Computer Vision

MoSIG options GVR and UIS
James L. Crowley

Fall Semester 23 November 2017

Lesson 5

Artificial Neural Networks, Back-Propagation and CNN

Lesson Outline:

1. Introduction

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

1.2. Artificial Neural Networks

Artificial Neural Networks, also referred to as “Multi-layer Perceptrons”, are computational structures composed a weighted sums of “neural” units. Each neural unit is composed of a weighted sum of input units, followed by a non-linear decision function.

Note that the term “neural” is misleading. The computational mechanism of a neural network is only loosely inspired from neural biology. Neural networks do NOT implement the same learning and recognition algorithms as biological systems.

The approach was first proposed by Warren McCullough and Walter Pitts in 1943 as a possible universal computational model. During the 1950’s, Frank Rosenblatt developed the idea to provide a trainable machine for pattern recognition, called a Perceptron. The perceptron is an incremental learning algorithm for linear classifiers. The first Perceptron, constructed in 1956, was a room-sized analog computer that learned recognition functions. However, both the learning algorithm and the resulting recognition algorithm are easily implemented as computer programs, and future perceptrons were implemented as programs. .

In 1969, Marvin Minsky and Seymour Papert of MIT published a book entitled "Perceptrons", that claimed to document the fundamental limitations of the perceptron approach. Notably, they demonstrated that a one-level perceptron could not be constructed to perform an "exclusive OR".

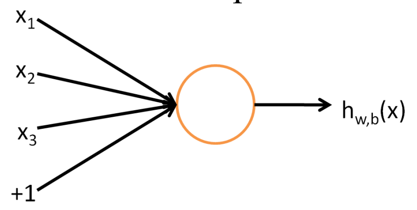
In the 1970s, frustrations with the limits of Artificial Intelligence research based on Symbolic Logic led a small community of researchers to explore the perceptron based approach. In 1973, Steven Grossberg, showed that a two layered perceptron could overcome the problems raised by Minsky and Papert, and solve many problems that plagued symbolic AI. In 1975, Paul Werbos developed an algorithm referred to as "Back-Propagation" that uses gradient descent to learn the parameters for perceptrons from classification errors with training data.

During the 1980's, Neural Networks went through a period of popularity with researchers showing that Networks could be trained to provide simple solutions to problems such as recognizing handwritten characters, recognizing spoken words, and steering a car on a highway. However, results were overtaken by more mathematically sound approaches for statistical pattern recognition such as support vector machines and boosted learning.

In 1998, Yves LeCun showed that convolutional networks composed from many layers could outperform other approaches for recognition problems. Unfortunately such networks required extremely large amounts of data and computation. Around 2010, with the emergence of cloud computing combined with planetary-scale data, training and using convolutional networks became practical. Since 2012, Deep Networks have outperformed other approaches for recognition tasks common to Computer Vision, Speech and Robotics. A rapidly growing research community currently seeks to extend the application beyond recognition to generation of speech and robot actions. Notably, just about any algorithm can be used to train a network, often yielding a solution that executes faster.

1.3. The Artificial Neuron

The simplest possible neural network is composed of a single neuron.



A "neuron" is a computational unit that integrates information from a vector of features, $\vec{X}$, to compute the likelihood of a hypothesis, $h_{w,b}()$

$$a = h_{\vec{w},b}(\vec{X})$$

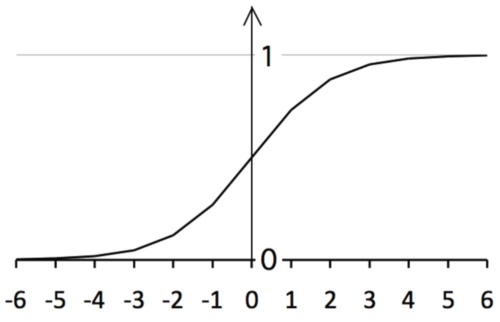
The neuron is composed of a weighted sum of input values

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_D x_D + b$$

followed by a non-linear "activation" function, $f(z)$ (sometimes written $\phi(z)$)

$$a = h_{\vec{w},b}(\vec{X}) = f(\vec{w}^T \vec{X} + b)$$

Many different activation functions may be used.
A popular choice for activation function is the sigmoid: $f(z) = \frac{1}{1+e^{-z}}$



This function is useful because the derivative is: $\frac{df(z)}{dz} = f(z)(1-f(z))$

This gives a decision function: if $h_{\vec{w},b}(\vec{X}) >$ 0.5 POSITIVE else NEGATIVE

Other popular decision functions include the hyperbolic tangent and the softmax.

