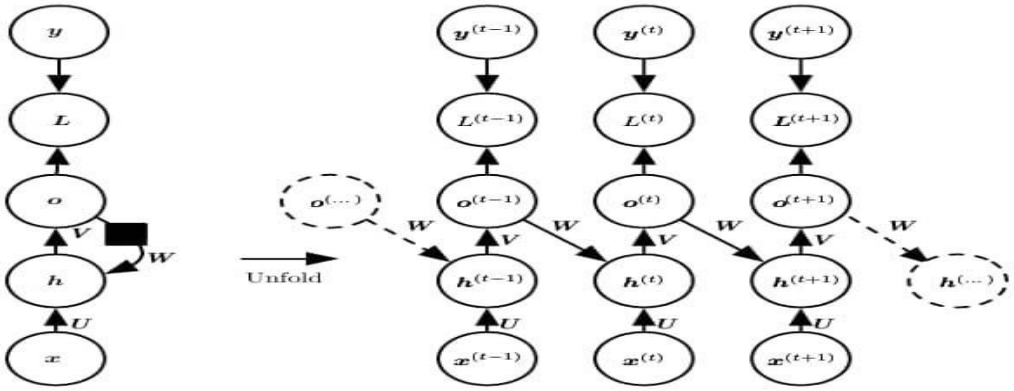

Figure 10.4: An RNN whose only recurrence is the feedback connection from the output to the hidden layer. At each time step t, the input is $\boldsymbol{x}_t$, the hidden layer activations are $\boldsymbol{h}^{(t)}$, the outputs are $\boldsymbol{o}^{(t)}$, the targets are $\boldsymbol{y}^{(t)}$ and the loss is $L^{(t)}$. *(Left)*Circuit diagram. *(Right)*Unfolded computational graph. Such an RNN is less powerful (can express a smaller set of functions) than those in the family represented by figure 10.3. The RNN in figure 10.3 can choose to put any information it wants about the past into its hidden representation $\boldsymbol{h}$ and transmit $\boldsymbol{h}$ to the future. The RNN in this figure is trained to put a specific output value into $\boldsymbol{o}$, and $\boldsymbol{o}$ is the only information it is allowed to send to the future. There are no direct connections from $\boldsymbol{h}$ going forward. The previous $\boldsymbol{h}$ is connected to the present only indirectly, via the predictions it was used to produce. Unless $\boldsymbol{o}$ is very high-dimensional and rich, it will usually lack important information from the past. This makes the RNN in this figure less powerful, but it may be easier to train because each time step can be trained in isolation from the others, allowing greater parallelization during training, as described in section 10.2.1.

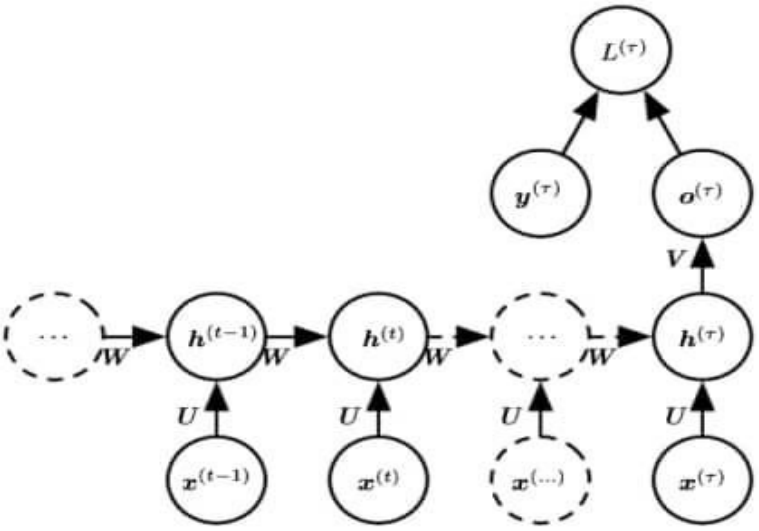

Figure 10.5: Time-unfolded recurrent neural network with a single output at the end of the sequence. Such a network can be used to summarize a sequence and produce a fixed-size representation used as input for further processing. There might be a target right at the end (as depicted here) or the gradient on the output $\boldsymbol{o}^{(t)}$ can be obtained by back-propagating from further downstream modules.

This is a **time-unfolded RNN** (meaning it's shown across time steps), but unlike earlier figures (10.3 and 10.4), here:

The RNN reads an **entire sequence of inputs**

$$x^{(t-1)}, x^{(t)}, \dots, x^{(\tau)}$$

It **updates its hidden state** at each step

$$h^{(t)} = f(Ux^{(t)} + Wh^{(t-1)})$$

It **produces a single output only at the final time step** τ

That final hidden state $h^{(\tau)}$summarizes all the information from the sequence.

Only then do we compute:

$$o^{(\tau)} = Vh^{(\tau)},\ \hat{y}^{(\tau)} = softmax(o^{(\tau)}),\ L^{(\tau)} = loss(\hat{y}^{(\tau)}, y^{(\tau)})$$

10.2.1 Teacher Forcing and Networks with Output Recurrence

- In a normal **Recurrent Neural Network (RNN)**:

Each step depends on the **previous hidden state**.

- h(t) = f(h(t-1), x(t))

means:

- The hidden layer (h) **remembers the past**.So, the network has a kind of **memory**.
- **Example:**

  If you're reading a sentence:

"I love to eat ..."

- The hidden state remembers that this sentence is about "eating,"so ,when predicting the next word, it uses that memory.

Output-recurrent network

- In this version, the network **does not** pass the hidden state forward. It only passes the **output** from the previous step.
- h(t) = f(o(t-1), x(t))
- That means:
- The network uses only what it **predicted** before, not the memory (hidden state) of what it *learned* before.
- Since it has no memory (h(t-1)),it forgets the deeper meaning or details of what happened before.
- It depends only on what came out last time (o(t-1)).
- So, if your last output was slightly wrong, the next prediction will also be based on that wrong output → error grows.
- **What is "Teacher Forcing"?**

During **training**, we help the model by feeding it the **correct output** from the training data instead of its own prediction.

- It involves using the actual target output from the training data (ground truth) as the input for the next time step during training, rather than the network's own prediction at the previous step.
- This helps guide the network more closely to the correct sequence, leading to faster convergence and reducing error accumulation during training.
- The name comes from the analogy of a "**teacher**" giving the correct answer (ground truth) to a student (the network) at each step, so the student stays on track rather than compounding earlier mistakes.
- In a practical application, imagine you have an RNN that needs to predict the next word in a sentence. Without Teacher Forcing, the RNN would use its previous predictions to make the next prediction. However, this can lead to incorrect predictions, as the RNN might get stuck in a loop of incorrect predictions.

- With Teacher Forcing, you would use a pre-trained language model as the teacher. **At each time step, the teacher would provide the correct word that should come next in the sentence, and the RNN would learn from this guidance.** Over time, the RNN would learn to predict the correct words more accurately.

