Intelligent Systems: Reasoning and Recognition

James L. Crowley

MoSIG M1 Winter Semester 2021
Lesson 9 9 March 2021

Generative Networks: Auto-Encoders, Variational Autoencoders and Generative Adversarial Networks

Outline:

Background Reading

Kingma, D.P., Mohamed, S., Rezende, D.J. and Welling, M., Semi-supervised learning with deep generative models. In Advances in neural information processing systems (pp. 3581-3589), NIPS 2014.

- Radford A, Metz L, Chintala S. Unsupervised representation learning with deep convolutional generative adversarial networks.

Notation

x_d	A feature. An observed or measured value.
$\vec{X}$	A vector of D features.
D	The number of dimensions for the vector $\vec{X}$
$\{\vec{X}_m\}$ $\{y_m\}$	Training samples for learning.
M	The number of training samples.
$a_j^{(l)}$	the activation output of the j^{th} neuron of the l^{th} layer.
$w_{ij}^{(l)}$	the weight for the unit i of layer $l–1$ and the unit j of layer l.
b_j^l	the bias term for j^{th} unit of layer l.
ρ	The sparsity parameter

Key Equations

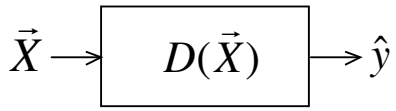
The average activation at layer l: $$\hat{\rho}_j = \frac{1}{M}\sum_{m=1}^{M} a_{j,m}^{(1)}$$

The autoencoder cost function: $$L_{sparse}(W, B; \vec{X}_m, y_m) = \frac{1}{2}(\vec{a}_m^{(2)} - \vec{X}_m)^2 + \beta \sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j)$$

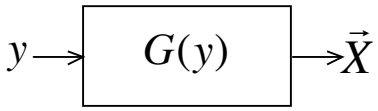
the Kullback-Leibler Divergence: $$\sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j) = \sum_{j=1}^{N^{(1)}} \left(\rho \log \frac{\rho}{\hat{\rho}_j} + (1-\rho) \log \frac{1-\rho}{1-\hat{\rho}_j} \right)$$

Generative Networks

Deep learning was originally invented for recognition. The same technology can be used for generation. Up to now we have looked at what are called "discriminative" techniques. These are techniques that attempt to discriminate a class label, y from a feature vector, $\vec{X}$.



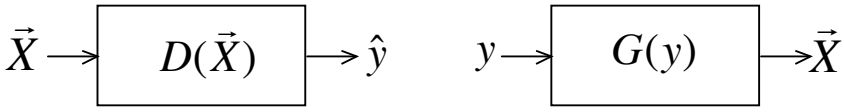
The same process can be used to learn a network that generates $\vec{X}$ given a code y. This is called a "generative" process.



Given an observable random variable $\vec{X}$, and a target variable, gradient descent allows us to learn a joint probability distribution, $P(\vec{X},\vec{Y})$, where $\vec{X}$, is generally composed of continuous variables, and $\vec{Y}$ is generally a discrete set of classes represented by a binary vector.

A discriminative model gives a conditional probability distribution $P(\vec{Y} \mid \vec{X})$.
A generative model gives a conditional probability $P(\vec{X} \mid \vec{Y})$

We can combine a discriminative process for one data set with a generative process from another and use these to make synthetic outputs.



A classic example is an autoencoder. However, to learn the Autoencoder we need to change use a new form of loss function based on entropy.

Cross-Entropy and the Kullback-Leibler Divergence

The entropy of a random variable is the average level of "information", "surprise", or "uncertainty" inherent in the variable's possible outcomes. Consider a set of M random scalars $\{X_m\}$ with N possible values, [1,N].

Formally: $\forall m = 1, M: \; h(X_m) \leftarrow h(X_m) + 1$

From this training set we can compute a probability distribution $P(X_m = x)$ more commonly written as $P(x)$

$$P(X_m = x) = \frac{1}{M} h(x)$$

The information in any one observation is

$$I(X_m = x) = -\log_2\left(P(x)\right)$$

Using a log of base 2 gives us information measured in binary digits (bits). The negative sign assures that the number is always positive or zero, because the log of a number less than 1 is negative.

Information expresses the number of bits needed to encode and transmit the value for an event.

Low probability events are surprising and convey more information.
High probability events are unsurprising and convey less information.

For example, consider x to have N=2 values, say 1, or 2. Then the P(X=x)=0.5 and the information is I(X) is 1 bit. If X had 8 possible values then, all equally likely, then P(X=x) = 1/8 =1/(2^3) and the information is –3 bits.

Entropy

For a set of M observations, the entropy is the expected value from the information from the observations. The entropy of the distribution measures the surprise (or information) obtained from an observation of a sample in the distribution. .

For a distribution *P(x)* of feature values X with N possible values, the entropy is

$$H(X) = -\sum_{x=1}^{N} P(x)\log_2\left(P(x)\right)$$

For example, for tossing a coin, there are two possible outcomes (N=2).
The probability of each outcome is P(X=x)=1/2.
This is the situation of maximum entropy

$$H(X) = -\sum_{x=1}^{2} P(x)\log_2(P(x)) = -\sum_{x=1}^{2} \frac{1}{2}\log_2\left(\frac{1}{2}\right) = -\sum_{x=1}^{2} \frac{1}{2}(-1) = 1$$

This is the most uncertain case. Similarly, in the case where there are N possible values for X, and all values are equally likely, then $P(X_m = x) = \frac{1}{N}$ and

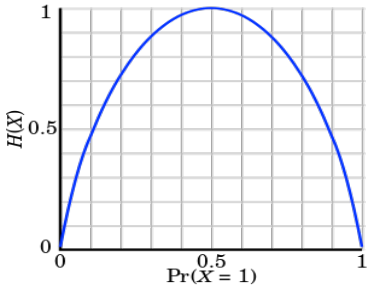
$$H(X) = -\sum_{x=1}^{N} \frac{1}{N}\log_2\left(\frac{1}{N}\right) = -\left(\frac{N}{N}\right)\log_2\left(\frac{1}{N}\right) = -\log_2\left(\frac{1}{N}\right)$$

For example, for 4 values, the entropy is 2 bits. It would require 2 bits to communicate an observation. On the other hand, consider when the distribution is a Dirac function, where X_m is always the same value of x_o,

$$P(x) = \delta(x - x_o) = \begin{cases} 1 & \text{if } x = x_o \\ 0 & \text{otherwise} \end{cases}$$

In this case, the value of X_m will always be x_o. Thus there is no information in an observation and the entropy will be zero. There is no surprise in an observation.

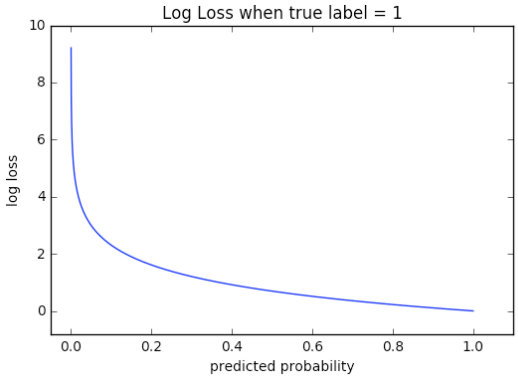
For any other distribution, Entropy measures the non-uniformity of the distribution.



Copied from https://en.wikipedia.org/wiki/Entropy_(information_theory)

Cross entropy

Cross-entropy is a measure of the difference between two probability distributions for a given set of events. Cross-entropy can be thought of as the total entropy between the distributions.



Copied from (https://ml-cheatsheet.readthedocs.io/en/latest/loss_functions.html)

Cross-entropy loss can be used to measure the performance of a classification model whose output is a probability value between 0 and 1, as with the sigmoid or soft-max. Cross-entropy loss increases as the predicted probability diverges from the actual label. So predicting a probability of .012 when the actual observation label is 1 would be bad and result in a high loss value. A perfect model would have a log-loss of 0.

Binary Cross-entropy loss is useful for training binary classifiers with the sigmoid activation function. Categorical Cross-Entropy is used to train a a multi-class network where softmax activation is used to output a probability distribution, $\vec{a}^{(out)}$, over the K classes .

Binary cross entropy

For a network with a single activation output, $a^{(out)}$

$$a^{(out)} = f(z^{(L)}) = \frac{1}{1+e^{-z^{(L)}}} = \frac{e^{z^{(L)}}}{e^{z^{(L)}}+1}$$

The Binary cross entropy is

$$C(a_m, y_m) = y_k \log(a_m) + (1-y_m)\log(1-a_m)$$

Categorical Cross Entropy Loss

For a network with a vector of K activation outputs, $\vec{a}^{(out)}$ with indicator vector $\vec{y}$ we calculate a separate loss for each target class.

The output activation is the softmax is

$$a_k = f(z_k) = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

and the Categorical cross entropy is

$$C(\vec{a}^{(out)}, \vec{y}) = -\sum_{k=1}^{K} y_k \log(a_k^{(out)})$$

When the indicators variables are encoded with one-hot encoding (Binary encoding with one variable for each output class), only the positive class where $y_k = 1$ is included in the loss. All other K-1 activations are multiplied by 0. In this case .

$$C(\vec{a}^{(out)}, \vec{y}) = \frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}$$

Where z_k is the linear input for the positive case. The derivative for the positive activations is

$$\frac{\partial a_k}{\partial z_k} = \frac{\partial f(z_k)}{\partial z_k} = \frac{\partial}{\partial z_k}\left(-\log\left(\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}\right)\right) = \left(\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}} - 1\right)$$

The derivative for the negative class activations.

$$\frac{\partial a_k}{\partial z_k} = \frac{\partial f(z_k)}{\partial z_k} = \frac{\partial}{\partial z_k}\left(-\log\left(\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}\right)\right) = \left(\frac{e^{z_k}}{\sum_{k=1}^{K} e^{z_k}}\right)$$

The Kullback-Leibler Divergence

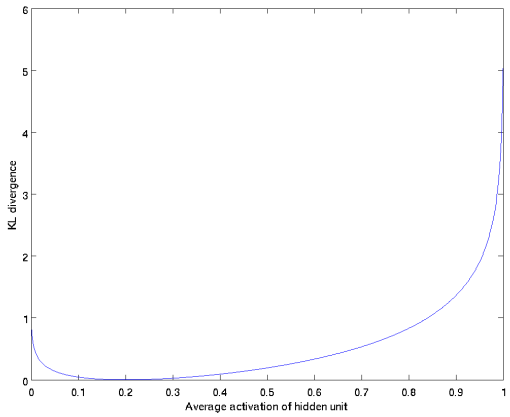
The Kullback-Leibler divergence, $D_{KL}(P \| Q)$ also known as the relative entropy of Q with respect to P measures the divergence between two distributions, P(X) and Q(X).