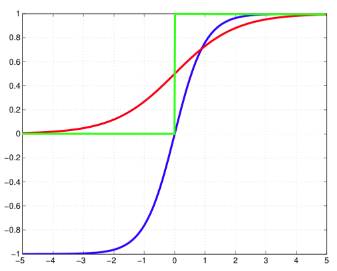
The hyperbolic Tangent: $f(z) = \tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$

The hyperbolic tangent is a rescaled form of sigmoid ranging over [-1,1]

We can also use the step function: $f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{if } z < 0 \end{cases}$

Or the sgn function: $f(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$

Plot of Sigmoid (red), Hyperbolic Tangent (Blue) and Step Function (Green)



The softmax function is often used for multi-class networks. For K classes:

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

The rectified linear function is popular for deep learning because of a trivial derivative:

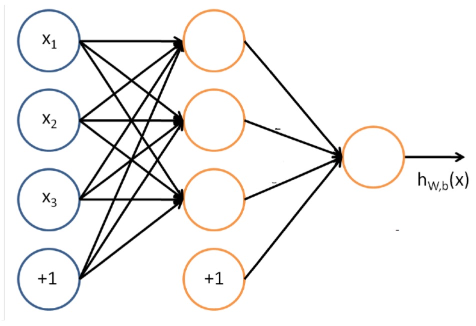
Relu: $f(z) = \max(0, z)$

While Relu is discontinuous at z=0, for $z > 0$: $\frac{df(z)}{dz} = 1$

Note that the choice of decision function will determine the target variable "y" for supervised learning.

1.4. The Neural Network model

A neural network is a multi-layer assembly of neurons. For example, this is a 2-layer network:



The circles labeled +1 are the bias terms.
The circles on the left are the input terms. Some authors, notably in the Stanford tutorials, refer to this as Level 1.

We will NOT refer to this as a level (or, if necessary, level L=0).
The rightmost circle is the output layer, also called L.
The circles in the middle are referred to as a "hidden layer". In this example there is a single hidden layer and the total number of layers is L=2.

The parameters carry a superscript, referring to their layer.

We will use the following notation:

L	The number of layers (Layers of non-linear activations).
l	The layer index. *l* ranges from 0 (input layer) to L (output layer)

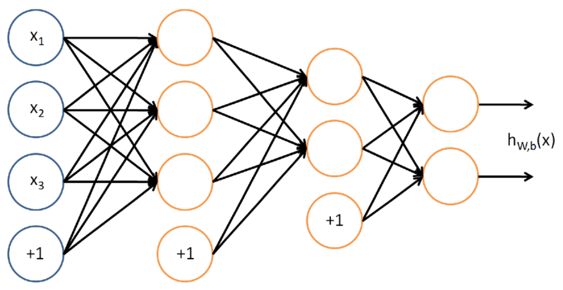
$N^{(l)}$	The number of units in layer *l*. $N^{(0)}=D$
$a_j^{(l)}$	The activation output of the *j*th neuron of the *l*th layer.
$w_{ij}^{(l)}$	The weight from the unit *i* of layer *l-1* for the unit *j* of layer *l*.
$b_j^{(l)}$	The bias term for *j*th unit of the *l*th layer
f(z)	A non-linear activation function, such as a sigmoid, tanh, or soft-max

For example: $a_1^{(2)}$ is the activation output of the first neuron of the second layer.
$W_{13}^{(2)}$ is the weight for neuron 1 from the first level to neuron 3 in the second level.

