Project-1

Fully Connected Neural Network

**Objective:**

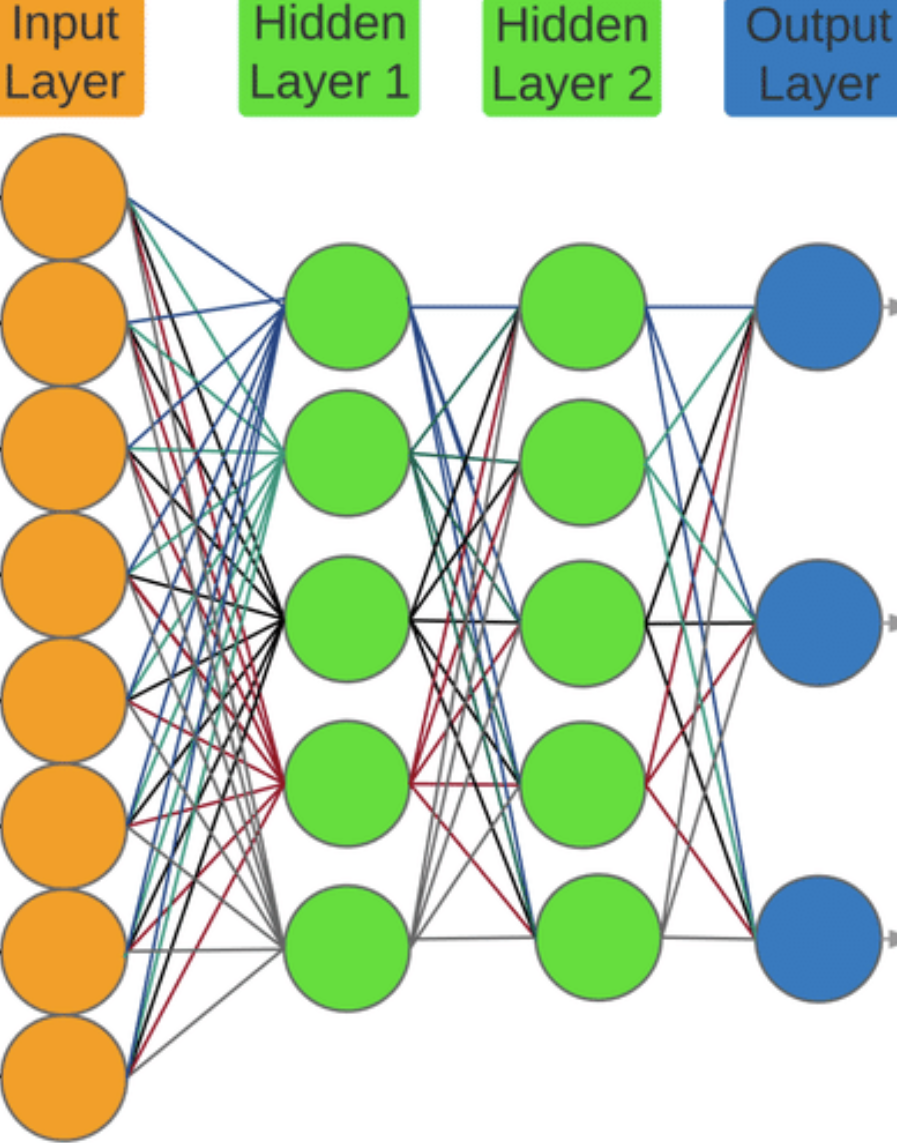
Build and train a Feedforward fully connected neural network with three layers (including the output layer) using low level python program, to perform classification of Iris and MNIST datasets and analyze the results from different parameters.

**Introduction:**

An artificial neuron is a replication of our biological neuron, which takes input and one or more inputs and sums them to generate an output activation. Several such neurons are connected to form an artificial neural network.

Feedforward neural network is simplest type of artificial neural network which propagates in forward direction only without looping.

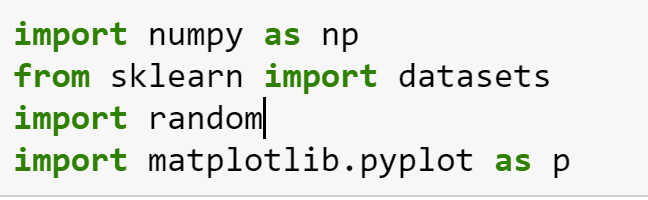
This network has one input layer, hidden layers(optional) and output layer.



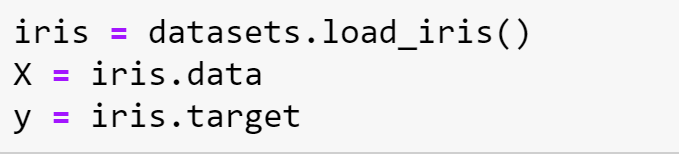
Each neuron in one layer has directed connections to the neurons of the subsequent layer. Each input is separately weighted, and the sum is passed through a non-linear function known as an activation function. Here, *ReLU* and *Softmax* activation functions are used for hidden layers and output layer, respectively and Weights are generated using *Kaiming initialization*. Network learns through *back-propagation* and weights are adjusted using *gradient descent* method.

**Question 1: Iris dataset species classification**

1. Import below libraries

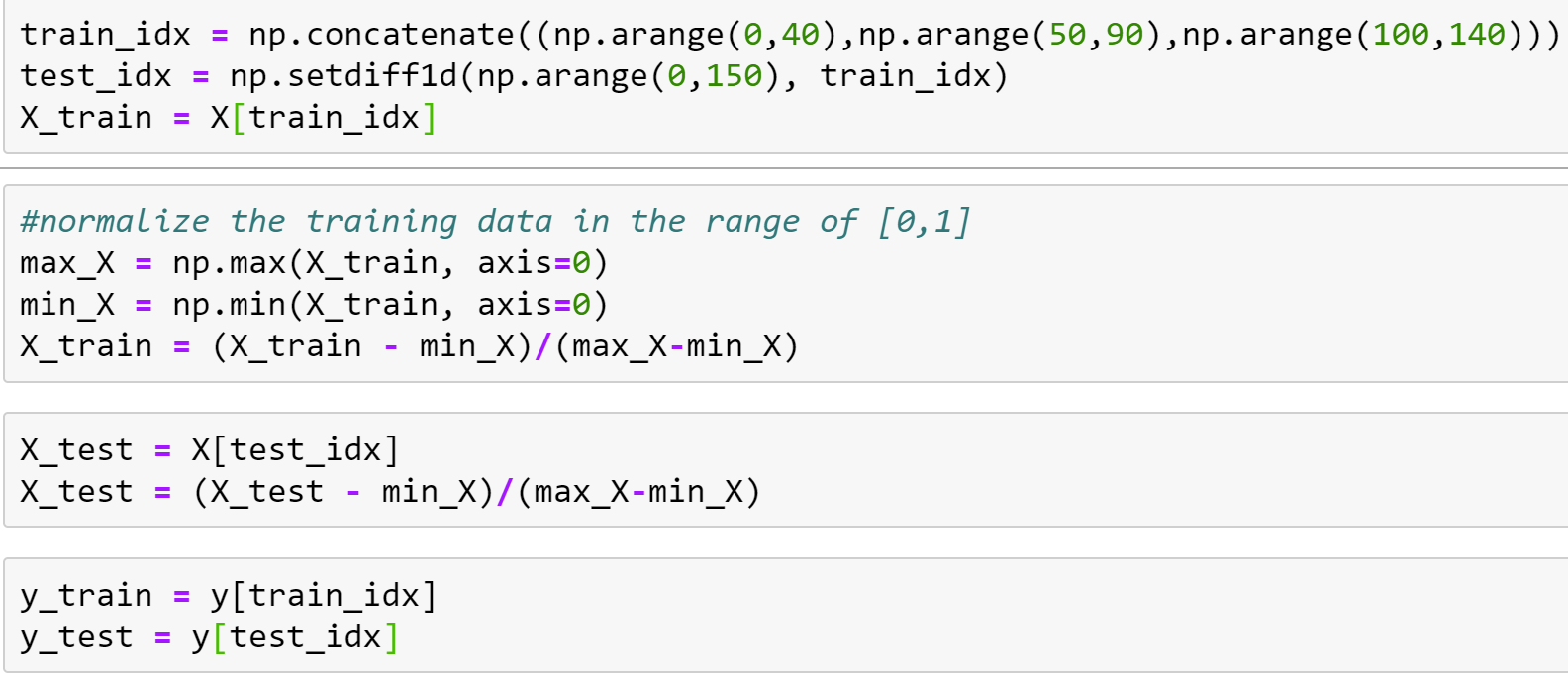


2. **Load Iris dataset** from sklearn.datasets module



3. **Train-test split**

Dataset contain 150 samples, divide the samples into 120 training samples and 30 test samples.

