LibraGPU Documentation

November 2021

Contents

1 deeppoly_gpu.py

Variable Description

Note: "*d_*" added as prefix to variable name represents device/GPU memory following the CUDA naming convention.

MNIL Stores the maximum number of neurons in any layer.

NOL Stores number of layers in the network.

NOI Stores number of distinct bounds we are considering parallelly.

affine Matrix to store affine functions of each neuron of each layer of the network in the form of coefficients and base with respect to the previous layer in the network.

- type: numpy(3-D, float32)
- size: (NOL+1, MNIL + 1, MNIL + 1)

relu Matrix to store *relu_status* for each neuron of each layer of the network obtained from each initial. *relu_status* is a tuple consisting of slope and y-coeff for less than equal in-equality and slope and y-coeff for greater than equal in-equation.

- type: numpy(4-D, float32)
- size: (NOI, NOL+1, MNIL + 1, 4)

var_index Dictionary to store row and column number for each variable/neuron to easily obtain its position in *affine* and *relu*

- type: Dict(Key:string, Value:(float32, float32))
- size: (NOI, No of neurons in network)

active_pattern Matrix to store *active_status* after forward analysis for each neuron of each layer of the network obtained from each initial.'0' signifies particular relu activation always gives 0.'1' represent that it always gives out the input.'2' signifies it can be activated or deactivated in the given range.

- type: numpy(3-D, float32)
- size: (NOI,NOL+1,MNIL + 1)

if_activation Matrix to store if there is a activation (RELU) for each neuron of each layer of the network.

- type: numpy(2-D, float32)

- size: (NOL+1, MNIL + 1)

ln_coeff_lte Stores the less than equal in-equation for all neurons in the current layer with respect to a certain previous layer obtained from each initial. It is updated until current layer is expressed in terms of layer-0

- type: numpy(3-D, float32)
- size: (NOI, MNIL + 1, MNIL + 1)

ln_coeff_gte Same as above but for greater than equal.

ineq_prev_lte Stores the less than equal in-equation for all the neurons of the previous layer with respect to neurons of the layer before for each initial.

- type: numpy(3-D, float32)
- size: (NOI, MNIL + 1, MNIL + 1)

ineq_prev_gte Same as above but for greater than equal.

l1_lte Stores the less than equal in-equation for all the neurons of the current layer with respect to the first layer obtained from each initial.

- type: numpy(3-D, float32)
- size: (NOI, MNIL + 1, MNIL + 1)

l1_gte Same as above but for greater than equal.

l1_lb Store lower bound of nodes in layer-0 for each initial.

- type: numpy(2-D, float32)
- size: (NOI, MNIL + 1)

l1_ub Same as above but for upper bound.

lbs Store lower bound for each neuron of current layer obtained from each initial.

- type: numpy(2-D, float32)
- size: (NOI, MNIL + 1)

ubs Same as above but for upper bound.

print_mode Determines the amount of detail to be printed during pre-ananlysis.

netGPU Tuple consisting of:

- *affine*
- *if_activation*
- *var_index*
- *inv_var_index*: *var_index* with keys and values switched
- *outNode*: set of neurons present in output layer
- *dims*: store the shape of each layer
- *l1_lb*
- *l1_ub*
- *sensitive*: input variable that is sensitive.
- *max_diff*: maximum difference between non sensitive bounds.

def *get_bounds_GPU*

Input : GPU memory: *l1_lte*, *l1_gte* *l1_lb* and *l1_ub*

Output: Tuple consisting of *lbs* and *ubs* in GPU memory

Purpose: Given bounds of input and a layer represented in form of layer-0, it return the bounds of its neurons

bound_helper GPU implemented helper using cuda.jit of numba.

Algorithm Perform the following steps:

- Allocate GPU memory to store *lbs* and *ubs* as zeros.
- Threads per block is set as the min of (64,16) and (NOI, MNIL). Blocks per grid set to cover all the elements.
- Parallelly consider each neuron in current layer and find its bound using in-equation with respect to layer-0. This is performed using *bound_helper*
- For lower bound consider lower than equal in-equation. If a coefficient is negative multiply with *l1_ub* of that variable, otherwise multiply with *l1_lb* of that variable and add to lower bound. Similarly, do the opposite for upper bound.

def ***relu_compute_GPU***

Input : GPU Memory: *lbs*, *ubs*, *relu_layer*, *if_activation* and *active_pattern*

Output: Update GPU Memory: *relu_layer* and *active_pattern*

Purpose: It computes the *relu_status* and *activation_pattern* for each neuron based on its bounds.

relu_compute_helper GPU implemented helper using cuda.jit of numba.

