Pattern Recognition and Machine Learning

James L. Crowley

ENSIMAG 3 - MMIS | Fall Semester 2019
Lessons 10 | 22 Jan 2020

Autoencoders, Deconvolution and GANs

Outline

Notation 2
Key Equations 2
Generative Networks 3
AutoEncoders 4
The Sparsity Parameter 5
Kullback-Leibler Divergence 6
Variational Autoencoders 8
Deconvolution 9
Locating objects in images with deconvolution 10
Generative Adversarial Networks. 12
Generative Networks 12
GAN Learning as Min-Max Optimization. 12

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.
- Radford A, Metz L, Chintala S. Unsupervised representation learning with deep convolutional generative adversarial networks. arXiv preprint arXiv:1511.06434. 2015 Nov 19.

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, $\vec{X}$.

$$\vec{X} \longrightarrow \boxed{\;D(\vec{X})\;} \longrightarrow \hat{y}$$

In order to minimize the number of errors, we maximized the probability that $P(\hat{y} = y \mid \vec{X})$

We saw that we could use back-propagation to learn the weights and biases for a network to maximizes $P(\hat{y} = y \mid \vec{X})$ for a training set.

The same process can be used to learn a network that generates $\hat{\vec{X}}$ given a code y such that we maximize the $P(\hat{\vec{X}} = \vec{X} \mid y)$. This is called a "generative" process.

$$y \longrightarrow \boxed{\;G(y)\;} \longrightarrow \vec{X}$$

Given an observable random variable $\vec{X}$, and a target variable, $\vec{Y}$, 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 is a conditional probability distribution $P(\vec{Y} \mid \vec{X})$.
A generative model, is a conditional probability $P(\vec{X} \mid \vec{Y})$
The classical Bayesian approach relies on estimating $P(\vec{X},\vec{Y})$ using Bayes rule:

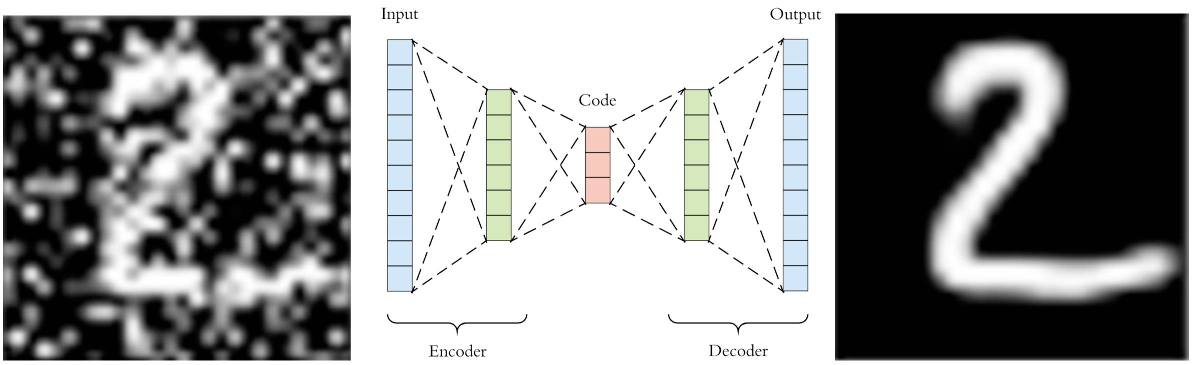
$$P(\vec{Y} \mid \vec{X})p(\vec{X}) = P(\vec{Y},\vec{X}) = p(\vec{X} \mid \vec{Y})P(\vec{Y})$$

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

$$\vec{X} \longrightarrow \boxed{\;D(\vec{X})\;} \longrightarrow \hat{y} \qquad y \longrightarrow \boxed{\;G(y)\;} \longrightarrow \vec{X}$$

A classic example is an autoencoder.

