Generative Networks and the AutoEncoder

James L. Crowley[1]

[1] Grenoble Insitut Polytechnique, Univ. Grenoble Alpes
http://crowley-coutaz.fr/jlc/jlc.html

Abstract. Neural networks were invented to classify signals. However, networks can also be used for generate signals. This chapter introduces generative networks, and shows that discriminative networks can be combined with generative networks to produce an autoencoder. Autoencoders can be trained with self-supervised learning to provide a compact code for signals. This code can be used to reconstruct clean copies of noisy signals. With a simple modification to the loss function using information theory, an autoencoder can be used for unsupervised discovery of categories in data, providing the basis for self-supervised learning that is at the heart of Transformers. To explain this modification, this chapter reviews basic concepts from information theory including entropy, cross entropy and the Kullback-Leiblier (KL) divergence. The chapter concludes with brief presentations of Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs).

Learning Objectives: This chapter provides students with an introduction to generative neural networks and autoencoders, covering fundamental concepts from information theory as well as well as applications such as Variational Autoencoders (VAEs) and Generative Adversarial Networks (GANs). Mastering the material in this chapter will enable students to understand how a neural network can be trained to generate signals, and how an auto-encoder can be used for unsupervised and self-supervised learning. Students will acquire an understanding of fundamental concepts from information theory such as entropy and sparsity. Students will be able to explain how generative networks can be combined with discriminative networks to construct generative adversarial networks.

Keywords: Generative Networks, autoencoders, entropy, Kulback-Leibler divergence, Variational Autoencoders (VAEs), Generative Adversarial Networks (GANs).

1 Generative Networks

Neural networks were originally invented to recognize categories of phenomena in signals. For recognition, a network is trained to map a feature vector, $\vec{X}$, into a pre-

dicted category label, $\hat{y}$, taken from a discrete set of possible categories.[1] Such networks are called discriminative networks.

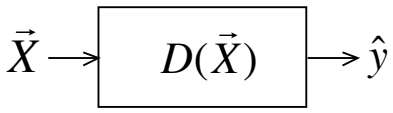

Fig 1. A discriminative network maps a feature vector, $\vec{X}$, into a predicted category label, $\hat{y}$.

In principle, with enough data and computing, networks can be trained to estimate any computable function. In particular, networks can be trained to generate a typical feature vector from a category label. This is called a generative network. With sufficient data and computing, generative networks can be trained to generate realistic imitations for any signal including image, sounds, video, and motor commands for coordinated robot motions.

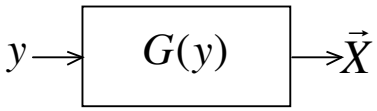

Fig. **2** A generative network maps a category label to a typical feature vector.

A discriminative network can be paired with a generative network to create an autoencoder. The autoencoder maps an input vector onto a compact code vector. This code vector can then be used to generate a clean copy of the signal, as shown in figure 2. When trained with a least squares loss function, as described in chapter 1, the autoencoder represents a signal with the largest principal values that would be found by eigenspace analysis of the training data. Augmenting the least-squares loss function with an information theoretic measure of sparsity enables the autoencoder to learn a minimal set of independent components that can be used to provide an abstract representation for a data set.

[1] The predicted category label is displayed with a "hat" symbol to distinguish it as an estimation that may vary from the true label, y.

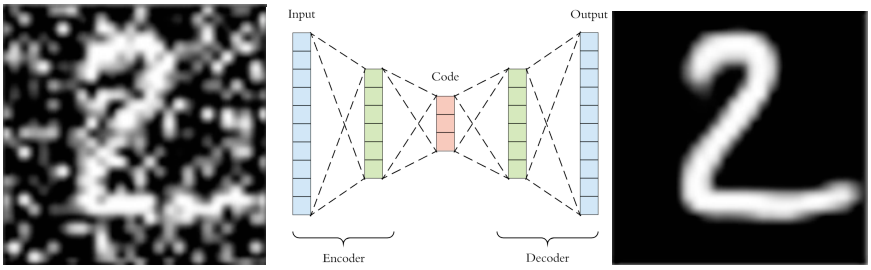

Fig. **3** An autoencoder combines a disciminative encoder with a generative decoder to create clean (denoised) copies of a signal.

