Intelligent Systems: Reasoning and Recognition

James L. Crowley

MoSIG M1 — Winter Semester 2019-2020
Lesson 7 — 3 March 2020

Perceptrons and Gradient Descent

Outline

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

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 will not converge, and must be stopped after a certain number of iterations.

The Perceptron is an "on-line" learning algorithm. 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, $\vec{X}_m$,

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

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, η. 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 miss-classified training sample. If the training data are not separable then it will continue to loop forever.

Algorithm:

```
$\vec{w}^{(0)} \leftarrow 0$; $b^{(i)} \leftarrow 0$, $i \leftarrow 0$; set $\eta$ (for example $\eta = 10^{-3}$)
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.
```

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 to loop. Thus it is necessary to have a limit the number of iterations.

The fact that the algorithm requires separable training data is a major weakness.
This was later overcome by reformulating the algorithm using a soft decision surface and Gradient descent as explained below.

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

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

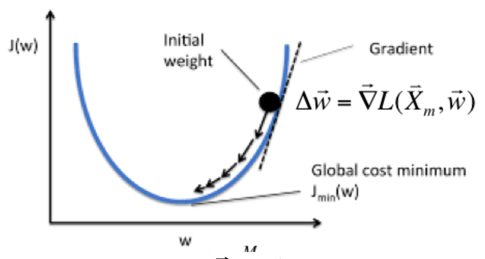
The gradient of a function $f(\vec{X})$ at a point $\vec{X}$ is the direction and rate of fastest increase or decrease (greatest slope). The direction of the gradient is the direction of greatest slope, the magnitude is the slope (rate of change) in that direction.

To use gradient descent to estimate the parameters for a discriminant $y = g(\vec{X},\vec{w})$, we need to define a loss function. A popular loss function is the Sum of squared errors.

A popular loss for a model $g(\vec{X},\vec{w})$ with parameters $\vec{w}$ is the mean square error for a training set of M samples $\{\vec{X}_m\}$ $\{y_m\}$:

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

The gradient is the vector derivative with respect to the model parameters, $\vec{w}$.
To find a local minimum of a function using gradient descent, we update the function by subtracting corrections proportional to the negative of the gradient of the function at the current point. The derivative of the loss is zero at the optimum $\vec{w}$ and positive for other values. We can move to the optimum by subtracting a part of the derivative.



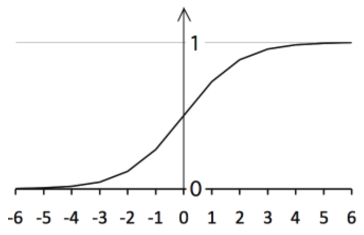
For the best model, the gradient is zero. $\vec{\nabla}L(\vec{X}_m,\vec{w}) = 0$

The model parameters $\vec{w}$ that minimize the loss are said to be "optimum". Note that the parameters are only optimum for the training set. A different training set may give a different optimum.

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 changes the discriminant to $g(\vec{X},\vec{w}) = \sigma\left(\vec{w}^T\vec{X}\right)$

The classifier is now IF $\sigma\left(\vec{w}^T\vec{X}\right) \geq 0.5$ THEN P ELSE N

This changes our target variables to y ∈ {0, 1}.
In this case, y easily generalized to the multiclass case K > 2 by writing y as a binary vector $\vec{y}$, with 1 for the target class and 0 elsewhere.

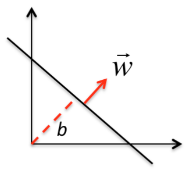
Note that the derivative is: $\frac{d\sigma(z)}{dz} = \sigma(z)(1-\sigma(z))$. We need this for the gradient.

Homogenous Coordinate Notation for Linear Models

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

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

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.



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 $\vec{w}$.

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

The linear model can then be expressed in homogeneous coordinates as: $z = \vec{w}^T \vec{X}$

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

Loss Function

Assume M samples of training data $\vec{X}_m$ with indicator variables $y_m \in \{0, 1\}$.

The vector, $\vec{X}_m$, has D dimensions. The indicator y_m, gives the expected result for the vector. $y_m \in \{0, 1\}$

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.

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

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

The loss for an individual sample is $L(\vec{X}_m, \vec{w}) = \frac{1}{2}\delta_m^2 = \frac{1}{2}\left(y_m - g(\vec{X}_m, \vec{w})\right)^2$

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 Gradient Descent Algorithm for Perceptron Learning

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

The discriminant function (model) is $g(\vec{X},\vec{w}) = \sigma\left(\vec{w}^T\vec{X}\right)$

The classifier is IF $\sigma\left(\vec{w}^T\vec{X}\right) \geq 0.5$ THEN P ELSE N

The error for any sample is: $\delta_m = (y_m - \sigma(\vec{w}^T\vec{X}_m))$

The loss function is the sum of squared errors:

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

To determine the optimum values for the parameters, $\vec{w}$, we will iteratively refine the model to reduce the loss function. 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, $\vec{X}_m$. With a bit of algebra and calculus we can show that 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 by

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

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

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

in the presence of noise.

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.

Batch mode

The individual training sample are random noisy. 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. 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. $\vec{w}^{(i)} = \vec{w}^{(i-1)} + \eta \Delta \vec{w}$

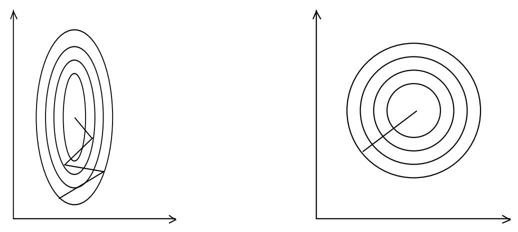
It is common to divide the training data into "folds" and update with the average of each fold.

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.

Gradient Descent Algorithm (Batch mode)

Initialization: *(i=0)* and set $\vec{w}^{(0)}$ to some initial (random) vector.
Choose some value for the learning rate η (typically 0.001)

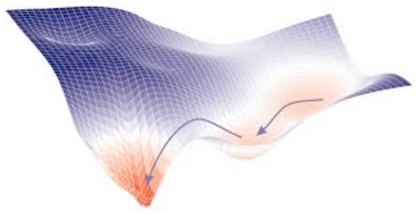
```
WHILE $\|L(\vec{w}^{(i+1)}) - L(\vec{w}^{(i)})\| > \varepsilon$ DO
        $i \leftarrow i+1$
        $\Delta\vec{w}^{(i)} = \frac{1}{M}\sum_{m=1}^{M}\Delta\vec{w}_m^{(i)}$ where $\Delta\vec{w}_m^{(i)} = -\delta_m\vec{X}_m = (y_m - g(\vec{X}_m, \vec{w}^{(i)}))\vec{X}_m$
        $\vec{w}^{(i)} = \vec{w}^{(i-1)}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

Stochastic Gradient Descent

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.



With Stochastic gradient descent we choose a single training sample randomly and update the model with that sample.

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. Updating independently with each training data is less efficient, but less likely to be trapped in a local minima.