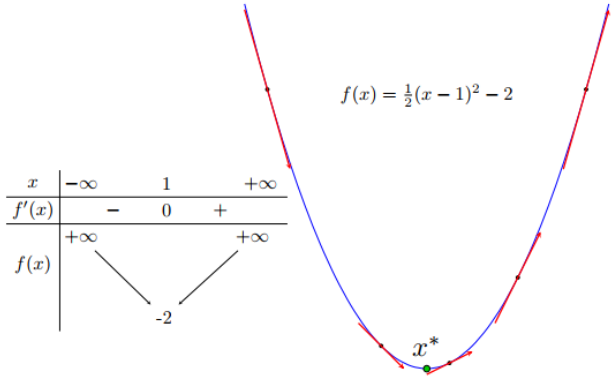
1. Gradient Descent

Trong các bài toán tối ưu, là giải phương trình đạo hàm bằng 0 nhưng trên thực tế phương trình này rất khó để giải ra đúng nghiệm. Do đó, người ta xuất phát từ một điểm gần với nghiệm của bài toán và dùng phép lặp để tiến gần đến nghiệm cần tìm, khi ấy đạo hàm tiến về 0 -> Gradient Descent.

1.1 Gradient Descent cho hàm một biến.



Trong hình trên, tại điểm local minimum (GT cực tiểu) x* cũng là global minimum (GTNN) , đạo hàm là 0, các điểm bên trái có dạo hàm âm (-), bên phải có đạo hàm dương (+) .

Xét hàm số 1 biến f: R -> R, giả sử x_t là điểm tìm được sau t vòng lặp. Ta phải tìm cách đưa x_t về càng gần x* càng tốt.

- Nếu tại x_t , đạo hàm của f > 0 thì x_t nằm bên phải so với x*, để x_t tiến lại gần x*, ta phải dịch x_t về phía bên trái (phía âm – ngược với đạo hàm) và ngược lại:
  $x_{t+1} = x_t + \Delta$ với Δ là một đại lượng ngược dấu với $f'(x_t)$
- x_t càng xa x* về phía bên phải thì $f'(x_t)$ càng lớn hơn 0 nên cần di chuyển nhiều hơn -> lượng di chuyển Δ tỉ lệ thuận với $-f'(x_t)$

Suy ra $x_{t+1} = x_t - \eta f'(x_t)$ trong đó η (eta) là tốc độ học (learning rate)

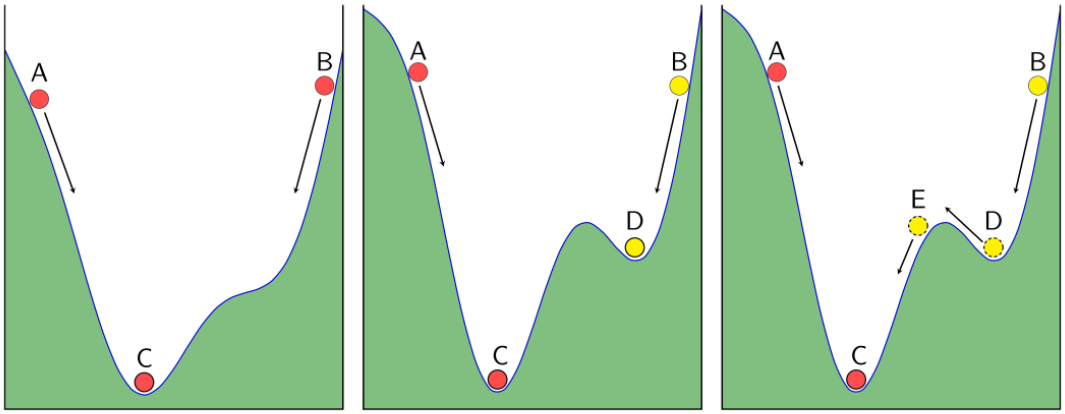
1.2 Gradient Descent cho hàm nhiều biến

Giả sử ta cần tìm global minimun cho hàm số $f(\theta)$ trong đó θ là tập hợp các tham số cần tối ưu. Đạo hàm của hàm số đc kí hiệu là $\nabla\theta f(\theta)$. Tương tự như đạo hàm một biến, GD cho hàm nhiều biến cũng chọn một điểm dự đoán ban đầu θ_0, sau đó dùng vòng lặp để cập nhật đến khi đạo hàm tiến về 0 theo quy tắc

$$\theta_{t+1} = \theta_t - \eta\nabla\theta f(\theta)$$

Hay viết đơn giản hơn $\theta \leftarrow \theta - \eta\nabla\theta f(\theta)$

1.3 GD với momentum



Xét vd sau, trong TH đầu tiên dù chọn điểm ban đầu là A hay B thì cuối cùng hòn bi sẽ lăn xuống C (global minimum)

