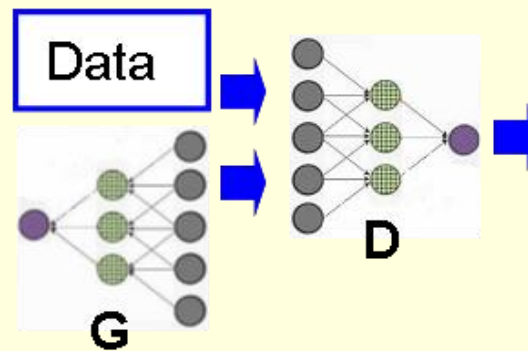


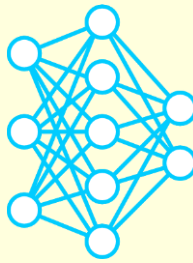
# Deep Learning by Example on Biowulf

## Class #4: Generative Adversarial Networks and their application to bioimage synthesis

Gennady Denisov, PhD



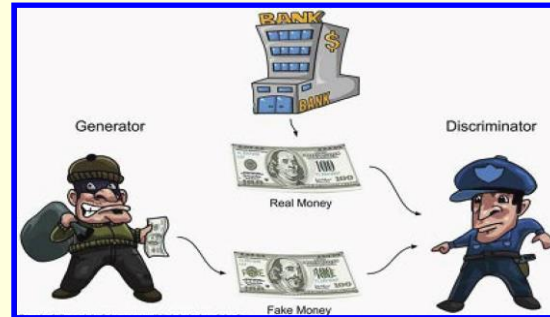
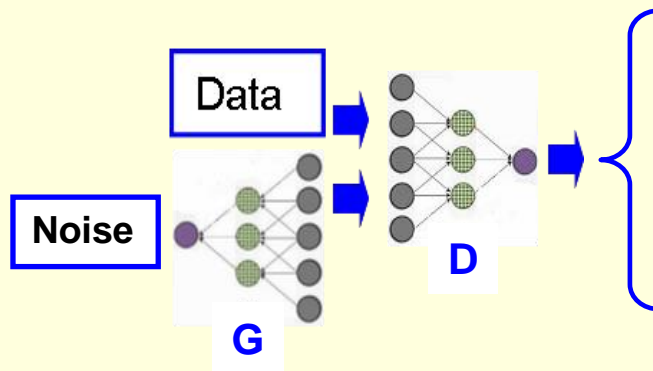
# Intro and goals



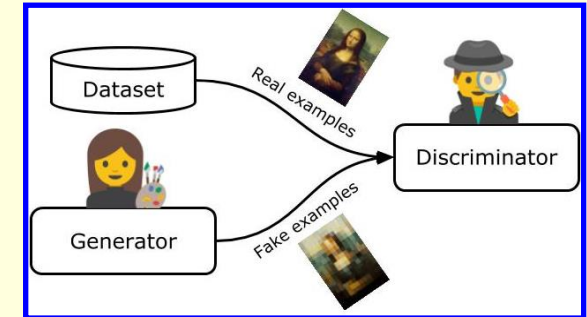
*I. Goodfellow et al., Generative Adversarial Nets. NIPS Proc. 2014*

## What is a GAN?

- A composite network comprising 2 subnetworks: **G**enerator and **D**iscriminator
- The **G** produces fake data from scratch/noise; learns to **trick** the **D**
- The **D** compares fake data against the true data; learns to **expose** the **G**



Counterfeiter vs police analogy



Forger vs art dealer/critique analogy

## Features:

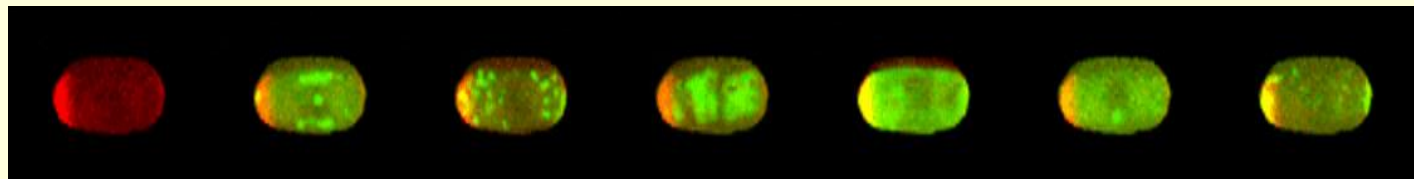
**Generative model:** the goal is to generate new, **synthetic** instances of data that can pass for real data  
**G** and **D** are trained by **pitting** one against the other – thus the **adversarial**, i.e. antagonistic, or confrontational  
“...The most interesting idea in ML in the last 10 years.” (Yann LeCun)

## Examples:

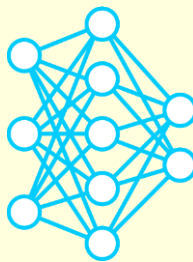
Generating face images



**BioGANs: GANs for biological image synthesis**

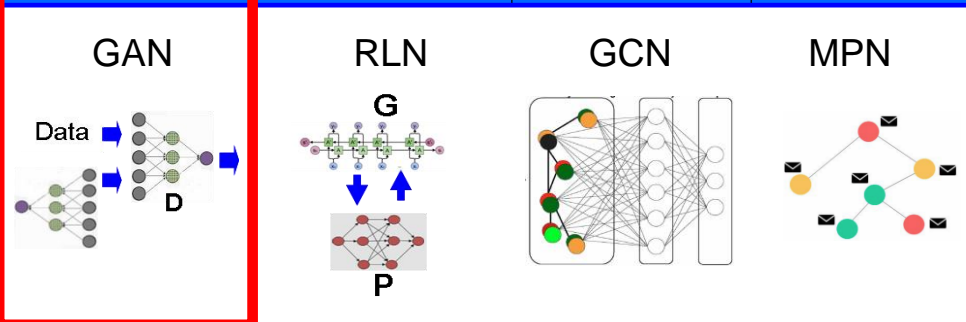


# Examples overview



Class #	1	2	3	4	5	6	7
Bio app	Bioimage segmentation / fly brain connectome	Genomics / prediction of function of non-coding DNA	Genomics / reduction of dimensionality of cancer transcriptome	Bioimage synthesis / developmental biology	Drug molecule design	Genomics / classification of cancer types	Drug molecule property prediction
Neural network type	Convolutional	Recurrent or 1D-Convolutional	Autoencoder	Generative Adversarial	Reinforcement Learning	Graph Convolutional	Message Passing
ML type	Supervised	Supervised	Unsupervised	Unsupervised	Reinforcement	Supervised	Supervised

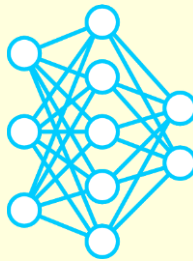
- **unsupervised ML**
- **generative model**, functionally similar to VAE
- **composite network** comprising two subnetworks
- the subnetworks are **trained interactively**, by playing a **minimax game**
- **GAN flavors**: GAN, DCGAN, WGAN, WGAN-GP



# Deep Convolutional GAN (DCGAN): a simple example

tensors, units, layers, parameters, hyperparameters, convolution

A.Radford et al, arXiv:1511.06434v2 (2016)



## RNN/1D CNN prototype example from class #2:

**Input:** a set of **training sequences** of 0's and 1's with **binary labels** assigned to each sequence depending on whether or not a certain (unknown) **motif** is present in the sequence