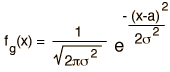
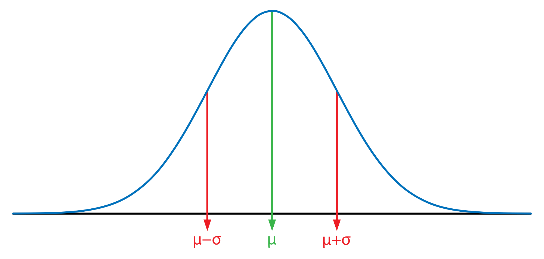


*Below are few functions that comes handy during model building.*

4. Weights Initilization:

In this model, we have 3 layers including output layer. So, we can code a weights function which can be called for each layer.

Gaussian distribution is a bell-shaped curve and the exponential function of the form,

Mean = sum of values/number of values

Variance = standard deviation square

Kaiming initialization: an initialization measure in neural network, which compensates the reducing or magnifying effect of inputs exponentially in the activation function.

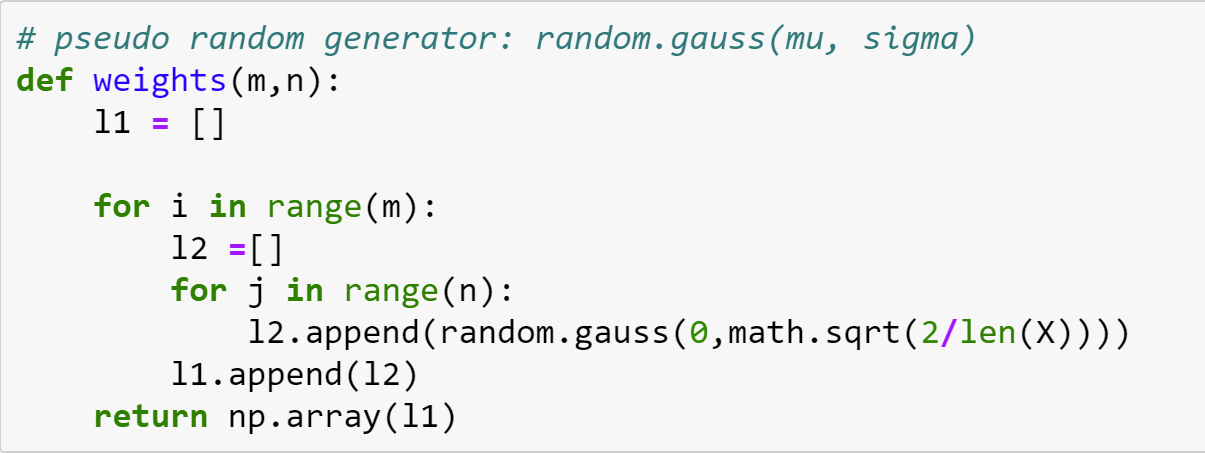
Initiating schema of Zero centered Gaussian with variance 2/number of inputs does this.

In our model, weights are sampled from a Gaussian distribution with Kaiming initialization,

i.e., mean = 0

variance = 2/number of inputs

Python has *random.gauss(mu, sigma)* built in function which takes mean and standard deviation as inputs and returns a random weight.



Weights function takes the matrix dimensions as input, generates gaussian random weight and returns weight matrix.

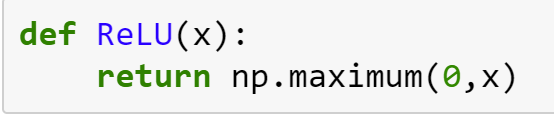
5. **Activation function**: The purpose of the activation function is to introduce non-linearity into the output of a neuron.

***ReLU activation*:**

Rectified linear activation function (ReLU) is a linear function that takes input and returns same as output if its positive value else returns zero.

i.e., output = input ……………if +ve

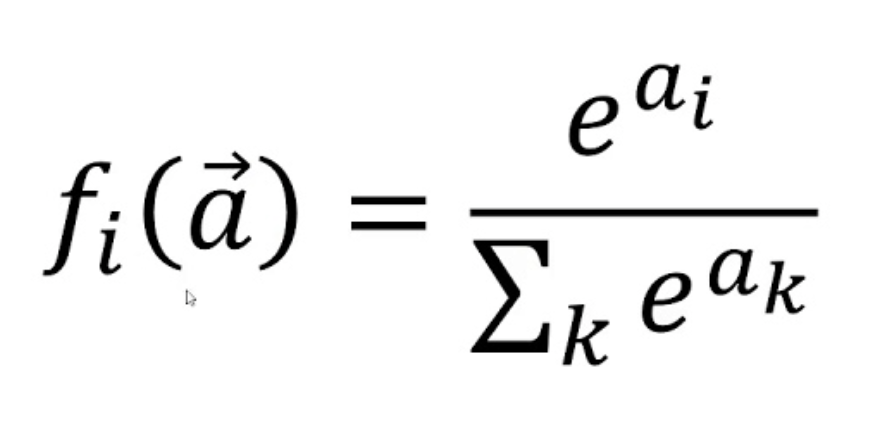
= 0 …………… if -zero or -ve

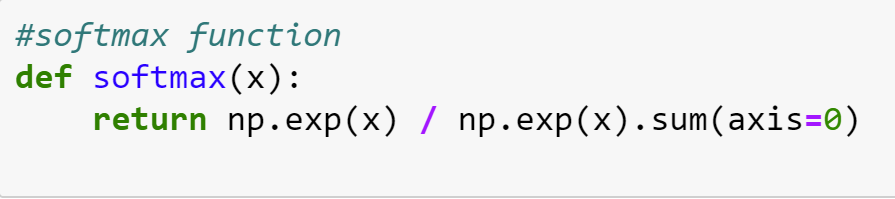


***Softmax activation*:**

Softmax is an activation function that scales numbers/logits into probabilities.

Mathematically,

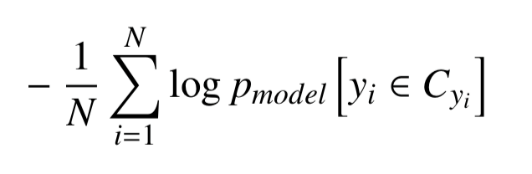




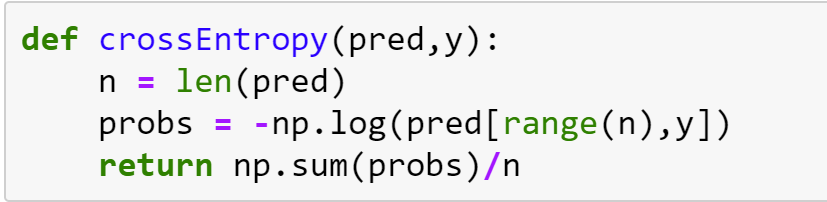
6. Cross – entropy

It’s a loss function in machine learning. Cross-entropy is a measure of the difference between two probability distributions for a given random variable or set of events.

Mathematically,



Where p is predicted and y is target values.

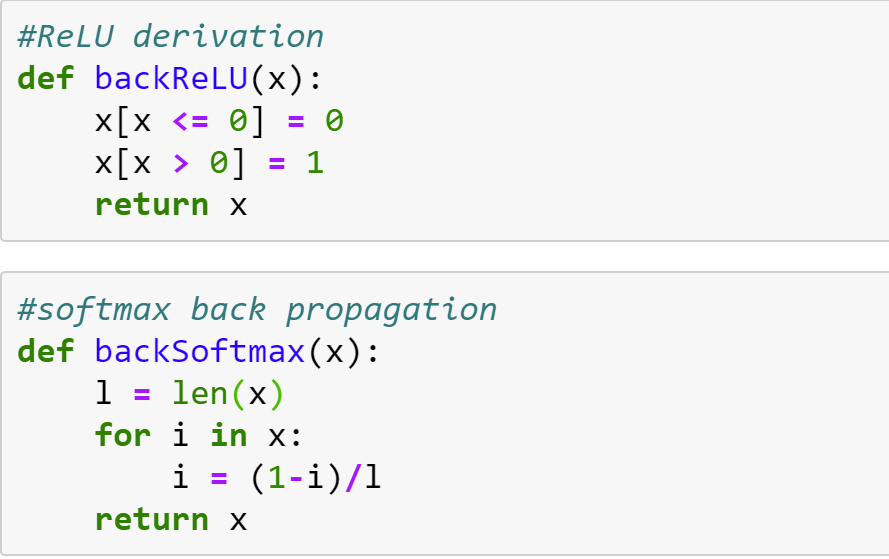


7. **Back propagation:**

Back propagation is a popular learning technique in neural networks. The error from predefined error function (stated above) is fed backwards through the network and weights are adjusted to reduce error. This cyclic process is continued till the error converges. At the state, the network has learned certain target function.