Trường hợp thứ 2 và thứ 3 tùy vào điểm ban đầu ở A hay B mà kết quả viên bi sẽ là C hay D. Điều này tương tự khi ta áp dụng Gradient Descent, khi bắt đầu ở B thì khi tới D, đạo hàm sẽ bằng 0 vì D là điểm local minimum. Nhưng kết quả mong muốn là ở C -> thuật toán momentum ra đời để giải quyết vấn đề này.

Trong GD, chúng ta cần tính toán lượng thay đổi để cập nhật lại vị trí. Đối với vd trên là vị trí của hòn bi sẽ phụ thuộc vào vận tốc, giả sử mỗi vòng lặp là một đv thời gian thì vị trí mới của hòn bi sẽ là $\theta_{t+1} = \theta_t - v_t$ (dấu – thể hiện việc phải di chuyển ngược với đạo hàm). Để v_t vừa mang tính chất là đà (vận tốc), vừa mang tính chất độ dốc (đạo hàm) thì ta lấy tổng có trọng số của chúng:

$$v_t = \gamma v_{t-1} + \eta \nabla \theta f(\theta)$$

trong đó γ thường được chọn là một giá trị nhỏ hơn gần bằng 1 (~0.9).

Suy ra vị trí mới sẽ là:

$$\theta \leftarrow \theta - v_t = \theta - \eta \nabla \theta f(\theta) - \gamma v_{t-1}$$

1.4 Nesterov Accelerated Gradient

Momentum giúp hòn bi vượt qua được dốc local minimum nhưng khi gần tới đích thì momentum tốn khá nhiều thời gian để dừng lại vì có đà. Để giải quyết vấn đề này ta sử dụng kỹ thuật Nersterov Accelerated Gradient giúp thuật toán hội tụ nhanh hơn.

Thay vì sử dụng gradient của điểm hiện tại $f'(\theta_{t-1})$ để cập nhật vị trí tiếp theo θ_t thì NAG dùng gradient của điểm được dự đoán là xấp xỉ vị trí tiếp theo $f'(\theta_{t-1} - \gamma v_{t-1})$.
Công thức cập nhật:

$$v_t = \gamma v_{t-1} + \eta \nabla \theta f(\theta - \gamma v_{t-1})$$
$$\theta \leftarrow \theta - v_t$$

1.5 Stochastic Gradient Descent

Batch Gradient Descent

Các thuật toán GD được đề cập ở trên còn được gọi là batch GD (batch: tất cả) vì mỗi lần cập nhật các tham số θ thì chúng sử dụng tất cả các điểm dữ liệu của X. Khi cơ sở dữ liệu lớn thì khối lượng tính toán cực kỳ lớn gây tốn thời gian nên một thuật toán đơn giản hơn được sử dụng là Stochastic Gradient Descent (SGD)

Stochastic Gradient Descent

Tại mỗi vòng lặp ta cập nhật θ dựa vào đạo hàm của hàm mất mát tại một điểm dữ liệu x_i.Vì hàm mất mát thường được lấy trung bình trên mỗi điểm dữ liệu nên đạo hàm tại 1 điểm cũng được kỳ vọng là gần với hàm mất mát tại mọi điểm dữ liệu. Áp dụng thuật toán này ta phải chấp nhận sai số nhưng tính toán nhanh hơn.

