1 Perceptron Preliminaries

Given data to learn from: input vectors $\vec{x}$ and produces scalar output values t. For example:

$$\begin{matrix} x_0 & x_1 & x_2 & x_3 & t \\ x_0 & x_1 & x_2 & x_3 & t \\ x_0 & x_1 & x_2 & x_3 & t \\ x_0 & x_1 & x_2 & x_3 & t \\ x_0 & x_1 & x_2 & x_3 & t \\ x_0 & x_1 & x_2 & x_3 & t \\ & & \vdots & & \end{matrix}$$

We want to find a function that given $\vec{x}$ produces a scalar y such that $y = t$ for all training data and generalizes well to yet unseen values of $\vec{x}$. That is, given $\vec{x}$, we want $t - y$ to be minimized across all training data. Our idea of generalization will come from assuming a linear transformation of $\vec{x}$ is able to smoothly model all missing cases. This, of course, might not be the case.

1.1 Perceptron Model

A **perceptron** does a linear transformation of an input vector $\vec{x}$ and produces an output y. The linear transformation is created by using a weight matrix W. For example, for an input, $\vec{x}$, from $\mathbb{R}^4$ to a single output $y \in \mathbb{R}$, W is a 4×1 matrix:

$$[x_0\ x_1\ x_2\ x_3] \cdot \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \end{bmatrix} = y$$

That is $\vec{x} \cdot W = y$. Now $t - y$ can be computed. If we want to improve the performance of W then we can adjust the elements of W by using the $t - y$ value: $w_i = w_i + \Delta_i$.

$$\begin{aligned} \Delta_0 &= \eta x_0(t - y) \\ \Delta_1 &= \eta x_1(t - y) \\ \Delta_2 &= \eta x_2(t - y) \\ \Delta_3 &= \eta x_3(t - y) \end{aligned}$$

At this point W is a single column matrix. To get $W = W + \Delta$ to work, Δ needs to be a single column matrix. We need to go from the single row $\vec{x}$ to a column using transpose:

$$\Delta = \eta(\vec{x})^{\mathsf{T}}(t - y)$$

Note at this point in our discussion η and $(t - y)$ is just scalars.

1.2 Bias

The problem with this linear transform model is 0 must map to 0. There is no way to move the zero point! This is like using $y = mx + b$ for a line without the b. So we will include a **bias node** into our model. We can fit it into our current model by making an extra input that is constant!

$$[x_0\ x_1\ x_2\ x_3\ 1] \cdot \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ w_3 \\ w_4 \end{bmatrix} = y$$

Now w_4 is always added into the sum to shift the sum. Note w_4 can be negative and is adjusted with the other weights to make a better fit of y to t in the test data. So our $\vec{x}$ and W are altered to include a bias term.

1.3 Multiple Outputs

For many problems you want t to be able to be $\vec{t}$. This is easily handled in our model by using more columns in W.

$$[x_0\ x_1\ x_2\ x_3\ 1] \cdot \begin{bmatrix} w_{00} & w_{01} \\ w_{10} & w_{21} \\ w_{20} & w_{31} \\ w_{30} & w_{41} \\ w_{40} & w_{41} \end{bmatrix} = [y_0\ y_1]$$

So now $\vec{x} \cdot W = \vec{y}$ which is then compared to the training data in which t is now a vector of expected outputs $\vec{t}$.

So how does *Delta* work in this case:

$$\Delta = \eta(\vec{x})^{\intercal}(\vec{t} - \vec{y}) \tag{1}$$
$$W = W + \Delta \tag{2}$$

The dimensions work because $(\vec{x})^{\intercal}$ is a single column matrix and $(\vec{t} - \vec{y})$ is a single row matrix. This produces a matrix with the same dimensions as W!

1.4 Training Blocks

Updating based on the training data can be done one training case at a time in a random order or can be batched into blocks of training cases. Sometimes blocking is good and sometimes it can be bad. This algorithm is basically a stochastic optimization. We hope that the deltas will guide the W through weight space to an optimal solution but sometimes we can be trapped or misdirected.

Let's look at a block of training data and how to do an adjustment.

$$\begin{bmatrix} x_0 & x_1 & x_2 & x_3 & 1 \\ x_0 & x_1 & x_2 & x_3 & 1 \\ x_0 & x_1 & x_2 & x_3 & 1 \\ x_0 & x_1 & x_2 & x_3 & 1 \\ x_0 & x_1 & x_2 & x_3 & 1 \\ x_0 & x_1 & x_2 & x_3 & 1 \\ & & \vdots & & \end{bmatrix} \text{ and } \begin{bmatrix} t \\ t \\ t \\ t \\ t \\ t \\ \vdots \end{bmatrix}$$

Call the input for training data with bias the matrix X. The target results T is a matrix of corresponding target vectors $\vec{t}$ for each $vecx$ in X. For block training we have:

$$Y = X \centerdot W$$

which gives us all of the answers $\vec{y}$ predicted for each $\vec{x}$. Our differences are now the matrix subtraction $T - Y$. What we want to use is Equation 4 for each vector $\vec{x}$ in X and sum the results across $\vec{x}$. This could be done by summing the columns of $T - Y$ and using Equation 4 for each vector $\vec{x}$ in X. But a more compact way to compute the same thing in one matrix multiply:

$$\Delta = \eta X^{\intercal}(T - Y) \tag{3}$$

$$W = W + \Delta \tag{4}$$

1.5 Sigmoids

There are two main kinds of outputs we want from a perceptron. The first is the raw real number that is output of linear transformation. The second is a decision made by interpreting the value of the output of the linear transform. This would be something like all $\vec{x}$ for which $y >$ threshold. A very useful way, as we will see when we get to multilayer networks, is to use a sigmoid function.

Figure 1 shows two classic sigmoid functions. The green sigmoid curve has asymptotes at 0 and 1 and a derivative or slope of $s/4$ at 0:

$$\frac{1}{1 + e^{-sx}}$$

The blue sigmoid has asymptotes at -1 and 1 and the same derivative of $s/2$ at 0:

$$\frac{2}{1 + e^{-sx}} - 1$$

The perceptron model can be modified by applying a sigmoid function to the elements of Y as cartooned using a classic element by element map function on the matrix Y from Equation zzz:

$$Y = \text{map}(X \centerdot W, \text{sigmoid})$$

For more on map functions see Wikipedia: Map_(higher-order_function).

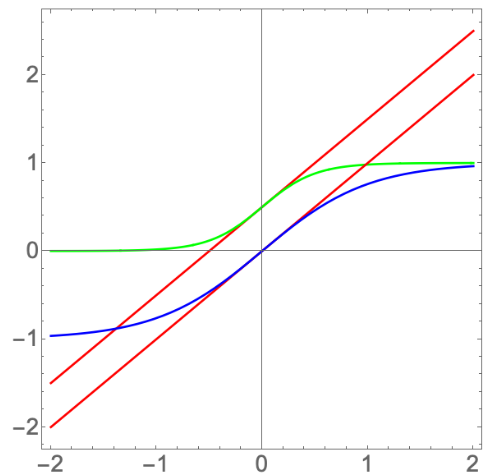

Figure 1: Sigmoids in the range -1 to 1 and in the range 0 to 1 with slopes of 1 as demonstrated by the diagonal red lines.