PyTorch and GAN

Charlie Lu

What is PyTorch

- Automatic differentiation engine
- Ndarray library with GPU support
- Gradient based optimization package
- Utilities (data loading, etc.)

Why PyTorch

- Python-first philosophy
- Tape-based autograd system (reverse-mode automatic differentiation)
- Decipherable stack traces. Easy to debug.
- Easily extend the Python ecosystem (numpy, scipy, Cython)

Dynamically Compiled Graph (DCG)

Debugging made easy.

print(foo)

PBD to set breakpoints

No obscure error message hidden behind abstractions

Identify bottlenecks with your favorite Python profiler such as SnakeViz

More flexible architectures

No compilation time (All core kernels pre-compiled)

Linear style of programing (Chainer, Dynet, TF-eager)

JIT compilation to fuse and optimize operations in graph

Lazy evaluation (cache subgraphs)

Graph compiled every iteration (Can implement some crazy/stupid research ideas easily e.g. random number of layers in hidden cells every every batch)

PyTorch

Pro

- Dynamic Graph
- Easy to Debug
- Pythonic
- Numpy integration

Con

- No mobile support
- Only Unix support for now
- Not as good for distributed yet

TensorFlow

Pro

- Keras
- TensorBoard
- Tensorflow Serving

Con

- Static Graph
- Hard to debug
- Slow

How to install?

```
conda install pytorch torchvision cuda80 -c soumith
```

Packaged with Cuda, Cudnn, NCCL already

Initialize a random tensor

```
In [4]: torch.Tensor(5, 3)
```

```
Out[4]:  2.4878e+04  4.5692e-41  2.4878e+04
         4.5692e-41 -2.9205e+19  4.5691e-41
         1.2277e-02  4.5692e-41 -4.0170e+19
         4.5691e-41  1.2277e-02  4.5692e-41
         0.0000e+00  0.0000e+00  0.0000e+00
        [torch.FloatTensor of size 5x3]
```

From a uniform distribution

```
In [5]: torch.Tensor(5, 3).uniform_(-1, 1)
```

```
Out[5]: -0.2767 -0.1082 -0.1339
        -0.6477  0.3098  0.1642
        -0.1125 -0.2104  0.8962
        -0.6573  0.9669 -0.3806
         0.8008 -0.3860  0.6816
        [torch.FloatTensor of size 5x3]
```

Get it's shape

```
In [6]: x = torch.Tensor(5, 3).uniform_(-1, 1)
        print(x.size())
```

```
torch.Size([5, 3])
```

Tensors

Data type	Tensor
32-bit floating point	torch.FloatTensor
64-bit floating point	torch.DoubleTensor
16-bit floating point	torch.HalfTensor
8-bit integer (unsigned)	torch.ByteTensor
8-bit integer (signed)	torch.CharTensor
16-bit integer (signed)	torch.ShortTensor
32-bit integer (signed)	torch.IntTensor
64-bit integer (signed)	torch.LongTensor

Simple mathematical operations

```
y = x * torch.randn(5, 3)
print(y)
```

```
 0.2200 -0.0368  0.4494
-0.2577 -0.0343  0.1587
-0.7503 -0.1729  0.0453
 0.9296 -0.1067 -0.6402
-0.3276  0.0158 -0.0552
[torch.FloatTensor of size 5x3]
```

```
y = x / torch.sqrt(torch.randn(5, 3) ** 2)
print(y)
```

```
 0.2820 -0.1633 -4.4346
-1.6809  0.2066 -0.8261
-0.6464  0.9758  0.2542
 0.5789  0.1890 -0.4662
 5.3183  0.0236 -0.1403
[torch.FloatTensor of size 5x3]
```

Broadcasting

```
print (x.size())
y = x + torch.randn(5, 1)
print(y)
```

```
torch.Size([5, 3])

 0.1919 -0.5006 -1.2410
-0.8080  0.1407 -0.6193
-1.6629 -0.1580 -0.3921
 1.0395  0.7069 -0.1459
 1.9027  1.4343  1.2299
[torch.FloatTensor of size 5x3]
```

Reshape

```
y = torch.randn(5, 10, 15)
print(y.size())
print(y.view(-1, 15).size())  # Same as doing y.view(50, 15)
print(y.view(-1, 15).unsqueeze(1).size()) # Adds a dimension at index 1.
print(y.view(-1, 15).unsqueeze(1).squeeze().size())
# If input is of shape: (Ax1xBxCx1xD)(Ax1xBxCx1xD) then the out Tensor will be of shape: (AxBxCxD)
(AxBxCxD)
print()
print(y.transpose(0, 1).size())
print(y.transpose(1, 2).size())
print(y.transpose(0, 1).transpose(1, 2).size())
print(y.permute(1, 2, 0).size())
```

```
torch.Size([5, 10, 15])
torch.Size([50, 15])
torch.Size([50, 1, 15])
torch.Size([50, 15])

torch.Size([10, 5, 15])
torch.Size([5, 15, 10])
torch.Size([10, 15, 5])
torch.Size([10, 15, 5])
```

Repeat

```
print(y.view(-1, 15).unsqueeze(1).expand(50, 100, 15).size())
print(y.view(-1, 15).unsqueeze(1).expand_as(torch.randn(50, 100, 15)).size())
```

```
torch.Size([50, 100, 15])
torch.Size([50, 100, 15])
```

Concatenate

```
# 2 is the dimension over which the tensors are concatenated
print(torch.cat([y, y], 2).size())
# stack concatenates the sequence of tensors along a new dimension.
print(torch.stack([y, y], 0).size())
```

```
torch.Size([5, 10, 30])
torch.Size([2, 5, 10, 15])
```

Advanced Indexing

```
y = torch.randn(2, 3, 4)
print(y[[1, 0, 1, 1]].size())

# PyTorch doesn't support negative strides yet so ::-1 does not work.
rev_idx = torch.arange(1, -1, -1).long()
print(y[rev_idx].size())
```

```
torch.Size([4, 3, 4])
torch.Size([2, 3, 4])
```

Numpy Bridge

```
import numpy as np
A = torch.ones(3, 2)
B = A.numpy()
C = np.random.randn(2, 3)
A @ B
_.shape # (3, 3)
```

```
import torch
A = np.ones(3, 2)
B = torch.from_numpy(A)
C = torch.randn(2, 3)
A @ B
_.size() # (3, 3)
```

