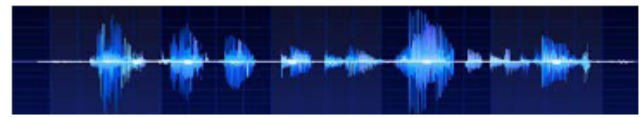
Deep Learning

RECURRENT NEURAL NETWORKS (RNN)

Introduction to RNN

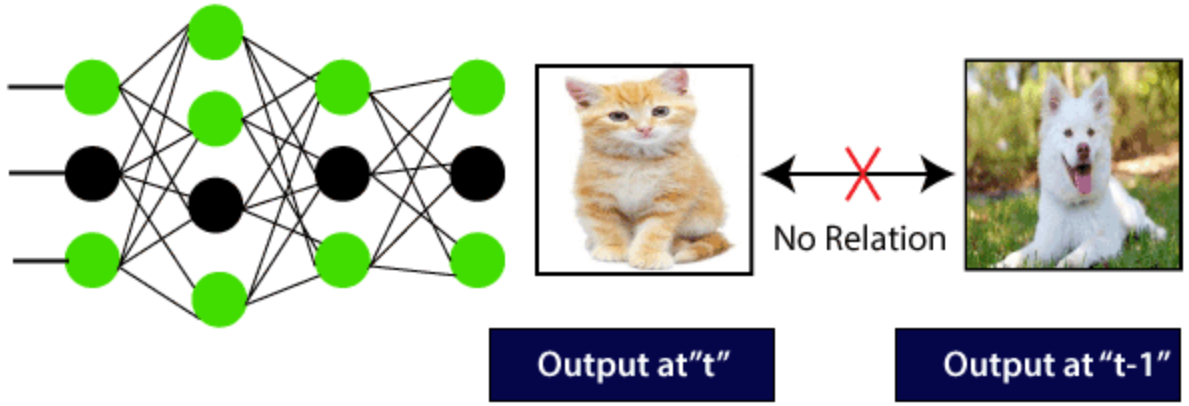
- A **Recurrent Neural Network (RNN)** is a type of neural network designed for processing sequences of data by maintaining a form of memory (or context) over time.
- It is especially well-suited for tasks involving sequential data, such as time series prediction, natural language processing, and speech recognition.

Examples of Sequence data	Input Data	Output
Speech Recognition		*This is RNN*
Machine Translation	Hello, I am Pankaj.	Hallo, ich bin Pankaj. हेलो, मैं पंकज हूं।
Language Modeling	Recurrent neural ? based ? model	**network** **language**
Named Entity Recognition	Pankaj lives in Munich	**Pankaj** lives in **Munich** ***person*** ***location***
Sentiment Classification	There is nothing to like in this movie.	★☆☆☆☆
Video Activity Analysis		Punching

Why do we need *Sequential Modeling*?

Classification with CNN

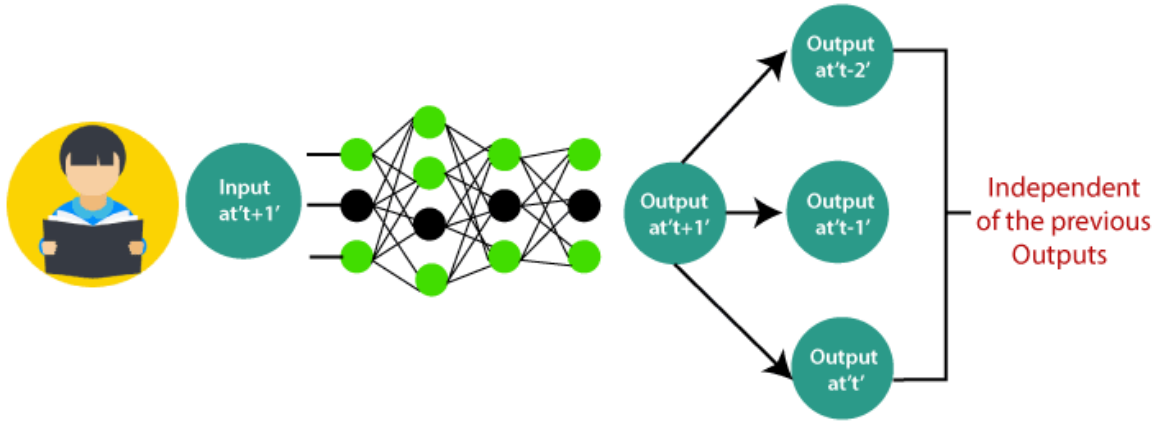
- Consider an **image classification** use-case where we have **trained** the neural network to **classify images** of some **animals**.
- Let's feed an image of a cat or a dog; the network provides an output with the **corresponding label** to the picture of a cat or a dog.



- Here, the first output being a cat will not influence the previous output, which is a dog. This means that output at a time 't' is autonomous of output at the time 't-1

Sequential Information

- Consider the scenario where we will require the use of the last **obtained** output:
- Ex: concept is the same as reading a **book**. With every page we move forward into, we need the **understanding** of previous pages to make complete sense of the **information** in most of the **cases**.



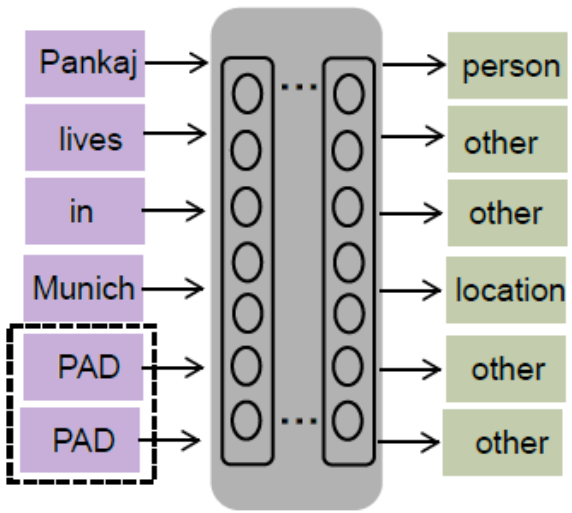
- With the help of the feed-forward network, the new output at the **time 't+1**? has no relation with outputs at either time t, **t-1, t-2**.
- So, the feed-forward network cannot be used when predicting a word in a sentence as it will have no absolute relation with the previous set of words.

Inputs, Outputs can be different lengths in different examples

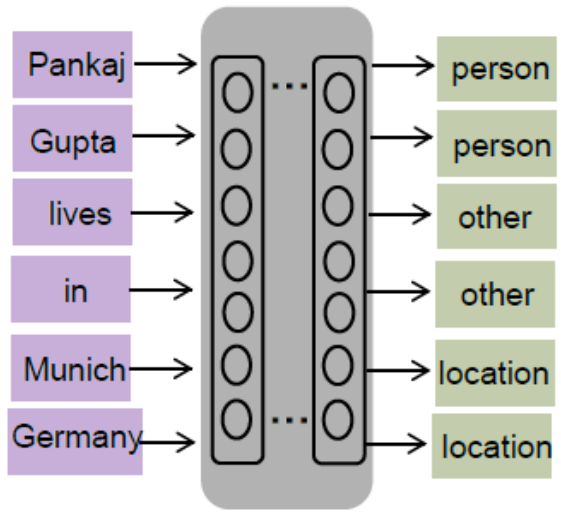
Example:

Sentence1: Pankaj lives in Munich

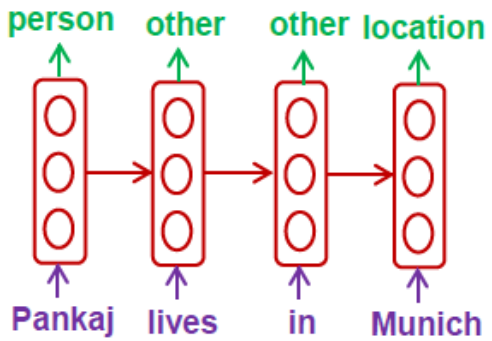
Sentence2: Pankaj Gupta lives in Munich DE



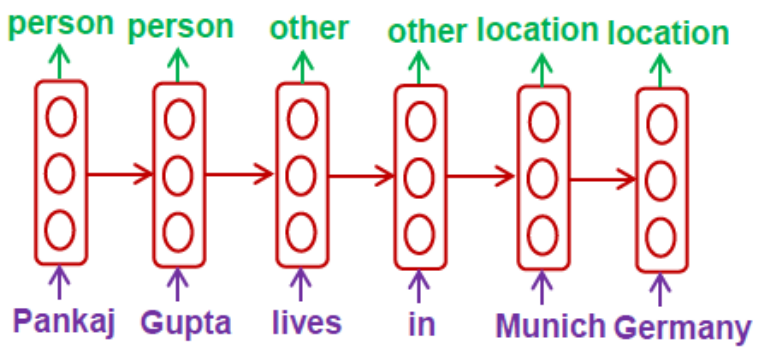
FF-net / CNN



FF-net / CNN



Models variable length sequences



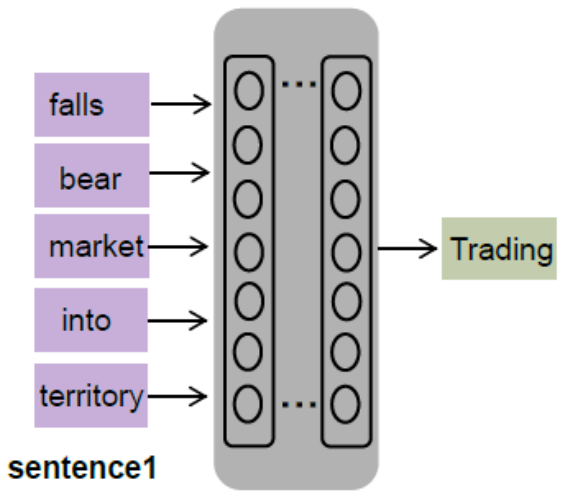
Sequential model: RNN

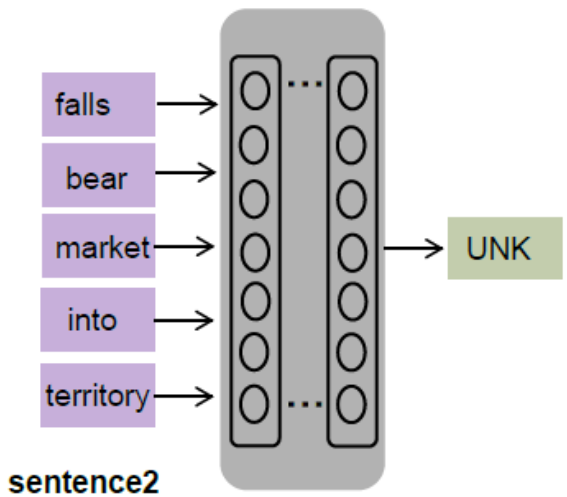
Share Features learned across different positions or time steps

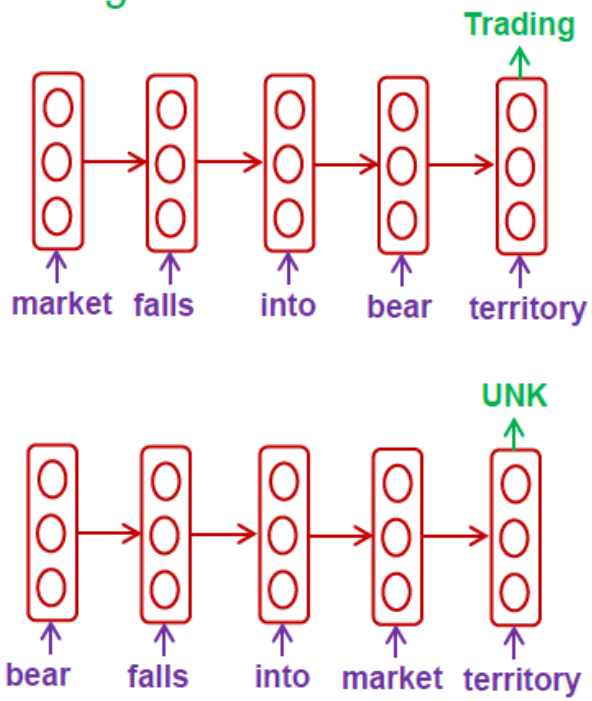
Example:

Sentence1: *Market falls into bear territory* → *Trading/Marketing*

Sentence2: Bear falls into market territory → *UNK*



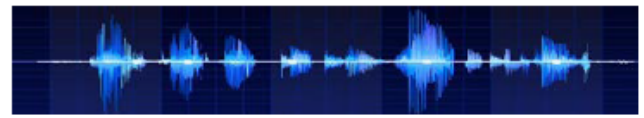




Language concepts,

Word ordering,

Syntactic & semantic information

Examples of Sequence data	Input Data	Output
Speech Recognition		*This is RNN*
Machine Translation	Hello, I am Pankaj.	Hallo, ich bin Pankaj. हेलो, मैं पंकज हूं।
Language Modeling	Recurrent neural ? based ? model	**network** **language**
Named Entity Recognition	Pankaj lives in Munich	**Pankaj** lives in **Munich** ***person*** ***location***
Sentiment Classification	There is nothing to like in this movie.	★☆☆☆☆
Video Activity Analysis		Punching

Why CNNs and Feedforward Networks Don’t Work Well for Sequential Data?

- **No Memory**: They can't remember past inputs or maintain a context across a sequence.
- **Fixed Input Size**: They struggle with variable-length inputs common in tasks like speech or text.
- **Lack of Temporal Understanding**: They can't model dependencies between time steps or sequence elements.
- **Order Agnostic**: Feedforward networks and CNNs don’t capture the order of the data, which is crucial in sequential tasks.

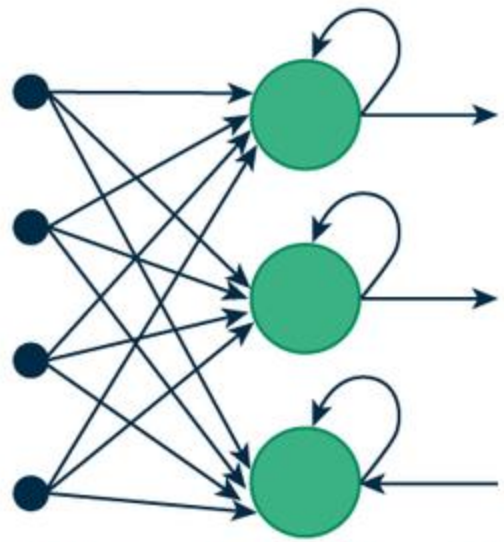
“The **bank** was crowded, and people were standing in long lines”

In this sentence, the word "bank" can have multiple meanings:

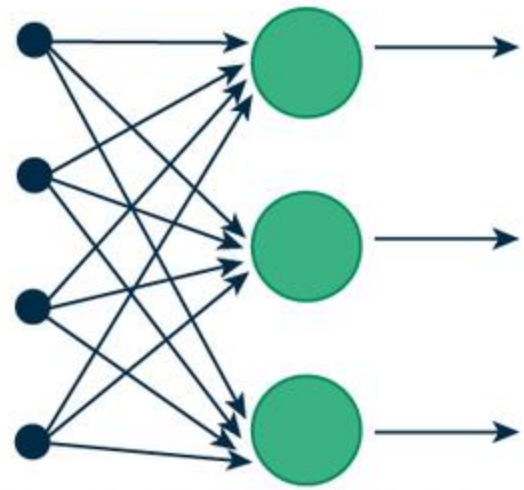
•**Without Context**: "Bank" could refer to either a **financial institution** (where people deposit or withdraw money) or the **side of a river**.