During back propagation, the ReLU and softmax activation funtions are differentiated. Below are the differentiated ReLU and softmax functions,

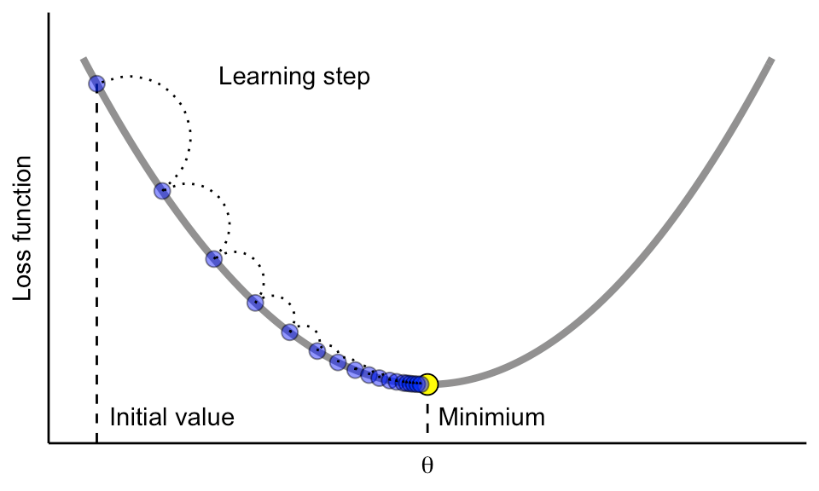
If value greater than 0, derivative of input is 1.



Gradient descent method is used to adjust the weights.

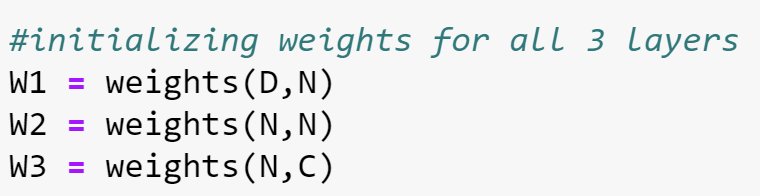
8. **Gradient descent:**

The gradient descent algorithm calculates the *derivative* of the loss curve at the starting point and propagates along the negative slope downhill to reach the local minima. At this point the loss is minimum. It’s the optimum model state.

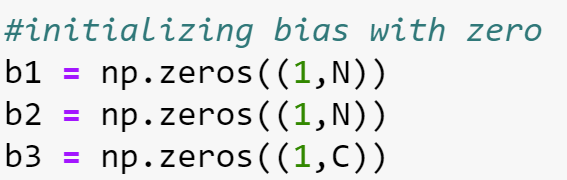


9. **Model Building:**

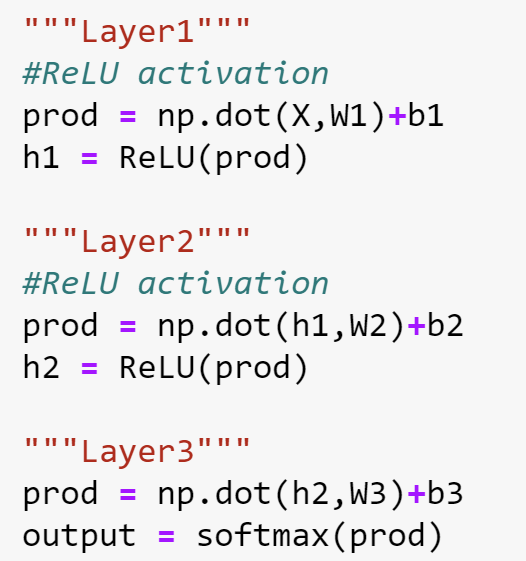
* Initiate the weights using *weights* functions, dimensions of matrix is the length or number of neurons in the past and next layer of the connection.



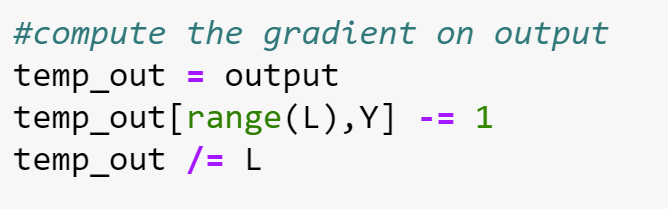
* Initialize bias to zero



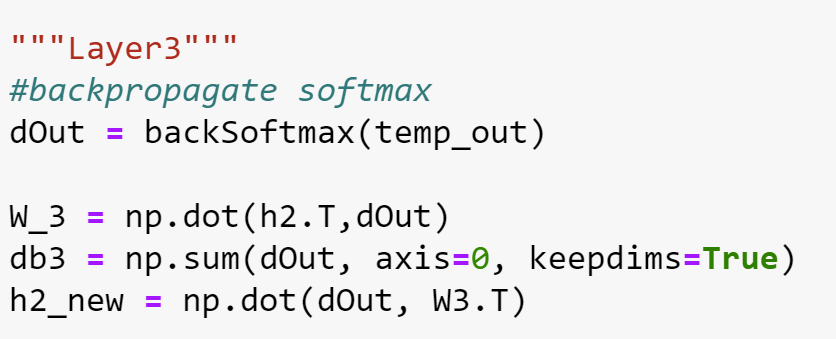
* Initiate while loop for 1000 ephocs
* Each neuron is connected to next layer neuron using function “(**input\*Weight)+Bias**” and the result is fed to activation functions.
* Used ReLU activation function for 2 hidden layers and softmax function for output layer.



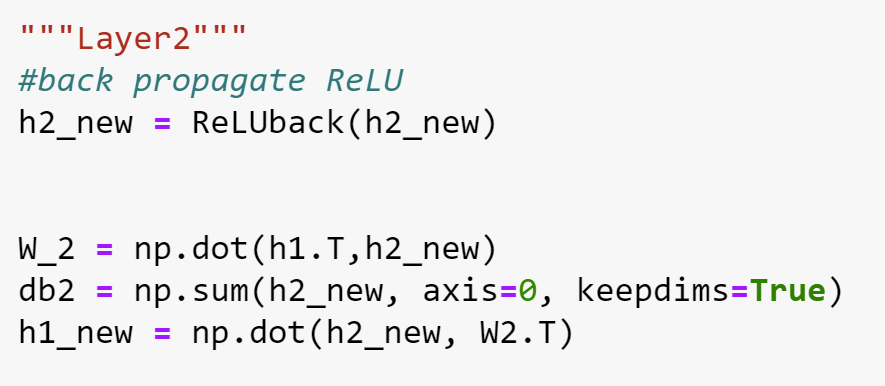
* Calculate loss using crossEntropy function mentioned above.
* Back propagation along the network, gradient is performed on output from third layer



* Now differentiate softmax function using *backSoftmax()* and calculate weights and bias as follows



* Next, differentiate ReLU activation function between layer1 and 2 using *backReLU()* and calculates weights,bias.



* Find gradient of ReLU activation function between input layer and layer1. Calculate weights and bias.
* Update weights in each iteration by multiplying with regularization coefficient and learning rate.
* Loop is exited after 1000 ephocs and final predicted values are returned.

**Analysis:**

* For Iris test dataset, with regularization coefficient 0.5 and learning rate 0.01

iteration 100: loss 3.280248, accuracy 66.666667

iteration 200: loss 3.279923, accuracy 56.666667

iteration 300: loss 3.247499, accuracy 63.333333

iteration 400: loss 3.338385, accuracy 70.000000

iteration 500: loss 3.333895, accuracy 76.666667

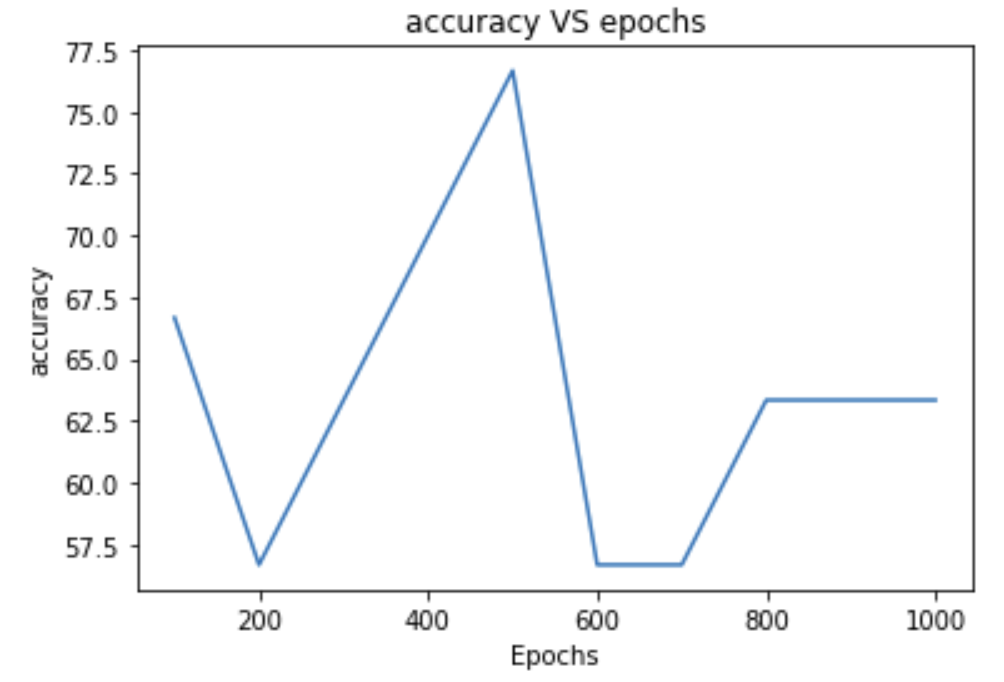
iteration 600: loss 3.351606, accuracy 56.666667

iteration 700: loss 3.329123, accuracy 56.666667

iteration 800: loss 3.314018, accuracy 63.333333

iteration 900: loss 3.305540, accuracy 63.333333

iteration 1000: loss 3.300662, accuracy 63.333333



From the graph, the peak accuracy for model is obtained at 500 ephoc and later it decreased. It is optimal to choose the model at 500 iteration.

