**Report on PyTorch and Plotly**

What is PyTorch?

As we all know it, PyTorch is the scientific computing package. Below are the few uses of PyTorch:

1. PyTorch supports native support for Python and the use of all its libraries.
2. It is actively used in the development of Facebook.
3. PyTorch guarantees a simple to utilize API which guarantees convenience and better understanding while coding
4. A Dynamic Computational Graph framework is a system of libraries, interfaces, and components that provide a flexible, programmatic, runtime interface that facilitates the construction and modification of systems by connecting operations. Which means that for every point of code execution, we can build the graph as we go along that can be manipulated at the runtime based on the needs.
5. Lastly, PyTorch has the support for Cuda. Cuda is basically Nvidia’s brainchild and their native technology. The support for Cuda ensures that the code can run on a graphical processing unit thereby decreasing the time. This increases the overall performance of the system. CUDA stands for computer unified device architecture.

PyTorch Installation:

One can install pytorch by running the following lines in the command prompt.

If using anaconda, try

conda install pytorch torchvision -c pytorch

Or

pip3 install http://download.pytorch.org/whl/cu90/torch-1.0.0-cp36-cp36m-win\_amd64.whl  
pip3 install torchvision

PyTorch- Concepts

We will first understand the basic PyTorch terminologies. Let us start with tensor.

1. Tensor is an imperative N-dimensional array running on GPU
2. Variables are nodes in the computational graph which is used to store the data and the gradients.
3. We have modules which are used in the neural network layer to store states. States are also called Learnable Weights.

Let's explore some of the simple code implementations using PyTorch.

1. tr=torch.rand(5,3) will generate a random matrix of 5 rows and 3 columns.
2. X=torch.zeros(5,3,dtype=torch.long) will generate a 0 matrix with 5 rows and 3 columns withdatatype long.
3. X=torch.tensor([5.5,3]) will create a tensor with data 5.5 and 3.

To find the size of the tensor, you need to type x.size() and this will return you the tensor size.

If you want to create a tensor based on an existing tensor, these methods will reuse properties of the input tensor, unless new values are provided by user.

x=x.new\_ones(5,3,dtype=torch.double)

x=torch.randn\_like(x,dtype=torch.float)

Also, we can perform all the tuple operations like addition,

Eg: y= torch.rand(5,3)

x+y

Or

torch.add(x,y)

Converting a torch tensor to a NumPy array or vice-versa is very much simple.

y=x.numpy() will return a NumPy array where x is a tensor

To convert it back,

x=torch.from\_numpy(y) will return a tensor back..

The NumPy Bridge

Tensors are very fast and efficient when compared to NumPy. For example, consider the below codes:

A=np.random.rand(3000,3000).astype(np.float32)

B=np.random.rand(d,d).astype(p.float32)

C=A.dot(B)

A=torch.rand(d,d).cuda()

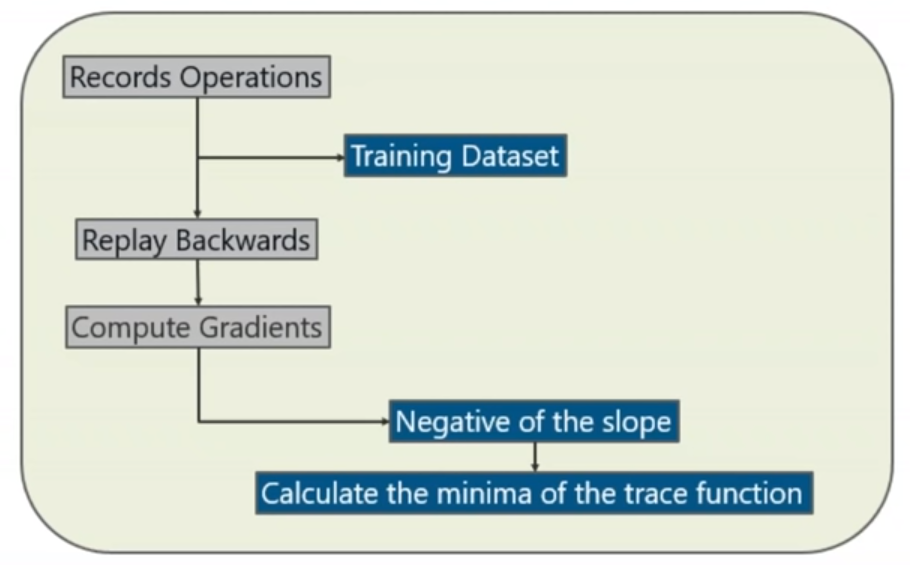
B=torch.rand(d,d).cuda()

