Deep Probabilistic Models

Part II: Generative Adversarial Networks and Stochastic Backpropagation

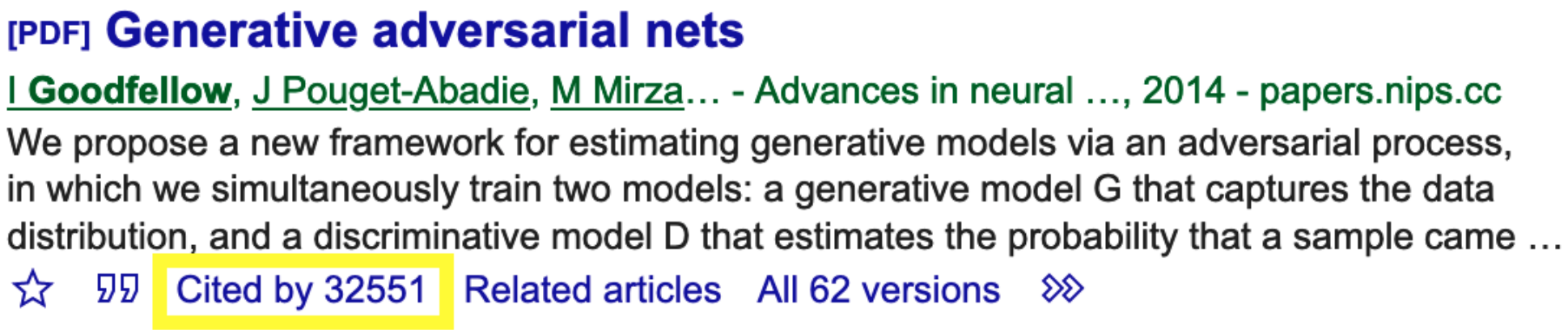
Robert Salomone

AMSI Winter School, 2021

Part II: Roadmap

- **Generative Adversarial Networks**
  - Training when $T(\boldsymbol{Z})$ need not have a tractable likelihood (and using a loss with special properties)
- **Stochastic Backpropagation**
  - Estimating the gradient of expectations.
- A Trip to the **GAN Zoo**.
- **Discussion on GAN vs. Flows**
- **GAN Training** and **Improving GAN Performance** with Alternative Loss Functions
  - Wasserstein GAN
  - Least Squares GAN
- **GAN Extensions**
  - Conditional GAN: $p(\boldsymbol{x} \mid \boldsymbol{y})$
  - Adding Discrete Variables
    - The Gumbel-Softmax / Concrete Distribution

Everybody loves GAN...



- Why is this?
  - **State-of-the-Art**: GANs tend to make the best fake pictures (we will discuss why).
  - **Ease of Use**: Conceptually they are much simpler than directed or undirected graphical models which came before.
    - Also, you don't need to learn about MCMC or Variational Inference to use them like other methods
  - There are many follow up papers along different themes:
    - **We Fixed It!** : GANs have some problems and everyone wants to solve them with new approaches.
    - **We Proved Stuff** : GANs are conceptually interesting, and can lead to interesting theory. .
    - **We Can Use It**: There are a number of possible applications, extensions to particular data types and purposes.

Adversarial Networks and Loss Functions

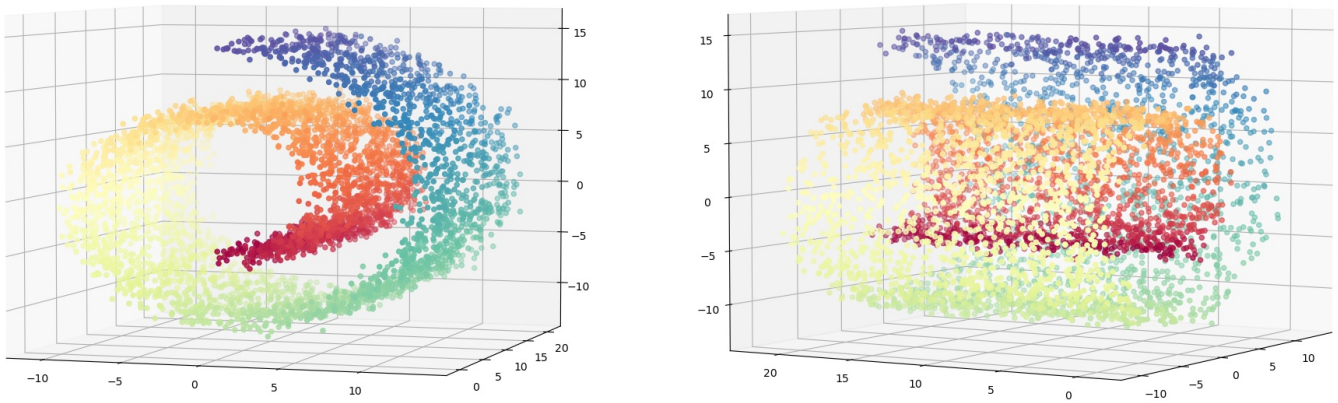
- For some target space $\mathcal{X}$, consider a **neural network** $g_{\boldsymbol{\theta}_g} : \mathbb{R}^m \to \mathcal{X}$ and let $\mathbf{Z} \sim \mathcal{N}(0, \mathrm{I}_m)$.
- The **generative form** of the model for our data is $\mathbf{X} = g(\mathbf{Z}; \boldsymbol{\theta}_g)$, where and $g : \mathbb{R}^m \to \mathcal{X}$ is some neural network parametrized by $\boldsymbol{\theta}_g$, which we will call our **generator**.
- Forgetting that the probability density (likelihood) may not be tractable for the moment - it is interesting to **note that $\mathcal{Z}$ need not be the same dimension as $\mathcal{X}$**.
- For non-trivial transforms $g_{\boldsymbol{\theta}_g}$, $\mathbf{X}$ is still a **random** object.

Manifolds

- The model implicitly assumes that **whatever you are modelling is a bunch of normally distributed independent**, **latent** factors that have been pushed through a neural net.
- Thus, if we could **somehow** train these models, we would also obtain a sort-of **manifold learning** algorithm in reverse.
    - We would have that $g_{\boldsymbol{\theta}_g}$ maps from **latent space** that is trained to be close to $\mathcal{N}(0, \mathrm{I}_m)$ for the data (and if it generalizes well, to the the population) to the **manifold** that the data lives on.
    - In a sense, it would implicitly identify some disentangled aspects of the data.
    - You can't invert $g_{\boldsymbol{\theta}}$ in this case to obtain the latent representation (which maybe you really want to do), but we will learn about machinery to do that later in the course.

Manifold Hypothesis

- The **manifold hypothesis** posits that high-dimensional data lives around a **low-dimensional manifold**.



- The above data could, **in principle**, be fit extremely well (but not perfectly!) by a model $g_{\boldsymbol{\theta}_g} : \mathbb{R}^2 \to \mathbb{R}^3$.
- Actually, you will turn in principle to **in reality** in Tutorial 1. However, for now, **we need to deal** with the fact that we don't have a likelihood!

Returning to Training Issue

- **Issue**: We don't know the likelihood for our model, so we can't train it via maximum likelihood. We need some valid **alternative loss** function.
- Solution: **Adversarial Training**
  - Goodfellow et al., (2014). *Generative adversarial nets*. NIPS' 2014.
  - However, the basic idea of adversarial training does predate the above paper.

Auxilliary Network

- **Auxilliary**: make things more complicated and adds new stuff, but in a way that helps you achieve your goal / solve your problems.
- We will introduce an **auxilliary** neural network $d(\,\cdot\,;\boldsymbol{\theta}_d)$ called the **discriminator**. Here, the parameter $\boldsymbol{\theta}_d$ comes from a set $\boldsymbol{\Theta}_d$ of potential parameters.
    - We don't really care about it on its own so much as we care about **using it** for our purposes.
- The key to **adversarial learning** is to try to get the optimal $\boldsymbol{\theta}_g^\star$ that solves, for some **value function** V,

$$\boldsymbol{\theta}_g^\star = \arg\min_{\theta_{\mathbf{g}} \in \boldsymbol{\Theta}_{\mathbf{g}}} \left\{ \max_{\theta_{\mathbf{d}} \in \boldsymbol{\Theta}_{\mathbf{d}}} V(\theta_{\mathbf{g}}, \theta_{\mathbf{d}}) \right\}$$

- In words: **we want the g_{θ_g} that minimizes the best possible performance of d_{θ_d}**
- This is a saddle point optimization problem.
- The introduction of d creates like an "artificial" likelihood for us to optimize.

Making it concrete...

$$\boldsymbol{\theta}_g^\star = \arg\min_{\boldsymbol{\theta}_g \in \boldsymbol{\Theta}_g} \left\{ \max_{\theta_{\mathbf{d}} \in \boldsymbol{\Theta}_{\mathbf{d}}} V(\theta_{\mathbf{g}}, \theta_{\mathbf{d}}) \right\}$$

- Ok, so we know what g does, but how do we make d do something useful?
- **Answer:** We can make d a **classifier** of whether a sample is "real" and from the data-generating distribution, or "fake" (i.e., with distribution given by $g_{\boldsymbol{\theta}}(\boldsymbol{Z})$).
    - It will output a value in $(0, 1)$.

Derivation of Loss Function

- Consider a $\text{Bernoulli}(p)$ random variable. The log-likelihood is $\log(p)$ if $y = 1$ and $\log(1-p)$ if $y = 0$. Recalling that $d(\cdot\,; \boldsymbol{\theta}_d)$ outputs a value in $(0, 1)$, we can define

$$V(\boldsymbol{\theta}_g, \boldsymbol{\theta}_d) = \underbrace{\frac{1}{n}\sum_{k=1}^{n} \log d(\mathbf{x_k};\, \boldsymbol{\theta}_d)}_{\text{avg. log-likelihood for real samples } (y=1)} + \underbrace{\mathbb{E}_g \log(1 - d(\boldsymbol{X}; \boldsymbol{\theta}_d))}_{\text{avg. log-likelihood for fake samples } (y=0)}$$

- Recall that the discriminator wants to **maximize** the above function, while the generator wants to **minimize** it.
  - If we trained L via **gradient descent**, this has the interpretation of both the generator and discriminator learning by "**playing a game against each other**".
- Note also that the first term is independent of the generator, so we may view it as minimizing two separate loss functions (note the negative sign with L_d as the discriminator wants to **maximize**):

$$L_g(\boldsymbol{\theta}_g) = \mathbb{E}_g \log(1 - d(\boldsymbol{X}; \boldsymbol{\theta}_d))$$

$$L_d(\boldsymbol{\theta}_d) = -\left(\frac{1}{n}\sum_{k=1}^{n} \log d(\mathbf{x_k};\, \boldsymbol{\theta}_d) + \mathbb{E}_g \log(1 - d(\boldsymbol{X}; \boldsymbol{\theta}_d))\right)$$

Expectations in our loss functions (!?)

