# Appendix

1. **Main codes**

In the first step, the Data\_Generator.py should be run to extract the features and SAT runtime for each circuit. This code generates two pickle files with \_X and \_Y suffixes that instead of each \_, there is circuit name.

* **Data\_Generator.py:**

import numpy as np

import os

import itertools

import copy

from collections import Counter

from bench2cnf import simple\_read\_bench, tseytin\_t, cnf\_to\_matrix

import pickle as pk

import scipy.stats as ss

import pandas as pd

#from scipy.sparse import csgraph

#from grakel.graph import floyd\_warshall

from scipy.stats import skew

from scipy.stats import kurtosis

def Time\_calculation(Time\_List):

Time\_Target = float(Time\_list[0])\*60 + float(Time\_list[1].split('s')[0])

return Time\_Target

def ent(data):

""" input probabilities to get the entropy """

entropy = ss.entropy(data)

return entropy

data\_dir = os.path.join(os.getcwd(),'Dataset 2\EncryptedBenches')

df = pd.read\_csv(os.path.join(os.getcwd(),'Dataset 2\decryption\_time.csv'))

df1 = df[["CircuitName"]]

CircuitNames = df1.values.tolist()

CircuitNames = [x[0] for x in CircuitNames]

CircuitNames = set(CircuitNames)

for each\_benchmark in CircuitNames:

each\_df = df.loc[df['CircuitName']==each\_benchmark]

All\_data\_features = []

Time\_Targets = []

#print(each\_benchmark)

for ind in each\_df.index:

Time = each\_df['Time'][ind]

if Time =='-1':

#continue

Time\_Target = float(Time)

else:

Time\_list = Time.split('m') # The first and second elements are minute and second, respectively.

Time\_Target = Time\_calculation(Time\_list)

CircuitName = each\_df['CircuitName'][ind]

EncryptionScheme = each\_df['EncryptionScheme'][ind]

KeyLength = each\_df['KeyLength'][ind]

Fraction = each\_df['Fraction'][ind]

bench\_file\_suffix = "\_keys" if Fraction== 0 else "\_per" # It determines the name of bench file which could be formed by \_per or \_keys

percent\_obfuscated = Fraction if Fraction != 0 else KeyLength

Primary\_Link\_0 = os.path.join(data\_dir,CircuitName,EncryptionScheme,'1')

Primary\_Link\_1 = os.path.join(Primary\_Link\_0,str(percent\_obfuscated) + bench\_file\_suffix + ".bench")

wires = simple\_read\_bench(Primary\_Link\_1)

cnf\_clause\_count, cnf\_content = tseytin\_t(wires)

incidence\_mat = cnf\_to\_matrix(cnf\_content)

if incidence\_mat.size == 0:

continue

feat = []

# check if there is zero rows

zero\_row\_id = (incidence\_mat == 0).all(1)

incidence\_mat = incidence\_mat[~zero\_row\_id] # remove all zero rows

cnf\_clause\_count, cnf\_variable\_counts = incidence\_mat.shape # refresh var and clause num

# derive literal degree and clause degree

var\_degree = np.sum(np.abs(incidence\_mat), axis=0)

clause\_degree = np.sum(np.abs(incidence\_mat), axis=1)

# build literal graph and extract literal degree and features

var\_graph = np.zeros((cnf\_variable\_counts, cnf\_variable\_counts))

for i in range(cnf\_clause\_count):

non\_zero\_list = np.where(incidence\_mat[i] != 0)

pair\_combo = [pair for pair in itertools.product(non\_zero\_list[0], repeat=2)]

for p in pair\_combo:

var\_graph[p] = 1

var\_graph\_degree = np.sum(var\_graph, axis=0)

#var\_graph\_floyd = floyd\_warshall(var\_graph)

#var\_graph\_floyd\_degree = np.sum(var\_graph\_floyd, axis=0)

# extract positive and negative features from literal-clause graph

positive\_mat = copy.deepcopy(incidence\_mat)

positive\_mat[positive\_mat == -1] = 0

negative\_mat = copy.deepcopy(incidence\_mat)

negative\_mat[negative\_mat == 1] = 0

negative\_mat[negative\_mat == -1] = 1

positive\_mat\_0 = np.sum(positive\_mat, axis=0)

negative\_mat\_0 = np.sum(negative\_mat, axis=0)

positive\_mat\_1 = np.sum(positive\_mat, axis=1)

negative\_mat\_1 = np.sum(negative\_mat, axis=1)

p\_ratio\_0 = positive\_mat\_0 / (positive\_mat\_0 + negative\_mat\_0)

n\_ratio\_0 = negative\_mat\_0 / (positive\_mat\_0 + negative\_mat\_0)

p\_ratio\_1 = positive\_mat\_1 / (negative\_mat\_1 + positive\_mat\_1)

n\_ratio\_1 = negative\_mat\_1 / (negative\_mat\_1 + positive\_mat\_1)

# count ratio of binary and ternary clause

bin\_tern\_cnt = Counter(np.sum(np.abs(incidence\_mat), axis=1))

bin\_ratio = bin\_tern\_cnt[2] / sum(bin\_tern\_cnt.values())

tern\_ratio = bin\_tern\_cnt[3] / sum(bin\_tern\_cnt.values())

feat.append(tern\_ratio)

feat.append(bin\_ratio)

feat.append(cnf\_variable\_counts) # 0

feat.append(cnf\_clause\_count) # 1

feat.append(float(cnf\_clause\_count / cnf\_variable\_counts)) # 2

# extract entropy for each feature

feat.append(ent(var\_degree))

feat.append(ent(clause\_degree))

feat.append(ent(var\_graph\_degree))

feat.append(ent(p\_ratio\_0))

feat.append(ent(p\_ratio\_1))

feat.append(ent(n\_ratio\_0))

feat.append(ent(n\_ratio\_1))

#feat.append(ent(var\_graph\_floyd\_degree))

feat.append(np.mean(var\_degree))

feat.append(np.mean(clause\_degree))

feat.append(np.mean(var\_graph\_degree))

feat.append(np.mean(p\_ratio\_0))

feat.append(np.mean(p\_ratio\_1))

feat.append(np.mean(n\_ratio\_0))

feat.append(np.mean(n\_ratio\_1))

#feat.append(np.mean(var\_graph\_floyd\_degree))

feat.append(np.var(var\_degree))

feat.append(np.var(clause\_degree))

feat.append(np.var(var\_graph\_degree))

feat.append(np.var(p\_ratio\_0))

feat.append(np.var(p\_ratio\_1))

feat.append(np.var(n\_ratio\_0))

feat.append(np.var(n\_ratio\_1))

