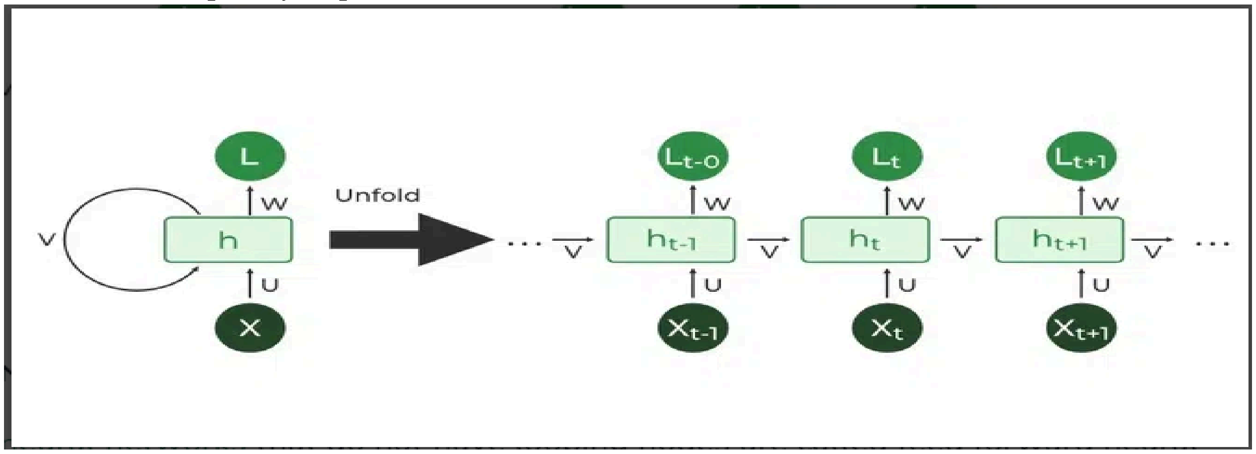
Unit 6

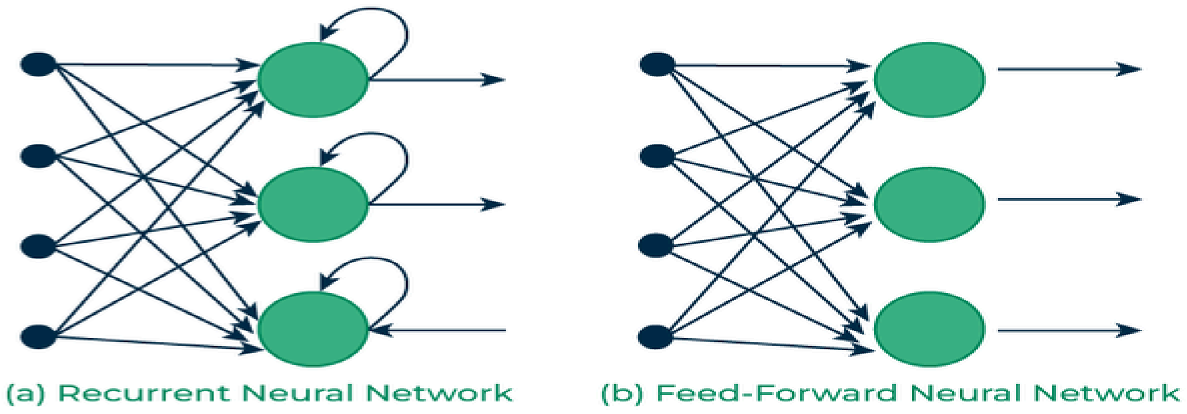
Recurrent Neural Networks (RNN): RNN architecture, Backpropagation through time (BPTT), Vanishing and Exploding Gradients, Truncated BPTT, Long Short Term Memory network and variants of RNN (Gated Recurrent Units, Bidirectional LSTMs, Bidirectional RNNs), Encoder Decoder Models, Attention Mechanism, transformers.

What is Recurrent Neural Network (RNN)?
Recurrent Neural Network(RNN) is a type of Neural Network where the output from the previous step is fed as input to the current step. In traditional neural networks, all the inputs and outputs are independent of each other. Still, in cases when it is required to predict the next word of a sentence, the previous words are required and hence there is a need to remember the previous words. Thus RNN came into existence, which solved this issue with the help of a Hidden Layer. The main and most important feature of RNN is its **Hidden state**, which remembers some information about a sequence. The state is also referred to as *Memory State* since it remembers the previous input to the network. It uses the same parameters for each input as it performs the same task on all the inputs or hidden layers to produce the output. This reduces the complexity of parameters, unlike other neural networks.



Recurrent Neural Network

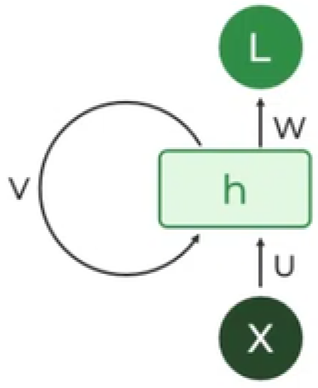
How RNN differs from Feedforward Neural Network?
Artificial neural networks that do not have looping nodes are called feed forward neural networks. Because all information is only passed forward, this kind of neural network is also referred to as a multi-layer neural network.
Information moves from the input layer to the output layer – if any hidden layers are present – unidirectionally in a feedforward neural network. These networks are appropriate for image classification tasks, for example, where input and output are independent. Nevertheless, their inability to retain previous inputs automatically renders them less useful for sequential data analysis.



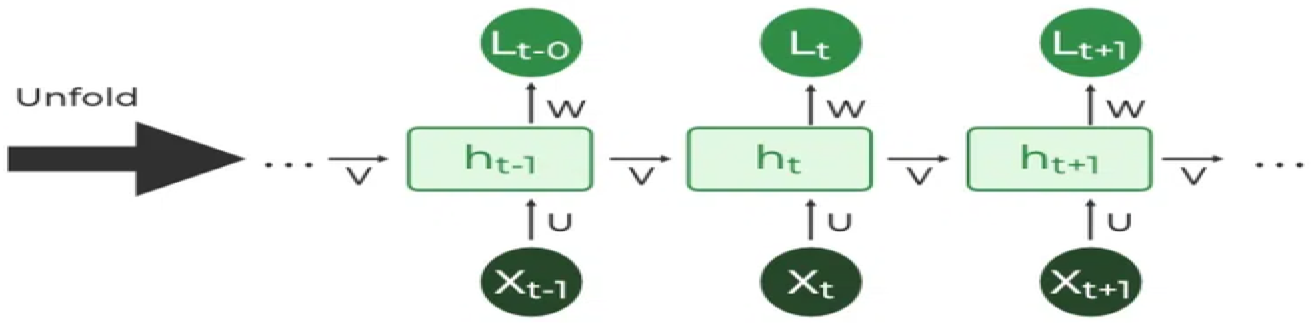
Recurrent Vs Feedforward networks

Recurrent Neuron and RNN Unfolding

The fundamental processing unit in a Recurrent Neural Network (RNN) is a Recurrent Unit, which is not explicitly called a "Recurrent Neuron." This unit has the unique ability to maintain a hidden state, allowing the network to capture sequential dependencies by remembering previous inputs while processing. Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) versions improve the RNN's ability to handle long-term dependencies.



Recurrent Neuron



RNN Unfolding

Given a dataset of time series data, outline the steps to implement a basic RNN from scratch or using a deep learning framework like TensorFlow or PyTorch. Include the architecture and training process.

Implementing a basic Recurrent Neural Network (RNN) for time series data involves several key steps, including data preprocessing, model architecture design, training, and evaluation. Below is an outline of these steps using a deep learning framework like TensorFlow or PyTorch.