Tensor methods (Numpy in comments when different) `

```
# Calculation
torch.min(x, [dim])
torch.max(x, [dim])
torch.trace
torch.sum
torch.cumsum
torch.mean
torch.std
torch.prod
torch.mm # np.dot
torch.cumprod
torch.cat # np.concatenate
torch.cmul # np.multiply
torch.sort(x, [dim])

# Shape manipulation
x.reshape
x.resize
x.transpose() or x.t() # x.T
x.view() # x.flatten
x.squeeze
```

```
# Attributes
x.size() # x.shape
x.stride()
x.dim() # x.ndim
x.data()
x.nElement() # x.size
x.type() # x.dtype

# Constructors
torch.Tensor(2,2) # np.empty([2,2])
torch.eye
torch.ones
torch.zeros
torch.diag
torch.Tensor([[1,2],[3,4]]) # np.array([[1,2],[3,4]])
x.clone() # np.copy(x)
torch.range(0,9) # np.arange(10)
torch.linspace(1,
torch.logspace
```

```python
import numpy as numpy


N, D_in, H, D_out = 64, 1000, 100, 10

x = np.random.randn(N, D_in)
y = np.random.randn(N, D_out)

w1 = np.random.randn(D_in, H)
w2 = np.random.randn(H, D_out)

learning_rate = 1e-6

for t in range(500):
    # foward pass
    h = x.dot(w1)
    h_relu = np.maimum(h, 0)
    y_pred = h_relu.dot(w2)

    loss = np.square(y_pred - y).sum()
    print(t, loss)

    # backprop
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.T.dot(grad_y_pred)
    grad_h_relu = grad_y_pred.dot(w2.T)
    grad_h = grad_h_relu.copy()
    grad_h[h < 0] = 0
    grad_w1 = x.T.dot(grad_h)

    # update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2

```

```python
import torch

N, D_in, H, D_out = 64, 1000, 100, 10
dtype = torch.FloatTensor # Use torch.cuda.FloatTensor for GPU

x = torch.randn(N, D_in).type(dtype)
y = torch.randn(N, D_out).type(dtype)

w1 = torch.randn(D_in, H).type(dtype)
w2 = torch.randn(H, D_out).type(dtype)

learning_rate = 1e-6

for t in range(500):
    # foward pass
    h = x.mm(w1)
    h_relu = h.clamp(min=0)
    y_pred = h_relu.mm(w2)

    loss = (y_pred - y).pow(2).sum()
    print(t, loss)

    # backprop
    grad_y_pred = 2.0 * (y_pred - y)
    grad_w2 = h_relu.t().mm(grad_y_pred)
    grad_h_relu = grad_y_pred.mm(w2.t())
    grad_h = grad_h_relu.clone()
    grad_h[h < 0] = 0
    grad_w1 = x.t().mm(grad_h)

    # update weights
    w1 -= learning_rate * grad_w1
    w2 -= learning_rate * grad_w2

```

Variables

- Use to wrap a tensor and records operations applied to it
- Usually for differentiation
- Gradient of the loss w.r.t. this variable is accumulated into `.grad`
- Tell variable to needs to store gradient with requires_grad=True

```
x = Variable(torch.Tensor(5, 3).uniform_(-1, 1), requires_grad=True)
y = Variable(torch.Tensor(5, 3).uniform_(-1, 1), requires_grad=True)
z = x ** 2 + 3 * y
z.backward(gradient=torch.ones(5, 3))
```

```
# eq computes element-wise equality
torch.eq(x.grad, 2 * x)
```

```
Variable containing:
 1  1  1
 1  1  1
 1  1  1
 1  1  1
 1  1  1
[torch.ByteTensor of size 5x3]
```

```
y.grad
```

```
Variable containing:
 3  3  3
 3  3  3
 3  3  3
 3  3  3
 3  3  3
[torch.FloatTensor of size 5x3]
```

```
x = Variable(torch.Tensor(5, 3).uniform_(-1, 1), requires_grad=True)
y = Variable(torch.Tensor(5, 3).uniform_(-1, 1), requires_grad=True)
z = x ** 2 + 3 * y
dz_dx = torch.autograd.grad(z, x, grad_outputs=torch.ones(5, 3))
dz_dy = torch.autograd.grad(z, y, grad_outputs=torch.ones(5, 3))
```

Autograd

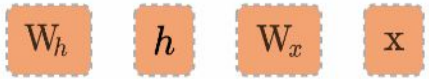
Calling `.backward()` clears the current computation graph.

To retain the graph after the backward pass u `loss.backward(retain_graph=True)`. T lets you reuse intermediate variables

Clear gradients each time with `.zero_grad()`

or they will overwrite previous gradients

A graph is created on the fly




```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```

Running on GPU

Tensors can be moved onto GPU using the `.cuda` function.

```
# let us run this cell only if CUDA is available
if torch.cuda.is_available():
    x = x.cuda()
    y = y.cuda()
    x + y
```

GPU support

```
x = torch.cuda.HalfTensor(5, 3).uniform_(-1, 1)
y = torch.cuda.HalfTensor(3, 5).uniform_(-1, 1)
torch.matmul(x, y)
```

