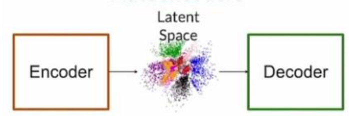
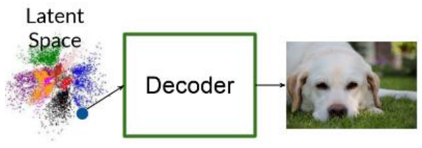
Build Basic Generative Adversial Networks (GANs)

Generative model

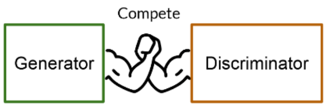
- Discriminative model: X → Y (P(Y|X))
- Generative model: ξ, Y → X (P(X|Y))
  - ξ: a vector of random noises used to ensure not the same image generated every time.
- **Types of generative models:**
  - **Variational Autoencoder (VAE):**



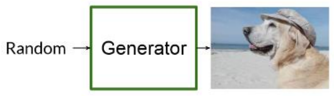
    - VAE feeds realistic images into the encoder
    - Encoder:
      - find a good way to represent the image in the latent space (a vector, e.g., (6.2,-3,21)).
      - Actually encodes the image onto a whole distribution and sample a point on the distribution and feed it to the decoder. This adds a little noise since different points can be sampledd on this distribution.
    - VAE takes the latent representation or a point close to it and put it through the decoder.
    - Decoder:
      - Reconstruct the realistic image that the encoder saw before.
    - After training, lop off the encoder, and pick random point in the latent space and feed it to decoder.



  - **Generative Adversarial Networks (GAN):**



    - Input random noise vector(e.g., (1.2,3,-5)) into the generator.
    - Generator:
      - Very similar to the decoder in VAE
      - Take noise(random) vector as input → NN → image output(pixels)
    - Discriminator:
      - Look at fake image generated by generator and the real image and simultaneously try to figure out which ones are real and which ones are fake.
    - After training:

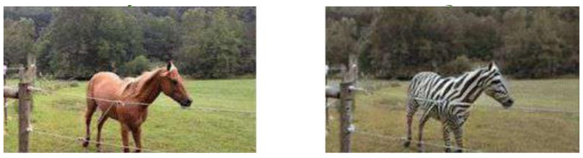


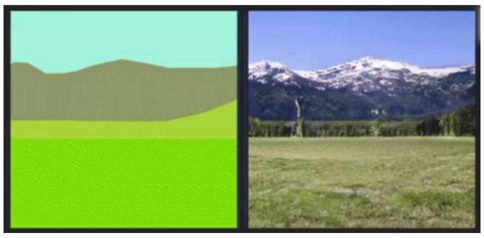
Real Life GANs

- StyleGAN2

  - Mimics the distribution of the training data

  - Face Generation
  - Karras, Tero, et al. "Analyzing and improving the image quality of stylegan." Proceedings of the IEEE/CVF
    Conference on Computer Vision and Pattern Recognition. 2020.
- CycleGAN

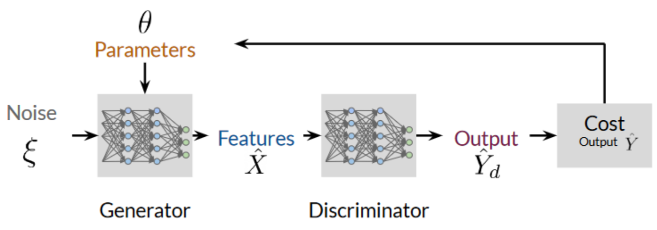


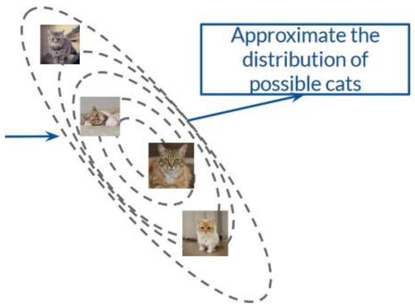
  - Take image from one domain the transform it into another.
  - Park, Taesung, et al. "Semantic image synthesis with spatially-adaptive normalization." Proceedings of the IEEE
    Conference on Computer Vision and Pattern Recognition. 2019.
- GauGAN
  - 
  - Doodles → Pictures
  - Park, Taesung, et al. "Semantic image synthesis with spatially-adaptive normalization." Proceedings of the IEEE Conference on
    Computer Vision and Pattern Recognition. 2019.
- Zakharov, Egor, et al. "Few-shot adversarial learning of realistic neural talking head models." Proceedings of the IEEE
  International Conference on Computer Vision. 2019.
- 3D-GAN
  - Wu, Jiajun, et al. "Learning a probabilistic latent space of object shapes via 3d generative-adversarial
    modeling." Advances in neural information processing systems. 2016.

Intuition behind GANs

-
- Generator learns to make fakes that look real
- Discriminator learns to distinguish real from fake
- Procedure:
  - A dataset of real images
  - A generator and a discriminator
  - Train the discriminator to distinguish real images
  - When the generator produces a batch of paintings, the generator will know in what direction to go on and improve, by looking at the scores assigned to her work by the discriminator.
  - At the end, fakes look real

Generator:

- Learning:
  - 

  - It learns P(X)
  - 

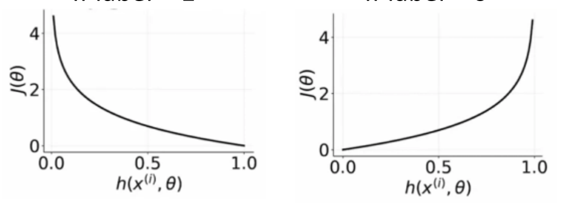
center: more common
  - Features: picture
  - Sampling:
    - Weighted are saved, input new noise vectors and output new (dog/cat) images.

Binary Cross Entropy (BCE)

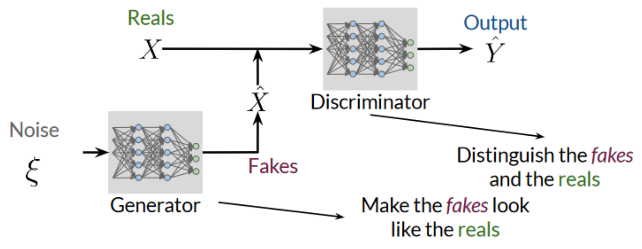
$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^{(i)} \log h(x^{(i)}, \theta) + (1 - y^{(i)}) \log(1 - h(x^{(i)}, \theta))]$$

- m: batch size
-

If label = 1 If label = 0

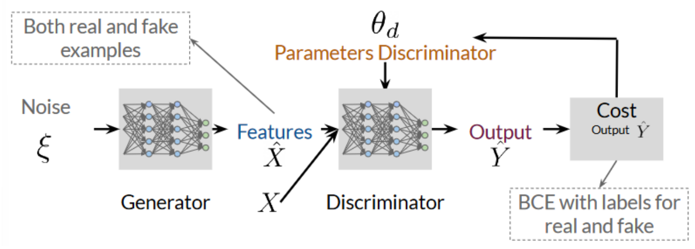


GANs Model

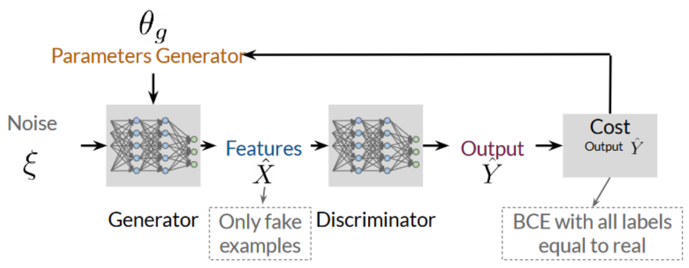


Reals
X
Output
$\hat{Y}$
Discriminator
$\hat{X}$
Noise
ξ
Generator
Fakes
Distinguish the fakes and the reals
Make the fakes look like the reals

Training Discriminator:



Training Generator:



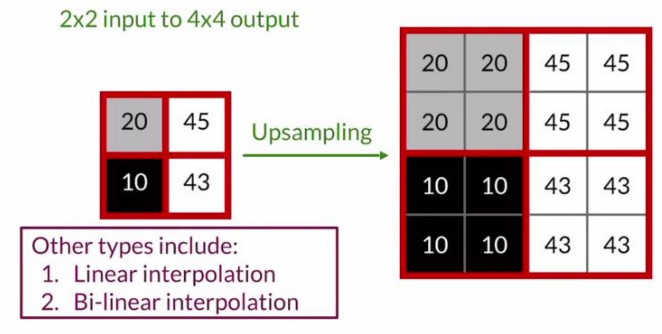
Note:

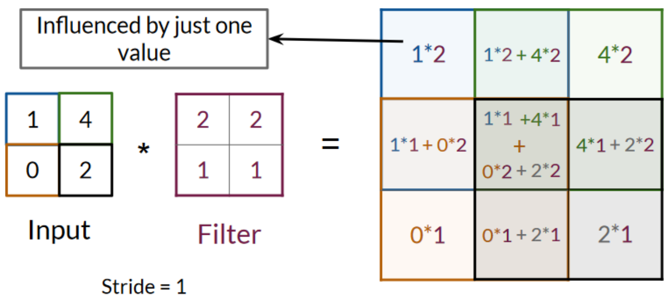
- GANs train in an alternating fashion
- The two models should always be at a similar "skill" level
  - We should train both generator and discriminator at the same time, if we use a superior discriminator at first, It gives 100% fake for fake iamges, there is no way for generator to improve.

Batch normalization:

- Introduces learnable shift and scale factors.
- During test, the running statistics from training are used.
- Advantages:
  - smooths the cost function
  - reduces the internal covariate shift
  - speeds up learning

Upsampling:

- Increases the size of the input
- Nearest Neighbors
  - 

- Transposed Convolution
  - A transposed convolution learns a filter to upsample.
  - 

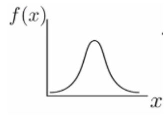
    - Values in the filter are learned.
  - Problem:
    - Checkerboard issue:
    - Odena, et al., "Deconvolution and Checkerboard Artifacts", Distill, 2016. http://doi.org/10.23915/distill.00003
  - Solution:
    - Using upsampling followed by convolution.

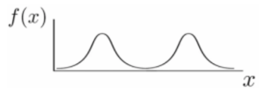
Wasserstein GANs with Gradient Penalty

Existing problem:

- GANs can get stuck generating only one output, such as a single dog breed, from a diverse dog breed dataset.
- This occurs because the discriminator judges images as entirely real or fake.
- The generator repeats golden retriever image.
- Learning stopped, which is problematic for the network's development.
- Binary cross-entropy loss exacerbates the issue by pushing the discriminator towards extreme values.

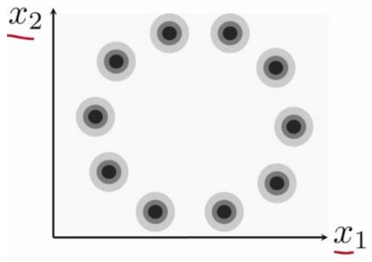
Mode:

- A high concentration of observations within a data distribution.
- Intuition: Any peak on the density function is a mode
    - 

        - In a normal distribution, the mean is the single mode.
    - 

        - Bimodal and multimodal distributions exist with multiple peaks.

Mode Collapse

- Mode collapse occurs when the generator gets stuck generating one mode. The discriminator will eventually learn to differentiate the generator's fakes when this happens and outskill it, ending the model's learning.
    - Handwritten digits example:
        - 

        - Features x_1 and x_2 represent dimensions of handwritten digits.
        - Probability density distribution with peaks (modes) for each digit.
        - Observations of the same digit cluster around these modes.
- Model Collapse with handwritten digits example:
    - Discriminator struggles with distinguishing ones and sevens.
    - Generator exploits this weakness, overproducing ones and sevens.
    - Generator might further collapse to producing only ones, as the discriminator adjusts (can distinguish sevens as fake better than ones).
- Summary:
    - Modes are peaks in the distribution of features
    - Typical with real-world datasets
    - Mode collapse happens when the generator gets stuck in one mode

Problem with BCE Loss

- The generator aims to maximize this cost, indicating effective deception.
- The discriminator seeks to minimize the cost, signifying accurate classification.
- Generator focuses only on the fake side, lacking insight into the real samples.

Minimax Game

- Simplified: $\min_{d} \max_{g} -[\mathbb{E}(\log(d(x))) + \mathbb{E}(1 - \log(d(g(z))))]$
- GAN training is often likened to a minimax game.
- The overarching goal is to make the generated data distribution mimic the real data distribution closely.
- BCE loss function approximates minimizing a complex function to align these distributions.

Challenges

- Discriminator's task is generally simpler compared to the complex creative process of the generator.
- Initially, the discriminator struggles to differentiate between real and generated distributions, providing useful gradients to the generator.
- As training progresses, the discriminator becomes better at distinguishing the two distributions.
- Real distribution centers around a value of 1.
- Generated distribution gravitates towards 0.
- the real/fake distributions are far apart
- An improved discriminator yields less informative feedback, issuing gradients close to zero.
- This leads to the vanishing gradient problem, where the generator lacks clear direction for improvement.

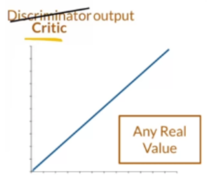
Summary

- **Saturation of the Discriminator**: When the real and fake distributions are clearly distinguishable (far apart), the discriminator can easily tell them apart. In such cases, the discriminator's confidence in its classifications is very high, leading to a state known as saturation.
- **Loss Function Characteristics**: In the saturated regime, the BCE loss function used to train the discriminator flattens out. Mathematically, this means that for fake inputs, the term $(1 - y^{(i)}) \log(1 - h(x^{(i)}, \theta))]$ becomes very close to zero, vice versa.
- **Vanishing Gradient Problem**: In such flat regions of the loss function, the gradient with respect to the generator's parameters becomes very small or even vanishes.
- **Lack of Direction for Improvement**: When the gradient vanishes, the generator does not receive clear feedback on how to adjust its parameters to improve.

Earth Mover's Distance (EMD)

- Earth mover's distance (EMD) is a function that measure how different two distributions are based on distance and amount that needs to be moved.
- Doesn't have flat regions when the distributions are very different
- Approximating EMD solves the problems associated with BCE

Wasserstein Loss (W-Loss)

- $\min_{g} \max_{c} \; \mathbb{E}(c(x)) - \mathbb{E}(c(g(z)))$
- Approximate the Earth Mover's Distance
- Calculates the difference between expected values of real vs. fake predictions by the critic (c(x)).
- Critic (formerly the discriminator) maximizes the distance between real and fake distribution evaluations.
- Generator minimizes the distance to make fake images appear real.
- No logs in the function, as critic's outputs are unbounded.
    - 

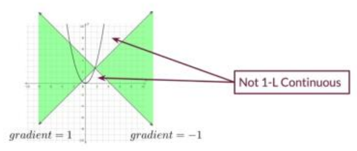
W-Loss vs BCW Loss

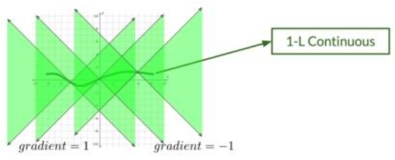
BCW Loss	W-Loss
Discriminator outputs between 0 and 1	Critic outputs any number
-[E(log(d(x)))+E(1-log(d(g(z))))]	E(c(x))-E(c(g(z)))

Advantages of W-Loss

- Avoids vanishing gradient problem because it is not bounded.
- Mitigates mode collapse by providing consistent feedback to the generator.
- Approximates Earth Mover's Distance without complex mathematical expressions.

Condition on Wasserstein Critic

- Critic needs to be 1-Lipschitz (1-L) Continuous.
    - The norm of the gradient should be at most 1 for every point
    - 

    - Critic's neural network needs to be 1-L Continuous when using W-Loss
        - This condition ensures that W-Loss is validly approximating Earth Mover's Distance.

Importance of 1-Lipschitz Continuity in Wasserstein Loss:

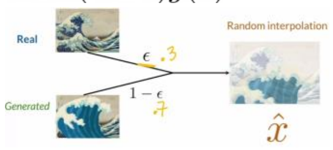
- **Valid EMD Approximation**: The theoretical foundation of Wasserstein Loss relies on the concept of optimal transport, which requires a 1-Lipschitz function to ensure that the Wasserstein distance is properly computed. If the critic could give infinitely large scores, the Wasserstein distance would not be meaningful, and the GAN could exploit these scores without truly improving the generated distribution.
- **Stability of Gradients**: Enforcing the critic to be 1-Lipschitz ensures that the gradients used during training are bounded. This prevents exploding gradients, leading to more stable and reliable training of the GAN.
- **Meaningful Feedback**: With 1-Lipschitz continuity, the feedback (gradients) provided to the generator remains useful throughout the training process. It avoids scenarios where the critic's output saturates, which could lead to vanishing gradients and hinder the generator's ability to improve.

1-Lipschitz Continuity Enforcement

- Weight Clipping
  - Process:
    - After updating the weights during gradient descent, any weights outside of a predefined interval are clipped to the interval's bounds.
  - Drawbacks:
    - Can overly restrict the critic's capacity to learn, limiting its ability to find optimal solutions.
    - Not enforce Lipschitz continuity enough if the interval is too large.
    - Requires careful hyperparameter tuning.
- Gradient Penalty

$$\min_{g} \max_{c} \ \mathbb{E}(c(x)) - \mathbb{E}(c(g(z))) + \lambda \underbrace{\text{reg}}_{\text{Regularization of the critic's gradient}}$$

  - Process:
    - Adds a regularization term to the loss function weighted by λ that penalizes the critic when its gradient norm exceeds one.
  - Implementation:
    - Sample points by interpolating between real and fake data points using a random weight epsilon.
    - Interpolation:
      - $\epsilon x + (1 - \epsilon) g(z)$



    - Regularization term:
      - $\mathbb{E}(||\nabla c(\hat{x})||_2 - 1)^2$ penalize more when further away from 1
  - Advantages:
    - More flexible and less restrictive than weight clipping.
    - Encourages, rather than strictly enforces, 1-Lipschitz continuity.
    - Tends to work better.

