Assignment 4

This is an basecode for assignment 4 of Artificial Intelligence class (CSCE-4613), Spring 2025

In [2]:
```
import torch
import torch.nn as nn
import torchvision
import time
import numpy as np
import matplotlib.pyplot as plt
from sklearn.metrics import confusion_matrix
import seaborn as sns
```

Binary Network

Define a binary network class

In [3]:
```
class BinaryNetwork(nn.Module):
  def __init__(self):
    # YOUR CODE HERE
    super(BinaryNetwork, self).__init__()
    self.layer1 = nn.Linear(2, 4)
    self.layer2 = nn.Linear(4, 1)

  def forward(self, x):
    # x has the size of (batch size x 2)
    # YOUR CODE HERE
    x = torch.sigmoid(self.layer1(x))
    x = torch.sigmoid(self.layer2(x))
    return x
```

Define data generator

In [4]:
```
def generate_data(operator = "AND"):
  assert operator in ["AND", "OR", "XOR", "NOR"], "%s operator is not valid" % operat
  data = []
  label = []
  for i in range(2):
    for j in range(2):
      data.append([i, j])
      if operator == "AND":
        label.append(i & j)
      elif operator == "OR":
        label.append(i | j)
      elif operator == "XOR":
        label.append(i ^ j)
      else:
        label.append(not (i | j))
  data = torch.as_tensor(data, dtype = torch.float32)
```

```
  label = torch.as_tensor(label, dtype = torch.float32)
  return data, label
```

Define the training framework

In [5]:
```
model = BinaryNetwork()
model.train()
print(model)
operator = "XOR"
inputs, labels = generate_data(operator = operator)

n_iters = 1000
learning_rate = 0.1

optim = torch.optim.SGD(params = model.parameters(), lr = learning_rate, momentum=0.9
criterion = nn.BCELoss()  # Binary Cross Entropy Loss
losses = []  # To track progress

for i in range(1, n_iters + 1):

  # WRITE YOUR CODE TO COMPUTE OUTPUTS, LOSS, ACCURACY, AND OPTIMIZE MODEL
  # outptus = ???
  outputs = model(inputs)
  loss = criterion(outputs, labels.view(-1, 1))
  losses.append(loss.item())

  # Backpropagation and optimization
  optim.zero_grad()
  loss.backward()
  optim.step()
  # loss = ???
  # accuracy = ???
  # optimize the model
  # loss = 0.0
  # accuracy = 0.0
  predicted = (outputs >= 0.5).float()
  accuracy = (predicted.view(-1) == labels).sum().item() / len(labels) * 100

  if i % 5 == 0:
    print("[%d/%d]. Loss: %0.4f. Accuracy: %0.2f" % (i, n_iters, loss, accuracy))

model.eval()
# WRITE YOUR CODE TO CALCULATE THE FINAL ACCURACY
# accuracy = ???
# accuracy = 0.0
with torch.no_grad():
    outputs = model(inputs)
    predicted = (outputs >= 0.5).float()
    accuracy = (predicted.view(-1) == labels).sum().item() / len(labels) * 100
print("Final Accuracy: %0.2f" % (accuracy))

torch.save(model.state_dict(), "%s_Network.pth" % operator)

import matplotlib.pyplot as plt
plt.figure(figsize=(10, 5))
plt.plot(losses)
```