```
 0.2456  1.1543  0.5376  0.4358 -0.0369
 0.8247 -0.4143 -0.7188  0.3953  0.2573
-0.1346  0.7329  0.5156  0.0864 -0.1349
-0.3555  0.3135  0.3921 -0.1428 -0.1368
-0.4385  0.5601  0.6533 -0.2793 -0.5220
[torch.cuda.HalfTensor of size 5x5 (GPU 0)]
```

Loading Data

`torch.utils.data.Dataset` is an abstract class representing a dataset.

Your custom dataset should inherit Dataset and override the following methods:

__len__ so that len(dataset) returns the size of the dataset.

__getitem__ to support the indexing such that dataset[i]

can be used to get iith sample

The following dataset loaders are available:

- MNIST
- COCO (Captioning and Detection)
- LSUN Classification
- ImageFolder
- Imagenet-12
- CIFAR10 and CIFAR100
- STL10
- SVHN
- PhotoTour

> Datasets have the API: - `__getitem__` - `__len__` They all subclass from `torch.utils.data.Dataset` Hence, they can all be multi-threaded (python multiprocessing) using standard torch.utils.data.DataLoader.

Subclassing nn.Module

- Abstract class that defines methods needed to train neural network
- Subclass nn.Module by overloading `forward()` and `__init__()` methods
- Modules can also contain other Modules, allowing to nest them in a tree structure.

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # 1 input channel, 6 output channels, 5x5 convolution
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.conv2 = nn.Conv2d(6, 16, 5)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:] # all dimensions except batch
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

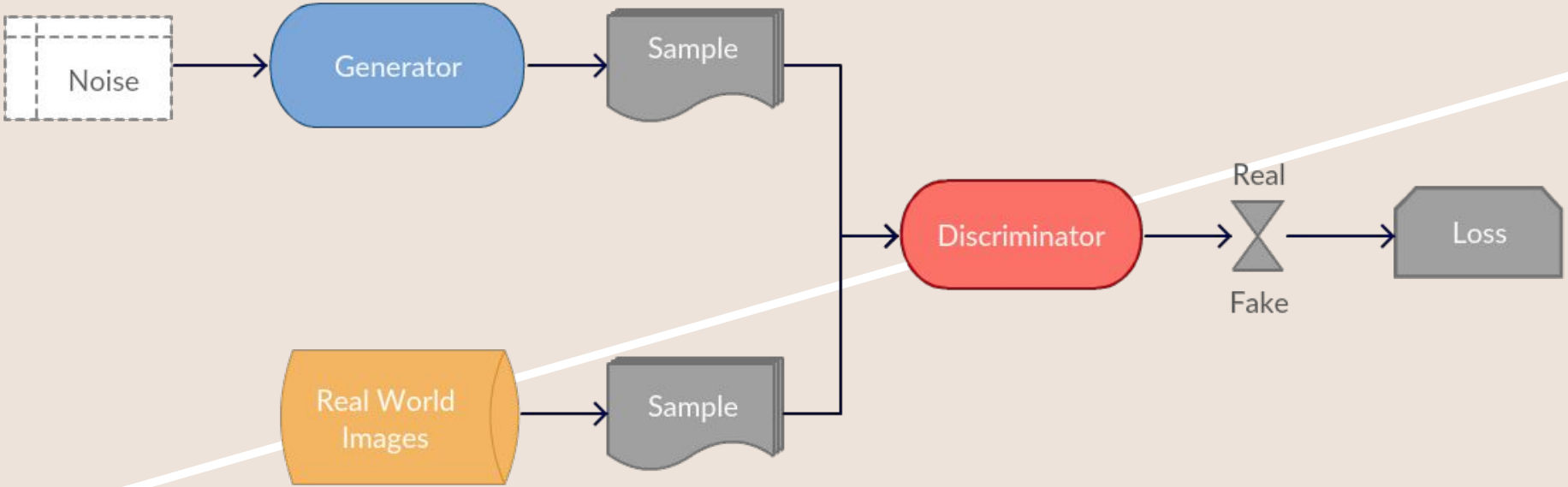
Multiprocessing

torch.multiprocessing is a drop in replacement for Python's multiprocessing module.

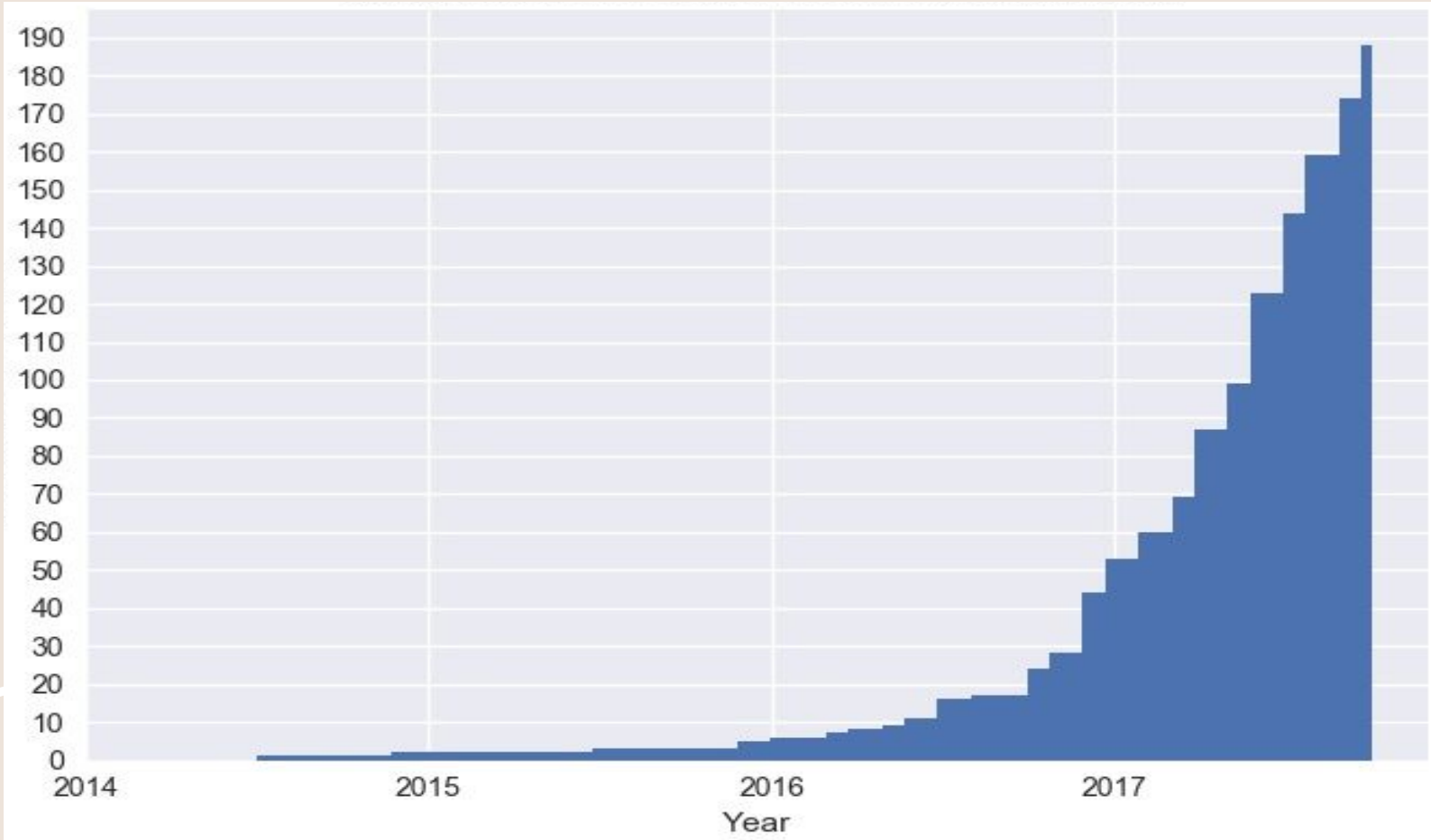
It supports the exact same operations, but extends it, so that all tensors sent through a multiprocessing.Queue, will have their data moved into shared memory and will only send a handle to another process.

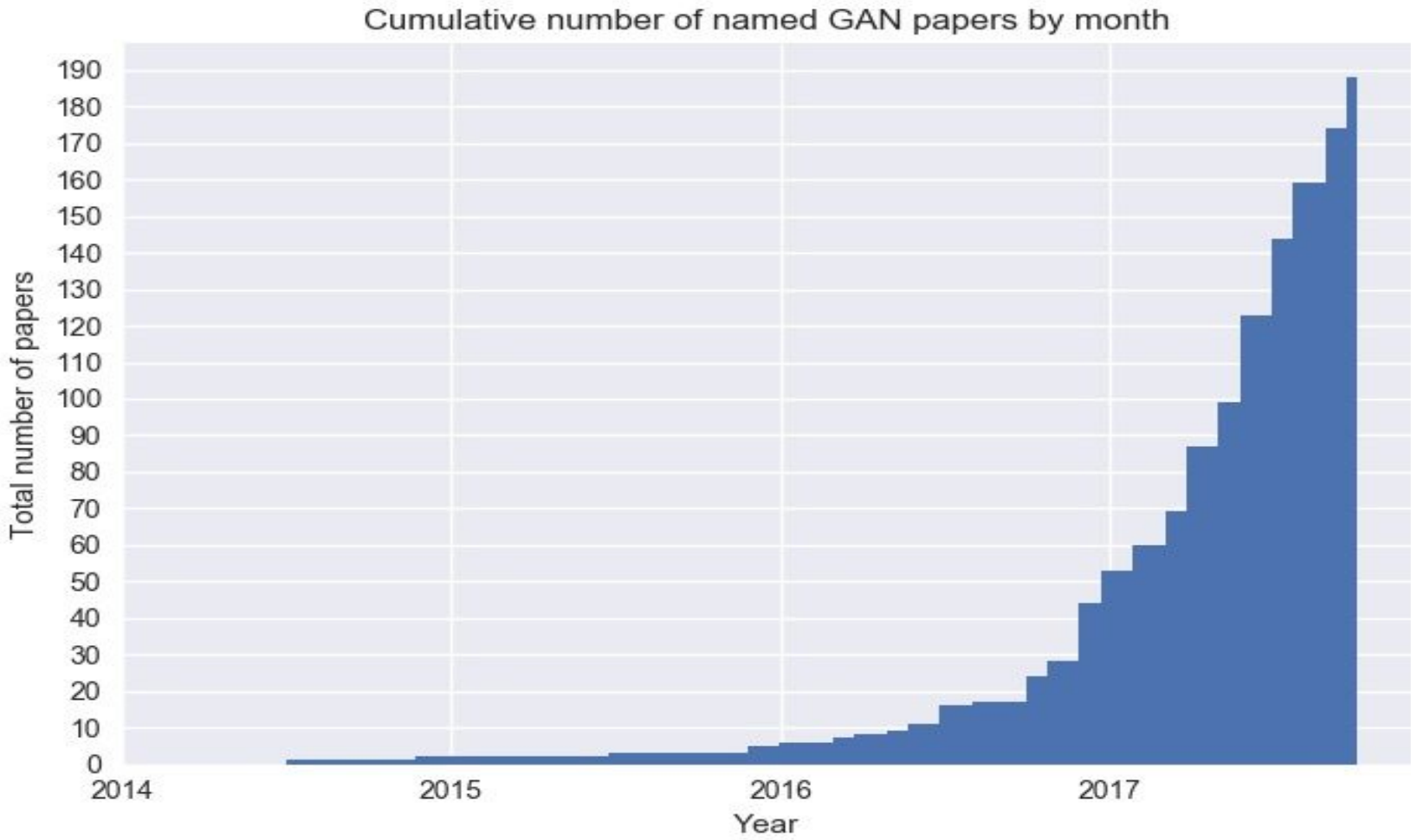
```
import torch.multiprocessing as mp
from model import MyModel

