In [1]:
```
# these are some necessary libs, but feel free to import whatever you need
import torch
import torch.nn as nn
import torch.nn.functional as F
import numpy as np
from matplotlib import pyplot as plt
import math
```

In [2]:
```
### don't change this
### seed everything for reproducibility
def seed_everything():
    seed = 42
    np.random.seed(seed)
    torch.manual_seed(seed)
    torch.cuda.manual_seed(seed)
```

2. Prepare for the Transformer

2(a) Implement a scaled dot product

In [3]:
```
def scaled_dot_product(q, k, v):
    d_k = q.size()[-1]

    # TODO: put your code below
    # values should be the final output
    # attention should be the n by n attention matrix (the dot product after softmax)


    scores = torch.matmul(q, k.transpose(-2, -1))
    scores = scores/math.sqrt(d_k)

    attention = F.softmax(scores, dim=-1)

    values = torch.matmul(attention, v)

    #========================#
    return values, attention
```

In [4]:
```
### set what you get
### do not modify this cell
seed_everything()
seq_len, d_k = 3, 2
q = torch.randn(seq_len, d_k)
k = torch.randn(seq_len, d_k)
v = torch.randn(seq_len, d_k)
values, attention = scaled_dot_product(q, k, v)
print("Q\n", q)
print("K\n", k)
print("V\n", v)
print("Values\n", values)
print("Attention\n", attention)
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

In [5]:
```python
def masked_scaled_dot_product(q, k, v, mask):
    # the mask will be in shape n by n, it indicates the interaction between specific pair of tokens need not be considered
    d_k = q.size()[-1]

    # TODO: put your code below
    # values should be the final output
    # attention should be the n by n attention matrix (the dot product after softmax)

    scores = torch.matmul(q, k.transpose(-2, -1))
    scores = scores/math.sqrt(d_k)

    attention = F.softmax(scores.masked_fill_(mask, 1e-9), dim=-1)

    values = torch.matmul(attention, v)

    #=======================#
    return values, attention
```

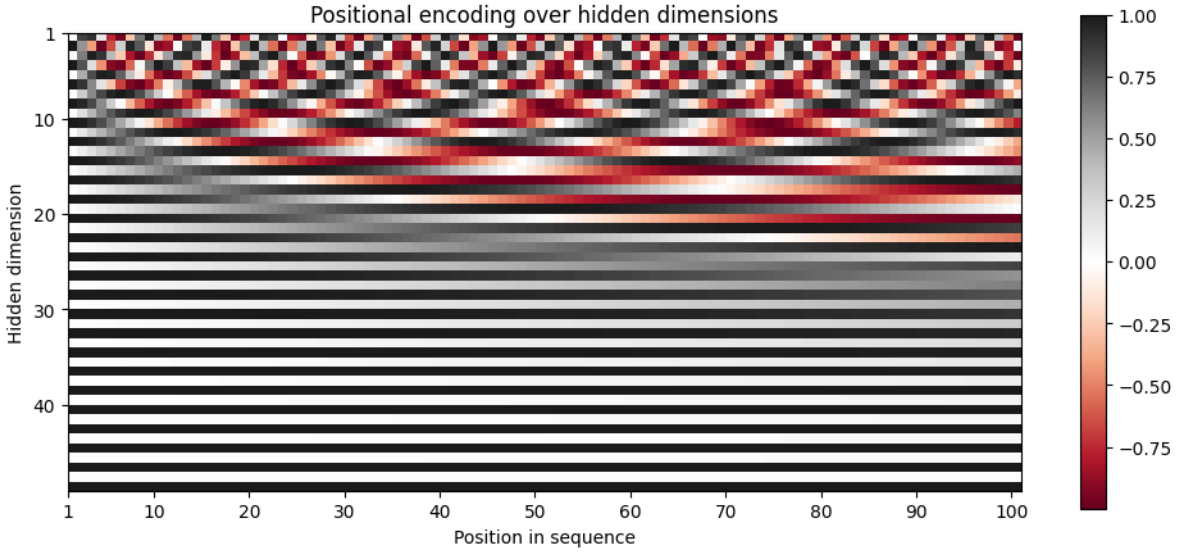
In [6]:
```python
### set what you get
### do not modify this cell
seed_everything()
seq_len, d_k = 3, 2
# create a low triangular mask
# looks like this
# 1 0 0
# 1 1 0
# 1 1 1
# this will also be helpful for masked attention in the decoder
mask = torch.tril(torch.ones(seq_len, seq_len)) == 0
q = torch.randn(seq_len, d_k)
k = torch.randn(seq_len, d_k)
v = torch.randn(seq_len, d_k)
values, attention = masked_scaled_dot_product(q, k, v, mask)
print("Q\n", q)
print("K\n", k)
print("V\n", v)
print("Values\n", values)
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([[ 0.5855, -0.2564],
        [ 0.4815, -0.0872],
        [ 0.2246,  0.5556]])
Attention
 tensor([[0.4439, 0.2781, 0.2781],
        [0.3792, 0.3290, 0.2918],
        [0.1303, 0.4630, 0.4067]])
```

2(c) Positional encoding (from original Transformer)

In [7]:
```python
class PositionalEncoding(nn.Module):

    def __init__(self, d_model, max_len=5000):
        """
        Inputs
            d_model - Hidden dimensionality of the input.
            max_len - Maximum length of a sequence to expect.
        """
        super().__init__()

        # Create matrix of [SeqLen, HiddenDim] representing the positional encoding for max_len inputs
        pe = torch.zeros(max_len, d_model)

        # ================================== #
        # TODO: put your code below
        pos = torch.arange(0, max_len, dtype=torch.float).unsqueeze(dim=1)

        i = torch.arange(0, d_model, step=2, dtype=torch.float)

        pe[:, 0::2] = torch.sin(pos / (10000 ** (i/d_model)))
        pe[:, 1::2] = torch.cos(pos / (10000 ** (i/d_model)))
        # ================================== #
        # register_buffer => Tensor which is not a parameter, but should be part of the modules state.
        # Used for tensors that need to be on the same device as the module.
        # persistent=False tells PyTorch to not add the buffer to the state dict (e.g. when we save the model)
        self.register_buffer('pe', pe, persistent=False)

    def forward(self, x):
        pe = self.pe[:x.size(1), :]

        #TODO: put your code below

        return pe
```

In [8]:
```
### visualize the positional encoding and see what you get, do not modify this

pe_block = PositionalEncoding(d_model=48, max_len=100)
pe = pe_block.pe.squeeze().T.cpu().numpy()

