In [153]:
```python
import sys
import numpy as np
import torch as torch
import torch.nn as nn
import torch.nn.functional as F
import pycosat
```

In [2]:
```python
#class(neurosat) Basic Idea layed out here:
    #def L_init, C_init - random vectors
    #def C_msg, L_msg, L_vote - MLP's
    #def L_update, C_update - LSTM, L_update takes in output from L_MSG and th
e current hidden state L_up_hidden
    #we need to store the output of running through the LTSM, first output is
 l at time t, and second is L_hid at time t

    #now we have our system set up: we have to define the message passing, set
T = 30 or something for example
    #def message_passing(self, M):
    #for i in range (1,30):
        #run the figure 8 system as we mentioned before. store The final L

    #now we just run L_final through the system L_vote which outputs a vector
 of size 2n
    #compute means, sigmoid, etc.


    #def training
    #just straightforward, compute the cross entropy loss and do backprop.





#class MLP
    #def MLP nn.module
    #def forward standard stuff, takes input L_init which is just a vector
```

The Model

In [157]:
```
class MLP(nn.Module):
    def __init__(self):
        self.input_dim = 128
        self.hidden_dim = [400,200]
        self.output_dim = 128
        model = nn.Sequential(nn.Linear(input_dim, hidden_dim[0]),nn.ReLU(),
                     nn.Linear(hidden_dim[0],hidden_dim[1]),
                      nn.ReLU(),
                     nn.Linear(hidden_dim[1],output_dim),nn.LogSoftmax())

   #alternatively can do
   # def forward(self, x):
      #  layer1 = torch.matmul(x,weights[0])... + Bias .. etc
      #





class NeuroSAT(nn.Module):
    def __init__(self, M,num_vars, num_clauses, is_sat):

        self.M = M #adjacency matrix
        self.num_vars = num_vars # number of variables in the problem input
        self.num_clauses = num_clauses #number of clauses in the problem imput
        self.is_sat = is_sat # True or False based on if problem is satisfiabl
e or not

        self.embed_d = 128 #number of dimensions for embedding

        self.L_init = torch.randn(1,embed_d)
        self.C_init = torch.randn(1,embed_d)

        self.L_msg = MLP(input_dim, hidden_dim)
        self.C_msg = MLP(input_dim, hidden_dim)

        self.L_update = nn.LSTM(input_dim, hidden_dim, n_layers) #i think pape
r uses n_layers = 3
        self.C_update = nn.LSTM(input_dim, hidden_dim, n_layers)

        #ok we have intialized all the models we need, now we need to chain th
em together


    def pass_messages(M):

        #initi_hidden_state = torch.randn(n_layers,batch_size, hidden_dim)
        #initi_cell_state = torch.randn(n_layers,batch_size, hidden_dim)
        #initi_states = (initi_hidden_state,initi_cell_state)

        inputmat = M # M is the input adjacency matrix, has dimensions 2N rows
C columns, N is #variables, C is # of clauses


        # initialize current and hidden states for the LSTM, all have 128 colu
mns
```

```
        L_current_state= tile(self.L_init,0,2*num_variables) #tile L_init embe
d_d number of times 2N x D matrix
        C_current_state = tile(self.C_init,0,num_clauses) #C x D matrix, so ou
r MLP should have input layer as D dimensions.

        L_hidden_state = torch.zeros(num_literals,embed_d)
        C_hidden_state= torch.zeros(num_clauses,embed_d)

        L_states = (L_current_state,L_hidden_state)
        C_states = (C_current_state,C_hidden_state)


        t = 0
        T =30

        while t < T: #number of times we message pass

            LC_pre_msgs = self.LC_msg(L_current_state)
            LC_msgs = torch.matmul(M.t(),LC_pre_msgs)

            _,C_states = self.C_update(LC_msgs, C_states)

            CL_pre_msgs = self.CL_msg(C_current_state)
            CL_msgs = torch.matmul(M,CL_pre_msgs)

            _,L_states = self.L_update(CL_msgs, L_states) #need to add some co
ncat thing here, to incorporate the flip operator

            C_hidden_state = C_states[1]
            C_current_state = C_states[0]

            L_hidden_state = L_states[1]
            L_current_state = L_states[0]

            t+=1


        self.final_lits = L_current_state
        self.final_clauses = C_current_state

    def compute_logits(self):

        self.all_votes = self.L_vote(self.final_lits) # n_lits x 1
        self.avg_vote = torch.mean(self.all_votes) # scalar

        self.logit = torch.log(self.avg_vote/(1-self.avg_vote))
        return self.logit


    #create a tile function to do the tiling they want in the paper
    def tile(a, dim, n_tile):

        init_dim = a.size(dim)
        repeat_idx = [1] * a.dim()
        repeat_idx[dim] = n_tile
        a = a.repeat(*(repeat_idx))
        order_index = torch.LongTensor(np.concatenate([init_dim * np.arange(n_
```

```
tile) + i for i in range(init_dim)]))
        return torch.index_select(a, dim, order_index)

def run_model(n_vars,n_clauses,is_sat,M, train_iters = 15):

    if is_sat == True:
        target = 1
    else:
        target = 0

    model = NeuroSAT(n_vars,n_clauses,is_sat,M)

    logit_input = model.compute_logits()
    #build optimizer
    optimizer = optim.Adam(model.parameters(), lr = .005)

    #still need to finish this part
    for epoch in range(0,train_iters):
        model.train()


        #loss function
        sig = nn.Sigmoid()
        loss = nn.BCEloss() #binary cross entropy
        output_loss = loss(sig(logit_input),target)

        output_loss.backward()

        optimizer.step()
```