The above network would be described by:

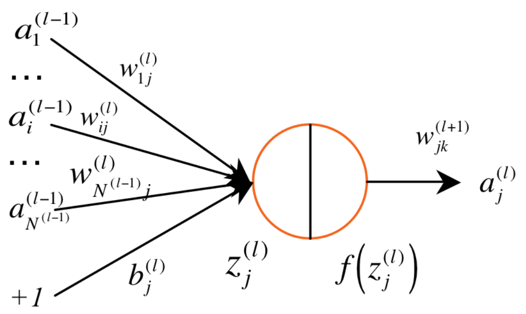
$$a_1^{(1)} = f(w_{11}^{(1)}X_1 + w_{21}^{(1)}X_2 + w_{31}^{(1)}X_3 + b_1^{(1)})$$
$$a_2^{(1)} = f(w_{12}^{(1)}X_1 + w_{22}^{(1)}X_2 + w_{32}^{(1)}X_3 + b_2^{(1)})$$
$$a_3^{(1)} = f(w_{13}^{(1)}X_1 + w_{23}^{(1)}X_2 + w_{33}^{(1)}X_3 + b_3^{(1)})$$
$$h_{\vec{w},b}(\vec{X}) = a_1^{(2)} = f(w_{11}^{(2)}a_1^{(1)} + w_{21}^{(2)}a_2^{(1)} + w_{31}^{(2)}a_3^{(1)} + b_1^{(2)})$$

This can be generalized to multiple layers. For example:



$\vec{h}(\vec{X}_m)$ is the vector of network outputs (one for each class).

Each unit is defined as follows:



The notation for a multi-layer network is

$\vec{a}^{(0)} = \vec{X}$	is the input layer. $a_i^{(0)} = X_d$
l	is the current layer under discussion.
$N^{(l)}$	is the number of activation units in layer *l*. $N^{(0)} = D$
i,j,k	Unit indices for layers *l-1, l* and *l+1*: $i \rightarrow j \rightarrow k$
$w_{ij}^{(l)}$	is the weight for the unit *i* of layer *l-1* feeding to unit *j* of layer *l*.

(We use the subscript is *j,i* to respect matrix notation convention.)

$a_j^{(l)}$ is the activation output of the *j*th unit of the layer *l*

$b_j^{(l)}$ the bias term feeding to unit *j* of layer *l*.

$z_j^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}$ is the weighted input to *j*th unit of layer *l*

f(z) is a non-linear decision function, such as a sigmoid, tanh(), or soft-max

$a_j^{(l)} = f(z_j^{(l)})$ is the activation output for the *j*th unit of layer *l*

In deriving the back-propagation algorithm for learning, we will use

$$z_j^{(l)} = \sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)} \qquad z_k^{(l+1)} = \sum_{j=1}^{N^{(l)}} w_{jk}^{(l+1)} a_j^{(l)} + b_k^{(l+1)}$$

$$a_j^{(l)} = f\left(\sum_{i=1}^{N^{(l-1)}} w_{ij}^{(l)} a_i^{(l-1)} + b_j^{(l)}\right) \qquad a_k^{(l+1)} = f\left(\sum_{j=1}^{N^{(l)}} w_{jk}^{(l+1)} a_j^{(l)} + b_k^{(l+1)}\right)$$

It can be more convenient to represent this using vectors:

$$\vec{z}^{(l)} = \begin{bmatrix} z_1^{(l)} \\ z_2^{(l)} \\ \vdots \\ z_{N_l}^{(l)} \end{bmatrix} \qquad \vec{a}^{(l)} = \begin{bmatrix} a_1^{(l)} \\ a_2^{(l)} \\ \vdots \\ a_{N_l}^{(l)} \end{bmatrix}$$

and to write the weights and bias at each level l as a k by j Matrix,

$$W^{(l)} = \begin{pmatrix} w_{11}^{(l)} & \cdots & w_{1i}^{(l)} & \cdots & w_{1N^{(l-1)}}^{(l)} \\ \vdots & \ddots & \vdots & \iddots & \vdots \\ w_{j1}^{(l)} & \cdots & w_{ji}^{(l)} & \cdots & w_{jN^{(l-1)}}^{(l)} \\ \vdots & \iddots & \vdots & \ddots & \vdots \\ w_{N^{(l)}1}^{(l)} & \cdots & w_{N^{(l)}i}^{(l)} & \cdots & w_{N^{(l)}N^{(l-1)}}^{(l)} \end{pmatrix} \qquad \vec{b}^{(l)} = \begin{pmatrix} b_1^l \\ \vdots \\ b_i^l \\ \vdots \\ b_{N^{(l-1)}}^l \end{pmatrix}$$

(note: To respect matrix notation, we have reversed the order of i and j in the subscripts.)

