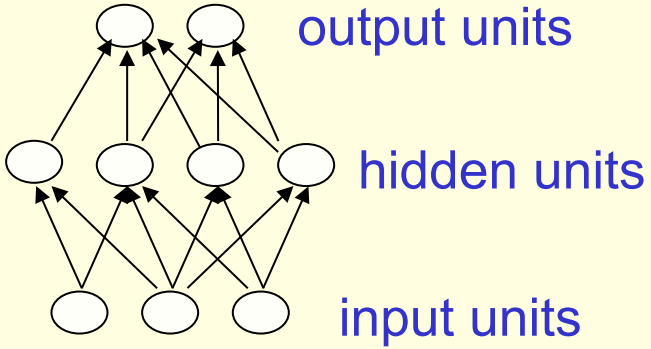
Neural Networks for Machine Learning

Lecture 2a
An overview of the main types of neural network architecture

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

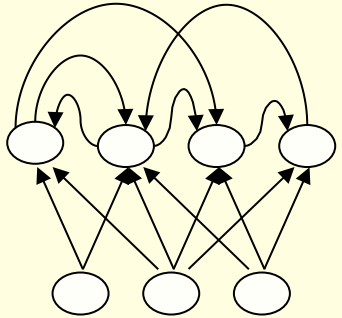
Feed-forward neural networks

- These are the commonest type of neural network in practical applications.
  - The first layer is the input and the last layer is the output.
  - If there is more than one hidden layer, we call them “deep” neural networks.
- They compute a series of transformations that change the similarities between cases.
  - The activities of the neurons in each layer are a non-linear function of the activities in the layer below.



Recurrent networks

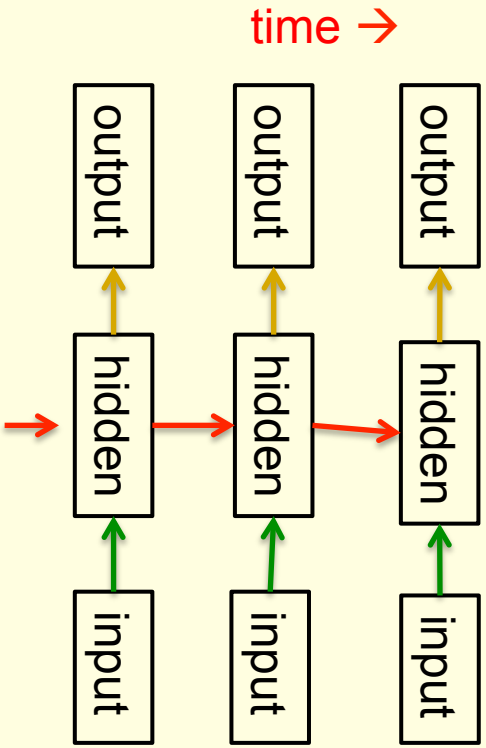
- These have directed cycles in their connection graph.
  - That means you can sometimes get back to where you started by following the arrows.
- They can have complicated dynamics and this can make them very difficult to train.
  - There is a lot of interest at present in finding efficient ways of training recurrent nets.
- They are more biologically realistic.



Recurrent nets with multiple hidden layers are just a special case that has some of the hidden→hidden connections missing.

Recurrent neural networks for modeling sequences

- Recurrent neural networks are a very natural way to model sequential data:
  - They are equivalent to very deep nets with one hidden layer per time slice.
  - Except that they use the same weights at every time slice and they get input at every time slice.
- They 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.



An example of what recurrent neural nets can now do (to whet your interest!)

- Ilya Sutskever (2011) trained a special type of recurrent neural net to predict the next character in a sequence.

- After training for a long time on a string of half a billion characters from English Wikipedia, he got it to generate new text.
  - It generates by predicting the probability distribution for the next character and then sampling a character from this distribution.
  - The next slide shows an example of the kind of text it generates. Notice how much it knows!

Some text generated one character at a time by Ilya Sutskever's recurrent neural network

In 1974 Northern Denver had been overshadowed by CNL, and several Irish intelligence agencies in the Mediterranean region. However, on the Victoria, Kings Hebrew stated that Charles decided to escape during an alliance. The mansion house was completed in 1882, the second in its bridge are omitted, while closing is the proton reticulum composed below it aims, such that it is the blurring of appearing on any well-paid type of box printer.

