Pattern Recognition and Machine Learning

James L. Crowley

ENSIMAG 3 - MMIS
Lessons 7

Fall Semester 2019
18 Dec 2019

Perceptrons and Gradient Descent

<u>Outline</u>

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.
$\hat{y} = g(\vec{X},\vec{w})$	A model that predicts $\vec{y}$ from $\vec{X}$.
$\vec{w}$	The parameters of the model.
$\{\vec{X}_m\}$ $\{y_m\}$	Training samples for learning.
M	The number of training samples.
$L(\vec{w}) = \frac{1}{2M}\sum_{m=1}^{M}(y_m - g(\vec{X}_m,\vec{w}))^2$	The average Loss for the function $\hat{y} = g(\vec{X},\vec{w})$
	MSE Loss estimates the cost of errors as the Mean Square Error.

Perceptrons

History

The Perceptron is an incremental learning algorithm for linear classifiers invented by Frank Rosenblatt in 1956. 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.

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 Perceptron Classifier

When the two classes of data can be separated by a linear decision boundary, the data is said to be "separable". The perceptron is an on-line learning algorithm that learns a linear decision boundary (hyper-plane) for separable training data. If the training data is non-separable, the method may not converge, and must be stopped after a certain number of iterations.

Learning can be incremental and "on-line". At any time, new training samples can be used to update the perceptron. The perceptron algorithm uses errors in classifying the training data to update the decision boundary plane until there are no more errors.

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

The algorithm requires a learning rate, η.

A training sample is correctly classified if:

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

The Perceptron Learning Algorithm

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

Algorithm:

```
    $\vec{w}^{(0)} \leftarrow 0$; $b^{(i)} \leftarrow 0$, . $\leftarrow 0$;
    WHILE update DO
        update $\leftarrow$ 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 $\leftarrow$ 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.
```

Where η is a small "learning rate" coefficient (typically 0.01}
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 continue to loop. Thus it is necessary to have a limit the number of iterations.

The fact that the algorithm required separable training data was a major weakness.
This was later overcome by reformulating the algorithm using Gradient descent as explained below.

Gradient Descent

Gradient descent is a popular algorithm for estimating parameters for a large variety of models. Here we will illustrate the approach with estimation of parameters for a linear model $g(\vec{X},\vec{w})$ for a dependent variable y. This can be extended to estimation of a model for a classifier.

Linear Models

The equation $\vec{w}^T\vec{X}+b=0$ is a hyper-plane in a D-dimensional space. $\vec{W}$ is the normal to the plane. b is the perpendicular distance to the origin.

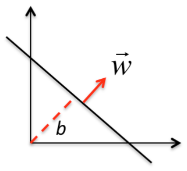
In this case, the linear model has the form

$$y = g(\vec{X},\vec{w}) = \vec{w}^T\vec{X}+b = w_1x_1 + w_2x_2 + \ldots + w_Dx_D + b$$

The vector $\vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{pmatrix}$ and the bias b are the "parameters" of the model $\hat{y}$.

The linear surface $g(\vec{X},\vec{w}) = \vec{w}^T\vec{X}+b$ defines a boundary between the regions where y< 0 and where y > 0.

$\vec{w} = \begin{pmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{pmatrix}$ is the normal to the hyperplane and b is a constant term.



Homogeneous Coordinate Notation

With a linear model, it is possible to include the bias in the model vector by adding an additional term to the model $\vec{w}$ and the feature vector $\vec{X}$. This is referred to as Homogeneous Coordinates.

Homogeneous coordinates provide a unified notation for geometric operations and is widely used in Computer Vision, Computer Graphics and Robotics.

The linear model can be expressed in homogeneous coordinates as:

$$z = \vec{w}^T \vec{X} \qquad \text{where} \quad \vec{X} = \begin{pmatrix} x_1 \\ \vdots \\ x_D \\ 1 \end{pmatrix} \text{ and } \vec{w} = \begin{pmatrix} w_1 \\ \vdots \\ w_D \\ b \end{pmatrix}$$

This gives a linear model for a D dimensional space with *D+1* parameters.

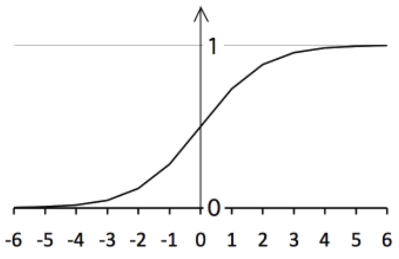
$$z = \vec{w}^T \vec{X} = \begin{pmatrix} w_1 & \cdots & w_D & b \end{pmatrix} \begin{pmatrix} x_1 \\ \vdots \\ x_2 \\ 1 \end{pmatrix}$$

To use this as a discriminant, we need to follow the model with a non-linear decision function, *f(z)*.

