Neural Networks

David Carlson

May 17, 2021

Perceptrons

- Artificial neuron

Perceptrons

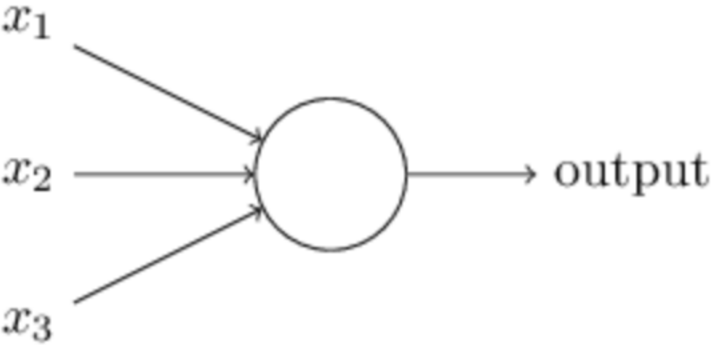
- Artificial neuron
- Developed in 1950's and 60's by Frank Rosenblatt

Perceptrons

- Artificial neuron
- Developed in 1950's and 60's by Frank Rosenblatt
- Perceptrons no longer common — main neuron model is sigmoid neurons — but perceptrons help motivate

Perceptrons

- Artificial neuron
- Developed in 1950's and 60's by Frank Rosenblatt
- Perceptrons no longer common — main neuron model is sigmoid neurons — but perceptrons help motivate
- Takes several binary inputs and produces a single binary output



Perceptron Weights

- In example, three inputs, x_1, x_2, x_3 (could be more or fewer)

Perceptron Weights

- In example, three inputs, x_1, x_2, x_3 (could be more or fewer)
- Also a weight for each input, w_1, w_2, w_3

Perceptron Weights

- In example, three inputs, x_1, x_2, x_3 (could be more or fewer)
- Also a weight for each input, w_1, w_2, w_3
- Real numbers expressing the importance of the respective inputs to the output

Perceptron Weights

- In example, three inputs, x_1, x_2, x_3 (could be more or fewer)
- Also a weight for each input, w_1, w_2, w_3
- Real numbers expressing the importance of the respective inputs to the output
- Neuron's output, 0 or 1, is determined by whether weighted sum $\sum_j w_j x_j$ is less than or greater than a threshold

Perceptron Weights

- In example, three inputs, x_1, x_2, x_3 (could be more or fewer)
- Also a weight for each input, w_1, w_2, w_3
- Real numbers expressing the importance of the respective inputs to the output
- Neuron's output, 0 or 1, is determined by whether weighted sum $\sum_j w_j x_j$ is less than or greater than a threshold
- Just like weights, threshold is a real number and a parameter of the neuron

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{threshold} \end{cases}$$

Perceptron Model

- Model decision-making

Perceptron Model

- Model decision-making
- Larger w for an input the more that factor matters to the ultimate decision

Perceptron Model

- Model decision-making
- Larger w for an input the more that factor matters to the ultimate decision
- By varying weights and threshold, can get different models of decision-making

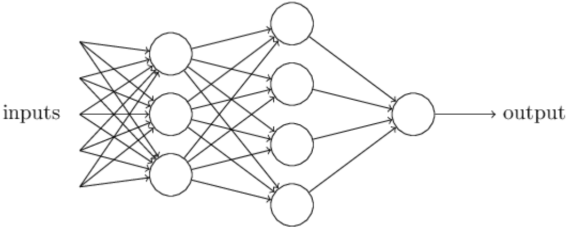
Perceptron Model

- Model decision-making
- Larger w for an input the more that factor matters to the ultimate decision
- By varying weights and threshold, can get different models of decision-making
- A perceptron can weigh up different kinds of evidence in order to make decisions

Perceptron Model

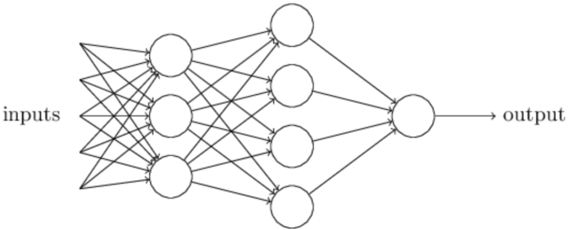
- Model decision-making
- Larger w for an input the more that factor matters to the ultimate decision
- By varying weights and threshold, can get different models of decision-making
- A perceptron can weigh up different kinds of evidence in order to make decisions
- Complex network of perceptrons could model subtle decisions

Perceptron Model (cont.)



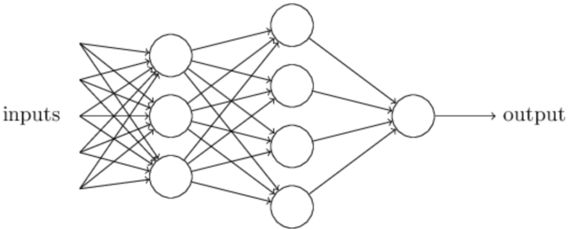
- First column of perceptrons — called the first layer of perceptrons — is making three simple decisions by weighing input evidence

Perceptron Model (cont.)



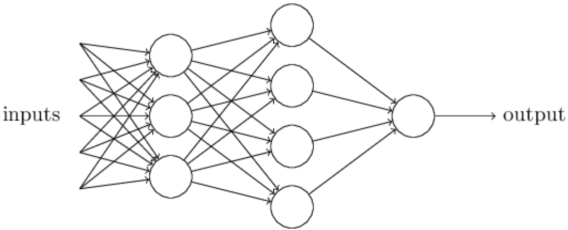
- First column of perceptrons — called the first layer of perceptrons — is making three simple decisions by weighing input evidence
- Second layer is making a decision by weighing results of first layer

Perceptron Model (cont.)



