Chapter 4

Mechanism Based Neural Networks

Neural Networks for unsupervised problems

Mechanism Based Neural Networks

Neural Networks for unsupervised problems

1- Classification/Regression **Supervised Learning**

2- Pattern Association **Supervised Learning**

3- Applications without target patterns (Unsupervised Learning)

1 - Pattern Sorting **(Hard/Soft) Maximum to minimum sorting/…**

2 - Pattern Clustering **(Hard/Soft) data space partitioning (Similarity/Distance/….)**

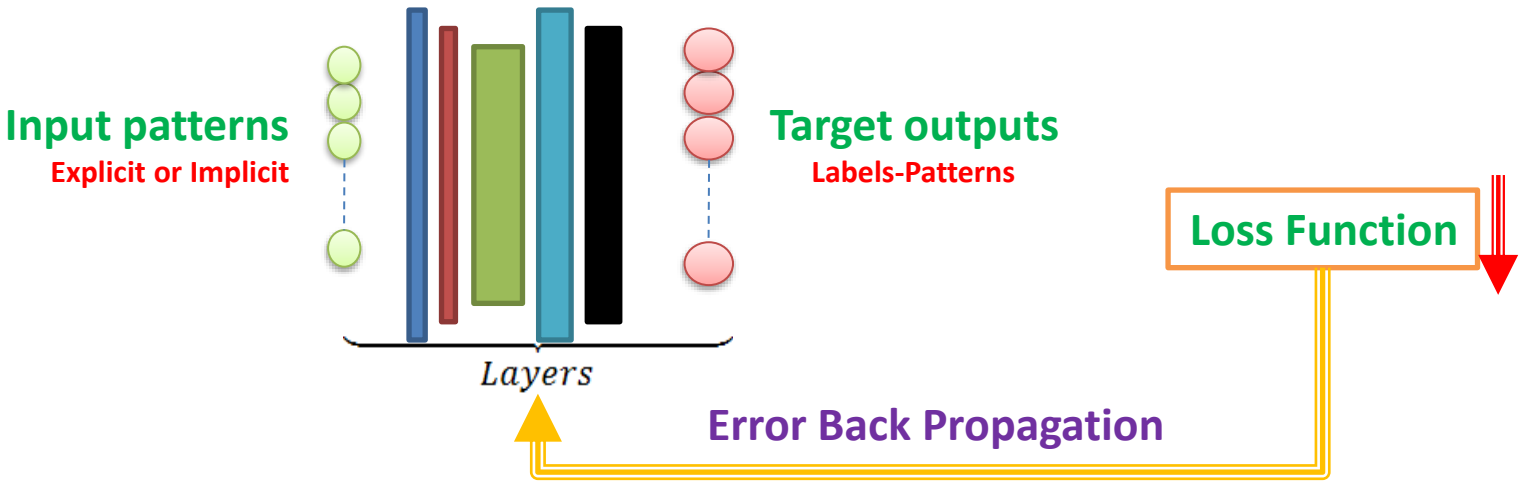
3 - Pattern Generation **Image Enhancement / Recommender Systems/ Fashion Design / Image Painting / Image Description/Text to Image…**

4 - Control tasks **Anti Oscillation/Optimal control /Loop shaping /….**

5 - …

"Two Common Design Approaches in NNs"

1- Layer-Based Design (For supervised Problems)



- Partitioning layers (MLP-MAdaline-….)
- Mapping layers (MLP-…)
- Encoding Layers (Auto-Encoder/RBM…)
- Decoding layers (Auto-Encoder/RBM…)
- Convolutional layers (CNN)
- Pooling/Subsampling layers (CNN)
- Recurrent Layers (RNN-LSTM-GRU…)
- Softmax/normalizing (CNN-…)

2-Mechnism-Based Design (For Unsupervised Problems)

Learning Strategies

(a variant parallel search)

1. Competitive based Learning
2. Cooperative based Learning
3. Adversarial based Learning
4. ….

Features Which Should be Satisfied in Solutions

Some Geometric features
(or) Statistical features
(or) Stability/Performance features
Within or between Distances–Similarity indices
–Distribution: Propagation-Density-
...surrounding-

Fixed –Weight Competitive Nets

- MaxNet

  A learning mechanism to determine the absolute largest node

- Mexican Hat

  A learning mechanism to determine soft largest node

- Hamming Net

  A learning mechanism to assign vectors to a certain number of vectors

MaxNet Lippmann 1987 (competitive mechanism)

Aim: there are m>1 nodes, it is desired to pick the node which is the largest.

Recurrent Structure

Architecture

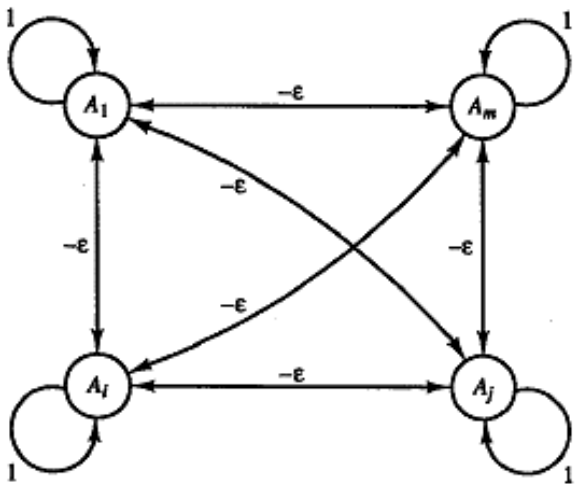


Figure 4.1 MAXNET.

Activation Function

$$f(x) = \begin{cases} x & \text{if } x \geqslant 0 \\ 0 & \text{otherwise.} \end{cases}$$

Weights/threshold

$$w_{ij} = \begin{cases} 1 & \text{if } i = j \\ -\epsilon & \text{if } i \neq j \end{cases}$$

$$0 < \epsilon < \frac{1}{m}$$

Updating Rule

$$a_j(\text{ new }) = f\left[a_j(\text{ old }) - \epsilon \sum_{k \neq j} a_k(\text{ old })\right]$$

MaxNet Algorithm/Example

Step 0. Initialize activations and weights (set $0 < \epsilon < \frac{1}{m}$):

$a_j(0)$ input to node A_j,

$$w_{ij} = \begin{cases} 1 & \text{if } i = j; \\ -\epsilon & \text{if } i \neq j. \end{cases}$$

Step 1. While stopping condition is false, do Steps 2–4.

Step 2. Update the activation of each node: For $j = 1, \ldots, m$,

$$a_j(\text{new}) = f[a_j(\text{old}) - \epsilon \sum_{k \neq j} a_k(\text{old})].$$

Step 3. Save activations for use in next iteration:

$$a_j(\text{old}) = a_j(\text{new}), j = 1, \ldots, m.$$

Step 4. Test stopping condition:
If more than one node has a nonzero activation, continue; otherwise, stop.

$$\boldsymbol{a}^{new} = \boldsymbol{f}\left(\begin{bmatrix} 1 & -\varepsilon & \cdots & -\varepsilon \\ -\varepsilon & 1 & \cdots & -\varepsilon \\ \vdots & \vdots & \ddots & \vdots \\ -\varepsilon & -\varepsilon & \cdots & 1 \end{bmatrix} \boldsymbol{a}^{old}\right)$$

Nonlinear Difference Equation

Example 4.1 Using a MAXNET

Consider the action of a MAXNET with four neurons and inhibitory weights $\epsilon = 0.2$ when given the initial activations (input signals)

$a_1(0) = 0.2$ $a_2(0) = 0.4$ $a_3(0) = 0.6$ $a_4(0) = 0.8$

The activations found as the net iterates are

$a_1(1) = 0.0$	$a_2(1) = 0.08$	$a_3(1) = 0.32$	$a_4(1) = 0.56$
$a_1(2) = 0.0$	$a_2(2) = 0.0$	$a_3(2) = 0.192$	$a_4(2) = 0.48$
$a_1(3) = 0.0$	$a_2(3) = 0.0$	$a_3(3) = 0.096$	$a_4(3) = 0.442$
$a_1(4) = 0.0$	$a_2(4) = 0.0$	$a_3(4) = 0.008$	$a_4(4) = 0.422$
$a_1(5) = 0.0$	$a_2(5) = 0.0$	$a_3(5) = 0.0$	$a_4(5) = 0.421$

Why $0 < \epsilon < \frac{1}{m}$?

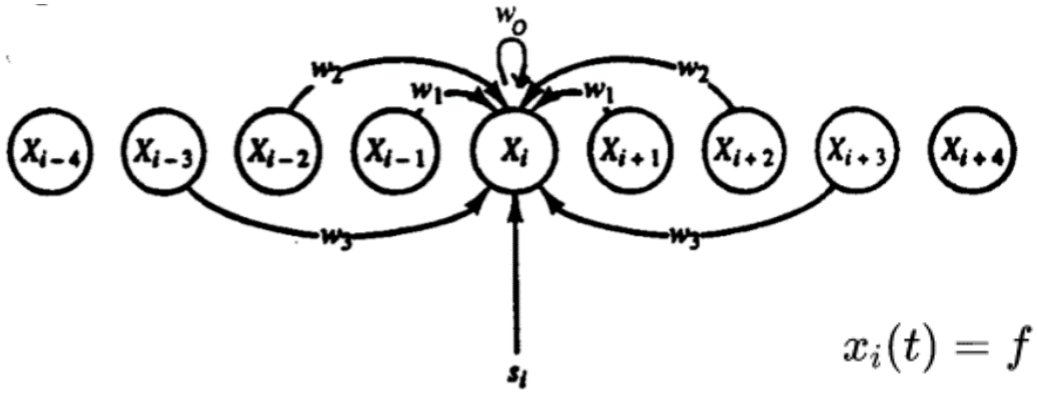
If the threshold is chosen so small …

If the threshold is chosen so the maximum .....

If some nodes are equal.....

Mexican Hat **Kohonen 1989** (competitive/cooperative mechanism)

Aim: there are m>1 arranged nodes, it is desired to pick the node which makes a soft maximum node, with some near adjacent nodes.



$$x_i(t) = f\left[s_i(t) + \sum_k w_k x_{i+k}(t-1)\right]$$

Figure 4.2 Mexican Hat interconnections for unit X_i.

For each node Xi there are:

1. some nearer nodes: X_j ,j∈{ i-R_1 …i+ R_1} which are wired with positive weights (cooperative nodes).
2. Some farer nodes: X_j ,j∈{ i-R_2 …i-R_1+1}U{ i+R_1+1,….i+R_2} wired with negative weights (competitive nodes)
3. other nodes which are out of cooperative or competitive nodes : X_j , j∈{….,i-R_2-1}U{i+R_2+1,…}, they do not wired to Xi (zero weight) .

Algorithm

The algorithm given here is similar to that presented by Kohonen [1989a]. The nomenclature we use is as follows:

R_2	Radius of region of interconnections; X_i is connected to units X_{i+k} and X_{i-k} for $k = 1, \ldots, R_2$.
R_1	Radius of region with positive reinforcement; $R_1 < R_2$.
w_k	Weight on interconnections between X_i and units X_{i+k} and X_{i-k}: w_k is positive for $0 \leq k \leq R_1$, w_k is negative for $R_1 < k \leq R_2$.
x	Vector of activations.
x_old	Vector of activations at previous time step.
t_max	Total number of iterations of contrast enhancement.
s	External signal.

Step 0. Initialize parameters *t_max*, R_1, R_2 as desired.
Initialize weights:

$w_k = C_1$ for $k = 0, \ldots, R_1$ $(C_1 > 0)$

$w_k = C_2$ for $k = R_1 + 1, \ldots, R_2$ $(C_2 < 0)$.

Initialize **x_old** to **0**.

Step 1. Present external signal **s**:

$\mathbf{x} = \mathbf{s}$.

Save activations in array **x_old** (for $i = 1, \ldots, n$):

$x\_old_i = x_i$.

Set iteration counter: $t = 1$.

Step 2. While t is less than *t_max*, do Steps 3–7.

Step 3. Compute net input ($i = 1, \ldots, n$):

$$x_i = C_1 \sum_{k=-R_1}^{R_1} x\_old_{i+k} + C_2 \sum_{k=-R_2}^{-R_1-1} x\_old_{i+k} + C_2 \sum_{k=R_1+1}^{R_2} x\_old_{i+k}.$$

Step 4. Apply activation function (ramp function from 0 to *x_max*, slope 1):

$x_i = \min(x\_max, \max(0, x_i))$ $(i = 1, \ldots, n)$.

Step 5. Save current activations in **x_old**:

$x\_old_i = x_i$ $(i = 1, \ldots, n)$.

Step 6. Increment iteration counter:

$t = t + 1$.

Step 7. Test stopping condition:
If $t < t\_max$, continue; otherwise, stop.

In a computer implementation of the algorithm, one simple method of dealing

Application

Example 4.2 Using the Mexican Hat Algorithm

We illustrate the Mexican Hat algorithm for a simple net with seven units. The activation function for this net is

$$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{if } 0 \le x \le 2 \\ 2 & \text{if } 2 < x. \end{cases}$$

Step 0. Initialize parameters:

$R_1 = 1;$

$R_2 = 2;$

$C_1 = 0.6;$

$C_2 = -0.4.$

Step 1. *(t = 0).*
The external signal is (0.0, 0.5, 0.8, 1.0, 0.8, 0.5, 0.0), so

$\mathbf{x} = (0.0, 0.5, 0.8, 1.0, 0.8, 0.5, 0.0).$

Save in x_old:

$\mathbf{x\_old} = (0.0, 0.5, 0.8, 1.0, 0.8, 0.5, 0.0).$

Step 2. *(t = 1).*
The update formulas used in Step 3 are listed as follows for reference:

$x_1 = 0.6\, x\_old_1 + 0.6\, x\_old_2 - 0.4\, x\_old_3$

$x_2 = 0.6\, x\_old_1 + 0.6\, x\_old_2 + 0.6\, x\_old_3 - 0.4\, x\_old_4$

$x_3 = -0.4\, x\_old_1 + 0.6\, x\_old_2 + 0.6\, x\_old_3 + 0.6\, x\_old_4 - 0.4\, x\_old_5$

$x_4 = -0.4\, x\_old_2 + 0.6\, x\_old_3 + 0.6\, x\_old_4 + 0.6\, x\_old_5 - 0.4\, x\_old_6$

$x_5 = -0.4\, x\_old_3 + 0.6\, x\_old_4 + 0.6\, x\_old_5 + 0.6\, x\_old_6 - 0.4\, x\_old_7$

$x_6 = -0.4\, x\_old_4 + 0.6\, x\_old_5 + 0.6\, x\_old_6 + 0.6\, x\_old_7$

$x_7 = -0.4\, x\_old_5 + 0.6\, x\_old_6 + 0.6\, x\_old_7.$

Step 3. *(t = 1).*

$x_1 = 0.6(0.0) + 0.6(0.5) - 0.4(0.8) = -0.2$

$x_2 = 0.6(0.0) + 0.6(0.5) + 0.6(0.8) - 0.4(1.0) = 0.38$

$x_3 = -0.4(0.0) + 0.6(0.5) + 0.6(0.8) + 0.6(1.0) - 0.4(0.8) = 1.06$

$x_4 = -0.4(0.5) + 0.6(0.8) + 0.6(1.0) + 0.6(0.8) - 0.4(0.5) = 1.16$

$x_5 = -0.4(0.8) + 0.6(1.0) + 0.6(0.8) + 0.6(0.5) - 0.4(0.0) = 1.06$

$x_6 = -0.4(1.0) + 0.6(0.8) + 0.6(0.5) + 0.6(0.0) = 0.38$

$x_7 = -0.4(0.8) + 0.6(0.5) + 0.6(0.0) = -0.2.$

Step 4.

$\mathbf{x} = (0.0, 0.38, 1.06, 1.16, 1.06, 0.38, 0.0).$

Steps 5–7. Bookkeeping for next iteration.

Step 3. *(t = 2).*

$x_1 = 0.6(0.0) + 0.6(0.38) - 0.4(1.06) = -0.196$

$x_2 = 0.6(0.0) + 0.6(0.38) + 0.6(1.06) - 0.4(1.16) = 0.39$

$x_3 = -0.4(0.0) + 0.6(0.38) + 0.6(1.06) + 0.6(1.16) - 0.4(1.06) = 1.14$

$x_4 = -0.4(0.38) + 0.6(1.06) + 0.6(1.16) + 0.6(1.06) - 0.4(0.38) = 1.66$

$x_5 = -0.4(1.06) + 0.6(1.16) + 0.6(1.06) + 0.6(0.38) - 0.4(0.0) = 1.14$

$x_6 = -0.4(1.16) + 0.6(1.06) + 0.6(0.38) + 0.6(0.0) = 0.39$

$x_7 = -0.4(1.06) + 0.6(0.38) + 0.6(0.0) = -0.196$

Step 4.

$\mathbf{x} = (0.0, 0.39, 1.14, 1.66, 1.14, 0.39, 0.0).$

Steps 5–7. Bookkeeping for next iteration.
The pattern of activations is shown for $t = 0$, 1, and 2 in Figure 4.3.

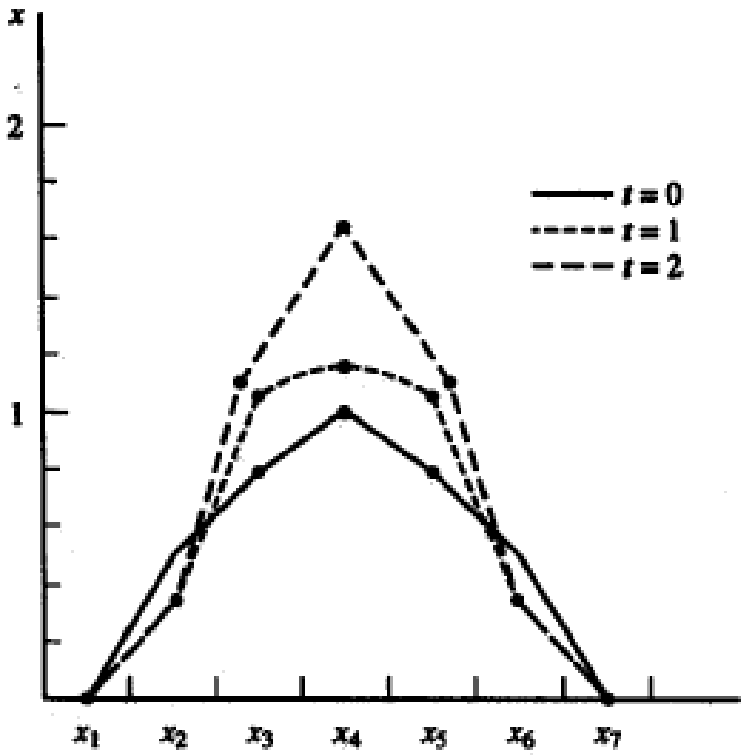


Figure 4.3 Results for Mexican Hat example.

Hamming Net **LippMann1987**
(competitive mechanism)

- There are some bipolar reference vectors. Using hamming distance index, every input vector assigns to a reference vector with minimum hamming distance.

For bipolar vectors *x* and y with n dimension.

$$\mathbf{x}\cdot\mathbf{y} = a - d,$$

a= number of equal bits

d= number of not equal bits

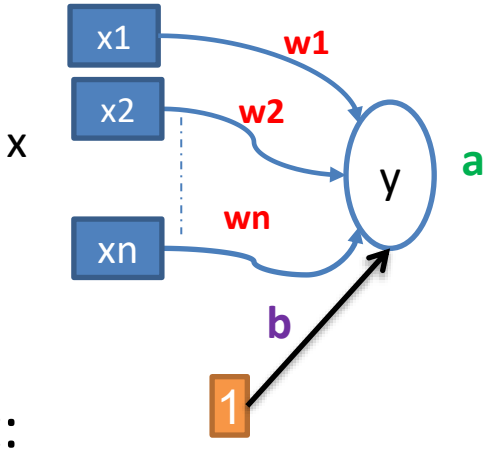
Also, it is known

$$d = n - a$$

Then we have:

$$a = x\left(\frac{y}{2}\right) + \left(\frac{n}{2}\right)$$

Assuming the weight vector as w=y/2 and the bias as b=n/2, we can make a mechanism to determine the similarity of x and y.



One can say that:

"the output of the above network shows the similarity of each exogenous x to y"

Assuming there are two n-dimensional references vectors y1 and y2

In Following network :

In the first layer, the similarity of x to references vectors: y_1=e(1) and y_2=e(2), is computed.

In the second layer, a MAXNET determines that which of two nodes has maximum value or which reference vectors (e(1) or e(2)) is more similar to x.

