

Deep Learning for Computer Vision
Professor Vineeth n Balasubramanian
Department of Computer Science and Engineering
Indian Institute of Technology, Hyderabad
Lecture 53
Recurrent Neural Networks: Introduction

(Refer Slide Time: 00:16)

The slide features the IIT Hyderabad logo on the left and the NPTEL logo below it. The main title 'Recurrent Neural Networks: An Introduction' is centered in a red box. Below the title, the presenter's name 'Vineeth N Balasubramanian' and affiliation 'Department of Computer Science and Engineering, Indian Institute of Technology, Hyderabad' are listed. A small IIT Hyderabad logo is at the bottom center. A video inset at the bottom left shows the presenter speaking. A progress bar at the bottom indicates 'Vineeth N B. (IIT-H)', '58.1 Introduction to RNNs', and '1 / 25'.

So far, we have seen how CNNs, Convolution Neural Networks work, how back propagation is used to train them; how they can be used for image classification; what loss functions we use; how do you visualize and understand CNNs. We have also seen how CNNs can be extended for other tasks such as detection, segmentation, face recognition, as well as miscellaneous other tasks. This week, we will move on to another pillar of the space of deep learning which is recurrent neural networks.

Recurrent neural networks are intended to process time sequences, which means in terms of applying it to computer vision, they become relevant when you have videos as input.

(Refer Slide Time: 01:17)



Review: Questions

How to find bias in a model?

Change a particular attribute/feature in question, and see if the prediction changes!



Vineeth N B (IIT-H)

§8.1 Introduction to RNNs

2 / 25

Before we get started on the topic, we had one question that we left behind from last week, when we completed our lectures on CNNs for human understanding. And the question, an important one in today's times was; how do you find bias in a model?

If you had a face recognition model, how do you know if the model is biased to a certain segment of population or the dataset? Hope you had a chance to think about it. This is an important problem in today's times, when such technologies are being deployed at a fairly large scale, the simple check is if you think there could be a particular feature or an attribute such as say, the background of a particular person, then you could change that attribute in your input and see if the prediction changes.

If this happens, the model is relying on that feature to make its prediction. And if you think that should not exist, you should try to regularize based on that attribute, use data augmentation methods to vary that attribute's value and ensure the model does not change its prediction.

(Refer Slide Time: 02:54)



Acknowledgements

- This lecture's slides are based on:
 - **Lecture 10** of Stanford's **CS231n** course Fei-Fei Li
 - **Lecture 13** of IIT Madras' **CS7015** course by Mitesh Khapra



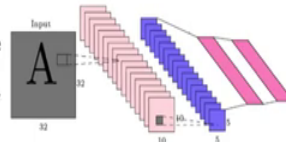
We will now move on to recurrent neural networks. This lecture's slides are based on the CS231n course by Fei-Fei Li and the course from IIT Madras by Mitesh Khapra.

(Refer Slide Time: 03:08)



Sequence Learning Problems

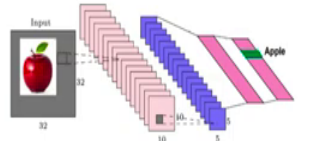
- In feedforward and convolutional neural networks, size of the input was always fixed
- E.g., we fed fixed size (32×32) images to convolutional neural networks for image classification
- Each input to network was independent of previous or future inputs





Sequence Learning Problems

- In feedforward and convolutional neural networks, size of the input was always fixed
- E.g., we fed fixed size (32×32) images to convolutional neural networks for image classification
- Each input to network was independent of previous or future inputs
- Computations, outputs and decisions for two successive images are completely independent of each other



Vineeth N B (IIT-H)

§8.1 Introduction to RNNs

4 / 25

The broader context of recurrent neural networks is sequence learning problems. So far, with both feed forward and convolutional neural networks, the size of the input was always fixed. Given a CNN, you always fed say a 32×32 image for image classification. Each input was also independent of the previous or future inputs and the computations, outputs, decisions for two successive images were completely independent of each other.

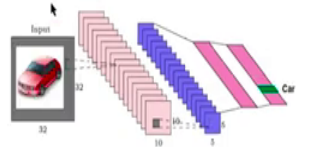
What do we mean? This also relates to the common property called IID which we associate with datasets in machine learning. IID stands for Identically Independently Distributed, if we have a data set of images, we assume that each image in the data set is identically independently distributed, which means that irrespective of which order you provide these images in, your model is likely to learn the same prediction function at the end.

(Refer Slide Time: 04:41)



Sequence Learning Problems

- In feedforward and convolutional neural networks, size of the input was always fixed
- E.g., we fed fixed size (32×32) images to convolutional neural networks for image classification
- Each input to network was independent of previous or future inputs
- Computations, outputs and decisions for two successive images are completely independent of each other



Vineeth N B (IIT-H)

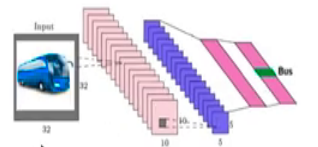
§8.1 Introduction to RNNs

4 / 25



Sequence Learning Problems

- In feedforward and convolutional neural networks, size of the input was always fixed
- E.g., we fed fixed size (32×32) images to convolutional neural networks for image classification
- Each input to network was independent of previous or future inputs
- Computations, outputs and decisions for two successive images are completely independent of each other



Vineeth N B (IIT-H)

§8.1 Introduction to RNNs

4 / 25

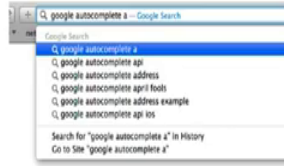
So irrespective of the input being say, this image or this image or this image, obviously the class label changes, but nothing else changes in the model parameters or the architecture.

(Refer Slide Time: 04:56)



Sequence Learning Problems

- Consider task of text auto completion



Credit: John Johnston



Vineeth N B (IIT-H)

§8.1 Introduction to RNNs

5 / 25



Sequence Learning Problems

- Consider task of text auto completion



Credit: John Johnston



Vineeth N B (IIT-H)

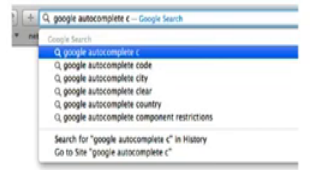
§8.1 Introduction to RNNs

5 / 25



Sequence Learning Problems

- Consider task of text auto completion
- Successive inputs are no longer independent!
- Length of inputs and number of predictions you need to make are not fixed
- Underlying model is performing same task across all contexts (*input*: character, *output*: character)
- Known as **sequence learning problems**



Credit: John Johnston



Vineeth N B (IIT-H)

58.1 Introduction to RNNs

5 / 25

However, there are another category of problems known as sequence learning problems, which requires something more than what we have seen so far. If you take this example of text auto completion, something that you perhaps encounter on a daily basis, see, if we said Google autocomplete a, you have a certain set of recommendations.