- First column of perceptrons — called the first layer of perceptrons — is making three simple decisions by weighing input evidence
- Second layer is making a decision by weighing results of first layer
- Perceptron in second level can make a decision at a more abstract and complex level

Perceptron Model (cont.)



- First column of perceptrons — called the first layer of perceptrons — is making three simple decisions by weighing input evidence
- Second layer is making a decision by weighing results of first layer
- Perceptron in second level can make a decision at a more abstract and complex level
- Many-layer network can engage in sophisticated decision-making

Notational Conveniences

- Two notational changes to simplify the above

Notational Conveniences

- Two notational changes to simplify the above
- Write as a dot product, $w \cdot x \equiv \sum_j w_j x_j$

Notational Conveniences

- Two notational changes to simplify the above
- Write as a dot product, $w \cdot x \equiv \sum_j w_j x_j$
- w and x are now vectors whose components are the weights and inputs

Notational Conveniences

- Two notational changes to simplify the above
- Write as a dot product, $w \cdot x \equiv \sum_j w_j x_j$
- w and x are now vectors whose components are the weights and inputs
- Move the threshold to the other side of the inequality, known as perceptron's bias $b \equiv -\text{threshold}$

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Notational Conveniences

- Two notational changes to simplify the above
- Write as a dot product, $w \cdot x \equiv \sum_j w_j x_j$
- w and x are now vectors whose components are the weights and inputs
- Move the threshold to the other side of the inequality, known as perceptron's bias $b \equiv -\text{threshold}$

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

- Bias as measure of how easy it is to get the perceptron to output a 1, or to fire

Notational Conveniences

- Two notational changes to simplify the above
- Write as a dot product, $w \cdot x \equiv \sum_j w_j x_j$
- w and x are now vectors whose components are the weights and inputs
- Move the threshold to the other side of the inequality, known as perceptron's bias $b \equiv -\text{threshold}$

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

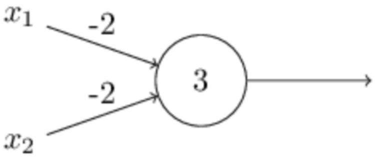
- Bias as measure of how easy it is to get the perceptron to output a 1, or to fire
- Really big bias $\rightarrow$ easy to get a 1

Elementary Logical Functions

- Perceptrons can also be used to compute the elementary logical functions such as AND and OR

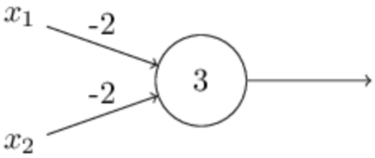
Elementary Logical Functions

- Perceptrons can also be used to compute the elementary logical functions such as AND and OR
- Suppose we have perceptron with two inputs, each with weight -2 and a bias of 3:



Elementary Logical Functions

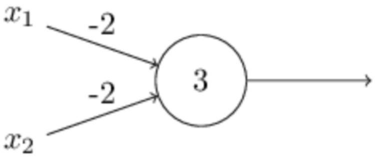
- Perceptrons can also be used to compute the elementary logical functions such as AND and OR
- Suppose we have perceptron with two inputs, each with weight -2 and a bias of 3:



- Input 00 produces a 1, since $-2 \times 0 - 2 \times 0 + 3 > 0$

Elementary Logical Functions

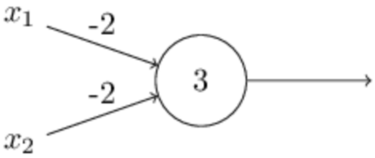
- Perceptrons can also be used to compute the elementary logical functions such as AND and OR
- Suppose we have perceptron with two inputs, each with weight -2 and a bias of 3:



- Input 00 produces a 1, since $-2 \times 0 - 2 \times 0 + 3 > 0$
- Input 01 and 10 produce a 1, but 11 produces a 0

Elementary Logical Functions

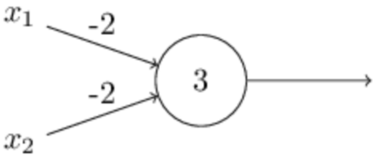
- Perceptrons can also be used to compute the elementary logical functions such as AND and OR
- Suppose we have perceptron with two inputs, each with weight -2 and a bias of 3:



- Input 00 produces a 1, since $-2 \times 0 - 2 \times 0 + 3 > 0$
- Input 01 and 10 produce a 1, but 11 produces a 0
- We have produced a NAND gate

Elementary Logical Functions

- Perceptrons can also be used to compute the elementary logical functions such as AND and OR
- Suppose we have perceptron with two inputs, each with weight -2 and a bias of 3:



- Input 00 produces a 1, since $-2 \times 0 - 2 \times 0 + 3 > 0$
- Input 01 and 10 produce a 1, but 11 produces a 0
- We have produced a NAND gate
- We can build any computation up from NAND gates

Learning

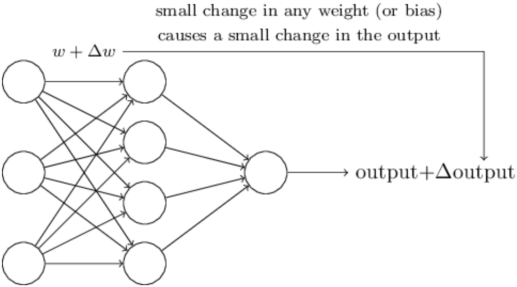
- Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem (e.g., classifying digits)

Learning

- Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem (e.g., classifying digits)
- To see how learning might work, suppose we make a small change in some weight (or bias) in the network

Learning

- Suppose we have a network of perceptrons that we'd like to use to learn to solve some problem (e.g., classifying digits)
- To see how learning might work, suppose we make a small change in some weight (or bias) in the network
- What we'd like is for this small change in weight to cause only a small corresponding change in the output from the network



Learning (cont.)

