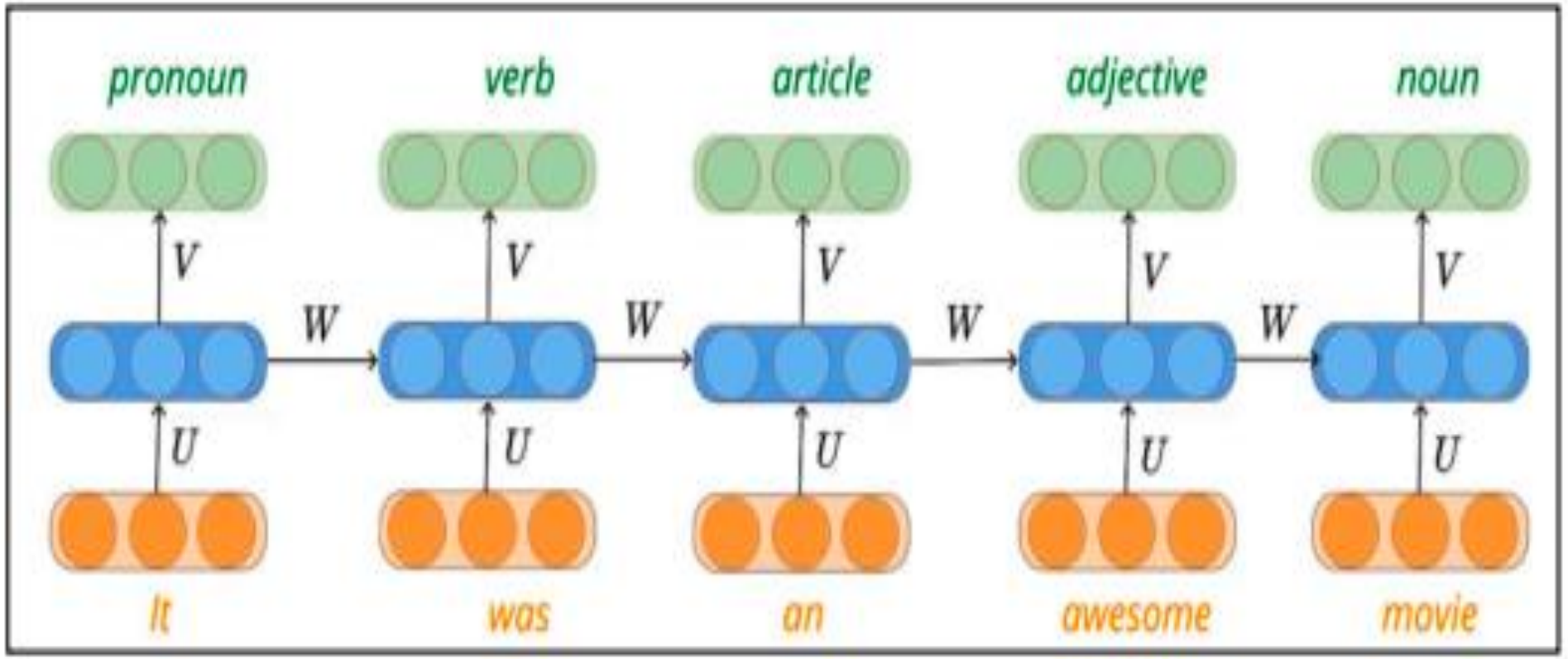
L10 Recurrent Neural Networks

Recurrent Neural Networks

- A recurrent neural network (RNN) is a deep learning model that is trained to process and convert a sequential data input into a specific sequential data output.
- Sequential data is data—such as words, sentences, or time-series data—where sequential components interrelate based on complex semantics and syntax rules.
- Traditional feedforward networks cannot be used for learning and prediction of sequential or time series data
- Feedforward neural networks are only meant for data points that are independent of each other.
- A mechanism is required to retain past or historical information to forecast future values.
- Note: RNNs are largely being replaced by transformer-based artificial intelligence (AI) and large language models (LLM), which are much more efficient in sequential data processing.

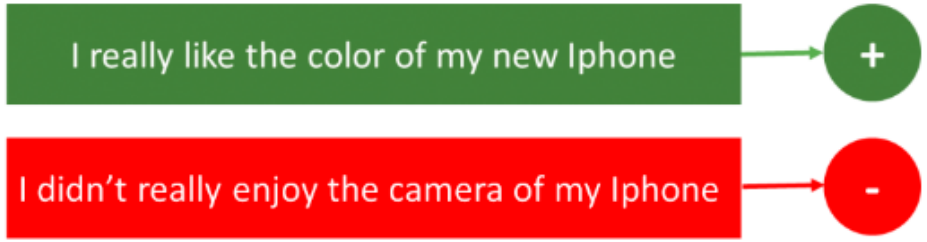
RNN – Application to Parts of Speech (POS) Tagging



RNN – Diversity of applications

- Do we need a network specially for dealing with sequences in information?
- Recurrent Neural Networks(RNNs) are suitable for problems dealing with sequential data and Natural Language Processing(NLP) problems
- Ability of recurrent neural networks lies in their diversity of application to deal with various input and output types.

1. Sentiment Classification –task of classifying tweets into positive and negative sentiment. Here the input would be a tweet of varying lengths, while output is of a fixed type and size.



RNN – Diversity of applications

2. Image Captioning –For an image we need a textual description.

- Here, 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

Time series data

3. Time Series Prediction - Any time series prediction such as predicting the prices of stocks in a particular month, can be solved using an RNN.

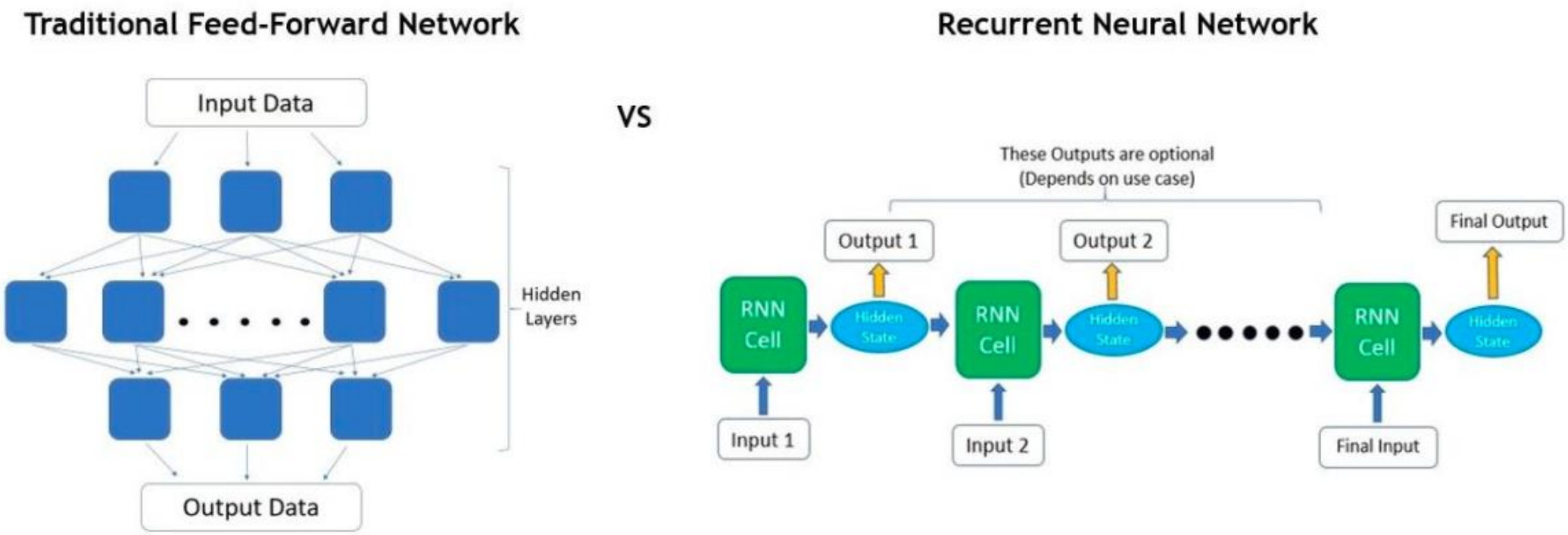
- A time series is a collection of data points gathered over a period of time and ordered chronologically.
- The primary characteristic of a time series is that it is indexed or listed in time order, which is a critical distinction from other types of data sets.
- To plot the points of time series data on a graph, one of the axes would always be time.
- Examples of time series analysis:
  - Electrical activity in the brain
  - Rainfall measurements
  - Stock prices
  - Number of sunspots
  - Annual retail sales
  - Monthly subscribers
  - Heartbeats per minute

RNN – Diversity of applications

4. Language Translation –we have text in a particular language, say English, and we wish to translate it in Hindi.

- Each language has its own semantics and would have varying lengths for the same sentence.
- So here the inputs as well as outputs are of varying lengths.
- RNNs can be used for mapping inputs to outputs of varying types, lengths and are generalized in their application.
- Architecture of RNN, considering the above applications – Feed forward networks vs RNN

Feed forward networks vs RNN



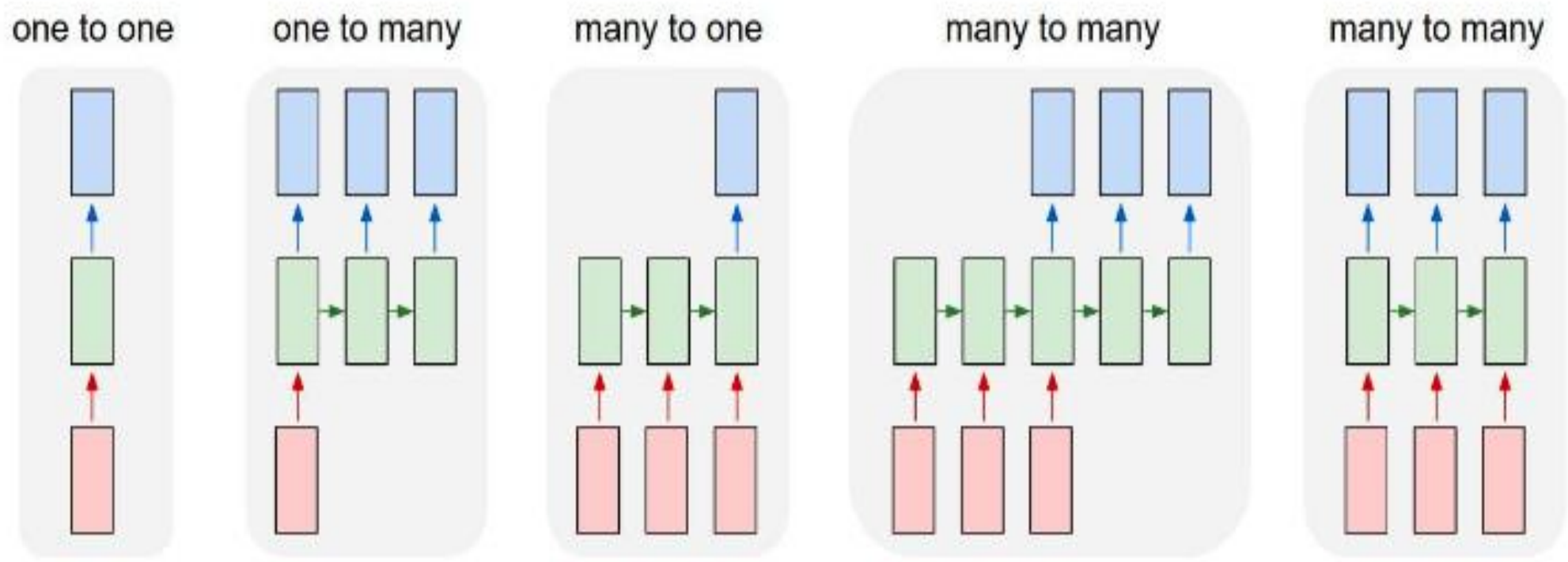
Feed forward networks vs RNN

- The main difference is in how the input data is taken in by the model.
- Feed-forward neural networks take in a fixed amount of input data all at the same time and produce a fixed amount of output each time.
- On the other hand, RNNs do not consume all the input data at once.
- Instead, they take the input data one at a time and in a sequence.
- At each step, the RNN does a series of calculations before producing an output.
- The output, known as the hidden state, is then combined with the next input in the sequence to produce another output.
- This process continues until the model is programmed to finish or the input sequence ends.