This can be used to define cross entropy as

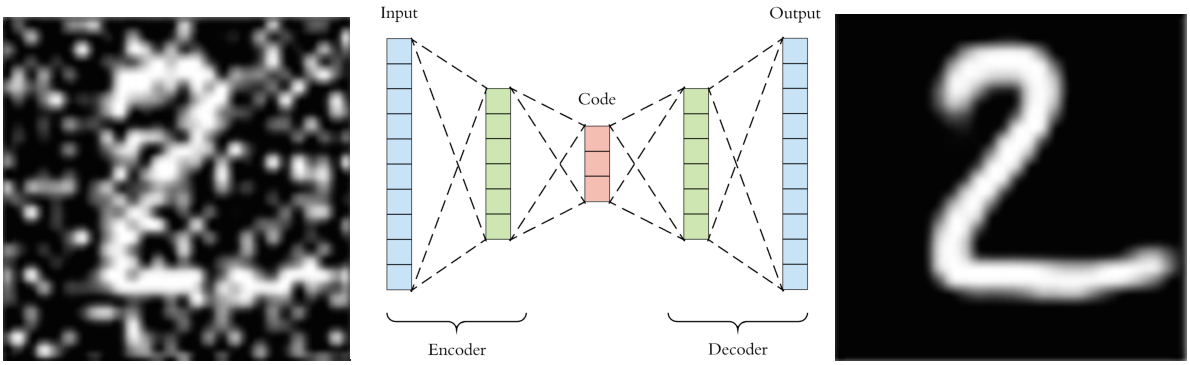
$$H(P,Q) = H(P) + D_{KL}(P \| Q)$$

We can use the Kullback-Leibler divergence to measure the divergence between a constant target activation, a, and an average observed activation for each unit, a_j.
The KL divergence between the desired and average activation is:

$$\sum_{j=1}^{N^{(1)}} KL(a \| a_j) = \sum_{j=1}^{N^{(1)}} \left(a \log \frac{a}{a_j} + (1-a) \log \frac{1-a}{1-a_j} \right)$$

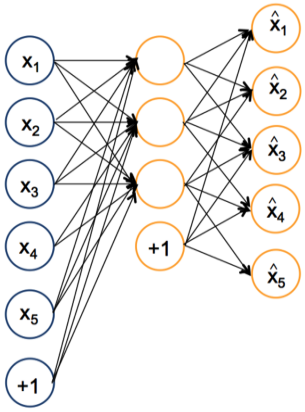


AutoEncoders



An auto-encoder is an unsupervised learning algorithm that uses back-propagation to learning a sparse set of features for describing the training data. Rather than try to learn a target variable, y_m, the auto-encoder tries to learn to reconstruct the input X using a minimum set of features (latent variables).

An Autocoder learns to reconstruct (generate) clean copies of data without noise. The Key concepts are:
1) The training data is the target. The error is the difference between input and output
2) Training is with standard back-propagation (or gradient descent), with the addition of a "sparsity term" to the loss function
3) Sparsity encodes the data with a minimum number of independent hidden units (Code vectors)



Using the notation from our 2 layer network, given an input feature vector $\vec{X}_m$ the auto-encoder learns $\{w_{ij}^{(1)}, b_j^{(1)}\}$ and $\{w_{jk}^{(2)}, b_k^{(2)}\}$ such that for each training sample, $\vec{a}_m^{(2)} = \hat{X}_m \approx \vec{X}_m$ using as few hidden units as possible.

Note that $N^{(2)} = D$ and that $N^{(1)} << N^{(2)}$

When the number of hidden units $N^{(2)}$ is less than the number of input units, D,

$\vec{a}_m^{(2)} = \hat{X}_m \approx \vec{X}_m$ is necessarily an approximation. The hidden units provide a "lossy" encoding for $\vec{X}_m$. This encoding can be used to suppress noise!

The error for back-propagation for each unit is a vector $\vec{\delta}_m^{(2)} = \vec{a}_m^{(2)} - \vec{X}_m$
with a component $\delta_{i,m}$ for component $x_{i,m}$ of the training sample $\vec{X}_m$

The hidden code is composed of independent "features" that capture some component of the input vector. Each cell of the code vector is driven by a receptive field whose sum of products with the receptive fields of other code cells is almost zero. That is, the code vectors are almost orthogonal. However, rather than minimizing the product of code vectors, sparsity seeks to generate the smallest set of code vectors that can reconstruct the training data without the noise. With an autoencoder the components may have some slight overlap. The average degree of independence is captured by a "sparsity parameter", $\hat{\rho}$.

