```
from sympy import *
import random
import re
import tokenize
from io import StringIO
import torch
from torch import nn
from torch.autograd import Variable
import pandas as pd
import torch.nn.functional as F
from tqdm import tqdm
import numpy as np
import math


device = torch.device("cuda" if torch.torch.cuda.is_available() else "cpu")


"""
A class for representing and generating expressions in Polish notation

Expressions are generated with the algorithm descibed in Appendix C of DEEP LEARNING F
which aims to weight deep, shallow, left-leaning, and right leaning expression trees a

We used polish notation for this project as it can more concisely represent expressior

"""

class Expression:

  """
  dictionary of operations with their corresponding arity as keys
  """
  _ops = {
    'sin' : 1,
    'cos' : 1,
    'tan' : 1,
    'square' : 1,
    'cube' : 1,
    'exp' : 1,
    'log': 1,

    '+' : 2,
    '-' : 2,
    '*' : 2,
    '/' : 2,
    '**' : 2,
  }
```

```
"""
Maps operations to anonymous function that generates an infix expression
"""
_infix_reps = {
    'sin': lambda args: f'sin({args[0]})',
    'cos': lambda args: f'cos({args[0]})',
    'tan': lambda args: f'tan({args[0]})',
    'square': lambda args: f'({args[0]})**2',
    'cube': lambda args: f'({args[0]})**3',
    'exp': lambda args: f'exp({args[0]})',
    'log': lambda args: f'log({args[0]})',

    '+': lambda args: f'({args[0]})+({args[1]})',
    '-': lambda args: f'({args[0]})-({args[1]})',
    '*': lambda args: f'({args[0]})*({args[1]})',
    '/': lambda args: f'({args[0]})/({args[1]})',
    '**': lambda args: f'({args[0]})**({args[1]})'
}
"""
unnormalized probabilities of each unary op
"""
_unary_op_probs = {
  'sin' : 1,
  'cos' : 1,
  'tan' : 2,
  'square' : 4,
  'cube' : 3,
  'exp' : 2,
  'log' : 1
}
"""
unnormalized probabilities of each binary op
"""
_bin_op_probs = {
  '+' : 3,
  '-' : 3,
  '*' : 2,
  '/' : 2,
  '**' : 1,
}
"""
maps sympy functions to ones contained in this class
"""
_from_sympy = {
    sin : 'sin',
    cos : 'cos',
    tan : 'tan',
    exp : 'exp',
    log : 'log',

    Add : '+',
```

```
    Mul : '*',
    Pow : '**',
  }
  """
    Generates a numpy array representing counts of possible trees of n internal nodes
    D(0, n) = 0
    D(e, 0) = L ** e
    D(e, n) = L * D(e - 1, n) + p_1 * D(e, n - 1) + p_2 * D(e + 1, n - 1)

    from Appendix C.2 of DEEP LEARNING FOR SYMBOLIC MATHEMATICS (Guillaume Lample, Fra
  """
  def _unary_binary_dist(self, size):

    #generating transposed version
    D = np.zeros((size * 2 + 1, size))
    D[:,0] = self._num_leaves ** np.arange(size * 2 + 1)
    D[0,0] = 0

    for n in range(1, size):
      for e in range(1, size * 2):
        D[e, n] = self._num_leaves * D[e - 1, n] + self._num_unary_ops * D[e, n - 1] +
    return D[:,:size+1]

  """



  Samples a position of a node and arity

  from Appendix C.3 of DEEP LEARNING FOR SYMBOLIC MATHEMATICS (Guillaume Lample, Franc

  Parameters
    e -- number of empty nodes to sample from
    n -- number of operations
  """
  def _sample(self, e, n):

    P = np.zeros((e, 2))

    for k in range(e):
      P[k,0] = (self._num_leaves ** k) * self._num_unary_ops * self._unary_binary_dist
    for k in range(e):
      P[k,1] = (self._num_leaves ** k) * self._num_bin_ops * self._unary_binary_dist[e

    P /= self._unary_binary_dist[e,n]
    k = np.random.choice(2*e, p=P.T.flatten())

    arity = 1 if k < e else 2
    k = k % e
    return k , arity

  def _choose_unary_op(self):
```

```
