

Convolutional Neural Networks

CSE 6367: Computer Vision

Instructor: William J. Beks

Introduction

- A **neural network** uses a multitude of elemental nonlinear computing elements (called **neurons**) organized as networks similar to the way in which neurons are believed to be interconnected in the brain
- Interest in neural networks dates back to the early 1940s when neuron models in the form of binary threshold devices were proposed (McCulloch and Pitts, 1943)

Perceptron for Two Pattern Classes

- In its most basic form, the **perceptron** learns a linear decision function that dichotomizes two linearly separable training sets
- The response of the perceptron is based on a weighted sum of its inputs

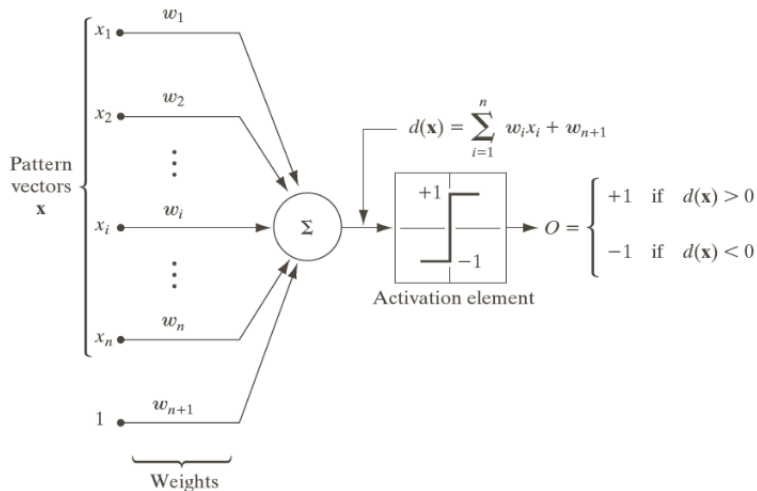
$$d(\mathbf{x}) = \sum_{i=1}^n w_i x_i + w_{n+1} \quad (1)$$

which is a linear decision function w.r.t the components of the pattern vectors

Perceptron for Two Pattern Classes

- The coefficients $w_i, i = 1, 2, \dots, n, n + 1$, called **weights**, modify the inputs before they are summed and fed into the threshold element
- In this sense, the weights are analogous to synapses in the human neural system
- The function that maps the output of the summing junction into the final output of the perceptron is called the **activation function**

Perceptron for Two Pattern Classes



Perceptron for Two Pattern Classes

- When $d(\mathbf{x}) > 0$, the threshold element causes the output of the perceptron to be +1, indicating that the pattern \mathbf{x} was recognized as belonging to class ω_1
- The reverse is true when $d(\mathbf{x}) < 0$
- When $d(\mathbf{x}) = 0$, \mathbf{x} lies on the decision surface separating the two pattern classes, giving an indeterminate condition

Perceptron for Two Pattern Classes

- The decision boundary implemented by the perceptron is obtained by setting equation (1) to zero

$$d(\mathbf{x}) = \sum_{i=1}^n w_i x_i + w_{n+1} = 0$$

or

$$w_1 x_1 + w_2 x_2 + \cdots + w_n x_n + w_{n+1} = 0$$

which is the equation of a hyperplane in n-dimensional pattern space

Perceptron for Two Pattern Classes

- Geometrically, the first n coefficients establish the orientation of the hyperplane
- The last coefficient, w_{n+1} , is proportional to the perpendicular distance from the origin to the hyperplane
- Thus, if $w_{n+1} = 0$, the hyperplane goes through the origin of the pattern space, if $w_j = 0$ the hyperplane is parallel to the x_j -axis

Perceptron for Two Pattern Classes

- A frequently used formulation is to augment the pattern vectors by appending an additional $(n + 1)$ st element, which is always equal to 1, regardless of class membership
- Specifically, an augmented pattern vector \mathbf{y} is created from a pattern vector \mathbf{x} by letting $y_i = x_i, i = 1, 2, \dots, n$, and appending the additional element $y_{n+1} = 1$

Perceptron for Two Pattern Classes

- By appending the additional element to the pattern vector, Equation (1) becomes

$$\begin{aligned}d(\mathbf{y}) &= \sum_{i=1}^{n+1} w_i y_i \\ &= \mathbf{w}^T \mathbf{y}\end{aligned}$$

where $\mathbf{y} = [y_1, y_2, \dots, y_n, 1]^T$ is now an **augmented pattern vector** and $\mathbf{w} = [w_1, w_2, \dots, w_n, w_{n+1}]^T$ is called the **weight vector**

- The key problem is to find \mathbf{w} by using a given training set of pattern vectors from each of the two classes