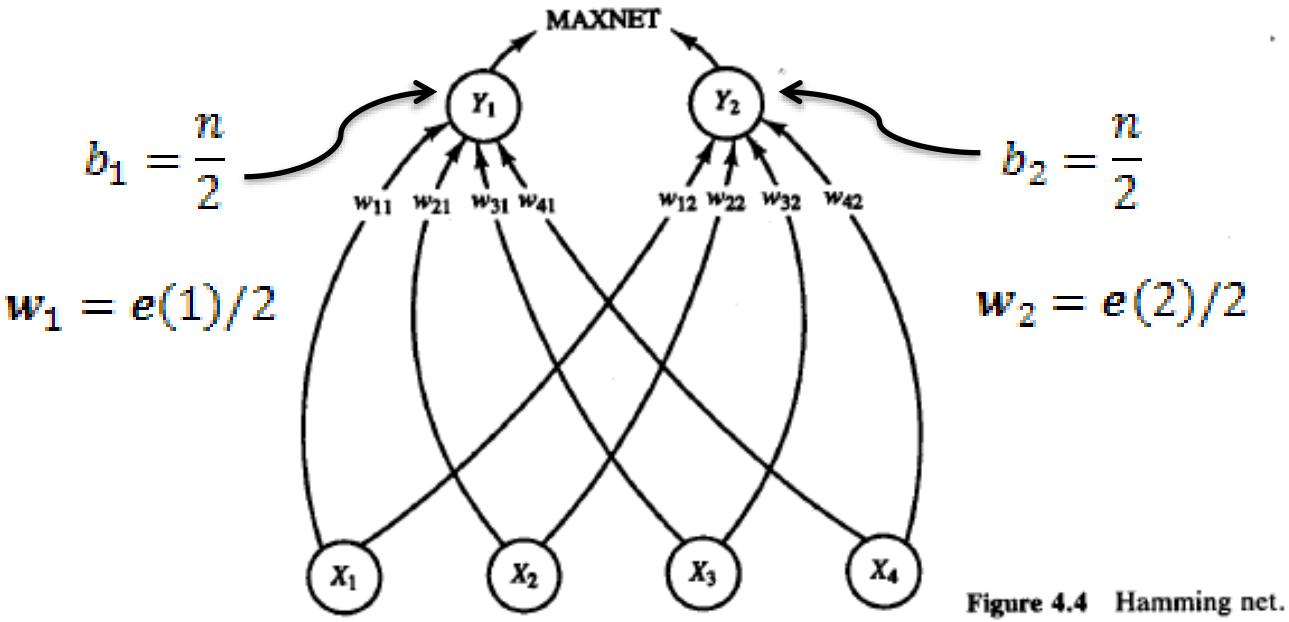
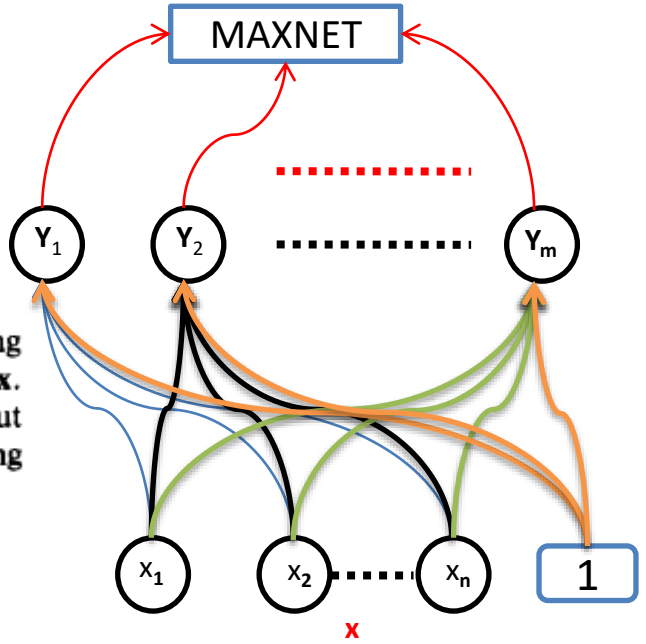


Figure 4.4 Hamming net.

Hamming Net For *m* reference vector



Application

Given a set of m bipolar exemplar vectors, $\mathbf{e}(1), \mathbf{e}(2), \ldots, \mathbf{e}(m)$, the Hamming net can be used to find the exemplar that is closest to the bipolar input vector $\mathbf{x}$. The net input y_in_j to unit Y_j, gives the number of components in which the input vector and the exemplar vector for unit $\mathbf{Y}_j$ $\mathbf{e}(j)$, agree (n minus the Hamming distance between these vectors).

The nomenclature we use is as follows:

n	number of input nodes, number of components of any input vector;
m	number of output nodes, number of exemplar vectors;
$\mathbf{e}(j)$	the jth exemplar vector:

$$\mathbf{e}(j) = (e_1(j), \ldots, e_i(j), \ldots, e_n(j)).$$

The application procedure for the Hamming net is:

Step 0. To store the m exemplar vectors, initialize the weights:

$$w_{ij} = \frac{e_i(j)}{2}, (i = 1, \ldots, n; j = 1, \ldots, m).$$

And initialize the biases:

$$b_j = \frac{n}{2}, (j = 1, \ldots, m).$$

The application procedure for the Hamming net is:

Step 1. For each vector $\mathbf{x}$, do Steps 2–4.

Step 2. Compute the net input to each unit Y_j:

$$y\_in_j = b_j + \sum_i x_i w_{ij}, (j = 1, \ldots, m).$$

Step 3. Initialize activations for MAXNET:

$$y_j(0) = y\_in_j, (j = 1, \ldots, m).$$

Step 4. MAXNET iterates to find the best match exemplar

Example 4.3 A Hamming net to cluster four vectors

Given the exemplar vectors

$$\mathbf{e}(1) = (1, -1, -1, -1)$$

and

$$\mathbf{e}(2) = (-1, -1, -1, 1),$$

the Hamming net can be used to find the exemplar that is closest to each of the bipolar input patterns, (1, 1, −1, −1), (1, −1, −1, −1), (−1, −1, −1, 1), and (−1, −1, 1, 1).

Step 0. Store the m exemplar vectors in the weights:

$$\mathbf{W} = \begin{bmatrix} .5 & -.5 \\ -.5 & -.5 \\ -.5 & -.5 \\ -.5 & .5 \end{bmatrix}.$$

Initialize the biases:

$$b_1 = b_2 = 2.$$

Step 1. For the vector $\mathbf{x} = (1, 1, -1, -1)$, do Steps 2–4.

Step 2. $y\_in_1 = b_1 + \sum_i x_i w_{i1}$

$$= 2 + 1 = 3;$$

$$y\_in_2 = b_2 + \sum_i x_i w_{i2}$$

$$= 2 - 1 = 1.$$

These values represent the Hamming similarity because (1, 1, −1, −1) agrees with $\mathbf{e}(1) = (1, -1, -1, -1)$ in the first, third, and fourth components and because (1, 1, −1, −1) agrees with $\mathbf{e}(2) = (-1, -1, -1, 1)$ in only the third component.

Step 3. $y_1(0) = 3;$

$$y_2(0) = 1.$$

Step 4. Since $y_1(0) > y_2(0)$, MAXNET will find that unit Y_1 has the best match exemplar for input vector $\mathbf{x} = (1, 1, -1, -1)$.

Step 1. For the vector $\mathbf{x} = (1, -1, -1, -1)$, do Steps 2–4.

Step 2. $y\_in_1 = b_1 + \sum_i x_i w_{i1}$

$$= 2 + 2 = 4;$$

$$y\_in_2 = b_2 + \sum_i x_i w_{i2}$$

$$= 2 + 0 = 2.$$

Note that the input vector agrees with $\mathbf{e}(1)$ in all four components and agrees with $\mathbf{e}(2)$ in the second and third components.

Step 3. $y_1(0) = 4;$

$$y_2(0) = 2.$$

Step 4. Since $y_1(0) > y_2(0)$, MAXNET will find that unit Y_1 has the best match exemplar for input vector $\mathbf{x} = (1, -1, -1, -1)$.

Step 1. For the vector $\mathbf{x} = (-1, -1, -1, 1)$, do Steps 2–4.

Step 2. $y\_in_1 = b_1 + \sum_i x_i w_{i1}$

$$= 2 + 0 = 2;$$

$$y\_in_2 = b_2 + \sum_i x_i w_{i2}$$

$$= 2 + 2 = 4.$$

The input vector agrees with $\mathbf{e}(1)$ in the second and third components and agrees with $\mathbf{e}(2)$ in all four components.

Step 3. $y_1(0) = 2;$

$$y_2(0) = 4.$$

Step 4. Since $y_2(0) > y_1(0)$, MAXNET will find that unit Y_2 has the best match exemplar for input vector $\mathbf{x} = (-1, -1, -1, 1)$.

Self Organizing Map (SOM) Kohonen 1989

- **It is aimed to :**

  "divide all input n-dimensional patterns to m different clusters "

  "Some of *m* clusters may be created as null sets"

Desired Solution:

clusters should have small within distances and large between distances

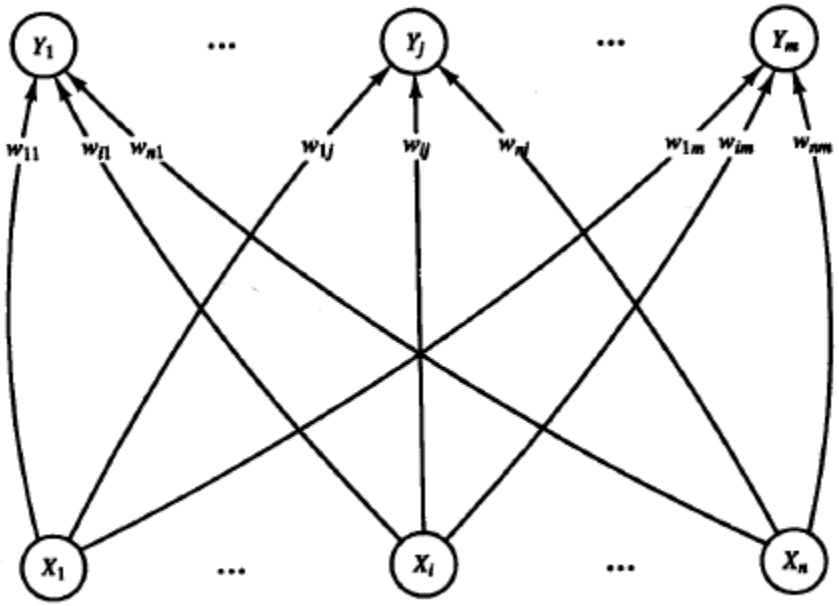


Figure 4.5 Kohonen self-organizing map.

IN "SOM" Each Neuron represents a cluster

Each neuron has a form of Neighborhoods

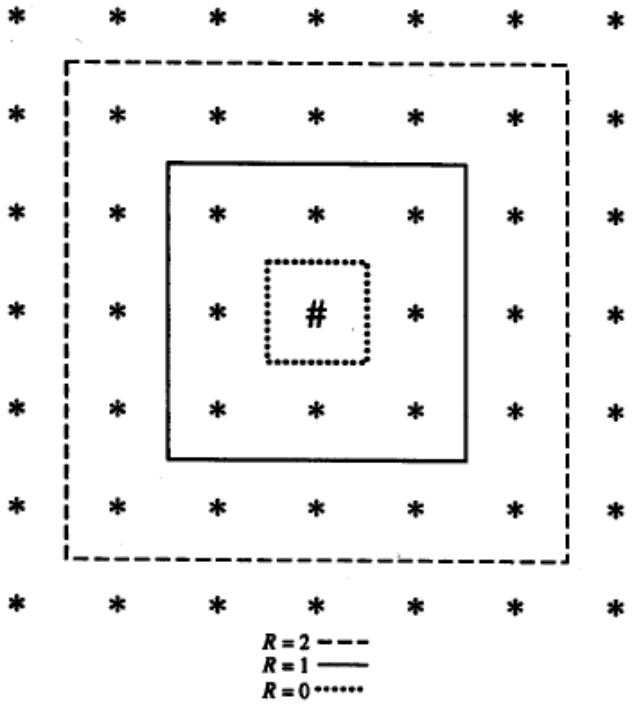


Figure 4.7 Neighborhoods f
rectangular grid.

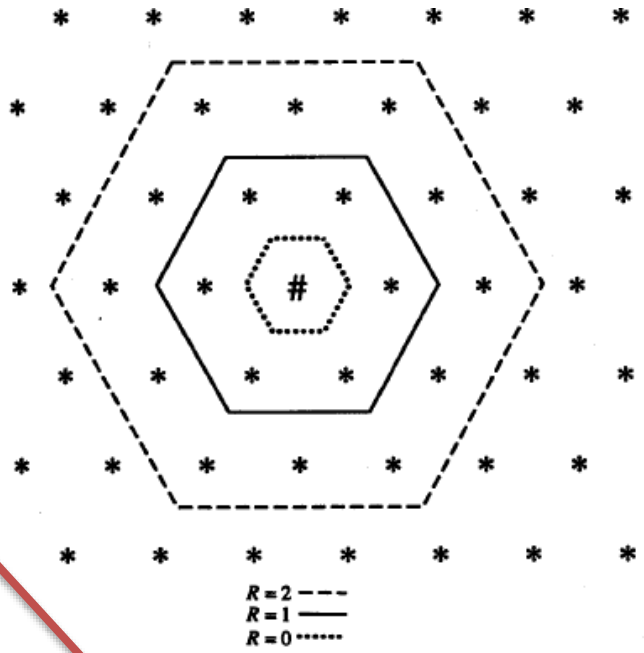


Figure 4.8 Neighborhoods for
hexagonal grid.

SOM Algorithm

4.2.2 Algorithm

Step 0. Initialize weights w_{ij}. (Possible choices are discussed below.)
Set topological neighborhood parameters.
Set learning rate parameters.

Step 1. While stopping condition is false, do Steps 2–8.

Step 2. For each input vector **x**, do Steps 3–5.

Step 3. For each j, compute:

$$D(j) = \sum_i (w_{ij} - x_i)^2.$$

Step 4. Find index J such that $D(J)$ is a minimum.

Step 5. For all units j within a specified neighborhood of J, and for all i:

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + \alpha[x_i - w_{ij}(\text{old})].$$

Step 6. Update learning rate.

Step 7. Reduce radius of topological neighborhood at specified times.

Step 8. Test stopping condition.

Some Notes about SOM Algorithm

1. α as learning rate can be slowly reduced linearly or geometrically (step 6).
2. The Initial considered large learning rate α causes that nodes soon get randomly values among the data points.
3. **The updating rule is the same gradient of squared error. The algorithm acts similar to k-mean clustering.**
4. **The initial considered neighborhood radius cause some nodes make a cooperation to find a cluster in a certain area.**
5. **By gradually decreasing the neighborhood radius the cooperative between near nodes is vanished and they start to compete with each other.**

Two Examples

Simple example

Example 4.4 A Kohonen self-organizing map (SOM) to cluster four vectors

Let the vectors to be clustered be

$$(1, 1, 0, 0); (0, 0, 0, 1); (1, 0, 0, 0); (0, 0, 1, 1).$$

The maximum number of clusters to be formed is

$$m = 2.$$

Suppose the learning rate (geometric decrease) is

$$\alpha(0) = .6,$$

$$\alpha(t + 1) = .5\, \alpha(t).$$

With only two clusters available, the neighborhood of node J (Step 4) is set so that only one cluster updates its weights at each step (i.e., $R = 0$).

Step 0. Initial weight matrix:

$$\begin{bmatrix} .2 & .8 \\ .6 & .4 \\ .5 & .7 \\ .9 & .3 \end{bmatrix}.$$

Initial radius:

$$R = 0.$$

Initial learning rate:

$$\alpha(0) = 0.6.$$

Step 1. Begin training.

Step 2. For the first vector, (1, 1, 0, 0), do Steps 3–5.

Step 3.

$$D(1) = (.2 - 1)^2 + (.6 - 1)^2 + (.5 - 0)^2 + (.9 - 0)^2 = 1.86;$$

$$D(2) = (.8 - 1)^2 + (.4 - 1)^2 + (.7 - 0)^2 + (.3 - 0)^2 = 0.98.$$

Step 4. The input vector is closest to output node 2, so

$$J = 2.$$

Step 5. The weights on the winning unit are updated:

$$w_{i2}(\text{new}) = w_{i2}(\text{old}) + .6\,[x_i - w_{i2}(\text{old})]$$

$$= .4\, w_{i2}(\text{old}) + .6\, x_i.$$

This gives the weight matrix

$$\begin{bmatrix} .2 & .92 \\ .6 & .76 \\ .5 & .28 \\ .9 & .12 \end{bmatrix}.$$

Step 2. For the second vector, (0, 0, 0, 1), do Steps 3–5.

Step 3.

$$D(1) = (.2 - 0)^2 + (.6 - 0)^2 + (.5 - 0)^2 + (.9 - 1)^2 = 0.66;$$

$$D(2) = (.92 - 0)^2 + (.76 - 0)^2 + (.28 - 0)^2 + (.12 - 1)^2 = 2.2768.$$

Step 4. The input vector is closest to output node 1, so

$$J = 1.$$

Step 5. Update the first column of the weight matrix:

$$\begin{bmatrix} .08 & .92 \\ .24 & .76 \\ .20 & .28 \\ .96 & .12 \end{bmatrix}.$$

Step 2. For the third vector, (1, 0, 0, 0), do Steps 3–5.

Step 3.

$$D(1) = (.08 - 1)^2 + (.24 - 0)^2 + (.2 - 0)^2 + (.96 - 0)^2 = 1.8656;$$

$$D(2) = (.92 - 1)^2 + (.76 - 0)^2 + (.28 - 0)^2 + (.12 - 0)^2 = 0.6768.$$

Step 4. The input vector is closest to output node 2, so

$$J = 2.$$

Step 5. Update the second column of the weight matrix:

$$\begin{bmatrix} .08 & .968 \\ .24 & .304 \\ .20 & .112 \\ .96 & .048 \end{bmatrix}.$$

Step 2. For the fourth vector, (0, 0, 1, 1), do Steps 3–5.

Step 3.

$$D(1) = (.08 - 0)^2 + (.24 - 0)^2 + (.2 - 1)^2 + (.96 - 1)^2 = 0.7056;$$

$$D(2) = (.968 - 0)^2 + (.304 - 0)^2 + (.112 - 1)^2 + (.048 - 1)^2 = 2.724.$$

Step 4.

$$J = 1.$$

Step 5. Update the first column of the weight matrix:

$$\begin{bmatrix} .032 & .968 \\ .096 & .304 \\ .680 & .112 \\ .984 & .048 \end{bmatrix}.$$

Step 6. Reduce the learning rate:

$$\alpha = .5\,(0.6) = .3$$

The weight update equations are now

$$w_{ij}(\text{new}) = w_{ij}(\text{old}) + .3\,[x_i - w_{ij}(\text{old})]$$
$$= .7w_{ij}(\text{old}) + .3x_i.$$

The weight matrix after the second epoch of training is

$$\begin{bmatrix} .016 & .980 \\ .047 & .360 \\ .630 & .055 \\ .999 & .024 \end{bmatrix}$$

Modifying the adjustment procedure for the learning rate so that it decreases geometrically from .6 to .01 over 100 iterations (epochs) gives the following results:

Iteration 0: Weight matrix: $\begin{bmatrix} .2 & .8 \\ .6 & .4 \\ .5 & .7 \\ .9 & .3 \end{bmatrix}$

Iteration 1: Weight matrix: $\begin{bmatrix} .032 & .970 \\ .096 & .300 \\ .680 & .110 \\ .980 & .048 \end{bmatrix}$

Iteration 2: Weight matrix: $\begin{bmatrix} .0053 & .9900 \\ -.1700 & .3000 \\ .7000 & .0200 \\ 1.0000 & .0086 \end{bmatrix}$

Iteration 10: Weight matrix: $\begin{bmatrix} 1.5e\text{-}7 & 1.0000 \\ 4.6e\text{-}7 & .3700 \\ .6300 & 5.4e\text{-}7 \\ 1.0000 & 2.3e\text{-}7 \end{bmatrix}$

Iteration 50: Weight matrix: $\begin{bmatrix} 1.9e\text{-}19 & 1.0000 \\ 5.7e\text{-}15 & .4700 \\ .5300 & 6.6e\text{-}15 \\ 1.0000 & 2.8e\text{-}15 \end{bmatrix}$

Iteration 100: Weight matrix: $\begin{bmatrix} 6.7e\text{-}17 & 1.0000 \\ 2.0e\text{-}16 & .4900 \\ .5100 & 2.3e\text{-}16 \\ 1.0000 & 1.0e\text{-}16 \end{bmatrix}$

These weight matrices appear to be converging to the matrix

$$\begin{bmatrix} 0.0 & 1.0 \\ 0.0 & 0.5 \\ 0.5 & 0.0 \\ 1.0 & 0.0 \end{bmatrix},$$

the first column of which is the average of the two vectors placed in cluster 1 and the second column of which is the average of the two vectors placed in cluster 2.

Character Recognition

Examples 4.5–4.7 show typical results from using a Kohonen self-organizing map to cluster input patterns representing letters in three different fonts. The input patterns for fonts 1, 2, and 3 are given in Figure 4.9. In each of the examples, 25 cluster units are available, which means that a maximum of 25 clusters may be formed. Results are shown only for the units that are actually the winning unit for some input pattern after training. The effect of the topological structure is seen in the contrast between Example 4.5 (in which there is no structure), Example 4.6 (in which there is a linear structure as described before), and Example 4.7 (in which a rectangular structure is used). In each example, the learning rate is reduced linearly from an initial value of .6 to a final value of .01.

