```
# prompt: mount to google drive

from google.colab import drive
drive.mount('/content/drive')
%cd "/content/drive/MyDrive/CPEN_455/Assignment_2"
```

```
Mounted at /content/drive
/content/drive/MyDrive/CPEN_455/Assignment_2
```

```
import torch
import numpy as np
from torch.utils.data import Dataset
import torch.nn as nn
import torch.nn.functional as F
import math
```

```
class SubstringDataset(Dataset):
    LETTERS = list('cpen')

    def __init__(self, seed, dataset_size, str_len=20):
        super().__init__()
        self.str_len = str_len
        self.dataset_size = dataset_size
        self.rng = np.random.default_rng(seed)
        self.strings, self.labels = self._create_dataset()

    def __getitem__(self, index):
        return self.strings[index], self.labels[index]

    def __len__(self):
        return self.dataset_size

    def _create_dataset(self):
        strings, labels = [], []
        for i in range(self.dataset_size):
            label = i%2
            string = self._generate_random_string(bool(label))
            strings.append(string)
            labels.append(label)
        return strings, labels

    def _generate_random_string(self, has_cpen):
        while True:
            st = ''.join(self.rng.choice(SubstringDataset.LETTERS, size=self.str_len))
            if ('cpen' in st) == has_cpen:
                return st
```

```
class Tokenizer():
    def __init__(self) -> None:
        self.vocab = {
            '[CLS]': 0,
            'c': 1,
            'p': 2,
            'e': 3,
            'n': 4,
        }

    def tokenize_string(self, string, add_cls_token=True) -> torch.Tensor:
        """
        Tokenize the input string according to the above vocab

        START BLOCK
        """
        tokens=[]
        token_ids=[]

        if add_cls_token:
            tokens.append(self.vocab['[CLS]'])

        #split the string into individual characters
        tokens.extend(list(string))

        #convert each token into corresponding id using the vocab
        for token in tokens:
```

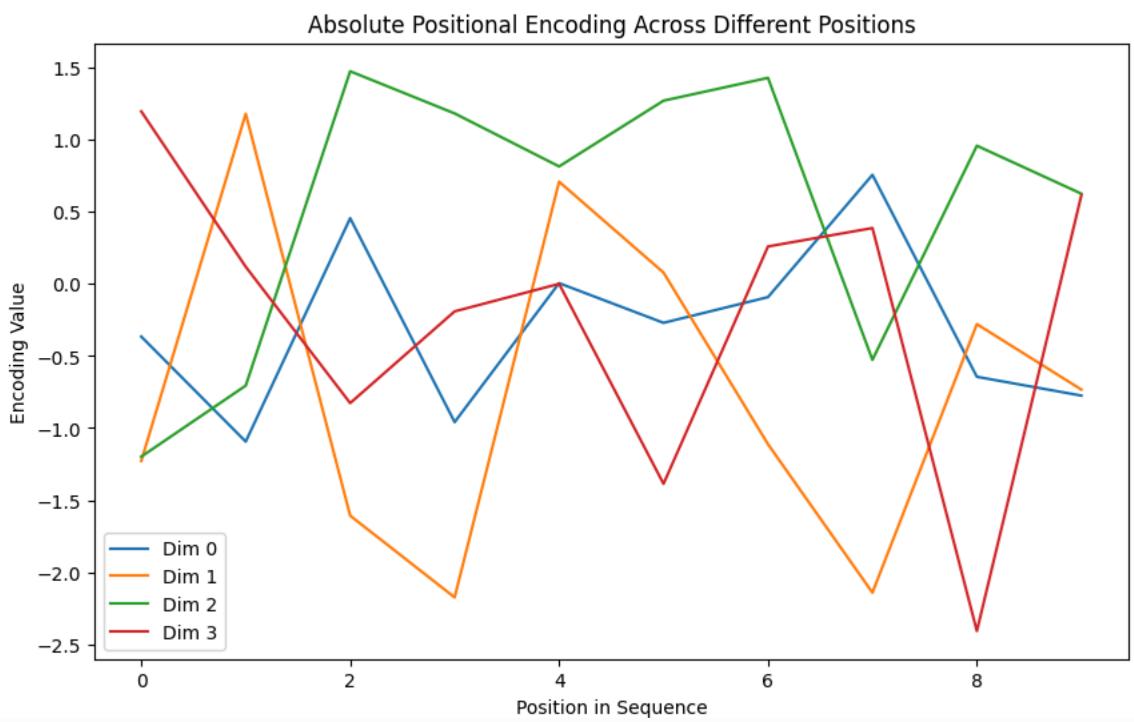
```
            if token == 0:
                token_ids.append(0)
            else:
                token_ids.append(self.vocab[token])

        # creat one hot matrix for token ids
        num_tokens=len(token_ids)
        dvoc=len(self.vocab)
        one_hot=torch.zeros(num_tokens,dvoc)

        #convert the list of tokens into one hot
        token_ids_tensor=torch.tensor(token_ids)
        one_hot=F.one_hot(token_ids_tensor,num_classes=dvoc)

        tokenized_string = one_hot
        """
        END BLOCK
        """
        return tokenized_string

    def tokenize_string_batch(self, strings, add_cls_token=True):
        X = []
        for s in strings:
            X.append(self.tokenize_string(s, add_cls_token=add_cls_token))
        return torch.stack(X, dim=0)



class AbsolutePositionalEncoding(nn.Module):
    MAX_LEN = 256
    def __init__(self, d_model):
        super().__init__()
        self.W = nn.Parameter(torch.empty((self.MAX_LEN, d_model)))
        nn.init.normal_(self.W)

    def forward(self, x):
        """
        args:
            x: shape B x N x D
        returns:
            out: shape B x N x D
        START BLOCK
        """
        #Adds the positional encoding to the input embedding X
        # X has shape batch_size, sequence_length, and d_model
        B,N,D=x.shape;
        #slicing the positonal encoding matrix, unsqueeze adds a batch dimension
        positional_encoding=self.W[:N,:].unsqueeze(0)

        out = x+positional_encoding
        """
        END BLOCK
        """
