Generative Adversarial Networks and Image2Image Translation

University of Salzburg Deep Learning Online Course Series

Christof Kauba and Georg Wimmer

Universität Salzburg
Department of Computer Sciences
A-5020 Salzburg, Austria

16th October 2023

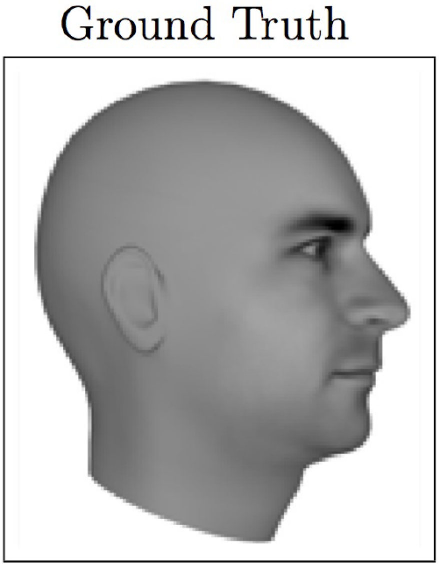
Table of Contents

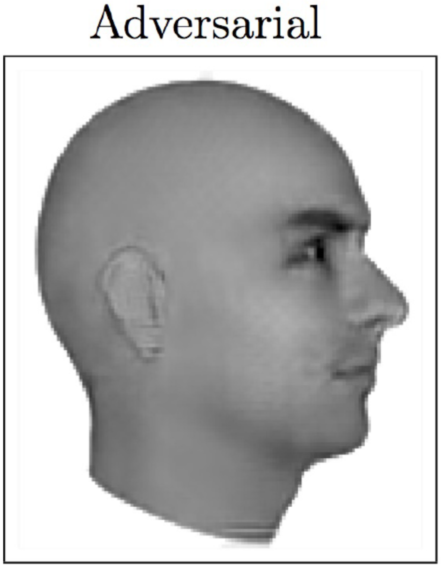
Outline

What is a GAN?

- A generative adversarial network (or short: GAN) is an approach to generative modelling using deep learning methods (more on generative modelling shortly)
  - Generative: Learning a generative model
  - Adversarial: Trained in an adversarial setting (game theoretic approach)
  - Network: Based on Deep Neural Networks
- GANs were first introduced by Goodfellow et al. in 2014: Generative Adversarial Nets
- Have the ability to generate realistic examples across a range of problem domains:
  - image to image translation tasks, e.g. translating daylight to night pictures
  - generating photorealistic photos of objects, scenes and people
  - image super-resolution
  - overcoming limited data with the help of GAN generated data
- GANs revolutionized generative modelling by producing crisp, high-resolution images

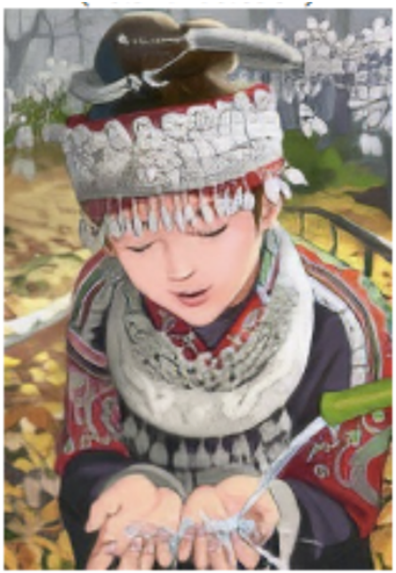
GAN Examples (1)

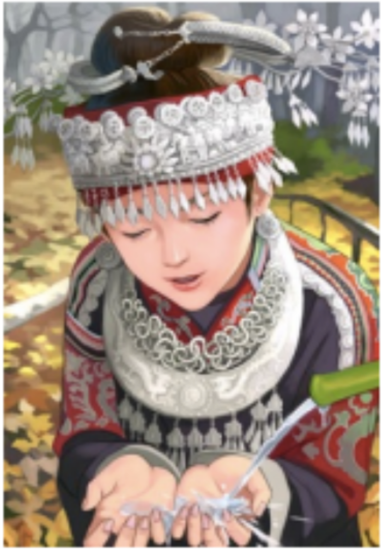




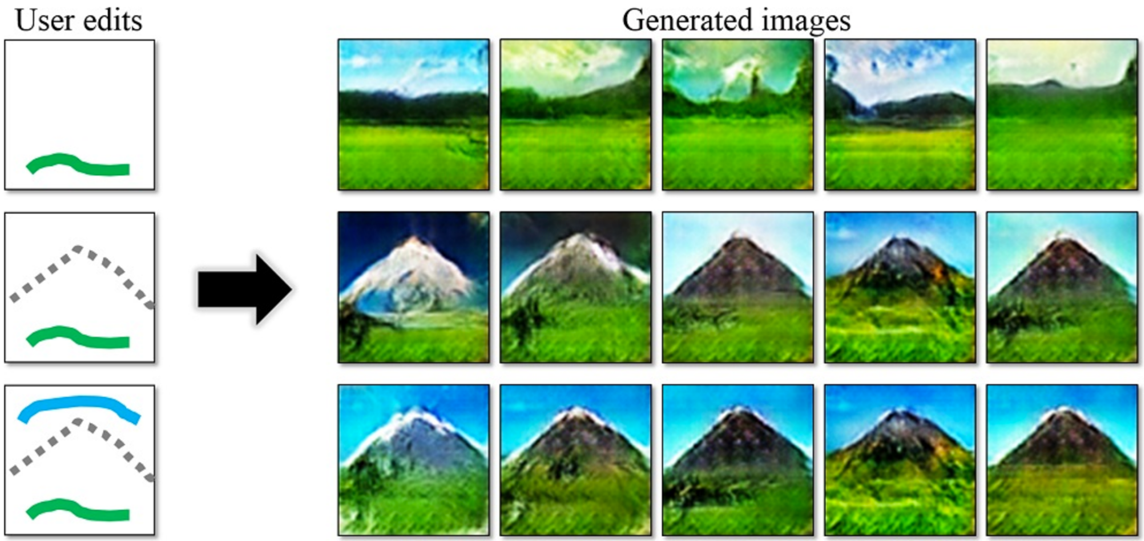
GAN Examples (2)

- Which of the following images is computer generated?





GAN Examples (3)



Literature - Papers I

- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y. **Generative adversarial nets**, NIPS (2014).
- Goodfellow, Ian **NIPS 2016 Tutorial: Generative Adversarial Networks**, NIPS (2016).
- Radford, A., Metz, L. and Chintala, S., **Unsupervised representation learning with deep convolutional generative adversarial networks**. arXiv preprint arXiv:1511.06434. (2015)
- Salimans, T., Goodfellow, I., Zaremba, W., Cheung, V., Radford, A., & Chen, X. Improved techniques for training gans. NIPS (2016).
- Chen, X., Duan, Y., Houthooft, R., Schulman, J., Sutskever, I., & Abbeel, P. InfoGAN: Interpretable Representation Learning by Information Maximization Generative Adversarial Nets, NIPS (2016).
- Zhao, Junbo, Michael Mathieu, and Yann LeCun. Energy-based generative adversarial network. arXiv preprint arXiv:1609.03126 (2016).
- Mirza, Mehdi, and Simon Osindero. Conditional generative adversarial nets. arXiv preprint arXiv:1411.1784 (2014).

Literature - Papers II

- Isola, P., Zhu, J. Y., Zhou, T., & Efros, A. A. **Image-to-image translation with conditional adversarial networks.** arXiv preprint arXiv:1611.07004. (2016).
- Reed, S., Akata, Z., Yan, X., Logeswaran, L., Schiele, B., & Lee, H. **Generative adversarial text to image synthesis**. JMLR (2016).
- Antipov, G., Baccouche, M., & Dugelay, J. L. **Face Aging With Conditional Generative Adversarial Networks**. arXiv preprint arXiv:1702.01983. (2017).
- Liu, Ming-Yu, and Oncel Tuzel. Coupled generative adversarial networks. NIPS (2016).
- Denton, E.L., Chintala, S. and Fergus, R., 2015. Deep Generative Image Models using a Laplacian Pyramid of Adversarial Networks. NIPS (2015).
- Dumoulin, V., Belghazi, I., Poole, B., Lamb, A., Arjovsky, M., Mastropietro, O., & Courville, A. Adversarially learned inference. arXiv preprint arXiv:1606.00704 (2016).

Literature - Books

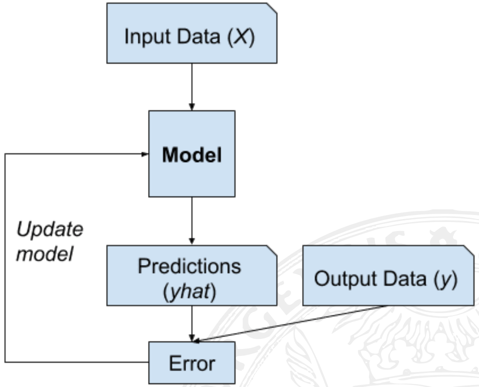
- Goodfellow, Ian, Yoshua Bengio, and Aaron Courville. Deep learning. MIT press, 2016.
- Brownlee, Jason. Generative adversarial networks with python: deep learning generative models for image synthesis and image translation. Machine Learning Mastery, 2019.
- Foster, David. Generative deep learning: teaching machines to paint, write, compose, and play. O'Reilly Media, 2019.
- Géron, Aurélien. Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems. O'Reilly Media, Inc., 2019.
- Ketkar, Nikhil, and Eder Santana. Deep learning with Python. Vol. 1. Berkeley, CA: Apress, 2017.
- Buduma, Nikhil, and Nicholas Locascio. Fundamentals of deep learning: Designing next-generation machine intelligence algorithms. O'Reilly Media, Inc., 2017.
- Osinga, Douwe. Deep learning cookbook: practical recipes to get started quickly. O'Reilly Media, Inc., 2018.
- Michelucci, Umberto. Applied Deep Learning. Apress, 2018.

Literature - Online

- A gentle introduction to GANs: `https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/`
- Online Version of the Book "Deep Learning" by Ian Goodfellow and Yoshua Bengio and Aaron Courville: `https://www.deeplearningbook.org/`
- Various slides by Ian Goodfellow: `https://www.iangoodfellow.com/slides/`
- Google Online Course - GAN Introduction: `https://developers.google.com/machine-learning/gan`
- A list of all named GANs: `https://github.com/hindupuravinash/the-gan-zoo`
- Tips and tricks to make GANs work: `https://github.com/soumith/ganhacks`

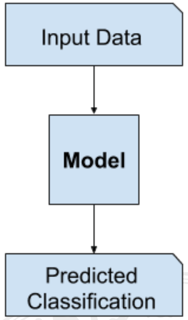
From Predictive to Generative Modelling (1)

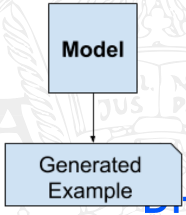
- Predictive modelling is a typical machine learning task involving using a model to make a prediction
- Training the model requires a training dataset, comprised of multiple samples (training samples)
  - Each sample has input variables (X) and output class labels (y)
  - Model is trained by showing examples of inputs, having it predict the output labels, comparing the predicted to the real labels and correcting the model to make the outputs more correct
- This is typical supervised learning task



From Predictive to Generative Modelling (2)

- In the previous CNN approaches we mainly covered discriminative/predictive models:
  - E.g. given an image X, we can predict a label y
  - Hence, it estimates $P(y|X)$
- Discriminative modelling has several key limitations
  - We cannot model $P(X)$, i.e. the probability of seeing a certain image
  - Hence, we cannot sample from $P(X)$, i.e. we cannot generate new images
- Generative models are able to cope with the above mentioned problems
  - Can model $P(X)$
  - Thus, generate new images which are drawn according to $P(X)$





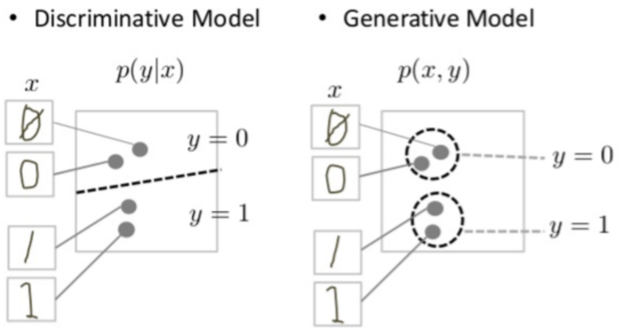
From Predictive to Generative Modelling (3)

