ECE 601: Homework 3

Wei Xu
Email: xu1639@purdue.edu
Due date: 11:59 PM, Jan. 29, 2024
(Spring 2024)

1 Becoming Familiar with the Primer

1.1 Comparing the multi-neuron case with the handcrafted network and the `torch.nn` based network

Note: The code used for this subsection is borrowed from the `ComputationalGraphPrimer` package (https://engineering.purdue.edu/kak/distCGP/).

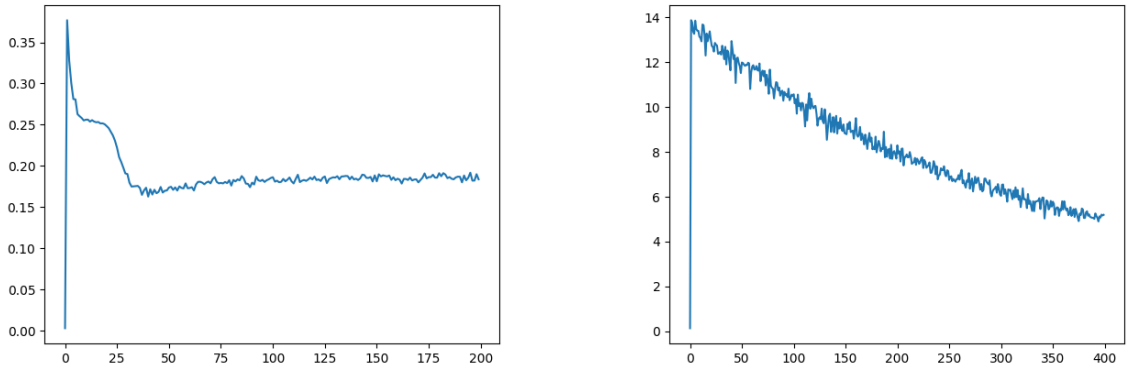
The last ten lines of the loss output from the handcrafted network are shown below. And the loss curve is shown in Fig. 1(a).

```
...
[iter=19001]  loss = 0.1868
[iter=19101]  loss = 0.1800
[iter=19201]  loss = 0.1883
[iter=19301]  loss = 0.1826
[iter=19401]  loss = 0.1855
[iter=19501]  loss = 0.1917
[iter=19601]  loss = 0.1822
[iter=19701]  loss = 0.1822
[iter=19801]  loss = 0.1899
[iter=19901]  loss = 0.1838
```

The last ten lines of the loss output from the `torch.nn` network are shown below. And the loss curve is shown in Fig. 1(b).

```
...
[iter=39001]  loss = 5.0162
[iter=39101]  loss = 5.2571
[iter=39201]  loss = 5.1446
[iter=39301]  loss = 5.0601
[iter=39401]  loss = 4.9013
[iter=39501]  loss = 5.1261
[iter=39601]  loss = 5.0779
[iter=39701]  loss = 5.2043
[iter=39801]  loss = 5.1673
[iter=39901]  loss = 5.1952
```

It is noticed from Fig. 1, the `torch.nn` network converges more stable and the loss decreases all the way, but the handcrafted network shows some unstable behaviors, like the loss increasing during the late training.



(a) handcrafted network

(b) `torch.nn` network

Figure 1: Loss curves for multi-neuron case.

1.2 Comparing the one-neuron case with the handcrafted network and the `torch.nn` based network

Note: The code used for this subsection is borrowed from the `ComputationalGraphPrimer` package (https://engineering.purdue.edu/kak/distCGP/).

The last ten lines of the loss output from the handcrafted network are shown below. And the loss curve is shown in Fig. 2(a).

```
...
[iter=39001]  loss = 0.1884
[iter=39101]  loss = 0.1906
[iter=39201]  loss = 0.1924
[iter=39301]  loss = 0.1929
[iter=39401]  loss = 0.1960
[iter=39501]  loss = 0.1883
[iter=39601]  loss = 0.1885
[iter=39701]  loss = 0.1932
[iter=39801]  loss = 0.1877
[iter=39901]  loss = 0.1906
```

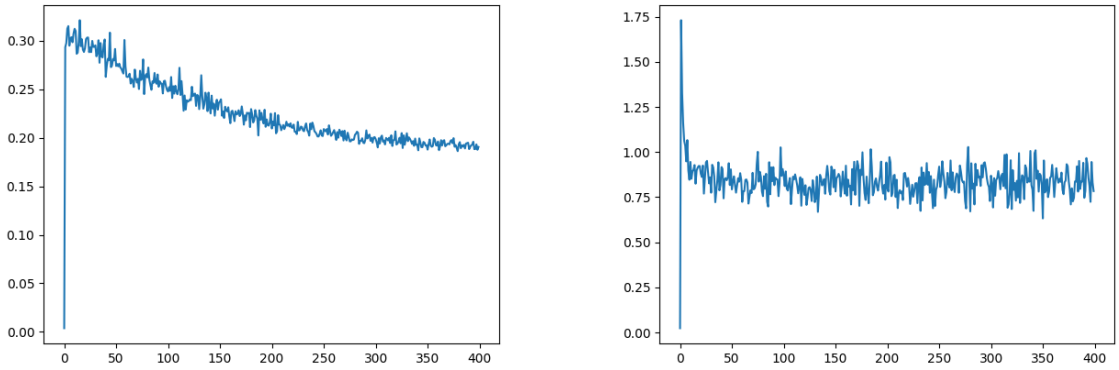
To test the `torch.nn` network, some modifications are needed in `verify_with_torchnn.py` to run for this one-neuron case. The modifications and the last ten lines of the loss output are shown below. And the loss curve is shown in Fig. 1(b).

```
...
cgp = ComputationalGraphPrimer(
                ...
                # learning_rate = 1e-6, # For the multi-neuron option
                    ↪ below
                ...
                learning_rate = 5 * 1e-2, # Also for the one-neuron
                    ↪ option below
                ...
      )
...
```

```
cgp.run_training_with_torchnn('one_neuron', training_data) ## (A)
    ↪ REMEMBER to also change learning_rate above
# cgp.run_training_with_torchnn('multi_neuron', training_data) ## (B)
    ↪ REMEMBER to also change learning_rate above
```

```
...
[iter=39001]  loss = 0.7465
[iter=39101]  loss = 0.7844
[iter=39201]  loss = 0.9668
[iter=39301]  loss = 0.9348
[iter=39401]  loss = 0.8580
[iter=39501]  loss = 0.8124
[iter=39601]  loss = 0.7243
[iter=39701]  loss = 0.9452
[iter=39801]  loss = 0.8305
[iter=39901]  loss = 0.7839
```

It is noticed from Fig. 2, the `torch.nn` network converges much faster, but the handcrafted network converges slower.



(a) handcrafted network

(b) `torch.nn` network

Figure 2: Loss curves for one-neuron case.

1.3 Understanding of SGD+ and Adam

SGD with momentum (SGD+). According to Slides 108 and 109 in Week 3, using the Vanilla Gradient Descent is prone to oscillations in the convergence curve when encountering "valleys". One solution to this problem is to apply momentum, namely SGD with momentum (SGD+). Specifically, the idea is to add momentum to the gradient descent process, making the updating faster in the dimension in which the gradient is unchanged and the updating slower in the dimension in which the gradient has changed. This speeds up convergence and reduces oscillations. The update process is

$$v_t = \mu * v_{t-1} + g_t \tag{1}$$

$$p_t = p_{t-1} - \text{lr} * v_t \tag{2}$$

where v_t is the step size at t (typically $v_0 = 0$), g_t is the gradient at t, p_t is the model weights at t, lr is the learning rate, and $\mu \in [0, 1]$ is the momentum factor. μ controls how much the previous step

size contributes to the current step size. A larger μ means that the previous step size is more heavily weighted in the current step size calculation. The current direction of descent is determined by the combination of the direction of descent accumulated from the history and the current gradient. And the closer to the current moment, the more influence it exerts. The advantages of SGD+ are the ability to pass local minima, the ability to speed up convergence, as well as the ability to suppress oscillations during gradient descent. When checking the PyTorch function of SGD (https://pytorch.org/docs/stable/generated/torch.optim.SGD.html), this parameter is set as 0 by default. But I think a value of 0.5 is good to start from.

Adaptive Moment Estimation (Adam). SGD+ considers the first-order momentum, to which Adam adds the second-order momentum. Slides 117 and 118 in Week 3 provides good explanation to this optimizer. Firstly, calculate the biased first moment estimate m_t with

$$m_t = \beta_1 * m_{t-1} + (1 - \beta_1) * g_t \tag{3}$$

where β_1 is the first exponential decay rate, controlling computing running averages of the gradient. Then, calculate the biased second raw moment estimate v_t with

$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2) * (g_t)^2 \tag{4}$$

where β_2 is the second exponential decay rate, controlling computing running averages of the squared gradient. Notice that the initial values $m_0 = 0$ and $v_0 = 0$, then the following values m_t and v_t ($t \leq 1$) are biased toward 0, especially during the beginning training phase. So these two values are corrected by

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{5}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{6}$$

Finally, the model weights are updated by

$$p_t = p_{t-1} - \text{lr} * \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}} \tag{7}$$

where ϵ is incorporated to avoid the denominator being 0, which is set as 10^{-8} in PyTorch by default. With the second moment item, Adam is more robust and can help converge faster compared with SGD+.

2 Programming Task

2.1 One-neuron classifier

Note: The code for this subsection is borrowed from the `ComputationalGraphPrimer` package (https://engineering.purdue.edu/kak/distCGP/).

The code for this task is shown below. To make the code concise, I only put the comments in the lines that I modified. The `OneNeuronSGDCGP` class is for Vanilla Gradient Descent. The `run_training_loop_one_neuron_model` method is rewritten to return the loss record. The `OneNeuronSGDPlusCGP` class is for SGD+. A new parameter, momentum factor, is added in the `__init__` method. The `run_training_loop_one_neuron_model` method is rewritten to return the loss record. The `backprop_and_update_params_one_neuron_model` method is rewritten to apply