        return out

class MultiHeadAttention(nn.Module):
    MAX_LEN = 256

    def __init__(self, d_model, n_heads, rpe):
        super().__init__()
        assert d_model % n_heads == 0, "Number of heads must divide number of dimensions"
        self.n_heads = n_heads
        self.d_model = d_model
        self.d_h = d_model // n_heads
        self.rpe = rpe
        self.Wq = nn.ParameterList([nn.Parameter(torch.empty((d_model, self.d_h))) for _ in range(n_heads)])
        self.Wk = nn.ParameterList([nn.Parameter(torch.empty((d_model, self.d_h))) for _ in range(n_heads)])
        self.Wv = nn.ParameterList([nn.Parameter(torch.empty((d_model, self.d_h))) for _ in range(n_heads)])
        self.Wo = nn.Parameter(torch.empty((d_model, d_model)))

        if rpe:
            # -MAX_LEN, -MAX_LEN+1, ..., -1, 0, 1, ..., MAX_LEN-1, MAXLEN
            self.rpe_w = nn.ParameterList([nn.Parameter(torch.empty((2*self.MAX_LEN+1, ))) for _ in range(n_heads)])

        for h in range(self.n_heads):
            nn.init.xavier_normal_(self.Wk[h])
            nn.init.xavier_normal_(self.Wq[h])
            nn.init.xavier_normal_(self.Wv[h])
```

```python
            if rpe:
                nn.init.normal_(self.rpe_w[h])
        nn.init.xavier_normal_(self.Wo)

    def forward(self, key, query, value):
        """
        args:
            key: shape B x N x D
            query: shape B x N x D
            value: shape B x N x D
        return:
            out: shape B x N x D
        START BLOCK
        """
        B, N, D = query.shape
        head_outputs = []  # To collect outputs from each head

        for h in range(self.n_heads):
            # Compute per-head projections:
            Q = torch.matmul(query, self.Wq[h])  # (B, N, d_h)
            K = torch.matmul(key, self.Wk[h])      # (B, N, d_h)
            V = torch.matmul(value, self.Wv[h])    # (B, N, d_h)

            # Compute scaled dot-product scores: (B, N, N)
            scores = torch.bmm(Q, K.transpose(1, 2))

            if self.rpe:
                # Create relative position indices: shape (N, N)
                pos_indices = (
                    torch.arange(N, device=query.device).unsqueeze(0) -
                    torch.arange(N, device=query.device).unsqueeze(1)
                )
                # Shift indices to be non-negative: values in [0, 2*MAX_LEN]
                pos_indices = pos_indices + self.MAX_LEN
                # Lookup relative bias for head h: shape (N, N)
                relative_bias = self.rpe_w[h][pos_indices]
                # Expand to batch dimension and add to scores
                scores = scores + relative_bias.unsqueeze(0)

            # Scale scores
            scores = scores / (self.d_h ** 0.5)
            # Compute attention weights with softmax: shape (B, N, N)
            attn_weights = torch.softmax(scores, dim=-1)
            # Compute weighted sum of values: shape (B, N, d_h)
            head_output = torch.bmm(attn_weights, V)
            head_outputs.append(head_output)

        # Concatenate outputs from all heads: shape (B, N, d_model)
        concat = torch.cat(head_outputs, dim=-1)
        # Final linear projection: shape (B, N, d_model)
        out = torch.matmul(concat, self.Wo)

        """
        END BLOCK
        """
        return out
```

```python
# Instantiate Absolute Positional Encoding
d_model = 16  # Embedding dimension
seq_len = 10  # Sequence length
batch_size = 2  # Number of samples

# Create a dummy input tensor (random embeddings)
input_tensor = torch.randn(batch_size, seq_len, d_model)

# Initialize the Positional Encoding module
pos_encoding = AbsolutePositionalEncoding(d_model)