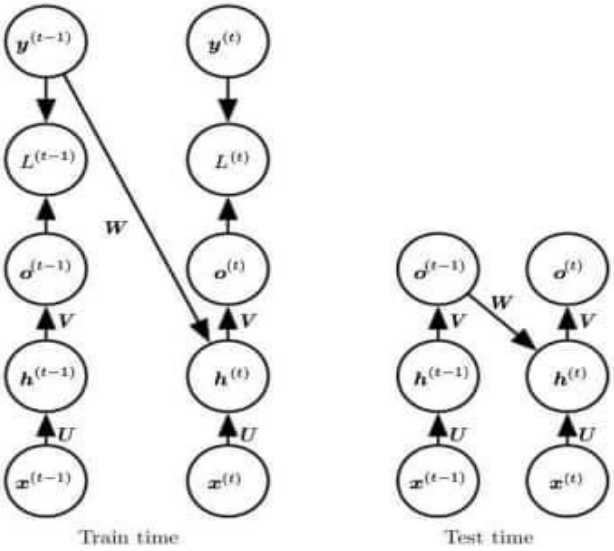

Figure 10.6: Illustration of teacher forcing. Teacher forcing is a training technique that is applicable to RNNs that have connections from their output to their hidden states at the next time step. *(Left)*At train time, we feed the *correct output* $\boldsymbol{y}^{(t)}$ drawn from the train set as input to $\boldsymbol{h}^{(t+1)}$. *(Right)*When the model is deployed, the true output is generally not known. In this case, we approximate the correct output $\boldsymbol{y}^{(t)}$ with the model's output $\boldsymbol{o}^{(t)}$, and feed the output back into the model.

Left side → Train time (Teacher Forcing)

- Look carefully at the arrows:

  y(t-1) → h(t)

  x(t) → h(t)

  h(t) → o(t)

- That big diagonal arrow W from **y(t-1) → h(t)** shows that we are feeding the **true previous output** from the training data into the model.

So, at time t, the model sees:

- The current input x(t)
- The correct previous output y(t-1) and learns to predict the next output o(t).

The model is guided using the **correct previous output** every time (teacher forcing).

Right side → Test time (Deployment)

- Now, during **testing**, there's no teacher and no correct answers yet because we're trying to make predictions.
- So, the model has to **use its own output** from the last step.

instead of **y(t-1) (true output),** we feed back the model's **own predicted output o(t-1).**

- At test time, we don't have the true **y(t-1)** ,we must use the model's own **o(t-1).**

- If the model makes a small mistake at one time step, that wrong output is fed back in, and it can make bigger mistakes later.

This is called **error accumulation** or **exposure bias**.

Teacher forcing helps the model learn faster by using the correct previous outputs during training. But at test time, the model must rely on its own predictions — and that can lead to errors building up over time.

10.2.2 Computing the Gradient in a Recurrent Neural Network

What is gradient computation in an RNN?

- When we train a neural network, we want to adjust weight (U, V, W, etc.) so that the loss (error) becomes smaller.
- To do this, we compute the gradient of the loss with respect to each weight. This tells us *how much* each weight affects the loss ,so we can update them.
- In normal feedforward networks, we use Backpropagation. In RNNs, we use the same idea, but because the same weights are used at every time step, we need to apply backpropagation over time hence BPTT (Backpropagation Through Time).
- Computing the gradient through a recurrent neural network is straightforward. One simply applies the generalized back-propagation algorithm … to the unrolled computational graph."

This means:

- We "unroll" the RNN over all its time steps. So instead of loops, it looks like a long chain of repeated layers (for t=1,2,3,...).
- Then we can apply normal backpropagation through that chain.
- There's no special new formula — just standard backprop but done across time.

BPTT = normal backpropagation applied to the unrolled RNN.

- The nodes of our computational graph include parameters U, V, W, b, and c, as well as x(t), h(t), o(t), and L(t).
- **How we compute gradients (recursively backward)**

They say:

- "For each node N we need to compute $\nabla_n L$ recursively, based on the gradient computed at nodes that follow it in the graph."

This means:

- Start from the **end of the sequence** (the last time step).
- Compute gradients of the loss with respect to outputs first.
- Then move backward step by step to earlier hidden states.

- At each step, you use the gradient from the *next* step (that's recursion).
- So you move backward through time → "backpropagation **through time**."

The start of recursion

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

simply says:

- The gradient of the loss with respect to itself is 1 (the starting point).
- That's just how backprop starts , you begin from the final loss.

in simple words

- When we do **backpropagation** (the process of computing gradients), we always start from the **final output of the network** which is the **loss** L.
- The loss L is a single number that tells how far our prediction is from the correct answer. At the end of training, we want to find how every neuron, weight, and activation contributed to that loss.

  To do this, we "**walk backward**" through the network.
- But before we walk backward, we must start somewhere , we start from the loss itself.
- Backpropagation uses the **chain rule** of derivatives:

$$\frac{dL}{dx} = \frac{dL}{dy} \cdot \frac{dy}{dx}$$

- To use the chain rule, we need to know how the **loss** changes with respect to itself:

$$\frac{dL}{dL} = 1$$

- It's obvious — if you change the loss by 1, the loss changes by 1. That's our **starting point** for backpropagation.
- In an RNN, we have a sequence of losses at different time steps, e.g.:

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

- Each $L^{(t)}$is the loss for time step t.
- When we start backpropagation for the last time step t:

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

- Now we'll move backward from the **output layer** → to the **hidden layer**, and then continue backward through **time**.

Gradient at the output layer

Equation (10.18):

$$(\nabla_{o^{(t)}} L)_i = \hat{y}_i^{(t)} - 1_{i,y^{(t)}}$$

- The gradient at the output $o(t)$is the **difference between the predicted probability** (ŷ(t)) and the **true label** (y(t)).
- This is exactly what happens in a softmax + cross-entropy loss:
  - If the model predicts 0.8 for the correct class, the gradient is small (0.8 - 1 = -0.2).
  - If it predicts 0.1, the gradient is large (0.1 - 1 = -0.9).
- This tells the model how much to adjust its output so the prediction gets closer to the correct label.

Gradient at the last hidden layer (time τ)

Equation (10.19):

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

- The hidden layer $h(\tau)$affects the output $o(\tau)$through weight matrix V.
- So, to get the gradient at the hidden layer, we just multiply the **output gradient** by the transpose of V.
- This tells us how the loss changes with respect to the hidden state.

Moving backward in time

- Now, we go one step earlier: $h(t)$,for $t < \tau$.