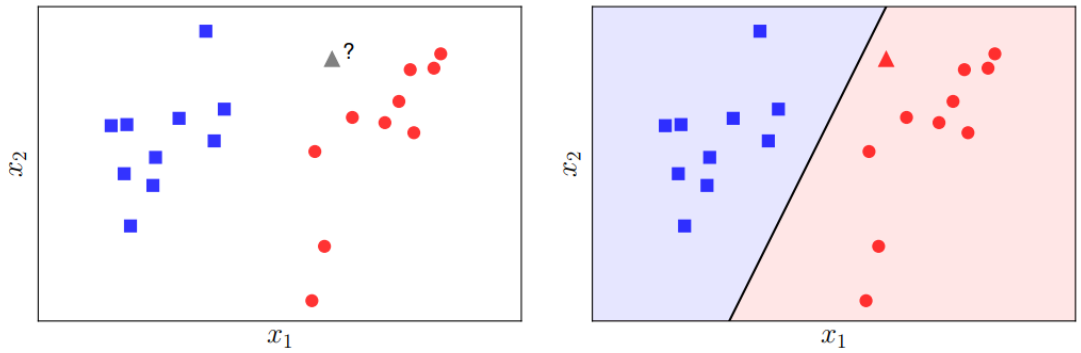
Mini-batch Gradient Descent

Khác với SGD, mini-batch không sử dụng 1 điểm dữ liệu mà sử dụng k điểm dữ liệu cho mỗi lần cập nhật θ (k > 1 và k nhỏ hơn N rất nhiều). Bằng việc xáo trộn ngẫu nhiên dữ liệu rồi chia toàn bộ dữ liệu thành các mini-batch, mỗi mini-batch có k điểm dữ liệu. Để chạy hết 1 lượt, ta dùng N/k vòng lặp.

2. Perceptron Learning Algorithm

2.1 Ý tưởng

Thuật toán Perceptron Learning Algorithm (PLA) được thiết kế cho bài toán phân lớp nhị phân chỉ với 2 lớp dữ liệu.



Trong hình trên, có 2 class đã được gán nhãn tương ứng với màu xanh và màu đỏ, từ đó hãy dự đoán màu cảu một điểm dữ liệu mới là tam giác màu xám. Ở đây, mỗi class chiếm một khôn gian lãnh thổ nhất định, việc cần làm là tìm ranh giới giữa các không gian lãnh thổ đó. Nếu điểm dữ liệu đó thuộc về không gian lãnh thổ nào thì nó sẽ có nhãn tương ứng. Trong VD, tam giác được phân vào lớp đỏ sau khi xác định được ranh giới. PLA chính là thuật toán tìm ranh giới siêu phẳng cho bài toán phân lớp nhị phân (nếu có tồn tại).

2.2 Thuật toán

Cách phân lớp của PLA

Giả sử X = [x1, x2, …, xN] $\in R^{dxN}$ là ma trận chứa các điểm dữ liệu huấn luyện mà mỗi x_i là một điểm dữ liệu trong không gian d chiều. Nhãn tương ứng là vector hàng y = [y1, y2, …, yN] với $y_i = 1$ hoặc $y_i = -1$

Tại một thời điểm, giả sử ta tìm đc ranh giới là một siêu phẳng có phương trình $f_w(x) = w_0 + w_1x_1 + … + w_dx_d = w^Tx = 0$ với $x = [1, x_1, x_2, … x_d]$

Trong không gian 2 chiều như vd trên, giả sử đường thẳng cần tìm là $w_1x_1 + w_2x_2 + w_0 = 0$. Nếu các điểm nằm về cùng 1 phía của đường thẳng này thì hàm số $f_w(x)$ của chúng sẽ cùng dấu.

Giả sử là (+) và (-) tương ứng với các giá trị 1 và -1, nếu w là nghiệm của bài toán Perceptron, thì label của một điểm dữ liệu mới được xác định:

label(x) = 1 nếu $w^Tx >= 0$

label(x) = -1 o.w

Hay label(x) = sgn(w^Tx) (sgn là hàm xác định dấu và giả sử sgn(0) = 1)

Xây dựng hàm mất mát

Xét điểm x_i bất kì với nhãn y_i. Nếu nó bị phân lớp lỗi thì $y_i$sgn(w^Tx) = -1, suy ra tổng số điểm bị phân lớp lỗi là:

$$J_1(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \text{sgn}(\mathbf{w}^T\mathbf{x}_i))$$

Trong đó M là tập hợp các điểm bị phân lớp lỗi ứng với mỗi tham số **w**. Mục đích là tìm **w** sao $J_1(\mathbf{w}) = 0$ (không có điểm bị phân lớp lỗi). Tuy nhiên hàm sgn() là hàm rời rạc, khó tối ưu nên xét hàm sau:

$$J(\mathbf{w}) = \sum_{\mathbf{x}_i \in \mathcal{M}} (-y_i \mathbf{w}^T\mathbf{x_i})$$

Khi một điểm x_i bị phân lỗi nằm càng xa ranh giới thì giá trị $-y_iw^Tx_i$ sẽ càng lớn -> hàm mất mát lớn lên -> điểm lỗi càng xa ranh giới thì mức độ trừng phạt nặng hơn.