Wasserstein Loss with Gradient Penalty

```python
def gradient_penalty(gradient):
    # Flatten the gradients so that each row captures one image
    gradient = gradient.view(len(gradient), -1)
    # Calculate the magnitude of every row
    gradient_norm = gradient.norm(2, dim=1)
    # Penalize the mean squared distance of the gradient norms from 1
    penalty = torch.mean((gradient_norm-1)**2)
    return penalty
```

- $\min_g \max_c \mathbb{E}(c(x)) - \mathbb{E}(c(g(z))) + \lambda\mathbb{E}(||\nabla c(\hat{x})||_2 - 1)^2$
    - Makes the GAN less prone to mode collapse and vanishing gradient
    - Regularization term tries to make the critic be 1-L continuous, for the loss function to be continuous and differentiable.

Singular Value Decomposition (SVD)

- A generalization of eigendecomposition.
- Used to factorize a matrix as W = UΣV^T, where U, V are orthogonal matrices (rotate) and Σ is a matrix of singular values on its diagonal (stretch).

$$\Sigma = \begin{bmatrix} \sigma_1 & & \\ & \ddots & \\ & & \sigma_n \end{bmatrix}$$

- σ_1 and σ_n are the largest and smallest singular values, respectively.
- Intuitively, larger values correspond to larger amounts of stretching a matrix can apply to another vector. Following this notation, σW = σ_1

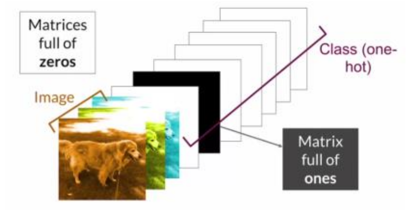
Spectral Normalization

- The spectral norm of a matrix W is the matrix's largest singular value, typically represented as σ(W).
    - Largest singular value of a matrix represent the maximum stretching factor of the matrix when it acts on a vector. It is the first entry in Σ in SVD.
- Applying SVD to Spectral Normalization:
    - Implemented as `torch.nn.utils.spectral_norm`.
    - Divide every value in the matrix by its spectral norm. As a result, a spectrally normalized matrix can be expressed as:
        - $\overline{W}_{SN} = \frac{W}{\sigma(W)}$
    - In practice, computing the SVD of W is expensive, so the authors of the SN-GAN instead approximate the left and right singular vector, ṽ and ũ respectively, through power iteration such that σ(W) ≈ ũ^T W ṽ.
    - Starting from random initialization, ũ and ṽ are updated according to:
        - $$\tilde{u} := \frac{W^{\mathsf{T}}\tilde{u}}{||W^{\mathsf{T}}\tilde{u}||_2}$$
          $$\tilde{v} := \frac{W\tilde{v}}{||W\tilde{v}||_2}$$
    - In practice, one round of iteration is sufficient to achieve satisfactory performance" as per the authors.

Unconditional generation:

- Get outputs from a random class
- Training dataset doesn't need to be labeled

Conditional generation:

- You get what you ask for
- Training dataset needs to be labeled
- Generator input:
    - The class is passed to the generator as one-hot vectors
    - (batch size, noise_vector_size + one-hot size)
- Discriminator input:
    - The class is passed to the discriminator as one-hot matrices
    - 

    - (batch size, one hot size+C, W, H)
- The discriminator receives the class label and will classify the images based on if they look like real images from that specific class or not.

Extracted code:

```
mnist image: (1,28,28)
batch size = 128
generator forward input: (128, 64)
--> fake: (128, 74, 1, 1)
image_one_hot_labels = one_hot_labels[:, :, None, None]
image_one_hot_labels.shape: torch.Size([128, 10, 1, 1])
image_one_hot_labels = image_one_hot_labels.repeat(1, 1, mnist_shape[1], mnist_shape[2])
image_one_hot_labels.shape: torch.Size([128, 10, 28, 28])
fake_image_and_labels = torch.cat([fake,image_one_hot_labels],dim=1)
fake_image_and_labels.shape: torch.Size([128, 11, 28, 28])
real_image_and_labels = torch.cat([real,image_one_hot_labels],dim=1
real_image_and_labels.shape: torch.Size([128, 11, 28, 28])
discriminatorforward input fake_image_and_label: (128,11,28,28)
--> pred: (128,1,1,1)
--> pred.view(len(pred), -1): (128,1)
discriminatorforward input real_image_and_label: (128,11,28,28)
--> pred: (128,1,1,1)
--> pred.view(len(pred), -1): (128,1)
==========================================================================================
disc_fake_loss = criterion(disc_fake_pred, torch.zeros_like(disc_fake_pred))
disc_real_loss = criterion(disc_real_pred, torch.ones_like(disc_real_pred))
disc_loss = (disc_fake_loss + disc_real_loss) / 2
disc_loss.backward(retain_graph=True)
disc_opt.step()
discriminator_losses += [disc_loss.item()]
gen_opt.zero_grad()
fake_image_and_labels = combine_vectors(fake, image_one_hot_labels)
disc_fake_pred = disc(fake_image_and_labels)
gen_loss = criterion(disc_fake_pred, torch.ones_like(disc_fake_pred))
gen_loss.backward()
gen_opt.step()
generator_losses += [gen_loss.item()]
```

Variational Lower Bound:

- Information entropy:
  - $H(X) = -\sum_{i=1}^{n} P(x_i) \log P(x_i)$
    - The amount of "information" in the distribution X.
    - The information entropy of n fair coins is n bits.
    - $H(X) = -(P(x_1)\log P(x_1) + P(x_2)\log P(x_2))$
      $H(X) = -(0.5\log 0.5 + 0.5\log 0.5)$
      $H(X) = -(0.5 \times (-1) + 0.5 \times (-1))$
      $H(X) = 1 \text{ bit}$
  - Mutual information:
    - I(X;Y) = H(X) – H(X|Y)
      - Reduction of uncertainty in X when Y is observed.

Kullback-Leibler (KL) divergence:

- to quantify the difference between two probability distributions.
- KL divergence of distribution Q from distribution P:
  - For discrete distribution:
    - $D_{KL}(P\|Q) = \sum_x P(x) \log\left(\frac{P(x)}{Q(x)}\right)$
  - For continuous distribution:
    - $D_{KL}(P\|Q) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$

InfoGan

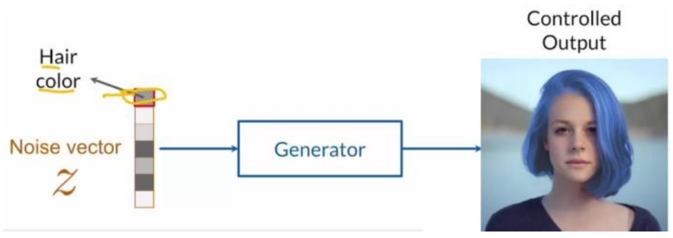
- InfoGAN: Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets: https://arxiv.org/abs/1606.03657
- to separate your model into two parts:
  - Z, corresponding to truly random noise
  - C corresponding to the "latent code."
    - a "hidden" condition in a conditional generator, and you'd like it to have an interpretable meaning.
  - InfoGan gets *c*, which is just some random set of numbers, to be more interpretable than any dimension in a typical GAN
    - mutual information:
      - want each dimension of the latent code to be as obvious a function as possible of the generated images.
- We would like to:
  - Maximize I(c;G(z,c))
    - The mutual information between the latent code *c* and the generated images G(z, c).
    - it's difficult to know P(c|G(z,c))
      - Add a second output to the discriminator to predict P(c|G(z,c))
- The Kullback-Leibler divergence between the true and approximate distribution:
  - $\Delta = D_{KL}(P(\bullet|x)||Q(\bullet|x))$

- Lower bound for the mutual information:

$$\begin{aligned} I(c; G(z,c)) &= H(c) - H(c|G(z,c)) \\ &= \mathbb{E}_{x\sim G(z,c)}[\mathbb{E}_{c'\sim P(c,x)} \log P(c'|x)] + H(c) \ \textit{(by definition of H)} \\ &= \mathbb{E}_{x\sim G(z,c)}[\Delta + \mathbb{E}_{c'\sim P(c,x)} \log Q(c'|x)] + H(c) \ \textit{(approximation error)} \\ &\geq \mathbb{E}_{x\sim G(z,c)}[\mathbb{E}_{c'\sim P(c,x)} \log Q(c'|x)] + H(c) \ \textit{(KL divergence is non-negative)} \end{aligned}$$

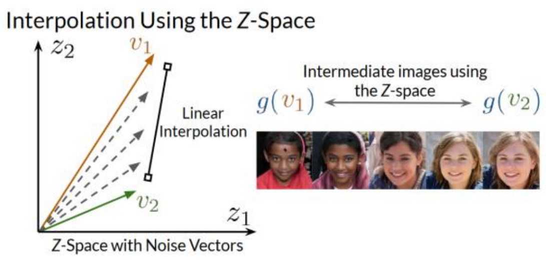
- Minimax game:
    - $$\min_G \max_D V(D,G) = \mathbb{E}(\log D(x)) + \mathbb{E}(\log(1 - D(G(z)))).$$
    - To encourage mutual information, this game is updated for Q to maximize mutual information:
        - $$\min_{G,Q} \max_D V(D,G) - \lambda I(c; G(z,c))$$

Controllable Generation

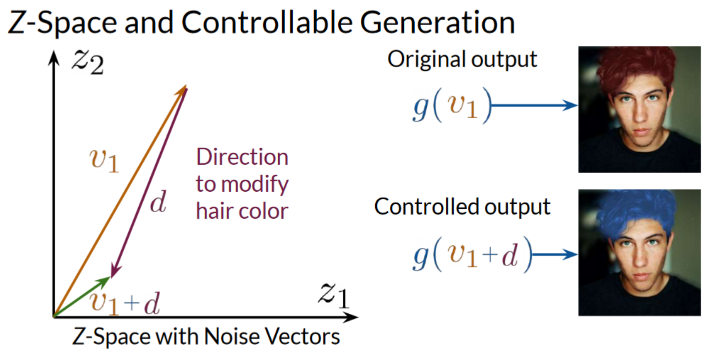
- Change specific features of the output
- Tweak the input noise vector to get different features on the output
    - 

- Controllable Generation vs Conditional Generation:
    - Controllable:
        - Examples with the features that you want.
        - Training dataset doesn't need to be labeled.
        - Manipulate the z vector input.
    - Conditional:
        - Examples from the classes you want.
        - Training dataset needs to be labeled.
        - Append a class vector to the input.

Vector Algebra in the Z-space

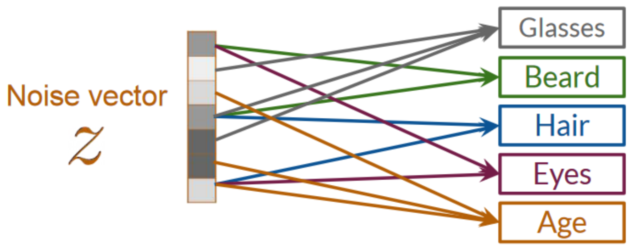
- Controllable Generation
    - Achieved by manipulating the noise vector z that's input into the generator
- Interpolation between GAN outputs
    - Intermediary examples are created between two generated observations.
        - Example: an intermediate output between digit 8 and 9
    - Process:
        - Manipulate input vectors from z-space (vector space of noise vectors)
        - 
Interpolation Using the Z-Space
z_2
v_1
Linear Interpolation
v_2
z_1
Z-Space with Noise Vectors
Intermediate images using the Z-space
$g(v_1)$
$g(v_2)$

- Z-Space and Controllable Generation:


Z-Space and Controllable Generation
z_2
v_1
d
Direction to modify hair color
v_1+d
z_1
Z-Space with Noise Vectors
Original output
$g(v_1)$
Controlled output
$g(v_1+d)$

Challenges with controllable Generation

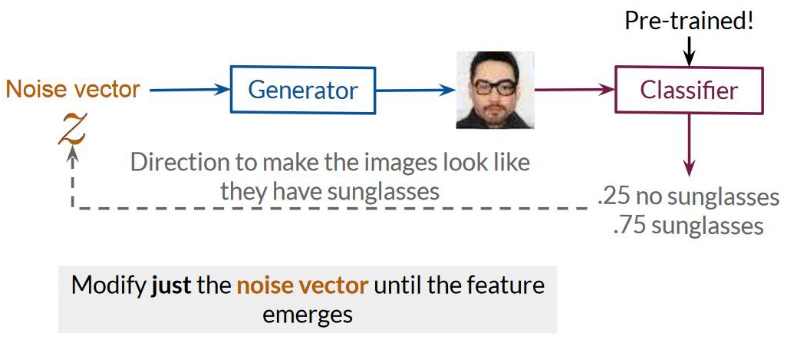
- Feature Correlation
  - Difficult in modifying a single feature without affecting correlated features.
    - In an ideal scenario, one could add a beard to a female's image by moving in a specific direction in Z-space.
    - However, due to high correlation between beard presence and masculine facial features in training datasets, adding a beard may unintentionally alter other facial features.
- Z-space Entanglement



  - Entangled Z-space leads to simultaneous changes in multiple uncorrelated features.
  - Attempting to add glasses may also influence the presence of a beard or hair.
  - Modifying apparent age might unintentionally change eye and hair color.
  - It is not possible to control single output features
  - Cause:
    - Commonly arises if Z-space has insufficient dimensions to map features individually.
    - General issue in training generative models, not only due to dimensionality but also other factors.

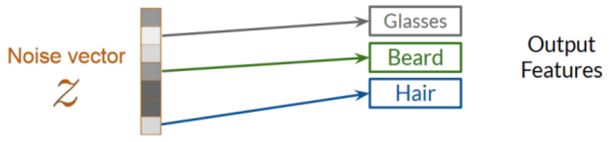
Classifier Gradient

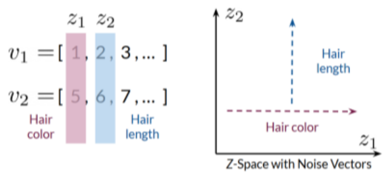
- Adjusting z-space vectors based on desired features, like changing hair length.


Pre-trained!
Noise vector
Generator
Classifier
z
Direction to make the images look like they have sunglasses
.25 no sunglasses
.75 sunglasses
Modify **just** the noise vector until the feature emerges

-
- Methodology
    - Employ a classifier to detect the presence of a feature (e.g., sunglasses) in generated images.
    - Modify noise vectors in the direction of the gradient that increases the likelihood of the feature being present, as identified by the classifier.
- Direction Identification
    - Image Generation: Pass a batch of noise vectors through the GAN generator to create images.
    - Feature Classification: Submit these images to a classifier that determines the presence of the target feature (sunglasses in the example).
    - Gradient Modification: Adjust the noise vectors in the direction of the gradient based on the classifier's feedback.
    - Repeat the process, penalizing classifications without the desired feature until the classifier detects the feature in the images.
- Requirements
    - Pre-trained Classifier: accurately detect the feature you wish to control.
    - Alternative: Train your own classifier if a suitable one is not available.
    - No Generator Modification: The generator's weights remain frozen during this process.
- Advantages and Disadvantages
    - Pros
        - Leveraging Existing Tools: Utilizes available classifiers to direct feature control.
        - Efficiency: Simplifies the process of controlling features in GAN outputs.
    - Cons
        - Dependency on Classifier: Requires a precise pre-trained classifier for the specific feature.
        - Possibility of Training Requirement: May need to train a new classifier if one doesn't exist, which can be time-consuming.

Disentanglement

- Disentangled Z-Space:
    - 

    - Latent Factors of Variation:
        - the components of noise vectors that influence specific features in the output while remaining hidden (first two in below example).
        - 
z_1 z_2
$v_1 = [1, 2, 3, \ldots]$
$v_2 = [5, 6, 7, \ldots]$
Hair color
Hair length
z_2
Hair length
Hair color
z_1
Z-Space with Noise Vectors
    - Control:
        - Changing a value in a noise vector dimension affects only the corresponding feature without altering others.
    - Desired Feature Change:
        - To modify a feature, adjust the value of the specific z-space dimension associated with it.
- Methods to Encourage Disentanglement:
    - Supervised Learning:
        - Labeling Data:
            - Incorporate information from the class into the noise vector, eliminating the need for additional class vectors.
        - Challenges:
            - Difficult with continuous classes, such as labeling hair length in thousands of faces.
        - Partial Labeling:
            - Even limited class labels can guide the generator toward disentanglement.
    - Unsupervised Learning:
        - Regularization:
            - Incorporate a regularization term in the loss function to promote association of noise vector indices with distinct output features.
            - penalizing the differences from the original class with L2 regularization. This L2 regularization would apply a penalty for this difference using the L2 norm and this would just be an additional term on the loss function.

Evaluating GANs

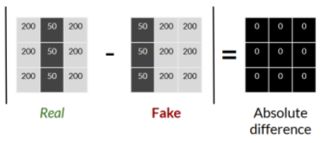
Challenges:

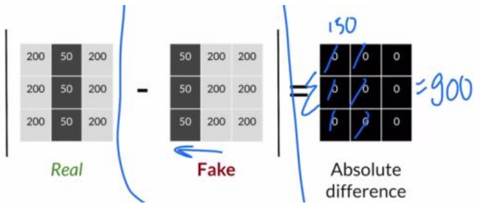
- Comparison with Supervised Learning:
    - In supervised learning, such as classification tasks, correctness can be measured against labeled datasets. GANs lack this clarity as they generate new, unlabeled data.
- GAN Output:
    - Random noise is input into the GAN, generating images without a clear standard for correctness.
- Role of Discriminator:
    - The discriminator in a GAN, which differentiates real from fake images, often overfits to the generator it's paired with, making it ineffective for evaluating other generators.

