In [2]:

```
# Important Library

!pip install sympy
from sympy import *
```

```
Requirement already satisfied: sympy in /opt/conda/lib/python3.7/site-packages (1.10)
Requirement already satisfied: mpmath>=0.19 in /opt/conda/lib/python3.7/site-packages (fr
om sympy) (1.2.1)
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting
behaviour with the system package manager. It is recommended to use a virtual environment
instead: https://pip.pypa.io/warnings/venv
```

# Tasks for Prospective GSoC 2022 Applicants for the Symbolic Calculation Project

In [1]:

```
import sympy as sp
from sympy import simplify
import numpy as np
from sympy.abc import x
from sympy.parsing.sympy_parser import parse_expr
from random import randint,choice
from tqdm.notebook import trange, tqdm
from torch.utils.data.dataset import Dataset
from torch.utils.data import DataLoader
import os
import io
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import math
import time
import random
device = "cuda" if torch.cuda.is_available() else "cpu"
```

**These utility functions are take from original paper implementation of FAIR's [Deep Learning for Symbolic Mathematics] and some of them updated accordingly to fit this situation in hand**

## Common Task 1. Dataset preprocessing

#### Utility

In [2]:

```
OPERATORS = {
    # Elementary functions
    'add': 2,
    'sub': 2,
    'mul': 2,
    'div': 2,
    'pow': 2,
    'rac': 2,
    'inv': 1,
    'pow':2,
    'sqrt': 1,
    'exp': 1,
    'ln': 1,
```

```
    # Trigonometric Functions
    'sin': 1,
    'cos': 1,
    'tan': 1,
    'cot': 1,
    'sec': 1,
    'csc': 1,
    # Trigonometric Inverses
}

SYMPY_OPERATORS = {

        # Elementary functions
        sp.Add: 'add',
        sp.Mul: 'mul',
        sp.Pow: 'pow',
        sp.exp: 'exp',
        sp.log: 'ln',
        # Trigonometric Functions
        sp.sin: 'sin',
        sp.cos: 'cos',
        sp.tan: 'tan',
        sp.cot: 'cot',
        sp.sec: 'sec',
        sp.csc: 'csc',

    }
variable= {
            'x': sp.Symbol('x', real=True),
}


operators = sorted(list(SYMPY_OPERATORS.values()))
base = 10
balanced = False

def _sympy_to_prefix(op, expr):
    """
    Parse a SymPy expression given an initial root operator.
    """
    n_args = len(expr.args)

    assert (op == 'add' or op == 'mul') and (n_args >= 2) or (op != 'add' and op != 'mul
') and (1 <= n_args <= 2)

    # square root
    if op == 'pow' and isinstance(expr.args[1], sp.Rational) and expr.args[1].p == 1 and
expr.args[1].q == 2:
        return ['sqrt'] + sympy_to_prefix(expr.args[0])

    # parse children
    parse_list = []
    for i in range(n_args):
        if i == 0 or i < n_args - 1:
            parse_list.append(op)
        parse_list += sympy_to_prefix(expr.args[i])

    return parse_list

def write_int(val):
    """
    Convert a decimal integer to a representation in the given base.
    The base can be negative.
    In balanced bases (positive), digits range from -(base-1)//2 to (base-1)//2
    """
    base = 10
    balanced = False
    res = []
    max_digit = abs(base)
    if balanced:
        max_digit = (base - 1) // 2
    else:
```

```
        if base > 0:
            neg = val < 0
            val = -val if neg else val
    while True:
        rem = val % base
        val = val // base
        if rem < 0 or rem > max_digit:
            rem -= base
            val += 1
        res.append(str(rem))
        if val == 0:
            break
    if base < 0 or balanced:
        res.append('INT')
    else:
        res.append('INT-' if neg else 'INT+')
    return res[::-1]



def sympy_to_prefix(expr):
    """
    Convert a SymPy expression to a prefix one.
    """
    if isinstance(expr, sp.Symbol):
        return [str(expr)]
    elif isinstance(expr, sp.Integer):