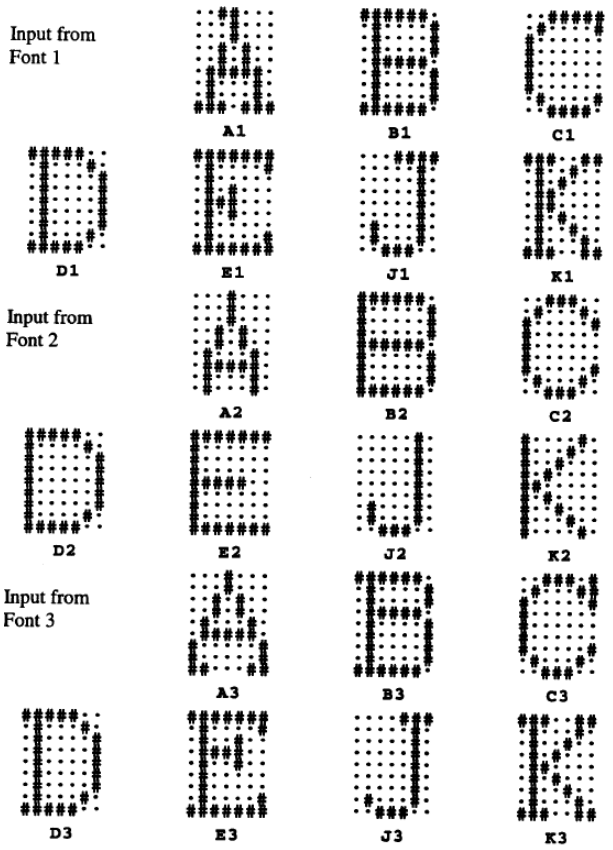


Figure 4.9 Training input patterns for character recognition examples.

Example 4.5 A SOM to cluster letters from different fonts: no topological structure

If no structure is assumed for the cluster units, i.e., if only the winning unit is allowed to learn the pattern presented, the 21 patterns form 5 clusters:

UNIT	PATTERNS
3	C1, C2, C3
13	B1, B3, D1, D3, E1, K1, K3, E3
16	A1, A2, A3
18	J1, J2, J3
24	B2, D2, E2, K2

Example 4.6 A SOM to cluster letters from different fonts: linear structure

A linear structure (with $R = 1$) gives a better distribution of the patterns onto the available cluster units. The winning node J and its topological neighbors ($J + 1$ and $J - 1$) are allowed to learn on each iteration. Note that in general, the neighboring nodes that learn do not initially have weight vectors that are particularly close to the input pattern.

UNIT	PATTERNS	UNIT	PATTERNS
6	K2	20	C1, C2, C3
10	J1, J2, J3	22	D2
14	E1, E3	23	B2, E2
16	K1, K3	25	A1, A2, A3
18	B1, B3, D1, D3		

Example 4.7 A SOM to cluster letters from different fonts: diamond structure

In this example, a simple two-dimensional topology is assumed for the cluster units, so that each cluster unit is indexed by two subscripts. If unit X_{IJ} is the winning unit, the units $X_{I+1,J}$, $X_{I-1,J}$, $X_{I,J+1}$, and $X_{I,J-1}$ also learn. This gives a diamond to-

i\j	1	2	3	4	5
1		J1, J2, J3		D2	
2	C1, C2, C3		D1, D3		B2, E2
3		B1		K2	
4			E1, E3, B3		A3
5		K1, K3		A1, A2	

Figure 4.10 Character recognition with rectangular grid.

pology, rather than the entire rectangle illustrated in Figure 4.7. The results are shown in Figure 4.10.

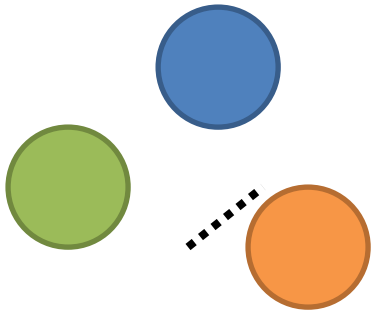
Extensions of SOM

For different cluster shapes

- Standard SOM (hyper circle clusters) k means/subtractive

$$D(j) = \sum_{i=1}^{n} (wij - x_i)^2 = (\boldsymbol{w}_j - \boldsymbol{x})^T (\boldsymbol{w}_j - \boldsymbol{x})$$

$$w_{ij}^{+} = w_{ij}^{-} - \eta \frac{\partial D(j)}{\partial w_{ij}}\bigg|_{w_{ij}^{-}} \quad i\epsilon\{1,2,\ldots,n\}$$

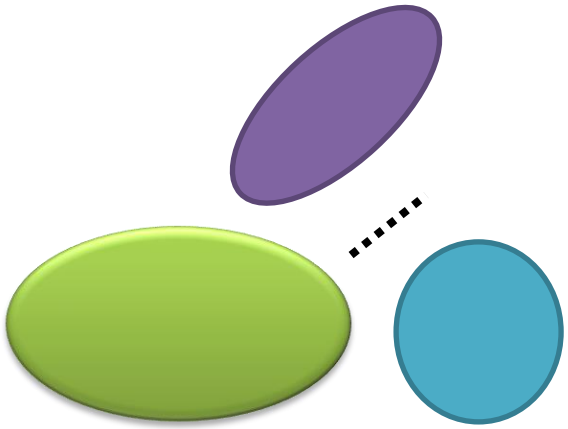


- Extended SOM (hyper ellipse clusters) G. K. / G. G.

$$D(j) = (\boldsymbol{w}_j - \boldsymbol{x})^T \boldsymbol{\Sigma}_j (\boldsymbol{w}_j - \boldsymbol{x})$$

$$w_{ij}^{+} = w_{ij}^{-} - \eta \frac{\partial D(j)}{\partial w_{ij}}\bigg|_{w_{ij}^{-}} \quad i\epsilon\{1,2,\ldots,n\}$$

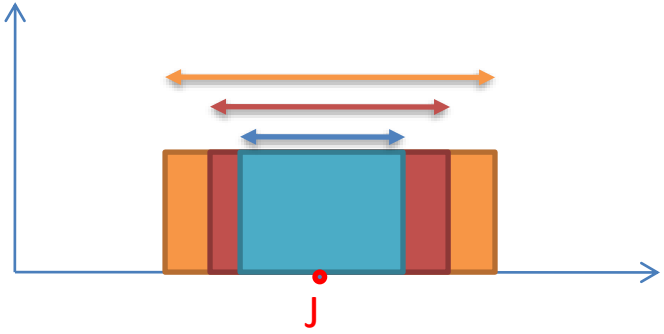
$$\sigma_{hkJ}^{+} = \sigma_{hkJ}^{-} - \eta \frac{\partial D(j)}{\partial \sigma_{hkJ}}\bigg|_{\sigma_{hkJ}^{-}} \quad h,k\epsilon\{1,2,\ldots,n\}$$



Extensions of SOM

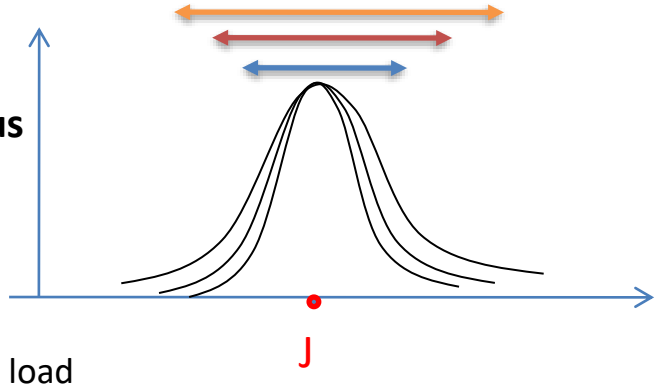
For Fuzzy version

Adjacent Radius



Cooperative neurons (Hard)

Soft Adjacent Radius



Cooperative neurons (Soft)

More Computational load
With better accuracy in clustering

Other mechanism for clustering

- **LVQ1,2: Learning Vector Quantization** (Kohonen,.. 1989,90)

Some supervised learning mechanisms for classification purposes, based on distance concept

- **Counter-propagation Network** (Hecht-Nielson 1987,88)

Some mechanisms to compress data.

- **ART: Adaptive Resonance Theory** (Capenter-Grossberg 1987)

Some mechanisms to cluster data based on weighted distance concept .

Generative Adversarial Network (GAN)

Introductory guide to Generative Adversarial Networks (GANs) and their promise!

Faizan Shaikh, June 15, 2017

• Introduction

Neural Networks have made great progress.

1. They now recognize images and voice at levels comparable to humans.
2. They are also able to understand natural language with a good accuracy.

Let us see a few examples where we need human creativity (at least as of now):

•Train an artificial author which can write an article and explain data science concepts to a community in a very simplistic manner by learning from past articles on Analytics Vidhya.

• You are not able to buy a painting from a famous painter which might be too expensive. Can you create an artificial painter which can paint like any famous artist by learning from his / her past collections?

GANs: A mechanism to generate the desired patterns.

It seems we should develop a mechanism in order to generate fake patterns which are so similar to real patterns.

Ian Goodfellow (in 2014) introduce GANs for such purposes.

- Yann LeCun, a prominent figure in Deep Learning Domain said in his Quora session that:

"(GANs), and the variations that are now being proposed is the most interesting idea in the last 10 years in ML, in my opinion."

But what is a GAN?

Let us take an analogy to explain the concept:

If you want to get better at something, say chess; what would you do? You would compete with an opponent better than you. Then you would analyze what you did wrong, what he / she did right, and think on what could you do to beat him / her in the next game.

You would repeat this step until you defeat the opponent. This concept can be incorporated to build better models. So simply, for getting a powerful hero (viz generator), we need a more powerful opponent (viz discriminator)!

Another analogy from real life

A slightly more real analogy can be considered as a relation between forger and an investigator.

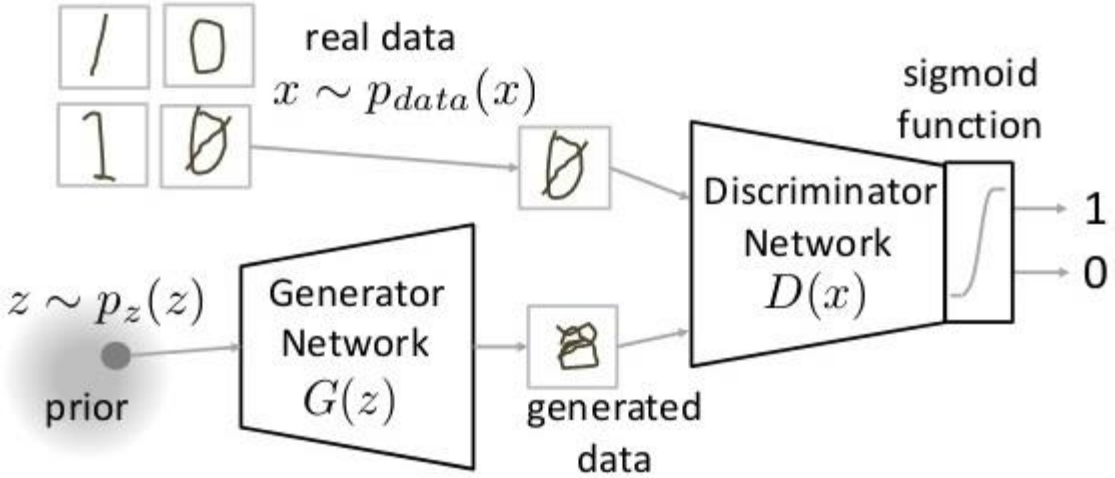
The task of a forger is to create fraudulent imitations of original paintings by famous artists. If this created piece can pass as the original one, the forger gets a lot of money in exchange of the piece.

On the other hand, an art investigator's task is to catch these forgers who create the fraudulent pieces. How does he do it? He knows what are the properties which sets the original artist apart and what kind of painting he should have created. He evaluates this knowledge with the piece in hand to check if it is real or not.

This contest of forger vs investigator goes on, which ultimately makes world class investigators (and unfortunately world class forger); a battle between good and evil.

How do GANs work?

As we saw, there are two main components of a GAN – Generator Neural Network and Discriminator Neural Network.



About the above Figure:

1. The Generator Network takes an random input and tries to generate a sample of data.

- In the above image, we can see that generator G(z) takes a input z from p(z), where z is a sample from probability distribution p(z).

2. It then generates a data which is then fed into a discriminator network D(x).
3. The task of Discriminator Network is to take input either from the real data or from the generator and try to predict whether the input is real or generated.
   It takes an input x from $p_{data}(x)$ where $p_{data}(x)$ is our real data distribution. D(x) then solves a binary classification problem using sigmoid function giving output in the range 0 to 1.

- Let us define the notations we will be using to formalize our GAN,
- Pdata(x) -> the distribution of real data
  X -> sample from pdata(x)
  P(z) -> distribution of generator
  Z -> sample from p(z)
  G(z) -> Generator Network
  D(x) -> Discriminator Network

Now the training of GAN is done (as we saw above) as a **"fight between generator and discriminator."**

This can be represented mathematically as:

$$\min_G \max_D V(D,G)$$

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

•In our function V(D, G) the first term is entropy that the data from real distribution (pdata(x)) passes through the discriminator (also known as (aka) best case scenario).

The discriminator tries to maximize this to 1.

The second term is entropy that the data from random input (p(z)) passes through the generator, which then generates a fake sample which is then passed through the discriminator to identify the fakeness (aka worst case scenario).

In this term, discriminator tries to maximize it to 0

(i.e. the log probability that the data from generated is fake is equal to 0).

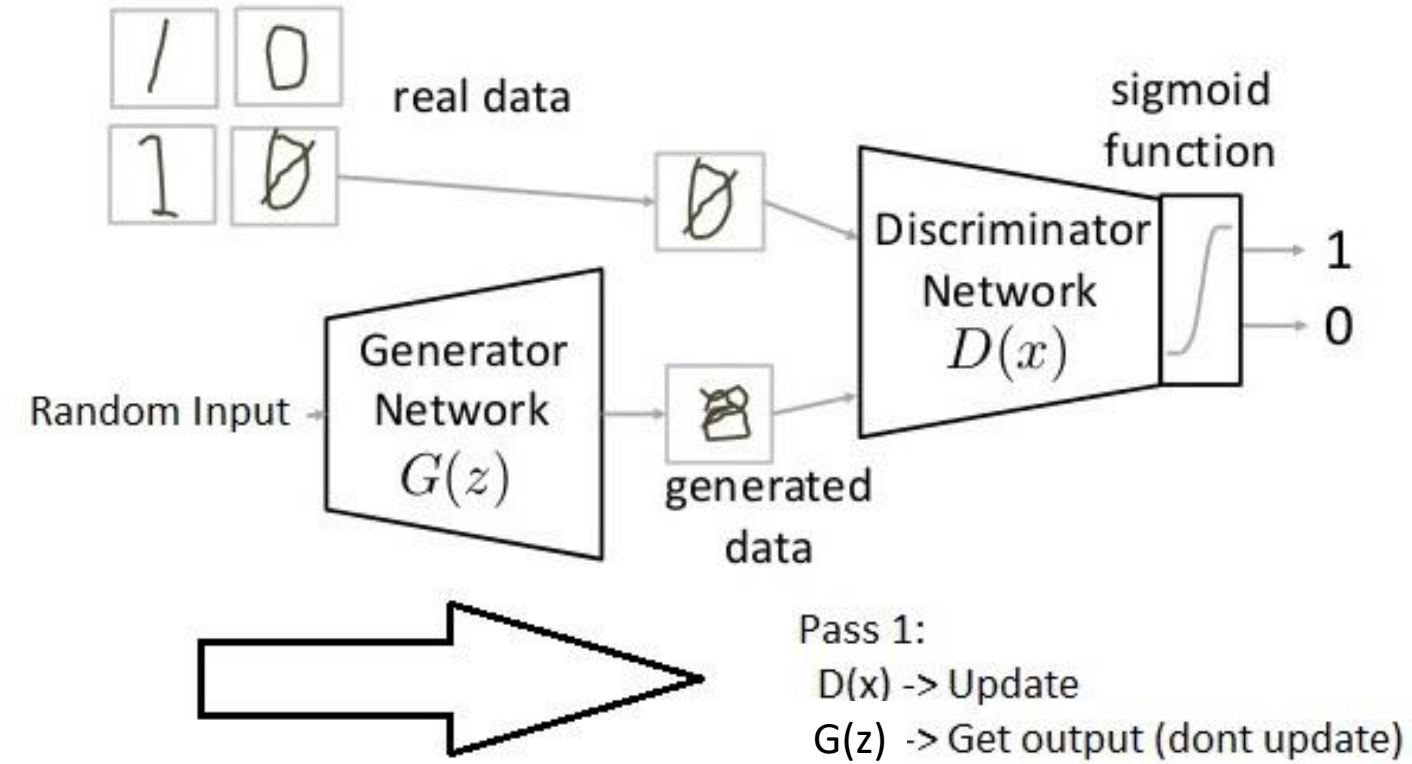
of training a GAN is taken from game theory called the minimax game.

So overall, the discriminator is trying to maximize our function V. On the other hand, **the task of generator is exactly opposite, i.e. it tries to minimize the function V** so that the differentiation between real and fake data is bare minimum.

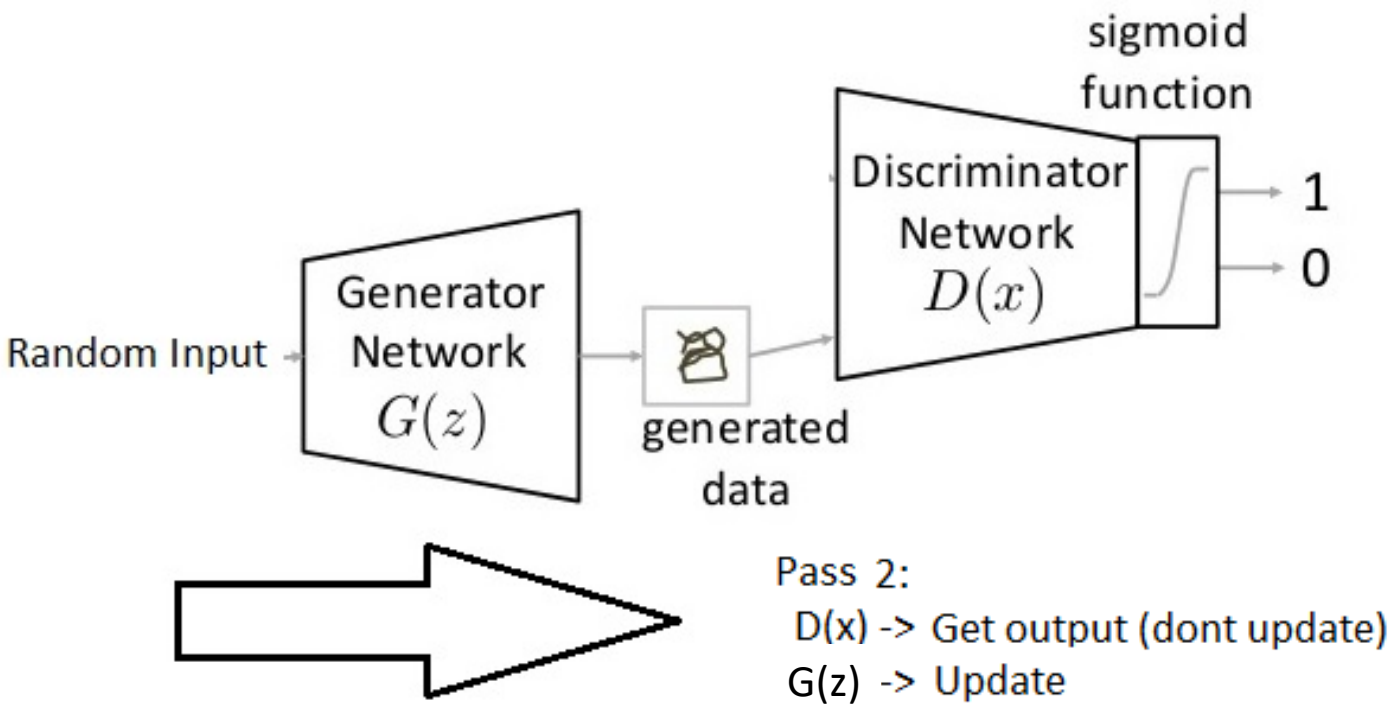
This, in other words is a cat and mouse game between generator and discriminator!

Note: This method of training a GAN is taken from game theory called the minimax game.

Pass 1: Train discriminator and freeze generator (freezing means setting training as false. The network does only forward pass and no backpropagation is applied).



Pass 2: Train generator and freeze discriminator.



