CE6146
Introduction to Deep Learning
Review

Chia-Ru Chung

Department of Computer Science and Information Engineering

National Central University

2023/12/07

20231130 Exercise

1. **B**	2. **A**	3. **B**	4. **D**	5. **B**
6. **A**	7. **C**	8. **B**	9. **C**	10. **B**
11. **B**	12. **D**	13. **D**	14. **B**	15. **C**
16. **A**	17. **D**	18. **C**	19. **B**	20. **C**

Assignment 2 – Task 3

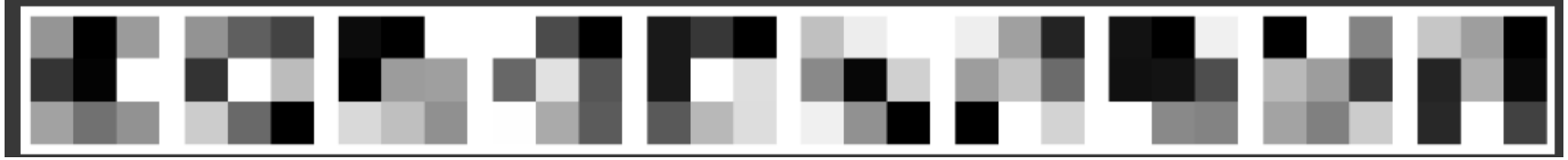
- Implementation:
  1. **Data Preprocessing: Normalize pixel values and reshape images for the autoencoder input.**
  2. ```
     data = fetch_olivetti_faces()
     faces = data.images
     target = data.target
     ```
     ler with convolutional layers to handle the spatial data
  3. ```
     faces
     ```
     ```
     array([[[0.30991736, 0.3677686 , 0.41735536, ..., 0.37190083,
              0.3305785 , 0.30578512],
             [0.3429752 , 0.40495867, 0.43801653, ..., 0.37190083,
              0.338843  , 0.3140496 ],
             [0.3429752 , 0.41735536, 0.45041323, ..., 0.38016528,
              0.338843  , 0.29752067],
             ...,
             [0.21487603, 0.20661157, 0.2231405 , ..., 0.15289256,
              0.16528925, 0.17355372],
             [0.20247933, 0.2107438 , 0.2107438 , ..., 0.14876033,
              0.16115703, 0.16528925],
     ```
     ta, optimizing the encoder to capture the essential
  4. to evaluate model performance.
  5. **nts of the first encoder layer to interpret learned**

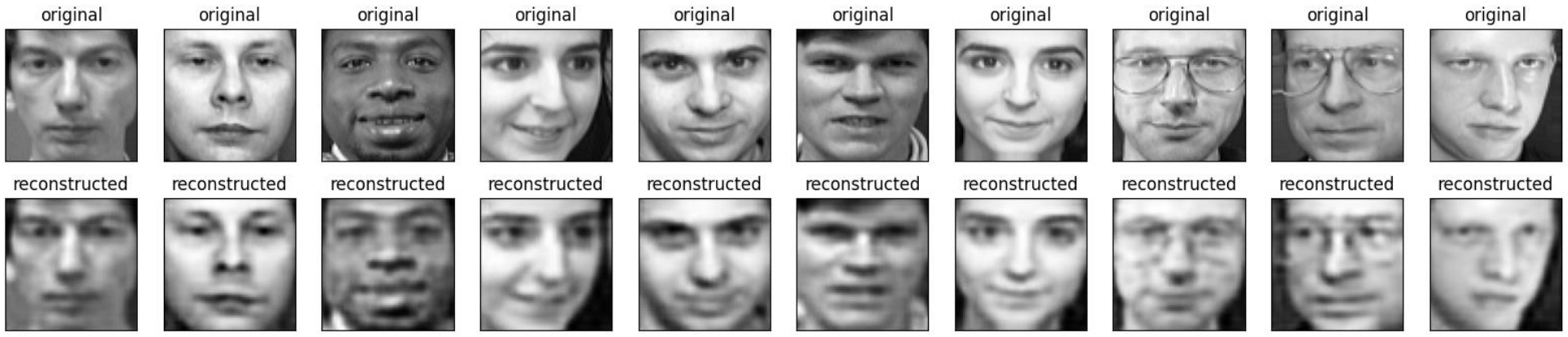
Feature Visualization (1/2)

- Visualizing the weights of the first encoder layer in an autoencoder generally means examining the filters learned by the first convolutional layer.
- For instance, if the first encoder layer is a Conv2D layer with 64 filters, each filter has a set of weights.
- Visualizing these weights helps understand what features the network is learning to recognize in the early stages of processing.
- In the case of an autoencoder on image data, each filter captures specific patterns, edges, or textures in the images.

