NYC DATA SCIENCE ACADEMY

Neural Networks

Data Science Bootcamp

Outline

- Part 1: Neural Networks & Perceptrons
- Part 2: Sigmoid Neurons
- Part 3: Network Topology & Hidden Features
- Part 4: Backpropagation Learning with Gradient Descent
- Part 5: Additional Remarks on Neural Networks
- Part 6: Review

PART 1

Neural Networks & Perceptrons

Neural Networks

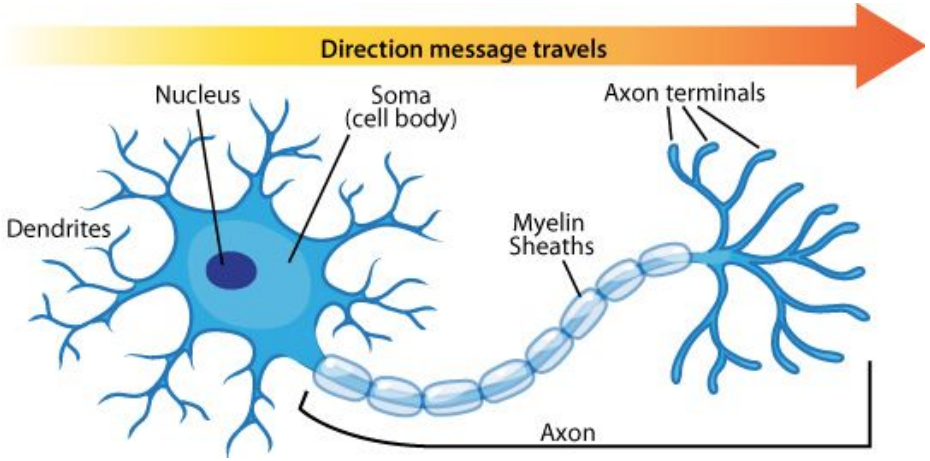
- A neural network is a supervised methodology that models the relationship between a set of input signals and an output signal in order to perform either classification or regression.

- Neural networks differ from other machine learning algorithms because of their sheer complexity; the underlying models are based on composite mathematical systems that often render accurate results, but are near-impossible to interpret.

- Models that transform a series of inputs into an output in an obfuscated manner are often referred to as black box processes because the mechanism seems to be hidden from view.

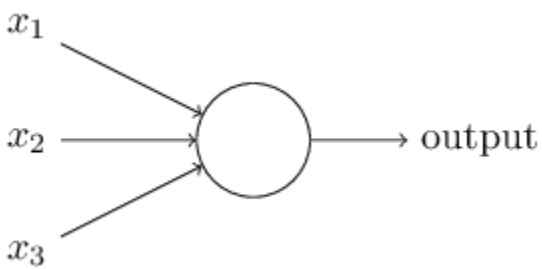
The Biological Model of Neurons

- The underlying model behind artificial neural networks was derived from our understanding of how a biological brain responds to stimuli from sensory inputs:
  - A human brain uses a complicated network of interconnected cells called neurons in order to parallel-process input.
  - Signals are received by the neuron's dendrites through a biochemical process that weights impulses based on their relative importance.
  - Should a threshold be reached by accumulating impulses, the neuron is said to fire; its impulse is then passed to neighboring neurons.



The Artificial Model of Neurons: The Perceptron

- The model of an artificial neuron is very closely related to the biological model. Often, a single artificial neuron is depicted using a directed network diagram that helps connect the inputs to the output:



- The most basic type of artificial neuron is called a perceptron.
  - A perceptron takes in various binary inputs $x_1, x_2, \ldots, x_n$, and produces a single binary output.
  - Each input has a corresponding weight $w_1, w_2, \ldots, w_n$ that expresses the importance of the input in determining the output.
  - But how is the output determined?

The Artificial Model of Neurons: The Perceptron

- For a perceptron, the output is also binary (i.e., it is either 0 or 1); determining whether the output should be 0 or 1 is relatively simple:
  - Each input variable x_i is considered alongside its corresponding weight w_i.
  - The sum of the various input/weight combinations is calculated.
  - Should the grand sum be greater than a certain threshold, then the perceptron returns a 1; otherwise, the perceptron returns a 0.

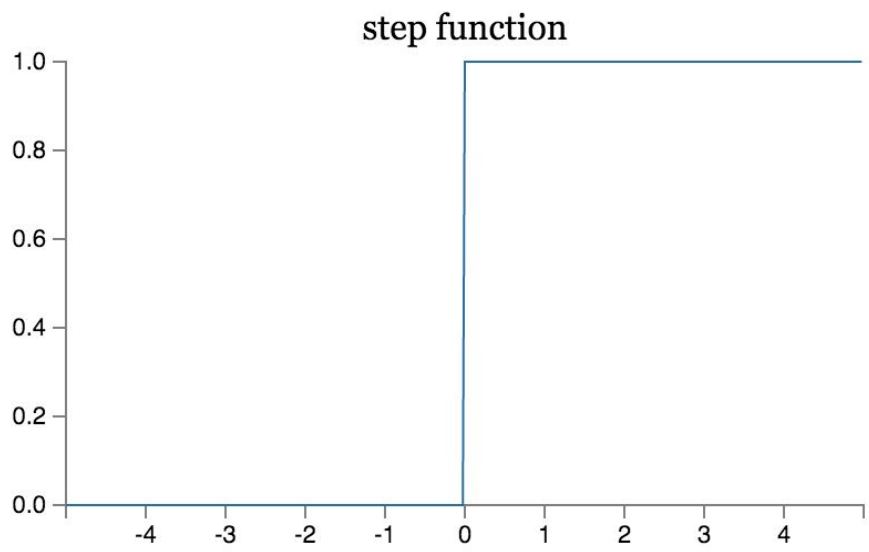
- Mathematically, this boils down to the following step-function relationship:

$$output = \begin{cases} 0 & \sum_i w_i x_i \leq threshold \\ 1 & \sum_i w_i x_i > threshold \end{cases}$$

- Note how similar this process is to the firing of a neuron in the human brain.

The Artificial Model of Neurons: The Perceptron

- Graphically, the perceptron determines the output as a step-function:



- Here:
  - The x-axis represents the sum of all the input/weight combinations.
    - 0 on the x-axis represents the threshold.
  - The y-axis represents whether the decision has been made (i.e., whether the neuron fires).

The Perceptron: A Simple Example

- Essentially, the perceptron is a simple way of making a decision by weighing the evidence at hand and comparing it to a threshold that represents a willingness to make that decision.

- Consider the following example of deciding whether or not you decide to attend lecture. You make your decision based on the following:
  - x_1: Is the weather going to make your commute pleasant?
  - x_2: Is the lecture topic particularly interesting?
  - x_3: Did you finish your project?

- Explain the scenarios and your ultimate decision under the following conditions:
  - $w_1 = 2$, $w_2 = 2$, $w_3 = 6$, and the threshold is 5.
  - $w_1 = 2$, $w_2 = 2$, $w_3 = 6$, and the threshold is 3.

The Perceptron: Threshold & Bias

- Often, it is easier to express the decision-making process of a perceptron by simplifying the notation as follows:

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

- The process is identical to the threshold representation we saw before; however, there are two small improvements:
  - The summation of weights and inputs is represented by a dot product.
  - The threshold has been moved to the other side of the equation. This new term is referred to as the bias, and relates to the aforementioned threshold as follows:

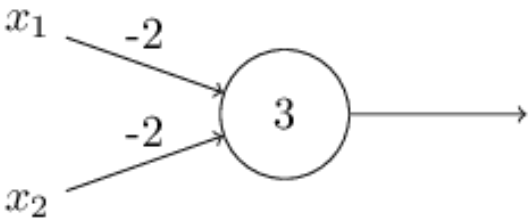
$$bias = -threshold$$

- While the terminology and notation can sometimes get unwieldy, the perceptron process boils down to the same principles of modeling basic decision-making.

The Perceptron: Modeling Logical Gates

- Another way we can think about perceptrons is as basic models for various logical gates (e.g., and, or, not-and, etc.). These are basic logical decision-making tools given binary inputs.