•**With Context**: The phrase "people were standing in long lines" provides crucial contextual information that suggests this is likely a **financial institution**

•The meaning of "bank" is ambiguous when taken out of context, but the surrounding words (like "crowded" and "standing in long lines") clarify the intended meaning. This is a prime example of why **contextual understanding** is important in language tasks.



(a) Recurrent Neural Network



(b) Feed-Forward Neural Network

RNN

- A **Recurrent Neural Network (RNN)** is a type of neural network designed for processing sequences of data by maintaining a form of memory (or context) over time. It is especially well-suited for tasks involving sequential data, such as time series prediction, natural language processing, and speech recognition.
- **Key Features**
  - **Sequential Processing**:
    - Unlike traditional feedforward neural networks, which assume that all inputs are independent of each other, RNNs allow information to persist.
    - They process sequences of inputs one step at a time, maintaining a hidden state (memory state) that carries information about the previous steps in the sequence.
  - **Hidden State:**
    - The hidden state in an RNN serves as memory, enabling the network to retain information from earlier inputs. This state is updated at each step based on both the current input and the previous hidden state.
  - **Shared Weights:**
    - RNNs use the same set of weights across different time steps, which allows them to generalize well to varying sequence lengths.

Architecture Of Recurrent Neural Network

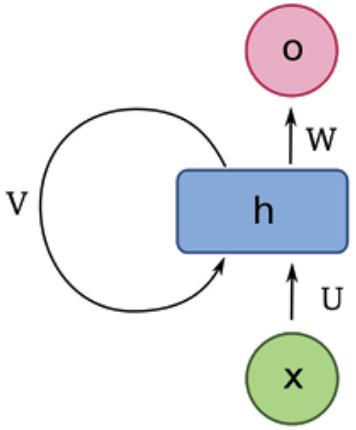
A Recurrent Neural Network (RNN) consists of the following basic architectural components:

1.Input Layer

1. Takes sequential data as input, where each time step corresponds to one input in the sequence.

2.Hidden Layer (Recurrent Layer)

1. Contains recurrent neurons that maintain a **hidden state (memory)** by passing information from one time step to the next. The hidden state stores information from previous time steps, enabling the model to capture sequential dependencies.
2. Typically involves **activation functions** (e.g., tanh or ReLU) to process input and maintain non-linearity.



3.Recurrent Connections (Feedback Loops)

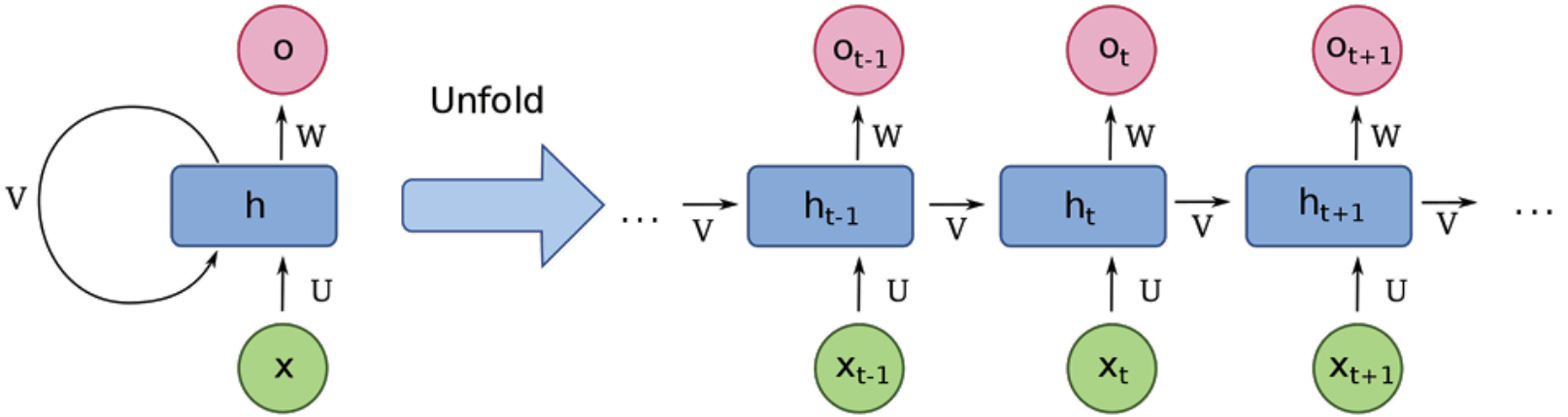
1. Enables information to be passed from one time step to the next, allowing the network to retain context and learn from past data.

4. Output Layer

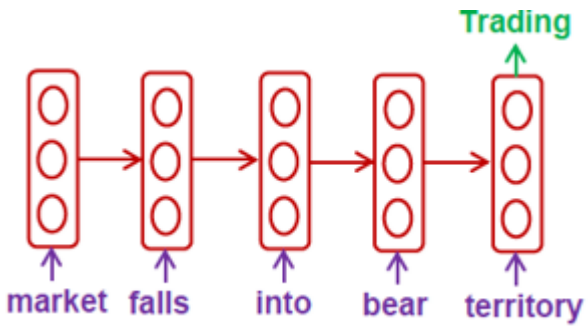
•Produces the final output at each time step or after processing the entire sequence.

•Common activation functions include softmax (for classification) or linear (for regression tasks).

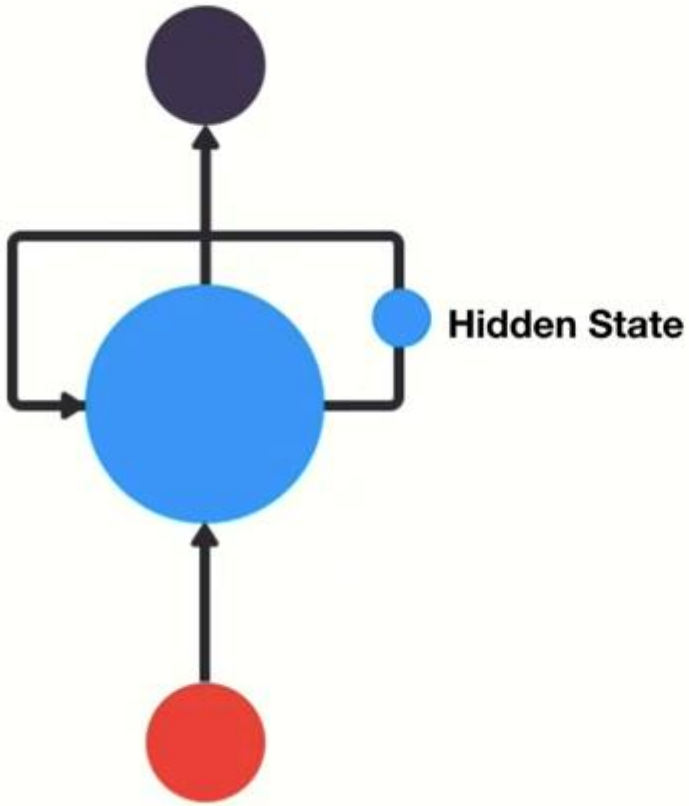
Architecture Of Recurrent Neural Network

