```
#######
#######   sgLearnNil4.ipynb
#######   a notebook for machine learning proofs for classification of nilpotent
#######   carlos.simpson@univ-cotedazur.fr
#######   https://github.com/carlostsimpson/sg-learn
#######   distributed under: GNU General Public License v3.0
#######
```

```
import numpy as np
import matplotlib.pyplot as plt
import random
import time
import copy
import math
import datetime

import gc

class CoherenceError(Exception):
    pass
```

```
from __future__ import print_function
import torch

import torch.nn as nn
import torch.nn.functional as F

from torch import optim

```

```
# try to put everything on gpu if it exists

if torch.cuda.is_available():
    Dvc = torch.device("cuda:0")
    print("Running on GPU cuda:0")
else:
    Dvc = torch.device("cpu")
    print("Running on CPU")

CpuDvc = torch.device("cpu")

```

```
#Dvc = torch.device("cpu")  # in case you need to force it to cpu (e.g. cuda com
```

```
###
```

```
def zbinary(depth,z):  # the smallest digits are first here (i.e. reads backward
    binnum = [int(i) for i in bin(z)[2:]]
    bnlength = len(binnum)
    if bnlength > depth:
```

```
    if bnlength > depth:
        print("warning: applying zbinary to",z,"with depth",depth,"but bin lengt
    outputarray = torch.zeros((depth),dtype = torch.bool,device=Dvc)
    for j in range(depth):
        if j < bnlength:
            outputarray[j] = (binnum[bnlength - j-1] == 1)
        else:
            outputarray[j] = False
    return outputarray

def binaryz(depth,binarray):
    thez = 0
    for i in range(depth):
        thez += binarray.to(torch.int)[i] * 2**i
    return thez

def binaryzbatch(length,depth,binarray_batch):
    zbatch = torch.zeros((length),dtype = torch.int64,device=Dvc)
    for i in range(depth):
        zbatch += binarray_batch.to(torch.int)[:,i] * (2**i)
    return zbatch

def composepermutations(vector1,vector2):
        vector2i64 = vector2.to(torch.int64)
        composition = vector1[vector2i64]
        return composition

def composedetections(length,detection1,detection2):  # outputs the result of de
    output = torch.zeros((length),dtype = torch.bool,device=Dvc)
    output[detection1] = detection2
    return output

def memReport(style):  # by QuantScientist Solomon K @smth
    if style == 'memory':
        print("gc memory report")
        for obj in gc.get_objects():
            if torch.is_tensor(obj):
                print(type(obj), obj.size())
    if style == 'mg':
        print("gc memory and garbage report::",end=' ')
        allobjects = gc.get_objects()
        all_length = len(allobjects)
        elements = torch.zeros((all_length),dtype = torch.int64,device=Dvc)
        count = 0
        loc = 0
        for obj in allobjects:
            if torch.is_tensor(obj):
                count += 1
                elements[loc]= torch.numel(obj)
            loc += 1
        elcount = elements.sum(0)
        print("there are",count,"torch tensors in play with",itp(elcount),"eleme
        #
        values,indices = torch.sort(elements,descending = True)
        upper = 5
        if upper > count:
```

```
            upper = count
        for i in range(upper):
            indi = indices[i]
            obji = allobjects[i]
            print(obji.size())
        print("|||")
        for obj in gc.garbage:
            if torch.is_tensor(obj):
                print(type(obj), obj.size())
    return
```

```
def arangeic(x):
    ar = torch.arange(x,dtype = torch.int64,device = Dvc)
    return ar

def itp(x):  # integer to print
    return nump(x)

def itt(x):  # integer to torch
    if torch.is_tensor(x):
      return x
    else:
      return torch.tensor(x,device = Dvc)

def itf(x):  # integer to torch.float
    return itt(x).to(torch.float)

torch_pi = torch.acos(torch.zeros(1)).item() * 2

def tdetach(x):
    if torch.is_tensor(x):
        return x.detach()
    else:
        return x

def nump(x):
    if torch.is_tensor(x):
        return x.detach().to(CpuDvc).numpy()
    else:
        return x

def numpr(x,k):
    return np.round(nump(x),k)

def numpi(x):
    return nump(x.to(torch.int))
```

```
###
```

```
class Historical :
    def __init__(self,hlength_max):
        self.hlength_max = hlength_max
        self.hwidth = 20
        self.hfwidth = 10
        self.prwidth = 20
        #
        self.hlength = 0
        #
        self.histi = torch.zeros((self.hlength_max,self.hwidth),dtype = torch.in
        self.histf = torch.zeros((self.hlength_max,self.hfwidth),dtype = torch.f
        #
        self.proofrecord = torch.zeros((self.hlength_max,self.prwidth),dtype = t
        self.prcursor = 0
        self.local_tweak_cursor = 0
        self.global_tweak_cursor = 0
        #
        self.current_proof_valency_frequency = torch.zeros((10),dtype = torch.in
        self.current_proof_impossible_count = 0
        self.current_proof_done_count = 0
        self.current_proof_passive_count = 0
        self.current_proof_benchmark = 0
        #
        self.title_text_sigma_proof = None
        self.title_text_sigma_train = None
        #
        self.training_counter = 0
        #
        self.proof_nodes_max = 200000
        #
        self.D = {
            'Global':0,
            'Local':1,
            'LocalCE':2,
            'Benchmark':3,
            'Regular':4,
            'Adaptive':5,
            'Uniform':6,
            'Parameters':7,
            'Driver':8,
            'Model':9,
            'Loss':10,
            'Training':11,
            'BenchmarkProof':12,
            'FullProof':13,
            'DropoutProof':14,
        }



    def reset_current_proof(self):
        self.current_proof_valency_frequency[:] = 0
        self.current_proof_impossible_count = 0
        self.current_proof_done_count = 0
```

```
        self.current_proof_passive_count = 0
        self.current_proof_benchmark = 0
        return


    def record_current_proof(self,benchmark = False):
        if self.prcursor >= self.hlength_max:
            print("proof data recording overflow")
            return
        self.proofrecord[self.prcursor,0] = self.current_proof_impossible_count
        self.proofrecord[self.prcursor,1] = self.current_proof_done_count
        self.proofrecord[self.prcursor,2] = self.current_proof_passive_count
        self.proofrecord[self.prcursor,3:13] = self.current_proof_valency_freque
        self.proofrecord[self.prcursor,14] = 0
        if benchmark:
            self.proofrecord[self.prcursor,14] = 1
            print("recorded benchmark proof",end=' ')
        else:
            print("recorded full proof",end=' ')
        self.print_proof_recordi(self.prcursor,Pp)
        self.prcursor += 1
        return

    def print_proof_recordi(self,i,Pp):
        #
        upperval = Pp.beta + 2
        if upperval > 10:
            upperval = 10
        impcount = itp(self.proofrecord[i,0])
        donecount = itp(self.proofrecord[i,1])
        passivecount = itp(self.proofrecord[i,2])
        valency = nump(self.proofrecord[i,3:3 + upperval])
        #
        print(". done",donecount,"impossible",impcount,"passive",passivecount,"b
        return

    def print_proof_records(self,Pp):
        #
        ###
        if Pp.profile_filter_on:
            prof_filt = 'on'
        else:
            prof_filt = 'off'
        if Pp.halfones_filter_on:
            ho_filt = 'on'
        else:
            ho_filt = 'off'
        bl_iter = Pp.basicloop_iterations
        bl_train = Pp.basicloop_training_iterations
        #
        global_p = itp(Pp.global_params)
        local_p = itp(Pp.local_params)
        ###
        #
        print("-----------------------------------------------------------------")
```

```
        print("proof records for a,b =",itp(Pp.alpha),itp(Pp.beta),"model with n
        print(F'with {bl_iter} basic loops per proof and {bl_train} training seg
        print(F'global model has {global_p} and local rank+score model has {loca
        print("------------------------------------------------------------------")
        proof_number = 0
        for i in range(self.prcursor):
            if self.proofrecord[i,14] > 0:
                print("benchmark proof",end = ' ')
            else:
                print("proof",itp(proof_number),end=' ')
                proof_number += 1
            self.print_proof_recordi(i,Pp)
        print("------------------------------------------------------------------")
        return



    def noiselevel(self,P,count_tensor):
        counterf = count_tensor.to(torch.float)
        counter_period_units = counterf / P.noise_period
        phase = counter_period_units * 2 * torch_pi
        one_plus_cos_over_two = (torch.cos(phase) +1.)/2
        decay = P.noise_decay ** counter_period_units
        level = P.noise_level * one_plus_cos_over_two * decay
        return level


    def reset(self):
        self.histi[:,:] = 0
        self.histf[:,:] = 0.
        self.hlength = 0
        self.training_counter = 0
        self.prcursor = 0
        self.local_tweak_cursor = 0
        self.global_tweak_cursor = 0
        print("reinitialized history")
        return

    def increment(self):
        if self.hlength >= self.hlength_max:
            print("Historical at maximum length, can't add any new entries")
            raise CoherenceError("exiting")
        cursor = self.hlength
        self.hlength += 1
        assert 0 <= cursor < self.hlength_max
        return cursor

    def record_parameters(self,alpha,beta):
        cursor = self.increment()
        self.histi[cursor,0] = self.D['Parameters']
        self.histi[cursor,1] = alpha
        self.histi[cursor,2] = beta
        return

    def record_driver(self,alpha,beta):
```

```
        cursor = self.increment()
        self.histi[cursor,0] = self.D['Driver']
        self.histi[cursor,1] = alpha
        self.histi[cursor,2] = beta
        return

    def record_model(self,n):
        cursor = self.increment()
        self.histi[cursor,0] = self.D['Model']
        self.histi[cursor,1] = n
        return


    def record_loss(self,style,L1_loss,MSE_loss):
        cursor = self.increment()
        self.histi[cursor,0] = self.D['Loss']
        if style != 'global' and style != 'local' and style != 'local_ce':
            raise CoherenceError("unsupported style in record_loss")
        if style == 'global':
            self.histi[cursor,1] = self.D['Global']
        if style == 'local':
            self.histi[cursor,1] = self.D['Local']
        if style == 'local_ce':
            self.histi[cursor,1] = self.D['LocalCE']
        if torch.is_tensor(MSE_loss):
            MSE_loss_detach = MSE_loss.detach()
        else:
            MSE_loss_detach = MSE_loss
        self.histf[cursor,0] = L1_loss.detach()
        self.histf[cursor,1] = MSE_loss_detach
        #
        self.training_counter += 1
        return


    def record_training(self,style,iterations,explore_pre_pool,example_pre_pool,
        cursor = self.increment()
        #
        self.histi[cursor,0] = self.D['Training']
        if style != 'global' and style != 'local':
            raise CoherenceError("unsupported style in record_training")
        if style == 'global':
            self.histi[cursor,1] = self.D['Global']
        if style == 'local':
            self.histi[cursor,1] = self.D['Local']
        self.histi[cursor,2] = iterations
        self.histi[cursor,3] = explore_pre_pool
        self.histi[cursor,4] = example_pre_pool
        self.histi[cursor,5] = example_pool
        return

    def record_full_proof(self,M,steps,cumulative_nodes,done_nodes):
        cursor = self.increment()
        #
        if M.benchmark:
            self.histi[cursor,0] = self.D['BenchmarkProof']
```

```
            self.histi[cursor,0] = self.D['BenchmarkProof']
        else:
            self.histi[cursor,0] = self.D['FullProof']
        self.histi[cursor,1] = steps
        self.histi[cursor,2] = cumulative_nodes
        self.histi[cursor,3] = done_nodes
        return


    def record_dropout_proof(self,style,dropout,steps,ECN):
        cursor = self.increment()
        #
        if style != 'regular' and style != 'adaptive' and style != 'uniform':
            raise CoherenceError("unsupported style in record_dropout_proof")
        #
        self.histi[cursor,0] = self.D['DropoutProof']
        if style == 'regular':
            self.histi[cursor,1] = self.D['Regular']
        if style == 'adaptive':
            self.histi[cursor,1] = self.D['Adaptive']
        if style == 'uniform':
            self.histi[cursor,1] = self.D['Uniform']
        self.histi[cursor,2] = dropout
        self.histi[cursor,3] = steps
        ecnr = torch.round(ECN).to(torch.int64)
        self.histi[cursor,4] = ecnr
        return


    def print_history(self):
        length = self.hlength
        print("--  --  --  --  --  --  --  --  --  --  --  --  --  --  --")
        print("     printing history of length",itp(length))
        print("--  --  --  --  --  --  --  --  --  --  --  --  --  --  --")
        for cursor in range(length):
            tag = self.histi[cursor,0]
            a = itp(self.histi[cursor,1])
            b = itp(self.histi[cursor,2])
            c = itp(self.histi[cursor,3])
            d = itp(self.histi[cursor,4])
            e = itp(self.histi[cursor,5])
            f = itp(self.histi[cursor,6])
            #
            x = numpr(self.histf[cursor,0],3)
            y = numpr(self.histf[cursor,1],3)
            #
            print("(",cursor,")--",end='')
            #
            if tag == self.D['Parameters']:
                print("setting parameters for alpha=",a,"beta=",b)
            #
            if tag == self.D['Driver']:
                print("initialize driver for alpha=",a,"beta=",b)
            #
            if tag == self.D['Model']:
                print("initialize model with n =",a)
```

```
            print("including model with x",a)
        #
        if tag == self.D['Loss']:
            if a == self.D['Global']:
                print("test global model, L1 loss",x,"MSE loss",y)
            if a == self.D['Local']:
                print("test local model, L1 loss",x,"MSE loss",y)
            if a == self.D['LocalCE']:
                print("test local model, CE loss",x)
        #
        if tag == self.D['Training']:
            if a == self.D['Global']:
                print("training global model",end=' ')
            if a == self.D['Local']:
                print("training local model",end=' ')
            print("iterations",b,"explore prepool",c,"example prepool",d,"ex
        #
        if tag == self.D['BenchmarkProof']:
            print("BENCHMARK PROOF in",a,"steps",b,"cumulative nodes",c,"don
        #
        if tag == self.D['FullProof']:
            print("FULL PROOF in",a,"steps",b,"cumulative nodes",c,"done lea
        #
        if tag == self.D['DropoutProof']:
            print("proof with dropout style",end='')
            if a == self.D['Regular']:
                print(" regular ",end='')
            if a == self.D['Adaptive']:
                print(" adaptive ",end='')
            if a == self.D['Uniform']:
                print(" uniform ",end='')
            print("threshold",b,"in",c,"steps with estimated cumulative node
    #
    print("--  --  --  --  --  --  --  --  --  --  --  --  --  --  --")
    print("     end printing history of length",itp(length))
    print("--  --  --  --  --  --  --  --  --  --  --  --  --  --  --")
    #
    return



def graph_history(self,P,style):
    #
    alpha = P.alpha
    beta = P.beta
    nu = P.model_n
    #
    ###
    if P.profile_filter_on:
        prof_filt = 'on'
    else:
        prof_filt = 'off'
    if P.halfones_filter_on:
        ho_filt = 'on'
    else:
        ho filt = 'off'
```

```
        bl_iter = P.basicloop_iterations
        bl_train = P.basicloop_training_iterations
        #
        global_p = itp(P.global_params)
        local_p = itp(P.local_params)
        ###
        plotcount = 0
        for cursor in range(self.hlength):
            tag = self.histi[cursor,0]
            if tag == self.D['FullProof']:
                plotcount += 1
        baseline = 0
        ecn_graph = torch.zeros((plotcount),dtype = torch.int64,device=Dvc)
        attempts = arangeic(plotcount)
        attempt_number = 0
        for cursor in range(self.hlength):
            tag = self.histi[cursor,0]
            b = self.histi[cursor,2]
            if tag == self.D['BenchmarkProof']:
                baseline = b
            if tag == self.D['FullProof']:
                ecn_graph[attempt_number] = b
                attempt_number += 1
        ecn_graph = torch.clamp(ecn_graph,0,self.proof_nodes_max)
        plt.clf()
        if style == 'big':
            plt.figure(figsize = (17,8))
        else:
            plt.figure(figsize = (8,4))
        ###
        phrase1 = F'Proofs for a= {alpha}, b={beta} for model with n= {nu} and h
        phrase2 = F'with {bl_iter} basic loops per proof and {bl_train} training
        phrase3 = F'global model has {global_p} and local rank+score model has {
        plt.title(phrase1 + '\n' + phrase2 + '\n' + phrase3)
        ###
        plt.xlabel('number of training rounds')
        plt.ylabel('cumulative nodes')
        plt.plot([0,plotcount - 1],[itp(baseline),itp(baseline)],'deepskyblue')
        plt.plot(nump(attempts),nump(ecn_graph),'.-')
        plt.show()
        #
        pcg = 0
        pcl = 0
        for cursor in range(self.hlength):
            tag = self.histi[cursor,0]
            a = self.histi[cursor,1]
            if tag == self.D['Loss']:
                if a == self.D['Global']:
                    pcg += 1
                if a == self.D['Local']:
                    pcl += 1
                if a == self.D['LocalCE']:
                    pcl += 1
        L1graph_local = torch.zeros((pcl),dtype = torch.float,device=Dvc)
        MSEgraph_local = torch.zeros((pcl),dtype = torch.float,device=Dvc)
```

```
        CEgraph_local = torch.zeros((pcl),dtype = torch.float,device=Dvc)
        L1graph_global = torch.zeros((pcg),dtype = torch.float,device=Dvc)
        MSEgraph_global = torch.zeros((pcg),dtype = torch.float,device=Dvc)
        measurements_local = arangeic(pcl)
        measurements_global = arangeic(pcg)
        measurement_number_local = 0
        measurement_number_global = 0
        for cursor in range(self.hlength):
            tag = self.histi[cursor,0]
            a = self.histi[cursor,1]
            x = self.histf[cursor,0]
            y = self.histf[cursor,1]
            if tag == self.D['Loss']:
                if a == self.D['Global']:
                    L1graph_global[measurement_number_global] = x
                    MSEgraph_global[measurement_number_global] = y
                    measurement_number_global += 1
                if a == self.D['Local']:
                    L1graph_local[measurement_number_local] = x
                    MSEgraph_local[measurement_number_local] = y
                    measurement_number_local += 1
                if a == self.D['LocalCE']:
                    CEgraph_local[measurement_number_local] = x
                    measurement_number_local += 1
        L1avg_global = (L1graph_global[0:pcg-2] + L1graph_global[1:pcg-1] + L1gr
        L1avg_global = torch.clamp(L1avg_global,0.,0.2)
        L1avg_local = (L1graph_local[0:pcl-2] + L1graph_local[1:pcl-1] + L1graph
        #
        L1avg_local_red = L1avg_local /2.  # so it fits on the graph better, wit
        #
        L1avg_local_red = torch.clamp(L1avg_local_red,0.,0.2)
        MSEavg_global = (MSEgraph_global[0:pcg-2] + MSEgraph_global[1:pcg-1] + M
        MSEavg_global = torch.clamp(MSEavg_global,0.,0.2)
        MSEavg_local = (MSEgraph_local[0:pcl-2] + MSEgraph_local[1:pcl-1] + MSEg
        MSEavg_local = torch.clamp(MSEavg_local,0.,0.2)
        CEavg_local = (CEgraph_local[0:pcl-2] + CEgraph_local[1:pcl-1] + CEgraph
        CEavg_local = CEavg_local / 10.
        CEavg_local = torch.clamp(CEavg_local,0.,0.2)
        #
        #
        noiselevel = HST.noiselevel(P,measurements_global)
        #
        plt.clf()
        if style == 'big':
            plt.figure(figsize = (17,8))
        else:
            plt.figure(figsize = (8,4))
        ###
        phrase1b = F'Training for a= {alpha}, b={beta} for model with n= {nu}, {
        # phrase 2 is the same as before
        plt.title(phrase1b + '\n' + phrase2 + '\n' + phrase3)
        ###
        plt.xlabel('training')
        plt.ylabel('loss')
        #
```

```
        plt.plot(nump(measurements_local[5:pcl]),numpr(L1avg_local_red[3:pcl-2],
        plt.plot(nump(measurements_global[5:pcg]),numpr(L1avg_global[3:pcg-2],4)
        #
        plt.plot(nump(measurements_local[5:pcl]),numpr(MSEavg_local[3:pcl - 2],5
        plt.plot(nump(measurements_global[5:pcg]),numpr(MSEavg_global[3:pcg - 2]
        #
        plt.plot(nump(measurements_global[5:pcg]),numpr(noiselevel[3:pcg-2],5),l
        #
        plt.legend()
        plt.show()
        ###
        self.print_proof_records(P)
        return
```

```
###
```

```
class SymmetricGroup :
    def __init__(self,p):
        #
        if p < 1:
            print("can't initialize a symmetric group with size",p)
            raise CoherenceError("exiting")
        if p > 9:
            print("symmetric group size",p,"is probably going to cause a memory
            raise CoherenceError("exiting")
        #
        self.p = p
        #self.subgroups_max = subgroups_max # this is not currently used
        #
        self.gtlength = 1
        for i in range(self.p):
            self.gtlength *= (i+1)
        #
        self.grouptable = self.makegrouptable()
        self.gtbinary = self.makegrouptablebinary()
        #self.multiplicationtable = self.makemult()
        #self.inverse = self.makeinverse()
        self.inversetable = self.makeinversetable()
        #


    def symmetricgrouptable(self,k):
        assert k > 0
        if k == 1:
```

```
          sgt = torch.zeros((1),dtype = torch.int64,device = Dvc)
          return 1,sgt
      length_prev,sgtprev = self.symmetricgrouptable(k-1)
      length = length_prev * k
      krange = torch.arange((k),dtype = torch.int64,device=Dvc)  # same as ara
      krangevx = krange.view(k,1).expand(k,length_prev)
      krangevx2 = krange.view(k,1,1).expand(k,length_prev,k-1)
      #
      sgtprev_vx = sgtprev.view(1,length_prev,k-1).expand(k,length_prev,k-1)
      #
      krange1vxr = krange.view(k,1).expand(k,k).reshape(k*k)
      krange2vxr = krange.view(1,k).expand(k,k).reshape(k*k)
      gappedtablev = krange2vxr[(krange1vxr != krange2vxr)]
      gappedtable = gappedtablev.view(k,k-1)
      #
      afterpart = gappedtable[krangevx2,sgtprev_vx]
      beforepart = krange.view(k,1,1).expand(k,length_prev,1)
      newtablev = torch.cat((beforepart,afterpart),2)
      newtable = newtablev.view(length,k)
      return length,newtable

  def makegrouptable(self):
      length,table = self.symmetricgrouptable(self.p)
      assert length == self.gtlength
      #print("making group table for symmetric group, as an array of shape",ta
      return table


  def findpermutation(self,batchlength,vector):
      vectorvx = vector.view(batchlength,1,self.p).expand(batchlength,self.gtl
      grouptablevx = self.grouptable.view(1,self.gtlength,self.p).expand(batch
      detection = (vectorvx == self.grouptable).all(2)
      values,permutation = torch.max((detection.to(torch.int)),1)
      assert (values == 1).all(0)
      return permutation


  def makemult(self):
      if self.p > 7:
        print("multiplication table for symmetric group of size",self.p,"would
        raise CoherenceError("exiting")
      print("setting up multiplication table...",end=' ')
      mult = torch.zeros((self.gtlength,self.gtlength),dtype = torch.int64,dev
      for x in range(self.gtlength):
          xvector = self.grouptable[x]
          comptable = xvector[self.grouptable]
          mult[x,:] = self.findpermutation(self.gtlength,comptable)
      print("done")
      return mult

  def makeinverse(self):
      invdetection = (self.multiplicationtable == 0)
      values,inverse = torch.max((invdetection.to(torch.int)),1)
      return inverse