#feat.append(np.var(var\_graph\_floyd\_degree))

feat.append(skew(var\_degree))

feat.append(skew(clause\_degree))

feat.append(skew(var\_graph\_degree))

feat.append(skew(p\_ratio\_0))

feat.append(skew(p\_ratio\_1))

feat.append(skew(n\_ratio\_0))

feat.append(skew(n\_ratio\_1))

#feat.append(skew(var\_graph\_floyd\_degree))

feat.append(kurtosis(var\_degree))

feat.append(kurtosis(clause\_degree))

feat.append(kurtosis(var\_graph\_degree))

feat.append(kurtosis(p\_ratio\_0))

feat.append(kurtosis(p\_ratio\_1))

feat.append(kurtosis(n\_ratio\_0))

feat.append(kurtosis(n\_ratio\_1))

#feat.append(kurtosis(var\_graph\_floyd\_degree))

feat.append([var\_degree])

feat.append([clause\_degree])

feat.append([var\_graph\_degree])

feat.append([p\_ratio\_0])

feat.append([p\_ratio\_1])

feat.append([n\_ratio\_0])

feat.append([n\_ratio\_1])

#feat.append(var\_graph\_floyd\_degree)

All\_data\_features.append(feat)

Time\_Targets.append(Time\_Target)

pk.dump(All\_data\_features, open(os.path.join(os.getcwd(),'{}\_X.pk'.format(each\_benchmark)), 'wb'))

pk.dump(Time\_Targets, open(os.path.join(os.getcwd(),'{}\_Y.pk'.format(each\_benchmark)), 'wb'))

print("Data Generating for " + each\_benchmark + " is completed")

* **Kmeans\_elbow\_code.py:**

In order to determine the best number of clusters for IC labeling, the Kmeans\_elbow\_code.py should be run after running Data\_Generator.py.

import numpy as np

import \_pickle as pk

import os

from sklearn.cluster import KMeans

from yellowbrick.cluster import KElbowVisualizer

CircuitNames = ['74181','74182','74283','74L85','c1196','c1238','c1355','c17',

'c2670','c3540','c432','c499','c5315','c6288','c7552','c880','s1196','s1196a',

's1238','s1238a']

All\_times = []

for each\_benchmark in CircuitNames:

Times = pk.load(open(os.path.join(os.getcwd(),each\_benchmark + "\_Y.pk"), 'rb'))

All\_times.extend(Times)

Times\_array = np.asarray(All\_times)

Times\_array = np.expand\_dims(Times\_array,axis=1)

###Removing -1 values from the Times\_array

r,c = np.where(Times\_array == -1)

Times\_array = np.delete(Times\_array, r)

Times\_array = np.expand\_dims(Times\_array,axis=1)

kmeans = KMeans()

Max\_Number\_clusters = 7

elbow\_obj = KElbowVisualizer(kmeans, k=Max\_Number\_clusters,

metric='silhouette',

timings=False)

# Fit the data to the elbow technique for determining the optimum number of clusters

elbow\_obj.fit(Times\_array)

elbow\_obj.show(outpath="elbow\_result.png")

* **Kmeans\_code.py**

The third main code of this project is Kmeans\_code.py. By running this code, k-means clustering method is applied to the SAT runtimes with different number of clusters. This code is used for investing the threshold values between clusters and formulating the mapping labeling function.

import numpy as np

import \_pickle as pk

import os

import matplotlib.pyplot as plt

from sklearn.cluster import KMeans

#from scipy.io import savemat

def Clusters\_division(Times\_array,Clustering\_result,n\_clusters):

Clutsers = dict()

for x in range(0,n\_clusters):

Clutsers[x] = Times\_array[Clustering\_result==x]

return Clutsers

def plot\_clusters(Clusters,n\_clusters, val):

plt.figure(figsize=(10,5), dpi=150)

Legends = ["Cluster " + str(x) for x in range(0,n\_clusters)]

for x,y in Clusters.items():

plt.plot(y, np.zeros\_like(y) + val, 'x')

plt.ylim(-0.01, 0.01)

plt.legend(Legends)

plt.title("Scatter plot and clustering result with K="+str(n\_clusters))

plt.xlabel("SAT Runtime")

plt.savefig("K-means\_result\_n="+ str(n\_clusters) + ".jpg")

plt.show()

plt.close()

CircuitNames = ['74181','74182','74283','74L85','c1196','c1238','c1355','c17',

'c2670','c3540','c432','c499','c5315','c6288','c7552','c880','s1196','s1196a',

's1238','s1238a']

All\_times = []

for each\_benchmark in CircuitNames:

Times = pk.load(open(os.path.join(os.getcwd(),each\_benchmark + "\_Y.pk"), 'rb'))

All\_times.extend(Times)

Times\_array = np.asarray(All\_times)

Times\_array = np.expand\_dims(Times\_array,axis=1)

### Removing -1 values from the Times\_array

r,c = np.where(Times\_array == -1)

Times\_array = np.delete(Times\_array, r)

Times\_array = np.expand\_dims(Times\_array,axis=1)

Number\_of\_clusters = 5

Each\_clutsering\_result = dict()

Criterions = dict()

for n\_clusters in range(2,Number\_of\_clusters+1):

kmeans = KMeans(n\_clusters=n\_clusters, random\_state=0)

kmeans.fit(Times\_array)

Clustering\_result = kmeans.labels\_

Criterions[n\_clusters] = kmeans.inertia\_

Clusters = Clusters\_division(Times\_array,Clustering\_result,n\_clusters)

Each\_clutsering\_result[n\_clusters] = Clusters

plot\_clusters(Clusters,n\_clusters, 0)

* **SFS\_code.py**

After determine the threshold value and data generating, SFS\_code.py should be run to select the best scalar features. In this code, m2 is a variable, which determines the threshold value in seconds between SAT-resilient and SAT-vulnerable classes. Also, CircuitNames should be assigned to a circuit name that the proposed framework will be applied to it.

import numpy as np

import sklearn

from mlxtend.feature\_selection import SequentialFeatureSelector as SFS

from mlxtend.plotting import plot\_sequential\_feature\_selection as plot\_sfs

import matplotlib.pyplot as plt

from scipy.io import savemat

from util import read\_ic

def Best\_Comb(Results):

Avg\_metrics = []

for index, each\_result in Results.items():

Avg\_metrics.append(each\_result["avg\_score"])

Best\_com\_index = Avg\_metrics.index(max(Avg\_metrics))

return Results[Best\_com\_index+1]['feature\_idx'], np.asarray(Avg\_metrics)

CircuitNames = ['c1355','c2670','c3540','c5315','c6288','c7552']

CircuitNames = [CircuitNames[1]]

