InData Labs

Generative Modeling with Convolutional Neural Networks

Denis Dus

Data Scientist at InData Labs

What we will discuss

1. Discriminative vs Generative modeling
2. Convolutional Neural Networks
3. How to train a neural network to fool another network
4. How to train a neural network to produce photorealistic images (GANs here)
5. What difficulties can we face and how to avoid them
6. How could GAN framework be used in real problems

Probabilistic Data Modeling

Discriminative model

Goal - to recover **conditional** distribution $\mathbf{P(y|x; \theta)}$

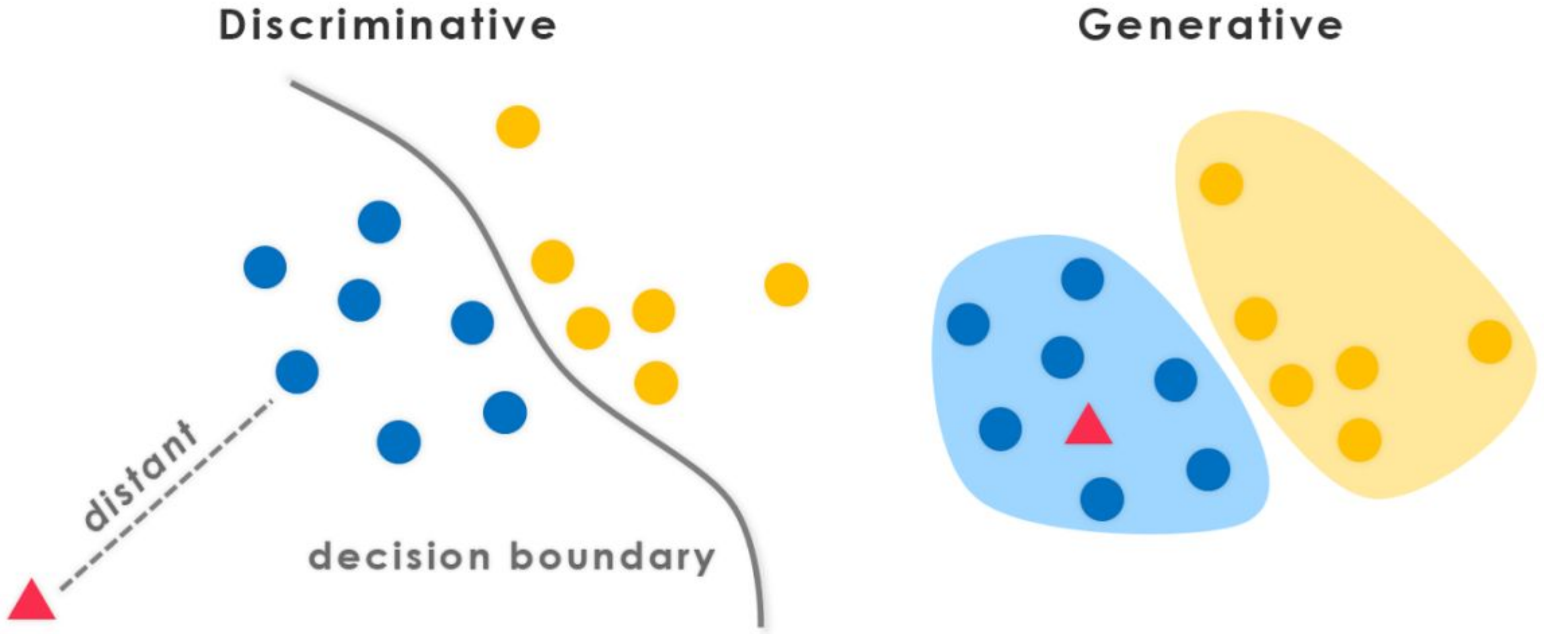
Decision boundary = $\{\mathbf{x, P(y=1|x; \theta) = P(y=0|x; \theta)}\}$

Generative model

Goal - to recover the **joint** distribution $\mathbf{P(x, y; \theta)}$

$P(y|x) = P(x, y; \theta) / P(x) = P(y)P(x|y; \theta) / P(x)$

Discriminative vs Generative

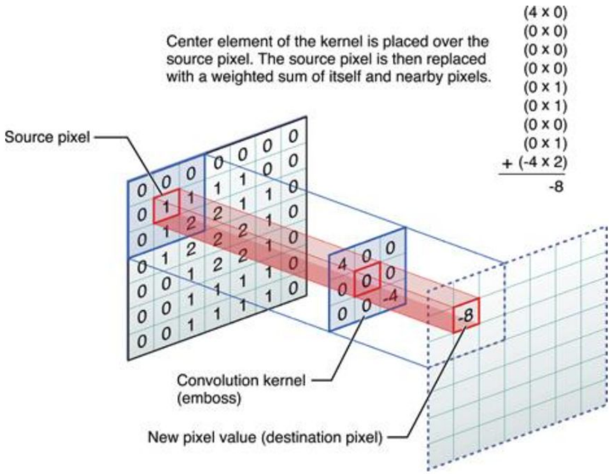


CNNs are strong discriminators

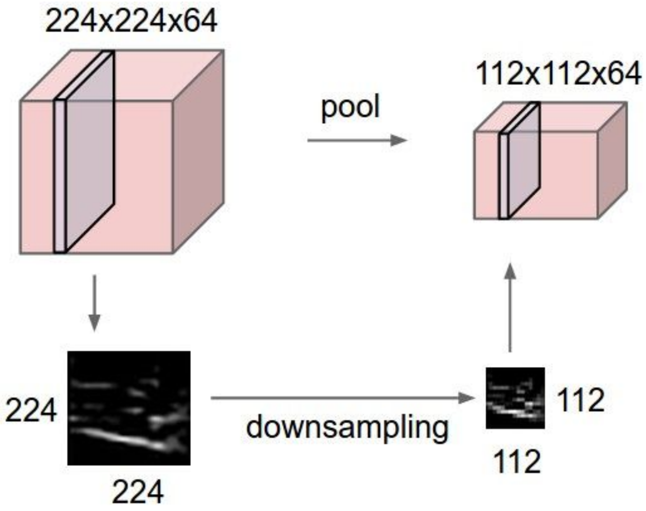
CNNs: Architectures

2D convolution:

$$(g * f)(x, y) = \sum_{(a,b) \in A} g(a, b) f(x - a, y - b)$$



Spatial pooling:



CNNs: Architectures

1. **Typical “formula” of Convolutional Neural Network**

INPUT→[[CONV→(BN)→ReLU] * N→(POOL)] * M→[FC→(BN)→ReLU] * K→FC

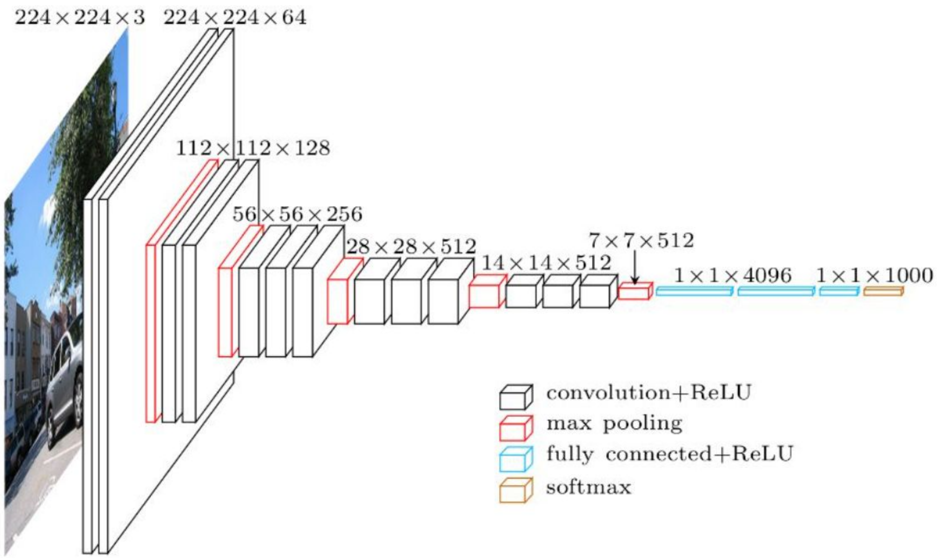
2. **Variations**

ResNets / Inceptions (v1-v4) / ...