Also, loss value did not converge within 1000 iterations.

Let’s increase the learning rate to 0.02 and validate,

iteration 100: loss 3.374374, accuracy 63.333333

iteration 200: loss 3.276881, accuracy 63.333333

iteration 300: loss 3.347270, accuracy 56.666667

iteration 400: loss 3.369842, accuracy 63.333333

iteration 500: loss 3.369842, accuracy 63.333333

iteration 600: loss 3.369844, accuracy 63.333333

iteration 700: loss 3.369844, accuracy 63.333333

iteration 800: loss 3.369844, accuracy 63.333333

iteration 900: loss 3.369844, accuracy 63.333333

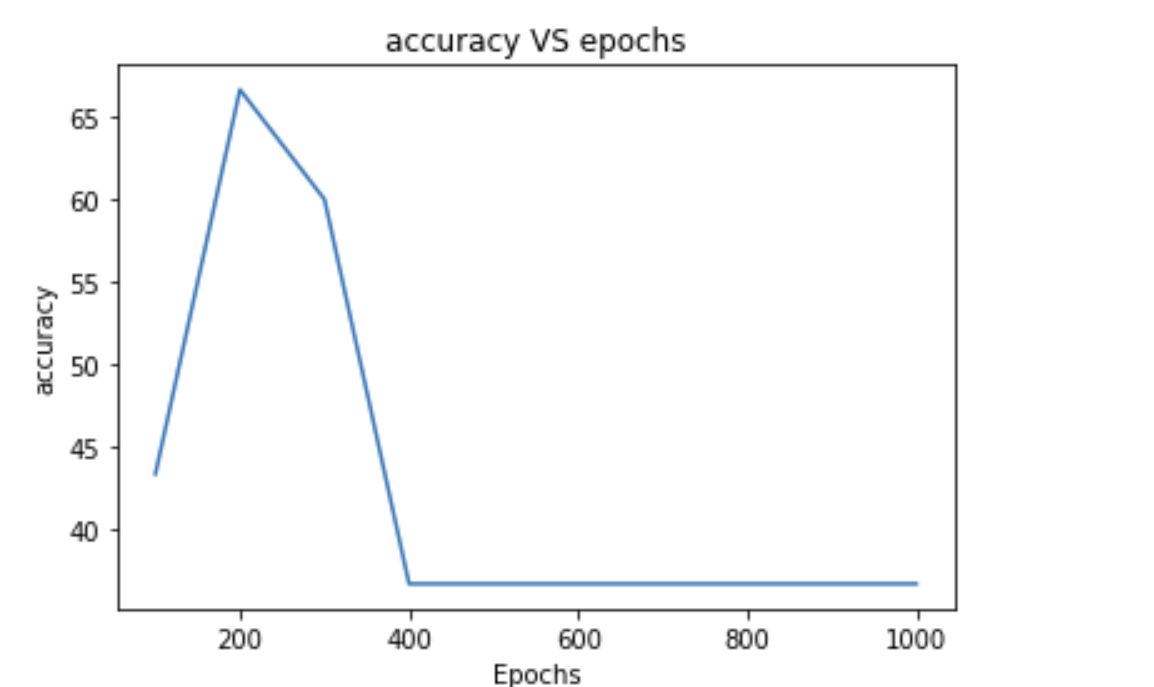
iteration 1000: loss 3.369844, accuracy 63.333333

loss converged at 600 echoc. It implies loss function converges fast with the increase in learning rate.

Now, lets analyze the effect of regularization coefficient,

With reg = 0.5 and alpha = 0.01, accuracy is 76%

With reg = 0.9 and alpha = 0.01, accuracy is 66%



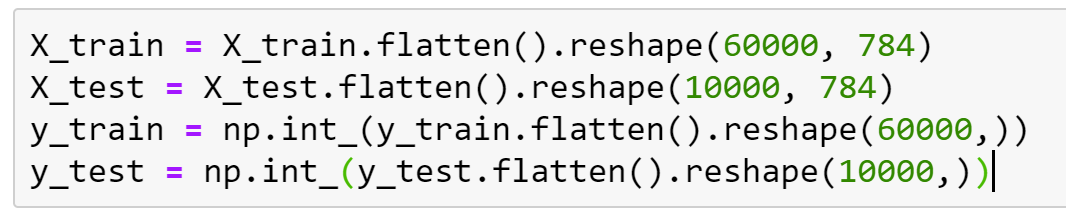
**Summary:**

* Loss function converges quickly with increase in learning rate.
* Accuracy decreased with increase in regularization value. This is because with increased reg value the model is generalized for unseen data and avoids overfitting.
* It is optimum to choose model at peak value of accuracy.

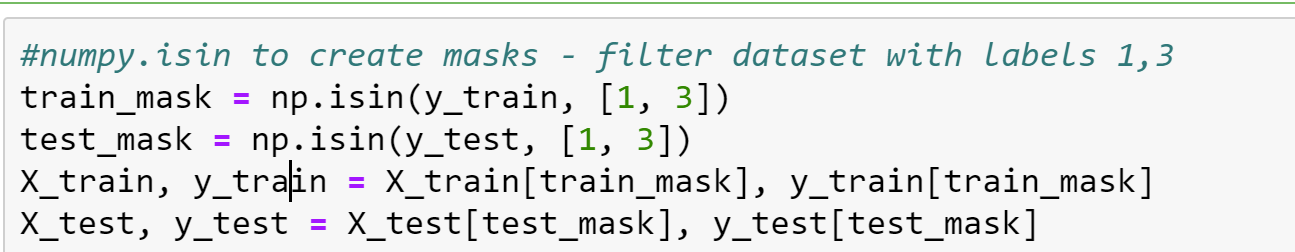
**Question2 - MNIST dataset**

Same model is used for analyzing MNIST dataset. Few functions are additions.

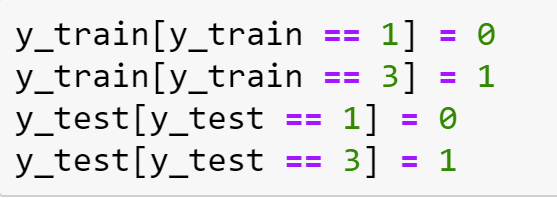
After loading the dataset, flatten and resize the data to convert into compatible array for the model.



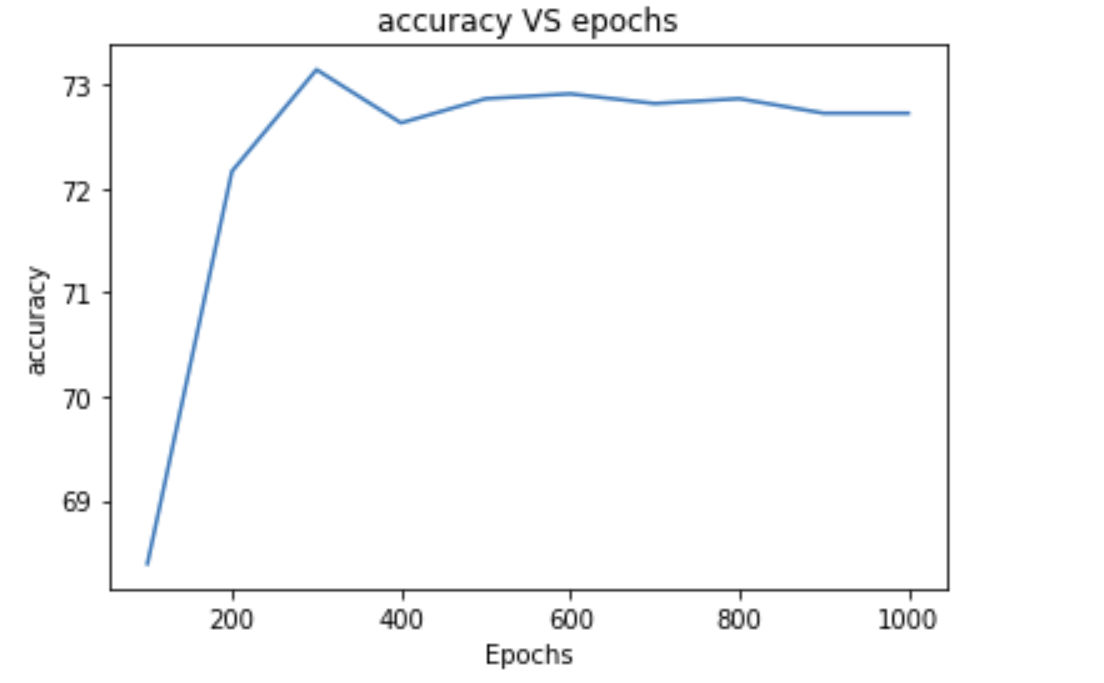
Our dataset is too large and takes more computation time. So reduce the features by filtering data related to digits 1,3 only. This is done by masking the dataset using isin() in numpy.