```

```
    def makeinversetable(self):
        gl = self.gtlength
        p=self.p
        tablevx = self.grouptable.view(gl,p,1).expand(gl,p,p)
        yrangevx = arangeic(p).view(1,1,p).expand(gl,p,p)
        delta = (tablevx == yrangevx).to(torch.int64)
        values,inversetable = torch.max(delta,1)
        return inversetable



    ##########@ for the list of subgroups ##########

    def subgroupgen(self,thesubgroup,thex):  # outputs the subgroup generated by
        currentsubset = torch.zeros((self.gtlength),dtype = torch.bool,device=Dv
        currentsubset[thesubgroup] = True
        currentsubset[thex] = True
        for i in range(1000):
            currentlength = currentsubset.to(torch.int).sum(0).clone()
            cl2 = currentlength * currentlength
            #print("current length",itp(currentlength))
            mtcurrent1 = self.multiplicationtable[currentsubset]
            mtcurrent1p = mtcurrent1.permute(1,0)
            mtcurrent2 = mtcurrent1p[currentsubset]
            mtcurrent2vx = mtcurrent2.view(1,cl2).expand(self.gtlength,cl2)
            grouparangevx = arangeic(self.gtlength).view(self.gtlength,1).expand
            products = (grouparangevx == mtcurrent2vx).any(1)
            currentsubset = currentsubset | products
            newlength = currentsubset.to(torch.int).sum(0)
            if newlength == currentlength:
                #print("break")
                break
        return currentsubset

    def findsubgroup(self,thesubgroup):
        currentsglist = self.subgroup[0:self.sglistlength,:]
        thesubgroupvx = thesubgroup.view(1,self.gtlength).expand(self.sglistleng
        findsg = (currentsglist == thesubgroupvx).all(1)
        if findsg.any(0):
            assert findsg.to(torch.int).sum(0) == 1
            sgrange = arangeic(self.sglistlength)
            sgnumber = itp(sgrange[findsg][0])
            return True,sgnumber
        else:
            return False, None


    def addnextsubgroup(self):
        for k in range(self.sglistlength):
            thesubgroup = self.subgroup[k]
            #print("try subgroup",k)
            for x in range(self.gtlength):
                if not thesubgroup[x]:
                    #print("try x=",x)
                    sggenx = self.subgroupgen(thesubgroup,x)
```

```
                    fsg,sgnumber = self.findsubgroup(sggenx)
                    if not fsg:
                        self.subgroup[self.sglistlength] = sggenx
                        self.sgsize[self.sglistlength] = sggenx.to(torch.int).su
                        #print("add subgroup",itp(self.sglistlength),"of length"
                        self.sglistlength += 1
                        return True
        return False

    def createsubgrouplist(self):
        # the identity
        self.sglistlength = 1
        self.subgroup.masked_fill_(truetensor,False)
        self.sgsize.masked_fill_(truetensor,0)
        self.sgsize[0] = 1
        self.subgroup[0,0] = True
        for i in range(self.subgroups_max - 1):
            ansg = self.addnextsubgroup()
            if not ansg:
                break
        print("created a list of",itp(self.sglistlength),"subgroups")
        return


    def findsubgroupbatch(self,batchsize,sgbatch):
        currentsglist = self.subgroup[0:self.sglistlength,:]
        currentsglistvx = currentsglist.view(1,self.sglistlength,self.gtlength).
        sgbatchvx = sgbatch.view(batchsize,1,self.gtlength).expand(batchsize,sel
        findsg = (currentsglist == thesubgroupvx).all(2)
        #
        assert(findsg.to(torch.int).sum(1) == 1).all(0)
        #
        sglistarangevx = arangeic(self.sglistlength).view(1,self.sglistlength).e
        sglistarangevxv = sglistarangevx.reshape(batchsize*self.sglistlength)
        findsgv = findsg.reshape(batchsize*self.sglistlength)
        output = sglistarangevxv[findsgv]
        return output

    def makegrouptablebinary(self):
        p = self.p
        gl = self.gtlength
        if p > 7:
            print("warning: not making binary table for p=",itp(p),"> 7")
            return None
        blength = 2**p
        brange = arangeic(blength)
        zbinarytable = torch.zeros((blength,p),dtype = torch.bool,device=Dvc)
        for z in range(blength):
            zbinarytable[z,:] = zbinary(p,z)
        gtb = self.grouptable.view(gl,1,p).expand(gl,blength,p).reshape(gl*bleng
        brange = arangeic(blength).view(1,blength,1).expand(gl,blength,p).reshap
        #
        gtb_mod = zbinarytable[brange,gtb]
        #
        gt_binaryv = binaryzbatch(gl*blength,p,gtb_mod)
        gt_binarym = gt_binaryv.view(gl,blength,
```

```
        gt_binary = gt_binaryv.view(gl,blength)
        #print("made group table binary")
        return gt_binary
```

```
### this is the cell where the neural networks are created. We include input and
### template assertions in case one wants to experiment different network archit
###

class PrepareInputLayer(nn.Module):

    def __init__(self,pp):
        super(PrepareInputLayer, self).__init__()
        self.pp = pp
        self.a = pp.alpha
        self.b = pp.beta
        self.bz = self.b + 1
        #
        self.n = self.pp.model_n
        #
        a=self.a
        bz=self.bz
        self.channels = 5*bz + 2*a
        #


    def forward(self,Data):
        #
        a = self.a
        b = self.b
        bz = self.bz
        #
        length = Data['length']
        prod = Data['prod']
        left = Data['left']
        right = Data['right']
        ternary = Data['ternary']
        #
        ############################
        # input template assertions:
        #
        assert prod.dtype == torch.bool
        assert left.dtype == torch.bool
        assert right.dtype == torch.bool
        assert ternary.dtype == torch.bool
        #
        assert prod.size() == torch.Size([length,a,a,bz])
        assert left.size() == torch.Size([length,a,bz,2])
        assert right.size() == torch.Size([length,bz,a,2])
        assert ternary.size() == torch.Size([length,a,a,a,2])
        #
        ############################
```

```
        #
        prod_data = prod.permute(0,3,1,2).view(length,bz,a,a)
        left_data = left.permute(0,3,2,1).reshape(length,2,bz,a,1).expand(length
        right_data = right.permute(0,3,1,2).reshape(length,2,bz,1,a).expand(leng
        ternary_data = ternary.permute(0,4,2,1,3).reshape(length,2,a,a,a)
        with torch.no_grad():
            prod_f = prod_data.float()
            prod_denom = prod_f.sum(1).view(length,1,a,a).expand(length,bz,a,a)
            prod_denom = torch.clamp(prod_denom,1.,100.)
            prod_ren = prod_f / prod_denom
            prod_ren = (bz * prod_ren) - 1.
            #
            left_f = left_data.float()
            left_denom = left_f.sum(1).view(length,1,bz,a,a).expand(length,2,bz,
            left_denom = torch.clamp(left_denom,1.,100.)
            left_ren = (left_f / left_denom).view(length,2*bz,a,a)
            left_ren = left_ren - 0.5
            #
            right_f = right_data.float()
            right_denom = right_f.sum(1).view(length,1,bz,a,a).expand(length,2,b
            right_denom = torch.clamp(right_denom,1.,100.)
            right_ren = (right_f / right_denom).view(length,2*bz,a,a)
            right_ren = right_ren - 0.5
            #
            ternary_f = ternary_data.float()
            ternary_denom = ternary_f.sum(1).view(length,1,a,a,a).expand(length,
            ternary_denom = torch.clamp(ternary_denom,1.,100.)
            ternary_ren = (ternary_f / ternary_denom).reshape(length,2*a,a,a)
            ternary_ren = ternary_ren - 0.5
            #
            initial_data = torch.cat((prod_ren,left_ren,right_ren,ternary_ren),1
        #
        assert initial_data.size() == torch.Size([length,self.channels,a,a])
        #
        return initial_data, prod_data



class OutputLayerScalar(nn.Module):

    def __init__(self,pp,channels_in,channels_mid):
        super(OutputLayerScalar, self).__init__()
        self.pp = pp
        self.a = self.pp.alpha
        #
        self.channels_in = channels_in
        self.channels_mid = channels_mid
        #
        self.lrl = nn.LeakyReLU()
        #
        self.layerD1 = nn.Linear(channels_in,channels_mid,1)
        self.layerD2 = nn.Linear(channels_mid,channels_mid,1)
        self.layerD3 = nn.Linear(2*channels_mid,1,1)


```

```
    def forward(self,layer_data):
        #
        length = layer_data.size()[0]
        #
        layer_in = layer_data.view(length,self.channels_in)
        yD1 = self.lrl(self.layerD1(layer_in))
        yD2 = self.lrl(self.layerD2(yD1))
        yD3 = self.layerD3(torch.cat((yD1,yD2),1))
        #
        yScore = yD3.view(length)
        #
        return yScore

class OutputLayer2d(nn.Module):

    def __init__(self,pp,channels_array,channels_mid,channels_extra):
        super(OutputLayer2d, self).__init__()
        self.pp = pp
        self.a = self.pp.alpha
        #
        self.channels_array = channels_array
        self.channels_in = self.channels_array * self.a * self.a
        self.channels_mid = channels_mid
        self.channels_extra = channels_extra
        #
        self.lrl = nn.LeakyReLU()
        #
        self.layerD1 = nn.Linear(self.channels_in,channels_mid,1)
        self.layerD2 = nn.Linear(channels_mid,channels_mid,1)
        self.layerD3 = nn.Linear(2*channels_mid,channels_extra * self.a*self.a,1
        #
        self.layerD4 = nn.Conv2d(self.channels_array + self.channels_extra,self.
        self.layerD5 = nn.Conv2d(self.channels_mid,self.channels_mid,1)
        self.layerD6 = nn.Conv2d(2*self.channels_mid + self.channels_extra,1,1)

    def forward(self,layer_data):
        #
        length = layer_data.size()[0]
        #
        layer_in = layer_data.view(length,self.channels_in)
        yD1 = self.lrl(self.layerD1(layer_in))
        yD2 = self.lrl(self.layerD2(yD1))
        yD3 = self.layerD3(torch.cat((yD1,yD2),1))
        #
        yD3v = yD3.view(length,self.channels_extra,self.a,self.a)
        layer_array = layer_data.view(length,self.channels_array,self.a,self.a)
        yDcat = torch.cat((yD3v,layer_array),1)
        yD4 = self.lrl(self.layerD4(yDcat))
        yD5 = self.lrl(self.layerD5(yD4))
        yD6 = self.layerD6(torch.cat((yD3v,yD4,yD5),1))
        #
        yScore = yD6.view(length,self.a,self.a)
        #
        return yScore

```

```
158
159
160
161
162
163
164 class SGNetProcess(nn.Module):
165
166     def __init__(self,pp):
167         super(SGNetProcess, self).__init__()
168         #
169         self.pp = pp
170         self.a = self.pp.alpha
171         self.bz = self.pp.betaz
172         #
173         #
174         self.lrl = nn.LeakyReLU()
175         #
176         self.prep = PrepareInputLayer(self.pp)
177         #
178         self.channels = self.prep.channels
179         #
180         self.n = self.pp.model_n
181         n = self.n
182         #
183         self.array_channels = n
184         self.process_channels = self.array_channels*self.a*self.a
185         #
186         self.convA1 = nn.Conv2d(self.channels,8*n,1)
187         self.convA2 = nn.Conv2d(8*n,8*n,[5,1],padding = [2,0],padding_mode = 'ci
188         self.convA3 = nn.Conv2d(8*n,8*n,[1,5],padding = [0,2],padding_mode = 'ci
189         self.convA4 = nn.Conv2d(8*n,8*n,[5,1],padding = [2,0],padding_mode = 'ci
190         self.convA5 = nn.Conv2d(8*n,8*n,[1,5],padding = [0,2],padding_mode = 'ci
191         self.convA5tg = nn.Conv2d(40*n,8*n,1)
192         self.convA6 = nn.Conv2d(8*n,8*n,[5,1],padding = [2,0],padding_mode = 'ci
193         self.convA7 = nn.Conv2d(9*n,8*n,[1,5],padding = [0,2],padding_mode = 'ci
194         self.convA8 = nn.Conv2d(8*n,8*n,[5,1],padding = [2,0],padding_mode = 'ci
195         self.convA9 = nn.Conv2d(8*n,8*n,[1,5],padding = [0,2],padding_mode = 'ci
196         #
197         self.convB = nn.Conv2d(40*n,n,1)
198         #
199         self.convC1 = nn.Conv1d(8*n*self.a*self.a,8*n,1,groups=8*n)
200         self.convC2 = nn.Conv1d(8*n,n*self.a*self.a,1,groups = n)
201         #
202         ##
203
204     def forward(self,Data):
205         #
206         initial_data,prod_data = self.prep(Data)
207         length = Data['length']
208         n = self.n
209         #
210         yA1 = self.lrl(self.convA1(initial_data))
211         yA2 = self.lrl(self.convA2(yA1))
212         yA3 = self.lrl(self.convA3(yA2))
```

```
        yA4 = self.lrl(self.convA4(yA3))
        yA5 = self.lrl(self.convA5(yA4))
        yA5tg = self.lrl(self.convA5tg(torch.cat((yA1,yA2,yA3,yA4,yA5),1)))
        yA6 = self.lrl(self.convA6(yA5tg))
        #
        yA4side = yA4.view(length,8*n*self.a*self.a,1)
        yC1 = self.lrl(self.convC1(yA4side))
        yC2 = self.lrl(self.convC2(yC1)).view(length,n,self.a,self.a)
        yA6side = torch.cat((yA6,yC2),1)
        #
        yA7 = self.lrl(self.convA7(yA6side))
        yA8 = self.lrl(self.convA8(yA7))
        yA9 = self.lrl(self.convA9(yA8))
        #
        yB = self.lrl(self.convB(torch.cat((yA1,yA3,yA5tg,yA7,yA9),1)))
        #
        yProcessed = yB.view(length,self.process_channels)
        #
        return yProcessed


class SGNetGlobal(nn.Module):

    def __init__(self,pp):
        super(SGNetGlobal, self).__init__()
        #
        self.pp = pp
        self.a = self.pp.alpha
        #
        self.process = SGNetProcess(self.pp)
        #
        self.outlayer = OutputLayerScalar(self.pp,self.process.process_channels,


    def forward(self,Data):
        #
        # see the PrepareInputLayer() class above for the input template asserti
        #
        yProcessed = self.process(Data)
        #
        yScore = self.outlayer(yProcessed)
        #
        ##############################
        # output template assertions:
        length = Data['length']
        assert yScore.dtype == torch.float
        assert yScore.size() == torch.Size([length])
        ##############################
        #
        return yScore




class SGNetLocal(nn.Module):
```

```
    def __init__(self,pp):
        super(SGNetLocal, self).__init__()
        # an affine operation: y = Wx + b
        #
        self.pp = pp
        self.a = self.pp.alpha
        #
        self.process = SGNetProcess(self.pp)
        #
        self.outlayer = OutputLayer2d(self.pp,self.process.array_channels,32,4)


    def forward(self,Data):
        #
        # see the PrepareInputLayer() class above for the input template asserti
        #
        yProcessed = self.process(Data)
        #
        yScore = self.outlayer(yProcessed)
        #
        ##############################
        # output template assertions:
        length = Data['length']
        a = self.a
        assert yScore.dtype == torch.float
        assert yScore.size() == torch.Size([length,a,a])
        #
        ##############################
        #
        return yScore

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



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


class SgModel :
    def __init__(self,pp):
        self.pp = pp
        #
        self.network = SGNetGlobal(self.pp).to(Dvc)
        #
        self.network2 = SGNetLocal(self.pp).to(Dvc)
        #
        self.average_local_loss = itf(1.0)
        #
        #print(self.network)
        print("set up model network and network2")
        #
        self.benchmark = False
        #
```

```
        #
        self.learning_rate = 0.002  # was 0.002, then 0.003, ...
        self.momentum = 0.95
        #self.weight_decay = 0.0001
        #
        #self.optimizer = optim.SGD(self.network.parameters(), lr=self.learning_
        #self.optimizer2 = optim.SGD(self.network2.parameters(), lr=self.learnin
        self.optimizer = optim.SGD(self.network.parameters(), lr=self.learning_r
        self.optimizer2 = optim.SGD(self.network2.parameters(), lr=self.learning
        #
        self.criterionCE = nn.CrossEntropyLoss()
        self.criterionA = nn.L1Loss()
        self.criterionB = nn.MSELoss()
        #
        self.network2_trainable = True
        #
        pp.global_params, pp.local_params = self.modelcount()
        #
        self.softmax = nn.Softmax(dim = 1)


    def modelcount(self):
        network_param = sum(p.numel() for p in self.network.parameters() if p.re
        print("network parameters",itp(network_param))
        network2_param = sum(p.numel() for p in self.network2.parameters() if p.
        print("network2 parameters",itp(network2_param))
        return network_param, network2_param

    def tweak_network(self,N,density,epsilon):
        for p in N.parameters():
            if p.requires_grad:
                tirage_density = torch.rand(p.size(),device=Dvc)
                modif = torch.rand(p.size(),device=Dvc)
                modif *= (tirage_density < density).to(torch.float) * epsilon
                factor = modif + 1.0
                with torch.no_grad():
                    p *= factor
        return




    def save_model(self,filename):  # tries to save the two model state dicts an
        # I haven't tried these but they should probably mostly work and are inc
        torch.save({
                'network_state_dict': self.network.state_dict(),
                'network2_state_dict': self.network2.state_dict(),
                'optimizer_state_dict': self.optimizer.state_dict(),
                'optimizer2_state_dict': self.optimizer2.state_dict(),
                }, filename)
        print("saved to",filename)
        return

    def load_model(self,filename):  # tries to load from the file
        #
```

```
        loadedmodels = torch.load(filename)
        network_state_dict = loadedmodels['network_state_dict']
        network2_state_dict = loadedmodels['network2_state_dict']
        optimizer_state_dict = loadedmodels['optimizer_state_dict']
        optimizer2_state_dict = loadedmodels['optimizer2_state_dict']
        #
        self.network.load_state_dict(network_state_dict)
        self.network2.load_state_dict(network2_state_dict)
        self.optimizer.load_state_dict(optimizer_state_dict)
        self.optimizer2.load_state_dict(optimizer2_state_dict)
        #
        self.network.train()
        self.network2.train()
        print("loaded from",filename)
        return


class ProtoModel :
    def __init__(self,pp,style):
        self.pp = pp
        self.style = style
        self.a = pp.alpha
        self.a3z = self.a*self.a*self.a + 1
        self.b = pp.beta
        self.bz = self.b + 1
        #
        self.random_order = torch.randperm(self.a*self.a)
        self.spiral, self.spiral_mix = self.makespiral()
        self.rays = self.makerays()
        #
        self.network = SGNetGlobal(self.pp).to(Dvc)
        # there is no network2
        self.network2_trainable = False
        #
        #print(self.network)
        print("set up the proto-model network---network2 is not trainable, this
        #
        self.benchmark = True
        #
        self.learning_rate = 0.002
        #
        self.optimizer = optim.SGD(self.network.parameters(), lr=self.learning_r
        #
        self.criterionA = nn.L1Loss()
        self.criterionB = nn.MSELoss()
        #
        self.softmax = nn.Softmax(dim = 1)

    def makespiral(self):
        spiral_order = torch.zeros((self.a,self.a),dtype = torch.int64,device=Dv
        spiral_order[0,0] = 0
        count = 1
        for x in range(1,self.a):
            spiral_order[x,x] = count
            count += 1
```

```
            for y in range(x):
                spiral_order[x,y] = count
                count += 1
                spiral_order[y,x] = count
                count += 1
        spiral_orderf = spiral_order.to(torch.float) / itt(self.a*self.a).to(tor
        thresh = self.pp.spiral_mix_threshold
        over_thresh = (spiral_order > thresh).view(self.a*self.a)
        spiral_mix = spiral_order.clone().view(self.a*self.a)
        spiral_mix[over_thresh] = arangeic(self.a*self.a)[over_thresh] + thresh
        spiral_mixf = spiral_mix.to(torch.float) / itt(self.a*self.a).to(torch.f
        return spiral_orderf, spiral_mixf

    def makerays(self):
        a = self.a
        ray_order = torch.zeros((self.a,self.a),dtype = torch.int64,device=Dvc)
        count = 0
        for x in range(a):
            ray_order[x,x] = count
            count += 1
        for x in range(a-1):
            for y in range(x+1,a):
                ray_order[x,y] = count
                count += 1
                ray_order[y,x] = count
                count += 1
        ray_orderf = ray_order.to(torch.float) / itt(a*a).to(torch.float)
        return ray_orderf

    def virtual_score(self,Data):
        length = Data['length']
        if self.style == 'random':
            output = torch.rand((length,self.a,self.a),device=Dvc)
        if self.style == 'random_order':
            output = self.random_order.view(1,self.a,self.a).expand(length,self.
        if self.style == 'spiral':
            output = self.spiral.view(1,self.a,self.a).expand(length,self.a,self
        if self.style == 'spiral_mix':
            output = self.spiral_mix.view(1,self.a,self.a).expand(length,self.a,
        if self.style == 'rays':
            output = self.rays.view(1,self.a,self.a).expand(length,self.a,self.a
        return output

    def network2(self,Data):
        #
        a = self.a
        #
        length = Data['length']
        prod = Data['prod']
        #
        prodsum = prod.to(torch.int64).sum(3)
        availablexyv = (prodsum > 1).view(length*a*a)
        #
        vsv = self.virtual_score(Data).reshape(length*a*a)
        #
```

```
        vsv[~availablexyv] = 100.
        #
        values,xyvector = torch.min(vsv.view(length,a*a),1)
        #
        vs = vsv.view(length,a*a)
        return vs
```

```

```

```
### classes Relations1 to Relations4 are viewed as chapters
```

```
class Relations1 :
    def __init__(self,pp):
        #
        self.pp = pp
        #
        self.alpha = self.pp.alpha
        self.alpha2 = self.alpha * self.alpha
        self.alpha3 = self.alpha * self.alpha * self.alpha
        self.alpha3z = self.alpha3 + 1
        self.beta = self.pp.beta
        self.betaz = self.beta +1
        #
        #self.model = self.mm
        #
        self.qvalue = self.pp.qvalue
        #
        self.ascore_max = self.pp.ascore_max
        #
        self.infosize =  self.pp.infosize
        self.pastsize = self.pp.pastsize
        self.futuresize = self.pp.futuresize
        #
        #
        self.ar1 = arangeic(self.alpha).view(self.alpha,1).expand(self.alpha,sel
        self.ar2 = arangeic(self.alpha).view(1,self.alpha).expand(self.alpha,sel
        self.ida = (self.ar1 == self.ar2)
        #
        self.a3r1 = arangeic(self.alpha3).view(self.alpha3,1).expand(self.alpha3
        self.a3r2 = arangeic(self.alpha3).view(1,self.alpha3).expand(self.alpha3
        self.eqa3 = (self.a3r1 == self.a3r2)
        #
        self.a3zr1 = arangeic(self.alpha3z).view(self.alpha3z,1).expand(self.alp
        self.a3zr2 = arangeic(self.alpha3z).view(1,self.alpha3z).expand(self.alp
        self.eqa3z = (self.a3zr1 == self.a3zr2)
        #
        self.iblength = 2 ** (2*self.beta)
        self.ibarray = torch.zeros((self.iblength,2*self.beta),dtype = torch.boo
```

```
        for z in range(self.iblength):
            self.ibarray[z] = zbinary(2*self.beta , z)
        #
        self.betazsubsets = self.makebetazsubsets()  # at location j,:,: it is f
        self.quantities = self.betazsubsets[:,0:self.beta].to(torch.int).sum(1)
        #


    ##### general manipulation of data

    def printprod(self,Data,i):
        length = Data['length']
        prod = Data['prod']
        a = self.alpha
        bz = self.betaz
        #
        assert i < length
        #
        printarray = torch.zeros((a,a),dtype = torch.int,device=Dvc)
        printarray += 9 * (10 ** bz)
        for p in range(bz):
            printarray += (10 ** p) * (prod[i,:,:,p].to(torch.int))
        print(nump(printarray))
        return

    def printrandomprods(self,Data,number):
        length = Data['length']
        upper = number
        if upper > length:
            upper = length
        indices = torch.randperm(length,device=Dvc)
        for i in range(upper):
            indexi = indices[i]
            print("---------------------------------------")
            self.printprod(Data,indexi)
        print("---------------------------------------")
        return


    def makebetazsubsets(self):
        b = self.beta
        bz = self.betaz
        bpower = 2**b
        subsets = torch.ones((bpower,bz),dtype = torch.bool,device=Dvc)
        for z in range(bpower):
            subsets[z,0:b] = zbinary(b,z)
        return subsets



    def nulldata(self):
        length = torch.tensor(0)
        Output = {
            'length': length,
            'depth': None,
            'prod': None
```

```
            'prod': None,
            'left': None,
            'right': None,
            'ternary': None,
            'info': None,
        }
        return Output

    def copydata(self,Data):
        if Data['length'] == 0:
            return self.nulldata()
        Output = {}
        Output['length'] = itt(Data['length']).clone().detach()
        for ky in Data.keys():
            if ky != 'length':
                Output[ky] = (Data[ky]).clone().detach()
        return Output


    def duplicatedata(self,Data):  # like copydata but it puts the same objects
        if Data['length'] == 0:
            return self.nulldata()
        Output = {}
        Output['length'] = itt(Data['length']).clone().detach()
        for ky in Data.keys():
            if ky != 'length':
                Output[ky] = Data[ky]
        return Output


    def deletedata(self,Data):
        #del Data['length']  # better avoid doing that
        datakeyslist = list(Data.keys())
        for ky in datakeyslist:
            if ky != 'length':
                del Data[ky]
        del Data
        return



    def appenddata(self,Data1,Data2): # appends Data2 to Data1 and outputs the r
        # there is a case where Data1 == None then we just output Data2
        assert set(Data1.keys()) == set(Data2.keys())
        if Data1['length'] == 0:
            return self.copydata(Data2)
        if Data2['length'] == 0:
            return self.copydata(Data1)
        Output = {}
        Output['length'] = Data1['length'] + Data2['length']
        #
        for ky in Data1.keys():
            if ky != 'length':
                Output[ky] = torch.cat((Data1[ky],Data2[ky]),0)
        return Output

