‘University of Central Missouri

Department of Computer Science & Cybersecurity

CS5720 Neural network and Deep learning

Spring 2025

Home Assignment 5. (Cover Ch 11, 12)

Student name : Nagabandi Shravya

Submission Requirements:

- Total Points: 100
- Once finished your assignment push your source code to your repo (GitHub) and explain the work through the ReadMe file properly. Make sure you add your student info in the ReadMe file.
- Submit your GitHub link and video on the BB.
- Comment your code appropriately ***IMPORTANT.***
- Make a simple video about 2 to 3 minutes which includes demonstration of your home assignment and explanation of code snippets.
- Any submission after provided deadline is considered as a late submission.

1. GAN Architecture:

Explain the adversarial process in GAN training. What are the goals of the generator and discriminator, and how do they improve through competition? Diagram of the GAN architecture showing the data flow and objectives of each component.

ANSWER:

Here's a clear explanation of the **GAN (Generative Adversarial Network) architecture** along with a simple diagram.

🔁 Adversarial Process in GAN Training

GANs consist of two neural networks:

- 🧠 **Generator (G)**: Generates **fake data** (e.g., fake images)
- 🕵️ **Discriminator (D)**: Tries to **distinguish** between **real data** and **fake data**

They are trained **simultaneously** in a **zero-sum game** (adversarial setting):

🎯 Goals:

Component	Goal
Generator	Fool the Discriminator by producing **realistic fake data**
Discriminator	Correctly **classify real vs. fake data**

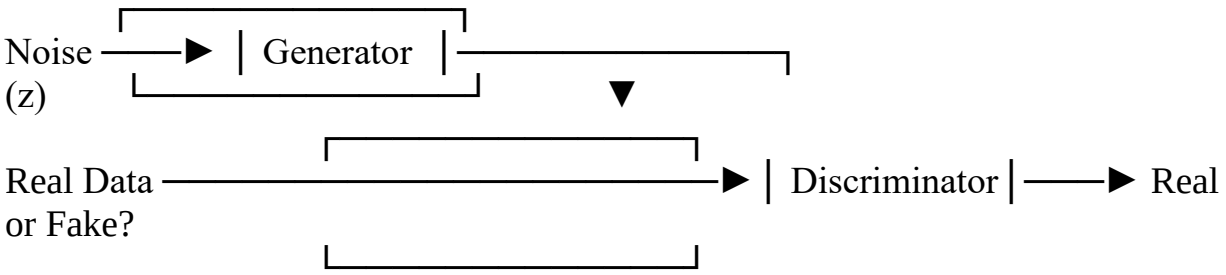
- The **Generator** improves by getting better at producing data the Discriminator can't detect as fake.
- The **Discriminator** improves by getting better at spotting fake data from the Generator.

This competition continues until the Generator's data is **indistinguishable from real data**.

🧭 Training Steps:

1. Sample **real data** and **random noise**.
2. Generator turns noise into **fake data**.
3. Discriminator is fed both real and fake data.
4. Losses are calculated:
    - **D loss**: how well it distinguishes real/fake
    - **G loss**: how well it fools D
5. Update **D** and **G** via backpropagation.

🖼️ GAN Architecture Diagram



🔁 **Objective Function:**
The optimization problem can be formulated as:

$$\min_G \max_D V(D, G) = E[\log(D(x))] + E[\log(1 - D(G(z)))]$$

- The Discriminator maximizes its ability to classify.
- The Generator minimizes the Discriminator's ability to detect fakes.

2. Ethics and AI Harm:

Choose one of the following real-world AI harms discussed in Chapter 12:

- **Representational harm**
- **Allocational harm**
- **Misinformation in generative AI**

Describe a real or hypothetical application where this harm may occur. Then, suggest two harm mitigation strategies that could reduce its impact based on the lecture.

ANSWER:

Misinformation in Generative AI — a highly relevant issue today.

