

```

1 # these are some necessary libs, but feel free to import whatever you need
2 import torch
3 import torch.nn as nn
4 import torch.nn.functional as F
5 import numpy as np
6 from matplotlib import pyplot as plt
7 import math

1 ### don't change this
2 ### seed everything for reproducibility
3 def seed_everything():
4     seed = 42
5     np.random.seed(seed)
6     torch.manual_seed(seed)
7     torch.cuda.manual_seed(seed)

```

▼ 2. Prepare for the Transformer

▼ 2(a) Implement a scaled dot product

```

1 def scaled_dot_product(q, k, v):
2     d_k = q.size()[-1]
3     # TODO: put your code below
4     # values should be the final output
5     # attention should be the n by n attention matrix (the dot product after softmax)
6     scores = torch.matmul(q, k.transpose(-2, -1)) / torch.sqrt(torch.tensor(d_k, dtype=torch.float32))
7     attention = torch.softmax(scores, dim=-1)
8     values = torch.matmul(attention, v)
9     =====#
10    return values, attention

1 ### set what you get
2 ### do not modify this cell
3 seed_everything()
4 seq_len, d_k = 3, 2
5 q = torch.randn(seq_len, d_k)
6 k = torch.randn(seq_len, d_k)
7 v = torch.randn(seq_len, d_k)
8 values, attention = scaled_dot_product(q, k, v)
9 print("Q\n", q)
10 print("K\n", k)
11 print("V\n", v)
12 print("Values\n", values)
13 print("Attention\n", attention)
14 assert attention.shape == (seq_len, seq_len)

Q
tensor([[ 0.3367,  0.1288],
       [ 0.2345,  0.2303],
       [-1.1229, -0.1863]])

K
tensor([[ 2.2082, -0.6380],
       [ 0.4617,  0.2674],
       [ 0.5349,  0.8094]])

V
tensor([[ 1.1103, -1.6898],
       [-0.9890,  0.9580],
       [ 1.3221,  0.8172]])

Values
tensor([[ 0.5698, -0.1520],
       [ 0.5379, -0.0265],
       [ 0.2246,  0.5556]])

Attention
tensor([[ 0.4028,  0.2886,  0.3086],
       [ 0.3538,  0.3069,  0.3393],
       [ 0.1303,  0.4630,  0.4067]])

```

▼ 2(b) Try masked scaled-dot product

```

1 def masked_scaled_dot_product(q, k, v, mask):
2     # the mask will be in shape n by n, it indicates the interaction between specific pair of tokens need not be considered
3     d_k = q.size()[-1]
4     scores = torch.matmul(q, k.transpose(-2, -1)) / torch.sqrt(torch.tensor(d_k, dtype=torch.float32))
5     scores = scores + mask
6     attention = torch.softmax(scores, dim=-1)
7     values = torch.matmul(attention, v)
8
9     # TODO: put your code below
10    # values should be the final output
11    # attention should be the n by n attention matrix (the dot product after softmax)
12
13
14    ######
15    return values, attention

1 #### set what you get
2 #### do not modify this cell
3 seed_everything()
4 seq_len, d_k = 3, 2
5 # create a low triangular mask
6 # looks like this
7 # 1 0 0
8 # 1 1 0
9 # 1 1 1
10 # this will also be helpful for masked attention in the decoder
11 mask = torch.tril(torch.ones(seq_len, seq_len)) == 0
12 q = torch.randn(seq_len, d_k)
13 k = torch.randn(seq_len, d_k)
14 v = torch.randn(seq_len, d_k)
15 values, attention = masked_scaled_dot_product(q, k, v, mask)
16 print("Q\n", q)
17 print("K\n", k)
18 print("V\n", v)
19 print("Values\n", values)
20 print("Attention\n", attention)

Q
tensor([[ 0.3367,  0.1288],
       [ 0.2345,  0.2303],
       [-1.1229, -0.1863]])
K
tensor([[ 2.2082, -0.6380],
       [ 0.4617,  0.2674],
       [ 0.5349,  0.8094]])
V
tensor([[ 1.1103, -1.6898],
       [-0.9890,  0.9580],
       [ 1.3221,  0.8172]])
Values
tensor([[ 0.3851,  0.3733],
       [ 0.8267,  0.2842],
       [ 0.2246,  0.5556]])
Attention
tensor([[ 0.1988,  0.3872,  0.4140],
       [ 0.2235,  0.1939,  0.5827],
       [ 0.1303,  0.4630,  0.4067]])