- Note that we want to minimize both

$$L_g(\boldsymbol{\theta}_g) = \mathbb{E}_g \log(1 - d(\boldsymbol{X}; \boldsymbol{\theta}_d))$$

$$L_d(\boldsymbol{\theta}_d) = -\left(\frac{1}{n} \sum_{k=1}^{n} \log d(\mathbf{x_k}; \boldsymbol{\theta}_d) + \mathbb{E}_g \log(1 - d(\boldsymbol{X}; \boldsymbol{\theta}_d)) \right)$$

- Ordinarily with **neural networks**, we use **backpropagation** to obtain the gradients, but we have the additional issue of the expectation to deal with.
- Now, we need to take (unbiased) expectations with respect to derivatives. This is called **stochastic backpropagation** (or **stochastic optimization** depending on who you ask).

Stochastic Backpropagation

- The first technique is known by a number of names: e.g., **Infinitesimal Perturbation Analysis**, **Pathwise Differentiation**, **Reparametrization Gradient**, **Reparametrization Trick**.
- Suppose that $\boldsymbol{X}$ can be represented as $\boldsymbol{X} = T(\boldsymbol{Z}; \boldsymbol{\theta})$ where $\boldsymbol{Z}$ crucially **does not depend on $\boldsymbol{\theta}$**. Then, for some function of interest $\mathcal{L}$,

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{X}}(\,\cdot\,;\boldsymbol{\theta})}[\mathcal{L}(\boldsymbol{X})] = \nabla_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{Z}}} \mathcal{L}(T(\boldsymbol{Z}; \boldsymbol{\theta}))$$

- From the right hand side above, we proceed, subject to mild regularity conditions, as

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{Z}}} \mathcal{L}(T(\boldsymbol{z}; \boldsymbol{\theta})) = \nabla_{\boldsymbol{\theta}} \int p_{\boldsymbol{Z}}(\boldsymbol{z}) \mathcal{L}(T(\boldsymbol{z}; \boldsymbol{\theta})) \mathrm{d}\boldsymbol{x} = \int p_{Z}(\boldsymbol{z}) \nabla_{\boldsymbol{\theta}} \mathcal{L}\big(T(\boldsymbol{z}; \boldsymbol{\theta})\big) \mathrm{d}\boldsymbol{x} = \mathbb{E}_{p_{\boldsymbol{Z}}} \nabla_{\boldsymbol{\theta}} \mathcal{L}\big(T(\boldsymbol{z}; \boldsymbol{\theta})\big).$$

- And so, we can unbiasedly estimate the gradient of $\mathcal{L}$ if we just take the mean of the gradient of the loss for a bunch of samples (**very easy!**).
- The fact that such estimators typically have **very low variance** has started a revolution in ML since its "discovery" in 2014. However, what no one seems to notice is that the **stochastic simulation community** has been aware this since 1990! See the same paper linked above, but some measure theory is required!

Stochastic Backpropagation in the Non-Reparametrizable Case

- Suppose that you can not reparametrize. This is usually the case with **discrete** variables.
- This is also known by many names: e.g., **Score Function Method** and **REINFORCE**.

$$\nabla_{\boldsymbol{\theta}} \mathbb{E}_{p_{\boldsymbol{X}}(\cdot\,;\boldsymbol{\theta})} \mathcal{L}(\boldsymbol{X}) = \nabla_{\boldsymbol{\theta}} \int \mathcal{L}(\boldsymbol{x}) p_{\boldsymbol{X}}(\boldsymbol{x};\boldsymbol{\theta}) \mathrm{d}\boldsymbol{x} = \int \mathcal{L}(\boldsymbol{x}) \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{X}}(\boldsymbol{x};\boldsymbol{\theta}) \mathrm{d}\boldsymbol{x}$$
$$= \int \mathcal{L}(\boldsymbol{x}) \frac{p_{\boldsymbol{X}}(\boldsymbol{x};\boldsymbol{\theta})}{p_{\boldsymbol{X}}(\boldsymbol{x};\boldsymbol{\theta})} \nabla_{\boldsymbol{\theta}} p_{\boldsymbol{X}}(\boldsymbol{x};\boldsymbol{\theta}) \mathrm{d}\boldsymbol{x}.$$

- Using that $\nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{X}}(\cdot\,;\boldsymbol{\theta}) = \frac{\nabla_{\boldsymbol{\theta}} p_{\boldsymbol{X}}(\cdot\,;\boldsymbol{\theta})}{p_{\boldsymbol{X}}(\cdot\,;\boldsymbol{\theta})}$, the above is

$$= \int \mathcal{L}(\boldsymbol{x}) p_{\boldsymbol{X}}(\boldsymbol{x};\boldsymbol{\theta}) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{X}}(\boldsymbol{x};\boldsymbol{\theta}) \mathrm{d}\boldsymbol{x} = \mathbb{E}_{p_{\boldsymbol{X}}(\cdot\,;\boldsymbol{\theta})} \left[\mathcal{L}(\boldsymbol{X}) \nabla_{\boldsymbol{\theta}} \log p_{\boldsymbol{X}}(\boldsymbol{X}\,;\boldsymbol{\theta}) \right].$$

- **Key Takeaway:** In the non-reparametrizable case, you need to **tag on the score function** of the non-reparametrizable variables to your loss instead of taking its gradient.
- Score function estimator variance is typically pretty high! It is often unusable without sophisticated variance reduction techniques and many samples.

`torch.distributions`

- `Pytorch` has a built-in `distributions` module that **supports** stochastic backpropagation with automatic reparametrization.
- There are lots of distributions there, and it is extended (along with ability to use stochastic backpropagation) by the `pyro.distributions` module which you saw in the section on flow-based model.
- Using the `rsample` method on a distribution object instructs `PyTorch` to use reparametrization gradients (lower variance!)

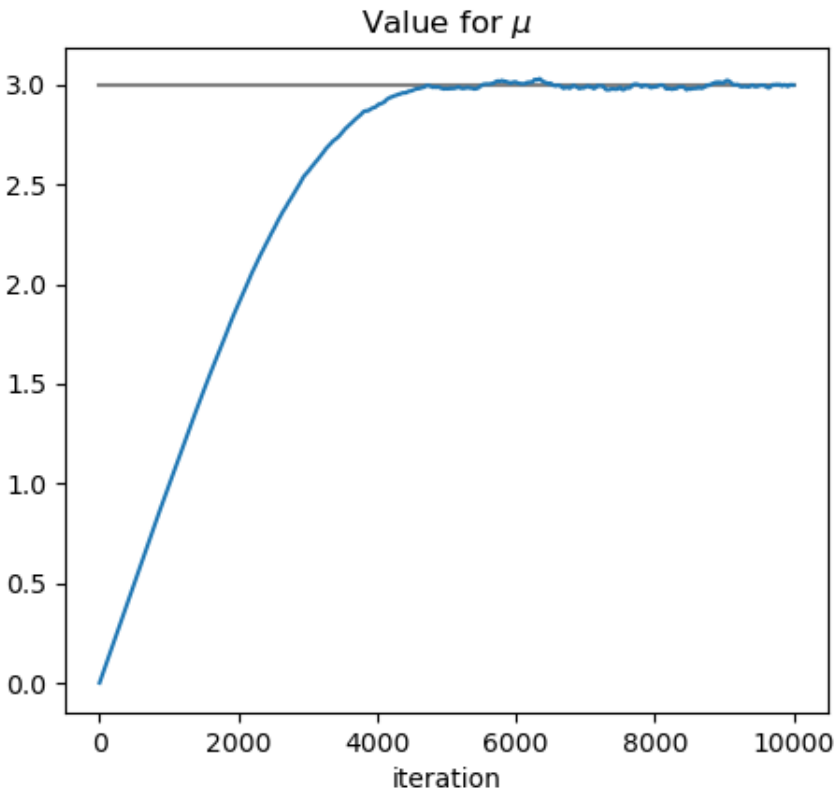
Simple Example

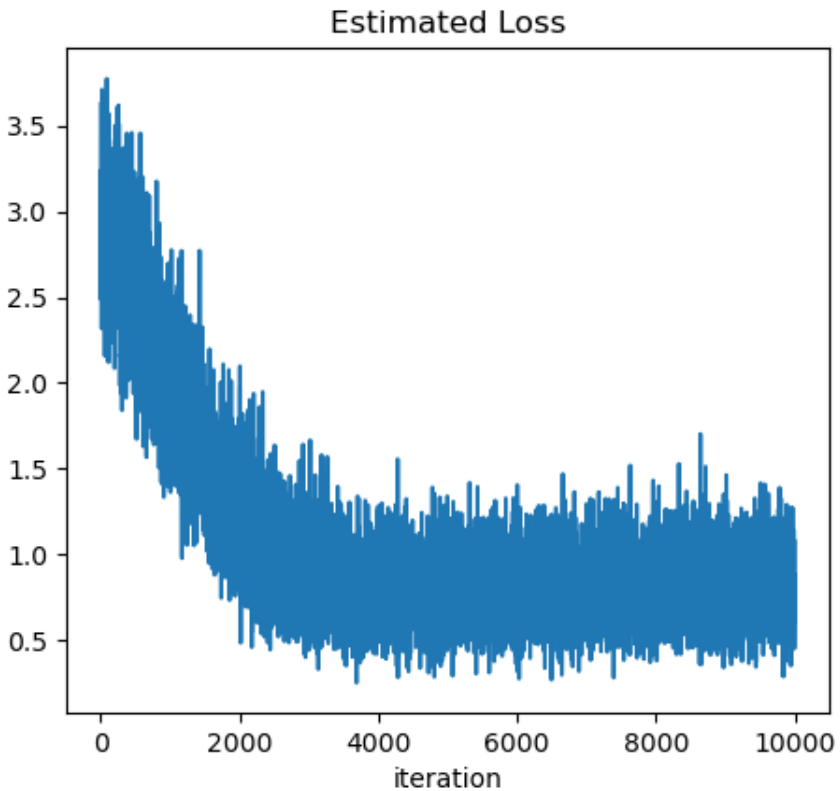
- The code below tries to solve the problem of finding

$$\mu^\star = \arg\min_{\mu \in \mathbb{R}} \{\mathbb{E}|X-3|\}, \quad X \sim \mathcal{N}(\mu, 1)$$

```
dist = torch.distributions.Normal(loc=0., scale=1.)  # create N(mu,1) RV
dist.loc.requires_grad_(True) # tells PyTorch we will want the gradient of dist.loc

adam = torch.optim.Adam([dist.loc]) # create an Adam optimizer object

for i in range(10000):
    X = dist.rsample() # sample in a manner that yields reparametrization gradients
    loss = torch.mean(torch.abs((X - 3))) # estimate the expected loss E[|X-3|]
    loss.backward() # stochastic backpropagation
    adam.step() # perform gradient descent
    adam.zero_grad() # zero out all the gradients
```

Example


Value for μ
3.0
2.5
2.0
1.5
1.0
0.5
0.0
0
2000
4000
6000
8000
10000
iteration


Estimated Loss
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0
2000
4000
6000
8000
10000
iteration

Returning to GANs

- Fortunately for us, GANs are by construction already in reparametrized form as $\boldsymbol{X} = g(\boldsymbol{Z}\,;\,\boldsymbol{\theta}_g)$! So, the gradient operator passes under the expectations and we obtain

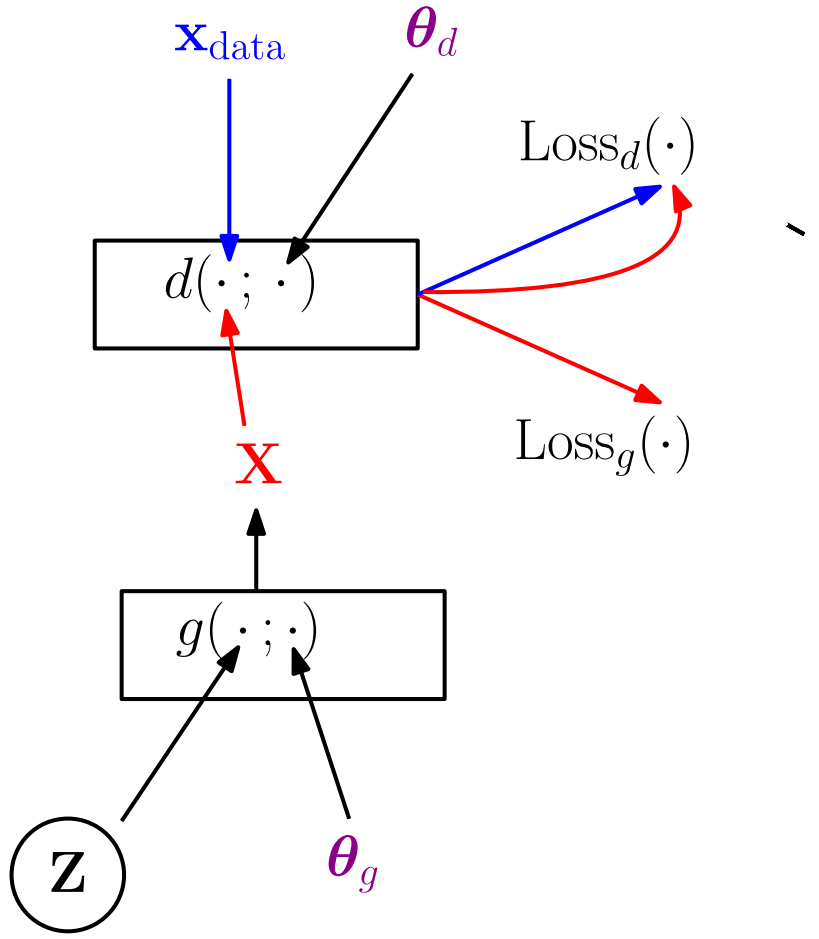
$$\nabla_{\boldsymbol{\theta}_g} L_g(\boldsymbol{\theta}_g) = \mathbb{E}_g \nabla_{\boldsymbol{\theta}_g} \log(1 - d(g(\boldsymbol{Z}; \boldsymbol{\theta}_g); \boldsymbol{\theta}_d))$$

$$\nabla_{\boldsymbol{\theta}_d} L_d(\boldsymbol{\theta}_g) = -\left(\frac{1}{n} \sum_{k=1}^{n} \nabla_{\boldsymbol{\theta}_d} \log d(\mathbf{x_k};\, \boldsymbol{\theta}_d) - \mathbb{E}_g \nabla_{\boldsymbol{\theta}_d} \log(1 - d(g(\boldsymbol{Z}; \boldsymbol{\theta}_g); \boldsymbol{\theta}_d)) \right)$$

- Stochastic backprop is very important and **we will use it** in Parts III and IV.
- Time permitting, I will show you a **cheat** that lets us reparametrize discrete variables using a distribution that is a continuous relaxation.

GAN Training Graph

- The training procedure is illustrated on the right.
- Objects that are **random** are circled.
- Objects written in **purple** are those being optimized.
- Red denotes the fake data, blue denotes the real data.
- Note that for backpropagation to occur, we need to go back through the random element X, hence stochastic backpropagation is required.



Minimalistic GAN Implementation

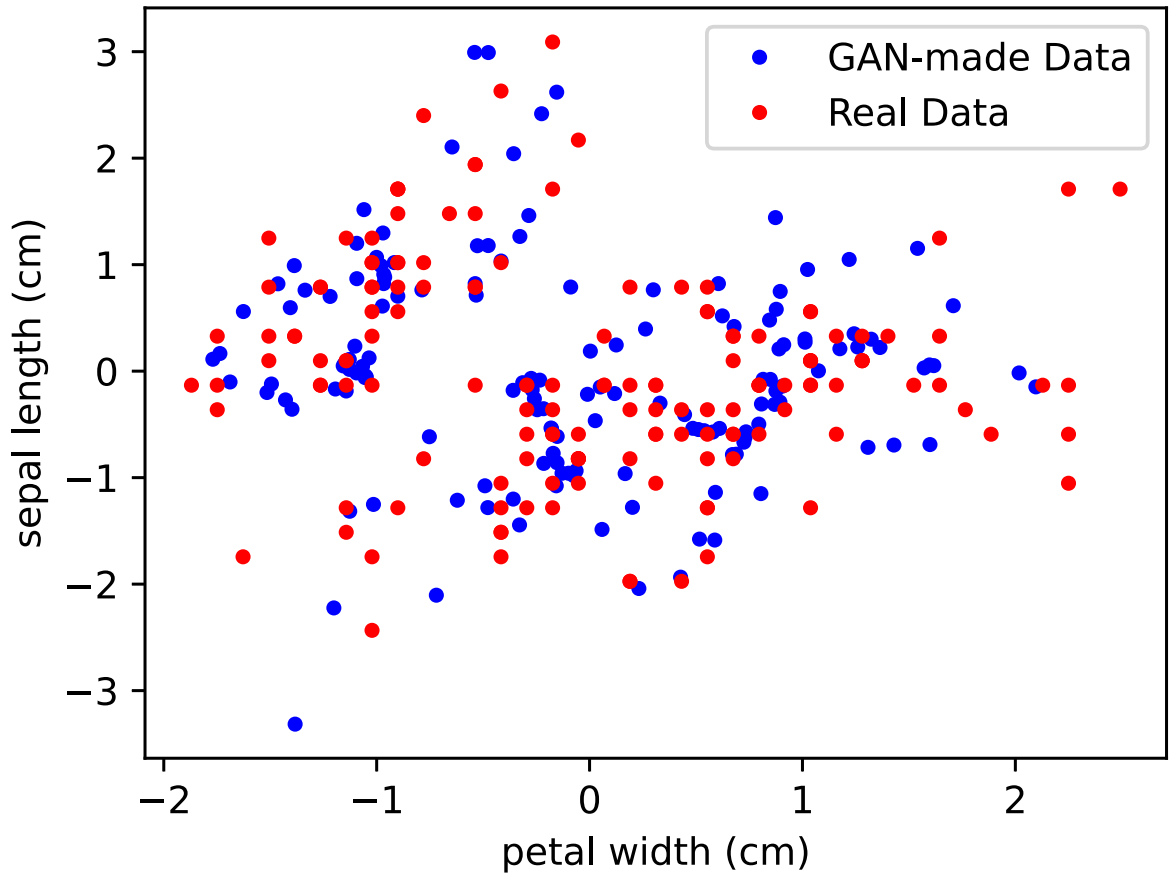
- Making a minimalistic GAN implementation in PyTorch (I call it **miniGAN**!) takes less than 50 lines of code!
- Class-Based Implementation, with two methods: (i) **Initialization (Constructor)**, (ii) **Training**

```
import torch as t
import torch.nn as nn

class miniGAN():
    def __init__(self, data, dimZ, n_hidden=25):
        dimX, self.dimZ  = data.shape[1], dimZ
        self.data = t.tensor(data, dtype=t.float)

        # create a generator net for the GAN
        self.g = nn.Sequential(nn.Linear(dimZ, n_hidden), nn.ReLU(),
                               nn.Linear(n_hidden, n_hidden), nn.ReLU(),
                               nn.Linear(n_hidden,dimX))

        # create a discriminator net for the GAN
        self.d = nn.Sequential(nn.Linear(dimX, n_hidden), nn.ReLU(),
                               nn.Linear(n_hidden, n_hidden), nn.ReLU(),
                               nn.Linear(n_hidden,1))
```

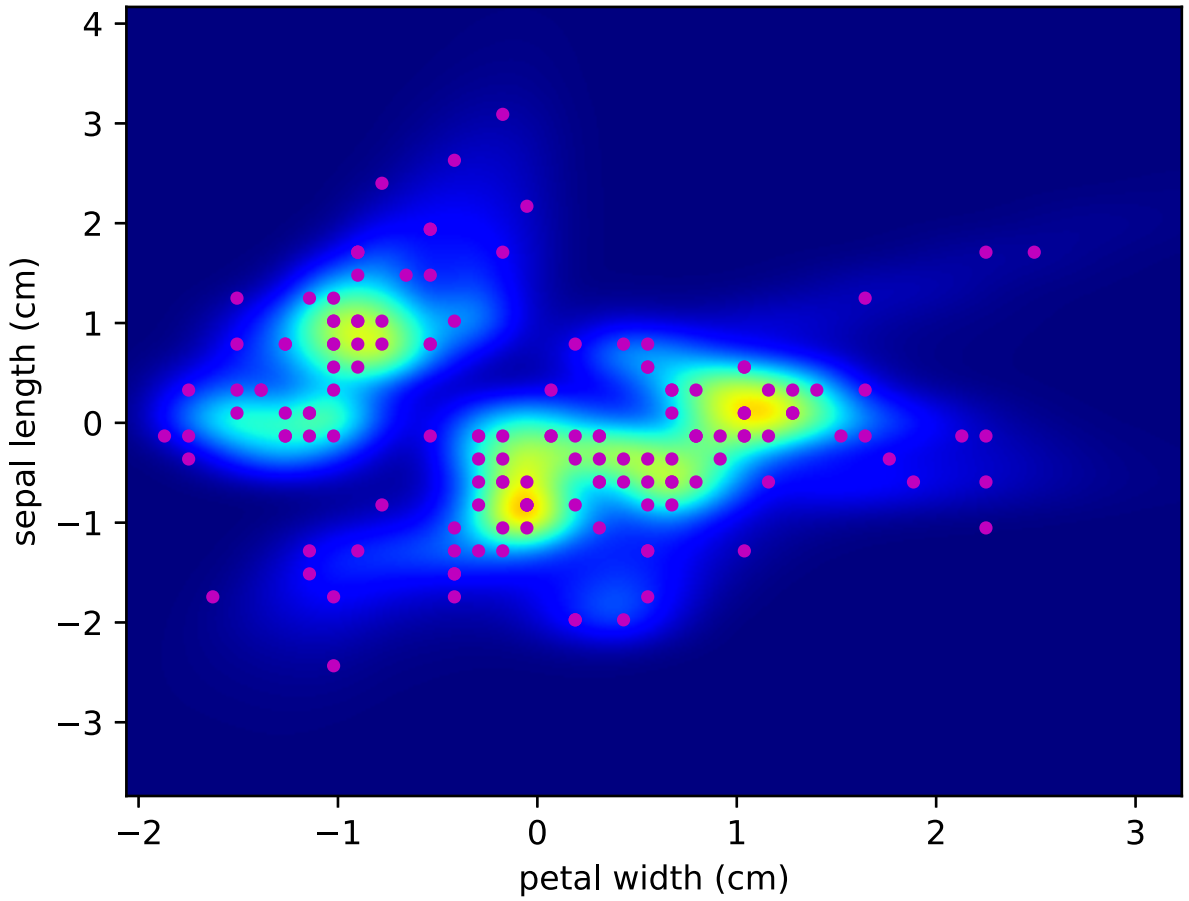
Training Procedure

```
def train_GAN(self, n_steps, n_samples = 128, d_steps=1):
    self.opt_g = torch.optim.Adam(self.g.parameters(), lr=1e-4)  # optimizer for g
    self.opt_d = torch.optim.Adam(self.d.parameters(), lr=2e-4) # optimizer for L

    for i in range(n_steps):
        for j in range(d_steps): # discriminator training
            self.opt_d.zero_grad() # clear accumulated gradients
            Z = t.randn(n_samples, self.dimZ) # draw Z ~ N(0,I)
            X = self.g(Z) # transform via the GAN's generator

            # L_d
            d_loss = -(t.mean(t.log(self.d(data))) + t.mean(t.log(1 - self.d(X))))
            d_loss.backward() # backprop to accumulate gradients
            self.opt_d.step() # take gradient descent step for theta_g

        self.opt_g.zero_grad() # clear accumulated gradients
        Z = t.randn(n_samples, self.dimZ)
        X = self.g(Z) # generate sample
        g_loss = t.mean(t.log(1-self.d(X))) # L_g
        g_loss.backward()
        self.opt_g.step()
```