3. **Tendencies**

A. Fully Convolutional Networks
B. Reduction of the number of parameters and computational complexity

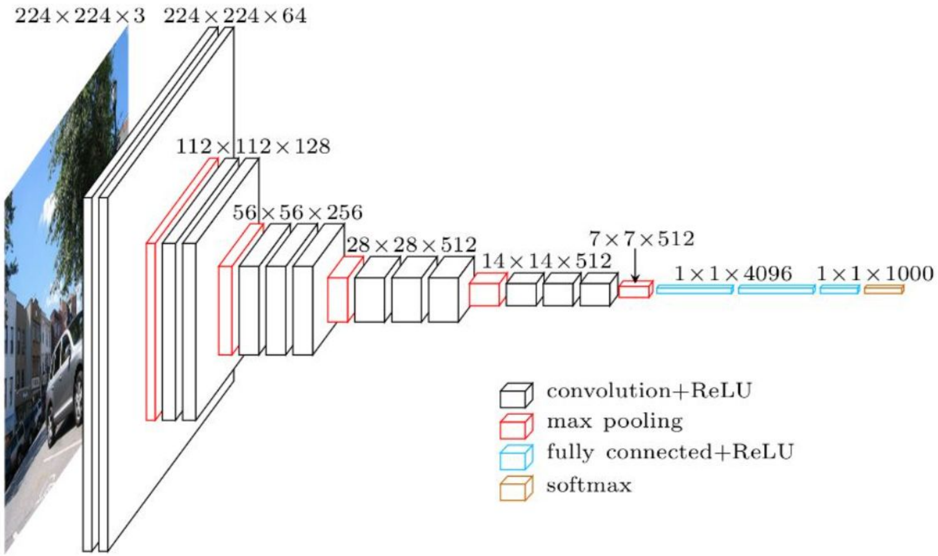
CNNs: VGG-16



Simonyan, Karen, and Zisserman. "Very deep convolutional networks for large-scale image recognition." (2014)

InData Labs

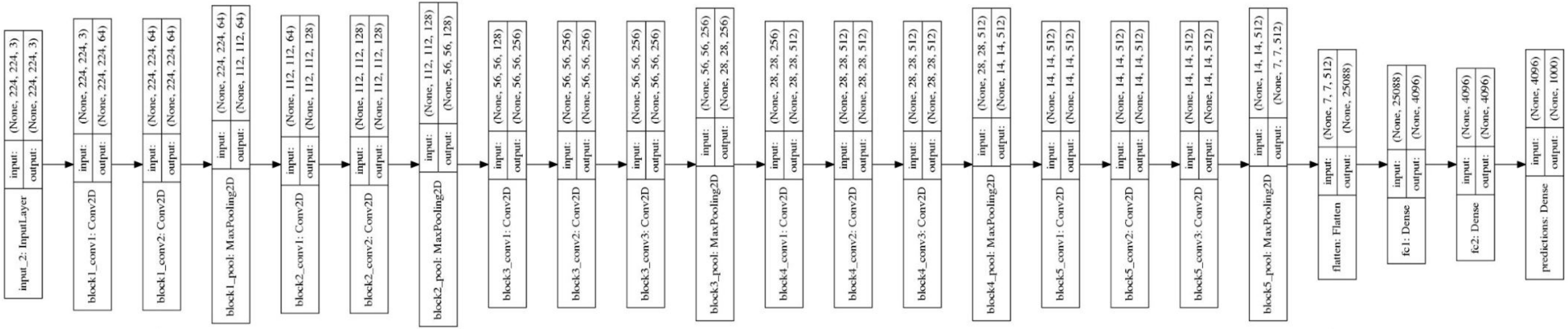
CNNs: VGG-16



State of the Art for 2014!

Simonyan, Karen, and Zisserman. "Very deep convolutional networks for large-scale image recognition." (2014)

CNNs: VGG-16 Layerwise



Block 1:

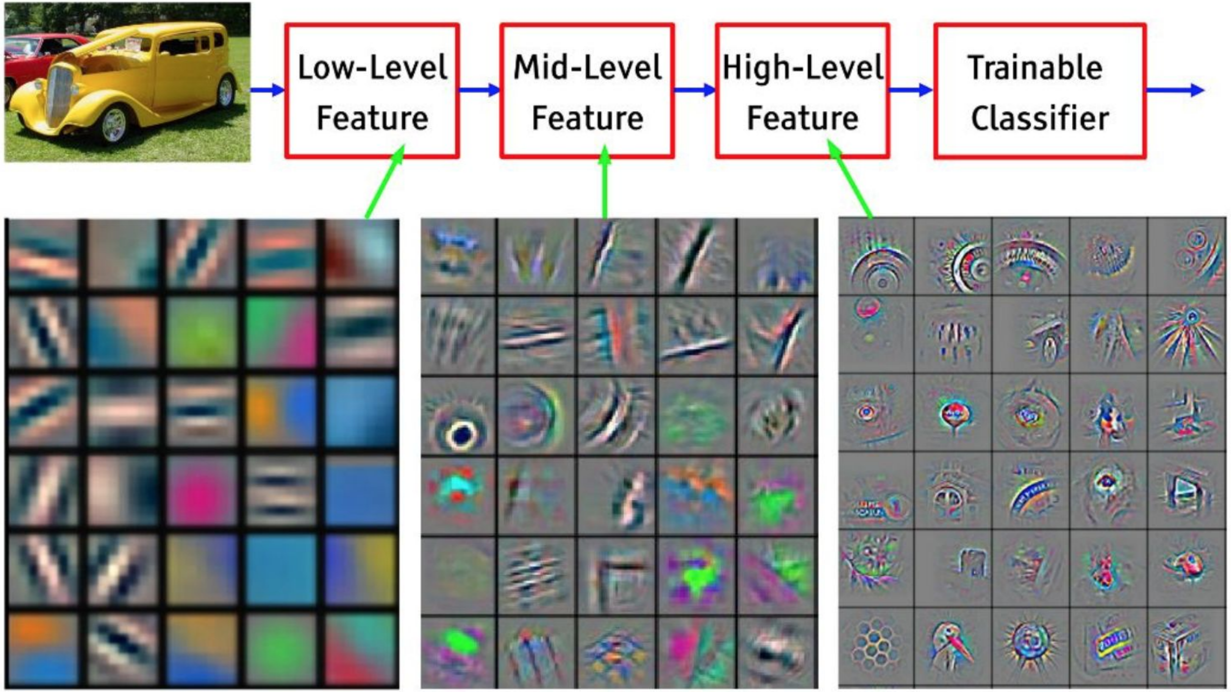
- Input shape: **224x224x3**
- 2 conv layer
- 64 filters of size 3x3
- 2x2 max pooling (stride 2)

Block 5:

- 3 conv layer
- 512 filters of size 3x3
- 2x2 max pooling (stride 2)
- Output shape: **7x7x512**

InData Labs

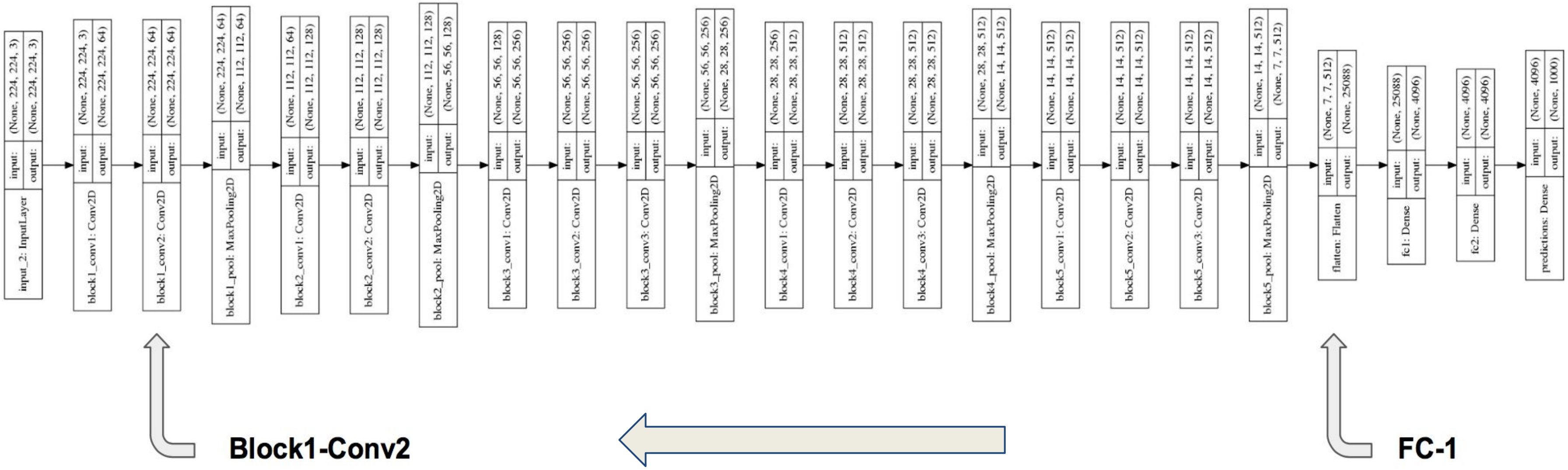
CNNs: Hierarchical Representation



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

InData Labs

CNNs: VGG-16 Layerwise

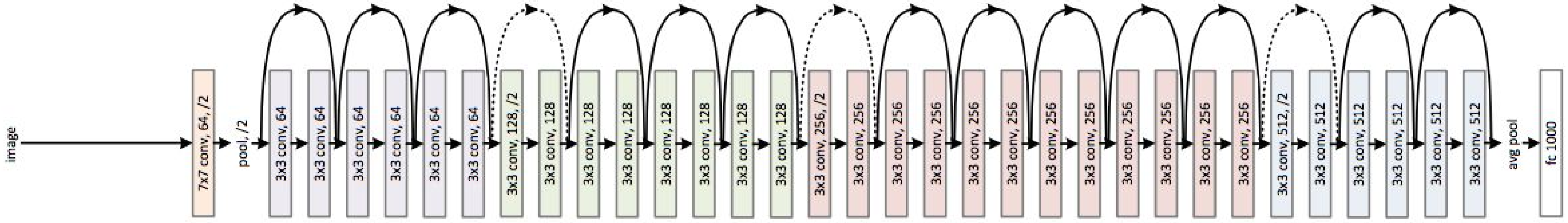


224x224x64 ≈ **3.2 mln** activations

Gradient flow

7x7x512x4096 ≈ **103 mln** parameters

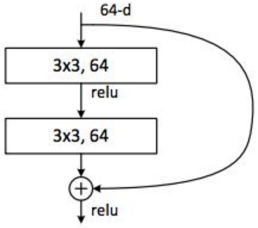
CNNs: ResNets

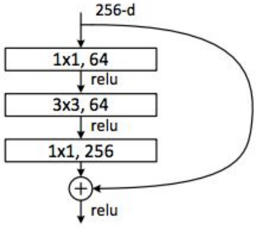


Input block:

- Inputs shape: **224x224x3**
- 64 filters of size 7x7 (stride 2)
- 3x3 max pooling (stride 2)
- Output shape: **56x56x64** (~200k activations)

Residual block(s):

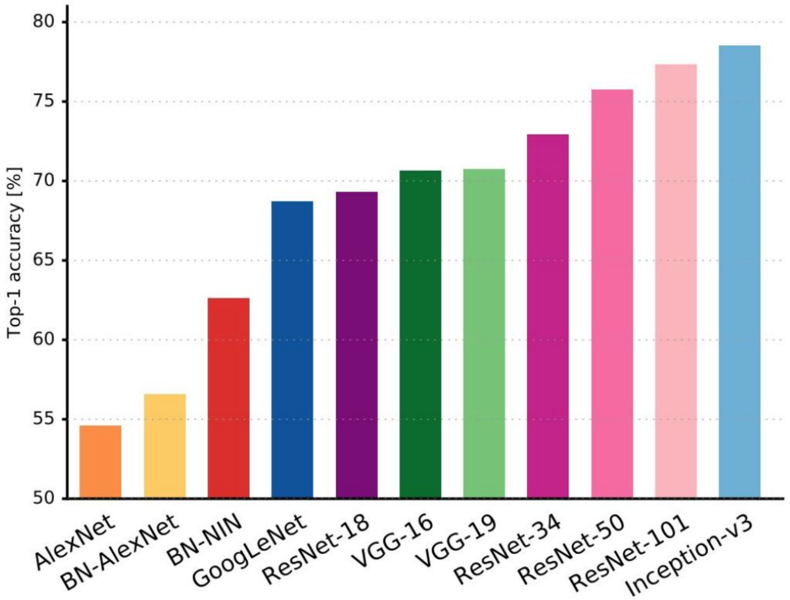


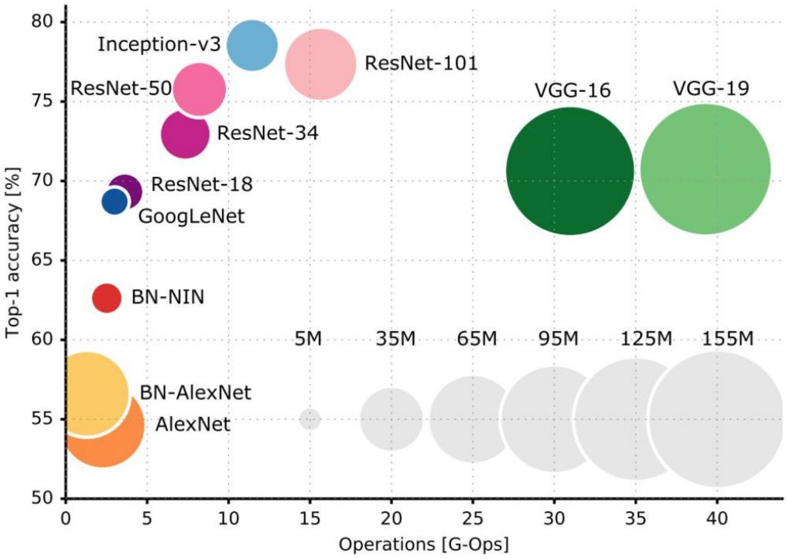


Pre-output block:

- Input shape: **7x7x512**
- Global Average Pooling
- Output shape: **1x1x512**

CNNs: ImageNet





https://m2dsupsdlclass.github.io

Could CNNs act as strong generators?

Generative CNNs: Idea

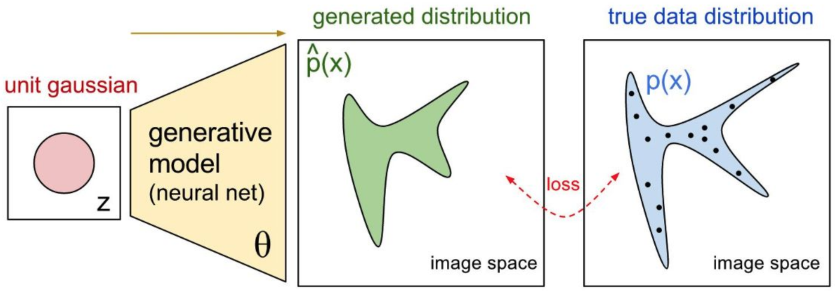
1. The joint distribution of image pixels is very complex.

2. It is impossible to approximate it with classical methods.

Generative CNNs: Idea

1. The joint distribution of image pixels is very complex.
2. It is impossible to approximate it with classical methods.
3. We'll look for a parametric transformation of the following type:

$$z \sim N_n(0, 1) \in R^n$$
$$G(z|\theta) : R^n \to I$$
$$G(z|\theta) \approx P(I)$$

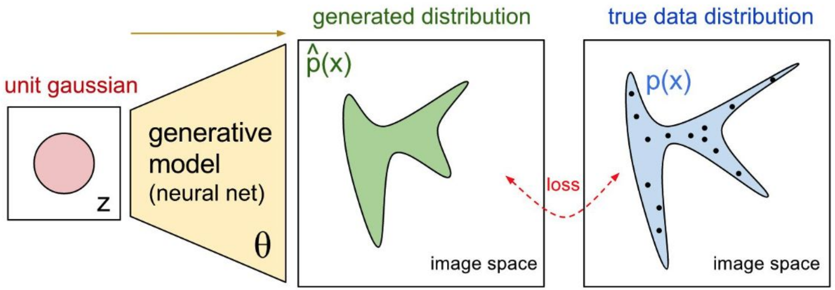


Generative CNNs: Idea

1. The joint distribution of image pixels is very complex.
2. It is impossible to approximate it with classical methods.
3. We'll look for a parametric transformation of the following type:

$$z \sim N_n(0,1) \in R^n$$
$$G(z|\theta) : R^n \to I$$
$$G(z|\theta) \approx P(I)$$

Something similar to "reparametrization trick"



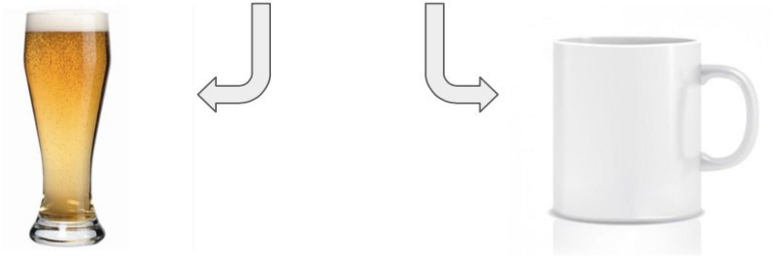
Generative CNNs: Idea

- We have strong discriminative networks
- Weights of the networks trained on ImageNet dataset are publicly available

  https://github.com/fchollet/deep-learning-models

- If you have such network, for every image x you can get

$$D(x) : I \to (p_0, \dots, p_{441}, \dots, p_{968}, \dots, p_{999})$$



InData Labs

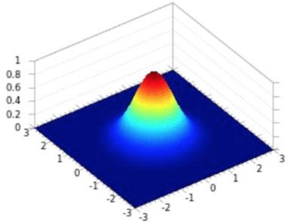
Generative CNNs: Idea

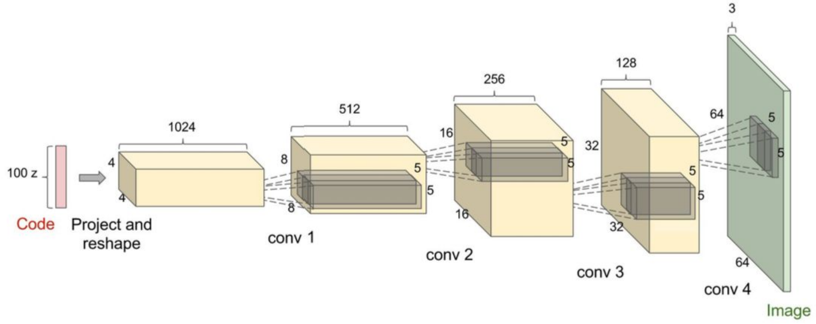
What should you do to generate images for the class “Cup”?