```

```
149
150     def indexselectdata(self,Data,indices):
151         #
152         if len(indices) == 0:
153             return self.nulldata()
154         #
155         Output = {}
156         #
157         Output['length'] = len(indices)
158         #
159         for ky in Data.keys():
160             if ky != 'length':
161                 Output[ky] = (Data[ky])[indices].clone().detach()
162         #
163         return Output
164
165
166     def detectsubdata(self,Data,detection):
167         #
168         assert len(detection) == Data['length']
169         #
170         sublength = detection.to(torch.int).sum(0)
171         if sublength == 0:
172             return self.nulldata()
173         #
174         Output = {}
175         #
176         Output['length'] = sublength
177         #
178         for ky in Data.keys():
179             if ky != 'length':
180                 Output[ky] = (Data[ky])[detection].detach()
181         #
182         return Output
183
184     def insertdata(self,Data,detection,SubData):
185         #
186         assert set(Data.keys()) == set(SubData.keys())
187         #
188         sublength = SubData['length']
189         assert detection.to(torch.int).sum(0) == sublength
190         #
191         if sublength == 0:
192             return Data
193         #
194         Output = {}
195         Output['length'] = itt(Data['length'])
196         #
197         for ky in Data.keys():
198             if ky != 'length':
199                 outputitem = Data[ky].clone().detach()
200                 outputitem[detection] = SubData[ky]
201                 Output[ky] = outputitem
202         #
203         return Output
```

```
        return output
```

```
    ##########################
```

```
    def filterpossible(self,Data):  # here we just filter out the impossible cas
        #
        length = Data['length']
        prod = Data['prod']
        #
        prodstats = prod.to(torch.int).sum(3)
        impossible = ((prodstats == 0).any(2)).any(1)
        #
        detection = ~impossible
        return detection
```

```
    def knowledge(self,Data):  # now it increases as we refine
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        bz = self.betaz
        #
        #
        length = Data['length']
        prod = Data['prod']  # mask of shape a.a.bz with boolean values
        left = Data['left']
        right = Data['right']
        ternary = Data['ternary']
        #
        if length == 0:
            zerokn = torch.zeros((1),dtype = torch.int,device=Dvc)
            return zerokn
        #
        output = torch.zeros((length),dtype = torch.int64,device=Dvc)
        output -= prod.to(torch.int64).view(length,a*a*bz).sum(1)
        output -= left.to(torch.int64).view(length,a*bz*2).sum(1)
        output -= right.to(torch.int64).view(length,bz*a*2).sum(1)
        output -= ternary.to(torch.int64).view(length,a*a*a*2).sum(1)
        return output
```

```
    def availablexy(self,length,prod):
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
```

```
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        prodsum = prod.to(torch.int64).sum(3)
        possible = ((prodsum > 0).all(2)).all(1)
        possiblexy = possible.view(length,1).expand(length,a2)
        #
        optionalxy = (prodsum > 1).view(length,a2)
        #
        available_xy = possiblexy & optionalxy
        return available_xy

    def availablexyp(self,length,prod):
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        prodsum = prod.to(torch.int64).sum(3)
        possible = ((prodsum > 0).all(2)).all(1)
        possiblexyp = possible.view(length,1,1,1).expand(length,a,a,bz)
        #
        optionalxyp = (prodsum > 1).view(length,a,a,1).expand(length,a,a,bz)
        #
        available_xyp = prod & possiblexyp & optionalxyp
        return available_xyp

    ###

    def upsplitting(self,Data,ivector,xvector,yvector,pvector):  # setting x.y =
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        if len(ivector) == 0:
            return self.rr1.nulldata()
        #
        UpData = self.indexselectdata(Data,ivector)
        length = UpData['length']
        prod = UpData['prod']
        #
        xrangevx = arangeic(a).view(1,a,1,1).expand(length,a,a,bz)
        yrangevx = arangeic(a).view(1,1,a,1).expand(length,a,a,bz)
        prangevx = arangeic(bz).view(1,1,1,bz).expand(length,a,a,bz)
        #
        xvectorvx = xvector.view(length,1,1,1).expand(length,a,a,bz)
        yvectorvx = yvector.view(length,1,1,1).expand(length,a,a,bz)
```

```
        pvectorvx = pvector.view(length,1,1,1).expand(length,a,a,bz)
        #
        newprod = prod & ( (xrangevx != xvectorvx) | (yrangevx != yvectorvx) | (
        #
        UpData['prod'] = newprod
        UpData['depth'] += 1
        #
        return UpData
```

```
class Relations2 :
    def __init__(self,pp):
        #
        self.pp = pp
        #
        self.rr1 = Relations1(pp)
        #
        self.alpha = self.pp.alpha
        self.alpha2 = self.alpha * self.alpha
        self.alpha2z = self.alpha2 + 1
        self.alpha3 = self.alpha * self.alpha * self.alpha
        self.alpha3z = self.alpha3 + 1
        self.beta = self.pp.beta
        self.betaz = self.beta +1
        #
        #
        self.halfones_count = 0
        self.impossible_basic_count = 0
        #




    def filterpossible(self,Data):  # here we just filter out the impossible cas
        #
        length = Data['length']
        prod = Data['prod']
        #
        prodstats = prod.to(torch.int).sum(3)
        impossible = ((prodstats == 0).any(2)).any(1)
        #
        detection = ~impossible
        return detection



    ########################################
    ##### new steps with prod, left, right, ternary

    def modifyternaryStep(self,Data):
        #
```

```
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length = Data['length']
        prod = Data['prod']
        left = Data['left']
        right = Data['right']
        ternary = Data['ternary']
        #
        #
        ivx = arangeic(length).view(length,1,1,1,1).expand(length,a,a,a,bz)
        xvx = arangeic(a).view(1,a,1,1,1).expand(length,a,a,a,bz)
        yvx = arangeic(a).view(1,1,a,1,1).expand(length,a,a,a,bz)
        zvx = arangeic(a).view(1,1,1,a,1).expand(length,a,a,a,bz)
        pvx = arangeic(bz).view(1,1,1,1,bz).expand(length,a,a,a,bz)
        #
        nter0_left = (prod[ivx,yvx,zvx,pvx] & left[ivx,xvx,pvx,0]).any(4)
        nter1_left = (prod[ivx,yvx,zvx,pvx] & left[ivx,xvx,pvx,1]).any(4)
        #
        nter0_right = (prod[ivx,xvx,yvx,pvx] & right[ivx,pvx,zvx,0]).any(4)
        nter1_right = (prod[ivx,xvx,yvx,pvx] & right[ivx,pvx,zvx,1]).any(4)
        #
        nter0v = (nter0_left & nter0_right)
        nter1v = (nter1_left & nter1_right)
        #
        newternary = ternary.clone()
        newternary[:,:,:,:,0] = ternary[:,:,:,:,0] & nter0v
        newternary[:,:,:,:,1] = ternary[:,:,:,:,1] & nter1v
        #
        NewData = self.rr1.copydata(Data)
        NewData['ternary'] = newternary.detach()
        #
        return NewData

    def modifyleftrightStep(self,Data):
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length = Data['length']
        prod = Data['prod']
        left = Data['left']
        right = Data['right']
        ternary = Data['ternary']
        #
        prodstats = prod.to(torch.int64).sum(3)
        unique = (prodstats == 1)
```

```
        unique = (prodsum3 == 1)
        #
        ivx = arangeic(length).view(length,1,1,1,1).expand(length,a,a,a,bz)
        xvx = arangeic(a).view(1,a,1,1,1).expand(length,a,a,a,bz)
        yvx = arangeic(a).view(1,1,a,1,1).expand(length,a,a,a,bz)
        zvx = arangeic(a).view(1,1,1,a,1).expand(length,a,a,a,bz)
        pvx = arangeic(bz).view(1,1,1,1,bz).expand(length,a,a,a,bz)
        #
        nleft0 = (( (~prod[ivx,yvx,zvx,pvx]) |  (~unique[ivx,yvx,zvx]) | ternary
        nleft1 = (( (~prod[ivx,yvx,zvx,pvx]) |  (~unique[ivx,yvx,zvx]) | ternary
        #
        nright0 = (( (~prod[ivx,xvx,yvx,pvx]) |  (~unique[ivx,xvx,yvx]) | ternar
        nright1 = (( (~prod[ivx,xvx,yvx,pvx]) |  (~unique[ivx,xvx,yvx]) | ternar
        #
        newleft = left.clone()
        newright = right.clone()
        #
        newleft[:,:,:,0] = left[:,:,:,0] & nleft0
        newleft[:,:,:,1] = left[:,:,:,1] & nleft1
        newright[:,:,:,0] = right[:,:,:,0] & (nright0.permute(0,2,1))
        newright[:,:,:,1] = right[:,:,:,1] & (nright1.permute(0,2,1))
        #
        NewData = self.rr1.copydata(Data)
        NewData['left'] = newleft.detach()
        NewData['right'] = newright.detach()
        #
        return NewData



    def modifyprodStep(self,Data):
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length = Data['length']
        prod = Data['prod']
        left = Data['left']
        right = Data['right']
        ternary = Data['ternary']
        #
        lvx = arangeic(length).view(length,1,1,1,1).expand(length,a,a,a,bz)
        xvx = arangeic(a).view(1,a,1,1,1).expand(length,a,a,a,bz)
        yvx = arangeic(a).view(1,1,a,1,1).expand(length,a,a,a,bz)
        zvx = arangeic(a).view(1,1,1,a,1).expand(length,a,a,a,bz)
        pvx = arangeic(bz).view(1,1,1,1,bz).expand(length,a,a,a,bz)
        #
        leftbin01 = (left[lvx,xvx,pvx,0] | ternary[lvx,xvx,yvx,zvx,1])
        leftbin10 = (left[lvx,xvx,pvx,1] | ternary[lvx,xvx,yvx,zvx,0])
        #
        rightbin01 = (right[lvx,pvx,zvx,0] | ternary[lvx,xvx,yvx,zvx,1])
```

```
        rightbin10 = (right[lvx,pvx,zvx,1] | ternary[lvx,xvx,yvx,zvx,0])
        #
        newprod = prod.clone()
        newprod = newprod & ( (leftbin01 & leftbin10).all(1) )
        newprod = newprod & ( (rightbin01 & rightbin10).all(3) )
        #
        NewData = self.rr1.copydata(Data)
        NewData['prod'] = newprod.detach()
        #
        return NewData


    def process(self,Data):
        length = Data['length']
        if length == 0:
            return Data
        #
        #
        OutputData = self.rr1.copydata(Data)
        nprod = Data['prod']
        nprodstats = nprod.to(torch.int64).sum(3)
        subset = ((nprodstats > 0).all(2)).all(1)
        NextData = self.rr1.detectsubdata(Data,subset)
        if subset.to(torch.int).sum(0) == 0:
            return OutputData
        for i in range(1000):
            priorknowledge = self.rr1.knowledge(NextData)
            #
            #
            NextData = self.modifyternaryStep(NextData)
            #
            NextData = self.modifyleftrightStep(NextData)
            #
            NextData = self.modifyprodStep(NextData)
            #
            nextknowledge = self.rr1.knowledge(NextData)
            nextdonedetect = (priorknowledge >= nextknowledge)
            subset_nextdone = composedetections(length,subset,nextdonedetect)
            NextDoneData = self.rr1.detectsubdata(NextData,nextdonedetect)
            OutputData = self.rr1.insertdata(OutputData,subset_nextdone,NextDone
            #
            subset = subset & (~subset_nextdone)
            if subset.to(torch.int).sum(0) == 0:
                break
            NextData = self.rr1.detectsubdata(NextData, ~nextdonedetect )
        return OutputData


    def impossibleFilter(self,Data):
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
```

```
        #
        length = Data['length']
        prod = Data['prod']
        left = Data['left']
        right = Data['right']
        ternary = Data['ternary']
        #
        prodstats = prod.to(torch.int64).sum(3)
        possible = ((prodstats > 0).all(2)).all(1)
        #
        leftv = left.view(length,a*bz,2)
        rightv = right.view(length,bz*a,2)
        ternaryv = ternary.view(length,a3,2)
        left_possible = (leftv.any(2)).all(1)
        right_possible = (rightv.any(2)).all(1)
        ternary_possible = (ternaryv.any(2)).all(1)
        #
        detection = (~possible) | (~left_possible) | (~right_possible) | (~terna
        return detection

    def profileFilter(self,Data):
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length = Data['length']
        prod = Data['prod']
        left = Data['left']
        right = Data['right']
        ternary = Data['ternary']
        #
        left_def = ( (left.to(torch.int64).sum(3)) == 1 )
        right_def = ( (right.to(torch.int64).sum(3)) == 1 )
        profile_def = ( left_def.all(1) ) &  ( right_def.all(2) )
        left_pro = ( left[:,:,:,0] ).permute(0,2,1)
        right_pro =  right[:,:,:,0]
        profile = torch.cat((left_pro,right_pro),2)
        profile_vx1 = profile.view(length,bz,1,2*a).expand(length,bz,bz,2*a)
        profile_vx2 = profile.view(length,1,bz,2*a).expand(length,bz,bz,2*a)
        same_profile = (profile_vx1 == profile_vx2).all(3)
        #
        profile_def1 = profile_def.view(length,bz,1).expand(length,bz,bz)
        profile_def2 = profile_def.view(length,1,bz).expand(length,bz,bz)
        #
        same_profile_def = (profile_def1 & profile_def2 & same_profile)
        prange1 = arangeic(bz).view(1,bz,1).expand(length,bz,bz)
        prange2 = arangeic(bz).view(1,1,bz).expand(length,bz,bz)
        #
        detection = (( (prange1 != prange2) & same_profile_def ).any(2)).any(1)
        #
        return detection

```

```
    def doneFilter(self,Data):
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length = Data['length']
        prod = Data['prod']
        ternary = Data['ternary']
        #
        prodstats = prod.to(torch.int64).sum(3)
        #
        binsum = ternary.view(length,a3,2).to(torch.int64).sum(2)
        ternary_all = (binsum == 1).all(1)
        #
        #detection = ( ((prodstats == 1).all(2)).all(1) ) | ternary_all
        detection = ( ((prodstats == 1).all(2)).all(1) )
        #
        return detection

    def halfonesFilter(self,Data):  # only look at cases where the number of
        # ones on the left is >= the number on the right
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length = Data['length']
        prod = Data['prod']
        left = Data['left']
        right = Data['right']
        #
        prodstats = prod.to(torch.int64).sum(3)
        leftstats = left.to(torch.int64).sum(3)
        rightstats = right.to(torch.int64).sum(3)
        #
        assert (((leftstats <= 1).all(2)).all(1)).all(0)
        #
        leftones = (left[:,:,:,1].to(torch.int64).sum(2)).sum(1)
        right_isone = (rightstats == 1) & right[:,:,:,1]
        rightones = (right_isone.to(torch.int64).sum(2)).sum(1)
        #
        #
        detection = (rightones > leftones)
        #
        return detection

    def filterdata(self,Data):  #
        #
        #
        impossibledetect = self.impossibleFilter(Data)
```

```
        profiledetect = self.profileFilter(Data)
        #
        if self.pp.profile_filter_on:
            impossibledetect = impossibledetect | profiledetect
        #
        self.impossible_basic_count += impossibledetect.to(torch.int64).sum(0)
        # experimental:
        if self.pp.halfones_filter_on:
            halfonesdetect = self.halfonesFilter(Data)
            self.halfones_count += (halfonesdetect & (~impossibledetect)).to(tor
            #
            impossibledetect = impossibledetect | halfonesdetect
        #
        donedetect = self.doneFilter(Data)
        donedetect = donedetect & (~impossibledetect)
        #
        activedetect = (~impossibledetect) & ~donedetect
        #
        return activedetect, donedetect, impossibledetect
```

```
class Relations3 :
    def __init__(self,pp):
        #
        self.pp = pp
        #
        self.rr2 = Relations2(pp)
        self.rr1 = self.rr2.rr1
        #
        self.alpha = self.pp.alpha
        self.alpha2 = self.alpha * self.alpha
        self.alpha2z = self.alpha2 + 1
        self.alpha3 = self.alpha * self.alpha * self.alpha
        self.alpha3z = self.alpha3 + 1
        self.beta = self.pp.beta
        self.betaz = self.beta +1
        #


    def printmultiplicities(self,Data):
        #
        dlength = self.alpha2 + 1
        #
        length = Data['length']
        depth = Data['depth']
        if length == 0:
            multiplicities = torch.zeros((dlength),dtype = torch.int,device=Dvc)
            print(nump(multiplicities))
```

```
            return
        #
        dmax,dindices = torch.max(depth,0)
        if dmax +2 > dlength:
            dlength = dmax +2
        if dlength > 50:
            dlength = 50
        #
        depthvx = depth.view(length,1).expand(length,dlength)
        drangevx = arangeic(dlength).view(1,dlength).expand(length,dlength)
        rectangle = (depthvx == drangevx)
        multiplicities = rectangle.sum(0)
        #
        #print("dlength",itp(dlength))
        print("multiplicities in active pool by depth:")
        print(nump(multiplicities))
        return

    def selectchunk(self,Data):
        #
        length = Data['length']
        depth = Data['depth']
        prod = Data['prod']
        #
        #
        assert length > 0
        #
        prodstats = prod.to(torch.int).sum(3)
        assert (((prodstats >0).all(2)).all(1)).all(0)
        optional = (prodstats > 1)
        assert ((optional.any(2)).any(1)).all(0)
        #
        values,indices = torch.sort(depth,0,descending = True)
        upper = length
        if upper > self.pp.chunksize:
            upper = self.pp.chunksize
        indices_upper = indices[0:upper]
        #
        cdetection = torch.zeros((length),dtype = torch.bool,device=Dvc)
        cdetection[indices_upper] = True
        #
        ChunkData = self.rr1.detectsubdata(Data,cdetection)
        #
        if self.pp.verbose:
            self.printmultiplicities(Data)
        #
        return ChunkData, cdetection

    def network_vcuts(self,M,Data,randomize):
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
```

```
        #
        length = Data['length']
        depth = Data['depth']
        prod = Data['prod']
        #
        assert length > 0
        #
        #
        prodstats = prod.to(torch.int64).sum(3)
        #
        assert (((prodstats > 0).all(2)).all(1)).all(0)
        #
        #
        availablexyr = self.rr1.availablexy(length,prod).reshape(length*a2)
        #
        networkscorer = M.network2(Data).detach().reshape(length*a2)
        #
        if randomize:
            #
            epsilon_tirage = torch.rand(length*a2,device=Dvc)
            alll = torch.clamp(M.average_local_loss,0.,0.5)
            epsilon_factor = M. average_local_loss * self.pp.perturbation_factor
            epsilon = torch.rand(length*a2,device=Dvc)
            networkscorer += (epsilon_factor * epsilon)
            #
            phase = Data['info'][:,self.pp.phase]
            tirage = torch.rand(length,device=Dvc)
            detection = (phase == 1) | ( (tirage < 0.1) & (phase == 2) )
            detectionvxr = detection.view(length,1).expand(length,a2).reshape(le
            randomscore = torch.rand(length*a2,device=Dvc)
            networkscorer[detectionvxr] = randomscore[detectionvxr]
            #
        #
        networkscorer = torch.clamp(networkscorer, -1., 10.)
        networkscorer[~availablexyr] = 20.
        networkscore = networkscorer.view(length,a2)
        #
        values,xyvector = torch.min(networkscore,1)
        #
        return xyvector


    def addvalencies(self,availablexyp,xyvector):  # adds into the HST file the
        # also adds the passive count (just the length of the xyvector)
        #
        a = self.alpha
        a2 = self.alpha2
        a2z = self.alpha2 +1
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length = len(xyvector)
```

```
        HST.current_proof_passive_count += length
        #
        availablexypv = availablexyp.view(length,a2,bz)
        lrange = arangeic(length)
        available_cuts = availablexypv[lrange,xyvector]
        valency = available_cuts.to(torch.int64).sum(1)
        for v in range(bz + 1):
            HST.current_proof_valency_frequency[v] += (valency == v).to(torch.in
        return

    def managesplit(self,M,DataToSplit,randomize):
        #
        a = self.alpha
        a2 = self.alpha2
        a2z = self.alpha2 +1
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        #
        length = DataToSplit['length']
        prod = DataToSplit['prod']
        #
        availablexyp = self.rr1.availablexyp(length,prod).view(length,a2,bz)
        #
        xyvector = self.network_vcuts(M,DataToSplit,randomize)
        #
        self.addvalencies(availablexyp,xyvector)
        #
        lrangevxr = arangeic(length).view(length,1).expand(length,bz).reshape(le
        xyvectorvxr = xyvector.view(length,1).expand(length,bz).reshape(length*b
        bzrangevxr = arangeic(bz).view(1,bz).expand(length,bz).reshape(length*bz
        #
        verticaldetect = availablexyp[lrangevxr,xyvectorvxr,bzrangevxr]
        #
        #
        ivector_vert = lrangevxr[verticaldetect]
        xyvector_vert = xyvectorvxr[verticaldetect]
        pvector_vert = bzrangevxr[verticaldetect]
        #
        prx = arangeic(a).view(a,1).expand(a,a).reshape(a2)
        pry = arangeic(a).view(1,a).expand(a,a).reshape(a2)
        #
        xvector_vert = prx[xyvector_vert]
        yvector_vert = pry[xyvector_vert]
        #
        NewData = self.rr1.upsplitting(DataToSplit,ivector_vert,xvector_vert,yve
        #
        #
        ndlength = NewData['length']
        #
        #
        AssocNewData = self.rr1.nulldata()
        detection = torch.zeros((ndlength),dtype = torch.bool,device=Dvc)
```

```
        newactive = torch.zeros((ndlength),dtype = torch.bool,device=Dvc)
        newdone = torch.zeros((ndlength),dtype = torch.bool,device=Dvc)
        newimpossible = torch.zeros((ndlength),dtype = torch.bool,device=Dvc)
        lower = 0
        for i in range(ndlength):
            assert lower < ndlength
            upper = lower + 1000
            if upper > ndlength:
                upper = ndlength
            detection[:] = False
            detection[lower:upper] = True
            NewDataSlice = self.rr1.detectsubdata(NewData,detection)
            AssocNewDataSlice = self.rr2.process(NewDataSlice)
            AssocNewData = self.rr1.appenddata(AssocNewData,AssocNewDataSlice)
            newactive_s,newdone_s,newimpossible_s = self.rr2.filterdata(AssocNew
            newactive[lower:upper] = newactive_s
            newdone[lower:upper] = newdone_s
            newimpossible[lower:upper] = newimpossible_s
            lower = upper
            if lower >= ndlength:
                break
        #
        NewActiveData = self.rr1.detectsubdata(AssocNewData,newactive)
        #
        NewDoneData = self.rr1.detectsubdata(AssocNewData,newdone)
        #
        if NewActiveData['length'] > 0:
            phase1 = ( NewActiveData['info'][:,self.pp.phase] == 1)
            phase2 = ( NewActiveData['info'][:,self.pp.phase] == 2)
            tirage = torch.rand(NewActiveData['length'], device=Dvc)
            phasechange = phase2 & (tirage < self.pp.splitting_probability)
            newphase = NewActiveData['info'][:,self.pp.phase].clone()
            newphase[phase1] = 0
            newphase[phasechange] = 1
            NewActiveData['info'][:,self.pp.phase] = newphase
        #
        HST.current_proof_impossible_count += newimpossible.to(torch.int64).sum(
        HST.current_proof_done_count += newdone.to(torch.int64).sum(0)
        #
        if self.pp.verbose:
            print(" >>>")
            print("DataToSplit",itp(DataToSplit['length']))
            print("NewData",itp(NewData['length']))
            print("NewActiveData",itp(NewActiveData['length']))
            print("NewDoneData",itp(NewDoneData['length']))
            print("---------------------------------")
        #
        return NewActiveData, NewDoneData
```

```
class Relations4 :
    def __init__(self,pp):
        #
```

```
        #
        self.pp = pp
        #
        self.rr3 = Relations3(pp)
        self.rr2 = self.rr3.rr2
        self.rr1 = self.rr3.rr1
        #
        self.alpha = self.pp.alpha
        self.alpha2 = self.alpha * self.alpha
        self.alpha3 = self.alpha * self.alpha * self.alpha
        self.alpha3z = self.alpha3 + 1
        self.beta = self.pp.beta
        self.betaz = self.beta +1
        #
        self.prooflooplength = self.pp.prooflooplength
        self.done_max = 1000  # should be self.pp.done_max
        #
        self.sleeptime = self.pp.sleeptime
        self.periodicity = self.pp.periodicity
        self.stopthreshold = self.pp.stopthreshold
        #
        self.SamplePool = self.rr1.nulldata()
        self.DroppedSamplePool = self.rr1.nulldata()
        # these are by convention active (not done or impossible)
        #
        self.donecount = itt(0)
        self.ECN = 0.
        #
        self.proofnumber = 0
        self.allnumbers = 0
        self.proofinstance = 0
        self.dropoutratio = 1.

    def resetsamples(self):
        self.SamplePool = self.rr1.nulldata()
        self.DroppedSamplePool = self.rr1.nulldata()
        #
        self.rr2.impossible_basic_count = 0
        self.rr2.halfones_count = 0
        self.dropoutratio = 1.
        return

    def printexamples(self,Data):
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length = Data['length']
        depth = Data['depth']
        prod = Data['prod']
        ternary = Data['ternary']
        #
```

```
        if length == 0:
            print("length 0, no examples to print")
            return
        #
        bini = ternary.to(torch.int64)
        ternary_print = bini[:,:,0] + 2*bini[:,:,1] - 1
        #
        permutation = torch.randperm((length),device=Dvc)
        upper = 5
        if upper > length:
            upper = length
        for i in range(upper):
            indexi = permutation[i]
            print("----------------------------------------")
            self.rr1.printprod(Data,indexi)
            print(nump(quad[indexi]))
            print(nump(ternary_print[indexi]))
        print("----------------------------------------")



    def transitionactive(self,ActivePool,cdetection,NewActiveData):
        #
        ResidualActive = self.rr1.detectsubdata(ActivePool,~cdetection)
        NextActivePool = self.rr1.appenddata(NewActiveData,ResidualActive)
        #
        NextActivePoolCopy = self.rr1.copydata(NextActivePool)
        self.rr1.deletedata(NextActivePool)
        #
        return NextActivePoolCopy

    def transitiondone(self,C,DonePool,DoneData,aplength):
        #
        idl = DoneData['length']
        #
        self.donecount += idl
        #
        #
        if self.pp.verbose:
            print("new done count is",itp(self.donecount))
        #
        #
        #NewDonePool = self.rr1.nulldata()
        NewDonePool = self.rr1.appenddata(DonePool,DoneData)
        ndlength = NewDonePool['length']
        if ndlength > self.done_max:
            #print("new done pool of length",itp(ndlength),"so we send to classi
            print("/",end='')
            DataToProcess = self.rr1.copydata(NewDonePool)
            NewDonePool = self.rr1.nulldata()
            C.process(DataToProcess)
            #
        return NewDonePool