- If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want

Learning (cont.)

- If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want
- And then we'd repeat this, changing the weights and biases over and over to produce better and better output $\rightarrow$ learning

Learning (cont.)

- If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want
- And then we'd repeat this, changing the weights and biases over and over to produce better and better output $\rightarrow$ learning
- The problem is that this is not what happens in network of perceptrons

Learning (cont.)

- If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want
- And then we'd repeat this, changing the weights and biases over and over to produce better and better output $\rightarrow$ learning
- The problem is that this is not what happens in network of perceptrons
- A small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip

Learning (cont.)

- If it were true that a small change in a weight (or bias) causes only a small change in output, then we could use this fact to modify the weights and biases to get our network to behave more in the manner we want
- And then we'd repeat this, changing the weights and biases over and over to produce better and better output $\rightarrow$ learning
- The problem is that this is not what happens in network of perceptrons
- A small change in the weights or bias of any single perceptron in the network can sometimes cause the output of that perceptron to completely flip
- That flip may then cause the behaviour of the rest of the network to completely change in some very complicated way

Sigmoid Neuron

- Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output

Sigmoid Neuron

- Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output
- Just like a perceptron, the sigmoid neuron has inputs, $x_1, x_2, \ldots$

Sigmoid Neuron

- Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output
- Just like a perceptron, the sigmoid neuron has inputs, $x_1, x_2, \ldots$
- But instead of being just 0 or 1, these inputs can also take on any values between 0 and 1

Sigmoid Neuron

- Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output
- Just like a perceptron, the sigmoid neuron has inputs, $x_1, x_2, \ldots$
- But instead of being just 0 or 1, these inputs can also take on any values between 0 and 1
- The output is no longer 0 or 1, it is $\sigma(w \cdot x + b)$, where σ is the sigmoid function (also called logistic function and logistic neurons)

$$\begin{aligned}\sigma(z) &= \frac{1}{1+e^{-z}} \\ &= \frac{1}{1+\exp(-\sum_j w_j x_j - b)}\end{aligned}$$

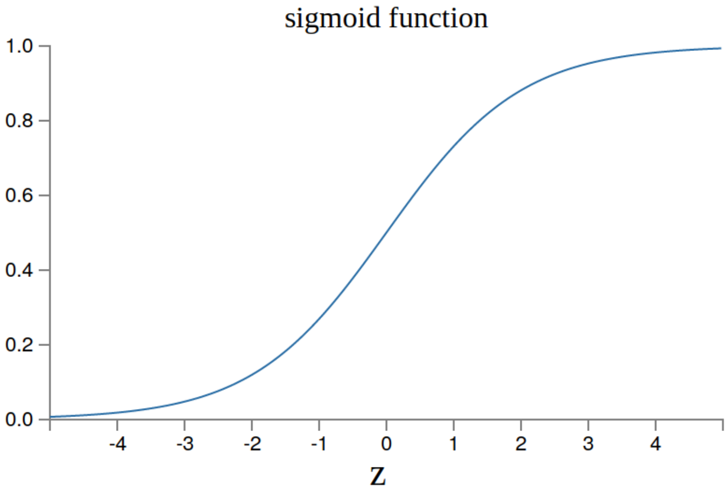
Sigmoid Neuron

- Sigmoid neurons are similar to perceptrons, but modified so that small changes in their weights and bias cause only a small change in their output
- Just like a perceptron, the sigmoid neuron has inputs, $x_1, x_2, \dots$
- But instead of being just 0 or 1, these inputs can also take on any values between 0 and 1
- The output is no longer 0 or 1, it is $\sigma(w \cdot x + b)$, where σ is the sigmoid function (also called logistic function and logistic neurons)

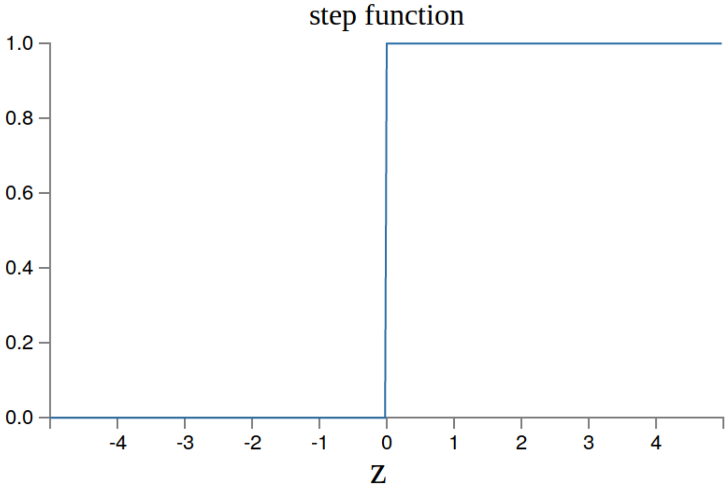
$$\begin{aligned} \sigma(z) &= \frac{1}{1 + e^{-z}} \\ &= \frac{1}{1 + \exp(-\sum_j w_j x_j - b)} \end{aligned}$$

- When very negative, close to 0, when very positive, close to 1

Sigmoid Shape



Perceptron Shape



Smoothness of the Sigmoid

- By using the actual sigmoid function we get a smoothed out perceptron

Smoothness of the Sigmoid

- By using the actual sigmoid function we get a smoothed out perceptron
- It is the smoothness of the sigmoid function that is crucial, not its exact form

