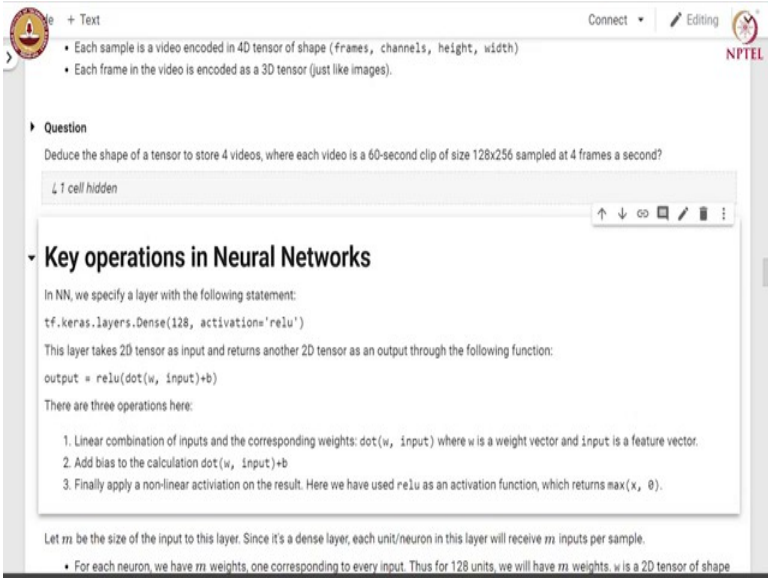


Practical Machine Learning
Dr. Ashish Tendulkar
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

Lecture – 11
Mathematical Foundations of Deep Learning - Contd.

[FL]. In the last session you studied basics of tensors and we also looked at tensors that we often encounter in practice. In this session we will focus on key operations on tensors in the context of Deep Learning.

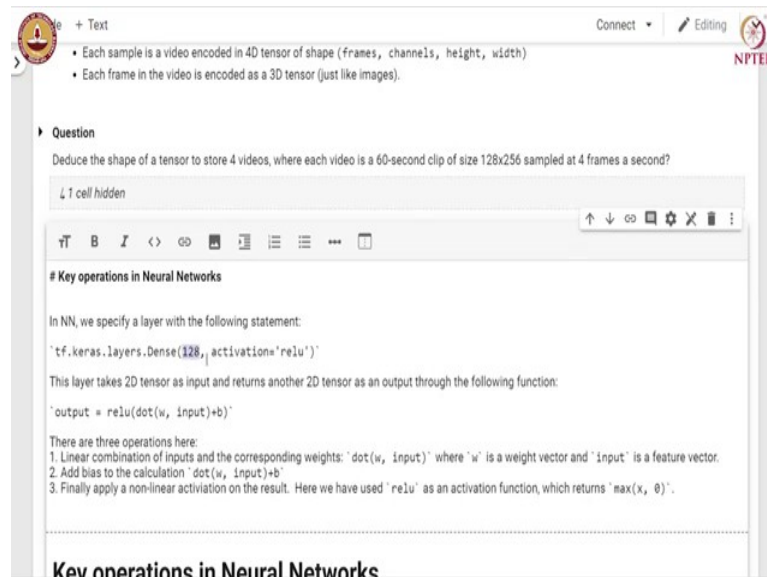
(Refer Slide Time: 00:29)



The screenshot shows a Jupyter Notebook interface with a title bar 'Text' and a 'Connect' button. The main content area contains a list of bullet points: 'Each sample is a video encoded in 4D tensor of shape (frames, channels, height, width)' and 'Each frame in the video is encoded as a 3D tensor (just like images)'. Below this is a 'Question' section with the text: 'Deduce the shape of a tensor to store 4 videos, where each video is a 60-second clip of size 128x256 sampled at 4 frames a second?'. A text input field contains '1 cell hidden'. Below the input field is a section titled 'Key operations in Neural Networks'. The text in this section reads: 'In NN, we specify a layer with the following statement: `tf.keras.layers.Dense(128, activation='relu')`. This layer takes 2D tensor as input and returns another 2D tensor as an output through the following function: `output = relu(dot(w, input)+b)`. There are three operations here: 1. Linear combination of inputs and the corresponding weights: `dot(w, input)` where `w` is a weight vector and `input` is a feature vector. 2. Add bias to the calculation `dot(w, input)+b`. 3. Finally apply a non-linear activation on the result. Here we have used `relu` as an activation function, which returns `max(x, 0)`. Let `m` be the size of the input to this layer. Since it's a dense layer, each unit/neuron in this layer will receive `m` inputs per sample. For each neuron, we have `m` weights, one corresponding to every input. Thus for 128 units, we will have `m` weights. `w` is a 2D tensor of shape

In neural network we specify the layer with `tf.keras.layers.dense()`. Here we use 128 units in a hidden layer and we use Relu as an activation function.

(Refer Slide Time: 00:44)



The slide is titled "Key operations in Neural Networks". It contains the following text:

- Each sample is a video encoded in 4D tensor of shape (frames, channels, height, width)
- Each frame in the video is encoded as a 3D tensor (just like images).

Question

Deduce the shape of a tensor to store 4 videos, where each video is a 60-second clip of size 128x256 sampled at 4 frames a second?

1 cell hidden

Key operations in Neural Networks

In NN, we specify a layer with the following statement:

```
'tf.keras.layers.Dense(128, activation='relu')'
```

This layer takes 2D tensor as input and returns another 2D tensor as an output through the following function:

```
'output = relu(dot(w, input)+b)'
```

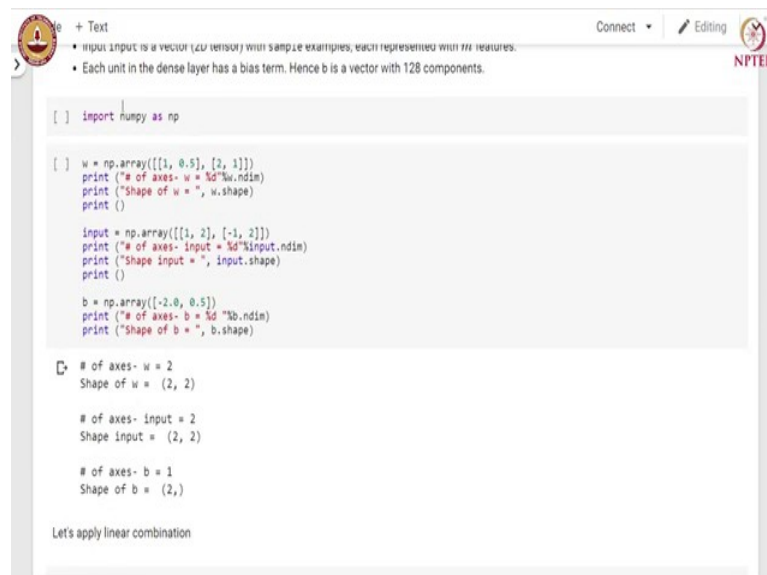
There are three operations here:

1. Linear combination of inputs and the corresponding weights: `'dot(w, input)'` where `'w'` is a weight vector and `'input'` is a feature vector.
2. Add bias to the calculation `'dot(w, input)+b'`
3. Finally apply a non-linear activation on the result. Here we have used `'relu'` as an activation function, which returns `'max(x, 0)'`.

Key operations in Neural Networks

The layer takes 2D tensor as an input and returns another 2D tensor as an output. The core operation that layer does is as follows, the layer computes `dot()` between the parameter vector and the input adds a bias vector to it. So, this becomes a linear combination and this linear combination is subjected to non-linear activation like Relu or Sigmoid. So there are three distinct operations here one is the linear combination of input and the corresponding weights and then we add bias in the calculation and finally we apply a non-linear activation on the result.

(Refer Slide Time: 01:50)



The slide shows Python code for a dense layer. It includes the following text:

- Input input is a vector (2D tensor) with sample examples, each represented with 128 features.
- Each unit in the dense layer has a bias term. Hence b is a vector with 128 components.

```
[ ] import numpy as np

[ ] w = np.array([[1, 0.5], [2, 1]])
print ("# of axes- w = %d"%w.ndim)
print ("Shape of w = ", w.shape)
print ()

input = np.array([[1, 2], [-1, 2]])
print ("# of axes- input = %d"%input.ndim)
print ("Shape input = ", input.shape)
print ()

b = np.array([-2.0, 0.5])
print ("# of axes- b = %d"%b.ndim)
print ("Shape of b = ", b.shape)
```

of axes- w = 2
Shape of w = (2, 2)

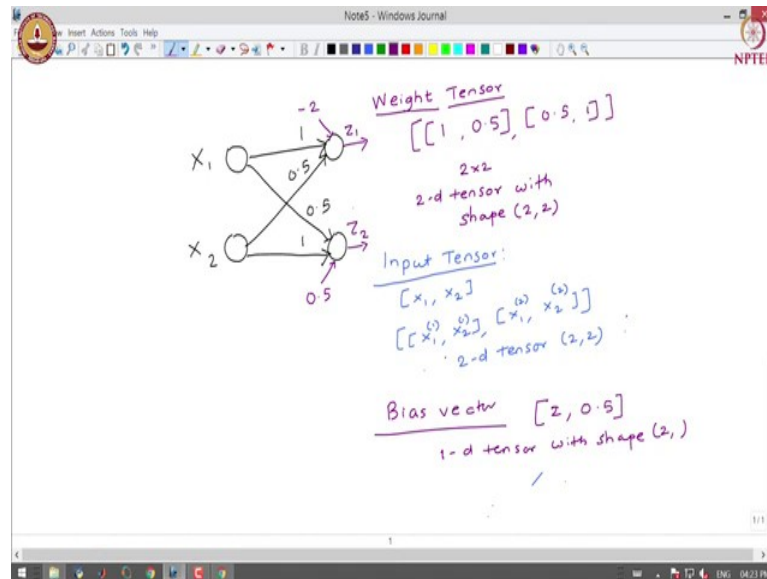
of axes- input = 2
Shape input = (2, 2)

of axes- b = 1
Shape of b = (2,)

Let's apply linear combination

Let us take a concrete example.

(Refer Slide Time: 02:12)



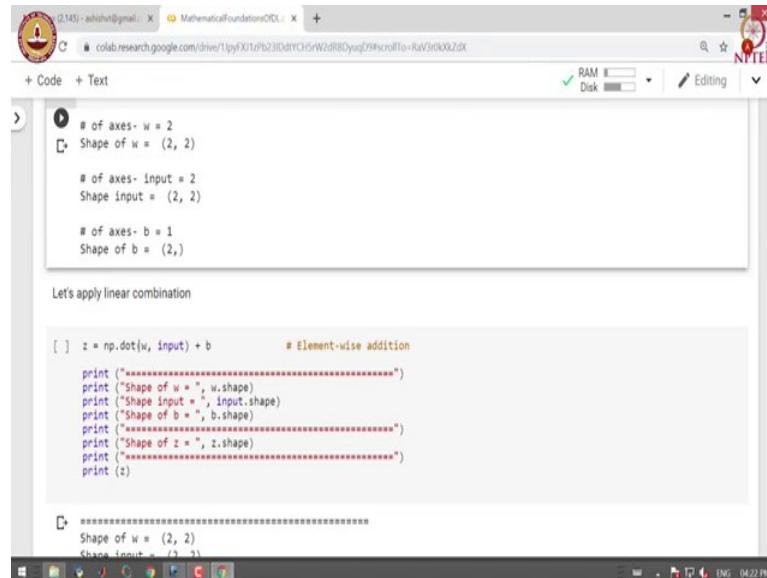
So, we have a toy neural network with two inputs let us say this is feature x_1 and this is feature x_2 . Then we have a layer with two units this particular layer is a dense layer, so it receives input from all the units from the previous layer. And let us set w are the parameters for each of the layers. So, for the first layer we have 1 and 0.5 as the weight and each of the unit has a bias term. This bias term is -2 for the first unit and 0.5 for the second unit. So, you can see that we can represent the weights of these units as tensor.

The first unit has weights 1 and 0.5 and the second unit has weight 0.5 and 1. So, you put this in a tensor. This is a matrix or alternatively this is a 2D tensor with shape 2 by 2. This is the weight tensor. You also have the input tensor and how does input tensor look like input tensor has two components x_1 and x_2 .

So, this is a vector. When we combine multiple examples it becomes a 2D tensor. We have a tensor which is again a 2D tensor which shape (2,2). In addition to that we also have a bias vector. So there is a bias unit for each of the hidden unit in the layer. We can represent the biases. Bias is a scalar quantity for individual unit, we combine all the biases we get a vector. So, we have 2 and 0.5 in this case this is the bias vector which is a 1D tensor with shape 2.

Let us say we have input, so we have w here which is $\begin{bmatrix} 1 & 0.5 \\ 2 & 1 \end{bmatrix}$; then we have input which is $\begin{bmatrix} 1 & 2 \\ -1 & 2 \end{bmatrix}$. So, let us look at the shape of each of these tensors and number of dimensions. So, you can see that w is a 2D tensor which shape 2×2 .

(Refer Slide Time: 07:21)



The screenshot shows a Google Colab notebook interface. At the top, there are tabs for 'MathematicalFoundationsOOL' and a search bar. Below the tabs, there's a 'Code' tab selected. The code cell contains the following text:

```
# of axes- w = 2
Shape of w = (2, 2)

# of axes- input = 2
Shape input = (2, 2)

# of axes- b = 1
Shape of b = (2,)
```

Below this, there's a text prompt: "Let's apply linear combination". Then, there's a code cell with the following code:

```
[ ] z = np.dot(w, input) + b          # Element-wise addition

print("*****")
print("Shape of w = ", w.shape)
print("Shape input = ", input.shape)
print("Shape of b = ", b.shape)
print("*****")
print("Shape of z = ", z.shape)
print("*****")
print(z)
```

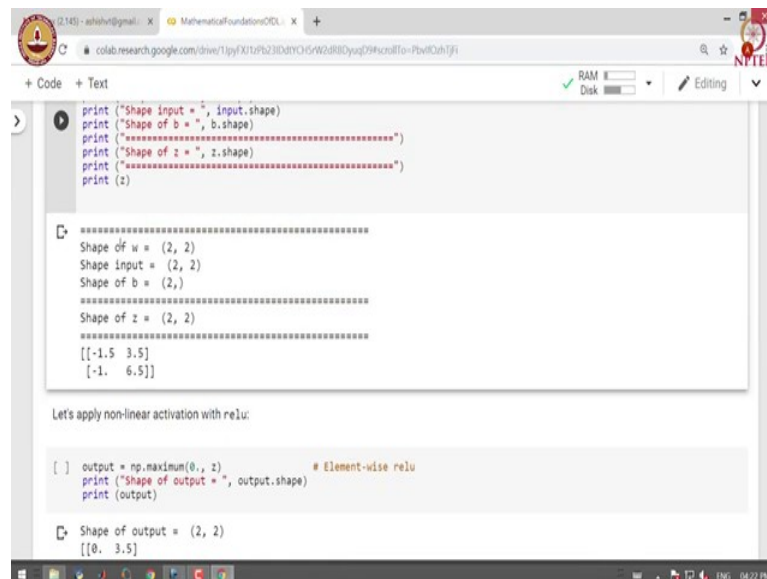
At the bottom, there's a code cell with the following output:

```
*****
Shape of w = (2, 2)
Shape input = (2, 2)
```

Input is also a 2D tensor which shape 2×2 . Here there are two examples and axes and bias is a 1D tensor with a shape of 2. We apply a linear combination. That means, we will perform a dot product between the weight vector and the input and we add a bias term to it.

Let us look at shape of each of the terms or each of the tensors that are involved here which is w , input and b and we also look at shape of the resulting tensor, z , and we will also print the value of z .

(Refer Slide Time: 08:04)



```
print("Shape input = ", input.shape)
print("Shape of b = ", b.shape)
print("=====")
print("Shape of z = ", z.shape)
print("=====")
print(z)

=====
Shape of w = (2, 2)
Shape input = (2, 2)
Shape of b = (2,)
=====
Shape of z = (2, 2)
=====
[[-1.5  3.5]
 [-1.   6.5]]

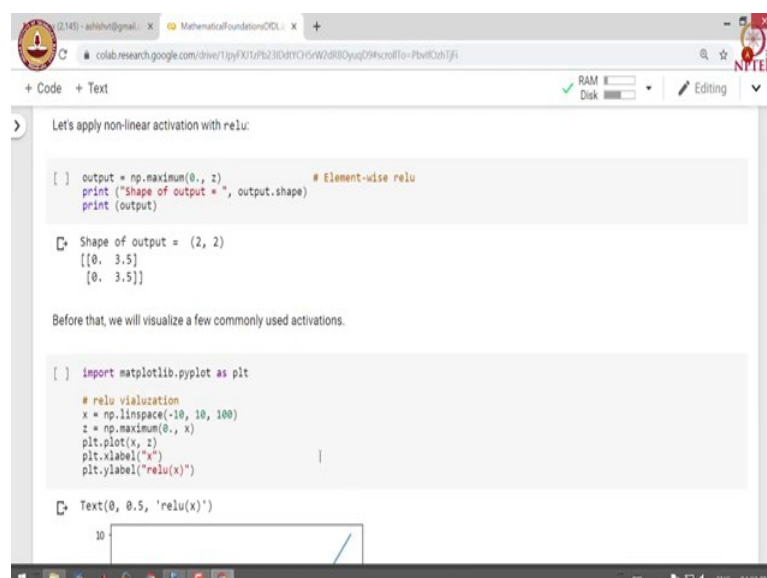
Let's apply non-linear activation with relu:

[ ] output = np.maximum(0., z)      # Element-wise relu
    print("Shape of output = ", output.shape)
    print(output)

Shape of output = (2, 2)
[[0.  3.5]
 [0.  6.5]]
```

You can see that all the tensors weight and a input tensors are 2D tensors with shape of 2 x 2 and bias is a vector which shape of 2, we get z as a 2D tensor which shape 2 x 2. So, we essentially get two outputs for each example. You can see that these two outputs are nothing but z coming from the first unit and z coming from the second unit. So, for every example that we input here we get z_1 and z_2 for each of the examples. So, here there are two examples so we get a 2D tensor with shape (2,2).

(Refer Slide Time: 09:21)



```
Let's apply non-linear activation with relu:

[ ] output = np.maximum(0., z)      # Element-wise relu
    print("Shape of output = ", output.shape)
    print(output)

Shape of output = (2, 2)
[[0.  3.5]
 [0.  6.5]]

Before that, we will visualize a few commonly used activations.

[ ] import matplotlib.pyplot as plt

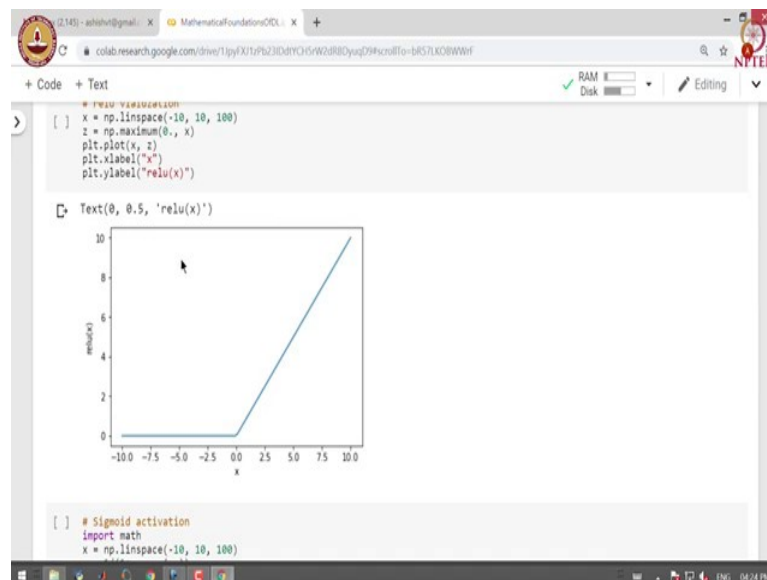
    # relu visualization
    x = np.linspace(-10, 10, 100)
    z = np.maximum(0., x)
    plt.plot(x, z)
    plt.xlabel("x")
    plt.ylabel("relu(x)")

Text(0, 0.5, 'relu(x)')
```

Now, let us apply Relu on the output of linear combination. So, relu essentially does a max between 0 and z . Let us print the shape of the output and the output itself.

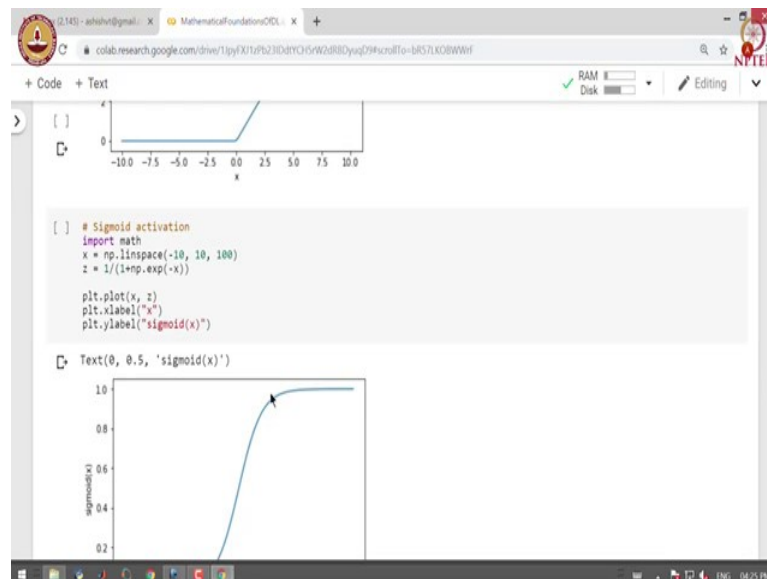
So, you can see that the output is also a 2D tensor of shape 2×2 and we get 0 when you apply relu on these values, because relu puts 0 for negative inputs and positive numbers are written as they are. Let us look at how relu activation function looks like.

(Refer Slide Time: 10:14)



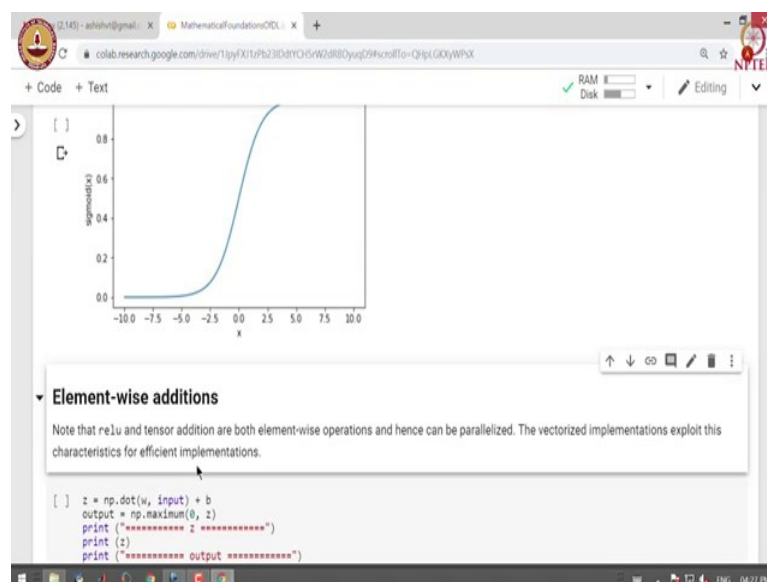
So, here we calculate the value of Relu between -10 to +10 at hundred samples. So, we have the value x on the x-axis and Relu on the y-axis. So, you can see that relu outputs 0 for the negative numbers and it outputs the positive number as it is. So we have a 45 degree line after 0.