Algorithm Perform the following steps:

- Threads per block is set as the min of (64,16) and (NOI, MNIL). Blocks per grid set to cover all the elements.
- Parallelly consider each neuron in current layer and obtain the corresponding slope and y-coeff for DeepPoly's relu approximation while remembering which neurons were active. This step is performed using *relu_compute_helper*
- The coefficients are calculated according to the four following cases:
  - If for current neuron *if_activation* is 0 then *relu_status* is set as $(1, 0, 1, 0)$ and *activation_status* as 2. This setting lead to $y_{relu} = y$ which is same as not having an activation.
  - If upper bound for the neuron (before relu) is less than zero. Then *relu_status* is set as $(0, 0, 0, 0)$ and *activation_status* is 0. Setting slope as 0 and *y_coeff* as 0 is same as forcing an equation of form $y_{relu} = 0$
  - If lower bound for the neuron (before relu) is greater than zero. Then *relu_status* is set as $(1, 0, 1, 0)$ and *activation_status* is 1. Setting slope as 1 and *y_coeff* as 1 is same as forcing an equation of form $y_{relu} = y$
  - Otherwise set *activation_status* as 2 then check which of the 2 deep poly approximation is stricter. If *(1,0,slope,y_coeff)* is stricter then its set as *relu_status*. Otherwise, *(0,0,slope,y_coeff)*. Where slope is obtained as ub/(ub - lb) and *y_coeff* as ub*lb/(lb-ub) for current neuron

  Values of *relu_layer* and *active_pattern* in GPU memory are updated and need not be returned def ***back_propagate_GPU***

Input : *affine*, *relu*, *layer*, *if_activation*, *active_pattern*

Output: Tuple consisting of *l1_lte* and *l1_gte*

Purpose: It represents a given layer in form of layer-0 by back-substituting

back_affine_helper GPU implemented helper for back substituting affine layers using cuda.jit of numba.

back_affine_GPU Uses *back_affine_helper* for back substituting affine layers.

back_relu_coeff_helper GPU implemented helper for back substituting coefficient terms of relu layers using cuda.jit of numba.

back_relu_base_helper GPU implemented helper for back substituting base terms of relu layers using cuda.jit of numba.

back_relu_GPU Uses *back_relu_coeff_helper* and *back_relu_base_helper* for back substituting relu layers.

Algorithm Perform the following steps:

- Allocate GPU memory to store copies(to be modified) of affine equations for current layer in *ln_coeff_gte* and *ln_coeff_lte*. These make sure the *affine* matrix remains intact
- Run a loop from current layer to the first and for each back substitute the relu activation and affine layer.
- First back substitute the relu if an activation was present for the previous layer. If the affine matrix defines the current matrix as $x = a * y + b * z + c$. And if y,z is produced using from slope s_y,s_z and y-coeff c_y,c_z on p and q by the reply layer $[y = s_y * p + c_y,\ z = s_z * q + c_z]$. Then using *back_relu_GPU* we obtain $x = a * s_y * p + b * s_z * q + (a * c_y + b * c_z + c)$ using the following steps
  - Obtain the base term for each neuron using each of its terms parallelly and y-coeff of the particular neuron using *back_relu_base_helper*. This is done first so can we directly modify the coefficients in the next step.
  - Obtain the updated coefficient terms for each neuron using each of its older coefficient terms and slope for relu of the particular neuron using *back_relu_coeff_helper*

- Next back substitute the affine layer is performed by running a loop to consider i^{th} term in all equations.
  - This axis is chosen as the outermost loop as each term in in equations of neuron are modified several times based on neurons of previous layer thus would require synchronization if were to be performed parallelly.
  - Let $x_7 <= x_5 + x_6$ and $x_7 >= x_5 + x_6$ be the initial in-equations ; $x_5 = x_3 + x_4$ and $x_6 = x_3 + x_4$ are the equations of layer-1 with respect to activation of layer-2. While obtaining in-equation for x_7 in terms of x_3 and x_4. Both x_5 and x_6 have a x_3 term to be considered for the same place at in the array for neuron x_7. Thus better to consider x_5 and x_6 sequentially. As mentioned earlier it can also be managed with synchronization but that would slow the process down.
  - Considering a particular term in previous step say x_5 and parallelly update each term of *l1_gte* and *l1_lte* using lte and gte in in-equation of x_5. In other words, parallelly consider each node in current layer and each term in its in-equation with respect to layer-0, and update those terms using coefficient of the same from in-equations of x_5. This is performed using *back_affine_helper*

