Generative Data Modeling with Networks Based on Adversarial Learning and Denoising Diffusion

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Tuesday 26$^{\text{th}}$ March, 2024 14:52

©2024 Avinash Kak, Purdue University

Preamble

To TOC To HowTo

When you create a probabilistic model for your data, you acquire the power to generate new samples of the data from the model. Depending on how good a job you did of modeling the data, the new samples you generate from the model may look deceptively similar to those in your data without being exactly the same as any one of them.

In general, probabilistic modeling may involve fitting a parametric form to the data, the choice of the form based on your understanding of the phenomenon that produced the data. Obviously, you would want to choose the parameters that can account for all of the observed data in a maximum-likelihood sense.

It may also happen that you are really NOT interested in fitting a parametric model to your data, but you are interested in generating new samples from the data nevertheless. In such cases, it is possible you could get away with just constructing a multi-dimensional histogram from the data and using a generator of some sort that would spit out new samples according to that histogram.

Preamble

Regardless of whether you have an analytic model for the data or just a good-quality histogram, **generating new samples is not easy**. It has been the subject of much research by probability theorists and statisticians the last several decades. The best techniques fall under the label Markov-Chain Monte-Carlo (MCMC) sampling and the most commonly used algorithm for MCMC sampling is the Metropolis-Hastings algorithm.

The basic intuition in these algorithms is based on conducting a random walk through the space in which the model is defined and subjecting each successive randomly generated sample to an acceptance test that is based on the model probability distribution. As you generate a candidate for the next sample at your current point on the walk, you subject the acceptance of the candidate to the ratio of the probabilities at the candidate point and the current point. In this manner, you bias the acceptance of a candidate sample in such a way that you end up with more samples in those portions of the model space where the probabilities are relatively high. The generation of the new samples is according to what is known as a proposal distribution. Since the acceptance of each sample is predicated on just the previous sample that was already accepted, we obviously have a Markov Chain. Hence the name MCMC for such algorithms.

Preamble (contd.)

The following link is to a Perl module I created several years ago for helping generate positive and negative training samples for a machine learning algorithm using the Metropolis-Hastings algorithm for sample selection:

https://metacpan.org/pod/Algorithm::RandomPointGenerator

The machine learning program in this case was for classifying land-cover data obtained from wide-area satellite imagery as described in

https://engineering.purdue.edu/RVL/Publications/CVIU_2016_Chang_Comandur_Park_Kak.pdf

Fast forward to deep learning: Just as it has demolished so many of our previous approaches to solving data engineering problems, probabilistic modeling of data has suffered the same fate. The deep learning based approaches to data modeling produce stunning results that nobody could have even dared dream just a few years back. I am sure you have heard about what media refers to as "deep fakes". That's what I am talking about. My goal in this lecture is to introduce you to deep learning based approaches to probabilistic data modeling with neural networks.

Preamble (contd.)

Deep learning has given two fundamentally different approaches to generative data modeling: (1) those that are based on Adversarial Learning; and (2) those that are based on Denoising Diffusion.

Adversarial Learning based approaches for data modeling began with the 2014 publication "*Generative Adversarial Nets*" by Goodfellow, et al.:

`https://arxiv.org/pdf/1406.2661.pdf`

And the **Denoising Diffusion** based approaches for data modeling came into prominence with the following three publications, the first entitled "*Deep Unsupervised Learning using Nonequilibrium Thermodynamics*" by Sohl-Dickstein et al. in 2015:

`https://arxiv.org/pdf/1503.03585.pdf`

the second entitled "*Denoising Diffusion Probabilistic Models*" by Ho et al. in 2020:

`https://arxiv.org/pdf/2006.11239.pdf`

and the third entitled "*Improved Denoising Diffusion Probabilistic Models*" by Nichol and Dhariwal in 2021:

`https://arxiv.org/pdf/2102.09672.pdf`

Preamble (contd.)

In both approaches to generative data modeling, the main idea is to learn the probability distribution that describes a training dataset and subsequently transform a noise vector into an instance of the learned distribution.

With **Adversarial Learning**, you have a Generator network and a Discriminator network. It is the Generator's job to transform a noise-vector into an image that would look like those in the training dataset. And it is the Discriminator's job to not trust the output of the Generator. Through the training iterations, the Generator tries to continually improve its ability to fool the Discriminator and, at the same, the Discriminator attempts to become better and better at telling the difference the real images in the training dataset and the so-called fakes produced by the Generator.

The data modeling approach with **Denoising Diffusion** is entirely different. It is best understood in terms of two Markov processes: (1) You have a diffusion process in which we add a bit of noise to a training image at each timestep until what you get is isotropic Gaussian noise; and, (2) You have a denoising process in which you start with zero-mean isotropic Gaussian noise and you remove from it a bit of noise one timestep at a time until what you get is a recognizable image. Learning consists of training a denoising neural network to remove the same amount of noise that was added during the diffusion process for the same timestep transition.

Preamble (contd.)

Essential to understanding both the approaches — those based on Adversarial Learning and those based on Denoising Diffusion — is having a good grasp of what's meant by the 'distance" or the "divergence" between two probability distributions.

For that reason, I'll start this lecture with a brief survey of the more popular distances and divergences between two given distributions.

For any such distance to be useful in a deep learning context, you would want to treat it as a loss for the backpropagation needed for updating the parameters θ_d and θ_g that I defined previously. That places an important constraint on what kinds of distances can actually be used a deep learning algorithm: the distance must be differentiable so that we can calculate the gradients of the loss with respect to the network parameters.

Over the years, for Adversarial Learning, the Wasserstein distance has emerged as a strong candidate for such a differentiable distance function. And that has led to a Generative Adversarial Network named WGAN that was presented by Arjovsky, Chintala, and Bottou in the following 2017 publication:

https://arxiv.org/pdf/1701.07875.pdf

Preamble (contd.)

This lecture can be divided roughly into three parts:

Part 1 deals with the fundamental ideas related to measuring the distances and the divergences between two probability distributions. This material is on Slides 11 through 51.

Part II deals with Adversarial Learning and it is on Slides 53 through 126.

Part III presents Denoising Diffusion. The material after Slide 126 deals with this topic.

Preamble – How to Learn from These Slides

Since it is a large slide deck, you may need some help with how to digest all the information that is presented here.

To that end, of the fundamental concepts covered in this lecture, you should focus on just the following three at your first reading:

- Fundamental to data modeling with adversarial learning and diffusion is understanding how to measure the distance (or the divergence) between two probability distributions, with one distribution representing the data you want to model and the other the "fakes" you would like to generate. At your first reading, it would be sufficient if you focus on understanding just the concept of KL-Divergence that's presented on slides 17 through 22.
- At your first reading, from the material I have presented on Adversarial Learning, try to just understand the architecture of DCGAN that is presented in Section 7. That's around 20 slides.
- In addition, your first-reading focus should be on just the fundamentals of Denoising Diffusion as described in Sections 11 and 12. That's a total of 25 slides.

That makes for a total of just 50 slides you need to focus on at the beginning. Only after you have understood the material in these 50 slides, you should take the time to look over what's in the rest of the slides. Most of that material covers a few additional fundamental measures of distances between probability distributions, network details, the dataset attributes, the results, etc.

Outline

Outline

Estimating the Distance Between Two Distributions

- Given two probability distributions, p_{data} and p_g, the former representing the training data and the latter an approximation to the former as learned by some machine-learning framework, the question is: As a measure of the dissimilarity of the two distributions, what is the distance between the two?

- Along the lines of a review of such distances that was presented in

  `https://arxiv.org/pdf/1701.07875.pdf`

  let's briefly review the following popular distances and divergences between a pair of probability distributions:

  - **Total Variation Distance**
  - **Kullback-Liebler Divergence**
  - **Jensen-Shannon Divergence**
  - **Earth Mover's Distance**
  - **Wasserstein Distance**

Outline

Total Variation (TV) Distance

- We start with a continuous random variable $\{X \mid x \in R^n\}$ and consider two different probability distributions (densities, really), denoted f and g, over X. The Total Variation (TV) distance between f and g is given by

$$d_{TV}(f, g) \;=\; \sup_A \left[\left| \int_A f(x)dx - \int_A g(x)dx \right| \quad : \quad A \subset R^n \right] \tag{1}$$

- What that says is that we check every subset A of the domain R^n and find the difference between the probability masses over that subset for the f and g densities. The largest value for this difference is the TV distance between the two.

- The important thing here is that the TV distance is a *metric*, in the sense that it satisfies all the conditions for a distance measure to be a metric: Must never be negative; must be symmetric; and must obey the triangle inequality.

TV for the Discrete Case

- Let's now consider the case when the random variable X is discretized. That is, the observed values for X are confined to the set shown below:

$$X = \{x_1, x_2, \ldots.., x_N\}$$

- We are now interested in the distance between two discrete probability distributions, to be denoted P and Q, over a countable set. These distributions must obviously satisfy the unit summation condition:

$$\sum_{i=1}^{N} P(x_i) = 1 \qquad \sum_{i=1}^{N} Q(x_i) = 1 \tag{2}$$

- In this case, the Total Variation distance is given by:

$$d_{TV}(P, Q) = \sup_{A} \left[\left| \sum_{x_i \in A} P(x_i) - \sum_{x_i \in A} Q(x_i) \right| \quad : \quad A \subset X \right] \tag{3}$$

TV for the Discrete Case (contd.)

- Let's now consider the following two subsets of the set X:

$$\begin{aligned} A_1 &= \{x_i \in X \mid P(x_i) \geq Q(x_i\} \\ A_2 &= \{x_i \in X \mid Q(x_i) < P(x_i\} \end{aligned} \tag{4}$$

- On account of the absolute value operator in Eq. (3), for the optimizing set A, it must either be the case that $P(x_i) \geq Q(x_i)$ or that $Q(x_i) \geq P(x_i)$. What that implies that both A_1 and A_2 are a part of the optimizing set A. However, since $A_1 \cup A_2 = X$, we can write for the discretized case:

$$\begin{aligned} d_{TV}(P, Q) &= \frac{1}{2} \sum_{x_i \in X} |P(x_i) - Q(x_i)| \\ &= \frac{1}{2} L_1(P, Q) \end{aligned} \tag{5}$$

where the L_1 norm is the Minkowski norm L_p with $p = 1$.

Kullback-Liebler (KL) Divergence

- Popularly known as KL-Divergence.

- In this case, let's start directly with the discrete case of a random variable X as stated in the first two bullets on Slide 14. The KL-Divergence between a true distribution P and its approximating distribution Q is given by

$$d_{KL}(P, Q) \quad = \quad \sum_{i=1}^{N} P(x_i) \log \frac{P(x_i)}{Q(x_i)} \tag{6}$$

- $d_{KL}(P, Q)$ is obviously the expectation of the ratios $\log \frac{P(x_i)}{Q(x_i)}$ with respect to the P distribution. For the ratios to be defined you must have $Q(x_i) > 0$ when $P(x_i) > 0$. $Q(x_i)$ is allowed to be zero when $P(x_i)$ is zero since $x \log x \to 0$ as $x \to 0+$.

- The logarithm shown above is taken to base 2 if the value of the divergence is required in bits. For natural logarithms, the value returned by KL Divergence is in nats.

KL-Divergence (contd.)

- Since, in general, $\log x$ can return negative and positive values as x increases from 0 to $+\infty$, and since a negative value for KL-divergence makes no sense, how can we be sure that the value of $d_{KL}(P, Q)$ is always non-negative?

- To see that the formula for $d_{KL}(P, Q)$ always returns a non-negative value, we first subject that formula to the following rewrites:

$$\begin{aligned} d_{KL}(P, Q) &= \sum_{i=1}^{N} P(x_i) \log \frac{P(x_i)}{Q(x_i)} \\ &= -\sum_{i=1}^{N} P(x_i) \log \frac{Q(x_i)}{P(x_i)} \\ &= -\sum_{i=1}^{N} P(x_i) \log \frac{P(x_i) + Q(x_i) - P(x_i)}{P(x_i)} \\ &= -\sum_{i=1}^{N} P(x_i) \log \left[1 + \frac{Q(x_i) - P(x_i)}{P(x_i)} \right] \\ &= -\sum_{i=1}^{N} P(x_i) \log(1 + a) \end{aligned} \tag{7}$$

KL-Divergence (contd.)

- In the last equation on the previous slide, $a = \frac{Q(x_i) - P(x_i)}{P(x_i)}$. The factor a is lower bounded by -1, which happens when $P(x_i)$ takes on the largest possible value of 1 and $Q(x_i)$ takes on the smallest possible value of 0.

- Using Jensen's inequality to take advantage of the concavity of $\log x$ over the interval $(0, \infty)$, one can show that for all $a > -1$, $\log(1 + a) \leq a$. The derivation on the previous slide can therefore be extended as follows:

$$
\begin{aligned}
d_{KL}(P, Q) \quad & \geq \quad -\sum_{i=1}^{N} P(x_i) \frac{Q(x_i) - P(x_i)}{P(x_i)} \\
& = \quad -\sum_{i=1}^{N} [Q(x_i) - P(x_i)] \\
& = \quad 0
\end{aligned} \tag{8}
$$

which implies that we are guaranteed that $d_{KL}(P, Q) \geq 0$.

KL-Divergence (contd.)

- KL-Divergence CANNOT be a *metric distance*, not the least because what it calculates is asymmetric with respect to its two args.

- Given its limitations — requiring $Q(x) > 0$ when $P(x) > 0$ and not being a metric distance — **students frequently want to know as to why KL-Divergence is as "famous" as it is in the estimation-theoretic literature**. One reason for that is its interpretation as relative entropy:

$$d_{KL}(P, Q) \quad = \quad H_P(Q) \quad - \quad H(P) \tag{9}$$

  which follows straightforwardly from the definition in Eq. (6). $H(P)$ is the entropy associated with the probability distribution P and $H_P(Q)$ the cross-entropy of an approximating distribution Q vis-a-vis the true distribution P. [See the definitions for $H(P)$ and $H_P(Q)$ on the next slide.]

- Since $d_{KL}(P, Q) \geq 0$, it must be the case that $H_P(Q) \geq H(P)$, **which constitutes a proof of the assertion made on Slide 17 of my Week 7 lecture that the smallest possible value for $H_P(Q)$ is $H(P)$.**

KL-Divergence (contd.)

- Whereas the entropy associated with a distribution P is defined as $H(P) = -\sum_{i=1}^{N} P(x_i) \log P(x_i)$, the cross-entropy of an approximate distribution Q with respect to a true distribution P is given by $H_P(Q) = -\sum_{i=1}^{N} P(x_i) \log Q(x_i)$.

  [**Entropy based interpretations of uncertainty are valuable for developing powerful algorithms for data engineering. See Sections 2 through 4 of my Decision Trees tutorial at the clickable link** `https://engineering.purdue.edu/kak/Tutorials/DecisionTreeClassifiers.pdf`.]

- The entropy based definition of KL-Divergence in Eq. (9) on the previous slide implies that the divergence is a measure of the uncertainty in the estimated distribution Q over and above what it is in the original distribution P. [See Slides 20 and 21 of my Week 7 lecture for why the entropy is a measure of uncertainty.]

- Understanding KL-Divergence is a stepping stone to learning the Jensen-Shannon divergence (and the closely related Jensen-Shannon distance) that I present starting with the next section.

KL-Divergence (contd.)

- In Python, a call like:

```
import scipy.stats
scipy.stats.entropy(P,Q)
```

with P and Q standing for two normalized (or unnormalized) histograms, returns the KL-Divergence of Q vis-a-vis P. If $Q(x)$ is zero where $P(x)$ is not, it will throw an exception.

- In the calls shown above, the two histogram arrays must be of equal length. You can specify the base of the logarithm with an optional 3^{rd} argument. The default for the base is e for the natural log.

- Finally, note that a commonly used notation for KL-Divergence as defined in Eq. (6) on Slide 17 is

$$d_{KL}(P \,||\, Q) \quad = \quad \sum_{i=1}^{N} P(x_i) \log \frac{P(x_i)}{Q(x_i)} \tag{10}$$

where you place two vertical bars between the two arguments of the function name on the left hand side of the equality sign.

Jensen-Shannon (JS) Divergence and Distance

- We again have a random variable X whose observed samples belong to the set:

$$X \quad = \quad \{x_1, x_2, ..., x_N\} \tag{11}$$

- And, as for the case of KL-Divergence, we consider a true probability distribution P and its approximation Q over the values taken on by the random variable. The Jensen-Shannon divergence, defined below, is a symmetrisized version of the KL-Divergence presented earlier in Eq. (6):

$$d_{JS}(P, Q) \quad = \quad d_{KL}(P, M) \ + \ d_{KL}(Q, M) \tag{12}$$

  where M is the mean distribution for P and Q, as given by

$$M \quad = \quad \frac{P + Q}{2} \tag{13}$$

- We can also talk about Jensen-Shannon *distance*, which is given by the square-root of the Jensen-Shannon Divergence:

$$dist_{JS}(P, Q) \quad = \quad \sqrt{d_{JS}(P, Q)} \tag{14}$$

JS Divergence and Distance (contd.)

- Both the divergence $d_{JS}(P, Q)$ and the distance $dist_{JS}(P, Q)$ are symmetric with respect to the arguments P and Q. Additionally, they do away with the "$Q(x) > 0$ when $P(x) > 0$" requirement of KL-Divergence.

- Since, as established earlier in these slides, the KL Divergence is always non-negative, the JS-Divergence is also non-negative.

- The value of $d_{JS}(P, Q)$ is always a real number in the closed interval $[0, 1]$. When the value is 0, the two distributions P and Q are identical. And when the value is 1, the two distributions are as different as they can possibly be.

- **Most significantly, $dist_{JS}(P, Q)$ is a valid metric distance.**

JS Divergence and Distance (contd.)

- Given two histogram arrays P and Q of equal length, normalized or unnormalized, a call like the following in Python

```
from scipy.spatial import distance
distance.jensenshannon(P,Q)
```

  directly returns the Jensen-Shannon *distance* between the two histograms. If you wanted the Jensen-Shannon *divergence*, you would need to square the answer returned. The function call implicitly normalizes the histogram arrays if you supply them otherwise.

- With regard to the role of the Jensen-Shannon divergence (and, therefore, also of the KL-Divergence) in the context of this lecture, **the authors Goodfellow et al. of "Generative Adversarial Nets" have argued that if the Discriminator in a GAN is trained to its optimum, the distribution learned by the Generator is guaranteed to be the one whose Jensen-Shannon divergence from the training-data distribution is minimized.**

Outline

Earth Mover's Distance

- **The distance function that the DL community is all excited about at the moment is the Wasserstein Distance.** The reason has to with the fact this is the only differentiable distance function and, because it is differentiable, a loss based on this distance function can be backpropagated directly for updating the weights in a network.

- However, in order to fully appreciate what exactly is measured by the Wasserstein Distance, you first have to understand what is known as the Earth Mover's Distance (EMD). **Note that many researchers use the two names interchangeably. I personally think of the Wasserstein Distance as a stochastic version of EMD.**

- My goal in this section is to introduce you to EMD. My intro to EMD is based on the following classic paper by Rubner, Tomasi, and Guibas:

`http://robotics.stanford.edu/~rubner/papers/rubnerIjcv00.pdf`

Earth Mover's Distance (contd.)

- To appreciate EMD, consider establishing similarity between two images on the basis of the histograms of their graylevels.

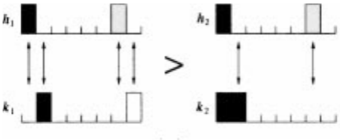
- Given two N-bin histograms f and g for the two images, you would not be too far off the mark if the first idea that pops up in your head would be to carry out a bin-by-bin comparison using a distance like:

$$d_{L_r}(f, g) = \left(\sum_{i=1}^{N} |g_i - h_i|^r \right)^{\frac{1}{r}} \tag{15}$$

- With $r = 1$, you'd be computing the L_1 distance between the two histograms, and with $r = 2$ the Euclidean distance. You will see both being used rather commonly, but you have to be careful as you will soon see. As mentioned on Slide 15, the general form of the distance shown above is known as the Minkowski distance.

Earth Mover's Distance (contd.)

- **That a distance function of the sort shown on the previous slide might give nonsensical answers for image similarity** is made beautifully clear by the following example from the Rubner et el. paper:



Comparing histograms

- In the figure shown above, first focus on the (h_1, k_1) histograms shown in the left column. The h_1 image has half its pixels very dark and the other half of the pixels very white. Perceptually, the k_1 image is going to look very similar to the h_1 image since the two dominant gray levels are merely shifted to the right by one unit. If the number of bins is, say, greater than 64, you will not even notice the shift.

Earth Mover's Distance (contd.)

- Next, focus on the (h_2, k_2) histograms in the figure on the previous slide. While the h_2 image has half its pixels very dark and the other half very white, the k_2 image contains only dark pixels.

- Therefore, to a human observer, the two images in the (h_1, k_1) pair will look very similar, while the two images in the (h_2, k_2) pair will look very different. **However, the d_{L_r} distance in Eq. (15) will give you exactly the opposite answer.**

- **Since distances like d_{L_r} in Eq. (15) cannot be trusted to yield meaningful results when comparing histograms for image similarity, EMD has emerged as a powerful alternative.**

- EMD is based on associating a cost with moving pixels from one bin to another in a hypothetical attempt that tries to make the two histograms as similar looking as possible, constructing an overall cost with all such pixel transfers, and then minimizing the overall cost.

Earth Mover's Distance (contd.)

- Consider the following as an example of the cost associated with moving a pixel from one bin to another in a one-dimensional grayscale histogram whose bins are one-unit wide:

$$c_{ij} \;=\; 1 \;-\; e^{-\alpha|i-j|} \tag{16}$$

  where you can think of $\alpha > 0$ as a heuristic parameter that is approximately proportional to the overall variability in the bin populations. It was shown by Rubner et al. that such a cost function is a metric. What it says is that cost of moving pixels from a bin to another close-by bins is close to zero. However, the costs go up if the transfer is between more widely separated bins.

- **The problem of comparing two histograms can now be stated as an instance of the classic "transportation simplex" problem in optimal transport theory for resource distribution, as explained on the next slide.**

Earth Mover's Distance (contd.)

- You have M providers of some resource who possess different quantities $(\{g_i | i = 1, \ldots, M\})$ of the resource and N consumers of the same resource whose needs vary according to $(\{h_j | j = 1, \ldots, N\})$.

- And you also have a cost estimate c_{ij} that is the cost of transporting a unit of the resource from the i^{th} provider to the j^{th} consumer.

- Our goal is to come up with with an optimum flow matrix F, whose f_{ij} element tells us how much of the resource to transport from the i^{th} provider to the j^{th} consumer. We must obviously solve the following minimization problem for F:

$$\min_F \sum_{i=1}^{M} \sum_{j=1}^{N} c_{ij} f_{ij} \tag{17}$$

  with the minimization subject to the constraints shown on the next slide.

Earth Mover's Distance (contd.)

- The minimization problem on the previous slide must be solved subject to the constraints:

$$f_{ij} \geq 0 \qquad i = 1, \ldots, M, \quad j = 1, \ldots, N \tag{18}$$

$$\sum_{j=1}^{N} f_{ij} \leq h_i \qquad i = 1, \ldots, M \tag{19}$$

$$\sum_{i=1}^{M} f_{ij} \leq g_j \qquad j = 1, \ldots, N \tag{20}$$

$$\sum_{i=1}^{M} \sum_{j=1}^{N} f_{ij} = min \left\{ \sum_{i=1}^{M} g_i, \quad \sum_{j=1}^{N} h_j \right\} \tag{21}$$

- All four constraints are straightforward because they are so intuitive. [The constraints in Eqs. (18) and (19) are straightforward: The flow can never be negative and the total outgoing flow from a provider cannot exceed what the provider has in stock. The constraint in Eq. (20) also makes sense since the accumulated in-flows for the j^{th} consumer should not exceed to total demand for that consumer. The constraint in Eq. (21) is important only when the total supply provided by all the providers is not equal to the total demand at all the consumers. Should there be such a disparity between total supply and total demand, summing all of elements of the flow matrix should not exceed the smaller of the total supply and the total demand.]

Earth Mover's Distance (contd.)

- Having calculated the optimal transport by solving the minimization problem described on the previous two slides, we use the following formula to compute the EMD between the suppliers distribution for the resource and the consumers distribution:

$$EMD(g,h) = \frac{\sum_{i=1}^{M}\sum_{j=1}^{N} c_{ij} f_{ij}}{\sum_{i=1}^{M}\sum_{j=1}^{N} f_{ij}} \tag{22}$$

  where we normalize the cost of the optimal transport of the goods by the total amount of the goods transported.

- Such optimization problems have received much attention by the OR (Operations Research) folks over the last several decades. We now have polynomial-time solutions for the problem that fall under the general category of "simplex algorithms for linear programming". Rubner et al. used such a solution in their work on retrieval from image databases and showed impressive results.

Earth Mover's Distance (contd.)

- It was shown by Rubner et al. that EMD is a metric when the supplier and the consumer distributions are normalized. For the case of comparing image histograms, we can say that EMD between two histograms is a metric for the case of normalized histograms.

- With that as an intro to EMD, the issue that should come up next would be whether it is possible to create a loss function directly from EMD for adversarial learning. I'll address this question later when I get into the differentiability of the different distance functions.

- For now, let's move on to the Wasserstein distance. **As mentioned earlier, I consider the Wasserstein distance to be a stochastic version of EMD.**

Wasserstein Distance

- Using $d_W(P, Q)$ to denote the Wasserstein distance between the distributions P and Q, here is its definition:

$$d_W(P, Q) \quad = \quad \inf_{\gamma(X,Y) \in \Gamma(P,Q)} E_{(X,Y) \sim \gamma} \Big[\|x \; - \; y\| \Big] \tag{23}$$

- In the above definition, $\Gamma(P, Q)$ is the set of all possible joint distributions $\gamma(X, Y)$ over two random variables X and Y such that the marginal of $\gamma(X, Y)$ with respect to X is P and the marginal of $\gamma(X, Y)$ with respect to Y is Q.

- Since the marginal of $\gamma(X, Y)$ with respect to X is $P(x)$ and the marginal of the same with respect to Y is $Q(x)$, $\gamma(X, Y)$ encodes in it the probability mass that must be shifted from the distribution P to the distribution Q if for whatever reason we wanted them to become identical. [If $\gamma(X, Y)$ encodes in it the probability mass that must be shifted from the distribution P to the distribution Q, is there any way to construct a "cost" — a single number — associated with this transfer of mass? The cost itself is proportional to the absolute difference between the value x for the random variable X and the value y for the random variable Y if the joint distribution $\gamma(X, Y)$ indicates there is a non-zero probability associated with mass transfer from x to y. For vector random variables, this would be the same as the norm $\|x - y\|$. In order to get a single-number cost, we would need to average the norm $\|x - y\|$ as indicated in Eq. (23) above.]

Wasserstein Distance (contd.)

- The $d_W(P, Q)$ distance is a metric as it obeys the constraints on metrics: its values are guaranteed to be non-negative, it is symmetric with respect to its args, and it obeys the triangle inequality. Let's now focus on what it might take to compute the Wasserstein distance.

- The infimum required on the right side of Eq. (23) says that from the set $\Gamma(P, Q)$ of all joint distributions defined in the second bullet on the previous slide, we need to zero in on the joint distribution $\gamma(X, Y)$ that minimizes the mean value of the normed difference $\|x - y\|$ with the sample pair (x, y) drawn from the joint distribution.

- In a computation based on a literal interpretation of the definition in Eq. (23), we are required to carry out a random experiment in which we sample the (infinite) set $\Gamma(P, Q)$ of the joint distributions for the two random variables X and Y for a candidate distribution $\gamma(X, Y)$.

Wasserstein Distance (contd.)

- Subsequently, in another random experiment, we sample the distribution $\gamma(X, Y)$ for specific values x and y for the random variables X and Y. We carry out the second random experiment repeatedly in order to form a good estimate for the average value for $\|x - y\|$. Subsequently, we go back to the first random experiment and choose a second candidate for $\gamma(X, Y)$, and so on. **Such a computation is obviously not feasible.**

- Fortunately, the infimum in the theoretical definition of Wasserstein Distance in Eq. (23) can be converted into a computationally tractable supremum calculated separately over the component distributions P and Q as shown below

$$d_W(P, Q) \quad = \quad \sup_{\|f\|_L \leq 1} \left[E_{x\sim P}\{f(x)\} \; - \; E_{y\sim Q}\{f(y)\} \right] \tag{24}$$

  for ALL 1-Lipschitz functions $f : X \rightarrow R$ where X is the domain from which the elements x and y mentioned above are drawn and R is the set of all reals.

Wasserstein Distance (contd.)

- The result shown in Eq. (24) is from a famous book in Optimal Transport Theory by Cédric Villani:

  https://cedricvillani.org/sites/dev/files/old_images/2012/08/preprint-1.pdf

- Despite the use of "ALL" for the family of 1-Lipschitz functions $f()$ in Eq. (24), **a better way to state the same thing would be that there exists a 1-Lipschitz function $f()$ for which the maximization shown on the right in Eq. (24) yields the value for the Wasserstein distance.**

- **But what is a k-Lipschitz Function?** A function $f : X \rightarrow R$ is a k-Lipschitz function if $|f(x_1) - f(x_2)| \leq k.d(x_1, x_2)$ for every $x_1, x_2 \in X$. Note that X is the domain of the function. In this definition, $d(.,.)$ is the metric distance defined on the domain of f. So $d(x1, x2)$ is the distance between the points x_1 and x_2.

Wasserstein Distance (contd.)

- In general, the Lipschitz functions allow us to prescribe functions with "levels" of continuity properties. The larger the value of the integer k, the more rapidly the function would be allowed to change when you go from a point x_1 to another point x_2 in its domain.

- In general, at all x in the domain X of f:

$$f(x) \quad = \quad \inf_{y \in X} [f(y) + k \cdot d(x,y)] \quad = \quad \sup_{y \in X} [f(y) - k \cdot d(x,y)] \tag{25}$$

- Note that the definition $|f(x) - f(y)| \ \leq \ k \cdot d(x,y)$ implies $f(y) - k.d(x,y) \ \leq \ f(x) \ \leq \ f(y) + k \cdot d(x,y)$ When you apply the definitions of infimum and supremum to these inequalities, you get the form shown in Eq. (25).

Wasserstein Distance (contd.)

- We are faced with the following questions if we want to use the form in Eq. (24) for computing the Wasserstein Loss in adversarial learning:
  - How do we find the function $f()$ that would solve the maximization problem in Eq. (24)?
  - The expectation operator $E()$ in Eq. (24) is meant to be applied over the entire domain of the distributions P and Q. How do we do that in a practical setting?
- I'll address each of these issues separately in Section 12 on how to use the Wasserstein distance for adversarial learning. That material begins on Slide 90.

Outline

A Random Experiment for Studying Differentiability

- The discussion in this section is an elaboration of the "learning parallel lines" example in the paper

  https://arxiv.org/pdf/1701.07875.pdf

- We start with a random variable Z whose values, z, are uniformly distributed over the unit interval $[0, 1]$.

- We assume that the ground-truth consists of Z-values on the y-axis in R^2 — this would presumably be our "training" data (to make an analogy with GAN training). Now imagine a GAN Generator that is also capable of producing the same kind of points in R^2 but the points produced by the Generator are offset horizontally by a learnable parameter θ. The true value of θ is obviously 0, but the Generator has to learn that during training.

- We use X as the random variable to denote the points on the ground-truth line in R^2 and Y to denote the points being produced by the Generator on another vertical line that is horizontally offset by θ.

Studying Differentiability (contd.)

- Let P denote the distribution for the ground-truth points X and Q the distribution for the GAN-generated points Y.

- Note again that the ground-truth points X are the set of all points $\{(0, z) \in R^2 | z \sim U[0, 1]\}$ and the GAN-generated points Y form the set $\{(\theta, z) \in R^2 | z \sim U[0, 1]\}$.

- The following figure illustrates the relationship between X, Y, and the sole learnable parameter θ.

```
       ^
(0,1)  |                    |
       |                    |         R^2 space
       |                    |
       |                    |
       |X                   |Y
       |                    |
       |                    |
       |                    |
       |                    |
(0,0)   ----------------------------------->

       <----- \theta ---->
```

Outline

Differentiability of Distance Functions

- Given the sets X and Y as defined on Slides 45 and 46, **we start with examining the differentiability of the Wasserstein Distance.**

- Given the definition that X is set of all points $\{x = (0, z) \in R^2 | z \sim U[0, 1]\}$ and Y is the set of all points $\{y = (\theta, z) \in R^2 | z \sim U[0, 1]\}$, we can say that the difference $\|x - y\|$ needed for calculating the Wasserstein distance using Eq. (23) on Slide 38 will always be equal to the value of the parameter θ.

- The same would be the case if we used the supremum based estimate of the Wasserstein distance using Eq. (24). Therefore, for the random experiment under consideration, we can claim:

$$d_W(P, Q) \quad = \quad \theta \tag{26}$$

- So we see that the Wasserstein distance is continuous and differentiable with respect to the learnable parameter θ. That makes it a good candidate as a loss function in a neural network.

Differentiability of Distance Functions (contd.)

- What is interesting is that the closely related EMD distance does not possess the property of differentiability with respect to the learnable parameters. That is because it involves comparing histograms directly. Since a histogram is a discretization of continuous values, it is not possible to backpropagate any partial derivatives through such a step.

- **Let's now consider the differentiability of KL-Divergence.**

- The definition of KL-Divergence provided earlier in Eq. (6) is for the case of random variables that take discrete values. But the "parallel lines" example involves two continuous random variables X and Y. Here is the definition of KL-Divergence for the continuous case:

$$d_{KL}(P, Q) \quad = \quad \int P(x) \log \frac{P(x)}{Q(x)} dx \tag{27}$$

- The scope of the variable x of integration is the space of all random outcomes over which both the distributions P and Q are defined.

Differentiability of Distance Functions (contd.)

- The last bullet on the previous implies that x must span both the lines X and Y for this integration. However, the sets X and Y are disjoint except when the Generator parameter θ equals zero.

- When X and Y are disjoint, we run headlong into the condition $Q(x) = 0$ when $P(x) > 0$ that makes the divergence d_{KL} become infinity. Hence we can write:

$$\begin{aligned} d_{KL}(P, Q) &= \quad 0 \qquad \theta = 0 \\ &= \quad +\infty \qquad \theta \neq 0 \end{aligned} \tag{28}$$

- Obviously, KL-Divergence is not differentiable with respect to the learnable parameter θ.

- **Next we take up the case of differentiability of JS-Divergence.**

Differentiability of Distance Functions (contd.)

- The formula for JS-Divergence was presented in Eq. (12) on Slide 24. Given two distributions P and Q, the formula in that equation requires that we first calculate the mean distribution M as defined in Eq. (13).
- For what follows, recall the fact that JS-Divergence is a symmetrization of KL-Divergence that is meant to get around the main shortcoming of the latter in those regions of the probability space where $Q(x) = 0$ whereas $P(x) > 0$.
- Note that M in Eq. (13) is a mixture distribution. By definition, given two separate distributions P and Q defined over the same set of random outcomes, a mixture means merely that the next sample will be drawn randomly either from P or from Q. Since the two component distributions P and Q in the mixture M are weighted equally (by a factor $\frac{1}{2}$), the individual distributions will be selected with equal probability for the realizations of M.
- On the next slide, we will consider the first term in the summation in Eq. (12). The result for the second term would be the same.

Differentiability of Distance Functions (contd.)

- Focusing on the case when the learnable parameter θ is nonzero, that is, when we are going to encounter the condition $Q(x) = 0$ when $P(x) > 0$ (which will happen on line X as explained previously for the case of differentiability of KL-Divergence), let's focus on the first term on the RHS in Eq. (12) on Slide 24:

$$\begin{aligned} d_{KL}(P, M) &= \int P(x) \log \frac{P(x)}{M(x)} dx \\ &= \int P(x) \left[\log P(x) - \log \frac{P(x) + Q(x)}{2} \right] dx \\ &= \int P(x) \left[\log P(x) - \log(P(x) + Q(x)) + \log 2 \right] dx \\ &= \int P(x) \log 2 \; dx \\ &= log 2 \end{aligned} \tag{29}$$

- As expected, the expressions on the RHS of Eq. (12) are now inoculated against going to infinity under the condition $Q(x) = 0$ when $P(x) > 0$.

Differentiability of Distance Functions (contd.)

- Since both the component expressions on the RHS of Eq. (12) lead to exactly the same result that is shown above, we can say that $d_{JS}(P,Q) \;=\; \log 2$ for the case $\theta \neq 0$.
- Therefore, we can write:

$$\begin{aligned} d_{JS}(P,Q) &= 0 \qquad \theta = 0 \\ &= log2 \qquad \theta \neq 0 \end{aligned} \tag{30}$$

  which is again not differentiable with respect to the parameter θ.

- **We next take up the differentiability of the Total Variation Distance**
- The Total Variation (TV) distance for the continuous case was defined in Eq. (1).
- That definition calls for identifying a subset A of the probability space defined by all possible outcomes that maximizes the difference between P's probability mass over A and Q's probability mass over A.

Differentiability of Distance Functions (contd.)

- When $\theta \neq 0$, we could choose for such an A the set X itself. Since the probability mass of P over this set equals 1 whereas the probability mass of Q over the same set equals 0. The difference of the two integrals in Eq. (1) on Slide 13 for such an A is the largest it can be — equal to 1.

- On the other hand, when the Generator's parameter θ equals 0, the sets X and Y become congruent. In this case, the difference of the two integrals in Eq. (1) would be zero.

- So we can write:

$$
\begin{aligned}
d_{TV}(P, Q) &= 0 \qquad \theta = 0 \\
&= 1 \qquad \theta \neq 0
\end{aligned}
\tag{31}
$$

- TV is obviously not a differentiable distance function.

Outline

PurdueShapes5GAN Dataset of Images

- I have created a dataset, `PurdueShapes5GAN`, for experimenting with the three GANs in version 2.0.3 (or higher) of the DLStudio module. **Each image in the dataset is of size 64×64.** The dataset consists of 20,000 images.

- **This dataset of rather small-sized images was created to make it easier to give classroom demonstrations of the training code and also for the students to be able to run the code on their laptops (at least those that come equipped with a GPU for graphics rendering, as many of them do these days).**

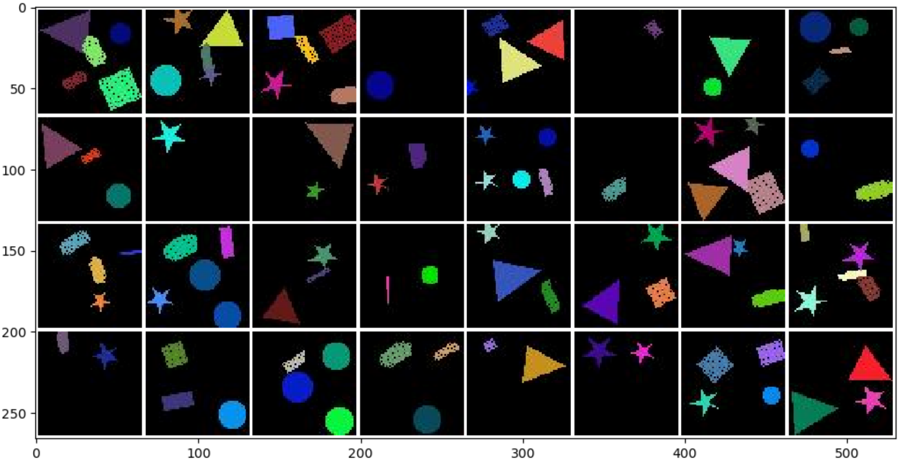
- The program that generates the `PurdueShapes5GAN` dataset is a modification of the script I used for the `PurdueShapes5MultiObject` dataset that I used previously in the lecture on semantic segmentation.

PurdueShapes5GAN Dataset (contd.)

- Compared to its predecessor semantic-segmentation dataset, the annotations that were needed for the semantic segmentation dataset (the bounding boxes and masks) are no longer necessary for adversarial learning of a probabilistic data model for a set of images. That makes a GAN dataset much simpler compared to a semantic-segmentation dataset.

- Each image in the `PurdueShapes5GAN` dataset contains a random number of up to five shapes: rectangle, triangle, disk, oval, and star. Each shape is located randomly in the image, oriented randomly, and assigned a random color. Since the orientation transformation is carried out without bilinear interpolation, it is possible for a shape to acquire holes in it. Shown in the next slide is a batchful of images that is processed in each iteration of the training loop. The batch size is 32.

PurdueShapes5GAN Dataset (contd.)



A batch of images from the PurdueShapes5GAN dataset

About the "Complexity" of the Dataset Images

- I would not be surprised if your first reaction to the dataset images is that they couldn't possibly present a great challenge to a data modeler.
- Shown in the next slide are enlarged views of two of the images on the previous slide. In addition to the sharp shape boundaries, you can also small holes inside the shapes.
- The holes that you see inside the shapes were caused by intentionally suppressing bilinear interpolation as the shapes were randomly reoriented.
- So the challenge for the data modeler would be its ability to not only reproduce the shapes while preserving the sharp edges, but also to incorporate the tiny holes inside the shapes, **and do so with the probabilities that reflect the training data.**

About the "Complexity" of the Images (contd.)

PurdueShapes5GAN Dataset (contd.)

You can download the dataset archive

```
datasets_for_AdversarialNetworks.tar.gz
```

through the link "Download the image dataset for Adversarial Learning" provided at the top of the HTML version of the main webpage for the DLStudio module (version 2.0.3 or higher). You would need to store it in the `ExamplesAdversarialLearning` directory of the distribution. Subsequently, you would need to execute the following command in that directory:

```
tar zxvf datasets_for_AdversarialNetworks.tar.gz
```

This command will create a `dataGAN` subdirectory and deposit the following dataset archive in that subdirectory:

```
PurdueShapes5GAN-20000.tar.gz
```

Now execute the following in the `dataGAN` directory:

```
tar zxvf PurdueShapes5GAN-20000.tar.gz
```

With that, you should be able to execute the adversarial learning based scripts in the `ExamplesAdversarialLearning` directory.

Outline

DCGAN Implementation in DLStudio

- The main goal of this section is to tell you about the implementation of DCGAN in DLStudio's co-class `AdversarialLearning`.
- DCGAN, short for "**Deep Convolutional Generative Adversarial Network**", was presented in a paper that I cited in the Preamble to this lecture.
- However, before actually getting into the DCGAN architecture, I need to take you back to the first paper that started the modern excitement in adversarial learning. I am talking about the 2014 publication "Generative Adversarial Nets" by Goodfellow, Pouget-Abadie, Mirza, Xu, Warde-Farley, Ozair, Courville, and Bengio that was also cited in the Preamble.
- The reason I need to take you back to this paper is because the basic training logic in DCGAN is the same as that proposed in the above cited publication by Goodfellow et al.

Adversarial Learning Requires Generator and Discriminator

- Adversarial learning as described in the Goodfellow et al. paper involves two networks, a Discriminator and a Generator. We can think of the Discriminator as a function $D(x, \theta_d)$ where x is the image and θ_d the weights in the Discriminator network. The $D(x, \theta_d)$ function **returns the probability that the input x is from the probability distribution that describes the training data.**

- Similarly, we can think of the Generator as a function $G(z, \theta_g)$ that maps noise vectors to images that we want to look like the images in our training data. The vector θ_g represents the learnable parameters in the Generator network.

- We assume that the training images are described by some probability distribution that we denote p_{data}. The goal of the Generator is to transform a noise vector, denoted z, into an image that should look like a training image.

Discriminator and Generator (contd.)

- Regarding z, we also assume that the noise vectors z are generated with a probability distribution $p_Z(z)$. Obviously, z is a realization of a vector random variable Z.

- The output of the Generator consists of images that correspond to some probability distribution that we will denote p_G. So you can think of the Generator as a function that transforms the probability distribution p_Z into the distribution p_G.

- The question now is how do we train the Discriminator and the Generator networks.

- The Discriminator is trained to maximize the probability of assigning the correct label to an input image that looks like it came from the same distribution as the training data.

Discriminator Training vs. Generator Training

- That is, for Discriminator training, we want the parameters θ_d to maximize the following expectation:

$$\max_{\theta_d} E_{x \sim p_{data}}[\log D(x)] \tag{32}$$

- The expression $x \sim p_{data}$ means that x was pulled from the distribution p_{data}. In other words, x is one of the training images.

- While we are training D to exhibit the above behavior, we train the Generator for the following minimization:

$$\min_{\theta_g} E_{z \sim p_Z}[\log(1 - D(G(z)))] \tag{33}$$

- Combining the two expressions shown above, we can express the combined optimization as:

$$\min_{\theta_g} \max_{\theta_d} \left[E_{x \sim p_{data}}[\log D(x)] \quad + \quad E_{z \sim p_Z}[\log(1 - D(G(z)))\right] \tag{34}$$

Discriminator Training vs. Generator Training (contd.)

- We'll translate the min-max form in Eq. (34) into a "protocol" for training the two networks.
- For each training batch of images, we will first update the parameters in the Discriminator network and then we'll do the same in the Generator network.
- If we use `nn.BCELoss` as the loss criterion for training the Discriminator, that will automatically take care of the logarithms in the expression shown on the previous slide.
- We first train the Discriminator by subjecting it to a maximization that involves the three steps listed on the next slide.
- Subsequently, we train the Generator by a minimization to be described on the slide that follows.

The Two Targets for Discriminator Training

- The maximization steps required for the Discriminator training:

1. The maximization of the first term in Eq. (34) requires that we use the target "1" for the network output $D(x)$.

2. The maximization of the second term in the same expression is a bit more involved since it requires applying the Discriminator network to the output of the Generator for noise input. The second term also requires that we now use "-1" as the target for the Discriminator.

   The phrase "we now use -1 as the target for the Discriminator" is to be taken figuratively. Since the Discriminator is a binary classifier (that's what you get with `nn.BCELoss`), its targets can only be 1 and 0. We use 1 as the target in Step 1 and 0 as the target in Step 2.

3. After we have calculated the two losses for the Discriminator, we can sum the losses and call `backwards()` on the sum for calculating the gradients of the loss with respect to its weights. A subsequent call to the `step()` of the optimizer would update the weights in the Discriminator network.

The Target for Generator Training

- For the training required for the Generator, only the second term inside the square brackets in Eq. (34) matters. We proceed through the following 4 steps:

  1. We note that the logarithm is a monotonically increasing function and also because the output $D(G(z))$ in the second term will always be between 0 and 1.
  2. Therefore, the needed minimization translates into maximizing $D(G(z))$ with respect to a target value of 1.
  3. With 1 as the target, we again find the `nn.BCELoss` associated with $D(G(z))$. We call `backwards()` on this loss.
  4. As you will see on Slide 78, subsequently we call the `step()` function of the optimizer to update the parameters ONLY in the Generator network.

How the GAN Code is Organized in `AdversarialLearning`

- Now that you have become familiar with the basic idea of Adversarial Learning for data modeling, it's time to get to know better the `AdversarialLearning` co-class in the DLStudio platform.

- All of the GAN related code is in the inner class `DataModeling` of the `AdversarialLearning` class.

- The code in the `DataModeling` class allows you to experiment with the following Discriminator-Generator pairs and Critic-Generator pairs [I'll be talking about "Critics" in the next section on Wasserstein GANs.]:

DG1: This is a Discriminator-Generator pair that corresponds to the original formulation of DCGAN.

DG2: This is a slight variant of the Discriminator-Generator pair in DG1.

CG1: This is a Critic-Generator pair for the Wasserstein GAN in Section 12.

CG2: This is another Critic-Generator pair for the Wasserstein GAN.

DG1: Discriminator and Generator Networks

- Slides 72 and 73 show the DCGAN networks for the DG1 Discriminator-Generator pair.
- Regarding the Discriminator network on Slide 72, I refer to the DCGAN network topology as the 4-2-1 network. Each layer of the Discriminator network carries out a strided convolution with a 4×4 kernel, a 2×2 stride, and a 1×1 padding for all but the final layer.
- The output of the final convolutional layer in the Discriminator is pushed through a sigmoid to yield a scalar value as the final output for each image in a batch.
- Next, on Slide 73, is the implementation of the DCGAN Generator in the example DG1. As was the case with the Discriminator network, you again see the 4-2-1 topology here.

DG1: Discriminator and Generator (contd.)

- Recall that a Generator's job is to transform a random noise vector into an image that is supposed to look like it came from the training dataset. (Most people refer to the images constructed from noise vectors in this manner as fakes.)

- As you will see in `run_gan_code()`, the starting noise vector is a 1×1 image with 100 channels. In order to output a 64×64 output image from the noise vector, the Generator code shown on Slide 73 uses the Transpose Convolution operator `nn.ConvTranspose2d` with a stride of 2.

- If `(H_in, W_in)` are the height and the width of the image at the input to a `nn.ConvTranspose2d` layer and `(H_out, W_out)` the same at the output, the input/output sizes are related by [See Slides 46 through 62 of my Week 8 Lecture on Semantic Segmentation]:

```
H_out = (H_in - 1) * s + k - 2 * p
W_out = (W_in - 1) * s + k - 2 * p
```

DG1: Discriminator and Generator (contd.)

- In the last bullet on the previous slide, `s` is the stride and `k` the size of the kernel. (I am assuming square strides, kernels, and padding).
- Therefore, each `nn.ConvTranspose2d` layer doubles the size of the input.

The Discriminator Network (DG1)

```
##############################   Discriminator-Generator DG1   ##############################
class DiscriminatorDG1(nn.Module):

    def __init__(self):
        super(AdversarialLearning.DataModeling.DiscriminatorDG1, self).__init__()
        self.conv_in = nn.Conv2d(  3,    64,      kernel_size=4,      stride=2,    padding=1)
        self.conv_in2 = nn.Conv2d( 64,   128,     kernel_size=4,      stride=2,    padding=1)
        self.conv_in3 = nn.Conv2d( 128,  256,     kernel_size=4,      stride=2,    padding=1)
        self.conv_in4 = nn.Conv2d( 256,  512,     kernel_size=4,      stride=2,    padding=1)
        self.conv_in5 = nn.Conv2d( 512,  1,       kernel_size=4,      stride=1,    padding=0)
        self.bn1  = nn.BatchNorm2d(128)
        self.bn2  = nn.BatchNorm2d(256)
        self.bn3  = nn.BatchNorm2d(512)
        self.sig = nn.Sigmoid()

    def forward(self, x):
        x = torch.nn.functional.leaky_relu(self.conv_in(x), negative_slope=0.2, inplace=True)
        x = self.bn1(self.conv_in2(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn2(self.conv_in3(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn3(self.conv_in4(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.conv_in5(x)
        x = self.sig(x)
        return x
```

The Generator Network (DG1)

```
class GeneratorDG1(nn.Module):
    def __init__(self):
        super(AdversarialLearning.DataModeling.GeneratorDG1, self).__init__()
        self.latent_to_image = nn.ConvTranspose2d(100, 512, kernel_size=4, stride=1, padding=0,bias=False)
        self.upsampler2 = nn.ConvTranspose2d( 512, 256, kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler3 = nn.ConvTranspose2d (256, 128, kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler4 = nn.ConvTranspose2d (128, 64,  kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler5 = nn.ConvTranspose2d(  64,  3,  kernel_size=4, stride=2, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(512)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(128)
        self.bn4 = nn.BatchNorm2d(64)
        self.tanh  = nn.Tanh()

    def forward(self, x):
        x = self.latent_to_image(x)
        x = torch.nn.functional.relu(self.bn1(x))
        x = self.upsampler2(x)
        x = torch.nn.functional.relu(self.bn2(x))
        x = self.upsampler3(x)
        x = torch.nn.functional.relu(self.bn3(x))
        x = self.upsampler4(x)
        x = torch.nn.functional.relu(self.bn4(x))
        x = self.upsampler5(x)
        x = self.tanh(x)
        return x
```

The Training Loop for DCGAN (DG1)

- The code shown on Slides 76 through 78 implements the training logic presented on Slides 65 through 67. It is meant for training a Discriminator-Generator based Adversarial Network. The implementation shown has borrowed several programming constructs from the "official" DCGAN implementation at GitHub.

- Sections of the training loop that begin in Lines (A) and (B) are for the Discriminator part of the training in Eq. (33). The statements in Part 1(a) implement the logic in the first bullet under Discriminator training on Slide 66. In these statements we use the target of "1" for the output of the Discriminator when it is invoked on a data image.

- The statements in Part 1(b) that begin at Line (B) implement the logic in the second bullet on the Slide 66. That is, now we subject the output of the Discriminator after it is applied to the Generator images to the target "-1".

The Training Loop for DCGAN (DG1) (contd.)

- The section of the code that begins in Line (C) is for Generator training through the steps outlined on Slide 67. The min part in Eq. (33) on Slide 64 requires that we minimize $1 - D(G(z))$ which, since D is constrained to lie in the interval (0,1), requires that we maximize $D(G(z))$. We accomplish that by applying the Discriminator to the output of the Generator and use 1 as the target for each image, as mentioned in the second bullet on Slide 67.

The Training Loop for DCGAN (DG1) (contd.)

```
def run_gan_code(self, dlstudio, advers, discriminator, generator, results_dir):
    #  Set the number of channels for the 1x1 input noise vectors for the Generator:
    nz = 100
    netD = discriminator.to(advers.device)
    netG = generator.to(advers.device)
    #  Initialize the parameters of the Discriminator and the Generator networks according to the
    #  definition of the "weights_init()" method:
    netD.apply(self.weights_init)
    netG.apply(self.weights_init)
    #  We will use the same noise batch to periodically check on the progress made for the Generator:
    fixed_noise = torch.randn(self.dlstudio.batch_size, nz, 1, 1, device=advers.device)
    #  Establish convention for real and fake labels during training
    real_label = 1
    fake_label = 0
    #  Adam optimizers for the Discriminator and the Generator:
    optimizerD = optim.Adam(netD.parameters(), lr=dlstudio.learning_rate, betas=(advers.beta1, 0.999))
    optimizerG = optim.Adam(netG.parameters(), lr=dlstudio.learning_rate, betas=(advers.beta1, 0.999))
    #  Establish the criterion for measuring the loss at the output of the Discriminator network:
    criterion = nn.BCELoss()
    #  We will use these lists to store the results accumulated during training:
    img_list = []
    G_losses = []
    D_losses = []
    iters = 0
    print("\n\nStarting Training Loop...\n\n")
    start_time = time.perf_counter()
```

(Continued on the next slide .....)

The Training Loop for DCGAN (DG1) (contd.)

(...... continued from the previous slide)

```
for epoch in range(dlstudio.epochs):
    g_losses_per_print_cycle = []
    d_losses_per_print_cycle = []
    for i, data in enumerate(self.train_dataloader, 0):
        ## Part 1(a) of Training (maximization of minmax objective for the Discriminator):    ## (A)
        netD.zero_grad()
        real_images_in_batch = data[0].to(advers.device)
        #  Need to know how many images we pulled in since at the tailend of the dataset,
        #  the number of images may not equal the user-specified batch size:
        b_size = real_images_in_batch.size(0)
        label = torch.full((b_size,), real_label, dtype=torch.float, device=advers.device)
        output = netD(real_images_in_batch).view(-1)
        errD_reals = criterion(output, label)
        errD_reals.backward()
        ## Part 1(b) of Training (maximization of the minmax object for the Discriminator
        ##                                                         when applied to fakes):   ## (B)
        noise = torch.randn(b_size, nz, 1, 1, device=advers.device)
        fakes = netG(noise)
        label.fill_(fake_label)
        ##  The call to fakes.detach() in the next statement returns a copy of the 'fakes' tensor
        ##  such that the copy that is returned does not exist in the computational graph. That is,
        ##  the copy of the tensor is removed from the computational graph. However, the original
        ##  'fakes' tensor continues to remain in the computational graph.  This ploy ensures that
        ##  a subsequent call to backward() in the 3rd statement below would only result in a
        ##  calculation of the gradients for the netD weights:
        output = netD(fakes.detach()).view(-1)
        errD_fakes = criterion(output, label)
        errD_fakes.backward()
        errD = errD_reals + errD_fakes         ##  This is only for the display of losses. Not for lear
        d_losses_per_print_cycle.append(errD)
        optimizerD.step()                      ##  Only the Discriminator weights are incremented
```

The Training Loop for DCGAN (DG1) (contd.)

(...... continued from the previous slide)

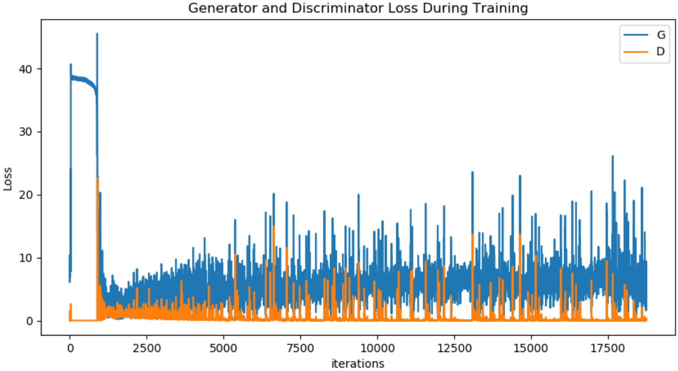
```
##  Part 2 of Training (minimization of the minmax objective for learning
##                                                    the Generator):            ## (C)
##
##  The min part requires that we MINIMIZE "1 - D(G(z))" which, since D is constrained to
##  lie in the interval (0,1), requires that we maximize D(G(z)).  We accomplish that by
##  applying the Discriminator to the output of the Generator and use 1 as the target:
netG.zero_grad()
label.fill_(real_label)
output = netD(fakes).view(-1)
errG = criterion(output, label)
g_losses_per_print_cycle.append(errG)
errG.backward()
optimizerG.step()

if i % 100 == 99:
    current_time = time.perf_counter()
    elapsed_time = current_time - start_time
    mean_D_loss = torch.mean(torch.FloatTensor(d_losses_per_print_cycle))
    mean_G_loss = torch.mean(torch.FloatTensor(g_losses_per_print_cycle))
    print("[epoch=%d/%d   iter=%4d   elapsed_time=%5d secs]     mean_D_loss=%7.4f
                                                                 mean_G_loss=%7.4f" %
               ((epoch+1),dlstudio.epochs,(i+1),elapsed_time,mean_D_loss,mean_G_loss))
    d_losses_per_print_cycle = []
    g_losses_per_print_cycle = []
```

NOTES:

- A statement like `label = torch.full((b_size,), real_label)` means that we want to set `label` to a single-axis tensor of size `b_size` and we want all its elements to be set to the value given by `real_label`.

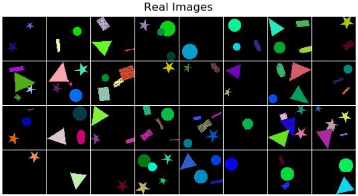
- A statement like `label.fill_(value)` means that the previously declared tensor `label` needs to be filled in-place with the specified value.

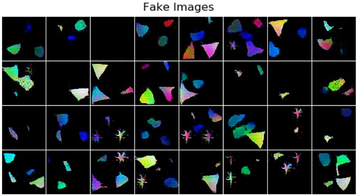
Losses vs. Iterations for DG1



Discriminator and Generator losses over 30 epochs of training

Comparing Real and Fake Images for DG1





At the end of 30 epochs of training, shown at left is a batch of real images and, at right, the images produced by the Generator from noise vectors

An Animated GIF of the Generator Output for DG1

The following animated GIF shows how the Generator's output evolves over 30 epochs using the same set of noise vectors.

https://engineering.purdue.edu/DeepLearn/pdf-kak/DG1_generation_animation.gif

Outline

Making Small Changes to the DCGAN Architecture (DG2)

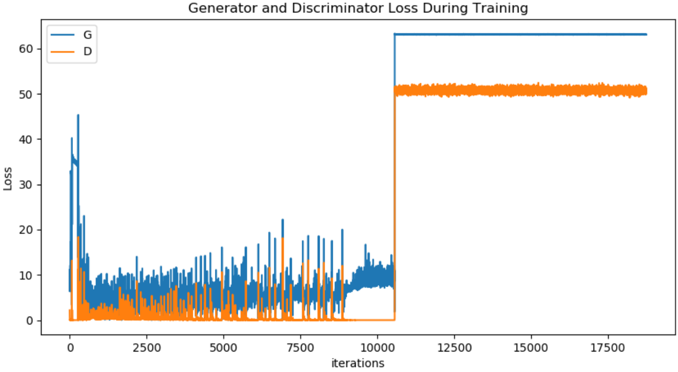
- My personal experience with the DCGAN architecture is that when it works, it produces beautiful results. However, as you change the initializations for the parameters, or as you make minor tweaks to the Generator and/or the Discriminator network, **more often than not, what you get is what is known as mode collapse.** Mode collapse means that the different randomly chosen noise vectors for the input to the Generator will yield the same garbage output.

- To illustrate what I mean, The Discriminator network shown on the next slide is the same as the one you saw earlier for the DCGAN implementation, except for the additional layer `self.extra` that the incoming image is routed through at the beginning of the network in `forward()`

- I have also defined a batch normalization layer `self.bnX` for the output of the extra layer `self.extra`.

```
#####################################   Discriminator-Generator DG2   ######################################
class DiscriminatorDG2(nn.Module):
    """
    This is essentially the same network as the DCGAN for DG1, except for the extra layer
    "self.extra" shown below.  We also declare a batchnorm for this extra layer in the form
    of "self.bnX".  In the implementation of "forward()", we invoke the extra layer at the
    beginning of the network.
    """
    def __init__(self, skip_connections=True, depth=16):
        super(AdversarialLearning.DataModeling.DiscriminatorDG2, self).__init__()
        self.conv_in = nn.Conv2d(  3,    64,      kernel_size=4,      stride=2,    padding=1)
        self.extra =   nn.Conv2d(  64,    64,      kernel_size=4,      stride=1,    padding=2)
        self.conv_in2 = nn.Conv2d( 64,   128,     kernel_size=4,      stride=2,    padding=1)
        self.conv_in3 = nn.Conv2d( 128,  256,     kernel_size=4,      stride=2,    padding=1)
        self.conv_in4 = nn.Conv2d( 256,  512,     kernel_size=4,      stride=2,    padding=1)
        self.conv_in5 = nn.Conv2d( 512,  1,       kernel_size=4,      stride=1,    padding=0)
        self.bn1  = nn.BatchNorm2d(128)
        self.bn2  = nn.BatchNorm2d(256)
        self.bn3  = nn.BatchNorm2d(512)
        self.bnX  = nn.BatchNorm2d(64)
        self.sig = nn.Sigmoid()
    def forward(self, x):
        x = torch.nn.functional.leaky_relu(self.conv_in(x), negative_slope=0.2, inplace=True)
        x = self.bnX(self.extra(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn1(self.conv_in2(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn2(self.conv_in3(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn3(self.conv_in4(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.conv_in5(x)
        x = self.sig(x)
        return x

class GeneratorDG2(nn.Module):
    """
    The Generator for DG2 is exactly the same as for the DG1.  So please the comment block for that
    Generator.
    """
    def __init__(self):
        super(AdversarialLearning.DataModeling.GeneratorDG2, self).__init__()
        self.latent_to_image = nn.ConvTranspose2d( 100,   512,  kernel_size=4, stride=1, padding=0, bias=False)
        self.upsampler2 = nn.ConvTranspose2d( 512, 256, kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler3 = nn.ConvTranspose2d (256, 128, kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler4 = nn.ConvTranspose2d (128, 64,  kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler5 = nn.ConvTranspose2d(  64,  3,  kernel_size=4, stride=2, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(512)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(128)
        self.bn4 = nn.BatchNorm2d(64)
        self.tanh  = nn.Tanh()
    def forward(self, x):
        x = self.latent_to_image(x)
        x = torch.nn.functional.relu(self.bn1(x))
        x = self.upsampler2(x)
        x = torch.nn.functional.relu(self.bn2(x))
        x = self.upsampler3(x)
        x = torch.nn.functional.relu(self.bn3(x))
        x = self.upsampler4(x)
        x = torch.nn.functional.relu(self.bn4(x))
        x = self.upsampler5(x)
        x = self.tanh(x)
        return x
########################################   DG2 Definition END   ########################################
```

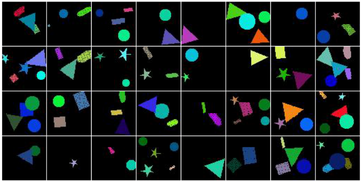
Losses vs. Iterations for DG2



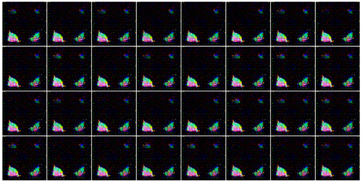
Discriminator and Generator losses over 30 epochs of training

Comparing Real and Fake Images for DG2

Real Images



Fake Images



At the end of 30 epochs of training, shown at left is a batch of real images and, at right, the images produced by the Generator from noise vectors

An Animated GIF of the Generator Output for DG2

The following animated GIF shows how the Generator's output evolves over 30 epochs using the same set of noise vectors for the case of a DCGAN with relatively minor alterations.

https://engineering.purdue.edu/DeepLearn/pdf-kak/DG2_generation_animation.gif

Outline

Wasserstein GAN Implementation in DLStudio

- This implementation is based on the paper "Wasserstein GAN" by Arjovsky, Chintala, and Bottou that I cited previously in the Preamble.

- You will find my implementation of Wasserstein GAN (WGAN) in DLStudio's co-class `AdversarialLearning`.

- As you would expect, WGAN is based on estimating the Wasserstein distance between the distribution that corresponds to the training images and the distribution that has been learned so far by the Generator. This distance was defined in Eq. (24) on Slide 38.

- The 1-Lipschitz function $f()$ that is required by the definition in Eq. (24) is implemented as a Critic — because, unlike what was the case for the Discriminator, the job of the Critic is NOT to accept or reject what is produced by the Generator, but to do what's mentioned on the next slide.

WGAN Implementation in DLStudio (contd.)

- **In a WGAN, a Critic's job is to become adept at estimating the Wasserstein distance between the distribution that corresponds to the training dataset and the distribution that has been learned by the Generator so far.**

- Since the Wasserstein distance is known to be differentiable with respect to the learnable weights in the Critic network, one can backprop the distance and update the weights in an iterative training loop. **This is roughly the idea of the Wasserstein GAN that is incorporated as a Critic-Generator pair CG1 in the Adversarial Networks class.**

- For the purpose of implementation, here is a rewrite of the Wasserstein distance presented earlier in Eq. (24) on Slide 38:

$$d_W(P_r, P_\theta) \quad = \quad \sup_{\|f\|_L \leq 1} \left[E_{x \sim P_r}\{f_w(x)\} \; - \; E_{z \sim P_z}\{f_w(g_\theta(z))\} \right] \tag{35}$$

WGAN Implementation in DLStudio (contd.)

- In the formula for Wasserstein distance shown on the previous slide, P_r is the "real" distribution that describes the training data and P_z describes the distribution of the noise vectors that are fed into the Generator for the production of the fake images. The Generator parameters are denoted θ and $g_\theta()$ stands for the function that describes the behavior of the Generator.

- Now that we have interpreted the role of the function $f_w()$ as a Critic — the Critic's job being to learn the function $f_w()$ — the question is how does the Critic make sure that the function being learned is 1-Lipschitz?

- A heuristic answer to the vexing question posed above was provided by the original authors the "Wasserstein GAN" paper. For lack of any available well-principled approach as a solution to this issue, they experimented with tightly clipping the values being learned for the weights in the Critic network.

WGAN Implementation in DLStudio (contd.)

- It stands to reason that the closer the clipping level is to zero from both the positive and the negative sides, the less likely that the gradient of the function being learned will exhibit large swings.

- The calculation of the Wasserstein distance using Eq. (35) also calls for averaging of the output of the Critic in order for the maximization to yield the desired distance. This can be taken care of by having the Critic go through multiple iterations of the update of its parameters for each iteration for the Generator.

- For implementation, the expression for the Wasserstein distance shown in Eq. (35) can be rewritten as:

$$d_W(P, Q) \quad = \quad \max_{\|f\|_L \leq 1} \left[E_{x \sim P}\{f(x)\} \; - \; E_{y \sim Q}\{f(y)\} \right] \tag{36}$$

WGAN Implementation in DLStudio (contd.)

- Note that Eq. (36) can also be interpreted as: There is guaranteed to exist a 1-Lipschitz continuous function $f()$ that when applied to the samples drawn from the distributions P and Q will yield the Wasserstein distance between the two distributions.

- Let C denote a Critic network that can learn the function $f()$. Remember, our overarching goal remains that we need to also learn a Generator network G that is capable of converting noise into samples that look like those from the distribution P.

- We seek to create a GAN that can learn a G that MINIMIZES the Wasserstein distance between the true distribution P and its learned approximation Q. At the same time, the GAN must discover a C that seeks to maximize the same distance (in the sense that the Critic learns how to maximally distrust the Generator G).

WGAN Implementation in DLStudio (contd.)

- We thus end up with the following minimax objective for the learning framework:

$$\min_G \max_C \left[E_{x \sim P}[C(x)] \; - \; E_{z \sim p_Z}[C(G(z))] \right] \tag{37}$$

- In comparing this minimax objective with the one shown earlier in Eq. (34) of Section 10, note that the two components of the argument to the minimax in that equation were additive, whereas we subtract them in the objective shown above. In Eq. (34), we had a Discriminator in the GAN and our goal was to maximize its classification performance for images that look like they came from the true distribution P. On the other hand, the goal of the Critic here is to learn to maximize the Wasserstein distance between the true distribution P and its learned approximation Q. Note that the distribution Q is for the images that are constructed by the Generator from zero-mean isotropic Gaussian noise samples z drawn from a distribution p_z, as shown above.

WGAN Implementation in DLStudio (contd.)

- As far as the Critic is concerned, the maximization needed in Eq. (37) can be achieved by using the following loss function:

$$\begin{aligned} \text{Critic Loss} &= E_{y \sim Q}[C(y)] \; - \; E_{x \sim P}[C(x)] \\ &= E_{z \sim p_z}[C(G(z))] \; - \; E_{x \sim P}[C(x)] \end{aligned} \tag{38}$$

- In the WGAN code shown in what follows, this is accomplished by using a "gradient target" of $+1$ for the mean of the output of the Critic when it sees the images produced by the Generator and the "gradient target" of -1 for the output of the Critic when it sees the training data directly.

- As to why we use the gradient targets of $+1$ and -1, it was shown by the original authors of WGAN that the optimal Critic C has unit gradient norm almost everywhere under P and Q. That is, the magnitude of the partial derivative of the output of the optimal C with respect to its input will almost always be 1.

The Critic and the Generator in DLStudio's WGAN

```
##########################################   Critic-Generator CG1   ########################################
class CriticCG1(nn.Module):
    """
    I have used the SkipBlockDN as a building block for the Critic network.  This I did with the hope
    that when time permits I may want to study the effect of skip connections on the behavior of the
    the critic vis-a-vis the Generator.  The final layer of the network is the same as in the
    "official" GitHub implementation of Wasserstein GAN.  And, as in WGAN, I have used the leaky ReLU
    for activation.
    """
    def __init__(self):
        super(AdversarialLearning.DataModeling.CriticCG1, self).__init__()
        self.conv_in = AdversarialLearning.DataModeling.SkipBlockDN(3, 64, downsample=True, skip_connections=True)
        self.conv_in2 = AdversarialLearning.DataModeling.SkipBlockDN( 64,   128,  downsample=True, skip_connections=False)
        self.conv_in3 = AdversarialLearning.DataModeling.SkipBlockDN(128,   256,  downsample=True, skip_connections=False)
        self.conv_in4 = AdversarialLearning.DataModeling.SkipBlockDN(256,   512,  downsample=True, skip_connections=False)
        self.conv_in5 = AdversarialLearning.DataModeling.SkipBlockDN(512,   1,  downsample=False, skip_connections=False)
        self.bn1  = nn.BatchNorm2d(128)
        self.bn2  = nn.BatchNorm2d(256)
        self.bn3  = nn.BatchNorm2d(512)
        self.final = nn.Linear(512, 1)
    def forward(self, x):
        x = torch.nn.functional.leaky_relu(self.conv_in(x), negative_slope=0.2, inplace=True)
        x = self.bn1(self.conv_in2(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn2(self.conv_in3(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.bn3(self.conv_in4(x))
        x = torch.nn.functional.leaky_relu(x, negative_slope=0.2, inplace=True)
        x = self.conv_in5(x)
        x = x.view(-1)
        x = self.final(x)
        x = x.mean(0)
        x = x.view(1)
        return x

class GeneratorCG1(nn.Module):
    """
    The Generator code remains the same as for the DCGAN shown earlier.
    """
    def __init__(self):
        super(AdversarialLearning.DataModeling.GeneratorCG1, self).__init__()
        self.latent_to_image = nn.ConvTranspose2d( 100,   512,  kernel_size=4, stride=1, padding=0, bias=False)
        self.upsampler2 = nn.ConvTranspose2d( 512, 256, kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler3 = nn.ConvTranspose2d (256, 128, kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler4 = nn.ConvTranspose2d (128, 64,  kernel_size=4, stride=2, padding=1, bias=False)
        self.upsampler5 = nn.ConvTranspose2d(  64,  3,  kernel_size=4, stride=2, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(512)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(128)
        self.bn4 = nn.BatchNorm2d(64)
        self.tanh  = nn.Tanh()
    def forward(self, x):
        x = self.latent_to_image(x)
        x = torch.nn.functional.relu(self.bn1(x))
        x = self.upsampler2(x)
        x = torch.nn.functional.relu(self.bn2(x))
        x = self.upsampler3(x)
        x = torch.nn.functional.relu(self.bn3(x))
        x = self.upsampler4(x)
        x = torch.nn.functional.relu(self.bn4(x))
        x = self.upsampler5(x)
        x = self.tanh(x)
        return x
```

Training the WGAN

- The code for training the Critic-Generator based WGAN shown next is based on the logic of a Wasserstein GAN as proposed by the original authors of WGAN. The implementation shown uses several programming constructs from the WGAN implementation at GitHub. I have also used several programming constructs from the DCGAN code at GitHub.

- The noise batch that is generated in Line (D) is used periodically check on the progress made by the Generator.

- The 'one' and 'minus_one' you see in Lines (E) and (F) are for training the Critic, 'minus_one' is for the part of the training with actual training images, and 'one' is for the part based on the images produced by the Generator.

- The inner 'while' loop in Line (G) is for updating the Critic in such a way that the discrimination function learned by the Critic satisfies the 1-Lipschitz condition.

Training the WGAN (contd.)

- The 1-Lipschitz condition is enforced by the clipping statements in Lines (H) and (I) along with the smoothing action of the inner 'while' loop.

- As mentioned previously, a minimization of the Wasserstein distance between the distribution that describes the training data and the distribution that has been learned so far by the Generator can be translated into a *maximization* of the difference of the *average outputs* of a 1-Lipschitz function as applied to the training images and as applied to the output of the Generator. Learning this 1-Lipschitz function is the job of the Critic.

- Training the Critic consists of two parts. In the first part that begins in Line (J), we apply the target 'one' to the training images and, in the second part that begins in Line (K), we use the target 'minus_one' for the output of the Critic when its input is the output of the Generator.

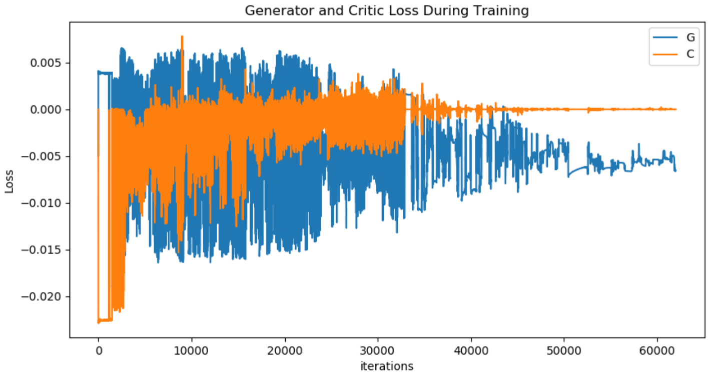
Training the WGAN (contd.)

- That brings us to the training of the Generator that begins in Line (L). We must start by turning off the `requires_grad` of the Critic parameters since the Critic and the Generator are meant to be updated independently.

Training the WGAN

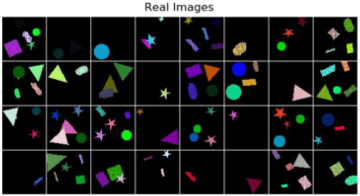
```
def run_wgan_code(self, dlstudio, adversarial, critic, generator, results_dir):
    nz = 100            #  Set the number of channels for the 1x1 input noise vectors for the Generator                    ## (A)
    netC = critic.to(advers.device)
    netG = generator.to(advers.device)
    netC.apply(self.weights_init)           # initialize Critic network parameters                                        ## (B)
    netG.apply(self.weights_init)           # initialize Generator network parameters                                     ## (C)
    fixed_noise = torch.randn(self.dlstudio.batch_size, nz, 1, 1, device=advers.device)                                   ## (D)
    one = torch.FloatTensor([1]).to(advers.device)                                                                        ## (E)
    minus_one = torch.FloatTensor([-1]).to(advers.device)                                                                 ## (F)
    #  Adam optimizers for the Critic and the Generator:
    optimizerC = optim.Adam(netC.parameters(), lr=dlstudio.learning_rate, betas=(adversarial.beta1, 0.999))
    optimizerG = optim.Adam(netG.parameters(), lr=dlstudio.learning_rate, betas=(adversarial.beta1, 0.999))
    img_list = []
    Gen_losses = []
    Cri_losses = []
    iters = 0
    gen_iterations = 0
    start_time = time.perf_counter()
    dataloader = self.train_dataloader
    clipping_thresh = self.adversarial.clipping_threshold
    for epoch in range(dlstudio.epochs):
        data_iter = iter(dataloader)
        i = 0
        ncritic   = 5
        while i < len(dataloader):
            for p in netC.parameters():
                p.requires_grad = True
            if gen_iterations < 25 or gen_iterations % 500 == 0:    # the choices 25 and 500 are from WGAN
                ncritic = 100
            ic = 0
            ##  The inner 'while' loop shown below calculates the expectations in Eq. (8) in the doc section
            ##  at the beginning of this file:
            while ic < ncritic and i < len(dataloader):                                                               ## (G)
                ic += 1
                for p in netC.parameters():                                                                           ## (H)
                    p.data.clamp_(-clipping_thresh, clipping_thresh)                                                  ## (I)
                ## Training the Critic with real images (Part 1):
                netC.zero_grad()                                                                                      ## (J)
                real_images_in_batch =  data_iter.next()
                i += 1
                real_images_in_batch =  real_images_in_batch[0].to(self.device)
                #  Need to know how many images we pulled in since at the tailend of the dataset, the
                #  number of images may not equal the user-specified batch size:
                b_size = real_images_in_batch.size(0)
                #  Note that a single scalar is produced for all the data in a batch.  This is probably
                #  the reason why what the Generator learns is somewhat fuzzy.
                critic_for_reals_mean = netC(real_images_in_batch)
                ## 'minus_one' is the gradient target:
                critic_for_reals_mean.backward(minus_one)

                ## Training the Critic with fake images (Part 2):                                                     ## (K)
                noise = torch.randn(b_size, nz, 1, 1, device=self.device)
                fakes = netG(noise)
                #  Again, a single number is produced for the whole batch:
                critic_for_fakes_mean = netC(fakes)
                ## 'one' is the gradient target:
                critic_for_fakes_mean.backward(one)
                wasser_dist = critic_for_reals_mean - critic_for_fakes_mean
                loss_critic = critic_for_fakes_mean - critic_for_reals_mean
                #  Update the Critic
                optimizerC.step()

            ## Training the Generator:                                                                               ## (L)
            for p in netC.parameters():
                p.requires_grad = False
            netG.zero_grad()
            #  This is again a single scalar based characterization of the whole batch of the Generator images:
            noise = torch.randn(b_size, nz, 1, 1, device=self.device)
            fakes = netG(noise)
            critic_for_fakes_mean = netC(fakes)
            loss_gen = critic_for_fakes_mean
            critic_for_fakes_mean.backward(minus_one)
            #  Update the Generator
            optimizerG.step()
            gen_iterations += 1
```

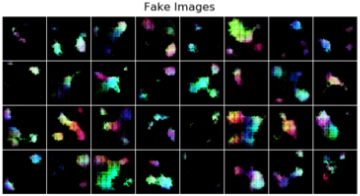
Losses vs. Iterations for WGAN



Critic and Generator losses over 500 epochs of training

Comparing Real and Fake Images for WGAN





At the end of 500 epochs of training, shown at left is a batch of real images and, at right, the images produced by the Generator from noise vectors

An Animated GIF of the Generator Output for WGAN

The following animated GIF shows how the Generator's output evolves over 30 epochs using the same set of noise vectors for the case of a DCGAN with relatively minor alterations.

https://engineering.purdue.edu/DeepLearn/pdf-kak/WGAN_generation_animation.gif

Outline

WGAN-GP: Improving WGAN with Gradient Penalty

- As you would guess, the name extension "-GP" stands for "Gradient Penalty".

- It was shown by the authors Gulrajani, Ahmed, Arjovsky, Dumouli, and Courville of the paper "Improved Training of Wasserstein GANs" that implementing a 1-Lipschitz constraint with weight clipping as discussed in the previous section biases the Critic towards learning rather simple probability distribution functions.

- In WGAN-GP, the performance of a WGAN is improved by putting to use the theoretical property that the optimal WGAN critic C has unit gradient norm almost everywhere under P and Q. [See Proposition 1, Corollary 1 of the paper cited above.]

WGAN-GP (contd.)

- On the basis of the property mentioned at the bottom of the previous slide, in a WGAN-GP, we add a Gradient Penalty term to the Critic Loss that was shown earlier in Eq. (38):

$$\text{Critic Loss} \quad = \quad \underbrace{E_{z\sim p_z}[C(G(z))] \; - \; E_{x\sim P}[C(x)]}_{\text{The original critic loss}} \quad + \quad \underbrace{\lambda[\|\nabla_{\hat{x}} C(\hat{x})\|^2 - 1]^2}_{\text{The Gradient Penalty (GP)}} \tag{39}$$

- To explain what the symbol $\hat{x}$ is doing in the GP term, note that the gradient is of the output of the 1-Lipschitz function (meaning the output of the Critic network) with respect to its input. Since the Critic network sees both the training samples and those produced by the Generator at its input, for the purpose of calculating this gradient, we first construct a fictitious sample by taking a weighted sum of a sample drawn from the training data and one produced by the Generator using a randomly chosen fractional number ϵ:

$$\hat{x} \quad = \quad \epsilon x \quad + \quad (1-\epsilon)\tilde{x} \tag{40}$$

WGAN-GP (contd.)

- Shown below is the Tensorflow code for calculating the Gradient Penalty as posted by the authors of the "Improved Training of Wasserstein GANs" paper:

```
if MODE == 'wgan-gp':
    epsilon = tf.random_uniform( shape=[BATCH_SIZE,1], minval=0., maxval=1. )
    interpolates = epsilon*real_data + ((1-epsilon)*fake_data)
    disc_interpolates = Discriminator(interpolates)
    gradients = tf.gradients(disc_interpolates, [interpolates])[0]
    slopes = tf.sqrt(tf.reduce_sum(tf.square(gradients), reduction_indices=[1]))
    gradient_penalty = tf.reduce_mean((slopes-1)**2)
```

- Shown below is the PyTorch version of the same code as posted by Marvin Cao (caogang) at GitHub:

```
def calc_gradient_penalty(netC, real_data, fake_data):
    epsilon = torch.rand(batch_size, 1).cuda()
    epsilon = epsilon.expand(real_data.size())
    interpolates = epsilon * real_data + ((1 - epsilon) * fake_data)
    interpolates = interpolates.requires_grad_(True).cuda()
    critic_interpolates = netC(interpolates)
    gradients = autograd.grad(outputs=critic_interpolates, inputs=interpolates,
                              grad_outputs=torch.ones(critic_interpolates.size()).cuda(),
                              create_graph=True, retain_graph=True, only_inputs=True)[0]
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean() * LAMBDA
    return gradient_penalty
```

WGAN-GP for Learning a Point Distribution in 2D

- In order to demonstrate how effective the gradient penalty is in improving the performance of a WGAN, I'll use the 8-Gaussian example from the "Improved Training of Wasserstein GANs" paper. 8-Gaussian refers to a multi-Gaussian distribution of points in an xy-plane. The centers of the eight Gaussians are equispaced on a unit circle around the origin of the plane. The width of each Gaussian is specified by the user. The code snippet shown below returns a batch of 256 points in the xy-plane each time the function `multi_gaussian_source()` is called:

```
def multi_gaussian_source():
    """
    A Python 'generator' function:  Each call to this function with the built-in "next()" will yield
    a fresh BATCH_SIZE (typically 256) number of points in the xy-plane.
    """
    scale = 2.
    centers = [ (1, 0), (-1, 0),  (0, 1), (0, -1),  (1. / np.sqrt(2), 1. / np.sqrt(2)),
                (1. / np.sqrt(2), -1. / np.sqrt(2)), (-1. / np.sqrt(2), 1. / np.sqrt(2)),
                (-1. / np.sqrt(2), -1. / np.sqrt(2))
              ]
    centers = [(scale * x, scale * y) for x, y in centers]
    while True:
        dataset = []
        #spread = 0.02
        spread = 0.1                          ## controls the spread of each Gaussian
        for i in range(BATCH_SIZE):
            point = np.random.randn(2) * spread
            center = random.choice(centers)
            point[0] += center[0]
            point[1] += center[1]
            dataset.append(point)
        dataset = np.array(dataset, dtype='float32')
        dataset /= 1.414  # stdev
        yield dataset
```

WGAN-GP for Learning a Point Distro (contd.)

- Given the data source shown on the previous slide for the ground-truth, we want to train a WGAN so that its Generator would transform noise into data samples (points in the xy-plane) that look like they came from the 8-Gaussians distribution. For the WGAN, we will use the Generator and the Critic classes as shown below:

```
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        main = nn.Sequential(
            nn.Linear(2, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, 2),
        )
        self.main = main

    def forward(self, noise):
        output = self.main(noise)
        return output

class Critic(nn.Module):
    def __init__(self):
        super(Critic, self).__init__()
        main = nn.Sequential(
            nn.Linear(2, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, 1),
        )
        self.main = main

    def forward(self, inputs):
        output = self.main(inputs)
        return output
```

WGAN-GP for Learning a Point Distro (contd.)

- As you can see on the previous slide, except for the last layer, the network layout for both the Generator and the Critic are identical. The output of the Generator is 2D because it is supposed to generate points in the xy-plane. On the other hand, the output of the Critic is a 1D value that expresses the Critic's confidence that the input is genuine or fake.

- The next code segment that follows is about the training of the WGAN. A highlight of the code shown is three calls to `backward()` for estimating the gradients of the Critic weights in lines (F), (K), and (M) and one call to the same for the Generator in line (U).

- To elaborate the code shown for WGAN training, the main loop starts in line (A). Each iteration of the main training loop involves training the Generator network once. At the same time, it requires that the Critic be taken through multiple updates in keeping with the requirements of the expectation operator in Eq. (38) and Eq. (39).

WGAN-GP for Learning a Point Distro (contd.)

- The multiple updates of the Critic in the inner loop start in line (B). The needs of the expectation operator in Eq. (39) are met by averaging both over multiple iterations of the inner loop that starts in line (B) and, in each of those iterations, by averaging over all the samples in a batch, as you will soon see.

- Each inner-loop update of the Critic entails first feeding it a batch of real data (typically 256 points in the xy-plane) in line (D). In keeping with the requirements of the expectation in Eq. (39), we find the mean of the output of the Critic for all the samples in the batch in line (E). The call to `backward()` in line (F) updates the gradients of the Critic weights for this phase of learning for the Critic. Note the target gradient of "-1" in the call to `backward()` in line (F).

- For the next phase of Critic learning, we feed it a batch of the fakes produced by the Generator in line (H).

WGAN-GP for Learning a Point Distro (contd.)

- With regard to phase of Critic learning at the bottom of the previous slide, for the same reason as mentioned earlier, this output of the Critic is averaged over the batch in line (J) and subject to a call to `backward()` in line (K).

- For the third and final phase of Critic learning, using the implementation shown earlier, we first estimate the gradient penalty in line (L) and then make the call on `backward()` in line (M) for the final updating the gradients of the Critic weights in this iteration of the inner loop for Critic training.

```
for iteration in range(ITERS):                                                                   ## (A)
    ## Update the Critic network:
    for p in netC.parameters():                 # reset the requires_grad attribute
        p.requires_grad = True                  #   this attribute is set to False in netG update

    for iter_d in range(CRITIC_ITERS):                                                            ## (B)
        ## The data_source supplies one BATCH_SIZE of 2D points from true distribution
        real_data = next(data_source)                                                             ## (C)
        real_data = torch.Tensor(real_data).requires_grad_(True).cuda()
        netC.zero_grad()
        # Train Critic network with real data:
        critic_for_reals = netC(real_data)                                                        ## (D)
        critic_for_reals_mean = critic_for_reals.mean()                                           ## (E)
        critic_for_reals_mean = torch.unsqueeze(critic_for_reals_mean, 0)
        critic_for_reals_mean.backward(minus_one)                                                 ## (F)
        # Train Critic with Generator output:
        noise = torch.randn(BATCH_SIZE, 2).requires_grad_(True).cuda()                            ## (G)
        fakes = netG(noise)                                                                       ## (H)
        critic_for_fakes = netC(fakes)                                                            ## (I)
        critic_for_fakes_mean = critic_for_fakes.mean()                                           ## (J)
        critic_for_fakes_mean = torch.unsqueeze(critic_for_fakes_mean,0)
        critic_for_fakes_mean.backward(one)                                                       ## (K)
```

WGAN-GP for Learning a Point Distro (contd.)

(...... continued from the previous slide)

```
    if MODE == "wgan-gp":
        # Train Critic with gradient penalty:
        gradient_penalty = calc_gradient_penalty(netC, real_data, fakes.data)                      ## (L)
        gradient_penalty.backward()                                                                ## (M)
        lossCritic = critic_for_fakes_mean - critic_for_reals_mean + gradient_penalty              ## (N)
    elif MODE == "wgan":
        lossCritic = critic_for_fakes_mean - critic_for_reals_mean                                 ## (O)
    wasser_dist = critic_for_reals_mean - critic_for_fakes_mean                                    ## (P)
    optim_critic.step()

## Update the Generator network:
for p in netC.parameters():                                                                        ## (Q)
    p.requires_grad = False
netG.zero_grad()
noise = torch.randn(BATCH_SIZE, 2).requires_grad_(True).cuda()                                     ## (R)
## A BATCH_SIZE of fakes coming from the Generator
fakes = netG(noise)                                                                                ## (S)
critic_for_fakes_g = netC(fakes)
critic_for_fakes_g_mean = critic_for_fakes_g.mean()                                                ## (T)
critic_for_fakes_g_mean = torch.unsqueeze(critic_for_fakes_g_mean,0)
critic_for_fakes_g_mean.backward(minus_one)                                                        ## (U)
lossGen = -critic_for_fakes_g_mean
optim_gen.step()
```

- The code file that follows is the full implementation of the WGAN code. On the code shown, what remains unexplained is the implementation of the function `display_distributions()` that plays an important role in depicting the effectiveness of using gradient penalty for training a WGAN. That implementation will be explained later in this section.

Implementation of `wgan_for_point_distros.py`

```
##  wgan_for_point_distros.py

import random
import matplotlib
matplotlib.use('Agg')
import matplotlib.pyplot as plt
import numpy as np
import torch
import torch.autograd as autograd
import torch.nn as nn
import torch.optim as optim
import sys, os, glob, time

MODE = 'wgan-gp'                  # Choose one of the two
#MODE = 'wgan'

DIM = 512                         # Dimensionality of nn.Linear layers
LAMBDA = .1                       # For estimating the contribution of GP to overall loss
CRITIC_ITERS = 5                  # How many critic iterations per generator iteration
BATCH_SIZE = 256                  # Batch size
ITERS = 100000                    # how many generator iterations to train for
#ITERS = 10000                    # how many generator iterations to train for
dir_name_for_results = 'results' + "_" + MODE

# =============================== Refresh directory for results ===============================
if os.path.exists(dir_name_for_results):
    files = glob.glob(dir_name_for_results + "/*")
    for file in files:
        if os.path.isfile(file):
            os.remove(file)
        else:
            files = glob.glob(file + "/*")
            list(map(lambda x: os.remove(x), files))
else:
    os.mkdir(dir_name_for_results)

since_beginning_dict = {'critic_loss' : [], 'wasser_dist': [], 'gen_loss': []}
since_last_flush_dict = {'critic_loss' : [], 'wasser_dist': [], 'gen_loss': []}

# ================================== Class Definitions ========================================
class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        main = nn.Sequential(
            nn.Linear(2, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, 2),
        )
        self.main = main
    def forward(self, noise):
        output = self.main(noise)
        return output

class Critic(nn.Module):
    def __init__(self):
        super(Critic, self).__init__()
        main = nn.Sequential(
            nn.Linear(2, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, DIM),
            nn.ReLU(True),
            nn.Linear(DIM, 1),
        )
        self.main = main
    def forward(self, inputs):
        output = self.main(inputs)
        return output
```

wgan_for_point_distros.py (contd.)

(...... continued from the previous slide)

```
def weights_init(m):
    """
    This function is used to initialize the learnable weights in the Critic and
    the Generator networks
    """
    classname = m.__class__.__name__
    if classname.find('Linear') != -1:
        m.weight.data.normal_(0.0, 0.02)
        m.bias.data.fill_(0)

frame_index = [0]

# ================================ Utility Functions ===================================
def display_distributions(real_data, netC, netG):
    """
    This very useful visualization function, written originally by the authors
    of the celebrated paper "Improved Training of Wasserstein GANs", does the
    following three things simultaneously:

    1) Creates a 128x128 array of points in a [-3,3]x[-3x3] box in the xy-plane.
       These points can subsequently be fed into the Critic for the values it
       would yield at each point in the array.  The value returned by the Critic
       at point (x,y) in the array would be Critic's confidence whether that
       (x,y) point belongs to the probability distribution for the training data.
       The surface formed by such Critic values is best visualized through
       equi-valued contours.

    2) The 'real_data' that is the first argument to this function is a batch-full
       (typically 256) points in the xy-plane that were produced by the function
       multi_gaussian_source(). This source represents the true training data
       for training the GAN. These points are shown in the xy-plane by orange
      '+' points.

    3) It takes a batch-full (typically 256) 2D noise vectors and sends them
       through the Generator network netG. The Generator network produces a
       2D point in the xy-plane for each 2D noise input.  The 256 points returned
       by the Generator are displayed as green 'x' points in the same xy-plane
       that is used for the above two items.
    """
    NPOINTS = 128
    RANGE = 3
    points = np.zeros((NPOINTS, NPOINTS, 2), dtype='float32')
    points[:,:,0] = np.linspace(-RANGE, RANGE, NPOINTS)[:,None]
    points[:,:,1] = np.linspace(-RANGE, RANGE, NPOINTS)[None,:]
    points = points.reshape((-1, 2))
    points = torch.Tensor(points).requires_grad_(False).cuda()
    ## Generate the Critic's value at each point at each (x,y) point
    ## created above:
    critic_map = netC(points).cpu().data.numpy()
    ## Now we need a batch-full of 2D noise vectors for feeding into
    ## the Generator:
    noise = torch.randn(BATCH_SIZE, 2).requires_grad_(False).cuda()
    fakes = netG(noise).cpu().detach().numpy()
    plt.clf()
    x = y = np.linspace(-RANGE, RANGE, NPOINTS)
    ## Display the Critic output value surface through contours:
    plt.contour(x, y, critic_map.reshape((len(x), len(y))).transpose())
    ## Display the 256 first-arg real_data points that were previously
    ## generated by the ground-truth source multi_gaussian_source():
    plt.scatter(real_data[:,0], real_data[:,1], c='orange', marker='+')
    ## Now display the 256 'fake' points returned by the Generator:
    plt.scatter(fakes[:,0], fakes[:,1], c='green', marker='x')
    plt.savefig(dir_name_for_results + "/" + 'frame' + str(frame_index[0]) + '.jpg')
    frame_index[0] += 1
```

(Continued on the next slide .....)

wgan_for_point_distros.py (contd.)

(...... continued from the previous slide)

```
def multi_gaussian_source():
    """
    A Python 'generator' function:  Each call to this function with the built-in "next()" will yield
    a fresh BATCH_SIZE (typically 256) number of points in the xy-plane.
    """
    scale = 2.
    centers = [
        (1, 0),
        (-1, 0),
        (0, 1),
        (0, -1),
        (1. / np.sqrt(2), 1. / np.sqrt(2)),
        (1. / np.sqrt(2), -1. / np.sqrt(2)),
        (-1. / np.sqrt(2), 1. / np.sqrt(2)),
        (-1. / np.sqrt(2), -1. / np.sqrt(2))
    ]
    centers = [(scale * x, scale * y) for x, y in centers]
    while True:
        dataset = []
#            spread = 0.02
        spread = 0.1                          ## controls the spread of each Gaussian
        for i in range(BATCH_SIZE):
            point = np.random.randn(2) * spread
            center = random.choice(centers)
            point[0] += center[0]
            point[1] += center[1]
            dataset.append(point)
        dataset = np.array(dataset, dtype='float32')
        dataset /= 1.414  # stdev
        yield dataset

def calc_gradient_penalty(netC, real_data, fake_data):
    """
    Implementation by Marvin Cao at GitHub
    """
    epsilon = torch.rand(BATCH_SIZE, 1).cuda()
    epsilon = epsilon.expand(real_data.size())
    interpolates = epsilon * real_data + ((1 - epsilon) * fake_data)
    interpolates = interpolates.requires_grad_(True).cuda()
    critic_interpolates = netC(interpolates)
    gradients = autograd.grad(outputs=critic_interpolates, inputs=interpolates,
                              grad_outputs=torch.ones(critic_interpolates.size()).cuda(),
                              create_graph=True, retain_graph=True, only_inputs=True)[0]
    gradient_penalty = ((gradients.norm(2, dim=1) - 1) ** 2).mean() * LAMBDA
    return gradient_penalty

# ========================= Create Network Instances and Train WGAN ==============================
netG = Generator().cuda()
netC = Critic().cuda()
netC.apply(weights_init)
netG.apply(weights_init)

# print(netG)
# print(netC)

optim_critic = optim.Adam(netC.parameters(), lr=1e-4, betas=(0.5, 0.9))
optim_gen = optim.Adam(netG.parameters(), lr=1e-4, betas=(0.5, 0.9))

one = torch.FloatTensor([1])
minus_one = one * -1

one = one.cuda()
minus_one = minus_one.cuda()

data_source = multi_gaussian_source()     ##  returns one BATCH_SIZE collection 2D points

start_time = time.perf_counter()
```


(Continued on the next slide .....)

wgan_for_point_distros.py (contd.)

(..... continued from the previous slide)

```
for iteration in range(ITERS):
    ## Update the Critic network
    for p in netC.parameters():                # reset the requires_grad attribute
        p.requires_grad = True                 #   this attribute is set to False in netG update

    for iter_d in range(CRITIC_ITERS):
        ## The data_source supplies one BATCH_SIZE of 2D points from true distribution
        real_data = next(data_source)
        real_data = torch.Tensor(real_data).requires_grad_(True).cuda()
        netC.zero_grad()
        # train Critic network with real data
        critic_for_reals = netC(real_data)
        critic_for_reals_mean = critic_for_reals.mean()
        critic_for_reals_mean = torch.unsqueeze(critic_for_reals_mean, 0)
        critic_for_reals_mean.backward(minus_one)
        # train Critic with fakes
        noise = torch.randn(BATCH_SIZE, 2).requires_grad_(True).cuda()
        fakes = netG(noise)
        critic_for_fakes = netC(fakes)
        critic_for_fakes_mean = critic_for_fakes.mean()
        critic_for_fakes_mean = torch.unsqueeze(critic_for_fakes_mean,0)
        critic_for_fakes_mean.backward(one)

        if MODE == "wgan-gp":
            # train with gradient penalty
            gradient_penalty = calc_gradient_penalty(netC, real_data, fakes.data)
            gradient_penalty.backward()
            lossCritic = critic_for_fakes_mean - critic_for_reals_mean + gradient_penalty
        elif MODE == "wgan":
            lossCritic = critic_for_fakes_mean - critic_for_reals_mean
        wasser_dist = critic_for_reals_mean - critic_for_fakes_mean
        optim_critic.step()

    ## Update the Generator network
    for p in netC.parameters():
        p.requires_grad = False
    netG.zero_grad()
    noise = torch.randn(BATCH_SIZE, 2).requires_grad_(True).cuda()
    ## A BATCH_SIZE of fakes coming from the Generator
    fakes = netG(noise)
    critic_for_fakes_g = netC(fakes)
    critic_for_fakes_g_mean = critic_for_fakes_g.mean()
    critic_for_fakes_g_mean = torch.unsqueeze(critic_for_fakes_g_mean,0)
    critic_for_fakes_g_mean.backward(minus_one)
    lossGen = -critic_for_fakes_g_mean
    optim_gen.step()

    ## Update the dicts for the losses and distance:
    since_last_flush_dict['critic_loss'].append(lossCritic.cpu().data.numpy()[0])
    since_last_flush_dict['wasser_dist'].append(wasser_dist.cpu().data.numpy()[0])
    since_last_flush_dict['gen_loss'].append(lossGen.cpu().data.numpy()[0])

    if iteration % 100 == 99:
        current_time = time.perf_counter()
        elapsed_time = int(current_time - start_time)
        prints = []
        for name, vals in since_last_flush_dict.items():
            prints.append("{}  {:.3f}".format(name, np.mean(vals)))
            since_beginning_dict[name] += vals
        print( "[iter: {:5d}  time: {:5d} secs]\t\t{}".format(iteration+1, elapsed_time, "\t".join(prints)) )
        since_last_flush_dict = {'critic_loss' : [], 'wasser_dist': [], 'gen_loss': []}
        real_data = next(data_source)
        display_distributions(real_data, netC, netG)
    iteration += 1
```

(Continued on the next slide .....)

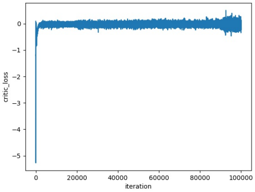
`wgan_for_point_distros.py` (contd.)

(...... continued from the previous slide)

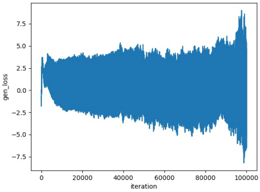
```
for name,vals in since_beginning_dict.items():
    x_vals = np.array(range(iteration))
    y_vals = [since_beginning_dict[name][x] for x in x_vals]
    plt.clf()
    plt.plot(x_vals, y_vals)
    plt.xlabel('iteration')
    plt.ylabel(name)
    plt.savefig(dir_name_for_results + "/" + name.replace(' ', '_')+'.jpg')
```

- Shown on the next slide are the Critic Loss, the Generator Loss, and the Wasserstein Distance calculated in the main training loop of the code.

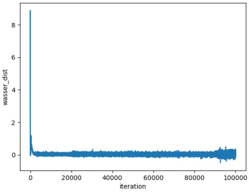
Losses vs. Iterations for WGAN-GP



Critic Loss



Generator Loss



Wasserstein Distance

Losses and distance based on 100,000 of training

Comparing GP with No-GP in Training a WGAN

- As to how effective using the Gradient Penalty is in improving the performance is of a WGAN is best visualized by using the function `display_distributions()` whose implementation is presented next. The code for this function is as provided by the original authors of the paper "Improved Training of Wasserstein GANs". This function does the following three things simultaneously:
  - Creates a 128×128 array of points in a $[-3, 3] \times [-3x3]$ box in the xy-plane. These points can subsequently be fed into the Critic for the values it would yield at each point. The value returned by the Critic at point (x,y) is Critic's confidence whether that point belongs to the probability distribution for the training data. The surface formed by such Critic values is best visualized through equi-valued contours.
  - The `real_data` that is the first argument to this function is a batch-full (typically 256) points in the xy-plane that were produced by the function `multi_gaussian_source()`. This source represents the ground-truth data for training the GAN. These points are shown in the xy-plane by orange '+' points.

Comparing GP with No-GP (contd.)

- It takes a batch-full (typically 256) 2D noise vectors and sends them through the Generator network netG. The Generator network produces a 2D point in the xy-plane for each 2D noise input. The 256 points returned by the Generator are displayed as green 'x' points in the same xy-plane that is used for the above two items.

- Shown below is the implementation of `display_distributions()`:

```
def display_distributions(real_data, netC, netG):
    NPOINTS = 128
    RANGE = 3
    points = np.zeros((NPOINTS, NPOINTS, 2), dtype='float32')
    points[:,:,0] = np.linspace(-RANGE, RANGE, NPOINTS)[:,None]
    points[:,:,1] = np.linspace(-RANGE, RANGE, NPOINTS)[None,:]
    points = points.reshape((-1, 2))
    points = torch.Tensor(points).requires_grad_(False).cuda()
    ## Generate the Critic's value at each point at each (x,y) point
    ## created above:
    critic_map = netC(points).cpu().data.numpy()
    ## Now we need a batch-full of 2D noise vectors for feeding into
    ## the Generator:
    noise = torch.randn(BATCH_SIZE, 2).requires_grad_(False).cuda()
    fakes = netG(noise).cpu().detach().numpy()
    plt.clf()
    x = y = np.linspace(-RANGE, RANGE, NPOINTS)
    ## Display the Critic output value surface through contours:
    plt.contour(x, y, critic_map.reshape((len(x), len(y))).transpose())
    ## Display the 256 first-arg real_data points that were previously
    ## generated by the ground-truth source multi_gaussian_source():
    plt.scatter(real_data[:,0], real_data[:,1], c='orange', marker='+')
    ## Now display the 256 'fake' points returned by the Generator:
    plt.scatter(fakes[:,0], fakes[:,1], c='green', marker='x')
    plt.savefig(dir_name_for_results + "/" + 'frame' + str(frame_index[0]) + '.jpg')
    frame_index[0] += 1
```

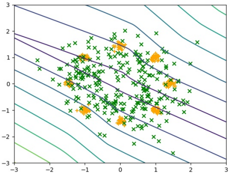
Comparing GP with No-GP (contd.)

- Shown in the next few slides is a side-by-side comparison of GP vs. no-GP on WGAN training at the same iteration index. The plots were produced by the function `display_distributions()` during a training session that consisted of 100,000 iterations.

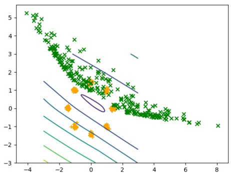
- As mentioned earlier, the orange '+' marks denote the points in the xy-plane as produced by the true 8-Gaussian distribution and the green 'x' marks denote the points that the Generator produced from purely noise input. The greater the overlap between the green 'x' points and the orange '+' points, the superior the performance of the Generator. In addition, you would want the clusters formed by the green 'x' points to be as tight as those formed by the orange '+' points. Finally, you would want all the green 'x' points to fall inside the $[-3, 3] \times [-3, 3]$ box in the xy-plane.

- The contours depict the value surface for the Critic.

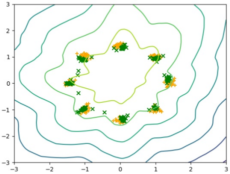
GP vs. No-GP Performance Comparison



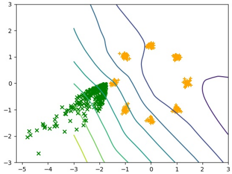
(a) With GP at iteration 10,000



(b) Without GP at iteration 10,000

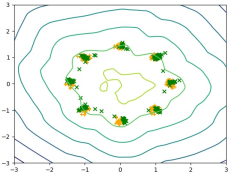


(a) With GP at iteration 20,000

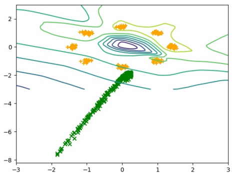


(b) Without GP at iteration 20,000

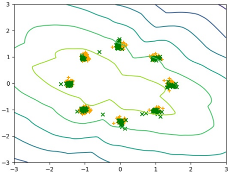
GP vs. No-GP Performance Comparison



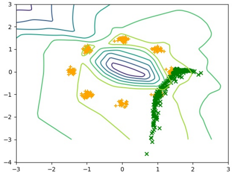
(a) With GP at iteration 30,000



(b) Without GP at iteration 30,000

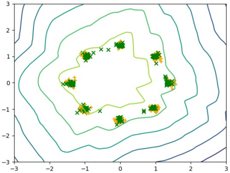


(a) With GP at iteration 40,000

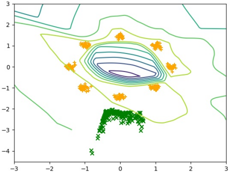


(b) Without GP at iteration 40,000

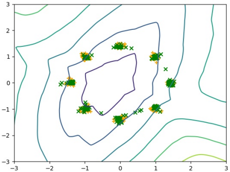
GP vs. No-GP Performance Comparison



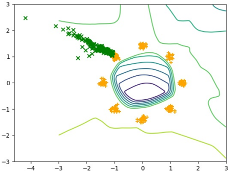
(a) With GP at iteration 50,000



(b) Without GP at iteration 50,000

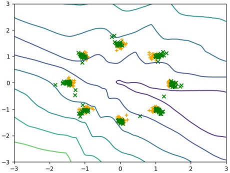


(a) With GP at iteration 60,000

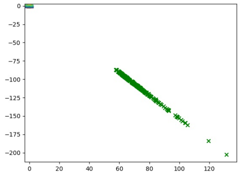


(b) Without GP at iteration 60,000

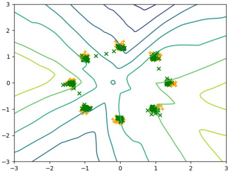
GP vs. No-GP Performance Comparison



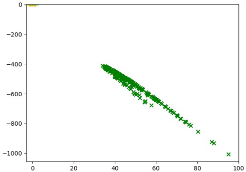
(a) With GP at iteration 70,000



(b) Without GP at iteration 70,000



(a) With GP at iteration 80,000



(b) Without GP at iteration 80,000

Outline

A Markov Chain Based Generative Model for Images

- Let the variable $\mathbf{x}$ denote an image. You could think of $\mathbf{x}$ as a tensor of shape $(3, H, W)$ where 3 stands for the number of color channels, and H and W for the height and the width of the image.

- Assume $\mathbf{x}$ is the output of a stochastic process and $\mathbf{p}(\mathbf{x})$ represents the probability that the process will output a specific image $\mathbf{x}$.

- About the stochastic process that would lead to the production of a given $\mathbf{x}$, we will assume for it to be a Markov Chain (which is a discrete-time version of a continuous-time Markov Process) that takes zero-mean isotropic Gaussian noise for its input and, **by denoising the input in stages**, leads to the output $\mathbf{x}$. By "stages", I mean one step at a time, which would allow for the extent of denoising to be carried out at each time step to be relatively small.

- For noise to be isotropic Gaussian, its covariance matrix must be $\sigma^2 I$ where I is the $d \times d$ Identity Matrix and σ a scalar that represents the isotropic variance in the d-dimensional data.

Markov Chain Based Generative Modeling (contd.)

- **So if we want to generate images by denoising isotropic Gaussian noise one small step at a time, one is faced with the question of deciding how much denoising to carry out at each time step.**
- **The question posed above is answered by creating another Markov chain that does the opposite of what's accomplished by the previous Markov chain — it will convert an image into isotropic Gaussian noise, again one small step at a time.**
- For the second Markov chain, let's assume for a moment that, by trial and error, we can figure out the "extent of noise to inject" at each time step such that, after T time steps, an image is converted into isotropic Gaussian noise. If we confine ourselves to using only isotropic Gaussian noise, the "extent of noise to inject" at each time step would be completely characterized by just two parameters: β for the variance and α for the mean. So all that we would need to know would be the values for these two parameters at each of the T time steps for the purpose of converting an image into noise.

Markov Chain Based Generative Modeling (contd.)

- At this point you might ask: How does knowing a value for α and β needed for the second Markov chain that progressively converts an image into noise help us figure out how much denoising to carry out at each time step in the first Markov chain?

- This is where you need a neural network. Using the known (α, β)-noise in going from one timestep to the next in the second Markov chain that progressively turns the image into isotropic Gaussian noise, the job assigned to the neural network would be to predict how much noise to take out during the transition for the corresponding time step in the first Markov chain for the purpose of denoising. I'll make this idea more precise in what follows.

- For creating a more precise description of what I have described so far, we need to do two things: (1) set the direction of the arrow of time for the two Markov chains; and (2) what mathematical notation to use for the two chains.

Markov Chain Based Generative Modeling (contd.)

- In order to conform to the notation used in the foundational papers, I'll refer to the two Markov processes as the Forward Markov Chain, denoted q-chain, and the Reverse Markov Chain, denoted p-chain.

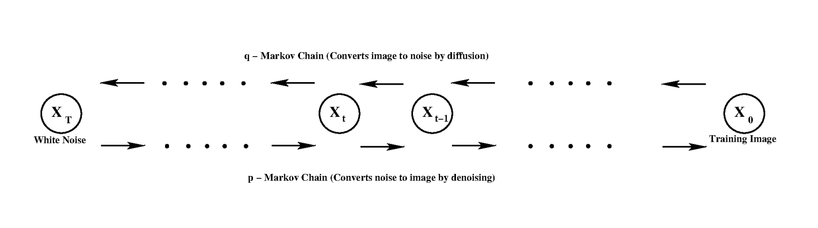
- The Forward Markov Chain will start at timestep 0 with an image from the training data and, by taking the image through noise-diffusing transitions, end up with isotropic Gaussian noise at timestep T. The Reverse Markov Chain will start at timestep T with zero-mean isotropic noise, take it through denoising Markov transitions to end up at timestep 0 in a recognizable image.

- I will use $\mathbf{q}(\mathbf{x}_t|\mathbf{x}_{t-1})$ for the transition probabilities for the Forward Markov Chain, and the notation $\mathbf{p}(\mathbf{x}_{t-1}|\mathbf{x}_t)$ for the transition probabilities for the Reverse Markov Chain. **Remember, that, despite its appearance, the transition probability $\mathbf{p}(\mathbf{x}_{t-1}|\mathbf{x}_t)$ conforms to the usual notation of $prob(next\_state \mid current\_state)$ for Markov chains because for this chain the arrow of time starts at $t = T$ and progresses to $t = 0$.**

Markov Chain Based Generative Modeling (contd.)

- Shown below are the two Markov chains, the `p-chain` for denoising and the `q-chain` for diffusion. Ah, I almost forgot to mention that the process of making the input image more and more like isotropic Gaussian noise in the `q-chain` is called diffusion.

- Next, I'll discuss how to link the state transition probabilities in the two chains and how to exploit that linkage through a neural network for effective denoising.



The `q-chain` is for diffusion, that is, for the purpose of making the image represented by $\mathbf{x}_0$ at right look more and more like isotropic Gaussian noise that is represented by $\mathbf{x}_T$ at left. And the `p-chain` is for learning how to progressively denoise the isotropic Gaussian noise at the left end of the pipeline in order to turn it into an image at the right end of the pipeline.

Markov Chain Based Generative Modeling (contd.)

- With the additional notation in place, I now go back to the question on Slide 130: **How to figure out how much denoising to carry out at each step in the `p-chain`?** As I mentioned there, we want a neural network to predict how much denoising to carry in going from time t to time $t-1$ in the `p-chain` based on how much noise was added in the `q-chain` when going from time $t-1$ to time t.

- Ideally, the extent of denoising in the `p-chain` in going from time t to time $t-1$ will equal the noise added in the `q-chain` in diffusing the image from time $t-1$ to time t.

- What that says that, at least notionally, in each cycle of training the neural network, we know what we want the neural network to predict. In other words, at least conceptually, we can define a "target" in each iteration of neural learning. As you know, in general, it is the "distance" between the target and what the neural network predicts that serves as the loss for the backpropagation step of SGD.

Markov Chain Based Generative Modeling (contd.)

- As it turns out, formulation of the loss function that has yielded high quality results with diffusion networks is more elaborate than what would be suggested by the previous bullet. The next section goes into formulating the loss function.

Outline

Starting with Maximum-Likelihood Estimation of the Parameters

- Since our goal is to generate images that appear to come from the same probability distribution that describes those in the training dataset, one could argue that our ultimate criterion for the training of the denoising `p-chain` defined in the previous section is the maximization of the probability associated with $\mathbf{x}_0$ emerging at the output of the `p-chain` when this image is fed at the input to the diffusing `q-chain`.

- That is, assuming that the image $\mathbf{x}_0$ is at the input to the `q-chain`, we want the learnable parameters in the denoising `p-chain` to be such that they associate the highest log-likelihood with the output of the `p-chain` to also be $\mathbf{x}_0$:

$$\widehat{\Theta}_{ML} \quad = \quad \underset{\Theta}{\operatorname{argmax}} \ \log p(\mathbf{x}_0) \tag{41}$$

  As to why such a criterion would constitute a maximum-likelihood estimate for the learnable parameters, see my "Holy Trinity" tutorial:

  https://engineering.purdue.edu/kak/Tutorials/Trinity.pdf

Maximum-Likelihood Estimation of the Parameters (contd.)

- If we had to translate the maximum-likelihood criterion of Eq. (41) directly into a loss function, we would need to change the RHS into something that would lend itself to achieving the same with a minimization (as opposed to a maximization) since that's what you do with neural learning – you minimize the loss.

- The standard way to do that is to express the RHS as negative log of the probability:

$$\mathcal{L} = -\log p(\mathbf{x}_0) \tag{42}$$

$$\widehat{\Theta}_{ML} = \underset{\Theta}{\operatorname{argmin}} \, \mathcal{L} \tag{43}$$

  where $\mathcal{L}$ is the loss.

- You might think of the negative-log-likelihood on the RHS in the first equation above as minimizing the logarithm of one minus the probability, which is the same thing as maximizing the probability. Recall that the logarithm is a monotonic function of its argument.

Maximum-Likelihood Estimation of the Parameters (contd.)

- Now we need to translate the log-likelihood criterion in the previous equation into something that is computationally feasible. Right off the bat, we realize that $\mathbf{x}_0$ is the final denoised image produced in the `p-chain` and its production depends obviously on all the *preceding* values $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_T$. More precisely speaking, the likelihood $p(\mathbf{x}_0)$ is a marginal of the joint probability distribution over all the data elements in the `p-chain`:

$$p(\mathbf{x}_0) \quad = \quad \int p\Big(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_T\Big) d\mathbf{x}_1 d\mathbf{x}_2 \ldots d\mathbf{x}_T \tag{44}$$

- The above can be expressed more compactly using the ':' notation as shown below, where the colon ':' has almost the same semantics as in Python:

$$p(\mathbf{x}_0) \quad = \quad \int p\Big(\mathbf{x}_{0:T}\Big) d\mathbf{x}_{1:T} \tag{45}$$

- As you will see, we can use the Markov property of the data to translate the joint distro shown above and also its role in the integral into a computational framework that actually works.

Joint Distros Over the Data in the Two Chains

- We first arrange the data sequences in the two Markov chains, `p-chain` and `q-chain`, in a temporal order that starts with the last item produced first:

$$\begin{array}{lcl} \text{q-chain:} & : & \mathbf{x}_T, \ \mathbf{x}_{T-1}, \ \ldots, \ \mathbf{x}_1, \ \mathbf{x}_0 \\ \text{p-chain:} & : & \mathbf{x}_0, \ \mathbf{x}_1, \ \ldots, \ \mathbf{x}_{T-1}, \ \mathbf{x}_T \end{array}$$

  As shown in the Figure on Slide 131, in the `q-chain`, the last item produced is $\mathbf{x}_T$ and, in the `p-chain`, the last item produced is $\mathbf{x}_0$.

- **In each chain, the last item produced depends probabilistically on all the preceding items.** Focusing first on the `q-chain`, we can express this dependence through the right-hand-side in the following equation:

$$q(\mathbf{x}_T, \mathbf{x}_{T-1}, \ldots, \mathbf{x}_1, \mathbf{x}_0) \quad = \quad q(\mathbf{x}_T \mid \mathbf{x}_{T-1}, \ldots, \mathbf{x}_1, \mathbf{x}_0) \cdot q(\mathbf{x}_{T-1}, \ldots, \mathbf{x}_1, \mathbf{x}_0) \qquad (46)$$

  The rewrite on the right-hand-side in the above equation follows directly from the Bayes' Theorem.

Joint Distros in the Two Chains (contd.)

- Invoking the Markov property that the conditional probability of a data element at time t given all the temporally *preceding* data elements at times $t-1$ through 0 depends only the data at the time instant $t-1$ and invoking the Bayes' rule recursively, we can write

$$
\begin{aligned}
q(\mathbf{x}_T, \mathbf{x}_{T-1}, \ldots, \mathbf{x}_1, \mathbf{x}_0) &= q(\mathbf{x}_T \mid \mathbf{x}_{T-1}, \ldots, \mathbf{x}_1, \mathbf{x}_0) \cdot q(\mathbf{x}_{T-1}, \ldots, \mathbf{x}_1, \mathbf{x}_0) \\
&= q(\mathbf{x}_T \mid \mathbf{x}_{T-1}) \cdot q(\mathbf{x}_{T-1} \mid \mathbf{x}_{T-2}, \ldots, \mathbf{x}_1, \mathbf{x}_0) \\
&= q(\mathbf{x}_T \mid \mathbf{x}_{T-1}) \cdot q(\mathbf{x}_{T-1} \mid \mathbf{x}_{T-2}), \ldots, q(\mathbf{x}_1 | \mathbf{x}_0) \cdot q(\mathbf{x}_0) \\
&= q(\mathbf{x}_0) \prod_{t=T}^{1} q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) \\
&= q(\mathbf{x}_0) \prod_{t=1}^{T} q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) \qquad (47)
\end{aligned}
$$

- For the `p-chain`, recognizing that its last element produced is $\mathbf{x}_0$ which depends on all of the *preceding* elements $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{T-1}, \mathbf{x}_T$, in that order, and using the Baye's rule we have the following rewrite for the joint probability over all the samples sequentially denoised:

$$
p(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{T-1}, \mathbf{x}_T) = p(\mathbf{x}_0 \mid \mathbf{x}_1, \ldots, \mathbf{x}_{T-1}, \mathbf{x}_T) \cdot p(\mathbf{x}_1, \ldots, \mathbf{x}_{T-1}, \mathbf{x}_T) \qquad (48)
$$

Joint Distros in the Two Chains (contd.)

- Applying the Markov property to the first conditional probability on the RHS of the previous equation and the Bayes' rule to the second, and doing so recursively, we get

$$\begin{aligned} p(\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{T-1}, \mathbf{x}_T) &= p(\mathbf{x}_0 \mid \mathbf{x}_1, \ldots, \mathbf{x}_{T-1}, \mathbf{x}_T) \cdot p(\mathbf{x}_1, \ldots, \mathbf{x}_{T-1}, \mathbf{x}_T) \\ &= p(\mathbf{x}_0 \mid \mathbf{x}_1) \cdot p(\mathbf{x}_1 \mid \mathbf{x}_2, \ldots, \mathbf{x}_{T-1}, \mathbf{x}_T) \\ &= p(\mathbf{x}_0 \mid \mathbf{x}_1) \cdot p(\mathbf{x}_1 \mid \mathbf{x}_2, \ldots, \mathbf{x}_{T-1} | \mathbf{x}_T) \cdot p(\mathbf{x}_T) \\ &= p(\mathbf{x}_T) \prod_{t=1}^{T} p(\mathbf{x}_{t-1} \mid \mathbf{x}_t) \qquad (49) \end{aligned}$$

- Using the same ':' notation you saw earlier in Eq. (45), the two joint probabilities can be expressed more compactly as

$$q(\mathbf{x}_{0:T}) = q(\mathbf{x}_0) \prod_{t=1}^{T} q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) \qquad (50)$$

$$p(\mathbf{x}_{0:T}) = p(\mathbf{x}_T) \prod_{t=1}^{T} p(\mathbf{x}_{t-1} \mid \mathbf{x}_t) \qquad (51)$$

Bringing the `q-chain` into the Marginal-Joint Distro Relationship for the `p-chain`

- **What I have shown in the last three slides is extremely elementary for Markov chains: Expressing the join distro as a product of the transition probabilities.** I did that mostly to refamiliarize you with the notation involved.

- Now I am going back to the relationship between the marginal and the joint in Eq. (45) and bring in the probabilities in the `q-chain` through what may seem like a backdoor (but with no liberties with the math involved). Eq. (45) can be rewritten as follows where we are multiplying and dividing by the same thing:

$$p(\mathbf{x}_0) \quad = \quad \int p\Big(\mathbf{x}_{0:T}\Big) \cdot \frac{q\Big(\mathbf{x}_{1:T}\big|\mathbf{x}_0\Big)}{q\Big(\mathbf{x}_{1:T}\big|\mathbf{x}_0\Big)} \cdot d\mathbf{x}_{1:T} \tag{52}$$

- **This rewrite will allow us to express the likelihood value on the left in a form that will have profound computational implications.**

Bringing in the q-chain (contd.)

- We new express Eq. (52) in the following form:

$$p(\mathbf{x}_0) = \int q\Big(\mathbf{x}_{1:T}|\mathbf{x}_0\Big) \cdot \frac{p\Big(\mathbf{x}_{0:T}\Big)}{q\Big(\mathbf{x}_{1:T}|\mathbf{x}_0\Big)} \cdot d\mathbf{x}_{1:T} \tag{53}$$

- Substituting for the joints from Eq. (50) and (51), we can now write

$$\begin{aligned} p(\mathbf{x}_0) &= \int q\Big(\mathbf{x}_{1:T}|\mathbf{x}_0\Big) \cdot \frac{p(\mathbf{x}_T)\prod_{t=1}^{T} p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)}{\prod_{t=1}^{T} q(\mathbf{x}_t \mid \mathbf{x}_{t-1})} \cdot d\mathbf{x}_{1:T} \\ &= \int q\Big(\mathbf{x}_{1:T}|\mathbf{x}_0\Big) \cdot p(\mathbf{x}_T) \cdot \prod_{t=1}^{T} \frac{p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)}{q(\mathbf{x}_t \mid \mathbf{x}_{t-1})} \cdot d\mathbf{x}_{1:T} \end{aligned} \tag{54}$$

- In case you are wondering as to what happened to the $q(\mathbf{x}_0)$ term in Eq. (50), note that the denominator in Eq. (53) is calling for the conditional-joint $q(\mathbf{x}_{1:T}|\mathbf{x}_0)$ and NOT just the joint $q(\mathbf{x}_{0:T})$. Using Bayes' Theorem, we know that $q(\mathbf{x}_{0:T}|\mathbf{x}_0) = q(\mathbf{x}_{0:T})/q(\mathbf{x}_0)$.

Computational Ramifications of the New Form for the Likelihood

- In terms of computational feasibility, there is a sea change between the form for the likelihood shown in Eq. (44) and the form shown in Eq. (54).

- Eq. (44) was NOT computationally feasible. The joint and the transition probabilities in the `p-chain` depend on how much denoising is carried out at each time step. However, those decisions cannot be made in a vacuum.

- Eq. (54), on the other hand, is a probabilistic integral of the sort I have described in my "Monte-Carlo" tutorial:

  https://engineering.purdue.edu/kak/Tutorials/MonteCarloInBayesian.pdf

- Since the image diffusion in `q-chain` is directly under our control, we get to decide what value we want to use for the transition probabilities $q(\mathbf{x}_t \mid \mathbf{x}_{t-1})$ and, therefore, also for the joint $q(\mathbf{x}_{1:T}|\mathbf{x}_0)$.

Computational Ramifications (contd.)

- Representing the likelihood in the manner shown in Eq. (54) has another hugely important ramification whose exact nature depends on how you plan to carry out the denoising steps in the `p-chain`.

- We could carry out denoising with a neural network whose job is to predict the next denoised version from the current denoised version according to the transition probability $p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$. Remember, as shown on Slide 131, for the `p-chain`, the time step $t-1$ comes *after* the time step t.

- Now if assume that the neural network can come pretty close to matching the prediction probability $p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)$ given what we have used for the diffusion transition probability $q(\mathbf{x}_t \mid \mathbf{x}_{t-1})$ at every time step, one can show that the integral in Eq. (54) can be computed exactly by using just one of the T time steps. [**Under these conditions, the `q-chain` and the `p-chain` constitute a quasi-static reversible process in statistical physics. I believe that stipulates that all the transition probabilities will be the same and, since the joint `q-distribution` in Eq. (54) can be expressed as a product of transition probabilities with the help of the Markovian assumption, that integration can be carried out exactly using only one of the T time steps. See the discussion by Sohl-Dickstein et al. (2015). They base their arguments on the papers by Jarzynski (2011) and Spinney and Ford (2012).**]

Expressing the Loss with KL-Divergence

- Substituting Eq. (54) in the RHS of Eq. (42) gives us a Loss function for the neural learning required in the `p-chain`. For computational reasons, it is more convenient to express the loss using KL-Divergence.

- To derive the KL-Divergence based expression for the loss, we first note that the integral in Eq. (54) is nothing but the expectation of the product of the second and the third terms with respect to the q-distribution. Therefore, Eq. (54) can be expressed more compactly as

$$p(\mathbf{x}_0) \quad = \quad \mathcal{E}_q\left[p(\mathbf{x}_T) \cdot \prod_{t=1}^{T} \frac{p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)}{q(\mathbf{x}_t \mid \mathbf{x}_{t-1})} \right] \tag{55}$$

- Substituting the above in Eq. (42) for the loss, we can write

$$\mathcal{L} \quad = \quad \mathcal{E}_q\left[-\log p(\mathbf{x}_T) \; - \; \sum_{t=1}^{T} \log \frac{p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)}{q(\mathbf{x}_t \mid \mathbf{x}_{t-1})} \right] \tag{56}$$

Expressing the Loss with KL-Divergence (contd.)

- I'll now follow the steps in Appendix A of the 2020 paper "Denoising Diffusion Probabilistic Models" by Ho et al. to convert the formula shown above into the loss function we need. To that end, we first separate out the timestep $t = 1$ from the summation:

$$\mathcal{L} \;=\; \mathcal{E}_q\left[-\log p(\mathbf{x}_T) \;-\; \sum_{t=2}^{T} \log \frac{p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)}{q(\mathbf{x}_t \mid \mathbf{x}_{t-1})} \;-\; \log \frac{p(\mathbf{x}_0 \mid \mathbf{x}_1)}{q(\mathbf{x}_1 \mid \mathbf{x}_0)} \right] \tag{57}$$

- Let's now focus on the denominator $q(\mathbf{x}_t|\mathbf{x}_{t-1})$ in the second term. We take advantage of the fact that $\mathbf{x}_0$ stands for an image from the training dataset and that the noise added at any time step in the `q-chain` is independent of the image itself. Therefore we can write:

$$\begin{aligned}
q(\mathbf{x}_t|\mathbf{x}_{t-1}) &= \frac{q(\mathbf{x}_t, \mathbf{x}_{t-1})}{q(\mathbf{x}_{t-1})} = \frac{q(\mathbf{x}_{t-1}, \mathbf{x}_t)}{q(\mathbf{x}_{t-1})} \\
&= \frac{q(\mathbf{x}_{t-1}, \mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)} \\
&= \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) \cdot q(\mathbf{x}_t|\mathbf{x}_0)}{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}
\end{aligned} \tag{58}$$

Expressing the Loss with KL-Divergence (contd.)

- Substituting Eq. (58) in Eq. (57), we can write

$$
\begin{aligned}
\mathcal{L} &= \mathcal{E}_q\left[-\log p(\mathbf{x}_T) - \sum_{t=2}^{T} \log \frac{p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)} \cdot \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)} - \log \frac{p(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)}\right] \\
&= \mathcal{E}_q\left[-\log p(\mathbf{x}_T) - \sum_{t=2}^{T} \log \frac{p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)} - \sum_{t=2}^{T} \log \frac{q(\mathbf{x}_{t-1}|\mathbf{x}_0)}{q(\mathbf{x}_t|\mathbf{x}_0)} - \log \frac{p(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)}\right] \\
&= \mathcal{E}_q\left[-\log p(\mathbf{x}_T) - \sum_{t=2}^{T} \log \frac{p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)} - \sum_{t=2}^{T} \log q(\mathbf{x}_{t-1}|\mathbf{x}_0) + \sum_{t=2}^{T} \log q(\mathbf{x}_t|\mathbf{x}_0) - \log \frac{p(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)}\right] \\
&= \mathcal{E}_q\left[-\log p(\mathbf{x}_T) - \sum_{t=2}^{T} \log \frac{p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)} - \log q(\mathbf{x}_1|\mathbf{x}_0) + \log q(\mathbf{x}_T|\mathbf{x}_0) - \log \frac{p(\mathbf{x}_0|\mathbf{x}_1)}{q(\mathbf{x}_1|\mathbf{x}_0)}\right] \\
&= \mathcal{E}_q\left[-\log \frac{p(\mathbf{x}_T)}{q(\mathbf{x}_T \mid \mathbf{x}_0)} - \sum_{t=2}^{T} \log \frac{p(\mathbf{x}_{t-1} \mid \mathbf{x}_t)}{q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0)} - \log p(\mathbf{x}_0|\mathbf{x}_1)\right] \qquad (59)
\end{aligned}
$$

[Of the five RHS expressions, in going from the first to the second, all we have done is to use the identity $\log A \cdot B = \log A + \log B$ in the 2^{nd} term of the first. Along the same lines, in going from the second expression to the third, all we have done is to express $\log A/B$ as $\log A - \log B$ in the third term of the second expression. In going from the third expression to the fourth on the RHS, when you expand out the 3^{rd} and the 4^{th} summations in the third expression, only the two items that are shown survive. In going from the fourth RHS expression to the last, we notice that the 3^{rd} term in the fourth expression is canceled out by the denominator in the last term and the 4^{th} term is merged with the first.]

Expressing the Loss with KL-Divergence (contd.)

- Now we are ready to express the loss through KL-Divergence. Let me first recall the definition of KL-Divergence presented in the second section of this lecture: Given two distributions P and Q, one of them to be construed as the true distribution and the other as an estimate for the true, the KL-Divergence between the two is given by:

$$d_{KL}(P, Q) \quad = \quad \sum_{i=1}^{N} P(x_i) \log \frac{P(x_i)}{Q(x_i)} \quad = \quad - \sum_{i=1}^{N} P(x_i) \log \frac{Q(x_i)}{P(x_i)} \tag{60}$$

- $d_{KL}(P, Q)$ is obviously the expectation of the ratios $\log \frac{P(x_i)}{Q(x_i)}$ with respect to the P distribution (putatively, the true distro in this def).

- Drawing a parallel between the Loss expression in Eq. (59) and the definition of KL-divergence shown above, in the loss expression we are averaging the ratio of the two distributions p and q with respect to q. So we treat q as the true and p as an approximation to q. Therefore, we can express the loss in Eq. (59) as

$$\mathcal{L} \quad = \quad \mathcal{E}_q \left[d_{KL}\Big(q(\mathbf{x}_T|\mathbf{x}_0), p(\mathbf{x}_T)\Big) \; + \; \sum_{t=2}^{T} d_{KL}\Big(q(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0), p(\mathbf{x}_{t-1}|\mathbf{x}_t)\Big) \; - \; \log p(\mathbf{x}_0|\mathbf{x}_1) \right] \tag{61}$$

Outline

Using Gaussian Noise for Diffusion

- A multivariate Gaussian distribution (also known as a Normal distribution) for data points that reside in a d-dimensional space is given by

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{d/2}\,|\Sigma|^{1/2}} e^{-\frac{1}{2}(\mathbf{x}-\mu)^T \Sigma^{-1}(\mathbf{x}-\mu)} \tag{62}$$

  where $|\Sigma|$ is the determinant of the $d \times d$ covariance matrix Σ, which must be symmetric and positive definite. We think of each data point $\vec{\mathbf{x}}$ as a d-dimensional *column* vector and μ as a similar vector for the mean of all the data.

- A commonly used notation for a Gaussian distribution is

$$p(\mathbf{x}) \quad = \quad \mathcal{N}(\mathbf{x}; \mu, \mathbf{\Sigma}) \tag{63}$$

  and the fact of drawing a random vector $\mathbf{x}$ from a Gaussian distribution expressed as

$$\mathbf{x} \quad \sim \quad \mathcal{N}(\mathbf{x}; \mu, \mathbf{\Sigma}) \tag{64}$$

- A Gaussian distribution is referred to as a Standard Normal distribution if all its component dimensions are independent and, for each dimension, we have zero mean and unit variance, as shown next.

Using Gaussian Noise for Diffusion (contd.)

- Here is an analytic form for a Standard Normal Distribution for a d-dimensional space:

$$p(\mathbf{x}) \;=\; \frac{1}{(2\pi)^{d/2}} e^{-\frac{1}{2}\mathbf{x}^T\mathbf{x}} \;=\; \mathcal{N}(\mathbf{x}; \mathbf{0}, \mathbf{I}) \tag{65}$$

- We'll assume that the noise that is added at each forward transition in the `q-chain` is described by the following Gaussian:

$$\mathbf{q}(\mathbf{x}_t|\mathbf{x}_{t-1}) \;=\; \mathcal{N}(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t\mathbf{I}) \tag{66}$$

  Recall that the forward transitions constitute the Diffusion Markov Process in which we start adding noise to the training image at timestep 0, continue this noise additions through the Markov transitions, until at timestep T the image is completely transformed into isotropic unit variance Gaussian noise $\mathcal{N}(\mathbf{x}; \mathbf{0}, \mathbf{I})$.

- Note β_t for the variance of the noise added at each transition. This is referred to as the Noise Schedule for the Diffusion Process. Also note that the mean value of the added noise at each transition depends both on Noise Schedule at that t and also the previous value of $\mathbf{x}_{t-1}$.

Using Gaussian Noise for Diffusion (contd.)

- If you are wondering about the mysterious sounding Noise Schedule defined by the noise variances $\{\beta_t | t = 1, \ldots, T-1\}$, the idea is that, starting at $t = 0$, these variances would be sufficient to transform the distribution that describes the training images into an isotropic Gaussian distribution $\mathcal{N}(\mathbf{x}_T; \mathbf{0}, \mathbf{I})$. If we succeed in doing so, then there is a chance that, for the Denoising Process in the reverse direction, we could incrementally transform pure isotropic Gaussian noise at timestep T and turn it into an image at timestep 0.

- For a deeper understanding of the relationship between the means and the variances in Eq. (66) used for the Forward Transitions, and of the Noise Schedule values used typically for β_t, you'll have to look through the three diffusion related papers on Slide 5 in Preamble.

- While the amount of noise we add during the forward diffusion process at each transition is under our control and would be given by Eq. (66), **how much noise to subtract at each corresponding reverse transition would be the job of a neural network**.

Computational Ramifications of Using Gaussian Noise

- Continuing with the thought in the last bullet on the previous slide, from the amount of noise added during the forward process, it will be the neural network's job to figure out how much noise to "subtract" in the reverse process in order to minimize the overall loss defined in Eq. (61) on Slide 149.

- From the standpoint of practical implementation, we now take advantage of the fact that, with Gaussian noise, an arbitrary number of forward diffusion transitions, each given by Eq. (66), can be combined into a single calculation. That is, starting with $t = 0$, instead of making several $\mathbf{q}(\mathbf{x}_t|\mathbf{x}_{t-1})$ transitions up to some arbitrary timestep t , we can do directly from $t = 0$ to the timestep t with the help of the following formula:

$$\mathbf{q}(\mathbf{x}_t|\mathbf{x}_0) = \mathcal{N}(\mathbf{x}_t; \sqrt{\bar{\alpha}_t}\mathbf{x}_0, (1-\bar{\alpha}_t)\mathbf{I}) \tag{67}$$

$$\mathbf{x}_t = \sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon \tag{68}$$

See the next slide for what the symbols ϵ, α_t, $\bar{\alpha}_t$ stand for.

Computational Ramifications of Gaussian Noise (Contd.)

- In Eq. (67) and (68) shown on the previous slide, we have

$$\epsilon \sim \mathcal{N}(\mathbf{0}, \mathbf{I}) \quad (69)$$

$$\alpha_t = 1 - \beta_t \quad (70)$$

$$\bar{\alpha_t} = \prod_{s=0}^{t} \alpha_s \quad (71)$$

- In the formulas shown above and on the previous slide, $\mathbf{x}_0$ is the training image and ϵ is the Gaussian noise added to this image at timestep $t = 0$, meaning at the beginning of the diffusion process.

- Eq. (68) is a thing of beauty unto itself. It makes explicit what exactly happens to an input image as it is subject to successive applications of diffusion.

[In casual conversations about diffusion it is not uncommon to hear that diffusion means "adding more and more noise" to a training image. But there is a serious problem with that mental imagery: Literally adding random noise to an image will not destroy the "structure" of the pixel-to-pixel variations in the values of the color in the input training image. It may become difficult to see those variations with the naked eye as more and more noise is "added", but they would always be there. On the other hand, Eq. (68) tells us that the input image is being progressively destroyed by its multiplication with a coefficient that becomes ever smaller as the timestep approaches T. Also note that the relationship between the mean and variance in Eq. (67) is dictated by the fact that as we take the "signal" away from the input image by the multiplicative effect in Eq. (68), we want to transfer that energy into the variance of the noise so that, on the average, the overall variance remains unchanged.]

Computational Ramifications of Gaussian Noise (Contd.)

- Previously, we expressed the Noise Schedule in terms of β_t, the variance of the noise to be added at each timestep t. From the relationship in Eq. (71), we could also express the Noise Schedule directly in terms of α values.

- So far we have the forward diffusion process under control. We know know that, if we so wished, we could choose a random timestep in the $(1, T)$ range and directly estimate the diffused version $\mathbf{x}_t$ of the training image at that point.

- The question now is: How does the reverse denoising process learn from the forward process especially if we want each iteration of training to involve a single arbitrarily chosen timestep t at which we want to calculate the result of forward diffusion? We want the reverse denoising process to learn from the forward diffusion process how much noise to subtract at the corresponding transition going in the opposite direction so that, ultimately, the final denoised result at $t = 0$ would be a recognizable image.

Computational Ramifications of Gaussian Noise (Contd.)

- Keep in mind that when the forward diffusion process transitions from timestep $t-1$ to t in the q-chain, that corresponds to the reverse denoising process transitioning from t to $t-1$ in the p-chain. Let the job of the neural network be to learn the network parameters θ to help us estimate the mean $\mu_\theta(\mathbf{x}_t)$ and the covariance $\Sigma_\theta(\mathbf{x}_t)$ that would allow us to predict the following transition probability in the p-chain:

$$\mathbf{p}_\theta(\mathbf{x}_{t-1}|\mathbf{x}_t) \quad = \quad \mathcal{N}(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t), \Sigma_\theta(\mathbf{x}_t)) \tag{72}$$

- Obviously, the parameters $\mu_\theta(\mathbf{x}_t)$ and $\Sigma_\theta(\mathbf{x}_t)$ must be learned subject to the minimization of the loss in Eq. (61) on Slide 149. Relevant to the transition in question, that requires to minimize the KL-Divergence between the conditional probability shown above and the posterior conditional probability $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ in the q-chain.

- If the forward transition probabilities in the q-chain are given by Eq. (67), one can use the Bayes' Rule to write the formula on the next slide for the posterior conditional probability $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$.

Computational Ramifications of Gaussian Noise (Contd.)

- Here is a formula for the posterior in the q-chain:

$$\mathbf{q}(\mathbf{x}_{t-1}|\mathbf{x}_t, \mathbf{x}_0) = \mathcal{N}\left(\mathbf{x}_{t-1}; \tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0), \tilde{\beta}_t \mathbf{I}\right) \tag{73}$$

where

$$\tilde{\mu}_t(\mathbf{x}_t, \mathbf{x}_0) = \frac{\sqrt{\bar{\alpha}}_{t-1}\beta_t}{1-\bar{\alpha}_t} \cdot \mathbf{x}_0 + \frac{\sqrt{\alpha_t}(1-\bar{\alpha}_{t-1})}{(1-\bar{\alpha}_t)} \tag{74}$$

$$\tilde{\beta}_t = \frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t} \cdot \beta_t \tag{75}$$

- Note that the parameters with the tildes, $\tilde{\mu}_t$ and $\tilde{\beta}_t$ are for "going backwards" in the q-chain itself, in the sense that these parameters help us create an estimate for posterior probability for a given transition probability in the forward direction.

- On the other hand, the barred parameter, $\bar{\alpha}_t$, as defined earlier in Eq. (71) on Slide 155 is a measure of cumulative variances of the noise transitions in the forward chain over the timesteps 0 through t.

- The loss for training the network is the KL-Divergence between two Gaussians, one for the predicted form in the p-chain as given by Eq. (72) and the other for the posterior in the q-chain as given by Eq. (73)

Computational Ramifications of Gaussian Noise (Contd.)

- Note that the loss form shown in Eq. (61) on Slide 149 also requires this KL-Divergence to be averaged with respect to the q-probabilities. In practice, that can be done by averaging over the instances in a batch.

- For a derivation of the posterior probability shown in Eqs. (73) through (75) and for a discussion of several importance practical aspects that go into the minimization of the loss, see the paper "Improved Denoising Diffusion Probabilistic Models" Nichol and Dhariwal.

- Asking a neural network to just predict the mean and the variance in a probability distribution is strongly evocative of how a VAE works. [**But there is a very important difference between a VAE neural network and the network we need for diffusion based modeling: A VAE is typically used for creating a reduced dimensional representation of an input image and it assumes that we can model each input image by a Gaussian distribution whose mean μ is the reduced-dimensionality output that you want and that, naturally, has an associated standard-deviation σ. Although a user of VAE will typically throw away the σ, it has a critical role to play during the training of the VAE network: In each cycle of training, you draw an ϵ from a standard normal distribution and you construct "$\mu + \epsilon \cdot \sigma$" as an exemplar of the output that should look like the input. The difference between the input and the exemplar output thus constructed can be turned into a loss. On the other hand, for diffusion based modeling, during training, we do NOT construct a putative exemplar for an image-like output using Eq. (68). All we do is to make the network better at estimating the means and the standard deviations of the p-chain transition probs through the minimization of the loss in Eq. (61).**]

Computational Ramifications of Gaussian Noise (Contd.)

- To summarize, during each iteration of training in which we focus on a single randomly chosen timestep in the $(1, T-1)$ range, all we want to do is to estimate the mean and the variance in the p-chain transition probability corresponding to that timestep.

- In other words, the training iterations do not involve reconstructing the denoised versions of $\mathbf{x}_t$ that you see in Eq. (68). It is only after a network has been trained in the manner described above that we can deploy the p-chain using all of its T timesteps to convert Gaussian isotropic noise at its input into a recognizable image at the other end.

- We still have the issue of how to set a value for T, the number of timesteps allowed. In general, the larger the value of T, the greater the ability of the trained model to represent a larger diversity in the training dataset. For a given T, the rate at which the the model achieves convergence also depends on the Noise Schedule.

α-β Related Quantities in the Calculations

- Calculation of the forward transition probabilities in the q-chain in Eq. (67), drawing a sample from it in Eq. (68), and, subsequently, computing the posteriors in Eq. (73) in the q-chain again, involves calculating various forms involving β and $\alpha = 1 - \beta$ coefficients in Eqs. (70), (71), (74) and (75).

- As mentioned earlier, each training cycle involves randomly choosing a value for the timestep *t*. For that timestep chosen, you must calculate the α-β forms required Eqs. (70), (71), (74) and (75). Shown on the next slide is a sample calculation of these forms.

- The calculation of the α-β related values begins with first setting the values of the β coefficients. Recall that *beta* is the variance of the noise sample drawn from a standard normal distribution. Using the recommended values for β_{start} and β_{end} as shown at the beginning of the script, we set the values of β's by diving the β-range by the number of timesteps.

α-β Related Quantities in the Calculations (contd.)

- Subsequently, we set the values for the α's as $1 - \beta$. The values of the products of α's, as required by Eq. (71), are obtained by invoking the `numpy.cumprod` function, which stands for "cumulative product of elements along the specified axis".

```
##  alpha_beta_calc.py

import numpy as np
np.set_printoptions(precision=3)                                                ## just for showing the values on this slide

num_timesteps  = 10                                                             ## IMPORTANT:  This is just for demo purposes here. Typically, num_timesteps = 1000

beta_start = 0.0001
beta_end =   0.02
betas = np.linspace( beta_start, beta_end, num_timesteps )
print("\n\nbetas: ", betas )                                                    ##  [0.000 0.002 0.005 0.007 0.009 0.011 0.013 0.016 0.018 0.02 ]

print("\n\n\n\nSome basic alpha related values:")
alphas = 1.0 - betas
alphas_cumprod = np.cumprod(alphas, axis=0)
alphas_cumprod_prev = np.append(1.0, alphas_cumprod[:-1])
alphas_cumprod_next = np.append(alphas_cumprod[1:], 0.0)

print("\nalphas: ", alphas)                                                     ##  [1.000 0.998 0.995 0.993 0.991 0.989 0.987 0.984 0.982 0.98 ]
print("\nalphas_cumprod: ", alphas_cumprod)                                     ##  [1.000  0.998 0.993 0.986 0.978 0.967 0.954 0.939 0.922 0.904]
print("\nalphas_cumprod_prev: ", alphas_cumprod_prev)                           ##  [1.000 1.000 0.998 0.993 0.986 0.978 0.967 0.954 0.939 0.922]
print("\nalphas_cumprod_next: ", alphas_cumprod_next)                           ##  [0.998 0.993 0.986 0.978 0.967 0.954 0.939 0.922 0.904 0.000]

print("\n\n\n\nalpha related values needed for calculating q-chain transitions q(x_t | x_{t-1})")
sqrt_alphas_cumprod = np.sqrt(alphas_cumprod)
sqrt_one_minus_alphas_cumprod = np.sqrt(1.0 - alphas_cumprod)
log_one_minus_alphas_cumprod = np.log(1.0 - alphas_cumprod)
sqrt_recip_alphas_cumprod = np.sqrt(1.0 / alphas_cumprod)
sqrt_recipm1_alphas_cumprod = np.sqrt(1.0 / alphas_cumprod - 1)

print("\nsqrt_alphas_cumprod: ", sqrt_alphas_cumprod)                           ##  [1.    0.999 0.997 0.993 0.989 0.983 0.977 0.969 0.96  0.951]
print("\nsqrt_one_minus_alphas_cumprod: ", sqrt_one_minus_alphas_cumprod)       ##  [0.01  0.049 0.083 0.117 0.15  0.183 0.215 0.247 0.279 0.31 ]
print("\nlog_one_minus_alphas_cumprod: ", log_one_minus_alphas_cumprod)         ##  [-9.21  -6.028 -4.973 -4.297 -3.797 -3.401 -3.074 -2.795 -2.553 -2.341]
print("\nsqrt_recip_alphas_cumprod: ", sqrt_recip_alphas_cumprod)               ##  [1.    1.001 1.003 1.007 1.011 1.017 1.024 1.032 1.041 1.052]
print("\nsqrt_recipm1_alphas_cumprod: ", sqrt_recipm1_alphas_cumprod)           ##  [0.01  0.049 0.083 0.117 0.151 0.186 0.22  0.255 0.29  0.326]
```

(Continued on the next slide .....)

α-β Related Quantities in the Calculations (contd.)

(...... continued from the previous slide)

```
print("\n\n\n\nalpha calculations for posterior q(x_{t-1} | x_t, x_0):")
posterior_variance = ( betas * (1.0 - alphas_cumprod_prev) / (1.0 - alphas_cumprod) )
# log calculation clipped because the posterior variance is 0 at the
# beginning of the diffusion chain.
posterior_log_variance_clipped = np.log( np.append(posterior_variance[1], posterior_variance[1:]) )
posterior_mean_coef1 = ( betas * np.sqrt(alphas_cumprod_prev) / (1.0 - alphas_cumprod) )
posterior_mean_coef2 = ( (1.0 - alphas_cumprod_prev)  *  np.sqrt(alphas)  / (1.0 - alphas_cumprod) )

print("\nposterior_variance: ", posterior_variance)
                                              ## [0.000e+00 9.586e-05 1.575e-03 3.425e-03 5.426e-03 7.506e-03 9.633e-03 1.179e-02 1.397e-02 1.617e-02]
print("\nposterior_log_variance_clipped: ", posterior_log_variance_clipped)   ## [-9.253 -9.253 -6.453 -5.677 -5.216 -4.892 -4.643 -4.44  -4.271 -4.125]
print("\nposterior_mean_coef1: ", posterior_mean_coef1)                       ## [1.    0.959 0.653 0.493 0.396 0.331 0.284 0.249 0.222 0.2  ]
print("\nposterior_mean_coef2: ", posterior_mean_coef2)                       ## [0.    0.041 0.347 0.507 0.604 0.669 0.716 0.751 0.778 0.8  ]
```

Calculating the Forward Transitions in the q-Chain

- The second function shown below is the function in the DDPM code library at GitHub that implements Eq. (68) on Slide 154 for calculating the sample $\mathbf{x}_t$ at a given timestamp t and for a given starting $\mathbf{x}_0$ represented by `x_start` in the code. Remember, $\mathbf{x}_0$ is the training image that is fed into the q-chain at timestep 0. The value of the parameter *noise* in the header of the second function is ϵ in Eq. (68). The first function, `_extract_into_tensor()` is called by the second function.

```
def _extract_into_tensor(arr, timesteps, broadcast_shape):
    """
    Source: https://github.com/lucidrains/denoising-diffusion-pytorch
    Extract values from a 1-D numpy array for a batch of indices.
    :param arr: the 1-D numpy array.
    :param timesteps: a tensor of indices into the array to extract.
    :param broadcast_shape: a larger shape of K dimensions with the batch
                            dimension equal to the length of timesteps.
    :return: a tensor of shape [batch_size, 1, ...] where the shape has K dims.
    """
    res = torch.from_numpy(arr).to(device=timesteps.device)[timesteps].float()
    while len(res.shape) < len(broadcast_shape):
        res = res[..., None]
    return res.expand(broadcast_shape)


def q_sample(self, x_start, t, noise=None):
    """
    Source: https://github.com/lucidrains/denoising-diffusion-pytorch
    The value returned is a sample from q(x_t | x_0)
    """
    if noise is None:
        noise = torch.randn_like(x_start)
    assert noise.shape == x_start.shape
    return ( _extract_into_tensor( self.sqrt_alphas_cumprod, t, x_start.shape)   *   x_start    +
             _extract_into_tensor( self.sqrt_one_minus_alphas_cumprod, t, x_start.shape)   *   noise
    )
```

Calculating the Posterior Probabilities in the q-Chain

- Shown below is the function in the same DDPM library that calculates the estimates for the posterior mean, $\tilde{\mu}_t$, and the posterior variance, $\tilde{\beta}_t$, in the q-chain based on the formulas in Eqs. (74) and (75) on Slide 158. The function `_extract_into_tensor()` is the same as defined on the previous slide.

```
def q_posterior_mean_variance(self, x_start, x_t, t):
    """
    Source: https://github.com/lucidrains/denoising-diffusion-pytorch

    Compute the mean and variance associated with the posterior in the q-chain:
        q(x_{t-1} | x_t, x_0)
    In the q-chain, the forward transitions look like q(x_t | q_{t-1}).  We can use
    Bayes' Rule to derive an expression for the posterior q(x_{t-1} | x_t, x_0) from
    the forward transition probabilities.  See Slide 158.
    """

    assert x_start.shape == x_t.shape
    posterior_mean = (
        _extract_into_tensor(self.posterior_mean_coef1, t, x_t.shape) * x_start
        + _extract_into_tensor(self.posterior_mean_coef2, t, x_t.shape) * x_t
    )
    posterior_variance = _extract_into_tensor(self.posterior_variance, t, x_t.shape)
    posterior_log_variance_clipped = _extract_into_tensor(  self.posterior_log_variance_clipped, t, x_t.shape  )
    assert (
        posterior_mean.shape[0]
        == posterior_variance.shape[0]
        == posterior_log_variance_clipped.shape[0]
        == x_start.shape[0]
    )
    return posterior_mean, posterior_variance, posterior_log_variance_clipped
```

Outline

The GenerativeDiffusion Module in DLStudio

- The diffusion based modeling code in DLStudio is in its co-class `GenerativeDiffusion`. Most of the code you see in this file was drawn from the sources at OpenAI's GitHub project "Improved Diffusion": https://github.com/openai/improved-diffusion

  [**The source code at the above website is the implementation for the work described in the paper "Improved Denoising Diffusion Probabilistic Models" by Nichol and Dhariwal that you can download from:** https://arxiv.org/pdf/2102.09672.pdf]

- The GitHub code library consists of roughly 3900 lines of Python distributed over 10 files in three different directories. The code you'll see in `GenerativeDiffusion` contains around 750 lines of that code.

- Despite the fact that the class and function names in the 750 lines of OpenAI code presented in `GenerativeDiffusion` are the same as in the original code, some of them have undergone significant refactoring.

  [**For example, the OpenAI code uses** `GaussianDiffusion` **class through a subclass named** `SpacedDiffusion` **that is defined in a utility file called** `script_util.py`. **I got rid of the** `SpacedDiffusion` **class and incorporated that functionality in a rather small function inside the** `GaussianDiffusion` **class.**]

DLStudio's GenerativeDiffusion (contd.)

- If you just want to look at the code in GenerativeDiffusion:

  https://engineering.purdue.edu/kak/distDLS/GenerativeDiffusion-2.4.2_CodeOnly.html

- The main classes in GenerativeDiffusion are:

  GaussianDiffusion: This provides the basic functionality for invoking the forward Markov transitions on the training images by incrementally injecting isotropic Gaussian noise into them.

  AttentionBlock: This class in GenerativeDiffusion is probably has the most detailed comments that explain each of the steps. For that reason, I am not going to say anything further about it here.

  UNetModel: The is the workhorse class in GenerativeDiffusion for learning.

  mUNet: The is a "remote possibility" replacement for UNetModel. I'd like to give it a try for educational purposes. However, its hyperparameter tuning for its usage will take time.

- The UNetModel class contains the following as its inner classes: TimestepBlock, Upsample, Downsample, and ResBlock.

The GaussianDiffusion Class

The main functions in the GaussianDiffusion class are:

q_sample() This function returns $\mathbf{x}_t$ as defined in Eq. (68) on Slide 154.

q_posterior_mean_and_variance() After you have carried out in the q-chain a forward calculation of $\mathbf{x}_t$ for timestep t, you must use the Bayes' Rule to estimate the parameters of the posterior conditional probability $q(\mathbf{x}_{t-1}|\mathbf{x}_t)$ so that it can be used for setting up the target for training the neural network in the p-chain. This function returns the mean and the variance of the posterior distribution according to the formulas in Eqs. (73) - (75) on Slide 158.

predict_xstart_in_pchain() Using the posterior distribution as returned by the previous function, this function constructs a posterior estimate for $\mathbf{x}_0$ for the $t-1$ to t transition in the p-chain. Note that xstart in the code is the same thing as the training image $\mathbf{x}_0$. Therefore, this function gives us the best possible timestep t approximation to the input training image.

The GaussianDiffusion Class (contd.)

training_losses This function calculates the mean-square error between the expected value for $\mathbf{x}_0$ as provided for the posterior distribution at timestep t and the output of the neural network for the same.

The following additional functions in the GaussianDiffusion class are used for image generation **after** the model has been trained.

p_sampler_for_image_generation() After the model is trained, you call this function that takes for input isotropic Gaussian noise and takes it through each of the $T-1$ transitions to generate a new image.

p_sample() This is a helper function called by the previous function.

The first four functions are used in the script RunCodeForDiffusion.py in the ExamplesDiffusion directory. And the last two in the script GenerateNewImageSamples.py in the same directory.

The UNetModel Class

- As stated previously, the job of the neural network is to learn from each $t-1$ to t transition in the q-chain how to reverse the diffusing effect through a corresponding denoising t to $t-1$ transition in the p-chain.

- The neural network used for that purpose is UNetModel. For a description of the network, see the OpenAI paper cited on Slide 167.

- For a brief summary of the network description, it is obviously a UNet as its name implies. Each layer of the UNet is made up a user-specified number of ResBlock units. The ResBlock itself is patterned after the building-block elements in the famous ResNet.

- As is commonly the case with UNets, the input image is progressively downsampled in the Encoder side of the UNet, as more and more information is thrown into the channel dimensions, and subsequently upsampled in the Decoder side with a corresponding reduction in the number of channels.

The ExamplesDiffusion Directory

- The `ExamplesDiffusion` directory in the DLStudio distro contains the three scripts listed below that you can use for your own experiments in denoising diffusion. **You have to run all three scripts for any single demonstration.** Make sure you have read the **README** in that directory for a description of how to execute the scripts.

RunCodeForDiffusion.py This is the script that starts the training of the neural network. The training will spit out a checkpoint every so often depending on what value you specified for the constructor parameter `save_interval` in the script `RunCodeForDiffusion.py`.

GenerateNewImageSamples.py You will execute this script when you want to test a checkpoint for its ability to generate new images. See the doc section of the script for how to call it. This script deposits all the generated images in a numpy ndarray archive.

VisualizeSamples.py You must execute this script to extract the individual images from the numpy archive mentioned above.,

The Dataset Used for the Demos in the ExamplesDiffusion Directory

- Unless you change the pathname that the constructor option data_dir points to in the script RunCodeForDiffusion.py, the code in the ExamplesDiffusion directory is meant to be run with the same dataset that I used earlier in this lecture for data modeling with Adversarial Learning. As you may recall, the name of that dataset if PurdueShapes5GAN.

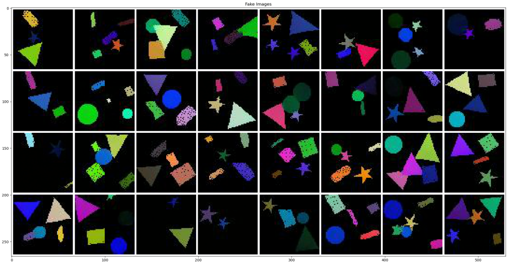
- See the Slides 53 through 59 for a description of that dataset. In particular, Slide 59 lists the steps for how to install the dataset.

- As implied by the first bullet above, you should be able to run the diffusion code on any image dataset just by changing the pathname setting of the data_dir constructor parameter in the script RunCodeForDiffusion.py.

Diffusion Modeling Results Obtained with the PurdueShapes5GAN Dataset

- This slide shows the results I obtain with the training based on the PurdueShapes5GAN dataset using the scripts in the ExamplesDiffusion directory of the DLStudio. Compare this result with that shown on Slide 80 for Adversarial Learning.

  [**IMPORTANT: This is NOT an apples-to-apples comparison between Adversarial Learning and Diffusion based modeling. With more epochs and additional hyperparameter tuning, the result shown on Slide 80 would be just as impressive as the one shown here.**]



These are the images generated by the p-chain at the checkpoint created at timestep 60,000 using the training batch-size of 32.

Diffusion Modeling Results Obtained with the CelebA Dataset

On this and the next few slides I will show some results obtained by Aditya Chauhan using the DLStudio code for diffusion modeling. These were again obtained using the scripts in the `ExamplesDiffusion` directory. Aditya is working on his Ph.D in RVL.

These are the images generated by the p-chain at the checkpoint created at timestep 10,000 using the training batch-size of 32.

Diffusion Modeling Results Obtained with CelebA (contd.)

These are the images generated by the p-chain at the checkpoint created at timestep 40,000 using the training batch-size of 32.

Diffusion Modeling Results Obtained with CelebA (contd.)

These are the images generated by the p-chain at the checkpoint created at timestep 80,000 using the training batch-size of 32.

Diffusion Modeling Results Obtained with CelebA (contd.)

These are the images generated by the p-chain at the checkpoint created at timestep 120,000 using the training batch-size of 32.