**Example:** 01011100101

**Task:** predict the label, or the occurrence of the **unknown** motif, in new, previously unseen sequences.

## DCGAN prototype example:

**Input:** a training set of only “good” sequences of 0's and 1's, i.e. **all** of them contain a certain motif

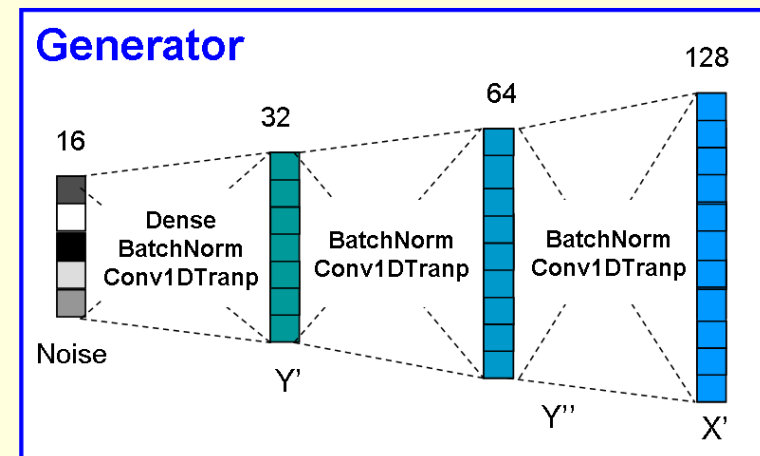
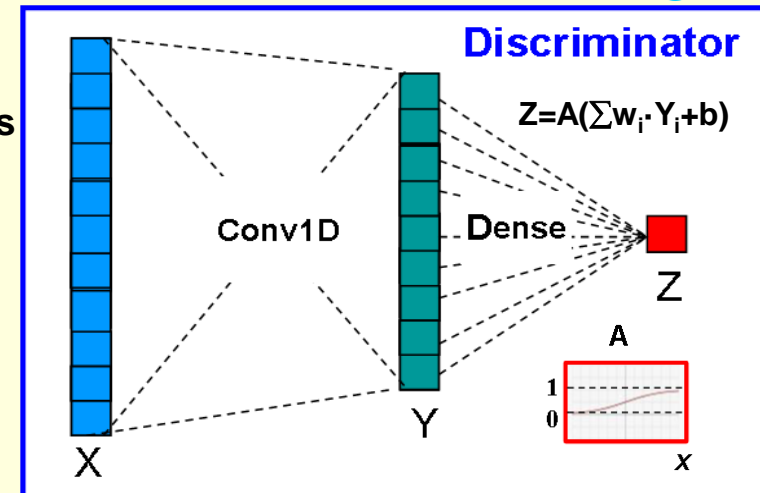
**Example:** 0101100110011001100111

**Task:** learn what makes all of the training sequences “good” and then generate new “good” sequences from scratch.

**Challenge:** only positive examples, no labels

## Architecture guidelines for stable DCGANs:

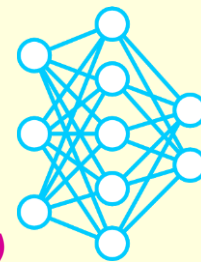
- Use **convolutions** (D) and **transposed convolutions** (G) instead of MaxPooling / Upsampling layers
- Use **BatchNormalization** in both the G and the D.
- Avoid Dense/Fully Connected hidden layers
- ...



# The Transposed Convolution (a.k.a. deconvolution, or fractional-strided convolution)

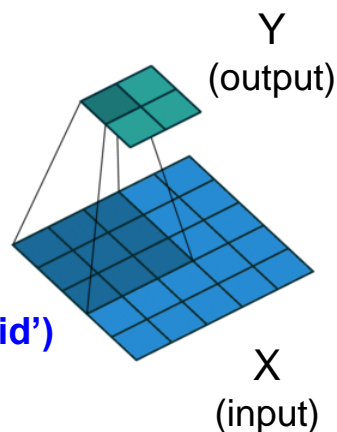
convolution, transposed convolution, stride, kernel size, padding

*V.Dumoulin, F.Visin - A guide to convolution arithmetic for deep learning (2018)*



## Conv2D

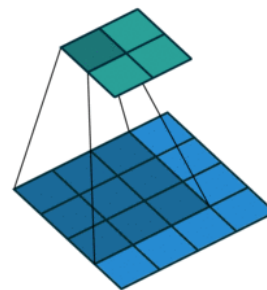
input size  $i = 5$   
kernel\_size  $k = 3$   
strides  $s = 2$   
padding  $p = 0$  ('valid')



output size  $o = 2$

## Conv2D

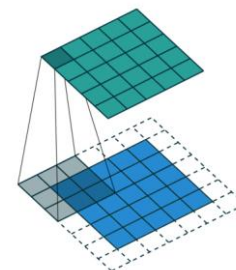
$i = 4$   
 $k = 3$   
 $s = 1$   
 $p = 0$



$o = 2$

## Conv2D

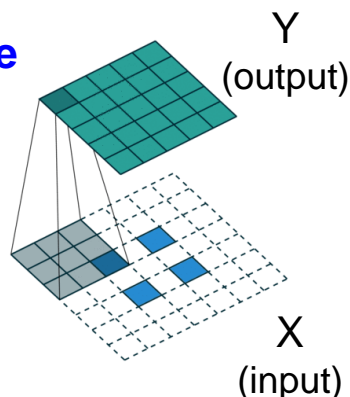
$i = 5$   
 $k = 3$   
 $s = 1$   
 $p = 1$  ('same')



$o = 5$

## Conv2DTranspose

input\_size  $i' = 2$   
kernel\_size  $k' = 3$   
strides  $s' = 2$   
padding  $p' = 2$