Symmetrically connected networks

- These are like recurrent networks, but the connections between units are symmetrical (they have the same weight in both directions).
  - John Hopfield (and others) realized that 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.
    - For example, they cannot model cycles.
- Symmetrically connected nets without hidden units are called “Hopfield nets”.

Symmetrically connected networks with hidden units

- These are called “Boltzmann machines”.
  - They are much more powerful models than Hopfield nets.
  - They are less powerful than recurrent neural networks.
  - They have a beautifully simple learning algorithm.

- We will cover Boltzmann machines towards the end of the course.

Neural Networks for Machine Learning

Lecture 2b
Perceptrons:
The first generation of neural networks

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

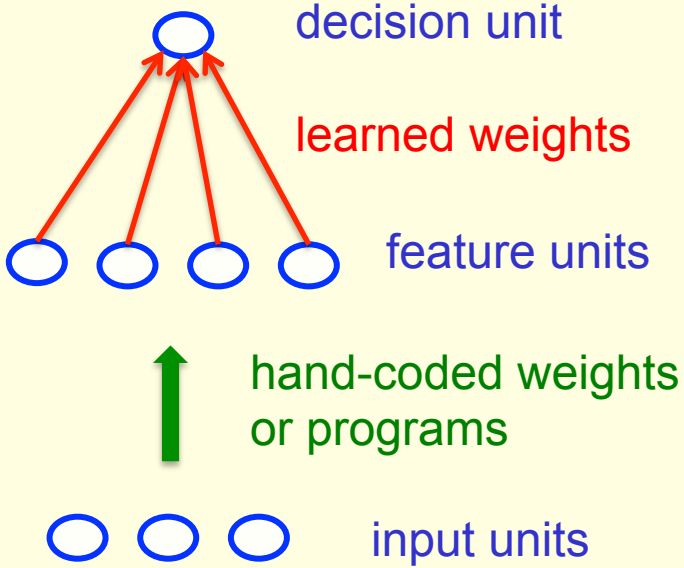
The standard paradigm for statistical pattern recognition

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

   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



The history of perceptrons

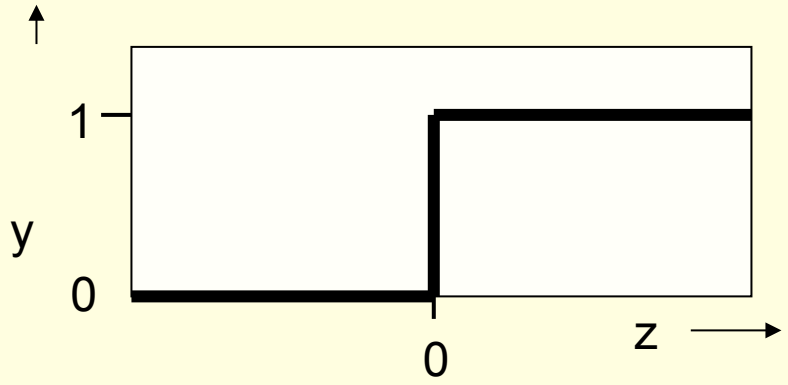
- They were popularised by Frank Rosenblatt in the early 1960's.
  - They appeared to have a very powerful learning algorithm.
  - Lots of grand claims were made for what they could learn to do.
- In 1969, Minsky and Papert published a book called "Perceptrons" that analysed what they could do and showed their limitations.
  - Many people thought these limitations applied to all neural network models.
- The perceptron learning procedure is still widely used today for tasks with enormous feature vectors that contain many millions of features.

Binary threshold neurons (decision units)

- McCulloch-Pitts (1943)
  - First compute a weighted sum of the inputs from other neurons (plus a bias).
  - Then output a 1 if the weighted sum exceeds zero.