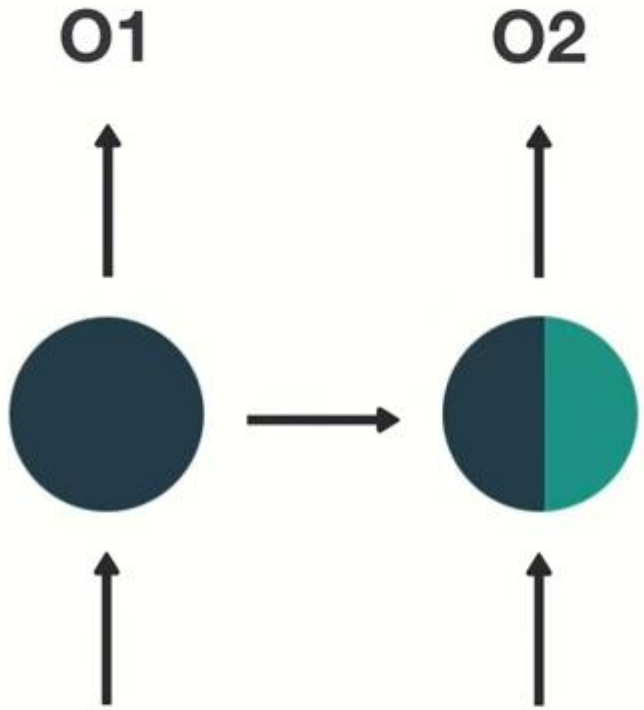


RNNs have the same input and output architecture as any other deep neural architecture. However, differences arise in the way information flows from input to output. Unlike Deep neural networks where we have different weight matrices for each Dense network in RNN, the weight across the network remains the same.

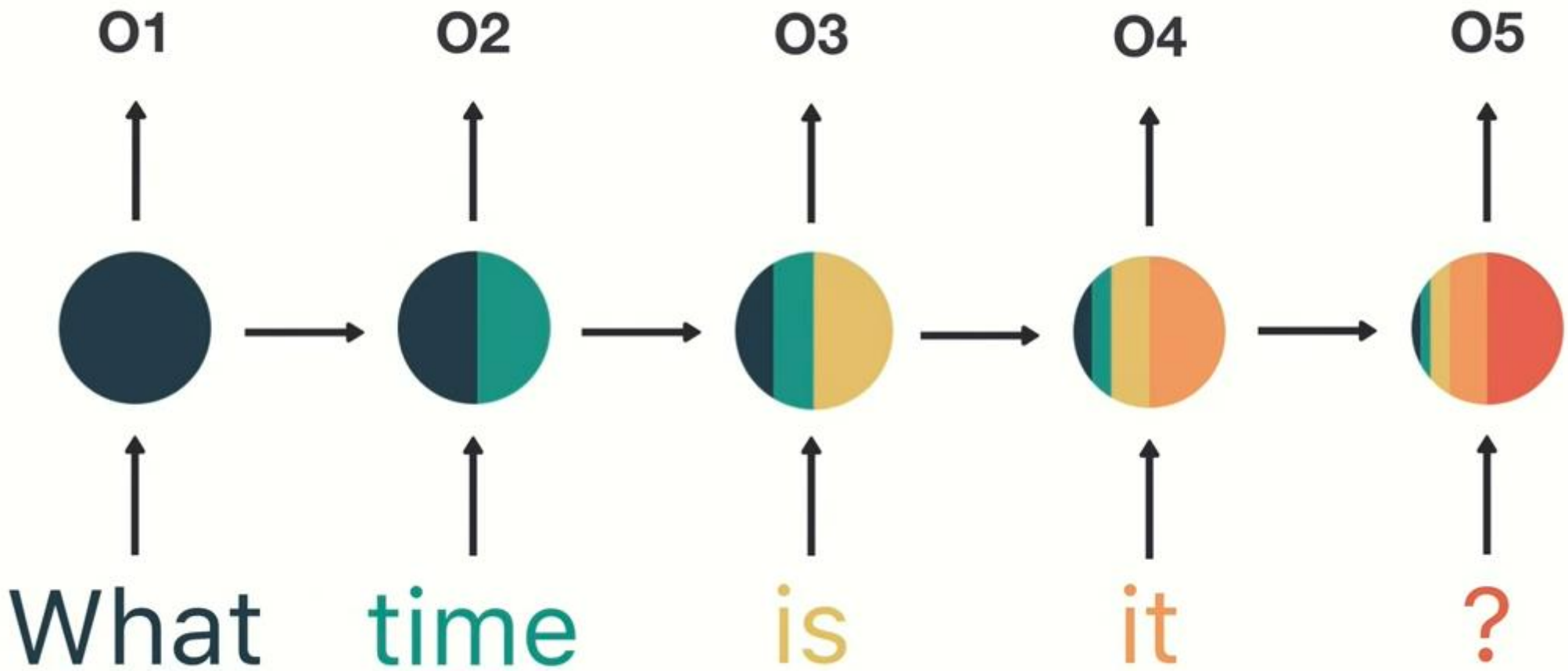


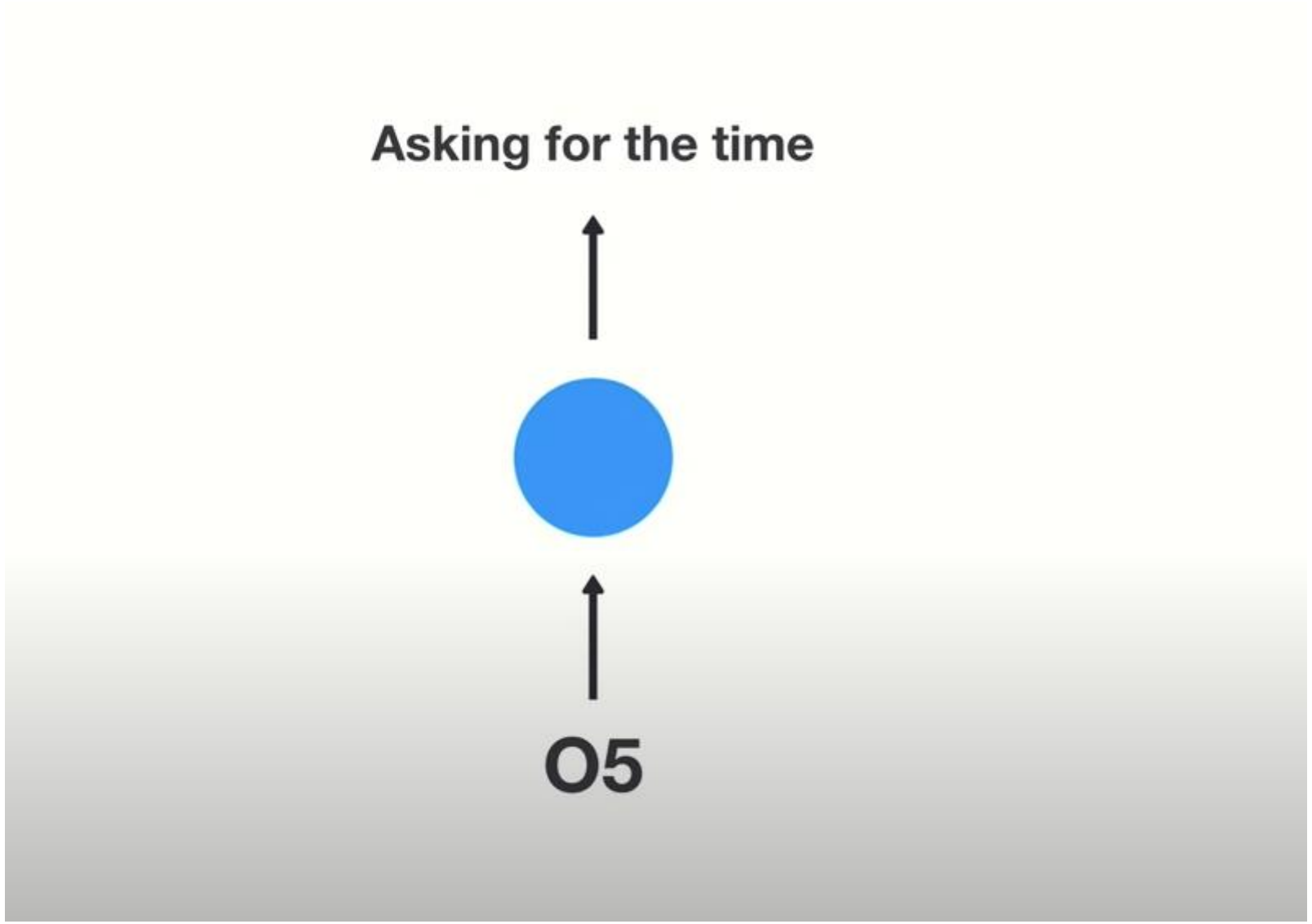
What time is it?