output\_size  $o' = 5$

## Conv2D:

$$i + 2*p = k + s*(o - 1)$$

'valid' padding:  $p = 0$

'same' padding:  $o = \text{round}(i / s)$

## Conv2DTranspose:

$$o' = i' + (i' - 1)*(s' - 1) + 2*p' - k' + 1$$

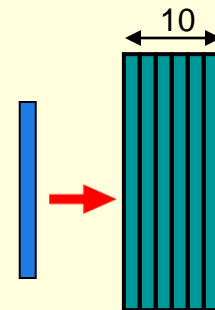
'valid' padding:  $p' = k' - 1$

'same' padding:  $o' = i' * s'$

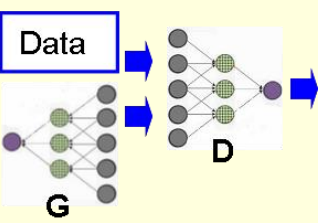
num. of  
filters /  
kernels

filter /  
kernel  
size

**Conv2DTranspose(10, 3, padding = 'same', ...)**

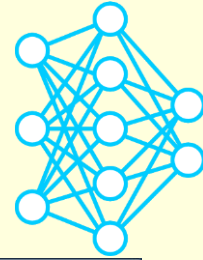






# The simple GAN code:

## (1) header and (2) defining a model



generator, discriminator, GAN, compile, loss, optimizer, trainable

### (1) Header:

- general Python imports
- Numpy imports
- Keras library imports

```
denisovga@biowulf:/data/denisovga/1_DL_Course/0_Intro
#!/usr/bin/env python
import os, re, random, collections
import numpy as np, tensorflow as tf
from tensorflow.keras.optimizers import RMSprop
from tensorflow.keras.layers import Conv1D, Flatten, \
Dense, Reshape, BatchNormalization, Activation, \
Conv1DTranspose

n_chan, f_size, seq_len, noise_len, init_len, seed, lr = 10, 16, 128, 50, 16, 2, 0.01
np.random.seed(seed); random.seed(seed); tf.random.set_seed(seed)
D = tf.keras.models.Sequential()
D.add(Conv1D(n_chan, f_size,
             activation='relu', input_shape=(seq_len, 1)))
D.add(Flatten())
D.add(Dense(1, activation='sigmoid'))
D.compile(loss='mse', optimizer=RMSprop(learning_rate=lr))
G = tf.keras.models.Sequential()
G.add(Dense(init_len*n_chan, input_shape=(noise_len,)))
G.add(Reshape((init_len, n_chan)))
G.add(BatchNormalization(momentum=0.1, epsilon=1.e-5))
for i in range(0, 2):
    G.add(Conv1DTranspose(n_chan, f_size, strides=2, padding="same"))
    G.add(BatchNormalization(momentum=0.8, epsilon=1.e-5))
G.add(Conv1DTranspose(1, f_size, strides=2, padding="same"))
G.add(Activation('sigmoid'))
GAN = Sequential()
GAN.add(G)
GAN.add(D)
GAN.trainable = False
GAN.compile(loss='mse', optimizer=RMSprop(learning_rate=lr))
```

### BatchNormalization layer

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$$

D outputs the probability that the input data is "good"

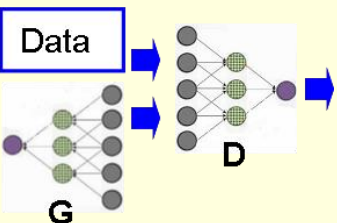
$$GAN = D(G(z))$$

- 1) like D, GAN outputs a probability
- 2) like G, GAN takes noise as input
- 3) only G weights are adjustable when training GAN

D

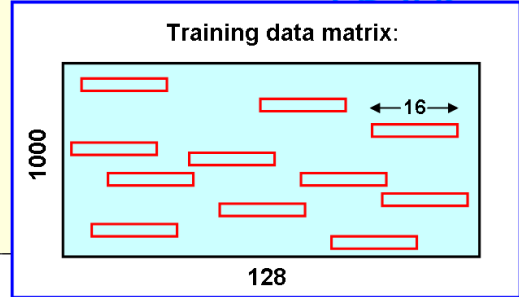
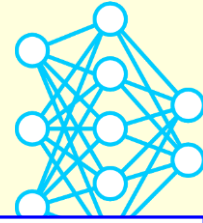
G

GAN



# The simple GAN code (cont.): getting data and training the model

motif, train\_on\_batch, epoch, batch size, predict



## (3) Get data

- motif

## (4) Run the model

- train\_on\_batch  
- num. of epochs  
- batch size

```
Select denisovga@biowulf:/usr/local/apps/DLBio/class4/bin

n_data,motif = 1000, "0011001100110011" if 1 else "0101010101010101"
x_str = [''.join([random.choice('01') for i in range(seq_len)]) \
          for j in range(n_data)]

for j in range(n_data):
    rint = np.random.randint(0, high=seq_len-len(motif))
    x_str[j]= x_str[j][:rint] + motif + x_str[j][(rint+len(motif)):]
x_train = np.reshape(np.array([[int(c) for c in x_str[j]] \
                                for j in range(n_data)]), [n_data,seq_len,1])

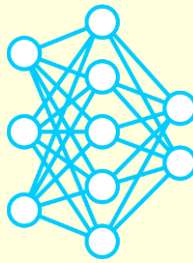
epochs, b_size, GAN_loss = 500, 10, 1.
for epoch in range(epochs+1):
    n_iterD,n_iterG=[5,0] if epoch <= 300 else [0,1]
    real_data = np.array(x_train[np.random.randint(0,x_train.shape[0],b_size)])
    real_data = np.reshape(real_data,(real_data.shape[0],real_data.shape[1],1))
    noise      = np.random.normal(0, 1, (b_size, noise_len))
    fake_data = G.predict(noise)
    for i in range(n_iterD):
        Dt_loss = D.train_on_batch(real_data, np.ones((b_size,1)))
        Df_loss = D.train_on_batch(fake_data, np.zeros((b_size,1)))
    for i in range(n_iterG):
        GAN_loss = GAN.train_on_batch(noise, np.ones((b_size,1)))
    if epoch%50 == 0:
        print(f'{epoch:5d}: Dt_loss= {Dt_loss:14.12f} ' + \
              f'Df_loss= {Df_loss:14.12f} ' + \
              f'GAN_loss= {GAN_loss:14.12f}')
```

### train\_on\_batch:

- train D: D (real data) → 1
- train D: D (fake data) → 0 (G weights frozen)
- train GAN: GAN (fake\_data) → 1 (D weights frozen)

# The GAN optimization objective

*A. Yadav, S. Shah et al., ICLR 2018*

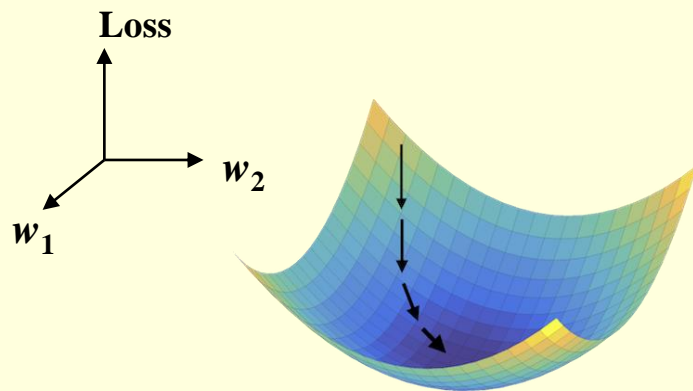


Re-write from the previous slide

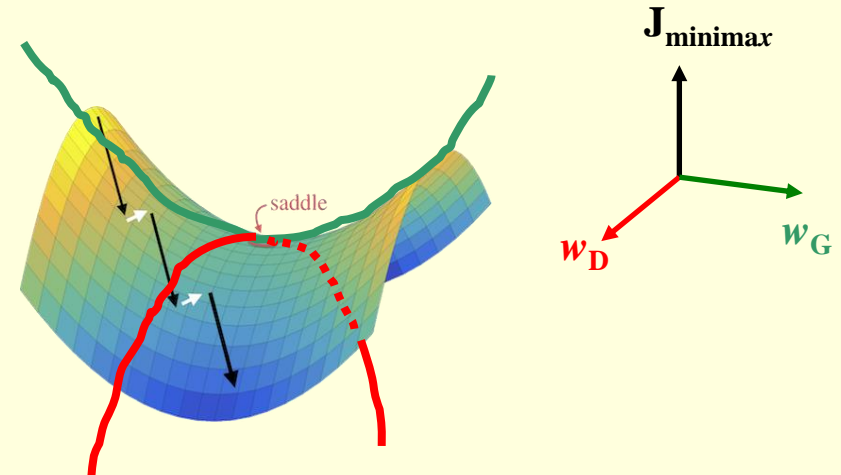
<b>real data</b> $\equiv \mathbf{x}$ ;	<b>fake data</b> $\equiv \mathbf{G}(\mathbf{z})$ ;	$\mathbf{z} \equiv \text{noise}$	
$\mathbf{D}(\mathbf{x}; \mathbf{w}_D) \rightarrow 1$	$\rightarrow$	$\ln \mathbf{D}(\mathbf{x}; \mathbf{w}_D)$	$\rightarrow \text{max}$
$\mathbf{D}(\mathbf{G}(\mathbf{z}; \mathbf{w}_G); \mathbf{w}_D) \rightarrow 0$	$\rightarrow$	$\ln(1 - \mathbf{D}(\mathbf{G}(\mathbf{z}; \mathbf{w}_G); \mathbf{w}_D))$	$\rightarrow \text{max}$
$\mathbf{D}(\mathbf{G}(\mathbf{z}; \mathbf{w}_G); \mathbf{w}_D) \rightarrow 1$	$\rightarrow$	$\ln(1 - \mathbf{D}(\mathbf{G}(\mathbf{z}; \mathbf{w}_G); \mathbf{w}_D))$	$\rightarrow \text{min}$