Code for generating CNF's like they do in the paper: (Note, does not work properly, will need to fix at some point, but gives a rough idea of how to generate the data as intended)

In [152]:

```
def generate_k_iclause(n, k):
    lits = np.random.choice(n, size=min(n, k), replace=False)
    return [x + 1 if random.random() < 0.5 else -(x + 1) for x in lits]

def gen_iclause_pair():
    n = 10
    iclauses = []
    iclauses.append(generate_k_iclause(n, 3))
    while True:
        kbase = 1
        k = kbase+np.random.geometric(.4)
        iclause = generate_k_iclause(n, k)

        if(pycosat.solve(iclauses) == 'UNSAT'):
            break
        else:
            iclauses.append(iclause)

    iclause_unsat = iclause
    iclause_sat = [- iclause_unsat[0] ] + iclause_unsat[1:]
    return n, iclauses, iclause_unsat, iclause_sat
```

This generates data like this:

```
In [106]: _,cnf,_,_ = gen_iclause_pair()
```

```
In [107]: cnf
```

```
Out[107]: [[-6, 10, -4],
 [10, -2],
 [7, -2, 5],
 [3, 6, -4, 7],
 [-10, 2],
 [-4, 5, -2, -10, 7, 8, -6],
 [4, -7],
 [-8, -5, 10],
 [8, 7],
 [-8, 1, 4],
 [3, -1]]
```

```
In [145]: def to_adj(cnf,is_sat):
    is_sat = is_sat
    cnf = cnf
    num_clauses = len(cnf)

    max_val = 0
    for i in range(0,len(cnf)-1):

        if(np.max(cnf[i]) > max_val):
            max_val = np.max(cnf[i])

    num_vars = max_val #for the paper, this is fixed at n =40
    num_literals = 2*max_val

    #create blank adj matrix, we will fill in later
    adj = torch.zeros(num_literals,num_clauses)

    #puts in 1's in the adjaceny matrix
    for clause_id, clause in enumerate(cnf):
        for i in clause:
            val = i
            if(val>0):
                adj[2*val-2][clause_id] = 1
            else:
                val = -1*val
                adj[2*val-1][clause_id] = 1

    return(adj,num_clauses,num_vars,is_sat)
```

Hence, we take an input list of lists cnf, and convert it to an adjacency matrix like this

Input:

```
In [106]: _,cnf,_,_ = gen_iclause_pair()

In [107]: cnf

Out[107]: [[-6, 10, -4],
           [10, -2],
           [7, -2, 5],
           [3, 6, -4, 7],
           [-10, 2],
           [-4, 5, -2, -10, 7, 8, -6],
           [4, -7],
           [-8, -5, 10],
           [8, 7],
           [-8, 1, 4],
           [3, -1]]
```

Output:

```
to_adj(cnf,True)

(tensor([[0., 0., 0., 0., 0., 0., 0., 0., 0., 1., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.],
         [0., 0., 0., 0., 1., 0., 0., 0., 0., 0., 0.],
         [0., 1., 1., 0., 0., 1., 0., 0., 0., 0., 0.],
         [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 1.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [1., 0., 0., 0., 0., 0., 1., 0., 0., 1., 0.],
         [0., 0., 0., 1., 0., 1., 0., 0., 0., 0., 0.],
         [0., 0., 1., 0., 0., 1., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
         [0., 0., 0., 1., 0., 0., 0., 0., 0., 0., 0.],
         [1., 0., 0., 0., 0., 1., 0., 0., 0., 0., 0.],
         [0., 0., 1., 1., 0., 1., 0., 0., 1., 0., 0.],
         [0., 0., 0., 0., 0., 0., 1., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 1., 0., 0., 1., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 1., 0., 1., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.],
         [1., 1., 0., 0., 0., 0., 0., 1., 0., 0., 0.],
         [0., 0., 0., 0., 1., 1., 0., 0., 0., 0., 0.]]), 11, 10, True)
```

```
to_ad
(tens
```