fig, ax = plt.subplots(nrows=1, ncols=1, figsize=(12,5))
pos = ax.imshow(pe, cmap="RdGy", extent=(1,pe.shape[1]+1,pe.shape[0]+1,1))
fig.colorbar(pos, ax=ax)
ax.set_xlabel("Position in sequence")
ax.set_ylabel("Hidden dimension")
ax.set_title("Positional encoding over hidden dimensions")
ax.set_xticks([1]+[i*10 for i in range(1,1+pe.shape[1]//10)])
ax.set_yticks([1]+[i*10 for i in range(1,1+pe.shape[0]//10)])
plt.show()
```



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

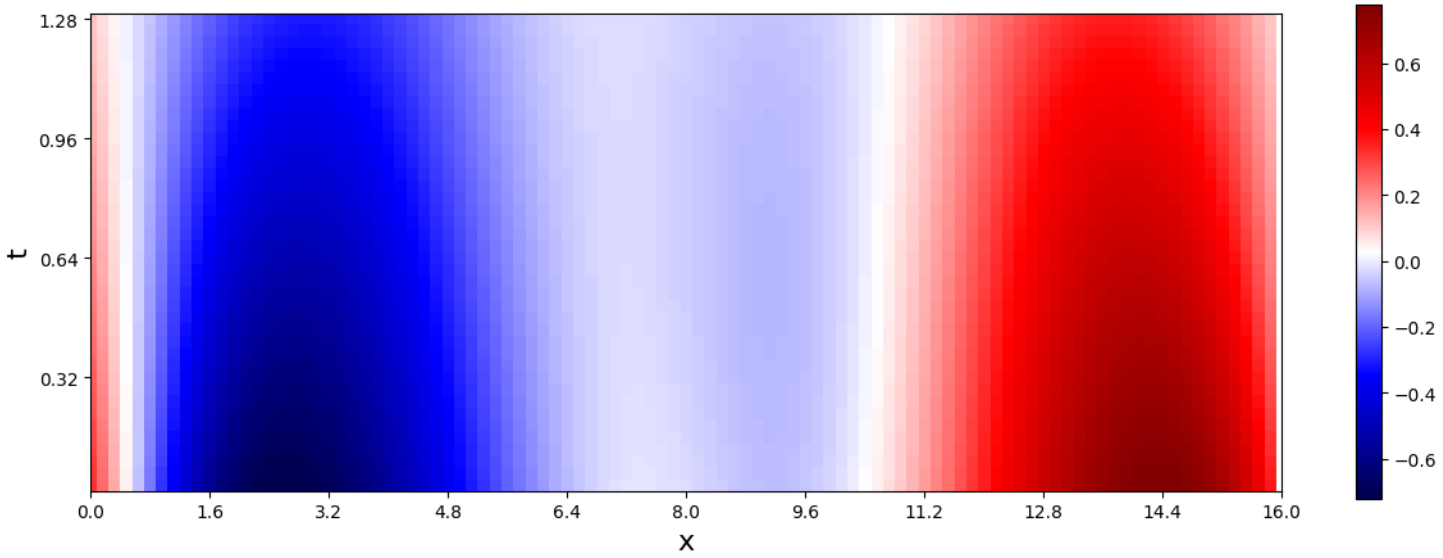
In [9]:
```
from google.colab import drive
drive.mount('/content/drive')

data_path = '/content/drive/MyDrive/24788-24789/24789-hw4-attention/'

### load data, modify this when you have a different path
train_data = np.load(data_path+'burgers_train.npy')
test_data = np.load(data_path+'burgers_test.npy')
print(train_data.shape, test_data.shape)
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, call drive.mount("/content/drive", force_remount=True).
(2048, 40, 100) (128, 40, 100)
```

Visualize a sequence (optional)

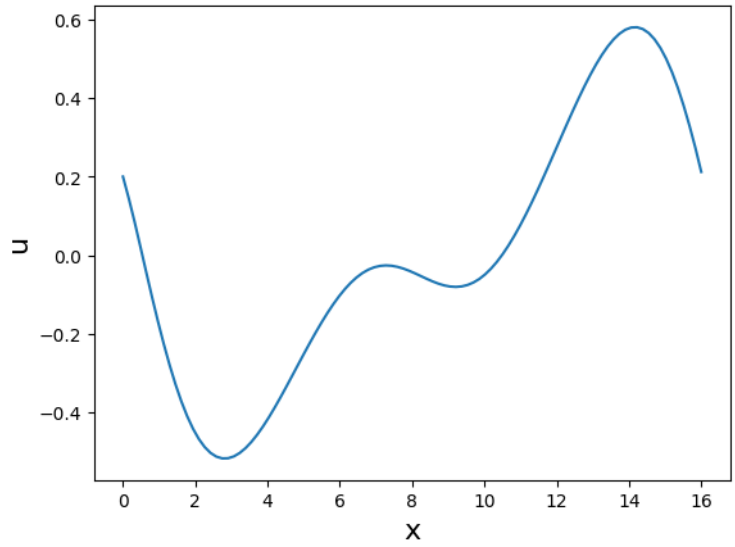
In [10]:
```
### You don't have to modify this function
from matplotlib.ticker import FormatStrFormatter

def show_field(field):

    fig, ax = plt.subplots(figsize=(15, 5))
    # mark y axis as time, x axis as space
    ax.set_xlabel('x',fontsize=16)
    ax.set_ylabel('t', fontsize=16)
    ax.set_xlim(0, 50+1e-5)

    ax.set_yticks(np.arange(0, 41, 10), [str(round(f, 2)) for f in np.linspace(0, 1.28, 5)][::-1])
    ax.set_xticks(np.arange(0, 101, 10),[str(round(f, 2)) for f in np.linspace(0, 16, 11)])

    im = ax.imshow(field,cmap='seismic')
    plt.colorbar(im)

    plt.show()
```

In [11]:
```
show_field(train_data[0])
```



In [12]:
```
# take a look at one of the snapshots
plt.plot(np.linspace(0, 16, 100), train_data[0, 20])
plt.gca().set_xlabel('x',fontsize=16)
plt.gca().set_ylabel('u', fontsize=16)
```

Out[12]: `Text(0, 0.5, 'u')`



Helper functions (do not modify!)

In [13]:
```python
### do not modify this cell
### ======================
class SimpleEncoder(nn.Module):
    # for embedding the first 10 time steps of darcy flow
    def __init__(self,
                 input_dim=100,    # we take number of grid points as number of feature for each frame
                 hidden_dim=64,   # hidden dimensionality of the encoder
                 emb_dim=128):
        super().__init__()
        # we use 1d depth-wise convolution to embed the input
        self.net = nn.Sequential(
            nn.Conv1d(input_dim, hidden_dim, kernel_size=1, stride=1, padding=0, groups=4),
            nn.GELU(),   # GELU is a better version of ReLU, for most tasks
            nn.Conv1d(hidden_dim, hidden_dim, kernel_size=1, stride=1, padding=0, groups=8),
            nn.GELU(),
            nn.Conv1d(hidden_dim, hidden_dim, kernel_size=1, stride=1, padding=0, groups=4),
            nn.GELU(),
            nn.Conv1d(hidden_dim, hidden_dim, kernel_size=1, stride=1, padding=0),
            nn.GELU(),
            nn.Conv1d(hidden_dim, emb_dim, kernel_size=1, stride=1, padding=0))

        self.to_out = nn.Sequential(
            nn.LayerNorm(emb_dim),
            nn.Linear(emb_dim, emb_dim),
        )

    def forward(self, x):
        # x will be in shape [b t n], n is number of grid points, but you can think it as feature dimension
        # we need to transpose it to [b n t] for the convolution
        x = x.transpose(1, 2)
        # now we can apply the convolution
        x = self.net(x)
        x = x.transpose(1, 2)
        return self.to_out(x)


class SimpleDecoder(nn.Module):
    # go back from the latent space to the physical space
    def __init__(self,
                 input_dim=128,     # latent dimensionality
                 hidden_dim=64,   # hidden dimensionality of the decoder
                 output_dim=100):
        super().__init__()
        self.ln = nn.LayerNorm(input_dim)
        self.net = nn.Sequential(
            nn.Conv1d(input_dim, hidden_dim, kernel_size=1, stride=1, padding=0),
            nn.GELU(),
            nn.Conv1d(hidden_dim, hidden_dim, kernel_size=1, stride=1, padding=0, groups=4),
            nn.GELU(),
            nn.Conv1d(hidden_dim, output_dim, kernel_size=1, stride=1, padding=0, groups=2),
        )