the SGD+ weight update equations. The `OneNeuronAdamCGP` class is for Adam. Three new parameters, first exponential decay rate, second exponential decay rate, and epsilon, are added in the `__init__` method. The `run_training_loop_one_neuron_model` method is rewritten to return the loss record. The `backprop_and_update_params_one_neuron_model` method is rewritten to apply the Adam weight update equations.

```
...
class OneNeuronSGDCGP(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs) ## inherit from CGP
    def run_training_loop_one_neuron_model(self, training_data):
        self.vals_for_learnable_params = {param: random.uniform(0,1)
            ↪ for param in self.learnable_params}
        self.bias = random.uniform(0,1)
        class DataLoader:
            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]]
                self.class_1_samples = [(item, 1) for item in self.
                    ↪ training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.
                    ↪ training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
                maxval = 0.0
                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]
                batch = [batch_data, batch_labels]
                return batch
        data_loader = DataLoader(training_data, batch_size=self.
            ↪ batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples_in_batch = data[0]
            class_labels_in_batch = data[1]
            y_preds, deriv_sigmoids =  self.
                ↪ forward_prop_one_neuron_model(data_tuples_in_batch)
```

```
            loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
                ↪  for i in range(len(class_labels_in_batch))])
            avg_loss_over_iterations += loss / float(len(
                ↪ class_labels_in_batch))
            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))
                avg_loss_over_iterations = 0.0
            y_errors_in_batch = list(map(operator.sub,
                ↪ class_labels_in_batch, y_preds))
            self.backprop_and_update_params_one_neuron_model(
                ↪ data_tuples_in_batch, y_preds, y_errors_in_batch,
                ↪ deriv_sigmoids)
        ## removed plt functions
        return loss_running_record ## return the loss record

class OneNeuronSGDPlusCGP(ComputationalGraphPrimer):
    def __init__(self, mu, *args, **kwargs):
        super().__init__(*args, **kwargs) ## inherit from CGP
        self.mu = mu ## momentum factor
    def run_training_loop_one_neuron_model(self, training_data):
        self.vals_for_learnable_params = {param: random.uniform(0,1)
            ↪ for param in self.learnable_params}
        self.bias = random.uniform(0,1)
        self.previous_v_vals = {param: 0 for param in self.
            ↪ learnable_params} ## initialize v_0 for weights
        self.previous_v_bias = 0 ## initialize v_0 for bias
        class DataLoader:
            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]]
                self.class_1_samples = [(item, 1) for item in self.
                    ↪ training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.
                    ↪ training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
                maxval = 0.0
                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]
            batch = [batch_data, batch_labels]
            return batch
    data_loader = DataLoader(training_data, batch_size=self.
        ↪ batch_size)
    loss_running_record = []
    i = 0
    avg_loss_over_iterations = 0.0
    for i in range(self.training_iterations):
        data = data_loader.getbatch()
        data_tuples_in_batch = data[0]
        class_labels_in_batch = data[1]
        y_preds, deriv_sigmoids =  self.
            ↪ forward_prop_one_neuron_model(data_tuples_in_batch)
        loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
            ↪  for i in range(len(class_labels_in_batch))])
        avg_loss_over_iterations += loss / float(len(
            ↪ class_labels_in_batch))
        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))
            avg_loss_over_iterations = 0.0
        y_errors_in_batch = list(map(operator.sub,
            ↪ class_labels_in_batch, y_preds))
        self.backprop_and_update_params_one_neuron_model(
            ↪ data_tuples_in_batch, y_preds, y_errors_in_batch,
            ↪ deriv_sigmoids)
    ## removed plt functions
    return loss_running_record ## return the loss record
def backprop_and_update_params_one_neuron_model(self,
    ↪ data_tuples_in_batch, predictions, y_errors_in_batch,
    ↪ deriv_sigmoids):
    input_vars = self.independent_vars
    input_vars_to_param_map = self.var_to_var_param[self.
        ↪ output_vars[0]]
    param_to_vars_map = {param : var for var, param in
        ↪ input_vars_to_param_map.items()}
    vals_for_learnable_params = self.vals_for_learnable_params
    for i,param in enumerate(self.vals_for_learnable_params):
        partial_of_loss_wrt_param = 0.0
        for j in range(self.batch_size):
            vals_for_input_vars_dict = dict(zip(input_vars, list(
                ↪ data_tuples_in_batch[j])))
            partial_of_loss_wrt_param += - y_errors_in_batch[j] *
                ↪ vals_for_input_vars_dict[param_to_vars_map[param
                ↪ ]] * deriv_sigmoids[j]
        partial_of_loss_wrt_param /= float(self.batch_size) ##
            ↪ average after the batch finishing
        self.previous_v_vals[param] = self.mu * self.
            ↪ previous_v_vals[param] + partial_of_loss_wrt_param ##
            ↪  update momentum memory
```

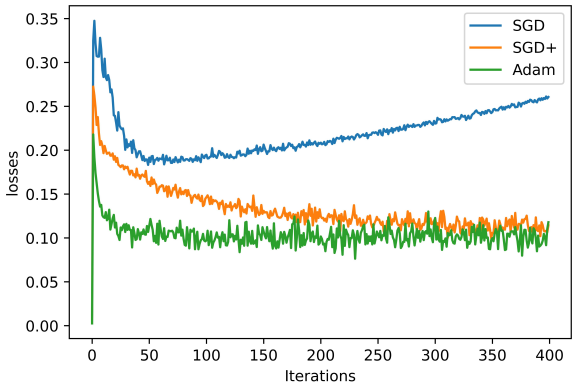
```
            step = self.learning_rate * self.previous_v_vals[param] ##
               ↪ scaled step size
            self.vals_for_learnable_params[param] -= step
        partial_of_loss = 0 ## initialize loss count
        for j in range(self.batch_size): ## loop for a batch
            partial_of_loss -= y_errors_in_batch[j] * deriv_sigmoids[j]
        partial_of_loss /= float(self.batch_size) ## average after the
           ↪ batch finishing
        self.previous_v_bias = self.mu * self.previous_v_bias +
           ↪ partial_of_loss ## update momentum memory
        self.bias -= self.learning_rate * self.previous_v_bias ##
           ↪ Update the bias

class OneNeuronAdamCGP(ComputationalGraphPrimer):
    def __init__(self, beta_1, beta_2, epsilon, *args, **kwargs):
        super().__init__(*args, **kwargs) ## inherit from CGP
        self.beta_1 = beta_1 ## first exponential decay rate
        self.beta_2 = beta_2 ## second exponential decay rate
        self.epsilon = epsilon ## epsilon
    def run_training_loop_one_neuron_model(self, training_data):
        self.vals_for_learnable_params = {param: random.uniform(0,1)
           ↪ for param in self.learnable_params}
        self.bias = random.uniform(0,1)
        self.previous_m_vals = {param: 0 for param in self.
           ↪ learnable_params} ## initialize m_0 for weights
        self.previous_m_bias = 0 ## initialize m_0 for bias
        self.previous_v_vals = {param: 0 for param in self.
           ↪ learnable_params} ## initialize m_0 for weights
        self.previous_v_bias = 0 ## initialize m_0 for bias
        class DataLoader:
            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]]
                self.class_1_samples = [(item, 1) for item in self.
                   ↪ training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.
                   ↪ training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
                maxval = 0.0
                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]
                batch = [batch_data, batch_labels]
                return batch
        data_loader = DataLoader(training_data, batch_size=self.
            ↪ batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples_in_batch = data[0]
            class_labels_in_batch = data[1]
            y_preds, deriv_sigmoids =  self.
                ↪ forward_prop_one_neuron_model(data_tuples_in_batch)
            loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
                ↪  for i in range(len(class_labels_in_batch))])
            avg_loss_over_iterations += loss / float(len(
                ↪ class_labels_in_batch))
            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))
                avg_loss_over_iterations = 0.0
            y_errors_in_batch = list(map(operator.sub,
                ↪ class_labels_in_batch, y_preds))
            self.backprop_and_update_params_one_neuron_model(
                ↪ data_tuples_in_batch, y_preds, y_errors_in_batch,
                ↪ deriv_sigmoids, i+1)
        ## removed plt functions
        return loss_running_record ## return the loss record
    def backprop_and_update_params_one_neuron_model(self,
        ↪ data_tuples_in_batch, predictions, y_errors_in_batch,
        ↪ deriv_sigmoids, iter): ## add iteration index as aparameter
        input_vars = self.independent_vars
        input_vars_to_param_map = self.var_to_var_param[self.
            ↪ output_vars[0]]
        param_to_vars_map = {param : var for var, param in
            ↪ input_vars_to_param_map.items()}
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            partial_of_loss_wrt_param = 0.0
            for j in range(self.batch_size):
                vals_for_input_vars_dict = dict(zip(input_vars, list(
                    ↪ data_tuples_in_batch[j])))
                partial_of_loss_wrt_param += - y_errors_in_batch[j] *
                    ↪ vals_for_input_vars_dict[param_to_vars_map[param
                    ↪ ]] * deriv_sigmoids[j]
            partial_of_loss_wrt_param /= float(self.batch_size) ##
                ↪ average after the batch finishing
            self.previous_m_vals[param] = self.beta_1 * self.
                ↪ previous_m_vals[param] + (1 - self.beta_1) *
                ↪ partial_of_loss_wrt_param ## update first momentum
```

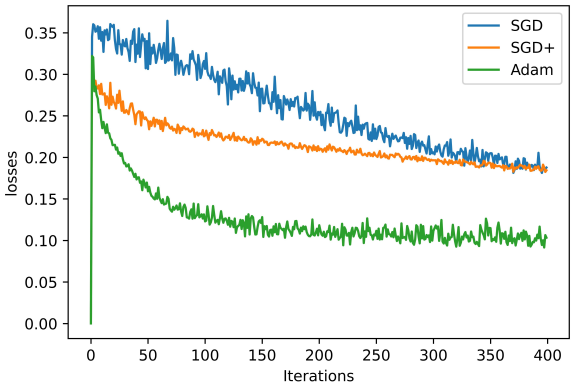
```
                ↪ memory
            self.previous_v_vals[param] = self.beta_2 * self.
                ↪ previous_v_vals[param] + (1 - self.beta_2) *
                ↪ partial_of_loss_wrt_param**2 ## update second
                ↪ momentum memory
            m_t_hat = self.previous_m_vals[param] / (1 - self.beta_1**
                ↪ iter) ## scaled first momentum memory
            v_t_hat = self.previous_v_vals[param] / (1 - self.beta_2**
                ↪ iter) ## scaled second momentum memory
            step = self.learning_rate * m_t_hat / np.sqrt(v_t_hat +
                ↪ self.epsilon) ## scaled step size
            self.vals_for_learnable_params[param] -= step
        partial_of_loss = 0 ## initialize loss count
        for j in range(self.batch_size): ## loop for a batch
            partial_of_loss -= y_errors_in_batch[j] * deriv_sigmoids[j]
        partial_of_loss /= float(self.batch_size) ## average after the
            ↪ batch finishing
        self.previous_m_bias = self.beta_1 * self.previous_m_bias + (1
            ↪ - self.beta_1) * partial_of_loss ## update first momentum
            ↪  memory
        self.previous_v_bias = self.beta_2 * self.previous_v_bias + (1
            ↪ - self.beta_2) * partial_of_loss**2 ## update second
            ↪ momentum memory
        m_t_hat = self.previous_m_bias / (1 - self.beta_1**iter) ##
            ↪ scaled first momentum memory
        v_t_hat = self.previous_v_bias / (1 - self.beta_2**iter) ##
            ↪ scaled second momentum memory
        self.bias -= self.learning_rate * m_t_hat / np.sqrt(v_t_hat +
            ↪ self.epsilon) ## Update the bias
...
## plot the learning curves
def plot_learning_curve(loss_SGD, loss_SGDPlus, loss_Adam, title, lr):
    plt.figure(figsize=(6, 4))
    plt.plot(loss_SGD, label='SGD')
    plt.plot(loss_SGDPlus, label='SGD+')
    plt.plot(loss_Adam, label='Adam')
    plt.xlabel('Iterations')
    plt.ylabel('losses')
    plt.legend()
    cf = plt.gcf()
    cf.savefig('%s_%s.jpg'%(title,lr), bbox_inches='tight', \
                format='jpg', dpi=1000)

## run one-neuron case with different optimizer
def run_one_neuron_diff_optimizer(lr):
    cgp_SGD = OneNeuronSGDCGP(
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
```

```
                debug = True,
    )
    cgp_SGD.parse_expressions()
    cgp_SGD.display_one_neuron_network()

    cgp_SGDPlus = OneNeuronSGDPlusCGP(
                mu = 0.5,
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    cgp_SGDPlus.parse_expressions()
    cgp_SGDPlus.display_one_neuron_network()

    cgp_Adam = OneNeuronAdamCGP(
                beta_1 = 0.9,
                beta_2 = 0.99,
                epsilon = 1e-8,
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    cgp_Adam.parse_expressions()
    cgp_Adam.display_one_neuron_network()

    training_data = cgp_SGD.gen_training_data()
    loss_SGD = cgp_SGD.run_training_loop_one_neuron_model(training_data
       ↪ )
    loss_SGDPlus = cgp_SGDPlus.run_training_loop_one_neuron_model(
       ↪ training_data)
    loss_Adam = cgp_Adam.run_training_loop_one_neuron_model(
       ↪ training_data)

    plot_learning_curve(loss_SGD, loss_SGDPlus, loss_Adam, 'one_neuron'
       ↪ , str(lr))
...
if __name__ == '__main__':
...
    if args.task == '1':
        for lr in [1e-2, 1e-3, 1e-4]:
            run_one_neuron_diff_optimizer(lr)
...
```