We can see that the weights are a 3rd order Tensor or vector of matrices, with one matrix for each level, The biases are a matrix (vector of vectors) with a vector for each level.

$$\vec{z}^{(l)} = W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)} \quad \text{and} \quad \vec{a}^{(l)} = f(\vec{z}^{(l)}) = f(W^{(l)}\vec{a}^{(l-1)} + \vec{b}^{(l)})$$

We can assemble the set of matrices $W^{(l)}$ into an 3rd order Tensor (Vector of matrices), W, and represent $\vec{a}^{(l)}$, $\vec{z}^{(l)}$ and $\vec{b}^{(l)}$ as matrices (vectors of vectors): A, Z, B.

So how to do we learn the weights W and biases B?

We could train a 2-class detector from a labeled training set $\{\vec{X}_m\},\{y_m\}$ using gradient descent. For more than two layers, we will need to use the more general "back-propagation" algorithm.

1.5. Backpropagation

Back-propagation adjusts the network the weights $w_{ij}^{(l)}$ and biases $b_j^{(l)}$ so as to minimize an error function between the network output $\vec{h}(\vec{X}_m;W,B)=\vec{a}^{(L)}$ and the target value $\vec{y}_m$ for the M training samples $\{\vec{X}_m\}$, $\{\vec{y}_m\}$.

This is an iterative algorithm that propagates an error term back through the hidden layers and computes a correction for the weights at each layer so as to minimize the error term.

This raises two questions:
1) How do we initialize the weights?
2) How do we compute the error term for hidden layers?

1) How do we initialize the weights?

A natural answer for the first question is to initialize the weights to 0.

By experience this causes problems. If the parameters all start with identical values, then the algorithm can end up learning the same value for all parameters. To avoid this, we initialize the parameters with a small random variable that is near 0, for example computed with a normal density with variance ε (typically 0.01).

$$\forall_{i,j,l}\, w_{ji}^{(l)} = \mathcal{N}(0;\varepsilon) \text{ and } \forall_{j,l}\, b_j^{(l)} = \mathcal{N}(0;\varepsilon) \text{ where } \mathcal{N} \text{ is a sample from a normal density.}$$

An even better solution is provided by Xavier GLORIOT's technique (see course web site on Xavier normalization). However that solution is too complex for today's lecture.

2) How do we compute the error term?

Back-propagation propagates the error term back through the layers, using the weights. We will present this for individual training samples. The algorithm can easily be generalized to learning from sets of training samples (Batch mode).

Given a training sample, $\vec{X}_m$, we first propagate the $\vec{X}_m$ through the L layers of the network (Forward propagation) to obtain a hypothesis $\vec{h}(\vec{X}_m;W,B) = \vec{a}^{(L)}$.

We then compute an error term. In the case, of a multi-class network, this is a vector, with k components, one output for each hypothesis. In this case the indicator vector would be a vector, with one component for each possible class:

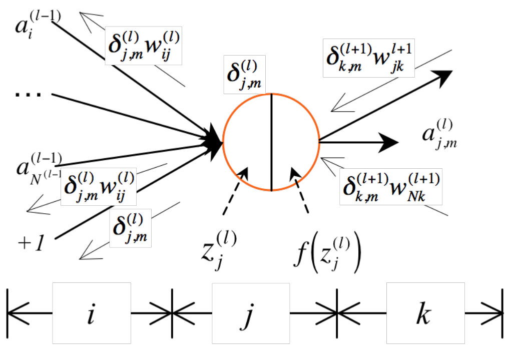
$$\vec{\delta}_m^{(L)} = \left(\vec{a}_m^{(L)} - \vec{y}_m\right) \qquad \text{or for each class k:} \quad \delta_{k,m}^{(L)} = \left(a_{k,m}^{(L)} - y_{k,m}\right)$$

This error term tells how much the unit was responsible for differences between the activation of the network $\vec{h}(\vec{X}_m;w_{jk}^{(l)},b_k^{(l)})$ and the target value $\vec{y}_m$.