        return write_int(int(str(expr)))
    elif isinstance(expr, sp.Rational):
        return ['div'] + write_int(int(expr.p)) + write_int(int(expr.q))
    elif expr == sp.E:
        return ['E']
    elif expr == sp.pi:
        return ['pi']
    elif expr == sp.I:
        return ['I']
    # SymPy operator
    for op_type, op_name in SYMPY_OPERATORS.items():
        if isinstance(expr, op_type):
            return _sympy_to_prefix(op_name, expr)
    # unknown operator
    raise Exception(f"Unknown SymPy operator: {expr}")


def parse_int(lst):
    """
    Parse a list that starts with an integer.
    Return the integer value, and the position it ends in the list.
    """
    base = 10
    balanced = False
    print(lst)
    val = 0
    if not (balanced and lst[0] == 'INT' or base >= 2 and lst[0] in ['INT+', 'INT-'] or
base <= -2 and lst[0] == 'INT'):
        raise InvalidPrefixExpression(f"Invalid integer in prefix expression")
    i = 0
    for x in lst[1:]:
        if not (x.isdigit() or x[0] == '-' and x[1:].isdigit()):
            break
        val = val * base + int(x)
        i += 1
    if base > 0 and lst[0] == 'INT-':
        val = -val
    return val, i + 1

def _prefix_to_infix(expr):
    """
    Parse an expression in prefix mode, and output it in either:
      - infix mode (returns human readable string)
      - develop mode (returns a dictionary with the simplified expression)
    """
```

```
    if len(expr) == 0:
        raise InvalidPrefixExpression("Empty prefix list.")
    t = expr[0]
    if t in operators:
        args = []
        l1 = expr[1:]
        for _ in range(OPERATORS[t]):
            i1, l1 = _prefix_to_infix(l1)
            args.append(i1)
        return write_infix(t, args), l1
    elif t in variable :
        return t, expr[1:]
    else:
        val, i = parse_int(expr)
        return str(val), expr[i:]

def prefix_to_infix(expr):
    """
    Prefix to infix conversion.
    """
    p, r = _prefix_to_infix(expr)
    if len(r) > 0:
        raise InvalidPrefixExpression(f"Incorrect prefix expression \"{expr}\". \"{r}\"
was not parsed.")
    return f'({p})'

def write_infix(token, args):
    """
    Infix representation.
    Convert prefix expressions to a format that SymPy can parse.
    """
    if token == 'add':
        return f'({args[0]})+({args[1]})'
    elif token == 'sub':
        return f'({args[0]})-({args[1]})'
    elif token == 'mul':
        return f'({args[0]})*({args[1]})'
    elif token == 'div':
        return f'({args[0]})/({args[1]})'
    elif token == 'pow':
        return f'({args[0]})**({args[1]})'
    elif token == 'rac':
        return f'({args[0]})**(1/({args[1]}))'
    elif token == 'abs':
        return f'Abs({args[0]})'
    elif token == 'inv':
        return f'1/({args[0]})'
    elif token in ['sign', 'sqrt', 'exp', 'ln', 'sin', 'cos', 'tan', 'cot', 'sec', 'csc'
, 'asin', 'acos', 'atan', 'acot', 'asec', 'acsc', 'sinh', 'cosh', 'tanh', 'coth', 'sech'
, 'csch', 'asinh', 'acosh', 'atanh', 'acoth', 'asech', 'acsch']:
        return f'{token}({args[0]})'
    elif token == 'derivative':
        return f'Derivative({args[0]},{args[1]})'
    elif token == 'f':
        return f'f({args[0]})'
    elif token == 'g':
        return f'g({args[0]},{args[1]})'
    elif token == 'h':
        return f'h({args[0]},{args[1]},{args[2]})'
    elif token.startswith('INT'):
        return f'{token[-1]}{args[0]}'
    else:
        return token
    raise InvalidPrefixExpression(f"Unknown token in prefix expression: {token}, with arg
uments {args}")

words = ['<s>','</s>'] + list(variable.keys()) +  operators + ['INT+', 'INT-', 'INT'] +[
str(i) for i in range(10)]
id2word = {i: s for i, s in enumerate(words)}
word2id = {s: i for i, s in id2word.items()}
```