Let’s look for G(z|θ) so that D(G(z|θ)) is close to predefined distribution.

$$D(x) : I \to (p_0, \dots, p_{441}, \dots, p_{968}, \dots, p_{999})$$

$$z \sim N_n(0, 1) \in R^n$$
$$G(z|\theta) : R^n \to I$$
$$D(G(z|\theta)) \approx (0.0, \dots, 0.0, \dots, 1.0, \dots, 0.0)$$
$$Loss(z, \theta) = -\sum_{j=0}^{999} p_j log D(G(z|\theta))_j = -log D(G(z|\theta))_{968} \to min_\theta$$

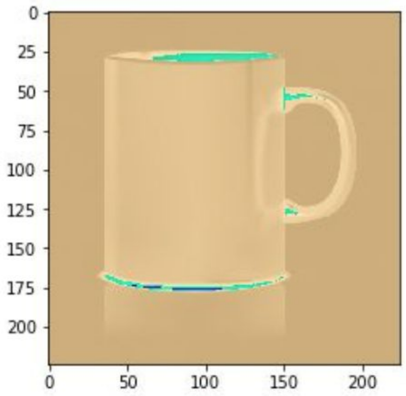




Generative CNNs: Practice - VGG-16

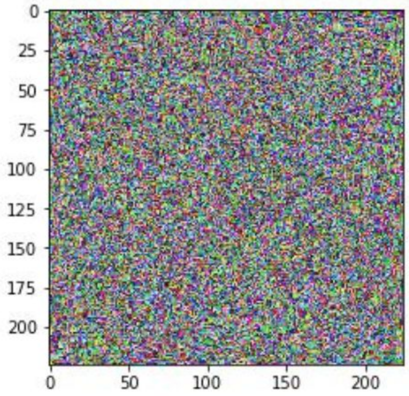
Before training the transformation G(z|θ)

D(x)




```
n03063599 / coffee_mug -- 0.4583
n07930864 / cup -- 0.3692
n03950228 / pitcher -- 0.0405
n04560804 / water_jug -- 0.0206
n03063689 / coffeepot -- 0.0176
```

D(G(z))



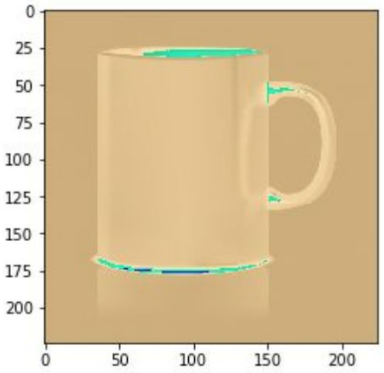

```
n03729826 / matchstick -- 0.0810
n04286575 / spotlight -- 0.0397
n03666591 / lighter -- 0.0334
n01930112 / nematode -- 0.0314
n03196217 / digital_clock -- 0.0303
```

InData Labs

Generative CNNs: Practice - VGG-16

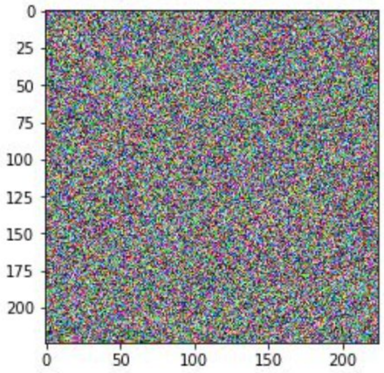
After training the transformation G(z|θ)

D(x)



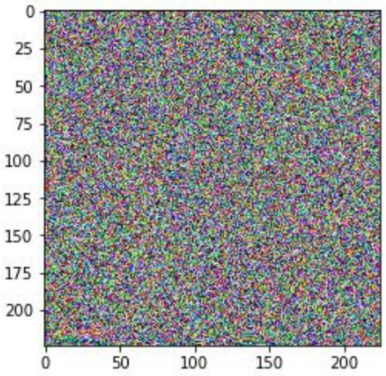

```
n03063599 / coffee_mug -- 0.4583
n07930864 / cup -- 0.3692
n03950228 / pitcher -- 0.0405
n04560804 / water_jug -- 0.0206
n03063689 / coffeepot -- 0.0176
```

D(G(z))




```
n07930864 / cup -- 0.8177
n03063599 / coffee_mug -- 0.1737
n03666591 / lighter -- 0.0029
n03443371 / goblet -- 0.0015
n03729826 / matchstick -- 0.0011
```

D(G(z))

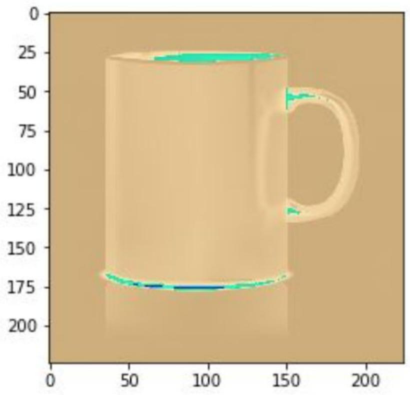



```
n07930864 / cup -- 0.8320
n03063599 / coffee_mug -- 0.1523
n03443371 / goblet -- 0.0061
n03062245 / cocktail_shaker -- 0.0020
n03476991 / hair_spray -- 0.0016
```

Generative CNNs: Practice - ResNet-50

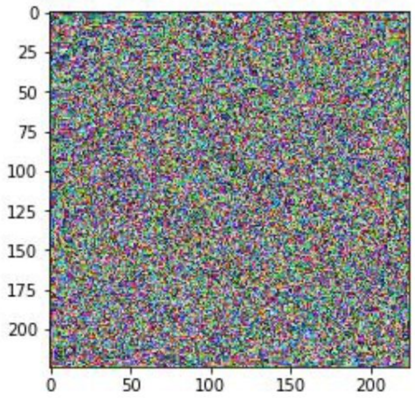
Before training the transformation G(z|θ)

D(x)




```
n07930864 / cup -- 0.5886
n03063599 / coffee_mug -- 0.3752
n04579145 / whiskey_jug -- 0.0066
n03063689 / coffeepot -- 0.0064
n07920052 / espresso -- 0.0049
```

D(G(z))

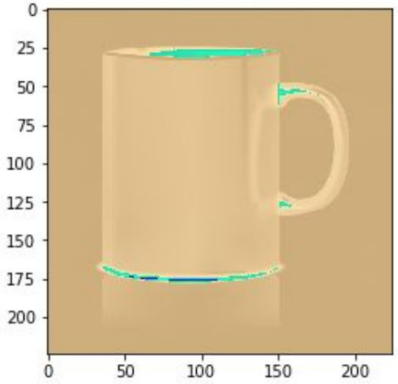



```
n03544143 / hourglass -- 0.0739
n02948072 / candle -- 0.0690
n03666591 / lighter -- 0.0480
n03729826 / matchstick -- 0.0255
n03804744 / nail -- 0.0225
```

Generative CNNs: Practice - ResNet-50

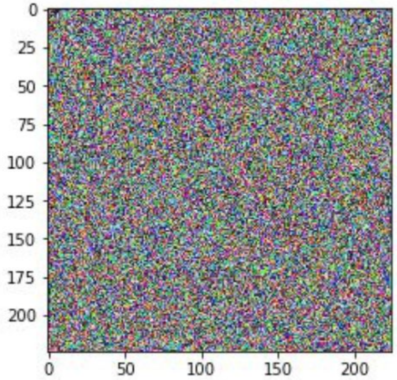
After training the transformation G(z|θ)

D(x)



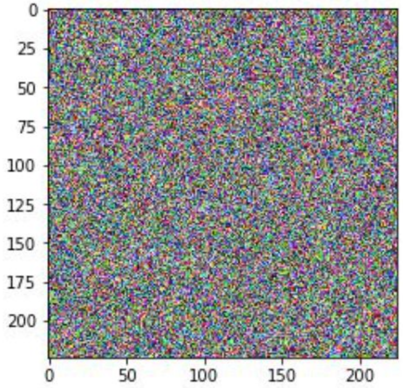
```
n07930864 / cup -- 0.5886
n03063599 / coffee_mug -- 0.3752
n04579145 / whiskey_jug -- 0.0066
n03063689 / coffeepot -- 0.0064
n07920052 / espresso -- 0.0049
```

D(G(z))



```
n07930864 / cup -- 1.0000
n03976467 / Polaroid_camera -- 0.0000
n04153751 / screw -- 0.0000
n04004767 / printer -- 0.0000
n04041544 / radio -- 0.0000
```

D(G(z))



```
n07930864 / cup -- 1.0000
n03976467 / Polaroid_camera -- 0.0000
n04153751 / screw -- 0.0000
n04004767 / printer -- 0.0000
n04041544 / radio -- 0.0000
```

Generative CNNs: Too Naive

- How many images of shape 224x224 pixel exists?

$$256^{3 \times 224 \times 224} = 256^{150528}$$

- Natural images are only a small part of this number
- **We did not require "naturalness" from G(z|θ)**!
- **Only a predetermined classification was required!**

InData Labs

CNNs: Hack’em all!