Steps to Implement a Basic RNN

1. Data Preprocessing

- **Load the Dataset**: Import the time series dataset (e.g., CSV file).
- **Normalize the Data**: Scale the data to a range suitable for training (e.g., using Min-Max normalization or Standardization).

```
python
Copy code
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()
data_scaled = scaler.fit_transform(data)
```

- **Create Sequences**: Convert the time series data into sequences suitable for RNN input. Define a window size (i.e., how many time steps you want to use as input).

```
python
Copy code
def create_sequences(data, window_size):
    X, y = [], []
    for i in range(len(data) - window_size):
        X.append(data[i:i + window_size])
        y.append(data[i + window_size])
    return np.array(X), np.array(y)

window_size = 10  # Example window size
X, y = create_sequences(data_scaled, window_size)
```

- **Split the Data**: Divide the dataset into training, validation, and test sets.

```
python
Copy code
train_size = int(len(X) * 0.7)
X_train, y_train = X[:train_size], y[:train_size]
X_val, y_val = X[train_size:train_size + int(len(X) * 0.2)],
y[train_size:train_size + int(len(X) * 0.2)]
X_test, y_test = X[train_size + int(len(X) * 0.2):], y[train_size +
int(len(X) * 0.2):]
```

2. Model Architecture

Using TensorFlow/Keras or PyTorch, define the architecture of the RNN.

Using TensorFlow/Keras:

```
python
Copy code
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import SimpleRNN, Dense

model = Sequential()
model.add(SimpleRNN(50, activation='tanh', input_shape=(X_train.shape[1],
X_train.shape[2])))
model.add(Dense(1))  # Output layer for regression
model.compile(optimizer='adam', loss='mean_squared_error')
```

Using PyTorch:

```
python
Copy code
import torch
import torch.nn as nn

class RNNModel(nn.Module):
    def __init__(self, input_size, hidden_size):
        super(RNNModel, self).__init__()
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, 1)

    def forward(self, x):
        out, _ = self.rnn(x)
        out = self.fc(out[:, -1, :])  # Use the last output
        return out

model = RNNModel(input_size=X_train.shape[2], hidden_size=50)
```

3. Training Process

Using TensorFlow/Keras:

```
python
Copy code
history = model.fit(X_train, y_train, validation_data=(X_val, y_val),
epochs=50, batch_size=32)
```

Using PyTorch:

```
python
Copy code
criterion = nn.MSELoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

for epoch in range(50):
    model.train()
    optimizer.zero_grad()
    output = model(torch.FloatTensor(X_train))
    loss = criterion(output, torch.FloatTensor(y_train))
    loss.backward()
    optimizer.step()

    # Validation step
    model.eval()
    val_output = model(torch.FloatTensor(X_val))
    val_loss = criterion(val_output, torch.FloatTensor(y_val))
    print(f'Epoch [{epoch+1}/50], Loss: {loss.item():.4f}, Val Loss:
{val_loss.item():.4f}')
```

4. Evaluation

After training, evaluate the model on the test set.

Using TensorFlow/Keras:

```
python
Copy code
test_loss = model.evaluate(X_test, y_test)
print(f'Test Loss: {test_loss:.4f}')
```

Using PyTorch:

```
python
Copy code
model.eval()
with torch.no_grad():
    test_output = model(torch.FloatTensor(X_test))
    test_loss = criterion(test_output, torch.FloatTensor(y_test))
print(f'Test Loss: {test_loss.item():.4f}')
```

5. Predictions

Make predictions using the trained model and reverse the normalization if necessary.

```
python
Copy code
predictions = model.predict(X_test)
predictions = scaler.inverse_transform(predictions)  # If using Min-Max scaling
```

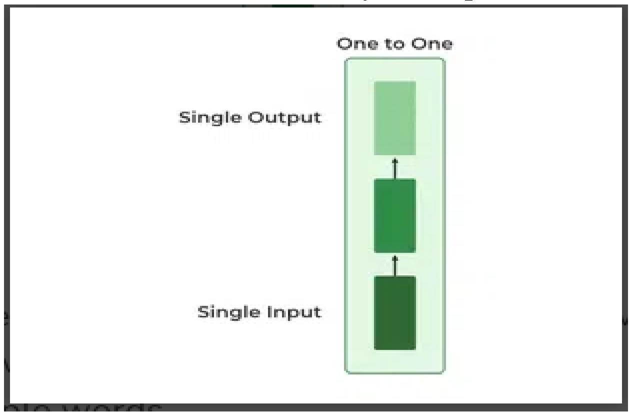
Types Of RNN

There are four types of RNNs based on the number of inputs and outputs in the network.

1. One to One
2. One to Many
3. Many to One
4. Many to Many

One to One

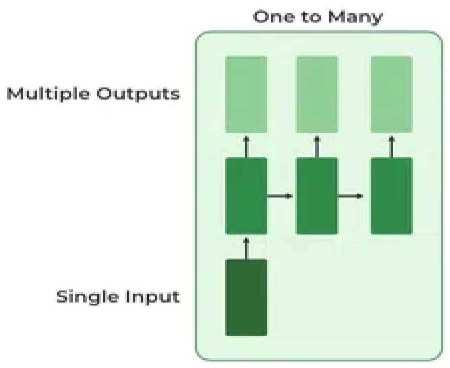
This type of RNN behaves the same as any simple Neural network it is also known as Vanilla Neural Network. In this Neural network, there is only one input and one output.



One to One RNN

One To Many

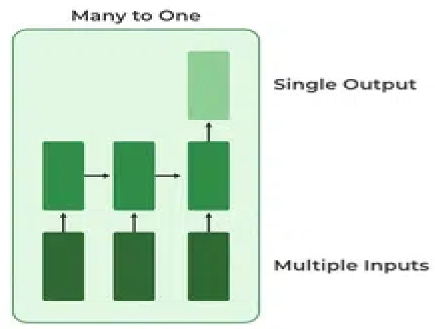
In this type of RNN, there is one input and many outputs associated with it. One of the most used examples of this network is Image captioning where given an image we predict a sentence having Multiple words.



One to Many RNN

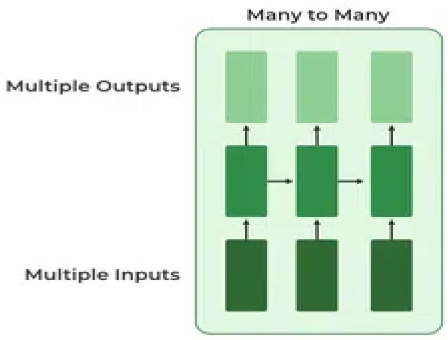
Many to One

In this type of network, Many inputs are fed to the network at several states of the network generating only one output. 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.



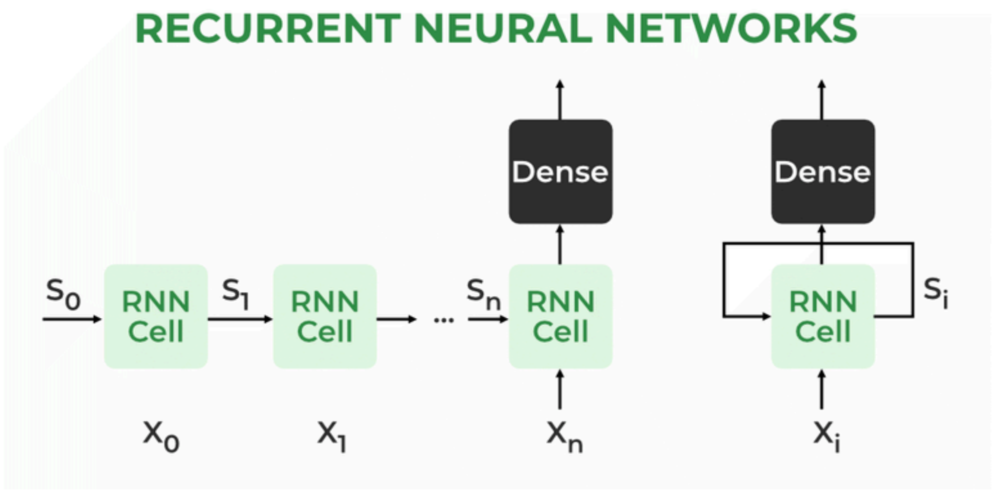
Many to One RNN

Many to Many
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.