Number\_of\_scaler\_features = 40

m2 = 59830

\_, X, Y, \_, \_, \_ = read\_ic(CircuitNames,Number\_of\_scaler\_features,m2,False)

X = np.asarray(X)

Y = np.asarray(Y)

### Feature Selection

from sklearn.neighbors import KNeighborsClassifier

Model = KNeighborsClassifier()

sfs1 = SFS(Model,

k\_features = 40,

forward = True,

cv = 10)

sfs1.fit(X,Y)

Results = sfs1.subsets\_

SF, Metrics = Best\_Comb(Results)

fig = plot\_sfs(sfs1.get\_metric\_dict())

plt.savefig('my\_plot.png')

plt.close

Mydic = {"Selected\_Features":SF, "Metrics":Metrics}

savemat('SF.mat',Mydic)

* **Run\_iterate\_10fold.py**

In the next step, Run\_iterate\_10fold.py should be run to train and validate the CNF-NET model for classifying each IC to SAT-resilient and SAT-vulnerable classes. In this code, m2 is a variable, which determines the threshold value in seconds between SAT-resilient and SAT-vulnerable classes. It is worth mention that this code trains and validates the CNF-NET model using 10-fold cross-validation technique.

import time

import argparse

import numpy as np

import os

import torch

import torch.nn as nn

import torch.optim as optim

from operator import itemgetter

from torch.utils.data import DataLoader

import pickle as pk

from sklearn.model\_selection import train\_test\_split

from sklearn.model\_selection import KFold

from model import CNFNet

from util import GraphDataset, chunks, print\_network, plot\_metric, read\_ic, read\_ic\_10fold

from scipy.io import savemat, loadmat

def Evaluate\_CM(L\_test,Predict):

NTP = 0

NTN = 0

NFP = 0

NFN = 0

for i in range(len(Predict)):

if (Predict[i]==1)and(L\_test[i]==1):

NTP = NTP + 1

elif (Predict[i]==0)and(L\_test[i]==0):

NTN = NTN + 1

elif (Predict[i]==1)and(L\_test[i]==0):

NFP = NFP + 1

else:

NFN = NFN + 1

Accuracy = (NTP+NTN)/(NTP+NTN+NFP+NFN)

Sensitivity = NTP/(NTP+NFN)

try:

Specificity = NTN/(NTN+NFP)

except:

Specificity = np.nan

F1 = (2\*NTP)/((2\*NTP)+NFP+NFN)

Loss = 1 - Accuracy

return Accuracy, Sensitivity, Specificity, F1, Loss

def Prediction(output,model):

\_, predicted = torch.max(output.data, 1)

return predicted

def Obtain\_Accuracy(output,labels,model):

correct = 0

total = 0

\_, predicted = torch.max(output.data, 1)

total += labels.size(0)

correct += (predicted == labels).sum().item()

Accuracy = correct/total

return Accuracy

parser = argparse.ArgumentParser()

parser.add\_argument('--no-cuda', action='store\_true', default=False, help='Disables CUDA training.')

parser.add\_argument('--fastmode', action='store\_true', default=False, help='Validate during training pass.')

parser.add\_argument('--seed', type=int, default=8, help='Random seed.')

parser.add\_argument('--epochs', type=int, default=1, help='Number of epochs to train.')

parser.add\_argument('--lr', type=float, default=0.01, help='Initial learning rate.')

parser.add\_argument('--weight\_decay', type=float, default=5e-4, help='Weight decay (L2 loss on parameters).')

parser.add\_argument('--hidden', type=int, default=32, help='Number of hidden units.')

parser.add\_argument('--dropout', type=float, default=0.5, help='Dropout rate (1 - keep probability).')

parser.add\_argument('--data', type=str, default='c499', help='Dataset name')

#CircuitNames = ['74181','74182','74283','74L85','c1196','c1238','c1355','c17',

# 'c2670','c3540','c432','c499','c5315','c6288','c7552','c880','s1196','s1196a',

# 's1238','s1238a']

CircuitNames = ['c1355','c2670','c3540','c5315','c6288','c7552']

CircuitNames = [CircuitNames[1]]

args = parser.parse\_args()

args.cuda = True

# Loading Selected Features indexes

Dict\_SF = loadmat("SF.mat")

SF = Dict\_SF["Selected\_Features"]

# Training setting

args.epochs = 25

args.num\_feat = SF.size

args.batch\_size = 32

# FC setting

args.energy\_input\_dim = SF.size + 7

Number\_of\_scaler\_features = 40

#m1 = 16780

m2 = 59830

Num\_iter = 1

All\_test\_acc = []

All\_test\_loss = []

All\_test\_predic = []

All\_test\_label = []

inc\_feat, feats, times = read\_ic\_10fold(CircuitNames,Number\_of\_scaler\_features,

m2,True,SF)

kf = KFold(n\_splits=2)

kf.get\_n\_splits(feats,times)

ACs = []

SEs = []

SPs = []

F1s = []

NPs = []

NNs = []

i=0

for train\_index, test\_index in kf.split(feats,times):

i = i + 1

inc\_feat\_tr, inc\_feat\_te, feat\_tr, feat\_te, labels\_tr, labels\_te = train\_test\_split([inc\_feat[p] for p in train\_index], feats[train\_index,:],

times[train\_index], test\_size=0.1)

model = CNFNet(args)

optimizer = optim.Adam(model.parameters(), lr=args.lr, weight\_decay=args.weight\_decay)

print\_network(model)

#cri = nn.CrossEntropyLoss(weight=torch.Tensor([float((times==1).sum()/(times==0).sum()) ,1]))

cri = nn.CrossEntropyLoss()

Tr\_incides = np.arange(len(inc\_feat\_tr))

Tr\_incides = torch.LongTensor(Tr\_incides)

test\_index = torch.LongTensor(test\_index)

graph\_loader = DataLoader(GraphDataset(Tr\_incides[:int(len(Tr\_incides) / args.batch\_size) \* args.batch\_size]),

batch\_size=args.batch\_size, shuffle=True)

train\_loss = []

eval\_loss = []

train\_acc = []

val\_acc = []

os.environ['CUDA\_VISIBLE\_DEVICES'] = '0'

torch.backends.cudnn.benchmark = True

# Training Model

for epoch in range(args.epochs):

for step, ids in enumerate(graph\_loader):

t = time.time()

model.train()

output = model(itemgetter(\*ids)(inc\_feat\_tr), feat\_tr[ids])

loss\_train = cri(output, labels\_tr[ids])

optimizer.zero\_grad()

loss\_train.backward()

optimizer.step()

print('Epoch: {:02d}/{:04d}'.format(epoch, step + 1),

'loss\_train: {:.4f}'.format(loss\_train.item()),

'time: {:.4f}s'.format(time.time() - t), end='\n')

if step % 10 == 0:

model.eval()