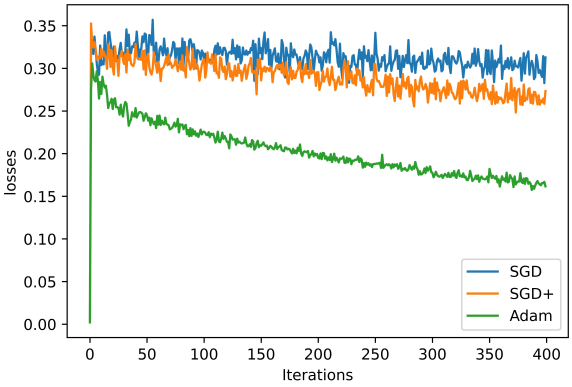
Fig. 3 shows the loss curves with the learning rates of 0.01, 0.001, and 0.0001 respectively. Generally speaking, Adam performs the best, and SGD does the worst. When the learning rate is large (as in Fig. 3(a)), Adam converges fastest but its loss curve is not very smooth, which is because it achieves the global minimum point and just oscillates around that point; SGD+ converges slower (but finally it achieves a similar level with Adam) and has more smooth loss curve; SGD diverges eventually, so the smoothness of its loss curve is meaningless. When the learning rate is moderate (as in Fig. 3(b)), Adam still converges fastest and gets the global minimum point; SGD+ and SGD converge slower and do not completely achieve the global minimum point eventually; SGD has the most oscillating loss curve in this case. When the learning rate is small (as in Fig. 3(c)), both Adam and SGD+ converge, but Adam is still faster and has the most smooth loss curve, and SGD even does not converge. Compared with Fig. 2(b), it can be found that `torch.nn` beats these three self-implemented optimizers in terms of the converging speed. The reason is that `PyTorch` is a very well-optimized framework, meaning there may be better optimization mechanisms in backpropagation, such as gradient clipping. In addition, this is almost the simplest neural network, the advantages of Adam and SGD+ can not be performed well.



(a) lr = 0.01



(b) lr = 0.001



(c) lr = 0.0001

Figure 3: Loss curves for one-neuron case with different learning rates.

2.2 Multi-neuron classifier

Note: The code for this subsection is borrowed from the `ComputationalGraphPrimer` package (https://engineering.purdue.edu/kak/distCGP/).

The code for this task is shown below. To make the code concise, I only put the comments in the lines that I modified. The `MultiNeuronSGDCGP` class is for Vanilla Gradient Descent. The `run_training_loop_one_neuron_model` method is rewritten to return the loss record. The `MultiNeuronSGDPlusCGP` class is for SGD+. A new parameter, momentum factor, is added in the `__init__` method. The `run_training_loop_one_neuron_model` method is rewritten to return the loss record. The `backprop_and_update_params_one_neuron_model` method is rewritten to apply the SGD+ weight update equations. The `MultiNeuronAdamCGP` class is for Adam. Three new parameters, first exponential decay rate, second exponential decay rate, and epsilon, are added in the `__init__` method. The `run_training_loop_one_neuron_model` method is rewritten to return the loss record. The `backprop_and_update_params_one_neuron_model` method is rewritten to apply the Adam weight update equations.

```
...
class MultiNeuronSGDCGP(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs) ## inherit from CGP
    def run_training_loop_multi_neuron_model(self, training_data):
        class DataLoader:
            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]]
                self.class_1_samples = [(item, 1) for item in self.
                    ↪ training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.
                    ↪ training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
                maxval = 0.0
                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]
                batch = [batch_data, batch_labels]
                return batch
        self.vals_for_learnable_params = {param: random.uniform(0,1)
            ↪ for param in self.learnable_params}
        self.bias = {i : [random.uniform(0,1) for j in range(self.
            ↪ layers_config[i])] for i in range(1, self.num_layers)}
        data_loader = DataLoader(training_data, batch_size=self.
            ↪ batch_size)
        loss_running_record = []
```

```
        i = 0
        avg_loss_over_iterations = 0.0
        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)
            predicted_labels_for_batch = self.forw_prop_vals_at_layers[
                ↪ self.num_layers-1]
            y_preds =  [item for sublist in  predicted_labels_for_batch
                ↪   for item in sublist]
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
                ↪  range(len(class_labels))])
            loss_avg = loss / float(len(class_labels))
            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))
                avg_loss_over_iterations = 0.0
            y_errors_in_batch = list(map(operator.sub, class_labels,
                ↪ y_preds))
            self.backprop_and_update_params_multi_neuron_model(y_preds,
                ↪  y_errors_in_batch)
        ## removed plt functions
        return loss_running_record ## return the loss record

class MultiNeuronSGDPlusCGP(ComputationalGraphPrimer):
    def __init__(self, mu, *args, **kwargs):
        super().__init__(*args, **kwargs) ## inherit from CGP
        self.mu = mu ## momentum factor
    def run_training_loop_multi_neuron_model(self, training_data):
        class DataLoader:
            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]]
                self.class_1_samples = [(item, 1) for item in self.
                    ↪ training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.
                    ↪ training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
                maxval = 0.0
                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]
                batch = [batch_data, batch_labels]
                return batch
        self.vals_for_learnable_params = {param: random.uniform(0,1)
            ↪ for param in self.learnable_params}
        self.bias =   {i : [random.uniform(0,1) for j in range(self.
            ↪ layers_config[i])] for i in range(1, self.num_layers)}
        self.previous_v_vals = {param: 0 for param in self.
            ↪ learnable_params} ## initialize v_0 for weights
        self.previous_v_bias = {i : [0 for j in range(self.
            ↪ layers_config[i])] for i in range(1, self.num_layers)} ##
            ↪  initialize v_0 for bias
        data_loader = DataLoader(training_data, batch_size=self.
            ↪ batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0
        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)
            predicted_labels_for_batch = self.forw_prop_vals_at_layers[
                ↪ self.num_layers-1]
            y_preds =  [item for sublist in  predicted_labels_for_batch
                ↪   for item in sublist]
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
                ↪  range(len(class_labels))])
            loss_avg = loss / float(len(class_labels))
            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))
                avg_loss_over_iterations = 0.0
            y_errors_in_batch = list(map(operator.sub, class_labels,
                ↪ y_preds))
            self.backprop_and_update_params_multi_neuron_model(y_preds,
                ↪  y_errors_in_batch)
        ## removed plt functions
        return loss_running_record ## return the loss record
    def backprop_and_update_params_multi_neuron_model(self, predictions
        ↪ , y_errors):
        pred_err_backproped_at_layers = [{i : [None for j in range(self
            ↪ .layers_config[i])] for i in range(self.num_layers)} for
            ↪ _ in range(self.batch_size)]
        partial_of_loss_wrt_params = {param : 0.0 for param in self.
            ↪ all_params}
```

```
        bias_changes =   {i : [0.0 for j in range(self.layers_config[i
            ↪ ])] for i in range(1, self.num_layers)}
        for b in range(self.batch_size):
            pred_err_backproped_at_layers[b][self.num_layers - 1] = [
                ↪ y_errors[b] ]
            for back_layer_index in reversed(range(1,self.num_layers)):
                input_vals = self.forw_prop_vals_at_layers[
                    ↪ back_layer_index -1]
                deriv_sigmoids =  self.gradient_vals_for_layers[
                    ↪ back_layer_index]
                vars_in_layer  =  self.layer_vars[back_layer_index]
                vars_in_next_layer_back  =  self.layer_vars[
                    ↪ back_layer_index - 1]
                vals_for_input_vars_dict =  dict(zip(
                    ↪ vars_in_next_layer_back, self.
                    ↪ forw_prop_vals_at_layers[back_layer_index - 1][b
                    ↪ ]))
                layer_params = self.layer_params[back_layer_index]
                transposed_layer_params = list(zip(*layer_params))
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        pred_err_backproped_at_layers[b][
                            ↪ back_layer_index - 1][k] = sum([self.
                            ↪ vals_for_learnable_params[
                            ↪ transposed_layer_params[k][i]] \
                             * pred_err_backproped_at_layers[b][
                                ↪ back_layer_index][i] for i in range(
                                ↪ len(vars_in_layer))])
                for j,var in enumerate(vars_in_layer):
                    layer_params = self.layer_params[back_layer_index][
                        ↪ j]
                    input_vars_to_param_map = self.var_to_var_param[var
                        ↪ ]
                    param_to_vars_map = {param : var for var, param in
                        ↪ input_vars_to_param_map.items()}
                    for i,param in enumerate(layer_params):
                        partial_of_loss_wrt_params[param] +=
                            ↪ pred_err_backproped_at_layers[b][
                            ↪ back_layer_index][j] * \
                             vals_for_input_vars_dict[param_to_vars_map[
                                ↪ param]] * deriv_sigmoids[b][j]
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        if back_layer_index-1 > 0:
                            bias_changes[back_layer_index-1][k] +=
                                ↪ pred_err_backproped_at_layers[b][
                                ↪ back_layer_index - 1][k] *
                                ↪ deriv_sigmoids[b][j]
        for param in partial_of_loss_wrt_params:
            partial_of_loss_wrt_param = partial_of_loss_wrt_params[
                ↪ param] /  float(self.batch_size)
            self.previous_v_vals[param] = self.mu * self.
                ↪ previous_v_vals[param] + partial_of_loss_wrt_param ##
                ↪  update momentum memory
```

```
            step = self.learning_rate * self.previous_v_vals[param] ##
               ↪ scaled step size
            self.vals_for_learnable_params[param] += step
        for layer_index in range(1,self.num_layers):
            for k in range(self.layers_config[layer_index]):
                self.previous_v_bias[layer_index][k] = self.mu * self.
                   ↪ previous_v_bias[layer_index][k] + (bias_changes[
                   ↪ layer_index][k] / float(self.batch_size)) ##
                   ↪ update momentum memory
                self.bias[layer_index][k]  +=  self.learning_rate *
                   ↪ self.previous_v_bias[layer_index][k] ## Update
                   ↪ the bias

class MultiNeuronAdamCGP(ComputationalGraphPrimer):
    def __init__(self, beta_1, beta_2, epsilon, *args, **kwargs):
        super().__init__(*args, **kwargs) ## inherit from CGP
        self.beta_1 = beta_1 ## first exponential decay rate
        self.beta_2 = beta_2 ## second exponential decay rate
        self.epsilon = epsilon ## epsilon
    def run_training_loop_multi_neuron_model(self, training_data):
        class DataLoader:
            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]]
                self.class_1_samples = [(item, 1) for item in self.
                   ↪ training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.
                   ↪ training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
                maxval = 0.0
                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]
                batch = [batch_data, batch_labels]
                return batch
        self.vals_for_learnable_params = {param: random.uniform(0,1)
           ↪ for param in self.learnable_params}
        self.bias =   {i : [random.uniform(0,1) for j in range( self.
           ↪ layers_config[i] ) ]  for i in range(1, self.num_layers)}
        self.previous_m_vals = {param: 0 for param in self.
```

```
            ↪ learnable_params} ## initialize m_0 for weights
        self.previous_m_bias = {i : [0 for j in range(self.
            ↪ layers_config[i])] for i in range(1, self.num_layers)} ##
            ↪  initialize m_0 for bias
        self.previous_v_vals = {param: 0 for param in self.
            ↪ learnable_params} ## initialize m_0 for weights
        self.previous_v_bias = {i : [0 for j in range(self.
            ↪ layers_config[i])] for i in range(1, self.num_layers)} ##
            ↪  initialize m_0 for bias
        data_loader = DataLoader(training_data, batch_size=self.
            ↪ batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0
        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)
            predicted_labels_for_batch = self.forw_prop_vals_at_layers[
                ↪ self.num_layers-1]
            y_preds =  [item for sublist in  predicted_labels_for_batch
                ↪   for item in sublist]
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
                ↪  range(len(class_labels))])
            loss_avg = loss / float(len(class_labels))
            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))
                avg_loss_over_iterations = 0.0
            y_errors_in_batch = list(map(operator.sub, class_labels,
                ↪ y_preds))
            self.backprop_and_update_params_multi_neuron_model(y_preds,
                ↪  y_errors_in_batch, i+1)
        ## removed plt functions
        return loss_running_record ## return the loss record
    def backprop_and_update_params_multi_neuron_model(self, predictions
        ↪ , y_errors, iter): ## add iteration index as aparameter
        pred_err_backproped_at_layers = [{i : [None for j in range(self
            ↪ .layers_config[i])] for i in range(self.num_layers)} for
            ↪ _ in range(self.batch_size)]
        partial_of_loss_wrt_params = {param : 0.0 for param in self.
            ↪ all_params}
        bias_changes =   {i : [0.0 for j in range(self.layers_config[i
            ↪ ])] for i in range(1, self.num_layers)}
        for b in range(self.batch_size):
            pred_err_backproped_at_layers[b][self.num_layers - 1] = [
                ↪ y_errors[b]]
            for back_layer_index in reversed(range(1,self.num_layers)):
                input_vals = self.forw_prop_vals_at_layers[
                    ↪ back_layer_index -1]
```