What time is it ?


01
02
03
04
05
What
time
is
it
?


Asking for the time
O5

How RNN Learns?

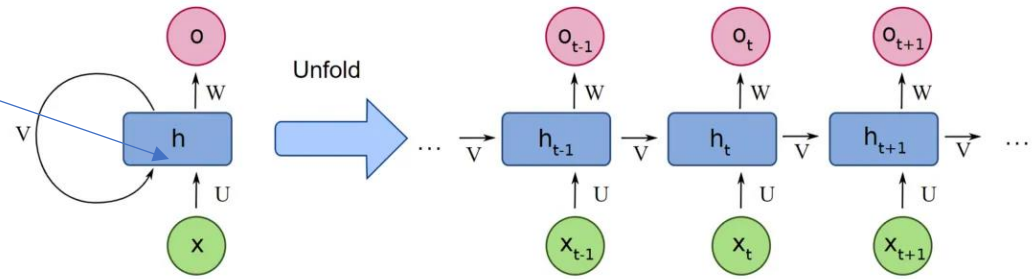
- **Recurrent Neural Networks (RNNs)** learn by adjusting their internal parameters (weights and biases) through a process called **backpropagation through time (BPTT)**. This allows them to learn from **sequential data** by maintaining a hidden state that evolves as they process each input step by step.
- **1. Forward Pass: Processing Sequential Data**
  - In the forward pass, the RNN processes one input at a time, updating its hidden state and generating an output at each time step.

The formula for calculating the current hidden state, h_t :

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

where,

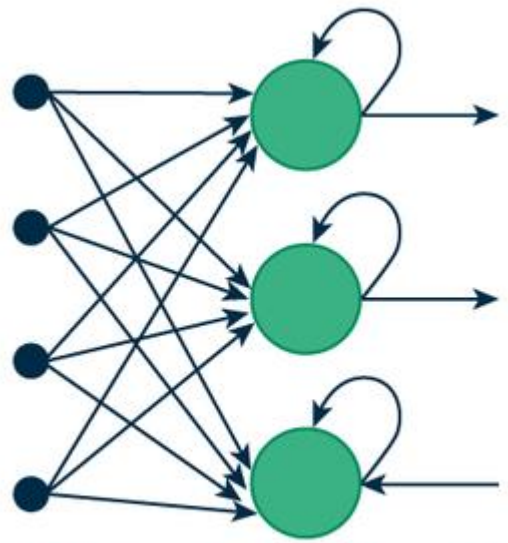
- h_t -> current state
- h_{t-1} -> previous state
- x_t -> input state



The hidden state h_t at time step t is updated based on the input x_t and the previous hidden state h_{t-1}:

U V

$$h_t = f(W_x x_t + W_h h_{t-1} + b)$$



(a) Recurrent Neural Network

Where:

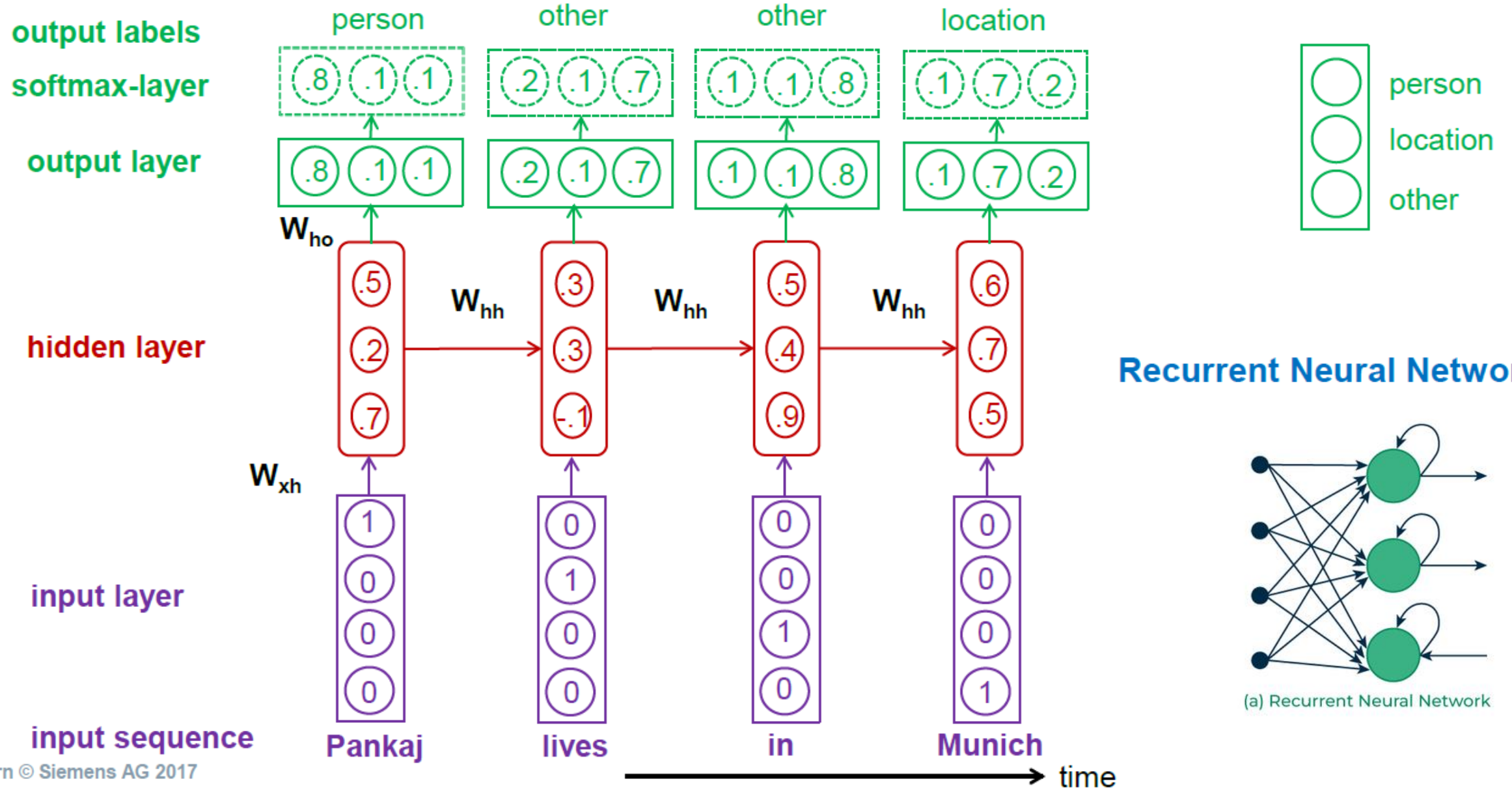
- f is an activation function (e.g., $\tanh$ or ReLU),
- W_x and W_h are weight matrices,
  - **Input weights** W_x: Connecting the current input x_t to the hidden state.
  - **Recurrent weights** W_h: Connecting the previous hidden state h_{t-1} to the current hidden state.
- b is a bias term.

