Practical No.1: Implementing a simple GAN architecture using a deep learning framework like TensorFlow or PyTorch.Train the GAN on a basic dataset such as MNIST (handwritten digits) or Fashion-MNIST

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
from torchvision.utils import save_image
import os


# Hyperparameters
latent_dim = 100
batch_size = 64
lr = 0.0002
epochs = 30
image_dir = "gan_images"
os.makedirs(image_dir, exist_ok=True)


# Data loader
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])
])
dataloader = DataLoader(
    datasets.MNIST(root='./data', train=True, download=True,
transform=transform),
```

```
    batch_size=batch_size, shuffle=True
)

# Generator
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 1024),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(1024, 28*28),
            nn.Tanh()
        )

    def forward(self, z):
        img = self.model(z)
        return img.view(z.size(0), 1, 28, 28)

# Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).___init__()
```

```
        self.model = nn.Sequential(
            nn.Linear(28*28, 512),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(256, 1),
            nn.Sigmoid()
        )

    def forward(self, img):
        img_flat = img.view(img.size(0), -1)
        validity = self.model(img_flat)
        return validity

# Initialize models
generator = Generator()
discriminator = Discriminator()

# Loss and optimizers
adversarial_loss = nn.BCELoss()
optimizer_G = optim.Adam(generator.parameters(), lr=lr, betas=(0.5, 0.999))
optimizer_D = optim.Adam(discriminator.parameters(), lr=lr, betas=(0.5, 0.999))

# Training loop
for epoch in range(epochs):
    for i, (imgs, _) in enumerate(dataloader):
```

```
        real_imgs = imgs
        valid = torch.ones(imgs.size(0), 1)
        fake = torch.zeros(imgs.size(0), 1)

        # Train Generator
        optimizer_G.zero_grad()
        z = torch.randn(imgs.size(0), latent_dim)
        gen_imgs = generator(z)
        g_loss = adversarial_loss(discriminator(gen_imgs), valid)
        g_loss.backward()
        optimizer_G.step()

        # Train Discriminator
        optimizer_D.zero_grad()
        real_loss = adversarial_loss(discriminator(real_imgs), valid)
        fake_loss = adversarial_loss(discriminator(gen_imgs.detach()), fake)
        d_loss = (real_loss + fake_loss) / 2
        d_loss.backward()
        optimizer_D.step()

        if i % 200 == 0:
            print(f"[Epoch {epoch}/{epochs}] [Batch {i}/{len(dataloader)}] [D loss: {d_loss.item():.4f}] [G loss: {g_loss.item():.4f}]")

    save_image(gen_imgs.data[:25], f"{image_dir}/{epoch:03d}.png", nrow=5, normalize=True)
```

Practical no 2: Building and training a very simple LLM from scratch.

```
import torch

import torch.nn as nn

import torch.nn.functional as F


# --- Hyperparameters -

-- block_size = 8

batch_size = 4

vocab = sorted(list(set("hello

world"))) vocab_size = len(vocab)

device = 'cuda' if torch.cuda.is_available()

else 'cpu' epochs = 1000

embed_size = 32


# --- Tokenizer ---

stoi = {ch: i for i, ch in

enumerate(vocab)} itos = {i: ch

for ch, i in stoi.items()}

encode = lambda s: [stoi[c] for c in s]

decode = lambda l: ''.join([itos[i] for i in l])


# --- Sample tiny

dataset --- data =

"hello world"

data = torch.tensor(encode(data), dtype=torch.long).to(device)
```

```
# --- Create dataset

--- def get_batch():

  ix = torch.randint(len(data) - block_size, (batch_size,))

  x = torch.stack([data[i:i+block_size] for i in ix])

  y = torch.stack([data[i+1:i+block_size+1] for

  i in ix]) return x, y
```

```
# --- Simple LLM ---
class TinyLLM(nn.Module):
  def __init__(self):
    super().__init__()
    self.embed = nn.Embedding(vocab_size, embed_size)
    self.fc = nn.Linear(embed_size * block_size, vocab_size)

  def forward(self, x):
    x = self.embed(x)  # (B, T, C)
    x = x.view(x.size(0), -1)  #
    flatten return self.fc(x) # (B,
    vocab_size)

model = TinyLLM().to(device)
optimizer = torch.optim.Adam(model.parameters(), lr=1e-3)

# --- Training loop ---
for epoch in range(epochs):
  x, y = get_batch()
  logits = model(x)
  loss = F.cross_entropy(logits, y[:, -1])
  optimizer.zero_grad()
  loss.backwa
  rd()
  optimizer.st
  ep()
  if epoch % 100 == 0:
```

```
    print(f"Epoch {epoch}, Loss: {loss.item():.4f}")

# --- Generation ---
def generate(model, start='h',
  max_new_tokens=20): model.eval()
  idx = torch.tensor([encode(start)], dtype=torch.long).to(device)
```

```
  for _ in
    range(max_new_tokens)
    : idx_cond = idx[:, -
    block_size:]
    # If idx_cond has less than block_size tokens, pad
    it with zeros if idx_cond.shape[1] < block_size:
      padding = torch.zeros(1, block_size - idx_cond.shape[1],
      dtype=torch.long).to(device) idx_cond = torch.cat([padding,
      idx_cond], dim=1)
    logits = model(idx_cond)
    probs = F.softmax(logits, dim=-1)
    next_id = torch.multinomial(probs,
    num_samples=1) idx = torch.cat((idx,
    next_id), dim=1)
  return decode(idx[0].tolist())


print("Generated text:", generate(model))
```