Perceptrons: An Overview

Harjas Monga and Vidhur Kumar

April 1, 2018

Contents

"We expect the perceptron to be the embryo of an electronic computer that will be able to walk, talk, see, write, reproduce itself and be conscious of its existence." - The New York Times, 1958.

1 What this report entails

It's monday morning. You're sitting in your mathematics lecture. The professor asks you to solve the following equation:

$$5 + 2 = ?$$

Within a fraction of a second, you yell "It's seven!" and the professor lauds you. You lean back, thinking "Pfsh! That was easy." You just unknowingly undervalued one of the most incredible tools that you have been blessed with: the human visual system. In each hemisphere of our brain, humans have a primary visual cortex (V1), containing 140 million neurons, with tens of billins of connections between them. There is an entire series of visual cortices doing progressively more complex image processing [1].

We carry in our heads a supercomputer. We do not realize how good we are at making sense of our milieu because all the work is done unconsciously. Only when we attempt to write computer programs to emulate our neural processes is when we truly realize the complexity of the human mind. Nevertheless, the curiosity of scientists persisted and it was in 1958 that Frank Rosenblatt first introduced the idea of a concrete artificial mind. He named it the Perceptron, hoping that "it may eventually be able to learn, make decisions, and translate languages."

It is the idea of the Perceptron that we explore in detail in this paper. We have divided it into five sections. First, we provide a gentle introduction to the concept of a perceptron. Second, we explore the mathematical foundations of a perceptron in depth to really understand what makes it work.

2 Introduction: What is a Perceptron?

The perceptron is an algorithm for learning a binary classifier: a function that maps its input x (a real-valued **vector**) to an output value $f(x)$ (a single binary value). Shown below is a mathematical definition:

$$f(x) = \begin{cases} 1 & wx + b > 0 \\ 0 & otherwise \end{cases}$$

where w is the weight corresponding to the input x, and b is the bias. $b = -threshold$. If the value of the weighted input is greater than the threshold, we output 1. If not, we output 0.

3 Workings: What drives a Perceptron?

3.1 Mathmatical Model of a single Perceptron

A perceptron a function that takes in several inputs and produces an output of either 0 or 1.

The input of perceptron is usually modeled as a column vector. Let $X(n \times 1)$ represent this column vector. The perceptron applies weights to each values in the X and then sums them together. In other words, the perceptron finds a linear combination of the matrix X as show below.

$$X = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{bmatrix}$$

Let α be a possible linear combination of X.

$$\alpha = \sum_{i=1}^{n} a_i x_i = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n$$

At first, the a_i are chosen randomly. However, during the training process they will be adjusted until they are able to produce the optimal output.

The next step of the perceptron utilizes an activation function. An activation function can be either 0 or 1, depending on a threshold value. The threshold value is selected based on the specific use of the perceptron. Below is an example of an activation function. Let T represent the threshold value.

$$f(x) = \begin{cases} 1 & x >= T \\ 0 & x < T \end{cases}$$

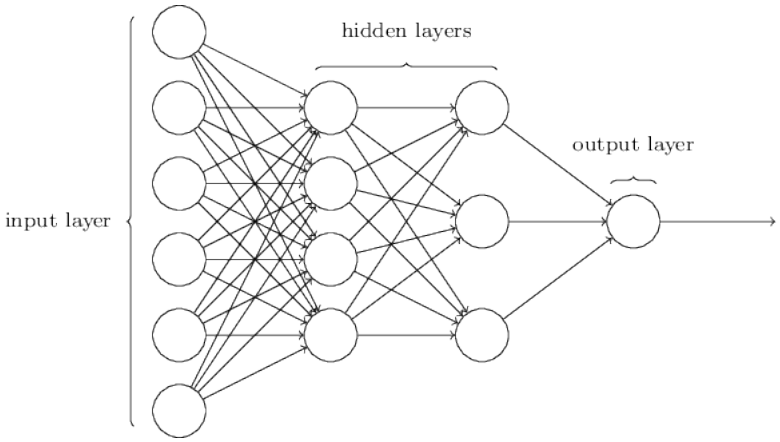
The output of the perceptron is the output of the activation function above(since this is a single layer perceptron).

3.2 Layering perceptrons

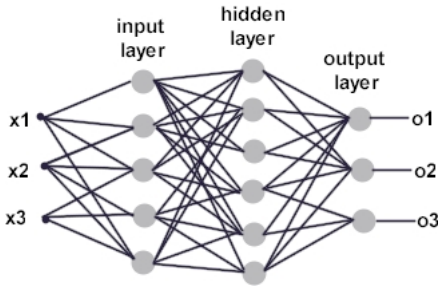
Perceptrons gain a lot more functionality when they are used in combination with other perceptrons. Let p_i represent a perceptron. Perceptrons are stacked together to build column vectors of perceptrons as show below.

$$P = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_3 \end{bmatrix}$$

A column vector of perceptrons is called a layer. There are three kinds of perceptron layers. There is an input layer. The input layer is always the first layer in any multilayered perceptron. Then, there is the output layer. This layer is always the last layer, and as the name suggests this is the output of the multilayer perceptron. In between the output layer and the input layer are the hidden layers. There can be any number of hidden layers in a perceptron. Each of these layers use the output of the layer right before it as its input.



The diagram aboves represents how a multilayer perceptron is structured. Notice how each perceptron takes in all the outputs from the layer before it. This multilayer perceptron only has 1 output, however, it is possible to structure a perceptron to output multiple values if needed. An example of a multilayer perceptron with multiple outputs is shown below.



The structure above is used when the problem is more complex. For example, a perceptron that takes in the pixel values of an image of a handwritten number and then outputs what it thinks the number is needs 10 outputs, 1 for each possible value.

A perceptron can be a very useful tool for difficult problems such as recognizing handwriting, because they can be used to build any possible logical function. Designing multilayered perceptrons to read handwriting is significantly easier than trying to write the explicit logical statements for it.

3.3 Training

3.3.1 Sigmoid Neuron

A perceptron as currently described is not very useful if you are trying to train a multilayer perceptron to work as intended. For this purpose we replace the perceptrons we have been currently using with the more powerful sigmoid neurons. A sigmoid neuron works very similarly to the standard perceptron. However, the key difference is that unlike a perceptron a sigmoid neuron can output any value between 0 and 1. Additionally, the method in which the output of the neuron is computed is different than the computation for the standard perceptron. Let column vector A represent the input for the sigmoid neuron.

$$A = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix}$$

$$\text{let } Z = \sum_{i=1}^{n} w_i a_i - b$$

where b is a bias and w_i is the weight being applied to the ith input. The bias is utilized instead of using the threshold as was used before with standard perceptron. Let $W = [w_i]$ so that Z can be written as

$$W = \begin{bmatrix} w_1 & w_2 & \dots & w_n \end{bmatrix}$$

$$Z = \begin{bmatrix} w_1 & w_2 & \dots & w_n \end{bmatrix} \bullet \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} - b = W \bullet A - b$$

The sigmoid function

$$\sigma(Z) = \frac{1}{1 + e^{-Z}}$$

is used to take any input and return a value between 0 and 1. This is useful because the value of Z is not constrained to $[0, 1]$. However, we need the output to be $\in [0, 1]$ for analysis and the next layer of perceptrons.

The sigmoid neuron is useful because it always the training algorithm to make small adjustments to the weights and biases, which in turn allows us to make small adjustments to the output of the multilayered perceptrons. Training is quite difficult without using sigmoid neurons because the values are binary. Thus, we cannot comprehend the impact of a small change to the output.

3.3.2 Cost Function

The Cost function is a formula that takes the perceptron output and the correct output as the input and outputs how bad or good the multilayer perceptron is. The training algorithm uses the cost function to determine what effects changes on the weights and biases have on the accuracy of the multilayered perceptron.

Let

$$A = \begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{21} & a_{22} & \dots & a_{2m} \\ \vdots & & & \\ a_{n1} & a_{n2} & \dots & a_{nm} \end{bmatrix} \quad C = \begin{bmatrix} c_{11} & c_{12} & \dots & c_{1m} \\ c_{21} & c_{22} & \dots & c_{2m} \\ \vdots & & & \\ c_{n1} & c_{n2} & \dots & c_{nm} \end{bmatrix}$$

Where the column vectors of A represent the desired output of the multilayered perceptron, and where the column vectors of C represent the actual output of the multilayered perceptron. If $C(w, b)$ is the cost function, then

$$C(w,b) = \frac{1}{2n}\sum_{i=1}^{m} ||(A^T)_i - (C^T)_i||^2$$

Where w and b represent the weights and biases that are needed to compute C.

3.3.3 Gradient Descent

Gradient descent is the algorithm that is used to improve the multilayered perceptron. The purpose of gradient descent is to minimize the cost function. This is because the accuracy of the multilayered perceptron is better when the cost function is closer to 0.

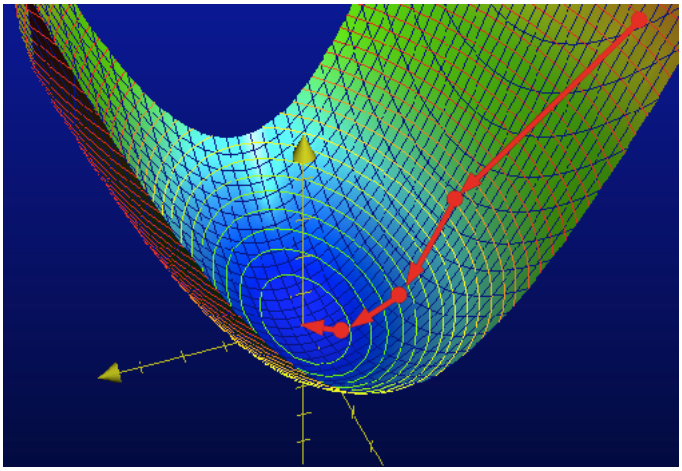
Gradient descent achieves this goal by computing the negative gradient of the cost function.

$$g(a) = -\nabla C(w,b)$$

The function $g(a)$ points in the direction in which the Cost function is decreasing the fastest, we know this to be true because $\nabla C(w,b)$ points in the direction the cost function increase the fastest by the definition of the gradient. The process by which the gradient of the cost function is computed is called backpropagation and is beyond the scope of this paper.

The $g(a)$ function is used to find the direction in which the all the weights and biases need to be modified to minimize the cost function. Once a small modification to the weights and biases are made $g(a)$ is recalculated and the weights and baises are changed again. This process continues until we find a local minimum of the cost function.

Gradient descent occurs in n dimensional space, where n is the number of weights and biases that the multilayered perceptron has. There are usually thousands of weights and biases in a multilayered perceptron. Because of this, it very difficult to visualize the process of gradient descent. However, it is possible to visualize a much simpler example in R^3.



It is clear from the diagram above, that the gradient descent algorithm moves the cost function closer and closer towards a local minimum.

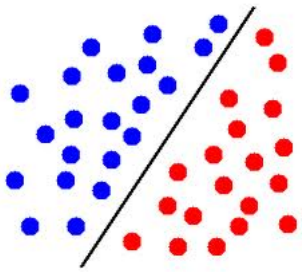
It should be reemphasized that this is local minimum, not a global minimum. This means that gradient descent does not necessarily give you the best answer. The task of finding a global minimum of the cost function is a much harder task to accomplish. Overall, the gradient descent algorithm is the most efficient method of finding a local minimum of the cost function. This can be proven using the Cauchy-Schwarz inequality. This is left as an exercise for the reader :).

4 Applications: What can I do with a Perceptron?

4.1 Linear Separability

As stated before, a perecptron is a binary classifier. A fairly obvious application of the concept is for classification. There are several instances where classification is necessary.

For this report, we have chosen a rudimentary example of a classification task: **Linear separability**. Consider a set of blue points and a set of red points on a two-dimensional plane. The two sets are linearly separable if there exists a line on the plane such that all of the blue points lie on one side of the plane and all of the red points lie on the other side.



Shown above is an instance of two-dimensional linear separation. This idea is immediately generalized to n-dimensional Euclidean spaces if the line is replaced by a hyperplane.

4.2 How is the Perceptron useful here?

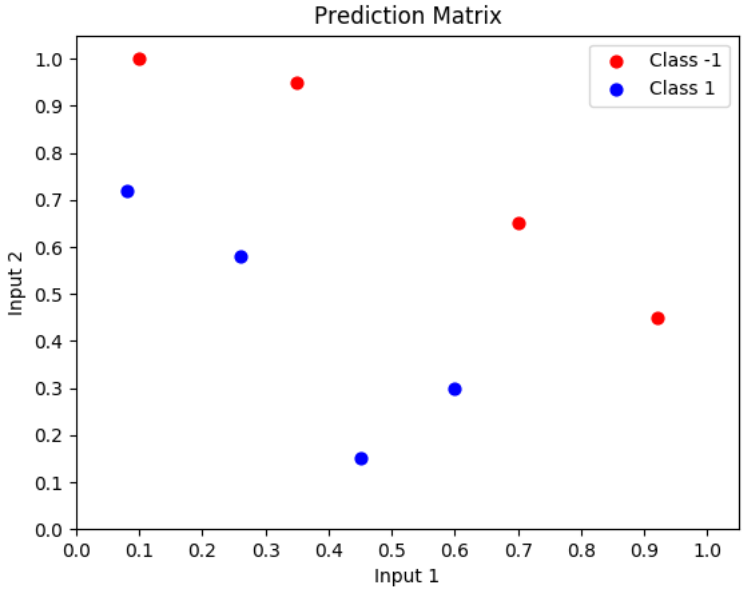
Recall that a Perceptron accepts a set of inputs and produces an output of either 0 or 1. We can pass in the coordinates of each point along with the bias (which we will explain later) and the expected output. This will be a $4x1$ column vector. We will call our input vector x.

$$x = \begin{bmatrix} b \\ i1 \\ i2 \\ y \end{bmatrix}$$

where b is the bias, $i1$ is the x-coordinate of the point, $i2$ the y-coordinate, and y is the expected output (0 or 1).

4.3 Implementing the Perceptron

We decided to use Python to implement the algorithm. We generated 10 random two-dimensional points such that they are linearly separable.



Shown above is are the 10 points plotted. Notice how there is a line (with slope ≈ -1) that can separate the two sets. The points above the line will be Class -1 and have a color of Red, while the points below the line will be Class 1 and have a color of Blue.

We will first implement a function *predict*() that will return the appropriate class for a specified set of inputs and their corresponding weights.

```
# Returns 1 if the weighted input sum is greater
# than a threshold.
def predict(inputs, weights):
    activation_threshold = 0.0

    # Obtaining the weighted sum of the inputs.
    for i, w in zip(inputs, weights):
        activation_threshold += w * i

    return 1.0 if activation_threshold >= 0.0 else 0.0
```

1. Initialize the *activation_threshold* to 0.0. Any value above it will return 1, and any value below will return 0.

2. Iterate through the inputs and their corresponding weights and calculate the weighted sum. The function **zip** lets us iterate through

both the inputs and the weights parallely. We store the sum in the *activation_threshold* variable.

3. Return the appropriate class depending on the value of the weighted sum.

Next, we implement a function *accuracy*() that will calculate the percentage accuracy of the classifier during each iteration.

```
# Returns the percentage accuracy of the classifier.
def accuracy(matrix, weights):
    num_correct = 0.0
    preds = []

    for i in range(len(matrix)):
        pred = predict(matrix[i][:-1], weights)
        preds.append(pred)

        if pred == matrix[i][-1]:
            num_correct += 1.0

    print("Predictions:", preds)

    return num_correct / float(len(matrix))
```

1. Initialize *num_correct* to 0.0 (number of correct predictions) and *preds* to an empty array (we will store multiple predictions in this array).

2. We iterate through the *matrix*. *matrix* is a two-dimensional array, i.e., a collection of data vectors. The function **len** allows us to calculate the number of data vectors in *matrix*.

3. We add the predicted class of each data vector to the *preds* array. Note that each prediction is itself an array, so this makes *preds* a two-dimensional array.

4. If the predicted class matches the expected class, then we increment $num_correct$.

5. We print out the list of predictions and return the ercentage accuracy of the predictions. The **float** function helps us prevent truncation (rounding down) of the answer to give us a better answer.

Finally, we implement our backpropogation algorithm, where we update the weights after each iteration to improve the predictions made by the perceptron.

```
# Backpropogation algorithm
def train_weights(matrix, weights, n_iteration=10,
l_rate=1.00, do_plot=False, stop_early=True,
verbose=True):

    for epoch in range(n_iteration):
        current_accuracy = accuracy(matrix, weights)
        print("\nEpoch %d \nWeights: " %epoch, weights)
        print("Accuracy: ", current_accuracy)

        # If the maximum accuracy is reached
        if current_accuracy == 1.0 and stop_early:
            break

        if do_plot:
            plot(matrix, weights, title="Epoch %d"
            %epoch)

        for i in range(len(matrix)):

            # Current prediction
            prediction = predict(matrix[i][:-1],
            weights)

            # Error in prediction
            error = matrix[i][-1] - prediction

            # Updating the weights
            for j in range(len(weights)):
                if verbose:
                    sys.stdout
                    .write("\tWeight[%d]: %0.5f ---> "
                    %(j, weights[j]))

                # Updating the weights.
                weights[j] +=
                (l_rate * error * matrix[i][j])
```

```
                if verbose:
                    sys.stdout
                    .write("%0.5f\n"
                    %(weights[j]))

    # Plotting the final matrix.
    plot(matrix, weights, title="Final epoch")
    return weights
```

1. An epoch is a forward and backward pass when it comes to training. The number of iterations is the sum of the forward and backward passes. We thus have $\frac{n_iteration}{2}$ number of iterations in this function. We save the current accuracy into *current_accuracy*.

2. Should our algorithm ever reach maximum accuracy, the **break** statement allows us to terminate the training process immediately.

3. Should we ever need to see the predictions visually each time (*do_plot* will have a value of **True** in this case), we plot the data and the line to separate them.

4. We then iterate through the *matrix* (which is again a two-dimensional array) and save the error of our prediction into *error*.

5. Using the error in our prediction, we then iterate through *weights* (this is an array of values that we change each time the *train_weights*() function is called. In each iteration, we update $weights[j]$ (which is the jth value in the *weights* array) using the following formula

$$w_j = w_j + (l * e_i * M_{i,j})$$

   where w_j is the j^{th} value in the *weights* array, l is the learning rate (a measure of how fast the perceptron should learn that was chosen arbitrarily), e_i is the error in prediction for the i^{th} data vector in *matrix*, and $M_{i,j}$ is the j^th value in the i^{th} data vector (the corresponding input to the weight).

6. The *verbose* variable is a boolean value. If it is **True**, we print out the value of the weights before and after they are updated.

7. Regardless of whether we plot the line and data points during each iteration, we plot the points and the line after the iteration terminates.

8. Finally, we return the updated *weights*.

4.4 The Results

Shown below are the plots from each epoch. Note that we start the number at 0. We have $\frac{10}{2} = 5$ epochs.

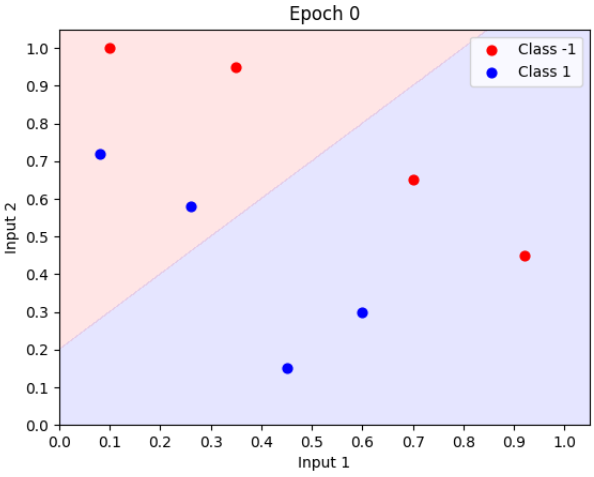
1. Epoch 0: Since we set the *weights* vector arbitrarily, our line of separation is highly inaccurate.

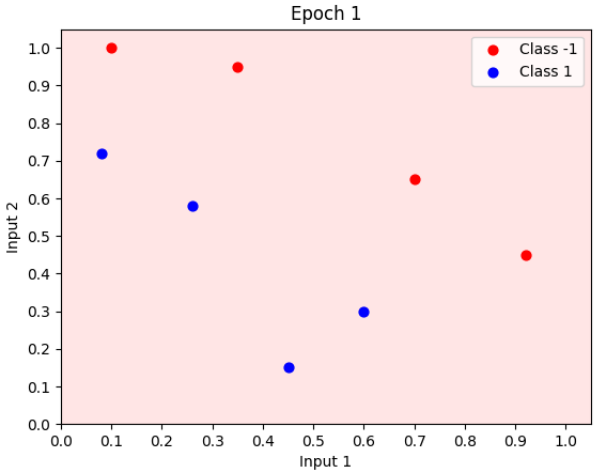
2. Epoch 1: Since we updated the weights, the perceptron 'realizes' that the line of separation was inaccurate due to the high error rate it would have received. Hence, we do not see much of a line in this epoch.

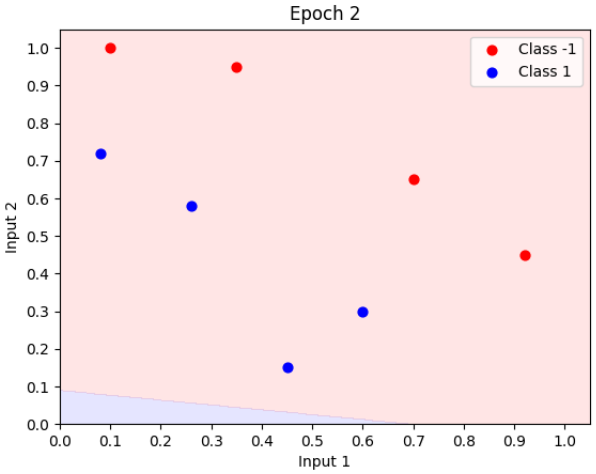
3. Epoch 2: We begin to see a line of separation. It is still inaccurate.

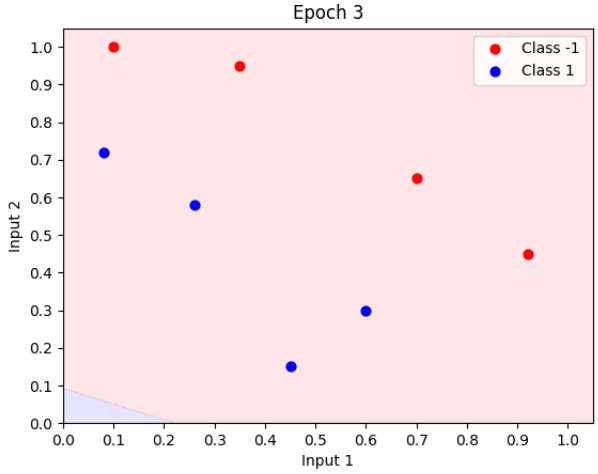
4. Epoch 3: The line of separation's slope has been adjusted quite significantly.

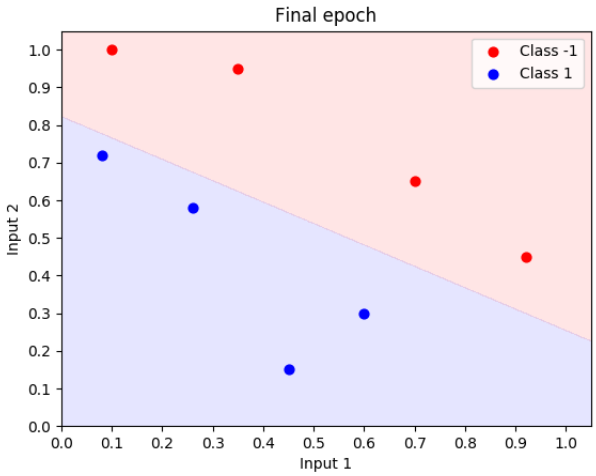
5. Epoch 4 (Final): We see a clear and accurate line separating the two sets of points.




Epoch 1
Class -1
Class 1
Input 2
Input 1
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0


Epoch 2
Class -1
Class 1
Input 2
Input 1
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0


Epoch 3
Class -1
Class 1
Input 2
Input 1
0.0 0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1.0



5 Conclusion: The Authors' Personal Thoughts

Perceptrons are a great tool for building logical functions that would be extremely difficult to explicitly express in standard logical statements. However, this is not where their true powers lie. The Perceptron is an old concept, but they are key in building the most powerful tools. Neural networks and machine learning algorithms build on these concepts and the mathematical foundations of perceptrons to build more powerful systems. These neural nets have already accomplished great feats such as beatings humans in chess and the Chinese game Go.

We believe these neural nets and machine learning algorithms have the potential to enable general Artificial Intelligence. However, there are a few qualms regarding general AI: it is important to realize that we are building a system that is smarter than us. There is no telling as to how it thinks because that is beyond our scope of thought. It is a possibility that this juggernaut elects to destroy humans because it considers us inefficient and error-prone. Hence, we urge our readers who aspire to work which such systems to tread carefully. The systems that you build to cure cancer might be the same system that results in the the extinction of the human race. Godspeed.