PURDUE UNIVERSITY

Elmore Family School of Electrical and Computer Engineering

Deep Learning

Homework 3

Adithya Sineesh

Email : asineesh@purdue.edu

Submitted: February 6, 2023

1 Theory

1.1 SGD+

In traditional Gradient Descent and Stochastic Gradient Descent, there is a tendency for the solution path to oscillate as it approaches the minimum. This problem can be overcome by using SGD+ which incorporates the concept of momentum. Its basic idea is to compare the current step size with the previous step size. If both point in the same direction, the solution path can accelerate toward the minimum else it has to be more cautious. This idea is implemented in the following formula

$$v_{t+1} = \mu v_t + g_{t+1}$$
$$p_{t+1} = p_t + \alpha v_{t+1}$$

where

g_{t+1} is the gradient of loss at the iteration $(t+1)$

v_{t+1} is the step size at iteration $(t+1)$

v_t is the step size at iteration (t)

p_{t+1} are the parameters at iteration $(t+1)$

p_t are the parameters at iteration (t)

α is the learning rate

μ is the weight given to the previous iteration step size

v_0 is initialized to zero.

1.2 Adam

The problem of sparse gradients is because a given input will not be equally sensitive to all the parameters of the network. As a result, it does not make sense to use the same learning rate for all the model parameters. Adaptive Moment Estimation (Adam) solves this problem along with the problem laid out in the previous section by calculating an average of the first moment along with the second moment as follows:

$$m_{t+1} = \beta_1 m_t + (1 - \beta_1) * g_{t+1}$$
$$v_{t+1} = \beta_2 v_t + (1 - \beta_2) * g_{t+1}^2$$

where

g_{t+1} is the gradient of loss at the iteration $(t + 1)$

m_{t+1} is the first moment average at iteration $(t + 1)$

m_t is the first moment average at iteration (t)

v_{t+1} is the second moment average at iteration $(t + 1)$

v_t is the second moment average at iteration (t)

β_1 and β_2 are the parameters of Adam

Like in the above section, v_0 and m_0 are set to 0. To address this flaw, the following correction is applied:

$$m_t = \frac{m_t}{1-\beta_1}$$
$$v_t = \frac{v_t}{1-\beta_2}$$

Now the model parameters can be updated as follows:

$$p_{t+1} = p_t + \alpha \frac{m_{t+1}}{\sqrt{v_{t+1}+\epsilon}}$$

where, p_{t+1} are the parameters at iteration $(t + 1)$

p_t are the parameters at iteration (t)

α is the learning rate

2 Experiment

In this homework, we compare the performance of the vanilla SGD, SDG+ and the Adam optimizers on a one-neuron and multi neuron classifier implemented in the CGP primer. My changes to the code were mainly on the backpropagation function. I also looked at the solutions of Spring 2022 for reference.

2.1 Task 1

For the first task, we compare the results of the optimizer on the one-neuron classifier. For SGD+ $\mu = 0.9$ and for Adam $\beta_1 = 0.9, \beta_2 = 0.99$. The experiments were run for the learning rates of 0.005 and 0.01.

2.2 Task 1

For the second, we compare the results of the optimizer on the multi-neuron classifier. For SGD+ $\mu = 0.9$ and for Adam $\beta_1 = 0.9, \beta_2 = 0.99$. The experiments were run for the learning rates of 0.005 and 0.01.

3 Results

3.1 Task 1

For one neuron classifier

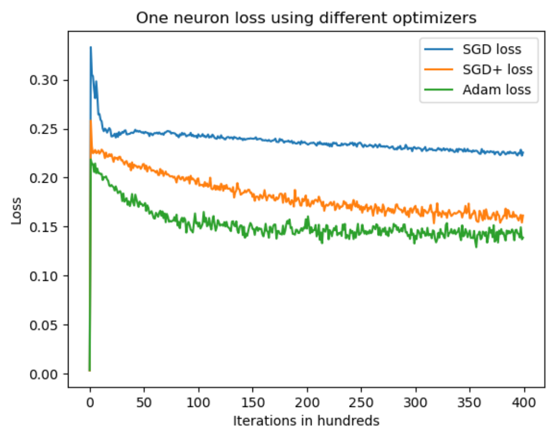

Figure 1: For Learning rate of 0.01

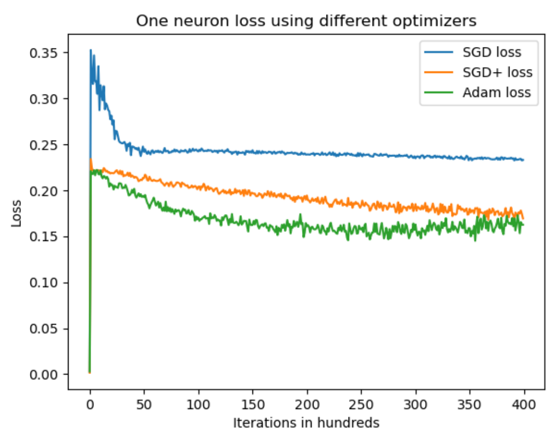

Figure 2: For Learning rate of 0.005

For both the learning rates, the Adam optimizer performs the best followed by the SGD+ optimizer and the vanilla SGD optimizer.

3.2 Task 2

For multi neuron classifier

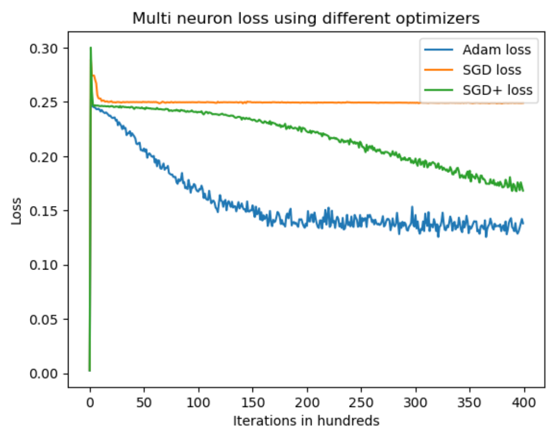

Figure 3: For Learning rate of 0.01

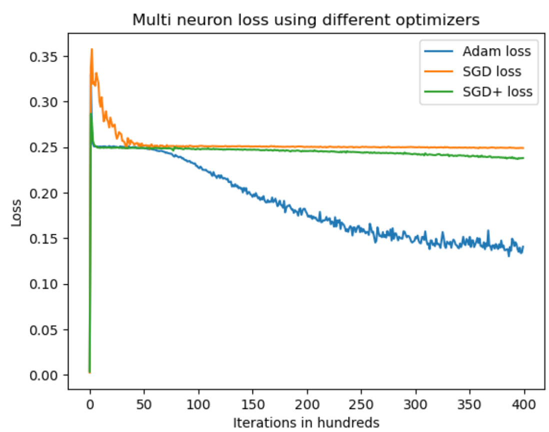

Figure 4: For Learning rate of 0.005

For both the learning rates, the Adam optimizer performs the best followed by the SGD+ optimizer and the vanilla SGD optimizer. SGD+ also performs way better with the learning rate of 0.01

In [1]:

```
import random
import numpy as np
import matplotlib.pyplot as plt
from ComputationalGraphPrimer import *
import operator
```

In [2]:

```
class SGDPlus(ComputationalGraphPrimer):
    def __init__(self,*args,**kwargs):
        super().__init__(*args,**kwargs)

    def run_training_loop_one_neuron_model(self, training_data,mu=0.0,SGDplus=False):
        """
        The training loop must first initialize the learnable parameters.  Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}

        self.bias = random.uniform(0,1)                   ## Adding the bias improves class discrimination.
                                                          ##   We initialize it to a random number.

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, the training dataset
            consists of a Python dict with two keys, 0 and 1, the former points to a list of
            all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
            classes have different means and variances.  The dimensionality of each data sample
            is set by the number of nodes in the input layer of the neural network.

            The data loader's job is to construct a batch of samples drawn randomly from the two
            lists mentioned above.  And it mush also associate the class label with each sample
            separately.
            """
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]]   ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]   ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])

            def _getitem(self):
                cointoss = random.choice([0,1])                            ## When a batch is created by getbatch(), we want the
                                                                           ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]                            ## First list for samples, the second for labels
                maxval = 0.0                                               ## For approximate batch data normalization
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch

        ##My input start
        self.bias_update = 0.0
        self.step = [0]*(len(self.learnable_params)+1)
        self.mu = mu if SGDplus else 0.0

        ##My input end


        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0                            ##  Average the loss over iterations for printing out
                                                                   ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(data_tuples)             ##  FORWARD PROP of data
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ##  Find loss
            loss_avg = loss / float(len(class_labels))                                              ##  Average the loss over bat
```

```
            avg_loss_over_iterations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))           ## Display average loss
                avg_loss_over_iterations = 0.0                                                ## Re-initialize avg loss
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(class_labels))
            data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
            data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
                                 [float(len(class_labels))] * len(class_labels) ))
            self.backprop_and_update_params_one_neuron_model(y_error_avg, data_tuple_avg, deriv_sigmoid_avg)     ## BACKPROP loss

        return loss_running_record


    def forward_prop_one_neuron_model(self, data_tuples_in_batch):
        """
        Forward propagates the batch data through the neural network according to the equations on
        Slide 50 of my Week 3 slides.

        As the one-neuron model is characterized by a single expression, the main job of this function is
        to evaluate that expression for each data tuple in the incoming batch.  The resulting output is
        fed into the sigmoid activation function and the partial derivative of the sigmoid with respect
        to its input calculated.
        """
        output_vals = []
        deriv_sigmoids = []
        for vals_for_input_vars in data_tuples_in_batch:
            input_vars = self.independent_vars                     ## This is a list of vars for the input nodes. For the
                                                                   ##   the One-Neuron example in the Examples directory
                                                                   ##   this is just the list [xa, xb, xc, xd]
            vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))   ## The current values at input

            exp_obj = self.exp_objects[0]                          ## To understand this, first see the definition of the
                                                                   ##   Exp class (search for the string "class Exp").
                                                                   ##   Each expression that defines the neural network is
                                                                   ##   represented by one Exp instance by the parser.
            output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict, self.vals_for_learnable_params)

            ## [Search for "self.bias" in this file.]  As mentioned earlier, adding bias improves class discrimination:
            output_val = output_val + self.bias

            output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))   ## Apply sigmoid activation (output confined to [0.0,1.0] inte

            deriv_sigmoid = output_val * (1.0 - output_val)        ## See Slide 59 for why we need partial deriv of Sigmoid at in

            output_vals.append(output_val)                         ## Collect output values for different input samples in batch

            deriv_sigmoids.append(deriv_sigmoid)                   ## Collect the Sigmoid derivatives for each input sample in ba
                                                                   ##   The derivatives that are saved during forward prop are sh
        return output_vals, deriv_sigmoids


    def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars, deriv_sigmoid):
        """
        As should be evident from the syntax used in the following call to backprop function,

           self.backprop_and_update_params_one_neuron_model( y_error_avg, data_tuple_avg, deriv_sigmoid_avg)
                                                                     ^^^            ^^^                ^^^
        the values fed to the backprop function for its three arguments are averaged over the training
        samples in the batch.  This in keeping with the spirit of SGD that calls for averaging the
        information retained in the forward propagation over the samples in a batch.

        See Slide 59 of my Week 3 slides for the math of back propagation for the One-Neuron network.
        """
        input_vars = self.independent_vars
        vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            ## My change start
            self.step[i] = (self.mu*self.step[i]) +  y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid

            ## Update the learnable parameters
            self.vals_for_learnable_params[param] += self.learning_rate*self.step[i]

        ## Update the bias
        self.bias_update = (self.mu*self.bias_update) + y_error * deriv_sigmoid
        self.bias += self.learning_rate*self.bias_update
            ##My change end
```