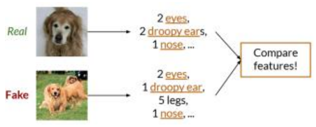
Desired properties of GANs

- Fidelity:
    - Refers to the quality and realism of generated images. High fidelity means images are realistic and crisp, not just realistic in a blurry or vague way.
    - Evaluation:
        - Measure how different a fake sample is from its closest real counterpart.
        - Look at the variation in fidelity across samples to ensure consistency, rather than having a few high-quality outputs.
- Diversity:
    - Indicates the range and variety of images a GAN can produce. A well-performing GAN should replicate the inherent diversity of the training set.
    - Evaluation:
        - Ensure that the GAN does not suffer from mode collapse, where only a limited set of outputs is generated.
        - Compare the spread of generated images to the spread of real images to assess whether the GAN captures the full diversity of the dataset.

Images Comparison

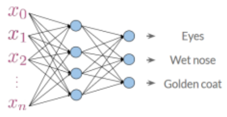
- Pixel Distance
    - Absolution difference:
        - 

        - Limitation:
            - 

- Feature Distance
  - Extract features that represent semantic information (e.g., two eyes, the position of the nose, presence of fur).
  - Compare images based on these extracted features.
  - 

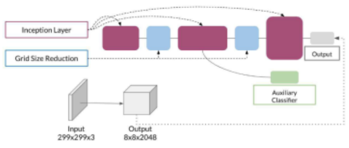
  - Advantages:
    - Less sensitive to minor variation like shifts or changes in the background.
    - Maintains the ability to discern high-level similarities and differences.

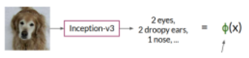
Feature Extract

- Features extracted can be used to measure feature distance
- Extracting features with classifiers.
  - 

- Using pre-trained classifiers.
  - Classifiers can be used as feature extractors by cutting the network at earlier layer.
  - The last pooling layer is most commonly used for feature extraction
  - But the last pooling layer maybe overfitted to the pre-trained dataset → try previous layers…

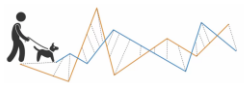
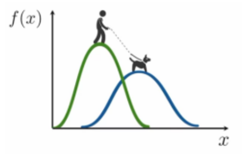
Inception-v3 and embeddings

- Architecture:
  - 

  - Method: Using Inception-v3 without the final fully connected layer
  - Output: 8x8x2048 from the last convolutional layer
  - Pooling Layer: Uses an 8x8 filter to produce a 2048-sized vector (embedding)
  - Advantage:
    - Reduces image representation from millions of pixel values to 2048 features (1000x reduction)
- Embedding:
  - Condenses image pixels to 2048 salient features.
    - 

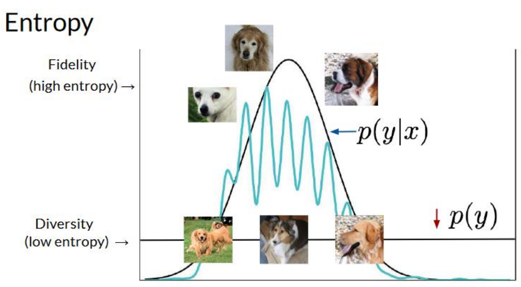
    - Similarity:
      - Embeddings with similar features will be closer in feature space.
    - Diversity:
      - Different images will have distant feature vectors.

Frechet Inception Distance (FID)

- 
- Intuition:
  - Figure out the minimum leash length you can give your dog without ever having to give them more slack during the walk.
- Univariate Normal Frechet Distance:
  - Frechant Distance between 2 normal distributions:
  - 

  - Equation:
    - $d(X,Y) = (\mu_x - \mu_y)^2 + (\sigma_x - \sigma_y)^2$
- Multivariate Normal Frechet Distance:
  - For multivariate normal distribution:
  - Equation:
    - $||\mu_x-\mu_y||^2 + Tr(\Sigma_x + \Sigma_y - 2*sqrt(\Sigma_x\Sigma_y))$
    - Σ = Covariance Matrix
- Application to GANs:
  - Real and Fake embeddings are two multivariate normal distributions
  - Real Embeddings (X):
    - μ_X : Mean of real image feature embeddings.
    - Σ_X : Covariance matrix of real image feature embeddings.
  - Fake Embeddings (Y):
    - μ_Y : Mean of fake image feature embeddings generated by GAN.
    - Σ_Y : Covariance matrix of fake image feature embeddings.
  - Comparison:
    - Fit a multivariate normal distribution to both embeddings.
    - Use the FID to calculate the distance between the two distributions
- Properties:
  - A lower FID score indicates that the fake embeddings more closely model the real embeddings.
  - This is no standard interpretable range for FID values; closer to zero is ideal.
  - Zero indicates indistinguishability between fakes and reals based on the computed statistics.
- Limitations:
  - The Inception-v3 model may not capture relevant features for certain tasks.
  - FID scores are biased and can vary with the number of samples used.
  - High computational cost due to large sample sizes.
  - Only mean and covariance statistics are used, neglecting other distribution moments like skew and kurtosis.
  - The assumption of multivariate normal distribution is approximate and may not be accurate.
  - Limited statistics used: only mean and covariance.

Entropy



- Fidelity(high entropy):
    - High probability for one class and low for others.
- Diversity(low entropy):
    - Across many samples, a variety of classes should be generated, indicating a diverse set of outputs.

Inception score:

- Aims:
    - To codify two desirable qualities of a generative model into a metric:
        - p(y|x) should be low entropy
        -
- Computed by summing over all images and averaging over all classes, then applying an exponent to create a human-readable score.

$$\mathrm{IS} = \exp(\mathbb{E}_{x\sim p_g} \underbrace{D_{KL}(p(y \mid x)\|p(y))}_{\text{KL Divergence}})$$

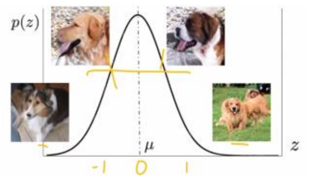
- Uses the Killback-Leibler divergence from p(y) to p(y|x) to measure the difference between the fidelity and diversity distributions.
    - How much information you can gain on p(y|x) given just p(y).
    - $$D_{KL}(p(y|x)\|p(y)) = \underbrace{p(y|x)}_{\substack{\text{Conditional distribution}\\ \text{(fidelity)}}} \log\left(\frac{p(y|x)}{\underbrace{p(y)}_{\substack{\text{Marginal distribution}\\ \text{(diversity)}}}}\right)$$
    - If marginal distribution p(y) is really uniform on different classees.
        - Difficult to guess p(y|x)
    - If marginal distribution p(y) is spiky around an area.
        - Probably guess p(y|x) was spiky around that area too.

Shortcomings:

- Exploitation: The score can be gamed by generating a single real image for each classifier class.
- Lack of Real Sample Comparison: It only evaluates generated samples without comparing them to real images.
- Classifier Dependency: Scores can be imprecise if the generated images are not closely related to the classifier's training data (e.g., ImageNet).

Sampling in Evaluation

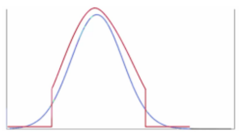
- Sampling techniques can significantly affect the evaluation outcome.
- For real images, sampling is random and uniform.
- For fake images, sampling typically uses z values from the training distribution.



Noise Vector Sampling

- GANs are usually trained with noise vectors from a normal distribution (mean = 0, standard deviation = 1).
- Sampling close to zero in the distribution leads to higher fidelity but lower diversity in generated images.

The Truncation Trick



- A post-training method to adjust the balance between fidelity and diversity.
- Cutting off the tails of the distribution at test time increases fidelity but decreases diversity.
- The extent of truncation is controlled by a hyperparameter.
- Sampling from the tails increases diversity but reduces fidelity.

Prior Noise Distributions

- Models are commonly trained on a normal distribution but can use others like uniform distribution.
- The normal distribution is favored because it allows for the application of the truncation trick.
- Experimenting with different prior distributions has not shown stark differences in outcomes.

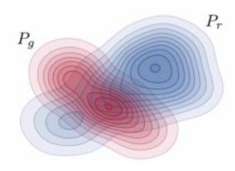
Impact on Evaluation Metrics

- FID scores are affected by the lack of diversity or fidelity.
- The truncation trick may worsen FID scores but can be useful for specific applications requiring high fidelity.

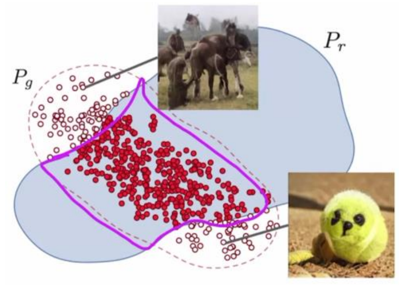
Human Evaluation: HYPE

- HYPE utilizes crowd-sourced evaluators to discern real from fake images.
- HYPE$_{time}$ tests the threshold at which an image is determined to be real or fake.
- HYPE$_{\infty}$ removes the time constraint for the evaluation.
- Quality control and managing evaluator learning effects are important for accuracy.

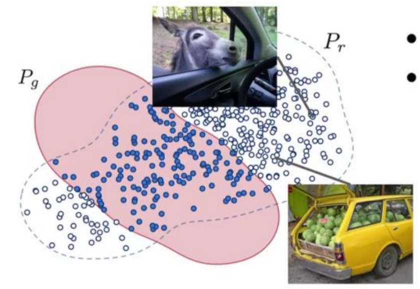
Precision and Recall

- 

- Precision:

  - Related to fidelity
  - Looks at overlap between reals and fakes, over how much extra gunk the generator produces (non-overlap red)
- Recall:

  - Related to diversity
  - Looks at overlap between reals and fakes, overa all the reals that the generator cannot model (non-overlap blue)

Evaluating GANs (Lab)

- set up generator model

```python
import torch
import numpy as np
from torch import nn
from tqdm.auto import tqdm
from torchvision import transforms
from torchvision.datasets import CelebA
from torchvision.utils import make_grid
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
torch.manual_seed(0) # Set for our testing purposes, please do not change!

class Generator(nn.Module):
    def __init__(self, z_dim=10, im_chan=3, hidden_dim=64):
        super(Generator, self).__init__()
        self.z_dim = z_dim
        self.gen = nn.Sequential(
            self.make_gen_block(z_dim, hidden_dim * 8),
            self.make_gen_block(hidden_dim * 8, hidden_dim * 4),
            self.make_gen_block(hidden_dim * 4, hidden_dim * 2),
            self.make_gen_block(hidden_dim * 2, hidden_dim),
            self.make_gen_block(hidden_dim, im_chan, kernel_size=4, final_layer=True),
        )

    def make_gen_block(self, input_channels, output_channels, kernel_size=3, stride=2, final_layer=False):
        if not final_layer:
            return nn.Sequential(
                nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride),
                nn.BatchNorm2d(output_channels),
                nn.ReLU(inplace=True),
            )
        else:
            return nn.Sequential(
                nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride),
                nn.Tanh(),
            )

    def forward(self, noise):
        x = noise.view(len(noise), self.z_dim, 1, 1)
        return self.gen(x)

def get_noise(n_samples, z_dim, device='cpu'):
    return torch.randn(n_samples, z_dim, device=device)
```

- load the pre-trained model

```python
z_dim = 64
image_size = 299
device = 'cuda'
transform = transforms.Compose([
    transforms.Resize(image_size),
    transforms.CenterCrop(image_size),
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
])
dataset = CelebA(".", download=True, transform=transform)
gen = Generator(z_dim).to(device)
gen.load_state_dict(torch.load(f"pretrained_celeba.pth", map_location=torch.device(device))["gen"])
gen = gen.eval()
```

- Inception-v3 network

```python
from torchvision.models import inception_v3
inception_model = inception_v3(pretrained=False)
inception_model.load_state_dict(torch.load("inception_v3_google-1a9a5a14.pth"))
inception_model.to(device)
inception_model = inception_model.eval() # Evaluation mode
inception_model.fc = nn.Identity()
```

- Frechet Distance
  - Univariate:
    - $d(X,Y) = (\mu_X - \mu_Y)^2 + (\sigma_X - \sigma_Y)^2$
  - Multivariate:
    - $d(X,Y) = \|\mu_X - \mu_Y\|^2 + \mathrm{Tr}\left(\Sigma_X + \Sigma_Y - 2\sqrt{\Sigma_X \Sigma_Y}\right)$

Code (multivariate):

```python
import scipy
def matrix_sqrt(x):
    y = x.cpu().detach().numpy()
    y = scipy.linalg.sqrtm(y)
    return torch.Tensor(y.real, device=x.device)

def frechet_distance(mu_x, mu_y, sigma_x, sigma_y):
    return torch.norm(mu_x-mu_y)**2 + torch.trace(sigma_x+sigma_y-2*(matrix_sqrt(sigma_x@sigma_y)))
```

- Putting it all together

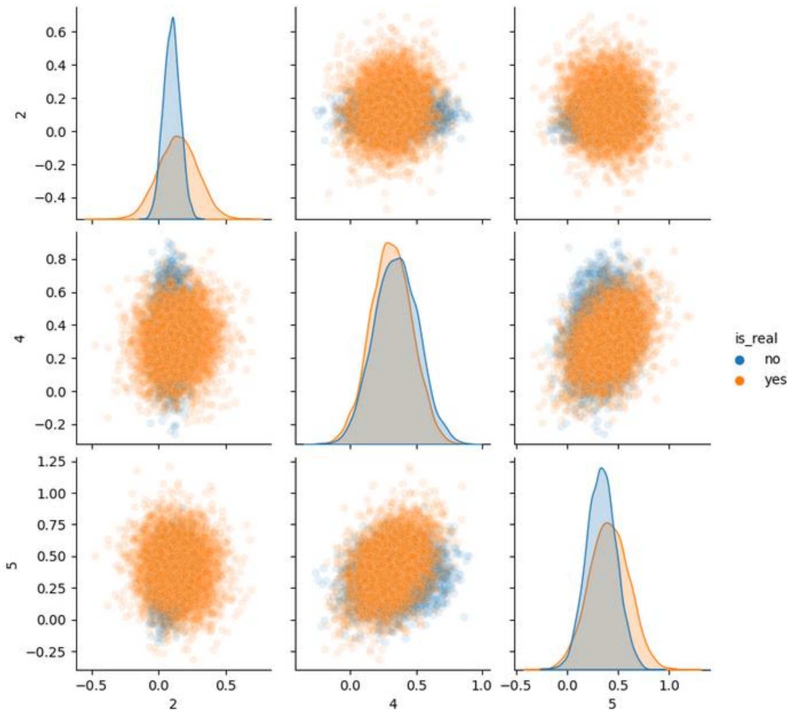
```python
def preprocess(img):
    img = torch.nn.functional.interpolate(img, size=(299, 299), mode='bilinear', align_corners=False)
    return img
def get_covariance(features):
    return torch.Tensor(np.cov(features.detach().numpy(), rowvar=False))
```

```python
fake_features_list = []
real_features_list = []
gen.eval()
n_samples = 512 # The total number of samples
batch_size = 4 # Samples per iteration
dataloader = DataLoader(
    dataset,
    batch_size=batch_size,
    shuffle=True)
cur_samples = 0
with torch.no_grad(): # You don't need to calculate gradients here, so you do this to save memory
    try:
        for real_example, _ in tqdm(dataloader, total=n_samples // batch_size): # Go by batch
            real_samples = real_example
            real_features = inception_model(real_samples.to(device)).detach().to('cpu') # Move features to CPU
            real_features_list.append(real_features)

            fake_samples = get_noise(len(real_example), z_dim).to(device)
            fake_samples = preprocess(gen(fake_samples))
            fake_features = inception_model(fake_samples.to(device)).detach().to('cpu')
            fake_features_list.append(fake_features)
            cur_samples += len(real_samples)
            if cur_samples > n_samples:
                break
    except:
        print("Error in loop")
```

```python
fake_features_all = torch.cat(fake_features_list)
real_features_all = torch.cat(real_features_list)
mu_fake = torch.mean(fake_features_all, 0)
mu_real = torch.mean(real_features_all, 0)
sigma_fake = get_covariance(fake_features_all)
sigma_real = get_covariance(real_features_all)
```

Visualization

```python
indices = [2, 4, 5]
fake_dist = MultivariateNormal(mu_fake[indices], sigma_fake[indices][:, indices])
fake_samples = fake_dist.sample((5000,))
real_dist = MultivariateNormal(mu_real[indices], sigma_real[indices][:, indices])
real_samples = real_dist.sample((5000,))

import pandas as pd
df_fake = pd.DataFrame(fake_samples.numpy(), columns=indices)
df_real = pd.DataFrame(real_samples.numpy(), columns=indices)
df_fake["is_real"] = "no"
df_real["is_real"] = "yes"
df = pd.concat([df_fake, df_real])
sns.pairplot(data = df, plot_kws={'alpha': 0.1}, hue='is_real')
plt.show()
```



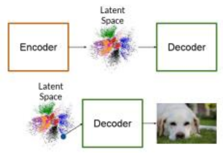
```python
with torch.no_grad():
    print(frechet_distance(mu_real, mu_fake, sigma_real, sigma_fake).item()) #>30
```

- Currently pretty high FID, lower is better.