- The second paradigm to learn a model, used in generative modelling, is to give the model only the input variables (X) with the problem not having any output variables (y)
- The model is constructed by extracting or summarising the patterns in the input data
- Generative models tackle a more difficult task than analogous discriminative models. Generative models have to model more
- There is no correction of the model as it is not predicting anything
- This lack of correction is in general referred to as **unsupervised learning**

Generative Modelling (1)

- In generative modelling our aim is to train a network that models a distribution, e.g. a distribution over images, and is then able to generate new images that are drawn from this distribution
  - Unsupervised learning task in machine learning that involves automatically discovering and learning regularities or patterns in given input data
  - Such that the model can be used to generate new examples that could have been drawn from the original data set
- E.g. a generative model for images might capture correlations like "things that look like boats are probably going to appear near things that look like water". These are very complicated distributions.
- In contrast, a discriminative model might learn the difference between "sailboat" or "not sailboat" by just looking for a few tell-tale patterns. It could ignore many of the correlations that the generative model must get right.
- If we look at the "data space":
  - Discriminative models try to draw boundaries in the data space
  - Generative models try to model how data is placed throughout the space

Generative Modelling (2)



- A straight forward way to judge the quality of the trained model is to sample from it
- There are 4 modern approaches to generative modelling:
  - **Generative Adversarial Networks**
  - Reversible architectures
  - Autoregressive models
  - Variational autoencoders
- All those approaches have different advantages and disadvantages
- We will focus on GANs in the following

Outline

GAN Overview

- GANs are a clever way of training a generative model by framing the problem as a supervised learning problem
- Based on two sub-models:
  - Generator model: trained to generate new examples
  - Discriminator model: tries to classify examples as either real (from the original data) or fake (generated)
- The two models are trained in a zero-sum game, adversarial, until the discriminator model is fooled about half of the time (game theoretic scenario)
- Today, most GANs are based on the DCGAN (Deep Convolutional Generative Adversarial Network) architecture

Deriving the GAN Architecture

- Based on the idea of generative modelling, we will briefly go through the original ideas and derive the basic GAN architecture
- The basic task we want to do is generate samples belonging to a particular data distribution $P_{data}(x)$
- Hence, use a distribution $P_G(x;\theta)$, parametrised by θ to approximate it:
  - Can e.g. $P_G(x;\theta)$ can be a Gaussian Mixture Model, with the parameter θ containing the means and variances of the Gaussian distributions
  - Our goal is to find/optimise θ such that $P_G(x;\theta)$ is as close as possible to $P_{data}(x)$ (resembles the original distribution)
- If we have the original data, we can just sample $\{x_1, x_2, ..., x_m\}$ from $P_{data}(x)$
- The likelihood of generating these x_is under P_G is: $L = \prod\limits_{i=1...m} P_G(x_i;\theta)$
- We can then find the optimal $\hat{\theta}$ by maximising L
- This is called the **Maximum Likelihood Estimation**

KL (Kullback-Leibler) Divergence

- Distance function often used in computer sciences
- Discrete: $D_{KL}(P|Q) = \sum_i P(i) log(\frac{P(i)}{Q(i)})$
- Continuous: $D_{KL}(P|Q) = \int_{-\infty}^{\infty} p(x) log(\frac{p(x)}{q(x)})$
- With:
  - Entropy: $-\sum_i P(i) log(P(i))$ - expected code length
  - Cross Entropy: $-\sum_i P(i) log(Q(i))$ - expected coding length using the optimal code for Q
  - $D_{KL} = \sum_i P(i) log(\frac{P(i)}{Q(i)}) = \sum_i P(i)[log(P(i)) - log(Q(i))]$ - extra bits needed
  - Jensen-Shannon Divergence (JSD): $JSD(P|Q) = \frac{1}{2} D_{KL}(P|M) + \frac{1}{2} D_{KL}(Q|M)$, $M = \frac{1}{2}(P + Q)$ - symmetric Kullback-Leibler divergence

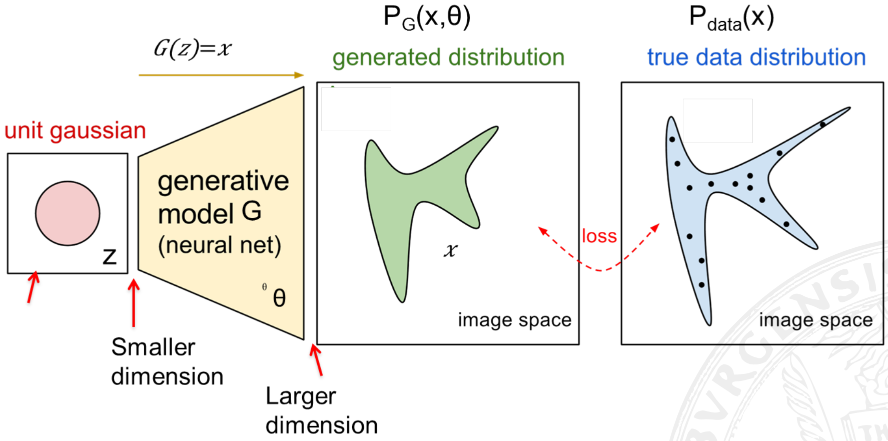
Maximum Likelihood Estimate

$$\hat{\theta} = \underset{\theta}{argmax}\{\prod_{i=1\dots m} P_G(x_i;\theta)\} \rightarrow \underset{\theta}{argmax}\{log(\prod_{i=1\dots m} P_G(x_i;\theta))\} =$$
$$\underset{\theta}{argmax}\{\textstyle\sum_{i=1\dots m} log(P_G(x_i;\theta))\},\ \ \{x_1,\dots,x_m\} \text{ sampled from } P_{data}(x)$$
$$= \underset{\theta}{argmax}\{\textstyle\sum_{i=1\dots m} P_{data}(x_i) log(P_G(x_i;\theta))\}\text{- this is cross entropy}$$
$$\approx \underset{\theta}{argmax}\{\textstyle\sum_{i=1\dots m} P_{data}(x_i) log(P_G(x_i;\theta)) - \textstyle\sum_{i=1\dots m} P_{data}(x_i) log(P_{data}(x_i))\}$$
$$= \underset{\theta}{argmin}\{KL(P_{data}(x)|P_G(x_i;\theta)) \text{ this is now the KL divergence}$$

- Remember that we started with P_G being a Gaussian mixture model, hence finding the best θ is still finding the best fitting Gaussians, which is limited in the type of data distributions that can be generated.
- The above derived ML approach does not work well
- ⇒Introduce the **GAN** that will change P_G completely instead of just estimating the parameters of a fixed type of P_G
- In essence, we want to find the best P_G which is more complex and structured than Gaussian distributions in order to approximate P_{data}

Deriving the GAN Architecture

- Let's use a neuronal network to resemble $P_G(x;\theta)$:



- $P_G(x) = \int_z P_{prior}(z) I_{[G(z)=x]} dz$
- Question is now: how do we compute this likelihood?

Basic Idea of a GAN (1)

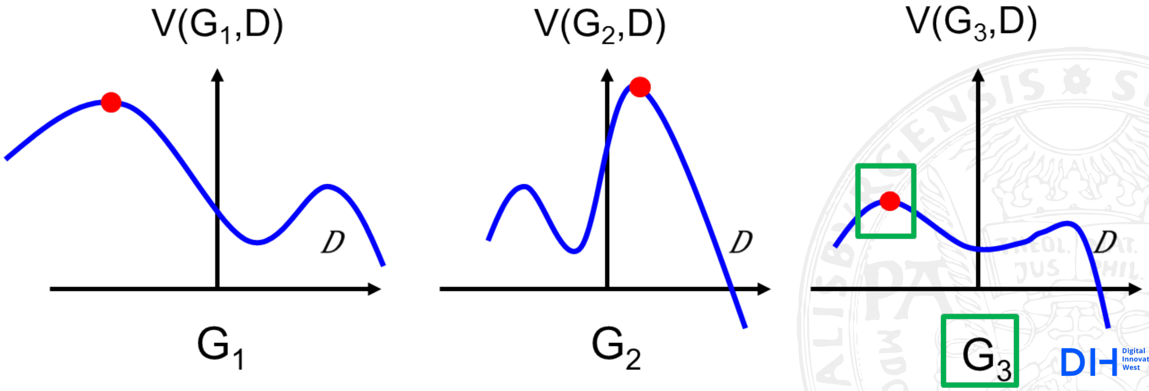
- Generator **G**
  - G is a function with an input z and an output x
  - Given a prior distribution $P_{prior}(z)$, G defines a probability distribution $P_G(x)$
- Discriminator **D**
  - D is also a function with an input x and a scalar value as output
  - It evaluates the difference between $P_G(x)$ and $P_{data}(x)$
- In order for D to be able to quantify the difference between $P_{data}(x)$ and $P_G(x)$ we need to define a cost function $V(G, D)$:

$$\hat{G} = \underset{G}{argmin}\{\underset{D}{max}\{V(G, D)\}\}$$

- Remember: Here we are changing the whole distribution G instead of just updating/changing its parameters (**difference to maximum likelihood estimate**)

Basic Idea of a GAN (2)

- $\hat{G} = \underset{G}{argmin}\{\underset{D}{max}\{V(G,D)\}\}$
- We pick a JSD (Jenson-Shannon Divergence) function:
  $V = E_{x\sim P_{data}}[log(D(x)] + E_{x\sim P_G}[log(1 - D(x))]$
- Given a generator G, $\underset{D}{max}\{V(G,D)\}$ evaluates the difference between the two distributions P_{data} and P_G
- So we pick the generator G such that P_G is most similar to P_{data}



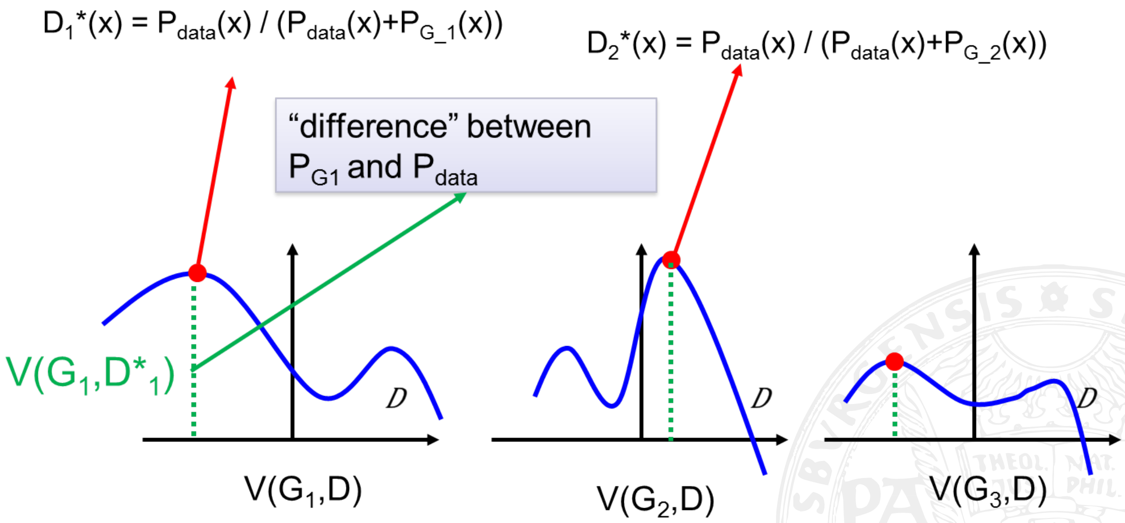
Basic Idea of a GAN (3)

- $\max_D\{V(G,D)\}$, $\dot{G} = \underset{G}{argmin}\{\max_D\{V(G,D)\}\}$
- Given G, we now want to find the optimal $\hat{D}$ maximising:

$$\begin{aligned} V &= E_{x\sim P_{data}}[log(D(x)] + E_{x\sim P_G}[log(1-D(x))] \\ &= \sum[P_{data}(x)log(D(x)) + P_G(x)log(1-D(x))] \end{aligned}$$

- Thus: $\hat{D}(x) = \frac{P_{data}(x)}{P_{data}(x)+P_G(x)}$
- Explanation:
  - Assuming that $D(x)$ can have any value and with a given x, the optimal $\hat{D}$ is maximising:
  - $f(D) = a \cdot log(D) + b \cdot log(1-D) \rightarrow \hat{D} = \frac{a}{a+b}$

Basic Idea of a GAN (4)



$D_1^*(x) = P_{data}(x) / (P_{data}(x)+P_{G\_1}(x))$
$D_2^*(x) = P_{data}(x) / (P_{data}(x)+P_{G\_2}(x))$
"difference" between P_{G1} and P_{data}
$V(G_1,D^*_1)$
D
D
D
$V(G_1,D)$
$V(G_2,D)$
$V(G_3,D)$

Basic Idea of a GAN (5)

- $\max_D\{V(G,D)\}$, $V = E_{x\sim P_{data}}[log(D(x)] + E_{z\sim P_G}[log(1-D(z))]$
- $\max_D\{V(G,D)\} = V(G,\hat{D})$, where $\hat{D}(x) = \frac{P_{data}}{P_{data}+P_G}$ and $1-\hat{D}(x) = \frac{P_G}{P_{data}+P_G}$
- using the definition of V:

$$\begin{aligned}\max_D\{V(G,D)\} &= E_{x\sim P_{data}}[log(\hat{D}(x)] + E_{x\sim P_G}[log(1-\hat{D}(x))] \\ &\approx \sum[P_{data}(x)log(\hat{D}(x)) + P_G(x)log(1-\hat{D}(x))]] \\ &= -2\cdot log(2) + 2\cdot JSD(P_{data}(x)|P_G(x))\end{aligned}$$

- Remember that: $JSD(P|Q) = \frac{1}{2}D_{KL}(P|M) + \frac{1}{2}D_{KL}(Q|M)$ where $M = \frac{1}{2}(P+Q)$ and $D_{KL}(P|Q) = \sum P(x)log(\frac{P(x)}{Q(x)})$

Summary of the Basic GAN Idea

- Generator G and a Discriminator D
- Goal: Looking for $\hat{G}$ such that:

$$\hat{G} = \underset{G}{argmin}\{\underset{D}{max}\{V(G,D)\}\}$$

- with $V = E_{x\sim P_{data}}[log(D(x)] + E_{z\sim P_G}[log(1 - D(z))]$
- Given the Generator G: $\underset{D}{max} V(G,D) = -2 \cdot log(2) + 2 \cdot JSD(P_{data}(x)|P_G(z))$
- What is now the optimal G?
  - G such that it minimises the JSD $(= 0)$:
  - In that case: $P_G(z) = P_{data}(x)$

How to find the Optimum in Practice (1)

- $V = E_{x \sim P_{data}}[log(D(x)] + E_{x \sim P_G}[log(1 - D(x))]$
- Assume we have G given, how do we compute $\underset{D}{max}\{V(G, D)\}$ (**note**: this corresponds to the loss function of the network)?
  - Sample $\{x_1, ..., x_m\}$ from $P_{data}(x)$
  - Sample $\{x_1^*, ..., x_m^*\}$ from the generator $P_G(x)$
  - Maximise:

$$V' = \frac{1}{m} \sum_{i=1...m} log(D(x_i)) + \frac{1}{m} \sum_{i=1...m} log(1 - D(x_i^*))$$

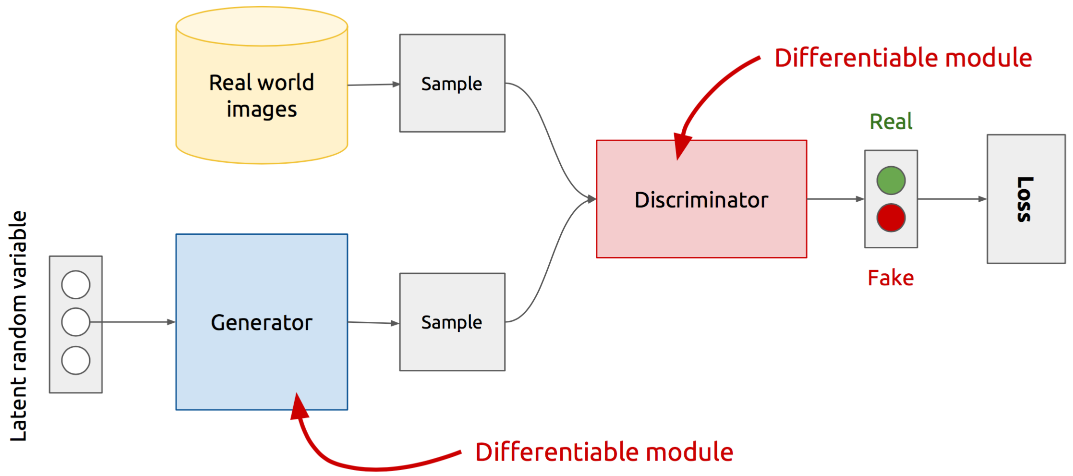
  - where x_i is a positive example D must accept and x_i^* is a negative example D must reject
- This is exactly what a binary classifier does:

How to find the Optimum in Practice (2)

- Binary Classifier:
  - Output is $D(x)$, minimise the cross-entropy
  - If x is a positive example $\rightarrow$ minimise $-log(D(x))$
  - If x is a negative example $\rightarrow$ minimise $-log(1 - D(x))$

- Hence the discriminator D is a binary classifier with its parameters θ_d
  - $\{x_1, ..., x_m\}$ from $P_{data}(x) \rightarrow$ positive examples
  - $\{x_1^*, ..., x_m^*\}$ from $P_G(x) \rightarrow$ negative examples
- Minimise $L = -V'$ or maximise $V' = \frac{1}{m}\sum_{i=1...m} log(D(x_i)) + \frac{1}{m}\sum_{i=1...m} log(1 - D(x_i^*))$
- This is done using gradient ascent/descent (see GAN training algorithm)

GAN Architecture Overview

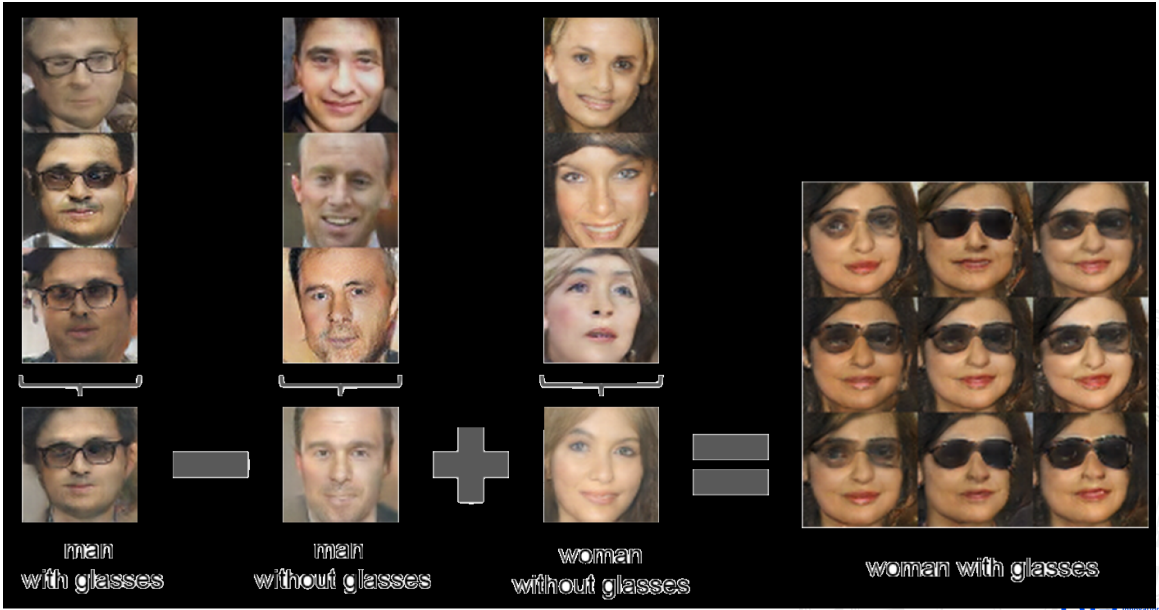


- Z is some random noise (Gaussian or Uniform)
- Z can be thought as the latent representation of the image

Generator Model

- Takes a fixed-length random vector z as input
  - Vector is drawn from a Gaussian random distribution (or Uniform)
  - Used to seed the generative process
- Generates a sample x in the desired output domain
- After generator training, points in the multi-dimensional vector space will correspond to points in the problem domain
- Vector space is often referred to as latent space.
  - Latent variables are important for the domain but are not directly observable
  - Latent space is a projection or compression of a data distribution, i.e. it provides high-level concepts of the observed raw data
- After the training, the generator is kept and used to generate new samples

Latent Space Captures Interesting Patterns



Generator Network - Some More Formal Requirements

- Must be differentiable (to be able to use SGD for training)
  - In theory, could use REINFORCE for discrete variables
- No invertibility requirement
- Trainable for any size of z
- Some guarantees require z to have higher dimension than x
- Can make x conditionally Gaussian given z but need not do so

Discriminator Model

- Takes an example from the domain as input (real or generated) and predicts a binary class label - real or fake (generated)
  - Real samples are from the training dataset
  - Fake samples are output by the generator model
- The discriminator is a usual classification model (predictive modelling)
- After the training process, the discriminator is discarded

- For training (see next slide) use SGD-like algorithm of choice (usually Adam) on two mini-batches simultaneously:
  - A mini-batch of training examples
  - A mini-batch of generated samples

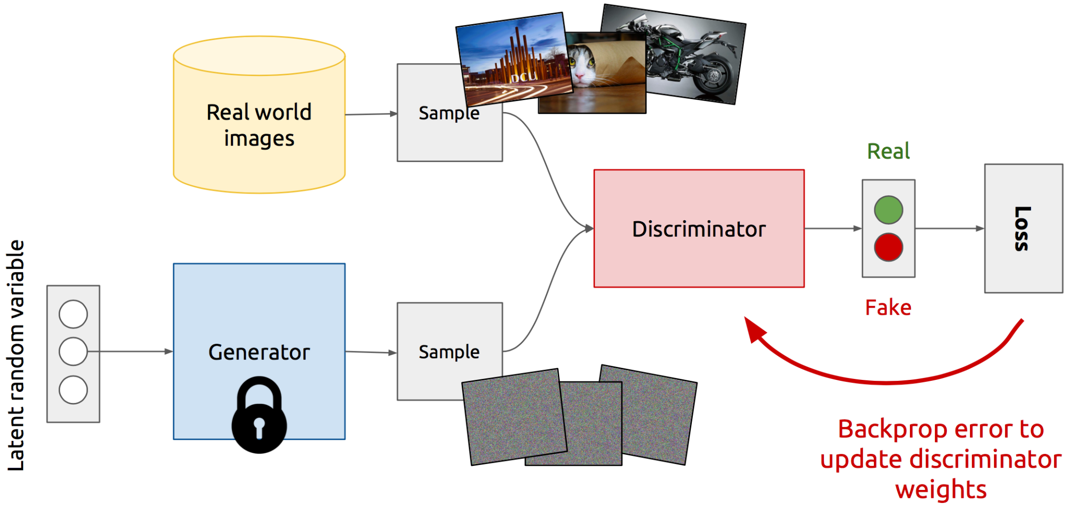
GAN Training (1)

- The two models, generator and discriminator, are trained together but in an alternating manner (not simultaneously)
  - Generator generates a batch of samples
  - These samples, along with real samples from the domain, are provided to the discriminator and classified as real or fake
  - Discriminator is updated to get better in discriminating the samples in the next round (while the generator is kept constant during the discriminator training phase)
  - Generator is updated based on how well it was able to fool the discriminator (discriminator is kept constant during the generator training phase)
  - Otherwise the generator would be trying to hit a moving target and might never converge
- Training is essentially playing a zero-sum game between generator and discriminator (competing against each other)
  - If the discriminator successfully identifies all samples, it is rewarded or not updated while the generator is penalised (and vice versa)

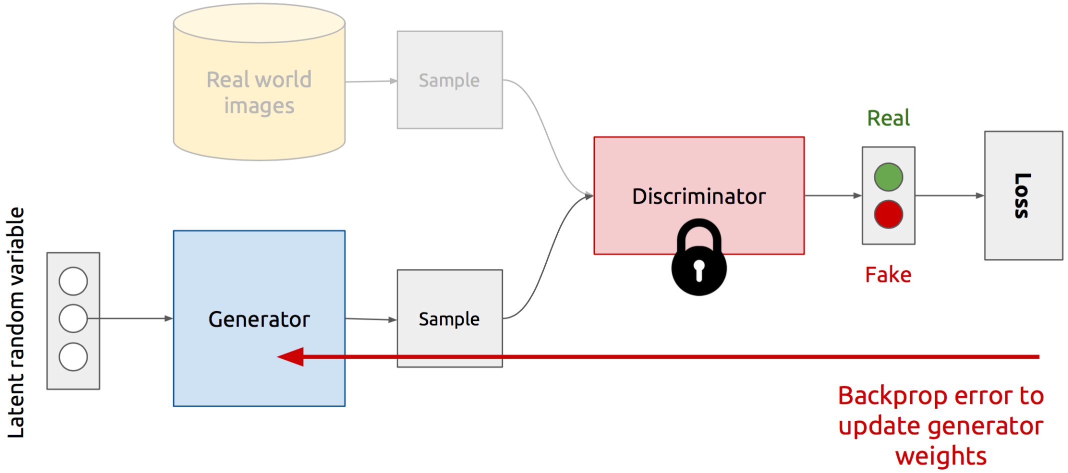
GAN Training (2)

- This progression poses a problem for convergence of the GAN as a whole:
  - The discriminator feedback gets less meaningful over time
  - If the GAN continues training past the point when the discriminator is giving completely random feedback
  - The generator starts to train on junk feedback, and its own quality may collapse
- As the generator improves with training, the discriminator performance gets worse because the discriminator can't easily tell the difference between real and fake. At a limit, the generator generates perfect replicas from the input domain each time and the discriminator is not able to distinguish between real and fake → predicts "unsure" each time (i.e. 50% real or fake)
  - In practice for a useful generator model we do not reach this point
  - For a GAN, convergence is often a fleeting, rather than stable, state

Training the Discriminator



Training the Generator (1)



Training the Generator (2)

- Backpropagation adjusts each weight in the right direction by calculating the weight's impact on the output
  - How the output would change if you changed the weight.
- **BUT**: the impact of a generator weight depends on the impact of the discriminator weights it feeds into
- Hence, backpropagation starts at the output and flows back through the discriminator into the generator
- At the same time, we don't want the discriminator to change during generator training.
  - Trying to hit a moving target would make a hard problem even harder for the generator
- After generating the examples and letting the discriminator classify a batch of samples (calculating the loss):
  - Backpropagate through both the discriminator and generator to obtain gradients
  - Use gradients to change only the generator weights

Alternating Training of the Generator and Discriminator

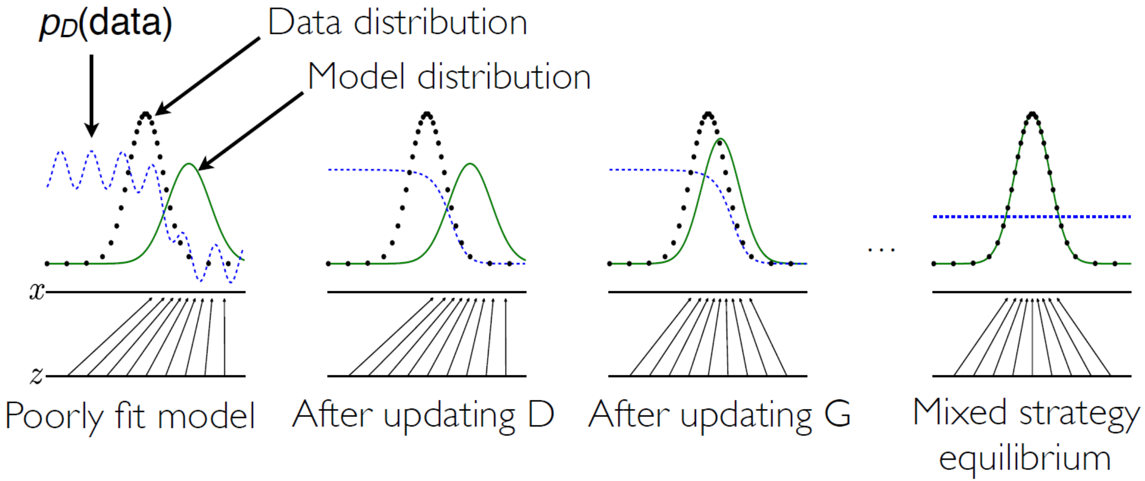


Figure: Alternating Training of the Generator and Discriminator

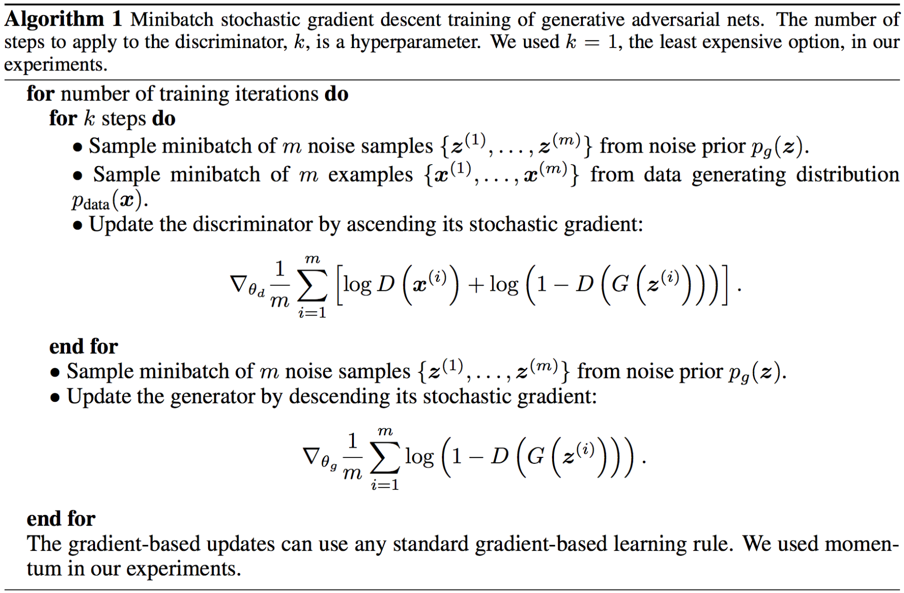
GAN Formulation (a bit more theoretic)

- $\underset{G}{min}\underset{D}{max} V(D, G)$
- Formulated as minimax as both are playing a zero-sum game, where:
  - Discriminator tries to maximise its reward $V(D, G)$
  - Generator tries to minimise Discriminator's reward (or maximise its loss):

$$V(D, G) = E_{x \sim p(x)}[logD(x)] + E_{z \sim p(z)}[log(1 - D(G(z)))]$$

- Nash equilibrium (game theoretic concept) of this game is achieved at:
  - $P_{data}(x) = P_{gen}(x)\ \forall x$
  - $D(x) = \frac{1}{2}\ \forall x$

GAN Training Algorithm

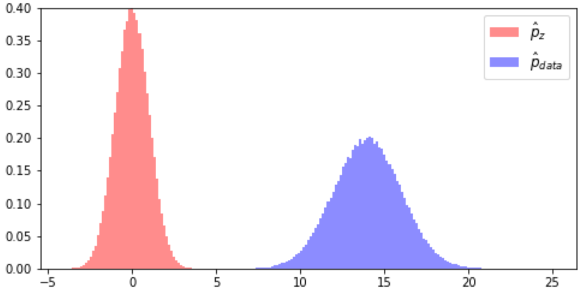


Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

```
for number of training iterations do
    for $k$ steps do
        • Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
        • Sample minibatch of $m$ examples $\{\boldsymbol{x}^{(1)}, \dots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.
        • Update the discriminator by ascending its stochastic gradient:
            $$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right)\right].$$
    end for
    • Sample minibatch of $m$ noise samples $\{\boldsymbol{z}^{(1)}, \dots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
    • Update the generator by descending its stochastic gradient:
        $$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$
end for
The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.
```

Exercise 1 - GAN principles

Exercise: Train a GAN that learns a simple training data distribution, e.g. a normal distribution. Material: Workshop Course Materials $\rightarrow$ GAN_principles_exercise.ipynb



Deep Convolutional GAN (DCGAN)

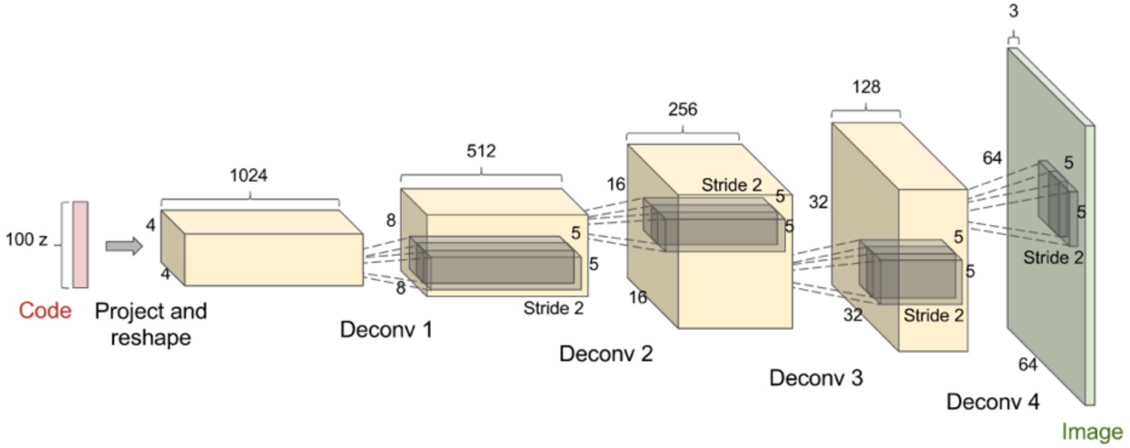


Figure: Deep Convolutional GAN Architecture

- Replace FC hidden layers with convolutions, Generator: Fractional-Strided convolutions
- Use batch Normalisation after each layer
- Inside Generator: Use ReLU for hidden layers, use Tanh for the output layer

Exercise 2 - DCGAN

Exercise: Train a DCGAN that learns the MNIST data distribution and is able to synthesize handwritten digits. Material: Workshop Course Materials → DCGAN_exercise.ipynb

Outline

Advantages and Challenges of GANs

- Advantages
  - Sampling (or sample generation) is straightforward
  - Training doesn't involve Maximum Likelihood estimation
  - Robust to overfitting (since the Generator never sees the actual training data, only presented to the Discriminator)
  - GANs are good at capturing the modes of the distribution (empirically)

- Challenges
  - Probability distribution $P(X)$ is implicit $\rightarrow$ not straightforward to compute $P(X)$; "Vanilla GANs" are only good for sampling/generation
  - Training a GAN is hard $\rightarrow$Non-Convergence; Mode-Collapse (explained below)

Importance of the Loss Function - Vanishing Gradients

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

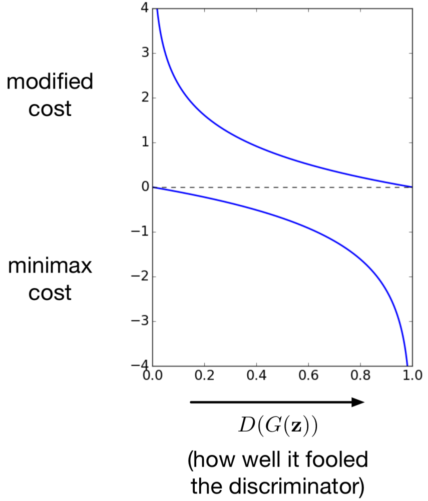
$$V(D,G) = E_{x\sim p(x)}[logD(x)] + E_{z\sim p(z)}[log(1-D(G(z)))]]$$

$$\nabla_{\theta_G} V(D,G) = \nabla_{\theta_G} E_{z\sim p(z)}[log(1-D(G(z)))]$$

- $\nabla_a log(1-\sigma(a)) = \frac{-\nabla_a \sigma(a)}{1-\sigma(a)} = \frac{-\sigma(a)(1-\sigma(a))}{1-\sigma(a)} = -\sigma(a) = -D(G(z))$
- Gradient goes to 0 if D is confident, i.e. $D(G(z)) \rightarrow 0$ (called saturation or vanishing gradient)

Vanishing Gradients - Solution

- Original minimax cost for the Generator:
  - $E_{z \sim p(z)}[log(1 - D(G(z)))]]$
- Modified cost for the Generator:
  - $E_{z \sim p(z)}[-log(D(G(z))))]]$
- Fixes the saturation problem



Vanishing Gradients - Wasserstein Loss (1)

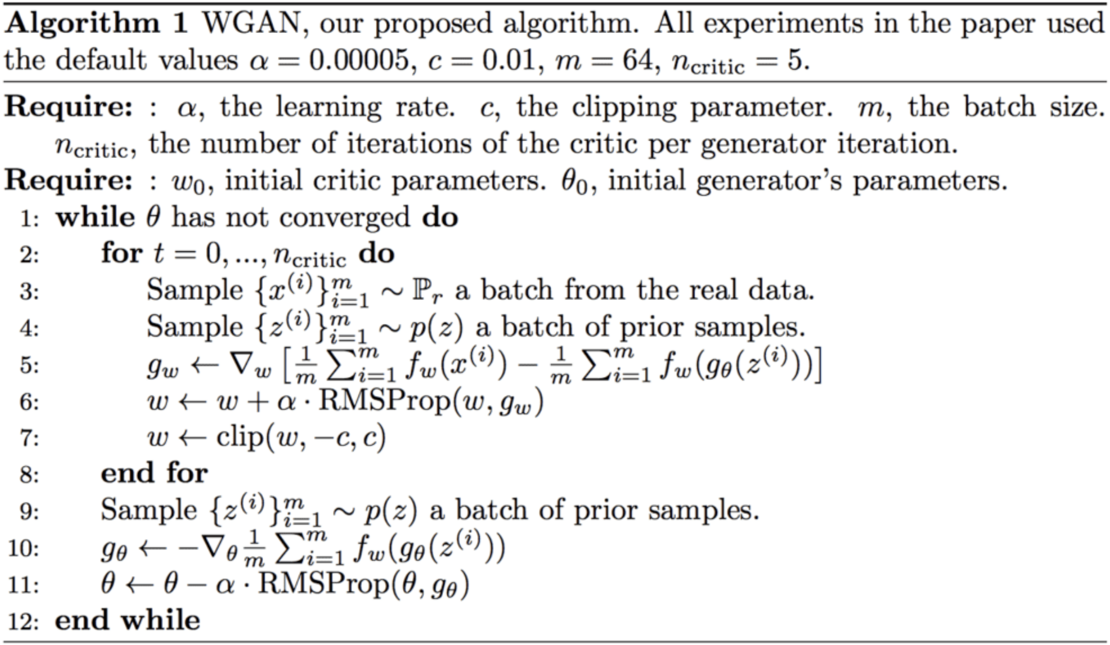
Wasserstein loss:

- Wasserstein loss is designed to prevent vanishing gradients even when the discriminator is trained to optimality
- Wasserstein loss function tries to increase the gap between the scores for real and generated images
- Discriminator (here called critic) loss: $D(x) - D(G(z))$ (discriminator tries to maximise this function)
- Generator loss: $D(G(z))$ (generator tries to maximise this function)
- The loss functions can be summarised as follows
  - Critic Loss = [average critic score on real images] – [average critic score on fake images]
  - Generator Loss = -[average critic score on fake images]
- Where the average scores are calculated across a mini-batch of samples
- This loss function depends is a modification of the GAN scheme (resulting in the so called **WGAN**) in which the discriminator does not actually classify instances

Vanishing Gradients - Wasserstein Loss (2)

- For each instance it outputs a number, but this number does not have to be less than one or greater than 0
  - Discriminator training just tries to make the output bigger for real instances than for fake instances
  - It can't really discriminate between real and fake
  - WGAN discriminator is actually called a "critic" instead of a "discriminator"
- Note: Theoretical justification for the WGAN requires that the weights throughout the GAN be clipped so that they remain within a constrained range
- Implementation of a WGAN requires a few minor changes to the DCGAN:
  - Use a linear activation function in the output layer of the critic model (instead of sigmoid)
  - Use Wasserstein loss to train the critic and generator models that promote larger difference between scores for real and generated images
  - Constrain critic model weights to a limited range after each mini batch update (e.g. [-0.01,0.01])

Vanishing Gradients - Wasserstein Loss (3)



Algorithm 1 WGAN, our proposed algorithm. All experiments in the paper used the default values $\alpha = 0.00005$, $c = 0.01$, $m = 64$, $n_{\text{critic}} = 5$.

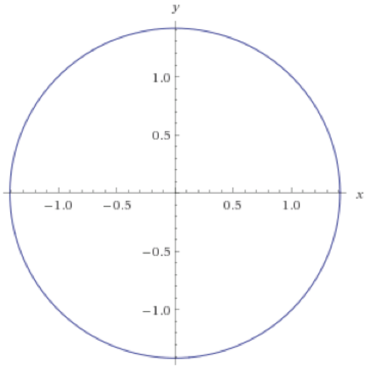
```
Require: : $\alpha$, the learning rate. $c$, the clipping parameter. $m$, the batch size.
    $n_{\text{critic}}$, the number of iterations of the critic per generator iteration.
Require: : $w_0$, initial critic parameters. $\theta_0$, initial generator's parameters.
 1: while $\theta$ has not converged do
 2:     for $t = 0, ..., n_{\text{critic}}$ do
 3:         Sample $\{x^{(i)}\}_{i=1}^{m} \sim \mathbb{P}_r$ a batch from the real data.
 4:         Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
 5:         $g_w \leftarrow \nabla_w \left[\frac{1}{m}\sum_{i=1}^{m} f_w(x^{(i)}) - \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))\right]$
 6:         $w \leftarrow w + \alpha \cdot \text{RMSProp}(w, g_w)$
 7:         $w \leftarrow \text{clip}(w, -c, c)$
 8:     end for
 9:     Sample $\{z^{(i)}\}_{i=1}^{m} \sim p(z)$ a batch of prior samples.
10:     $g_\theta \leftarrow -\nabla_\theta \frac{1}{m}\sum_{i=1}^{m} f_w(g_\theta(z^{(i)}))$
11:     $\theta \leftarrow \theta - \alpha \cdot \text{RMSProp}(\theta, g_\theta)$
12: end while
```

Non-Convergence (1)

- Deep Learning models (in general) involve a single "player"
  - Player tries to maximise its reward (or minimise its loss)
  - Stochastic Gradient Descent (SGD) in combination with backpropagation is used to find the optimal parameters
  - SGD has convergence guarantees (under certain conditions)
  - **Problem**: with non-convexity, it might converge to a local optimum: $\min_G L(G)$
- GANs involve two (or mode) players
  - Discriminator is trying to maximise its reward
  - Generator is trying to minimise Discriminators reward (zero-sum game): $\min_G \max_D V(D, G)$
  - SGD was never designed to find the Nash equilibrium of a game
  - **Problem**: might not converge to the Nash equilibrium at all

Non-Convergence (2)

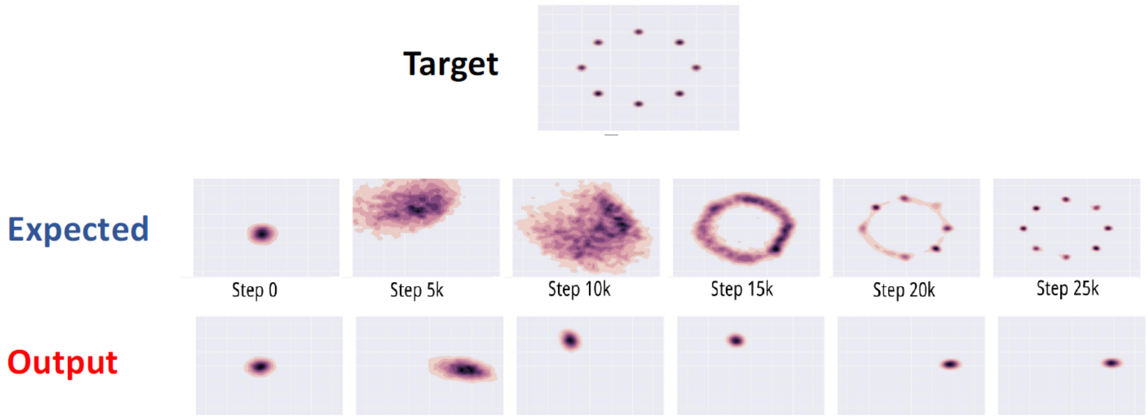
- Simple example, $\underset{x}{min}\underset{y}{max} V(x,y)$, let $V(x,y) = xy$
  - $\frac{\partial}{\partial x} = -y$, $\frac{\partial}{\partial y} = x$, $\frac{\partial^2}{\partial y^2} = \frac{\partial}{\partial x} = -y$ (differential equation's solution has sinusoidal terms)
  - State 1:$x > 0$, $y > 0$, $V > 0 \rightarrow$ increase y, decrease x
  - State 2: $x < 0$, $y > 0$, $V < 0 \rightarrow$ decrease y, decrease x
  - State 3: $x < 0$, $y < 0$, $V > 0 \rightarrow$ decrease y, increase x
  - State 4: $x > 0$, $y < 0$, $V < 0 \rightarrow$ increase y, increase x
  - State 5: $x > 0$, $y > 0$, $V > 0 \rightarrow$ increase y, decrease x
    - we are back to **State 1**!
- Even with a small learning rate, it will not converge at all!



Mode-Collapse (1)

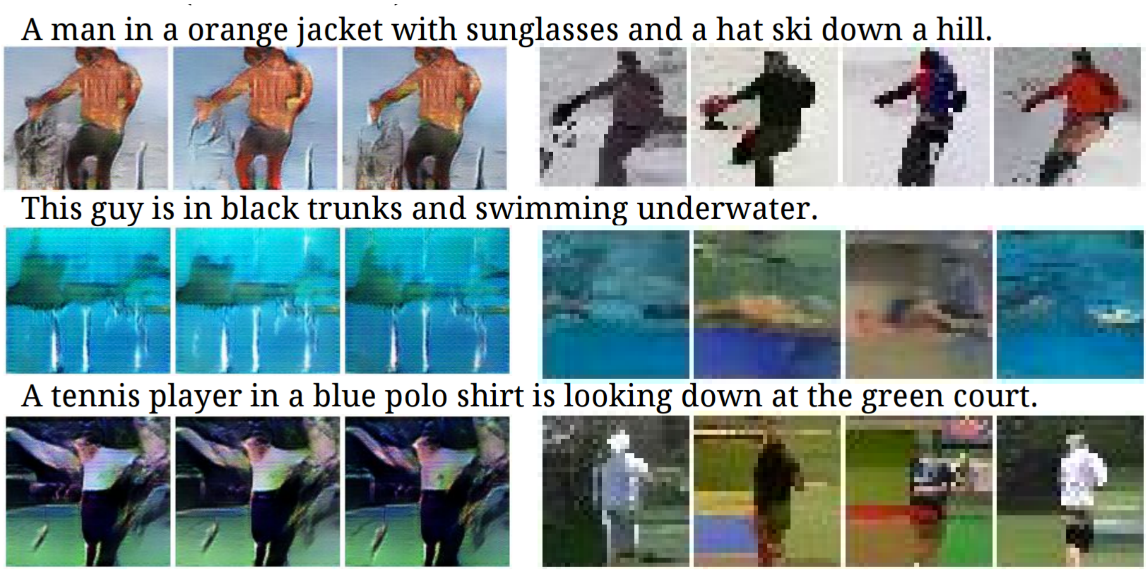
- Usually a GAN should produce a wide variety of outputs, e.g. a different face for every random input to a face generator
- However, if a generator produces an especially plausible output, the generator may learn to produce only that output
- In fact, the generator is always trying to find the one output that seems most plausible to the discriminator
- If the generator starts producing the same output (or a small set of outputs) over and over again, the discriminator's best strategy is to learn to always reject that output
- If the next generation of discriminator gets stuck in a local minimum and doesn't find the best strategy, then it's too easy for the next generator iteration to find the most plausible output for the current discriminator
- Each iteration of generator over-optimizes for a particular discriminator, and the discriminator never manages to learn its way out of the trap
- As a result the generators rotate through a small set of output types
- This form of GAN failure is called **mode collapse**

Mode-Collapse (2)



- Generator fails to output diverse samples and is stuck with only one "class" of samples or rotates through a small set of output types

Mode-Collapse Real Examples


A man in a orange jacket with sunglasses and a hat ski down a hill.
This guy is in black trunks and swimming underwater.
A tennis player in a blue polo shirt is looking down at the green court.

Solutions to Mode-Collapse: Wasserstein Loss and Unrolled GANs

Wasserstein loss:

- Alleviates mode collapse enabling to train the discriminator to optimality without worrying about vanishing gradients
- If the discriminator doesn't get stuck in local minima, it learns to reject the outputs that the generator stabilizes on
- Hence, the generator has to try something new to succeed, which prevents the mode collapse

Unrolled GANs:

- Use a generator loss function that incorporates not only the current discriminator's classifications, but also the outputs of future discriminator versions
- So the generator can't over-optimize for a single discriminator, again solving the mode-collapse problem

Solution to Mode-Collapse: Supervision with Labels

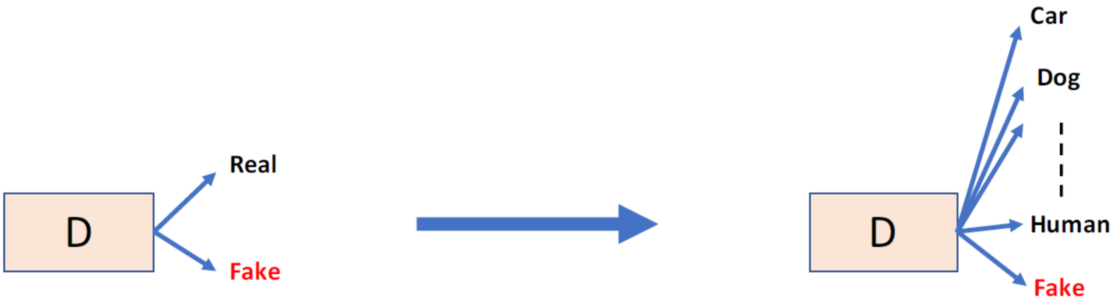


Figure: Instead of using only the two classes, use different labels for each subclass

- Additional label information of the real data might help the GAN to converge
- Empirically generates much better samples

One-Sided Label Smoothing

- Replace the default discriminator cost: `cross_entropy(1., discriminator(data)) + cross_entropy(0., discriminator(samples))`
- By the one-sided label smoothing cost: `cross_entropy(.9, discriminator(data)) + cross_entropy(0., discriminator(samples))`
- Or more general: `cross_entropy(1.-alpha, discriminator(data)) + cross_entropy(beta, discriminator(samples))`

- Do not smooth the negative labels!
- Benefits:
  - Does not reduce classification accuracy, only confidence
  - Prevents discriminator from giving very large gradient signal to generator
  - Prevents extrapolating to encourage extreme samples

Solution to Mode-Collapse: Mini-Batch GANs (1)

- At Mode-Collapse the Generator produces good samples, but only very few of them
  - Discriminator cannot tag them as fake
- To avoid the mode collapse, the Generator has to be rewarded for outputting diverse samples
  - How to reward sample diversity?
- To address this, the discriminator has to know about this edge-case (low sample diversity), by:
  - Letting the Discriminator look at the entire batch of samples instead of a single sample only
  - If there is lack of diversity, it will mark the examples as fake, even if they are good ones
- This will force the Generator to produce diverse samples instead of only very few good ones

Solution to Mode-Collapse: Mini-Batch GANs (2)

- Extract features that capture diversity within the mini-batch
  - E.g. use the L2 norm of the differences between all pairs of samples from the batch
- Feed those features to the discriminator along with the sample/image
- Feature values will differ between diverse and non-diverse batches
  - Hence, the Discriminator will rely on those features for classification as well
- In turn, this will:
  - Force the Generator to match those feature values with real data
  - Leading to the generation of diverse batches

Alternative Formulation of GAN Loss (1)

$$\min_{G}\max_{D} V(D,G) = E_{x\sim p(x)}[log D(x)] + E_{z\sim p(z)}[log(1 - D(G(z)))]]$$

$$D* = arg\max_{D} V(D,G) \quad G* = arg\min_{G} V(D,G)$$

- Here, the strategy of the Discriminator is $D(x) \rightarrow 1$, $D(G(z)) \rightarrow 0$
- Alternatively, flip the binary classification labels, i.e. **fake=1**, **real=0**

$$V(D,G) = E_{x\sim p(x)}[log(1 - D(x))] + E_{z\sim p(z)}[log(D(G(z)))]]$$

- Here, now the Discriminator's strategy is $D(x) \rightarrow 0$, $D(G(z)) \rightarrow 1$

Alternative Formulation of GAN Loss (2)

- If we only want to encode $D(x) \rightarrow 0$, $D(G(z)) \rightarrow 1$:

$$D* = argmax_D E_{x \sim p(x)}[log(1 - D(x))] + E_{z \sim p(z)}[log(D(G(z)))]$$

- We can use this: $D* = argmin_D E_{x \sim p(x)}[log(D(x))] + E_{z \sim p(z)}[log(1 - D(G(z)))]$
- Now we can replace the cross-entropy loss with any loss function (**Hinge Loss**)

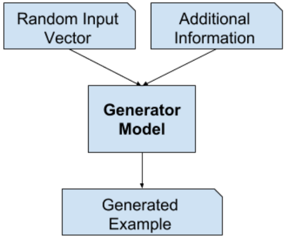
$$D* = argmin_D E_{x \sim p(x)}[D(x)] + E_{z \sim p(z)}[max(0, -D(G(z)))]$$

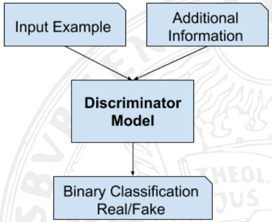
- Thus, instead of outputting probabilities, the Discriminator just has to output:
  - High values for fake samples
  - Low values for real samples
- Hence, we now have a much more general formulation of the loss function (more possibilities for suitable loss functions)

Outline

Conditional GAN Introduction

- Important extension to GANs for their use to conditionally generate output
- Generative model can be trained to generate new examples from the input domain, where the input (random vector from latent space) is provided with (conditioned by) some additional input
- Additional input could be a class value (e.g. a digit in case of handwritten digits)
- The discriminator is also conditioned, i.e. it is provided with both, the generated sample and the additional input
  - Teaches the generator to generate examples of that class (matching the additional input)



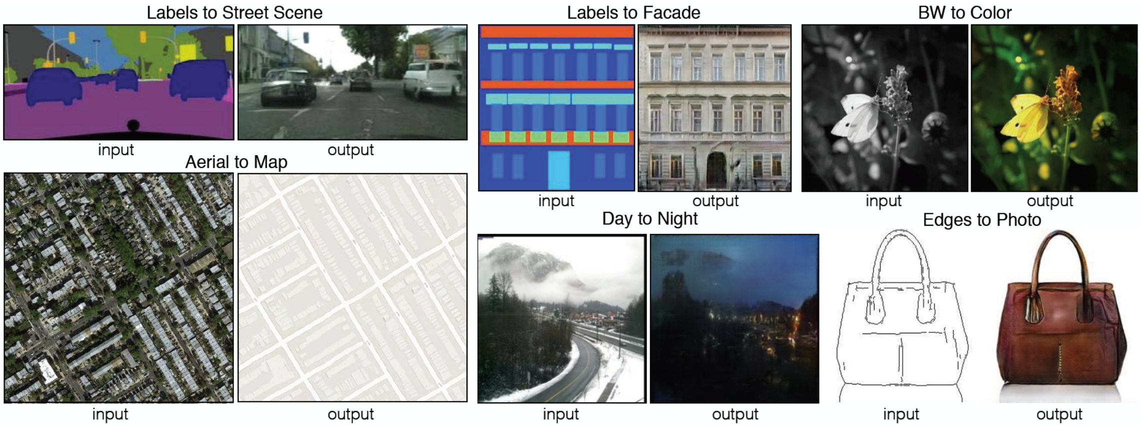


$$\underset{G}{min}\,\underset{D}{max}\, V(D,G) = E_{x\sim p(x)}[logD(x|\mathbf{y})] + E_{z\sim p(z)}[log(1 - D(G(z|\mathbf{y})))]]$$

Conditional GAN Advantages and Applications

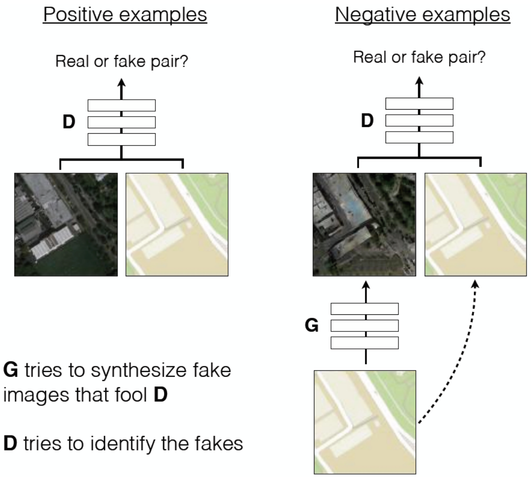
- Hence, a conditional GAN can be used to generate examples from a domain of a given type
- One step further: GAN models can be conditioned on an example from the domain, e.g. an image
- Leads to many practical applications of GANs when we have explicit supervision available:
    - text-to-image translation, image-to-image translation, ...
- GANs for transforming day to night images:
    - Discriminator is provided with examples of real and generated, night-time images
    - Discriminator is also conditioned on real daytime photos as input
    - Generator is provided with a random vector from the latent space (conditioned on) real daytime images as input

Conditional GAN Application: Image-to-Image Translation (1)



Conditional GAN Application: Image-to-Image Translation (2)

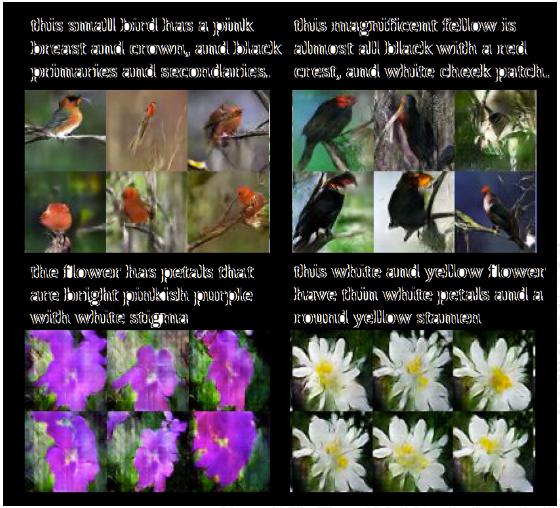
- Architecture: DCGAN-based architecture
- Training is conditioned on the images from the source domain
- Conditional GANs provide an effective way to handle many complex domains without worrying about designing structured loss functions explicitly
- Here, the loss is a weighted combination of the usual discriminator-based loss and a pixel-wise loss that penalizes the generator for departing from the source image too much


Positive examples
Negative examples
Real or fake pair?
Real or fake pair?
D
D
G
G tries to synthesize fake images that fool **D**
D tries to identify the fakes

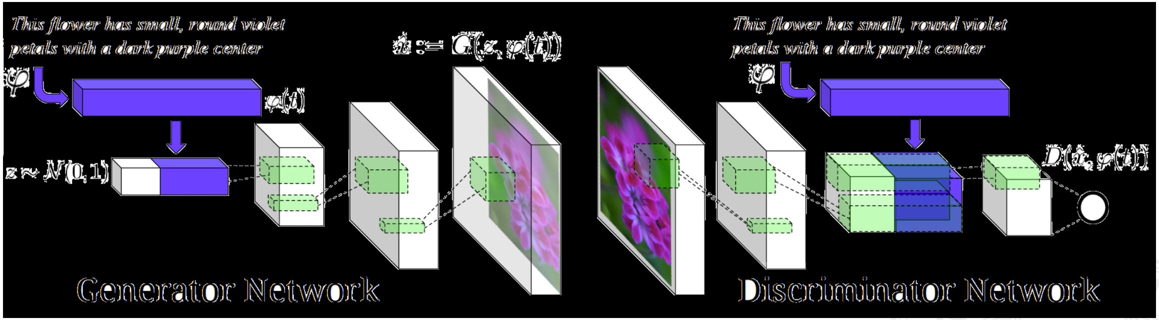
Conditional GAN Application: Text-to-Image Translation (1)

Motivation:

- Given a text description, generate images closely associated
- Uses a conditional GAN with the generator and discriminator being condition on "dense" text embedding

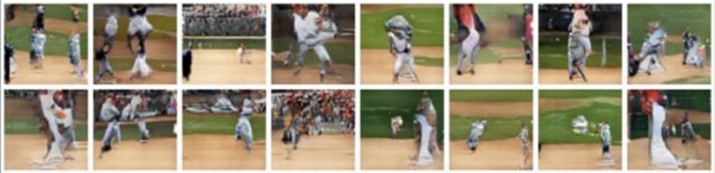
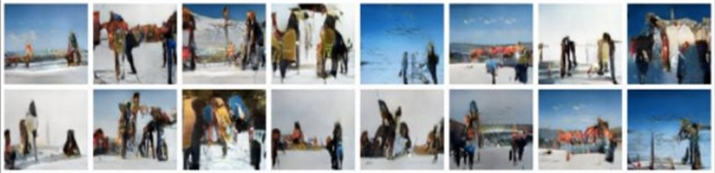
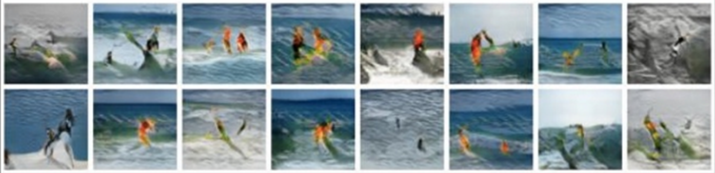

this small bird has a pink breast and crown, and black primaries and secondaries.
this magnificent fellow is almost all black with a red crest, and white cheek patch.
the flower has petals that are bright pinkish purple with white stigma
this white and yellow flower have thin white petals and a round yellow stamen

Conditional GAN Application: Text-to-Image Translation (2)

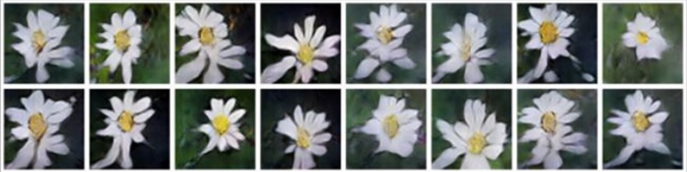
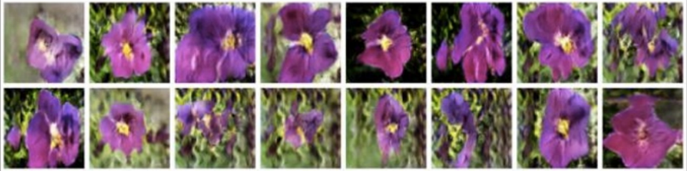
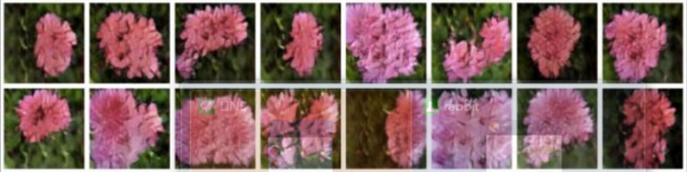

This flower has small, round violet petals with a dark purple center
φ
$\varphi(t)$
$z \sim \mathcal{N}(0,1)$
$\hat{x} := G(z, \varphi(t))$
Generator Network
This flower has small, round violet petals with a dark purple center
φ
$D(\hat{x}, \varphi(t))$
Discriminator Network

- Positive example: real image, right text
- Negative examples:
  - real image, wrong text
  - fake image, right text

Conditional GAN Application: Text-to-Image Translation (3)

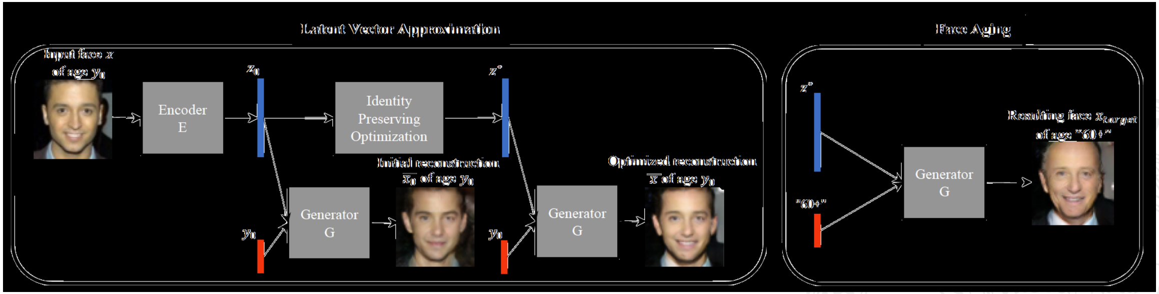
Caption	Image
a pitcher is about to throw the ball to the batter	
a group of people on skis stand in the snow	
a man in a wet suit riding a surfboard on a wave	

Conditional GAN Application: Text-to-Image Translation (4)

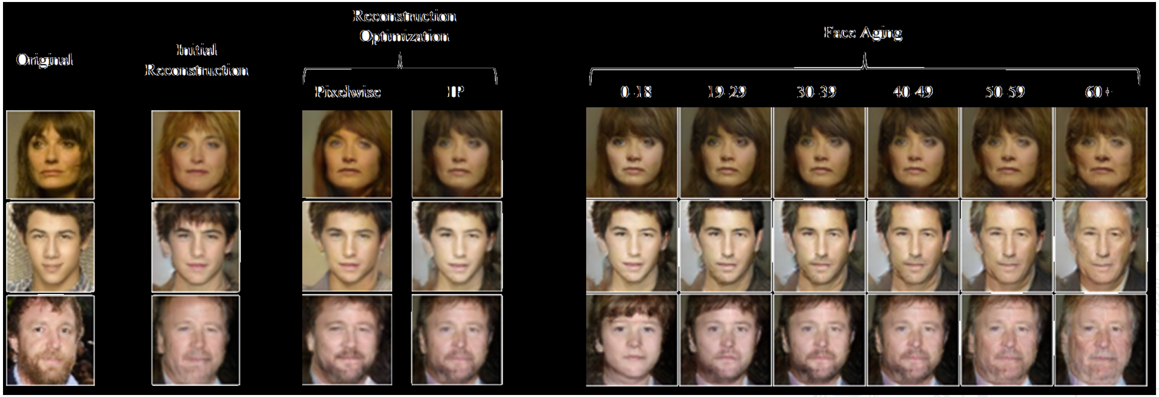
Caption	**Image**
this flower has white petals and a yellow stamen	
the center is yellow surrounded by wavy dark purple petals	
this flower has lots of small round pink petals	

Conditional GAN Application: Face Ageing (1)

- Differentiating Feature: Uses an Identity Preservation Optimization using an auxiliary network to get a better approximation of the latent code (z*) for an input image
- Latent code is then conditioned on a discrete (one-hot) embedding of age categories

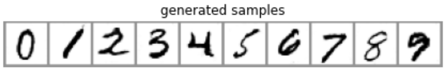

Latent Vector Approximation
Face Aging
Input face x of age y_0
Encoder E
z_0
Identity Preserving Optimization
z^*
Initial reconstruction $\bar{x}_0$ of age y_0
Optimized reconstruction $\bar{x}$ of age y_0
Generator G
y_0
Generator G
y_0
z^*
"60+"
Generator G
Resulting face x_{target} of age "60+"

Conditional GAN Application: Face Ageing (2)



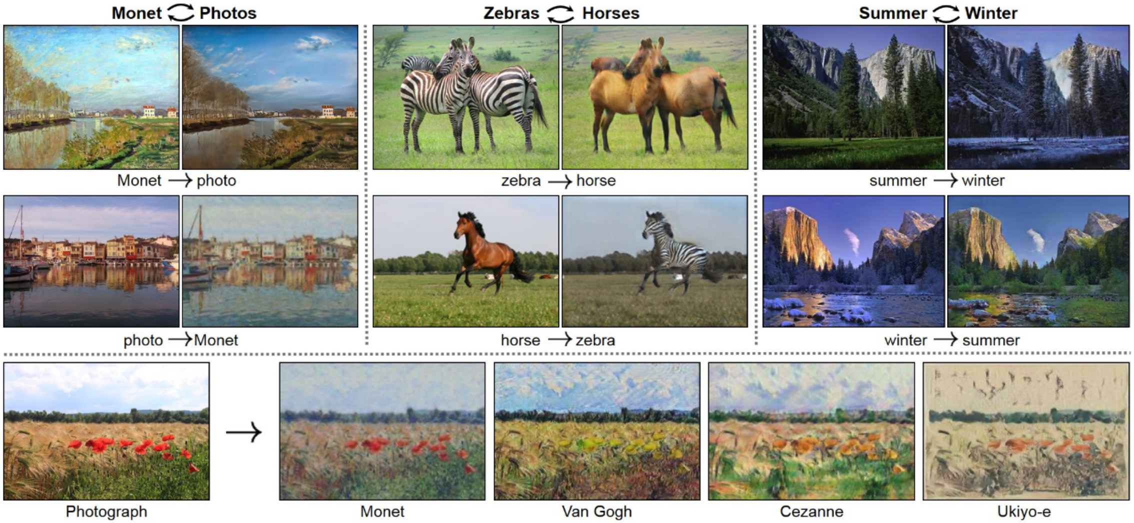
Exercise 3 - cGAN

Exercise: Train a conditional GAN that is able to synthesize specific handwritten digits.
Material: Workshop Course Materials → cGAN_exercise.ipynb



Outline

Cycle GAN Introduction

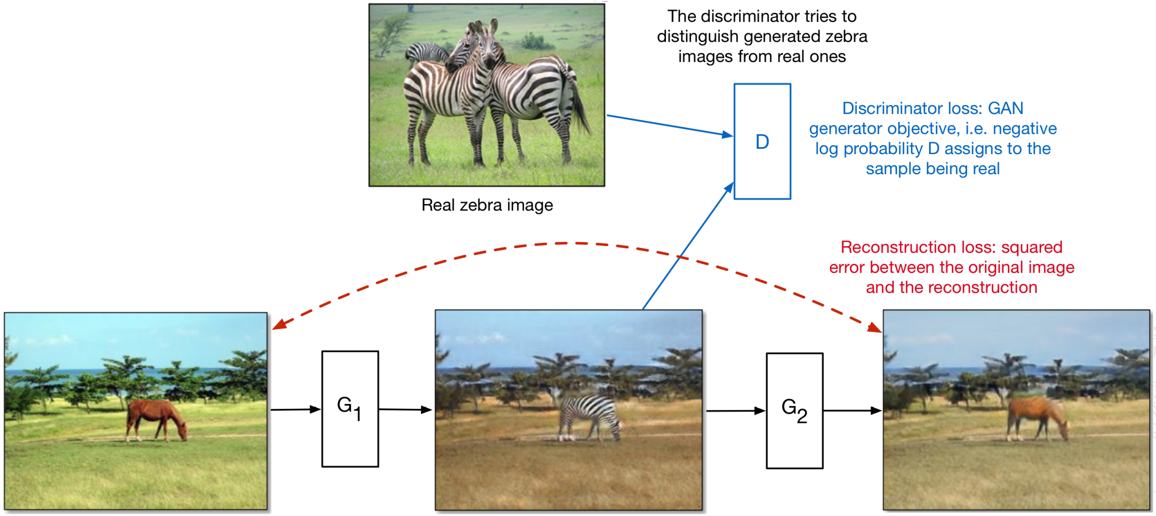


- Style transfer problem: change the style of an image while preserving its content
- Data: two unrelated collections of images, one for each style

Cycle GAN Basics

- If we had paired data (same content in both styles), this would be a supervised learning problem
- In reality, this is hard to find $\rightarrow$
- CycleGAN architecture learns to do it from unpaired data:
  - Train two different generator nets to go from style 1 to style 2, and vice versa
  - Make sure the generated samples of style 2 are indistinguishable from real images by a discriminator net
  - Make sure the generators are cycle-consistent: mapping from style 1 to style 2 and back again should give you almost the original image

Cycle GAN Basic Principle


The discriminator tries to distinguish generated zebra images from real ones
D
Discriminator loss: GAN generator objective, i.e. negative log probability D assigns to the sample being real
Real zebra image
Reconstruction loss: squared error between the original image and the reconstruction
G$_1$
G$_2$

Input image (real horse image)

Generator 1 learns to map from horse images to zebra images while preserving the structure

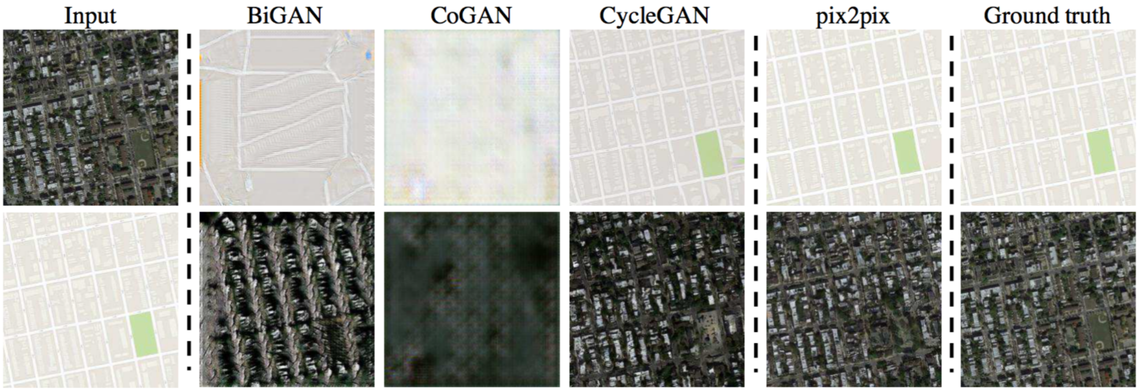
Generated sample

Generator 2 learns to map from zebra images to horse images while preserving the structure

Reconstruction

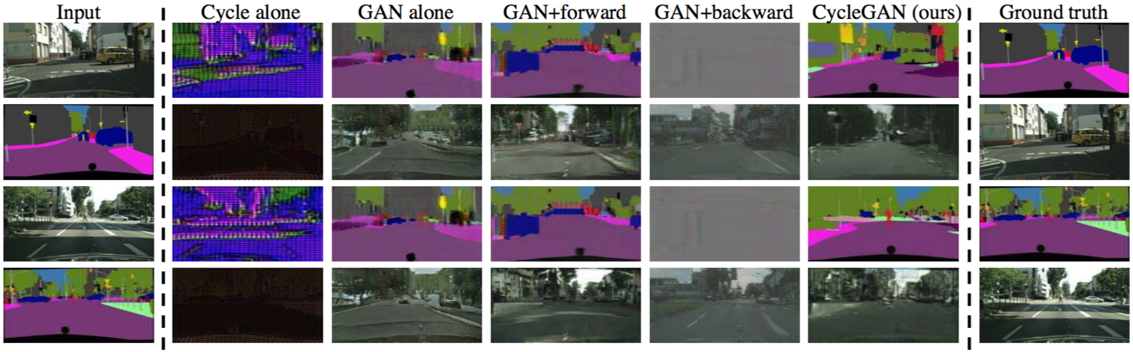
Cycle GAN Application 1

- Style transfer between aerial photos and maps:



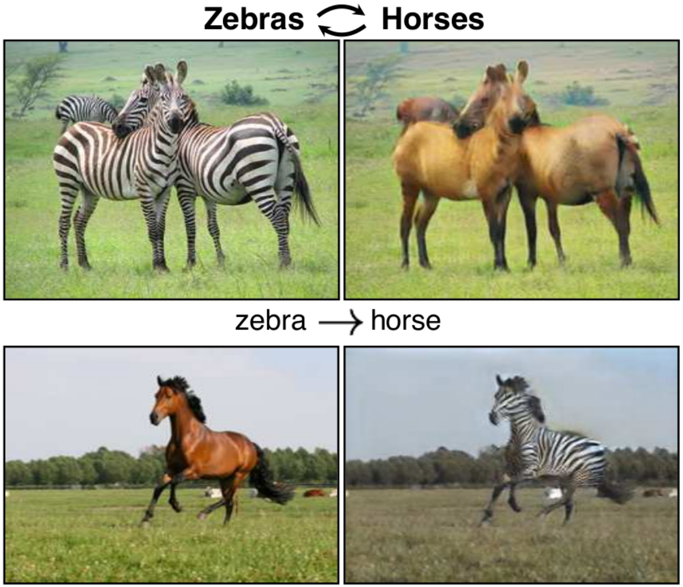
Cycle GAN Application 2

- Style transfer between road scenes and semantic segmentations (labels of every pixel in an image by object category):



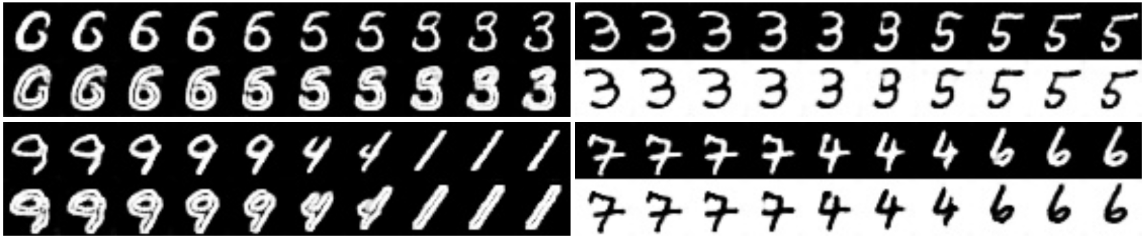
Exercise 4 - Image to Image translation

Exercise: Train a cycle GAN that translates an image of a horse into a zebra an vice versa.
Material: Workshop Course Materials → cycleGAN_exercise.ipynb



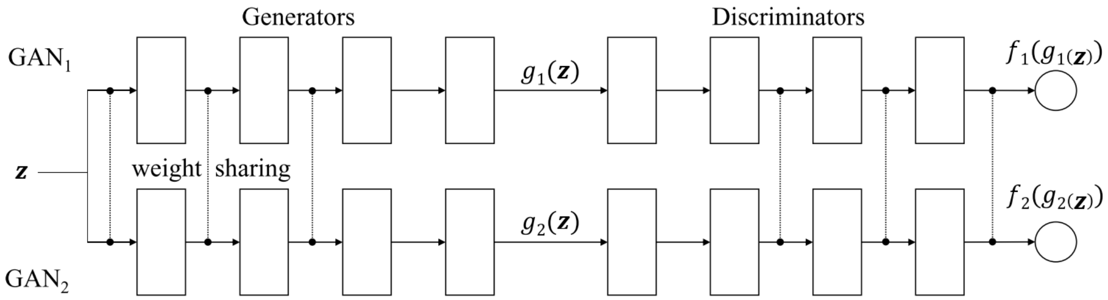
Outline

CoupledGAN Introduction



- Learning a joint distribution of multi-domain images
- Using GANs to learn the joint distribution with samples drawn from the marginal distributions
- Direct applications:
  - Domain adaptation
  - Image translation

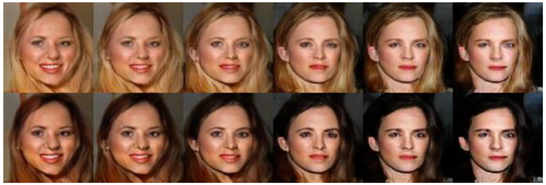
CoupledGAN Architecture

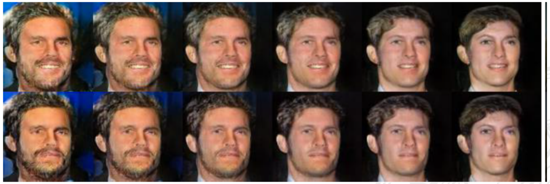


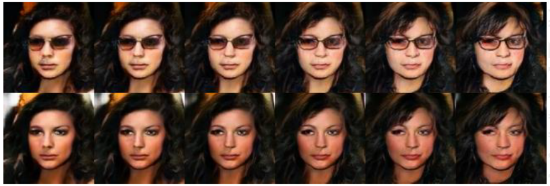
- Weight-sharing constraints the network to learn a joint distribution without corresponding supervision

CoupledGAN Application

- Some examples of generating facial images across different feature domains
- Corresponding images in a column are generated from the same latent code z







Outline

GAN Recommendations I

In the following, several practical tips and tricks for GAN training from: `https://github.com/soumith/ganhacks`

1. Normalize the inputs
   - normalize the images between -1 and 1
   - Use *tanh* as the last layer of the generator output
2. A modified loss function
   - Usual loss function to optimise G is: $min(log(1 - D))$, but in practice $max(log(D))$ is usually used because:
     - the first formulation has vanishing gradients early on
   - Also works well in practice:
     - Flip labels when training generator: real = fake, fake = real
3. Use a spherical Z
   - Don't sample from a Uniform distribution
   - Instead sample from a Gaussian distribution
   - When doing interpolations, do the interpolation via a great circle, rather than a straight line from point A to point B
   - Tom White's Sampling Generative Networks ref code `https://github.com/dribnet/plat` has more details

GAN Recommendations II

4. BatchNorm
   - Construct different mini-batches for real and fake, i.e. each mini-batch needs to contain only all real images or all generated images
   - If batch norm is not an option use instance normalization (for each sample, subtract mean and divide by standard deviation)
5. Avoid Sparse Gradients: ReLU, MaxPool
   - Stability of the GAN game suffers if you have sparse gradients
   - LeakyReLU = good (in both G and D)
   - For Downsampling, use: Average Pooling, Conv2d + stride
   - For Upsampling, use: PixelShuffle (`https://arxiv.org/abs/1609.05158`), ConvTranspose2d + stride
6. Use Soft and Noisy Labels
   - Label Smoothing, i.e. if you have two target labels: Real=1 and Fake=0, then for each incoming sample, if it is real, then replace the label with a random number between 0.7 and 1.2, and if it is a fake sample, replace it with 0.0 and 0.3 (for example)
   - Make the labels the noisy for the discriminator: occasionally flip the labels when training the discriminator

GAN Recommendations III

7. DCGAN / Hybrid Models
   - Use DCGAN when you can, it works and produces better results
   - If you cant use DCGANs and no model is stable, use a hybrid model : KL + GAN or VAE + GAN
8. Use stability tricks from RL
   - Experience Replay
     - Keep a replay buffer of past generations and occasionally show them
     - Keep checkpoints from the past of G and D and occasionally swap them out for a few iterations
   - All stability tricks that work for deep deterministic policy gradients
9. Use the ADAM Optimizer
   - Optimise Adam rules
   - Use SGD for discriminator and ADAM for generator
10. Track failures early
    - D loss goes to 0: failure mode
    - check norms of gradients: if they are over 100 things are screwing up
    - when things are working, D loss has low variance and goes down over time vs having huge variance and spiking
    - if loss of generator steadily decreases, then it's fooling D with garbage

GAN Recommendations IV

11. Don't balance loss via statistics (unless you have a good reason to)
    - Don't try to find a (number of G / number of D) schedule to uncollapse training
    - It's hard and we've all tried it.
    - If you do try it, have a principled approach to it, rather than intuition
    - E.g.: `while lossD > A: train D, while lossG > B: train G`
12. If you have labels, use them
    - If you have labels available, training the discriminator to also classify the samples: axillary GANs
13. Add noise to inputs, decay over time
    - Add some artificial noise to inputs to D (Arjovsky et. al., Huszar, 2016)
        - `http://www.inference.vc/instance-noise-a-trick-for-stabilising-gan-training/`
        - `https://openreview.net/forum?id=Hk4_qw5xe`
    - Adding Gaussian noise to every layer of generator (Zhao et. al. EBGAN)
        - Improved GANs: OpenAI code also has it (commented out)
14. Train discriminator more (sometimes)
    - Especially when you have noise
    - Hard to find a schedule of number of D iterations vs G iterations

GAN Recommendations V

15 Batch Discrimination
- Mixed results

16 Discrete variables in Conditional GANs
- Use an Embedding layer
- Add as additional channels to images
- Keep embedding dimensionality low and upsample to match image channel size

17 Use Dropouts in G in both train and test phase
- Provide noise in the form of dropout (50%)
- Apply on several layers of our generator at both training and test time
- `https://arxiv.org/pdf/1611.07004v1.pdf`

End

Thank you for your attention!

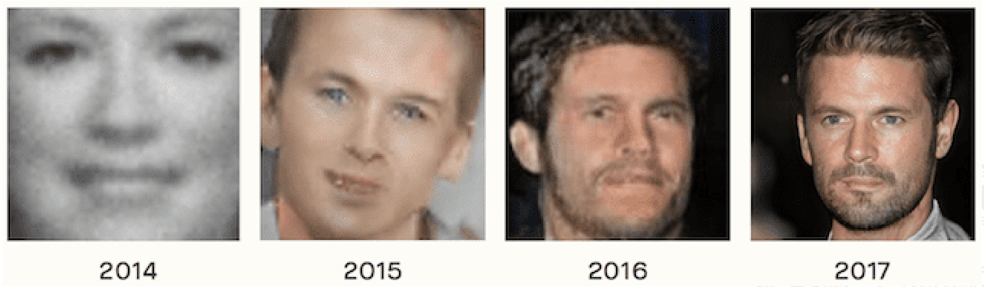


Figure: Example of the Progression in the Capabilities of GANs From 2014 to 2017

Questions?

Sources I

- Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A. and Bengio, Y. Generative adversarial nets, NIPS (2014).
- Goodfellow, Ian NIPS 2016 Tutorial: Generative Adversarial Networks, NIPS (2016).
- https://www.iangoodfellow.com/slides/
- https://github.com/soumith/ganhacks
- https://machinelearningmastery.com/what-are-generative-adversarial-networks-gans/
- https://www.deeplearningbook.org/