#val\_ids = list(chunks(val\_num, args.batch\_size))[:-1]

output\_eval = [model(inc\_feat\_te, feat\_te)]

#output\_eval = [model(itemgetter(\*\_)(inc\_feat), feats[\_]) for \_ in val\_ids]

loss\_val = np.mean([cri(output\_eval[0][\_], labels\_te[\_]).item() for \_ in range(len(labels\_te))])

#loss\_val = np.mean([cri(output\_eval[\_], times[val\_ids[\_]]).item() for \_ in range(len(val\_ids))])

print("Eval loss: {}".format(loss\_val), end='\n')

train\_acc.append(Obtain\_Accuracy(output,labels\_tr[ids],model))

val\_acc.append(Obtain\_Accuracy(output\_eval[0],labels\_te,model))

train\_loss.append(loss\_train.item())

eval\_loss.append(loss\_val.item())

# print training info

plot\_metric(range(len(train\_loss)), train\_loss, eval\_loss, '{}\_train\_{}'.format('loss',i),

'{}\_eval\_{}'.format('loss',i))

plot\_metric(range(len(train\_acc)), train\_acc, val\_acc, '{}\_train\_{}'.format('Acc',i),

'{}\_eval\_{}'.format('Acc',i))

print("Optimization Finished!")

model.eval()

test\_ids = list(chunks(test\_index, args.batch\_size))

output\_test = []

output\_test = [model(itemgetter(\*\_)(inc\_feat), feats[\_]) for \_ in test\_ids]

loss\_val = np.mean([cri(output\_test[\_], times[test\_ids[\_]]).item() for \_ in range(len(test\_ids))])

print("Test loss: {}".format(loss\_val))

test\_acc = Obtain\_Accuracy(output\_test[0],times[test\_ids[0]],model)

test\_pred = Prediction(output\_test[0],model)

NPs.append(np.sum(np.asarray(times[test\_ids[0]].tolist())==1))

NNs.append(np.sum(np.asarray(times[test\_ids[0]].tolist())==0))

AC, SE, SP, F1, \_ = Evaluate\_CM(test\_pred.tolist(),times[test\_ids[0]].tolist())

ACs.append(AC)

SEs.append(SE)

SPs.append(SP)

F1s.append(F1)

print(test\_acc)

All\_test\_acc.append(test\_acc)

All\_test\_loss.append(loss\_val)

All\_test\_acc = np.asarray(All\_test\_acc)

Mean\_All\_test\_acc = np.mean(All\_test\_acc)

All\_test\_loss = np.asarray(All\_test\_loss)

Mean\_All\_test\_loss = np.mean(All\_test\_loss)

Mydic = {"All\_test\_acc" : All\_test\_acc, "Mean\_All\_test\_acc" : Mean\_All\_test\_acc,

"Mean\_All\_test\_loss":Mean\_All\_test\_loss, "All\_test\_loss" : All\_test\_loss,'ACs':np.asarray(ACs),

'SEs':np.asarray(SEs), 'SPs':np.asarray(SPs), 'F1s':np.asarray(F1s)}

savemat('results.mat',Mydic)

1. **Modules and libraries**

* **bench2cnf.py**

This module is used for obtaining CNF representation for the bench file of an IC.

import argparse

import itertools

import numpy as np

import copy

class wire:

def \_\_init\_\_(self, name, type, operands, logic\_value, logic\_level, prob0, prob1, absprob,

fanout, mainout, tag, index):

self.name = name

self.type = type

self.operands = operands

self.logic\_value = logic\_value

self.logic\_level = logic\_level

self.prob0 = prob0

self.prob1 = prob1

self.absprob = absprob

self.fanout = fanout

self.mainout = mainout

self.tag = tag

self.index = index

def findsubsets(S, m):

return set(itertools.combinations(S, m))

def sub\_lists(list1):

# store all the sublists

sublist = [[]]

# first loop

for i in range(len(list1) + 1):

# second loop

for j in range(i + 1, len(list1) + 1):

# slice the subarray

sub = list1[i:j]

sublist.append(sub)

return sublist

def wire\_print(wire\_in):

print("w\_name: ", wire\_in.name)

print("w\_type: ", wire\_in.type)

for i in range(0, len(wire\_in.operands)):

print("w\_opr", i, ": ", wire\_in.operands[i].name)

print("w\_valu: ", wire\_in.logic\_value)

print("w\_glev: ", wire\_in.logic\_level)

print("w\_prob0:", wire\_in.prob0)

print("w\_prob1:", wire\_in.prob1)

print("w\_abs\_prob:", wire\_in.absprob)

print("w\_fanout:", wire\_in.fanout)

print("w\_mainout:", wire\_in.mainout)

print("index:", wire\_in.index, "\n")

def wire\_fanin\_cone(wire\_in, cone\_size):

index\_traverse = 0

if wire\_in.type == "inp":

return []

else:

fanin\_cone = [wire\_in]

cone\_size = cone\_size - 1

temp = copy.deepcopy(fanin\_cone[index\_traverse].operands)

while cone\_size != 0:

added\_bef = 0

if len(temp) == 0:

index\_traverse = index\_traverse + 1

if index\_traverse >= len(fanin\_cone):

cone\_size = 0

else:

temp = copy.deepcopy(fanin\_cone[index\_traverse].operands)

elif temp[0].type == "inp":

temp.remove(temp[0])

else:

for i in range(0, len(fanin\_cone)):

if fanin\_cone[i].name == temp[0].name:

added\_bef = 1

if added\_bef == 0:

fanin\_cone.append(temp[0])

cone\_size = cone\_size - 1

temp.remove(temp[0])

return fanin\_cone

def get\_unique\_fanin\_cone(wire\_in):

return uniquify\_wire\_list(get\_fanin\_cone(wire\_in))

def get\_fanin\_cone(wire\_in):

if wire\_in.type == "inp":

return []

else:

fanin\_cone = []

fanin\_cone.append(wire\_in)

for i in range(len(wire\_in.operands)):

fanin\_cone += get\_fanin\_cone(wire\_in.operands[i])

return fanin\_cone

def get\_fanin\_cone2(wire\_in, fanin\_cone):

if wire\_in.type != "inp":

fanin\_cone.add(wire\_in)

for i in range(len(wire\_in.operands)):

get\_fanin\_cone2(wire\_in.operands[i], fanin\_cone)

def uniquify\_wire\_list(wire\_list):

seen = set()

unique = []

for obj in wire\_list:

if obj.name not in seen:

unique.append(obj)

seen.add(obj.name)