Disadvantages of GANs

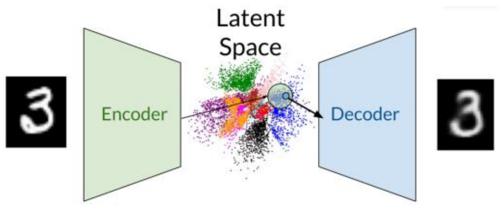
- Lack of intrinsic evaluation metrics:
  - There is no straightforward method to measure the performance of of GANs. Evaluation often requires subjective inspection of generated samples.
- Unsable training
  - GANs can suffer from unstable training and require considerable time.
- No density estimation
  - GANs do not naturally provide probability densities of features, which can be a disadvantage for tasks like anomaly detection.
- Inverting is not straightforward

Alternatives to GANs

- Generative Models:
  - $$\overset{\text{Noise}}{\xi}, \overset{\text{Class}}{Y} \rightarrow \overset{\text{Features}}{X}$$
    $$P(X|Y)$$
- VAEs
  - 

  - VAEs work with two models, an encoder and a decoder, that take a real image, find a good way of representing that image in latent space, and then reconstruct a realistic image. A GAN takes noise as input and never directly sees the real image.
  - Advantages:
    - Has density estimation
    - Invertible
    - Stable training
  - Disadvantages:
    - Lower quality results
  - Improved:
    - VQ-VAE, BigGN deep
- Autoregressive Models
  - (like RNN)
  - Based on previous pixels to generate next pixel, cannot see into future pixels.
- Flow Model
  - Uses invertible mappings between the noise and generated images
- Hybrid models
  - Apply concepts from multiple models

Variational Autoencoders (VAEs)



-
- Encoder and Decoder

```
class Encoder(nn.Module):
    def __init__(self, im_chan=1, output_chan=32, hidden_dim=16):
        super(Encoder, self).__init__()
        self.z_dim = output_chan
        self.disc = nn.Sequential(
            self.make_disc_block(im_chan, hidden_dim),
            self.make_disc_block(hidden_dim, hidden_dim * 2),
            self.make_disc_block(hidden_dim * 2, output_chan * 2, final_layer=True),
        )
    def make_disc_block(self, input_channels, output_channels, kernel_size=4, stride=2, final_layer=False):
        if not final_layer:
            return nn.Sequential(
                nn.Conv2d(input_channels, output_channels, kernel_size, stride),
                nn.BatchNorm2d(output_channels),
                nn.LeakyReLU(0.2, inplace=True),
            )
        else:
            return nn.Sequential(
                nn.Conv2d(input_channels, output_channels, kernel_size, stride),
            )
    def forward(self, image):
        disc_pred = self.disc(image)
        encoding = disc_pred.view(len(disc_pred), -1)
        return encoding[:, :self.z_dim], encoding[:, self.z_dim:].exp()
class Decoder(nn.Module):
    def __init__(self, z_dim=32, im_chan=1, hidden_dim=64):
        super(Decoder, self).__init__()
        self.z_dim = z_dim
        self.gen = nn.Sequential(
            self.make_gen_block(z_dim, hidden_dim * 4),
            self.make_gen_block(hidden_dim * 4, hidden_dim * 2, kernel_size=4, stride=1),
            self.make_gen_block(hidden_dim * 2, hidden_dim),
            self.make_gen_block(hidden_dim, im_chan, kernel_size=4, final_layer=True),
        )
    def make_gen_block(self, input_channels, output_channels, kernel_size=3, stride=2, final_layer=False):
        if not final_layer:
            return nn.Sequential(
                nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride),
                nn.BatchNorm2d(output_channels),
                nn.ReLU(inplace=True),
            )
        else:
            return nn.Sequential(
                nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride),
                nn.Sigmoid(),
            )
    def forward(self, noise):
        x = noise.view(len(noise), self.z_dim, 1, 1)
        return self.gen(x)
```

VAE - forward pass:

1. Real image input to encoder
2. Encoder outputs mean and standard deviation
3. Sample from distribution with the outputed mean and standard deviation
4. Take sampled value (vector/latent) as the input to the decoder
5. Get fake sample
6. Use reconstruction loss between the fake output of the decoder and the original real input to the encoder (more about this later - keep reading!)
7. Backpropagate through

```python
class VAE(nn.Module):
    def __init__(self, z_dim=32, im_chan=1, hidden_dim=64):
        super(VAE, self).__init__()
        self.z_dim = z_dim
        self.encode = Encoder(im_chan, z_dim)
        self.decode = Decoder(z_dim, im_chan)

    def forward(self, images):
        q_mean, q_stddev = self.encode(images)
        q_dist = Normal(q_mean, q_stddev)
        z_sample = q_dist.rsample()
        decoding = self.decode(z_sample)
        return decoding, q_dist
```

Machine Bias:

- Machine learning bias has a disproportionately negative effect on historically undeserved populations
- Proprietary risk assessment software:
    - Difficult to validate
    - Misses important considerations about people
- Define fairness

Training bias

- Training data
    - No variation in who or what is represented
    - Bias in collection methods
- Data labelling
    - Diversity of the labellers

Bias Evaluation

- Images can be biased to reflect “correctness” in the dominant culture

Model Architecture bias

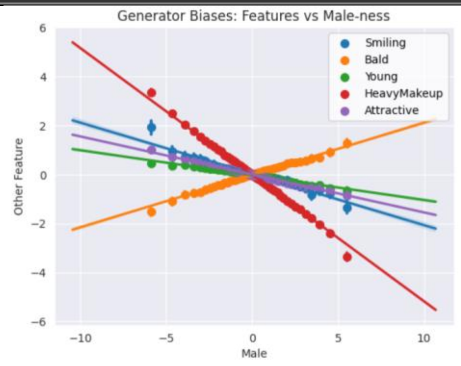
- Can be influenced by the coders who desigtned the architecture or optimized the code.

Feature Correlation

Generate a bunch of fake images with the generator

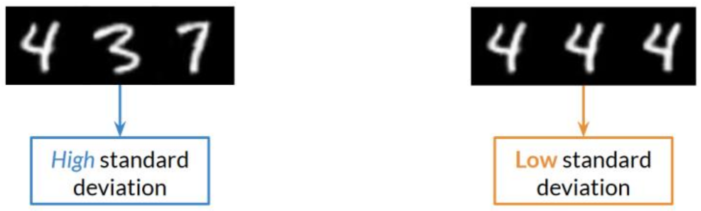
```python
n_images = 256
fake_image_history = []
classification_history = []
grad_steps = 30 # How many gradient steps to take
skip = 2 # How many gradient steps to skip in the visualization
feature_names = ["5oClockShadow", "ArchedEyebrows", "Attractive", "BagsUnderEyes", "Bald", "Bangs", "BigLips", "BigNose", "BlackHair", "BlondHair", "Blurry", "BrownHair", "BushyEyebrows", "Chubby", "DoubleChin", "Eyeglasses", "Goatee", "GrayHair", "HeavyMakeup", "HighCheekbones", "Male", "MouthSlightlyOpen", "Mustahe", "NarrowEyes", "NoBeard", "OvalFace", "PaleSkin", "PointyNose", "RecedingHairline", "RosyCheeks", "Sideburn", "Smiling", "StraightHair", "WavyHair", "WearingEarrings", "WearingHat", "WearingLipstick", "WearingNecklace", "WearingNecktie", "Young"]
n_features = len(feature_names)
target_feature = "Male"
target_indices = feature_names.index(target_feature)
noise = get_noise(n_images, z_dim).to(device)
new_noise = noise.clone().requires_grad_()
starting_classifications = classifier(gen(new_noise)).cpu().detach()
for i in range(grad_steps):
    opt.zero_grad()
    fake = gen(new_noise)
    fake_image_history += [fake]
    classifications = classifier(fake)
    classification_history += [classifications.cpu().detach()]
    fake_classes = classifications[:, target_indices].mean()
    fake_classes.backward()
    new_noise.data += new_noise.grad / grad_steps
new_noise = noise.clone().requires_grad_()
for i in range(grad_steps):
    opt.zero_grad()
    fake = gen(new_noise)
    fake_image_history += [fake]
    classifications = classifier(fake)
    classification_history += [classifications.cpu().detach()]
    fake_classes = classifications[:, target_indices].mean()
    fake_classes.backward()
    new_noise.data -= new_noise.grad / grad_steps
classification_history = torch.stack(classification_history)
```

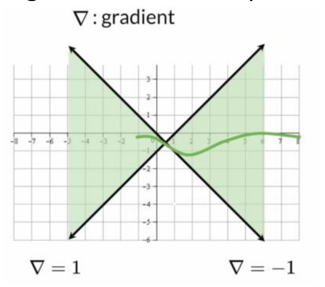
```
import seaborn as sns
other_features = ["Smiling", "Bald", "Young", "HeavyMakeup", "Attractive"]
classification_changes = (classification_history -
starting_classifications[None, :, :]).numpy()
for other_feature in other_features:
    other_indices = feature_names.index(other_feature)
    with sns.axes_style("darkgrid"):
        sns.regplot(
            x=classification_changes[:, :, target_indices].reshape(-1),
            y=classification_changes[:, :, other_indices].reshape(-1),
            fit_reg=True,
            truncate=True,
            ci=99,
            x_ci=99,
            x_bins=len(classification_history),
            label=other_feature
        )
plt.xlabel(target_feature)
plt.ylabel("Other Feature")
plt.title(f"Generator Biases: Features vs {target_feature}-ness")
plt.legend(loc=1)
plt.show()
```

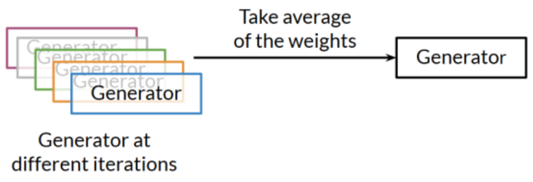


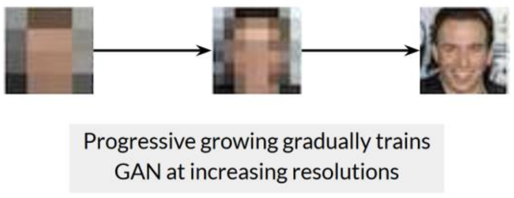
- This correlation detection can be used to reduce bias by penalizing this type of correlation in the loss during the training of the generator. However, currently there is no rigorous and accepted solution for debiasing GANs.

GANs Improvement

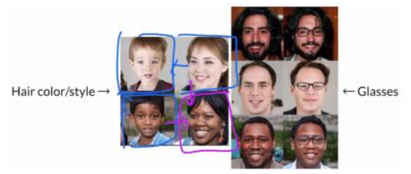
- Stability
  - Use batch standard deviation to encourage diversity
    - 

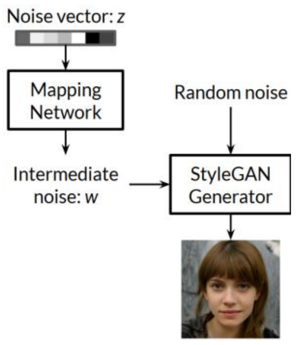
    - Pass a minibatch to the discriminator
      → disccriminator knows there is mode collapse going on.
  - Enforcing 1-Lipschitz continuity
    - E.g. WGAN-GP and Spectral Normalization
    - 

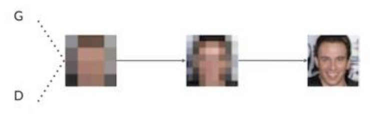
      - Prevent a model from both burning too fast and also keeping in valid W distance.
  - Moving average:
    - 
Generator
Take average of the weights
Generator
Generator at different iterations
    -

  - Progressive growing:
    - 
Progressive growing gradually trains GAN at increasing resolutions
- Capacity:
  - Larger models can use higher resolution images
- Diversity:
  - Having larger dataset

StyleGAN: Overview

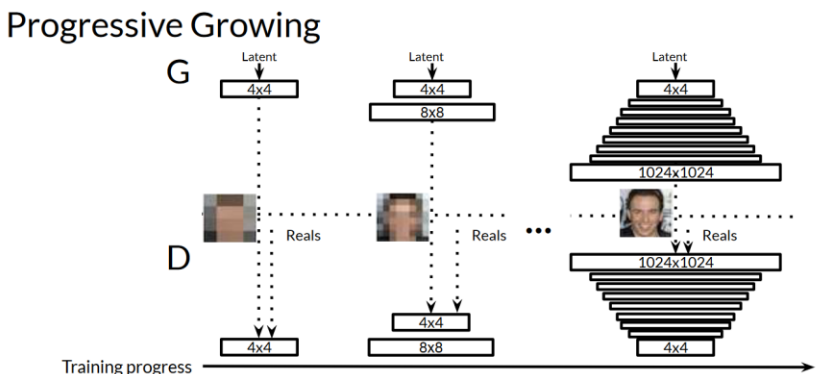
- Goals:
  - Greater fidelity on high-resolution images
  - Increased diversity of outputs
  - More control over image features
    - 

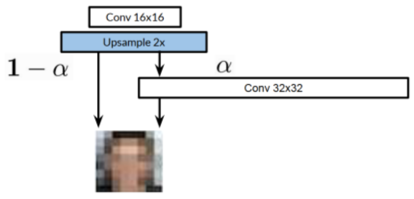
- Style in GANs
  - Can be any variation in the image, from larger, coarser styles to finer more detailed styles
  - The StyleGAN generator uses blocks that align with feature levels, where early blocks affect coarse features and later blocks affect finer details.
- StyleGAN Architecture
  - 

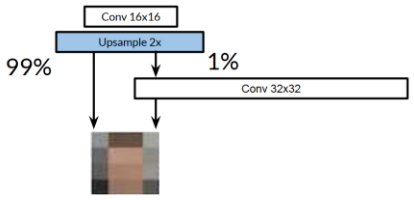
  - Unlike traditional GAN generators, StyleGAN uses a noise vector that first goes through a mapping network to generate an intermediate noise vector (W).
  - W is injected multiple times into the generator to create the image, adding controlled variations.
  - Additional random noise is introduced to provide stochastic variation, such as hair movement.
  - Backpropagation includes the mapping network as well as the discriminator and generator.
  - Adaptive Instance Normalization (AdaIN): This operation injects the intermediate noise into the generator layers, controlling various image styles.
- Progressive Growing
  - 

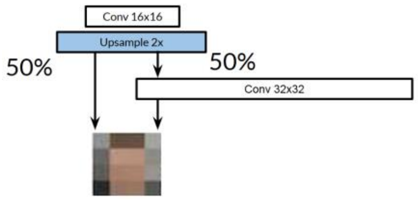
  - A technique to gradually increase the resolution during training, improving stability and quality.
  - Both the generator and discriminator start with low-resolution images and scale up as they stabilize.
  - The increase in resolution is scheduled and gradual during the training process to prevent abrupt changes that could destabilize learning.

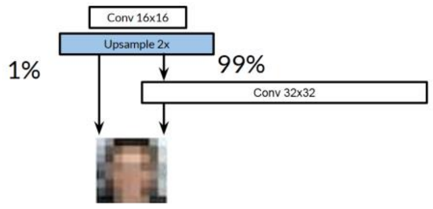
Progressive Growing

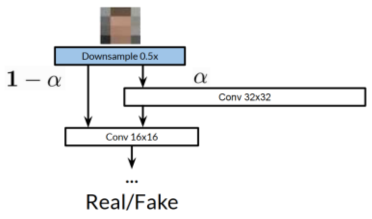


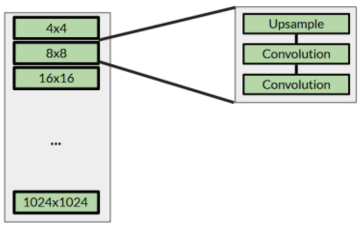
- Reals are also downsampled to respective size.
- Generator
    - 

    - Example:
        - 

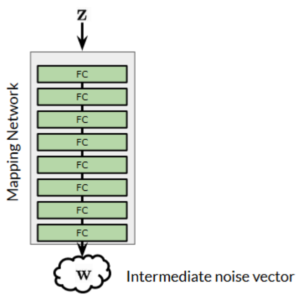
        - 1% from learned parameters → increase this one percent to rely more and more on these learned parameters →
        - 

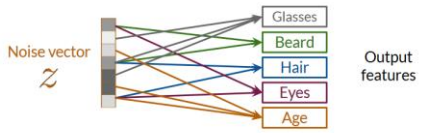
        - → until completely do not rely on this upsampling by itself aand just reply on these learned parameters (100%)
        - 

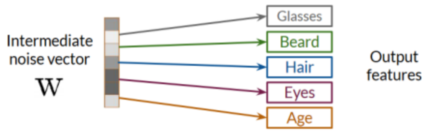
- Discriminator
  - 

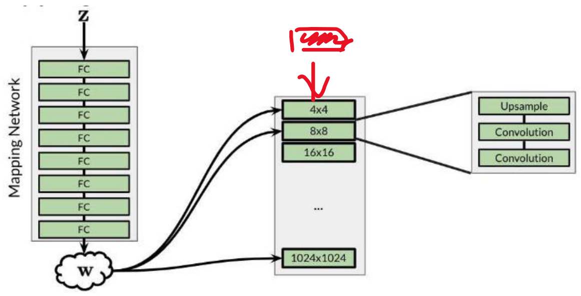
- Progressive Growing in Context
  - 

Noise Mapping Network



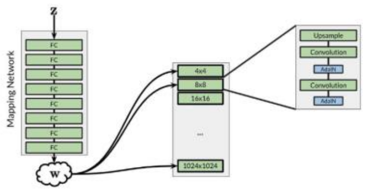
-
- Function:
    - Transforms noise vector Z into an intermediate vector W.
- Composition:
    - Consists of eight fully connected layers.
    - Includes activation functions between layers.
    - Known as a multilayer perceptron (MLP).
- Dimensions:
    - Input noise vector Z: 512-dimensional.
    - Output intermediate vector W: Maintains 512 dimensions but alters values.
- Z-space entanglement: not possible to control single output features:
    - 

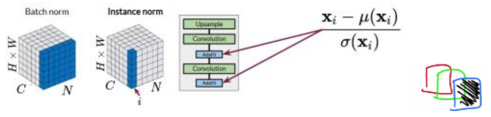
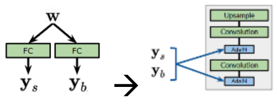
- W-space: less entangled: more possible to control single output features
    - 

- Mapping Network in Context
    - 

    - The network begins with a constant value (4x4x512) that is the same for every generated image
    - W is input at multiple points.