Feed forward networks vs RNN

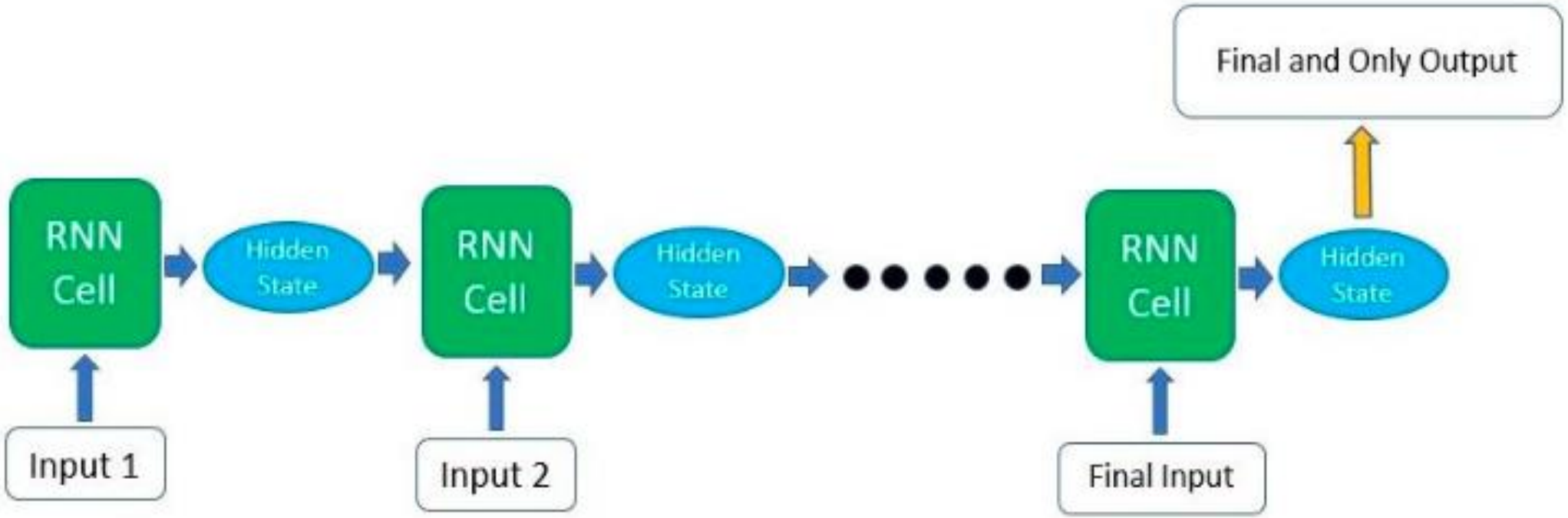
- Calculations at each time step consider the context of the previous time steps in the form of the hidden state.
- Being able to use this contextual information from previous inputs is the key essence to RNNs' success in sequential problems.
- Underlying principle of Recurrent Neural Networks is that the RNN cell is actually the exact same one and reused throughout.

Processing RNN



input and output size can come in different forms as above, yet they can still be fed into and extracted from the RNN model.

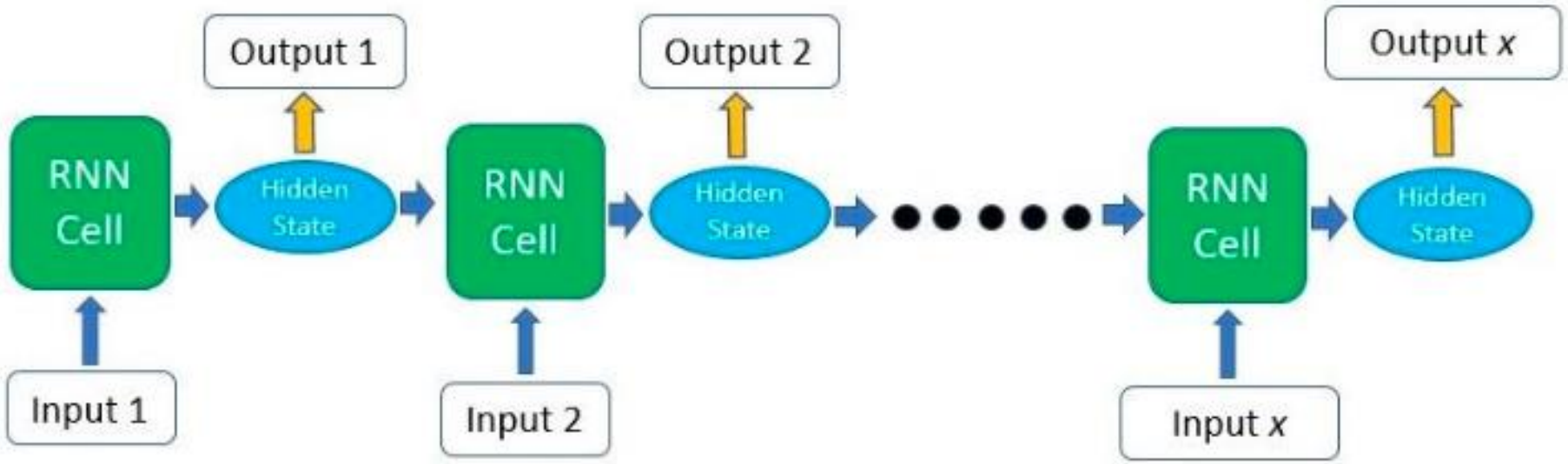
Many inputs to one output



Many inputs to one output

The final result is dependent on all the previous computations and inputs.

Many inputs to many outputs



Many inputs to many outputs

Here output information is taken from the hidden state produced at each step

Processing RNN

- For NLP
  - Text Classification: many-to-one
  - Text Generation: many-to-many
  - Machine Translation: many-to-many
  - Named Entity Recognition: many-to-many
  - Image Captioning: one-to-many
- For Time Series
  - Forecasting - many-to-many or many-to-one
  - Classification - many-to-one

RNN Activation function - tanh

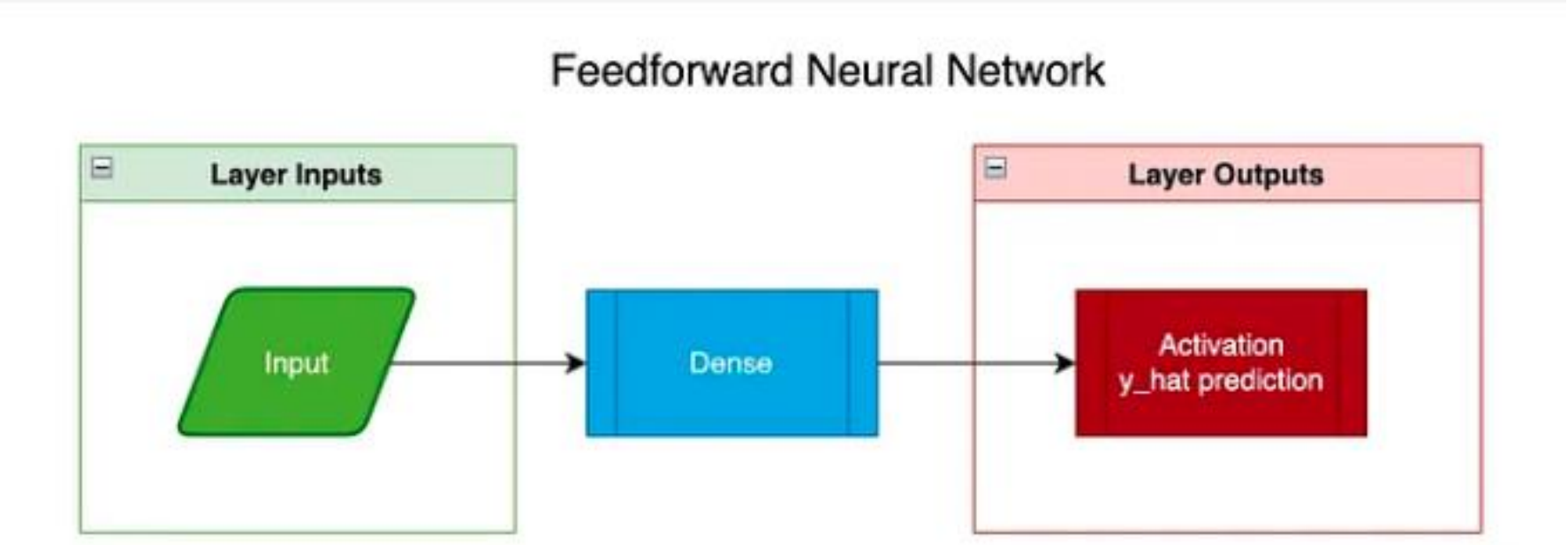
- Consider a simple recurrent neural network
- Assume the activation function is tanh
- The use of the hyperbolic tangent (tanh) function is a common activation function in recurrent neural networks (RNNs).
- It squashes its input values to the range of [-1, 1], providing a smooth transition between negative and positive values.