Let’s consider the following optimization problem

$$x \in I \subset R^{224x224x3}$$

$$D(x) : I \rightarrow (p_0, \dots, p_{441}, \dots, p_{968}, \dots, p_{999}) \approx (0.0, \dots, 0.0, \dots, 1.0, \dots, 0.0)$$

$$D(G(x|\theta)) = (p_0, \dots, p_{441}, \dots, p_{968}, \dots, p_{999}) \approx (0.0, \dots, 1.0, \dots, 0.0, \dots, 0.0)$$

$$G(x|\theta) = x + \theta, \theta \in R^{224x224x3}$$

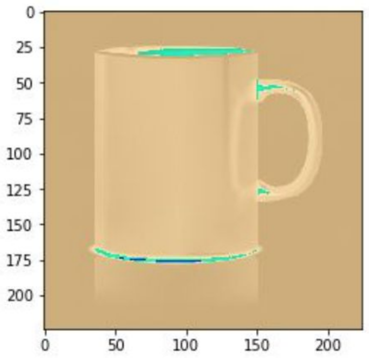
$$Loss(\theta) = -logD(G(x|\theta))_{441} + \|\theta\|_{L_1} \rightarrow min_\theta$$

```python
class TrainableNoiseLayer(Layer):

    def __init__(self, W_regularizer=None, **kwargs):
        self.W_regularizer = W_regularizer
        super(TrainableNoiseLayer, self).__init__(**kwargs)

    def build(self, input_shape):
        self.W = self.add_weight(name='kernel',
                                 shape=input_shape[1:],
                                 initializer='uniform',
                                 regularizer=self.W_regularizer,
                                 trainable=True)
        super(TrainableNoiseLayer, self).build(input_shape)

    def call(self, x):
        return x + self.W

    def compute_output_shape(self, input_shape):
        return input_shape
```

```python
def get_transformation_network(reg):
    image_input = Input(shape=(224, 224, 3))

    image_output = TrainableNoiseLayer(W_regularizer=l1(reg), name='noise_layer')(image_input)
    model = Model(inputs=image_input, outputs=image_output)

    return model
```

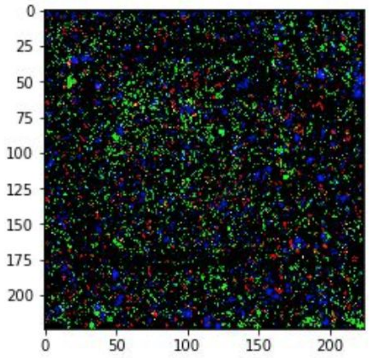
```python
def get_transformation_model():
    input_data = Input((224, 224, 3))
    t_output = T(input_data)
    d_output = D(t_output)

    model = Model(inputs=input_data, outputs=d_output)

    return model
```

CNNs: Hack'em all - ResNet-50



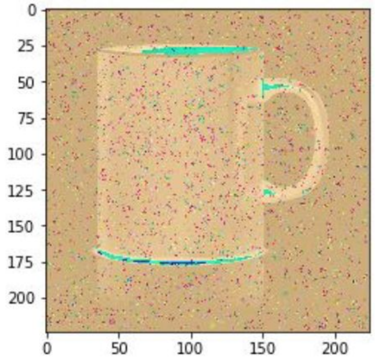
```
968
n07930864 / cup -- 0.5886
n03063599 / coffee_mug -- 0.3752
n04579145 / whiskey_jug -- 0.0066
n03063689 / coffeepot -- 0.0064
n07920052 / espresso -- 0.0049
```

+



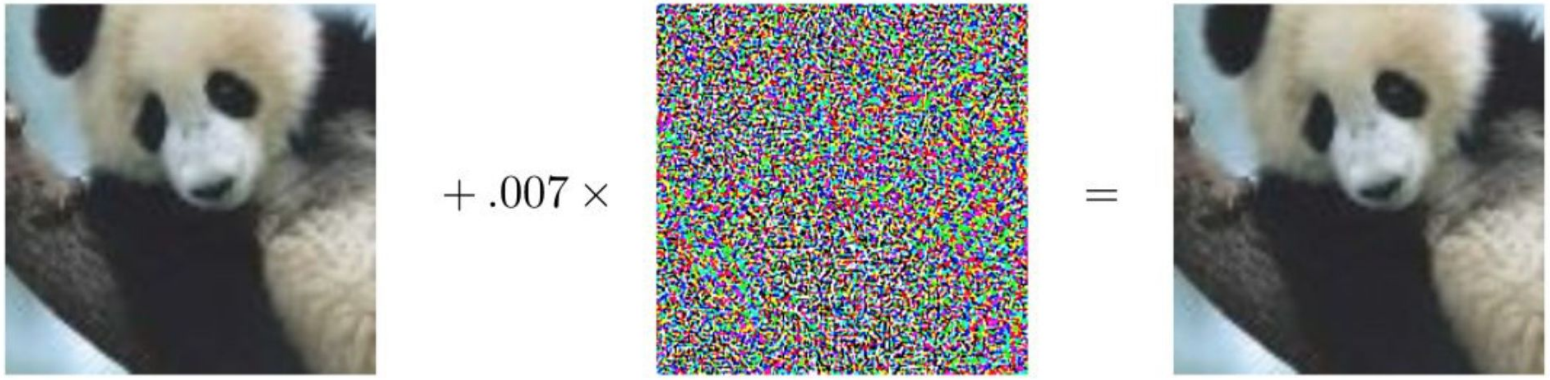
θ

=



```
441
n02823750 / beer_glass -- 0.6678
n03063599 / coffee_mug -- 0.0080
n07930864 / cup -- 0.0069
n03950228 / pitcher -- 0.0063
n02823428 / beer_bottle -- 0.0053
```

CNNs: Hack'em all - GoogLeNet



$\boldsymbol{x}$
"panda"
57.7% confidence

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence

$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

Ian J. Goodfellow, Explaining and Harnessing Adversarial Examples

CNNs: Adversarial Attack Competition

Research Prediction Competition

NIPS 2017: Targeted Adversarial Attack

Develop an adversarial attack that causes image classifiers to predict a specific target class

Overview Data Kernels Discussion Rules

Overview

- Description
- Evaluation
- Cloud Compute Credits
- Dataset
- Getting Started
- Submission Format
- Swag
- Timeline
- Using Kernels

This research competition doesn't follow Kaggle's normal submission process. See the Submission Format tab for more details.

Most existing machine learning classifiers are highly vulnerable to adversarial examples. An adversarial example is a sample of input data which has been modified very slightly in a way that is intended to cause a machine learning classifier to misclassify it. In many cases, these modifications can be so subtle that a human observer does not even notice the modification at all, yet the classifier still makes a mistake.

Adversarial examples pose security concerns because they could be used to perform an attack on machine learning systems, even if the adversary has no access to the underlying model.

To accelerate research on adversarial examples, Google Brain is organizing **Competition on Adversarial Attacks and Defenses** within the NIPS 2017 competition track.

The competition on Adversarial Attacks and Defenses consist of three sub-competitions:

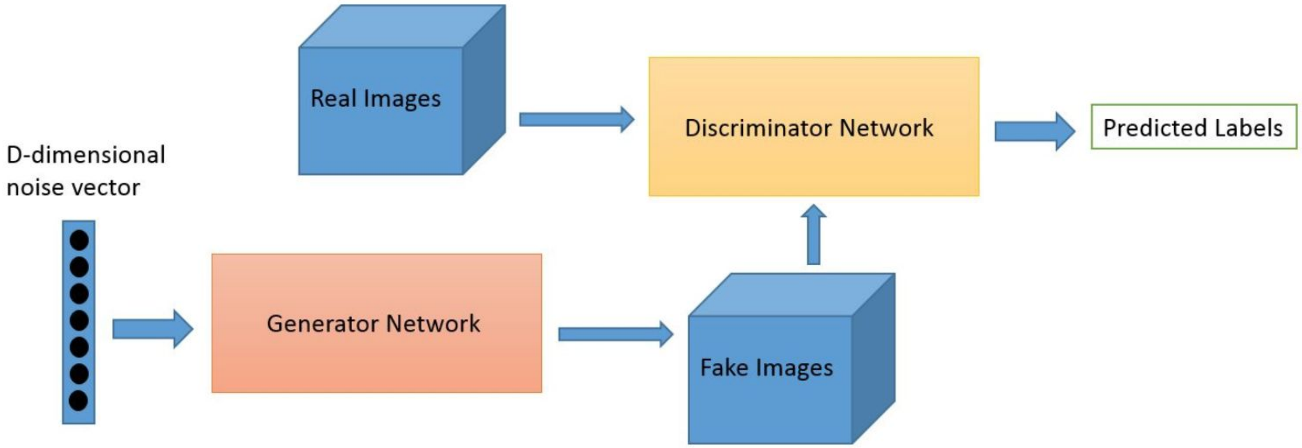
- Non-targeted Adversarial Attack. The goal of the non-targeted attack is to slightly modify source image in a way that image will be classified incorrectly by generally unknown machine learning classifier.
- Targeted Adversarial Attack. The goal of the targeted attack is to slightly modify source image in a way that image will be classified as specified target class by generally unknown machine learning classifier.
- Defense Against Adversarial Attack. The goal of the defense is to build machine learning classifier which is robust to adversarial example, i.e. can classify adversarial images correctly.

Generative Adversarial Networks

The statement of the problem lacks the requirement of image "naturalness".