Smoothness of the Sigmoid

- By using the actual sigmoid function we get a smoothed out perceptron
- It is the smoothness of the sigmoid function that is crucial, not its exact form
- The smoothness means that small changes in the weights and bias produce a small change in the output

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b$$

Smoothness of the Sigmoid

- By using the actual sigmoid function we get a smoothed out perceptron
- It is the smoothness of the sigmoid function that is crucial, not its exact form
- The smoothness means that small changes in the weights and bias produce a small change in the output

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b$$

- In words, Δoutput is a linear function of the changes to weights and biases

Smoothness of the Sigmoid

- By using the actual sigmoid function we get a smoothed out perceptron
- It is the smoothness of the sigmoid function that is crucial, not its exact form
- The smoothness means that small changes in the weights and bias produce a small change in the output

$$\Delta \text{output} \approx \sum_j \frac{\partial \text{output}}{\partial w_j} \Delta w_j + \frac{\partial \text{output}}{\partial b} \Delta b$$

- In words, Δoutput is a linear function of the changes to weights and biases
- This linearity makes it easy to choose small changes in the weights and biases to achieve any desired small change in the output

Interpreting Output

- One big difference between perceptrons and sigmoid neurons is that sigmoid neurons don't just output 0 or 1 but any real value between 0 and 1

Interpreting Output

- One big difference between perceptrons and sigmoid neurons is that sigmoid neurons don't just output 0 or 1 but any real value between 0 and 1
- This can be useful, e.g., if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network

Interpreting Output

- One big difference between perceptrons and sigmoid neurons is that sigmoid neurons don't just output 0 or 1 but any real value between 0 and 1
- This can be useful, e.g., if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network
- Sometimes it can be a nuisance, e.g. the input image is a 9 or not a 9

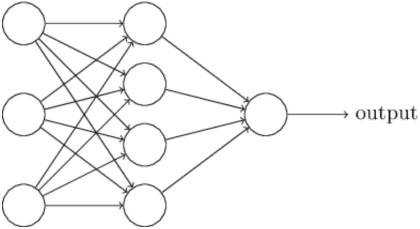
Interpreting Output

- One big difference between perceptrons and sigmoid neurons is that sigmoid neurons don't just output 0 or 1 but any real value between 0 and 1
- This can be useful, e.g., if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network
- Sometimes it can be a nuisance, e.g. the input image is a 9 or not a 9
- In practice set up a convention, e.g. if greater than 0.5 indicates a 9, otherwise does not

Interpreting Output

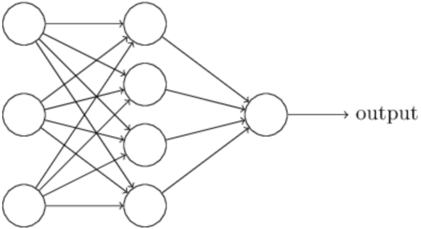
- One big difference between perceptrons and sigmoid neurons is that sigmoid neurons don't just output 0 or 1 but any real value between 0 and 1
- This can be useful, e.g., if we want to use the output value to represent the average intensity of the pixels in an image input to a neural network
- Sometimes it can be a nuisance, e.g. the input image is a 9 or not a 9
- In practice set up a convention, e.g. if greater than 0.5 indicates a 9, otherwise does not
- Be explicit about convention used

The Architecture of Neural Networks



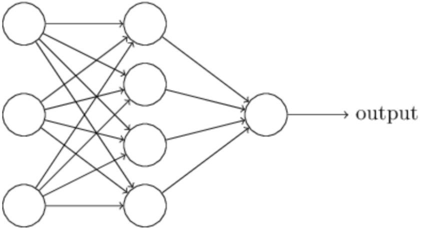
- Leftmost layer is the input layer, with input neurons

The Architecture of Neural Networks



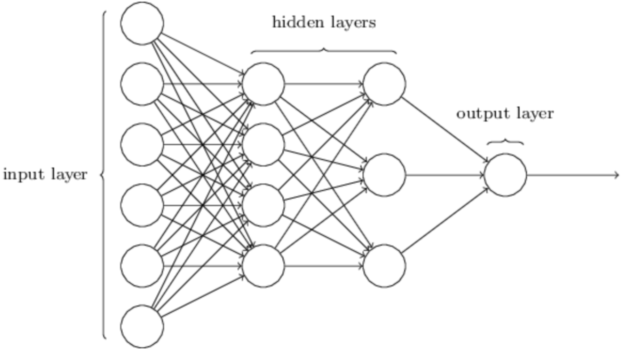
- Leftmost layer is the input layer, with input neurons
- Rightmost is output layer containing output neurons

The Architecture of Neural Networks



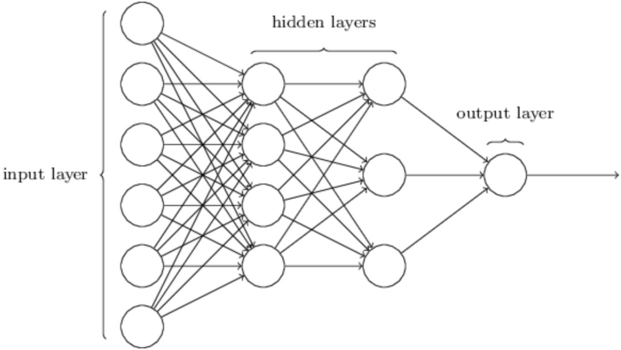
- Leftmost layer is the input layer, with input neurons
- Rightmost is output layer containing output neurons
- Middle layer is hidden layer, neither inputs nor outputs