  Once we have obtained the current equation in the form of layer-0 we can use *relu_compute_GPU* to update the *relu_state* and *active_pattern* and can continue to the next layer of relu.

def *active_convert*

Input : *active_pattern*

Output: list of tuples: activated and deactivated

Purpose: Convert *active_pattern* to a form compatible with further analysis steps

Algorithm Perform the following steps for each initial:

- Create empty lists for activated and deactivated
- Loop over each element of *active_pattern* while keeping a count.

- If current element is 0 then add current count and add the variable name to deactivated set
- If current element is 1 then add to activated set.
- Convert the lists to tuples and add to their respective list.

def *oneOutput*

Input : GPU memory: *affine*, $relu$, *l1_lb*, *l1_ub*; CPU memory: $if\_activation$, $outNodes$, inv_var_index, *sensitive* and *print_mode*

Output: list of *outcome*(s)

Purpose: Convert *outcome* to a form compatible with further analysis steps

Algorithm Perform the following steps for each initial.

- Consider each output node sequentially(say, out1)
- Generate a new layer with MNIL-1 neurons where each represent a neuron minus out1 excluding itself
- Run back substitution on this layer and obtain its lbs and ubs.
- If all the lbs obtained from this layer are greater than zero then outcome is out1
- Otherwise consider the next neuron, if none satisfy the condition, outcome is None.
- Append outcome from each initial to a list to be returned.
- if *print_mode* is 1 or 2 then print the outcome obtained from each initial.

def *noPrintCondense*

Input : GPU memory: *affine*, $relu$, $active\_pattern$, *l1_lb*, $l1_ub$, $if\_activation$ CPU memory: $if\_activation$

Purpose: Calculate for each of the layer sequentially to update $relu$ and $active\_pattern$. It is identified by *print_mode* 3

Algorithm Perform the following steps for each layer starting from 1 :

- Express each neuron in current layer in form of the first layer using $back\_propagate\_GPU$.

- If current layer has an activation then calculate lower bound and upper bound for each neuron using equation obtained in previous step. And use that to modify *relu* and *active_pattern* using *relu_compute_GPU*.

def *miniPrintCondense*

Similar to *noPrintCondense* while additionally printing little detail for each initial. Which include lower and upper bound for each neuron. It is identified by *print_mode* 2

def *detailedPrintCondense*

Similar to *noPrintCondense* while additionally printing detail for the first initial in each batch. It is identified by *print_mode* 1. The additional print represent:

- *if_activation* value and affine equation
- in-equations and bounds after back-substitution to layer 0
- *slope*, *y-coeff* and bounds obtained for Relu.

def *analyze*

Input : *netGPU*, *l1_lbL*, *l1_ubL*, *percent*, *L*, *print_mode*

Output: list of *activated*, *deactivated*, *outcome*, *l1_lb_list*, *l1_ub_list* and *percent*

Purpose: Split the initials and obtain results [*activated*, *deactivated*, *outcome*]

Algorithm Perform the following steps:

- Split initials in *l1_lbL*, *l1_ubL* using *commons.splitInitial()* to obtain *l1_lb_list* and *l1_ub_list*
- create lists to store result for the batches of initials.
- For each batch of initials perform the following:
  - Initialize cupy matrices directly for *relu* and *active_pattern*
  - Call any one of the three *PrintCondense* based on *print_mode*
  - Call *oneOutput* to obtain Outcome
  - Convert *activate_pattern* into the required sets using *active_convert*
  - Append the results of the current batch

- Update percentage by dividing by total no of initials formed from each input initial.
- Return the results

2 commons.py

def *convertInitial*

Input : *bounds*, *var_index*, *sensitive*

Output: *l1_lb*, *l1_ub*, $sensitiveNum$, *maxDiff*

Purpose: Obtain initials bounds as numpy arrays.

