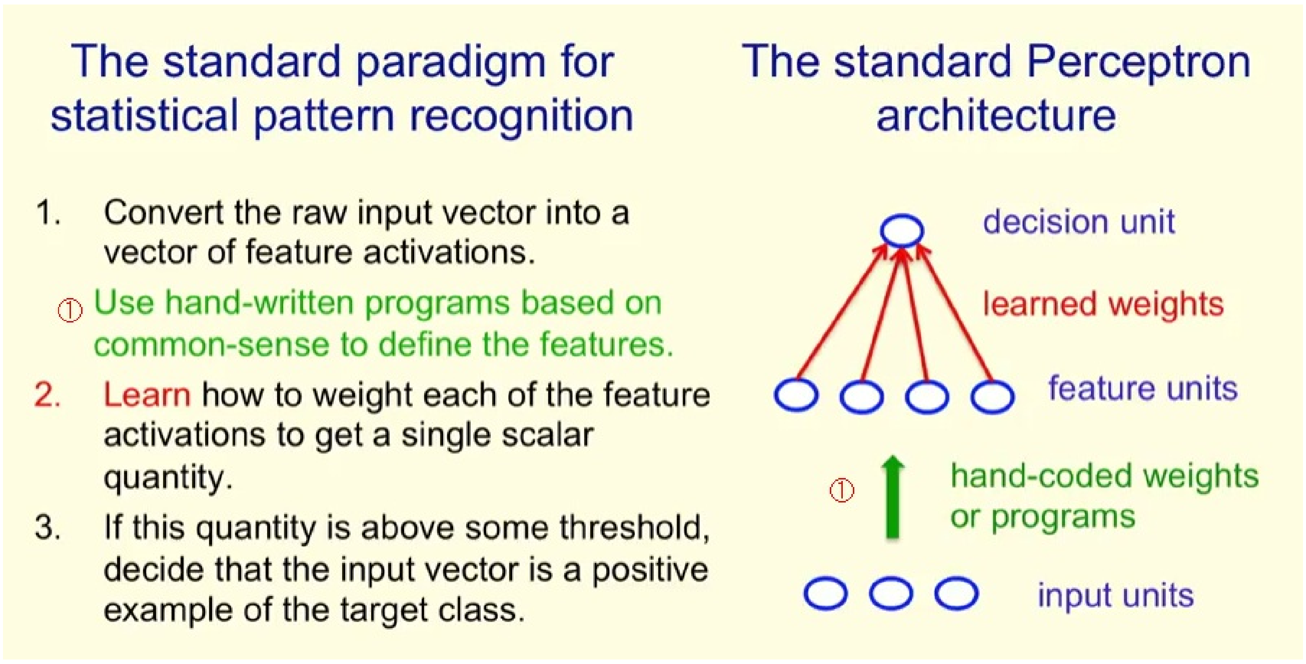
L2

- 2017/11/02 14:19
- An overview of the main types of neural network architecture
    - architecture means the way neurons are connected together.
    - Feed-forward neural networks
        - commonest type, one direction from input layer to hidden layers, to output layer
        - if no. hidden layer > 1, then Deep NN
        - function
            - They compute a series of transformations that change the similarities between cases/inputs.
                - make same content said by different speakers similar
                - while make different content said by the same speaker different
            - each (hidden) layer gives a more and more presice representation of the input.
            - the requires a non-linear activation function in each layer
    - Recurrent networks
        - intro
            - more powerful than FF NN
            - there are directed cycles in the connection graph
            - have complicated dynamics and this makes hard to train
                - a lot of interest at present in finding efficient ways of training recurrent nets.

            - more biologically realistic.
        - quite natural to model sequences/sequential data
            - hidden units are connected horizontally
            - at each time step the states of the hidden units determines the states of the hidden units of the next time step
            - hidden units act equivalently to the very deep FF nets with one hidden layer per time slice. Except that RE use the **same weights at every time slice** and they **get input at every time slice**
            - have the ability to remember information in their hidden state for a long time
                - But its very hard to train them to use this potential
    - Symmetrically connected networks
        - These are like recurrent networks, but the connections between units are symmetrical (they have the same weight in both directions)
        - symmetric networks are much easier to analyze than recurrent networks.
        - They are also more restricted in what they can do. because they obey an energy function.
        - Symmetrically connected nets without hidden units are called “Hopfield nets”

- The first generation of neural networks
    - The standard paradigm/procedure for statistical pattern recognition


The standard paradigm for statistical pattern recognition

1. Convert the raw input vector into a vector of feature activations.

   ① Use hand-written programs based on common-sense to define the features.
2. Learn how to weight each of the feature activations to get a single scalar quantity.
3. If this quantity is above some threshold, decide that the input vector is a positive example of the target class.