The minimax optimization objective:

$$J_{\text{minimax}} = \min_{\mathbf{w}_G} \max_{\mathbf{w}_D} E_{\text{data}} \{ \ln \mathbf{D}(\mathbf{x}; \mathbf{w}_D) \} + E_{\text{noise}} \{ \ln (1 - \mathbf{D}(\mathbf{G}(\mathbf{z}; \mathbf{w}_G); \mathbf{w}_D)) \}$$



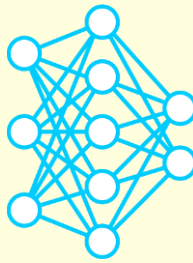
Standard neural net



Adversarial neural net

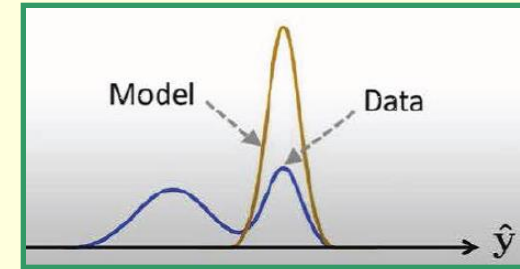


# Mode collapse



## What is mode collapse?

- an issue that often **occurs in GANs** due to **problems in training** when training/real **data comprise  $\geq 2$  types/"modes"**
- **generator** can only produce a **single type of output** or a small subset of types

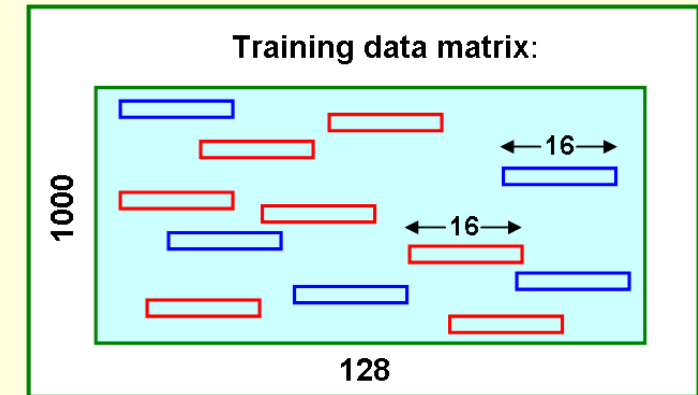


## The mode collapse prototype example:

**Input:** a training set of “good” sequences of 2 types (“modes”), each sequence containing a motif of one type

**Example:** 0101110011001100110011, 00010101010101010111010

**Task:** train a GAN on these data and then count the sequences of the two types in the data generated by model after training.

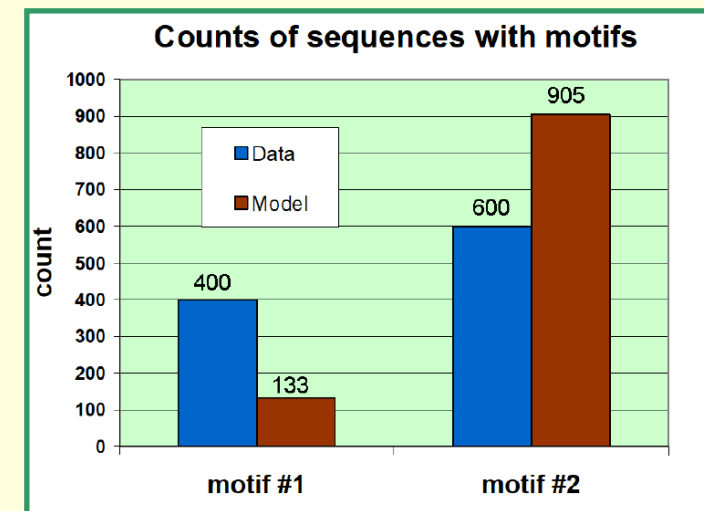


## Predictions from the model:

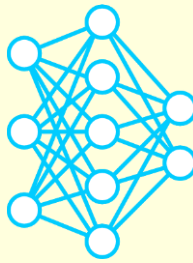
```
denisovga@biowulf:/data/denisovga/1_DL_Course/0_Intro
print("Prediction...")
G_count = [0, 0]
for i in range(1000):
    noise = np.random.normal(0,1,(1,noise_len))
    fake_data = G.predict(noise)
    G_seq = "".join([str(int(s)) for s \
                     in np.round(np.squeeze(fake_data))])
    if re.search(motif[0], G_seq): G_count[0] += 1
    if re.search(motif[1], G_seq): G_count[1] += 1
print("Training seqs matching motif0: " + str(T_count[0]) + \
      "/" + str(1000))
print("Training seqs matching motif1: " + str(T_count[1]) + \
      "/" + str(1000))
print("Generated seqs matching motif0: " + str(G_count[0]) + \
      "/" + str(1000))
print("Generated seqs matching motif1: " + str(G_count[1]) + \
      "/" + str(1000))
```

84,26 Bot

## Computed results:



# How to run the simple GAN examples on Biowulf?



## Executables

## Data with single motif

## Data with two motifs

```
denisovga@biowulf:/data/denisovga/1_DL_Course/0_Intro
$ sinteractive --gres=gpu:p100:1 --mem=4g

$ module load DLBio/class4
...
$ ls $DLBIO_BIN
simple_gan.py  mode_collapse.py

$ simple_gan.py
...
Training...
...
   0: Dt_loss= 0.058466792107   Df_loss=  0.366844356060   GAN_loss= 1.000000000000
  50: Dt_loss= 0.000000028773   Df_loss=  0.002305581467   GAN_loss= 1.000000000000
 100: Dt_loss= 0.000000000129   Df_loss=  0.000038872400   GAN_loss= 1.000000000000
...
500: Dt_loss= 0.000000000005   Df_loss=  0.000000004494   GAN_loss= 0.000000000000
Prediction...
Generated seqs matching motif: 999/1000
Random      seqs matching motif: 0/1000

$ mode_collapse.py
...
Training...
...
   0: Dt_loss= 0.057551991194   Df_loss=  0.501274228096   GAN_loss= 1.000000000000
  50: Dt_loss= 0.000000001346   Df_loss=  0.003325915430   GAN_loss= 1.000000000000
 100: Dt_loss= 0.000000000943   Df_loss=  0.000045182696   GAN_loss= 1.000000000000
...
500: Dt_loss= 0.000000000002   Df_loss=  0.000000004260   GAN_loss= 0.000000000000
Prediction...
Training seqs matching motif0: 400/1000
Training seqs matching motif1: 600/1000
Generated seqs matching motif0: 133/1000
Generated seqs matching motif1: 905/1000
```

16,1

All

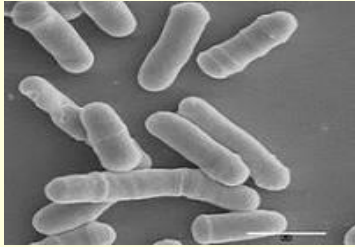
# Example 4. BioGANs: GANs for Biological Image Synthesis



A.Osokin e.a. *IEEE Int. Conf. on Computer Vision (ICCV), 2017*

<https://github.com/aosokin/biogans>

<https://hpc.nih.gov/apps/biogans.html>



Fission yeast cells



<b>Bgs4</b>	<b>Bgs4</b>	<b>Bgs4</b>	<b>Bgs4</b>	<b>Bgs4</b>	<b>Bgs4</b>	<b>Bgs4</b>
	+	+	+	+	+	+
	<b>Alp14</b>	<b>Arp3</b>	<b>Cki2</b>	<b>Mkh1</b>	<b>Sid2</b>	<b>Twa1</b>

**Proteins:** growth marker **Bgs4** and polarity factors **Alp14**, **Arp3**, **Cki2**, ...

**Biological task:** investigate how different polarity factors interact with one another

**Computational task:** train a GAN on available data and generate synthetic images that show localization of multiple polarity factors, together with **Bgs4**, at the same stage of cell growth cycle (i.e. the images that cannot be produced experimentally)

**Data:** the Localization Interdependency network (**LIN**) dataset

**The BioGANs pipeline (reimplemented in Keras from PyTorch):**

train.py

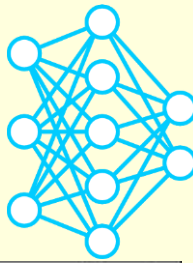


predict.py



visualize.py

# An overview of the BioGANs training code



**The Keras source code:**

**train.py, predict.py,**  
**visualize.py, gans.py,**  
**models.py, dataloader.py,**  
**options.py,**

## Header

- import statements
- parsing the command line options

## Getting data

- LIN dataset

## Define a (network) model

- models available:  
DCGAN,  
DCGAN-separable,  
DCGAN-starshaped

## Run the model

- GAN algorithms:  
(traditional) GAN  
WGAN  
WGAN-GP
- optimizer: RMSProp

<https://hpc.nih.gov/apps/biogans.html>

```
denisovga@biowulf:/data/denisovga/1_DL_Course/4_GANs
#!/usr/bin/env python

import os, sys, random
import numpy as np
import gans
from dataloader import get_data
from options import parse_training_arguments, process_options
from models import get_network_models

import tensorflow as tf
from tensorflow.keras.optimizers import Adam, RMSprop

# -----

if __name__ == '__main__':
    opt = parse_training_arguments()
    opt, DCGAN_model, gan_algorithm, optimizer = process_options("train", opt)

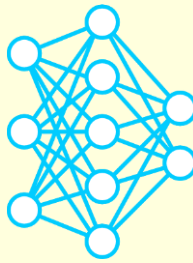
    # load data
    dataset, opt.n_classes = get_data(opt, "train")

    # Define a model
    os.environ['CUDA_VISIBLE_DEVICES'] = "0"
    if opt.num_gpus > 1:
        for j in range(1, opt.num_gpus):
            os.environ['CUDA_VISIBLE_DEVICES'] += "," + str(j)
    with tf.device('/cpu:0'):
        random.seed(opt.random_seed) # fix random seed
        netG, netD = get_network_models(DCGAN_model, opt, opt.red_portion)

    # Run the model
    if gan_algorithm == "GAN":
        gans.GAN(netG, netD, opt).train(dataset, opt)
    elif gan_algorithm == "WGAN":
        gans.WGAN(netG, netD, opt).train(dataset, opt)
    elif gan_algorithm == "WGAN-GP":
        gans.WGAN_GP(netG, netD, opt).train(dataset, opt)
    else:
        sys.exit("Undefined gan_algorithm: " + gan_algorithm + "\n")
```

39,9 A11

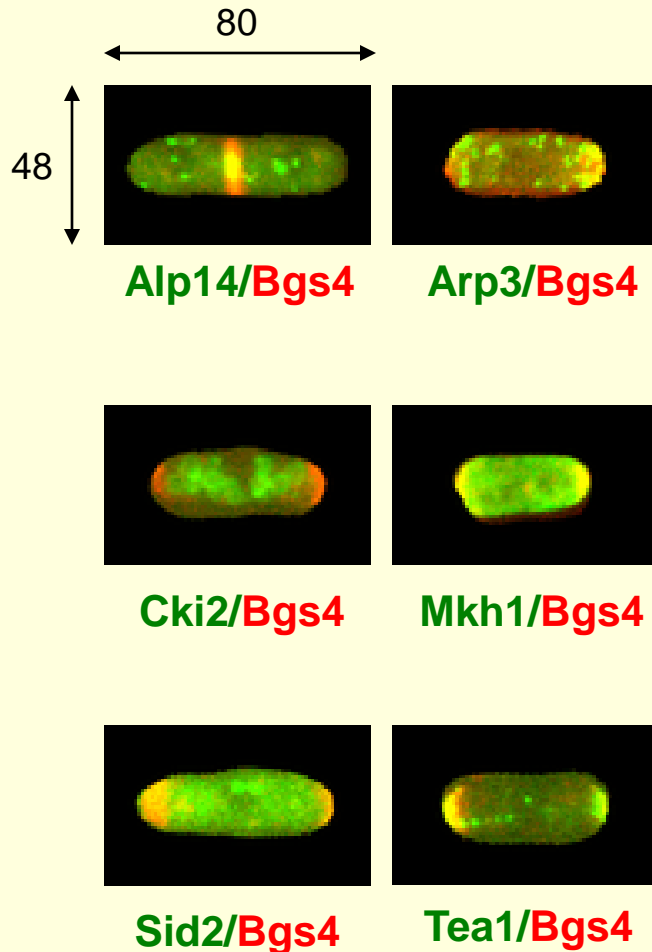
# BioGANs data: the Localization Interdependency Network (LIN) dataset