Non-linear decision Function

In order to use Gradient descent, we need to replace the decision function in the perceptron with a differentiable function *f(z)*. A popular choice for decision function is the sigmoid function:

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



This function is useful because the derivative is: $\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$

This gives a decision function: if $\sigma(\vec{w}^T \vec{X}) > 0.5$ POSITIVE else NEGATIVE

Loss Function

Assume M samples of training data $\vec{X}_m$ with indicator variables $y_m = \{0, 1\}$.
The vector, $\vec{X}_m$, has D dimensions. The indicator y_m, gives the expected result for the vector. Notice that the 2 class model where $y_m = \{0, 1\}$ can easily be extended to a K class model by replacing the scalar y_m with a binary vector $\vec{y}_m$.

For any training data sample $\vec{X}_m$ for which the true value of the model, $y_m = g(\vec{X}_m, \vec{w})$, is known, the error is the difference between the true value and the estimated value provided by the model.

$$\delta_m = y_m - g(\vec{X}_m, \vec{w})$$

The average error for the training set is

$$\delta = \frac{1}{M}\sum_{m=1}^{M}\delta_m = \frac{1}{M}\sum_{m=1}^{M}(y_m - g(\vec{X}_m, \vec{w}))$$

To estimate $\vec{w}$, we estimate a "Loss" function as the “Mean Square Error” (MSE).

The loss for an individual sample is

$$L(\vec{X}_m, \vec{w}) = \frac{1}{2}\left(y_m - g(\vec{X}_m, \vec{w})\right)^2$$

where we have included the term $\frac{1}{2}$ to simplify the algebra.
The average loss for the all *M* training samples, $\vec{X}_m$, is:

$$L(\vec{w}) = \frac{1}{2M}\sum_{m=1}^{M}\delta_m^2 = \frac{1}{2M}\sum_{m=1}^{M}(y_m - g(\vec{X}_m, \vec{w}))^2$$

The model parameters $\vec{w}$ that minimize the loss are said to be “optimum”.
For all other parameters, the loss increases.

That is the derivative of the loss is zero at the optimum and positive for other values.
Thus we need to find the derivative of the Loss function.
The sgn() function used in Rosenblatt's perceptron is not differentiable.

We seek to estimate that parameters $\vec{w}$ for a model expressed in homogenous coordinates, from a training set of *M* samples $\{\vec{X}_m\}\ \{y_m\}$

To determine the model, we will iteratively refine the model to reduce the errors.
The gradient of the loss is the derivative of the loss with respect to each model parameter.

$$\vec{\nabla}L(\vec{X}_m,\vec{w}) = \frac{\partial L(\vec{X}_m,\vec{w})}{\partial \vec{w}} = \begin{pmatrix} \frac{\partial L(\vec{X}_m,\vec{w})}{\partial w_1} \\ \frac{\partial L(\vec{X}_m,\vec{w})}{\partial w_2} \\ \vdots \\ \frac{\partial L(\vec{X}_m,\vec{w})}{\partial w_D} \end{pmatrix}$$

The gradient tells us how much to correct the model for each training sample, $\vec{X}_m$.

For the optimum model, the gradient is zero.

$$\vec{\nabla}L(\vec{X}_m,\vec{w}) = 0$$

The gradient of the loss is: $\vec{\nabla}L(\vec{X}_m,\vec{w}) = \frac{\partial L(\vec{X}_m,\vec{w})}{\partial \vec{w}} = -\left(y_m - \sigma(\vec{w}^T\vec{X}_m)\right)\vec{X}_m$

which we can also write as $\vec{\nabla}L(\vec{X}_m,\vec{w}) = -\delta_m\vec{X}_m$ because $\delta_m = (y_m - \sigma(\vec{w}^T\vec{X}_m))$

We can use the gradient to "correct" the model parameters for each training sample.

$$\Delta\vec{w}_m = -\delta_m\vec{X}_m$$

The correction is weighted by a very small learning rate "η" to stabilize learning.

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

Batch mode

The values for any individual training sample are random. Individual training samples will send the model in arbitrary directions. While, updating with each sample will eventually converge, but tends to be inefficient. Thus, it is generally more efficient to correct the model with the average of a large set of training samples. This is called "batch mode".

$$\Delta\vec{w} = \frac{1}{M}\sum_{m=1}^{M}\Delta\vec{w}_m = \frac{1}{M}\sum_{m=1}^{M}\delta_m \vec{X}_m$$

we then update the model with the average error. $\vec{w}^{(i)} = \vec{w}^{(i-1)} - \eta\Delta\vec{w}$

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. Even better is to divide the training data into "folds" and update with the average of each fold.