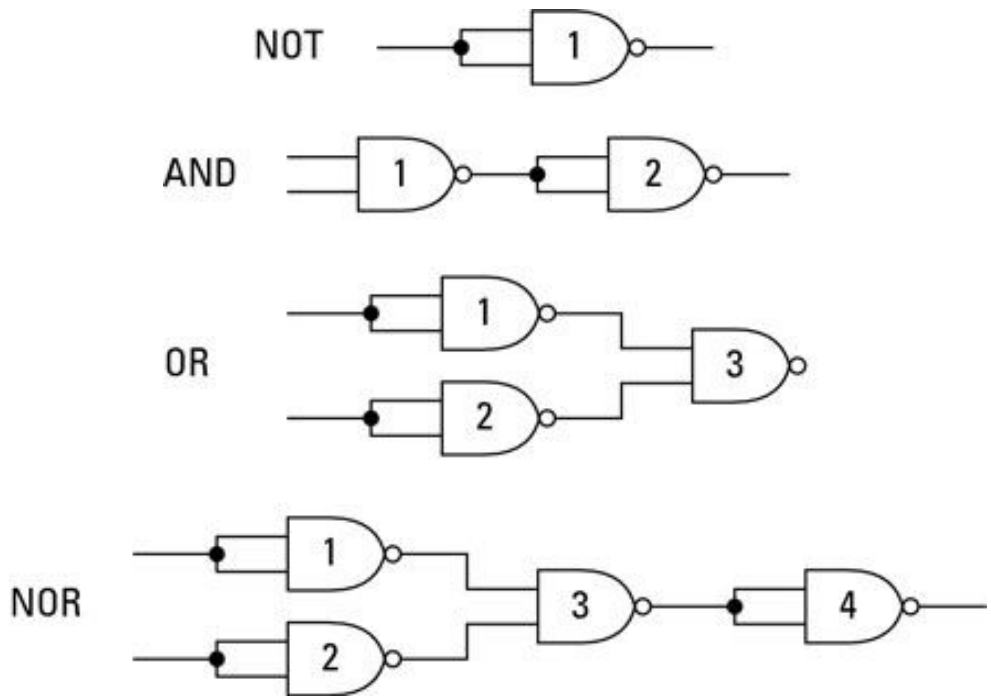
- For example, consider the following perceptron with weights $w_1 = w_2 = -2$ and bias $b = 3$:



  - If $x_1 = 0$ & $x_2 = 0$: (-2)(0) + (-2)(0) + 3 = 3 → output = 1
  - If $x_1 = 1$ & $x_2 = 0$: (-2)(1) + (-2)(0) + 3 = 1 → output = 1
  - If $x_1 = 0$ & $x_2 = 1$: (-2)(0) + (-2)(1) + 3 = 1 → output = 1
  - If $x_1 = 1$ & $x_2 = 1$: (-2)(1) + (-2)(1) + 3 = -1 → output = 0

The Perceptron: Modeling Logical Gates

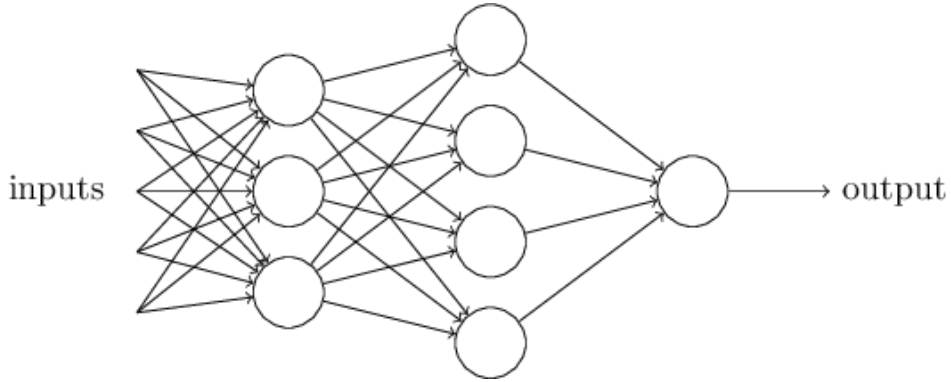
- The previous example shows that a single basic perceptron can actually model the logical NAND gate (i.e., not-and); this is not a trivial finding!
  - It can be shown that the NAND gate is universal for computation. In other words, any computation can be built up out of multiple NAND gates:



Making More Complex Decisions

- While the single perceptron has direct parallels to the biological neuron, it is a vast over-simplification of the human decision-making process. How can we improve upon this procedure to model more complex decisions?

- One way is to consider various perceptrons alongside one another; in this fashion, it seems plausible that a network of neurons might be more powerful.
  - The complexity/design of the network is often referred to as its topology.
  - Each column of neurons describes a layer of learning; each layer passes its outputs to a future layer for more abstract learning.



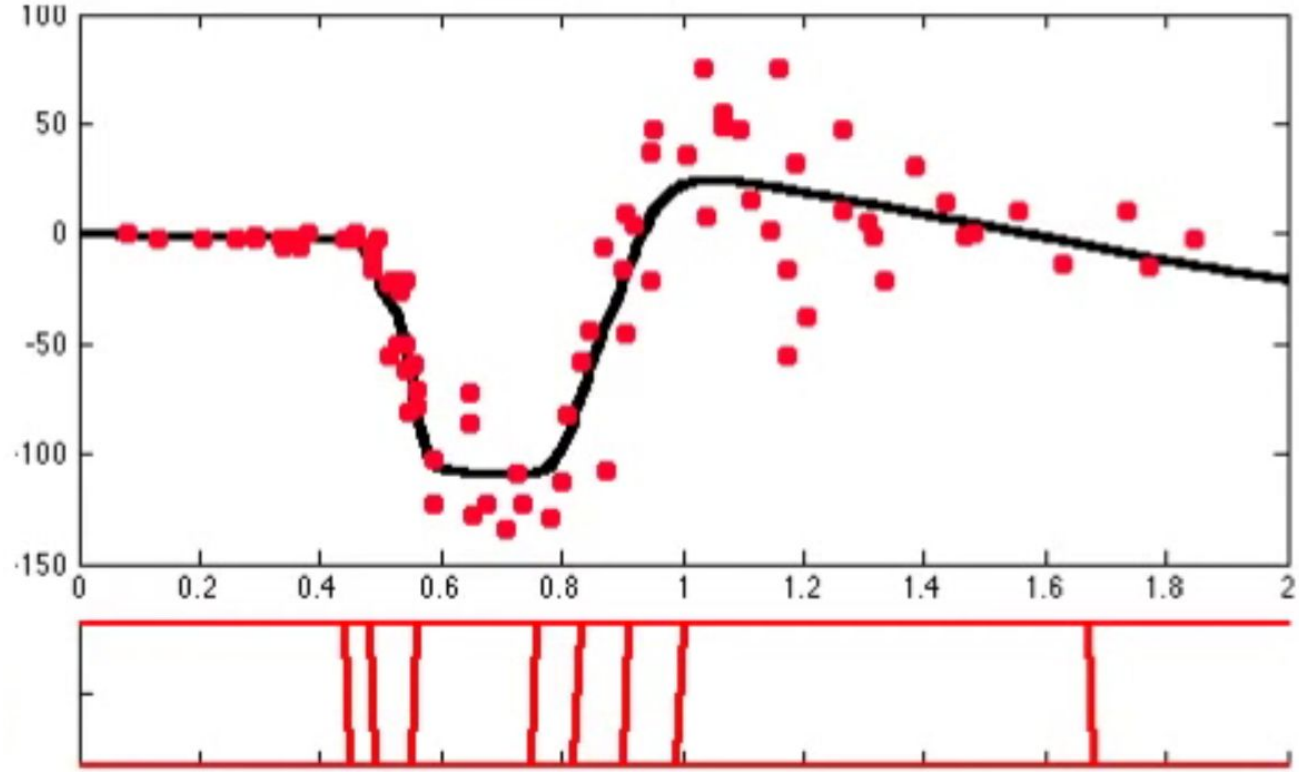
Making More Complex Decisions: Modeling Arbitrary Functions

- It can be shown that any bounded continuous function can be represented with a neural network (to an arbitrary degree of error ε) that has only one hidden layer:
  - Consider each hidden node turning on at a particular point of space.
  - Differences between these hidden values can hone in on particular region.
  - To approximate any function, choose weights that reproduce desired function values in specific regions.

- Essentially we are discretizing the function into arbitrarily small regions.

Making More Complex Decisions: Modeling Arbitrary Functions

- Another viewpoint of multiple hidden nodes turning on and off at particular points in order to model a complicated function together:



Image

Making More Complex Decisions: The Goal

- So what's the big deal?
  - Aren't perceptrons just glorified NAND gates?
  - Aren't perceptrons just glorified ways of writing complicated functions?

- Although this point of view is a bit limiting, it relies on the notion that we as the user are manually constructing networks of perceptrons with weights and biases in a specific fashion to recall some underlying logical or functional structure.