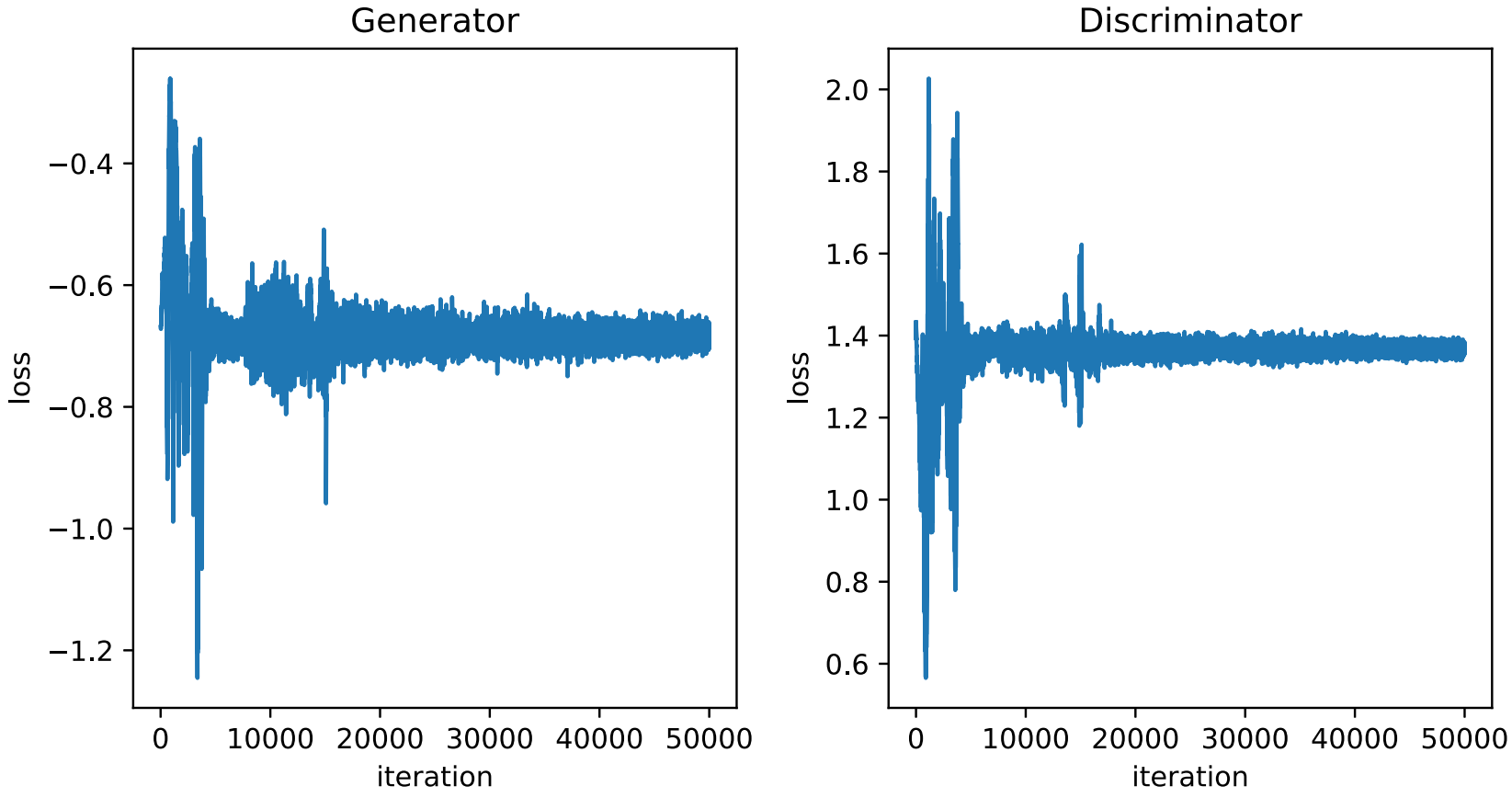
Result: GAN Trained on 2D Iris Data



Result: Kernel Density Estimate of $g_{\theta_g}(\boldsymbol{Z})$



Convergence



Key Takeaway

- GANs are not good at "learning the distribution that created the data", but **they are good at making samples that are "indistinguishable" from those from that distribution**.
- This should not be surprising, as **that is how they are designed**.
- This can be either an **incredible benefit**, or a major drawback depending on one's goal.
    - We will revisit this point again a little later.

A Journey in Latent Space

- For a fit GAN model, one can **take a walk in latent space** (i.e., $\boldsymbol{Z}$-space) by following a line/curve between two points, and looking at the (deterministic) path of generated observations in $\boldsymbol{X}$-space).
- Varying along only one of the learned independent components e.g., Z_k will visualise the effect of that **independent** component.

Figure 3: Digits obtained by linearly interpolating between coordinates in z space of the full model.

Figure from the original GAN paper.

A Quote...

"**One unusual capability of the GAN training procedure is that it can fit probability distributions that assign zero probability to the** training points. Rather than maximizing the log-probability of specific points, the generator net learns to **trace out a manifold** whose points resemble training points in some way.

Somewhat paradoxically, this means that **the model may assign a log-likelihood of negative infinity to the test set, while still representing a manifold that a human observer judges to capture the essence of the generation task**."

From Deep Learning by Goodfellow et al. (2015).

Do GANs optimize some divergence measure?

- In light of GAN training result in fits that behave very different from the $\mathrm{KL}(p||q)$ behaviour we get from flows, it begs the question of whether GANs are fitting *some* divergence measure implicitly.
- Thus, it is interesting to look at what the GAN objective is doing **probabilistically** in an idealized scenario.

What is with the GAN objective?

- The GAN training procedure is optimizing **something**, after all, we do have an objective function.
- One can obtain a little bit of encouragement by noting that if one had the optimal discriminator (i.e., out of all possible functions), then GAN training would actually be trying to minimize

$$C(g) = \max_d V(g, d) = \mathrm{KL}\left(p_{\text{data}} \middle|\middle| \frac{p_{\text{data}} + p_g}{2}\right) + \mathrm{KL}\left(p_g \middle|\middle| \frac{p_{\text{data}} + p_g}{2}\right)$$

which is proportional to the **Jensen-Shannon Divergence** between p_{data} and p_g.

- Actually, the proof is (in my opinion), quite neat and for those interested, can be found in the original paper.
- So, the loss function above is the same as saying "minimize JSD".
    - This helps one sleep better at night (maybe) knowing that GANs are **not just** some weird thing with two neural networks fighting each other.

MLE Revisited

- Recall that MLE is related to minimizing

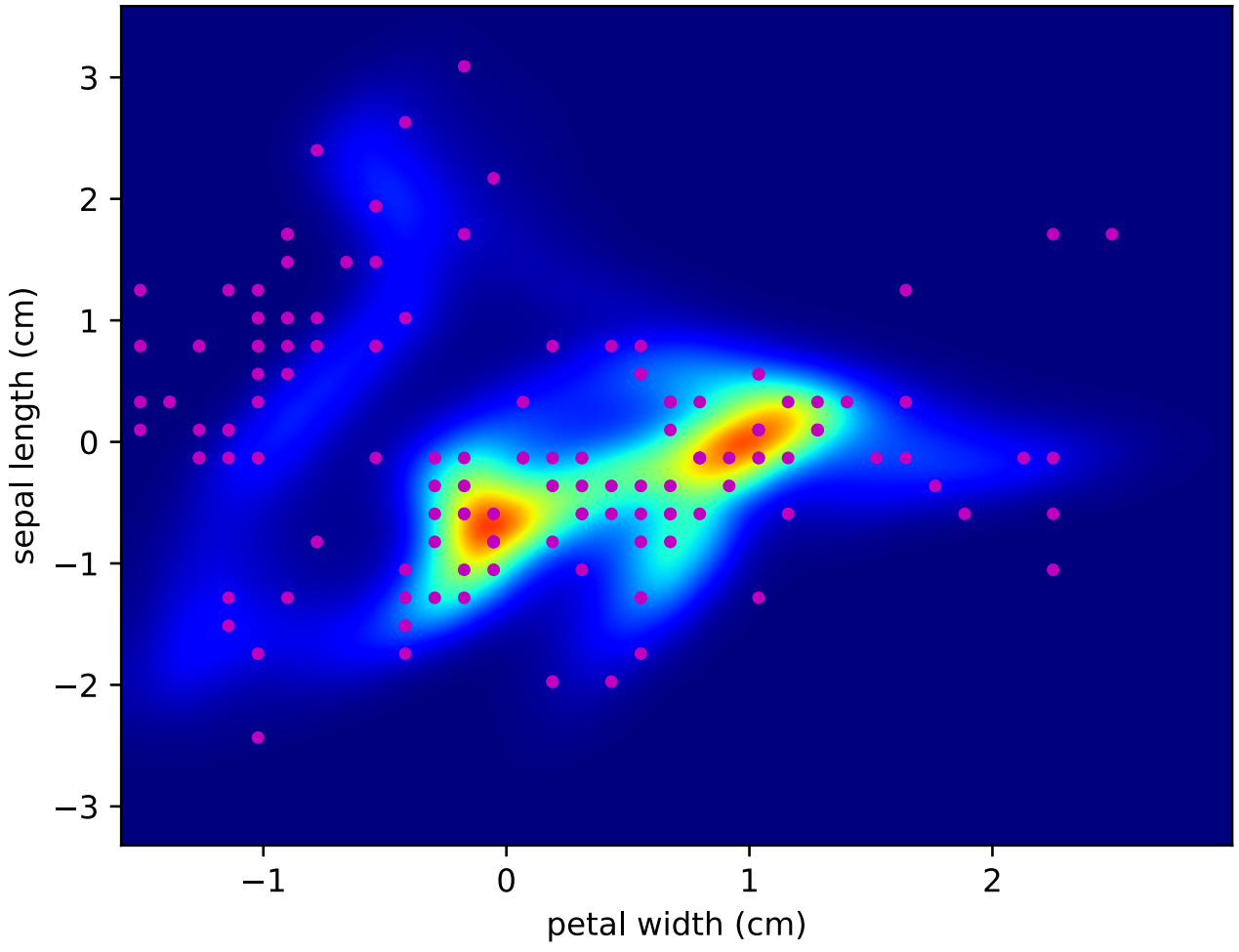
$$\mathrm{KL}(p_{\text{data}}||p_{\text{model}}) = \mathbb{E}_{p_{\text{data}}}\left[\log \frac{p_{\text{data}}(\boldsymbol{x})}{p_{\text{model}}(\boldsymbol{x})}\right].$$