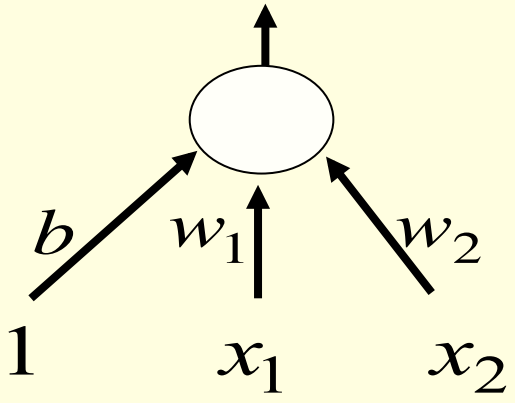
$$z = b + \sum_i x_i w_i$$

$$y = \begin{cases} 1 \text{ if } z \geq 0 \\ 0 \text{ otherwise} \end{cases}$$



How to learn biases using the same rule as we use for learning weights

- A threshold is equivalent to having a negative bias.
- We can avoid having to figure out a separate learning rule for the bias by using a trick:
  - A bias is exactly equivalent to a weight on an extra input line that always has an activity of 1.
  - We can now learn a bias as if it were a weight.



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.

Neural Networks for Machine Learning

Lecture 2c
A geometrical view of perceptrons

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

Warning!

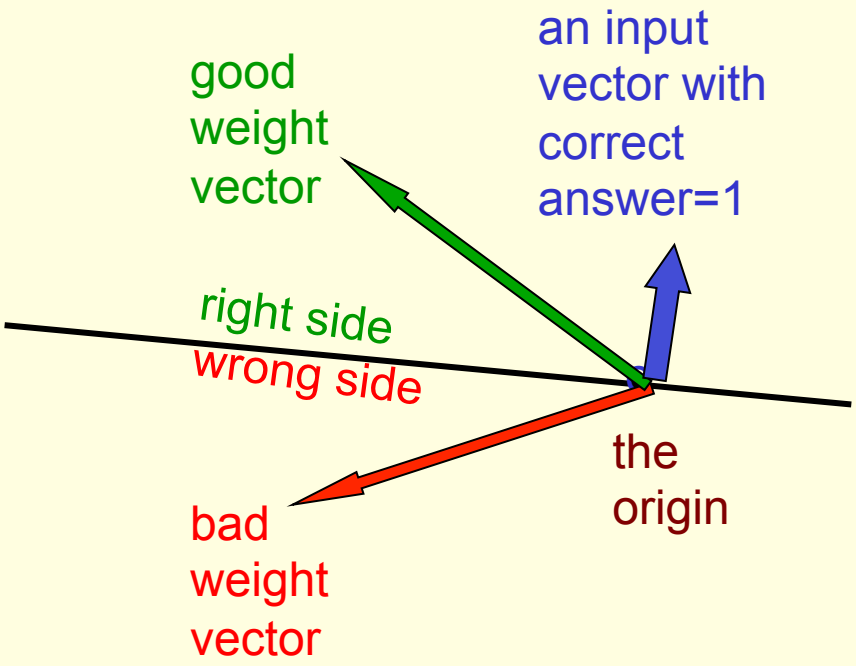
- For non-mathematicians, this is going to be tougher than the previous material.
  - You may have to spend a long time studying the next two parts.
- If you are not used to thinking about hyper-planes in high-dimensional spaces, now is the time to learn.
- To deal with hyper-planes in a 14-dimensional space, visualize a 3-D space and say “fourteen” to yourself very loudly. Everyone does it.
  - But remember that going from 13-D to 14-D creates as much extra complexity as going from 2-D to 3-D.

Weight-space

- This space has 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.