return unique

def wire\_dep(benchmark\_address, args):

inputs = []

outputs = []

wires = []

input\_array = []

temp = []

bench\_file = open(benchmark\_address)

index = 1

for line in bench\_file:

if "INPUT" in line:

inputs.append(line[line.find("(") + 1:line.find(")")])

input\_array.append(line[line.find("(") + 1:line.find(")")])

wires.append(wire(line[line.find("(") + 1:line.find(")")], "inp", [], "1", 0, 0.5, 0.5, 0, 0, 0, 0, index))

input\_array = []

index += 1

elif "OUTPUT" in line:

out\_name = line[line.find("(") + 1:line.find(")")]

outputs.append(out\_name)

elif " = " in line:

gate\_out = line[0: line.find(" =")]

gate\_type = line[line.find("= ") + 2: line.find("(")]

gate\_list\_inputs = line[line.find("(") + 1:line.find(")")]

gate\_oprs = gate\_list\_inputs.split(", ")

for i in range(0, len(gate\_oprs)):

found = False

for j in range(0, len(wires)):

if wires[j].name == gate\_oprs[i]:

found = True

temp.append(wires[j])

break

if not found:

print("ERROR")

print(gate\_out, gate\_oprs[i])

exit()

max\_level = 0

for i in range(0, len(temp)):

if temp[i].logic\_level > max\_level:

max\_level = temp[i].logic\_level

temp\_prob0 = 0.25

temp\_prob1 = 0.25

for i in range(0, len(temp)):

if len(temp) == 1:

if gate\_type == "NOT" or gate\_type == "not":

temp\_prob = temp[0].prob0

temp\_prob0 = temp[0].prob1

temp\_prob1 = temp\_prob

elif gate\_type == "BUFF" or gate\_type == "buff":

temp\_prob0 = temp[0].prob0

temp\_prob1 = temp[0].prob1

else:

if gate\_type == "NAND" or gate\_type == "nand":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob0 + temp\_prob1 \* temp[i].prob0 + temp\_prob0 \* temp[

i].prob1

temp\_prob1 = temp\_prob1 \* temp[i].prob1

elif gate\_type == "AND" or gate\_type == "and":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob0 + temp\_prob1 \* temp[i].prob0 + temp\_prob0 \* temp[

i].prob1

temp\_prob1 = temp\_prob1 \* temp[i].prob1

elif gate\_type == "NOR" or gate\_type == "nor":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob0

temp\_prob1 = 1 - temp\_prob0

elif gate\_type == "OR" or gate\_type == "or":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob0

temp\_prob1 = 1 - temp\_prob0

elif gate\_type == "XNOR" or gate\_type == "xnor":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob0 + temp\_prob1 \* temp[i].prob1

temp\_prob1 = 1 - temp\_prob0

elif gate\_type == "XOR" or gate\_type == "xor":

if i == 0:

temp\_prob0 = temp[i].prob0

temp\_prob1 = temp[i].prob1

else:

temp\_prob0 = temp\_prob0 \* temp[i].prob1 + temp\_prob1 \* temp[i].prob0

temp\_prob1 = 1 - temp\_prob0

if gate\_type == "NAND" or gate\_type == "nand":

temp\_prob = temp\_prob0

temp\_prob0 = temp\_prob1

temp\_prob1 = temp\_prob

if gate\_type == "NOR" or gate\_type == "nor":

temp\_prob = temp\_prob0

temp\_prob0 = temp\_prob1

temp\_prob1 = temp\_prob

wires.append(wire(gate\_out, gate\_type, temp, "1", max\_level + 1, temp\_prob0, temp\_prob1,

abs(temp\_prob0 - temp\_prob1), 0, 0, 0, index))

temp = []

index += 1

bench\_file.close()

# just check for correctness

for i in range(0, len(wires)):

for j in range(i + 1, len(wires)):

if wires[i].name == wires[j].name:

print(wires[i].name, wires[j].name)

print("ERROR in reading wires")

exit()

for i in range(0, len(wires)):

fanout\_temp = 0

for j in range(0, len(wires)):

if i != j:

for k in range(0, len(wires[j].operands)):

if wires[i].name == wires[j].operands[k].name:

fanout\_temp = fanout\_temp + 1

wires[i].fanout = fanout\_temp

return wires

def simple\_read\_bench(benchmark\_address):

wires = []

temp = []

bench\_file = open(benchmark\_address)

index = 1

for line in bench\_file:

if "#" in line:

continue

elif "INPUT" in line:

wires.append(wire(line[line.find("(") + 1:line.find(")")], "inp", [], "1", 0, 0, 0, 0, 0, 0, 0, index))

elif "OUTPUT" in line:

wires.append(wire(line[line.find("(") + 1:line.find(")")], "out", [], "1", 0, 0, 0, 0, 0, 0, 0, index))

elif " = " in line:

gate\_out = line[0: line.find(" =")]

gate\_type = line[line.find("= ") + 2: line.find("(")].lower()

gate\_list\_inputs = line[line.find("(") + 1:line.find(")")]

gate\_oprs = gate\_list\_inputs.split(",")

gate\_oprs = [x.strip(' ') for x in gate\_oprs]

for i in range(0, len(gate\_oprs)):

found = False

for j in range(0, len(wires)):

if wires[j].name == gate\_oprs[i]:

found = True

temp.append(wires[j])

break

if not found:

# print gate\_out, gate\_oprs[i]

temp.append(wire(gate\_oprs[i], "dummy", [], "1", 0, 0, 0, 0, 0, 0, 0, 0))

wires.append(wire(gate\_out, gate\_type, temp, "1", 0, 0, 0, 0, 0, 0, 0, index))

else:

continue

temp = []

index += 1

bench\_file.close()

for i in range(0, len(wires)):

for j in range(0, len(wires[i].operands)):

if wires[i].operands[j].type == "dummy":

found = False

for k in range(0, len(wires)):

if wires[k].name == wires[i].operands[j].name:

found = True

wires[i].operands[j] = wires[k]

break

if not found:

print(wires[i].operands[j].name)

print("ERROR1 in read\_circuit()")

exit()

# just to be sure!

for i in range(len(wires)):

if wires[i].name != wires[wires[i].index - 1].name:

print(wires[i].name)

print("ERROR2 in read\_circuit()")

exit()

return wires

def tseytin\_t(wires\_in):

cnf\_clause\_count = 0

cnf\_clause\_list = ""

for i in range(len(wires\_in)):

if wires\_in[i].type == "NOT" or wires\_in[i].type == "not":

cnf\_clause\_list += str(wires\_in[i].index) + " " + str(wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_list += "-" + str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_count += 2

elif wires\_in[i].type == "BUFF" or wires\_in[i].type == "buff":

cnf\_clause\_list += str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_list += "-" + str(wires\_in[i].index) + str(wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_count += 2

elif wires\_in[i].type == "AND" or wires\_in[i].type == "and":

cnf\_clause\_list += str(wires\_in[i].index)