Steps to train a GAN

- **Step 1: Define the problem.** Do you want to generate fake images or fake text. Here you should completely define the problem and collect data for it.
- **Step 2: Define architecture of GAN.** Define how your GAN should look like. Should both your generator and discriminator be multi layer perceptrons, or convolutional neural networks? This step will depend on what problem you are trying to solve.
- **Step 3: Train Discriminator on real data for n epochs.** Get the data you want to generate fake on and train the discriminator to correctly predict them as real. Here value n can be any natural number between 1 and infinity.
- **Step 4: Generate fake inputs for generator and train discriminator on fake data.** Get generated data and let the discriminator correctly predict them as fake.
- **Step 5: Train generator with the output of discriminator.** Now when the discriminator is trained, you can get its predictions and use it as an objective for training the generator. Train the generator to fool the discriminator.
- **Steps to train a GAN**
- **Step 6: Repeat step 3 to step 5 for a few epochs.**
- **Step 7: Check if the fake data manually if it seems legit. If it seems appropriate, stop training, else go to step 3.** This is a bit of a manual task, as hand evaluating the data is the best way to check the fakeness. When this step is over, you can evaluate whether the GAN is performing well enough.

A pseudocode of GAN training can be thought out as follows

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

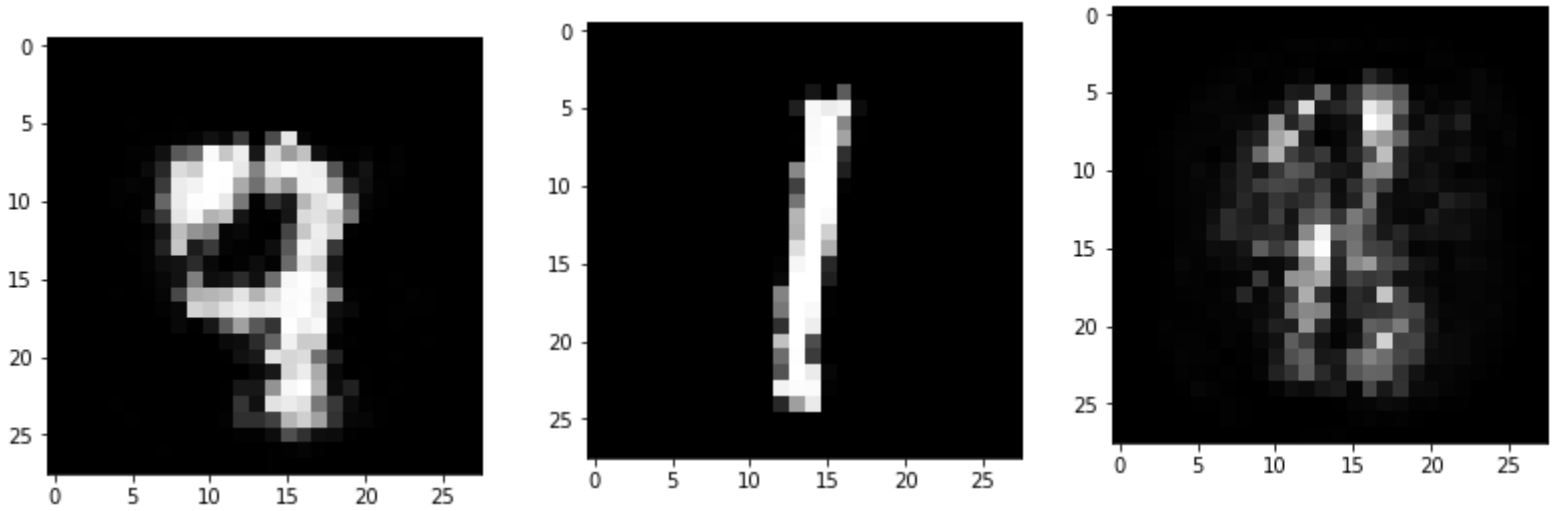
  - Sample minibatch of m noise samples $\{\boldsymbol{z}^{(1)}, \ldots, \boldsymbol{z}^{(m)}\}$ from noise prior $p_g(\boldsymbol{z})$.
  - Update the generator by descending its stochastic gradient:

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

end for

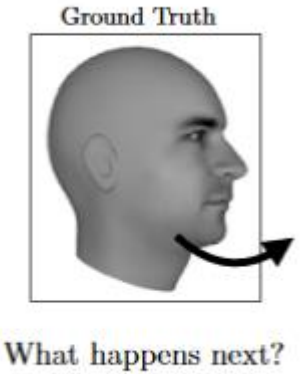
We will try to generate digits by training a GAN

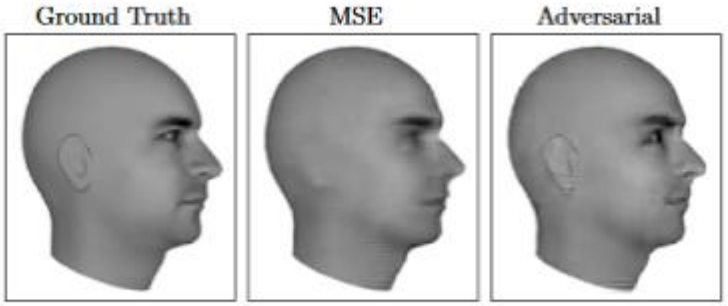
After training for 100 epochs, I got the following generated images



Applications of GAN

Predicting the next frame in a video : You train a GAN on video sequences and let it predict what would occur next





Increasing Resolution of an image : Generate a high resolution photo from a comparatively low resolution.

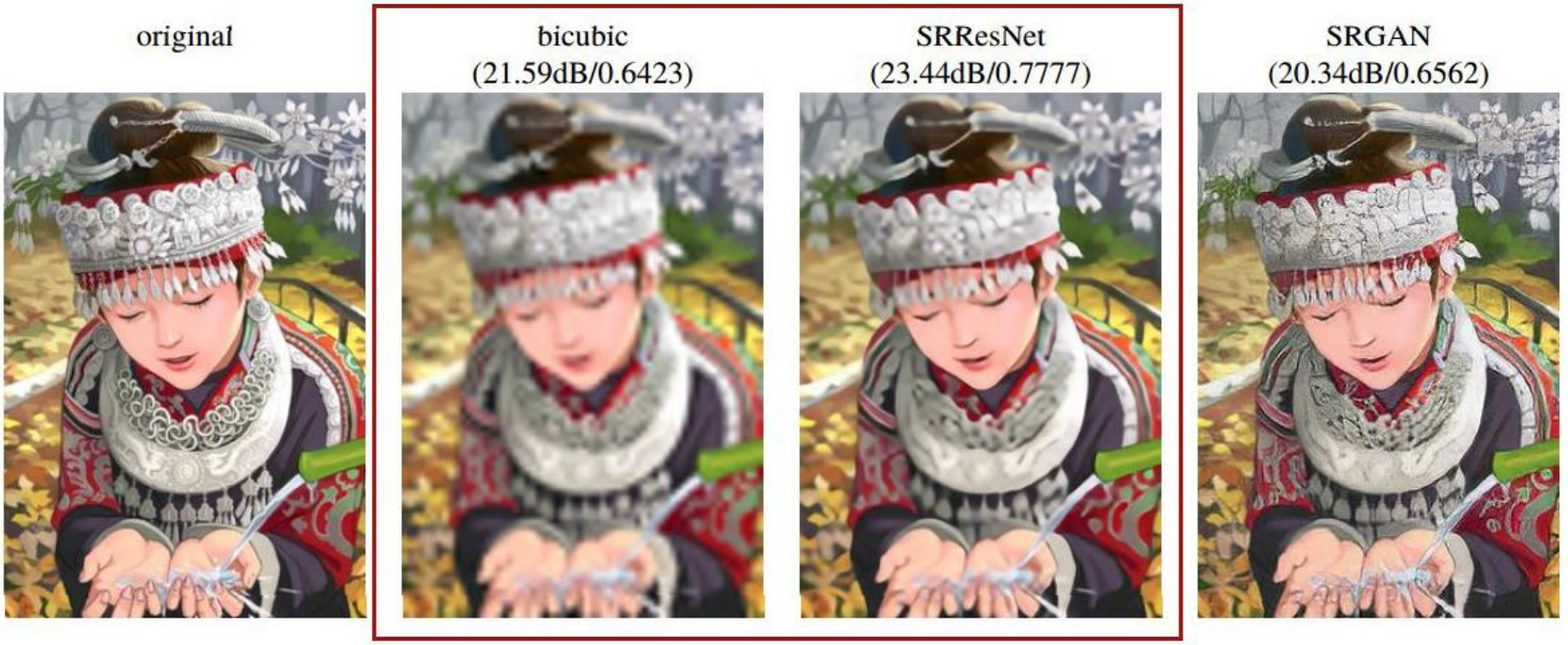
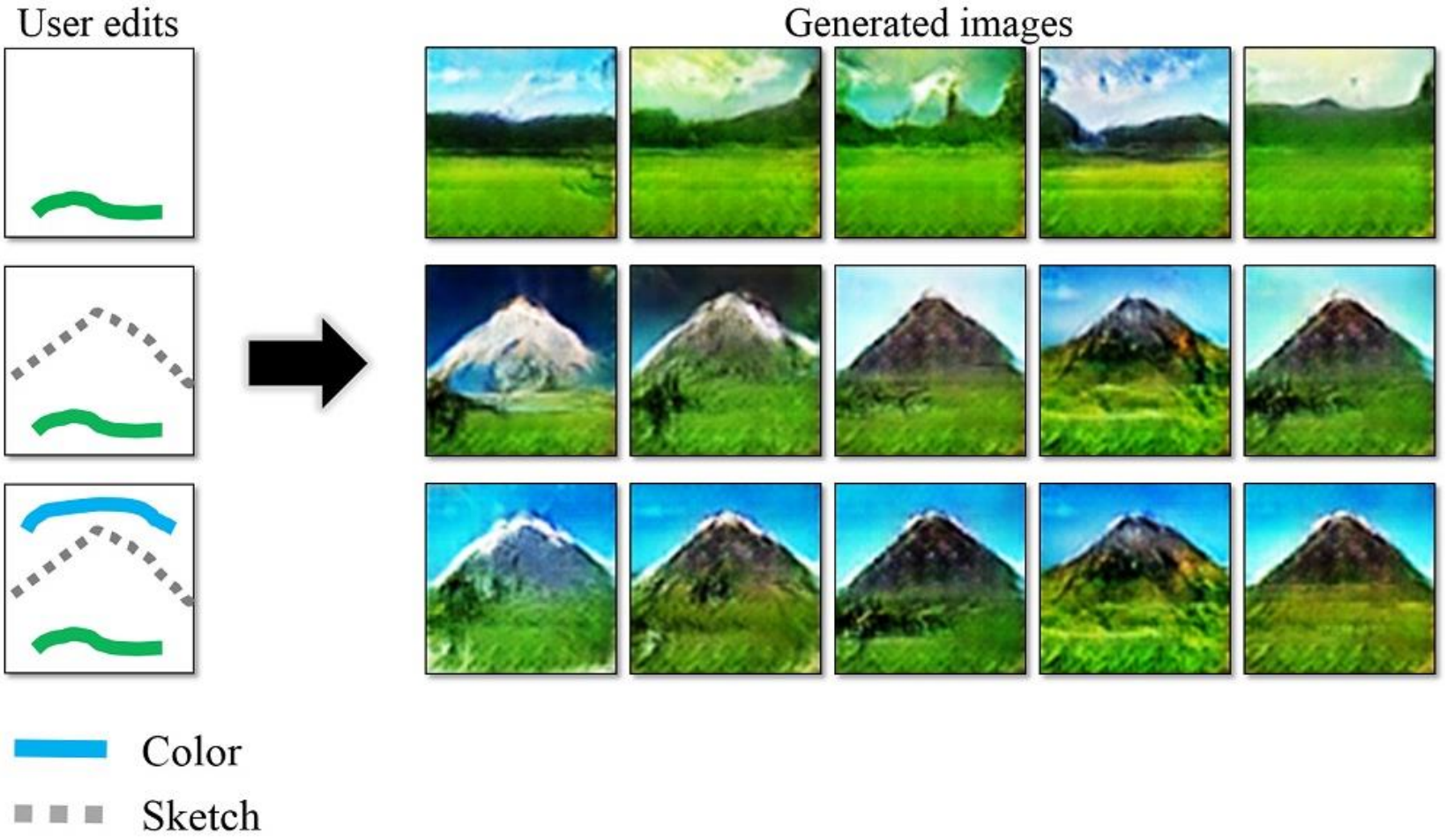


Image to Image Translation : Generate an image from another image. For example, given on the left, you have labels of a street scene and you can generate a real looking photo with GAN. On the right, you give a simple drawing of a handbag and you get a real looking drawing of a handbag.

Text to Image Generation : Just say to your GAN what you want to see and get a realistic photo of the

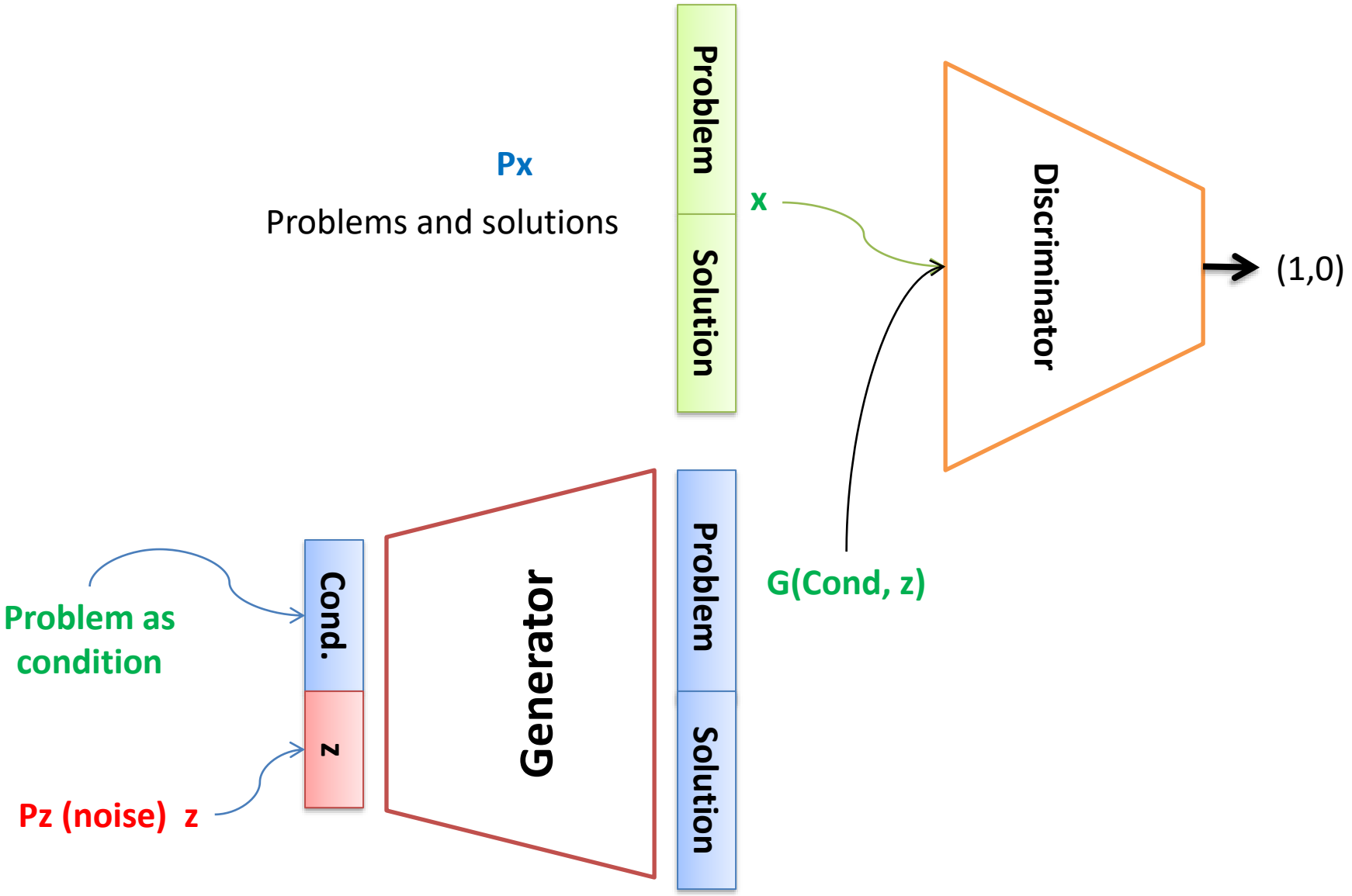
This bird has a yellow belly and tarsus, grey back, wings, and brown throat, nape with a black face

Interactive Image Generation(IGAN) : Draw simple strokes and let the GAN draw an impressive picture for you



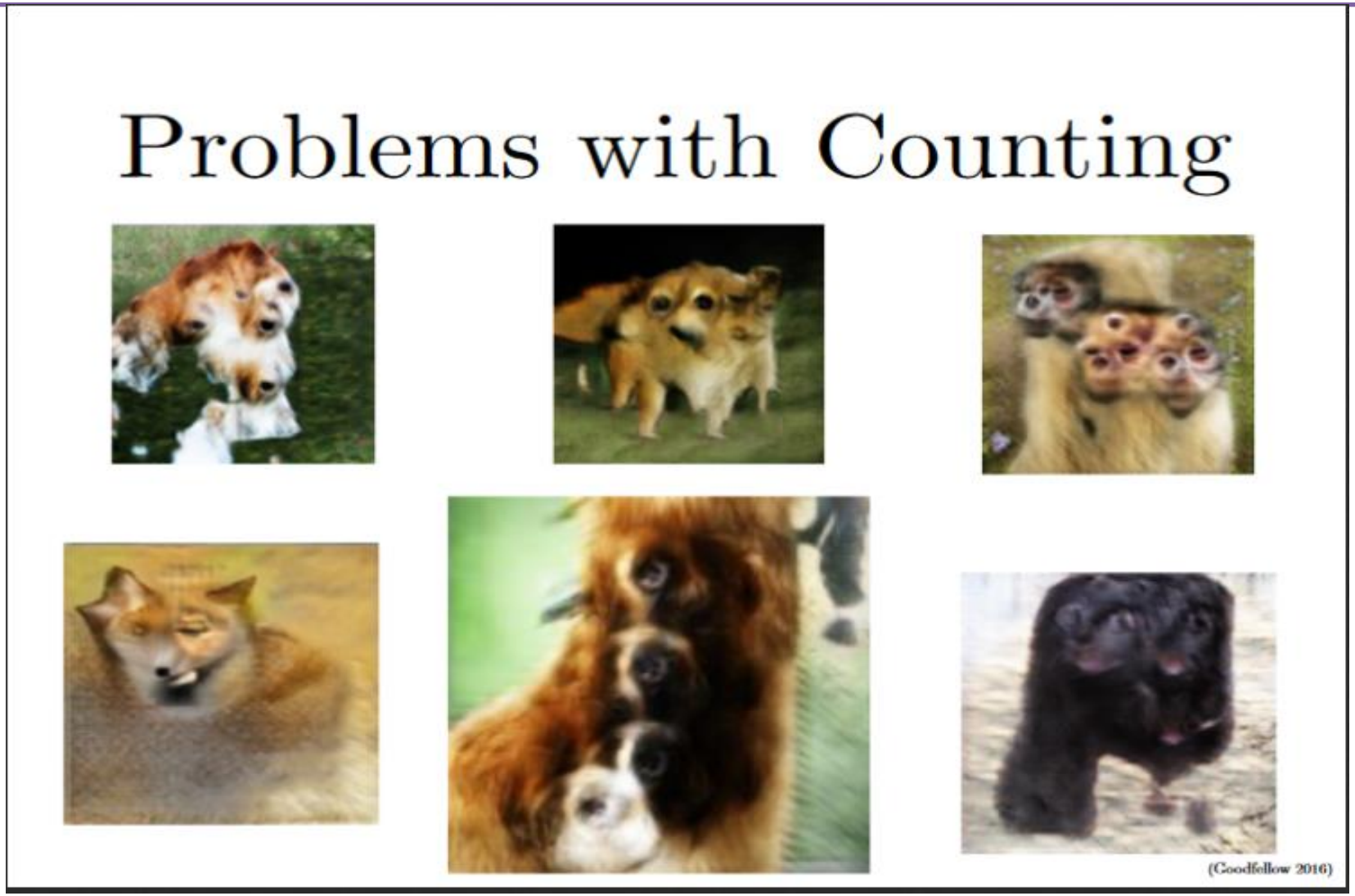
A General Plan

to generate **solutions** for challenging problems

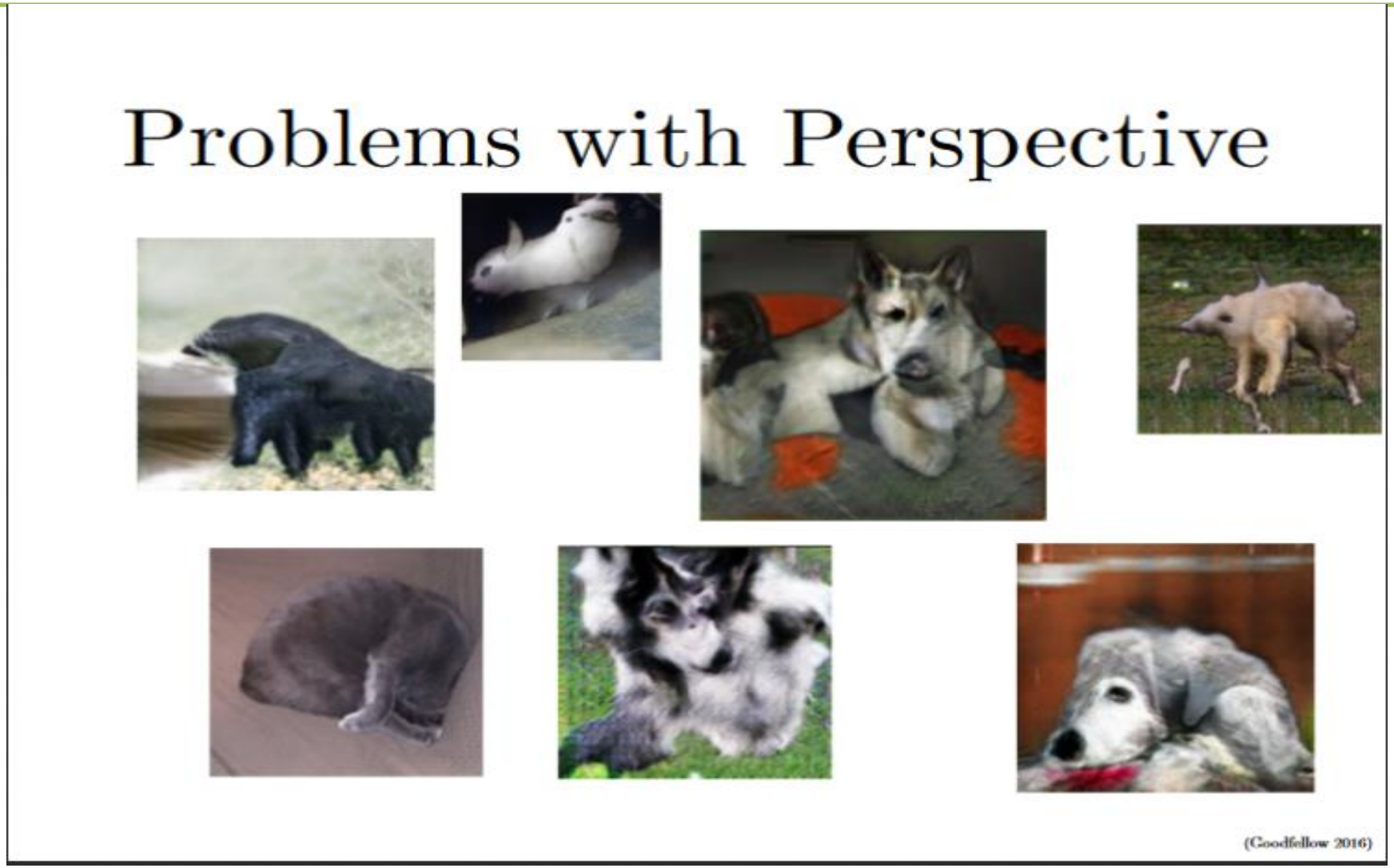


Challenges With GAN

Problem with Counting: GANs fail to differentiate how many of a particular object should occur at a location. As we can see below, it gives more number of eyes in the head than naturally present.



GANs fail to adapt to 3D objects. It doesn't understand perspective, i.e.difference between frontview and backview. As we can see below, it gives flat (2D)



Problems with Global Structures

Same as the problem with perspective, GANs do not understand a holistic structure. For example, in the bottom left image, it gives a generated image of a quadruple cow, i.e. a cow standing on its hind legs and simultaneously on all four legs. That is definitely not possible in real life!

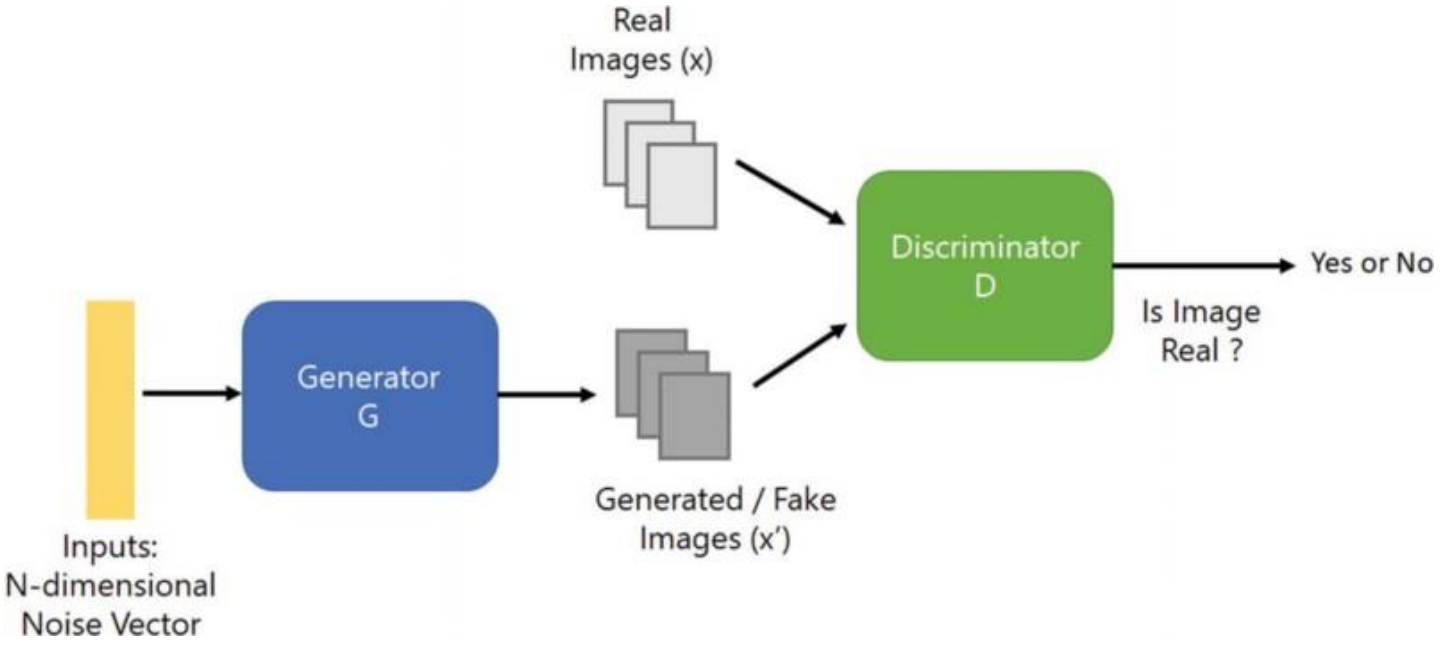
Some Extensions on GANs

1. DCGANs
2. BGAN
3. SRGAN
4. CGAN
5. Auxiliary Classifier GAN
6. Semi-Supervised GAN
7. StackGAN
8. Disco GANS
9. Flow based GANs
10. InfoGANs
11. Wasserstein GAN
12. Bidirectional GAN
13. Context-Conditional GAN
14. Context Encoder
15. Coupled GANs
16. CycleGAN
17. DualGAN
18. LSGAN
19. Pix2Pix
20. PixelDA
21. Wasserstein GAN GP
22. Adversarial Autoencoder

(1) Deep Convolutional GANs (DCGANs)

https://arxiv.org/abs/1710.10196 **(Jan 2016)**

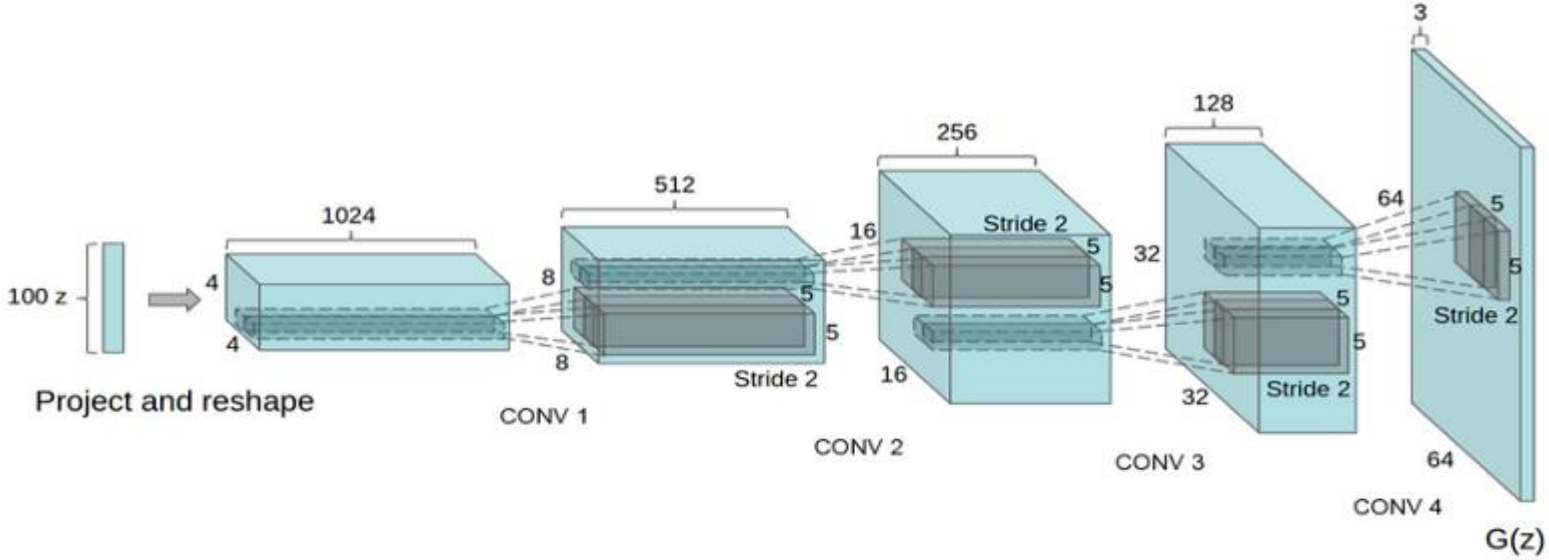
In this article, we will see how a neural net maps from random noise to an image matrix and how using Convolutional Layers in the generator network produces better results.



Original DCGAN architecture(*Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks*) have *four* convolutional layers for the **Discriminator** and *four* "four fractionally-strided convolutions" layers for the **Generator.**

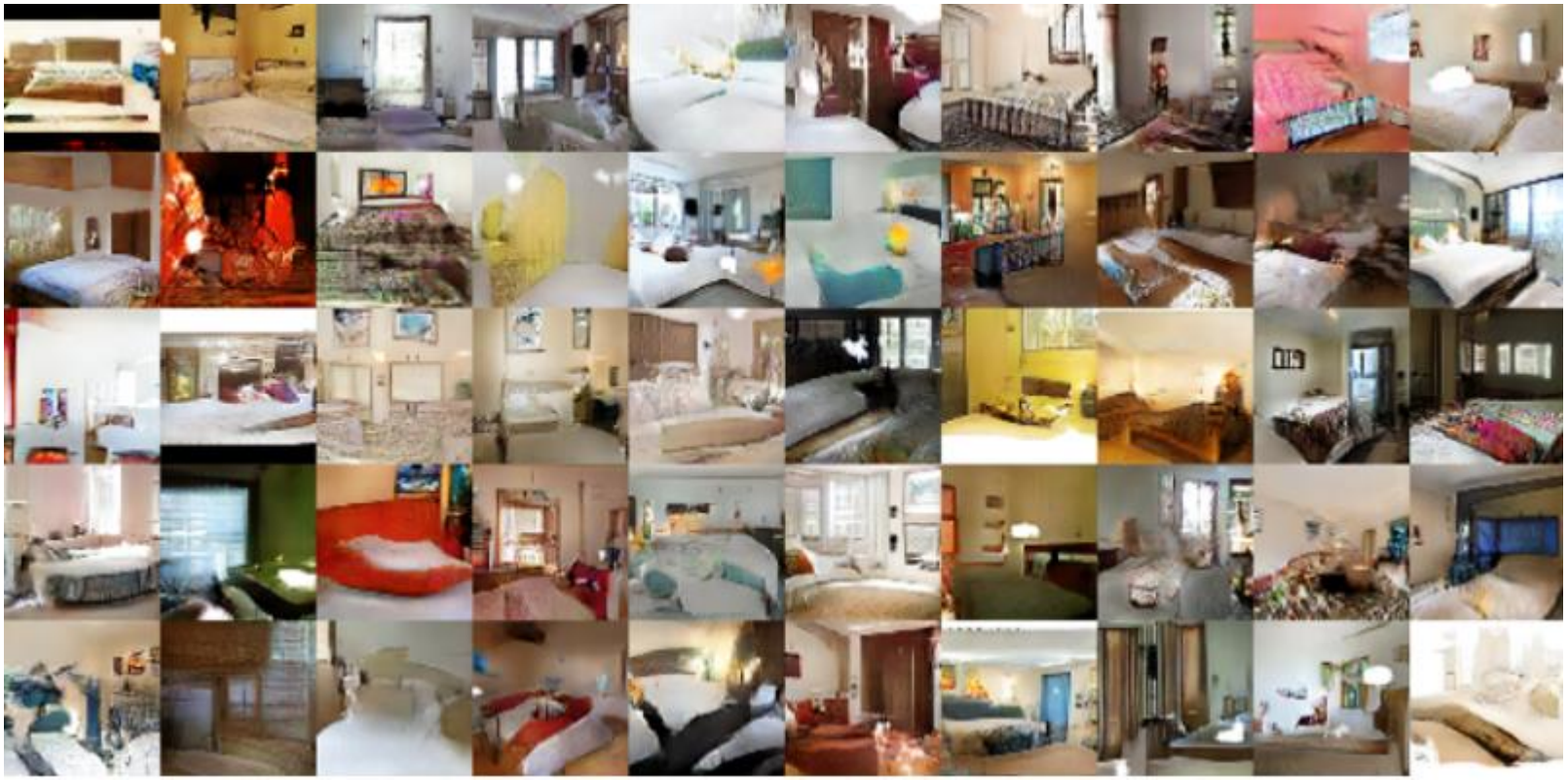
(DCGAN) Generator

This network takes in a 100x1 noise vector, denoted z, and maps it into the G(Z) output which is 64x64x3. This architecture is especially interesting the way the first layer expands the random noise. The network goes from 100x1 to 1024x4x4! This layer is denoted 'project and reshape'. We see that following this layer, classical convolutional layers are applied. In the diagram above we can see that the N parameter, (Height/Width), goes from 4 to 8 to 16 to 32, it doesn't appear that there is any padding, the kernel filter parameter F is 5x5, and the stride is 2. You may find this equation to be useful for designing your own convolutional layers for customized output sizes.



we see the network goes from

100x1 → 1024x4x4 → 512x8x8 → 256x16x16 → 128x32x32 → 64x64x3



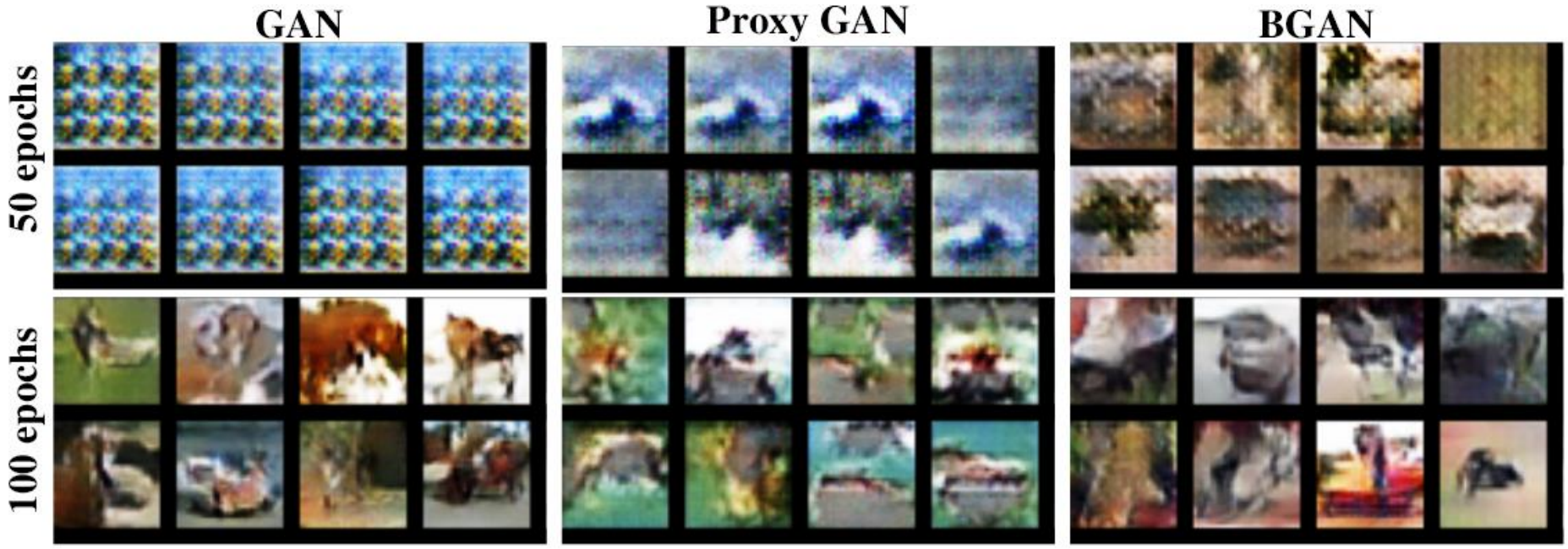
Above is the output from the network presented in the paper, citing that this came after 5 epochs of training. Pretty impressive stuff.

(2) Boundary-Seeking Generative Adversarial Networks(BGAN)

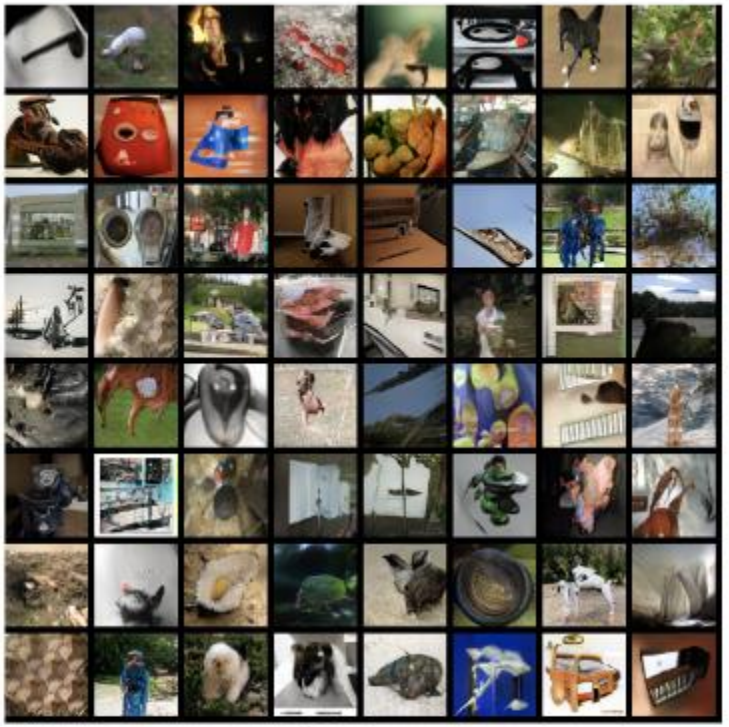
https://arxiv.org/abs/1702.08431 (2017)

- We introduce a method for training GANs with discrete data that uses the estimated difference measure from the discriminator to compute importance weights for generated samples, thus providing a policy gradient for training the generator.
- The importance weights have a strong connection to the decision boundary of the discriminator, and we call our method boundary-seeking GANs (BGANs).
- We demonstrate the effectiveness of the proposed algorithm with discrete image and character-based natural language generation.
- In addition, the boundary-seeking objective extends to continuous data, which can be used to improve stability of training, and we demonstrate this on Celeba, Large-scale Scene Understanding (LSUN) bedrooms, and Imagenet without conditioning.

Training a GAN with different generator loss functions and 5 updates for the generator for every update of the discriminator.
Over-optimizing the generator can lead to instability and poorer results depending on the generator objective function.
Samples for GAN and GAN with the proxy loss are quite poor at 50 discriminator epochs (250 generator epochs), while BGAN is noticeably better.
At 100 epochs, these models have improved, though are still considerably behind BGAN.



