Pattern Recognition and Machine Learning

James L. Crowley

ENSIMAG 3 - MMIS Fall Semester 2018
Lessons 8 10 Jan 2018

Artificial Neural networks

Outline

Using notation and figures from the Stanford Deep learning tutorial at:
http://ufldl.stanford.edu/tutorial/

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.

Key Equations

Feed Forward from Layer i to j:
$$a_j^{(l)} = f\left(\sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}\right)$$

Feed Forward from Layer j to k:
$$a_k^{(l+1)} = f\left(\sum_{j=1}^{N^{(l)}} w_{jk}^{(l+1)} a_j^{(l)} + b_k^{(l+1)}\right)$$

Back Propagation from Layer j to i:
$$\delta_{i,m}^{(l-1)} = \frac{\partial f(z_i^{(l-1)})}{\partial z_i^{(l-1)}} \sum_{j=1}^{N^{(l)}} w_{ij}^{(l)} \delta_{j,m}^{(l)}$$

Back Propagation from Layer k to j:
$$\delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{(l+1)}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)}$$

Weight and Bias Corrections for layer j:
$$\Delta w_{ij,m}^{(l)} = a_i^{(l-1)} \delta_{j,m}^{(l)}$$
$$\Delta b_{j,m}^{(l)} = \delta_{j,m}^{(l)}$$

Network Update Formulas:
$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \cdot \Delta w_{ij,m}^{(l)}$$
$$b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \Delta b_{j,m}^{(l)}$$

Convolutional Neural Networks.

Convolutional Neural Networks take inspiration from the Receptive Field model of biological vision systems proposed by Hubel and Weisel in 1968 to explain the organization of the visual cortex.

Fully connected Networks.

A fully connected network is a network where each unit at level $l+1$ receives activations from all units at level l.

If there are $N^{(l)}$ units at level l and $N^{(l+1)}$ units are level $l+1$ then a fully connected network requires learning $N^{(l)} \cdot N^{(l+1)}$ parameters. While this may be tractable for small examples, it quickly becomes excessive for practical problems, as found in computer vision or speech recognition.

For example, a typical image may have 1024 x 2048 = 2^{21} pixels. If we assume, say a 512 x 512 =2^{18} hidden units we have 2^{39} parameters to learn for a single class of image pattern. Clearly this is not practical (and, in any case unnecessary)

A common solution is to perform learning using a limited size window, and to use all possible windows as training data. This leads to a technique where we fix a window size at NxN input units and use all possible, overlapping, windows of size NxN from our training data to train the network.

We then use the same learned weights with every hidden cell. The resulting operation is equivalent to a “convolution” of the learned weights with the input signal and the learned weights are referred to as “receptive fields” in the neural network literature.

Local and Stationary Signals

Convolutional Neural Networks (CNNs) are used to interpret image and speech signals because both images and speech signals have two interesting theoretical properties: They are local and stationary.

1) Local. Local means that (most of) the required information can be found within a limited sized neighborhood of the signal. In fact, image information tends to be multi-scale, but this can be easily accommodated using multi-scale signal techniques using a scale invariant pyramid. Such a representation is "local" at multiple scales, with low-resolution scales providing context for higher resolution. This can be referred to as "multi-local".

2) Stationary. A stationary signal is a random (unknown) signal whose joint probability density function does not change when shifted in time (speech) or space (image). Image and Speech signals tend to have stationary statistics. Thus the same processing can be applied to every possible (overlapping) window.

There are exceptions to both rules, but these can be handled with established techniques.

What Window Size?

What window size should be used for a feature in a Convolutional Neural Network? This tends to depend on the problem. It is not uncommon to see tutorials proposal 5 x 5 image windows. The impressive results in category learning were obtained with a 2D image window of 11 x 11. It is common for authors to use 3x3 or 5x5. Most authors test a range of sizes and discover which works best.

Convolutional Neural Network for an image.

Convolutional Neural Network (CNN) can be used as feature detectors for image analysis. When used with images, a CNN provides K features at each pixel using convolution with K receptive fields. Each feature will be computed as a weighted sum of the pixels within an N x N window for each position in the level below.

Let us assume our input feature vector, $\vec{X}$, is an image of R rows and C columns $P(c,r)$. Note that we can always "flatten" the image by mapping the pixel, $P(i,j)$, onto a vector component x_d using

$$x_d = P(c,r) \text{ where } d = r \cdot C + c$$

However, such a mapping is not at all necessary. It will be more useful, to visualize the input vector a 2D image.

