Generative Adversarial Networks for Data Modeling

Lecture Notes on Deep Learning

Avi Kak and Charles Bouman

Purdue University

Friday 22\textsuperscript{nd} January, 2021  13:23
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.
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


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.
The modern excitement in adversarial learning for data modeling began with the 2014 publication "Generative Adversarial Nets" by Goodfellow, Pouget-Abadie, Mirza, Xu, Warde-Farley, Ozair, Courville, and Bengio:


Such learning involves two networks, a Discriminator network and a Generator network:

- We can think of the Discriminator network as a function $D(x, \theta_d)$ whose output is the probability that a sample $x$ comes from the probability distribution that describes the training data. The notation $\theta_d$ represents the learnable parameters in the Discriminator network.

- Similarly, we can think of the Generator network as a function $G(z, \theta_g)$ that maps noise vectors $z$ to samples that we want to look like the samples in our training data. The vector $\theta_g$ represents the learnable parameters in the Generator network.
If $p_{\text{data}}$ represents the probability distribution that describes the training data and $p_g$ represents the probability distribution that the Generator network has learned so far, the goal of deep learning for probabilistic data modeling would be estimate the best values for the parameters $\theta_d$ and $\theta_g$ so that some measure of the distance between distributions $p_{\text{data}}$ and $p_g$ is minimized.

What’s interesting is that the deep learning framework that was actually implemented by Goodfellow et al. did not directly minimize a distance between $p_{\text{data}}$ and $p_g$. Nevertheless, the authors were able to argue that if the Discriminator was trained to an optimum level, it was guaranteed to yield a solution for $p_g$ that would be a minimum Jensen-Shannon divergence approximation to $p_{\text{data}}$.

The above paragraph points to the following fact: In order to understand algorithms for probabilistic data modeling, you must first understand how to measure the “distance” between two probability distributions.
Preamble (contd.)

For the reason stated at the bottom of the previous slide, 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 $\theta_d$ and $\theta_g$ 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 last couple of years, 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 WassersteinGAN that was presented by Arjovsky, Chintala, and Bottou in the following 2017 publication:

As I mentioned on the previous slide, I'll start this lecture with a review of the distance functions for probability distributions. That will get us ready to talk about my implementation of DCGAN and WassersteinGAN in version 2.0.3 of the DLStudio module:

https://engineering.purdue.edu/kak/distDLS/

If you are already familiar with the module and for whatever reason you just need to “pip install” the latest version of the code, here is a link to its PyPi repository:

https://pypi.org/project/DLStudio/

The DCGAN that I mentioned above was first presented by Radford, Metz, and Chintala in the following 2016 publication:


It was the first fully convolutional implementation of a GAN.
<table>
<thead>
<tr>
<th>Section Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Distance Between Two Probability Distributions</td>
<td>10</td>
</tr>
<tr>
<td>Total Variation (TV) Distance</td>
<td>12</td>
</tr>
<tr>
<td>Kullback-Leibler Divergence</td>
<td>16</td>
</tr>
<tr>
<td>Jensen-Shannon Divergence and Distance</td>
<td>22</td>
</tr>
<tr>
<td>Earth Mover’s Distance</td>
<td>26</td>
</tr>
<tr>
<td>Wasserstein Distance</td>
<td>36</td>
</tr>
<tr>
<td>A Random Experiment for Studying Differentiability</td>
<td>43</td>
</tr>
<tr>
<td>Differentiability of Distance Functions</td>
<td>45</td>
</tr>
<tr>
<td>PurdueShapes5GAN Dataset for Adversarial Learning</td>
<td>53</td>
</tr>
<tr>
<td>DCGAN Implementation in DLStudio</td>
<td>60</td>
</tr>
<tr>
<td>Making Small Changes to DCGAN Architecture</td>
<td>72</td>
</tr>
<tr>
<td>Wasserstein GAN Implementation in DLStudio</td>
<td>78</td>
</tr>
</tbody>
</table>
# Distance Between Two Probability Distributions