The Sparsity Parameter

The sparsity $\hat{\rho}_j$ is the average activation for each of the hidden units *j=1* to $N^{(1)}$.

The auto-encoder will learn weights subject to a sparseness constraints specified by a target sparsity parameter ρ, typically set close to zero.

The simple, 2-layer auto-encoder is described by:

Level 0: $\vec{X}_m = \begin{pmatrix} x_{1,m} \\ \vdots \\ x_{D,m} \end{pmatrix}$ an input vector

level 1: $\vec{Y}_m = a_{j,m}^{(1)} = f(\sum_{i=1}^{D} w_{ij}^{(1)} x_{i,m} + b_j^{(1)})$ the code vector

level 2: $\hat{X}_m = a_{k,m}^{(2)} = f(\sum_{j=1}^{N^{(1)}} w_{jk}^{(2)} a_{j,m}^{(1)} + b_k^{(2)})$ the reconstruction of the input.

The output should approximate the input.

$$\vec{a}_m^{(2)} = \begin{pmatrix} a_1^{(2)} \\ \vdots \\ a_D^{(2)} \end{pmatrix} = \hat{X}_m \approx \vec{X}_m, \quad \text{with error } \vec{\delta}_m^{(2)} = \vec{a}_m^{(2)} - \vec{X}_m$$

The sparsity $\hat{\rho}_j$ for each hidden unit (code component) is computed as the average activation for the M training samples:

$$\hat{\rho}_j = \frac{1}{M}\sum_{m=1}^{M} a_{j,m}^{(1)}$$

The auto-encoder is trained to minimize the average sparsity. This is accomplished using back propagation, with a simple tweak to the cost function.

Standard back propagation tries to minimize a loss based on the sum of squared errors. The loss for each sample is.

$$C_m(\vec{X}_m, y_m) = \frac{1}{2}(\vec{a}_m^{(L)} - y_m)^2$$

For an auto-encoder, the target output is the input vector, and the loss is squared difference from the input vector:

$$C_m(\vec{X}_m, y_m) = \frac{1}{2}(\vec{a}_m^{(L)} - \vec{X}_m)^2$$

To impose "sparsity" we add an additional term to the loss.

$$C_m(\vec{X}_m, y_m) = \frac{1}{2}(\vec{a}_m^{(L)} - \vec{X}_m)^2 + \beta\sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j)$$

where $\sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j)$ is the Kullback-Leibler Divergence of the vector of hidden unit activations and β controls the importance of the sparsity parameter.

The average activation $\hat{\rho}_j$ is used to compute the correction. Thus you need to compute a forward pass on a batch of training data, before computing the back-propagation. Thus learning is necessarily batch mode.

The auto-encoder forces the hidden units to become approximately orthogonal, allowing a small correlation determined by the target sparsity, ρ. Thus the hidden units act as a form of basis space for the input vectors. The values of the hidden code layer are referred to as latent variables. The latent variables provide a compressed representation that reduces dimensionality and eliminates random noise.

To incorporate the KL divergence into back propagation, we replace

$$\delta_j^{(1)} = \frac{\partial f(z_j^{(1)})}{\partial z_j^{(1)}} \sum_{k=1}^{N^{(2)}} w_{jk}^{(2)} \delta_k^{(2)}$$

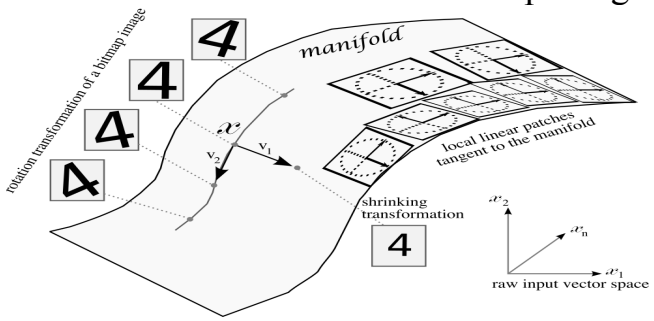
with

$$\delta_j^{(1)} = \frac{\partial f(z_j^{(1)})}{\partial z_j^{(1)}} \left(\sum_{k=1}^{N^{(2)}} w_{jk}^{(2)} \delta_k^{(2)} + \beta \left(-\frac{a}{a_j} + \frac{1-a}{1-a_j} \right) \right)$$