*J.Dodgson et al, <https://www.biorxiv.org/content/10.1101/116749v1.full>*

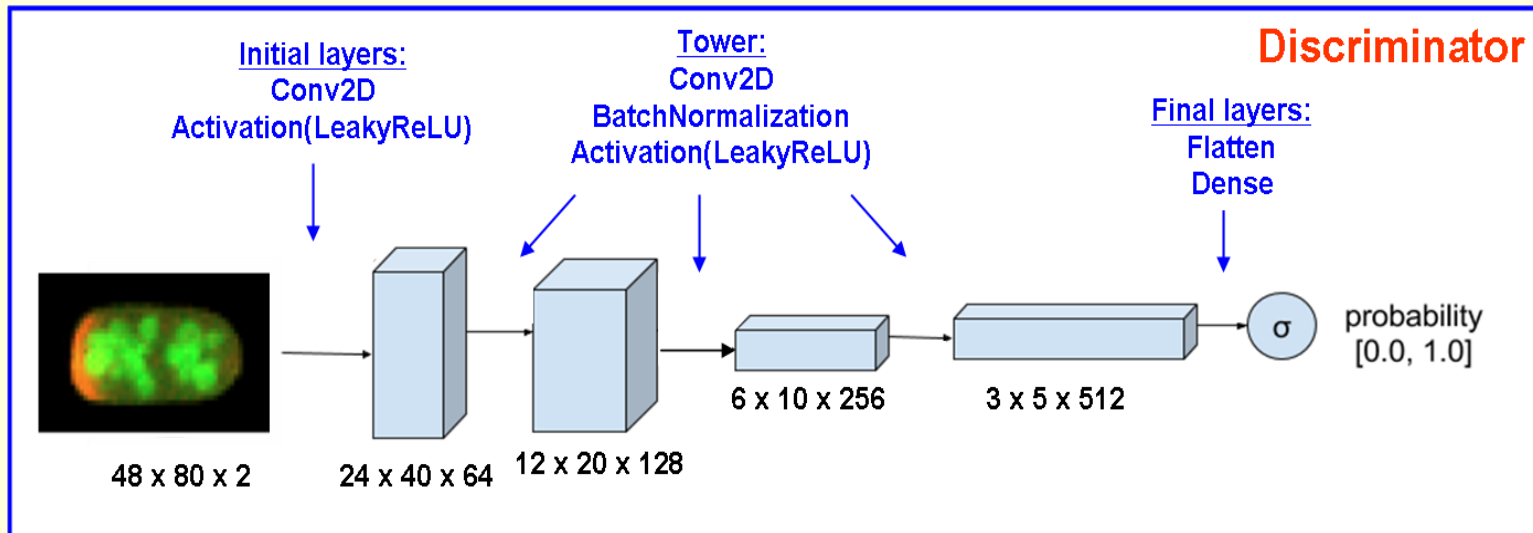
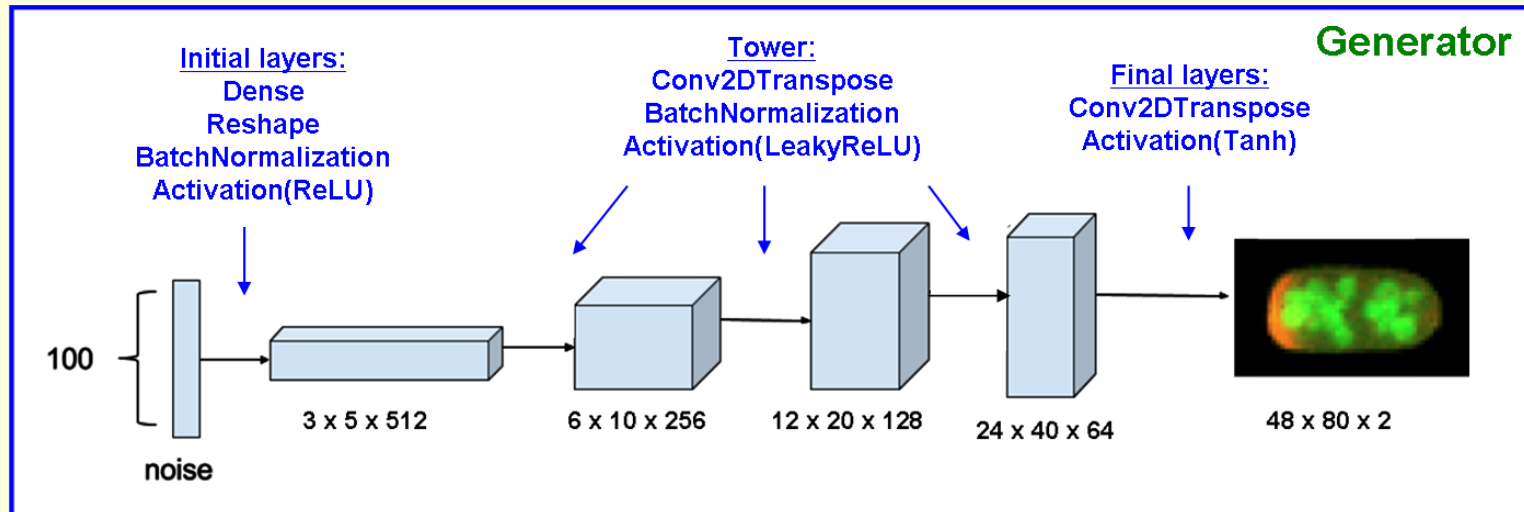
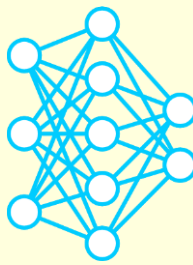
## Features:

- **2D fluorescence microscopy images** of Fission yeast cells, each  $(7 \div 14) \times 4 \mu\text{m}$
- 2-channel images of size is **48 x 80 pixels** (1 pixel = 100 nm)
- **red** channel = protein **Bgs4**, localizes in the **area of active growth**
- **green** channel = any of **41** different **polarity factors** that **define a cell geometry**
- **170,000 images** for **41** polarity factors available in the in the LIN dataset.
- the BioGANs application focuses on **Bgs4** and **6 polarity factors** **Alp14**, **Arp3**, **Cki2**, **Mkh1**, **Sid2** and **Tea1**, totaling to **26,909 images**





# The DCGAN model of the BioGANs application

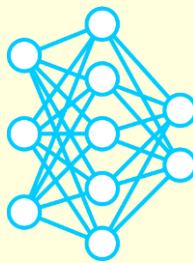


Features of the DCGAN architecture:

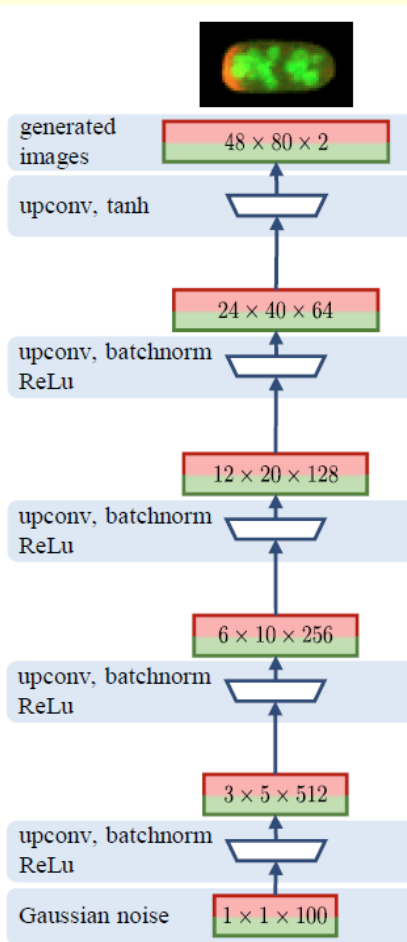
- the **most basic model** implemented as a part of the BioGANs application
- due to the **mode collapse**, can only generate **green channels** for a **subset** of polarity factors