Tối ưu hàm mất mát

Bằng cách dùng SGD, với 1 điểm dữ liệu x_i bị phân lớp lỗi, ta có:

$$J(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i\mathbf{w}^T\mathbf{x}_i; \quad \nabla_\mathbf{w} J(\mathbf{w}; \mathbf{x}_i; y_i) = -y_i\mathbf{x}_i$$

Vậy quy tắc cập nhật **w** sử dụng SGD là

$$\mathbf{w} \leftarrow \mathbf{w} - \eta(-y_i\mathbf{x}_i) = \mathbf{w} + \eta y_i\mathbf{x}_i$$

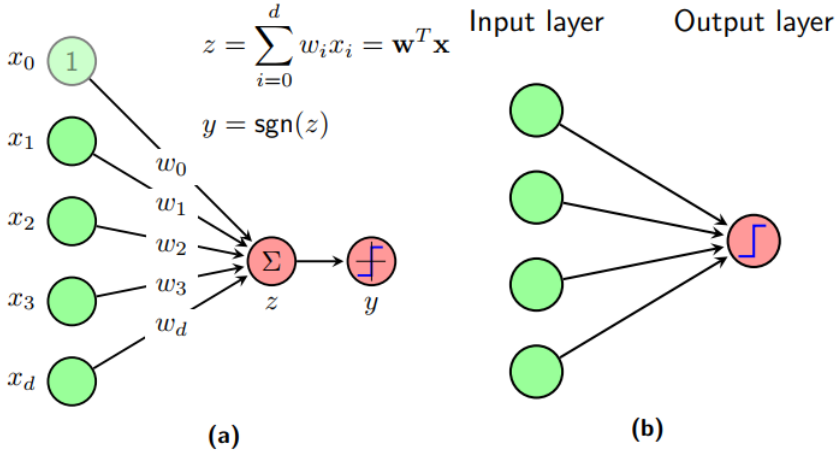
với η là learning rate. Trong PLA, η được chọn bằng 1. Ta có một quy tắc cập nhật rất gọn:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + y_i \mathbf{x}_i$$

Nhận xét

- PLA có thể cho vô số nghiệm khác nhau: nếu 2 lớp là linear separable thì có vô số đường thẳng ranh giới giữa 2 lớp dữ liệu đó. Tuy nhiên với mỗi đường thì kết quả dự đoán của mô hình có thể khác nhau.
- PLA đòi hỏi hai lớp dữ liệu phải linear separable: Khi mỗi lớp có 1 điểm bị nhiễu thì thuật toán này sẽ không dừng lại. Để khắc phục nhược điểm này, có thể dùng pocket algorithm để tìm một nghiệm tốt nhất (ít điểm lỗi nhất):
    1. Giới hạn số vòng lặp của PLA, đặt nghiệm w sau vòng lặp đầu tiên và số điểm lỗi vào pocket.
    2. Mỗi lần cập nhật w_t, ta đếm xem có bao nhiêu điểm bị phân lớp lỗi rồi đưa w_t có số điểm lỗi nhỏ hơn vào pocket. Cách này tương tự như tìm phần tử nhỏ nhất trong mảng 1 chiều.

2.3 Giới thiệu mô hình Neural network đầu tiên



Hàm số dự đoán đầu ra của perceptron label(x) = sgn(w^Tx) có thể được mô tả như trên chính là mô hình đơn giản của neural network.

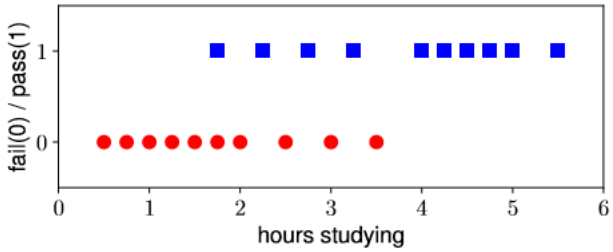
Đầu vào của network **x** được minh họa bằng các node màu lục với node x_0 luôn bằng 1. Tập hợp các nude màu lục gọi là tầng đầu vào (input layer). Các trọng số w_i được gán vào các mũi tên tới node $z = \Sigma w_i x_i = \mathbf{w}^T\mathbf{x}$ (i: 0 -> d). Node y = sgn(z) là output của network và hàm y = sgn(z) gọi là hàm kích hoạt.