If you say, Google autocomplete b, you have a certain set of recommendations and similarly for c. In such problems, successive inputs are no longer independent. To make a prediction at a particular state, you do need to look at the inputs at previous time steps. This is something that we did not need so far when we talked about feed forward networks or CNNs. And in this kind of problems, the length of the inputs and number of predictions you need to make may also not be fixed.

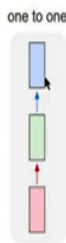
So an example here is if you say Google autocomplete c, the last word could be code or could be country, these have different lengths, and you want your model to be independent of such constraints. Another important takeaway in such problems is, irrespective of what you may have as the input, whether you had 2 words before the c or 3 words before the c, or 1 word before the c, the model has to perform the same task across all of these contexts.

Given a previous fair phrase, it has to now predict the next set of characters, that task does not change irrespective of at what time step you are trying to make the prediction. If we use a typed one word, you still have to predict the next set of characters. If the user typed 2 words, your

context increases, the temporal context increases in length, but what the task has to achieve remains the same. These kinds of problems are known as sequence learning problems.

(Refer Slide Time: 07:24)

Recurrent Neural Networks: Variants



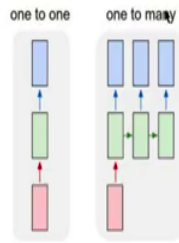
one to one

Vanilla Neural Networks

Vineeth N B. (IIT-H) 58.1 Introduction to RNNs 6 / 25

The diagram shows a vertical stack of three colored boxes: a red box at the bottom, a green box in the middle, and a blue box at the top. A red arrow points from the red box to the green box, and a blue arrow points from the green box to the blue box. The text 'one to one' is positioned above the stack. An arrow points from the text 'Vanilla Neural Networks' to the stack. The slide includes the NPTEL logo and a video feed of the presenter.

Recurrent Neural Networks: Variants



one to one one to many

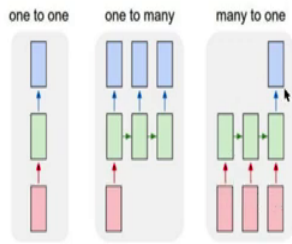
e.g. Image Captioning
image -> sequence of words

Vineeth N B. (IIT-H) 58.1 Introduction to RNNs 6 / 25

The diagram compares two architectures. On the left, 'one to one', shows a red box at the bottom, a green box in the middle, and a blue box at the top, with arrows pointing upwards. On the right, 'one to many', shows a red box at the bottom, a green box in the middle, and three blue boxes at the top. Arrows point from the red box to the green box, and from the green box to each of the three blue boxes. The text 'e.g. Image Captioning' and 'image -> sequence of words' is below the diagram. The slide includes the NPTEL logo and a video feed of the presenter.



Recurrent Neural Networks: Variants



e.g. **action prediction**
sequence of video frames -> action class



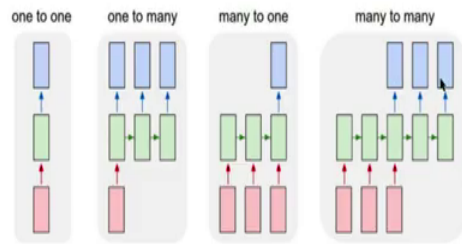
Vineeth N B. (IIT-H)

§8.1 Introduction to RNNs

6 / 25



Recurrent Neural Networks: Variants



E.g. **Video Captioning**
Sequence of video frames -> caption



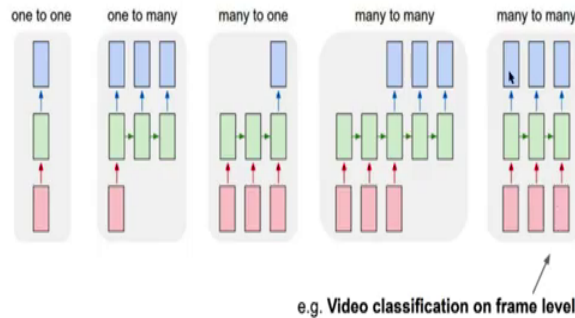
Vineeth N B. (IIT-H)

§8.1 Introduction to RNNs

6 / 25



Recurrent Neural Networks: Variants



Vineeth N B. (IIT-H)

§8.1 Introduction to RNNs

6 / 25

The neural network architecture that is designed to handle such sequence learning problems are known as recurrent neural networks. And we will see why they are called so soon. If you visualized a standard feed forward vanilla neural network as this kind of visualization where you have an input, say set of hidden layers and an output, then the variance of recurrent neural networks can be visualized as you could have a one to many variant where your input is at one time step but your output is actually a sequence of outputs over time steps.

What is an example? Image captioning, where your input is a single image, and your output is a sequence of words where one word has to follow the previous word to make a complete sentence. You call this one to many where you have one input and many outputs. Similarly, you could have a many to one architecture, where you have inputs at multiple time steps, but you make a prediction at one particular time step.



What is an example here? Action prediction, where you have a sequence of video frames as input, and the model has to give one particular class after seeing the full input sequence. For example, after seeing a complete video of say, a football game, maybe this particular outcome should predict whether that snippet of the video was a goal scoring event or not a goal scoring event. There is also a scenario of a many to many setting where given a set of multiple inputs over time steps, the output also has to have multiple outputs over time steps.

This is called many to many. And an example would be video captioning, where the input is a video, which is a set of frames over time and the output, a set of words over time again. You

could also have a slightly varied many to many settings, where you have a set of inputs over time, a set of outputs over time again. But the outputs and the inputs are synchronized at each input.


An example here could be video classification at frame level. So you have a video coming in as input to your model which means, remember video has a temporal component to it. But at the same time, you want to make predictions at the level of every frame. That would be a many to many setting in this particular context.

(Refer Slide Time: 10:30)



How do we model such tasks involving sequences?

- Account for dependence between inputs
- Account for variable number of inputs
- Make sure that function executed at each time step is the same. Why?



