Perceptron model

October 13, 2024

McCulloch-Pitts model and Perceptron

some historical account

Warren McCulloch and Walter Pitts in their 1943 paper, ”A Logical Calculus of Ideas Immanent in Nervous Activity,” proposed that neurons with a binary threshold activation function were analogous to first-order logic sentences. The McCulloch-Pitts neuron used binary inputs (1 for true, 0 for false) and produced a binary output based on a threshold value, typically 1.

A key issue with the McCulloch-Pitts neuron was its lack of a learning mechanism. In 1949, Donald Hebb introduced a synaptic learning rule in his book, The Organization of Behavior, known as Hebb's rule: “When an axon of cell A repeatedly or persistently helps fire cell B, some growth or metabolic change occurs in one or both cells, increasing A's efficiency in firing B.” Hebb suggested that simultaneous firing of two neurons strengthens their connection, a fundamental process for learning and memory.

Frank Rosenblatt, building on the McCulloch-Pitts neuron and Hebb's findings, developed the first perceptron, capable of learning by adjusting its weights incrementally. In his 1962 book, Principles of Neurodynamics, Rosenblatt claimed: “Give an elementary α perceptron, a stimulus world W, and any classification $C(W)$ with a solution; If all stimuli in W occur in any sequence and reoccur in finite time, an error correction procedure will always find a solution to $C(W)$ in finite time.”

Rosenblatt's claims created a divide between perceptron research and traditional symbol manipulation projects by Marvin Minsky. In 1969, Minsky and Seymour Papert co-authored Perceptrons: An Introduction to Computational Geometry, highlighting the perceptron's limitations, notably its inability to solve XOR and NXOR functions. They argued that perceptron research was doomed due to these limitations, leading to a decline in research until the 1980s.

Perceptron Decision Surface

The simplest network that implements interesting input-output function is the Perceptron. It has a single layer of weights connecting the inputs and output. Formally, the perceptron is defined by

$$y = \text{sign}\left(\sum_{i=1}^{N} W_i x_i - \theta\right) \tag{1}$$

In matrix form we have $y = \text{sign}(W \cdot X - \theta)$, where θ is the threshold parameter. Often, we ignore the threshold in perceptron analysis, treating it as another synaptic weight with a constant input of -1. This is because $W \cdot X - \theta = [W, \theta]^T [X, -1]$.

A perceptron performs a dichotomy, that is, a function from $\mathbb{R}^N$ to $\{-1, 1\}$ (or $\{0, 1\}$). The perceptron's action is visualized geometrically since W and x both have dimensionality N. The surface dividing points into two classes is a hyperplane defined by $W^T X = 0$, passing through the origin and perpendicular to W. With a non-zero threshold, the plane is $W^T X - \theta = 0$, perpendicular to W and separated from the origin by $\theta/|W|$. The plane is often referred to as W. The perceptron realizes linearly separable dichotomies, a small subset of all possible dichotomies.

The capacity of a perceptron

The capacity of a network is the number of functions (dichotomies) it can implement on a fixed set of inputs $\{X^1, ..., X^P\}$. For a perceptron, the architecture is defined by $y = \text{sign}(W^T X)$ where W is an N-dimensional vector. The set of possible dichotomies over $\{X^1, ..., X^P\}$ is $\{\text{sign}(W^T X^1), ..., \text{sign}(W^T X^P) | W \in \mathbb{R}^N\}$. The size of this set is the network's capacity. Generally, capacity depends on the specific inputs $\{x_1, ..., x_P\}$, but for a perceptron, it is independent of the input vectors' identity if they are in general position.

The general position means $\{X^1, ..., X^P\}$ does not have a subset of size N or less that is linearly dependent. If the inputs are in general position, the capacity depends only on the input dimension P and the number of inputs P, denoted as $C(P, N)$. Clearly, $C(P, N) \leq 2^P$ since the total number of dichotomies over P inputs is 2^P. Cover's Function Counting Theorem (1966) provides the exact value of $C(P, N)$:

$$C(P, N) = 2 \sum_{k=0}^{N-1} \binom{P-1}{k} \tag{2}$$

The binomial coefficient is defined as:

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

for $n \geq k$, otherwise $\binom{n}{k} = 0$.

If $P \leq N$, the vectors are generally in general position, and Cover's Theorem applies. For $P > N$, the vectors are linearly dependent, but Cover's Theorem still holds if they are in general position, which is typical for generic input vectors. Thus, Cover's theorem is broadly applicable.

Some consequences of Cover's theorem:

1. For $P \leq N$, the above sum is limited by $P1$, so it can be rewritten as

$$C(P, N) = 2 \sum_{k=0}^{P-1} \binom{P-1}{k} = 2(1+1)^{P-1} = 2^P$$

by using the familiar binomial expansion. In other words, all possible dichotomies can be realized.

2. for $P = 2N$,

$$C(2N, N) = 2 \sum_{k=0}^{N-1} \binom{2N-1}{k} = 2\frac{1}{2}(1+1)^{2N-1} = 2^{P-1}$$

This occurs because the expression for C includes precisely half of the complete binomial expansion of $(1+1)^{2N-1}$. Given that this expansion is symmetric (for instance, the initial few (odd) binomial expansions are (1, 1), (1, 3, 3, 1), and (1, 5, 10, 10, 5, 1)), we deduce that exactly half of all potential dichotomies can be achieved in this scenario.

3. For $P \gg N$, $C(P, N) \sim AP^N$ for some $A > 0$.

4. Another way to view $C(P, N)$ is to let P and N approach ∞ while keeping $N/P = \alpha$. As $N \to \infty$, $C(P, N)$ vs. N/P forms a step function. The step occurs at $N/P = 2$, where $C(2N, N) = 0.5$. Below $N/P = 2$, almost all dichotomies are possible; above it, almost none are possible.

Proof of Cover's Theorem

Perceptron learning Algorithm

Our goal is to train a perceptron to classify points. Given $\{X^1, ..., X^P\}$ and labels $\{y_0^1, ..., y_0^P\}$, we seek weights W such that $\text{sign}(W^T X^\mu) = y_0^\mu$ for all μ. We need an algorithm to find weights that separate the data.

The Perceptron learning algorithm updates incrementally, beginning with initial weights W_0. At iteration n, the weights W_{n-1} and the example (X^n, y_0^n) are utilized. The examples are shown sequentially: $1, 2, ..., P, 1, 2, ..., P, ...$.

If $y_0^n(W_{n-1}^T X^n) > 0$, the sample is classified correctly: assign $W_n = W_{n-1}$.

If $y_0^n(W_{n-1}^T X^n) < 0$, modify the weights: $W_n = W_{n-1} + \eta y_0^n X^n$. The adjustment of the weights can be represented as

$$W_n = W_{n-1} + \eta(y_0^n - y)X^n$$

where

$$y = \mathrm{sign}(W_{n-1}^T X^n)$$

As previously stated, the training samples are presented sequentially and repeated from the start after the Pth sample is processed. Upon completing a full cycle from the first to the last sample, we tally the errors made during that cycle (which corresponds to the number of updates to W). If no errors are found, the algorithm stops, indicating that all training samples are correctly classified. Fortunately, the following theorem ensures that the algorithm is effective if the data is linearly separable:

Perceptron Convergence Theorem: If P examples are given that are linearly separable, then the Perceptron Learning Algorithm will stop after a finite number of iterations to a vector W of connections that will divide the examples without errors, i.e. $\mathrm{sign}(W^T X^\mu) = y_\mu^0$ for every μ. Note: It is true that the number of iterations is finite, but it is usually larger than P because each example needs to be processed more than once.

Proof: We note W^* a weight vector that correctly separates the training examples: $\forall \mu, y_0^\mu = \mathrm{sign}(W^{*T} X^\mu)$. Let W_n be the student's weight vector in the n-th time step. Define the cosine similarity:

$$\cos(\theta_n) = \frac{W_n^T W^*}{\|W_n\| \|W^*\|} \tag{3}$$

The fundamental concept of the proof is that, if the algorithm continues indefinitely, the RHS of this equation exceeds 1 for large n. This occurs because W_n undergoes a biased random walk during the learning process. Since the bias is directed towards W^*, the numerator increases linearly with n. Conversely, because the change in W_n is biased away from the current weight vector W_{n-1}, the term $\|W_n\|$ in the denominator only increases as $\sqrt{n}$ and does not become large.

Multilayer neural network and supervised learning

As many of you may know, neural network models have been extremely successful in recent years as a tool for performing image recognition, classification, and they even beat the best go player in the world. The strategies for training such networks usually involve supervised learning. The network typically has a feedforward architecture, with one input layer, several hidden layers and one output layer. In an image classification task, the output could be a single number, for example, whether there is a cat in the picture or not. The output is compared to the ground truth, which is labelled by a human being. We can then define an error function

$$E = \frac{1}{2}\sum_k (r_k - t_k)^2, \tag{4}$$

where $\mathbf{t} = [t_1, ..., t_k, ...]^T$ is the target vector. Now if we have a batch of images, the mean cost function is the sum in the whole training set:

$$E(\mathbf{w}) = \frac{1}{N}\sum_{n=1}^{N} E_n(\mathbf{w}) \tag{5}$$

In Equation 5, N can be viewed as the number of images in a training set. For example, the ImagNet, popularized by Feifei Li, has millions of labelled images. We shall think E as a smooth function of the synaptic weights in the neural net. We can thus map the supervised learning problem to an optimization problem, and one simple way to reduce the cost function is to use gradient descent, so that

$$\mathbf{w}^{\tau+1} = \mathbf{w}^{\tau} - \eta\nabla E(\mathbf{w}^{\tau}) \tag{6}$$

Back Propagation

In the state-of-the-art deep neural network with many different layers, there are millions of synaptic weights. In this specific setting , computing the gradient involves a special technique, called back propagation. It is not mysterious. Here let's discuss how it works.

In a general feedforward neural network, each neuron computes a weighted sum of its inputs

$$I_j^l = \sum_i w_{ji} r_i^{l-1} \tag{7}$$

where the superscript l denotes the layer index. The activity of the postsynaptic unit r_j, is a nonlinear function of the synaptic current

$$r_j^l = F(I_j^l) \tag{8}$$

To compute the gradient, let's now look at individual synaptic connection w_{ji}. First we note that the cost function E_n depends on w_{ji} only via the synaptic current in neuron j. By applying the chain rule of the partial derivative, we have

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial I_j^l}\frac{\partial I_j^l}{\partial w_{ji}},$$

From Equation 7, we have

$$\frac{\partial I_j^l}{\partial w_{ji}} \equiv r_i^{l-1}.$$

We shall denote

$$\frac{\partial E_n}{\partial I_j^l} \equiv \delta_j^l,$$

and

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j^l r_i^{l-1}.$$

The meaning of δ_j can be understood immediately for the output unit I_j^N

$$\frac{\partial E_n}{\partial I_j^N} \equiv \delta_j^N = (r_j^N - t_j)F'(I_j^N),$$

which is proportional to the error signal. To compute δ_j in any hidden layer, we again use the chain rule

$$\frac{\partial E_n}{\partial I_j} = \sum_k \frac{\partial E_n}{\partial I_k}\frac{\partial I_k}{\partial I_j}$$

Here the sum is over all the neuron k in the next layer. Moreover, for each neuron in the next layer, we have

$$I_k^{l+1} = \sum_j w_{kj} r_j^l = \sum_j w_{kj} F(I_j^l),$$

and therefore

$$\frac{\partial I_k^{l+1}}{\partial I_j^l} = w_{kj} F'(I_j^l).$$

Putting everything together, we have the rules of *back propagation*

$$w_{ji} \leftarrow w_{ji} - \eta \delta_j^l r_i^{l-1} \tag{9}$$

$$\delta_j^l = F'(I_j^l) \sum_k \delta_k^{l+1} w_{kj} \tag{10}$$

Now, let's look at the physical meaning of this formula. The synaptic weight would update according to the product of the presynaptic neuron activity i and the error signal in neuron j. However, the error signal in neuron j is computed (or back propagated) from the error signals of all the neurons in the next layer, weighted by the output synaptic strengths from neuron j. This is *biologically implausible.*