* Gradient-Based Optimization

minimize, maximize $\Big\}$ $f(x)$ ⇝ Continuous, Smooth

$f(x)$ ↳ Objective function (criterion)

→ $y = f(x)$, $y, x \in \mathbb{R}$

derivative $f'(x) = \frac{dy}{dx}$ ⇝ Slope of $f(x)$ at x

$f(x+\epsilon) \simeq f(x) + \epsilon f'(x)$ ⇝ How to improve y by changing x

we know $f(x - \epsilon\, \text{sign}(f'(x))) < f(x)$ for $\ll \epsilon$

Reduce $f(x)$ by moving x in small steps with the opposite sign of the derivative

* $f'(x) = 0$ ⇝ no information where to move

↳ Critical points

For a function with multiple inputs $f: \mathbb{R}^n \to \mathbb{R}$ ⇝ Minimization: There must be only one scalar output

Partial derivatives $\frac{\partial}{\partial x_i} f(x)$ → measures the changes in x_i

Gradient generalizes the notion of derivative with respect to a vector

the gradient of f is a vector containing all partial derivatives

$$\nabla_x f(x)$$

Critical points $\nabla f(x) = 0$

Directional Derivative in the direction u (unit vector) → slope of function f in the direction of u, the derivative:

$$f(x + \alpha u)$$

evaluated at $\alpha = 0$ using chain rule

$$\frac{\partial}{\partial \alpha} f(x + \alpha u) \text{ evaluates } u^T \nabla_x f(x)$$

To minimize f we want to find the direction in which f decreases the fastest

$$\min_{u, u^T u = 1} u^T \nabla_x f(x) = \min_{u, u^T u = 1} \|u\|_2 \|\nabla_x f(x)\|_2 \cos\theta$$

θ → angle between u and ∇

$$\|u\|_2 = 1$$

and ignoring everything that do not depend of u → $\min_u \cos\theta$

we can decrease f by moving in the direction of the negative gradient

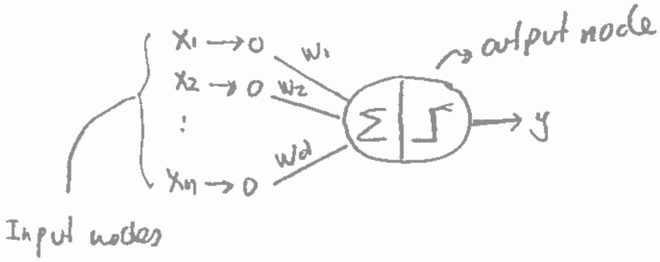
<u>steepest descent</u> or gradient <u>descent</u>

$$x' = x - \epsilon \nabla_x f(x)$$

↳ learning rate

* The basic architecture of Neural Networks

⇝ Single Computational Layer: The perceptron → single input layer and an output node



Training instance ⇝ $(\bar{X}, y)$

* where $\bar{X} = \underbrace{[x_1, x_2, \ldots x_n]}_{d \text{ features variables}}$

* $y \in \{+1, -1\}$ ⇝ observed value (binary class)

Examples: Credit card fraud
Spam emails
...

* Input layer → contains d nodes ⇝ does not perform any computation
↳ Transmit the features from $\bar{X}$

* edges weight $\bar{W} = [w_1, w_2, \ldots w_d]$ → connected to output node

* Output node: computes the linear function

$$\bar{W} \cdot \bar{X} = \sum_{i=1}^{d} w_i x_i$$

this is the activation function

↳ then the $\underbrace{\text{sign function}}_{\int}$ is used to predict the dependent variable of $\bar{X}$

↳ Then the prediction $\hat{y}$ is computed

$$\underbrace{\hat{y} = \text{sign}\{\bar{W} \cdot \bar{X}\} = \text{sign}\left\{\sum_{j=1}^{d} w_j x_j\right\}}_{\text{maps a } \mathbb{R} \text{ value to } \{-1, +1\}}$$

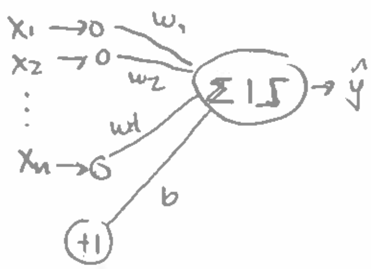
The error of prediction is then $E(\bar{X}) = \underset{\substack{\downarrow \\ \text{observable}}}{y} - \underset{\text{Prediction}}{\hat{y}}$ ⇝ $\{-2, 0, +2\}$

↳ Depending on the error the weights will be updated in the direction of the error gradient.

→ Consider a setting in which the feature variables are mean centered but the mean of the binary class prediction from $\{-1,+1\}$ is not $\emptyset$.

i.e Class distribution imbalanced

In this case we need to include an invariant → Bias term



$$\hat{y} = \text{sign}\{\bar{W}\cdot\bar{X}+b\} = \text{sign}\left\{\sum_{j=1}^{d} w_j x_j + b\right\}$$

* Bias neuron !

