question @87 86 views

Actions

One Neuron Averaging Confusion [Update]

Part 1 [Updated]

In the `run_training_loop_one_neuron_model` method the following variables are calculated and passed to `backprop_and_update_params_one_neuron_model`:

$$\text{y\_error\_avg} = \frac{1}{n}\sum_{i=1}^{n}(\bar{y}_i - y_i)$$

$$\text{deriv\_sigmoid\_avg} = \frac{1}{n}\sum_{i=1}^{n} y_i(1 - y_i)$$

$$\text{data\_tuple\_avg}_j = \frac{1}{n}\sum_{i=1}^{n} x_{ij}$$

Where x_i is training data i = (1…n). x_{ij} is training data i, variable j = (1…m).
$y_i = \sigma((\sum_{j=1}^{m} x_{ij}a_j) + b)$. a_j is a learnable parameter for variable j and b is the learnable bias. And $\bar{y}_i$ is the label for training data x_i.

The loss being $\text{Loss} = \frac{1}{n}\sum_{i=1}^{n} |\bar{y}_i - y_i|^2$.

Its partial derivative should be:

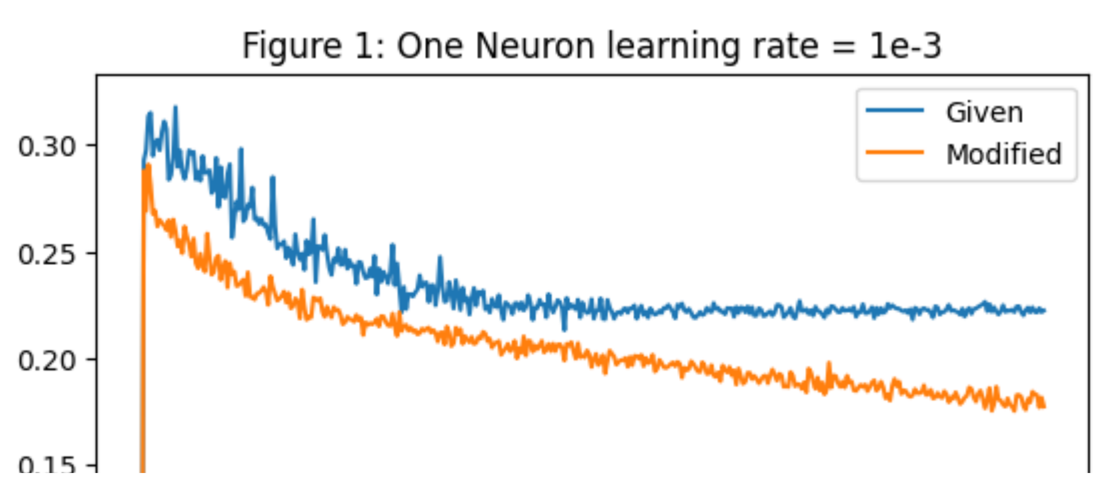
$$\frac{d\text{Loss}}{da_j} = \frac{1}{n}\sum_{i=1}^{n} -2(\bar{y}_i - y_i)y_i(1 - y_i)x_{ij}\ .$$

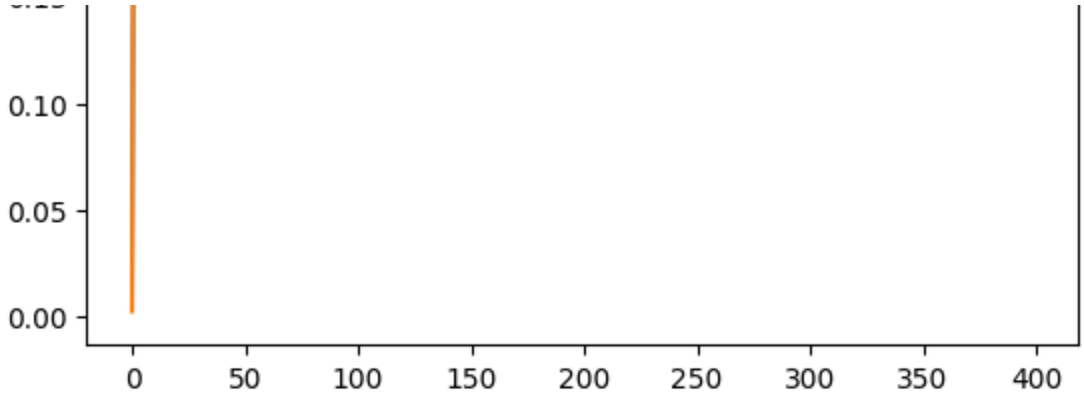
However in the `backprop_and_update_params_one_neuron_model` it seems to calculate it as