Feature Visualization (2/2)

```
weights = autoencoder.layers[1].get_weights()[0]
n = min(10, weights.shape[-1])    # Show first 10 filters
plt.figure(figsize=(20, 5))
for i in range(n):
        ax = plt.subplot(1, n, i + 1)
        plt.imshow(weights[:, :, 0, i], cmap='gray')
        plt.axis('off')
plt.show()
```





Guidelines for Training Autoencoders (1/5)

- **Data Preprocessing**
  - Normalization: Scale your input data so that the pixel values are in the range [0,1] or [-1,1]. This helps with the stability and efficiency of training.
  - Reshaping: For image data, reshape the inputs to add a channel dimension, e.g., converting (28, 28) images to (28, 28, 1) for grayscale images.

Guidelines for Training (2/5)

```
# Encoder
x = Conv2D(32, (3, 3), activation='relu', padding='same')(input_img)
x = MaxPooling2D((2, 2), padding='same')(x)
x = Conv2D(16, (3, 3), activation='relu', padding='same')(x)
encoded = MaxPooling2D((2, 2), padding='same')(x)

# Decoder
x = Conv2D(16, (3, 3), activation='relu', padding='same')(encoded)
x = UpSampling2D((2, 2))(x)
x = Conv2D(32, (3, 3), activation='relu', padding='same')(x)
x = UpSampling2D((2, 2))(x)
decoded = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(x)
```

- **Model Architecture**
  - Symmetry: An autoencoder typically has a symmetric architecture with the encoder and decoder being mirror images of each other.
  - Bottleneck: The middle layer, known as the bottleneck, should have fewer units than the input layer, forcing the model to learn a compressed representation of the input data.
  - Convolutional Layers: For image data, use “Conv2D” layers in the encoder to capture spatial hierarchies, and “Conv2DTranspose” or “UpSampling2D” followed by “Conv2D” in the decoder to reconstruct the image.

Guidelines for Training Autoencoders (3/5)

- **Training**
  - Optimizer: Common choices are Adam, SGD, etc. Select an optimizer that works well with your data and model architecture.
  - Batch Size: Choose a batch size according to the available memory. Smaller batches often work better but might increase training time.
  - Epochs: Set the number of epochs to train the model. Use early stopping to prevent overfitting.
  - Validation Set: Use a part of the data as a validation set to monitor the model's performance on unseen data during training.

Guidelines for Training Autoencoders (4/5)

- **Loss Function**
  - Reconstruction Loss: The most common loss function is Mean Squared Error (MSE) for regression tasks or Binary Cross-Entropy for binary classification tasks.
- **Overfitting and Regularization**
  - Dropout: Can be used in the encoder and decoder to prevent overfitting.
  - Regularization: L1 or L2 regularization in the dense layers can also help prevent overfitting.

Guidelines for Training Autoencoders (5/5)

- **Model Evaluation and Visualization**
  - Reconstruction Error: Measure the difference between the input and its reconstruction.
  - Visual Inspection: Compare input and output images side by side.
  - Latent Space Analysis: Visualize the latent space to understand how the autoencoder is organizing and representing the data.

Generative Adversarial Networks

Introduction (1/5)

- Generative Adversarial Networks (GANs) are a class of artificial intelligence algorithms used in unsupervised machine learning
- A GAN consists of two neural networks, namely the Generator and the Discriminator, which are trained simultaneously.
- The Generator aims to produce data that mimics some distribution (often the distribution of the training data), while the Discriminator aims to distinguish between genuine and generated data.

Introduction (2/5)

- Generator:
  - This network aims to produce data that is indistinguishable from some real data.
  - It starts with a random noise and gradually refines its output as the training process advances.
- Discriminator:
  - This network tries to differentiate between real and fake (generated) data.
  - It takes in both real and fake samples and assigns a probability that a given sample is real.

Introduction (3/5)

- The Generator and Discriminator are in essence co-adversaries, engaged in a kind of cat-and-mouse game, often described mathematically as a minimax game or a zero-sum game.
- In a zero-sum game, one player's gain or loss is exactly balanced by the losses or gains of another player.
- In the context of GANs, the Generator and Discriminator are essentially playing a zero-sum game against each other.