In [3]:

In [3]:

```
# predict a sympy function

def pred_to_sympy(model , expr):
  exp = []
  for i in expr:
    exp.append(sympy_to_prefix(i))
  t = [torch.LongTensor([word2id[w] for w in pref if w in word2id]) for pref in exp]
  lengths = torch.LongTensor([len(s) + 2 for s in t])
  sent = torch.LongTensor(lengths.max().item(), lengths.size(0)).fill_(1)
  assert lengths.min().item() > 2
  sent[0] = 0
  for i, s in enumerate(t):
        sent[1:lengths[i] - 1, i].copy_(s)
        sent[lengths[i] - 1, i] = 0
  src = sent
  return model.pred(src.to(device))
```

In [4]:

```
def idx_to_sp(idx, return_infix=False):
    """
    Convert an indexed prefix expression to SymPy.
    """
    prefix = [id2word[wid] for wid in idx]
    infix = prefix_to_infix(prefix)
    eq = sp.parse_expr(infix)
    return (eq, infix) if return_infix else eq

def convert_to_text(batch,id2word):
    """
    Convert a batch of sequences to a list of text sequences.
    """
    batch = batch.cpu().numpy()
    lengths = sum(batch != 1)

    slen, bs = batch.shape
    assert lengths.max() == slen-1 and lengths.shape[0] == bs
    assert (batch == 0).sum() == bs
    sequences = []

    for j in range(bs):
        words = []
        for k in range(lengths[j]):
            if batch[k, j] == 0:
                break
            words.append(id2word[batch[k, j]])
        sequences.append(" ".join(words))
    return sequences
```

**Generating Dataset**

**Here we have written the code for generating the Dataset**

## Algorithm

1. **Choose a Base Function fun from {sin,cos,tan,exp,log,sec,cosec,cot}**
2. **Generate a random Integer i from {1,2,3} (It is the number of operators to combine)**
3. **Iterate i times**
   - **Choose a combination operator opr from {+,*,/}**
   - **Choose a random integer k from {1,2....10}**
   - **Choose a Base Function fun_new from {sin,cos,tan,exp,log,sec,cosec,cot}**
   - **update fun <- fun opr k * fun_new**
4. **retun fun**

In [5]:

```
### Generating Functions in Sympy Format
### Inspired from Using Algorithm 2 from https://ml4sci.org/assets/faseroh.pdf

N = 100              #Number of functions to generate
debug = False        # For Debugging the Code


# Combination and Base Functions
Combination_Operator = [sp.Add , lambda x,y:x*sp.Pow(y , -1) , sp.Mul]
base_functions = [sp.exp,sp.log,sp.sin,sp.cos,sp.tan,sp.cot,sp.csc,sp.sec]


# Opening three files with 'append' for train,validation and test split respectively
f_train = open("data.train", mode='a', encoding='utf-8')
f_valid = open("data.valid", mode='a', encoding='utf-8')
f_test  = open("data.test", mode='a', encoding='utf-8')

# To generate 70% Train Data , 10% Validation Data and 20% Test Data
split = [7,2,1]

for i in trange(N):
  fun = choice(base_functions)(x)
  if debug : print(f"Initial function generated in Loop {i+1} is : {fun}")
  n = randint(1,3)  # Number of Operators to Combine
  if debug : print(f"Number of Operators Combine in Loop {i+1} is : {n}")
  for j in range(n-1):
    Operator =  choice(Combination_Operator)
    if debug : print(f" Operators Combine Choosen in Loop {i+1} in {j+1} is {Operator}")
    fun = Operator(fun , randint(1,10)*choice(base_functions)(x))

    # Simplyfying Function

  expr = simplify(fun)

  # Creating a line to put in file with its series upto 4th order
  line = (" ".join(sympy_to_prefix(expr)) + "\t" + " ".join( sympy_to_prefix(expr.series
(x,0,4).removeO())))

  if split[0] != 0:
    f_train.write(line + "\n")
    split[0] = split[0] - 1
  elif split[1] != 0:
    f_test.write(line + "\n")
    split[1] = split[1] - 1
  else:
    f_valid.write(line + "\n")
    split = [7,2,1]
  if debug : print(f"Function : {sp.simplify(fun)} and Expansion : {sp.simplify(fun).seri
es(x,0,4)}")

f_train.close()
f_valid.close()
f_test.close()
```