$$\frac{d\text{Loss}}{da_j} = -\text{y\_error\_avg} * \text{deriv\_sigmoid\_avg} * \text{data\_tuple\_avg}_j = -\left(\frac{1}{n}\sum_{i=1}^{n}(\bar{y}_i - y_i)\right)\left(\frac{1}{n}\sum_{i=1}^{n} y_i(1 - y_i)\right)$$

Which is not equivalent to the above. Am I missing something?

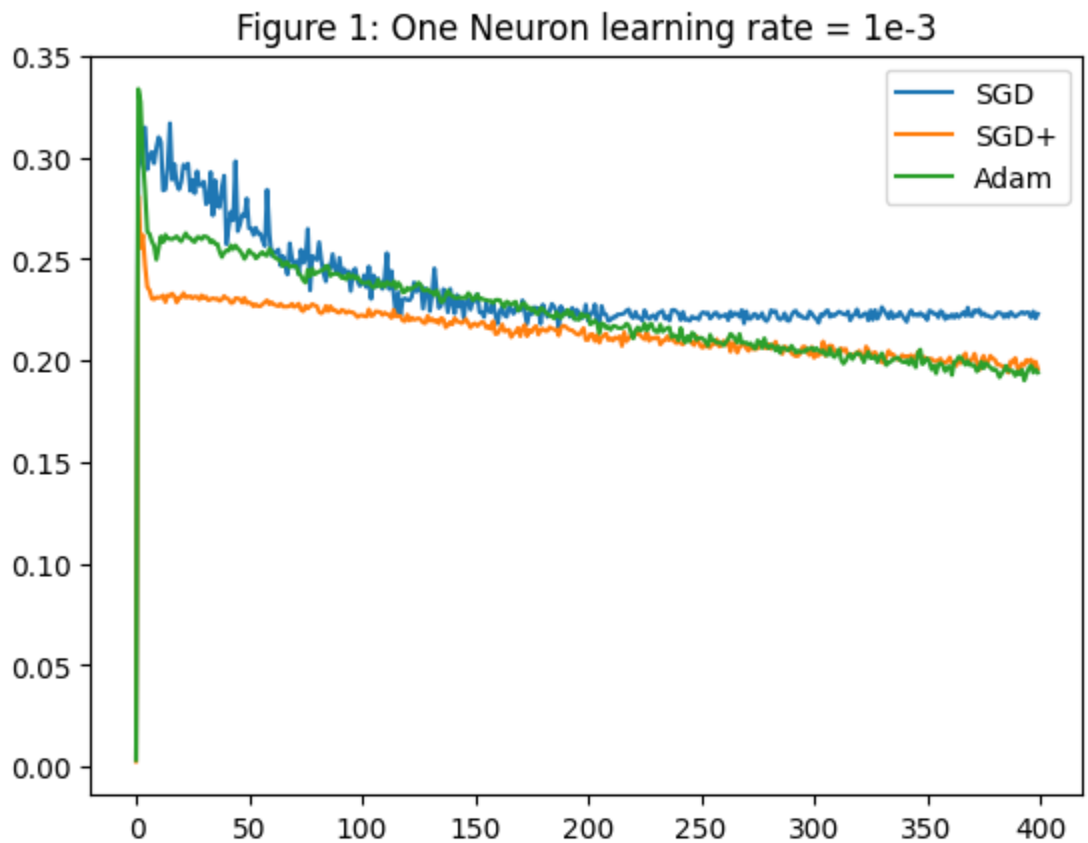
[Update]
Decided to return to this and reimplement the gradient calculation with the "correct" loss gradient. Here are the results. [Note that in the below, this is just SGD, no momentum or adaptive learning rate]