# BioGANs generator architectures: DCGAN, DCGAN-separable and DCGAN-starshaped

A.Osokin e.a. *IEEE Int. Conf. on Computer Vision (ICCV), 2017*

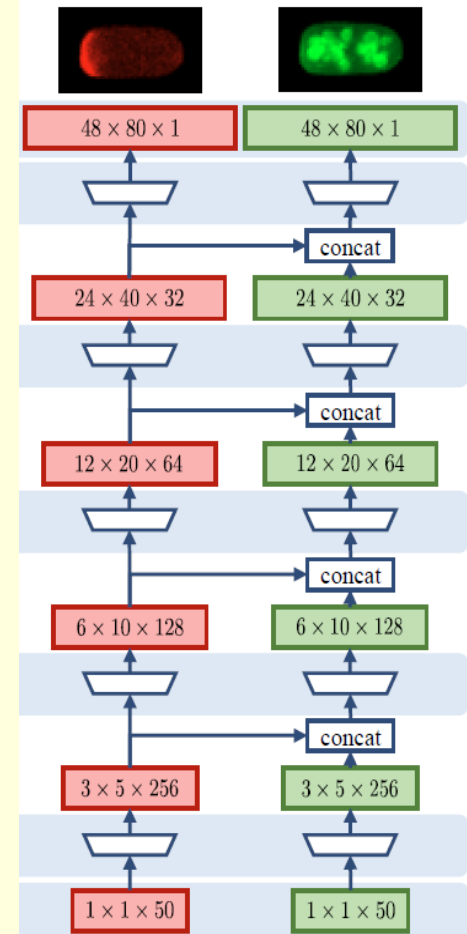


DCGAN



DCGAN-separable

DCGAN-starshaped

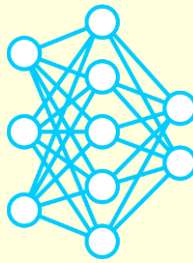


Two shortcomings of the DCGAN Generator:

- the signal in the green channel is not dependent on / influenced by the red channel
- cannot generate multiple green channels

# The Wasserstein GAN (WGAN)

M.Arjovsky et al, Wasserstein GAN – arXiv: 1701.07875 (2017)

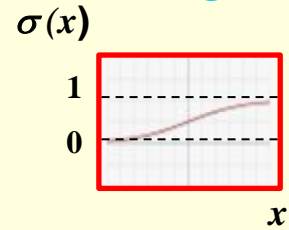


Problem with training vanilla GAN: **vanishing gradients** (discussed in class #2) due to the last/sigmoid layer in the Discriminator:

$$D(I, w) = \sigma(F(I, w)) \Rightarrow \nabla_w D = \sigma' \cdot \nabla_w F \rightarrow 0 \text{ at saturation}$$

WGAN ideas:

- **get rid of the  $\sigma$  layer** => can no longer use the BCE loss
- replace  $D$  with  $F$ ; rename  $F$  to **critic** and its output to **score  $s$**
- as a **new loss function**, use the **Earth Mover's distance (EMD)** between the distributions of the critic scores  $P_{Data}(s)$  and  $P_{Gen}(s)$



**EMD, a.k.a. Wasserstein loss = minimum amount of work to transform one distribution to another**

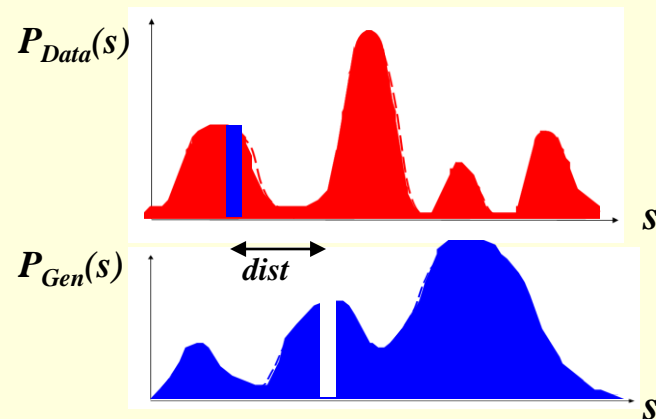
Binary cross entropy loss:

$$BCE = -\frac{1}{N} \sum_{i=1}^N y_i \cdot \log(p_i(w)) + (1 - y_i) \cdot \log(1 - p_i(w))$$

The approx. Wasserstein loss when the two distributions are similar/close:

$$EMD \approx -E[s^{Data} \cdot s^{Gen}]$$

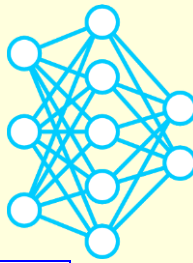
$EMD \rightarrow \min$   
forces the two distributions  
to have maxima  
at the same locations



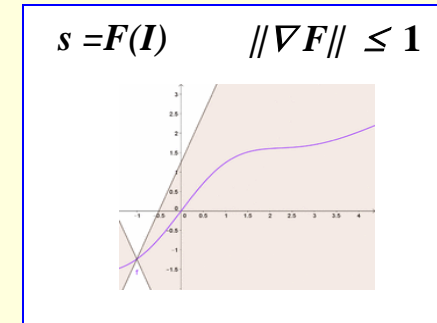
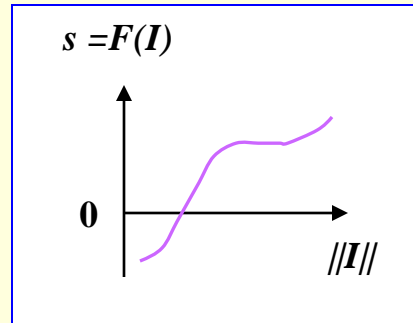
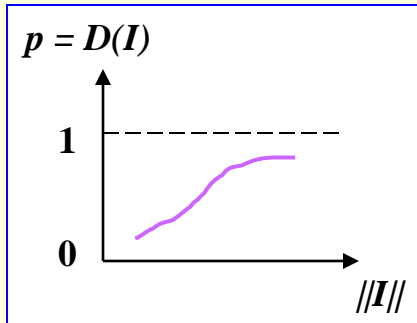
**Work = dist \* mass (=area of the piece)**

# WGAN with gradient penalty (WGAN-GP)

Gulrajani et al., Improved Training of Wasserstein GANs - arXiv:1704.00028v3 (2017)



How can we limit the growth of critic  $F$  to avoid exploding gradients / instability?



## (Vanilla) GAN:

use sigmoid activation:  $D(I) = \sigma(F(I))$

## WGAN:

clip the  $F$  weights that are beyond  $[-c, c]$

Data transform. by one layer:  $Z = A(\sum w_i \cdot X_i + b)$

## WGAN-GP:

penalize the loss to force  $\|\nabla F\|$  be close to 1

$$\text{WGAN-GP loss} = \text{EMD} + \lambda \cdot E[(\|\nabla F\| - 1)^2]$$

### WGAN features:

- (1) rename Discriminator  $D$  to Critic  $F$
- (2) use EMD loss
- (3) **clip all weights after each epoch ( $c = 0.01$ )**
- (4) use RMSProp optimizer with  $\text{lr} = 0.00005$

### WGAN-GP features:

- (1) } same as for WGAN
- (2) }
- (3) **penalize the loss using  $\nabla F$ ;  $\lambda \approx 10$**
- (4) use either RMSProp or Adam optimizer

## Training of WGAN(-GP) on data with $\|I\| \approx 1$ :

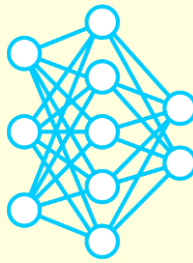
train\_on\_batch:

- train  $F$ :  $F(\text{real data}) \rightarrow -1$
- train  $F$ :  $F(\text{fake data}) \rightarrow 1$  (G weights frozen)
- train WGAN(-GP): WGAN(-GP) (fake data)  $\rightarrow -1$  (F weights frozen)

# The Root Mean Squared Propagation (RMSProp) optimizer

Slides: [http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture\\_slides\\_lec6.pdf](http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf)

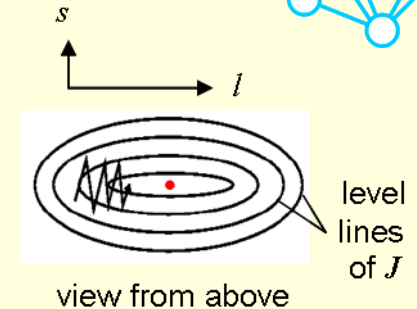
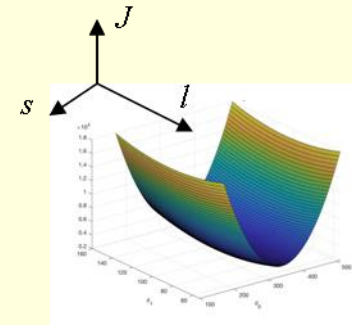
Video: <https://goo.gl/XUblyJ>



Basic gradient descent formula for updating weights:

$$w_{t+1} - w_t = -\gamma \cdot \nabla_w J(w_t)$$

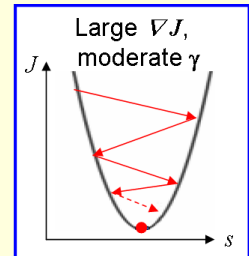
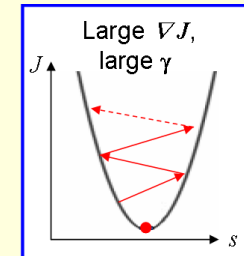
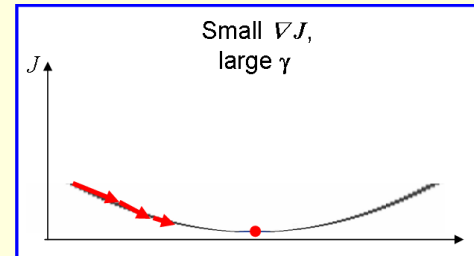
$w$  = vector of weights     $\gamma$  = learning rate  
 $t$  = update #     $\nabla_w J$  = gradient of the loss



The RMSprop gradient descent formula:

$$w_{t+1} = w_t - \frac{\text{effective } \gamma}{\sqrt{E[\nabla_w J(w_t)^2] + \varepsilon}} \cdot \nabla_w J(w_t)$$

$E[\dots]$  = running average of the magnitudes  
of recent gradient squares  
 $\varepsilon$  = small parameter



**Effective  $\gamma$ :**

- large when moving in the  $l$ -direction (with small  $\nabla J$ )
- moderate or small when moving in the  $s$ -direction (with large  $\nabla J$ )

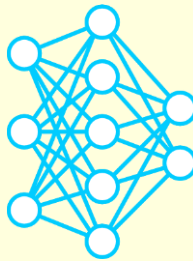
$$E[\nabla_w J(w)^2]_t = \rho \cdot E[\nabla_w J(w)^2]_{t-1} + (1 - \rho) \cdot \nabla_w J(w_t)^2$$

Computing the running average of  $(\nabla J)^2$ ,  
 $\rho \sim 0.9$

```
keras.optimizers.RMSprop( learning_rate=0.001, rho=0.9, epsilon=1e-07, ...)
```



# How to run the BioGANs application on Biowulf?



<https://hpc.nih.gov/apps/biogans.html>

```
denisovga@biowulf:/data/denisovga/1_DL_Course/4_GANs
sinteractive --mem=40g --gres=gpu:p100:1,lscratch:10

module load biogans

cp $BIOGANS_DATA/* .

ls $BIOGANS_SRC
predict.py  visualize.py  gans.py      options.py  utils.py
train.py    dataloader.py  models.py  __pycache__

train.py -d <data_folder> [-m <network_model>] [-a <gan_algorithm>]

# network_model = DCGAN, DCGAN-separable or DCGAN-starshaped (default)
# gan_algorithm = GAN, WGAN or WGAN-GP (default)

Example:
train.py -d data

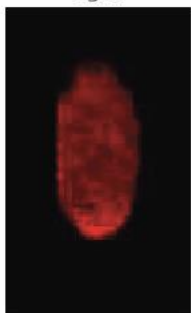
predict.py -i <checkpoint_file> [ other options ]

Example:
predict.py -i checkpoints/weights.generator.data.DCGAN-starshaped.WGAN-GP.6.h5

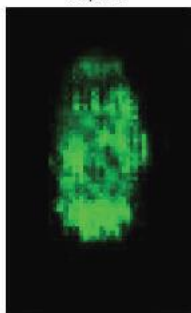
visualize.py -i <image_stack>

Example:
visualize.py -i images/stack7.DCGAN-starshaped.WGAN-GP.h5
```

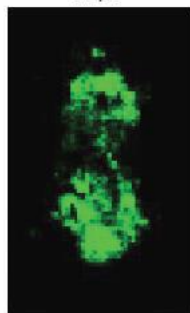
Bgs4



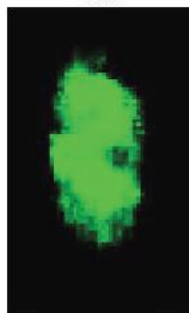
Alp14



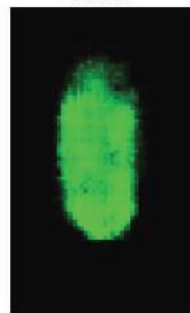
Arp3



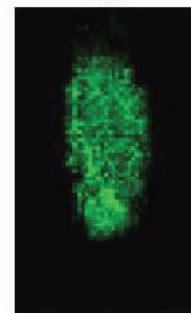
Cki2



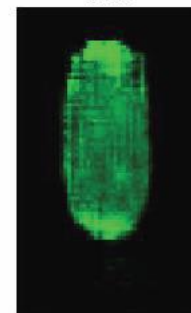
Mkh1



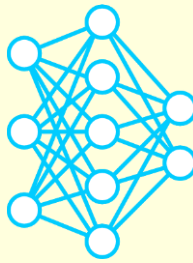
Sid2



Tea1



# Summary



## 1) Intro using a simple example

- simple **GAN** that can generate sequences containing certain motif(s):
  - D**iscriminator network is the same a model from class #2
  - G**enerator network produces a sequence from random noise
- the **Conv2DTranspose** (transposed convolution, a.k.a. deconvolution) layer
- the **BatchNormalization** layer
- the **train\_on\_batch** method
- the **mode collapse** issue

## 2) The BioGANs application:

- BioGANs data: the **LIN dataset**
- generator architectures: **DCGAN**, **DCGAN-separable** and **DCGAN-starshaped**
- discriminator/critic architectures: (vanilla) **GAN**, **WGAN** and **WGAN-GP**
- the **Earth Mover's distance (EMD) loss** and the **gradient penalty**
- the gradient descent-based optimization algorithm **RMSprop**