Algorithm Perform the following steps:

- Create *l1_lb*,*l1_ub* with zeros as per the number of variables in bounds
- loop over the bounds
  - If the current bound is for the sensitive variable. Store its index in $sensitiveNum$
  - If not sensitive and $ub - lb >$ *maxDiff* then update *maxDiff*. *maxDiff* later acts as *L_start*
  - Make an entry in *var_index* for the variable.
  - Add bound for the variable into *l1_lb*,*l1_ub*. The position is given using the loop counter.
- return the values

def *convertBound*

Input : *lbL*, *ubL*, *inv_var_index*, *sensitive*

Purpose: Convert the lists into a dictionary with variable names as keys. This provides a easier to understand print format while debugging.

Algorithm Perform the following steps:

- For each element in list *lbL*, *ubL*. And an entry in the dictionary with key given by variable name and value with a tuple having the lower and upper bound.

def ***splitInitial***

Input : $l1\_lbL$, $l1\_ubL$, $sensitiveNum$, L

Output: $l1\_lb$, $l1\_ub$

Purpose: Create new initials from a list of initials by splitting range of each variable into two.

Algorithm Perform the following steps:

- For each initial consider each variable one by one.
  - If variable is sensitive or if lower and upper bound are same or $ub - lb <= L * 2$ then don't split, keep [lb,ub]. For the last case $ub - lb <= L * 2$ means the difference of middle point from ub or lb, which is half of $ub - lb$, will be less than L
  - Otherwise, obtain two ranges [lb,mid],[mid,ub]
- Consider the Cartesian product of the ranges in previous step to obtain the new initials. This is done for each initial in the list of Initial given.
- Convert the above list of initials into list of numpy array each having a batch of a certain maximum number (eg: 2^{16}) of initials

def ***getNetShape***

Input : *layers*

Output: NOL, MNIL

Algorithm Perform the following steps:

- NOL is obtained as the length of *layers*
- Loop each layer and find the maximum no of coefficient among them and store in MNIL

def ***fillInput***

Input :*layers*, *activation*, *affine*, $dims$, $if\_activation$, var_index, $MNIL$

Purpose: Fill values using *layers* and *activation* into the other inputs

Algorithm Perform the following steps for the variable and its equation of each neuron in each layer of $layers$:

- Convert the equations into a array of coefficient by using *var_index* to obtain position in the array.
- Add the array above into an empty location in *affine* and store the location in *var_index* with variable name as key.

For each variable in *activations* set its corresponding position in $if\_activation$ as 1.

def *createNetworkGPU*

Input : *layers*, *bounds*, *activations*, *sensitive*, *outputs*

Output: tuple *netGPU*

Algorithm Perform the following steps:

- Obtain values for NOL and MNIL using $getNetShape$
- Internalize numpy matrices for *affine*, $if\_activation$, *dims*. And dictionary *var_index*
- Using *bounds* set *var_index* which have first layer elements as keys. Also fill *l1_lb* and *l1_ub* for each bound in *bounds* using $convertInitial$
- Fill *affine*, $if\_activation$, *dims* and *var_index* using $fillInput$
- Invert dictionary *var_index* and store as *inv_var_index*
- Store neuron index of outputs in $outNodes$ obtained using *outputs* and var_index
- Transfer *affine*, $if\_activation$ to GPU memory.
- Return the tuple

3 preanalysis_gpu.py

Variable Description

feasible Dictionary to store details regarding ranges that are fit for analysis.

- key: (activated neuron set, deactivated neuron set)
- value: list of (lower-bounds, upper-bounds, percent, outcome)

unfeasible List to store details regarding ranges that are unfit for analysis as too many disjunctions.

- element: (lower-bounds, upper-bounds, percent, outcome)

unbiased List to store details regarding ranges that can be classified as unbiased by the pre-analysis itself

- element: (lower-bounds, upper-bounds, percent, outcome)

def *iterPreanalysis*

Input : *l1_lbL*, *l1_ubL*, *netGPU*, L, U, *L_min*, *sensitive*, *percent*, *domains*

Purpose: Run pre-analysis for a set of initials which are split at first. Update *unbiased*, *feasible* and *unfeasible* and obtain a list of initials than can be further split for pre-analysis.