## Outline

1. **Distance Between Two Probability Distributions** 10
2. Total Variation (TV) Distance 12
3. Kullback-Leibler Divergence 16
4. Jensen-Shannon Divergence and Distance 22
5. Earth Mover’s Distance 26
6. Wasserstein Distance 36
7. A Random Experiment for Studying Differentiability 43
8. Differentiability of Distance Functions 45
9. PurdueShapes5GAN Dataset for Adversarial Learning 53
10. DCGAN Implementation in DLStudio 60
11. Making Small Changes to DCGAN Architecture 72
12. Wasserstein GAN Implementation in DLStudio 78
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 ML 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 pairs of probability distributions:

- **Total Variation Distance**
- **Kullback-Liebler Divergence**
- **Jensen-Shannon Divergence**
- **Earth Mover’s Distance**
- **Wasserstein Distance**
Total Variation (TV) Distance

Outline

1. Distance Between Two Probability Distributions 10
2. Total Variation (TV) Distance 12
3. Kullback-Leibler Divergence 16
4. Jensen-Shannon Divergence and Distance 22
5. Earth Mover’s Distance 26
6. Wasserstein Distance 36
7. A Random Experiment for Studying Differentiability 43
8. Differentiability of Distance Functions 45
9. PurdueShapes5GAN Dataset for Adversarial Learning 53
10. DCGAN Implementation in DLStudio 60
11. Making Small Changes to DCGAN Architecture 72
12. Wasserstein GAN Implementation in DLStudio 78

Purdue University
We start with a continuous random variable $\{X \mid x \in \mathbb{R}^n\}$ 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

$$d_{TV}(f, g) = \sup_{A} \left| \int_{A} f(x) dx - \int_{A} g(x) dx \right| : A \subset \mathbb{R}^n$$

What that says is that we check every subset $A$ of the domain $\mathbb{R}^n$ and find the total difference between the probability mass over that subset for both the $f$ and $g$ densities. The largest value for this difference is the TV distance between the two.

The important thing here is that TV 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.
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:

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

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

$$d_{TV}(P, Q) = \sup_{A} \left[ \left| \sum_{x_i \in A} P(x_i) - \sum_{x_i \in A} Q(x_i) \right| \right] \quad : \quad A \subset X \quad (3)$$
Let’s now consider the following two subsets of the set $X$:

$$
A_1 = \{ x_i \in X \mid P(x_i) \geq Q(x_i) \} \\
A_2 = \{ x_i \in X \mid Q(x_i) < P(x_i) \}
$$

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:

$$
d_{TV}(P, Q) = \frac{1}{2} \sum_{x_i \in X} |P(x_i) - Q(x_i)|
$$

$$
= \frac{1}{2} L_1(P, Q)
$$

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


Outline

1. Distance Between Two Probability Distributions 10
2. Total Variation (TV) Distance 12
3. Kullback-Leibler Divergence 16
4. Jensen-Shannon Divergence and Distance 22
5. Earth Mover’s Distance 26
6. Wasserstein Distance 36
7. A Random Experiment for Studying Differentiability 43
8. Differentiability of Distance Functions 45
9. PurdueShapes5GAN Dataset for Adversarial Learning 53
10. DCGAN Implementation in DLStudio 60
11. Making Small Changes to DCGAN Architecture 72
12. Wasserstein GAN Implementation in DLStudio 78
Kullback-Liebler 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

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

(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 $x \log x \rightarrow 0$ as $x \rightarrow 0^+$. 

- 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.
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:

\[
d_{KL}(P, Q) = \sum_{i=1}^{N} P(x_i) \log \frac{P(x_i)}{Q(x_i)}
\]

\[
= -\sum_{i=1}^{N} P(x_i) \log \frac{Q(x_i)}{P(x_i)}
\]

\[
= -\sum_{i=1}^{N} P(x_i) \log \frac{P(x_i) + Q(x_i) - P(x_i)}{P(x_i)}
\]

\[
= -\sum_{i=1}^{N} P(x_i) \log \left[ 1 + \frac{Q(x_i) - P(x_i)}{P(x_i)} \right]
\]

\[
= -\sum_{i=1}^{N} P(x_i) \log(1 + a) \quad (7)
\]
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:

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

$$= - \sum_{i=1}^{N} [Q(x_i) - P(x_i)]$$

$$= 0 \quad \text{(8)}$$

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 arguments.