The RNN then produces an output y_t at each time step based on the current hidden state h_t:

$$y_t = g(W_y h_t + b_y)$$

Where:

- g is an activation function (e.g., softmax for classification tasks),
- W_y is the output weight matrix.



Recurrent Neural Network

(a) Recurrent Neural Network

2. **Loss Calculation**

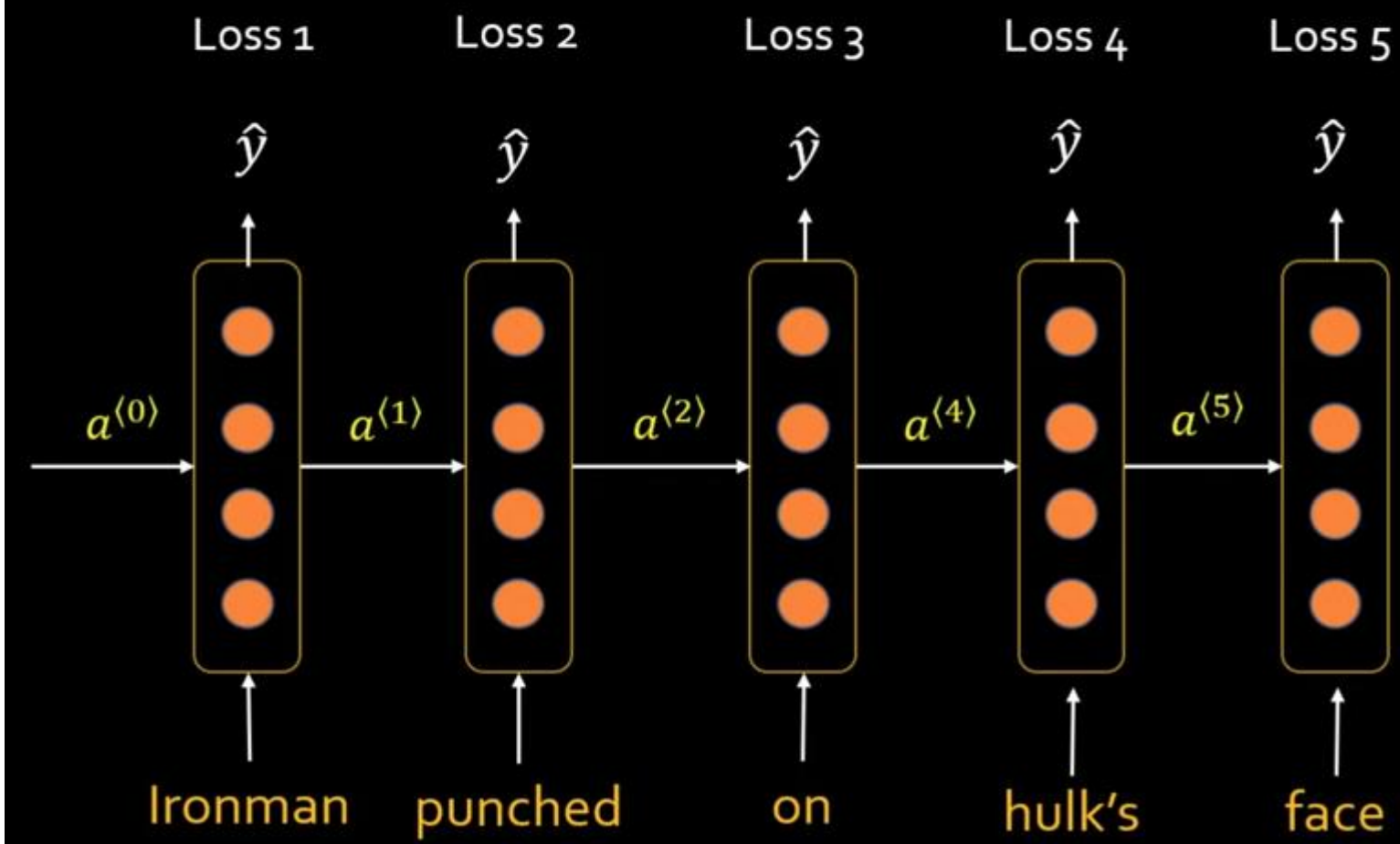
Once the RNN has processed the entire sequence, it compares its predicted outputs y_t with the actual target values $\hat{y}_t$ at each time step. A **loss function** (e.g., mean squared error for regression or cross-entropy for classification) calculates the error between the predicted and actual values:

$$\text{Loss} = \frac{1}{T} \sum_{t=1}^{T} L(y_t, \hat{y}_t)$$

Where T is the total number of time steps, and $L(y_t, \hat{y}_t)$ is the loss at time step t.

Training

Ironman punched on hulk's face → 1 0 0 1 0



Total Loss = Loss 1 + Loss 2 + Loss 3 + Loss4 + Loss 5

3. **Backpropagation Through Time (BPTT)**

Once the loss is calculated, the RNN needs to adjust its weights to minimize this loss. This is where **backpropagation through time (BPTT)** comes in. BPTT is an extension of regular backpropagation but applied to the sequential nature of RNNs.

The key difference in BPTT is that the error gradients are propagated **backward in time** across all time steps, rather than just across layers like in feedforward networks. This involves:

- Calculating the **gradient** of the loss with respect to the weights at each time step.
- **Accumulating gradients** across all time steps, since the weights are shared across the sequence (i.e., W_h is the same at each time step).

The gradient of the loss with respect to the parameters is computed using the chain rule, which involves "unrolling" the RNN over time. For each time step t, the gradient of the loss is calculated and propagated back through time to update the parameters.

4. Updating Weights

After computing the gradients using BPTT, the network updates its weights using **gradient descent** or an optimized variant like **Adam** or **RMSProp**. The update for each weight parameter is typically done as follows:

$$\theta \leftarrow \theta - \eta \cdot \frac{\partial L}{\partial \theta}$$

Where:

- θ represents the model parameters (e.g., W_x, W_h, W_y, b),
- η is the learning rate,
- $\frac{\partial L}{\partial \theta}$ is the gradient of the loss with respect to the parameter θ.