(Refer Slide Time: 11:03)



In the same manner you also visualize the sigmoid activation function by sampling hundred points between -10 to +10 and we calculated sigmoid function for each of these points.

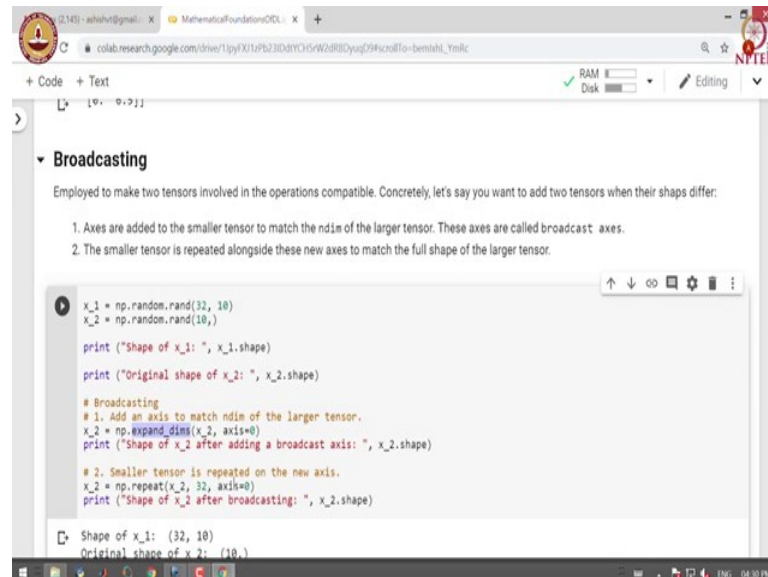
(Refer Slide Time: 11:15)



Sigmoid squashes the input between 0 to 1. As we go away from zero in the positive direction sigmoid tends to give 1 and as we go to the left of 0 in the negative direction, the sigmoid becomes closer and closer to 0. Now, note that Relu and tensor additions are both element wise operations and can be parallelized.

You will find vectorized implementation of these operations in deep neural network, that helps us to speed up the competitions. The vectorized implementation exploits the parallelization. Note that relu and tensor additions are both element wise operations and hence can be parallelized. The vectorized implementations exploit its characteristics for efficiency of competitions.

(Refer Slide Time: 12:24)



```
x_1 = np.random.rand(32, 10)
x_2 = np.random.rand(10,)

print("Shape of x_1: ", x_1.shape)
print("Original shape of x_2: ", x_2.shape)

# Broadcasting
# 1. Add an axis to match ndim of the larger tensor.
x_2 = np.expand_dims(x_2, axis=0)
print("Shape of x_2 after adding a broadcast axis: ", x_2.shape)

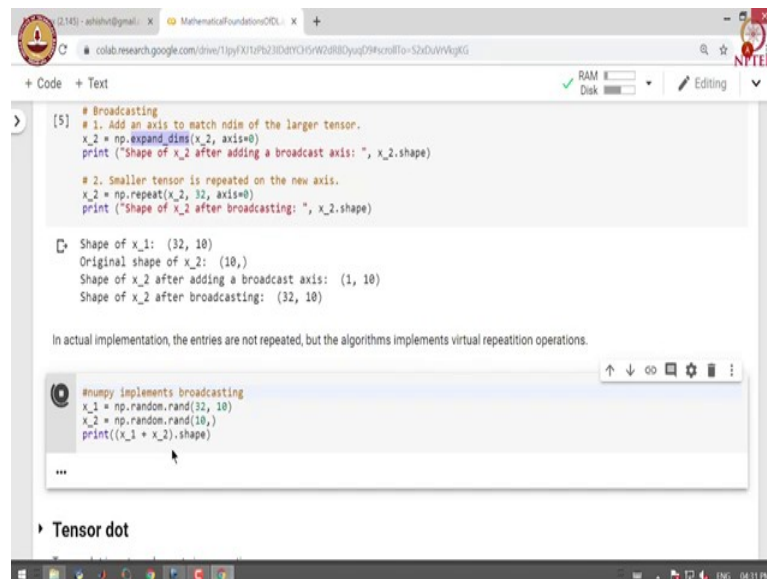
# 2. Smaller tensor is repeated on the new axis.
x_2 = np.repeat(x_2, 32, axis=0)
print("Shape of x_2 after broadcasting: ", x_2.shape)
```

Shape of x_1: (32, 10)
Original shape of x_2: (10,)

So, imagine a situation where we have to add two matrix which are not compatible in terms of the shapes, so this is where broadcasting helps us. In broadcasting we perform two steps so that two matrices become compatible in their shapes. We first start axes in the smaller tensor to match the dimension of the larger tensor these axes are called as broadcast axes. The smaller tensor is then repeated alongside these new axis to match the full shape of the larger tensor.

Let us take a concrete example. Let us say we have two tensors x_1 and x_2 . x_1 has shape of 32 x 10. Whereas, x_2 has a shape of 10. So the first tensor is a matrix and the second tensor is a vector. So, we will first expand the dimension of the second matrix along the zeroth axis. This will make sure that we have the same number of the same number of components on the zeroth axis and after that we repeat the x_2 tensor 32 times along the zeroth axis. So, that the shape of x_2 becomes fully compatible with the shape of x_1 .

(Refer Slide Time: 14:21)



```
# Broadcasting
# 1. Add an axis to match ndim of the larger tensor.
x_2 = np.expand_dims(x_2, axis=0)
print("Shape of x_2 after adding a broadcast axis: ", x_2.shape)

# 2. Smaller tensor is repeated on the new axis.
x_2 = np.repeat(x_2, 32, axis=0)
print("Shape of x_2 after broadcasting: ", x_2.shape)
```

Shape of x_1: (32, 10)
Original shape of x_2: (10,)
Shape of x_2 after adding a broadcast axis: (1, 10)
Shape of x_2 after broadcasting: (32, 10)

In actual implementation, the entries are not repeated, but the algorithm implements virtual repetition operations.