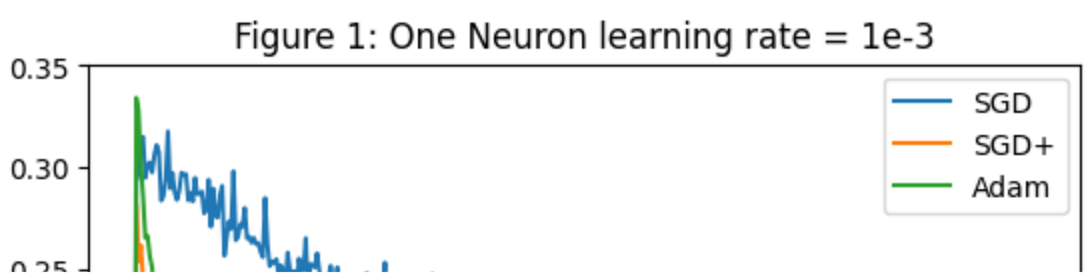


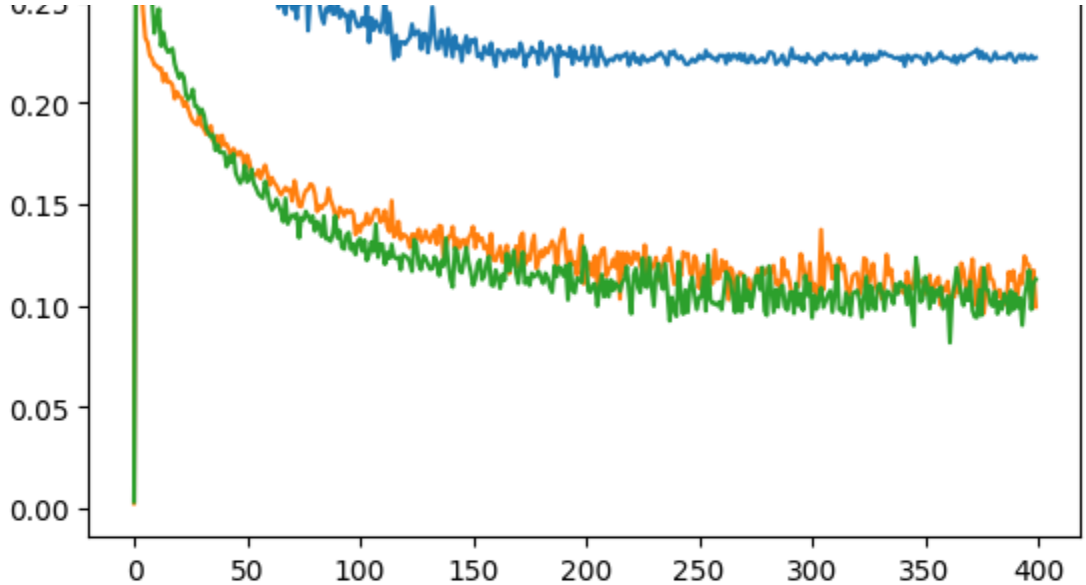
My Adam and SGD+ also perform better when using the modified gradient calculation. [Note that in the below graphs, the SGD plot is from the given gradient calculation, where the “averaging” is done prematurely]

Before:



After:





Here's the code for the modified version [Note: much of the beginning is the same]:

```
class CGP_SGD(ComputationalGraphPrimer):

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

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

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

                             gen_training_data(self)