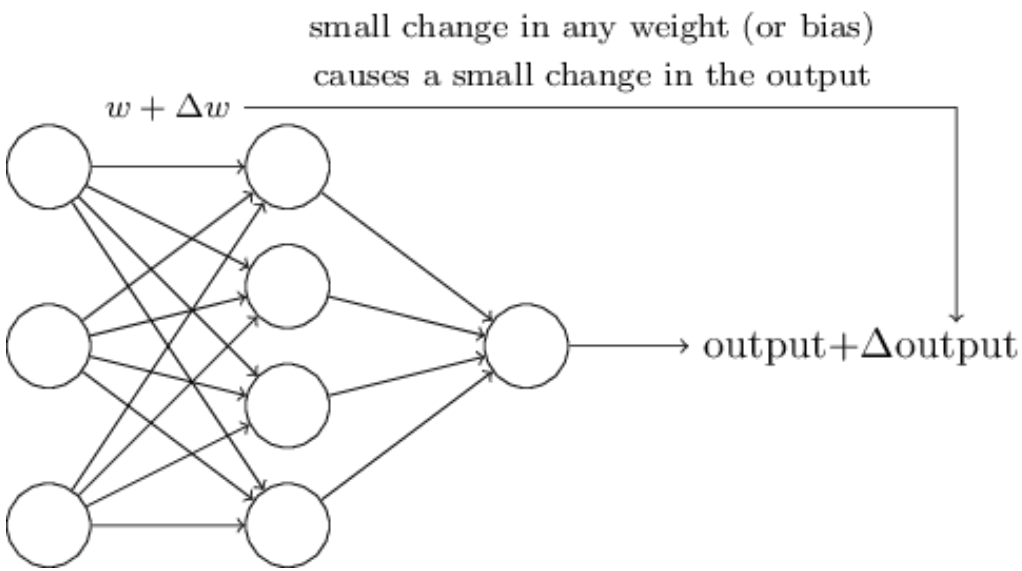
- Ideally, the goal is to come up with a particular learning algorithm that can automatically tune the weights and biases of an artificial neural network to match these logical/functional structures.
  - The vast benefit is that such a learning algorithm could construct solutions to problems where manual construction would fail because of the sheer complexity of the underlying relationship.

PART 2

Sigmoid Neurons

Developing a Learning Algorithm: Examining Small Changes

- Suppose we have a particular network of perceptrons, but we do not know the weights and biases we should assign to each perceptron because it's too complicated to understand the interrelationships among the connections.
  - Ideally, we would like a small change in either a particular weight or bias in the network to only cause a small change in the output; this would help us understand the relationships among the network.



Developing a Learning Algorithm: Examining Small Changes

- If it were true that a small change in a particular weight/bias caused only a small corresponding change in the output, we could use this information to tune our network in order to receive a better, more accurate output.

- To have our network learn, we could implement the following process:
  a. Change a weight or bias.
  b. Observe the change in the output.
     i. If the change made our prediction better, keep it and now manipulate a different weight.
     ii. If the change made our prediction worse, forget it and try again.

- Upon repeated steps, an accumulation of much smaller changes could induce a big change in the output that ultimately leads to greater accuracy.

The Problem with Perceptrons

- The basic underpinning of how a neural network learns relies on the fact that small changes in tuning parameters induce small changes on the output; however, this is not the case when our network is comprised of perceptrons.

- Recall that perceptrons are essentially step-functions that flip from "off" to "on" once a specific threshold (bias) value has been surpassed.
  - A small change in weights or bias for a particular perceptron could cause the output of the perceptron to completely switch.
  - Like a domino effect, this single switch would pass along to future perceptrons in the network and could end up completely changing the ultimate output drastically -- not a minor change at all!

- Using perceptrons, it doesn't seem like there is an easy way to gradually modify weights and biases to develop a robust learning algorithm.

The Sigmoid Neuron

- The easiest and most common way of overcoming the sensitivity issue presented by perceptron networks is by implementing a different kind of neuron called a sigmoid neuron. Sigmoid neurons have:
  - Inputs $x_1, x_2, \ldots, x_n$ and an overall output that takes on any values between 0 and 1.
  - Corresponding weights $w_1, w_2, \ldots, w_n$ that express the importance of the input in determining the output.
  - An overall bias (threshold).

- The sigmoid function is written as follows:

$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

The Sigmoid Neuron

- Combining these together, given the inputs, weights, and bias/threshold of a sigmoid neuron, the resulting equation for the output is as follows:

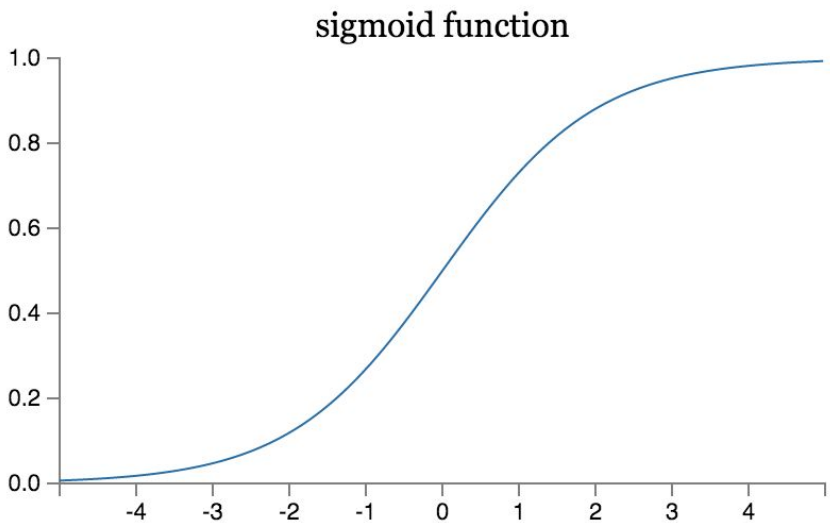
$$\sigma(w \cdot x + b) = \frac{1}{1 + e^{-(w \cdot x + b)}}$$

- Although the form of the sigmoid neuron appears to be quite different than that of the perceptron, they are at heart quite similar. Suppose $z = w \cdot x + b$:
  - If z is a large positive number, then $e^{-z} \approx 0$, and $\sigma(z) \approx 1$.
  - If z is a large negative number, then $e^{-z} \rightarrow \infty$, and $\sigma(z) \approx 0$.

- The differences between perceptron and sigmoid neurons are only truly observable when z is of intermediate magnitude.

The Sigmoid Neuron: Visually

- Graphically, the sigmoid neuron determines the output as a logistic function:



- Here:
  - The x-axis represents the sum of all the input/weight combinations and the bias transformed by the sigmoid function.
  - The y-axis represents the tendency for a decision to be made; note that the neuron always fires, just now on a scale instead of in an all-or-nothing way.

The Sigmoid Neuron: The Solution

- Why is the sigmoid neuron the solution to the problems the perceptron presents?
  - The sigmoid neuron is a smooth function that is easily differentiable at every point; it also has nice properties that will aid in simplifying future calculations. The perceptron is not a smooth function and is not easily differentiable.
  - It can be shown that the sigmoid function has the following property that relates small changes in weights and biases to changes in the output:

$$\Delta output \approx \sum_i \frac{\partial output}{\partial w_i} \Delta w_i + \frac{\partial output}{\partial b}$$

  - As we saw before, perceptrons do not have an easy way of quantifying this relationship because of their seemingly erratic and sensitive behavior.