2 The AutoEncoder

An autoencoder is an unsupervised learning algorithm that uses back-propagation to learning a sparse set of features for describing a set of data. In the early days of perceptron learning, the autoencoder was used to compensate for a lack of labeled training data required to develop the back-propagation algorithm for computing gradient descent [1]. With the autoencoder, the data is its own ground truth! The autoencoder was gradually recognized as a powerful technique for self-supervised learning [2].

Rather than learn to estimate a target class label, *y*, the auto-encoder learns to reconstruct an approximation, $\hat{X}$, for an input vector $\vec{X}$ using a minimum size code vector. The autoencoder is trained to preserve the forms that are common to all of the training data while ignoring random variations (noise). The hidden units provide a code vector that is said to represent the latent energy (or information) in the signal.

Let $\hat{X}$ be the reconstructed version for a signal, $\vec{X}$. The error from using the reconstructed version to replace the signal is the difference between the input and ouput. The square of this error can be used as a loss function, *C(-)* to learn a set of network parameters[2], $\vec{w}$, using gradient descent, as shown in equation 1.

$$C(\vec{X},\vec{w}) = (\hat{X} - \vec{X})^2 \qquad (1)$$

When computed with a loss function based on the least-squares error, the autoencoder converges to a code vector that captures a maximum quantity of signal energy from the training set for any given size of code vector. The theoretical minimum for such an encoding is provided by the eigenvectors of the training set. Thus, when computed with least squares error, the autoencoder provides an approximation for the principal components of the training data that express the largest energy. The addition of an information theoretic term for sparsity can force the auto-encoder to learn independent components of the data set, providing a minimal abstract representation for the data.

[2] In the following, we will simplify the notation by including the bias term, *b*, as a component of the network parameters, $\vec{w}$.

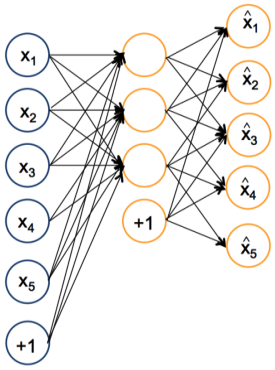

Fig 2. A 2-layer autoencoder with an input vector of *D=5*, and code vector of *N=3*.

Consider a two-layer autoencoder for a signal composed of *D* elements as shown in figure 2. The first (hidden) layer, referred to as layer 1, is composed of *N* hidden units that provide a code vector, such that *N* is smaller than *D*. The second layer, referred to as layer 2, is composed of *D* units that use this code vector to generate a clean copy of the input signal. Each of the *N* hidden code units is fed by *D* input signals plus a bias. Using the notation from chapter 1, these *D·(N+1)* parameters are $\{w_{ij}^{(1)}, b_j^{(1)}\}$. The *N(D+1)* weights and biases the second layer (layer 2) are $\{w_{jk}^{(2)}, b_k^{(2)}\}$.

By training with the least-squares error between the signal and its reconstruction, the network learns to provide an approximate copy $\hat{X}$ of the input signals, $\bar{X}$ using the *N* coefficients of the hidden code vector. The code vector provides a lossy encoding for $\bar{X}$. Training the network with a least-squares loss function trains the network to learn minimal set of features that approximates a form of eigenspace coding of the training data, as explained above. The weights and biases of each neural unit (the receptive fields) define an orthogonal basis space that can be used for a minimum energy reconstruction of input signals. However, the energy for any input pattern is typically spread over many of the coefficients of a code vector. For pattern recognition, what we would like is to learn a set of independent categories for the data, such that any input data vector would be reflected by a concentration of energy in only one of the hidden units, with the other hidden units nearly zero. In this way, the hidden units provide a form of "one-hot coding" for the categories in the training data. This can be done by training the autoencoder with a cost function that includes an additional term that forces the coefficients to be unique for each data sample. This additional cost term is referred to as "sparsity", represented by the symbol, ρ. Defining the loss function for sparsity requires some background from information theory.

3 Background from Information Theory

3.1 Entropy

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 integers, $\{X_m\}$, with N possible values in the range 1 to N. We can count the frequency of occurrence for each value with table of N cells, $h(x)$. This is commonly referred to as a histogram.

$$\forall m = 1, M : \; h(X_m) \leftarrow \; h(X_m) + 1 \tag{2}$$

From this training set we can compute a probability distribution that tells us the probability that any random variable X_m has the value x. This is written as $P(X_m=x)$ or more simply as $P(x)$.

$$P(x) = \frac{1}{M} h(x) \tag{3}$$

The information about the set $\{X_m\}$ provided by the observation that a sample X_m has value x, is measured with the formula:

$$I(x) = -\log_2\left(P(x)\right) \tag{4}$$

This is the formal definition of information used in information theory. In physics, the practice is to use logarithms of base 10. In informatics and computer science we generally prefer base 2 because a base 2 logarithm measures information in binary digits, referred to as bits. The information content in bits tells us the minimum number of base 2 (binary) digits that would be needed to encode the data in the training set. The negative sign assures that the number is always positive or zero, as 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 the case where X to has $N=2$ values and the histogram has a uniform distribution, $P(x)=0.5$.

$$I(x) = -\log_2\left(P(x)\right) = -\log_2\left(2^{-1}\right) = 1 \tag{5}$$

The information from a sample X_m is 1 bit. If X_m had 8 possible values then, all equally likely, then $P(X_m=x) = 1/8 = 2^{-3}$ and the information is 3 bits.

3.2 Computing 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 in-

formation) obtained from an observation of a sample in the distribution. For a distribution $P(x)$ of features with N possible values, the entropy is

$$H(X) = -\sum_{x=1}^{N} P(x)\log_2\left(P(x)\right) \tag{6}$$

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

$$H(X) = -\sum_{x=1}^{2} \frac{1}{2}\log_2\left(\frac{1}{2}\right) = 1 \tag{7}$$

This is the most uncertain case. In the case where N=4 values, the entropy is 2 bits. It would require 2 bits to communicate an observation. In the general case, there are N possible values for X, and all values are equally likely, then

$$H(X) = -\sum_{x=1}^{N} \frac{1}{N}\log_2\left(\frac{1}{N}\right) = -\log_2\left(\frac{1}{N}\right) \tag{8}$$

On the other hand, consider when the distribution is a Dirac function, where all samples X_m have the same value, x_o.

$$P(x) = \delta(x - x_o) = \begin{cases} 1 & \text{if } x = x_o \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

In this case, there is no surprise (no information) in the observation that $X_m=x_o$ and the entropy will be zero. For any distribution, Entropy measures the non-uniformity of the distribution.

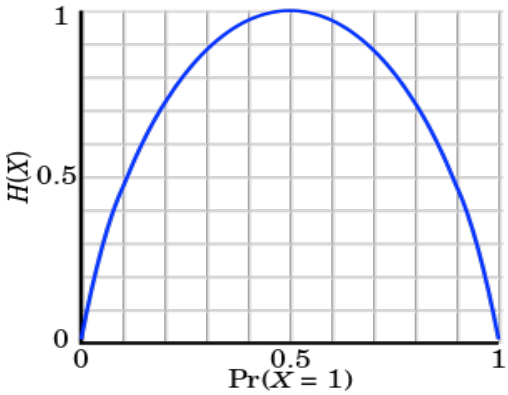

Fig. 3 Entropy measures the non-uniformity of a distribution. A uniform distribution, where all samples are equally likely, has an entropy of 1. (Image from [3], copied from wikipedia[3]).

[3] *Wikipedia, The Free Encyclopedia*, s.v. "Entropy (information theory)," (accessed August 20, 2022),

3.3 Cross entropy

Cross-entropy is a measure of the difference between two probability distributions. Cross-entropy can be thought of as the total entropy between two distributions and is a measure of the difference (or lack of similarity) of the two distributions. Perfectly matching distributions have a cross entropy of 0.

Cross-entropy loss is commonly used to measure the performance of a classification model whose output is a probability between 0 and 1, as with the sigmoid or softmax. 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 value for the log of the cross-entropy. A perfect model would have a log-loss of 0.

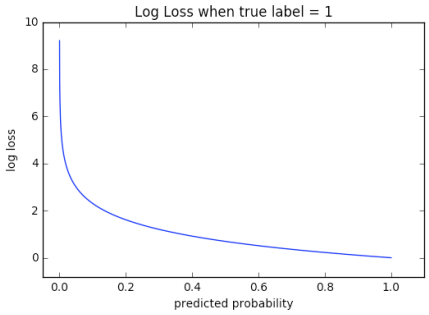

Fig. 4 Cross entry measures the difference of two distributions. Copied from "read the docs" [4]

Binary Cross-entropy loss is useful for training binary classifiers with the sigmoid activation function. Categorical Cross-Entropy is used to train a multi-class network where softmax activation is used to output a probability distribution, $P(a)$, over a set of output activations for K classes.

3.4 Binary cross entropy

For binary classification, a network has a single activation output, $a^{(out)}$:

$$a^{(out)} = f(z^{(L)}) = \frac{1}{1+e^{-z^{(L)}}} = \frac{e^{z^{(L)}}}{e^{z^{(L)}}+1} \tag{10}$$

For a target variable of y, the binary cross entropy is

$$H(a^{(out)}, y) = y\log(a^{(out)}) + (1-y)\log(1-a^{(out)}) \tag{11}$$

---

https://en.wikipedia.org/w/index.php?title=Entropy_(information_theory)&oldid=1101826646

[4] https://docs.readthedocs.io/en/stable/privacy-policy.html

3.5 Categorical Cross Entropy Loss

For a network with a vector of K activation outputs, $a_k^{(out)}$ with indicator vector y_k, each target class, $k = 1,...,K$, contributes to the cost (or loss) function used for gradient descent. This is represented by using a softmax activation function where the activation energy for each component, a^k, is

$$a_k = f(z^k) = \frac{e^{z^k}}{\sum_{k=1}^{K} e^{z^k}} \tag{12}$$

This results in a categorical cross entropy of

$$H(\vec{a}^{(out)}, \vec{y}) = -\sum_{k=1}^{K} y_k \log(a_k^{(out)}) \tag{13}$$

When the indicators variables are encoded with one-hot encoding (1 for the true class, zero for all others), only the positive class where $y_k = 1$ is included in the cost function. All of the other $K-1$ activations are multiplied by 0. In this case:

$$H(\vec{a}^{(out)}, \vec{y}) = \frac{e^{z^k}}{\sum_{k=1}^{K} e^{z^k}} \tag{14}$$

Where z_k is the linear sum of weighted activations. The derivative for the positive activations is

$$\frac{\partial a_k}{\partial z_k} = \frac{\partial f(z_k)}{\partial z_k} = \frac{\partial}{\partial z_k}\left(-\log\left(\frac{e^{z^k}}{\sum_{k=1}^{K} e^{z^k}}\right)\right) = \frac{e^{z^k}}{\sum_{k=1}^{K} e^{z^k}} - 1 \tag{15}$$

The derivative for the negative class activations.

$$\frac{\partial a_k}{\partial z_k} = \frac{e^{z^k}}{\sum_{k=1}^{K} e^{z^k}} \tag{16}$$

3.6 The Kullback-Leibler Divergence

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

$$D_{KL}(P \| Q) = \sum_{x=1}^{N} P(x) \log\left(\frac{P(x)}{Q(x)}\right) \tag{17}$$

The KL divergence can be used to define cross entropy as

$$H(P,Q) = H(P) + D_{KL}(P \| Q) \tag{18}$$

The KL divergence can also be to measure the divergence between a constant and a distribution. For example, KL divergence can provide a measure of the distance between a target activation energy, a, and a distribution of activation energy for a set of n units, a_n.

The KL divergence between a target activation, a, and a distribution of N activations, a_n, is:

$$D_{KL}(a \,||\, a_n) = \sum_{n=1}^{N} a \cdot \log\left(\frac{a}{a_n}\right) \tag{18}$$

Adding the KL divergence to the loss function for the input layer of an autoencoder forces the autoencoder to learn a sparse code vector resembling a one-hot coding vector for the independent categories of a training set, as explained below.

3.7 Sparsity

Sparsity measures the average activation energy for the N hidden units of an autoencoder from a batch of M samples. During training with M training samples, the Kullback-Leibler (KL) divergence can be used to compare the distribution of sparsity values for the N hidden (latent) code variables to a target value, ρ. Including a target for sparsity in the loss function during batch training constrains the network parameters to converge toward a configuration that concentrates activation to a single unit for each training sample. This forces the autoencoder to learn a code vector where each latent unit represents one of independent components of the training data. The independent components are assumed to represent independent categories of phenomena in the training set.

Consider a simple 2-layer autoencoder that learns to reconstruct data from a training set $\{\vec{X}_m\}$ of M training samples of dimension D features. Assume a code vector, $\vec{a}$, of $N<<D$ latent variables, a_n. For any input vector $\vec{X}$, the N coefficients of the code vector, $\vec{a}$, are computed by

$$a_n = f\left(\sum_{d=1}^{D} w_{dn}^{(1)} x_d + b_n^{(1)}\right) \tag{19}$$

where $f(\cdot)$ is a non-linear activation function, such as sigmoid or RELU. The sparsity for this vector is the average activation energy:

$$\rho = \frac{1}{N}\sum_{n=1}^{N} a_n \tag{20}$$

The D components of the output vector, $\hat{X}$, are computed as

$$\hat{x}_d = f\left(\sum_{n=1}^{N} w_{nd}^{(2)} a_n + b_d^{(2)}\right) \tag{21}$$

The least squares cost for using $\hat{X}_m$ as an estimate for $\vec{X}_m$ is:

$$C(\vec{X}_m;\vec{w}) = \frac{1}{2}(\hat{X}_m - \vec{X}_m)^2 \tag{21}$$

where $\vec{w}$ represents the network parameters including the bias terms as explained above in equation 1. The average least squares loss for a training set $\{\vec{X}_m\}$ of *M* training samples is

$$C(\{\vec{X}_m\};\vec{w}) = \frac{1}{2}\sum_{m=1}^{M}(\hat{X}_m - \vec{X}_m)^2 \tag{22}$$

The auto-encoder can be trained to minimize the sparsity by addition of a loss term that uses the KL divergence to measure the difference between a target sparsity, ρ, and the vector of sparsities from the training data, $\hat{\rho}_n$.

$$C(\{\vec{X}_m\};\vec{w}) = \frac{1}{2}\sum_{m=1}^{M}(\hat{X}_m - \vec{X}_m)^2 + \beta\sum_{n=1}^{N} D_{KL}(\rho \,\|\, \hat{\rho}_n) \tag{23}$$

where ρ is a target value for sparsity, typically close to zero, and β controls the importance of sparsity. Computing the cost requires a pass through the M training samples to compute the sparsity for the training data, and thus requires batch learning.

The auto-encoder forces the hidden units to become approximately independent, representing hidden, or latent, information in the data set. Incorporating KL divergence into back propagation requires adding the derivative of the KL divergence to the average error term used to compute backpropagation. Recall that when computed with least-squares error, the output error for m^{th} training sample is;

$$\vec{\delta}_m^{out} = \left(\hat{X}_m - \vec{X}_m\right) \tag{24}$$

The average error term for a batch of M samples is

$$\vec{\delta}^{out} = \sum_{m=1}^{M}\left(\hat{X}_m - \vec{X}_m\right) \tag{25}$$

This output error is used to compute the error energy for updating weights and biases with backpropagation, as explained in chapter 1. For example the error term for the D individual output units is

$$\vec{\delta}_d^{(2)} = \vec{\delta}^{(out)}\frac{\partial f(z_d^{(2)})}{\partial z_d^{(2)}} \tag{26}$$

where *f()* is the non-linear activation function, typically a sigmoid for the autoencoder. The error energy for the discriminative layer is:

$$\vec{\delta}_n^{(1)} = \frac{\partial f(z_n^{(2)})}{\partial z_n^{(2)}}\sum_{d=1}^{D} w_{dn}^{(2)}\vec{\delta}_d^{(2)} \tag{27}$$

The discriminator can be trained to learn a sparse code by adding a penalty term based on the KL divergence of the activation energies of the code vector to the error energy at level 1 as shown in equation 28.

$$\vec{\delta}_n^{(1)} = \frac{\partial f(z_n^{(2)})}{\partial z_n^{(2)}} \sum_{d=1}^{D} w_{dn}^{(2)} \vec{\delta}_d^{(2)} + \beta\left(-\frac{\rho}{\hat{\rho}_n} + \frac{1-\rho}{1-\hat{\rho}_n}\right) \tag{28}$$

where ρ is the target sparsity, and $\hat{\rho}_n$ is the sparsity (average activation energy) of the n^{th} hidden unit for the training set of M samples. The error energy computed with equation 28 is then used to update the weights and biases of the hidden units at level 1, as discussed in in the previous chapter.

4 Variational Autoencoders

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. 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) [4]. VAEs combine a discriminative network with a generative network. VAEs can be used to generate "deep fake" videos as well as realistic rendering of audio, video or movements.

Neural networks can be seen as learning a joint probability density function $P(\vec{X},Y)$ that associates that associates a continuous random vector, $\vec{X}$, with a discrete random category, Y. A discriminative model gives a conditional probability distribution $P(Y \mid \vec{X})$. A generative model gives a conditional probability $P(\vec{X} \mid Y)$. Driving the conditional probability $P(\vec{X} \mid Y)$ with a probability distribution P(Y) can be used to generate a signal that varies smoothly between learned outputs.

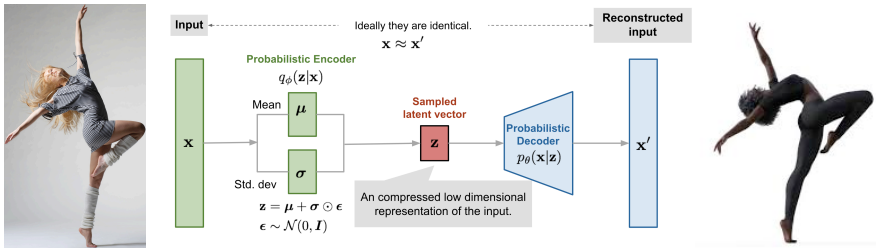

Fig 5. A Variational autoencoder generates an animated output based on the motion patterns of an input. (Image from Lilian Weng[5])

Normally, for an autoencoder, the latent vector is a one-hot encoded binary vector $\vec{Z}$ with k binary values. The VAE learns a probabilistic encoder that provides a mean

[5] https://lilianweng.github.io/posts/2018-08-12-vae/

and variance in this latent space, $\vec{Z}$. The probabilistic decoder is trained to generate a target output $\vec{Y}_m$ for the latent encoding for each training sample $\vec{X}_m$. Driving this decoder with the smoothly varying probability distribution leads to a smoothly varying output, $\hat{Y}$.

5. Generative Adversarial 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) [5]. [6], [7]. A Generative Adversarial Network places a generative network in competition with a Discriminative network.

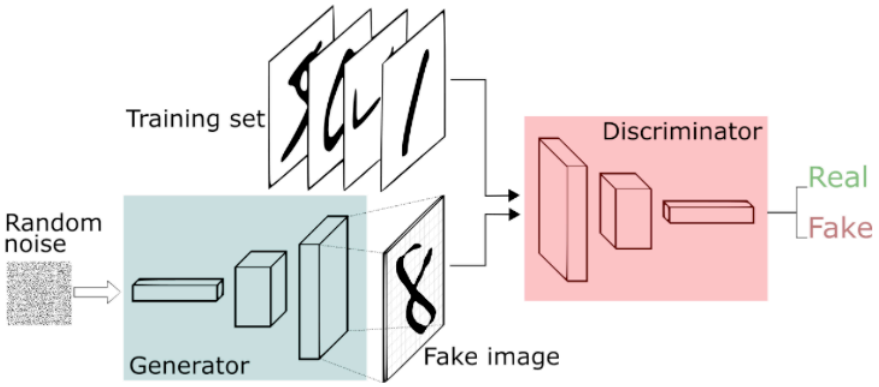

Fig 6. A Generative Adversarial Network combines a Discriminative network with a genrative network. (Image from Thalles Silva[6])

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.

---

[6] https://www.freecodecamp.org/news/an-intuitive-introduction-to-generative-adversarial-networks-gans-7a2264a81394

Bibliography

1. D. H. Ackley, G. E. Hinton and T. J. Sejnowski, A Learning Algorithm for Boltzmann Machines, Cognitive Science, 9, pp 147-168, 1985.
2. Hinton, G. E., and R. Zemel. "Autoencoders, minimum description length and Helmholtz free energy." Advances in neural information processing systems 6, 1993.
3. Cover, T. M. and Thomas, J. A. *Elements of Information Theory*. Wiley Inc., New York, 1991.
4. Kingma, D. P., and M. Welling. "Auto-encoding variational Bayes." arXiv preprint arXiv:1312.6114 2013.
5. Goodfellow, I., J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville and Bengio, Y. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014.
6. Generative adversarial networks: An overview (Creswell et al. 2018), Langr and Bok, GANs in Action: Deep learning with Generative Adversarial Networks (2019)
7. Creswell, A., T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, Generative adversarial networks: An overview. *IEEE signal processing magazine*, 35(1), pp53-65, 2018.