Vineeth N B. (IIT-H) 5.1 Introduction to RNNs 7 / 25

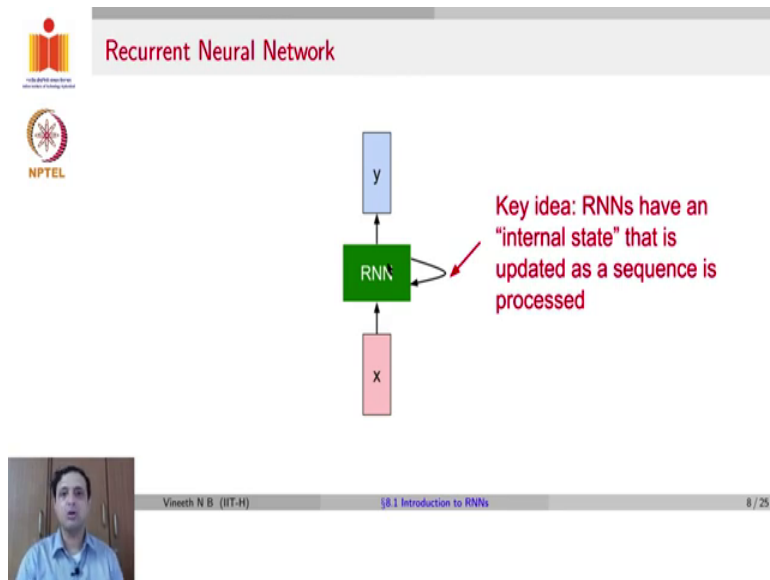
So how do we really model such tasks involve sequences, we saw high level architectures, but how do you actually learn the models with these architectures? How do we account for dependence between inputs which is common in time series data? How do you account for variable number of inputs? How do you ensure that, for example, if you take image captioning, it could be variable number of inputs or variable number of outputs, if you took image captioning, given an image, you could have a caption which has 7 words, given another image, you could have a caption, which is 10 words, how do you allow multiple outputs variable number of outputs, or similarly, variable number of inputs for another application.

A third consideration is how do you also ensure that the function executed at each time step is the same? Why do we need this? Irrespective of where a frame was in a video, you want to have the same logic to be able to execute your prediction. This is similar to what we said in

convolution, where we said that irrespective of whether a cat was on the top left, or the bottom right, we want to say that the cat exists in an image.

We overcame that problem through the idea of convolution. But how do we do that now over time series, and sequence learning problems is what we are going to talk about over the next few slides.

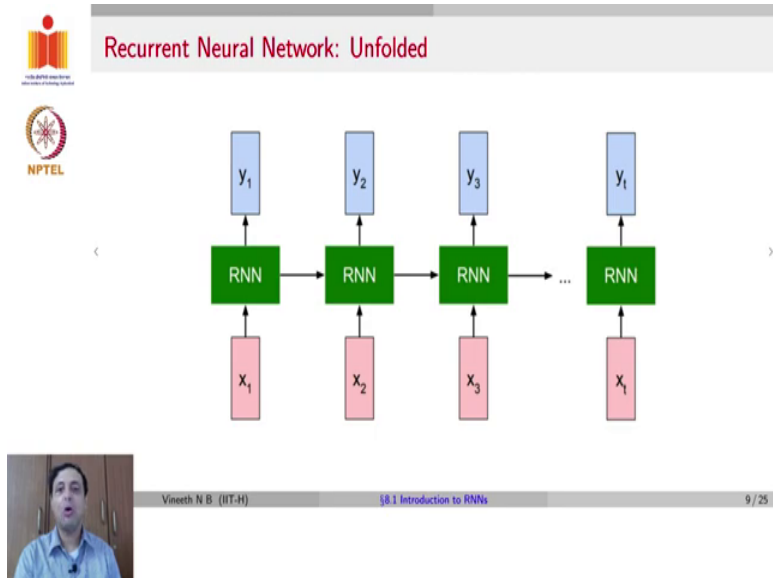
(Refer Slide Time: 12:10)



So the high level abstraction of a recurrent neural network is this figure here. You have an input x and output y . And in between, you have what is known as a recurrent neural network. As the word says, there is a recurrence here, which is now represented by this self-loop.

The key idea of RNNs is that they have an internal state, which is updated as a sequence is processed. One could view this internal state as some kind of a memory that retains some information from past data of the same time series. And this idea is something that we will talk about later this week also.

(Refer Slide Time: 13:02)



If one unfolded the recurrence, you would get a network such as this, where you have a set of inputs, x_1 to x_t over time. So that is a sequence of inputs that you have. And that is why we are looking at this as a sequence learning problem. And for each input, you can give that input to an RNN block. So this RNN block, could be a single hidden layer, could be multiple hidden layers, could have skipped connections, could have any other architectures inside this RNN block. But we are going to look at that as one single obstruction.

And the output of that RNN is the output of the model at that time step. This would be the entire unfolded RNN. And when you work with RNNs, you have to decide beforehand how much you want to unroll an RNN. What do we mean? Given a particular problem, you have to decide what this small t must be. Do you want to look at the past 20 frames to make a decision? Do you want to look at the past 100 frames to make a decision or do you want to look at the past 200 frames to make a decision.

This is something that has to be decided before you start learning an RNN. This t here would correspond to that length of the sequence that you want to look at while making the prediction. And once t is decided you could unroll the RNN this way where the same operations inside this RNN block is used for the input at every time step.

As we said earlier, depending on what application you are using RNNs for, you could have an output at each time step, or you could have an output only at the last time step or you could also

have a sequence of outputs that follows the input at the last time step. All those possibilities are variants of this abstraction that we see on the screen.

(Refer Slide Time: 15:19)

Recurrent Neural Network

We can process a sequence of vectors x by applying a **recurrence formula** at every time step:

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

h_t : new state
 f_{UW} : some function with parameters $U \times W$
 x_t : input vector at some time step
 h_{t-1} : old state

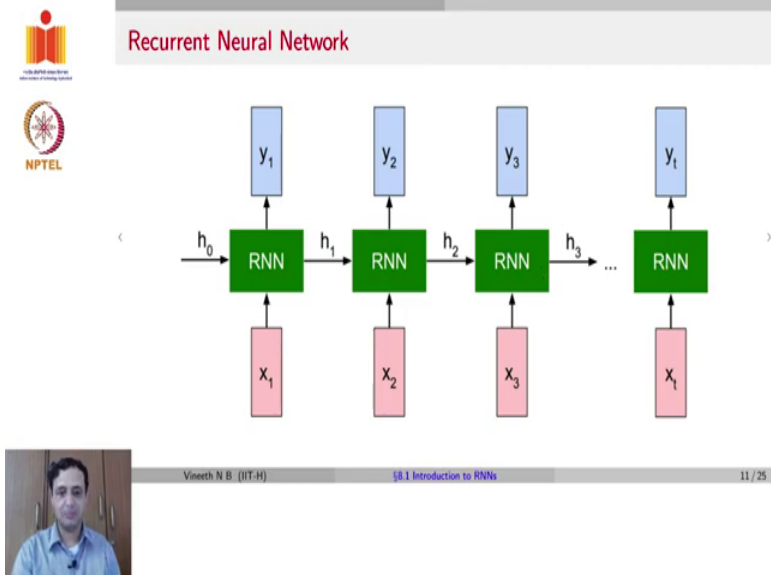
Diagram: An input vector x (pink box) is fed into an RNN block (green box). The RNN block has a feedback loop. The output of the RNN block is a vector y (blue box).