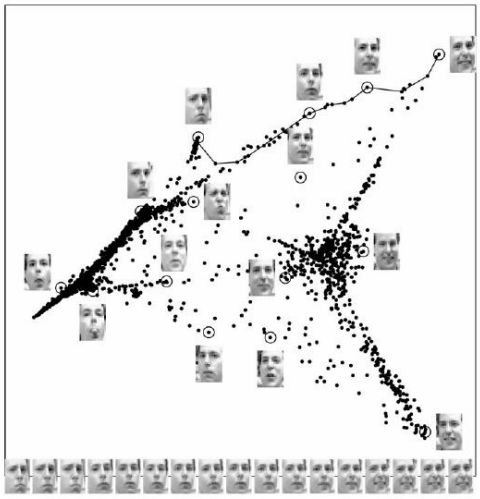
where $N^{(2)} = D$, the size of the size of the input and output vectors.
(The network output has the same number of components as the input).

Auto-Encoders Encode a Signal as Latent Variables.
AutoEncoders project the data onto a non-linear manifold that (should) provide a better representation of the latent space.

Affine Transformations of a Bitmap Image



Face Expressions for an individual



(Illustration from the NAACL 2013 lecture from R. Socher and C. Manning)

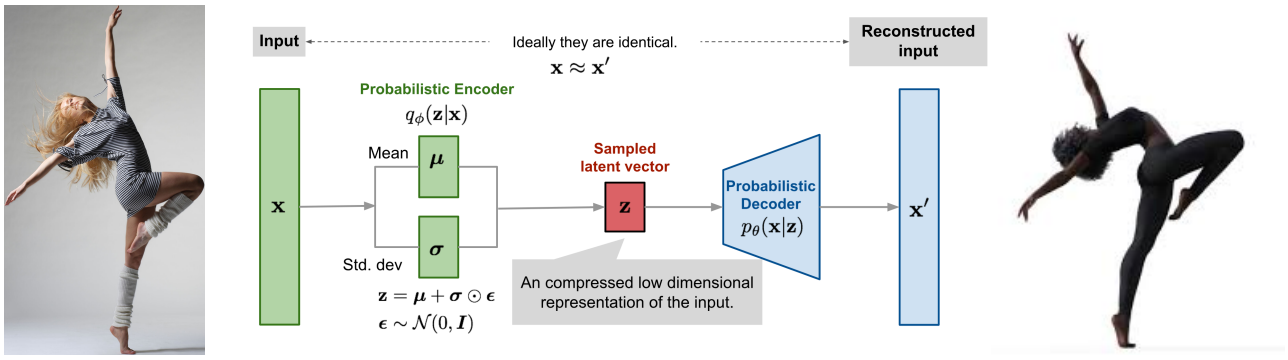
Positions on this manifold are expressed as vectors of latent variables.
Noise is not encoded in the latent variables.

Thus the latent variable can be used to reconstruct any signal on the manifold, without the presence of any signal (noise) that was not part of the manifold

The output of an auto-encoder can be used to drive a decoder to produce a filtered version of the encoded data or of another training set. However, the output from an auto-encoder is discrete.

Variational Autoencoders

We can adapt an auto-encoder to generate a *nearly* continuous output by replacing the code with a probabilistic code represented by a mean and variance.



This is called a Variational Autoencoder (VAE). VAEs combine a discriminative network with a generative network. VAEs can be used to generate "deep fake" videos sequences.

