Machine Learning: Lecture 4

Artificial Neural Networks

(Based on Chapter 4 of Mitchell T..,

Machine Learning, 1997)

Also see:

http://130.243.105.49/~lilien/ml/seminars/2007_02_01b-Janecek-Perceptron.pdf

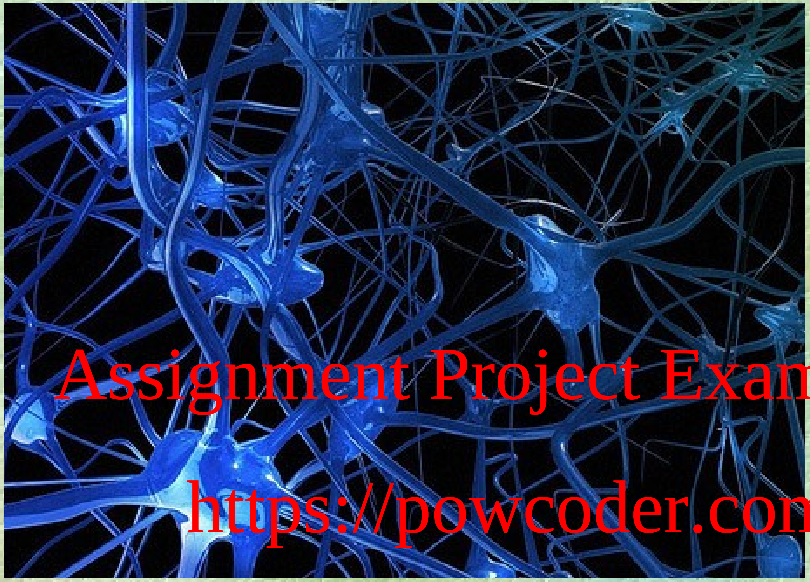
https://cs.stanford.edu/~quocle/tutorial1.pdf

https://cs.stanford.edu/~quocle/tutorial2.pdf

What is an Artificial Neural Network?

- It is a formalism for representing functions inspired from biological systems and composed of parallel computing units which each compute a simple function.
- Some useful computations taking place in ***Feedforward Multilayer*** Neural Networks are:
  - Summation
  - Multiplication
  - Threshold (e.g., $1/(1+e^{-x})$) [the sigmoidal threshold function]. Other functions are also possible

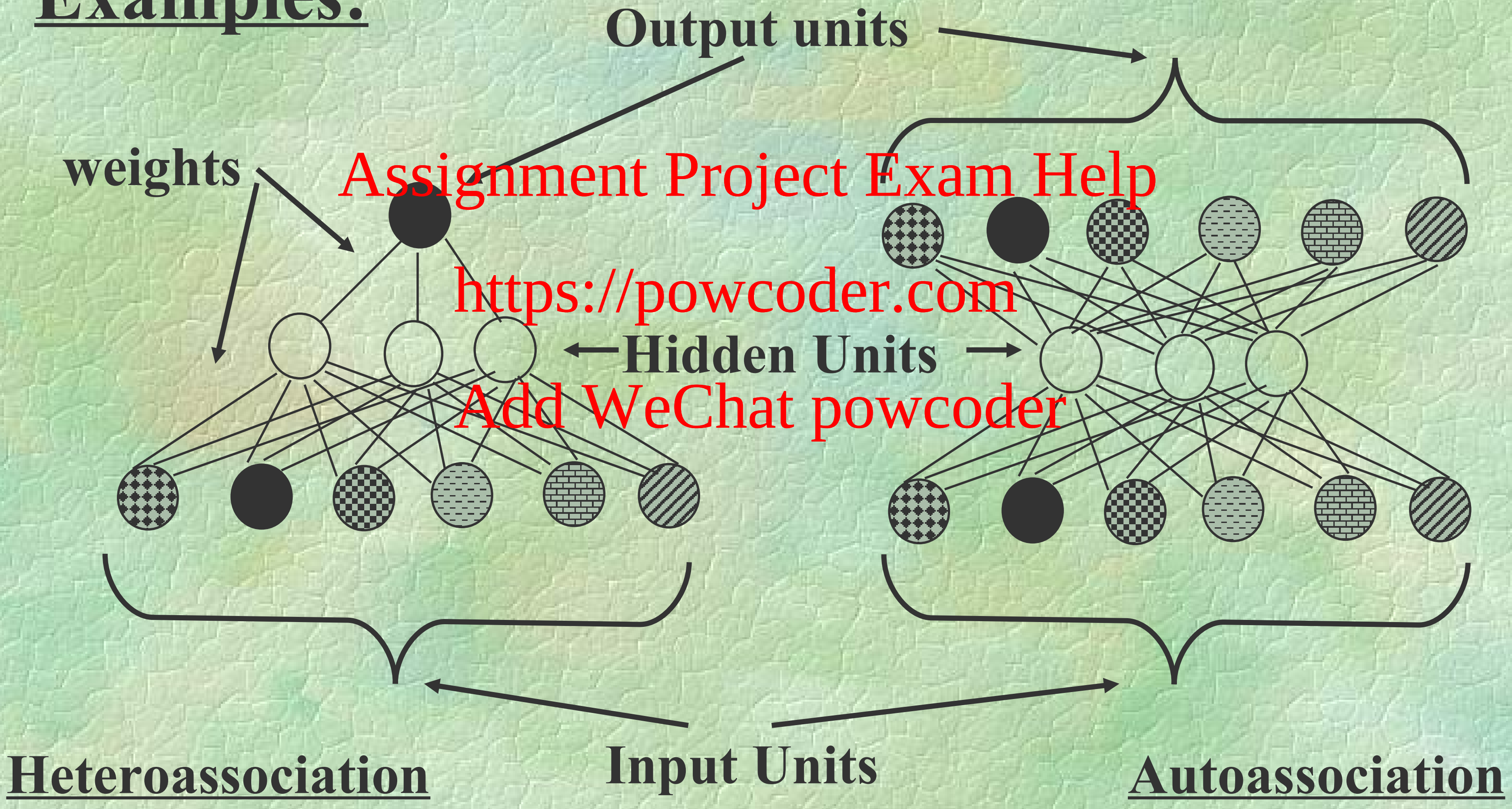
Biological Motivation



- Biological Learning Systems are built of very complex webs of interconnected neurons.
- Information-Processing abilities of biological neural systems must follow from highly parallel processes operating on representations that are distributed over many neurons
- ANNs attempt to capture this mode of

Multilayer Neural Network Representation

Examples:



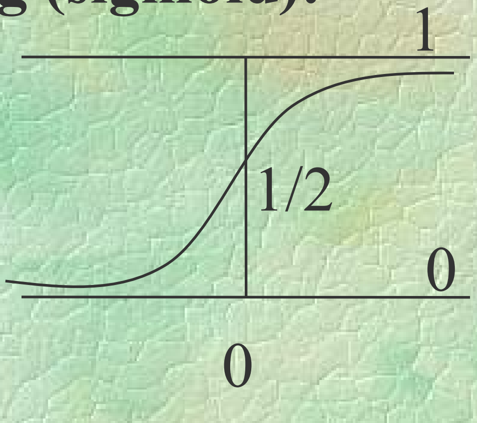
How is a function computed by a Multilayer Neural Network?