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += " -" + str(wires\_in[i].operands[j].index)

cnf\_clause\_list += " 0\n"

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += "-" + str(wires\_in[i].index) + " " + str(wires\_in[i].operands[j].index) + " 0\n"

cnf\_clause\_count += len(wires\_in[i].operands) + 1

elif wires\_in[i].type == "NAND" or wires\_in[i].type == "nand":

cnf\_clause\_list += "-" + str(wires\_in[i].index)

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += " -" + str(wires\_in[i].operands[j].index)

cnf\_clause\_list += " 0\n"

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += str(wires\_in[i].index) + " " + str(wires\_in[i].operands[j].index) + " 0\n"

cnf\_clause\_count += len(wires\_in[i].operands) + 1

elif wires\_in[i].type == "OR" or wires\_in[i].type == "or":

cnf\_clause\_list += "-" + str(wires\_in[i].index)

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += " " + str(wires\_in[i].operands[j].index)

cnf\_clause\_list += " 0\n"

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[j].index) + " 0\n"

cnf\_clause\_count += len(wires\_in[i].operands) + 1

elif wires\_in[i].type == "NOR" or wires\_in[i].type == "nor":

cnf\_clause\_list += str(wires\_in[i].index)

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += " " + str(wires\_in[i].operands[j].index)

cnf\_clause\_list += " 0\n"

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += "-" + str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[j].index) + " 0\n"

cnf\_clause\_count += len(wires\_in[i].operands) + 1

elif wires\_in[i].type == "XOR" or wires\_in[i].type == "xor":

operand\_subsets = sub\_lists(wires\_in[i].operands)

for j in range(0, len(operand\_subsets)):

if len(operand\_subsets[j]) % 2 == 0:

cnf\_clause\_list += "-" + str(wires\_in[i].index)

for k in range(0, len(wires\_in[i].operands)):

if wires\_in[i].operands[k] not in operand\_subsets[j]:

cnf\_clause\_list += " " + str(wires\_in[i].operands[k].index)

else:

cnf\_clause\_list += " -" + str(wires\_in[i].operands[k].index)

cnf\_clause\_list += " 0\n"

cnf\_clause\_count += 1

elif len(operand\_subsets[j]) % 2 == 1 and len(operand\_subsets[j]) != 0:

cnf\_clause\_list += str(wires\_in[i].index)

for k in range(0, len(wires\_in[i].operands)):

if wires\_in[i].operands[k] not in operand\_subsets[j]:

cnf\_clause\_list += " " + str(wires\_in[i].operands[k].index)

else:

cnf\_clause\_list += " -" + str(wires\_in[i].operands[k].index)

cnf\_clause\_list += " 0\n"

cnf\_clause\_count += 1

elif wires\_in[i].type == "XNOR" or wires\_in[i].type == "xnor":

operand\_subsets = sub\_lists(wires\_in[i].operands)

for j in range(0, len(operand\_subsets)):

if len(operand\_subsets[j]) % 2 == 1 and len(operand\_subsets[j]) != 0:

cnf\_clause\_list += "-" + str(wires\_in[i].index)

for k in range(0, len(wires\_in[i].operands)):

if wires\_in[i].operands[k] not in operand\_subsets[j]:

cnf\_clause\_list += " " + str(wires\_in[i].operands[k].index)

else:

cnf\_clause\_list += " -" + str(wires\_in[i].operands[k].index)

cnf\_clause\_list += " 0\n"

cnf\_clause\_count += 1

elif len(operand\_subsets[j]) % 2 == 0 and len(operand\_subsets[j]) != 0:

cnf\_clause\_list += str(wires\_in[i].index)

for k in range(0, len(wires\_in[i].operands)):

if wires\_in[i].operands[k] not in operand\_subsets[j]:

cnf\_clause\_list += " " + str(wires\_in[i].operands[k].index)

else:

cnf\_clause\_list += " -" + str(wires\_in[i].operands[k].index)

cnf\_clause\_list += " 0\n"

cnf\_clause\_count += 1

cnf\_clause\_list += str(wires\_in[i].index)

for j in range(0, len(wires\_in[i].operands)):

cnf\_clause\_list += " " + str(wires\_in[i].operands[j].index)

cnf\_clause\_list += " 0\n"

cnf\_clause\_count += 1

elif wires\_in[i].type == "MUX" or wires\_in[i].type == "mux":

cnf\_clause\_list += str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[1].index) + " " + str(

wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_list += str(wires\_in[i].index) + " -" + str(wires\_in[i].operands[2].index) + " -" + str(

wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_list += "-" + str(wires\_in[i].index) + " " + str(wires\_in[i].operands[1].index) + " " + str(

wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_list += "-" + str(wires\_in[i].index) + " " + str(wires\_in[i].operands[2].index) + " -" + str(

wires\_in[i].operands[0].index) + " 0\n"

cnf\_clause\_count += 4

return cnf\_clause\_count, cnf\_clause\_list

def cnf\_to\_matrix(cnf\_str):

"""Convert cnf file to bipartite graph matrix (num\_clauses,num\_clauses)

Args:

fpath: The path to the cnf file.

The first two lines of the file are notes.

The clauses start from the 3rd line.

Returns:

A numpy 2-dimensional array.

The rows are for clauses.

The columns are for literals.

example:

array([[ 0, 0, 0, 0, -1, 1],

[ 0, 0, 0, 1, 0, -1],

[ 0, 0, 0, 0, 1, -1]])

The example denotes a cnf file with 3 clauses and 6 parameters.

0 denotes that the literal in this column didn't show up in the

cluase of this row.

1 denotes the positive literal exists in this clause.

-1 denotes the negative literal exists in this clause.

"""

# lines = open(fpath).readlines()[2:]

lines = cnf\_str.split("\n")

literals = {}

clauses = []

for l in lines:

for ele in l.strip().split(" ")[:-1]:

literals[abs(int(ele))] = 0

num\_literals = len(literals)

num\_clauses = len(lines)

literals\_list = list(literals)

l\_to\_pos = {}

pos\_init = 0

for l in literals\_list:

l\_to\_pos[l] = pos\_init

pos\_init += 1

bi\_g = np.zeros((num\_clauses, num\_literals), dtype=np.int)

for i in range(num\_clauses):

for ele in lines[i].strip().split(" ")[:-1]:

literal = int(ele)

pos = l\_to\_pos[abs(literal)]

if literal > 0:

bi\_g[i, pos] = 1

else:

bi\_g[i, pos] = -1

return bi\_g

if \_\_name\_\_ == "\_\_main\_\_":

parser = argparse.ArgumentParser(description='ISCAS-85 bench to CNF Converter v1.0 (c) Hadi Mardani Kamali.')

parser.add\_argument("-b", action="store", required=True, type=str, help="benchmark path")

parser.add\_argument("-c", action="store", required=True, type=str, help="cnf path")

args = parser.parse\_args()

bench\_address = args.b

cnf\_address = args.c

wires = simple\_read\_bench(bench\_address)

cnf\_variable\_counts = len(wires)

cnf\_clause\_count, cnf\_content = tseytin\_t(wires)

cnf\_file\_content = "Circuit-CNF generated by BENCH2CNFv1.0 (c) Hadi Mardani Kamali\n"

cnf\_file\_content += "p cnf " + str(cnf\_variable\_counts) + " " + str(cnf\_clause\_count) + "\n" + cnf\_content

cnf\_file = open(cnf\_address, 'w')

cnf\_file.write(cnf\_file\_content)

cnf\_file.close()

print("cnf has been successfully written in", cnf\_address)

* **util.py**

This module has different function, which the most important of them are cnf\_to\_matrix, read\_ic, and read\_ic\_10fold. The cnf\_to\_matrix computes the matrix of CNF representation. The read\_ic is used to load the features and SAT-runtimes of each IC from the saved pickle files. The read\_ic\_10fold perform the same work as the read\_ic function based on 10-fold cross-validation technique.

import \_pickle as pk

import torch

from pylab import \*

from torch.utils.data import Dataset

def cnf\_to\_matrix(cnf\_str):

"""Convert cnf file to bipartite graph matrix (num\_clauses,num\_clauses)

Args:

fpath: The path to the cnf file.

The first two lines of the file are notes.

The clauses start from the 3rd line.

Returns:

A numpy 2-dimensional array.

The rows are for clauses.

The columns are for literals.

example:

array([[ 0, 0, 0, 0, -1, 1],

[ 0, 0, 0, 1, 0, -1],

[ 0, 0, 0, 0, 1, -1]])

The example denotes a cnf file with 3 clauses and 6 parameters.

0 denotes that the literal in this column didn't show up in the

cluase of this row.

1 denotes the positive literal exists in this clause.

-1 denotes the negative literal exists in this clause.

"""

# lines = open(fpath).readlines()[2:]

lines = cnf\_str.split("\n")

literals = {}

clauses = []

for l in lines:

for ele in l.strip().split(" ")[:-1]:

literals[abs(int(ele))] = 0

num\_literals = len(literals)

num\_clauses = len(lines)

literals\_list = list(literals)

l\_to\_pos = {}

pos\_init = 0

for l in literals\_list:

l\_to\_pos[l] = pos\_init

pos\_init += 1

bi\_g = np.zeros((num\_clauses, num\_literals), dtype=np.int)

for i in range(num\_clauses):

for ele in lines[i].strip().split(" ")[:-1]:

literal = int(ele)

pos = l\_to\_pos[abs(literal)]

if literal > 0:

bi\_g[i, pos] = 1

else:

bi\_g[i, pos] = -1

return bi\_g

def read\_ic(CircuitNames,q,m2,

Torch\_transform=True,

SF=[]):

"""read preprocessed data files"""

""" q is the start index for vector features """

""" m1 and m2 are the thresholds """

def Map\_labels(t):

if 0<=t<=m2:

return 1 #Label 1 refers to SAT-vulnerable samples

else:

return 0 #Label 0 refers to SAT-resilient samples

root\_dir = '{}\_{}.pk'

#times = [pk.load(open(root\_dir.format(c, 'Y'), 'rb')) for c in CircuitNames]

times = []

input = []

for c in CircuitNames:

T = pk.load(open(root\_dir.format(c, 'Y'), 'rb'))

times.extend(T)

X = pk.load(open(root\_dir.format(c, 'X'), 'rb'))

input.extend(X)

times = list(map(Map\_labels,times))

times = np.array(times)

input = np.array(input)

inc\_feat = [\_[q:] for \_ in input]

feat\_list = [\_[:q] for \_ in input]

#feat = [Each\_data[SF].squeeze(axis=0) for Each\_data in feat\_list]

if SF==[]:

feat = feat\_list

else:

feat = [Each\_data[SF].squeeze(axis=0) for Each\_data in feat\_list]

# feat = np.asarray(feat)

# feat = feat[:,SF]

# feat = feat.squeeze(axis=1)

print('Data size: {}'.format(times.size))

train\_rate = 0.8 # 80% of data is divided into the training set

val\_rate = 0.9 # 10% of data is divided into the validation set

test\_rate = 1 # 10% of data is divided into the testing set

DATA\_NUM = len(times)

data\_ind = np.arange(DATA\_NUM)

np.random.seed(8)

np.random.shuffle(data\_ind)

train\_num = sorted(data\_ind[range(int(DATA\_NUM \* train\_rate))])

val\_num = sorted(data\_ind[range(int(DATA\_NUM \* train\_rate), int(DATA\_NUM \* val\_rate))])

test\_num = sorted(data\_ind[range(int(DATA\_NUM \* val\_rate), int(DATA\_NUM \* test\_rate))])

if Torch\_transform == True:

train\_num = torch.LongTensor(train\_num)

val\_num = torch.LongTensor(val\_num)

test\_num = torch.LongTensor(test\_num)

#times = torch.FloatTensor(np.log1p(times))

times = torch.LongTensor(times)

feat = torch.FloatTensor(feat)

else:

pass

return inc\_feat, feat, times, train\_num, val\_num, test\_num

def read\_ic\_10fold(CircuitNames,q,m2,

Torch\_transform=True,

SF=[]):

"""read preprocessed data files"""

""" q is the start index for vector features """

""" m1 and m2 are the thresholds """

def Map\_labels(t):

if 0<=t<=m2:

return 1 #Label 1 refers to SAT-vulnerable samples

else:

return 0 #Label 0 refers to SAT-resilient samples

root\_dir = '{}\_{}.pk'

#times = [pk.load(open(root\_dir.format(c, 'Y'), 'rb')) for c in CircuitNames]

times = []

input = []

for c in CircuitNames:

T = pk.load(open(root\_dir.format(c, 'Y'), 'rb'))

times.extend(T)

X = pk.load(open(root\_dir.format(c, 'X'), 'rb'))

input.extend(X)

times = list(map(Map\_labels,times))

times = np.array(times)

input = np.array(input)

inc\_feat = [\_[q:] for \_ in input]

feat\_list = [\_[:q] for \_ in input]

#feat = [Each\_data[SF].squeeze(axis=0) for Each\_data in feat\_list]

if SF==[]:

feat = feat\_list

else:

feat = [Each\_data[SF].squeeze(axis=0) for Each\_data in feat\_list]

# feat = np.asarray(feat)

# feat = feat[:,SF]

# feat = feat.squeeze(axis=1)

# print('Data size: {}'.format(times.size))

# train\_rate = 0.8 # 80% of data is divided into the training set

# val\_rate = 0.9 # 10% of data is divided into the validation set

# test\_rate = 1 # 10% of data is divided into the testing set

# DATA\_NUM = len(times)

# data\_ind = np.arange(DATA\_NUM)

# np.random.seed(8)

# np.random.shuffle(data\_ind)

# train\_num = sorted(data\_ind[range(int(DATA\_NUM \* train\_rate))])

# val\_num = sorted(data\_ind[range(int(DATA\_NUM \* train\_rate), int(DATA\_NUM \* val\_rate))])

# test\_num = sorted(data\_ind[range(int(DATA\_NUM \* val\_rate), int(DATA\_NUM \* test\_rate))])

if Torch\_transform == True:

#train\_num = torch.LongTensor(train\_num)

#val\_num = torch.LongTensor(val\_num)

#test\_num = torch.LongTensor(test\_num)

#times = torch.FloatTensor(np.log1p(times))

times = torch.LongTensor(times)

feat = torch.FloatTensor(feat)

else:

pass

return inc\_feat, feat, times,

def print\_network(net):

"""print brief structure of neural networks"""

num\_params = 0

for param in net.parameters():

num\_params += param.numel()

print(net)

print('Total number of parameters: %d' % num\_params)

def plot\_metric(x, y, z, yc='train', zc='eval'):

"""plot result image and save in local files"""

matplotlib.style.use('seaborn')

plt.figure(figsize=(12, 6))

plt.xticks(fontsize=20)

plt.yticks(fontsize=20)

# plt.scatter(x, y)

# plot(x, y, markerfacecolor='salmon', markeredgewidth=1, markevery=slice(40, len(y), 70),

# linestyle=':', marker='o', color='crimson', linewidth=3, label='fit') # fit result

plot(x, y, color='crimson', linewidth=5, label=yc) # fit result

# plt.scatter(x, z)

# plot(x, y, markerfacecolor='salmon', markeredgewidth=1, markevery=slice(40, len(y), 70),

# linestyle=':', marker='o', color='crimson', linewidth=3, label='fit') # fit result

plot(x, z, color='blue', linewidth=5, label=zc) # fit result

plt.legend(loc="best", prop={'size': 20})

# plt.savefig('loss.png')

plt.savefig('{}-{}.png'.format(yc, zc))

plt.close()

class GraphDataset(Dataset):

"""wrap function for sampling training instances"""

def \_\_init\_\_(self, ids):

self.ids = ids

def \_\_getitem\_\_(self, index):

return self.ids[index]

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

return len(self.ids)

def chunks(l, n):

# For item i in a range that is a length of l,

for i in range(0, len(l), n):

# Create an index range for l of n items:

yield l[i:i + n]

def store\_model(model, name='test'):

torch.save(model, '{}'.format(name))

def restore\_model(path):

model = torch.load(path)

model.eval()

return model

* **model.py**

This module defines the main structure of CNF-NET model.

import torch

import torch.nn as nn

from layer import FC, IncidenceConvolution

class CNFNet(nn.Module):

def \_\_init\_\_(self, opt):

super(CNFNet, self).\_\_init\_\_()

self.opt = opt

# initialize CNFNet

self.energy\_kernel = IncidenceConvolution(opt)

self.fc = FC(opt)

def forward(self, inc, f):

# calculate energy kernel

y = self.energy\_kernel(inc)

# concatenate with CNF properties

z = torch.cat((f, y), 1)

# connect to fully-connected layers

z = self.fc(z)

return z

* **layer.py**

The layers and operations of the CNF-NET parts is defined in layer.py

import torch

import torch.nn as nn

import torch.sparse

from torch.nn.modules.module import Module

class IncidenceConvolution(Module):

"""Core operation of CNFNet"""

def \_\_init\_\_(self, opt):

super(IncidenceConvolution, self).\_\_init\_\_()

self.opt = opt

self.in\_features = 1

self.out\_features = opt.num\_feat

self.fc\_dim = opt.hidden

self.func1 = nn.Linear(1, self.fc\_dim)

self.func2 = nn.Linear(self.fc\_dim, self.fc\_dim)

self.func3 = nn.Linear(self.fc\_dim, self.fc\_dim)

self.func4 = nn.Linear(self.fc\_dim, 1)

def forward(self, inc\_m):

"""iterate instances and call energy function"""

out = []

for instance in inc\_m: # itearate on each matrix data

feat = []

for arr in instance:

feat.append(self.\_to\_kernel(arr))

out.append(torch.FloatTensor(feat))

return torch.stack(out)

def \_fc\_kernal(self, sig):

"""calculate the prediction given energy"""

return self.func4(self.func3(self.func2(self.func1(sig))))

def \_to\_kernel(self, arr):

"""calculate the energy representation of distribution"""

# arr[0] is a vector

l = torch.FloatTensor(arr[0])

# normalization

norm = l / torch.sum(l)

# FC and product

weight = torch.tensor([self.\_fc\_kernal(\_.unsqueeze(0)) for \_ in norm])

prod = torch.mul(weight, norm)

# sum

return torch.sum(prod)

def \_\_repr\_\_(self):

return self.\_\_class\_\_.\_\_name\_\_ + ' (' \

+ str(self.in\_features) + ' -> ' \

+ str(self.out\_features) + ')'

class FC(Module):

"""Fully connected layer for predicting runtime"""

def \_\_init\_\_(self, opt):

super(FC, self).\_\_init\_\_()

self.opt = opt

self.in\_features = opt.energy\_input\_dim

self.out\_features = 2

self.fc\_dim = opt.hidden

self.func1 = nn.Linear(self.in\_features, self.fc\_dim)

self.func2 = nn.Linear(self.fc\_dim, self.fc\_dim)

self.func3 = nn.Linear(self.fc\_dim, self.fc\_dim)

self.func4 = nn.Linear(self.fc\_dim, self.out\_features)

def forward(self, val):

return self.func4(self.func3(self.func2(self.func1(val))))

def \_\_repr\_\_(self):

return self.\_\_class\_\_.\_\_name\_\_ + ' (' \

+ str(self.in\_features) + ' -> ' \

+ str(self.out\_features) + ')'