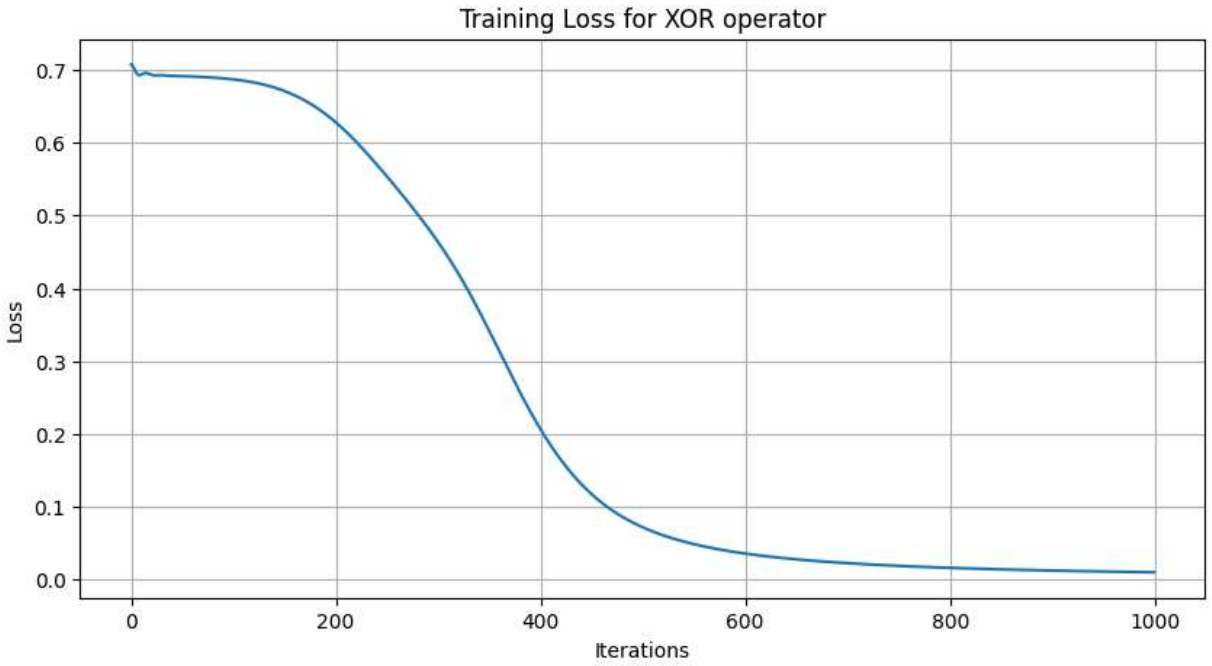
```
plt.title(f'Training Loss for {operator} operator')
plt.xlabel('Iterations')
plt.ylabel('Loss')
plt.grid(True)
plt.show()
  # model.load_state_dict(torch.load("%s_Network.pth" % operator)) # Load model in th
```

```
BinaryNetwork(
  (layer1): Linear(in_features=2, out_features=4, bias=True)
  (layer2): Linear(in_features=4, out_features=1, bias=True)
)
[5/1000]. Loss: 0.6986. Accuracy: 50.00
[10/1000]. Loss: 0.6928. Accuracy: 25.00
[15/1000]. Loss: 0.6956. Accuracy: 50.00
[20/1000]. Loss: 0.6937. Accuracy: 50.00
[25/1000]. Loss: 0.6923. Accuracy: 75.00
[30/1000]. Loss: 0.6928. Accuracy: 75.00
[35/1000]. Loss: 0.6923. Accuracy: 75.00
[40/1000]. Loss: 0.6918. Accuracy: 50.00
[45/1000]. Loss: 0.6917. Accuracy: 25.00
[50/1000]. Loss: 0.6915. Accuracy: 25.00
[55/1000]. Loss: 0.6912. Accuracy: 50.00
[60/1000]. Loss: 0.6909. Accuracy: 75.00
[65/1000]. Loss: 0.6906. Accuracy: 75.00
[70/1000]. Loss: 0.6903. Accuracy: 50.00
[75/1000]. Loss: 0.6899. Accuracy: 50.00
[80/1000]. Loss: 0.6895. Accuracy: 50.00
[85/1000]. Loss: 0.6890. Accuracy: 50.00
[90/1000]. Loss: 0.6884. Accuracy: 50.00
[95/1000]. Loss: 0.6878. Accuracy: 50.00
[100/1000]. Loss: 0.6871. Accuracy: 50.00
[105/1000]. Loss: 0.6863. Accuracy: 50.00
[110/1000]. Loss: 0.6853. Accuracy: 50.00
[115/1000]. Loss: 0.6842. Accuracy: 50.00
[120/1000]. Loss: 0.6830. Accuracy: 50.00
[125/1000]. Loss: 0.6816. Accuracy: 75.00
[130/1000]. Loss: 0.6800. Accuracy: 75.00
[135/1000]. Loss: 0.6782. Accuracy: 75.00
[140/1000]. Loss: 0.6762. Accuracy: 75.00
[145/1000]. Loss: 0.6739. Accuracy: 75.00
[150/1000]. Loss: 0.6714. Accuracy: 75.00
[155/1000]. Loss: 0.6686. Accuracy: 75.00
[160/1000]. Loss: 0.6655. Accuracy: 75.00
[165/1000]. Loss: 0.6622. Accuracy: 75.00
[170/1000]. Loss: 0.6585. Accuracy: 75.00
[175/1000]. Loss: 0.6545. Accuracy: 75.00
[180/1000]. Loss: 0.6501. Accuracy: 75.00
[185/1000]. Loss: 0.6454. Accuracy: 75.00
[190/1000]. Loss: 0.6404. Accuracy: 75.00
[195/1000]. Loss: 0.6350. Accuracy: 75.00
[200/1000]. Loss: 0.6292. Accuracy: 75.00
[205/1000]. Loss: 0.6230. Accuracy: 75.00
[210/1000]. Loss: 0.6165. Accuracy: 75.00
[215/1000]. Loss: 0.6097. Accuracy: 75.00
[220/1000]. Loss: 0.6025. Accuracy: 75.00
[225/1000]. Loss: 0.5951. Accuracy: 75.00
[230/1000]. Loss: 0.5875. Accuracy: 75.00
[235/1000]. Loss: 0.5797. Accuracy: 75.00
[240/1000]. Loss: 0.5717. Accuracy: 75.00
[245/1000]. Loss: 0.5635. Accuracy: 75.00
[250/1000]. Loss: 0.5553. Accuracy: 75.00
[255/1000]. Loss: 0.5469. Accuracy: 75.00
[260/1000]. Loss: 0.5385. Accuracy: 75.00
[265/1000]. Loss: 0.5299. Accuracy: 75.00
```

```
[270/1000]. Loss: 0.5212. Accuracy: 75.00
[275/1000]. Loss: 0.5124. Accuracy: 75.00
[280/1000]. Loss: 0.5035. Accuracy: 75.00
[285/1000]. Loss: 0.4943. Accuracy: 75.00
[290/1000]. Loss: 0.4849. Accuracy: 75.00
[295/1000]. Loss: 0.4753. Accuracy: 75.00
[300/1000]. Loss: 0.4654. Accuracy: 75.00
[305/1000]. Loss: 0.4551. Accuracy: 75.00
[310/1000]. Loss: 0.4445. Accuracy: 75.00
[315/1000]. Loss: 0.4335. Accuracy: 75.00
[320/1000]. Loss: 0.4220. Accuracy: 75.00
[325/1000]. Loss: 0.4100. Accuracy: 75.00
[330/1000]. Loss: 0.3976. Accuracy: 100.00
[335/1000]. Loss: 0.3848. Accuracy: 100.00
[340/1000]. Loss: 0.3715. Accuracy: 100.00
[345/1000]. Loss: 0.3579. Accuracy: 100.00
[350/1000]. Loss: 0.3439. Accuracy: 100.00
[355/1000]. Loss: 0.3298. Accuracy: 100.00
[360/1000]. Loss: 0.3156. Accuracy: 100.00
[365/1000]. Loss: 0.3014. Accuracy: 100.00
[370/1000]. Loss: 0.2872. Accuracy: 100.00
[375/1000]. Loss: 0.2733. Accuracy: 100.00
[380/1000]. Loss: 0.2596. Accuracy: 100.00
[385/1000]. Loss: 0.2463. Accuracy: 100.00
[390/1000]. Loss: 0.2333. Accuracy: 100.00
[395/1000]. Loss: 0.2208. Accuracy: 100.00
[400/1000]. Loss: 0.2088. Accuracy: 100.00
[405/1000]. Loss: 0.1973. Accuracy: 100.00
[410/1000]. Loss: 0.1864. Accuracy: 100.00
[415/1000]. Loss: 0.1760. Accuracy: 100.00
[420/1000]. Loss: 0.1662. Accuracy: 100.00
[425/1000]. Loss: 0.1569. Accuracy: 100.00
[430/1000]. Loss: 0.1482. Accuracy: 100.00
[435/1000]. Loss: 0.1400. Accuracy: 100.00
[440/1000]. Loss: 0.1323. Accuracy: 100.00
[445/1000]. Loss: 0.1252. Accuracy: 100.00
[450/1000]. Loss: 0.1185. Accuracy: 100.00
[455/1000]. Loss: 0.1123. Accuracy: 100.00
[460/1000]. Loss: 0.1065. Accuracy: 100.00
[465/1000]. Loss: 0.1011. Accuracy: 100.00
[470/1000]. Loss: 0.0961. Accuracy: 100.00
[475/1000]. Loss: 0.0914. Accuracy: 100.00
[480/1000]. Loss: 0.0870. Accuracy: 100.00
[485/1000]. Loss: 0.0830. Accuracy: 100.00
[490/1000]. Loss: 0.0792. Accuracy: 100.00
[495/1000]. Loss: 0.0757. Accuracy: 100.00
[500/1000]. Loss: 0.0724. Accuracy: 100.00
[505/1000]. Loss: 0.0693. Accuracy: 100.00
[510/1000]. Loss: 0.0664. Accuracy: 100.00
[515/1000]. Loss: 0.0637. Accuracy: 100.00
[520/1000]. Loss: 0.0612. Accuracy: 100.00
[525/1000]. Loss: 0.0589. Accuracy: 100.00
[530/1000]. Loss: 0.0566. Accuracy: 100.00
[535/1000]. Loss: 0.0546. Accuracy: 100.00
[540/1000]. Loss: 0.0526. Accuracy: 100.00
[545/1000]. Loss: 0.0507. Accuracy: 100.00
[550/1000]. Loss: 0.0490. Accuracy: 100.00
```

```
[555/1000]. Loss: 0.0474. Accuracy: 100.00
[560/1000]. Loss: 0.0458. Accuracy: 100.00
[565/1000]. Loss: 0.0443. Accuracy: 100.00
[570/1000]. Loss: 0.0430. Accuracy: 100.00
[575/1000]. Loss: 0.0416. Accuracy: 100.00
[580/1000]. Loss: 0.0404. Accuracy: 100.00
[585/1000]. Loss: 0.0392. Accuracy: 100.00
[590/1000]. Loss: 0.0381. Accuracy: 100.00
[595/1000]. Loss: 0.0370. Accuracy: 100.00
[600/1000]. Loss: 0.0360. Accuracy: 100.00
[605/1000]. Loss: 0.0350. Accuracy: 100.00
[610/1000]. Loss: 0.0341. Accuracy: 100.00
[615/1000]. Loss: 0.0332. Accuracy: 100.00
[620/1000]. Loss: 0.0324. Accuracy: 100.00
[625/1000]. Loss: 0.0315. Accuracy: 100.00
[630/1000]. Loss: 0.0308. Accuracy: 100.00
[635/1000]. Loss: 0.0300. Accuracy: 100.00
[640/1000]. Loss: 0.0293. Accuracy: 100.00
[645/1000]. Loss: 0.0286. Accuracy: 100.00
[650/1000]. Loss: 0.0280. Accuracy: 100.00
[655/1000]. Loss: 0.0274. Accuracy: 100.00
[660/1000]. Loss: 0.0268. Accuracy: 100.00
[665/1000]. Loss: 0.0262. Accuracy: 100.00
[670/1000]. Loss: 0.0256. Accuracy: 100.00
[675/1000]. Loss: 0.0251. Accuracy: 100.00
[680/1000]. Loss: 0.0246. Accuracy: 100.00
[685/1000]. Loss: 0.0241. Accuracy: 100.00
[690/1000]. Loss: 0.0236. Accuracy: 100.00
[695/1000]. Loss: 0.0231. Accuracy: 100.00
[700/1000]. Loss: 0.0227. Accuracy: 100.00
[705/1000]. Loss: 0.0223. Accuracy: 100.00
[710/1000]. Loss: 0.0218. Accuracy: 100.00
[715/1000]. Loss: 0.0214. Accuracy: 100.00
[720/1000]. Loss: 0.0211. Accuracy: 100.00
[725/1000]. Loss: 0.0207. Accuracy: 100.00
[730/1000]. Loss: 0.0203. Accuracy: 100.00
[735/1000]. Loss: 0.0200. Accuracy: 100.00
[740/1000]. Loss: 0.0196. Accuracy: 100.00
[745/1000]. Loss: 0.0193. Accuracy: 100.00
[750/1000]. Loss: 0.0190. Accuracy: 100.00
[755/1000]. Loss: 0.0187. Accuracy: 100.00
[760/1000]. Loss: 0.0184. Accuracy: 100.00
[765/1000]. Loss: 0.0181. Accuracy: 100.00
[770/1000]. Loss: 0.0178. Accuracy: 100.00
[775/1000]. Loss: 0.0175. Accuracy: 100.00
[780/1000]. Loss: 0.0172. Accuracy: 100.00
[785/1000]. Loss: 0.0170. Accuracy: 100.00
[790/1000]. Loss: 0.0167. Accuracy: 100.00
[795/1000]. Loss: 0.0165. Accuracy: 100.00
[800/1000]. Loss: 0.0162. Accuracy: 100.00
[805/1000]. Loss: 0.0160. Accuracy: 100.00
[810/1000]. Loss: 0.0158. Accuracy: 100.00
[815/1000]. Loss: 0.0155. Accuracy: 100.00
[820/1000]. Loss: 0.0153. Accuracy: 100.00
[825/1000]. Loss: 0.0151. Accuracy: 100.00
[830/1000]. Loss: 0.0149. Accuracy: 100.00
[835/1000]. Loss: 0.0147. Accuracy: 100.00
```

```
[840/1000]. Loss: 0.0145. Accuracy: 100.00
[845/1000]. Loss: 0.0143. Accuracy: 100.00
[850/1000]. Loss: 0.0141. Accuracy: 100.00
[855/1000]. Loss: 0.0140. Accuracy: 100.00
[860/1000]. Loss: 0.0138. Accuracy: 100.00
[865/1000]. Loss: 0.0136. Accuracy: 100.00
[870/1000]. Loss: 0.0134. Accuracy: 100.00
[875/1000]. Loss: 0.0133. Accuracy: 100.00
[880/1000]. Loss: 0.0131. Accuracy: 100.00
[885/1000]. Loss: 0.0129. Accuracy: 100.00
[890/1000]. Loss: 0.0128. Accuracy: 100.00
[895/1000]. Loss: 0.0126. Accuracy: 100.00
[900/1000]. Loss: 0.0125. Accuracy: 100.00
[905/1000]. Loss: 0.0123. Accuracy: 100.00
[910/1000]. Loss: 0.0122. Accuracy: 100.00
[915/1000]. Loss: 0.0121. Accuracy: 100.00
[920/1000]. Loss: 0.0119. Accuracy: 100.00
[925/1000]. Loss: 0.0118. Accuracy: 100.00
[930/1000]. Loss: 0.0117. Accuracy: 100.00
[935/1000]. Loss: 0.0115. Accuracy: 100.00
[940/1000]. Loss: 0.0114. Accuracy: 100.00
[945/1000]. Loss: 0.0113. Accuracy: 100.00
[950/1000]. Loss: 0.0112. Accuracy: 100.00
[955/1000]. Loss: 0.0111. Accuracy: 100.00
[960/1000]. Loss: 0.0109. Accuracy: 100.00
[965/1000]. Loss: 0.0108. Accuracy: 100.00
[970/1000]. Loss: 0.0107. Accuracy: 100.00
[975/1000]. Loss: 0.0106. Accuracy: 100.00
[980/1000]. Loss: 0.0105. Accuracy: 100.00
[985/1000]. Loss: 0.0104. Accuracy: 100.00
[990/1000]. Loss: 0.0103. Accuracy: 100.00
[995/1000]. Loss: 0.0102. Accuracy: 100.00
[1000/1000]. Loss: 0.0101. Accuracy: 100.00
Final Accuracy: 100.00
```