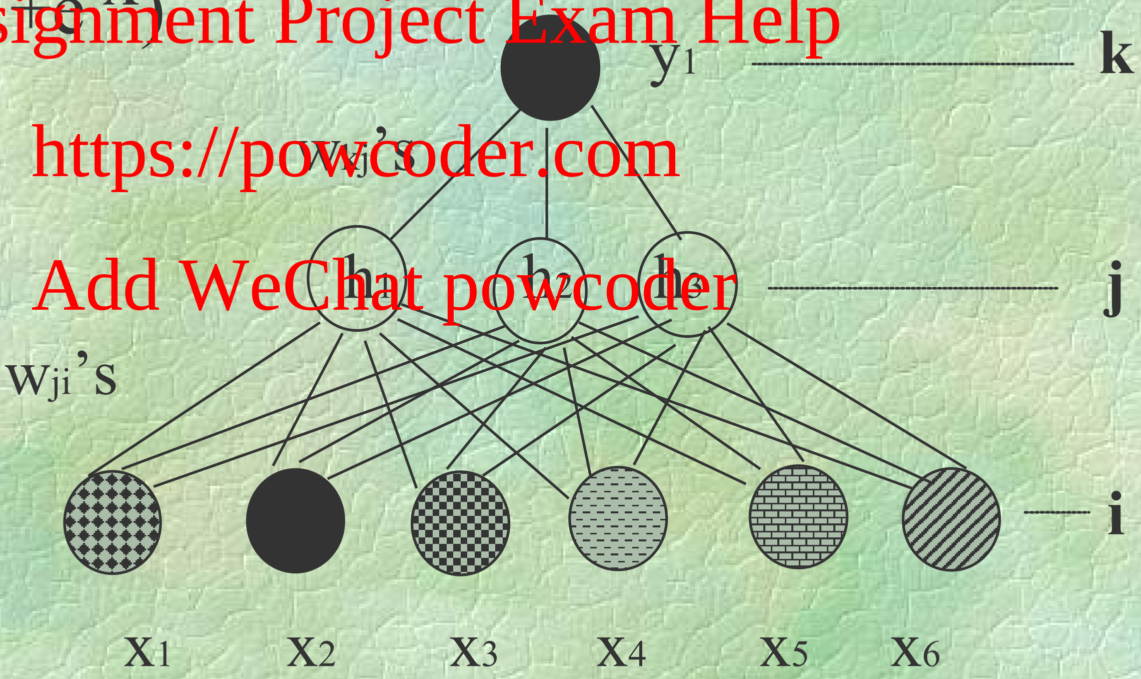
- $h_j = g(\sum_i w_{ji} . x_i)$
- $y_1 = g(\sum_j w_{kj} . h_j)$

where $g(x) = 1/(1+e^{-x})$

Typically, $y_1=1$ for positive example and $y_1=0$ for negative example

g (sigmoid):





Learning in Multilayer Neural Networks

- Learning consists of searching through the space of all possible matrices of weight values for a combination of weights that satisfies a database of positive and negative examples (multi-class as well as regression problems are possible).
- Note that a Neural Network model with a set of adjustable weights defines a restricted hypothesis space corresponding to a family of functions. The size of this hypothesis space can be increased or decreased by increasing or decreasing the number of hidden units present in the network.

Appropriate Problems for Neural Network Learning

- Instances are represented by many attribute-value pairs (e.g., the pixels of a picture. ALVINN [first self-driving car]).
- The target function output may be discrete-valued, real-valued, or a vector of several real- or discrete-valued attributes.
- The training examples may contain errors.
- Long training times are acceptable.
- Fast evaluation of the learned target function may be required.
- The ability for humans to understand the learned target function is not important.

History of Neural Networks *I*

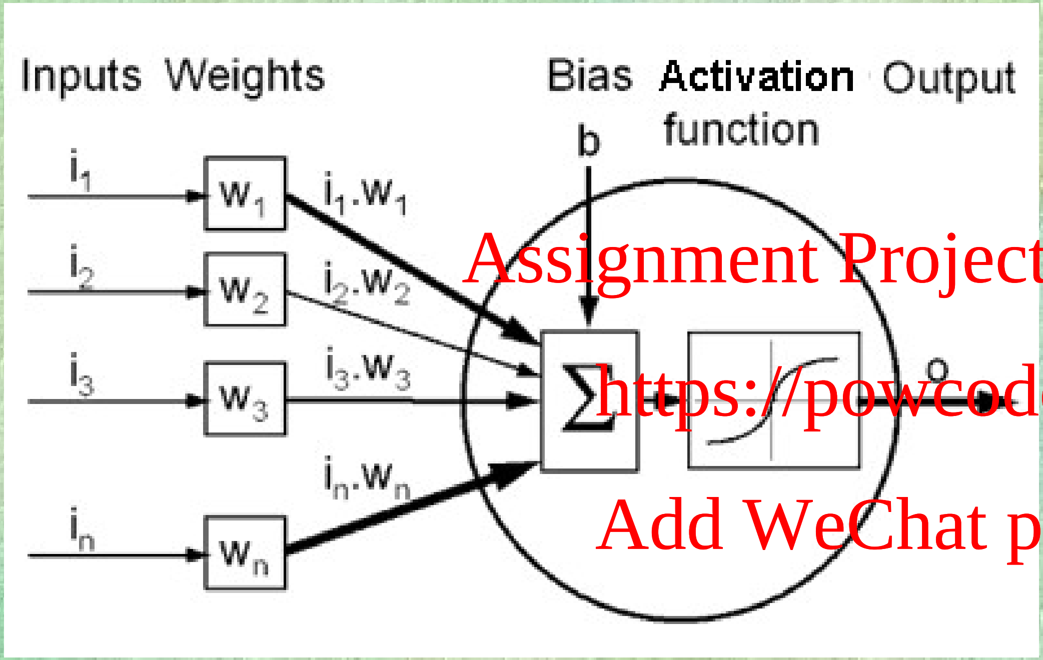
- 1943: McCulloch and Pitts proposed a model of a neuron --> Perceptron
- 1960s: Widrow and Hoff explored Perceptron networks (which they called "Adelines") and the delta rule.
- 1962: Rosenblatt proved the convergence of the perceptron training rule.
- 1969: Minsky and Papert showed that the Perceptron cannot deal with nonlinearly-separable data sets---even those that represent simple function such as X-OR.
- 1970-1985: Very little research on Neural Nets
- 1986: Invention of Backpropagation [Rumelhart and McClelland, but also Parker and earlier on: Werbos] which can learn from nonlinearly-separable data sets.
- Between 1985 and 1995: A lot of research in Neural Nets!

