## Lecture 2: Classification. Perceptron. Sigmoid classifiers.

- Classification problems. Error functions
- Perceptron
- Sigmoid classifiers



## Classification

- Given a data set $D \subset \mathcal{X} \times \mathcal{Y}$ where $\mathcal{Y}$ is a discrete set (usually with a smallish number of values), find a hypothesis $h \in \mathcal{H}$ which predicts "well" the existing data
- If $\mathcal{Y}$ has two possible values, e.g. $\mathcal{Y} = \{-1, 1\}$ or $\mathcal{Y} = \{0, 1\}$, this is called binary classification.
- Can we develop methods for classification as we did for regression?
- What does it take to develop a learning algorithm?

## Recall: Three decisions

- What should be the error function?
- What should be the hypothesis class?
- How are we going to find the best hypothesis in the class (the one that minimizes the error function)?



## Error functions for binary classification

- One worthy goal is to *minimize the number of misclassified examples*
- Suppose $\mathcal{Y} = \{-1, 1\}$ and the hypotheses $h_{\mathbf{w}} \in \mathcal{H}$ also output a $+1$ or $-1$
- An example $\langle \mathbf{x}, y \rangle$ is misclassified if $y h_{\mathbf{w}}(\mathbf{x})$ is negative.
- So a reasonable error function is just counting the number of examples correctly classified:

$$J(\mathbf{w}) = - \sum_{i \in \text{Misclassified}} y_i h_{\mathbf{w}}(\mathbf{x_i})$$

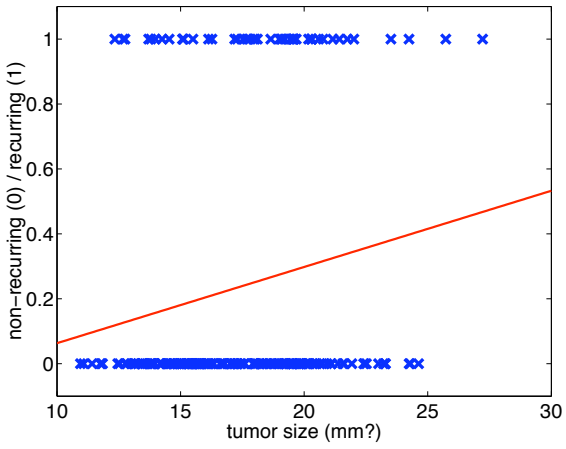
  This is called *0-1 loss*
- This function is not differentiable, so often we will still use the mean-squared error.

## Choosing the hypothesis class

- For regression, we used linear hypotheses (simple, nice)
- Is there an analogue for classification?
- What about linear hypotheses?



## Example: Wisconsin data



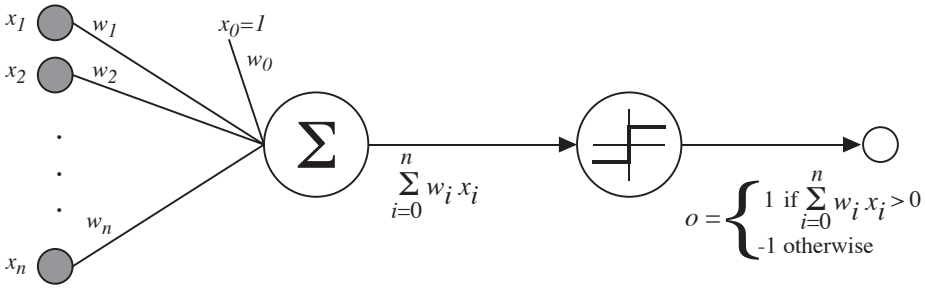

What is the meaning of the output in this case?

## Output of a classifier

- Useful predictions could be:
  - The predicted class
  - The probability that the example belongs to a given class
- Just applying linear regression as is gives us *neither*



## Perceptron

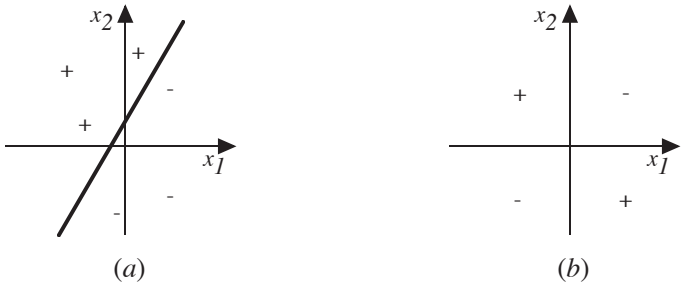



- We can take a linear combination and threshold it:

$$h_{\mathbf{w}}(\mathbf{x}) = \text{sgn}(\mathbf{w}^T\mathbf{x}) = \begin{cases} +1 & \text{if } \mathbf{w}^T\mathbf{x} > 0 \\ -1 & \text{otherwise} \end{cases}$$

  This is called a *perceptron*.
- The output is taken as the predicted class.