- 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:

  $$d_{KL}(P, Q) = H_P(Q) - H(P)$$

  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 $P$. [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.]
Perhaps the most important role that KL-Divergence plays in the ongoing discussion related to Adversarial Networks is that it is a stepping stone to learning the Jensen-Shannon divergence (and the closely related Jensen-Shannon distance) that is presented starting with the next slide.

In Python, a call like:

```python
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. The two histogram arrays must be of equal length. You can also specify the base of the logarithm with an optional third argument. The default for the base is $e$ for the natural algorithm.
Jensen-Shannon Divergence and Distance

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

$$X = \{x_1, x_2, \ldots, x_N\}$$  \hspace{1cm} (10)

- 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 symmetrized version of the KL-Divergence presented earlier in Eq. (6):

$$d_{JS}(P, Q) = d_{KL}(P, M) + d_{KL}(Q, M)$$  \hspace{1cm} (11)

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

$$M = \frac{P + Q}{2}$$  \hspace{1cm} (12)

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

$$\text{dist}_{JS}(P, Q) = \sqrt{d_{JS}(P, Q)}$$  \hspace{1cm} (13)
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 value is 0, the two distributions $P$ and $Q$ are identical. And when the value is 1, the two distributions as as different as they can be.

Most significantly, $dist_{JS}(P, Q)$ is a valid metric distance.
Given two histogram arrays $P$ and $Q$ of equal length, normalized or unnormalized, a call like the following in Python

```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.
Earth Mover's Distance

Outline

1. Distance Between Two Probability Distributions 10
2. Total Variation (TV) Distance 12
3. Kullback-Leibler Divergence 16
4. Jensen-Shannon Divergence and Distance 22
5. **Earth Mover's Distance** 26
6. Wasserstein Distance 36
7. A Random Experiment for Studying Differentiability 43
8. Differentiability of Distance Functions 45
9. PurdueShapes5GAN Dataset for Adversarial Learning 53
10. DCGAN Implementation in DLStudio 60
11. Making Small Changes to DCGAN Architecture 72
12. Wasserstein GAN Implementation in DLStudio 78

Purdue University
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
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:

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

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. 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 al. paper:

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.
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 \(d_Lr\) distance in Eq. (14) will give you exactly the opposite answer.

Since distances like \(d_Lr\) in Eq. (14) 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:

\[ c_{ij} = 1 - e^{-\alpha|i-j|} \]  

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 $M$ provides with different quantities ($\{g_i | i = 1, \ldots, M\}$) of some 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 $F$:

\[
\min_F \sum_{i=1}^{M} \sum_{j=1}^{N} c_{ij} f_{ij} \tag{16}
\]

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:

  \[
  f_{ij} \geq 0 \quad i = 1, \ldots, M, \quad j = 1, \ldots, N
  \]  
  \( (17) \)

  \[
  \sum_{j=1}^{N} f_{ij} \leq h_i \quad i = 1, \ldots, M
  \]  
  \( (18) \)

  \[
  \sum_{i=1}^{M} f_{ij} \leq g_j \quad j = 1, \ldots, N
  \]  
  \( (19) \)

  \[
  \sum_{i=1}^{M} \sum_{j=1}^{N} f_{ij} = \min \left\{ \sum_{i=1}^{M} g_i, \sum_{j=1}^{N} h_j \right\}
  \]  
  \( (20) \)

- All four constraints are straightforward because they are so intuitive.

  [The constraints in Eqs. (17) and (18) 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. (19) also makes sense since the accumulated in-flows for the \( j^{th} \) consumer should not exceed total demand for that consumer. The constraint in Eq. (20) 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:

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

(21)

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) types over the last several decades and the polynomial-time solutions provided 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.
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.
# Outline

1. **Distance Between Two Probability Distributions**
   - Page 10

2. **Total Variation (TV) Distance**
   - Page 12

3. **Kullback-Leibler Divergence**
   - Page 16

4. **Jensen-Shannon Divergence and Distance**
   - Page 22

5. **Earth Mover’s Distance**
   - Page 26

6. **Wasserstein Distance**
   - Page 36

7. **A Random Experiment for Studying Differentiability**
   - Page 43

8. **Differentiability of Distance Functions**
   - Page 45

9. **PurdueShapes5GAN Dataset for Adversarial Learning**
   - Page 53

10. **DCGAN Implementation in DLStudio**
    - Page 60

11. **Making Small Changes to DCGAN Architecture**
    - Page 72

12. **Wasserstein GAN Implementation in DLStudio**
    - Page 78
**Wasserstein Distance**