Dữ liệu đầu vào được đặt vào input layer, lấy tổng có trọng số lưu vào z rồi đi qua hàm kích hoạt để có kết quả ở y, có thể được giản lược như hình (b) Đây chính là dạng đơn giản nhất của một neural network.

3. Logistic regression

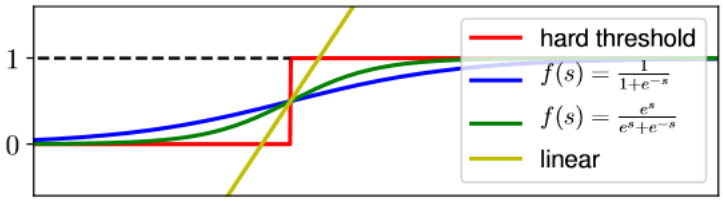
3.1 Giới thiệu

Logistic regression là một mô hình tuyến tính được áp dụng cho các bài toán nhị phân phân lớp và đầu ra có thể được thể hiện dưới dạng xác suất. Ví dụ, xác suất pass exam nếu biết thời gian ôn thi, xác suất trời mưa dựa trên những thông tin đo được,…



Xét ví dụ về thời gian ôn thi và kết quả thi của 20 sinh viên. Bài toán đặt ra là dựa vào thời gian ôn thi để đánh giá khả năng pass exam của sv. Nhìn vào dữ liệu ta có thể thấy thời gian học càng nhiều thì khả năng pass càng cao, tuy nhiên không có một giá trị có định nào để nói rằng khi thời gian học là bao nhiêu đó thì sẽ pass -> không linearly separable -> PLA không dùng được ở đây. Vì vậy thay vì dự đoán chính xác giá trị pass/ fail, ta đi dự đoán xác suất pass là bao nhiêu.

3.2 Mô hình logistic



Quan sát hình trên, ta có vài nhận xét:

- Đường màu vàng biểu diễn hàm kích hoạt tuyến tính, không bị chặn 2 đầu nên không phù hợp với bài toán có đầu ra thuộc khoảng [0,1]
- Đường màu đỏ là hàm kích hoạt của PLA, cũng không thể dùng
- Đường màu lam và lục có vẻ phù hợp với bài toán đang xét:
    1. Là các hàm liên tục trong khoảng [0,1]
    2. Giả sử ngưỡng là 0.5 thì các điểm càng xa ngưỡng về bên trái -> giá trị càng về 0 và ngược lại
    3. Hai hàm này (lam: **sigmoid**) có đạo hàm tại mọi nơi

Hàm sigmoid và tanh

Trong số các hàm số có ba tính chất nói trên, hàm *sigmoid*:

$$f(s) = \frac{1}{1+e^{-s}} \triangleq \sigma(s)$$

được sử dụng nhiều nhất, vì nó bị chặn trong khoảng $(0, 1)$. Thêm nữa,

$$\lim_{s\to-\infty} \sigma(s) = 0; \quad \lim_{s\to+\infty} \sigma(s) = 1$$

Thú vị hơn,

$$\sigma'(s) = \frac{e^{-s}}{(1+e^{-s})^2} = \frac{1}{1+e^{-s}}\frac{e^{-s}}{1+e^{-s}} = \sigma(s)(1-\sigma(s))$$

Với đạo hàm đơn giản, hàm sigmoid được sử dụng rộng rãi trong neural network.

Ngoài ra, hàm *tanh* cũng hay được sử dụng: $\tanh(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$.