Many to Many RNN

Recurrent Neural Network Architecture
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. It calculates state hidden state H**i** for every input **Xi . By using the following formulas:**
$h= \sigma(UX + Wh-1 + B)$
$Y = O(Vh + C)$
Hence
$Y = f(X, h, W, U, V, B, C)$
Here S is the State matrix which has element si as the state of the network at timestep i
The parameters in the network are W, U, V, c, b which are shared across timestep



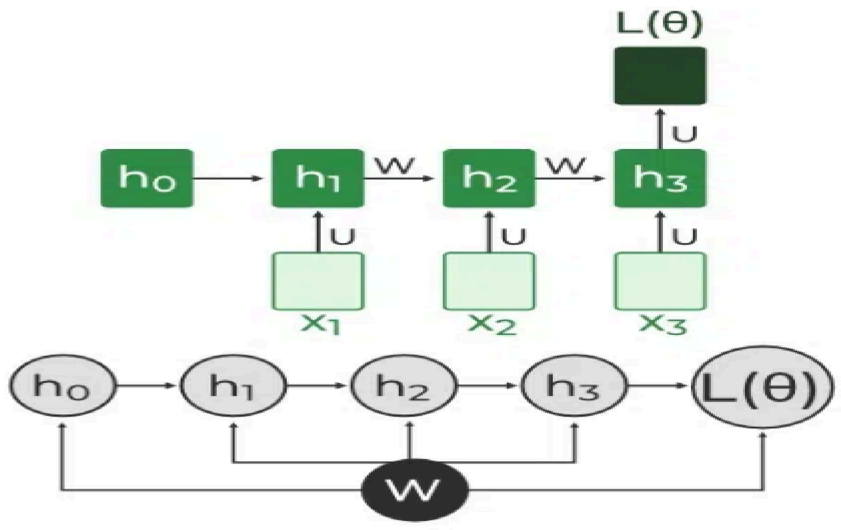
Recurrent Neural Architecture

How does RNN work?
The Recurrent Neural Network consists of multiple fixed activation function units, one for each time step. Each unit has an internal state which is called the hidden state of the unit. This hidden state signifies the past knowledge that the network currently holds at a given time step. This hidden state is updated at every time step to signify the change in the knowledge of the network about the past. The hidden state is updated using the following recurrence relation:-
The formula for calculating the current state:
ht=f(ht−1,xt)*ht*=*f*(*ht*−1,*xt*)
where,
- h_t -> current state
- h_{t-1} -> previous state
- x_t -> input state

Formula for applying Activation function(tanh)
$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t)$
where,
- w_{hh} -> weight at recurrent neuron
- w_{xh} -> weight at input neuron

The formula for calculating output:
$y_t = W_{hy}h_t$
- Y_t -> output
- W_{hy} -> weight at output layer

These parameters are updated using Backpropagation. However, since RNN works on sequential data here we use an updated backpropagation which is known as Backpropagation through time.
Backpropagation Through Time (BPTT)
In RNN the neural network is in an ordered fashion and since in the ordered network each variable is computed one at a time in a specified order like first h1 then h2 then h3 so on. Hence we will apply backpropagation throughout all these hidden time states sequentially.



Backpropagation Through Time (BPTT) In RNN

- L(θ)(loss function) depends on h3
- h3 in turn depends on h2 and W
- h2 in turn depends on h1 and W
- h1 in turn depends on h0 and W
- where h0 is a constant starting state.

$$\frac{\partial \mathbf{L}(\theta)}{\partial W} = \sum_{t=1}^{T} \frac{\partial \mathbf{L}(\theta)}{\partial W}$$

For simplicity of this equation, we will apply backpropagation on only one row

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \frac{\partial h_3}{\partial W}$$

We already know how to compute this one as it is the same as any simple deep neural network backpropagation.

$$\frac{\partial L(\theta)}{\partial h_3}$$

However, we will see how to apply backpropagation to this term ∂h3∂W∂*W*∂*h*3

As we know h3 = σ(Wh2 + b)

And In such an ordered network, we can't compute $\frac{\partial h_3}{\partial W}$ by simply treating h3 as a constant because as it also depends on W. the total derivative $\frac{\partial h_3}{\partial W}$ has two parts:

1. **Explicit:** $\frac{\partial h_3+}{\partial W}$ treating all other inputs as constant
2. **Implicit:** Summing over all indirect paths from h3 to W

Let us see how to do this

$$\begin{aligned}\frac{\partial h_3}{\partial W} &= \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial W} \\ &= \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\left[\frac{\partial h_2^+}{\partial W} + \frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial W}\right] \\ &= \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2}{\partial h_1}\left[\frac{\partial h_1^+}{\partial W}\right]\end{aligned}$$

For simplicity, we will short-circuit some of the paths

$$\frac{\partial h_3}{\partial W} = \frac{\partial h_3^+}{\partial W} + \frac{\partial h_3}{\partial h_2}\frac{\partial h_2^+}{\partial W} + \frac{\partial h_3}{\partial h_1}\frac{\partial h_1^+}{\partial W}$$

Finally, we have

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \cdot \frac{\partial h_3}{\partial W}$$

Where

$$\frac{\partial h_3}{\partial W} = \sum_{k=1}^{3} \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

Hence,

$$\frac{\partial L(\theta)}{\partial W} = \frac{\partial L(\theta)}{\partial h_3} \sum_{k=1}^{3} \frac{\partial h_3}{\partial h_k} \cdot \frac{\partial h_k}{\partial W}$$

This algorithm is called backpropagation through time (BPTT) as we backpropagate over all previous time steps

Issues of Standard RNNs

1. **Vanishing Gradient:** Text generation, machine translation, and stock market prediction are just a few examples of the time-dependent and sequential data problems that can be modelled with recurrent neural networks. You will discover, though, that the gradient problem makes training RNN difficult.
2. **Exploding Gradient:** An Exploding Gradient occurs when a neural network is being trained and the slope tends to grow exponentially rather than decay. Large error gradients that build up during training lead to very large updates to the neural network model weights, which is the source of this issue.

Training through RNN

1. A single-time step of the input is provided to the network.
2. Then calculate its current state using a set of current input and the previous state.
3. The current ht becomes ht-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 using Backpropagation through time.

Advantages and Disadvantages of Recurrent Neural Network

Advantages

1. An RNN remembers each and every piece of information through time. It is useful in time series prediction only because of the feature to remember previous inputs as well. This is called Long Short Term Memory.

2. Recurrent neural networks are even used with convolutional layers to extend the effective pixel neighborhood.

Disadvantages

1. Gradient vanishing and exploding problems.
2. Training an RNN is a very difficult task.
3. It cannot process very long sequences if using tanh or relu as an activation function.

Applications of Recurrent Neural Network

1. Language Modelling and Generating Text
2. Speech Recognition
3. Machine Translation
4. Image Recognition, Face detection
5. Time series Forecasting

Variation Of Recurrent Neural Network (RNN)

To overcome the problems like vanishing gradient and exploding gradient descent several new advanced versions of RNNs are formed some of these are as;

1. Bidirectional Neural Network (BiNN)
2. Long Short-Term Memory (LSTM)

Bidirectional Neural Network (BiNN)

A BiNN is a variation of a Recurrent Neural Network in which the input information flows in both direction and then the output of both direction are combined to produce the input. BiNN is useful in situations when the context of the input is more important such as Nlp tasks and Time-series analysis problems.

Long Short-Term Memory (LSTM)

Long Short-Term Memory works on the read-write-and-forget principle where given the input information network reads and writes the most useful information from the data and it forgets about the information which is not important in predicting the output. For doing this three new gates are introduced in the RNN. In this way, only the selected information is passed through the network.