**The above code is only for reproducibility , Actually I have ran this same code on my department server where it takes 2 days , on my computer it was not feasible to do this in so little time, I have attached the Gereated files too there are about 1800000 Training Samples**

**DataLoader**

In [6]:

```
def collate_fn( elements):
    """
    Collate samples into a batch.
    """
    #print(elements)
    x, y = zip(*elements)
```

```
    #nb_ops = [sum(int(word in OPERATORS) for word in seq) for seq in x]
    x = [torch.LongTensor([word2id[w] for w in seq if w in word2id]) for seq in x]
    y = [torch.LongTensor([word2id[w] for w in seq if w in word2id]) for seq in y]
    x = batch_sequences(x)
    y = batch_sequences(y)
    return x.to(device), y.to(device)


def batch_sequences(sequences):
    """
    Take as input a list of n sequences (torch.LongTensor vectors) and return
    a tensor of size (slen, n) where slen is the length of the longest
    sentence, and a vector lengths containing the length of each sentence.
    """
    #print(sequences)
    lengths = torch.LongTensor([len(s) + 2 for s in sequences])
    sent = torch.LongTensor(lengths.max().item(), lengths.size(0)).fill_(1)
    assert lengths.min().item() > 2

    sent[0] = 0
    for i, s in enumerate(sequences):
        sent[1:lengths[i] - 1, i].copy_(s)
        sent[lengths[i] - 1, i] = 0

    return sent
```

In [7]:

```
class Dataset_Loader(Dataset):
  r"""PyTorch Dataset class for loading data.

  This is where the data parsing happens.

  This class is built with reusability in mind.

  Arguments:

    path (:obj:`str`):
        Path to the data partition.

  """

  def __init__(self, path):

    # Check if path exists.
    if not os.path.isfile(path):
      # Raise error if path is invalid.
      raise ValueError('Invalid `path` variable! Needs to be a directory')

    self.SRC = []
    self.TRG = []
    # Since the labels are defined by folders with data we loop
    # through each label.



    with open(path, mode='r', encoding='utf-8') as f:
          lines = [line.rstrip() for line in f]
          self.data = [xy.split('\t') for  xy in lines]
          self.data = [xy for xy in self.data if len(xy) == 2]
    for src,trg in self.data:
      self.SRC.append(src)
      self.TRG.append(trg)

    # Number of examples.
    self.n_examples = len(self.SRC)

    return


  def __len__(self):
```

```
    r"""When used `len` return the number of examples.

    """

    return self.n_examples


  def __getitem__(self, item):
    r"""Given an index return an example from the position.

    Arguments:

      item (:obj:`int`):
          Index position to pick an example to return.

    Returns:
      :obj:`Dict[str, str]`: Dictionary of inputs that are used to feed
      to a model.

    """

    return self.SRC[item].split(),self.TRG[item].split()
```

In [ ]:

```
train_data_loader = Dataset_Loader('data.train')
valid_data_loader = Dataset_Loader('data.valid')
```

In [ ]:

```
train_iterator = DataLoader(
    train_data_loader,
    batch_size=256,
    shuffle=True,
    num_workers=0,
    collate_fn=collate_fn,
    pin_memory=False,
 )
valid_iterator = DataLoader(
    atc,
    batch_size=256,
    shuffle=True,
    num_workers=0,
    collate_fn=collate_fn,
    pin_memory=False,
 )
```

# Common Task 2. Use LSTM model

## LSTM Model

In [8]:

```
SEED = 1234

random.seed(SEED)
np.random.seed(SEED)
torch.manual_seed(SEED)
torch.cuda.manual_seed(SEED)
torch.backends.cudnn.deterministic = True
```