In [3]:

```
class Adam(ComputationalGraphPrimer):
    def __init__(self,*args,**kwargs):
        super().__init__(*args,**kwargs)

    def run_training_loop_one_neuron_model(self, training_data,beta1,beta2):
        """
        The training loop must first initialize the learnable parameters.  Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}

        self.bias = random.uniform(0,1)                   ## Adding the bias improves class discrimination.
                                                          ##   We initialize it to a random number.

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, the training dataset
            consists of a Python dict with two keys, 0 and 1, the former points to a list of
            all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
            classes have different means and variances.  The dimensionality of each data sample
            is set by the number of nodes in the input layer of the neural network.

            The data loader's job is to construct a batch of samples drawn randomly from the two
            lists mentioned above.  And it mush also associate the class label with each sample
            separately.
            """
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]]   ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]   ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])

            def _getitem(self):
                cointoss = random.choice([0,1])                            ## When a batch is created by getbatch(), we want the
                                                                           ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]                            ## First list for samples, the second for labels
                maxval = 0.0                                               ## For approximate batch data normalization
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch

        ##My input start
        self.bias_m = 0.0
        self.bias_v = 0.0

        self.bias_mh = 0.0
        self.bias_vh = 0.0

        self.step_m = [0]*(len(self.learnable_params)+1)
        self.step_v = [0]*(len(self.learnable_params)+1)
        self.step_mh = [0]*(len(self.learnable_params)+1)
        self.step_vh = [0]*(len(self.learnable_params)+1)

        self.beta1 = beta1
        self.beta2 = beta2
        self.m = 0

        ##My input end


        data_loader = DataLoader(training_data, batch_size=self.batch_size)
```

```
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0                                ##  Average the loss over iterations for printing out
                                                                       ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            self.m = i+1
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(data_tuples)              ##  FORWARD PROP of data
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ##  Find loss
            loss_avg = loss / float(len(class_labels))                                              ##  Average the loss over bat
            avg_loss_over_iterations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))                 ## Display average loss
                avg_loss_over_iterations = 0.0                                                     ## Re-initialize avg loss
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(class_labels))
            data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
            data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
                                 [float(len(class_labels))] * len(class_labels) ))
            self.backprop_and_update_params_one_neuron_model(y_error_avg, data_tuple_avg, deriv_sigmoid_avg)     ## BACKPROP loss

        return loss_running_record


    def forward_prop_one_neuron_model(self, data_tuples_in_batch):
        """
        Forward propagates the batch data through the neural network according to the equations on
        Slide 50 of my Week 3 slides.

        As the one-neuron model is characterized by a single expression, the main job of this function is
        to evaluate that expression for each data tuple in the incoming batch.  The resulting output is
        fed into the sigmoid activation function and the partial derivative of the sigmoid with respect
        to its input calculated.
        """
        output_vals = []
        deriv_sigmoids = []
        for vals_for_input_vars in data_tuples_in_batch:
            input_vars = self.independent_vars                    ## This is a list of vars for the input nodes. For the
                                                                  ##   the One-Neuron example in the Examples directory
                                                                  ##   this is just the list [xa, xb, xc, xd]
            vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))   ## The current values at input

            exp_obj = self.exp_objects[0]                         ## To understand this, first see the definition of the
                                                                  ##   Exp class (search for the string "class Exp").
                                                                  ##   Each expression that defines the neural network is
                                                                  ##   represented by one Exp instance by the parser.
            output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict, self.vals_for_learnable_params)

            ## [Search for "self.bias" in this file.]  As mentioned earlier, adding bias improves class discrimination:
            output_val = output_val + self.bias

            output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))   ## Apply sigmoid activation (output confined to [0.0,1.0] inte

            deriv_sigmoid = output_val * (1.0 - output_val)        ## See Slide 59 for why we need partial deriv of Sigmoid at in

            output_vals.append(output_val)                         ## Collect output values for different input samples in batch

            deriv_sigmoids.append(deriv_sigmoid)                   ## Collect the Sigmoid derivatives for each input sample in ba
                                                                   ##   The derivatives that are saved during forward prop are sh
        return output_vals, deriv_sigmoids


    def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars, deriv_sigmoid):
        """
        As should be evident from the syntax used in the following call to backprop function,

           self.backprop_and_update_params_one_neuron_model( y_error_avg, data_tuple_avg, deriv_sigmoid_avg)
                                                                  ^^^            ^^^                ^^^
        the values fed to the backprop function for its three arguments are averaged over the training
        samples in the batch.  This in keeping with the spirit of SGD that calls for averaging the
        information retained in the forward propagation over the samples in a batch.

        See Slide 59 of my Week 3 slides for the math of back propagation for the One-Neuron network.
        """
        input_vars = self.independent_vars
        vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            ## My change start
            self.step_m[i] = (self.beta1*self.step_m[i]) + (1-self.beta1)*(y_error * vals_for_input_vars_dict[input_vars[i]] * de
            self.step_mh[i] = self.step_m[i]/(1-self.beta1**self.m)
```

```
            self.step_v[i] = (self.beta2*self.step_v[i]) + (1-self.beta2)*((y_error * vals_for_input_vars_dict[input_vars[i]] * d
            self.step_vh[i] = self.step_v[i]/(1-self.beta2**self.m)

            ## Update the learnable parameters
            self.vals_for_learnable_params[param] += self.learning_rate * (self.step_mh[i]/(np.sqrt(self.step_vh[i])+10**-6))

        ## Update the bias
        self.bias_m = (self.beta1*self.bias_m) + (1-self.beta1)*(y_error * deriv_sigmoid)
        self.bias_mh = self.bias_m/(1-self.beta1**self.m)
        self.bias_v = (self.beta2*self.bias_v) + (1-self.beta2)*((y_error * deriv_sigmoid)**2)
        self.bias_vh = self.bias_v/(1-self.beta2**self.m)
        self.bias += self.learning_rate * (self.bias_m/(np.sqrt(self.bias_v)+10**-6))
            ##My change end
```

In [4]:

```
cgp1 = SGDPlus(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = 5 * 1e-3,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )


cgp1.parse_expressions()
training_data1 = cgp1.gen_training_data()
```

```
all variables: {'xb', 'xd', 'xc', 'xw', 'xa'}


learnable params: ['ab', 'bc', 'cd', 'ac']


dependencies: {'xw': ['xa', 'xb', 'xc', 'xd']}


expressions dict: {'xw': 'ab*xa+bc*xb+cd*xc+ac*xd'}


var_to_var_param dict:  {'xw': {'xa': 'ab', 'xb': 'bc', 'xc': 'cd', 'xd': 'ac'}}


node to int labels:  {'xa': 0, 'xb': 1, 'xc': 2, 'xd': 3, 'xw': 4}


independent vars: ['xb', 'xd', 'xc', 'xa']


leads_to dictionary: {'xb': {'xw'}, 'xd': {'xw'}, 'xc': {'xw'}, 'xw': set(), 'xa': {'xw'}}
```

In [5]:

```
cgp2 = Adam(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = 5 * 1e-3,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )


cgp2.parse_expressions()
training_data2 = cgp2.gen_training_data()
```

```
all variables: {'xb', 'xd', 'xc', 'xw', 'xa'}


learnable params: ['ab', 'bc', 'cd', 'ac']


dependencies: {'xw': ['xa', 'xb', 'xc', 'xd']}


expressions dict: {'xw': 'ab*xa+bc*xb+cd*xc+ac*xd'}


var_to_var_param dict:  {'xw': {'xa': 'ab', 'xb': 'bc', 'xc': 'cd', 'xd': 'ac'}}


node to int labels:  {'xa': 0, 'xb': 1, 'xc': 2, 'xd': 3, 'xw': 4}


independent vars: ['xb', 'xd', 'xc', 'xa']


leads_to dictionary: {'xb': {'xw'}, 'xd': {'xw'}, 'xc': {'xw'}, 'xw': set(), 'xa': {'xw'}}
```

In [6]:

```
loss1 = cgp1.run_training_loop_one_neuron_model(training_data1)
plt.plot(loss1,label = "SGD loss")

loss2 = cgp1.run_training_loop_one_neuron_model(training_data1,0.9,True)
plt.plot(loss2,label = "SGD+ loss")

loss3 = cgp2.run_training_loop_one_neuron_model(training_data2,0.9,0.99)
plt.plot(loss3,label = "Adam loss")

plt.xlabel("Iterations in hundreds")
plt.ylabel("Loss")
plt.title("One neuron loss using different optimizers")

plt.legend(loc = "upper right")
```

```
[iter=1]  loss = 0.0026
[iter=101]  loss = 0.3524
[iter=201]  loss = 0.3234
[iter=301]  loss = 0.3156
[iter=401]  loss = 0.3466
[iter=501]  loss = 0.3184
[iter=601]  loss = 0.3183
[iter=701]  loss = 0.3048
[iter=801]  loss = 0.3348
[iter=901]  loss = 0.2870
[iter=1001]  loss = 0.3142
[iter=1101]  loss = 0.3045
[iter=1201]  loss = 0.2978
[iter=1301]  loss = 0.3131
[iter=1401]  loss = 0.2880
[iter=1501]  loss = 0.2943
[iter=1601]  loss = 0.2921
[iter=1701]  loss = 0.2874
[iter=1801]  loss = 0.2859
```

In [ ]:

```

```

In [1]:

```
import random
import numpy as np
import matplotlib.pyplot as plt
from ComputationalGraphPrimer import *
import operator
```