Linearly Separable Classes

- For two training sets of augmented pattern vectors belonging to pattern classes ω_1 and ω_2 , respectively, let $\mathbf{w}(1)$ represent the initial weight vector
- Then, at the k th iterative step, if $\mathbf{y}(k) \in \omega_1$ and $\mathbf{w}^T(k)\mathbf{y}(k) \leq 0$, replace $\mathbf{w}(k)$ by

$$\mathbf{w}(k+1) = \mathbf{w}(k) + c\mathbf{y}(k)$$

where c is a positive correction increment

Linearly Separable Classes

- Conversely, if $\mathbf{y}(k) \in \omega_2$ and $\mathbf{w}^T(k)\mathbf{y}(k) \geq 0$, replace $\mathbf{w}(k)$ by

$$\mathbf{w}(k+1) = \mathbf{w}(k) - c\mathbf{y}(k)$$

where c is a positive correction increment

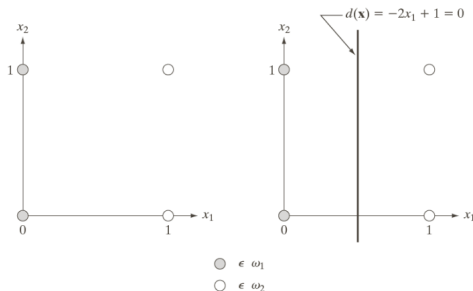
- Otherwise, $\mathbf{w}(k)$ is unchanged

$$\mathbf{w}(k+1) = \mathbf{w}(k)$$

Linearly Separable Classes

- This algorithm, referred to as the **fixed increment correction rule**, makes a change to \mathbf{w} only if the pattern being considered at the k th step in the training process is misclassified
- Convergence of the algorithm occurs when the entire training set for both classes is cycled through without any errors
- The algorithm converges in a finite number of steps if the training sets of patterns are linearly separable (**perceptron training theorem**)

Example: The Perceptron Algorithm



- Left: patterns belonging to two classes; Right: decision boundary determined by training

Example: The Perceptron Algorithm

- Consider two augmented training sets $\{[0, 0, 1]^T, [0, 1, 1]^T\}$ for class ω_1 and $\{[1, 0, 1]^T, [1, 1, 1]^T\}$ for class ω_2
- Letting $c = 1$, $\mathbf{w}(1) = \mathbf{0}$, and presenting the patterns in order results in the following sequence of steps

$$\mathbf{w}^T(1)\mathbf{y}(1) = [0 \ 0 \ 0] \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix} = 0, \quad \mathbf{w}(2) = \mathbf{w}(1) + \mathbf{y}(1) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

$$\mathbf{w}^T(2)\mathbf{y}(2) = [0 \ 0 \ 1] \begin{bmatrix} 0 \\ 1 \\ 1 \end{bmatrix} = 1, \quad \mathbf{w}(3) = \mathbf{w}(2) = \begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

Example: The Perceptron Algorithm

$$\mathbf{w}^T(3)\mathbf{y}(3) = [0 \ 0 \ 1] \begin{bmatrix} 1 \\ 0 \\ 1 \end{bmatrix} = 1, \quad \mathbf{w}(4) = \mathbf{w}(3) - \mathbf{y}(3) = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{w}^T(4)\mathbf{y}(4) = [-1 \ 0 \ 0] \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} = -1, \quad \mathbf{w}(5) = \mathbf{w}(4) = \begin{bmatrix} -1 \\ 0 \\ 0 \end{bmatrix}$$

where corrections in the weight vector were made in the first and third steps because of misclassifications

Example: The Perceptron Algorithm

- Since a solution is obtained only when the algorithm yields a complete error-free iteration through all the training patterns, the training set must be presented again
- The process continues by letting $\mathbf{y}(5) = 1, \mathbf{y}(6) = \mathbf{y}(2), \mathbf{y}(7) = \mathbf{y}(3)$, and $\mathbf{y}(8) = \mathbf{y}(4)$, and proceeding in the same manner
- Convergence is achieved at $k = 14$, yielding the solution weight vector $\mathbf{w}(14) = [-2, 0, 1]^T$, and the corresponding equation for the decision boundary is $d(\mathbf{x}) = -2x_1 + 1$

Nonseparable Classes

- In practice, linearly separable classes are the (rare) exception rather than the rule
- One of the methods for dealing with nonseparable classes is known as the **Widrow-Hoff**, or **least mean square delta rule** (LMS) for training perceptrons
- This method minimizes the error between the actual and desired response at any training step

Delta Correction Algorithm

- Consider the criterion function

$$J(\mathbf{w}) = \frac{1}{2}(r - \mathbf{w}^T \mathbf{y})^2 \quad (2)$$

where r is the desired response (i.e. $r = +1$ if $\mathbf{y} \in \omega_1$ and $r = -1$ if $\mathbf{y} \in \omega_2$)

- The task is to adjust \mathbf{w} incrementally in the direction of the negative gradient of $J(\mathbf{w})$ in order to seek the minimum which occurs when $r = \mathbf{w}^T \mathbf{y}$ (i.e. the minimum corresponds to correct classification)

Delta Correction Algorithm

- If $\mathbf{w}(k)$ represents the weight vector at the k th iterative step, a general gradient descent algorithm may be written as

$$\mathbf{w}(k+1) = \mathbf{w}(k) - \alpha \left[\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} \right]_{\mathbf{w}=\mathbf{w}(k)} \quad (3)$$

where $\mathbf{w}(k+1)$ is the new value of \mathbf{w} and $\alpha > 0$ gives the magnitude of the correction

Delta Correction Algorithm

- From equation (2)

$$\frac{\partial J(\mathbf{w})}{\partial \mathbf{w}} = -(r - \mathbf{w}^T \mathbf{y}) \mathbf{y}$$

- Substituting this result into equation (3) yields

$$\mathbf{w}(k+1) = \mathbf{w}(k) + \alpha[r(k) - \mathbf{w}^T(k)\mathbf{y}(k)]\mathbf{y}(k) \quad (4)$$

with the starting weight vector, $\mathbf{w}(1)$, being arbitrary

Delta Correction Algorithm

- By defining the change (delta) in the weight vector as

$$\Delta \mathbf{w} = \mathbf{w}(k+1) - \mathbf{w}(k)$$

we can write equation (4) in the form of a **delta correction algorithm**

$$\Delta \mathbf{w} = \alpha e(k) \mathbf{y}(k) \quad (5)$$

where

$$e(k) = r(k) - \mathbf{w}^T(k) \mathbf{y}(k) \quad (6)$$

is the error committed with $\mathbf{w}(k)$ when $\mathbf{y}(k)$ is presented

Delta Correction Algorithm

- If we change the weight vector in Equation (6) to $\mathbf{w}(k+1)$ the error becomes

$$e(k) = r(k) - \mathbf{w}^T(k+1)\mathbf{y}(k)$$

- The change in error then is

$$\begin{aligned}\Delta e(k) &= [r(k) - \mathbf{w}^T(k+1)\mathbf{y}(k)] - [r(k) - \mathbf{w}^T(k)\mathbf{y}(k)] \\ &= -[\mathbf{w}^T(k+1) - \mathbf{w}^T(k)]\mathbf{y}(k) \\ &= -\Delta\mathbf{w}^T\mathbf{y}(k)\end{aligned}$$

- Since $\Delta\mathbf{w} = \alpha e(k)\mathbf{y}(k)$ we have

$$\begin{aligned}\Delta e &= -\alpha e(k)\mathbf{y}^T(k)\mathbf{y}(k) \\ &= -\alpha e(k)\|\mathbf{y}(k)\|^2\end{aligned}$$

Delta Correction Algorithm

- Therefore, changing the weights reduces the error by a factor $\alpha ||\mathbf{y}(k)||^2$
- The next input starts the new adaption cycle, reducing the next error by a factor $\alpha ||\mathbf{y}(k+1)||^2$, and so on
- The choice of α controls stability and speed of convergence, a practical range for α is $0.1 < \alpha < 1.0$

Delta Correction Algorithm

- When the classes are separable, the algorithm *may* or *may not* produce a separating hyperplane, i.e. a mean square error solution does not imply a solution in the sense of the perceptron training theorem
- This uncertainty is the price of using an algorithm that converges under both the separable and nonseparable cases

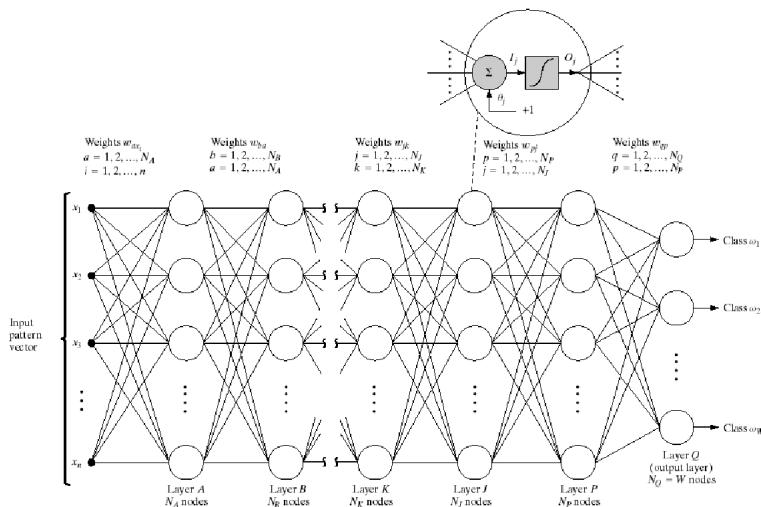
Review Question

- Apply the perceptron algorithm to the following pattern classes: $\omega_1 : \{[0, 0, 0]^T, [1, 0, 0]^T, [1, 0, 1]^T, [1, 1, 0]^T\}$ and $\omega_2 : \{[0, 0, 1]^T, [0, 1, 1]^T, [0, 1, 0]^T, [1, 1, 1]^T\}$ where $c = 1$ and $\mathbf{w}(1) = [-1, -2, -2, 0]^T$