def train(model):
    # Construct data_loader, optimizer, etc.
    for data, labels in data_loader:
        optimizer.zero_grad()
        loss_fn(model(data), labels).backward()
        optimizer.step()  # This updates shared parameters

if __name__ == '__main__':
    num_processes = 4
    model = MyModel()
    # NOTE: this is required for the ``fork`` method to work
    model.share_memory()
    processes = []
    for rank in range(num_processes):
        p = mp.Process(target=train, args=(model,))
        p.start()
        processes.append(p)
    for p in processes:
      p.join()
```

Generative Adversarial Networks



"Generative Adversarial Networks is the most interesting idea in the last ten years in machine learning." – Yann Lecun, Director of Faceboook AI Research


190
180
170
160
150
140
130
120
110
100
90
80
70
60
50
40
30
20
10
0
2014
2015
2016
2017
Year


Cumulative number of named GAN papers by month
Total number of papers
0
10
20
30
40
50
60
70
80
90
100
110
120
130
140
150
160
170
180
190
2014
2015
2016
2017
Year

input

output

A large bird has large thighs and large wings that have white wingbars

input

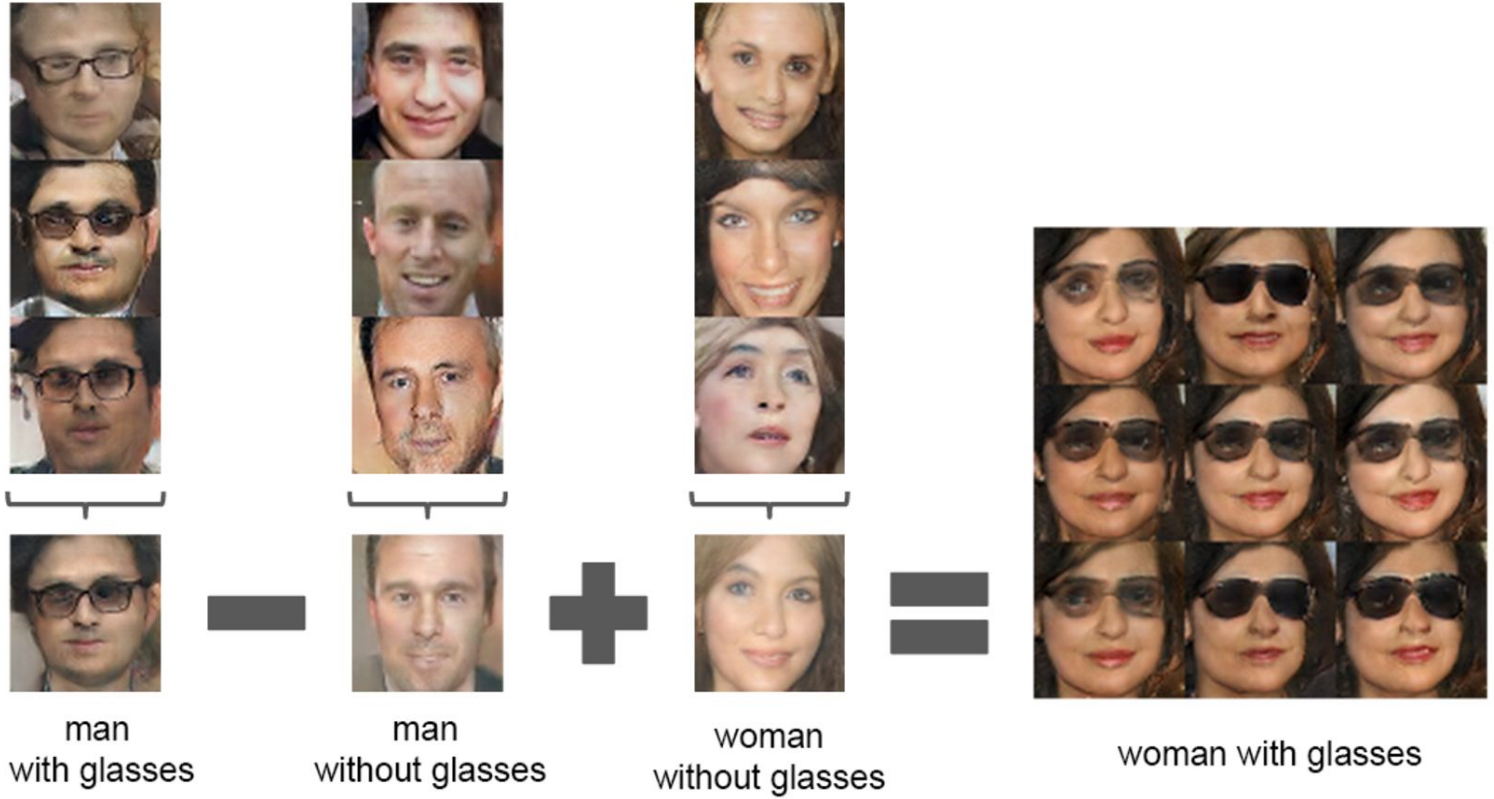
output

The small bird has a red head with feathers that fade from red to gray from head to tail

input

output

This flower has petals that are white and has pink shading	This flower has a lot of small purple petals in a dome-like configuration	This flower has long thin yellow petals and a lot of yellow anthers in the center	This flower is pink, white, and yellow in color, and has petals that are striped	This flower is white and yellow in color, with petals that are wavy and smooth	This flower has upturned petals which are thin and orange with rounded edges	This flower has petals that are dark pink with white edges and pink stamen



“Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”

Alec Radford, Luke Metz, Soumith Chintala

Figure 8: A ”turn” vector was created from four averaged samples of faces looking left vs looking right. By adding interpolations along this axis to random samples we were able to reliably transform their pose.

“Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks”

Alec Radford, Luke Metz, Soumith Chintala

input

outputs