```

```
    def transitionsamples(self,ActivePool,DroppedPool):
        #
        # transitioning samples
        #
        slength = self.SamplePool['length']
        aplength = ActivePool['length']
        if aplength == 0:
            assert DroppedPool['length'] == 0
            return
        apdepth = ActivePool['depth'].to(torch.int64)
        #
        aprectangle = ActivePool['info'][:,self.pp.sampleinfolower:self.pp.sampl
        #
        aprange = arangeic(aplength)
        #
        # now add the new sample locations to the rectangle
        newsloc = arangeic(aplength) + slength
        #
        aprectangle[aprange,apdepth] = newsloc
        #
        # this should modify active pool outside the present function:
        ActivePool['info'][:,self.pp.sampleinfolower:self.pp.sampleinfoupper] =
        # that isn't needed for dropped pool since it doesn't get refered back t
        #
        self.SamplePool = self.rr1.appenddata(self.SamplePool, ActivePool)
        self.DroppedSamplePool = self.rr1.appenddata(self.DroppedSamplePool, Dro
        #
        #print("sample pool has size",itp(self.SamplePool['length']))
        #Fws.trace("transition samples Active Pool",ActivePool,5,0,0,0)
        #Fws.trace("transition samples Sample Pool",self.SamplePool,5,0,0,0)
        #self.printsampleex()
        return


    def proofloop(self,Mstrat,Mlearn,C,Input,dropoutlimit):
        #
        self.resetsamples()
        #
        if dropoutlimit > 0:
            randomize = True
        else:
            randomize = False
        #
        InitialActiveData = self.rr2.process(Input)
        activedetect, donedetect, impossibledetect = self.rr2.filterdata(Initial
        implength = impossibledetect.to(torch.int).sum(0)
        donelength = donedetect.to(torch.int).sum(0)
        if self.pp.verbose:
            print("initial filter finds",itp(implength),"impossibilities and",it
        ActivePool = self.rr1.detectsubdata(InitialActiveData,activedetect)
        #
        napcount = 0
        #
        DonePool = self.rr1.detectsubdata(InitialActiveData,donedetect)
        self.donecount = itt(0)
```

```
        if ActivePool['length'] == 0:
            DonePool = self.transitiondone(DonePool,self.rr1.nulldata(),ActivePo
        #
        #
        self.ECN = itt(ActivePool['length']).to(torch.float).clone()
        EDN = self.ECN.clone()
        if self.pp.verbose:
            print("starting with ECN = EDN from initial active pool",numpr(self.
        #
        if dropoutlimit > 0:
            ActivePool,DroppedPool,newsum,droppedsum = self.dropoutdata(Mlearn,A
            if self.pp.dropout_style == 'adaptive':
                activelengthf = itt(ActivePool['length']).clone().to(torch.float
                self.ECN = (activelengthf + droppedsum)
                EDN = 0.
        #
        stepcount = 0
        for i in range(self.proofloopLength):
            stepcount += 1
            prooflength = i
            if ActivePool['length'] > 0:
                #
                PreAPL = itt(ActivePool['length']).clone()
                #
                if self.pp.verbose:
                    print("= = = = = =  loop",i,"= = = = =",end=' ')
                    print(itp(self.proofnumber),"/",itp(self.allnumbers),"<",itp
                else:
                    print(".",end = '')
                    if (i%50) == 49:
                        print(" ")
                    if (i%100) == 0:
                        print(i)
                napcount += 1
                #
                ChunkData, cdetection = self.rr3.selectchunk(ActivePool)
                #
                #
                CurrentData, DoneData = self.rr3.managesplit(Mstrat,ChunkData,ra
                #
                #
                ActivePool = self.transitionactive(ActivePool,cdetection,Current
                # do the following before dropout
                if dropoutlimit == 0:
                    EDN = itt(ActivePool['length']).clone().to(torch.float)
                    self.ECN += itt(CurrentData['length']).clone().to(torch.floa
                    if self.ECN > HST.proof_nodes_max:
                        print("break after maximum proof nodes")
                        break
                #
                if dropoutlimit > 0:
                    if self.pp.dropout_style == 'regular' or self.pp.dropout_sty
                        PostAPL = itt(ActivePool['length']).clone().to(torch.flo
                        ratio = PostAPL / PreAPL
                        EDN *= ratio
                        self.ECN += EDN
```

```
                        self.ECN += EDN
                    ActivePool,DroppedPool,newsum, droppedsum = self.dropoutdata
                    #
                    self.transitionsamples(ActivePool,DroppedPool)
                    #
                    if self.pp.dropout_style == 'adaptive':
                        activelengthf = itt(ActivePool['length']).clone().to(tor
                        self.ECN += (activelengthf + droppedsum)
                        EDN = 0.
                    #
                    #
                    #
                    if self.pp.verbose:
                        self.printexamples(ActivePool)
                #
                #
                if self.pp.verbose:
                    print("Active Pool has length",itp(ActivePool['length']),end
                    #
                    print("treated Chunk Data of length",itp(ChunkData['length']
                    print("yielding Current Data of length",itp(CurrentData['len
                    print("net active data gained",itp(CurrentData['length'] - C
                    #
                    print("Active Pool has length",itp(ActivePool['length']),end
                DonePool = self.transitiondone(C,DonePool,DoneData,ActivePool['l
                #
                gcc = gc.collect()
                if self.pp.verbose:
                    memReport('mg')
                    #
                    #
                    if 0 < dropoutlimit <= self.pp.chunksize:
                        print("Estimated nodes at this depth",numpr(EDN,1),"esti
                    if dropoutlimit == 0:
                        print("current nodes",numpr(EDN,1),"cumulative nodes",nu
                #
                if ActivePool['length'] == 0:
                    break
                if ActivePool['length'] > self.stopthreshold:
                    print("over threshold --------->>>>>>>>>>>>>>>>> stopping")
                    break
                #
                #
            if ActivePool['length'] == 0:
                break
            if ActivePool['length'] > self.stopthreshold:
                print("over threshold --------->>>>>>>>>>>>>>>>> stopping")
                break
            #
            #if (napcount%self.periodicity) == 0:
                #siesta(self.sleeptime)
            #
        #
        print("|||")
        activelength = ActivePool['length']
        donelength = DonePool['length']
```

```
        donelength = DonePool['length']
        if donelength > 0:
            C.process(DonePool)
            DonePool = self.rr1.nulldata()
        #
        if dropoutlimit == 0:
            cumulative_nodes = torch.round(self.ECN).to(torch.int64)
            HST.record_full_proof(Mstrat,stepcount,cumulative_nodes,self.donecou
        else:
            HST.record_dropout_proof(self.pp.dropout_style,dropoutlimit,stepcoun
        #
        if self.pp.verbose:
            if activelength > 0:
                print("there remained",itp(activelength),"active locations", end
            else:
                print("no further active locations", end=' ')
            print("done pool has length",itp(donelength))
            print("Estimated Cumulative Nodes at end of proof",numpr(self.ECN,1)
            print("done count is",itp(self.donecount))
            print("impossible basic count is",itp(self.rr2.impossible_basic_coun
            print("half ones count is",itp(self.rr2.halfones_count))
        return True, ActivePool, DonePool, prooflength

    def dropoutdata(self,M,Data,dropoutlimit):
        if self.pp.dropout_style == 'regular':
            NewData,DroppedData = self.dropoutdataRegular(Data,dropoutlimit)
            newsum = 0.
            droppedsum = 0.
        if self.pp.dropout_style == 'adaptive':
            NewData,DroppedData,newsum,droppedsum = self.dropoutdataAdaptive(M,D
        if self.pp.dropout_style == 'uniform':
            NewData,DroppedData,newsum,droppedsum = self.dropoutdataUniform(M,Da
        #
        return NewData,DroppedData, newsum,droppedsum


    def dropoutdataRegular(self,Data,dropoutlimit):
        length = Data['length']
        if length == 0:
            return Data, self.rr1.nulldata()
        permutation = torch.randperm(length,device = Dvc)
        upper = dropoutlimit
        if upper > length:
            NewData = self.rr1.copydata(Data)
            DroppedData = self.rr1.nulldata()
        else:
            indices = permutation[0:upper]
            indices_dropped = permutation[upper:length]
            NewData = self.rr1.indexselectdata(Data,indices)
            DroppedData = self.rr1.indexselectdata(Data,indices_dropped)
        return NewData,DroppedData

    def extent_sliced(self,M,Data):
        #
        length = Data['length']
        if length <= 1000:
```

```
            extent_log = M.network(Data).detach()
            extent_log = torch.clamp(extent_log,0.,8.)
            extent = 10**extent_log
            return extent
        extent = torch.zeros((length),dtype = torch.float,device=Dvc)
        lrange = arangeic(length)
        #
        lower = 0
        for i in range(length):
            upper = lower + 1000
            if upper > length:
                upper = length
            indices = lrange[lower:upper]
            DataSlice = self.rr1.indexselectdata(Data,indices)
            extent_log = M.network(DataSlice).detach()
            extent_log = torch.clamp(extent_log,0.,8.)
            extent[lower:upper] = 10**extent_log
            lower = upper
            if upper >= length:
                break
        return extent

    def dropoutdataAdaptive(self,M,Data,dropoutlimit):
        length = Data['length']
        if length == 0:
            return Data, self.rr1.nulldata(),0.,0.
        #
        extent = self.extent_sliced(M,Data)
        denom = extent.sum(0)
        proba = (dropoutlimit * extent)/denom
        tirage = torch.rand(length,device=Dvc)
        detection = (tirage < proba)
        indices1 = arangeic(length)[detection]
        dlength = detection.to(torch.int64).sum(0)
        if dlength > dropoutlimit:
            indices1 = arangeic(length)[detection]
            permutation = torch.randperm(dlength)
            indices2 = permutation[dropoutlimit:dlength]
            indices3 = indices1[indices2]
            detection[indices3] = False
        if  length <= dropoutlimit:
            NewData = self.rr1.copydata(Data)
            DroppedData = self.rr1.nulldata()
            newsum = extent.sum(0)
            droppedsum = 0.
        else:
            NewData = self.rr1.detectsubdata(Data,detection)
            DroppedData = self.rr1.detectsubdata(Data,(~detection))
            newsum = (extent[detection]).sum(0)
            droppedsum = (extent[~detection]).sum(0)
        return NewData,DroppedData, newsum, droppedsum


    def dropoutdataUniform(self,M,Data,dropoutlimit):
        length = Data['length']
```

```
        if length == 0:
            return Data, self.rr1.nulldata(),0.,0.
        #
        extent = self.extent_sliced(M,Data)
        #
        values,sort_indices = torch.sort(extent,0)
        fraction = itf(length) / itf(dropoutlimit)
        epsilon_multiplier = itf(length - dropoutlimit) / itf(length)
        epsilon_multiplier = torch.clamp(epsilon_multiplier,0.,1.)
        drange = arangeic(dropoutlimit).to(torch.float)
        tirage = torch.rand(dropoutlimit,device=Dvc)
        tirage2 = torch.rand(dropoutlimit,device=Dvc) - 0.5
        tirage2vx = tirage2.view(dropoutlimit,1).expand(dropoutlimit,dropoutlimi
        irange = arangeic(dropoutlimit).view(dropoutlimit,1).expand(dropoutlimit
        jrange = arangeic(dropoutlimit).view(1,dropoutlimit).expand(dropoutlimit
        tirage2_triangle = tirage2vx * ( (irange > jrange).to(torch.float) )
        tirage2_integral = tirage2_triangle.sum(0)
        epsilon = tirage * epsilon_multiplier
        drange_mod = drange + epsilon + (0.05 * tirage2_integral)
        #
        float_indices = drange_mod * fraction
        round_indices = torch.round(float_indices).to(torch.int64)
        round_indices = torch.clamp(round_indices,0,length-1)
        combined_indices = sort_indices[round_indices]
        detection = torch.zeros((length),dtype = torch.bool,device=Dvc)
        detection[combined_indices] = True
        #
        if  length <= dropoutlimit:
            NewData = self.rr1.copydata(Data)
            DroppedData = self.rr1.nulldata()
            newsum = extent.sum(0)
            droppedsum = 0.
        else:
            NewData = self.rr1.detectsubdata(Data,detection)
            DroppedData = self.rr1.detectsubdata(Data,(~detection))
            newsum = (extent[detection]).sum(0)
            droppedsum = (extent[~detection]).sum(0)
        return NewData,DroppedData, newsum, droppedsum



    def printsampleex(self):
        samplelength = self.SamplePool['length']
        if samplelength == 0:
            return
        upper = 20
        if upper > samplelength:
            upper = samplelength
        permutation = torch.randperm(samplelength,device=Dvc)
        depth = self.SamplePool['depth']
        points = self.SamplePool['info'][:,self.pp.samplepoints]
        for i in range(upper):
            ip = permutation[i]
            print("number",ip,"depth",itp(depth[ip]),"points",itp(points[ip]))
        return
```

```
445
446
447
448
449
450
```

```
class Classifier :    # this is a very first part of classification up to isomor
    def __init__(self,P):
        #
        #
        self.Pp = P
        self.rr4 = Relations4(self.Pp)
        self.rr3 = self.rr4.rr2
        self.rr2 = self.rr4.rr2
        self.rr1 = self.rr4.rr1
        self.alpha = self.Pp.alpha
        self.alpha2 = self.Pp.alpha2
        self.alpha3 = self.Pp.alpha3
        self.alpha3z = self.Pp.alpha3z
        self.beta = self.Pp.beta
        self.betaz = self.Pp.betaz
        #
        self.sga = SymmetricGroup(self.alpha)
        #
        self.eqlength = 0
        self.eqlist = None
        #
        #self.iota = 24*24*4
        self.matrix = self.choosematrix()
        self.ilength = 10
        self.indices1,self.indices2,self.indices3,self.indices4,self.vector=self


    def initialize(self):
        self.eqlength = 0
        self.eqlist = None
        return

    def choosestuff(self,ilength):
        assert ilength <= 48
        permutation1 = torch.randperm(48,device = Dvc)
        permutation2 = torch.randperm(48,device = Dvc)
        permutation3 = torch.randperm(48,device = Dvc)
        permutation4 = torch.randperm(48,device = Dvc)
        #
        indices1 = permutation1[0:ilength]
        indices2 = permutation2[0:ilength]
        indices3 = permutation3[0:ilength]
        indices4 = permutation4[0:ilength]
        #
        permutationV = torch.randperm(101,device = Dvc)
        vector = permutationV[0:ilength] - 50
        return indices1,indices2,indices3,indices4,vector
```

```
48
49      def choosematrix(self):
50          #
51          a = self.alpha
52          #
53          #thematrix = torch.zeros((a,a,a,a),dtype = torch.int64,device=Dvc)
54          for i in range(10):
55              permutation = torch.randperm((a*a*a*a),device=Dvc)
56              psquared = permutation*permutation
57              thematrix = psquared.view(a,a,a,a)
58          return thematrix
59
60      def geteq(self,Data):
61          #
62          a = self.alpha
63          a2 = self.alpha2
64          a3 = self.alpha3
65          a4 = self.alpha2 * self.alpha2
66          a3z = self.alpha3z
67          b = self.beta
68          bz = self.betaz
69          #
70          length = Data['length']
71          prod = Data['prod']
72          #
73          prodsum = prod.to(torch.int64).sum(3)
74          #
75          assert (((prodsum == 1).all(2)).all(1)).all(0)
76          #
77          values,table = torch.max(prod.to(torch.int64),3)
78          #
79          table1vx = table.view(length,a,a,1,1).expand(length,a,a,a,a)
80          table2vx = table.view(length,1,1,a,a).expand(length,a,a,a,a)
81          eq = (table1vx == table2vx)
82          iz = (table == b).view(length,a,a)
83          return eq,iz
84
85
86
87      def orderinvariantSlice(self,Data):
88          #
89          a = self.alpha
90          a2 = self.alpha2
91          a3 = self.alpha3
92          a4 = self.alpha2 * self.alpha2
93          a3z = self.alpha3z
94          b = self.beta
95          bz = self.betaz
96          #
97          length = Data['length']
98          #print("at start of orderinvariantSlice, length is",length)
99          prod = Data['prod']
100         #
101         eq,iz = self.geteq(Data)
102         #
103         eqview = eq.view(1,length,a,a,a,a)
```

```
        eqview = eq.view(1,length,a,a,a,a)
        izview = iz.view(1,length,a,a,1,1).expand(1,length,a,a,a,a)
        eqiz = torch.cat((eqview,izview),0)
        eqiz_p2 = eqiz.permute(0,1,2,3,5,4)
        eqiz_p3 = eqiz.permute(0,1,2,4,3,5)
        eqiz_p4 = eqiz.permute(0,1,2,4,5,3)
        eqiz_p5 = eqiz.permute(0,1,2,5,3,4)
        eqiz_p6 = eqiz.permute(0,1,2,5,4,3)
        #
        eqiz_c1 = torch.cat((eqiz,eqiz_p2,eqiz_p3,eqiz_p4,eqiz_p5,eqiz_p6),0)
        eqiz_c2 = eqiz_c1.permute(0,1,3,4,5,2)
        eqiz_c3 = eqiz_c1.permute(0,1,4,5,2,3)
        eqiz_c4 = eqiz_c1.permute(0,1,5,2,3,4)
        #
        eqiz_cat = torch.cat((eqiz_c1,eqiz_c2,eqiz_c3,eqiz_c4),0)  # size 48.len
        #
        eqiz_cat1 = eqiz_cat[self.indices1]
        eqiz_cat2 = eqiz_cat[self.indices2]
        eqiz_cat3 = eqiz_cat[self.indices3]
        eqiz_cat4 = eqiz_cat[self.indices4]
        #
        eqiz_andor = (eqiz_cat1 | eqiz_cat2)  & (eqiz_cat3 | eqiz_cat4)
        #
        eqiz_sum = ((eqiz_andor.to(torch.int64).sum(5)).sum(4)).sum(3)
        #
        vectorvx = self.vector.view(self.ilength,1,1).expand(self.ilength,length
        invariant = (eqiz_sum * vectorvx).sum(0)
        #
        return invariant

    def orderinvariant(self,Data):
        #
        a = self.alpha
        #
        length = Data['length']
        #
        lrange = arangeic(length)
        #
        invariant = torch.zeros((length,a),dtype = torch.int64,device=Dvc)
        #
        lower = 0
        for i in range(length):
            upper = lower + 100
            if upper > length:
                upper = length
            indices = lrange[lower:upper]
            DataSlice = self.rr1.indexselectdata(Data,indices)
            invariant[lower:upper] = self.orderinvariantSlice(DataSlice)
            lower = upper
            if lower >= length:
                break
        return invariant



    def printtestinvariant(self,length,invariant):
```

```
        #
        a = self.alpha
        #
        assert length > 0
        assert a > 1
        #
        invariant_sorted,indices = torch.sort(invariant,1)
        #
        invariant_pre = invariant_sorted[:,0:a-1]
        invariant_post = invariant_sorted[:,1:a]
        invariant_delta = invariant_post - invariant_pre
        #
        assert ((invariant_delta >= 0).all(1)).all(0)
        delta_avg = (invariant_delta.sum(1)).sum(0).to(torch.float) / (itf(lengt
        #
        delta_sum = (invariant_delta == 0).to(torch.int64).sum(1)
        print("invariant delta average is",numpr(delta_avg,1),"for length",itp(l
        for k in range(a-1):
            locnumber = (delta_sum == k).to(torch.int64).sum(0)
            print("number of locations with delta sum",itp(k),"is",itp(locnumber
        return


    def matrixinvariant(self,Data,ivector,gvector):
        #
        a = self.alpha
        #
        length = Data['length']
        #
        eq,iz = self.geteq(Data)
        #
        vlength = len(ivector)
        assert len(gvector) == vlength
        #
        irangevx = ivector.view(vlength,1,1,1,1).expand(vlength,a,a,a,a)
        grangevx = gvector.view(vlength,1,1,1,1).expand(vlength,a,a,a,a)
        xrangevx = arangeic(a).view(1,a,1,1,1).expand(vlength,a,a,a,a)
        yrangevx = arangeic(a).view(1,1,a,1,1).expand(vlength,a,a,a,a)
        zrangevx = arangeic(a).view(1,1,1,a,1).expand(vlength,a,a,a,a)
        wrangevx = arangeic(a).view(1,1,1,1,a).expand(vlength,a,a,a,a)
        #
        xtransform = self.sga.grouptable[grangevx,xrangevx]
        ytransform = self.sga.grouptable[grangevx,yrangevx]
        ztransform = self.sga.grouptable[grangevx,zrangevx]
        wtransform = self.sga.grouptable[grangevx,wrangevx]
        #
        eq_transform = eq[irangevx,xtransform,ytransform,ztransform,wtransform]
        #
        matrixvx = self.matrix.view(1,a,a,a,a).expand(vlength,a,a,a,a)
        matrix_invariant = ((eq_transform.to(torch.int64)) * matrixvx).view(vlen
        #
        return matrix_invariant

    def to_eqfunction(self,length,eq,iz):
        #
```

```
        a = self.alpha
        a2 = a*a
        #
        eqview = eq.view(length*a2,a2)
        izview = iz.view(length*a2)
        #
        numerical = arangeic(a2).view(1,a2).expand(length*a2,a2) + 1
        numerical_eq = numerical * (eqview.to(torch.int64))
        #
        values,eqfunctionv = torch.max(numerical_eq,1)
        eqfunctionv[izview] = a2
        eqfunction = eqfunctionv.view(length,a2)
        return eqfunction

    def from_eqfunction(self,length,eqfunction):
        #
        #
        a = self.alpha
        a2 = a*a
        #
        eqf1vx = eqfunction.view(length,a2,1).expand(length,a2,a2)
        eqf2vx = eqfunction.view(length,1,a2).expand(length,a2,a2)
        #
        eq = (eqf1vx == eqf2vx)
        #
        iz = (eqfunction.view(length,a2) == a2)
        #
        return eq,iz

    def transform_eqfunction(self,length,eq,iz,gvector):
        #
        a = self.alpha
        a2 = a*a
        #
        assert len(gvector)==length
        #
        eqv = eq.view(length,a,a,a,a)
        izv = iz.view(length,a,a)
        #
        irangevx = arangeic(length).view(length,1,1,1,1).expand(length,a,a,a,a)
        grangevx = gvector.view(length,1,1,1,1).expand(length,a,a,a,a)
        xrangevx = arangeic(a).view(1,a,1,1,1).expand(length,a,a,a,a)
        yrangevx = arangeic(a).view(1,1,a,1,1).expand(length,a,a,a,a)
        zrangevx = arangeic(a).view(1,1,1,a,1).expand(length,a,a,a,a)
        wrangevx = arangeic(a).view(1,1,1,1,a).expand(length,a,a,a,a)
        #
        xtransform = self.sga.grouptable[grangevx,xrangevx]
        ytransform = self.sga.grouptable[grangevx,yrangevx]
        ztransform = self.sga.grouptable[grangevx,zrangevx]
        wtransform = self.sga.grouptable[grangevx,wrangevx]
        #
        eq_transform = eqv[irangevx,xtransform,ytransform,ztransform,wtransform]
        #
        irangeZvx = arangeic(length).view(length,1,1).expand(length,a,a)
        grangeZvx = gvector.view(length,1,1).expand(length,a,a)
```

```
        xrangeZvx = arangeic(a).view(1,a,1).expand(length,a,a)
        yrangeZvx = arangeic(a).view(1,1,a).expand(length,a,a)
        #
        xtransformZ = self.sga.grouptable[grangeZvx,xrangeZvx]
        ytransformZ = self.sga.grouptable[grangeZvx,yrangeZvx]
        #
        iz_transform = izv[irangeZvx,xtransformZ,ytransformZ].view(length,a2)
        #
        eqfunction_transform = self.to_eqfunction(length,eq_transform,iz_transfo
        return eqfunction_transform



    def data_eqfunction_transform(self,Data,ivector,gvector):
        #
        a = self.alpha
        a2 = a*a
        #
        DataVector = self.rr1.indexselectdata(Data,ivector)
        length = DataVector['length']
        #
        eq,iz = self.geteq(DataVector)
        #
        assert len(gvector) == length
        #
        #eqfunction = self.to_eqfunction(length,eq,iz)
        #
        eqfunction_transform = self.transform_eqfunction(length,eq,iz,gvector)
        #
        return eqfunction_transform

    def uniqueinstances(self,length,eq_function):
        #
        a = self.alpha
        a2= a*a
        #
        assert length > 0
        #
        eqf1vx = eq_function.view(length,1,a2).expand(length,length,a2)
        eqf2vx = eq_function.view(1,length,a2).expand(length,length,a2)
        #
        equivalent = (eqf1vx == eqf2vx).all(2)
        #
        first = arangeic(length).view(length,1).expand(length,length)
        second = arangeic(length).view(1,length).expand(length,length)
        #
        isrep = ((~equivalent) | (first <= second)).all(1)
        #
        unique_length = isrep.to(torch.int64).sum(0)
        eq_unique = eq_function[isrep]
        #
        assert unique_length > 0
        #
        return unique_length,eq_unique