History of Neural Networks *II*

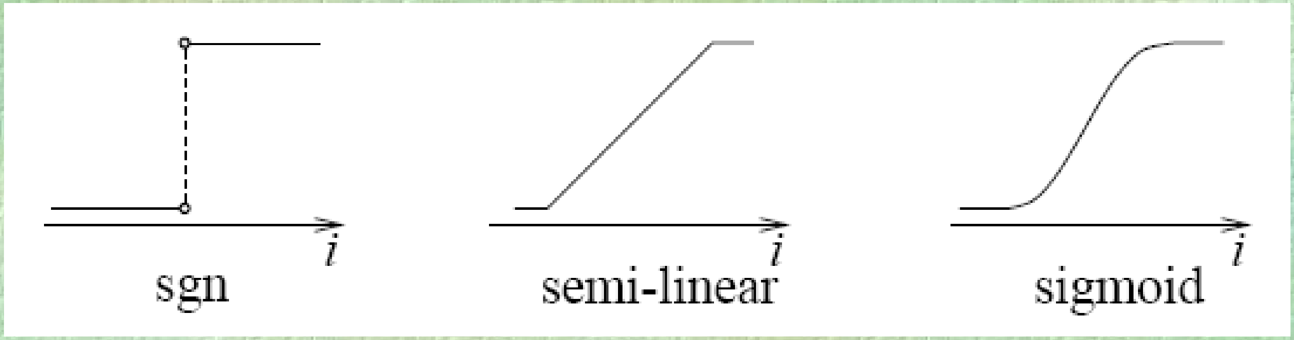
- **1995-2005: Support Vector Machines gain in popularity at the expense of Neural Nets. Neural Nets are still used in applications, but there is less theoretical research in the field.**
- **2005-today: With the advent of Deep learning, Neural Nets have regained popularity. This is thanks to powerful inexpensive parallel hardware and massive amounts of labeled data.**
- **To be continued…**

Simple Perceptrons I:

Processing Unit:

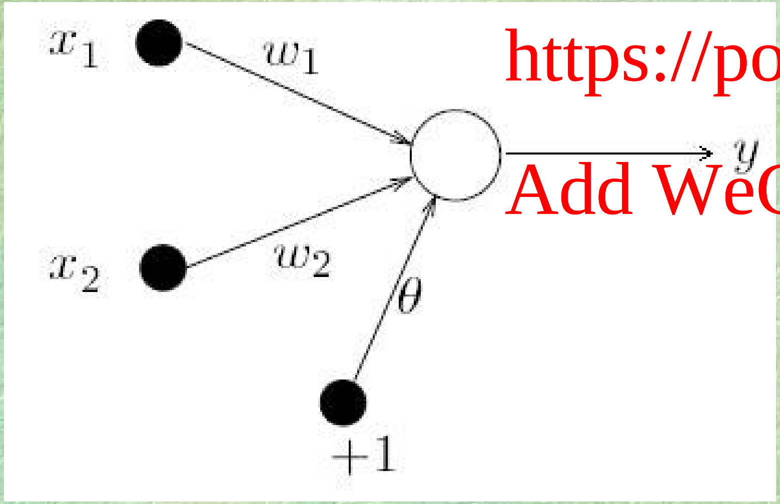


Activation Function:
Function which takes the total input and produces an output for the node given some threshold.



Simple Perceptrons II:

A Percepton is a single-layered Feedforward Neural Network.



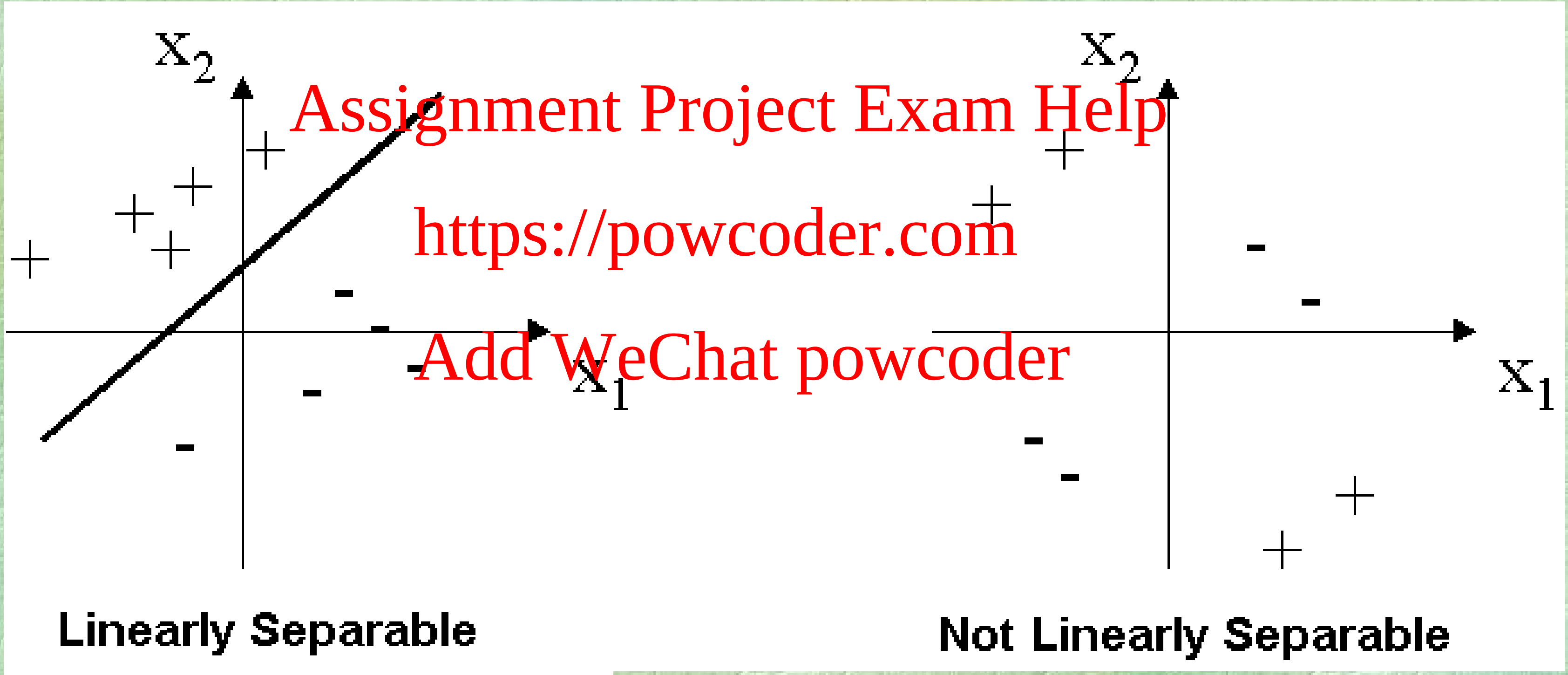
Its output function

$$y = \operatorname{sgn}\left(\sum_{i=1}^{2} w_i x_i + \theta\right)$$

$$\operatorname{sgn}(s) = \begin{cases} 1 & \text{if } s > 0 \\ -1 & \text{otherwise.} \end{cases}$$

can classify linearly separable patterns

Simple Perceptrons III:



$$w_1 x_1 + w_2 x_2 + \theta = 0$$

Perceptron Learning Algorithm

- We have a "training set" which is a set of input vectors used to train the perceptron.
- During training both w_i and θ *(bias)* are modified for convenience, let $w_0 = \theta$ and $x_0 = 1$
- *Let,* η, the *learning rate*, be a small positive number (small steps lessen the possibility of destroying correct classifications)
- Initialise w_i to some values

The Perceptron Training Rule

- $w_i \leftarrow w_i + \Delta w_i$
- $\Delta w_i \leftarrow \eta\ (t-y)\ x_i$ (t_i is the target or expected output of xi)

- Why does the training rule work?

