BME646 and ECE60146: Homework 3

# Mykhailo Tsysin

# Feb 5, 2023

In this homework, we will look more closely into how different modifications for stochastic gradient descent can improve deep neural network learning. For a full .py solution, please look at the corresponding file.

# Task 2.2

Outputs form two scripts:

one\_neuron\_classifier.py:

Diagram

Description automatically generated Chart

Description automatically generated

multi\_neuron\_classifier.py:

Diagram

Description automatically generated Chart

Description automatically generated

# Task 2.3-2.4

verify\_with\_torchnn.py

Multi neuron:

Chart, scatter chart

Description automatically generated

One neuron:

Chart

Description automatically generated

In both cases, we see faster convergence due to optimizations included in the torch.nn

# Task 2.5

In this part, we’re going to implement two popular optimizations to stochastic gradient descent. These are SGD+ and ADAM optimizers.

1. SGD+ is also known as a gradient descent with momentum. The main idea is to keep information about previous steps and make a next step based on both the current gradient and the stored information. The idea is that by averaging over multiple gradients, we can stay on the correct path and not get stuck in “elongated valleys” for too long. The mathematical form of this idea is presented below:
2. For ADAM optimization we go even further by taking advantage of both RMSprop and gradient descent with momentum. RMSprop keeps a weighted average of squares of recent steps and is a good indicator of whether we need to speed up or slow down learning in a particular direction. Mathematically:

For implementation, I changed the backpropagation function to include previous step memorization. The new modified class is a subclass of the original primer with 2 functions modified: Main training loop (only to initialize gradient storage) and backprop to implement the step size optimization logic. The \_\_init\_\_ method is also modified to allow for selection between different optimizations. The final code looks like this:

seed = 0

random.seed(seed)

np.random.seed(seed)

# torch.manual\_seed(seed)

# os.environ['PYTHONHASHSEED'] = str(seed)

class ComputationalGraphPrimerModified(ComputationalGraphPrimer):

def \_\_init\_\_(self, \*args, \*\*kwargs):

"""Initialize the new class instance in the same way as the base class"""

if 'modification' not in kwargs:

warnings.warn('You are using modified CGP without specifying the modification type. \

Proceed with caution, defaults to "no". \

Supported modifications: "no", "plus", "adam"')

self.modification = "no"

else:

self.modification = kwargs.pop('modification')

if self.modification == "plus":

if 'mu' not in kwargs:

raise ValueError('You are trying to use "plus" modification without "mu" argument, terminate execution')

else:

self.mu = kwargs.pop('mu')

if self.modification == "adam":

if 'beta1' not in kwargs or 'beta2' not in kwargs or 'epsilon' not in kwargs:

raise ValueError('You are trying to use "adam" modification without one of the required arguments, terminate execution')

else:

self.beta1 = kwargs.pop('beta1')

self.beta2 = kwargs.pop('beta2')

self.epsilon = kwargs.pop('epsilon')

# Parse the rest of the input

super().\_\_init\_\_(\*args, \*\*kwargs)

######################################################################################################

######################################### one neuron model ###########################################

######################################################################################################

def run\_training\_loop\_one\_neuron\_model(self, training\_data):

"""

The training loop must first initialize the learnable parameters. Remember, these are the

symbolic names in your input expressions for the neural layer that do not begin with the

letter 'x'. In this case, we are initializing with random numbers from a uniform distribution

over the interval (0,1).

UPDATE: We want to override this function to be able to support SGD+ (with momentum). The original code

form the primer is copied fully here for completeness. The things added in this homework are

designated with the following comment line:

<<<<<<<<<<<<<<<<<<<<<< NEW FOR HW3

"""

self.vals\_for\_learnable\_params = {param: random.uniform(0,1) for param in self.learnable\_params}

self.bias = random.uniform(0,1) ## Adding the bias improves class discrimination.

## We initialize it to a random number.

class DataLoader:

"""

To understand the logic of the dataloader, it would help if you first understand how

the training dataset is created. Search for the following function in this file:

gen\_training\_data(self)

As you will see in the implementation code for this method, the training dataset

consists of a Python dict with two keys, 0 and 1, the former points to a list of

all Class 0 samples and the latter to a list of all Class 1 samples. In each list,

the data samples are drawn from a multi-dimensional Gaussian distribution. The two

classes have different means and variances. The dimensionality of each data sample

is set by the number of nodes in the input layer of the neural network.