    def forward(self, x):
        # x will be in shape [b t c]
        x = self.ln(x)
        x = x.transpose(1, 2)
        x =  self.net(x).transpose(1, 2)  # [b t c] again
        return x



class FFN(nn.Module):
    # give to you for free, need to be used in the transformer
    def __init__(self, input_dim=128, hidden_dim=128, output_dim=128):
        super().__init__()
        self.net = nn.Sequential(
            nn.LayerNorm(input_dim),
            nn.Linear(input_dim, hidden_dim),
            nn.GELU(),
            nn.Linear(hidden_dim, output_dim),
        )

    def forward(self, x):
        return self.net(x)
```

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

In [14]:
```python
# you need to implement this
class CausalSelfAttention(nn.Module):

    def __init__(self,
                 dim,
                 dim_head,
                 num_heads,
                 dropout,       # dropout for attention matrix (Q^T K), not input or output
                 max_len=50
                 ):
        super().__init__()
        self.dim = dim
        self.inner_dim = dim_head * num_heads
        self.num_heads = num_heads
        self.dim_head = dim_head

        mask = torch.tril(torch.ones(max_len, max_len)) == 0
        self.mask = mask.cuda()
        # ================================================================
        #TODO: implement the module of the causal self attention
        self.w_qkv = nn.Linear(self.dim, 3*self.inner_dim)
        self.w_o = nn.Linear(self.inner_dim, self.dim)

        self.dropout = nn.Dropout(dropout)




        # ================================================================
        self._init_weights()

    def _init_weights(self):
        # you can implement some weight initialization here (optional), can provide some performance boost when doing it correctly
        nn.init.xavier_uniform_(self.w_qkv.weight)
        self.w_qkv.bias.data.fill_(0)

        nn.init.xavier_uniform_(self.w_o.weight)
        self.w_o.bias.data.fill_(0)

    def forward(self, x):
        # input is in shape [b t c], output is also in shape [b t c]
        B, L, C = x.size() # batch size, sequence length, embedding dimensionality (n_embd)

        # ================================================================
        #TODO: implement the forward pass of the causal self attention

        qkv = self.w_qkv(x)
        qkv = qkv.reshape(B, L, self.num_heads, 3*self.dim_head) # Split split dimension into each head
        qkv = qkv.permute(0, 2, 1, 3) # [batch size, number of heads, sequence length, head dimensions]
        q, k, v = qkv.chunk(3, dim=-1) # separate query, key and value matrices

        _, attention = masked_scaled_dot_product(q, k, v, self.mask[:L,:L])

        attention = self.dropout(attention)
        values = torch.matmul(attention, v)

        values = values.permute(0, 2, 1, 3) # [batch size, sequence length, number of heads, head dimensions]
        values = values.reshape(B, L, self.inner_dim) # [batch size, sequence length, inner dimension]

        out = self.w_o(values)

        return out
```

Use above attention block to build a PDE-GPT

In [15]:
```python
### you only have to fill a small part inside the forward function
class PDEGPT(nn.Module):
    def __init__(self,
                 num_layers,    # the only hyperparameter to play with, could start with sth like 6
                 ):
        super().__init__()

        self.transformer = nn.ModuleList([])
        for _ in range(num_layers):
            self.transformer.append(nn.ModuleList([
                nn.LayerNorm(128),
                CausalSelfAttention(dim=128, dim_head=128, num_heads=4, dropout=0.05),
                FFN(input_dim=128, hidden_dim=128, output_dim=128),
            ]))
        self.function_encoder = SimpleEncoder()
        self.function_decoder = SimpleDecoder()
        self.position_embedding =  PositionalEncoding(128, 40)

        # report number of parameters
        print(f"Total number of trainable parameters: {self.get_num_params()}" )

    def get_num_params(self):
        n_params = sum(p.numel() for p in self.parameters() if p.requires_grad)
        return n_params

    def forward(self, seq, noise=True, tstart=0):
        # seq in shape [batch_size, 49, 100] t=49, n=100, think about why it is 49 not 50

        b, t, n = seq.size()
        device = seq.device

        # forward the GPT model itself
        if noise:
            # add random walk noise to the input
            seq = seq + torch.cumsum(torch.randn_like(seq) * 0.003, dim=1)
        x = self.function_encoder(seq)   # [b t n] -> [b t c]
        x = x + self.position_embedding(x) # add position embedding

        for ln, attn_block, ffn in self.transformer:
            # ln: layer normalization
            # attn_block: self attention layer
            # ffn: feed forward network (a two-layer MLP)
            # ====================================================
            # TODO: implement one forward pass of the transformer (do not modify other part of this class)

            # attention
            attn_out = attn_block(x)
            x = x + attn_out
            x = ln(x)

            # feed-forward
            ffn_out = ffn(x)
            x = x + ffn_out
            x = ln(x)


            # ====================================================

        x = self.function_decoder(x)   # [b t c] -> [b t n]

        return x

    def predict(self, in_seq, predict_steps=30, tstart=0):
        # in_seq will be in shape [batch_size, 10, 100]
        out_seq = torch.zeros(in_seq.size(0), predict_steps, in_seq.size(2)).to(in_seq.device)
        for t in range(predict_steps):
            if self.training:
                shifted_in_seq = self.forward(in_seq, noise=True, tstart=tstart)
            else:
                shifted_in_seq = self.forward(in_seq, noise=False, tstart=tstart)
            in_seq = torch.cat((in_seq, shifted_in_seq[:, -1:]), dim=1)
            out_seq[:, t:t+1] = shifted_in_seq[:, -1:]

        return out_seq
```

Train the model

In [16]:

```
### do not modify this cell!
### ==============================
def train_a_gpt(lr,
                num_layers,
                batch_size,
                num_epochs):
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print('Using device:', device)
    # train a GPT model with num_layers
    model = PDEGPT(num_layers=num_layers)
    model = model.to(device)

    # build dataloader
    train_set = torch.utils.data.TensorDataset(torch.tensor(train_data, dtype=torch.float32))
    train_loader = torch.utils.data.DataLoader(train_set, batch_size=batch_size, shuffle=True, num_workers=4)

    test_set = torch.utils.data.TensorDataset(torch.tensor(test_data, dtype=torch.float32))
    test_loader = torch.utils.data.DataLoader(test_set, batch_size=8, shuffle=False, num_workers=4)

    optimizer = torch.optim.AdamW(model.parameters(), lr=lr)
    scheduler = torch.optim.lr_scheduler.OneCycleLR(optimizer, max_lr=lr, total_steps=num_epochs*len(train_loader), final_div_factor=1e3)

    train_history = []
    test_history = []
    loss_buffer = []
    for epoch in range(num_epochs):
        model.train()
        for i, seq in enumerate(train_loader):
            seq = seq[0].to(device)
            optimizer.zero_grad()
            # training in block, not always using teacher forcing
            if i % 3 == 0:
                loss = 0
                ct = 0
                for b in range(0, seq.size(1)-20, 10):
                    pred = model.predict(seq[:, b:b+10], predict_steps=10, tstart=b)
                    loss += F.mse_loss(pred, seq[:, b+10:b+20])
                    ct += 1
                loss /= ct
            else:
                pred = model(seq[:, :-1])
                loss = F.mse_loss(pred, seq[:, 1:])
            loss.backward()
            optimizer.step()
            loss_buffer.append(loss.item())

            if i % 20 == 0:
                print("epoch %d, iter %d, loss %.3f" % (epoch, i, np.mean(loss_buffer)))
                loss_buffer = []
        train_history.append(loss.item())
        scheduler.step()