C=torch.mm(A,B)

The first code is created using NumPy and simple operation such as matrix multiplication took 350ms. But in case of the second code, using tensors the same operation took 0.1ms.

AutoGrad Module

AutoGrad Module offers automatic differentiation for us.



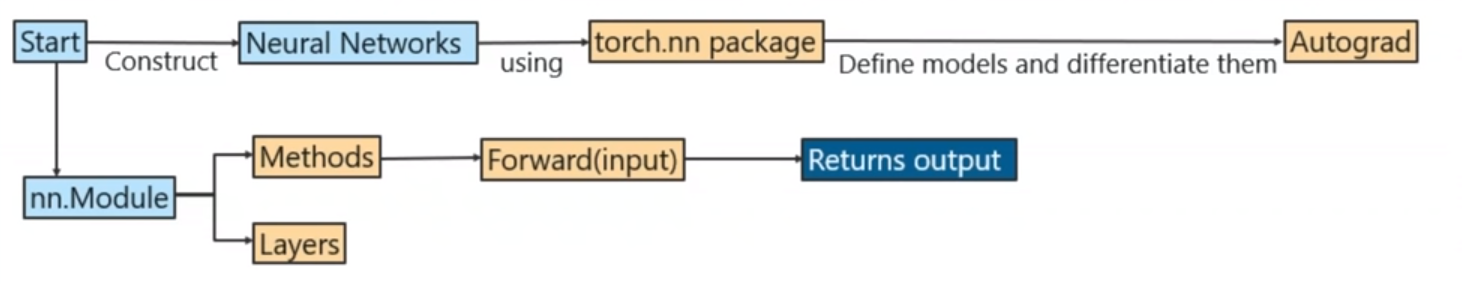
First, we record the operations based on the training dataset.

We replay backwards all the values in the data to reduce the losses at every stage. Before doing this, we need to compute the gradients.

Gradients are computed by finding the negative of the slope and calculating the minimum of the trace function of the graph obtained.

So, basically this saves time by calculating differentiation of all the parameters at forward pass.

Creating a Neural Network



1. First you construct the neural network. In our case, by using PyTorch’s torch.nn package
2. Here we define the models and automatically differentiate them. We will use the auto Grad Module for that purpose.
3. We have nn.Module which has methods and layers. In the methods, we have a method called as forward of input which is used to derive the output.
4. Here input is the parameter to the forward function. So, it is pretty much as simple as this, using the PyTorch package.

We have seen how to create a neural network in brief. Let’s deep dive more in the actual implementation.

Cancer Prediction:

This database can be found on UCI Machine Learning Repository: <https://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29>

It is also available through the UW CS ftp server: ftp ftp.cs.wisc.edu cd math-prog/cpo-dataset/machine-learn/WDBC/

Aim: The main aim of this project is to predict whether the patient has cancer or not based on several features. Attribute Information: 1) ID number 2) Diagnosis (M = malignant, B = benign) 3-32). The dataset has 569 rows and 33 columns. Features are computed from a digitized image of a fine needle aspirate (FNA) of a breast mass. They describe characteristics of the cell nuclei present in the image.