The data loader's job is to construct a batch of samples drawn randomly from the two

lists mentioned above. And it mush also associate the class label with each sample

separately.

"""

def \_\_init\_\_(self, training\_data, batch\_size):

self.training\_data = training\_data

self.batch\_size = batch\_size

self.class\_0\_samples = [(item, 0) for item in self.training\_data[0]] ## Associate label 0 with each sample

self.class\_1\_samples = [(item, 1) for item in self.training\_data[1]] ## Associate label 1 with each sample

def \_\_len\_\_(self):

return len(self.training\_data[0]) + len(self.training\_data[1])

def \_getitem(self):

cointoss = random.choice([0,1]) ## When a batch is created by getbatch(), we want the

## samples to be chosen randomly from the two lists

if cointoss == 0:

return random.choice(self.class\_0\_samples)

else:

return random.choice(self.class\_1\_samples)

def getbatch(self):

batch\_data,batch\_labels = [],[] ## First list for samples, the second for labels

maxval = 0.0 ## For approximate batch data normalization

for \_ in range(self.batch\_size):

item = self.\_getitem()

if np.max(item[0]) > maxval:

maxval = np.max(item[0])

batch\_data.append(item[0])

batch\_labels.append(item[1])

batch\_data = [item/maxval for item in batch\_data] ## Normalize batch data

batch = [batch\_data, batch\_labels]

return batch

##########################################################################################################################

# We need to create a storage for previous steps if we use modification <<<<<<<<<<<<<<<<<<<<<< NEW FOR HW3

if self.modification == "plus":

self.param\_gradients = {param: 0 for param in self.vals\_for\_learnable\_params}

self.bias\_gradient = 0

if self.modification == "adam":

# Need to add values for mk, vk

self.m = {param: 0 for param in self.vals\_for\_learnable\_params}

self.v = {param: 0 for param in self.vals\_for\_learnable\_params}

self.beta1\_pow\_k = 1 # Values of betas to the power of k

self.beta2\_pow\_k = 1

self.m\_bias = 0

self.v\_bias = 0

##########################################################################################################################

data\_loader = DataLoader(training\_data, batch\_size=self.batch\_size)

loss\_running\_record = []

i = 0

avg\_loss\_over\_iterations = 0.0 ## Average the loss over iterations for printing out

## every N iterations during the training loop.

for i in range(self.training\_iterations):

data = data\_loader.getbatch()

data\_tuples = data[0]

class\_labels = data[1]

y\_preds, deriv\_sigmoids = self.forward\_prop\_one\_neuron\_model(data\_tuples) ## FORWARD PROP of data

loss = sum([(abs(class\_labels[i] - y\_preds[i]))\*\*2 for i in range(len(class\_labels))]) ## Find loss

loss\_avg = loss / float(len(class\_labels)) ## Average the loss over 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\_error\_avg, data\_tuple\_avg, deriv\_sigmoid\_avg) ## BACKPROP loss

# plt.figure()

# plt.plot(loss\_running\_record)

# plt.show()

return loss\_running\_record

def backprop\_and\_update\_params\_one\_neuron\_model(self, y\_error, vals\_for\_input\_vars, deriv\_sigmoid):

"""

As should be evident from the syntax used in the following call to backprop function,

self.backprop\_and\_update\_params\_one\_neuron\_model( y\_error\_avg, data\_tuple\_avg, deriv\_sigmoid\_avg)

^^^ ^^^ ^^^

the values fed to the backprop function for its three arguments are averaged over the training

samples in the batch. This in keeping with the spirit of SGD that calls for averaging the

information retained in the forward propagation over the samples in a batch.

See Slide 59 of my Week 3 slides for the math of back propagation for the One-Neuron network.

"""

input\_vars = self.independent\_vars

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\_dict = dict(zip(input\_vars, list(vals\_for\_input\_vars)))

##########################################################################################################################

# vals\_for\_learnable\_params = self.vals\_for\_learnable\_params <<<<<<<<<<<<<<<<<<<<<< NEW FOR HW3

if self.modification == "no": # Keep original code

for i, param in enumerate(self.vals\_for\_learnable\_params):

## Calculate the next step in the parameter hyperplane

# step = self.learning\_rate \* y\_error \* vals\_for\_input\_vars\_dict[input\_vars[i]] \* deriv\_sigmoid

step = self.learning\_rate \* y\_error \* vals\_for\_input\_vars\_dict[param\_to\_vars\_map[param]] \* deriv\_sigmoid