        model.eval()
        test_losses = []
        with torch.no_grad():
            for i, seq in enumerate(test_loader):
                seq = seq[0].to(device)
                pred = model.predict(seq[:, :10])
                loss = F.mse_loss(pred, seq[:, 10:])

                test_losses.append(loss.item())
        print("epoch %d, test loss %.3f" % (epoch, np.mean(test_losses)))
        test_history.append(np.mean(test_losses))

    return model, train_history, test_history


def eval_and_visualize(trained_model):
    device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
    print('Using device:', device)
    test_set = torch.utils.data.TensorDataset(torch.tensor(test_data, dtype=torch.float32))
    test_loader = torch.utils.data.DataLoader(test_set, batch_size=8, shuffle=False, num_workers=4)

    trained_model.eval()
    test_losses = []
    with torch.no_grad():
        for i, seq in enumerate(test_loader):
            seq = seq[0].to(device)
            pred = trained_model.predict(seq[:, :10])
            loss = F.mse_loss(pred, seq[:, 10:])

            test_losses.append(loss.item())

        print('Final evaluation error:', np.mean(test_losses))

        # randomly pick a sample to evaluate and visualize
        seed_everything()
        idx = np.random.randint(0, len(test_data))
        seq = torch.tensor(test_data[idx:idx+1], dtype=torch.float32).to(device)
        pred = trained_model.predict(seq[:, :10])
        pred_seq = torch.cat((seq[:, :10], pred), dim=1).cpu().numpy()[0]

        plt.plot(np.linspace(0, 16, 100), seq.cpu().numpy()[0, -1], label='ground truth')
        plt.scatter(np.linspace(0, 16, 100), pred_seq[-1], label='prediction', c='g')
        plt.legend()
        plt.show()
```

```
    return
```

In [17]:
```
### you can change the hyperparameter of "train_a_gpt" to see how it affects the performance, please do not modify the other parts
### ==============================

torch.cuda.empty_cache()
# change the parameters to see how it affects the performance
model, train_history, test_history = train_a_gpt(lr=1.05e-3, num_layers=10, batch_size=32, num_epochs=150)
eval_and_visualize(model)

plt.plot(train_history, label='train loss')  # simply visualize the training loss
plt.plot(test_history, label='test loss')
plt.legend()
```

```
Using device: cuda:0
Total number of trainable parameters: 3018916
```

```
/usr/local/lib/python3.9/dist-packages/torch/utils/data/dataloader.py:561: UserWarning: This DataLoader will create 4 worker processes in total. O
ur suggested max number of worker in current system is 2, which is smaller than what this DataLoader is going to create. Please be aware that exce
ssive worker creation might get DataLoader running slow or even freeze, lower the worker number to avoid potential slowness/freeze if necessary.
  warnings.warn(_create_warning_msg(
```

```
epoch 0, iter 0, loss 0.451
epoch 0, iter 20, loss 0.548
epoch 0, iter 40, loss 0.561
epoch 0, iter 60, loss 0.551
epoch 0, test loss 0.642
epoch 1, iter 0, loss 0.586
epoch 1, iter 20, loss 0.554
epoch 1, iter 40, loss 0.535
epoch 1, iter 60, loss 0.533
epoch 1, test loss 0.617
epoch 2, iter 0, loss 0.574
epoch 2, iter 20, loss 0.517
epoch 2, iter 40, loss 0.500
epoch 2, iter 60, loss 0.512
epoch 2, test loss 0.586
epoch 3, iter 0, loss 0.448
epoch 3, iter 20, loss 0.491
epoch 3, iter 40, loss 0.487
epoch 3, iter 60, loss 0.466
epoch 3, test loss 0.554
epoch 4, iter 0, loss 0.477
epoch 4, iter 20, loss 0.458
epoch 4, iter 40, loss 0.444
epoch 4, iter 60, loss 0.447
epoch 4, test loss 0.515
epoch 5, iter 0, loss 0.391
epoch 5, iter 20, loss 0.428
epoch 5, iter 40, loss 0.409
epoch 5, iter 60, loss 0.397
epoch 5, test loss 0.481
epoch 6, iter 0, loss 0.421
epoch 6, iter 20, loss 0.395
epoch 6, iter 40, loss 0.379
epoch 6, iter 60, loss 0.359
epoch 6, test loss 0.446
epoch 7, iter 0, loss 0.402
epoch 7, iter 20, loss 0.360
epoch 7, iter 40, loss 0.339
epoch 7, iter 60, loss 0.342
epoch 7, test loss 0.409
epoch 8, iter 0, loss 0.332
epoch 8, iter 20, loss 0.325
epoch 8, iter 40, loss 0.322
epoch 8, iter 60, loss 0.299
epoch 8, test loss 0.376
epoch 9, iter 0, loss 0.328
epoch 9, iter 20, loss 0.290
epoch 9, iter 40, loss 0.299
epoch 9, iter 60, loss 0.279
epoch 9, test loss 0.342
epoch 10, iter 0, loss 0.251
epoch 10, iter 20, loss 0.277
epoch 10, iter 40, loss 0.254
epoch 10, iter 60, loss 0.243
epoch 10, test loss 0.310
epoch 11, iter 0, loss 0.262
epoch 11, iter 20, loss 0.235
epoch 11, iter 40, loss 0.227
epoch 11, iter 60, loss 0.232
epoch 11, test loss 0.279
epoch 12, iter 0, loss 0.206
epoch 12, iter 20, loss 0.218
epoch 12, iter 40, loss 0.210
epoch 12, iter 60, loss 0.196
epoch 12, test loss 0.249
epoch 13, iter 0, loss 0.182
epoch 13, iter 20, loss 0.189
epoch 13, iter 40, loss 0.192
epoch 13, iter 60, loss 0.163
epoch 13, test loss 0.224
epoch 14, iter 0, loss 0.175
epoch 14, iter 20, loss 0.171
epoch 14, iter 40, loss 0.158
epoch 14, iter 60, loss 0.161
epoch 14, test loss 0.202
epoch 15, iter 0, loss 0.153
epoch 15, iter 20, loss 0.148
epoch 15, iter 40, loss 0.141
epoch 15, iter 60, loss 0.134
epoch 15, test loss 0.179
epoch 16, iter 0, loss 0.139
epoch 16, iter 20, loss 0.131
epoch 16, iter 40, loss 0.128
epoch 16, iter 60, loss 0.119
epoch 16, test loss 0.160
epoch 17, iter 0, loss 0.100
epoch 17, iter 20, loss 0.112
epoch 17, iter 40, loss 0.110
epoch 17, iter 60, loss 0.105
epoch 17, test loss 0.143
epoch 18, iter 0, loss 0.103
epoch 18, iter 20, loss 0.105
epoch 18, iter 40, loss 0.092
epoch 18, iter 60, loss 0.091
epoch 18, test loss 0.127
epoch 19, iter 0, loss 0.091
epoch 19, iter 20, loss 0.084
```

