# Pattern Recognition and Machine Learning

James L. Crowley

ENSIMAG 3 - MMIS — Fall Semester 2018
Lessons 9 — 25 Jan 2019

# Deconvolution, Autoencoders and GANs

**Outline**

## Support Papers

Simonyan, K. and Zisserman, A., Very deep convolutional networks for large-scale image recognition. In Proc. International Conference on Learning Representations. ICLR, 2014.

Noh, H., Hong, S. and Han, B. Learning deconvolution network for semantic segmentation. In ICCV 2015 (pp. 1520-1528), 2015.

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.

## 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 $l$. |
| $\rho$ | The sparsity parameter |

### Key Equations

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

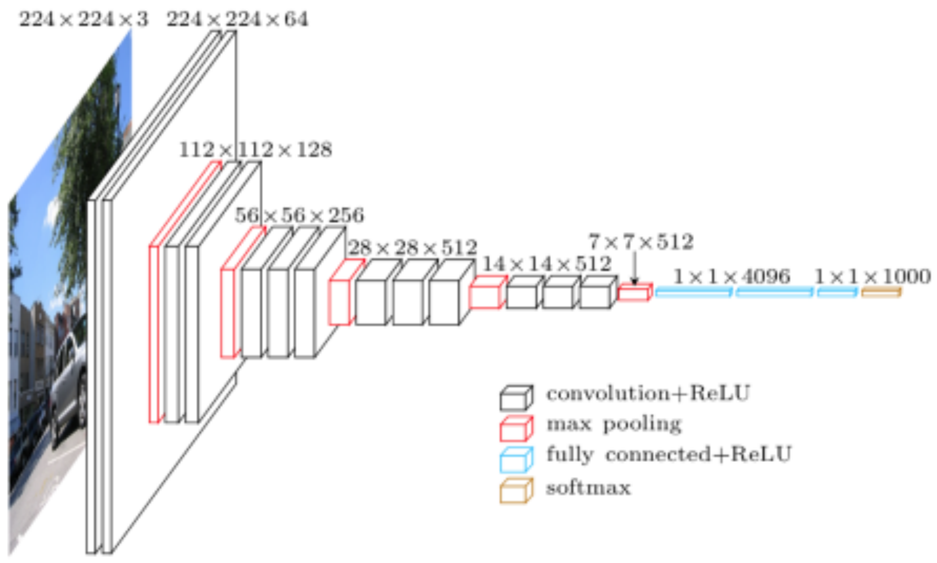
The autoencoder cost function: $L_{sparse}(W,B;\vec{X}_m,y_m) = \frac{1}{2}(\vec{a}_m^{(2)} - \vec{X}_m)^2 + \beta\sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j)$

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

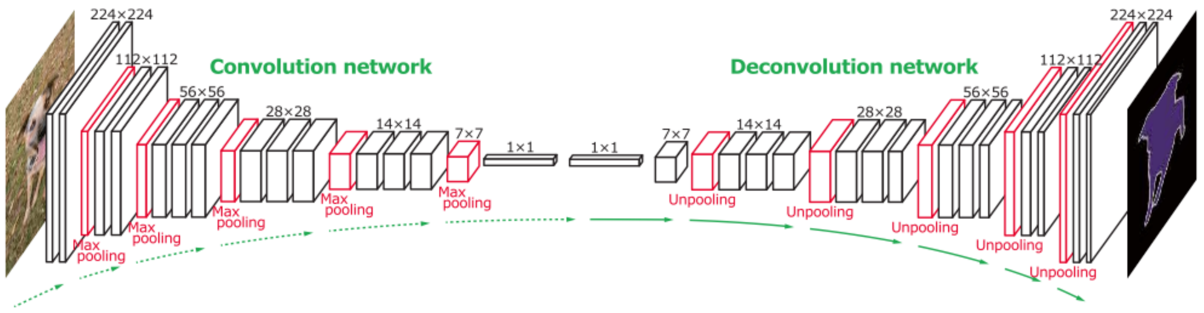
## Deconvolution

Deconvolution allows display of the part of an image that has been recognized by a convolutional net. This provides a coarse pixel-wise label map for recognized classes, that segments the image into regions corresponding to recognized classes. The following is an example from a 16 layer VGG network (taken from noh et al 2015)

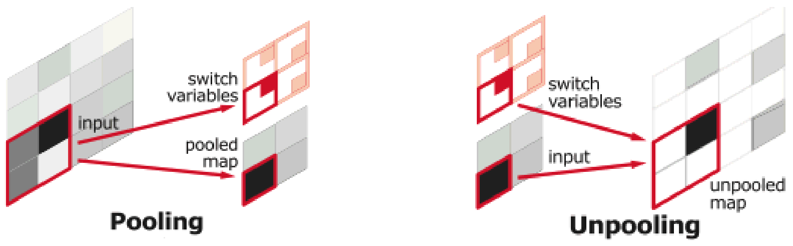
Recall that VGG processes a 224x224 window of an image with 19 layers. The first 17 layers use 3x3 convolutions, relu and 2x2 max pooling after layers 2, 4, 7, 10 and 13. The depths are D=64 (layers 1, 2), D=128 (layers 3, 4, 5), D=256 (layers 6, 7,8,9). D=512 (layers 10 to 17). Layer 18 is a 1 x 1 convolution with depth 4096. Layer 19 is 1 x 1 x 1000.



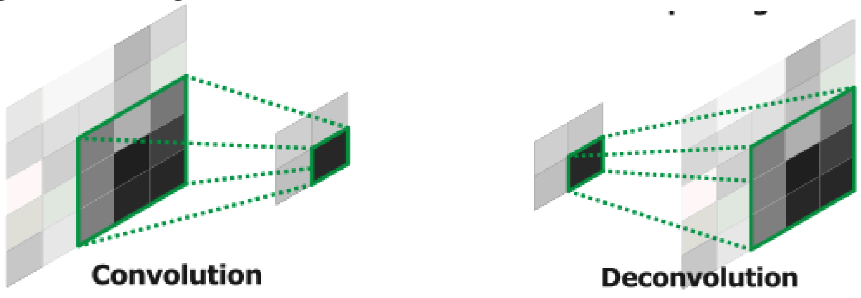
The deconvolution network is a mirror image, replacing pooling with "un-pooling" and convolution with "deconvolution".



With Max pooling, unpooling requires remembering which unit was selected for each pooling operation. This is done with a "switch Variable" that records the selected unit. The output is a larger sparse layer in which 3/4 of the activations are zero.



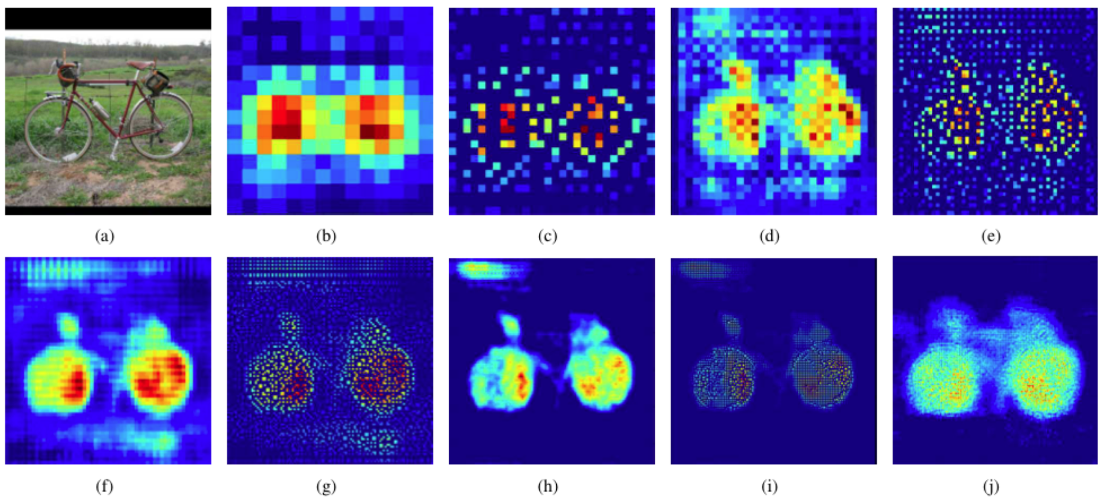
Deconvolution multiplies each non-zero activation by the learned receptive field. These are then summed to create the fully activated layer. The boundary is cropped to obtain an image at the original window size.



De-convolution treats the learned receptive field as basis functions, and uses the un-pooled weights as the amplitude for each basis.

The following shows an example with deconvolution of the VGG net of a bicycle.

(a) is the original image. The other images show the results of max-pooling for the 14x14, 28x28, 56x56, 112x112, and 224x224 layers

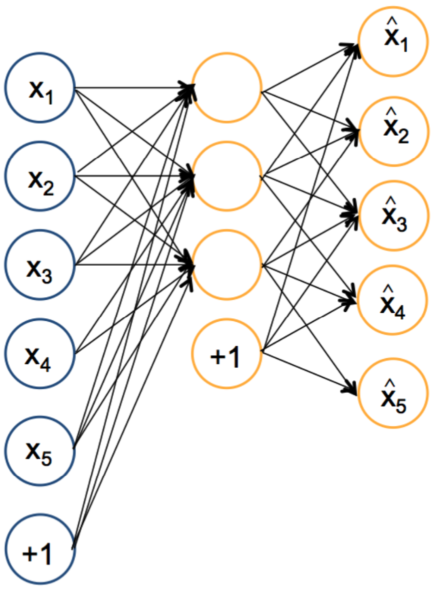


## AutoEncoders

We can use an auto-encoder to learn a set of K receptive fields, $w_k(u,v)$ for a data set for use with a convolutional neural network.