## Update the learnable parameters

self.vals\_for\_learnable\_params[param] += step

self.bias += self.learning\_rate \* y\_error \* deriv\_sigmoid ## Update the bias

elif self.modification == "plus": # Logic for "plus" modification

for param in self.vals\_for\_learnable\_params:

# Calculate new grad from old grads

gradient = (y\_error \* vals\_for\_input\_vars\_dict[param\_to\_vars\_map[param]] \* deriv\_sigmoid + # New part (essentially negative gradient)

self.mu \* self.param\_gradients[param]) # Old part

# Update the learnable parameters

self.vals\_for\_learnable\_params[param] += self.learning\_rate \* gradient

# Save for future calculations

self.param\_gradients[param] = gradient

gradient = y\_error \* deriv\_sigmoid + \

self.mu \* self.bias\_gradient # Negative gradient for bias

# Update bias term

self.bias += self.learning\_rate \* gradient ## Update the bias

self.bias\_gradient = gradient # Update bias gradient.

elif self.modification == "adam":

self.beta1\_pow\_k \*= self.beta1 # Update powers of betas

self.beta2\_pow\_k \*= self.beta2

for param in self.vals\_for\_learnable\_params:

# Calculate new grad from old grads

m = ((1 - self.beta1) \* y\_error \* vals\_for\_input\_vars\_dict[param\_to\_vars\_map[param]] \* deriv\_sigmoid + # New part (essentially negative gradient)

self.beta1 \* self.m[param])

v = ((1 - self.beta2) \* (y\_error \* vals\_for\_input\_vars\_dict[param\_to\_vars\_map[param]] \* deriv\_sigmoid) \*\* 2 + # New part (essentially negative gradient)

self.beta2 \* self.v[param]) # Old part

m\_hat = m / (1 - self.beta1\_pow\_k)

v\_hat = v / (1 - self.beta2\_pow\_k)

# Update the learnable parameters

self.vals\_for\_learnable\_params[param] += self.learning\_rate \* m\_hat / math.sqrt(v\_hat + self.epsilon)

# Save for future calculations

self.m[param] = m

self.v[param] = v

m\_bias = ((1 - self.beta1) \* y\_error \* deriv\_sigmoid +

self.beta1 \* self.m\_bias) # Negative gradient for bias

v\_bias = ((1 - self.beta2) \* (y\_error \* deriv\_sigmoid)\*\*2 +

self.beta2 \* self.v\_bias) # Negative gradient for bia

m\_bias\_hat = m\_bias / (1 - self.beta1\_pow\_k)

v\_bias\_hat = v\_bias / (1 - self.beta2\_pow\_k)

# Update bias term

self.bias += self.learning\_rate \* m\_bias\_hat / math.sqrt(v\_bias\_hat + self.epsilon) ## Update the bias

self.m\_bias = m\_bias

self.v\_bias = v\_bias

else:

raise ValueError("Wrong modification supplied")

######################################################################################################

The new parts are highlighted. For single neuron model, we used the following parameters:

beta1 = 0.9, beta2 = 0.99, epsilon = 0.00000001, mu = 0.95, lr = 5\*10-2

We can afford a big learning rate since our model is not very sophisticated.

The result for a single neuron is as follows:

Chart, histogram

Description automatically generated

We see that both ADAM and SGD+ perform a lot better than simple SGD. ADAM converges slightly faster, but since they both arrive at the minimum quite early, the difference is not too noticeable.

A more significant difference can be seen with lr = 0.01:

Chart, histogram

Description automatically generated

For multi-layer NN, we did similar modifications:

######################################################################################################

######################################## multi neuron model ##########################################

######################################################################################################

def run\_training\_loop\_multi\_neuron\_model(self, training\_data):

class DataLoader:

"""

To understand the logic of the data loader, it would help if you first understand how

the training dataset is created. Search for the following function in this file:

gen\_training\_data(self)

As you will see in the implementation code for this method, the training dataset

consists of a Python dict with two keys, 0 and 1, the former points to a list of

all Class 0 samples and the latter to a list of all Class 1 samples. In each list,

the data samples are drawn from a multi-dimensional Gaussian distribution. The two

classes have different means and variances. The dimensionality of each data sample

is set by the number of nodes in the input layer of the neural network.

The data loader's job is to construct a batch of samples drawn randomly from the two

lists mentioned above. And it mush also associate the class label with each sample

separately.