The hidden state $h(t)$affects **two things**:

- The output at that time $o(t)$,
- The next hidden state $h(t+1)$.
- So when we compute the gradient for $h(t)$, we must combine **both** of those effects.
- That's what Equation (10.20) says:

$$y\ddot{y}_{h^{(t)}} L = \left(\frac{\partial h^{(t+1)}}{\partial h^{(t)}}\right)^{\top} y\ddot{y}_{h^{(t+1)}} L + \left(\frac{\partial o^{(t)}}{\partial h^{(t)}}\right)^{\top} y\ddot{y}_{o^{(t)}} L$$

- Gradient for hidden state at time t = (gradient flowing from the future hidden state) + (gradient flowing from the current output).
- We are adding up all influences that $h(t)$had on the total loss — both directly (through output) and indirectly (through the next hidden state).

Plugging in the actual formulas (Eq. 10.21)

- For tanh activation and the weight matrices W(hidden-to-hidden) and V(hidden-to-output), this becomes:

$$y\ddot{y}_{h^{(t)}} L = W^{\top} y\ddot{y}_{h^{(t+1)}} L \, \boldsymbol{diag}\left(1 - \left(h^{(t+1)}\right)^{\leftrightarrow \ddot{y}}\right) + V^{\top} y\ddot{y}_{o^{(t)}} L$$

$$V^{\top} y\ddot{y}_{o^{(t)}} L$$

- This is the **direct effect** of the current hidden state on the output error — the same as before at the last step.

$$W^{\top} y\ddot{y}_{h^{(t+1)}} L$$

- This term brings in the **error coming from the next time step**, because $h(t+1)$depends on $h(t)$.

So this is how gradients **flow backward through time**.

diag(1 - h²)

- This is the **derivative of the tanh activation function**.
  It tells how sensitive the next hidden state is to the current one.
  The smaller this term is, the more the gradient "shrinks" → that's where **vanishing gradients** come from.

RNN is predicting something step by step (like next words or next numbers).

At each time step:

- You calculate an error at the output (how wrong you were).
- That error is **sent backward**:
  - Directly to the hidden layer of that step,
  - And also, further back in time through the hidden-to-hidden connections.

Each hidden layer therefore gets two signals:

- How much it caused the output to be wrong (local error),
- How much it influenced future outputs (through the chain of hidden states).

By combining both, we know how to adjust weights for all steps.

10.2.3 Recurrent Networks as Directed Graphical Models

What is a "Directed Graphical Model"?

A directed graphical model is just a way of showing how variables depend on each other using arrows (→).
Each arrow shows that one variable provides information used to predict another.
"The losses L(t) were cross-entropies between training targets y(t) and outputs o(t)..."
It's saying:
At each time step t, we compare the predicted output $o(t)$with the true output $y(t)$.
The loss function measures how close they are.
Usually, this is cross-entropy loss, because RNN outputs are treated as probabilities.

RNN predicts a probability for each possible next value — and we use cross-entropy to measure how "wrong" that probability is.

- "As with a feedforward network, it is possible to use almost any loss…
  we usually use the cross-entropy associated with that distribution."

That means:

- Depending on the task, you can use different loss functions:
  - For **classification (predicting next word, character, etc.)** → cross-entropy.
  - For **regression (predicting next numeric value)** → mean squared error (MSE).

- Cross-entropy works best when the output represents a **probability distribution** (like next word prediction).
- So, the loss function depends on **what type of output** the RNN produces.

Predictive log-likelihood training

- "When we use a predictive log-likelihood training objective… we train the RNN to estimate the conditional distribution of the next sequence element y(t) given the past inputs."

This just means:

- The RNN is learning **to predict the next item** in a sequence,
- Based on everything it has seen so far (all past inputs).
- The **log-likelihood** is a fancy way of saying we want to make the **correct next prediction as likely as possible**.
- We maximize:

$$\log p\ (y^{(t)}, |, x^{(1)} \ldots x^{(t)})$$

- That's Equation (10.29).

"Given all the inputs up to now, predict the next output y(t)."

When the RNN also uses previous outputs

- "If the model includes connections from the output at one time step to the next time step…"
- Then the RNN also uses previous **outputs** as inputs for predicting the next step.
  That's Equation (10.30):

$$\log p\ (y^{(t)}, |, x^{(1)}, \ldots, x^{(t)}, y^{(1)} \ldots y^{(t-1)})$$

- "Predict the next output y(t) using all inputs up to now and all previous outputs."
- This is the **teacher forcing setup** you saw before:
- During training, you feed in the **true y(t-1)** (correct output).
- During testing, you feed in the **predicted output** o(t-1).

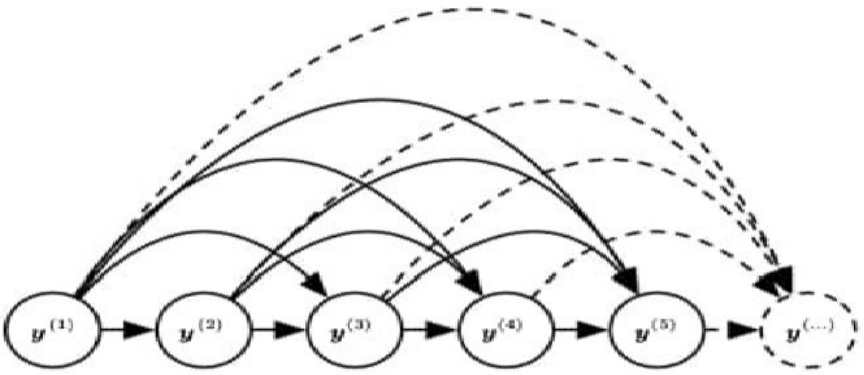

Figure 10.7: Fully connected graphical model for a sequence $y^{(1)}, y^{(2)}, \dots, y^{(t)}, \dots$: every past observation $y^{(i)}$ may influence the conditional distribution of some $y^{(t)}$ (for $t > i$), given the previous values. Parametrizing the graphical model directly according to this graph (as in equation 10.6) might be very inefficient, with an ever growing number of inputs and parameters for each element of the sequence. RNNs obtain the same full connectivity but efficient parametrization, as illustrated in figure 10.8.

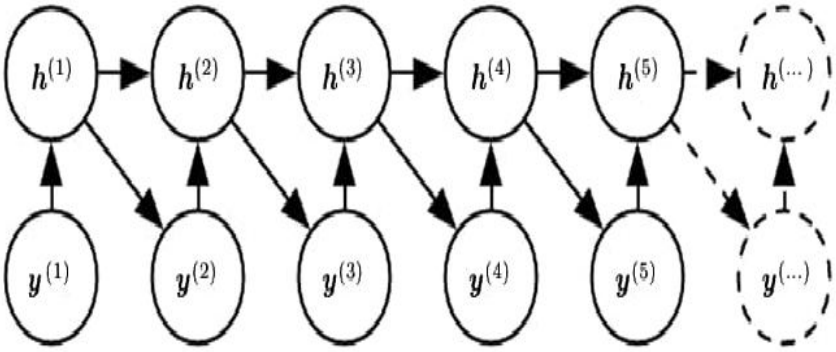

Figure 10.8: Introducing the state variable in the graphical model of the RNN, even though it is a deterministic function of its inputs, helps to see how we can obtain a very efficient parametrization, based on equation 10.5. Every stage in the sequence (for $\boldsymbol{h}^{(t)}$ and $\boldsymbol{y}^{(t)}$) involves the same structure (the same number of inputs for each node) and can share the same parameters with the other stages.

