CS 5330: Pattern Recognition and Computer Vision

Northeastern University

Lab 11: Generative AI

* *Contributed by Fall 2024 TAs: Byunghyun Ko, Yihan Wang and Taiwei Cui*

Generative AI

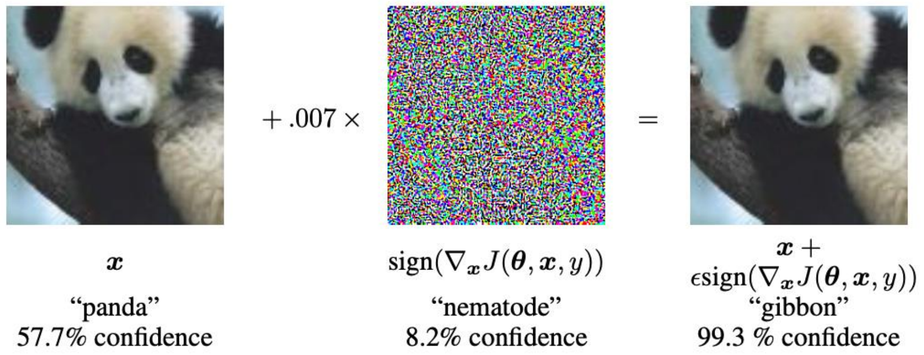
1. Introduction to Generative AI
2. Adversarial Images
3. Generative Adversarial Networks
4. Diffusion Models

Introduction to Generative AI

- Generative AI: involves models and techniques that create new content, such as images, videos, or text.
- Generative AI has applications in:
  - **Digital Art**: Generating creative artworks.
  - **Content Creation**: Automating design and animation processes.
  - **Medical Imaging**: Synthesizing CT or MRI scans for research.
  - And many more!

Adversarial Images

- Adversarial Images: images altered slightly to trick AI models
  - An example of this would be misclassifying a “stop” sign as “yield”
- The purpose of adversarial images is to test model robustness and improve security against adversarial attacks
- A technique to accomplish this is to add noise
  - For example, through the FGSM (Fast Gradient Sign) method



https://medium.com/@zachariaharungeorge/a-deep-dive-into-the-fast-gradient-sign-method-611826e34865

FGSM Example

- **cv2.addWeighted(src1, alpha, src2, beta, gamma)**:
  - src1: First input array (e.g., original image).
  - alpha: Weight of the first array.
  - src2: Second input array (e.g., adversarial noise).
  - beta: Weight of the second array.
  - gamma: Scalar added to each sum.
- This function combines two images with specified weights and adds a scalar value.

FGSM Example

Code

```
import cv2
import numpy as np

# Load original image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)

# Create noise
noise = np.random.normal(0, 25, image.shape).astype('uint8')

# Add noise to create adversarial image
adversarial_image = cv2.addWeighted(image, 1.0, noise, 0.1, 0)

cv2.imshow('Adversarial Image', adversarial_image)
cv2.waitKey(0)
```

cv2.addWeighted(src1, alpha, src2, beta, gamma):

- src1: First input array (e.g., original image).
- alpha: Weight of the first array.
- src2: Second input array (e.g., adversarial noise).
- beta: Weight of the second array.
- gamma: Scalar added to each sum.

Generative Adversarial Networks

- GANs (Generative Adversarial Networks) consist of two models:
  - Generator: Produces fake images from random noise.
  - Discriminator: Distinguishes between real and fake images.
- The models are trained together in a competitive setting.
- **Training Objective**:
  - Generator tries to "fool" the discriminator by producing realistic samples.
  - Discriminator learns to better detect fake images.
  - Adversarial loss ensures both networks improve simultaneously.

Generative Adversarial Networks

- Generator
  - Creates fake images from random noise

Code

```
generator = nn.Sequential(
  nn.Linear(latent_dim, 128),  # Fully connected layer: maps input noise to 128 features
  nn.ReLU(),                   # Activation function: introduces non-linearity
  nn.Linear(128, 256),         # Expands the feature space to 256 dimensions
  nn.ReLU(),                   # Another activation function to make learning non-linear
  nn.Linear(256, 28 * 28),     # Output layer: maps features to 784 pixels (28x28 image)
  nn.Tanh()                    # Scales output pixel values to the range [-1, 1]
)
```

Generative Adversarial Networks

- Discriminator
  - Classifies images as real (1) or fake(0)