The standard Perceptron architecture

decision unit

learned weights

feature units

① hand-coded weights or programs

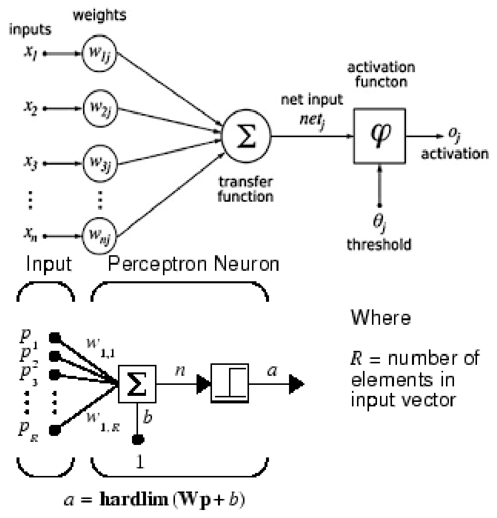
input units

- details
  - for ①, this step can't learn, it is hand-code. here we have to try many kinds of feature sets to find the best one that helps address the problem.
  - for learn,
    - What we learn is **how to weight each of the feature activations**, in order to get a single scalar quantity.
    - So the **weights on the features represent how much evidence the feature gives you**, **in favor or against the hypothesis** that the current input is an example of the kind of pattern you want to recognize.
  - decision unit - binary threshold neuron
    - the decision unit in a perceptron is a binary threshold neuron.
    - They compute a weighted sum of inputs, adding the bias, and make 0/1 decision

based on this sum.

- treat bias
    - treat it as weight, when introduce x0 = 1 in the feature vector

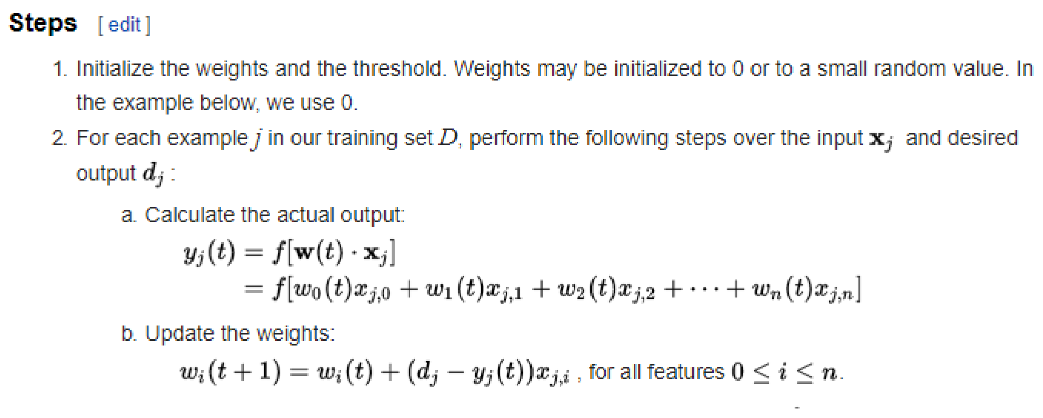
- perceptron
    - A perceptron is a particular example of a statistical pattern recognition system.
    - Intro
        - the **perceptron** is an algorithm for supervised learning of binary classifiers
        - It is a type of linear classifier,
            - i.e. a classification algorithm that makes its predictions based on a linear predictor function combining a set of weights with the feature vector.
        - The algorithm allows for online learning
    - Architecture



    - there are actually many different kinds of perceptrons, the standard kind is shown above, called an alpha perceptron.
        - raw inputs => feature => learning weights => threshold => predicts
    - Perceptrons history

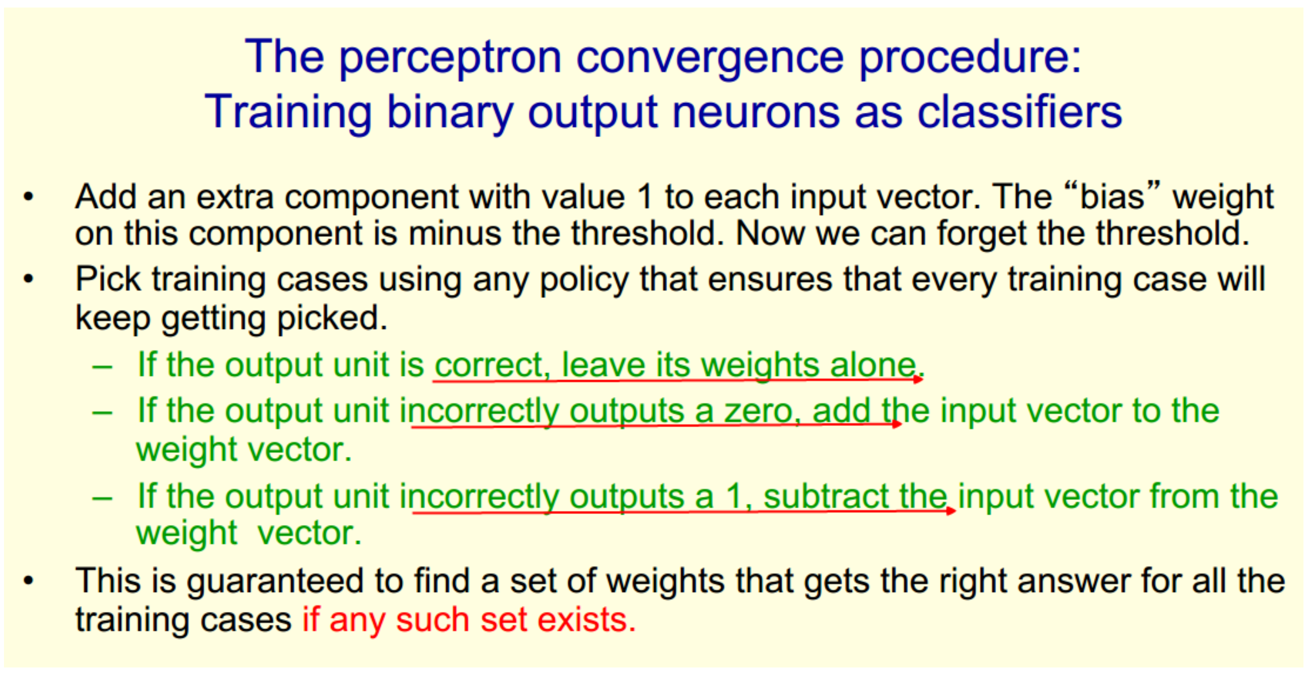
- popularized in the early 1960s by Frank Rosenblatt
- Minsky and Papert published a book called Perceptrons that analyzed what perceptrons could do and showed their limitations. but limitation in perceptrons can be overgeneralized to all NN models

- The perceptron convergence procedure



Steps [edit]

1. Initialize the weights and the threshold. Weights may be initialized to 0 or to a small random value. In the example below, we use 0.
2. For each example j in our training set D, perform the following steps over the input $\mathbf{x}_j$ and desired output d_j :
   a. Calculate the actual output:
   $$\begin{aligned} y_j(t) &= f[\mathbf{w}(t)\cdot\mathbf{x}_j] \\ &= f[w_0(t)x_{j,0}+w_1(t)x_{j,1}+w_2(t)x_{j,2}+\dots+w_n(t)x_{j,n}] \end{aligned}$$
   b. Update the weights:
   $w_i(t+1)=w_i(t)+(d_j-y_j(t))x_{j,i}$, for all features $0\le i\le n$.



The perceptron convergence procedure: Training binary output neurons as classifiers

- Add an extra component with value 1 to each input vector. The "bias" weight on this component is minus the threshold. Now we can forget the threshold.
- Pick training cases using any policy that ensures that every training case will keep getting picked.
  - If the output unit is correct, leave its weights alone.
  - If the output unit incorrectly outputs a zero, add the input vector to the weight vector.
  - If the output unit incorrectly outputs a 1, subtract the input vector from the weight vector.
- This is guaranteed to find a set of weights that gets the right answer for all the training cases if any such set exists.

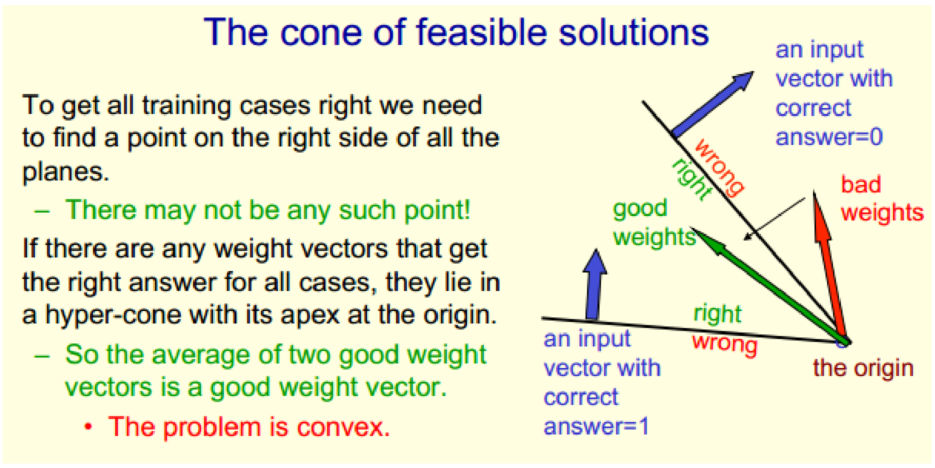
- but some times such set doesn't exist. The weights depends on the feature sets, which means only if we can select out a good feature set, we can guarantee the perceptron convergence.

- A geometrical view of perceptrons when learning
  - SUMMAY
    - 构建了一个空间，用来描述选择weight的位置，是每个training point

构建的cone。

        - input vector是一个指示性项量，input vector 出去bias是一hypeplane。weight x input vector 的符号该与该 input vector的符号对应

    - reminder
        - In this figure, we're going to get a geometrical understanding of what happens when a perceptron learns. To do this, we have to think in terms of a weight space. It's a high dimensional space in which each point corresponds to a particular setting for all the weights. In this phase, we can represent the training cases as planes and learning consists of trying to get the weight vector on the right side of all the training planes.
        - remember that going from 13-D to 14-D creates as much extra complexity as going from 2-D to 3-D
    - Weight-space
        - in this space, one dimension per weight.
        - A point in the space represents a particular setting of all the weights.
        - Assuming that we have eliminated the threshold, each training case can be represented as a hyperplane through the origin.
            - The weights must lie on one side of this hyper-plane to get the answer correct.



The cone of feasible solutions

To get all training cases right we need to find a point on the right side of all the planes.

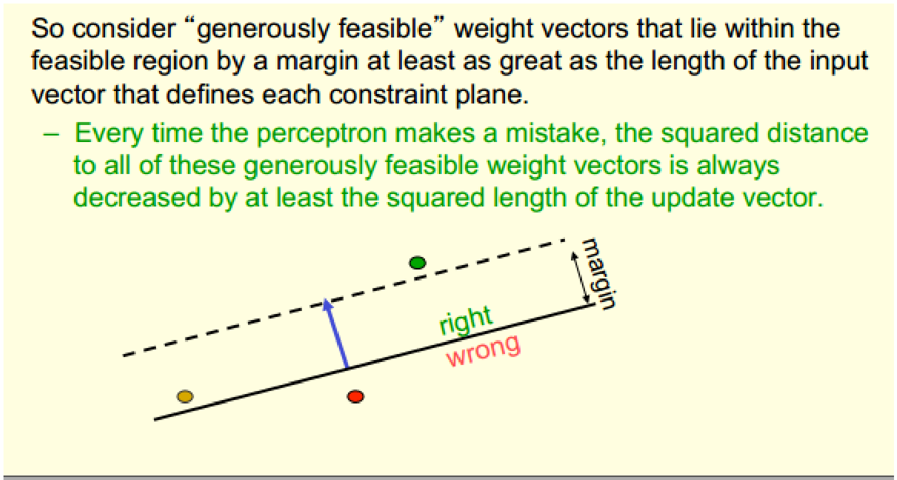
- There may not be any such point!

If there are any weight vectors that get the right answer for all cases, they lie in a hyper-cone with its apex at the origin.

- So the average of two good weight vectors is a good weight vector.
    - The problem is convex.

an input vector with correct answer=0
wrong
right
bad weights
good weights
right
wrong
an input vector with correct answer=1
the origin

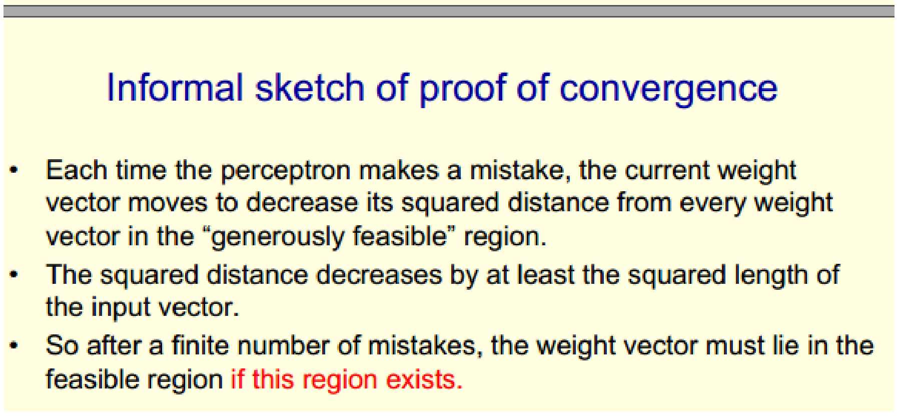
- Why the learning works
    - 酌情看待吧，在应用方面不太重要，具体参照PPT



So consider "generously feasible" weight vectors that lie within the feasible region by a margin at least as great as the length of the input vector that defines each constraint plane.

– Every time the perceptron makes a mistake, the squared distance to all of these generously feasible weight vectors is always decreased by at least the squared length of the update vector.

margin
right
wrong



Informal sketch of proof of convergence

- Each time the perceptron makes a mistake, the current weight vector moves to decrease its squared distance from every weight vector in the "generously feasible" region.
- The squared distance decreases by at least the squared length of the input vector.
- So after a finite number of mistakes, the weight vector must lie in the feasible region if this region exists.

- What perceptrons can't do
    - Reasons: These limitations stem from the kinds of features you use.
        - If you use the right features, you could do almost anything.
        - If you use the wrong features, they're extremely limited in what the learning part can do.
        - **So the difficulty of learning is to learn the right features. And you need to learn the features for complex problems.**
        - So the reason that neural network research came to a halt in the late 1960s and early 1970s is that perceptrons were shown to be very limited
    - The limitations of Perceptrons
        - If you are allowed to choose enough features by hand, you can do almost anything.
            - e.g. a 2-featured perceptron can't always classify more than 4 points
                - that is we need almost m^1/2 features to classify m points, if they're binary

      - so this can't be generalized.
    - But once the hand-coded features have been determined, there are very strong limitations on what a perceptron can learn.
      - a set of features for a typical set of training set/a typical problem
      - only works for linearly separable data sets
  - Discriminating simple patterns under translation with wrap-around
    - To deal with such transformations, a Perceptron needs to use multiple feature units to recognize transformations of informative sub-patterns.
    - So the tricky part of pattern recognition must be solved by the hand-coded feature detectors, not the learning procedure.
  - Conclusion and solution - Learning with hidden units
    - Networks without hidden units are very limited in the input-output mappings they can learn to model.
      - More layers of linear units do not help. Its still linear.
      - Fixed output non-linearities are not enough
    - so We need multiple layers of adaptive, non-linear hidden units. but the problem is training
      - We need an efficient way of adapting all the weights, not just the last layer, like the perceptron. This is hard
      - Learning the weights going into hidden units is equivalent to learning features.
      - This is difficult because nobody is telling us directly what the hidden units should do.
      - the, real problem is, how do we figure out how to learn these weights go into hidden units so that the hidden units turn into the kinds of feature detectors we need for

solving a problem, when nobody is telling us what the featured detector should be.