In [2]:

```
class SGDPlus(ComputationalGraphPrimer):
    def __init__(self,*args,**kwargs):
        super().__init__(*args,**kwargs)

    def run_training_loop_one_neuron_model(self, training_data,mu=0.0,SGDplus=False):
        """
        The training loop must first initialize the learnable parameters.  Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}

        self.bias = random.uniform(0,1)                   ## Adding the bias improves class discrimination.
                                                          ##   We initialize it to a random number.

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, the training dataset
            consists of a Python dict with two keys, 0 and 1, the former points to a list of
            all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
            classes have different means and variances.  The dimensionality of each data sample
            is set by the number of nodes in the input layer of the neural network.

            The data loader's job is to construct a batch of samples drawn randomly from the two
            lists mentioned above.  And it mush also associate the class label with each sample
            separately.
            """
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]]   ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]   ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])

            def _getitem(self):
                cointoss = random.choice([0,1])                            ## When a batch is created by getbatch(), we want the
                                                                           ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]                            ## First list for samples, the second for labels
                maxval = 0.0                                               ## For approximate batch data normalization
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch

        ##My input start
        self.bias_update = 0.0
        self.step = [0]*(len(self.learnable_params)+1)
        self.mu = mu if SGDplus else 0.0

        ##My input end


        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0                            ##  Average the loss over iterations for printing out
                                                                   ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(data_tuples)              ##  FORWARD PROP of data
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ##  Find loss
            loss_avg = loss / float(len(class_labels))                                              ##  Average the loss over bat
```

```
            avg_loss_over_iterations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))            ## Display average loss
                avg_loss_over_iterations = 0.0                                                ## Re-initialize avg loss
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(class_labels))
            data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
            data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
                                 [float(len(class_labels))] * len(class_labels) ))
            self.backprop_and_update_params_one_neuron_model(y_error_avg, data_tuple_avg, deriv_sigmoid_avg)     ## BACKPROP loss

        return loss_running_record


    def forward_prop_one_neuron_model(self, data_tuples_in_batch):
        """
        Forward propagates the batch data through the neural network according to the equations on
        Slide 50 of my Week 3 slides.

        As the one-neuron model is characterized by a single expression, the main job of this function is
        to evaluate that expression for each data tuple in the incoming batch.  The resulting output is
        fed into the sigmoid activation function and the partial derivative of the sigmoid with respect
        to its input calculated.
        """
        output_vals = []
        deriv_sigmoids = []
        for vals_for_input_vars in data_tuples_in_batch:
            input_vars = self.independent_vars                    ## This is a list of vars for the input nodes. For the
                                                                  ##   the One-Neuron example in the Examples directory
                                                                  ##   this is just the list [xa, xb, xc, xd]
            vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))   ## The current values at input

            exp_obj = self.exp_objects[0]                         ## To understand this, first see the definition of the
                                                                  ##   Exp class (search for the string "class Exp").
                                                                  ##   Each expression that defines the neural network is
                                                                  ##   represented by one Exp instance by the parser.
            output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict, self.vals_for_learnable_params)

            ## [Search for "self.bias" in this file.]  As mentioned earlier, adding bias improves class discrimination:
            output_val = output_val + self.bias

            output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))   ## Apply sigmoid activation (output confined to [0.0,1.0] inte

            deriv_sigmoid = output_val * (1.0 - output_val)        ## See Slide 59 for why we need partial deriv of Sigmoid at in

            output_vals.append(output_val)                         ## Collect output values for different input samples in batch

            deriv_sigmoids.append(deriv_sigmoid)                   ## Collect the Sigmoid derivatives for each input sample in ba
                                                                   ##   The derivatives that are saved during forward prop are sh
        return output_vals, deriv_sigmoids


    def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars, deriv_sigmoid):
        """
        As should be evident from the syntax used in the following call to backprop function,

           self.backprop_and_update_params_one_neuron_model( y_error_avg, data_tuple_avg, deriv_sigmoid_avg)
                                                                     ^^^             ^^^                  ^^^
        the values fed to the backprop function for its three arguments are averaged over the training
        samples in the batch.  This in keeping with the spirit of SGD that calls for averaging the
        information retained in the forward propagation over the samples in a batch.

        See Slide 59 of my Week 3 slides for the math of back propagation for the One-Neuron network.
        """
        input_vars = self.independent_vars
        vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            ## My change start
            self.step[i] = (self.mu*self.step[i]) +  y_error * vals_for_input_vars_dict[input_vars[i]] * deriv_sigmoid

            ## Update the learnable parameters
            self.vals_for_learnable_params[param] += self.learning_rate*self.step[i]

        ## Update the bias
        self.bias_update = (self.mu*self.bias_update) + y_error * deriv_sigmoid
        self.bias += self.learning_rate*self.bias_update
            ##My change end
```

In [3]:

```
class Adam(ComputationalGraphPrimer):
    def __init__(self,*args,**kwargs):
        super().__init__(*args,**kwargs)

    def run_training_loop_one_neuron_model(self, training_data,beta1,beta2):
        """
        The training loop must first initialize the learnable parameters.  Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}

        self.bias = random.uniform(0,1)                   ## Adding the bias improves class discrimination.
                                                          ##   We initialize it to a random number.

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, the training dataset
            consists of a Python dict with two keys, 0 and 1, the former points to a list of
            all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
            classes have different means and variances.  The dimensionality of each data sample
            is set by the number of nodes in the input layer of the neural network.

            The data loader's job is to construct a batch of samples drawn randomly from the two
            lists mentioned above.  And it mush also associate the class label with each sample
            separately.
            """
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]]   ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]   ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])

            def _getitem(self):
                cointoss = random.choice([0,1])                            ## When a batch is created by getbatch(), we want the
                                                                           ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]                            ## First list for samples, the second for labels
                maxval = 0.0                                               ## For approximate batch data normalization
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch

        ##My input start
        self.bias_m = 0.0
        self.bias_v = 0.0

        self.bias_mh = 0.0
        self.bias_vh = 0.0

        self.step_m = [0]*(len(self.learnable_params)+1)
        self.step_v = [0]*(len(self.learnable_params)+1)
        self.step_mh = [0]*(len(self.learnable_params)+1)
        self.step_vh = [0]*(len(self.learnable_params)+1)

        self.beta1 = beta1
        self.beta2 = beta2
        self.m = 0

        ##My input end


        data_loader = DataLoader(training_data, batch_size=self.batch_size)
```

```
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0                                    ##  Average the loss over iterations for printing out
                                                                           ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            self.m = i+1
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(data_tuples)             ##  FORWARD PROP of data
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ##  Find loss
            loss_avg = loss / float(len(class_labels))                                               ##  Average the loss over bat
            avg_loss_over_iterations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))                 ## Display average loss
                avg_loss_over_iterations = 0.0                                                      ## Re-initialize avg loss
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(class_labels))
            data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
            data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
                                 [float(len(class_labels))] * len(class_labels) ))
            self.backprop_and_update_params_one_neuron_model(y_error_avg, data_tuple_avg, deriv_sigmoid_avg)     ## BACKPROP loss

        return loss_running_record


    def forward_prop_one_neuron_model(self, data_tuples_in_batch):
        """
        Forward propagates the batch data through the neural network according to the equations on
        Slide 50 of my Week 3 slides.

        As the one-neuron model is characterized by a single expression, the main job of this function is
        to evaluate that expression for each data tuple in the incoming batch.  The resulting output is
        fed into the sigmoid activation function and the partial derivative of the sigmoid with respect
        to its input calculated.
        """
        output_vals = []
        deriv_sigmoids = []
        for vals_for_input_vars in data_tuples_in_batch:
            input_vars = self.independent_vars                     ## This is a list of vars for the input nodes. For the
                                                                   ##   the One-Neuron example in the Examples directory
                                                                   ##   this is just the list [xa, xb, xc, xd]
            vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))   ## The current values at input

            exp_obj = self.exp_objects[0]                          ## To understand this, first see the definition of the
                                                                   ##   Exp class (search for the string "class Exp").
                                                                   ##   Each expression that defines the neural network is
                                                                   ##   represented by one Exp instance by the parser.
            output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict, self.vals_for_learnable_params)

            ## [Search for "self.bias" in this file.]  As mentioned earlier, adding bias improves class discrimination:
            output_val = output_val + self.bias

            output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))    ## Apply sigmoid activation (output confined to [0.0,1.0] inte

            deriv_sigmoid = output_val * (1.0 - output_val)         ## See Slide 59 for why we need partial deriv of Sigmoid at in

            output_vals.append(output_val)                          ## Collect output values for different input samples in batch

            deriv_sigmoids.append(deriv_sigmoid)                    ## Collect the Sigmoid derivatives for each input sample in ba
                                                                    ##   The derivatives that are saved during forward prop are sh
        return output_vals, deriv_sigmoids


    def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_vars, deriv_sigmoid):
        """
        As should be evident from the syntax used in the following call to backprop function,

           self.backprop_and_update_params_one_neuron_model( y_error_avg, data_tuple_avg, deriv_sigmoid_avg)
                                                                     ^^^             ^^^                 ^^^
        the values fed to the backprop function for its three arguments are averaged over the training
        samples in the batch.  This in keeping with the spirit of SGD that calls for averaging the
        information retained in the forward propagation over the samples in a batch.

        See Slide 59 of my Week 3 slides for the math of back propagation for the One-Neuron network.
        """
        input_vars = self.independent_vars
        vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            ## My change start
            self.step_m[i] = (self.beta1*self.step_m[i]) + (1-self.beta1)*(y_error * vals_for_input_vars_dict[input_vars[i]] * de
            self.step_mh[i] = self.step_m[i]/(1-self.beta1**self.m)
```

```
            self.step_v[i] = (self.beta2*self.step_v[i]) + (1-self.beta2)*((y_error * vals_for_input_vars_dict[input_vars[i]] * d
            self.step_vh[i] = self.step_v[i]/(1-self.beta2**self.m)

            ## Update the learnable parameters
            self.vals_for_learnable_params[param] += self.learning_rate * (self.step_mh[i]/(np.sqrt(self.step_vh[i])+10**-6))

        ## Update the bias
        self.bias_m = (self.beta1*self.bias_m) + (1-self.beta1)*(y_error * deriv_sigmoid)
        self.bias_mh = self.bias_m/(1-self.beta1**self.m)
        self.bias_v = (self.beta2*self.bias_v) + (1-self.beta2)*((y_error * deriv_sigmoid)**2)
        self.bias_vh = self.bias_v/(1-self.beta2**self.m)
        self.bias += self.learning_rate * (self.bias_m/(np.sqrt(self.bias_v)+10**-6))
            ##My change end
```

In [4]:

```
cgp1 = SGDPlus(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = 1e-2,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )


cgp1.parse_expressions()
training_data1 = cgp1.gen_training_data()
```

```
all variables: {'xa', 'xb', 'xc', 'xw', 'xd'}


learnable params: ['ab', 'bc', 'cd', 'ac']


dependencies: {'xw': ['xa', 'xb', 'xc', 'xd']}


expressions dict: {'xw': 'ab*xa+bc*xb+cd*xc+ac*xd'}


var_to_var_param dict:  {'xw': {'xa': 'ab', 'xb': 'bc', 'xc': 'cd', 'xd': 'ac'}}


node to int labels:  {'xa': 0, 'xb': 1, 'xc': 2, 'xd': 3, 'xw': 4}


independent vars: ['xa', 'xb', 'xc', 'xd']


leads_to dictionary: {'xa': {'xw'}, 'xb': {'xw'}, 'xc': {'xw'}, 'xw': set(), 'xd': {'xw'}}
```