Pre-processing the data:

This is the first step for every machine learning project. Before we start building the model, we need to make our data suitable for processing. I have first divided my data into training and testing data using train\_test\_split from sklearn.model\_selection package. Here I have considered the 80-20 proportion for training and testing. Since my output can be either Malignant or Benign, I have manually substituted the values of Benign as 0.0 and Malignant as 1.0 using

op\_data=[0.0 if i=='B' else 1.0 for i in op\_data]

The next step is to standardize/ normalise the data. This is highly recommended else the model will not give good results. For doing this, I have used the inbuilt class StandardScaler from sklearn.pre\_processing.

scaler = StandardScaler()  
train\_data=scaler.fit\_transform(train\_data)

Building the model:

For this, I have created a new Class and then added the nn.Module class in the constructor. Then inside the init method, I have specified different hidden layers. So, my init method has different variables instantiated with the different activation functions.

def \_\_init\_\_(self, input\_dim,hidden\_dim, output\_dim):  
 super(LinearRegressionModel, self).\_\_init\_\_()  
 self.fc1=nn.Linear(input\_dim,hidden\_dim)  
 self.sigmoid=nn.Sigmoid()  
 self.linear = nn.Linear(hidden\_dim, output\_dim)

Since, the model is a sequential model, the output that I got from previous layer is then passed to the next layer. So, my forward\_layer is defined as below:

def forward(self, x):  
 out=self.fc1(x)  
 out=self.sigmoid(out)  
 out = self.linear(out)  
 return out

The output from the linear function is given to the Sigmoid Function and then from the Sigmoid function, it is given to another linear function and then the result is returned. We can have as many layers as we want.

After this, I have created an object ‘model’ and instantiated that to CustomModel class that I have defined above. I have also specified few more objects such as the loss function, learning rate and the optimizer.

criterion = nn.MSELoss()

learning\_rate = 0.01

optimizer = torch.optim.SGD(model.parameters(), lr=learning\_rate)

Training the model:

Here, I have selected epoch size as 2000. It means that the algorithm will run 2000 times in a loop. Now, for our model to work, it needs an input in the form of Variables. So first, we need to convert our tensors to Variable. For each run, I had mentioned optimizer.zero\_grad(). This is to clear information about the gradients from the previous run. After this, I had calculated loss byrunning below code:

Loss=criterion(outputs,labels)

Next step is to get the gradients with respect to the parameters. This is done by

Loss.backward()

The last step in training the model is performing optimizer.step(). This will update the parameters.

Making Predictions:

This step is very simple. All we need to do is pass the required test sample in the model method.

Eg: Predicted=model(Variable(torch.from\_numpy(x\_test)))

We can then calculate the accuracy by comparing the predicted values with the actual output.

Plotting:

Plotly is used for composing, editing and creating interactive data visualisation via the web.

If you visit the online website <https://plot.ly>, you can access tons of its features like customization and plotting. For example, you can use the chart studio through which you can make data on the fly. You can also use the white label feature to have editors and make your own small analytic products. You can also use a dash feature for making the analysis of the data. You also have a sql client through which you can upload data or fetch data from another SQL databases.

Now, you have 2 options to choose where you want to display the plot image. First is the online webpage, wherein you may have to register and create an account and then pass the userid and token for every request.

Eg:

plotly.tools.set\_credentials\_file(username='Praj3', api\_key='i20RRQ43GBPSvLkth')

The next step is plotting. Since I wanted to display multiple images in the same page, I have created subplots and specified all the individual titles. The next step is to create a trace. For that, you need to specify the x and y values and mention the typeof plot(Eg.go.scatter for scatter plot and go.Heatmap for matrix plot)

trace = go.Scatter(  
 x = np.arange(1,100),  
 y = np.arange(1,100),  
 name='residuals',  
 mode = 'markers'  
)