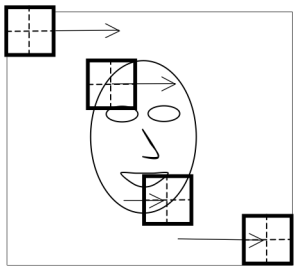
Note that in general, the image will be a color image. In this case, each pixel has 3 color values. Each pixel *(c,r)* is a color vector, $\vec{P}(c,r)$, represented by 3 integers between 0 and 255 representing Red, Green and Blue.

$$\vec{P}(c,r) = \begin{pmatrix} R \\ G \\ B \end{pmatrix}$$

In the literature on CNNs, the colors are referred to as "channels", and the number of channels is called the Depth.

The CNN will describe each possible NxN window by multiplying by K filters (or kernels) $w_k(u,v)$, of size NxN. If the image is a D valued color image, then each filter is a tensor of size NxNxD, $\vec{w}_k(u,v)$. To keep things simple, let us assume a "black and white" image composed only gray values from 0 to 255. (8 bits per pixel).

The CNN will independently describe the large set of overlapping NxN windows ranging from the upper left corner of the image to the lower right corner. Let us refer to each such window as $R_{cr}(u,v)$



If we consider the position of the window as its upper left corner, then for each position from *c=1, r=1 to c =C-N+1, r=R-N+1*:

$$R_{cr}(u,v) = P(c+u-1, r+v-1) \quad \textit{for u, v from (1, 1), to (N,N).}$$

The K filters area applied to each such window as a vector product, followed by a non-linear decision function, resulting in an activation at each position (i,j). We could write this as:

$$a_k(c,r) = f(\sum_{u,v} w_k(u,v) R_{cr}(u,v) + b_k^{(1)})$$

Note that written as a convolution, the formula would be

$$a_k(c,r) = f(\sum_{u,v} w_k(u,v) P(c-u, r-v) + b_k)$$

Note that when written as a convolution, we no longer have need for the "window" symbol *R(u,v)*. The K filters are directly applied at each image position.

The result is a "feature map" of k features at each position $a_k(c,r)$, with k values at each position *(c,r)*

The receptive fields, $w_k(u,v)$ can be learned using back-propagation, from a training set where each window is labeled with a target class, using an "indicator" image *y(c,r)*. For multiple target classes, the indicator image is a vector image, $\vec{y}(c,r)$. More classically, *y(c,r)* is a binary image with 1 at each location that contains the target class and 0 elsewhere.

Hyperperameters:

CNNs are typically configured with a number of "hyper-parameters":

Depth: This is the number D of channels for each image pixel. For a color image, this would be D=3. Note that with multiple hidden layers, depth is sometimes used to refer to the number of filters, K, applied at each position.

Stride: Stride is the step size, S, between window positions. By default it may be 1, but for larger windows, it is possible define larger step sizes.

Spatial Extent: This is the size of the filter, NxN.

Zero-Padding: Size of region at the border of the feature map that is filled with zeros in order to preserve the image size (typically N/2).

Pooling

Pooling is a form of non-linear down-sampling that partitions the image into non-overlapping regions and computes a representative value for each region.

Pooling is typically performed over contiguous regions of the image. In this case, the stride equals the pooling window size. The CNN feature image is partitioned into small non-overlapping rectangular regions, typically of size 2x2 or 4x4.

Several non-linear functions can be used. These include Max, Average, Median, and Histograms. Max pooling seems to be the most popular.

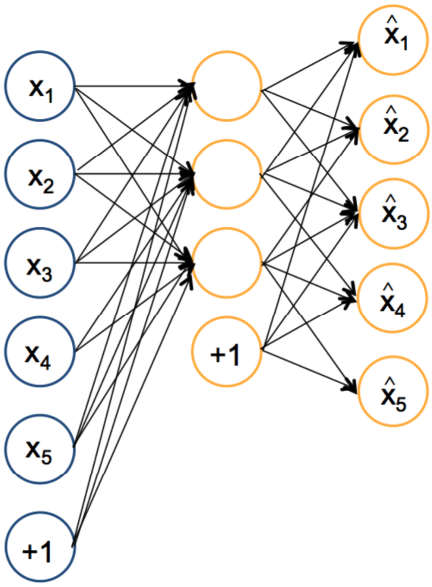
For example, the SIFT operation, in Computer vision, uses local histograms over a 4x4 window.

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, we can say that it 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 ρ 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.