# Pass the input tensor through the positional encoding
output = pos_encoding(input_tensor)

# Check shape consistency
print("Input Shape:", input_tensor.shape)  # Expected: (2, 10, 16)
print("Output Shape:", output.shape)  # Expected: (2, 10, 16)
# Ensure positional encoding is being added (should be different from input)
```

```
print("Output Different from Input:", not torch.allclose(input_tensor, output))

# Extract positional encodings applied to both sequences
pos_enc_1 = output[0] - input_tensor[0]  # Encoding for first batch element
pos_enc_2 = output[1] - input_tensor[1]  # Encoding for second batch element

# Check if positional encodings across different batches are the same
print("Same Positional Encoding for All Batches:", torch.allclose(pos_enc_1, pos_enc_2))

# Create a tensor with zeros to isolate positional encoding effect
zero_tensor = torch.zeros(batch_size, seq_len, d_model)
pos_only_output = pos_encoding(zero_tensor)  # Only positional encoding remains

# Check if different positions have different encodings
pos_variation = torch.all(pos_only_output[:, 0, :] != pos_only_output[:, 1, :])
print("Different Positions Have Different Encodings:", pos_variation)
# Extract encodings for the first position across different batches
pos_0_batch_1 = pos_only_output[0, 0, :]
pos_0_batch_2 = pos_only_output[1, 0, :]

# Check if encoding for position 0 is the same across batches
print("Same Encoding for Same Position Across Batches:", torch.allclose(pos_0_batch_1, pos_0_batch_2))

import matplotlib.pyplot as plt

# Extract encodings for visualization
pos_encoding_values = pos_only_output[0].detach().numpy()  # Take first batch

# Plot the positional encoding for the first few dimensions
plt.figure(figsize=(10, 6))
for i in range(min(4, d_model)):  # Plot first 4 dimensions
    plt.plot(range(seq_len), pos_encoding_values[:, i], label=f'Dim {i}')

plt.xlabel("Position in Sequence")
plt.ylabel("Encoding Value")
plt.title("Absolute Positional Encoding Across Different Positions")
plt.legend()
plt.show()
```

```
Input Shape: torch.Size([2, 10, 16])
Output Shape: torch.Size([2, 10, 16])
Output Different from Input: True
Same Positional Encoding for All Batches: True
Different Positions Have Different Encodings: tensor(True)
Same Encoding for Same Position Across Batches: True
```



```
class TransformerLayer(nn.Module):
    def __init__(self, d_model: int, n_heads: int, prenorm: bool, rpe: bool):
        super().__init__()
        self.d_model = d_model
        self.n_heads = n_heads
        self.prenorm = prenorm
        self.attention = MultiHeadAttention(d_model, n_heads, rpe=rpe)
        self.fc_W1 = nn.Parameter(torch.empty((d_model, 4*d_model)))
        self.fc_W2 = nn.Parameter(torch.empty((4*d_model, d_model)))
        self.relu = nn.ReLU()
        self.ln1 = nn.LayerNorm(d_model)
        self.ln2 = nn.LayerNorm(d_model)

        nn.init.xavier_normal_(self.fc_W1)
        nn.init.xavier_normal_(self.fc_W2)

    def forward(self, x):
        """
        args:
            x: shape B x N x D
        returns:
            out: shape B x N x D
        START BLOCK
        """
        if self.prenorm:
            attention_input = self.ln1(x)
            attention_output = self.attention(attention_input, attention_input, attention_input)
            x = x + attention_output

            feed_forward_input = self.ln2(x)
            feed_forward_hidden = self.relu(torch.matmul(feed_forward_input, self.fc_W1))
            feed_forward_output = torch.matmul(feed_forward_hidden, self.fc_W2)
            out = x + feed_forward_output
        else:
            attention_output = self.attention(x, x, x)
            x=self.ln1(x+attention_output)

            feed_forward_hidden = self.relu(torch.matmul(x, self.fc_W1))
            feed_forward_output = torch.matmul(feed_forward_hidden, self.fc_W2)
            out = self.ln2(x + feed_forward_output)

        """
        END BLOCK
        """
        return out


class ModelConfig:
    n_layers = 4
    input_dim = 5
    d_model = 256
    n_heads = 4
    prenorm = True
    pos_enc_type = 'ape' # 'ape': Abosolute Pos. Enc., 'rpe': Relative Pos. Enc.
    output_dim = 1 # Binary output: 0: invalid, 1: valid

    def __init__(self, **kwargs):
        for k, v in kwargs.items():
            assert hasattr(self, k)
            self.__setattr__(k, v)

class TransformerModel(nn.Module):
    def __init__(self, cfg: ModelConfig):
        super().__init__()
        self.cfg = cfg
        self.enc_W = nn.Parameter(torch.empty((cfg.input_dim, cfg.d_model)))
        if cfg.pos_enc_type == 'ape':
            self.ape = AbsolutePositionalEncoding(d_model=cfg.d_model)
        self.transformer_layers = nn.ModuleList([
            TransformerLayer(d_model=cfg.d_model, n_heads=cfg.n_heads, prenorm=cfg.prenorm, rpe=cfg.pos_enc_type == 'rpe') for _ in range(cfg
        ])
        self.dec_W = nn.Parameter(torch.empty((cfg.d_model, cfg.output_dim)))

        nn.init.xavier_normal_(self.enc_W)
        nn.init.xavier_normal_(self.dec_W)

    def forward(self, x):
        """
```

```
        args:
            x: shape B x N x D_in
        returns:
            out: shape B x N x D_out
        START BLOCK
        """
        x=x.type(torch.float32)
        #encoder project the input tokens from d_voc to d_model
        x=torch.matmul(x,self.enc_W)
        #apply the absolution positional encoding
        if self.cfg.pos_enc_type == 'ape':
            x=self.ape(x)
        #apply the transformer layers
        for layer in self.transformer_layers:
            x=layer(x)

        #decoder map each token from d_model to d_out
        out = torch.matmul(x,self.dec_W)
        """
        END BLOCK
        """
        return out


from torch.optim import lr_scheduler

class CustomScheduler(lr_scheduler._LRScheduler):
    def __init__(self, optimizer, total_steps, warmup_steps=1000):
        self.total_steps = total_steps
        self.warmup_steps = warmup_steps
        super().__init__(optimizer)

    def get_lr(self):
        """
        Compute the custom scheduler with warmup and cooldown
        Hint: self.last_epoch contains the current step number
        START BLOCK
        """
        #mult_factor = 1.0
        current_step = self.last_epoch
        if current_step < self.warmup_steps:
            mult_factor = current_step / self.warmup_steps
        elif current_step <= self.total_steps:
            mult_factor = (self.total_steps-current_step) / (self.total_steps - self.warmup_steps)
        else:
            mult_factor = 0.0

        """
        END BLOCK
        """
        return [group['initial_lr'] * mult_factor for group in self.optimizer.param_groups]


import torch.optim as optim
from torch.utils.data import Dataset, DataLoader

class TrainerConfig:
    lr = 0.003
    train_steps = 5000
    batch_size = 256
    evaluate_every = 100
    device = 'cpu'

    def __init__(self, **kwargs):
        for k, v in kwargs.items():
            assert hasattr(self, k)
            self.__setattr__(k, v)

class Trainer:
    def __init__(self, model, cfg: TrainerConfig):
        self.cfg = cfg
        self.device = cfg.device
        self.tokenizer = Tokenizer()
        self.model = model.to(self.device)