```

▼ 2(c) Positional encoding (from original Transformer)

```

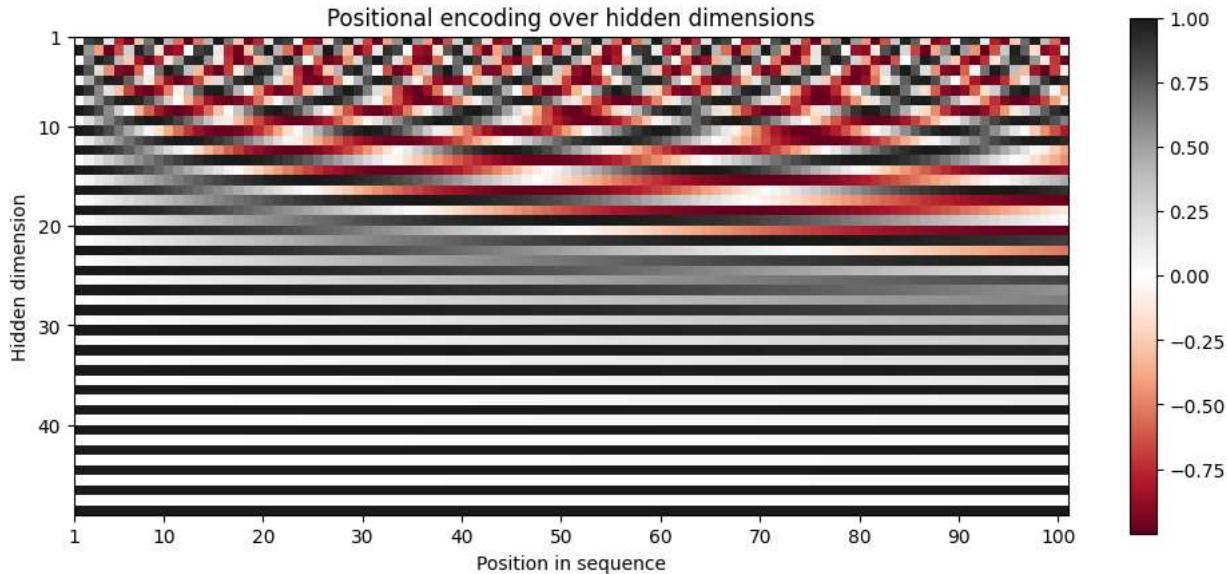
1 import torch.nn as nn
2 import torch
3
4 class PositionalEncoding(nn.Module):
5
6     def __init__(self, d_model, max_len=5000):
7         """
8             Inputs
9                 d_model - Hidden dimensionality of the input.
10                max_len - Maximum length of a sequence to expect.
11            """
12            super().__init__()
13
14            # Create matrix of [SeqLen, HiddenDim] representing the positional encoding for max_len inputs
15            pe = torch.zeros(max_len, d_model)
16            position = torch.arange(0, max_len, dtype=torch.float).unsqueeze(1)

```

```

17     div_term = torch.exp(torch.arange(0, d_model, 2).float() * (-math.log(10000.0) / d_model))
18     pe[:, 0::2] = torch.sin(position * div_term)
19     pe[:, 1::2] = torch.cos(position * div_term)
20     pe = pe.unsqueeze(0)
21
22     # register_buffer => Tensor which is not a parameter, but should be part of the modules state.
23     # Used for tensors that need to be on the same device as the module.
24     # persistent=False tells PyTorch to not add the buffer to the state dict (e.g. when we save the model)
25     self.register_buffer('pe', pe, persistent=False)
26
27     def forward(self, x):
28         x = x + self.pe[:, :x.size(1)]
29         return x
30
31
32     ### visualize the positional encoding and see what you get, do not modify this
33
34     pe_block = PositionalEncoding(d_model=48, max_len=100)
35     pe = pe_block.pe.squeeze().T.cpu().numpy()
36
37     fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(12,5))
38     pos = ax.imshow(pe, cmap="RdGy", extent=(1,pe.shape[1]+1,pe.shape[0]+1,1))
39     fig.colorbar(pos, ax=ax)
40     ax.set_xlabel("Position in sequence")
41     ax.set_ylabel("Hidden dimension")
42     ax.set_title("Positional encoding over hidden dimensions")
43     ax.set_xticks([1]+[i*10 for i in range(1,1+pe.shape[1]//10)])
44     ax.set_yticks([1]+[i*10 for i in range(1,1+pe.shape[0]//10)])
45     plt.show()

