Neural Networks

Shree K. Nayar

Monograph: FPCV-5-4

Module: Perception

Series: First Principles of Computer Vision

Computer Science, Columbia University

June, 2025

FPCV Channel

FPCV Website

In this lecture we will describe what a neural network is, how such a network can be constructed, and how it can be used to solve a wide range of detection and recognition problems.

Neural Networks

Shree K. Nayar
Columbia University

Topic: Neural Networks, Module: Perception
First Principles of Computer Vision

1

First, let us look at a few common visual recognition problems. Say that our goal is to develop a system for detecting and recognizing faces. We have already seen a few approaches to this problem. One is the use of Haar features and support vector machines (SVM) to classify images as either face images or non-face images. We have also discussed the use of principal component analysis (PCA) to compute eigenfaces, which can be used to both detect and recognize faces. In short, we know how to develop fairly robust systems for detecting and recognizing faces.

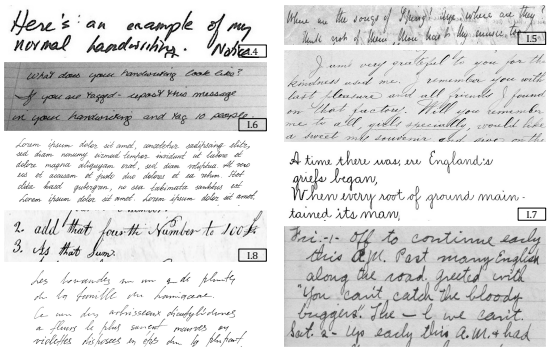
Detecting Faces

Support Vector Machine, Principal Component Analysis, etc.

2

Here is a more challenging problem: recognizing handwritten text. Shown here are various samples of handwritten text, and we can see tremendous variability between them. If we look at a single letter, we can see that it appears very differently from one person's writing to the next. Clearly, this is a hard recognition problem, but one that is critical in the context of optical character recognition.

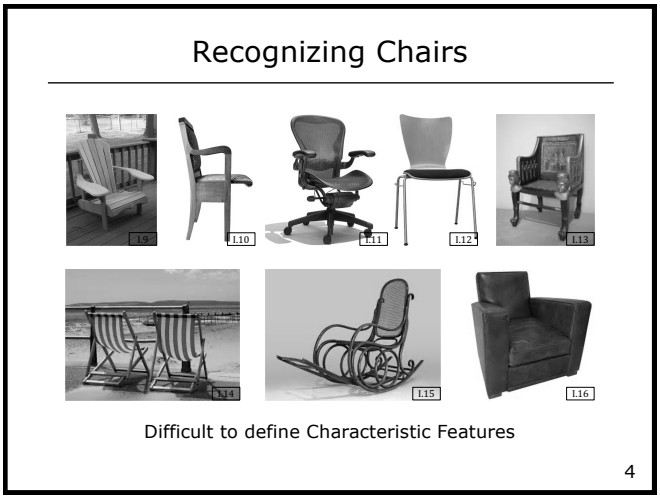
Recognizing Handwritten Text



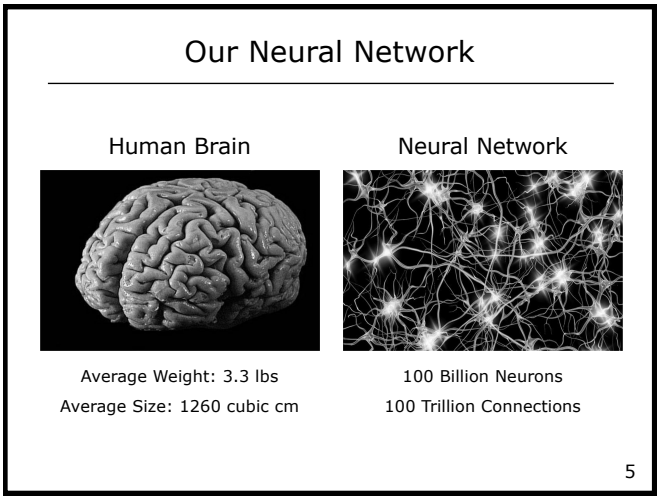
High Variability in Characters

3

Here is a recognition problem that on first glance appears simple. We want to develop a system with a very specific purpose—to recognize chairs. All of these chairs have the same function but are very different in appearance as they differ dramatically in terms of their 3D geometry and their material properties. How do we humans recognize chairs with ease, and how do we build a system that can do the same?

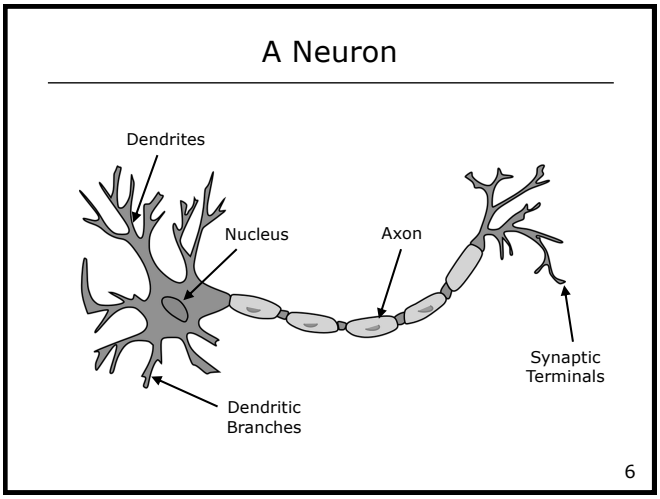


That brings us to the human brain. While we are not good at making precise quantitative measurements, we are great at making qualitative judgements such as recognizing all the objects in slide 4 as chairs.

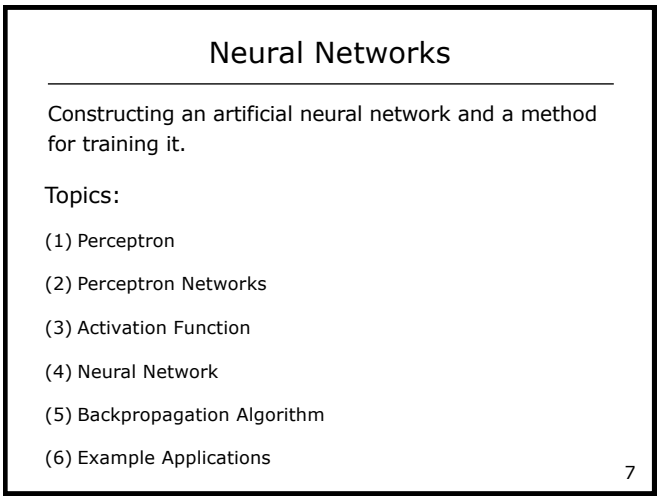


The brain has an average weight of about 3.3 pounds and an average volume of about 1200 cubic centimeters. Although the brain only accounts for about 2% of a person's body weight, it consumes about 20% of the energy owing to all the processing it does. As shown on the right, the brain is nothing but a network of neurons. There are approximately 100 billion neurons and roughly 100 trillion connections between the neurons. This complex network enables us to seamlessly perform the wide range of perception tasks we rely on to function in the world.

The basic building block of the human brain is the neuron. It is a single nerve cell with a nucleus that receives signals from other neurons via branches called dendrites. It takes all the inputs from its dendrites and performs a simple computation to produce an output, which it transmits as electrical impulses, or action potentials, via a thin fiber called the axon. The axon can vary in length from one millimeter to one meter. The other end of the axon has synaptic terminals which are used to make connections with dendrites of other neurons. The synaptic terminals transfer the electrical impulses that they receive via the axon, to dendrites of other neurons that are connected to them, using a process that is both electrical and chemical that involves potassium and sodium ions. From a computational perspective, the neuron is simple in terms of the operation it performs. However, when very large numbers of neurons come together to create a network, they can achieve complex mappings from inputs to outputs.



In this lecture, we will start by discussing a simple type of neuron, called a perceptron. We will show that a perceptron behaves like a linear classifier, and that a network of perceptrons functions like a complex linear classifier.



We know that linear classifiers are very useful, but they are limited in the class of problems they can solve. To make the network more powerful, we need to modify the perceptron, in particular the function that maps its inputs to its output, which is called the activation function. By changing the activation function of the perceptron from a step function to a sigmoid function, we get a neuron that has attractive mathematical properties. We present the architecture of a typical neural network and describe how the network can be trained using gradient descent to perform a task.

For even a relatively small network, the use of gradient descent for training can prove computationally prohibitive. We present the backpropagation algorithm, which uses the chain rule of derivatives, to dramatically reduce (by orders of magnitude) the computations needed to train a network. Finally, we conclude with a few examples of the use of neural networks to solve vision problems.

Perceptron

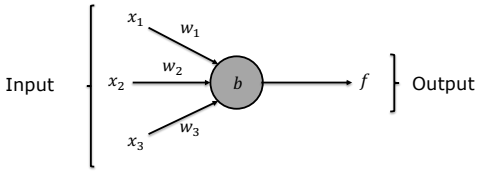
Shree K. Nayar
Columbia University

Topic: Neural Networks, Module: Perception
First Principles of Computer Vision

8

Perceptron

Takes several inputs and gives a single binary output (Decision)



$$f = \begin{cases} 0 & \text{if } \sum_j w_j x_j \le -b \\ 1 & \text{if } \sum_j w_j x_j > -b \end{cases} \qquad f = \begin{cases} 0 & \text{if } \boldsymbol{w}.\boldsymbol{x} + b \le 0 \\ 1 & \text{if } \boldsymbol{w}.\boldsymbol{x} + b > 0 \end{cases} \quad \boxed{1}$$

where, $w_j = weights$
$b = bias\ (threshold)$

[Rosenblatt 1958]

9

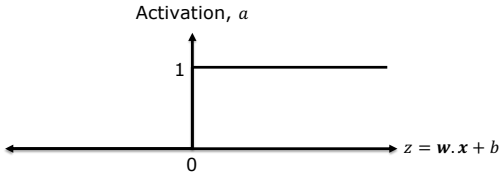
A perceptron takes in several inputs, shown as x_1 through x_3 here, and produces a binary output, f. It does this by multiplying the inputs with weights, w_1 through w_3, finding the sum, and comparing it with a threshold b, which is also called a bias. If the sum is greater than the bias, the output f of the perceptron is 1, and if the sum is less than the bias, it is 0. Given a perceptron with weights w_j and bias b, we can write all the weights as a vector $\boldsymbol{w}$ and all the inputs as a vector $\boldsymbol{x}$ to get the expression $\boxed{1}$.

We denote $\boldsymbol{w}.\boldsymbol{x} + b$ in expression $\boxed{1}$ in slide 9 as z. The perceptron essentially applies a function, called an activation function, to z to produce its output. This activation function is the step function, or the heavy side function.

Activation Function of a Perceptron

Let: $z = \boldsymbol{w}.\boldsymbol{x} + b$

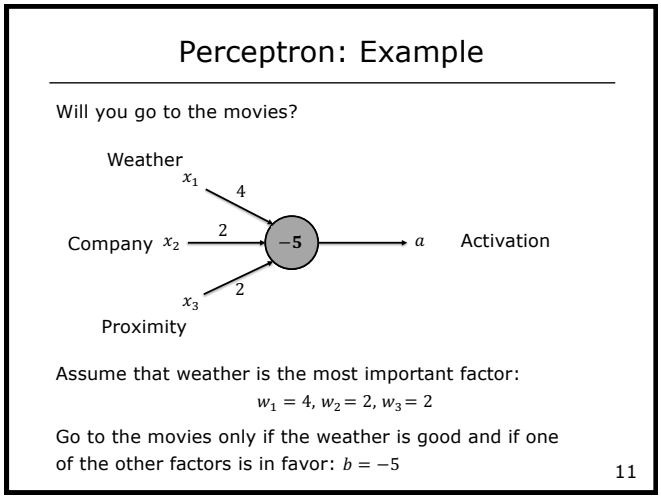
Activation: $a = f(z) = \begin{cases} 0 & \text{if } z \le 0 \\ 1 & \text{if } z > 0 \end{cases}$



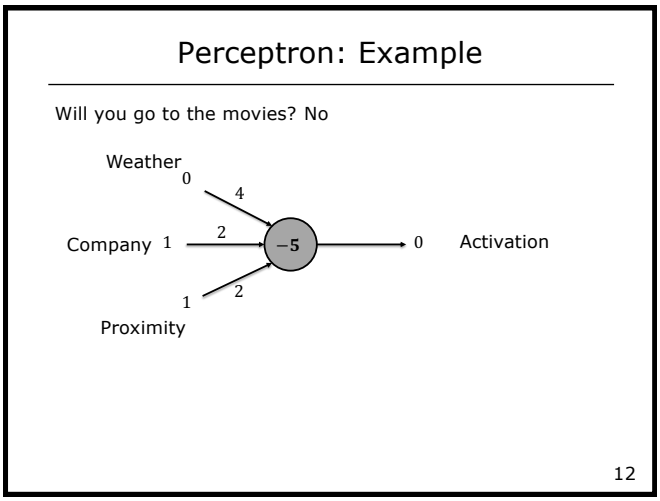
Activation Function for a Perceptron is a Step Function.

10

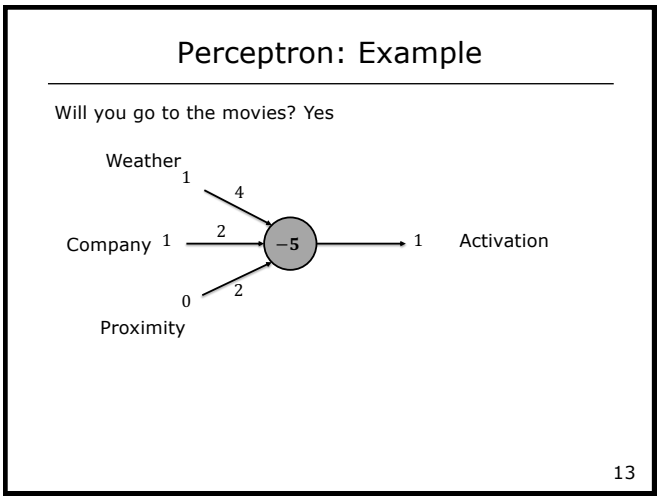
Let us look at what we can do with a single perceptron. Here is a simple example. Say, we wanted to decide whether to go to the movies. We want to make this decision based on three factors: whether the weather is good, whether we have company or not, and whether we are close to the theater or not. Let us assume that the weather is the most important factor. That is, if the weather is bad, we are not going to the movies, irrespective of the other two factors. We can assign a relatively large weight (say, 4) for the weather and lower weights (say, 2 and 2) for the other two factors. Let us say that we are willing to go to the movies only if the weather is good and at least one of the other two factors is favorable. To achieve that, we can choose a bias of -5.



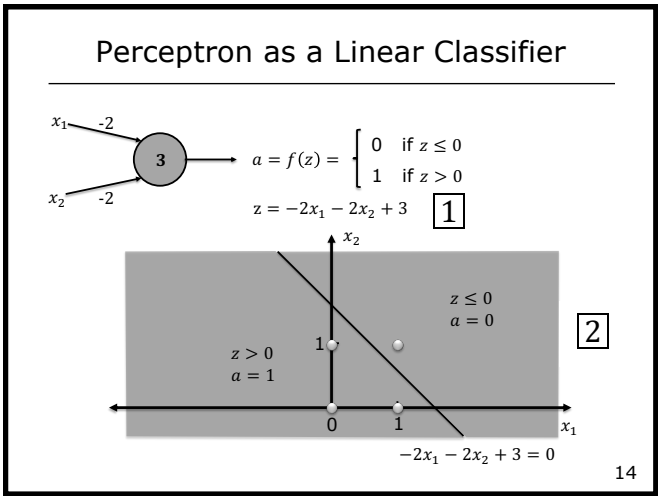
Let us see how this perceptron functions. If the weather is bad but we have company and the theater is close by, we have 1 times 2 plus 1 times 2, which is 4. 4 minus 5 (the bias) is -1, which is less than zero. Therefore, the output is zero, which means we will not go to the movies.



Now, if the weather is good and we happen to have company, then we have 1 times 4 plus 1 times 2, which is 6. 6 minus 5 (the bias) is 1, which is greater than zero. Therefore, the output is one. In this case, the decision is to go to the movies.

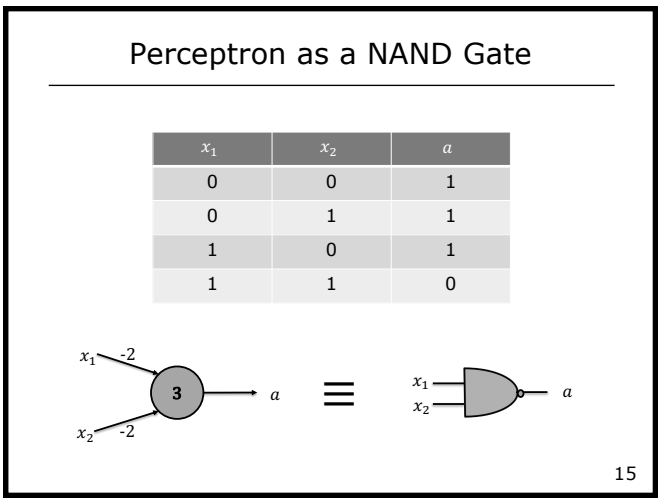


Now, let us take a closer look at what a single perceptron does. Here is a perceptron with two inputs, x_1 and x_2, with equal weights of -2, and a bias of 3. In this case, z is given by expression 1. The output a of the perceptron is the step function applied to z.



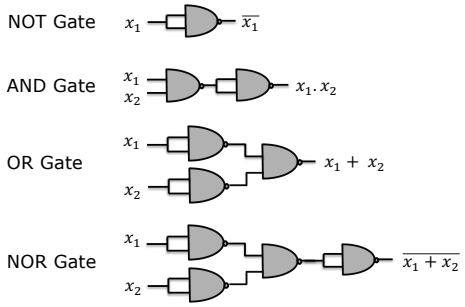
Let us look at the space of inputs (x_1, x_2), which is shown at the bottom 2. Note that the right-hand side of equation 1 is a straight line in the input space. If (x_1, x_2) lies on, or to the right of, the straight line, z will be less than or equal to zero, and therefore the output a will be zero. On the other hand, if the input (x_1, x_2) lies on the left side of the line, then z is greater than zero and the output a will be one. Therefore, the perceptron behaves like a linear classifier as it uses a straight line to split the two-dimensional input space.

Now let us assume that the inputs to the above perceptron are binary: 0 or 1. Then, there are only four possible inputs: $(0,0)$, $(1,0)$, $(0,1)$, and $(1,1)$. From slide 14, we know what the output is going to be for each of these inputs. As shown in the table, the output will be 1 for the input $(0, 0)$, 1 for $(0, 1)$, 1 for $(1, 0)$, and 0 for $(1, 1)$. Therefore, this perceptron behaves exactly like a NAND logic gate, which is shown on the right.



This is interesting because we know that a NAND gate is a universal logic gate. This means that all the other logic gates that are familiar to us (NOT, AND, OR, and NOR) can be constructed using one or more NAND gates. Therefore, since a NAND gate can be implemented using a perceptron, perceptrons are universal for computation.

NAND: Universal Logic Gate

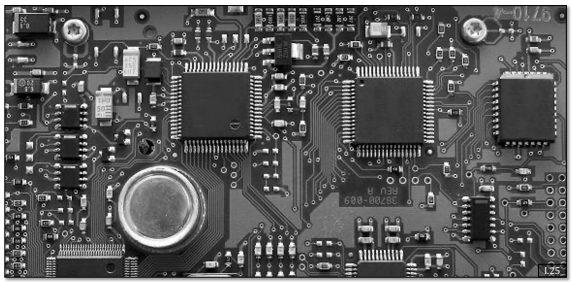


Thus, Perceptrons are Universal for Computation.

16

What this really means is that, given any digital logic circuit, i.e., a circuit that can be constructed using the above gates, it can be modeled as a network of perceptrons, regardless of its complexity.

Perceptrons: Computational Universality



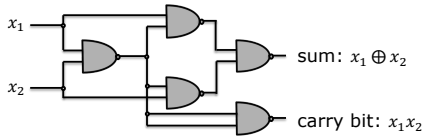
Any Digital Logic Circuit can be implemented using Perceptrons

17

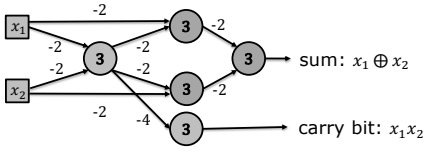
As an example, let us look at a simple digital logic circuit, namely, a 1-bit adder. The circuit for the adder has two inputs, x_1 and x_2, a series of NAND gates, and two outputs. One output is the sum, or the exclusive OR, of x_1 and x_2, and the second output is the carry bit, which is the AND of x_1 and x_2. Since each one of these NAND gates can be represented using the perceptron in slide 15, we can create a perceptron network that is exactly equivalent to the 1-bit adder.

Perceptrons: 1-bit Adder

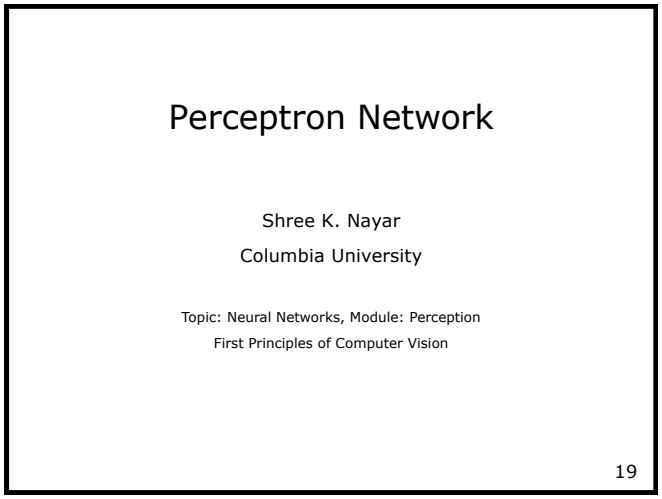
Digital Logic Circuit

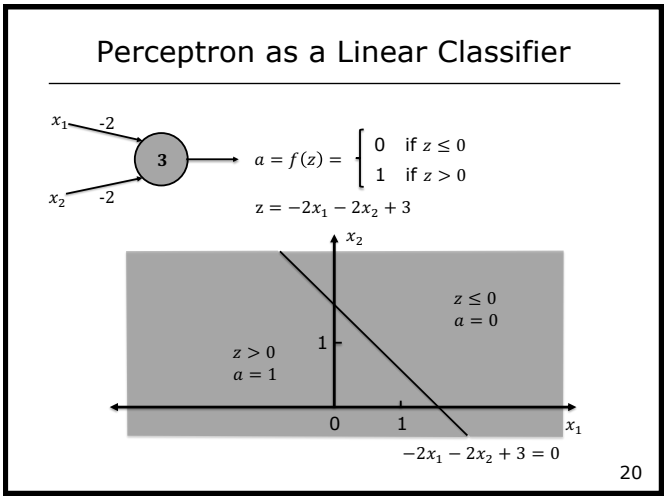


Equivalent Perceptron Network

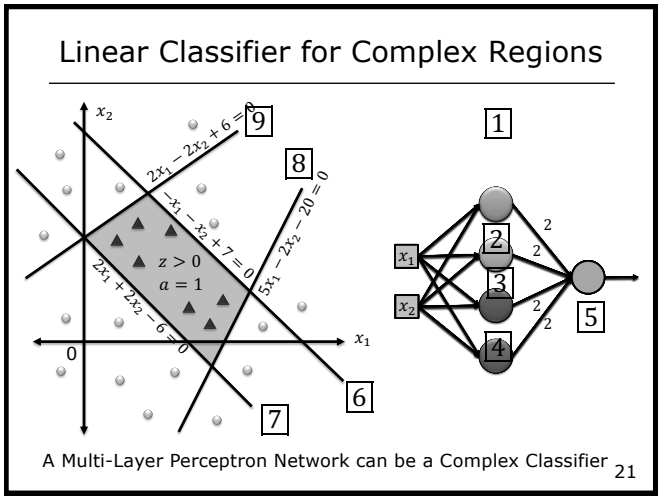


18





Once again, consider the single perceptron shown on the right. We know that its decision boundary is a straight line and the position and orientation of this line can be varied using the weights and biases of the perceptron. Let us now explore what we can do with a network of perceptrons.

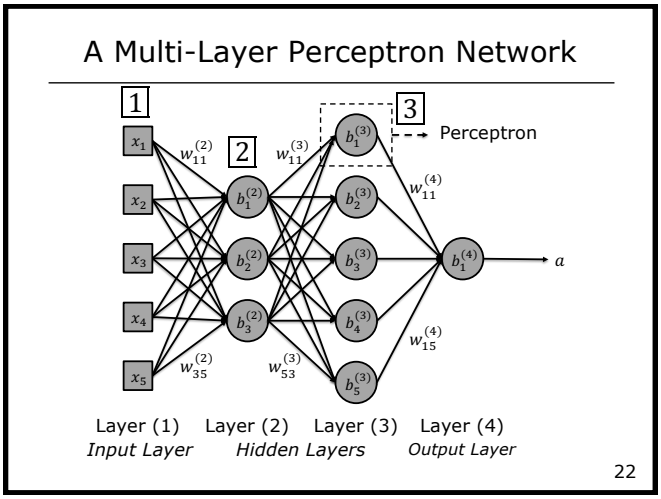


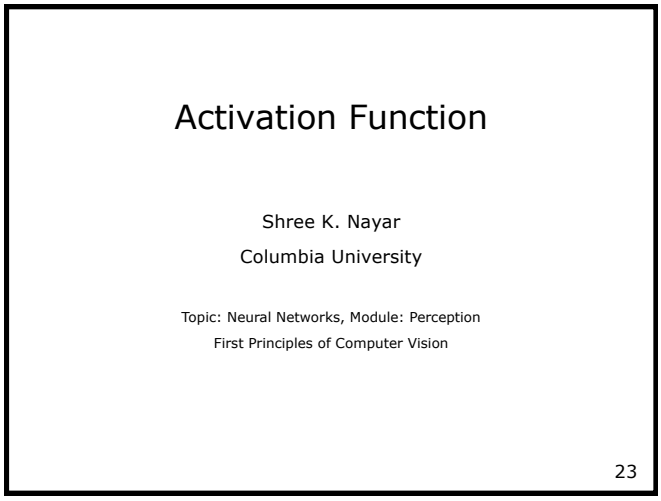
Now consider the following problem. We are given a set of points that belong to two classes (triangles and dots) in a two-dimensional input space (x_1, x_2). We wish to build a classifier that can distinguish between the triangles and the dots. That is, given a novel input, we wish to determine whether it lies in shaded region or not.

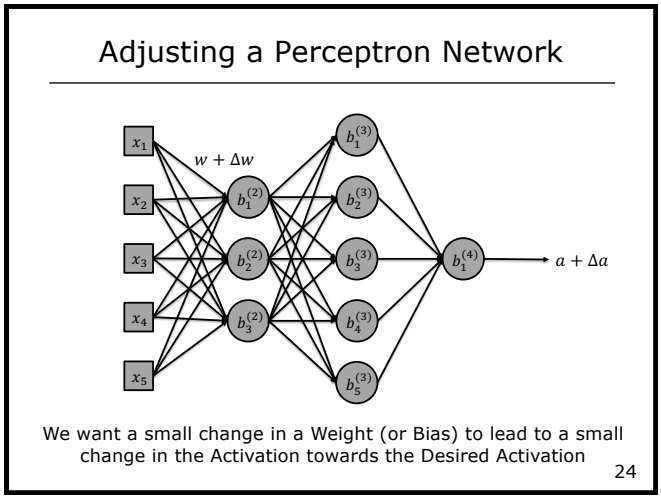
Consider the perceptron network shown on the right. Let us look at the perceptron labeled 1. We can choose its weights and bias such that it serves as a linear classifier corresponding to the line 6. The perceptron's output will be one only when the input lies below line 6. Similarly, the weights and biases of perceptrons 2, 3, and 4 are chosen to create classifiers 7, 8, and 9, respectively.

We know that when an input lies in the shaded region, all four perceptrons would produce an output of 1. To produce the final output, we use perceptron 5 whose inputs are the outputs of perceptrons 1, 2, 3, and 4. If we set the four weights of perceptron 5 to 2, and its bias to -7, it will produce an output of 1 only when all the four perceptrons behind it produce an output of 1. We therefore have constructed a perceptron network that serves as a linear classifier for a complex region. While our input space is two-dimensional in this example, such a classifier can be created for an input space of any dimensionality.

We can also construct a perceptron network that has multiple layers. The first layer in a multi-layer network is referred to as the input layer 1, which are not perceptrons but rather just the inputs to the network. In the example shown here, we have two layers of perceptrons, 2 and 3, called the hidden layers. This network happens to be a fully connected one, meaning that each perceptron in it takes as input the output of every perceptron in the preceding layer. The final output layer has a single perceptron, the output of which is the final output of the classifier. If an application requires multiple outputs, the output layer would have multiple perceptrons.

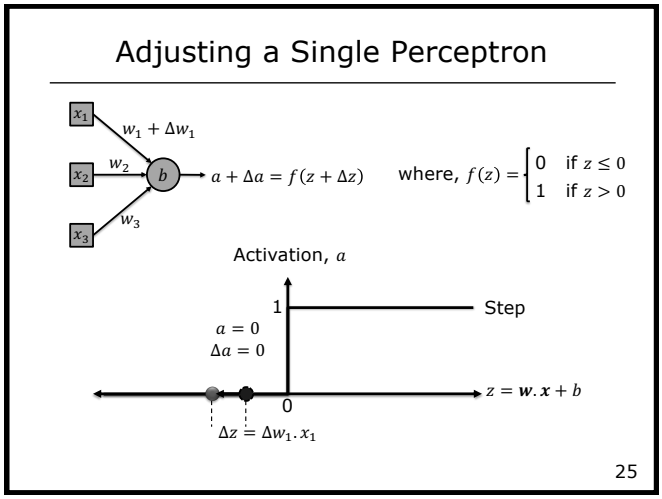


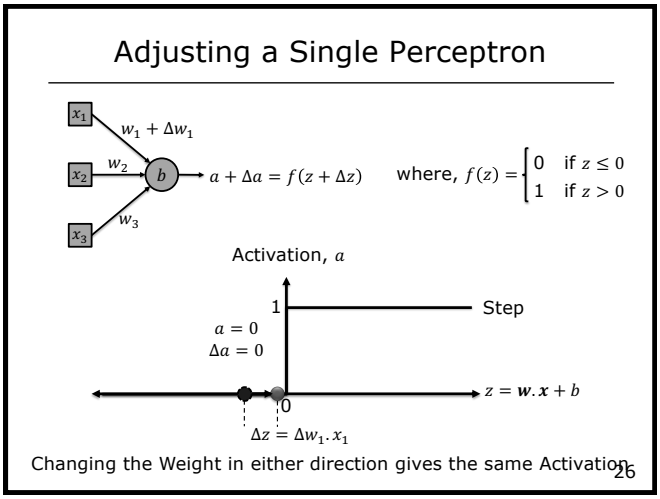


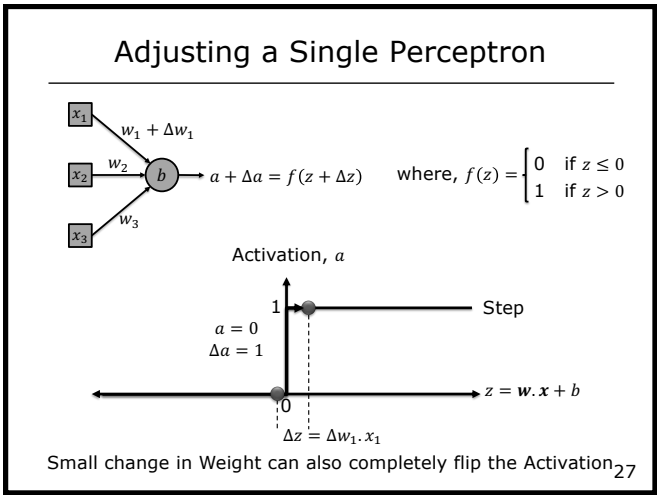


As discussed before, the activation function of a perceptron is the step function (also called the heavy side function). We will first discuss some fundamental challenges posed by the use of the step function, and then show how these challenges can be addressed by modifying the step function.

On the right is the perceptron network we described earlier. Ultimately, we want to find the weights and biases (the parameters) of this network to solve a given vision task. In other words, we want to adjust the parameters of the network so that we eventually end up with a network that produces a desired output for any given input. To find the optimal parameters of the network, we need the network to satisfy the property that a small change in any given parameter produces a small change in the output of the network.

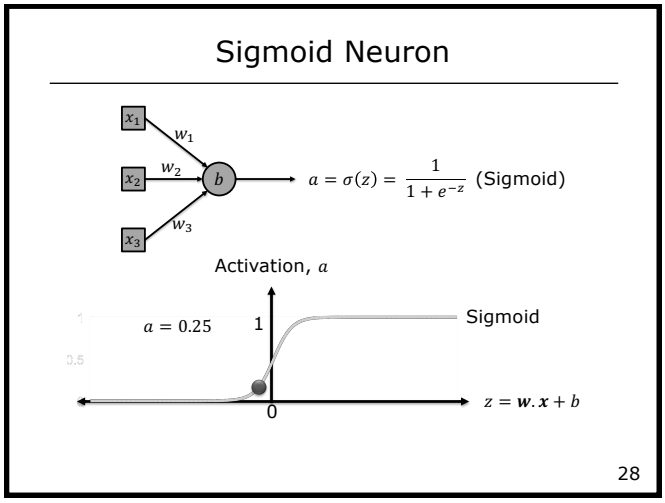


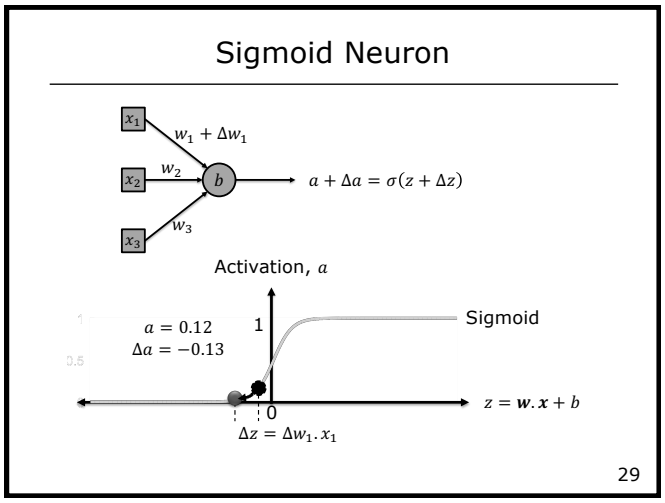


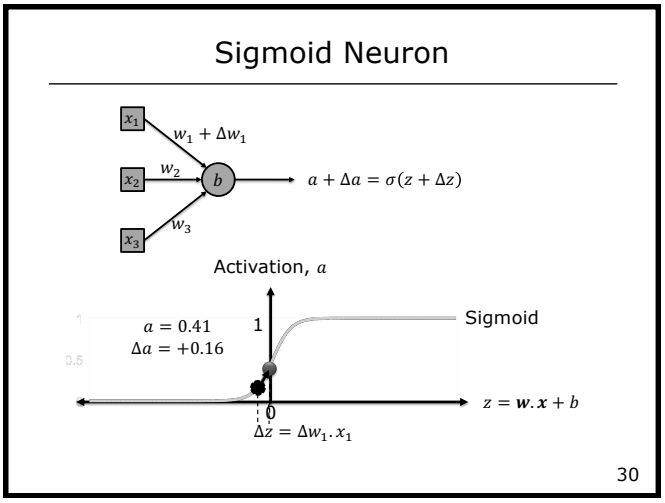


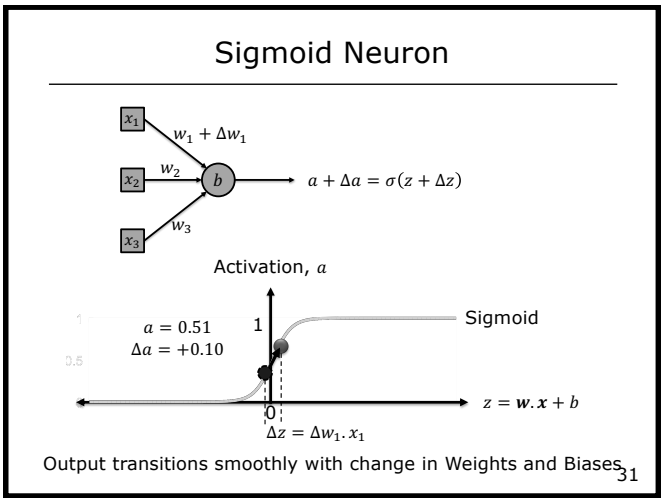
Consider the perceptron in slide 25. Let us see what happens to the output a when we make a change to the weight w_1. We have z which is $\boldsymbol{w}.\boldsymbol{x}$ plus b. The output a is the result of the step function applied to z. We begin with z less than zero, which makes a equal to 0. When we change w_1 by adding Δw_1, assume that z drops further in value and hence a remains 0. If we instead subtract Δw_1 from w_1 as shown in slide 26, z increases in value but remains negative and hence a remains zero. In short, neither increasing nor decreasing w_1 by Δw_1 changes the output of the perceptron. Now, as shown in slide 27, if we further reduce w_1 by Δw_1, z becomes positive and suddenly a jumps to 1.

Therefore, in the case of a perceptron, a small change in a weight or a bias can either result in no change in the output or a sudden change in it. This consequence of using a step function as the activation function makes a network of perceptrons difficult to work with. Therefore, we would like to use a smoother activation function that makes it easier to interpret the effects of changing the parameters of a network.









That brings us to the sigmoid neuron. In this case, the activation function is a sigmoid function. As shown in slide 28, the sigmoid is a slightly blurred version of the step function, and can be written as 1 over 1 plus e to the power of -z.

In comparison to a perceptron, the behavior of a sigmoid neuron is more predictable. As before, if we change the parameter w_1 by adding Δw_1 to it (slide 29), we see that the output a reduces. We therefore know that if we want to increase a we need to subtract Δw_1 from w_1 (slides 30 and 31). In other words, in the case of a network made of sigmoid neurons, the output varies smoothly with respect to changes in the weights and biases of the network. This makes the network easier to work with compared to a network of perceptrons with the step activation function.

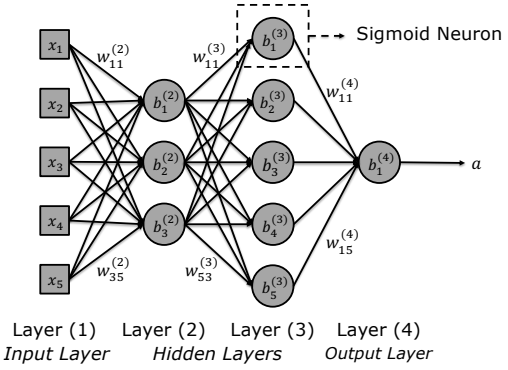
Neural Network

Shree K. Nayar
Columbia University

Topic: Neural Networks, Module: Perception
First Principles of Computer Vision

32

A Neural Network



33

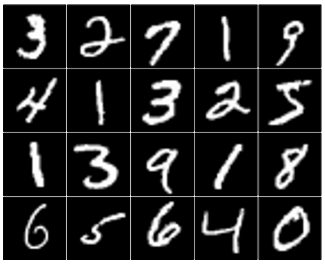
Now let us take a look at how we can construct a neural network and train it to solve a specific vision problem. Shown here is a simple neural network. We have replaced the perceptrons here with sigmoid neurons. As in slide 24, we have an input layer and two hidden layers (Layer (2) and Layer (3)), each with a set of neurons. In practice, we can use a deeper network with more hidden layers. As mentioned before, while we have a single output neuron here, we could have several output neurons if required by the task. Additionally, we have a fully connected network here, where each neuron takes as input the outputs of all the neurons in the previous layer. It could instead take as input only a subset of the outputs from the previous layer.

Now let us construct a neural network to solve a specific problem, namely, recognizing handwritten digits. To train our network we will use the MNIST database which has a large collection of segmented and binarized handwritten digits. Each digit sits in a bounding box that is 20 x 20 pixels in size. This image is centered in a larger image, which is 28 x 28 pixels in size.

From the examples shown here, we can see that this seemingly simple recognition problem is in fact challenging. For example, the three samples of the digit 3 are written in very different styles.

Recognizing Characters

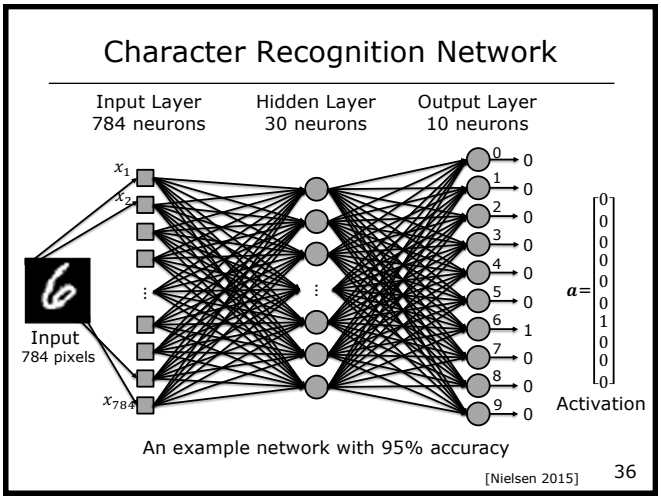
Recognize segmented images of handwritten decimal digits



[MNIST 1998]

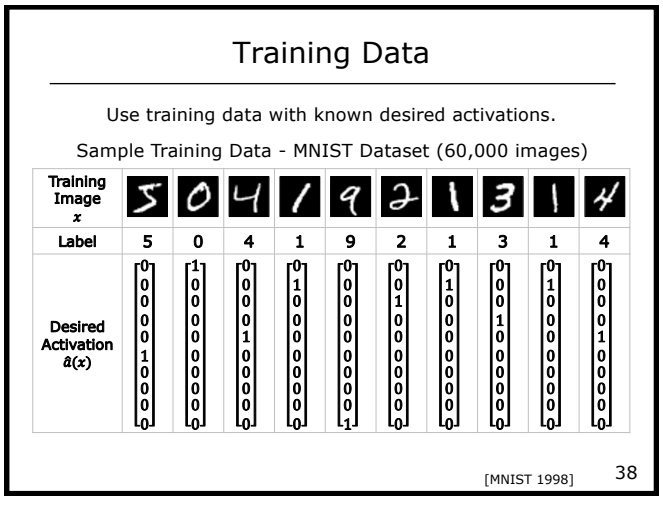
34

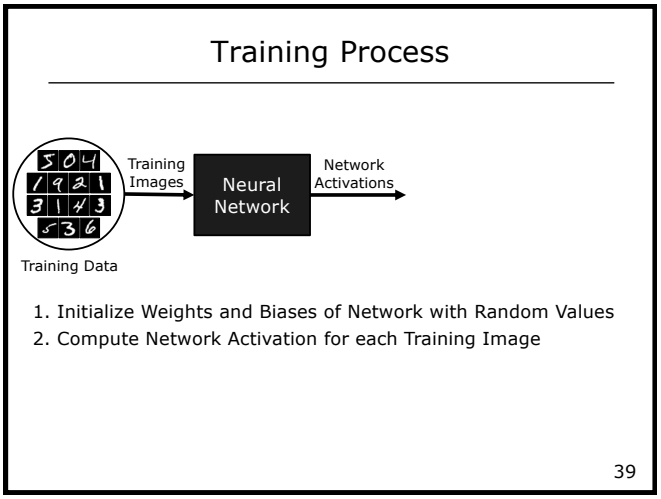
We want to configure this neural network such that it takes as input a handwritten digit and produce 10 outputs, one for each of the digits. In the example shown here, we would like the output to be high (1) for the digit 6, and low (0) for all the other digits. The digit in an input image must be represented the same way as the digits in the training images discussed in slide 34. That is, each input digit should be rescaled and centered within an image of size 28 x 28 pixels. The input image should also be binarized like the training images.

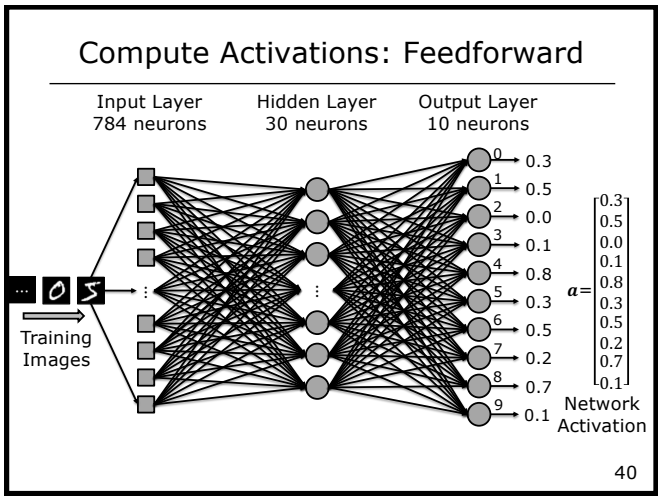


The network shown here was developed by Nielsen. It has an input layer with 784 (28 x 28 pixels) inputs, a single hidden layer with 30 neurons, and an output layer with 10 neurons, one for each of the 10 digits. This network has been shown to perform with an accuracy of about 95%, which is impressive given the difficulty of the problem and small size of the network.

Next, we describe the important process of training the above network. Our training data is the MNIST dataset discussed in slide 34, which has a total of roughly 60,000 images. Human experts were used to determine the digit in each training image. Each image $\boldsymbol{x}$ therefore is given a label that is represented as a desired activation vector $\widehat{\boldsymbol{a}}(\boldsymbol{x})$ that has 10 elements, with a 1 for the element that corresponds to the digit, and 0 elsewhere.

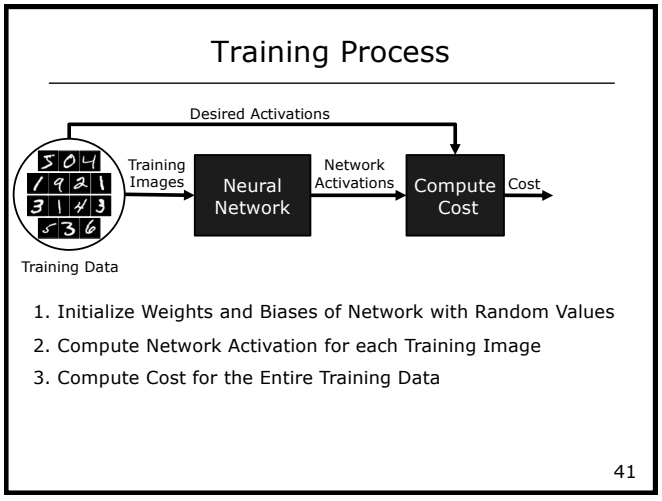




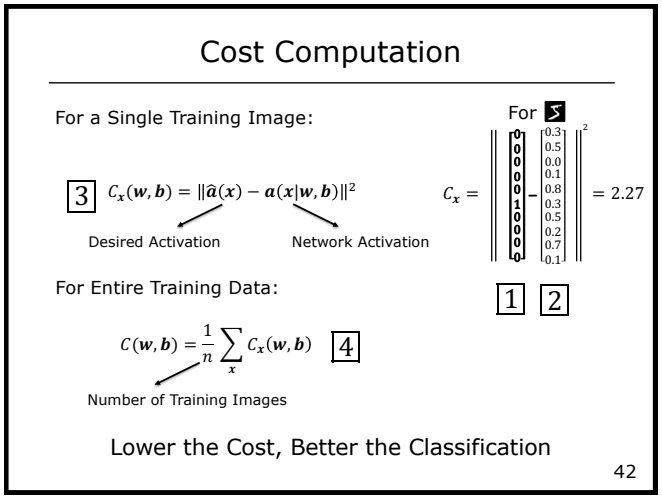


Now let us look at how the training process works. We begin by initializing the network with random weights and biases. Then, for each training image, we calculate the network activations (10 outputs). This is referred to as the feedforward process where, for each image, the input layer is computed and fed into the hidden layer. The outputs of the hidden layer are computed and fed into the output layer. Finally, the outputs (activations) of the output layer are computed.

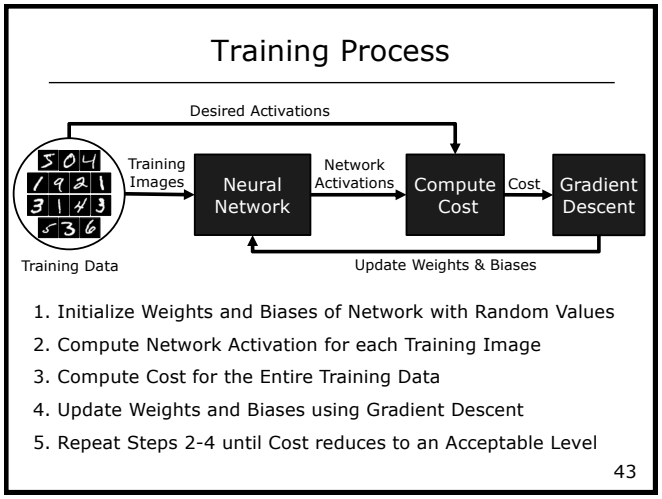
We now have an activation $\boldsymbol{a}$ for each of the training images. Since the parameters of the network were initialized with random values, the computed activations are not going to match the desired activations. By using all the computed activations, we are going to compute the cost for the entire training data. Our goal then is to iteratively adjust the parameters of the network to arrive at a minimum cost.

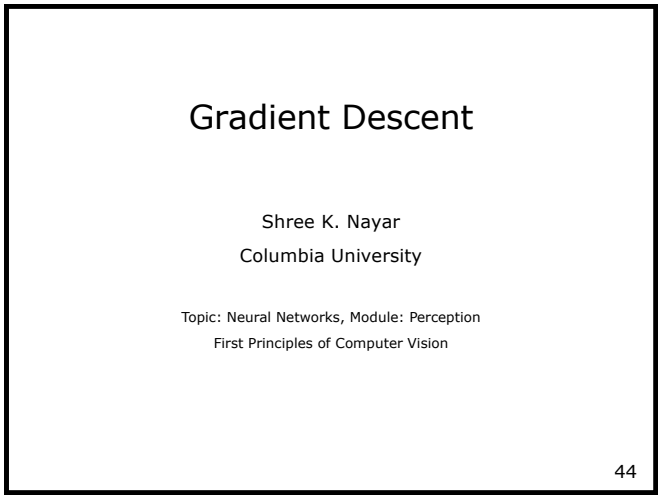


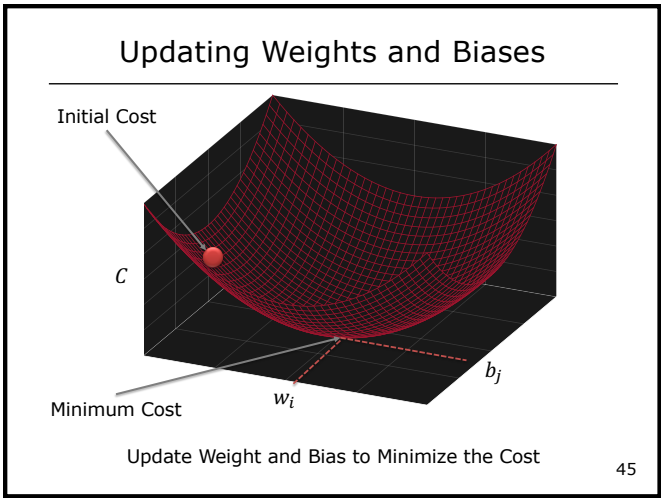
Let us take a look at how we compute the above cost. For any given training image $\boldsymbol{x}$, we have our desired activation vector $\hat{\boldsymbol{a}}$ [1], which has 0s for everything except the digit that $\boldsymbol{x}$ represents. Let us say that our random network provides us the activation vector $\boldsymbol{a}$ [2]. We take the Euclidean distance between the two vectors and square it to get the cost C_x for the image $\boldsymbol{x}$ [3]. Next, we compute the average cost C over the entire set of training images [4]. Note that the lower this cost is, the better the performance of the network.



Now that we have the average cost C, we want to adjust the weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ of the network so as to lower the average cost. The method we use for doing this is gradient descent, which we will describe shortly. It is a well-known iterative method for optimization, and it can be used to find the local minimum of any differentiable function. We repeat steps 2 through 4, until the cost C goes below some acceptable level, at which point the network is trained.







We are going to train our neural network by adjusting the weights and biases of the network using an optimization method known as gradient descent. As discussed in slide 43, our goal is to find the weights $\boldsymbol{w}$ and biases $\boldsymbol{b}$ of the network that minimize the average cost C. Let us first take the example of a network with two parameters, one weight w_i and one bias b_j. Let us assume that the function that relates C to w_i and b_j is the one shown in slide 45. We know the cost for specific values of w_i and b_j (red dot) and we wish to find the values of w_i and b_j that correspond to the minimum cost, i.e., the lowest point of the cost function. Since we don't know the cost function a-priori, our goal is to iteratively update w_i and b_j so that we eventually arrive at the minimum cost. We do this by moving w_i and b_j in the direction that is opposite to the gradient of C.

We can use Taylor series to estimate the gradient of the cost C at the current parameter values w_i and b_j. Let us say we have a two-dimensional function $f(x, y)$. If we want to find the value of f at x plus Δx and y plus Δy, the Taylor series tells us that it is approximately equal to the sum of $f(x, y)$, the derivative of f with respect to x times Δx, and the derivative of f with respect to y times Δy. Therefore, the change ΔC in the cost due to the change $(\Delta w_i, \Delta b_j)$ in the parameters is given by equation 1. This can be written in vector form as in equation 2. This, in turn, can be written as ∇C times $\Delta \mathbf{v}$, where ∇C is the gradient of C and $\Delta \mathbf{v}$ includes the parameter changes $(\Delta w_i, \Delta b_j)$ 3.

Change in Cost

From Taylor Series:

$$f(x + \Delta x, y + \Delta y) \approx f(x, y) + \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y$$

$$\Delta f(x, y) \approx \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial y}\Delta y$$

Applying to Cost Function:

$$\Delta C \approx \frac{\partial C}{\partial w_i}\Delta w_i + \frac{\partial C}{\partial b_j}\Delta b_j \quad 1$$

In Vector Form:

$$\Delta C = \begin{bmatrix} \frac{\partial C}{\partial w_i} & \frac{\partial C}{\partial b_j} \end{bmatrix} \cdot \begin{bmatrix} \Delta w_i \\ \Delta b_j \end{bmatrix} \quad 2$$

Change in Cost: $\Delta C = \nabla \boldsymbol{C} \,.\, \Delta \mathbf{v}$ 3

Gradient · Change in Weight and Bias

46

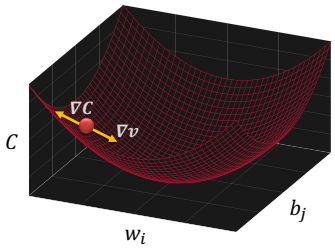
Gradient Descent

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_i} & \frac{\partial C}{\partial b_j} \end{bmatrix}, \quad \Delta \mathbf{v} = \begin{bmatrix} \Delta w_i \\ \Delta b_j \end{bmatrix}$$

$\Delta C = \nabla \boldsymbol{C} \,.\, \Delta \mathbf{v}$

Let $\Delta \mathbf{v} = -\nabla \boldsymbol{C}$ 1

$\Delta C = -\|\nabla \boldsymbol{C}\|^2$ 2



47

Gradient Descent

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w_i} & \frac{\partial C}{\partial b_j} \end{bmatrix}, \quad \Delta \mathbf{v} = \begin{bmatrix} \Delta w_i \\ \Delta b_j \end{bmatrix}$$

$\Delta C = \nabla \boldsymbol{C} \,.\, \Delta \mathbf{v}$

Let $\Delta \mathbf{v} = -\eta \nabla \boldsymbol{C}$

$\Delta C = -\eta \|\nabla \boldsymbol{C}\|^2$

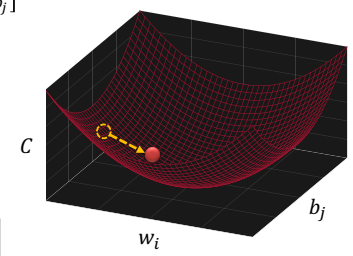
where η is the Learning Rate

For Each Step:

$$w_i \rightarrow w_i' = w_i - \eta \frac{\partial C}{\partial w_i}$$

$$b_j \rightarrow b_j' = b_j - \eta \frac{\partial C}{\partial b_j}$$

3



48

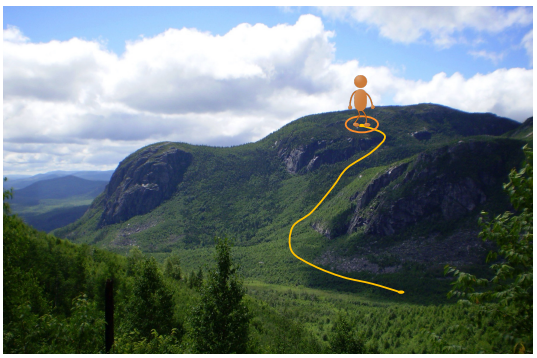
We know that ∇C, that is, the gradient of C, is the direction in which the cost function C is steepest. Since we wish to get to the minimum of the C, this suggests that we want to change our parameters in the direction opposite to ∇C. Therefore, we can set $\Delta\mathbf{v}$ equal to the negative of ∇C, as in equation $\boxed{1}$. Then, the change in cost, ΔC, would the negative of the square of the magnitude of ∇C, as in equation $\boxed{2}$. In summary, in each iteration of our gradient descend, we compute ∇C and then simply add -∇C to the current parameter values $\mathbf{v}$.

As shown in slide 48, we can control the rate at which we change the parameters $\mathbf{v}$ by introducing a parameter η, which is called the learning rate. The learning rate determines how fast we are going to descend the cost function C. With this modification, the update rule $\boxed{3}$ involves finding the derivatives of C with respect to w_i and b_j, multiplying the derivatives with η, and subtracting the results from the current parameter values w_i and b_j, respectively. This gives us our new parameter values, w_i' and b_j'. This process is repeated until we arrive at the minimum of C, at which point, the derivatives C with respect to w_i and b_j will equal zero.

Let us examine the effect of the learning rate η. If we use a very small value for η, we are going to move slowly towards the minimum of C. If we use a moderate value for η, we are going to move faster and converge quicker. If we use a large value for η, we may overshoot the minimum of the cost function and bounce around a bit before finding it. Even worse, we stand the risk of jumping over the minimum to an entirely different part of the cost function, thereby converging at a local minimum that is not the global minimum. In short, the learning rate must be chosen with care.

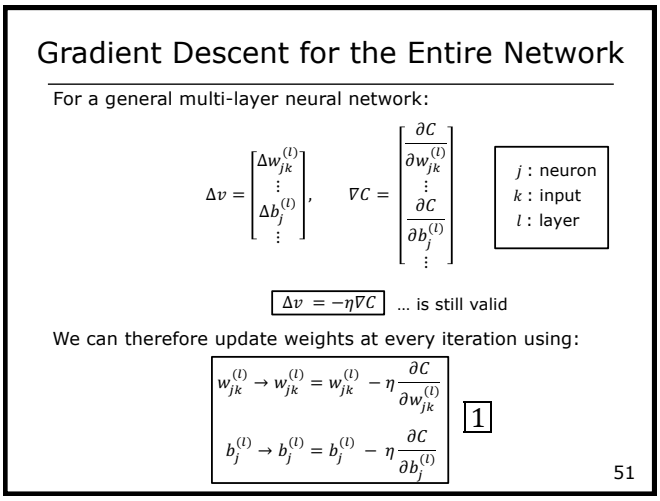
To gain a bit more intuition for how gradient descent works, let us look at this illustration. Assume you are at the top of the hill, as shown. Your goal is to get to the lowest point on the terrain. All you need to do is to find the gradient of the terrain right under your foot. This gradient will point in the direction of steepest climb, and so you take a step in the opposite direction. If you simply repeat this process, you will end up at the lowest point of the terrain.

Gradient Descent: The Intuition

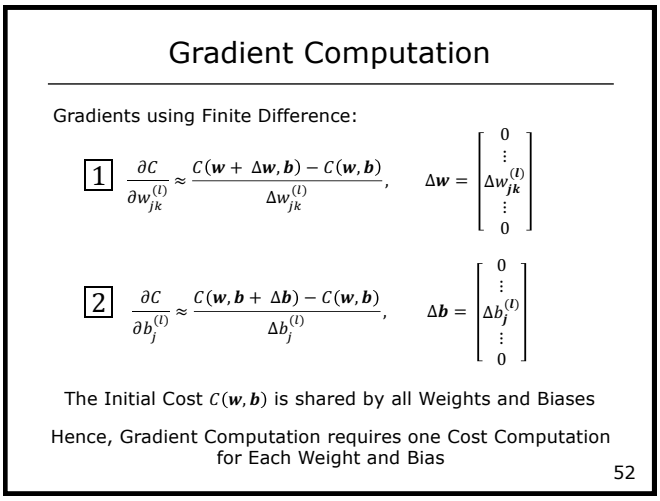


50

In slide 46, we considered a network with just two parameters, w_i and b_j. In a real network, we could have millions of parameters. It turns out that the approach described above works just the same for any number of parameters. For a general multilayered neural network, the change in parameters $\Delta\mathbf{v}$ and the gradient ∇C are both vectors with as many elements as there are weights and biases in the network. Here, ${w_{jk}}^{(l)}$ is the k^{th} weight of the j^{th} neuron in the l^{th} layer, and ${b_j}^{(l)}$ is the bias of the j^{th} neuron in the l^{th} layer. Once we have computed the gradient of C by computing the derivatives of C with respect to all the weights and biases, we can use the same update rule as in slide 48 to adjust all the weights and biases [1].



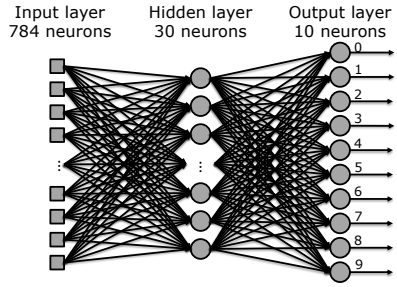
The one thing we have not discussed is how to compute the gradient C. For this, we use finite differences. We represent all the weights of the network as vector $\boldsymbol{w}$, and all the biases as a vector $\boldsymbol{b}$. Suppose we want to find the derivative of C with respect to the weight ${w_{jk}}^{(l)}$. We construct a parameter change vector $\Delta\boldsymbol{w}$ which is zero everywhere, except for the element corresponding to ${w_{jk}}^{(l)}$. Next, we compute the cost $C(\boldsymbol{w},\boldsymbol{b})$ and the cost $C(\boldsymbol{w}+\Delta\boldsymbol{w},\boldsymbol{b})$ using the feedforward computation described in slide 40. Then, we can find the derivative of C with respect ${w_{jk}}^{(l)}$ using equation [1]. Similarly, we can find the derivative of C with respect to a bias ${b_j}^{(l)}$ using equation [2]. Once the derivatives with respect to all the parameters of the network have been computed, we can use the update rule in slide 51 to adjust the parameters.



Note that in each iteration of gradient descent, $C(\boldsymbol{w},\boldsymbol{b})$ needs to be computed only once. However, $C(\boldsymbol{w}+\Delta\boldsymbol{w},\boldsymbol{b})$ needs to be computed as many times as there are weights, and $C(\boldsymbol{w},\boldsymbol{b}+\Delta\boldsymbol{b})$ needs to be computed as many times as there are biases. Clearly, this makes gradient descent extremely inefficient for any reasonably sized network.

Let us revisit Nielsen's network for the recognition of handwritten digits. In this case, since there are 23,820 weights and 40 biases, for each gradient computation, we need 23,861 cost computations. With that in mind, we can do a rough calculation of the number of computations involved in a single iteration of gradient descent while training this network. By computations, we mean multiplications, as additions are relatively inexpensive.

Gradient Computation Complexity



#Weights : 784 x 30 + 30 x 10 = 23820
#Biases : 30 +10 = 40

For Each Gradient Computation, we need
23861 Cost Computations

53

The number of multiplications needed to compute the cost for a single training image (using feedforward) equals the number of weights, which is 23,820. Since the number of training images in the MNIST dataset is 60,000, the number of multiplications required to compute the average cost is $1.4 \text{ x } 10^9$. The number of average cost computations needed to find the derivatives of the cost with respect to all the weights and biases is 23,861 (slide 53). Therefore, the total number of multiplications required for one iteration of gradient descent is $3.4 \text{ x } 10^{13}$, clearly a very big number! Can we do better?

Gradient Computation Complexity

# of Multiplications req. to Compute Cost for Single Image (C_x)	: 23,820
# of Training Images used to Compute the Average Cost (C)	: 60,000
# of Multiplications req. to compute Average Cost (C)	: $23{,}820 \text{ x } 60{,}000$ $1.4 \text{ x } 10^9$
# of Average Cost (C) Computations req. to Compute Gradient (∇C) w.r.t all Weights and Biases	: 23,861
Total # of Multiplications req. for One Iteration of Gradient Descent	: $23{,}861 \text{ x } 1.4 \text{ x } 10^9$ $3.4 \text{ x } 10^{13}$

Can we do better?

54

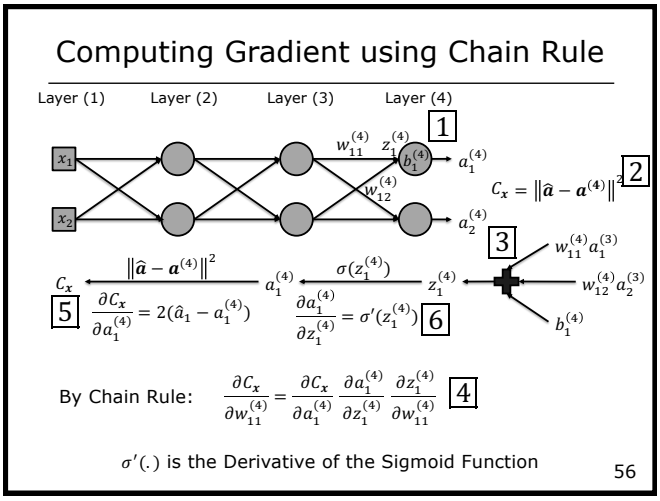
A major breakthrough in the realm of neural networks was the invention of the backpropagation algorithm, which dramatically reduces the complexity of gradient descent. This invention has been critical to the success that neural networks is having today. We will now describe backpropagation in detail.

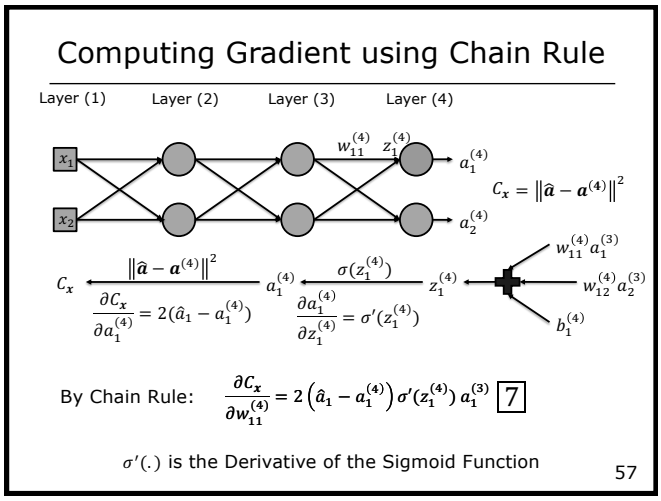
Backpropagation Algorithm

Shree K. Nayar
Columbia University

Topic: Neural Networks, Module: Perception
First Principles of Computer Vision

55





Let us see how backpropagation works. In this very simple network, we have two inputs (Layer 1): x_1, and x_2. We also have two hidden layers (Layers 2 and 3), with two neurons in each of them. Finally, we have an output layer (Layer 4) with two neurons. First, we are going to focus on one neuron in the output layer, marked as 1. Let us look at the cost of this network for an image $\boldsymbol{x}$. That cost, C_x, is the desired activation $\hat{\boldsymbol{a}}$ minus the actual activation $\boldsymbol{a}^{(4)}$ (see 2).

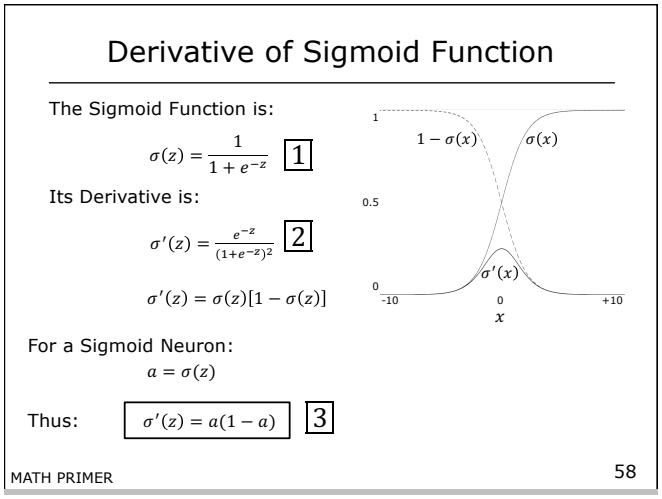
Now consider a single element of $\boldsymbol{a}^{(4)}$, namely, $a_1^{(4)}$. We know that $a_1^{(4)}$ depends on $z_1^{(4)}$ via the activation function, which is the sigmoid function, i.e., sigma of $z_1^{(4)}$. Now, $z_1^{(4)}$, in turn, depends on the activations of the previous layer, through the weights $w_{11}^{(4)}$ and $w_{12}^{(4)}$ and the bias $b_1^{(4)}$ of the neuron marked as 3.

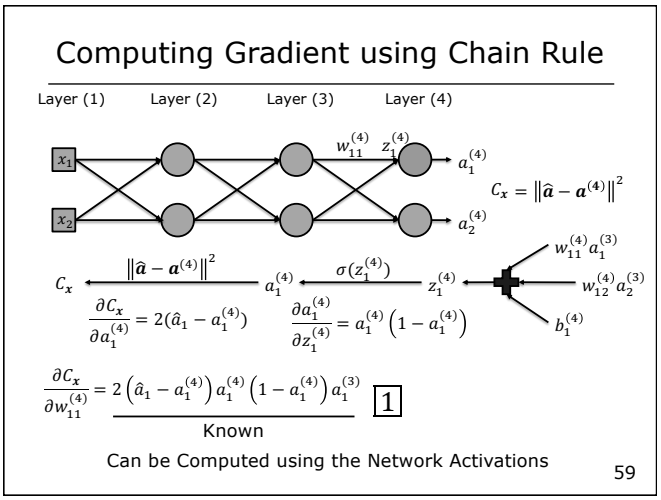
Let us pick one parameter $w_{11}^{(4)}$. We want to find the derivative of the cost C_x with respect to $w_{11}^{(4)}$, which can be expressed using the chain rule for derivates (see 4). It equals the derivative of C_x with respect to $a_1^{(4)}$ times the derivative of $a_1^{(4)}$ with respect to $z_1^{(4)}$, times the derivative of $z_1^{(4)}$ with respect to $w_{11}^{(4)}$.

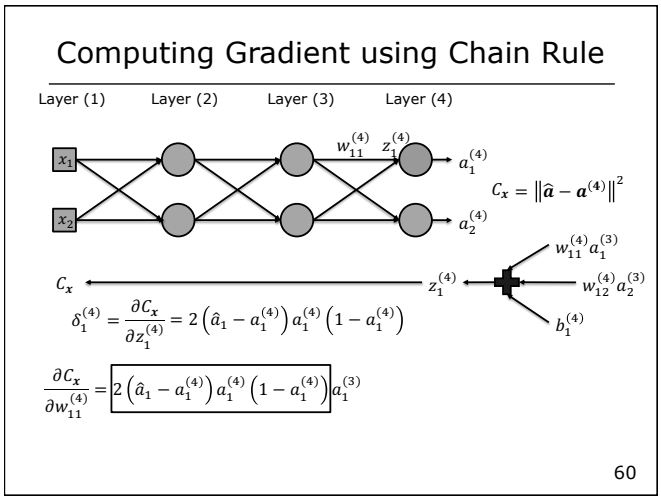
We would like to find the right substitutions for the three derivatives in 4. We can write out the expression for the first derivative 5 and show that it equals 2 times $\hat{a}_1$ minus $a_1^{(4)}$. The second derivative in 4 is the derivative of the sigmoid function with respect to $z_1^{(4)}$, which is given by 6.

If we plug 5 and 6 into 4, we are left with one derivative, namely, the derivative of $z_1^{(4)}$ with respect to $w_{11}^{(4)}$, which is $a_1^{(3)}$. The final expression for the derivative of the cost C_x with respect to $w_{11}^{(4)}$ is given by 7. Note that most of the terms involve activations, except one, which is the derivative of the sigmoid. This brings us to an important property of the sigmoid.

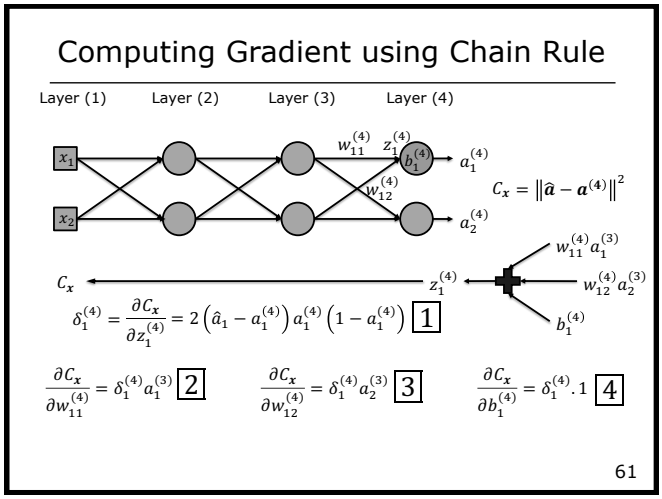
Here is our sigmoid function 1. Its derivative, which we will call σ' of *z*, is given by 2. This can be further simplified as the sigmoid of *z* multiplied by 1 minus the sigmoid of *z*. We know that the sigmoid of *z* is the activation a of the neuron. So, σ', the derivative of the sigmoid, is a times 1 minus a, or, in other words, the output of the neuron times 1 minus the output (see 3).

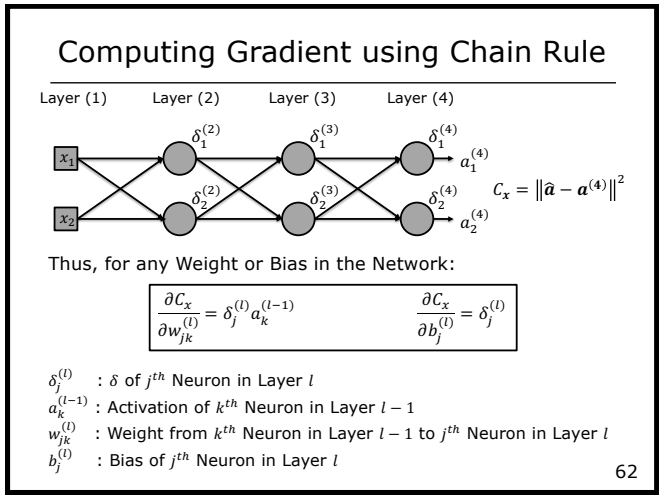


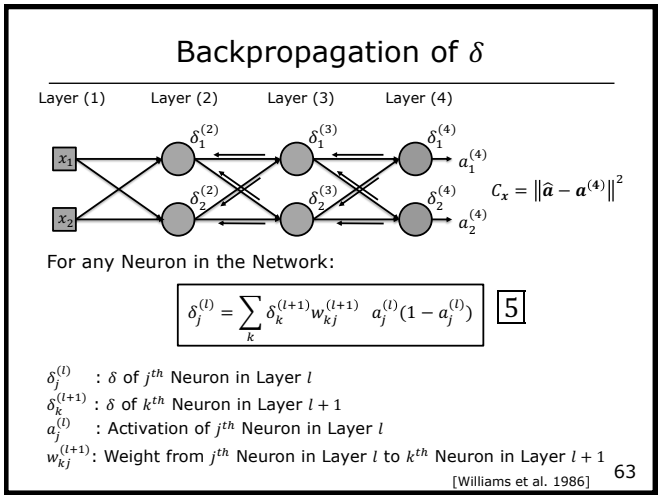


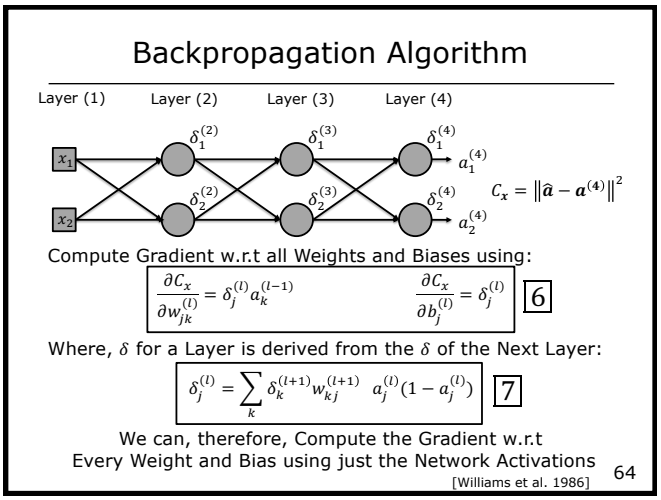


Now, in slide 57, we can replace σ' with $a_1^{(4)}$ times 1 minus $a_1^{(4)}$ to get 1. We have therefore found the derivative of the cost with respect to parameter $w_{11}^{(4)}$ without using any finite differences, but instead by computing the product of activations from Layer (4) and Layer (3).









In slide 60, all the terms (boxed) that include the activation $a_1^{(4)}$ in Layer 4 are denoted as $\delta_1^{(4)}$ in slide 61 (see 1). We see that $\delta_1^{(4)}$ can be used to compute the derivative of the cost C_x with respect to $w_{11}^{(4)}$ 2, $w_{12}^{(4)}$ 3 and $b_1^{(4)}$ 4. We have therefore found the derivatives of the cost with respect to all the parameters of the first neuron in the output layer. We can do the same for the parameters of the second neuron of the output layer as well. Note that all of these derivatives are computed without doing any finite differences!

More generally, as shown in slide 62, as long as we know the $\delta's$ for any layer, we can find the derivatives of the cost with respect to the parameters of the neurons for that layer. What is left is to determine the $\delta's$ for the layers that come before the output layer. Once again, by using the chain rule we can show that, if we know the $\delta's$ for any layer, we can find the $\delta's$ of the previous layer using the $\delta's$ of the current layer, the weights of the current later, and the activations of the previous layer (see 5). This process of computing all the derivatives of the cost function is called backpropagation.

Slide 64, summarizes the backpropagation algorithm. First, the $\delta's$ for all the layers are computed using expression 6. These $\delta's$ are then used to compute the cost derivatives respect to all the weights and biases of the network using the expressions in 7.

We can now discuss how gradient descent works with backpropagation. As before, we first initialize the network with random weights and biases. Then, for each training image, we compute the activations using feedforward (slide 40). Next, we compute all the $\delta's$ for the neurons in the output layer. Using them, we compute the $\delta's$ for all the previous layers (slide 63). Once we have all the $\delta's$, we know how to compute the derivatives of the cost with respect to all the weights and biases of the network (slide 63).

Gradient Descent with Backpropagation

1. Initialize Network with Random Weights and Biases
2. For each Training Image:
   a. Compute Activations for the Entire Network
   b. Compute δ for Neurons in the Output Layer using Network Activation and Desired Activation:
   $$\delta_j^{(L)} = 2\left(a_j^{(L)} - a_j^{(L)}\right) a_j^{(L)}(1 - a_j^{(L)})$$
   c. Compute δ for all Neurons in the previous Layers:
   $$\delta_j^{(l)} = \sum_k \delta_k^{(l+1)} w_{kj}^{(l+1)} a_j^{(l)}(1 - a_j^{(l)})$$
   d. Compute Gradient of Cost w.r.t each Weight and Bias for the Training Image using δ:
   $$\frac{\partial C_x}{\partial w_{jk}^{(l)}} = \delta_j^{(l)} a_k^{(l-1)} \qquad \frac{\partial C_x}{\partial b_j^{(l)}} = \delta_j^{(l)}$$

65

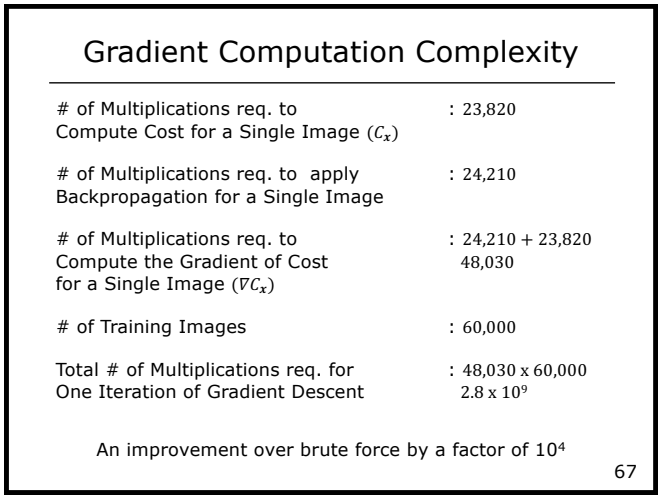
After repeating the above process for all the training images, we compute the average of each cost derivative over all the training images. These average derivatives are used to perform gradient descent, i.e., adjust all the weights and biases of the network. All of the above steps correspond to a single iteration of gradient descent. These steps are repeated until the cost of the network is below an acceptable level.

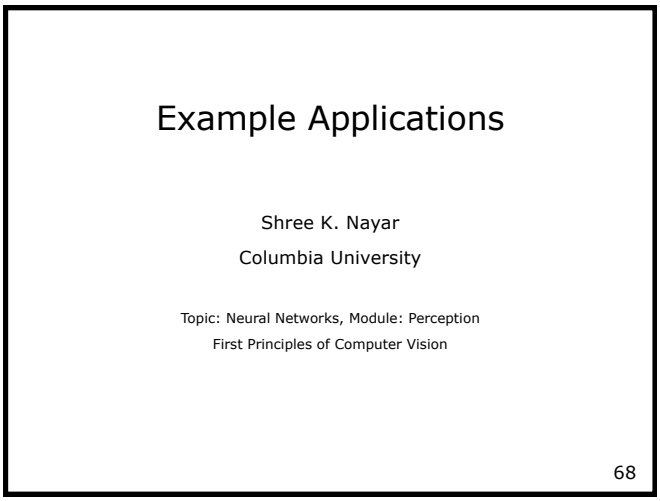
Gradient Descent with Backpropagation

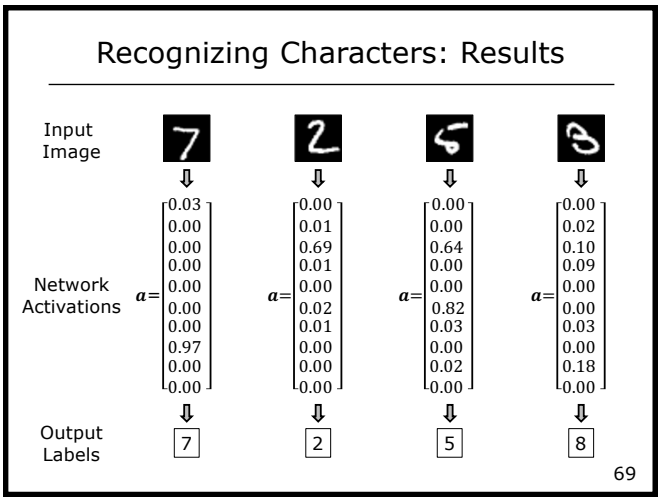
3. Average the Gradient w.r.t. each Weight and Bias over the Entire Training Set:
   $$\frac{\partial C}{\partial w_{jk}^{(l)}} = \frac{1}{n}\sum \frac{\partial C_x}{\partial w_{jk}^{(l)}} \qquad \frac{\partial C}{\partial b_j^{(l)}} = \frac{1}{n}\sum \frac{\partial C_x}{\partial b_j^{(l)}}$$
4. Update the Weights and Biases using Gradient Descent
   $$w_{jk}^{(l)} \to w_{jk}^{(l)} = w_{jk}^{(l)} - \eta \frac{\partial C}{\partial w_{jk}^{(l)}} \qquad b_j^{(l)} \to b_j^{(l)} = b_j^{(l)} - \eta \frac{\partial C}{\partial b_j^{(l)}}$$
5. Repeat Steps 2-4 till Cost reduces below an acceptable level

66

Let us now look at the computational complexity of training with backpropagation. Remember that without backpropagation we needed 3.4×10^{13} multiplications to train Nielsen's network using the MNIST dataset (slides 53 and 54). The number of multiplications required to compute the cost for a single image using feedforward is 23,820. The number of multiplications required to apply backpropagation for a single image is 24,210. The sum of these two (48,030) is the number of multiplications needed to compute the derivatives with respect to all the weights and biases for a single image. Since we have 60,000 training images, the total number of multiplications needed for a single iteration of gradient descent is 2.8×10^9, which is a significant improvement of 10^4.

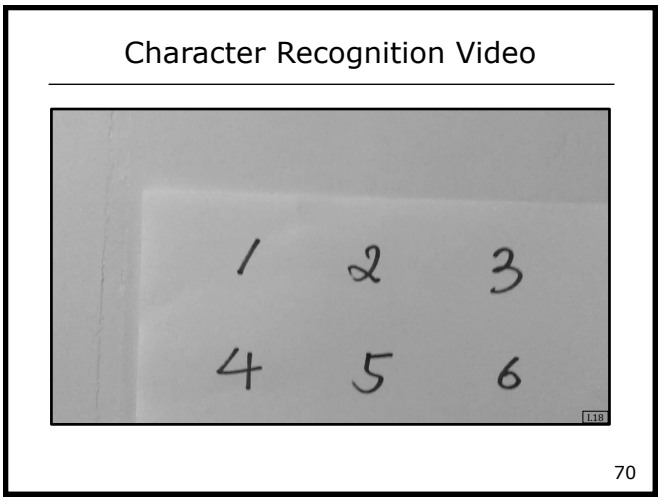




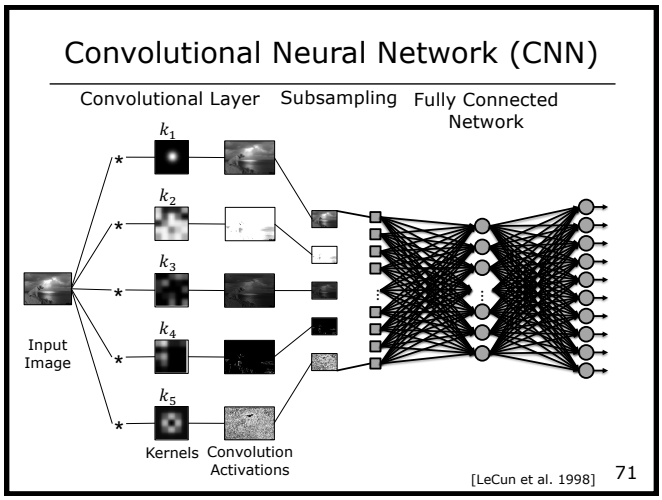


Today, neural networks are used to solve a wide range of computer vision problems. Let us look at a few examples. We'll start with the one we have been talking about, which is recognizing handwritten digits using Nielsen's network. On the top of slide 69 are a few inputs and on the bottom are the activations produced by the network. In the first example (the digit 7), we get a high activation for 7 and zero, or close to it, for all the others. This is true for the second example as well, where we get a high activation only for 2. In the third case, which we assume is a 5, we get high activations for both 2 and 5, and activation for 5 is larger than that for 2. The last one is more interesting. We don't really know what the input is in this case; it could be a 3 or an 8. In this case, we get large values for 2, 3, and 8, but 8 has the highest activation.

Here is a real-time implementation of the above digit recognition system running as an app on a phone. When the camera of the phone is pointed at a digit, the digit is immediately recognized. Please view the online lecture video to see the app in action.

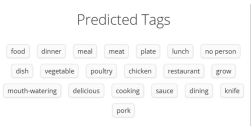


One of the major innovations in the context of neural networks is the idea of a convolutional neural network (CNN). It was proposed by Yann LeCun and has proved to be highly effective in wide range of vision applications. In vision, we often take an input image and first process it using various filters to extract information that is most relevant to the task. Often, the filters we use are linear filters which we know from our lecture on image processing can be implemented as convolutions. It turns out that the application of filters to an image can be incorporated into a neural network. In fact, during the training process, the network can learn the best filters to use for the task. Shown here is a CNN that has learned the kernels k_1 through k_5. During inference, these kernels are applied to the input image and the results and passed on to subsequent layers of the network.



In this example, a network developed by ClarifAI was training on millions of images to generate keywords (tags) that reveal what is in the image. In this example, the system produces several tags, including, “food,” “dinner,” and “chicken.” This type of a system can be used to automatically generate tags for images, making it easy to search for images with specific items that are of interest to the user.

Autotagging Photos



[Clarifai.com]

72

For this image, the ClarifAI network produces the tags “city skyline,” “skyscraper,” “office,” “harbor,” “water,” “waterfront,” etc.

Autotagging Photos

[Clarifai.com]

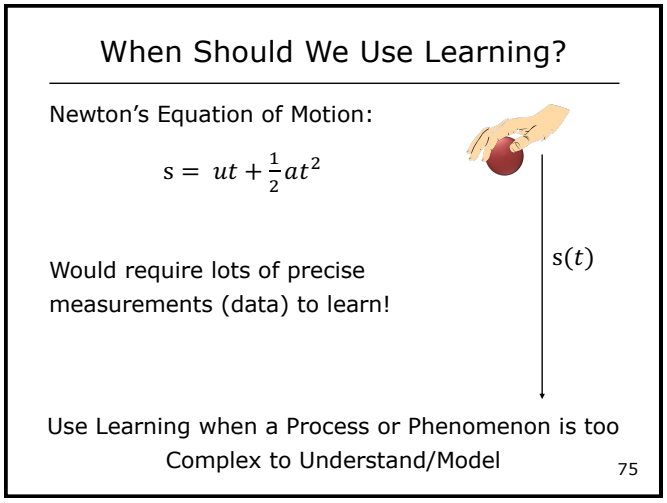
73

This convolutional neural network developed by Karpathy has been trained to recognize the activity in a video. It outputs three of the highest ranked activities (top-left corner), giving each one a confidence score. The three highest ranked activities for this video are “mountain unicycling,” “canyoning,” and “base jumping.” Please view the online lecture video to see how the system correctly recognizes other activities, such as kayaking, basketball, wheelchair basketball, mixed martial arts, etc.

Video Classification using CNN

[Karpathy 2014]

74



In this lecture series, we have focused on the first principles of computer vision. Since deep learning is extremely popular today, is it worth knowing the first principles of vision, or for that matter, the first principles of any field? Given a task, why not just train a neural network with tons of data to solve the task? Unfortunately, such an approach can prove to be both inadequate and unsatisfying. Given any mapping problem and enough data to train a network, it is likely we will very quickly get to a level of performance that is impressive, although not quite the performance we would like. Therein lies the problem. The last mile of getting a purely learning-based system to achieve the performance an application requires can prove to be a very long mile. It can also be a cumbersome one.

Consider dropping a ball from a height, as shown in the slide. Say we want to know the distance s the ball travels as a function of time t. Thanks to Newton and first principles, we know that $s = ut + \frac{1}{2}at^2$, where u is the initial velocity of the ball and a is the acceleration due to gravity. Today, we could use a machine to learn the relationship between s and t. We could gather a bunch of our friends and have them drop objects from various heights, and measure using a stopwatch when each object hits the ground. Then, we could train a network to learn the relationship between s and t. Imagine the process of collecting this data. Think about how much data we would need to get a result that is really close to Newton's equation. Perhaps most importantly, consider the fact that we have not gained any deep insights about the world we live in.

So, if you are a student who is keen on building a career in a field like computer vision, my advice would be to always think about a problem using first principles. Use this approach to make as much progress as you can. You will not only find it satisfying, but you may also gain knowledge and insights along the way that change the way you pose problems in the future. If you arrive at a stage where first principles cannot get you to an elegant solution, but first principles seem to hint that the problem probably can be posed as a complex mapping problem that has a solution, it would make complete sense for you to use your favorite machine learning algorithm to perform the mapping.

In short, first principles and machine learning can not only coexist, but, in fact, be symbiotic.

References and Credits

Shree K. Nayar

Columbia University

Topic: Neural Networks, Module: Perception

First Principles of Computer Vision

77

References: Papers

[Rosenblatt 1958] Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." *Psychological review* 65.6 (1958): 386.

[MNIST 1998] LeCun, Yann, Corinna Cortes, and Christopher JC Burges. "The MNIST database of handwritten digits." URL http://yann. lecun. com/exdb/mnist (1998).

[Williams et al. 1986] Williams, D. R. G. H. R., and G. E. Hinton. "Learning representations by back-propagating errors." *Nature* 323 (1986): 533-536.

[LeCun et al. 1998] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." *Proceedings of the IEEE* 86.11 (1998): 2278-2324.

[Karpathy 2014] Karpathy, Andrej, et al. "Large-scale video classification with convolutional neural networks." *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*. 2014.

78

References: Books

[Nielsen 2015] Michael A. Nielsen , "Neural Networks and Deep Learning", Determination Press, 2015

[Rumelhart et al. 1988] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." *Cognitive modeling* 5.3 (1988): 1.

79

Image Credits

I.1 http://www.dvidshub.net/image/1890918/faces-okinawa-jungle/ DoD Disclaimer: The appearance of U.S. Department of Defense (DoD) visual information does not imply or constitute DoD endorsement. Public Domain.

I.2 https://www.flickr.com/photos/tataimitra/9434857938/ Rajashri Mitra. Licensed under CC BY 2.0.

I.3 https://pixabay.com/en/faces-children-eyes-look-portrait-767358/ Public Domain.

I.4 https://www.flickr.com/photos/dan4th/3266978771 Dan4th Nicholas. Licensed under CC BY 2.0.

I.5 https://en.wikipedia.org/wiki/To_Autumn Public Domain.

I.6 https://www.flickr.com/photos/romitagirl67/15982674248 RomitaGirl67. Licensed under CC BY 2.0.

I.7 https://en.wikisource.org/wiki/Popular_Science_Monthly/Volume_44/November_1893/An_Argument_for_Vertical_Handwriting#/media/File:PSM_v44_D097_Results_of_improved_handwriting_after_four_weeks_of_practice.jpg/ Public Domain.

I.8 https://www.flickr.com/photos/peagreenchick/384744354 Peagreengirl. Licensed under CC BY 2.0.

I.9 https://www.flickr.com/photos/michiganmoves/3765498811 Debra Drummond. Licensed under CC BY-SA 2.0.

80

Image Credits

I.10 https://commons.wikimedia.org/wiki/File:Streit_Shakespeare_chair_side_view.jpg David R.J. Stiennon. Licensed under CC BY-SA 3.0.

I.11 https://commons.wikimedia.org/wiki/File:Aeron_chair_JN.jpg/ Public Domain.

I.12 https://pixabay.com/en/chair-butterfly-chair-sit-chairs-1355327/ Public Domain.

I.13 https://commons.wikimedia.org/wiki/File:Sitamun_chair_replica_1.jpg Alensha. Licensed under CC BY-SA 3.0.

I.14 https://www.geograph.org.uk/photo/2463536 William. Licensed under CC BY-SA 2.0.

I.15 https://commons.wikimedia.org/wiki/File:Moravsk%C3%A1_galerie_02_-_Thonet-Mundus_k%C5%99eslo.jpg Dominik Matus. Licensed under CC BY-SA 3.0.

I.16 https://commons.wikimedia.org/wiki/File:Art_deco_club_chair.jpg Didouner. Licensed under CC BY-SA 3.0.

I.18 https://www.youtube.com/watch?v=lkuucfUW36k Jimmy Ren. Used with permission.

I.19 https://clarifai.com

I.21 A. Karpathy.

81

Acknowledgements: Thanks to Pranav Sukumar, Joel Salzman, Tracy Cui and Jenna Everard for their help with transcription, editing and proofreading**.**

References

[Nielsen 2015] Michael A. Nielsen, "Neural Networks and Deep Learning", Determination Press, 2015.

[Rumelhart et al. 1988] Rumelhart, David E., Geoffrey E. Hinton, and Ronald J. Williams. "Learning representations by back-propagating errors." *Cognitive modeling* 5.3 (1988): 1.

[Rosenblatt 1958] Rosenblatt, Frank. "The perceptron: a probabilistic model for information storage and organization in the brain." Psychological review 65.6 (1958): 386.

[MNIST 1998] LeCun, Yann, Corinna Cortes, and Christopher JC Burges. "The MNIST database of handwritten digits." URL http://yann.lecun.com/exdb/mnist (1998).

[Williams et al. 1986] Williams, D. R. G. H. R., and G. E. Hinton. "Learning representations by back-propagating errors." Nature 323 (1986): 533-536.

[LeCun et al. 1998] LeCun, Yann, et al. "Gradient-based learning applied to document recognition." Proceedings of the IEEE 86.11 (1998): 2278-2324.

[Karpathy 2014] Karpathy, Andrej, et al. "Large-scale video classification with convolutional neural networks." Proceedings of the IEEE conference on Computer Vision and Pattern Recognition. 2014.

[Szeliski 2022] Computer Vision: Algorithms and Applications, Szeliski, R., Springer, 2022.

[Nayar 2022E] Image Processing I, Nayar, S. K., Monograph FPCV-1-4, First Principles of Computer Vision, Columbia University, New York, March 2022.

[Nayar 2022F] Image Processing II, Nayar, S. K., Monograph FPCV-1-5, First Principles of Computer Vision, Columbia University, New York, March 2022.

[Nayar 2025B] Face Detection, Nayar, S. K., Monograph FPCV-2-5, First Principles of Computer Vision, Columbia University, New York, February 2025.

[Nayar 2025L] Object Tracking, Nayar, S. K., Monograph FPCV-5-1, First Principles of Computer Vision, Columbia University, New York, May 2025.

[Nayar 2025M] Image Segmentation, Nayar, S. K., Monograph FPCV-5-2, First Principles of Computer Vision, Columbia University, New York, May 2025.

[Nayar 2025N] Appearance Matching, Nayar, S. K., Monograph FPCV-5-3, First Principles of Computer Vision, Columbia University, New York, May 2025.