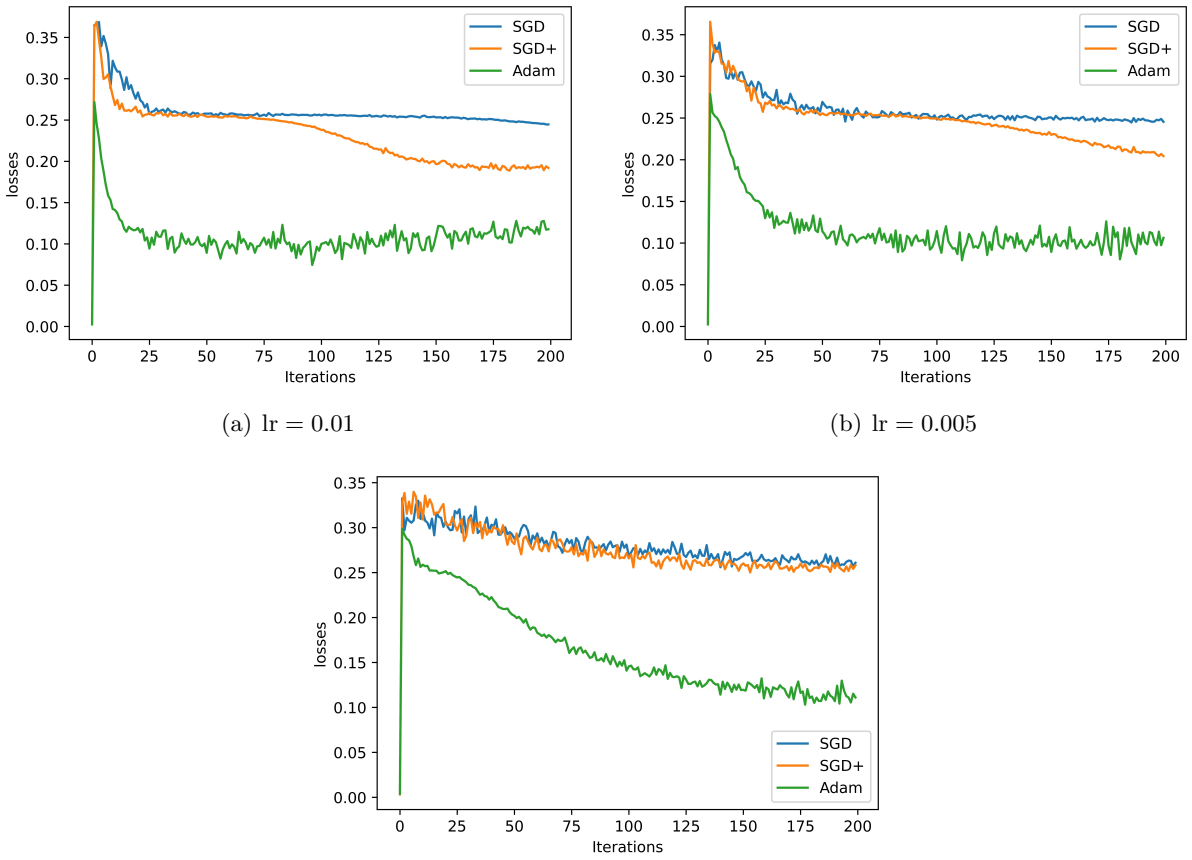
```
                deriv_sigmoids =  self.gradient_vals_for_layers[
                    ↪ back_layer_index]
                vars_in_layer  =  self.layer_vars[back_layer_index]
                vars_in_next_layer_back  =  self.layer_vars[
                    ↪ back_layer_index - 1]
                vals_for_input_vars_dict =  dict(zip(
                    ↪ vars_in_next_layer_back, self.
                    ↪ forw_prop_vals_at_layers[back_layer_index - 1][b
                    ↪ ]))
                layer_params = self.layer_params[back_layer_index]
                transposed_layer_params = list(zip(*layer_params))
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        pred_err_backproped_at_layers[b][
                            ↪ back_layer_index - 1][k] = sum([self.
                            ↪ vals_for_learnable_params[
                            ↪ transposed_layer_params[k][i]] \
                             * pred_err_backproped_at_layers[b][
                                ↪ back_layer_index][i] for i in range(
                                ↪ len(vars_in_layer))])
                for j,var in enumerate(vars_in_layer):
                    layer_params = self.layer_params[back_layer_index][
                        ↪ j]
                    input_vars_to_param_map = self.var_to_var_param[var
                        ↪ ]
                    param_to_vars_map = {param : var for var, param in
                        ↪ input_vars_to_param_map.items()}
                    for i,param in enumerate(layer_params):
                        partial_of_loss_wrt_params[param] += -
                            ↪ pred_err_backproped_at_layers[b][
                            ↪ back_layer_index][j] * \
                             vals_for_input_vars_dict[param_to_vars_map[
                                ↪ param]] * deriv_sigmoids[b][j]
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        if back_layer_index-1 > 0:
                            bias_changes[back_layer_index-1][k] += -
                                ↪ pred_err_backproped_at_layers[b][
                                ↪ back_layer_index - 1][k] *
                                ↪ deriv_sigmoids[b][j]
        for param in partial_of_loss_wrt_params:
            partial_of_loss_wrt_param = partial_of_loss_wrt_params[
                ↪ param] / float(self.batch_size)
            self.previous_m_vals[param] = self.beta_1 * self.
                ↪ previous_m_vals[param] + (1 - self.beta_1) *
                ↪ partial_of_loss_wrt_param ## update first momentum
                ↪ memory
            self.previous_v_vals[param] = self.beta_2 * self.
                ↪ previous_v_vals[param] + (1 - self.beta_2) *
                ↪ partial_of_loss_wrt_param**2 ## update second
                ↪ momentum memory
            m_t_hat = self.previous_m_vals[param] / (1 - self.beta_1**
                ↪ iter) ## scaled first momentum memory
            v_t_hat = self.previous_v_vals[param] / (1 - self.beta_2**
```

```
                ↪ iter) ## scaled second momentum memory
            step = self.learning_rate * m_t_hat / np.sqrt(v_t_hat +
                ↪ self.epsilon) ## scaled step size
            self.vals_for_learnable_params[param] -= step
        ## Finally we update the biases at all the nodes that aggregate
            ↪ data:
        for layer_index in range(1,self.num_layers):
            for k in range(self.layers_config[layer_index]):
                self.previous_m_bias[layer_index][k] = self.beta_1 *
                    ↪ self.previous_m_bias[layer_index][k] + (1 - self.
                    ↪ beta_1) * (bias_changes[layer_index][k] / float(
                    ↪ self.batch_size)) ## update first momentum memory
                self.previous_v_bias[layer_index][k] = self.beta_2 *
                    ↪ self.previous_v_bias[layer_index][k] + (1 - self.
                    ↪ beta_2) * (bias_changes[layer_index][k] / float(
                    ↪ self.batch_size))**2 ## update second momentum
                    ↪ memory
                m_t_hat = self.previous_m_bias[layer_index][k] / (1 -
                    ↪ self.beta_1**iter) ## scaled first momentum
                    ↪ memory
                v_t_hat = self.previous_v_bias[layer_index][k] / (1 -
                    ↪ self.beta_2**iter) ## scaled second momentum
                    ↪ memory
                self.bias[layer_index][k] -= self.learning_rate *
                    ↪ m_t_hat / np.sqrt(v_t_hat + self.epsilon) ##
                    ↪ Update the bias
...
## run multi-neuron case with different optimizer
def run_multi_neuron_diff_optimizer(lr):
    cgp_SGD = MultiNeuronSGDCGP(
                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 = lr,
                training_iterations = 20000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    cgp_SGD.parse_multi_layer_expressions()
    cgp_SGD.display_multi_neuron_network()

    cgp_SGDPlus = MultiNeuronSGDPlusCGP(
                mu = 0.5,
                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 = lr,
                training_iterations = 20000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    cgp_SGDPlus.parse_multi_layer_expressions()
    cgp_SGDPlus.display_multi_neuron_network()

    cgp_Adam = MultiNeuronAdamCGP(
                beta_1 = 0.9,
                beta_2 = 0.99,
                epsilon = 1e-8,
                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 = lr,
                training_iterations = 20000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    cgp_Adam.parse_multi_layer_expressions()
    cgp_Adam.display_multi_neuron_network()

    training_data = cgp_SGD.gen_training_data()
    loss_SGD =  cgp_SGD.run_training_loop_multi_neuron_model(
       ↪ training_data)
    loss_SGDPlus = cgp_SGDPlus.run_training_loop_multi_neuron_model(
       ↪ training_data)
    loss_Adam = cgp_Adam.run_training_loop_multi_neuron_model(
       ↪ training_data)

    plot_learning_curve(loss_SGD, loss_SGDPlus, loss_Adam, '
       ↪ multi_neuron', str(lr))
...
if __name__ == '__main__':
    ...
    if args.task == '2':
        for lr in [1e-2, 5e-3, 1e-3]:
            run_multi_neuron_diff_optimizer(lr)
    ...
```

Fig. 4 shows the loss curves with the learning rates of 0.01, 0.005, and 0.001 respectively. Adam performs the best with no surprise, and SGD still does the worst. When the learning rate is large (as in Fig. 4(a)), Adam converges fastest, and its loss curve is smooth during the converging, but it oscillates after achieving the global minimum point with the same reason mentioned in the last subsection; SGD+ converges slower with a smooth curve and seems hard to find the optimal point; SGD only converges a little. When the learning rate is moderate (as in Fig. 4(b)), Adam still

converges fastest but a little slower with this smaller learning rate; SGD+ and SGD act similarly at the beginning (both oscillate), but SGD+ converges more after the half training. When the learning rate is small (as in Fig. 3(c)), only Adam converges, but it takes a longer time; SGD+ and SGD both do not converge, oscillating obviously all the way. Compared with Fig. 1(b), `torch.nn` performs better than SGD+ and SGD, but it is not as good as Adam. Although `torch.nn` is optimized well, dealing with more complex problems (more layers), Adam wins with better logic.