"""

def \_\_init\_\_(self, training\_data, batch\_size):

self.training\_data = training\_data

self.batch\_size = batch\_size

self.class\_0\_samples = [(item, 0) for item in self.training\_data[0]] ## Associate label 0 with each sample

self.class\_1\_samples = [(item, 1) for item in self.training\_data[1]] ## Associate label 1 with each sample

def \_\_len\_\_(self):

return len(self.training\_data[0]) + len(self.training\_data[1])

def \_getitem(self):

cointoss = random.choice([0,1]) ## When a batch is created by getbatch(), we want the

## samples to be chosen randomly from the two lists

if cointoss == 0:

return random.choice(self.class\_0\_samples)

else:

return random.choice(self.class\_1\_samples)

def getbatch(self):

batch\_data,batch\_labels = [],[] ## First list for samples, the second for labels

maxval = 0.0 ## For approximate batch data normalization

for \_ in range(self.batch\_size):

item = self.\_getitem()

if np.max(item[0]) > maxval:

maxval = np.max(item[0])

batch\_data.append(item[0])

batch\_labels.append(item[1])

batch\_data = [item/maxval for item in batch\_data] ## Normalize batch data

batch = [batch\_data, batch\_labels]

return batch

"""

The training loop must first initialize the learnable parameters. Remember, these are the

symbolic names in your input expressions for the neural layer that do not begin with the

letter 'x'. In this case, we are initializing with random numbers from a uniform distribution

over the interval (0,1).

"""

self.vals\_for\_learnable\_params = {param: random.uniform(0,1) for param in self.learnable\_params}

self.bias = [random.uniform(0,1) for \_ in range(self.num\_layers-1)] ## Adding the bias to each layer improves

## class discrimination. We initialize it

## to a random number.

##########################################################################################################################

# We need to create a storage for previous steps if we use modification <<<<<<<<<<<<<<<<<<<<<< NEW FOR HW3

if self.modification == "plus":

self.param\_gradients = {param: 0 for param in self.vals\_for\_learnable\_params}

self.bias\_gradients = [0 for \_ in range(self.num\_layers-1)]

if self.modification == "adam":

# Need to add values for mk, vk,

self.m = {param: 0 for param in self.vals\_for\_learnable\_params}

self.v = {param: 0 for param in self.vals\_for\_learnable\_params}

self.beta1\_pow\_k = 1 # Values of betas to the power of k

self.beta2\_pow\_k = 1

self.m\_bias = [0 for \_ in range(self.num\_layers-1)]

self.v\_bias = [0 for \_ in range(self.num\_layers-1)]

##########################################################################################################################

data\_loader = DataLoader(training\_data, batch\_size=self.batch\_size)

loss\_running\_record = []

i = 0

avg\_loss\_over\_iterations = 0.0 ## Average the loss over iterations for printing out

## every N iterations during the training loop.

for i in range(self.training\_iterations):

data = data\_loader.getbatch()

data\_tuples = data[0]

class\_labels = data[1]

self.forward\_prop\_multi\_neuron\_model(data\_tuples) ## FORW PROP works by side-effect

predicted\_labels\_for\_batch = self.forw\_prop\_vals\_at\_layers[self.num\_layers-1] ## Predictions from FORW PROP

y\_preds = [item for sublist in predicted\_labels\_for\_batch for item in sublist] ## Get numeric vals for predictions

loss = sum([(abs(class\_labels[i] - y\_preds[i]))\*\*2 for i in range(len(class\_labels))]) ## Calculate loss for batch

loss\_avg = loss / float(len(class\_labels)) ## Average the loss over batch

avg\_loss\_over\_iterations += loss\_avg ## Add to Average loss over iterations

if i%(self.display\_loss\_how\_often) == 0:

avg\_loss\_over\_iterations /= self.display\_loss\_how\_often

loss\_running\_record.append(avg\_loss\_over\_iterations)

print("[iter=%d] loss = %.4f" % (i+1, avg\_loss\_over\_iterations)) ## Display avg loss

avg\_loss\_over\_iterations = 0.0 ## Re-initialize avg-over-iterations loss

y\_errors = list(map(operator.sub, class\_labels, y\_preds))

y\_error\_avg = sum(y\_errors) / float(len(class\_labels))

self.backprop\_and\_update\_params\_multi\_neuron\_model(y\_error\_avg, class\_labels) ## BACKPROP loss

# plt.figure()

# plt.plot(loss\_running\_record)

# plt.show()

return loss\_running\_record

def backprop\_and\_update\_params\_multi\_neuron\_model(self, y\_error, class\_labels):

"""

First note that loop index variable 'back\_layer\_index' starts with the index of

the last layer. For the 3-layer example shown for 'forward', back\_layer\_index

starts with a value of 2, its next value is 1, and that's it.