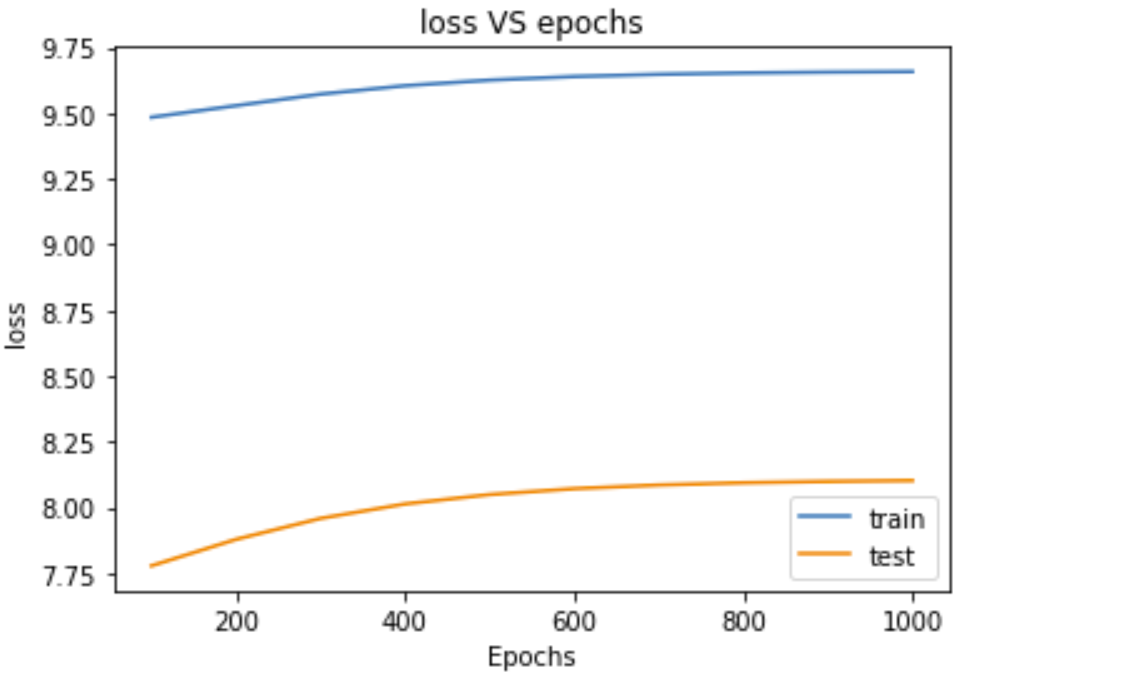
Assign labels to target dataset for ease of computation. Also called as one hot encoding.



With test dataset the model accuracy is 73%



**a. Plot training and validation loss against epochs**



For the model to be perfect, train loss = test loss.

Here, train loss> test loss. It means that this model is underfitted. On the positive note this model is not overfitting. Let’s see if this condition changes with other parameters.

Since we opted cross-entropy logarithmic function for calculating loss, the loss curves are parabolic.

**b. Impact of the size of mini-batch**

The two main things to consider when optimizing mini-batch size are the time efficiency of training and the noisiness of the gradient estimate.

let size of each mini batch be 1000 samples. This dataset have 12873 samples so we divide it into 13 batches.

Divide the train dataset into batches of 1000 samples each and train the model. Noted the computation time for full batch and mini batch approach.

training on full train dataset

iteration 100: loss 9.515293, accuracy 77.029441

iteration 200: loss 9.576103, accuracy 79.725006

iteration 300: loss 9.628226, accuracy 80.299852

iteration 400: loss 9.665075, accuracy 80.268780

iteration 500: loss 9.689263, accuracy 80.284316

iteration 600: loss 9.704585, accuracy 80.261011

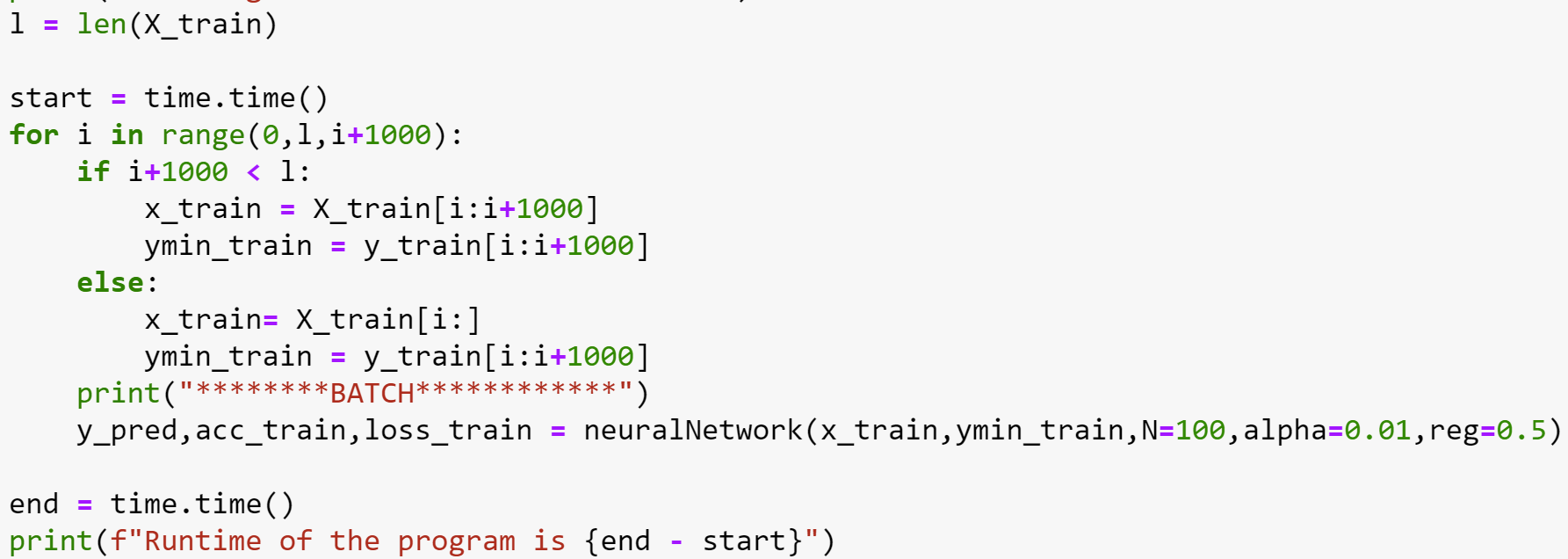
iteration 700: loss 9.714109, accuracy 80.237707

iteration 800: loss 9.719967, accuracy 80.198866

iteration 900: loss 9.723548, accuracy 80.152257

iteration 1000: loss 9.725729, accuracy 80.152257

Runtime of the program is 265.0452997684479



training on mini-bath train dataset

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 6.921725, accuracy 78.700000

iteration 200: loss 6.947344, accuracy 81.800000

iteration 300: loss 6.975276, accuracy 83.400000

iteration 400: loss 6.996613, accuracy 83.600000

iteration 500: loss 7.011129, accuracy 83.700000

iteration 600: loss 7.020499, accuracy 83.800000

iteration 700: loss 7.026385, accuracy 83.800000

iteration 800: loss 7.030028, accuracy 83.800000

iteration 900: loss 7.032262, accuracy 83.800000

iteration 1000: loss 7.033626, accuracy 83.800000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 7.023447, accuracy 78.900000

iteration 200: loss 7.149087, accuracy 83.500000

iteration 300: loss 7.248945, accuracy 84.600000

iteration 400: loss 7.315431, accuracy 85.400000

iteration 500: loss 7.357914, accuracy 85.600000

iteration 600: loss 7.369411, accuracy 86.200000

iteration 700: loss 7.360320, accuracy 87.400000

iteration 800: loss 7.321042, accuracy 87.400000

iteration 900: loss 7.326286, accuracy 87.700000

iteration 1000: loss 7.338779, accuracy 88.200000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 7.071975, accuracy 76.000000

iteration 200: loss 7.185641, accuracy 77.900000

iteration 300: loss 7.280337, accuracy 77.300000

iteration 400: loss 7.346238, accuracy 77.000000

iteration 500: loss 7.389098, accuracy 77.200000

iteration 600: loss 7.416097, accuracy 76.800000

iteration 700: loss 7.432822, accuracy 76.600000

iteration 800: loss 7.443087, accuracy 76.300000

iteration 900: loss 7.449354, accuracy 76.400000

iteration 1000: loss 7.453167, accuracy 76.200000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 6.991623, accuracy 70.900000