RNN cell

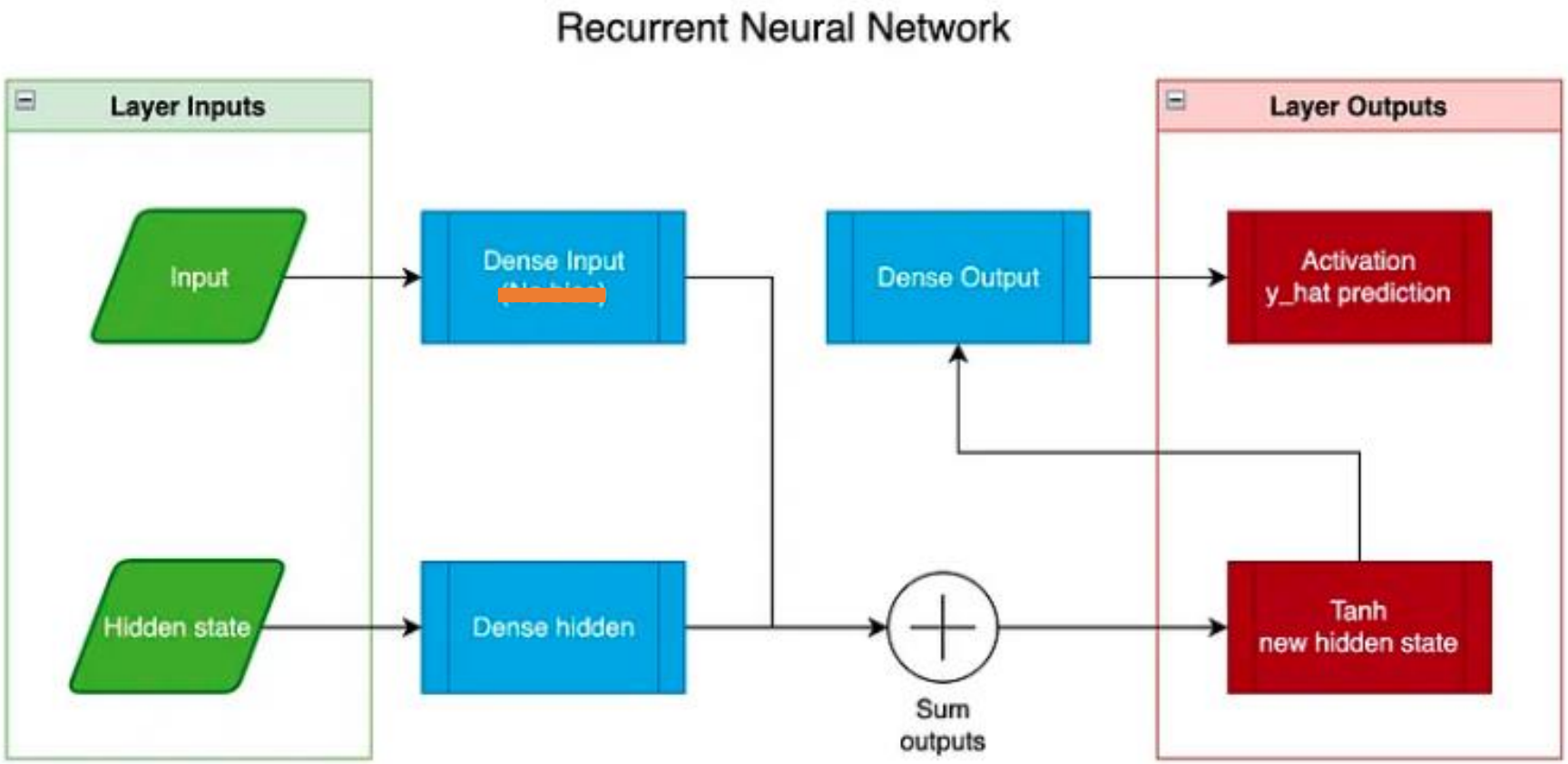
- Inside a single RNN layer we have 3 weight matrices as well as 2 input tensors and 2 output tensors.
- A single neuron in a feed forward network will take 1 input, 1 weight and 1 output

	Feedforward	RNN
Inputs	1	2
Weights	1	3
Outputs	1	2

Feed Forward neural network layer architecture



Recurrent neural network layer architecture



	Feedforward	RNN
Inputs	1	2
Weights	1	3
Outputs	1	2

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

Note: Bias is also applied

RNN Layer Architecture

- Recurrent Nets introduce a new concept called “hidden state”
- Hidden state is another input based on previous layer outputs.
- For the first layer, start it with zeros meaning that there will be no hidden layer before the hidden layer of the input
- RNNs are fed in a different way than feedforward networks.
- Because we are working with sequences, the order that we input the data matters
- So, each time, we feed the net, we have to input a single item in the sequence.
- Ex: For a stock price, input the stock price for each day. For a text, enter a single letter/word each time.
- So, enter one step at a time because we need to compute the hidden state on each iteration
- This hidden state will hold previous information so the next sequence we input will have data from previous runs by summing the matrices

RNN Layer Architecture - Inputs

1.Input tensor: This tensor should be 1 single step in the sequence.

Ex: If our total sequence is 100 characters of text, then the input will be 1 single character of text

2. Hidden state tensor: This tensor is the hidden state.

For the first run of each entire sequence, this tensor will be filled with zeros.

Ex: For 10 sequences of 100 characters each (a text of 1000 characters in total) then for each sequence we will generate a hidden state filled with zeros.

RNN Layer Architecture - Weight Matrices

1. Input Dense: Dense matrix used to compute inputs (just like feedforward).

2. Hidden Dense: Dense matrix used to compute hidden state input.

3. Output Dense: Dense matrix used to compute the result of activation(input_dense + hidden_dense)

RNN Layer Architecture - Outputs

1. New hidden state: New hidden state tensor which is activation(input_dense + hidden_dense). We will use this as input on the next iteration in the sequence.

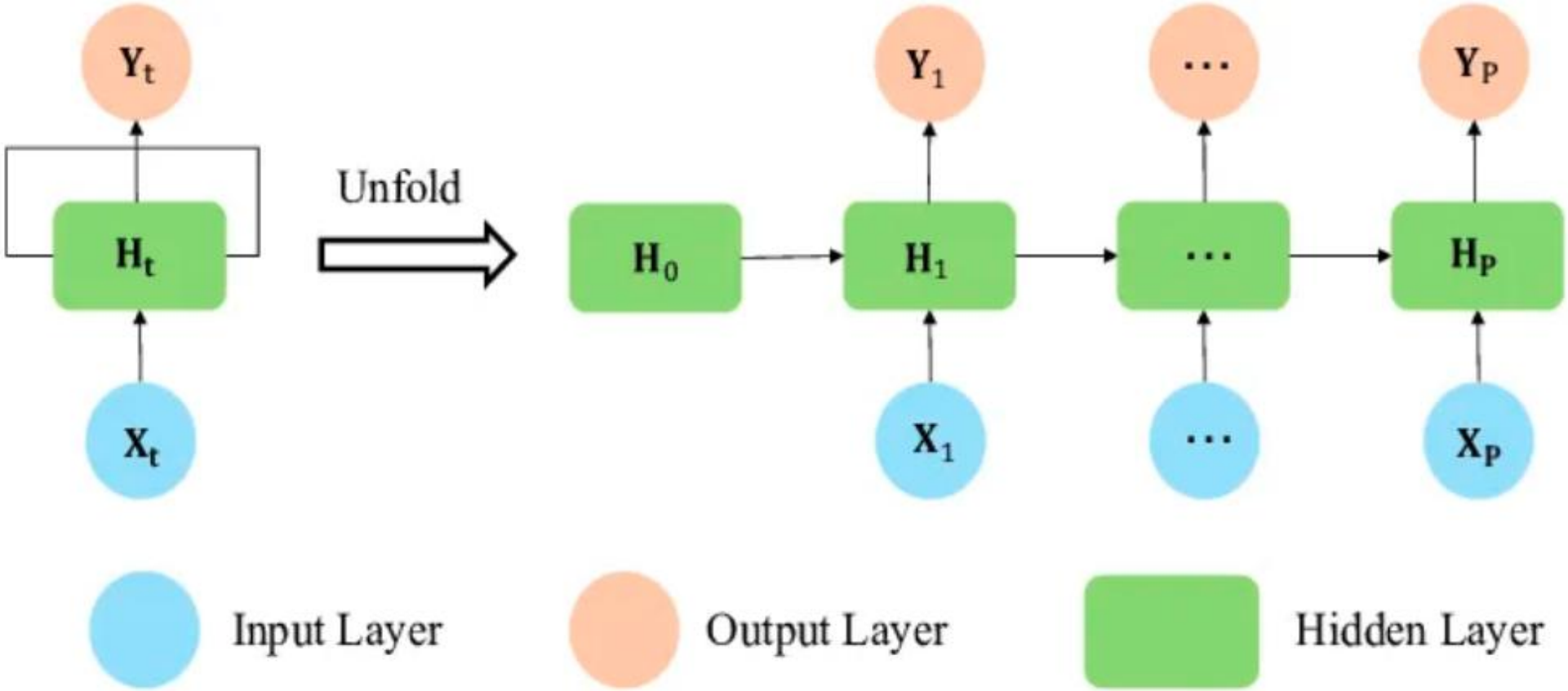
2. Output: activation(output_dense). This is our prediction vector. which means it is like the feedforward output prediction vector

Unrolled or Unfolded RNN

- We do not start thinking from scratch every second.
- We understand each word based on our understanding of previous words.
- We do not throw everything away and start thinking from scratch again. Our thoughts have persistence.

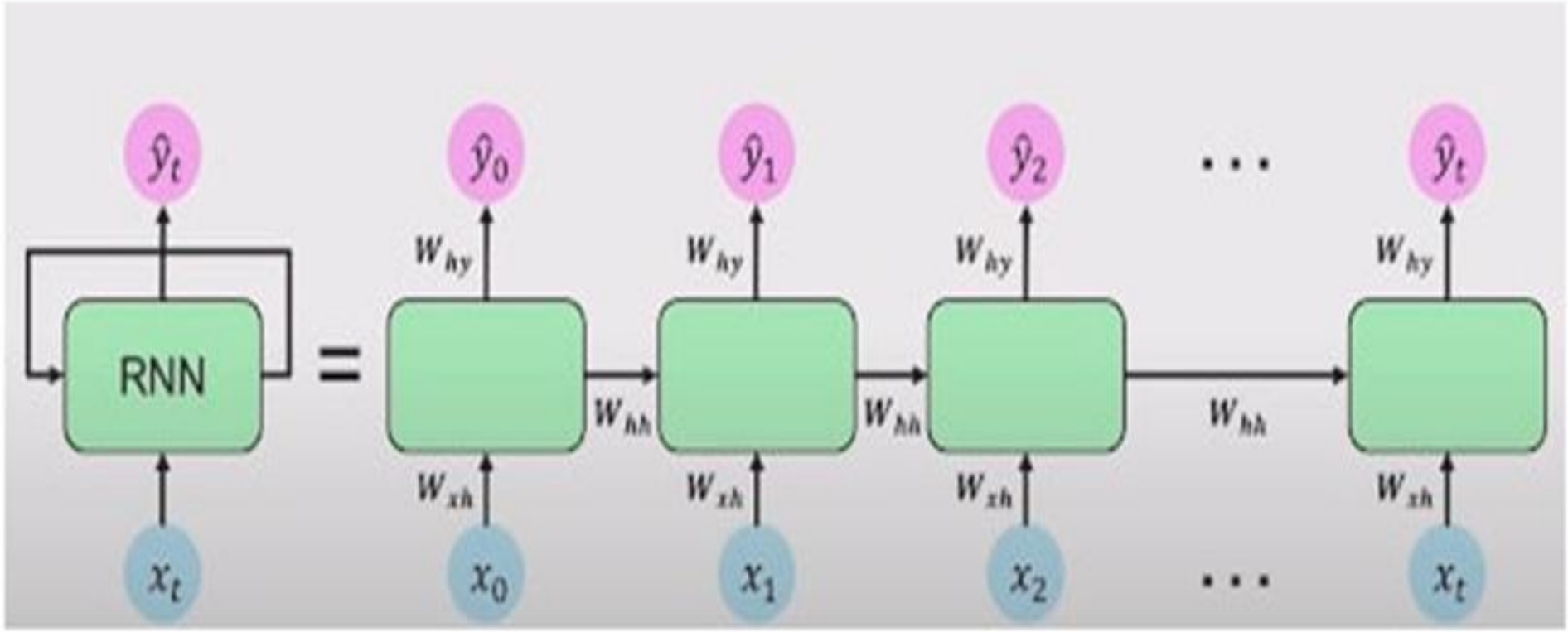
- Ex: To classify what kind of event is happening at every point in a movie.
- A traditional neural network cannot use its reasoning about previous events in the film to inform later ones.
- Recurrent neural networks address this issue. They are networks with loops in them, allowing information to persist.
- Sequence length specifies the number of steps to unroll the RNN
- Unfolding/parameter sharing is better than using different parameters per position: less parameters to estimate, generalize to various length.