If $t= y$, $\Delta w_i= 0$ → No change (Good!)

If $t= +1$ and $y= -1$, $\Delta w_i =2$ →will increase (Good!)

If $t= -1$ and $y= +1$, $\Delta w_i =-2$ →will decrease (Good!)

Perceptron Convergence and Delta rule

- **Theorem:** The Perceptron Training rule converges within a finite number of iterations provided that the training data is *linearly separable* and that a sufficient *small* η is used,
- Convergence is *not assured* if the data is *not linearly separable*.
- ➔ Gradient Descent and the Delta Rule: the delta rule converges toward a best-fit approximation to the target concept when the training examples are not linearly separable ➔ Basis for Backpropagation

Backpropagation: Purpose and Implementation

- **Purpose:** To compute the weights of a feedforward **multilayer** neural network adaptatively, given a set of labeled training examples.

- Method: By minimizing the following cost function (the sum of square error):

$$E = 1/2 \sum_{n=1}^{N} \sum_{k=1}^{K} [y_k^n - f_k(x^n)]^2$$

where N is the total number of training examples and K, the total number of output units (useful for multiclass problems) and f_k is the function implemented by the neural net

Backpropagation: Overview

- Backpropagation works by applying the ***gradient descent*** rule to a feedforward network.
- The algorithm is composed of two parts that get repeated over and over until a pre-set maximal number of ***epochs***, ***EPmax***.
- Part I, the ***feedforward*** pass: the activation values of the hidden and then output units are computed.
- Part II, the ***backpropagation*** pass: the weights of the network are updated--starting with the hidden to output weights and followed by the input to hidden weights--with respect to the sum of squares error and through a series of weight update rules called the ***Delta Rule***.

Backpropagation: The Delta Rule I

- **For the hidden to output connections** (easy case)
- $\Delta w_{kj} = -\eta\, \partial E / \partial w_{kj}$

$$= \eta \sum_{n=1}^{N} [y_k^n - f_k(x^n)]\, g'(h_k^n)\, V_j^n$$

$$= \eta \sum_{n=1}^{N} \delta_k^n V_j^n$$

with

- η corresponding to the ***learning rate*** (an extra parameter of the neural net)
- $h_k^n = \sum_{j=0}^{M} w_{kj} V_j^n$
- $V_j^n = g(\sum_{i=0}^{d} w_{ji} x_i^n)$ and $\forall$ $\delta_k^n = g'(h_k^n)(y_k^n - f_k(x^n))$

M is the number of hidden units and d the number of input units

Backpropagation: The Delta Rule II

- **For the input to hidden connections** (hard case: no pre-fixed values for the hidden units)
- $\Delta w_{ji} = -\eta\, \partial E / \partial w_{ji}$

  $= -\eta\, \Sigma_{n=1}^{N}\, \partial E / \partial V_j^n\; \partial V_j^n / \partial w_{ji}$ (Chain Rule)

  $= \eta\, \Sigma_{k,n} [y_k^n - f_k(x^n)]\, g'(h_k^n)\, w_{kj}\, g'(h_j^n)\, x_i^n$

  $= \eta\, \delta_k^n w_{kj} g'(h_j^n) x_i^n$

  $= \eta\, \Sigma_{n=1}^{N} \delta_j^n\, x_i^n$ with

  - $h_j^n = \Sigma_{i=0}^{d}\, w_{ji} x_i^n$
  - $\delta_j^n = g'(h_j^n)\, \Sigma_{k=1}^{K}\, w_{kj}\, \delta_k^n$
  - and all the other quantities already defined

Backpropagation: The Algorithm

1. Initialize the weights to small random values; create a random pool of all the training patterns; set ***EP***, the number of epochs of training to 0.
2. Pick a training pattern μ from the remaining pool of patterns and propagate it forward through the network.
3. Compute the deltas, δ_k^{μ} for the output layer.
4. Compute the deltas, δ_j^{μ} for the hidden layer by propagating the error backward.
5. Update all the connections such that

   $\mathbf{w}_{ji}^{New} = \mathbf{w}_{ji}^{Old} + \Delta \mathbf{w}_{ji}$ and $\mathbf{w}_{kj}^{New} = \mathbf{w}_{kj}^{Old} + \Delta \mathbf{w}_{kj}$
6. If any pattern remains in the pool, then go back to Step 2. If all the training patterns in the pool have been used, then set $\mathbf{EP = EP+1}$, and if $\mathbf{EP} \langle \mathbf{EP}_{Max}$, then create a random pool of patterns and go to Step 2. If $\mathbf{EP = EP}_{Max}$, then stop.

Backpropagation: The Momentum

- To this point, Backpropagation has the disadvantage of being too slow if η is small and it can oscillate too widely if η is large.
- To solve this problem, we can add a ***momentum*** to give each connection some inertia, forcing it to change in the direction of the downhill "force".

- <u>**New Delta Rule:**</u>

$$\Delta w_{pq}(t+1) = -\eta \, \partial E/\partial w_{pq} + \alpha \, \Delta w_{pq}(t)$$

where p and q are any input and hidden, or, hidden and outpu units; t is a time step or epoch; and α is the momentum parameter which regulates the amount of inertia of the weights.