“Image-to-Image Translation with Conditional Adversarial Networks”

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros

“Image-to-Image Translation with Conditional Adversarial Networks”

Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, Alexei A. Efros

http://3dgan.csail.mit.edu/

https://www.youtube.com/watch?v=9reHvktowLY

http://carlvondrick.com/tinyvideo/

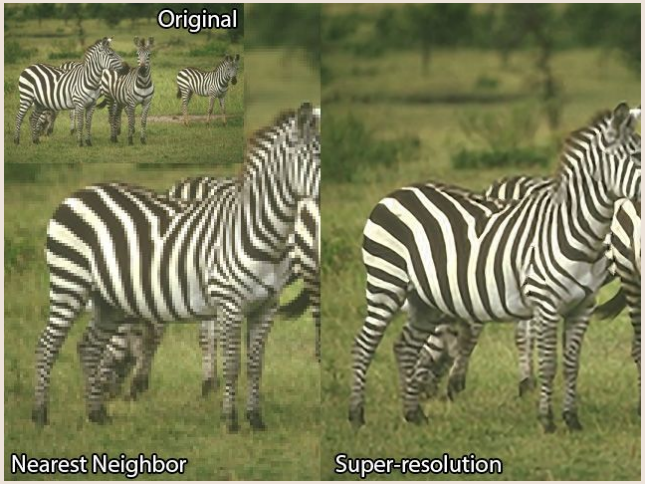
Generative Models

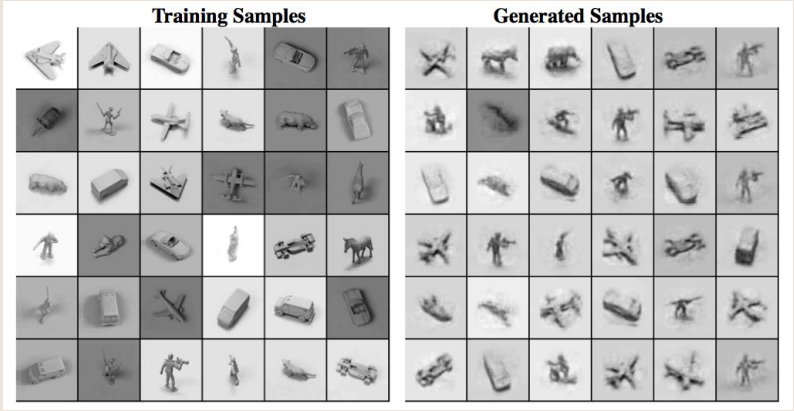
Simulate possible futures for planning in Reinforcement Learning

Missing data/ unlabeled data - semi-supervised/unsupervised learning

Multimodal outputs

Super resolution of single images





Types of Generative models

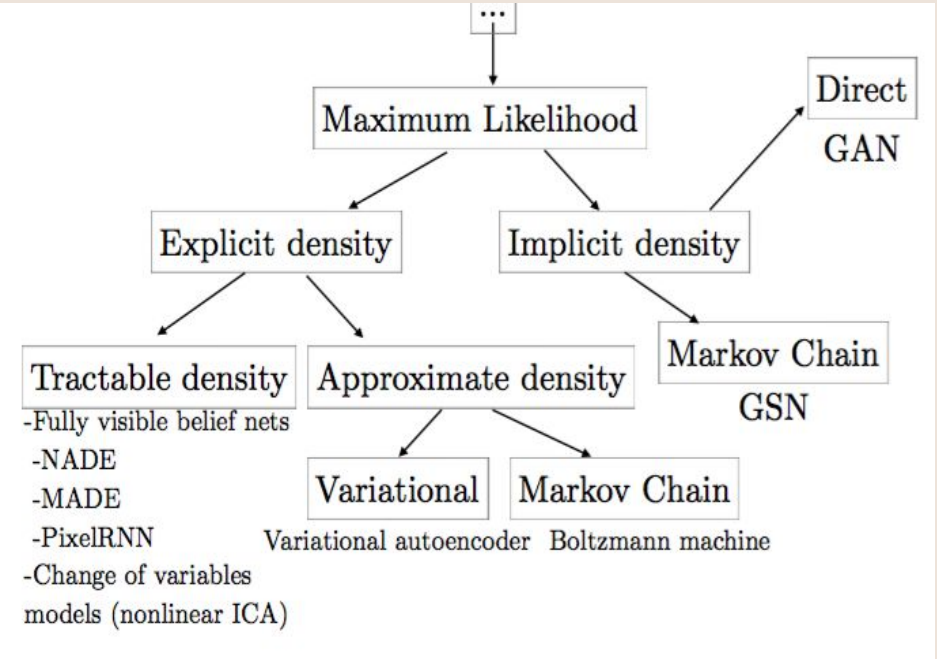


Figure 9: Deep generative models that can learn via the principle of maximim likelihood differ with respect to how they represent or approximate the likelihood. On the left branch of this taxonomic tree, models construct an explicit density, $p_{\text{model}}(\boldsymbol{x}; \boldsymbol{\theta})$, and thus an explicit likelihood which can be maximized. Among these explicit density models, the density may be computationally tractable, or it may be intractable, meaning that to maximize the likelihood it is necessary to make either variatioanl approximations or Monte Carlo approximations (or both). On the right branch of the tree, the model does not explicitly represent a probability distribution over the space where the data lies. Instead, the model provides some way of interacting less directly with this probability distribution. Typically the indirect means of interacting with the probability distribution is the ability to draw samples from it. Some of these implicit models that offer the ability to sample from the distribution do so using a Markov Chain; the model defines a way to stochastically transform an existing sample in order to obtain another sample from the same distribution. Others are able to generate a sample in a single step, starting without any input. While the models used for GANs can sometimes be constructed to define an explicit density, the training algorithm for GANs makes use only of the model's ability to generate samples. GANs are thus trained using the strategy from the rightmost leaf of the tree: using an implicit model that samples directly from the distribution represented by the model.

From Goodfellow's Tutorial on GANs NIPS 2016


Epoch
00'10"05
30
INSERT COIN
Discriminator
Generator
D
G
Angkor Wat

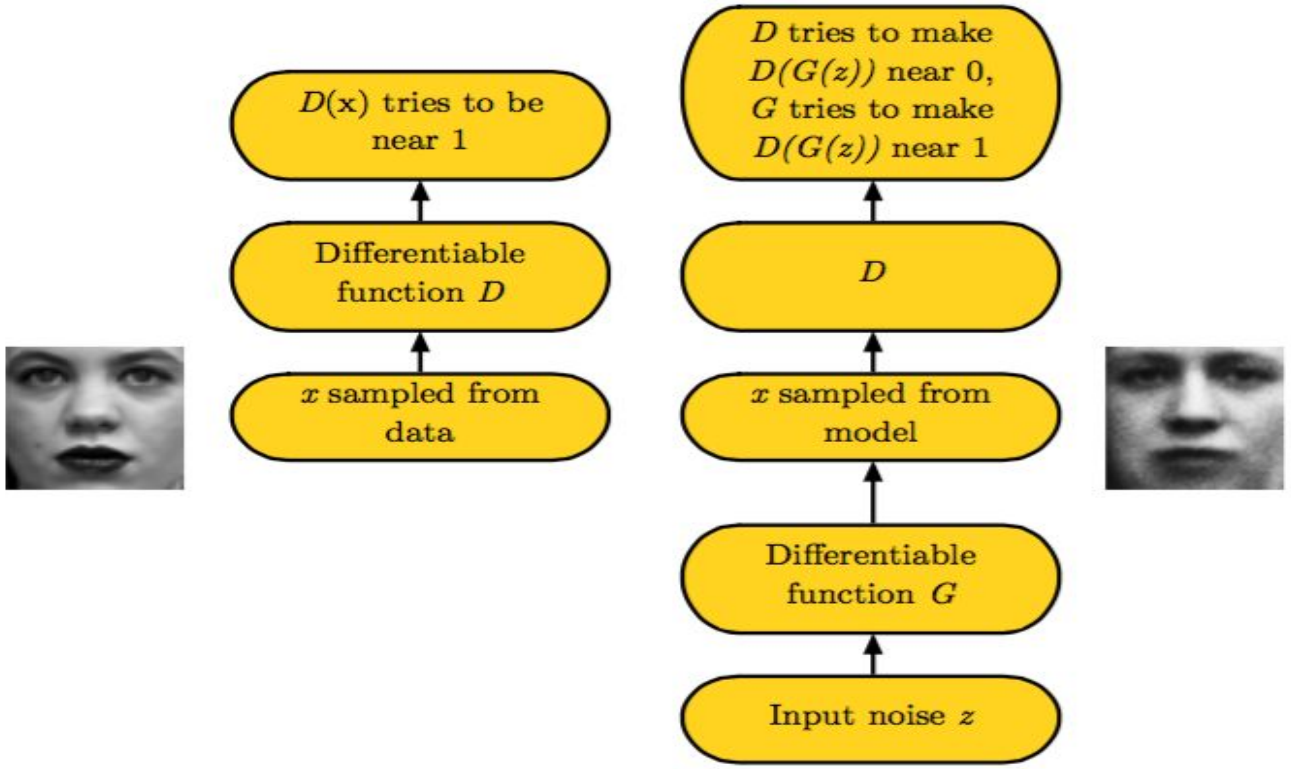


Figure 12: The GAN framework pits two adversaries against each other in a game. Each player is represented by a differentiable function controlled by a set of parameters. Typically these functions are implemented as deep neural networks. The game plays out in two scenarios. In one scenario, training examples $\boldsymbol{x}$ are randomly sampled from the training set and used as input for the first player, the discriminator, represented by the function D. The goal of the discriminator is to output the probability that its input is real rather than fake, under the assumption that half of the inputs it is ever shown are real and half are fake. In this first scenario, the goal of the discriminator is for $D(\boldsymbol{x})$ to be near 1. In the second scenario, inputs $\boldsymbol{z}$ to the generator are randomly sampled from the model's prior over the latent variables. The discriminator then receives input $G(\boldsymbol{z})$, a fake sample created by the generator. In this scenario, both players participate. The discriminator strives to make $D(G(z))$ approach 0 while the generative strives to make the same quantity approach 1. If both models have sufficient capacity, then the Nash equilibrium of this game corresponds to the $G(\boldsymbol{z})$ being drawn from the same distribution as the training data, and $D(\boldsymbol{x}) = \frac{1}{2}$ for all $\boldsymbol{x}$.

D: Detective

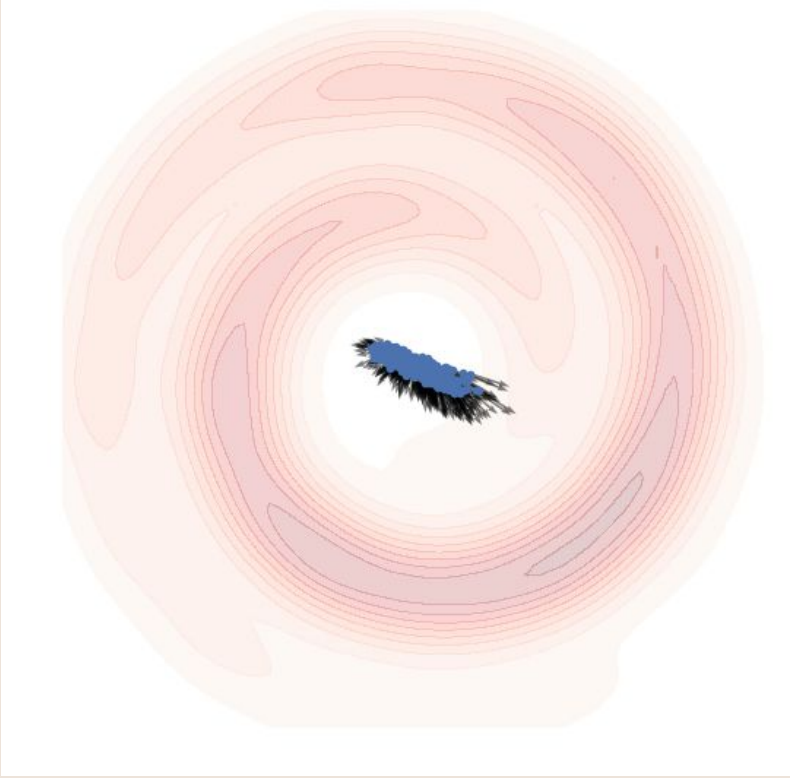
R: Real Data

G: Generator (Forger)

I: Input for Generator

Game Theory

generative model minimising KL divergence from some Swiss-roll toy data



Goal is to find Nash Equilibrium of a non-convex game with continuous, high dimensional parameters

Equilibrium is saddle point of discriminator loss

Generator minimizes the log-probability of the discriminator being correct

Can represent as a two player zero-sum Minimax game

Resembles Jensen-Shannon divergence

http://www.inference.vc/an-alternative-update-rule-for-generative-adversarial-networks/

Losses