The Architecture of Neural Networks



- Some networks have multiple hidden layers

The Architecture of Neural Networks



- Some networks have multiple hidden layers
- Confusingly, sometimes multiple layer networks are called multilayer perceptrons or MLPs

Network Design

- The design of the input and output layers in a network is often straightforward

Network Design

- The design of the input and output layers in a network is often straightforward
- For example, suppose we're trying to determine whether a handwritten image depicts a 9 or not

Network Design

- The design of the input and output layers in a network is often straightforward
- For example, suppose we're trying to determine whether a handwritten image depicts a 9 or not
- A natural way to design the network is to encode the intensities of the image pixels into the input neurons

Network Design

- The design of the input and output layers in a network is often straightforward
- For example, suppose we're trying to determine whether a handwritten image depicts a 9 or not
- A natural way to design the network is to encode the intensities of the image pixels into the input neurons
- If the image is a 64 by 64 greyscale image, then we'd have $4,096 = 64 \times 64$ input neurons, with the intensities scaled appropriately between 0 and 1

Network Design

- The design of the input and output layers in a network is often straightforward
- For example, suppose we're trying to determine whether a handwritten image depicts a 9 or not
- A natural way to design the network is to encode the intensities of the image pixels into the input neurons
- If the image is a 64 by 64 greyscale image, then we'd have $4,096 = 64 \times 64$ input neurons, with the intensities scaled appropriately between 0 and 1
- The output layer will contain just a single neuron, with output values of less than 0.5 indicating input image is not a 9, and values greater than 0.5 indicating input image is a 9

Network Design (cont.)

- While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers

Network Design (cont.)

- While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers
- In particular, it's not possible to sum up the design process for the hidden layers with a few simple rules of thumb

Network Design (cont.)

- While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers
- In particular, it's not possible to sum up the design process for the hidden layers with a few simple rules of thumb
- Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behaviour they want out of their nets

Network Design (cont.)

- While the design of the input and output layers of a neural network is often straightforward, there can be quite an art to the design of the hidden layers
- In particular, it's not possible to sum up the design process for the hidden layers with a few simple rules of thumb
- Instead, neural networks researchers have developed many design heuristics for the hidden layers, which help people get the behaviour they want out of their nets
- For example, such heuristics can be used to help determine how to trade off the number of hidden layers against the time required to train the network.

Feedforward Neural Networks

- Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer

Feedforward Neural Networks

- Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer
- Such networks are called feedforward neural networks

Feedforward Neural Networks

- Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer
- Such networks are called feedforward neural networks
- This means there are no loops in the network — information is always fed forward, never fed back

Feedforward Neural Networks

- Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer
- Such networks are called feedforward neural networks
- This means there are no loops in the network — information is always fed forward, never fed back
- If we did have loops, we'd end up with situations where the input to the sigmoid function depended on the output

Feedforward Neural Networks

- Up to now, we've been discussing neural networks where the output from one layer is used as input to the next layer
- Such networks are called feedforward neural networks
- This means there are no loops in the network — information is always fed forward, never fed back
- If we did have loops, we'd end up with situations where the input to the sigmoid function depended on the output
- That'd be hard to make sense of, and so we don't allow such loops

Recurrent Neural Networks

- There are other models of artificial neural networks in which feedback loops are possible

Recurrent Neural Networks

- There are other models of artificial neural networks in which feedback loops are possible
- These models are called recurrent neural networks

Recurrent Neural Networks

- There are other models of artificial neural networks in which feedback loops are possible
- These models are called recurrent neural networks
- The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent

Recurrent Neural Networks

- There are other models of artificial neural networks in which feedback loops are possible
- These models are called recurrent neural networks
- The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent
- That firing can stimulate other neurons, which may fire a little while later, also for a limited duration

Recurrent Neural Networks

- There are other models of artificial neural networks in which feedback loops are possible
- These models are called recurrent neural networks
- The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent
- That firing can stimulate other neurons, which may fire a little while later, also for a limited duration
- That causes still more neurons to fire, and so over time we get a cascade of neurons firing

Recurrent Neural Networks

- There are other models of artificial neural networks in which feedback loops are possible
- These models are called recurrent neural networks
- The idea in these models is to have neurons which fire for some limited duration of time, before becoming quiescent
- That firing can stimulate other neurons, which may fire a little while later, also for a limited duration
- That causes still more neurons to fire, and so over time we get a cascade of neurons firing
- Loops don't cause problems in such a model, since a neuron's output only affects its input at some later time, not instantaneously

A Simple Network to Classify Handwritten Digits

- We can split the problem of recognizing handwritten digits into two sub-problems

A Simple Network to Classify Handwritten Digits

- We can split the problem of recognizing handwritten digits into two sub-problems
- First, we'd like a way of breaking an image containing many digits into a sequence of separate images, each containing a single digit

A Simple Network to Classify Handwritten Digits

- We can split the problem of recognizing handwritten digits into two sub-problems
- First, we'd like a way of breaking an image containing many digits into a sequence of separate images, each containing a single digit
- For example, we'd like to break the image into six separate images

A Simple Network to Classify Handwritten Digits

- We can split the problem of recognizing handwritten digits into two sub-problems
- First, we'd like a way of breaking an image containing many digits into a sequence of separate images, each containing a single digit
- For example, we'd like to break the image into six separate images

- We humans solve this segmentation problem with ease, but it's challenging for a computer program to correctly break up the image

A Simple Network to Classify Handwritten Digits

- We can split the problem of recognizing handwritten digits into two sub-problems
- First, we'd like a way of breaking an image containing many digits into a sequence of separate images, each containing a single digit
- For example, we'd like to break the image into six separate images