(a) lr = 0.01

(b) lr = 0.005

(c) lr = 0.001

Figure 4: Loss curves for multi-neuron case with different learning rates.

2.3 Exploring hyperparameters of Adam

Note: The code for this subsection is borrowed from the `ComputationalGraphPrimer` package (https://engineering.purdue.edu/kak/distCGP/).

The code for this task is shown below. To test different hyperparameters, `run_one_neuron_diff_adam` function is created and called in a `for` loop with different parameters. In this task, only one-neuron model is used.

```
...
## run one-neuron case with adam using different beta_1 and beta_2
def run_one_neuron_diff_adam(beta_1, beta_2):
    cgp_Adam = OneNeuronAdamCGP(
                beta_1 = beta_1,
                beta_2 = beta_2,
```

```
                epsilon = 1e-8,
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = 1e-3,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    cgp_Adam.parse_expressions()
    cgp_Adam.display_one_neuron_network()
    training_data = cgp_Adam.gen_training_data()
    start_time = time.time()
    loss_Adam = cgp_Adam.run_training_loop_one_neuron_model(
       ↪ training_data)
    time_used = time.time() - start_time
    final_loss = loss_Adam[-1]
    min_loss = min(loss_Adam[1:])
    return time_used, final_loss, min_loss

if __name__ == '__main__':
    ...
    if args.task == '3':
        beta_list = []
        time_list = []
        final_l_list = []
        min_l_list = []
        for beta_1 in [0.8, 0.9, 0.99]:
            for beta_2 in [0.8, 0.9, 0.99]:
                time_used, final_loss, min_loss =
                   ↪ run_one_neuron_diff_adam(beta_1, beta_2)
                beta_list.append((beta_1, beta_2))
                time_list.append(time_used)
                final_l_list.append(final_loss)
                min_l_list.append(min_loss)
        for i in range(len(beta_list)):
            print('Performance with beta_1 %.2f and beta_2 %.2f \n----
               ↪ time: %fs; final loss: %f; minimum loss: %f'%(
               ↪ beta_list[i][0], beta_list[i][1], time_list[i],
               ↪ final_l_list[i], min_l_list[i]))
```

Output:

```
Performance with beta_1 0.80 and beta_2 0.80
---- time: 8.037810s; final loss: 0.097885; minimum loss: 0.085376
Performance with beta_1 0.80 and beta_2 0.90
---- time: 8.147538s; final loss: 0.107459; minimum loss: 0.084445
Performance with beta_1 0.80 and beta_2 0.99
---- time: 8.146399s; final loss: 0.105753; minimum loss: 0.083214
Performance with beta_1 0.90 and beta_2 0.80
---- time: 8.216441s; final loss: 0.091616; minimum loss: 0.083619
Performance with beta_1 0.90 and beta_2 0.90
---- time: 8.260545s; final loss: 0.090110; minimum loss: 0.079896
```

```
Performance with beta_1 0.90 and beta_2 0.99
---- time: 8.219999s; final loss: 0.085230; minimum loss: 0.079815
Performance with beta_1 0.99 and beta_2 0.80
---- time: 8.225816s; final loss: 0.095684; minimum loss: 0.088200
Performance with beta_1 0.99 and beta_2 0.90
---- time: 8.173840s; final loss: 0.099921; minimum loss: 0.087710
Performance with beta_1 0.99 and beta_2 0.99
---- time: 8.106234s; final loss: 0.112987; minimum loss: 0.081915
```

β_1 is used to maintain the direction of the gradient from the previous step, making the update more stable and helping to get out of local minimum points. β_2 prevents the gradient from being updated too aggressively, which is similar to adding a penalty term. The performance of Adam with different β_1 and β_2 is shown in Tab. 1. The running times are similar, meaning these two parameters do not affect the running time for each iteration. The final losses are various, but it does not matter, since the loss curve oscillates. But the minimum losses are important. When β_1 is a constant, the minimum loss decreases with increasing β_2. When β_2 is a constant, the minimum loss is smallest with $\beta_1 = 0.9$ (neither too large nor too small!). The best combination is $\beta_1 = 0.9$ and $\beta_2 = 0.99$. These are also the recommended values from the Adam paper https://arxiv.org/abs/1412.6980.

Table 1: performance of Adam with different hyperparameters.

β_1	β_2	running time (s)	final loss	minimum loss
0.8	0.8	8.0378	0.0979	0.0854
	0.9	8.1475	0.1075	0.0844
	0.99	8.1464	0.1058	0.0832
0.9	0.8	8.2164	0.0916	0.0836
	0.9	8.2605	0.0901	0.0799
	0.99	8.2200	0.0852	0.0798
0.99	0.8	8.2258	0.0957	0.0882
	0.9	8.1738	0.0999	0.0877
	0.99	8.1062	0.1130	0.0819

3 Source Code

```
import sys,os,os.path
import numpy as np
import re
import operator
import itertools
import math
import random
import argparse
import time
import torch
from collections import deque
import copy
import matplotlib.pyplot as plt
import networkx as nx

from ComputationalGraphPrimer import *