```



▼ 3. Build your own "GPT" for 1D Burgers' prediction

```

1     ### load data, modify this when you have a different path
2     train_data = np.load('/content/burgers_train.npy')
3     test_data = np.load('/content/burgers_test.npy')
4     print(train_data.shape, test_data.shape)

(2048, 40, 100) (128, 40, 100)

```

▼ Visualize a sequence (optional)

```

1     ### You don't have to modify this function
2     from matplotlib.ticker import FormatStrFormatter
3
4     def show_field(field):
5
6         fig, ax = plt.subplots(figsize=(15, 5))
7         # mark y axis as time, x axis as space

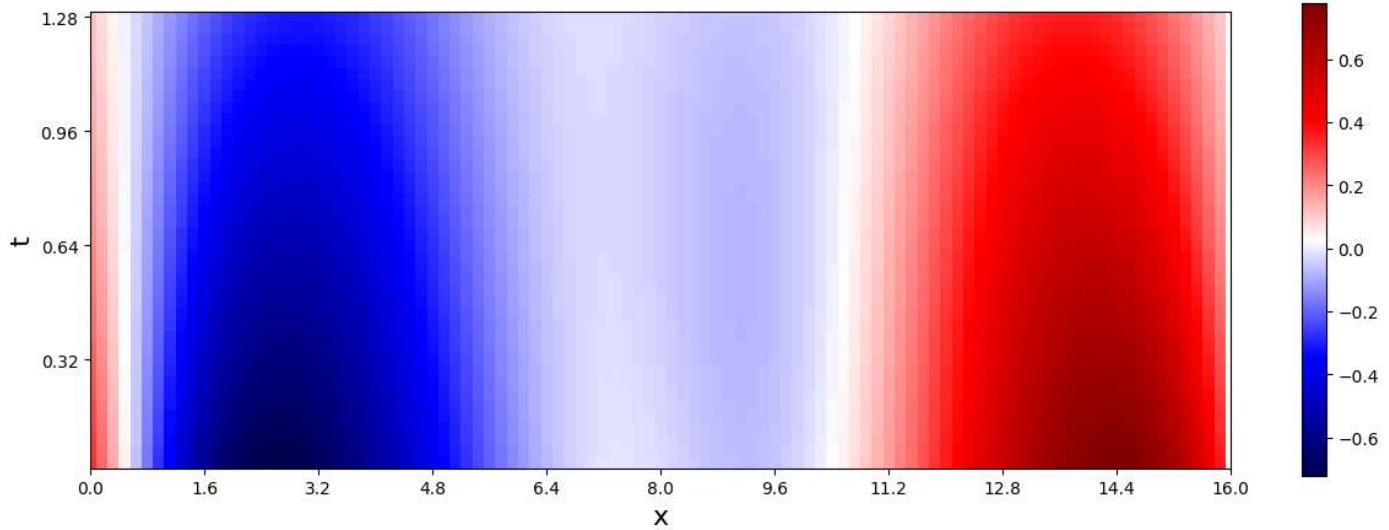
```

```

8     ax.set_xlabel('x', fontsize=16)
9     ax.set_ylabel('t', fontsize=16)
10    ax.set_xlim(0, 50+1e-5)
11
12    ax.set_yticks(np.arange(0, 41, 10), [str(round(f, 2)) for f in np.linspace(0, 1.28, 5)][::-1])
13    ax.set_xticks(np.arange(0, 101, 10), [str(round(f, 2)) for f in np.linspace(0, 16, 11)])
14
15    im = ax.imshow(field, cmap='seismic')
16    plt.colorbar(im)
17
18    plt.show()