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

$$d_W(P, Q) = \inf_{\gamma(X, Y) \in \Gamma(P, Q)} E(X, Y) \sim \gamma \left[ \|x - y\| \right]$$

- 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. (22) above.
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. (22) 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. (22), 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 $\gamma(X, Y)$. 
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. (22) can be converted into a computationally tractable supremum calculated separately over the component distributions $P$ and $Q$ as shown below

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

(23)

for ALL 1-Lipschitz functions $f : X \to R$ 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. (23) is from a famous book in Optimal Transport Theory by Cédric Villani:


- Despite the use of ”ALL” for the family of 1-Lipschitz functions $f()$ in Eq. (23), 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. (23) 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 $|f(x_1) - f(x_2)| \leq k \cdot d(x_1, x_2)$ 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 $x_2$. 
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 $f$:

$$f(x) = \inf_{y \in X} [f(y) + k \cdot d(x, y)] = \sup_{y \in X} [f(y) - k \cdot d(x, y)]$$  \hspace{1cm} (24)$$

Note that the definition $|f(x) - f(y)| \leq k \cdot d(x, y)$ implies $f(y) - k \cdot d(x, y) \leq f(x) \leq f(y) + k \cdot d(x, y)$ When you apply the definitions of infimum and supremum to these inequalities, you get the form shown in Eq. (24).
We are faced with the following questions if we want to use the form in Eq. (23) for computing the Wasserstein Loss in adversarial learning:

- How do we find the function $f()$ that would solve the maximization problem in Eq. (23)?

- The expectation operator $E()$ in Eq. (23) 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 the section on how to use the Wasserstein distance for adversarial learning.
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Distance Between Two Probability Distributions</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>Total Variation (TV) Distance</td>
<td>12</td>
</tr>
<tr>
<td>3</td>
<td>Kullback-Leibler Divergence</td>
<td>16</td>
</tr>
<tr>
<td>4</td>
<td>Jensen-Shannon Divergence and Distance</td>
<td>22</td>
</tr>
<tr>
<td>5</td>
<td>Earth Mover’s Distance</td>
<td>26</td>
</tr>
<tr>
<td>6</td>
<td>Wasserstein Distance</td>
<td>36</td>
</tr>
<tr>
<td>7</td>
<td>A Random Experiment for Studying Differentiability</td>
<td>43</td>
</tr>
<tr>
<td>8</td>
<td>Differentiability of Distance Functions</td>
<td>45</td>
</tr>
<tr>
<td>9</td>
<td>PurdueShapes5GAN Dataset for Adversarial Learning</td>
<td>53</td>
</tr>
<tr>
<td>10</td>
<td>DCGAN Implementation in DLStudio</td>
<td>60</td>
</tr>
<tr>
<td>11</td>
<td>Making Small Changes to DCGAN Architecture</td>
<td>72</td>
</tr>
<tr>
<td>12</td>
<td>Wasserstein GAN Implementation in DLStudio</td>
<td>78</td>
</tr>
</tbody>
</table>
The discussion in this section is an elaboration of the “learning parallel lines” example in the paper


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

The “true” data generated by $Z$ falls on a line in $R^2$ — this would presumably be our “training” data (to make an analogy with GAN training). Now imagine a Generator that is also capable of producing points in $R^2$, but its output is a function of a single parameter, $\theta$, which is the value of the offset from the true line.

We use $X$ as the random variable to denote the points on the true line in $R^2$ and $Y$ to denote the points being produced by the Generator using its parameter value for $\theta$ from the output of $Z$. 
Outline

1. Distance Between Two Probability Distributions 10
2. Total Variation (TV) Distance 12
3. Kullback-Leibler Divergence 16
4. Jensen-Shannon Divergence and Distance 22
5. Earth Mover’s Distance 26
6. Wasserstein Distance 36
7. A Random Experiment for Studying Differentiability 43
8. Differentiability of Distance Functions 45
9. PurdueShapes5GAN Dataset for Adversarial Learning 53
10. DCGAN Implementation in DLStudio 60
11. Making Small Changes to DCGAN Architecture 72
12. Wasserstein GAN Implementation in DLStudio 78
We start with the differentiability of the Wasserstein Distance.