Vineeth N B. (IIT-H) 8.1 Introduction to RNNs 10 / 25

Let us go into some more details now. So we said it is a recurrent neural network. And we also said that we want this RNN block here to repeat the same operation for the inputs at every time step. So what is that recurrence operation that we want to actually use? The recurrence formula at every time step for us would be given an input x_t at a particular time step.

And $h_t - 1$ is the state of the RNN in the previous time step, then you are going to apply some function, this function are the parameters inside the RNN with some weights, U and W . And you get an output h_t at that time step which is passed onto the next stage of the RNN, which process the input at the next time step.

(Refer Slide Time: 16:19)



Here is a visualization. At time step t_1 , given an input x_1 and h_0 , an initial state, the RNN processes, these 2 inputs h_0 and x_1 , using weights, say U and W which are matrices of weights, and outputs a y_1 if an output is required at that time step, and a state h_1 , which is passed on to the next time step of the RNN.

But the next time step, the RNN receives x_2 and h_1 as input, uses the same weights U and W as the previous time step. Remember, we said we want the RNN to treat every frame as a similar kind of input and perform similar operations, although they are different, although they are differently placed in time. So the RNN at the second time step outputs y_2 if an output is required at that stage, depending on the problem, and h_2 which is its state hidden state, which is passed down to the RNN at the third time step. And this is continued all the way till the last time step.

(Refer Slide Time: 17:40)

Recurrent Neural Network

We can process a sequence of vectors x by applying a **recurrence formula** at every time step:

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

Notice: the same function and the same set of parameters are used at every time step.

Vineeth N B (IIT-H) 58.1 Introduction to RNNs 12 / 25

So the recurrence formula, just to repeat is $h_t = f_{UW}(x_t, h_{t-1})$ where U and W are those weight matrices. Given input x_t and h_{t-1} , the key observation here is that the same function and the same set of parameters are used at every time step. That does not change. And that is where the recurrence actually lies in a recurrent neural network.

(Refer Slide Time: 18:10)

(Simple) Recurrent Neural Network

The state consists of a single "hidden" vector h :

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

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

$y_t = \text{SoftMax}(Vh_t)$

Sometimes called a "Vanilla RNN" or an "Elman RNN" after Prof. Jeffrey Elman

Vineeth N B (IIT-H) 58.1 Introduction to RNNs 13 / 25

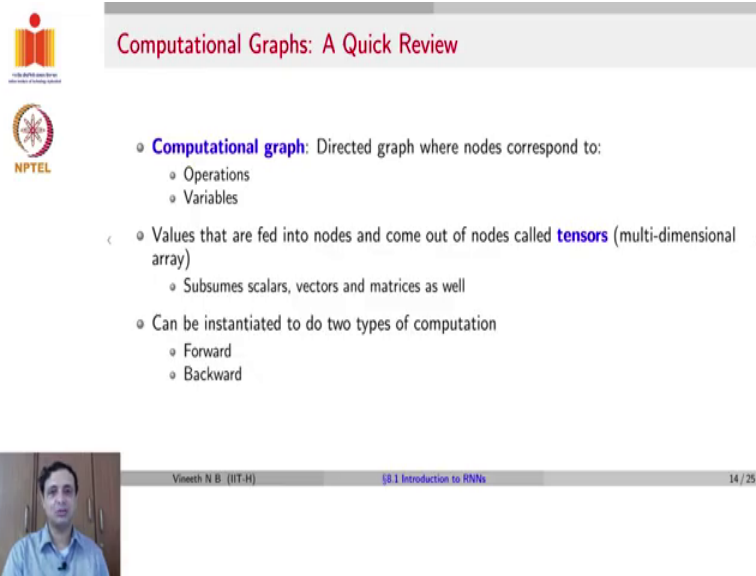
So you could now think of using activation functions depending on the task that you have. So this recurrence relation that we just wrote here, would now be $h_t = \tanh(Ux_t + Wh_{t-1})$.

y_t could have a softmax of Vh_t . So if you had an output h_t that comes out of this RNN, then you have a set of weights V here which multiplies Vh_t .

And then if you are dealing with a classification problem, you apply a softmax on that Vh_t to give your final output of the RNN. And the weights here going from x to RNN would be U and the weights that look at h_{t-1} and give a h_t would be W . So you can see that there are 3 matrices of weights. Ofcourse W could change based on how many layers you have inside the RNN block.

Otherwise, you have 3 sets of weights here that operate on input, hidden state, and hidden state to output. This is sometimes called also an Elman RNN. After Professor Jeffrey Elman, who was responsible with initiating these ideas.

(Refer Slide Time: 19:56)



The slide features the NPTEL logo on the left and a title bar at the top. The main content is a bulleted list defining computational graphs and tensors. At the bottom left, there is a small video feed of the presenter, and at the bottom right, there is a footer with the presenter's name, the course title, and the slide number.


- **Computational graph:** Directed graph where nodes correspond to:
 - Operations
 - Variables
- Values that are fed into nodes and come out of nodes called **tensors** (multi-dimensional array)
 - Subsumes scalars, vectors and matrices as well
- Can be instantiated to do two types of computation
 - Forward
 - Backward

Vineeth N.B. (IIT-H) 3.1 Introduction to RNNs 14 / 25

To go forward and understand how the RNN is actually trained, We will take one step back and look at the notion of what are known as computational graphs. Computational graphs are a very common approach to implement neural networks, both inference and how they are back propagated today, including in frameworks such as Py Torch or tensor flow. So we will review computational graphs in general first, and then talk about how computational graphs are defined for RNNs. So a computational graph is a directed graph whose nodes correspond to either operations or variables.


The values that are fed into the nodes and come out of the nodes are tensors, tensors recall are multi-dimensional arrays generally, anything greater than 2-dimensions, or 2 dimensional generalization of a vector would be a matrix, a generalization of a matrix would generally be called a tensor. A tensor generally subsumes scalars, vectors, matrices into all of them, tensors is the more general term to represent these quantities. And as I just mentioned, the computational graph can be instantiated, to do both a forward pass through a neural network, as well as a backward pass through a neural network.