```
epoch 19, iter 40, loss 0.088
epoch 19, iter 60, loss 0.080
epoch 19, test loss 0.114
epoch 20, iter 0, loss 0.081
epoch 20, iter 20, loss 0.078
epoch 20, iter 40, loss 0.078
epoch 20, iter 60, loss 0.071
epoch 20, test loss 0.104
epoch 21, iter 0, loss 0.074
epoch 21, iter 20, loss 0.072
epoch 21, iter 40, loss 0.064
epoch 21, iter 60, loss 0.062
epoch 21, test loss 0.093
epoch 22, iter 0, loss 0.072
epoch 22, iter 20, loss 0.058
epoch 22, iter 40, loss 0.063
epoch 22, iter 60, loss 0.061
epoch 22, test loss 0.086
epoch 23, iter 0, loss 0.048
epoch 23, iter 20, loss 0.056
epoch 23, iter 40, loss 0.056
epoch 23, iter 60, loss 0.051
epoch 23, test loss 0.077
epoch 24, iter 0, loss 0.050
epoch 24, iter 20, loss 0.048
epoch 24, iter 40, loss 0.050
epoch 24, iter 60, loss 0.050
epoch 24, test loss 0.072
epoch 25, iter 0, loss 0.052
epoch 25, iter 20, loss 0.046
epoch 25, iter 40, loss 0.047
epoch 25, iter 60, loss 0.040
epoch 25, test loss 0.067
epoch 26, iter 0, loss 0.042
epoch 26, iter 20, loss 0.044
epoch 26, iter 40, loss 0.039
epoch 26, iter 60, loss 0.042
epoch 26, test loss 0.067
epoch 27, iter 0, loss 0.047
epoch 27, iter 20, loss 0.041
epoch 27, iter 40, loss 0.039
epoch 27, iter 60, loss 0.037
epoch 27, test loss 0.058
epoch 28, iter 0, loss 0.034
epoch 28, iter 20, loss 0.037
epoch 28, iter 40, loss 0.037
epoch 28, iter 60, loss 0.034
epoch 28, test loss 0.054
epoch 29, iter 0, loss 0.029
epoch 29, iter 20, loss 0.035
epoch 29, iter 40, loss 0.030
epoch 29, iter 60, loss 0.037
epoch 29, test loss 0.051
epoch 30, iter 0, loss 0.027
epoch 30, iter 20, loss 0.031
epoch 30, iter 40, loss 0.030
epoch 30, iter 60, loss 0.031
epoch 30, test loss 0.049
epoch 31, iter 0, loss 0.030
epoch 31, iter 20, loss 0.030
epoch 31, iter 40, loss 0.034
epoch 31, iter 60, loss 0.028
epoch 31, test loss 0.048
epoch 32, iter 0, loss 0.019
epoch 32, iter 20, loss 0.031
epoch 32, iter 40, loss 0.026
epoch 32, iter 60, loss 0.028
epoch 32, test loss 0.045
epoch 33, iter 0, loss 0.027
epoch 33, iter 20, loss 0.025
epoch 33, iter 40, loss 0.026
epoch 33, iter 60, loss 0.027
epoch 33, test loss 0.044
epoch 34, iter 0, loss 0.026
epoch 34, iter 20, loss 0.027
epoch 34, iter 40, loss 0.025
epoch 34, iter 60, loss 0.024
epoch 34, test loss 0.044
epoch 35, iter 0, loss 0.024
epoch 35, iter 20, loss 0.026
epoch 35, iter 40, loss 0.024
epoch 35, iter 60, loss 0.024
epoch 35, test loss 0.043
epoch 36, iter 0, loss 0.019
epoch 36, iter 20, loss 0.025
epoch 36, iter 40, loss 0.021
epoch 36, iter 60, loss 0.024
epoch 36, test loss 0.037
epoch 37, iter 0, loss 0.021
epoch 37, iter 20, loss 0.024
epoch 37, iter 40, loss 0.021
epoch 37, iter 60, loss 0.022
epoch 37, test loss 0.035
epoch 38, iter 0, loss 0.018
epoch 38, iter 20, loss 0.020
epoch 38, iter 40, loss 0.021
epoch 38, iter 60, loss 0.021
```