```

```
    def addSlice(self,length,eq_function):
        #
        a = self.alpha
        a2= a*a
        #
        ulength,eq_unique = self.uniqueinstances(length,eq_function)
        #
        alength = self.eqlength
        #
        if alength == 0:
            self.eqlist = eq_unique
            self.eqlength = ulength
            return
        alreadyvx = self.eqlist.view(1,alength,a2).expand(ulength,alength,a2)
        newvx = eq_unique.view(ulength,1,a2).expand(ulength,alength,a2)
        #
        already_known = ( (alreadyvx == newvx).all(2) ).any(1)
        #
        detection = ~already_known
        detected_length = detection.to(torch.int64).sum(0)
        eq_detected = eq_unique[detection]
        #
        #
        self.eqlist = torch.cat((self.eqlist,eq_detected),0)
        #
        self.eqlength += detected_length
        #
        return

    def addinstances(self,length,eq_function):
        #
        if length == 0:
            #print("no instances to add")
            return
        #
        lower = 0
        for i in range(length):
            upper = lower + 500
            if upper > length:
                upper = length
            length_slice = upper - lower
            eq_slice = eq_function[lower:upper]
            self.addSlice(length_slice,eq_slice)
            lower = upper
            if lower >= length:
                break
        return

    def processBasic(self,Data):
        #
        a = self.alpha
        assert a > 1
        #
        length = Data['length']
```

```
        length = Data['length']
        print("current eqlength",itp(self.eqlength),"processing data of length",
        #
        eq,iz = self.geteq(Data)
        #
        eq_function = self.to_eqfunction(length,eq,iz)
        #
        self.addinstances(length,eq_function)
        #
        return

    def process(self,Data):
        #
        a = self.alpha
        assert a > 1
        #
        length = Data['length']
        gl = self.sga.gtlength
        #
        invariant = self.orderinvariant(Data)
        #
        lrangevxr = arangeic(length).view(length,1,1).expand(length,gl,a).reshap
        gtvxr =self.sga.grouptable.view(1,gl,a).expand(length,gl,a).reshape(leng
        #
        invariant_gt = invariant[lrangevxr,gtvxr]
        detection = ((invariant_gt[:,0:a-1]) <= (invariant_gt[:,1:a])).all(1)
        dlength = detection.to(torch.int64).sum(0)
        ivector = (arangeic(length).view(length,1).expand(length,gl).reshape(len
        gvector = (arangeic(gl).view(1,gl).expand(length,gl).reshape(length*gl))
        #
        verification = torch.zeros((length),dtype = torch.bool,device=Dvc)
        verification[ivector] = True
        #
        assert verification.all(0)
        #
        eq_function = self.data_eqfunction_transform(Data,ivector,gvector)
        #
        self.addinstances(dlength,eq_function)
        #
        return

    def checklocationSlice(self,length,eq_function):
        #
        a = self.alpha
        a2 = a*a
        #
        location = torch.zeros((length),dtype = torch.int64,device=Dvc)
        location[:] = -1
        #
        alength = self.eqlength
        #
        predetect_already = (self.eqlist.view(alength,a2)).sum(1)
        predetect_current = (eq_function.view(length,a2)).sum(1)
        predetect_already_vxr = predetect_already.view(alength,1).expand(alength
        predetect_current_vxr = predetect_current.view(1,length).expand(alength,
        #
```

```
        #
        predetection = (predetect_already_vxr == predetect_current_vxr)
        #
        alrangevxr = arangeic(alength).view(alength,1).expand(alength,length).re
        lrangevxr = arangeic(length).view(1,length).expand(alength,length).resha
        #
        alrvector = alrangevxr[predetection]
        currvector = lrangevxr[predetection]
        #
        eq_already = (self.eqlist.view(alength,a2))[alrvector]
        eq_current = (eq_function.view(length,a2))[currvector]
        #
        detection = (eq_already == eq_current).all(1)
        #
        already_detected = alrvector[detection]
        current_detected = currvector[detection]
        #
        location[current_detected] = already_detected
        #
        return location

    def checklocation(self,length,eq_function):
        #
        a = self.alpha
        a2 = a*a
        #
        location = torch.zeros((length),dtype = torch.int64,device=Dvc)
        #
        lower = 0
        for i in range(length):
            upper = lower + 100
            if upper > length:
                upper = length
            length_slice = upper - lower
            eq_slice = eq_function[lower:upper]
            location[lower:upper] = self.checklocationSlice(length_slice,eq_slic
            lower = upper
            if lower >= length:
                break
        return location




    def highestlocation(self,length,eq_function):
        #
        a = self.alpha
        a2 = a*a
        #
        gl = self.sga.gtlength
        #
        eq_vxr = eq_function.view(length,1,a2).expand(length,gl,a2).reshape(leng
        gvector = arangeic(gl).view(1,gl).expand(length,gl).reshape(length*gl)
        grouplength = length*gl
        #
        eq_iz = self.from_eqfunction(grouplength,eq_vxr)
```

```
        eq_transform = self.transform_eqfunction(grouplength,eq,iz,gvector)
        #
        location = self.checklocation(grouplength,eq_transform)
        locationv = location.view(length,gl)
        highest,indices = torch.max(locationv,1)
        return highest

    def sieve(self):
        #
        alength = self.eqlength
        print("doing sieve on",itp(alength),"locations")
        #
        highest_location = torch.zeros((alength),dtype = torch.int64,device=Dvc)
        #
        lower = 0
        for i in range(alength):
            print("---",lower)
            upper = lower + 100
            if upper > alength:
                upper = alength
            eq_slice = self.eqlist[lower:upper]
            length_slice = upper - lower
            #
            highest_location[lower:upper] = self.highestlocation(length_slice,eq
            lower = upper
            if lower >= alength:
                break
        #
        negative_detect = (highest_location < 0)
        negative_count = negative_detect.to(torch.int64).sum(0)
        print("unfortunate locations with negative values :",itp(negative_count)
        alrange = arangeic(alength)
        good_detect = (highest_location <= alrange)
        good_count = good_detect.to(torch.int64).sum(0)
        print("detected",itp(good_count),"good locations")
        print("finished sieve")
        return
```

```
class Learner :    # training the neural networks
    def __init__(self,Rr4):
        #
        #
        self.rr4 = Rr4
        self.pp = self.rr4.pp
        self.rr3 = self.rr4.rr3
        self.rr2 = self.rr4.rr2
        self.rr1 = self.rr4.rr1
        self.alpha = self.pp.alpha
        self.alpha2 = self.pp.alpha2
        self.alpha3 = self.pp.alpha3
        self.alpha3z = self.pp.alpha3z
```

```
        self.beta = self.pp.beta
        self.betaz = self.pp.betaz
        #
        #self.mm = Mm
        self.explore_max = self.pp.explore_max
        self.examples_max = self.pp.examples_max
        self.new_examples_max = self.pp.new_examples_max
        self.new_explore_max = self.pp.new_explore_max
        self.outlier_max = self.pp.outlier_max
        self.new_outliers_max = self.pp.new_outliers_max
        #
        self.trainingprint = True
        #
        # all the examples are by convention active (not done or impossible)
        self.OutlierPrePool = self.rr1.nulldata()
        self.ExplorePrePool = self.rr1.nulldata()
        self.ExamplesPrePool = self.rr1.nulldata()
        self.Examples = self.rr1.nulldata()
        self.extent_log = None  # log10 of the number of nodes below that node,
        # so this number of nodes is never 0
        self.localscores = None  # at (x,y) it is log10 of the sum score over (x
        #
        #self.sga = SymmetricGroup(self.alpha)
        #self.sgb = SymmetricGroup(self.beta)
        #
        self.globalL1level = 1.0


    def pruneExamples(self,M): # remove the last ones, those were the earliest a
        #
        elength = self.Examples['length']
        epplength = self.ExamplesPrePool['length']
        xlength = self.ExplorePrePool['length']
        olength = self.OutlierPrePool['length']
        #
        if elength > self.examples_max:
            #self.specialPruneExamples(M)
            permutation = torch.randperm(elength,device=Dvc)
            indices = permutation[0:self.examples_max]
            self.Examples = self.rr1.indexselectdata(self.Examples,indices)
            self.extent_log = self.extent_log[indices]
            self.localscores = self.localscores[indices]
        if epplength > self.examples_max:
            permutation = torch.randperm(epplength,device=Dvc)
            indices = permutation[0:self.examples_max]
            self.ExamplesPrePool = self.rr1.indexselectdata(self.ExamplesPrePool
        if xlength > self.explore_max:
            permutation = torch.randperm(xlength,device=Dvc)
            indices = permutation[0:self.explore_max]
            self.ExplorePrePool = self.rr1.indexselectdata(self.ExplorePrePool,i
        if olength > self.outlier_max:
            permutation = torch.randperm(olength,device=Dvc)
            indices = permutation[0:self.outlier_max]
            self.OutlierPrePool = self.rr1.indexselectdata(self.OutlierPrePool,i
        return
```

```
69
70
71      def check_availability(self,Data,comment):
72          #
73          prod = Data['prod']
74          #
75          available = (((prod.any(3)).all(2)).all(1)).all(0)
76          if not available:
77              print("with comment",comment)
78              raise CoherenceError("availability problem in data that should be ac
79          return
80
81
82
83      def prepoolSamples(self,M,Data,new_examples): # prepend new ones so that the
84          #
85          #
86          splength = Data['length']
87          #
88          if splength == 0:
89              return
90          permutation = torch.randperm(splength,device=Dvc)
91          upper = new_examples
92          if upper > splength:
93              upper = splength
94          indices = permutation[0:upper]
95          NewExamples = self.rr1.indexselectdata(Data,indices)
96          self.check_availability(NewExamples,"prepoolSamples")
97          self.ExamplesPrePool = self.rr1.appenddata(NewExamples,self.ExamplesPreP
98          #
99          self.pruneExamples(M)
100         return
101
102     def prepoolExplore(self,M,Data,new_examples): # prepend new ones so that the
103         #
104         #
105         splength = Data['length']
106         #
107         if splength == 0:
108             return
109         ppe_length = torch.round(itf(new_examples)/M.average_local_loss).to(torc
110         rectangle = arangeic(splength).view(splength,1).expand(splength,ppe_leng
111         permutation = torch.randperm(splength*ppe_length,device=Dvc)
112         upper = ppe_length
113         #
114         indices = rectangle[permutation[0:upper]]
115         NewExamples = self.rr1.indexselectdata(Data,indices)
116         self.check_availability(NewExamples,"prepoolExplore")
117         self.ExplorePrePool = self.rr1.appenddata(NewExamples,self.ExplorePrePoo
118         #
119         self.pruneExamples(M)
120         return
121
122     def transferexamples(self,M,TransferData,extent,localscore,epp_throw_detecti
123         # detection tells us the ones to remove from the epp
124         if self.Examples['length'] == 0:
```

```
        if self.Examples['length'] == 0:
            self.extent_log = extent
            self.localscores = localscore
        else:
            self.extent_log = torch.cat((extent,self.extent_log),0)
            self.localscores = torch.cat((localscore,self.localscores),0)
        self.Examples = self.rr1.appenddata(TransferData,self.Examples)
        #
        keep_detection = ~epp_throw_detection
        self.ExamplesPrePool = self.rr1.detectsubdata(self.ExamplesPrePool,keep_
        #
        delta = self.abs_diff_sup(M,TransferData,localscore)
        seuil = self.pp.outlier_threshold * M.average_local_loss
        NewOutliers = self.rr1.detectsubdata(TransferData,(delta > seuil))
        self.OutlierPrePool = self.rr1.appenddata(NewOutliers,self.OutlierPrePoo
        #
        self.pruneExamples(M)
        return

    def scoreExamples(self,M):
        epplength = self.ExamplesPrePool['length']
        if epplength == 0:
            print("no examples to locally score")
            return
        #
        permutation = torch.randperm(epplength,device=Dvc)
        upper = self.new_examples_max
        if upper > epplength:
            upper = epplength
        indices = permutation[0:upper]
        #
        epp_throw_detection = torch.zeros((epplength),dtype = torch.bool,device=
        epp_throw_detection[indices] = True
        #
        TransferData = self.rr1.indexselectdata(self.ExamplesPrePool,indices)
        #
        print("transfering data of length",itp(TransferData['length']))
        #
        xyscore_log, xyscore_min, LocalExamples = self.calculatescoresLocal(M,Tr
        #
        self.transferexamples(M,TransferData,xyscore_min,xyscore_log,epp_throw_d
        #
        self.check_availability(LocalExamples,"LocalExamples in scoreExamples")
        self.prepoolExplore(M,LocalExamples,self.new_examples_max)
        #
        print("examples has size",itp(self.Examples['length']))
        print("example pre pool has size",itp(self.ExamplesPrePool['length']))
        return

    def scoreExplore(self,M):
        xlength = self.ExplorePrePool['length']
        epplength = self.ExamplesPrePool['length']
        if xlength == 0:
            print("no examples to locally score")
            return
        #
```

```
        #
        permutation = torch.randperm(xlength,device=Dvc)
        current_explore = 2*self.new_examples_max
        if current_explore > self.new_explore_max:
            current_explore = self.new_explore_max
        upper = current_explore
        if upper > xlength:
            upper = xlength
        indices = permutation[0:upper]
        #
        epp_throw_detection = torch.zeros((epplength),dtype = torch.bool,device=
        #
        TransferData = self.rr1.indexselectdata(self.ExplorePrePool,indices)
        #
        print("transfering explore data of length",itp(TransferData['length']))
        #
        xyscore_log, xyscore_min, LocalExamples = self.calculatescoresLocal(M,Tr
        #
        self.transferexamples(M,TransferData,xyscore_min,xyscore_log,epp_throw_d
        #
        self.check_availability(LocalExamples,"LocalExamples in scoreExplore")
        self.prepoolExplore(M,LocalExamples,current_explore)
        #
        print("examples has size",itp(self.Examples['length']))
        print("explore pre pool has size",itp(self.ExplorePrePool['length']))
        return

    def scoreOutlier(self,M):
        xlength = self.OutlierPrePool['length']
        epplength = self.ExamplesPrePool['length']
        if xlength == 0:
            print("no outliers to locally score")
            return
        #
        permutation = torch.randperm(xlength,device=Dvc)
        #
        upper = self.new_outliers_max
        if upper > xlength:
            upper = xlength
        indices = permutation[0:upper]
        #
        epp_throw_detection = torch.zeros((epplength),dtype = torch.bool,device=
        #
        TransferData = self.rr1.indexselectdata(self.OutlierPrePool,indices)
        #
        print("transfering outlier data of length",itp(TransferData['length']))
        #
        xyscore_log, xyscore_min, LocalExamples = self.calculatescoresLocal(M,Tr
        #
        delta = self.abs_diff_sup(M,TransferData,xyscore_log)
        seuil = self.pp.outlier_threshold * M.average_local_loss
        indices_throw = indices[(delta < seuil)]
        throw_detection = torch.zeros((xlength),dtype = torch.bool,device=Dvc)
        throw_detection[indices_throw] = True
        self.OutlierPrePool = self.rr1.detectsubdata(self.OutlierPrePool,(~throw
        #
```

```
        self.transferexamples(M,TransferData,xyscore_min,xyscore_log,epp_throw_d
        #
        print("outlier pre pool has size",itp(self.OutlierPrePool['length']))
        return

    def abs_diff_sup(self,M,Data,xyscore_log):
        #
        a = self.alpha
        a2 = self.alpha2
        #
        length = Data['length']
        prod = Data['prod']
        #
        availablexyf = self.rr1.availablexy(length,prod).view(length,a,a)
        net2 = M.network2(Data)
        predicted = net2.view(length,a,a).detach()
        delta_abs = torch.abs((availablexyf * (xyscore_log - predicted))).view(l
        delta_sup,indices = torch.max(delta_abs,1)
        return delta_sup


    def addscoredexamples(self,M):
        SamplePool = self.rr4.SamplePool
        DroppedPool = self.rr4.DroppedSamplePool
        #
        splength = SamplePool['length']
        dplength = DroppedPool['length']
        #
        sprectangle = SamplePool['info'][:,self.pp.sampleinfolower:self.pp.sampl
        #
        splrangevxr = arangeic(splength).view(splength,1).expand(splength,200).r
        sprectangler = sprectangle.reshape(splength*200)
        sp_detection = (sprectangler >= 0) & (sprectangler < splrangevxr)
        #
        ivector = splrangevxr[sp_detection]
        jvector = sprectangler[sp_detection]
        # note that jvector < ivector because of the second condition in sp_dete
        # jvector is any previous location strictly above the ivector location i
        #
        incidence = torch.zeros((splength,splength),dtype = torch.float,device=D
        #
        incidence[ivector,jvector] = 1.
        #
        #
        spnodes = incidence.sum(0) + 1. # this should be the number of nodes bel
        #
        if dplength > 0:
            dprectangle = DroppedPool['info'][:,self.pp.sampleinfolower:self.pp.
            #
            dincidence = torch.zeros((dplength,splength),dtype = torch.float,dev
            #
            dplrangevxr = arangeic(dplength).view(dplength,1).expand(dplength,20
            dprectangler = dprectangle.reshape(dplength*200)
            dp_detection = (dprectangler >= 0)
            #
```

```
            idvector = dplrangevxr[dp_detection]
            jdvector = dprectangler[dp_detection]
            #
            dincidence[idvector,jdvector] = 1.
            #
            #
            dpextent_log = M.network(DroppedPool).detach()
            # this approximates log10 of (the number of nodes at or below a drop
            dpextent_log = torch.clamp(dpextent_log,0.,9.)
            dpextent = (10**dpextent_log)
            dpextentv = dpextent.view(1,dplength)
            extent_transfer = (torch.matmul(dpextentv,dincidence)).view(splength
            #
            extent = spnodes + extent_transfer
        else:
            extent = spnodes
        #
        logextent = torch.log10(extent)
        #
        #
        permutation = torch.randperm(splength,device=Dvc)
        upper = self.new_examples_max
        if upper > splength:
            upper = splength
        indices = permutation[0:upper]
        TransferData = self.rr1.indexselectdata(SamplePool,indices)
        transfer_extent = logextent[indices]
        #
        epp_throw_detection = torch.zeros((self.ExamplesPrePool['length']),dtype
        #
        xyscore_log, xyscore_min, LocalExamples = self.calculatescoresLocal(M,Tr
        #
        positivephase = ( TransferData['info'][:,self.pp.phase] > 0 )
        phased_extent = transfer_extent.clone()
        phased_extent[positivephase] = xyscore_min[positivephase]
        #
        self.transferexamples(M,TransferData,phased_extent,xyscore_log,epp_throw
        #
        return



    def noisetensor(self,thetensor):
        #
        length = len(thetensor)
        tirage = torch.rand(length,device=Dvc)
        noiselevel = HST.noiselevel(self.pp,torch.tensor(HST.training_counter,de
        bruit = (tirage < noiselevel)
        ntensor = ((~bruit) & thetensor) | (bruit & (~thetensor))
        return ntensor


    def noise(self,Data):
        #
        a = self.alpha
```

```
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length = Data['length']
        prodv = Data['prod'].view(length*a*a*bz)
        leftv = Data['left'].view(length*a*bz*2)
        rightv = Data['right'].view(length*bz*a*2)
        ternaryv = Data['ternary'].view(length*a*a*a*2)
        #
        nprod = self.noisetensor(prodv).view(length,a,a,bz)
        nleft = self.noisetensor(leftv).view(length,a,bz,2)
        nright = self.noisetensor(rightv).view(length,bz,a,2)
        nternary = self.noisetensor(ternaryv).view(length,a,a,a,2)
        #
        NoiseData = self.rr1.copydata(Data)
        NoiseData['prod'] = nprod
        NoiseData['left'] = nleft
        NoiseData['right'] = nright
        NoiseData['ternary'] = nternary
        #
        return NoiseData


    def printexamplescores(self,number):
        ExamplePool = self.Examples
        xplength = ExamplePool['length']
        xpdepth = ExamplePool['depth']
        if xplength == 0:
            print("no examples to print")
            return
        #
        print("example pool has",itp(xplength),"elements")
        #
        permutation = torch.randperm(xplength,device=Dvc)
        #
        upper = number
        if upper > xplength:
            upper = xplength
        for i in range(upper):
            ip = permutation[i]
            idepth = xpdepth[ip]
            iextent = self.extent_log[ip]
            print("sample number",itp(ip),"depth",itp(idepth),"log extent",numpr
        return

    def selectminibatch(self,minibatchsize):
        ExamplePool = self.Examples
        xplength = ExamplePool['length']
        score = self.extent_log
        if xplength < 10:
            print("not enough examples to train on")
            return False,None,None
```

```
        #
        permutation = torch.randperm(xplength,device=Dvc)
        upper = minibatchsize
        if upper > xplength:
            upper = xplength
        indices = permutation[0:upper]
        DataBatch = self.rr1.indexselectdata(ExamplePool,indices)
        scorebatch = score[indices]
        return True,DataBatch,scorebatch



    def trainingGlobal(self,M,numberofbatches,iterationsperbatch,style,minibatch
        #
        if style != 'score-A' and style != 'score-B' and style != 'score-C':
            raise CoherenceError("only allowed styles are score-A or score-B or
        #
        if self.trainingprint:
            print("/",style,numberofbatches,iterationsperbatch,partname,"/",end
        for s in range(numberofbatches):
            smb,DataBatch, scorebatch = self.selectminibatch(minibatchsize)
            if not smb:
                print("exit training")
                return
            #
            for i in range(iterationsperbatch):
                #
                M.optimizer.zero_grad()
                #
                NoiseData = self.noise(DataBatch)
                #
                predictedscore = M.network(NoiseData)
                #
                if style == 'score-A':
                    loss = M.criterionA(predictedscore,scorebatch)
                if style == 'score-B':
                    loss = M.criterionB(predictedscore,scorebatch)
                if style == 'score-C':
                    lossA = M.criterionA(predictedscore,scorebatch)
                    lossB = M.criterionB(predictedscore,scorebatch)
                    loss = (lossA + lossB)/2
                loss.backward()
                M.optimizer.step()
            #
        #
        print("-",end = ' ')
        #
        return

    def printlossaftertrainingGlobal(self,M,minibatchsize,topicture):
        #
        smb,DataBatch,scorebatch = self.selectminibatch(minibatchsize)
        if not smb:
            print("data too small")
            return
        #
```

```
        #
        mblength = DataBatch['length']
        predictedscore = M.network(DataBatch)
        lossa= M.criterionA(predictedscore,scorebatch)
        lra = numpr(lossa,3)
        lossb = M.criterionB(predictedscore,scorebatch)
        lrb = numpr(lossb,3)
        #
        with torch.no_grad():
            self.globalL1level += lossa
            self.globalL1level *= 0.5
            self.globalL1level = torch.clamp(self.globalL1level,0.005,1.0)
        #
        print("on ",itp(mblength),"values network -- L1 loss",lra,"MSE loss",lrb
        #
        if topicture:
            #
            print("global L1 loss level",numpr(self.globalL1level,4))
            #
            HST.record_loss('global',lossa,lossb)
            #
            dotsize = torch.zeros((mblength),dtype = torch.int,device=Dvc)
            dotsize[:]=2
            dotsize_np = nump(dotsize)
            #
            calcscore_npr = numpr(scorebatch,3)
            predscore_npr = numpr(predictedscore,3)
            #
            scoremax,index = torch.max(scorebatch,0)
            linelimit = numpr(scoremax,1)
            #
            #
            plt.clf()
            plt.scatter(calcscore_npr,predscore_npr,dotsize_np)
            #
            plt.plot([0.0,linelimit],[0.0,0.0],'g-',lw=1)
            plt.plot([0.0,0.0],[0.0,linelimit],'g-',lw=1)
            plt.plot([0.0,linelimit],[0.0,linelimit],'r-',lw=1)
            plt.show()
        return

    def learningGlobal(self,M,globaliterations):
        #
        self.printlossaftertrainingGlobal(M,500,True)
        #
        tweak_cursor = HST.global_tweak_cursor
        tdensity = self.pp.tweak_density * (self.pp.tweak_decay ** tweak_cursor)
        tepsilon = self.pp.tweak_epsilon * (self.pp.tweak_decay ** tweak_cursor)
        HST.global_tweak_cursor += 1
        print("tweaking global network at cursor",itp(tweak_cursor),"with densit
        M.tweak_network(M.network,tdensity,tepsilon)
        self.printlossaftertrainingGlobal(M,500,True)
        print("training",end = ' ')
        #
        explore_pre_length = self.ExplorePrePool['length']
        example_pre_length = self.ExamplesPrePool['length']
```

```
        example_pre_length = self.Examples['length']
        example_length = self.Examples['length']
        HST.record_training('global',globaliterations,explore_pre_length,example
        #
        for g in range(globaliterations):
            print("/",end=' ')
            self.trainingGlobal(M,2,20,'score-C',20,'mb20')
            self.trainingGlobal(M,1,30,'score-C',30,'mb30')
            self.trainingGlobal(M,2,10,'score-C',40,'mb40')
            self.trainingGlobal(M,5,10,'score-C',60,'mb60')
            self.trainingGlobal(M,3,8,'score-C',20,'mb20')
            self.trainingGlobal(M,3,5,'score-C',40,'mb40')
            self.trainingGlobal(M,5,3,'score-C',30,'mb30')
            #
            print(" ")
            self.printlossaftertrainingGlobal(M,300,False)
            print("  ")
        self.printlossaftertrainingGlobal(M,500,True)
        print("=================    end score training   =================")
        return

    ######## Local stuff

    def calculatescoresLocal(self,M,Data):
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length = Data['length']
        prod = Data['prod']
        #
        xypscore_exp = torch.zeros((length,a,a,bz),dtype = torch.float,device=Dv
        #
        availablexyp = self.rr1.availablexyp(length,prod)
        availablexypr = availablexyp.reshape(length*a*a*bz)
        #
        lrangevxr = arangeic(length).view(length,1,1,1).expand(length,a,a,bz).re
        xrangevxr = arangeic(a).view(1,a,1,1).expand(length,a,a,bz).reshape(leng
        yrangevxr = arangeic(a).view(1,1,a,1).expand(length,a,a,bz).reshape(leng
        prangevxr = arangeic(bz).view(1,1,1,bz).expand(length,a,a,bz).reshape(le
        #
        ivector = lrangevxr[availablexypr]
        xvector = xrangevxr[availablexypr]
        yvector = yrangevxr[availablexypr]
        pvector = prangevxr[availablexypr]
        #
        NewData = self.rr1.upsplitting(Data,ivector,xvector,yvector,pvector)
        #
        #
        ndlength = NewData['length']
        #
        #
```

```
        LocalExamples = self.rr1.nulldata()
        detection = torch.zeros((ndlength),dtype = torch.bool,device=Dvc)
        newextent_exp = torch.zeros((ndlength),dtype = torch.float,device=Dvc)
        # that should be the (approximation of) the number of nodes below and in
        newactive = torch.zeros((ndlength),dtype = torch.bool,device=Dvc)
        lower = 0
        for i in range(ndlength):
            assert lower < ndlength
            upper = lower + 1000
            if upper > ndlength:
                upper = ndlength
            detection[:] = False
            detection[lower:upper] = True
            NewDataSlice = self.rr1.detectsubdata(NewData,detection)
            AssocNewDataSlice = self.rr2.process(NewDataSlice)
            newactive_s,newdone_s,newimpossible_s = self.rr2.filterdata(AssocNew
            #
            ActiveNewDataSlice = self.rr1.detectsubdata(AssocNewDataSlice,newact
            LocalExamples = self.rr1.appenddata(ActiveNewDataSlice,LocalExamples
            #
            predictedscore_s = M.network(AssocNewDataSlice).detach()
            if torch.isnan(predictedscore_s).any(0):
                raise CoherenceException("predicted score nan")
            # recall that approximates log10 of (the number of nodes below and i
            predictedscore_s_clamp = torch.clamp(predictedscore_s,0.,8.)
            predictedscore_s_exp = 10.**predictedscore_s_clamp
            predictedscore_s_exp[newdone_s] = 0.0
            predictedscore_s_exp[newimpossible_s] = 0.1
            newextent_exp[lower:upper] = predictedscore_s_exp
            newactive[lower:upper] = newactive_s
            lower = upper
            if lower >= ndlength:
                break
        #
        xypscore_exp[ivector,xvector,yvector,pvector] = newextent_exp
        #
        xyscore_sum = xypscore_exp.sum(3)
        #
        xyscore_log = torch.log10(xyscore_sum + 1.)  # here the +1. is for the u
        #
        # now replace the global scores by these ones too: here output the min o
        availablexyr = self.rr1.availablexy(length,prod).view(length*a2)
        xyscorer = xyscore_log.view(length*a2)
        xyscorer_mod = torch.clamp(xyscorer,0.,8.)
        xyscorer_mod[~availablexyr] = 20.
        xyscore_min,xysm_indices = torch.min(xyscorer_mod.view(length,a2),1)
        assert (xyscore_min < 10.).all(0)
        #
        return xyscore_log, xyscore_min, LocalExamples


    def printsomelocalscores(self,howmany):
        elength = self.Examples['length']
        if elength == 0:
            print("no examples")
```

```
            return
        lsk = self.localscores_known
        detection = lsk
        detection_length = detection.to(torch.int64).sum(0)
        if detection_length == 0:
            print("no known score locations to print")
            return
        ivector = arangeic(elength)[detection]
        permutation = torch.randperm(detection_length,device=Dvc)
        upper = howmany
        if upper > detection_length:
            upper = detection_length
        indices_pre = permutation[0:upper]
        indices = ivector[indices_pre]
        #
        assert detection[indices].all(0)
        #
        for i in range(upper):
            indexi = indices[i]
            lsi = self.localscores[indexi]
            print(numpr(lsi,2))
        return

    def predictedscoreLocal(self,M,ivector,xvector,yvector):
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        #
        length = self.Examples['length']
        prod = self.Examples['prod']
        #
        availablexy = self.rr1.availablexy(length,prod).view(length,a,a)
        #
        assert availablexy[ivector,xvector,yvector].all(0)
        #
        Data = self.rr1.indexselectdata(self.Examples,ivector)
        dlength = Data['length']
        dlrange = arangeic(dlength)
        #
        NoiseData = self.noise(Data)
        #
        pre_score = M.network2(NoiseData)
        #
        dlrangevxa = dlrange.view(dlength,1).expand(dlength,a)
        arangevx = arangeic(a).view(1,a).expand(dlength,a)
        #
        predictedscore = pre_score[dlrange,xvector,yvector]
        return predictedscore