Unrolled or Unfolded RNN



Note:RNNs represented as time unrolled version above is just a representation and not a transformation. The weight matrices and biases are not time dependent in the forward pass and remain the same in every time step.

Unrolled or Unfolded RNN



RNN Example 1 - Input

- Input Data: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20]
- We are looking at 5 (seq_len) previous value to predict the next 2 values.
- Divide it into 4 batches of sequence length = 5

```
data = torch.Tensor([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])

print("Data: ", data.shape, "\n\n", data)
```

```
Data:  torch.Size([20])

 tensor([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12., 13., 14.,
        15., 16., 17., 18., 19., 20.])
```

RNN Example 1 - Input

- Input
- torch.nn.RNN has two inputs - input and h_0 ie. the input sequence and the hidden-layer at t=0.
- input is the sequence which is fed into the network. It should be of size (seq_len(L), batch (N), input_size(H_{in}). If batch_first=True, the input size is (batch, seq_len, input_size). input shape: (N, L, H_{in})
- h_0 is the initial hidden state of the network.
- It is of the size (num_layers * num_directions (D), batch (N), hidden_size (H_{out})) where num_layers is the number of stacked RNNs. num_directions = 2 for bidirectional RNNs and 1 otherwise h_0 shape: (D * num_layers, N, H_{out})
- If we do not initialize the hidden layer, it will be auto-initialised by PyTorch to be all zeros.

RNN Example 1 - Output

- Output
- torch.nn.RNN has two outputs - out and hidden.
- out is the output of the RNN from all timesteps from the last RNN layer.
- It is of the size (seq_len, batch, num_directions * hidden_size).
- If batch_first=True, the output size is (batch, seq_len, num_directions * hidden_size). out shape: (N, L, D* H_{out})
- h_n is the hidden value from the last time-step of all RNN layers.
- It is of the size (num_layers * num_directions, batch, hidden_size). h_n is unaffected by batch_first=True h_n shape: (D * num_layers, N, H_{out})
- Note: In current RNN (including nn.RNN, nn.GRU, nn.LSTM) implementation, the batch_first will only affect the dimension order of input and output variables but not hidden
- The final hidden state corresponding to the representation of the full sequence and its shape follows the same rules as the initial hidden state

RNN Example 1 - Configuration

- # Number of features used as input. (Number of columns)
- INPUT_SIZE = 1
- # Number of previous time stamps taken into account.
- SEQ_LENGTH = 5
- # Number of features in last hidden state ie. number of output time steps to predict.
- HIDDEN_SIZE = 1
- # Number of stacked rnn layers.
- NUM_LAYERS = 1
- # We have total of 20 rows in our input.
- # We divide the input into 4 batches where each batch has only 1 row. Each row corresponds to a sequence of length 5.
- BATCH_SIZE = 4

RNN Example 1 – nn. RNN() & Weight Initialization

- # Initialize the RNN
- rnn = nn.RNN(input_size=INPUT_SIZE, hidden_size=HIDDEN_SIZE, num_layers = 1, batch_first=True)
- # Initialize the RNN and set the weights to 1 for verifying results
- rnn.weight_ih_l0.data.fill_(1)
- rnn.weight_hh_l0.data.fill_(1)
- rnn.bias_ih_l0.data.fill_(1)
- rnn.bias_hh_l0.data.fill_(1)

- Note: _l0 (level 0) suffix indicates that they belong to the first layer
- Weights and biases are tuned while learning

RNN Example 1 – nn. RNN() & Weight Initialization and state_dict

```
rnn = nn.RNN(input_size=INPUT_SIZE, hidden_size=HIDDEN_SIZE, num_layers = 1, batch_first=True)
rnn_state= rnn.state_dict()
print("Initial rnn state=\n", rnn_state)
# Initialize the RNN and set the weights to 1 for verifying results
rnn.weight_ih_l0.data.fill_(1)
rnn.weight_hh_l0.data.fill_(1)
rnn.bias_ih_l0.data.fill_(1)
rnn.bias_hh_l0.data.fill_(1)
rnn_state= rnn.state_dict()
print("After weight initialization, rnn state=\n", rnn_state)
```

RNN Example 1 – nn. RNN() & Weight Initialization and state_dict Output

```
Initial rnn state=
 OrderedDict([('weight_ih_l0', tensor([[0.2851]])), ('weight_hh_l0', tensor([[-0.6531]])), ('bias_ih_l0', tensor([0.8445])), ('bias_hh_l0', tensor([-0.2
546]))])
After weight initialization, rnn state=
 OrderedDict([('weight_ih_l0', tensor([[1.]])), ('weight_hh_l0', tensor([[1.]])), ('bias_ih_l0', tensor([1.])), ('bias_hh_l0', tensor([1.]))])
```

Note: _l0 (level 0) suffix indicates that they belong to the first layer
Ex: 'weight_ih_l0', tensor([[0.2851] in the output

RNN Example 1

```
# input size : (batch, seq_len, input_size)
inputs = data.view(BATCH_SIZE, SEQ_LENGTH, INPUT_SIZE)
# out shape = (batch, seq_len, num_directions * hidden_size)
# h_n shape  = (num_layers * num_directions, batch, hidden_size)
out, h_n = rnn(inputs)
print('Input: ', inputs.shape, '\n', inputs)
print('\nOutput: ', out.shape, '\n', out)
print('\nHidden: ', h_n.shape, '\n', h_n)
```

- input shape = [4, 5, 1]
- out shape = [4, 5, 1]
- h_n shape = [1, 4, 1]

RNN Example 1

```
import torch
import torch.nn as nn
data = torch.Tensor([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20])
print("Data: ", data.shape, "\n\n", data)
# Number of features used as input. (Number of columns)
INPUT_SIZE = 1
# Number of previous time stamps taken into account.
SEQ_LENGTH = 5
# Number of features in last hidden state ie. number of output timesteps to predict.
HIDDEN_SIZE = 1
# Number of stacked rnn layers.
NUM_LAYERS = 1
# We have total of 20 rows in our input.
# We divide the input into 4 batches
# Each row corresponds to a sequence of length 5.
# Each batch contains 5 rows because our SEQ_LENGTH = 5.
# We are using only a single feature as input INPUT_SIZE = 1
#BATCH_SIZE = 4
BATCH_SIZE = 4
```

RNN Example 1

```
rnn = nn.RNN(input_size=INPUT_SIZE, hidden_size=HIDDEN_SIZE, num_layers = 1, batch_first=True)
# Initialize the RNN and set the weights to 1 for verifying results
rnn.weight_ih_l0.data.fill_(1)
rnn.weight_hh_l0.data.fill_(1)
rnn.bias_ih_l0.data.fill_(1)
rnn.bias_hh_l0.data.fill_(1)
# input size : (batch, seq_len, input_size)
inputs = data.view(BATCH_SIZE, SEQ_LENGTH, INPUT_SIZE)
# out shape = (batch, seq_len, num_directions * hidden_size)
# h_n shape  = (num_layers * num_directions, batch, hidden_size)
out, h_n = rnn(inputs)
print('Input: ', inputs.shape, '\n', inputs)
print('\nOutput: ', out.shape, '\n', out)
print('\nHidden: ', h_n.shape, '\n', h_n)
```

RNN Example 1 - Output

```
Data:  torch.Size([20])

 tensor([ 1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10., 11., 12., 13., 14.,
        15., 16., 17., 18., 19., 20.])
Input:  torch.Size([4, 5, 1])
 tensor([[[ 1.],
          [ 2.],
          [ 3.],
          [ 4.],
          [ 5.]],

         [[ 6.],
          [ 7.],
          [ 8.],
          [ 9.],
          [10.]],

         [[11.],
          [12.],
          [13.],
          [14.],
          [15.]],

         [[16.],
          [17.],
          [18.],
          [19.],
          [20.]]])
```

```
Output:  torch.Size([4, 5, 1])
 tensor([[[0.9951],
          [0.9999],
          [1.0000],
          [1.0000],
          [1.0000]],

         [[1.0000],
          [1.0000],
          [1.0000],
          [1.0000],
          [1.0000]],

         [[1.0000],
          [1.0000],
          [1.0000],
          [1.0000],
          [1.0000]],

         [[1.0000],
          [1.0000],
          [1.0000],
          [1.0000],
          [1.0000]]], grad_fn=<TransposeBackward1>)

Hidden:  torch.Size([1, 4, 1])
 tensor([[[1.0000],
          [1.0000],
          [1.0000],
          [1.0000]]], grad_fn=<StackBackward0>)
```

Analyzing output

- In the output, the last row in each batch of out is present in h_n.
- For simple RNNs(without batch), the last element of the output is the final hidden state

Analyzing output for a single batch

```
data = torch.Tensor([1, 2, -1, 1])
print("Data: ", data.shape, "\n\n", data)
INPUT_SIZE = 1
SEQ_LENGTH = 4
HIDDEN_SIZE = 1
NUM_LAYERS = 1
BATCH_SIZE = 1
rnn = nn.RNN(input_size=INPUT_SIZE,
hidden_size=HIDDEN_SIZE, num_layers = 1,
batch_first=True)
```

```
Input:  torch.Size([1, 4, 1])
 tensor([[[ 1.],
         [ 2.],
         [-1.],
         [ 1.]]])
Output:  torch.Size([1, 4, 1])
 tensor([[[0.9951],
         [0.9999],
         [0.9640],
         [0.9993]]],
grad_fn=<TransposeBackward1>
)
Hidden:  torch.Size([1, 1, 1])
 tensor([[[0.9993]]],
grad_fn=<StackBackward0>)
```

Linear Activation function in RNN

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

- h t is the hidden state at time t,
- x t is the input at time t,
- h (t−1) is the hidden state of the previous layer at time t-1 or the initial hidden state at time 0.
- Nonlinearity used is tanh
- Note: RNNs by design are similar to deep neural networks. RNNs have input vectors, weight vectors, hidden states and output vectors.

.RNN() - Pytorch

- torch.nn.RNN(input_size, hidden_size, num_layers=1, nonlinearity='tanh', bias=True, batch_first=False, dropout=0.0, bidirectional=False, device=None, dtype=None)
  - **input_size** – The number of expected features in the input *x*
  - **hidden_size** – The number of features in the hidden state *h*
  - **num_layers** – Number of recurrent layers. E.g., setting `num_layers=2` would mean stacking two RNNs together to form a *stacked RNN*, with the second RNN taking in outputs of the first RNN and computing the final results. Default: 1
  - **nonlinearity** – The non-linearity to use. Can be either `'tanh'` or `'relu'`. Default: `'tanh'`
  - **bias** – If `False`, then the layer does not use bias weights *b_ih* and *b_hh*. Default: `True`
  - **batch_first** – If `True`, then the input and output tensors are provided as (*batch, seq, feature*) instead of (*seq, batch, feature*). Note that this does not apply to hidden or cell states. See the Inputs/Outputs sections below for details. Default: `False`
  - **dropout** – If non-zero, introduces a *Dropout* layer on the outputs of each RNN layer except the last layer, with dropout probability equal to `dropout`. Default: 0
  - **bidirectional** – If `True`, becomes a bidirectional RNN. Default: `False`

.RNN() – Pytorch – Sizes of input (x_t) & h_0

- Inputs: input, h_0
- **input**: tensor of shape (*L*, *Hin*) for unbatched input
- (L,N,H_{in}) when batch_first=False
- or (N,L, H_{in}) when batch_first=True containing the features of the input sequence.

- **h_0**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the initial hidden state for the input sequence batch. Defaults to zeros if not provided.

where:

$$
\begin{aligned}
N &= \text{batch size} \\
L &= \text{sequence length} \\
D &= 2 \text{ if bidirectional=True otherwise } 1 \\
H_{in} &= \text{input\_size} \\
H_{out} &= \text{hidden\_size}
\end{aligned}
$$

.RNN() – Pytorch Sizes of output & h_n

- Outputs: output, h_n
- output: tensor of shape
- (L, $D*H_{out}$) for unbatched input,
- (L, N, $D* H_{out}$) when batch_first=False or
- (N, L, $D*H_{out}$) when batch_first=True containing the output features (h_t) from the last layer of the RNN, for each t

  - **h_n**: tensor of shape $(D * \text{num\_layers}, H_{out})$ for unbatched input or $(D * \text{num\_layers}, N, H_{out})$ containing the final hidden state for each element in the batch.

Hidden to Output parameters (W_{ho} & b_{ho})

- In PyTorch, when we create an instance of nn.RNN
  - weights and biases for the input-to-hidden (*Wih*, *bih)* and
  - hidden-to-hidden layers *(Whh*, *bhh)* are set.

$$h_t = \tanh(x_t W_{ih}^T + b_{ih} + h_{t-1} W_{hh}^T + b_{hh})$$

- The nn.RNN module in PyTorch
  - does not have an explicit hidden-to-output weight matrix and bias parameter like a fully connected layer.
- Since we have one-to-many or many- to-many network, to produce an output O[t] at every timestep, we need another weight matrix that accepts a hidden state and project it to an output.
- Need to define an additional linear layer (nn.Linear) for this purpose.
- $O^_t = W_{ho} * h_t + b_{ho}$ --- Here h_t is from above equation
- The output of this linear layer is the final output of RNN model here.

.RNN() – Pytorch – Sizes of weights and biases

Variables

- **weight_ih_l[k]** – the learnable input-hidden weights of the k-th layer, of shape (*hidden_size, input_size*) for *k = 0*. Otherwise, the shape is (*hidden_size, num_directions * hidden_size*)
- **weight_hh_l[k]** – the learnable hidden-hidden weights of the k-th layer, of shape (*hidden_size, hidden_size*)
- **bias_ih_l[k]** – the learnable input-hidden bias of the k-th layer, of shape (*hidden_size*)
- **bias_hh_l[k]** – the learnable hidden-hidden bias of the k-th layer, of shape (*hidden_size*)

W_{ho} – the learnable hidden to output layer weights of kth layer of shape (out_features, hidden_size)
b_{ho} - the learnable hidden to output layer weights of kth layer of shape (out_features)

Note: For simple RNN having only one direction, num_directions=1. Hence consider weight_ih_l[k] is of size (hidden_size, input_size) for the sake of simplicity
The weight matrices and biases do not change with time unroll and the same weight matrices and biases are used in every time step.

Program -Hidden to Output parameters (W_{ho} & b_{ho})

```
import torch
import torch.nn as nn

class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, out_features):
        super(SimpleRNN, self).__init__()
        self.rnn = nn.RNN(input_size=input_size, hidden_size=hidden_size, num_layers=num_layers, batch_first=True)
        self.linear_layer = nn.Linear(in_features=hidden_size, out_features=out_features)

    def forward(self, x):
        # Forward pass through the RNN
        output, hidden_state = self.rnn(x)
        # Forward pass through the linear layer (hidden-to-output)
        hidden_to_output = self.linear_layer(hidden_state[-1])
        return output, hidden_state, hidden_to_output
```

Program -Hidden to Output parameters (W_{ho} & b_{ho})

```
# Example input tensor
Input_size = 1
Hidden_size = 1
Num_layers = 1
Batch_size = 1
Sequence_length = 4
Out_features = 2
x = torch.ones((Batch_size, Sequence_length, Input_size))
print("x=", x)
# Instantiate the SimpleRNN model
model = SimpleRNN(Input_size, Hidden_size, Num_layers, Out_features)
# Initialize weights and biases to 1 for both RNN and linear layer
for name, param in model.named_parameters():
    if 'weight' in name:
        nn.init.ones_(param.data)
    elif 'bias' in name:
        nn.init.ones_(param.data)
```

Program -Hidden to Output parameters (W_{ho} & b_{ho})

```
# Forward pass through the SimpleRNN model
output, hidden_state, hidden_to_output = model(x)
# Print the results
print("output=", output)
print("hidden_state=", hidden_state)
print("hidden_state[-1]=", hidden_state[-1])
print("Output from the hidden state to output layer:")
print(hidden_to_output)
# Print initialized weights and biases
for name, param in model.named_parameters():
    print(f"{name}: {param}")
total_params = sum(
            param.numel() for param in model.parameters() if param.requires_grad
)
print("total number of trainable parameters=", total_params)
```

Program -Hidden to Output parameters (W_{ho} & b_{ho})

x= tensor([[[1.],
[1.],
[1.],
[1.]]])
output= tensor([[[0.9951],
[0.9993],
[0.9993],
[0.9993]]],
grad_fn=<TransposeBackward1>)
hidden_state= tensor([[[0.9993]]],
grad_fn=<StackBackward0>)
hidden_state[-1]= tensor([[0.9993]],
grad_fn=<SelectBackward0>)
Output from the hidden state to output layer:
tensor([[1.9993, 1.9993]],
grad_fn=<AddmmBackward0>)

rnn.weight_ih_l0: Parameter containing:
tensor([[1.]], requires_grad=True)
rnn.weight_hh_l0: Parameter containing:
tensor([[1.]], requires_grad=True)
rnn.bias_ih_l0: Parameter containing:
tensor([1.], requires_grad=True)
rnn.bias_hh_l0: Parameter containing:
tensor([1.], requires_grad=True)
linear_layer.weight: Parameter containing:
tensor([[1.],
[1.]], requires_grad=True)
linear_layer.bias: Parameter containing:
tensor([1., 1.], requires_grad=True)
total number of trainable parameters= 8

Note: linear_layer(hidden_state[-1])=W·h+b
=linear_layer.weight * hidden_state[-1] + linear_layer.bias
= 1* 0.9993+1= 1.9993

Counting total number of parameters in RNN

```
total_params = sum(
        param.numel() for param in model.parameters() if param.requires_grad
)
print("total number of trainable parameters=", total_params)
```

- model.parameters(): PyTorch modules have a method called parameters() which returns an iterator over all the parameters.
- param.numel(): use the Iterator object returned by the model.parameters() and calculate the number of elements in it using the .numel() function
- sum(...): add up all the groups of parameters (a Module might contain submodules as layers)
- .requires_grad property of Tensor is used to determine if it is a trainable parameter

Counting total number of parameters in RNN

Character-Level Language Models

- Consider training of RNN character-level language models.
- Input the RNN with a text and ask it to model the probability distribution of the next character in the sequence given a sequence of previous characters.
- So, the problem is to generate new text one character at a time.
- As a working example, consider a vocabulary of four possible letters “helo”
- We have to train an RNN on the training sequence “hello”.
- This training sequence is a source of 4 separate training examples:
  - 1. The probability of “e” should be likely given the context of “h”,
  - 2. “l” should be likely in the context of “he”,
  - 3. “l” should also be likely given the context of “hel”, and finally
  - 4. “o” should be likely given the context of “hell”.

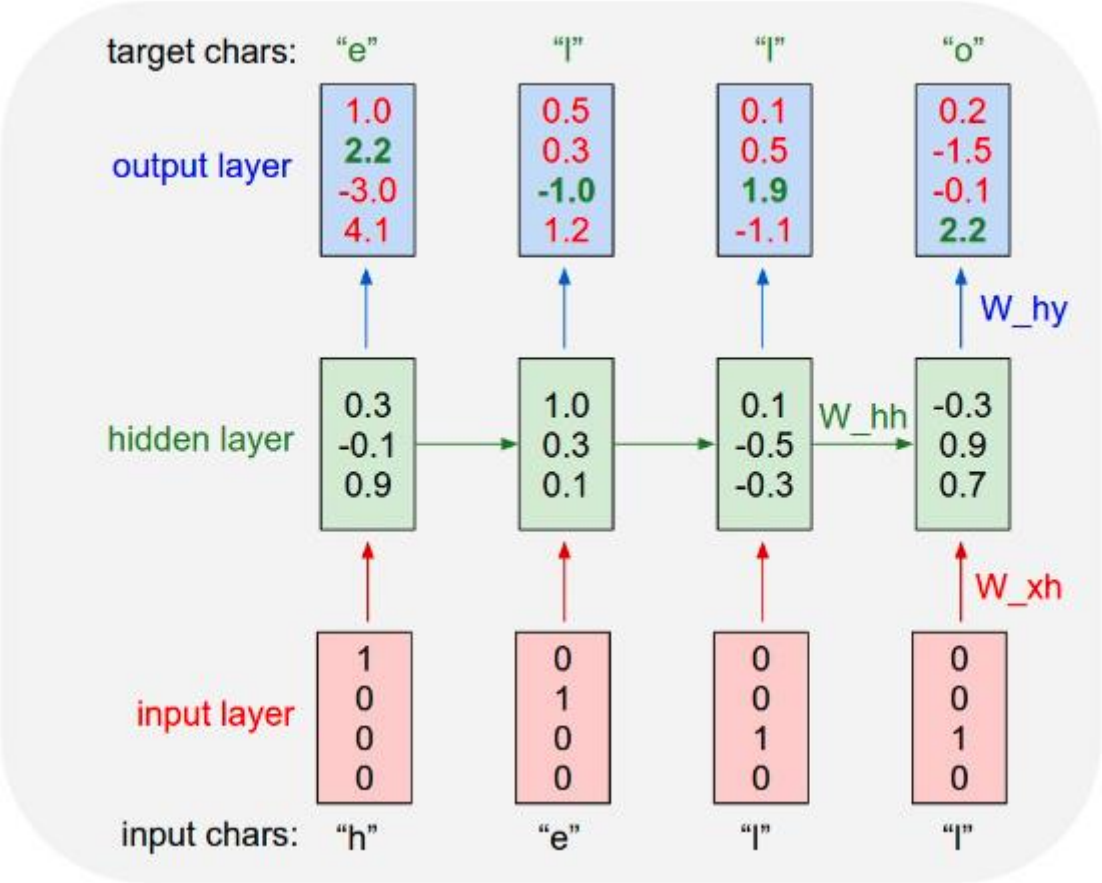
Counting total number of parameters in RNN

Character-Level Language Models

- Encode each character into a vector using 1-of-k encoding (i.e. all zero except for a single one at the index of the character in the vocabulary),
- Feed them into the RNN one at a time
- We will then observe a sequence of 4-dimensional output vectors (one dimension per character), which we interpret as the confidence the RNN currently assigns to each character coming next in the sequence.

Counting total number of parameters in RNN

Character-Level Language Models



- RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons).
- Shows the activations in the forward pass when the RNN is fed the characters "hell" as input.
- The output layer contains confidences the RNN assigns for the next character (vocabulary is "h,e,l,o");
- We want the green numbers to be high and red numbers to be low.