** The goal ~ minimize the error in prediction.

↳ This implies minimize number of missclassifications

↳ (least squares form)

Lets have a dataset D

↳ Training instances $(\bar{X}, y)$

$$\underbrace{\text{Minimize}_{\bar{W}}\ L = \sum_{(\bar{X},y)\in D} (y-\hat{y})^2 = \sum_{(\bar{X},y)\in D} (y - \text{sign}\{\bar{W}\cdot\bar{X}\})^2}_{\text{Loss function}}$$

sign function is not differentiable

~ This is not suitable for gradient-descent methods

Therefore the perceptron algorithm uses a smooth approximation of the gradient

$$\nabla L_{smooth} = \sum_{(\bar{X},y)\in D} (y-\hat{y})\bar{X}$$ ~ not a true gradient

when the data point $\bar{X}$ is fed into the network the weight vector is updated.

$$\bar{W} \Leftarrow \bar{W} + \alpha(y-\hat{y})\bar{X}$$

↓

Regulates the learning rate of the NN

} Cicles through all training samples in random order

↓

Each cycle is refered as epochs

$$\bar{W} \Leftarrow \bar{W} + \alpha E(\bar{X})\bar{X}$$ ~ stochastic gradient descent method

mini-batch $\bar{W} \Leftarrow \bar{W} + \sum_{\bar{X}\in S} E(\bar{X})\bar{X}$

The type model proposed in the perceptron is a <u>linear model</u>

$\bar{W} \cdot \bar{X} = 0$ ⇝ defines a linear hyperplane

$\bar{W} = (w_1, w_2 \dots w_d)$ ⇝ d-dimensional vector normal to the hyperplane

$\bar{W} \cdot \bar{X}$ → positive for values of $\bar{x}$ on one side of the hyperplane
↳ negative for values on the other side

Data should be linearly separable

How to obtain a differentiable function to optimize?

Let's write the error function of 0/1 loss function for a data point $(\bar{X}_i, y_i)$

$$L_i^{(0/1)} = \frac{1}{2}(y_i - \text{sign}\{\bar{W} \cdot \bar{X}_i\})^2 = 1 - y_i \, \text{sign}\{\bar{W} \cdot \bar{X}_i\}$$

↳ causes non-differentiability
↓
Drop the sign function and set negative values to $\emptyset$ leading to

$$L_i = \max\{-y_i(\bar{W} \cdot \bar{X}_i), 0\}$$

and updates

$$\bar{W} \Leftarrow \bar{W} - \alpha \nabla_W L_i$$

$\left.\begin{array}{l} y_i^2 \\ \text{sign}\{\bar{W} \cdot \bar{X}_i\}^2 \end{array}\right\}$ evaluate to 1

** Choice of activation and Loss functions

Depending on the target type different activation functions should be implemented. they could be

* linear
* non-linear

$\hat{y} = \Phi(\bar{W} \cdot \bar{X})$ ⇝ activation function

a neuron computes 2 functions

$\bar{X} \{ \xrightarrow{\bar{W}} (\Sigma \mid \Phi) \rightarrow h = \Phi(\bar{W} \cdot \bar{X})$

$\bar{X} \{ \xrightarrow{\bar{W}} (\Sigma) \underbrace{a_h = \bar{W} \cdot \bar{X}}_{\text{Pre-activation value}} (\Phi) \rightarrow h = \Phi(a_h)$ ⇝ Post-activation value

⇝ This can be used in different type of analysis (i.e. backpropagation)

Activation Functions

1. (linear) $\Phi(v) = v$ ⇝ output real value.

2. (sign) $\Phi(v) = \text{sign}(v)$ ⇝ maps binary outputs

3. (sigmoid) $\Phi(v) = \frac{1}{1+e^{-v}}$ ⇝ $(0,1)$ interpreted as probabilities

4. (tanh) $\Phi(v) = \frac{e^{2v}-1}{e^{2v}+1}$ ⇝ similar shape as sigmoid translated/re-scaled $[-1,1]$

↓

outputs both positive and negative

$\tanh(v) = 2 \cdot \text{sigmoid}(2v) - 1$

Piece-wise linear activation functions

5. (Rectified Linear Unit) $\Phi(v) = \max\{v, \phi\}$
6. (Hard tanh) $\Phi(v) = \max\{\min[v,1], -1\}$

} Useful for multi-layer networks

** Choice and number of output nodes

→ Tied to the activation function

i.e K classifications → K outputs can be used ⇝ outputs $\bar{v} = \{v_1, v_2, \ldots, v_K\}$

(softmax) $\Phi(\bar{v})_i = \frac{\exp(v_i)}{\sum_{j=1}^{K} \exp(v_j)}$ $\forall i \in \{1, \ldots K\}$ ⇝ Probabilities of K-classes

** Choice of loss function

Depends on output

→ least-squares regression with numeric output → (squared loss) $L = (y - \hat{y})^2$

→ hinge-loss $y \in \{-1, +1\}$; Re valued prediction $\hat{y}$ → $L = \max\{0, 1 - y \cdot \hat{y}\}$ (SVMs)

Multi-way

1. Binary targets (Logistic Regression) → $L = \log(1 + \exp(-y \cdot \hat{y}))$

↳ alt sigmoid

2. Categorical Targets. $L = -\log(\hat{y}_r)$

↳ per instance