    def train(self, train_dataset, val_dataset):
        optimizer = optim.Adam(self.model.parameters(), lr=self.cfg.lr)
        scheduler = CustomScheduler(optimizer, self.cfg.train_steps)
```

```
        train_dataloader = DataLoader(train_dataset, shuffle=True, batch_size=self.cfg.batch_size)
        for step in range(self.cfg.train_steps):
            self.model.train()
            batch = next(iter(train_dataloader))
            strings, y = batch
            x = self.tokenizer.tokenize_string_batch(strings)

            optimizer.zero_grad()
            loss, _ = self.compute_batch_loss_acc(x, y)
            loss.backward()
            optimizer.step()
            scheduler.step()
            if step % self.cfg.evaluate_every == 0:
                val_loss, val_acc = self.evaluate_dataset(val_dataset)
                print(f"Step {step}: Train Loss={loss.item()}, Val Loss: {val_loss}, Val Accuracy: {val_acc}")

    def compute_batch_loss_acc(self, x, y):
        """
        Compute the loss and accuracy of the model on batch (x, y)
        args:
            x: B x N x D_in
            y: B
        return:
            loss, accuracy
        START BLOCK
        """
        #forward pass through the model
        #x: (X,N,D_in) to (B,N,D_out)
        out= self.model(x.to(self.device))
        #extract the output [CLS] token
        #this has dimension of (B,D_out)
        cls_logics = out[:,0,:]
        cls_logics = cls_logics.squeeze(-1) #remove the last dimension

        loss=F.binary_cross_entropy_with_logits(cls_logics,y.float().to(self.device))
        #compute the prediciton labels, the threshold output is at 0.5
        prediction= (torch.sigmoid(cls_logics)>0.5).float()
        #calculate the accuracy by comparing prediciton with groud truth
        acc=torch.mean((prediction==y.to(self.device)).float())

        #loss, acc = torch.tensor([1.0]), torch.tensor([0.0])
        """
        END BLOCK
        """
        return loss, acc

    @torch.no_grad()
    def evaluate_dataset(self, dataset):
        self.model.eval()
        dataloader = DataLoader(dataset, shuffle=False, batch_size=self.cfg.batch_size)
        final_loss, final_acc = 0.0, 0.0
        for batch in dataloader:
            strings, y = batch
            x = self.tokenizer.tokenize_string_batch(strings)
            loss, acc = self.compute_batch_loss_acc(x, y)
            final_loss += loss.item() * x.size(0)
            final_acc += acc.item() * x.size(0)
        return final_loss / len(dataset), final_acc / len(dataset)



"""
In case you were not successful in implementing some of the above classes,
you may reimplement them using pytorch available nn Modules here to receive the marks for part 1.8
If your implementation of the previous parts is correct, leave this block empty.
START BLOCK
"""


"""
END BLOCK
"""
def run_transformer():
    device = 'cuda' if torch.cuda.is_available() else 'cpu'
    model = TransformerModel(ModelConfig())
    trainer = Trainer(model, TrainerConfig(device=device))
    parantheses_size=16
```

```
    print("Creating datasets.")
    train_dataset = SubstringDataset(seed=1, dataset_size=10_000, str_len=parantheses_size)
    val_dataset = SubstringDataset(seed=2, dataset_size=1_000, str_len=parantheses_size)
    test_dataset = SubstringDataset(seed=3, dataset_size=1_000, str_len=parantheses_size)

    print("Training the model.")
    trainer.train(train_dataset, val_dataset)
    test_loss, test_acc = trainer.evaluate_dataset(test_dataset)
    print(f"Final Test Accuracy={test_acc}, Test Loss={test_loss}")


run_transformer()
```

```
Creating datasets.
