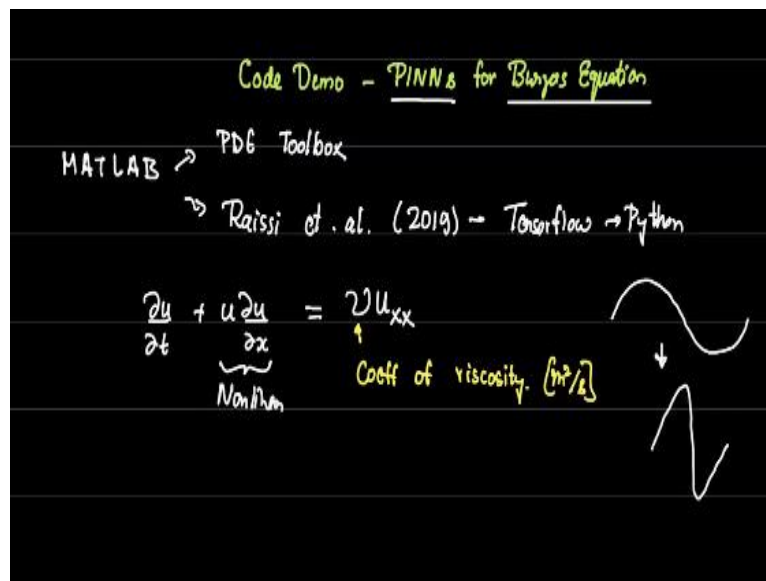


Inverse Methods in Heat Transfer
Prof. Balaji Srinivasan
Department of Mechanical Engineering
Indian Institute of Technology-Madras

Lecture - 64
Code Walkthrough for PINNs in Burgers Equation

Welcome back. We saw some very simple examples of code demos for neural networks in the last couple of videos. What we are going to do is do an actual full code demo of a physics informed neural network just as I had described last week. This is taken directly from MATLABs examples.

(Refer Slide Time: 00:41)



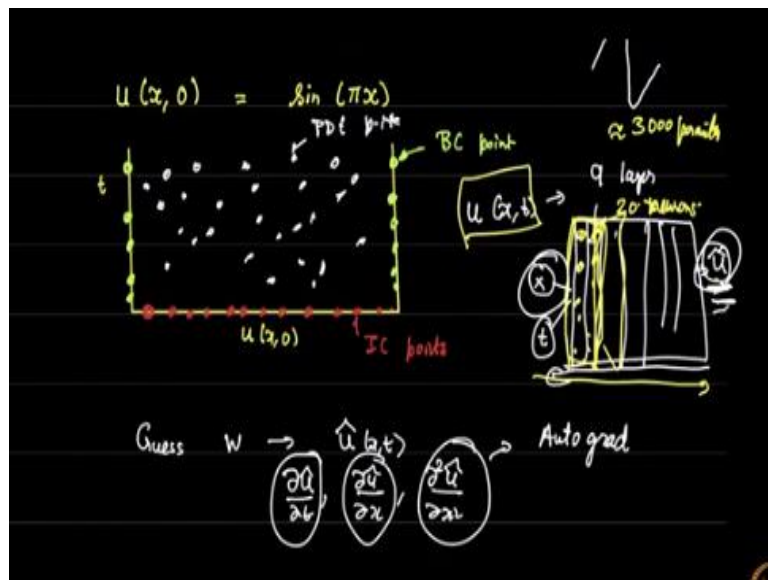
So, MATLAB has a PDE toolbox, which is available to you during the duration of this course in case you have taken this course for credit. Of course, the original paper from which this is, is the Raissi and Karniadakis paper that I had referred to in the last week. This is the 2019 paper. So, these people have, these researchers have given their direct code on TensorFlow.

The code is the code repository etc., are downloadable from their website. This is of course a Python code. But I am going to show the demo since we have been using MATLAB throughout, I will just simply show some salient points of this code. Now I have mentioned that this is for Burgers equation. So, Burgers equation is a PDE. The PDE is like this $\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x}$, so this is a nonlinear equation.

This term being nonlinear is equal to some constant times u_{xx} . Those of you who know the Navier-Stokes equation, which you should since you are taking this course as inverse methods in heat transfer, would know that this is known as some coefficient of viscosity or kinematic coefficient kinematic viscosity. In this case, this is sort of a pseudo coefficient. It is in case u represents velocity; its units are meter square per second.

This basically tells you what the viscous or the diffusion term is in this equation. Now typically, Burgers equation, even if you start smooth, can actually lead to, so even if you start with an initial condition like this, it can sharpen and become shocks, which is why it is sort of a good example for Navier-Stokes especially compressible Navier-Stokes equations. So, I am taking an example like I said, with some initial conditions that look like this, which is directly right out of the paper. So, this is the initial condition.

(Refer Slide Time: 03:05)



So, when you write the initial condition, you would write $u(x, 0)$ is some function, let us say $\sin(\pi x)$. I will show you the exact function the researchers used and indeed what we will be using for our code here. So similarly, we will use some specific value of μ , which the researchers used, but that is not sufficient. So, if we simply say that $u(x, 0)$ is given. We want the solution in time, okay so this is time.

So, you need some boundary condition here as well. So, you can use periodic boundary conditions or fixed boundary conditions, a lot of possibilities exist. I will show you the choice that the researchers made once again. Now these points here are the initial

condition points. As I told you during our PINN discussions, these can be fairly arbitrary, unlike when we use a finite difference or a finite volume solution.

Now these green points are the boundary condition points. These red points are the initial condition points. But that is not sufficient. We need a lot of PDE points. So, what does this mean? It means that you want to satisfy all these three in a least square sense. So let us say we arbitrarily put some points and we will call this PDE points. So, what is the idea? We simply say u I know is a function of x and t .

Instead of that, I represent this diagrammatically as x t a neural network. Now this is not one single hidden layer, but here is my output u or \hat{u} . So, we are going to take if I remember right, I will show you this in the code, we are going to take about nine layers. So, you are going to have nine hidden layers here and each one is going to have around 20 neurons, okay? So, this is what means.