Introduction (4/5)

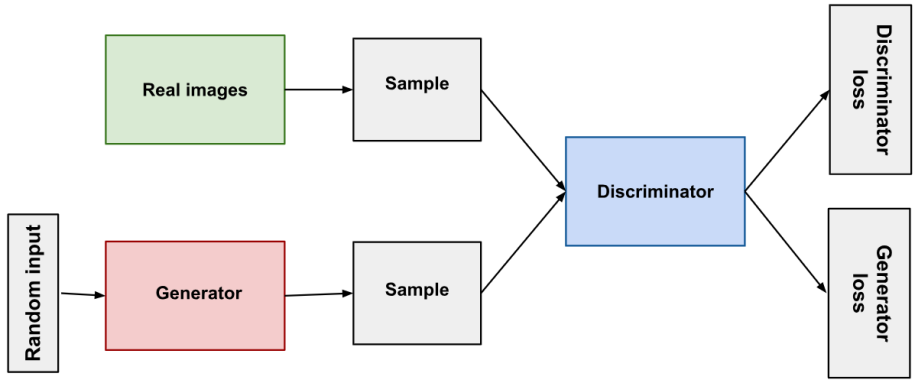
- Suppose we have a generative model which takes a random noise as input and generates a data point. We want the generated data point to be of good quality; hence, we should somehow judge its quality.
- One way to judge it is to observe the generated sample and assess its quality visually. In this case, the judge is a human. However, **we cannot take derivative of human's judgment** for optimization

- GAN, proposed in 2014, has the same idea but it can take derivative of the judgment. For that, uses a classifier as the judge rather than a human. Hence, we have a generator generating a sample and a binary classifier (or discriminator) to classify the generated sample as a real or generated sample.
- This classifier can be a pre-trained network which is already trained by some real and generated (fake) data points. However, GAN puts a step ahead and lets the classifier be **trained simultaneously** with training the generator. This is the core idea of adversarial learning where the classifier, also called the discriminator, and the generator compete with one another; hence, they **make each other stronger gradually by this competition**.

Source: https://bghojogh.github.io/pages/uoguelph/engg-6600-01-s23/

Introduction (5/5)

- A Generative Adversarial Network (GAN) is a class of machine learning systems where two neural networks contest with each other in a game.
- Conceived as a generative model, it's an approach to unsupervised learning.
- A GAN consists of two parts:
  - Generator: This network generates new data instances.
  - Discriminator: This network evaluates them.



Source: https://developers.google.com/machine-learning/gan/gan_structure?hl=zh-tw

Adversarial Learning (1/3)

- The original GAN, also called the vanilla GAN.
- Consider a d-dimensional dataset with n data points, i.e., $\{x_i \in \mathbb{R}^d\}_{i=1}^n$. In GAN, we have a generator G which takes a p-dimensional random noise $z \in \mathbb{R}^p$ as input and outputs a d-dimensional generated point $x \in \mathbb{R}^d$. Hence, it is the mapping $G : z \rightarrow x$ where $G(z) = x$.

Source: https://bghojogh.github.io/pages/uoguelph/engg-6600-01-s23/

Adversarial Learning (2/3)

- Let the distribution of random noise be denoted by $z \sim p_z(z)$. We want the generated $\tilde{x}$ to be very similar to some original (or real) data point x in the dataset. We need a module to judge the quality of the generated point to see how similar it is to the real point. A good candidate for the judge is a **classifier**, also called the **discriminator**. The discriminator (also called the critic), denoted by D : x → [0, 1], is a binary classifier which classifies the generated point as a real or generated point:

$$D(x) = \begin{cases} 1 & \text{if x is real,} \\ 0 & \text{if x is generated (fake).} \end{cases}$$

The perfect discriminator outputs one for real points and zero for generated points.

Source: https://bghojogh.github.io/pages/uoguelph/engg-6600-01-s23/

Adversarial Learning (3/3)

- Let us train the discriminator **simultaneously** while we are training the generator. This makes the discriminator D and the generator G stronger gradually while they compete each other.
- The generator tries to generate realistic points to fool the discriminator and make it a hard time to distinguish the generated point from a real point.
- The discriminator tries to discriminate the fake (i.e., generated) point from a real point.
- When one of them gets stronger in training, the other one tries to become stronger to be able to compete. Therefore, there is an **adversarial game** between the generator and the discriminator. This game **is zero-sum** because whatever one of them loses, the other wins.