PART 3

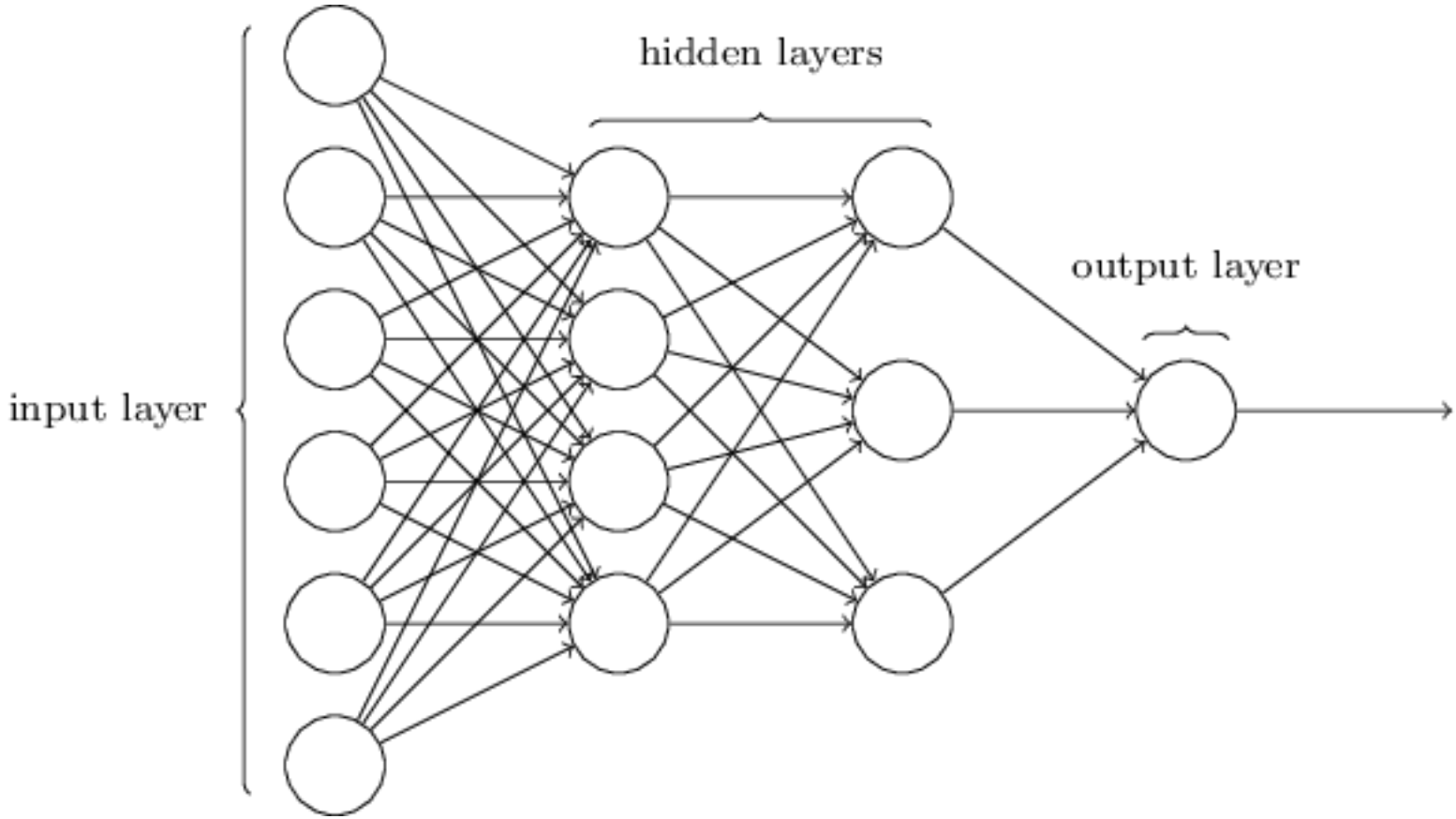
Network Topology & Hidden Features

Network Topology

- We briefly touched on the architecture of neural networks before; it will help to nail down some terminology before proceeding into more complicated examples of neural networks:
  - The leftmost layer in a network is called the input layer. Neurons within this layer are called input neurons.
  - The rightmost layer in a network is called the output layer. Neurons within this layer are called output neurons.
  - The middle layers in a network are called the hidden layers. Neurons within these layers are called hidden neurons.
    - Neural networks with more than one hidden layer are said to be deep.

- **NB:** Occasionally, networks with multiple layers are referred to as multilayer perceptrons or MLPs; however, despite the name, each neuron is made up of sigmoid neurons instead of perceptrons.

Network Topology: Visually



Image

Hidden Layers as Data Features

- The input and output layers make intuitive sense:
  - With the input layer, you know what goes in because it is simply your raw data.
  - With the output layer, you know what is supposed to come out because it is simply your desired target.

- What is really going on within the hidden layers of a neural network? This is essentially where the black box idea comes into play.
  - The idea is that the hidden layers actually represent the features of your data; as you pass from layer to layer, the features become increasingly complex.
  - The combination of various neuron outputs in the hidden layers intend to essentially feature engineer aspects of your data.

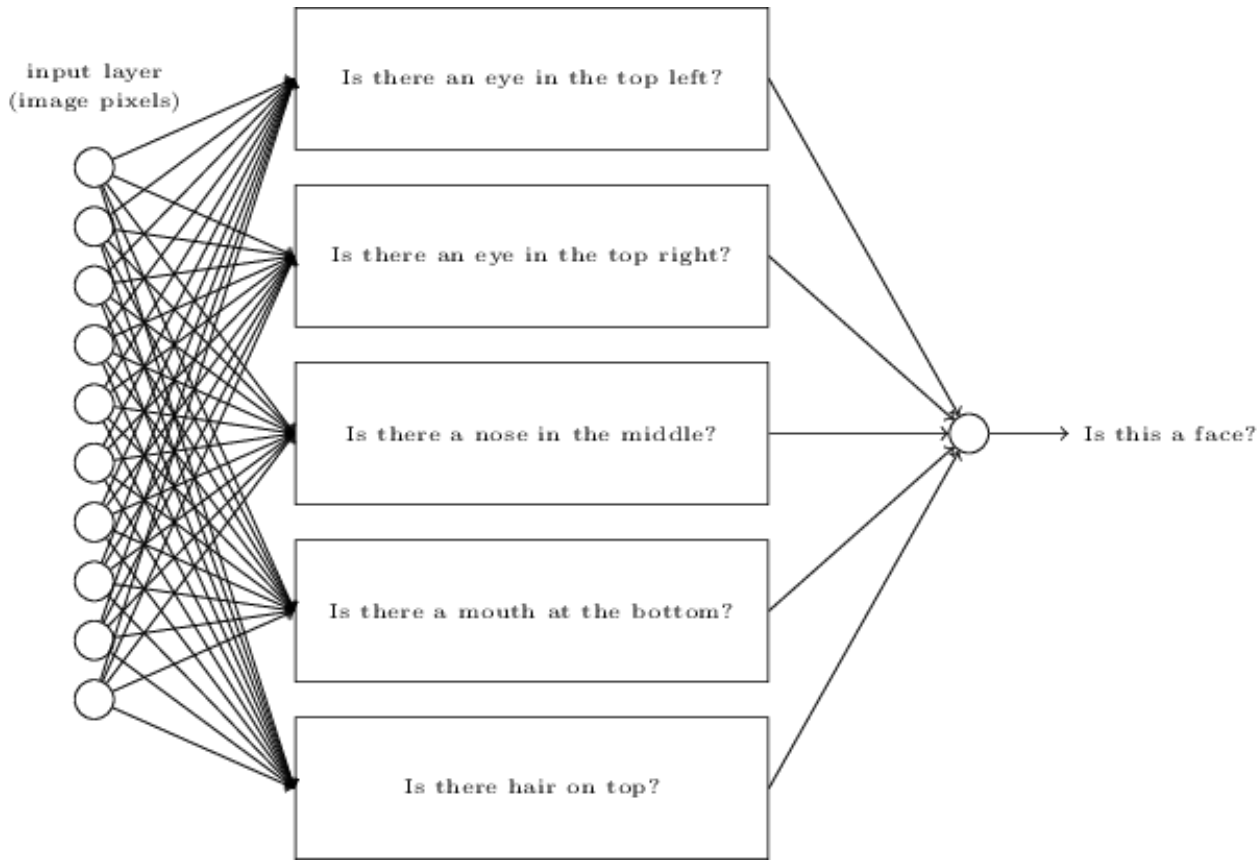
Hidden Layers as Data Features: An Example

- Let's take a step back from neural networks and try to define a way in which we might recognize whether or not a photograph contains a face. A common way to do this is to break down the problem into smaller sub-problems, such as:
  - Does the image have an eye in the top left?
  - Does the image have an eye in the top right?
  - Does the image have a nose in the center?
  - Does the image have a mouth in the bottom center?
  - Does the image have hair at the top?
  - Etc.

- Should we determine that many of these questions have answers "yes" or "probably," then we would classify the image as a face. Otherwise, if most of the answers to these questions are "no" or "probably not," then we would not classify the image as a face.

Hidden Layers as Data Features: An Example

- Intuitively, we can connect the the idea of answering each of these sub-questions as neurons embedded within a hidden layer of a neural network as follows:



Image

Hidden Layers as Data Features: An Example

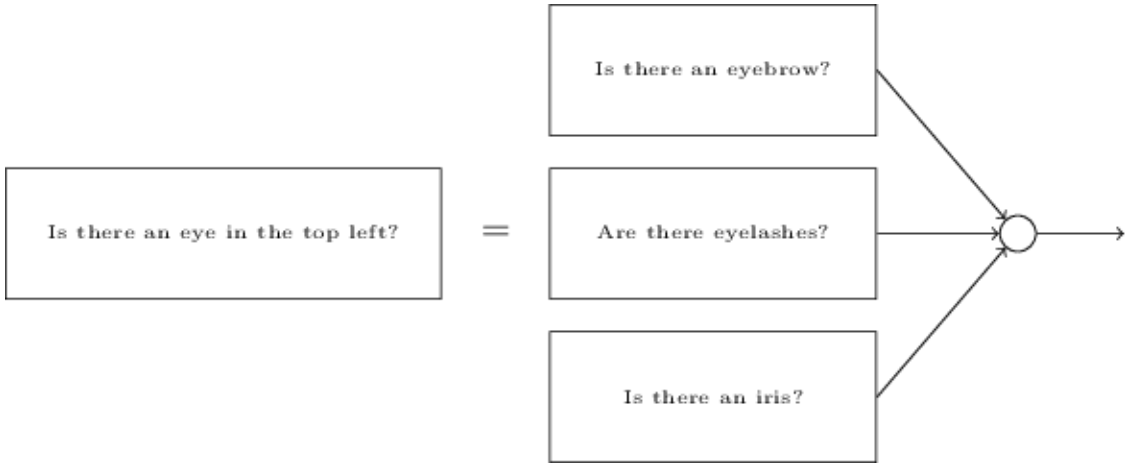
- Furthermore, each of the sub-problems can themselves be decomposed into even smaller sub-problems. Take, for example, a decomposition of the first question "Does the image have an eye in the top left?":
  - Does the image have an eyebrow in the top left?
  - Does the image have eyelashes in the top left?
  - Does the image have an iris in the top left?
  - Etc.