Training the model.
Step 0: Train Loss=0.8324471712112427, Val Loss: 0.8218480076789856, Val Accuracy: 0.5
Step 100: Train Loss=0.6744582653045654, Val Loss: 0.8055356850624085, Val Accuracy: 0.5
Step 200: Train Loss=0.6803422570228577, Val Loss: 0.8192860145568848, Val Accuracy: 0.5
Step 300: Train Loss=0.7267546057701111, Val Loss: 1.408002130508423, Val Accuracy: 0.5
Step 400: Train Loss=0.6992008686065674, Val Loss: 0.7272521233558655, Val Accuracy: 0.501000005722046
Step 500: Train Loss=0.6816691160202026, Val Loss: 0.6839823460578919, Val Accuracy: 0.5780000038146973
Step 600: Train Loss=0.6814118027687073, Val Loss: 0.7430353198051453, Val Accuracy: 0.5479999995231628
Step 700: Train Loss=0.4814717471599579, Val Loss: 0.4873188362121582, Val Accuracy: 0.7740000009536743
Step 800: Train Loss=0.2850826680660248, Val Loss: 0.3236431775093079, Val Accuracy: 0.8609999990463257
Step 900: Train Loss=0.1329212784767151, Val Loss: 0.2657454788684845, Val Accuracy: 0.9020000033378601
Step 1000: Train Loss=0.4263167679309845, Val Loss: 0.4200049068927765, Val Accuracy: 0.8199999971389771
Step 1100: Train Loss=0.2860735356807709, Val Loss: 0.3140209743976593, Val Accuracy: 0.873999994277954
Step 1200: Train Loss=0.4402539134025574, Val Loss: 0.4082981927394867, Val Accuracy: 0.8420000057220459
Step 1300: Train Loss=0.3677079975605011, Val Loss: 0.4811311271190643, Val Accuracy: 0.7709999985694885
Step 1400: Train Loss=0.16593897342681885, Val Loss: 0.25520105266571047, Val Accuracy: 0.892
Step 1500: Train Loss=0.14632222056388855, Val Loss: 0.19613075113296508, Val Accuracy: 0.9310000019073487
Step 1600: Train Loss=0.27466124296188354, Val Loss: 0.20735597813129425, Val Accuracy: 0.9120000009536743
Step 1700: Train Loss=0.15080130100250244, Val Loss: 0.161088765501976, Val Accuracy: 0.9399999923706055
Step 1800: Train Loss=0.17404071986675262, Val Loss: 0.10365226888656616, Val Accuracy: 0.9630000004768372
Step 1900: Train Loss=0.1321725994348526, Val Loss: 0.14862710237503052, Val Accuracy: 0.9449999995231628
Step 2000: Train Loss=0.13474540412425995, Val Loss: 0.2050581386089325, Val Accuracy: 0.9279999928474426
Step 2100: Train Loss=0.17414957284927368, Val Loss: 0.24146976828575134, Val Accuracy: 0.9039999928474426
Step 2200: Train Loss=0.051034703850746155, Val Loss: 0.07936307013034821, Val Accuracy: 0.9679999980926514
Step 2300: Train Loss=0.1000417098402977, Val Loss: 0.1328218854665756, Val Accuracy: 0.9540000028610229
Step 2400: Train Loss=0.015248378738760948, Val Loss: 0.10649180388450623, Val Accuracy: 0.9710000038146973
Step 2500: Train Loss=0.029919583350419998, Val Loss: 0.06253491657972336, Val Accuracy: 0.9790000038146973
Step 2600: Train Loss=0.022209027782082558, Val Loss: 0.09286408388614655, Val Accuracy: 0.9710000038146973
Step 2700: Train Loss=0.024518994614481926, Val Loss: 0.09522221589088439, Val Accuracy: 0.9730000038146973
Step 2800: Train Loss=0.03328895941376686, Val Loss: 0.10376494538784027, Val Accuracy: 0.9770000014305115
Step 2900: Train Loss=0.0033365427516400814, Val Loss: 0.07948715901374817, Val Accuracy: 0.9810000014305115
Step 3000: Train Loss=0.006934033706784248, Val Loss: 0.11024098479747772, Val Accuracy: 0.9810000014305115
Step 3100: Train Loss=0.002483553718775511, Val Loss: 0.09256954145431519, Val Accuracy: 0.9820000014305115
Step 3200: Train Loss=0.0011270155664533377, Val Loss: 0.06823994278907776, Val Accuracy: 0.9839999957084655
Step 3300: Train Loss=0.10717727988958359, Val Loss: 0.17250247502326965, Val Accuracy: 0.9719999933242798