Code

```
discriminator = nn.Sequential(
    nn.Linear(28 * 28, 256),  # Input layer: takes flattened 28x28 image (784 pixels)
    nn.LeakyReLU(0.2),       # Activation: allows small gradients for negative inputs (slope = 0.2)
    nn.Linear(256, 128),     # Hidden layer: reduces features to 128 dimensions
    nn.LeakyReLU(0.2),       # Another activation for non-linearity
    nn.Linear(128, 1),       # Output layer: maps features to a single value (real/fake score)
    nn.Sigmoid()             # Activation: outputs probability [0, 1] (real = 1, fake = 0)
)
```

Generative Adversarial Networks

- Training Loop
    - Alternates between training the discriminator and generator
- The code below is for **discriminator training**

Code

```
# Calculate loss for real images
real_loss = criterion(discriminator(real_imgs), real_labels)  # How well the discriminator classifies real images as real
# Calculate loss for fake images
fake_loss = criterion(discriminator(fake_imgs.detach()), fake_labels)  # How well the discriminator classifies fake images as fake
# Total loss for the Discriminator
d_loss = real_loss + fake_loss  # Combine losses for real and fake images
# Backpropagation to compute gradients
d_loss.backward()  # Update weights to minimize the Discriminator's loss
# Apply the weight updates
optimizer_D.step()  # Update Discriminator parameters using the computed gradients
```

Generative Adversarial Networks

- **Generator Training**
  - Encourages the generator to make images that fool the discriminator

Code

```
# Get the discriminator's predictions for the fake images
fake_preds = discriminator(fake_imgs)  # Check how "real" the discriminator thinks the fake images are

# Calculate the Generator's loss
g_loss = criterion(fake_preds, real_labels)  # Goal: Fool the discriminator into thinking fake images are real

# Backpropagation to compute gradients for the Generator
g_loss.backward()  # Update Generator's weights to minimize its loss

# Apply the weight updates
optimizer_G.step()  # Update Generator parameters using the computed gradients
```

Diffusion Models

- Diffusion models add noise to data and then learn the reverse the process
- We use diffusion models in generative AI because they generate **highly realistic images** and **handle complex data distributions**
  - Diffusion models handle complex data distributions better than GANs
- Two processes in diffusion models
  - Forward: gradually corrupt an image by adding small amounts of Gaussian noise at each step
  - Reverse: learn to predict and remove noise using a neural network

Diffusion Models

- Adding Noise (**forward**)

Code

```
# Load and preprocess the image
image = cv2.imread('image.jpg', cv2.IMREAD_GRAYSCALE)
image = cv2.resize(image, (128, 128)) / 255.0  # Normalize to range [0, 1]

# Add Gaussian noise in steps
for i in range(5):  # 5 noise addition steps
    noise = np.random.normal(0, 0.1 * (i + 1), image.shape)  # Increase noise level
    noisy_image = np.clip(image + noise, 0, 1)  # Ensure values are within [0, 1]
    cv2.imshow(f"Step {i+1}", (noisy_image * 255).astype('uint8'))  # Show noisy image
cv2.waitKey(0)  # Wait for user input to close the windows
```

np.random.normal(loc, scale, size):
- Generates Gaussian noise.
- loc: Mean of the distribution (set to 0 for standard Gaussian).
- scale: Standard deviation (controls noise intensity, increases with step).
- size: Shape of the noise array (same as the image).

np.clip(array, min, max):
- Ensures pixel values remain in the valid range [0, 1].
- array: Input array.
- min: Minimum allowed value.
- max: Maximum allowed value.

Diffusion Models

- Denoising (**reverse**)
  - This step mimics the reverse diffusion process by reducing noise using Gaussian smoothing.

Code

```
# Denoising using GaussianBlur
denoised_image = cv2.GaussianBlur(noisy_image, (5, 5), 0)


cv2.imshow("Denoised Image", (denoised_image * 255).astype('uint8'))
cv2.waitKey(0)
```

cv2.GaussianBlur(src, ksize, sigmaX):

- Smoothens the image by applying a Gaussian filter.
- src: Input image (e.g., noisy_image).
- ksize: Kernel size for the filter (e.g., (5, 5) for a 5x5 filter).
- sigmaX: Standard deviation in the X direction (0 lets OpenCV compute it automatically).
- Returns: Blurred (denoised) image.