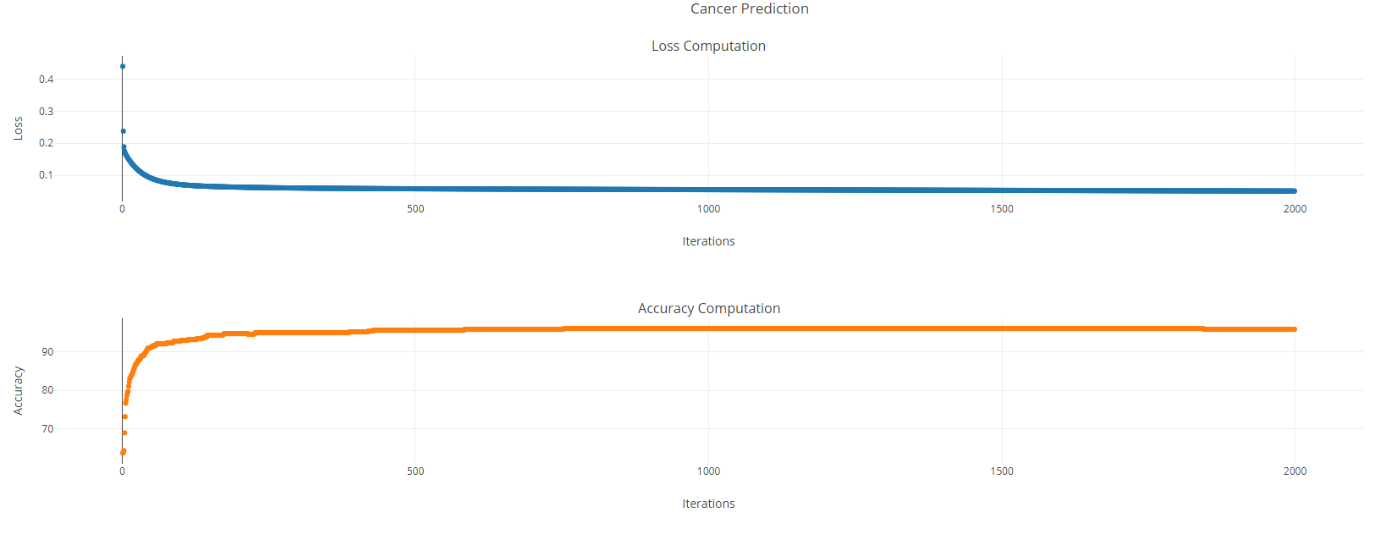
Now, you need to append the traces in the main fig. You can do this by typing below lines. Note that it will print row wise in this case.

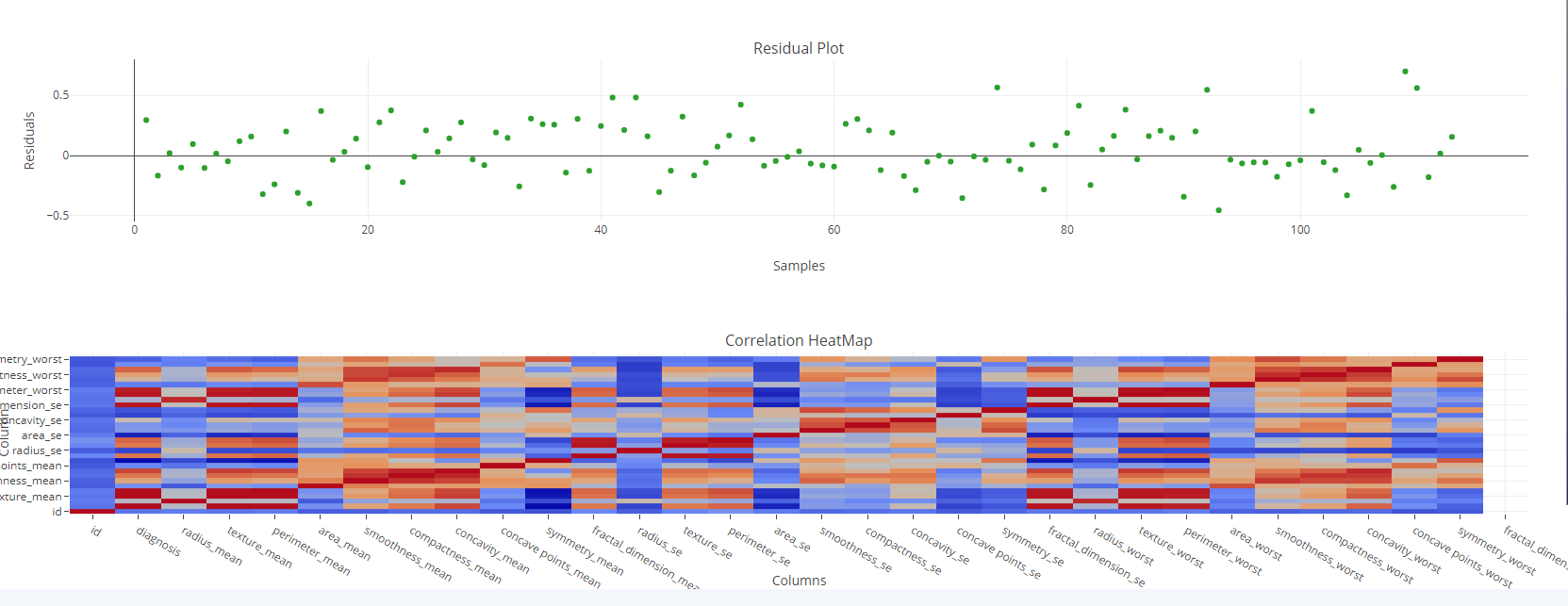
fig.append\_trace(trace1,1,1 )

fig.append\_trace(trace2,2,1 )

fig.append\_trace(trace3,2,1 )

Since, I have used the browser to display the plots, py.plot(fig,filename=’xyz’). This will open a new tab in the browser and show all the plots.

The first graph is the Loss Computation Graph and the second is the Accuracy graph



The difference between the actual value and the predicted value is called as residuals. The 3rd plot shows the residual plot. The 4th plot is the correlation matrix plot between all the columns. The darker the colour more is the correlation between them

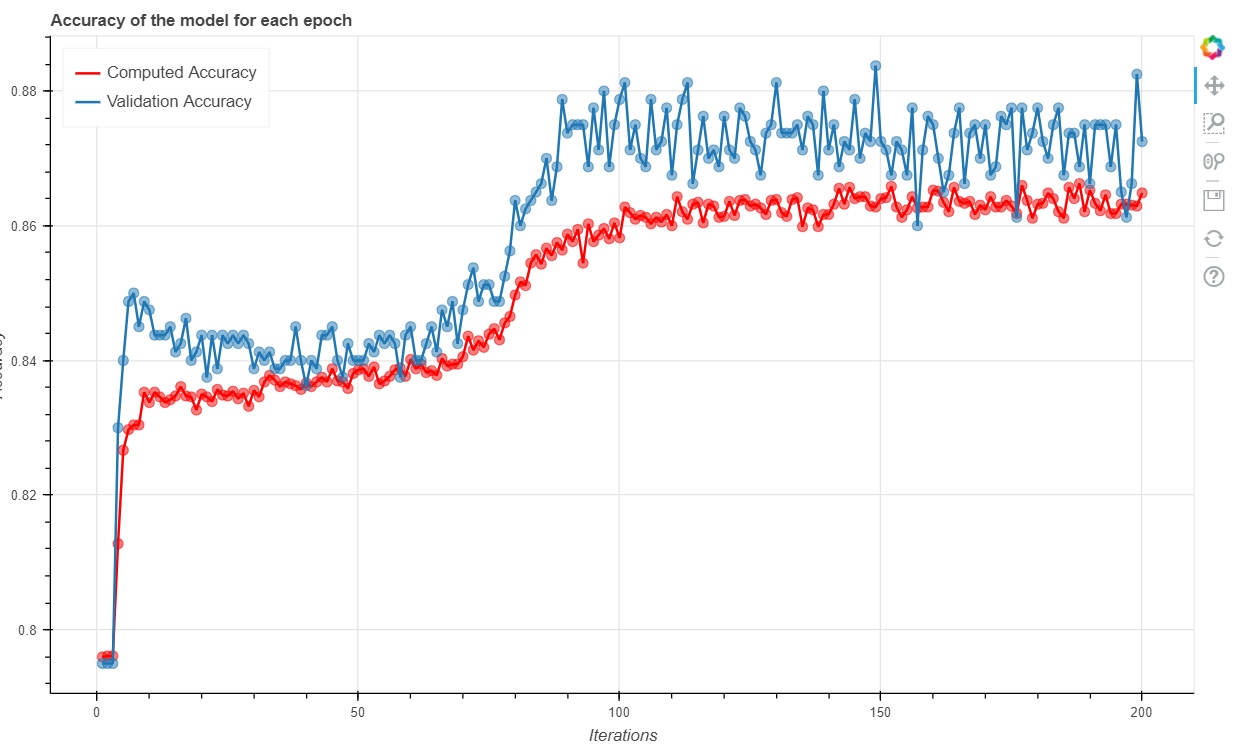
Comparison:

I have also performed the prediction of Churning Model using PyTorch. Earlier, I had used keras for doing the prediction. The steps that I followed are almost the same that I have mentioned above. That’s the beauty of pytorch and keras. You just have to pre-process the data, specify some of the parameter metadata and pass that to the model. The internal working (Computing the loss, Computing the accuracy, etc.) are all internally performed by the model itself.

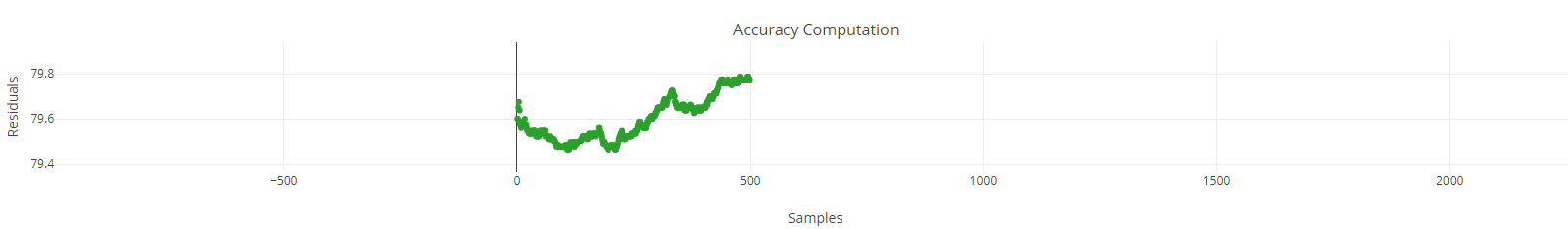
For the analysis, I have visualised similar data for both plotly and bokeh. However, I found that plotly is much more convenient to use and simpler than bokeh. The latest version of bokeh specifically needs a lot of coding to plot even a simple boxplot.

Regarding the accuracy and loss, I found that both keras and PyTorch gave almost similar results if the number of hidden layers and activation function are same. So, I changed some of the configurations and compared the results and I found a slight difference. So, I feel that both the keras and PyTorch give similar outputs when all the other parameters are same. It’s just that Keras is simple to code.