- Once again, the cumulative answers to these questions will help us determine whether or not there actually is an eye in the top left of the image.

- We can imagine that the hidden layers of a neural network are continually breaking down harder questions by attempting to first answer multiple, easier to answer questions.

Hidden Layers as Data Features: An Example

- Once again, we can connect the the idea of answering each of these broken down sub-questions as neurons embedded within an earlier hidden layer of a neural network as follows:

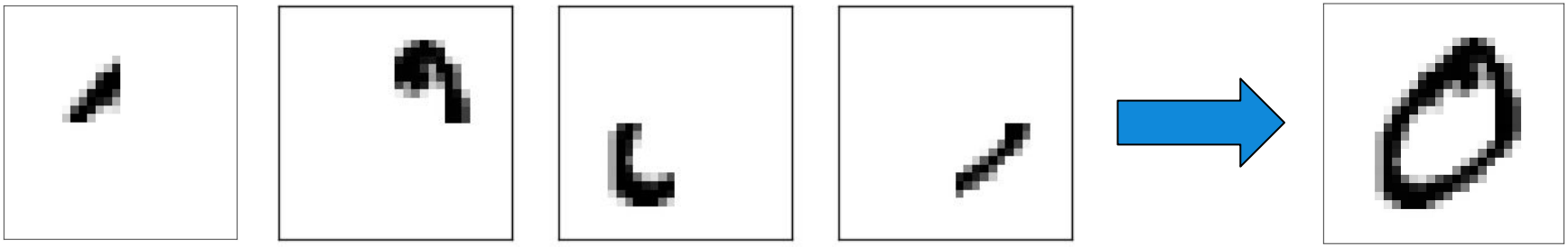


- The whole idea of hidden layers is to create a hierarchy that relies on the notion that the answers to a slew of easy questions will ultimately help answer one massively difficult question.

Hidden Layers as Data Features: Visually

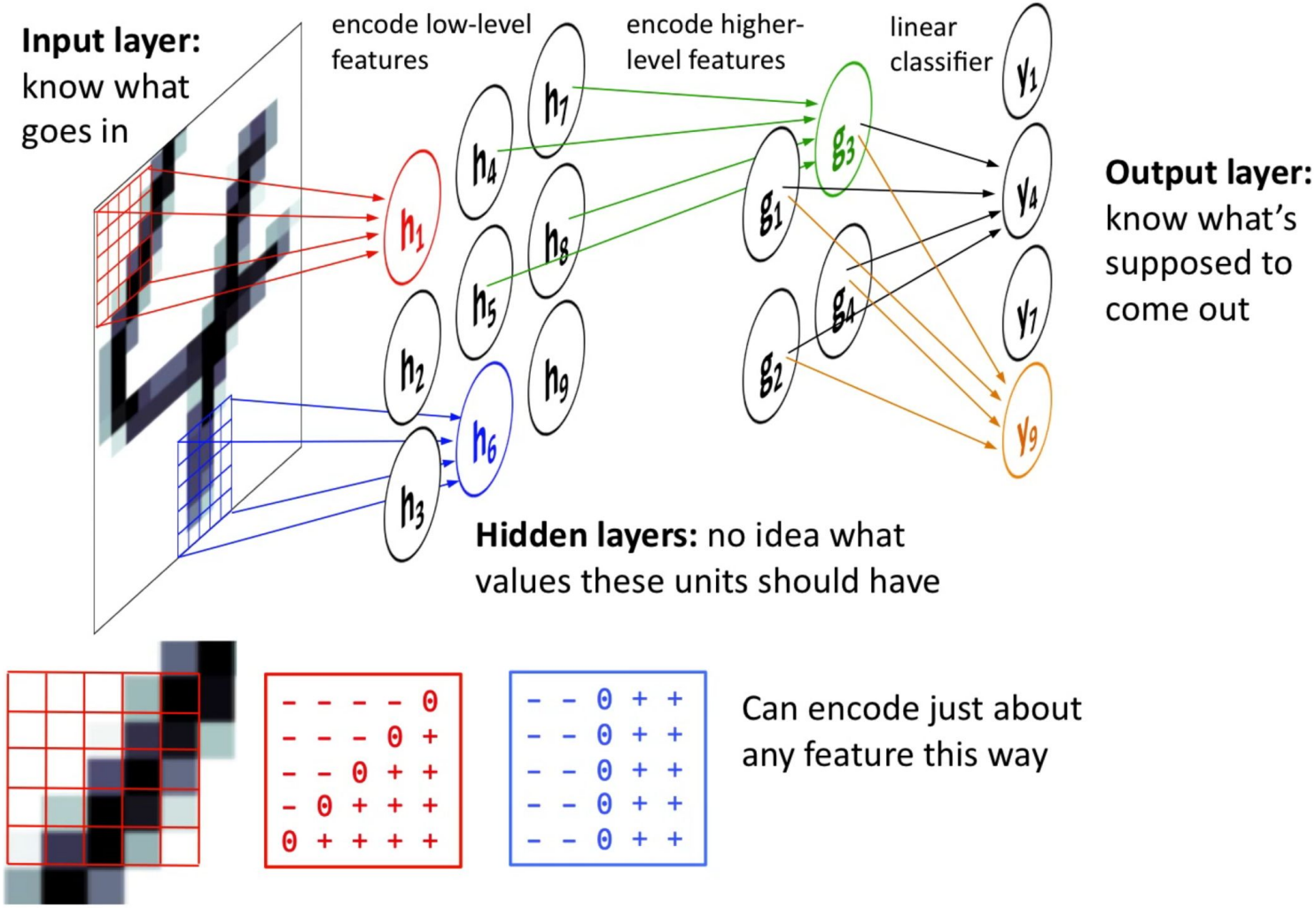
- Suppose we are trying to classify images of handwritten digits:
  - The intensity of each pixel of an image will be inputs of our neural network.
  - The output will be the likelihood of belonging to a particular class (digit).
  - The hidden layers will try to uncover various types of structure among digits.

- Suppose that four of the hidden nodes are weighing pixels in the following regions particularly high:



- Then, as a group, these hidden nodes would inform the notion that the observed digit has a high likelihood of truly being a 0.

Hidden Layers as Data Features: Visually



Putting it All Together

- What do we know so far?
  - We saw earlier that neurons (e.g., perceptrons or sigmoid neurons) can be used in series to construct any arbitrary logical gate or function.
  - We also just learned that that the purpose of hidden layers is to encode the various low- and high-level features of our data in a hierarchical manner.
  - Great! Let's combine the two notions and build our neural network to recognize all the aspects of our data from scratch!

- Not so fast...
  - Unfortunately, these hidden features and their corresponding weights, biases, and connections are often not intuitively conceivable.
  - We still have no clear way to determine the best combinations given our dataset, especially for deeper hidden layers.
  - We still need to figure out a way to help our neural network learn!

PART 4

Backpropagation Learning with Gradient Descent

Backpropagation: The Basic Idea

- A network topology that has not yet learned anything is essentially a blank slate; analogous to a puppy, it must be trained with repetition and experience:
  - Weights are initially random because of the non-existence of knowledge.
  - As a neural network processes input data, connections between the various neurons can be strengthened or weakened depending on how they ultimately seem to affect the output.
  - The errors that are initially made are back-propagated through the neural network, and the connections among neurons are changed in an effort to reduce this error.

- The backpropagation algorithm that is commonly implemented to train a neural network is extremely computationally expensive, even in its most efficient form.
  - The benefit of using backpropagation is that it helps yield highly accurate results because it tunes the network based on the errors themselves.

Backpropagation: The General Algorithm

- From the most general perspective, the backpropagation algorithm iterates through many cycles (epochs) of two processes:
  - The forward phase, in which the neurons are activated in sequence from the input layer, through the hidden layers, and lastly the output layer.
    - The predicted values are recorded.
  - The backward phase, in which the network's current output resulting from the forward phase is compared to the true target values in the training data.
    - The error is propagated backwards through the network from the output layer, through the hidden layers, and back to the input layer in order to modify the connections with the goal of reducing the error.
  - Repeat the cycle until a certain stopping criterion is reached.