    return np.random.choice(tuple(self._unary_op_probs.keys()), p=self._unary_op_norm_

  def _choose_bin_op(self):
    return np.random.choice(tuple(self._bin_op_probs.keys()), p=self._bin_op_norm_prob

  def _choose_leaf(self):
    if(random.random() < 0.3):
      return 'x'
    return random.randrange(0,10)

  def _gen_from_sympy(self, expr):
    self._rep = []
    stack = [expr]
    while(len(stack) != 0):
      expr = stack.pop()
      #print(expr, self._rep)
      if isinstance(expr, Symbol):
        self._rep.append(str(expr))
      elif isinstance(expr, Integer):
        self._rep.append(str(expr))
      elif isinstance(expr, Rational):
        self._rep.append('/')

        args = str(expr).split('/')
        self._rep.append(str(args[0]))
        self._rep.append(str(args[1]))
      elif expr == E:
        self._rep.append('e')
      elif expr == pi:
        self._rep.append('pi')
      elif expr == I:
        self._rep.append('i')


      else:
        for i in range(len(expr.args) - 1):
          self._rep.append(self._from_sympy[type(expr)])

        for item in expr.args:
          stack.append(item)

  def _gen_random(self, num_ops):
    self._num_leaves = 1
    self._num_bin_ops = len(self._bin_op_probs.keys())
    self._num_unary_ops = len(self._unary_op_probs.keys())

    self._unary_binary_dist = self._unary_binary_dist(num_ops + 1)

    self._bin_op_norm_prob = np.fromiter(self._bin_op_probs.values(), dtype=float)
    self._bin_op_norm_prob /= self._bin_op_norm_prob.sum()
```

```
    self._unary_op_norm_prob = np.fromiter(self._unary_op_probs.values(), dtype=float)
    self._unary_op_norm_prob /= self._unary_op_norm_prob.sum()

    rep = [None]
    e = 1
    skipped = 0
    for n in range(num_ops, 0, - 1):
      k, arity = self._sample(e, n)
      skipped += k
      if arity == 1:
        op = self._choose_unary_op()

        #O(N) is bad for this. TODO: change to a dynamic programming approach so it is
        encountered_empty = 0
        pos = 0
        for i in range(len(rep)):
          if(rep[i] == None):
            encountered_empty += 1
          if encountered_empty == skipped + 1:
            pos = i
            break


        rep = rep[:pos] + [op] + [None] + rep[pos + 1:]
        e = e - k
      else:
        op = self._choose_bin_op()

        encountered_empty = 0
        pos = 0
        for i in range(len(rep)):
          if(rep[i] == None):
            encountered_empty += 1
          if encountered_empty == skipped + 1:
            pos = i
            break

        rep = rep[:pos] + [op] + [None] + [None] + rep[pos + 1:]
        e = e - k + 1

    for i in range(len(rep)):
      if(rep[i] is None):
        rep[i] = self._choose_leaf()
    self._rep = rep


  def __init__(self, expr=None, num_ops=None):

    if(expr is not None):
      self._gen_from_sympy(expr)
    else:
```

```
      self._gen_random(num_ops)



  def to_infix(self):
    stack = []

    for i in range(len(self._rep) - 1, -1, -1):
      token = self._rep[i]

      if token in self._ops:
        arity = self._ops[token]

        args = stack[-arity:]
        stack = stack[:-arity]

        stack.append(self._infix_reps[token](args))
      else:
        stack.append(token)
    return stack.pop()
  def get_rep(self):
    return self._rep


def taylor_series(f_str, a, order):
  x = symbols('x')
  f = parse_expr(f_str)
  ret = f.subs(x, a)
  for i in range(1,order + 1):
    #print(i)
    f = diff(f,x)
    ret = ret + (f*(x-a))/factorial(i)
  return ret


def test_expr():
  for i in range(10):
    expr = Expression(num_ops=3)