```

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

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

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

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

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

```
                    batch_data.append(item[0])
                    batch_labels.append(item[1])
                batch_data = [item/maxval for item in batch_data]          ## Norma
lize 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                                   ##  Avera
ge the loss over iterations for printing out
                                                                          ##    ev
ery N iterations during the training loop.
        for i in range(self.training_iterations):
            data = data_loader.getbatch()
            data_tuples = data[0]
            class_labels = data[1]
            y_preds, deriv_sigmoids =  self.forward_prop_one_neuron_model(data_tupl
es)               ##  FORWARD PROP of data
            loss = sum([(abs(class_labels[i] - y_preds[i]))**2 for i in range(len(c
lass_labels))])  ##  Find loss
            loss_avg = loss / float(len(class_labels))
##  Average the loss over batch
            avg_loss_over_iterations += loss_avg
            if i%(self.display_loss_how_often) == 0:
                avg_loss_over_iterations /= self.display_loss_how_often
                loss_running_record.append(avg_loss_over_iterations)
                print("[iter=%d]  loss = %.4f" %  (i+1, avg_loss_over_iterations))
## Display average loss
                avg_loss_over_iterations = 0.0
## Re-initialize avg loss
            y_errors = list(map(operator.sub, class_labels, y_preds))
            y_error_avg = sum(y_errors) / float(len(class_labels))
            deriv_sigmoid_avg = sum(deriv_sigmoids) / float(len(class_labels))
            data_tuple_avg = [sum(x) for x in zip(*data_tuples)]
            data_tuple_avg = list(map(operator.truediv, data_tuple_avg,
                                     [float(len(class_labels))] * len(class_labels)
))
            self.backprop_and_update_params_one_neuron_model(y_errors, data_tuples,
deriv_sigmoids)     ## MOIZ - CHANGED
        # plt.figure()
        # plt.plot(loss_running_record)
        # plt.show()

    def backprop_and_update_params_one_neuron_model(self, y_error, vals_for_input_v
ars, deriv_sigmoid):
```

```
        gamma = .9

        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_input_vars = [x for x in zip(*vals_for_input_vars)]
## MOIZ - ADDED
        vals_for_input_vars_dict =  dict(zip(input_vars, list(vals_for_input_var
s)))
        vals_for_learnable_params = self.vals_for_learnable_params
        for i,param in enumerate(self.vals_for_learnable_params):
            ## Calculate the next step in the parameter hyperplane
            grad = 0
            input_vals = vals_for_input_vars_dict[param_to_vars_map[param]]
            for k in range(len(y_error)):
## MOIZ - GRAD CALC
                error = y_error[k]
                val = input_vals[k]
                dsig = deriv_sigmoid[k]
                grad += -2 * error * dsig * val
            grad /= float(len(y_error))
            self.vals_for_learnable_params[param] -= self.learning_rate * grad

        grad = 0
        for k in range(len(y_error)):
## MOIZ - GRAD CALC
            error = y_error[k]
            dsig = deriv_sigmoid[k]
            grad += -2 * error * dsig
        grad /= float(len(y_error))
        self.bias -= self.learning_rate * grad
```

run code snippet

Part 2

~~Also I should note that the original backpropagate method adds the step instead of subtracting, so it would be going in the direction of the gradient, towards increasing value.~~ **[this part solved]**

hw3

~ An instructor (Fangda Li) endorsed this question ~

Edit good question 2 Updated 4 months ago by Moiz Rasheed

S **the students' answer,** *where students collectively construct a single answer*

Actions

If you look back at the slides where the partial derivative of the loss is calculated, there is a factor of -2 in the expression. In the code, this factor is dropped. So instead of subtracting, you must add the step in the update. If you add the -2 back into your calculation of the gradient, then you need to subtract the step in your update.

Edit undo thanks | 1 Updated 4 months ago by Anonymous Scale

i **the instructors' answer,** *where instructors collectively construct a single answer*

You are correct -- the averaging in the calculation of partial gradient should only take place once. Prof. Kak has acknowledged this issue, however, at this time being, he will update his module in a future time. Thank you for pointing this out.

undo thanks | 1 Updated 4 months ago by Fangda Li

followup discussions, *for lingering questions and comments*

@87_f1

Actions

Moiz Rasheed 4 months ago

Thanks, missed the negative in the derivative calculation. Still don't understand why the averaging is okay though.

helpful! | 0

Reply to this followup discussion

Start a new followup discussion

Compose a new followup discussion