- Recall also that it has a **large penalty** when $p_{\text{data}} > 0$ but p_{model} is close to zero.
  - KL divergence explodes if $p_{\text{model}}(\boldsymbol{x}) \to 0$ where $p_{\text{data}}(\boldsymbol{x}) > 0$.
  - This encourages finding a p_{model} that assigns probability mass where p_{data} has it.
- However, if $p_{\text{data}}(\boldsymbol{x})$ is close to zero in an area, the value of $p_{\text{model}}(\boldsymbol{x})$ has very little effect.
  - The interpretation is that MLE is not penalized for generating out-of-distribution (i.e., "fake-looking") samples.
  - This explains why models trained with GAN loss tend to produce more "**plausible**" images than those trained with MLE.
- However, using the GAN has it's own issues....

GAN: The Downsides

- **Mode Collapse**
  - The Generator learns only a mode of the target distribution.
  - *Underlying Issue:* GAN training encourages the generator to find an output that seems plausible to the discriminator. This can be a very special subset of the possible space.
- **Convergence**: Training is not guaranteed to converge under any practical settings.
- **Vanishing Gradients**
  - Recall the issue with the **sigmoid** function and vanishing gradients.
- **Validation**: The learned distribution is **implicit**, i.e., we do not actually know $p_{\boldsymbol{X}}$.
  - Model validation is difficult, we can't look at the (log)-likelihood of a test set of data to assess fit.
  - We also know it won't necessarily work well there anyways.

Mode Collapse



But, this person also does not exist!

- Adversarial Training produces good **quality** samples in terms of being **plausible**.
- Of course, efforts have been made to improve upon the downsides of the original GAN...

Addressing the Downsides

NSGAN: Non-Saturating (Loss) GAN

- When p_g is different to p_{data}, training is unstable, because the term below will have small gradient signal,

$$L_g^{\text{GAN}}(\boldsymbol{\theta}_g) = \mathbb{E}_g \log(1 - d(\boldsymbol{X}; \boldsymbol{\theta}_d)).$$

- We can interpret the above as *penalising* the generator for making samples that the discriminator considers fake.
    - Loss minimization with the above is equivalent to the task *"minimize the probability that the discriminator thinks generated samples are fake"*.
- However, the **non-saturating GAN** uses instead

$$L_g^{\text{NS GAN}} = -\mathbb{E}_g \log d(\boldsymbol{X};\, \boldsymbol{\theta}_d),$$

which *encourages* the generator to make samples that the discriminator considers real.

- Loss minimization with the above is equivalent to the task *"maximize the probability that the discriminator thinks generated samples are real"*.
- Fixes gradient issues but says goodbye to the nice Jensen-Shannon Divergence result. Fortunately, there is a quite recent paper that investigates NS-GAN as divergence minimization.

NS-GAN and Friends

- Actually, NS-GAN was in the original GAN paper, but there have been **so many new GANs since**...

A GAN for All Seasons

- **Many** types of GANs:

- DCGAN
- Wasserstein GAN
- Improved WGAN
- Relaxed WGAN
- Least Squares GAN
- Cramer GAN
- Energy Based GAN
- Margin Adaptation GAN
- MAGAN
- PresGAN
- TP-GAN
- Bayesian GAN

- DiscoGAN
- DualGAN
- CycleGAN
- StarGAN
- MoCoGAN
- SAGAN
- FlowGAN
- BigGAN
- SeqGAN
- RankGAN
- AnoGAN
- ⋮

A GAN for All Seasons

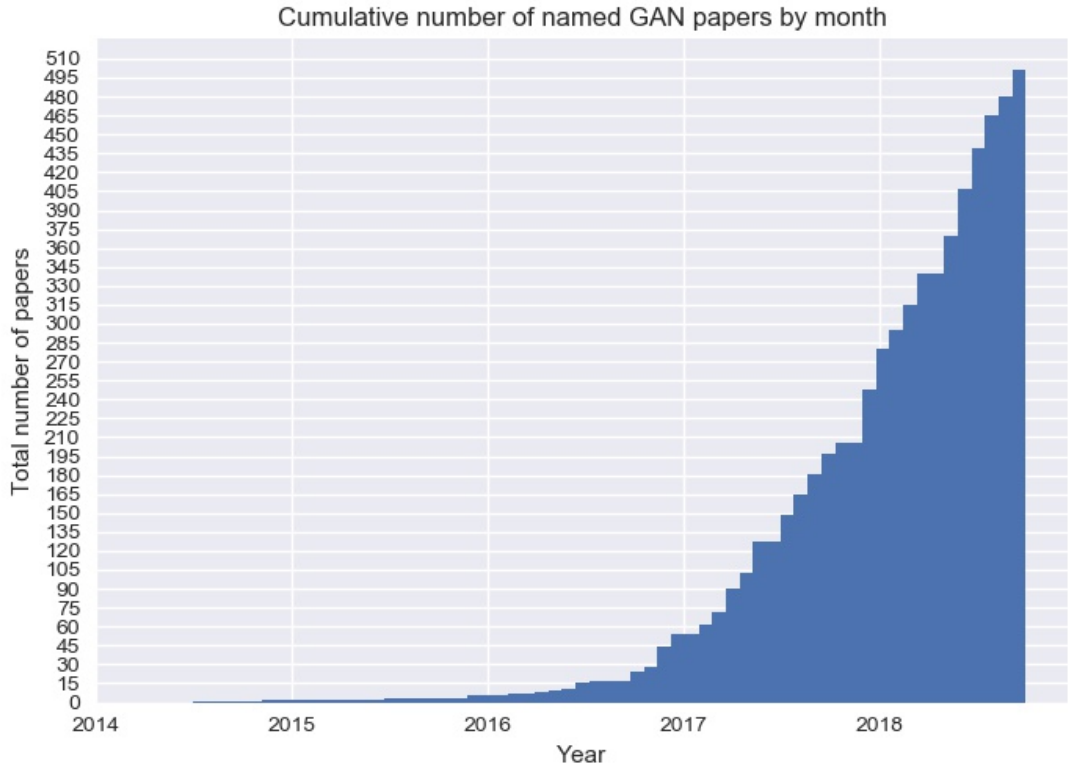
- **We will look at a few…**

- DCGAN
- **Wasserstein GAN**
- Improved WGAN
- Relaxed WGAN
- **Least Squares GAN**
- Cramer GAN
- Energy Based GAN
- Margin Adaptation GAN
- MAGAN
- PresGAN
- TP-GAN
- Bayesian GAN

- DiscoGAN
- DualGAN
- CycleGAN
- StarGAN
- MoCoGAN
- SAGAN
- **FlowGAN**
- BigGAN
- SeqGAN
- RankGAN
- AnoGAN
- ⋮

A Trip to the (GAN) Zoo

- The GAN Zoo is a nice compendium of papers (though it is only current to late 2018!).



Flow-GAN

- Grover et al., (2018), Flow-GAN: Combining Maximum Likelihood and Adversarial Learning in Generative Models, AAAI 2018.
- Idea is simple: Make $g(\cdot; \boldsymbol{\theta}_g)$ a normalizing flow.
    - Then, you have a tractable likelihood but you can pretend you don't and just train using adversarial methods.
    - Compare results using adversarial loss and maximum likelihood for the same class of models.
- The result...

"When trained adversarially, Flow-GANs generate high-quality samples but attain extremely poor log-likelihood scores, inferior even to a mixture model memorizing the training data; the **opposite** is true when trained by maximum likelihood." - Grover et al., 2018

Combining Adversarial and Log-Likelihood Loss

- If you want you can be really fancy and make a loss that interpolates between MLE and GAN Loss with a parameter $\lambda \in \mathbb{R}_+$:

$$V(g_{\theta_g}, d_{\theta_d}) = V(g, d) - \lambda \mathbb{E}_{p_{\text{data}}} \left[\log p_g(\boldsymbol{X}\,; \boldsymbol{\theta}) \right]$$

- Above, $\lambda = 0$ is pure GAN objective, and $\lambda \to \infty$ is pure (negative) log-likelihood objective. Any $\lambda \in (0, \infty)$ is some interpolation between the two.

Wasserstein GAN (Arjovsky et al., 2017)

Wasserstein GAN

- The linked paper has beautiful theory (and >7000 citations!), but methodologically can be summarized as: *Change the loss function so that you minimize* **Wasserstein** *distance instead of* **Jensen-Shannon Divergence**.
- Before we had,

$$C(g) = \max_d V(g, d) = \mathrm{KL}\left(p_{\text{data}} \middle|\middle| \frac{p_{\text{data}} + p_g}{2}\right) + \mathrm{KL}\left(p_g \middle|\middle| \frac{p_{\text{data}} + p_g}{2}\right)$$

- Wasserstein GAN replaces the **discriminator** function d with a **critic** function f and aims to minimize

$$C(g) = \mathcal{W}(p_{\text{data}}, p_{\text{g}}) = \sup_{f \in \mathcal{F}} \left\{\mathbb{E}_{p_{\text{data}}} f(\boldsymbol{X}) - \mathbb{E}_{p_g} f(\boldsymbol{X})\right\}$$

where $\mathcal{F}$ is the set of (scalar valued) 1-Lipschitz functions (we will define what that means shortly).

- Interesting (to me at least), the above is not the definition of the Wasserstein Distance, but is equivalent to it by Kantorovich-Rubinstein Duality (the proof requires some analytical techniques).

Lipschitz Functions

That is, functions f that satisfy, for all $\boldsymbol{x}_1, \boldsymbol{x}_2$,

$$|f(\boldsymbol{x}_1) - f(\boldsymbol{x}_2)| \leq K||\boldsymbol{x}_1 - \boldsymbol{x}_2||.$$

for $K = 1$. More generally, any function satisfying the above for some other $K > 0$ is called K-Lipschitz.

- We approximate $\mathcal{F}$ with a **very flexible class** of Lipschitz functions (Lipschitz neural net).
- Note that the value function is an **Integral Probability Metric**:

$$\mathcal{W}(p_{\text{data}}, p_{\text{g}}) = \sup_{f \in \mathcal{F}} \left\{ \mathbb{E}_{p_{\text{data}}} f(\boldsymbol{X}) - \mathbb{E}_{p_g} f(\boldsymbol{X}) \right\}$$

- Thus, other function classes may be used.
    - For example, we can obtain explicit solutions for the supremmum over $\mathcal{F}$ when that class of functions is the unit ball in a Reproducing Kernel Hilbert Space (see e.g., SteinGAN or MMDGan).

Wasserstein GAN (Arjovsky et al., 2017)

- The value function becomes

$$V(\boldsymbol{\theta}_g, \boldsymbol{\theta}_f) = \sup_{f \in \mathcal{F}} \left\{ \mathbb{E}_{p_{\text{data}}} f(\boldsymbol{X}) - \mathbb{E}_{p_g} f(\boldsymbol{X}) \right\}$$

- There (now) exist a number of neural networks that enforce Lipschitz constraints, but a simple way is to simply "clip" the parameters to lie in a fixed range $[-c, c]$. You just do this after each gradient step.

- Depending on c, the net has a different Lipschitz constant, however if the Lipschitz constant is not one, one obtains something simply **proportional** to the Wasserstein distance so everything is OK!

- Of course...

  Weight clipping is a clearly terrible way *to enforce a Lipschitz constraint*. - Arjovsky et al., 2017 (WGAN Paper)

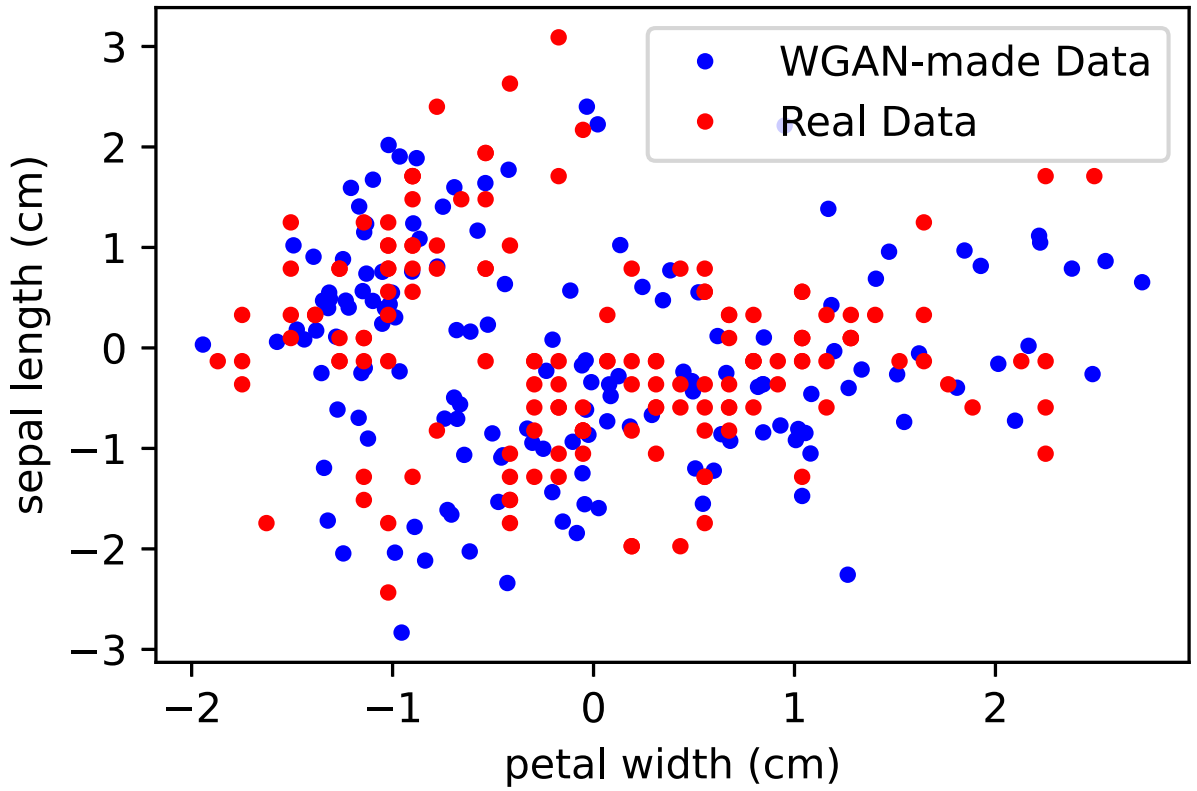
  - But then again...

  "**In no experiment did we see evidence of mode collapse** *for the WGAN algorithm.*" - Arjovsky et al., 2017 (WGAN Paper)

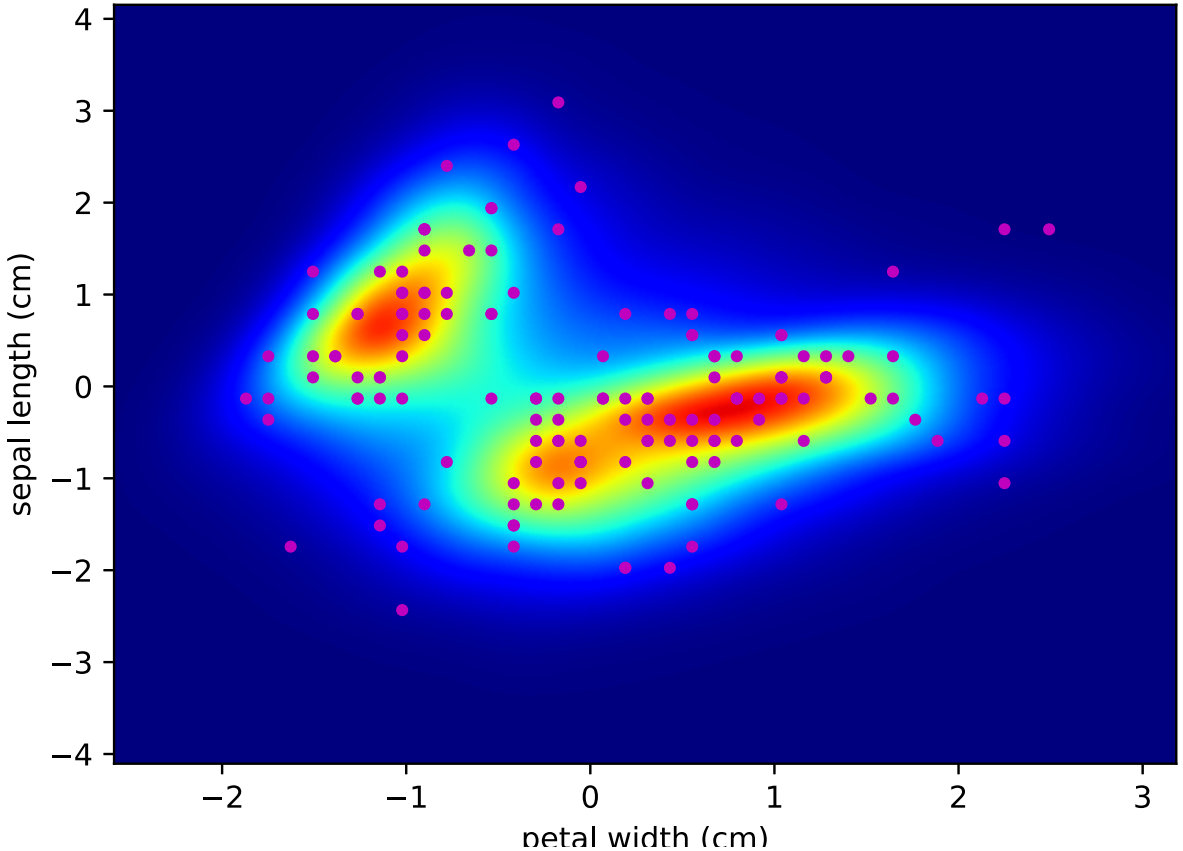
WGAN Implementation

```
for i in range(n_steps):
    for j in range(d_steps):
        self.opt_d.zero_grad()
        Z = t.randn(n_samples, self.dimZ)
        X = self.g(Z)
        d_loss = -(t.mean(self.d(self.data)) - t.mean(self.d(X)))

        d_loss.backward()

        self.opt_d.step()

        for p in self.d.parameters
            p.data.clamp_(-0.01, 0.01)


    self.opt_g.zero_grad()
    X = self.g(t.randn(n_samples, self.dimZ))
    g_loss = -t.mean(self.d(X))

    g_loss.backward()
    self.opt_g.step()
```

WGAN Result


WGAN-made Data
Real Data
sepal length (cm)
petal width (cm)
3
2
1
0
−1
−2
−3
−2
−1
0
1
2

WGAN Result: Density Estimate



Least Squares GAN

Least Squares GAN (LS-GAN)

- Proposed in the paper:

Mao et al., (2017), **Least Squares Generative Adversarial networks**, Proceedings of the IEEE International Conference on Computer Vision (ICCV), 2017, pp. 2794-2802.

- Idea is again simple: use a **least squares** loss.

$$L_g(\boldsymbol{\theta}_g) = \mathbb{E}_{p_{\text{data}}}\left[(d(\boldsymbol{X}) - b)^2\right] + \mathbb{E}_{p_g}\left[(d(\boldsymbol{X}) - a)^2\right]$$

$$L_d(\boldsymbol{\theta}_d) = \mathbb{E}_{p_g}\left[(d(\boldsymbol{X}) - c)^2\right]$$

- In the above
  - a is the discriminator's target for fake data
  - b is the discriminator's target for real data
  - c is the value the generator is trying to make the discriminator assign to its data.

- **Motivation**: Gradients vanish when fake samples that are "bad" are still considered plausible by the discriminator. They should get a bigger penalty.

LS-GAN: How good is it?

- Provided $b - c = 1$ and $b - a = 2$, minimizing the aforementioned losses is equivalent to minimizing the specific *Pearson χ^2-Divergence*

$$\chi^2 \left(\frac{p_d + p_g}{2} \middle\| p_g \right)$$

where

$$\chi^2(p||q) = \mathbb{E}_q \left[\left(\frac{p(\boldsymbol{X})}{q(\boldsymbol{X})} - 1 \right)^2 \right] = \mathbb{V}\mathrm{ar}_q \left[\frac{p(\boldsymbol{X})}{q(\boldsymbol{X})} \right].$$

- One choice of a, b, c satisfying the conditions to obtain the χ^2 divergence is

$$L_g(\boldsymbol{\theta}_g) = \mathbb{E}_{p_{\text{data}}} \left[(d(\boldsymbol{X}) - 1)^2 \right] + \mathbb{E}_{p_g} \left[(d(\boldsymbol{X}) + 1)^2 \right]$$

$$L_d(\boldsymbol{\theta}_d) = \mathbb{E}_{p_g} \left[(d(\boldsymbol{X}))^2 \right]$$

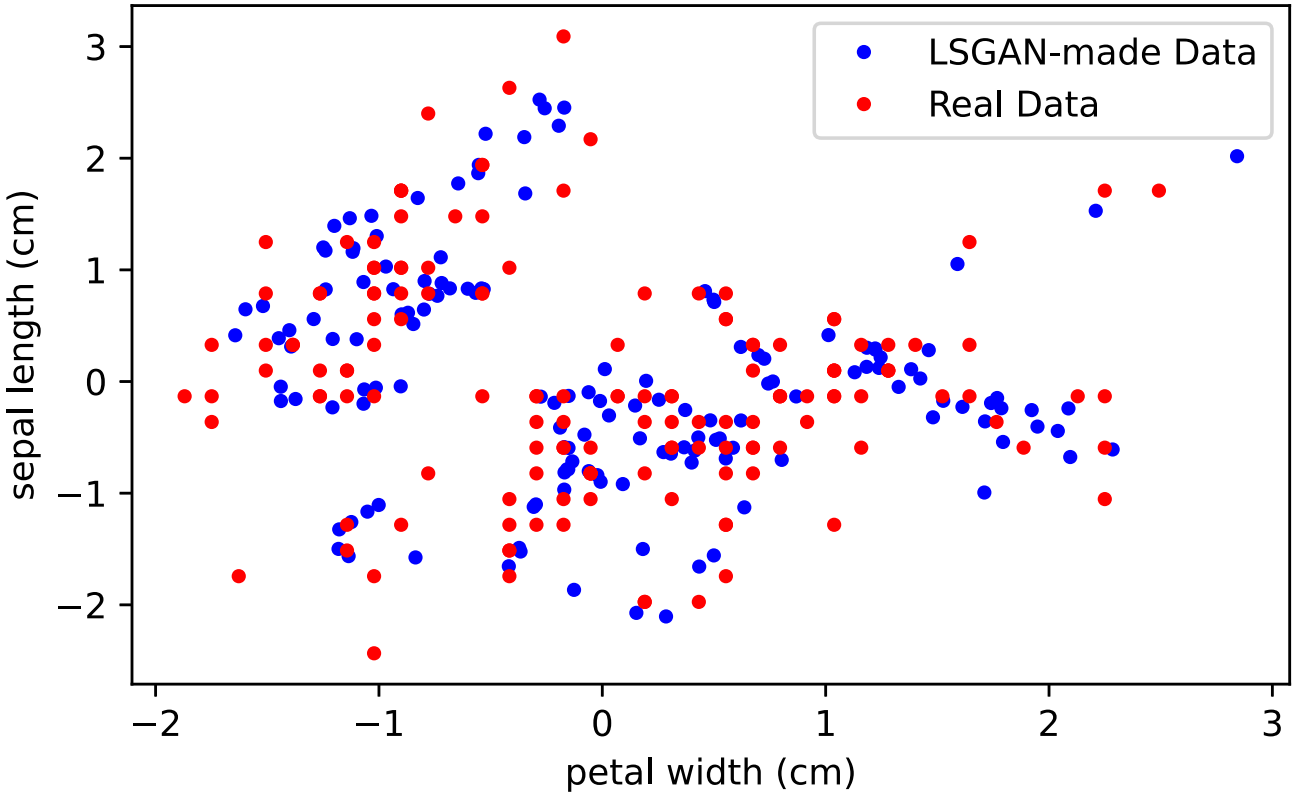
- Another option (that has no **known** divergence) is to just use binary labels (1 for real, 0 for fake, generator wants the discriminator to think it is generating samples with value 1). **This is what the authors used in their experiments**.

LS-GAN Implementation

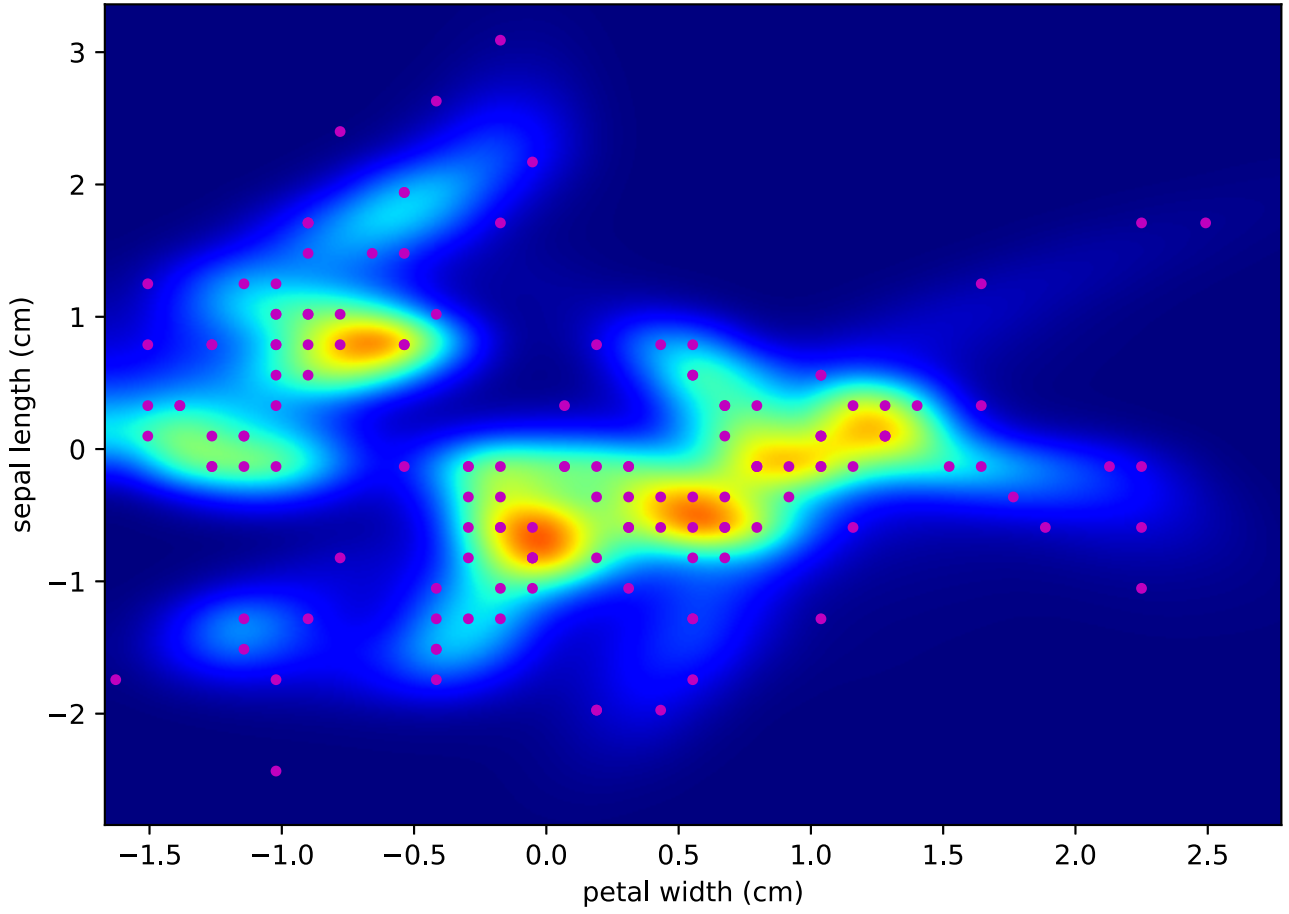
- The implementation of LS-GAN is actually **very** simple, arguably **simpler**.

```
def train_GAN(self, n_steps, n_samples = 128, d_steps=1):

     self.opt_g = torch.optim.Adam(self.g.parameters(), lr=1e-4)
     self.opt_d = torch.optim.Adam(self.d.parameters(), lr=2e-4)

     for i in range(n_steps):
         for j in range(d_steps):
             self.opt_d.zero_grad()
             X = self.generate_samples(n_samples)
             d_loss = (t.mean((self.d(self.data)-1)**2) + t.mean((self.d(X))**2))/2
             d_loss.backward()
             self.opt_d.step()

         self.opt_g.zero_grad()
         X = self.generate_samples(n_samples)
         g_loss = t.mean((self.d(X)-1)**2)/2
         g_loss.backward()
         self.opt_g.step()
```

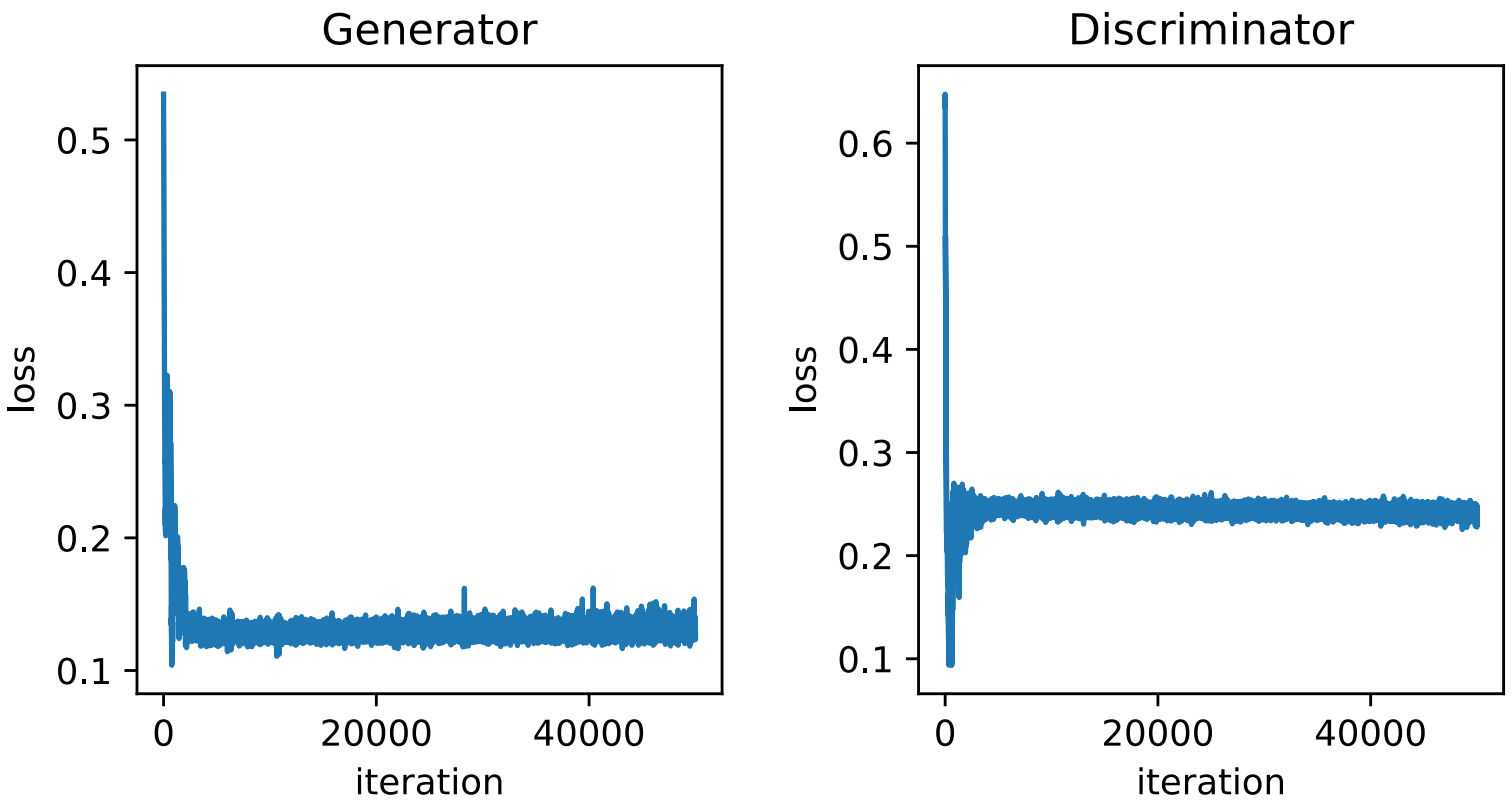
LSGAN Result



LSGAN Result: Density Estimate



Convergence



LSGAN

- Authors argue that LS-GAN training is more stable, produces better quality samples (images), and that LS-GAN seems more robust to mode collapse.
- You will get to play around testing some GANs on a similar example in the first tutorial!
    - I have made a class for you which has three GAN implementations in one. :)

OK, enough with the alternate loss functions, lets try to accomplish something a little different...

Conditional GAN (CGAN)

Conditional GAN (Mizra & Osindero, 2014)

- Learns a **conditional distribution** : $p(\boldsymbol{x}|\boldsymbol{y})$
- The conditioning variable $\boldsymbol{y}$ can be *any* kind of information, e.g., class labels or continuous data.
  - Often, $\boldsymbol{y}$ is referred to as the **context** variable.
- Simply feed $\boldsymbol{y}$ to both generator and discriminator (additional input).

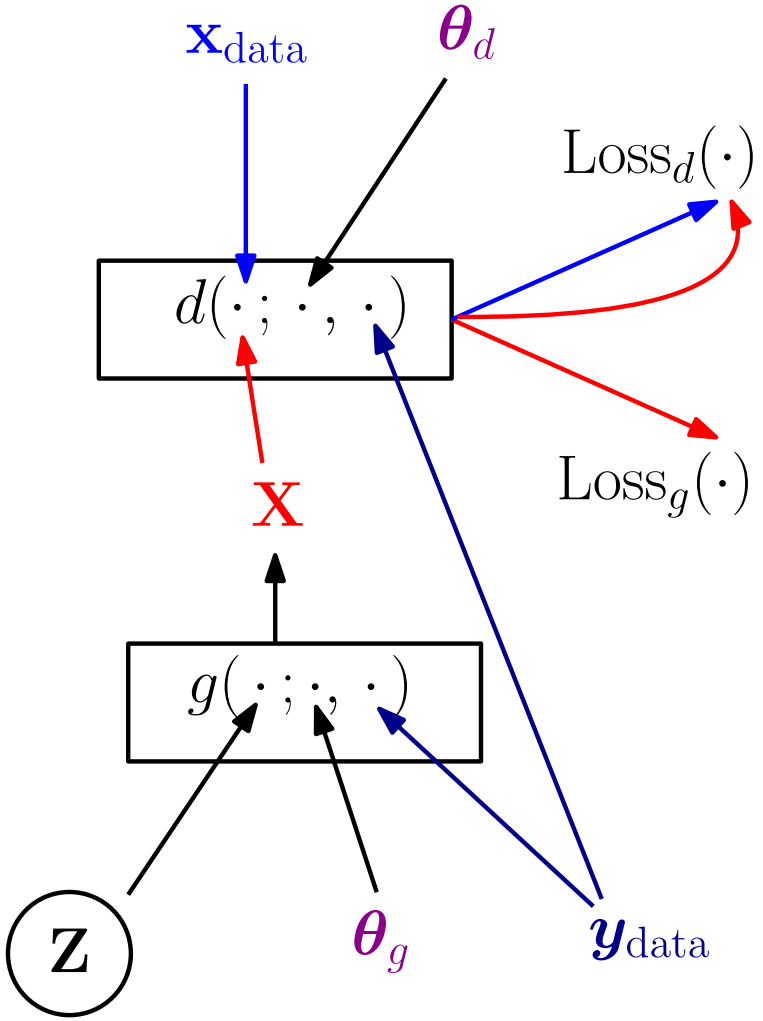
$$L_g^{\text{CGAN}}(\boldsymbol{\theta}_g) = \mathbb{E}_g \log(1 - d(\boldsymbol{X}, \boldsymbol{Y}; \boldsymbol{\theta}_d))$$

$$L_d^{\text{CGAN}}(\boldsymbol{\theta}_g) = -\left(\frac{1}{n} \sum_{k=1}^{n} \log d(\mathbf{x_k}, \boldsymbol{y}_k;\, \boldsymbol{\theta}_d) - \mathbb{E}_g \log(1 - d(g(\boldsymbol{Z}, \boldsymbol{Y}), \boldsymbol{Y}; \boldsymbol{\theta}_d)) \right)$$

- Same principle applies for other loss functions (e.g., W-CGAN, LS-CGAN).
- In principle you can make a joint model over the "context" by first fitting the marginal $p(\boldsymbol{y})$, and then $p(\boldsymbol{x}|\boldsymbol{y})$ via CGAN.
  - Can also train jointly if you desire.

Conditional GAN

- Conditional GAN is a straightforward extension, one just needs to add extra inputs to d and g and feed both the data.



Key Takeaway

- GANs are not good at "learning the distribution that created the data", but they are good at making samples that are "indistinguishable" from those from that distribution.
- This should not be surprising, as **that is how they are designed**.

Recommended Survey Articles

- An early tutorial summary that discusses GAN as well as other generative models you will see / have seen in this course:
  - Goodfellow, I. (2016). **NIPS 2016 tutorial: Generative adversarial networks**.
- Simple to read survey giving some recent developments (and overviews ways of assessing performance)
  - Pan, Z., Yu, W., Yi, X., Khan, A., Yuan, F., & Zheng, Y. (2019). **Recent progress on generative adversarial networks (GANs): A survey**. IEEE Access, 7, 36322-36333.
- A comparison study that makes one wonder (somewhat) about all the GAN variants…
  - Lucic, M., Kurach, K., Michalski, M., Gelly, S., & Bousquet, O. (2018). **Are GANS created equal? a large-scale study**. Advances in Neural Information Processing Systems 31 (NeurIPS 2018)

Bonus: Reparametrization of Categorical Variables (approximately)

Bonus: Reparametrization of Categorical Variables (approximately)

- Suppose that we wish our GAN to output values from a categorical distribution.
- As a toy example, consider the following generative model for a dependent Bernoulli vector.

$$\boldsymbol{Z} \sim \mathrm{N}(\mathbf{0}, \boldsymbol{I})$$

$$\boldsymbol{\eta} = \mathrm{sigmoid}(g(\boldsymbol{Z}; \boldsymbol{\theta}_g))$$

$$X_k | \boldsymbol{Z} \sim_{\mathrm{ind}} \mathrm{Bernoulli}(\eta_k), \quad k = 1, \ldots, d..$$

- Problem here is that stochastic backpropagation requires the **score function** estimator as discrete variables are *not* reparametrizable in a useful way.
- Later we will see that we can train such models using **Variational Learning** to deal with the intractable likelihood, but for now we do not want to use the **score function method** for stochastic backpropagation (remember it performs very poorly).

Categorical Variables

- Introducing the **softmax** distribution

$$p_{\text{softmax}}(k\,;\, w_1, \ldots, w_K) = \frac{\exp(w_k)}{\sum_{k=1}^{K} \exp(w_k)}, \quad k = 1, \ldots, K$$

- Note that the above is simply an (unconstrained) reparametrization of the **multinomial distribution**.
  - Softmax has $\boldsymbol{w} \in \mathbb{R}^K$ instead of the multinomial's $\boldsymbol{p} \in \mathcal{S}^K$, where $\mathcal{S}^K$ is the set of $\boldsymbol{p}$ such that $\boldsymbol{p} \geq 0$ elementwise and $\sum_k p_k = 1$.
- It is still of course a **discrete** distribution. We wish to have reparametrization gradients but this is not possible.

The Gumbel-Softmax aka Concrete Distribution

- Let's do something just a little "dodgy" that will allow us to obtain reparametrization gradients through categorical distributions.
- It is possible to create a **continuous relaxation** of the aforementioned discrete distributions so we can obtain reparametrization gradients.
- Two names because it was independently proposed by (**at the same conference in the same year, no less**)
  - E. Jang, S. Gu, and B. Poole. **Categorical Reparameterization with Gumbel-Softmax** (2017), ICLR 2017.
  - C. J. Maddison, A. Mnih, and Y. W. Teh. **The Concrete Distribution: A Continuous Relaxation of Discrete Random Variables** (2017), ICLR 2017.
- Parameters are $\alpha_1, \ldots, \alpha_K \in \mathbb{R}_+$. Simulation is obtained by draw $G \sim \text{Gumbel}$ and return

$$X_k = \frac{\exp(\lambda^{-1} \cdot (\log a_k + G_k))}{\sum_{j=1}^{n} \exp(\lambda^{-1} \cdot (\log \alpha_j + G_i))}$$

- In the limit that $\lambda \to 0$, samples match those from an associated softmax (multinomial) distribution.

Gumbel-Softmax Distribution in PyTorch

- The module `torch.distributions` has it implemented as `RelaxedBernoulli` and `RelaxedOneHotCategorical`

```
import torch

p = torch.tensor([0.5])
distX = torch.distributions.RelaxedBernoulli(probs = p, temperature = 1)
X = distX.rsample()
```