Stochastic Gradient Gradient calls for the backpropagated loss to be averaged over

the samples in a batch. To explain how this averaging is carried out by the

backprop function, consider the last node on the example shown in the forward()

function above. Standing at the node, we look at the 'input' values stored in the

variable "input\_vals". Assuming a batch size of 8, this will be list of

lists. Each of the inner lists will have two values for the two nodes in the

hidden layer. And there will be 8 of these for the 8 elements of the batch. We average

these values 'input vals' and store those in the variable "input\_vals\_avg". Next we

must carry out the same batch-based averaging for the partial derivatives stored in the

variable "deriv\_sigmoid".

Pay attention to the variable 'vars\_in\_layer'. These store the node variables in

the current layer during backpropagation. Since back\_layer\_index starts with a

value of 2, the variable 'vars\_in\_layer' will have just the single node for the

example shown for forward(). With respect to what is stored in vars\_in\_layer', the

variables stored in 'input\_vars\_to\_layer' correspond to the input layer with

respect to the current layer.

"""

# backproped prediction error:

pred\_err\_backproped\_at\_layers = {i : [] for i in range(1,self.num\_layers-1)}

pred\_err\_backproped\_at\_layers[self.num\_layers-1] = [y\_error]

for back\_layer\_index in reversed(range(1,self.num\_layers)):

input\_vals = self.forw\_prop\_vals\_at\_layers[back\_layer\_index -1]

input\_vals\_avg = [sum(x) for x in zip(\*input\_vals)]

input\_vals\_avg = list(map(operator.truediv, input\_vals\_avg, [float(len(class\_labels))] \* len(class\_labels)))

deriv\_sigmoid = self.gradient\_vals\_for\_layers[back\_layer\_index]

deriv\_sigmoid\_avg = [sum(x) for x in zip(\*deriv\_sigmoid)]

deriv\_sigmoid\_avg = list(map(operator.truediv, deriv\_sigmoid\_avg,

[float(len(class\_labels))] \* len(class\_labels)))

vars\_in\_layer = self.layer\_vars[back\_layer\_index] ## a list like ['xo']

vars\_in\_next\_layer\_back = self.layer\_vars[back\_layer\_index - 1] ## a list like ['xw', 'xz']

layer\_params = self.layer\_params[back\_layer\_index]

## note that layer\_params are stored in a dict like

## {1: [['ap', 'aq', 'ar', 'as'], ['bp', 'bq', 'br', 'bs']], 2: [['cp', 'cq']]}

## "layer\_params[idx]" is a list of lists for the link weights in layer whose output nodes are in layer "idx"

transposed\_layer\_params = list(zip(\*layer\_params)) ## creating a transpose of the link matrix

backproped\_error = [None] \* len(vars\_in\_next\_layer\_back)

for k,varr in enumerate(vars\_in\_next\_layer\_back):

for j,var2 in enumerate(vars\_in\_layer):

backproped\_error[k] = sum([self.vals\_for\_learnable\_params[transposed\_layer\_params[k][i]] \*

pred\_err\_backproped\_at\_layers[back\_layer\_index][i]

for i in range(len(vars\_in\_layer))])

# deriv\_sigmoid\_avg[i] for i in range(len(vars\_in\_layer))])

pred\_err\_backproped\_at\_layers[back\_layer\_index - 1] = backproped\_error

# vals\_for\_learnable\_params = self.vals\_for\_learnable\_params <<<<<<<<<<<<<<<<<<<<<< NEW FOR HW3

if self.modification == "no": # Keep original code

for j,var in enumerate(vars\_in\_layer):

layer\_params = self.layer\_params[back\_layer\_index][j]

for i,param in enumerate(layer\_params):

gradient\_of\_loss\_for\_param = input\_vals\_avg[i] \* pred\_err\_backproped\_at\_layers[back\_layer\_index][j]

step = self.learning\_rate \* gradient\_of\_loss\_for\_param \* deriv\_sigmoid\_avg[j]

self.vals\_for\_learnable\_params[param] += step

self.bias[back\_layer\_index-1] += self.learning\_rate \* sum(pred\_err\_backproped\_at\_layers[back\_layer\_index]) \