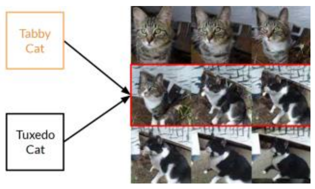
Adaptive Instance Normalization (AdaIN)

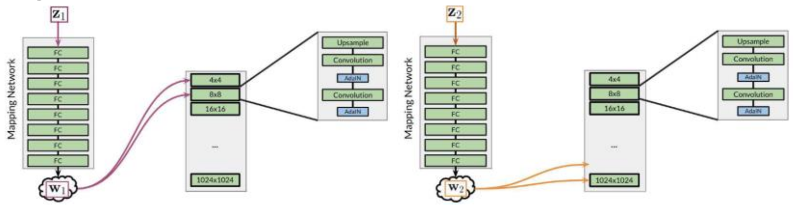
- AdaIN in context:
    - 

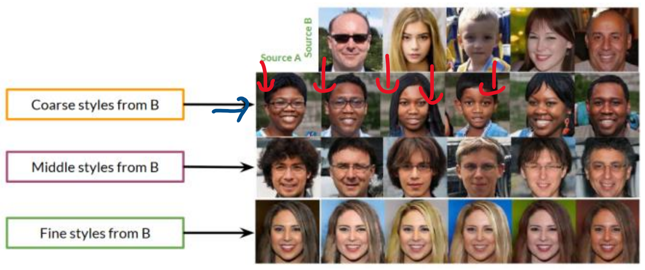
    - w is added in every blocks
- Procedure:
    - 
Batch norm
Instance norm
H × W
C
N
i
Upsample
Convolution
AdaIN
$\frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)}$
    - Step 1: Normalize convolution outputs using Instance Normalization
        - Only look at one example, one channel
    - 

    - Step 2: Apply adaptive styles using the internediate noise vector w
- Details:
    - $$\text{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \underbrace{\frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)}}_{\text{Step 1: Instance normalization}} + \mathbf{y}_{b,i}$$
    - $$\underbrace{\text{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i}}_{\text{Step 2: Adaptive styles}}$$
- Explanation:
    - Every block controls styles at that block. And at the next block, it will be overwritten by the next AdaIN that nomalizes the previous outputs.

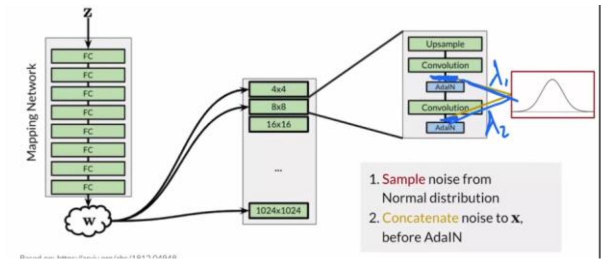
Style mixing

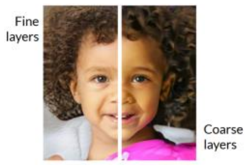
- 

- Style Mixing in Context:
  - 

    - Inject w_1 in some points, go through in AdaIN, w_2 in the other points.
    - Switch off between w_1 and w_2 can be at any point.
    - In this case, w_1 is controlling more coarse features, w_2 is controlling finer features → more diverse outputs.
- Example:
  - 

Stochastic Variation

- Stochastic Noise in Context:
  - 

- Example:
  - 

    - Injecting random noise into the final layer vs earlier layer
  -
    - Different extra noise values create stochastic variation

Components of StyleGAN

Truncation Trick

```
from scipy.stats import truncnorm
def get_truncated_noise(n_samples, z_dim, truncation):
  truncated_noise = truncnorm.rvs(-1*truncation, truncation, size=(n_samples, z_dim))
  return torch.Tensor(truncated_noise)
```

Mapping z → w

```
class MappingLayers(nn.Module):
    def __init__(self, z_dim, hidden_dim, w_dim):
        super().__init__()
        self.mapping = nn.Sequential(
            nn.Linear(z_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim, hidden_dim),
            nn.ReLU(),
            nn.Linear(hidden_dim,w_dim)
        )
    def forward(self, noise):
        return self.mapping(noise)
    def get_mapping(self):
        return self.mapping
```

Random noise Injection

- Random noise injection that occurs before every AdaIN block.

```
class InjectNoise(nn.Module):
    def __init__(self, channels):
        super().__init__()
        self.weight = nn.Parameter(
            torch.randn(channels)[None, :, None, None]
        )
    def forward(self, image):
        noise_shape = (image.shape[0], 1, image.shape[2], image.shape[3])
        noise = torch.randn(noise_shape, device=image.device)
        return image + self.weight * noise
```

Adaptive Instance Normalization (AdaIN)

$$\mathrm{AdaIN}(\mathbf{x}_i, \mathbf{y}) = \mathbf{y}_{s,i} \frac{\mathbf{x}_i - \mu(\mathbf{x}_i)}{\sigma(\mathbf{x}_i)} + \mathbf{y}_{b,i}$$

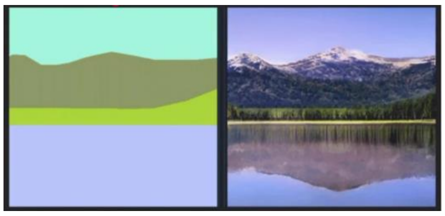
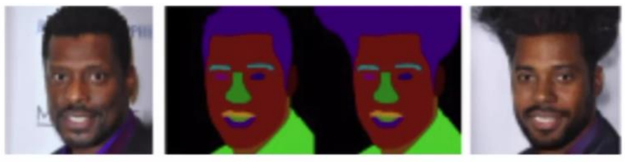
```
class AdaIN(nn.Module):
    def __init__(self, channels, w_dim):
        super().__init__()
        self.instance_norm = nn.InstanceNorm2d(channels)
        self.style_scale_transform = nn.Linear(w_dim, channels)
        self.style_shift_transform = nn.Linear(w_dim, channels)
    def forward(self, image, w):
        normalized_image = self.instance_norm(image)
        style_scale = self.style_scale_transform(w)[:, :, None, None]
        style_shift = self.style_shift_transform(w)[:, :, None, None]
        transformed_image = style_scale*normalized_image + style_shift
        return transformed_image
```

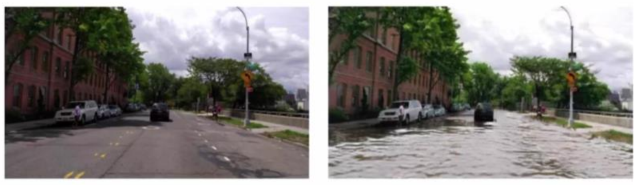
Progressive growing

```python
class MicroStyleGANGeneratorBlock(nn.Module):
    def __init__(self, in_chan, out_chan, w_dim, kernel_size, starting_size, use_upsample=True):
        super().__init__()
        self.use_upsample = use_upsample
        if self.use_upsample:
            self.upsample = nn.Upsample((starting_size), mode='bilinear')
        self.conv = nn.Conv2d(in_chan, out_chan, kernel_size, padding=1)
        self.inject_noise = InjectNoise(out_chan)
        self.adain = AdaIN(out_chan, w_dim)
        self.activation = nn.LeakyReLU(0.2)
    def forward(self, x, w):
        if self.use_upsample:
            x = self.upsample(x)
        x = self.conv(x)
        x = self.inject_noise(x)
        x = self.adain(x, w)
        x = self.activation(x)
        return x
```

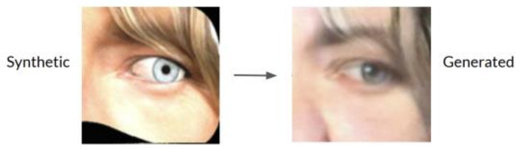
```python
class MicroStyleGANGenerator(nn.Module):
    def __init__(self,
                 z_dim,
                 map_hidden_dim,
                 w_dim,
                 in_chan,
                 out_chan,
                 kernel_size,
                 hidden_chan):
        super().__init__()
        self.map = MappingLayers(z_dim, map_hidden_dim, w_dim)
        self.starting_constant = nn.Parameter(torch.randn(1, in_chan, 4, 4))
        self.block0 = MicroStyleGANGeneratorBlock(in_chan, hidden_chan, w_dim, kernel_size, 4, use_upsample=False)
        self.block1 = MicroStyleGANGeneratorBlock(hidden_chan, hidden_chan, w_dim, kernel_size, 8)
        self.block2 = MicroStyleGANGeneratorBlock(hidden_chan, hidden_chan, w_dim, kernel_size, 16)
        self.block1_to_image = nn.Conv2d(hidden_chan, out_chan, kernel_size=1)
        self.block2_to_image = nn.Conv2d(hidden_chan, out_chan, kernel_size=1)
        self.alpha = 0.2
    def upsample_to_match_size(self, smaller_image, bigger_image):
        return F.interpolate(smaller_image, size=bigger_image.shape[-2:], mode='bilinear')
    def forward(self, noise, return_intermediate=False):
        x = self.starting_constant
        w = self.map(noise)
        x = self.block0(x, w)
        x_small = self.block1(x, w) # First generator run output
        x_small_image = self.block1_to_image(x_small)
        x_big = self.block2(x_small, w) # Second generator run output
        x_big_image = self.block2_to_image(x_big)
        x_small_upsample = self.upsample_to_match_size(x_small_image, x_big_image)
        interpolation = self.alpha * (x_big_image) + (1-self.alpha) * (x_small_upsample)
        if return_intermediate:
            return interpolation, x_small_upsample, x_big_image
        return interpolation
```

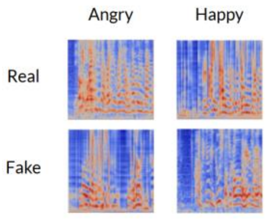
Overview of GAN Applications

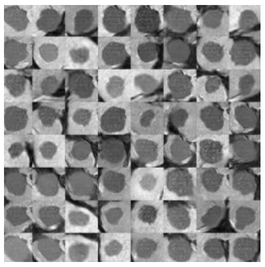
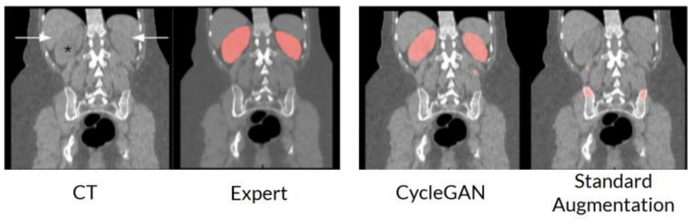
- Image-to-image translation – and extensions to other modalities such as text, audio, and video
    - GauGAN
        - 
    - Super-Resolution GAN
    - Multimodal image-to-image translation
    - Text-to-image
    - Image + face landmarks ➔ talking heads
- Image editing, art, and media
    - Image filters (instagram filters)
    - Image editing software
        - 

    - Stylized images
        - Democratized art
    - Data augmentation
    - Deepfakes:
        -
- Medicine and climate change applications
    - Simulating tissues
    - Climate change:
        - 

Data Augmentation: Method and Uses

- Use cases:
  - Gaze detection https://arxiv.org/abs/1711.09767
  - 

    - Synthetic(fake eye)
  - Speech emotion recognition
  - 

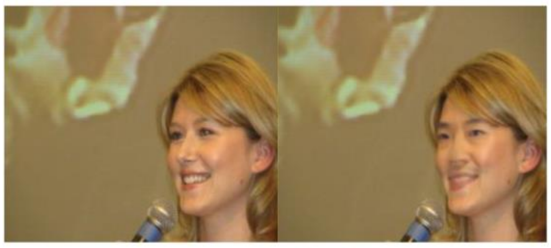
  - https://pdfs.semanticscholar.org/395b/ea6f025e599db710893acb6321e2a1898a1f.pdf
  - Synthetic liver lesions: https://arxiv.org/abs/1803.01229
  - 
- Pros:
  - Better than hand-crafted synthetic examples
  - Generate more labeled examples
  - Improve downstream model generalization
  - https://www.nature.com/articles/s41598-019-52737-x/figures/3
    - 

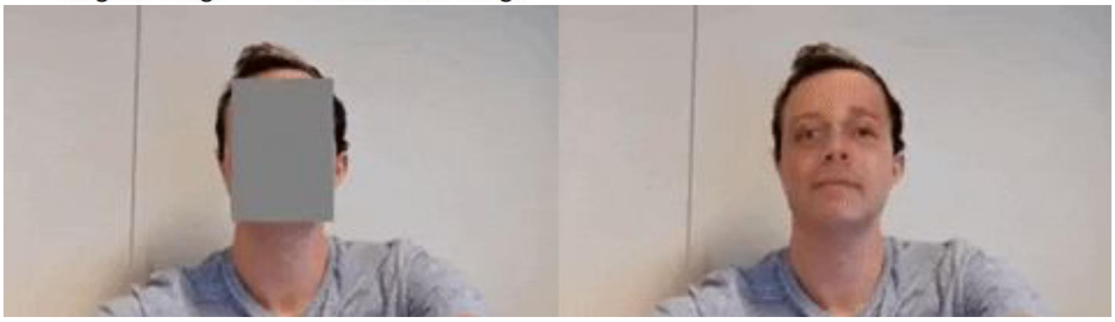
- Cons:
  - Diversity is limited to the data available
  - Can overfit to the real training data
  - Not useful when overfit to real data

GANs for privacy

- Pros:
  - Training with GAN data approaches real data accuracy
- Cons:
  - GAN sample is nearly identical to a real sample.

GANs for Anonymity

- 

- 
- Pros:
  - Provide safe environment for expression to
    - Stigmatized groups
    - Assault victims
    - Witnesses
    - Activists
- Cons:
  - Deepfakes put words into people’s mouths

Data Augmentation

Generator:

```
class Generator(nn.Module):
    def __init__(self, input_dim=10, im_chan=3, hidden_dim=64):
        super(Generator, self).__init__()
        self.input_dim = input_dim
        self.gen = nn.Sequential(
            self.make_gen_block(input_dim, hidden_dim * 4, kernel_size=4),
            self.make_gen_block(hidden_dim * 4, hidden_dim * 2, kernel_size=4, stride=1),
            self.make_gen_block(hidden_dim * 2, hidden_dim, kernel_size=4),
            self.make_gen_block(hidden_dim, im_chan, kernel_size=2, final_layer=True),
        )
    def make_gen_block(self, input_channels, output_channels, kernel_size=3, stride=2, final_layer=False):
        if not final_layer:
            return nn.Sequential(
                nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride),
                nn.BatchNorm2d(output_channels),
                nn.ReLU(inplace=True),
            )
        else:
            return nn.Sequential(
                nn.ConvTranspose2d(input_channels, output_channels, kernel_size, stride),
                nn.Tanh(),
            )
    def forward(self, noise):
        x = noise.view(len(noise), self.input_dim, 1, 1)
        return self.gen(x)
def get_noise(n_samples, input_dim, device='cpu'):
    return torch.randn(n_samples, input_dim, device=device)
def combine_vectors(x, y):
    return torch.cat([x, y], 1)
def get_one_hot_labels(labels, n_classes):
    return F.one_hot(labels, n_classes)
```

Classifier:

```
class Classifier(nn.Module):
    def __init__(self, im_chan, n_classes, hidden_dim=32):
        super(Classifier, self).__init__()
        self.disc = nn.Sequential(
            self.make_classifier_block(im_chan, hidden_dim),
            self.make_classifier_block(hidden_dim, hidden_dim * 2),
            self.make_classifier_block(hidden_dim * 2, hidden_dim * 4),
            self.make_classifier_block(hidden_dim * 4, n_classes, final_layer=True),
        )
    def make_classifier_block(self, input_channels, output_channels, kernel_size=3, stride=2, final_layer=False):
        if not final_layer:
            return nn.Sequential(
                nn.Conv2d(input_channels, output_channels, kernel_size, stride),
                nn.BatchNorm2d(output_channels),
                nn.LeakyReLU(0.2, inplace=True),
            )
        else:
            return nn.Sequential(
                nn.Conv2d(input_channels, output_channels, kernel_size, stride),
            )
    def forward(self, image):
        class_pred = self.disc(image)
        return class_pred.view(len(class_pred), -1)
```

Combine sample

```
def combine_sample(real, fake, p_real):
    make_fake = torch.rand(len(real)) > p_real
    target_images = real.clone()
    target_images[make_fake] = fake[make_fake]
    return target_images
```