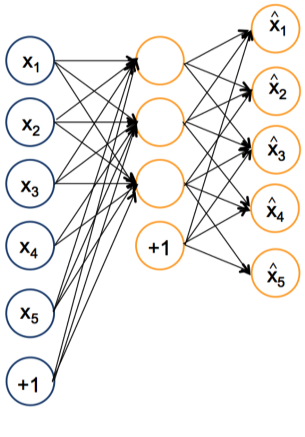
AutoEncoders



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 endodes the data with a minimum number of independent hidden units (Code vectors)

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



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

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

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

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

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

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

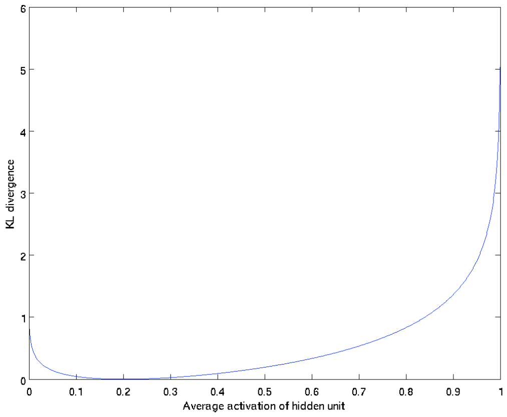
$$L_m(\vec{X}_m, y_m) = \frac{1}{2}(\vec{X}_m - \vec{a}_m^{(L)})^2 + \beta \sum_{j=1}^{N^{(1)}} KL(\rho \,||\, \hat{\rho}_j)$$

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.

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

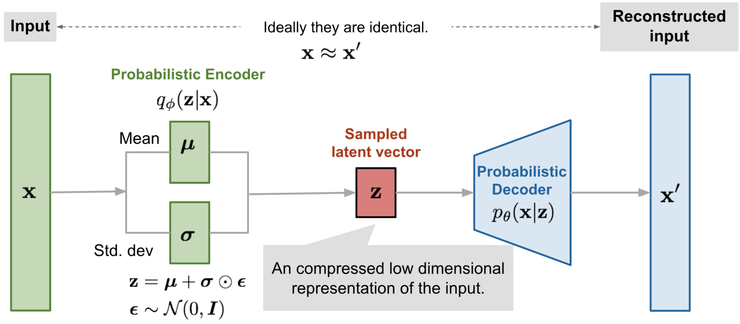
Note you need the average activation $\hat{\rho}_j$ for the training data to compute the correction. Thus you need to compute a forward pass on all the entire batch of training data, before computing the back-propagation on any of the training samples. This can be a problem if the number of training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

The auto-encoder forces the hidden units to become approximately orthogonal, allowing a small correlation determined by 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.

Variational Autoencoders

The output of an autoencoder 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.


Input
Ideally they are identical.
$\mathbf{x} \approx \mathbf{x}'$
Reconstructed input
Probabilistic Encoder
$q_\phi(\mathbf{z}|\mathbf{x})$
Mean
$\boldsymbol{\mu}$
$\mathbf{x}$
$\boldsymbol{\sigma}$
Std. dev
$\mathbf{z} = \boldsymbol{\mu} + \boldsymbol{\sigma} \odot \boldsymbol{\epsilon}$
$\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$
Sampled latent vector
$\mathbf{z}$
An compressed low dimensional representation of the input.
Probabilistic Decoder
$p_\theta(\mathbf{x}|\mathbf{z})$
$\mathbf{x}'$

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.