Importance of GANs

- **Data Generation:** One of the most obvious utilities of GANs is the generation of new data that mimics the distribution of the training data. This is useful in domains like artwork creation, where GANs can create realistic images, or in natural language processing, where they can generate human-like text.
- **Data Augmentation:** For tasks where the dataset is limited, GANs can create additional data to augment the existing dataset. This is particularly useful in fields like healthcare, where obtaining more real data is often difficult or expensive.
- **Anomaly Detection:** Because GANs learn the distribution of the training data, they can be used to identify data points that do not fit this distribution. This is useful in fields like fraud detection and network security.
- **Complexity:** GANs are capable of capturing complex, high-dimensional distributions. This makes them powerful tools for tasks that involve understanding the underlying distribution of complex datasets.
- **Representation Learning:** While the Generator is obviously useful, the Discriminator also learns useful features of the data in the process of trying to distinguish between real and fake. These learned features can often be used for other machine learning tasks, a practice known as transfer learning.
- **Research and Innovation:** GANs have become a cornerstone in the machine learning research community, inspiring a wide range of variations and applications, from image-to-image translation (like turning satellite images into maps) to drug discovery.

Historical Background

- GANs were introduced by Ian Goodfellow and his colleagues in 2014. The concept revolutionized the way we think about generating data in an unsupervised manner.
- Evolution
  - DCGAN (Deep Convolutional GAN): Improved stability in training.
  - CycleGAN: For unpaired image-to-image translation.
  - WGAN (Wasserstein GAN): Introduced Wasserstein loss for more stable training.
  - BigGAN: GANs that can generate high-resolution and high-quality images.
  - StyleGAN & StyleGAN2: Introduced style-based generators.

Source: https://github.com/hindupuravinash/the-gan-zoo

Core Components – Generator (G)

- Definition: A neural network that takes a random noise vector as input and produces synthetic samples as output.
- Objective: To generate data that is indistinguishable from real data.
- Terminology:
  - Latent Space: The high-dimensional space from which random vectors are sampled as inputs for the generator.
  - Upsampling: The process of increasing the dimensions of the data, often used in the generator to transform a low-dimensional random vector into a high-dimensional synthetic sample.

Latent Space in GANs

- The latent space is a high-dimensional space from which the random noise vectors are sampled as input to the Generator.
- It serves as a compact representation of the data's inherent features.
- In most GANs, you randomly sample from a predefined distribution (usually a Gaussian distribution) to generate noise vectors.
- The latent space evolves implicitly as the Generator learns to map these random vectors to realistic data.

Upsampling in Generator

- Upsampling in the context of the Generator refers to the process of increasing the spatial dimensions (height and width) of the input tensor, often achieved through operations like deconvolution or transposed convolution.
- Mechanism:
  - Transposed Convolution: This operation essentially reverses the effect of a regular convolution. It maps a low-resolution input to a higher resolution output.
  - Pixel Shuffle: Another technique that rearranges elements in a tensor to expand its dimensions.
  - Nearest Neighbor / Bilinear Interpolation: Simple yet effective techniques to increase tensor dimensions by filling in new values based on neighboring values.

Why Upsampling Is Important

- Data Generation: The Generator starts with a low-dimensional noise vector. To produce complex, high-dimensional output such as images, upsampling is essential.
- Detail Recovery: Upsampling helps in generating finer details as the tensor goes through the network, enabling the production of high-quality synthetic data.
- Upsampling is a critical process in the generator's architecture, especially in GANs designed for tasks like image generation. It allows the network to transform low-resolution, abstract representations into more detailed and complex outputs.

Components in Generator (G)

- Input Layer: Takes a random noise vector *z* sampled from a latent space.
- Hidden Layers: Comprises a series of fully connected, convolutional, or transposed convolutional layers.
- Output Layer: Produces the synthetic data, typically matching the dimensions of the real data.

Operations in Generator (G)

- Upsampling: Uses transposed convolutional layers or other techniques to transform a low-dimensional noise vector into a high-dimensional synthetic sample.
- Activation Functions: Commonly uses ReLU or Leaky ReLU in hidden layers and tanh or Sigmoid in the output layer, depending on the data type.

Core Components – Discriminator (D)

- Definition: A neural network that tries to distinguish between real and generated samples.
- Objective: To correctly classify a given sample as real or fake.
- Terminology:
  - Downsampling: The process of reducing the dimensions of the data, often used in the discriminator to transform a high-dimensional sample into a scalar classification output.
  - Real Score: The output of the discriminator when given a real sample, ideally close to 1.

Downsampling in Discriminator

- Downsampling refers to the process of reducing the spatial dimensions of the input tensor. In GANs, this is typically accomplished using strided convolutions in the Discriminator.
- Mechanism:
  - Strided Convolution: A convolutional operation with a stride greater than 1, which effectively reduces the dimensions of the output tensor.
  - Pooling Layers: Operations like MaxPooling or AveragePooling can also be used, though they are less common in modern GAN architectures.
  - Global Average Pooling: Sometimes used to convert a 2D tensor into a 1D tensor before a fully connected layer for classification.