Difference between RNN and Simple Neural Network

RNN is considered to be the better version of deep neural when the data is sequential. There are significant differences between the RNN and deep neural networks they are listed as:

Recurrent Neural Network	Deep Neural Network
Weights are same across all the layers number of a Recurrent Neural Network	Weights are different for each layer of the network
Recurrent Neural Networks are used when the data is sequential and the number of inputs is not predefined.	A Simple Deep Neural network does not have any special method for sequential data also here the number of inputs is fixed
The Numbers of parameter in the RNN are higher than in simple DNN	The Numbers of Parameter are lower than RNN

Recurrent Neural Network	Deep Neural Network
Exploding and vanishing gradients is the the major drawback of RNN	These problems also occur in DNN but these are not the major problem with DNN

Vanishing and Exploding Gradients

What is Vanishing Gradient?
The vanishing gradient problem is a challenge that emerges during backpropagation when the derivatives or slopes of the activation functions become progressively smaller as we move backward through the layers of a neural network. This phenomenon is particularly prominent in deep networks with many layers, hindering the effective training of the model. The weight updates becomes extremely tiny, or even exponentially small, it can significantly prolong the training time, and in the worst-case scenario, it can halt the training process altogether.
Why the Problem Occurs?
During backpropagation, the gradients propagate back through the layers of the network, they decrease significantly. This means that as they leave the output layer and return to the input layer, the gradients become progressively smaller. As a result, the weights associated with the initial levels, which accommodate these small gradients, are updated little or not at each iteration of the optimization process.
The vanishing gradient problem is particularly associated with the sigmoid and hyperbolic tangent (tanh) activation functions because their derivatives fall within the range of 0 to 0.25 and 0 to 1, respectively. Consequently, extreme weights becomes very small, causing the updated weights to closely resemble the original ones. This persistence of small updates contributes to the vanishing gradient issue.
The sigmoid and tanh functions limit the input values to the ranges [0,1] and [-1,1], so that they saturate at 0 or 1 for sigmoid and -1 or 1 for Tanh. The derivatives at points becomes zero as they are moving. In these regions, especially when inputs are very small or large, the gradients are very close to zero. While this may not be a major concern in shallow networks with a few layers, it is a more pronounced issue in deep networks. When the inputs fall in saturated regions, the gradients approach zero, resulting in little update to the weights of the previous layer. In simple networks this does not pose much of a problem, but as more layers are added, these small gradients, which multiply between layers, decay significantly and consequently the first layer tears very slowly , and hinders overall model performance and can lead to convergence failure.
How can we identify?
Identifying the vanishing gradient problem typically involves monitoring the training dynamics of a deep neural network.

- One key indicator is observing model weights **converging to 0** or stagnation in the improvement of the model's performance metrics over training epochs.
- During training, if the **loss function fails to decrease** significantly, or if there is erratic behavior in the learning curves, it suggests that the gradients may be vanishing.
- Additionally, examining the gradients themselves during backpropagation can provide insights. **Visualization techniques**, such as gradient histograms or norms, can aid in assessing the distribution of gradients throughout the network.

How can we solve the issue?

- **Batch Normalization** : Batch normalization normalizes the inputs of each layer, reducing internal covariate shift. This can help stabilize and accelerate the training process, allowing for more consistent gradient flow.
- **Activation function**: Activation function like **Rectified Linear Unit (ReLU)** can be used. With **ReLU,** the gradient is 0 for negative and zero input, and it is 1 for positive input, which helps alleviate the vanishing gradient issue. Therefore, ReLU operates by replacing poor enter values with 0, and 1 for fine enter values, it preserves the input unchanged.
- **Skip Connections and Residual Networks (ResNets)**: Skip connections, as seen in ResNets, allow the gradient to bypass certain layers during backpropagation. This facilitates the flow of information through the network, preventing gradients from vanishing.
- **Long Short-Term Memory Networks (LSTMs) and Gated Recurrent Units (GRUs)**: In the context of recurrent neural networks (RNNs), architectures like LSTMs and GRUs are designed to address the vanishing gradient problem in sequences by incorporating gating mechanisms .
- **Gradient Clipping**: Gradient clipping involves imposing a threshold on the gradients during backpropagation. Limit the magnitude of gradients during backpropagation, this can prevent them from becoming too small or exploding, which can also hinder learning.

What is Exploding Gradient?

The exploding gradient problem is a challenge encountered during training deep neural networks. It occurs when the gradients of the network's loss function with respect to the weights (parameters) become excessively large.

Why Exploding Gradient Occurs?

The issue of exploding gradients arises when, during backpropagation, the derivatives or slopes of the neural network's layers grow progressively larger as we move backward. This is essentially the opposite of the vanishing gradient problem.

The root cause of this problem lies in the weights of the network, rather than the choice of activation function. High weight values lead to correspondingly high derivatives, causing significant deviations in new weight values from the previous ones. As a result, the gradient fails to converge and can lead to the network oscillating around local minima, making it challenging to reach the global minimum point.

In summary, exploding gradients occur when weight values lead to excessively large derivatives, making convergence difficult and potentially preventing the neural network from effectively learning and optimizing its parameters.

As we discussed earlier, the update for the weights during backpropagation in a neural network is given by:

$$\Delta W_i = -\alpha \cdot \frac{\partial L}{\partial W_i}$$

where,

- ΔW_i : The change in the weight WiWi
- **α**: The learning rate, a hyperparameter that controls the step size of the update.
- **L**: The loss function that measures the error of the model.
- $\frac{\partial L}{\partial W_i}$: The partial derivative of the loss function with respect to the weight Wi, which indicates the gradient of the loss function with respect to that weight.

The exploding gradient problem occurs when the gradients become very large during backpropagation. This is often the result of gradients greater than 1, leading to a rapid increase in values as you propagate them backward through the layers.
Mathematically, the update rule becomes problematic when$|\nabla Wi|>1$, causing the weights to increase exponentially during training.
How can we identify the problem?
Identifying the presence of exploding gradients in deep neural network requires careful observation and analysis during training. Here are some key indicators:

- The loss function exhibits erratic behavior, oscillating wildly instead of steadily decreasing suggesting that the network weights are being updated excessively by large gradients, preventing smooth convergence.
- The training process encounters "NaN" (Not a Number) values in the loss function or other intermediate calculations..
- If network weights, during training exhibit significant and rapid increases in their values, it suggests the presence of exploding gradients.
- Tools like TensorBoard can be used to visualize the gradients flowing through the network.

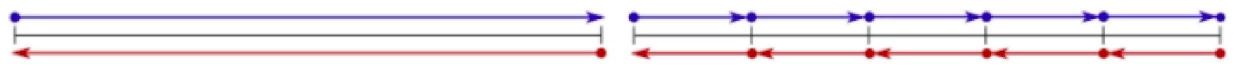
How can we solve the issue?

- **Gradient Clipping**: It sets a maximum threshold for the magnitude of gradients during backpropagation. Any gradient exceeding the threshold is clipped to the threshold value, preventing it from growing unbounded.
- **Batch Normalization:** This technique normalizes the activations within each mini-batch, effectively scaling the gradients and reducing their variance. This helps prevent both vanishing and exploding gradients, improving stability and efficiency.

Truncated BPTT

In sequence prediction challenges, recurrent neural networks can learn the temporal dependence across several time steps.
Backpropagation Through time is a version of the Backpropagation method used to train modern recurrent neural networks like the Long Short-Term Memory network. Truncated Backpropagation Through Time is a modified version of this approach that is more efficient on sequence prediction problems with very long sequences.
Choosing how many timesteps to utilize as input while training recurrent neural networks like LSTMs using Truncated Backpropagation Through Time is an important setup choice. That is, how to break down your extremely long input sequences into subsequences for optimal efficiency.
The input is considered as fixed-length subsequences in truncated backpropagation through time (TBPTT). The hidden state of the previous subsequence is passed on Cas input to the following subsequence in the forward pass. On the other hand, the computed gradient values are dropped at the end of each subsequence as we move back in gradient computation. The gradient values at time t' are employed in every time step t if $t < t'$ in normal backpropagation. If t'-t exceeds the subsequence length, the gradients do not flow from t' to t in shortened backpropagation.

Difference between BPTT and TBPTT
Graphical representation of BPTT and TBPTT is shown below:


(a) BPTT (b) Truncated BPTT

The full history of activations and inputs in the forward pass (blue arrow above representing hidden/internal state flow) must be stored for use in the backpropagation step in standard backpropagation through time (BPTT) (red arrow shows gradient flow). This can be both computationally and memory intensive, especially for a character language model.
But TBPTT reduces the number of timesteps utilized on the backward pass, allowing it to estimate rather than calculate the gradient used to update the weights.

Why use TBPTT over BPTT?

Truncated Backpropagation Through Time (BPTT) offers the computational benefits of BPTT while eliminating the need for a complete retrace through the whole data sequence at each stage. However, truncation favors short-term dependencies: the gradient estimate of truncated BPTT is biassed. Therefore, it does not benefit from the stochastic gradient theory's convergence guarantees.

Preparing Sequence Data for TBPTT

The number of timesteps utilized in the forward and backward passes of BPTT is determined by how you divide up your sequence data.

Use Data As-Is

If the number of timesteps in each sequence is small, such as tens or a few hundred, you can utilize the input sequences as-is. TBPTT has been suggested to have practical limitations of 200 to 400 timesteps. You can reshape the sequence observations as timesteps for the input data if your sequence data is smaller than or equal to this range.
For example, if you had a collection of 100 univariate sequences with 25 timesteps, you could reshape it into 100 samples, 25 timesteps, and 1 feature, or [100, 25, 1].

Naive Data Split

If your input sequences are large, such as hundreds of timesteps, you may need to divide them up into many contiguous subsequences.
This will need the implementation of a stateful LSTM in Keras, with the internal state being retained across sub-sequence input and only being reset at the conclusion of a true fuller input sequence.
If you had 100 input sequences with 50,000 timesteps, for example, each one might be broken into 100 subsequences with 500 timesteps. One input sequence would provide 100 samples, resulting in a total of 10,000 original samples. Keras' input would have a dimensionality of 10,000 samples, 500 timesteps, and 1 feature, or [10000, 500, 1]. It would be necessary to take care to preserve the state throughout every 100 subsequences and to explicitly or implicitly reset the internal state after every 100 samples.

Domain-Specific Data Split

Knowing the proper number of timesteps to generate a useful estimate of the error gradient can be difficult.
We can generate a model rapidly using the naïve technique, but the model may not be optimal. Alternatively, while learning the issue, we may utilize domain-specific knowledge to predict the number of timesteps that will be important to the model. If the sequence problem is a regression time series, looking at the autocorrelation and partial autocorrelation plots might help you decide on the number of timesteps to use.

Systematic Data Split
You can systematically examine a suite of possible subsequence lengths for your sequence prediction challenge rather than guessing at a reasonable number of timesteps.
You might do a grid search over each sub-sequence length and pick the arrangement that produces the best overall model.
Lean Heavily On Internal State With TBPTT
Each timestep of your sequence prediction problem may be reformulated as having one input and one output.
If you had 100 sequences of 50 timesteps, for example, each timestep would be a new sample. The original 100 samples would be multiplied by 5,000. The three-dimensional input would be [5000, 1, 1], or 5,000 samples, 1 timestep, and 1 feature.
Again, this would need preserving the internal state of the sequence at each timestep and resetting it at the conclusion of each real sequence (50 samples).
Decouple Forward and Backward Sequence Length
For the forward and backward passes of Truncated Backpropagation Through Time, the Keras deep learning package was utilized to support a variable number of timesteps.
In essence, the number of timesteps on input sequences may be used to specify the k1 parameter, while the "truncate gradient" argument on the LSTM layer could be used to specify the k2 parameter.

LSTM *Classic*

Long Short-Term Memory (LSTM), introduced by Sepp Hochreiter and Jürgen Schmidhuber in 1997, is a type of recurrent neural network (RNN) architecture designed to handle long-term dependencies. The key innovation of LSTM lies in its ability to selectively store, update, and retrieve information over extended sequences, making it particularly well-suited for tasks involving sequential data.

The structure of an LSTM network comprises memory cells, input gates, forget gates, and output gates. Memory cells serve as the long-term storage, input gates control the flow of new information into the memory cells, forget gates regulate the removal of irrelevant information, and output gates determine the output based on the current state of the memory cells. This intricate architecture enables LSTMs to effectively capture and remember patterns in sequential data while mitigating the vanishing and exploding gradient problems that often plague traditional RNNs.

The strengths of LSTMs lie in their ability to model long-range dependencies, making them especially useful in tasks such as natural language processing, speech recognition, and time series prediction. They excel in scenarios where the relationships between elements in a sequence are complex and extend over significant periods. LSTMs have proven effective in various applications, including machine translation, sentiment analysis, and handwriting recognition. Their robustness in handling sequential data with varying time lags has contributed to their widespread adoption in both academia and industry.

Bidirectional LSTM (BiLSTM)

Bidirectional Long Short-Term Memory (BiLSTM) is an extension of the traditional LSTM architecture that incorporates bidirectional processing to enhance its ability to capture contextual information from both past and future inputs. Introduced as an improvement over unidirectional LSTMs, BiLSTMs are particularly effective in tasks where understanding the context of a sequence in both directions is crucial, such as natural language processing and speech recognition.

The structure of a BiLSTM involves two separate LSTM layers—one processing the input sequence from the beginning to the end (forward LSTM), and the other processing it in reverse order (backward LSTM). The outputs from both directions are concatenated at each time step, providing a comprehensive representation that considers information from both preceding and succeeding elements in the sequence. This bidirectional approach enables BiLSTMs to capture richer contextual dependencies and make more informed predictions.

The strengths of BiLSTMs lie in their ability to capture long-range dependencies and contextual information more effectively than unidirectional LSTMs. By processing sequences in both directions, BiLSTMs excel in tasks such as named entity recognition, sentiment analysis, and machine translation, where understanding the context of a word or phrase requires considering both its past and future context. The bidirectional nature of BiLSTMs makes them versatile and well-suited for a wide range of sequential data analysis applications.

BiLSTMs are commonly used in natural language processing tasks, including part-of-speech tagging, named entity recognition, and sentiment analysis. They are also applied in speech recognition, where bidirectional processing helps in capturing relevant phonetic and contextual information. Additionally, BiLSTMs find use in time series prediction and biomedical data analysis, where considering information from both directions enhances the model's ability to discern meaningful patterns in the data.

Gated Recurrent Unit (GRU)

Diagrammatically, a Gated Recurrent Unit (GRU) looks more complicated than a classical LSTM. In fact, it's a bit simpler, and due to its relative simplicity trains a little faster than the traditional LSTM. GRUs combine the gating functions of the input gate j and the forget gate f into a single update gate z.

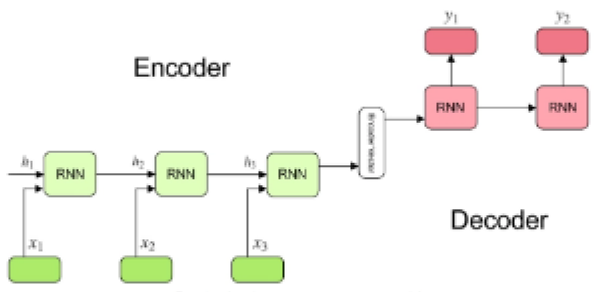
Practically that means that cell state positions earmarked for forgetting will be matched by entry points for new data. Another key difference of the GRU is that the cell state and hidden output h have been combined into a single hidden state layer, while the unit also contains an intermediate, internal hidden state.

The strengths of GRUs lie in their ability to capture dependencies in sequential data efficiently, making them well-suited for tasks where computational resources are a constraint. GRUs have demonstrated success in various applications, including natural language processing, speech recognition, and time series analysis. They are especially useful in scenarios where real-time processing or low-latency applications are essential due to their faster training times and simplified structure.

GRUs are commonly used in natural language processing tasks such as language modeling, machine translation, and sentiment analysis. In speech recognition, GRUs excel at capturing temporal dependencies in audio signals. Moreover, they find applications in time series forecasting, where their efficiency in modeling sequential dependencies is valuable for predicting future data points. The simplicity and effectiveness of GRUs have contributed to their adoption in both research and practical implementations, offering an alternative to more complex recurrent architectures.

Encoder Decoder Models

The encoder-decoder model is a technique of using RNN's for sequence-to-sequence prediction problems. Encoder-decoder was initially developed for machine translation problems, but it has been proven successful at related sequence-to-sequence prediction problems such as question answering. This architecture is very new, been pioneered in 2014, although it has been adopted as the core technology inside Google's translate service.
For a greater understanding of the structure of the encoder-decoder model, previous knowledge of RNN/LSTM/GRU is helpful.
Architecture Of Encoder-Decoder Model
The general structure of the sequence to sequence model commonly used consists of 3 parts: encoder, intermediate vector, and decoder.
Encoder-It accepts a single data element of the input sequence at each step, processes it, collects information for that element, and transfers it forward.
Intermediate vector: It is the final state obtained from the encoder part of the model. This vector contains information about the entire input sequence to help the decoder make accurate predictions.
Decoder- It gives the entire sentence. It predicts an output at each step.

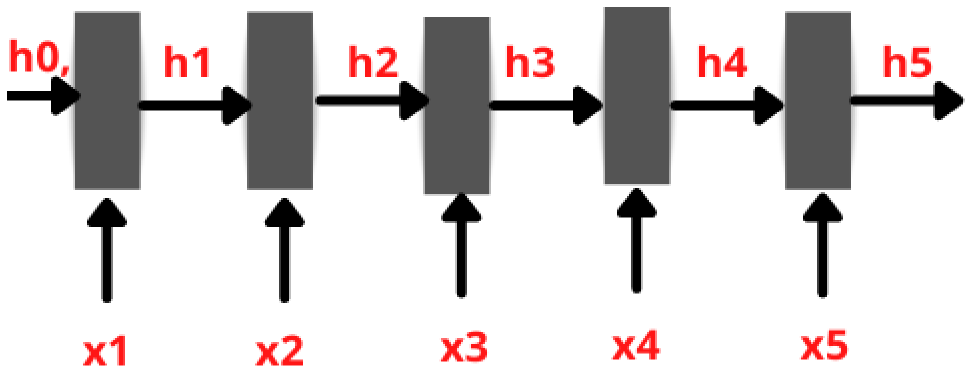


Encoder-decoder sequence to sequence model

Working Of Encoder-Decoder Model
Encoder
The encoder is an LSTM/GRU cell. An encoder part is a stack of several recurrent units (LSTM or GRU cells for better performance). Each unit accepts a single element of the input sequence, collects information for that element, and propagates it forward. The outputs of the encoder are rejected, and only internal states are used.
Enoder cells take only one element as input at a time, so if the input sequence is length x, then the encoder takes x time steps to read the entire sentence.

- Xt is the input at each time step t.
- ht is internal states at time step t of the encoder cell.
- Yt is the final output of encoder cells at time step t.

Let us consider an example of English to Hindi translation. Consider the English sentence- "My name is Mayank Goyal." This sentence contains five words (My, name, is, Mayank, Goyal). Here,
X1 =' My.'
X2=’name'
X3= 'is'
X4=’Mayank'
X5=’Goyal'.
Therefore LSTM/GRU will read this sequence word by word in 5-time steps as follows-



Each input(Xt) is represented as a vector using the word embedding, converting each word into a fixed-length vector. The hidden states h(i) is computed using the formula:

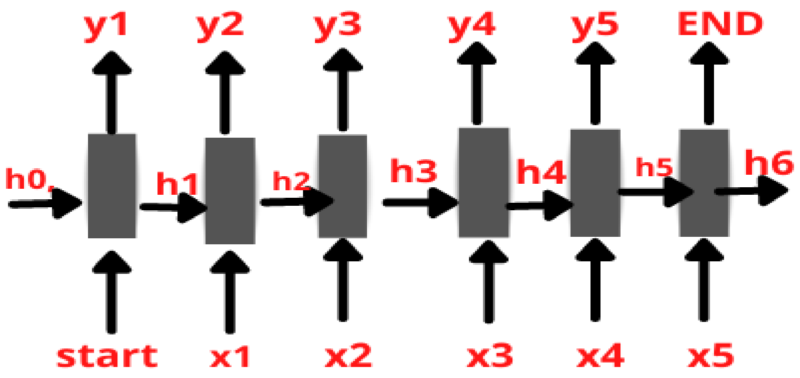
The final state, h5, contains the crux of the entire input sequence.

Intermediate Vector

It is the final hidden state produced from the encoder part of the model. It is calculated using the formula above. This vector encapsulates the information for all input elements to help the decoder cell units to make accurate predictions. It is the initial hidden state of the decoder part of the model.

Decoder Part

Let us understand the working of the decoder part during the training phase. Take the running example of translating My name Mayank Goyal to its Hindi conversion. The decoder also returns the output sequence word by word like an encoder. So we have to generate the output — मेरा नाम मयंक गोयल है| We will add START_ and _END at the stat and end of the sequence for the decoder to recognize the beginning and the ending of the sentence. So after applying changes, our output sentence will be START_मेरा नाम मयंक गोयल है_END.
Let us understand the working visually-



Where,
y1=मेरा
y2=नाम
y3=मयंक
y4= गोयल
y5= है
The initial states (ho) of the decoder are set to the final states of the encoder. We can think that the decoder is trained on the information collected by the encoder. Now, we input the START_ so that the decoder generates the next word. Moreover, after the last word in the Hindi sentence, we make the decoder learn to predict the _END. Decoder consists of a stack of several recurrent units where each unit predicts an output yt at each time step t. Each unit accepts a hidden state from the previous unit and produces an output and its hidden state. We compute the hidden state hi using the formula given below:

As we can see, we are using the previous hidden state to compute the next one. The output yt at each time step t is calculated using the formula given below:

We calculate the output using the hidden state and the respective weight W(S) at the current time step. We use Softmax to create a probability vector that will help us determine the final output (e.g., the word in the question-answering problem). At last, we calculate the loss on the predicted outputs from each step, and the errors are backpropagated through time to update the model's parameters. The final states of the decoder are discarded as we get the output. Hence it is of no use.
That is how the encoder-decoder model works.

Applications
Google Translation
The Encoder-decoder model reads an input sentence, understands the message and the concepts, then translates it into a second language. Google Translate is built upon an encoder-decoder structure.
Sentimental Analysis
Encoder-decoder models understand the meaning of the input sentence and output a sentiment score. The sentiment score is usually rated between -1 (negative) and 1 (positive), where 0 is neutral. It is used in call centers to analyze the client's emotions and reactions to specific keywords or company discounts.

Video/Image Captioning

These models generate a sentence describing an image. The image is fed as the input and outputs a sequence of words. This also works with videos.

Attention Mechanism

An attention mechanism is an Encoder-Decoder kind of neural network architecture that allows the model to focus on specific sections of the input while executing a task. It dynamically assigns weights to different elements in the input, indicating their relative importance or relevance. By incorporating attention, the model can selectively attend to and process the most relevant information, capturing dependencies and relationships within the data. This mechanism is particularly valuable in tasks involving sequential or structured data, such as natural language processing or computer vision, as it enables the model to effectively handle long-range dependencies and improve performance by selectively attending to important features or contexts.

Recurrent models of visual attention use reinforcement learning to focus attention on key areas of the image. A recurrent neural network governs the peek network, which dynamically selects particular locations for exploration over time. In classification tasks, this method outperforms convolutional neural networks. Additionally, this framework goes beyond image identification and may be used for a variety of visual reinforcement learning applications, such as helping robots choose behaviours to accomplish particular goals. Although the most basic use of this strategy is supervised learning, the use of reinforcement learning permits more adaptable and flexible decision-making based on feedback from past glances and rewards earned throughout the learning process.

The application of attention mechanisms to image captioning has substantially enhanced the quality and accuracy of generated captions. By incorporating attention, the model learns to focus on pertinent image regions while creating each caption word. The model can synchronize the visual and textual modalities by paying attention to various areas of the image at each time step thanks to the attention mechanism. By focusing on important objects or areas in the image, the model is able to produce captions that are more detailed and contextually appropriate. The attention-based image captioning models have proven to perform better at catching minute details, managing complicated scenes, and delivering cohesive and educational captions that closely match the visual material.

The attention mechanism is a technique used in machine learning and natural language processing to increase model accuracy by focusing on relevant data. It enables the model to focus on certain areas of the input data, giving more weight to crucial features and disregarding unimportant ones. Each input attribute is given a weight based on how important it is to the output in order to accomplish this. The performance of tasks requiring the utilization of the attention mechanism has significantly improved in areas including speech recognition, image captioning, and machine translation.

How Attention Mechanism Works

An attention mechanism in a neural network model typically consists of the following steps:

1. **Input Encoding:** The input sequence of data is represented or embedded using a collection of representations. This step transforms the input into a format that can be processed by the attention mechanism.

2. **Query Generation:** A query vector is generated based on the current state or context of the model. This query vector represents the information the model wants to focus on or retrieve from the input.
3. **Key-Value Pair Creation:** The input representations are split into key-value pairs. The keys capture the information that will be used to determine the importance or relevance, while the values contain the actual data or information.
4. **Similarity Computation:** The similarity between the query vector and each key is computed to measure their compatibility or relevance. Different similarity metrics can be used, such as dot product, cosine similarity, or scaled dot product.

   where,
   - hs: Encoder source hidden state at position s
   - yi: Encoder Target hidden state at the position i
   - W: Weight Matrix
   - v : Weight vector
5. **Attention Weights Calculation:** The similarity scores are passed through a softmax function to obtain attention weights. These weights indicate the importance or relevance of each key-value pair.

6. **Weighted Sum:** The attention weights are applied to the corresponding values, generating a weighted sum. This step aggregates the relevant information from the input based on their importance determined by the attention mechanism.

   Here,

   Ts: Total number of key-value pairs (source hidden states) in the encoder.
7. **Context Vector:** The weighted sum serves as a context vector, representing the attended or focused information from the input. It captures the relevant context for the current step or task.
8. **Integration with the Model:** The context vector is combined with the model's current state or hidden representation, providing additional information or context for subsequent steps or layers of the model.
9. **Repeat**: Steps 2 to 8 are repeated for each step or iteration of the model, allowing the attention mechanism to dynamically focus on different parts of the input sequence or data.

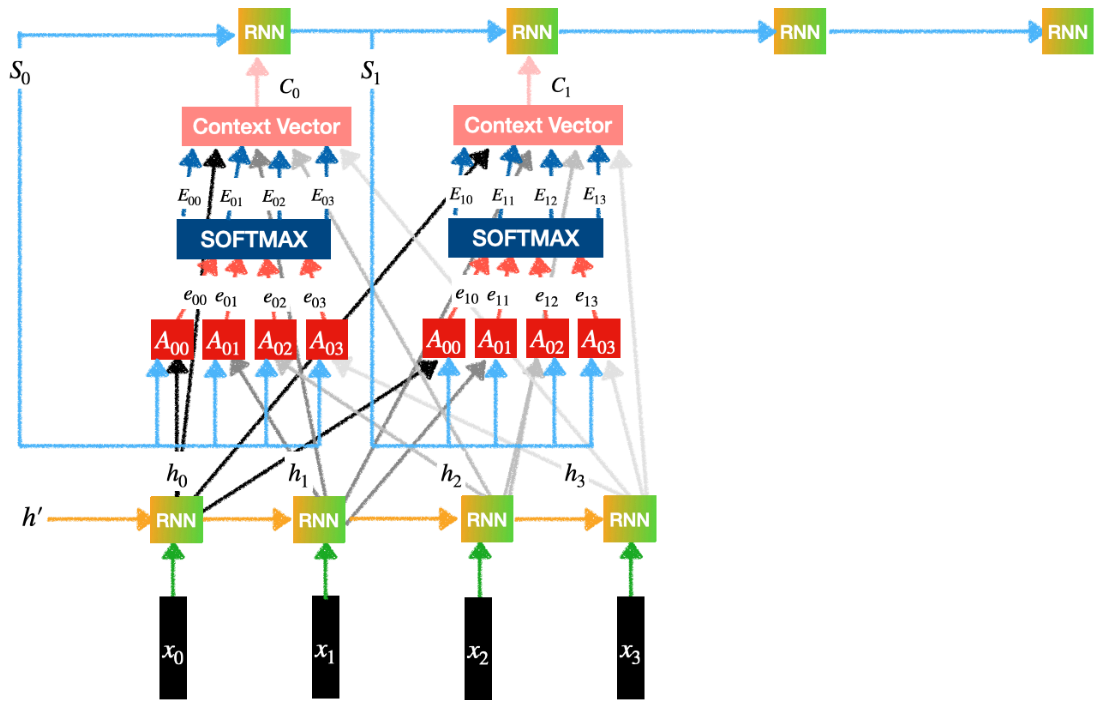
By incorporating an attention mechanism, the model can effectively capture dependencies, emphasize important information, and adaptively focus on different elements of the input, leading to improved performance in tasks such as machine translation, text summarization, or image recognition.

Attention Mechanism Architecture for Machine Translation

The attention mechanism architecture in machine translation involves three main components: Encoder, Attention, and Decoder. The Encoder processes the input sequence and generates hidden states. The Attention component computes the relevance between the current target hidden state and the encoder's hidden states, generating attention weights. These weights are used to compute a context vector that captures the relevant information from the encoder's hidden states. Finally, the Decoder takes the context vector and generates the output sequence. This architecture allows the model to focus on different parts of the input sequence during the translation process, improving the alignment and quality of the translations. We can observe 3 sub-parts or components of the Attention Mechanism architecture :

- Encoder
- Attention
- Decoder

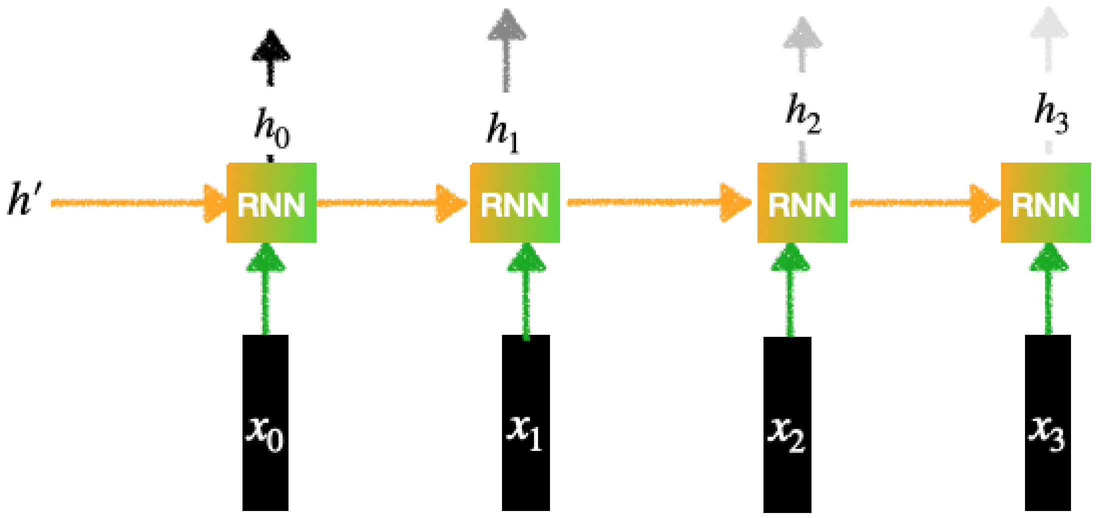
Consider the following Encoder-Decoder architecture with Attention.


RNN
RNN
RNN
RNN
S_0
S_1
C_0
C_1
Context Vector
Context Vector
E_{00} E_{01} E_{02} E_{03}
E_{10} E_{11} E_{12} E_{13}
SOFTMAX
SOFTMAX
e_{00} e_{01} e_{02} e_{03}
e_{10} e_{11} e_{12} e_{13}
A_{00} A_{01} A_{02} A_{03}
A_{00} A_{01} A_{02} A_{03}
h_0 h_1 h_2 h_3
h'
RNN
RNN
RNN
RNN
x_0 x_1 x_2 x_3

Encoder-Decoder with Attention

Encoder:

The encoder applies recurrent neural networks (RNNs) or transformer-based models to iteratively process the input sequence. The encoder creates a hidden state at each step that contains the data from the previous hidden state and the current input token. The complete input sequence is represented by these hidden states taken together.



Encoder

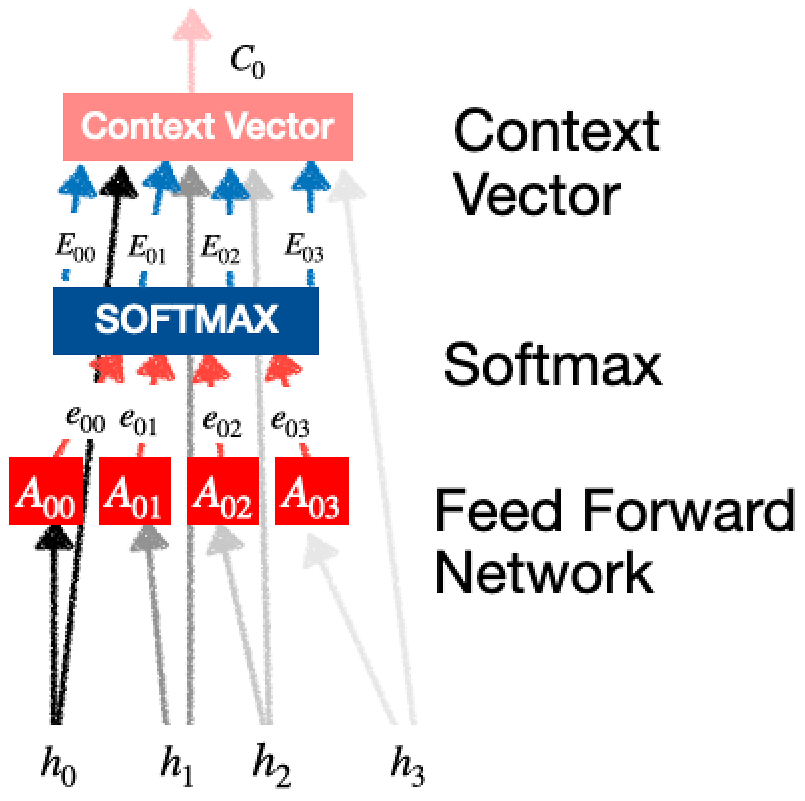
Contains an RNN layer (Can be LSTMs or GRU):

1. Let's, there are 4 words sentence then inputs will be:
2. Each input goes through an Embedding Layer, It can be RNN, LSTM, GRU or trnasformers
3. Each of the inputs generates a hidden representation.
4. This generates the outputs for the Encoder:

Attention:

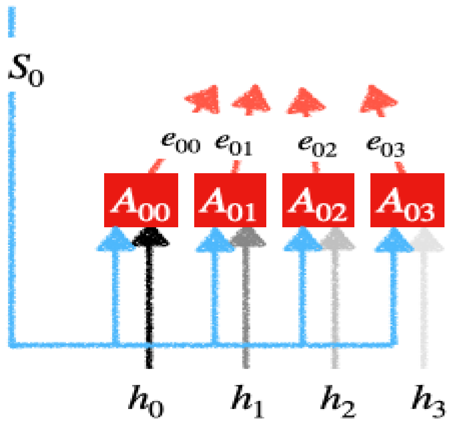
The attention component computes the importance or relevance of each encoder's hidden state with respect to the current target hidden state. It generates a context vector that captures the relevant information from the encoder's hidden states. The attention mechanism can be represented mathematically as follows:

- Our goal is to generate the context vectors.
- For example, context vector tells us how much importance/ attention should be given the inputs: .
- This layer in turn contains 3 subparts:
    - Feed Forward Network
    - Softmax Calculation
    - Context vector generation


C_0
Context Vector
Context Vector
E_{00} E_{01} E_{02} E_{03}
SOFTMAX
Softmax
e_{00} e_{01} e_{02} e_{03}
A_{00} A_{01} A_{02} A_{03}
Feed Forward Network
h_0 h_1 h_2 h_3

attention

Feed Forward Network:
The feed-forward network is responsible for transforming the target hidden state into a representation that is compatible with the attention mechanism. It takes the target hidden state h(t-1) and applies a linear transformation followed by a non-linear activation function (e.g., ReLU) to obtain a new representation


S_0
e_{00} e_{01} e_{02} e_{03}
A_{00} A_{01} A_{02} A_{03}
h_0 h_1 h_2 h_3

Feed-Forward-Network

Each is a simple feed-forward neural network with one hidden layer. The input for this feed-forward network is:

- Previous Decoder state
- The output of Encoder states.

Each unit generates outputs: .i.e

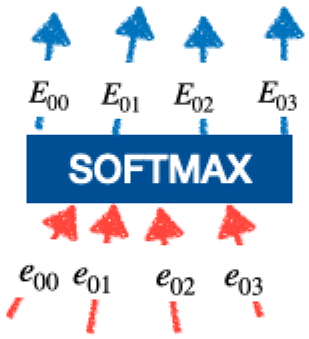
.

Here,

- g can be any activation function such as sigmoid, tanh, or ReLu.

Attention Weights or Softmax Calculation:

A softmax function is then used to convert the similarity scores into attention weights. These weights govern the importance or attention given to each encoder's hidden state. Higher weights indicate higher relevance or importance.

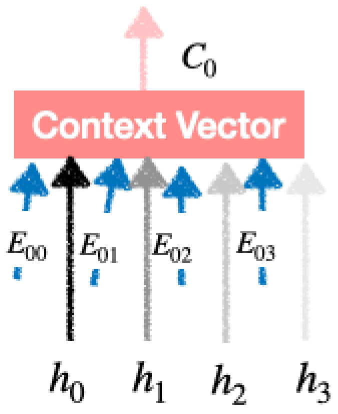


softmax calculation

These are called the attention weights. It decides how much importance should be given to the inputs .

Contact Vector Generation:

Context Vector: The context vector is a weighted sum of the encoder's hidden states, where the attention weights serve as the weights for the summation. It represents a specific arrangement of the encoder's hidden states pertinent to generating the current token.



context vector generation

.
We find in the same way and feed it to different RNN units of the Decoder layer. So this is the final vector which is the product of (Probability Distribution) and (Encoder's output) which is nothing but the attention paid to the input words.

Decoder:

The context vector is fed into the decoder along with the current hidden state of the decoder in order to predict the next token in the output sequence. Until the decoder generates the entire output sequence, this process is done recursively.

We feed these Context Vectors to the RNNs of the Decoder layer. Each decoder produces an output which is the translation for the input words.