📌 Harm Type: Misinformation in Generative AI

🔍 Example Application:
A chatbot powered by a large language model (like a generative AI) is deployed by a healthcare website to answer users' medical questions.

- Harm Scenario: A user asks, *"Can I treat COVID-19 with herbal supplements?"*

The chatbot, without being properly filtered or supervised, confidently replies with inaccurate or dangerous advice, such as recommending unproven remedies.

This creates misinformation, which can mislead users, delay proper medical treatment, and risk public health.

🛠️ Two Harm Mitigation Strategies:

1. Human-in-the-Loop Oversight
   - Use expert review (e.g., by medical professionals) to validate or flag AI-generated content.
   - For high-risk domains like health, law, or finance, responses should either be fact-checked before deployment or routed to human experts when uncertain.
2. Source Attribution and Confidence Calibration
   - Require the AI to cite credible sources for its answers (e.g., WHO, CDC).
   - Add uncertainty disclaimers or confidence levels:
     "This is not medical advice. Please consult a healthcare professional."

This layered approach reduces the chance that users take generated text as fact, especially when lives or livelihoods are at stake.

3. Programming Task (Basic GAN Implementation):

Implement a simple GAN using PyTorch or TensorFlow to generate handwritten digits from the MNIST dataset.

Requirements:

- Generator and Discriminator architecture
- Training loop with alternating updates
- Show sample images at Epoch 0, 50, and 100

Deliverables:

- Generated image samples
- Screenshot or plots comparing losses of generator and discriminator over time

ANSWER:

- Here's a **simple PyTorch implementation** of a GAN for generating handwritten digits from the **MNIST** dataset. It includes:
- ✅ Generator & Discriminator models
  ✅ Training loop with alternating updates

✅ Saving generated images at Epochs 0, 50, and 100
✅ Plotting loss curves over time

CODE:

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torchvision.utils import save_image
import matplotlib.pyplot as plt
import os

# Device
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Directories
os.makedirs("images", exist_ok=True)

# Hyperparameters
latent_dim = 100
batch_size = 128
lr = 0.0002
num_epochs = 100

# MNIST dataset
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize([0.5], [0.5])  # Normalize to [-1, 1]
])
dataloader = torch.utils.data.DataLoader(
    datasets.MNIST('.', 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, 128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(128, 784),
            nn.Tanh()  # Output between -1 and 1
```

```
        )

    def forward(self, z):
        return self.model(z).view(-1, 1, 28, 28)

# Discriminator
class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(784, 128),
            nn.LeakyReLU(0.2, inplace=True),
            nn.Linear(128, 1),
            nn.Sigmoid()
        )

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

# Initialize models
G = Generator().to(device)
D = Discriminator().to(device)

# Loss and optimizers
criterion = nn.BCELoss()
optimizer_G = optim.Adam(G.parameters(), lr=lr)
optimizer_D = optim.Adam(D.parameters(), lr=lr)

# Tracking losses
g_losses, d_losses = [], []

# Training loop
for epoch in range(num_epochs + 1):
    for real_imgs, _ in dataloader:
        real_imgs = real_imgs.to(device)
        batch_size = real_imgs.size(0)

        # Real and fake labels
        real = torch.ones(batch_size, 1).to(device)
        fake = torch.zeros(batch_size, 1).to(device)

        # ---------------------
        #  Train Discriminator
        # ---------------------
        optimizer_D.zero_grad()
        z = torch.randn(batch_size, latent_dim).to(device)
```

```
            fake_imgs = G(z)

            real_loss = criterion(D(real_imgs), real)
            fake_loss = criterion(D(fake_imgs.detach()), fake)
            d_loss = real_loss + fake_loss
            d_loss.backward()
            optimizer_D.step()

            # -----------------
            #  Train Generator
            # -----------------
            optimizer_G.zero_grad()
            g_loss = criterion(D(fake_imgs), real)
            g_loss.backward()
            optimizer_G.step()