Other kinds of neural networks

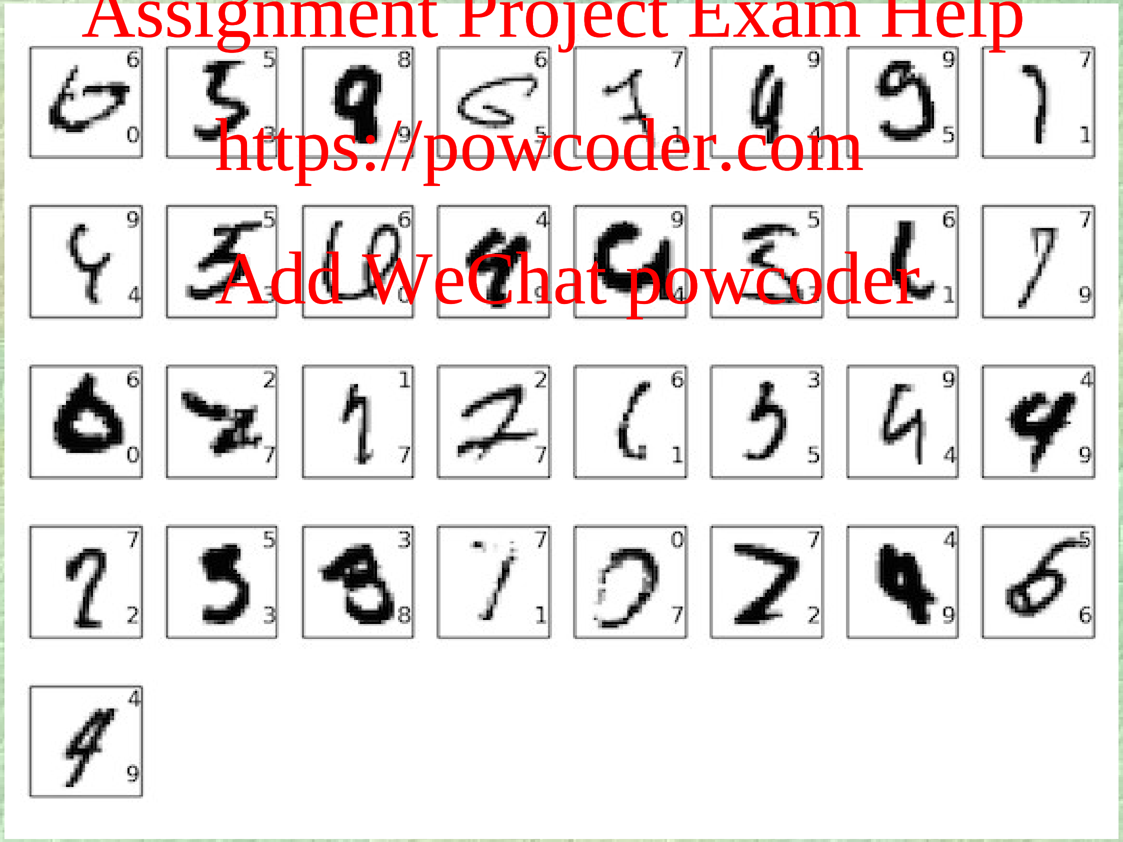
- Hopfield Networks
- Learning Vector Quantization (LVQ)
- Radial Basis Function (RBF) Networks
- Self Organizing Maps (SOM)
- Recurent Networks
- Boltzmann Machine
- Long Short Term Memory (LSTM) Networks
- Convolutional Networks

Convolutional Networks

- Deep Convolutional Networks have been very successful on image classification problems.
- We first describe a shallow convolutional network and then expand it to a deep architecture,
- Convolutional networks are based on three basic ideas:
  - *local receptive fields*
  - *shared weights*
  - *pooling*

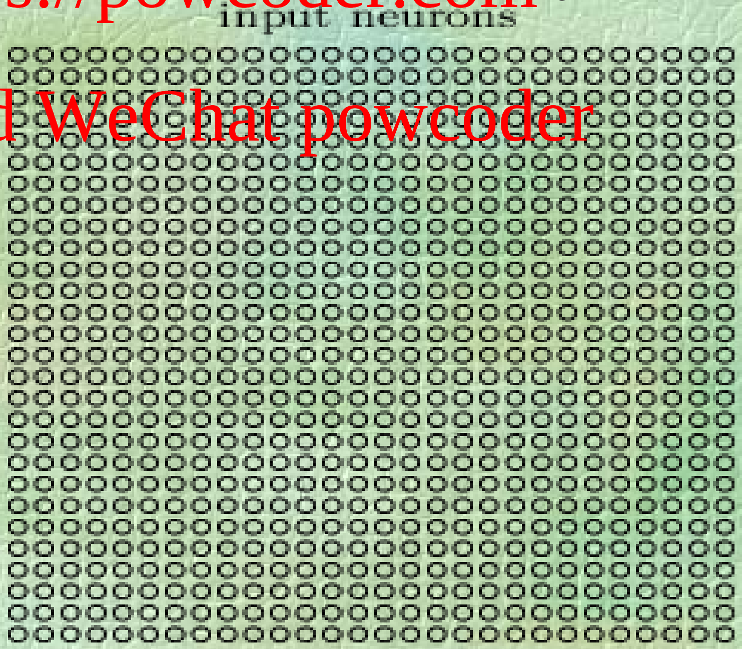
Example of a problem convolutional networks can solve

- Classify the following images in to one of the 10 digit classes (0, 1, 2, …, 9)



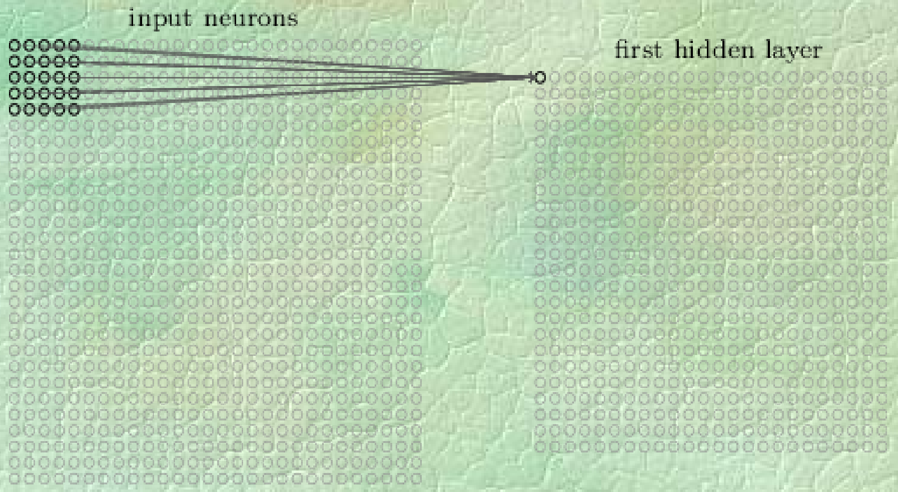
Local Receptive fields I

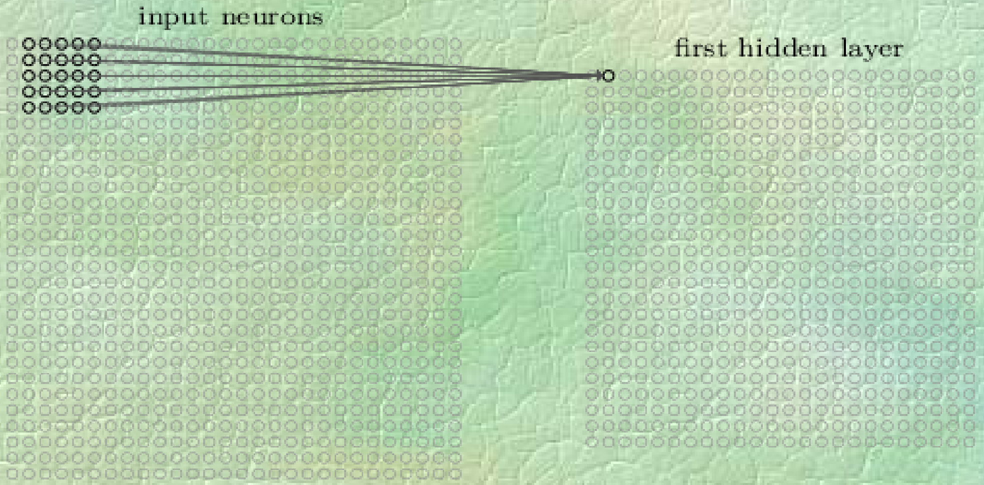
- Instead of being represented as unidimensional, we will think of input layer as two-dimensional.
- Each neuron represents the intensity of one of the image's pixels.