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.

The auto-encoder provides a limited basis set for reconstruction.
Mathematically, the auto-encoder maps the input signal (or image) onto a manifold.



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 error for back-propagation for each unit is $\delta_{k,m}^{(2)} = a_{k,m}^{(2)} - x_{i,m}$
For each component $x_{i,m}$ of the training sample $\vec{X}_m$

## The Sparsity Parameter

The auto-encoder will learn weights subject to a sparseness constraints specified by a sparsity parameter $\hat{\rho}_j = \rho$, typically set close to zero. The sparsity parameter $\rho$ is the average activation for the hidden units.

The auto-encoder is described by:

Level 0: $\vec{X}_m = \begin{pmatrix} x_{1,m} \\ \vdots \\ x_{D,m} \end{pmatrix}$ Composed of window extracted from *P(c,r)*

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

level 2: $a_{k,m}^{(2)} = f(\sum_{j=1}^{N^{(1)}} w_{jk}^{(2)} a_{j,m}^{(1)} + b_k^{(2)})$

Desired output $\vec{a}_m^{(2)} = \begin{pmatrix} a_1^{(2)} \\ \vdots \\ a_D^{(2)} \end{pmatrix} = \hat{X}_m \approx \vec{X}_m$, with error $\delta_{k,m}^{(2)} = a_{k,m}^{(2)} - x_{i,m}$

The average activation $\hat{\rho}_j$ is computed as the average activation for each of the $N^{(1)}$ hidden units, *j=1 to* $N^{(1)}$ for the M training samples:

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

The auto-encoder can be learned by back-propagation using a minor change to the cost function.

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

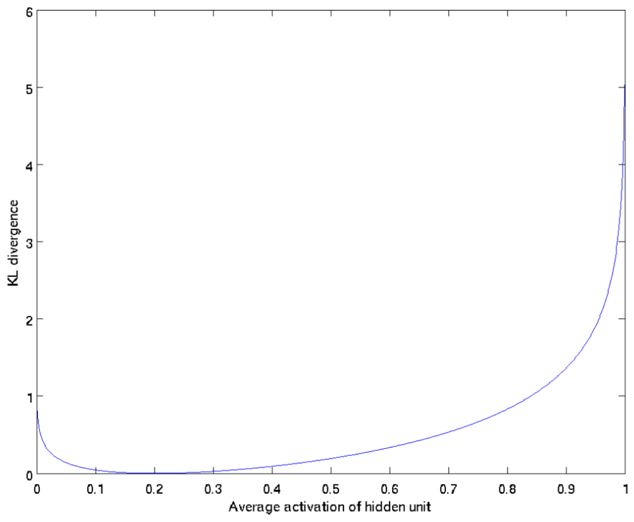
where $\sum_{j=1}^{N^{(1)}} KL(\rho \,||\, \hat{\rho}_j)$ is the Kullback-Leibler Divergence of the hidden unit activations and $\beta$ controls the weight of the sparsity parameter.

(Don't panic - this is easy to do).

## Kullback-Leibler Divergence

The KL divergence between the desired and average activation is:

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



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

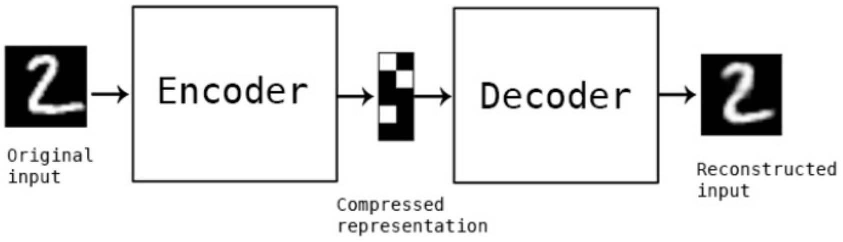
$$\delta_j^{(1)} = \frac{\partial f(z_j^{(1)})}{\partial z_j^{(1)}} \left( \sum_{k=1}^{N^{(2)}} w_{jk}^{(2)} \delta_k^{(2)} + \beta \left( -\frac{\rho}{\hat{\rho}_j} + \frac{1-\rho}{1-\hat{\rho}_j} \right) \right)$$

where $N^{(2)} = D$.