Digit Classification

Define Digit Classification Network

In [6]:
```
class DigitNetwork(nn.Module):
  def __init__(self):
    super(DigitNetwork, self).__init__()
    # YOUR CODE HERE
    self.conv1 = nn.Conv2d(1, 32, kernel_size=3, stride=1, padding=1)
    self.conv2 = nn.Conv2d(32, 64, kernel_size=3, stride=1, padding=1)
    self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
    self.fc1 = nn.Linear(64 * 7 * 7, 128)
    self.fc2 = nn.Linear(128, 10)
    self.dropout = nn.Dropout(0.25)

  def forward(self, x):
    # x has the size of (batch size x 1 x height x height)
    # YOUR CODE HERE
    x = self.pool(torch.relu(self.conv1(x)))
    x = self.pool(torch.relu(self.conv2(x)))
    x = x.view(-1, 64 * 7 * 7)
    x = torch.relu(self.fc1(x))
    x = self.dropout(x)
    x = self.fc2(x)
    return x

  def count_parameters(self):
    """Count and return the number of parameters in each layer and total"""
    params_count = {}

    # Count parameters in each layer
    params_count['conv1'] = sum(p.numel() for p in self.conv1.parameters())
    params_count['conv2'] = sum(p.numel() for p in self.conv2.parameters())
    params_count['fc1'] = sum(p.numel() for p in self.fc1.parameters())
    params_count['fc2'] = sum(p.numel() for p in self.fc2.parameters())

    # Calculate total parameters
    params_count['total'] = sum(p.numel() for p in self.parameters())

    return params_count
```

Define Data Generator

In [7]:
```
def create_data_generator(batch_size = 32, root = "data"):
  train_dataset = torchvision.datasets.MNIST(root = root,
                                             train = True,
                                             transform = torchvision.transforms.ToTen
                                             download = True)
  test_dataset = torchvision.datasets.MNIST(root = root,
                                            train = False,
                                            transform = torchvision.transforms.ToTen
                                            download = True)
  train_loader = torch.utils.data.DataLoader(train_dataset,
                                             batch_size = batch_size,
                                             shuffle = True)
  test_loader = torch.utils.data.DataLoader(test_dataset,
```

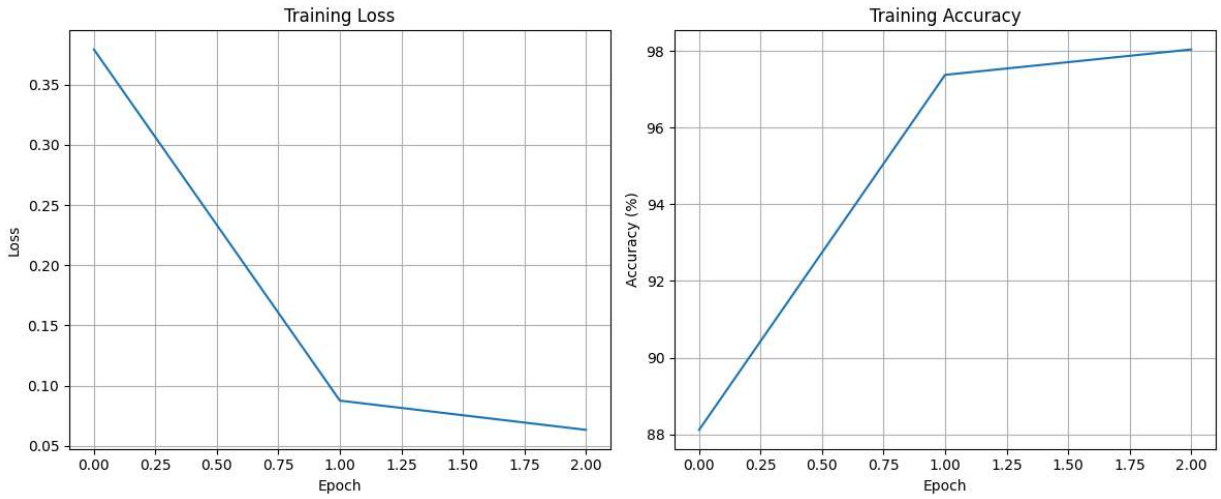
```
                                                batch_size = batch_size,
                                                shuffle = False)
  return train_loader, test_loader
```