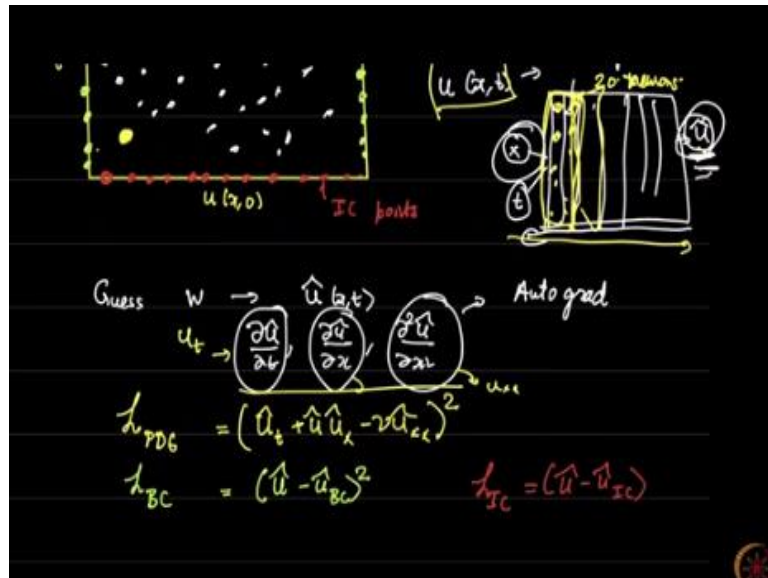
So, we have 20 neurons each or 20 to 25 neurons in each one of these layers. All this means is you have is a fairly complex function, which has a whole bunch of basically unknowns or a whole bunch of parameters. How many? So, you can see that every two layers, we are going to have approximately 400. 400 plus 400 multiplied by 8. So, you are going to have approximately 3000 parameters.

This is very few parameters honestly for a typical large neural network, not for a PINN network, but for a large neural network, these number of parameters are not very large. So, this is actually a reasonably sized neural network, just 20 neurons each with 9 layers, okay. So, once we have these what happens? So, you forward propagate through this, for some given set of parameters.

You should remember this from our, so for a guess of w the forward propagate you get u hat. But not only do you get u hat, you can actually estimate $\frac{\partial \hat{u}}{\partial t}, \frac{\partial \hat{u}}{\partial x}$. You can also estimate $\frac{\partial^2 \hat{u}}{\partial x^2}$. How do we do that? Once I know the function u , obviously, I can find out what $\frac{\partial \hat{u}}{\partial t}, \frac{\partial \hat{u}}{\partial x}, \frac{\partial^2 \hat{u}}{\partial x^2}$ is?

And the trick to this is to use auto grad or automatic differentiation. And automatically this is basically just like we achieve $\frac{\partial J}{\partial w}$ you can achieve $\frac{\partial \hat{u}}{\partial x}$ by a simple calculation through backprop, okay? But this backprop has a different purpose from the backprop that we usually use in neural networks that is to update the weights. This is simply to calculate what these terms are. So, once you do this, then you have loss terms.

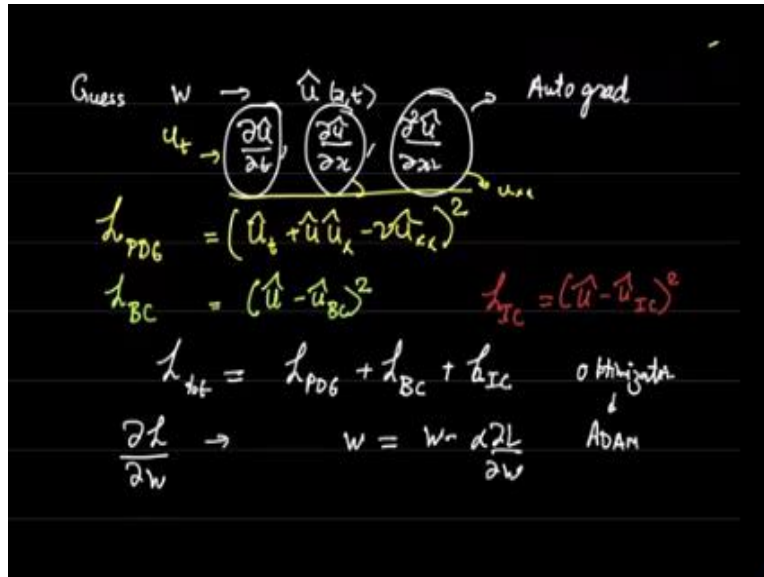
(Refer Slide Time: 07:32)



So, loss PDE (L_{PDE}) of course is, at every PDE point, you go here and calculate these three terms and see. Thus, I am going to call this for simplicity make up u_t, u_x, u_{xx} . So, I am going to simply check is $(\hat{u}_t + \hat{u} \hat{u}_x - \mathcal{U} \hat{u}_{xx})^2$, is this term 0 or not? Typically, obviously since we are randomly guessing at this point it is not going to be 0. So, u square this.

Now at the BC points, you go and check at the BC points, you check is u satisfying whatever the u boundary condition is. Similarly at the initial condition points you check is u satisfying the whatever my initial condition set, and we obviously have to square these, okay?

