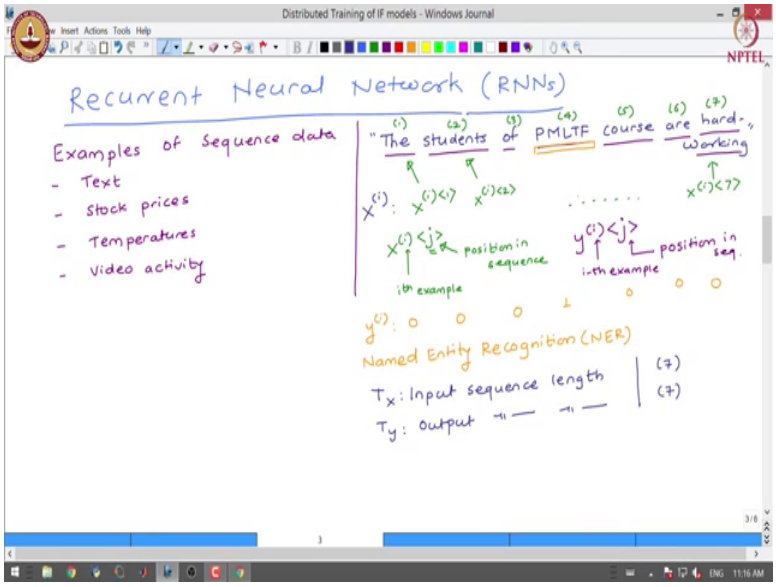
Practical Machine Learning
Dr. Ashish Tendulkar
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Lecture - 29
Recurrent Neural Networks

[FL] Welcome to the next session of our course on Practical Machine Learning with TensorFlow 2.0. So, far in this course we have build machine learning models for structured data for images. Now, we will try to build models for other type of data which is sequential data.

(Refer Slide Time: 00:41)



So, there are lot of examples in real life, where we encounter sequence data. Sentences in language is an example of a sequential data.

“All the students in practical machine learning course are hardworking”

This particular sentence has different words. And these words are dependent on each other. If I jumbled the words the sentence will not have any meaning. So, this is an example of a sequence data.

Another sequence data could be the stock prices, temperatures on everyday are examples of a sequence data. So, we cannot use the traditional neural network models, like feed forward neural networks for modeling the sequence data efficiently. There are couple of reasons for that. First of all different sequences can have a different lengths and it is hard to model such a data with feed forward neural networks.

Apart from that the sequence dependency is not taken into account when we use something like feed forward neural network to model these sequences. Here, we will learn what is called as a Recurrent Neural Network or RNNs that are used for modeling sequential data. Let us understand how RNN look like and how they function. So, let us first list down some of the example of sequence data: text, stock prices, temperatures, video activity etc.

So, let us come up with a notation to represent the sequence data. So, let us have the sentence: "The students of PMLTF course are hardworking", is the sequence. Since it is a sequence we have the following notation. So, let us say this is an example $x^{(i)}$ and this $x^{(i)}$ example has multiple words or tokens in it. So, there are words are hardworking. So, hardworking is a single token. So, there are tokens and let us represent these tokens with slightly different syntax, this i^{th} example this is the first token.

So, we use $x^{(i)<j>}$ where it is i^{th} sequence and j^{th} positioned in sequence. There are 7 words here in this example. So, we have $x^{(i)<1>}, x^{(i)<2>}, x^{(i)<3>}, \ldots, x^{(i)<7>}$.

Let us say if we are solving a problem of identifying named entities. So, in this case "PMLTF" is an example of a named entity, this is a name of the course. So, there will be specific label assigned to each of the words. So, let us say $y^{(i)}$ is the labelled sequence for this particular sentence. So, all the labels this is not a named entity not a named entity not a named entity this is a named entity.