In [5]:

```
cgp2 = Adam(
               one_neuron_model = True,
               expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
               output_vars = ['xw'],
               dataset_size = 5000,
               learning_rate = 1e-2,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )


cgp2.parse_expressions()
training_data2 = cgp2.gen_training_data()
```

```
all variables: {'xa', 'xb', 'xc', 'xw', 'xd'}


learnable params: ['ab', 'bc', 'cd', 'ac']


dependencies: {'xw': ['xa', 'xb', 'xc', 'xd']}


expressions dict: {'xw': 'ab*xa+bc*xb+cd*xc+ac*xd'}


var_to_var_param dict:  {'xw': {'xa': 'ab', 'xb': 'bc', 'xc': 'cd', 'xd': 'ac'}}


node to int labels:  {'xa': 0, 'xb': 1, 'xc': 2, 'xd': 3, 'xw': 4}


independent vars: ['xa', 'xb', 'xc', 'xd']


leads_to dictionary: {'xa': {'xw'}, 'xb': {'xw'}, 'xc': {'xw'}, 'xw': set(), 'xd': {'xw'}}
```

In [6]:

```
loss1 = cgp1.run_training_loop_one_neuron_model(training_data1)
plt.plot(loss1,label = "SGD loss")

loss2 = cgp1.run_training_loop_one_neuron_model(training_data1,0.9,True)
plt.plot(loss2,label = "SGD+ loss")

loss3 = cgp2.run_training_loop_one_neuron_model(training_data2,0.9,0.99)
plt.plot(loss3,label = "Adam loss")

plt.xlabel("Iterations in hundreds")
plt.ylabel("Loss")
plt.title("One neuron loss using different optimizers")

plt.legend(loc = "upper right")
```

```
[iter=1]  loss = 0.0030
[iter=101]  loss = 0.3332
[iter=201]  loss = 0.3052
[iter=301]  loss = 0.3036
[iter=401]  loss = 0.2904
[iter=501]  loss = 0.2811
[iter=601]  loss = 0.2984
[iter=701]  loss = 0.2837
[iter=801]  loss = 0.2645
[iter=901]  loss = 0.2650
[iter=1001]  loss = 0.2590
[iter=1101]  loss = 0.2575
[iter=1201]  loss = 0.2500
[iter=1301]  loss = 0.2474
[iter=1401]  loss = 0.2500
[iter=1501]  loss = 0.2489
[iter=1601]  loss = 0.2460
[iter=1701]  loss = 0.2482
[iter=1801]  loss = 0.2506
```

In [ ]:

```

```

In [1]:

```
import random
import numpy as np
import matplotlib.pyplot as plt
from ComputationalGraphPrimer import *
import operator
from numpy import nan
```

In [2]:

```python
class SGDPlus(ComputationalGraphPrimer):
    def __init__(self,*args,**kwargs):
        super().__init__(*args,**kwargs)

    def run_training_loop_multi_neuron_model(self, training_data,mu=0.0,SGDplus=False):

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, the training dataset
            consists of a Python dict with two keys, 0 and 1, the former points to a list of
            all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
            classes have different means and variances.  The dimensionality of each data sample
            is set by the number of nodes in the input layer of the neural network.

            The data loader's job is to construct a batch of samples drawn randomly from the two
            lists mentioned above.  And it mush also associate the class label with each sample
            separately.
            """
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]]    ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]    ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])

            def _getitem(self):
                cointoss = random.choice([0,1])                            ## When a batch is created by getbatch(), we want the
                                                                           ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]                            ## First list for samples, the second for labels
                maxval = 0.0                                               ## For approximate batch data normalization
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch


        """
        The training loop must first initialize the learnable parameters.  Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}

        self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]      ## Adding the bias to each layer improves
                                                                                 ##   class discrimination. We initialize it
                                                                                 ##   to a random number.

        ##My input start
        self.bias_update = [0.0]*(self.num_layers+1)
        self.step = [[0]*(len(self.learnable_params)+1)]*(self.num_layers+1)
        self.mu = mu if SGDplus else 0.0

        ##My input end


        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0                                          ##  Average the loss over iterations for printing out
                                                                                 ##    every N iterations during the training loop.
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            self.forward_prop_multi_neuron_model(data_tuples)                              ## FORW PROP works by side-effect
            predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.num_layers-1]      ## Predictions from FORW PROP
            y_preds =  [item for sublist in  predicted_labels_for_batch  for item in sublist]  ## Get numeric vals for predictions
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ## Calculate loss for batch
            loss_avg = loss / float(len(class_labels))                                          ## Average the loss over batch
            avg_loss_over_iterations += loss_avg                                               ## Add to Average loss over iterations
```

```
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))          ## Display avg loss
                avg_loss_over_iterations = 0.0                                              ## Re-initialize avg-over-iterations loss
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels)      ## BACKPROP loss
        return loss_running_record



    def forward_prop_multi_neuron_model(self, data_tuples_in_batch):
        """
        During forward propagation, we push each batch of the input data through the
        network.  In order to explain the logic of forward, consider the following network
        layout in 4 nodes in the input layer, 2 nodes in the hidden layer, and 1 node in
        the output layer.

                               input

                                 x                                          x = node

                                 x         x|                               | = sigmoid activation
                                                     x|
                                 x         x|

                                 x

                             layer_0    layer_1    layer_2


        In the code shown below, the expressions to evaluate for computing the
        pre-activation values at a node are stored at the layer in which the nodes reside.
        That is, the dictionary look-up "self.layer_exp_objects[layer_index]" returns the
        Expression objects for which the left-side dependent variable is in the layer
        pointed to layer_index.  So the example shown above, "self.layer_exp_objects[1]"
        will return two Expression objects, one for each of the two nodes in the second
        layer of the network (that is, layer indexed 1).

        The pre-activation values obtained by evaluating the expressions at each node are
        then subject to Sigmoid activation, followed by the calculation of the partial
        derivative of the output of the Sigmoid function with respect to its input.

        In the forward, the values calculated for the nodes in each layer are stored in
        the dictionary

                        self.forw_prop_vals_at_layers[ layer_index ]

        and the gradients values calculated at the same nodes in the dictionary:

                        self.gradient_vals_for_layers[ layer_index ]

        """
        self.forw_prop_vals_at_layers = {i : [] for i in range(self.num_layers)}
        self.gradient_vals_for_layers = {i : [] for i in range(1, self.num_layers)}
        for vals_for_input_vars in data_tuples_in_batch:
            self.forw_prop_vals_at_layers[0].append(vals_for_input_vars)
            for layer_index in range(1, self.num_layers):
                input_vars = self.layer_vars[layer_index-1]
                if layer_index == 1:
                    vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
                output_vals_arr = []
                gradients_val_arr = []
                ##  In the following loop for forward propagation calculations, exp_obj is the Exp object
                ##  that is created for each user-supplied expression that specifies the network.  See the
                ##  definition of the class Exp (for 'Expression') by searching for "class Exp":
                for exp_obj in self.layer_exp_objects[layer_index]:
                    output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict,
                                                                     self.vals_for_learnable_params, input_vars)
                    ## [Search for "self.bias" in this file.]  As mentioned earlier, adding bias to each
                    ##  layer improves class discrimination:
                    output_val = output_val + self.bias[layer_index-1]
                    ## apply sigmoid activation:
                    output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))
                    output_vals_arr.append(output_val)
                    ## calculate partial of the activation function as a function of its input
                    deriv_sigmoid = output_val * (1.0 - output_val)
                    gradients_val_arr.append(deriv_sigmoid)
                    vals_for_input_vars_dict[ exp_obj.dependent_var ] = output_val
                self.forw_prop_vals_at_layers[layer_index].append(output_vals_arr)
                ##  See the bullets in red on Slides 70 and 73 of my Week 3 slides for what needs
                ##  to be stored during the forward propagation of data through the network:
                self.gradient_vals_for_layers[layer_index].append(gradients_val_arr)


    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels):
        """
        First note that loop index variable 'back_layer_index' starts with the index of
        the last layer.  For the 3-layer example shown for 'forward', back_layer_index
        starts with a value of 2, its next value is 1, and that's it.
```

```
        Stochastic Gradient Gradient calls for the backpropagated loss to be averaged over
        the samples in a batch.  To explain how this averaging is carried out by the
        backprop function, consider the last node on the example shown in the forward()
        function above.  Standing at the node, we look at the 'input' values stored in the
        variable "input_vals".  Assuming a batch size of 8, this will be list of
        lists. Each of the inner lists will have two values for the two nodes in the
        hidden layer. And there will be 8 of these for the 8 elements of the batch.  We average
        these values 'input vals' and store those in the variable "input_vals_avg".  Next we
        must carry out the same batch-based averaging for the partial derivatives stored in the
        variable "deriv_sigmoid".

        Pay attention to the variable 'vars_in_layer'.  These store the node variables in
        the current layer during backpropagation.  Since back_layer_index starts with a
        value of 2, the variable 'vars_in_layer' will have just the single node for the
        example shown for forward(). With respect to what is stored in vars_in_layer', the
        variables stored in 'input_vars_to_layer' correspond to the input layer with
        respect to the current layer.
        """
        # backproped prediction error:
        pred_err_backproped_at_layers = {i : [] for i in range(1,self.num_layers-1)}
        pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
        for back_layer_index in reversed(range(1,self.num_layers)):
            input_vals = self.forw_prop_vals_at_layers[back_layer_index -1]
            input_vals_avg = [sum(x) for x in zip(*input_vals)]
            input_vals_avg = list(map(operator.truediv, input_vals_avg, [float(len(class_labels))] * len(class_labels)))
            deriv_sigmoid =  self.gradient_vals_for_layers[back_layer_index]
            deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
            deriv_sigmoid_avg = list(map(operator.truediv, deriv_sigmoid_avg,
                                                          [float(len(class_labels))] * len(class_labels)))
            vars_in_layer  =  self.layer_vars[back_layer_index]                 ## a list like ['xo']
            vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]   ## a list like ['xw', 'xz']

            layer_params = self.layer_params[back_layer_index]
            ## note that layer_params are stored in a dict like
                ##     {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']], 2: [['cp', 'cq']]}
            ## "layer_params[idx]" is a list of lists for the link weights in layer whose output nodes are in layer "idx"
            transposed_layer_params = list(zip(*layer_params))         ## creating a transpose of the link matrix

            backproped_error = [None] * len(vars_in_next_layer_back)
            for k,varr in enumerate(vars_in_next_layer_back):
                for j,var2 in enumerate(vars_in_layer):
                    backproped_error[k] = sum([self.vals_for_learnable_params[transposed_layer_params[k][i]] *
                                               pred_err_backproped_at_layers[back_layer_index][i]
                                               for i in range(len(vars_in_layer))])
#                                               deriv_sigmoid_avg[i] for i in range(len(vars_in_layer))])
            pred_err_backproped_at_layers[back_layer_index - 1]  =  backproped_error
            input_vars_to_layer = self.layer_vars[back_layer_index-1]
            for j,var in enumerate(vars_in_layer):
                layer_params = self.layer_params[back_layer_index][j]
                ##  Regarding the parameter update loop that follows, see the Slides 74 through 77 of my Week 3
                ##  lecture slides for how the parameters are updated using the partial derivatives stored away
                ##  during forward propagation of data. The theory underlying these calculations is presented
                ##  in Slides 68 through 71.
                for i,param in enumerate(layer_params):
                    gradient_of_loss_for_param = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j]

                    #My change start
                    self.step[back_layer_index-1][i] = (self.mu*self.step[back_layer_index-1][i]) +  gradient_of_loss_for_param * deriv_s


                    self.vals_for_learnable_params[param] += self.step[back_layer_index-1][i]*self.learning_rate


            self.bias_update[back_layer_index-1] = (self.mu*self.bias_update[back_layer_index-1]) + sum(pred_err_backproped_at_layers[bac
                                                                * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)
            self.bias[back_layer_index-1] += self.learning_rate * self.bias_update[back_layer_index-1]

            ##My change end
    ########################################################################################################
```

