

Week 13 – Responsible AI

Lecture Transcript with Comprehensive Explanations

Goal and Learning Objectives

- Understand the concepts of trust and governance specific to generative AI.
- Identify the main mitigation tools that help enforce responsible AI practices.
- Relate legal and policy frameworks to practical AI development workflows.

Running Environment

- Learning management system: Moodle / Canvas.
- Browser: Chrome or Firefox (latest version recommended).
- No special software installation required for this conceptual lecture.

Data Preparation

The week focuses on policy rather than data engineering, therefore no data-preprocessing code is needed. Nevertheless, a typical workflow for generative AI projects would involve:

1. Collect raw data (images, text, audio).
2. Clean and de-identify personal information.
3. Split into training, validation, and test sets.
4. Load with a `DataLoader` and optionally visualise samples.

Code Outline

```
def visualize_batch(batch):
    """
    Display a grid of images from a batch.
    Parameters
    -----
    batch : torch.Tensor
        Tensor of shape (B, C, H, W) where B is batch size.
    Returns
    -----
    None
    """
    # 1. Convert tensor to NumPy
    # 2. Arrange images in a grid
    # 3. Use matplotlib to show the grid
    pass
```

- **Why visualise?** It helps spot bias, artifacts, or privacy leaks before model training.

1 Generator Architecture

Definition

Generator – A neural network that transforms random noise z into synthetic data that resembles the real distribution.

- **Analogy:** Think of a chef who receives a random assortment of ingredients (z) and follows a recipe (the network) to create a dish that looks like a real meal.
- **Purpose:** Produce data for the discriminator to evaluate.

Crucial Insight

Why use a linear layer first? It projects the low-dimensional noise vector z (e.g., $z_dim = 100$) into a higher-dimensional space that can be reshaped into feature maps.

Key Takeaway

Batch normalization stabilises training by keeping activations centred and with unit variance.

Remember

LeakyReLU prevents dead neurons by allowing a small gradient when the unit is not active.

Generator Workflow Diagram

($1 \times 28 \times 28$);
[- i] (in) – (lin); [- i] (lin) – (bn); [- i] (bn) – (act1); [- i] (act1) – (act2); [- i] (act2) – (act3); [- i] (act3) – (final); [- i] (final) – (out);

Remember

Dimensions are annotated to make it clear how the vector expands to an image-sized tensor.

2 Discriminator Architecture

Definition

Discriminator – A neural network that receives either a real sample x or a generated sample \hat{x} and outputs a probability that the input is real.

- **Analogy:** A customs officer who inspects luggage (input) and decides whether it comes from a trusted source.
- **Purpose:** Provide feedback to the generator through the loss signal.

Crucial Insight

Why use dropout? It forces the discriminator to rely on multiple features, reducing over-fitting to the current generator's artifacts.

Key Takeaway

LeakyReLU is preferred over ReLU because the discriminator must stay sensitive to subtle differences in fake images.

Discriminator Workflow Diagram

Remember

The dimension reduction from 784 to 1 shows how the discriminator compresses information into a single realism score.

3 Training Mechanism

Training Loop Diagram

Crucial Insight

Why detach fake images for the discriminator? It prevents gradients from flowing back into the generator during the discriminator update, ensuring each network is trained independently in its step.

Loss Functions and Optimizers

- **Binary Cross Entropy (BCE) Loss** – Measures the distance between predicted probability and ground-truth label (real = 1, fake = 0).
- **Adam Optimizer** – Adaptive learning-rate method, widely used for both generator and discriminator because it converges quickly on noisy gradients.

Crucial Insight

Why BCE instead of MSE? BCE directly models the probability of realism, which aligns with the adversarial objective.

Key Takeaway

Learning rate $lr = 2 \times 10^{-4}$ is a common starting point; too high destabilises the delicate balance between networks.

4 Implementation Details

1. Hyper-parameters

- Noise dimension $z_dim = 100$.
- Batch size $batch_size = 64$.
- Learning rate $lr = 2 \times 10^{-4}$.
- Number of epochs $epochs = 50$.

2. Architecture Choices

- Linear layers are sufficient for simple MNIST-style images.
- BatchNorm improves gradient flow for the generator.
- LeakyReLU prevents dead units in the discriminator.

Crucial Insight

Choosing a shallow network reduces over-fitting on small datasets but may limit expressive power for high-resolution images.

5 Performance Monitoring

- Plot generator and discriminator loss curves after each epoch.
- Visualise a grid of generated images to detect mode collapse.
- Use the `visualize_batch` function (see Section “Data Preparation”) to inspect real vs. fake samples.

Code Outline

```
def plot_losses(g_losses, d_losses):
    """
    Plot generator and discriminator losses over training.
    Parameters
    -----
    g_losses : list of float
        Generator loss per iteration.
    d_losses : list of float
        Discriminator loss per iteration.
    Returns
    -----
    None
    """
    plt.figure(figsize=(10,5))
    plt.plot(g_losses, label='Generator')
    plt.plot(d_losses, label='Discriminator')
    plt.xlabel('Iteration')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()
```

6 Sample Generation (Inference)

- Switch the generator to evaluation mode: `generator.eval()`.
- Sample a new noise vector $z \sim \mathcal{N}(0, 1)$.
- Pass z through the generator to obtain $\hat{x} = G(z)$.
- Post-process (e.g., denormalise) and display the image.

Code Outline

```
def generate_samples(generator, n=16, z_dim=100):  
    generator.eval()  
    with torch.no_grad():  
        z = torch.randn(n, z_dim, device=device)  
        fake = generator(z)  
    return fake
```

Remember

Running in `torch.no_grad()` mode saves memory and speeds up inference because gradients are not needed.

7 Common Implementation Errors

- Forgetting to call `detach()` on fake images before the discriminator step – leads to simultaneous updates and training collapse.
- Using ReLU in the discriminator – can cause dead neurons when many inputs are negative.
- Mismatching dimensions between generator output and discriminator input – triggers shape errors.
- Not resetting gradients with `optimizer.zero_grad()` – results in gradient accumulation and unstable learning.
- Using a learning rate that is too large – causes oscillations and failure to converge.

Key Takeaway

Always verify tensor shapes with `tensor.shape` before forwarding through a layer.

8 Extensions and Advanced Topics

1. **Conditional GANs** – Incorporate class labels into both generator and discriminator to control the generated output.
2. **Wasserstein GAN (WGAN)** – Replaces BCE loss with Earth-Mover distance; improves training stability.
3. **Spectral Normalisation** – Constrains the Lipschitz constant of the discriminator for better convergence.

4. **Progressive Growing** – Starts with low-resolution images and gradually increases resolution; useful for high-quality synthesis.

Crucial Insight

Each extension addresses a specific failure mode of the vanilla GAN: mode collapse, gradient vanishing, or training instability.