Local Receptive fields II

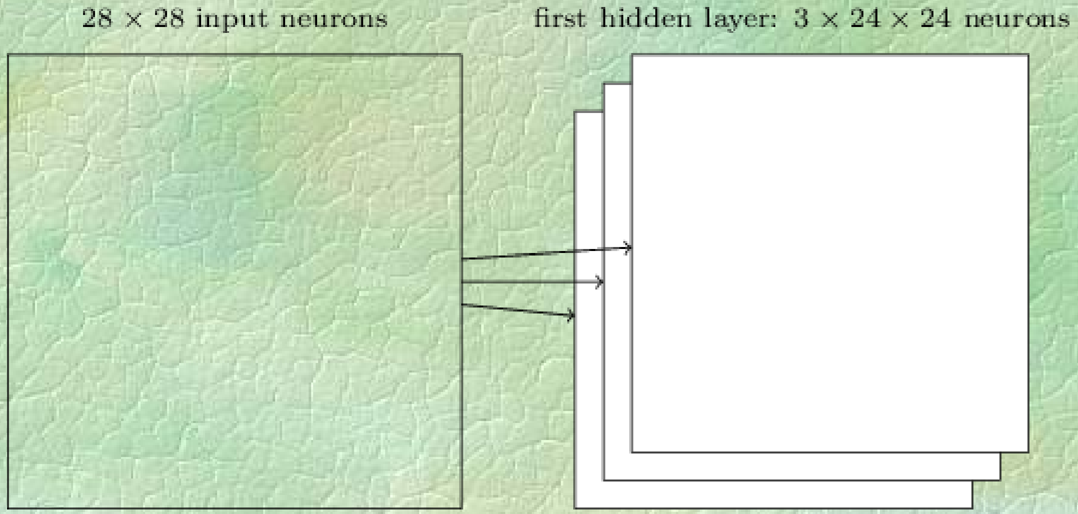
- The input pixels are connected to a layer of hidden neurons but not every input pixel is connected to every hidden neuron. Instead, we only make connections in small, localized regions of the input image. E.g., all the neurons in a 5 x 5 pixel region are connected to one hidden unit unit; the next (overlapping) 5x5 pixel region is connected to a second hidden unit and so on.





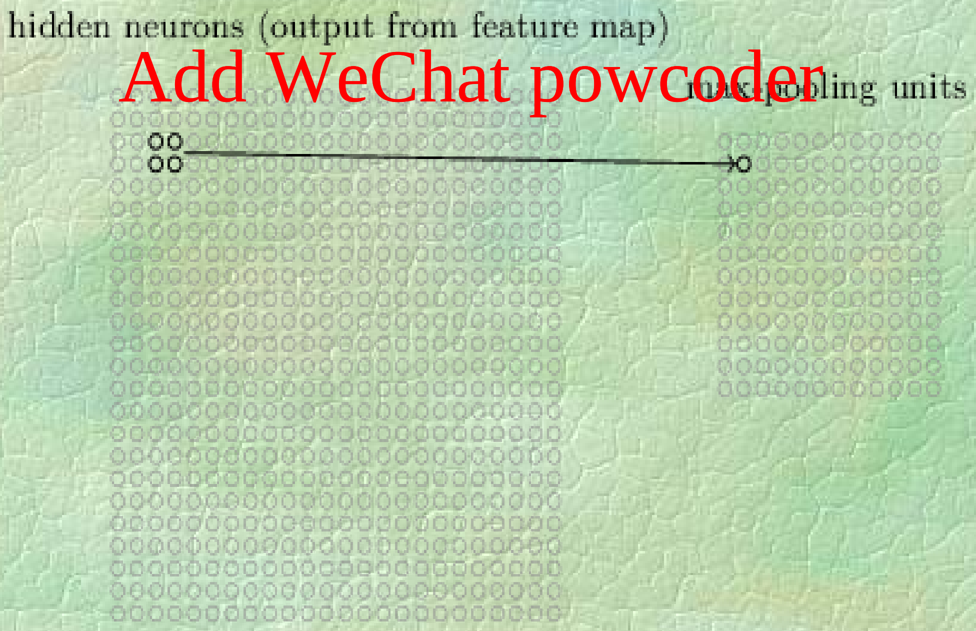
Shared Weights and Biases

- The weights and bias going from the input layer to one hidden unit are the same for each local receptive field and hidden unit.
- That means that all the neurons in the first hidden layer detect exactly the same feature (e.g., an edge) in the input image, but they each do so at different locations. The map from the input layer to the hidden layer is sometimes called a *feature map.*
- If we want to detect different kinds of features, we need different more than one feature map. A complete hidden convolutional layer consists of several different feature maps:



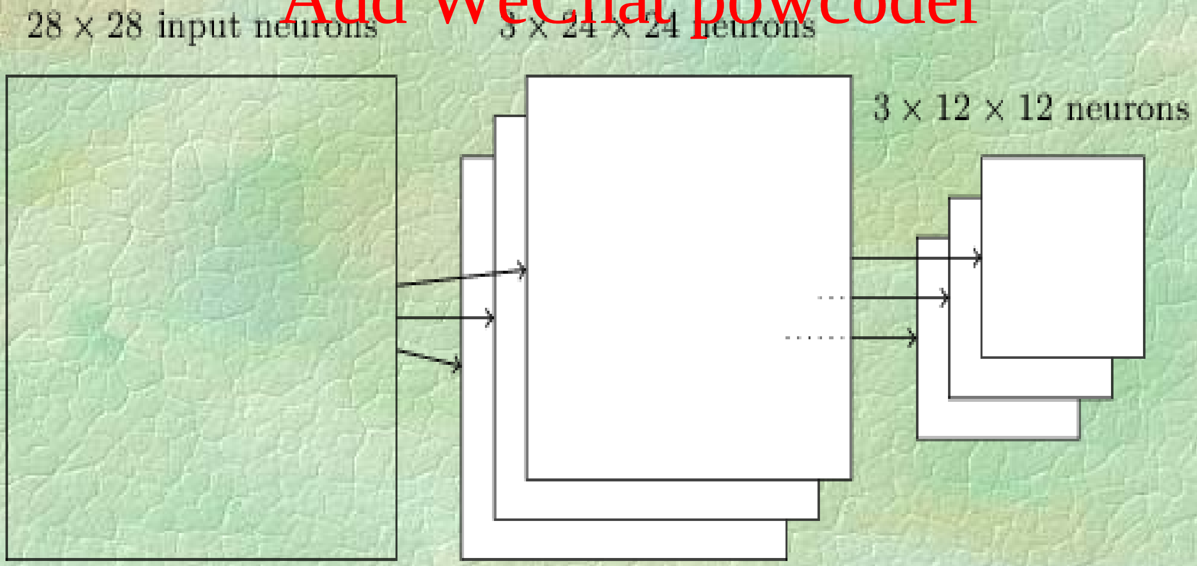
Pooling Layers I

- Pooling layers are usually used immediately after convolutional layers.
- What the pooling layers do is simplify the information in the output from the convolutional layer.
- A common strategy is max-pooling, where a pooling unit simply outputs the maximum activation in the 2x2 input region