In [3]:

```
class Adam(ComputationalGraphPrimer):
    def __init__(self,*args,**kwargs):
        super().__init__(*args,**kwargs)

    def run_training_loop_multi_neuron_model(self, training_data,beta1,beta2):

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, the training dataset
            consists of a Python dict with two keys, 0 and 1, the former points to a list of
            all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
            classes have different means and variances.  The dimensionality of each data sample
            is set by the number of nodes in the input layer of the neural network.

            The data loader's job is to construct a batch of samples drawn randomly from the two
            lists mentioned above.  And it mush also associate the class label with each sample
            separately.
            """
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]]    ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]    ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])

            def _getitem(self):
                cointoss = random.choice([0,1])                            ## When a batch is created by getbatch(), we want the
                                                                           ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]                            ## First list for samples, the second for labels
                maxval = 0.0                                               ## For approximate batch data normalization
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch


        """
        The training loop must first initialize the learnable parameters.  Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}

        self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]      ## Adding the bias to each layer improves
                                                                                 ##   class discrimination. We initialize it
                                                                                 ##   to a random number.

        ##My input start
        self.bias_m = [0.0]*(self.num_layers+1)
        self.bias_v = [0.0]*(self.num_layers+1)
        self.bias_mh = [0.0]*(self.num_layers+1)
        self.bias_vh = [0.0]*(self.num_layers+1)

        self.step_m = [[0]*(len(self.learnable_params)+1)]*(self.num_layers+1)
        self.step_v = [[0]*(len(self.learnable_params)+1)]*(self.num_layers+1)
        self.step_mh = [[0]*(len(self.learnable_params)+1)]*(self.num_layers+1)
        self.step_vh = [[0]*(len(self.learnable_params)+1)]*(self.num_layers+1)

        self.beta1 = beta1
        self.beta2 = beta2
        self.m = 0

        ##My input end


        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0                                          ##  Average the loss over iterations for printing out
                                                                                 ##    every N iterations during the training loop.
```

```
        for i in range(self.training_iterations):
            self.m = i+1
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            self.forward_prop_multi_neuron_model(data_tuples)                                  ## FORW PROP works by side-effect
            predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.num_layers-1]      ## Predictions from FORW PROP
            y_preds =  [item for sublist in  predicted_labels_for_batch  for item in sublist]  ## Get numeric vals for predictions
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ## Calculate loss for batch
            loss_avg = loss / float(len(class_labels))                                         ## Average the loss over batch
            avg_loss_over_iterations += loss_avg                                              ## Add to Average loss over iterations
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))            ## Display avg loss
                avg_loss_over_iterations = 0.0                                                ## Re-initialize avg-over-iterations loss
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels)      ## BACKPROP loss
        return loss_running_record



    def forward_prop_multi_neuron_model(self, data_tuples_in_batch):
        """
        During forward propagation, we push each batch of the input data through the
        network.  In order to explain the logic of forward, consider the following network
        layout in 4 nodes in the input layer, 2 nodes in the hidden layer, and 1 node in
        the output layer.

                               input

                                 x                                          x = node

                                 x         x|                               | = sigmoid activation
                                                     x|
                                 x         x|

                                 x

                             layer_0    layer_1    layer_2


        In the code shown below, the expressions to evaluate for computing the
        pre-activation values at a node are stored at the layer in which the nodes reside.
        That is, the dictionary look-up "self.layer_exp_objects[layer_index]" returns the
        Expression objects for which the left-side dependent variable is in the layer
        pointed to layer_index.  So the example shown above, "self.layer_exp_objects[1]"
        will return two Expression objects, one for each of the two nodes in the second
        layer of the network (that is, layer indexed 1).

        The pre-activation values obtained by evaluating the expressions at each node are
        then subject to Sigmoid activation, followed by the calculation of the partial
        derivative of the output of the Sigmoid function with respect to its input.

        In the forward, the values calculated for the nodes in each layer are stored in
        the dictionary

                        self.forw_prop_vals_at_layers[ layer_index ]

        and the gradients values calculated at the same nodes in the dictionary:

                        self.gradient_vals_for_layers[ layer_index ]

        """
        self.forw_prop_vals_at_layers = {i : [] for i in range(self.num_layers)}
        self.gradient_vals_for_layers = {i : [] for i in range(1, self.num_layers)}
        for vals_for_input_vars in data_tuples_in_batch:
            self.forw_prop_vals_at_layers[0].append(vals_for_input_vars)
            for layer_index in range(1, self.num_layers):
                input_vars = self.layer_vars[layer_index-1]
                if layer_index == 1:
                    vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
                output_vals_arr = []
                gradients_val_arr = []
                ##  In the following loop for forward propagation calculations, exp_obj is the Exp object
                ##  that is created for each user-supplied expression that specifies the network.  See the
                ##  definition of the class Exp (for 'Expression') by searching for "class Exp":
                for exp_obj in self.layer_exp_objects[layer_index]:
                    output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict,
                                                                     self.vals_for_learnable_params, input_vars)
                    ## [Search for "self.bias" in this file.]  As mentioned earlier, adding bias to each
                    ##  layer improves class discrimination:
                    output_val = output_val + self.bias[layer_index-1]
                    ## apply sigmoid activation:
                    output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))
                    output_vals_arr.append(output_val)
                    ## calculate partial of the activation function as a function of its input
                    deriv_sigmoid = output_val * (1.0 - output_val)
                    gradients_val_arr.append(deriv_sigmoid)
                    vals_for_input_vars_dict[ exp_obj.dependent_var ] = output_val
                self.forw_prop_vals_at_layers[layer_index].append(output_vals_arr)
```

```
            ##  See the bullets in red on Slides 70 and 73 of my Week 3 slides for what needs
            ##  to be stored during the forward propagation of data through the network:
            self.gradient_vals_for_layers[layer_index].append(gradients_val_arr)


    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels):
        """
        First note that loop index variable 'back_layer_index' starts with the index of
        the last layer.  For the 3-layer example shown for 'forward', back_layer_index
        starts with a value of 2, its next value is 1, and that's it.

        Stochastic Gradient Gradient calls for the backpropagated loss to be averaged over
        the samples in a batch.  To explain how this averaging is carried out by the
        backprop function, consider the last node on the example shown in the forward()
        function above.  Standing at the node, we look at the 'input' values stored in the
        variable "input_vals".  Assuming a batch size of 8, this will be list of
        lists. Each of the inner lists will have two values for the two nodes in the
        hidden layer. And there will be 8 of these for the 8 elements of the batch.  We average
        these values 'input vals' and store those in the variable "input_vals_avg".  Next we
        must carry out the same batch-based averaging for the partial derivatives stored in the
        variable "deriv_sigmoid".

        Pay attention to the variable 'vars_in_layer'.  These store the node variables in
        the current layer during backpropagation.  Since back_layer_index starts with a
        value of 2, the variable 'vars_in_layer' will have just the single node for the
        example shown for forward(). With respect to what is stored in vars_in_layer', the
        variables stored in 'input_vars_to_layer' correspond to the input layer with
        respect to the current layer.
        """
        # backproped prediction error:
        pred_err_backproped_at_layers = {i : [] for i in range(1,self.num_layers-1)}
        pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
        for back_layer_index in reversed(range(1,self.num_layers)):
            input_vals = self.forw_prop_vals_at_layers[back_layer_index -1]
            input_vals_avg = [sum(x) for x in zip(*input_vals)]
            input_vals_avg = list(map(operator.truediv, input_vals_avg, [float(len(class_labels))] * len(class_labels)))
            deriv_sigmoid =  self.gradient_vals_for_layers[back_layer_index]
            deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
            deriv_sigmoid_avg = list(map(operator.truediv, deriv_sigmoid_avg,
                                                             [float(len(class_labels))] * len(class_labels)))
            vars_in_layer  =  self.layer_vars[back_layer_index]                 ## a list like ['xo']
            vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]   ## a list like ['xw', 'xz']

            layer_params = self.layer_params[back_layer_index]
            ## note that layer_params are stored in a dict like
                ##     {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']], 2: [['cp', 'cq']]}
            ## "layer_params[idx]" is a list of lists for the link weights in layer whose output nodes are in layer "idx"
            transposed_layer_params = list(zip(*layer_params))         ## creating a transpose of the link matrix

            backproped_error = [None] * len(vars_in_next_layer_back)
            for k,varr in enumerate(vars_in_next_layer_back):
                for j,var2 in enumerate(vars_in_layer):
                    backproped_error[k] = sum([self.vals_for_learnable_params[transposed_layer_params[k][i]] *
                                               pred_err_backproped_at_layers[back_layer_index][i]
                                               for i in range(len(vars_in_layer))])
#                                               deriv_sigmoid_avg[i] for i in range(len(vars_in_layer))])
            pred_err_backproped_at_layers[back_layer_index - 1]  =  backproped_error
            input_vars_to_layer = self.layer_vars[back_layer_index-1]
            for j,var in enumerate(vars_in_layer):
                layer_params = self.layer_params[back_layer_index][j]
                ##  Regarding the parameter update loop that follows, see the Slides 74 through 77 of my Week 3
                ##  lecture slides for how the parameters are updated using the partial derivatives stored away
                ##  during forward propagation of data. The theory underlying these calculations is presented
                ##  in Slides 68 through 71.
                for i,param in enumerate(layer_params):
                    gradient_of_loss_for_param = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j]

                    #My change start
                    self.step_m[back_layer_index-1][i] = (self.beta1*self.step_m[back_layer_index-1][i]) + (1-self.beta1)*(gradient_of_los
                    self.step_mh[back_layer_index-1][i] = self.step_m[back_layer_index-1][i]/(1-(self.beta1**self.m))
                    self.step_v[back_layer_index-1][i] = (self.beta2*self.step_v[back_layer_index-1][i]) + (1-self.beta2)*((gradient_of_l
                    self.step_vh[back_layer_index-1][i] = self.step_v[back_layer_index-1][i]/(1-(self.beta2**self.m))
                    self.vals_for_learnable_params[param] += self.learning_rate * (self.step_mh[back_layer_index-1][i]/(np.sqrt(self.step_


            self.bias_m[back_layer_index-1] = (self.beta1*self.bias_m[back_layer_index-1]) + (1-self.beta1)*(sum(pred_err_backproped_at_l
                                                                        * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg))
            self.bias_mh[back_layer_index-1] = self.bias_m[back_layer_index-1]/(1-(self.beta1**self.m))
            self.bias_v[back_layer_index-1] = (self.beta2*self.bias_v[back_layer_index-1]) + (1-self.beta2)*((sum(pred_err_backproped_at_
                                                                        * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg))**2)
            self.bias_vh[back_layer_index-1] = self.bias_v[back_layer_index-1]/(1-(self.beta2**self.m))
            self.bias[back_layer_index-1] += self.learning_rate * (self.bias_mh[back_layer_index-1]/(np.sqrt(self.bias_vh[back_layer_inde

            ##My change end
    ######################################################################################################
```