Algorithm Perform the following steps:

- Auto-tune value of U by increasing it by 1. If $L <= L\_min$ then set L as L_min and U as U_max. When $L = L\_min$ it's the last recursive call thus U is set as U_max. $L < L\_min$ occurs when L_start in *preanalysis* is same as L_min and first call of *iterPreanalysis* starts with $L\_start/2$ thus it needs to be reset to L_min. $L < L\_min$ occurs for the edge case in which the input bounds is cannot be split at all.
- initialize lists *l1_lbN* and *l1_ubN* [postfix 'N' represents new]
- Run a preanalysis specified by *l1_lbL*, *l1_ubL*, *netGPU* and *domains* to obtain *activatedL2*, *deactivatedL2*, *outcomeL2*, *lbL2*,*ubL2* [postfix 'L2' represents list of list]. Consider each element of this list of lists as follows

- If *outcome* is not a none then for the particular range the same value is always returned thus the range is unbiased
- Otherwise if the no of neuron for which activation status is unknown(2) is less than equal U then append the range to the list in dictionary feasible for key (activated,deactivated)
- Otherwise if L/2 is greater than equal to l_min append lb and ub to $l1\_lbN$ and $l1\_ubN$
- If none of the above condition satisfy then the range is unfeasible and is thus added to the list

- If $l1\_lbN$ and $l1\_ubN$ are not empty then run $iterPreanalysis$ with $l1\_lbN$, $l1\_ubN$ and $L/2$ instead of $l1\_lbL$, $l1\_ubL$ and L

def *boundDict*

Input : lbL, ubL, inv_var_index, $sensitive$

Purpose: Convert the list into a dictionary with variable names as keys

Algorithm Perform the following steps:

- For each element in list lbL, ubL. And an entry in the dictionary with key given by variable name and value with a dictionary having the lower and upper bound.

def *updateJSON*

Input : $json\_out$, inv_var_index, $sensitive$, $unbiased$, $unfeasible$

Purpose: Update dictionary that would finally be saved as a JSON.

Algorithm Perform the following steps:

- Create a list using element in *unbiased* converted to a dictionary. For ranges use *boundDict* to get the required format. Add set this list with key "fair" in $json\_out$
- Similarly perform for unfeasible with key "unknown"

def ***convertBound***

Input : lbL, ubL, inv_var_index, $sensitive$

Purpose: Convert the lists into a dictionary with variable names as keys

Algorithm Perform the following steps:

- For each element in list lbL, ubL. And an entry in the dictionary with key given by variable name and value with a tuple having the lower and upper bound.

def ***convertFeasible***

Input : $feasibles$, inv_var_index, $sensitive$

Purpose: Convert the bounds in feasible to a format compatible with analysis with the help of $convertBound$

def ***compressRange***

Input : $rangeL$

Purpose: Merge consecutive ranges in $rangeL$ whenever possible

Algorithm Perform the following steps:

- If the list is empty return itself.
- Otherwise loop over the following step until a loop causes no merges.
  - Create an empty list $rangeLN$. Maintain a range called $rangeC$.
  - Loop over each range in $rangeL$
    - If $rangeC$ doesnt have same outcome as next element in $rangeL$ set flag false
    - If upper bound for a variable in $rangeC$ is same as lower bound for same variable for the next element in $rangeL$ set variable as pos. If it happens for more than one variable set flag false.
    - Other than to satisfy the previous case, the value of lower and upper bound for each variable in $rangeC$ should be same as next element in $rangeL$. Otherwise set flag as false.

    * If flag is true and pos is set. Merge the ranges for pos and update *rangeC* and continue.
    * Otherwise append *rangeC* to *rangeLN*
  - replace *rangeL* with *rangeLN*

def *compressRangeFeasible*

Input : *feasibles*

Purpose: Merge consecutive ranges under the same key whenever possible

Algorithm Perform the following steps:

- merge ranges for each key with the help of *compressRange*

def *preanalysis*

Input : *json_out*, *config*, *domains*

Output : *json_out*, *prioritized*, *time_sec*, *feasiblePe*, *fairP*

Purpose: Perform GPU-preanalysis as defined by *domain* on *config*

Algorithm Perform the following steps:

- Obtain *L_min*, *U_start*, *U_max* from *config*
- Use *createNetworkGPU* from *commons.py* to obtain the tuple containing the detail of the network.
- Call *iterPreanalysis* which recursively preforms preanalysis by auto-tuning L and U
- Use *compressRange* to compress the ranges for *unbiased* and *unfeasible*. Use *compressRangeFeasible* for *feasible*
- Using *convertFeasible* convert feasible to a format suitable for analysis. Apply *compressFeasible* and *priorityFeasible* on feasible to obtain prioritized. These two functions are exactly the same as that for CPU preanalysis.
- Update *json_out* using *updateJSON*
- Set *feasiblePe*, *fairP* as (*unbiasedP*+*feasibleP*) and *unbiasedP* respectively. This is done to follow the same naming convention as CPU preanalysis.
- Return the output

4 symbolic_gpu.py

Being very similar to *deeppoly_gpu.py* thus only the notable differences are pointed out.