10.2.4 Modeling Sequences Conditioned on Context with RNNs

RNNs takes a sequence of vectors x(t) fort = 1, . . . , τ as input.

Another option is to take only a single vector x as input.

When x is a fixed-size vector, we can simply make it an extra input of the RNN that generates the y sequence.

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.

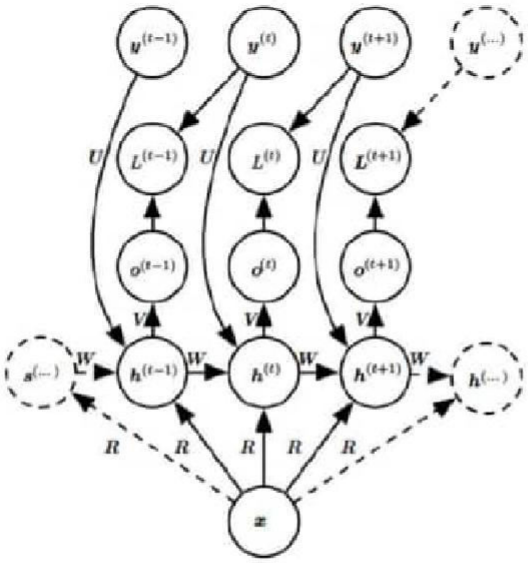

Figure 10.9: An RNN that maps a fixed-length vector $\boldsymbol{x}$ into a distribution over sequences $\boldsymbol{Y}$. This RNN is appropriate for tasks such as image captioning, where a single image is used as input to a model that then produces a sequence of words describing the image. Each element $\boldsymbol{y}^{(t)}$ of the observed output sequence serves both as input (for the current time step) and, during training, as target (for the previous time step).

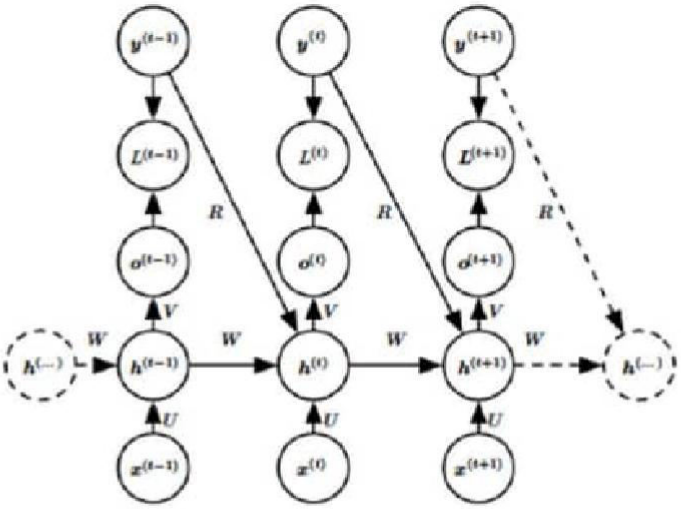

Figure 10.10: A conditional recurrent neural network mapping a variable-length sequence of $\boldsymbol{x}$ values into a distribution over sequences of $\boldsymbol{y}$ values of the same length. Compared to figure 10.3, this RNN contains connections from the previous output to the current state. These connections allow this RNN to model an arbitrary distribution over sequences of $\boldsymbol{y}$ given sequences of $\boldsymbol{x}$ of the same length. The RNN of figure 10.3 is only able to represent distributions in which the $\boldsymbol{y}$ values are conditionally independent from each other given the $\boldsymbol{x}$ values.

10.3 Bidirectional RNNs:

- All the recurrent networks we've discussed so far have a "causal" structure, meaning the state at time t captures information only from the past inputs $(x^{(1)},\ldots,x^{(t-1)})$ (and the current input $x^{(t)}$.
- In some cases, these models also incorporate information from past y values into the current state, provided those y values are accessible.

- In some tasks, the interpretation of the current input depends on the entire input sequence, not just past or present information.
- For instance, in **speech recognition**, determining the correct phoneme for the current sound might rely on the next few phonemes due to co-articulation or even on future words because of linguistic dependencies.
- If multiple interpretations of a word are acoustically plausible, context from both the future and the past may be needed to resolve ambiguity.
- This concept applies to other sequence-to-sequence tasks as well, such as **handwriting recognition**.

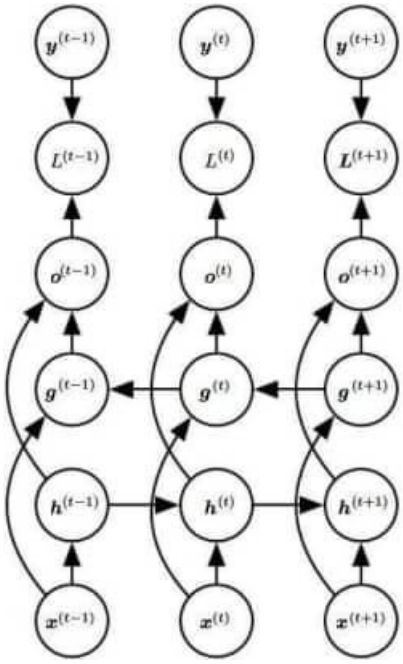

Fig 3. Computation of a typical bidirectional recurrent neural network

- Bidirectional Recurrent Neural Networks (Bidirectional RNNs) were introduced by Schuster and Paliwal in 1997 to address the need for utilizing both past and future context when processing sequential data. These networks process input sequences in both forward and backward directions, enabling the model to capture information from the entire sequence.
- Bidirectional RNNs have proven highly effective in tasks requiring this capability, such as:
- Handwriting recognition (Graves et al., 2008; Graves and Schmidhuber, 2009).
- Speech recognition (Graves and Schmidhuber, 2005; Graves et al., 2013).
- Bioinformatics (Baldi et al., 1999).
- Their ability to integrate context from both directions makes them powerful for sequence-to-sequence tasks.
- Bidirectional RNNs combine two RNNs: one processes the sequence forward in time (from the start to the end), and the other processes it backward in time (from the end to the start). In this architecture:

- $h^{(t)}$: Represents the state of the forward-moving RNN.
- $g^{(t)}$: Represents the state of the backward-moving RNN.
- This setup allows the output units $o^{(t)}$ to create a representation that leverages information from both the past and the future.
- Unlike feedforward networks, convolutional networks, or standard RNNs with fixed-size look-ahead buffers, bidirectional RNNs do not require specifying a fixed-size window around t. Instead, they naturally focus on the input values near t, making them highly effective for tasks that benefit from bidirectional context.
- The concept of bidirectional RNNs can be extended to **2-dimensional inputs**, such as images, by introducing **four RNNs** that process the data in different directions: up, down, left, and right. At each position (i,j) in a 2D grid, an output Oi,j can compute a representation that captures mostly local information while also incorporating long-range dependencies, provided the RNN effectively learns to carry such information.
- Compared to convolutional networks:
- RNNs applied to images are generally **more computationally expensive**.
- They enable **long-range lateral interactions** between features within the same feature map, which convolutional networks cannot achieve as easily.
- The forward propagation equations for these RNNs show they include:
- A **convolution operation** to compute bottom-up input for each layer.
- **Recurrent propagation** across the feature map, incorporating lateral interactions.
- This approach allows RNNs to leverage both local and global context in image data, providing a more comprehensive understanding of spatial relationships (Visin et al., 2015; Kalchbrenner et al., 2015).