CelebA



Imagenet

LSUN

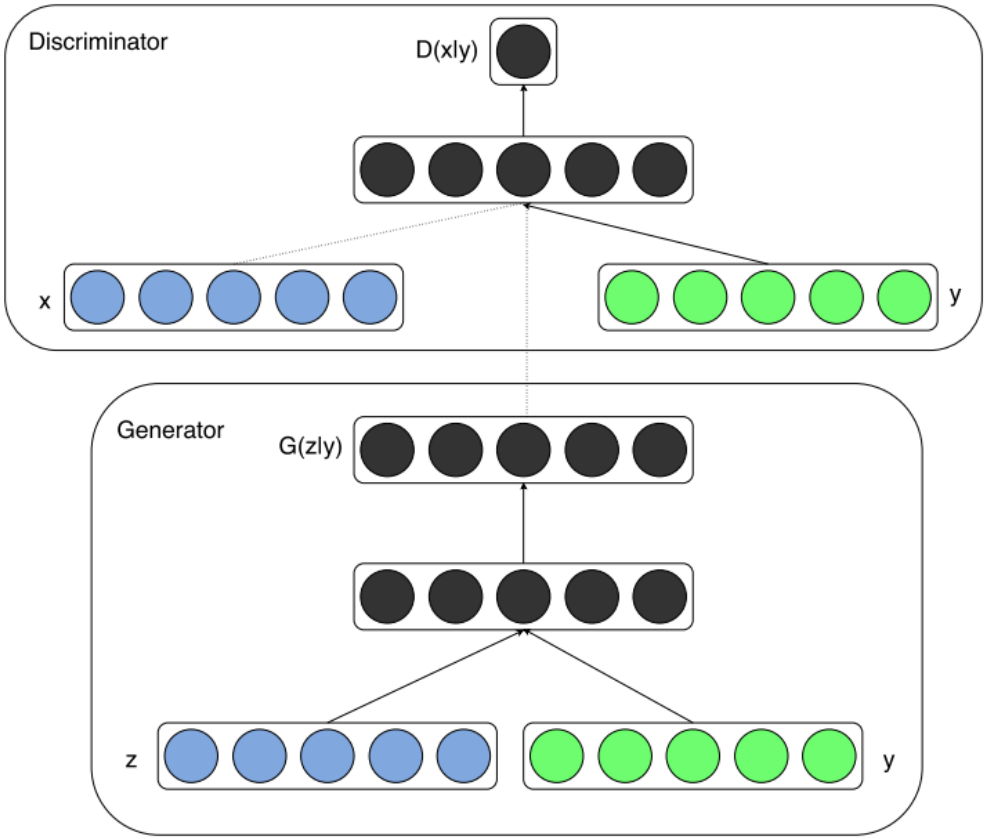
Figure 3: Highly realistic samples from a generator trained with BGAN on the CelebA and LSUN datasets. These models were trained using a deep ResNet architecture with gradient norm regularization (Roth et al., 2017). The Imagenet model was trained on the full 1000 label dataset without conditioning.

(3) Conditional GANs (cGANs)

Mehdi Mirza, Simon Osindero

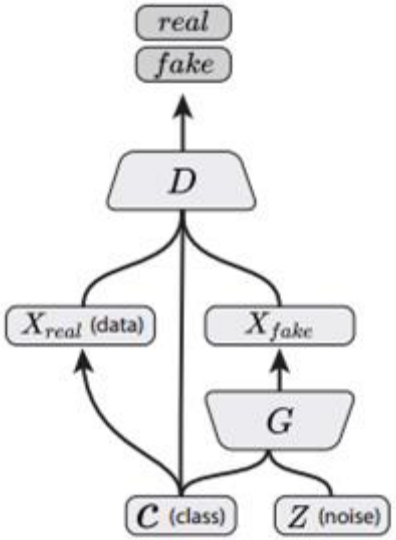
(Submitted on 6 Nov 2014)

Source: https://arxiv.org/pdf/1411.1784.pdf



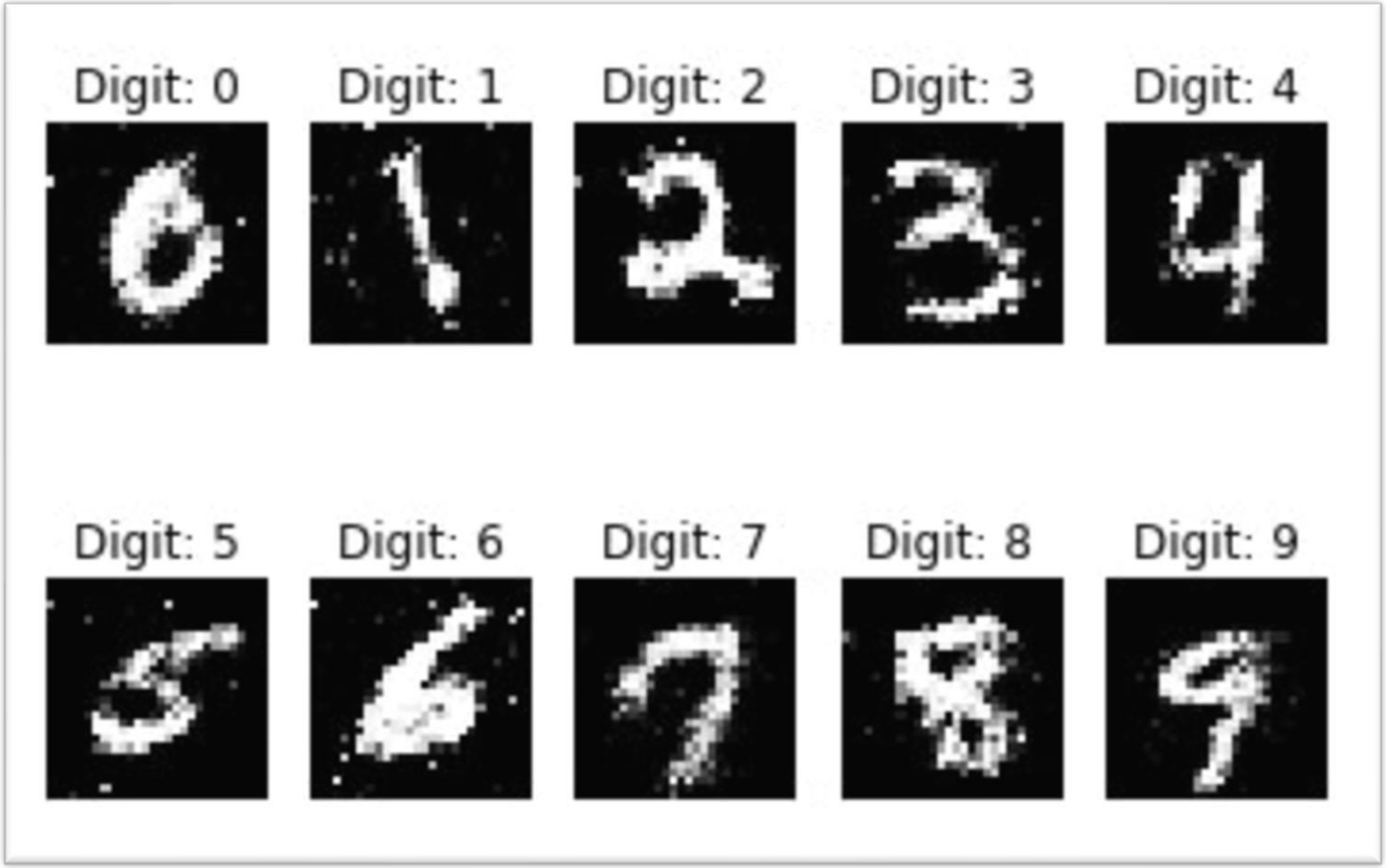
1. These GANs use extra label information and result in better quality images and are able to control how generated images will look. cGANs learn to produce better images by exploiting the information fed to the model.

2. it does give the end-user a mechanism for controlling the Generator output



Conditional GAN
(Mirza & Osindero, 2014)

The results of Conditional GANs are very impressive. They allow for much greater control over the final output from the generator



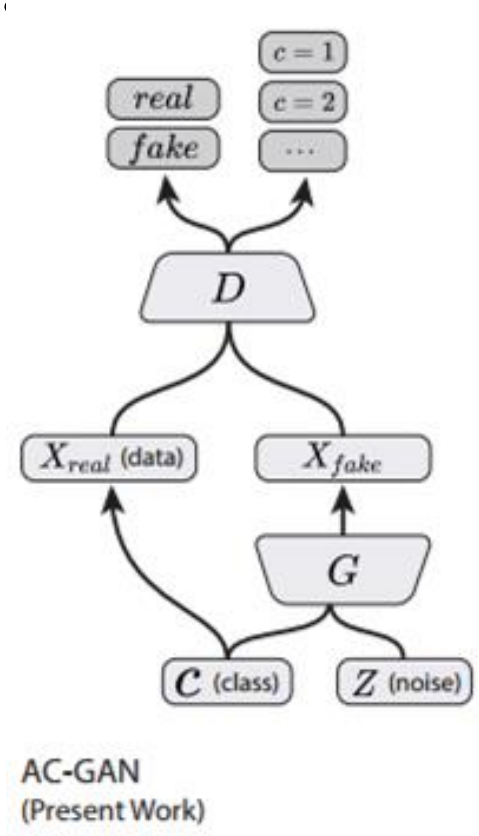
Results testing Conditional GANs on MNIST

AC-GAN

Auxiliary Classifier GANs

By adding an auxiliary classifier to the discriminator of a GAN, the discriminator produces not only a probability distribution over sources but also probability distribution over the class labels.

Source: Augustus Odena, Christopher Olah, Jonathon Shlens. Conditional Image Synthesis with Auxiliary Classifier GANs. 2016.



AC-GAN
(Present Work)

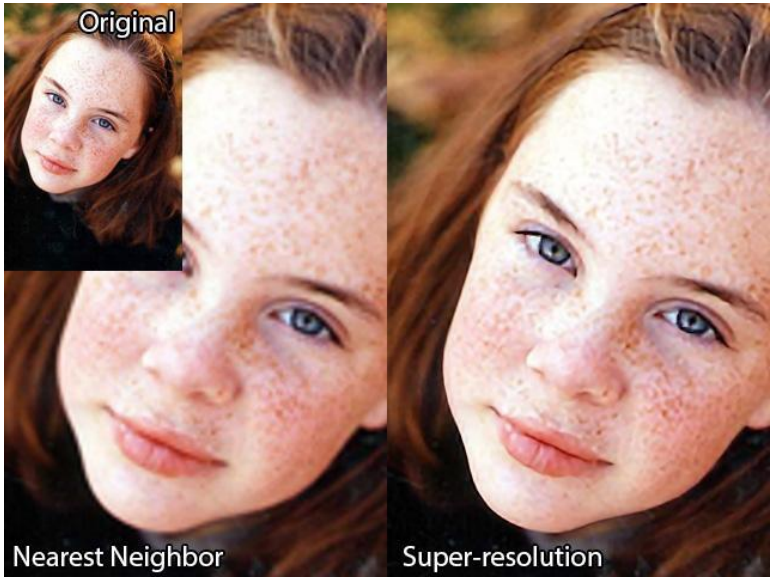
The sample generated images from CIFAR-10 dataset.

In the AC-GAN paper, 100 different GAN models each handle 10 different classes from the ImageNet dataset consisting of 1,000 different object categories

The sample generated images from ImageNet dataset.

(4) Super Resolution GAN (SR GAN)

- Super-resolution is a task concerned with upscaling images from low-resolution sizes such as 90 x 90, into high-resolution sizes such as 360 x 360.



Source: Photo-Realistic Single Image Super-Resolution Using a Generative Adversarial Network.
Christian Ledig, Lucas Theis, Ferenc Huszar, Jose Caballero, Andrew Cunningham, Alejandro Acosta, Andrew Aitken, Alykhan Tejani, Johannes Totz, Zehan Wang, Wenzhe Shi.

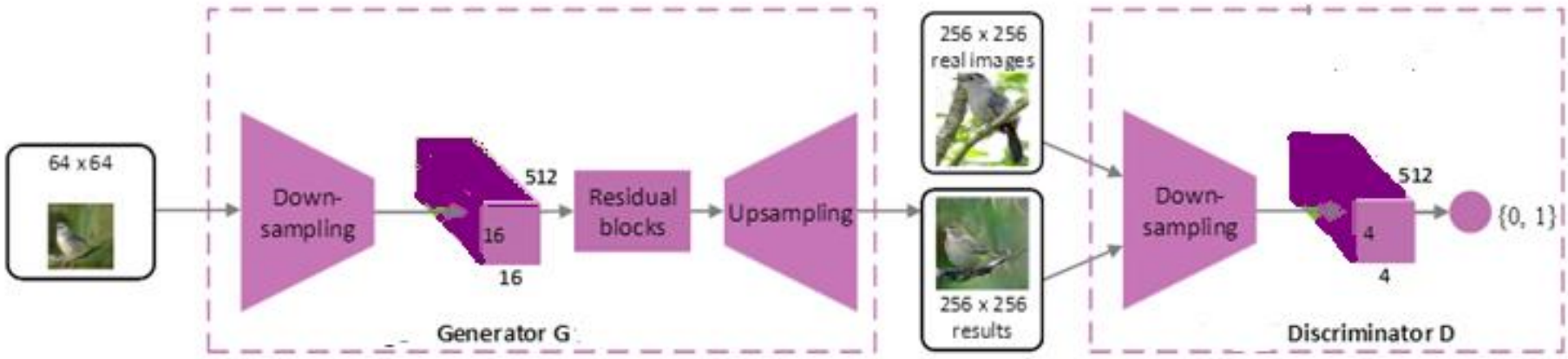
4× SRGAN (proposed)

original

In this example, 90 x 90 to 360 x 360 is denoted as an up-scaling factor of 4x.

•These networks learn a mapping from the low-resolution patch through a series of convolutional, fully-connected, or transposed convolutional layers into the high-resolution patch.

For example, this network could take a 64x 64low-resolution patch, convolve over it a couple times such that the feature map is something like 16 x 16 x 512, flatten it into a vector, apply a couple of fully-connected layers, reshape it, and finally, up-sample it into a 256x 256 high-resolution patch through transposed convolutional layers.



(5) StackGAN

Text to Photo-realistic Image Synthesis with Stacked generative Adversarial Networks, Han Zhang, ...

- The authors of this paper propose a solution to the problem of synthesizing high-quality images from text descriptions in computer vision. They propose Stacked Generative Adversarial Networks (StackGAN) to generate 256x256 photo-realistic images conditioned on text descriptions. They decompose the hard problem into more manageable sub-problems through a sketch-refinement process.

The Stage-I GAN sketches the primitive shape and colors of the object based on the given text description, yielding Stage-I low-resolution images. The Stage-II GAN takes Stage-I results and text descriptions as inputs, and generates high-resolution images with photo-realistic details.

The architecture of the proposed StackGAN.

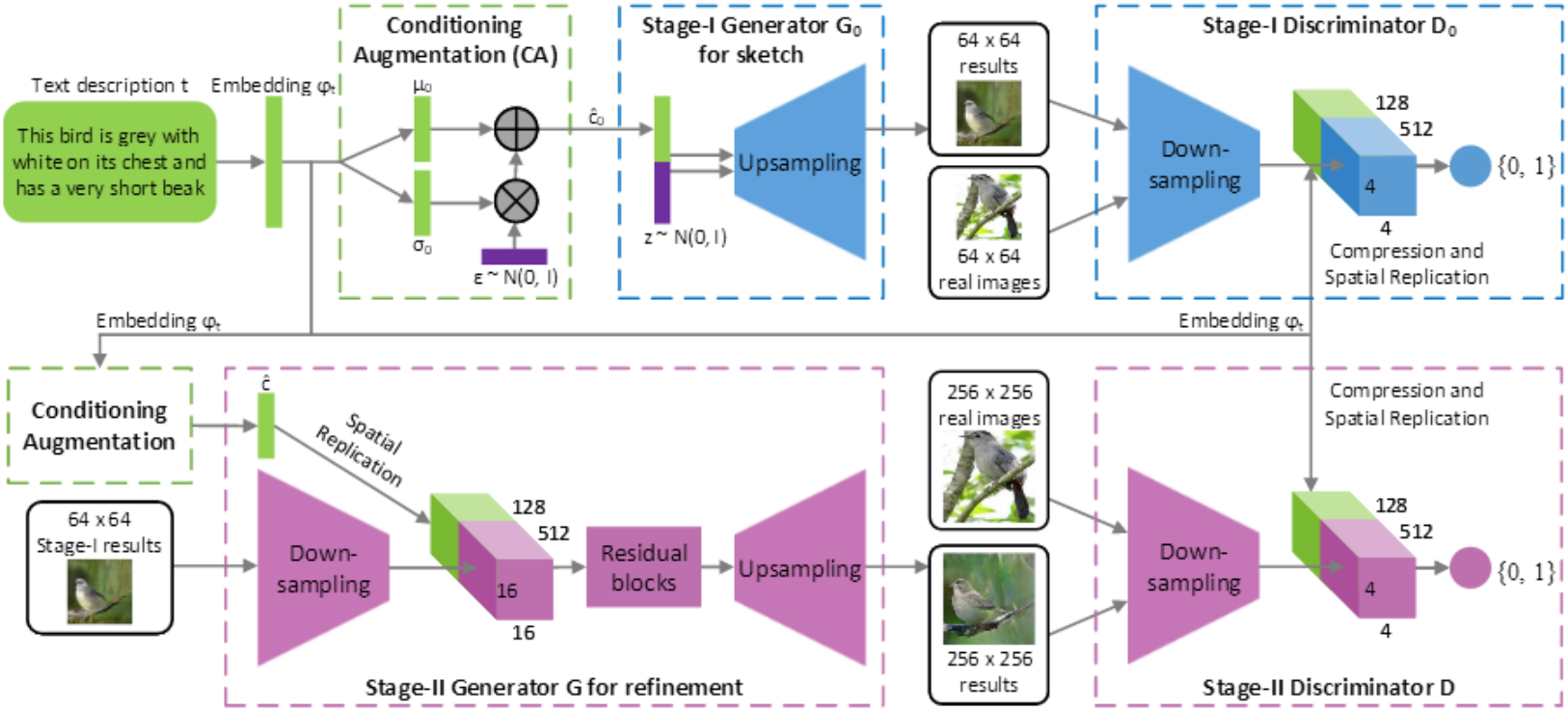


Figure 2. The architecture of the proposed StackGAN. The Stage-I generator draws a low-resolution image by sketching rough shape and basic colors of the object from the given text and painting the background from a random noise vector. Conditioned on Stage-I results, the Stage-II generator corrects defects and adds compelling details into Stage-I results, yielding a more realistic high-resolution image.

Comparison

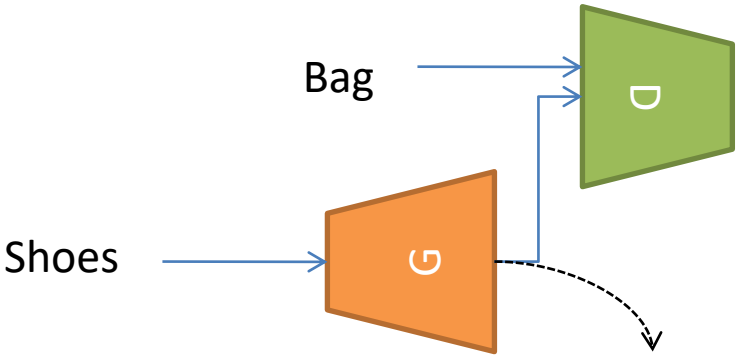
Figure 3. Example results by our StackGAN, GAWWN [24], and GAN-INT-CLS [26] conditioned on text descriptions from CUB test set.

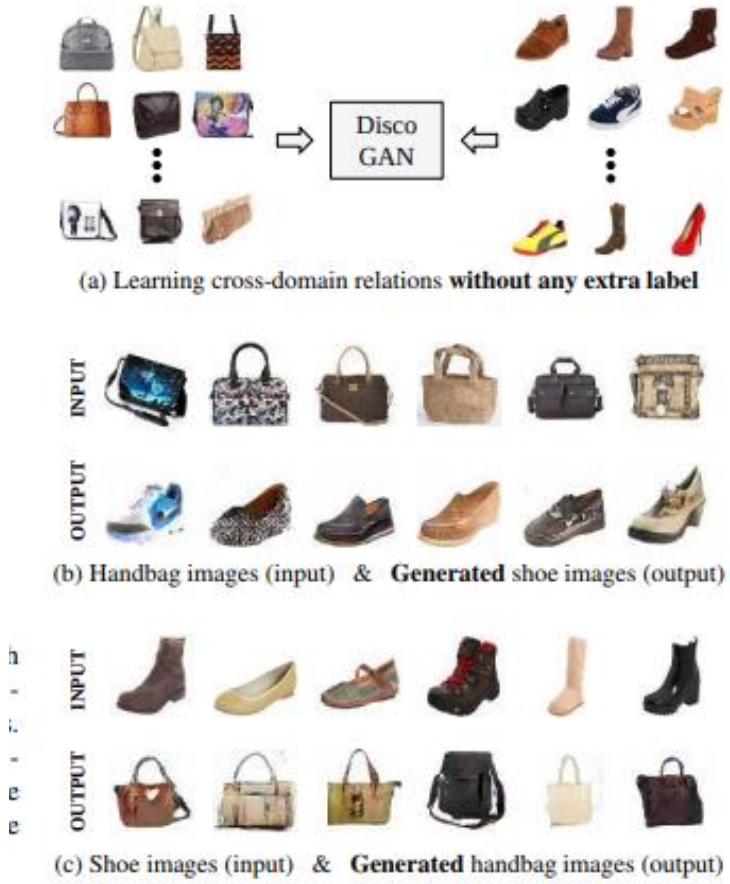
Comparison

Figure 4. Example results by our StackGAN and GAN-INT-CLS [26] conditioned on text descriptions from Oxford-102 test set (leftmost four columns) and COCO validation set (rightmost four columns).

