Reasons to study neural computation

- To understand how the brain actually works.
  - Its very big and very complicated and made of stuff that dies when you poke it around. So we need to use computer simulations.
- To understand a style of parallel computation inspired by neurons and their adaptive connections.
  - Very different style from sequential computation.
    - should be good for things that brains are good at (e.g. vision)
    - Should be bad for things that brains are bad at (e.g. 23 x 71)
- To solve practical problems by using novel learning algorithms inspired by the brain (this course)
  - Learning algorithms can be very useful even if they are not how the brain actually works.

Linear neurons

- These are simple but computationally limited
  - If we can make them learn we may get insight into more complicated neurons.

bias

i^{th} input

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

output

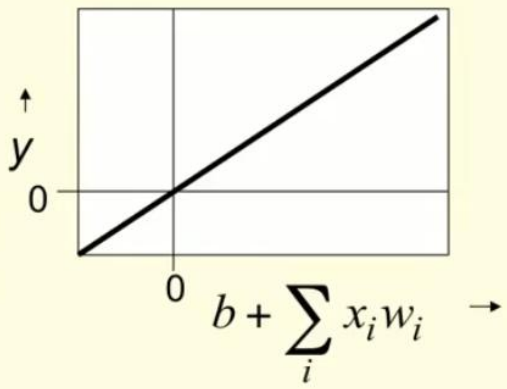
index over input connections

weight on i^{th} input

Linear neurons

- These are simple but computationally limited
  - If we can make them learn we may get insight into more complicated neurons.

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



Binary threshold neurons

- McCulloch-Pitts (1943): influenced Von Neumann.
  - First compute a weighted sum of the inputs.
  - Then send out a fixed size spike of activity if the weighted sum exceeds a threshold.
  - McCulloch and Pitts thought that each spike is like the truth value of a proposition and each neuron combines truth values to compute the truth value of another proposition!

Binary threshold neurons

- There are two equivalent ways to write the equations for a binary threshold neuron:

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

$$\boxed{\theta = -b}$$

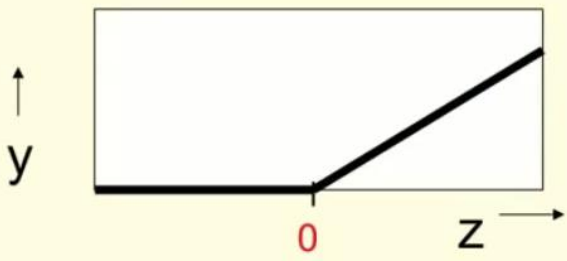
$$y = \begin{cases} 1 \text{ if } z \geq \theta \\ 0 \text{ otherwise} \end{cases} \qquad\qquad y = \begin{cases} 1 \text{ if } z \geq 0 \\ 0 \text{ otherwise} \end{cases}$$

Rectified Linear Neurons
(sometimes called linear threshold neurons)

They compute a linear weighted sum of their inputs.
The output is a non-linear function of the total input.

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

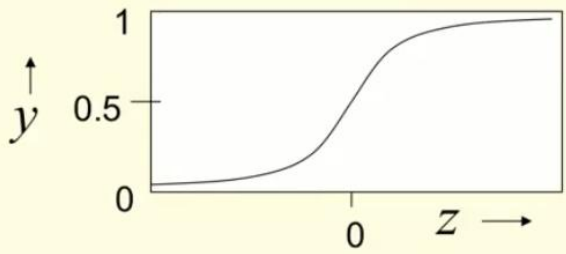
$$y = \begin{cases} z \text{ if } z > 0 \\ 0 \text{ otherwise} \end{cases}$$



Sigmoid neurons

- These give a real-valued output that is a smooth and bounded function of their total input.
  - Typically they use the logistic function
  - They have nice derivatives which make learning easy (see lecture 3).

$$z = b + \sum_i x_i w_i \qquad y = \frac{1}{1+e^{-z}}$$

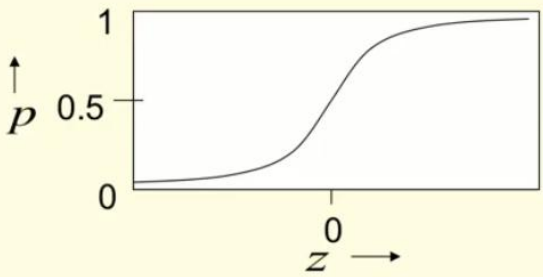


- Sigmoid neuron – intrinsically random
-

Stochastic binary neurons

- These use the same equations as logistic units.
  - But they treat the output of the logistic as the probability of producing a spike in a short time window.
- We can do a similar trick for rectified linear units:
  - The output is treated as the Poisson rate for spikes.

$$z = b + \sum_i x_i w_i \qquad p(s=1) = \frac{1}{1+e^{-z}}$$



Binary threshold neurons

- McCulloch-Pitts (1943): influenced Von Neumann.
  - First compute a weighted sum of the inputs.
  - Then send out a fixed size spike of activity if the weighted sum exceeds a threshold.
  - McCulloch and Pitts thought that each spike is like the truth value of a proposition and each neuron combines truth values to compute the truth value of another proposition!

Types of learning task

- Supervised learning
  - Learn to predict an output when given an input vector.
- Reinforcement learning
  - Learn to select an action to maximize payoff.
- Unsupervised learning
  - Discover a good internal representation of the input.

Reinforcement learning

- In reinforcement learning, the output is an action or sequence of actions and the only supervisory signal is an occasional scalar reward.
  - The goal in selecting each action is to maximize the expected sum of the future rewards.
  - We usually use a discount factor for delayed rewards so that we don't have to look too far into the future.
- Reinforcement learning is difficult:
  - The rewards are typically delayed so its hard to know where we went wrong (or right).
  - A scalar reward does not supply much information.
- This course cannot cover everything and reinforcement learning is one of the important topics we will not cover.

How supervised learning typically works

- We start by choosing a model-class: $y = f(\mathbf{x}; \mathbf{W})$
  - A model-class, *f*, is a way of using some numerical parameters, **W**, to map each input vector, **x**, into a predicted output y.
- Learning usually means adjusting the parameters to reduce the discrepancy between the target output, t, on each training case and the actual output, y, produced by the model.
  - For regression, $\frac{1}{2}(y-t)^2$ is often a sensible measure of the discrepancy.

Unsupervised learning

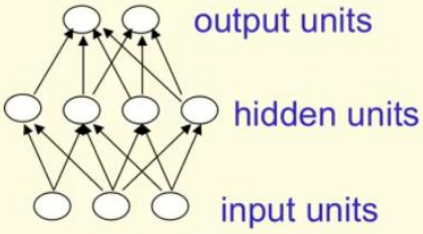
- For about 40 years, unsupervised learning was largely ignored by the machine learning community
  - Some widely used definitions of machine learning actually excluded it.
  - Many researchers thought that clustering was the only form of unsupervised learning.
- It is hard to say what the aim of unsupervised learning is.
  - One major aim is to create an internal representation of the input that is useful for subsequent supervised or reinforcement learning.
  - You can compute the distance to a surface by using the disparity between two images. But you don't want to learn to compute disparities by stubbing your toe thousands of times.

Other goals for unsupervised learning

- It provides a compact, low-dimensional representation of the input.
  - High-dimensional inputs typically live on or near a low-dimensional manifold (or several such manifolds).
  - Principal Component Analysis is a widely used linear method for finding a low-dimensional representation.
- It provides an economical high-dimensional representation of the input in terms of learned features.
  - Binary features are economical.
  - So are real-valued features that are nearly all zero.
- It finds sensible clusters in the input.
  - This is an example of a very sparse code in which only one of the features is non-zero.

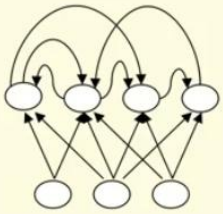
Feed-forward neural networks

- 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

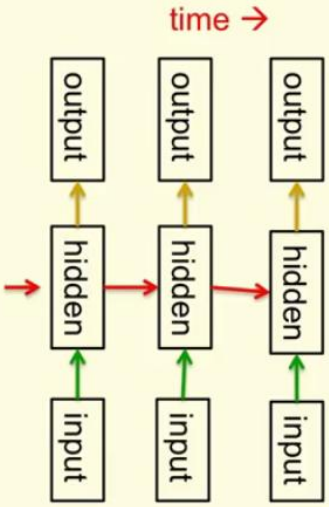
- These have directed cycles in their connection graph.
  - That means you can sometimes get back to where you started by following the arrows.
- They can have complicated dynamics and this can make them very difficult to train.
  - There is a lot of interest at present in finding efficient ways of training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



Symmetrically connected networks

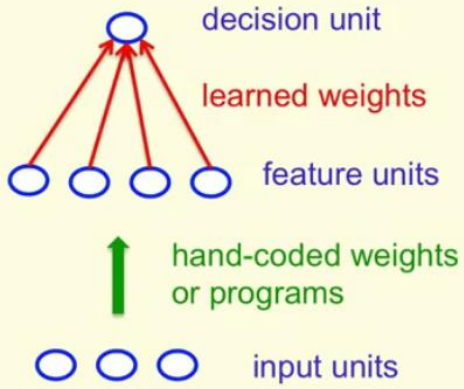
- These are like recurrent networks, but the connections between units are symmetrical (they have the same weight in both directions).
  - John Hopfield (and others) realized that symmetric networks are much easier to analyze than recurrent networks.
  - They are also more restricted in what they can do. because they obey an energy function.
    - For example, they cannot model cycles.

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

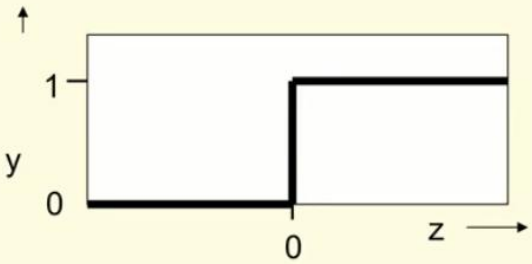


Binary threshold neurons (decision units)

- 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.

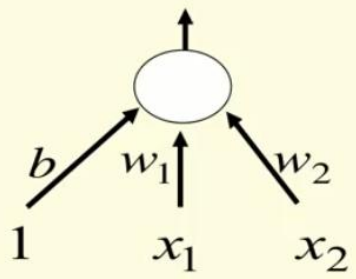
$$z = b + \sum_i x_i w_i$$

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



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.



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.

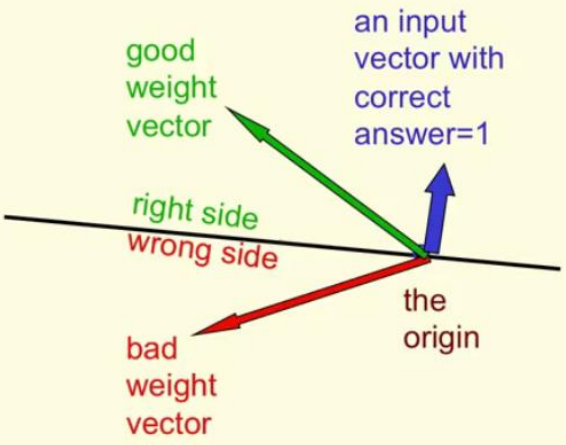
Weight-space

- 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)

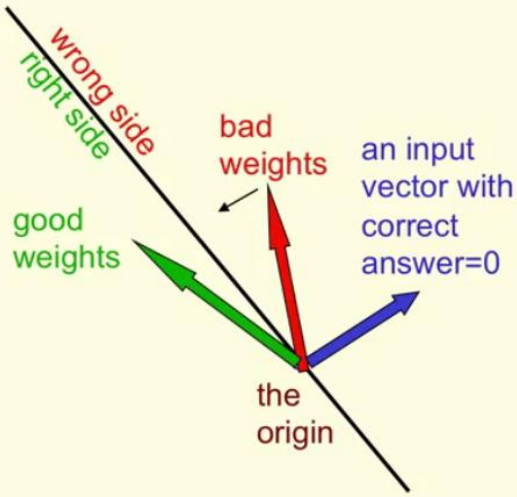
- The plane goes through the origin and is perpendicular to the input vector.
- On one side of the plane the output is wrong because the scalar product of the weight vector with the input vector has the wrong sign.



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.



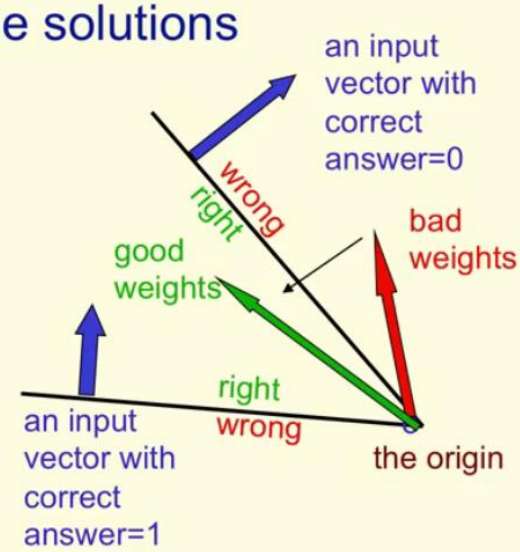
The cone of feasible solutions

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!

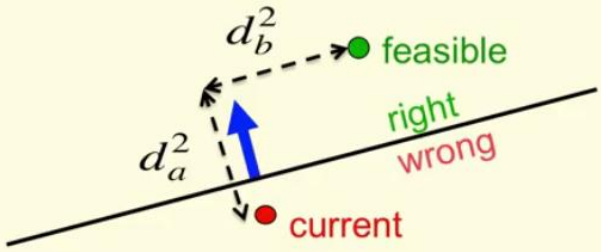
If there are any weight vectors that get the right answer for all cases, they lie in a hyper-cone with its apex at the origin.

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



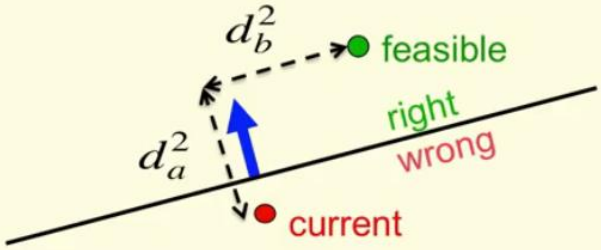
Why the learning procedure works (first attempt)

- Consider the squared distance $d_a^2 + d_b^2$ between any feasible weight vector and the current weight vector.



Why the learning procedure works (first attempt)

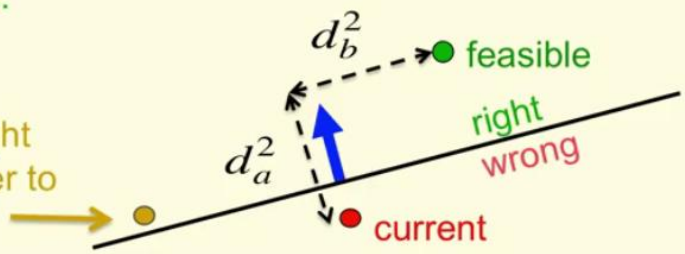
- Consider the squared distance $d_a^2 + d_b^2$ between any feasible weight vector and the current weight vector.
  - Hopeful claim: Every time the perceptron makes a mistake, the learning algorithm moves the current weight vector closer to all feasible weight vectors.



Why the learning procedure works (first attempt)

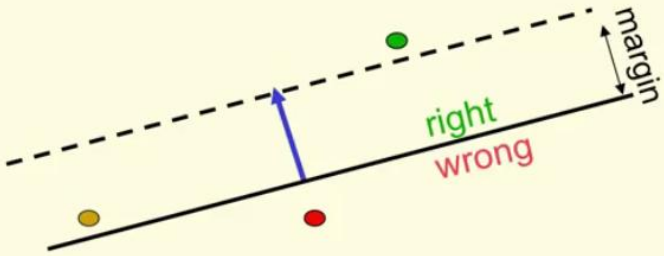
- Consider the squared distance $d_a^2 + d_b^2$ between any feasible weight vector and the current weight vector.
  - Hopeful claim: Every time the perceptron makes a mistake, the learning algorithm moves the current weight vector closer to all feasible weight vectors.

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



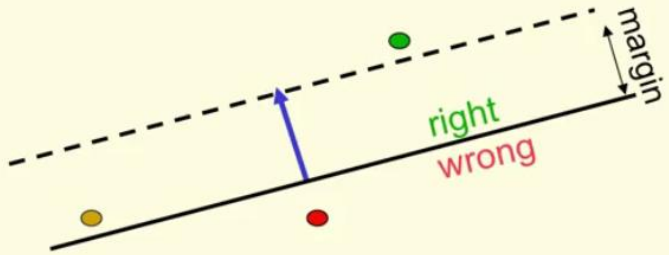
Why the learning procedure works

- 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.



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.



Assumption that there's a generously feasible weight vector!

If there's no vector then the PLA will fail.

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.

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!

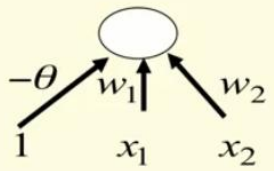
  Positive cases (same): (1,1) → 1; (0,0) → 1

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

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

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

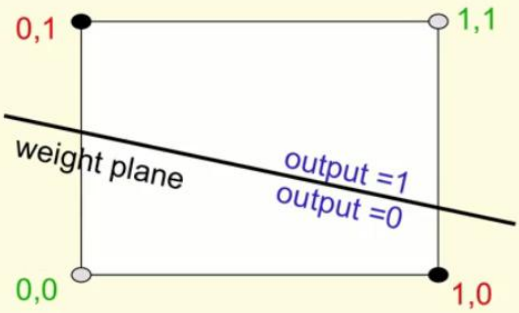
$$w_1 < \theta, \quad w_2 < \theta$$



A geometric view of what binary threshold neurons cannot do

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

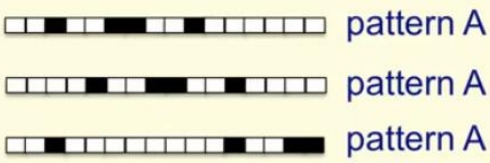
- Each input vector is a point in this space.
- A weight vector defines a plane in data-space.
- The weight plane is perpendicular to the weight vector and misses the origin by a distance equal to the threshold.



The positive and negative cases cannot be separated by a plane

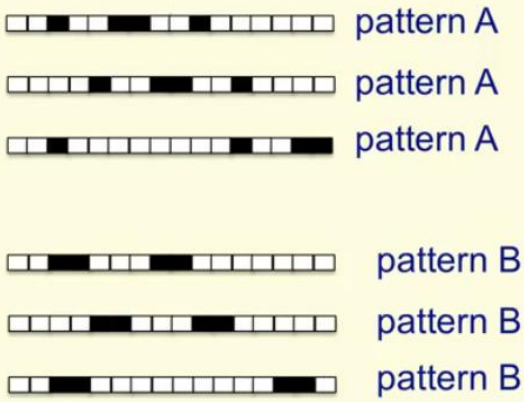
Discriminating simple patterns under translation with wrap-around

- 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!



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.

Weight vectors – learning them alone won't suffice.

We've to learn the feature detectors as well.

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.

Why the perceptron learning procedure cannot be generalised to hidden layers

- The perceptron convergence procedure works by ensuring that every time the weights change, they get closer to every "generously feasible" set of weights.
  - This type of guarantee cannot be extended to more complex networks in which the average of two good solutions may be a bad solution.
- So "multi-layer" neural networks do not use the perceptron learning procedure.
  - They should never have been called multi-layer perceptrons.

A different way to show that a learning procedure makes progress

- Instead of showing the weights get closer to a good set of weights, show that the actual output values get closer the target values.
  - This can be true even for non-convex problems in which there are many quite different sets of weights that work well and averaging two good sets of weights may give a bad set of weights.
  - It is not true for perceptron learning.
- The simplest example is a linear neuron with a squared error measure.

Linear neurons (also called linear filters)

- The neuron has a real-valued output which is a weighted sum of its inputs
- The aim of learning is to minimize the error summed over all training cases.
  - The error is the squared difference between the desired output and the actual output.

weight vector

$$y = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$$

neuron's estimate of the desired output

input vector

Why don't we solve it analytically?

- It is straight-forward to write down a set of equations, one per training case, and to solve for the best set of weights.
  - This is the standard engineering approach so why don't we use it?
- Scientific answer: We want a method that real neurons could use.
- Engineering answer: We want a method that can be generalized to multi-layer, non-linear neural networks.
  - The analytic solution relies on it being linear and having a squared error measure.
  - Iterative methods are usually less efficient but they are much easier to generalize.

A toy example to illustrate the iterative method

- Each day you get lunch at the cafeteria.
  - Your diet consists of fish, chips, and ketchup.
  - You get several portions of each.
- The cashier only tells you the total price of the meal
  - After several days, you should be able to figure out the price of each portion.
- The iterative approach: Start with random guesses for the prices and then adjust them to get a better fit to the observed prices of whole meals.

Solving the equations iteratively

- Each meal price gives a linear constraint on the prices of the portions:

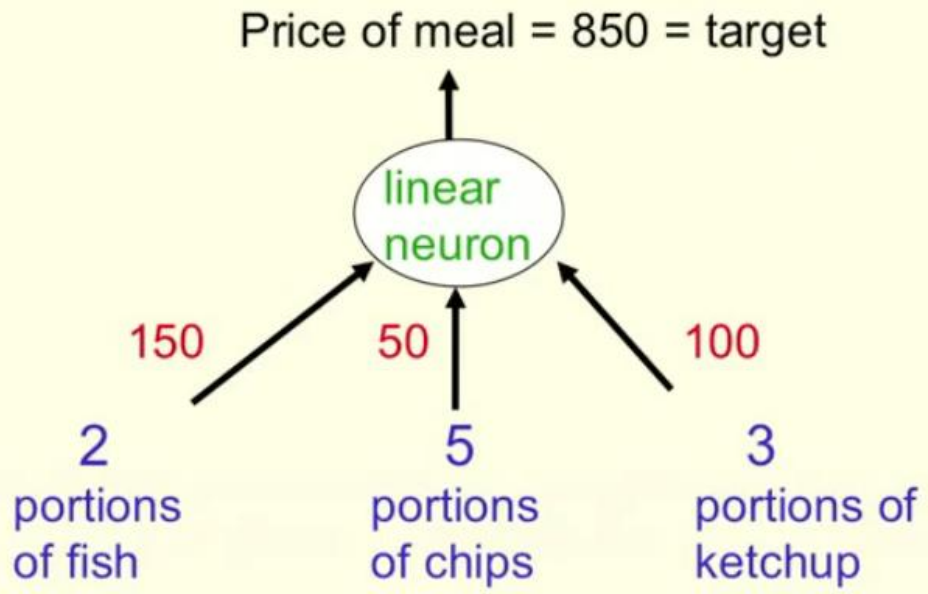
$$price = x_{fish} w_{fish} + x_{chips} w_{chips} + x_{ketchup} w_{ketchup}$$

- The prices of the portions are like the weights in of a linear neuron.

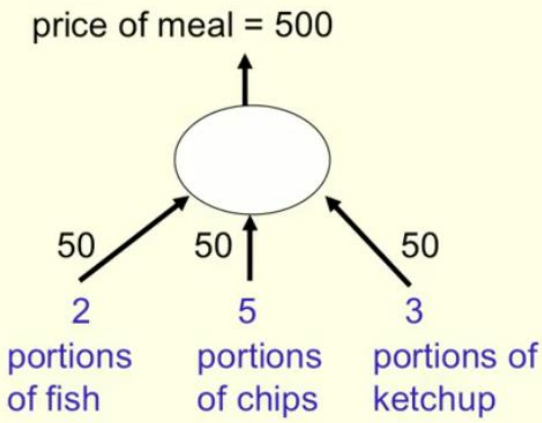
$$\mathbf{w} = (w_{fish}, w_{chips}, w_{ketchup})$$

- We will start with guesses for the weights and then adjust the guesses slightly to give a better fit to the prices given by the cashier.

The true weights used by the cashier



A model of the cashier with arbitrary initial weights



- Residual error = 350
- The "delta-rule" for learning is:
  $\Delta w_i = \varepsilon \, x_i \, (t - y)$
- With a learning rate ε of 1/35, the weight changes are +20, +50, +30
- This gives new weights of 70, 100, 80.
  - Notice that the weight for chips got worse!

Epsilon is the learning rate

Deriving the delta rule

- Define the error as the squared residuals summed over all training cases:

$$E = \frac{1}{2} \sum_{n \in training} (t^n - y^n)^2$$

- Now differentiate to get error derivatives for weights

$$\frac{\partial E}{\partial w_i} = \frac{1}{2} \sum_n \frac{\partial y^n}{\partial w_i} \frac{dE^n}{dy^n}$$

$$= -\sum_n x_i^n \, (t^n - y^n)$$

- The batch delta rule changes the weights in proportion to their error derivatives summed over all training cases

$$\Delta w_i = -\varepsilon \frac{\partial E}{\partial w_i} = \sum_n \varepsilon \, x_i^n (t^n - y^n)$$

Behaviour of the iterative learning procedure

- Does the learning procedure eventually get the right answer?
  - There may be no perfect answer.
  - By making the learning rate small enough we can get as close as we desire to the best answer.
- How quickly do the weights converge to their correct values?
  - It can be very slow if two input dimensions are highly correlated. If you almost always have the same number of portions of ketchup and chips, it is hard to decide how to divide the price between ketchup and chips.

The relationship between the online delta-rule and the learning rule for perceptrons

- In perceptron learning, we increment or decrement the weight vector by the input vector.
  - But we only change the weights when we make an error.
- In the online version of the delta-rule we increment or decrement the weight vector by the input vector scaled by the residual error and the learning rate.
  - So we have to choose a learning rate. This is annoying.

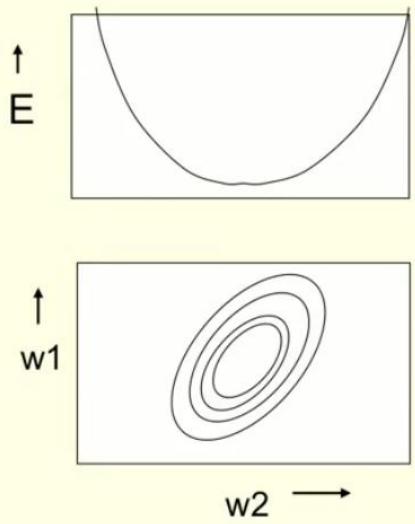
Effect of learning rate

Low – lot of time!

High – system impact?

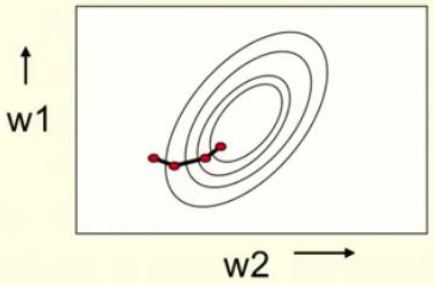
The error surface in extended weight space

- The error surface lies in a space with a horizontal axis for each weight and one vertical axis for the error.
  - For a linear neuron with a squared error, it is a quadratic bowl.
  - Vertical cross-sections are parabolas.
  - Horizontal cross-sections are ellipses.
- For multi-layer, non-linear nets the error surface is much more complicated.

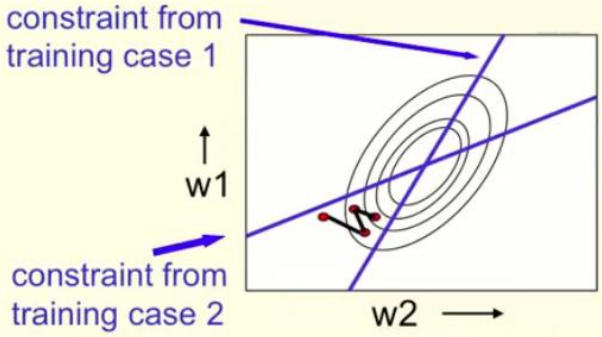


Online versus batch learning

- The simplest kind of batch learning does steepest descent on the error surface.
  - This travels perpendicular to the contour lines.

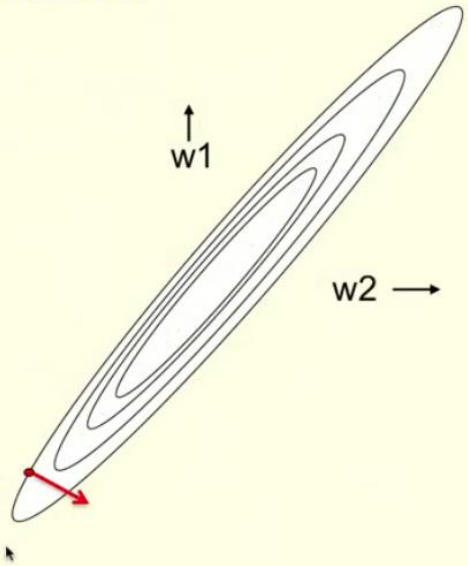


- The simplest kind of online learning zig-zags around the direction of steepest descent:



Why learning can be slow

- If the ellipse is very elongated, the direction of steepest descent is almost perpendicular to the direction towards the minimum!
  - The red gradient vector has a large component along the short axis of the ellipse and a small component along the long axis of the ellipse.
  - This is just the opposite of what we want.



Logistic neurons

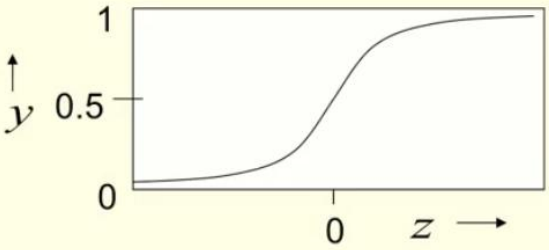
- These give a real-valued output that is a smooth and bounded function of their total input.

$$z = b + \sum_i x_i w_i \qquad y = \frac{1}{1+e^{-z}}$$

Logistic neurons

- These give a real-valued output that is a smooth and bounded function of their total input.
  - They have nice derivatives which make learning easy.

$$z = b + \sum_i x_i w_i \qquad y = \frac{1}{1+e^{-z}}$$



The derivatives of a logistic neuron

- The derivatives of the logit, z, with respect to the inputs and the weights are very simple:

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

$$\frac{\partial z}{\partial w_i} = x_i \qquad \frac{\partial z}{\partial x_i} = w_i$$

- The derivative of the output with respect to the logit is simple if you express it in terms of the output:

$$y = \frac{1}{1+e^{-z}}$$

$$\frac{dy}{dz} = y\,(1-y)$$

Using the chain rule to get the derivatives needed for learning the weights of a logistic unit

- To learn the weights we need the derivative of the output with respect to each weight:

$$\frac{\partial y}{\partial w_i} = \frac{\partial z}{\partial w_i}\frac{dy}{dz} = x_i \; y\,(1-y)$$

$$\frac{\partial E}{\partial w_i} = \sum_n \frac{\partial y^n}{\partial w_i}\frac{\partial E}{\partial y^n} = -\sum_n x_i^n \; y^n \; (1-y^n) \; (t^n - y^n)$$

Using the chain rule to get the derivatives needed for learning the weights of a logistic unit

- To learn the weights we need the derivative of the output with respect to each weight:

$$\frac{\partial y}{\partial w_i} = \frac{\partial z}{\partial w_i}\frac{dy}{dz} = x_i \; y\,(1-y)$$

delta-rule

$$\frac{\partial E}{\partial w_i} = \sum_n \frac{\partial y^n}{\partial w_i}\frac{\partial E}{\partial y^n} = -\sum_n \boxed{x_i^n} \; y^n \; (1-y^n) \; \boxed{(t^n - y^n)}$$

delta-rule

$$\frac{\partial E}{\partial w_i} = \sum_n \frac{\partial y^n}{\partial w_i}\frac{\partial E}{\partial y^n} = -\sum_n \boxed{x_i^n} \; \boxed{y^n \; (1-y^n)} \; \boxed{(t^n - y^n)}$$