## Decision surface of a perceptron




- The decision surface is a line (examples on each side are classified differently)
- Represents some useful functions.
  Example: what weights represent $AND(x_1, x_2)$?
- But some functions not linearly separable! E.g. XOR.
  To represent them, we would need networks of perceptron-like elements.



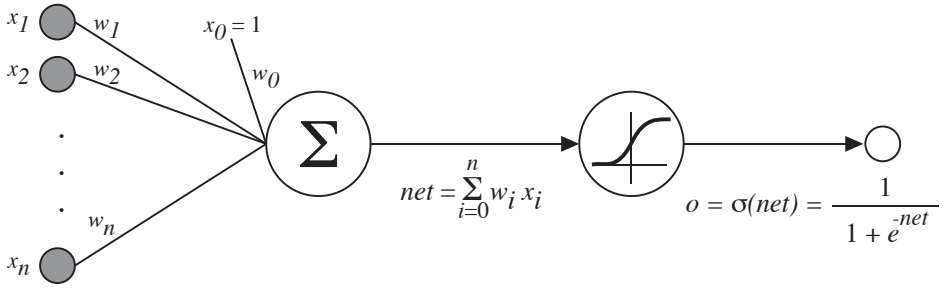
## How do we find the best $\mathbf{w}$?

- "Simulate" stochastic gradient descent on the 0-1 loss function:
  - Initialize $\mathbf{w}$ somehow.
  - While some misclassified samples remain:
    1. Choose a misclassified sample, $i$.
    2. $\mathbf{w} \leftarrow \mathbf{w} + \alpha y_i \mathbf{x}_i$, where $\alpha$ is a step-size parameter.
- If the data is linearly separable, then under the appropriate conditions on $\alpha$ this converges to a $\mathbf{w}$ with zero error.
- If the data is not linearly separable, the weights oscillate
- Can we actually come up with a hypothesis for which we can do honest gradient descent?

## Sigmoid unit

Idea: We want a *soft threshold*:

- Nicer math
- Closer to biological neurons




$\sigma(x)$ is the sigmoid function: $\frac{1}{1+e^{-x}}$

Nice property: $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$



## Logistic (sigmoid) hypothesis

$$h_{\mathbf{w}}(\mathbf{x}) = \sigma(\mathbf{w}^{\mathbf{T}}\mathbf{x}) = \frac{1}{1 + e^{-\mathbf{w}^{\mathbf{T}}\mathbf{x}}}$$

- We will want to determine a "good" weight vector $\mathbf{w}$
- To make that precise, we want a vector $\mathbf{w}$ that minimizes the sum-squared error in the prediction: output should be as close as possible to $0$ for examples of class $0$, and similarly for class $1$

## Minimizing sum-squared error

- Error function is $J(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}(y_i - h_{\mathbf{w}}(\mathbf{x_i}))^2$
- The gradient is given by:

$$\nabla J = -\sum_{i=1}^{m}(y_i - h_{\mathbf{w}}(\mathbf{x_i}))\nabla h_{\mathbf{w}}(\mathbf{x_i})$$

- For sigmoid hypotheses, we have:

$$\nabla h_{\mathbf{w}}(\mathbf{x_i}) = h_{\mathbf{w}}(\mathbf{x_i})(1 - h_{\mathbf{w}}(\mathbf{x_i}))\mathbf{x_i}$$

- We now have a *non-linear system of equations* in $\mathbf{w}$
- There is no nice, closed-form solution like in the case of linear hypotheses...