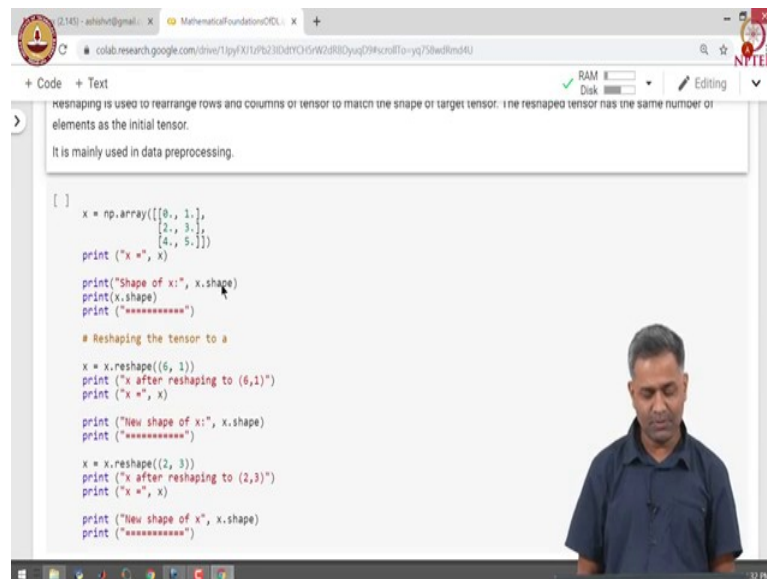
```
#numpy implements broadcasting
x_1 = np.random.rand(32, 10)
x_2 = np.random.rand(10,)
print((x_1 + x_2).shape)
```

Tensor dot

Let us look at the shapes of tensors that are involved here shape of x_1 is 32 x 10 the original shape of x_2 was 10. So, x_1 was a 2 D tensor and x_2 was a 1D tensor. After adding broadcast axis we get a 2 D tensor which shape 1 x 10 and then we broadcast x_2 along the broadcast axis to get a shape of 32 x 10. So, essentially you first add broadcast axis and then copied x_2 along each of those axis. In actual implementation the entries are not repeated, but algorithms implement these operations virtually.

So, *numpy* implements broadcasting. So if we try to add a 2D tensor with 1D tensor *numpy* automatically does the broadcasting and we can see the shape of the output which will be same as the shape of the 2D tensor.

(Refer Slide Time: 15:34)



```
resaping is used to rearrange rows and columns of tensor to match the shape of target tensor. The resaped tensor has the same number of elements as the initial tensor. It is mainly used in data preprocessing.
```

```
[ ]
x = np.array([[0., 1.],
              [2., 3.],
              [4., 5.]])
print ("x =", x)

print("Shape of x:", x.shape)
print(x.shape)
print ("*****")

# Resaping the tensor to a
x = x.reshape((6, 1))
print ("x after reshaping to (6,1)")
print ("x =", x)

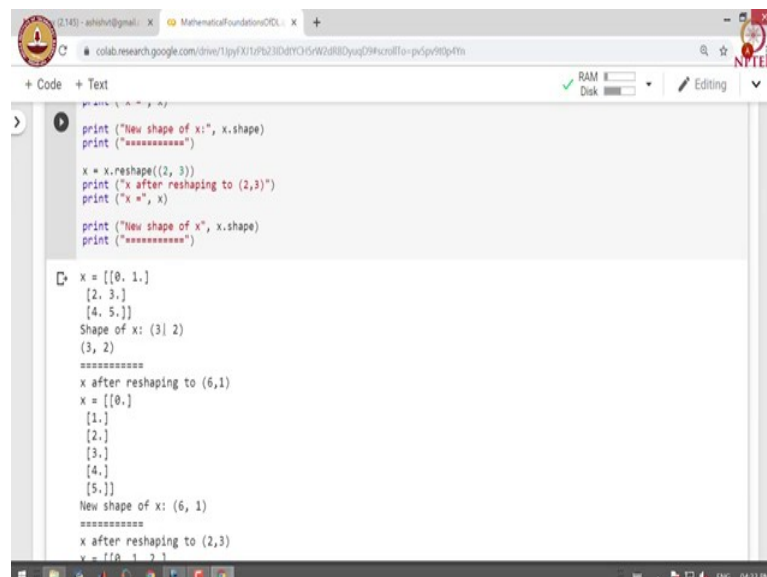
print ("New shape of x:", x.shape)
print ("*****")

x = x.reshape((2, 3))
print ("x after reshaping to (2,3)")
print ("x =", x)

print ("New shape of x", x.shape)
print ("*****")
```

Let us look at the next operation in a tensor which is reshaping, reshaping is used to rearrange the rows and columns of tensor to match the shape of the target tensor. The reshape tensor has the same number of elements as the initial tensor. Reshaping is mainly used in data preprocessing. So, let us take a concrete example of tensor.