```
epoch 38, test loss 0.034
epoch 39, iter 0, loss 0.024
epoch 39, iter 20, loss 0.020
epoch 39, iter 40, loss 0.018
epoch 39, iter 60, loss 0.023
epoch 39, test loss 0.034
epoch 40, iter 0, loss 0.015
epoch 40, iter 20, loss 0.020
epoch 40, iter 40, loss 0.020
epoch 40, iter 60, loss 0.020
epoch 40, test loss 0.033
epoch 41, iter 0, loss 0.017
epoch 41, iter 20, loss 0.022
epoch 41, iter 40, loss 0.019
epoch 41, iter 60, loss 0.017
epoch 41, test loss 0.032
epoch 42, iter 0, loss 0.011
epoch 42, iter 20, loss 0.019
epoch 42, iter 40, loss 0.020
epoch 42, iter 60, loss 0.017
epoch 42, test loss 0.032
epoch 43, iter 0, loss 0.017
epoch 43, iter 20, loss 0.019
epoch 43, iter 40, loss 0.021
epoch 43, iter 60, loss 0.018
epoch 43, test loss 0.031
epoch 44, iter 0, loss 0.023
epoch 44, iter 20, loss 0.020
epoch 44, iter 40, loss 0.018
epoch 44, iter 60, loss 0.016
epoch 44, test loss 0.033
epoch 45, iter 0, loss 0.013
epoch 45, iter 20, loss 0.017
epoch 45, iter 40, loss 0.016
epoch 45, iter 60, loss 0.019
epoch 45, test loss 0.029
epoch 46, iter 0, loss 0.015
epoch 46, iter 20, loss 0.016
epoch 46, iter 40, loss 0.016
epoch 46, iter 60, loss 0.017
epoch 46, test loss 0.028
epoch 47, iter 0, loss 0.014
epoch 47, iter 20, loss 0.016
epoch 47, iter 40, loss 0.017
epoch 47, iter 60, loss 0.016
epoch 47, test loss 0.029
epoch 48, iter 0, loss 0.012
epoch 48, iter 20, loss 0.017
epoch 48, iter 40, loss 0.015
epoch 48, iter 60, loss 0.016
epoch 48, test loss 0.027
epoch 49, iter 0, loss 0.016
epoch 49, iter 20, loss 0.015
epoch 49, iter 40, loss 0.016
epoch 49, iter 60, loss 0.015
epoch 49, test loss 0.028
epoch 50, iter 0, loss 0.014
epoch 50, iter 20, loss 0.016
epoch 50, iter 40, loss 0.016
epoch 50, iter 60, loss 0.014
epoch 50, test loss 0.027
epoch 51, iter 0, loss 0.015
epoch 51, iter 20, loss 0.014
epoch 51, iter 40, loss 0.014
epoch 51, iter 60, loss 0.015
epoch 51, test loss 0.026
epoch 52, iter 0, loss 0.015
epoch 52, iter 20, loss 0.016
epoch 52, iter 40, loss 0.014
epoch 52, iter 60, loss 0.013
epoch 52, test loss 0.025
epoch 53, iter 0, loss 0.013
epoch 53, iter 20, loss 0.014
epoch 53, iter 40, loss 0.014
epoch 53, iter 60, loss 0.014
epoch 53, test loss 0.026
epoch 54, iter 0, loss 0.016
epoch 54, iter 20, loss 0.015
epoch 54, iter 40, loss 0.014
epoch 54, iter 60, loss 0.014
epoch 54, test loss 0.025
epoch 55, iter 0, loss 0.009
epoch 55, iter 20, loss 0.015
epoch 55, iter 40, loss 0.015
epoch 55, iter 60, loss 0.013
epoch 55, test loss 0.024
epoch 56, iter 0, loss 0.011
epoch 56, iter 20, loss 0.016
epoch 56, iter 40, loss 0.013
epoch 56, iter 60, loss 0.013
epoch 56, test loss 0.024
epoch 57, iter 0, loss 0.011
epoch 57, iter 20, loss 0.014
epoch 57, iter 40, loss 0.013
epoch 57, iter 60, loss 0.015
epoch 57, test loss 0.025
epoch 58, iter 0, loss 0.011
```

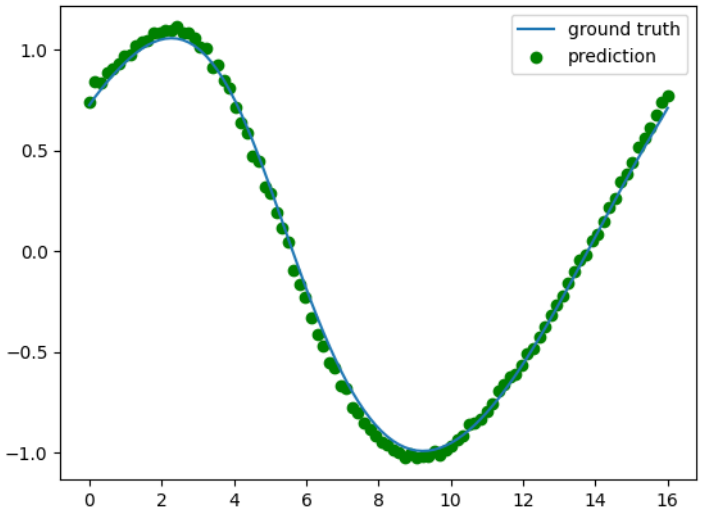
```
epoch 58, iter 20, loss 0.014
epoch 58, iter 40, loss 0.013
epoch 58, iter 60, loss 0.012
epoch 58, test loss 0.024
epoch 59, iter 0, loss 0.013
epoch 59, iter 20, loss 0.013
epoch 59, iter 40, loss 0.015
epoch 59, iter 60, loss 0.012
epoch 59, test loss 0.024
epoch 60, iter 0, loss 0.014
epoch 60, iter 20, loss 0.012
epoch 60, iter 40, loss 0.013
epoch 60, iter 60, loss 0.013
epoch 60, test loss 0.022
epoch 61, iter 0, loss 0.013
epoch 61, iter 20, loss 0.012
epoch 61, iter 40, loss 0.012
epoch 61, iter 60, loss 0.014
epoch 61, test loss 0.025
epoch 62, iter 0, loss 0.008
epoch 62, iter 20, loss 0.013
epoch 62, iter 40, loss 0.014
epoch 62, iter 60, loss 0.013
epoch 62, test loss 0.023
epoch 63, iter 0, loss 0.010
epoch 63, iter 20, loss 0.011
epoch 63, iter 40, loss 0.012
epoch 63, iter 60, loss 0.013
epoch 63, test loss 0.023
epoch 64, iter 0, loss 0.015
epoch 64, iter 20, loss 0.012
epoch 64, iter 40, loss 0.011
epoch 64, iter 60, loss 0.013
epoch 64, test loss 0.021
epoch 65, iter 0, loss 0.011
epoch 65, iter 20, loss 0.012
epoch 65, iter 40, loss 0.013
epoch 65, iter 60, loss 0.010
epoch 65, test loss 0.021
epoch 66, iter 0, loss 0.013
epoch 66, iter 20, loss 0.013
epoch 66, iter 40, loss 0.010
epoch 66, iter 60, loss 0.012
epoch 66, test loss 0.023
epoch 67, iter 0, loss 0.011
epoch 67, iter 20, loss 0.012
epoch 67, iter 40, loss 0.012
epoch 67, iter 60, loss 0.012
epoch 67, test loss 0.021
epoch 68, iter 0, loss 0.009
epoch 68, iter 20, loss 0.011
epoch 68, iter 40, loss 0.011
epoch 68, iter 60, loss 0.012
epoch 68, test loss 0.021
epoch 69, iter 0, loss 0.011
epoch 69, iter 20, loss 0.012
epoch 69, iter 40, loss 0.011
epoch 69, iter 60, loss 0.012
epoch 69, test loss 0.023
epoch 70, iter 0, loss 0.012
epoch 70, iter 20, loss 0.012
epoch 70, iter 40, loss 0.012
epoch 70, iter 60, loss 0.011
epoch 70, test loss 0.019
epoch 71, iter 0, loss 0.010
epoch 71, iter 20, loss 0.012
epoch 71, iter 40, loss 0.011
epoch 71, iter 60, loss 0.011
epoch 71, test loss 0.019
epoch 72, iter 0, loss 0.011
epoch 72, iter 20, loss 0.012
epoch 72, iter 40, loss 0.011
epoch 72, iter 60, loss 0.010
epoch 72, test loss 0.018
epoch 73, iter 0, loss 0.010
epoch 73, iter 20, loss 0.011
epoch 73, iter 40, loss 0.010
epoch 73, iter 60, loss 0.012
epoch 73, test loss 0.020
epoch 74, iter 0, loss 0.008
epoch 74, iter 20, loss 0.010
epoch 74, iter 40, loss 0.010
epoch 74, iter 60, loss 0.011
epoch 74, test loss 0.019
epoch 75, iter 0, loss 0.011
epoch 75, iter 20, loss 0.010
epoch 75, iter 40, loss 0.012
epoch 75, iter 60, loss 0.011
epoch 75, test loss 0.020
epoch 76, iter 0, loss 0.007
epoch 76, iter 20, loss 0.011
epoch 76, iter 40, loss 0.011
epoch 76, iter 60, loss 0.010
epoch 76, test loss 0.019
epoch 77, iter 0, loss 0.008
epoch 77, iter 20, loss 0.010
epoch 77, iter 40, loss 0.010
```