To keep things simple, let us consider the case of a two class network, so that $\delta_m^{(L+1)}$, $h(\vec{X}_m)$, $a_m^{(L+1)}$, and y_m are scalars. The results are easily generalized to vectors for multi-class networks. At the output layer, the "error" for each training sample is:

$$\delta_m^{(L)} = \left(a_m^{(L)} - y_m\right)$$

For the hidden units in layers $l \le L$ the error $\delta_j^{(l)}$ is based on a weighted average of the error terms for $\delta_k^{(l+1)}$.


$a_i^{(l-1)}$
$\delta_{j,m}^{(l)} w_{ij}^{(l)}$
$\delta_{j,m}^{(l)}$
$\delta_{k,m}^{(l+1)} w_{jk}^{l+1}$
. . .
$a_{j,m}^{(l)}$
$a_{N^{(l-1)}}^{(l-1)}$
$\delta_{j,m}^{(l)} w_{ij}^{(l)}$
$\delta_{k,m}^{(l+1)} w_{Nk}^{(l+1)}$
$+1$
$\delta_{j,m}^{(l)}$
$z_j^{(l)}$
$f\left(z_j^{(l)}\right)$
i
j
k

We compute error terms, $\delta_j^{(l)}$ for each unit j in layer l back to $l = l–1$ using the sum of errors times the corresponding weights times the derivative of the activation function.

$$\delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{l+1}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)} \qquad \delta_{i,m}^{(l-1)} = \frac{\partial f(z_i^{(l-1)})}{\partial z_i^{(l-1)}} \sum_{j=1}^{N^{(l)}} w_{ij}^{(l)} \delta_{j,m}^{(l)}$$

For the sigmoid activation function. $f(z) = \frac{1}{1+e^{-z}}$ the derivative is: $\frac{df(z)}{dz} = f(z)(1-f(z))$

For $a_j^{(l)} = f(z_j^{(l)})$ this gives:

$$\delta_{j,m}^{(l)} = a_{j,m}^{(l)}(1 - a_{j,m}^{(l)}) \cdot \sum_{k=1}^{N^{(l+1)}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)}$$

This error term can then used to correct the weights and bias terms leading from layer *j* to layer *i*.

$$\Delta w_{ij,m}^{(l)} = a_i^{(l-1)} \delta_{j,m}^{(l)}$$
$$\Delta b_{j,m}^{(l)} = \delta_{j,m}^{(l)}$$

Note that the corrections $\Delta w_{ij,m}^{(l)}$ and $\Delta b_{j,m}^{(l)}$ are NOT applied until after the error has propagated all the way back to layer *l=1*, and that when *l=1,* $a_i^{(0)} = x_i$.

For "batch learning", the corrections terms, $\Delta w_{ji,m}^{(l)}$ and $\Delta b_{j,m}^{(l)}$are averaged over M samples of the training data and then only an average correction is applied to the weights.

$$\Delta w_{ij}^{(l)} = \frac{1}{M}\sum_{m=1}^{M} \Delta w_{ij,m}^{(l)} \qquad \Delta b_{j}^{(l)} = \frac{1}{M}\sum_{m=1}^{M} \Delta b_{j,m}^{(l)}$$

then

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \cdot \Delta w_{ij}^{(l)} \quad b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \Delta b_j^{(l)}$$

where η is the learning rate.

Back-propagation is equivalent to computing the gradient of the loss function for each layer of the network. A common problem with gradient descent is that the loss function can have local minimum. This problem can be minimized by regularization. A popular regularization technique for back propagation is to use "momentum"

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \cdot \Delta w_{ij}^{(l)} + \mu \cdot w_{ij}^{(l)}$$
$$b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \Delta b_j^{(l)} + \mu \cdot b_j^{(l)}$$

where the terms $\mu \cdot w_j^{(l)}$ and $\mu \cdot b_j^{(l)}$ serves to stabilize the estimation.

The back-propagation algorithm may be continued until all training data has been used. For batch training, the algorithm may be repeated until all error terms, $\delta_{j,m}^{(l)}$, are a less than a threshold.