```

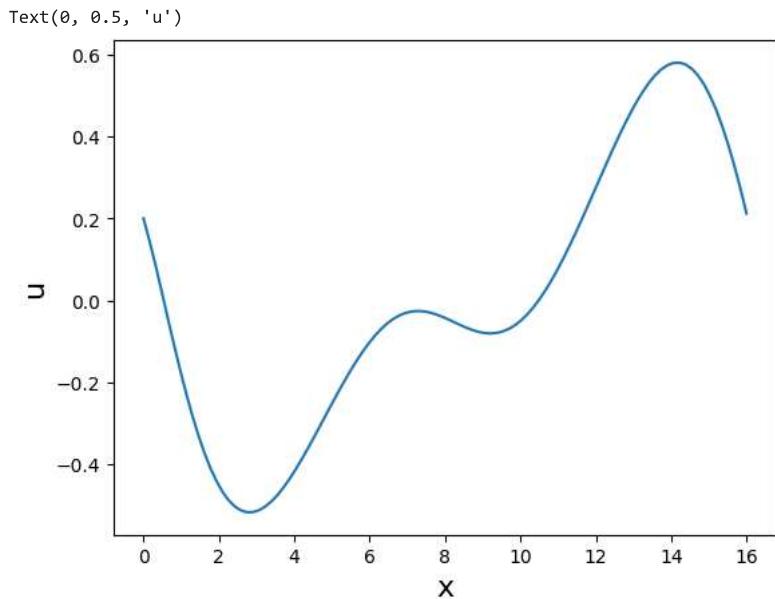
```
1 show_field(train_data[0])
```



```

1 # take a look at one of the snapshots
2 plt.plot(np.linspace(0, 16, 100), train_data[0, 20])
3 plt.gca().set_xlabel('x', fontsize=16)
4 plt.gca().set_ylabel('u', fontsize=16)

```



▼ Helper functions (do not modify!)

```

1 ### do not modify this cell
2 =====
3 class SimpleEncoder(nn.Module):

```

```

4     # for embedding the first 10 time steps of darcy flow
5     def __init__(self,
6         input_dim=100,      # we take number of grid points as number of feature for each frame
7         hidden_dim=64,    # hidden dimensionality of the encoder
8         emb_dim=128):
9         super().__init__()
10        # we use 1d depth-wise convolution to embed the input
11        self.net = nn.Sequential(
12            nn.Conv1d(input_dim, hidden_dim, kernel_size=1, stride=1, padding=0, groups=4),
13            nn.GELU(),    # GELU is a better version of ReLU, for most tasks
14            nn.Conv1d(hidden_dim, hidden_dim, kernel_size=1, stride=1, padding=0, groups=8),
15            nn.GELU(),
16            nn.Conv1d(hidden_dim, hidden_dim, kernel_size=1, stride=1, padding=0, groups=4),
17            nn.GELU(),
18            nn.Conv1d(hidden_dim, hidden_dim, kernel_size=1, stride=1, padding=0),
19            nn.GELU(),
20            nn.Conv1d(hidden_dim, emb_dim, kernel_size=1, stride=1, padding=0))
21
22        self.to_out = nn.Sequential(
23            nn.LayerNorm(emb_dim),
24            nn.Linear(emb_dim, emb_dim),
25        )
26
27    def forward(self, x):
28        # x will be in shape [b t n], n is number of grid points, but you can think it as feature dimension
29        # we need to transpose it to [b n t] for the convolution
30        x = x.transpose(1, 2)
31        # now we can apply the convolution
32        x = self.net(x)
33        x = x.transpose(1, 2)
34        return self.to_out(x)
35
36
37 class SimpleDecoder(nn.Module):
38     # go back from the latent space to the physical space
39     def __init__(self,
40         input_dim=128,      # latent dimensionality
41         hidden_dim=64,    # hidden dimensionality of the decoder
42         output_dim=100):
43         super().__init__()
44         self.ln = nn.LayerNorm(input_dim)
45         self.net = nn.Sequential(
46             nn.Conv1d(input_dim, hidden_dim, kernel_size=1, stride=1, padding=0),
47             nn.GELU(),
48             nn.Conv1d(hidden_dim, hidden_dim, kernel_size=1, stride=1, padding=0, groups=4),
49             nn.GELU(),
50             nn.Conv1d(hidden_dim, output_dim, kernel_size=1, stride=1, padding=0, groups=2),
51         )
52
53    def forward(self, x):
54        # x will be in shape [b t c]
55        x = self.ln(x)
56        x = x.transpose(1, 2)
57        x = self.net(x).transpose(1, 2)  # [b t c] again
58        return x
59
60
61
62 class FFN(nn.Module):
63     # give to you for free, need to be used in the transformer
64     def __init__(self, input_dim=128, hidden_dim=128, output_dim=128):
65         super().__init__()
66         self.net = nn.Sequential(
67             nn.LayerNorm(input_dim),
68             nn.Linear(input_dim, hidden_dim),
69             nn.GELU(),
70             nn.Linear(hidden_dim, output_dim),
71         )
72
73    def forward(self, x):
74        return self.net(x)