```

```
## Set ramdom seed
def set_random_seed(seed):
    random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
    np.random.seed(seed)
    torch.backends.cudnn.deterministic = True
    torch.backends.cudnn.benchmarks = False
    os.environ['PYTHONHASHSEED'] = str(seed)

class OneNeuronSGDCGP(ComputationalGraphPrimer):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs) ## inherit from CGP
    def run_training_loop_one_neuron_model(self, training_data):
        self.vals_for_learnable_params = {param: random.uniform(0,1)
            ↪ for param in self.learnable_params}
        self.bias = random.uniform(0,1)
        class DataLoader:
            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]]
                self.class_1_samples = [(item, 1) for item in self.
                    ↪ training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.
                    ↪ training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
                maxval = 0.0
                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]
                batch = [batch_data, batch_labels]
                return batch
        data_loader = DataLoader(training_data, batch_size=self.
            ↪ batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples_in_batch = data[0]
```

```
67              class_labels_in_batch = data[1]
68              y_preds, deriv_sigmoids =  self.
                    ↪ forward_prop_one_neuron_model(data_tuples_in_batch)
69              loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
                    ↪  for i in range(len(class_labels_in_batch))])
70              avg_loss_over_iterations += loss / float(len(
                    ↪ class_labels_in_batch))
71              if i%(self.display_loss_how_often) == 0:
72                  avg_loss_over_iterations /= self.display_loss_how_often
73                  loss_running_record.append(avg_loss_over_iterations)
74                  print("[iter=%d]  loss = %.4f" % (i+1,
                        ↪ avg_loss_over_iterations))
75                  avg_loss_over_iterations = 0.0
76              y_errors_in_batch = list(map(operator.sub,
                    ↪ class_labels_in_batch, y_preds))
77              self.backprop_and_update_params_one_neuron_model(
                    ↪ data_tuples_in_batch, y_preds, y_errors_in_batch,
                    ↪ deriv_sigmoids)
78          ## removed plt functions
79          return loss_running_record ## return the loss record
80
81  class OneNeuronSGDPlusCGP(ComputationalGraphPrimer):
82      def __init__(self, mu, *args, **kwargs):
83          super().__init__(*args, **kwargs) ## inherit from CGP
84          self.mu = mu ## momentum factor
85      def run_training_loop_one_neuron_model(self, training_data):
86          self.vals_for_learnable_params = {param: random.uniform(0,1)
                ↪ for param in self.learnable_params}
87          self.bias = random.uniform(0,1)
88          self.previous_v_vals = {param: 0 for param in self.
                ↪ learnable_params} ## initialize v_0 for weights
89          self.previous_v_bias = 0 ## initialize v_0 for bias
90          class DataLoader:
91              def __init__(self, training_data, batch_size):
92                  self.training_data = training_data
93                  self.batch_size = batch_size
94                  self.class_0_samples = [(item, 0) for item in self.
                        ↪ training_data[0]]
95                  self.class_1_samples = [(item, 1) for item in self.
                        ↪ training_data[1]]
96              def __len__(self):
97                  return len(self.training_data[0]) + len(self.
                        ↪ training_data[1])
98              def _getitem(self):
99                  cointoss = random.choice([0,1])
100                 if cointoss == 0:
101                     return random.choice(self.class_0_samples)
102                 else:
103                     return random.choice(self.class_1_samples)
104             def getbatch(self):
105                 batch_data,batch_labels = [],[]
106                 maxval = 0.0
107                 for _ in range(self.batch_size):
108                     item = self._getitem()
```

```
109                     if np.max(item[0]) > maxval:
110                         maxval = np.max(item[0])
111                     batch_data.append(item[0])
112                     batch_labels.append(item[1])
113                 batch_data = [item/maxval for item in batch_data]
114                 batch = [batch_data, batch_labels]
115                 return batch
116         data_loader = DataLoader(training_data, batch_size=self.
                ↪ batch_size)
117         loss_running_record = []
118         i = 0
119         avg_loss_over_iterations = 0.0
120         for i in range(self.training_iterations):
121             data = data_loader.getbatch()
122             data_tuples_in_batch = data[0]
123             class_labels_in_batch = data[1]
124             y_preds, deriv_sigmoids =  self.
                    ↪ forward_prop_one_neuron_model(data_tuples_in_batch)
125             loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
                    ↪  for i in range(len(class_labels_in_batch))])
126             avg_loss_over_iterations += loss / float(len(
                    ↪ class_labels_in_batch))
127             if i%(self.display_loss_how_often) == 0:
128                 avg_loss_over_iterations /= self.display_loss_how_often
129                 loss_running_record.append(avg_loss_over_iterations)
130                 print("[iter=%d]  loss = %.4f" %  (i+1,
                        ↪ avg_loss_over_iterations))
131                 avg_loss_over_iterations = 0.0
132             y_errors_in_batch = list(map(operator.sub,
                    ↪ class_labels_in_batch, y_preds))
133             self.backprop_and_update_params_one_neuron_model(
                    ↪ data_tuples_in_batch, y_preds, y_errors_in_batch,
                    ↪ deriv_sigmoids)
134         ## removed plt functions
135         return loss_running_record ## return the loss record
136     def backprop_and_update_params_one_neuron_model(self,
            ↪ data_tuples_in_batch, predictions, y_errors_in_batch,
            ↪ deriv_sigmoids):
137         input_vars = self.independent_vars
138         input_vars_to_param_map = self.var_to_var_param[self.
                ↪ output_vars[0]]
139         param_to_vars_map = {param : var for var, param in
                ↪ input_vars_to_param_map.items()}
140         vals_for_learnable_params = self.vals_for_learnable_params
141         for i,param in enumerate(self.vals_for_learnable_params):
142             partial_of_loss_wrt_param = 0.0
143             for j in range(self.batch_size):
144                 vals_for_input_vars_dict = dict(zip(input_vars, list(
                        ↪ data_tuples_in_batch[j])))
145                 partial_of_loss_wrt_param += - y_errors_in_batch[j] *
                        ↪ vals_for_input_vars_dict[param_to_vars_map[param
                        ↪ ]] * deriv_sigmoids[j]
146             partial_of_loss_wrt_param /= float(self.batch_size) ##
                    ↪ average after the batch finishing
```

```
147             self.previous_v_vals[param] = self.mu * self.
                    ↪ previous_v_vals[param] + partial_of_loss_wrt_param ##
                    ↪  update momentum memory
148             step = self.learning_rate * self.previous_v_vals[param] ##
                    ↪ scaled step size
149             self.vals_for_learnable_params[param] -= step
150         partial_of_loss = 0 ## initialize loss count
151         for j in range(self.batch_size): ## loop for a batch
152             partial_of_loss -= y_errors_in_batch[j] * deriv_sigmoids[j]
153         partial_of_loss /= float(self.batch_size) ## average after the
                ↪ batch finishing
154         self.previous_v_bias = self.mu * self.previous_v_bias +
                ↪ partial_of_loss ## update momentum memory
155         self.bias -= self.learning_rate * self.previous_v_bias ##
                ↪ Update the bias
156
157 class OneNeuronAdamCGP(ComputationalGraphPrimer):
158     def __init__(self, beta_1, beta_2, epsilon, *args, **kwargs):
159         super().__init__(*args, **kwargs) ## inherit from CGP
160         self.beta_1 = beta_1 ## first exponential decay rate
161         self.beta_2 = beta_2 ## second exponential decay rate
162         self.epsilon = epsilon ## epsilon
163     def run_training_loop_one_neuron_model(self, training_data):
164         self.vals_for_learnable_params = {param: random.uniform(0,1)
                ↪ for param in self.learnable_params}
165         self.bias = random.uniform(0,1)
166         self.previous_m_vals = {param: 0 for param in self.
                ↪ learnable_params} ## initialize m_0 for weights
167         self.previous_m_bias = 0 ## initialize m_0 for bias
168         self.previous_v_vals = {param: 0 for param in self.
                ↪ learnable_params} ## initialize m_0 for weights
169         self.previous_v_bias = 0 ## initialize m_0 for bias
170         class DataLoader:
171             def __init__(self, training_data, batch_size):
172                 self.training_data = training_data
173                 self.batch_size = batch_size
174                 self.class_0_samples = [(item, 0) for item in self.
                        ↪ training_data[0]]
175                 self.class_1_samples = [(item, 1) for item in self.
                        ↪ training_data[1]]
176             def __len__(self):
177                 return len(self.training_data[0]) + len(self.
                        ↪ training_data[1])
178             def _getitem(self):
179                 cointoss = random.choice([0,1])
180                 if cointoss == 0:
181                     return random.choice(self.class_0_samples)
182                 else:
183                     return random.choice(self.class_1_samples)
184             def getbatch(self):
185                 batch_data,batch_labels = [],[]
186                 maxval = 0.0
187                 for _ in range(self.batch_size):
188                     item = self._getitem()
```

```
189                     if np.max(item[0]) > maxval:
190                         maxval = np.max(item[0])
191                     batch_data.append(item[0])
192                     batch_labels.append(item[1])
193                 batch_data = [item/maxval for item in batch_data]
194                 batch = [batch_data, batch_labels]
195                 return batch
196         data_loader = DataLoader(training_data, batch_size=self.
                ↪ batch_size)
197         loss_running_record = []
198         i = 0
199         avg_loss_over_iterations = 0.0
200         for i in range(self.training_iterations):
201             data = data_loader.getbatch()
202             data_tuples_in_batch = data[0]
203             class_labels_in_batch = data[1]
204             y_preds, deriv_sigmoids =  self.
                    ↪ forward_prop_one_neuron_model(data_tuples_in_batch)
205             loss = sum([(abs(class_labels_in_batch[i] - y_preds[i]))**2
                    ↪  for i in range(len(class_labels_in_batch))])
206             avg_loss_over_iterations += loss / float(len(
                    ↪ class_labels_in_batch))
207             if i%(self.display_loss_how_often) == 0:
208                 avg_loss_over_iterations /= self.display_loss_how_often
209                 loss_running_record.append(avg_loss_over_iterations)
210                 print("[iter=%d]  loss = %.4f" %  (i+1,
                        ↪ avg_loss_over_iterations))
211                 avg_loss_over_iterations = 0.0
212             y_errors_in_batch = list(map(operator.sub,
                    ↪ class_labels_in_batch, y_preds))
213             self.backprop_and_update_params_one_neuron_model(
                    ↪ data_tuples_in_batch, y_preds, y_errors_in_batch,
                    ↪ deriv_sigmoids, i+1)
214         ## removed plt functions
215         return loss_running_record ## return the loss record
216     def backprop_and_update_params_one_neuron_model(self,
            ↪ data_tuples_in_batch, predictions, y_errors_in_batch,
            ↪ deriv_sigmoids, iter): ## add iteration index as aparameter
217         input_vars = self.independent_vars
218         input_vars_to_param_map = self.var_to_var_param[self.
                ↪ output_vars[0]]
219         param_to_vars_map = {param : var for var, param in
                ↪ input_vars_to_param_map.items()}
220         vals_for_learnable_params = self.vals_for_learnable_params
221         for i,param in enumerate(self.vals_for_learnable_params):
222             partial_of_loss_wrt_param = 0.0
223             for j in range(self.batch_size):
224                 vals_for_input_vars_dict = dict(zip(input_vars, list(
                        ↪ data_tuples_in_batch[j])))
225                 partial_of_loss_wrt_param += - y_errors_in_batch[j] *
                        ↪ vals_for_input_vars_dict[param_to_vars_map[param
                        ↪ ]] * deriv_sigmoids[j]
226             partial_of_loss_wrt_param /= float(self.batch_size) ##
                    ↪ average after the batch finishing
```

```
227             self.previous_m_vals[param] = self.beta_1 * self.
                    ↪ previous_m_vals[param] + (1 - self.beta_1) *
                    ↪ partial_of_loss_wrt_param ## update first momentum
                    ↪ memory
228             self.previous_v_vals[param] = self.beta_2 * self.
                    ↪ previous_v_vals[param] + (1 - self.beta_2) *
                    ↪ partial_of_loss_wrt_param**2 ## update second
                    ↪ momentum memory
229             m_t_hat = self.previous_m_vals[param] / (1 - self.beta_1**
                    ↪ iter) ## scaled first momentum memory
230             v_t_hat = self.previous_v_vals[param] / (1 - self.beta_2**
                    ↪ iter) ## scaled second momentum memory
231             step = self.learning_rate * m_t_hat / np.sqrt(v_t_hat +
                    ↪ self.epsilon) ## scaled step size
232             self.vals_for_learnable_params[param] -= step
233         partial_of_loss = 0 ## initialize loss count