Step 3400: Train Loss=0.005291105248034, Val Loss: 0.06020575249195099, Val Accuracy: 0.9869999957084655
Step 3500: Train Loss=0.0028998113702982664, Val Loss: 0.10761301279067993, Val Accuracy: 0.9799999933242798
Step 3600: Train Loss=0.02811107039451599, Val Loss: 0.08431151688098908, Val Accuracy: 0.9810000047683716
Step 3700: Train Loss=0.0215397160500288, Val Loss: 0.08907735681533814, Val Accuracy: 0.9800000014305115
Step 3800: Train Loss=0.010724999941885471, Val Loss: 0.09527887618541718, Val Accuracy: 0.9869999933242798
Step 3900: Train Loss=0.0007156722131185234, Val Loss: 0.08691400885581971, Val Accuracy: 0.9869999933242798
Step 4000: Train Loss=0.000380924524506554, Val Loss: 0.09763926112651825, Val Accuracy: 0.9869999933242798
Step 4100: Train Loss=0.00035033724270761013, Val Loss: 0.09062716436386109, Val Accuracy: 0.9869999933242798
Step 4200: Train Loss=0.0002559619606472552, Val Loss: 0.09469612550735473, Val Accuracy: 0.9879999933242798
Step 4300: Train Loss=0.0001261242723558098, Val Loss: 0.09638606250286103, Val Accuracy: 0.9859999933242798
Step 4400: Train Loss=0.00010600949462968856, Val Loss: 0.09629008078575134, Val Accuracy: 0.9859999933242798
Step 4500: Train Loss=0.00016317001427523792, Val Loss: 0.09720353150367737, Val Accuracy: 0.9869999933242798
Step 4600: Train Loss=8.905789582058787e-05, Val Loss: 0.09783836674690247, Val Accuracy: 0.9869999933242798
Step 4700: Train Loss=0.0001371238031424582, Val Loss: 0.09764800631999969, Val Accuracy: 0.9869999933242798
Step 4800: Train Loss=0.00012985234207008034, Val Loss: 0.09850402557849884, Val Accuracy: 0.9869999933242798
Step 4900: Train Loss=0.0001994653430301696, Val Loss: 0.09867901563644409, Val Accuracy: 0.9869999933242798
Final Test Accuracy=0.9879999966621399, Test Loss=0.0576315695643425
```

## Unit Tests

```
import random
import numpy as np
import torch.optim as optim

def seed_all():
    torch.manual_seed(0)
    random.seed(0)
    np.random.seed(0)

class TransformerUnitTest:
    def __init__(self, gt_vars: dict, verbose=False):
```

```
    def __init__(self, gt_vars: dict, verbose=False):
        self.gt_vars = gt_vars
        self.verbose = verbose

    def test_all(self):
        self.test_tokenizer()
        self.test_ape()
        self.test_mha()
        self.test_transformer_layer()
        self.test_transformer_model()
        self.test_scheduler()
        self.test_loss()

    def test_tokenizer(self):
        seed_all()
        self.check_correctness(
            Tokenizer().tokenize_string('ccpeen', add_cls_token=True),
            self.gt_vars['tokenizer_1'],
            "Tokenization with cls class"
        )
        self.check_correctness(
            Tokenizer().tokenize_string('cpppencpen', add_cls_token=False),
            self.gt_vars['tokenizer_2'],
            "Tokenization without cls class"
        )

    def test_ape(self):
        seed_all()
        ape_result = AbsolutePositionalEncoding(128)(torch.randn((8, 12, 128)))
        self.check_correctness(ape_result, self.gt_vars['ape'], "APE")

    def test_mha(self):
        seed_all()
        mha_result = MultiHeadAttention(d_model=128, n_heads=4, rpe=False)(
            torch.randn((8, 12, 128)), torch.randn((8, 12, 128)), torch.randn((8, 12, 128))
        )
        self.check_correctness(
            mha_result,
            self.gt_vars['mha_no_rpe'],
            "Multi-head Attention without RPE"
        )