Gradient Descent Algorithm (Batch mode)

Initialization: *(i=0)* and set $\vec{w}^{(0)}$ to some initial (random). values,
for example, for a linear model, using m=1, 2 $w_d^{(0)} = \frac{y_2 - y_1}{x_{d2} - x_{d1}}$, $b^{(0)} = y_2 - y_1$
Choose some value for the learning rate η (typically 0.001)

```
WHILE $\|L(\vec{w}^{(i+1)}) - L(\vec{w}^{(i)})\| > \varepsilon$ DO
    $i \leftarrow i+1$
    $\Delta\vec{w} = \frac{1}{M}\sum_{m=1}^{M}\vec{\nabla}L(\vec{w}^{(i-1)}, \vec{X}_m)$
    $\vec{w}^{(i)} = \vec{w}^{(i-1)} - \eta\Delta w$
    $L(\vec{w}^{(i)}) = \frac{1}{2M}\sum_{m=1}^{M}(y_m - g(\vec{X}_m, \vec{w}^{(i)}))^2$
END
```

The algorithm halts when the change in $\Delta L(\vec{w}^{(i)})$ becomes small: $\|L(\vec{w}^{(i)}) - L(\vec{w}^{(i-1)})\| < \varepsilon$
For some small constant ε. (or after "N" iterations.)

It is common to make many iterations through the training data. Each pass through all the training data is called an epoch.

Practical Considerations for Gradient Descent

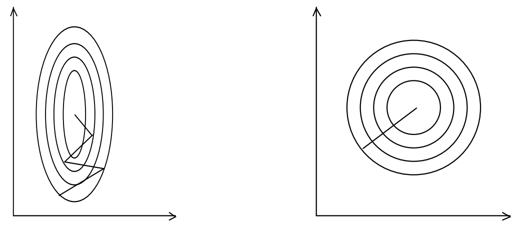
The following are some practical issues concerning gradient descent.

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. Make. 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.

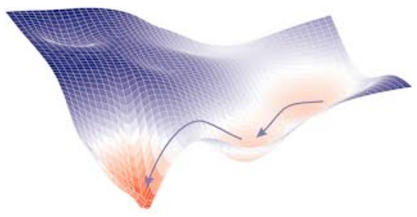
For example, $$\forall_{m=1}^{M} : x_{dm} := \frac{x_{dm} - \min(x_d)}{\max(x_d) - \min(x_d)}$$



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

Stochastic Gradient Descent

Batch gradient descent will generally converge much more efficiently than updating with each sample. However, gradient descent assumes that the loss function is convex. This depends on the training data. In many real examples, the Loss function is not completely convex but contains local minimum.

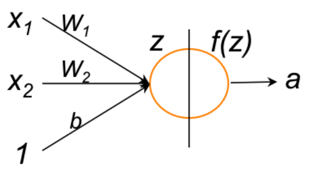


You can see the training data as creating a cloud of possible error vectors. Batch gradient descent steps by the average error from the cloud. While this is efficient, it can be trapped in local minima. While updating independently with each training data may be less efficient, it is less likely to be trapped in a local minima.

Batch gradient descent may efficiently converge to a local minimum and become stuck. This can be avoided with stochastic gradient descent. With Stochastic gradient descent we choose a single training sample randomly and update with that sample.

The Artificial Neuron

Since the 1980's the linear classifier (or perceptron) is referred to as an artificial neuron by the much of the machine learning community.



A artificial neuron is a computational unit that integrates information from a vector of features, $\vec{X}$, to compute the likelihood of a hypothesis, $h_{w,b}(X)$

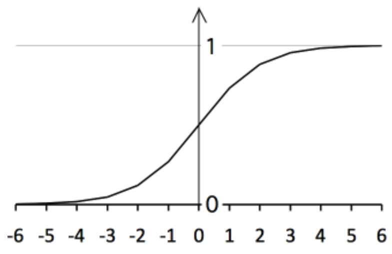
$$a = h_{\vec{w},b}(\vec{X}) = f(\vec{w}^T \vec{X} + b)$$

The neuron is composed of a weighted sum of input values

$$z = w_1 x_1 + w_2 x_2 + \ldots + w_D x_D + b$$

followed by a non-linear "activation" function, $f(z)$ (sometimes written $\phi(z)$)

Many different activation functions may be used.
A popular choice for activation function is the sigmoid: $\sigma(z) = \frac{1}{1+e^{-z}}$



This function is useful because the derivative is: $\frac{df(z)}{dz} = f(z)(1 - f(z))$

This gives a decision function: if $h_{\vec{w},b}(\vec{X}) > 0.5$ POSITIVE else NEGATIVE