Note you need the average activation $\hat{\rho}_j$ to compute the correction. Thus you need to compute a forward pass on all the training data, before computing the back-propagation on any of the training samples. This can be a problem if the number of training samples is large.

The auto-encoder forces the hidden units to become approximately orthogonal, allowing a small correlation determined by $\rho$. Thus the hidden units act as a form of basis space for the input vectors.

Autoencoders encode input signals as a sum of basis vectors. The values of the basis vectors are referred to as latent variables. The latent varibles provide a compressed representation that reduces dimensionality and eliminates random noise. The output of an autoencoder can be used to drive a decoder to produce a filtered version of the input.



The simplest form of decoder would be a one layer weighted sum of latent variables times the basis vectors.  A better technique is to learn the decoder as network using back-propagation.

# Generative Adversarial Networks.

## Generative Networks

Given an observable random variable $\vec{X}$, and a target variable, $\vec{Y}$, a generative model is a joint probability distribution, $P(\vec{X},\vec{Y})$.

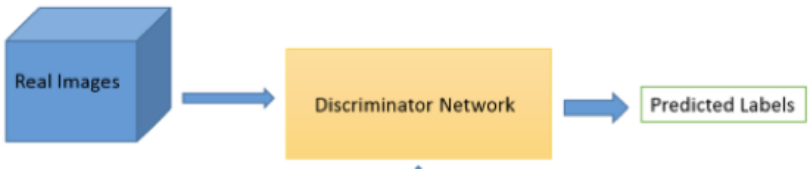
A discriminative model is a model is a conditional probability distribution $P(\vec{Y} \mid \vec{X}=\vec{x})$. In classification, $\vec{X}$, is generally composed of continuous variables, and $\vec{Y}$ is generally a discrete set of classes.

The classical Bayesian approach relies on estimating $P(\vec{X},\vec{Y})$ using Bayes rule:

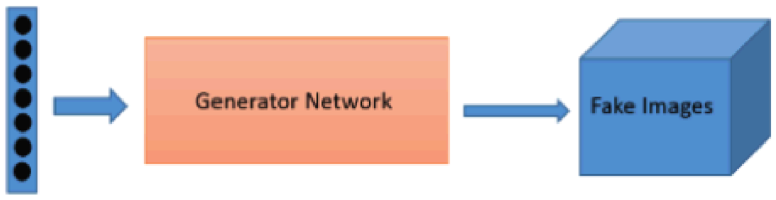
$$P(\vec{Y} \mid \vec{X})p(\vec{X}) = P(\vec{Y},\vec{X}) = p(\vec{X} \mid)P(\vec{Y})$$

The networks seen in the previous lectures learn discriminative models using back-propagation for gradient descent to estimate $P(\vec{Y} \mid \vec{X}=\vec{x})$.

Discriminative Neural networks take a signal as an input and output the likelihood the one or more target classes are in the signal



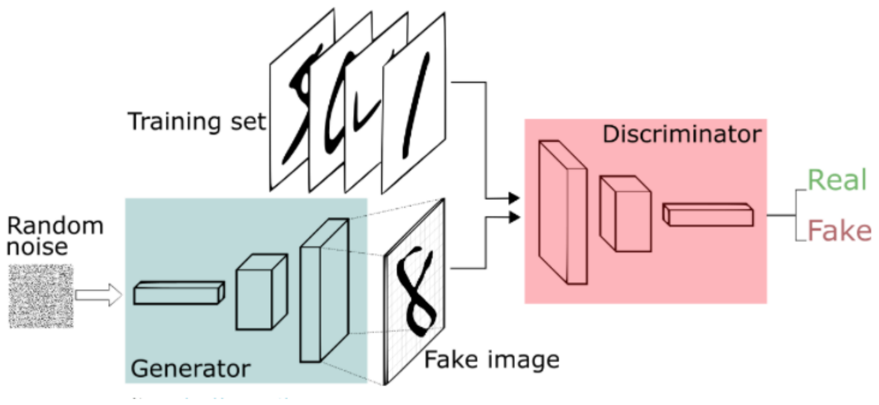
A Generative network runs the other way, generating a realistic fake signal from a random (or arbitrary) input.



The generative process back propagates derivatives using back-propagation to learn a generator network.

## Generative Adversarial Networks

A Generative Adversarial Network (GAN) places a generative network in competition with an 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, $\theta_g$, 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 $\theta_d$ that estimates the likelihood that $\vec{X}$ belongs to the set described by the model $\theta_d$.

The generator $\hat{\vec{X}} = G(\vec{z}, \theta_g)$ 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 :

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