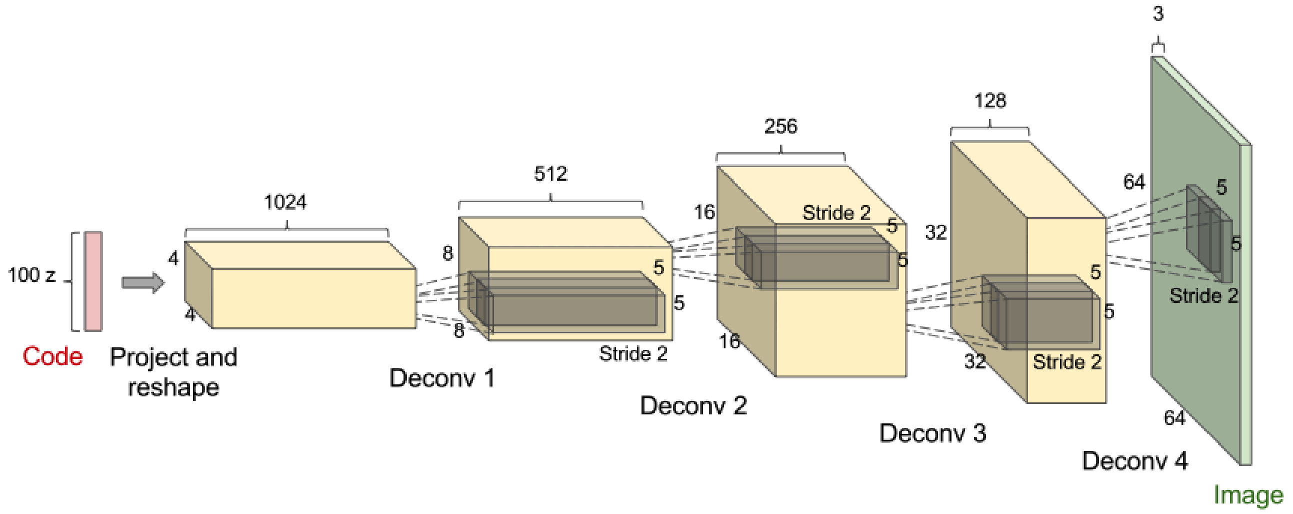
$$\vec{\delta}_m^{(2)} = \vec{a}_m^{(2)} - \vec{X}_m$$

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

However, a generative convolutional network is not so obvious. We need to be able to deconvolve.

Deconvolution

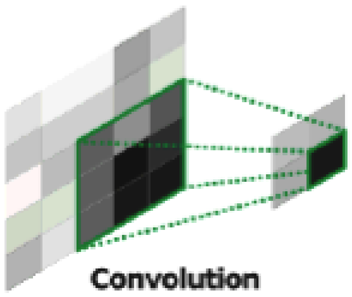
For larger signals such as images, we need to use a convolutional network to avoid the explosion in the number of parameters. For example, the following is the architecture for a network called DCGAN. The network takes 100 random numbers as an input (or code) and outputs an color image of size 64x64x3

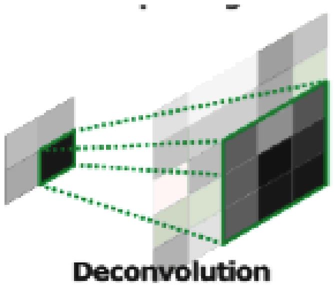


The first fully connected layer is a 4 x 4 array of 1024 cells (Dept = 1024). Total number of cells is 16 K. This layer has 160 K weights and 16 K biases to train.

This is deconvolved into an 8 x 8 by 512 layer, where deconvolution projects each of the cells in the 4x4 layer onto an overlapping set of 5x5 receptive field with a stride of 2. The process is repeated to create a 3rd layer that is 16x16x245 and then a 4th layer that is 32 x 32 by 128. The final output is a 5th layer with 64 x64 pixels of 3 colors.

Deconvolution multiplies each activation at level *l* by the learned receptive field to create an image at level *l+1*. Overlapping projections of receptive fields are then summed to create the layer *l+1*. In some cases, the boundary is cropped to obtain an image at the target window size. Deconvolution uses a stride to create a larger image.





De-convolution treats the learned receptive field as basis functions, and uses the activation at level l to create a weighted sum of bases at level $l+1$.

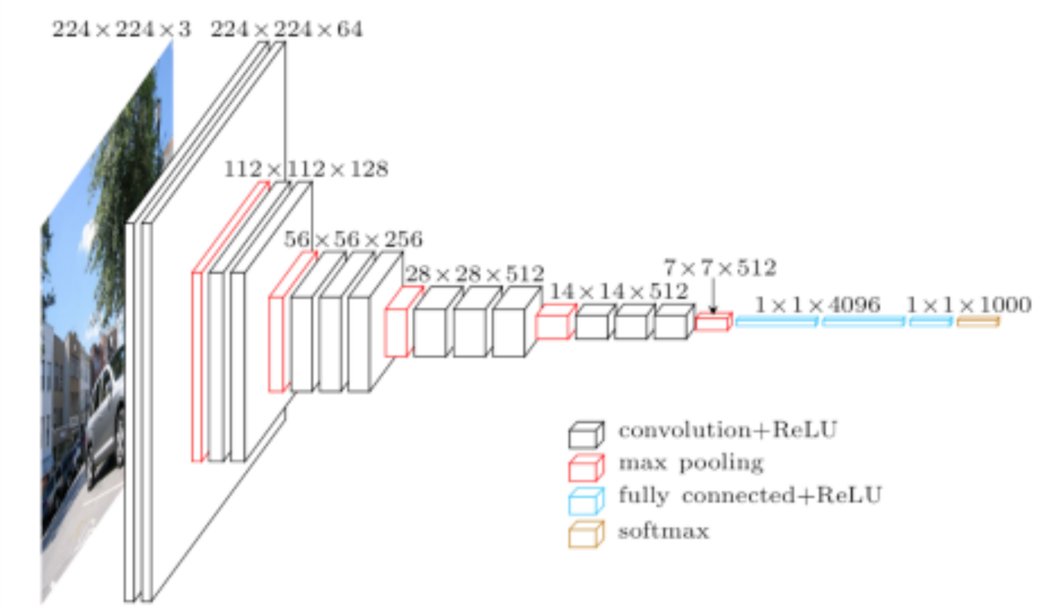
The stride acts as the opposite of pooling. For 2x2 average pooling, de-convolution simply projects 4 displaced copies of the receptive field onto a 2 x 2 grid of overlapping receptive fields. These are then summed to give an image.

Locating objects in images with deconvolution.

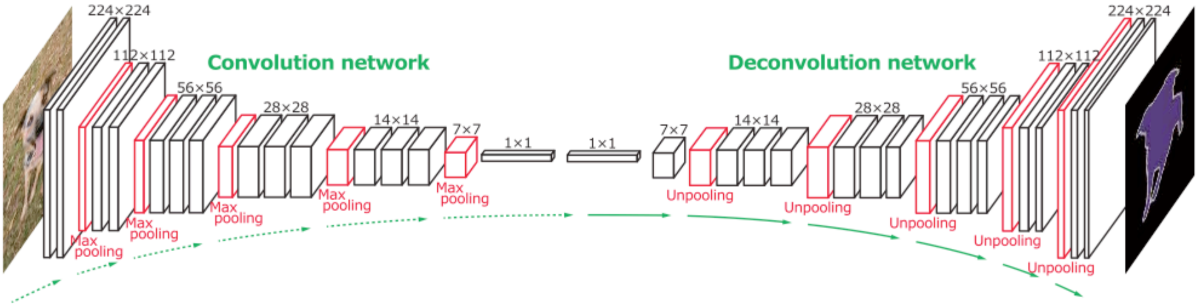
Deconvolution is often used with convolutional networks to determine the location of a detected pattern in an image. Deconvolution allows display of the part of an image that has been recognized by a convolutional net. This provides a coarse pixel-wise label map that segments the image into regions corresponding to recognized classes.

The following is an example from a 16 layer VGG network (taken from noh et al 2015)

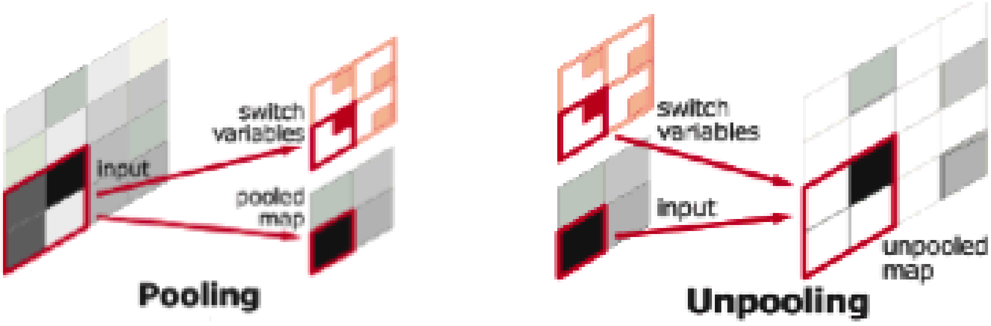
Recall that VGG processes a 224x224 window of an image with 19 layers. The first 17 layers use 3x3 convolutions, relu and 2x2 max pooling after layers 2, 4, 7, 10 and 13. The depths are D=64 (layers 1, 2), D=128 (layers 3, 4, 5), D=256 (layers 6, 7,8,9). D=512 (layers 10 to 17). Layer 18 is a 1 x 1 convolution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

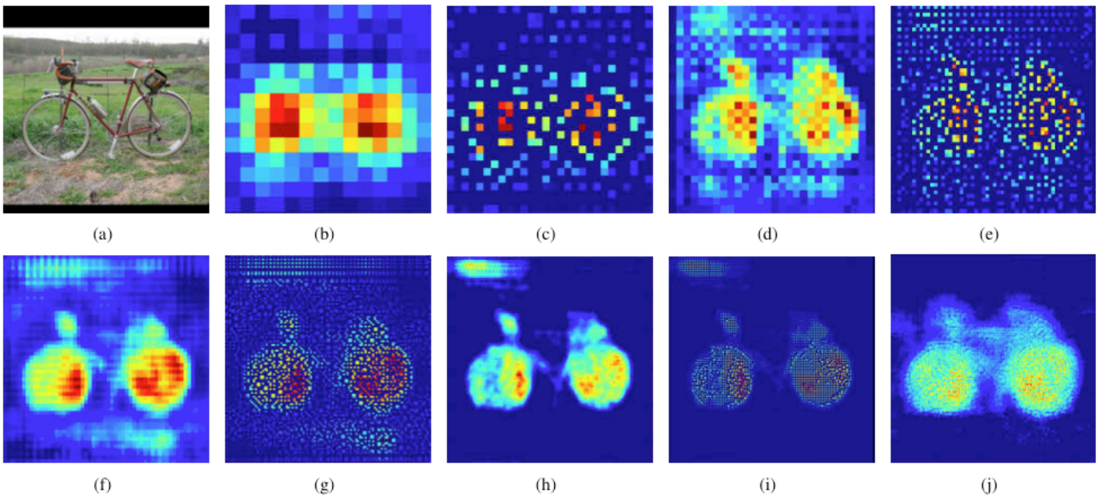


VGG uses max pooling. With Max pooling, unpooling requires remembering which unit was selected for each pooling operation. This is done with a "switch Variable" that records the selected unit. The output is a larger sparse layer in which 3/4 of the activations are zero.



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

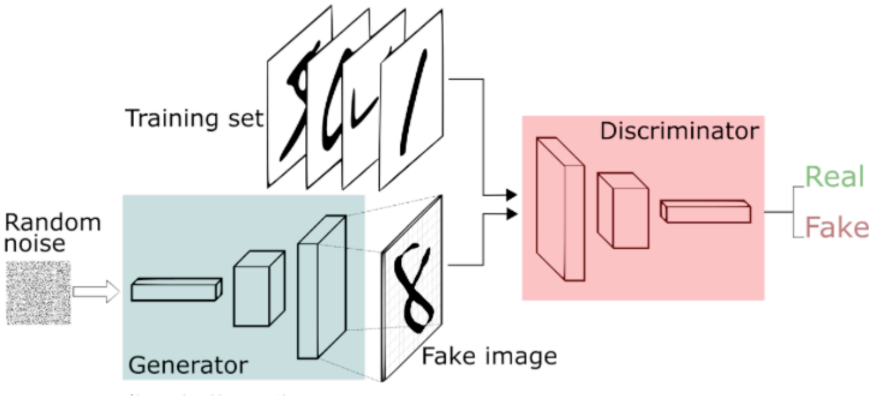


Generative Adversarial Networks.

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

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