(Refer Slide Time: 21:35)




Computational Graphs: Creating Expressions

- Nice way to think about mathematical expressions
- Consider the expression:
 $e = (a + b) * (b + 1)$
- 3 operations:
 - 2 additions
 - 1 multiplication
- Intermediate steps
 - $c = a + b$
 - $d = b + 1$
 - $e = c * d$


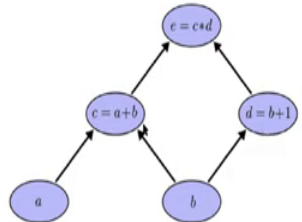


Vineeth N.B. (IIT-H) 58.1 Introduction to RNNs 15 / 25



Computational Graphs: Creating Expressions

- Nice way to think about mathematical expressions
- Consider the expression:
 $e = (a + b) * (b + 1)$
- 3 operations:
 - 2 additions
 - 1 multiplication
- Intermediate steps
 - $c = a + b$
 - $d = b + 1$
 - $e = c * d$



Credit: Christopher Olah
Vineeth N.B. (IIT-H) 58.1 Introduction to RNNs 15 / 25

Let us take a simple example before we talk about neural networks, let us talk about how we can use computational graphs to create expressions and compute them. So if you had the expression, $e = (a + b) * (b + 1)$, this is an arithmetic expression. There are 3 operations here, 2 additions, and 1 multiplication. And one could write this entire expression or operation as a sequence of intermediate operations. What are those intermediate operations, you can first say $c = a + b$, $d = b + 1$, $e = c * d$. So you are saying the first term here is c, the second term here is d.

And then you finally multiply the two to get an e. This seems trivial. Why are we talking about this? One could now write this as a computational graph. You have values a and b, given as input, you compute c, given by $a + b$, then you compute d which depends only on b which becomes $b + 1$. And then you combine c and d to get your final output e which is $c * d$.

(Refer Slide Time: 23:12)

The slide features the NPTEL logo on the left. The title is "Computational Graphs: Evaluating Expressions". Below the title, a list of steps for evaluation is provided:

- To evaluate the expression
 - Set input variable to certain values
 - Compute nodes up through the graph

The computational graph consists of five nodes: a root node $e = c * d$ with value 6; a left child node $c = a + b$ with value 3; a right child node $d = b + 1$ with value 2; a leaf node $a = 2$; and a leaf node $b = 1$. Red boxes highlight the values in each node, and red arrows indicate the flow of computation from the leaf nodes up to the root.

Credit: Christopher Olah
Vineeth N B (IIT-H) §8.1 Introduction to RNNs 16 / 25

How do you evaluate these expressions? Let us say you had specific values of a and b given to you. How do you evaluate this expression using a computational graph? You would set the input variable to certain values that are given to you. For example, a is a particular value, and b is a particular value. And then you would compute the nodes up the graph from those variable instance values all the way till the root node to get your final answer.

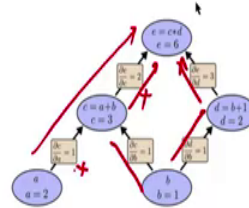
So it would look something like this, you set $a = 2$, you set $b = 1$, you first compute $c = a + b$ which means you get $c = 3$. Similarly, $d = 2$. And then you combine them and get your final answer for e which is, $e = 6$.

(Refer Slide Time: 24:03)



Computational Graphs: Computing Derivatives

- How?
- Key is to understand derivatives on edges (where changes - e.g. how a affects c - are tracked)
- We then apply **sum rule** and **product rule** appropriately to gradients
- General rule is to sum over all possible paths from one node to other, multiplying derivatives on each edge of path together



Credit: Christopher Olah
Vineeth N.R. (IIT-H)

§8.1 Introduction to RNNs

17 / 25

That was nice and simple but how do you use such a computational graph for the backward pass if you want to do back propagation or in other words, how do you use computational graph to compute gradients or derivatives. The key here is that to understand gradients, you need to understand the edges of a computational graph, because the edges denote how a particular variable affects another variable. For example, if you had the same computational graph, you have a at this leaf node here and you have $c = a + b$ at a parent of that leaf node.

Now one could write $\partial c / \partial a$, as you have $c = a + b$. Hence, $\partial c / \partial a$ will be equal to 1. Similarly, on this edge connecting b and c , $\partial c / \partial b$ will also be 1, because $c = a + b$, and so on and so forth for every set of edges in this graph. So how are we doing this, we are applying the standard sum rule and product rule appropriately to the gradients to compute each of those gradients at the edges. And to get your final gradient, you could sum over all possible paths from one node to another, multiplying the derivatives on each edge of the path together.

So for example, if we want $\partial e / \partial a$, we look at all possible paths that take us from a to e . So in this particular case, that would be just this, this path, you multiply all the gradients that are on that path, so in this case, it will be 1×2 . And that gives you the gradient of $\partial e / \partial a$. Similarly, if you had to compute $\partial e / \partial b$, you would have to consider all possible paths that take you from b to e . You multiply the gradients in each of these paths, and add up both these paths contributions, and you will get your final gradient.

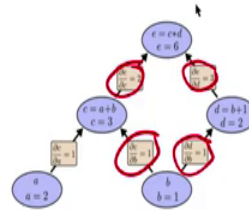
(Refer Slide Time: 26:21)



Computational Graphs: Computing Derivatives

- How?
- Key is to understand derivatives on edges (where changes - e.g. how a affects c - are tracked)
- We then apply **sum rule** and **product rule** appropriately to gradients
- General rule is to sum over all possible paths from one node to other, multiplying derivatives on each edge of path together
- E.g. to get derivative of e w.r.t. b :

$$\frac{\partial e}{\partial b} = 1 * 2 + 1 * 3$$



Credit: Christopher Olah
Vineeth N B. (IT-H)

8.1 Introduction to RNNs

17 / 25

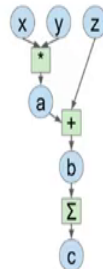
So once again, $\partial e / \partial b$ in this case would be $1 \times 2 + 1 \times 3$. That is how you would use a computational graph to get your gradients in a particular expression.

(Refer Slide Time: 26:37)



Computational Graphs: PyTorch Example

- In PyTorch, for e.g., changes are tracked on the go during forward pass allowing for dynamic graph creation
- Gradients are calculated only when **backward()** function is triggered



```
import torch
from torch.autograd import Variable

#---- Define Variables to build computational graph ----#
x = Variable(torch.tensor([1.0, 2.0]).cuda(), requires_grad = True)
y = Variable(torch.tensor([2.0, 3.0]).cuda(), requires_grad = True)
z = Variable(torch.tensor([4.0, 3.0]).cuda(), requires_grad = True)

#---- Forward Pass ----#
a = x * y
b = a + 2
c = torch.sum(b)

#---- Compute Gradients ----#
c.backward()

print(x.grad.data) # out = [2., 3.]
print(y.grad.data) # out = [1., 2.]
print(z.grad.data) # out = [1., 1.]
```



Vineeth N B. (IT-H)

8.1 Introduction to RNNs

18 / 25

So, to give you a more concrete example, in a framework that you may be using, which is PyTorch, PyTorch changes are tracked on the go during the forward pass, allowing for the creation of such a computational graph, which is called a dynamic graph. And when you say `loss.backward`, when you compute when you actually train a neural network in PyTorch, if you say `c.backward` for instance, then it could be loss for training a neural network or in if you are

just evaluating an expression, you could just say `c.backward` in this particular example that we saw.