```

```
    def adapt_local_scores(self,ivector,xvector,yvector):
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        ExamplePool = self.Examples
        #
        length = len(ivector)
        prod = (ExamplePool['prod'])[ivector]
        availablexyv = self.rr1.availablexy(length,prod).reshape(length*a*a)
        #
        available_count = availablexyv.view(length,a*a).to(torch.int64).sum(1)
        available_count_xf = available_count.view(length,1).expand(length,a*a).t
        available_count_xf -= 1.
        available_count_xf = torch.clamp(available_count_xf,1.,100.)
        #
        score = self.localscores[ivector].reshape(length*a*a)
        score[~availablexyv] = 100.
        values,indices = torch.sort(score.view(length,a*a),1)
        #
        position = torch.zeros((length,a*a),dtype = torch.float,device=Dvc)
        #
        lrangevx = arangeic(length).view(length,1).expand(length,a*a)
        a2rangevx = arangeic(a*a).view(1,a*a).expand(length,a*a)
        position[lrangevx,indices] = a2rangevx.to(torch.float) / available_count
        #
        position_score = position + score.view(length,a*a)
        #
        adapted_score = position_score.view(length,a,a)[arangeic(length),xvector
        return adapted_score

    def selectminibatchLocal(self,minibatchsize):
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        ExamplePool = self.Examples
        xplength = ExamplePool['length']
        if xplength == 0:
            return False,None,None,None,None
        xpprod = ExamplePool['prod']
        xp_availablexy = self.rr1.availablexy(xplength,xpprod).view(xplength,a,a
        #
        irangevxr = arangeic(xplength).view(xplength,1,1).expand(xplength,a,a).r
        xrangevxr = arangeic(a).view(1,a,1).expand(xplength,a,a).reshape(xplengt
        yrangevxr = arangeic(a).view(1,1,a).expand(xplength,a,a).reshape(xplengt
        #
```

```
        avdetect = xp_availablexy[irangevxr,xrangevxr,yrangevxr]
        #
        ivector_all = irangevxr[avdetect]
        xvector_all = xrangevxr[avdetect]
        yvector_all = yrangevxr[avdetect]
        #
        avlength = avdetect.to(torch.int64).sum(0)
        #
        permutation = torch.randperm(avlength,device=Dvc)
        mblength = minibatchsize
        if mblength > avlength:
            mblength = avlength
        indices = permutation[0:mblength]
        #
        ivector = ivector_all[indices]
        xvector = xvector_all[indices]
        yvector = yvector_all[indices]
        #
        scorebatch = self.adapt_local_scores(ivector,xvector,yvector)
        #
        return True,ivector,xvector,yvector,scorebatch

    def trainingLocal(self,M,numberofbatches,iterationsperbatch,style,minibatchs
        #
        a = self.alpha
        #
        if style != 'score-A' and style != 'score-B' and style != 'score-C':
            raise CoherenceError("only allowed styles are score-A or score-B or
        #
        if self.trainingprint:
            print("/",style,numberofbatches,iterationsperbatch,partname,"/",end
        for s in range(numberofbatches):
            smb,ivector,xvector,yvector,scorebatch = self.selectminibatchLocal(m
            if not smb:
                print("exit training")
                return
            #
            for i in range(iterationsperbatch):
                #
                M.optimizer2.zero_grad()
                #
                predictedscore = self.predictedscoreLocal(M,ivector,xvector,yvec
                #
                if style == 'score-C':
                    lossA = M.criterionA(predictedscore,scorebatch)
                    lossB = M.criterionB(predictedscore,scorebatch)
                    loss = (lossA + lossB) / 2.
                loss.backward()
                M.optimizer2.step()
            #
        #
        print("-",end = ' ')
        #
        return

    def printlossaftertrainingLocal(self,M,minibatchsize,epochsnum):
```

```
    def printlossaftertrainingLocal(self,M,minibatchsize,topicture):
        #
        a=M.pp.alpha
        #
        smb,ivector,xvector,yvector,scorebatch = self.selectminibatchLocal(minib
        if not smb:
            print("data too small")
            return
        #
        mblength = len(ivector)
        #
        predictedscore = self.predictedscoreLocal(M,ivector,xvector,yvector)
        #
        lossa= M.criterionA(predictedscore,scorebatch)
        lra = numpr(lossa,3)
        #
        lossb = M.criterionB(predictedscore,scorebatch)
        lrb = numpr(lossb,3)
        #
        lossa_detach = lossa.detach()
        M.average_local_loss = (0.9 * M.average_local_loss + 0.1 * lossa_detach)
        #
        print("on ",itp(mblength),"values network -- L1 loss",lra,"MSE loss",lrb
        #
        if topicture:
            #
            HST.record_loss('local',lossa,lossb)
            #
            print("average local loss",numpr(M.average_local_loss,4))
            #
            dotsize = torch.zeros((mblength),dtype = torch.int,device=Dvc)
            dotsize[:]=2
            dotsize_np = nump(dotsize)
            #
            calcscore_npr = numpr(scorebatch,3)
            predscore_npr = numpr(predictedscore,3)
            #
            scoremax,index = torch.max(scorebatch,0)
            linelimit = numpr(scoremax,1)
            #
            plt.clf()
            plt.scatter(calcscore_npr,predscore_npr,dotsize_np)
            #
            #linelimit = 1.0
            plt.plot([0.0,linelimit],[0.0,0.0],'g-',lw=1)
            plt.plot([0.0,0.0],[0.0,linelimit],'g-',lw=1)
            plt.plot([0.0,linelimit],[0.0,linelimit],'r-',lw=1)
            #
            plt.show()
        return

    def learningLocal(self,M,globaliterations):
        #
        self.printlossaftertrainingLocal(M,500,True)
        #
        tweak_cursor = HST.local_tweak_cursor
```

```
        tweak_cursor = HST.local_tweak_cursor
        tdensity = self.pp.tweak_density * (self.pp.tweak_decay ** tweak_cursor)
        tepsilon = self.pp.tweak_epsilon * (self.pp.tweak_decay ** tweak_cursor)
        HST.local_tweak_cursor += 1
        print("tweaking local network at cursor",itp(tweak_cursor),"with density
        M.tweak_network(M.network2,tdensity,tepsilon)
        self.printlossaftertrainingLocal(M,500,True)
        print("training",end = ' ')
        #
        explore_pre_length = self.ExplorePrePool['length']
        example_pre_length = self.ExamplesPrePool['length']
        example_length = self.Examples['length']
        HST.record_training('local',globaliterations,explore_pre_length,example_
        #
        for g in range(globaliterations):
            print("/",end=' ')
            self.trainingLocal(M,3,20,'score-C',20,'mb20')
            self.trainingLocal(M,1,15,'score-C',30,'mb30')
            self.trainingLocal(M,3,4,'score-C',60,'mb60')
            self.trainingLocal(M,3,10,'score-C',40,'mb40')
            self.trainingLocal(M,3,3,'score-C',20,'mb20')
            self.trainingLocal(M,3,2,'score-C',40,'mb40')
            self.trainingLocal(M,3,1,'score-C',30,'mb30')
            #
            print(" ")
            self.printlossaftertrainingLocal(M,300,False)
            print("  ")
        self.printlossaftertrainingLocal(M,500,True)
        print("=================    end score training   =================")
        return

```

```
class Driver :    # to run everything, it includes the sieve for instances sigma
    def __init__(self,P):
        #
        #
        self.Pp = P
        self.rr4 = Relations4(self.Pp)
        self.rr3 = self.rr4.rr2
        self.rr2 = self.rr4.rr2
        self.rr1 = self.rr4.rr1
        self.alpha = self.Pp.alpha
        self.alpha2 = self.Pp.alpha2
        self.alpha3 = self.Pp.alpha3
        self.alpha3z = self.Pp.alpha3z
        self.beta = self.Pp.beta
        self.betaz = self.Pp.betaz
        #
        self.sga = SymmetricGroup(self.alpha)
        #
        self.zbinatable = self.makezbinatable()
        #
        self.init_length,self.init_left_table = self.make_init_left_table()
        #
        self.donecount_collection = 0
```

```
        self.ECN_collection = 0.
        self.ECN_average = 0.
        #
        self.Cc = Classifier(self.Pp)
        #
        self.Ll = Learner(self.rr4)
        #
        HST.record_driver(self.alpha,self.beta)



    def printprod(self,prod,loc):
        #
        a=self.alpha
        a2=a*a
        a3=a*a*a
        a3z = a3+1
        b=self.beta
        bz = b+1
        #
        prodi = prod[loc]
        prarray = torch.zeros((a,a),dtype = torch.int64,device=Dvc)
        for x in range(a):
            for y in range(a):
                column = prod[loc,x,y]
                prarray[x,y] = self.printcolumn(column)
        print(nump(prarray))
        return

    def printcolumn(self,column):
        #
        a=self.alpha
        a2=a*a
        a3=a*a*a
        a3z = a3+1
        b=self.beta
        bz = b+1
        #
        assert len(column) == bz
        #
        column_sum = column.to(torch.int64).sum(0)
        #
        if column_sum == 0:
            return -8
        #
        if column_sum == 1:
            value,prvalue = torch.max(column.to(torch.int64),0)
            return prvalue
        if column_sum == bz:
            return -1
        exponents = arangeic(bz)
        powers = 10**exponents
        prvalue = (column.to(torch.int64) * powers).sum(0)
        prvalue += 3 * (10 ** bz)
        return prvalue
```

```
    def printcolumn2(self,column):
        #
        assert len(column) == 2
        #
        column_sum = column.to(torch.int64).sum(0)
        #
        if column_sum == 0:
            return -8
        if column_sum == 2:
            return 3
        #
        return column[1]

    def printleft(self,left,loc):
        #
        a=self.alpha
        a2=a*a
        a3=a*a*a
        a3z = a3+1
        b=self.beta
        bz = b+1
        #
        prarray = torch.zeros((a,bz),dtype = torch.int64,device=Dvc)
        for x in range(a):
            for y in range(bz):
                column = left[loc,x,y]
                prarray[x,y] = self.printcolumn2(column)
        print(nump(prarray))
        return

    def printright(self,right,loc):
        #
        a=self.alpha
        a2=a*a
        a3=a*a*a
        a3z = a3+1
        b=self.beta
        bz = b+1
        #
        prarray = torch.zeros((bz,a),dtype = torch.int64,device=Dvc)
        for x in range(bz):
            for y in range(a):
                column = right[loc,x,y]
                prarray[x,y] = self.printcolumn2(column)
        print(nump(prarray))
        return

    def print_prod_left_right(self,Data,i,as_sigma):
        prod = Data['prod']
        left = Data['left']
        right = Data['right']
        #
        print("-------------------------------------------------")
        print("at instance",itp(as_sigma),"the prod, left and right are respecti
```

```
        self.printprod(prod,i)
        self.printleft(left,i)
        self.printright(right,i)
        print("-----------------------------------------------------")
        return

    def print_just_left(self,Data,i,as_sigma):
        left = Data['left']
        #
        print("-----------------------------------------------------")
        print("at instance",itp(as_sigma),"the left multiplication matrix is")
        self.printleft(left,i)
        print("-----------------------------------------------------")
        return


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

    def makezbinatable(self):
        #
        a = self.alpha
        #
        power = 2**a
        #
        zbatable = torch.zeros((power,a),dtype = torch.bool,device=Dvc)
        for z in range(power):
            zbatable[z,:] = zbinary(a,z)
        return zbatable

    def collection(self,m):  # this doesn't work perfectly, it needs to be re-si
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        power = 2**a
        gl = self.sga.gtlength
        #
        gtbin = self.sga.gtbinary
        #
        if m <= 0:
            print("m <= 0 not allowed")
            raise CoherenceError("exiting")
        if m == 1:
            zrangevx = arangeic(power).view(1,power).expand(gl,power)
            detection = (gtbin >= zrangevx).all(0)
            clength = detection.to(torch.int64).sum(0)
            collec = arangeic(power)[detection].view(clength,m)
            #
            zbatablev = self.zbinatable.view(power,1,a)
            collec_bin = zbatablev[detection]
            #
            previous_possibilities = torch.ones((clength,power),dtype = torch.bo
```

```
            previous_possibilities = torch.ones((clength,power),dtype = torch.bo
            #
            previous_subgroup = torch.ones((clength,gl),dtype = torch.bool,devic
        #
        else:
            cp,collec_prev,collec_bin_prev,poss_prev,subgroup_prev = self.collec
            #
            detection_prev = poss_prev.view(cp*power)
            #
            subgroup_prev_vxr = subgroup_prev.view(cp,1,gl).expand(cp,power,gl).
            z_current = arangeic(power).view(1,power,1).expand(cp,power,gl).resh
            g_current = arangeic(gl).view(1,gl).expand(cp*power,gl)
            detection_current = ((~subgroup_prev_vxr) | ( gtbin[g_current,z_curr
            #
            detection = detection_prev & detection_current
            #
            clength = detection.to(torch.int64).sum(0)
            collec_prev_next = collec_prev.view(cp,1,m-1).expand(cp,power,m-1).r
            zrangevxr = arangeic(power).view(1,power,1).expand(cp,power,1).resha
            new_prev = collec_prev_next[detection]
            zvector = zrangevxr[detection]
            collec = torch.cat((new_prev,zvector),1)
            #
            collec_bin_prev_next = collec_bin_prev.view(cp,1,m-1,a).expand(cp,po
            new_bin_prev = collec_bin_prev_next[detection]
            zbatablevxr = self.zbinatable.view(1,power,1,a).expand(cp,power,1,a)
            zbavector = zbatablevxr[detection]
            collec_bin = torch.cat((new_bin_prev,zbavector),1)
            #
            poss_prev_vxr = poss_prev.view(cp,1,power).expand(cp,power,power).re
            previous_possibilities = poss_prev_vxr[detection]
            #
            subgroup_prev_vxr = subgroup_prev.view(cp,1,gl).expand(cp,power,gl).
            previous_subgroup = subgroup_prev_vxr[detection]
        #
        grange_vx = arangeic(gl).view(1,gl,1).expand(clength,gl,m)
        collec_vx = collec.view(clength,1,m).expand(clength,gl,m)
        transform = gtbin[grange_vx,collec_vx]
        transform_sort,t_indices = torch.sort(transform,2)
        #
        subgroup = (collec_vx == transform_sort).all(2)
        #
        #
        next_transform = gtbin.view(1,gl,power).expand(clength,gl,power)
        prev_bound = (collec[:,m-1]).view(clength,1,1).expand(clength,gl,power)
        prev_subgroupvx = previous_subgroup.view(clength,gl,1).expand(clength,gl
        #
        next_possib_prev = ( (~prev_subgroupvx) | (prev_bound <= next_transform)
        possibilities = next_possib_prev & previous_possibilities
        #
        return clength,collec,collec_bin,possibilities,subgroup

    def collectiontest(self,amount):
        #
        a = self.alpha
        a2 = self.alpha2
```

```
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        #
        clength,collec,collec_bin,possibilities,subgroup = self.collection(b)
        print("collection has length",itp(clength))
        upper = 20
        if upper > clength:
            upper = clength
        print("first",itp(upper),"elements are as follows")
        #
        collec_sum = (collec_bin.to(torch.int64).sum(2)).sum(1)
        collec_sum1 = collec_bin.to(torch.int64).sum(1)
        collec_sum2 = collec_bin.to(torch.int64).sum(2)
        for q in range(a*b+1):
            freq = (collec_sum == q).to(torch.int64).sum(0)
            print("for amount",itp(q),"frequency",itp(freq))
        amount_detect = (collec_sum == amount)
        #
        ad_freq = amount_detect.to(torch.int64).sum(0)
        collec_detect = collec_bin[amount_detect]
        print("ad freq",itp(ad_freq))
        for i in range(ad_freq):
            print("--------------------------")
            print(numpi(collec_detect[i]))
            print("collec_sum1",nump((collec_sum1[amount_detect])[i]))
            print("collec_sum2",nump((collec_sum2[amount_detect])[i]))
        return

    def lex_lt(self,width,z1,z2):
        #
        assert width > 0
        #
        if width == 1:
            z1new = z1[:,0]
            z2new = z2[:,0]
            lt = (z1new < z2new)
            return lt
        z1prev = z1[:,0:width-1]
        z2prev = z2[:,0:width-1]
        #
        lt_prev = self.lex_lt(width-1,z1prev,z2prev)
        #
        eq_prev = (z1prev == z2prev).all(1)
        #
        z1new = z1[:,width-1]
        z2new = z2[:,width-1]
        lt = ( lt_prev | (eq_prev & (z1new < z2new)) )
        return lt


    def collection_sieve(self):
        #
```

```
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        power = 2**a
        gl = self.sga.gtlength
        #
        gtbin = self.sga.gtbinary
        #
        clength,collec,collec_bin,possibilities,subgroup = self.collection(b)
        #
        collecvxr = collec.view(clength,1,b).expand(clength,gl,b).reshape(clengt
        grangevxr = arangeic(gl).view(1,gl,1).expand(clength,gl,b).reshape(cleng
        #
        transform = gtbin[grangevxr,collecvxr]
        transform_sort, t_indices = torch.sort(transform,1)
        #
        transform_ltv = self.lex_lt(b,transform_sort,collecvxr)
        transform_lt = transform_ltv.view(clength,gl)
        #
        throw = transform_lt.any(1)
        #
        detection = (~throw)
        sieved_length = detection.to(torch.int64).sum(0)
        sieved_collection = collec[detection]
        sieved_collection_bin = collec_bin[detection]
        #
        return sieved_length, sieved_collection, sieved_collection_bin

    def sieve_test(self):
        sieved_length, sieved_collection, sieved_collection_bin = self.collectio
        print("sieved collection has length",itp(sieved_length))
        return


    def make_init_left_table(self):
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length, sieved_collection, sieved_collection_bin = self.collection_sieve
        #
        init_left_table = torch.zeros((length,a,bz,2),dtype = torch.bool,device=
        #
        left_value = sieved_collection_bin.permute(0,2,1)
        init_left_table[:,:,0:b,1] = left_value
        init_left_table[:,:,0:b,0] = ~left_value
        init_left_table[:,:,b,0] = True
```

```
        #
        print(" ")
        print("available left table instances are 0 <= sigma <",itp(length))
        print("   ---> these should be noted as the possible values of sigma for
        #
        zerocolumn = (~left_value).all(1)
        zerocolumn_number = zerocolumn.to(torch.int64).sum(1)
        overhalf = (2*zerocolumn_number > b)
        overhalf_indices = arangeic(length)[overhalf]
        print("   ")
        print("locations with > half zero columns are",nump(overhalf_indices))
        print("   ---> it is suggested not to use the sigma instances in this li
        print("   ")
        #
        return length,init_left_table

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



    def initialdata(self,instancevector,dropoutlimit):
        #
        a = self.alpha
        a2 = self.alpha2
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        length = len(instancevector)
        prod = torch.ones((length,a,a,bz),dtype = torch.bool,device=Dvc)
        #
        left = self.init_left_table[instancevector]
        right = torch.ones((length,bz,a,2),dtype = torch.bool,device=Dvc)
        right[:,b,:,1] = False
        #
        if self.Pp.verbose:
            print("initial prod at 0")
            self.printprod(prod,0)
            print("initial left at 0")
            print(numpi(left[0,:,:,1]))
            print("initial right at 0")
            print(numpi(right[0,:,:,1]))
        #
        depth = torch.zeros((length),dtype = torch.int,device=Dvc)
        #
        ternary = torch.ones((length,a,a,a,2),dtype = torch.bool,device=Dvc)
        #
        info = torch.zeros((length,self.Pp.infosize),dtype = torch.int64,device=
        info[:,self.Pp.sampleinfolower:self.Pp.sampleinfoupper] = -1
        #
        #
        RawData = {
            'length': length,
            'depth': depth,
```

```
            'prod': prod,
            'left': left,
            'right': right,
            'ternary': ternary,
            'info': info,
        }
        #
        if dropoutlimit > 0:
            rectangle = arangeic(length).view(length,1).expand(length,dropoutlim
            index_slice = (torch.randperm(length*dropoutlimit,device=Dvc))[0:dro
            indices = rectangle[index_slice]
            AugmentedData = self.rr1.indexselectdata(RawData,indices)
            # set phase
            tirage = torch.rand((dropoutlimit),device = Dvc)
            phase1_upper = 0.8 * self.Pp.splitting_probability
            phase12_upper = 0.8
            phase1 = (tirage < phase1_upper)
            phase2 = (tirage < phase12_upper) & (~phase1)
            AugmentedData['info'][:,self.Pp.phase][phase2] = 2
            AugmentedData['info'][:,self.Pp.phase][phase1] = 1
            return AugmentedData
        return RawData

    def print_instances(self):
        #
        instancevector, training_instances, proof_title = self.instance_chooser(
        InitialData = self.initialdata(instancevector,0)
        length = InitialData['length']
        print(proof_title)
        print("initial data from this collection of instances has length",itp(le
        for i in range(length):
            as_sigma = instancevector[i]
            #self.print_prod_left_right(InitialData,i,as_sigma)
            self.print_just_left(InitialData,i,as_sigma)
        print("- - - - - - - - - - - - - - - - - - - - -")
        return


    def classificationproof(self,Mstrat,Mlearn,dropoutlimit,proving_instances,ti
        print("---   ---   ---   ---   ---   ---   ---   ---   ---")
        print("               classification proof")
        print("---   ---   ---   ---   ---   ---   ---   ---   ---")
        #
        HST.title_text_sigma_proof = title_text
        #
        self.rr4.proofnumber = 0
        self.rr4.proofinstance = 0
        self.rr4.allnumbers = 1
        #
        if dropoutlimit == 0:
            HST.reset_current_proof()
        #
        self.Cc.initialize()
        #
        InitialData = self.initialdata(proving_instances,dropoutlimit)
```

```
        #
        if dropoutlimit > 0:
            AssocInitialData = self.rr2.process(InitialData)
            activedetect,donedetect,impossibledetect = self.rr2.filterdata(Assoc
            ActiveInitialData = self.rr1.detectsubdata(AssocInitialData,activede
            #
            explore_upper = self.Pp.root_injection
            if explore_upper > dropoutlimit:
                explore_upper = dropoutlimit
            self.Ll.prepoolExplore(Mlearn,ActiveInitialData,explore_upper)
        #
        pl,ActivePool,DonePool,prooflength = self.rr4.proofloop(Mstrat,Mlearn,se
        #
        print("---   ---   ---   ---   ---   ---   ---   ---   ---")
        print("this proof was treating alpha =",itp(self.alpha),"beta =",itp(sel
        print("proof ended after step number",itp(prooflength))
        if ActivePool['length'] > 0:
            print("proof outputs Active Data of length",itp(ActivePool['length']
        print("proof has done count",itp(self.rr4.donecount),end=' ')
        print("and estimated cumulative nodes",numpr(self.rr4.ECN,1))
        self.donecount_collection += self.rr4.donecount
        self.ECN_collection += self.rr4.ECN
        print("classifier eq pool has length",itp(self.Cc.eqlength))
        #
        if dropoutlimit == 0:
            if Mstrat.benchmark:
                HST.record_current_proof(benchmark = True)
            else:
                HST.record_current_proof()
        #
        print("---   ---   ---   ---   ---   ---   ---   ---   ---")
        print("          classification proof done")
        print("---   ---   ---   ---   ---   ---   ---   ---   ---")
        print("===   ===   ===   ===   ===   ===   ===   ===   ===   ===   ===
        #
        return

    #### mini programs for creation of the instancevector_title object (it is re

    def InAll(self):
        instance_vector = arangeic(self.init_length)
        #
        title_text = F'for all sigma instances'
        #
        return instance_vector, instance_vector, title_text

    def InOne(self,instance):
        assert 0 <= instance < self.init_length
        instance_vector = torch.zeros((1),dtype = torch.int64,device=Dvc)
        instance_vector[0] = instance
        #
        title_text = F'for sigma instance {instance}'
        #
        return instance_vector, instance_vector, title_text

    def InSeg(self,lower,upper):
```

```
    def InSeg(self,lower,upper):
        assert 0 <= lower < self.init_length
        assert lower < upper
        if upper > self.init_length:
            print("retracting the upper value to",self.init_length)
        upper_mod =upper
        if upper_mod > self.init_length:
            upper_mod = self.init_length
        instance_vector = arangeic(self.init_length)[lower:upper]
        #
        title_text = F'for sigma instances in range {lower}:{upper_mod}'
        #
        return instance_vector, instance_vector, title_text

    def InSkip(self,skip,lower,upper):
        assert 0 <= lower < self.init_length
        assert lower < upper
        upper_mod =upper
        if upper_mod > self.init_length:
            upper_mod = self.init_length
        detection = torch.zeros((self.init_length),dtype = torch.bool,device=Dvc
        detection[lower:upper] = True
        detection[skip] = False
        training_vector = arangeic(self.init_length)[detection]
        #
        proving_vector = torch.zeros((1),dtype = torch.int64,device=Dvc)
        proving_vector[0] = skip
        #
        title_text = F'training for sigma in range {lower}:{upper} skipping and
        #
        return proving_vector, training_vector, title_text

    def InList(self,proving_list,training_list):
        proving_vector = torch.tensor(proving_list,dtype = torch.int64,device=Dv
        training_vector = torch.tensor(training_list,dtype = torch.int64,device=
        #
        title_text = F'training for sigma instances in list {training_list} and
        #
        return proving_vector, training_vector, title_text



    def basicloop(self,Mstrat,Mlearn,training_instances,title_text):
        #
        dropout2 = 300
        #
        HST.title_text_sigma_train = title_text
        #
        for i in range(self.Pp.basicloop_iterations):
            print("------      ------      basic loop",itp(i),"------      -----
            #
            #
            self.Pp.dropout_style = 'regular'
            self.classificationproof(Mstrat,Mlearn,dropout2,training_instances,t
            self.Ll.prepoolSamples(Mlearn,self.rr4.SamplePool,self.Ll.new_exampl
            #
```

```
            #
            self.Ll.scoreExamples(Mlearn)
            #
            self.Pp.dropout_style = 'adaptive'
            #
            self.classificationproof(Mstrat,Mlearn,dropout2,training_instances,t
            self.Ll.prepoolSamples(Mlearn,self.rr4.SamplePool,self.Ll.new_exampl
            self.classificationproof(Mstrat,Mlearn,100,training_instances,title_
            self.Ll.addscoredexamples(Mlearn)
            self.Ll.prepoolSamples(Mlearn,self.rr4.SamplePool,self.Ll.new_exampl
            #
            self.Ll.scoreExamples(Mlearn)
            self.Ll.scoreExplore(Mlearn)
            self.Ll.scoreOutlier(Mlearn)
            print("------  global learning ---   ----",itp(i),"------      -----
            self.Ll.learningGlobal(Mlearn,self.Pp.basicloop_training_iterations)
            if Mlearn.network2_trainable:
                print("------  local learning  ---   ----",itp(i),"------      -
                self.Ll.learningLocal(Mlearn,self.Pp.basicloop_training_iteratio
            else:
                print("network 2 is not trainable")
            print("======      ======      ======      ======      ======      =
            gcc = gc.collect()
            memReport('mg')
            print("======      ======      ======      ======      ======      =
        print("end of",itp(self.Pp.basicloop_iterations),"iterations of the basi
        print("======      ======      ======      ======      ======      =====
        return

    def basicloop_classificationproof(self,Mstrat,Mlearn,proving_instances,train
        #
        print("suggested number of proof cycles: between 20 and 50")
        runs = int(input("input the number of proof cycles to do : "))
        for s in range(runs):
            Dd.basicloop(Mstrat,Mlearn,training_instances,title_text)
            print(">>>",s+1,"   (out of ",runs," )")
            Dd.classificationproof(Mstrat,Mlearn,0,proving_instances,title_text)
            HST.graph_history(self.Pp,'big')
        return

    #### the following function automates the process of choosing a collection o

    def instance_chooser(self):
        print("choose instances, this chooser allows : all, one, seg, skip  (do
        instance_type = input("input type : ")
        if instance_type != "all" and instance_type != "one" and instance_type !
            print("please use one of : all one seg skip")
            raise CoherenceError("exiting")
        if instance_type == "all":
            print(F"this will do all sigma instances in the existing range 0 <=
            proving_instances, training_instances, title_text = self.InAll()
        if instance_type == "one":
            print(F"to do a single sigma instance, choose in range 0 <= sigma <
            sigma_instance = int(input("input sigma instance : "))
            proving_instances, training_instances, title_text = self.InOne(sigma
        if instance_type == "seg":
```

```
            print("to do sigma instances in a segment lower <= sigma < upper")
            segment_lower = int(input("input lower : "))
            segment_upper = int(input("input upper : "))
            proving_instances, training_instances, title_text = self.InSeg(segme
        if instance_type == "skip":
            print("to train on a segment skipping a single value, and do proofs
            segment_lower = int(input("input training segment lower : "))
            segment_upper = int(input("input training segment upper : "))
            skip = int(input("input instance to prove, and to skip during traini
            proving_instances, training_instances, title_text = self.InSkip(skip
        return proving_instances, training_instances, title_text
```

```
class Parameters :    # records various parameters
    def __init__(self):
        #
        print("please enter alpha, beta and model_n")
        print("alpha = | A - A^2 | and beta = | A^2 - A^3 |, we are doing |A^3 -
        print("ranges 2 <= alpha, beta <= 6 and alpha + beta <= 10, GPU needed f
        print("model_n governs the size of the neural networks")
        print("suggested value n=4, can go to n=8 for more difficult cases on GP
        alpha = int(input("input alpha : "))
        beta = int(input("input beta : "))
        model_n = int(input("input model_n : "))
        #
        #
        if alpha < 2 or beta < 2 or alpha > 6 or beta > 6 or alpha + beta > 10:
            print("suggested range is alpha,beta in [2,...,6] and alpha + beta <
            raise CoherenceError("exiting")
        if model_n < 1 or model_n > 12:
            print("suggested range for model_n is [2,...,12]")
            raise CoherenceError("exiting")
        #
        self.model_n = model_n  # could go up to 10 (that was what we did before
        #
        self.alpha = alpha
        self.alpha2 = alpha*alpha
        self.alpha3 = alpha*alpha*alpha
        self.alpha3z = self.alpha3+1
        self.beta = beta
        self.betaz = self.beta +1
        #
        HST.record_parameters(self.alpha,self.beta)
        #
        self.verbose = False
        #
        #
        self.rdetect_max = 1000
        #
        self.global_params = 0
        self.local_params = 0
        #
        # for rr1:
```

```
        #
        self.qvalue = 0.9
        #
        self.ascore_max = 7 # it is going to be a sum of a 0,1,2 and a 0,2,4
        #
        self.pastsize = 100
        self.futuresize = 1000
        #
        # for rr2:
        #
        self.profile_filter_on = True
        self.halfones_filter_on = True
        #
        # for rr3:
        self.chunksize = 1000  # reduced this: for the 5,5 case 1000 led to memo
        self.chunksize_extra = 10
        self.exponent = 0.9
        self.chunkconstant = 300
        self.chunkextent = 10
        #
        self.ukseuil = 0.5
        #
        self.newratio = 0.8  # how much we use the model rather than benchmark v
        self.search_epsilon = 0.3  # the proportion of randomized strategy choic
        #
        # for rr4:
        self.prooflooplength = 4000
        self.done_max = 30000
        #
        #self.sleeptime = 120  # use this on a laptop
        self.sleeptime = 0 # was 5
        self.periodicity = 5
        self.stopthreshold = 100000  # too big for a notebook utilisation
        if torch.cuda.is_available(): self.trainingiterations = 4  # was 8
        else: self.trainingiterations = 1
        #
        self.root_injection = 100
        self.randomize_q = 0.7
        self.randomize_factor = 0.1
        self.perturbation_factor = 0.3
        #
        self.spiral_mix_threshold = 10
        self.dropout_style = 'regular'
        #
        #
        # info ranges   ###################
        #
        self.infosize =  300   # currently we just need up to 250 for sample inf
        #
        self.samplepoints = 15
        self.sampleinfolower = 50
        self.sampleinfoupper = 250 # on ecrase vilower
        #
        self.fulldata_location = 25
        self.phase = 12
        #
```

```
        #
        ###################################
        #
        self.basicloop_iterations = 3
        self.basicloop_training_iterations = 3
        #
        self.tweak_start = 200
        self.tweak_step = 4
        self.tweak_density = 0.05
        self.tweak_epsilon = 0.02
        self.tweak_decay = 0.9
        #
        self.splitting_probability = 1. - ( 10**(   -2. / ( (itt(self.alpha).to(
        print("splitting probability",numpr(self.splitting_probability,4) )
        #
        self.outlier_threshold = 2.5
        #
        ##### learner parameters
        #
        self.explore_max = 30000
        self.examples_max = 5000
        self.new_examples_max = 300
        self.new_explore_max = 400
        self.outlier_max = 2000
        self.new_outliers_max = 300
        #
        self.noise_level = 0.05
        self.noise_period = 50.
        self.noise_decay = 0.5  # decay per period
        #
```

```
###### end of basic program cells ###############
#def stopcompile():
```

```
##### to set up the environment: please run this cell and enter the desired alph
#####    ( first suggested values are alpha = 3, beta = 2 and model_n = 4 )

HST = Historical(10000)
HST.proof_nodes_max = 100000  #  can be used to set the upper limit for proof le
#
Pp = Parameters()  # this will ask for alpha, beta and model_n
#
Pp.basicloop_iterations = 3  # default is 3
Pp.basicloop_training_iterations = 3  # default is 3
#
Dd = Driver(Pp)
#
```

```
# the following could be set to False to turn off those additional filters
Pp.profile_filter_on = True  # the default is True
Pp.halfones_filter_on = True  # the default is True
#
#####
```

```
#####  please run this cell to (re)initialize the model
#
Mm = SgModel(Pp)
Pp.spiral_mix_threshold = 4
Mmr = ProtoModel(Pp,'spiral_mix')
#
#####
```

```