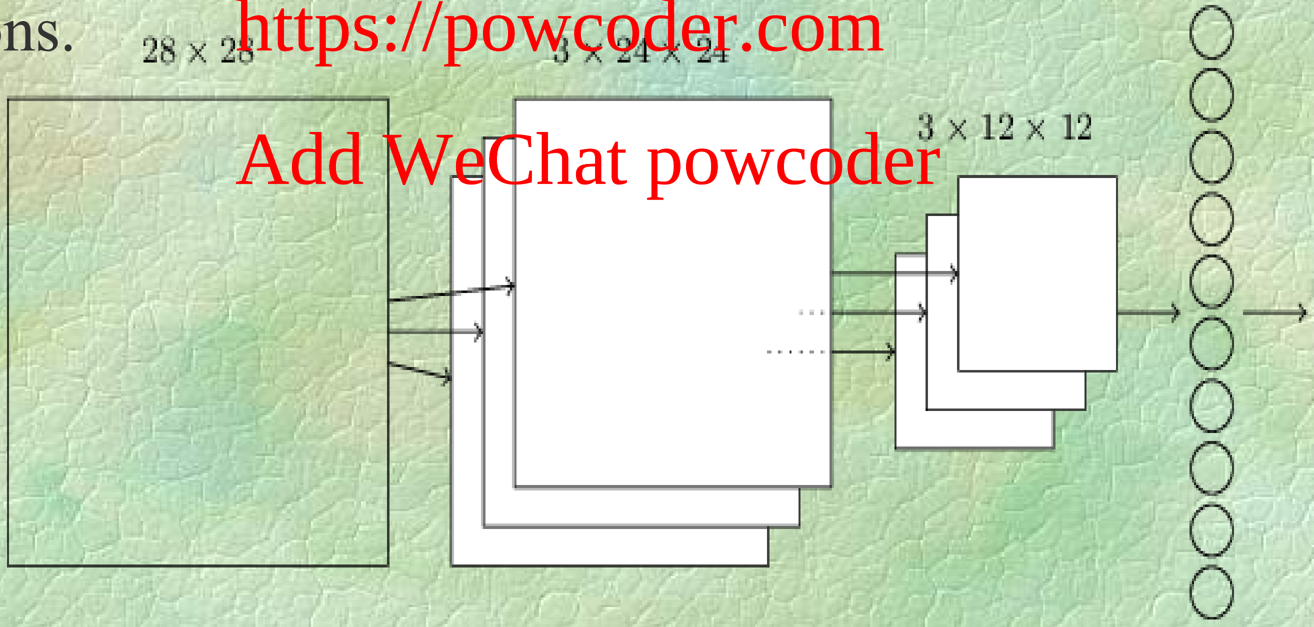
Pooling Layers II

- We can think of max-pooling as a way for the network to ask whether a given feature is found anywhere in a region of the image. It then throws away the exact positional information. The intuition is that once a feature has been found, its exact location isn't as important as its rough location relative to other features. A big benefit is that there are many fewer pooled features, and so this helps reduce the number of parameters needed in later layers.



Putting it all together

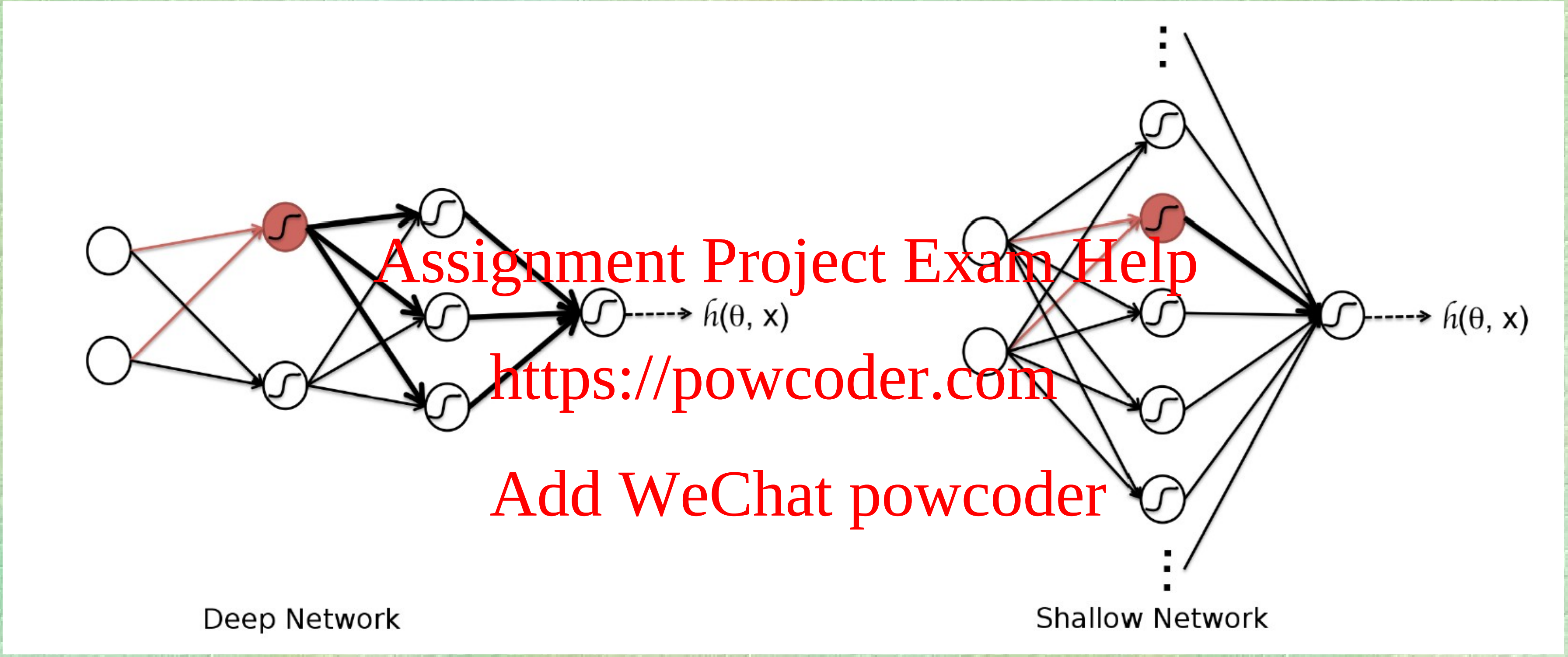
- The final layer of connections in the network is a fully-connected layer. That is, this layer connects *every* neuron from the max-pooled layer to every one of the output neurons.



Deep Learning I

- The Neural Networks we have previously seen can be thought of as shallow Neural Networks which only contain a single hidden layer.
- The hidden layer of these networks computes a new representation of the data that makes the learning from the hidden layer to the output layer easier.
- Although the hidden layer representation is not well understood, experimental results lead researchers to believe that it is quite powerful.
- For many years, researchers have also believed that using more than one hidden layers could lead to even more effective representations.