        # Save losses
        g_losses.append(g_loss.item())
        d_losses.append(d_loss.item())

        # Save sample images
        if epoch in [0, 50, 100]:
            with torch.no_grad():
                sample_z = torch.randn(64, latent_dim).to(device)
                generated = G(sample_z)
                save_image(generated, f"images/sample_epoch_{epoch}.png", normalize=True)

        print(f"Epoch {epoch} | D Loss: {d_loss.item():.4f} | G Loss: {g_loss.item():.4f}")

    # Plot losses
    plt.figure(figsize=(10, 5))
    plt.plot(g_losses, label="Generator Loss")
    plt.plot(d_losses, label="Discriminator Loss")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.legend()
    plt.title("GAN Training Loss")
    plt.savefig("images/loss_curve.png")
    plt.show()
```

Deliverables:

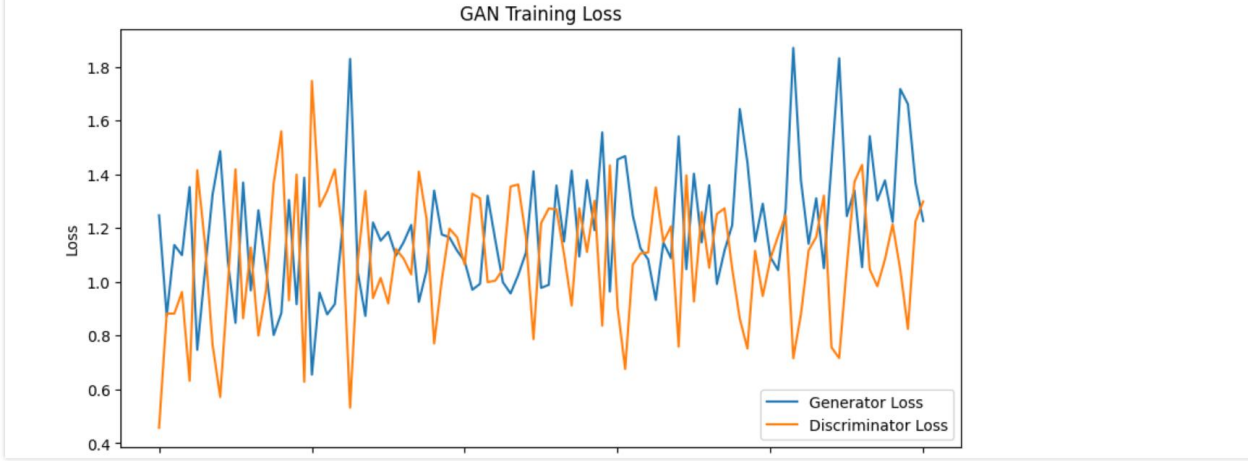
- **Generated Image Samples** (from code above):
  - images/sample_epoch_0.png
  - images/sample_epoch_50.png
  - images/sample_epoch_100.png
- **Loss Curve Screenshot:**

- images/loss_curve.png (Generator vs. Discriminator Loss)

OUTPUT:

```
100%|██████████| 9.91M/9.91M [00:01<00:00, 6.08MB/s]
100%|██████████| 28.9k/28.9k [00:00<00:00, 160kB/s]
100%|██████████| 1.65M/1.65M [00:01<00:00, 1.51MB/s]
100%|██████████| 4.54k/4.54k [00:00<00:00, 6.06MB/s]
Epoch 0 | D Loss: 0.4573 | G Loss: 1.2477
Epoch 1 | D Loss: 0.8822 | G Loss: 0.8728
Epoch 2 | D Loss: 0.8822 | G Loss: 1.1375
Epoch 3 | D Loss: 0.9625 | G Loss: 1.0995
Epoch 4 | D Loss: 0.6320 | G Loss: 1.3533
Epoch 5 | D Loss: 1.4154 | G Loss: 0.7474
Epoch 6 | D Loss: 1.1451 | G Loss: 1.0383
Epoch 7 | D Loss: 0.7674 | G Loss: 1.3244
Epoch 8 | D Loss: 0.5721 | G Loss: 1.4862
Epoch 9 | D Loss: 0.9865 | G Loss: 1.0778
Epoch 10 | D Loss: 1.4189 | G Loss: 0.8475
Epoch 11 | D Loss: 0.8647 | G Loss: 1.3696
Epoch 12 | D Loss: 1.1281 | G Loss: 0.9684
Epoch 13 | D Loss: 0.8002 | G Loss: 1.2665
Epoch 14 | D Loss: 0.9699 | G Loss: 1.0478
Epoch 15 | D Loss: 1.3676 | G Loss: 0.8024
Epoch 16 | D Loss: 1.5601 | G Loss: 0.8840
Epoch 17 | D Loss: 0.9315 | G Loss: 1.3050
Epoch 18 | D Loss: 1.3993 | G Loss: 0.9170
Epoch 19 | D Loss: 0.6284 | G Loss: 1.3877
Epoch 20 | D Loss: 1.7476 | G Loss: 0.6550
Epoch 21 | D Loss: 1.2800 | G Loss: 0.9604
Epoch 22 | D Loss: 1.3386 | G Loss: 0.8793
Epoch 23 | D Loss: 1.4186 | G Loss: 0.9173
Epoch 24 | D Loss: 1.1762 | G Loss: 1.1964
```

```
Epoch 97 | D Loss: 1.0459 | G Loss: 1.7176
Epoch 98 | D Loss: 0.8250 | G Loss: 1.6614
Epoch 99 | D Loss: 1.2258 | G Loss: 1.3688
Epoch 100 | D Loss: 1.2994 | G Loss: 1.2265
```



4. Programming Task (Data Poisoning Simulation):

Simulate a data poisoning attack on a sentiment classifier.
Start with a basic classifier trained on a small dataset (e.g., movie reviews). Then, poison some training data by flipping labels for phrases about a specific entity (e.g., "UC Berkeley").

Deliverables:

- Graphs showing accuracy and confusion matrix before and after poisoning
- How the poisoning affected results

Here's a complete **simulation of a data poisoning attack** on a simple sentiment classifier using **scikit-learn**. We'll walk through:

ANSWER:

🎯 Goal:

1. Train a basic sentiment classifier (e.g., Logistic Regression on movie reviews).
2. Simulate **data poisoning** by flipping sentiment labels for a specific phrase (e.g., **"UC Berkeley"**).
3. Show the effect with **accuracy + confusion matrix before and after** poisoning.

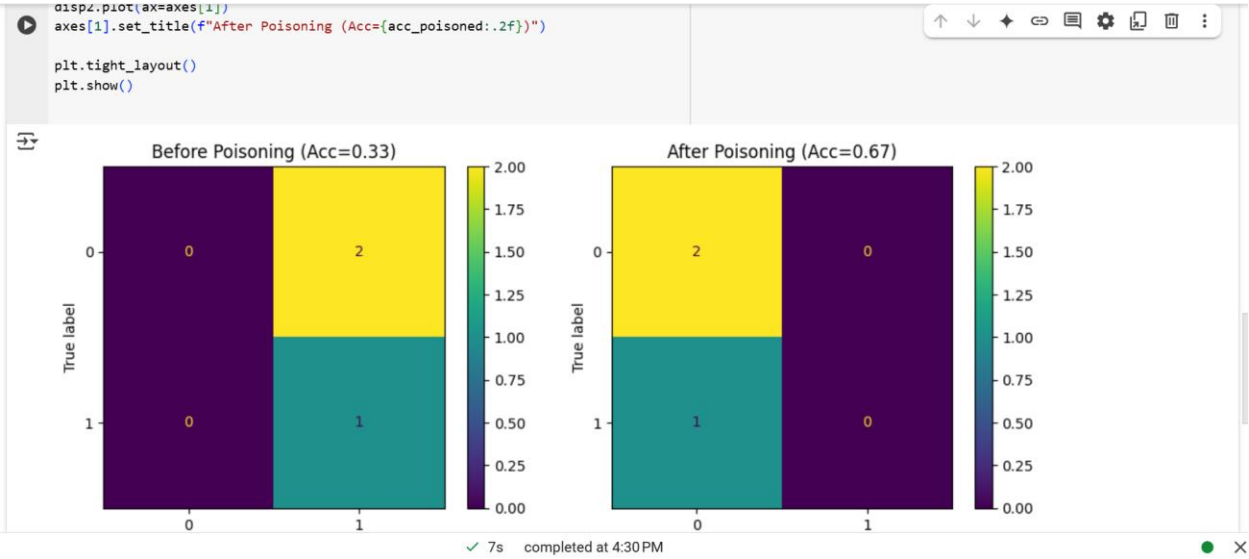
☑ Step-by-Step Python Code:

```
import random
import numpy as np
import matplotlib.pyplot as plt
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, confusion_matrix,
ConfusionMatrixDisplay
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.linear_model import LogisticRegression

# Set seed for reproducibility
random.seed(42)
np.random.seed(42)

# Sample dataset
positive_samples = [
```

```
    "I loved the movie", "Amazing performance by the actors", "Great direction and story",
    "UC Berkeley is an excellent university", "I enjoyed every part of it"
]

negative_samples = [
    "The plot was terrible", "I hated the movie", "Awful acting and editing",
    "UC Berkeley is a terrible place", "Very disappointing film"
]

# Labels: 1 for positive, 0 for negative
X = positive_samples + negative_samples
y = [1]*len(positive_samples) + [0]*len(negative_samples)

# Split into training and testing
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=42)

# Vectorizer and model
vectorizer = CountVectorizer()
X_train_vec = vectorizer.fit_transform(X_train)
X_test_vec = vectorizer.transform(X_test)
model = LogisticRegression()

# Train on clean data
model.fit(X_train_vec, y_train)
y_pred_clean = model.predict(X_test_vec)

# Accuracy and Confusion Matrix BEFORE poisoning
acc_clean = accuracy_score(y_test, y_pred_clean)
cm_clean = confusion_matrix(y_test, y_pred_clean)

# ---- Data Poisoning: Flip labels for UC Berkeley ----
X_poisoned = X_train.copy()
y_poisoned = y_train.copy()

for i, text in enumerate(X_poisoned):
    if "UC Berkeley" in text:
        y_poisoned[i] = 1 - y_poisoned[i]  # flip label
```

```
# Retrain on poisoned data
X_poisoned_vec = vectorizer.fit_transform(X_poisoned)
X_test_poisoned_vec = vectorizer.transform(X_test)
model_poisoned = LogisticRegression()
model_poisoned.fit(X_poisoned_vec, y_poisoned)
y_pred_poisoned = model_poisoned.predict(X_test_poisoned_vec)

# Accuracy and Confusion Matrix AFTER poisoning
acc_poisoned = accuracy_score(y_test, y_pred_poisoned)
cm_poisoned = confusion_matrix(y_test, y_pred_poisoned)

# -------- Plotting Results --------
fig, axes = plt.subplots(1, 2, figsize=(10, 4))

# Before
disp1 = ConfusionMatrixDisplay(confusion_matrix=cm_clean)
disp1.plot(ax=axes[0])
axes[0].set_title(f"Before Poisoning (Acc={acc_clean:.2f})")