(Refer Slide Time: 08:42)



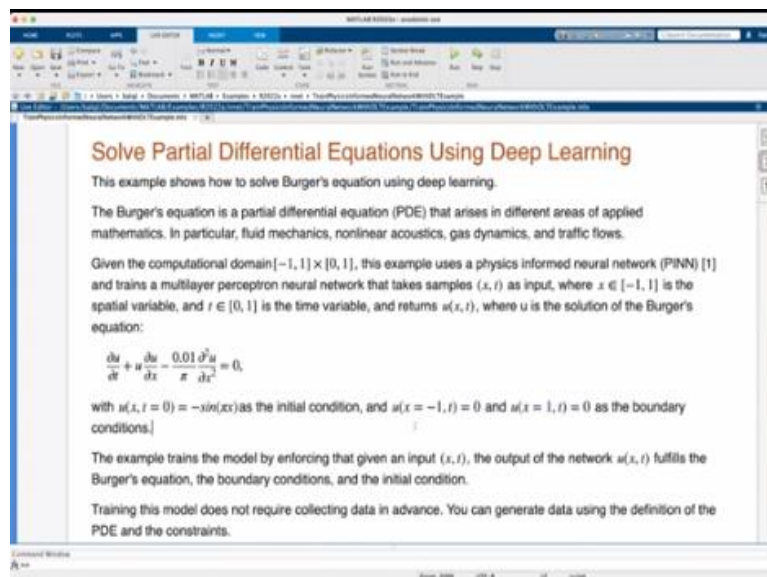
Then we can say that the total loss is let us call it,

$$L_{tot} = L_{PDE} + L_{BC} + L_{IC}$$

So, this is the loss. Then you find out $\frac{\partial L}{\partial w}$, okay? Calculate that, then $w = w - \alpha \frac{\partial L}{\partial w}$. The code that I am going to show you is the slightly different optimization scheme. It is still based on gradient descent, but it uses something called ADAM. ADAM is simply a different optimization scheme.

So, ADAM is an optimization scheme. That is a variant of gradient descent, okay? So, I will now show you the code and you can see the results within MATLAB. So, if you wait for a few seconds, I will show you the code.

(Refer Slide Time: 09:37)

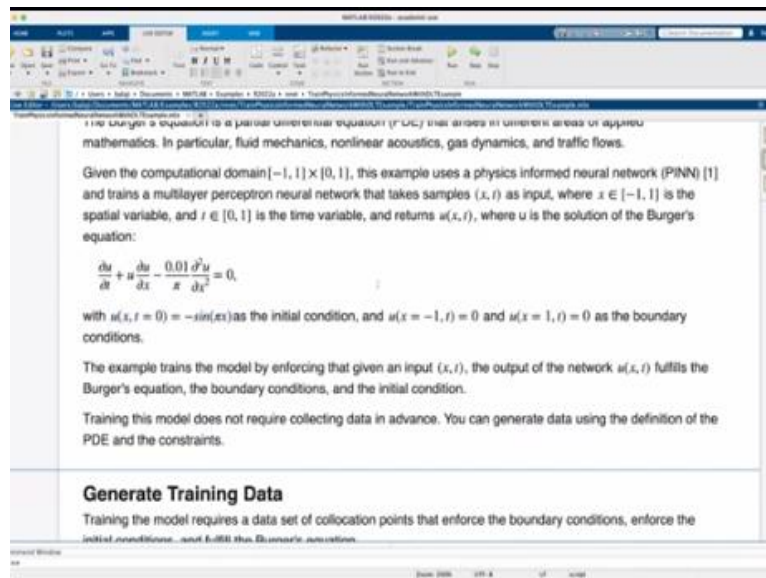


So, this is the example code which exists. So, I am not going to share this code online. You can basically go to MATLAB and look for this particular example, which is called the train physics informed neural networks example. So, I would request you to search that. Again, this will be available only during the duration of the course. The purpose of this code of course, is to show you how such codes look in practice.

So let us look at this. We have these three things. We have the PDE, which will bring us the PDE loss. You can see $u_t + uu_x$ equal to this term minus 0.01 by π this is new. 0.01 by π is just the viscosity coefficient. Now you can see that the initial condition was not $\sin(\pi x)$ as I had said, but it was $-\sin(\pi x)$ just to get a particular waveform which goes positive and then it goes negative as x becomes positive here, okay?

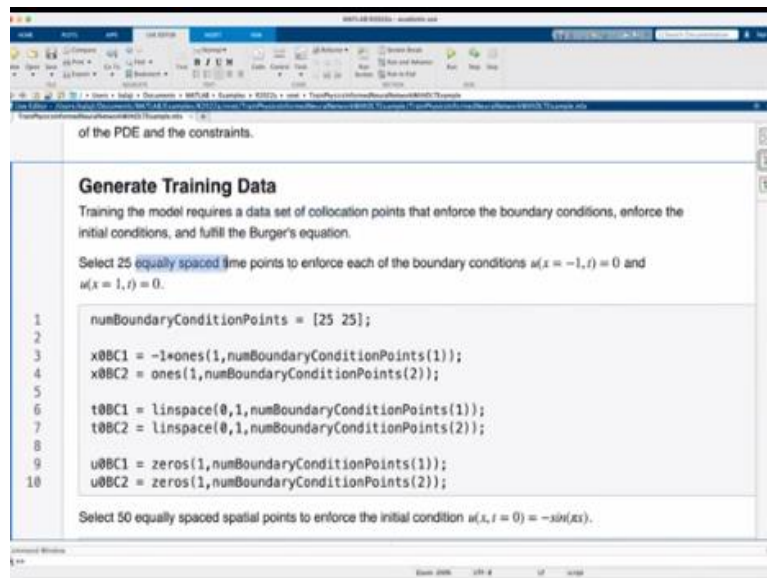
Now the boundary condition you can see here is that on the left and on the right, they have basically ensured that the velocity stays 0 if u is the velocity, okay?

(Refer Slide Time: 11:03)



So, you can see that we train the model by for a given input x t , if it lies at the boundary, you try to satisfy the boundary condition. If it lies at the initial condition you try to satisfy the initial condition and everywhere else you basically satisfy the PDE.

(Refer Slide Time: 11:19)



You do not require to collect data in advance. This this is an important point, okay. So, you do not require to collect data in advance, basically because the data is this. The only other data you will need is these points, initial and boundary conditions. But otherwise, you have PDE data, because we know the PDE is satisfied at every single point in the bundle.

So even though this says generating generate training data, all we are doing is collecting a set of points where we are going to impose the PDE, okay? So, you see that the boundary conditions at left and at the right, the green points that I showed you earlier in the video, these are enforced by keeping 25 equally spaced points on the left and on the right, okay? So now they are just collecting these.

This is the left boundary condition; this is the right boundary condition. Similarly, these are the temporal boundary conditions. Temporal boundary conditions meaning initial condition and the final condition. I did not draw this, this is at the top just like I had some condition at the bottom, they had put some other condition at the top, okay? So, the left BC is set to be 0, because that is what it is set to be here.