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

Therefore, the value of the Expectation operator in Eq. (22) will also be equal to $\theta$. In other words, for the random experiment under consideration:

\[ d_W(P, Q) = \theta \] (25)

So we see that the Wasserstein distance is continuous and differentiable with respect to the learnable parameter $\theta$. 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:

$$d_{KL}(P, Q) = \int P(x) \log \frac{P(x)}{Q(x)} \, dx$$  \hspace{1cm} (26)

- 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.
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 $\theta$ 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:

$$d_{KL}(P, Q) = \begin{cases} 0 & \theta = 0 \\ +\infty & \theta \neq 0 \end{cases}$$

(27)

Obviously, KL-Divergence is not differentiable with respect to the learnable parameter $\theta$.

Next we take up the case of differentiability of JS-Divergence.
The formula for JS-Divergence was presented in Eq. (11). Given two distributions \( P \) and \( Q \), using the formula in that equation requires that we first calculate the mean distribution \( M \) as defined in Eq. (12).

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. (12) 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 \( M \).
Focusing on the case when the learnable parameter $\theta$ 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. (11):

$$d_{KL}(P, M) = \int P(x) \log \frac{P(x)}{M(x)} \, dx$$

$$= \int P(x) \left[ \log P(x) - \log \frac{P(x) + Q(x)}{2} \right] \, dx$$

$$= \int P(x) \left[ \log P(x) - \log(P(x) + Q(x)) + \log 2 \right] \, dx$$

$$= \int P(x) \log 2 \, dx$$

$$= \log 2$$  \hspace{1cm} (28)

As expected, the expressions on the RHS of Eq. (11) are now innoculated against going to infinity under the condition $Q(x) = 0$ when $P(x) > 0$. 

Purdue University 50
Since both the component expressions on the RHS of Eq. (11) 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:

\[
\begin{align*}
    d_{JS}(P, Q) &= 0 \quad \text{if } \theta = 0 \\
    &= \log 2 \quad \text{if } \theta \neq 0
\end{align*}
\]

(29)

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 \( A \).
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) for such an $A$ is the largest it can be — equal to 1.

On the other hand, when the Generator’s parameter $\theta$ 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:

$$d_{TV}(P, Q) = \begin{cases} 0 & \theta = 0 \\ 1 & \theta \neq 0 \end{cases}$$

(30)

TV is obviously not a differentiable distance function.
Outline

1. Distance Between Two Probability Distributions 10
2. Total Variation (TV) Distance 12
3. Kullback-Leibler Divergence 16
4. Jensen-Shannon Divergence and Distance 22
5. Earth Mover’s Distance 26
6. Wasserstein Distance 36
7. A Random Experiment for Studying Differentiability 43
8. Differentiability of Distance Functions 45
9. **PurdueShapes5GAN Dataset for Adversarial Learning** 53
10. DCGAN Implementation in DLStudio 60
11. Making Small Changes to DCGAN Architecture 72
12. Wasserstein GAN Implementation in DLStudio 78

Purdue University
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 $64 \times 64$.

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 (assuming it comes 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.
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.)

Figure: 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.)
You can download the dataset archive

    datasets_for_AdversarialNetworks.tar.gz

through the link ”Download the image dataset for AdversarialNetworks” 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 ExamplesAdversarialNetworks 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
Outline

1. Distance Between Two Probability Distributions 10
2. Total Variation (TV) Distance 12
3. Kullback-Leibler Divergence 16
4. Jensen-Shannon Divergence and Distance 22
5. Earth Mover’s Distance 26
6. Wasserstein Distance 36
7. A Random Experiment for Studying Differentiability 43
8. Differentiability of Distance Functions 45
9. PurdueShapes5GAN Dataset for Adversarial Learning 53
10. DCGAN Implementation in DLStudio 60
11. Making Small Changes to DCGAN Architecture 72
12. Wasserstein GAN Implementation in DLStudio 78

Purdue University
DCGAN Implementation in DLStudio

The main goal of this section is to tell you about the implementation of DCGAN in Version 2.0.3 (or higher) of the DLStudio module.

DCGAN, short for "Deep Convolutional Generative Adversarial Network", was presented in a paper that I cited in the Preamble section.

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 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 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 $\theta_d$ 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 $\theta_g$ 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.
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 corresponds 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.
That is, for Discriminator training, we want the parameters $\theta_d$ to maximize the following expectation:

$$\max_{\theta_d} E_{x \sim p_{\text{data}}} [\log D(x)]$$  \hspace{1cm} (31)

The expression $x \sim p_{\text{data}}$ means that $x$ was pulled from the distribution $p_{\text{data}}$. 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:

$$\min_{\theta_g} E_{z \sim p_Z} [\log (1 - D(G(z)))]$$  \hspace{1cm} (32)

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

$$\min_{\theta_g} \max_{\theta_d} \left[ E_{x \sim p_{\text{data}}} [\log D(x)] + E_{z \sim p_Z} [\log (1 - D(G(z)))] \right]$$  \hspace{1cm} (33)
Let’s translate this min-max form in Eq. (33) 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 in the Generator network. If we use `nn.BCELoss` as the loss criterion, that will automatically take care of the logarithms in the expression shown above. First the Discriminator training through maximization:

- The maximization of the first term simply requires that we use the target ”1” for the network output $D(x)$.

- The maximization of the second term above 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.

- 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.
For the training required for the Generator, only the second term inside the square brackets in Eq. (33) matters. We proceed as follows:

- 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.

- Therefore, the needed minimization translates into maximizing $D(G(z))$ with respect to a target value of 1.

- With 1 as the target, we again find the `nn.BCELoss` associated with $D(G(z))$. We call `backwards()` on this loss — making sure that we have turned off `requires_grad()` on the Discriminator parameters as we are updating the Generator parameters.

- A subsequent call to the `step()` for the optimizer would update the weights in the Generator network.
The explanation presented above is how the training is carried out for the DCGAN implementations DG1 and DG2 in the `AdversarialNetworks` class of the DLStudio module.

However, before I show the actual training loop used, I must introduce to the Discriminator and Generator networks in the DCGAN section of the code. I have shown this pair of networks starting on the next slide.

This is an implementation of the DCGAN Discriminator. 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 4x4 kernel, a 2x2 stride and a 1x1 padding for all but the final layer. The output of the final convolutional layer is pushed through a sigmoid to yield a scalar value as the final output for each image in a batch.

This is an implementation of the DCGAN Generator. As was the case with the Discriminator network, you again see the 4-2-1 topology here. 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. (We refer to the images constructed from noise vectors in this manner as fakes.) As you will later in the `gan` code()}
DCGAN Implementation in DLStudio

### Discriminator-Generator DG1

**class DiscriminatorDG1(nn.Module):**

```python
""
This is an implementation of the DCGAN Discriminator. 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 4x4 kernel, a 2x2 stride and a 1x1 padding for all but the final layer. The output of the final convolutional layer is pushed through a sigmoid to yield a scalar value as the final output for each image in a batch.
""
def __init__(self):
    super(AdversarialNetworks.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
```

**class GeneratorDG1(nn.Module):**

```python
""
This is an implementation of the DCGAN Generator. As was the case with the Discriminator network, you again see the 4-2-1 topology here. 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. (We refer to the images constructed from noise vectors in this manner as fakes.) As you will later in the "run_gan_code()", the starting noise vector is a 1x1 image with 100 channels. In order to output 64x64 output images, the network shown below use 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 size pairs are related by

\[
H_{out} = (H_{in} - 1) * s + k - 2 * p
\]
\[
W_{out} = (W_{in} - 1) * s + k - 2 * p
\]

were s is the stride and k the size of the kernel. (I am assuming square strides, kernels, and padding). Therefore, each nn.ConvTranspose2d layer shown below doubles the size of the input.
""
def __init__(self):
    super(AdversarialNetworks.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(128)
    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
```
Losses vs. Iterations for DG1