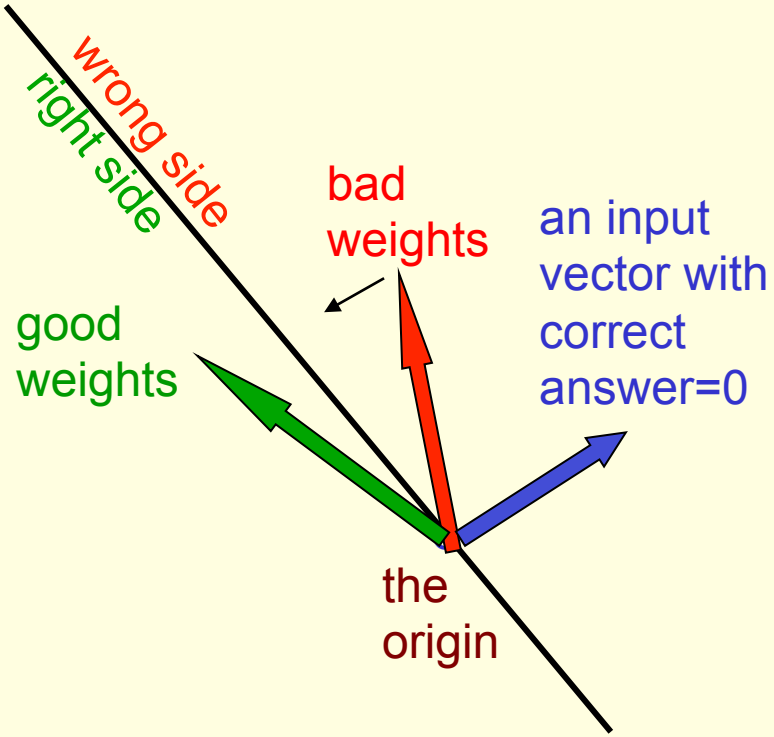
Weight space

- Each training case defines a plane (shown as a black line)
  - The plane goes through the origin and is perpendicular to the input vector.
  - On one side of the plane the output is wrong because the scalar product of the weight vector with the input vector has the wrong sign.



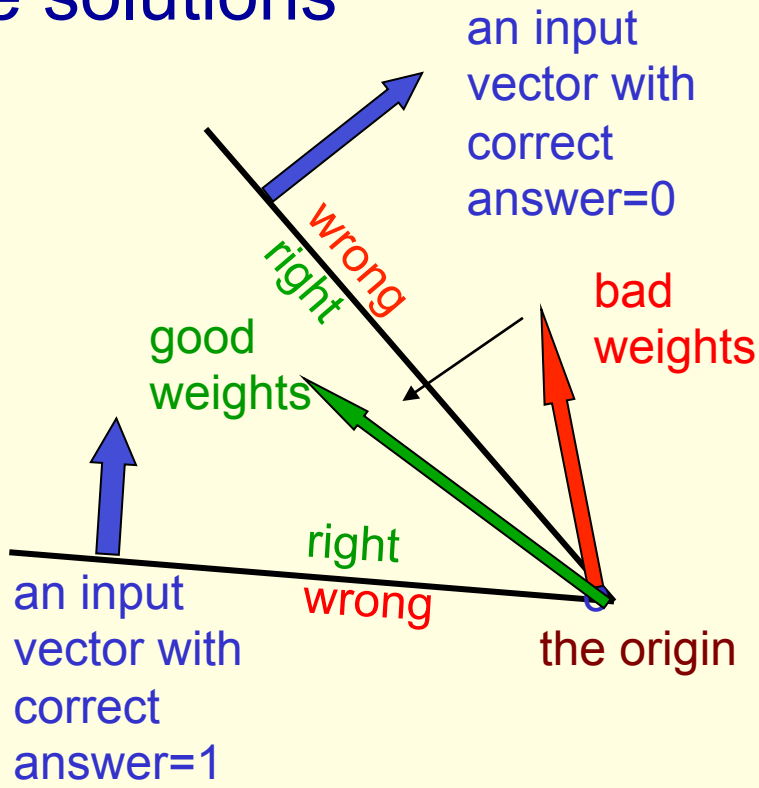
Weight space

- Each training case defines a plane (shown as a black line)
  - The plane goes through the origin and is perpendicular to the input vector.
  - On one side of the plane the output is wrong because the scalar product of the weight vector with the input vector has the wrong sign.



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.