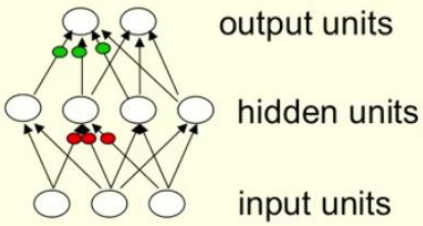
extra term = slope of logistic

Learning with hidden units (again)

- Networks without hidden units are very limited in the input-output mappings they can model.
- Adding a layer of hand-coded features (as in a perceptron) makes them much more powerful but the hard bit is designing the features.
  - We would like to find good features without requiring insights into the task or repeated trial and error where we guess some features and see how well they work.
- We need to automate the loop of designing features for a particular task and seeing how well they work.

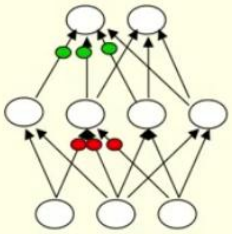
Learning by perturbing weights
(this idea occurs to everyone who knows about evolution)

- Randomly perturb one weight and see if it improves performance. If so, save the change.
  - This is a form of reinforcement learning.
  - Very inefficient. We need to do multiple forward passes on a representative set of training cases just to change one weight. Backpropagation is much better.
  - Towards the end of learning, large weight perturbations will nearly always make things worse, because the weights need to have the right relative values.



Learning by using perturbations

- We could randomly perturb all the weights in parallel and correlate the performance gain with the weight changes.
  - Not any better because we need lots of trials on each training case to "see" the effect of changing one weight through the noise created by all the changes to other weights.
- A better idea: Randomly perturb the activities of the hidden units.
  - Once we know how we want a hidden activity to change on a given training case, we can compute how to change the weights.
  - There are fewer activities than weights, but backpropagation still wins by a factor of the number of neurons.



The idea behind backpropagation

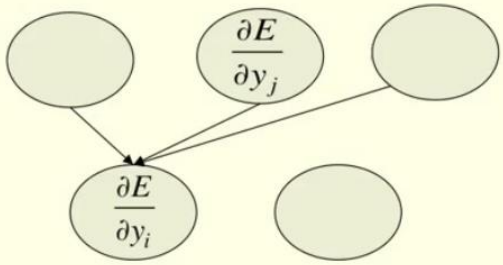
- We don't know what the hidden units ought to do, but we can compute how fast the error changes as we change a hidden activity.
  - Instead of using desired activities to train the hidden units, use error derivatives w.r.t. hidden activities.
  - Each hidden activity can affect many output units and can therefore have many separate effects on the error. These effects must be combined.
- We can compute error derivatives for all the hidden units efficiently at the same time.
  - Once we have the error derivatives for the hidden activities, its easy to get the error derivatives for the weights going into a hidden unit.

Sketch of the backpropagation algorithm on a single case

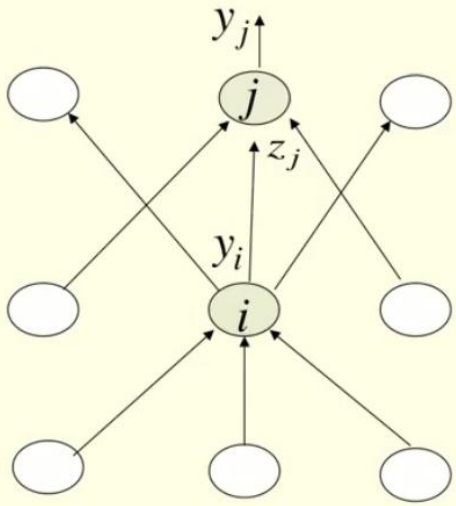
- First convert the discrepancy between each output and its target value into an error derivative.
- Then compute error derivatives in each hidden layer from error derivatives in the layer above.
- Then use error derivatives *w.r.t.* activities to get error derivatives *w.r.t.* the incoming weights.

$$E = \frac{1}{2} \sum_{j \in output} (t_j - y_j)^2$$

$$\frac{\partial E}{\partial y_j} = -(t_j - y_j)$$



Backpropagating dE/dy



$$\frac{\partial E}{\partial z_j} = \frac{dy_j}{dz_j}\frac{\partial E}{\partial y_j} = y_j(1-y_j)\frac{\partial E}{\partial y_j}$$

$$\frac{\partial E}{\partial y_i} = \sum_j \frac{dz_j}{dy_i}\frac{\partial E}{\partial z_j} = \sum_j w_{ij}\frac{\partial E}{\partial z_j}$$

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j} = y_i\frac{\partial E}{\partial z_j}$$