Define the training framework

In [8]:
```
cuda = torch.cuda.is_available()
batch_size = 64 #32
train_loader, test_loader = create_data_generator(batch_size)
model = DigitNetwork()
print(model)

params = model.count_parameters()
print("Parameters per layer:")
for layer, count in params.items():
    print(f"{layer}: {count}")

if cuda:
  model.cuda()
  print("Using GPU for training")
else:
  print("Using CPU for training")

n_epochs = 3 #1
learning_rate = 0.01 #0.1
optim = torch.optim.SGD(params = model.parameters(), lr = learning_rate, momentum=0.9
loss_fn = nn.CrossEntropyLoss()

losses = []
accuracies = []
batch_times = []
total_start_time = time.time()

model.train()
for epoch in range(1, n_epochs + 1):
  epoch_loss = 0.0
  correct = 0
  total = 0
  for idx, (images, labels) in enumerate(train_loader):
    # WRITE YOUR CODE TO COMPUTE OUTPUTS, LOSS, ACCURACY, AND OPTIMIZE MODEL
    # outptus = ???
    # loss = ???
    # accuracy = ???
    # optimize the model
    batch_start_time = time.time()
    if cuda:
      images = images.cuda()
      labels = labels.cuda()

    outputs = model(images)
    loss = loss_fn(outputs, labels)
    optim.zero_grad()
    loss.backward()
    optim.step()

    _, predicted = torch.max(outputs.data, 1)
```

```
    total += labels.size(0)
    correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total

    epoch_loss += loss.item()
    batch_end_time = time.time()
    batch_times.append(batch_end_time - batch_start_time)
    if idx % 100 == 0:
      print("Epoch [%d/%d]. Iter [%d/%d]. Loss: %0.2f. Accuracy: %0.2f" % (epoch, n_e

  epoch_avg_loss = epoch_loss / len(train_loader)
  losses.append(epoch_avg_loss)
  accuracies.append(accuracy)

  print(f"Epoch {epoch} completed. Average loss: {epoch_avg_loss:.4f}, Accuracy: {acc

total_training_time = time.time() - total_start_time
avg_batch_time = np.mean(batch_times)
print(f"Total training time: {total_training_time:.2f} seconds")
print(f"Average time per batch: {avg_batch_time:.4f} seconds")

torch.save(model.state_dict(), "MNIST_Network.pth")

plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(losses)
plt.title('Training Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(accuracies)
plt.title('Training Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.grid(True)
plt.tight_layout()
plt.savefig('training_metrics.png')
plt.show()
```

```
DigitNetwork(
  (conv1): Conv2d(1, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (conv2): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (pool): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (fc1): Linear(in_features=3136, out_features=128, bias=True)
  (fc2): Linear(in_features=128, out_features=10, bias=True)
  (dropout): Dropout(p=0.25, inplace=False)
)
Parameters per layer:
conv1: 320
conv2: 18496
fc1: 401536
fc2: 1290
total: 421642
Using CPU for training
Epoch [1/3]. Iter [1/938]. Loss: 2.29. Accuracy: 12.50
Epoch [1/3]. Iter [101/938]. Loss: 0.74. Accuracy: 46.60
Epoch [1/3]. Iter [201/938]. Loss: 0.37. Accuracy: 65.13
Epoch [1/3]. Iter [301/938]. Loss: 0.29. Accuracy: 73.39
Epoch [1/3]. Iter [401/938]. Loss: 0.18. Accuracy: 78.24
Epoch [1/3]. Iter [501/938]. Loss: 0.33. Accuracy: 81.50
Epoch [1/3]. Iter [601/938]. Loss: 0.08. Accuracy: 83.79
Epoch [1/3]. Iter [701/938]. Loss: 0.10. Accuracy: 85.47
Epoch [1/3]. Iter [801/938]. Loss: 0.22. Accuracy: 86.73
Epoch [1/3]. Iter [901/938]. Loss: 0.08. Accuracy: 87.79
Epoch 1 completed. Average loss: 0.3791, Accuracy: 88.12%
Epoch [2/3]. Iter [1/938]. Loss: 0.05. Accuracy: 98.44
Epoch [2/3]. Iter [101/938]. Loss: 0.09. Accuracy: 97.06
Epoch [2/3]. Iter [201/938]. Loss: 0.05. Accuracy: 97.08
Epoch [2/3]. Iter [301/938]. Loss: 0.11. Accuracy: 97.11
Epoch [2/3]. Iter [401/938]. Loss: 0.11. Accuracy: 97.16
Epoch [2/3]. Iter [501/938]. Loss: 0.06. Accuracy: 97.23
Epoch [2/3]. Iter [601/938]. Loss: 0.05. Accuracy: 97.25
Epoch [2/3]. Iter [701/938]. Loss: 0.25. Accuracy: 97.31
Epoch [2/3]. Iter [801/938]. Loss: 0.08. Accuracy: 97.36
Epoch [2/3]. Iter [901/938]. Loss: 0.11. Accuracy: 97.34
Epoch 2 completed. Average loss: 0.0876, Accuracy: 97.38%
Epoch [3/3]. Iter [1/938]. Loss: 0.02. Accuracy: 100.00
Epoch [3/3]. Iter [101/938]. Loss: 0.03. Accuracy: 97.83
Epoch [3/3]. Iter [201/938]. Loss: 0.09. Accuracy: 97.87
Epoch [3/3]. Iter [301/938]. Loss: 0.01. Accuracy: 97.89
Epoch [3/3]. Iter [401/938]. Loss: 0.06. Accuracy: 97.92
Epoch [3/3]. Iter [501/938]. Loss: 0.08. Accuracy: 97.98
Epoch [3/3]. Iter [601/938]. Loss: 0.08. Accuracy: 97.98
Epoch [3/3]. Iter [701/938]. Loss: 0.03. Accuracy: 98.02
Epoch [3/3]. Iter [801/938]. Loss: 0.12. Accuracy: 98.05
Epoch [3/3]. Iter [901/938]. Loss: 0.04. Accuracy: 98.06
Epoch 3 completed. Average loss: 0.0633, Accuracy: 98.04%
Total training time: 82.63 seconds
Average time per batch: 0.0227 seconds
```

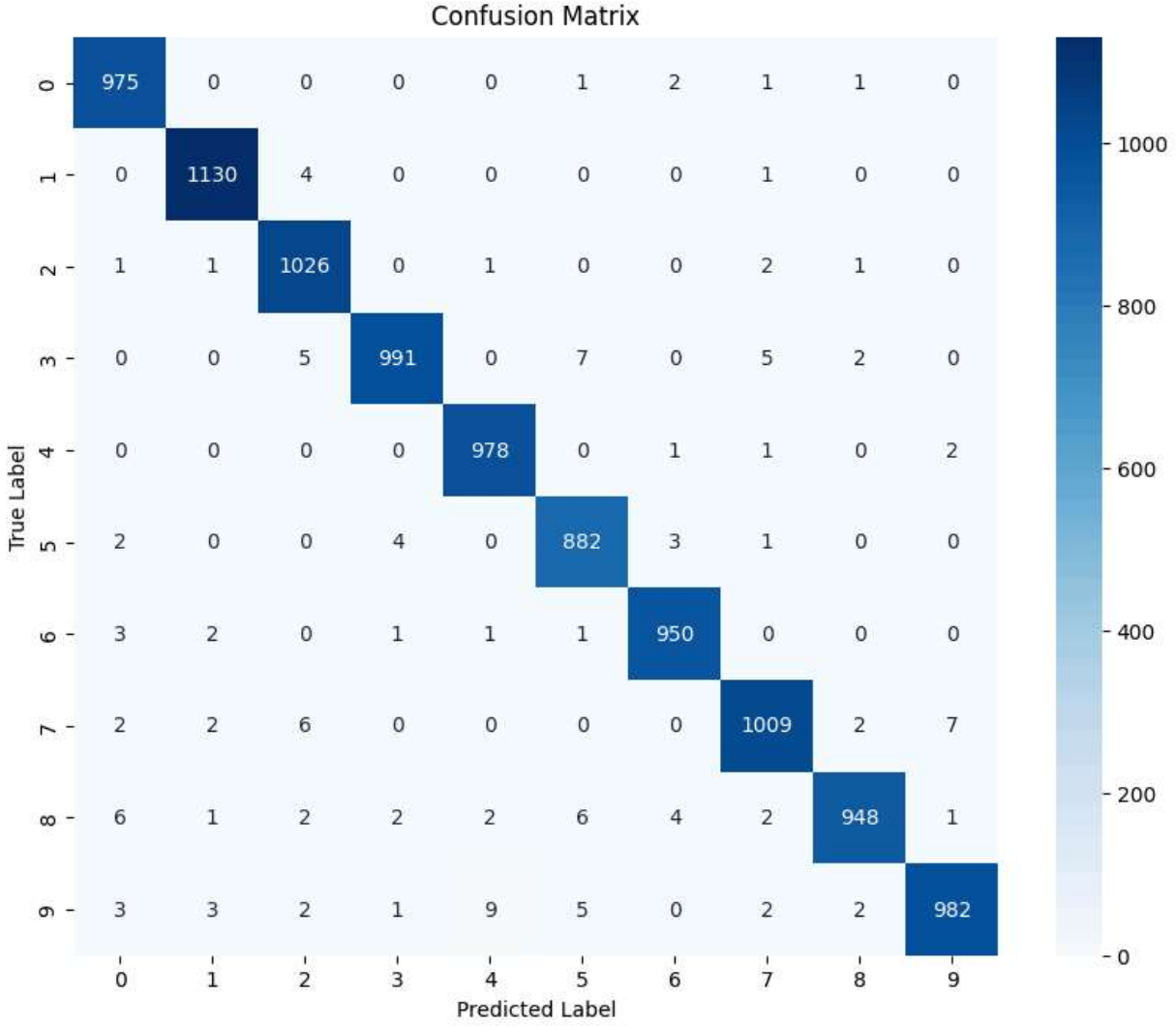


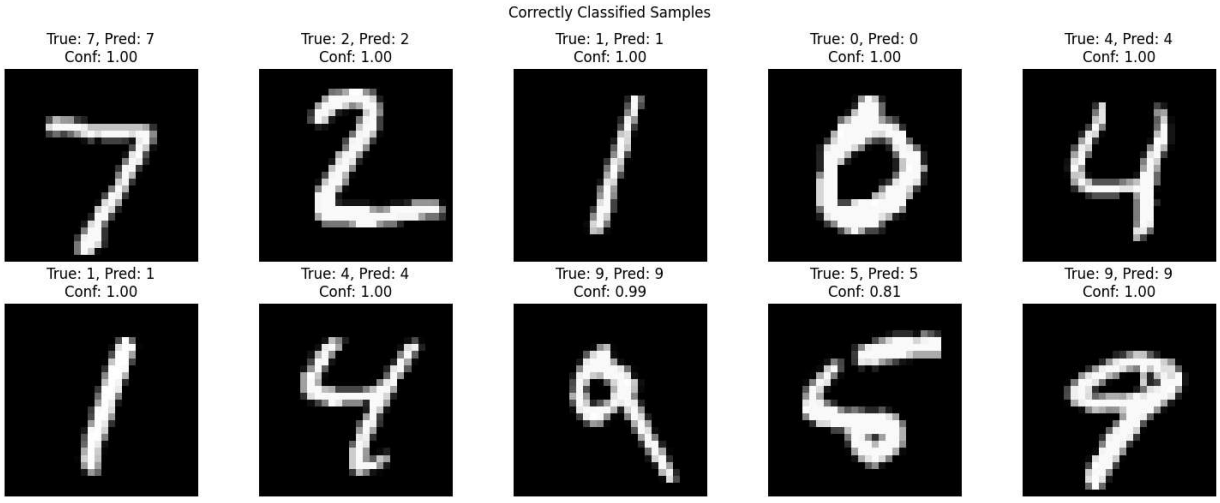
Define the evaluation framework

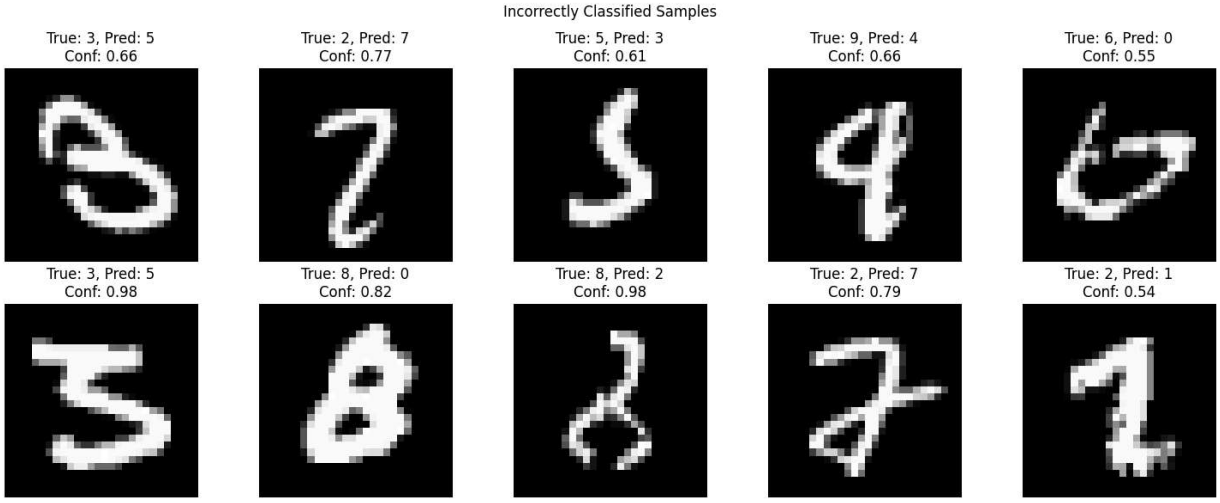
In [9]:
```
cuda = torch.cuda.is_available()
batch_size = 1
train_loader, test_loader = create_data_generator(batch_size)
model = DigitNetwork()
if cuda:
  model.cuda()
model.eval()
model.load_state_dict(torch.load("MNIST_Network.pth"))

all_preds = []
all_labels = []
inference_times = []
total_accuracy = 0.0

correct_samples = []
incorrect_samples = []
correct_confidences = []
incorrect_confidences = []

with torch.no_grad():
  for idx, (images, labels) in enumerate(test_loader):
  # WRITE YOUR CODE TO COMPUTE ACCURACY
  # accuracy = ???
    # accuracy = 0.0
    if cuda:
      images = images.cuda()
      labels = labels.cuda()

    start_time = time.time()
    outputs = model(images)
    end_time = time.time()
    inference_times.append(end_time - start_time)
    probs = torch.nn.functional.softmax(outputs, dim=1)
    confidence, predicted = torch.max(probs, 1)
    correct = (predicted == labels).sum().item()
    accuracy = 100 * correct / labels.size(0)
    all_preds.append(predicted.cpu().item())
    all_labels.append(labels.cpu().item())
```

```
    if correct == 1 and len(correct_samples) < 10:
      correct_samples.append((images.cpu().numpy(), labels.cpu().item(), predicted.cp
      correct_confidences.append(confidence.cpu().item())
    elif correct == 0 and len(incorrect_samples) < 10:
      incorrect_samples.append((images.cpu().numpy(), labels.cpu().item(), predicted.
      incorrect_confidences.append(confidence.cpu().item())

    total_accuracy += accuracy

    if idx % 2000 == 0:
      print("Iter [%d/%d]. Accuracy: %0.2f" % (idx + 1, len(test_loader), accuracy))

avg_inference_time = np.mean(inference_times)
print("Final Accuracy: %0.2f" % (total_accuracy / len(test_loader)))
print("Average inference time per sample: %.5f seconds" % avg_inference_time)
cm = confusion_matrix(all_labels, all_preds)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.savefig('confusion_matrix.png')
plt.show()
if correct_samples:
    fig, axes = plt.subplots(2, 5, figsize=(15, 6))
    axes = axes.flatten()

    for i, ((img, label, pred), conf) in enumerate(zip(correct_samples, correct_confi
        if i < 10:
            axes[i].imshow(img.squeeze(), cmap='gray')
            axes[i].set_title(f'True: {label}, Pred: {pred}\nConf: {conf:.2f}')
            axes[i].axis('off')

    plt.suptitle('Correctly Classified Samples')
    plt.tight_layout()
    plt.savefig('correct_samples.png')
    plt.show()

# Visualize incorrectly classified samples
if incorrect_samples:
    fig, axes = plt.subplots(2, 5, figsize=(15, 6))
    axes = axes.flatten()

    for i, ((img, label, pred), conf) in enumerate(zip(incorrect_samples, incorrect_c
        if i < 10:
            axes[i].imshow(img.squeeze(), cmap='gray')
            axes[i].set_title(f'True: {label}, Pred: {pred}\nConf: {conf:.2f}')
            axes[i].axis('off')

    plt.suptitle('Incorrectly Classified Samples')
    plt.tight_layout()
    plt.savefig('incorrect_samples.png')
    plt.show()
```

```
Iter [1/10000]. Accuracy: 100.00
Iter [2001/10000]. Accuracy: 100.00
Iter [4001/10000]. Accuracy: 100.00
Iter [6001/10000]. Accuracy: 100.00
Iter [8001/10000]. Accuracy: 100.00
Final Accuracy: 98.71
Average inference time per sample: 0.00049 seconds
```