- We humans solve this segmentation problem with ease, but it's challenging for a computer program to correctly break up the image
- Once the image has been segmented, the program then needs to classify each individual digit

The Segmentation Problem

- We'll focus on writing a program to solve the second problem, that is, classifying individual digits

The Segmentation Problem

- We'll focus on writing a program to solve the second problem, that is, classifying individual digits
- We do this because it turns out that the segmentation problem is not so difficult to solve, once you have a good way of classifying individual digits

The Segmentation Problem

- We'll focus on writing a program to solve the second problem, that is, classifying individual digits
- We do this because it turns out that the segmentation problem is not so difficult to solve, once you have a good way of classifying individual digits
- There are many approaches to solving the segmentation problem

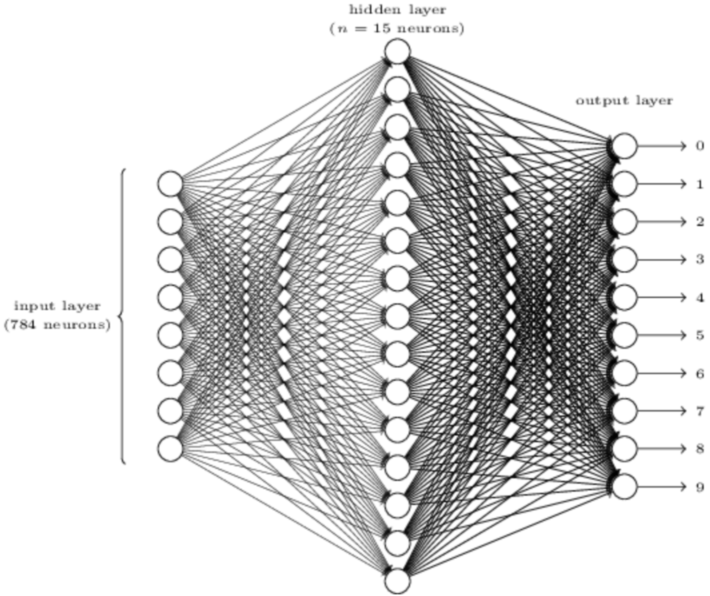
The Segmentation Problem

- We'll focus on writing a program to solve the second problem, that is, classifying individual digits
- We do this because it turns out that the segmentation problem is not so difficult to solve, once you have a good way of classifying individual digits
- There are many approaches to solving the segmentation problem
- So instead of worrying about segmentation we'll concentrate on developing a neural network which can solve the more interesting and difficult problem, namely, recognizing individual handwritten digits

The Segmentation Problem

- We'll focus on writing a program to solve the second problem, that is, classifying individual digits
- We do this because it turns out that the segmentation problem is not so difficult to solve, once you have a good way of classifying individual digits
- There are many approaches to solving the segmentation problem
- So instead of worrying about segmentation we'll concentrate on developing a neural network which can solve the more interesting and difficult problem, namely, recognizing individual handwritten digits
- To recognize individual digits we will use a three-layer neural network

Three-Layer Neural Network



The Input Layer

- The input layer of the network contains neurons encoding the values of the input pixels

The Input Layer

- The input layer of the network contains neurons encoding the values of the input pixels
- Our training data for the network will consist of many 28 by 28 pixel images of scanned handwritten digits, and so the input layer contains $784 = 28 \times 28$ neurons

The Input Layer

- The input layer of the network contains neurons encoding the values of the input pixels
- Our training data for the network will consist of many 28 by 28 pixel images of scanned handwritten digits, and so the input layer contains $784 = 28 \times 28$ neurons
- For simplicity I've omitted most of the 784 input neurons in the diagram above

The Input Layer

- The input layer of the network contains neurons encoding the values of the input pixels
- Our training data for the network will consist of many 28 by 28 pixel images of scanned handwritten digits, and so the input layer contains $784 = 28 \times 28$ neurons
- For simplicity I've omitted most of the 784 input neurons in the diagram above
- The input pixels are greyscale, with a value of 0.0 representing white, a value of 1.0 representing black, and in between values representing gradually darkening shades of grey

The Hidden Layer

- The second layer of the network is a hidden layer

The Hidden Layer

- The second layer of the network is a hidden layer
- We denote the number of neurons in this hidden layer by n, and we'll experiment with different values for n

The Hidden Layer

- The second layer of the network is a hidden layer
- We denote the number of neurons in this hidden layer by n, and we'll experiment with different values for n
- The example shown illustrates a small hidden layer, containing just $n = 15$ neurons

The Output Layer

- The output layer of the network contains 10 neurons

The Output Layer

- The output layer of the network contains 10 neurons
- If the first neuron fires, i.e., has an output ≈ 1, then that will indicate that the network thinks the digit is a 0

The Output Layer

- The output layer of the network contains 10 neurons
- If the first neuron fires, i.e., has an output ≈ 1, then that will indicate that the network thinks the digit is a 0
- If the second neuron fires then that will indicate that the network thinks the digit is a 1, and so on

The Output Layer

- The output layer of the network contains 10 neurons
- If the first neuron fires, i.e., has an output ≈ 1, then that will indicate that the network thinks the digit is a 0
- If the second neuron fires then that will indicate that the network thinks the digit is a 1, and so on
- A little more precisely, we number the output neurons from 0 through 9, and figure out which neuron has the highest activation value

The Output Layer

- The output layer of the network contains 10 neurons
- If the first neuron fires, i.e., has an output ≈ 1, then that will indicate that the network thinks the digit is a 0
- If the second neuron fires then that will indicate that the network thinks the digit is a 1, and so on
- A little more precisely, we number the output neurons from 0 through 9, and figure out which neuron has the highest activation value
- If that neuron is, say, neuron number 6, then our network will guess that the input digit was a 6

Learning with Gradient Descent

- What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x

Learning with Gradient Descent

- What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x
- To quantify how well we're achieving this goal we define a cost function:

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2$$

Learning with Gradient Descent

- What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x
- To quantify how well we're achieving this goal we define a cost function:

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2$$

- Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x

Learning with Gradient Descent

- What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x
- To quantify how well we're achieving this goal we define a cost function:

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2$$

- Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x
- Quadratic loss function, also known as mean squared error or MSE

Learning with Gradient Descent

- What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x
- To quantify how well we're achieving this goal we define a cost function:

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2$$

- Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x
- Quadratic loss function, also known as mean squared error or MSE
- Strictly non-negative

Learning with Gradient Descent

- What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x
- To quantify how well we're achieving this goal we define a cost function:

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2$$

- Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x
- Quadratic loss function, also known as mean squared error or MSE
- Strictly non-negative
- Small when $y(x)$ is approximately equal to output a for all training inputs x

Learning with Gradient Descent

- What we'd like is an algorithm which lets us find weights and biases so that the output from the network approximates $y(x)$ for all training inputs x
- To quantify how well we're achieving this goal we define a cost function:

$$C(w, b) = \frac{1}{2n} \sum_x ||y(x) - a||^2$$

- Here, w denotes the collection of all weights in the network, b all the biases, n is the total number of training inputs, a is the vector of outputs from the network when x is input, and the sum is over all training inputs, x
- Quadratic loss function, also known as mean squared error or MSE
- Strictly non-negative
- Small when $y(x)$ is approximately equal to output a for all training inputs x
- Minimize cost $C(w, b)$ as function of weights and biases $\rightarrow$ gradient descent

Quadratic Cost

- Aren't we primarily interested in the number of images correctly classified by the network?

Quadratic Cost

- Aren't we primarily interested in the number of images correctly classified by the network?
- The problem with that is that the number of images correctly classified is not a smooth function of the weights and biases in the network

Quadratic Cost

- Aren't we primarily interested in the number of images correctly classified by the network?
- The problem with that is that the number of images correctly classified is not a smooth function of the weights and biases in the network
- For the most part, making small changes to the weights and biases won't cause any change at all in the number of training images classified correctly

Quadratic Cost

- Aren't we primarily interested in the number of images correctly classified by the network?
- The problem with that is that the number of images correctly classified is not a smooth function of the weights and biases in the network
- For the most part, making small changes to the weights and biases won't cause any change at all in the number of training images classified correctly
- That makes it difficult to figure out how to change the weights and biases to get improved performance

Quadratic Cost

- Aren't we primarily interested in the number of images correctly classified by the network?
- The problem with that is that the number of images correctly classified is not a smooth function of the weights and biases in the network
- For the most part, making small changes to the weights and biases won't cause any change at all in the number of training images classified correctly
- That makes it difficult to figure out how to change the weights and biases to get improved performance
- If we instead use a smooth cost function like the quadratic cost it turns out to be easy to figure out how to make small changes in the weights and biases so as to get an improvement in the cost

Quadratic Cost

- Aren't we primarily interested in the number of images correctly classified by the network?
- The problem with that is that the number of images correctly classified is not a smooth function of the weights and biases in the network
- For the most part, making small changes to the weights and biases won't cause any change at all in the number of training images classified correctly
- That makes it difficult to figure out how to change the weights and biases to get improved performance
- If we instead use a smooth cost function like the quadratic cost it turns out to be easy to figure out how to make small changes in the weights and biases so as to get an improvement in the cost
- That's why we focus first on minimizing the quadratic cost, and only after that will we examine the classification accuracy.

Gradient Descent

- Let's suppose we're trying to minimize some function, $C(v)$

Gradient Descent

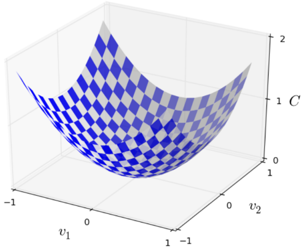
- Let's suppose we're trying to minimize some function, $C(v)$
- This could be any real-valued function of many variables, $\mathbf{v} = v_1, v_2, \ldots$

Gradient Descent

- Let's suppose we're trying to minimize some function, $C(v)$
- This could be any real-valued function of many variables, $\mathbf{v} = v_1, v_2, \ldots$
- Note that I've replaced the w and b notation by v to emphasize that this could be any function — we're not specifically thinking in the neural networks context any more

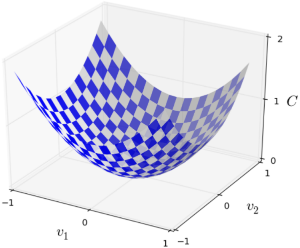
Gradient Descent

- Let's suppose we're trying to minimize some function, $C(v)$
- This could be any real-valued function of many variables, $\mathbf{v} = v_1, v_2, \ldots$
- Note that I've replaced the w and b notation by v to emphasize that this could be any function — we're not specifically thinking in the neural networks context any more
- To minimize $C(v)$ it helps to imagine C as a function of just two variables, which we'll call v_1 and v_2



Gradient Descent

- Let's suppose we're trying to minimize some function, $C(v)$
- This could be any real-valued function of many variables, $\mathbf{v} = v_1, v_2, \ldots$
- Note that I've replaced the w and b notation by v to emphasize that this could be any function — we're not specifically thinking in the neural networks context any more
- To minimize $C(v)$ it helps to imagine C as a function of just two variables, which we'll call v_1 and v_2



- We'd like to find where C achieves its global minimum

Gradient Descent (cont.)

