

Week 2: Types of neural network architectures

- Feed-forward neural networks

- Recurrent neural networks

- Symmetrically connected neural networks

Perceptron: The first generation of neural networks

- The standard Perceptron architecture

- Types of Perceptron neural networks

- A geometrical view of the perceptrons

 - Weight space

 - The cone of feasible solutions

 - Why the learning works

 - What perceptrons can't do - Limitations of perceptrons

Week 3: Backpropagation algorithm - The backpropagation learning procedure

- Learning the weights of a linear neuron

 - Why the perceptron learning procedure cannot be generalized to hidden layers

 - A different way to show that a learning procedure makes progress

 - Linear neurons (linear filters)

 - Deriving the delta rule (with the purpose of changing weights slightly)

 - Behaviour of the iterative learning procedure

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

- The error surface for a linear neuron

 - The error surface in extended weight space

 - Online versus batch learning

 - Why learning can be slow

- Learning the weights of a logistic output neuron

 - Logistic neurons

- The backpropagation algorithm

 - Learning by perturbing (nhiều loạn) weights (not work very well)

 - The idea behind backpropagation

 - Sketch of the backpropagation algo on a single case

 - How to use the derivatives computed by the backpropagation algo

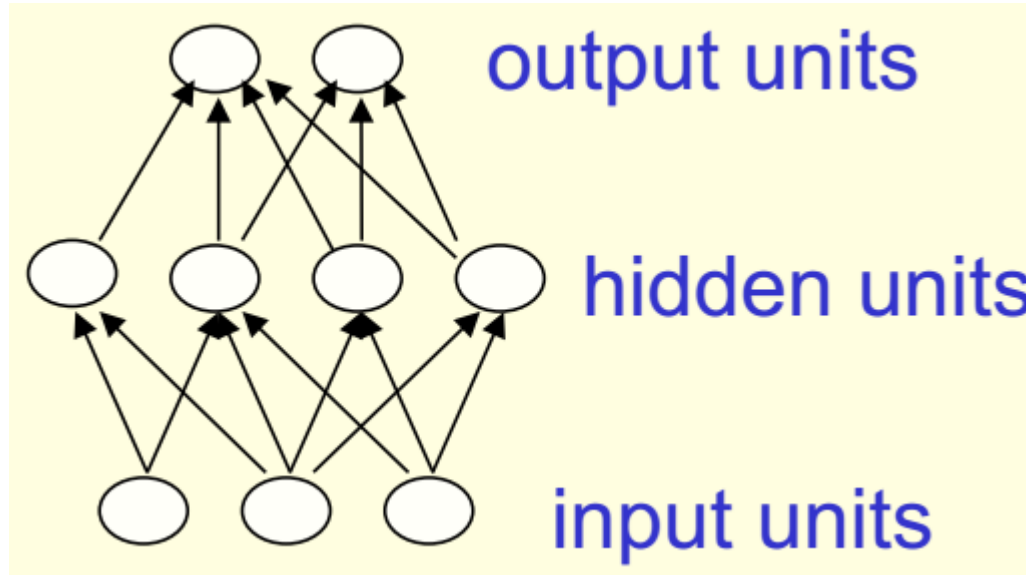
Week 4: Learning feature vectors for words

- 4.1. Learning to predict the next word

Week 2: Types of neural network architectures

Feed-forward neural networks

- having the input layer, (hidden layers), the output in this ordering.

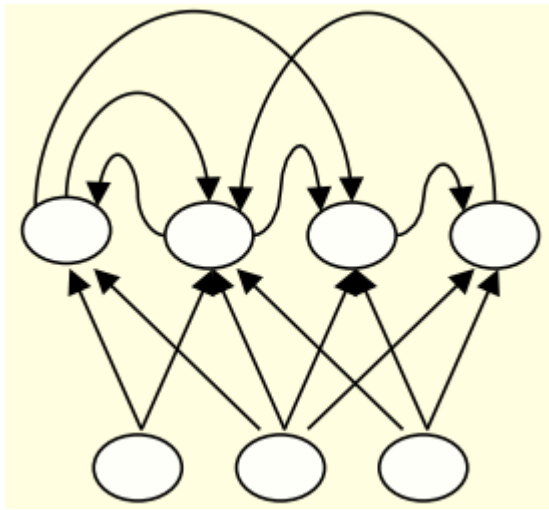


The first layer is the input and the last layer is the output. If there is more than (\geq) one hidden layer
→ **deep** neural networks.

The activities of the neurons in each layer are a non-linear function of the activities in the layer **below**.
→ compute a series of transformations that change the similarities between cases.

Recurrent neural networks

- having additional cycles, which can come back the previous nodes from the arrows.
- different from the FFNN is that have the same weight at each time slice (one hidden layer per time slice), and each input is captured at each time slice → the output also is in each time slice.
- suitable for modeling the sequence data.



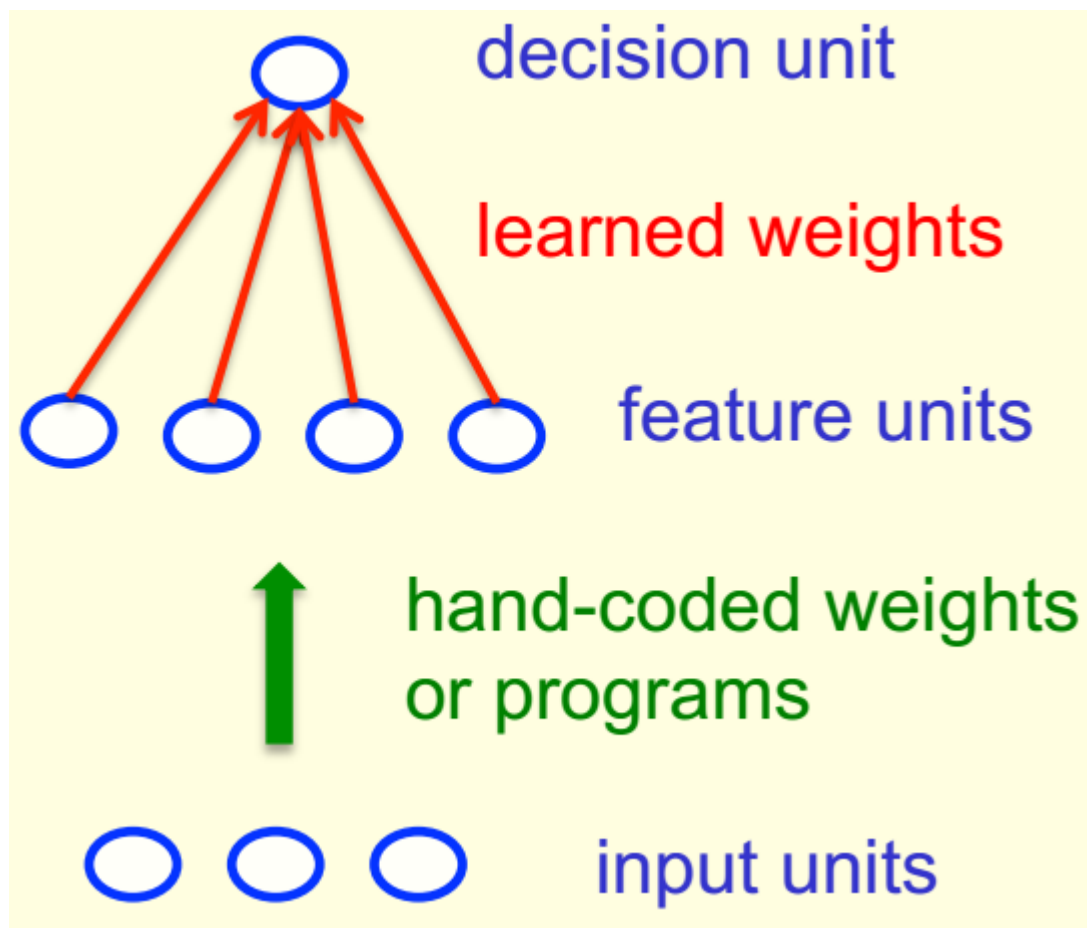
Symmetrically connected neural networks

- like recurrent networks, but the connections between units are symmetrical (same weight in both directions).
- more restricted, since obey an energy function (?) (cannot model cycles).
- symmetrically connected nets **without** hidden units = Hopfield nets.

Perceptron: The first generation of neural networks

The standard Perceptron architecture

- is a particular example of a statistical pattern recognition system.



- Convert the raw input vector into a vector of feature activation (use **hand-written programs, based on common-sense to define the features**).
- **Learn** how to weight each of the feature activations to get a single scalar quantity (feature activities \times the learned weights \rightarrow the sum of feature activities \times learned weights = this quantity).
- If this quantity is above ($>$) some threshold, decide that the input vector is a positive example of the target class.

Types of Perceptron neural networks

1. Binary threshold neurons (decision units)

- is also called McCulloch-Pitts.
- first, compute the weighted sum of the inputs from other neurons (plus a bias):

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

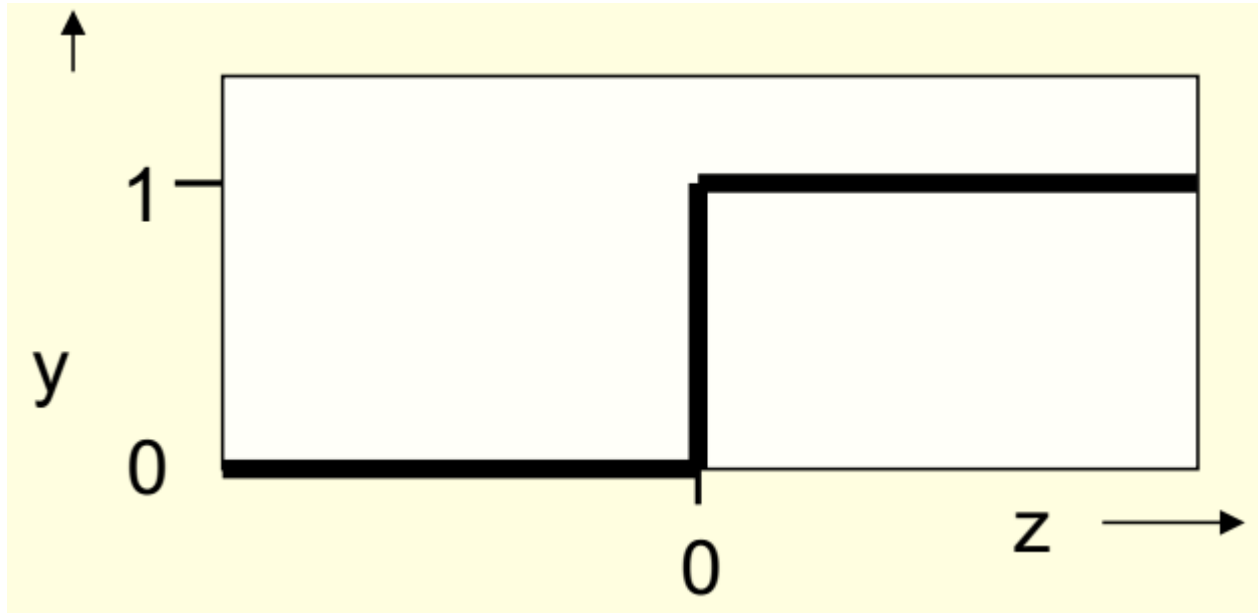
In matrix form:

$$z = b + x^T w$$

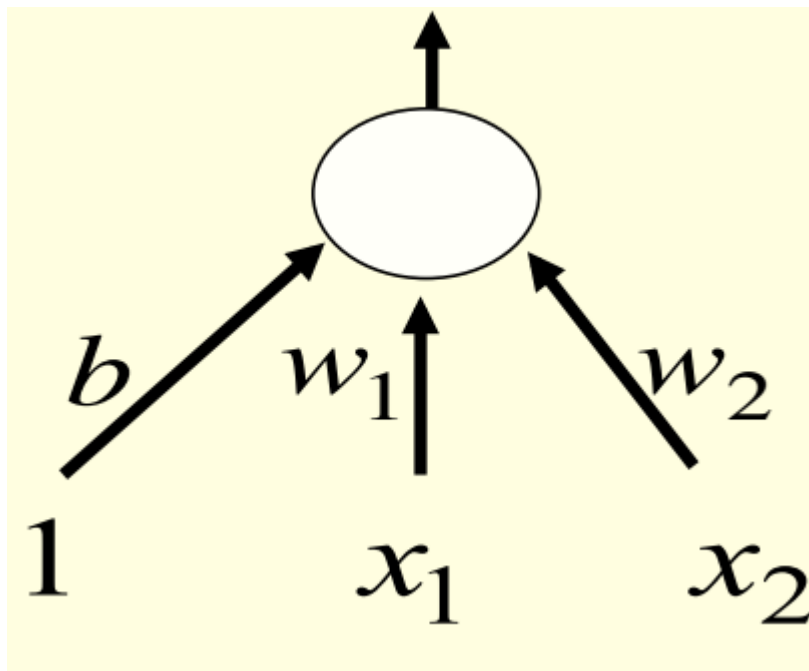
with b is a bias.