    print(f"Expression {i+1}:")
    print(f"\tTokenixed prefix: ", expr.get_rep())
    print(f"\tInfix: ", expr.to_infix())
test_expr()
```

```
Expression 1:
        Tokenixed prefix:  ['/', 'exp', '+', 9, 3, 'x']
        Infix:  (x)/(exp((3)+(9)))
Expression 2:
        Tokenixed prefix:  ['*', 'square', 'x', 'cube', 4]
        Infix:  ((4)**3)*((x)**2)
Expression 3:
        Tokenixed prefix:  ['square', 'tan', '+', 7, 'x']
        Infix:  (tan((x)+(7)))**2
```

```
    Expression 4:
            Tokenixed prefix:  ['-', '/', 'cube', 0, 'x', 8]
            Infix:  (8)-((x)/((0)**3))
    Expression 5:
            Tokenixed prefix:  ['cube', 'cube', 'exp', 'x']
            Infix:  ((exp(x))**3)**3
    Expression 6:
            Tokenixed prefix:  ['exp', '/', '-', 'x', 2, 7]
            Infix:  exp((7)/((2)-(x)))
    Expression 7:
            Tokenixed prefix:  ['*', '-', 'x', 'square', 6, 3]
            Infix:  (3)*(((6)**2)-(x))
    Expression 8:
            Tokenixed prefix:  ['*', '-', '+', 4, 7, 3, 'x']
            Infix:  (x)*((3)-((7)+(4)))
    Expression 9:
            Tokenixed prefix:  ['/', 'tan', 9, '*', 'x', 2]
            Infix:  ((2)*(x))/(tan(9))
    Expression 10:
            Tokenixed prefix:  ['-', 'exp', 6, '**', 'x', 4]
            Infix:  ((4)**(x))-(exp(6))
```

```
def gen_pair(ops=3):
  expr = Expression(num_ops=ops)
  tay = taylor_series(expr.to_infix(), Symbol('a'), 4)
  tay_rep = Expression(expr=tay)
  return expr, tay_rep



class FunctionDataset(torch.utils.data.Dataset):
  """
  not proud of the specification of both ops and sequence length but it works for now
  """
  def __init__(self, ops=3, max_seq_length=32, num_items=100):

    raw_input = []
    raw_output = []

    while len(raw_input) < num_items:
      expr, tay = gen_pair(ops)
      #we only need the postfix representation
      expr = expr.get_rep()
      tay = tay.get_rep()

      #discards expressions too long and nan values
      if(len(expr) + 2 <= max_seq_length and len(tay) + 2 <= max_seq_length and tay !=
        #insert start and end tokens
        expr.insert(0,'<SOS>')
        expr.append('<EOS>')

        tay.insert(0,'<SOS>')
        tay.append('<EOS>')
```

```
        raw_input.append(expr)
        raw_output.append(tay)

    #generate vocab
    self.vocab = set()
    for expr, tay in zip(raw_input, raw_output):
      self.vocab |= set(expr) |(set(tay))

    #token -> idx
    self.token_to_idx = {value : index + 1 for index, value in enumerate(self.vocab)}

    #idx -> token
    self.idx_to_token = {index + 1 : value for index, value in enumerate(self.vocab)}

    self.input = []
    self.output = []
    for raw_expr, raw_tay in zip(raw_input, raw_output):
      expr = [self.token_to_idx[token] for token in raw_expr] + [0] * (max_seq_length
      tay = [self.token_to_idx[token] for token in raw_tay] + [0] * (max_seq_length -

      self.input.append(torch.tensor(expr, dtype=torch.long, device=device))
      self.output.append(torch.tensor(tay, dtype=torch.long, device=device))

  def __len__(self):
    return len(self.input)

  def __getitem__(self, idx):
    return self.input[idx].to(device), self.output[idx].to(device)

  def get_alphabet(self):
    return self.vocab
```

```
d = FunctionDataset(num_items=1000)
train_idx = list(range(0, int(9*len(d)/10)))
test_idx = list(range(int(9*len(d)/10), len(d)))
train_dataset = torch.utils.data.Subset(d, train_idx)
test_dataset = torch.utils.data.Subset(d, test_idx)
```

```
class Encoder(nn.Module):
