SWE 212 Analysis of Programming Languages

Week 9 Summary of '*Learning Representations by Back-Propagating Errors'*, by David E. Rumelhart, Geoffrey E. Hinton & Ronald J. Williams

Samyak Jhaveri

The Final weekly summary!

This paper by, now godfathers of modern AI like Geoffrey Hinton, proposes the idea of designing a self-organizing neural network that, at its core, has a powerful synaptic modification rule that will enable it to develop hidden layers in the network. The learning procedure decides under what circumstances the hidden units should be active in order to achieve the desired input-output behavior. This means it can decide what these units should represent i.e., the hidden units are going to learn to represent some features of the input domain.

Suppose a network exists such that it has an input layer having input units at the base, a number of intermediate (hidden) layers in the middle, and an output layer with output units at the top. Connections within a layer or from a higher layer to a lower layer is not allowed. However, connections can skip the intermediate layers. The input units of the input layer are given an input vector. The states of the units in each intermediate layers are determined by the equation

$$x_j = \sum_i y_i w_{ij}$$

Where
x_j = total input
y_i = outputs
w_{ij} = weights of the inter-unit connections

These units can also be assigned biases by giving each of these units an extra weight. This extra weight is called a bias which is equal to the threshold of the opposite sign.

The output of the above function should be a function of the input that is called the activation function. In this paper, the sigmoid activation is used which is expressed as follows:

$$y_j = \frac{1}{1 + e^{-x_j}}$$

Repeating this procedure through the layers of the networks, it can be said that the data is being fed forward towards the output layer.

The objective of a neural network is to arrive at a set of weights that produce an output vector that is closest to the desired output vector. The error in a network can be computed by comparing the actual vs desired output result vectors for every case of the set of weights in the network units. Mathematically, error can be described as:

$$E = \frac{1}{2}\sum_{c}\sum_{j}(y_{j,c} - d_{j,c})^2$$

Where

c is index over cases,

j is an index over output units

y is the actual state of the output, and

d is the desired state

The network is intended to minimize the error. This is done by updating the weights. E can be minimized by gradient descent by computing the partial derivatives of E w.r.t each weight in the network. Further, we would like to know how the error changes as the weights are changed. Mathematically, it is expressed as :

$$\frac{\partial E}{\partial w}$$

This can be determined from how the error changes as the outputs are changed.

$$\frac{\partial E1}{\partial y_j}\sum_{j}(y_j - d_j)^2 = (y_j - d_j)$$

This tells us how the error changes in relation to outputs y. We can also calculate how y changes and x changes with

$$\frac{\partial y_j}{\partial w_{i,j}}$$

The sigmoid function is expressed as

$$y_j = \sigma(x_j)$$

Which can further be used to come up with the derivative:

$$\frac{\partial y_j}{\partial x_j} = \sigma(x_j).(1 - \sigma(x_j))$$

When the states of the units of each layer are determined by theinput they receive from a lower unit, it is called a forward pass. Conversely, the backward pass propagates derivatives from the top layer to the bottom layer.

The Backward Pass

Starts with computing $\frac{\partial E}{\partial y}$.

The formula for Error E when differentiated can be obtained as :

$$\frac{\partial E}{\partial y_j} = y_j - d_j$$

Further applying the chain rule and then substituting the value of $\partial E/\partial xj$ we get

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial y_j} \cdot y_j(1 - y_j)$$

Now, if we know how the error changes with the output y and how y changes with the input x and how x changes with the weight w, they can all be changed together to reveal how the error changes with the weights using gradient descent. Therefore we get:

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial x} \cdot \frac{\partial x}{\partial w}$$

With $\frac{\partial E}{\partial w}$, a simple way to implement gradient descent would be to change each weight proportionally to the accumulated gradient

$$\Delta w = -\epsilon \frac{\partial E}{\partial w}$$

This is a simpler method and is easily implemented on computers with parallel hardware. An acceleration method can also be used to improve its performance , that uses the current radiant to modify the velocities of the point-point weight space instead of its position.

$$\Delta w(t) = -\epsilon \frac{\partial E}{\partial w} + \alpha \Delta w(t - 1)$$

Here, t is increased incrementally by 1 for each sweep through the whole set of input-output cases, and **α** is an exponential decay factor between 0 and 1 that determines the relative contribution of the current gradient and earlier gradients to the weight change.

To conclude, this learning model is not the most plausible model for learning in brains. However, the use of hidden layers as internal representations using gradient descent may have interesting implications and uses.