```

▼ Implement a multi-head self-attention module (with causal mask)

```
1 # you need to implement this
2 import torch.nn as nn
3 import torch
4
5 class CausalSelfAttention(nn.Module):
6
7     def __init__(self,
8                  dim,
9                  dim_head,
10                 num_heads,
11                 dropout,          # dropout for attention matrix ( $Q^T K$ ), not input or output
12                 max_len=50
13                 ):
14         super().__init__()
15         self.dim = dim
16         self.inner_dim = dim_head * num_heads
17         self.num_heads = num_heads
18         self.dim_head = dim_head
19         self.max_len = max_len
20
21         # Create the linear layers for projecting the input into queries, keys, and values
22         self.query = nn.Linear(dim, self.inner_dim)
23         self.key = nn.Linear(dim, self.inner_dim)
24         self.value = nn.Linear(dim, self.inner_dim)
25
26         # Create the output projection layer
27         self.proj = nn.Linear(self.inner_dim, dim)
28
29         # Create a mask to prevent attention beyond the current position
30         self.register_buffer('mask', torch.tril(torch.ones(max_len, max_len), diagonal=-1))
31         self.dropout = nn.Dropout(dropout)
32
33     def _init_weight(self):
34
35         nn.init.xavier_uniform_(self.query.weight)
36         self.query.bias.data.fill_(0)
37         nn.init.xavier_uniform_(self.key.weight)
38         self.key.bias.data.fill_(0)
39         nn.init.xavier_uniform_(self.value.weight)
40         self.value.bias.data.fill_(0)
41         nn.init.xavier_uniform_(self.proj.weight)
42         self.proj.bias.data.fill_(0)
43
44     def forward(self, x):
45         # Input is in shape [B, L, C], output is also in shape [B, L, C]
46         B, L, C = x.size()
47
48         # Project the input into queries, keys, and values using the linear layers
49         q = self.query(x).view(B, L, self.num_heads, self.dim_head).transpose(1, 2)
50         k = self.key(x).view(B, L, self.num_heads, self.dim_head).transpose(1, 2)
51         v = self.value(x).view(B, L, self.num_heads, self.dim_head).transpose(1, 2)
52
53         values, attention = masked_scaled_dot_product(q,k,v, self.mask[:L,:L])
54         ...
55
56         # Compute the dot product of queries and keys for each head
57         dot_products = torch.matmul(q, k.transpose(-2, -1)) / torch.sqrt(torch.tensor(self.dim_head, dtype=torch.float32).to(x.device))
58
59         # Apply the mask to prevent attention beyond the current position
60         mask = self.mask[:L, :L].unsqueeze(0).unsqueeze(0).to(x.device)
61         dot_products = dot_products.masked_fill(mask == 0, float('-inf'))
62
63         # Apply the softmax function to get the attention weights for each head
64         attention_weights = torch.softmax(dot_products, dim=-1)
65
66         # Apply dropout to the attention weights
67         attention_weights = self.dropout(attention_weights)
68
69         # Compute the weighted sum of values using the attention weights for each head
70         values = torch.matmul(attention_weights, v)
71         ...
72
73         attention = self.dropout(attention)
74         values = torch.matmul(attention,v)
75         values = values.transpose(1, 2).contiguous().view(B, L, self.inner_dim)
76
77         # Project the concatenated values using the output projection layer
78         output = self.proj(values)
```

```

78     return output
79