Multilayer Feedforward Neural Networks

- A multilayer **feedforward neural network** can handle *multiclass* pattern recognition problems independent of whether or not the classes are separable
- These networks consist of layers of structurally identical computing nodes (neurons) arranged such that the output of every neuron in one layer feeds into the input of every neuron in the next layer

Multilayer Feedforward Neural Network Model



Basic Architecture

- The number of neurons in the first layer, called layer A , is N_A (often, $N_A = n$, the dimensionality of the input vectors)
- The number of neurons in the output layer, called layer Q , is denoted N_Q ($N_Q = W$, the number of classes that the neural network has been trained to recognize)
- The network recognizes an input vector \mathbf{x} as belonging to class ω_i if the i th output of the network is “high” while all other outputs are “low”

Basic Architecture

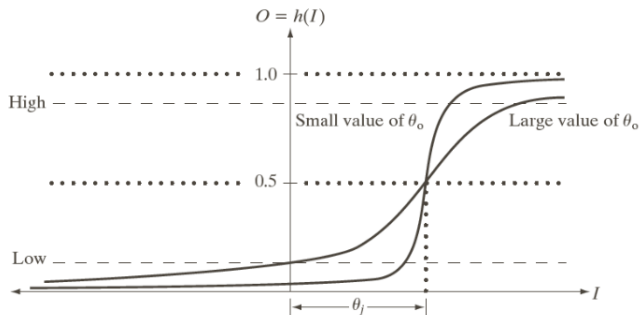
- Each neuron takes the same form as the perceptron model except that the hard-limiting activation function has been replaced by a soft-limiting **sigmoid** function

$$h_j(l_j) = \frac{1}{1 + e^{-(l_j + \theta_j)/\theta_0}} \quad (7)$$

where $l_j, j = 1, 2, \dots, N_j$, is the input to the activation element of each node in layer J of the network, θ_j is an offset, and θ_0 controls the shape of the sigmoid function

- Differentiability along all paths of the neural network is required in the development of the training rule

Basic Architecture



- The sigmoidal activation function

Basic Architecture

- The input to a node in any layer is the weighted sum of the outputs from the previous layer
- Letting layer K denote the layer preceding layer J gives the input to the activation element of each node in layer J , denoted I_j :

$$I_j = \sum_{k=1}^{N_k} w_{jk} O_k \quad (8)$$

for $j = 1, 2, \dots, N_j$, where N_j is the number of nodes in J , N_k is the number of nodes in K , and w_{jk} are the weights modifying the outputs O_k of the nodes in K before they are fed into the nodes in J

Basic Architecture

- The outputs of K are

$$O_k = h_k(I_k) \quad (9)$$

for $k = 1, 2, \dots, N_k$

- Substituting equation (8) into (7) yields the following form of the activation function

$$h_j(I_j) = \frac{1}{1 + e^{-(\sum_{k=1}^{N_k} w_{jk} O_k + \theta_j)/\theta_0}} \quad (10)$$

Training by Backpropagation

- During training, adapting the neurons in the output layer is a simple matter because the desired output of each node is known
- The main problem in training a multilayer network lies in adjusting the weights in the **hidden layers**, i.e. in those other than the output layer

Training by Backpropagation: Output Layer

- The total squared error between the desired responses, r_q , and the corresponding actual responses, O_p , of nodes in (output) layer Q , is

$$E_Q = \frac{1}{2} \sum_{q=1}^{N_Q} (r_q - O_q)^2 \quad (11)$$

where N_Q is the number of nodes in output layer Q

Training by Backpropagation: Output Layer

- The objective is to develop a training rule, similar to the delta rule, that allows adjustment of the weights in each of the layers in a way that seeks a minimum to equation (11)
- Adjusting the weights in proportion to the partial derivatives of the error w.r.t the weights achieves this result, i.e.

$$\Delta w_{qp} = -\alpha \frac{\partial E_Q}{\partial w_{qp}} \quad (12)$$

where layer P precedes layer Q , Δw_{qp} is as defined in equation (5), and α is a positive correction increment

Training by Backpropagation: Output Layer

- E_Q is a function of the outputs, O_q , which in turn are functions of the inputs I_Q
- Using the chain rule, we evaluate the partial derivative of E_Q as follows

$$\frac{\partial E_Q}{\partial w_{qp}} = \frac{\partial E_Q}{\partial I_q} \frac{\partial I_q}{\partial w_{qp}} \quad (13)$$

Training by Backpropagation: Output Layer

- From equation (8)

$$\frac{\partial I_q}{\partial w_{qp}} = \frac{\partial}{\partial w_{qp}} \sum_{p=1}^{N_p} w_{qp} O_p = O_p \quad (14)$$

- Substituting equations (13) and (14) into (12) yields

$$\Delta w_{qp} = -\alpha \frac{\partial E_Q}{\partial I_q} O_p = \alpha \delta_q O_p \quad (15)$$

where

$$\delta_q = -\frac{\partial E_Q}{\partial I_q} \quad (16)$$

Training by Backpropagation: Output Layer

- In order to compute $\partial E_Q / \partial I_q$, we use the chain rule

$$\delta_q = -\frac{\partial E_Q}{\partial I_q} = -\frac{\partial E_Q}{\partial O_q} \frac{\partial O_q}{\partial I_q} \quad (17)$$

- From equation (11)

$$\frac{\partial E_Q}{\partial O_q} = -(r_q - O_q) \quad (18)$$

and from equation (9)

$$\frac{\partial O_q}{\partial I_q} = \frac{\partial}{\partial I_q} h_q(I_q) = h'_q(I_q) \quad (19)$$

Training by Backpropagation: Output Layer

- Substituting equations (17) and (18) into (16) gives

$$\delta_q = (r_q - O_q)h'_q(I_q) \quad (20)$$

which is proportional to the error quantity $(r_q - O_q)$

- Substitution of equations (15) through (17) into (14) finally yields

$$\begin{aligned} \Delta w_{qp} &= \alpha(r_q - O_q)h'_q(I_q)O_p \\ &= \alpha\delta_q O_p \end{aligned} \quad (21)$$

Training by Backpropagation: Output Layer

- After $h_q(I_q)$ has been specified, all the terms in equation (21) are known or can be observed in the network, i.e. upon presentation of any training pattern to the input of the network, we know what the desired response, r_q , of each output node should be
- The value O_q of each output node can be observed as can I_q , the input to the activation elements of layer Q , and O_p , the output of the nodes in layer P
- Thus, we know how to adjust the weights that modify the links between the last and next-to-last layers of the network

Training by Backpropagation: Hidden Layers

- Continuing to work our way back from the output layer, we now analyze what happens at layer P
- Proceeding in the same manner yields

$$\begin{aligned}\Delta w_{pj} &= \alpha(r_p - O_p)h'_p(I_p)O_j \\ &= \alpha\delta_p O_j\end{aligned}\tag{22}$$

where the error term is

$$\delta_p = (r_p - O_p)h'_p(I_p)\tag{23}$$

Training by Backpropagation: Hidden Layers

- With the exception of r_p , all the terms in equations (21) and (22) either are known or can be observed in the network
- The term r_p makes no sense in a hidden layer because we do not know what the response of a hidden node in terms of the pattern membership should be

Training by Backpropagation: Hidden Layers

- We may specify what we want the response r to be only at the outputs of the network where the final pattern classification takes place
- If we knew that information at hidden nodes, there would be no need for further layers
- Therefore, we have to find a way to restate δ_p in terms of quantities that are known or can be observed in the network

Training by Backpropagation: Hidden Layers

- Going back to equation (17)

$$\delta_q = -\frac{\partial E_p}{\partial I_p} = -\frac{\partial E_p}{\partial O_p} \frac{\partial O_p}{\partial I_p} \quad (24)$$

- The term $\partial O_p / \partial I_p$ presents no difficulties, as before it is

$$\frac{\partial O_p}{\partial I_p} = \frac{\partial h_p(I_p)}{\partial I_p} = h'_p(I_p) \quad (25)$$

which is known once h_p is specified because I_p can be observed

Training by Backpropagation: Hidden Layers

- The term that produced r_p was the derivative $\partial E_p / \partial O_p$ so this term must be expressed in a way that does not contain r_p
- Using the chain rule, we write the derivative as

$$\begin{aligned} -\frac{\partial E_p}{\partial O_p} &= -\sum_{q=1}^{N_Q} \frac{\partial E_p}{\partial I_q} \frac{\partial I_q}{\partial O_p} = \sum_{q=1}^{N_Q} \left(-\frac{\partial E_p}{\partial I_q} \right) \frac{\partial}{\partial O_p} \sum_{p=1}^{N_p} w_{qp} O_p \\ &= \sum_{q=1}^{N_Q} \left(-\frac{\partial E_p}{\partial I_q} \right) w_{qp} \\ &= \sum_{q=1}^{N_Q} \delta_q w_{qp} \end{aligned} \tag{26}$$

Training by Backpropagation: Hidden Layers

- Substituting equations (24) and (25) into (23) yields the desired expression for δ_p :

$$\delta_p = h'_p(l_p) \sum_{q=1}^{N_Q} \delta_q w_{qp} \quad (27)$$

- δ_p can be computed now because all its terms are known
- Thus, equations (21) and (26) completely establish the training rule for layer P

Training by Backpropagation: Hidden Layers

- The importance of equation (27) is that it computes δ_p from the quantities δ_q and w_{qp} , which are terms that were computed in the layer immediately following layer P
- After the error term and weights have been computed for layer P , these quantities may be similarly used to compute the error and weights for the layer immediately preceding P
- In other words, we have found a way to propagate the error back into the network starting with the error at the output layer

Training by Backpropagation Summary

- For any layers K and J , where layer K immediately precedes layer J , compute the weights w_{jk} , which modify the connections between the these two layers by using

$$\Delta w_{jk} = \alpha \delta_j O_k \quad (28)$$

Training by Backpropagation Summary

- If layer J is the output layer, δ_j is

$$\delta_j = (r_j - O_j)h'_j(I_j) \quad (29)$$

- If layer J is a hidden layer and layer P is the next layer (to the right), then δ_j is given by

$$\delta_j = h'_j(I_j) \sum_{p=1}^{N_p} \delta_p w_{jp} \quad (30)$$

for $j = 1, 2, \dots, N_j$

Training by Backpropagation Summary

- Using the activation function in equation (10) with $\theta_0 = 1$ yields

$$h'_j(l_j) = O_j(1 - O_j)$$

in which case equations (29) and (30) assume the following attractive forms

$$\delta_j = (r_j - O_j)O_j(1 - O_j)$$

for the output layer, and

$$\delta_j = O_j(1 - O_j) \sum_{p=1}^{N_p} \delta_p w_{jp}$$

for the hidden layers, $j = 1, 2, \dots, N_J$

Feedforward Neural Network Summary

- Equations (28) through (30) constitute the generalized delta rule for training a multilayer feedforward neural network
- The process starts with an arbitrary (but not all equal) set of weights throughout the network
- Then, application of the generalized delta rule at any iterative step involves two basic phases

Feedforward Neural Network Summary

- In the first phase, a training vector is presented to the network and is allowed to propagate through the layers to compute the output O_j for each node
- The outputs O_q of the nodes in the output layer are then compared against their desired responses, r_p , to generate the error terms δ_q
- In the second phase, a backward pass through the network during which the appropriate error signal is passed to each node and the corresponding weight changes are made

Feedforward Neural Network Summary

- In a successful training session, the network error decreases with the number of iterations and the procedure converges to a stable set of weights that exhibit only small fluctuations with additional training
- A pattern has been classified correctly during training if the response of the node in the output layer associated with the pattern class from which the pattern was obtained is high, while all the other nodes have outputs that are low

Feedforward Neural Network Summary

- After training, all feedback paths are disconnected and the network classifies patterns using the parameters established during the training phase
- Any input pattern is allowed to propagate through the various layers, and the pattern is classified as belonging to the class of the output node that was high, while all the others were low
- If more than one output is labeled high, then the choice is one of declaring a misclassification or simply assigning the pattern to the class of the output node with the highest numerical value

Example: Object Recognition



Shape 1



Shape 2



Shape 3



Shape 4



Shape 1



Shape 2



Shape 3



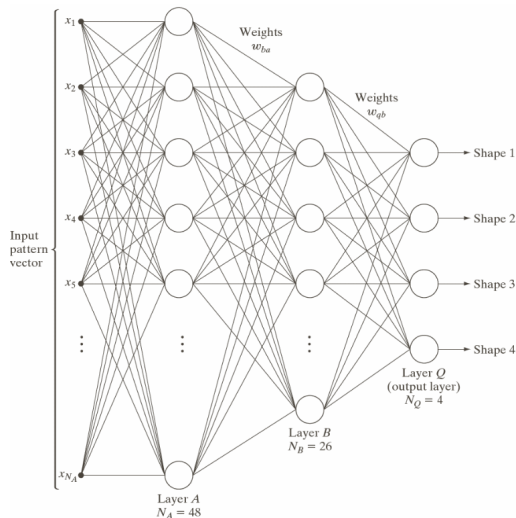
Shape 4

- A feedforward neural network can be trained to recognize reference shapes and their noisy versions

Example: Object Recognition

- Pattern vectors are generated by computing the normalized signatures of the shapes and then obtaining 48 uniformly spaced samples of each signature
- The number of nodes in the first layer is chosen to be 48, corresponding to the dimensionality of the input vectors, the second layer is heuristically specified as having 26 nodes

Example: Object Recognition



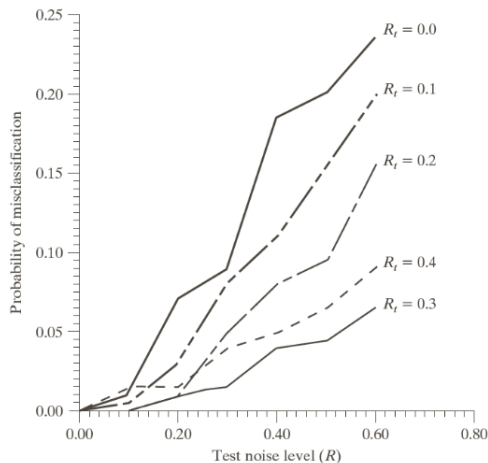
Example: Object Recognition

- In the first part of training, the weights are initialized to small random values with zero mean and the network is trained on the noise-free samples
- The network is said to have learned the shapes from all four classes when for any training pattern from class ω_i , the elements of the output layer yields $O_i \geq 0.95$ and $O_q \leq 0.05$ for $q = 1, 2, \dots, N_Q; q \neq i$

Example: Object Recognition

- In the second part of training with noisy samples, each contour pixel in a noise-free shape was assigned a probability V of retaining its original coordinate, and $R = 1 - V$ of being randomly assigned to the coordinates of one of its eight neighboring pixels
- The test set consists of 100 noisy patterns of each class generated by varying R between 0.1 and 0.6, giving a total of 400 patterns

Example: Object Recognition



Review Question

- Two pattern classes in 2D are distributed in such a way that the patterns of class ω_1 lie randomly along a circle of radius r_1 . Similarly, the patterns of class ω_2 lie randomly along a circle of radius r_2 , where $r_2 = 2r_1$. Specify the structure of a neural network with the minimum number of layers and nodes needed to properly classify the patterns of these two classes.

Neural Networks for Images

- In computer vision, why can't we just flatten an image and feed it through a multilayer neural network?
- Images are high dimensional vectors, it would take an enormous amount of parameters to characterize the network

Neural Networks for Images

- A multilayer neural network uses one perceptron for each input (e.g. one per image pixel multiplied by three for RGB images)
- The amount of weights quickly becomes unmanageable, e.g. for a 64×64 color image there are 12,288 weights that must be trained
- To address this problem, a **convolutional neural network** (CNN) reduces the number of parameters with an architecture specifically adapted for vision tasks

Neural Networks for Images

- A CNN is very similar to a traditional feedforward neural network in that they are comprised of nodes (neurons) that self-optimize through learning
- The major difference between a CNN and other neural networks is that CNNs are primarily designed and used for pattern recognition within images

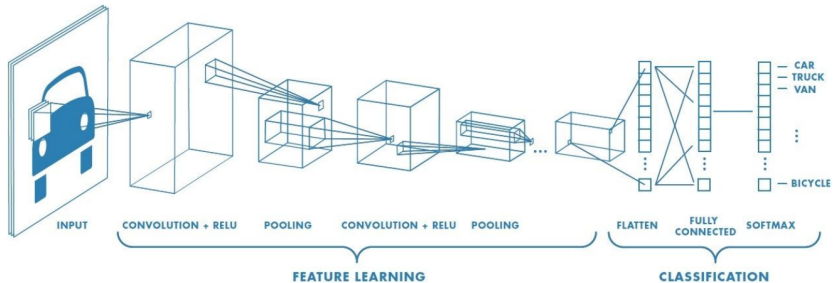
Overfitting

- **Overfitting** occurs when a machine learning algorithm models the training data too well and thus negatively impacts the performance of the model on new data
- For neural networks, increasing the number of hidden layers does not help overfitting since we do not have unlimited computational power and time for training
- The less parameters required to train the less likely the network will overfit, therefore improving the performance of the model

CNN Architecture

- The nodes within a CNN are organized into height, width, and depth spatial dimensions (e.g. an input layer can have dimensionality $64 \times 64 \times 3$ with a final output layer of dimensionality $1 \times 1 \times n$ where n represents the possible number of classes)
- CNNs are comprised of three types of layers: **convolutional**, **pooling**, and **fully connected**