That is why this says u 0 BC equal to zeros. Similarly, on the right also you have zeros. So, rest of it are just MATLAB details that I will lead to.

(Refer Slide Time: 12:50)

```

6     t0BC1 = linspace(0,1,numBoundaryConditionPoints(1));
7     t0BC2 = linspace(0,1,numBoundaryConditionPoints(2));
8
9     u0BC1 = zeros(1,numBoundaryConditionPoints(1));
10    u0BC2 = zeros(1,numBoundaryConditionPoints(2));
11
12    Select 50 equally spaced spatial points to enforce the initial condition  $u(x,t=0) = -\sin(\pi x)$ .
13
14    numInitialConditionPoints = 50;
15    x0IC = linspace(-1,1,numInitialConditionPoints);
16    t0IC = zeros(1,numInitialConditionPoints);
17    u0IC = -sin(pi*x0IC);
18
19    Group together the data for initial and boundary conditions.
20
21    X0 = [x0IC x0BC1 x0BC2];
22    T0 = [t0IC t0BC1 t0BC2];
23    U0 = [u0IC u0BC1 u0BC2];
24
25    Select 10,000 points to enforce the output of the network to fulfill the Burger's equation.
26
27    numInternalCollocationPoints = 10000;

```

Initial conditions are 50 points, as I said here. So, you can see this. And the initial conditions are now set to $-\sin(\pi x)$, okay? So, 50 points here, 25 points each on the left and on the right, okay?

(Refer Slide Time: 13:07)

```

15    u0IC = -sin(pi*x0IC);
16
17    Group together the data for initial and boundary conditions.
18
19    X0 = [x0IC x0BC1 x0BC2];
20    T0 = [t0IC t0BC1 t0BC2];
21    U0 = [u0IC u0BC1 u0BC2];
22
23    Select 10,000 points to enforce the output of the network to fulfill the Burger's equation -- PDE points
24
25    numInternalCollocationPoints = 10000;
26
27    pointSet = sobolset(2);
28    points = net(pointSet,numInternalCollocationPoints);
29
30    dataX = 2*points(:,1)-1;
31    dataT = points(:,2);
32
33    Create an array datastore containing the training data.
34
35    ds = arrayDatastore([dataX dataT]);

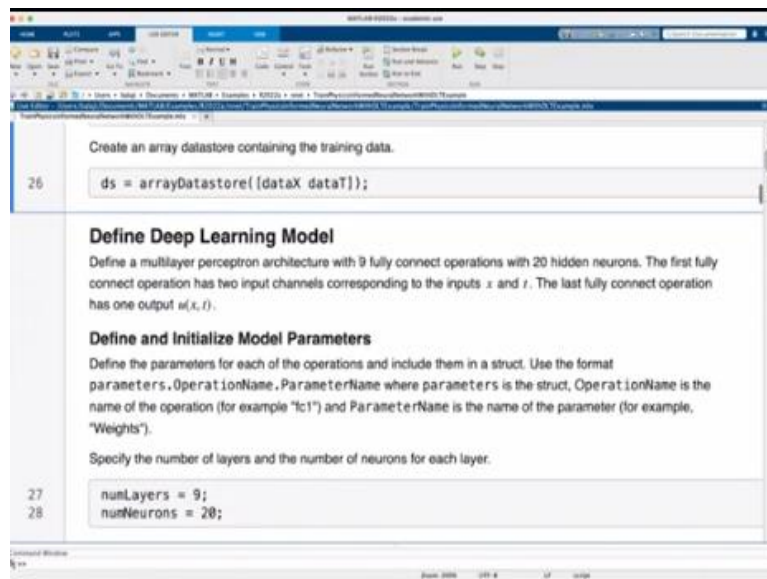
```

So, this now has been sent. Now these points are basically the PDE points that I talked about, okay? So, these are the PDE points. Those randomly spurned white points that I have shown in the video earlier, are basically these PDE points. And these are 10,000 points that are put in the entire domain, okay?

So, we have just randomly chosen, there is something called a sobolset. We do not care about that. So, once we put that we have got a whole bunch of points now. So, ds is the

data set, which is the bunch of points where you are going to impose. These are the x's and t's where we are going to impose the PDE.

(Refer Slide Time: 13:50)



```
26 ds = arrayDatastore([dataX dataT]);

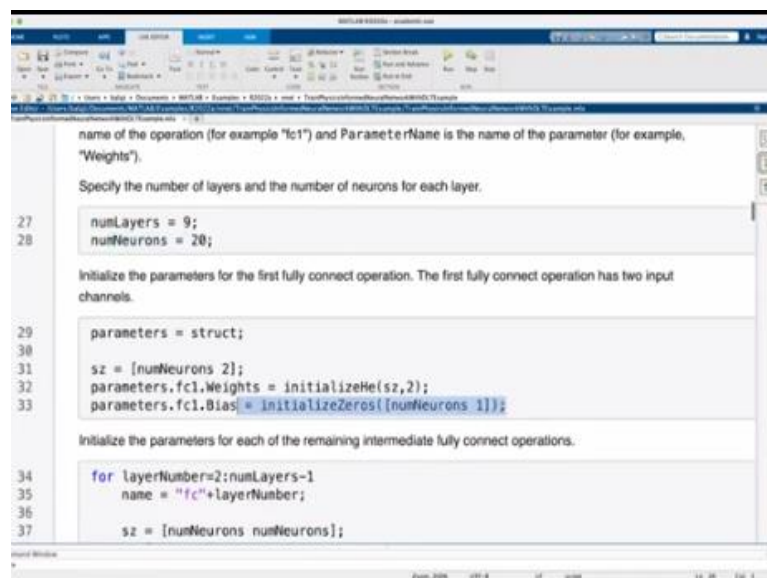
Define Deep Learning Model
Define a multilayer perceptron architecture with 9 fully connect operations with 20 hidden neurons. The first fully connect operation has two input channels corresponding to the inputs x and t. The last fully connect operation has one output u(x,t).

Define and Initialize Model Parameters
Define the parameters for each of the operations and include them in a struct. Use the format parameters.OperationName,ParameterName where parameters is the struct, OperationName is the name of the operation (for example "fc1") and ParameterName is the name of the parameter (for example, "Weights").

Specify the number of layers and the number of neurons for each layer.
27 numLayers = 9;
28 numNeurons = 20;
```

So now here it is. As I said, I seem to have remembered correctly, we have 9 fully connected operations with 20 hidden neurons each. So, this will be 3000 plus parameters, okay? So, you have x and t at the input, and you have just one single output on the outside which is u, okay?