Backpropagation

ReLU Example

In [10]:
```python
# https://pytorch.org/tutorials/beginner/pytorch_with_examples.html#pytorch-defining-
class MyReLU(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        """
        In the forward pass we receive a Tensor containing the input and return
        a Tensor containing the output. ctx is a context object that can be used
        to stash information for backward computation. You can cache arbitrary
        objects for use in the backward pass using the ctx.save_for_backward method.
        """
        ctx.save_for_backward(input)
        return input.clamp(min=0)

    @staticmethod
    def backward(ctx, grad_output):
        """
        In the backward pass we receive a Tensor containing the gradient of the loss
        with respect to the output, and we need to compute the gradient of the loss
        with respect to the input.
        """
        input, = ctx.saved_tensors
        grad_input = grad_output.clone()
        grad_input[input < 0] = 0
        return grad_input
```

Sigmoid Function

In [11]:
```python
class MySigmoid(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input):
        # input is a N x C tensor, N is the batch size, C is the dimension of input
        ctx.save_for_backward(input)
        output = 1.0 / (1.0 + torch.exp(-input))
        ctx.save_for_backward(output)
        return output
        # YOUR CODE HERE
        # return output of sigmoid function

    @staticmethod
    def backward(ctx, grad_output):
        input, = ctx.saved_tensors
        output, = ctx.saved_tensors
        grad_input = grad_output * output * (1 - output)
        return grad_input
        # YOUR CODE HERE
        # return grad_input
```

Fully Connected Layer

In [12]:
```python
class MyLinearFunction(torch.autograd.Function):
    """
    We can implement our own custom autograd Functions by subclassing
    torch.autograd.Function and implementing the forward and backward passes
    which operate on Tensors.
    """

    @staticmethod
    def forward(ctx, input, weights, bias):
        # input is a N x C tensor, N is the batch size, C is the dimension of input
        # weights is a C x D tensor, C and D are the dimension out input and ouput
        # bias is D tensor
        ctx.save_for_backward(input, weights, bias)
        # YOUR CODE HERE
        # return output of linear function
        return torch.matmul(input, weights) + bias


    @staticmethod
    def backward(ctx, grad_output):
        input, weights, bias = ctx.saved_tensors
        grad_input = torch.matmul(grad_output, weights.t())
        grad_weights = torch.matmul(input.t(), grad_output)
        grad_bias = grad_output.sum(dim=0)

        # YOUR CODE HERE
        return grad_input, grad_weights, grad_bias
```

```
class MyLinearLayer(nn.Module):
  # You don't modify this layer
  def __init__(self, in_features = 2, out_features = 4):
    super(MyLinearLayer, self).__init__()
    self.weights = nn.Parameter(torch.randn(in_features, out_features))
    self.bias = nn.Parameter(torch.zeros(out_features))
    self.linear_fn = MyLinearFunction.apply

  def forward(self, input):
    return self.linear_fn(input, self.weights, self.bias)
```

Testing Your Implementation

In [13]:
```
class MyLinearNetwork(nn.Module):
  def __init__(self):
    super(MyLinearNetwork, self).__init__()
    self.linear_1 = MyLinearLayer(28 * 28, 128)
    self.sigmoid_fn = MySigmoid.apply
    self.linear_2 = MyLinearLayer(128, 10)
    self.softmax_fn = nn.Softmax(dim=1)

  def forward(self, x):
    size = x.size()
    x = x.reshape(size[0], -1) # Flatten images
    x = self.linear_1(x)
    x = self.sigmoid_fn(x)
    x = self.linear_2(x)
    if self.training == False:
      x = self.softmax_fn(x)
    return x
```

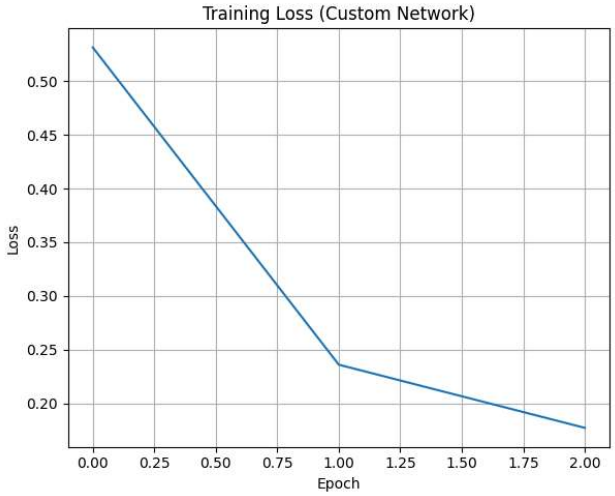
In [14]:
```
cuda = torch.cuda.is_available()
batch_size = 32
train_loader, test_loader = create_data_generator(batch_size)
model = MyLinearNetwork()
print(model)
if cuda:
  model.cuda()
  print("Using GPU for training")
else:
  print("Using CPU for training")
n_epochs = 3
learning_rate = 0.1
optim = torch.optim.SGD(params = model.parameters(), lr = learning_rate, momentum=0.9
loss_fn = nn.CrossEntropyLoss()

losses = []
accuracies = []
batch_times = []
total_start_time = time.time()

model.train()
for epoch in range(1, n_epochs + 1):
  epoch_loss = 0.0
  correct = 0
  total = 0
```