Hàm mất mát

Với mô hình màu lam và lục như trên, ta giả sử xác suất để một điểm x rơi vào lớp thứ nhất là $f(w^Tx)$ và rơi vào lớp thứ 2 la $1 - f(w^Tx)$

$$p(y_i = 1|x_i, w) = f(w^Tx_i)$$

$$p(y_i = 0|x_i, w) = 1 - f(w^Tx_i)$$

Việc cần làm là tìm w sao cho với các điểm dữ liệu có nhãn $y_i = 1$ thì $f(w^Tx_i)$ gần với 1 và ngược lại. Đặt $z_i = f(w^Tx_i)$, ta có thể viết lại:

$$p(y_i|\mathbf{x}_i; \mathbf{w}) = z_i^{y_i}(1-z_i)^{1-y_i}$$

Xét bài toán với ma trận dữ liệu $X = [x_1, x_2, \ldots, x_N] \in R^{dxN}$ và vector đầu ra tương ứng $\mathbf{y} = [y_1, y_2, \ldots, y_N]$. Ta cần tối ưu bài toán:

$$w = \underset{w}{\text{argmax}}\ p(y|X,w)$$

Đây chính là bài toán Maximum Likelihood Estimation với tham số mô hình w cần được ước lượng. Giả sử các điểm dữ liệu là ngẫu nhiên dộc lập ta được

$$p(\mathbf{y}|\mathbf{X}; \mathbf{w}) = \prod_{i=1}^{N} p(y_i|\mathbf{x}_i; \mathbf{w}) = \prod_{i=1}^{N} z_i^{y_i}(1-z_i)^{1-y_i}$$

Lấy logarit tự nhiên**, đổi dấu** và lấy trung bình la thu được hàm số

$$J(\mathbf{w}) = -\frac{1}{N}\log p(\mathbf{y}|\mathbf{X}; \mathbf{w}) = -\frac{1}{N}\sum_{i=1}^{N}(y_i \log z_i + (1-y_i)\log(1-z_i))$$

Với z_i là hàm số của w và x_i. Hàm J(w) chính là hàm mất mát của logistic regreesion, ta cần đi tìm w để J(w) đạt **GTNN**

Tối ưu hàm mất mát

Để tối ưu hàm mất mát của regression có thể dùng SGD. Tại mỗi vòng lặp, w sẽ cập nhật dựa vào một điểm dữ liệu (x_i, y_i) ngẫu nhiên, khi đó:

$$J(\mathbf{w}; \mathbf{x}_i, y_i) = -(y_i \log z_i + (1 - y_i) \log(1 - z_i))$$
$$\nabla_{\mathbf{w}} J(\mathbf{w}; \mathbf{x}_i, y_i) = -(\frac{y_i}{z_i} - \frac{1 - y_i}{1 - z_i})(\nabla_{\mathbf{w}} z_i) = \frac{z_i - y_i}{z_i(1 - z_i)}(\nabla_{\mathbf{w}} z_i)$$

Qua một số bước tính toán, ta tìm được:

$$\nabla_w J(w, x_i, y_i) = (\sigma(w^T x_i) - y_i)x_i$$

Suy ra công thức cập nhật nghiệm của nó là:

$$w \leftarrow w - \eta(\sigma(w^T x_i) - y_i)x_i \qquad \textit{với } \eta \textit{ là learning rate dương}$$

Logistic regression với weight decay

Để tránh overfiiting, các neural network thường sử dụng weight deay (sử dụng một đại lượng tỉ lệ với bình phương norm 2 của vector hệ số w được công vào hàm mất mát để hạn chế độ lớn của hệ số). Hàm mất mát trở thành:

$$\bar{J}(\mathbf{w}) = \frac{1}{N} \sum_{i=1}^{N} \left(-y_i \log z_i - (1 - y_i) \log(1 - z_i) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2 \right)$$

CT cập nhật cũng thay đổi:

$$w \leftarrow w - \eta((\sigma(w^T x_i) - y_i)x_i - \lambda w)$$

3.3 Tính chất của Logistic Regression

- Logistic Regression thường được sử dụng trog các bài toán phân lớp. Sau khi tìm được xác suất của mỗi lớp, ta chỉ việc xác định lớp nào có xác suất cao hơn thì điểm dữ liệu mới thuộc về lớp đó. Thông thường lấy 0.5 làm ngưỡng.
- Đường ranh giới tạo bởi Logistic Regreesion là một siêu phẳng: Giả sử những điểm có đầu ra lớn hơn 0.5 thuộc lớp 1, những điểm còn lại thuộc lớp 0. Như vậy có thể nói lớp 1 tạo thành 1 nửa không gian $w^T x > 0$, lớp 0 thuộc về nửa còn lại. Ranh giới giữa hai class này là siêu phẳng $w^T x = 0$ -> Phân lớp tuyến tính.
- Logistic không yêu cầu giả thiết linearly separable
- Ngưỡng ra quyết định có thể thay đổi trong các bài toán thực tế