1.6. Summary of Backpropagation

The Back-propagation algorithm can be summarized as:

1) Initialize the network and a set of correction vectors:

$$\forall_{i,j,l}\ w_{ji}^{(l)} = \mathcal{N}(0;\varepsilon)$$
$$\forall_{i,l}\ b_j^{(l)} = \mathcal{N}(0;\varepsilon)$$
$$\forall_{i,j,l}\ \Delta w_{ji}^{(l)} = 0$$
$$\forall_{i,l}\ \Delta b_j^{(l)} = 0$$

where $\mathcal{N}$ is a sample from a normal density, and ε is a small value.

2) For each training sample, $\vec{X}_m$, propagate $\vec{X}_m$ through the network (forward propagation) to obtain a hypothesis $\vec{h}(\vec{X}_m;W,B)$. Compute the error and propagate this back through the network:

a) Compute the error term: $\delta_m^{(L)} = \left(h(\vec{X}_m) - y_m\right) = \left(a_m^{(L)} - y_m\right)$

b) Propagate the error back from $l=L-1$ to $l=1$:

$$\delta_{j,m}^{(l)} = \frac{\partial f(z_j^{(l)})}{\partial z_j^{(l)}} \sum_{k=1}^{N^{(l+1)}} w_{jk}^{(l+1)} \delta_{k,m}^{(l+1)} \qquad \delta_{i,m}^{(l-1)} = \frac{\partial f(z_i^{(l-1)})}{\partial z_i^{(l-1)}} \sum_{j=1}^{N^{(l)}} w_{ji}^{(l)} \delta_{j,m}^{(l)}$$

c) Use the error to set a vector of correction weights.

$$\Delta w_{ij,m}^{(l)} = a_i^{(l-1)} \delta_{j,m}^{(l)} \qquad \Delta b_{j,m}^{(l)} = \delta_{j,m}^{(l)}$$

3) For all layers, *l=1 to L*, update the weights and bias using a learning rate, η

$$w_{ij}^{(l)} \leftarrow w_{ij}^{(l)} - \eta \cdot \Delta w_{ij,m}^{(l)} + \mu \cdot w_{ij}^{(l)}$$
$$b_j^{(l)} \leftarrow b_j^{(l)} - \eta \cdot \Delta b_{j,m}^{(l)} + \mu \cdot b_j^{(l)}$$

Note that this last step can be done with an average correction matrix obtained from many training samples (Batch mode), providing a more efficient algorithm.

2. 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.

2.1. 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.

2.2. 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 <u>local</u> and <u>stationary</u>.

1) <u>Local</u>. 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) <u>Stationary</u>. 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.

2.3. 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.

2.4. 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.

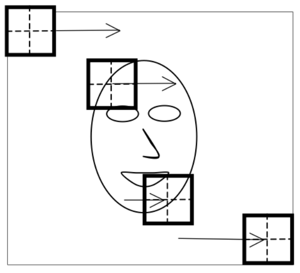
Note that in general, the image will be a color image. In this case, each pixel has 3 color values. Each pixel *(c,r)* is a color vector, $\vec{P}(c,r)$, represented by 3 integers between 0 and 255 representing Red, Green and Blue.

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

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*:

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

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:

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

Note that written as a convolution, the formula would be

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

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).

2.5. 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.

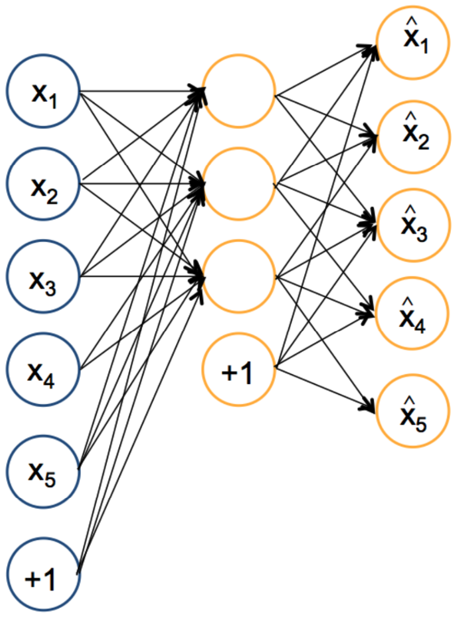
For example, the SIFT operation, in Computer vision, uses local histograms over a 4x4 window.