Variable Description

symb Matrix to store *symb_status* for each neuron of each layer of the network obtained from each initial. *symb_status* is a tuple of consisting of a value set to 1 if current node is symbolic(otherwise 0), upper and lower bound of neuron if symbolic.

- type: numpy(4-D, float32)
- size: (NOI, NOL+1, MNIL + 1, 3)

def *relu_compute_GPU*

Input : GPU memory: *lbs*, *ubs*, *symb_layer*, $if\_activated$ and *active_pattern*

Output: Update *symb_layer* and *active_pattern*

Purpose: It computes the *symb_status* and *activation_pattern* for each neuron based on its bounds.

relu_compute_helper GPU implemented helper using cuda.jit of numba.

Algorithm Perform the following steps:

- Threads per block is set as the min of (64,16) and (NOI, MNIL). Blocks per grid set to cover all the elements.
- Parallelly consider each neuron in current layer and if it doesn't have a completely positive or negative bound then store the upper bound and zero as lower. At the same time update *active_pattern* using *relu_compute_helper*
- The coefficients are calculated according to the four following cases:
  - If for current node $if\_activation$ is 0. Then *symb_status* us set to (0,0,0) and *activation_status* as 2.
  - If upper bound for the neuron (before relu) is less than zero. Then *symb_status* is set as $(0, 0, 0)$ and *activation_status* is 0

- If lower bound for the neuron (before relu) is greater than zero. Then *symb_status* is set as $(0, 0, 0)$ and *activation_status* is 1
- Otherwise *symb_status* is set as $(1, ubsC, 0)$ where ubsC is the upperbound of current neuron obtained from *ubs* and *activation_status* is 2

Values of *relu_layer* and *active_pattern* are updated and need not be returned

def *back_propagate_GPU*

Input : *affine*, *symb*, *layer*, *if_activation*

Output: Tuple consisting of *l1_lte* and *l1_gte* item[Purpose:] It represents a given layer in form of layer-0 by back-substituting

back_affine_helper GPU implemented helper for back substituting affine layers using cuda.jit of numba.

back_affine_GPU Uses *back_affine_helper* for back substituting affine layers.

back_relu_helper GPU implemented helper for back substituting coefficient terms of relu layers using cuda.jit of numba.

back_relu_GPU Uses *back_relu_coeff_helper* and *back_relu_base_helper* for back substituting affine layers.

Algorithm Perform the following steps:

- Allocate GPU memory to store copies(to be modified) of affine equations for current layer in *ln_coeff_gte* and *ln_coeff_lte*. These make sure the *affine* matrix remains intact
- Run a loop from current layer to the first and for each back substitute the relu activation and affine layer.
- First back substitute the relu if an activation was present for the previous layer. If the affine matrix defines the current matrix as $x = a * y + b * z + c$. If the first element in tuple for y in symb is 1 and for z is 0 then replace y by its upper bound and lower bound in the required in equation. And keep z at it is. This is performed using *back_relu_GPU*
- Next back substitute the affine layer is performed by running a loop to consider i^{th} term in all equations.

- This is done in a way identical to Deep-poly except for Symbolic a variable is substituted only when first element in tuple for that variable in symb is 0. Otherwise it's bounds have directly been added during relu step.

Once we have obtained the current equation in the form of layer-0 we can use *relu_compute_GPU* to update the *relu_state* and *active_pattern* and can continue to the next layer of relu.

5 neurify_gpu.py

Being very similar to *deeppoly_gpu.py* thus only the notable differences are pointed out.

Variable Description

lbs_low Store lower bound of the Less than equal in-equation for each neuron of current layer obtained for each initial.

- type: numpy(2-D,float32)
- size: (NOI,MNIL + 1)

ubs_low Same as above but for upper bound.

lbs_up Same as *lbs_low* but for greater than equal in-equation

ubs_up Same as above but for upper bound.

def *relu_compute_GPU*

Input : GPU memory: *lbs_low*, *ubs_low*, *lbs_up*, *ubs_up*, *symb_layer*, *if_activation* and *active_pattern*

Output: Update *relu_layer* and *active_pattern*

Purpose: It computes the *relu_status* and *activation_pattern* for each neuron based on its bounds.

relu_compute_helper GPU implemented helper using cuda.jit of numba.