## Problems

- If the hypothesis class $\mathcal{H}$ is complicated, we may not be able to solve for the optimal parameter vector $\mathbf{w}$.
- Even in the case of linear regression, if the data set is large, we may not be able to compute $(X^T X)^{-1}$, because the matrix is not invertible, or this process may be very slow
- We could sub-sample the data (select fewer attributes and instances), but this will lose information
- A more general procedure for optimizing an error function: **gradient descent**

## Finding the zeros of a function

- Suppose you have a function $f : \Re \rightarrow \Re$ and we want to find an extremum $u^*$
- At the extremum, $f'(u^*) = 0$, but we may not be able to solve this analytically
- We start with an initial point $u_0$
- We can compute the derivative $f'(u_i)$ at the current point $u_i$
- We step in the direction that moves the derivative towards $0$:

$$u_{i+1} = u_i - \alpha_i f'(u_i)$$

  where $\alpha_i$ is a parameter
- If $f$ is smooth, and we are careful about choosing $\alpha_i$, this is guaranteed to take us to a point where $f'(u_i) = 0$

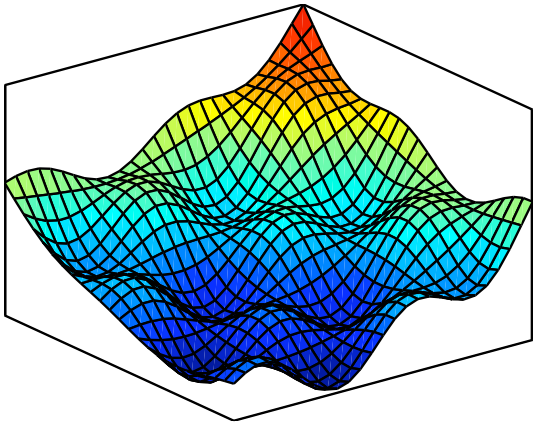


## Application to machine learning

- The function $f$ is an *error function*
- This procedure will take us to a *local optimum* (minimum or maximum)
- In a few special cases, e.g. linear regression, the error function has only one global minimum, but such cases are rare

## Gradient descent: Multivariate setting

- The gradient of $f$ at a point $\langle u_1, u_2, \ldots, u_n \rangle$ can be thought of as a vector indicating which way is "uphill".



- If this is an error function, we want to move"downhill" on it, i.e., in the direction opposite to the gradient



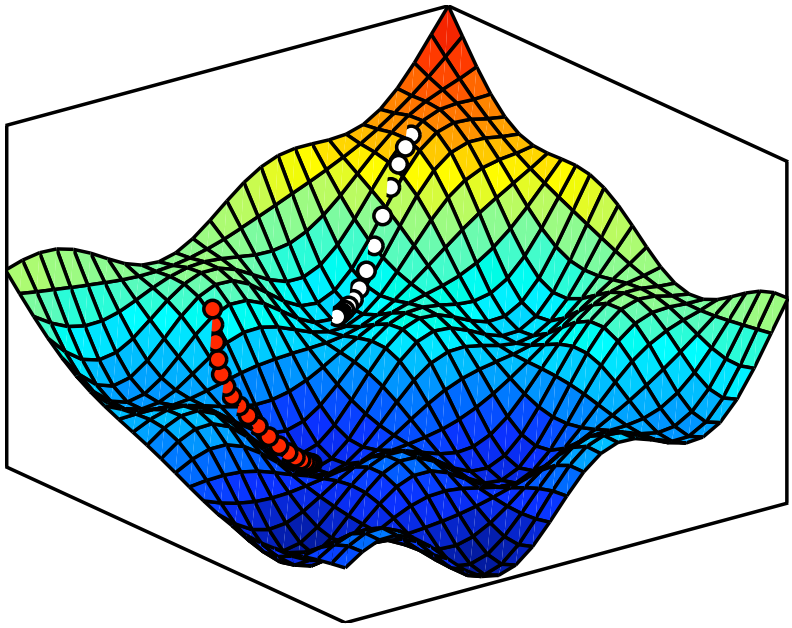
## Gradient descent

- The basic algorithm assumes that $\nabla f$ is easily computed
- We want to produce a sequence of vectors $\mathbf{u^1}, \mathbf{u^2}, \mathbf{u^3}, \ldots$ with the goal that:
  - $f(\mathbf{u^1}) > f(\mathbf{u^2}) > f(\mathbf{u^3}) > \ldots$
  - $\lim_{i \to \infty} \mathbf{u^i} = \mathbf{u}$ and $\mathbf{u}$ is locally optimal.
- The algorithm: Given $\mathbf{u^0}$, do for $i = 0, 1, 2, \ldots$

$$\mathbf{u^{i+1}} = \mathbf{u^i} - \alpha_i \nabla f(\mathbf{u^i}) \ ,$$

where $\alpha_i > 0$ is the *step size* or *learning rate* for iteration $i$.