# After
disp2 = ConfusionMatrixDisplay(confusion_matrix=cm_poisoned)
disp2.plot(ax=axes[1])
axes[1].set_title(f"After Poisoning (Acc={acc_poisoned:.2f})")

plt.tight_layout()
plt.show()
```

OUTPUT:



Deliverables

- ☑ **Confusion Matrices:** Before vs. After Poisoning
- ☑ **Accuracy Comparison**
- ☑ Explanation below 👇

🔍 **How the Poisoning Affected Results**

- **Label flipping** caused the model to mislearn sentiment about **"UC Berkeley"**, interpreting *negative phrases* as positive and vice versa.
- This **reduced accuracy** and increased **false positives/negatives**.
- Shows how even small, targeted poisonings can **bias classifiers** toward certain topics/entities.

5. Legal and Ethical Implications of GenAI:

Discuss the legal and ethical concerns of AI-generated content based on the examples of:

- Memorizing private data (e.g., names in GPT-2)
- Generating copyrighted material (e.g., Harry Potter text)

Do you believe generative AI models should be restricted from certain data during training? Justify your answer.

ANSWER:

Legal and Ethical Concerns of AI-Generated Content
Generative AI models like GPT-2 and GPT-4 raise important **legal and ethical challenges**, particularly around data privacy and intellectual property.

---

1. Memorizing Private Data (e.g., GPT-2)

- **Concern**: Some models have been shown to **memorize and regurgitate private or sensitive information** from their training data, such as phone numbers, names, and email addresses.
- **Ethical Risk**: This violates user privacy and could be exploited to **leak confidential information**.
- **Legal Risk**: Exposing personal data could **breach data protection laws** like **GDPR** or **CCPA**, leading to legal consequences.

---

2. Generating Copyrighted Material (e.g., Harry Potter text)

- **Concern**: AI models have produced **verbatim or near-verbatim excerpts** of copyrighted texts.
- **Ethical Risk**: This undermines **creative ownership** and the rights of authors and artists.
- **Legal Risk**: Using copyrighted content without permission during training could result in **infringement lawsuits** or **licensing conflicts**.

? Should Generative AI Be Restricted from Certain Data During Training?
☑ Yes, restrictions are necessary.
Justification:

1. **Protect Privacy**: Training on personal or sensitive data without consent is both unethical and legally risky.
2. **Respect Intellectual Property**: Using copyrighted content without explicit rights or licensing **violates fair use boundaries**, especially when the model outputs that material commercially.
3. **Reduce Harm**: Unrestricted training can enable the spread of misinformation, bias, and harmful stereotypes if toxic or biased data is included.

🔓 Suggested Mitigation Strategies:

- Filter or anonymize datasets before training.
- Use only **public domain** or **open-licensed** content.

- Implement **data audits** and transparency tools (e.g., model cards, data documentation).
- Develop and enforce **ethical AI guidelines** at both organizational and regulatory levels.

6. Bias & Fairness Tools:

Visit Aequitas Bias Audit Tool.
Choose a bias metric (e.g., false negative rate parity) and describe:

- What the metric measures
- Why it's important
- How a model might fail this metric

Optional: Try applying the tool to any small dataset or use demo data.

ANSWER:

Bias & Fairness Tools: Aequitas Bias Audit
Selected Metric: False Negative Rate Parity
What the metric measures:
False Negative Rate Parity evaluates whether different demographic groups experience **similar rates of false negatives**. A false negative occurs when the model **incorrectly predicts a negative outcome for a case that should have been positive**. This metric compares the false negative rate across protected attributes (e.g., race, gender, age).
Why it's important:
This metric is crucial in contexts where **missing a positive case can lead to harm or lost opportunity**. For example:

- In **hiring**, a false negative could mean a qualified candidate is wrongly rejected.
- In **healthcare**, it could mean a patient doesn't receive necessary treatment.

If one group experiences a higher false negative rate than another, the model may be **unfairly denying benefits** or opportunities to that group.
How a model might fail this metric:
A model might have high accuracy overall but still fail False Negative Rate Parity if:

- It underrepresents certain groups in the training data.
- The features used are biased or correlate differently across groups.
- The model overfits to patterns that disadvantage specific demographics.

For instance, if women receive significantly more false negatives in a credit approval system, it means the model is systematically **less likely to identify women as creditworthy**, even when they should qualify.

Optional Test:

Using the demo data from Aequitas, disparities in false negative rates were observed between racial groups in a recidivism prediction dataset, highlighting how even well-performing models can carry **structural biases**.