  def __init__(self, vocab_size, embedding_dim=512, num_layers=2, hidden_size=512, drc
    super(Encoder, self).__init__()
    self.embedding_dim = embedding_dim
    self.num_layers = num_layers
    self.hidden_size = hidden_size

    self.embedding = nn.Embedding(
```

```
            num_embeddings=vocab_size,
            embedding_dim=self.embedding_dim
        )
        self.lstm = nn.LSTM(
            input_size=self.embedding_dim,
            hidden_size=self.hidden_size,
            num_layers=self.num_layers,
            dropout=dropout,
        )
    """
    input shape (SEQUENCE_LENGTH, BATCH_SIZE)
    h,c shape (HIDDEN_SIZE)
    """
    def forward(self, x):
        embed = self.embedding(x)
        output, (h,c) = self.lstm(embed)
        return h, c



class Decoder(nn.Module):
    def __init__(self, vocab_size, embedding_dim=512, num_layers=2, hidden_size=512, dro
        super(Decoder, self).__init__()
        self.embedding_dim = embedding_dim
        self.num_layers = num_layers
        self.output_size = vocab_size
        self.hidden_size = hidden_size

        self.embedding = nn.Embedding(
            num_embeddings=vocab_size,
            embedding_dim=self.embedding_dim
        )
        self.lstm = nn.LSTM(
            input_size=self.embedding_dim,
            hidden_size=self.hidden_size,
            num_layers=self.num_layers,
            dropout=0.2,
        )
        self.out = nn.Linear(self.hidden_size, self.output_size)
        self.softmax = nn.LogSoftmax(dim=2)
        self.to(device)
    """
    input shape (BATCH_SIZE)
    output shape
    """
    def forward(self, input, h_0, c_0):
        embedded = self.embedding(input.unsqueeze(0))
        output, (h,c) = self.lstm(embedded, (h_0, c_0))
        output = self.out(output)
        output = self.softmax(output)
```

```
      return output.squeeze(0), h , c


  class Model(nn.Module):
    def __init__(self, encoder, decoder):
      super(Model, self).__init__()
      self.encoder = encoder
      self.decoder = decoder
      self.to(device)
    """
    Input tensor of shape (SEQUENCE_LENGTH, BATCH_SIZE)
    Output tensor of shape (SEQUENCE_LENGTH, BATCH_SIZE, VOCAB_SIZE)

    if tgt is none use teacher forecasting
    """
    def forward(self, input, tgt=None):
      if len(input.shape) < 2:
          input = input.unsqueeze(1)
      batch_size = input.shape[1]
      h, c = enc(input)
      target = torch.zeros(batch_size, dtype=torch.long).to(device)
      if tgt is None:
        max_seq_length = input.shape[0]
        target[:] = d.token_to_idx['<SOS>']
      else:
        max_seq_length = tgt.shape[1]
        target[:] = tgt[:,0]
      outputs = torch.zeros(max_seq_length, batch_size, dec.output_size, dtype=torch.flo
      for i in range(max_seq_length):
          prediction, h, c = dec(target, h, c)
          outputs[i] = prediction
          if tgt is None:
            target = prediction.argmax(dim=1)
          else:
            target = tgt[:,i]
      return outputs



  enc = Encoder(len(d.get_alphabet()) + 1)
  dec = Decoder(len(d.get_alphabet()) + 1)
  m = Model(enc,dec).to(device)


  def test_epoch_LSTM(model, test_loader, criterion, batch_size=4):
    model.eval()
    total_loss = 0
    total_items = 0
    num_correct = 0
    for src, tgt in tqdm(test_loader):
      src = src.to(device)
      tgt = tgt.to(device)
```

```
    pred = model(src.squeeze().T, tgt=tgt[:,:-1])
    pred = pred.permute((1,2,0))
    tgt_out = tgt[:,1:]

    loss = criterion(pred, tgt_out)

    total_loss += loss.item()