234         for j in range(self.batch_size): ## loop for a batch
235             partial_of_loss -= y_errors_in_batch[j] * deriv_sigmoids[j]
236         partial_of_loss /= float(self.batch_size) ## average after the
                ↪ batch finishing
237         self.previous_m_bias = self.beta_1 * self.previous_m_bias + (1
                ↪ - self.beta_1) * partial_of_loss ## update first momentum
                ↪  memory
238         self.previous_v_bias = self.beta_2 * self.previous_v_bias + (1
                ↪ - self.beta_2) * partial_of_loss**2 ## update second
                ↪ momentum memory
239         m_t_hat = self.previous_m_bias / (1 - self.beta_1**iter) ##
                ↪ scaled first momentum memory
240         v_t_hat = self.previous_v_bias / (1 - self.beta_2**iter) ##
                ↪ scaled second momentum memory
241         self.bias -= self.learning_rate * m_t_hat / np.sqrt(v_t_hat +
                ↪ self.epsilon) ## Update the bias
242
243 class MultiNeuronSGDCGP(ComputationalGraphPrimer):
244     def __init__(self, *args, **kwargs):
245         super().__init__(*args, **kwargs) ## inherit from CGP
246     def run_training_loop_multi_neuron_model(self, training_data):
247         class DataLoader:
248             def __init__(self, training_data, batch_size):
249                 self.training_data = training_data
250                 self.batch_size = batch_size
251                 self.class_0_samples = [(item, 0) for item in self.
                        ↪ training_data[0]]
252                 self.class_1_samples = [(item, 1) for item in self.
                        ↪ training_data[1]]
253             def __len__(self):
254                 return len(self.training_data[0]) + len(self.
                        ↪ training_data[1])
255             def _getitem(self):
256                 cointoss = random.choice([0,1])
257                 if cointoss == 0:
258                     return random.choice(self.class_0_samples)
259                 else:
260                     return random.choice(self.class_1_samples)
```

```
261             def getbatch(self):
262                 batch_data,batch_labels = [],[]
263                 maxval = 0.0
264                 for _ in range(self.batch_size):
265                     item = self._getitem()
266                     if np.max(item[0]) > maxval:
267                         maxval = np.max(item[0])
268                     batch_data.append(item[0])
269                     batch_labels.append(item[1])
270                 batch_data = [item/maxval for item in batch_data]
271                 batch = [batch_data, batch_labels]
272                 return batch
273         self.vals_for_learnable_params = {param: random.uniform(0,1)
               ↪ for param in self.learnable_params}
274         self.bias = {i : [random.uniform(0,1) for j in range(self.
               ↪ layers_config[i])] for i in range(1, self.num_layers)}
275         data_loader = DataLoader(training_data, batch_size=self.
               ↪ batch_size)
276         loss_running_record = []
277         i = 0
278         avg_loss_over_iterations = 0.0
279         for i in range(self.training_iterations):
280             data = data_loader.getbatch()
281             data_tuples = data[0]
282             class_labels = data[1]
283             self.forward_prop_multi_neuron_model(data_tuples)
284             predicted_labels_for_batch = self.forw_prop_vals_at_layers[
                   ↪ self.num_layers-1]
285             y_preds =  [item for sublist in  predicted_labels_for_batch
                   ↪   for item in sublist]
286             loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
                   ↪  range(len(class_labels))])
287             loss_avg = loss / float(len(class_labels))
288             avg_loss_over_iterations += loss_avg
289             if i%(self.display_loss_how_often) == 0:
290                 avg_loss_over_iterations /= self.display_loss_how_often
291                 loss_running_record.append(avg_loss_over_iterations)
292                 print("[iter=%d]  loss = %.4f" %  (i+1,
                       ↪ avg_loss_over_iterations))
293                 avg_loss_over_iterations = 0.0
294             y_errors_in_batch = list(map(operator.sub, class_labels,
                   ↪ y_preds))
295             self.backprop_and_update_params_multi_neuron_model(y_preds,
                   ↪  y_errors_in_batch)
296         ## removed plt functions
297         return loss_running_record ## return the loss record
298
299 class MultiNeuronSGDPlusCGP(ComputationalGraphPrimer):
300     def __init__(self, mu, *args, **kwargs):
301         super().__init__(*args, **kwargs) ## inherit from CGP
302         self.mu = mu ## momentum factor
303     def run_training_loop_multi_neuron_model(self, training_data):
304         class DataLoader:
305             def __init__(self, training_data, batch_size):
```

```
306                 self.training_data = training_data
307                 self.batch_size = batch_size
308                 self.class_0_samples = [(item, 0) for item in self.
                        ↪ training_data[0]]
309                 self.class_1_samples = [(item, 1) for item in self.
                        ↪ training_data[1]]
310             def __len__(self):
311                 return len(self.training_data[0]) + len(self.
                        ↪ training_data[1])
312             def _getitem(self):
313                 cointoss = random.choice([0,1])
314                 if cointoss == 0:
315                     return random.choice(self.class_0_samples)
316                 else:
317                     return random.choice(self.class_1_samples)
318             def getbatch(self):
319                 batch_data,batch_labels = [],[]
320                 maxval = 0.0
321                 for _ in range(self.batch_size):
322                     item = self._getitem()
323                     if np.max(item[0]) > maxval:
324                         maxval = np.max(item[0])
325                     batch_data.append(item[0])
326                     batch_labels.append(item[1])
327                 batch_data = [item/maxval for item in batch_data]
328                 batch = [batch_data, batch_labels]
329                 return batch
330         self.vals_for_learnable_params = {param: random.uniform(0,1)
                ↪ for param in self.learnable_params}
331         self.bias =   {i : [random.uniform(0,1) for j in range(self.
                ↪ layers_config[i])] for i in range(1, self.num_layers)}
332         self.previous_v_vals = {param: 0 for param in self.
                ↪ learnable_params} ## initialize v_0 for weights
333         self.previous_v_bias = {i : [0 for j in range(self.
                ↪ layers_config[i])] for i in range(1, self.num_layers)} ##
                ↪  initialize v_0 for bias
334         data_loader = DataLoader(training_data, batch_size=self.
                ↪ batch_size)
335         loss_running_record = []
336         i = 0
337         avg_loss_over_iterations = 0.0
338         for i in range(self.training_iterations):
339             data = data_loader.getbatch()
340             data_tuples = data[0]
341             class_labels = data[1]
342             self.forward_prop_multi_neuron_model(data_tuples)
343             predicted_labels_for_batch = self.forw_prop_vals_at_layers[
                    ↪ self.num_layers-1]
344             y_preds =  [item for sublist in  predicted_labels_for_batch
                    ↪   for item in sublist]
345             loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
                    ↪  range(len(class_labels))])
346             loss_avg = loss / float(len(class_labels))
347             avg_loss_over_iterations += loss_avg
```

```
348             if i%(self.display_loss_how_often) == 0:
349                 avg_loss_over_iterations /= self.display_loss_how_often
350                 loss_running_record.append(avg_loss_over_iterations)
351                 print("[iter=%d]  loss = %.4f" %  (i+1,
                        ↪ avg_loss_over_iterations))
352                 avg_loss_over_iterations = 0.0
353             y_errors_in_batch = list(map(operator.sub, class_labels,
                    ↪ y_preds))
354             self.backprop_and_update_params_multi_neuron_model(y_preds,
                    ↪  y_errors_in_batch)
355         ## removed plt functions
356         return loss_running_record ## return the loss record
357     def backprop_and_update_params_multi_neuron_model(self, predictions
            ↪ , y_errors):
358         pred_err_backproped_at_layers = [{i : [None for j in range(self
                ↪ .layers_config[i])] for i in range(self.num_layers)} for
                ↪ _ in range(self.batch_size)]
359         partial_of_loss_wrt_params = {param : 0.0 for param in self.
                ↪ all_params}
360         bias_changes =   {i : [0.0 for j in range(self.layers_config[i
                ↪ ])] for i in range(1, self.num_layers)}
361         for b in range(self.batch_size):
362             pred_err_backproped_at_layers[b][self.num_layers - 1] = [
                    ↪ y_errors[b] ]
363             for back_layer_index in reversed(range(1,self.num_layers)):
364                 input_vals = self.forw_prop_vals_at_layers[
                        ↪ back_layer_index -1]
365                 deriv_sigmoids =  self.gradient_vals_for_layers[
                        ↪ back_layer_index]
366                 vars_in_layer  =  self.layer_vars[back_layer_index]
367                 vars_in_next_layer_back  =  self.layer_vars[
                        ↪ back_layer_index - 1]
368                 vals_for_input_vars_dict =  dict(zip(
                        ↪ vars_in_next_layer_back, self.
                        ↪ forw_prop_vals_at_layers[back_layer_index - 1][b
                        ↪ ]))
369                 layer_params = self.layer_params[back_layer_index]
370                 transposed_layer_params = list(zip(*layer_params))
371                 for k,var1 in enumerate(vars_in_next_layer_back):
372                     for j,var2 in enumerate(vars_in_layer):
373                         pred_err_backproped_at_layers[b][
                                ↪ back_layer_index - 1][k] = sum([self.
                                ↪ vals_for_learnable_params[
                                ↪ transposed_layer_params[k][i]] \
374                                 * pred_err_backproped_at_layers[b][
                                    ↪ back_layer_index][i] for i in range(
                                    ↪ len(vars_in_layer))])
375                 for j,var in enumerate(vars_in_layer):
376                     layer_params = self.layer_params[back_layer_index][
                            ↪ j]
377                     input_vars_to_param_map = self.var_to_var_param[var
                            ↪ ]
378                     param_to_vars_map = {param : var for var, param in
                            ↪ input_vars_to_param_map.items()}
```

```
                    for i,param in enumerate(layer_params):
                        partial_of_loss_wrt_params[param] +=
                            ↪ pred_err_backproped_at_layers[b][
                            ↪ back_layer_index][j] * \
                             vals_for_input_vars_dict[param_to_vars_map[
                                ↪ param]] * deriv_sigmoids[b][j]
                for k,var1 in enumerate(vars_in_next_layer_back):
                    for j,var2 in enumerate(vars_in_layer):
                        if back_layer_index-1 > 0:
                            bias_changes[back_layer_index-1][k] +=
                                ↪ pred_err_backproped_at_layers[b][
                                ↪ back_layer_index - 1][k] *
                                ↪ deriv_sigmoids[b][j]
        for param in partial_of_loss_wrt_params:
            partial_of_loss_wrt_param = partial_of_loss_wrt_params[
                ↪ param] /  float(self.batch_size)
            self.previous_v_vals[param] = self.mu * self.
                ↪ previous_v_vals[param] + partial_of_loss_wrt_param ##
                ↪  update momentum memory
            step = self.learning_rate * self.previous_v_vals[param] ##
                ↪ scaled step size
            self.vals_for_learnable_params[param] += step
        for layer_index in range(1,self.num_layers):
            for k in range(self.layers_config[layer_index]):
                self.previous_v_bias[layer_index][k] = self.mu * self.
                    ↪ previous_v_bias[layer_index][k] + (bias_changes[
                    ↪ layer_index][k] / float(self.batch_size)) ##
                    ↪ update momentum memory
                self.bias[layer_index][k]  +=  self.learning_rate *
                    ↪ self.previous_v_bias[layer_index][k] ## Update
                    ↪ the bias

class MultiNeuronAdamCGP(ComputationalGraphPrimer):
    def __init__(self, beta_1, beta_2, epsilon, *args, **kwargs):
        super().__init__(*args, **kwargs) ## inherit from CGP
        self.beta_1 = beta_1 ## first exponential decay rate
        self.beta_2 = beta_2 ## second exponential decay rate
        self.epsilon = epsilon ## epsilon
    def run_training_loop_multi_neuron_model(self, training_data):
        class DataLoader:
            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]]
                self.class_1_samples = [(item, 1) for item in self.
                    ↪ training_data[1]]
            def __len__(self):
                return len(self.training_data[0]) + len(self.
                    ↪ training_data[1])
            def _getitem(self):
                cointoss = random.choice([0,1])
                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 = [],[]
                maxval = 0.0
                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]
                batch = [batch_data, batch_labels]
                return batch
        self.vals_for_learnable_params = {param: random.uniform(0,1)
            ↪ for param in self.learnable_params}
        self.bias =   {i : [random.uniform(0,1) for j in range( self.
            ↪ layers_config[i] ) ]  for i in range(1, self.num_layers)}
        self.previous_m_vals = {param: 0 for param in self.