(Refer Slide Time: 16:07)



```
print ("New shape of x:", x.shape)
print ("*****")

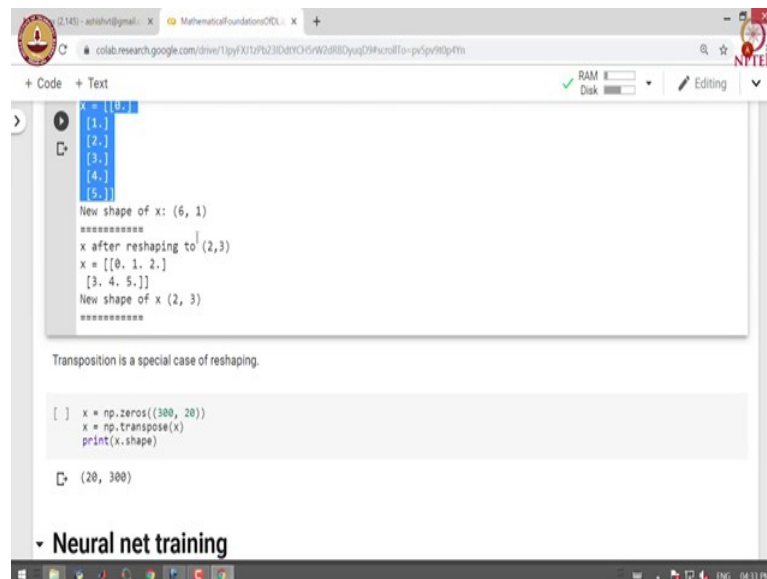
x = x.reshape((2, 3))
print ("x after reshaping to (2,3)")
print ("x =", x)

print ("New shape of x", x.shape)
print ("*****")
```

```
x = [[0. 1.]
      [2. 3.]
      [4. 5.]]
Shape of x: (3) 2)
(3, 2)
*****
x after reshaping to (6,1)
x = [[0.]
      [1.]
      [2.]
      [3.]
      [4.]
      [5.]]
New shape of x: (6, 1)
*****
x after reshaping to (2,3)
x = [[0. 1. 2.]
      [3. 4. 5.]
      [6. 7. 8.]
      [9. 10. 11.]
      [12. 13. 14.]
      [15. 16. 17.]]
New shape of x: (2, 3)
(2, 3)
*****
```

The shape of the tensor is 3 x 2 we are going to reshape it into a tensor of shape 6 x 1. So, to reshaping it to a tensor of shape 6 x 1 we get the we get a reshape tensor.

(Refer Slide Time: 16:28)



```
x = [[0.],
      [1.],
      [2.],
      [3.],
      [4.],
      [5.]]
New shape of x: (6, 1)
*****
x after reshaping to (2,3)
x = [[0. 1. 2.]
      [3. 4. 5.]]
New shape of x (2, 3)
*****

Transposition is a special case of reshaping.

[ ] x = np.zeros((300, 20))
    x = np.transpose(x)
    print(x.shape)

(20, 300)
```

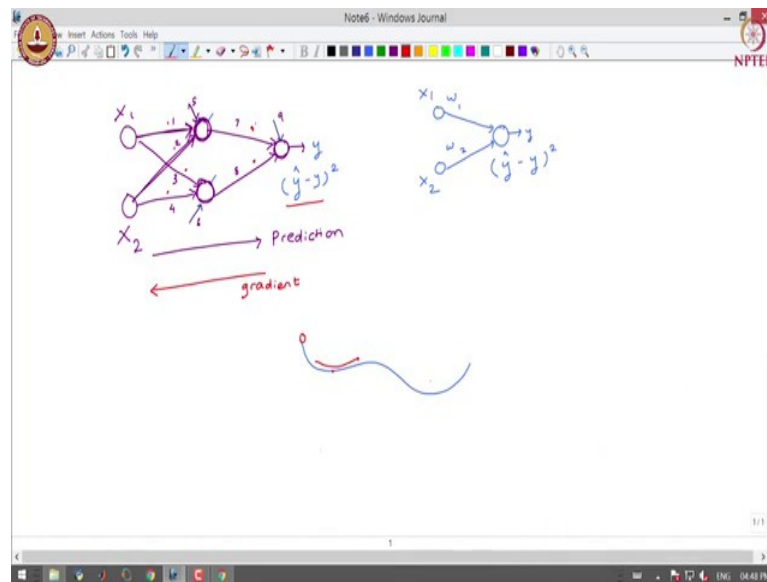
Neural net training

Later we reshape this tensor to a target tensor of shape 2 x 3, here what happens is that first three values are copied in the first row and the next three values are copied in the next row. So, this is how we get a (2, 3) 2D tensor.

Transposition is a special case of reshaping. In transpose rows becomes column and columns becomes rows. So, `np.transpose` does the transpose of the matrix the origin original shape of the matrix was 300 x 20, after transpose operation it became 20 x 300.

Having studied tensors and key operations on the tensor, let us move on to understand how neural network training is performed. We will first formulate the problem of training in neural network.

(Refer Slide Time: 17:33)



Let us take a toy neural network with two inputs one hidden layer with two units and an output layer with a single unit. These are the inputs x_1 and x_2 , these are two hidden units they also have bias term.

So, the problem of training here is to estimate the weights of each of the units in the neural network. This particular unit has three weights the weight corresponding with this connection and to a bias. So, there are three weights for this particular unit, in the same manner there are three weights corresponding to the this particular unit. This particular unit also has three weights one corresponding to each of the connection, there are two weights corresponding to the inputs coming from the previous layer and one bias term.

The problem of training is: given the training data you want to estimate the parameters of this neural network model. How many parameters are there? There are total nine parameters. In this particular network and our job is to come up with weights of these nine parameters such that the loss function is minimized.

We studied in previous sessions that we define loss function for each of the machine learning algorithm. So, in case of neural network, if you are solving a regression problem we use least-square as a loss function. If you are solving a binary classification problem we use `binary_crossentropy_loss` as the loss function and we have to find out this parameters such that the loss function is minimized.

We use gradient descent and stochastic gradient descent as basic techniques for solving the optimization problem or for parameter estimation. There is a difference between the way we apply gradient descent in standard machine learning algorithms and neural network. Let us try to understand that difference that will help us to appreciate the complexity of training neural networks.

So, along with this neural network, we will draw let us define a regression problem involving two variables: x_1 and x_2 . So, in this particular regression problem, we have two weights one corresponding to x_1 and the second is corresponding to x_2 . So, weights are w_1 and w_2 . The problem is to identify these weights, such that the least square error between the prediction and the actual output is minimized.

Remember that when you apply gradient descent, we find out the gradient of the loss function with respect to each of the parameters. In case of linear regression we get the output over here, the prediction over here, and once we get a prediction let us say \hat{y} is a prediction and if you know the actual y . We calculate the loss for that particular instance of the parameter value.