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_z \left[e^{\sigma^{-1}(D(G(z)))} \right]$$

Maximum Likelihood version

$$J^{(D)} = -\frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}} \log D(\boldsymbol{x}) - \frac{1}{2}\mathbb{E}_{\boldsymbol{z}} \log\left(1 - D\left(G(\boldsymbol{z})\right)\right)$$

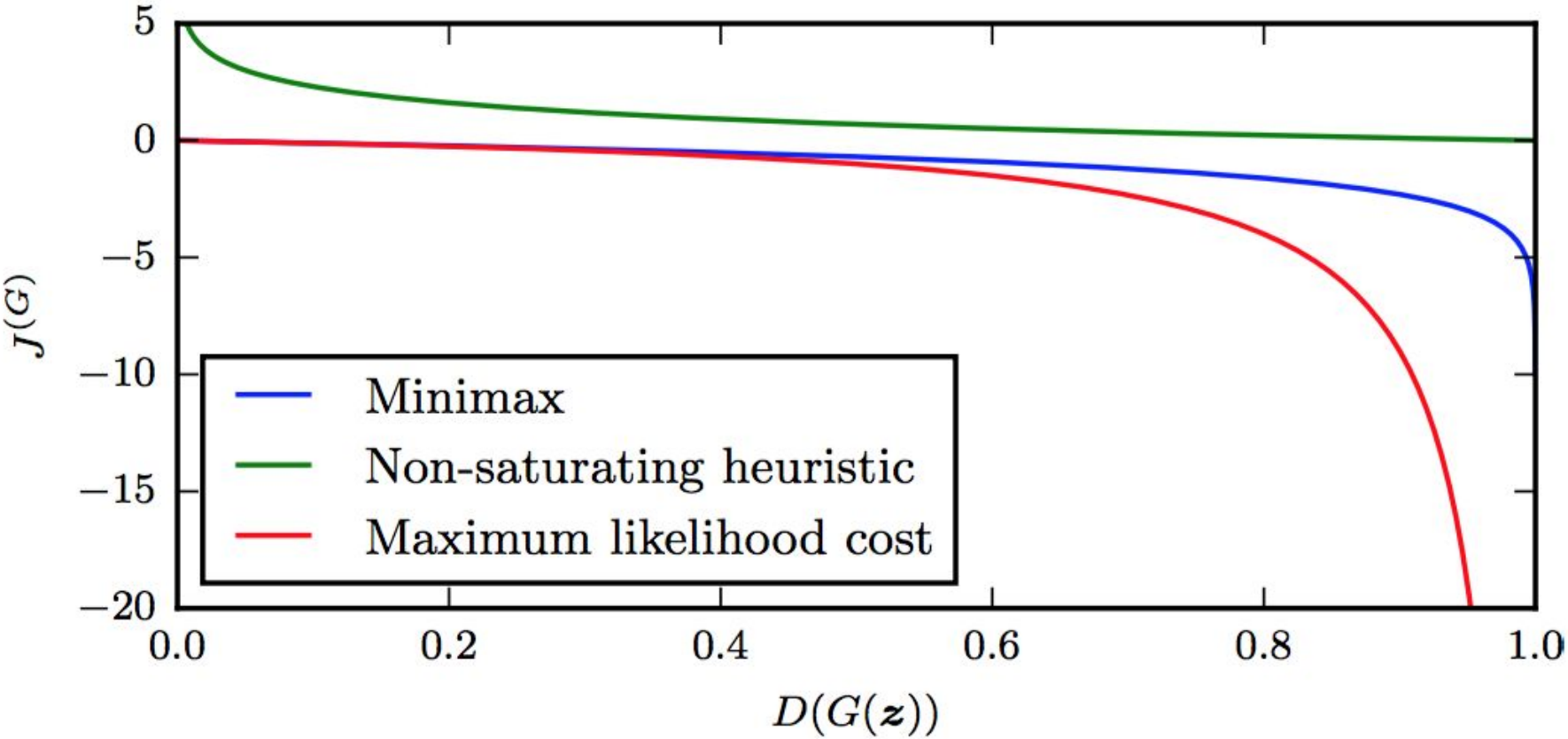
$$J^{(G)} = -J^{(D)}$$

Minimax version

$$J^{(D)} = -\frac{1}{2}\mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}} \log D(\boldsymbol{x}) - \frac{1}{2}\mathbb{E}_{\boldsymbol{z}} \log\left(1 - D\left(G(\boldsymbol{z})\right)\right)$$

$$J^{(G)} = -\frac{1}{2}\mathbb{E}_{\boldsymbol{z}} \log D\left(G(\boldsymbol{z})\right)$$

Non-saturating version



https://danieltakeshi.github.io/2017/03/05/understanding-generative-adversarial-networks/

Simultaneous GD

Generalization of gradient descent

Non-convergence is biggest problem with GAN

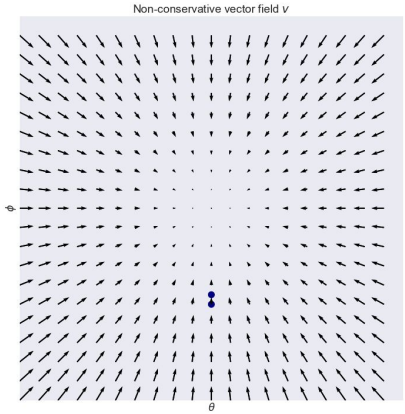
May not approach saddle point because of non-conservative vector field

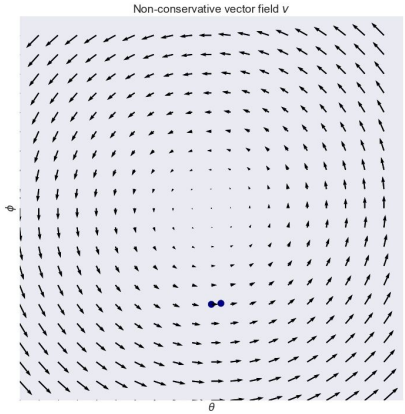
Can oscillate indefinitely (See figure)

Mode collapse is worst form of non-convergence (generates same version of example; doesn't generate diverse sample set)

http://www.inference.vc/my-notes-on-the-numerics-of-gans/

https://arxiv.org/abs/1705.10461





Directions of research

How to stabilize training and prevent mode collapse? Wasserstein loss, F-GAN

How to evaluate GAN performance? (General problem to generative models)

How to use discrete outputs with GAN? Sequences of characters and words

Supervised Discriminator for semi-supervised learning. Have multiple classes instead of just real and fake. Odena 2016, Salimans et al 2016

Connections to RL. “GANs as actor-critic” Pfau and Vinyals 2016. “GANs as inverse reinforcement” Finn et al 2016. “GANs for imitation learning” Ho and Ermon 2016

“Bayesian GANs” Saatci and Wilson 2017.

GAN hacks

https://github.com/soumith/ganhacks

Normalize inputs

Sample noise from gaussian instead of uniform

Use Batchnorm

Avoid sparse gradients:ReLU, MaxPool

Use SGD for D and ADAM optimizer for G

Use Label smoothing and occasionally flip labels when training D

Use Dropout in G in both train and test

DCGAN in PyTorch

https://github.com/pytorch/examples/blob/master/dcgan/main.py