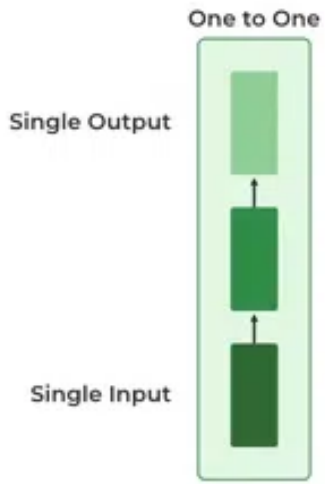
This process is repeated for multiple sequences (during training) until the RNN converges, meaning the loss is minimized, and the model has learned the correct relationships in the sequence data.

Training through RNN

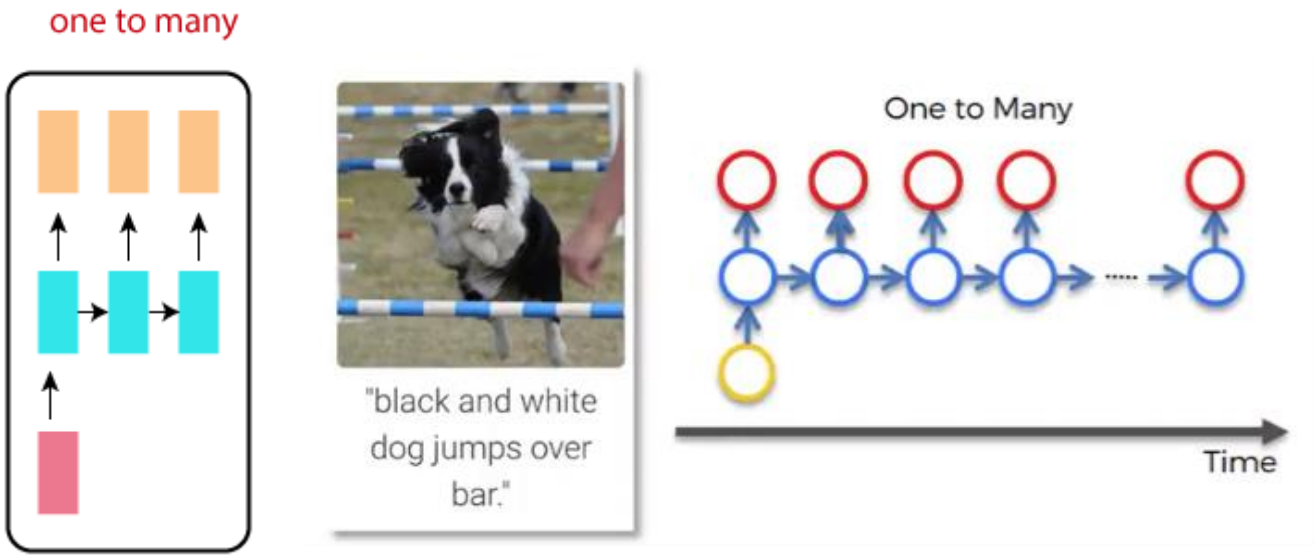
1. A single time step of the input is provided to the network.
2. Then calculate its current state using set of current input and the previous state.
3. The current h_t becomes h_{t-1} for the next time step.
4. One can go as many time steps according to the problem and join the information from all the previous states.
5. Once all the time steps are completed the final current state is used to calculate the output.
6. The output is then compared to the actual output i.e the target output and the error is generated.
7. The error is then back-propagated to the network to update the weights and hence the network (RNN) is trained.

Types Of RNN

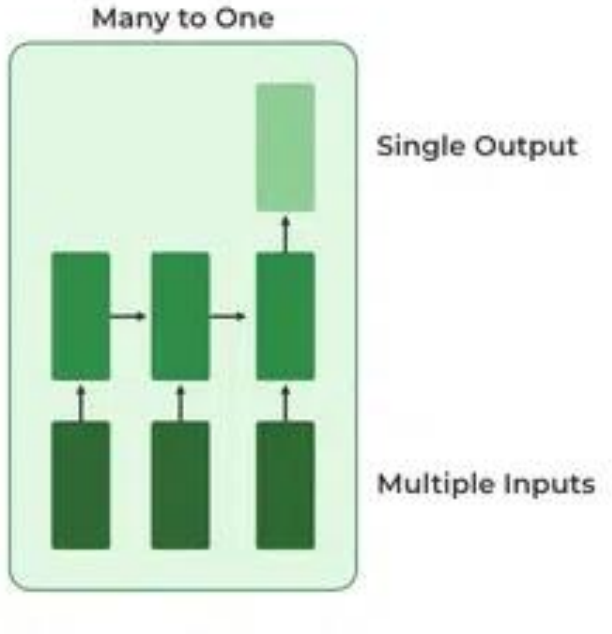
- There are four types of RNNs based on the number of inputs and outputs in the network.
  - One to One
  - One to Many
  - Many to One
  - Many to Many
- **One to One**
  - This type of RNN behaves the same as any simple Neural network it is also known as Vanilla Neural Network. In this Neural network, there is only one input and one output.
  - It deals with a fixed size of the input to the fixed size of output, where they are independent of previous information/output.
  - Example: Image classification.

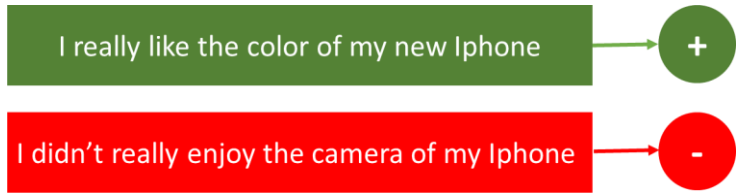


- **One To Many (Vector to sequences)**
  - In this type of RNN, there is one input and many outputs associated with it.
    - **Ex: Image Captioning** – Here, let's say we have an image for which we need a textual description. So we have a single input – the image, and a series or sequence of words as output. Here the image might be of a fixed size, but the output is a description of varying lengths

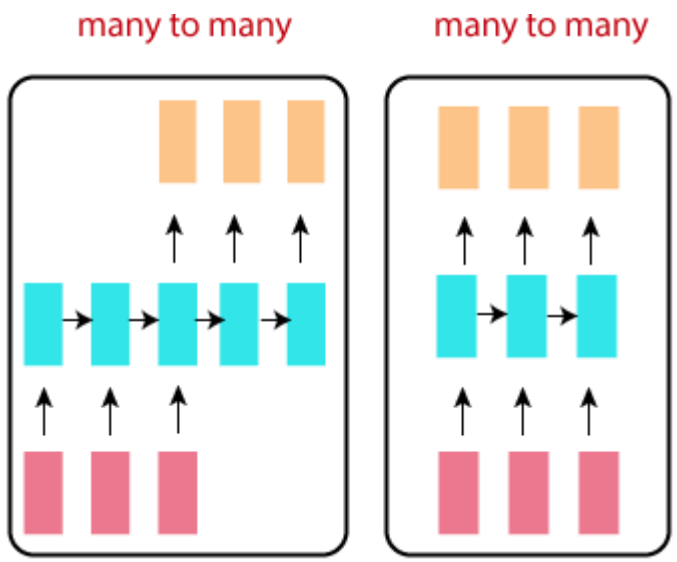


- **Many to One (sequence to vector)**
  - In this type of network, Many inputs are fed to the network at several states of the network generating only one output.
    - Ex: This type of network is used in the problems like sentimental analysis. Where we give multiple words as input and predict only the sentiment of the sentence as output.





- **Many to Many (encoder-decoder)**
  - In this type of neural network, there are multiple inputs and multiple outputs corresponding to a problem.
    - One Example of this Problem will be language translation. In language translation, we provide multiple words from one language as input and predict multiple words from the second language as output.