We calculate loss over here and we compute the gradient of the loss with respect to both the parameters in this case. Here the situation is slightly more complicated. We get the value of the y at the output layer. Here we can calculate the loss. Now, we have to compute the derivative of this loss with respect to the parameters and we do not know the loss at each layer or at each unit.

So, that makes it complicated. So we are getting the loss at the final layer and our challenge is to find out what is the contribution to the loss by individual units in the neural network. It is important to understand that this is the direction of prediction. It is called as forward pass. In forward pass we pass the values we perform linear combination followed by non-linear activation steps repetitively in each layer to get the prediction.

And once you get a prediction we can find out a loss and our job is to propagate this particular loss in the reverse direction. So, we calculate the gradient with respect to the direct connections to the output layer and then we apply chain rule of derivative successively to find out losses at intermediate levels and this is called as backpropagation. This is the direction in which gradients are propagated.

So, this is the backward pass, so we use a backpropagation to propagate that gradient of the loss function with respect to parameters in the network. So, in the modern deep learning packages like TensorFlow the gradient operation is already implemented using symbolic differentiation. So, we do not really implement backpropagation algorithm by hand. But instead we call this particular gradient API or gradient function to calculate gradient of the loss function with respect to each of the parameters. So, we do not focus a lot on back propagation in this course.

So, neural network implements a variation of stochastic gradient descent for faster optimizations. One class of optimization method focus on applying adaptive learning rate instead of using the same learning rate.

As in case of classical gradient descent they try to adopt the learning rate, so that convergence can be attained faster. Adam and RMSProp are optimization algorithms in this particular class. T

he other set of algorithms which are worth mentioning here are momentum-based algorithms. So, in case of deep learning the loss function is a non convex loss function. So, there is a great chance that we will get stuck into one of the local minima and we won't be able to come out of that if you apply classical algorithms.

So, here we use what is called as momentum based strategies, where we calculate the momentum at a point and use that to get out of the local minima. Think of momentum based algorithm using a ball and let us say you are sliding this particular ball on the slope if the ball has enough momentum it will not get stuck at a local minima. But it will get it will slide pass local minima and will go towards the next local minima.

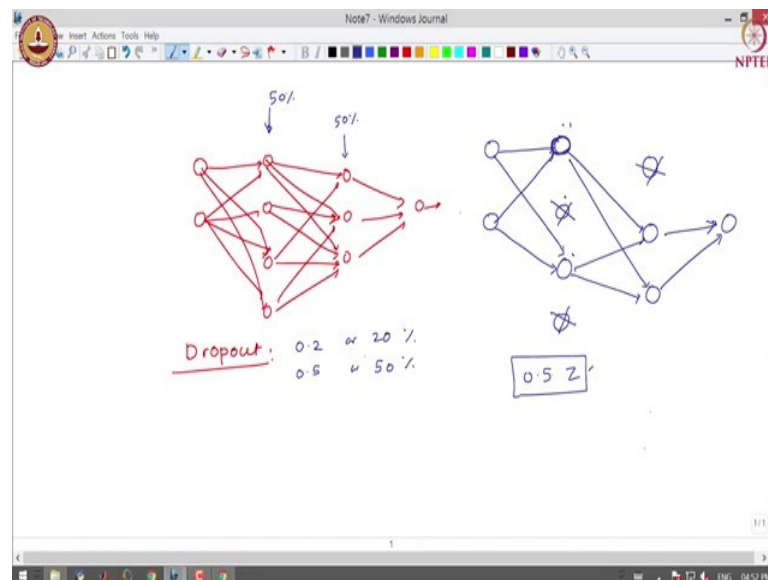
So, momentum based strategies are used to tackle problems with respect to local minima, that we often encounter in deep learning loss functions. We use some of the novel regularization strategies for neural network apart from l_1 and l_2 regularization that we apply in classical machine learning algorithms.

We also apply techniques like early stopping. In case of early stopping we apply a simple strategy of stopping training early. This helps us to prevent overtraining of the model. We keep track of the training error and the validation error and if validation error is not

improving after few iterations we stopped the training. So, this is possibly an automated way of deciding when to stop the training.

Apart from early stopping there is another clever idea called dropout which is popularly used as regularize mechanism in neural network. Let us understand idea of dropout through a concrete example.

(Refer Slide Time: 29:01)



Let us say this a toy neural network. In case of dropout, you define a dropout rate let us say dropout rate is 0.2 or 20 %. So, what happens is 20 % of the node in each layer where we define dropout are dropped during an epoch. So for example, let us define a dropout rate of 50 % at this layer. So, in one of the epoch what might happen is we look at the node and let us say we flip a coin if coin turns in the head we decide to return the node otherwise we drop it. So, let us say we decide to drop this node and this particular node. Let us say we also apply 50 % dropout in this layer. So, this could happen in one of the epochs, in the next epoch we again check we again decide to randomly shut down 50 % nodes in each of these layers.

In the second round it could be possible that this particular node and this node is dropped or this node and this node is dropped and you can see that every time we drop node we are effectively cutting the connections to and from that particular node. So, this gives us a new kind of a neural network architecture and we are training a lot of such kind of different architectures in one training iteration.

So we apply dropout during training. In order to compensate for drop out during prediction time we multiply the activation of each of the unit where drop out is applied by the factor of dropout. So, in this case what will happen is the activations out of this particular node we will be scaled by 50 %, because that was the dropout rate. So, this is how we apply dropout as a regularization strategy in case of neural network.

When we face the problem over fitting in neural network, we either try to get more examples or we apply one of these regularization strategies. Whenever we are faced with under fitting problem in neural network we can increase the complexity of the model by simply adding more layers to the network.

In this session, we looked at mathematical foundations of deep learning. We studied basics of tensors. The tensors that we encounter in practice the key tensor operations in deep learning and some of the basics of training and regularization in the deep learning. With this session, you are now well equipped with basic understanding of machine learning flow. From the next session onwards we will start diving deeper into practical or implementation aspect of machine learning pipeline with TensorFlow.