10.4 Encoder-Decoder Sequence-to-Sequence Architectures

What RNNs can already do

- They can connect sequences and vectors in three main ways:

1. Sequence → Fixed-size Vector

2. Fixed-size Vector → Sequence

Image captioning

- Input = image features (from CNN)
- Output = caption words (a sequence)
- The RNN starts with the image vector and *generates words one by one*.

3. Sequence → Sequence (same length)

The problem with all these

- All three of the above cases assume that:
- Input and output sequences are either of fixed size or the same length.
- But in many real-world tasks, the input and output sequences are not the same length.

Question Answering	Question sentence	Short answer phrase
Speech Recognition	Long audio waveform (many frames)	Short text like "Hello world"

we need a way to:

Read the entire input sequence,

store its meaning,

then generate an output sequence of any length.

That's exactly what Encoder–Decoder (Seq2Seq) does.

Seq2Seq combines two RNNs:

- **Encoder: reads and compresses the input sequence into a *context vector* (C).**
- **Decoder: takes that context and *generates* the output sequence.**

Input and output can now have different lengths.

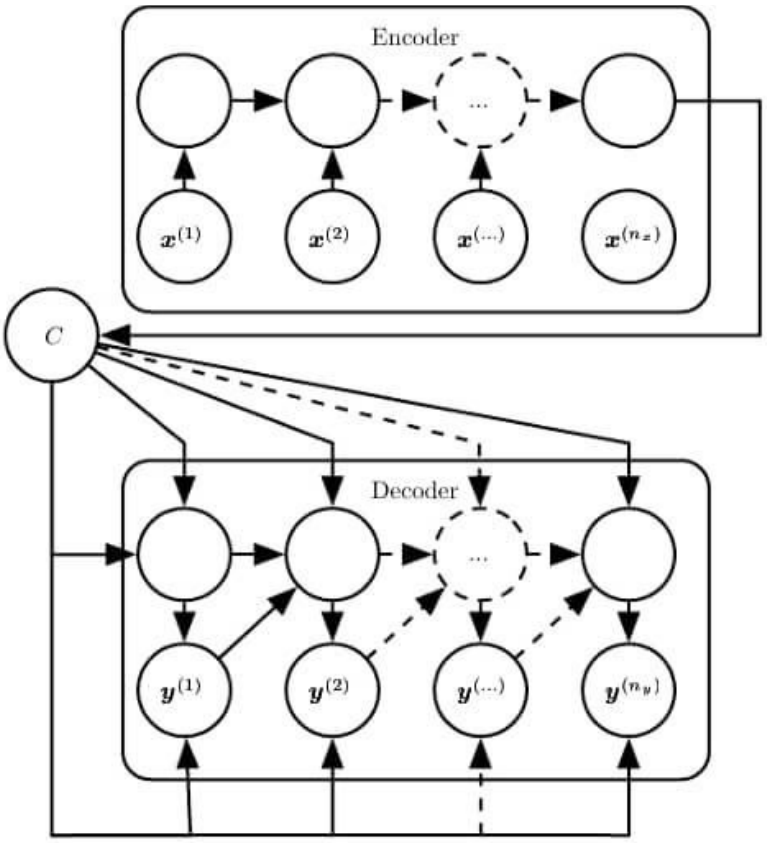

Figure 10.12: Example of an encoder-decoder or sequence-to-sequence RNN architecture, for learning to generate an output sequence $(\mathbf{y}^{(1)}, \ldots, \mathbf{y}^{(n_y)})$ given an input sequence $(\mathbf{x}^{(1)}, \mathbf{x}^{(2)}, \ldots, \mathbf{x}^{(n_x)})$. It is composed of an encoder RNN that reads the input sequence and a decoder RNN that generates the output sequence (or computes the probability of a given output sequence). The final hidden state of the encoder RNN is used to compute a generally fixed-size context variable C which represents a semantic summary of the input sequence and is given as input to the decoder RNN.

Encoder

- The top box ("Encoder") receives input sequence values $x^{(1)}, x^{(2)}, \ldots, x^{(nx)}$.
- The arrows show how each step passes information forward through hidden states (RNN's memory).
- The final hidden state (rightmost circle) becomes the context vector C.

Context C

- This single vector summarizes the entire input sequence.
- It's a compressed representation that contains the "meaning" or "essence" of the input.

Decoder

- The bottom box ("Decoder") starts with the context C.
- It uses C to generate the output sequence step-by-step.
- Each output $y^{(t)}$also depends on the previous ones (just like a normal RNN).

- So, Cacts as a bridge between the input and output sequences.

How it's trained

- Training tries to maximize:
- $P\left(y^{(1)}, y^{(2)}, \ldots, y^{(n_y)} \mid x^{(1)}, x^{(2)} \ldots x^{(n_x)}\right)$

  This means:

- Given the input sequence x, the model should assign high probability to the correct output sequence y.
- Both encoder and decoder RNNs are trained together, end-to-end.
- Limitation: the bottleneck problem
- "One clear limitation of this architecture is when the context C has a dimension that is too small to properly summarize a long sequence."

This means:

- If the input is long (e.g., a long sentence), a single fixed-size C can't hold all the information.
- Important details get lost ,the decoder forgets early parts of the input.
- This problem was discovered by Bahdanau et al. (2015).
- Solution → Attention mechanism
- Bahdanau's improvement:
- Instead of a single fixed vector C, make it variable-length.
- Let the decoder "look back" at the encoder's hidden states as it generates each output.

This is the Attention mechanism it lets the decoder selectively focus on different parts of the input sequence

10.5 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 deep 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 of a 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 relate 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:

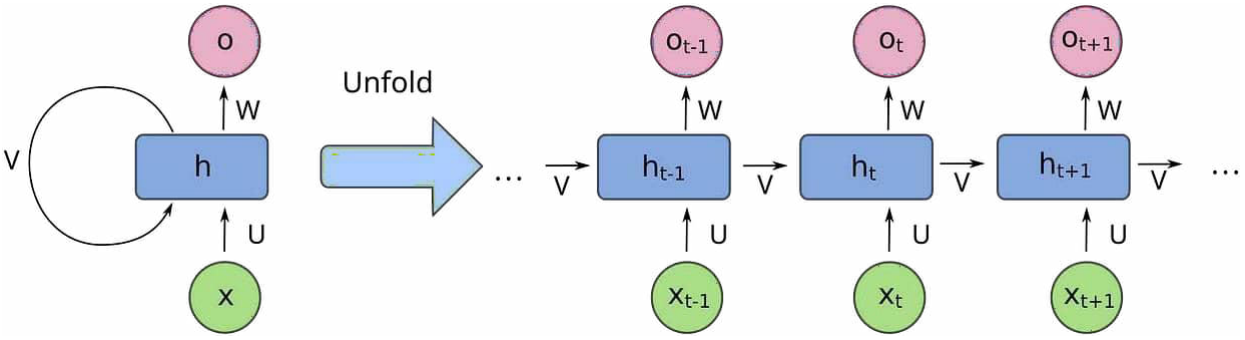

Fig 4. Architecture of a deep neural network.

In the above figure 4, the deep RNN has three hidden layers. The arrows indicate the flow of information 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.

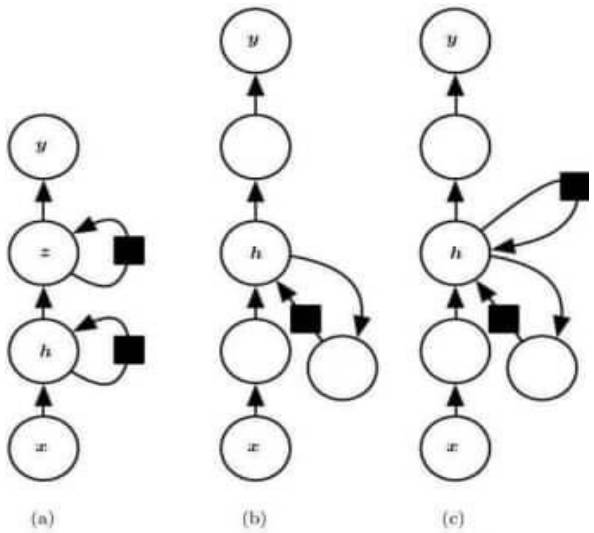

Fig 5. A recurrent neural network can be made deep in many ways.

(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 computation (z) between them, creating a deeper network.
- The hierarchical nature of hidden states aligns with the idea that the hidden recurrent state can be broken into groups.
- This hierarchical arrangement introduces multiple processing steps (hidden layers), allowing the RNN to learn more abstract representations over time.

(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 modeling 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.

10.6Recursive Neural Networks:

Recursive neural networks2 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.

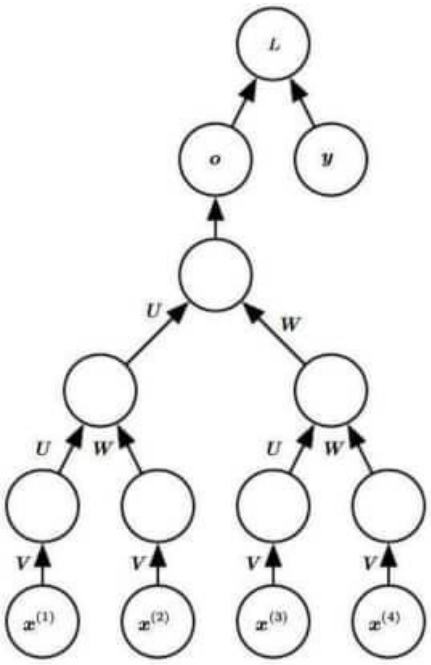

Figure 5. A typical graph of recursive neural network.

- 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.

- A notable advantage of recursive networks over recurrent networks is their ability to significantly reduce the depth (measured by the number of compositions of nonlinear operations) from τ to O (log τ) for a sequence of length **τ**, which can help address long-term dependencies. However, determining the optimal tree structure remains an open question. One approach is to use a fixed tree structure, such as a balanced binary tree, that does not depend on the data.
- 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 for 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 the tree.
- The computations performed at each node are not limited to the conventional artificial neuron operations (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 effectively model relationships between concepts (Weston et al., 2010; Bordes et al., 2012) when the concepts are represented as continuous vector embeddings.

10.10The Long Short-Term Memory and Other Gated RNNs:

The introduction of **self-loops** to create paths that allow gradients to flow for extended durations is a key innovation of the original **Long Short-Term Memory (LSTM)** model by Hochreiter and Schmidhuber (1997). A significant improvement was later made by Gers et al. (2000), where the weight of the self-loop is conditioned on the **context** rather than being fixed. By using a **gated mechanism** (controlled by another hidden unit), the time scale of integration can be dynamically adjusted. This means that even with fixed parameters, an LSTM can adapt the integration time scale based on the input sequence, as the time constants are generated by the model itself.

The LSTM has achieved remarkable success across a variety of applications, including:

- **Unconstrained handwriting recognition** (Graves et al., 2009),
- **Speech recognition** (Graves et al., 2013; Graves and Jaitly, 2014),
- **Handwriting generation** (Graves, 2013),
- **Machine translation** (Sutskever et al., 2014),
- **Image captioning** (Kiros et al., 2014b; Vinyals et al., 2014b; Xu et al., 2015), and

- **Parsing** (Vinyals et al., 2014a).
- This dynamic gating mechanism enables LSTMs to adapt effectively to different input patterns, making them highly versatile and powerful in sequence modeling tasks.
- LSTM (Long Short-Term Memory) is a recurrent neural network (RNN) architecture widely used in Deep Learning. It excels at capturing long-term dependencies, making it ideal for sequence prediction tasks.
- Unlike traditional neural networks, LSTM incorporates feedback connections, allowing it to process entire sequences of data, not just individual data points. This makes it highly effective in understanding and predicting patterns in sequential data like time series, text, and speech.
- LSTM has become a powerful tool in artificial intelligence and deep learning, enabling breakthroughs in various fields by uncovering valuable insights from sequential data.

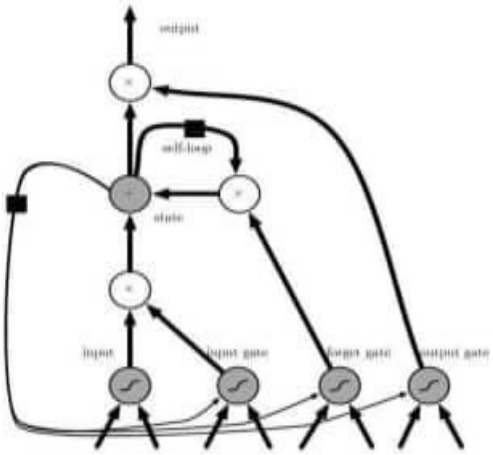

Fig 6. Architecture of LSTM.