            ↪ learnable_params} ## initialize m_0 for weights
        self.previous_m_bias = {i : [0 for j in range(self.
            ↪ layers_config[i])] for i in range(1, self.num_layers)} ##
            ↪  initialize m_0 for bias
        self.previous_v_vals = {param: 0 for param in self.
            ↪ learnable_params} ## initialize m_0 for weights
        self.previous_v_bias = {i : [0 for j in range(self.
            ↪ layers_config[i])] for i in range(1, self.num_layers)} ##
            ↪  initialize m_0 for bias
        data_loader = DataLoader(training_data, batch_size=self.
            ↪ batch_size)
        loss_running_record = []
        i = 0
        avg_loss_over_iterations = 0.0
        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)
            predicted_labels_for_batch = self.forw_prop_vals_at_layers[
                ↪ self.num_layers-1]
            y_preds =  [item for sublist in  predicted_labels_for_batch
                ↪   for item in sublist]
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in
                ↪  range(len(class_labels))])
            loss_avg = loss / float(len(class_labels))
            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))
                avg_loss_over_iterations = 0.0
            y_errors_in_batch = list(map(operator.sub, class_labels,
                ↪ y_preds))
```

```
455             self.backprop_and_update_params_multi_neuron_model(y_preds,
                    ↪  y_errors_in_batch, i+1)
456         ## removed plt functions
457         return loss_running_record ## return the loss record
458     def backprop_and_update_params_multi_neuron_model(self, predictions
            ↪ , y_errors, iter): ## add iteration index as aparameter
459         pred_err_backproped_at_layers = [{i : [None for j in range(self
                ↪ .layers_config[i])] for i in range(self.num_layers)} for
                ↪ _ in range(self.batch_size)]
460         partial_of_loss_wrt_params = {param : 0.0 for param in self.
                ↪ all_params}
461         bias_changes =   {i : [0.0 for j in range(self.layers_config[i
                ↪ ])] for i in range(1, self.num_layers)}
462         for b in range(self.batch_size):
463             pred_err_backproped_at_layers[b][self.num_layers - 1] = [
                    ↪ y_errors[b]]
464             for back_layer_index in reversed(range(1,self.num_layers)):
465                 input_vals = self.forw_prop_vals_at_layers[
                        ↪ back_layer_index -1]
466                 deriv_sigmoids =  self.gradient_vals_for_layers[
                        ↪ back_layer_index]
467                 vars_in_layer  =  self.layer_vars[back_layer_index]
468                 vars_in_next_layer_back  =  self.layer_vars[
                        ↪ back_layer_index - 1]
469                 vals_for_input_vars_dict =  dict(zip(
                        ↪ vars_in_next_layer_back, self.
                        ↪ forw_prop_vals_at_layers[back_layer_index - 1][b
                        ↪ ]))
470                 layer_params = self.layer_params[back_layer_index]
471                 transposed_layer_params = list(zip(*layer_params))
472                 for k,var1 in enumerate(vars_in_next_layer_back):
473                     for j,var2 in enumerate(vars_in_layer):
474                         pred_err_backproped_at_layers[b][
                                ↪ back_layer_index - 1][k] = sum([self.
                                ↪ vals_for_learnable_params[
                                ↪ transposed_layer_params[k][i]] \
475                          * pred_err_backproped_at_layers[b][
                                    ↪ back_layer_index][i] for i in range(
                                    ↪ len(vars_in_layer))])
476                 for j,var in enumerate(vars_in_layer):
477                     layer_params = self.layer_params[back_layer_index][
                            ↪ j]
478                     input_vars_to_param_map = self.var_to_var_param[var
                            ↪ ]
479                     param_to_vars_map = {param : var for var, param in
                            ↪ input_vars_to_param_map.items()}
480                     for i,param in enumerate(layer_params):
481                         partial_of_loss_wrt_params[param] += -
                                ↪ pred_err_backproped_at_layers[b][
                                ↪ back_layer_index][j] * \
482                          vals_for_input_vars_dict[param_to_vars_map[
                                ↪ param]] * deriv_sigmoids[b][j]
483                 for k,var1 in enumerate(vars_in_next_layer_back):
484                     for j,var2 in enumerate(vars_in_layer):
```

```
485                         if back_layer_index-1 > 0:
486                             bias_changes[back_layer_index-1][k] += -
                                    ↪ pred_err_backproped_at_layers[b][
                                    ↪ back_layer_index - 1][k] *
                                    ↪ deriv_sigmoids[b][j]
487         for param in partial_of_loss_wrt_params:
488             partial_of_loss_wrt_param = partial_of_loss_wrt_params[
                    ↪ param] / float(self.batch_size)
489             self.previous_m_vals[param] = self.beta_1 * self.
                    ↪ previous_m_vals[param] + (1 - self.beta_1) *
                    ↪ partial_of_loss_wrt_param ## update first momentum
                    ↪ memory
490             self.previous_v_vals[param] = self.beta_2 * self.
                    ↪ previous_v_vals[param] + (1 - self.beta_2) *
                    ↪ partial_of_loss_wrt_param**2 ## update second
                    ↪ momentum memory
491             m_t_hat = self.previous_m_vals[param] / (1 - self.beta_1**
                    ↪ iter) ## scaled first momentum memory
492             v_t_hat = self.previous_v_vals[param] / (1 - self.beta_2**
                    ↪ iter) ## scaled second momentum memory
493             step = self.learning_rate * m_t_hat / np.sqrt(v_t_hat +
                    ↪ self.epsilon) ## scaled step size
494             self.vals_for_learnable_params[param] -= step
495         ## Finally we update the biases at all the nodes that aggregate
                ↪  data:
496         for layer_index in range(1,self.num_layers):
497             for k in range(self.layers_config[layer_index]):
498                 self.previous_m_bias[layer_index][k] = self.beta_1 *
                        ↪ self.previous_m_bias[layer_index][k] + (1 - self.
                        ↪ beta_1) * (bias_changes[layer_index][k] / float(
                        ↪ self.batch_size)) ## update first momentum memory
499                 self.previous_v_bias[layer_index][k] = self.beta_2 *
                        ↪ self.previous_v_bias[layer_index][k] + (1 - self.
                        ↪ beta_2) * (bias_changes[layer_index][k] / float(
                        ↪ self.batch_size))**2 ## update second momentum
                        ↪ memory
500                 m_t_hat = self.previous_m_bias[layer_index][k] / (1 -
                        ↪ self.beta_1**iter) ## scaled first momentum
                        ↪ memory
501                 v_t_hat = self.previous_v_bias[layer_index][k] / (1 -
                        ↪ self.beta_2**iter) ## scaled second momentum
                        ↪ memory
502                 self.bias[layer_index][k] -= self.learning_rate *
                        ↪ m_t_hat / np.sqrt(v_t_hat + self.epsilon) ##
                        ↪ Update the bias
503
504 ## plot the learning curves
505 def plot_learning_curve(loss_SGD, loss_SGDPlus, loss_Adam, title, lr):
506     plt.figure(figsize=(6, 4))
507     plt.plot(loss_SGD, label='SGD')
508     plt.plot(loss_SGDPlus, label='SGD+')
509     plt.plot(loss_Adam, label='Adam')
510     plt.xlabel('Iterations')
511     plt.ylabel('losses')
```

```
    plt.legend()
    cf = plt.gcf()
    cf.savefig('%s_%s.jpg'%(title,lr), bbox_inches='tight', \
                format='jpg', dpi=1000)

## run one-neuron case with different optimizer
def run_one_neuron_diff_optimizer(lr):
    cgp_SGD = OneNeuronSGDCGP(
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    cgp_SGD.parse_expressions()
    cgp_SGD.display_one_neuron_network()

    cgp_SGDPlus = OneNeuronSGDPlusCGP(
                mu = 0.5,
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    cgp_SGDPlus.parse_expressions()
    cgp_SGDPlus.display_one_neuron_network()

    cgp_Adam = OneNeuronAdamCGP(
                beta_1 = 0.9,
                beta_2 = 0.99,
                epsilon = 1e-8,
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = lr,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    cgp_Adam.parse_expressions()
    cgp_Adam.display_one_neuron_network()

    training_data = cgp_SGD.gen_training_data()
```

```
566     loss_SGD = cgp_SGD.run_training_loop_one_neuron_model(training_data
            ↪ )
567     loss_SGDPlus = cgp_SGDPlus.run_training_loop_one_neuron_model(
            ↪ training_data)
568     loss_Adam = cgp_Adam.run_training_loop_one_neuron_model(
            ↪ training_data)
569
570     plot_learning_curve(loss_SGD, loss_SGDPlus, loss_Adam, 'one_neuron'
            ↪ , str(lr))
571
572 ## run multi-neuron case with different optimizer
573 def run_multi_neuron_diff_optimizer(lr):
574     cgp_SGD = MultiNeuronSGDCGP(
575                 num_layers = 3,
576                 layers_config = [4,2,1], # num of nodes in each layer
577                 expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
578                                'xz=bp*xp+bq*xq+br*xr+bs*xs',
579                                'xo=cp*xw+cq*xz'],
580                 output_vars = ['xo'],
581                 dataset_size = 5000,
582                 learning_rate = lr,
583                 training_iterations = 20000,
584                 batch_size = 8,
585                 display_loss_how_often = 100,
586                 debug = True,
587     )
588     cgp_SGD.parse_multi_layer_expressions()
589     cgp_SGD.display_multi_neuron_network()
590
591     cgp_SGDPlus = MultiNeuronSGDPlusCGP(
592                 mu = 0.5,
593                 num_layers = 3,
594                 layers_config = [4,2,1], # num of nodes in each layer
595                 expressions = ['xw=ap*xp+aq*xq+ar*xr+as*xs',
596                                'xz=bp*xp+bq*xq+br*xr+bs*xs',
597                                'xo=cp*xw+cq*xz'],
598                 output_vars = ['xo'],
599                 dataset_size = 5000,
600                 learning_rate = lr,
601                 training_iterations = 20000,
602                 batch_size = 8,
603                 display_loss_how_often = 100,
604                 debug = True,
605     )
606     cgp_SGDPlus.parse_multi_layer_expressions()
607     cgp_SGDPlus.display_multi_neuron_network()
608
609     cgp_Adam = MultiNeuronAdamCGP(
610                 beta_1 = 0.9,
611                 beta_2 = 0.99,
612                 epsilon = 1e-8,
613                 num_layers = 3,
614                 layers_config = [4,2,1], # num of nodes in each layer
615                 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 = lr,
                training_iterations = 20000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    cgp_Adam.parse_multi_layer_expressions()
    cgp_Adam.display_multi_neuron_network()

    training_data = cgp_SGD.gen_training_data()
    loss_SGD =  cgp_SGD.run_training_loop_multi_neuron_model(
        ↪ training_data)
    loss_SGDPlus = cgp_SGDPlus.run_training_loop_multi_neuron_model(
        ↪ training_data)
    loss_Adam = cgp_Adam.run_training_loop_multi_neuron_model(
        ↪ training_data)

    plot_learning_curve(loss_SGD, loss_SGDPlus, loss_Adam, '
        ↪ multi_neuron', str(lr))

## run one-neuron case with adam using different beta_1 and beta_2
def run_one_neuron_diff_adam(beta_1, beta_2):
    cgp_Adam = OneNeuronAdamCGP(
                beta_1 = beta_1,
                beta_2 = beta_2,
                epsilon = 1e-8,
                one_neuron_model = True,
                expressions = ['xw=ab*xa+bc*xb+cd*xc+ac*xd'],
                output_vars = ['xw'],
                dataset_size = 5000,
                learning_rate = 1e-3,
                training_iterations = 40000,
                batch_size = 8,
                display_loss_how_often = 100,
                debug = True,
    )
    cgp_Adam.parse_expressions()
    cgp_Adam.display_one_neuron_network()
    training_data = cgp_Adam.gen_training_data()
    start_time = time.time()
    loss_Adam = cgp_Adam.run_training_loop_one_neuron_model(
        ↪ training_data)
    time_used = time.time() - start_time
    final_loss = loss_Adam[-1]
    min_loss = min(loss_Adam[1:])
    return time_used, final_loss, min_loss

if __name__ == '__main__':
    # '1' -- one-neuron classifier with different learning rate
    # '2' -- multi-neuron classifier with different learning rate
```

```
    # '3' -- compare different beta_1 and beta_2 in adam with one-
        ↪ neuron case
    parser = argparse.ArgumentParser()
    parser.add_argument('-t', '--task', type=str, default='1',\
        help='choose a task', choices=['1','2','3'])
    args = parser.parse_args()

    seed = 0
    # set_random_seed(seed)

    if args.task == '1':
        for lr in [1e-2, 1e-3, 1e-4]:
            run_one_neuron_diff_optimizer(lr)

    if args.task == '2':
        for lr in [1e-2, 5e-3, 1e-3]:
            run_multi_neuron_diff_optimizer(lr)

    if args.task == '3':
        beta_list = []
        time_list = []
        final_l_list = []
        min_l_list = []
        for beta_1 in [0.8, 0.9, 0.99]:
            for beta_2 in [0.8, 0.9, 0.99]:
                time_used, final_loss, min_loss =
                    ↪ run_one_neuron_diff_adam(beta_1, beta_2)
                beta_list.append((beta_1, beta_2))
                time_list.append(time_used)
                final_l_list.append(final_loss)
                min_l_list.append(min_loss)
        for i in range(len(beta_list)):
            print('Performance with beta_1 %.2f and beta_2 %.2f \n----
                ↪ time: %fs; final loss: %f; minimum loss: %f'%(
                ↪ beta_list[i][0], beta_list[i][1], time_list[i],
                ↪ final_l_list[i], min_l_list[i]))
```