French was the official language of the colony of French Indochina, comprising modern-day Vietnam, Laos, and Cambodia. It continues to be an administrative language in Laos and Cambodia, although its influence has waned in recent years.

Translate into : French

Le français était la langue officielle de la colonie de l'Indochine française, comprenant le Vietnam d'aujourd'hui, le Laos et le Cambodge. Il continue d'être une langue administrative au Laos et au Cambodge, bien que son influence a décliné au cours des dernières années.

Advantages

- RNN can model a sequence of data so that each sample can be assumed to be dependent on previous ones.
- A recurrent neural network is even used with convolutional layers to extend the active pixel neighborhood.

- **Disadvantages:**

- Gradient vanishing and exploding problems.
- Training an RNN is a very difficult task.
- It cannot process very long sequences

Vanishing and Exploding Gradient Problem

- Regular RNNs might have a difficulty in learning long range dependencies.
  - Ex: if we have a sentence like “The man who ate my pizza has purple hair”.
  - In this case, the description purple hair is for the man and not the pizza. So this is a long dependency.
- If we backpropagate the error in this case, we would need to apply the chain rule. To calculate the error after the third time step with respect to the first one –
- ∂L/∂W =∂L/∂y3 *∂y3/∂h3 *∂h3/∂y2 *∂y2/∂h1 .. and there is a long dependency.
- If any one of the gradients approached 0, all the gradients would rush to zero exponentially fast due to the multiplication. Such states would no longer help the network to learn anything. This is known as the vanishing gradient problem.

- Imagine you have the following:
  - Output Layer Gradient: 0.1
  - Gradient for Layer 2: $0.1 \times 0.1 = 0.01$
  - Gradient for Layer 1: $0.01 \times 0.1 = 0.001$
- By the time the gradients reach the first few layers, they are so small (close to zero) that these layers do not get significant updates, making it impossible for them to learn effectively. This leads to a network that **learns slowly** or becomes unable to capture important features in the input, especially in early layers.

- **Exploding gradient problem**: In the exploding gradient problem, the gradients become too large as they propagate backward through the network, causing huge updates to the weights. This results in unstable training, with the loss function either oscillating wildly or becoming too large to compute, causing the network to diverge.
- Example: Let's assume you are training a simple feedforward neural network with several layers. Assume that we have gradients larger than 1, such as using the ReLU activation function in poorly initialized networks:
  - **During backpropagation, gradients are multiplied by numbers greater than 1.**
  - **As these values are multiplied through the layers, the gradients become exponentially larger.**
  - **Imagine you have the following:**
    - Output Layer Gradient: 5
    - Gradient for Layer 2: 5 × 5 = 25
    - Gradient for Layer 1: 25 × 5 = 125

- As gradients explode, the weight updates become extremely large. This leads to:
  - **Instability in learning**: The weights change drastically from one iteration to the next, preventing the model from converging to a solution.
  - **Oscillating or Diverging Loss**: The loss function might increase instead of decreasing, or oscillate wildly, making it impossible to achieve good performance.

Different RNN Architectures

There are different variations of RNNs that are being applied practically in machine learning problems:

Bidirectional recurrent neural networks (BiRNN)

In BRNN, inputs from future time steps are used to improve the accuracy of the network. It is like having knowledge of the first and last words of a sentence to predict the middle words.

Gated Recurrent Units (GRU)

These networks are designed to handle the vanishing gradient problem. They have a reset and update gate. These gates determine which information is to be retained for future predictions.

Long Short Term Memory (LSTM)

LSTMs were also designed to address the vanishing gradient problem in RNNs. LSTM use three gates called input, output and forget gate. Similar to GRU, these gates determine which information to retain.

Appendix

BPTT: Gradient Calculation

To train the RNN using gradient-based optimization (like SGD), we need to compute the gradients of the total loss with respect to the model parameters: W_h, W_x, and W_y.

Key Gradients:

1. **Gradient of the Total Loss with respect to Output Weights W_y**: Since the output at each time step depends only on the hidden state at that time step h_t:

$$\frac{\partial \mathcal{L}_{\text{total}}}{\partial W_y} = \sum_{t=1}^{T} \frac{\partial \mathcal{L}_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial W_y}$$

   - $\frac{\partial \mathcal{L}_t}{\partial y_t}$ is the gradient of the loss at time step t with respect to the output.
   - $\frac{\partial y_t}{\partial W_y} = h_t$ is the gradient of the output with respect to W_y.

2. **Gradient of the Total Loss with respect to Hidden State at Time Step t:**

$$\frac{\partial \mathcal{L}_{\text{total}}}{\partial h_t} = \frac{\partial \mathcal{L}_t}{\partial y_t} \cdot \frac{\partial y_t}{\partial h_t} + \sum_{k=t+1}^{T} \frac{\partial \mathcal{L}_k}{\partial h_k} \cdot \frac{\partial h_k}{\partial h_t}$$

This recursive equation reflects the fact that the loss at later time steps depends on hidden states from earlier time steps. The second term accumulates the influence of the hidden state h_t on the losses at all later steps k.

The recurrence here gives rise to the **backpropagation through time**, where the gradients are propagated back across multiple time steps.

3. **Gradient of the Loss with respect to the Recurrent Weights W_h**: To compute the gradient with respect to W_h, we need to backpropagate through the recurrent connection in the hidden state update.

$$\frac{\partial \mathcal{L}_{\text{total}}}{\partial W_h} = \sum_{t=1}^{T} \left(\frac{\partial \mathcal{L}_{\text{total}}}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_h} \right)$$

Since h_t depends on h_{t-1} through W_h, we have:

$$\frac{\partial h_t}{\partial W_h} = \frac{\partial h_t}{\partial h_{t-1}} \cdot \frac{\partial h_{t-1}}{\partial W_h}$$

This creates the recurrence, and we need to compute the gradient recursively back through time.

4. **Gradient of the Loss with respect to Input Weights W_x**: Similarly, the gradient with respect to W_x is:

$$\frac{\partial \mathcal{L}_{\text{total}}}{\partial W_x} = \sum_{t=1}^{T} \left(\frac{\partial \mathcal{L}_{\text{total}}}{\partial h_t} \cdot \frac{\partial h_t}{\partial W_x} \right)$$

Since h_t directly depends on x_t through W_x, this is simpler than the gradient with respect to W_h:

$$\frac{\partial h_t}{\partial W_x} = x_t$$

Text vectorization/ embedding

- Vectorizing text for **sequential data processing** involves converting raw text into numerical representations that models can process.

Some terminologies

- **Document**
  - A document is a single text data point. **For Example,** a review of a particular product by the user.
- **Corpus**
  - It a collection of all the documents present in our dataset.
- **Feature**
  - Every unique word in the corpus is considered as a feature.

For Example, Let's consider the 2 documents shown below:

Sentences: Dog hates a cat. It loves to go out and play. Cat loves to play with a ball.

We can build a corpus from the above 2 documents just by combining them.

Corpus = "Dog hates a cat. It loves to go out and play. Cat loves to play with a ball."

And features will be all unique words:

Fetaures: ['and', 'ball', 'cat', 'dog', 'go', 'hates', 'it', 'loves', 'out', 'play', 'to', 'with']

We will call it a feature vector. Here we remember that we will remove 'a' by considering it a single character.

Reference

- Fool-proof RNN explanation | What are RNNs, how do they work? (youtube.com)

- Illustrated Guide to Recurrent Neural Networks: Understanding the Intuition (youtube.com)