(Refer Slide Time: 14:13)



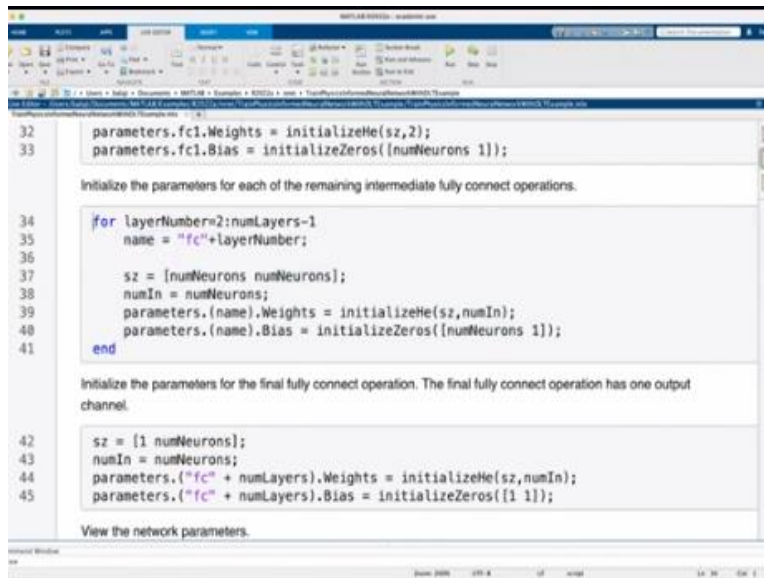
```
name of the operation (for example "fc1") and ParameterName is the name of the parameter (for example, "Weights").
Specify the number of layers and the number of neurons for each layer.
27 numLayers = 9;
28 numNeurons = 20;

Initialize the parameters for the first fully connect operation. The first fully connect operation has two input channels.
29 parameters = struct;
30
31 sz = [numNeurons 2];
32 parameters.fc1.Weights = initializeHe(sz,2);
33 parameters.fc1.Bias = initializeZeros([numNeurons 1]);

Initialize the parameters for each of the remaining intermediate fully connect operations.
34 for layerNumber=2:numLayers-1
35     name = "fc"+layerNumber;
36     sz = [numNeurons numNeurons];
```

So here you have number of layers is 9. Number of neurons is 20. So, I am going to skip this because now you can see weights, biases, etc. Defined here, there is a certain type of initialization. So, these are random initializations that we are giving for weights solved k initialization, okay?

(Refer Slide Time: 14:32)



```
32 parameters.fc1.Weights = initializeHe(sz,2);
33 parameters.fc1.Bias = initializeZeros([numNeurons 1]);

Initialize the parameters for each of the remaining intermediate fully connect operations.

34 for layerNumber=2:numLayers-1
35     name = "fc"+layerNumber;
36
37     sz = [numNeurons numNeurons];
38     numIn = numNeurons;
39     parameters.(name).Weights = initializeHe(sz,numIn);
40     parameters.(name).Bias = initializeZeros([numNeurons 1]);
41 end

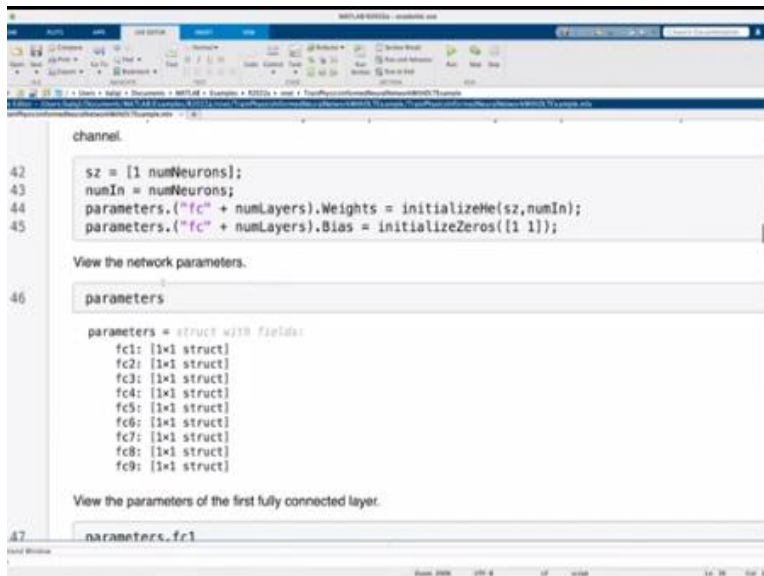
Initialize the parameters for the final fully connect operation. The final fully connect operation has one output channel.

42 sz = [1 numNeurons];
43 numIn = numNeurons;
44 parameters.(name).Weights = initializeHe(sz,numIn);
45 parameters.(name).Bias = initializeZeros([1 1]);

View the network parameters.
```

This exactly is just a neural network definition. Now when I showed you the XOR example, I was going step by step. And I think I hope at least that that was more readable than this. But if you know MATLAB or indeed if you go to Python and use one of these frameworks, you will see that that is reasonably understandable once you actually go ahead and implement a few codes, okay?