So, this is these are the labels or this is the labeled sequence for this particular example in the context of named entity recognition (NER) task. We use the same representation to denote label of individual tokens. So, for example, we will use $y^{(i)<j>}$. Make sure you are familiar or you understand this notation properly before diving into the main

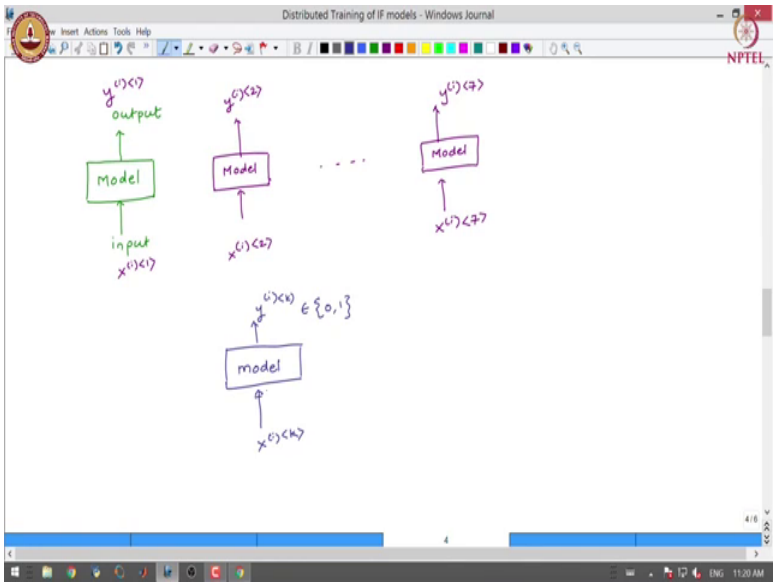
concepts of RNN. Apart from that we have couple of other variables one is T_x: which is input sequence length and T_y which is output sequence length. In this case in the case of this example we have 7 inputs and we have 7 outputs.

It is not necessary that length of both input and output sequence should be the same. In fact, there are a lot of examples where these two lengths do not match. So, let us say if you are solving this NER problem, where given a sentence you are interested in tagging the named entities.

You can imagine that we will have sentences which are of different lengths. If you use the traditional machine learning algorithms or the algorithms that we have learnt so far in the class, they will not work. If we have different length input sequences other drawback of these algorithms is that we will not be able to share the weights across different positions.

It might be helpful to share weights learn at a particular position. In the sequence with the other position so, that the other position gets benefited by whatever was learned in some other positions in the sequence. So, weight sharing is one of the main drawbacks, if you use some model like feed forward neural network model for sequence modeling task.

(Refer Slide Time: 11:33)



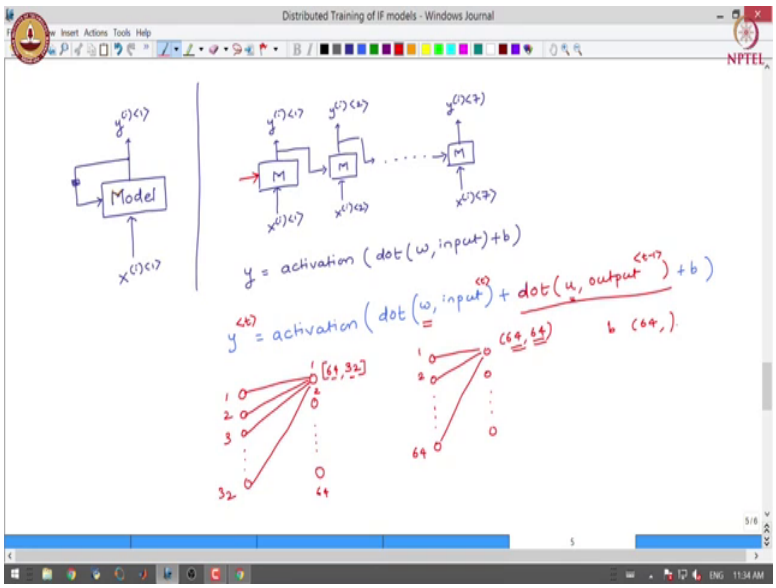
So, far the models that we studied are of this type. We use to give some input and the model used to give us the output. If you want to use exactly the same idea for the task of NER, we will have to put a model at every position. So, in the context of example you would have to put 7 such kind of models to one at each position.