- The figure 5 depicts the block diagram of the LSTM recurrent network "cell." Cells are connected recurrently to each other, replacing the usual hidden units of ordinary recurrent networks.
- An input feature is computed with a regular artificial neuron unit. Its value can be accumulated into the state if the sigmoidal input gate allows it. The state unit has a linear self-loop whose weight is controlled by the forget gate.
- The output of the cell can be shut off by the output gate. All the gating units have a sigmoid nonlinearity, while the input unit can have any squashing nonlinearity.
- The state unit can also be used as an extra input to the gating units. The black square indicates a delay of a single time step.

  Just like a simple RNN, an LSTM also has a hidden state where H(t-1) represents the hidden state of the previous timestamp and Ht is the hidden state of the current timestamp. In addition to that, LSTM also has a cell state represented by C(t-1) and C(t) for the previous and current timestamps, respectively.

- Example of LTSM Working: Let's take an example to understand how LSTM works. Here we have two sentences separated by a full stop. The first sentence is "Bob is a nice person," and the second sentence is "Dan, on the Other hand, is evil". It is very clear, in the first sentence, we are talking about Bob, and as soon as we encounter the full stop(.), we started talking about Dan.
- As we move from the first sentence to the second sentence, our network should realize that we are no more talking about Bob. Now our subject is Dan. Here, the Forget gate of the network allows it to forget about it. Let's understand the roles played by these gates as per the LSTM architecture.
- Forget Gate: In a cell of the LSTM neural network, the first step is to decide whether we should keep the information from the previous time step or forget it. Here is the equation for forget gate.

$$f_i^{(t)} = \sigma\left(b_i^f + \sum_j U_{i,j}^f x_j^{(t)} + \sum_j W_{i,j}^f h_j^{(t-1)}\right)$$

- forget gate unit $f_i^{(t)}$ (for time step t and cell i), that sets this weight to a value between 0 and 1 via a sigmoid unit.
- where $x^{(t)}$ is the current input vector and $h^{(t)}$ is the current hidden layer vector, containing the outputs of all the LSTM cells, and b^f, U^f, W^f are respectively biases, input weights and recurrent weights for the forget gates.
- The LSTM cell internal state is thus updated as follows, but with a conditional self-loop weight $f_i^{(t)}$:

$$s_i^{(t)} = f_i^{(t)} s_i^{(t-1)} + g_i^{(t)} \sigma\left(b_i + \sum_j U_{i,j} x_j^{(t)} + \sum_j W_{i,j} h_j^{(t-1)}\right)$$

- where b, U and W respectively denote the biases, input weights and recurrent weights into the LSTM cell. The external input gate unit $g_i^{(t)}$ is computed similarly to the forget gate (with a sigmoid unit to obtain a gating value between 0 and 1), but with its own parameters:

$$g_i^{(t)} = \sigma\left(b_i^g + \sum_j U_{i,j}^g x_j^{(t)} + \sum_j W_{i,j}^g h_j^{(t-1)}\right)$$

The output $h_i^{(t)}$ of the LSTM cell can also be shut off, via the output gate $q_i^{(t)}$, which also uses a sigmoid unit for gating:

$$h_i^{(t)} = \tanh\left(s_i^{(t)}\right) q_i^{(t)}$$

$$q_i^{(t)} = \sigma\left(b_i^o + \sum_j U_{i,j}^o x_j^{(t)} + \sum_j W_{i,j}^o h_j^{(t-1)}\right)$$

- which has parameters b^o, U^o, W^o for its biases, input weights and recurrent weights, respectively. Among the variants, one can choose to use the cell state $s_i^{(t)}$ as an extra input (with its weight) into the three gates of the i-th unit, as shown in figure 6.

Other Gated RNNs:

- Which pieces of the LSTM architecture are actually necessary? What other successful architectures could be designed that allow the network to dynamically control the time scale and forgetting behaviour of different units?
- Some answers to these questions are given with the recent work on gated RNNs, whose units are also known as gated recurrent units.
- The main difference with the LSTM is that a single gating unit simultaneously controls the forgetting factor and the decision to update the state unit. The update equations are the following:

$$h_i^{(t)} = u_i^{(t-1)} h_i^{(t-1)} + (1 - u_i^{(t-1)})\sigma\left(b_i + \sum_j U_{i,j} x_j^{(t-1)} + \sum_j W_{i,j} r_j^{(t-1)} h_j^{(t-1)}\right)$$

- where u stands for “update” gate and r for “reset” gate. Their value is defined as usual:

$$u_i^{(t)} = \sigma\left(b_i^u + \sum_j U_{i,j}^u x_j^{(t)} + \sum_j W_{i,j}^u h_j^{(t)}\right)$$

$$r_i^{(t)} = \sigma\left(b_i^r + \sum_j U_{i,j}^r x_j^{(t)} + \sum_j W_{i,j}^r h_j^{(t)}\right)$$

- The reset and update gates can selectively "ignore" specific parts of the state vector. The **update gates** function as conditional leaky integrators, allowing any dimension to be linearly gated.
- At one extreme of the sigmoid function, they can copy the current value directly, while at the other extreme, they can completely discard it and replace it with the new **target state** value, which the leaky integrator aims to reach.

- Meanwhile, the **reset gates** determine which parts of the state contribute to the computation of the next target state. This introduces an additional **nonlinear effect** that influences the relationship between the previous state and the future state.

- Many more variants around this theme can be designed. For example, the reset gate (or forget gate) output could be shared across multiple hidden units. Alternately, the product of a global gate (covering a whole group of units, such as an entire layer) and a local gate (per unit) could be used to combine global control and local control.

○ Optimization for Long-Term Dependencies:

Clipping gradients is a popular technique in optimization, especially for training recurrent neural networks (RNNs) and their variants (like LSTMs and GRUs), to address the issue of exploding gradients that arise when dealing with long-term dependencies.

1. Problem of Exploding Gradients

- When training deep networks or RNNs over long sequences, the gradients of the loss function with respect to the model parameters can sometimes grow exponentially during backpropagation through time (BPTT).
- This occurs when the product of gradients of weight matrices over multiple time steps becomes excessively large, leading to unstable updates in gradient-based optimization.
- Exploding gradients result in:
  - Overshooting the optimal parameters,
  - Poor convergence, or
  - NaNs (numerical instability).

2. What is Gradient Clipping?

Gradient clipping is a method to stabilize training by preventing the gradients from growing too large. The idea is simple:

- If the norm of the gradient exceeds a predefined threshold, the gradient is rescaled so that its norm matches the threshold.
- This ensures that gradient updates remain controlled and do not explode.

- In tasks with long-term dependencies, the gradients must propagate through many time steps. While vanishing gradients hinder learning of long-term dependencies, exploding gradients can destabilize the network. Gradient clipping directly addresses this instability by:

  - Keeping gradient updates within a controlled range,
  - Allowing the model to still learn meaningful information across long time sequences, and
  - Preventing catastrophic updates that cause the training to diverge.