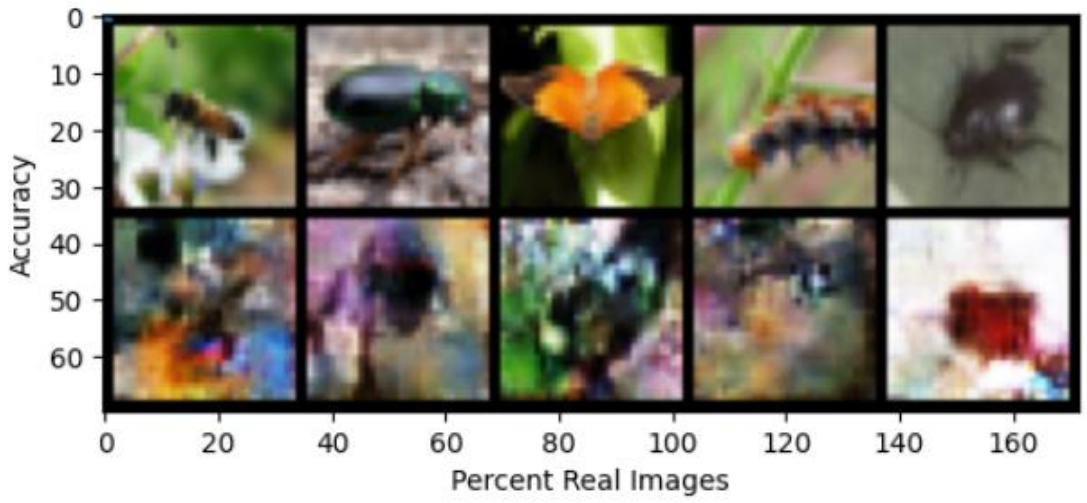
Training

```python
def find_optimal():
    gen_names = [
        "gen_1.pt",
        "gen_2.pt",
        "gen_3.pt",
        "gen_4.pt"
    ]
    best_p_real, best_gen_name = 0.6, "gen_4.pt"
    return best_p_real, best_gen_name

def augmented_train(p_real, gen_name):
    gen = Generator(generator_input_dim).to(device)
    gen.load_state_dict(torch.load(gen_name))
    classifier = Classifier(cifar100_shape[0], n_classes).to(device)
    classifier.load_state_dict(torch.load("class.pt"))
    criterion = nn.CrossEntropyLoss()
    batch_size = 256
    train_set = torch.load("insect_train.pt")
    val_set = torch.load("insect_val.pt")
    dataloader = DataLoader(
        torch.utils.data.TensorDataset(train_set["images"], train_set["labels"]),
        batch_size=batch_size,
        shuffle=True
    )
    validation_dataloader = DataLoader(
        torch.utils.data.TensorDataset(val_set["images"], val_set["labels"]),
        batch_size=batch_size
    )
    display_step = 1
    lr = 0.0002
    n_epochs = 20
    classifier_opt = torch.optim.Adam(classifier.parameters(), lr=lr)
    cur_step = 0
    best_score = 0
    for epoch in range(n_epochs):
        for real, labels in dataloader:
            real = real.to(device)
            labels = labels.to(device)
            one_hot_labels = get_one_hot_labels(labels.to(device), n_classes).float()
            classifier_opt.zero_grad()
            cur_batch_size = len(labels)
            fake_noise = get_noise(cur_batch_size, z_dim, device=device)
            noise_and_labels = combine_vectors(fake_noise, one_hot_labels)
            fake = gen(noise_and_labels)

            target_images = combine_sample(real.clone(), fake.clone(), p_real)
            labels_hat = classifier(target_images.detach())
            classifier_loss = criterion(labels_hat, labels)
            classifier_loss.backward()
            classifier_opt.step()
            if cur_step % display_step == 0 and cur_step > 0:
                classifier_val_loss = 0
                classifier_correct = 0
                num_validation = 0
                with torch.no_grad():
                    for val_example, val_label in validation_dataloader:
                        cur_batch_size = len(val_example)
                        num_validation += cur_batch_size
                        val_example = val_example.to(device)
                        val_label = val_label.to(device)
                        labels_hat = classifier(val_example)
                        classifier_val_loss += criterion(labels_hat, val_label) * cur_batch_size
                        classifier_correct += (labels_hat.argmax(1) == val_label).float().sum()
                    accuracy = classifier_correct.item() / num_validation
                    if accuracy > best_score:
                        best_score = accuracy
            cur_step += 1
    return best_score
def eval_augmentation(p_real, gen_name, n_test=20):
    total = 0
    for i in range(n_test):
        total += augmented_train(p_real, gen_name)
    return total / n_test
best_p_real, best_gen_name = find_optimal()
performance = eval_augmentation(best_p_real, best_gen_name)
print(f"Your model had an accuracy of {performance:0.1%}")
assert performance > 0.512
print("Success!")
```

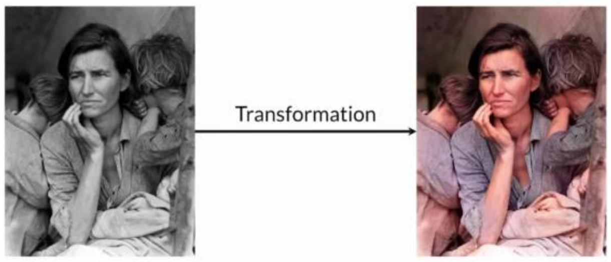
```
accuracies = []
p_real_all = torch.linspace(0, 1, 21)
for p_real_vis in tqdm(p_real_all):
    accuracies += [eval_augmentation(p_real_vis, best_gen_name, n_test=4)]
plt.plot(p_real_all.tolist(), accuracies)
plt.ylabel("Accuracy")
_ = plt.xlabel("Percent Real Images")
```

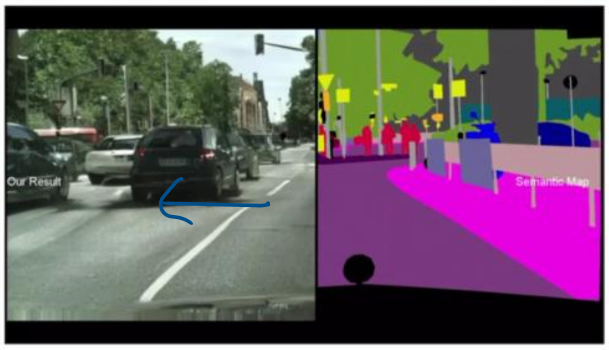
```
def show_tensor_images(image_tensor, num_images=25, size=(3, 32, 32), nrow=5, show=True):
    '''
    Function for visualizing images: Given a tensor of images, number of images, and
    size per image, plots and prints the images in an uniform grid.
    '''
    image_tensor = (image_tensor + 1) / 2
    image_unflat = image_tensor.detach().cpu()
    image_grid = make_grid(image_unflat[:num_images], nrow=nrow)
    plt.imshow(image_grid.permute(1, 2, 0).squeeze())
    if show:
        plt.show()
```

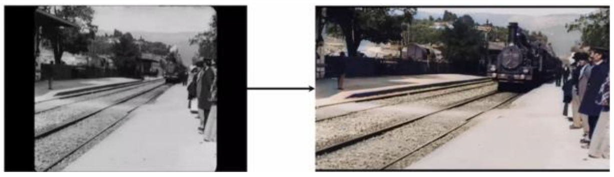
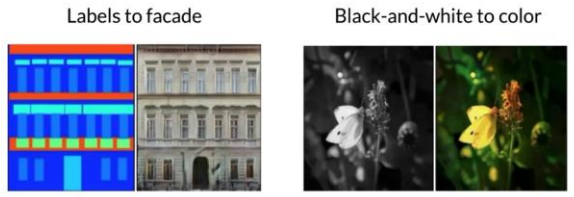
```
examples = [4, 41, 80, 122, 160]
train_images = torch.load("insect_train.pt")["images"][examples]
train_labels = torch.load("insect_train.pt")["labels"][examples]
one_hot_labels = get_one_hot_labels(train_labels.to(device), n_classes).float()
fake_noise = get_noise(len(train_images), z_dim, device=device)
noise_and_labels = combine_vectors(fake_noise, one_hot_labels)
gen = Generator(generator_input_dim).to(device)
gen.load_state_dict(torch.load(best_gen_name))
fake = gen(noise_and_labels)
show_tensor_images(torch.cat([train_images.cpu(), fake.cpu()]))
```

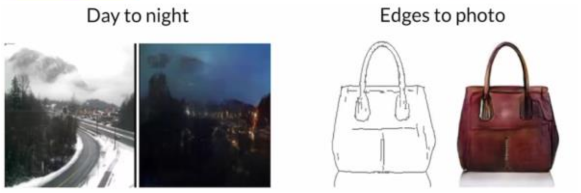


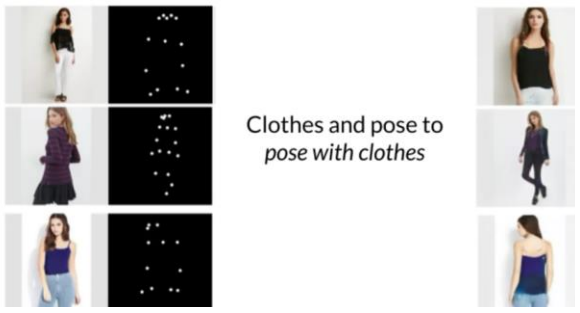
Pix2Pix:

- Image-to-Image Translation
    - 

    - 

    - 
- Paired image-to-image translation
    - 

    - 

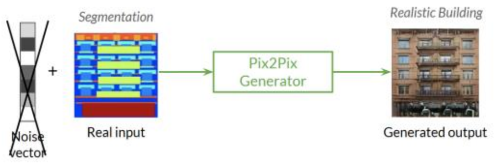
    - 

- Text – to – image
- Neural talking head

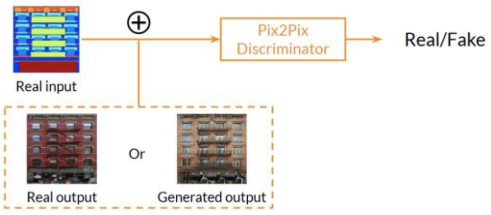
Pix2Pix Overview

https://arxiv.org/abs/1611.07004

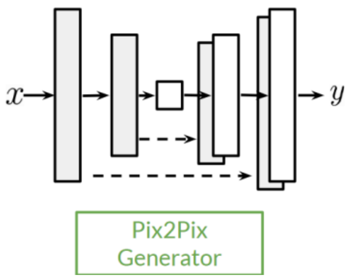
Generator:

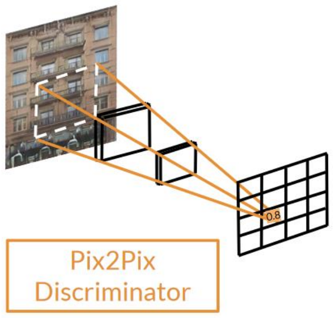
- 

Discriminator:

- 

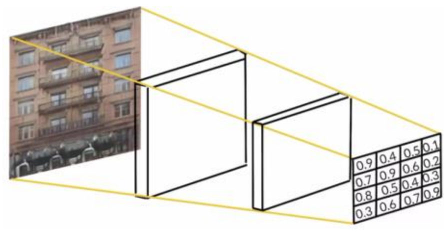
Upgrades:

- 

- https://arxiv.org/abs/1611.07004
- 

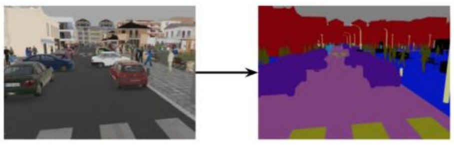
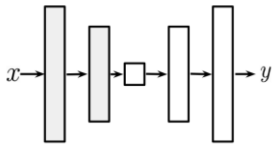
- https://arxiv.org/abs/1803.07422

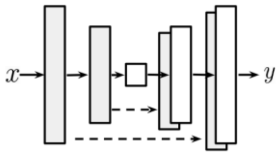
Pix2Pix Discriminator: PatchGAN



- Loss: BCE loss
- Values inside the matrix → the probability of that patch being real.
- Fake Image input → all values inside the matrix close to zero, vice veresa.

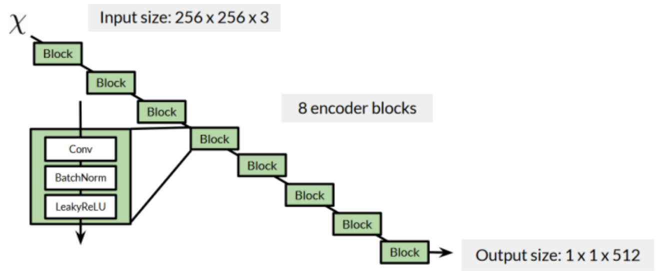
Pix2Pix: U-Net

- Image segmentation:
  - 
    - The segmentation map of the car can be of different types (no correct answer)
- U-Net Framework:
  - Encoder-Decoder
    - 

  - Skip Connections
    - 

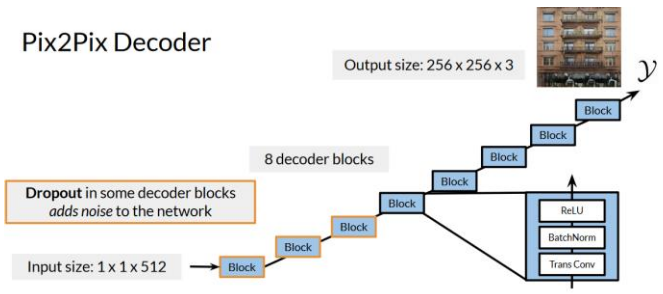
    - Forward Pass:
      - Spatial Details: Retains fine-grained spatial information for precise segmentation.
      - Contextual Integration: Combines low-level details with high-level semantic information.
    - Backward Pass:
      - Gradient Flow: Enhances gradient propagation to earlier layers, preventing vanishing gradients.
      - Training Stability: Improves network training stability and convergence.

Pix2Pix Encoder:

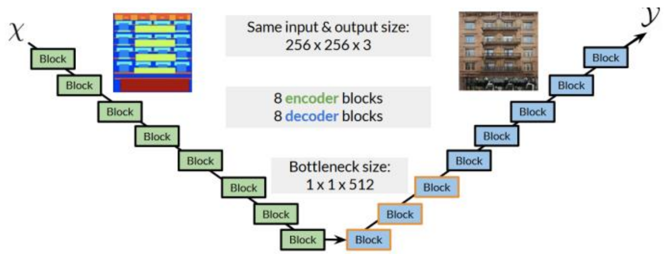
- 

conv's stride: 2

Pix2Pix Decoder:

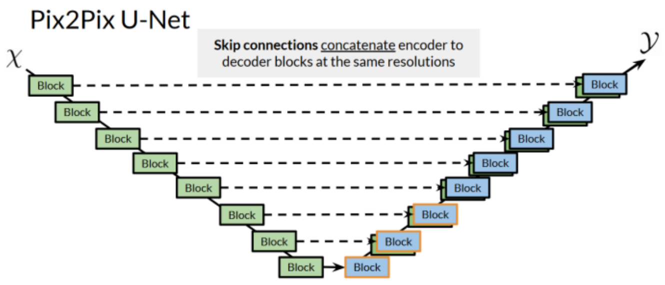
- 

Pix2Pix Encoder-Decoder:

- 

χ
Same input & output size: 256 x 256 x 3
y
8 encoder blocks
8 decoder blocks
Bottleneck size: 1 x 1 x 512
Block

Pix2Pix U-Net:

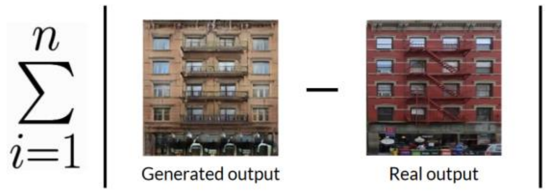
- 

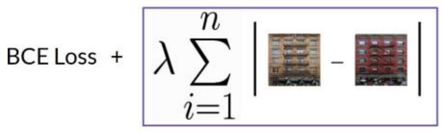
Pix2Pix U-Net
Skip connections concatenate encoder to decoder blocks at the same resolutions
χ
y
Block

U-Net summary

- Pix2Pix's generator is a U-Net
- U-Net is an encoder-decoder, with same-size inputs and outputs
- U-Net uses skip connections

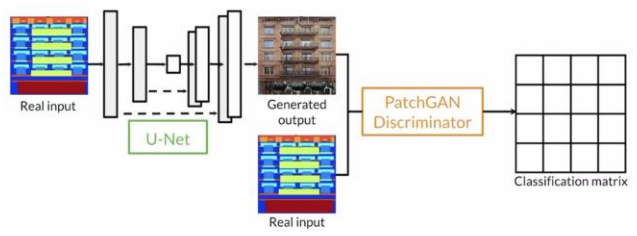
Pix2Pix: Pixel Distance Loss term

- $$\min_{g} \max_{c} \text{ Adversarial Loss} + \lambda * \text{Pixel loss term}$$
- Pixel distance used for L1 regularization to encourage generated image and real output image to be similar.
- 

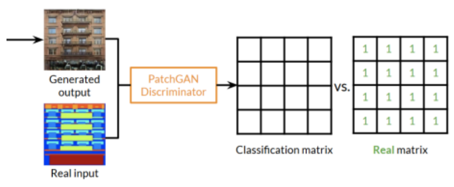
- Pix2Pix Generator loss:
    - 

- Pix2Pix adds a Pixel Distance Loss term to the generator loss function
- This loss term calculates the difference between the fake and the real target outputs
- Softly encourages the generator with this additional supervision
    - The target output labels are the supervision
    - Generator essentially “sees” these labels

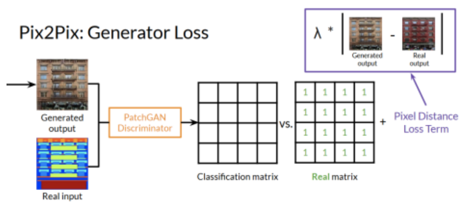
Pix2Pix: Putting it all together



Discriminator loss:

- 

Generator Loss:

- 

Pix2Pix Advancements

- Pix2PixHD
- GauGAN

U-Net Assignment

```python
class ContractingBlock(nn.Module):
    def __init__(self, input_channels):
        super(ContractingBlock, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, input_channels*2, kernel_size=3)
        self.conv2 = nn.Conv2d(input_channels*2, input_channels*2, kernel_size=3)
        self.activation = nn.ReLU()
        self.maxpool = nn.MaxPool2d(kernel_size=2, stride=2)
    def forward(self, x):
        x = self.conv1(x)
        x = self.activation(x)
        x = self.conv2(x)
        x = self.activation(x)
        x = self.maxpool(x)
        return x
    def get_self(self):
        return self
```

Crop function:

- Crop the image from the contracting path and concatenate it to the current image on the expanding path → form a skip connection.

```python
def crop(image, new_shape):
    middle_height = image.shape[2] // 2
    middle_width = image.shape[3] // 2
    starting_height = middle_height - new_shape[2] // 2
    final_height = starting_height + new_shape[2]
    starting_width = middle_width - new_shape[3] // 2
    final_width = starting_width + new_shape[3]
    cropped_image = image[:,:,starting_height:final_height, starting_width:final_width]
    return cropped_image
```

ExpandingBlock:

```python
class ExpandingBlock(nn.Module):
    def __init__(self, input_channels):
        super(ExpandingBlock, self).__init__()
        self.upsample = nn.Upsample(scale_factor=2, mode='bilinear', align_corners=True)
        self.conv1 = nn.Conv2d(input_channels, input_channels // 2, kernel_size=2)
        self.conv2 = nn.Conv2d(input_channels, input_channels // 2, kernel_size=3)
        self.conv3 = nn.Conv2d(input_channels // 2, input_channels //2, kernel_size=3)
        self.activation = nn.ReLU() # "each followed by a ReLU"
    def forward(self, x, skip_con_x):
        x = self.upsample(x)
        x = self.conv1(x)
        skip_con_x = crop(skip_con_x, x.shape)
        x = torch.cat([x, skip_con_x], axis=1)
        x = self.conv2(x)
        x = self.activation(x)
        x = self.conv3(x)
        x = self.activation(x)
        return x
```