3.4 Dùng Logistic Regression (phân lớp nhị phân) cho phân lớp đa lớp

One-vs-one

Xây dựng nhiều bộ lớp nhị phân cho từng cặp dữ liệu (LR12, LR13, …). Dữ liệu mới được đưa vào tất cả các bộ phân lớp nhị phân trên, sau cùng kq là lớp có số điểm dữ liệu (hoặc xác suất) được phân vào nhiều nhất. Cách làm này tốn nhiều thời gian, và phải qua nhiều bộ nhị phân không cần thiết.

Phân tầng

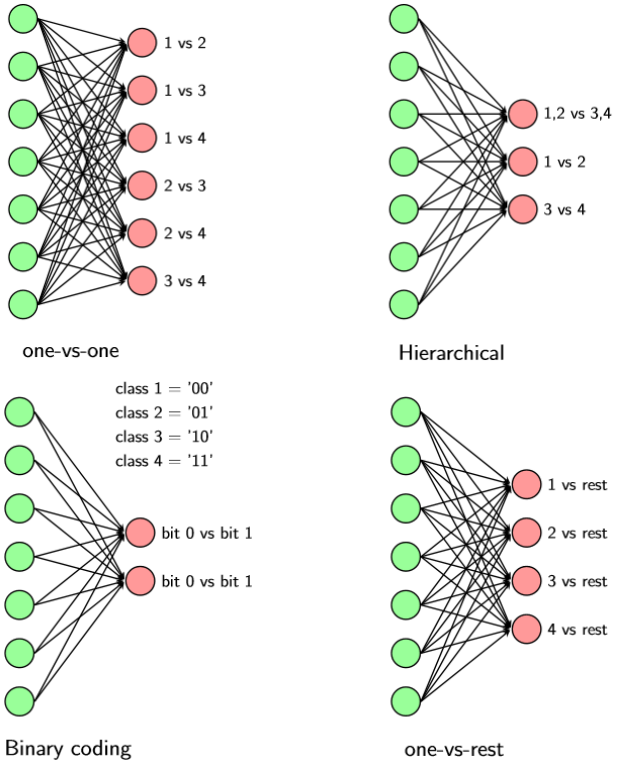
Ta có thể thấy ý tưởng qua vd sau:
Giả sử cần phân 4 lớp dữ liệu chữ số 4, 5, 6, 7. Đầu tiên ta xây dựng bộ phân tầng (4,7) vs (5,6) vì chúng có những nét tương đồng. Sau đó dựng thêm 2 bộ phân tầng 4 vs 7 và 5 vs 6. Cách này đơn giản hơn One-vs-One nhưng sai sót cao hơn bởi khi bị sai ở một tầng nào đó thì chắc chắn những tầng phía dưới sẽ sai.

Birnary coding

Mã hóa output của mỗi lớp bằng mã nhị phân, bộ nhị phân thứ nhất đi tìm bit thứ nhất, bộ thứ tìm bit thứ 2, …Cách làm này giảm đáng kể số bộ lớp nhị phân nhưng yêu cầu số lớp phải là lũy thừa của 2, nếu không sẽ nhận được giá trị không thuộc lớp nào

One-vs-rest

Kỹ thuật này được sử dụng nhiều nhất. Với C lớp thì ta sẽ đi xây dựng C bộ phân lớp nhị phân. Bộ thứ C_i phân biết lớp thứ i với các lớp còn lại. Kq là lớp mà điểm rơi vào có xác suất cao nhất



4. Softmax Regreesion

4.1 Giới thiệu

Ở bài toán phân lớp nhị phân sử dụng logistic regreesion, đầu ra là 1 số trong khoảng (0,1) đóng vai trò là xác suất để đầu vào là 1 trong 2 lớp. Ý tưởng này được mở rộng cho bài toán phân lớp đa lớp trong mô hình softmax regreesion với đầu ra có C node tương ứng với C node ở đầu vào và giá trị mỗi node là xác suất để đầu vào rời vào từng lớp. Lúc này, tham số mô hình sẽ là tập hợp cá tham số w_i ứng với từng node, ta được ma trận trọng số $W = [w_1, w_2,\ldots, w_C]$ ứng với mỗi node ở output layer

4.2 Phân tích toán học

Công thức của softmax function

Ta cần một mô hình xác suất sao cho với mỗi input x, nhận được các a_i là xác suất để input rơi vào lớp thứ i. Điều kiện ở đây là các a_i phải dương và có tổng bằng 1.

Với $z_i = (w_i)^T x$ càng lớn thì xác suất dữ liệu rơi vào lớp i càng cao và z_i có thể nhận giá trị âm nên ta có thể dùng hàm sau:

$$a_i = \frac{\exp(z_i)}{\sum_{j=1}^{C} \exp(z_j)}, \quad \forall i = 1, 2, \ldots, C$$

Hàm này gọi là softmax function. Lúc này có thể coi rằng:

$$p(y_k = i|x_k;W) = a_i$$

(xs để x_k rơi vào lớp i mà a_i với W là tham số ma trận)

Khi các z_i quá lớn, việc tính toán có thể xảy ra hiện tượng tràn số, nên ta có thể biến đổi biểu thức ban đầu thành:

$$\frac{\exp(z_i)}{\sum_{j=1}^{C} \exp(z_j)} = \frac{\exp(-c)\exp(z_i)}{\exp(-c)\sum_{j=1}^{C} \exp(z_j)} = \frac{\exp(z_i - c)}{\sum_{j=1}^{C} \exp(z_j - c)}$$

Với c là hằng số bất kỳ, trong thực nghiệm ta chọn c = maxz_i

Cross entropy

Với mỗi đầu vào x, qua softmax network sẽ cho ra đầu ra dự đoán là vector a, trong khi **đó đầu ra thực sự** là **vector ở dạng one-hot** y.

Hàm mất mát của softmax regression được xây dựng dựa trên bài toán tối thiểu sự khác nhau giữa đầu ra dự đoán a và đầu ra thực sự y. Ta dùng cross entropy để đo khoảng cách giữa 2 vector xác suất. Giá trị của Cross Entropy đạt GTNN khi 2 vector bằng nhau và càng lớn khi 2 vector càng lệch nhau.

Cross Entropy giữa 2 vector phân phối rời rạc p và q được định nghĩa:

$$H(\mathbf{p}, \mathbf{q}) = -\sum_{i=1}^{C} p_i \log q_i$$

Hàm mất mát

Trong TH có C lớp dữ liệu, mất mát giữa đầu ra dự đoán và đầu ra thực sự của một điểm sữ liệu x_i với label (one-hot) y_i được tính là:

$$J_i(\mathbf{W}) \triangleq J(\mathbf{W}; \mathbf{x}_i, \mathbf{y}_i) = -\sum_{j=1}^{C} y_{ji} \log(a_{ji})$$

Qua một số bước tính toán, ta được

$$J(\mathbf{W}; \mathbf{X}, \mathbf{Y}) = -\frac{1}{N} \sum_{i=1}^{N} \log(a_{y_i,i})$$

Với $a_{yi,I}$ là phần từ thứ y_i của vector a_i. (y_i là chỉ số để giá trị để $y_{ij} = 1$)

Để tránh overfitting, ta them vào một đại lượng tỉ lệ với norm 2 của W.

$$\bar{J}(\mathbf{W}; \mathbf{X}, \mathbf{Y}) = -\frac{1}{N} \left(\sum_{i=1}^{N} \log(a_{y_i,i}) + \frac{\lambda}{2} \|\mathbf{W}\|_F^2 \right)$$

Tối ưu hàm mất mát

Sử dụng Mini-batch Gradient Descent để tối ưu hàm mất mát này. Từ hàm mất mát ở trên, lấy đạo hàm, sau vài bước tính toán ta được:

$$\nabla_{\mathbf{W}} J(\mathbf{W}) = \frac{1}{N} \sum_{i=1}^{N} \mathbf{x}_i \mathbf{e}_i^T = \frac{1}{N} \mathbf{X} \mathbf{E}^T$$

Với E = A – Y. Suy ra công thức cập nhật:

$$\mathbf{W} \leftarrow \mathbf{W} - \frac{\eta}{N_b} \mathbf{X}_b \mathbf{E}_b^T$$

Với N_b là kích thước mỗi batch.