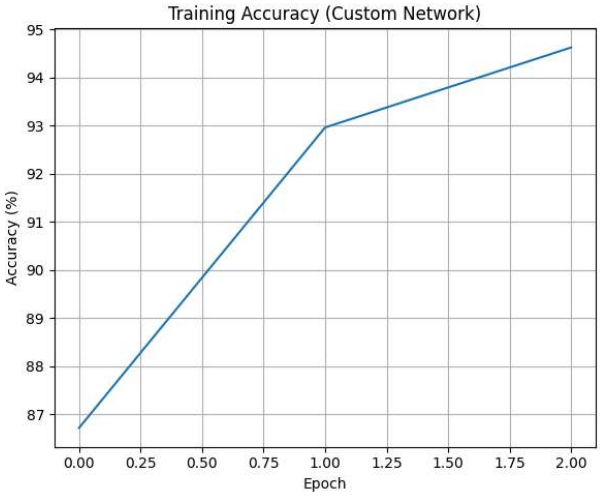
```
  for idx, (images, labels) in enumerate(train_loader):
    # WRITE YOUR CODE TO COMPUTE OUTPUTS, LOSS, ACCURACY, AND OPTIMIZE MODEL
    # outptus = ???
    # loss = ???
    # accuracy = ???
    # optimize the model
    # loss = 0.0
    # accuracy = 0.0
    batch_start_time = time.time()
    if cuda:
      images = images.cuda()
      labels = labels.cuda()
    outputs = model(images)
    loss = loss_fn(outputs, labels)
    optim.zero_grad()
    loss.backward()
    optim.step()
    _, predicted = torch.max(outputs.data, 1)
    total += labels.size(0)
    correct += (predicted == labels).sum().item()
    accuracy = 100 * correct / total
    epoch_loss += loss.item()
    batch_end_time = time.time()
    batch_times.append(batch_end_time - batch_start_time)

    if idx % 100 == 0:
      print("Epoch [%d/%d]. Iter [%d/%d]. Loss: %0.2f. Accuracy: %0.2f" % (epoch, n_e
  epoch_avg_loss = epoch_loss / len(train_loader)
  losses.append(epoch_avg_loss)
  accuracies.append(accuracy)

  print(f"Epoch {epoch} completed. Average loss: {epoch_avg_loss:.4f}, Accuracy: {acc

# Report training time
total_training_time = time.time() - total_start_time
avg_batch_time = np.mean(batch_times)
print(f"Total training time: {total_training_time:.2f} seconds")
print(f"Average time per batch: {avg_batch_time:.4f} seconds")

# Plot training metrics
plt.figure(figsize=(12, 5))
plt.subplot(1, 2, 1)
plt.plot(losses)
plt.title('Training Loss (Custom Network)')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(accuracies)
plt.title('Training Accuracy (Custom Network)')
plt.xlabel('Epoch')
plt.ylabel('Accuracy (%)')
plt.grid(True)
plt.tight_layout()
plt.savefig('custom_network_metrics.png')
plt.show()
```

```
batch_size = 1  # For inference time measurement
train_loader, test_loader = create_data_generator(batch_size)
if cuda:
  model.cuda()
total_accuracy = 0.0
all_preds = []
all_labels = []
inference_times = []
model.eval()
with torch.no_grad():
  for idx, (images, labels) in enumerate(test_loader):
    # WRITE YOUR CODE TO COMPUTE ACCURACY
    # accuracy = ???
    # accuracy = 0.0
    if cuda:
          images = images.cuda()
          labels = labels.cuda()

    start_time = time.time()
    outputs = model(images)
    end_time = time.time()
    inference_times.append(end_time - start_time)

    _, predicted = torch.max(outputs.data, 1)
    accuracy = 100 * (predicted == labels).sum().item() / labels.size(0)

    all_preds.append(predicted.cpu().item())
    all_labels.append(labels.cpu().item())

    total_accuracy += accuracy

    if idx % 2000 == 0:
      print("Iter [%d/%d]. Accuracy: %0.2f" % (idx + 1, len(test_loader), accuracy))

avg_inference_time = np.mean(inference_times)
print("Final Accuracy: %0.2f" % (total_accuracy / len(test_loader)))
print("Average inference time per sample: %.5f seconds" % avg_inference_time)

# Generate confusion matrix
cm = confusion_matrix(all_labels, all_preds)
plt.figure(figsize=(10, 8))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix (Custom Network)')
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.savefig('custom_network_confusion_matrix.png')
plt.show()
```