```

▼ Use above attention block to build a PDE-GPT

```

1 ##### you only have to fill a small part inside the forward function
2 class PDEGPT(nn.Module):
3     def __init__(self,
4                  num_layers,      # the only hyperparameter to play with, could start with sth like 6
5                  ):
6         super().__init__()
7
8         self.transformer = nn.ModuleList([])
9         for _ in range(num_layers):
10             self.transformer.append(nn.ModuleList([
11                 nn.LayerNorm(128),
12                 CausalSelfAttention(dim=128, dim_head=128, num_heads=4, dropout=0.05),
13                 FFN(input_dim=128, hidden_dim=128, output_dim=128),
14             ]))
15         self.function_encoder = SimpleEncoder()
16         self.function_decoder = SimpleDecoder()
17         self.position_embedding = PositionalEncoding(128, 40)
18
19         # report number of parameters
20         print(f"Total number of trainable parameters: {self.get_num_params()}" )
21
22     def get_num_params(self):
23         n_params = sum(p.numel() for p in self.parameters() if p.requires_grad)
24         return n_params
25
26     def forward(self, seq, noise=True, tstart=0):
27         # seq in shape [batch_size, 49, 100] t=49, n=100, think about why it is 49 not 50
28
29         b, t, n = seq.size()
30         device = seq.device
31
32         # forward the GPT model itself
33         if noise:
34             # add random walk noise to the input
35             seq = seq + torch.cumsum(torch.randn_like(seq) * 0.003, dim=1)
36             x = self.function_encoder(seq)    # [b t n] -> [b t c]
37             x = x + self.position_embedding(x) # add position embedding
38
39             for ln, attn_block, ffn in self.transformer:
40                 # ln: layer normalization
41                 # attn_block: self attention layer
42                 # ffn: feed forward network (a two-layer MLP)
43                 # =====
44
45
46                 x_attn = attn_block(x)
47                 x = x + x_attn
48                 x_ln = ln(x)
49                 x_ffn = ffn(x_ln)
50
51
52                 x = x + x_ffn
53                 x = ln(x)
54                 # =====
55
56
57             x = self.function_decoder(x)    # [b t c] -> [b t n]
58
59             return x
60
61
62     def predict(self, in_seq, predict_steps=30, tstart=0):
63         # in_seq will be in shape [batch_size, 10, 100]
64         out_seq = torch.zeros(in_seq.size(0), predict_steps, in_seq.size(2)).to(in_seq.device)
65         for t in range(predict_steps):
66             if self.training:
67                 shifted_in_seq = self.forward(in_seq, noise=True, tstart=tstart)
68             else:
69                 shifted_in_seq = self.forward(in_seq, noise=False, tstart=tstart)
70             in_seq = torch.cat((in_seq, shifted_in_seq[:, -1:]), dim=1)

```

```

71         out_seq[:, t:t+1] = shifted_in_seq[:, -1:]
72
73     return out_seq