- then, output a 1 if the weighted sum exceeds zero.

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



- How to learn biases using the same rule as used for learning weights
 - the bias = the negative of the threshold ($b = -\theta$) and vice versus.
 - use the 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 = $-\theta$ (threshold).
 - Pick training cases using any policy that ensures every training case will keep getting picked without waiting too long. Then we have picked a training case with 3 options:
 - the output unit is correct → don’t change the weights.

- the output unit incorrectly outputs a zero (the output should be a 1) → **add** the input vector **to** the weight vector of perceptron.
- the output unit incorrectly outputs a one (the output should be a 0) → **subtract** the input vector **from** the weight vector.

Suppose we have a 2-dimensional input $x = (0.5, -0.5)$ connected to a neuron with weights $w = (2, -1)$ and the bias $b = 0.5$.

Furthermore, suppose the target for x is $t = 0$.

In this case, we will use a binary threshold neuron for the output so that

$$y = \begin{cases} 1 & \text{if } x^T w + b \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

What will the weights and bias be after applying one step of the perceptron learning algorithm?

- ☐ $w = (1.5, -0.5), b = -1.5$
- ☒ $w = (1.5, -0.5), b = -0.5$

Correct

x is currently misclassified as 1 instead of 0, so the perceptron algorithm will proceed as follows:

$$(w, b) \leftarrow (w, b) - (x, 1) = (2, -1, 0.5) - (0.5, -0.5, 1) = (1.5, -0.5, -0.5).$$

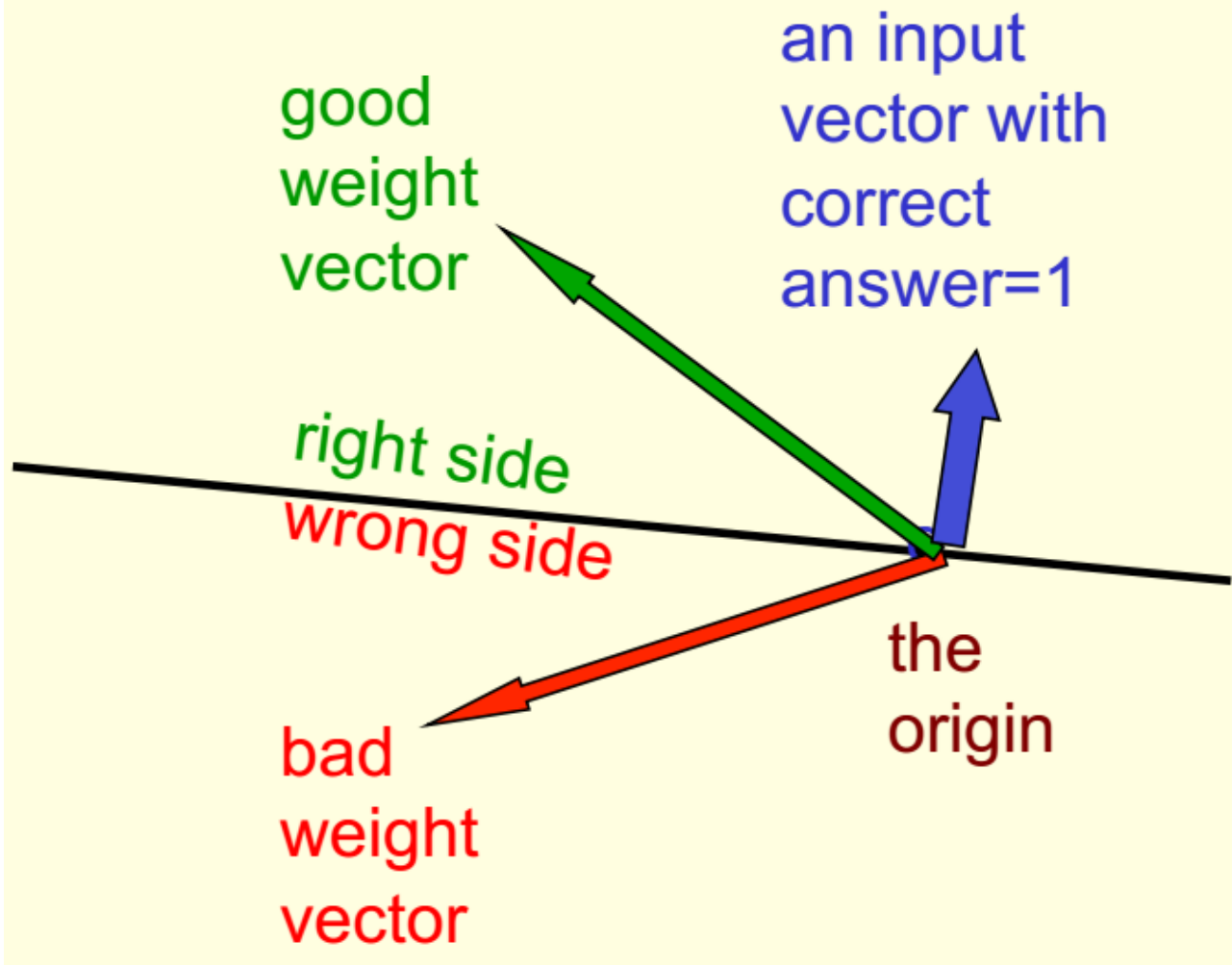
- ☐ $w = (2.5, -1.5), b = 0.5$
- ☐ $w = (-1.5, 0.5), b = 1.5$

A geometrical view of the perceptrons

Weight space

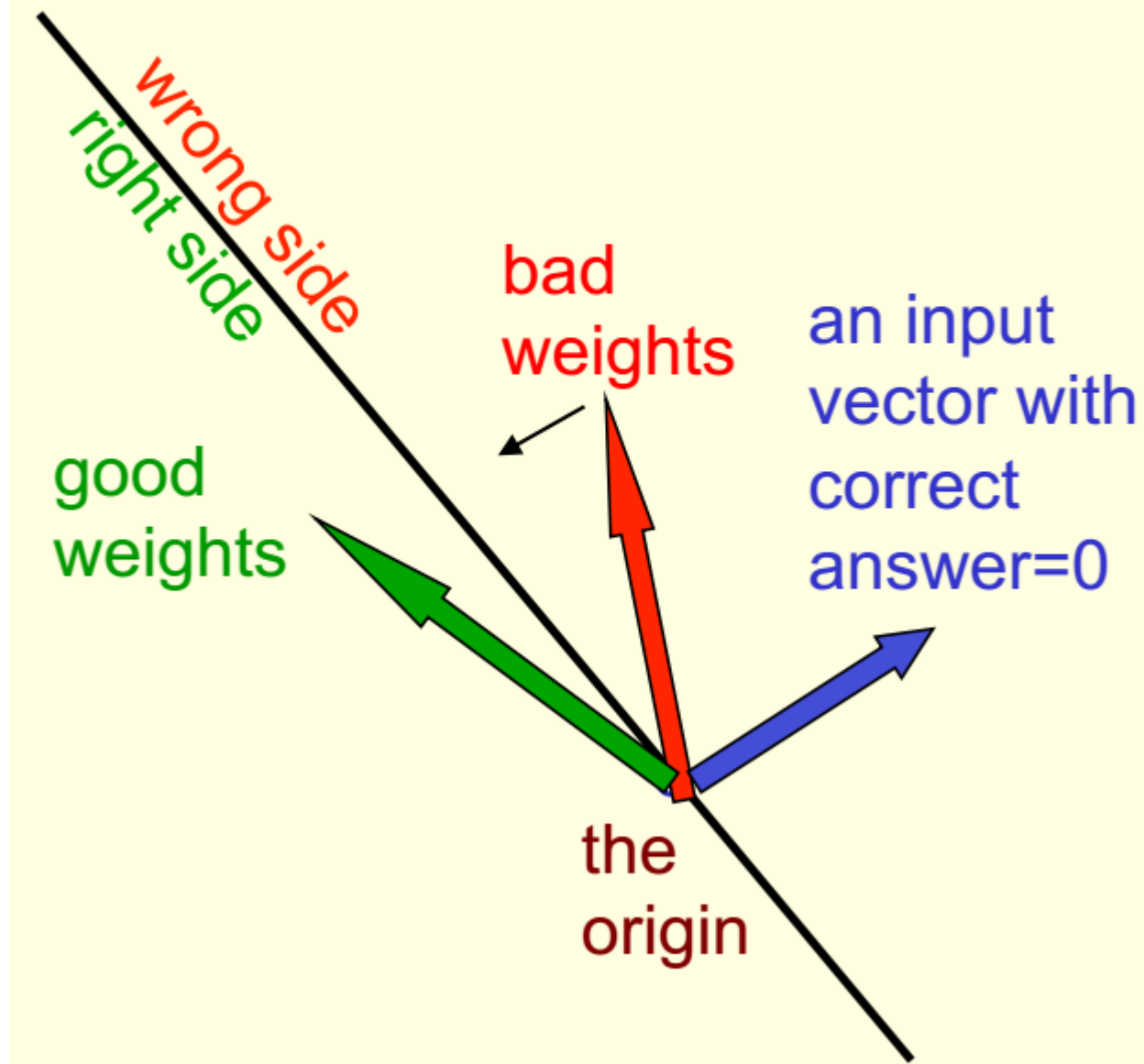
- one dimension per weight.
- a point in the space represents a particular setting of all the weights.
- **except from** the threshold (the bias weight term), each training case can be represented as a hyperplane through the origin in the weight space (mỗi mẫu training được coi như một siêu phẳng đi qua gốc tọa độ trong không gian trọng số n-chiều).
 - the weights lie on one side of this hyper-plane to get the answer correct.

Weight space



- Explanation of the above image:
 - A training case defines a plane, which is 2D in the above image (the black line).
 - The plane goes through the origin and perpendicular to the input vector (the blue vector).
 - Consider a training case in which the correct answer is one (is 1) → the weight vector of the correct answer must to be on the correct side of the hyperplane (the same side of the hyperplane as the direction in which the training vector points) (cùng chiều với training vector chỉ đến bên nào của siêu phẳng) (in here is the green vector).
 - Any weight vector like the green one, the angle with the input vector (the blue one) will be $\leq 90^\circ$ → the scalar product (the dot product) of the input vector and the weight vector > 0 (since $\cos(\alpha \leq 90^\circ) > 0$ and because of eliminating the threshold, so just compare with 0 only).
 - Conversely for the wrong output (the output is 0) (the red vector).
- For the target correct answer is 0, we have another image like the below:

Weight space



In the weight space view, the weights represent points while the inputs represent planes. Another term for what the inputs represent is:

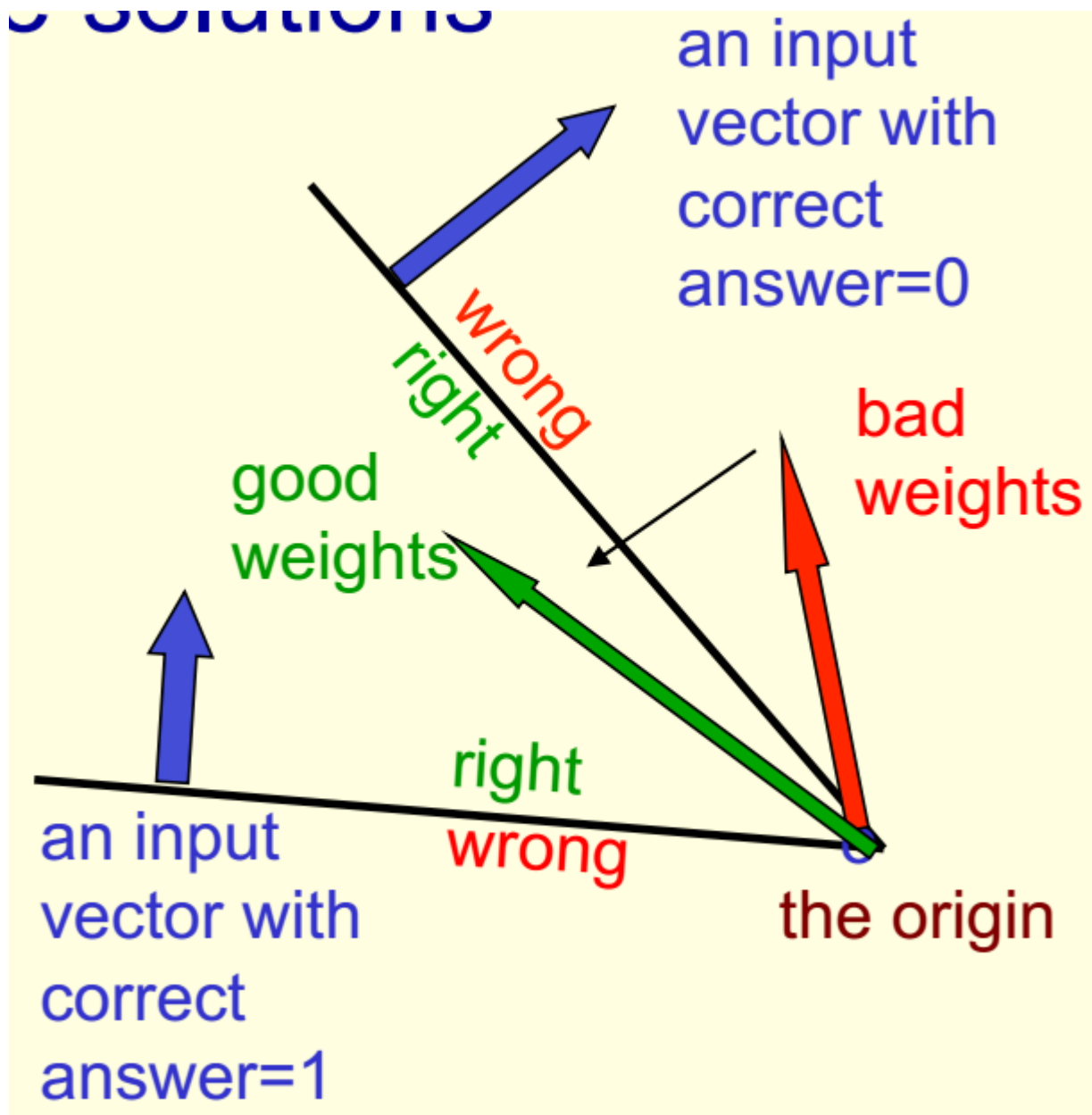
- ☐ Vectors
- ☒ Constraints

Correct

We can think of the inputs as partitioning the space into two halves. Weights lying in one half will get the answer correct while on the other half they will give the incorrect answer (which half is determined by the output class). That is, the inputs will *constrain* the set of weights that give the correct classification results.

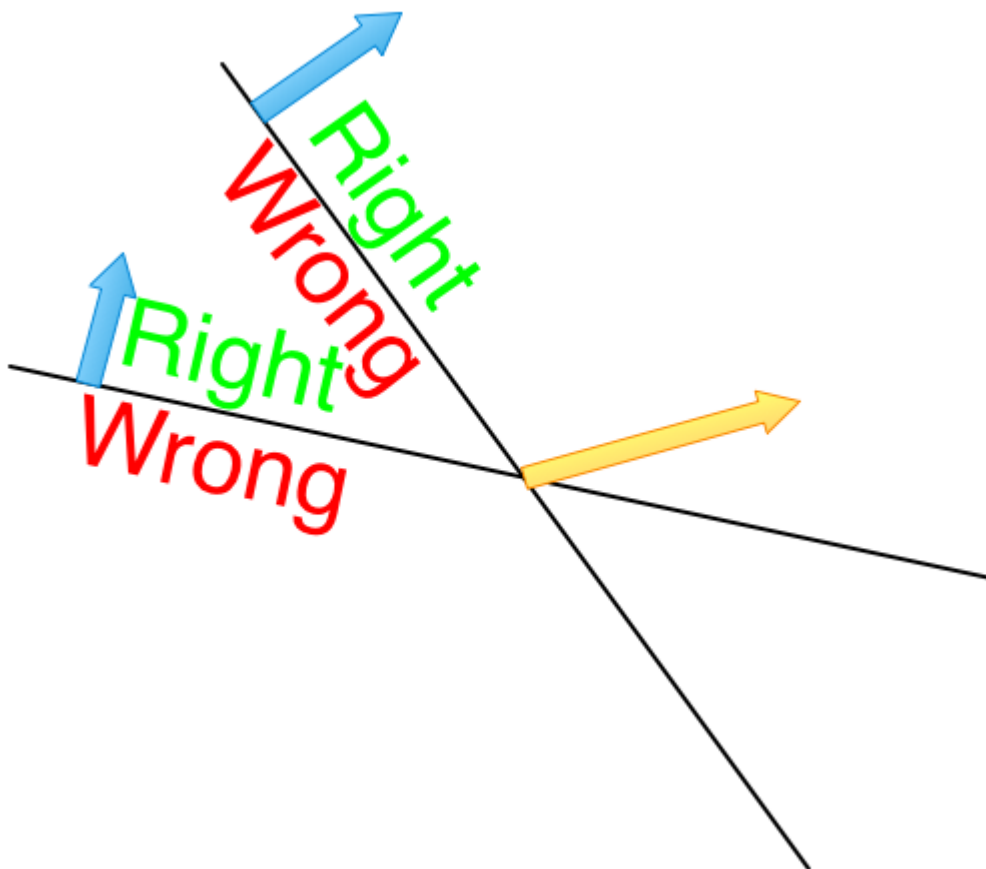
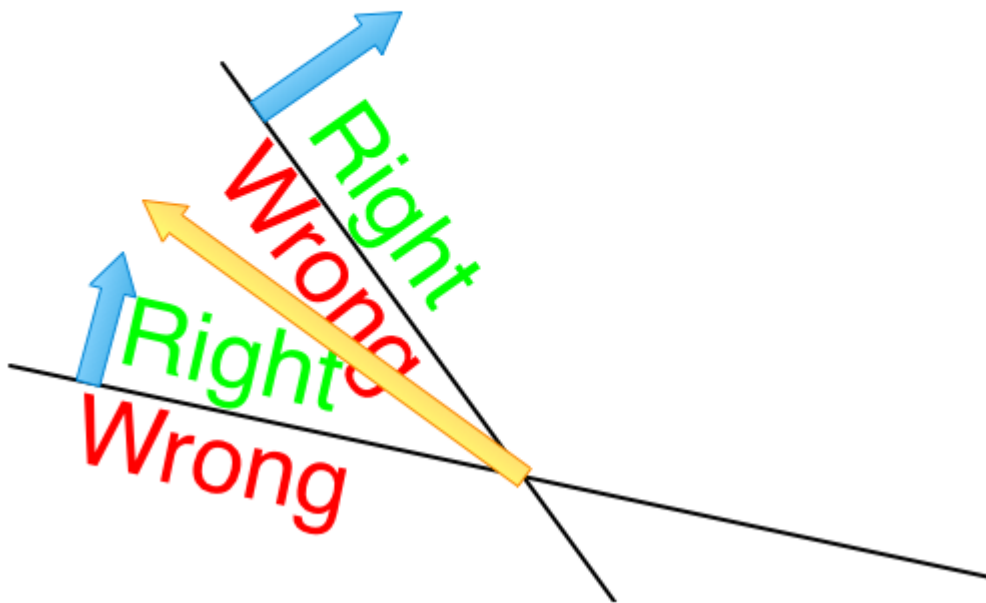
- ☐ Weights
- ☐ Space

The cone of feasible solutions

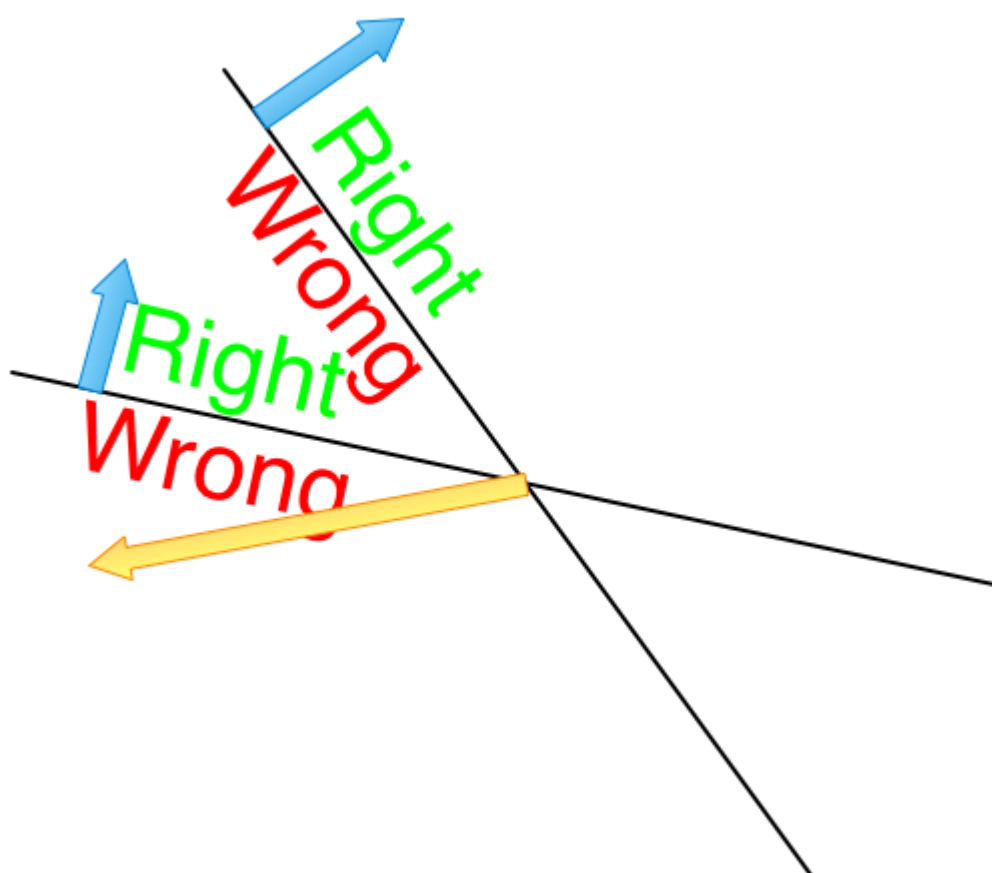
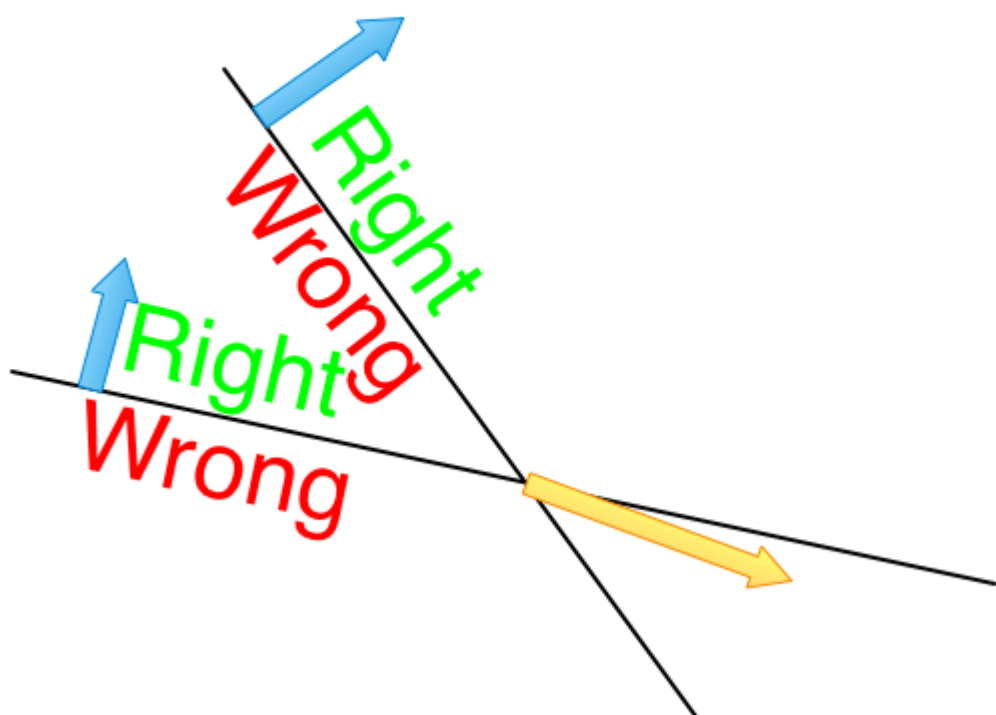


- For the above image, any weight vectors inside the cone will get the correct answer for both training cases. There doesn't have to be any cone like that (if it could be there are no weight vectors that get the right answers for all of the training cases).
- To get all training cases right, we need 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**. → the average of **two** good weight vectors is a good weight vector (is also a solution itself - also in that cone).
→ the problem is convex.

Consider two inputs that both have a label of 1. In the pictures below they are represented as hyperplanes and the feasible region for an input is any point within the region labelled "Right". Which of the following weight vectors, represented by a yellow arrow, will correctly classify the two inputs?

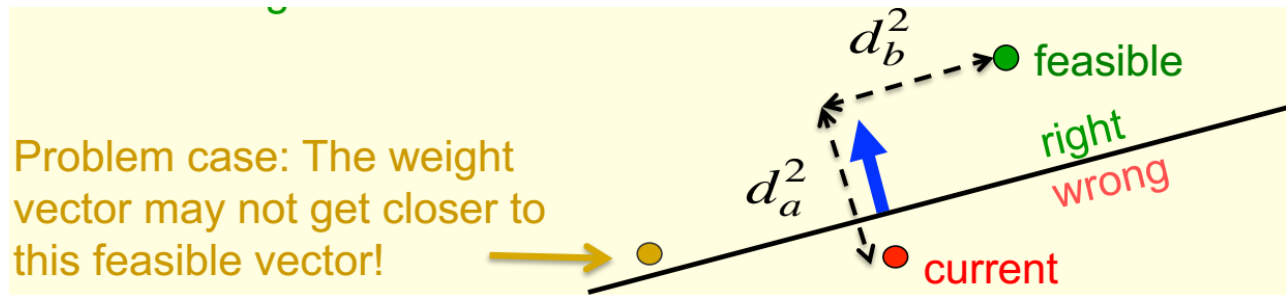


Correct



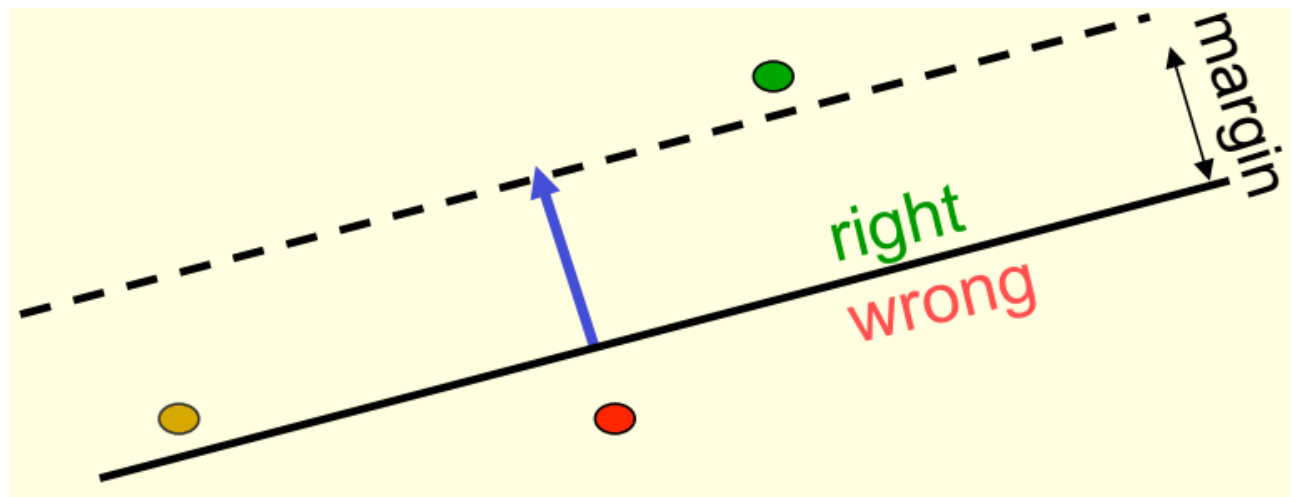
Why the learning works

- the perceptron learning procedure will get the weights into the cone of the feasible solutions (the convergence rule).
- Notation:
 - a feasible vector: a solution weight vector that get correct answer for all training cases (if these vectors exist) (is shown in the green dot in the following image).



- start with weight vector that gets some of the training cases wrong (the red dot).
 - The idea for proof: When getting a training case wrong, the **current** weight vector will be updated to closer to every **feasible** weight vector.
 - By the way of considering the squared distance $d_a^2 + d_b^2$ between the **current** weight vector and any **feasible** weight vector (the squared distance *along the line of the input vector* that defines the training case (d_a) and another squared difference *orthogonal* to that line (d_b)).
 - d_b is not change, but d_a will be gotten smaller.
- Every time the perceptron makes a mistake, the learning algo moves the **current** weight vector closer to all **feasible** weight vectors.

- However, unfortunately, there is **feasible** weight vector is very near the black line (the golden dot), the **current** weight vector is just in the wrong side, and the input vector is quite big → adding the input vector to the count weight vector, we get the further away from that golden feasible weight vector. (we need to fix it up by the **margin** as following).
- Define the **generously feasible** weight vectors = a weight vector that not only gets every training case right, but gets it right by at least a **certain margin**.
- The **margin** is at least as great as the length of the input vector that defines each constraint plane (\in the feasible region).
- Now, we have the cone of the feasible solutions and another cone of generously feasible solutions which gets the everything right answer by at least the size of the input vector.



→ 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 (input) vector (giảm một lượng ít nhất bằng với bình phương độ dài của vector đầu vào).

→ The informal sketch of a proof of convergence:

1. Every 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 **margin**)
2. The squared distance decreases **by at least** the squared length of the input vector and assuming that none of the input vectors are infinitesimally small (và giả thiết rằng không có vector đầu vào nào là nhỏ một cách vô hạn (nhỏ vô hạn)). That means:
3. After a finite number of mistakes, the weight vector must lie in the **feasible** region if this region exists.
 - Note: it doesn't have to lie in the **generously feasible** region, but it has to be into the **feasible** region to stop it making mistakes.

True or false: assuming the input and weights are finite, just because a set of inputs has a feasible space *does not* mean that they also have a generously feasible space.

☐ True

☒ False

Correct

Giả sử đầu vào và trọng số là hữu hạn, chỉ bởi vì **một tập hợp đầu vào** có khu vực khả thi (không gian khả thi - khả nghiệm) không có nghĩa rằng (điều đó không có nghĩa rằng) chúng cũng có một không gian khả thi mở rộng.

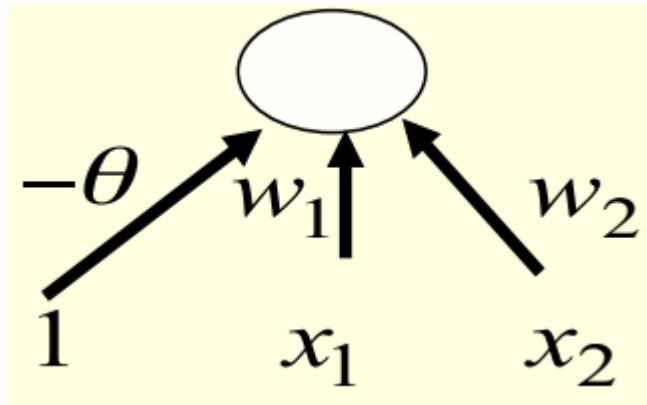
What perceptrons can't do - Limitations of perceptrons

1. For binary threshold neurons, an output unit cannot tell if two single bit feature are the same or not (XOR function):

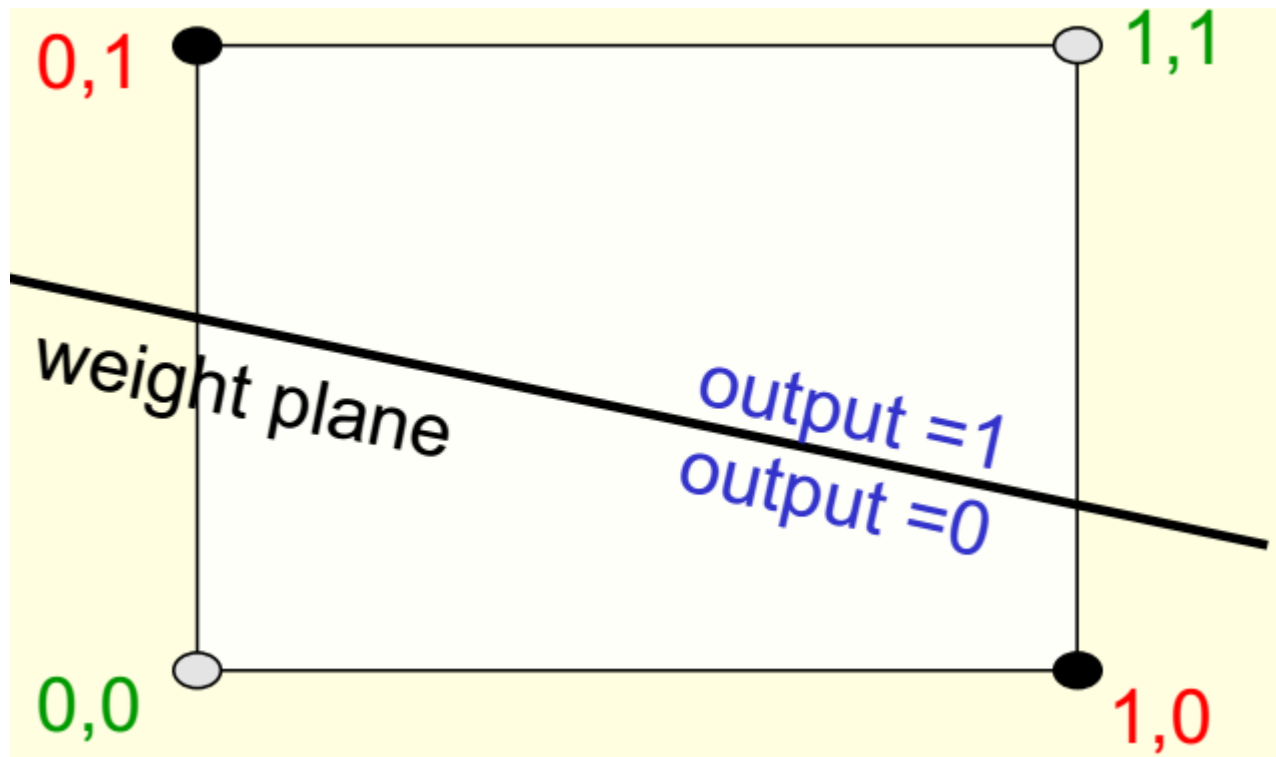
- Positive cases (same): $(1, 1) \rightarrow 1$; $(0, 0) \rightarrow 1$
- Negative cases (different): $(1, 0) \rightarrow 0$; $(0, 1) \rightarrow 0$
- The four input-output pairs give four inequalities that are impossible to satisfy:

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

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



- A geometric view of limitations of binary threshold neurons:



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

- Each input vector is a point in the “input-data” space.
- A weight vector defines a plane in “data-space”.
- The weight plane is perpendicular to the weight vector and misses the origin (cách gốc tọa độ) by a distance equal to the threshold.

For the above image, the positive and negative cases cannot be separated by a **plane**. Two data cases in green, we need to put an output of one, and two data cases in red, an output of zero.

Assuming that no three points are collinear, what is the minimum number of 2D inputs do we need before we can construct a dataset that the perceptron cannot solve? Note that with the bias these inputs will be 3D, but the extra dimension is trivially set to 1.

☐ 2

☐ 3

☒ 4

Correct

One way to see this is to write out a general linear system:

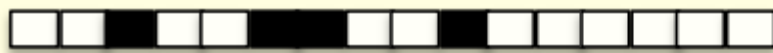
$$\begin{pmatrix} a & b & 1 \\ c & d & 1 \\ e & f & 1 \end{pmatrix}$$

We can see that since no three points are collinear, this system must be invertible, meaning it is possible to satisfy any set of constraints defined by the variables. An impossible to solve case was shown in the lecture for 4 points.

☐ 5

2. A devastating example for perceptrons is more general is when trying and discriminating simple patterns that have to be retain the identity (giữ lại được đặc tính riêng biệt của chúng) when you translate them with wrap-around.

- Discriminating simple patterns under translation with wrap-around:
- want to recognize the simple pattern and want to recognize it even it's translated.
- For example, suppose use pixels as features.
- The question: can a binary threshold unit discriminate between different patterns that have the same number of **on** pixels? (one is positive example and other is negative example) → the answer is **no** if the discrimination have to work when the patterns are translated and if the patterns can wrap-around then translate.



pattern A



pattern A



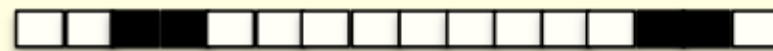
pattern A



pattern B



pattern B



pattern B

Suppose we have three patterns that belong to two classes. The first class contains patterns with 4 pixels on while the second class contains patterns with either 1 or 3 pixels on. Can a binary decision unit classify these correctly if we allow for translations with wraparound?

☒ Yes

Correct

If we set the weight coming from each pixel to be 1, and the bias to be -3.5, then any example with 3 pixels on will receive a total activation of -0.5, any example with 1 pixel on will receive a total activation of -2.5 and any example with 4 pixels on will receive an activation of 0.5. This correctly classifies the dataset.

☐ No

→ The perceptrons cannot discriminate (recognize) the classes (patterns) under translation if the wrap-around is allowed. Because a perceptron cannot learn to do if the transformations form a group (but translations with wrap-around form a group).

→ To deal with, a perceptron need to use multiple feature units to recognize transformations of informative sub-patterns (the **tricky** part of pattern recognition must be solved by the hand-coded feature detectors, not the learning procedure).

→ Learning with hidden units.

- To sum up, the limitations of perceptrons are:

- The table look-up won't generalize. (Bảng tra cứu không tổng quát - XOR function).
- Cannot recognize the patterns that their transformations form a group.
- only consider to discriminate linearly.
- For hidden units:
 - more layers of linear units do not help (its still linear).
 - fixed output non-linearities are not enough.
 - multiple layers of *adaptive*, non-linear hidden units (the difficulty is how we can train such nets?)
 - need an efficient way of adapting **all** the weights, not just the last layer.
 - learning the weights going into hidden units = learning features (difficult since nobody tells directly what the hidden units should do and not should do).

Week 3: Backpropagation algorithm - The backpropagation learning procedure

Learning the weights of a linear neuron

Why the perceptron learning procedure cannot be generalized to hidden layers

- The task of the perceptron convergence procedure is to ensure that every time, the weights change, they (the weights) 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).
- **multi-layer** neural networks do **not** use the perceptron learning procedure (→ should **never** have been called multi-layer perceptrons).
 - a different way to make the learning procedure works.

A different way to show that a learning procedure makes progress

- Instead of proofing the weights get closer to a good set of weights, show that the **actual** output values get closer to the **target** output values.
- This has many advantages because of:
 - 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 (bad solution)) (← solving the limitation of perceptrons).
 - This learning is not true for perceptron learning.
- Let's explore the simplest example is a linear neuron with a squared error measure.

Linear neurons (linear filters)

- is the neuron has the real-valued outputs which are the weighted sum of its inputs. (các giá trị đầu ra là số thực và các số thực này là tổng có tính trọng số của các giá trị đầu vào).

$$y = \sum_i w_i x_i = w^T x$$

The diagram shows the equation $y = \sum_i w_i x_i = \mathbf{w}^T \mathbf{x}$ on a yellow background. A blue arrow points from the text "weight vector" to the \mathbf{w} term. Another blue arrow points from the text "input vector" to the \mathbf{x} term. A third blue arrow points from the text "neuron's estimate of the desired output" to the y term.

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

weight vector

input vector

neuron's estimate of the desired output

- The aim of learning is to **minimize** the error summed over all training cases with the error is the **squared difference** between the **desired** output and the **actual** output.
- This problem can be solved by iterative solution or analytic solution (where to write down a set of equations, one equation per training case, and to solve for the best set of weights).
- But do **not** use analytic method, because of its relying on it (the analytic solution) being linear and having a squared error measure.
- Iterative methods are usually less efficient but they are much more easier to generalize (a method that real neurons could use and a method that can be generalized to multi-layer, non-linear neural networks).
- The iterative approach: Start with random guesses for the weights and then adjust the guesses **slightly** to give a better fit to the observed (actual) output given.

Question 1:

What is the estimate of the price of the meal after running the iterative algorithm for one step ?

Recall that

The correct price for portions (2, 5, 3) is 850.

Initial guess of the weights (50, 50, 50) gave an estimate of 500.

After running the iterative algorithm for one step the revised guess of the weights is (70, 100, 80)

880

Correct Response

$70 * 2 + 100 * 5 + 80 * 3 = 880$. Note that we are closer to the correct answer now. Also note that the learning changed the weight of chips from 50 to 100 even though the initial guess of 50 was the correct weight. So individual weights did not all get closer to their correct values, but the final output did.

Question 2:

You have seen how the delta rule can be used to minimize the error by looking at one training case at a time. What is a good reason for using this kind of iterative method?

☐ We can avoid referring to minimizing the squared error as *linear regression*.

Un-selected is correct

☐ We can prove the convergence of the algorithm using the same reasoning as the perceptron algorithm.

Un-selected is correct

☒ An iterative method will scale more gracefully to larger datasets.

Correct

The first option is false but is intended to remind you that minimizing the squared error of a set of examples is equivalent to linear regression. The second option is also false because the perceptron convergence proof uses the fact that the weights always get

- ☒ An iterative method will scale more gracefully to larger datasets.

Correct

The first option is false but is intended to remind you that minimizing the squared error of a set of examples is equivalent to linear regression. The second option is also false because the perceptron convergence proof uses the fact that the weights always get closer to a set of optimal weights, which doesn't happen in this case. The third and fourth options are correct and complimentary: this general approach scales extremely well to large datasets and has the additional advantage that we can collect data as we run the iterative algorithm (that is, we run a step, go out and collect another example, run another step) meaning that we can keep collecting new data and learning better models. This approach to optimization is also known as *online learning*.

- ☐ We don't have to collect all of the examples before we begin learning.

This should be selected

well to large datasets and has the additional advantage that we can collect data as we run the iterative algorithm (that is, we run a step, go out and collect another example, run another step) meaning that we can keep collecting new data and learning better models. This approach to optimization is also known as *online learning*.

- ☒ We don't have to collect all of the examples before we begin learning.

Correct

The first option is false but is intended to remind you that minimizing the squared error of a set of examples is equivalent to linear regression. The second option is also false because the perceptron convergence proof uses the fact that the weights always get closer to a set of optimal weights, which doesn't happen in this case. The third and fourth options are correct and complimentary: this general approach scales extremely well to large datasets and has the additional advantage that we can collect data as we run the iterative algorithm (that is, we run a step, go out and collect another example, run another step) meaning that we can keep collecting new data and learning better models. This approach to optimization is also known as *online learning*.

Deriving the delta rule (with the purpose of changing weights slightly)

- The aim is to find the equation of Δw_i per one training case.
 - Define the error as “the squared residuals summed over all training case”.

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

t^n : the target/actual/desired output value.

y^n : the estimated output value.

use $\frac{1}{2}$ to eliminate with the exponential factor (2) when taking derivative.

- Differentiate to get error derivatives for weights (the partial derivative w.r.t weights).

$$\begin{aligned} \frac{\partial E}{\partial w_i} &= \frac{1}{2} \sum_n \frac{\partial y^n}{\partial w_i} \frac{dE^n}{dy^n} \text{ (chain rule)} \\ &= - \sum_n x_i^n (t^n - y^n) \end{aligned}$$

- The **batch** (one training case) delta rule changes the weights in proportion to (the learning rate ϵ) their error derivatives **summed over all training cases**.

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

Behaviour of the iterative learning procedure

- Does the learning procedure eventually get the right answer?
 - may be no perfect solution.
 - making the learning rate (ϵ) small enough \rightarrow get as close as desiring to the best answer (solution).
- How quickly do the weights converge to their correct values?
 - It depends.
 - It can be very slow if two input dimensions are highly correlated.

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

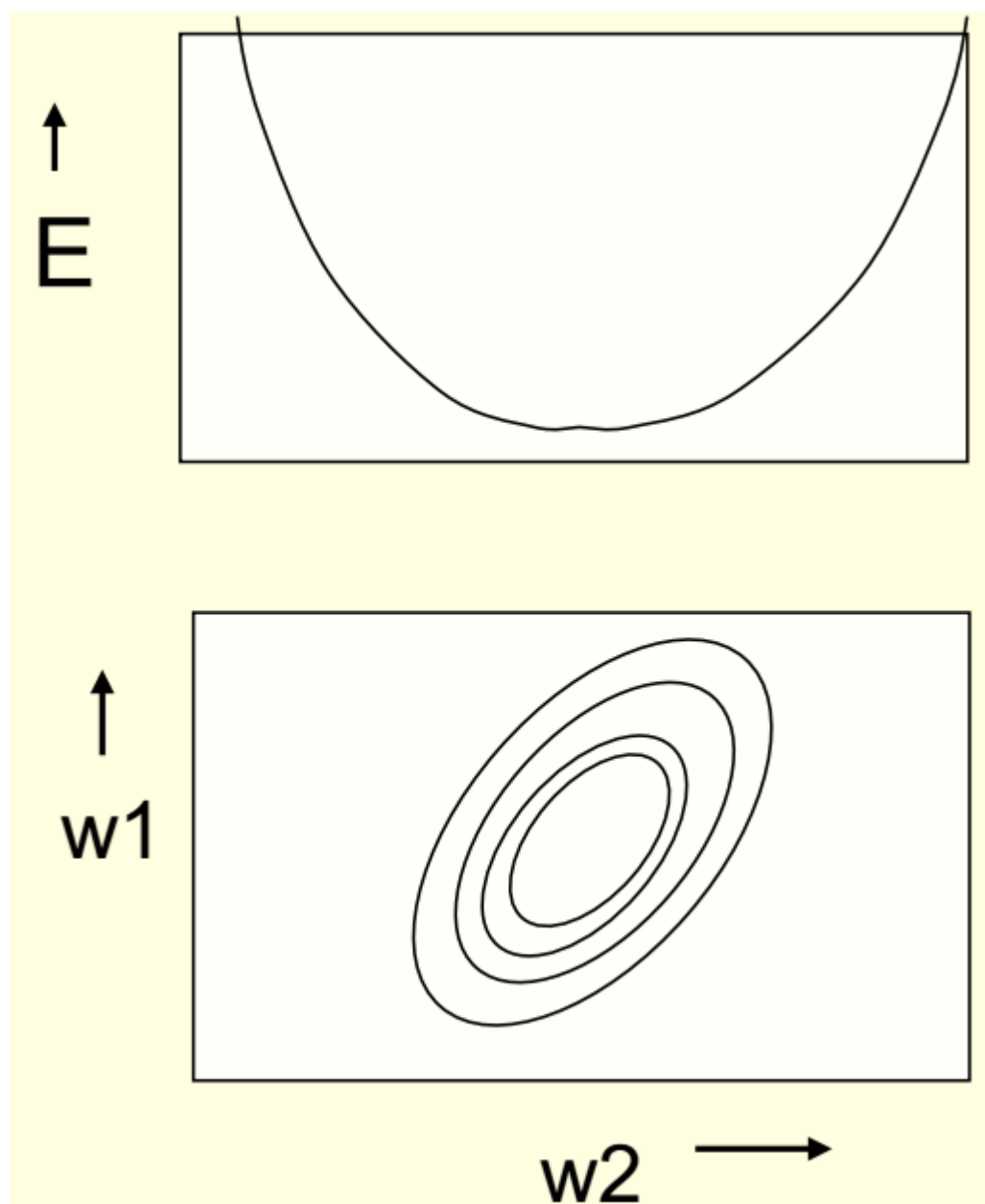
Perceptrons	Online delta-rule (the online version of the delta-rule)
increment/decrement the weight vector by the input vector $\Delta w_i = \pm x$	increment/decrement the weight vector by the input vector scaled by the residual error and the learning rate $\Delta w_i = \epsilon x(t^n - y^n)$
only change when making an error	

Perceptrons	Online delta-rule (the online version of the delta-rule)
	choose a learning rate

The error surface for a linear neuron

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 (the error surface) is a quadratic bowl.
 - Vertical cross-sections are parabolas.
 - Horizontal cross-sections are ellipses.



Question:

Suppose we have a dataset of two training points.

$$\begin{aligned}x_1 &= (1, -1) & t_1 &= 0 \\x_2 &= (0, 1) & t_2 &= 1\end{aligned}$$

Consider a network with two input units connected to a linear neuron with weights $w = (w_1, w_2)$. What is the equation for the error surface when using a squared error loss function ?

Hint: The squared error is defined as $\frac{1}{2} (w^T x_1 - t_1)^2 + \frac{1}{2} (w^T x_2 - t_2)^2$

☒ $E = \frac{1}{2} (w_1^2 + 2w_2^2 - 2w_1w_2 - 2w_2 + 1)$

Correct

The error summed over all training cases is

$$E = \frac{1}{2} (w_1 - w_2 - 0)^2 + \frac{1}{2} (0w_1 + w_2 - 1)^2 = \frac{1}{2} (w_1^2 + 2w_2^2 - 2w_1w_2 - 2w_2 + 1).$$

Note the quadratic form of the energy surface. For any fixed value of E , the contours

☐ $E = \frac{1}{2} (w_1^2 + 2w_2^2 - 2w_1w_2 - 2w_2 + 1)$

Correct

The error summed over all training cases is

$$E = \frac{1}{2} (w_1 - w_2 - 0)^2 + \frac{1}{2} (0w_1 + w_2 - 1)^2 = \frac{1}{2} (w_1^2 + 2w_2^2 - 2w_1w_2 - 2w_2 + 1).$$

Note the quadratic form of the energy surface. For any fixed value of E , the contours will be ellipses. For fixed values of w_1 , we get a parabolic relation between E and w_2 . Similarly for fixed values of w_2 .

☐ $E = \frac{1}{2} ((w_1 - w_2 - 1)^2 + w_2^2)$

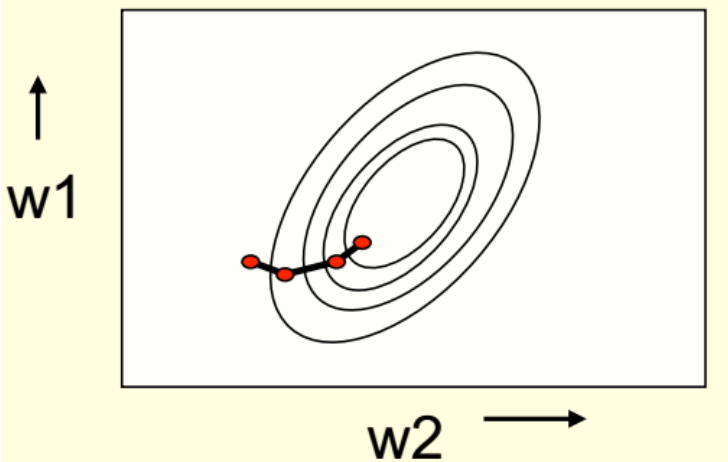
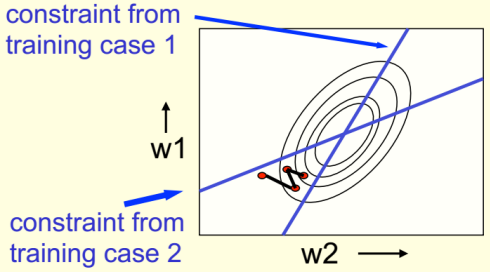
☐ $E = \frac{1}{2} (w_1^2 - 2w_2^2 + 2w_1w_2 + 1)$

☐ $E = \frac{1}{2} (w_1 - w_2)^2$

- For multi-layer, non-linear nets, the error surface is much more complicated.

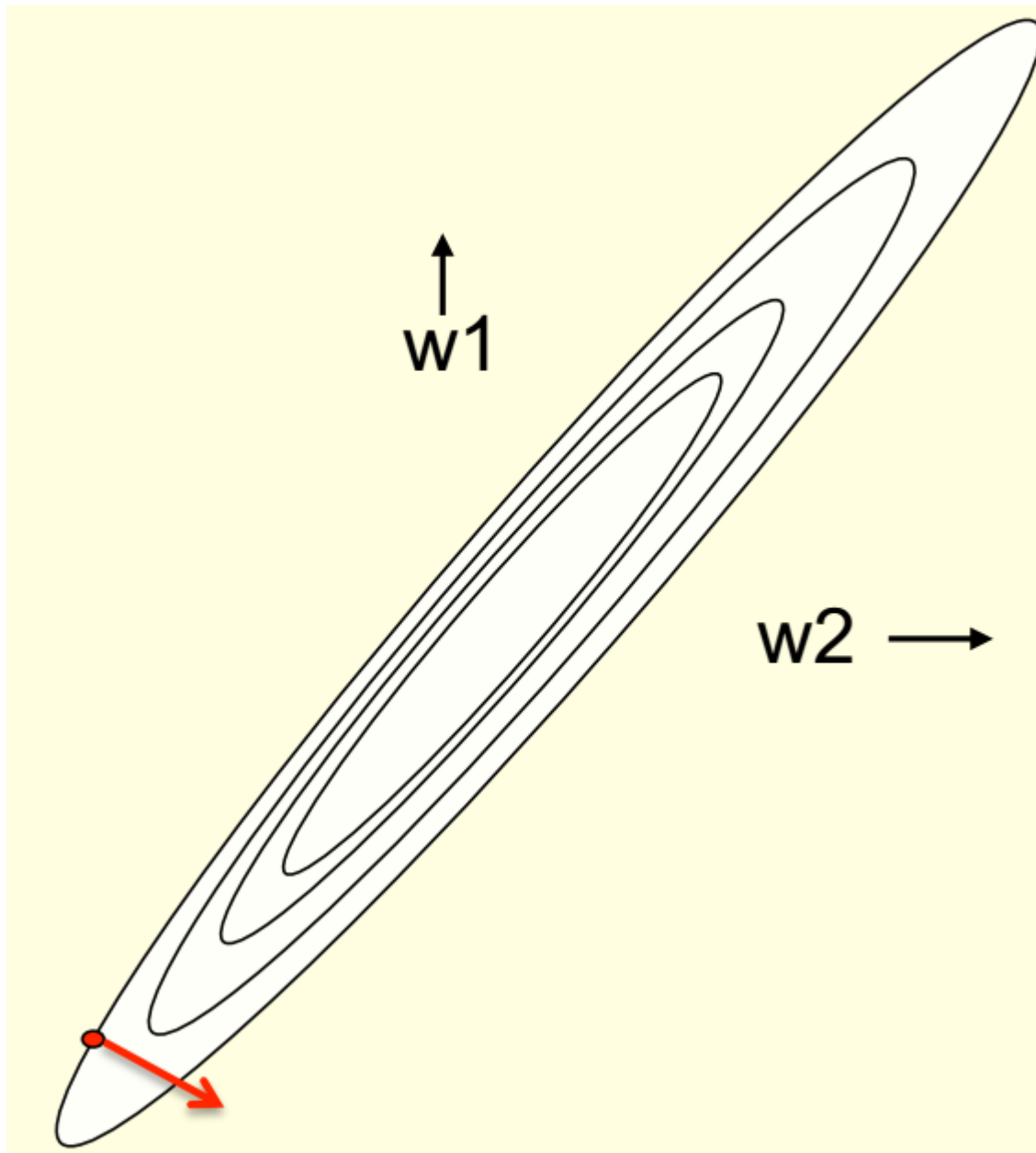
Online versus batch learning

- From the error surface, getting the picture of what's happening as we do gradient descent learning using the delta rule. What the delta rule does is it computes the derivative of the error w.r.t the weights, and if changing the weights in proportion to that derivative, that's equivalent to doing steepest descent on the error surface.
- For error surface (which are elliptical contour lines), what the delta rule does is taking the right angles to those elliptical contour lines (perpendicular to).
→ the batch learning (gradient in summed over **all** training cases).
- However, we could also do online learning (where after **each** training case, changing the weights in proportion to the gradient for that single training case) → much more like we did in perceptrons.
- The **change** in the weights moves towards one of these constraint planes (the blue line).
- There are two training cases (in the right image).
 - To get the first training case correct, must lie on the first blue line. And to get the second training case correct, must lie on the another blue line.
 - Start with one of those red points, compute the gradient on the first training case, then the delta rule will move perpendicularly towards that line.
 - And with other training case, move perpendicularly towards the other line.
 - If alternative between the two training cases, zig-zag backwards and forwards, moving towards the solution point which is where those two lines intersect (where the set of weights that is correct for both training cases).

The simplest kind of batch learning	The simplest kind of online learning
<p>does steepest descent on the error surface (this travels perpendicular to the contour lines) - làm sụt giảm theo hướng có độ dốc dốc nhất, tức là hướng di chuyển vuông góc với các đường cong</p>	<p>zig-zags around the direction of steepest descent - zig-zag theo hướng sụt giảm là mạnh nhất (theo hướng có độ dốc dốc nhất)</p>
	

Why learning can be slow

- Condition that makes learning very slow: If the ellipse is very elongated (thuôn dài ở 2 cực của elip) (is happen when lines correspond to training cases is almost parallel), and when seeing in the gradient descent, the direction of steepest descent is almost perpendicular to the direction towards the minimum.

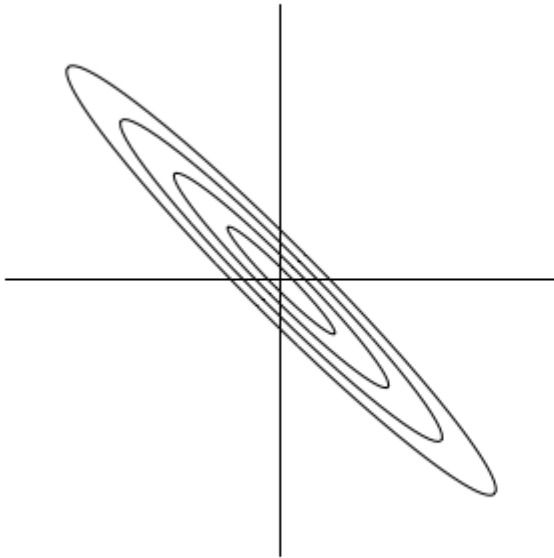


For the above image of the error surface, the red line is the gradient vector that has a long component along the short axis of the ellipse and a small component along the long axis of the ellipse. It means that, if we change each weight in proportion to the learning rate times the error derivative (in the simple steepest descent), it will towards the direction that the red gradient vector is longer, which is opposite of what we want (and is smaller in direction where we want to move a long way).

→ the gradient quickly move across the narrow axis of the ellipse and will take a long time to take along the long axis of the ellipse (that is of what we don't want).

Question:

If our initial weight vector is $(w_1, 0)$ for some real number w_1 , then on which of the following error surfaces can we expect steepest descent to converge poorly? Check all that apply.

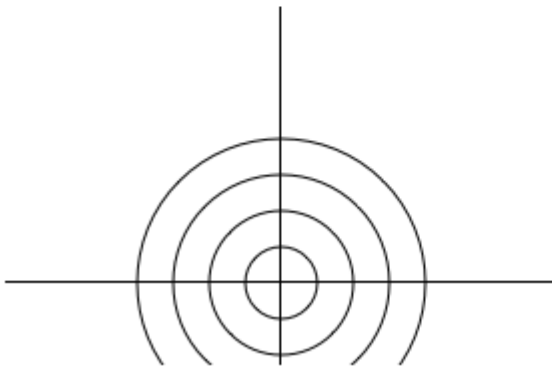


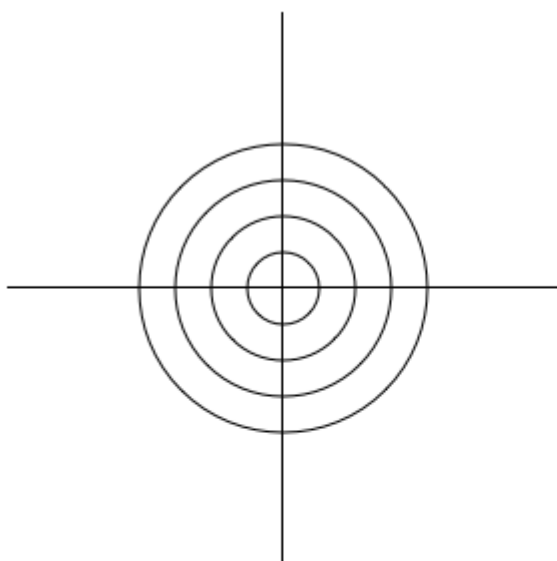
Correct

The first one is similar to the picture shown in the lectures. It is a diagonally oriented

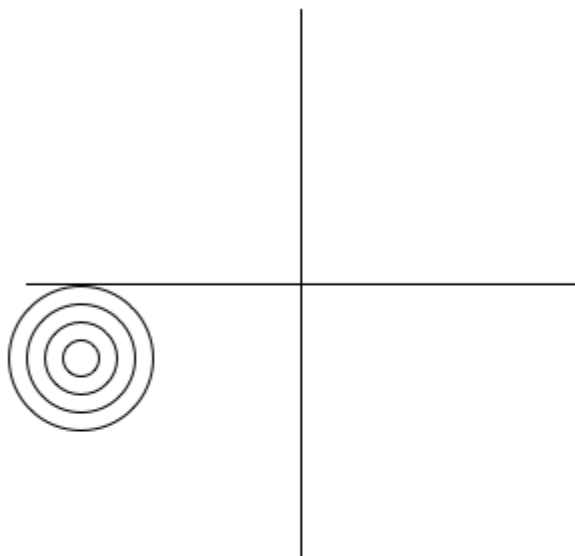
Correct

The first one is similar to the picture shown in the lectures. It is a diagonally oriented ellipse and steepest descent will still tend to zig-zag on this error surface. Even though the second and third have different minima locations and scalings, the steepest descent direction will still take you very close to the minimum with the appropriate learning rate. The last case is tricky: even though the shape is an ellipse, the initial weight vector starts off somewhere along the x axis, and so again the steepest descent direction points directly toward the minimum. In other words, there is zero gradient along the vertical axis and therefore we are simply minimizing a parabola along one dimension from that point to get to the minimum.





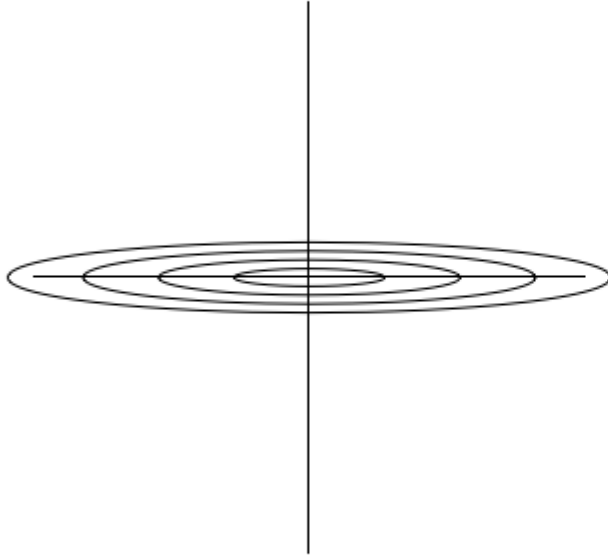
Un-selected is correct



Un-selected is correct



Un-selected is correct



Un-selected is correct

Learning the weights of a logistic output neuron

- Aim: to extend the learning rule of linear output neurons to multi-layer nets of non-linear neurons, it is needed two steps:
 - extend the learning rule to a **single** non-linear neuron (use logistic neurons, besides there are many non-linear neurons instead).
 - central issue: learning rule of multiple layers of features.

Logistic neurons

- Task: Generalize the learning rule of linear neuron to the non-linear (for example, logistic) neurons.
 - Logit z :

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

- Output y :

$$y = \frac{1}{1 + \exp(-z)}$$

- gives a real-valued output that is a smooth and bounded function of the total input.
- having nice derivatives which make learning easy.

Question:

The range of the function $y = \frac{1}{1+e^{-z}}$ is between 0 and 1. Another way of interpreting the logistic unit is that it is modelling:

- ☐ The probability of the inputs given the outputs.
- ☒ The probability of the outputs given the inputs.

Correct

- The derivatives of a logistic neuron:
 - The derivatives of the logit z w.r.t. the inputs and the weights are:

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

$$\frac{\partial z}{\partial w_i} = x_i, \quad \frac{\partial z}{\partial x_i} = w_i$$

- The derivative of the output y w.r.t. the logit z is:

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

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

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

$$\frac{dy}{dz} = \frac{-1(-e^{-z})}{(1 + e^{-z})^2} = \left(\frac{1}{1 + e^{-z}} \right) \left(\frac{e^{-z}}{1 + e^{-z}} \right) = y(1 - y)$$

$$\text{because } \frac{e^{-z}}{1 + e^{-z}} = \frac{(1 + e^{-z}) - 1}{1 + e^{-z}} = \frac{(1 + e^{-z})}{1 + e^{-z}} \frac{-1}{1 + e^{-z}} = 1 - y$$

- Using the chain rule to get the derivatives needed for learning the weights of a logistic unit (because we consider the **single** non-linear neuron)
- To learn the weights, need the derivative of the output y w.r.t. each weight w_i

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

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

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

delta-rule

extra term = slope of logistic

→ the derivative of the output **y** of the single logistic neuron w.r.t. each weight **w_i** is minus the sum of all the row of training cases and of the value on input line **x_iⁿ** times the residual (the difference) between the target and the output (on the actual output of the neuron), with **the extra term** which is come from the slope of the logistic function (that is **yⁿ(1 - yⁿ)**)

→ a slight modification of the delta rule gives the gradient descent learning rule for training a logistic unit.

The backpropagation algorithm

Learning by perturbing (nhiều loạn) weights (not work very well)

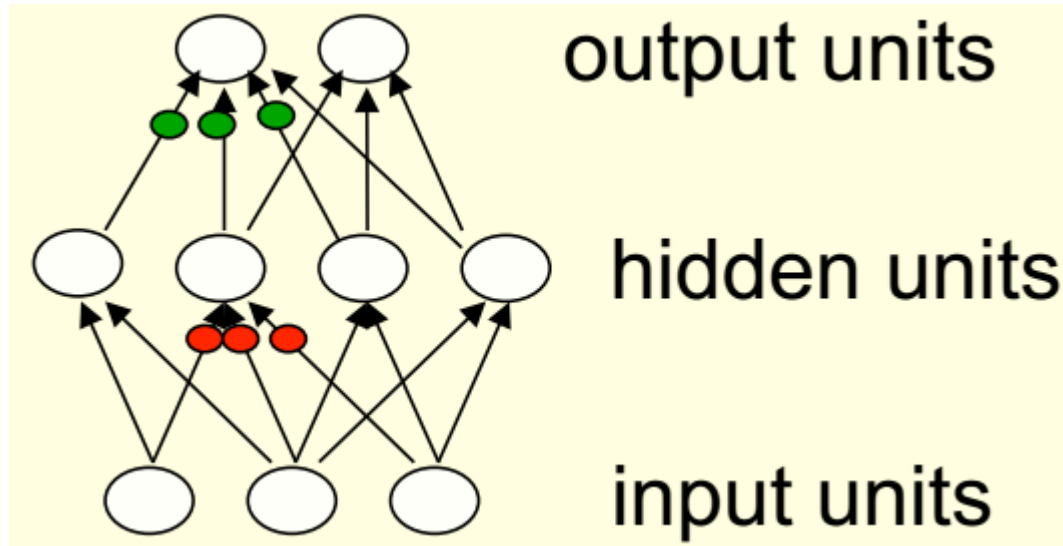
1. Overview

- the idea is based on the evolution or is a form of reinforcement learning (học theo cách cải thiện, tiến triển).
- randomly perturb (nhiều loạn) one weight and see if it improves performance. If so, save the change.
- actions: making the small change and checking whether that pays off. If it does, decide to perform that action.

2. Discuss

- Disad: very inefficient.
→ Reason: To make a change of each weight, need to do multiple forward passes on a **representative set of training cases**. Then, have to see if changing that weight improves things, and **can't** judge that by **one** training case alone.
- Relative to this method of randomly changing weight, and seeing if it helps, the backpropagation is much more efficient, usually more efficient by a factor of the number of weights in the network (could be millions).

- 2nd disad (another problem with randomly changing weights and seeing if it helps) is that: towards the end of learning, any large change in weight perturbations will nearly always make things **worse** (because the weights have to have the right relative values to work properly)
→ towards the end of learning not only do you have to do a lot of work to decide whether each of these changes helps but also the changes themselves have to be very small.



3. Improvement for perturbations of weights to learn

- randomly perturb all the weights **in parallel** and then correlate the performance gain with the weight changes.
→ not better at all, because need to do lots of trials on each training case with different random perturbation of all the weights, in order to “see” the effect of changing one weight through the noise created by all the changing of the other weights.
- another better idea: randomly perturb the **activities** of the hidden units, instead of perturbing the **weight**. Once decided that perturbing the activity of a hidden unit on a particular training case, then can **compute** how to change the weights. Since there are many fewer activities than weights, there’s less things that randomly exploring → the algo makes more efficient, makes better.
- **BUT** those approaches are still much less efficient than **backpropagation**. Backpropagation still wins by a factor of the number of neurons.

Question:

Instead of randomly perturbing the weights randomly and looking at the change in error, one can try the following approach:

1. For each weight parameter w_i , perturb w_i by adding a small (say, 10^{-5}) constant *epsilon* and evaluate the error (call this E_i^+)
2. Now reset w_i back to the original parameter and perturb it again by subtracting the same small constant ϵ and evaluate the error again (call this E_i^-).
3. Repeat this for each weight index i .
4. Upon completing this, we update the weights vector by: $w_i \leftarrow w_i - \eta \frac{(E_i^+ - E_i^-)}{2\epsilon}$

for some learning rate η .

True or false: for an appropriately chosen η , repeating this procedure will find the minimum of the error surface for a linear output neuron.

☒ True

Correct

☒ True

Correct

Another name for this procedure is the finite difference approximation. This procedure approximates the gradient of the error with the respect to the weights (think of the definition of a derivative). In effect, the procedure states that we approximate the gradient and then take a step of the steepest descent method. Although this method works, the backpropagation algorithm finds the exact gradient much more efficiently.

☐ False

The idea behind backpropagation

- Cause: don't know what the hidden units ought to do (hidden units = don't know what their states are), but can compute **how fast** the error changes as changing a hidden activity on a particular training case.
 - instead of using activities of the hidden units as the desired states to train the hidden units, use **the error derivatives w.r.t. hidden activities** $\frac{\partial \text{error}}{\partial \text{hidden activities}}$.
- Since each hidden activity affect many output units, then, therefore, have many separate, different effects on the error and these effects can be combined.

- With backpropagation, can compute the error derivatives for **all** hidden units **at the same time**.
→ once have known the error derivatives w.r.t. hidden activities, it's easier to get the error derivatives w.r.t. weights going into a hidden unit.

Question:

Backpropagation can be used for which kind of neurons ?

☒ Logistic neurons

Correct

Backpropagation works with derivatives. In a binary threshold neuron the derivatives of the output function are zero so the error signal will not be able to propagate through it.

☐ Binary threshold neurons

Un-selected is correct

Sketch of the backpropagation algo on a single case

- Step 1: Write the error equation and compute the error derivative w.r.t. the output

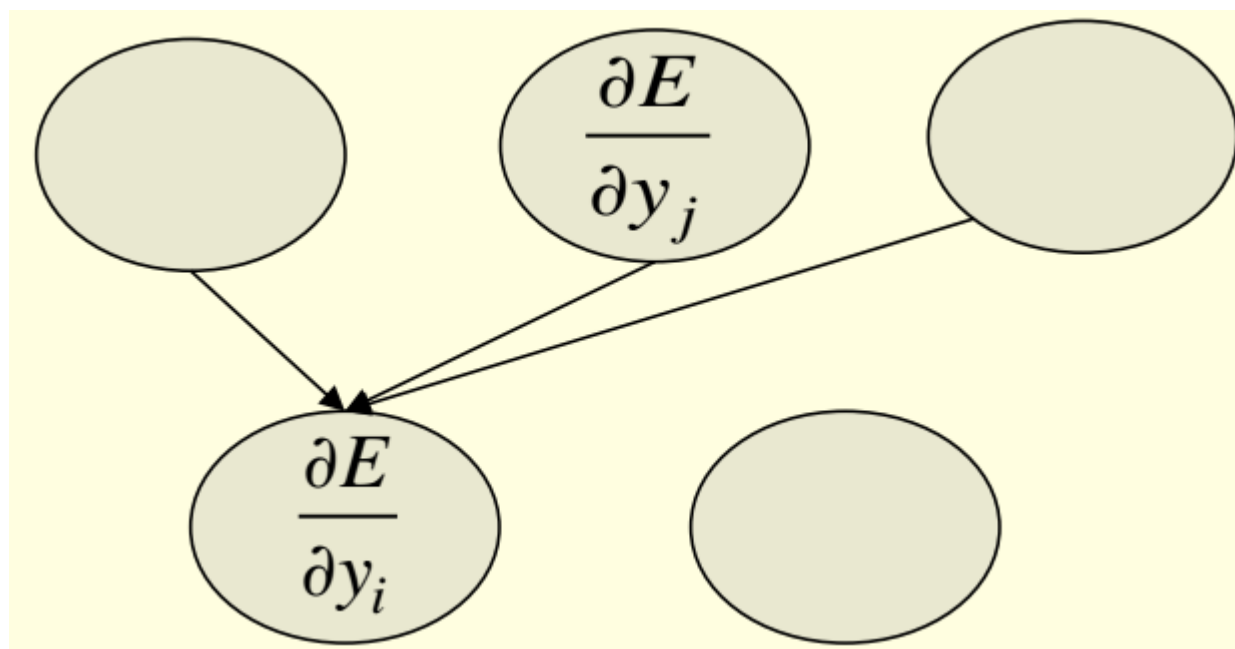
$$E = \frac{1}{2} \sum_{j \in \text{training}} (t_j - y_j)^2$$

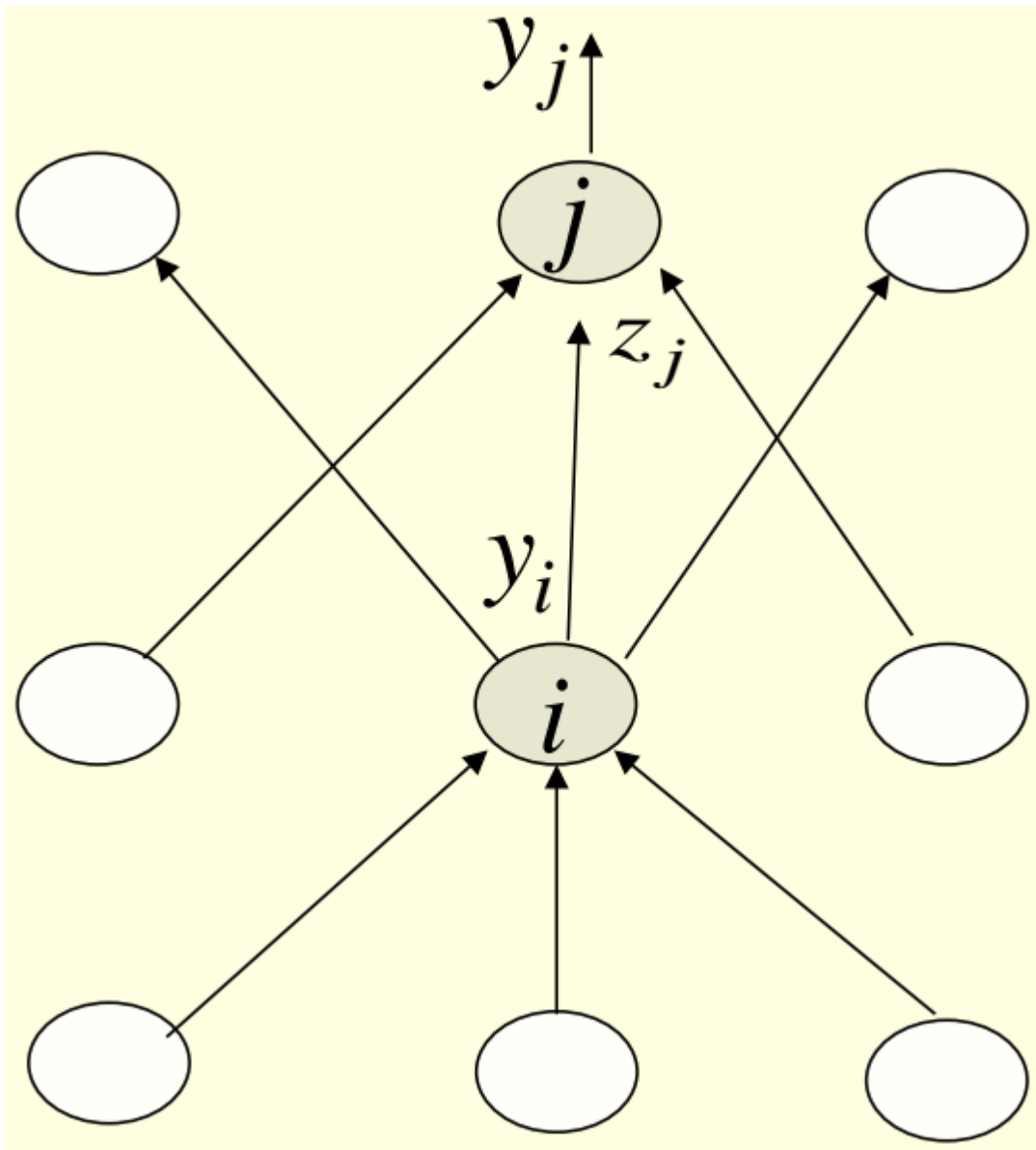
for all training cases ($j \in \text{training}$)

with t_j : the target output, y_j : the actual output for a single training case.

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

- Step 2: Compute error derivatives in each hidden layer from error derivatives in the layer **above**.





For current layer, denote that layer is with index j (above), the previous layer with index i (below).

The y_j is the output of layer j , y_i is the output of layer i , z_j is the logit coming into the layer j

We want to compute the backpropagation $\frac{\partial E}{\partial y}$ for previous layer i

For layer j : The error derivative w.r.t. the **activity** of layer j

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

For layer i : The error derivative w.r.t. the **output** of layer i

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

- Step 3: Use error derivatives w.r.t. **activities** to get error derivatives w.r.t. the **incoming weights**.
 → The error derivative w.r.t. the **weights** coming from layer i to layer j

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

→ From **current** layer, we use the error derivatives w.r.t. **activities** and **output** of this layer in order to compute the error derivatives w.r.t. **output** of the **previous** layer, and then obtain the error derivatives w.r.t the **weights** coming from previous layer to current layer.

How to use the derivatives computed by the backpropagation algo

1. Converting error derivatives into a learning procedure

- The backpropagation algo is an efficient way of computing the error derivative $\frac{dE}{dw}$ for **every** weight on a **single** training case. (it is not learning algo → need to do number of other things to get a proper learning procedure)
- To get a fully specified learning procedure, need to make a lot of other decisions about how to use these error derivatives: Some of these decisions are:
 - Optimization issues: How to use the error derivatives on **individual** cases to discover a good **set** of weights? (**Lecture 6**). (about how going to optimize?)
 - Generalization issues: How to ensure that the **learned weights** work **well** for cases did **not** be **seen** during training? (**Lecture 7**). (how to ensure that the weights that are learned will generalize well)
- Now, a brief overview of these two sets of issues is introduced.

2. Optimization issues in using the weight derivatives (how to use the weight derivatives)

- How **often** to update the weights? (Methods of updating weights)
 - Online: after **each** training case. (compute the error derivatives on a training case using backpropagation and then make a small change to the weights) → zig-zag case because of different error derivatives on different training case, but on average, if make the weight changes small enough, it will go in the right direction.
 - Full batch: after **a full sweep** through the training data. (more sensible to use full batch training where doing a full sweep through all of the training data and adding together all of the error derivatives that is gotten on the individual cases, then making a small step in that direction) (a problem is that if starting off with a bad set of weights and having a very big training set, we don't want to do all that work of going through the whole training case in order to fix up some weights that we know are pretty bad)
 - Mini-batch: after **a small sample** of training cases. (taking small random sample of training cases and going in that direction) (really only need to look at a few training cases before getting a reasonable idea of what direction we want to move the weights in → don't need to look at a large number of training cases until we get towards the end of learning) (a little bit of zig-zagging, not nearly as much zig

zagging as if in online learning where using one training case at a time) (usually used in training big neural networks on big data sets)

- How **much** to update (more in *Lecture 6*) (how big a change we make)
 - Use a fixed learning rate? (just by hand try and pick some fixed learning rate, then learn the weights by changing each weight by the derivative that is computed times that learning rate = $\frac{\partial E}{\partial w} \times \delta$)
 - Adapt the global learning rate? (more sensible to adapt the learning rate = oscillating around, if the error keeps going up and down, then reduce the learning rate, but if we're making steady progress, might increase the learning rate)
 - Adapt the learning rate on each connection separately? (so that some weights learn rapidly and other weights learn more slowly)
 - Don't use steepest descent (hướng có độ dốc dốc nhất)? (if we had a very elongated (giãn dài) ellipse, we see that the direction of the steepest descent is almost at wrong angles to the direction to the minimum that we want to find) (typically, towards the end of learning, of most learning problems, so there's much better directions to go in than the direction of steepest descent)

Question:

When we perform online learning (using steepest descent), we look at each data example, compute the gradient of the error on that case and then take a little step in the direction of the gradient. In offline (also known as batch) learning, we look at each example, compute the gradient, sum these gradients up and then take a (possibly much bigger) step in the direction of the sum of the gradients.

True or false: for one pass over the dataset, these procedures are equivalent in that if we took all of the gradients after each update of the online learning procedure and added them up, then we will get the same gradient as the offline method.

Follow-up question: which method do we expect to be more *stable* with respect to the choice of learning rate? Here we define stable to mean that the learning procedure will converge to a minimum.

Check one box from the left column and one box from the right column.

☐ True

Un-selected is correct

☒ False

Correct

Then intuitive reason why the answer is false is that in online learning we start moving after seeing just one data point. This takes us to a different region of the parameter space where the gradients will now differ. The message is that these methods are fundamentally different.

The key to understanding the second part is to notice that what we really care about (from an optimization standpoint) is minimizing the error with respect to the entire training set. That is, there is some error surface that is the sum of all of the individual error surfaces from each example. When we do offline learning, we directly minimize this total error while in online learning, each step is minimizing a *different* error function that only *approximates* the total error.

☒ Online

This should not be selected

☐ Offline

☒ Online

This should not be selected

☐ Offline

This should be selected

☐ Online

Un-selected is correct

☒ Offline

Correct

Then intuitive reason why the answer is false is that in online learning we start moving after seeing just one data point. This takes us to a different region of the parameter space where the gradients will now differ. The message is that these methods are fundamentally different.

The key to understanding the second part is to notice that what we really care about (from an optimization standpoint) is minimizing the error with respect to the entire training set. That is, there is some error surface that is the sum of all of the individual error surfaces from each example. When we do offline learning, we directly minimize this total error while in online learning, each step is minimizing a *different* error function that only *approximates* the total error.

Explanation:

Offline learning trying to optimize with respect to all entire of training data, so it directly minimize the total error while online learning just is done in each data example, so it minimize a different error that approximately the total error → not stable.

3. Overfitting: The downside of using powerful models

- The training data **contains** information about the regularities in the mapping from input to output. But it also contains **two** types of noise:
 - The target values may be **unreliable** (a minor worry).
 - There is **sampling error** (There will be accidental regularities (những quy tắc do sự cố, do vô tình, do tai nạn, do ngẫu nhiên chủ quan) just because of the particular training cases that were chosen).
- When fitting the model, it cannot tell which regularities (những quy tắc) are real and which are caused by **sampling error**.
 - → fitting both kinds of regularity.
 - If the model is very flexible, it can model the sampling error really well (nếu có một mô hình rất linh hoạt, mô hình này có thể mô hình các lỗi lấy mẫu một cách rất tốt, rất hiệu quả) → this is a disaster (thảm họa).

Question:

For the points shown in the slides, which of the following will help in preventing overfitting?
Check all that apply.

☐ Fit small degree polynomials to the data.

This should be selected

☒ Constrain the coefficients of the polynomial to have small values.

Correct

Polynomials with small degrees and small coefficients help in preventing overfitting because they are "simpler". Getting more data will reduce uncertainty and cover unobserved parts of the input space.

☒ Get more data points.

Correct

Polynomials with small degrees and small coefficients help in preventing overfitting

Correct

Polynomials with small degrees and small coefficients help in preventing overfitting because they are "simpler". Getting more data will reduce uncertainty and cover unobserved parts of the input space.

☒ Get more data points.

Correct

Polynomials with small degrees and small coefficients help in preventing overfitting because they are "simpler". Getting more data will reduce uncertainty and cover unobserved parts of the input space.

☐ Fit the data with a polynomial of degree greater than the number of training points.

Un-selected is correct

For the points shown in the slides, which of the following will help in preventing overfitting?
Check all that apply.

☒ Fit small degree polynomials to the data.

Correct

Polynomials with small degrees and small coefficients help in preventing overfitting because they are "simpler". Getting more data will reduce uncertainty and cover unobserved parts of the input space.

☒ Constrain the coefficients of the polynomial to have small values.

Correct

Polynomials with small degrees and small coefficients help in preventing overfitting because they are "simpler". Getting more data will reduce uncertainty and cover unobserved parts of the input space.

☒ Get more data points.

- Way to reduce overfitting: (How to prevent the network from overfitting very badly if use the large network)
 - weight-decay (try and keep the weights of the networks small, try and keep many of the weights at 0, idea: make the model simpler)
 - weight-sharing (make the model simpler by insisting that many of the weights have exactly the same value as each other, *next lecture*, how weight-sharing is used)
 - early stopping (peek at a fake test set while training and stop when performance gets decent)
 - model averaging (train lots of different neural nets, and average them together in the hopes that will reduce the errors)
 - Bayesian fitting of neural nets (just a fancy form of model averaging)
 - dropout (try and make the model more robust by randomly emitting hidden units when training it)
 - generative pre-training (more complicated) (*end of the course*)
 - more detail in *Lecture 7*

Week 4: Learning feature vectors for words

4.1. Learning to predict the next word

