Machine Intelligence

Christopher Bender

Contents

1 The Learning Problem

In order to understand how machines are capable of solving problems that require intelligence, we must first understand exactly what sorts of problems we are trying to solve and what it means to have solved such a problem.

Consider the following situation: An email provider wants to create a spam filter that is capable of recognizing whether any given email is spam or not. This sort of task is remarkably easy for humans, yet it is difficult to explicitly program an algorithm that recognizes spam because there is no well-defined set of characteristics common to all spam emails. In essence, the key difficulty is formalizing the subjectivity of the task into an algorithm that a computer can execute.

We may rephrase the situation more explicitly: Let E be the set of all emails, and $S := \{\text{Spam}, \text{ Not Spam}\}$ be the set of possible decisions. There exists some ideal spam-detecting function $\gamma^* : E \to S$ that perfectly determines whether or not a given email is spam. Let $\mathcal{H}$ be the set of all possible mappings from E to S. Our goal is to find a function $h \in \mathcal{H}$ that identically matches (or, at the least, approximates) γ^*.

How can we find h? The most popular approach, known as *supervised learning*, utilizes a large dataset $\mathcal{D}$ of example emails and their corresponding spam / not spam designations. That is, $\mathcal{D}$ consists of pairs (e, s) for $e \in E$ and $s \in S$ that have been externally verified to be true. This approach makes sense, as learning from data is how humans themselves approach the problem of detecting spam; we are unable to produce a perfect set of properties that characterizes spam emails, but we still retain a vague notion of what a spam email is because we have seen many examples.

We may summarize our current observations with the following chart:

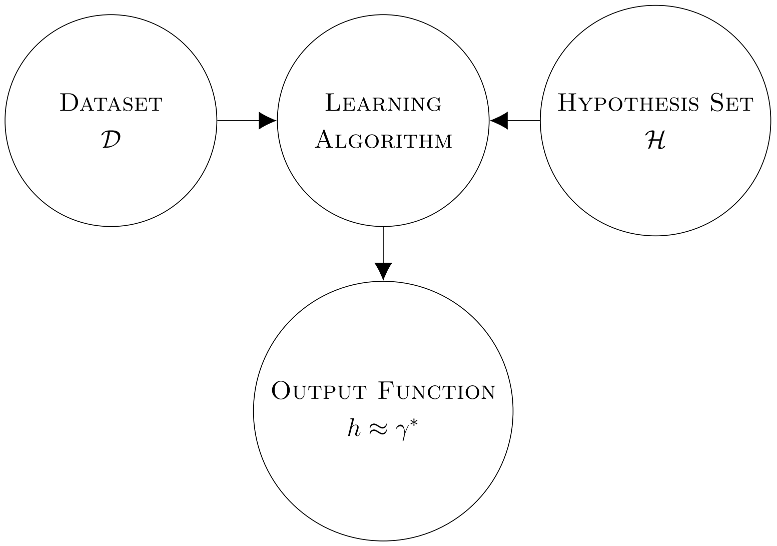

DATASET $\mathcal{D}$
LEARNING ALGORITHM
HYPOTHESIS SET $\mathcal{H}$
OUTPUT FUNCTION $h \approx \gamma^*$

Figure 1: A high-level outline of supervised learning.

Now, we may turn our attention to creating a learning algorithm that selects some hypothesis $h \in \mathcal{H}$ that approximates γ^*. Since the only known instance of a pattern-recognizer complex enough for the task is the human brain, we can look to neuroscience for motivation.

2 The Brain and Artificial Neural Networks

The brain consists of discrete cells, called neurons, organized in a network of layers. These layers are ordered in a hierarchy, with low-level layers taking raw sensory input and higher-level layers interpreting that input. This tiered process allows for high-level concepts such as emotion, art, and language to evolve from low-level input, such as light and shadow.

For example, consider the neural layers associated with recognizing an object. The following model shows how low-level concepts can produce high-level conclusions.

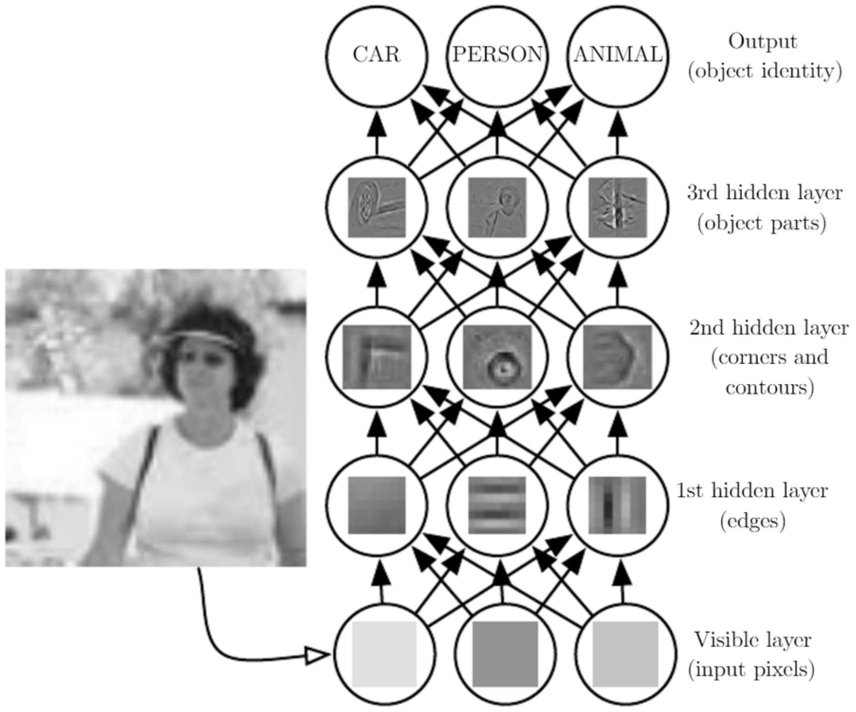

CAR
PERSON
ANIMAL
Output (object identity)
3rd hidden layer (object parts)
2nd hidden layer (corners and contours)
1st hidden layer (edges)
Visible layer (input pixels)

Figure 2: An example of hierarchical neural function. Image taken from [2].

We can see that individual neurons have very simple jobs: for example, recognizing what an edge looks like, or knowing that two perpendicular edges form a corner. However, collectively, the neurons are capable of performing an extremely complex task.

Thus, we now have sufficient motivation to introduce the concept of an *artificial neural network* (ANN). An ANN is a collection N of neurons partitioned into disjoint subsets $l_1, l_2, ..., l_L$ known as *layers*. A neuron $n \in N$ is a function that takes the real-valued outputs of the previous layer and maps to some other real number. Visually, we represent the neurons as nodes in a directed graph, where an arc is placed between two neurons if the first feeds its output into the second. l_1 is referred to as the "input layer" because it takes raw input values and feeds them into the network, while the layers $l_2, l_3, ..., l_{L-1}$ are known as the "hidden layers" because they are not directly interacted with. l_L is referred to as the "output layer" and often consists of a single neuron.

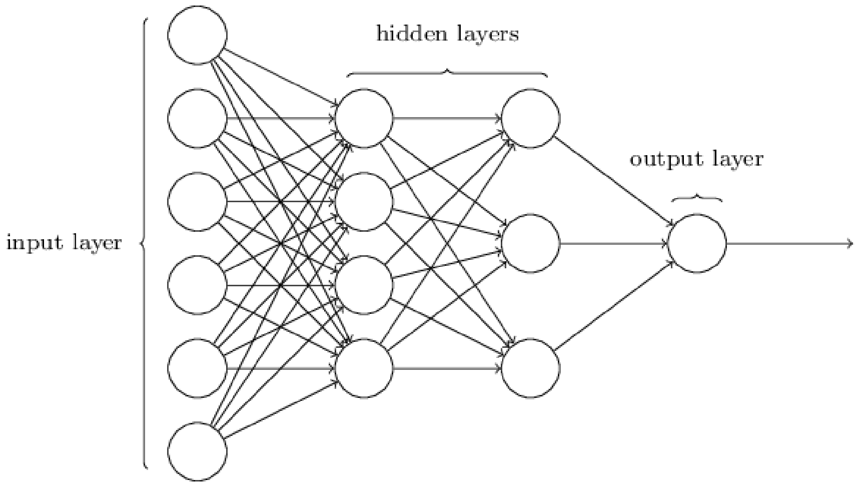


Figure 3: An artificial neural network. Image taken from [4].

For our purposes, we will only consider *feedforward networks*, or networks that only pass information forward between layers. That is, our networks will have no feedback loops where information from layer l_i is passed to layer l_j for $j \leq i$.

We begin by analyzing the simplest type of neural network, that of a single neuron.

3 The Perceptron

3.1 Overview

The perceptron is one of the oldest machine learning models, dating back to the 1950s. The model consists of a single neuron that maps some input vector $\boldsymbol{x} = [x_1, x_2, ..., x_n]^T \in \mathbb{R}^n$ to a binary decision $d \in \{-1, 1\}$ according to the mapping h.

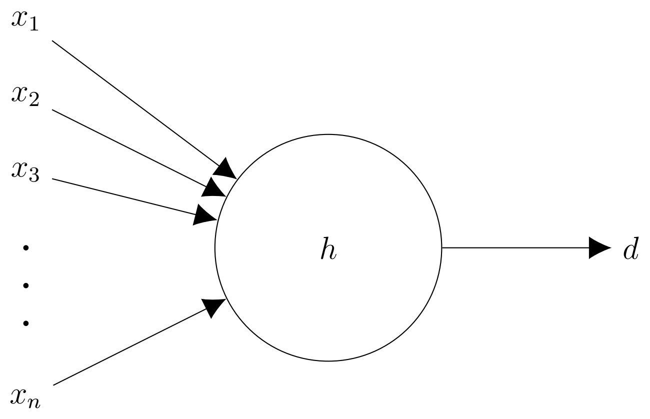


Figure 4: A perceptron model.

There are many different ways to construct a function $h : \mathbb{R}^n \to \{-1, 1\}$, but a highly effective method is to weight each of the coordinates of $\boldsymbol{x}$ by some fixed real number, and add the new weighted coordinates together. If the result is at least some threshold value, then we return 1. Else, we return -1.

That is, we find a vector of weights $\boldsymbol{w} = [w_1, w_2, ..., w_n]^T \in \mathbb{R}^n$ and a threshold value $t \in \mathbb{R}$ such that

$$h_{\boldsymbol{w}}(\boldsymbol{x}) := \begin{cases} 1 & \text{if } \sum_{i=1}^{n} w_i x_i \geq t \\ -1 & \text{otherwise} \end{cases}$$

We can rewrite this more compactly to yield

$$h_{\boldsymbol{w}}(\boldsymbol{x}) := \operatorname{sign}(\boldsymbol{w} \cdot \boldsymbol{x} - t),$$

where $\operatorname{sign}(x)$ is the sign of x (± 1) and is arbitrarily assigned to be 1 when $x = 0$.

To simplify our formula for h, we can account for our threshold factor t in another weight of $\boldsymbol{w}$. So, we now seek to find a vector $\boldsymbol{w} \in \mathbb{R}^{n+1}$ such that the approximating function

$$h_{\boldsymbol{w}}(\boldsymbol{x}) := \operatorname{sign}(\boldsymbol{w} \cdot \boldsymbol{x})$$

perfectly classifies all of our training examples $\boldsymbol{x} \in \{1\} \times \mathbb{R}^n$.

3.2 Learning Algorithm

We will now describe the algorithm by which we learn the weight vector $\boldsymbol{w}$. Suppose that there exists some dataset $\mathcal{D}$ of points $(\boldsymbol{x}, d)$ of training examples and their corresponding binary labels.

Begin by assigning zeroes to the entries of $\boldsymbol{w}$. We then iteratively improve $\boldsymbol{w}$ by the following process:

1. If all of the data points in $\mathcal{D}$ are correctly classified by our current function h, then we terminate the learning algorithm as we have found an ideal weight vector. Else, suppose that there exists some data point $p = (\boldsymbol{x}, d)$ for $x \in \mathbb{R}^{n+1}$ and $d \in \{-1, 1\}$ that is misclassified.
2. Add $d\boldsymbol{x}$ to $\boldsymbol{w}$.
3. Return to the first step.

We will now informally show why this algorithm is effective; that is, we will show that the addition of $d\boldsymbol{x}$ to $\boldsymbol{w}$ is indeed beneficial to the classification of the elements of $\mathcal{D}$. A formal proof of the convergence of this learning algorithm will follow at the end of the section.

Since our learning algorithm misclassifies p, we know that $d \neq h(\boldsymbol{x})$. Thus, we have that

$$d \cdot (\boldsymbol{w} \cdot \boldsymbol{x}) < 0$$

because d and $h(\boldsymbol{x})$ are of opposite signs.

Let $\boldsymbol{w}^* := \boldsymbol{w} + d\boldsymbol{x}$. We have that

$$\begin{aligned} d \cdot (\boldsymbol{w}^* \cdot \boldsymbol{x}) &= d \cdot (\boldsymbol{w} \cdot \boldsymbol{x}) + d^2 \cdot (\boldsymbol{x} \cdot \boldsymbol{x}) \\ &= d \cdot (\boldsymbol{w} \cdot \boldsymbol{x}) + ||\boldsymbol{x}||^2 \\ &> d \cdot (\boldsymbol{w} \cdot \boldsymbol{x}). \end{aligned}$$

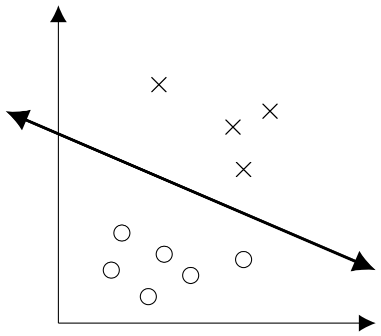
Note that a positive value of $d \cdot (\boldsymbol{w} \cdot \boldsymbol{x})$ is desirable because, in that case, the guess $h(\boldsymbol{x})$ and the actual value d agree in sign. Thus, we see that the change from $\boldsymbol{w}$ to $\boldsymbol{w}^*$ does indeed improve our guess for our particular input vector $\boldsymbol{x}$ because it forces the negative value $d \cdot (\boldsymbol{w} \cdot \boldsymbol{x})$ to become less negative.

3.3 Limitations

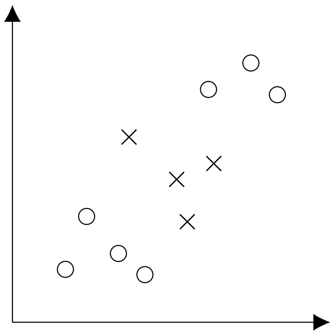
Given the relatively simple form of the perceptron decision algorithm, it is of no surprise that there are many cases in which the algorithm does not perform well, if at all.

Note that the function $h_{\boldsymbol{w}}(\boldsymbol{x}) = \text{sign}(\boldsymbol{w} \cdot \boldsymbol{x})$ divides the $(n+1)$-dimensional input space into two half-spaces along a single n-dimensional hyperplane. This implies that our given algorithm can only converge perfectly when our data is *linearly separable*, or when such a dividing hyperplane exists.

For example, consider the following dataset in two dimensions. The ideal classification of the input vectors is designated by X's and O's. We note that, since there exists a line that divides the data precisely according to the proper classification, the data is linearly separable, so our perceptron algorithm will indeed find such a line.



In the following example, however, it is clear that no such separating line exists. Here, the perceptron algorithm will not terminate and will fail to find a separating line.



3.4 A Proof of Convergence

We will now provide a rigorous proof that our perceptron learning algorithm will indeed converge on any linearly separable dataset.

Suppose that $\mathcal{X} := \{\boldsymbol{x}_1, \boldsymbol{x}_2, ..., \boldsymbol{x}_N\} \subset \{1\} \times \mathbb{R}^n$ is a linearly separable dataset of size N of vectors in $\{1\} \times \mathbb{R}^n$. Also, let $y_k \in \{-1, 1\}$ for $1 \leq k \leq N$ denote the corresponding correct classification of $\boldsymbol{x}_k$. Since $\mathcal{X}$ is linearly separable, there exists some ideal weight vector $\boldsymbol{w}^* \in \mathbb{R}^{n+1}$ such that $\boldsymbol{w}^*$ correctly classifies all the $\boldsymbol{x}_i$. That is,

$$y_k = h_{\boldsymbol{w}^*}(\boldsymbol{x}_k) := \operatorname{sign}(\boldsymbol{w}^* \cdot \boldsymbol{x}_k)$$

for all k. Note that the function $h_{\boldsymbol{w}^*}(\boldsymbol{x})$ does not change when $\boldsymbol{x}$ is scaled by some positive constant, so we may assume without loss of generality that both the vectors in $\mathcal{X}$ and the vector $\boldsymbol{w}^*$ are of unit length.

Let $\boldsymbol{w} : \mathbb{Z}_{\geq 0} \to \mathbb{R}^{n+1}$ be given by

$$\boldsymbol{w}(t) = \boldsymbol{w}(t-1) + y_k \boldsymbol{x}_k,$$

where $\boldsymbol{x_k}$ is any vector that is misclassified by $\boldsymbol{w}(t-1)$ and $\boldsymbol{w}(0)$ is defined to be the zero vector. A note should be made about the defined domain of $\boldsymbol{w}$: Our recursion can only continue as long as there exists some misclassified vector $\boldsymbol{x}_k$, so $\boldsymbol{w}$ may only be defined on some subset of $\mathbb{Z}_{\geq 0}$. However, in such a case, we may terminate the process, as we have found an adequate separating vector $\boldsymbol{w}$.

To prove convergence, we will take the inequality

$$\cos(\theta_{\boldsymbol{w},\boldsymbol{w}^*}) = \frac{\boldsymbol{w} \cdot \boldsymbol{w}^*}{||\boldsymbol{w}|| \cdot ||\boldsymbol{w}^*||} = \frac{\boldsymbol{w} \cdot \boldsymbol{w}^*}{||\boldsymbol{w}||} \leq 1$$

(where $\theta_{\boldsymbol{w},\boldsymbol{w}^*}$ denotes the angle between $\boldsymbol{w}$ and $\boldsymbol{w}^*$) and convert it into an inequality in terms of the number of iterations of our algorithm. To do this, we will first find an alternative formulation of the numerator:

Let

$$\rho := \min_{1 \leq k \leq N} y_k \cdot (\boldsymbol{w}^* \cdot \boldsymbol{x}_k).$$

Since $\boldsymbol{w}^*$ correctly classifies $\boldsymbol{x}_k$, we know that $y_k = h_{\boldsymbol{w}^*}(\boldsymbol{x}_k)$ for all k. Thus, we have that

$$y_k \cdot (\boldsymbol{w}^* \cdot \boldsymbol{x}_k) > 0$$

for $1 \leq k \leq N$. So, $\rho > 0$.

Now, consider the recursion

$$\boldsymbol{w}(t) = \boldsymbol{w}(t-1) + y_k \boldsymbol{x}_k.$$

We multiply by $\boldsymbol{w}^*$ to yield that

$$\begin{aligned} \boldsymbol{w}^* \cdot \boldsymbol{w}(t) &= \boldsymbol{w}^* \cdot \boldsymbol{w}(t-1) + y_k \cdot (\boldsymbol{w}^* \cdot x_k) \\ &\geq \boldsymbol{w}^* \cdot \boldsymbol{w}(t-1) + \rho \\ &\geq t\rho, \end{aligned}$$

where the last inequality is achieved because the process can be iterated down to $t = 0$.

Now that we have a bound for $\boldsymbol{w}^* \cdot \boldsymbol{w}(t)$, we seek to find a bound for $||\boldsymbol{w}(t)||$. We have that

$$\begin{aligned} ||\boldsymbol{w}(t)||^2 &= \boldsymbol{w}(t) \cdot \boldsymbol{w}(t) \\ &= (\boldsymbol{w}(t-1) + y_k \boldsymbol{x}_k) \cdot (\boldsymbol{w}(t-1) + y_k \boldsymbol{x}_k) \\ &= ||\boldsymbol{w}(t-1)||^2 + 2y_k \cdot (\boldsymbol{x}_k \cdot \boldsymbol{w}(t-1)) + y_k^2 \cdot ||\boldsymbol{x}_k||. \end{aligned}$$

However, because $\boldsymbol{w}(t-1)$ misclassifies $\boldsymbol{x}_k$, we have that

$$\begin{aligned} &||\boldsymbol{w}(t-1)||^2 + 2y_k \cdot (\boldsymbol{x}_k \cdot \boldsymbol{w}(t-1)) + y_k^2 \cdot ||\boldsymbol{x}_k|| \\ &\qquad \leq ||\boldsymbol{w}(t-1)||^2 + y_k^2 \cdot ||\boldsymbol{x}_k|| \\ &\qquad = ||\boldsymbol{w}(t-1)||^2 + 1. \end{aligned}$$

Thus, $||\boldsymbol{w}(t)||^2 \leq ||\boldsymbol{w}(t-1)||^2 + 1$. Iterating down to $t = 0$ as before, we get that

$$||\boldsymbol{w}(t)||^2 \leq t,$$

so

$$||\boldsymbol{w}(t)|| \leq \sqrt{t}.$$

Finally, we will combine these two inequalities with our initial cosine inequality to yield that

$$1 \geq \frac{\boldsymbol{w}^* \cdot \boldsymbol{w}(t)}{||\boldsymbol{w}(t)||} \geq \frac{t\rho}{\sqrt{t}} = \rho\sqrt{t}.$$

Rearranging, we get that $t \leq \frac{1}{\rho^2}$, thereby showing that the system will converge in at most $\frac{1}{\rho^2}$ iterations, as desired.

$\square$

4 Multilayer Networks

4.1 Necessity and Overview

As was mentioned in the previous section, the perceptron is a linear binary classifier, so the types of functions that it will be able to approximate well are limited. In particular, it is unable to approximate functions that either do not map onto $\{1, -1\}$ or are nonlinear in nature. Even if we extend the single-neuron perceptron algorithm to a multilayer system, the neural network will still only be able to predict linearly since a composition of linear function is linear.

As was briefly mentioned in the last section, a multilayer artificial neural network consists of a series of layers of individual neurons. Each of these neurons take the outputs from the previous layer, weight each of them by some real-valued factor, add up the weighted inputs, and apply a bias factor, just as the perceptron did. However, all of these operations are linear in each of the neuron's inputs, so we must add a nonlinear operation to the mix.

To introduce nonlinearity into the system, we will introduce the idea of an *activation function*. As applied to multilayer networks, activation functions are nonlinear scalar-to-scalar functions applied to neurons after they compute the weighted sum of prior nodes' outputs, as seen in the following diagram:

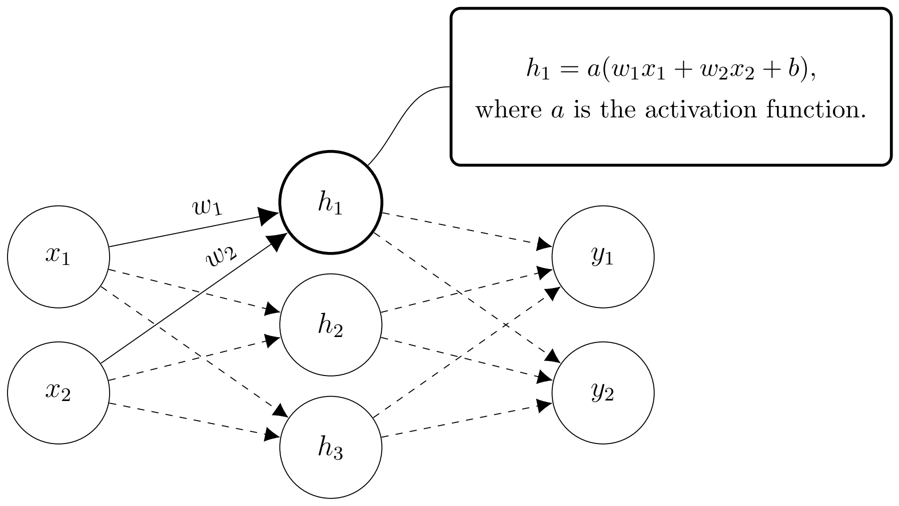

$h_1 = a(w_1x_1 + w_2x_2 + b)$,
where a is the activation function.
w_1
w_2
x_1
x_2
h_1
h_2
h_3
y_1
y_2

Figure 5: An activation function applied to a particular node.

Activation functions are examples of *hyperparameters*, parameters which are not learned by the system but rather chosen for the specific problem at hand. Other examples of hyperparameters are the number of neural layers, the number of neurons in each layer, the cost function, and, most importantly, the learning rate (we will define cost functions and the learning rate in the following sections).

Though there exist dozens of different activation function configurations which each have their own domain of application, we will focus on one of the most common for our purposes.

The *logistic sigmoid* $\sigma : \mathbb{R} \to (0, 1)$ is the main source of nonlinearity within our neural network and is defined by

$$\sigma(z) := \frac{1}{1 + \exp(-z)}$$

for all $z \in \mathbb{R}$. This function is particularly prevalent within neural training because of its rough representation of a biological neuron's firing pattern.

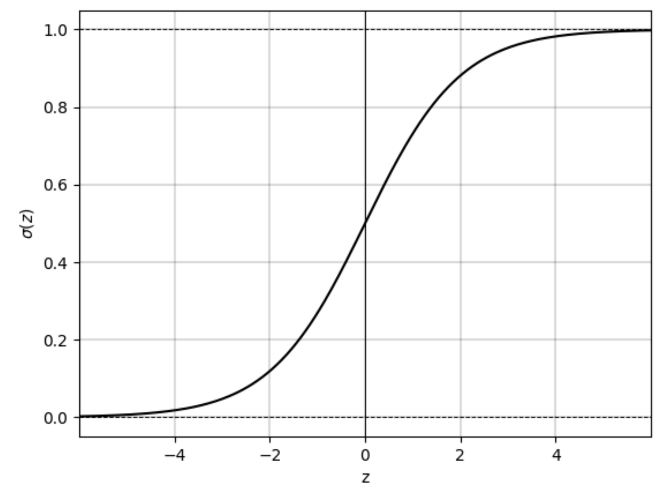


Figure 6: A plot of the logistic sigmoid.

4.2 Vectorization

Although it can be helpful to think of neural networks as layers of individual nodes with values being passed from layer to layer, we will introduce new notation in order to allow for more mathematical analysis. Although the notation is a bit cumbersome at first, it will allow for a deeper evaluation of these systems.

First, the weights: Denote $\boldsymbol{w}^\ell$ for $\ell > 1$ as the matrix containing the weights that connect the $(\ell - 1)$-th layer and the ℓ-th layer. This matrix has entries such that w_{ij}^ℓ is the value of the weight from neuron j in layer $\ell - 1$ to neuron i in layer ℓ.

For the biases, we similarly denote $\boldsymbol{b}^\ell$ for $\ell > 1$ as the vector containing the biases for the ℓ-th layer, where b_i^ℓ is the bias for neuron i in layer ℓ.

Finally, we write $\boldsymbol{a}^\ell$ for $\ell > 1$ as the vector containing the outputs after the activation function is applied from the ℓ-th layer of neurons, where a_i^ℓ is the output from neuron i in layer ℓ. When $\ell = 1$, let $\boldsymbol{a}^1$ be the input into our system.

For clarity, the following diagram shows a few elements of the weight matrices and bias vectors:

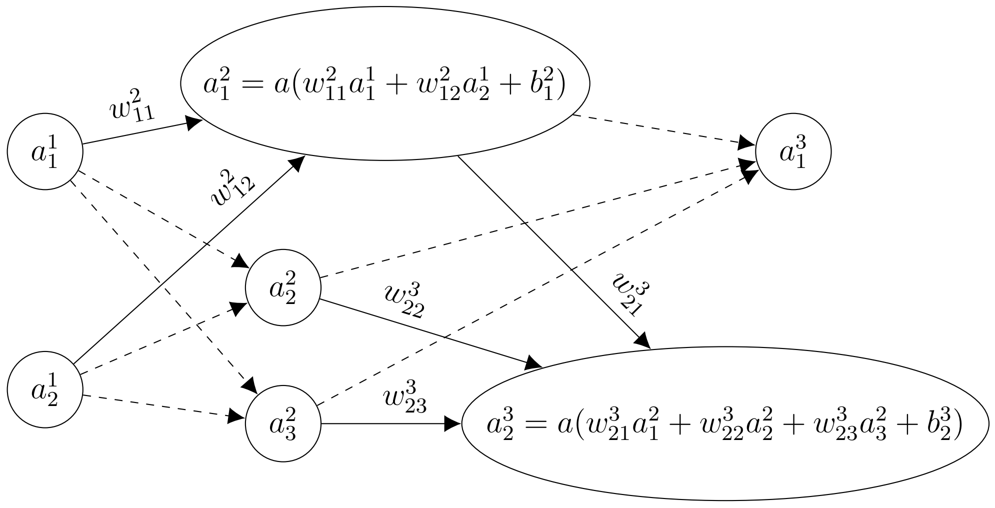

$a_1^2 = a(w_{11}^2 a_1^1 + w_{12}^2 a_2^1 + b_1^2)$
w_{11}^2
a_1^1
w_{12}^2
a_1^3
a_2^2
w_{22}^3
w_{21}^3
a_2^1
a_3^2
w_{23}^3
$a_2^3 = a(w_{21}^3 a_1^2 + w_{22}^3 a_2^2 + w_{23}^3 a_3^2 + b_2^3)$

Figure 7: A few elements of vectorized weights and biases.

With this notation, we can write the essential equation

$$\boldsymbol{a}^\ell = a(\boldsymbol{w}^\ell \boldsymbol{a}^{\ell-1} + \boldsymbol{b}^\ell)$$

for $\ell > 1$, where a is our activation function. Note, however, that the quantity $\boldsymbol{w}^\ell \boldsymbol{a}^{\ell-1} + \boldsymbol{b}^\ell$ is a vector. So, if a is a scalar-to-scalar function as it was with the logistic sigmoid, then we simply apply a elementwise to its input.

Now, this equation is vital in two different ways: First, it allows us to view a neural network as a sequence of activation vectors $\boldsymbol{a}^\ell$, rather than as a complex network of weights and biases. Second, it gives us a recursive formula for computing this sequence of vectors. And so, the above equation is often referred to as the *feedforward algorithm* because it allows us to feed the input vector $\boldsymbol{a}^1$ into the system and return an output vector $\boldsymbol{a}^L$.

4.3 Cost Functions

The last section showed us how to compute the neural ouput given a sequence of weight matrices and bias vectors. However, we still need to choose these weights and biases such that the system models our ideal function. To do this, we will define the notion of a *cost function*:

Suppose $\mathbb{R}^n$ is the set of all possible parameters of our model. A cost function $C : \mathbb{R}^n \to \mathbb{R}$ is a measure of the "badness" of a particular set of parameters. That is, a cost function is a hyperparameter that represents how close $\boldsymbol{x} \in \mathbb{R}^n$ is to the underlying function γ^* that the data was generated from.

To make this idea more concrete, we can consider the problem of linear regression. In this example, $n = 2$ because every linear model is defined by a pair of reals, the slope and the intercept. The most common cost function is that used in least squares regression,

$$C(m,b) := \frac{1}{n}\sum_i((mx_i + b) - y_i)^2,$$

where the sum is taken over all (x_i, y_i) in the dataset. Note that the function

$$C'(m,b) := \frac{1}{n}\sum_i |(mx_i + b) - y_i|$$

is, in some sense, more "natural" of a choice for the cost function, yet it is rarely used. As we will see, it will become important to be able to compute the derivative of the cost function with respect to each parameter. Thus, we often hand-select C not only on the basis of approximation to γ^*, but also on the basis of mathematical convenience.

In the following diagram, we see a few elements $\boldsymbol{x}$ of $\mathbb{R}^2$ and their corresponding cost $C(\boldsymbol{x})$.

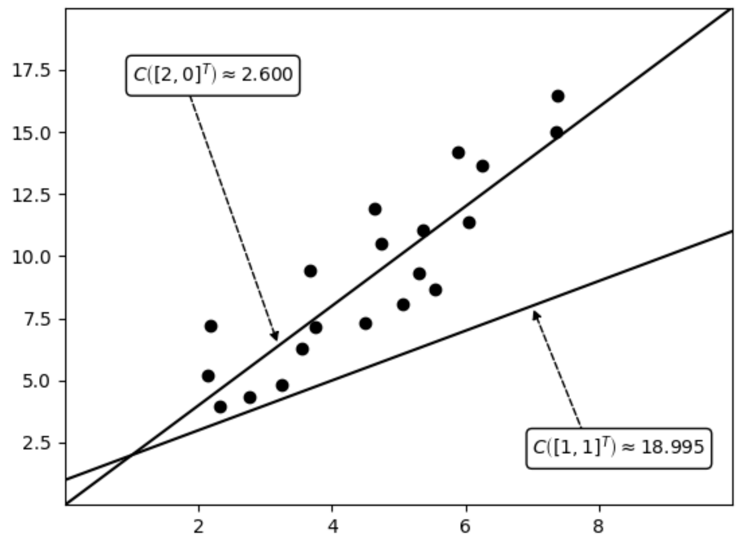


Figure 8: Two linear models and their corresponding cost functions.

So, our goal is to minimize $C(\boldsymbol{x})$. In the case that $\boldsymbol{x}$ is single dimensional, we know that the derivative of C with respect to its input x tells us how to move x in order to decrease C. That is,

$$C(x_0 - \eta \cdot C'(x_0)) < C(x_0)$$

for sufficiently small $\eta > 0$, assuming that $C'(x_0) \neq 0$.

However, our cost functions rarely act on single-dimensional inputs. To generalize to higher dimensions, we note that we would like to move our input vector $\boldsymbol{x} \in \mathbb{R}^n$ in the direction in which $C(\boldsymbol{x})$ decreases the fastest. Denote

$$\mathcal{U} := \{\boldsymbol{u} \in \mathbb{R}^n : ||\boldsymbol{u}|| = 1\}$$

as the set of all possible unit vector directions in which our input can move. We seek to minimize the derivative of C in the direction of $u \in \mathcal{U}$, which is given by

$$\frac{\mathrm{d}}{\mathrm{d}\alpha} C(\boldsymbol{x} + \alpha \boldsymbol{u})\Big|_{\alpha=0} .$$

Using the generalized chain rule and the fact that $\boldsymbol{a} \cdot \boldsymbol{b} = ||\boldsymbol{a}|| \cdot ||\boldsymbol{b}|| \cdot \cos\theta$, we have that

$$\begin{aligned}
\frac{\mathrm{d}}{\mathrm{d}\alpha} C(\boldsymbol{x} + \alpha \boldsymbol{u}) \Big|_{\alpha=0} &= \boldsymbol{u} \cdot \nabla C(\boldsymbol{x} + \alpha \boldsymbol{u}) \Big|_{\alpha=0} \\
&= \boldsymbol{u} \cdot \nabla C(\boldsymbol{x}) \\
&= ||\boldsymbol{u}|| \cdot ||\nabla C(\boldsymbol{x})|| \cdot \cos\theta \\
&= ||\nabla C(\boldsymbol{x})|| \cdot \cos\theta.
\end{aligned}$$

Since $||\nabla C(\boldsymbol{x})||$ does not depend on the choice of $\boldsymbol{u}$, we have that our directional derivative is minimized when $\theta = \pi$, or when $\boldsymbol{u}$ points in the opposite direction of $\nabla C(\boldsymbol{x})$. Thus, we have that our cost function $C(\boldsymbol{x})$ is minimized the fastest when $\boldsymbol{x}$ is moved in the direction of $-\nabla C(\boldsymbol{x})$.

This proposed method of minimization is known as *gradient descent* and is the most common method of minimizing cost functions. As in the single-dimensional case, the method of gradient descent is based on the fact that

$$C(\boldsymbol{x} - \eta \nabla C(\boldsymbol{x})) < C(\boldsymbol{x})$$

for sufficiently small $\eta > 0$.

4.4 Backpropagation

In the past two sections, we have seen both how to compute the output of a neural network given its weights and biases and how we define a "good" set of parameters for our model. In this section, we will present the final idea necessary to construct a good model; we will explore how we actually adjust our weights and biases so that we reduce our cost function C.

To proceed, we need two assumptions regarding our cost function C. First, the function must be able to be written as a sum

$$C = \frac{1}{n} \sum_{\boldsymbol{x} \in \mathcal{D}} C_{\boldsymbol{x}}$$

of individual cost functions over the elements in our dataset $\mathcal{D}$. This assumption is necessary because it allows us to consider each data point in isolation as we evaluate how to change our weights and biases.

Second, we must assume that C is a function of the output vector $\boldsymbol{a}^L$ only. That is, C does not consider the input vector or any of the hidden layers' vectors as it computes the cost of a particular training example. This makes sense, as neural networks ought to function as "black box" function approximators.

Our goal in this section is to find an explicit formula that tells us how to adjust our weights and biases in order to reduce the value of our cost function. This means that we will need to compute the quantities $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ for each weight w and each bias b in our network in order to adjust each of w and b accordingly.

In order to compute these partial derivatives we introduce an intermediate quantity know as the *neural error*. We will find a formula for the neural error given the outputs from each layer, and then we will relate the neural error to the values of $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$ for each weight w and bias b.

First, define the *weighted sum* vector $\boldsymbol{z}^\ell$ such that entry z_i^ℓ is the weighted sum at neuron i of layer ℓ before the activation function is applied. That is,

$$\boldsymbol{z}^\ell = \boldsymbol{w}^\ell \boldsymbol{a}^{\ell-1} + \boldsymbol{b}^\ell$$

for all layers $\ell > 1$.

Now, for each layer $\ell > 1$ in the network, let the *neural error* vector δ^ℓ satisfy

$$\delta_i^\ell = \frac{\partial C}{\partial z_i^\ell}.$$

Since C most directly depends on the output layer from our network, we begin by finding a formula for δ^L. We have that

$$\begin{aligned}\delta_i^L = \frac{\partial C}{\partial z_i^L} &= \sum_{1 \le j \le n} \frac{\partial C}{\partial a_j^L} \cdot \frac{\partial a_j^L}{\partial z_i^L} \\ &= \frac{\partial C}{\partial a_i^L} \cdot \frac{\partial a_i^L}{\partial z_i^L} \\ &= \frac{\partial C}{\partial a_i^L} \cdot a'(z_i^L),\end{aligned}$$

where the second equality was achieved by the generalized chain rule and the third equality was achieved by the fact that a_j^L does not depend on z_i^L for all $j \neq i$.

Note that, just as we did for the weights and biases, it will be computationally helpful to write a vectorized form of the above equation. If $\nabla_{\boldsymbol{a}^L} C$ denotes the vector of partial derivatives of C with respect to the entries of $\boldsymbol{a}^L$, then the above equation becomes

$$\delta^L = \nabla_{\boldsymbol{a}^L} C \odot a'(\boldsymbol{z}^L)$$

where $\odot$ denotes the Hadamard product, the entrywise multiplication of two vectors.

Note that all of the quantities in this formula for δ^L are easily computable after the feedforward algorithm: We have already computed $\boldsymbol{z}^L$, so the quantity $a'(\boldsymbol{z}^L)$ is easy to compute. Similarly, we know an explicit formula for C in terms of the output activations $\boldsymbol{a}^L$ so the vector $\nabla_{\boldsymbol{a}^L} C$ falls out easily.

So, we have derived a formula for the neural error in the last layer L. We now introduce a recursive formula to find δ^ℓ in terms of $\delta^{\ell+1}$:

As seen previously, we have that

$$\delta_i^\ell = \frac{\partial C}{\partial z_i^\ell} = \frac{\partial C}{\partial a_i^\ell} \cdot a'(z_i^\ell).$$

Using the chain rule, we have that

$$\begin{aligned}\frac{\partial C}{\partial a_i^\ell} \cdot a'(z_i^\ell) &= a'(z_i^\ell) \cdot \sum_j \frac{\partial C}{\partial z_j^{\ell+1}} \cdot \frac{\partial z_j^{\ell+1}}{\partial a_i^\ell} \\ &= a'(z_i^\ell) \cdot \sum_j \delta_j^{\ell+1} \cdot w_{ji}^{\ell+1} \\ &= a'(z_i^\ell) \cdot \left(w_{*,i}^{\ell+1} \cdot \delta^{\ell+1}\right),\end{aligned}$$

where $w_{*,i}^{\ell+1}$ denotes the i-th column of the $(\ell+1)$-th weight matrix. As before, we can write this new equation in vector form to yield that

$$\delta^\ell = \left((\boldsymbol{w}^{\ell+1})^T \delta^{\ell+1}\right) \odot a'(\boldsymbol{z}^\ell).$$

Combining this recursion with the initial formula for δ^L, we can find the neural error in each layer. Now, we will relate the neural error δ^ℓ to our desired values of $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$.

Expanding using the chain rule, we get that

$$\begin{aligned}\frac{\partial C}{\partial w_{ij}^\ell} &= \frac{\partial C}{\partial z_i^\ell} \cdot \frac{\partial z_i^\ell}{\partial w_{ij}^\ell} \\ &= \delta_i^\ell \cdot a_j^{\ell-1},\end{aligned}$$

where $\frac{\partial z_i^\ell}{\partial w_{ij}^\ell}$ equals $a_j^{\ell-1}$ because $z_i^\ell = w_{i,*}^\ell a^{\ell-1} + b_i^\ell$. So, we have found our desired formula for $\frac{\partial C}{\partial w}$.

Now, for $\frac{\partial C}{\partial b}$:

$$\begin{aligned}\frac{\partial C}{\partial b_i^\ell} &= \frac{\partial C}{\partial z_i^\ell} \cdot \frac{\partial z_i^\ell}{\partial b_i^\ell} \\ &= \frac{\partial C}{\partial z_i^\ell} \cdot 1 \\ &= \delta_i^\ell.\end{aligned}$$

Thus, we have the four following fundamental equations:

$$\boxed{\begin{array}{ll} \delta^L = \nabla_{\boldsymbol{a}^L} C \odot a'(\boldsymbol{z}^L) & \frac{\partial C}{\partial w_{ij}^\ell} = \delta_i^\ell \cdot a_j^{\ell-1} \\ \delta^\ell = \left((\boldsymbol{w}^{\ell+1})^T \delta^{\ell+1}\right) \odot a'(\boldsymbol{z}^\ell) & \frac{\partial C}{\partial b_i^\ell} = \delta_i^\ell \end{array}}$$

4.5 Completing the Learning Algorithm

Now that we have seen both the structure of neural networks and essential relationships regarding updating weights and biases, we are ready to create formal algorithms that can be practically implemented.

Using the ideas covered in the Vectorization section, we can construct a feedforward algorithm that feeds an input vector through the neural network:

Algorithm 1 Feedforward Algorithm

Input: The input vector $\boldsymbol{a}^1$.

Output: The output vector $\boldsymbol{a}^L$.

```
1: function FEEDFORWARD(inputVector)
2:     currentVector ← inputVector
3:     while hasNextLayer do
4:         zVector ← currentLayerWeights · currentVector + currentLayerBiases
5:         currentVector ← activationFunction(zVector)
6:     return currentVector
```

Next, we can construct a backpropagation algorithm that will take a point $(\boldsymbol{x}, \boldsymbol{y})$ from our dataset and tell us how to change our weights and biases in order to reduce the cost function $C_{\boldsymbol{x}}$ for that particular data point. This is done by returning a pair $(\nabla B, \nabla W)$. ∇B consists of a list of vectors such that the i-th element of the ℓ-th vector is $\frac{\partial C_{\boldsymbol{x}}}{\partial b_i^\ell}$. Similarly, ∇W contains a list of matrices such that ij-th element of the ℓ-th matrix is $\frac{\partial C_{\boldsymbol{x}}}{\partial w_{ij}^\ell}$.

Algorithm 2 Backpropagation Algorithm

Input: The input vector $\boldsymbol{a}^1$ and its corresponding desired output $\boldsymbol{a}^L$.
Output: The pair $(\nabla B, \nabla W)$.

```
function BACKPROP(x, y)
    currentActivation $\leftarrow$ x
    activationsList $\leftarrow$ [x]
    zsList $\leftarrow$ [ ]
    while hasNextLayer do
        z $\leftarrow$ layerWeights $\cdot$ currentActivation + layerBiases
        append z to zsList
        currentActivation $\leftarrow$ activationFunction(z)
        append currentActivation to activationsList
    $\delta \leftarrow$ costDerivative(activationsList[-1], y) $\odot$ activationDerivative(z)
    nablaB[-1] $\leftarrow \delta$
    nablaW[-1] $\leftarrow \delta \cdot$ transpose(activationsList[-2])
    $\ell \leftarrow 2$
    while $\ell <$ numLayers do
        z $\leftarrow$ zsList[-$\ell$]
        $\delta \leftarrow$ (tranpose(weights[-$\ell$ + 1]) $\cdot$ $\delta$) $\odot$ activationDerivative(z)
        nablaB[-$\ell$] $\leftarrow \delta$
        nablaW[-$\ell$] $\leftarrow \delta \cdot$ transpose(activationsList[-$\ell$ - 1])
        $\ell \leftarrow \ell + 1$
    return (nablaB, nablaW)
```

The function activationDerivative($\boldsymbol{z}$) should take the vector $\boldsymbol{z}$ and apply the derivative of the activation function, a', to each of its elements. Also, the function costDerivative(activation, y) should return a vector of partial derivatives of $C_{\boldsymbol{x}}$ with respect to each element of activation. That is,

$$(\text{costDerivative}(\boldsymbol{a}^L, \boldsymbol{y}))_i = \frac{\partial C_{\boldsymbol{x}}}{\partial a_i^L}.$$

For example, if we were to use the cost function

$$C = \frac{1}{n} \sum_{\boldsymbol{x}} C_{\boldsymbol{x}}$$

where

$$C_{\boldsymbol{x}} = ||\boldsymbol{a}^L - \boldsymbol{y}||^2,$$

then we would have that

$$(\text{costDerivative}(\boldsymbol{a}^L, \boldsymbol{y}))_i = 2(a_i^L - y_i).$$

Now that we have feedforward and backpropagation algorithms, we will create a final algorithm to house our main logic. The algorithm will take the training dataset, run gradient descent, and update the weights and biases.

Algorithm 3 Controller

```
Input: A list of training pairs $(x, y)$, the total number of epochs, the mini-batch
    size, and the learning rate.
 1: function MAIN(trainingData, epochs, miniBatchSize, learningRate)
 2:     epoch $\leftarrow$ 1
 3:     while epoch $\leq$ epochs do
 4:         randomly shuffle trainingData
 5:         while hasNextMiniBatch do
 6:             for all $(x, y)$ in miniBatch do
 7:                 (nablaB, nablaW) = (nablaB, nablaW) + backprop(x,y)
 8:             while hasNextLayer do
 9:                 layerBiases = layerBiases - $\frac{\text{learningRate}}{\text{miniBatchSize}}\cdot$ layerNablaB
10:                 layerWeights = layerWeights - $\frac{\text{learningRate}}{\text{miniBatchSize}}\cdot$ layerNablaW
11:         epoch $\leftarrow$ epoch + 1
```

We will now describe the intuition behind the other three arguments of our controller algorithm: epochs, miniBatchSize, and learningRate.

We previously mentioned that, since the cost function C can be written as $C = \frac{1}{n}\sum_{\boldsymbol{x}} C_x$, we can write

$$\frac{\partial C}{\partial w} = \frac{1}{n}\sum_{\boldsymbol{x}} \frac{\partial C_{\boldsymbol{x}}}{\partial w}$$

and

$$\frac{\partial C}{\partial b} = \frac{1}{n}\sum_{\boldsymbol{x}} \frac{\partial C_{\boldsymbol{x}}}{\partial b}$$

for each weight w and bias b. Since we oftentimes have tens of thousands of training examples, however, we do not need to sum over all inputs $\boldsymbol{x}$ in order to get a pretty good idea of the values of $\frac{\partial C}{\partial w}$ and $\frac{\partial C}{\partial b}$. Instead, we can pick a few random examples and sum over those values of $\frac{\partial C_{\boldsymbol{x}}}{\partial w}$ and $\frac{\partial C_{\boldsymbol{x}}}{\partial b}$. Since random choice is involved, this augmented method is known as *stochastic gradient descent*. The variable "miniBatchSize" will control the size of this smaller sample.

Next, the variable "epochs" controls the total number of passes that will be made over the training data. Finally, "learningRate" controls how far each weight and bias will move once a gradient is calculated. Larger learning rates results in faster learning, but too high of a learning rate can result in erratic behavior as our weights and biases jump over minima.

Using only these three algorithms, we can train and test our neural network to fit any arbitrary dataset. As we will see, our neural network will be able to learn relatively complex patterns with a fair degree of accuracy.

4.6 A Proof of Gradient Descent Convergence

Just as we did for perceptrons, we will now show that our method of gradient descent will indeed converge to the minimum value. We will present a proof for the non-stochastic version of the gradient descent algorithm, though similar arguments can be used to prove convergence with stochasticity.

In order to prove that gradient descent will converge to the minimum, we need three assumptions about our cost function $C(\boldsymbol{x})$:

First, we must assume that ∇C is *Lipschitz continuous*. That is, there must exist some constant $L \in \mathbb{R}$ such that

$$||\nabla C(\boldsymbol{x}) - \nabla C(\boldsymbol{y})|| \leq L \cdot ||\boldsymbol{x} - \boldsymbol{y}||$$

for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$. In essence, this definition is saying that the gradient of C cannot change wildly.

Second, C must both be strictly convex and have a unique global minimum $\boldsymbol{x}^*$.

Finally, we must have that our learning rate η satisfy $0 < \eta \leq \frac{1}{L}$. As we will see, these three conditions are enough to specify a cost function and learning rate that are well-behaved enough to guarantee convergence.

First, a lemma:

Lemma. *If ∇f is Lipschitz continuous with constant L, then*

$$|f(\boldsymbol{y}) - f(\boldsymbol{x}) - \nabla f(\boldsymbol{x}) \cdot (\boldsymbol{y} - \boldsymbol{x})| \leq \frac{L}{2} \cdot ||\boldsymbol{y} - \boldsymbol{x}||^2$$

for all $\boldsymbol{x}, \boldsymbol{y} \in \mathbb{R}^n$.

Proof. Let g satisfy

$$g(t) := f\left(\boldsymbol{x} + t \cdot (\boldsymbol{y} - \boldsymbol{x})\right)$$

for all $t \in \mathbb{R}$. Note that $g(0) = f(\boldsymbol{x})$ and $g(1) = f(\boldsymbol{y})$. We then have that

$$\begin{aligned}
|f(\boldsymbol{y}) - &f(\boldsymbol{x}) - \nabla f(\boldsymbol{x}) \cdot (\boldsymbol{y} - \boldsymbol{x})| \\
&= |g(1) - g(0) - \nabla f(\boldsymbol{x}) \cdot (\boldsymbol{y} - \boldsymbol{x})| \\
&= \left| \int_0^1 g'(t)\ dt - \nabla f(\boldsymbol{x}) \cdot (\boldsymbol{y} - \boldsymbol{x}) \right| \\
&\leq \int_0^1 |g'(t) - \nabla f(\boldsymbol{x}) \cdot (\boldsymbol{y} - \boldsymbol{x})|\ dt \\
&= \int_0^1 |\nabla f\left(\boldsymbol{x} + t \cdot (\boldsymbol{y} - \boldsymbol{x})\right) \cdot (\boldsymbol{y} - \boldsymbol{x}) - \nabla f(\boldsymbol{x}) \cdot (\boldsymbol{y} - \boldsymbol{x})|\ dt
\end{aligned}$$

By Cauchy-Schwarz and the supposition that ∇f is Lipschitz continuous,

$$\begin{aligned}
&\leq ||\boldsymbol{y} - \boldsymbol{x}|| \cdot \int_0^1 ||\nabla f\left(\boldsymbol{x} + t \cdot (\boldsymbol{y} - \boldsymbol{x})\right) - \nabla f(\boldsymbol{x})||\ dt \\
&\leq ||\boldsymbol{y} - \boldsymbol{x}|| \cdot \int_0^1 ||L \cdot t \cdot (\boldsymbol{y} - \boldsymbol{x})||\ dt \\
&= L \cdot ||\boldsymbol{y} - \boldsymbol{x}||^2 \cdot \int_0^1 t\ dt \\
&= \frac{L}{2} \cdot ||\boldsymbol{y} - \boldsymbol{x}||^2
\end{aligned}$$

as desired.

$\square$

We can use this lemma on ∇C with $\boldsymbol{y} = \boldsymbol{x} - \eta \nabla f(\boldsymbol{x})$ to get that

$$\begin{aligned}
f(\boldsymbol{y}) &\leq f(\boldsymbol{x}) + \nabla f(\boldsymbol{x}) \cdot (-\eta \nabla f(\boldsymbol{x})) + \frac{L}{2} \cdot || - \eta \nabla f(\boldsymbol{x})||^2 \\
&= f(\boldsymbol{x}) - \eta \cdot \left(1 - \frac{\eta L}{2}\right) ||\nabla f(\boldsymbol{x})||^2
\end{aligned}$$

From the supposition that $0 < \eta \leq \frac{1}{L}$, we have that

$$f(\boldsymbol{y}) \leq f(\boldsymbol{x}) - \frac{\eta}{2} \cdot ||\nabla f(\boldsymbol{x})||^2$$

Now, since C is convex, we have that $f(\boldsymbol{b}) \geq f(\boldsymbol{a}) + \nabla f(\boldsymbol{a}) \cdot (\boldsymbol{b} - \boldsymbol{a})$ for all $\boldsymbol{a}, \boldsymbol{b}$. (That is, the first-order Taylor approximation is always an underestimate.)

Applying this observation for $\boldsymbol{a} = \boldsymbol{x}$ and $\boldsymbol{b} = \boldsymbol{x}^*$, where $\boldsymbol{x}^*$ is the global minimum, we have that

$$\begin{aligned} f(\boldsymbol{y}) &\leq f(\boldsymbol{x}^*) + \nabla f(\boldsymbol{x}) \cdot (\boldsymbol{x} - \boldsymbol{x}^*) - \frac{\eta}{2} \cdot ||\nabla f(\boldsymbol{x})||^2 \\ &= f(\boldsymbol{x}^*) + \frac{1}{2\eta}\left(||\boldsymbol{x} - \boldsymbol{x}^*||^2 - ||(\boldsymbol{x} - \eta \nabla f(\boldsymbol{x})) - \boldsymbol{x}^*||^2\right). \end{aligned}$$

If we write our sequence of approximations as $\boldsymbol{x}^0, \boldsymbol{x}^1, ...$ such that

$$\boldsymbol{x}^k = \boldsymbol{x}^{k-1} - \eta \nabla f(\boldsymbol{x}^{k-1})$$

for all $k \geq 1$, we can rewrite the above observation as

$$f(\boldsymbol{x}^k) \leq f(\boldsymbol{x}^*) + \frac{1}{2\eta}\left(||\boldsymbol{x}^{k-1} - \boldsymbol{x}^*||^2 - ||\boldsymbol{x}^k - \boldsymbol{x}^*||^2\right).$$

Summing over all such k and noticing that the sum telescopes, we get that

$$\begin{aligned} \sum_{k=1}^{K} \left(f(\boldsymbol{x}^k) - f(\boldsymbol{x}^*)\right) &\leq \frac{1}{2\eta} \sum_{k=1}^{K} \left(||\boldsymbol{x}^{k-1} - \boldsymbol{x}^*||^2 - ||\boldsymbol{x}^k - \boldsymbol{x}^*||^2\right) \\ &= \frac{1}{2\eta}\left(||\boldsymbol{x}^0 - \boldsymbol{x}^*||^2 - ||\boldsymbol{x}^K - \boldsymbol{x}^*||^2\right) \\ &\leq \frac{||\boldsymbol{x}^0 - \boldsymbol{x}^*||^2}{2\eta}. \end{aligned}$$

Finally, since $\boldsymbol{x}^0, \boldsymbol{x}^1, ...$ is a nonincreasing sequence, we have that

$$f(\boldsymbol{x}^K) - f(\boldsymbol{x}^*) \leq f(\boldsymbol{x}^k) - f(\boldsymbol{x}^*)$$

for all $K \geq k$. Thus, we get that

$$f(\boldsymbol{x}^K) - f(\boldsymbol{x}^*) \leq \frac{1}{K} \sum_{k=1}^{K} \left(f(\boldsymbol{x}^k) - f(\boldsymbol{x}^*)\right).$$

So, by our prior result, we conclude that

$$f(\boldsymbol{x}^K) - f(\boldsymbol{x}^*) \leq \frac{||\boldsymbol{x}^0 - \boldsymbol{x}^*||^2}{2K\eta},$$

showing that our sequence does indeed approach $\boldsymbol{x}^*$, as desired.

□

5 Comparisons

5.1 Overview

In this section, we will explore the practical implementations of both the perceptron and the multilayer neural network models. First, we define our dataset to learn:

The MNIST dataset consists of 65,000 handwritten digits and their corresponding labels. The 28×28 grayscale images were collected by the National Institute of Standards and Technology and were modified such that the center of mass of each digit lies at the center of the image. The digits are represented as vectors of size $28^2 = 784$ of values between 0 and 1, where 0 represents pure white and 1 represents pure black.

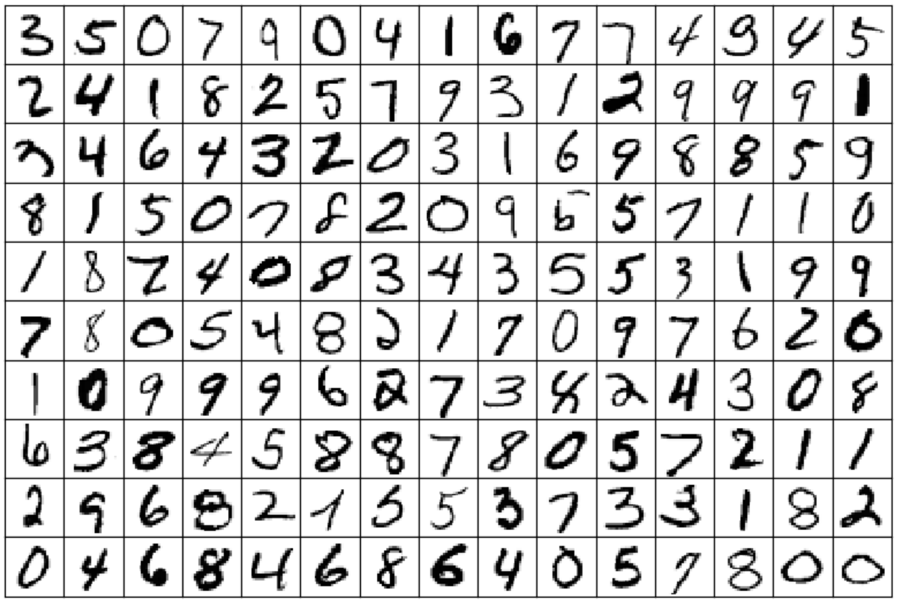


Figure 9: Some samples from the MNIST dataset.

We will split the dataset into a training set of size 55,000 and a test set of size 10,000. Our models will learn on the training set and then will be evaluated on the test set. This split ensures that our models are actually learning how to evaluate digits and not just recognizing the digits that they have already seen.

Handwritten digit recognition is a well-explored problem within machine learning, so algorithms have been developed that can recognize digits with a less than a 0.3% error rate, better than human levels of error. The current best of 0.23% is considered near-ideal, as many digits in the MNIST dataset are difficult for even humans to evaluate correctly.

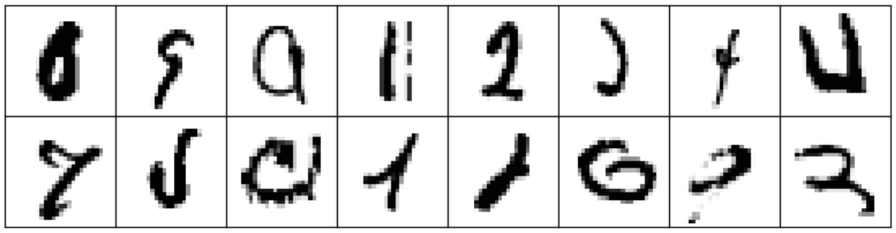


Figure 10: A few MNIST examples that are difficult to classify.

5.2 Perceptron

In this section, we will implement a perceptron learning algorithm that will classify MNIST digits.

First, we need to slightly modify the algorithm previously introduced. We noted in our proof of convergence that our dataset must be linearly separable in order for perceptron learning to terminate. Since our dataset is almost surely not linearly separable, we introduce the following modified perceptron algorithm:

Algorithm 4 Pocket Perceptron Algorithm

Input: A list of training pairs (x, y) and the total number of epochs.

```
1: function POCKETPERCEPTRON(trainingData, epochs)
2:     initialize weightVector with random values
3:     bestWeight ← weightVector
4:     bestNumCorrect ← 0
5:     epoch ← 1
6:     while epoch ≤ epochs do
7:         for all (x, y) in trainingData do
8:             predictedSign ← sign(weightVector · x)
9:             if predictedSign ≠ actualSign then
10:                weightVector ← weightVector + actualSign · x
11:                numCorrect ← 0
12:                for all (xPrime, yPrime) in trainingData do
13:                    if sign(weightVector · xPrime) = actualSign then
14:                        numCorrect ← numCorrect + 1
15:                if numCorrect > bestNumCorrect then
16:                    bestNumCorrect ← numCorrect
17:                    bestWeight ← weightVector
18:        epoch ← epoch + 1
19:    return bestWeight
```

This algorithm is the same as the perceptron algorithm introduced earlier, except this "Pocket Perceptron" algorithm keeps the best weight vector in its 'pocket' and returns the best weight that it has seen after a certain number of epochs.

If we run this algorithm with 50 epochs in Python to learn a classifier for all $\binom{10}{2} = 45$ pairs of digits, we get the following accuracy rates:

Pair	Accuracy (%)	Pair	Accuracy (%)	Pair	Accuracy (%)
(0, 1)	99.9	(1, 8)	97.7	(4, 5)	98.0
(0, 2)	96.1	(1, 9)	99.5	(4, 6)	97.8
(0, 3)	97.7	(2, 3)	81.0	(4, 7)	97.8
(0, 4)	99.7	(2, 4)	97.4	(4, 8)	98.9
(0, 5)	97.9	(2, 5)	96.4	(4, 9)	64.6
(0, 6)	97.1	(2, 6)	97.0	(5, 6)	93.1
(0, 7)	98.8	(2, 7)	96.3	(5, 7)	99.0
(0, 8)	94.3	(2, 8)	91.1	(5, 8)	76.8
(0, 9)	95.9	(2, 9)	93.4	(5, 9)	94.3
(1, 2)	98.8	(3, 4)	99.4	(6, 7)	99.3
(1, 3)	99.3	(3, 5)	91.5	(6, 8)	92.2
(1, 4)	99.8	(3, 6)	97.2	(6, 9)	99.4
(1, 5)	92.9	(3, 7)	92.7	(7, 8)	94.7
(1, 6)	99.4	(3, 8)	90.1	(7, 9)	84.3
(1, 7)	96.3	(3, 9)	94.8	(8, 9)	91.8

Figure 11: Accuracy rates for all pairs of binary classifiers.

As we would expect, the perceptron can classify dissimilar digits well, while similar digits prove to be harder. For example, the $(0, 1)$ classifier has an accuracy rate of 99.9%, while the $(4, 9)$ classifier has a rate of 64.6%.

For a visual interpretation of what the perceptron is doing, we can look at the weight vector $\boldsymbol{w}$ learned by the $(0, 1)$ classifier. In the following figure, we see two representations of $\boldsymbol{w}$. The first heat map consists of the positive entries of $\boldsymbol{w}$, while the second only shows the negative entries. In each map, darker colors are given to entries with greater absolute value.

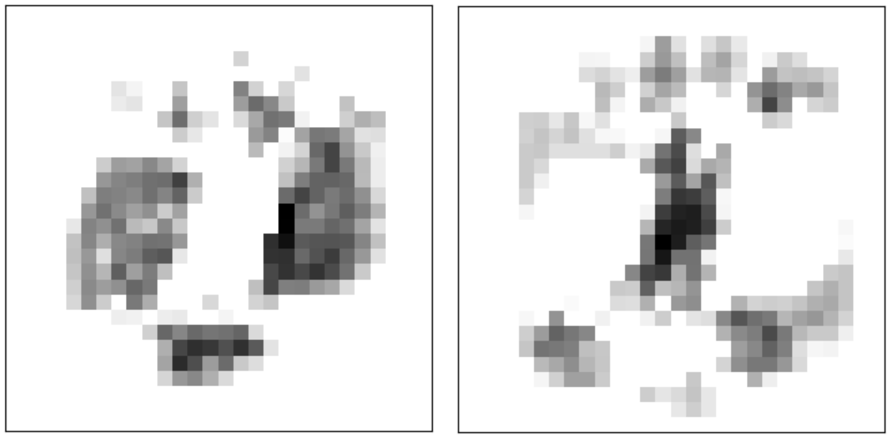


Figure 12: Two heat maps of $\boldsymbol{w}$ for the $(0, 1)$ binary classifier.

As we would expect, our $(0, 1)$ perceptron has learned that pixels shaded around the center of the image usually denote a '0' and pixels shaded in the center of the image correspond to a '1'.

In order to compare perceptron learning and multilayer learning, we need to modify our perceptron algorithm such that it can take a single input image and return its predicted 0 to 9 digit value. To do this, we consider a "tally vector" $\boldsymbol{t}$ of size 10. We initialize $\boldsymbol{t}$ with entries of 0. For each pair (a, b) of digits, we run our image through the perceptron corresponding to the pair (a, b). If the perceptron predicts that the image is of type a, we add 1 to the a-th entry in $\boldsymbol{t}$ and -1 to the b-th entry in $\boldsymbol{t}$. Otherwise, if the perceptron predicts b, we add 1 to the b-th entry and -1 to the a-th entry. This allows us to tally up each perceptron's prediction to create an overall estimate.

Using this method of perceptron tallying, we arrive at an overall accuracy rate of 77.60%. Considering that most of the individual perceptrons have accuracy rates over 95%, this low overall accuracy may be surprising. This reveals a fundamental issue with perceptrons: Though they can be useful for binary classification, there is no easy or intuitive way to generalize perceptron learning to multi-class problems.

5.3 Multilayer Network

Now, we will take a look at a practical implementation of a multilayer system. Using the feedforward, backpropagation, and controller algorithms introduced in the previous section, we can implement a neural network in Python. Our particular network will contain $28^2 = 784$ input neurons, 50 neurons in a single hidden layer, and 10 output neurons corresponding to our ten possible digit labels.

If we run stochastic gradient descent on this network for 50 epochs with a mini-batch size of 20 and a learning rate of 2.0, we get an accuracy rate of 95.61% on our 10,000 test images.

This improvement in accuracy over the Pocket Perceptron's 77.60% confirms our previous intuition that neural networks ought to be able to approximate more complex functions that either individual perceptrons or a composition of perceptrons.

6 Conclusion

Through our shift from single-neuron algorithms to multi-layer systems, we have seen an increase in complexity: Nonlinear functions are more complex than linear ones, multi-neuron and multi-layer systems are more complex than a single neuron, and multi-categorical classification is more complex than binary classification. And in turn, we have seen a corresponding increase in the approximating potential of our system; our multi-layer neural network improved classification accuracy by nearly 20%. It may be natural to ask, then, will more complexity on top of our current architecture improve our accuracy? Perhaps we could create three-dimensional neural networks or have non-static weights and biases.

Surprisingly, the answer is no. A high-level result within machine learning theory known as the *universal approximation theorem* states that any feedforward neural network with a single hidden layer of a sufficiently large number of neurons (exactly the type that we created) is able to approximate any continuous function on $\mathbb{R}^n$ arbitrarily well. (For more details, see [2].)

This result shows us that, though there exist infinite variations of learning algorithms to explore, we have established all of the structure necessary to solve any learning problem, from digit classification to human intelligence on the whole.

References

[1] Abu-Mostafa, Y. S., Magdon-Ismail, M., & Lin, H. (2012). *Learning from Data: A Short Course.* Pasadena, CA: AML Book.

[2] Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning.* Cambridge, MA: MIT Press.

[3] Gordon, G., & Tibshirani, R. (2012). *Gradient Descent Revisited* [Slides]. Retrieved from `http://www.cs.cmu.edu/~ggordon/10725-F12/slides/05-gd-revisited.pdf`.

[4] Nielsen, M. A. (2015). *Neural Networks and Deep Learning.* Determination Press.