Large-Scale Deep Learning:

- Large-Scale Deep Learning refers to the practice of training deep neural networks with a massive amount of data and/or large model architectures. It involves addressing challenges related to computational resources, optimization, and scalability, while ensuring that models can learn from extensive datasets in a reasonable time frame.
- Here's a deeper look into the main aspects of large-scale deep learning:

1. Key Challenges in Large-Scale Deep Learning:

- a. Data Size and Quality ○ Big Data: Large-scale deep learning often involves datasets that are too large to fit into memory or process on a single machine. For instance, training on millions or billions of labelled data points.
  - Data Preprocessing: Handling, cleaning, augmenting, and managing large datasets require specialized tools and distributed computing frameworks.
- b. Model Size and Complexity ○ Modern deep learning models, particularly in areas like computer vision, speech recognition, and natural language processing (NLP), can have millions or billions of parameters.
  - These models can be highly memory-intensive and computationally expensive to train.
- c. Computational Resources ○ Parallelism: Training large-scale models requires distributing computations across multiple machines or GPUs, a process known as distributed training.
  - GPUs and TPUs: GPUs (Graphics Processing Units) and TPUs (Tensor Processing Units) are specialized hardware designed to accelerate deep learning operations, especially in large-scale training tasks.
- d. Optimization and Convergence ○ Training large-scale models on large datasets can lead to challenges with optimization, such as local minima, exploding/vanishing gradients, and slow convergence.

2. Techniques for Scaling Deep Learning

- Distributed Training ○ Data Parallelism: The dataset is divided into smaller chunks, and each chunk is processed by a different machine or GPU. The gradients from all machines are then aggregated to update the model weights.
  - ○ Model Parallelism: The model is split across different devices, with each device handling a subset of the model's parameters.
- Gradient Clipping ○ As previously discussed, gradient clipping is used to prevent exploding gradients when training large models, particularly on long sequences (e.g., RNNs and LSTMs).

3. Scaling Up with Hardware

- GPUs and TPUs ○ GPUs are widely used for deep learning because they are designed to handle the matrix operations common in neural networks more efficiently than CPUs.
  - ○ TPUs (Tensor Processing Units) are specialized hardware accelerators built by Google specifically for deep learning tasks. They are optimized for training and inference of large models in TensorFlow.
  - ○ Large-scale training often involves using clusters of GPUs or TPUs, allowing simultaneous computations over multiple devices and speeding up training time.
- Cloud Computing and Distributed Systems ○ Cloud providers like Amazon Web Services (AWS), Google Cloud, and Microsoft Azure offer scalable infrastructures for training large models. This allows deep learning practitioners to rent vast amounts of computational power on demand.
  - ○ Cloud platforms often provide auto-scaling features, which automatically scale resources up or down depending on the computational requirements of the task.

Speech recognition: Speech recognition is the process of converting spoken language into written text. It involves understanding, interpreting, and transcribing spoken words into a format that can be processed by computers. It is a crucial component of various applications like voice assistants (e.g., Siri, Google Assistant), transcription services, and voice-controlled devices.

- Key Features of Speech Recognition:
  - Accuracy and Speed: They can process speech in real-time or near real-time, providing quick responses to user inputs.
  - Multi-Language Support: Support for multiple languages and dialects, allowing users from different linguistic backgrounds to interact with technology in their native language.

- Background Noise Handling: This feature is crucial for voice-activated systems used in public or outdoor settings.

- **Challenges in Speech Recognition**
  - Variability in Speech
    - Accents and Dialects: Different accents, dialects, and regional variations in pronunciation can make speech recognition more challenging.
    - Background Noise: Speech recognition systems can struggle when background noise is present, such as in crowded environments or when multiple speakers are involved.
    - Speech Rate: Variations in the speed at which people speak can impact recognition accuracy. Fast speech, slurred speech, or pauses between words can be problematic.
  - Ambiguity in Speech
    - Homophones: Words that sound the same but have different meanings (e.g., "see" and "sea") can confuse the model.
    - Contextual Meaning: Words can have different meanings depending on the context (e.g., "bank" as a financial institution or the side of a river).
    - Out-of-Vocabulary Words: If a word has not been seen in the training data, the system may struggle to recognize it correctly.
  - Real-Time Processing
    - For applications like voice assistants and real-time transcription, the model must recognize speech instantaneously with minimal delay.
    - Efficient latency management is crucial for keeping the user experience smooth and responsive.

- **Natural Language Processing (NLP):** Natural Language Processing (NLP) is a branch of artificial intelligence (AI) focused on enabling computers to understand, interpret, and generate human language. NLP combines computational linguistics, machine learning, and deep learning to process and analyze large amounts of natural language data, such as text or speech.

- Key Tasks in Natural Language Processing:
- NLP encompasses various tasks that are fundamental to understanding and processing human language. Some of these tasks include:
- Tokenization: The process of breaking down text into smaller units, such as words, sentences, or sub word units. For example, the sentence "Natural language processing is fun!" can be tokenized into the words "Natural," "language," "processing," "is," "fun," and "!".

- Part-of-Speech (POS) Tagging: This task involves identifying the grammatical components of each token in a sentence. It labels words as nouns, verbs, adjectives, etc., to understand their roles within the sentence.
- Named Entity Recognition (NER): NER involves identifying and classifying entities (such as names of people, organizations, locations, dates, etc.) in the text.
- Sentiment Analysis: This technique determines the sentiment or emotional tone of a piece of text. It helps classify text as positive, negative, or neutral, and can be used to analyze social media, product reviews, etc.
- Machine Translation: The process of automatically translating text from one language to another. Google Translate and Deeply are popular examples of machine translation systems.
- Text Summarization: This involves condensing a piece of text into a shorter summary while preserving its essential meaning. There are two types of summarizations:
  - Extractive Summarization: Selects important sentences from the original text.
  - Abstractive Summarization: Generates new sentences to convey the core ideas of the original text.
- Speech Recognition: Converting spoken language into written text. This task involves recognizing words in spoken language and transcribing them into text format.
- **Challenges in NLP:**
- Ambiguity: Natural language is inherently ambiguous. Words can have multiple meanings depending on the context (e.g., "bank" can refer to a financial institution or the side of a river). Resolving this ambiguity is a key challenge for NLP systems.
- Sarcasm and Irony: NLP models often struggle with understanding sarcasm, irony, and other figurative language because they require nuanced understanding of context and tone.
- Multilingual NLP: NLP systems trained in one language may not perform well in others. Building multilingual systems that can handle many languages is a complex challenge, but there has been progress with multilingual models like mBERT (Multilingual BERT).
- Low-Resource Languages: Many languages still lack sufficient training data or resources, making it harder to build high-performing models for these languages.
- Bias and Fairness: NLP models trained on large corpora may inherit biases present in the data, leading to discriminatory or unfair outcomes in applications like hiring, law enforcement, or healthcare.