Neural Network

Birds inspired us to fly, burdock plants inspired Velcro, and nature has inspired countless more inventions. It seems only logical, then, to look at the brain’s architecture for inspiration on how to build an intelligent machine. This logic sparked artificial neural networks (ANNs): a Machine Learning model inspired by the networks of biological neurons found in our brains. However, although birds inspired planes, they didn’t have to flap their wings.

Similarly, ANNs have gradually become quite different from their biological cousins. Some researchers even argue that we should drop the biological analogy altogether lest we restrict our creativity to biologically plausible systems. ANNs are at the very core of Deep Learning. They are versatile, powerful, and scalable, making them ideal for tackling large and highly complex Machine Learning tasks such as classifying billions of images (e.g., Google Images), powering speech recognition services (e.g., Apple’s Siri), recommending the best videos to watch to hundreds of millions of users every day (e.g., YouTube), or learning to beat the world champion at the game of Go (DeepMind’s AlphaGo).

Part of the inspiration for the study of artificial neural networks (ANNs) came from the realization that biological learning systems are composed of extremely intricate networks of interconnected neurons. To develop artificial neural networks, roughly speaking, a densely connected collection of simple units is used as the building blocks. These units each take in multiple real-valued inputs, possibly including the outputs of other units, and output a single real-valued output, which may then be used as the input for numerous additional units.

Although ANNs are inspired by biological neural systems, they do not fully capture their complexities. Additionally, some of the features discussed in this article are incompatible with biological systems. In contrast to biological neurons, which produce a complex time series of spikes, ANNs output a single constant value.

Pomerleau's system ALVINN provides a classic example of ANN learning, as it employs a learned ANN to drive an autonomous car traveling at regular speeds on public roadways. The neural network receives a 30 x 32 grid of pixel intensities from a forward-pointing camera mounted on the vehicle. The network output determines which way the car is guided. For about 5 minutes, the ANN mimics a human driver's steering commands. ALVINN has used its learned networks to effectively travel at speeds of up to 70 mph and distances of 90 miles on public highways, in the left lane of a split highway with other vehicles.

One type of ANN system is based on a unit called a perceptron, perceptron takes a vector of real-valued inputs, calculates a linear combination of these inputs, and then outputs a 1 if the result is greater than some threshold and -1 otherwise. More precisely, given inputs xl through xn, the output o(x1, . . . , xn,) computed by the perceptron is:

1 if w0 + w1x1 + w2x2 + … + wnxn > 0

o(x1, . . . , xn,) =

(-1) otherwise

where each wi is a real-valued constant or weight, that determines the contribution of input xi to the perceptron output. Notice the quantity (-wO) is a threshold that the weighted combination of inputs w1x1 + . . . + wnxn must surpass for the perceptron to output a 1.

A single perceptron can be used to represent many boolean functions. For example, if we assume boolean values of 1 (true) and -1 (false), then one way to use a two-input perceptron to implement the AND function is to set the weights wo = -3, and wl = wz = .5. This perceptron can be made to represent the OR function instead by altering the threshold to wo = -.3. In fact, AND and OR can be viewed as special cases of m-of-n functions: that is, functions where at least m of the n inputs to the perceptron must be true. The OR function corresponds to rn = 1 and the AND function to m = n. Any m-of-n function is easily represented using a perceptron by setting all input weights to the same value (e.g., 0.5) and then setting the threshold w0 accordingly. Perceptrons can represent all of the primitive boolean functions AND, OR, NAND (1 AND), and NOR (1 OR). Unfortunately, however, some boolean functions cannot be represented by a single perceptron, such as the XOR function whose value is 1 if and only if xl # xz. Note the set of linearly nonseparable training examples shown in Figure 4.3(b) corresponds to this XOR function. The ability of perceptrons to represent AND, OR, NAND, and NOR is important because every boolean function can be represented by some network of interconnected units based on these primitives. In fact, every boolean function can be represented by some network of perceptrons only two levels deep, in which the inputs are fed to multiple units, and the outputs of these units are then input to a second, final stage. One way is to represent the boolean function in disjunctive normal form (i.e., as the disjunction (OR) of a set of conjunctions (ANDs) of the inputs and their negations). Note that the input to an AND perceptron can be negated simply by changing the sign of the corresponding input weight. Because networks of threshold units can represent a rich variety of functions and because single units alone cannot, we will generally be interested in learning multilayer networks of threshold units.

Although we are interested in learning networks of many interconnected units, let us begin by understanding how to learn the weights for a single perceptron. Here the precise learning problem is to determine a weight vector that causes the perceptron to produce the correct 1 output for each of the given training examples. One way to learn an acceptable weight vector is, to begin with random weights, then iteratively apply the perceptron to each training example, modifying the perceptron weights whenever it misclassifies an example. This process is repeated, iterating through the training examples as many times as needed until the perceptron classifies all training examples correctly. Weights are modified at each step according to the perceptron training rule, which revises the weight wi associated with input xi according to the rule: wi ← wi wi, where, wi = (t - o)xi. Here t is the target output for the current training example, o is the output generated by the perceptron, and is a positive constant called the learning rate. The role of the learning rate is to moderate the degree to which weights are changed at each step. It is usually set to some small value (e.g., 0.1) and is sometimes made to decay as the number of weight-tuning iterations increases. Suppose the perceptron outputs a -1 when the target output is + 1. To make the perceptron output a + 1 instead of - 1 in this case, the weights must be altered to increase the value of w x. For example, if xi > 0, then increasing wi will bring the perceptron closer to correctly classifying this example.

Although the perceptron rule finds a successful weight vector when the training examples are linearly separable, it can fail to converge if the examples are not linearly separable. A second training rule, called the delta rule, is designed to overcome this difficulty. If the training examples are not linearly separable, the delta rule converges toward a best-fit approximation to the target concept. The delta rule works by using gradient descent to explore the hypothesis space of possible weight vectors for the weights that best suit the training examples. The BACKPROPAGATION algorithm, which can learn networks with several interconnected units, relies on gradient descent. Gradient descent is crucial for learning algorithms that need to search through complex hypothesis spaces with several continuously parameterized hypotheses. To understand the delta training rule, consider training an unthresholded perceptron (a linear unit with output o).

o(w) wx

Thus, a linear unit corresponds to the first stage of a perceptron, without the threshold. To derive a weight learning rule for linear units, let us begin by specifying a measure for the training error of a hypothesis (weight vector), relative to the training examples.

Ew12dDtd-od2

where D is the set of training examples, td is the target output for training example d, and od is the output of the linear unit for training example d. By this definition, E(w) is simply half the squared difference between the target output td and the linear unit output od, summed over all training examples. Here we characterize E as a function of w , because the linear unit output o depends on this weight vector. Of course, E also depends on the particular set of training examples.

To better comprehend the gradient descent technique, consider visualizing the whole hypothesis space of alternative weight vectors and their corresponding E values. Gradient descent search finds a weight vector that minimizes E by starting with an arbitrary initial weight vector and then updating it incrementally. At each step, the weight vector is adjusted in the direction that results in the sharpest drop down the error surface. This process continues until the global minimal error is met.

Implementation:

import numpy as np

#Sigmoid function as activation function

def sigmoid(x):

return 1/(1+np.exp(-x))

#Derivative of sigmoid

def sigmoid\_derivative(x):

return x\*(1-x)

class NeuralNetwork:

def \_\_init\_\_(self, input\_size, hidden\_size,output\_size):

pass

def forward(self,inputs):

pass

def backward(self,targets, learning\_rate):

pass

def train(self, inputs, targets, learning\_rate, epoch)

pass