Let's add it:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$



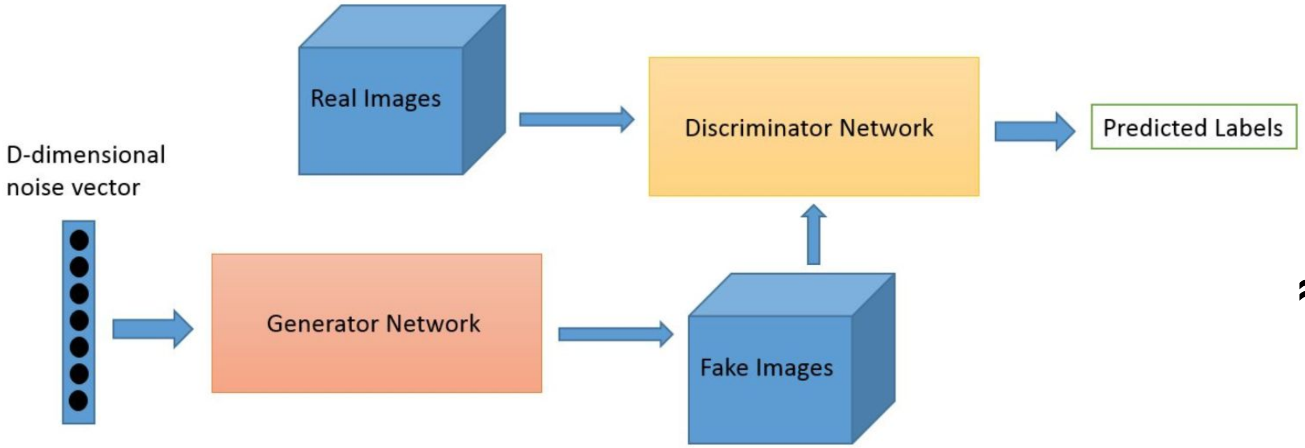
Ian J. Goodfellow, Generative Adversarial Networks

Generative Adversarial Networks

The statement of the problem lacks the requirement of image "naturalness".

Let's add it:

$$\min_G \max_D V(D,G) = \mathbb{E}_{\boldsymbol{x}\sim p_{\text{data}}(\boldsymbol{x})}[\log D(\boldsymbol{x})] + \mathbb{E}_{\boldsymbol{z}\sim p_{\boldsymbol{z}}(\boldsymbol{z})}[\log(1 - D(G(\boldsymbol{z})))]$$



$\approx$ **Jensen-Shannon Divergence Minimization**

Ian J. Goodfellow, Generative Adversarial Networks

InData Labs

Generative Adversarial Networks

Algorithm 1 Minibatch stochastic gradient descent training of generative adversarial nets. The number of steps to apply to the discriminator, k, is a hyperparameter. We used $k = 1$, the least expensive option, in our experiments.

for number of training iterations **do**

  for k steps **do**

   • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.

   • Sample minibatch of m examples $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ from data generating distribution $p_{\text{data}}(\boldsymbol{x})$.

   • Update the discriminator by ascending its stochastic gradient:

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D\left(\boldsymbol{x}^{(i)}\right) + \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right) \right].$$

  end for

  • Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.

  • Update the generator by descending its stochastic gradient:

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} \log\left(1 - D\left(G\left(\boldsymbol{z}^{(i)}\right)\right)\right).$$

end for

The gradient-based updates can use any standard gradient-based learning rule. We used momentum in our experiments.

InData Labs

GAN: Discriminator

```python
class DownConvBlock(nn.Module):

    def __init__(self, in_size, out_size,
                 kernel_size=3, activation=F.elu, p=0.2):
        super(DownConvBlock, self).__init__()

        self.p = p
        self.in_size = in_size
        self.out_size = out_size
        self.kernel_size = kernel_size
        self.padding_size = (kernel_size - 1) // 2

        self.conv = nn.Conv2d(
            in_size, out_size, stride=2,
            kernel_size=self.kernel_size,
            padding=self.padding_size
        )

        if self.p is not None:
            self.dropout = nn.Dropout2d(self.p)

        self.activation = activation

        weights_init(self.conv)

    def forward(self, x):
        out = self.activation(self.conv(x))

        if self.p is not None:
            out = self.dropout(out)

        return out
```

```python
class Discriminator(nn.Module):

    def __init__(self, p=0.2):
        super(Discriminator, self).__init__()

        self.conv_block1 = DownConvBlock(3, 64, p=p)
        self.conv_block2 = DownConvBlock(64, 128, p=p)
        self.conv_block3 = DownConvBlock(128, 256, p=p)
        self.conv_block4 = DownConvBlock(256, 512, p=p)

        self.linear_1 = nn.Linear(
            in_features=512 * 4 * 4, out_features=1024
        )

        self.linear_2 = nn.Linear(
            in_features=1024, out_features=1
        )

        self.dropout = nn.Dropout(p)

        weights_init(self.linear_1)
        weights_init(self.linear_2)

    def forward(self, x):
        x = self.conv_block1(x)
        x = self.conv_block2(x)
        x = self.conv_block3(x)
        x = self.conv_block4(x)

        x = x.view((-1, 512 * 4 * 4))

        x = F.elu(self.linear_1(x))
        x = self.dropout(x)

        x = F.sigmoid(self.linear_2(x))

        return x
```

InData Labs

GAN: Generator

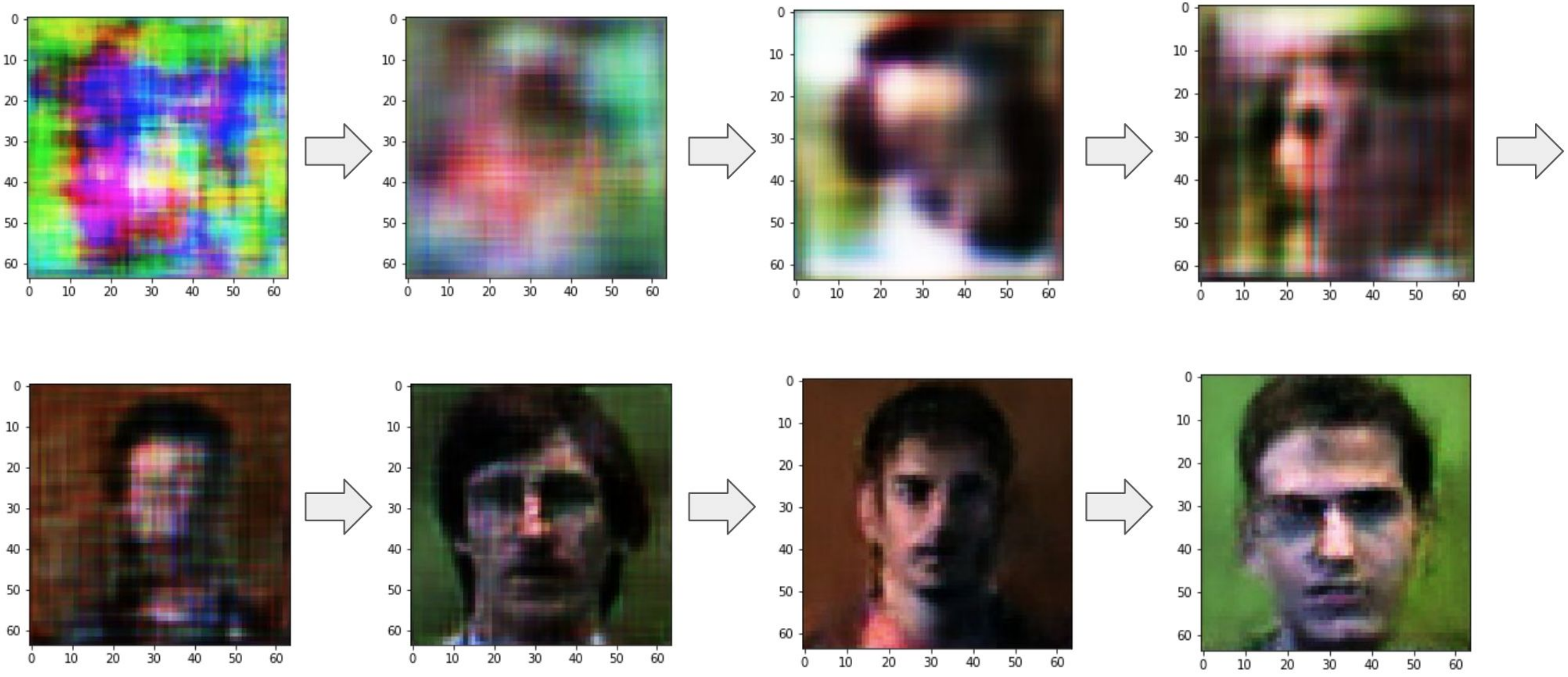
```python
class UpConvBlock(nn.Module):

    def __init__(self, in_size, out_size,
                 kernel_size=3, activation=F.elu, p=0.3):
        super(UpConvBlock, self).__init__()

        self.p = p
        self.in_size = in_size
        self.out_size = out_size
        self.kernel_size = kernel_size
        self.padding_size = (kernel_size - 1) // 2

        self.activation = activation

        self.up = nn.UpsamplingNearest2d(scale_factor=2)

        self.conv = nn.Conv2d(
            self.in_size, self.out_size,
            kernel_size=self.kernel_size,
            padding=self.padding_size
        )

        self.bn = nn.BatchNorm2d(self.out_size)

        if self.p is not None:
            self.dropout = nn.Dropout2d(p)

        weights_init(self.conv)

    def forward(self, x):
        out = self.up(x)
        out = self.bn(self.conv(out))
        out = self.activation(out)

        if self.p is not None:
            out = self.dropout(out)

        return out
```

```python
class Generator(nn.Module):

    def __init__(self, input_size, input_channels,
                 p_linear=0.2, p_conv=None):
        super(Generator, self).__init__()

        self.p_linear = p_linear
        self.p_conv = p_conv
        self.input_size = input_size
        self.input_channels = input_channels

        self.linear = nn.Linear(
            in_features=input_size,
            out_features=input_channels * 4 * 4
        )

        if self.p_linear is not None:
            self.linear_dropout = nn.Dropout(p=p_linear)

        self.up_block1 = UpConvBlock(input_channels, 512, kernel_size=5, p=self.p_conv)
        self.up_block2 = UpConvBlock(512, 256, kernel_size=5, p=self.p_conv)
        self.up_block3 = UpConvBlock(256, 128, kernel_size=5, p=self.p_conv)
        self.up_block4 = UpConvBlock(128, 64, kernel_size=3, p=self.p_conv)

        self.last = nn.Conv2d(64, 3, kernel_size=3, padding=1)

        weights_init(self.linear)
        weights_init(self.last)

    def forward(self, z):
        x = F.elu(self.linear(z))

        if self.p_linear is not None:
            x = self.linear_dropout(x)

        x = x.view((-1, self.input_channels, 4, 4))

        up1 = self.up_block1(x)
        up2 = self.up_block2(up1)
        up3 = self.up_block3(up2)
        up4 = self.up_block4(up3)

        return F.tanh(self.last(up4))
```