```
epoch 77, iter 60, loss 0.010
epoch 77, test loss 0.020
epoch 78, iter 0, loss 0.008
epoch 78, iter 20, loss 0.010
epoch 78, iter 40, loss 0.009
epoch 78, iter 60, loss 0.011
epoch 78, test loss 0.018
epoch 79, iter 0, loss 0.009
epoch 79, iter 20, loss 0.011
epoch 79, iter 40, loss 0.010
epoch 79, iter 60, loss 0.009
epoch 79, test loss 0.019
epoch 80, iter 0, loss 0.010
epoch 80, iter 20, loss 0.009
epoch 80, iter 40, loss 0.010
epoch 80, iter 60, loss 0.012
epoch 80, test loss 0.019
epoch 81, iter 0, loss 0.009
epoch 81, iter 20, loss 0.010
epoch 81, iter 40, loss 0.009
epoch 81, iter 60, loss 0.010
epoch 81, test loss 0.018
epoch 82, iter 0, loss 0.010
epoch 82, iter 20, loss 0.011
epoch 82, iter 40, loss 0.009
epoch 82, iter 60, loss 0.009
epoch 82, test loss 0.020
epoch 83, iter 0, loss 0.008
epoch 83, iter 20, loss 0.010
epoch 83, iter 40, loss 0.010
epoch 83, iter 60, loss 0.009
epoch 83, test loss 0.017
epoch 84, iter 0, loss 0.008
epoch 84, iter 20, loss 0.010
epoch 84, iter 40, loss 0.011
epoch 84, iter 60, loss 0.011
epoch 84, test loss 0.019
epoch 85, iter 0, loss 0.009
epoch 85, iter 20, loss 0.010
epoch 85, iter 40, loss 0.009
epoch 85, iter 60, loss 0.009
epoch 85, test loss 0.018
epoch 86, iter 0, loss 0.010
epoch 86, iter 20, loss 0.010
epoch 86, iter 40, loss 0.010
epoch 86, iter 60, loss 0.010
epoch 86, test loss 0.019
epoch 87, iter 0, loss 0.006
epoch 87, iter 20, loss 0.009
epoch 87, iter 40, loss 0.008
epoch 87, iter 60, loss 0.010
epoch 87, test loss 0.018
epoch 88, iter 0, loss 0.013
epoch 88, iter 20, loss 0.010
epoch 88, iter 40, loss 0.010
epoch 88, iter 60, loss 0.008
epoch 88, test loss 0.017
epoch 89, iter 0, loss 0.009
epoch 89, iter 20, loss 0.008
epoch 89, iter 40, loss 0.009
epoch 89, iter 60, loss 0.010
epoch 89, test loss 0.017
epoch 90, iter 0, loss 0.009
epoch 90, iter 20, loss 0.008
epoch 90, iter 40, loss 0.009
epoch 90, iter 60, loss 0.009
epoch 90, test loss 0.017
epoch 91, iter 0, loss 0.009
epoch 91, iter 20, loss 0.010
epoch 91, iter 40, loss 0.009
epoch 91, iter 60, loss 0.008
epoch 91, test loss 0.017
epoch 92, iter 0, loss 0.008
epoch 92, iter 20, loss 0.010
epoch 92, iter 40, loss 0.008
epoch 92, iter 60, loss 0.009
epoch 92, test loss 0.016
epoch 93, iter 0, loss 0.011
epoch 93, iter 20, loss 0.010
epoch 93, iter 40, loss 0.009
epoch 93, iter 60, loss 0.010
epoch 93, test loss 0.017
epoch 94, iter 0, loss 0.006
epoch 94, iter 20, loss 0.009
epoch 94, iter 40, loss 0.009
epoch 94, iter 60, loss 0.008
epoch 94, test loss 0.017
epoch 95, iter 0, loss 0.011
epoch 95, iter 20, loss 0.009
epoch 95, iter 40, loss 0.008
epoch 95, iter 60, loss 0.009
epoch 95, test loss 0.018
epoch 96, iter 0, loss 0.009
epoch 96, iter 20, loss 0.009
epoch 96, iter 40, loss 0.009
epoch 96, iter 60, loss 0.008
epoch 96, test loss 0.017
```

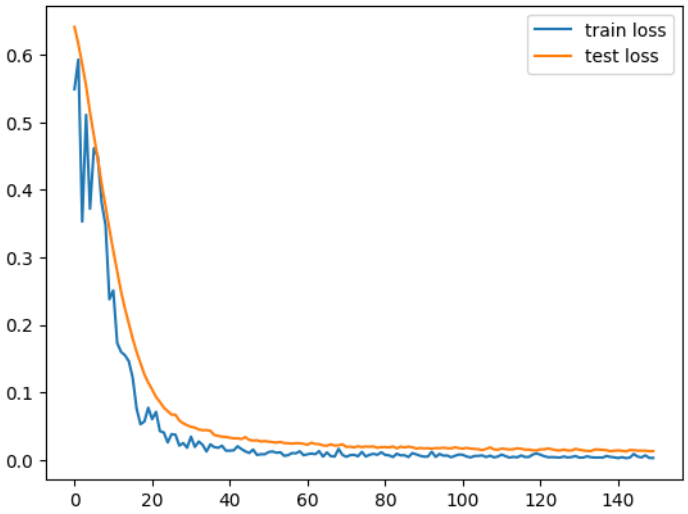
```
epoch 97, iter 0, loss 0.006
epoch 97, iter 20, loss 0.009
epoch 97, iter 40, loss 0.008
epoch 97, iter 60, loss 0.008
epoch 97, test loss 0.017
epoch 98, iter 0, loss 0.008
epoch 98, iter 20, loss 0.009
epoch 98, iter 40, loss 0.008
epoch 98, iter 60, loss 0.008
epoch 98, test loss 0.018
epoch 99, iter 0, loss 0.006
epoch 99, iter 20, loss 0.010
epoch 99, iter 40, loss 0.009
epoch 99, iter 60, loss 0.008
epoch 99, test loss 0.017
epoch 100, iter 0, loss 0.008
epoch 100, iter 20, loss 0.007
epoch 100, iter 40, loss 0.009
epoch 100, iter 60, loss 0.008
epoch 100, test loss 0.017
epoch 101, iter 0, loss 0.008
epoch 101, iter 20, loss 0.010
epoch 101, iter 40, loss 0.009
epoch 101, iter 60, loss 0.008
epoch 101, test loss 0.018
epoch 102, iter 0, loss 0.005
epoch 102, iter 20, loss 0.008
epoch 102, iter 40, loss 0.008
epoch 102, iter 60, loss 0.008
epoch 102, test loss 0.016
epoch 103, iter 0, loss 0.010
epoch 103, iter 20, loss 0.008
epoch 103, iter 40, loss 0.009
epoch 103, iter 60, loss 0.007
epoch 103, test loss 0.016
epoch 104, iter 0, loss 0.008
epoch 104, iter 20, loss 0.008
epoch 104, iter 40, loss 0.007
epoch 104, iter 60, loss 0.008
epoch 104, test loss 0.016
epoch 105, iter 0, loss 0.009
epoch 105, iter 20, loss 0.007
epoch 105, iter 40, loss 0.007
epoch 105, iter 60, loss 0.008
epoch 105, test loss 0.014
epoch 106, iter 0, loss 0.006
epoch 106, iter 20, loss 0.009
epoch 106, iter 40, loss 0.008
epoch 106, iter 60, loss 0.007
epoch 106, test loss 0.016
epoch 107, iter 0, loss 0.007
epoch 107, iter 20, loss 0.008
epoch 107, iter 40, loss 0.008
epoch 107, iter 60, loss 0.008
epoch 107, test loss 0.018
epoch 108, iter 0, loss 0.010
epoch 108, iter 20, loss 0.008
epoch 108, iter 40, loss 0.008
epoch 108, iter 60, loss 0.008
epoch 108, test loss 0.015
epoch 109, iter 0, loss 0.008
epoch 109, iter 20, loss 0.009
epoch 109, iter 40, loss 0.008
epoch 109, iter 60, loss 0.006
epoch 109, test loss 0.015
epoch 110, iter 0, loss 0.006
epoch 110, iter 20, loss 0.008
epoch 110, iter 40, loss 0.007
epoch 110, iter 60, loss 0.008
epoch 110, test loss 0.017
epoch 111, iter 0, loss 0.009
epoch 111, iter 20, loss 0.008
epoch 111, iter 40, loss 0.007
epoch 111, iter 60, loss 0.008
epoch 111, test loss 0.016
epoch 112, iter 0, loss 0.008
epoch 112, iter 20, loss 0.008
epoch 112, iter 40, loss 0.009
epoch 112, iter 60, loss 0.006
epoch 112, test loss 0.015
epoch 113, iter 0, loss 0.005
epoch 113, iter 20, loss 0.008
epoch 113, iter 40, loss 0.007
epoch 113, iter 60, loss 0.008
epoch 113, test loss 0.016
epoch 114, iter 0, loss 0.007
epoch 114, iter 20, loss 0.007
epoch 114, iter 40, loss 0.008
epoch 114, iter 60, loss 0.007
epoch 114, test loss 0.017
epoch 115, iter 0, loss 0.005
epoch 115, iter 20, loss 0.008
epoch 115, iter 40, loss 0.007
epoch 115, iter 60, loss 0.007
epoch 115, test loss 0.016
epoch 116, iter 0, loss 0.006
epoch 116, iter 20, loss 0.007
```