## Encoder

In [9]:

```
class Encoder(nn.Module):
```

```
    def __init__(self, input_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()

        self.hid_dim = hid_dim
        self.n_layers = n_layers
        self.embedding = nn.Embedding(input_dim, emb_dim)
        self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers, dropout = dropout)
        self.dropout = nn.Dropout(dropout)

    def forward(self, src):
        embedded = self.dropout(self.embedding(src))
        outputs, (hidden, cell) = self.rnn(embedded)
        return hidden, cell
```

### Decoder

In [10]:

```
class Decoder(nn.Module):
    def __init__(self, output_dim, emb_dim, hid_dim, n_layers, dropout):
        super().__init__()

        self.output_dim = output_dim
        self.hid_dim = hid_dim
        self.n_layers = n_layers
        self.embedding = nn.Embedding(output_dim, emb_dim)
        self.rnn = nn.LSTM(emb_dim, hid_dim, n_layers, dropout = dropout)
        self.fc_out = nn.Linear(hid_dim, output_dim)
        self.dropout = nn.Dropout(dropout)

    def forward(self, input, hidden, cell):

        input = input.unsqueeze(0)
        embedded = self.dropout(self.embedding(input))
        output, (hidden, cell) = self.rnn(embedded, (hidden, cell))
        prediction = self.fc_out(output.squeeze(0))
        return prediction, hidden, cell
```

### seq2seq Model

In [11]:

```
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, device):
        super().__init__()

        self.encoder = encoder
        self.decoder = decoder
        self.device = device

        assert encoder.hid_dim == decoder.hid_dim, \
            "Hidden dimensions of encoder and decoder must be equal!"
        assert encoder.n_layers == decoder.n_layers, \
            "Encoder and decoder must have equal number of layers!"

    def forward(self, src, trg, teacher_forcing_ratio = 0.5):

        batch_size = trg.shape[1]
        trg_len = trg.shape[0]
        trg_vocab_size = self.decoder.output_dim
        outputs = torch.zeros(trg_len, batch_size, trg_vocab_size).to(self.device)
        hidden, cell = self.encoder(src)
        input = trg[0,:]

        for t in range(1, trg_len):
            output, hidden, cell = self.decoder(input, hidden, cell)
            outputs[t] = output
            teacher_force = random.random() < teacher_forcing_ratio
```

```
            top1 = output.argmax(1)
            input = trg[t] if teacher_force else top1
        return outputs

    def pred(self,src):
        input = src[0,:].to(device)
        hidden, cell = self.encoder(src)
        output, hidden, cell = self.decoder(input, hidden, cell)
        result = output.argmax(1)
        cram = result == 0
        lst = []
        lst.append(result.tolist())
        for i in range(100):
              output, hidden, cell = self.decoder(result, hidden, cell)
              result = output.argmax(1)
              result[cram] = 1
              cram[result == 0] = True
              lst.append(result.tolist())
              if all(result == 1) : break

        return torch.tensor(lst)
```

## HyperParameters

In [12]:

```
INPUT_DIM = len(word2id)
OUTPUT_DIM = len(word2id)
ENC_EMB_DIM = 256
DEC_EMB_DIM = 256
HID_DIM = 512
N_LAYERS = 2
ENC_DROPOUT = 0.2
DEC_DROPOUT = 0.2

enc = Encoder(INPUT_DIM, ENC_EMB_DIM, HID_DIM, N_LAYERS, ENC_DROPOUT)
dec = Decoder(OUTPUT_DIM, DEC_EMB_DIM, HID_DIM, N_LAYERS, DEC_DROPOUT)

model = Seq2Seq(enc, dec, device).to(device)
```

In [13]:

```
def init_weights(m):
    for name, param in m.named_parameters():
        nn.init.uniform_(param.data, -0.08, 0.08)

model.apply(init_weights)
```

Out[13]:

```
Seq2Seq(
  (encoder): Encoder(
    (embedding): Embedding(27, 256)
    (rnn): LSTM(256, 512, num_layers=2, dropout=0.2)
    (dropout): Dropout(p=0.2, inplace=False)
  )
  (decoder): Decoder(
    (embedding): Embedding(27, 256)
    (rnn): LSTM(256, 512, num_layers=2, dropout=0.2)
    (fc_out): Linear(in_features=512, out_features=27, bias=True)
    (dropout): Dropout(p=0.2, inplace=False)
  )
)
```