```

```
def stopcompile():
```

```
################################################################################
########
######## do a classificationproof with Mmr to set benchmark, then one with Mm fo
######## then do basicloop_classificationproof  for example for 50 iterations
######## the basicloop_classificationproof can be repeated. Cumulative results a
########
######## (a first suggested value for (alpha,beta)=(3,2) could be sigma = 3)
########
################################################################################
```

```
#####
#
proving_instances, training_instances, title_text = Dd.instance_chooser()
#
HST.reset()
Dd.classificationproof(Mmr,Mm,0,proving_instances,title_text)  # sets the benchm
Dd.classificationproof(Mm,Mm,0,proving_instances,title_text)   # to record proof
```

```
Dd.basicloop_classificationproof(Mm,Mm,proving_instances,training_instances,titl
```

```
################################################################################
```

```
##### this is the end of the main notebook part, further optional cells are incl
##### also note that the previous basicloop_classificationproof cell can be repe
def stopcompile():
```

```
################################################################################
```

```
##### a function to view the initial data with instance input as above
Dd.print_instances()
```

```
###### InList use case (this isn't covered by the instance chooser)
######
## modify the following as desired: proving list, training list
proving_instances, training_instances, title_text = Dd.InList([6,7],[5,8,9,10,11
#
HST.reset()
Dd.classificationproof(Mmr,Mm,0,proving_instances,title_text)
Dd.classificationproof(Mm,Mm,0,proving_instances,title_text)
```

```
Dd.basicloop_classificationproof(Mm,Mm,proving_instances,training_instances,titl
```

```
################################################################################
```

```
def stopcompile():
```

```
################################################################################
```

```
################################################################################
```

```
#### do the proofs in a range

for i in range(13):
    print("instance",i)
    proving_instances, training_instances, title_text = Dd.InOne(i)
    Dd.classificationproof(Mm,Mm,0,proving_instances,title_text)
```

```
################################################################################
```

```
def stopcompile():
```

```
################################################################################
```

```
#### next: the class used to prove the theoretical minimum (this is for alpha,be
#### note that the parameters and model should be initialized for (3,2)
```

```
class Minimizer :    # this becomes the first element of the relations datatype
    def __init__(self,model,sigma,cutx,cuty,cutp):
        #
        #
        self.Mm = model
        self.Pp = self.Mm.pp
        self.Dd = Driver(self.Pp)
        assert self.Pp.alpha == 3
        assert self.Pp.beta == 2
        #
        self.sigma = sigma
        #
```

```
        #
        print("Minimizer for sigma =",sigma)
        #
        self.rr4 = Relations4(self.Pp)
        self.rr3 = self.rr4.rr3
        self.rr2 = self.rr4.rr2
        self.rr1 = self.rr4.rr1
        self.alpha = self.Pp.alpha
        self.alpha2 = self.Pp.alpha2
        self.alpha3 = self.Pp.alpha3
        self.alpha3z = self.Pp.alpha3z
        self.beta = self.Pp.beta
        self.betaz = self.Pp.betaz
        #
        self.length_max = 500000
        #
        instancevector, trainingvector, proof_title = self.Dd.InOne(self.sigma)
        self.InitialData = self.Dd.initialdata(instancevector,0)
        #
        self.CurrentData = self.rr1.nulldata()
        self.FullData = self.rr1.nulldata()
        self.FullDoneData = self.rr1.nulldata()
        #
        self.up = torch.zeros((self.length_max),dtype = torch.int64,device=Dvc)
        self.xvalue = torch.zeros((self.length_max),dtype = torch.int64,device=D
        self.yvalue = torch.zeros((self.length_max),dtype = torch.int64,device=D
        self.pvalue = torch.zeros((self.length_max),dtype = torch.int64,device=D
        #
        self.lowerbound = torch.zeros((self.length_max),dtype = torch.int64,devi
        self.upperbound = torch.zeros((self.length_max),dtype = torch.int64,devi
        #
        self.down = torch.zeros((self.length_max,self.alpha,self.alpha,self.beta
        self.down[:,:,:,:] = -2
        #
        self.split = torch.zeros((self.length_max),dtype = torch.bool,device=Dvc
        self.inplay = torch.zeros((self.length_max),dtype = torch.bool,device=Dv
        #
        self.availablexyp = torch.zeros((self.length_max,self.alpha,self.alpha,s
        self.availablexy = torch.zeros((self.length_max,self.alpha,self.alpha),d
        #
        self.cutx = cutx
        self.cuty = cuty
        self.cutp = cutp
        print("cut to check is x y p =",cutx,cuty,cutp)
        #
        self.combo_all()


    def next_stage_data(self,DataToSplit):
        #
        a = self.alpha
        a2 = self.alpha2
        a2z = self.alpha2 +1
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
```

```
        b = self.beta
        bz = self.betaz
        #
        #
        length = DataToSplit['length']
        prod = DataToSplit['prod']
        #
        availablexyp = self.rr1.availablexyp(length,prod).view(length,a2,bz)
        #
        avxyp_amount = availablexyp.to(torch.float).sum(2).sum(1).sum(0)
        avxyp_denom = itt(length).to(torch.float)
        if avxyp_denom < 0.1:
            avxyp_denom = 1.
        availablexyp_average = avxyp_amount / avxyp_denom
        ##print("average available xyp is",numpr(availablexyp_average,2))
        #
        lrangevxr = arangeic(length).view(length,1,1).expand(length,a2,bz).resha
        xyvectorvxr = arangeic(a2).view(1,a2,1).expand(length,a2,bz).reshape(len
        bzrangevxr = arangeic(bz).view(1,1,bz).expand(length,a2,bz).reshape(leng
        #
        verticaldetect = availablexyp[lrangevxr,xyvectorvxr,bzrangevxr]
        #
        #
        ivector_vert = lrangevxr[verticaldetect]
        xyvector_vert = xyvectorvxr[verticaldetect]
        pvector_vert = bzrangevxr[verticaldetect]
        #
        prx = arangeic(a).view(a,1).expand(a,a).reshape(a2)
        pry = arangeic(a).view(1,a).expand(a,a).reshape(a2)
        #
        xvector_vert = prx[xyvector_vert]
        yvector_vert = pry[xyvector_vert]
        #
        #
        NewData = self.rr1.upsplitting(DataToSplit,ivector_vert,xvector_vert,yve
        #
        #
        ndlength = NewData['length']
        #
        #
        AssocNewData = self.rr1.nulldata()
        detection = torch.zeros((ndlength),dtype = torch.bool,device=Dvc)
        newactive = torch.zeros((ndlength),dtype = torch.bool,device=Dvc)
        newdone = torch.zeros((ndlength),dtype = torch.bool,device=Dvc)
        newimpossible = torch.zeros((ndlength),dtype = torch.bool,device=Dvc)
        lower = 0
        for i in range(ndlength):
            assert lower < ndlength
            upper = lower + 1000
            if upper > ndlength:
                upper = ndlength
            detection[:] = False
            detection[lower:upper] = True
            NewDataSlice = self.rr1.detectsubdata(NewData,detection)
            AssocNewDataSlice = self.rr2.process(NewDataSlice)
            AssocNewData = self.rr1.appenddata(AssocNewData,AssocNewDataSlice)
```

```
            newactive_s,newdone_s,newimpossible_s = self.rr2.filterdata(AssocNew
            newactive[lower:upper] = newactive_s
            newdone[lower:upper] = newdone_s
            newimpossible[lower:upper] = newimpossible_s
            lower = upper
            if lower >= ndlength:
                break
        #
        NewActiveData = self.rr1.detectsubdata(AssocNewData,newactive)
        #
        NewDoneData = self.rr1.detectsubdata(AssocNewData,newdone)
        #
        assert len(newactive) == len(ivector_vert)
        assert newactive.to(torch.int64).sum(0) == NewActiveData['length']
        #
        return ivector_vert,xvector_vert,yvector_vert,pvector_vert,newactive,new
```

```
    def manage_initial_stage(self):
        #
        self.FullData = self.rr1.copydata(self.InitialData)
        #
        fdlength = self.FullData['length']
        assert fdlength == 1
        #
        #
        self.up[0] = -1
        self.xvalue[0] = -1
        self.yvalue[0] = -1
        self.pvalue[0] = -1
        #
        self.lowerbound[0] = 1
        self.upperbound[0] = 1000
        #
        #
        self.split[0] = False
        self.inplay[0] = True
        #
        newactive_length = self.InitialData['length']
        newactive_prod = self.InitialData['prod']
        #
        new_availablexyp = self.rr1.availablexyp(newactive_length,newactive_prod
        new_availablexy = new_availablexyp.any(3)
        #
        self.availablexyp[0] = new_availablexyp
        self.availablexy[0] = new_availablexy
        #
        new_fdlength = self.FullData['length']
        #
        self.FullData['info'][:,self.Pp.fulldata_location] = arangeic(new_fdleng
        #
```

```
        ##print("initialized full data that now has length",itp(new_fdlength))
        return


    def manage_next_stage(self):
        #
        fdlength = self.FullData['length']
        if fdlength == 0:
            self.manage_initial_stage()
            return
        #
        fd_split = self.split[0:fdlength]
        fd_inplay = self.inplay[0:fdlength]
        #
        fd_current = fd_inplay & ~fd_split
        DataToSplit = self.rr1.detectsubdata(self.FullData,fd_current)
        #
        fd_index = arangeic(fdlength)[fd_current]
        #
        ivector_vert,xvector_vert,yvector_vert,pvector_vert,newactive,newdone,ne
        #
        inew = fd_index[ivector_vert[newactive]]
        xnew = xvector_vert[newactive]
        ynew = yvector_vert[newactive]
        pnew = pvector_vert[newactive]
        #
        newactive_length = NewActiveData['length']
        newactive_prod = NewActiveData['prod']
        #
        lower = itt(fdlength).clone()
        upper = fdlength + newactive_length
        if upper > self.length_max:
            print("upper is",itp(upper),"whereas max length is",itp(self.length_
            raise CoherenceError("length overflow")
        self.FullData = self.rr1.appenddata(self.FullData,NewActiveData)
        self.FullDoneData = self.rr1.appenddata(self.FullDoneData,NewDoneData)
        #
        self.up[lower:upper] = inew
        self.xvalue[lower:upper] = xnew
        self.yvalue[lower:upper] = ynew
        self.pvalue[lower:upper] = pnew
        #
        self.lowerbound[lower:upper] = 1
        self.upperbound[lower:upper] = 1000
        #
        assert (~self.split[inew]).all(0)
        assert self.inplay[inew].all(0)
        #
        self.split[inew] = True
        self.split[lower:upper] = False
        self.inplay[lower:upper] = True
        #
        self.down[inew,xnew,ynew,pnew] = arangeic(newactive_length) + lower
        #
        newimpdone = newimpossible | newdone
```

```
        inew_id = fd_index[ivector_vert[newimpdone]]
        xnew_id = xvector_vert[newimpdone]
        ynew_id = yvector_vert[newimpdone]
        pnew_id = pvector_vert[newimpdone]
        #
        self.down[inew_id,xnew_id,ynew_id,pnew_id] = -1
        #
        new_availablexyp = self.rr1.availablexyp(newactive_length,newactive_prod
        new_availablexy = new_availablexyp.any(3)
        #
        self.availablexyp[lower:upper] = new_availablexyp
        self.availablexy[lower:upper] = new_availablexy
        #
        #
        new_fdlength = self.FullData['length']
        #
        self.FullData['info'][:,self.Pp.fulldata_location] = arangeic(new_fdleng
        #
        ##print("added",itp(newactive_length),"new instances to full data that n
        return



    def fd_location(self,Data):
        length = Data['length']
        assert length > 0
        return Data['info'][:,self.Pp.fulldata_location]



    def bounding_proofloop(self,Mstrat,fd_instances):
        #
        self.upperbound[fd_instances] = 1
        #
        Input = self.rr1.indexselectdata(self.FullData, fd_instances)
        #
        InitialActiveData = self.rr2.process(Input)
        activedetect, donedetect, impossibledetect = self.rr2.filterdata(Initial
        #
        ActivePool = self.rr1.detectsubdata(InitialActiveData,activedetect)
        #
        stepcount = 0
        #
        ##print("at step",itp(stepcount),"currently proof nodes for fd instances
        ##print(nump(self.upperbound[fd_instances]))
        #
        for i in range(self.rr4.prooflooplength):
            stepcount += 1
            prooflength = i
            if ActivePool['length'] > 0:
                #
                print(".",end = '')
                if (i%50) == 49:
                    print(" ")
                if (i%100) == 0:
```

```
                    print(i)
                #
                #
                ChunkData, cdetection = self.rr3.selectchunk(ActivePool)
                #
                #
                ProofCurrentData, DoneData = self.rr3.managesplit(Mstrat,ChunkDa
                #
                #
                ActivePool = self.rr4.transitionactive(ActivePool,cdetection,Pro
                # do the following before dropout
                #if dropoutlimit == 0:
                    #EDN = itt(ActivePool['length']).clone().to(torch.float)
                    #self.ECN += itt(CurrentData['length']).clone().to(torch.flo
                    # this from proofloop indicates that we should add to our no
                #
                if ProofCurrentData['length'] > 0:
                    current_fd_location = self.fd_location(ProofCurrentData)
                    fd_loc_list, fd_loc_counts = torch.unique(current_fd_locatio
                    self.upperbound[fd_loc_list] = self.upperbound[fd_loc_list]
                #
                ##print("at step",itp(stepcount),"currently proof nodes for fd i
                ##print(nump(self.upperbound[fd_instances]))
                #
                gcc = gc.collect()
                #
                if ActivePool['length'] == 0:
                    break
                if ActivePool['length'] > self.rr4.stopthreshold:
                    print("over threshold --------->>>>>>>>>>>>>>>>> stopping")
                    break
                #
                #
            if ActivePool['length'] == 0:
                break
            if ActivePool['length'] > self.rr4.stopthreshold:
                print("over threshold --------->>>>>>>>>>>>>>>>> stopping")
                break
            #
            #
        #
        print("|||")
        #
        ##print("proofs finished, resulting in proof nodes for fd instances in q
        ##print(nump(self.upperbound[fd_instances]))
        #
        return

    def calculate_current_upperbound(self):
        fdlength = self.FullData['length']
        if fdlength == 0:
            print("no upper bounds to calculate")
            return
        #
        fd_split = self.split[0:fdlength]
        fd_inplay = self.inplay[0:fdlength]
```

```
        fd_inplay = self.inplay[0:fdlength]
        #
        fd_current = fd_inplay & ~fd_split
        #
        fd_instances = arangeic(fdlength)[fd_current]
        #
        self.bounding_proofloop(self.Mm,fd_instances)
        return

    def recursive_bound_step(self):
        #
        fdlength = self.FullData['length']
        assert fdlength > 0
        #
        fd_up  = self.up[0:fdlength]
        fd_xvalue  = self.xvalue[0:fdlength]
        fd_yvalue  = self.yvalue[0:fdlength]
        fd_pvalue  = self.pvalue[0:fdlength]
        #
        fd_lowerbound  = self.lowerbound[0:fdlength]
        fd_upperbound  = self.upperbound[0:fdlength]
        #
        fd_down  = self.down[0:fdlength]
        #
        fd_split  = self.split[0:fdlength]
        fd_inplay  = self.inplay[0:fdlength]
        #
        fd_current = fd_inplay & ~fd_split
        fd_lookat = fd_inplay & fd_split
        fd_uppable = (fd_up >= 0)
        #
        fd_availablexyp  = self.availablexyp[0:fdlength]
        fd_availablexy  = self.availablexy[0:fdlength]
        #
        #fd_impdone_xyp = (fd_down == -1)
        #
        fd_lowerbound_xyp = torch.zeros((fdlength,self.alpha,self.alpha,self.bet
        fd_upperbound_xyp = torch.zeros((fdlength,self.alpha,self.alpha,self.bet
        #
        ivalue_uppable = fd_up[fd_uppable]
        xvalue_uppable = fd_xvalue[fd_uppable]
        yvalue_uppable = fd_yvalue[fd_uppable]
        pvalue_uppable = fd_pvalue[fd_uppable]
        #
        fd_lowerbound_xyp[ivalue_uppable,xvalue_uppable,yvalue_uppable,pvalue_up
        fd_upperbound_xyp[ivalue_uppable,xvalue_uppable,yvalue_uppable,pvalue_up
        #
        fd_lowerbound_xy_r = fd_lowerbound_xyp.sum(3).reshape(fdlength*self.alph
        fd_upperbound_xy_r = fd_upperbound_xyp.sum(3).reshape(fdlength*self.alph
        fd_availablexy_r = fd_availablexy.reshape(fdlength*self.alpha*self.alpha
        #
        fd_lowerbound_xy_r[~fd_availablexy_r] = 10000
        fd_upperbound_xy_r[~fd_availablexy_r] = 10000
        #
        fd_lowerbound_xy_rv = fd_lowerbound_xy_r.view(fdlength,self.alpha*self.a
        fd_upperbound_xy_rv = fd_upperbound_xy_r.view(fdlength,self.alpha*self.a
```

```
        fd_upperbound_xy_rv = fd_upperbound_xy_r.view(fdlength,self.alpha*self.a
        #
        fd_lowerbound_min,fdlbmindices = torch.min(fd_lowerbound_xy_rv,1)
        fd_upperbound_min,fdubmindices = torch.min(fd_upperbound_xy_rv,1)
        #
        fd_lowerbound_new = fd_lowerbound.clone()
        fd_upperbound_new = fd_upperbound.clone()
        fd_lowerbound_new[fd_lookat] = fd_lowerbound_min[fd_lookat]
        fd_upperbound_new[fd_lookat] = fd_upperbound_min[fd_lookat]
        #
        ##print("new lower bound")
        ##print(nump(fd_lowerbound_new))
        ##print("new upper bound")
        ##print(nump(fd_upperbound_new))
        #
        self.lowerbound[0:fdlength] = fd_lowerbound_new
        self.upperbound[0:fdlength] = fd_upperbound_new
        return

    def recursive_bound(self,iterations):
        #
        fdlength = self.FullData['length']
        assert fdlength > 0
        #
        lower_all = self.lowerbound[0:fdlength].sum(0)
        upper_all = self.upperbound[0:fdlength].sum(0)
        ##print("init with lower sum",itp(lower_all),"upper sum",itp(upper_all))
        for i in range(iterations):
            self.recursive_bound_step()
            lower_new = self.lowerbound[0:fdlength].sum(0)
            upper_new = self.upperbound[0:fdlength].sum(0)
            ##print("iteration",i,"with lower sum",itp(lower_new),"upper sum",it
            #
            if lower_new == lower_all and upper_new == upper_all:
                ##print("stabilizes")
                break
            else:
                lower_all = lower_new
                upper_all = upper_new
        ##print("done with recursive bound steps")
        return



    def remove_from_play(self,subset):
        #
        #
        fdlength = self.FullData['length']
        #
        fd_up  = self.up[0:fdlength]
        fd_up_mod = torch.clamp(fd_up,0,fdlength)
        #
        inplay_prev = self.inplay[0:fdlength].to(torch.int64).sum(0)
        ##print("previous in play count is",itp(inplay_prev))
        ##print("removing",itp(subset.to(torch.int64).sum(0)),"locations from pl
        #
```

```
        #
        self.inplay[0:fdlength] = self.inplay[0:fdlength] & (~subset)
        #
        for i in range(100):
            inplay_count = self.inplay[0:fdlength].to(torch.int64).sum(0)
            inplay_up = self.inplay[fd_up_mod]
            self.inplay[0:fdlength] = self.inplay[0:fdlength] & inplay_up
            if self.inplay[0:fdlength].to(torch.int64).sum(0) == inplay_count:
                break
        inplay_new = self.inplay[0:fdlength].to(torch.int64).sum(0)
        ##print("new in play count is",itp(inplay_new))
        #
        return

    def random_remove_from_play(self,threshold):
        fdlength = self.FullData['length']
        tirage = torch.rand((fdlength),device=Dvc)
        subset = (tirage < threshold)
        subset[0] = False
        self.remove_from_play(subset)
        return

    def leave_in_play(self,instance):  # for now we assume that this is at the f
        fdlength = self.FullData['length']
        subset = torch.ones((fdlength),dtype = torch.bool,device=Dvc)
        subset[0] = False
        subset[instance] = False
        self.remove_from_play(subset)
        return

    def initial_cut(self,x,y,p):
        fdlength = self.FullData['length']
        #assert fdlength == 28
        instance = self.down[0,x,y,p]
        subset = torch.ones((fdlength),dtype = torch.bool,device=Dvc)
        subset[0] = False
        subset[instance] = False
        self.remove_from_play(subset)
        return

    def prune(self):
        #
        fdlength = self.FullData['length']
        #
        fd_lowerbound = self.lowerbound[0:fdlength]
        fd_upperbound = self.upperbound[0:fdlength]
        fd_inplay = self.inplay[0:fdlength]
        #
        badlocations = (fd_lowerbound > fd_upperbound) & fd_inplay
        badlocations_count = badlocations.to(torch.int64).sum(0)
        #
        ##if badlocations_count > 0:
            ##print("warning, we found",itp(badlocations_count),"bad locations")
        ##else:
            ##print("all locations are good")
        #
```

```
        attained = (fd_lowerbound == fd_upperbound) & fd_inplay
        attained_count = attained.to(torch.int64).sum(0)
        #
        up_mod = torch.clamp(self.up[0:fdlength], 0, fdlength)
        #
        upperbound_up = self.upperbound[up_mod]
        nonoptimal = (fd_lowerbound + 1) >= upperbound_up
        nonoptimal[0] = False
        nonoptimal_count = nonoptimal.to(torch.int64).sum(0)
        #
        to_remove = nonoptimal | attained
        #
        ##print("found",itp(attained_count),"attained and",itp(nonoptimal_count)
        self.remove_from_play(to_remove)
        ##print("done pruning")
        return


    def combo_init(self):
        #print("------ initial combo segments ---------")
        #print("------ manage next stage (two iterations)")
        self.manage_next_stage()
        self.manage_next_stage()
        print("------ make initial cut at",self.cutx,self.cuty,self.cutp)
        self.initial_cut(self.cutx,self.cuty,self.cutp)
        #print("------ calculate current upper bound")
        self.calculate_current_upperbound()
        #print("------ recursive bound")
        self.recursive_bound(100)
        #print("------ prune")
        self.prune()
        #print("------ done with initial combo segment ---------")
        return

    def combo_step(self):
        #print("------ combo segment ---------")
        checkdone = self.check_done()
        if checkdone:
            print("|||")
            return 'done'
        #print("------ manage next stage")
        self.manage_next_stage()
        #print("------ calculate current upper bound")
        self.calculate_current_upperbound()
        #print("------ recursive bound")
        self.recursive_bound(100)
        #print("------ prune")
        self.prune()
        #print("------ done with combo segment ---------")
        return 'continue'

    def check_done(self):
        fdlength = self.FullData['length']
        #
        fd_lowerbound = self.lowerbound[0:fdlength]
```

```
        fd_upperbound = self.upperbound[0:fdlength]
        fd_inplay = self.inplay[0:fdlength]
        #
        inplay_count = fd_inplay.to(torch.int64).sum(0)
        if inplay_count > 1:
            return False
        else:
            assert fd_inplay[0]
            cutx = self.cutx
            cuty = self.cuty
            cutp = self.cutp
            cut_instance = self.down[0,cutx,cuty,cutp]
            assert self.lowerbound[cut_instance] == self.upperbound[cut_instance
            print("at initial cut location",cutx,cuty,cutp,"lower bound = upper
            print("this was for sigma =",self.sigma)
            #
            lb_next = torch.zeros((self.alpha,self.alpha),dtype = torch.int64,de
            ub_next = torch.zeros((self.alpha,self.alpha),dtype = torch.int64,de
            for x in range(self.alpha):
                for y in range(self.alpha):
                    if self.availablexy[cut_instance,x,y]:
                        lb_next[x,y] = 1
                        ub_next[x,y] = 1
                        for p in range(self.betaz):
                            if self.availablexyp[cut_instance,x,y,p]:
                                down_xyp = self.down[cut_instance,x,y,p]
                                if down_xyp > 0:
                                    lb_next_xyp = self.lowerbound[down_xyp]
                                    assert lb_next_xyp > 0
                                    lb_next[x,y] += lb_next_xyp
                                    #
                                    ub_next_xyp = self.upperbound[down_xyp]
                                    ub_next[x,y] += ub_next_xyp
            #
            ##print("lower bounds for the next cut locations x,y are as follows:
            ##print(nump(lb_next))
            ##print("upper bounds")
            ##print(nump(ub_next))
            ##print("full data length was",itp(fdlength))
            ##self.show_neural_network_results()
            #
        return True

    def check_done_print(self):
        fdlength = self.FullData['length']
        #
        fd_lowerbound = self.lowerbound[0:fdlength]
        fd_upperbound = self.upperbound[0:fdlength]
        fd_inplay = self.inplay[0:fdlength]
        #
        inplay_count = fd_inplay.to(torch.int64).sum(0)
        if inplay_count > 1:
            print("not done: there are remaining locations in play")
            return False
        else:
```

```
            assert fd_inplay[0]
            cutx = self.cutx
            cuty = self.cuty
            cutp = self.cutp
            cut_instance = self.down[0,cutx,cuty,cutp]
            assert self.lowerbound[cut_instance] == self.upperbound[cut_instance
            print("at initial cut location",cutx,cuty,cutp,"lower bound = upper
            print("this was for sigma =",self.sigma)
            #
            lb_next = torch.zeros((self.alpha,self.alpha),dtype = torch.int64,de
            ub_next = torch.zeros((self.alpha,self.alpha),dtype = torch.int64,de
            for x in range(self.alpha):
                for y in range(self.alpha):
                    if self.availablexy[cut_instance,x,y]:
                        lb_next[x,y] = 1
                        ub_next[x,y] = 1
                        for p in range(self.betaz):
                            if self.availablexyp[cut_instance,x,y,p]:
                                down_xyp = self.down[cut_instance,x,y,p]
                                if down_xyp > 0:
                                    lb_next_xyp = self.lowerbound[down_xyp]
                                    assert lb_next_xyp > 0
                                    lb_next[x,y] += lb_next_xyp
                                    #
                                    ub_next_xyp = self.upperbound[down_xyp]
                                    ub_next[x,y] += ub_next_xyp
            #
            print("lower bounds for the next cut locations x,y are as follows:")
            print(nump(lb_next - 1))
            print("upper bounds")
            print(nump(ub_next - 1))
            print("lower and upper bounds should coincide. Add 1 to plug back in
            print("full data length was",itp(fdlength))
            #self.show_neural_network_results()
            #
        return True

    def show_neural_network_results(self):  # not currently working, also our ne
        #
        fdlength = self.FullData['length']
        #
        subset = torch.zeros((fdlength),dtype = torch.bool,device=Dvc)
        subset[0:500] = True
        TruncatedFullData = self.rr1.detectsubdata(self.FullData,subset)
        fd_network_output = self.Mm.network(TruncatedFullData).detach()
        net2 = M.network2(TruncatedFullData)
        fd_network2_output = net2.detach()
        #
        fdn2_exp_rootv = (10**fd_network2_output[0]).view(self.alpha*self.alpha)
        avxy_rootv = self.availablexy[0].view(self.alpha*self.alpha)
        fdn2_exp_rootv[~avxy_rootv] = 0.
        fdn2_exp_rootmod = fdn2_exp_rootv.view(self.alpha,self.alpha)
        print("network 2 output at root")
        print(numpr(fdn2_exp_rootmod,2))
        #
```

```
        cutx = self.cutx
        cuty = self.cuty
        cutp = self.cutp
        cut_instance = self.down[0,cutx,cuty,cutp]
        #
        fd_network2_cut_instance = fd_network2_output[cut_instance]
        fdn2_exp_v = (10**fd_network2_cut_instance).view(self.alpha*self.alpha)
        print("network 2 gives the following matrix (after exponentiating base 1
        avxy_v = self.availablexy[cut_instance].view(self.alpha*self.alpha)
        fdn2_exp_v[~avxy_v] = 0.
        fdn2_exp_mod = fdn2_exp_v.view(self.alpha,self.alpha)
        print(numpr(fdn2_exp_mod,2))
        #
        fdn_exp_sum = torch.zeros((self.alpha,self.alpha),dtype = torch.float,de
        for x in range(self.alpha):
            for y in range(self.alpha):
                if self.availablexy[cut_instance,x,y]:
                    fdn_exp_sum[x,y] = 1.
                    for p in range(self.betaz):
                        if self.availablexyp[cut_instance,x,y,p]:
                            down_xyp = self.down[cut_instance,x,y,p]
                            if down_xyp > 0:
                                fdn_next_xyp = 10**(fd_network_output[down_xyp])
                                fdn_exp_sum[x,y] += fdn_next_xyp
        #
        print("summing the results of the first network gives the following matr
        print(numpr(fdn_exp_sum,2))
        print("= = = = = =")
        return

    def combo_all(self):
        self.combo_init()
        for i in range(20):
            ##print("===   ===   ===   ===   ===   ===   ===   ===   ===   ===
            ##print("===   ===   ===   ===   ===   ===   ===   ===   ===   ===
            print(">>>>>>>>>>>>>>>>>>>> combo step iteration number",i)
            ##print("===   ===   ===   ===   ===   ===   ===   ===   ===   ===
            ##print("===   ===   ===   ===   ===   ===   ===   ===   ===   ===
            step_result = self.combo_step()
            if step_result == 'done':
                break
        ##print("===   ===   ===   ===   ===   ===   ===   ===   ===   ===   ===
        print("combo all is completed.")
        return




class MinimizerHistory :    # this becomes the first element of the relations da
    def __init__(self,model):
        #
        #
        self.Mm = model
        self.Pp = self.Mm.pp
        self.Dd = Driver(self.Pp)
        #
```

```
        #
        sigma = int(input("input sigma : "))
        self.sigma = sigma
        #
        print("Minimizer History for sigma =",sigma)
        print("with profile_filter_on =",self.Pp.profile_filter_on,"and halfones

        #
        self.rr4 = Relations4(self.Pp)
        self.rr3 = self.rr4.rr3
        self.rr2 = self.rr4.rr2
        self.rr1 = self.rr4.rr1
        self.alpha = self.Pp.alpha
        self.alpha2 = self.Pp.alpha2
        self.alpha3 = self.Pp.alpha3
        self.alpha3z = self.Pp.alpha3z
        self.beta = self.Pp.beta
        self.betaz = self.Pp.betaz
        #
        instancevector, trainingvector, proof_title = self.Dd.InOne(self.sigma)
        self.InitialData = self.Dd.initialdata(instancevector,0)
        #
        self.results = torch.zeros((self.alpha,self.alpha,self.betaz),dtype = to
        #
        idlength = self.InitialData['length']
        idprod = self.InitialData['prod']
        self.availablexyp = self.rr1.availablexyp(idlength,idprod)[0]
        #

    def print_results(self):
        #
        results_sum = self.results.sum(2)
        #
        minimal = results_sum[0,0]
        for x in range(self.alpha):
            for y in range(self.alpha):
                if results_sum[x,y] < minimal:
                    minimal = results_sum[x,y]
        root_minimum = minimal + 1
        print("=+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=
        print("=+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=
        print("               aggregation of results for sigma =",itp(self.sigma
        print("=+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=
        print("=+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=   =+=
        #
        print("summed results according to initial cut location are:")
        print(nump(results_sum))
        print("the minimal number of nodes including the root is",itp(root_minim
        print("this was with profile_filter_on =",self.Pp.profile_filter_on,"and
        print("-----------------------------------------------------------------
        return

    def minimize_all(self):
        #
        for x in range(self.alpha):
            for y in range(self.alpha):
```

```
                for p in range(self.betaz):
                    if self.availablexyp[x,y,p]:
                        Min = Minimizer(self.Mm,self.sigma,x,y,p)
                        cut_instance = Min.down[0,x,y,p]
                        assert Min.lowerbound[cut_instance] == Min.upperbound[cu
                        self.results[x,y,p] = Min.lowerbound[cut_instance]
        self.print_results()
        return
```

```
Pp.profile_filter_on = True
Pp.halfones_filter_on = True
```

```
MH = MinimizerHistory(Mmr)  # asks to choose sigma
MH.minimize_all()  # this does all the cases in a row
```

```
def stopcompile():
```

```
Min = Minimizer(Mm,3,0,0,0)  # individual cases: sigma, x, y, p
Min.check_done_print()
#time.sleep(60)
```

```
Min = Minimizer(Mm,3,0,0,1) # individual cases: sigma, x, y, p
Min.check_done_print()
#time.sleep(60)
```

```
Min = Minimizer(Mm,3,0,0,2) # individual cases: sigma, x, y, p
Min.check_done_print()
#time.sleep(60)
```

```