- A general function, C, may be a complicated function of many variables, and it won't usually be possible to just eyeball the graph to find the minimum

Gradient Descent (cont.)

- A general function, C, may be a complicated function of many variables, and it won't usually be possible to just eyeball the graph to find the minimum
- Using calculus to analytically solve the minimum will not work with several (often billions of) variables

Gradient Descent (cont.)

- A general function, C, may be a complicated function of many variables, and it won't usually be possible to just eyeball the graph to find the minimum
- Using calculus to analytically solve the minimum will not work with several (often billions of) variables
- Randomly choose a starting point for an (imaginary) ball, and then simulate the motion of the ball as it rolled down to the bottom of the valley

Gradient Descent (cont.)

- A general function, C, may be a complicated function of many variables, and it won't usually be possible to just eyeball the graph to find the minimum
- Using calculus to analytically solve the minimum will not work with several (often billions of) variables
- Randomly choose a starting point for an (imaginary) ball, and then simulate the motion of the ball as it rolled down to the bottom of the valley
- We could do this simulation simply by computing derivatives (and perhaps some second derivatives) of C — those derivatives would tell us everything we need to know about the local shape of the valley, and therefore how our ball should roll

Gradient Descent (cont.)

- To make this question more precise, let's think about what happens when we move the ball a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction

Gradient Descent (cont.)

- To make this question more precise, let's think about what happens when we move the ball a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction
- Calculus tells us that C changes as follows

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

Gradient Descent (cont.)

- To make this question more precise, let's think about what happens when we move the ball a small amount Δv_1 in the v_1 direction, and a small amount Δv_2 in the v_2 direction
- Calculus tells us that C changes as follows

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2$$

- We're going to find a way of choosing Δv_1 and Δv_2 so as to make ΔC negative; i.e., we'll choose them so the ball is rolling down into the valley

Mathematical Notation

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2}\right)^T$$

$$\Delta C \approx \nabla C \cdot \Delta v$$

$$\Delta v = -\eta \nabla C, \eta > 0$$

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla C||^2$$

$$v \rightarrow v' = v - \eta \nabla C$$

Learning Rate

- To make gradient descent work correctly, we need to choose the learning rate η to be small enough that the above equation is a good approximation

Learning Rate

- To make gradient descent work correctly, we need to choose the learning rate η to be small enough that the above equation is a good approximation
- If we don't, we might end up with $\Delta C > 0$

Learning Rate

- To make gradient descent work correctly, we need to choose the learning rate η to be small enough that the above equation is a good approximation
- If we don't, we might end up with $\Delta C > 0$
- At the same time, we don't want η to be too small, since that will make the changes Δv tiny, and thus the gradient descent algorithm will work very slowly

Learning Rate

- To make gradient descent work correctly, we need to choose the learning rate η to be small enough that the above equation is a good approximation
- If we don't, we might end up with $\Delta C > 0$
- At the same time, we don't want η to be too small, since that will make the changes Δv tiny, and thus the gradient descent algorithm will work very slowly
- In practical implementations, η is often varied so that the equation remains a good approximation, but the algorithm isn't too slow

Learning Rate

- To make gradient descent work correctly, we need to choose the learning rate η to be small enough that the above equation is a good approximation
- If we don't, we might end up with $\Delta C > 0$
- At the same time, we don't want η to be too small, since that will make the changes Δv tiny, and thus the gradient descent algorithm will work very slowly
- In practical implementations, η is often varied so that the equation remains a good approximation, but the algorithm isn't too slow
- We'll see later how this works

Stochastic Gradient Descent

- There are a number of challenges in applying gradient descent, but for our purposes understanding one is critical

Stochastic Gradient Descent

- There are a number of challenges in applying gradient descent, but for our purposes understanding one is critical
- The cost function is an average over costs for individual training examples

Stochastic Gradient Descent

- There are a number of challenges in applying gradient descent, but for our purposes understanding one is critical
- The cost function is an average over costs for individual training examples
- In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then average them

Stochastic Gradient Descent

- There are a number of challenges in applying gradient descent, but for our purposes understanding one is critical
- The cost function is an average over costs for individual training examples
- In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then average them
- When number of training points is very large this can take a long time

Stochastic Gradient Descent

- There are a number of challenges in applying gradient descent, but for our purposes understanding one is critical
- The cost function is an average over costs for individual training examples
- In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then average them
- When number of training points is very large this can take a long time
- Stochastic gradient descent: estimate the gradient by computing it for a small sample of randomly chosen training inputs

Stochastic Gradient Descent

- There are a number of challenges in applying gradient descent, but for our purposes understanding one is critical
- The cost function is an average over costs for individual training examples
- In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then average them
- When number of training points is very large this can take a long time
- Stochastic gradient descent: estimate the gradient by computing it for a small sample of randomly chosen training inputs
- Averaging over small sample produces a good estimate of the true gradient and speeds up learning

Stochastic Gradient Descent

- There are a number of challenges in applying gradient descent, but for our purposes understanding one is critical
- The cost function is an average over costs for individual training examples
- In practice, to compute the gradient ∇C we need to compute the gradients ∇C_x separately for each training input, x, and then average them
- When number of training points is very large this can take a long time
- Stochastic gradient descent: estimate the gradient by computing it for a small sample of randomly chosen training inputs
- Averaging over small sample produces a good estimate of the true gradient and speeds up learning
- Of course, the estimate won't be perfect — there will be statistical fluctuations — but it doesn't need to be perfect: all we really care about is moving in a general direction that will help decrease C, and that means we don't need an exact computation of the gradient