(Refer Slide Time: 15:00)



```
channel.

42 sz = [1 numNeurons];
43 numIn = numNeurons;
44 parameters.(name).Weights = initializeHe(sz,numIn);
45 parameters.(name).Bias = initializeZeros([1 1]);

View the network parameters.

46 parameters

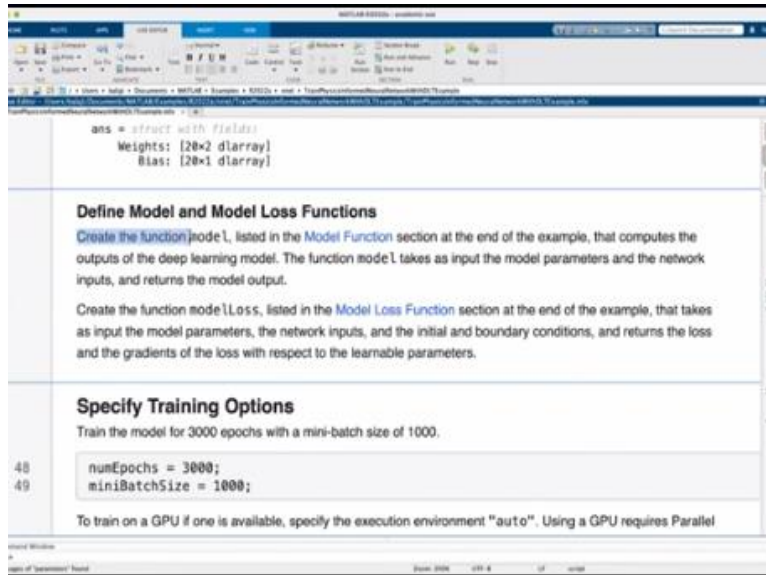
parameters = struct with fields:
    fc1: [1x1 struct]
    fc2: [1x1 struct]
    fc3: [1x1 struct]
    fc4: [1x1 struct]
    fc5: [1x1 struct]
    fc6: [1x1 struct]
    fc7: [1x1 struct]
    fc8: [1x1 struct]
    fc9: [1x1 struct]

View the parameters of the first fully connected layer.

47 parameters.fc1
```

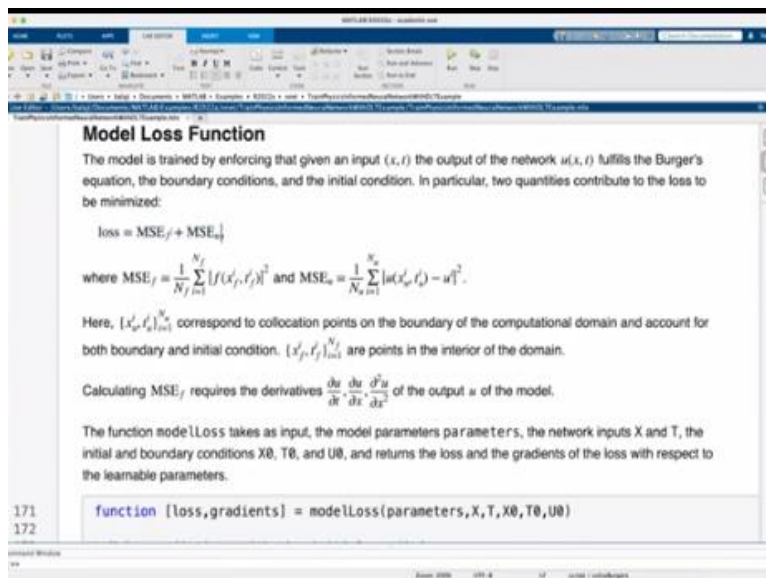
So here is simply some detail that shows you that a lot of fully connected layers are there and you have lots of weights in the middle, okay.

(Refer Slide Time: 15:11)



So now here is a model, so you have a model function.

(Refer Slide Time: 15:16)



You have a model loss function. Notice here, the model loss function is simply this MSE_f means the same as what I called L_{PDE} , the PDE loss. And MSE_u is basically all the terms of this sort where you have error from the boundary condition or the initial condition or if you have some data points also in the middle, you can add those as well. And here are of course, the PDE loss functions.

(Refer Slide Time: 15:48)

```

171 function [loss,gradients] = modelLoss(parameters,X,T,X0,T0,U0)
172
173 % Make predictions with the initial conditions.
174 U = model(parameters,X,T);
175
176 % Calculate derivatives with respect to X and T.
177 gradientsU = dgradient(sum(U,"all"),{X,T},EnableHigherDerivatives=true);
178 Ux = gradientsU(1);
179 Ut = gradientsU(2);
180
181 % Calculate second-order derivatives with respect to X.
182 Uxx = dgradient(sum(Ux,"all"),X,EnableHigherDerivatives=true);
183
184 % Calculate lossF. Enforce Burger's equation.
185 f = Ut + U.*Ux - (0.01./pi).*Uxx;
186 zeroTarget = zeros(size(f), "like", f);
187 lossF = mse(f, zeroTarget);
188
189 % Calculate lossU. Enforce initial and boundary conditions.
190 U0Pred = model(parameters,X0,T0);
191 lossU = mse(U0Pred, U0);
192
193 % Combine losses.

```

So, you can see here this is basically assuming some parameters are known. So, parameters here are just the w's. So, for a given x and t, you can do a forward prop, and that gives you u. That is what used to be our y hat. You can also calculate gradients, okay? So basically, ignore all this term, but basically you can calculate gradients, can calculate x gradient, you can calculate t gradient and you can also calculate the second gradient u_{xx} .

And here it is, here is our simple loss. Our simple loss is $u_t + uu_x - \mu u_{xx}$. This should be square and that sits in here, okay? We say that it has to be squared by using MSE. Similarly, initial and boundary conditions are put together and predicted here in just the single term loss u.

And we say that collect all these at all these points x_0, t_0 , which puts together all the points the initial condition boundary condition all those points and simply says, whatever prediction you made here, compare it with what I wanted the value to be and just add the mean square error and that will see the loss. And at this point, you basically calculate the gradient.

(Refer Slide Time: 17:09)

```

200 function U = model(parameters,X,T)
201
202 XT = [X;T];
203 numLayers = numel(fieldnames(parameters));
204
205 % First fully connect operation.
206 weights = parameters.fcl.Weights;
207 bias = parameters.fcl.Bias;
208 U = fullyconnect(XT,weights,bias);
209
210 % tanh and fully connect operations for remaining layers.
211 for i=2:numLayers
212     name = "fc" + i;
213
214     U = tanh(U);
215
216     weights = parameters.(name).Weights;
217     bias = parameters.(name).Bias;
218     U = fullyconnect(U, weights, bias);
219 end
220 end

```

This here is the forward model. Instead of using the sigmoid you can see that they have used the *tanh*. And that it is a fully connected layer in middle of every *tanh*. *Tanh* as I had told you in the nonlinearity chapter in the last week, that *tanh* tends to work better than sigmoid. A *tanh* works better because its derivative at 0 is 1 whereas the sigmoid derivative is 0.25.

And that tends to work worse as it goes through multiple layers. I had also told you that ReLU works even better. But the reason we did not use ReLU here is this term.

(Refer Slide Time: 17:52)

Solve Partial Differential Equations Using Deep Learning

This example shows how to solve Burger's equation using deep learning.

The Burger's equation is a partial differential equation (PDE) that arises in different areas of applied mathematics. In particular, fluid mechanics, nonlinear acoustics, gas dynamics, and traffic flows.

Given the computational domain $[-1, 1] \times [0, 1]$, this example uses a physics informed neural network (PINN) [1] and trains a multilayer perceptron neural network that takes samples (x, t) as input, where $x \in [-1, 1]$ is the spatial variable, and $t \in [0, 1]$ is the time variable, and returns $u(x, t)$, where u is the solution of the Burger's equation:

$$\frac{\partial u}{\partial t} + u \frac{\partial u}{\partial x} - \frac{0.01}{x} \frac{\partial^2 u}{\partial x^2} = 0,$$

with $u(x, t = 0) = -\sin(\pi x)$ as the initial condition, and $u(x = -1, t) = 0$ and $u(x = 1, t) = 0$ as the boundary conditions.

The example trains the model by enforcing that given an input (x, t) , the output of the network $u(x, t)$ fulfills the Burger's equation, the boundary conditions, and the initial condition.

Training this model does not require collecting data in advance. You can generate data using the definition of the PDE and the constraints.

The term u_{xx} , since ReLU is a linear function, u_{xx} will always turn to 0. So, if you have a u_{xx} term, you cannot use ReLU with any physics informed neural networks because

we want the second derivative to be nonzero. Obviously, if we give a model which does not give a second derivative at all, you are going to get u_{xx} is 0 by default everywhere.
(Refer Slide Time: 18:20)

```

// iteration = 0;
78
79 for epoch = 1:numEpochs
80     reset(mbq);
81
82     while hasdata(mbq)
83         iteration = iteration + 1;
84
85         XT = next(mbq);
86         X = XT(1,:);
87         T = XT(2,:);
88
89         % Evaluate the model loss and gradients using dlfeval and the
90         % modelLoss function.
91         [loss,gradients] = dlfeval(accfun,parameters,X,T,X0,T0,U0);
92
93         % Update learning rate.
94         learningRate = initialLearnRate / (1+decayRate*iteration);
95
96         % Update the network parameters using the adamupdate function.
97         [parameters,averageGrad,averageSqGrad] = adamupdate(parameters,gradients,averageSqGrad,iteration,learningRate);
98     end
99
100

```

So, once you do this, you have just a simple step here. You can see this number of epochs, etc. You go they also have a more fancy way of running this where the learning rate is actually updated. This is another variant of gradient descent. And you can see this term saying ADAM update. So, as I had told you, this is a variant once again of gradient descent. ADAM is an optimization.

So, you can see an ADAM solver etc. All the other terms are just in order to make this whole thing run, okay?

(Refer Slide Time: 18:58)

109

Evaluate Model Accuracy
 For values of r at 0.25, 0.5, 0.75, and 1, compare the predicted values of the deep learning model with the true solutions of the Burger's equation using the L^2 error.
 Set the target times to test the model at. For each time, calculate the solution at 1001 equally spaced points in the range $[-1,1]$.

```

110 tTest = [0.25 0.5 0.75 1];
111 numPredictions = 1001;
112 XTest = linspace(-1,1,numPredictions);
113
114 figure
115
116 for i=1:numel(tTest)
117     t = tTest(i);
118     TTest = t*ones(1,numPredictions);
119

```

And once we do that, you can just like the last time when we checked y versus y hat, you can now evaluate model accuracy if it runs fully. So, what I will show you now is we would not have the time to go through the full run, obviously this takes time.

(Refer Slide Time: 19:13)

```
210 % tanh and fully connect operations for remaining layers.
211 for i=2:numLayers
212     name = "fc" + i;
213
214     U = tanh(U);
215
216     weights = parameters.(name).Weights;
217     bias = parameters.(name).Bias;
218     U = fullyconnect(U, weights, bias);
219 end
220
221 end
```

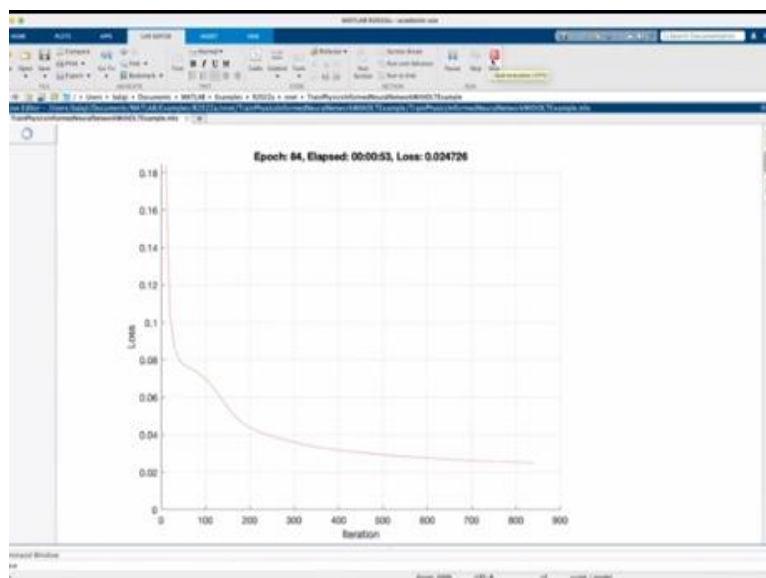
References

1. Maziar Raissi, Paris Perdikaris, and George Em Karniadakis | Physics Informed Deep Learning (Part I): Data-driven Solutions of Nonlinear Partial Differential Equations <https://arxiv.org/abs/1711.10561>
2. C. Basdevant, M. Deville, P. Haldenwang, J. Lacroix, J. Ouazzani, R. Peyret, P. Orlandi, A. Patera, Spectral and finite difference solutions of the Burgers equation, Computers & fluids 14 (1988) 23–41.

Copyright 2020 The MathWorks, Inc.

The references are given here at the bottom, okay? So this is the first paper is the Raissi and Karniadakis paper. Perdikaris is another professor at Penn State who works a lot on physics informed neural networks. So let me just run this quickly. And I want to show you just how these proceeds. We cannot show you the end. I will show you the end result on the MATLAB website. So hopefully this we see some results here.

(Refer Slide Time: 19:49)



So yes, you can now see the number of epochs and you can see it updating, okay? So you can see that the loss updates itself and generally tends to go down. You will see

some spike ups sometime in the middle. So as the loss goes down, this is why we typically plot the loss versus the number of iterations just to see that the whole thing is behaving as expected. So once the loss comes to one standard value, we say that the loss has saturated.

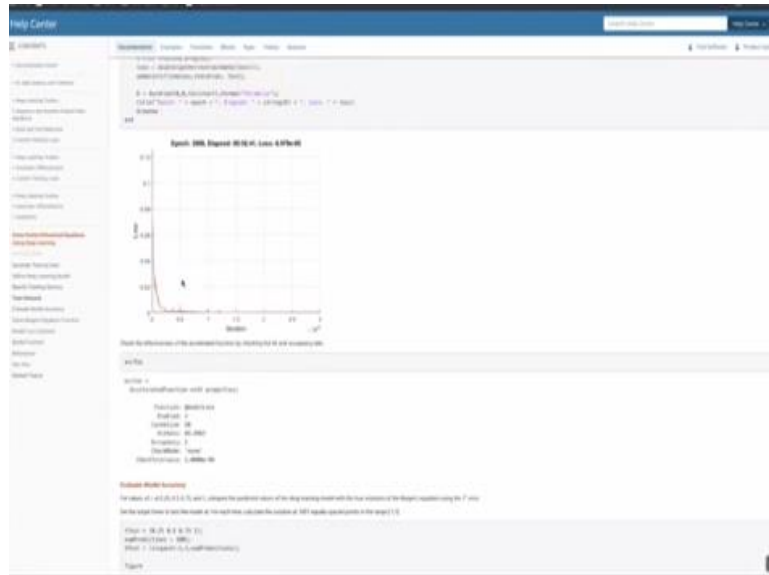
So, we have now about 75, 76 iterations here. I will right now stop it and show you the results on a different thing where on MATLAB's official website.

(Refer Slide Time: 20:37)



So, you can go to this website written here. If you go to the deep learning toolbox, and then look at solve partial differential equations using deep learning, this is actually available within MATLAB's website. If you say open in MATLAB online and during, in case you are taking this course for credit, during the course of this course or during the course of being enrolled in this NPTEL course, you will be able to open this in MATLAB and actually run it and you can check how it works. So, the same code here, I just want to show you the final results.

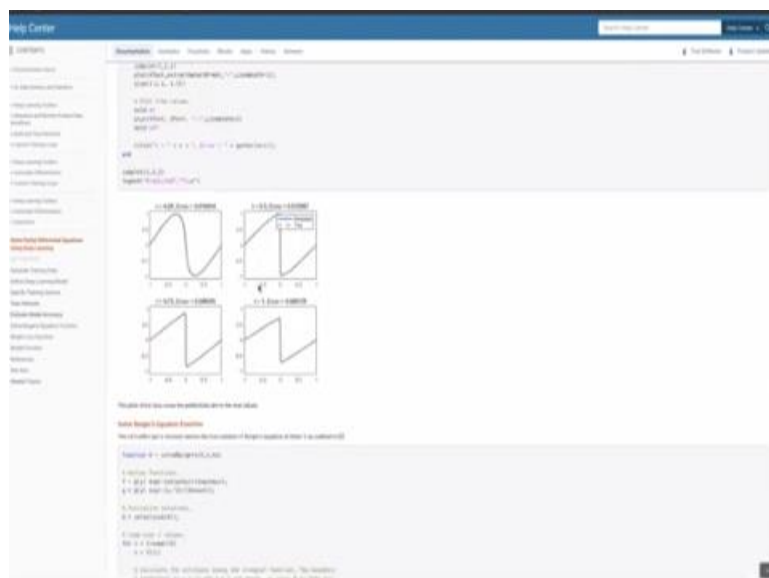
(Refer Slide Time: 21:18)



You can see here that this ran for 3000 epochs, because that was what the number of epochs was defined to be. You can see here number of epochs is 3000. Yeah, this is important. I had not mentioned it while showing you the code earlier that the mini batch size is 1000. Now what does mini batch size mean? Remember that we had a mini batch gradient descent, batch gradient descent and gradient descent.

So, we had 10,000 points here, but only 1000 of them will be taken each time for updating the loss function. That is what mini batch size of 1000 means here, okay. So, we run that. And you can see that after some time, the loss basically gets approximately to 0, at least in comparison to the original thing. You can see losses 10 power -5, okay? So, you can also see the final predictions.

(Refer Slide Time: 22:15)



The red lines here are the actual Ground Truth, which we never looked at by the way. Please remember that, in this case the way PINN works, we never actually looked at the Ground Truth, the red truth, because we were only looking at the derivatives. So that is the clever part. We did not impose Ground Truth on u , but Ground Truth on derivatives of u . So, we found out the relationship between derivatives of u .

You can see that the prediction is pretty good. So, in this video, you basically saw that PINNs can work pretty well even on a seemingly reasonably complex problem provided you give sufficient data and you give sufficient compute on this data. So, I will see you in the next video.