Algorithm Perform the following steps:

- Threads per block is set as the min of (64,16) and (NOI, MNIL). Blocks per grid set to cover all the elements.
- Parallelly consider each neuron in current layer and obtain the corresponding slope and y-coeff for Neurify's relu approximation also remember which neurons were active. At the same time update *active_pattern* both using *relu_compute_helper*
- If for current neuron *if_activation* is zero then set *relu_status* as (1,0,1,0) and *active_pattern* as 2.
- The *active_pattern* are calculated according to the following cases:
  - If *ubs_up* of current neuron is less than zero. Then *activation_status* is set to 0.

- If lbs_low of current neuron is greater than equal to zero. Then $activation\_status$ is set to 1.
- Otherwise $activation\_status$ is set to 2

- The first element of $relu\_status$ tuple for current neuron is set to zero and the zeroth element is obtained using the following cases.
  - If ubs_low of current neuron is less than equal to zero. Then value is 0.
  - If lbs_low of current neuron is greater than equal to zero. Then value is 1.
  - Otherwise value is set by slope obtained by $ubs\_low/(ubs\_low - lbs\_low)$ of current neuron
- The third element of $relu\_status$ tuple for current neuron is set to zero and the second element is obtained using the following cases.
  - If ubs_up of current neuron is less than equal to zero. Then value is 0.
  - If lbs_up of current neuron is greater than equal to zero. Then value is 1.
  - Otherwise value is set by slope obtained by $ubs\_up/(ubs\_up - lbs\_up)$ of current neuron. In this case the third element is set by the slope obtained by $-ubs\_up * lbs\_up/(ubs\_up - lbs\_up)$

  Values of $relu\_layer$ and $active\_pattern$ are updated and need not be returned

6 product_gpu.py

It uses the other abstract domains using import. Most variables and functions are similar to those discuss thus only the notably different are presented below.

Name-Space Description

smbG *symbolic_gpu.*

neuG *neurify_gpu.*

dpG *deeppoly_gpu.*

Variable Description

relu_dp relu as in *deeppoly_gpu.*

relu_neu relu as in *neurify_gpu.*

def *oneOutput*

Input :*d_affine*, *d_relu_dp*, *d_relu_neu*, *d_symb*, *if_activation*, *d_l1_lb*, *d_l1_ub*, *outNodes*, *inv_var_index* and *domian*

Output: outcome

Purpose: Convert *outcome* to a form compatible with further analysis steps

Algorithm Perform the following steps for each initial.

- Consider each output node sequentially(say, out1)
- Generate a new layer with MNIL-1 neurons where each represent a neuron minus out1 excluding itself.
- Back substitute a single layer for each of the *domain* and share the bound(max lower and min upper) and repeat until layer one is reached. And lbs and ubs of the layer can be obtained
- If all the lbs obtained from this layer are greater than zero then outcome is out1
- Otherwise consider the next neuron, if none satisfy the condition, outcome is None.
- Append outcome from each initial to a list to be returned.

def ***noPrintCondense***

Input : GPU memory: *affine*, $relu_dp$, $relu_neu$, *symb*,*active_pattern*, *l1_lb*, *l1_ub*, *f_act_pattern* CPU memory: *if_activation* and *domains*

Output: Modify *relu_dp*, *relu_neu*, *symb*, *active_pattern*

Purpose: Calculate for each of the layer sequentially to update *relu_dp*, *relu_neu*, *symb* and *active_pattern*.

Algorithm Perform the following steps for each layer starting from 1 :

- Express each neuron in current layer in form of the first layer using *back_propagate_GPU* for each domain present in *domains*
- For each pair of in-equation obtained find the lbs and ubs and append to a list.
- Find the max of lower bounds min of upper bounds across abstract domain for each neuron. This gives us the narrowest bound.
- If current layer has an activation then use the bounds above for each neuron to modify *relu_dp*, *relu_neu*, *symb*, *active_pattern* using *relu_compute_GPU*.

def ***analyze***

Same as that for *deeppoly_gpu* along with the following change-sAn additional input *domain*, a list domain names as strings. Instead of just *relu*; *relu_dp*, *relu_neu*, *symb* are created as per requirement specified by *domains*