3. 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 $\vec{X}_m$

3.1. 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: $a^{(1)}_{j,m} = f(\sum_{i=1}^{D} w^{(1)}_{ij} x_{i,m} + b^{(1)}_j)$

level 2: $a^{(2)}_{k,m} = f(\sum_{j=1}^{N^{(1)}} w^{(2)}_{jk} a^{(1)}_{j,m} + b^{(2)}_k)$

Desired output $\vec{a}^{(2)}_m = \begin{pmatrix} a^{(2)}_1 \\ \vdots \\ a^{(2)}_D \end{pmatrix} = \hat{X}_m \approx \vec{X}_m$, with error $\delta^{(2)}_{k,m} = a^{(2)}_{k,m} - x_{i,m}$

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:

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

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

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

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

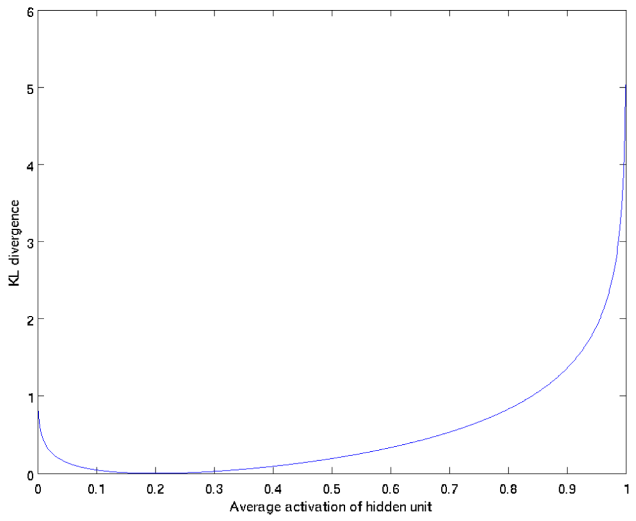
and β controls the weight of the sparsity parameter.

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

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

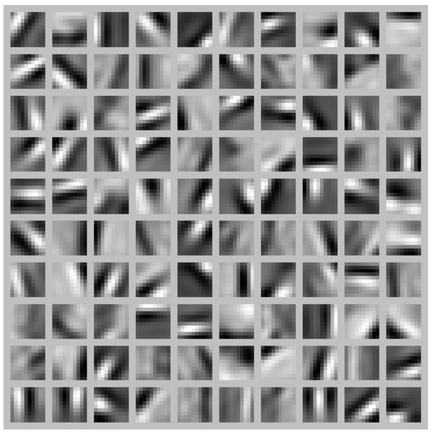
Note you need the average activation $\hat{\rho}_j$ to compute the correction. Thus you need to compute a forward pass on all the training data, before computing the back-propagation on any of the training samples. This can be a problem if the number of training samples is large.