In [4]:

```
cgp1 = SGDPlus(
               num_layers = 3,
               layers_config = [4,2,1],                        # num of nodes in each layer
               expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                              'xz=bp*xp+bq*xq+br*xr+bs*xs',
                              'xo=cp*xw+cq*xz'],
               output_vars = ['xo'],
               dataset_size = 5000,
               learning_rate = 5e-3,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )

cgp1.parse_multi_layer_expressions()
training_data1 = cgp1.gen_training_data()
```

```
self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs*xs'], 2: ['xo=cp*xw+cq*xz']}


[layer index: 1] all variables: {'xw', 'xz', 'xq', 'xr', 'xs', 'xp'}


[layer index: 1] learnable params: {'aq', 'as', 'bp', 'br', 'bs', 'bq', 'ar', 'ap'}


[layer index: 1] dependencies: {'xw': ['xp', 'xq', 'xr', 'xs'], 'xz': ['xp', 'xq', 'xr', 'xs']}


[layer index: 1] expressions dict: {'xw': 'ap*xp+aq*xq+ar*xr+as*xs', 'xz': 'bp*xp+bq*xq+br*xr+bs*xs'}


[layer index: 1] var_to_var_param dict: {'xw': {'xp': 'ap', 'xq': 'aq', 'xr': 'ar', 'xs': 'as'}, 'xz': {'xp': 'bp', 'x
q': 'bq', 'xr': 'br', 'xs': 'bs'}}
```

In [5]:

```
cgp2 = Adam(
               num_layers = 3,
               layers_config = [4,2,1],                        # num of nodes in each layer
               expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                              'xz=bp*xp+bq*xq+br*xr+bs*xs',
                              'xo=cp*xw+cq*xz'],
               output_vars = ['xo'],
               dataset_size = 5000,
               learning_rate = 5e-3,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )

cgp2.parse_multi_layer_expressions()
training_data2 = cgp2.gen_training_data()
```

```
self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs*xs'], 2: ['xo=cp*xw+cq*xz']}


[layer index: 1] all variables: {'xw', 'xz', 'xq', 'xr', 'xs', 'xp'}


[layer index: 1] learnable params: {'aq', 'as', 'bp', 'br', 'bs', 'bq', 'ar', 'ap'}


[layer index: 1] dependencies: {'xw': ['xp', 'xq', 'xr', 'xs'], 'xz': ['xp', 'xq', 'xr', 'xs']}


[layer index: 1] expressions dict: {'xw': 'ap*xp+aq*xq+ar*xr+as*xs', 'xz': 'bp*xp+bq*xq+br*xr+bs*xs'}


[layer index: 1] var_to_var_param dict: {'xw': {'xp': 'ap', 'xq': 'aq', 'xr': 'ar', 'xs': 'as'}, 'xz': {'xp': 'bp', 'x
q': 'bq', 'xr': 'br', 'xs': 'bs'}}
```

In [6]:

```
loss3 = cgp2.run_training_loop_multi_neuron_model(training_data2,0.9,0.99)
plt.plot(loss3,label = "Adam loss")


loss1 = cgp1.run_training_loop_multi_neuron_model(training_data1)
plt.plot(loss1,label = "SGD loss")

loss2 = cgp1.run_training_loop_multi_neuron_model(training_data1,0.9,True)
plt.plot(loss2,label = "SGD+ loss")



plt.xlabel("Iterations in hundreds")
plt.ylabel("Loss")
plt.title("Multi neuron loss using different optimizers")

plt.legend(loc = "upper right")
```

```
[iter=1]  loss = 0.0048
[iter=101]  loss = 0.3295
[iter=201]  loss = 0.2680
[iter=301]  loss = 0.2533
[iter=401]  loss = 0.2507
[iter=501]  loss = 0.2506
[iter=601]  loss = 0.2506
[iter=701]  loss = 0.2506
[iter=801]  loss = 0.2504
[iter=901]  loss = 0.2505
[iter=1001]  loss = 0.2506
[iter=1101]  loss = 0.2502
[iter=1201]  loss = 0.2506
[iter=1301]  loss = 0.2507
[iter=1401]  loss = 0.2507
[iter=1501]  loss = 0.2504
[iter=1601]  loss = 0.2497
[iter=1701]  loss = 0.2503
[iter=1801]  loss = 0.2511
```

In [ ]:

In [1]:

```
import random
import numpy as np
import matplotlib.pyplot as plt
from ComputationalGraphPrimer import *
import operator
from numpy import nan
```

In [2]:

```
class SGDPlus(ComputationalGraphPrimer):
    def __init__(self,*args,**kwargs):
        super().__init__(*args,**kwargs)

    def run_training_loop_multi_neuron_model(self, training_data,mu=0.0,SGDplus=False):

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, the training dataset
            consists of a Python dict with two keys, 0 and 1, the former points to a list of
            all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
            classes have different means and variances.  The dimensionality of each data sample
            is set by the number of nodes in the input layer of the neural network.

            The data loader's job is to construct a batch of samples drawn randomly from the two
            lists mentioned above.  And it mush also associate the class label with each sample
            separately.
            """
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]]    ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]    ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])

            def _getitem(self):
                cointoss = random.choice([0,1])                            ## When a batch is created by getbatch(), we want the
                                                                           ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]                            ## First list for samples, the second for labels
                maxval = 0.0                                               ## For approximate batch data normalization
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch


        """
        The training loop must first initialize the learnable parameters.  Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}

        self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]      ## Adding the bias to each layer improves
                                                                                 ##   class discrimination. We initialize it
                                                                                 ##   to a random number.

        ##My input start
        self.bias_update = [0.0]*(self.num_layers+1)
        self.step = [[0]*(len(self.learnable_params)+1)]*(self.num_layers+1)
        self.mu = mu if SGDplus else 0.0

        ##My input end


        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0                                          ##  Average the loss over iterations for printing
                                                                                 ##    every N iterations during the training loo
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
```

```
            self.forward_prop_multi_neuron_model(data_tuples)                    ## FORW PROP works by side-effect
            predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.num_layers-1]      ## Predictions from FORW PROP
            y_preds =  [item for sublist in  predicted_labels_for_batch  for item in sublist]  ## Get numeric vals for prediction
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ## Calculate loss for batch
            loss_avg = loss / float(len(class_labels))                           ## Average the loss over batch
            avg_loss_over_iterations += loss_avg                                 ## Add to Average loss over iterati
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))          ## Display avg loss
                avg_loss_over_iterations = 0.0                                   ## Re-initialize avg-over-iteration
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels)      ## BACKPROP loss
        return loss_running_record



    def forward_prop_multi_neuron_model(self, data_tuples_in_batch):
        """
        During forward propagation, we push each batch of the input data through the
        network.  In order to explain the logic of forward, consider the following network
        layout in 4 nodes in the input layer, 2 nodes in the hidden layer, and 1 node in
        the output layer.

                               input

                                 x                                       x = node

                                 x         x|                            | = sigmoid activation
                                                     x|
                                 x         x|

                                 x

                             layer_0    layer_1    layer_2


        In the code shown below, the expressions to evaluate for computing the
        pre-activation values at a node are stored at the layer in which the nodes reside.
        That is, the dictionary look-up "self.layer_exp_objects[layer_index]" returns the
        Expression objects for which the left-side dependent variable is in the layer
        pointed to layer_index.  So the example shown above, "self.layer_exp_objects[1]"
        will return two Expression objects, one for each of the two nodes in the second
        layer of the network (that is, layer indexed 1).

        The pre-activation values obtained by evaluating the expressions at each node are
        then subject to Sigmoid activation, followed by the calculation of the partial
        derivative of the output of the Sigmoid function with respect to its input.

        In the forward, the values calculated for the nodes in each layer are stored in
        the dictionary

                        self.forw_prop_vals_at_layers[ layer_index ]

        and the gradients values calculated at the same nodes in the dictionary:

                        self.gradient_vals_for_layers[ layer_index ]

        """
        self.forw_prop_vals_at_layers = {i : [] for i in range(self.num_layers)}
        self.gradient_vals_for_layers = {i : [] for i in range(1, self.num_layers)}
        for vals_for_input_vars in data_tuples_in_batch:
            self.forw_prop_vals_at_layers[0].append(vals_for_input_vars)
            for layer_index in range(1, self.num_layers):
                input_vars = self.layer_vars[layer_index-1]
                if layer_index == 1:
                    vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
                output_vals_arr = []
                gradients_val_arr = []
                ##  In the following loop for forward propagation calculations, exp_obj is the Exp object
                ##  that is created for each user-supplied expression that specifies the network.  See the
                ##  definition of the class Exp (for 'Expression') by searching for "class Exp":
                for exp_obj in self.layer_exp_objects[layer_index]:
                    output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict,
                                                                      self.vals_for_learnable_params, input_vars)
                    ## [Search for "self.bias" in this file.]  As mentioned earlier, adding bias to each
                    ##  layer improves class discrimination:
                    output_val = output_val + self.bias[layer_index-1]
                    ## apply sigmoid activation:
                    output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))
                    output_vals_arr.append(output_val)
                    ## calculate partial of the activation function as a function of its input
                    deriv_sigmoid = output_val * (1.0 - output_val)
                    gradients_val_arr.append(deriv_sigmoid)
                    vals_for_input_vars_dict[ exp_obj.dependent_var ] = output_val
```

```
            self.forw_prop_vals_at_layers[layer_index].append(output_vals_arr)
            ##  See the bullets in red on Slides 70 and 73 of my Week 3 slides for what needs
            ##  to be stored during the forward propagation of data through the network:
            self.gradient_vals_for_layers[layer_index].append(gradients_val_arr)


    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels):
        """
        First note that loop index variable 'back_layer_index' starts with the index of
        the last layer.  For the 3-layer example shown for 'forward', back_layer_index
        starts with a value of 2, its next value is 1, and that's it.

        Stochastic Gradient Gradient calls for the backpropagated loss to be averaged over
        the samples in a batch.  To explain how this averaging is carried out by the
        backprop function, consider the last node on the example shown in the forward()
        function above.  Standing at the node, we look at the 'input' values stored in the
        variable "input_vals".  Assuming a batch size of 8, this will be list of
        lists. Each of the inner lists will have two values for the two nodes in the
        hidden layer. And there will be 8 of these for the 8 elements of the batch.  We average
        these values 'input vals' and store those in the variable "input_vals_avg".  Next we
        must carry out the same batch-based averaging for the partial derivatives stored in the
        variable "deriv_sigmoid".

        Pay attention to the variable 'vars_in_layer'.  These store the node variables in
        the current layer during backpropagation.  Since back_layer_index starts with a
        value of 2, the variable 'vars_in_layer' will have just the single node for the
        example shown for forward(). With respect to what is stored in vars_in_layer', the
        variables stored in 'input_vars_to_layer' correspond to the input layer with
        respect to the current layer.
        """
        # backproped prediction error:
        pred_err_backproped_at_layers = {i : [] for i in range(1,self.num_layers-1)}
        pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
        for back_layer_index in reversed(range(1,self.num_layers)):
            input_vals = self.forw_prop_vals_at_layers[back_layer_index -1]
            input_vals_avg = [sum(x) for x in zip(*input_vals)]
            input_vals_avg = list(map(operator.truediv, input_vals_avg, [float(len(class_labels))] * len(class_labels)))
            deriv_sigmoid =  self.gradient_vals_for_layers[back_layer_index]
            deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
            deriv_sigmoid_avg = list(map(operator.truediv, deriv_sigmoid_avg,
                                                             [float(len(class_labels))] * len(class_labels)))
            vars_in_layer  =  self.layer_vars[back_layer_index]                  ## a list like ['xo']
            vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]   ## a list like ['xw', 'xz']

            layer_params = self.layer_params[back_layer_index]
            ## note that layer_params are stored in a dict like
                ##     {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']], 2: [['cp', 'cq']]}
            ## "layer_params[idx]" is a list of lists for the link weights in layer whose output nodes are in layer "idx"
            transposed_layer_params = list(zip(*layer_params))         ## creating a transpose of the link matrix

            backproped_error = [None] * len(vars_in_next_layer_back)
            for k,varr in enumerate(vars_in_next_layer_back):
                for j,var2 in enumerate(vars_in_layer):
                    backproped_error[k] = sum([self.vals_for_learnable_params[transposed_layer_params[k][i]] *
                                               pred_err_backproped_at_layers[back_layer_index][i]
                                               for i in range(len(vars_in_layer))])
#                                               deriv_sigmoid_avg[i] for i in range(len(vars_in_layer))])
            pred_err_backproped_at_layers[back_layer_index - 1]  =  backproped_error
            input_vars_to_layer = self.layer_vars[back_layer_index-1]
            for j,var in enumerate(vars_in_layer):
                layer_params = self.layer_params[back_layer_index][j]
                ##  Regarding the parameter update loop that follows, see the Slides 74 through 77 of my Week 3
                ##  lecture slides for how the parameters are updated using the partial derivatives stored away
                ##  during forward propagation of data. The theory underlying these calculations is presented
                ##  in Slides 68 through 71.
                for i,param in enumerate(layer_params):
                    gradient_of_loss_for_param = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j]

                    #My change start
                    self.step[back_layer_index-1][i] = (self.mu*self.step[back_layer_index-1][i]) +  gradient_of_loss_for_param *


                    self.vals_for_learnable_params[param] += self.step[back_layer_index-1][i]*self.learning_rate


            self.bias_update[back_layer_index-1] = (self.mu*self.bias_update[back_layer_index-1]) + sum(pred_err_backproped_at_la
                                                                               * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg)
            self.bias[back_layer_index-1] += self.learning_rate * self.bias_update[back_layer_index-1]

            ##My change end
    ######################################################################################################
```