(6) Discover Cross-Domain Relations with GANs(Disco GANS)

- The authors of this paper propose a method based on generative adversarial networks that learns to discover relations between different domains (without any extra labels). Using the discovered relations, the network transfers style from one domain to another..





(7) Flow-Based GANs

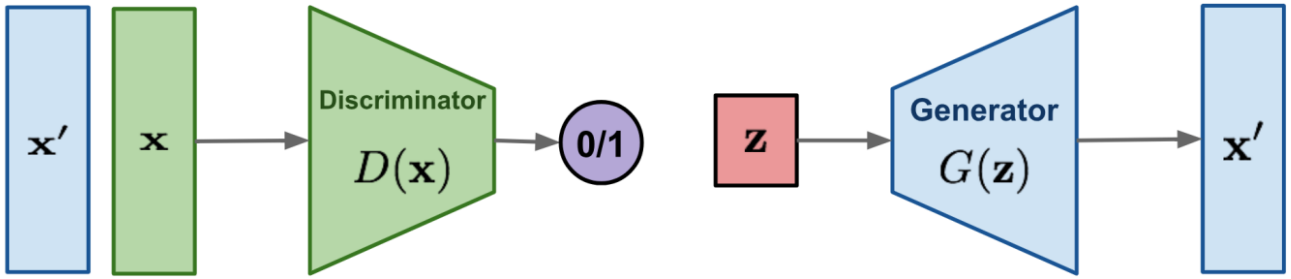
Oct 13, 2018 by Lilian Weng

Here is a quick summary of the difference between GAN, VAE, and flow-based generative models:

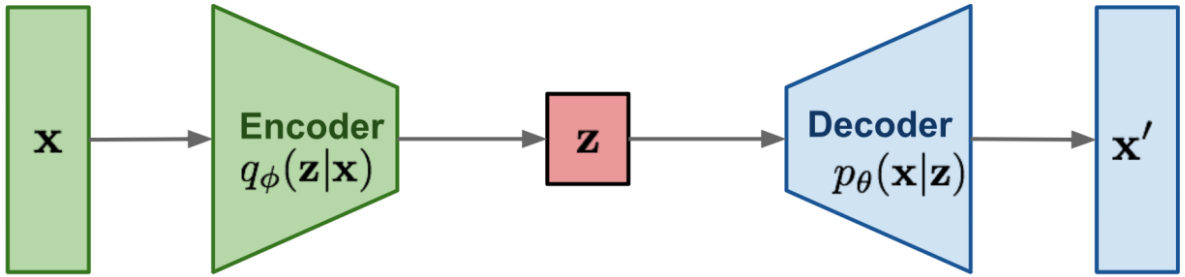
1. Generative adversarial networks: GAN provides a smart solution to model the data generation, an unsupervised learning problem, as a supervised one. The discriminator model learns to distinguish the real data from the fake samples that are produced by the generator model. Two models are trained as they are playing a minimax game.
2. Variational autoencoders: VAE inexplicitly optimizes the log-likelihood of the data by maximizing the evidence lower bound (ELBO).
3. Flow-based generative models: A flow-based generative model is constructed by a sequence of invertible transformations. Unlike other two, the model explicitly learns the data distribution $p(\mathbf{x})$ and therefore the loss function is simply the negative log-likelihood.

Types of Generative Models

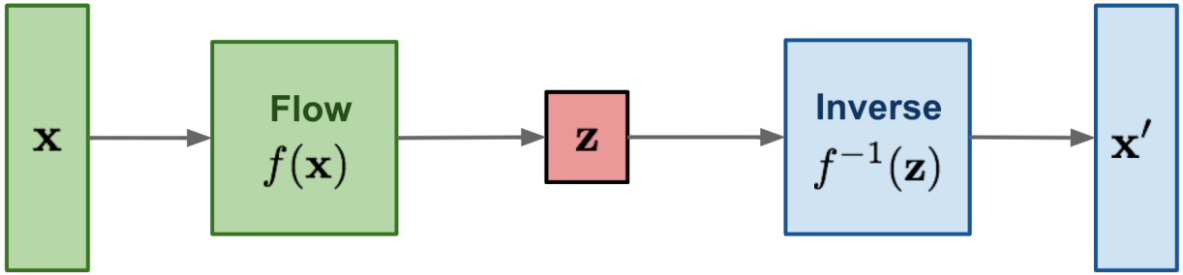
GAN: minimax the classification error loss.



VAE: maximize ELBO.



Flow-based generative models: minimize the negative log-likelihood

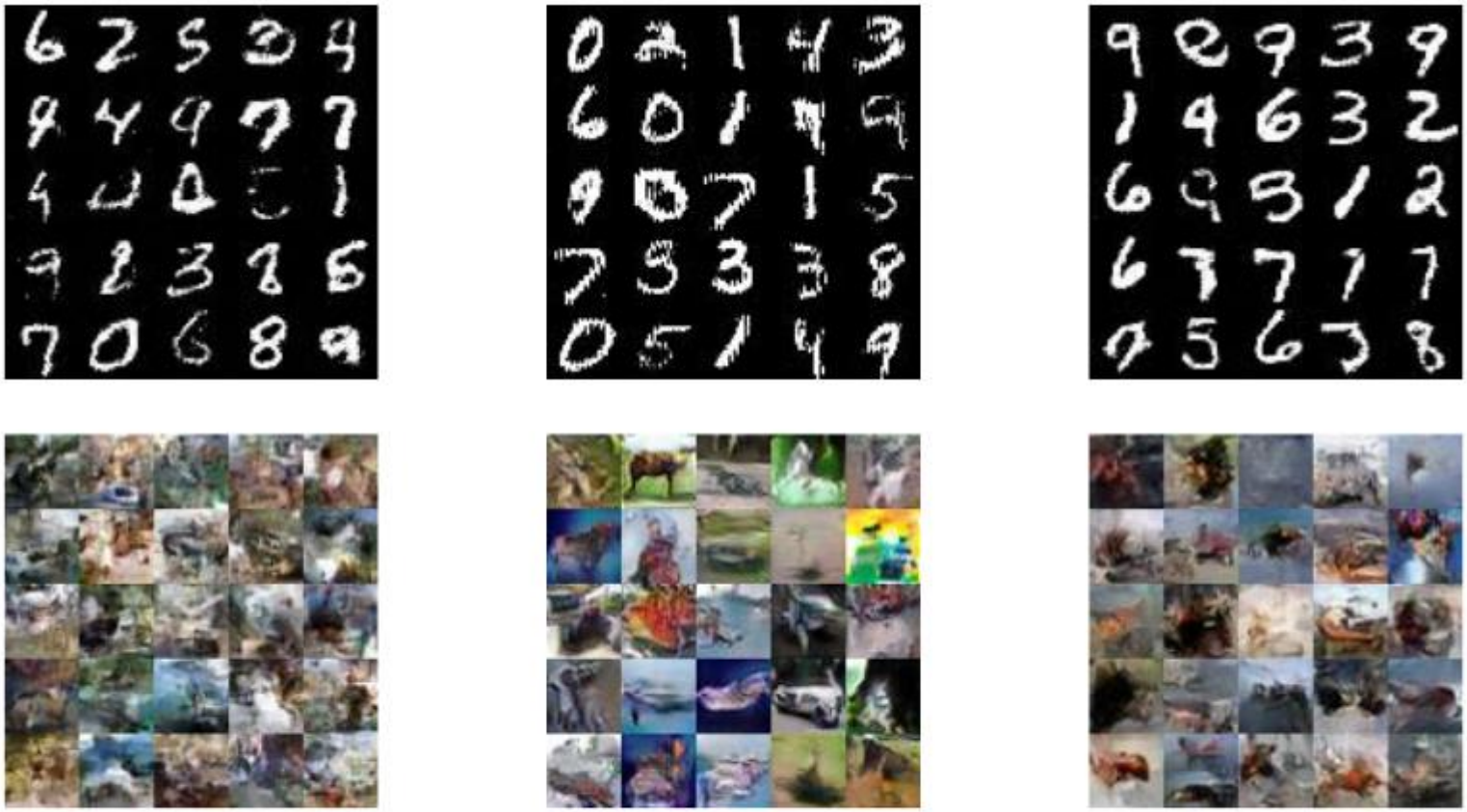


Toward using both maximum likelihood and adversarial training

1. Implicit models such as generative adversarial networks (GAN) often generate better samples compared to explicit models trained by maximum likelihood.

2. However, we know that the method based on maximum likelihood explicitly learn the probability density function of the input data.

3. To bridge this gap, we propose Flow-GANs, a generative adversarial network for which we can perform Exact likelihood evaluation, thus supporting both adversarial and maximum likelihood training.



(a) MLE (b) ADV (c) Hybrid

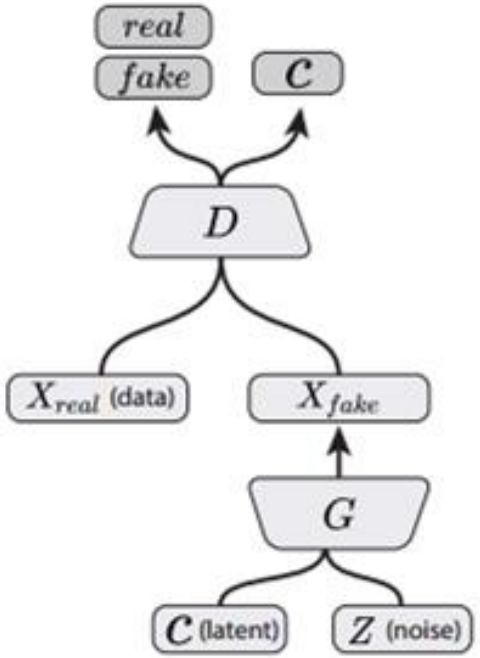
Figure 1: Samples generated by Flow-GAN models with different objectives for MNIST (**top**) and CIFAR-10 (**bottom**).

(8) InfoGANs

Interpretable Representation Learning by Information Maximizing Generative Adversarial Nets
NIPS 2016

- **InfoGANs**

  InfoGAN is an information-theoretic extension to the GAN that is able to learn disentangled representations in an unsupervised manner. InfoGANs are used when your dataset is very complex, when you'd like to train a cGAN and the dataset is not labelled, and when you'd like to see the most important features of your images.



InfoGAN
(Chen, et al., 2016)

(a) Varying c_1 on InfoGAN (Digit type)

(b) Varying c_1 on regular GAN (No clear meaning)

(c) Varying c_2 from -2 to 2 on InfoGAN (Rotation)

(d) Varying c_3 from -2 to 2 on InfoGAN (Width)

(a) Azimuth (pose)

(b) Elevation

(c) Lighting

(d) Wide or Narrow

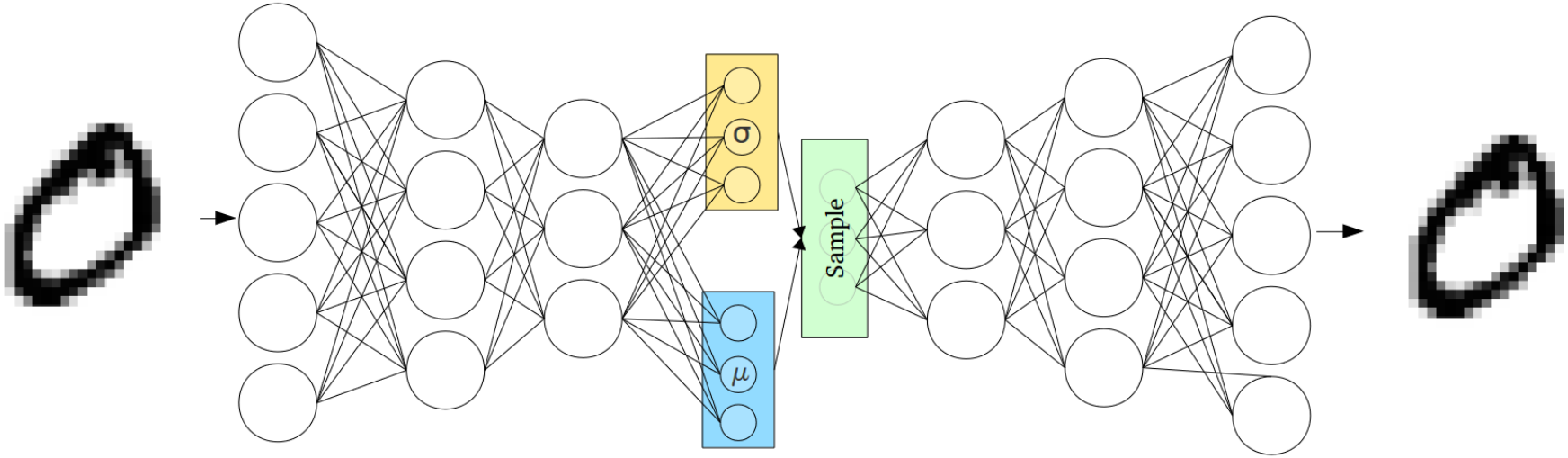
(a) Rotation

(b) Width

Variational Autoencoder

Irhum Shafkat, Feb 4, 2018

- **A Standard Variational Autoencoder**



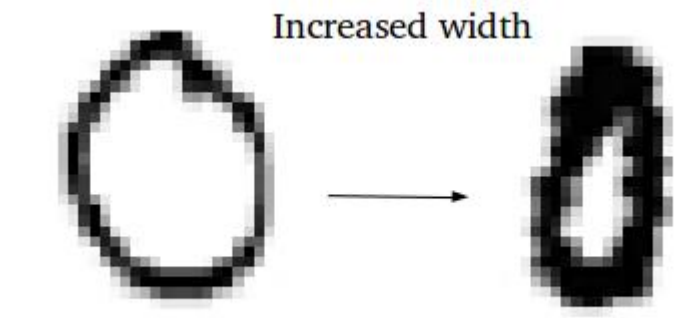
Variational Autoencoders (VAEs) are powerful generative models
Comparable with GANs

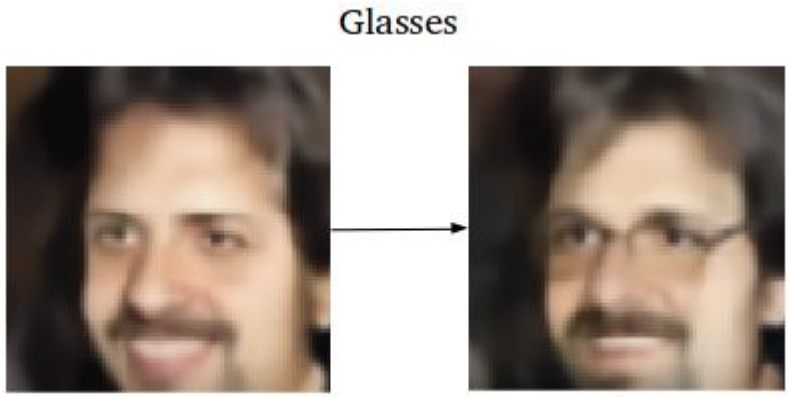
1. In contrast to the more standard uses of neural networks as regressors or classifiers, Variational Autoencoders (VAEs) are powerful **generative** models, now having applications as diverse as from generating fake human faces, to producing purely synthetic music.
2. **And why they're so useful in creating your own generative text, art and even music.**

But first, why VAEs?

When using generative models, you could simply want to generate a random, new output, that looks similar to the training data, and you can certainly do that too with VAEs.

But more often, you'd like to alter, or explore variations on *data you already have*, and not just in a random way either, but in a desired, *specific* direction. **This is where VAEs work better than any other method currently available**.



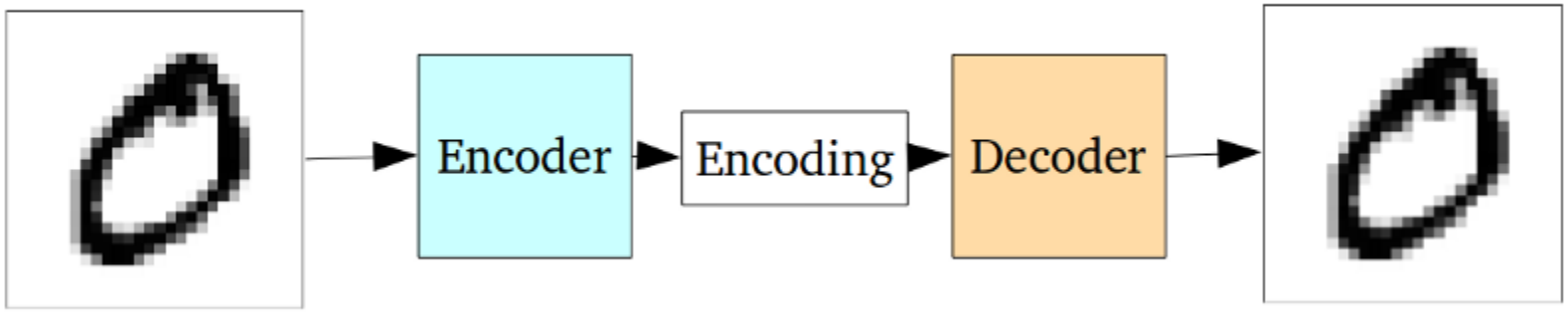


Like a "Conditional GAN" in an Image to Image Translation or specifically like a "InfoGan"

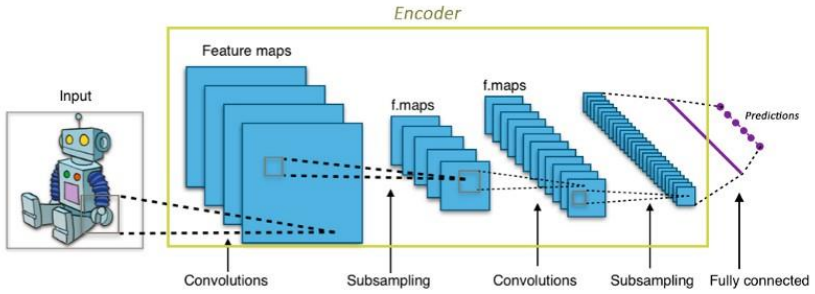
An standard autoencoder

- An autoencoder network is actually a pair of two connected networks, an encoder and a decoder.
- An encoder network takes in an input, and converts it into a smaller, dense representation, which the decoder network can use to convert it back to the original input.

A standard Autoencoder



An Encoder can be supposed as a CNN

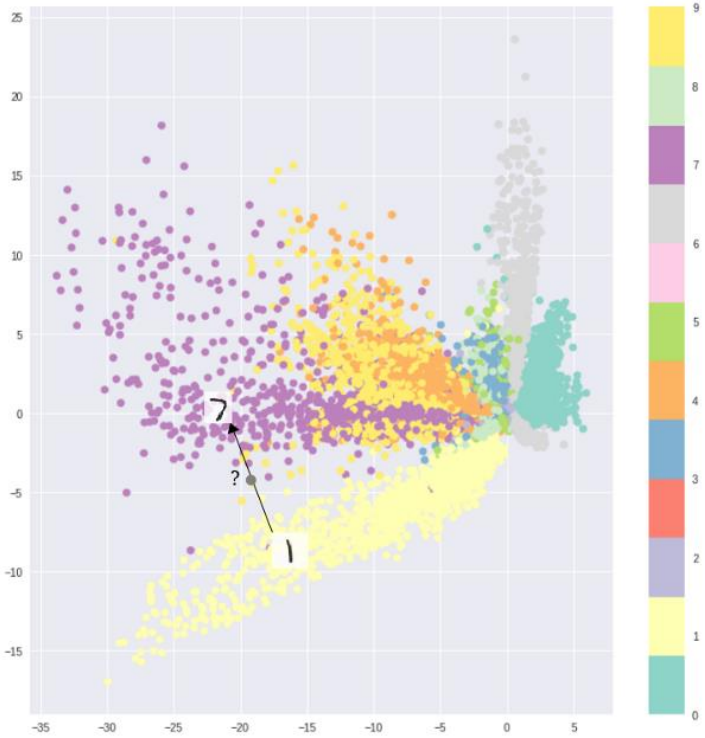


Learning an autoencoder

- The entire network is usually trained as a whole. The loss function is usually either the mean-squared error or cross-entropy between the output and the input, known as the *reconstruction loss*, which penalizes the network for creating outputs different from the input.
- As the encoding (which is simply the output of the hidden layer in the middle) has far less units than the input, the encoder must choose to discard information. The encoder learns to preserve as much of the relevant information as possible in the limited encoding, and intelligently discard irrelevant parts. The decoder learns to take the encoding and properly reconstruct it into a full image. Together, they form an autoencoder.