In [14]:

```
def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

print(f'The model has {count_parameters(model):,} trainable parameters')
```

```
The model has 7,384,091 trainable parameters
```

In [15]:

```
optimizer = optim.Adam(model.parameters())
```

In [16]:

```
TRG_PAD_IDX = 1

criterion = nn.CrossEntropyLoss(ignore_index = TRG_PAD_IDX)
```

In [21]:

```
def train(model, iterator, optimizer, criterion, clip):

    model.train()

    epoch_loss = 0

    for i, batch in enumerate(iterator):

        src = batch[0]
        trg = batch[1]
        if debug : print(f"Batch {i} have input dimension {src.shape} and output dimensi
on {trg.shape}")
        optimizer.zero_grad()

        output = model(src, trg)

        #trg = [trg len, batch size]
        #output = [trg len, batch size, output dim]

        output_dim = output.shape[-1]

        output = output[1:].view(-1, output_dim)
        trg = trg[1:].view(-1)

        #trg = [(trg len - 1) * batch size]
        #output = [(trg len - 1) * batch size, output dim]

        loss = criterion(output, trg)

        loss.backward()

        torch.nn.utils.clip_grad_norm_(model.parameters(), clip)

        optimizer.step()

        epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

In [22]:

```
def evaluate(model, iterator, criterion):

    model.eval()

    epoch_loss = 0

    with torch.no_grad():

        for i, batch in enumerate(iterator):

            src = batch[0]
            trg = batch[1]

            output = model(src, trg, 0) #turn off teacher forcing

            #trg = [trg len, batch size]
```

```
            #output = [trg len, batch size, output dim]

            output_dim = output.shape[-1]

            output = output[1:].view(-1, output_dim)
            trg = trg[1:].view(-1)

            #trg = [(trg len - 1) * batch size]
            #output = [(trg len - 1) * batch size, output dim]

            loss = criterion(output, trg)

            epoch_loss += loss.item()

    return epoch_loss / len(iterator)
```

In [23]:

```
def epoch_time(start_time, end_time):
    elapsed_time = end_time - start_time
    elapsed_mins = int(elapsed_time / 60)
    elapsed_secs = int(elapsed_time - (elapsed_mins * 60))
    return elapsed_mins, elapsed_secs
```

In [ ]:

```
N_EPOCHS = 30
CLIP = 1

best_valid_loss = float('inf')

for epoch in range(N_EPOCHS):

    start_time = time.time()

    train_loss = train(model, train_iterator, optimizer, criterion, CLIP)
    valid_loss = evaluate(model, valid_iterator, criterion)

    end_time = time.time()

    epoch_mins, epoch_secs = epoch_time(start_time, end_time)

    if valid_loss < best_valid_loss:
        best_valid_loss = valid_loss
        torch.save(model.state_dict(), 'tut1-model.pt')

    print(f'Epoch: {epoch+1:02} | Time: {epoch_mins}m {epoch_secs}s')
    print(f'\tTrain Loss: {train_loss:.3f} | Train PPL: {math.exp(train_loss):7.3f}')
    print(f'\t Val. Loss: {valid_loss:.3f} |  Val. PPL: {math.exp(valid_loss):7.3f}')
```

**I have runned the Model Several times and saved the model as tut1-mode (3).pt , We can call the model from following codel**

In [17]:

```
model.load_state_dict(torch.load('../input/modelpy1/tut1-model (3).pt'))
```

Out[17]:

```
<All keys matched successfully>
```

## Testing

In [18]:

```
test_loader = Dataset_Loader('../input/testingdata/data.test')
```

In [19]:

```
test_iterator = DataLoader(
    test_loader,
    batch_size=256,
    shuffle=True,
    num_workers=0,
    collate_fn=collate_fn,
    pin_memory=False,
 )
```

In [24]:

```
test_loss = evaluate(model,test_iterator, criterion)
```

In [25]:

```
print(f"Test Loss: {test_loss}" )
```

```
Test Loss: 0.05521275207651698
```

In [ ]:

```
```