You can see that these models are independent of each other. They are not taking into account the sequence information, which has proven to be useful in the task like NER.

So, how do we really use the sequence information and use that information in the modeling? Here we have different models and these different models are learning weights independent of different inputs.

So, there is no parameter sharing that is happening. You might argue that instead of having *n* different models, can we have a single model. That is perfectly a valid approach instead of having *n* different models. We just have sometimes a single model that simply takes some position as input then in this model outputs $y^{(i)<j>}$. This $y^{(i)<j>}$ in the context of NER is {0, 1}. This model does not take into account the sequential information and it is completely ignored as far as this kind of modeling is concerned.

(Refer Slide Time: 15:12)



So, in order to take into account the sequential dependencies of labels . We can build a model let us say to begin with it takes first token as input. And it produces the label for

the first token. We then pass this label back into the model. This denotes the time delay of 1. This is a compact representation of the recurrent neural network models or RNNs.

So, let me expand this. So, we are having I will use *M* to denote the model. The first input which is producing the output. And this output is fed into the model that take second token as an input and produces output for that. We do this for the entire length on the sequence.

So, in this in our case till token number 7. So far in the neural network model we used to obtain *y* as activation. We used to do linear combination and we use to apply non-linear activation. Now, we will change this slightly.

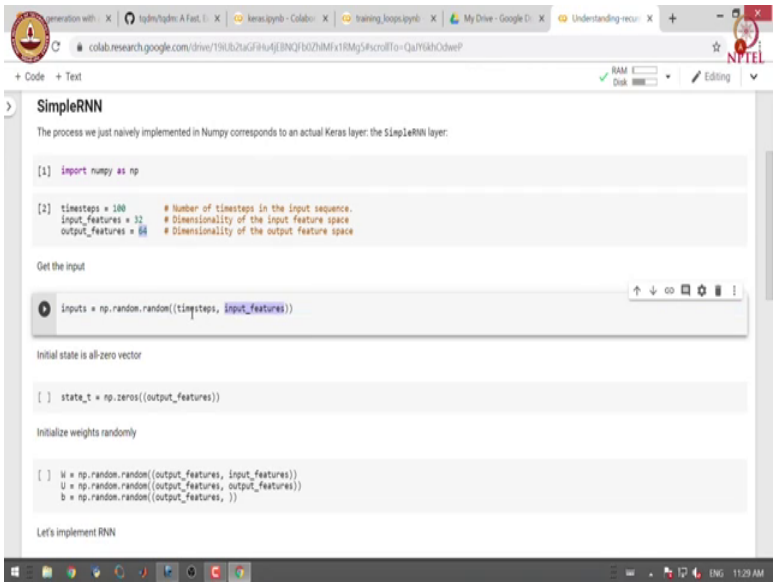
So, let us say we want to calculate it for t^{th} location:

$$y^{<t>} = activation(dot(w, input^{<t>}) + dot(u, output^{<t-1>}) + b)$$

So, we are adding this extra term that takes additional input which is the output of the previous node. So, in case of the first node we set this input to 0. Exactly the same calculation that we are doing, we are doing couple of dot products one between the input and its weights.

So, with this simple change I have incorporated the label information from the previous node in my calculation. So, let us try to understand this through a simple implementation.

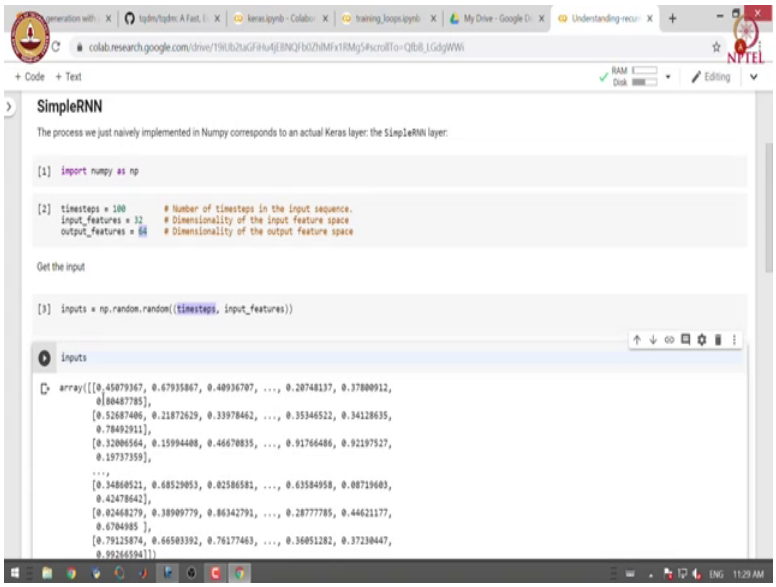
(Refer Slide Time: 20:26)



Here we have input sequence having 100 positions in it. These positions are referred to as timesteps. Each position has 32 features associated with it. So, the dimensionality of the input feature space is 32.

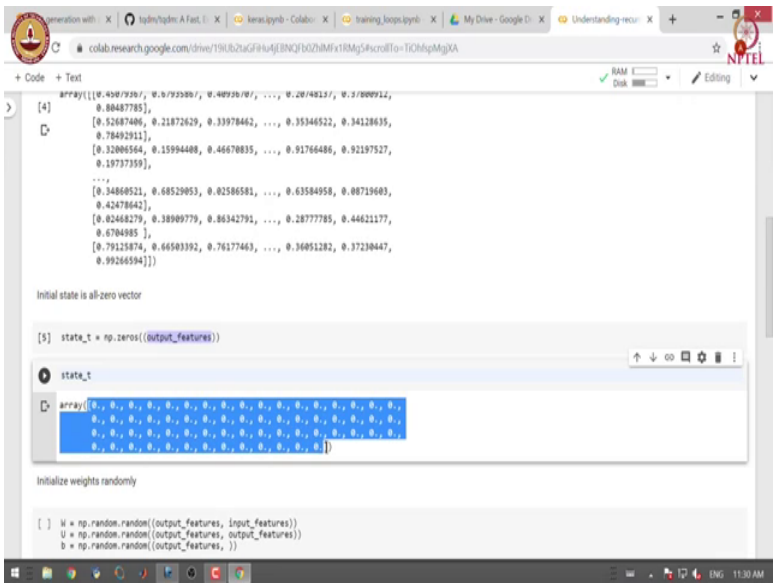
And we want to output 64 values for each of the position. So, 64 becomes the dimensionality of the output feature space. We generate input with a random number generator from NumPy. So, we generate 32 features for 100 timesteps.

(Refer Slide Time: 21:47)



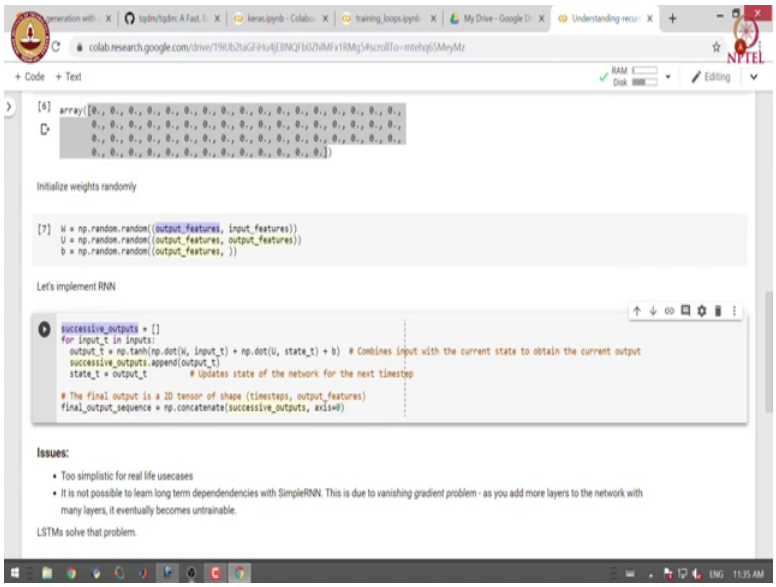
So, you can see that there are a bunch of random numbers that are there in the input. There are 32 numbers for 100 time steps.

(Refer Slide Time: 21:58)



We will initialize the state or the output from the previous step to all zeros. Since we expect 64 numbers to come out as output we use np.zeros() on 64 positions. So, if you look at what is there in the state? If you look at the content of the state it is all zero. So, we have a zero vector of length 64.

(Refer Slide Time: 22:52)



We initialize three weight vectors. So, we have a weight vector *W*; corresponding to the input, weight vector *U* corresponding to the output from the previous step and *b* is the bias unit. Let us understand the dimensionality of weight vectors corresponding to inputs and the previous output.

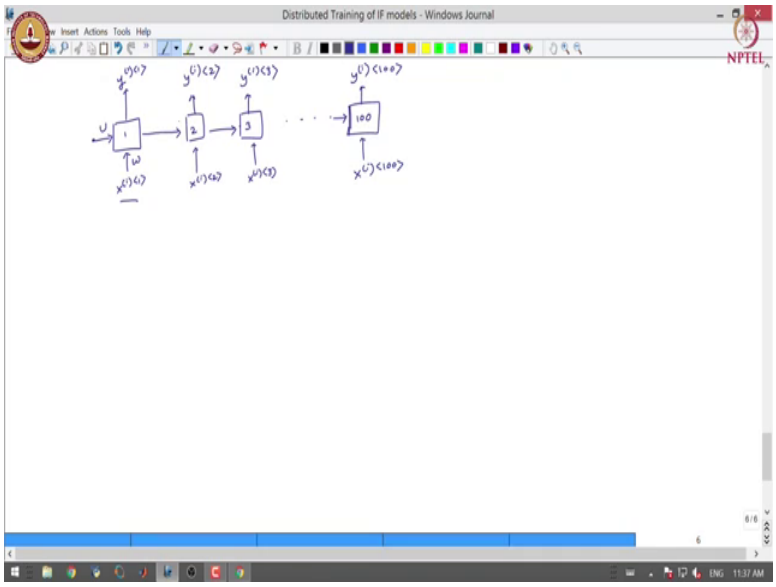
So, in our specific examples, we have 32 inputs and we need 64 outputs. So, we need to use 64 units at every position. So, each unit has 32 weights. So, we have a weight matrix which is 64 x 32. So, for each unit we have got 32 features.

In case of the weight matrix for the previous output there are 64 previous outputs and we want 64 outputs. So, there are 64 inputs. So, there are 64 weights for each of the unit over here. We have 64 by 64 matrix corresponding to the weights of the previous output.

And we have a bias unit under since there are 64 units. Here it is a vector having 64 values in it. We have weight vector. So, we have weigh matrix of shape output features by input features. We have weight matrix for previous output of shape output features by output features.

And we have a vector corresponding to the bias term. Given these vectors we have randomly initialize them. Let us look at how the RNN computation happens. So, here we are using a successive_outputs array to store the output coming from each time step. So, let me demonstrate what is going to happen in this case.

(Refer Slide Time: 26:57)



So, we have we have a sequence of 100 positions. So, we have a RNN model which has got which process, a sequence of length 100. In every time step we will take this input, we will multiply with the weight vector corresponding to the input.

Then we take the activation or the output from the previous node multiply that with *U*. Add the bias unit and we get the output *y*, which is feed into the next level. So, we will do this particular calculation in each loop. So, we find the output using a tanh as an activation function. Note: in RNNs we use tanh as activation function. tanh have been shown to be yielding better results in case of RNNs.

So, we perform dot product between the input and its parameter vector again, the previous output and its parameter vector and we add the bias term to it. This output we store in the successive output for future reference. And we reset the state to the output or we update the state of the network for the next timestamp.

So, let us run this and we can see that the entire output is concatenated and stored in this particular output sequence. Each node gives an output which has got 64 numbers in it. But there are problems with such a simplistic model. So, RNN models the simple this is called a simple RNN models this is too simplistic for real life situations.

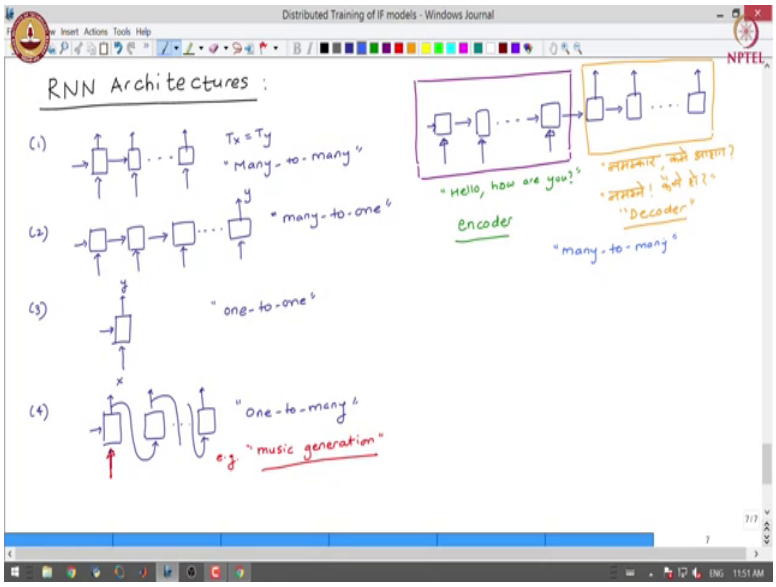
It is not possible to learn long range dependencies using this simple RNN models. This happens mainly due to the problem of vanish ingredients. Vanishing gradient affects us when we tried to add more layers in the network that already has many layers. The effect of that is that the model becomes eventually untenable. So, the varnishing gradient problem is a big issue in simple RNNs.

We will study techniques to tackle the problem of vanishing gradients. We do some simple tweaks to the architecture to take care of the vanishing gradient problem. Hope you understood with this example how simple RNN works or the basic concept behind a RNN. To summarize it is a very simple concept. We use a model and we share the weights across all the time steps in the sequence the model weights are shared.

We apply the model on the input coming at each time step. We get the output and that output is freed into the next time step. That is used to calculate the output. So, this output is combined with a dot product of input with its parameter vector. We also do dot product of this output with its parameter vector, we add a bias term and perform tanh activation on that.

We proceed in this manner till the end of the sequence. In this case we are getting the output from each of the node that need not be the case. There are cases where the length of the output sequence may not be the same as the length of the input sequence. So, let us try to look at some of those interesting architectures that we encounter often in practice. So, the content over here are taken from Andrej Karpathy's blog on Unreasonable effectiveness of RNNs.

(Refer Slide Time: 33:35)



So, we have already looked at one architecture, where there was output corresponding to each input. So, in this case $T_x = T_y$. These also called as many-to-many architecture. There could be a case where we do not get the input at every node, but instead we get output only at the last node. So, this is called as many-to-one.

Let us say if you want to perform sentiment analysis by scanning the sentences. It could potentially use many-to-one architecture. The third the third architecture is essentially one-to-one. We have one unit it give one input and we get one output.

Is this the same as the traditional networks that we use? There could also be a case where there is a single input and it is producing multiple outputs. This is called one-to-many. So, in this case this is the input and everything else is output. So, this kind of architecture is used in Music Generating RNNs.

There is another way in which we experience many-to-many architecture in RNN. Let us take an example of Machine translation, where it is a we give input sentence in English language.

We want output in let say Tamil language or in Marathi or in Hindi or in Guajarati. So, each one of them are individual machine translation task. So, there is an RNN,

corresponding to the input sequence. This is a RNN corresponding to input sequence. Here the output is shifted by some delay.

So, this particular part this could be corresponding to the sentence in English. It is a “Hello, how are you?” and let us and this particular part is the translation in Hindi. For example, it could be “Namaste, Kaise ho?”. Here this is called encoder and this part is called decoder and the output is shifted by the length of the input. This is also an example of many-to many-architectures.

So, these are different type of RNN architectures we offer encounter in practice. We will be showing demonstrations of some of these RNN architectures through Colab notebooks.