\* sum(deriv\_sigmoid\_avg)/len(deriv\_sigmoid\_avg)

elif self.modification == "plus": # Logic for "plus" modification

for j,var in enumerate(vars\_in\_layer):

layer\_params = self.layer\_params[back\_layer\_index][j]

for i,param in enumerate(layer\_params):

gradient\_of\_loss\_for\_param = input\_vals\_avg[i] \* pred\_err\_backproped\_at\_layers[back\_layer\_index][j] \* deriv\_sigmoid\_avg[j] \

+ self.mu \* self.param\_gradients[param]

self.vals\_for\_learnable\_params[param] += self.learning\_rate \* gradient\_of\_loss\_for\_param

self.param\_gradients[param] = gradient\_of\_loss\_for\_param

gradient\_of\_loss\_for\_bias = sum(pred\_err\_backproped\_at\_layers[back\_layer\_index]) \* sum(deriv\_sigmoid\_avg)/len(deriv\_sigmoid\_avg) + \

self.mu \* self.bias\_gradients[back\_layer\_index-1]

self.bias[back\_layer\_index-1] += self.learning\_rate \* gradient\_of\_loss\_for\_bias

self.bias\_gradients[back\_layer\_index-1] = gradient\_of\_loss\_for\_bias

elif self.modification == "adam":

self.beta1\_pow\_k \*= self.beta1 # Update powers of betas

self.beta2\_pow\_k \*= self.beta2

for j,var in enumerate(vars\_in\_layer):

layer\_params = self.layer\_params[back\_layer\_index][j]

for i,param in enumerate(layer\_params):

m = (1 - self.beta1) \* input\_vals\_avg[i] \* pred\_err\_backproped\_at\_layers[back\_layer\_index][j] \* deriv\_sigmoid\_avg[j] \

+ self.beta1 \* self.m[param]

v = (1 - self.beta2) \* (input\_vals\_avg[i] \* pred\_err\_backproped\_at\_layers[back\_layer\_index][j] \* deriv\_sigmoid\_avg[j])\*\*2 \

+ self.beta2 \* self.v[param]

m\_hat = m / (1 - self.beta1\_pow\_k)

v\_hat = v / (1 - self.beta2\_pow\_k)

self.vals\_for\_learnable\_params[param] += self.learning\_rate \* m\_hat / math.sqrt(v\_hat + self.epsilon)

self.m[param] = m

self.v[param] = v

m\_bias = (1 - self.beta1) \* sum(pred\_err\_backproped\_at\_layers[back\_layer\_index]) \* sum(deriv\_sigmoid\_avg)/len(deriv\_sigmoid\_avg) \

+ self.beta1 \* self.m\_bias[back\_layer\_index-1]

v\_bias = (1 - self.beta2) \* (sum(pred\_err\_backproped\_at\_layers[back\_layer\_index]) \* sum(deriv\_sigmoid\_avg)/len(deriv\_sigmoid\_avg))\*\*2 \

+ self.beta2 \* self.v\_bias[back\_layer\_index-1]

m\_bias\_hat = m\_bias / (1 - self.beta1\_pow\_k)

v\_bias\_hat = v\_bias / (1 - self.beta2\_pow\_k)

self.bias[back\_layer\_index-1] += self.learning\_rate \* m\_bias\_hat / math.sqrt(v\_bias\_hat + self.epsilon)

self.m\_bias[back\_layer\_index-1] = m\_bias

self.v\_bias[back\_layer\_index-1] = v\_bias

else:

raise ValueError("Wrong modification supplied")

######################################################################################################

The parameters used are:

beta1 = 0.9, beta2 = 0.99, epsilon = 0.00000001, mu = 0.95, lr = 5\*10-3

The resulting graph is :

Chart, line chart, histogram

Description automatically generated

For lr = 10-2 The optimizations also exhibit superior performance:Chart, histogram

Description automatically generated

Both ADAM and SGD+ outperform simple SGD by a large margin. ADAM converges faster, although both arrive at the same minimum.

Note that in the equation for both SGD and ADAM we store old values of modified gradient which are not multiplied by the learning rate. Thus, to achieve the same result, we need to multiply the resulting new modified gradient (weighted sum of the old modified gradient and a new raw gradient) by the learning rate to get a step size.

We also see that for both one neuron and multi-neuron the optimized step size was able to achieve the suggested target of 0.16.