And when you dot `c.backward`, the Py Torch framework triggers the computation of the gradients. And you can then print out the gradients of every edge for every gradient in your expression computation.

(Refer Slide Time: 27:36)

Computational Graphs: MLP

$$h = \tanh(Wx + b)$$

$$y = Vh + a$$

$$f(u, v) = u + v$$

The slide also includes logos for NPTEL and a video thumbnail of the presenter, along with a credit line: 'Credit: Yoav Artzi CS5740' and 'Vineeth N B. (IT-H) | 8.1 Introduction to RNNs | 19 / 25'.

What about computational graphs for neural networks feed forward neural networks. So if you had a couple of layers in a neural network, given an input x , let us say your hidden layer h is $\tanh(Wx + b)$, $Wx + b$ is your linear combination of your inputs x .

And let us assume you have a nonlinear activation function \tanh on each of those nodes and that gives you a vector h , then you apply another set of weights v on h to give you your final output y , where b and a are your biases. If this was your mathematical representation of a feed forward neural network, your corresponding computation graph would look like your first inputs are W and x based on that, you get a certain output here and then you combine that with $+b$ and you then get a certain output.

And then finally, you take that and apply \tanh and get your $\tanh(u)$. So this first step here can be represented as a computational graph, that is a sequence that takes Wx and b as inputs and outputs h . Similarly, now for the next operation, you can now consider h as the input, V as the

other input a as the other input. And the computational graph at this time would be this set of operations. Obviously, the input to one of the nodes in this computational graph is the h , which came as an output from previous computational graph.

(Refer Slide Time: 29:22)

Back to RNNs: Computational Graph

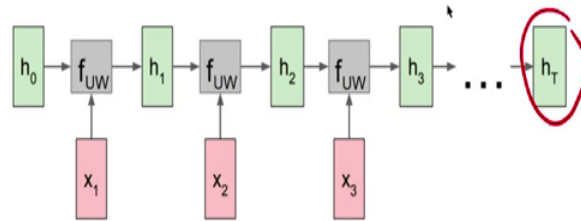
Vineeth N B. (IIT-H) 8.1 Introduction to RNNs 20 / 25

Back to RNNs: Computational Graph

Vineeth N B. (IIT-H) 8.1 Introduction to RNNs 20 / 25



Back to RNNs: Computational Graph



Vineeth N.B. (IIT-H)

§8.1 Introduction to RNNs

20 / 25

Now coming back to RNNs, since RNNs are a slightly complex, let us try to view RNNs from the point of view of computational graphs. So given input x_1 and h_0 , the input is given to f_{UW} which is what we talked about earlier. And f_{UW} outputs h_1 , then h_1 and x_2 , go to f_{UW} again, which is the same function that is evaluated at each time step as we already mentioned. Then f_{UW} at second time step gives you h_2 and this is again recurrently given at the next time step, and so on and so forth, until you get the final hidden state at the last time step of the sequence.

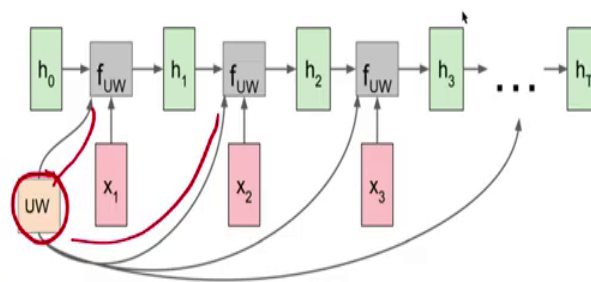
(Refer Slide Time: 30:13)



Back to RNNs: Computational Graph



Re-use the same weight matrix at every time-step



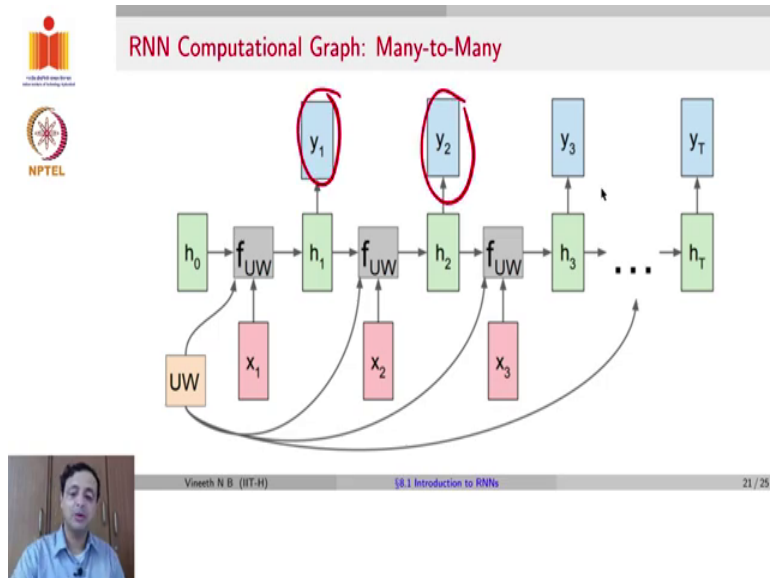
Vineeth N.B. (IIT-H)

§8.1 Introduction to RNNs

20 / 25

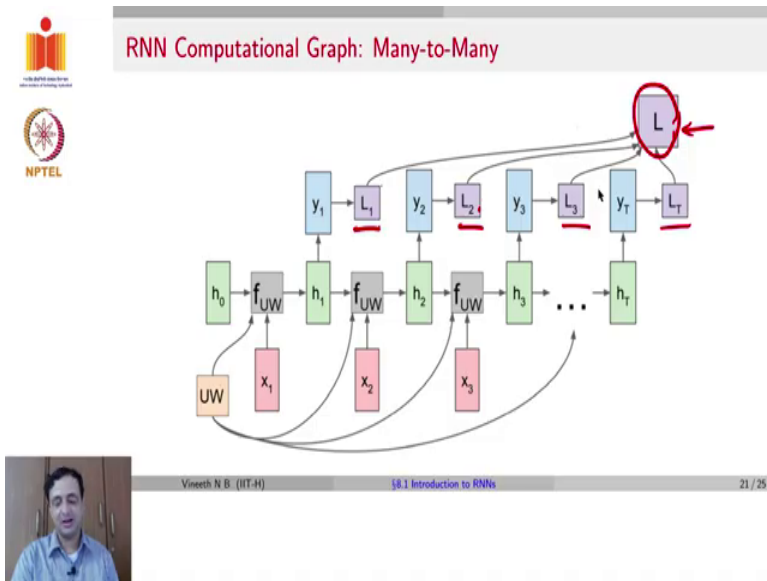
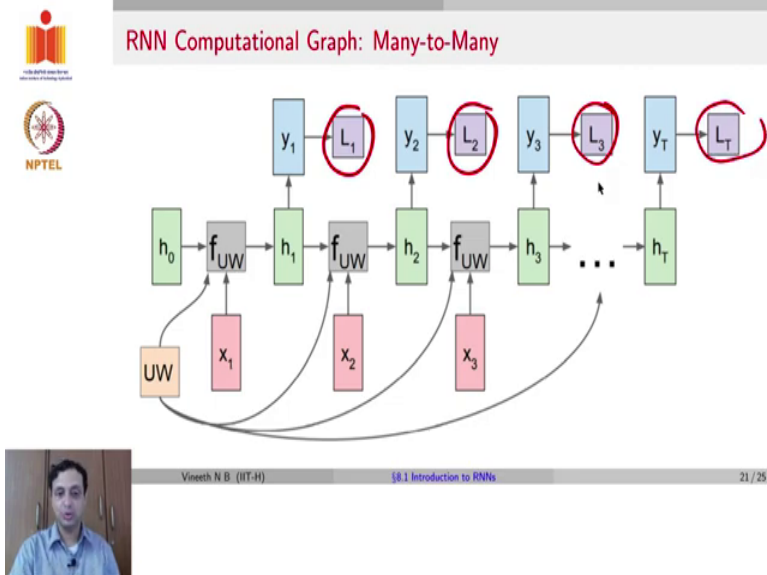
So from a computational graph perspective, the inputs, UW here which are the weights of the matrix of the, the weight matrices rather, are the same for every time step. And that is what is shown by these arrows that go into a f_{UW} at every time step.

(Refer Slide Time: 30:36)



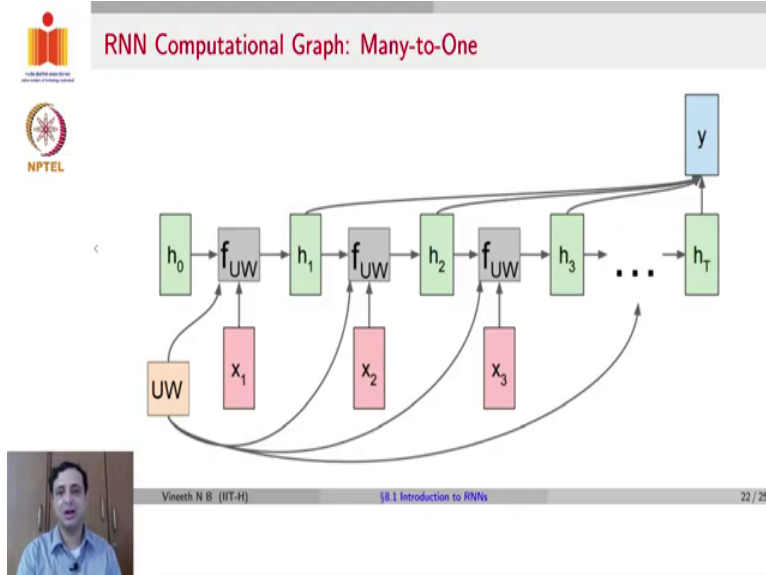
If you had a many to many RNN, you would only slightly change this, because now you are going to have outputs at every time step, which you did not have in the previous RNN that we just saw.

(Refer Slide Time: 30:51)



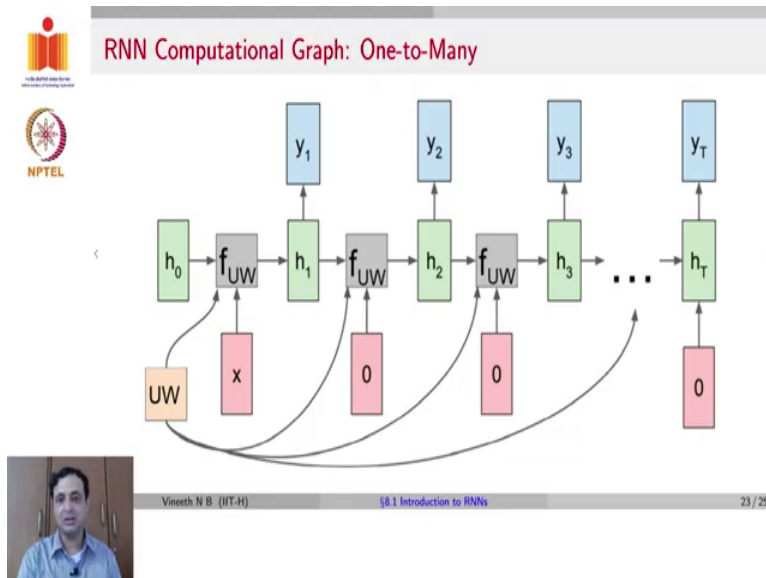
You could now have an output L_1 that comes out of the many to many RNN which is what is finally given as output to the user. You could also look at combining all of these $L_1, L_2, L_3, \dots, L_t$ to give your final L as the output. Why does this matter? If you look at say video captioning, which would have been an instance of such an many to many architecture, you could look at L_1, L_2, L_3, L_t so on and so forth, as variables that hold each word of the caption, and L be the final caption that is given as output of the RNN.

(Refer Slide Time: 31:39)



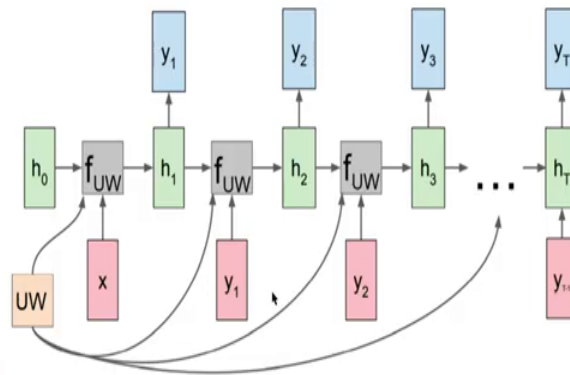
If you considered many to one architecture the computational graph would look something like this, the initial part more or less, remains the same. But the output y now is only at the last time step; is this complete? Not really, in terms of the computational graph, you also need the connections where the y could be an output that also receives h_1, h_2, h_3 so far, and so on, as also inputs to compute the final output value.