    total_items += (tgt_out != 0).sum(dim=(0,1))

    num_correct += (torch.logical_and((logits.argmax(dim=2) == tgt_out), (tgt_out != (
  return total_loss, num_correct / total_items


def train_epoch_LSTM(model, train_loader, optimizer, criterion, batch_size=4):
  model.train()
  total_loss = 0
  total_items = 0
  num_correct = 0
  for src, tgt in tqdm(train_loader):
    src = src.to(device)
    tgt = tgt.to(device)

    pred = model(src.squeeze().T,tgt=tgt[:,:-1])

    pred = pred.permute((1,2,0))
    tgt_out = tgt[:,1:]
    loss = criterion(pred, tgt_out)

    optimizer.zero_grad()
    loss.backward()
    optimizer.step()

    total_loss += loss.item()
    total_items += (tgt_out != 0).sum(dim=(0,1))

    num_correct += (torch.logical_and((pred.argmax(dim=1) == tgt_out), (tgt_out != 0))
  return total_loss, num_correct / total_items


def train_LSTM(model, train_dataset,  test_dataset, batch_size=32, epochs=50):
  train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shu
  test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuff
  criterion = nn.CrossEntropyLoss()


  optim = torch.optim.Adam(model.parameters(), lr=1e-3)
  for e in range(epochs):
    train_loss, train_acc = train_epoch_LSTM(model, train_loader, optim, criterion, ba
    test_loss, test_acc = train_epoch_LSTM(model, test_loader, optim, criterion, batch
    print(f'Epoch: {e + 1} Training Loss: {train_loss} Training Accuracy: {train_acc}
```

```
train_LSTM(m, train_dataset, test_dataset, batch_size=32)
```

```
100%|██████████| 29/29 [00:02<00:00, 12.40it/s]
100%|██████████| 4/4 [00:00<00:00, 12.54it/s]
Epoch: 1 Training Loss: 33.54634836316109 Training Accuracy: 0.1103273630142211
100%|██████████| 29/29 [00:02<00:00, 12.17it/s]
100%|██████████| 4/4 [00:00<00:00, 12.72it/s]
Epoch: 2 Training Loss: 16.89692533016205 Training Accuracy: 0.306927114725112
100%|██████████| 29/29 [00:02<00:00, 12.78it/s]
100%|██████████| 4/4 [00:00<00:00, 13.01it/s]
Epoch: 3 Training Loss: 14.930061250925064 Training Accuracy: 0.393018633127212
100%|██████████| 29/29 [00:02<00:00, 13.37it/s]
100%|██████████| 4/4 [00:00<00:00, 13.92it/s]
Epoch: 4 Training Loss: 13.757144123315811 Training Accuracy: 0.431542783975601
100%|██████████| 29/29 [00:02<00:00, 12.93it/s]
100%|██████████| 4/4 [00:00<00:00, 11.98it/s]
Epoch: 5 Training Loss: 13.655325770378113 Training Accuracy: 0.461566299200058
100%|██████████| 29/29 [00:02<00:00, 12.92it/s]
100%|██████████| 4/4 [00:00<00:00, 11.31it/s]
Epoch: 6 Training Loss: 12.540652394294739 Training Accuracy: 0.476035445928573
100%|██████████| 29/29 [00:02<00:00, 12.35it/s]
100%|██████████| 4/4 [00:00<00:00, 11.27it/s]
Epoch: 7 Training Loss: 11.688833743333817 Training Accuracy: 0.500813901424408
100%|██████████| 29/29 [00:02<00:00, 11.68it/s]
100%|██████████| 4/4 [00:00<00:00,  9.46it/s]
Epoch: 8 Training Loss: 11.284646302461624 Training Accuracy: 0.5116657614707947
100%|██████████| 29/29 [00:02<00:00, 12.06it/s]
100%|██████████| 4/4 [00:00<00:00, 13.42it/s]
Epoch: 9 Training Loss: 10.787398785352707 Training Accuracy: 0.5210707187652588
100%|██████████| 29/29 [00:02<00:00, 12.77it/s]
100%|██████████| 4/4 [00:00<00:00, 12.80it/s]
Epoch: 10 Training Loss: 10.22880694270134 Training Accuracy: 0.5395188927650452
100%|██████████| 29/29 [00:02<00:00, 12.96it/s]
100%|██████████| 4/4 [00:00<00:00, 12.56it/s]
Epoch: 11 Training Loss: 9.765019744634628 Training Accuracy: 0.5608609318733215
100%|██████████| 29/29 [00:02<00:00, 12.60it/s]
100%|██████████| 4/4 [00:00<00:00, 13.84it/s]
Epoch: 12 Training Loss: 9.277985289692879 Training Accuracy: 0.5653825402259827
100%|██████████| 29/29 [00:02<00:00, 13.19it/s]
100%|██████████| 4/4 [00:00<00:00, 11.74it/s]
Epoch: 13 Training Loss: 8.85831581056118 Training Accuracy: 0.5939591526985168
100%|██████████| 29/29 [00:02<00:00, 12.93it/s]
100%|██████████| 4/4 [00:00<00:00, 12.35it/s]
Epoch: 14 Training Loss: 8.514876186847687 Training Accuracy: 0.6080665588378906
100%|██████████| 29/29 [00:02<00:00, 12.77it/s]
100%|██████████| 4/4 [00:00<00:00, 13.36it/s]
Epoch: 15 Training Loss: 8.178459718823433 Training Accuracy: 0.6214505434036255
100%|██████████| 29/29 [00:02<00:00, 12.98it/s]
100%|██████████| 4/4 [00:00<00:00, 13.02it/s]
Epoch: 16 Training Loss: 7.890336871147156 Training Accuracy: 0.633387565612793
100%|██████████| 29/29 [00:02<00:00, 12.95it/s]
100%|██████████| 4/4 [00:00<00:00, 13.32it/s]
Epoch: 17 Training Loss: 7.656392619013786 Training Accuracy: 0.6404412984848022
100%|██████████| 29/29 [00:02<00:00, 12.55it/s]
100%|██████████| 4/4 [00:00<00:00, 14.10it/s]
```

```
Epoch: 18 Training Loss: 7.175373286008835 Training Accuracy: 0.6648580431938171
100%|██████████| 29/29 [00:02<00:00, 12.98it/s]
100%|██████████| 4/4 [00:00<00:00, 13.10it/s]
Epoch: 19 Training Loss: 7.184923782944679 Training Accuracy: 0.6594321131706238
100%|██████████| 29/29 [00:02<00:00, 12.70it/s]
```

```
#adapted from https://torchtutorialstaging.z5.web.core.windows.net/beginner/translatio
class PositionalEncoding(nn.Module):
    def __init__(self, emb_size: int, dropout, maxlen: int = 5000):
        super(PositionalEncoding, self).__init__()
        den = torch.exp(- torch.arange(0, emb_size, 2) * math.log(10000) / emb_size)
        pos = torch.arange(0, maxlen).reshape(maxlen, 1)
        pos_embedding = torch.zeros((maxlen, emb_size))
        pos_embedding[:, 0::2] = torch.sin(pos * den)
        pos_embedding[:, 1::2] = torch.cos(pos * den)
        pos_embedding = pos_embedding.unsqueeze(-2)

        self.dropout = nn.Dropout(dropout)
        self.register_buffer('pos_embedding', pos_embedding)

    def forward(self, token_embedding):
        return self.dropout(token_embedding +
                            self.pos_embedding[:token_embedding.size(0),:])

def generate_square_subsequent_mask(sz):
    mask = (torch.triu(torch.ones((sz, sz), device=device)) == 1).transpose(0, 1)
    mask = mask.float().masked_fill(mask == 0, float('-inf')).masked_fill(mask == 1, f
    return mask

def create_mask(src, tgt):
  src_seq_len = src.shape[0]
  tgt_seq_len = tgt.shape[0]

  tgt_mask = generate_square_subsequent_mask(tgt_seq_len)
  src_mask = torch.zeros((src_seq_len, src_seq_len), device=device).type(torch.bool)

  src_padding_mask = (src == 0).transpose(0, 1)
  tgt_padding_mask = (tgt == 0).transpose(0, 1)
  return src_mask, tgt_mask, src_padding_mask, tgt_padding_mask

class TransformerModel(nn.Module):
    def __init__(self, num_encoder_layers, nhead, num_decoder_layers,
                 emb_size, src_vocab_size, tgt_vocab_size,
                 dim_feedforward:int = 512, dropout:float = 0.1):
        super(TransformerModel, self).__init__()
        encoder_layer = nn.TransformerEncoderLayer(d_model=emb_size, nhead=nhead,
                                                   dim_feedforward=dim_feedforward)
        self.transformer_encoder = nn.TransformerEncoder(encoder_layer, num_layers=num
        decoder_layer = nn.TransformerDecoderLayer(d_model=emb_size, nhead=nhead,
                                                   dim_feedforward=dim_feedforward)
        self.transformer_decoder = nn.TransformerDecoder(decoder_layer, num_layers=num
```

```
        self.generator = nn.Linear(emb_size, tgt_vocab_size)
        self.emb_size = emb_size
        self.src_tok_emb = self.embedding = nn.Embedding(src_vocab_size, emb_size)
        self.tgt_tok_emb = self.embedding = nn.Embedding(tgt_vocab_size, emb_size)
        self.positional_encoding = PositionalEncoding(emb_size, dropout=dropout)

    def forward(self, src, trg, src_mask,
                tgt_mask, src_padding_mask,
                tgt_padding_mask, memory_key_padding_mask):
        src_emb = self.positional_encoding(self.src_tok_emb(src)* math.sqrt(self.emb_s
        tgt_emb = self.positional_encoding(self.tgt_tok_emb(trg)* math.sqrt(self.emb_s
        memory = self.transformer_encoder(src_emb, src_mask, src_padding_mask)
        outs = self.transformer_decoder(tgt_emb, memory, tgt_mask, None,
                                        tgt_padding_mask, memory_key_padding_mask)
        return self.generator(outs)


model = TransformerModel(num_encoder_layers=6, nhead=8, num_decoder_layers=6,
                 emb_size=512, src_vocab_size=(len(d.get_alphabet()) + 1), tgt_vocab_s
                 dim_feedforward = 512, dropout = 0.2).to(device)


def train_epoch_transformer(model, train_loader, optimizer, criterion, batch_size):
  model.train()
  total_loss = 0
  num_correct = 0
  total_items = 0
  for src, tgt in tqdm(train_loader):
      src = src.to(device).T
      tgt = tgt.to(device).T

      tgt_input = tgt[:-1, :]

      src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(src, tgt_ir

      logits = model(src, tgt_input, src_mask, tgt_mask,
                             src_padding_mask, tgt_padding_mask, src_padding_mask)

      optimizer.zero_grad()

      tgt_out = tgt[1:,:]
      loss = criterion(logits.reshape(-1, logits.shape[-1]), tgt_out.reshape(-1))

      loss.backward()
      optimizer.step()

      total_loss += loss.item()
      total_items += (tgt_out != 0).sum(dim=(0,1))

      num_correct += (torch.logical_and((logits.argmax(dim=2) == tgt_out), (tgt_out !=
  return total_loss / len(train_loader), num_correct / total_items
```

```
def test_epoch_transformer(model, test_loader, criterion, batch_size):
  model.eval()
  total_loss = 0
  num_correct = 0
  total_items = 0
  for src, tgt in tqdm(train_loader):
      src = src.to(device).T
      tgt = tgt.to(device).T

      tgt_input = tgt[:-1, :]

      src_mask, tgt_mask, src_padding_mask, tgt_padding_mask = create_mask(src, tgt_ir

      logits = model(src, tgt_input, src_mask, tgt_mask,
                                src_padding_mask, tgt_padding_mask, src_padding_mask)


      tgt_out = tgt[1:,:]
      loss = criterion(logits.reshape(-1, logits.shape[-1]), tgt_out.reshape(-1))


      total_loss += loss.item()
      total_items += (tgt_out != 0).sum(dim=(0,1))

      num_correct += (torch.logical_and((logits.argmax(dim=2) == tgt_out), (tgt_out !=
  return total_loss / len(train_loader), num_correct / total_items



def train_transformer(model, train_dataset, test_dataset, batch_size=32, epochs=60):
  train_loader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shu
  test_loader = torch.utils.data.DataLoader(test_dataset, batch_size=batch_size, shuf
  criterion = nn.CrossEntropyLoss()

  optim = torch.optim.Adam(model.parameters(), lr=1e-4, betas=(0.9, 0.98), eps=1e-9)
  for e in range(epochs):
    train_loss, train_acc = train_epoch_transformer(model, train_loader, optim, criter
    test_loss, test_acc = train_epoch_transformer(model, test_loader, optim, criterion
    print(f'Epoch: {e + 1} Training Loss: {train_loss} Training Accuracy: {train_acc}


train_transformer(model,train_dataset, test_dataset)
```

```
Epoch: 41 Training Loss: 0.10132470971037602 Training Accuracy: 0.8482346806355
100%|██████████| 29/29 [00:03<00:00,  8.60it/s]
100%|██████████| 4/4 [00:00<00:00,  9.70it/s]
Epoch: 42 Training Loss: 0.10397960245609283 Training Accuracy: 0.84427565336227
100%|██████████| 29/29 [00:02<00:00, 10.96it/s]
100%|██████████| 4/4 [00:00<00:00, 10.78it/s]
Epoch: 43 Training Loss: 0.10139972793644872 Training Accuracy: 0.84952068328857
100%|██████████| 29/29 [00:02<00:00, 10.66it/s]
100%|██████████| 4/4 [00:00<00:00, 11.15it/s]
Epoch: 44 Training Loss: 0.09904626654139881 Training Accuracy: 0.85060590505599
100%|██████████| 29/29 [00:03<00:00,  9.22it/s]
```

```
100%|██████████| 4/4 [00:00<00:00,  9.37it/s]
Epoch: 45 Training Loss: 0.08841892704367638 Training Accuracy: 0.8686923384666
100%|██████████| 29/29 [00:02<00:00, 10.91it/s]
100%|██████████| 4/4 [00:00<00:00, 11.94it/s]
Epoch: 46 Training Loss: 0.08973115194460442 Training Accuracy: 0.8601917028427
100%|██████████| 29/29 [00:02<00:00, 11.22it/s]
100%|██████████| 4/4 [00:00<00:00, 10.57it/s]
Epoch: 47 Training Loss: 0.0850652033655808 Training Accuracy: 0.864170730113983
100%|██████████| 29/29 [00:02<00:00, 11.64it/s]
100%|██████████| 4/4 [00:00<00:00, 11.26it/s]
Epoch: 48 Training Loss: 0.0819415944660532 Training Accuracy: 0.870500981807708
100%|██████████| 29/29 [00:02<00:00, 11.99it/s]
100%|██████████| 4/4 [00:00<00:00, 11.94it/s]
Epoch: 49 Training Loss: 0.07542850411143796 Training Accuracy: 0.8873214125633
100%|██████████| 29/29 [00:02<00:00, 11.78it/s]
100%|██████████| 4/4 [00:00<00:00, 12.19it/s]
Epoch: 50 Training Loss: 0.07204312381559405 Training Accuracy: 0.8934707641601
100%|██████████| 29/29 [00:02<00:00, 11.83it/s]
100%|██████████| 4/4 [00:00<00:00, 11.81it/s]
Epoch: 51 Training Loss: 0.07371213120119326 Training Accuracy: 0.8900343775749
100%|██████████| 29/29 [00:02<00:00, 12.14it/s]
100%|██████████| 4/4 [00:00<00:00, 11.42it/s]
Epoch: 52 Training Loss: 0.06792061287781288 Training Accuracy: 0.8956411480903
100%|██████████| 29/29 [00:02<00:00, 11.78it/s]
100%|██████████| 4/4 [00:00<00:00, 10.38it/s]
Epoch: 53 Training Loss: 0.06084464850096867 Training Accuracy: 0.9139084815979
100%|██████████| 29/29 [00:02<00:00, 11.98it/s]
100%|██████████| 4/4 [00:00<00:00, 12.19it/s]
Epoch: 54 Training Loss: 0.05910650589342775 Training Accuracy: 0.9128233194351
100%|██████████| 29/29 [00:02<00:00, 11.79it/s]
100%|██████████| 4/4 [00:00<00:00, 11.35it/s]
Epoch: 55 Training Loss: 0.06835682073543811 Training Accuracy: 0.9012479782104
100%|██████████| 29/29 [00:02<00:00, 11.95it/s]
100%|██████████| 4/4 [00:00<00:00, 11.76it/s]
Epoch: 56 Training Loss: 0.068217765282968 Training Accuracy: 0.8969072103500366
100%|██████████| 29/29 [00:02<00:00, 12.08it/s]
100%|██████████| 4/4 [00:00<00:00, 11.83it/s]
Epoch: 57 Training Loss: 0.057248392816761445 Training Accuracy: 0.914631962776
100%|██████████| 29/29 [00:02<00:00, 12.26it/s]
100%|██████████| 4/4 [00:00<00:00, 12.27it/s]
Epoch: 58 Training Loss: 0.049999234234464576 Training Accuracy: 0.926388144493
100%|██████████| 29/29 [00:02<00:00, 12.37it/s]
100%|██████████| 4/4 [00:00<00:00, 12.38it/s]
Epoch: 59 Training Loss: 0.04878363674827691 Training Accuracy: 0.9218665361404

100%|██████████| 29/29 [00:02<00:00, 12.31it/s]
100%|██████████| 4/4 [00:00<00:00, 12.66it/s]Epoch: 60 Training Loss: 0.0461628
```

✓ 3m 52s completed at 9:13 AM