CNN Architecture



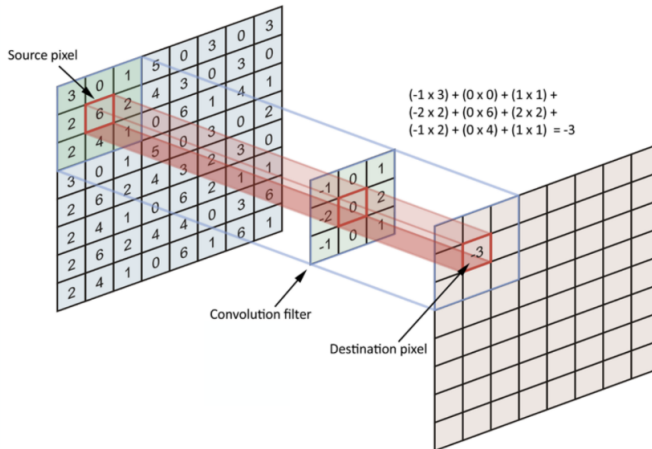
CNN Architecture

- The basic functionality of a CNN can be broken down into four key components
 - ① The input layer contains the pixel values of the image
 - ② The convolutional layer determines the output of the nodes which are connected to local regions of the input
 - ③ The pooling layer performs downsampling to further reduce the number of parameters
 - ④ The fully connected layers perform the same operations as a feedforward neural network
- Through these four components, a CNN transforms the input to produce class scores for classification and regression purposes

Convolutional Layer

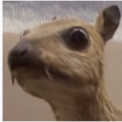

- The convolutional layer focuses on the use of learnable kernels (filters)
- When data arrives at a convolutional layer it is convolved with a kernel to produce an **activation map**
- As the kernel is applied to the input, a scalar product is computed for each value in that kernel

Convolutional Layer





Convolutional Layer

Edge detection

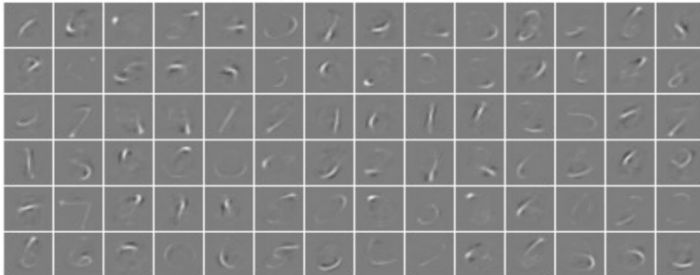
 * $\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$ = 

Sharpen

 * $\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$ = 

Kernel

Convolutional Layer



- Activations take from the first convolutional layer after training on the MNIST database of handwritten digits

Convolutional Layer

- To prevent the model from becoming too big to effectively train, every node in the convolutional layer is only connected to a small region of the input
- The dimensionality of this region is commonly referred to as the **receptive field size** of the node
- For example, given an input RGB image of size $64 \times 64 \times 3$, if we set the receptive field size to 6×6 then we have a total of 108 weights for each node (compared to 12,288 weights in a standard neural network)

Convolutional Layer

- Convolutional layers are also able to significantly reduce the complexity of the model using three hyperparameters: **depth**, **stride**, and **zero padding**
 - Depth - The number of nodes within each layer, reducing the number of nodes can drastically reduce the size of the model, but it can also reduce the pattern recognition capabilities
 - Stride - The amount by which the kernel shifts across the input, a small stride results in a heavily overlapped receptive field and large activations
 - Zero Padding - The process of padding the border of the input, it provides an effective method for further control over the dimensionality of output

Rectified Linear Units (ReLU)

- After each convolution layer, it is common practice to apply a nonlinear layer
- The purpose of this layer is to introduce nonlinearity into the system, typically using a rectified linear unit (ReLU), which combats the vanishing gradient problem occurring in sigmoids
- The idea of the ReLU is simple: negative values are set to zero and positive values remain unchanged

Pooling Layer

- The pooling layer aims to gradually reduce the dimensionality of the representation thus further reducing the number of parameters in the network
- In many CNNs, a max pooling layer with a 2×2 kernel and stride of 2 is applied, this scales the activation map down to 25% of the original size

Pooling Layer

Max Pooling

29	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

2 x 2
pool size

100	184
12	45

Average Pooling

31	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

2 x 2
pool size

36	80
12	15

Fully Connected Layer

- The fully connected layer contains nodes which are directly connected to the nodes in the adjacent layer
- This is analogous to the way in which the nodes are arranged in a feedforward neural network

What Does a CNN Learn?

- Each layer in a CNN learns filters of increasing complexity
- The first layers learn basic feature detection filters: edges, corners, etc.
- The middle layers learn filters that detect parts of objects (i.e. for faces they may learn to respond to eyes, noses, etc.)
- The last layers have higher-level representations, they learn to recognize full objects in different positions, orientation, and scale

Summary

- Traditional multilayer neural networks do not scale well for images, ignore information pertaining to the pixel position and correlation with neighbors, and cannot handle translations
- A CNN can intelligently adapt to the properties of an image since a pixel's position and neighborhood have semantic meaning, and elements of interest can appear anywhere in the image