Deep Learning II



It has been shown experimentally that in order to get the same level of performance as in a deep network, one has to use a shallow network with many more hidden units. That is much more computationally costly.

Deep Learning III

- The idea of deep learning is to use many hidden layers (more than 2) to construct more and more abstract sets of features automatically. It is believed that, at least on certain types of applications such as vision, speech recognition and natural language processing problems, deep learning allows a network to "discover" better representations than those generated by human beings.
- However, until recently, processing neural networks with large numbers of layers was not done effectively. That is because the gradient on which training methods are based has a tendency to vanish or explode when considered throughout many hidden layers.

Deep Learning IV

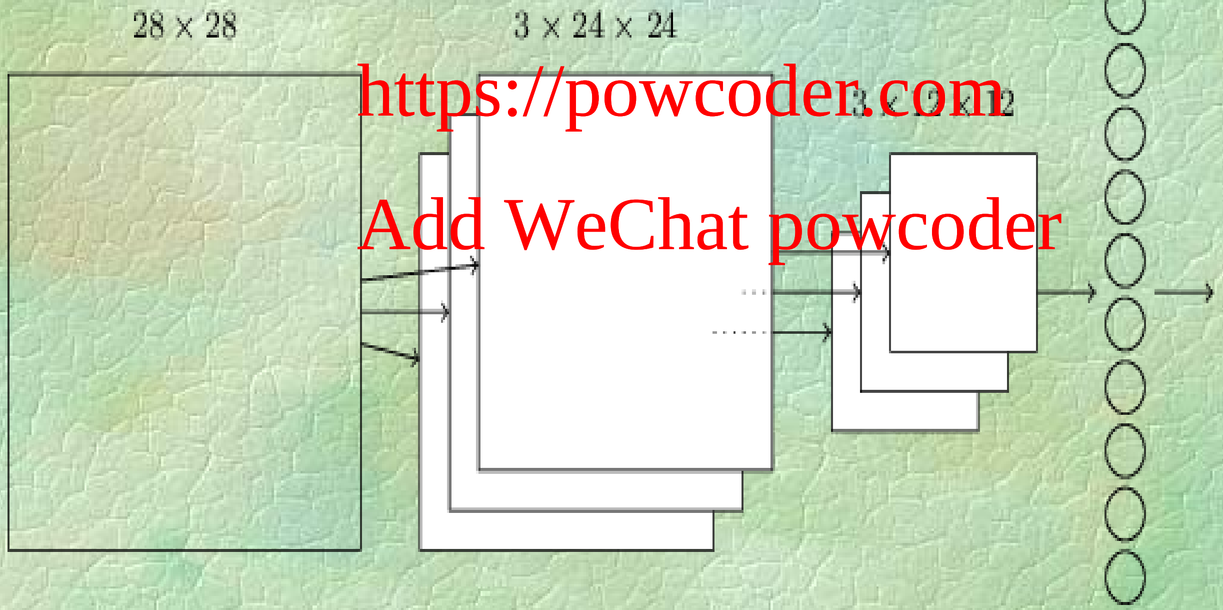
- This issue has been tackled using different previously known tricks together and using more powerful machines.
- These tricks include
  - using more powerful regularization techniques and convolutional layers, to reduce overfitting;
  - using rectified linear units instead of sigmoid neurons to speed up training;
  - using GPUs and being willing to train for a long period of time;
  - using a large number of training examples, also to avoid overfitting.

Why does Deep Learning work so well?

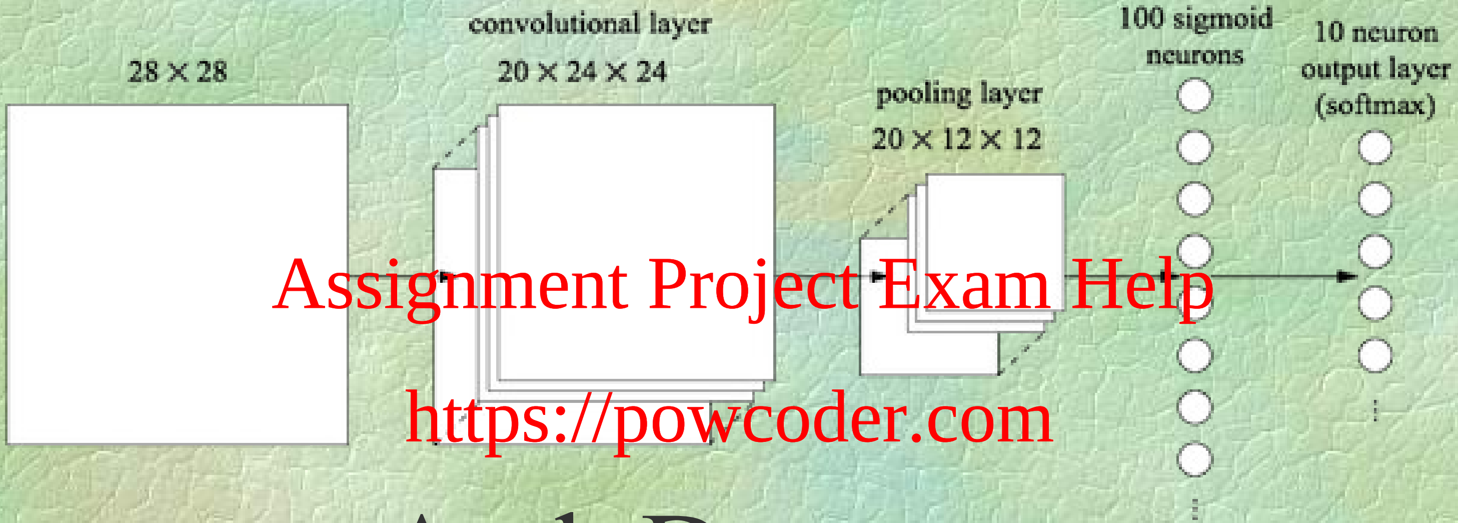
- The reason is not well understood yet, but the intuition is that the multiple layers allow the input to be decomposed hierarchically into more and more abstract and meaningful representations.
- This decomposition also helps implement complex functions more concisely and possibly more accurately.
- In addition, the learned representations are more appropriate for neural network learning than those created by humans. For example, the number of legs an animal has may be an important abstract representational concept for a human being to reason about the animal, but it may not be the most relevant one for a neural, distributed representation of that animal. The representation learned by a deep network are abstract the way the number of legs is abstract, but is not meaningful to us the way it is to the network.

An Example: Deep Convolutional Networks

Reminder: Shallow Convolutional Network:



Going Deeper…



And Deeper…

Add another convolutional + pooling layer after the first one and before the sigmoid layer. That layer will learn abstractions of the abstraction!

Does Deep learning always work well?

- No:
  - Deep learning requires a huge amount of training data
  - Deep learning training takes time since it needs to process a lot of data and update many connections. It also requires powerful computers.
  - Deep learning requires the tuning of many parameters and the use of many computational tricks. These tricks take time to figure out.
  - Not every domain is appropriate for Deep learning.
- However, well done and run on appropriate domains, Deep learning obtains impressive results.