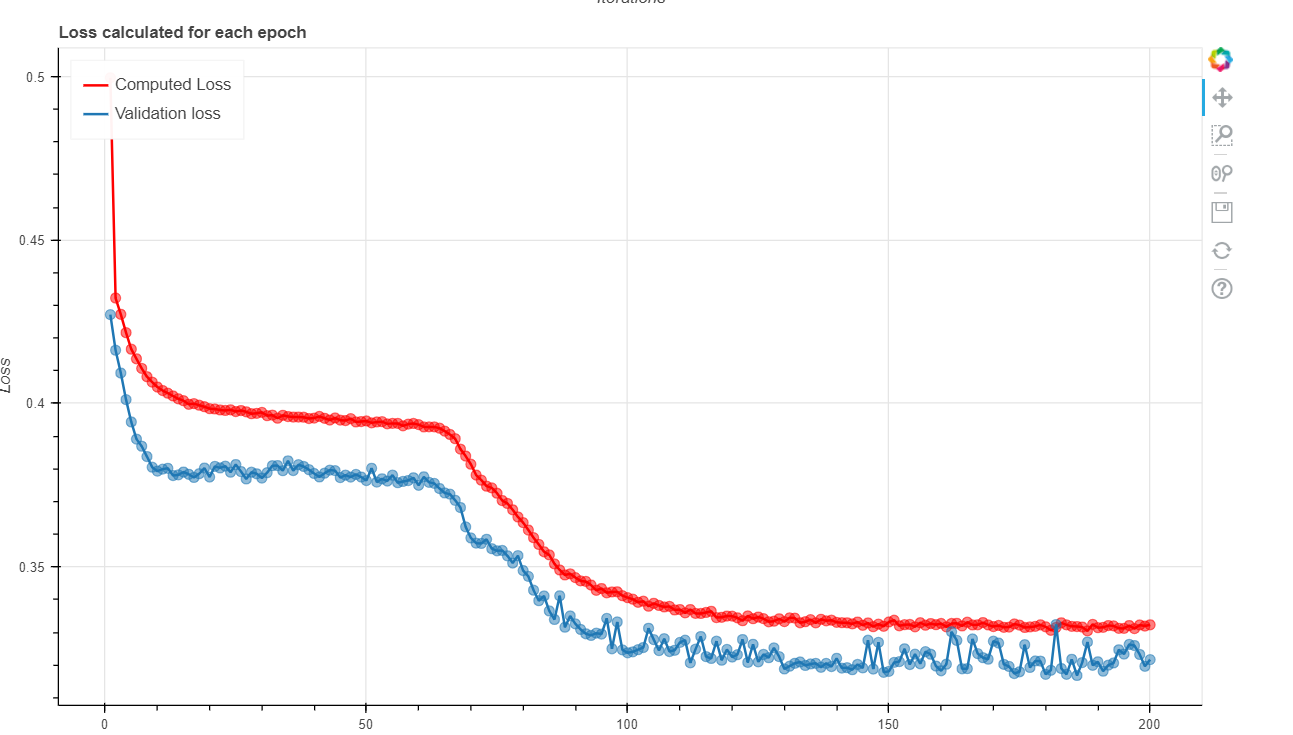
Plot Comparison:



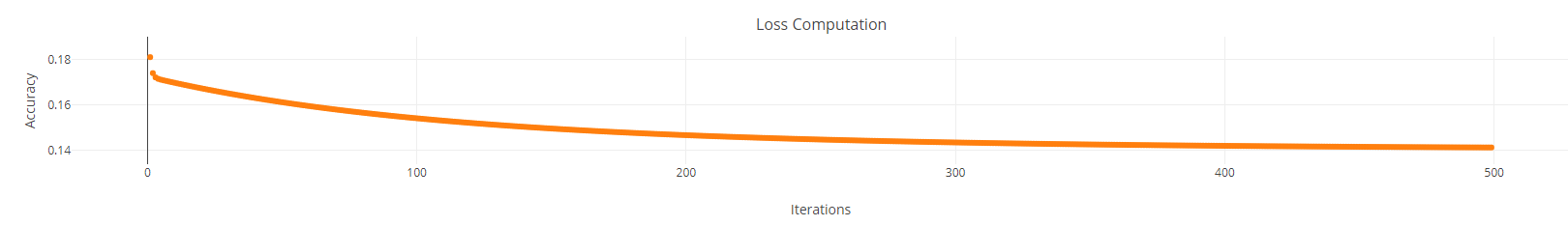
Accuracy Plotting using Bokeh For Churn Prediction Model



Accuracy Plotting using Plotly For Churn Prediction Model



Loss using Bokeh For Churn Prediction Model



Loss using Plotly For Churn Prediction Model

Conclusion:

Compared with keras, PyTorch is newer, yet it can perform almost anything. Similar to keras, it hides the internal implementation thereby making the code look tidier. I feel plotly is clearly the winner when compared with bokeh. Plotly does not require you to write complex lines of code for visualization.