- Still, the connections between all of the neurons is very complicated; how does the algorithm know the best way to modify the weights among connections?

Gradient Descent: The General Algorithm

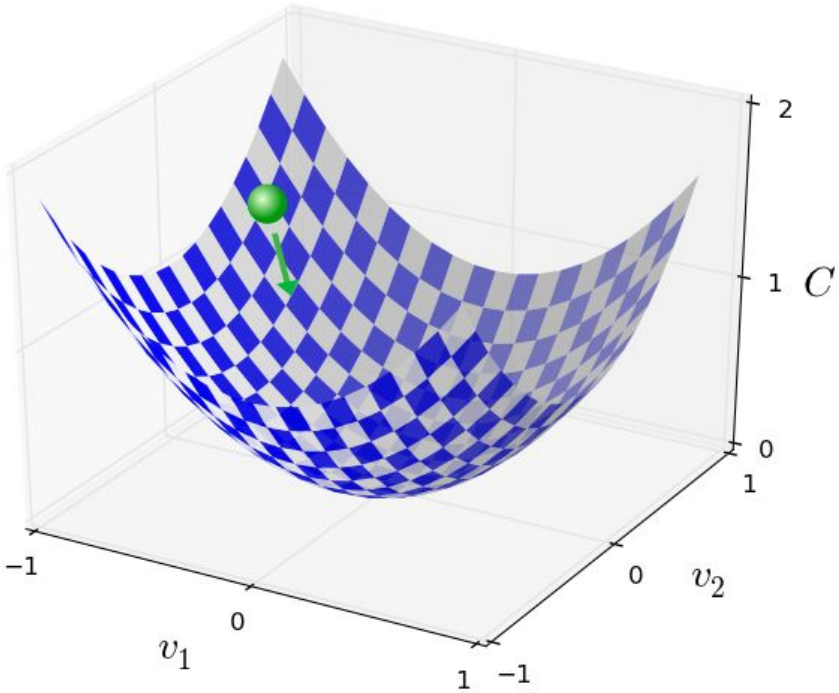
- The backpropagation algorithm relies on the idea of gradient descent, which essentially dictates the ways in which the various parameters of the neural network should be changed at each step.

- In gradient descent, the derivative of each neuron's activation function is used to identify the gradient in the direction of each of the incoming weights.
  - The algorithm will attempt to change the weights in such a way that will result in the greatest reduction of error.

- **NB:** In order to implement gradient descent, it is necessary that we have a differentiable function. This is why it was so important to use sigmoid neurons instead of perceptrons!

Gradient Descent: Visually

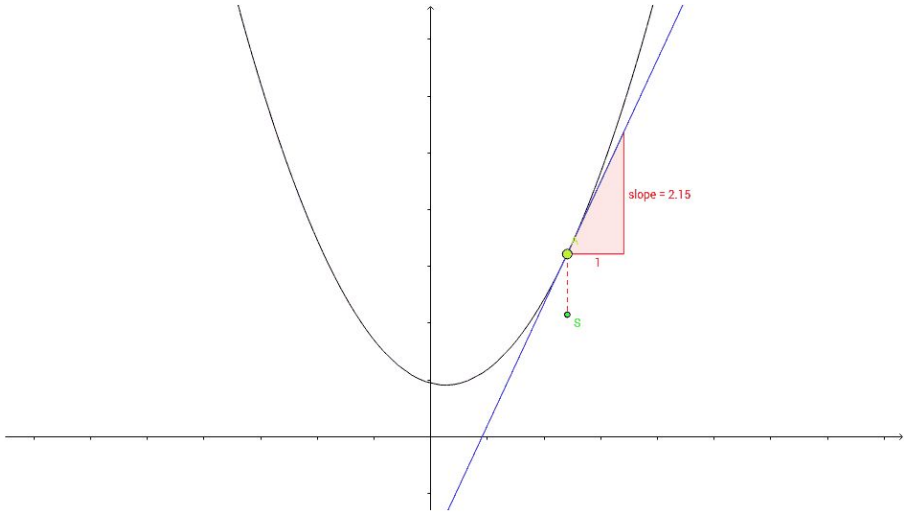
- Suppose you had a cost function C that is defined by a relationship among the variables v_1 and v_2. We desire to minimize this function as much as possible.
  - Gradient descent will assess the gradient at a specific point in order to push v_1 and v_2 in the direction that will minimize C.



Image

The Derivative: Useful Interpretations

- Before we can apply the backpropagation algorithm using gradient descent, it will be beneficial to quickly review some interpretations of the derivative of $f(x)$:
  - It describes how fast the function f changes instantaneously at the value x.
  - It determines whether the function f will increase or decrease if we increase the value of x.
  - It informs whether x is higher or lower than an optimum value (maximum or minimum).

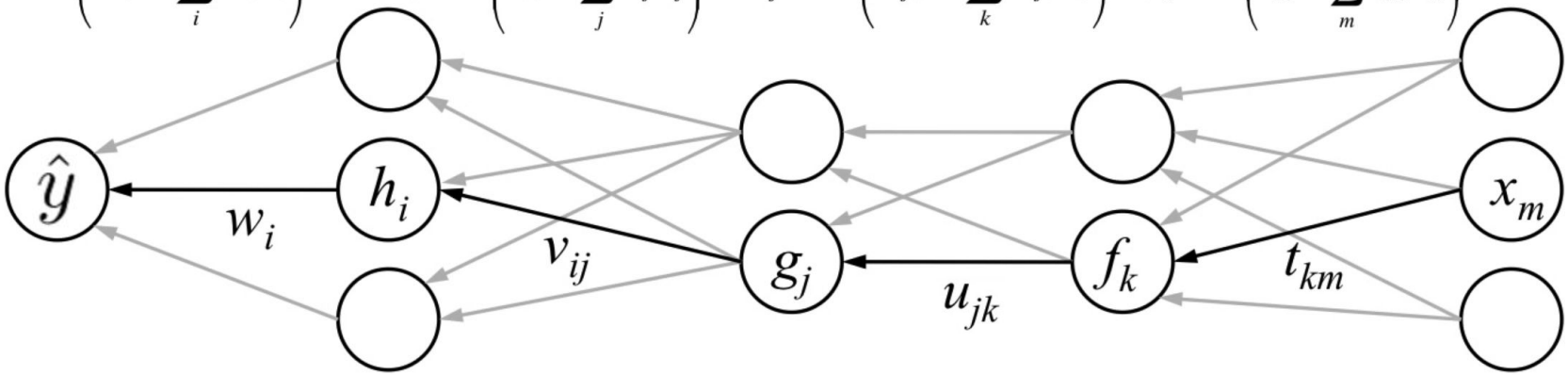


The Derivative: Useful Properties

- Furthermore, it will be useful to implement the following properties of the derivative in the backpropagation algorithm:
  - The chain rule:
$$\frac{\partial}{\partial x} f(g(x)) = \frac{\partial}{\partial g} f(g(x)) \cdot \frac{\partial}{\partial x} g(x)$$
  - The derivative of a sum is the same as the sum of the derivatives:
$$\frac{\partial}{\partial x} \sum_i f_i(x) = \sum_i \frac{\partial}{\partial x} f_i(x)$$
  - The derivative of a sum with respect to one element of the sum collapses:
$$\frac{\partial}{\partial x_k} \sum_i a_i f(x_i) = a_k \frac{\partial}{\partial x_k} f(x_k)$$
  - The derivative of the sigmoid function:
$$\sigma'(x) = \sigma(x)(1 - \sigma(x))$$
  - The derivative of the sigmoid function with the chain rule:
$$\frac{\partial}{\partial x} \sigma(f_x) = \sigma(f_x)(1 - \sigma(f_x)) \frac{\partial}{\partial x} f_x$$