Neural Networks for Machine Learning

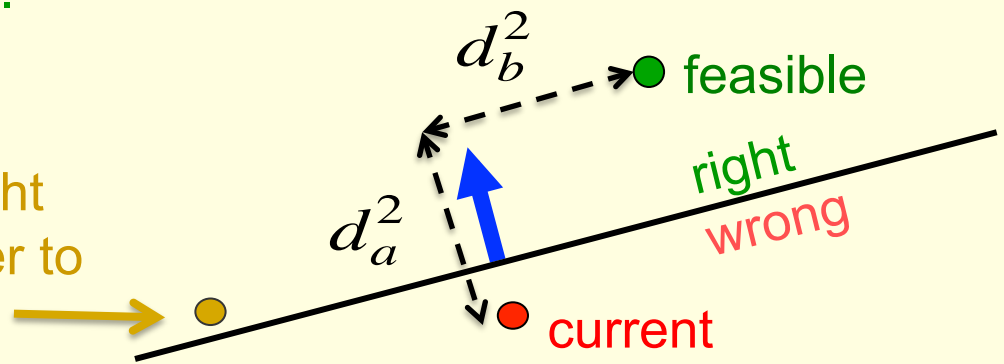
Lecture 2d
Why the learning works

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

Why the learning procedure works (first attempt)

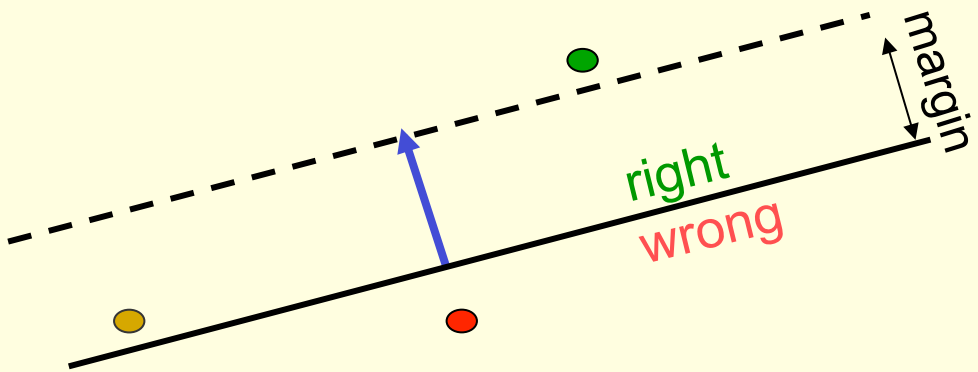
- Consider the squared distance $d_a^2 + d_b^2$ between any feasible weight vector and the current weight vector.
  - Hopeful claim: Every time the perceptron makes a mistake, the learning algorithm moves the current weight vector closer to all feasible weight vectors.

Problem case: The weight vector may not get closer to this feasible vector!



Why the learning procedure works

- 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.



This is interesting because the input vector (plane) is moving a unit vector distance every time perceptron makes a mistake so as to fine-tune the correct prediction.

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.

Neural Networks for Machine Learning

Lecture 2e
What perceptrons can’t do

Geoffrey Hinton
with
Nitish Srivastava
Kevin Swersky

The limitations of Perceptrons

- If you are allowed to choose the features by hand and if you use enough features, you can do almost anything.
  - For binary input vectors, we can have a separate feature unit for each of the exponentially many binary vectors and so we can make any possible discrimination on binary input vectors.
    - This type of table look-up won't generalize.
- But once the hand-coded features have been determined, there are very strong limitations on what a perceptron can learn.

What binary threshold neurons cannot do

- A binary threshold output unit cannot even tell if two single bit features are the same!

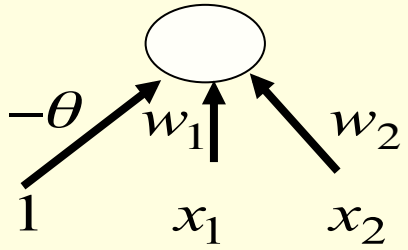
  Positive cases (same): (1,1) → 1; (0,0) → 1

  Negative cases (different): (1,0) → 0; (0,1) → 0

- The four input-output pairs give four inequalities that are impossible to satisfy:

$$w_1 + w_2 \geq \theta, \quad 0 \geq \theta$$

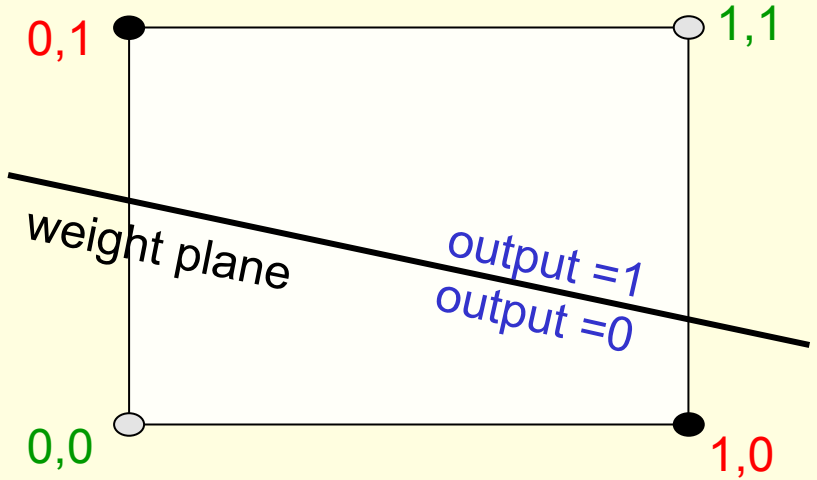
$$w_1 < \theta, \qquad w_2 < \theta$$



A geometric view of what binary threshold neurons cannot do

Imagine “data-space” in which the axes correspond to components of an input vector.

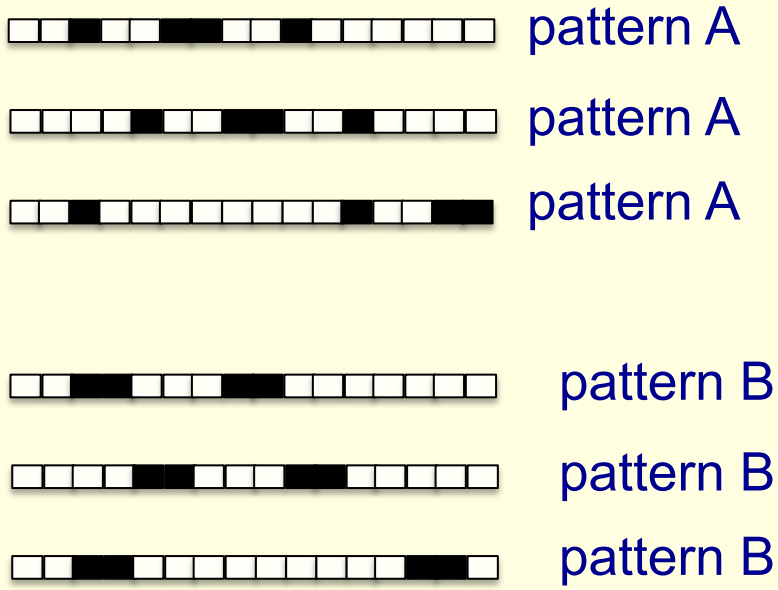
- Each input vector is a point in this space.
- A weight vector defines a plane in data-space.
- The weight plane is perpendicular to the weight vector and misses the origin by a distance equal to the threshold.



The positive and negative cases cannot be separated by a plane

Discriminating simple patterns under translation with wrap-around

- Suppose we just use pixels as the features.
- Can a binary threshold unit discriminate between different patterns that have the same number of on pixels?
  - Not if the patterns can translate with wrap-around!



Sketch of a proof that a binary decision unit cannot discriminate patterns with the same number of on pixels
(assuming translation with wraparound)

- For pattern A, use training cases in all possible translations.
  - Each pixel will be activated by 4 different translations of pattern A.
  - So the total input received by the decision unit over all these patterns will be four times the sum of all the weights.
- For pattern B, use training cases in all possible translations.
  - Each pixel will be activated by 4 different translations of pattern B.
  - So the total input received by the decision unit over all these patterns will be four times the sum of all the weights.
- But to discriminate correctly, every single case of pattern A must provide more input to the decision unit than every single case of pattern B.
  - This is impossible if the sums over cases are the same.

Why this result is devastating for Perceptrons

- The whole point of pattern recognition is to recognize patterns despite transformations like translation.
- Minsky and Papert's "Group Invariance Theorem" says that the part of a Perceptron that learns cannot learn to do this if the transformations form a group.
  - Translations with wrap-around form a group.
- 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.

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.
- We need multiple layers of adaptive, non-linear hidden units. But how can we train such nets?
  - We need an efficient way of adapting all the weights, not just the last layer. 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.