Why Downsampling Is Important

- Classification: The Discriminator needs to classify the input data as real or fake. Downsampling helps in reducing the computational complexity and focuses on essential features for classification.
- Feature Extraction: As the dimensions reduce, the Discriminator learns more abstract and complex features of the data, aiding in better classification.

Components in Discriminator (D)

- Input Layer: Takes in data samples, which could either be real or generated by the Generator.
- Hidden Layers: Typically comprises a series of convolutional layers and fully connected layers.
- Output Layer: Outputs a scalar value representing the probability that the input sample is real.

Operations in Discriminator (D)

- Downsampling: Employs strided convolutions or pooling layers to reduce the dimensions of the input data.
- Activation Functions: Often uses Leaky ReLU in hidden layers and Sigmoid in the output layer to output probabilities.

Advanced Concepts (1/2)

- Mode Collapse:
  - Definition: A phenomenon where the generator starts to produce very similar or identical samples for different input vectors.
  - Solution: Techniques like minibatch discrimination, spectral normalization, etc., are employed to combat mode collapse.
- Nash Equilibrium:
  - Definition: A state in which neither the generator nor the discriminator can improve their performance given the current state of the other.
  - Significance: It is considered the optimal stopping point for GAN training, although reaching true Nash equilibrium is difficult in practice.

Advanced Concepts (2/2)

- Stability and Convergence:
  - Definition: Describes the behavior of the GAN during the training process.
  - Challenges: GANs are notorious for being difficult to train; they may not converge or may become unstable.
  - Solutions: Various architectural tweaks and training techniques have been proposed, such as gradient clipping, learning rate annealing, and different normalization methods.

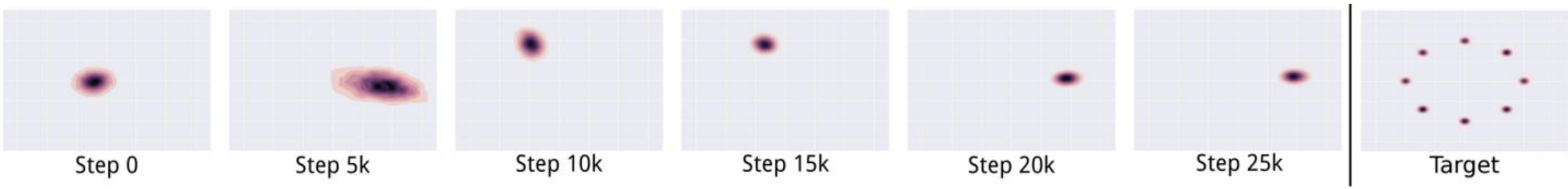
Mode Collapse Problem in GAN (1/

- We expect from a GAN to learn a meaningful latent space of z so that every specific value of z maps to a specific generated data point x. Also, nearby z values in the latent space should be mapped to similar but a little different generations.
- The mode collapse problem, also known as the Helvetica scenario, is a common problem in GAN models. It refers to when the generator cannot learn a perfectly meaningful latent space as was explained. Rather, it learns to map several different z values to the same generated data point.
- Mode collapse usually happens in GAN when the distribution of training data, $p_{data}(x)$, has multiple modes.

Source: https://bghojogh.github.io/pages/uoguelph/engg-6600-01-s23/

Mode Collapse Problem in GAN (2/

- An example of mode collapse is illustrated in the following figure which shows training steps of a GAN model when the training data is a mixture of Gaussians. In different training steps, GAN learns to map all z values to one of the modes of mixture. When the discriminator learns to reject generation of some mode, the generator learns to map all z values to another mode. However, it never learns to generate all modes of the mixture. We expect GAN to map some part, and not all parts, of the latent space to one of the modes so that all modes are covered by the whole latent space.



Source: https://bghojogh.github.io/pages/uoguelph/engg-6600-01-s23/

Causes of Mode Collapse

- Generator's Limitation: The generator might find a certain type of output that consistently fools the discriminator. Once it discovers this "shortcut", it tends to keep producing variations of this output rather than exploring a broader range of possibilities.
- Discriminator's Feedback: If the discriminator's feedback to the generator is not sufficiently informative, it can lead the generator to focus on a narrow set of outputs that seem to work well.
- Training Dynamics: The adversarial nature of GANs can sometimes lead to unstable training dynamics where the generator gets stuck in producing specific outputs to deceive the discriminator.

Consequences of Mode Collapse

- Lack of Diversity: The primary consequence of mode collapse is the lack of diversity in the generated samples, which defeats the purpose of GANs aiming to produce varied and realistic outputs.
- Reduced Utility: For applications that require a rich variety of generated samples (e.g., data augmentation, artistic creation), mode collapse significantly reduces the utility of the GAN.
- Biased Learning: The generator's biased output can lead to misleading conclusions about the performance and capability of the GAN, especially in tasks requiring a representative sample of data.

Addressing Mode Collapse (1/2)

- Modified Architectures and Training Procedures: Implementing alternative GAN architectures (like Wasserstein GANs) or modifying the training procedure can help in mitigating mode collapse by providing more stable training dynamics. (https://arxiv.org/abs/1701.07875)
- Regularization Techniques: Applying regularization techniques like gradient penalty can encourage the generator to explore more diverse outputs.

Addressing Mode Collapse (2/2)

- Mini-Batch Discrimination: This technique involves providing the discriminator with information about a batch of samples instead of individual samples, encouraging the generator to produce diverse outputs.
- Experience Replay: Storing previously generated samples and showing them to the discriminator again in later stages can prevent the generator from focusing only on new types of samples that fool the discriminator.
- Monitoring and Adjusting Training: Carefully monitoring the training process and adjusting the learning rates or other hyperparameters can also help in preventing mode collapse.

Nash Equilibrium

- Nash Equilibrium is a concept from game theory that describes a situation in which each player's strategy is optimal given the strategy of their opponent(s).
- In simpler terms, in a Nash Equilibrium, no player can improve their payoff by unilaterally deviating from their current strategy, assuming the other players keep their strategies unchanged.
- In GAN training, reaching a Nash Equilibrium means that the Generator has become so good at generating fake data that the Discriminator can no longer distinguish between real and fake data.
  - At this point, both networks are said to be in equilibrium because neither can improve without the other changing.

Nash Equilibrium – Mathematical Representation

- In the context of GANs, the Nash Equilibrium can be represented by the minimax function:

$$\min_G \max_D V(D, G)$$

- Here, V(D,G) is the value function that both the Generator (G) and the Discriminator (D) are trying to optimize.
- The Generator aims to minimize this function, while the Discriminator aims to maximize it.
- In a Nash Equilibrium, neither can improve their objective function value without the other also changing their strategy.

Nash Equilibrium – Challenges

- Computational Complexity:

  Finding the Nash Equilibrium in a high-dimensional, non-convex game like GANs is computationally intensive.

- Mode Collapse:

  Before reaching Nash Equilibrium, GANs can suffer from issues like mode collapse, where the Generator produces limited varieties of samples.

Researchers often aim for a “local” Nash Equilibrium, where the networks are relatively stable, and the generated data is of high quality, even if it's not a perfect equilibrium.

Training Dynamics – Adversarial Training

- The fundamental idea behind GANs is adversarial training, where the Generator and the Discriminator are trained simultaneously but with opposing objectives.
  - Objective of the Generator (G): To generate data that is indistinguishable from real data.
  - Objective of the Discriminator (D): To correctly identify whether the given data is real or generated by the Generator.
- The training process iteratively updates both G and D, ideally until they reach a Nash equilibrium, where neither can improve without the other changing as well.

Loss Functions (1/2)

- The loss function, usually a form of Binary Cross-Entropy loss, measures how well each network is doing its job. It's common to see the following setup:
  - The first term encourages D to correctly classify real samples.
  - The second term encourages D to correctly classify fake samples.
  - Discriminator Loss: $L_D = -\log\big(D(x)\big) - \log\big(1 - D(G(z))\big)$
  - Generator Loss: $L_G = -\log(D(G(z)))$
    - This term encourages G to generate samples that D classifies as real.

Loss Functions (2/2)

- GANs try to replicate a probability distribution.
- They should therefore use loss functions that reflect the distance between the distribution of the data generated by the GAN and the distribution of the real data.
- Two common GAN loss functions:
  - minimax loss
  - Wasserstein loss

Source: https://developers.google.com/machine-learning/gan/loss

Minimax Loss (1/2)

- The generator tries to minimize the following function while the discriminator tries to maximize it: $\mathbb{E}_x[\log D(x)] + \mathbb{E}_z[\log(1 - D(G(z))]$
  - D(x) is the discriminator's estimate of the probability that real data instance x is real.
  - E_x is the expected value over all real data instances.
  - G(z) is the generator's output when given noise z.
  - D(G(z)) is the discriminator's estimate of the probability that a fake instance is real.
  - E_z is the expected value over all random inputs to the generator (in effect, the expected value over all generated fake instances G(z)).
  - The formula derives from the cross-entropy between the real and generated distributions.

Source: https://developers.google.com/machine-learning/gan/loss

Minimax Loss (2/2)

- The generator can't directly affect the log(D(x)) term in the function, so, for the generator, minimizing the loss is equivalent to minimizing log(1 - D(G(z))).

Source: https://developers.google.com/machine-learning/gan/loss

Wasserstein Loss

- This loss function depends on a modification of the GAN scheme (called "Wasserstein GAN" or "WGAN") in which the discriminator does not actually classify instances.
- Because WGAN can't really discriminate between real and fake, the WGAN discriminator is actually called a "critic" instead of a "discriminator".
- Critic Loss: D(x) - D(G(z))

  The discriminator tries to maximize this function. In other words, it tries to maximize the difference between its output on real instances and its output on fake instances.

- Generator Loss: D(G(z))

  The generator tries to maximize this function. In other words, It tries to maximize the discriminator's output for its fake instances.

Source: https://developers.google.com/machine-learning/gan/loss

Data Flow in GAN (1/2)

1. Random noise vectors are sampled from the latent space.
2. These vectors are fed into the Generator.
3. Upsampling layers in the Generator transform these vectors into high-dimensional data.
4. This generated data and real data are fed into the Discriminator.
5. Downsampling layers in the Discriminator reduce the data dimensions.
6. The Discriminator then classifies the downsampled data as real or fake.

Data Flow in GAN (2/2)

1. Starting Point

   The Generator begins with a set of random noise (usually a normal distribution).

2. Data Generation

   This noise is input to the Generator, which transforms it into a data instance.

3. Data Evaluation

   The Discriminator takes in both real data (from the actual dataset) and fake data (from the Generator) and tries to distinguish between the two.

Evaluation Metrics for GANs (1/3)

- Fréchet Inception Distance (FID):
  - FID measures the similarity between the generated data and real data distributions in the feature space of a pre-trained Inception network.
  - The FID between two multivariate Gaussians $N(\mu_1, \Sigma_1)$ and $N(\mu_2, \Sigma_2)$ is defined as: $\text{FID} = ||\mu_1 - \mu_2||^2 + \text{Tr}(\Sigma_1 + \Sigma_2 - 2(\Sigma_1\Sigma_2)^{\frac{1}{2}})$. μ_1, Σ_1 are the mean and covariance of features from the real data, and μ_2, Σ_2 are from the generated data.

Evaluation Metrics for GANs (2/3)

- Inception Score (IS)
  - IS uses the Inception network to classify generated images into different classes and computes a score based on the distribution of these classes.
  - For a set of N generated samples, the IS is given by: IS $=$ $\exp(\frac{1}{N}\sum_{i=1}^{N} KL(p(y|x_i)||p(y))$. $p(y|x_i)$ is the conditional class distribution for sample x_i, and $p(y)$ is the marginal class distribution.

Evaluation Metrics for GANs (3/3)

- Wasserstein Distance (For WGANs)
  - This metric is specific to Wasserstein GANs (WGANs) and measures the Earth Mover's Distance between the real and generated data distributions.
  - In the context of WGANs, the Wasserstein distance is given by the Discriminator‘s loss: $W = \max_D \mathbb{E}_{x \sim P_{\text{real}}}[D(x)] - \mathbb{E}_{x \sim P_{\text{fake}}}[D(x)]$.$\mathbb{E}$ represents the expected value, and P_{real}, P_{fake} are the real and fake data distributions..

GAN Training

- Because a GAN contains two separately trained networks, its training algorithm must address two complications:
  - GANs must juggle two different kinds of training (generator and discriminator).
  - GAN convergence is hard to identify.

Source: https://developers.google.com/machine-learning/gan/training

How to Train a GAN (1/5)

1. Train the Discriminator
   - Provide it with real data and fake data from the Generator.
   - Update the Discriminator's weights based on its ability to correctly classify the real and fake data.
2. Train the Generator
   - Generate new data.
   - Pass it to the Discriminator. If the Discriminator classifies the output as real, then the Generator is doing well.
   - Update the Generator's weights based on the Discriminator’s response.
3. Iterate
   - Repeat this process until the Generator produces realistic data. The ideal endpoint is when the Discriminator can no longer reliably distinguish fake data from real data.

How to Train a GAN (2/5)

1. Initialize Networks: Initialize the Generator (G) and Discriminator (D) with random weights.
2. Preprocess Data: Prepare your real data samples, often involving normalization and data augmentation.