In [3]:

```
class Adam(ComputationalGraphPrimer):
    def __init__(self,*args,**kwargs):
        super().__init__(*args,**kwargs)

    def run_training_loop_multi_neuron_model(self, training_data,beta1,beta2):

        class DataLoader:
            """
            To understand the logic of the dataloader, it would help if you first understand how
            the training dataset is created.  Search for the following function in this file:

                             gen_training_data(self)

            As you will see in the implementation code for this method, the training dataset
            consists of a Python dict with two keys, 0 and 1, the former points to a list of
            all Class 0 samples and the latter to a list of all Class 1 samples.  In each list,
            the data samples are drawn from a multi-dimensional Gaussian distribution.  The two
            classes have different means and variances.  The dimensionality of each data sample
            is set by the number of nodes in the input layer of the neural network.

            The data loader's job is to construct a batch of samples drawn randomly from the two
            lists mentioned above.  And it mush also associate the class label with each sample
            separately.
            """
            def __init__(self, training_data, batch_size):
                self.training_data = training_data
                self.batch_size = batch_size
                self.class_0_samples = [(item, 0) for item in self.training_data[0]]    ## Associate label 0 with each sample
                self.class_1_samples = [(item, 1) for item in self.training_data[1]]    ## Associate label 1 with each sample

            def __len__(self):
                return len(self.training_data[0]) + len(self.training_data[1])

            def _getitem(self):
                cointoss = random.choice([0,1])                            ## When a batch is created by getbatch(), we want the
                                                                           ##   samples to be chosen randomly from the two lists
                if cointoss == 0:
                    return random.choice(self.class_0_samples)
                else:
                    return random.choice(self.class_1_samples)

            def getbatch(self):
                batch_data,batch_labels = [],[]                            ## First list for samples, the second for labels
                maxval = 0.0                                               ## For approximate batch data normalization
                for _ in range(self.batch_size):
                    item = self._getitem()
                    if np.max(item[0]) > maxval:
                        maxval = np.max(item[0])
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]          ## Normalize batch data
                batch = [batch_data, batch_labels]
                return batch


        """
        The training loop must first initialize the learnable parameters.  Remember, these are the
        symbolic names in your input expressions for the neural layer that do not begin with the
        letter 'x'.  In this case, we are initializing with random numbers from a uniform distribution
        over the interval (0,1).
        """
        self.vals_for_learnable_params = {param: random.uniform(0,1) for param in self.learnable_params}

        self.bias = [random.uniform(0,1) for _ in range(self.num_layers-1)]      ## Adding the bias to each layer improves
                                                                                 ##   class discrimination. We initialize it
                                                                                 ##   to a random number.

        ##My input start
        self.bias_m = [0.0]*(self.num_layers+1)
        self.bias_v = [0.0]*(self.num_layers+1)
        self.bias_mh = [0.0]*(self.num_layers+1)
        self.bias_vh = [0.0]*(self.num_layers+1)

        self.step_m = [[0]*(len(self.learnable_params)+1)]*(self.num_layers+1)
        self.step_v = [[0]*(len(self.learnable_params)+1)]*(self.num_layers+1)
        self.step_mh = [[0]*(len(self.learnable_params)+1)]*(self.num_layers+1)
        self.step_vh = [[0]*(len(self.learnable_params)+1)]*(self.num_layers+1)

        self.beta1 = beta1
        self.beta2 = beta2
        self.m = 0

        ##My input end
```

```
        data_loader = DataLoader(training_data, batch_size=self.batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0                                        ##  Average the loss over iterations for printing
                                                                               ##    every N iterations during the training loo
        for i in range(self.training_iterations):
            self.m = i+1
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            self.forward_prop_multi_neuron_model(data_tuples)                             ## FORW PROP works by side-effect
            predicted_labels_for_batch = self.forw_prop_vals_at_layers[self.num_layers-1]     ## Predictions from FORW PROP
            y_preds =  [item for sublist in  predicted_labels_for_batch  for item in sublist]  ## Get numeric vals for prediction
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(class_labels))])  ## Calculate loss for batch
            loss_avg = loss / float(len(class_labels))                                    ## Average the loss over batch
            avg_loss_over_iterations += loss_avg                                         ## Add to Average loss over iterati
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))          ## Display avg loss
                avg_loss_over_iterations = 0.0                                           ## Re-initialize avg-over-iteration
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            self.backprop_and_update_params_multi_neuron_model(y_error_avg, class_labels)      ## BACKPROP loss
        return loss_running_record


    def forward_prop_multi_neuron_model(self, data_tuples_in_batch):
        """
        During forward propagation, we push each batch of the input data through the
        network.  In order to explain the logic of forward, consider the following network
        layout in 4 nodes in the input layer, 2 nodes in the hidden layer, and 1 node in
        the output layer.

                               input

                                 x                                          x = node

                                 x         x|                               | = sigmoid activation
                                                     x|
                                 x         x|

                                 x

                             layer_0     layer_1     layer_2


        In the code shown below, the expressions to evaluate for computing the
        pre-activation values at a node are stored at the layer in which the nodes reside.
        That is, the dictionary look-up "self.layer_exp_objects[layer_index]" returns the
        Expression objects for which the left-side dependent variable is in the layer
        pointed to layer_index.  So the example shown above, "self.layer_exp_objects[1]"
        will return two Expression objects, one for each of the two nodes in the second
        layer of the network (that is, layer indexed 1).

        The pre-activation values obtained by evaluating the expressions at each node are
        then subject to Sigmoid activation, followed by the calculation of the partial
        derivative of the output of the Sigmoid function with respect to its input.

        In the forward, the values calculated for the nodes in each layer are stored in
        the dictionary

                        self.forw_prop_vals_at_layers[ layer_index ]

        and the gradients values calculated at the same nodes in the dictionary:

                        self.gradient_vals_for_layers[ layer_index ]

        """
        self.forw_prop_vals_at_layers = {i : [] for i in range(self.num_layers)}
        self.gradient_vals_for_layers = {i : [] for i in range(1, self.num_layers)}
        for vals_for_input_vars in data_tuples_in_batch:
            self.forw_prop_vals_at_layers[0].append(vals_for_input_vars)
            for layer_index in range(1, self.num_layers):
                input_vars = self.layer_vars[layer_index-1]
                if layer_index == 1:
                    vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_vars)))
                output_vals_arr = []
                gradients_val_arr = []
                ##  In the following loop for forward propagation calculations, exp_obj is the Exp object
                ##  that is created for each user-supplied expression that specifies the network.  See the
                ##  definition of the class Exp (for 'Expression') by searching for "class Exp":
                for exp_obj in self.layer_exp_objects[layer_index]:
                    output_val = self.eval_expression(exp_obj.body , vals_for_input_vars_dict,
```

```
                                                                self.vals_for_learnable_params, input_vars)
                    ## [Search for "self.bias" in this file.]  As mentioned earlier, adding bias to each
                    ##  layer improves class discrimination:
                    output_val = output_val + self.bias[layer_index-1]
                    ## apply sigmoid activation:
                    output_val = 1.0 / (1.0 + np.exp(-1.0 * output_val))
                    output_vals_arr.append(output_val)
                    ## calculate partial of the activation function as a function of its input
                    deriv_sigmoid = output_val * (1.0 - output_val)
                    gradients_val_arr.append(deriv_sigmoid)
                    vals_for_input_vars_dict[ exp_obj.dependent_var ] = output_val
                self.forw_prop_vals_at_layers[layer_index].append(output_vals_arr)
                ##  See the bullets in red on Slides 70 and 73 of my Week 3 slides for what needs
                ##  to be stored during the forward propagation of data through the network:
                self.gradient_vals_for_layers[layer_index].append(gradients_val_arr)


    def backprop_and_update_params_multi_neuron_model(self, y_error, class_labels):
        """
        First note that loop index variable 'back_layer_index' starts with the index of
        the last layer.  For the 3-layer example shown for 'forward', back_layer_index
        starts with a value of 2, its next value is 1, and that's it.

        Stochastic Gradient Gradient calls for the backpropagated loss to be averaged over
        the samples in a batch.  To explain how this averaging is carried out by the
        backprop function, consider the last node on the example shown in the forward()
        function above.  Standing at the node, we look at the 'input' values stored in the
        variable "input_vals".  Assuming a batch size of 8, this will be list of
        lists. Each of the inner lists will have two values for the two nodes in the
        hidden layer. And there will be 8 of these for the 8 elements of the batch.  We average
        these values 'input vals' and store those in the variable "input_vals_avg".  Next we
        must carry out the same batch-based averaging for the partial derivatives stored in the
        variable "deriv_sigmoid".

        Pay attention to the variable 'vars_in_layer'.  These store the node variables in
        the current layer during backpropagation.  Since back_layer_index starts with a
        value of 2, the variable 'vars_in_layer' will have just the single node for the
        example shown for forward(). With respect to what is stored in vars_in_layer', the
        variables stored in 'input_vars_to_layer' correspond to the input layer with
        respect to the current layer.
        """
        # backproped prediction error:
        pred_err_backproped_at_layers = {i : [] for i in range(1,self.num_layers-1)}
        pred_err_backproped_at_layers[self.num_layers-1] = [y_error]
        for back_layer_index in reversed(range(1,self.num_layers)):
            input_vals = self.forw_prop_vals_at_layers[back_layer_index -1]
            input_vals_avg = [sum(x) for x in zip(*input_vals)]
            input_vals_avg = list(map(operator.truediv, input_vals_avg, [float(len(class_labels))] * len(class_labels)))
            deriv_sigmoid =  self.gradient_vals_for_layers[back_layer_index]
            deriv_sigmoid_avg = [sum(x) for x in zip(*deriv_sigmoid)]
            deriv_sigmoid_avg = list(map(operator.truediv, deriv_sigmoid_avg,
                                                       [float(len(class_labels))] * len(class_labels)))
            vars_in_layer  =  self.layer_vars[back_layer_index]                 ## a list like ['xo']
            vars_in_next_layer_back  =  self.layer_vars[back_layer_index - 1]   ## a list like ['xw', 'xz']

            layer_params = self.layer_params[back_layer_index]
            ## note that layer_params are stored in a dict like
                ##     {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']], 2: [['cp', 'cq']]}
            ## "layer_params[idx]" is a list of lists for the link weights in layer whose output nodes are in layer "idx"
            transposed_layer_params = list(zip(*layer_params))         ## creating a transpose of the link matrix

            backproped_error = [None] * len(vars_in_next_layer_back)
            for k,varr in enumerate(vars_in_next_layer_back):
                for j,var2 in enumerate(vars_in_layer):
                    backproped_error[k] = sum([self.vals_for_learnable_params[transposed_layer_params[k][i]] *
                                               pred_err_backproped_at_layers[back_layer_index][i]
                                               for i in range(len(vars_in_layer))])
#                                               deriv_sigmoid_avg[i] for i in range(len(vars_in_layer))])
            pred_err_backproped_at_layers[back_layer_index - 1]  =  backproped_error
            input_vars_to_layer = self.layer_vars[back_layer_index-1]
            for j,var in enumerate(vars_in_layer):
                layer_params = self.layer_params[back_layer_index][j]
                ##  Regarding the parameter update loop that follows, see the Slides 74 through 77 of my Week 3
                ##  lecture slides for how the parameters are updated using the partial derivatives stored away
                ##  during forward propagation of data. The theory underlying these calculations is presented
                ##  in Slides 68 through 71.
                for i,param in enumerate(layer_params):
                    gradient_of_loss_for_param = input_vals_avg[i] * pred_err_backproped_at_layers[back_layer_index][j]

                    #My change start
                    self.step_m[back_layer_index-1][i] = (self.beta1*self.step_m[back_layer_index-1][i]) + (1-self.beta1)*(gradie
                    self.step_mh[back_layer_index-1][i] = self.step_m[back_layer_index-1][i]/(1-(self.beta1**self.m))
                    self.step_v[back_layer_index-1][i] = (self.beta2*self.step_v[back_layer_index-1][i]) + (1-self.beta2)*((gradi
                    self.step_vh[back_layer_index-1][i] = self.step_v[back_layer_index-1][i]/(1-(self.beta2**self.m))
                    self.vals_for_learnable_params[param] += self.learning_rate * (self.step_mh[back_layer_index-1][i]/(np.sqrt(s
```

```
            self.bias_m[back_layer_index-1] = (self.beta1*self.bias_m[back_layer_index-1]) + (1-self.beta1)*(sum(pred_err_backpro
                                                                        * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg))
            self.bias_mh[back_layer_index-1] = self.bias_m[back_layer_index-1]/(1-(self.beta1**self.m))
            self.bias_v[back_layer_index-1] = (self.beta2*self.bias_v[back_layer_index-1]) + (1-self.beta2)*((sum(pred_err_backpr
                                                                        * sum(deriv_sigmoid_avg)/len(deriv_sigmoid_avg))**2)
            self.bias_vh[back_layer_index-1] = self.bias_v[back_layer_index-1]/(1-(self.beta2**self.m))
            self.bias[back_layer_index-1] += self.learning_rate * (self.bias_mh[back_layer_index-1]/(np.sqrt(self.bias_vh[back_la
            ##My change end
            ##################################################################################################
```

In [4]:

```
cgp1 = SGDPlus(
               num_layers = 3,
               layers_config = [4,2,1],                         # num of nodes in each layer
               expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                              'xz=bp*xp+bq*xq+br*xr+bs*xs',
                              'xo=cp*xw+cq*xz'],
               output_vars = ['xo'],
               dataset_size = 5000,
               learning_rate = 1e-2,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )

cgp1.parse_multi_layer_expressions()
training_data1 = cgp1.gen_training_data()
```

```
self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs*xs'], 2: ['xo=cp*xw+cq*x
z']}


[layer index: 1] all variables: {'xw', 'xz', 'xr', 'xp', 'xq', 'xs'}


[layer index: 1] learnable params: {'bp', 'ar', 'bs', 'ap', 'as', 'br', 'bq', 'aq'}


[layer index: 1] dependencies: {'xw': ['xp', 'xq', 'xr', 'xs'], 'xz': ['xp', 'xq', 'xr', 'xs']}


[layer index: 1] expressions dict: {'xw': 'ap*xp+aq*xq+ar*xr+as*xs', 'xz': 'bp*xp+bq*xq+br*xr+bs*xs'}


[layer index: 1] var_to_var_param dict: {'xw': {'xp': 'ap', 'xq': 'aq', 'xr': 'ar', 'xs': 'as'}, 'xz': {'xp':
```

In [5]:

```
cgp2 = Adam(
               num_layers = 3,
               layers_config = [4,2,1],                    # num of nodes in each layer
               expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
                              'xz=bp*xp+bq*xq+br*xr+bs*xs',
                              'xo=cp*xw+cq*xz'],
               output_vars = ['xo'],
               dataset_size = 5000,
               learning_rate = 1e-2,
#               learning_rate = 5 * 1e-2,
               training_iterations = 40000,
               batch_size = 8,
               display_loss_how_often = 100,
               debug = True,
      )

cgp2.parse_multi_layer_expressions()
training_data2 = cgp2.gen_training_data()
```

```
self.layer_expressions:  {1: ['xw=ap*xp+aq*xq+ar*xr+as*xs', 'xz=bp*xp+bq*xq+br*xr+bs*xs'], 2: ['xo=cp*xw+cq*x
z']}


[layer index: 1] all variables: {'xw', 'xz', 'xr', 'xp', 'xq', 'xs'}


[layer index: 1] learnable params: {'bp', 'ar', 'bs', 'ap', 'as', 'br', 'bq', 'aq'}


[layer index: 1] dependencies: {'xw': ['xp', 'xq', 'xr', 'xs'], 'xz': ['xp', 'xq', 'xr', 'xs']}


[layer index: 1] expressions dict: {'xw': 'ap*xp+aq*xq+ar*xr+as*xs', 'xz': 'bp*xp+bq*xq+br*xr+bs*xs'}


[layer index: 1] var_to_var_param dict: {'xw': {'xp': 'ap', 'xq': 'aq', 'xr': 'ar', 'xs': 'as'}, 'xz': {'xp':
```

In [6]:

```
loss3 = cgp2.run_training_loop_multi_neuron_model(training_data2,0.9,0.99)
plt.plot(loss3,label = "Adam loss")


loss1 = cgp1.run_training_loop_multi_neuron_model(training_data1)
plt.plot(loss1,label = "SGD loss")

loss2 = cgp1.run_training_loop_multi_neuron_model(training_data1,0.9,True)
plt.plot(loss2,label = "SGD+ loss")



plt.xlabel("Iterations in hundreds")
plt.ylabel("Loss")
plt.title("Multi neuron loss using different optimizers")

plt.legend(loc = "upper right")
```

```
[iter=1]  loss = 0.0023
[iter=101]  loss = 0.2523
[iter=201]  loss = 0.2470
[iter=301]  loss = 0.2456
[iter=401]  loss = 0.2453
[iter=501]  loss = 0.2448
[iter=601]  loss = 0.2436
[iter=701]  loss = 0.2433
[iter=801]  loss = 0.2438
[iter=901]  loss = 0.2430
[iter=1001]  loss = 0.2412
[iter=1101]  loss = 0.2427
[iter=1201]  loss = 0.2412
[iter=1301]  loss = 0.2404
[iter=1401]  loss = 0.2397
[iter=1501]  loss = 0.2404
[iter=1601]  loss = 0.2395
[iter=1701]  loss = 0.2384
[iter=1801]  loss = 0.2372
```

In [ ]:

```

```