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

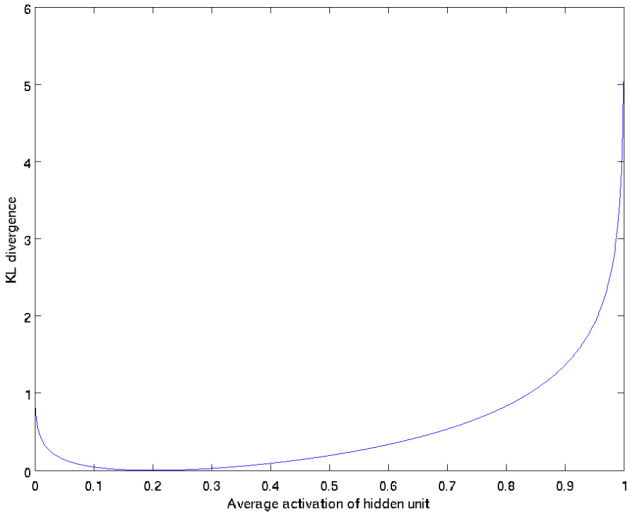
where $\sum_{j=1}^{N^{(1)}} KL(\rho \,\|\, \hat{\rho}_j)$ is the Kullback-Leibler Divergence of the hidden unit activations and β controls the weight of the sparsity parameter.

(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

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

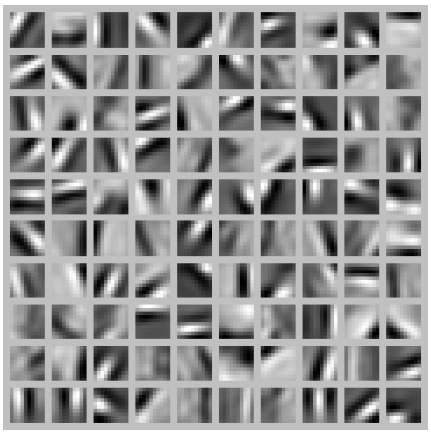
where $N^{(2)} = D$.

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.

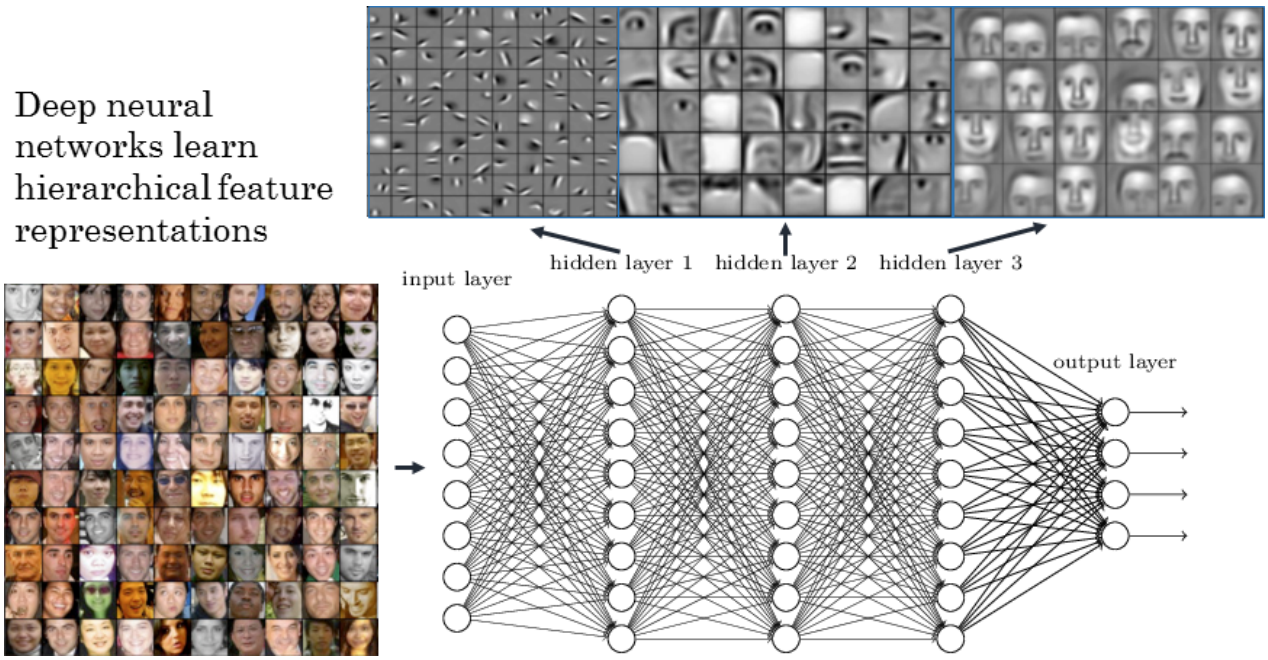
The auto-encoder forces the hidden units to become approximately orthogonal, allowing a small correlation determined by ρ.
Thus the hidden units act as a form of basis space for the input vectors.

Examples of the Hidden Units given by Autoencoders

An example of 100 hidden units learned by a sparse auto-encoder from images:



When applied at multiple levels and trained on face images this can give recognizable features:



or when trained on YouTube videos: Cats

when trained with car images: