Pattern Recognition and Machine Learning

James L. Crowley

ENSIMAG 3 - MMIS
Lesson 4

Fall Semester 2020
25 November 2020

# Perceptrons and Gradient Descent

**Outline**

Notation ........................................................2

Perceptrons ........................................................3
History ........................................................3
The Perceptron Classifier ........................................................3
The Perceptron Learning Algorithm........................................................5

Artificial Neural Networks ........................................................6
The Artificial Neuron ........................................................7
Homogeneous Coordinate Notation ........................................................8

Gradient Descent ........................................................9
Loss (Cost) Function ........................................................9
Feature Scaling ........................................................11
Local Minima ........................................................12
Batch mode........................................................13
Stochastic Gradient Descent........................................................13

## Notation

| | |
|---|---|
| $x_d$ | A feature. An observed or measured value. |
| $\vec{X}$ | A vector of D features. |
| D | The number of dimensions for the vector $\vec{X}$ |
| $\vec{y}$ | A dependent variable to be estimated. |
| $a = f(\vec{w}^T \vec{X} + b)$ | A (neural) model that predicts $a$ from $\vec{X}$. |
| $\vec{w}, b$ | The parameters of the model. |
| $\{\vec{X}_m\}$ $\{y_m\}$ | Training samples for learning the model. |
| $M$ | The number of training samples. |
| $C_m = \frac{1}{2}(a_m - y_m)^2$ | The Loss (or cost) for the function for computing $a_m = f(\vec{w}^T \vec{X}_m + b)$ |
| $\vec{\nabla} C_m = \frac{\partial C_m}{\partial \vec{w}}$ | The gradient (vector derivative) of the cost. |

# Perceptrons

## History

The Perceptron is an incremental learning algorithm for linear classifiers invented by Frank Rosenblatt in 1956. The approach was first proposed by Warren McCullough and Walter Pitts in 1943 as a possible universal computational model. During the 1950's, Frank Rosenblatt developed the idea to provide a trainable machine for pattern recognition. The first Perceptron was a room-sized analog computer that implemented Rosenblatz's learning function for recognition. However, it was soon recognized that both the learning algorithm and the resulting recognition algorithm are easily implemented as computer programs.

## The Perceptron Classifier

The perceptron is an on-line learning algorithm that learns a linear decision boundary (hyper-plane) for separable training data. As an "on-line" learning algorithm, new training samples can be used at any time to update the recognition algorithm. However, if the training data is non-separable, the method will not converge, and must be stopped after a certain number of iterations.

The Perceptron algorithm uses errors in classifying the training data to iteratively update the hyper-plane decision boundary. Updates may be repeated until no errors exist.

Assume a training set of $M$ observations $\{\vec{X}_m\}$ of D features, with indicators variables, $\{y_m\}$ where

$$\vec{X}_m = \begin{pmatrix} x_{1m} \\ x_{2m} \\ \vdots \\ x_{Dm} \end{pmatrix} \text{ and } y_m = \{-1, +1\}$$

The indicator variable, $\{y_m\}$, tells the class label for each sample.
For binary pattern detection,

$y_m = +1$ for examples of the target class (class 1)
$y_m = -1$ for all others (class 2)

The Perceptron will learn the coefficients, $\vec{w}, b$, for a linear boundary

$$\vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{pmatrix} \text{ and } b$$

Such that for all training data, $\vec{X}_m$,

$\vec{w}^T \vec{X}_m + b \geq 0$ for Class 1 and $\vec{w}^T \vec{X}_m + w_0 < 0$ for Class 2.

Note that $\vec{w}^T \vec{X}_m + b \geq 0$ is the same as $\vec{w}^T \vec{X}_m \geq -b$.
Thus b can be considered as a threshold on the product: $\vec{w}^T \vec{X}_m$

The decision function is the sgn() function: $\operatorname{sgn}(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ -1 & \text{if } z < 0 \end{cases}$
Where $z = \vec{w}^T \vec{X}_m + b$

A training sample is correctly classified if:

$$y_m \cdot \left(\vec{w}^T \vec{X}_m + b\right) \geq 0$$

The algorithm requires a learning rate, $\eta$. Typically set to a very small number such as $\eta = 10^{-3}$

## The Perceptron Learning Algorithm

The algorithm will continue to loop through the training data until it makes an entire pass without a single misclassified training sample. If the training data are not separable then it will continue to loop forever.

Algorithm:

```
$\vec{w}^{(0)} \leftarrow 0$; $b^{(i)} \leftarrow 0$, $i \leftarrow 0$; set $\eta$ (for example $\eta = 10^{-3}$)
WHILE update DO
    update ← FALSE;
    FOR $m = 1$ TO $M$ DO
        IF $y_m \cdot \left(\vec{w}^{(i)T}\vec{X}_m + b^{(i)}\right) < 0$ THEN
            update ← TRUE
            $\vec{w}^{(i+1)} \leftarrow \vec{w}^{(i)} + \eta \cdot y_m \cdot \vec{X}_m$
            $b^{(i+1)} \leftarrow b^{(i)} + \eta \cdot y_m$
            $i \leftarrow i + 1$
        END IF
    END FOR
END WHILE.
```

Notice that the weights are a linear combination of training data that were incorrectly classified.

The final classifier is: if $\vec{w}^{(i)T}\vec{X}_m + b^{(i)} \geq 0$ then P else N.

If the data is not separable, then the Perceptron will not converge, and will continue an infinite loop. Thus it is necessary to have a limit the number of iterations.

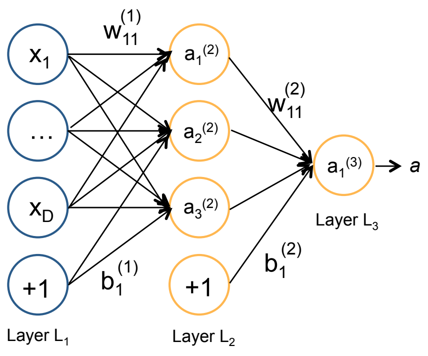
In 1969, Marvin Minsky and Seymour Papert of MIT published a book entitled “Perceptrons”, that claimed to document the fundamental limitations of the perceptron approach. Notably, they claimed that a linear classifier could not be constructed to perform an “exclusive OR”. While this is true for a one-layer perceptron, it is not true for multi-layer perceptrons.

The fact that the algorithm requires separable training data WAS a major weakness. This limitation was later overcome by reformulating the algorithm using a soft decision surface and Gradient descent.
The result was promoted as a form of "Artificial Neural Network".

## Artificial Neural Networks

In the 1970s, frustrations with the limits of Artificial Intelligence research based on Symbolic Logic led a small community of researchers to explore the perceptron based approach. In 1973, Steven Grossberg, showed that a two layered perceptron could overcome the problems raised by Minsky and Papert, and solve many problems that plagued symbolic AI. In 1975, Paul Werbos developed an algorithm referred to as "Back-Propagation" that uses gradient descent to learn the parameters for perceptrons from classification errors with training data. Back-propagation is a parallel form of Gradient descent easily implemented on a SIMD parallel computer.

Artificial Neural Networks are computational structures composed a weighted sums of "neural" units. Each neural unit is composed of a weighted sum of input units, followed by a non-linear decision function.

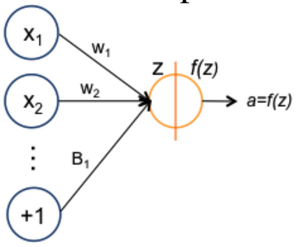


Note that the term "neural" is misleading. The computational mechanism of a neural network is only loosely inspired from neural biology. Neural networks do NOT implement the same learning and recognition algorithms as biological systems.

During the 1980's, Neural Networks went through a period of popularity with researchers showing that Networks could be trained to provide simple solutions to problems such as recognizing handwritten characters, recognizing spoken words, and steering a car on a highway. However, results were overtaken by more mathematically sound approaches for statistical pattern recognition such as support vector machines and boosted learning.

## The Artificial Neuron

The simplest possible neural network is composed of a single neuron.



A "neuron" is a computational unit that integrates information from a vector of features, $\vec{X}$, to compute the likelihood of an activation, $a$.

$$a = f(z)$$

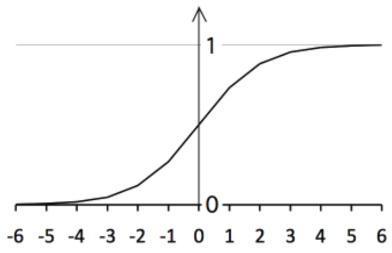
The neuron is composed of a weighted sum of input values

$$z = w_1 x_1 + w_2 x_2 + ... + w_D x_D + b$$

followed by a non-linear "activation" function, $f(z)$

$$a = f(z) = f(\vec{w}^T \vec{X} + b)$$

A popular choice for activation function is the sigmoid: $\sigma(z) = \dfrac{1}{1+e^{-z}}$



The sigmoid is useful because the derivative is: $\dfrac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$

For the sigmoid, the target function is $y_m \in \{0,1\}$, enabling easy generalization to multi-class decisions. This can give a decision function:

if $f(\vec{w}^T \vec{X} + b) \geq 0.5$ the P else N

We will use Gradient descent to learn the best weights and bias for a training set of M samples $\{\vec{X}_m\}$ with indicator variables $\{y_m\}$.

## Homogeneous Coordinate Notation

We would like to be able to treat the bias, *b*, as one of the coefficients of the function $f(\vec{w}^T\vec{X}+b)$. We can do this using homogeneous coordinates.

With homogeneous coordinates, we add an additional constant term to the input feature vector $\vec{X}$. This allows us to include the bias in the model vector $\vec{w}$.

$$\vec{X} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \\ 1 \end{pmatrix} \text{ and } \vec{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_D \\ b \end{pmatrix}$$

In this case, b is simply the D+1th coefficient of $\vec{w}$, and the linear model is expressed as: $z = \vec{w}^T\vec{X}$

$$z = \vec{w}^T\vec{X} = \begin{pmatrix} w_1 & \cdots & w_D & b \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_D \\ 1 \end{pmatrix}$$

We use Gradient Descent to estimate the coefficients $\vec{w}$.

## Gradient Descent

Gradient descent is a first-order iterative optimization algorithm for finding the local minimum of a differentiable function. Gradient descent is a popular algorithm for estimating parameters for a large variety of models.

The gradient of a scalar-valued differentiable function of several variables, $f(\vec{X})$ is vector derivatives:

$$\vec{\nabla} f(\vec{X}) = \frac{\partial f(\vec{X})}{\partial \vec{X}} = \begin{pmatrix} \frac{\partial f(\vec{X})}{\partial x_1} \\ \frac{\partial f(\vec{X})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\vec{X})}{\partial x_D} \end{pmatrix}$$

The gradient of a function $f(\vec{X})$ at a point $\vec{X}$ is the direction and rate of change for the greatest slope of a surface. The direction of the gradient is the direction of greatest slope, the magnitude is the gradient is the rate of change in that direction.

To find a local minimum of a function using gradient descent, we iteratively update the function by subtracting corrections proportional to the gradient of the function at the current point. To use this to determine the parameters for a perceptron (or neural unit), we must introduce the notion of a Loss or cost for an error.

### Loss (Cost) Function

The Loss (or cost) function is the cost of an error for classifying a data sample $\vec{X}_m$ with ground truth $y_m$ using with network parameters $\vec{w}$. (using homogeneous coordinates).

Assume M samples of training data $\vec{X}_m$ with indicator variables $y_m$. The vector, $\vec{X}_m$, has D dimensions. The indicator $y_m$, gives the expected result for the vector.

The cost (or Loss) for using the weights and biases $\vec{w}$ to discriminate $\vec{X}_m$ is $C_m$

$$C_m = \frac{1}{2}\left(a_m - y_m\right)^2$$

Where we have multiplied by "1/2" to simplify the algebra.

The gradient of the cost with respect to each of the parameters tells us how much each parameter contributed to the error. We will use these to define a vector of correction factors for each parameter.

$$\vec{\nabla} C_m = \frac{\partial C_m}{\partial \vec{w}} = \begin{pmatrix} \frac{\partial C_m}{\partial w_1} \\ \vdots \\ \frac{\partial C_m}{\partial w_D} \\ \frac{\partial C_m}{\partial b} \end{pmatrix} = \begin{pmatrix} \Delta w_1 \\ \vdots \\ \Delta w_D \\ \Delta b \end{pmatrix}$$

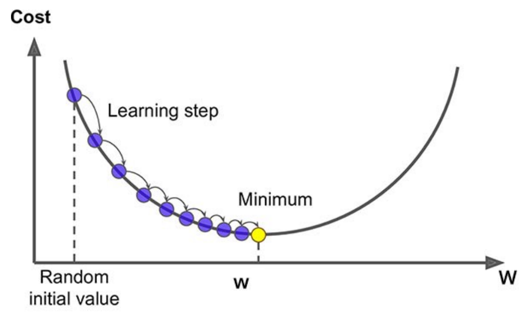
In order to evaluate these derivatives, we use the chain rule. Each gradient term can provides a correction term for the function parameters. For a single neural unit:

$$\Delta w_1 = \frac{\partial C_m}{\partial w_1} = \frac{\partial C_m}{\partial a_m} \cdot \frac{\partial a_m}{\partial z_m} \cdot \frac{\partial z_m}{\partial w_1}$$

To correct the network, we will subtract a fraction of this change from each of the network parameters. Because the training data typically contains many unmodelled phenomena (noise), the correction is weighted by a (very small) learning rate "η" to stabilize learning

$$\vec{w}^{(i)} = \vec{w}^{(i-1)} - \eta \Delta \vec{w}_m$$

The fraction, η, is referred to as the Learning rate. Typical values for η are from η=0.01 to η=0.001.



(Drawing recovered from the internet - Source unknown)

The "optimum" coefficients are the coefficients that provide the smallest loss. To determine the optimum coefficients, we iteratively refine the model to reduce the errors, by subtracting a part of the derivative from the model parameters.

Ideally for the optimum parameters, both the loss and the gradient are zero. For all other parameters, the loss increases. With real data, this will rarely be obtained because of noise in the training data.

Noise (unmodeled phenomena) will drive individual updates in random directions. A small learning rate is used to limit noise from driving the parameters too far from the optimum.

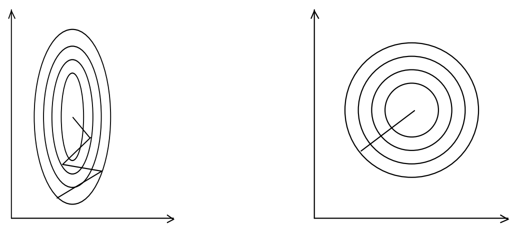
Warning: If you evaluate gradient descent by hand with real data, do not expect to easily see a path to convergence. Typically, arriving at the optimum requires a LOT of training data and MANY passes through the training data. Each pass through the training data is referred to as an "epoch". Gradient descent may require many epochs to reach an optimal (minimum loss) model.

**Feature Scaling**

For a training set $\{\vec{X}_m\}$ of M training samples with D values, if the individual features do not have a similar range of values, than large values will dominate the gradient. Small errors in this dimension are magnified.

One way to assure sure that features have similar ranges is to normalize the training data. A simple technique is to normalize the range of sample values.

For example, $$\forall_{m=1}^{M} : x_{dm} := \frac{x_{dm} - \min(x_d)}{\max(x_d) - \min(x_d)}$$



After estimating the model, use $\max(x_d)$ and $\min(x_d)$ to project the data back to the original space.

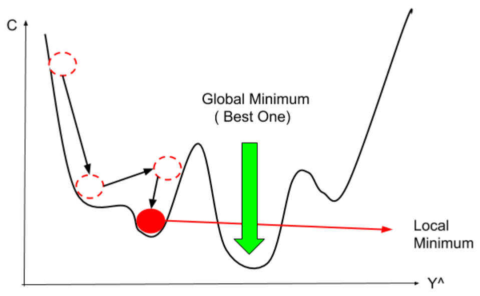
Note that the 2D surface shown here would correspond to two parameters, for example w, b for a single neural unit with a scalar input x. The actual surface is hyper-dimensional and not easy to visualize.

## Local Minima

Gradient descent assumes that the loss function is convex. However, the loss function depends on real data $\vec{X}_m$ with unmodeled phenomena (noise).

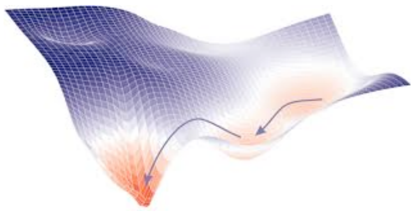
$$C_m = \frac{1}{2}(a_m - y_m)^2$$

Noise in the training samples $\{\vec{X}_m\}$ can create a non-convex loss with local minima.



(Drawing recovered from the internet - Source unknown)

In fact the gradient has MANY parameters, and the Loss function is evaluated in a very high dimensional space. It is helpful to see the data as a hyper-dimensional cloud descending (flowing over) a complex hyper-dimensional surface.



(Drawing recovered from the internet - Source unknown)

## Batch mode

Individual training samples will send the model in arbitrary directions. While, updating with each sample will eventually converge, this tends to be costly. A more efficient approach is to correct the model with the average of a large set of training samples. The training data is typically divided into "folds" and the model is updated with the average of each fold.

This is called "batch mode".

$$\Delta\vec{w} = \frac{1}{M}\sum_{m=1}^{M}\Delta\vec{w}_m = \frac{1}{M}\sum_{m=1}^{M}\vec{\nabla}C_m$$

The model is then updated with the average error.

$$\vec{w}^{(i)} = \vec{w}^{(i-1)} - \eta\Delta\vec{w}$$

## Stochastic Gradient Descent

Batch gradient descent often efficiently converges to a local minimum and becomes stuck. This can be avoided with stochastic gradient descent. With Stochastic gradient descent, a single training sample is randomly selected and used to update the model. This will send the model in random directions, that eventually flow to the global minima. While much less efficient than batch mode, this is less likely to become stuck in local minima.