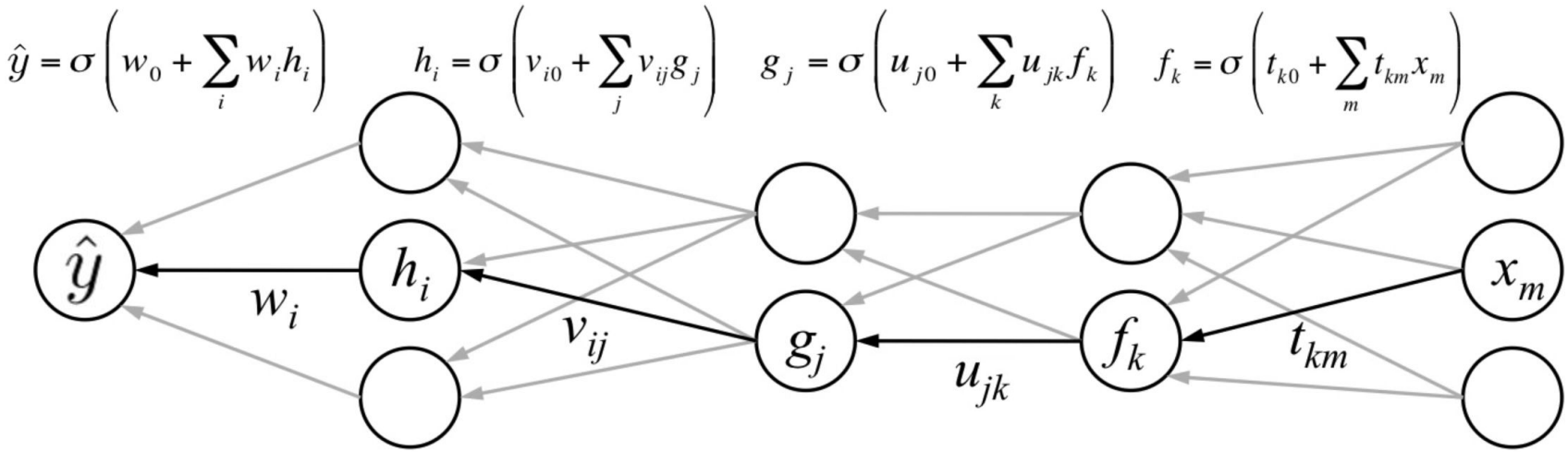
Backpropagation: Formally

$$\hat{y} = \sigma\left(w_0 + \sum_i w_i h_i\right) \qquad h_i = \sigma\left(v_{i0} + \sum_j v_{ij} g_j\right) \qquad g_j = \sigma\left(u_{j0} + \sum_k u_{jk} f_k\right) \qquad f_k = \sigma\left(t_{k0} + \sum_m t_{km} x_m\right)$$



- Suppose the above represents our neural network. For each layer in the network, we would like to determine:
  - The partial derivative of the error in respect to each neuron (i.e., should the neuron's value be higher or lower?).
  - The partial derivative of the error in respect to each input (i.e., should the input weight's value be higher or lower?).

Backpropagation: Formally

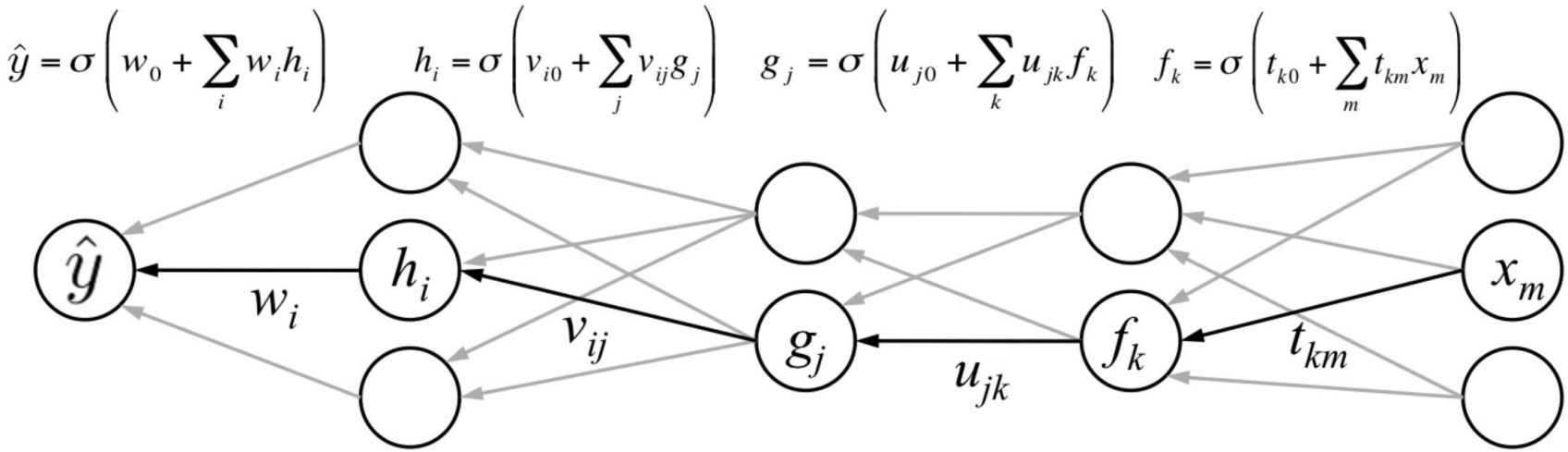


- Define the error as follows:

$$E = \frac{1}{2}(\hat{y} - y)^2$$

- **NB:** The choice to scale by ½ is somewhat arbitrary; however, it will make our future computations simpler.

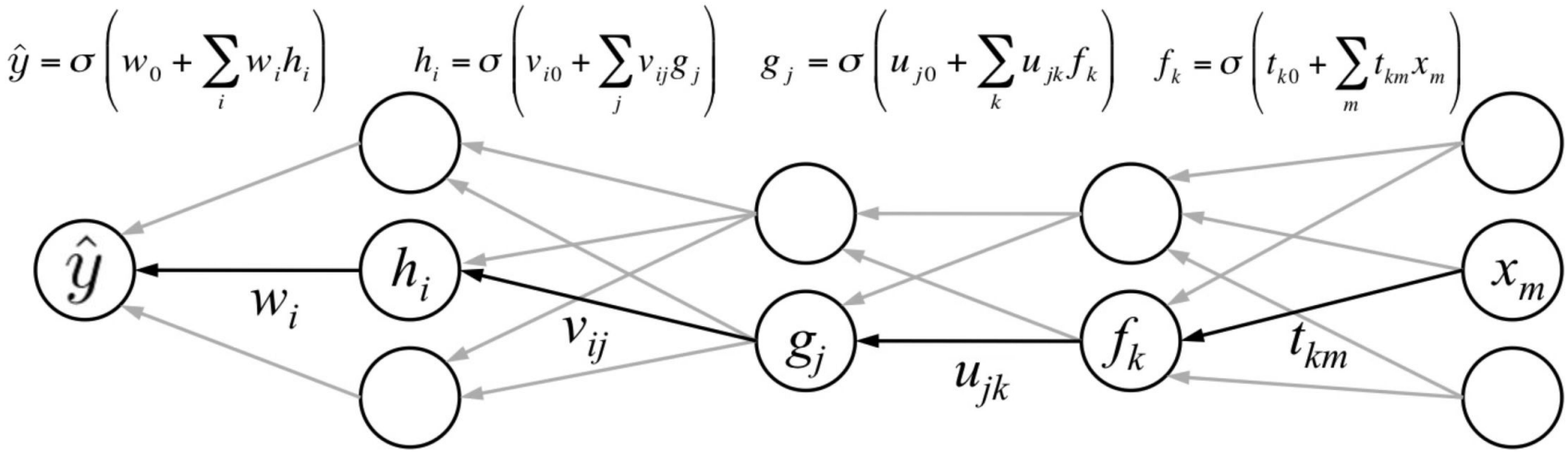
Backpropagation: Formally



- What happens if we differentiate the error E with respect to one of the neurons in the last hidden layer:

$$\frac{\partial E}{\partial h_i} = (\hat{y} - y)\frac{\partial \hat{y}}{\partial h_i}$$
$$= (\hat{y} - y)\hat{y}(1 - \hat{y})w_i$$