The problem with standard autoencoders for generation

- The fundamental problem with autoencoders, for generation, is that the latent space they convert their inputs to and where their encoded vectors lie, may not be continuous, or allow easy interpolation.



For example, training an autoencoder on the MNIST dataset, and visualizing the encodings from a 2D latent space reveals the formation of distinct clusters.

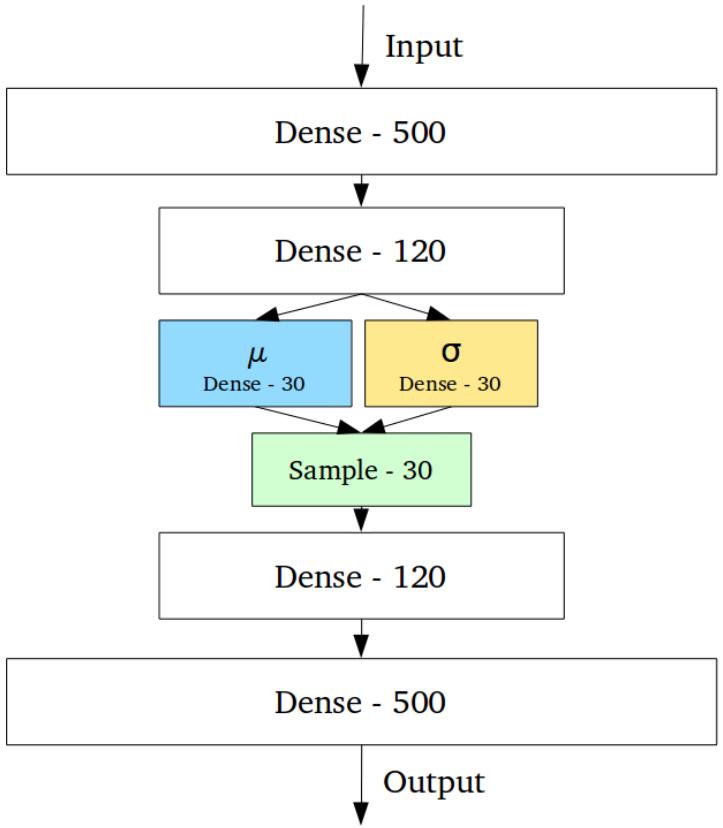
This makes sense, as distinct encodings for each image type makes it far easier for the decoder to decode them. This is fine if you're just *replicating* the same images.

But when you're building a *generative* model, you **don't** want to prepare to *replicate* the same image you put in. You want to randomly sample from the latent space, or generate variations on an input image, from a continuous latent space.

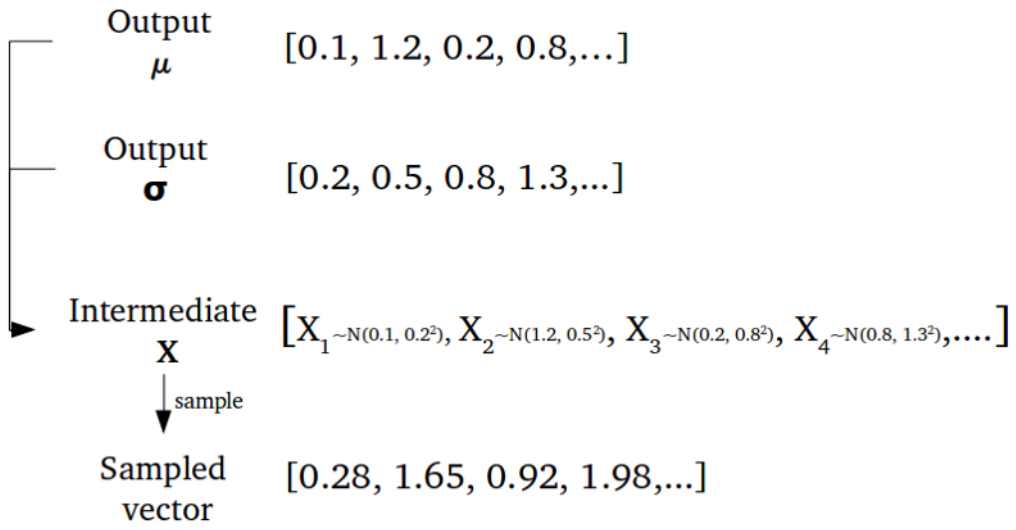
If the space has discontinuities (eg. gaps between clusters) and you sample/generate a variation from there, the decoder will simply generate an unrealistic output, because the decoder has *no idea* how to deal with that region of the latent space. During training, it *never saw* encoded vectors coming from that region of latent space.

Variational Autoencoders

- Variational Autoencoders (VAEs) have one fundamentally unique property that separates them from vanilla autoencoders, and it is this property that makes them so useful for generative modeling: their latent spaces are, *by design,* continuous, allowing easy random sampling and interpolation.
- It achieves this by doing something that seems rather surprising at first: making its encoder not output an encoding vector of size *n,* rather, outputting two vectors of size n: a vector of means**, μ**, and another vector of standard deviations, **σ**.

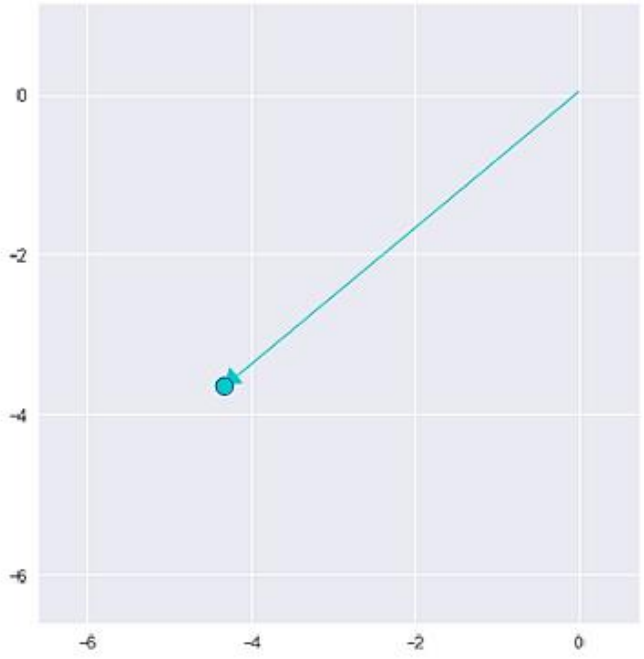


- They form the parameters of a vector of random variables of length n, with the *i* th element of **μ** and **σ** being the mean and standard deviation of the *i* th random variable, **X** i, from which we sample, to obtain the sampled encoding which we pass onward to the decoder:

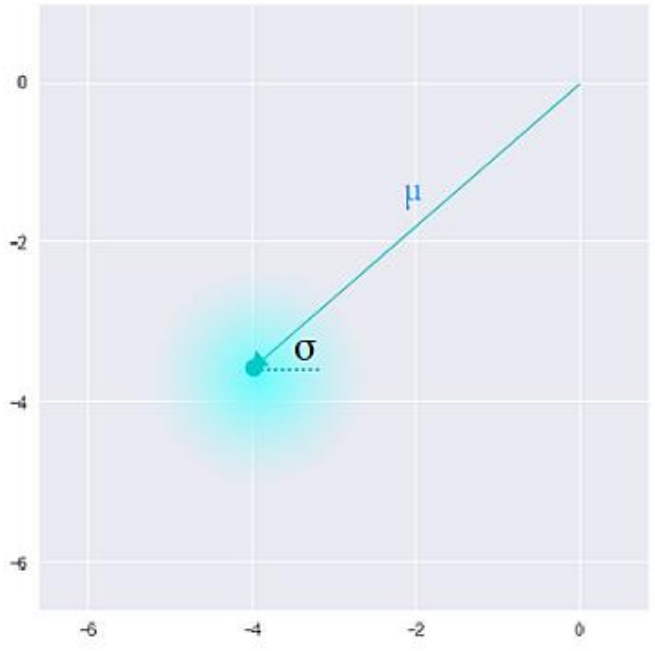


- Stochastically generating encoding vectors

- This stochastic generation means, that even for the same input, while the mean and standard deviations remain the same, the actual encoding will somewhat vary on every single pass simply due to sampling.



Standard Autoencoder
(direct encoding coordinates)

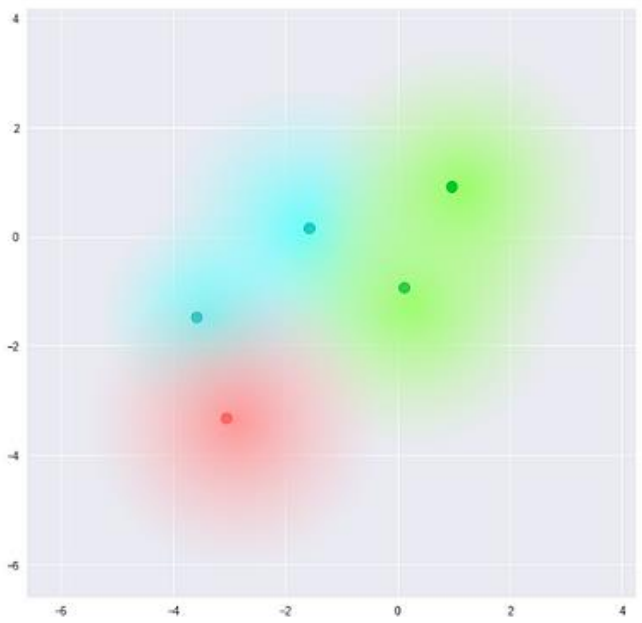


Variational Autoencoder
(μ and σ initialize a probability distribution)

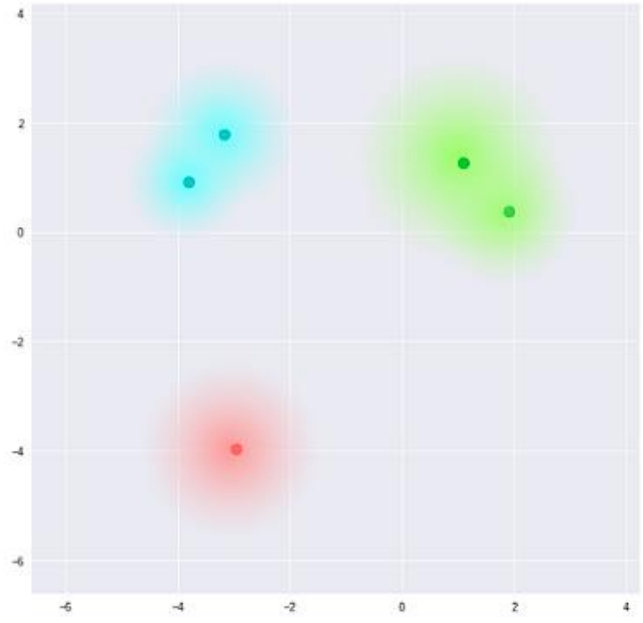
- Intuitively, the mean vector controls where the encoding of an input should be centered around, while the standard deviation controls the “area”, how much from the mean the encoding can vary.
- As encodings are generated at random from anywhere inside the “circle” (the distribution), the decoder learns that not only is a single point in latent space referring to a sample of that class, but all nearby points refer to the same as well.
- This allows the decoder to not just decode single, specific encodings in the latent space (leaving the decodable latent space discontinuous), but ones that slightly vary too, as the decoder is exposed to a range of variations of the encoding of the same input during training.

- The model is now exposed to a certain degree of local variation by varying the encoding of one sample, resulting in smooth latent spaces on a local scale, that is, for similar samples.
- Ideally, we want overlap between samples that are not very similar too, in order to interpolate *between* classes.
- However, since there are *no limits* on what values vectors **μ** and **σ** can take on, the encoder can learn to generate very different **μ** for different classes, clustering them apart, and minimize **σ**, making sure the encodings themselves don't vary much for the same sample (that is, less uncertainty for the decoder).
- This allows the decoder to efficiently reconstruct the *training* data.

- What we ideally want are encodings, *all* of which are as close as possible to each other while still being distinct, allowing smooth interpolation, and enabling the construction of *new* samples.



What we require



What we may inadvertently end up with

loss function :
Kullback–Leibler divergence

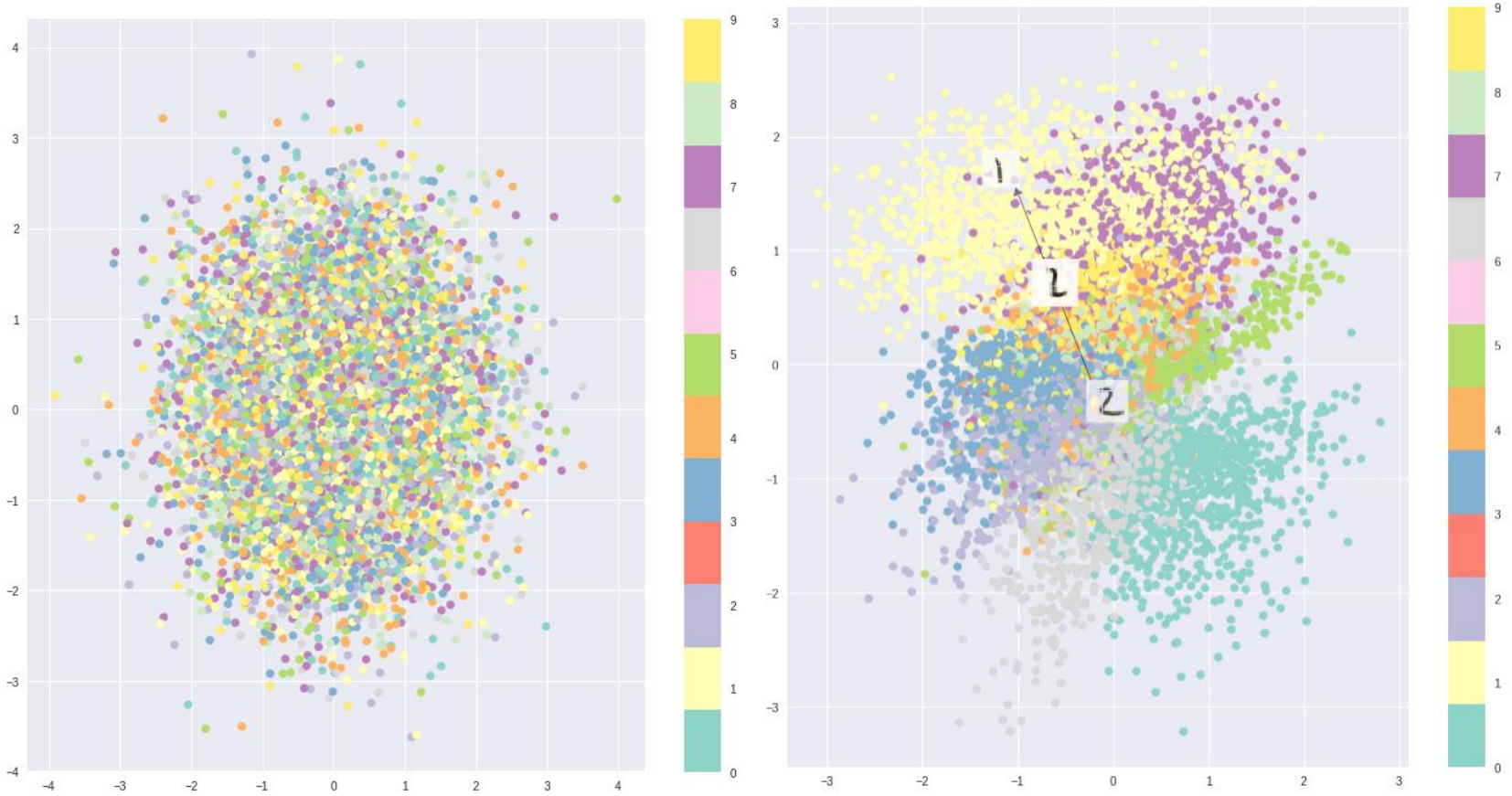
- In order to force this, we introduce the Kullback–Leibler divergence (KL divergence[2]) into the loss function. The KL divergence between two probability distributions simply measures how much they *diverge* from each other. Minimizing the KL divergence here means optimizing the probability distribution parameters **(μ** and **σ)** to closely resemble that of the target distribution.

$$\sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1$$

For VAEs, the KL loss is equivalent to the *sum* of all the KL divergences between the *component* $X_i \sim N(\mu_i, \sigma_i^2)$ in **X**, and the standard normal[3]. It's minimized when $\mu_i = 0$, $\sigma_i = 1$.

- Intuitively, this loss encourages the encoder to distribute all encodings (for all types of inputs, eg. all MNIST numbers), evenly around the center of the latent space.
- If it tries to “cheat” by clustering them apart into specific regions, away from the origin, it will be penalized.
- Now, using purely KL loss results in a latent space results in encodings densely placed randomly, near the center of the latent space, with little regard for similarity among nearby encodings.
- The decoder finds it impossible to decode anything meaningful from this space, simply because there really isn’t any meaning.

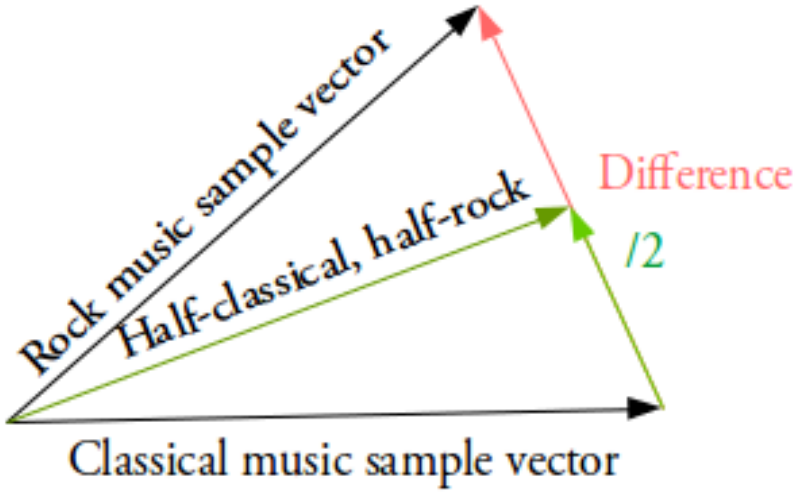
Optimizing the two together, however, results in the generation of a latent space which maintains the similarity of nearby encodings on the *local scale* via clustering, yet *globally,* is very densely packed near the latent space origin (compare the axes with the original).

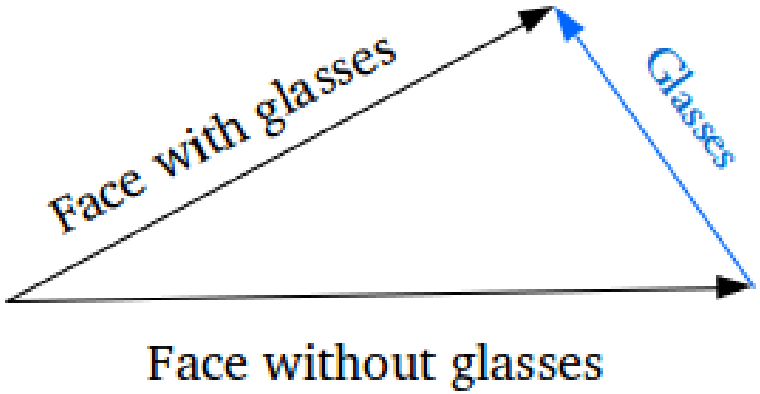


- Intuitively, this is the equilibrium reached by the *cluster-forming* nature of the reconstruction loss, and the *dense packing* nature of the KL loss, forming distinct clusters the decoder can decode.
- This is great, as it means when randomly generating, if you sample a vector from the same prior distribution of the encoded vectors, $N(\mathbf{0}, \mathbf{I})$, the decoder will successfully decode it.
- And if you're interpolating, there are no sudden gaps between clusters, but a *smooth mix of features* a decoder can understand.

Vector arithmetic

- So how do we actually produce these smooth interpolations we speak of? From here on out, it's simple vector arithmetic in the latent space.
- Interpolating between samples. For example, if you wish to generate a new sample halfway between two samples, just find the difference between their mean (**μ**) vectors, and add half the difference to the original, and then simply decode it.
- Adding new features to samples What about generating *specific features*, such as generating glasses on a face? Find two samples, one with glasses, one without, obtain their encoded vectors from the encoder, and save the difference. Add this new "glasses" vector to any other face image, and decode it.





Where to from here?

- There are plenty of further improvements that can be made over the **variational autoencoder**. You could indeed, replace the standard fully-connected dense encoder-decoder with a convolutional-deconvolutional encoder-decoder pair, such as this project[4], to produce great synthetic human face photos.
- You could even train an autoencoder using LSTM encoder-decoder pairs (using a modified version of the seq2seq architecture) for *sequential, discrete* data (something not possible with methods such as GANs), to produce synthetic text, or even interpolate between MIDI samples such as Google Brain's Magenta's MusicVAE[5]:

Thank you