iteration 200: loss 7.068132, accuracy 74.000000

iteration 300: loss 7.129992, accuracy 75.000000

iteration 400: loss 7.172818, accuracy 75.300000

iteration 500: loss 7.200650, accuracy 74.900000

iteration 600: loss 7.218186, accuracy 75.100000

iteration 700: loss 7.229052, accuracy 75.100000

iteration 800: loss 7.235723, accuracy 75.000000

iteration 900: loss 7.239795, accuracy 75.000000

iteration 1000: loss 7.242274, accuracy 75.000000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 6.977350, accuracy 76.200000

iteration 200: loss 7.045839, accuracy 78.200000

iteration 300: loss 7.106553, accuracy 76.900000

iteration 400: loss 7.149389, accuracy 76.600000

iteration 500: loss 7.177370, accuracy 75.600000

iteration 600: loss 7.195028, accuracy 74.200000

iteration 700: loss 7.205975, accuracy 74.200000

iteration 800: loss 7.212696, accuracy 74.200000

iteration 900: loss 7.216800, accuracy 74.200000

iteration 1000: loss 7.219298, accuracy 74.200000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 6.876126, accuracy 85.100000

iteration 200: loss 6.889091, accuracy 86.600000

iteration 300: loss 6.915359, accuracy 87.800000

iteration 400: loss 6.937661, accuracy 87.900000

iteration 500: loss 6.953429, accuracy 87.900000

iteration 600: loss 6.963790, accuracy 87.900000

iteration 700: loss 6.970359, accuracy 87.800000

iteration 800: loss 6.974444, accuracy 87.600000

iteration 900: loss 6.976958, accuracy 87.600000

iteration 1000: loss 6.978495, accuracy 87.600000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 6.982425, accuracy 74.100000

iteration 200: loss 7.072755, accuracy 77.200000

iteration 300: loss 7.145380, accuracy 76.800000

iteration 400: loss 7.195246, accuracy 75.500000

iteration 500: loss 7.227469, accuracy 75.100000

iteration 600: loss 7.247700, accuracy 74.600000

iteration 700: loss 7.260210, accuracy 74.500000

iteration 800: loss 7.267879, accuracy 74.400000

iteration 900: loss 7.272559, accuracy 74.300000

iteration 1000: loss 7.275406, accuracy 74.200000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 6.947014, accuracy 74.200000

iteration 200: loss 7.002663, accuracy 78.400000

iteration 300: loss 7.053527, accuracy 78.300000

iteration 400: loss 7.090114, accuracy 78.600000

iteration 500: loss 7.114286, accuracy 78.700000

iteration 600: loss 7.129641, accuracy 78.800000

iteration 700: loss 7.139199, accuracy 78.900000

iteration 800: loss 7.145081, accuracy 78.900000

iteration 900: loss 7.148677, accuracy 79.000000

iteration 1000: loss 7.150868, accuracy 79.000000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 6.927278, accuracy 78.000000

iteration 200: loss 6.958115, accuracy 80.200000

iteration 300: loss 6.987032, accuracy 80.000000

iteration 400: loss 7.008108, accuracy 78.900000

iteration 500: loss 7.022135, accuracy 78.200000

iteration 600: loss 7.031085, accuracy 78.200000

iteration 700: loss 7.036670, accuracy 78.100000

iteration 800: loss 7.040112, accuracy 78.100000

iteration 900: loss 7.042219, accuracy 78.000000

iteration 1000: loss 7.043503, accuracy 78.300000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 6.990335, accuracy 75.400000

iteration 200: loss 7.075491, accuracy 77.400000

iteration 300: loss 7.143730, accuracy 76.900000

iteration 400: loss 7.190586, accuracy 76.500000

iteration 500: loss 7.220877, accuracy 76.700000

iteration 600: loss 7.239901, accuracy 76.900000

iteration 700: loss 7.251667, accuracy 76.900000

iteration 800: loss 7.258882, accuracy 76.800000

iteration 900: loss 7.263285, accuracy 76.800000

iteration 1000: loss 7.265963, accuracy 76.800000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 6.970633, accuracy 68.900000

iteration 200: loss 7.049750, accuracy 71.700000

iteration 300: loss 7.111320, accuracy 72.600000

iteration 400: loss 7.153284, accuracy 73.000000

iteration 500: loss 7.180343, accuracy 73.200000

iteration 600: loss 7.197319, accuracy 73.400000

iteration 700: loss 7.207804, accuracy 72.700000

iteration 800: loss 7.214238, accuracy 72.800000

iteration 900: loss 7.218162, accuracy 72.800000

iteration 1000: loss 7.220549, accuracy 72.700000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 6.930130, accuracy 74.700000

iteration 200: loss 7.005766, accuracy 76.200000

iteration 300: loss 7.070272, accuracy 76.300000

iteration 400: loss 7.115786, accuracy 76.700000

iteration 500: loss 7.145626, accuracy 76.800000

iteration 600: loss 7.164513, accuracy 76.800000

iteration 700: loss 7.176247, accuracy 76.800000

iteration 800: loss 7.183461, accuracy 76.800000

iteration 900: loss 7.187869, accuracy 76.800000

iteration 1000: loss 7.190553, accuracy 76.800000

\*\*\*\*\*\*\*\*BATCH\*\*\*\*\*\*\*\*\*\*\*\*

iteration 100: loss 6.823950, accuracy 75.486827

iteration 200: loss 6.865839, accuracy 79.266896

iteration 300: loss 6.910637, accuracy 79.610538

iteration 400: loss 6.944219, accuracy 79.266896

iteration 500: loss 6.966784, accuracy 78.579611

iteration 600: loss 6.981237, accuracy 78.579611

iteration 700: loss 6.990273, accuracy 77.892325

iteration 800: loss 6.995848, accuracy 77.777778

iteration 900: loss 6.999261, accuracy 77.777778

iteration 1000: loss 7.001343, accuracy 77.663230

Runtime of the program is 217.00770568847656

Observation:

* computation time is reduced in mini-batch gradient method.
* The value of loss is reduced in mini-batch

Mini-batch method can be opted for optimizing time efficiency and noise of gradient.

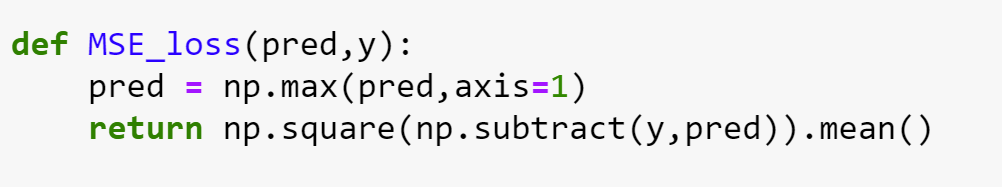
**c. Compare loss functions - Mean squared error vs. Cross-entropy.**

Mean squared error (MSE) is other type of error calculation function. In this the error is calculated by squaring the difference of predicted and target values, summing up all the errors and finding its average.

Equation:

Image courtesy of freecodecamp.org

Where n = batch size, y=target, yhat=predicted



Training the model for both loss functions separately,

using mse loss

iteration 100: loss 0.476190, accuracy 73.805640

iteration 200: loss 0.476189, accuracy 81.542764

iteration 300: loss 0.476188, accuracy 82.389497

iteration 400: loss 0.476187, accuracy 82.280743

iteration 500: loss 0.476187, accuracy 82.389497

iteration 600: loss 0.476187, accuracy 82.754603

iteration 700: loss 0.476187, accuracy 82.933271

iteration 800: loss 0.476187, accuracy 82.948808

iteration 900: loss 0.476187, accuracy 82.987649

iteration 1000: loss 0.476187, accuracy 82.964344

using cross-entropy loss

iteration 100: loss 9.557826, accuracy 70.208964

iteration 200: loss 9.680051, accuracy 69.618582

iteration 300: loss 9.771560, accuracy 69.424377

iteration 400: loss 9.832569, accuracy 69.168026

iteration 500: loss 9.871410, accuracy 69.168026

iteration 600: loss 9.895598, accuracy 69.704032

iteration 700: loss 9.910485, accuracy 69.688495

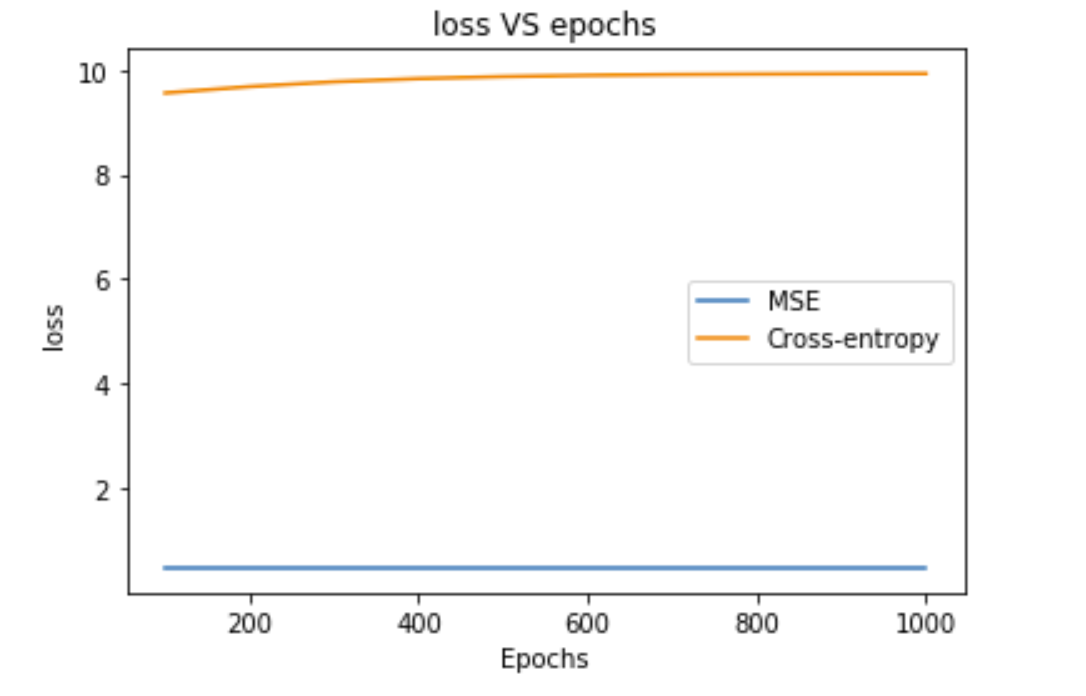
iteration 800: loss 9.919587, accuracy 69.696263