Feature Map Block (final layer of unet)

- Map each pixel to a pixel using a 1x1 conv

```python
class FeatureMapBlock(nn.Module):
    def __init__(self, input_channels, output_channels):
        super(FeatureMapBlock, self).__init__()
        self.conv = nn.Conv2d(input_channels, output_channels, kernel_size=1)
    def forward(self, x):
        x = self.conv(x)
        return x
```

Unet

```python
class UNet(nn.Module):
    def __init__(self, input_channels, output_channels, hidden_channels=64):
        super(UNet, self).__init__()
        self.upfeature = FeatureMapBlock(input_channels, hidden_channels)
        self.contract1 = ContractingBlock(hidden_channels)
        self.contract2 = ContractingBlock(hidden_channels * 2)
        self.contract3 = ContractingBlock(hidden_channels * 4)
        self.contract4 = ContractingBlock(hidden_channels * 8)
        self.expand1 = ExpandingBlock(hidden_channels * 16)
        self.expand2 = ExpandingBlock(hidden_channels * 8)
        self.expand3 = ExpandingBlock(hidden_channels * 4)
        self.expand4 = ExpandingBlock(hidden_channels * 2)
        self.downfeature = FeatureMapBlock(hidden_channels, output_channels)
    def forward(self, x):
        x0 = self.upfeature(x)
        x1 = self.contract1(x0)
        x2 = self.contract2(x1)
        x3 = self.contract3(x2)
        x4 = self.contract4(x3)
        x5 = self.expand1(x4, x3)
        x6 = self.expand2(x5, x2)
        x7 = self.expand3(x6, x1)
        x8 = self.expand4(x7, x0)
        xn = self.downfeature(x8)
        return xn
```

```python
def train():
    dataloader = DataLoader(
        dataset,
        batch_size=batch_size,
        shuffle=True)
    unet = UNet(input_dim, label_dim).to(device)
    unet_opt = torch.optim.Adam(unet.parameters(), lr=lr)
    cur_step = 0
    for epoch in range(n_epochs):
        for real, labels in tqdm(dataloader):
            cur_batch_size = len(real)
            # Flatten the image
            real = real.to(device)
            labels = labels.to(device)
            unet_opt.zero_grad()
            pred = unet(real)
            unet_loss = criterion(pred, labels)
            unet_loss.backward()
            unet_opt.step()
            if cur_step % display_step == 0:
                print(f"Epoch {epoch}: Step {cur_step}: U-Net loss: {unet_loss.item()}")
                show_tensor_images(
                    crop(real, torch.Size([len(real), 1, target_shape, target_shape])),
                    size=(input_dim, target_shape, target_shape)
                )
                show_tensor_images(labels, size=(label_dim, target_shape, target_shape))
                show_tensor_images(torch.sigmoid(pred), size=(label_dim, target_shape, target_shape))
            cur_step += 1
```

Pix2Pix Lab

```
def show_tensor_images(image_tensor, num_images=25, size=(1, 28, 28)):
    image_shifted = image_tensor
    image_unflat = image_shifted.detach().cpu().view(-1, *size)
    image_grid = make_grid(image_unflat[:num_images], nrow=5)
    plt.imshow(image_grid.permute(1, 2, 0).squeeze())
    plt.show()
```

PatchGan Discriminator

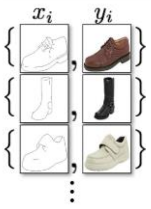
```
class Discriminator(nn.Module):
    def __init__(self, input_channels, hidden_channels=8):
        super(Discriminator, self).__init__()
        self.upfeature = FeatureMapBlock(input_channels, hidden_channels)
        self.contract1 = ContractingBlock(hidden_channels, use_bn=False)
        self.contract2 = ContractingBlock(hidden_channels * 2)
        self.contract3 = ContractingBlock(hidden_channels * 4)
        self.contract4 = ContractingBlock(hidden_channels * 8)
        self.final = nn.Conv2d(hidden_channels * 16, 1, kernel_size=1)
    def forward(self, x, y):
        x = torch.cat([x, y], axis=1)
        x0 = self.upfeature(x)
        x1 = self.contract1(x0)
        x2 = self.contract2(x1)
        x3 = self.contract3(x2)
        x4 = self.contract4(x3)
        xn = self.final(x4)
        return xn
```

```
def get_gen_loss(gen, disc, real, condition, adv_criterion, recon_criterion, lambda_recon):
    fake = gen(condition)
    disc_fake_hat = disc(fake, condition)
    gen_adv_loss = adv_criterion(disc_fake_hat, torch.ones_like(disc_fake_hat))
    gen_rec_loss = recon_criterion(real, fake)
    gen_loss = gen_adv_loss + lambda_recon * gen_rec_loss
    return gen_loss
```

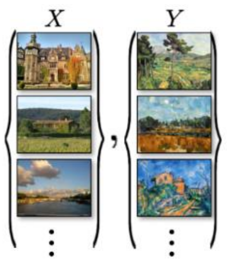
Pix2Pix Training

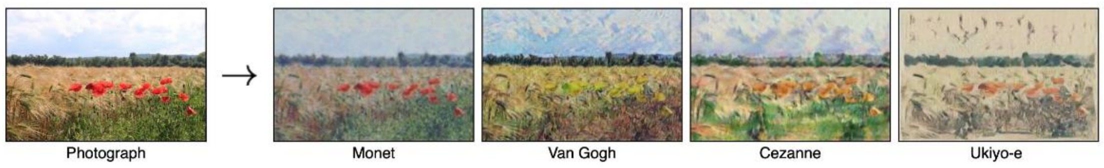
```python
import numpy as np
def train(save_model=False):
    mean_generator_loss = 0
    mean_discriminator_loss = 0
    dataloader = DataLoader(dataset, batch_size=batch_size, shuffle=True)
    cur_step = 0
    for epoch in range(n_epochs):
        for image, _ in tqdm(dataloader):
            image_width = image.shape[3]
            condition = image[:, :, :, :image_width // 2]
            condition = nn.functional.interpolate(condition, size=target_shape)
            real = image[:, :, :, image_width // 2:]
            real = nn.functional.interpolate(real, size=target_shape)
            cur_batch_size = len(condition)
            condition = condition.to(device)
            real = real.to(device)
            disc_opt.zero_grad() # Zero out the gradient before backpropagation
            with torch.no_grad():
                fake = gen(condition)
            disc_fake_hat = disc(fake.detach(), condition) # Detach generator
            disc_fake_loss = adv_criterion(disc_fake_hat, torch.zeros_like(disc_fake_hat))
            disc_real_hat = disc(real, condition)
            disc_real_loss = adv_criterion(disc_real_hat, torch.ones_like(disc_real_hat))
            disc_loss = (disc_fake_loss + disc_real_loss) / 2
            disc_loss.backward(retain_graph=True) # Update gradients
            disc_opt.step() # Update optimizer
            gen_opt.zero_grad()
            gen_loss = get_gen_loss(gen, disc, real, condition, adv_criterion, recon_criterion, lambda_recon)
            gen_loss.backward() # Update gradients
            gen_opt.step() # Update optimizer
            mean_discriminator_loss += disc_loss.item() / display_step
            mean_generator_loss += gen_loss.item() / display_step
            if cur_step % display_step == 0:
                if cur_step > 0:
                    print(f"Epoch {epoch}: Step {cur_step}: Generator (U-Net) loss: {mean_generator_loss}, Discriminator loss: {mean_discriminator_loss}")
                else:
                    print("Pretrained initial state")
                show_tensor_images(condition, size=(input_dim, target_shape, target_shape))
                show_tensor_images(real, size=(real_dim, target_shape, target_shape))
                show_tensor_images(fake, size=(real_dim, target_shape, target_shape))
                mean_generator_loss = 0
                mean_discriminator_loss = 0
                # You can change save_model to True if you'd like to save the model
                if save_model:
                    torch.save({'gen': gen.state_dict(),
                        'gen_opt': gen_opt.state_dict(),
                        'disc': disc.state_dict(),
                        'disc_opt': disc_opt.state_dict()
                    }, f"pix2pix_{cur_step}.pth")
            cur_step += 1
train()
```

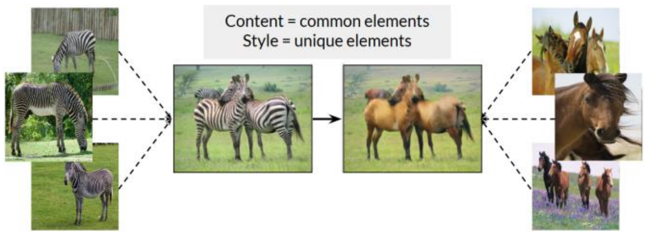
Paired image-to-image translation

-
Edges to photo
- 

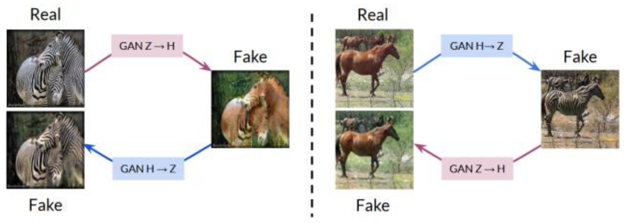
Unpaired image-to-image translation

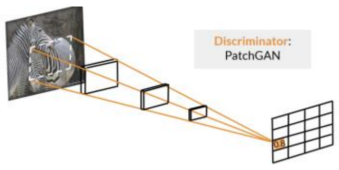
- unsupervised
- Mapping between two piles of image styles
- Finding commanlities and differences
-
Monet to photo
- 

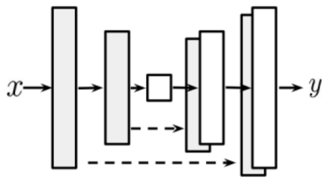
- 

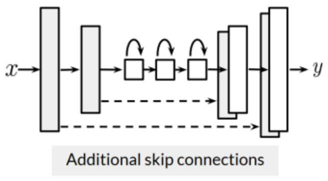
- Mapping between two piles:
  - 

CycleGAN Overview

- Two GANs:
    - 

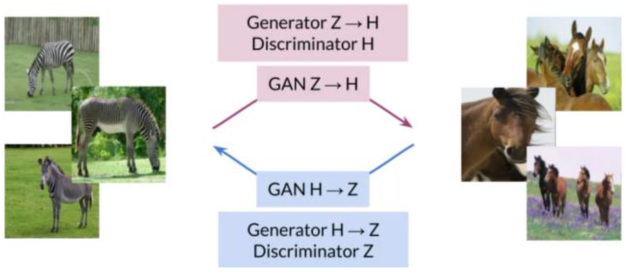
- Discriminator:
    - 

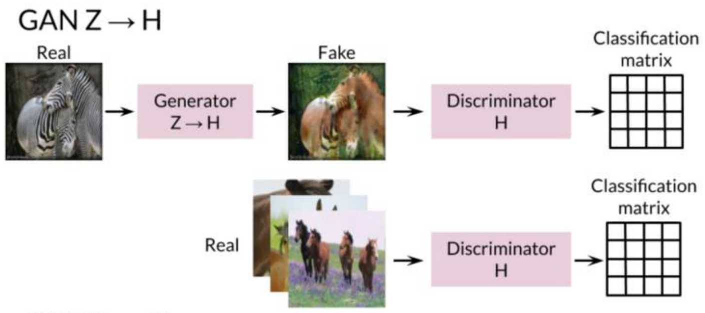
    - PatchGAN
- Generator:
    - 

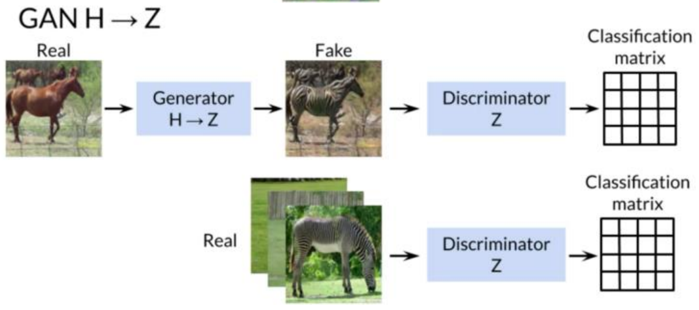
    - Similar to U-Net
- Improved generator:
    - 

    - U-Net + DCGAN generator with resblock

TWO GANs

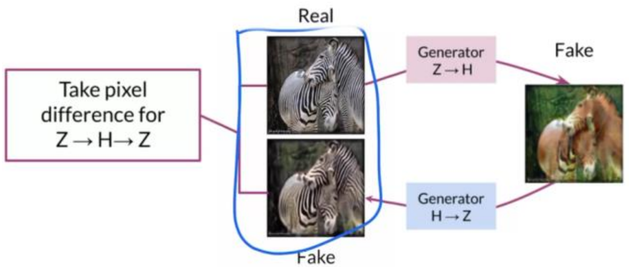
- Components:
  - Two generators + two discriminators
  - 

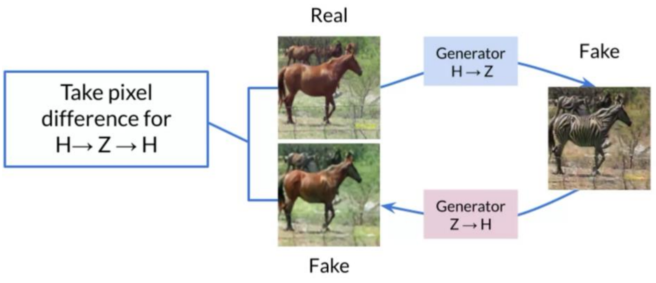
  - 

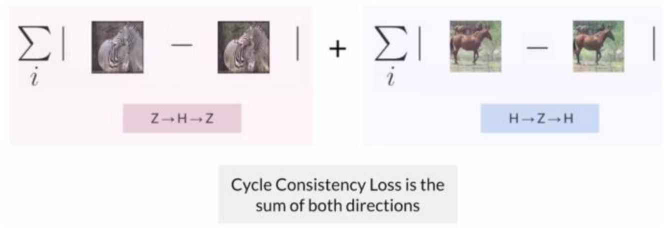
  - 

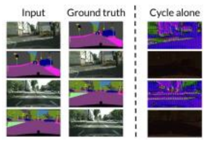
- Inputs:
  - The inputs to the generators and discriminators are similar to Pix2Pix, except:
    - There are no real target outputs
    - Each discriminator is in charge of one pile of images

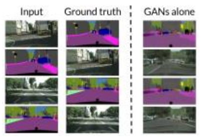
Cycle Consistency

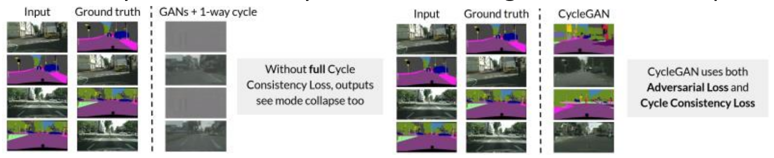
- When styles between 2 piles are transferred, the original content can be recovered.
- Cycle Consistency loss:
  - 

  - 

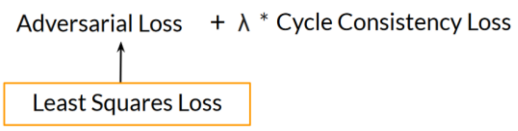
  - 

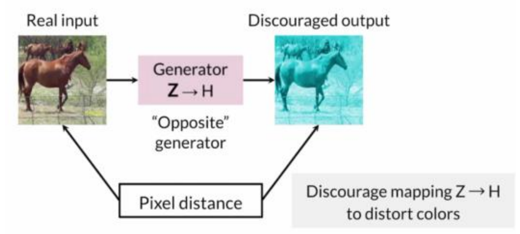
  - There is only one optimizer for both of the generators.
  - There is only one loss term that both of the generators are using.
  - Loss:
    - Adversarial Loss + λ * Cycle Consistency Loss
- Ablation Studies
  - 

  - Removed Adversarial GAN Loss, outputs are not realistic
  - 

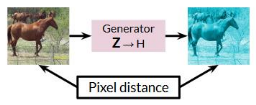
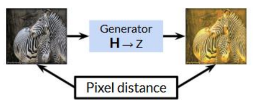
  - Remove Cycle Consistency Loss → show signs of mode collapse
  - 
Input
Ground truth
GANs + 1-way cycle
Without full Cycle Consistency Loss, outputs see mode collapse too
Input
Ground truth
CycleGAN
CycleGAN uses both Adversarial Loss and Cycle Consistency Loss
- Cycle consistency helps transfer uncommon style elements between the two GANs, while maintaining common content
- Add an extra loss term to each generator to softly encourage cycle consistency
- Cycle consistency is used in both directions

Least square Loss (Mean Square Error):

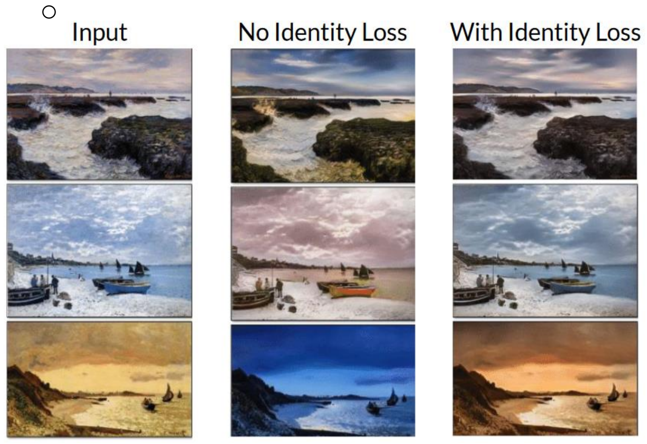
- Used as the adversarial loss function in CycleGAN
- Discriminator:
    - $$\mathbb{E}_{\boldsymbol{x}}\left[(D(\boldsymbol{x})-1)^2\right]+\mathbb{E}_{\boldsymbol{z}}\left[(D(G(\boldsymbol{z}))-0)^2\right]$$
- Generator:
    - $$\mathbb{E}_{\boldsymbol{z}}\left[(D(G(\boldsymbol{z}))-1)^2\right]$$
- Context of Least Squares Loss:
    - 

- Identity Loss:
    - 

- Context of identity Loss:
    - Adversarial Loss + λ * Cycle Consistency Loss

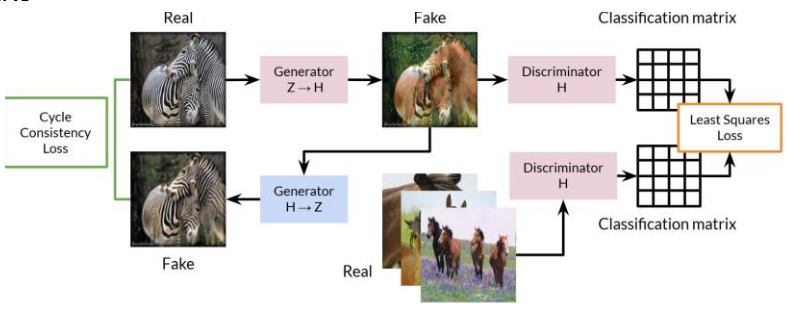
      +  + 

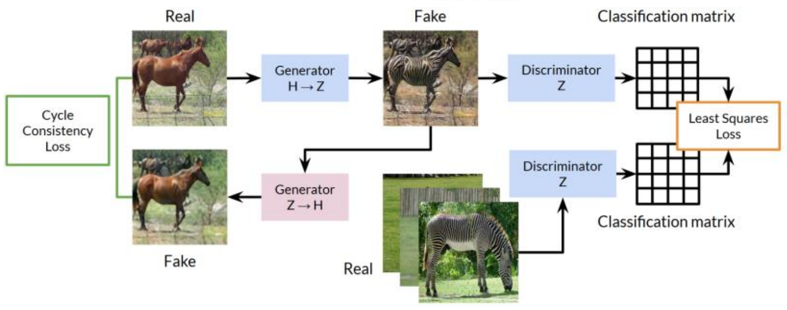
    - Adversarial Loss + λ_1* Cycle Consistency Loss

      + λ_2* Identity Loss
    - 

      Identity Loss helps preserve original photo color
-
- To calcullate the identity loss using a zebra photo, we need to use Generator H → Z since we want to preserve color in images by seeing how the model does when mapping from the input to itself.

Putting CycleGAN together:

- Two GANs
    - 

    - 

- Loss:
    - Generator Loss = Least Squares Adversarial Loss H→Z + Cycle Consistency Loss H→Z→H + Identity Loss H→Z + Least Squares Adversarial Loss Z→H + Cycle Consistency Loss Z→H→Z + Identity Loss Z→H

Applications of CycleGAN

- Social Media Filters: Snapchat-like aging, gender swap, etc.
- Photorealistic Edits: Transforming zebras to horses, altering seasons in a scene.
- Artistic Style Transfer: Converting photos to Monet style paintings and vice versa.
- Scene Object Transformation: Changing specific elements in a scene while maintaining high fidelity, exemplified by a GIF showing a horse turning into a zebra without affecting the background.

Data Augmentation with CycleGAN

- Medical Imaging: Generating paired images for cases hard to capture naturally (e.g., with/without a tumor).
- Removing or adding tumors in images for training classification and segmentation models.
- Monitoring tumor growth.
- CT Scan Segmentation: Producing realistic segmentations for improved training compared to standard augmentation methods.

Variants of CycleGAN for Unpaired Image-to-Image Translation

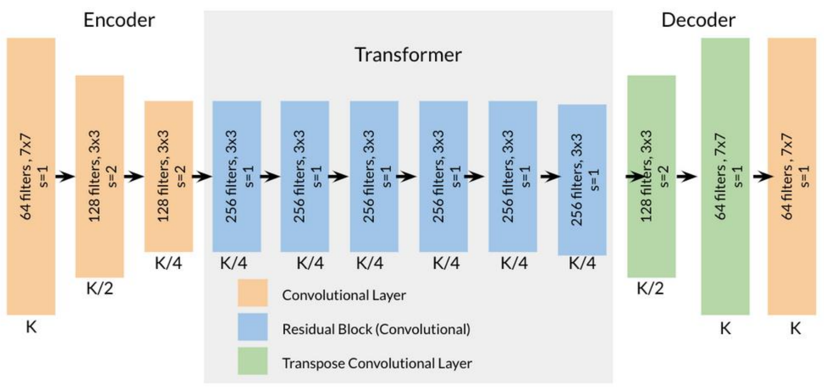
- UNIT (Unsupervised Image-to-Image Translation)
  - Based on the concept of a shared latent space.
  - Capable of translating between two domains (e.g., day and night scenes) using the same noise vector.
  - Allows for bidirectional mapping maintaining the same content across styles.
- MUNIT (Multimodal UNIT)
  - Extends UNIT by enabling translation from one domain to multiple styles in another domain.
  - Discovers various styles within a category (e.g. different shoe styles) without explicit labeling.
  - Combines VAE (Variational Autoencoder) inspiration with GAN (Generative Adversarial Network) components for realistic image generation.

CycleGAN lab:

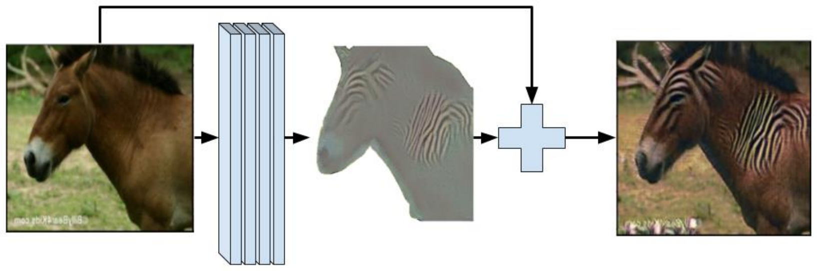
Library setup:

```python
import torch
from torch import nn
from tqdm.auto import tqdm
from torchvision import transforms
from torchvision.utils import make_grid
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
torch.manual_seed(0)

def show_tensor_images(image_tensor, num_images=25, size=(1, 28, 28)):
    image_tensor = (image_tensor + 1) / 2
    image_shifted = image_tensor
    image_unflat = image_shifted.detach().cpu().view(-1, *size)
    image_grid = make_grid(image_unflat[:num_images], nrow=5)
    plt.imshow(image_grid.permute(1, 2, 0).squeeze())
    plt.show()
import glob
import random
import os
from torch.utils.data import Dataset
from PIL import Image
class ImageDataset(Dataset):
    def __init__(self, root, transform=None, mode='train'):
        self.transform = transform
        self.files_A = sorted(glob.glob(os.path.join(root, '%sA' % mode) + '/*.*'))
        self.files_B = sorted(glob.glob(os.path.join(root, '%sB' % mode) + '/*.*'))
        if len(self.files_A) > len(self.files_B):
            self.files_A, self.files_B = self.files_B, self.files_A
        self.new_perm()
        assert len(self.files_A) > 0, "Make sure you downloaded the horse2zebra images!"
    def new_perm(self):
        self.randperm = torch.randperm(len(self.files_B))[:len(self.files_A)]
    def __getitem__(self, index):
        item_A = self.transform(Image.open(self.files_A[index % len(self.files_A)]))
        item_B = self.transform(Image.open(self.files_B[self.randperm[index]]))
        if item_A.shape[0] != 3:
            item_A = item_A.repeat(3, 1, 1)
        if item_B.shape[0] != 3:
            item_B = item_B.repeat(3, 1, 1)
        if index == len(self) - 1:
            self.new_perm()
        return (item_A - 0.5) * 2, (item_B - 0.5) * 2
    def __len__(self):
        return min(len(self.files_A), len(self.files_B))
```

Generator:



Residual Block:



```
class ResidualBlock(nn.Module):
    def __init__(self, input_channels):
        super(ResidualBlock, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, input_channels, kernel_size=3, padding=1, padding_mode='reflect')
        self.conv2 = nn.Conv2d(input_channels, input_channels, kernel_size=3, padding=1, padding_mode='reflect')
        self.instancenorm = nn.InstanceNorm2d(input_channels)
        self.activation = nn.ReLU()
    def forward(self, x):
        original_x = x.clone()
        x = self.conv1(x)
        x = self.instancenorm(x)
        x = self.activation(x)
        x = self.conv2(x)
        x = self.instancenorm(x)
        return original_x + x
```

Contracting and Expanding blocks

```
class ContractingBlock(nn.Module):
    def __init__(self, input_channels, use_bn=True, kernel_size=3, activation='relu'):
        super(ContractingBlock, self).__init__()
        self.conv1 = nn.Conv2d(input_channels, input_channels * 2, kernel_size=kernel_size, padding=1, stride=2, 
padding_mode='reflect')
        self.activation = nn.ReLU() if activation == 'relu' else nn.LeakyReLU(0.2)
        if use_bn:
            self.instancenorm = nn.InstanceNorm2d(input_channels * 2)
        self.use_bn = use_bn
    def forward(self, x):
        x = self.conv1(x)
        if self.use_bn:
            x = self.instancenorm(x)
        x = self.activation(x)
        return x
```

```python
class ExpandingBlock(nn.Module):
    def __init__(self, input_channels, use_bn=True):
        super(ExpandingBlock, self).__init__()
        self.conv1 = nn.ConvTranspose2d(input_channels, input_channels // 2, kernel_size=3, stride=2, padding=1, output_padding=1)
        if use_bn:
            self.instancenorm = nn.InstanceNorm2d(input_channels // 2)
        self.use_bn = use_bn
        self.activation = nn.ReLU()
    def forward(self, x):
        x = self.conv1(x)
        if self.use_bn:
            x = self.instancenorm(x)
        x = self.activation(x)
        return x
class FeatureMapBlock(nn.Module):
    def __init__(self, input_channels, output_channels):
        super(FeatureMapBlock, self).__init__()
        self.conv = nn.Conv2d(input_channels, output_channels, kernel_size=7, padding=3, padding_mode='reflect')
    def forward(self, x):
        x = self.conv(x)
        return x
```

Generator:

```python
class Generator(nn.Module):
    def __init__(self, input_channels, output_channels, hidden_channels=64):
        super(Generator, self).__init__()
        self.upfeature = FeatureMapBlock(input_channels, hidden_channels)
        self.contract1 = ContractingBlock(hidden_channels)
        self.contract2 = ContractingBlock(hidden_channels * 2)
        res_mult = 4
        self.res0 = ResidualBlock(hidden_channels * res_mult)
        self.res1 = ResidualBlock(hidden_channels * res_mult)
        self.res2 = ResidualBlock(hidden_channels * res_mult)
        self.res3 = ResidualBlock(hidden_channels * res_mult)
        self.res4 = ResidualBlock(hidden_channels * res_mult)
        self.res5 = ResidualBlock(hidden_channels * res_mult)
        self.res6 = ResidualBlock(hidden_channels * res_mult)
        self.res7 = ResidualBlock(hidden_channels * res_mult)
        self.res8 = ResidualBlock(hidden_channels * res_mult)
        self.expand2 = ExpandingBlock(hidden_channels * 4)
        self.expand3 = ExpandingBlock(hidden_channels * 2)
        self.downfeature = FeatureMapBlock(hidden_channels, output_channels)
        self.tanh = torch.nn.Tanh()
    def forward(self, x):
        x0 = self.upfeature(x)
        x1 = self.contract1(x0)
        x2 = self.contract2(x1)
        x3 = self.res0(x2)
        x4 = self.res1(x3)
        x5 = self.res2(x4)
        x6 = self.res3(x5)
        x7 = self.res4(x6)
        x8 = self.res5(x7)
        x9 = self.res6(x8)
        x10 = self.res7(x9)
        x11 = self.res8(x10)
        x12 = self.expand2(x11)
        x13 = self.expand3(x12)
        xn = self.downfeature(x13)
        return self.tanh(xn)
```

PatchGAN Discriminator

```python
class Discriminator(nn.Module):
    def __init__(self, input_channels, hidden_channels=64):
        super(Discriminator, self).__init__()
        self.upfeature = FeatureMapBlock(input_channels, hidden_channels)
        self.contract1 = ContractingBlock(hidden_channels, use_bn=False, kernel_size=4, activation='lrelu')
        self.contract2 = ContractingBlock(hidden_channels * 2, kernel_size=4, activation='lrelu')
        self.contract3 = ContractingBlock(hidden_channels * 4, kernel_size=4, activation='lrelu')
        self.final = nn.Conv2d(hidden_channels * 8, 1, kernel_size=1)

    def forward(self, x):
        x0 = self.upfeature(x)
        x1 = self.contract1(x0)
        x2 = self.contract2(x1)
        x3 = self.contract3(x2)
        xn = self.final(x3)
        return xn
```

Training:

```python
import torch.nn.functional as F
adv_criterion = nn.MSELoss()
recon_criterion = nn.L1Loss()
n_epochs = 20
dim_A = 3
dim_B = 3
display_step = 200
batch_size = 1
lr = 0.0002
load_shape = 286
target_shape = 256
device = 'cuda'
```

Load dataset:

```python
transform = transforms.Compose([
    transforms.Resize(load_shape),
    transforms.RandomCrop(target_shape),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
])
import torchvision
dataset = ImageDataset("horse2zebra", transform=transform)
```

Initalize generators and discriminators:

```python
gen_AB = Generator(dim_A, dim_B).to(device)
gen_BA = Generator(dim_B, dim_A).to(device)
gen_opt = torch.optim.Adam(list(gen_AB.parameters()) + list(gen_BA.parameters()), lr=lr, betas=(0.5, 0.999))
disc_A = Discriminator(dim_A).to(device)
disc_A_opt = torch.optim.Adam(disc_A.parameters(), lr=lr, betas=(0.5, 0.999))
disc_B = Discriminator(dim_B).to(device)
disc_B_opt = torch.optim.Adam(disc_B.parameters(), lr=lr, betas=(0.5, 0.999))
def weights_init(m):
    if isinstance(m, nn.Conv2d) or isinstance(m, nn.ConvTranspose2d):
        torch.nn.init.normal_(m.weight, 0.0, 0.02)
    if isinstance(m, nn.BatchNorm2d):
        torch.nn.init.normal_(m.weight, 0.0, 0.02)
        torch.nn.init.constant_(m.bias, 0)
pretrained = True
if pretrained:
    pre_dict = torch.load('cycleGAN_100000.pth')
    gen_AB.load_state_dict(pre_dict['gen_AB'])
    gen_BA.load_state_dict(pre_dict['gen_BA'])
    gen_opt.load_state_dict(pre_dict['gen_opt'])
    disc_A.load_state_dict(pre_dict['disc_A'])
    disc_A_opt.load_state_dict(pre_dict['disc_A_opt'])
    disc_B.load_state_dict(pre_dict['disc_B'])
    disc_B_opt.load_state_dict(pre_dict['disc_B_opt'])
else:
    gen_AB = gen_AB.apply(weights_init)
    gen_BA = gen_BA.apply(weights_init)
    disc_A = disc_A.apply(weights_init)
    disc_B = disc_B.apply(weights_init)
```

Discriminator Loss:

```python
def get_disc_loss(real_X, fake_X, disc_X, adv_criterion):
    disc_fake = disc_X(fake_X)
    disc_fake_loss = adv_criterion(disc_fake,torch.zeros_like(disc_fake))
    disc_real = disc_X(real_X)
    disc_real_loss = adv_criterion(disc_real, torch.ones_like(disc_real))
    disc_loss = (disc_fake_loss + disc_real_loss)/2
    return disc_loss
```

Generator Loss:

Adversarial Loss (MSE):

```python
def get_gen_adversarial_loss(real_X, disc_Y, gen_XY, adv_criterion):
    fake_Y = gen_XY(real_X)
    fake_disc_Y = disc_Y(fake_Y)
    adversarial_loss = adv_criterion(fake_disc_Y,torch.ones_like(fake_disc_Y))
    return adversarial_loss, fake_Y
```

Identity Loss:

```python
def get_identity_loss(real_X, gen_YX, identity_criterion):
    identity_X = gen_YX(real_X)
    identity_loss = identity_criterion(real_X,identity_X)
    return identity_loss, identity_X
```

Cycle Consistency Loss:

```python
def get_cycle_consistency_loss(real_X, fake_Y, gen_YX, cycle_criterion):
    cycle_X = gen_YX(fake_Y)
    cycle_loss = cycle_criterion(real_X, cycle_X)
    return cycle_loss, cycle_X
```

Generator Loss (Total)

```python
def get_gen_loss(real_A, real_B, gen_AB, gen_BA, disc_A, disc_B, adv_criterion, identity_criterion, cycle_criterion, lambda_identity=0.1, lambda_cycle=10):
    adversarial_loss_AB, fake_B = get_gen_adversarial_loss(real_A,disc_B,gen_AB,adv_criterion)
    adversarial_loss_BA, fake_A = get_gen_adversarial_loss(real_B,disc_A,gen_BA,adv_criterion)
    adversarial_loss = adversarial_loss_AB + adversarial_loss_BA
    identity_loss_A, _ = get_identity_loss(real_A, gen_BA, identity_criterion)
    identity_loss_B, _ = get_identity_loss(real_B, gen_AB, identity_criterion)
    identity_loss = identity_loss_A + identity_loss_B
    cycle_consistency_loss_BA, _ = get_cycle_consistency_loss(real_A, fake_B, gen_BA, cycle_criterion)
    cycle_consistency_loss_AB, _ = get_cycle_consistency_loss(real_B, fake_A, gen_AB, cycle_criterion)
    cycle_consistency_loss = cycle_consistency_loss_BA + cycle_consistency_loss_AB
    gen_loss = adversarial_loss + lambda_identity * identity_loss + lambda_cycle * cycle_consistency_loss
    return gen_loss, fake_A, fake_B
```