For a fully connected network, decoding is fairly obvious. The network input is a binary vector $\vec{Y}$ with k binary values y_k, with one for each target class. This is a code. The output for a training sample $\vec{Y}_m$ is an approximation of a feature vector belonging to the code class, $\hat{\vec{X}}_m$

$$\vec{a}_m^{(2)} = \hat{X}_m \approx \vec{X}_m$$

and the error is the difference between a output and the actual members of the class.

$$\vec{\delta}_m^{(2)} = \vec{a}_m^{(2)} - \vec{X}_m$$

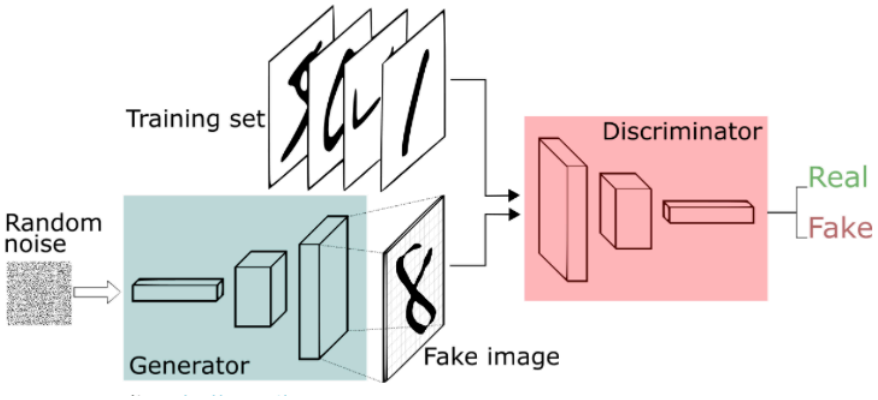
The average error for at training set $\{\vec{Y}_m\}$, $\{\vec{X}_m\}$ can be used to drive back-propagation.

Generative Adversarial Networks.

Generative Networks

It is possible to put a discriminative network together with a generative network and have them train each other. This is called a Generative Adversarial Network (GAN).

A Generative Adversarial Network places a generative network in competition with a Discriminative network.



The two networks compete in a zero-sum game, where each network attempts to fool the other network. The generative network generates examples of an image and the discriminative network attempts to recognize whether the generated image is realistic or not. Each network provides feedback to the other, and together they train each other. The result is a technique for unsupervised learning that can learn to create realistic patterns. Applications include synthesis of images, video, speech or coordinated actions for robots.

Generally, the discriminator is first trained on real data. The discriminator is then frozen and used to train the generator. The generator is trained by using random inputs to generate fake outputs. Feedback from the discriminator drives gradient ascent by back propagation. When the generator is sufficiently trained, the two networks are put in competition.

GAN Learning as Min-Max Optimization.

The generator is a function $\hat{\vec{X}} = G(\vec{z},\theta_g)$, where $G()$ is a differentiable function computed as a multi-layer perceptron, with trainable parameters, θ_g, and z is an input random vector with model $p_z(\vec{z})$, and $\hat{\vec{X}}$ is a synthetic (fake) pattern.

The discriminator is a differentiable function $D(\vec{X},\theta_d)$ computed as a multi-layer perceptron with parameters θ_d that estimates the likelihood that $\vec{X}$ belongs to the set described by the model θ_d.

The generator $\hat{\hat{X}} = G(\vec{z}, \theta_g)$ is trained to minimize $Log(1 - D(G(\vec{z}, \theta_g)))$

The perceptrons $D()$ and $G()$ play a two-player zero-sum min-max game with a value function *V(D, G)*:

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

In practice, this may not give sufficient gradient to learn. To avoid this, the discriminator is first trained on real data. The generator is then trained with the discriminator held constant. When the generator is sufficiently trained, the two networks are put in competition, providing unsupervised learning.

The discriminator is trained by ascending the gradient to seek a max:

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

The generator is trained by seeking a minimum of the gradient:

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