```
epoch 116, iter 40, loss 0.008
epoch 116, iter 60, loss 0.008
epoch 116, test loss 0.015
epoch 117, iter 0, loss 0.008
epoch 117, iter 20, loss 0.007
epoch 117, iter 40, loss 0.008
epoch 117, iter 60, loss 0.007
epoch 117, test loss 0.015
epoch 118, iter 0, loss 0.006
epoch 118, iter 20, loss 0.008
epoch 118, iter 40, loss 0.008
epoch 118, iter 60, loss 0.006
epoch 118, test loss 0.014
epoch 119, iter 0, loss 0.008
epoch 119, iter 20, loss 0.008
epoch 119, iter 40, loss 0.007
epoch 119, iter 60, loss 0.007
epoch 119, test loss 0.014
epoch 120, iter 0, loss 0.008
epoch 120, iter 20, loss 0.009
epoch 120, iter 40, loss 0.008
epoch 120, iter 60, loss 0.008
epoch 120, test loss 0.015
epoch 121, iter 0, loss 0.009
epoch 121, iter 20, loss 0.008
epoch 121, iter 40, loss 0.008
epoch 121, iter 60, loss 0.006
epoch 121, test loss 0.015
epoch 122, iter 0, loss 0.008
epoch 122, iter 20, loss 0.007
epoch 122, iter 40, loss 0.007
epoch 122, iter 60, loss 0.007
epoch 122, test loss 0.017
epoch 123, iter 0, loss 0.005
epoch 123, iter 20, loss 0.007
epoch 123, iter 40, loss 0.008
epoch 123, iter 60, loss 0.007
epoch 123, test loss 0.015
epoch 124, iter 0, loss 0.004
epoch 124, iter 20, loss 0.007
epoch 124, iter 40, loss 0.006
epoch 124, iter 60, loss 0.008
epoch 124, test loss 0.014
epoch 125, iter 0, loss 0.005
epoch 125, iter 20, loss 0.006
epoch 125, iter 40, loss 0.007
epoch 125, iter 60, loss 0.007
epoch 125, test loss 0.014
epoch 126, iter 0, loss 0.007
epoch 126, iter 20, loss 0.007
epoch 126, iter 40, loss 0.007
epoch 126, iter 60, loss 0.007
epoch 126, test loss 0.015
epoch 127, iter 0, loss 0.006
epoch 127, iter 20, loss 0.007
epoch 127, iter 40, loss 0.008
epoch 127, iter 60, loss 0.006
epoch 127, test loss 0.014
epoch 128, iter 0, loss 0.005
epoch 128, iter 20, loss 0.007
epoch 128, iter 40, loss 0.006
epoch 128, iter 60, loss 0.007
epoch 128, test loss 0.014
epoch 129, iter 0, loss 0.008
epoch 129, iter 20, loss 0.008
epoch 129, iter 40, loss 0.007
epoch 129, iter 60, loss 0.007
epoch 129, test loss 0.016
epoch 130, iter 0, loss 0.006
epoch 130, iter 20, loss 0.006
epoch 130, iter 40, loss 0.007
epoch 130, iter 60, loss 0.007
epoch 130, test loss 0.015
epoch 131, iter 0, loss 0.006
epoch 131, iter 20, loss 0.007
epoch 131, iter 40, loss 0.006
epoch 131, iter 60, loss 0.007
epoch 131, test loss 0.014
epoch 132, iter 0, loss 0.006
epoch 132, iter 20, loss 0.007
epoch 132, iter 40, loss 0.007
epoch 132, iter 60, loss 0.007
epoch 132, test loss 0.013
epoch 133, iter 0, loss 0.008
epoch 133, iter 20, loss 0.007
epoch 133, iter 40, loss 0.007
epoch 133, iter 60, loss 0.006
epoch 133, test loss 0.013
epoch 134, iter 0, loss 0.006
epoch 134, iter 20, loss 0.006
epoch 134, iter 40, loss 0.006
epoch 134, iter 60, loss 0.007
epoch 134, test loss 0.015
epoch 135, iter 0, loss 0.005
epoch 135, iter 20, loss 0.007
epoch 135, iter 40, loss 0.007
epoch 135, iter 60, loss 0.006
```

```
epoch 135, test loss 0.015
epoch 136, iter 0, loss 0.005
epoch 136, iter 20, loss 0.006
epoch 136, iter 40, loss 0.007
epoch 136, iter 60, loss 0.007
epoch 136, test loss 0.015
epoch 137, iter 0, loss 0.005
epoch 137, iter 20, loss 0.007
epoch 137, iter 40, loss 0.006
epoch 137, iter 60, loss 0.007
epoch 137, test loss 0.014
epoch 138, iter 0, loss 0.005
epoch 138, iter 20, loss 0.007
epoch 138, iter 40, loss 0.006
epoch 138, iter 60, loss 0.006
epoch 138, test loss 0.013
epoch 139, iter 0, loss 0.007
epoch 139, iter 20, loss 0.007
epoch 139, iter 40, loss 0.007
epoch 139, iter 60, loss 0.006
epoch 139, test loss 0.013
epoch 140, iter 0, loss 0.006
epoch 140, iter 20, loss 0.006
epoch 140, iter 40, loss 0.006
epoch 140, iter 60, loss 0.006
epoch 140, test loss 0.014
epoch 141, iter 0, loss 0.006
epoch 141, iter 20, loss 0.007
epoch 141, iter 40, loss 0.007
epoch 141, iter 60, loss 0.005
epoch 141, test loss 0.013
epoch 142, iter 0, loss 0.007
epoch 142, iter 20, loss 0.006
epoch 142, iter 40, loss 0.006
epoch 142, iter 60, loss 0.006
epoch 142, test loss 0.012
epoch 143, iter 0, loss 0.005
epoch 143, iter 20, loss 0.006
epoch 143, iter 40, loss 0.006
epoch 143, iter 60, loss 0.006
epoch 143, test loss 0.014
epoch 144, iter 0, loss 0.006
epoch 144, iter 20, loss 0.006
epoch 144, iter 40, loss 0.006
epoch 144, iter 60, loss 0.006
epoch 144, test loss 0.014
epoch 145, iter 0, loss 0.008
epoch 145, iter 20, loss 0.007
epoch 145, iter 40, loss 0.007
epoch 145, iter 60, loss 0.006
epoch 145, test loss 0.013
epoch 146, iter 0, loss 0.007
epoch 146, iter 20, loss 0.006
epoch 146, iter 40, loss 0.006
epoch 146, iter 60, loss 0.006
epoch 146, test loss 0.013
epoch 147, iter 0, loss 0.004
epoch 147, iter 20, loss 0.006
epoch 147, iter 40, loss 0.005
epoch 147, iter 60, loss 0.006
epoch 147, test loss 0.013
epoch 148, iter 0, loss 0.006
epoch 148, iter 20, loss 0.006
epoch 148, iter 40, loss 0.006
epoch 148, iter 60, loss 0.006
epoch 148, test loss 0.013
epoch 149, iter 0, loss 0.005
epoch 149, iter 20, loss 0.007
epoch 149, iter 40, loss 0.006
epoch 149, iter 60, loss 0.006
epoch 149, test loss 0.013
Using device: cuda:0
Final evaluation error: 0.0127186065656133
```



Out[17]: <matplotlib.legend.Legend at 0x7f242630c640>



In [17]: