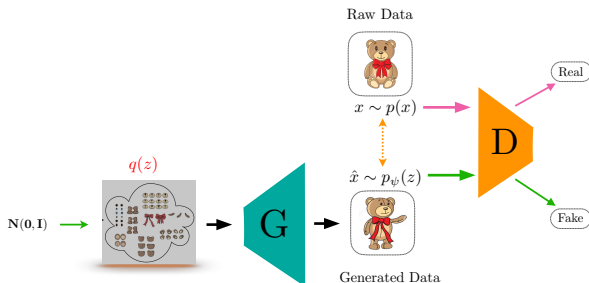


11-695: AI Engineering

GAN Implementation

Spring 2019

- ① GAN: Review
- ② Implementing GAN: A case of MNIST
- ③ (Some) Training Problems
- ④ (Some) Work to Improve Training
- ⑤ GAN Evaluation

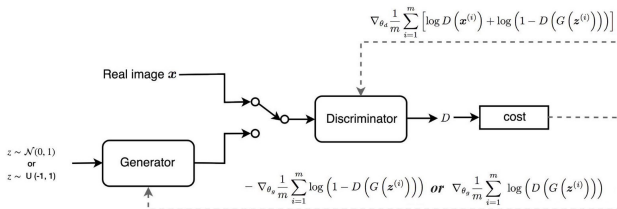


- Objective:

$$\min_{\psi} \max_{\theta} [\mathbb{E}_{x \sim p(x)} \log D_{\theta}(x) + (1 - \mathbb{E}_{z \sim p(z)} \log D_{\theta}(G_{\psi}(z)))]$$

- We alternate in that:

- Make G constant, let D **maximize** the objective
- Make D constant, let G **minimize** the objective



- Empirical expectation for gradient of D_θ :

$$\nabla_\theta = \frac{1}{m} \sum_{i=1}^m [\log D_\theta(x^{(i)}) + \log(1 - D_\theta(G_\psi(z^{(i)})))]$$

- Empirical expectation for gradient of G_ψ :

$$\nabla_\psi = \frac{1}{m} \sum_{i=1}^m [\log(1 - D_\theta(G_\psi(z^{(i)})))]$$

- Batching for D_θ : half real and half fake.

- ① GAN: Review
- ② Implementing GAN: A case of MNIST
- ③ (Some) Training Problems
- ④ (Some) Work to Improve Training
- ⑤ GAN Evaluation

generator.py

```
1 class Generator(tf.keras.Model):
2     def __init__(self, ...):
3         self.fc1 = layers.Dense(6 * 6 * 128, activation=tf.tanh)
4         self.conv1 = layers.Conv2DTranspose(...)
5         self.conv2 = layers.Conv2DTranspose(...)
6
7     def call(self, inputs):
8         """ Input is the noise """
9         x = self.fc1(inputs)
10        x = tf.reshape(x, shape=self._pre_conv_shape)
11        x = self.deconv1(x)
12        x = self.deconv2(x)
13        return x
```

- Recall: images data $\rightarrow$ we use Conv/Deconv
- Normally, “expanding” the noise to the desired output dimension
- Similar to AutoEncoder, it’s natural to use DeConv
- Here we use a similar ops: $\text{TransposeConv} = \text{padding} + \text{conv}$

discriminator.py

```
1 class Discriminator(tf.keras.Model):
2     def __init__(self, ...):
3         self.conv1 = layers.Conv2D(...)
4         self.pool1 = layers.AveragePooling2D(...)
5         ...
6         self.fc2 = layers.Dense(1, activation=None)
7
8     def call(self, inputs):
9         """ Input is the noise """
10        x = tf.reshape(x, some_shape)
11        x = self.conv1(x)
12        x = self.pool1(x)
13        ...
14        x = self.fc2(x)
15        return x
```

- Recall: just a binary case of image classification
- As opposed to Generator, it uses Convolution
- By now, you can see the similarity between GAN and AE?

discriminator_loss.py

```
1 loss_on_real = tf.losses.sigmoid_cross_entropy(  
2     tf.ones_like(discriminator_real_outputs),  
3     discriminator_real_outputs,  
4     label_smoothing=0.25)  
5  
6 loss_on_generated = tf.losses.sigmoid_cross_entropy(  
7     tf.zeros_like(discriminator_gen_outputs), discriminator_gen_outputs)  
8  
9 loss = loss_on_real + loss_on_generated  
10 return loss
```