iteration 900: loss 9.925132, accuracy 69.750641

iteration 1000: loss 9.928502, accuracy 69.742873

From the above output, mean square error loss << cross-entropy loss and accuracy obtained by using MSE is higher than cross-entropy loss function.

So, for this model MSE loss function is optimal compared to cross-entropy function.



Loss curve is almost horizontal for MSE and parabolic for cross entropy.

**d. Impact of initialization. How does it affect the gradients?**

The trajectory of optimization is largely dependent on the initialization.

Let’s train the model with different weight initialization functions and analyze the difference.

Case 1: gaussian distribution mu=0, variance = 2/n

gaussian mu=0,v=(2/n)

iteration 100: loss 9.485478, accuracy 78.831663

iteration 200: loss 9.555691, accuracy 78.676299

iteration 300: loss 9.615341, accuracy 78.419949

iteration 400: loss 9.657254, accuracy 77.619824

iteration 500: loss 9.684661, accuracy 77.697506

iteration 600: loss 9.701982, accuracy 77.487765

iteration 700: loss 9.712734, accuracy 77.472229

iteration 800: loss 9.719341, accuracy 77.938321

iteration 900: loss 9.723377, accuracy 77.938321

iteration 1000: loss 9.725835, accuracy 77.938321

Case 2: gaussian distribution mu=0, variance = 1

gaussian mu=0,v=1

iteration 100: loss 10.523761, accuracy 51.161345

iteration 200: loss 10.184831, accuracy 56.668997

iteration 300: loss 10.022731, accuracy 61.454206

iteration 400: loss 9.948554, accuracy 64.670240

iteration 500: loss 9.899825, accuracy 69.222403

iteration 600: loss 9.873980, accuracy 70.504156

iteration 700: loss 9.865645, accuracy 71.490717

iteration 800: loss 9.868440, accuracy 69.999223

iteration 900: loss 9.890904, accuracy 70.659520

iteration 1000: loss 9.910059, accuracy 69.812786

Case 3: Zero initialization

zero initialization

iteration 100: loss 9.462887, accuracy 47.626816

iteration 200: loss 9.462887, accuracy 47.626816

iteration 300: loss 9.462887, accuracy 52.373184

iteration 400: loss 9.462887, accuracy 47.626816

iteration 500: loss 9.462887, accuracy 52.373184

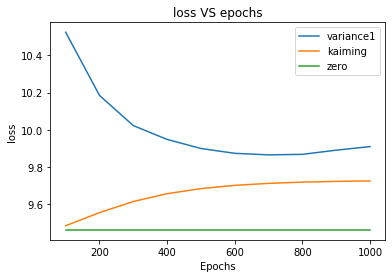
iteration 600: loss 9.462887, accuracy 52.373184

iteration 700: loss 9.462887, accuracy 47.626816

iteration 800: loss 9.462887, accuracy 47.626816

iteration 900: loss 9.462887, accuracy 52.373184

iteration 1000: loss 9.462887, accuracy 52.373184

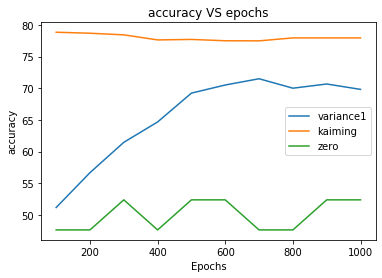


There is visible variation of loss curve with weights initialization.

For zero initialization loss is constant and curve is horizontal.

Loss function of kaiming initialization conversed slightly earlier than that with variance =1.

Loss\_zero < Loss\_kaiming < Loss1



Accuracy: kaiming > variance1 > zero initialization

Kaiming initialization is optimal for this model.

**e. Learning rate.**

Learning rate 0.01

iteration 100: loss 9.536474, accuracy 74.932028

iteration 200: loss 9.619102, accuracy 77.526606

iteration 300: loss 9.687328, accuracy 77.441156

iteration 400: loss 9.734934, accuracy 77.433388

iteration 500: loss 9.765990, accuracy 77.231415

iteration 600: loss 9.785598, accuracy 76.780859

iteration 700: loss 9.797763, accuracy 76.749786

iteration 800: loss 9.805237, accuracy 76.804164

iteration 900: loss 9.809803, accuracy 76.804164

iteration 1000: loss 9.812583, accuracy 76.641032

Learning rate 0.02

iteration 100: loss 9.549554, accuracy 82.389497

iteration 200: loss 9.643054, accuracy 82.016624

iteration 300: loss 9.686586, accuracy 81.449546

iteration 400: loss 9.703686, accuracy 81.465082

iteration 500: loss 9.710098, accuracy 81.371864

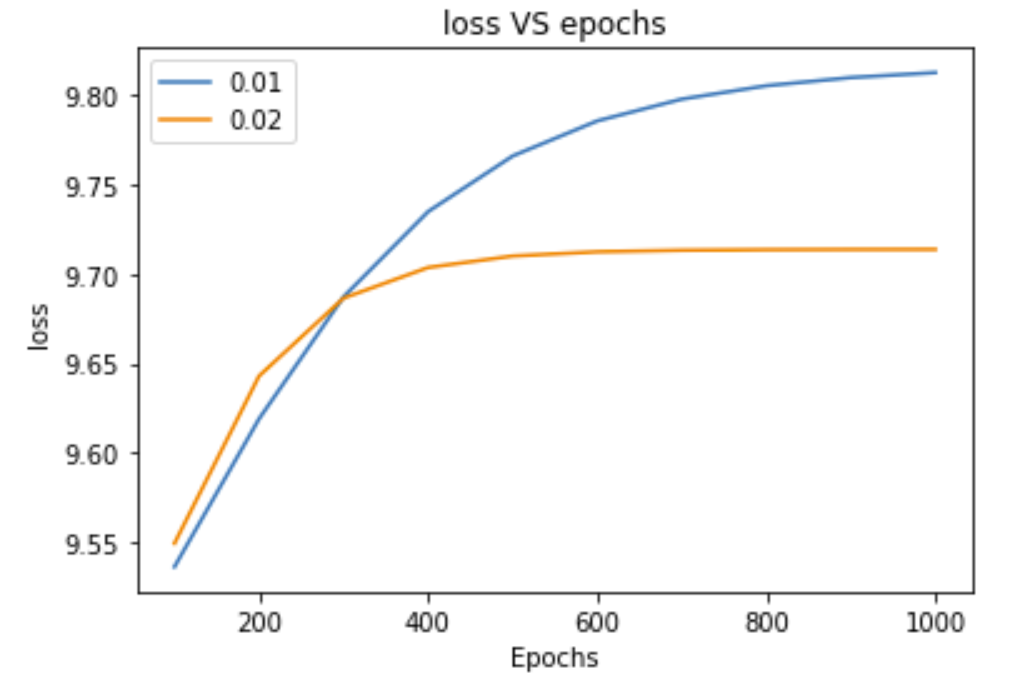
iteration 600: loss 9.712465, accuracy 81.371864

iteration 700: loss 9.713334, accuracy 81.371864

iteration 800: loss 9.713652, accuracy 81.371864

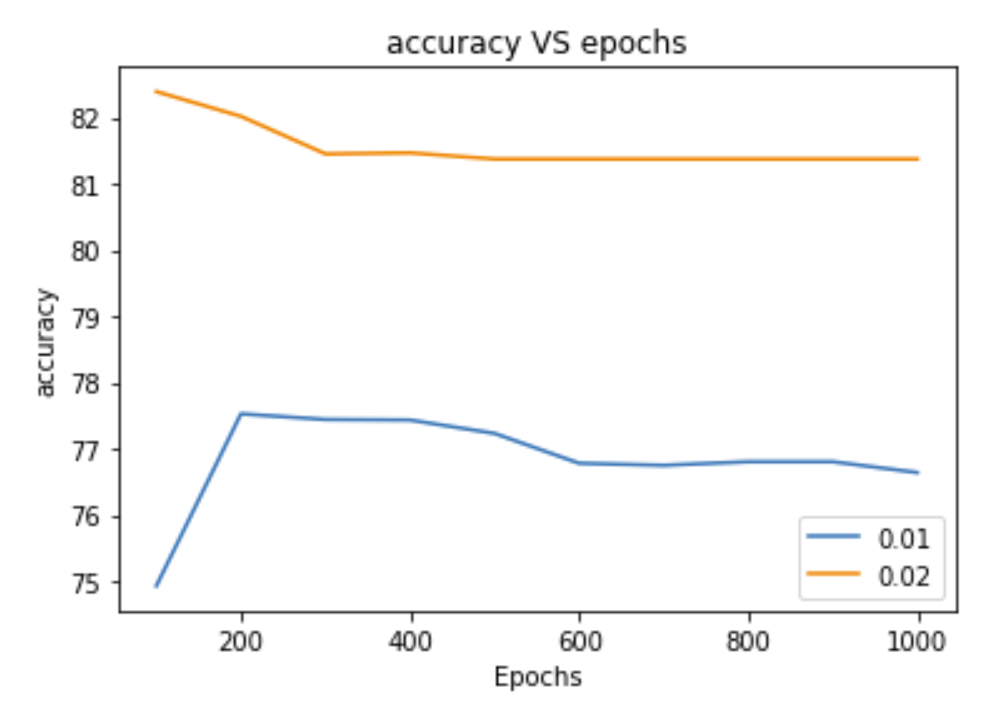
iteration 900: loss 9.713769, accuracy 81.371864

iteration 1000: loss 9.713812, accuracy 81.371864



For learning rate 0.01 loss couldn’t converge within 1000 epochs.

For learning rate 0.02 loss function converged quickly in 700 ephocs.



This model seems to have higher accuracy at larger leaning rate.

With increase in learning rate, loss function converges quickly, and model learns in less ephocs.

**f. Impact of width vs. depth on the model performance.**

Width in the neural network refers to the number of layers and the depth refers to number of neurons for each layer.

Case 1: depth = 100, width = 3

iteration 100: loss 9.455006, accuracy 76.687641

iteration 200: loss 9.461160, accuracy 81.169890

iteration 300: loss 9.462418, accuracy 86.063855

iteration 400: loss 9.462771, accuracy 87.314534

iteration 500: loss 9.462859, accuracy 88.293327

iteration 600: loss 9.462880, accuracy 88.844869

iteration 700: loss 9.462886, accuracy 88.837101

iteration 800: loss 9.462887, accuracy 87.904917

iteration 900: loss 9.462887, accuracy 87.159170

iteration 1000: loss 9.462887, accuracy 86.087159

Case2: depth = 50, width = 3

iteration 100: loss 9.459304, accuracy 75.701080

iteration 200: loss 9.461442, accuracy 81.837955

iteration 300: loss 9.462510, accuracy 82.451643

iteration 400: loss 9.462798, accuracy 82.498252

iteration 500: loss 9.462866, accuracy 82.661384

iteration 600: loss 9.462883, accuracy 82.544861

iteration 700: loss 9.462886, accuracy 82.210829

iteration 800: loss 9.462887, accuracy 81.713664

iteration 900: loss 9.462887, accuracy 81.232036

iteration 1000: loss 9.462887, accuracy 81.138818

Case3: depth = 150, width = 3

iteration 100: loss 9.452686, accuracy 73.790103

iteration 200: loss 9.459571, accuracy 80.563971

iteration 300: loss 9.461852, accuracy 87.516507

iteration 400: loss 9.462624, accuracy 89.707139

iteration 500: loss 9.462839, accuracy 88.169036

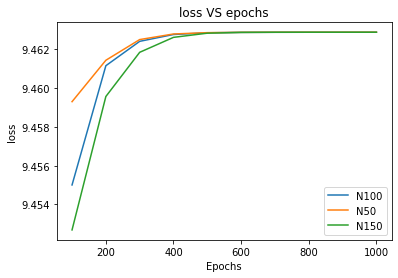
iteration 600: loss 9.462879, accuracy 83.927600

iteration 700: loss 9.462886, accuracy 85.962868

iteration 800: loss 9.462887, accuracy 88.720578

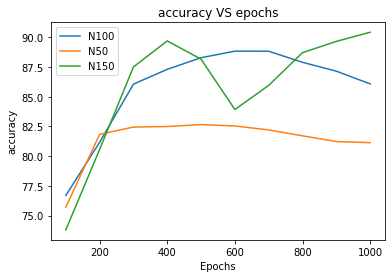
iteration 900: loss 9.462887, accuracy 89.668298

iteration 1000: loss 9.462887, accuracy 90.437349



All the three loss curves converge at same loss value at nearly same epoch.

At constant width, depth of the neural network doesn’t have much impact on the loss function.



Accuracy: N150>N100>N50

With the increase in depth of neural network, the accuracy of the model increases.

**g. The role of various activations functions such as ReLU, sigmoid, hyperbolic**

**tangent.**

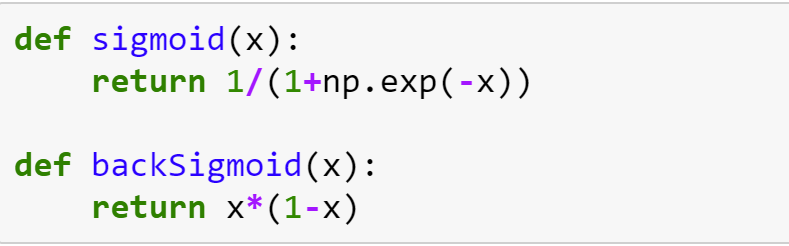
ReLU function is described in this report earlier.

**Sigmoid:** It is also called as logistic activation function. The function range between (0,1)

f(x)=1/(1+exp(-x))

derivative: df(x)=f(x)\*(1-f(x))

code for sigmoid function and its gradient:



**Hyperbolic tangent**:

used as a nonlinear activation function between layers of a neural network.

F(x) = tanh(x)

Derivative: dF(x) = 1-F(x)^2



Case 1: ReLU activation function

iteration 100: loss 9.449308, accuracy 81.869028

iteration 200: loss 9.458659, accuracy 85.364717

iteration 300: loss 9.461867, accuracy 86.382351

iteration 400: loss 9.462677, accuracy 88.806028

iteration 500: loss 9.462857, accuracy 89.971258

iteration 600: loss 9.462881, accuracy 90.041171

iteration 700: loss 9.462886, accuracy 90.219840

iteration 800: loss 9.462887, accuracy 90.320827

iteration 900: loss 9.462887, accuracy 90.460654

iteration 1000: loss 9.462887, accuracy 90.802455

Case 2: sigmoid activation function

iteration 100: loss 9.462867, accuracy 63.971102

iteration 200: loss 9.462887, accuracy 42.732852

iteration 300: loss 9.462887, accuracy 43.470830

iteration 400: loss 9.462887, accuracy 57.748777

iteration 500: loss 9.462887, accuracy 39.951837

iteration 600: loss 9.462887, accuracy 36.160957

iteration 700: loss 9.462887, accuracy 64.833372

iteration 800: loss 9.462887, accuracy 52.373184

iteration 900: loss 9.462887, accuracy 47.626816

iteration 1000: loss 9.462887, accuracy 52.373184

Case 3: hyperbolic tangent activation function

iteration 100: loss 9.462887, accuracy 52.373184

iteration 200: loss 9.462887, accuracy 52.373184

iteration 300: loss 9.462887, accuracy 47.626816

iteration 400: loss 9.462887, accuracy 52.373184

iteration 500: loss 9.462887, accuracy 47.626816

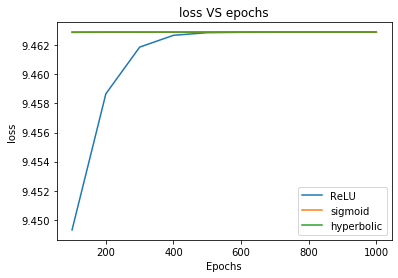
iteration 600: loss 9.462887, accuracy 52.373184

iteration 700: loss 9.462887, accuracy 47.619048

iteration 800: loss 9.462887, accuracy 47.626816

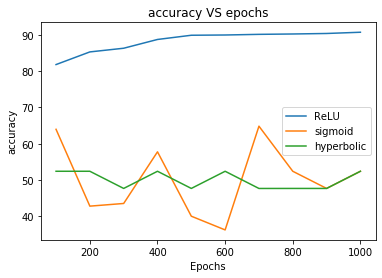
iteration 900: loss 9.462887, accuracy 47.626816

iteration 1000: loss 9.462887, accuracy 52.373184



All three functions converged at same value of Loss.

Loss of sigmoid and hyperbolic functions are constant and equal. Loss curve is horizontal.



For this dataset ReLU has highest accuracy and is optimal. Average accuracy of sigmoid and hyperbolic functions is almost same.