Figure: Discriminator and Generator losses over 30 epochs of training
Comparing Real and Fake Images for DG1

Figure: 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
Making Small Changes to DCGAN Architecture

Outline

1. Distance Between Two Probability Distributions 10
2. Total Variation (TV) Distance 12
3. Kullback-Leibler Divergence 16
4. Jensen-Shannon Divergence and Distance 22
5. Earth Mover’s Distance 26
6. Wasserstein Distance 36
7. A Random Experiment for Studying Differentiability 43
8. Differentiability of Distance Functions 45
9. PurdueShapes5GAN Dataset for Adversarial Learning 53
10. DCGAN Implementation in DLStudio 60
11. Making Small Changes to DCGAN Architecture 72
12. Wasserstein GAN Implementation in DLStudio 78
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`. 
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(AdversarialNetworks.DataModeling.DiscriminatorDG2, 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)

    if skip_connections:
        self.bn1 = nn.BatchNorm2d(128)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(512)
        self.bn4 = nn.BatchNorm2d(64)
        self.bn5 = nn.BatchNorm2d(1)
    else:
        self.bn1 = nn.BatchNorm2d(128)
        self.bn2 = nn.BatchNorm2d(256)
        self.bn3 = nn.BatchNorm2d(512)
        self.bn4 = 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
Making Small Changes to DCGAN Architecture

**Losses vs. Iterations for DG2**

**Figure:** Discriminator and Generator losses over 30 epochs of training
Comparing Real and Fake Images for DG2

Figure: 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.

## Outline

1. Distance Between Two Probability Distributions 10
2. Total Variation (TV) Distance 12
3. Kullback-Leibler Divergence 16
4. Jensen-Shannon Divergence and Distance 22
5. Earth Mover’s Distance 26
6. Wasserstein Distance 36
7. A Random Experiment for Studying Differentiability 43
8. Differentiability of Distance Functions 45
9. PurdueShapes5GAN Dataset for Adversarial Learning 53
10. DCGAN Implementation in DLStudio 60
11. Making Small Changes to DCGAN Architecture 72
12. Wasserstein GAN Implementation in DLStudio 78
This implementation is based on the paper ”Wasserstein GAN” by Arjovsky, Chintala, and Bottou that I cited previously in the Preamble.

You will find an implementation of Wasserstein GAN (WGAN) in DLStudio, Version 2.0.3 or higher, in the enclosing class AdversarialNetworks.

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. (23).

The 1-Lipschitz function $f()$ that is required by the definition in Eq. (23) 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.
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. (23):

\[
\text{d}_{W}(P_{r}, P_{\theta}) = \sup_{\|f\|_{L} \leq 1} \left[E_{x \sim P_{r}} \{f_{w}(x)\} - E_{z \sim P_{z}} \{f_{w}(g_{\theta}(z))\}\right]
\]  

(34)
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 $\theta$ 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.
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. Experimentally, they demonstrated that this heuristic actually worked on real data.

The calculation of the Wasserstein distance using Eq. (34) also calls for significant averaging of the output of the Critic in order the maximization to yield the desired distance. This can be taken care of my having the Critic go through multiple iterations of the update of its parameters for each iteration for the Generator.
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 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.

```python
class CriticCG1(nn.Module):
    def __init__(self):
        super(AdversarialNetworks.DataModeling.CriticCG1, self).__init__()
        self.conv_in = AdversarialNetworks.DataModeling.SkipBlockDN(3, 64, downsample=True, skip_connections=True)
        self.conv_in2 = AdversarialNetworks.DataModeling.SkipBlockDN(64, 128, downsample=True, skip_connections=False)
        self.conv_in3 = AdversarialNetworks.DataModeling.SkipBlockDN(128, 256, downsample=True, skip_connections=False)
        self.conv_in4 = AdversarialNetworks.DataModeling.SkipBlockDN(256, 512, downsample=True, skip_connections=False)
        self.conv_in5 = AdversarialNetworks.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):
    def __init__(self):
        super(AdversarialNetworks.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
```

The Generator code remains the same as for the DCGAN shown earlier.
Figure: Critic and Generator losses over 500 epochs of training
Comparing Real and Fake Images for WGAN

Figure: 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.