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

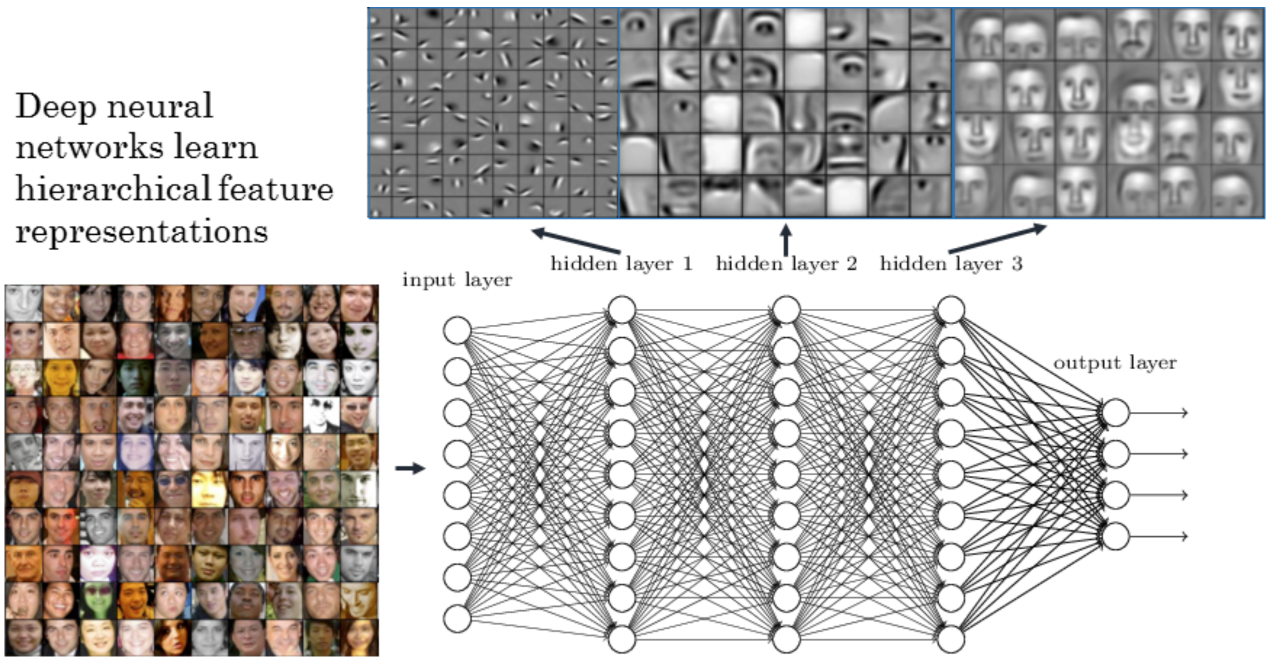
Thus the hidden units act as a form of basis space for the input vectors.

3.3. Examples of the Hidden Units given by Autoencoders

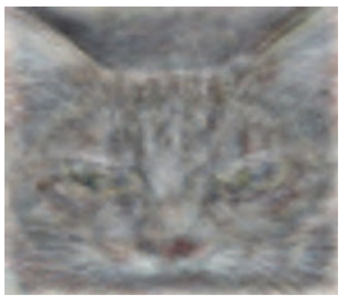
An example of 100 hidden units learned by a sparse auto-encoder from images:



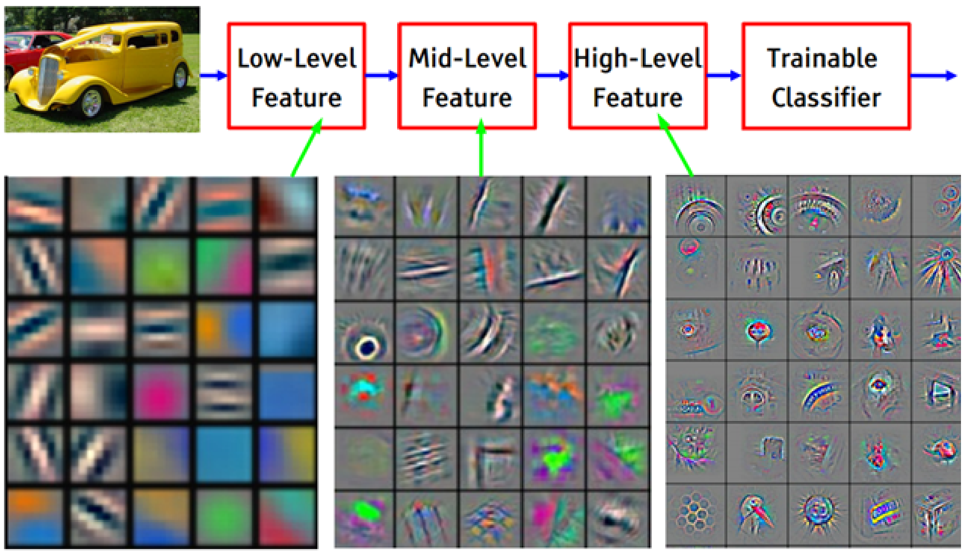
When applied at multiple levels and trained on face images this can give recognizable features:



or when trained on YouTube videos: Cats



when trained with car images:


Low-Level Feature
Mid-Level Feature
High-Level Feature
Trainable Classifier