(Refer Slide Time: 32:14)





RNN Computational Graph: One-to-Many



Vineeth N B (IIT-H)

8.1 Introduction to RNNs

23 / 25

In case of a one to many setting, you could have this kind of a computational graph, where the input is only x at the first time step. But you now have outputs at every time step y_1, y_2, y_3 through y_t one question here is, what would you give as inputs at intermediate time steps? Why do we need to talk about this? Remember, f_{UW} is the same function at every time step. We already know that the first time step f_{UW} needed 2 inputs x and h_{naught} , which means at every time step, it does need 2 inputs. What do we do?

One option, we can input zeros at every time step, that could be one option that we can provide. Another option, which is also sometimes followed is the output of the previous time step could also be given as input at the next time step. By adjusting for dimensions, you could use the output at the previous time step also as input to the next time step. Why is this relevant? To generate the next word in a caption, you could provide the previous word in the caption to help the model generate the appropriate next word.

(Refer Slide Time: 33:44)

Example: Character-level Language Model

- Vocabulary: [h,e,l,o]
- At test time, sample characters one at a time, feed output back to model



Sample "h"
Softmax: 0.03, 0.4, 0.56, 0.13
output layer: 1.0, 2.2, -3.0, -4.1
hidden layer: 0.3, -0.1, 0.9
input layer: 1, 0, 0, 0
input chars: h
One-hot vector

Vineeth N B. (IIT-H) 58.1 Introduction to RNNs 24 / 25

To give a more tangible example, so let us consider a RNN model. Let us assume that the inputs and the outputs come from a vocabulary, h, e, l, o. And your input character, let us say is h, which is denoted as 1 0 0 0 0. Such a representation is also known as a one hot vector. A one hot vector is a vector which has a 1 at the position of the input that is given there at the position from the vocabulary, and 0 elsewhere, since h was the first character here, if your input characters h, the corresponding 1 hot vector would be 1 0 0 0 0.

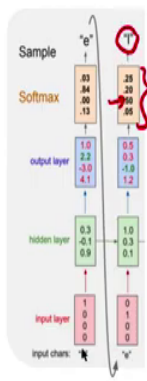
And given that as inputs, let us assume that you have a hidden layer, you have an output layer. And let us assume that the output layer had these set of outputs and doing a softmax you end up getting a certain set of probabilities where the second one has the highest probability, which means you are going to predict the next character as e.


(Refer Slide Time: 34:55)

Example: Character-level Language Model

- Vocabulary: [h,e,l,o]
- At test time, sample characters one at a time, feed output back to model







Vineeth N B. (IIT-H)

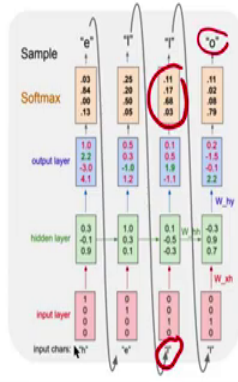
§8.1 Introduction to RNNs


24 / 25

Example: Character-level Language Model

- Vocabulary: [h,e,l,o], <>
- At test time, sample characters one at a time, feed output back to model





Vineeth N B. (IIT-H)

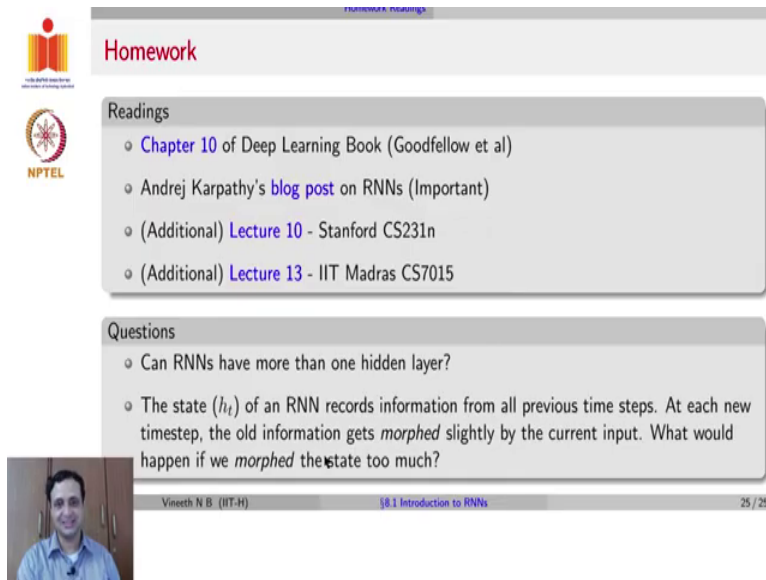
§8.1 Introduction to RNNs

24 / 25

Now this character is given as input to the next time step. Remember, we just said the y_1 is given as input to the next time step as x_2 . So e because of position of e as the second index in your vocabulary will be given by one hot vector 0 1 0 0. And this is then passed through the same network, and you now get a softmax output as 0.25, 0.2, 0.5, 0.05. The highest one is the third one, which is an L. And that is the output of the RNN at that time step, that L is then again given an input to the next time step, and L is denoted as 0 0 1 0, because of the position of L, in that in the vocabulary.

Once again you do this process, and it once again gives L as the output, that L is given to the next time step. And the output then is O and at some point, the entire RNN model can give an end token as the output to know where the set of outputs end. So an end token would be a default character in the vocabulary and that would be predicted when the RNN model decides to end the generation of that particular word.

(Refer Slide Time: 36:24)



The screenshot shows a slide titled "Homework" with two main sections: "Readings" and "Questions".

Readings

- Chapter 10 of Deep Learning Book (Goodfellow et al)
- Andrej Karpathy's [blog post](#) on RNNs (Important)
- (Additional) [Lecture 10](#) - Stanford CS231n
- (Additional) [Lecture 13](#) - IIT Madras CS7015

Questions

- Can RNNs have more than one hidden layer?
- The state (h_t) of an RNN records information from all previous time steps. At each new timestep, the old information gets *morphed* slightly by the current input. What would happen if we *morphed* the state too much?

At the bottom of the slide, there is a small video inset of a man, the text "Vineeth N B. (IIT-H)", "§8.1 Introduction to RNNs", and "25 / 25".

As homework, your readings would be chapter 10 of the Deep Learning book. Andrej Karpathy's excellent blog posts on RNNs. And additionally, if you would like to go through the Stanford CS231n course or the IIT Madras CS7015 course. In the next lecture, we will talk about how these RNNs can be trained using back propagation. Before that, a couple of questions for you. Can RNNs have more than one hidden layer?

We answered that question, but think about it to ensure it can. And the next question is, the state h_t of an RNN records information from all previous time steps. Each new time step, in a sense, the old information gets morphed slightly by the current input. So you have an h_{t-1} that comes, you have an x_t that comes. The previous steps state is now modified due to the input that comes at this time step.

What would happen if we morph the state too much? Rather, what would happen if you pay a lot of attention only to the current input and lesser the previous time steps state? Think about it and we will discuss this soon.