## Example gradient descent traces





## Convergence

- Convergence depends in part on the $\alpha_i$.
- If they are too large (such as constant) oscillation or "bubbling" may occur.
  (This suggests the $\alpha_i$ should tend to zero as $i \rightarrow \infty$.)
- If they are too small, the $\mathbf{u^i}$ may not move far enough to reach a local minimum.

## Robbins-Monroe conditions

- The $\alpha_i$ are a Robbins-Monroe sequence if:
  - $\sum_{i=0}^{\infty} \alpha_i = +\infty$
  - $\sum_{i=0}^{\infty} \alpha_i^2 < \infty$
- E.g., $\alpha_i = \frac{1}{i+1}$ (averaging)
- E.g., $\alpha_i = \frac{1}{2}$ for $i = 1 \ldots T$, $\alpha_i = \frac{1}{2^2}$ for $i = T+1, \ldots (T+1) + 2T$ etc
- These conditions, along with appropriate conditions on $f$ are sufficient to ensure convergence of the $\mathbf{u^i}$.
- Many variants are possible: e.g., we may use at each step *a random vector* with mean $\nabla f(\mathbf{u^i})$; this is *stochastic gradient descent*.



## "Batch" versus "On-line" optimization

- Often in machine learning the error function, $J$, is a sum of errors attributed to each instance: ($J = J_1 + J_2 + \ldots + J_m$.)
- In *batch gradient descent*, the true gradient is computed at each step:

$$\nabla J = \nabla J_1 + \nabla J_2 + \ldots \nabla J_m.$$

- In *on-line gradient descent*, at each iteration one instance, $i \in \{1, \ldots, m\}$, is chosen at random and only $\nabla J_i$ is used in the update.
- Why prefer one or the other?

## "Batch" versus "On-line" optimization

- Batch is simple, repeatable.
- On-line:
  - Requires less computation per step.
  - Randomization may help escape poor local minima.
  - Allows working with a stream of data, rather than a static set (hence "on-line").



## Termination

There are many heuristics for deciding when to stop gradient descent.

1. Run until $||\nabla f||$ is smaller than some threshold.
2. Run it for as long as you can stand.
3. Run it for a short time from 100 different starting points, see which one is doing best, goto 2.
4. . . .

## Batch gradient descent for linear regression

- Start with an initial guess for $\mathbf{w}$
- Repeatedly change $\mathbf{w}$ to make $J(\mathbf{w})$ smaller:

$$w_j \leftarrow w_j - \alpha \frac{\partial}{\partial w_j} J(\mathbf{w}), \ \forall j = 0 \ldots n$$

- For linear hypotheses, we get:

$$w_j \leftarrow w_j + \alpha \sum_{i=1}^{m} (y_i - h_{\mathbf{w}}(\mathbf{x_i})) x_{i,j}$$

- This method is also known as *LMS update rule* or *Widrow-Hoff* learning rule



## Incremental (stochastic) gradient descent

1. Sample (choose) a training example $\langle \mathbf{x}, y \rangle$, in a randomized way
2. $$w_j \leftarrow w_j + \alpha (y - h_{\mathbf{w}}(\mathbf{x})) x_j$$
3. Repeat at will

Advantages:

- Better for large data sets
- Often faster than batch gradient descent
- Less prone to local minima

## Back to sigmoid neurons

- Error function is $J(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{m}(y_i - h_{\mathbf{w}}(\mathbf{x_i}))^2$
- The gradient is given by:

$$\nabla J = -\sum_{i=1}^{m}(y_i - h_{\mathbf{w}}(\mathbf{x_i}))\nabla h_{\mathbf{w}}(\mathbf{x_i})$$

- For sigmoid hypotheses, we have:

$$\nabla h_{\mathbf{w}}(\mathbf{x_i}) = h_{\mathbf{w}}(\mathbf{x_i})(1 - h_{\mathbf{w}}(\mathbf{x_i}))\mathbf{x_i}$$

- We obtain the weight update rule:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\sum_{i}(y_i - h_{\mathbf{w}}(\mathbf{x_i}))h_{\mathbf{w}}(\mathbf{x_i})(1 - h_{\mathbf{w}}(\mathbf{x_i}))\mathbf{x_i}$$

- We can again do batch or on-line updates
- There can be *lots of local minima*