- Recall: $\nabla_{\theta} = \frac{1}{m} \sum_{i=1}^m [\log D_{\theta}(x^{(i)}) + \log(1 - D_{\theta}(G_{\psi}(z^{(i)})))]$
- There are 2 components: for real and for fake images
- Useful regularization trick: label smoothing to model uncertainty.
E.g. 0.9 and 0.1 instead of 1.0 and 0.0

generator_loss.py

```
1 loss = tf.losses.sigmoid_cross_entropy(  
2     tf.ones_like(discriminator_gen_outputs), discriminator_gen_outputs)  
3  
4 return loss
```

- Recall: $\nabla_{\psi} = \frac{1}{m} \sum_{i=1}^m [\log(1 - D_{\theta}(G_{\psi}(z^{(i)})))]$
- It has only the latter term as compared to Discriminator
- Why do we use sigmoid?

forward.py

```
1 total_discriminator_loss = 0.0
2 for batch in all_batches:
3     with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
4         generated_images = generator(noise)
5
6         discriminator_gen_outputs = discriminator(generated_images)
7         discriminator_real_outputs = discriminator(images)
8
9         discriminator_loss_val = discriminator_loss(discriminator_real_outputs,
10                                                    discriminator_gen_outputs)
11         total_discriminator_loss += discriminator_loss_val
12
13         generator_loss_val = generator_loss(discriminator_gen_outputs)
14         total_generator_loss += generator_loss_val
```

- Alternating the updates for G and D
- We need 2 types of gradient, then 2 tapes

backward.py

```
1 total_generator_loss = 0.0
2 total_discriminator_loss = 0.0
3 for (num, images) in enumerate(dataset):
4     with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:
5         ...
6
7     # outside tape context:
8     generator_grad = gen_tape.gradient(generator_loss_val,
9                                       generator.variables)
10    discriminator_grad = disc_tape.gradient(discriminator_loss_val,
11                                           discriminator.variables)
12
13    generator_optimizer.apply_gradients(
14        zip(generator_grad, generator.variables))
15    discriminator_optimizer.apply_gradients(
16        zip(discriminator_grad, discriminator.variables))
```

- We need to update G and D separately, making use of 2 tapes
- Usually, Adam performs well

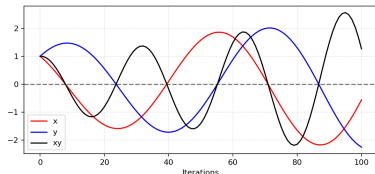
input.py

```
1     noise = tf.random_uniform(  
2         shape=[current_batch_size, noise_dim],  
3         minval=-1.,  
4         maxval=1.,  
5         seed=batch_index)
```

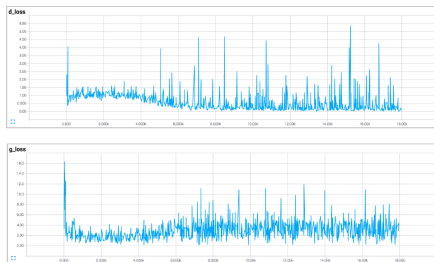
- Usually needs normalization
- Bound it in $[-1, 1]$
- Instead of Uniform, use Gaussian (box vs. ball)
- Sometimes also need to add some noise such as Gaussian
- Play more with [▶ code](#)

- ① GAN: Review
- ② Implementing GAN: A case of MNIST
- ③ (Some) Training Problems
- ④ (Some) Work to Improve Training
- ⑤ GAN Evaluation

So far: easy code, so as training? NO!

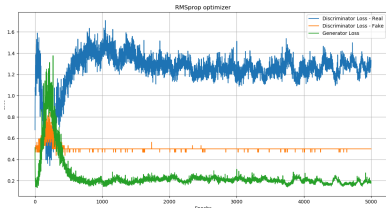


- We have 2 networks, and we alternate in updating them
 - Can cause “oscillation” or “overshooting”
 - Training similar to the EM algorithm
- Recall: To achieve Nash-Equilibrium in this non-cooperative game: result is constant regardless one player’s action
- Example: G to update x for $\min_x f_1(x) = xy$ and D to update y for $\max_y f_2(y) = xy$
 - $\nabla_{f_1} = y \Rightarrow x^{t+1} = x^t - \eta y$ while $\nabla_{f_2} = -x \Rightarrow y^{t+1} = y^t + \eta x$
 - Oscillation happens $\rightarrow$ Hard to achieve equilibrium



- Recall for G_ψ : $\nabla_\psi = \frac{1}{m} \sum_{i=1}^m [\log(1 - D_\theta(G_\psi(z^{(i)})))]$
- Recall: The job of D is easier: binary classification
- Right from the start, G is bad, D is much better:
 - so always $\log(1 - D_\theta(G_\psi(z^{(i)}))) = 0$
 - means G learns nothing
 - (relation to Posterior collapse in VAE?)

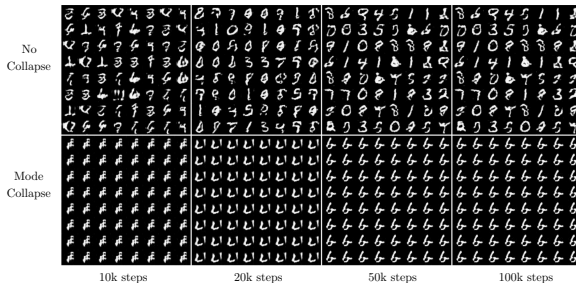
Image credit: carpedm20 github



- Be careful: Losses visualization might not be so interpretable!
- The optimization objective is different in GAN
 - In others, you optimize towards the ground truths
 - Now you are optimizing towards your opponent
- It can happen: loss increases, but the result quality *also* increases!!!
- But if D loss goes to zero, things might be wrong.

Image credit: Mirantha Jayathilaka (Medium.com), Soumith Chintala

Mode (Modal) Collapse

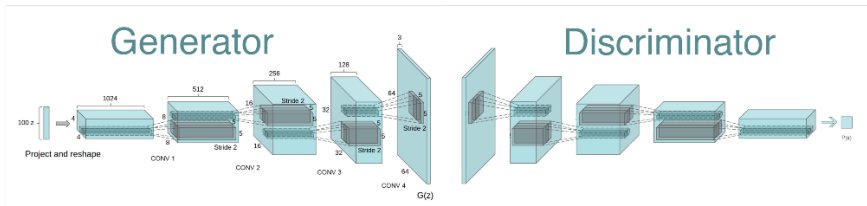


- Real world distribution is multi-modal and complex
- Can be “stuck” in one mode by pushing to optimize it hard
 - E.g. Always generate (almost perfect) digit 1 in MNIST
 - In (theoretical) optimization perspective, nothing’s wrong → tricky, machine is not human!
 - But G over-simplifies the data distribution
- A painfully hard yet very common problem

Image credit: Luke Metz *et al.* 2017

- ① GAN: Review
- ② Implementing GAN: A case of MNIST
- ③ (Some) Training Problems
- ④ (Some) Work to Improve Training
- ⑤ GAN Evaluation

- Deep Convolutional (DCGAN)
- Stacked GAN (SGAN)
- Progressive GAN
- Self-Attention GAN (SAGAN)



- Convolutional instead of Max Pooling (non-invertable)
- Use TransposeConv for upsampling
- No FC layer
- ReLU for G (except last layer uses Tanh), Leaky ReLU for D
- Use Batch normalization BN.

dcgan_generator.py

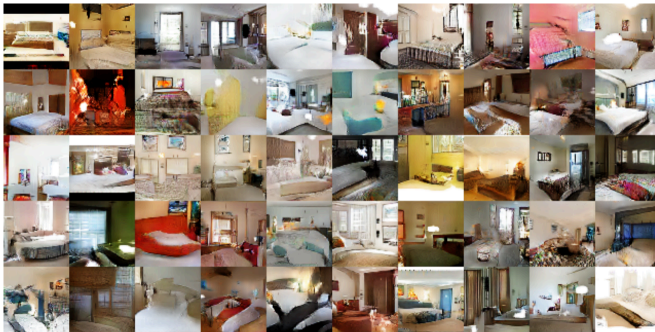
```
1 model = tf.keras.Sequential()
2 model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
3 model.add(layers.BatchNormalization())
4 model.add(layers.LeakyReLU())
5
6 model.add(layers.Reshape((7, 7, 256)))
7 assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size
8 model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1),
9                                 padding='same', use_bias=False))
10 assert model.output_shape == (None, 7, 7, 128)
11 model.add(layers.BatchNormalization())
12 model.add(layers.LeakyReLU())
13
14 model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2),
15                                 padding='same', use_bias=False))
16 assert model.output_shape == (None, 14, 14, 64)
17 model.add(layers.BatchNormalization())
18 model.add(layers.LeakyReLU())
19
20 model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2),
21                                 padding='same', use_bias=False, activation='tanh'))
```

- Note the changes: no FC, BatchNorm and LeakyReLU and Tanh

dcgan_discriminator.py

```
1 model = tf.keras.Sequential()
2 model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',
3                           input_shape=[28, 28, 1]))
4 model.add(layers.LeakyReLU())
5 model.add(layers.Dropout(0.3))
6
7 model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
8 model.add(layers.LeakyReLU())
9 model.add(layers.Dropout(0.3))
10
11 model.add(layers.Flatten())
12 model.add(layers.Dense(1))
```

- Note the changes: no intermediate FC, Dropout and Leaky ReLU
- Play more with [code](#)



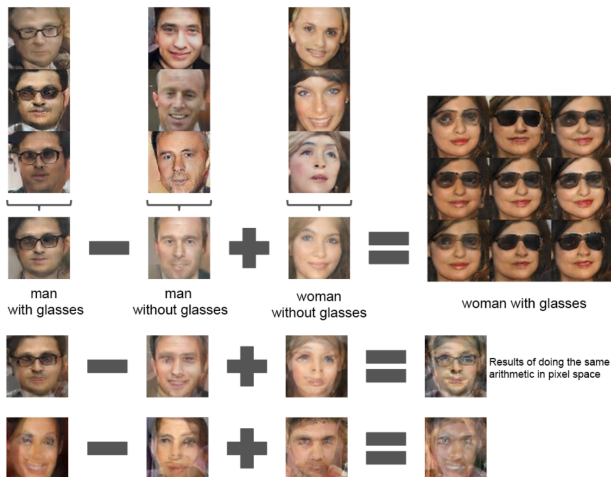
- Mark an important milestone in GAN development
- Still a popular and widely-adopted one

Image credit: Alec Radford & Luke Metz & Soumith Chintala 2016



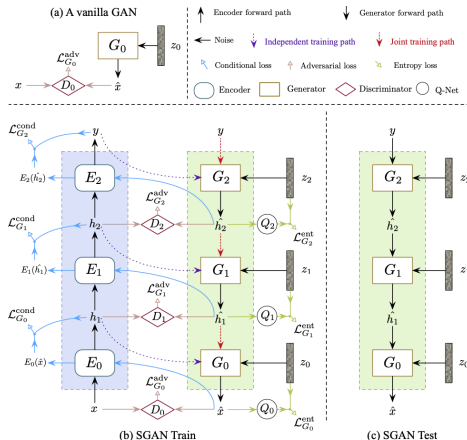
- Transition is smooth
- E.g. 6th row: the TV changed smoothly to the window

Image credit: Alec Radford & Luke Metz & Soumith Chintala 2016



- Similar to word2vec interpolation

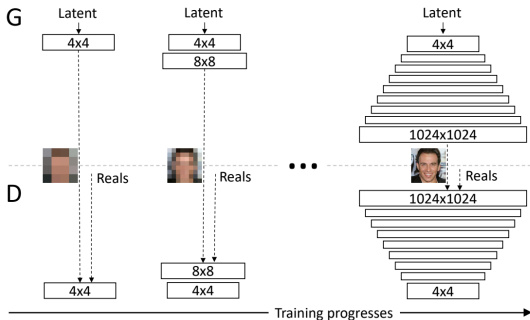
Image credit: Alec Radford & Luke Metz & Soumith Chintala 2016



- Divide and Conquer approach: Big Model = 3 sub-Models
- Train them sequentially (hierarchically)



- Sharp and fine results



- Different approach from SGAN: train the whole model
- But improve the model over time → better resolution



Figure 5: 1024×1024 images generated using the CELEBA-HQ dataset. See Appendix F for a larger set of results, and the accompanying video for latent space interpolations.

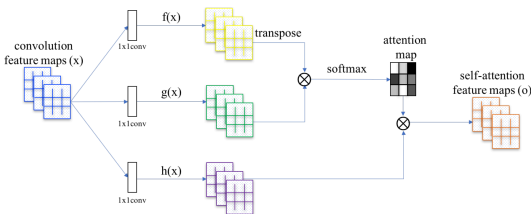
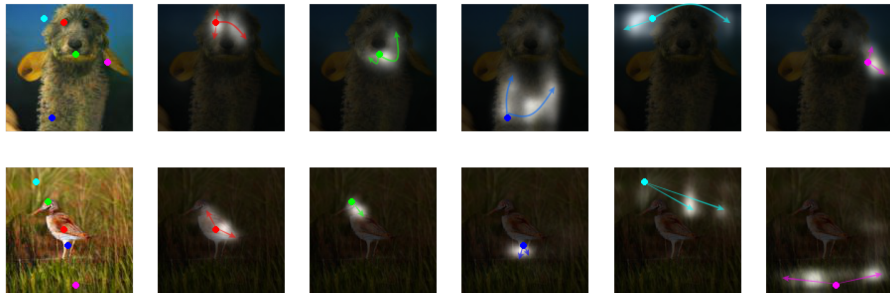


Figure 2: The proposed self-attention mechanism. The $\otimes$ denotes matrix multiplication. The softmax operation is performed on each row.

- Image $x \in \mathbb{R}^{C \times N}$ transformed to $f(x) = W_f x, g(x) = W_g x$
- Attention score: $\beta_{j,i} = \frac{\exp(s_{ij})}{\sum_{k=1}^N \exp(s_{kj})}$ where $s_{kj} = f(x_k)^T g(x_j)$
- Output is $o = (o_1, o_2, \dots, o_N) \in \mathbb{R}^{C \times N}$ where $o_j = \sum_{k=1}^N \beta_{j,i} h(x_i)$, with $h(x_i) = W_h x_i$
- Shapes: $W_g \in \mathbb{R}^{\bar{C} \times C}, W_f \in \mathbb{R}^{\bar{C} \times C}, W_h \in \mathbb{R}^{C \times C}$

Image credit: Han Zhang *et al.* 2018



- Attention and spectral normalization for both D and G
- Use Hinge loss
- Generation leverages cues from all feature locations (instead of local)



Figure 4: 128×128 examples randomly generated by the baseline model and our models “SN on G/D ” and “SN on $G/D+TTUR$ ”.

- Mark a milestone: able to generate full ImageNet samples

- Recall the problem: Zero-sum game loss is nice in theory, but unfortunately not (yet) in practice for training GAN
- Possible solutions:
 - Make the loss non-zero-sum
 - Use Wasserstein distance (WGAN)
 - Use other losses (more later)

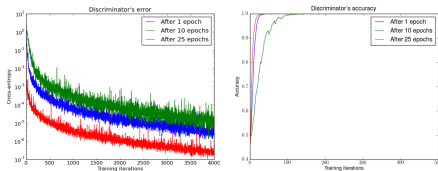


Figure 1: First, we trained a DCGAN for 1, 10 and 25 epochs. Then, with the generator fixed we train a discriminator from scratch. We see the error quickly going to 0, even with very few iterations on the discriminator. This even happens after 25 epochs of the DCGAN, when the samples are remarkably good and the supports are likely to intersect, pointing to the non-continuity of the distributions. Note the logarithmic scale. For illustration purposes we also show the accuracy of the discriminator, which goes to 1 in sometimes less than 50 iterations. This is 1 even for numerical precision, and the numbers are running averages, pointing towards even faster convergence.

- Recall for G_ψ : $\nabla_\psi = \frac{1}{m} \sum_{i=1}^m [\log(1 - D_\theta(G_\psi(z^{(i)})))]$ tends to be zero at beginning $\rightarrow G$ learns nothing
- Now for D_θ :
 - When it gets bad, it's even worse: G_ψ learns incorrectly
 - When it's perfect, G_ψ has no useful feedback to correct itself
 - When it improves, D's loss will go to zero quickly

Image credit: Martin Arjovsky and Leon Bottou 2017

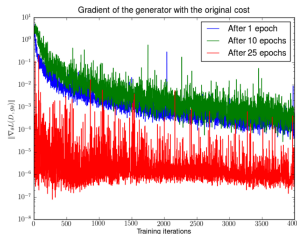


Figure 2: First, we trained a DCGAN for 1, 10 and 25 epochs. Then, with the generator fixed we train a discriminator from scratch and measure the gradients with the original cost function. We see the gradient norms decay quickly, in the best case 5 orders of magnitude after 4000 discriminator iterations. Note the logarithmic scale.

- Now for D_θ :
 - When it's perfect, G_ψ has no useful feedback to correct itself
 - And its gradients will also go to zero quickly $\rightarrow$ learns slowly to learns nothing
 - When it improves, D's loss will go to zero quickly,
 - and G's gradients will go to zeros quickly as well! Very bad!

Image credit: Martin Arjovsky and Leon Bottou 2017

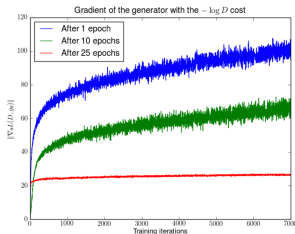
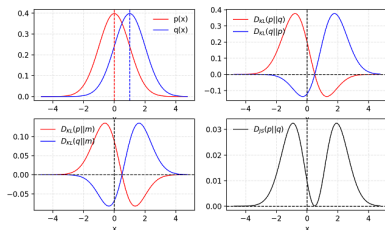
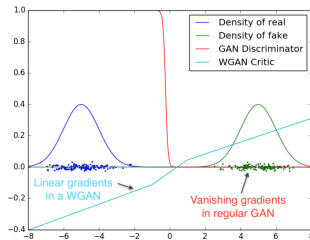


Figure 3: First, we trained a DCGAN for 1, 10 and 25 epochs. Then, with the generator fixed we train a discriminator from scratch and measure the gradients with the $-\log D$ cost function. We see the gradient norms grow quickly. Furthermore, the noise in the curves shows that the variance of the gradients is also increasing. All these gradients lead to updates that lower sample quality notoriously.

- Change $\log(1 - D_{\theta}(G_{\psi}(z^{(i)})))$ to $-\log(D_{\theta}(G_{\psi}(z^{(i)})))$
- A theory backed-up solution
- Recall: don't be paralyzed by losses visualization $\rightarrow$ don't do early stopping.

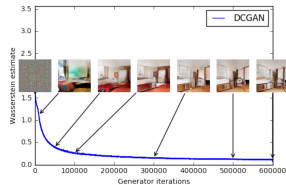
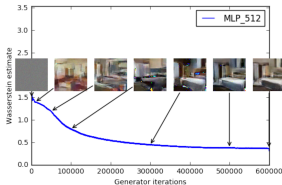
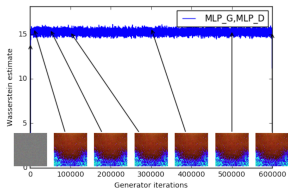


- It minimizes $D_{JS}(p||q) = \frac{1}{2}D_{KL}(p||\frac{p+q}{2}) + \frac{1}{2}D_{KL}(q||\frac{p+q}{2})$
- It's symmetric, but recall from VAE lectures: only Reverse KL forces the learning of the appx distribution q close to p
- D_{JS} and D_{KL} has limitation: it considers similar disjointed distributions different. E.g. $\mathbf{N}(\mathbf{1}, \mathbf{I})$ and $\mathbf{N}(-\mathbf{1}, \mathbf{I})$



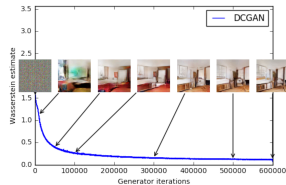
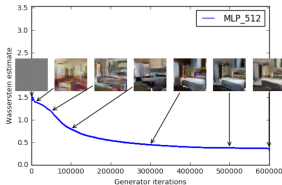
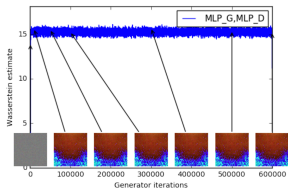
- $W(p, q) = \inf_{\gamma \in \Pi(p, q)} \mathbb{E}_{(x, y) \sim \gamma} [\|x - y\|]$ where $\Pi(p, q)$: set of all joint distribution $\gamma(x, y)$ whose marginals are p and q respectively
- Intuition: cost to move the “mass” to transform/morph q to p
- Borrowed from Optimal Transport, *a.k.a* Earth Moving distance, well-backed theory

WGAN reflects training progress



- Loss does not decrease when training is not good
- And decreases when results are better and better

WGAN reflects training progress



- Loss does not decrease when training is not good
- And increases when results are better and better



Figure 5: Algorithms trained with a DCGAN generator. Left: WGAN algorithm. Right: standard GAN formulation. Both algorithms produce high quality samples.



Figure 6: Algorithms trained with a generator without batch normalization and constant number of filters at every layer (as opposed to duplicating them every time as in [18]). Aside from taking out batch normalization, the number of parameters is therefore reduced by a bit more than an order of magnitude. Left: WGAN algorithm. Right: standard GAN formulation. As we can see the standard GAN failed to learn while the WGAN still was able to produce samples.



Figure 7: Algorithms trained with an MLP generator with 4 layers and 512 units with ReLU nonlinearities. The number of parameters is similar to that of a DCGAN, but it lacks a strong inductive bias for image generation. Left: WGAN algorithm. Right: standard GAN formulation. The WGAN method still was able to produce samples, lower quality than the DCGAN, and of higher quality than the MLP of the standard GAN. Note the significant degree of mode collapse in the GAN MLP.

- G can learn even when D is very good
- No mode collapse in authors' experiments

- GAN: $L_D = \mathbb{E}[\log D(x)] + \mathbb{E}[\log(1 - D(G(z)))]$ and $L_G = \mathbb{E}[\log D(G(z))]$
- LSGAN: $L_D = \mathbb{E}[(D(x) - 1)^2] + \mathbb{E}[D(G(z))^2]$ and $L_G = \mathbb{E}[(D(G(z)) - 1)^2]$
- WGAN: $L_D = \mathbb{E}[D(x)] - \mathbb{E}[D(G(z))]$ and $L_G = \mathbb{E}[D(G(z))]$
- WGAN-GP: $L_D = L_D^{WGAN} + \lambda \mathbb{E}[((|\alpha x - (1 - \alpha G(z))| - 1)^2)]$
- DRAGAN: $L_D = L_D^{GAN} + \lambda \mathbb{E}[((|\alpha x - (1 - \alpha x_p)| - 1)^2)]$ and $L_G = L_{GAN}$
- None of them really outperforms others¹

¹<https://arxiv.org/pdf/1711.10337.pdf>

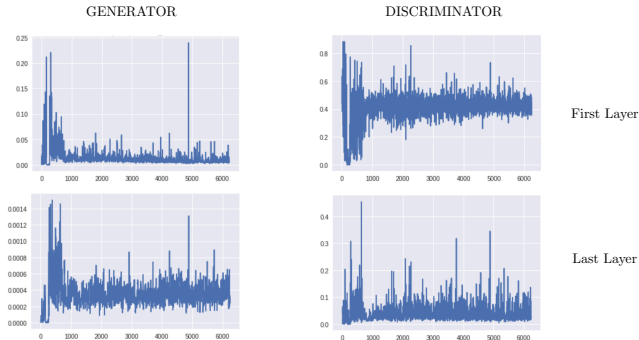
- Already mentioned:
 - Inject noise to input/noise
 - Ball-like noise space (e.g. Gaussian)
 - Noise space normalization
 - Label smoothing for D
- Monitor gradients
- Flip the labels (certainly, for D)
- Focus on one mode at a time
- Dropout in training and *testing* for G
- It's unsupervised, but if you have labels, exploit them
- SGD for D, Adam for G
- Historical Averaging $\|\theta - \frac{1}{t}\theta_i\|^2$
- History replay (from DeepRL): D optimizes many G states
- Leaky ReLU, AVG Pooling, avoid sparse ops: ReLU, Max Pooling

Credit: Tim Salimans *et al.* 2016, Soumith Chintala, Johanthan Hui and ++

Monitor (Vanishing) Gradients

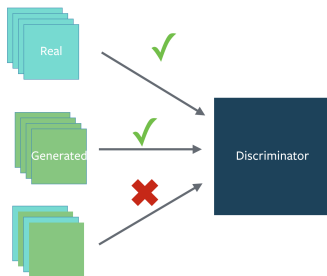


- Monitor gradients of each layer for *both* D and G
- Check gradients norm: should not be > 100



- Inject noise for training input and smooth D's labels
- Flip the labels (similar to injecting noise): fake = 1 and real = 0
- Don't try to find a good schedule to uncollapse training. If so, have a principle, e.g. no update D if $\text{loss}(D) < c$
- Or sometimes, training D more than G

Image Credit: Utkarsh Desai (Medium.com)



- Label-based batch normalization
- Do single-label minibatches, don't mix
- Instance-based normalization: $x = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}}$

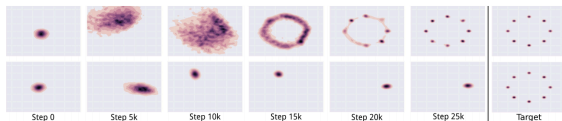


Figure 2: Unrolling the discriminator stabilizes GAN training on a toy 2D mixture of Gaussians dataset. Columns show a heatmap of the generator distribution after increasing numbers of training steps. The final column shows the data distribution. The top row shows training for a GAN with 10 unrolling steps. Its generator quickly spreads out and converges to the target distribution. The bottom row shows standard GAN training. The generator rotates through the modes of the data distribution. It never converges to a fixed distribution, and only ever assigns significant probability mass to a single data mode at once.

- Train one mode well
- Unroll K steps for updating G 's gradients
- D gets updated constantly
- Mode collapse is yet a very hard problem!

- ① GAN: Review
- ② Implementing GAN: A case of MNIST
- ③ (Some) Training Problems
- ④ (Some) Work to Improve Training
- ⑤ GAN Evaluation

- Recall: Optimization in GAN is 2 players try to fool each other, not directly towards the ground truths
- Human is still the most credible one.
- But it's qualitative and subjective.
- Quantitative metrics:
 - Inception Score (IS) (Salimans *et al* 2016)
 - Frechet Inception Distance (FID) (Heusel *et al.* 2017)
 - Precision, Recall and F1

Samples							
Model	Real data	Our methods	-VBN+BN	-L+HA	-LS	-L	-MBF
Score $\pm$ std.	11.24 $\pm$.12	8.09 $\pm$.07	7.54 $\pm$.07	6.86 $\pm$.06	6.83 $\pm$.06	4.36 $\pm$.04	3.87 $\pm$.03

- Use InceptionNet to classify generated images $x \sim p(x)$
- There are 1000 classes (ImageNet)
- High quality generated image: $p(y|x)$ has low entropy
- Generated images are diverse: $p(y) = \int_x p(y|x)dx$ has high entropy
- $\rightarrow$ their distance should be *high* for a good model:

$$IS = \exp(\mathbb{E}_{x \sim p(x)} [\mathbf{D}_{KL}[p(y|x) \| p(y)]]) = \exp(H(y) - H(Y|x))$$

- Downsides:
 - Costly, needs a lot of samples to measure variety
 - Unable to cover edge case: 1 image per class

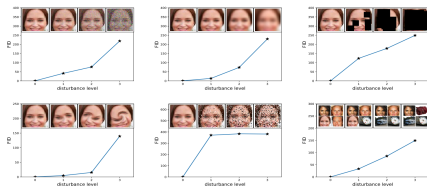


Figure 3: FID is evaluated for **upper left:** Gaussian noise, **upper middle:** Gaussian blur, **upper right:** implanted black rectangles, **lower left:** swirled images, **lower middle:** salt and pepper noise, and **lower right:** CelebA dataset contaminated by ImageNet images. The disturbance level rises from zero and increases to the highest level. The FID captures the disturbance level very well by monotonically increasing.

- Take Pool3 features from InceptionNet
- Assume real distribution $\mathbf{N}(\mu, \Sigma)$ and fake $\mathbf{N}(\mu_{\mathbf{w}}, \Sigma_{\mathbf{w}})$
- Their distance should be *low* for a good model:
$$FID = \|\mu - \mu_{\mathbf{w}}\|_2^2 + Tr(\Sigma + \Sigma_{\mathbf{w}} - 2(\Sigma \Sigma_{\mathbf{w}})^{1/2})$$
- Downsides:
 - Still relies on feature extraction and has expensive cost
 - Not robust against mode collapse

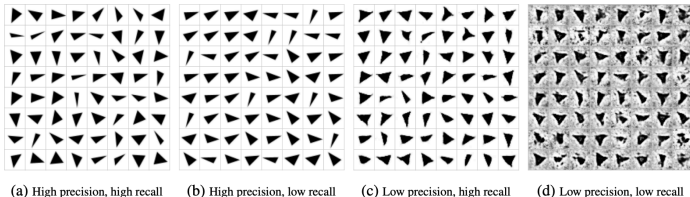


Figure 2: Samples from models with (a) high recall and precision, (b) high precision, but low recall (lacking in diversity), (c) low precision, but high recall (can decently reproduce triangles, but fails to capture convexity), and (d) low precision and low recall.

- P, R and F1 are classical metrics
- IS captures P only, FID captures both P and R
- Those metrics are complementary to IS and FID