```

```
###### next: a class that can be used to search for specific examples, it was a
```

```
class FindWeirdStuff :
    def __init__(self,Dd,Mm):
        #
        #
        self.Dd = Dd
        self.Ll = self.Dd.Ll
        self.Mm = Mm
        #
        self.Pp = self.Dd.Pp
        #
        self.rr4 = self.Dd.rr4
        self.rr3 = self.rr4.rr3
        self.rr2 = self.rr4.rr2
```

```
        self.rr1 = self.rr4.rr1
        self.alpha = self.Pp.alpha
        self.alpha2 = self.Pp.alpha2
        self.alpha3 = self.Pp.alpha3
        self.alpha3z = self.Pp.alpha3z
        self.beta = self.Pp.beta
        self.betaz = self.Pp.betaz
        #

    def sample_box(self,xlower,xupper,ylower,yupper):
        #
        smb,DataBatch,scorebatch = self.Ll.selectminibatch(500)
        #
        if not smb:
            print("select mini batch fails")
            return False,None
        predictedscore = self.Mm.network(DataBatch)
        #
        detectionx = (xlower < scorebatch) & (scorebatch < xupper)
        detectiony = (ylower < predictedscore) & (predictedscore < yupper)
        detection = detectionx & detectiony
        #
        DetectedData = self.rr1.detectsubdata(DataBatch,detection)
        #
        return True, DetectedData

    def printright(self,right,loc):
        #
        a=self.alpha
        a2=a*a
        a3=a*a*a
        a3z = a3+1
        b=self.beta
        bz = b+1
        #
        prarray = torch.zeros((bz,a),dtype = torch.int64,device=Dvc)
        for x in range(bz):
            for y in range(a):
                column = right[loc,x,y]
                prarray[x,y] = self.Dd.printcolumn2(column)
        print(nump(prarray))
        return

    def print_prod_left_right(self,Data,i):
        prod = Data['prod']
        left = Data['left']
        right = Data['right']
        #
        print("-----------------------------------------------------")
        print("at location",itp(i),"the prod, left and right are respectively")
        self.Dd.printprod(prod,i)
        self.Dd.printleft(left,i)
        self.printright(right,i)
        print("-----------------------------------------------------")
        return
```

```
69
70     def print_column_bool(self,column):
71         clength = len(column)
72         toprint = 3 * (10 ** (clength))
73         for q in range(clength):
74             toprint += (column[q].to(torch.int64) * (10 ** q) )
75         return toprint
76
77     def print_bool_tensor(self,a,b,c,btensor):
78         prarray = torch.zeros((a,b),dtype = torch.int64,device=Dvc)
79         for x in range(a):
80             for y in range(b):
81                 column = btensor[x,y]
82                 prarray[x,y] = self.print_column_bool(column)
83         print(nump(prarray))
84         return
85
86
87     def print_one_sample_from_box(self,xlower,xupper,ylower,yupper):
88         #
89         sb,DetectedData = self.sample_box(xlower,xupper,ylower,yupper)
90         #
91         if not sb:
92             print("exit from one sample box")
93             return
94         length = DetectedData['length']
95         if length == 0:
96             print("didn't detect any samples in this box")
97             return
98         #
99         AssocData = self.rr2.process(DetectedData)
100        activedetect, donedetect, impossibledetect = self.rr2.filterdata(AssocDa
101        permutation = torch.randperm(length,device=Dvc)
102        loc = permutation[0]
103        print("before processing:")
104        self.print_prod_left_right(DetectedData,loc)
105        print("after processing:")
106        self.print_prod_left_right(AssocData,loc)
107        print("the status of this location is :",end = ' ')
108        if activedetect[loc]:
109            print("active")
110        if donedetect[loc]:
111            print("done")
112        if impossibledetect[loc]:
113            print("impossible")
114        return
115
116    def av_root(self,DataToSplit,i):
117        a = self.alpha
118        a2 = self.alpha2
119        a2z = self.alpha2 +1
120        a3 = self.alpha3
121        a3z = self.alpha3z
122        b = self.beta
123        bz = self.betaz
```

```
        #
        #
        length = DataToSplit['length']
        prod = DataToSplit['prod']
        #
        availablexypi = self.rr1.availablexyp(length,prod).view(length,a,a,bz)[i
        #
        print("at root, availablexyp is:")
        self.print_bool_tensor(a,a,bz,availablexypi)
        #
        return

    def split_by_hand(self,DataToSplit,i,x,y,p):
        a = self.alpha
        a2 = self.alpha2
        a2z = self.alpha2 +1
        a3 = self.alpha3
        a3z = self.alpha3z
        b = self.beta
        bz = self.betaz
        #
        #
        length = DataToSplit['length']
        prod = DataToSplit['prod']
        #
        availablexypi = self.rr1.availablexyp(length,prod).view(length,a,a,bz)[i
        #
        available_instance = availablexypi[x,y,p]
        if available_instance:
            print("this instance",itp(x),itp(y),itp(p),"is available")
        else:
            print("this instance",itp(x),itp(y),itp(p),"is not available")
            return
        #
        ivector = torch.zeros((1),dtype = torch.int64,device=Dvc)
        ivector[:] = i
        xvector = torch.zeros((1),dtype = torch.int64,device=Dvc)
        xvector[:] = x
        yvector = torch.zeros((1),dtype = torch.int64,device=Dvc)
        yvector[:] = y
        pvector = torch.zeros((1),dtype = torch.int64,device=Dvc)
        pvector[:] = p
        #
        #
        NewData = self.rr1.upsplitting(DataToSplit,ivector,xvector,yvector,pvect
        #
        #
        ndlength = NewData['length']
        assert ndlength == 1
        #
        AssocNewData = self.rr2.process(NewData)
        #
        newactive,newdone,newimpossible = self.rr2.filterdata(AssocNewData)
        if newactive[0]:
            print("active")
```

```
        if newdone[0]:
            print("done")
        if newimpossible[0]:
            print("impossible")
        #
        AND_prod = AssocNewData['prod']
        AND_length = 1
        AND_availablexyp = self.rr1.availablexyp(AND_length,AND_prod).view(1,a,a
        print("after cut at",itp(x),itp(y),itp(p),"availablexyp is:")
        self.print_bool_tensor(a,a,bz,AND_availablexyp)
        print("prod is")
        self.print_bool_tensor(a,a,bz,AND_prod[0])
        #
        #
        return AND_prod

    def show_cut_column(self,sigma,x,y):  # shows the results of (x,y,p) cut sta
        #
        instancevector, trainingvector, title_text = self.Dd.InOne(sigma)
        InitialData = self.Dd.initialdata(instancevector, 0)
        #
        self.av_root(InitialData,0)
        #
        for p in range(self.betaz):
            #
            andprod = self.split_by_hand(InitialData,0,x,y,p)
        return

    def searchprod(self,Data,trprod):
        #
        a = self.alpha
        bz = self.betaz
        #
        length = Data['length']
        prod = Data['prod']
        #
        if length == 0:
            print("length is 0")
            return 0,None
        prodv = prod.view(length,a*a*bz)
        trprodv = trprod.view(1,a*a*bz).expand(length,a*a*bz)
        #
        detection = (prodv == trprodv).all(1)
        #
        detected_indices = arangeic(length)[detection]
        detected_length = detection.to(torch.int64).sum(0)
        print("detected",itp(detected_length),"occurences out of",itp(length))
        return detected_length, detected_indices


    def tracer(self,trprod):
        #
        print("Examples:",end=' ')
        dl,di = self.searchprod(self.Ll.Examples,trprod)
        print("ExamplesPrePool:",end=' ')
```

```
        dl,di = self.searchprod(self.Ll.ExamplesPrePool,trprod)
        print("ExplorePrePool:",end=' ')
        dl,di = self.searchprod(self.Ll.ExplorePrePool,trprod)
        print("OutlierPrePool:",end=' ')
        dl,di = self.searchprod(self.Ll.OutlierPrePool,trprod)
        #
        #
        return

    def tracer_root(self,sigma):
        #
        instancevector, trainingvector, title_text = self.Dd.InOne(sigma)
        InitialData = self.Dd.initialdata(instancevector, 0)
        #
        trprod = InitialData['prod'][0]
        #
        self.tracer(trprod)
        #
        return

    def tracer_subroot(self,sigma,x,y,p):
        #
        instancevector, trainingvector, title_text = self.Dd.InOne(sigma)
        InitialData = self.Dd.initialdata(instancevector,0)
        #
        trprod = self.split_by_hand(InitialData,0,x,y,p)
        #
        self.tracer(trprod)
        #
        return

    def trace(self,stringtoprint,Data,sigma,x,y,p):
        #
        instancevector, trainingvector, title_text = self.Dd.InOne(sigma)
        InitialData = self.Dd.initialdata(instancevector,0)
        #
        trprod = self.split_by_hand(InitialData,0,x,y,p)
        #
        print(stringtoprint,end=' ')
        dl,di = self.searchprod(Data,trprod)
        return
```

```
Fws = FindWeirdStuff(Dd,Mm)
```

```
### a few things to do with that
```

```
Fws.tracer_root(5)
```

```
Fws.tracer_subroot(5,0,0,0)
```

```
Fws.show_cut_column(5,2,1)
```

```
Fws.print_one_sample_from_box(0.25,0.35,0.0,0.2)
```

```
Fws.print_one_sample_from_box(0.0,0.05,0.0,0.2)
```

```

```

```

```

```
##############################################################################
##############################################################################
##############################################################################

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

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

##############################################################################
##############################################################################
##############################################################################
```

```

```

```

```

```

```

```

```

```

```