GAN Framework: Real Data

InData Labs

GAN Framework: Generator Evolution



InData Labs

GAN: Difficulties

1. Generation of large-sized images (more than 128x128 pixels)
2. The variability of real-world images
3. There is no guarantee that the game converges to an equilibrium state
4. It is difficult to keep D(x) and G(z) “on the same quality level”
5. Initially, D(x) has a very simple problem (in comparison with G(z))
6. Quality assessment of G(z)

InData Labs

GAN: State of the Art

Figure 4: Samples generated during semi-supervised training on CIFAR-10 with feature matching (Section 3.1, *left*) and minibatch discrimination (Section 3.2, *right*).

Ian Goodfellow, Improved Techniques for Training GANs

GAN: State of the Art

Figure 3: Generated bedrooms after five epochs of training. There appears to be evidence of visual under-fitting via repeated noise textures across multiple samples such as the base boards of some of the beds.

Alec Radford & Luke Metz, Unsupervised Representation Learning with DCGANs

GAN: Variability and Size

Figure 6: Samples generated from the ImageNet dataset. *(Left)* Samples generated by a DCGAN. *(Right)* Samples generated using the techniques proposed in this work. The new techniques enable GANs to learn recognizable features of animals, such as fur, eyes, and noses, but these features are not correctly combined to form an animal with realistic anatomical structure.

Ian Goodfellow, Improved Techniques for Training GANs

InDataLabs

GAN: Designing the D(x)

1. Normalize the input images in [-1.0, 1.0]
2. Should have less parameters than G(x)
3. Use Dropout / Spatial Dropout
4. 2x2 MaxPooling → Convolution2D + Stride = 2
5. ReLU → LeakyReLU
6. Adaptive L2 regularization / Label Smoothing / Instance Noise / ...
7. Balancing of the min-max the game

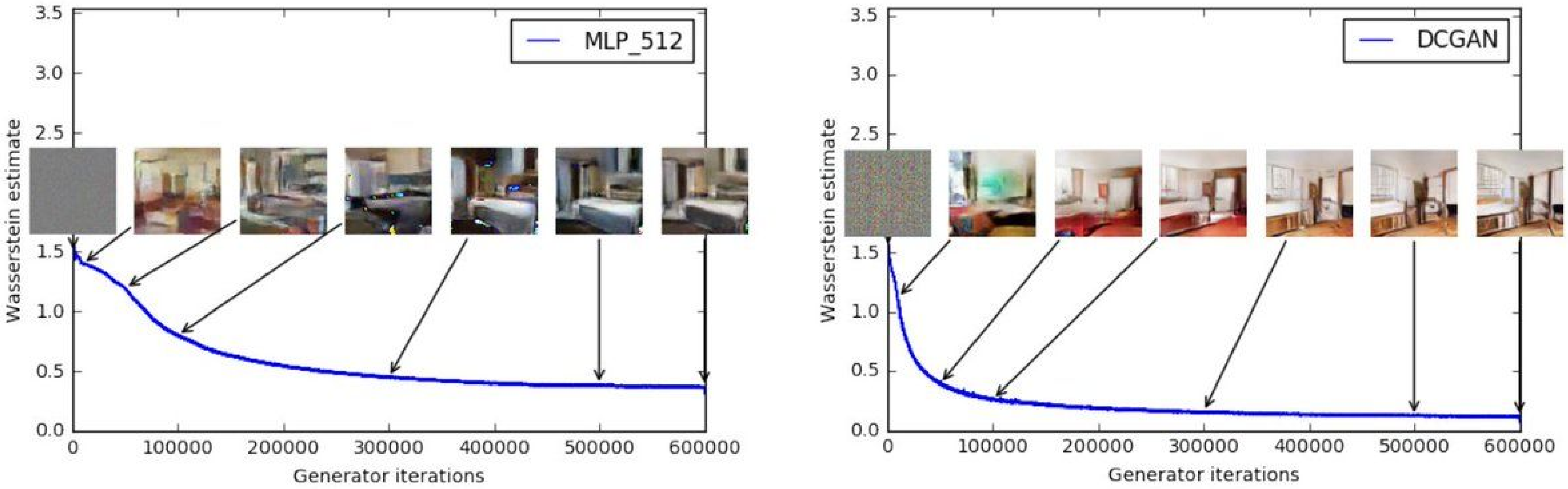
https://github.com/soumith/ganhacks

Ian Goodfellow, Improved Techniques of Training GANs

InData Labs

GAN: Designing the D(x)

In 2017: just use Wasserstein GAN



GAN: Designing the G(x)

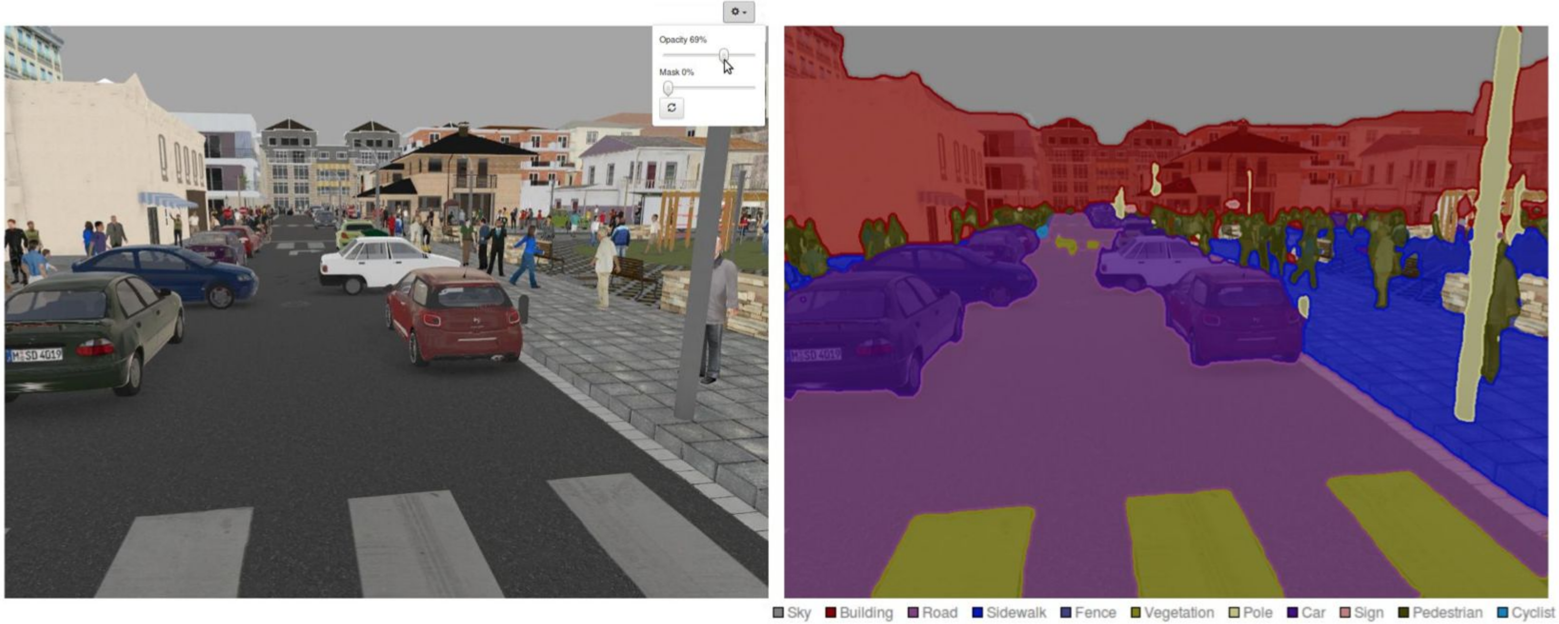
1. Tanh on the last layer
2. min log(1-D(G(z)) ➔ max log(D(G(z))
3. Should have more parameters than D(x)
4. Use Dropout / Spatial Dropout
5. UpSampling2D / Deconvolution2D
6. ReLU ➔ LeakyReLU
7. Batch Normalization

https://github.com/soumith/ganhacks

Ian Goodfellow, Improved Techniques of Training GANs

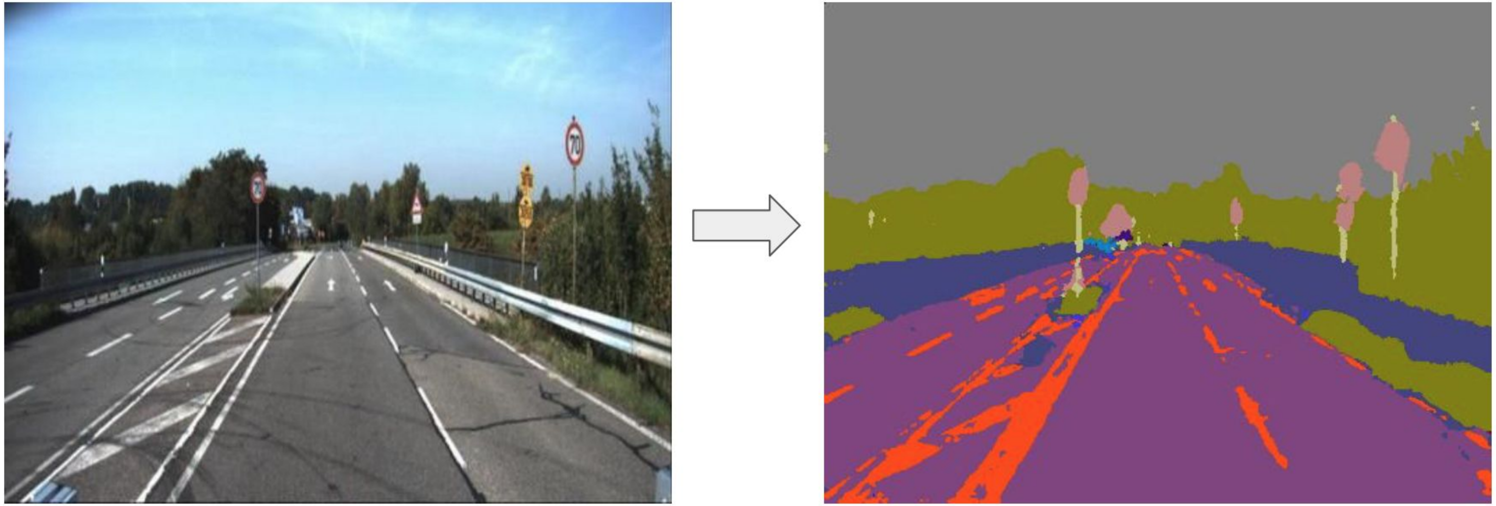
GAN: Image Segmentation

- One of the main Computer Vision tasks
- Objective function based on per-pixel loss functions



GAN: Image Segmentation

- Objective function based on per-pixel loss functions ...
- ... which do not reflect many aspects of segmentation quality

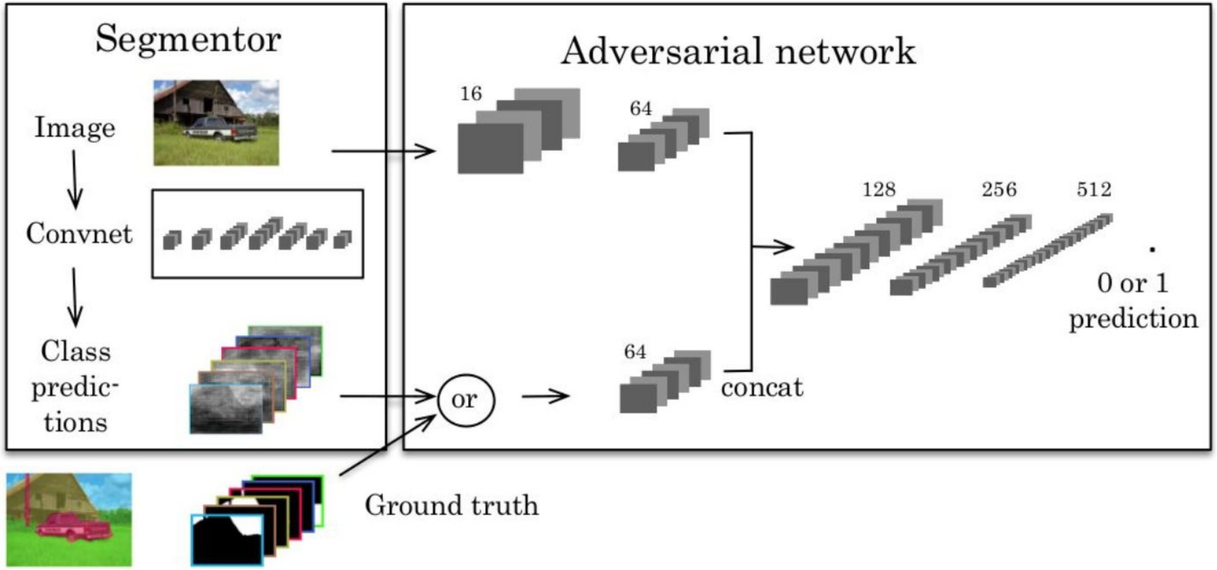


https://github.com/alexgkendall/SegNet-Tutorial

InData Labs

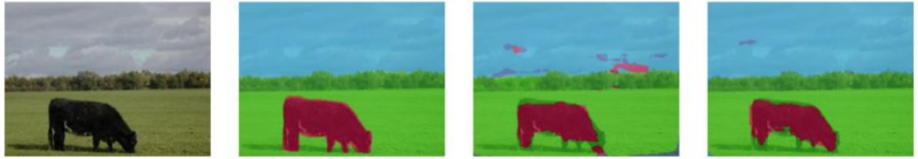
GAN: Image Segmentation

$$\ell(\boldsymbol{\theta}_s, \boldsymbol{\theta}_a) = \sum_{n=1}^{N} \ell_{\text{mce}}(s(\boldsymbol{x}_n), \boldsymbol{y}_n) - \lambda \Big[\ell_{\text{bce}}(a(\boldsymbol{x}_n, \boldsymbol{y}_n), 1) + \ell_{\text{bce}}(a(\boldsymbol{x}_n, s(\boldsymbol{x}_n)), 0) \Big]$$



Pauline Luc, Semantic Segmentation using Adversarial Networks

GAN: Image Segmentation



input | ground truth | no adversarial | with adversarial

Figure 3: Segmentations on Stanford Background. Class probabilities without (first row) and with (second row) adversarial training. In the last row the class labels are superimposed on the image.

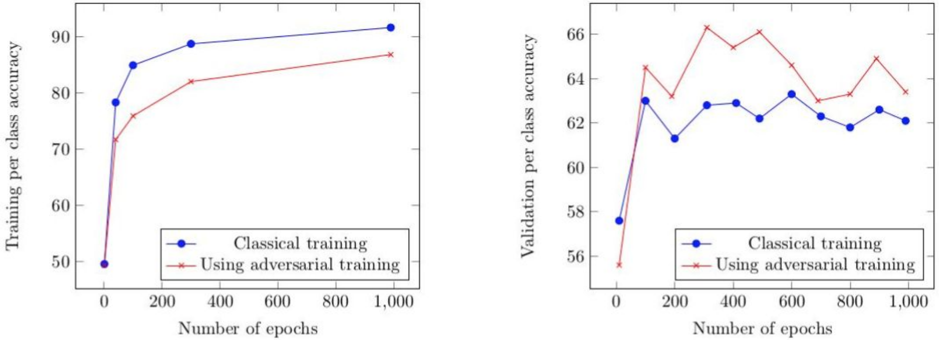


Figure 4: Per-class accuracy across training ephochs on the Stanford Background dataset on train data (left) and validation data (right), with and without adversarial training.

Pauline Luc, Semantic Segmentation using Adversarial Networks

Many other interesting things

1. **Types of GAN:** CGAN, WGAN, Stacked GAN, Cycle GAN, ...
2. **Alternative models:** VAE, Pix2Pix, ...
3. **Other areas of application:** generation of text/audio/video
4. **Other practical tasks:** Image Super-Resolution, ...

InData Labs

Summary

1. CNNs perfectly cope with the CV tasks
2. But there is a serious problem with “Adversarial Examples”
3. GAN - is a very interesting idea at the intersection of areas of mathematics
4. GAN is not only fun, but also useful
5. Future improvements in this area are expected

InData Labs

Thank you!

Denis Dus
Data Scientist at InData Labs

d_dus@indatalabs.com

www.indatalabs.com   @indatalabs   @indatalabs