Backpropagation: Formally



- Let's go one level deeper:

$$\frac{\partial E}{\partial g_j} = (\hat{y} - y)\frac{\partial \hat{y}}{\partial g_j}$$

$$= (\hat{y} - y)\hat{y}(1 - \hat{y}) \sum_i w_i \frac{\partial h_i}{\partial g_j}$$

$$= (\hat{y} - y)\hat{y}(1 - \hat{y}) \sum_i w_i h_i (1 - h_i) v_{ij}$$

Backpropagation: Formally

- First we found that the following is true for the first layer of backpropagation:

$$\frac{\partial E}{\partial h_i} = (\hat{y} - y)\hat{y}(1 - \hat{y})w_i$$

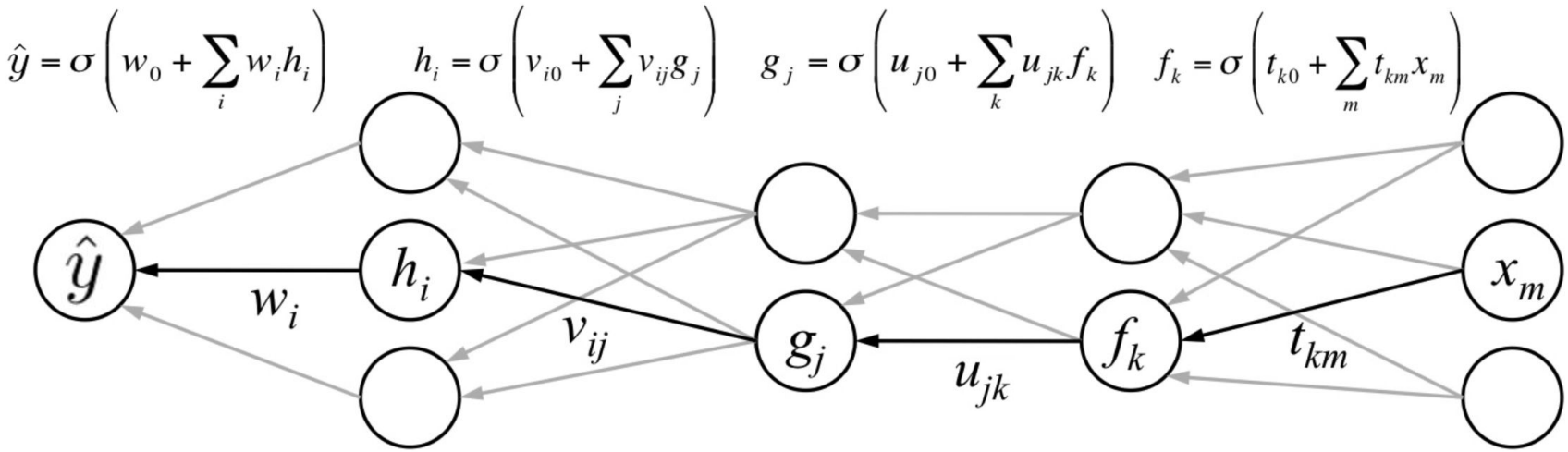
- Then we found that the following is true for the second layer of backpropagation:

$$\frac{\partial E}{\partial g_j} = (\hat{y} - y)\hat{y}(1 - \hat{y})\sum_i w_i h_i (1 - h_i) v_{ij}$$

- We begin to see this nesting nature; thus, we can express the partial derivative of a deeper layer of backpropagation with that of a shallower layer:

$$\frac{\partial E}{\partial g_j} = \sum_i h_i (1 - h_i) v_{ij} \frac{\partial E}{\partial h_i}$$

Backpropagation: Formally



- What if we wanted to differentiate the error E with respect to one of the edges?

$$
\begin{aligned}
\frac{\partial E}{\partial u_{jk}} &= (\hat{y} - y)\frac{\partial \hat{y}}{\partial u_{jk}} \\
&= (\hat{y} - y)\hat{y}(1 - \hat{y}) \sum_i w_i \frac{\partial h_i}{\partial u_{jk}} \\
&= (\hat{y} - y)\hat{y}(1 - \hat{y}) \sum_i w_i h_i (1 - h_i) v_{ij} \frac{\partial g_j}{\partial u_{jk}} \\
&= (\hat{y} - y)\hat{y}(1 - \hat{y}) \sum_i w_i h_i (1 - h_i) v_{ij} g_j (1 - g_j) f_k
\end{aligned}
$$

Backpropagation: Formally

- Once again, we previously found that the following is true:

$$\frac{\partial E}{\partial g_j} = (\hat{y} - y)\hat{y}(1 - \hat{y}) \sum_i w_i h_i (1 - h_i) v_{ij}$$

- Then we found that the following is true:

$$\frac{\partial E}{\partial u_{jk}} = (\hat{y} - y)\hat{y}(1 - \hat{y}) \sum_i w_i h_i (1 - h_i) v_{ij} g_j (1 - g_j) f_k$$

- We again see this nesting nature; thus, we can express the partial derivative of a deeper edge of backpropagation with that of a shallower layer's node:

$$\frac{\partial E}{\partial u_{jk}} = g_j (1 - g_j) f_k \frac{\partial E}{\partial g_j}$$

Backpropagation: Formally

- Once we have computed all of the gradients for each of the weights in our network, we gain insight into how slight perturbations of the weights relate to the overall error. Now, it is easy to decide whether we should increase or decrease each weight!

- Given a particular weight w_i in our network and the gradient of the error in respect to that weight, we update w_i as follows:

$$w_i' = w_i - \eta \frac{\partial E}{\partial w_i}$$

- Here:
  - If the gradient is positive, we shift w_i away from the increasing tendency.
  - If the gradient is negative, we shift w_i towards the decreasing tendency.
  - **NB:** η is a small positive number referred to as the learning rate.

PART 5

Additional Remarks on Neural Networks

The Benefit of Gradient Descent

- Why are we using gradient descent in the first place? Couldn't we have just iterated across a bunch of different combinations of weight values and chosen the one that yielded the minimum test error?
  - The problem is that the curse of dimensionality is extremely harsh!

- Suppose we only had one weight to optimize and that it only took about 0.04 seconds to iterate through 1,000 different values of this single weight.
  - In this case, choosing the best weight among the options is simple and fast.

- Suppose we now had two weights to optimize. To maintain the same precision as before, we would have to iterate through $1{,}000^2$ = 1,000,000 combinations!
  - In this case, it would take about 40 seconds to compute all the different combinations, which is definitely not as fast!

The Benefit of Gradient Descent

- Suppose now we had three weights we wish to optimize. Now:
  - We need to check over $1{,}000^3$ = 1,000,000,000 combinations to ensure the same level of precision as our original calculation.
  - This would take over 11 hours to compute!

- Lastly, suppose we had nine weights we wish to optimize. Now:
  - We need to check over $1{,}000^9$ = 1,000,000,000,000,000,000,000,000,000 combinations to ensure the same level of precision as our original calculation.
  - This would take over 1,268,391,679,350,583 years to compute; literally longer than the universe has been in existence!

- Solving the same problem, the method of gradient descent would only need to execute about 100 iterations -- so much faster!

Notes on Local Minimums

- What if our cost function doesn't always go in the same direction? What if it's possible that we reach a plateau or a local minimum? In other words, what happens when the problem is non-convex?

- We chose our error function the way that we did not only because it makes the calculation of derivatives a bit cleaner, but also because we are exploiting the often generally convex nature of quadratic equations.

- Also, another possible way around this is to update our weights sequentially instead of all at once (i.e., implementing stochastic gradient descent); in this manner, we might be able to avoid local minimums by sequentially following gradients in varying directions.

A Summary of Common Terminology

- The concepts of neural networks build upon ideas we have previously seen in respect to other machine learning algorithms -- particularly regression. While the vocabulary is different, the terms generally point to similar concepts:

Statistical Term	Neural Network Term
Coefficient	Weight
Predictor	Input
Response	Output
Observation	Exemplar
Parameter Estimation	Training/Learning
Steepest Descent	Back-Propagation
Intercept	Bias Term
Derived Predictor	Hidden Node
Penalty Function	Weight Decay

Pros & Cons of Neural Networks

- Pros:
  - Neural network models can be extremely flexible; with sufficient data they can effectively model curvatures, interactions, plateaus, step functions, etc.
  - Standard regression assumptions (e.g., the true residuals are independent, normally distributed, and have constant variance) are not required.
  - Outliers tend to have a limited influence in comparison to standard regression approaches.

- Cons:
  - The method depends on the availability of large datasets and is extremely computationally expensive.
  - Model parameters are vastly uninterpretable.
  - It is easy to overfit or underfit the training data.
  - Diagnostic tests are not widely developed.

PART 6

Review

Review

- Part 1: Neural Networks & Perceptrons
  - Neural Networks
  - The Biological Model of Neurons
  - The Artificial Model of Neurons
  - The Perceptron
    - A Simple Example
    - Threshold & Bias
    - Modeling Logical Gates
  - Making More Complex Decisions
    - Modeling Arbitrary Functions
    - The Goal
- Part 2: Sigmoid Neurons
  - Developing a Learning Algorithm: Examining Small Changes
  - The Problem with Perceptrons
  - The Sigmoid Neuron
    - Visually
    - The Solution
- Part 3: Network Topology & Hidden Features
  - Network Topology
    - Visually
  - Hidden Layers as Data Features
    - An Example
    - Visually
  - Putting it All Together
- Part 4: Backpropagation Learning with Gradient Descent
  - Backpropagation: The Basic Idea

Review

- Backpropagation: The General Algorithm
- Gradient Descent: Visually
- The Derivative
  - Useful Interpretations
  - Useful Properties
- Backpropagation: Formally

- Part 5: Additional Remarks on Neural Networks
  - The Benefit of Gradient Descent
  - Notes on Local Minimums
  - A Summary of Terminology
  - Pros & Cons of Neural Networks

- Part 6: Review