```

▼ Train the model

```

1 ##### do not modify this cell!
2 =====
3 def train_a_gpt(lr,
4                 num_layers,
5                 batch_size,
6                 num_epochs):
7     device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
8     print('Using device:', device)
9     # train a GPT model with num_layers
10    model = PDEGPT(num_layers=num_layers)
11    model = model.to(device)
12
13    # build dataloader
14    train_set = torch.utils.data.TensorDataset(torch.tensor(train_data, dtype=torch.float32))
15    train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True, num_workers=4)
16
17    test_set = torch.utils.data.TensorDataset(torch.tensor(test_data, dtype=torch.float32))
18    test_loader = torch.utils.data.DataLoader(test_set, batch_size=8, shuffle=False, num_workers=4)
19
20    optimizer = torch.optim.AdamW(model.parameters(), lr=lr)
21    scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=lr, total_steps=num_epochs*len(train_loader), final_div_factor=1e3)
22
23    train_history = []
24    test_history = []
25    loss_buffer = []
26    for epoch in range(num_epochs):
27        model.train()
28        for i, seq in enumerate(train_loader):
29            seq = seq[0].to(device)
30            optimizer.zero_grad()
31            # training in block, not always using teacher forcing
32            if i % 3 == 0:
33                loss = 0
34                ct = 0
35                for b in range(0, seq.size(1)-20, 10):
36                    pred = model.predict(seq[:, b:b+10], predict_steps=10, tstart=b)
37                    loss += F.mse_loss(pred, seq[:, b+10:b+20])
38                    ct += 1
39                loss /= ct
40            else:
41                pred = model(seq[:, :-1])
42                loss = F.mse_loss(pred, seq[:, 1:])
43                loss.backward()
44                optimizer.step()
45                loss_buffer.append(loss.item())
46
47            if i % 20 == 0:
48                print("epoch %d, iter %d, loss %.3f" % (epoch, i, np.mean(loss_buffer)))
49                loss_buffer = []
50            train_history.append(loss.item())
51            scheduler.step()
52
53        model.eval()
54        test_losses = []
55        with torch.no_grad():
56            for i, seq in enumerate(test_loader):
57                seq = seq[0].to(device)
58                pred = model.predict(seq[:, :10])
59                loss = F.mse_loss(pred, seq[:, 10:])
60
61                test_losses.append(loss.item())
62        print("epoch %d, test loss %.3f" % (epoch, np.mean(test_losses)))
63        test_history.append(np.mean(test_losses))
64
65    return model, train_history, test_history
66
67
68 def eval_and_visualize(trained_model):
69     device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

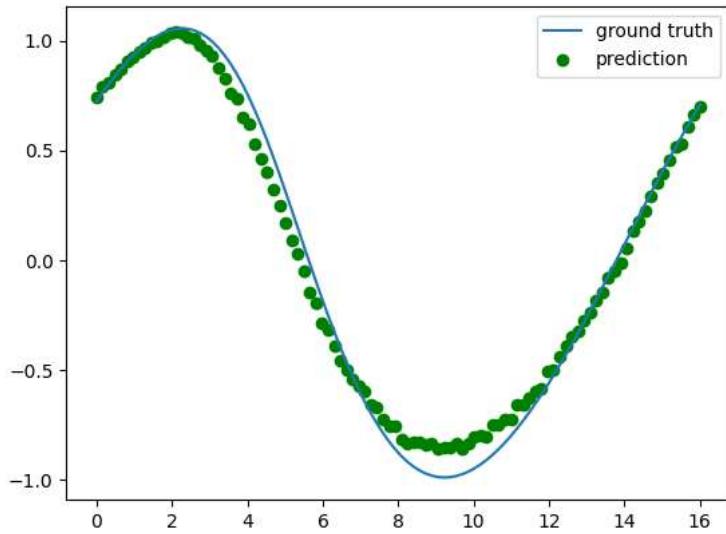
```

```
70 print('Using device:', device)
71 test_set = torch.utils.data.TensorDataset(torch.tensor(test_data, dtype=torch.float32))
72 test_loader = torch.utils.data.DataLoader(test_set, batch_size=8, shuffle=False, num_workers=4)
73
74 trained_model.eval()
75 test_losses = []
76 with torch.no_grad():
77     for i, seq in enumerate(test_loader):
78         seq = seq[0].to(device)
79         pred = trained_model.predict(seq[:, :10])
80         loss = F.mse_loss(pred, seq[:, 10:])
81
82         test_losses.append(loss.item())
83
84 print('Final evaluation error:', np.mean(test_losses))
85
86 # randomly pick a sample to evaluate and visualize
87 seed_everything()
88 idx = np.random.randint(0, len(test_data))
89 seq = torch.tensor(test_data[idx:idx+1], dtype=torch.float32).to(device)
90 pred = trained_model.predict(seq[:, :10])
91 pred_seq = torch.cat((seq[:, :10], pred), dim=1).cpu().numpy()[0]
92
93 plt.plot(np.linspace(0, 16, 100), seq.cpu().numpy()[0, -1], label='ground truth')
94 plt.scatter(np.linspace(0, 16, 100), pred_seq[-1], label='prediction', c='g')
95 plt.legend()
96 plt.show()
97
98 return
```

1 ### you can change the hyperparameter of "train_a_gpt" to see how it affects the performance, please do not modify the other parts
2 #####
3
4 torch.cuda.empty_cache()
5 # change the parameters to see how it affects the performance
6 model, train_history, test_history = train_a_gpt(lr=1e-3, num_layers=6, batch_size=32, num_epochs=200)
7 eval_and_visualize(model)
8
9 plt.plot(train_history, label='train loss') # simply visualize the training loss
10 plt.plot(test_history, label='test loss')
11 plt.legend()



```
epoch 198, iter 60, loss 0.004
epoch 198, test loss 0.018
epoch 199, iter 0, loss 0.004
epoch 199, iter 20, loss 0.005
epoch 199, iter 40, loss 0.003
epoch 199, iter 60, loss 0.004
epoch 199, test loss 0.016
Using device: cuda:0
Final evaluation error: 0.015520559885771945
```



```
<matplotlib.legend.Legend at 0x7fbc6804f7f0>
```