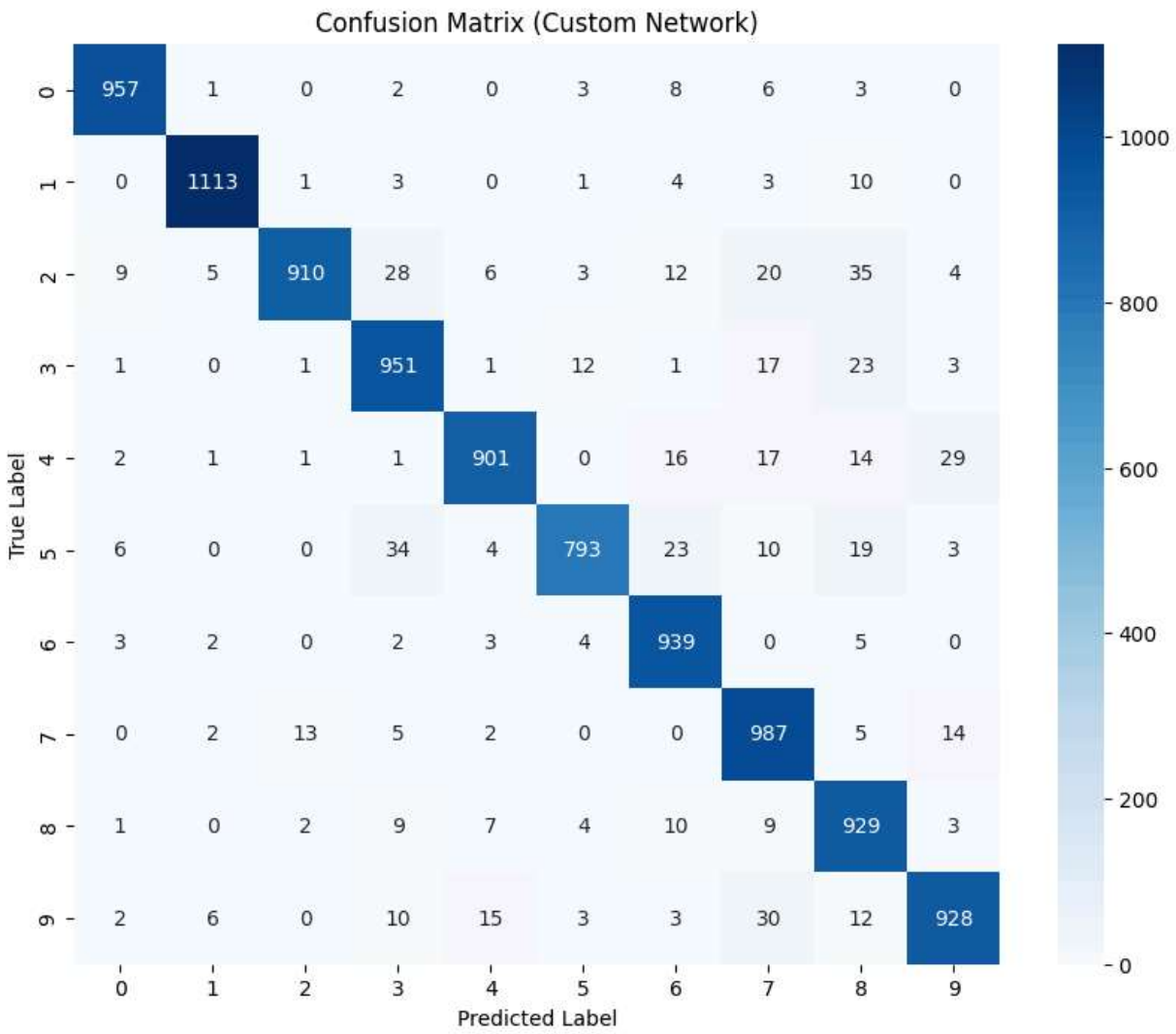
```
MyLinearNetwork(
  (linear_1): MyLinearLayer()
  (linear_2): MyLinearLayer()
  (softmax_fn): Softmax(dim=1)
)
Using CPU for training
Epoch [1/3]. Iter [1/1875]. Loss: 14.56. Accuracy: 6.25
Epoch [1/3]. Iter [101/1875]. Loss: 0.61. Accuracy: 61.42
Epoch [1/3]. Iter [201/1875]. Loss: 0.39. Accuracy: 70.58
Epoch [1/3]. Iter [301/1875]. Loss: 0.10. Accuracy: 74.56
Epoch [1/3]. Iter [401/1875]. Loss: 0.32. Accuracy: 77.35
Epoch [1/3]. Iter [501/1875]. Loss: 0.92. Accuracy: 79.06
Epoch [1/3]. Iter [601/1875]. Loss: 0.39. Accuracy: 80.49
Epoch [1/3]. Iter [701/1875]. Loss: 0.68. Accuracy: 81.52
Epoch [1/3]. Iter [801/1875]. Loss: 0.14. Accuracy: 82.16
Epoch [1/3]. Iter [901/1875]. Loss: 0.79. Accuracy: 82.92
Epoch [1/3]. Iter [1001/1875]. Loss: 0.30. Accuracy: 83.52
Epoch [1/3]. Iter [1101/1875]. Loss: 0.50. Accuracy: 84.01
Epoch [1/3]. Iter [1201/1875]. Loss: 0.57. Accuracy: 84.56
Epoch [1/3]. Iter [1301/1875]. Loss: 0.38. Accuracy: 84.97
Epoch [1/3]. Iter [1401/1875]. Loss: 0.19. Accuracy: 85.33
Epoch [1/3]. Iter [1501/1875]. Loss: 0.27. Accuracy: 85.69
Epoch [1/3]. Iter [1601/1875]. Loss: 0.31. Accuracy: 85.99
Epoch [1/3]. Iter [1701/1875]. Loss: 0.19. Accuracy: 86.29
Epoch [1/3]. Iter [1801/1875]. Loss: 0.17. Accuracy: 86.59
Epoch 1 completed. Average loss: 0.5314, Accuracy: 86.72%
Epoch [2/3]. Iter [1/1875]. Loss: 0.07. Accuracy: 100.00
Epoch [2/3]. Iter [101/1875]. Loss: 0.11. Accuracy: 93.19
Epoch [2/3]. Iter [201/1875]. Loss: 0.40. Accuracy: 92.71
Epoch [2/3]. Iter [301/1875]. Loss: 0.13. Accuracy: 92.86
Epoch [2/3]. Iter [401/1875]. Loss: 0.59. Accuracy: 92.71
Epoch [2/3]. Iter [501/1875]. Loss: 0.47. Accuracy: 92.66
Epoch [2/3]. Iter [601/1875]. Loss: 0.06. Accuracy: 92.58
Epoch [2/3]. Iter [701/1875]. Loss: 0.24. Accuracy: 92.55
Epoch [2/3]. Iter [801/1875]. Loss: 0.12. Accuracy: 92.62
Epoch [2/3]. Iter [901/1875]. Loss: 0.37. Accuracy: 92.74
Epoch [2/3]. Iter [1001/1875]. Loss: 0.32. Accuracy: 92.74
Epoch [2/3]. Iter [1101/1875]. Loss: 0.37. Accuracy: 92.80
Epoch [2/3]. Iter [1201/1875]. Loss: 0.29. Accuracy: 92.81
Epoch [2/3]. Iter [1301/1875]. Loss: 0.15. Accuracy: 92.84
Epoch [2/3]. Iter [1401/1875]. Loss: 0.21. Accuracy: 92.81
Epoch [2/3]. Iter [1501/1875]. Loss: 0.10. Accuracy: 92.86
Epoch [2/3]. Iter [1601/1875]. Loss: 0.11. Accuracy: 92.85
Epoch [2/3]. Iter [1701/1875]. Loss: 0.09. Accuracy: 92.90
Epoch [2/3]. Iter [1801/1875]. Loss: 0.57. Accuracy: 92.92
Epoch 2 completed. Average loss: 0.2360, Accuracy: 92.96%
Epoch [3/3]. Iter [1/1875]. Loss: 0.02. Accuracy: 100.00
Epoch [3/3]. Iter [101/1875]. Loss: 0.05. Accuracy: 94.80
Epoch [3/3]. Iter [201/1875]. Loss: 0.16. Accuracy: 94.98
Epoch [3/3]. Iter [301/1875]. Loss: 0.05. Accuracy: 94.67
Epoch [3/3]. Iter [401/1875]. Loss: 0.45. Accuracy: 94.82
Epoch [3/3]. Iter [501/1875]. Loss: 0.16. Accuracy: 94.83
Epoch [3/3]. Iter [601/1875]. Loss: 0.10. Accuracy: 94.79
Epoch [3/3]. Iter [701/1875]. Loss: 0.05. Accuracy: 94.70
Epoch [3/3]. Iter [801/1875]. Loss: 0.19. Accuracy: 94.74
Epoch [3/3]. Iter [901/1875]. Loss: 0.25. Accuracy: 94.64
Epoch [3/3]. Iter [1001/1875]. Loss: 0.21. Accuracy: 94.61
```

```
Epoch [3/3]. Iter [1101/1875]. Loss: 0.34. Accuracy: 94.54
Epoch [3/3]. Iter [1201/1875]. Loss: 0.21. Accuracy: 94.57
Epoch [3/3]. Iter [1301/1875]. Loss: 0.13. Accuracy: 94.63
Epoch [3/3]. Iter [1401/1875]. Loss: 0.06. Accuracy: 94.66
Epoch [3/3]. Iter [1501/1875]. Loss: 0.47. Accuracy: 94.65
Epoch [3/3]. Iter [1601/1875]. Loss: 0.07. Accuracy: 94.65
Epoch [3/3]. Iter [1701/1875]. Loss: 0.01. Accuracy: 94.65
Epoch [3/3]. Iter [1801/1875]. Loss: 0.11. Accuracy: 94.64
Epoch 3 completed. Average loss: 0.1772, Accuracy: 94.62%
Total training time: 27.28 seconds
Average time per batch: 0.0014 seconds
```






```
Iter [1/10000]. Accuracy: 100.00
Iter [2001/10000]. Accuracy: 100.00
Iter [4001/10000]. Accuracy: 100.00
Iter [6001/10000]. Accuracy: 0.00
Iter [8001/10000]. Accuracy: 100.00
Final Accuracy: 94.08
Average inference time per sample: 0.00023 seconds
```


Confusion Matrix (Custom Network)
True Label
Predicted Label
0 957 1 0 2 0 3 8 6 3 0
1 0 1113 1 3 0 1 4 3 10 0
2 9 5 910 28 6 3 12 20 35 4
3 1 0 1 951 1 12 1 17 23 3
4 2 1 1 1 901 0 16 17 14 29
5 6 0 0 34 4 793 23 10 19 3
6 3 2 0 2 3 4 939 0 5 0
7 0 2 13 5 2 0 0 987 5 14
8 1 0 2 9 7 4 10 9 929 3
9 2 6 0 10 15 3 3 30 12 928
0 1 2 3 4 5 6 7 8 9
1000
800
600
400
200
0