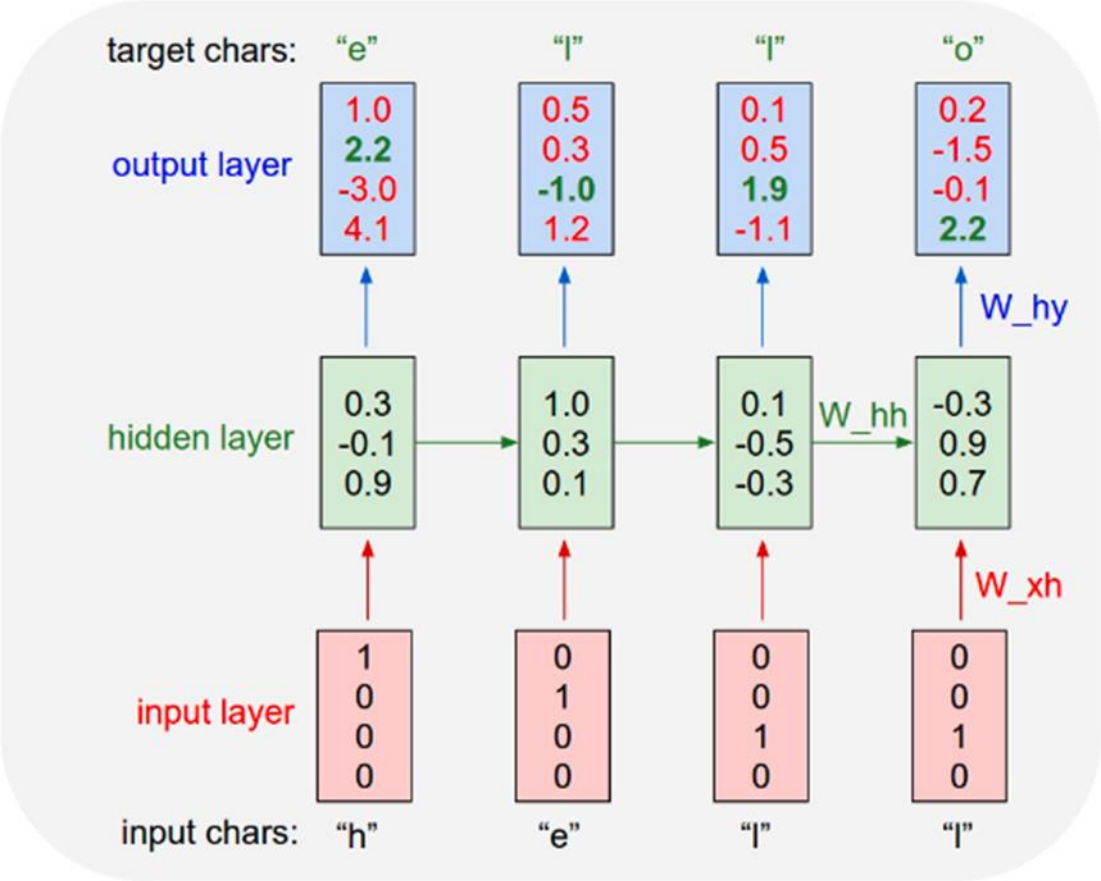
Counting total number of parameters in RNN

Character-Level Language Models

- In the first time step when the RNN saw the character “h” it assigned confidence of 1.0 to the next letter being “h”, 2.2 to letter “e”, -3.0 to “l”, and 4.1 to “o”.
- Since in our training data (the string “hello”) the next correct character is “e”, we would like to increase its confidence (green) and decrease the confidence of all other letters (red).
- Similarly, we have a desired target character at every one of the 4 time steps that we would like the network to assign a greater confidence to.
- Training loop: backpropagation algorithm is applied to figure out in what direction we should adjust every one of its weights to increase the scores of the correct targets (green bold numbers).
- Then parameter update is performed which updates every weight a tiny amount in this gradient direction.
- Feeding the same inputs to the RNN after the parameter update we would find that the scores of the correct characters (e.g. “e” in the first time step) would be slightly higher (e.g. 2.3 instead of 2.2), and the scores of incorrect characters would be slightly lower.
- Repeat this process many times until the network converges and its predictions are eventually consistent with the training data in that correct characters are always predicted next.

Counting total number of parameters in RNN

Character-Level Language Models



Count total number of parameters

RNN with 4-dimensional input and output layers, and a hidden layer of 3 units (neurons).

Computation in Recurrent neural networks

The computation in Recurrent neural networks

- can be decomposed into three blocks of parameters and associated transformations:

1. From the input to the hidden state – input layer
2. From the previous hidden state to the next hidden state – hidden layer
3. From the hidden state to the output – output layer

Counting total number of parameters in RNN

- Input size or in-features is H_{in} , hidden-size or hidden state features is H_{out} out-features - k
- W_{ih} shape is (hidden-size, input-size) – (H_{out} , H_{in}) and
- bias b_{ih} shape is (hidden-size) – (H_{out})
- W_{hh} shape is (hidden-size, hidden-size) – (H_{out} , H_{out}) and
- bias b_{hh} shape is (hidden-size) – (H_{out})
- W_{ho} shape is (out-features, hidden-size) - (k, H_{out}) and
- bias b_{ho} shape is (out-features) - k
- To get the total number of parameters in RNN, get the
  - number of parameters of the input layer, plus
  - number of parameters in the hidden layer, and
  - number of parameters in the output layer

Counting total number of parameters in RNN

- Input size or in-features is H_{in} , hidden-size or hidden state features is H_{out} out-features - k
- W_{ih} shape is (hidden-size, input-size) – (H_{out} , H_{in}) and bias b_{ih} shape is (hidden-size) – (H_{out})
- W_{hh} shape is (hidden-size, hidden-size) – (H_{out} , H_{out}) and bias b_{hh} shape is (hidden-size) – (H_{out})
- W_{ho} shape is (out-features, hidden-size) - (k, H_{out}) and bias b_{ho} shape is (out-features) - k
- In the input layer, the number of parameters comes from the size of the vocabulary, or input features- H_{in}.
- In the hidden layer, the number of parameters comes from the number of units of the layer, H_{out}
- Since the input is H_{in} and the hidden layer has H_{out} , the W params of the input layer W_{ih} would be (H_{out} , H_{in}) and bias b_{ih} shape is (hidden-size) – (H_{out})
- Since the network is recurrent, meaning it 'calls' itself over and over, then the W parameters in the hidden layer W_{hh} will be (hidden-size, hidden-size) – (H_{out} , H_{out})) and bias b_{hh} shape is (hidden-size) – (H_{out})

Counting total number of parameters in RNN

- Input size or in-features is H_{in} , hidden-size or hidden state features is H_{out} out-features - k
- W_{ih} shape is (hidden-size, input-size) – (H_{out} , H_{in}) and bias b_{ih} shape is (hidden-size) – (H_{out})
- W_{hh} shape is (hidden-size, hidden-size) – (H_{out} , H_{out}) and bias b_{hh} shape is (hidden-size) – (H_{out})
- W_{ho} shape is (out-features, hidden-size) - (k, H_{out}) and bias b_{ho} shape is (out-features) – (k)
- Output layer is linear layer with W_{ho} = (out-features, in-features) = (k, H_{out})and bias size for output layer is k

Total number of parameters =

Parameters in input layer + Parameters in hidden layer + Parameters in output layer

(H_{out} * H_{in}) + (H_{out}) + (H_{out} * H_{out}) + (H_{out}) + (k * H_{out}) + k

Counting total number of parameters in RNN

Character-Level Language Models

```
import torch
import torch.nn as nn

class SimpleRNN(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers, out_features):
        super(SimpleRNN, self).__init__()
        self.rnn = nn.RNN(input_size=input_size, hidden_size=hidden_size, num_layers=num_layers, batch_first=True)
        self.linear_layer = nn.Linear(in_features=hidden_size, out_features=out_features)

    def forward(self, x):
        # Forward pass through the RNN
        output, hidden_state = self.rnn(x)
        # Forward pass through the linear layer (hidden-to-output)
        hidden_to_output = self.linear_layer(hidden_state[-1])
        return output, hidden_state, hidden_to_output
```

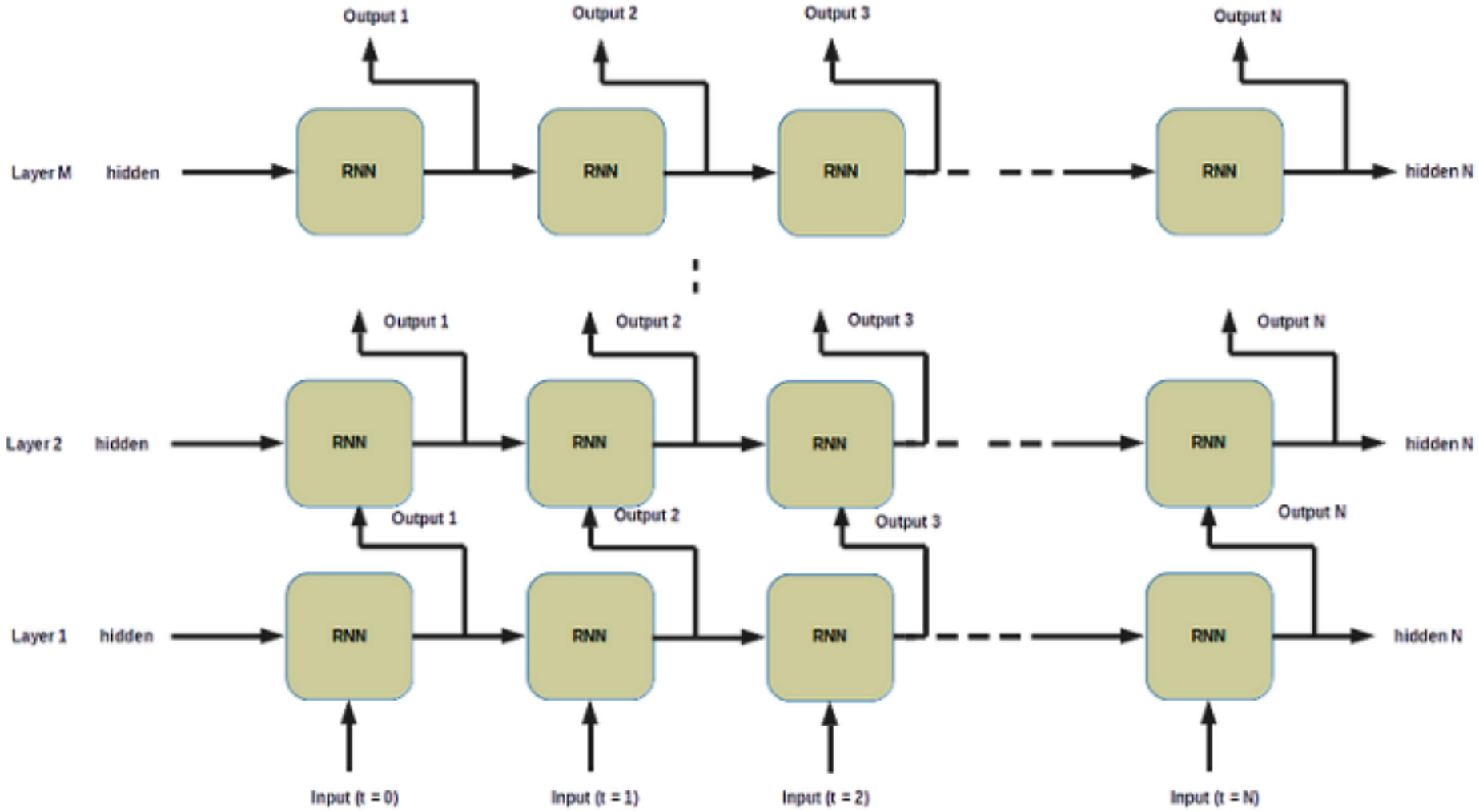
Counting total number of parameters in RNN

Character-Level Language Models

```
# Example input tensor
Input_size = 4
Hidden_size = 3
Num_layers = 1
Batch_size = 1
Sequence_length = 4
Out_features = 4
x = torch.ones((Batch_size, Sequence_length, Input_size))
print("x=", x)
# Instantiate the SimpleRNN model
model = SimpleRNN(Input_size, Hidden_size, Num_layers, Out_features)
# Initialize weights and biases to 1 for both RNN and linear layer
for name, param in model.named_parameters():
    if 'weight' in name:
        nn.init.ones_(param.data)
    elif 'bias' in name:
        nn.init.ones_(param.data)
```

```
x= tensor([[[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
output= tensor([[[1.0000, 1.0000, 1.0000],
         [1.0000, 1.0000, 1.0000],
         [1.0000, 1.0000, 1.0000],
         [1.0000, 1.0000, 1.0000]]], grad_fn=<TransposeBackward1>)
hidden_state= tensor([[[1.0000, 1.0000, 1.0000]]], grad_fn=<StackBackward0>)
hidden_state[-1]= tensor([[1.0000, 1.0000, 1.0000]], grad_fn=<SelectBackward0>)
Output from the hidden state to output layer:
tensor([[4.0000, 4.0000, 4.0000, 4.0000]], grad_fn=<AddmmBackward0>)
rnn.weight_ih_l0: Parameter containing:
tensor([[1., 1., 1., 1.],
        [1., 1., 1., 1.],
        [1., 1., 1., 1.]], requires_grad=True)
rnn.weight_hh_l0: Parameter containing:
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], requires_grad=True)
rnn.bias_ih_l0: Parameter containing:
tensor([1., 1., 1.], requires_grad=True)
rnn.bias_hh_l0: Parameter containing:
tensor([1., 1., 1.], requires_grad=True)
linear_layer.weight: Parameter containing:
tensor([[1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.],
        [1., 1., 1.]], requires_grad=True)
linear_layer.bias: Parameter containing:
tensor([1., 1., 1., 1.], requires_grad=True)
total number of trainable parameters= 43
```

Simple (Unidirectional) RNN

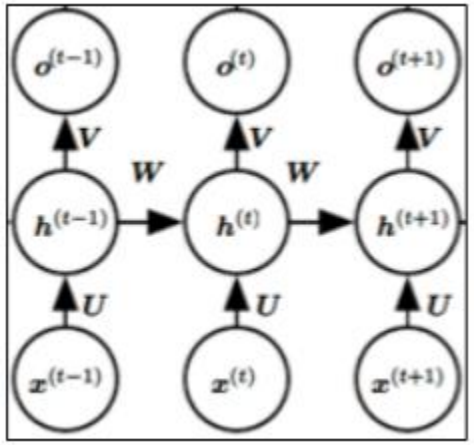


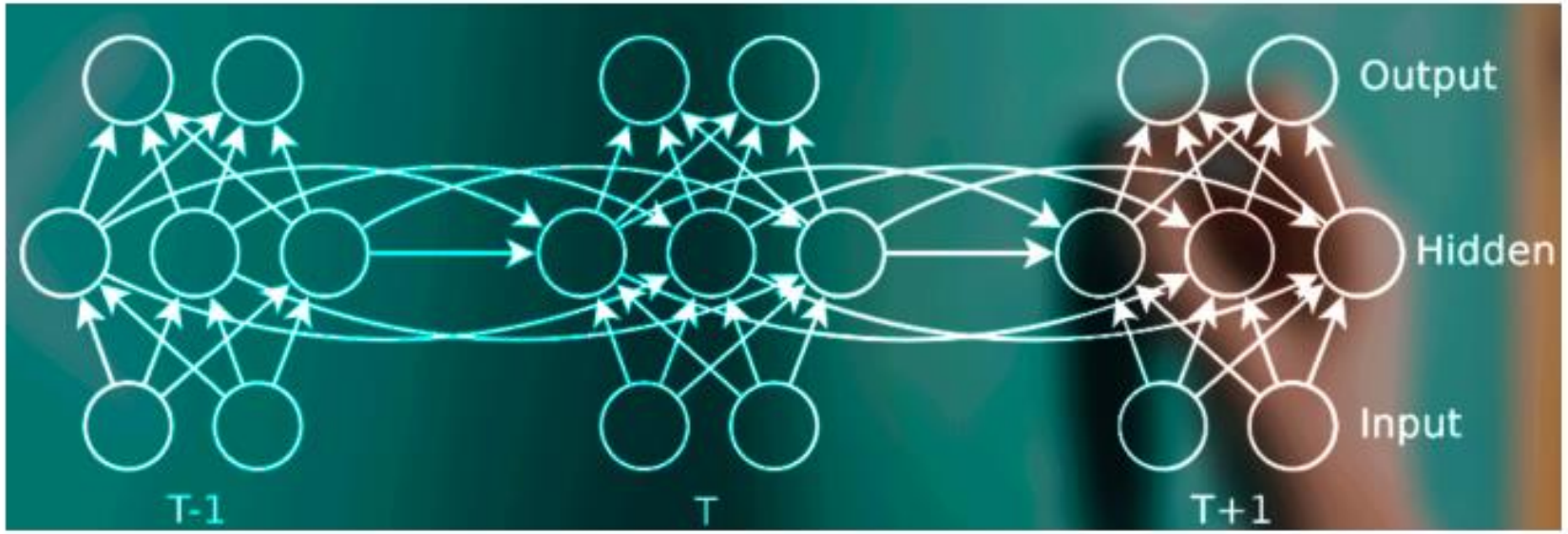
There are N time steps horizontally and
M layers vertically.

Feed input at t = 0 and initially hidden to RNN cell and the output to hidden

Feed to the same RNN cell with next input sequence at t = 1 and keep feeding the hidden output to the all input sequence.

A unfolded computational graph

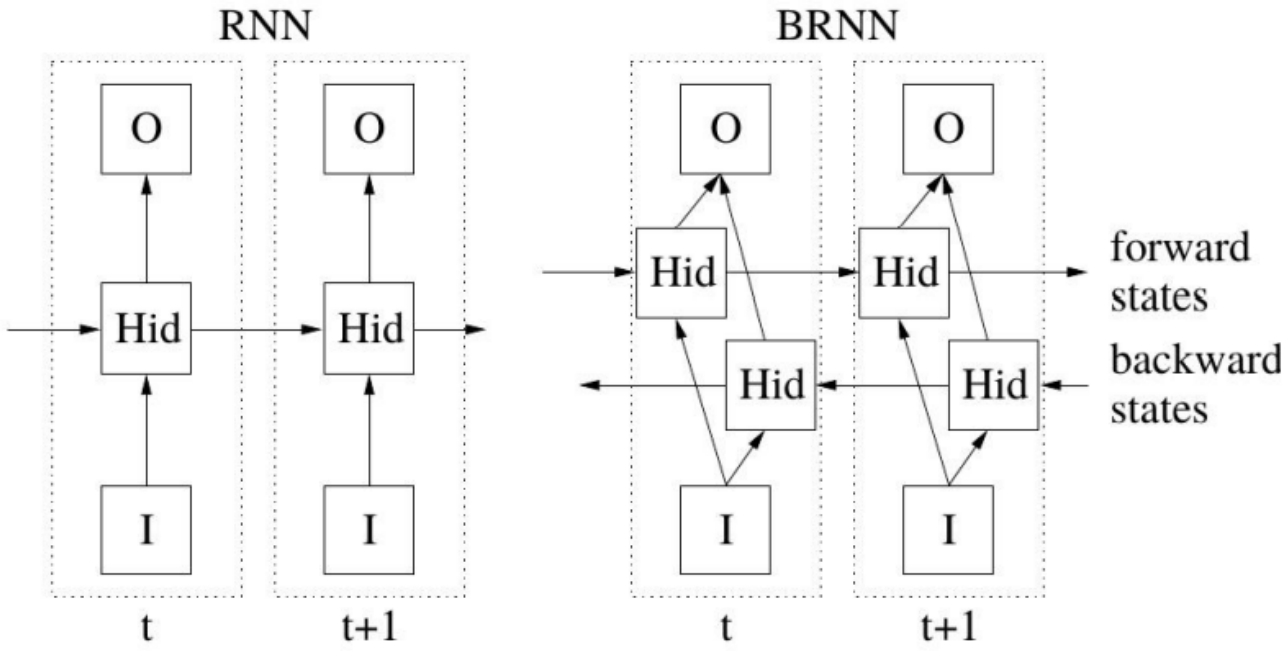




Two units in input layer, instead of 1
Three units in hidden layer, instead of 1
Two units in output layer, instead of 1

Need for bidirectionality

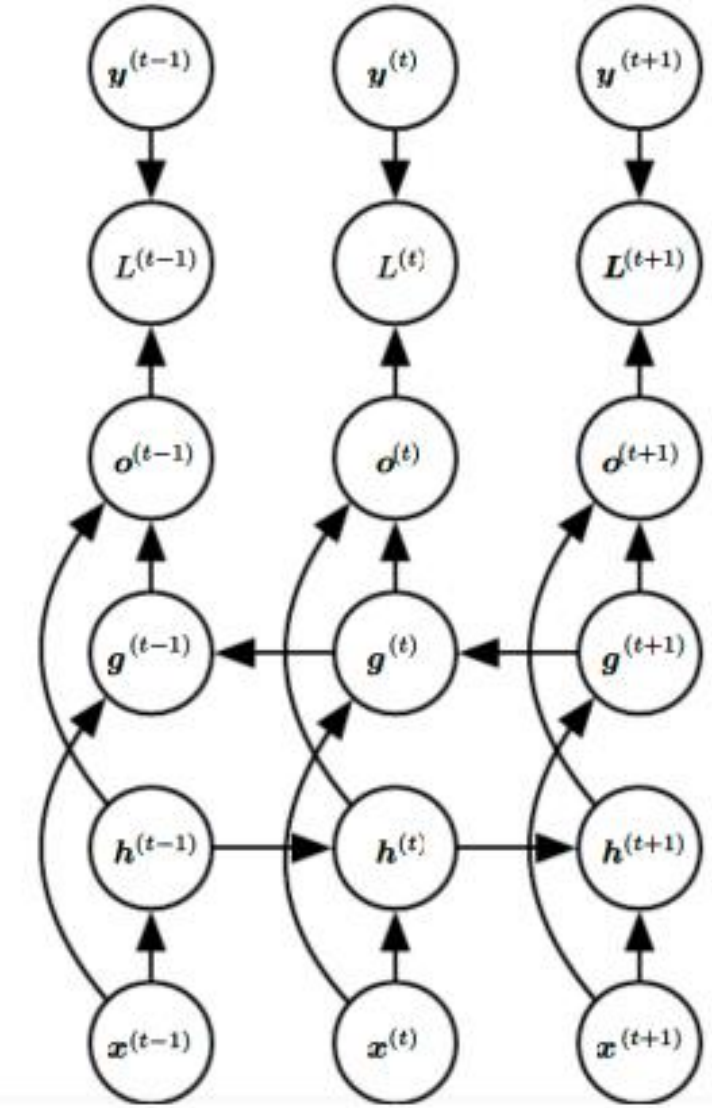
- In speech recognition, the correct interpretation of the current sound may depend on the next few phonemes because of coarticulation and the next few words because of linguistic dependencies
- Also true of handwriting recognition



A bidirectional RNN

- A bidirectional RNN
- Combine an RNN that moves forward through time from the start of the sequence
- Another RNN that moves backward through time beginning from the end of the sequence
- A bidirectional RNN consists of two RNNs which are stacked on the top of each other.
    - The one that processes the input in its original order and the one that processes the reversed input sequence.
    - The output is then computed based on the hidden state of both RNNs.

A bidirectional RNN

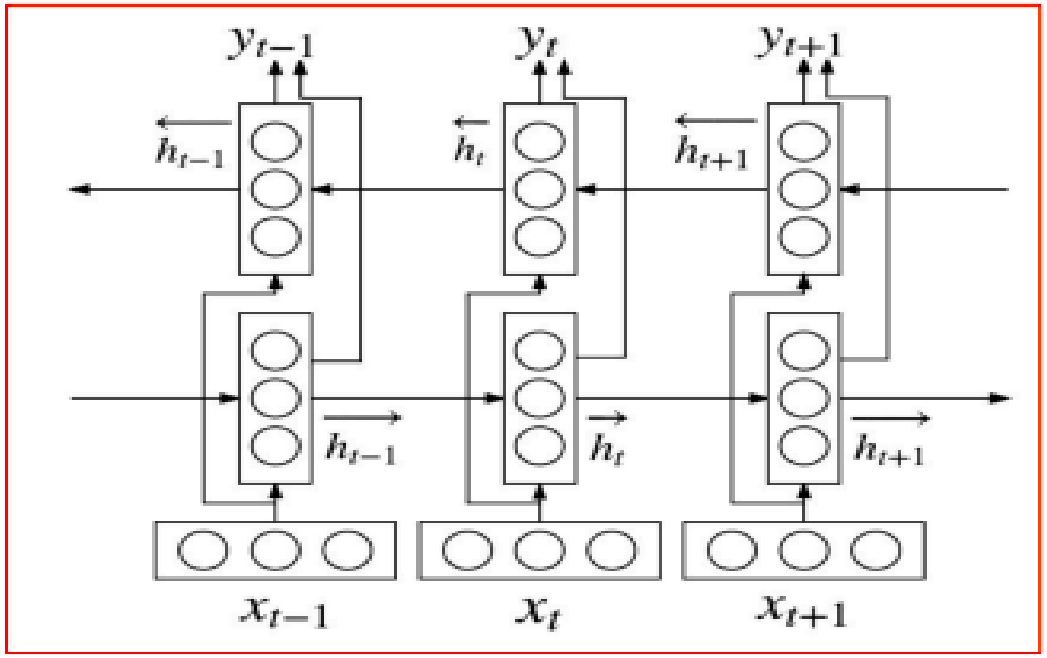


Maps input sequences x to target sequences y with loss L(t) at each step t

- h Recurrence propagates to the right
- g recurrence propagates to the left.
- This allows output units o(t) to compute a representation that depends both the past and the future

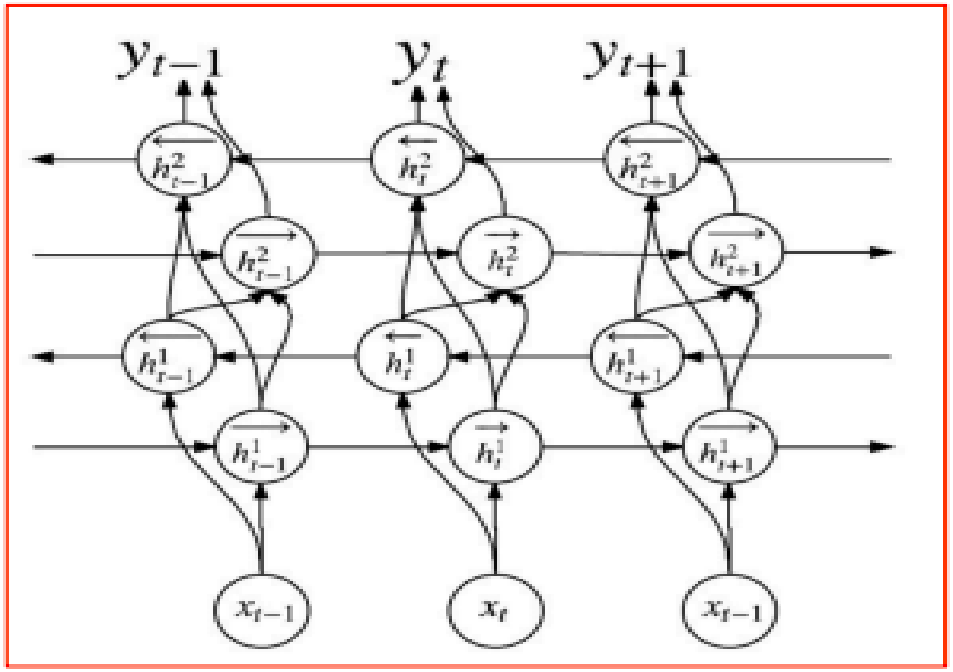
Parameters of a Bidirectional RNN

Bidirectional RNN



Deep Bidirectional RNN

Has multiple layers per time step



Vanishing Gradient & Exploding Gradient problem

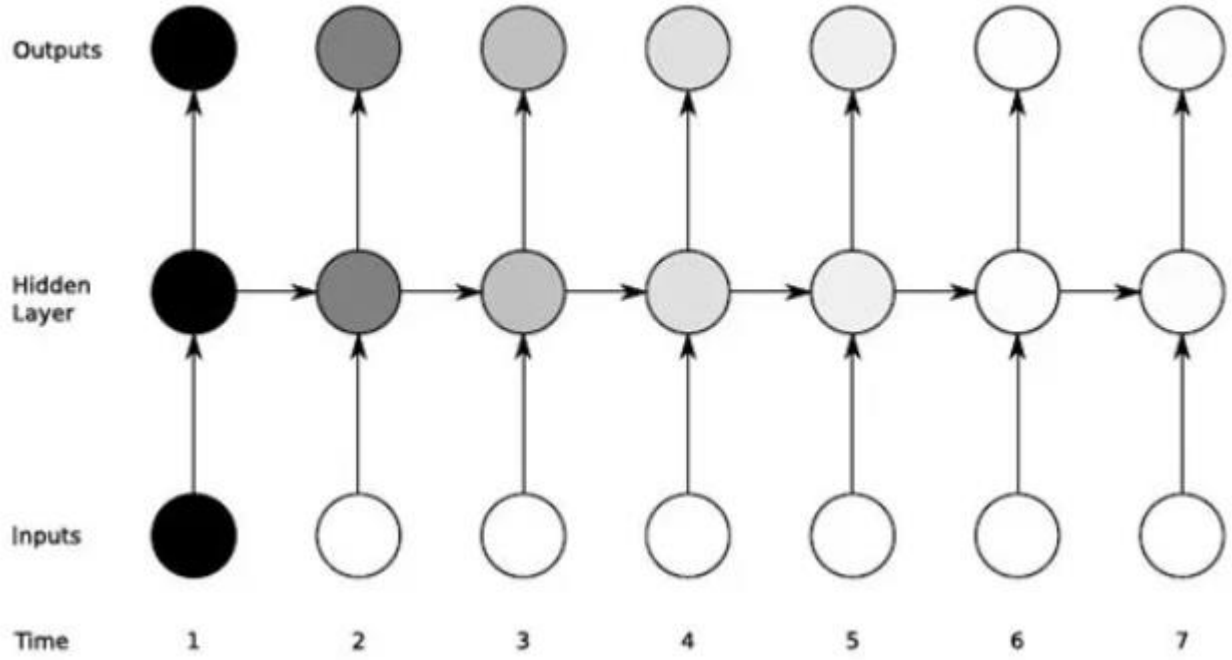
- *H' — Previous output to be fed in the current input*
- H' < 1 — When we multiply with <1, we get even a smaller number. As we keep moving backwards in the layer while back propagating, the value will keep on decreasing so much that it will almost tend to zero. Without any deltas, we are unable to change w in the backward layers. And thus tuning it will become a major problem. This is called Vanishing Gradient problem.
- H' > 1 — When we multiply with >1, we get even a larger number. As we keep moving backwards in the layer while back propagating, the value will keep on increasing so much that it will almost tend to infinite. Because of which we will not be able to reach local minima of errors and will keep on jumping in gradient descent graph. This is called Exploding Gradient problem. While training, if the slope tends to grow exponentially rather than decaying, we get an Exploding Gradient.
- Note: Typical feed-forward neural nets can cope with these exponential effects because they only have a few hidden layers.
- In an RNN trained on long sequences (e.g. 100 time steps) the gradients can easily explode or vanish
- Suppose a computational graph having repeated multiplication by a matrix W
- After t steps this is equivalent to multiplying by W^t and multiplying by itself many times, the product W^t will either vanish or explode depending on the magnitude of w

Vanishing Gradient illustration

- RNNs are unable to work with sequences that are very long (long sentences or long speeches).
- The effect of a given input on the hidden layer (and thus the output) decays exponentially as a function of time (or sequence length).
- RNNs suffer from insensitivity to input for long sequences (sequence length approximately greater than 10 time steps).
- The problem with RNNs is that as time passes by and they get fed more and more new data, RNNs “forget” about the previous data they have seen
- As the previous data gets diluted between the new data, the transformation from activation function, and the weight multiplication.
- This means they have a good short term memory, but a problem when trying to remember things that have seen many time steps in the past.

Vanishing Gradient illustration – Sensitivity of hidden nodes to the gradient

- The shades of the nodes indicate the sensitivity of the network nodes to the input at a given time.
- Darker the shade the greater is the sensitivity and vice-versa.
- The sensitivity decays as we move from timestep =1 to timestep=7 very fast.
- The network forgets the first input (so cannot memorize long sequences).



Vanishing Gradient illustration

- RNNs tend to struggle with problem domains where long range dependencies are critical.
- Ex1: An example of a long range dependency could be generating a paragraph where the subject is identified to be female. Several sentences later, the model needs to remember this so it will use the correct pronoun ‘she’ when referring to the subject.
- These models struggle because they are very deep neural networks.
- As back-propagation works by updating parameters iteratively from the last layer, it becomes increasingly difficult.
- This issue is commonly known as the vanishing/exploding gradient problem

Vanishing gradient problem

- The vanishing and exploding gradient problems imply that a basic RRN can exhibit only short-term memory-based behaviors
- Therefore RNN cannot determine very long product chains during BackPropagation Through Time (BPTT).
- The problem of the exploding gradient can be resolved using the technique of the gradient clipping at a predefined threshold value.
- The problem of the vanishing gradient is more difficult resolution.
- LSTMs offered a solution to this problem by designing a more complex cell that would be passed additional inputs allowing long range dependencies to be preserved.

RNN- Advantages

- RNNs have various advantages, such as:
  - Ability to handle sequence data
  - Ability to handle inputs of varying lengths
  - Ability to store or “memorize” historical information
- The disadvantages are:
  - The computation can be very slow.
  - The network does not take into account long-term dependencies to make decisions.
  - Vanishing gradient problem, where the gradients used to compute the weight update may get very close to zero, preventing the network from learning new weights. The deeper the network, the more pronounced this problem is.

Different RNN Architectures

1. Bidirectional Recurrent Neural Networks (BRNN)

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

2. 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.

3. Long Short Term Memory (LSTM)

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