        mha_result_rpe = MultiHeadAttention(d_model=128, n_heads=8, rpe=True)(
            torch.randn((8, 12, 128)), torch.randn((8, 12, 128)), torch.randn((8, 12, 128))
        )
        self.check_correctness(
            mha_result_rpe,
            self.gt_vars['mha_with_rpe'],
            "Multi-head Attention with RPE"
        )

    def test_transformer_layer(self):
        seed_all()
        for prenorm in [True, False]:
            transformer_layer_result = TransformerLayer(
                d_model=128, n_heads=4, prenorm=prenorm, rpe=False
            )(torch.randn((8, 12, 128)))
            self.check_correctness(
                transformer_layer_result,
                self.gt_vars[f'transformer_layer_prenorm_{prenorm}'],
                f"Transformer Layer Prenorm {prenorm}"
            )

    def test_transformer_model(self):
        seed_all()
        transformer_model_result = TransformerModel(
            ModelConfig(d_model=128, prenorm=True, pos_enc_type='ape')
        )(torch.randn((8, 12, 5)))
        self.check_correctness(
            transformer_model_result,
            self.gt_vars['transformer_model_result'],
            f"Transformer Model"
        )

    def test_scheduler(self):
        model = TransformerModel(ModelConfig())
        optimizer = optim.Adam(model.parameters(), lr=0.001)
        scheduler = CustomScheduler(optimizer, 10_000)
        optimizer.step()
```

```
        scheduler.step(521)
        self.check_correctness(
            torch.tensor([optimizer.param_groups[0]['lr']]),
            self.gt_vars['scheduler_1'],
            f"Scheduler Warmup"
        )
        scheduler.step(2503)
        self.check_correctness(
            torch.tensor([optimizer.param_groups[0]['lr']]),
            self.gt_vars['scheduler_2'],
            f"Scheduler Cooldown"
        )

    def test_loss(self):
        seed_all()
        model = TransformerModel(ModelConfig())
        trainer = Trainer(model, TrainerConfig(device='cpu'))
        loss_result, _ = trainer.compute_batch_loss_acc(
            torch.randn((8, 12, 5)),
            torch.ones(8).float(),
        )
        self.check_correctness(
            loss_result,
            self.gt_vars['loss'],
            f"Batch Loss"
        )

    def check_correctness(self, out, gt, title):
        try:
            diff = (out - gt).norm()
        except:
            diff = float('inf')
        if diff < 1e-4:
            print(f"[Correct] {title}")
        else:
            print(f"[Wrong] {title}")
            if self.verbose:
                print("-----")
                print("Expected: ")
                print(gt)
                print("Received: ")
                print(out)
                print("-----")
```

```
#!gdown 1-2-__6AALEfqhfew3sJ2QiCE1-rrFMnQ -q -O unit_tests.pkl
import pickle
with open('unit_tests.pkl', 'rb') as f:
    gt_vars = pickle.load(f)
```

```
TransformerUnitTest(gt_vars, verbose=False).test_all()
```

```
[Correct] Tokenization with cls class
[Correct] Tokenization without cls class
[Correct] APE
[Correct] Multi-head Attention without RPE
[Correct] Multi-head Attention with RPE
[Correct] Transformer Layer Prenorm True
[Correct] Transformer Layer Prenorm False
[Correct] Transformer Model
[Correct] Scheduler Warmup
[Correct] Scheduler Cooldown
[Correct] Batch Loss
/usr/local/lib/python3.11/dist-packages/torch/optim/lr_scheduler.py:240: UserWarning: The epoch parameter in `scheduler.step()` was not
  warnings.warn(EPOCH_DEPRECATION_WARNING, UserWarning)
```