How to Train a GAN (3/5)

3. Training Loop: Typically involves the following steps:

Step 1: Train the Discriminator

1.1 Sample a mini-batch of real data x from the data distribution.

1.2 Generate a mini-batch of fake data G(z), where z is a random noise vector.

1.3 Compute the Discriminator's loss on real and fake data. Commonly used loss is Binary Cross-Entropy: $L_D = -\log\big(D(x)\big) - \log\big(1 - D(G(z))\big)$

1.4 Update the Discriminator's weights using backpropagation to minimize L_D.

How to Train a GAN (4/5)

3. Training Loop: Typically involves the following steps:

   Step 2: Train the Generator

   2.1 Generate a new mini-batch of fake data G(z).

   2.2 Compute the Generator's loss. The aim is to fool the Discriminator into thinking the fake samples are real: $L_G = -\log(D(G(z)))$

   2.3 Update the Generator's weights using backpropagation to minimize L_G.

How to Train a GAN (5/5)

3. Training Loop: Typically involves the following steps:

   Step 3: Check Convergence

   3.1 Monitor the losses L_D and L_G.

   3.2 Optionally, evaluate using metrics like Fréchet Inception Distance (FID) or Inception Score (IS).

   3.3 If the networks have converged or met specific criteria, exit the loop; otherwise, return to Step 1.

Alternating Training (1/2)

- GAN training proceeds in alternating periods:
  - The discriminator trains for one or more epochs.
  - The generator trains for one or more epochs.
  - Repeat steps 1 and 2 to continue to train the generator and discriminator networks.
- As the generator improves with training, the discriminator performance gets worse because the discriminator can't easily tell the difference between real and fake.

Source: https://developers.google.com/machine-learning/gan/training

Alternating Training (2/2)

- This progression poses a problem for convergence of the GAN as a whole: the discriminator feedback gets less meaningful over time.
- If the GAN continues training past the point when the discriminator is giving completely random feedback, then the generator starts to train on junk feedback, and its own quality may collapse.
- For a GAN, convergence is often a fleeting, rather than stable, state.

Source: https://developers.google.com/machine-learning/gan/training

Guidelines for Training GANs (1/5)

- **Careful Design of Network Architecture**
  - Symmetry Between Generator and Discriminator: Ensure that the complexity of the generator and discriminator is balanced. An overly powerful discriminator can lead to the generator's failure to learn.
  - Use of Convolutional Layers: For image-related tasks, convolutional layers are more effective in the discriminator. Similarly, deconvolutional (transposed convolutional) layers are effective in the generator.

Guidelines for Training GANs (2/5)

- **Choice of Loss Function**
  - Standard GAN Loss: The original GAN paper proposes a minimax loss function, but this can lead to vanishing gradients.
  - Alternative Loss Functions: Wasserstein loss (in WGANs) or Least Squares loss (in LSGANs) can provide more stable training and alleviate some common issues like mode collapse.

Guidelines for Training GANs (3/5)

- **Regularization and Normalization Techniques**
  - Batch Normalization: Use batch normalization in both the generator and the discriminator to stabilize training.
  - Dropout: Implementing dropout in the discriminator can prevent overfitting.
- **Training Procedure**
  - Balance in Training: Alternate between training the discriminator and the generator. Avoid training one significantly more than the other.
  - Learning Rate and Optimizers: Use a lower learning rate and consider different optimizers for the generator and discriminator (e.g., Adam or RMSprop).

Guidelines for Training GANs (4/5)

- **Monitoring and Tuning**
  - Track Losses: Monitor the loss of both networks to understand their learning progress.
  - Adjust Hyperparameters: Be prepared to fine-tune hyperparameters such as learning rates, batch sizes, or the architecture itself based on the performance.
- **Dealing with Mode Collapse**
  - Diversity in Training Data: Ensure that the training data is diverse enough to encourage the generator to learn a variety of outputs.
  - Mini-batch Discrimination: This technique can encourage the generator to produce diverse samples.

Guidelines for Training GANs (5/5)

- **Experiment with Training Techniques**
  - Two Time-Scale Update Rule (TTUR): Using different learning rates for the generator and discriminator can be beneficial.
  - Label Smoothing and Noise: Introducing soft labels or adding noise to discriminator inputs can prevent overfitting.
- **Evaluate GANs Properly**
  - Qualitative Evaluation: Regularly generate samples and visually inspect them.
  - Quantitative Metrics: Use metrics like Inception Score (IS) and Fréchet Inception Distance (FID) to quantitatively evaluate the quality of generated samples.

Thank you