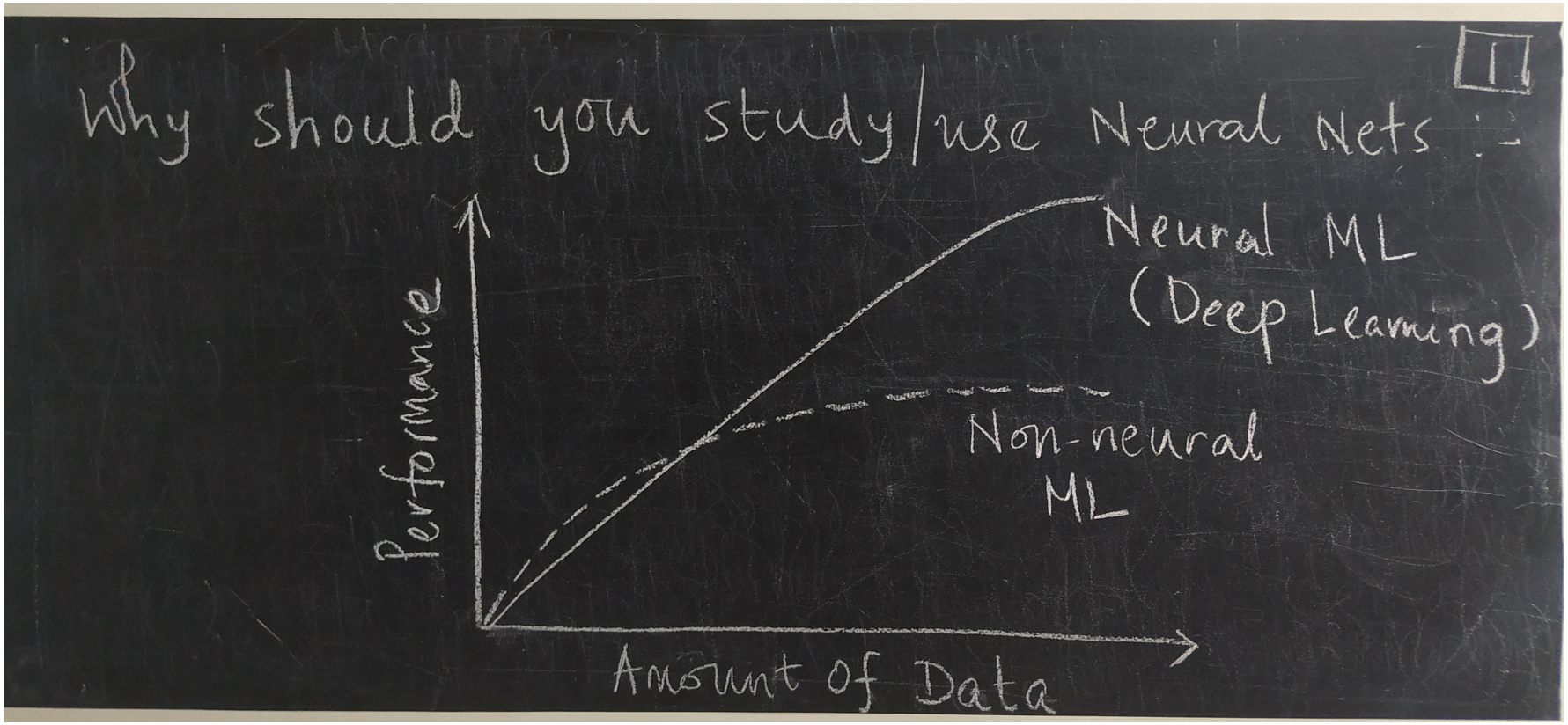
Neural Networks

Machine Learning (BITS F464)

Why study NNs?



In this series:

- NNs from a Bayesian Perspective
- Logistic Regression as a NN (Perceptron*)
- Issues with perceptron
- Kernel trick*
- Layered NNs (Multilayer Perceptron)
  - Backpropagation – Derivation
- Activation functions

(*Later, we will link these with Support Vector Machines)

NNs from a Bayesian Perspective I

- Model has a structure (π) and parameters ($\mathbf{w}$).
- We represent our prior beliefs on what the model parameters are. This results in a prior distribution over model's parameters: $p(\mathbf{w})$.
- As we collect more data $\mathbf{D} = (\mathbf{X}, \mathbf{Y})$, we upate the prior distribution and turn it into a posterior distribution:

$$p(\mathbf{w}|\mathbf{D}) = \frac{p(\mathbf{D}|\mathbf{w})p(\mathbf{w})}{p(\mathbf{D})}$$

or,

$$p(\mathbf{w}|\mathbf{X}, \mathbf{Y}) = \frac{p(\mathbf{Y}|\mathbf{X}, \mathbf{w})p(\mathbf{w})}{p(\mathbf{Y}|\mathbf{X})}$$

The first term in the numerator is called likelihood and it represents how likely the data is, given model's parameters $\mathbf{w}$.

NNs from a Bayesian Perspective II

- A neural net's goal is to estimate the likelihood $p(\mathbf{Y}|\mathbf{X}, \mathbf{w})$. (Recall that: you were maximising the likelihood by minimizing the MSE for linear models)
- To find the best model weights are the Maximum Likelihood Estimates (MLE) of the weights:

$$\begin{aligned}\mathbf{w}^{MLE} &= \underset{\mathbf{w}}{\text{argmax}} \ \log p(\mathbf{D}|\mathbf{w}) \\ &= \underset{\mathbf{w}}{\text{argmax}} \ \sum_i \log p(y_i|\mathbf{x}_i, \mathbf{w})\end{aligned}$$

NNs from a Bayesian Perspective III

- Alternatively, we can use our prior knowledge, represented as prior distribution over the weight parameters and maximise the *posterior* distribution. This is called the Maximum Aposteriori Estimates (MAP) of $\mathbf{w}$:

$$\begin{aligned}\mathbf{w}^{MAP} &= \underset{\mathbf{w}}{\text{argmax}} \ \log p(\mathbf{w}|\mathbf{D}) \\ &= \underset{\mathbf{w}}{\text{argmax}} \ \log p(\mathbf{D}|\mathbf{w}) + \log p(\mathbf{w}) \\ &= \underset{\mathbf{w}}{\text{argmax}} \ \sum_i \log p(y_i|\mathbf{x}_i, \mathbf{w}) + \log p(\mathbf{w})\end{aligned}$$

- The term $\log p(\mathbf{w})$ acts as a regularization term.
- Choosing a Gaussian distribution with mean 0 as the prior, we get the mathematical equivalence of L_2 regularisation.

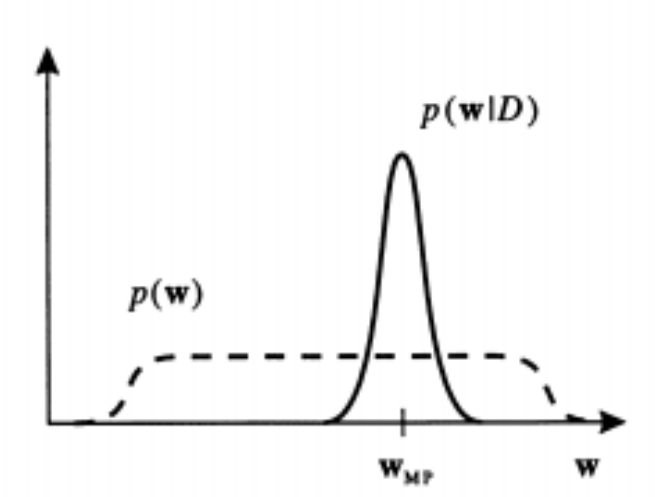
NNs from a Bayesian Perspective IV

Summary:

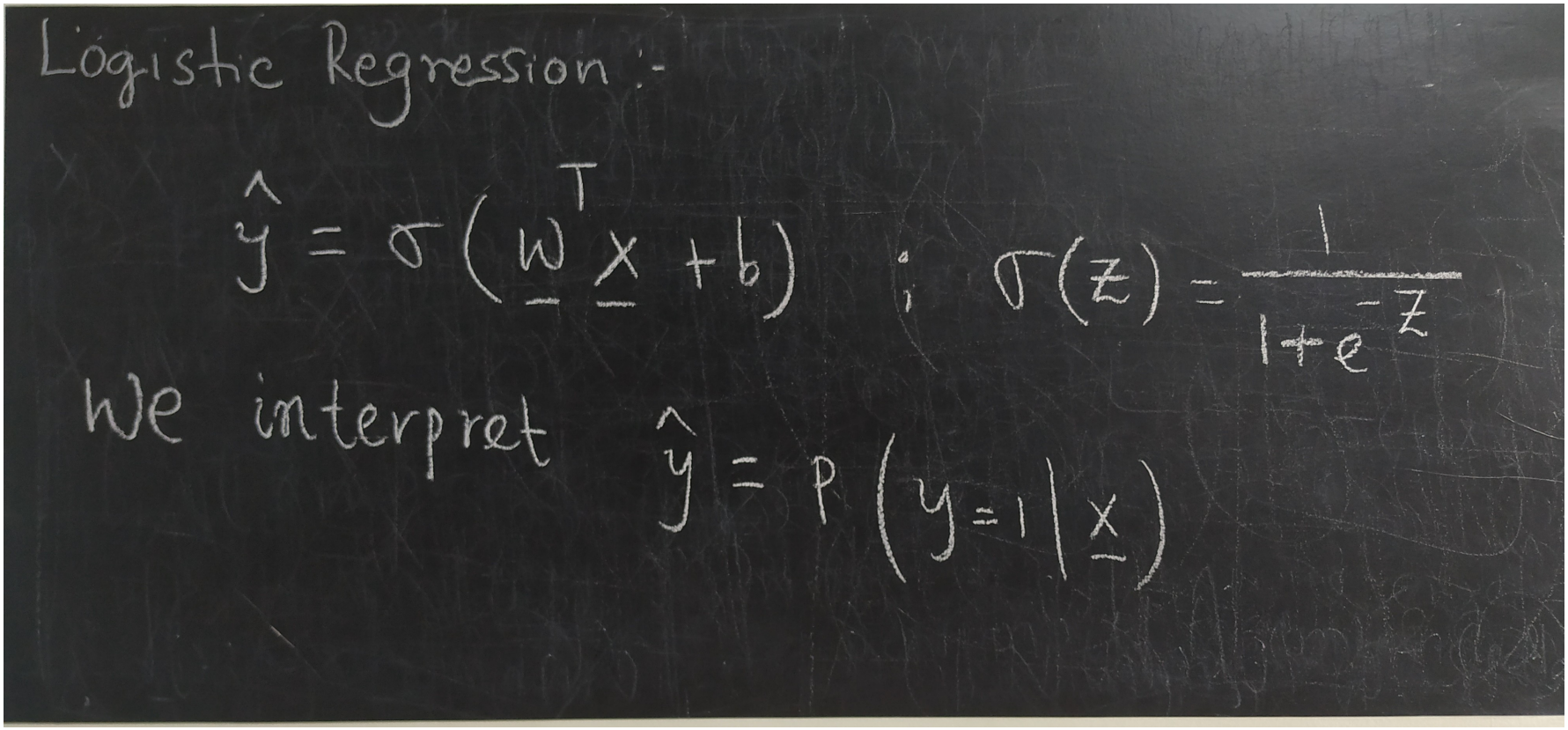
- We intend to find a posterior distribution over weights $\mathbf{w}$

$$p(\mathbf{w}|\mathbf{D}) = \frac{p(\mathbf{D}|\mathbf{w})p(\mathbf{w})}{p(\mathbf{D})} = \frac{p(\mathbf{D}|\mathbf{w})p(\mathbf{w})}{\int p(\mathbf{D}|\mathbf{w})p(\mathbf{w})d\mathbf{w}}$$

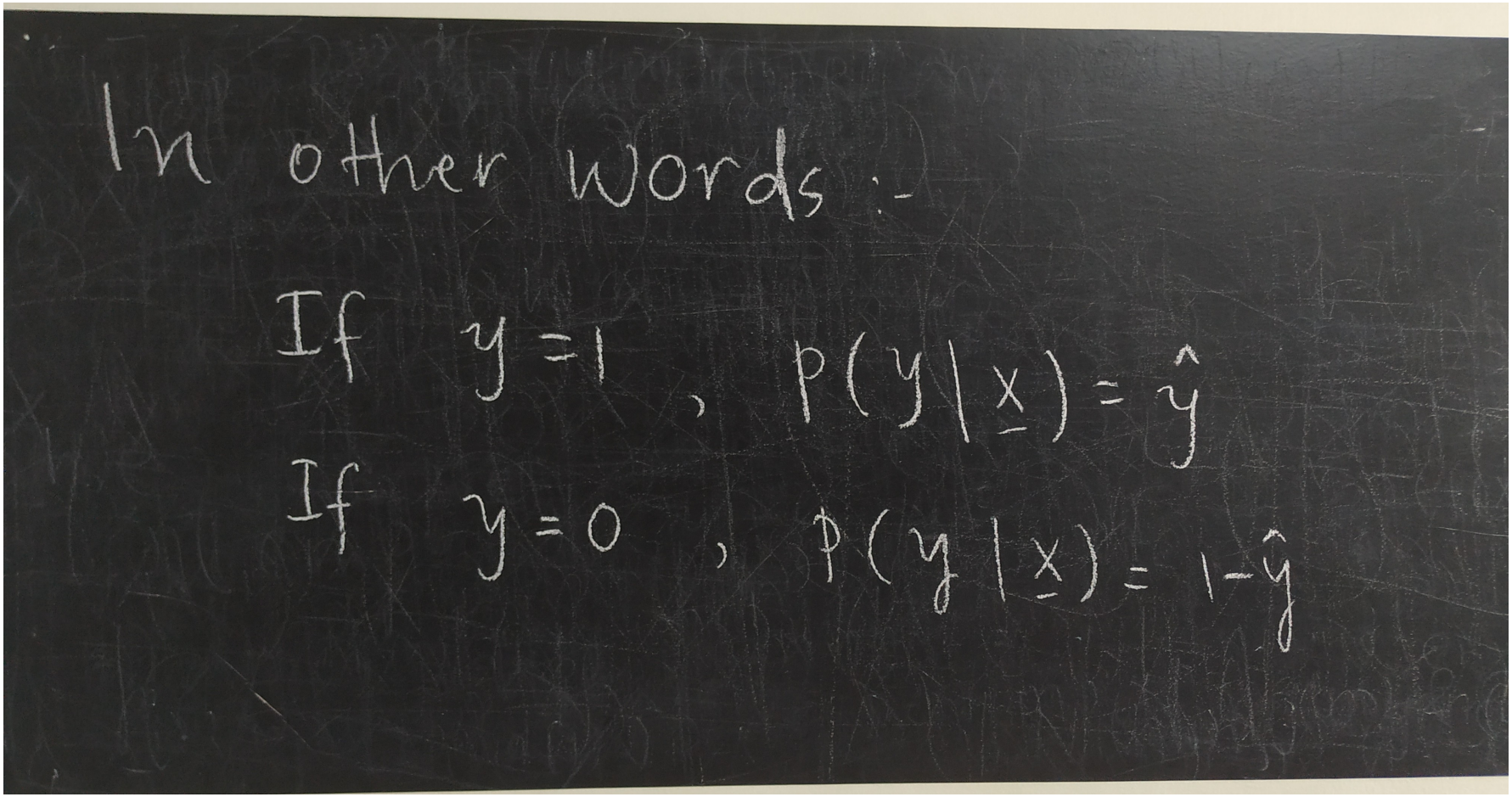
- Learning the weights means changing our belief about the weights from prior $p(\mathbf{w})$ to posterior $p(\mathbf{w}|\mathbf{D})$ as a consequence of seeing data $\mathbf{D}$:



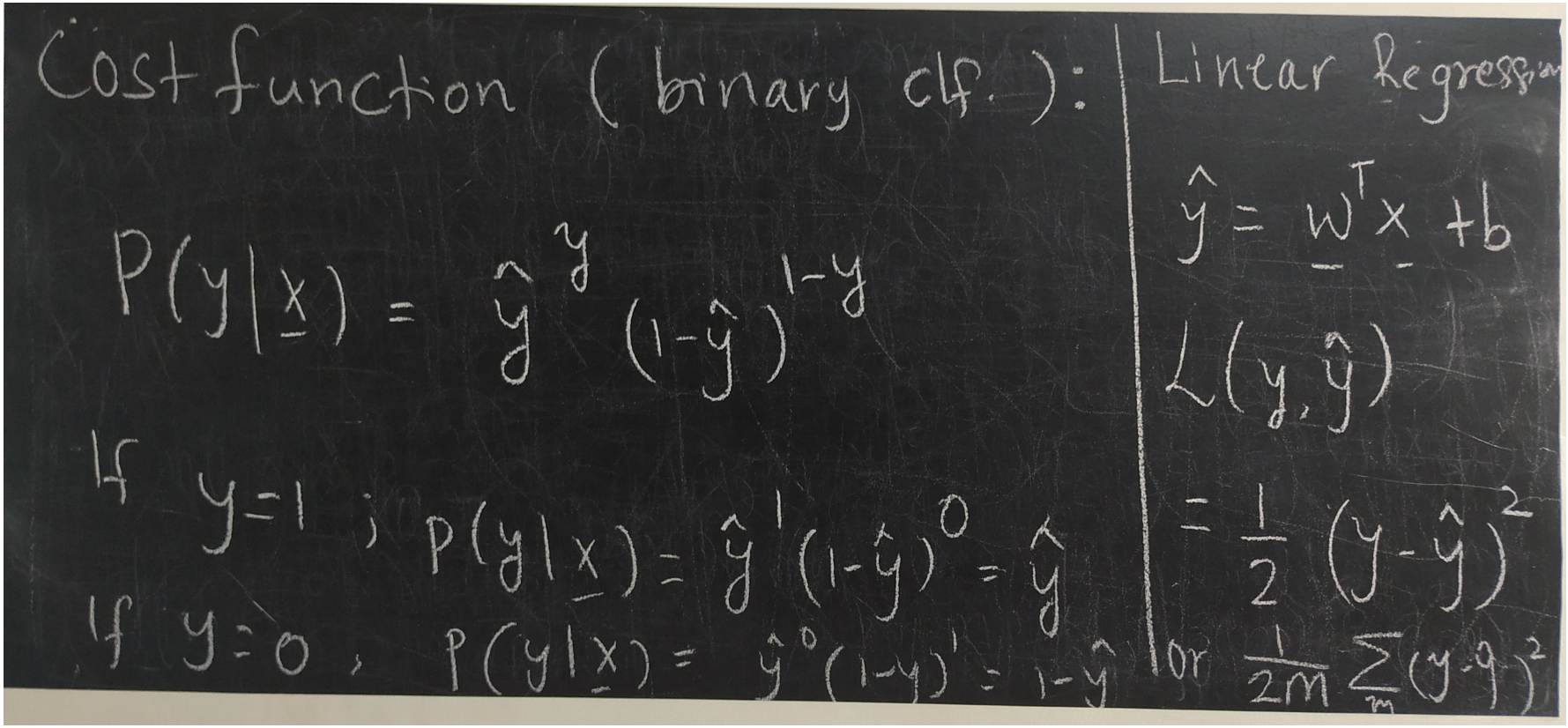
Logistic Regression as NN I



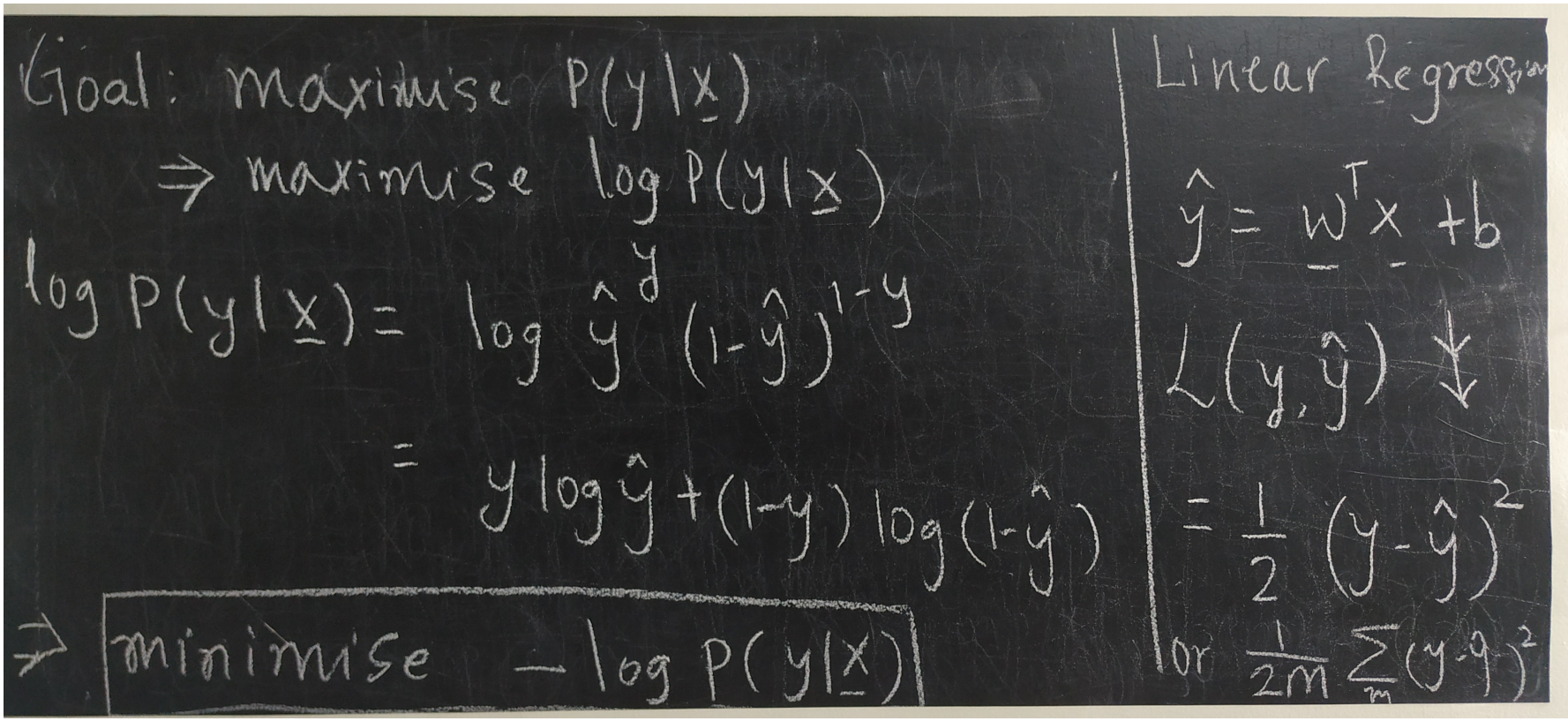
Logistic Regression as NN II



Logistic Regression as NN III



Logistic Regression as NN IV

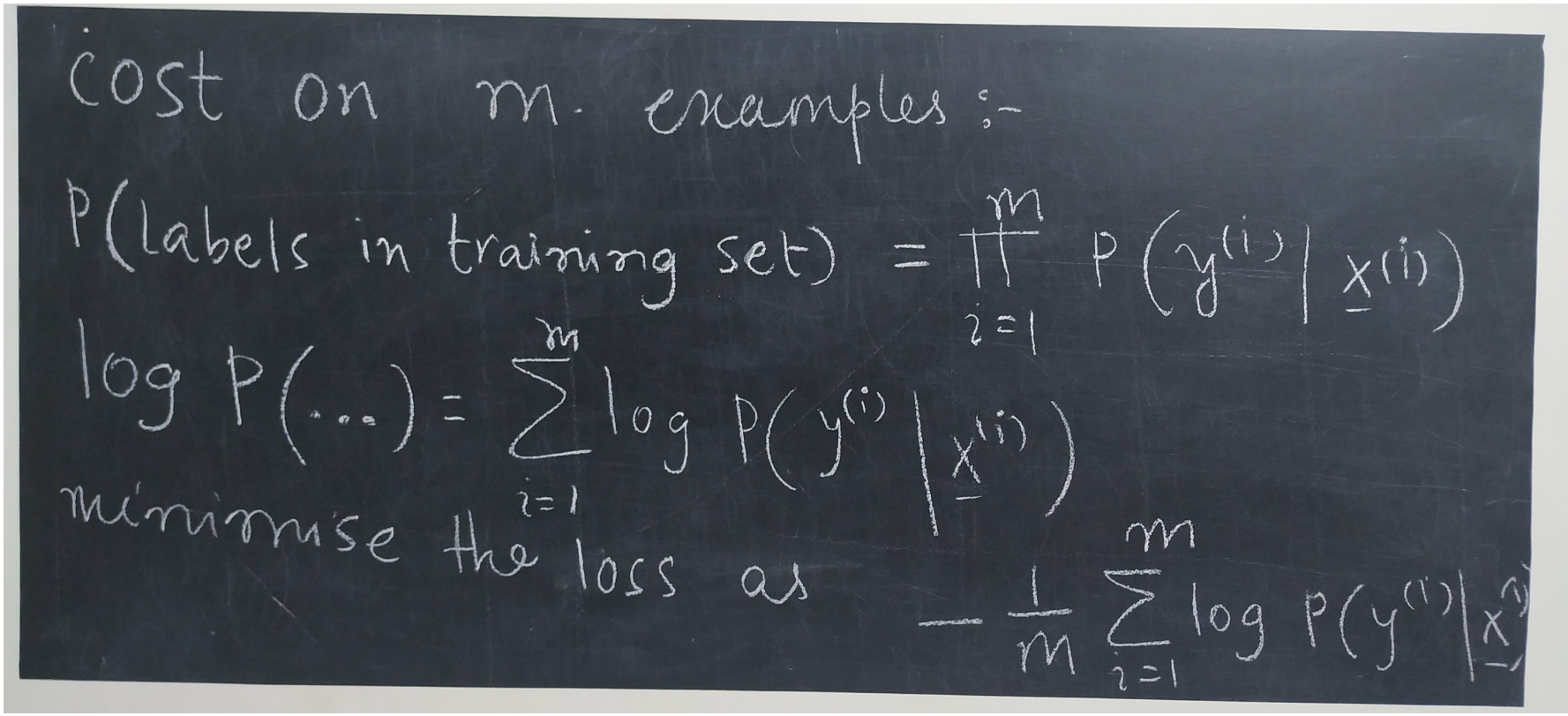


The cost function: $-\{y \log(\hat{y}) + (1-y)\log(1-\hat{y})\}$ is known as **cross-entropy loss function**.

Logistic Regression as NN V

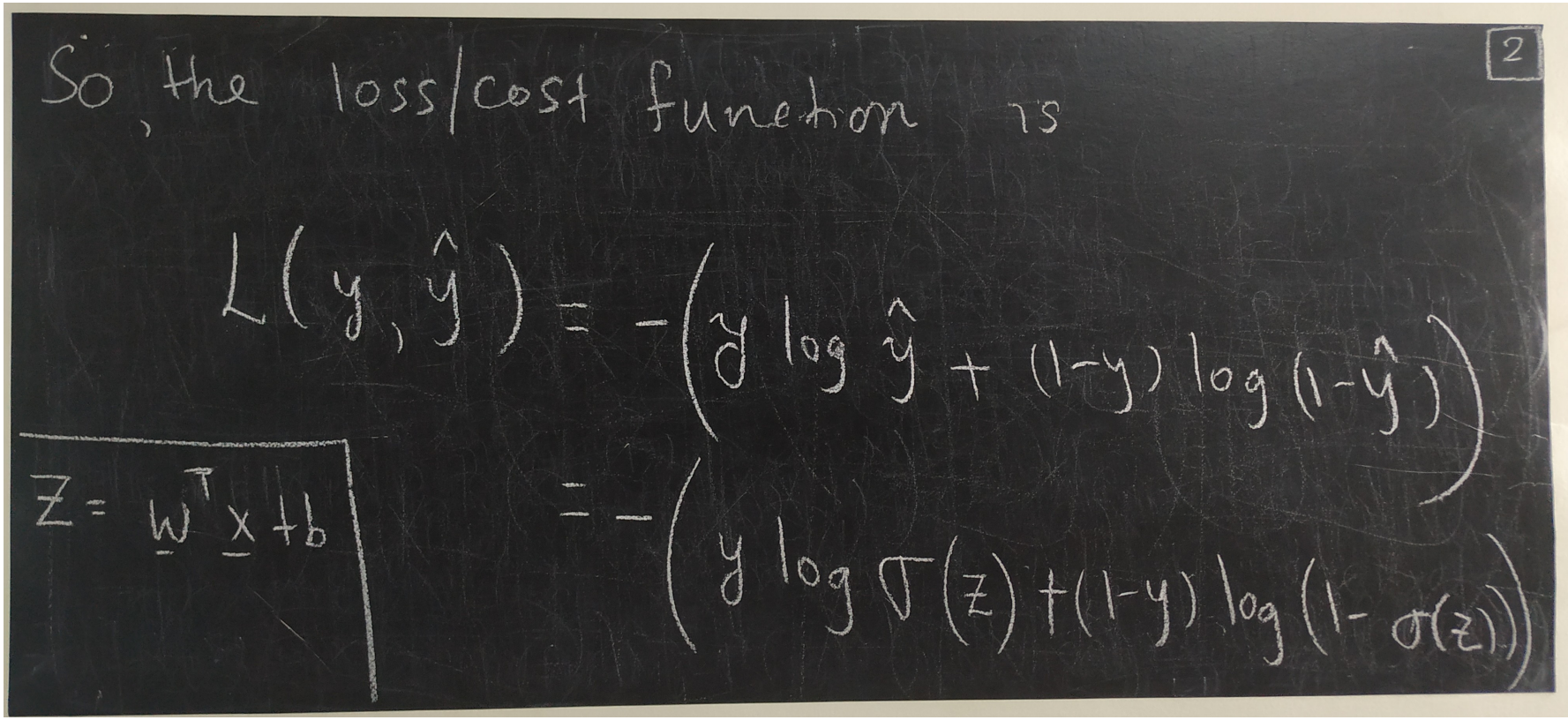
- Cross-entropy function computes the entropy between two different probability distribution.
- As $\hat{y} \in [0, 1]$, we can see that $(\hat{y}, 1 - \hat{y})$ is the probability distribution obtained from the logistic model, given any example $\mathbf{x}$ or $\underline{x}$.
- *Where is the other probabibility distribution?* We can treat the labels as a probability distribution. Any example $\mathbf{x}$ is associated with a label 1 or 0 (1 or 2, or whatever). When we convert these to one-hot encoded vector we get the labelling class 1: [1,0], class 2:[0,1]. This itself is a probability distribution. Therefore, each example is associated with a true probability distribution that the logistic model tries to reach.

Logistic Regression as NN VI



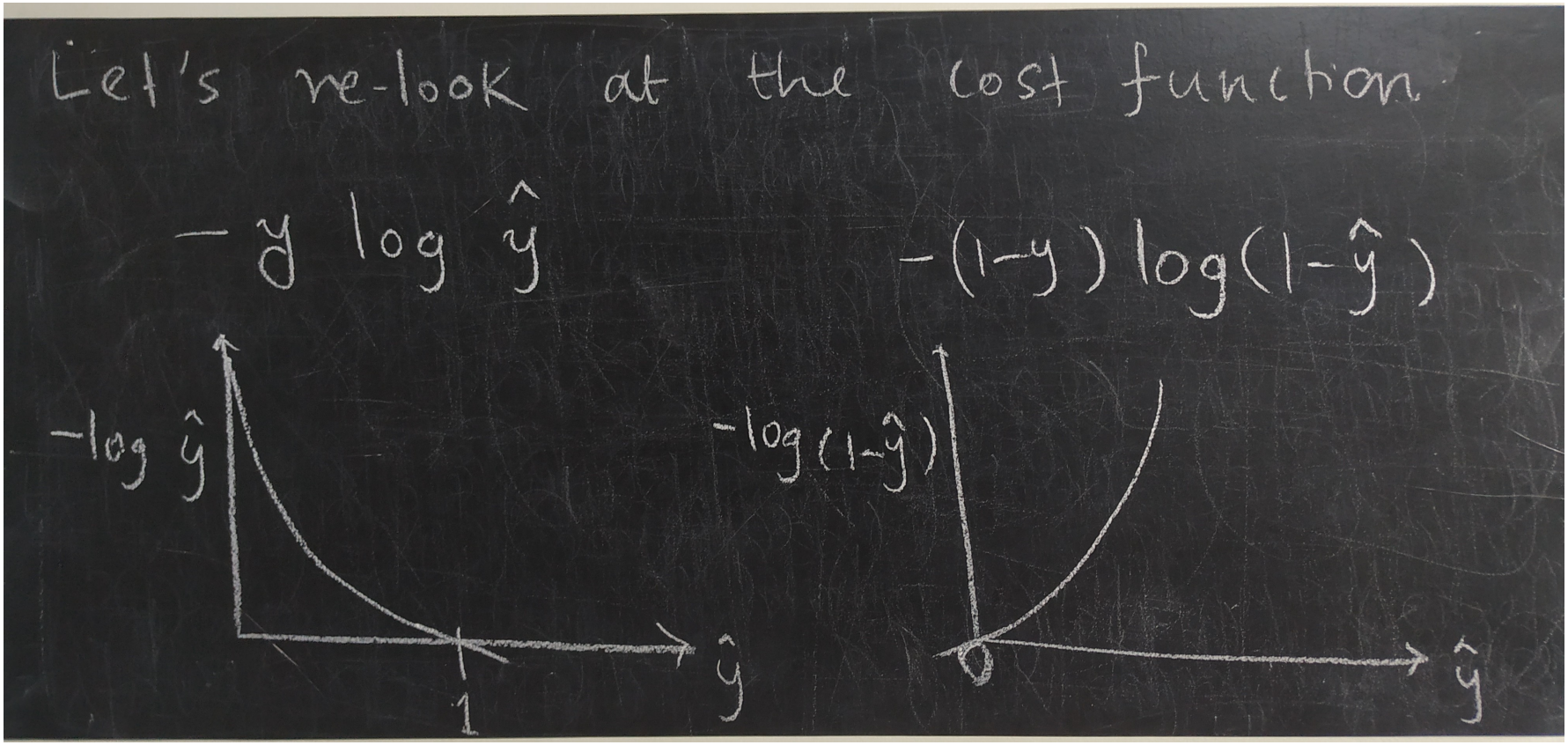
Logistic Regression as NN VII

For convenience, let's stay with the loss for a single example:

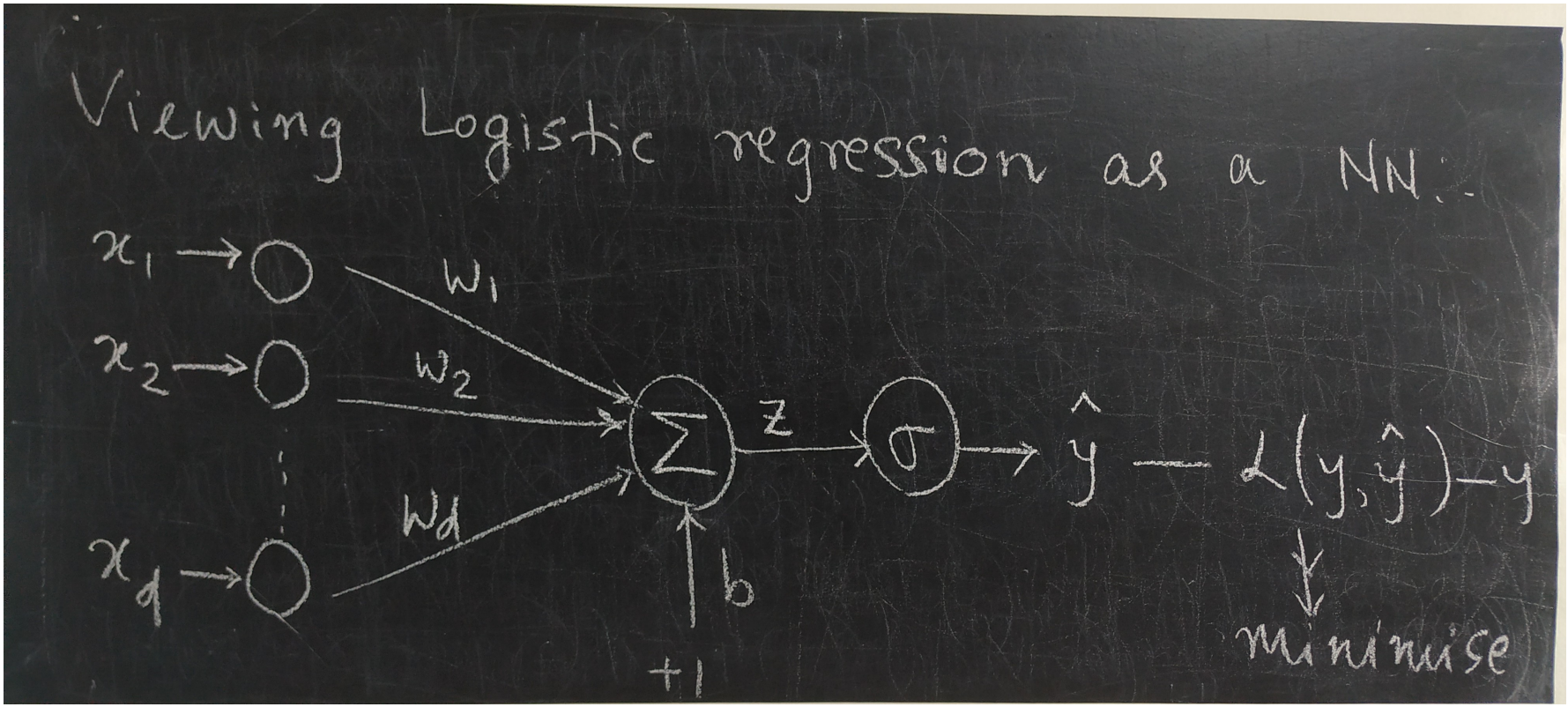


Logistic Regression as NN VIII

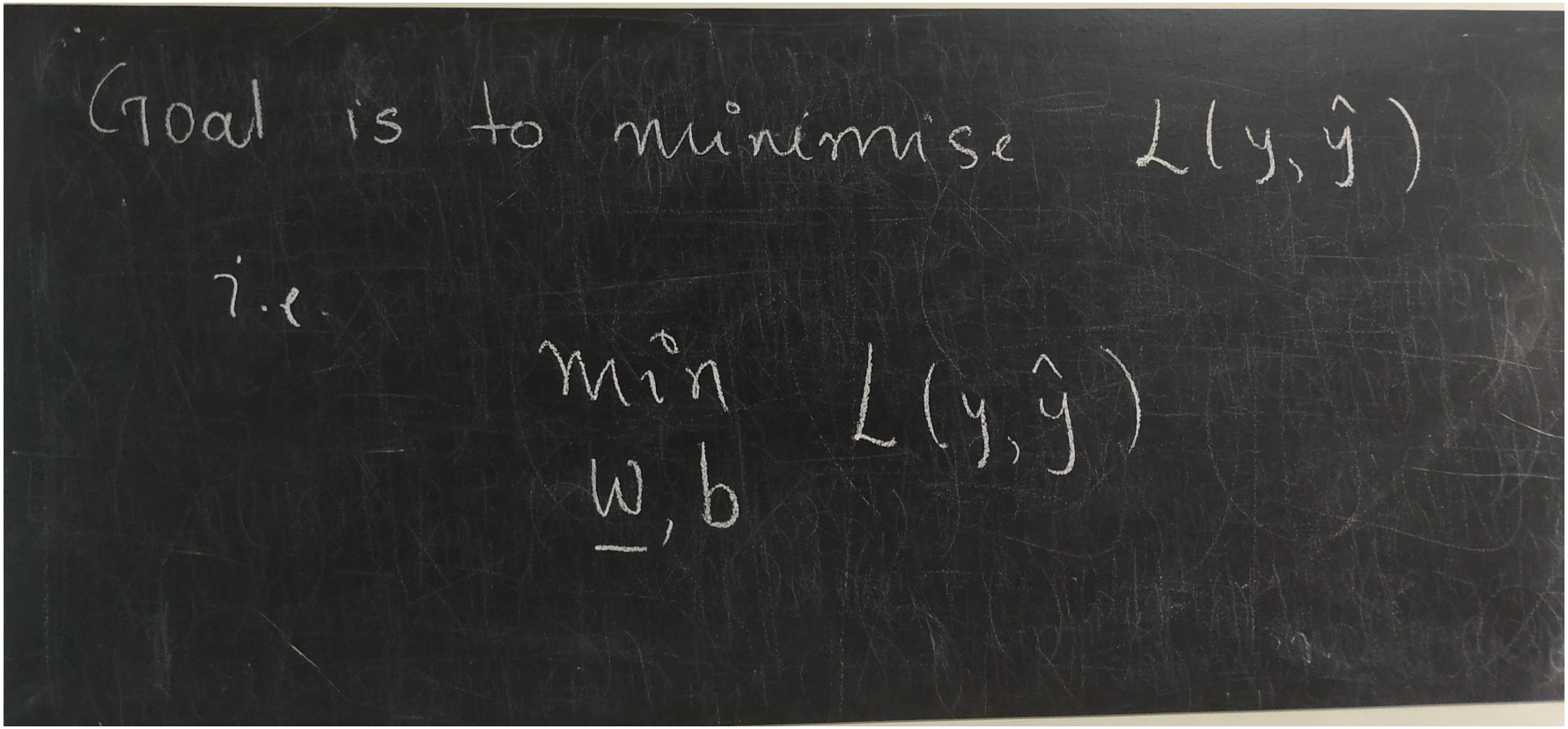
What do these two terms in the loss try to achieve?



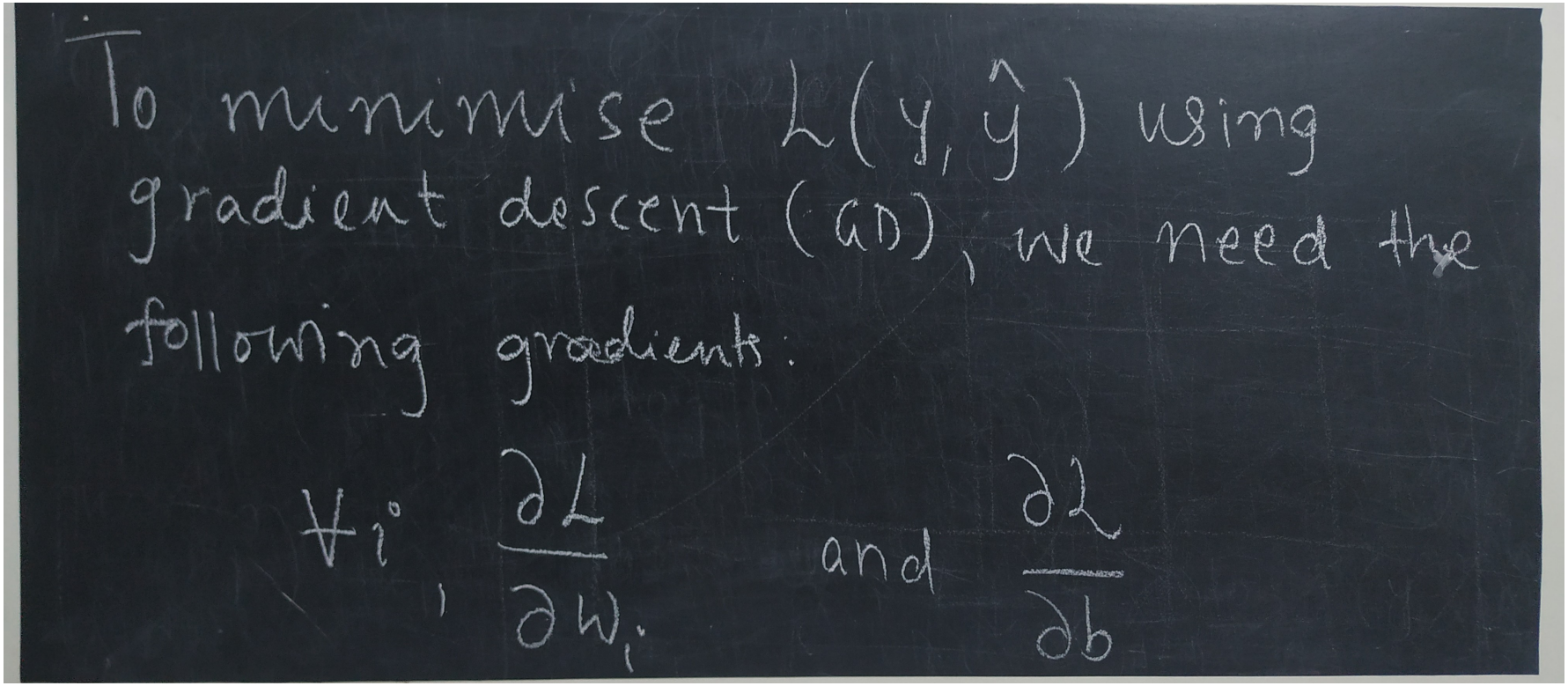
Logistic Regression as NN IX



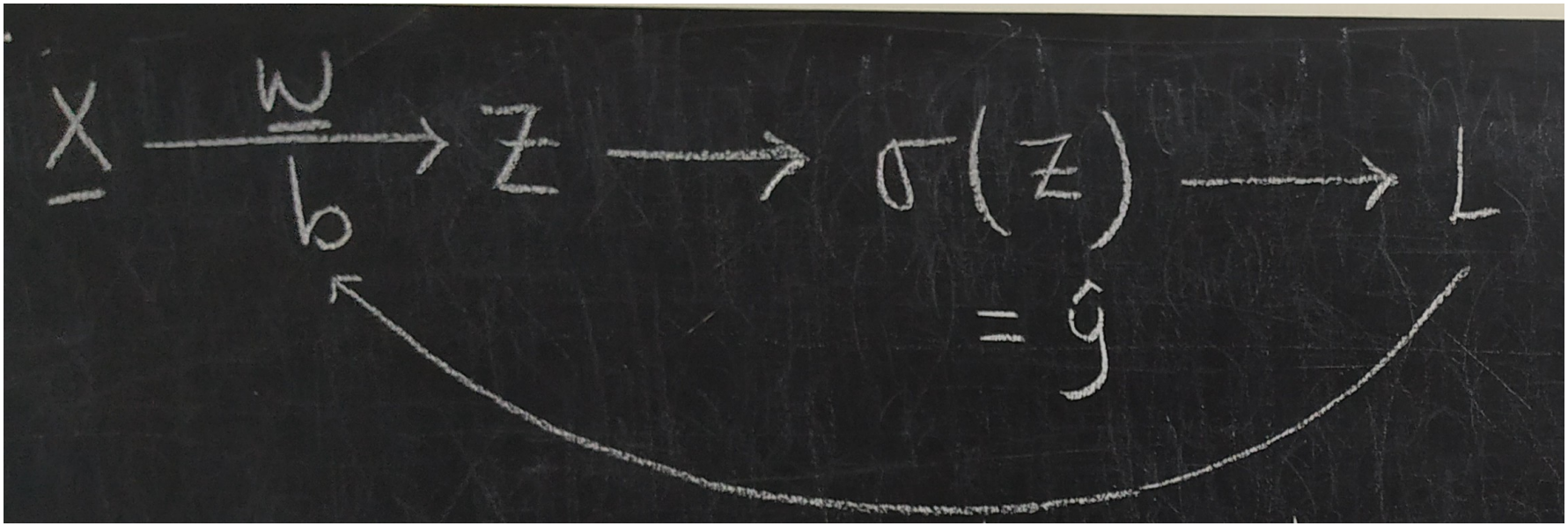
Logistic Regression as NN X



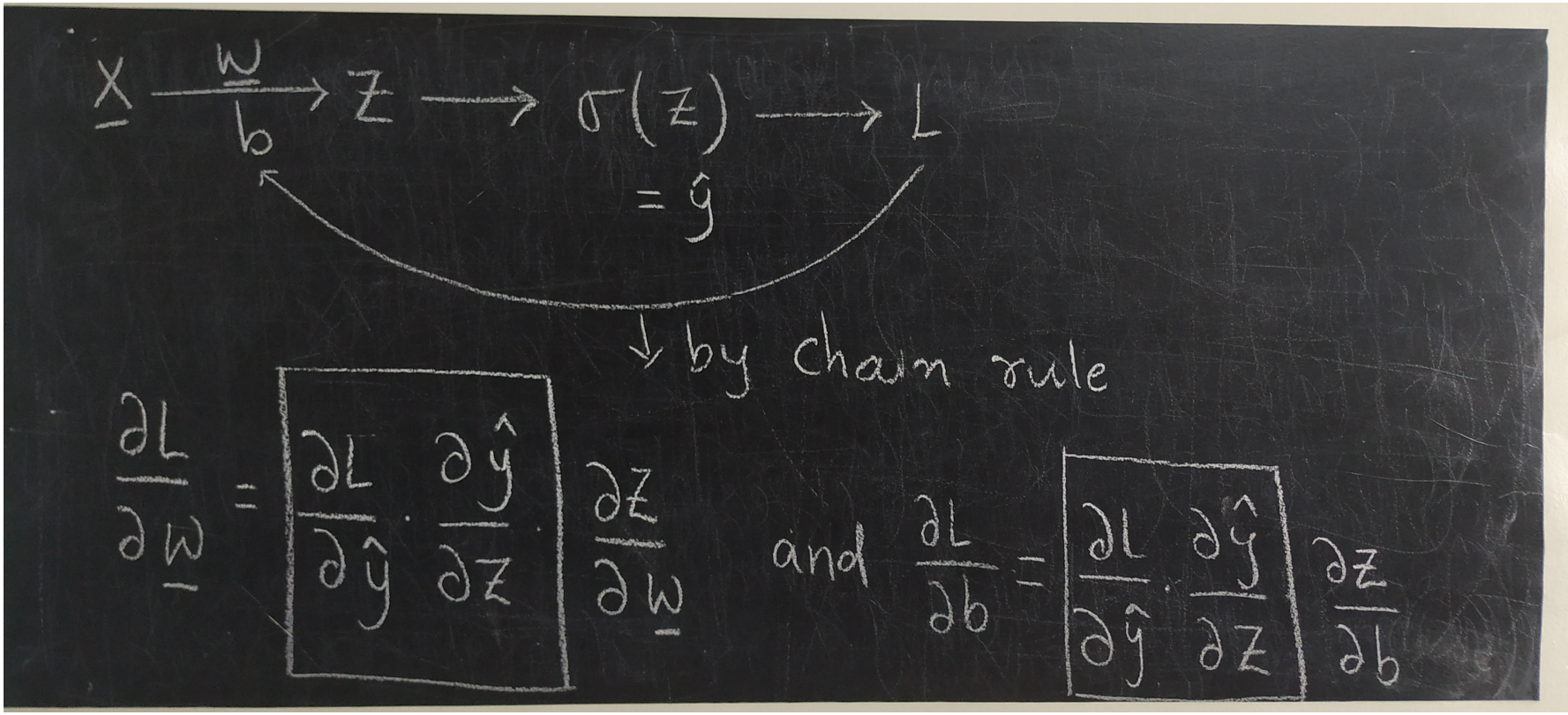
Logistic Regression as NN XI



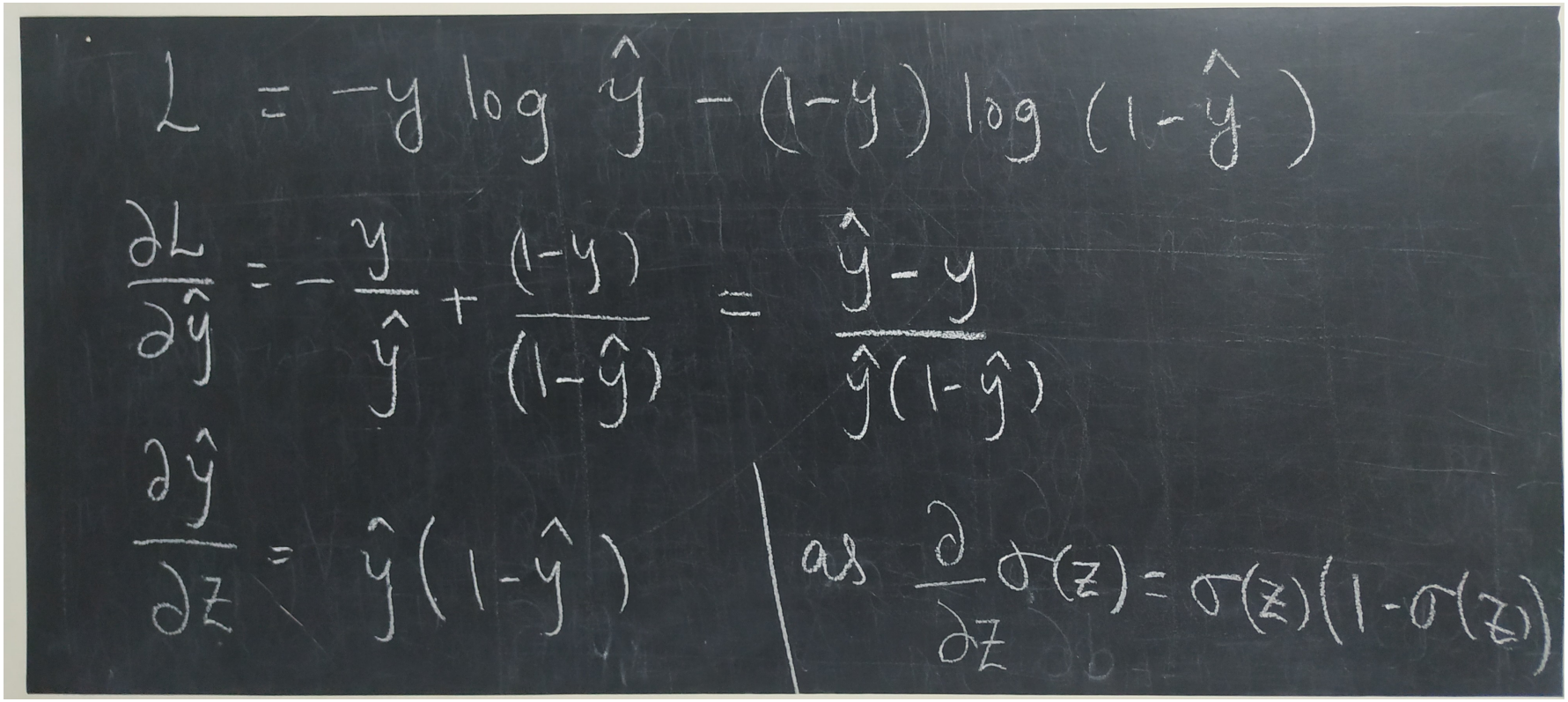
Logistic Regression as NN XII



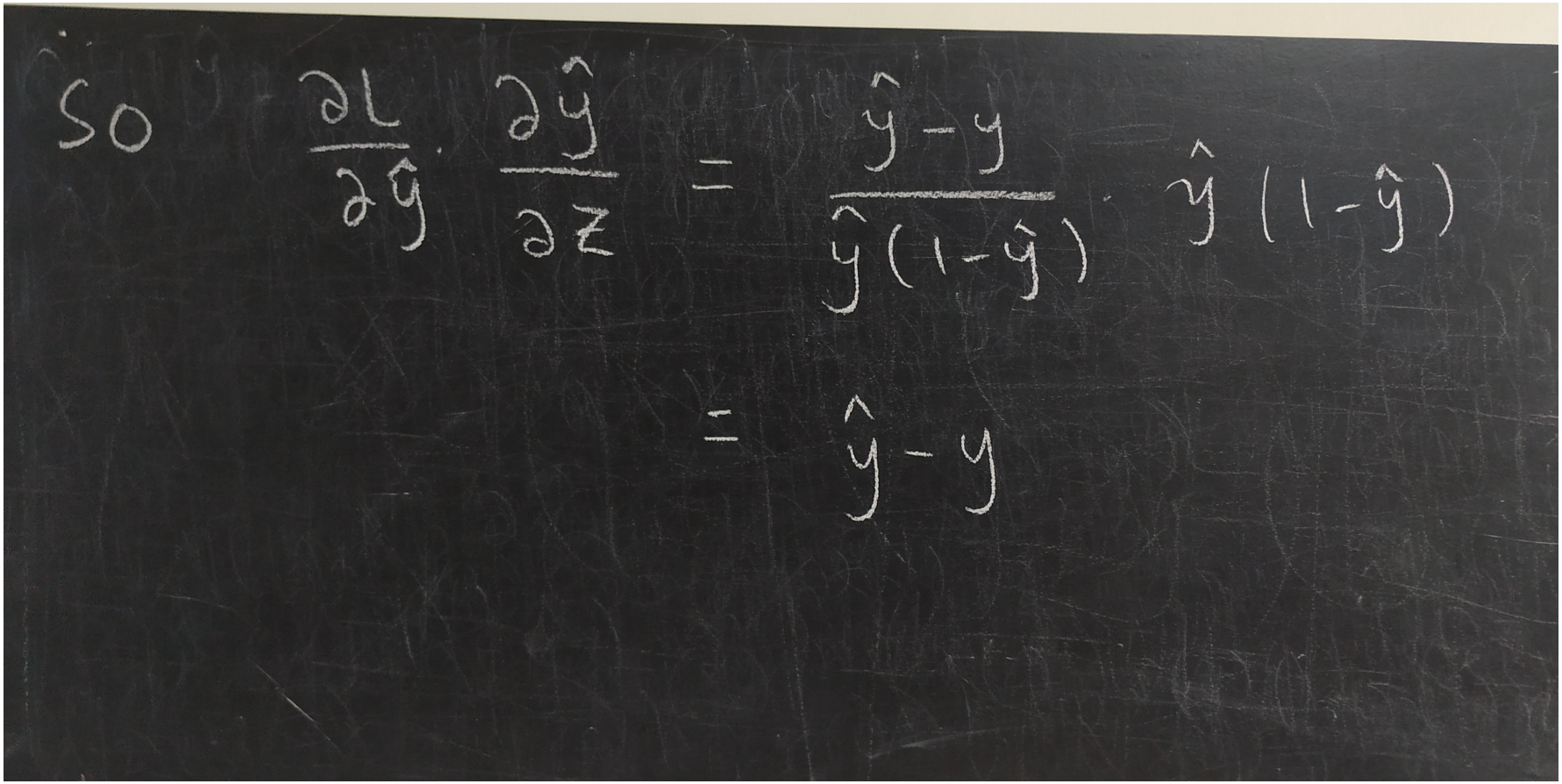
Logistic Regression as NN XIII



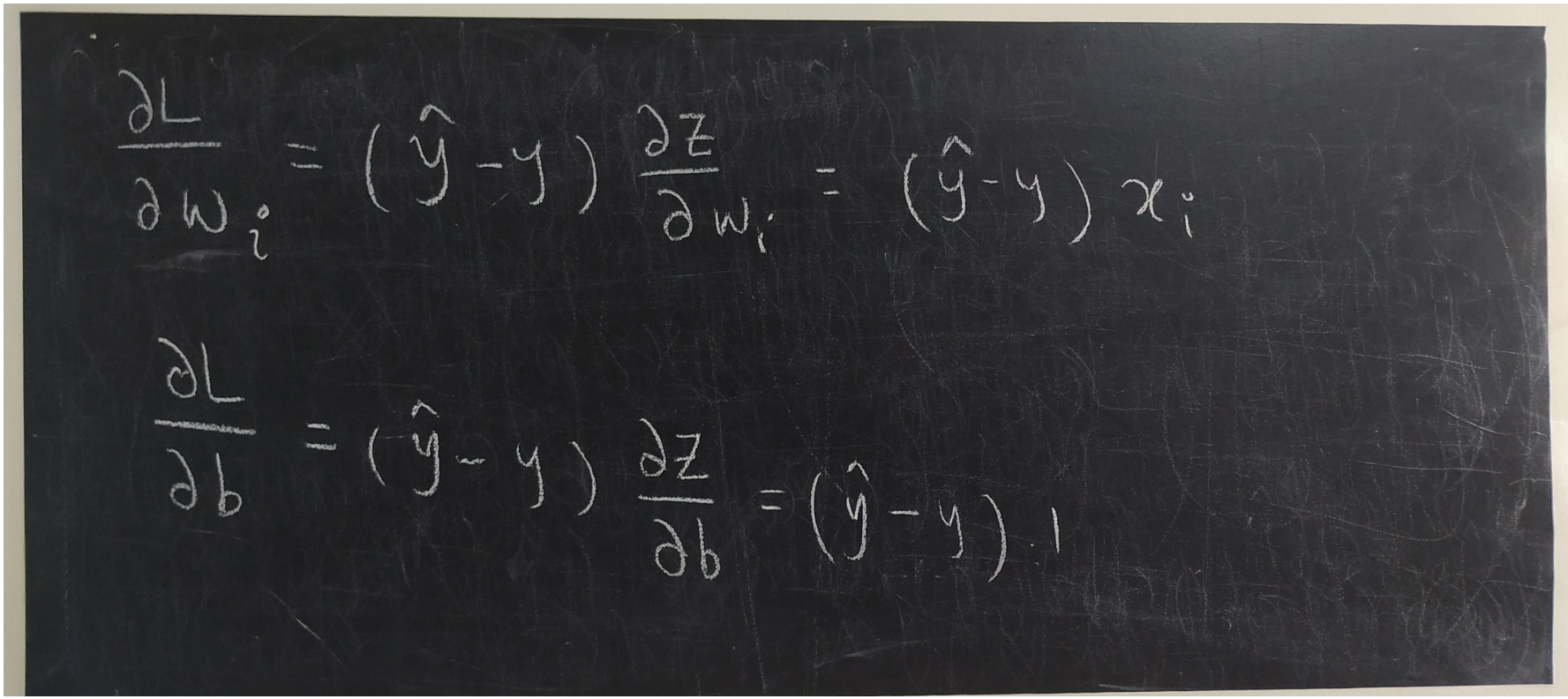
Logistic Regression as NN XIV



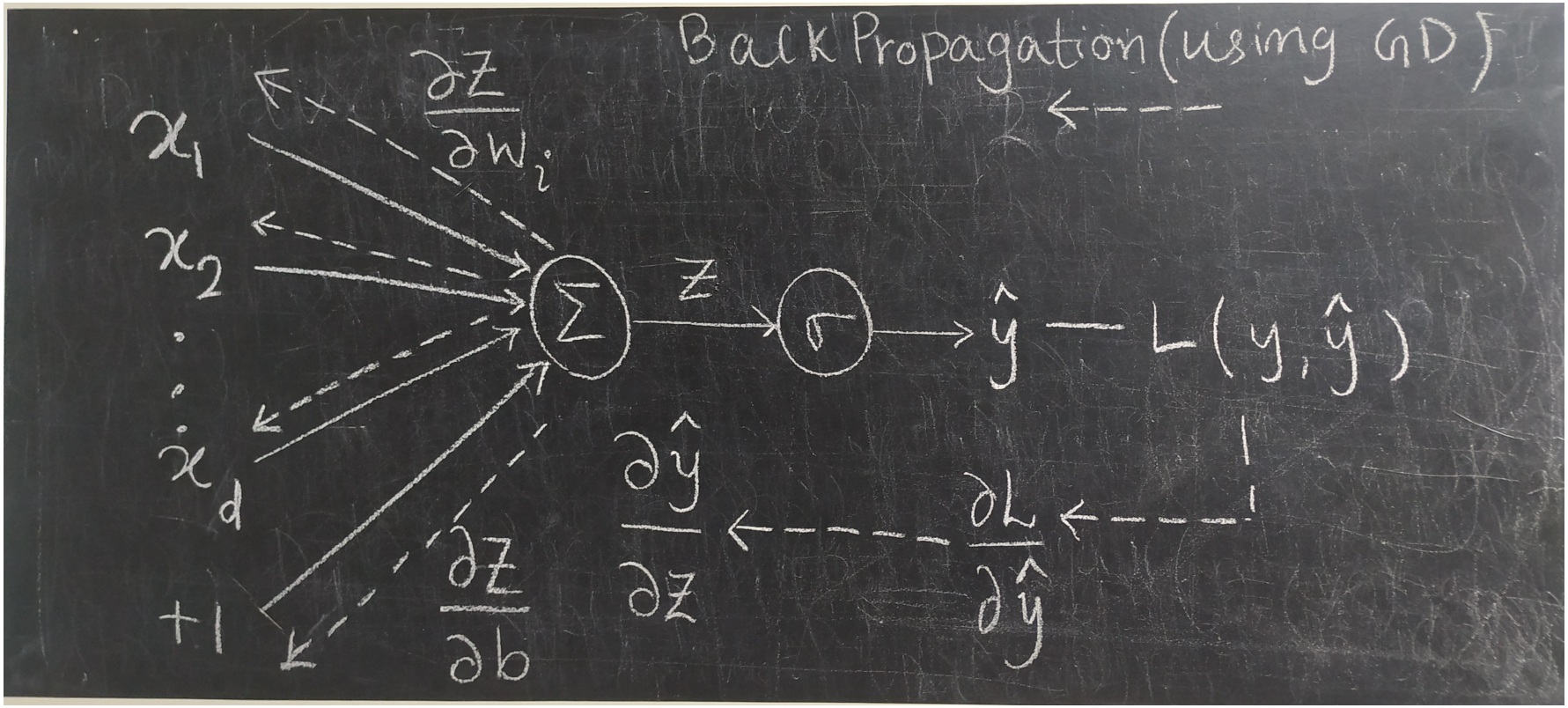
Logistic Regression as NN XV



Logistic Regression as NN XVI



Logistic Regression as NN XVII



Logistic Regression as NN XVIII

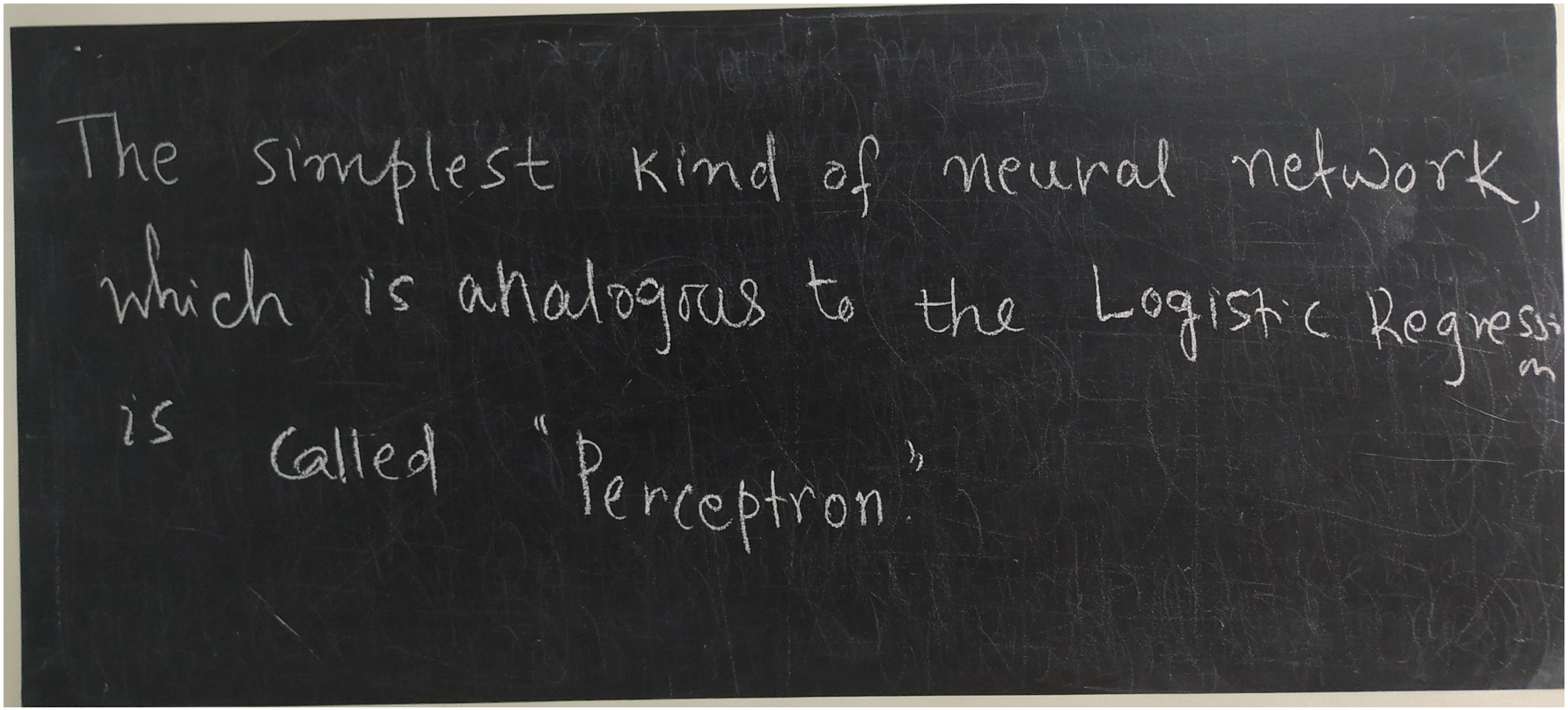
To update the parameters:

$$w_i = w_i - \alpha \frac{\partial \mathcal{L}}{\partial w_i}, \ \forall i \in \{1, \ldots, d\}$$

$$b = b - \alpha \frac{\partial \mathcal{L}}{\partial b}$$

where, $\alpha \in (0, 1]$ is the learning rate.

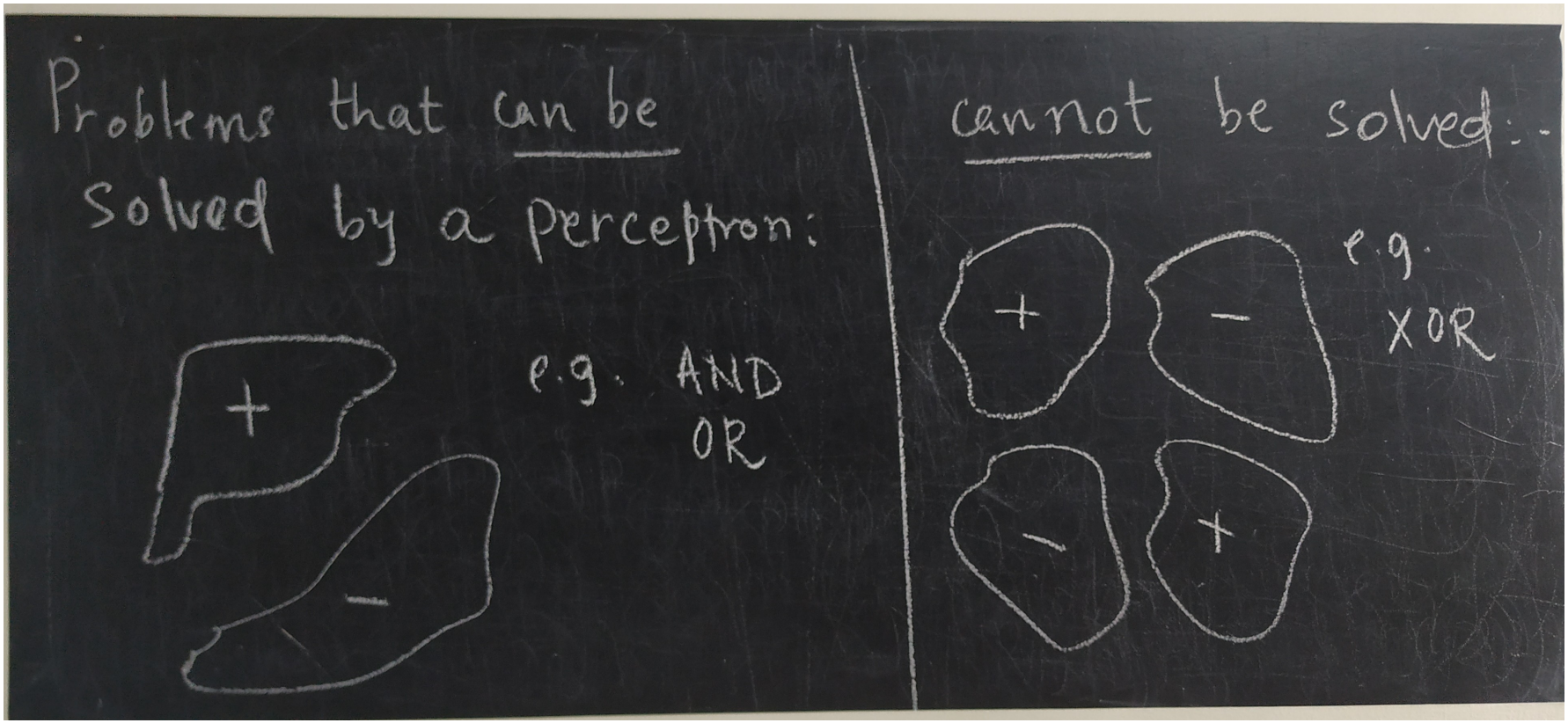
Logistic Regression as NN XIX



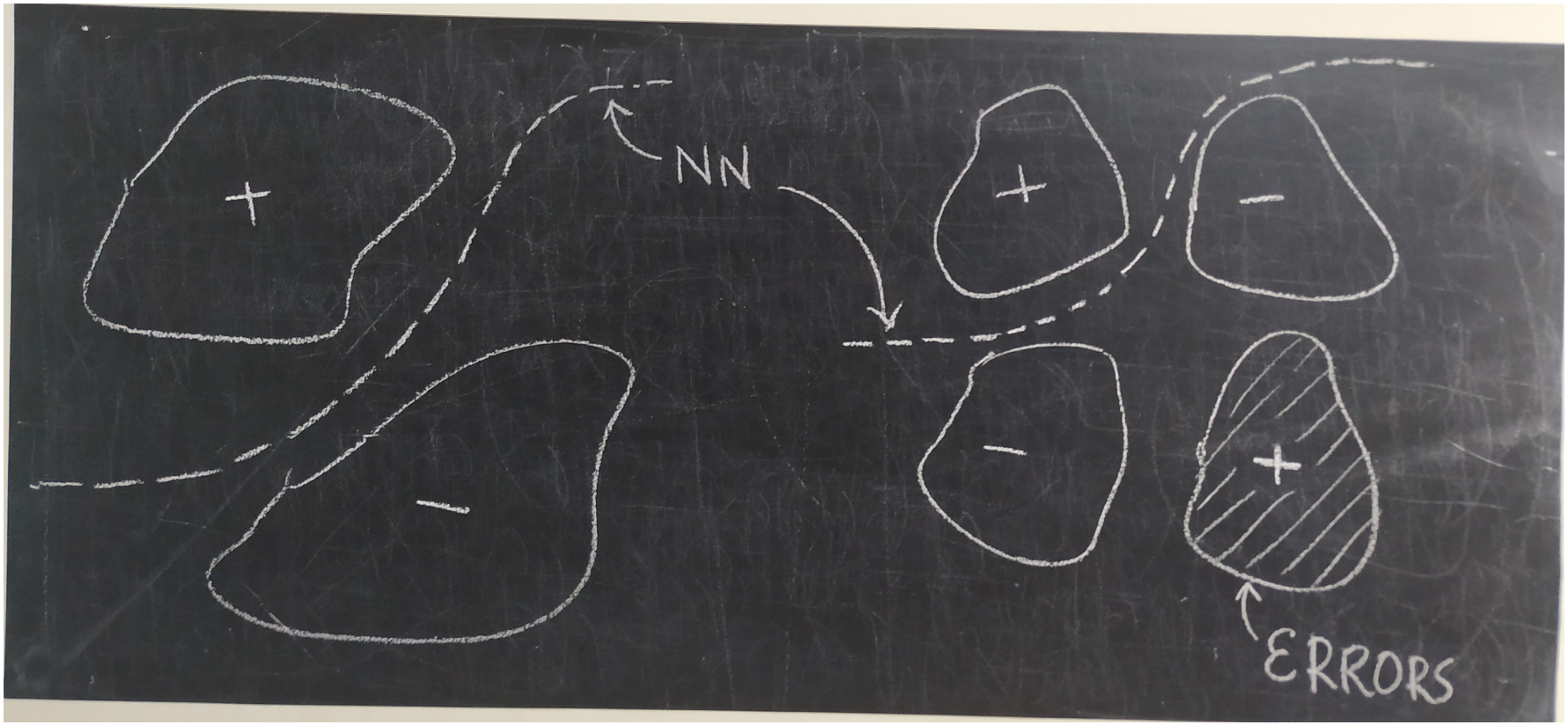
Logistic Regression as NN XX

In the NN world, these parameters **w** and b are called **synaptic weights** or "simply" **weights**; z is called the **net input**. The function that operates on the net input in any neuron is called **activation function**.

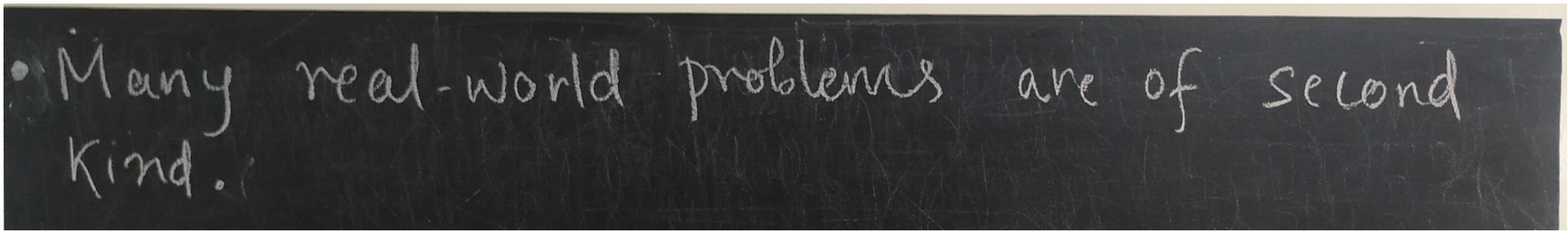
Issues with Perceptron I



Issues with Perceptron II



Issues with Perceptron III



Solution

Represent the features in a transformed feature space on which learning the model will become easier:

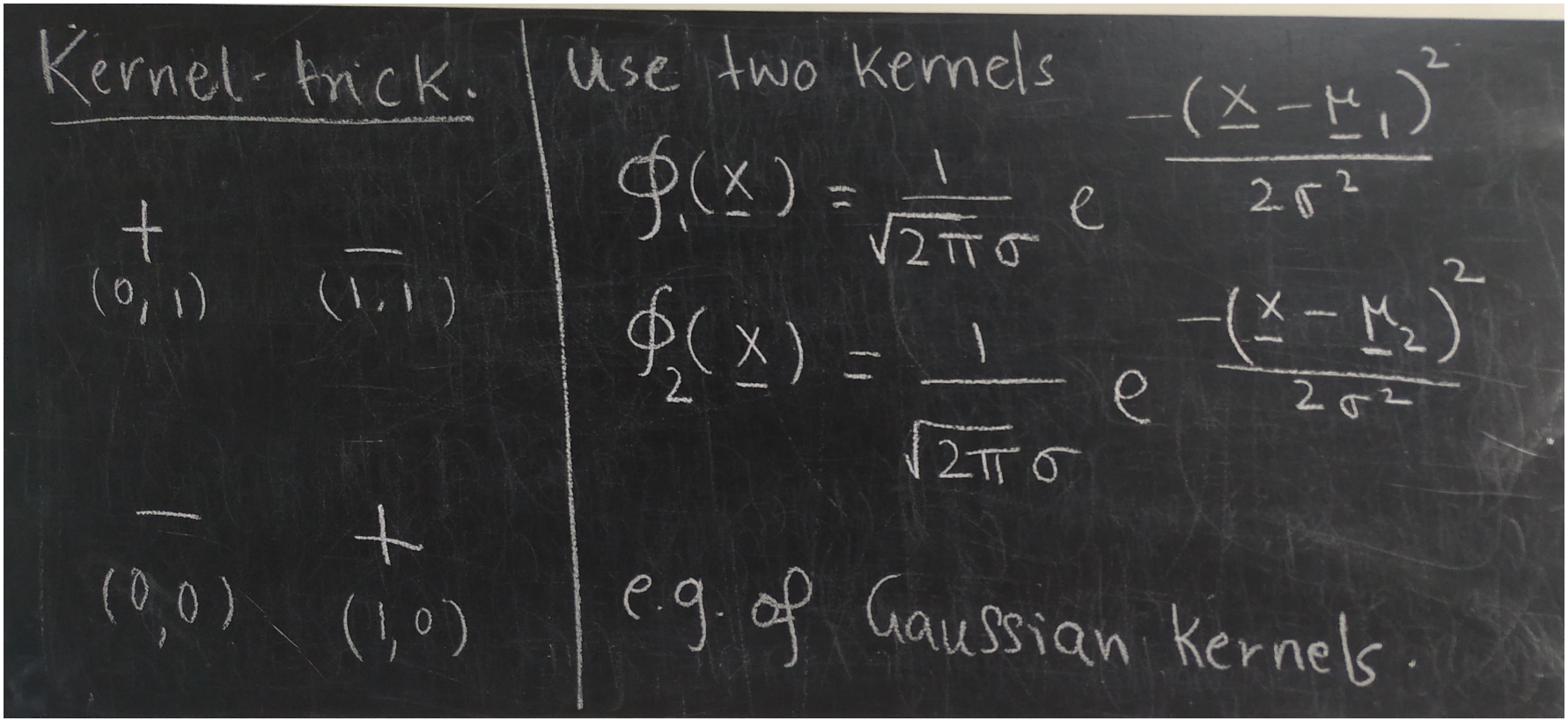
- Using kernel trick (for less-complex problem)
- By multiple layers of feature transformation (for hard problems)

This forms the basis for:

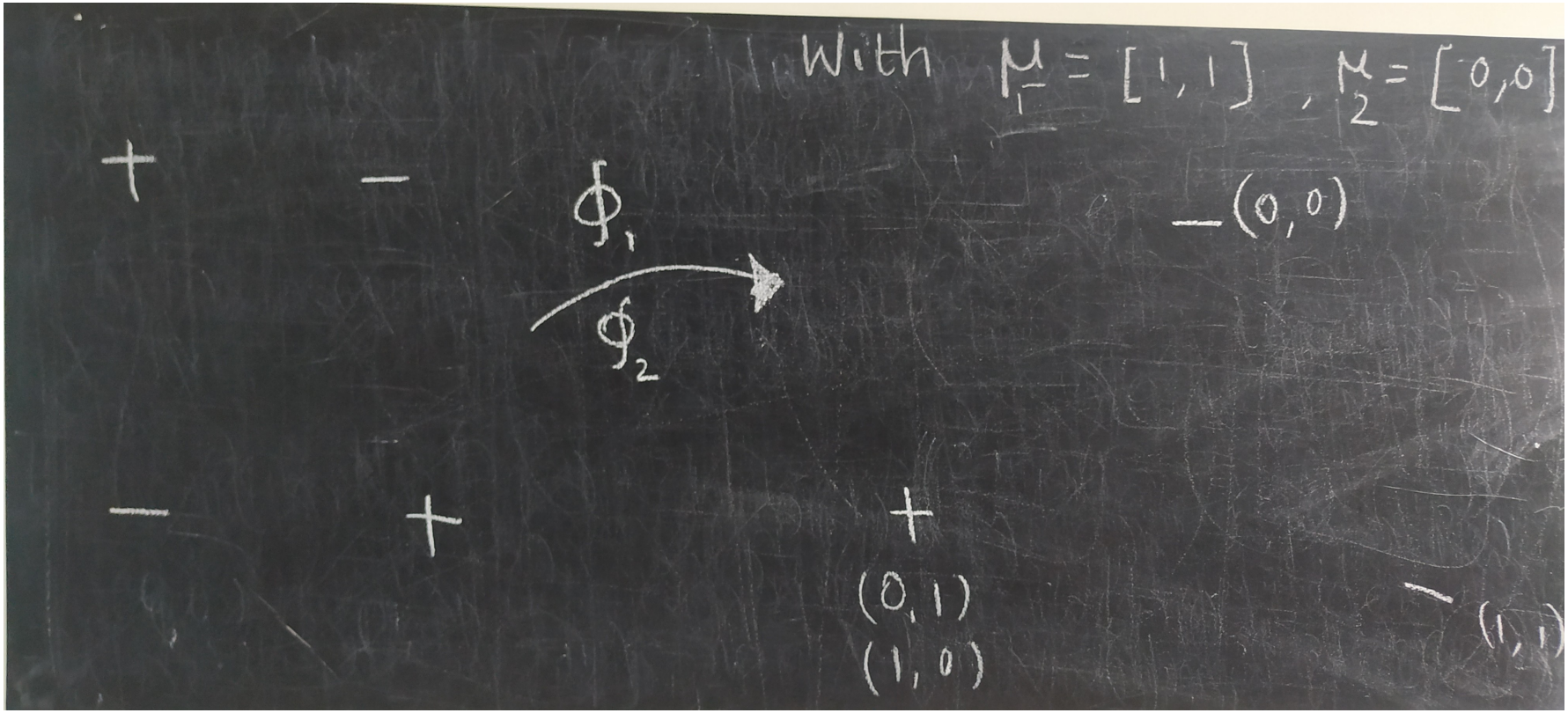
- Kernelised Neural Nets (e.g. Radial-basis Function Nets)
- Multilayered Neural Nets (e.g. Multilayer Perceptron)

Kernel Trick I

We limit our discussion to a particular kind of kernel only.



Kernel Trick II



Much of theory on RBF-Nets (next) are based on RBF kernels.

RBF-Net I

- Here, we solve the problem of classifying nonlinearly separable patterns by proceeding in a hybrid manner using two stages:
  - The first stages transforms the set of nonlinearly separable samples (patterns) into a new set, which under certain conditions, are more likely to be linearly separable in the new sample space (Mathematically justified by Cover's paper in 1965)
  - The second stage completes the solution to the prescribed classification problem by using least-squares estimation (or any other method).

RBF-Net II

- RBF networks are has a single hidden layer and an output layer.
  - The inputs take the inputs in **x**
  - The hidden units, applies a nonlinear transformation from the input space to the hidden (feature) space. [For most applications, the dimensionality of this layer should be high, so as to satisfy the Cover's theorem] (*stage 1*)
  - The output layer produces the final ouptut of the network that is then compared with the true output (*stage 2*).

RBF-Net III

Cover's Theorem (1965): A complex pattern-classification problem, cast in a high-dimensional space nonlinearly, is more likely to be linearly separable than in a low-dimensional space, provided that the space is not densely populated.

RBF-Net IV

- Recall that the models that we have studied so far is:

$$\mathbf{w}^{\mathsf{T}}\mathbf{x} = 0$$

where, $\mathbf{w}$ and $\mathbf{x}$ are d-dimensional vectors.

- Now, in stage 1, we first transform $\mathbf{x}$ via a non-linear kernel $\varphi(\cdot)$, and in stage 2, we build the model:

$$\mathbf{w}^{\mathsf{T}}\varphi(\mathbf{x}) = 0$$

where, φ contains several radial-basis functions $\{\phi_1, \ldots, \phi_m\}$.

- Also, $d < m < N$, where N is the number of data points. (Why this value of m? – See next)

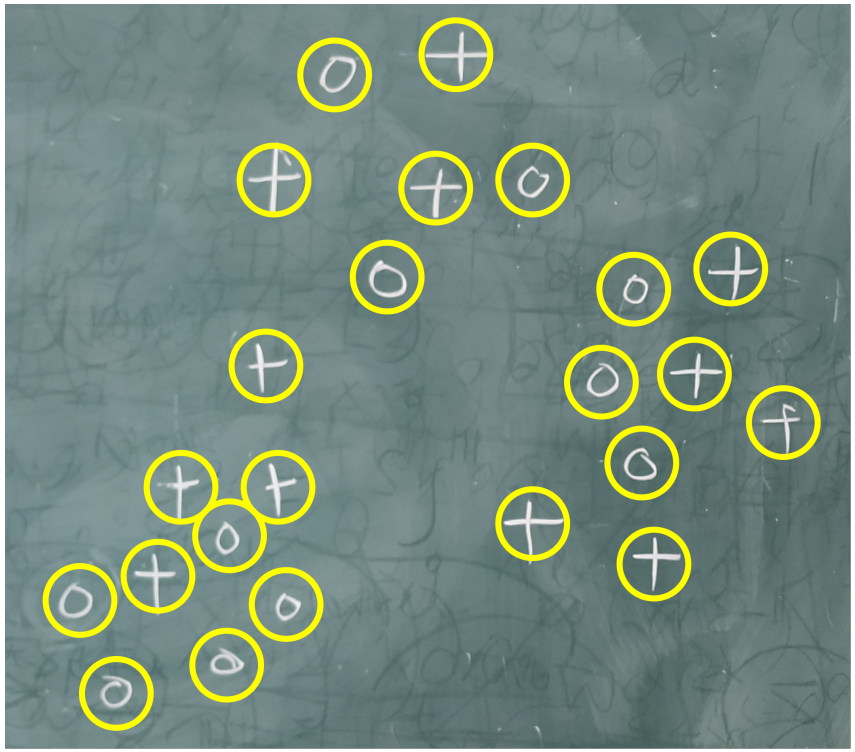
RBF-Net V

The radial-basis function ϕ_k computes a scalar value based on the similarity of a datapoint $\mathbf{x}_i$ and the center $\boldsymbol{\mu}_k$:

$$\phi_k(\mathbf{x}_i) = \frac{1}{\sqrt{2\pi\sigma_k^2}} e^{-\frac{||\mathbf{x}_i - \boldsymbol{\mu}_k||^2}{2\sigma_k^2}}$$

RBF-Net VI

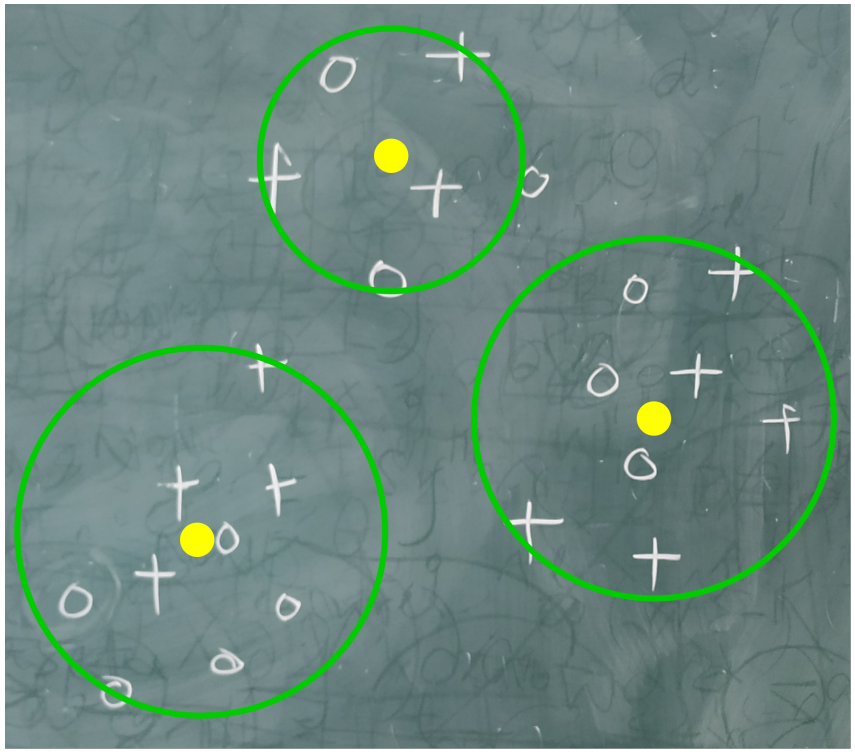
Case $m = N$ Here, it is suggesting that each datapoint be a center $\boldsymbol{\mu}_i = \mathbf{x}_i$; $1 \leq i \leq m$.



(This was the initial proposal for RBF-nets, which were based on interpolation theory)

RBF-Net VII

Case $m < N$ Here, it is suggesting that a representative of a group be a center $\boldsymbol{\mu}_i = \mathbf{c}_i$; $1 \leq i \leq m$.



Parameter: m and $\boldsymbol{\mu}$

There is a close relationship between obtaining these two parameters:

- Use k-means clustering on data $\mathbf{X}$ with $k = m$.
- The centroids($\mathbf{c}_i$s) obtained after the convergence of k-means are chosen to be $\boldsymbol{\mu}_i$s.

The k-means algorithm can be based on:

- a fixed k and $k > d$
- there are various methods to obtain a good value for k such as the elbow method, silhouette method or cross-validation.

RBF-Net IX

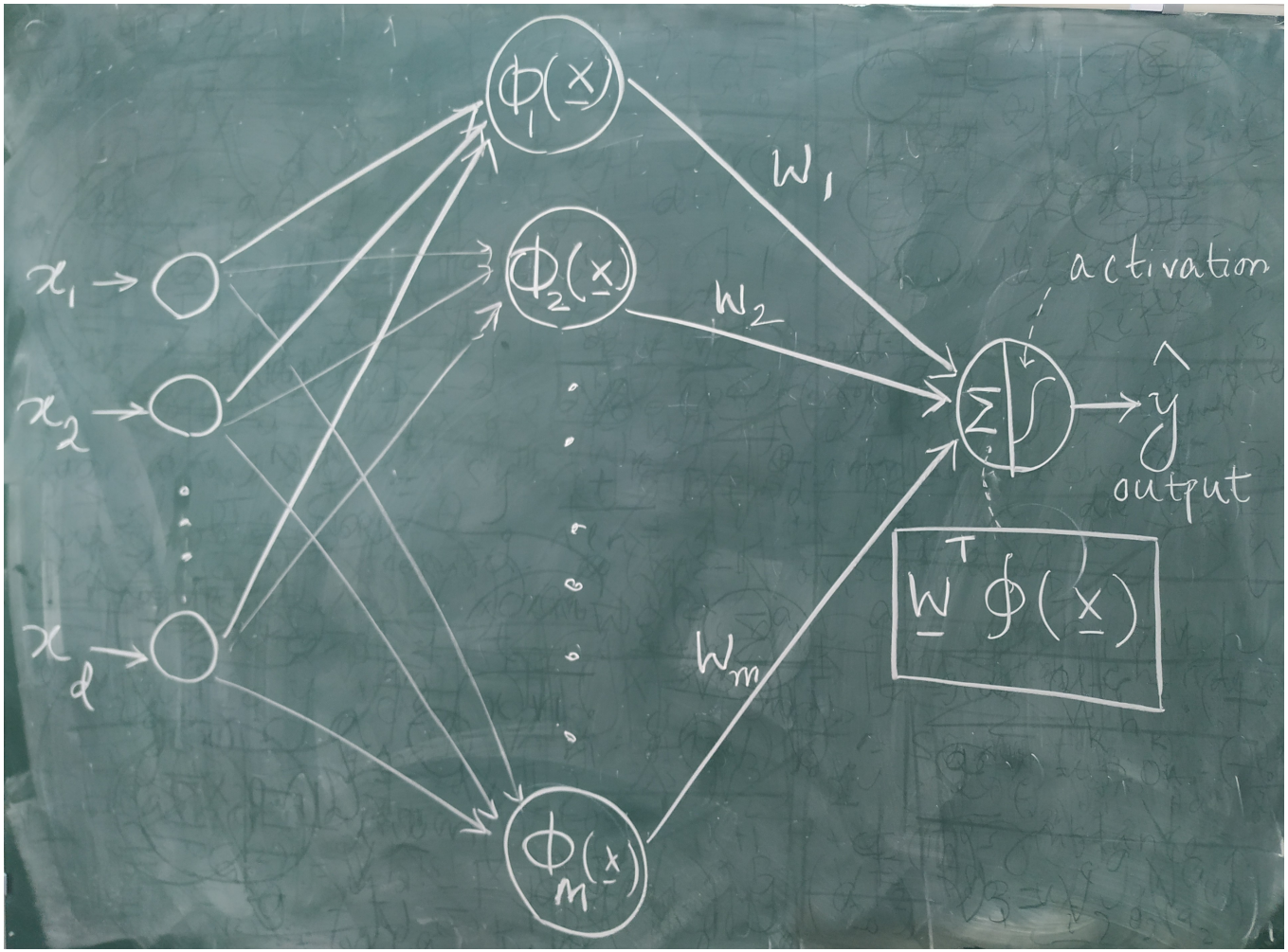
Parameter: σ

Note that so far we have not discussed about σs in the basis functions. We note these points:

- The σ parameter behaves as a 'radius' (hence the name "radial basis") of the circular spread from the center $\mathbf{c}$ or $\boldsymbol{\mu}$.
- These can be obtained from the k-means clusters discussed in the previous slide.
- These parameters are fixed to same value for all the basis functions $\phi_1, \ldots, \phi_k$.
- A typical value is 1.
- Other computationally complicated alternatives can be chosen such as treating σ (for a fixed value) or each σ_i (for $1 \leq i \leq k$) as *hyperparameters* and tuning it via cross-validation.

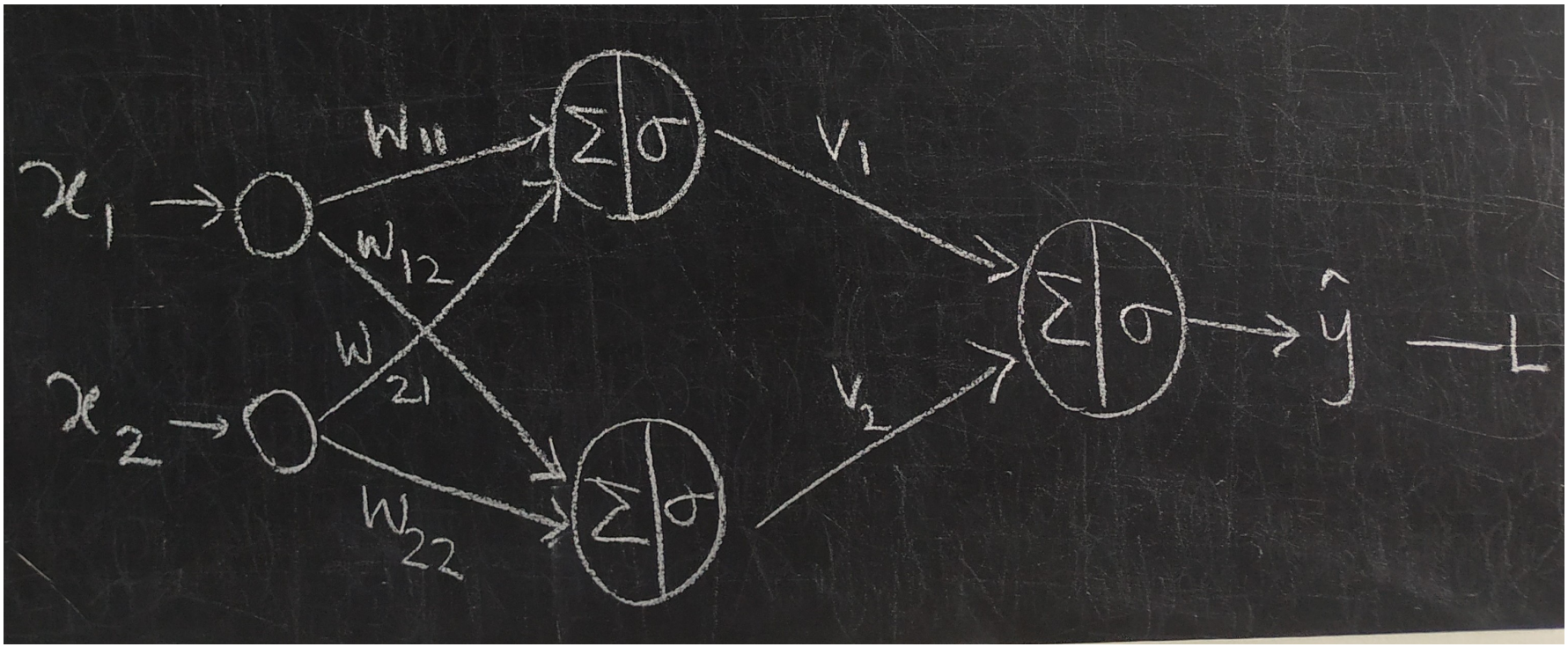
RBF-Net X

Final structure of RBF net

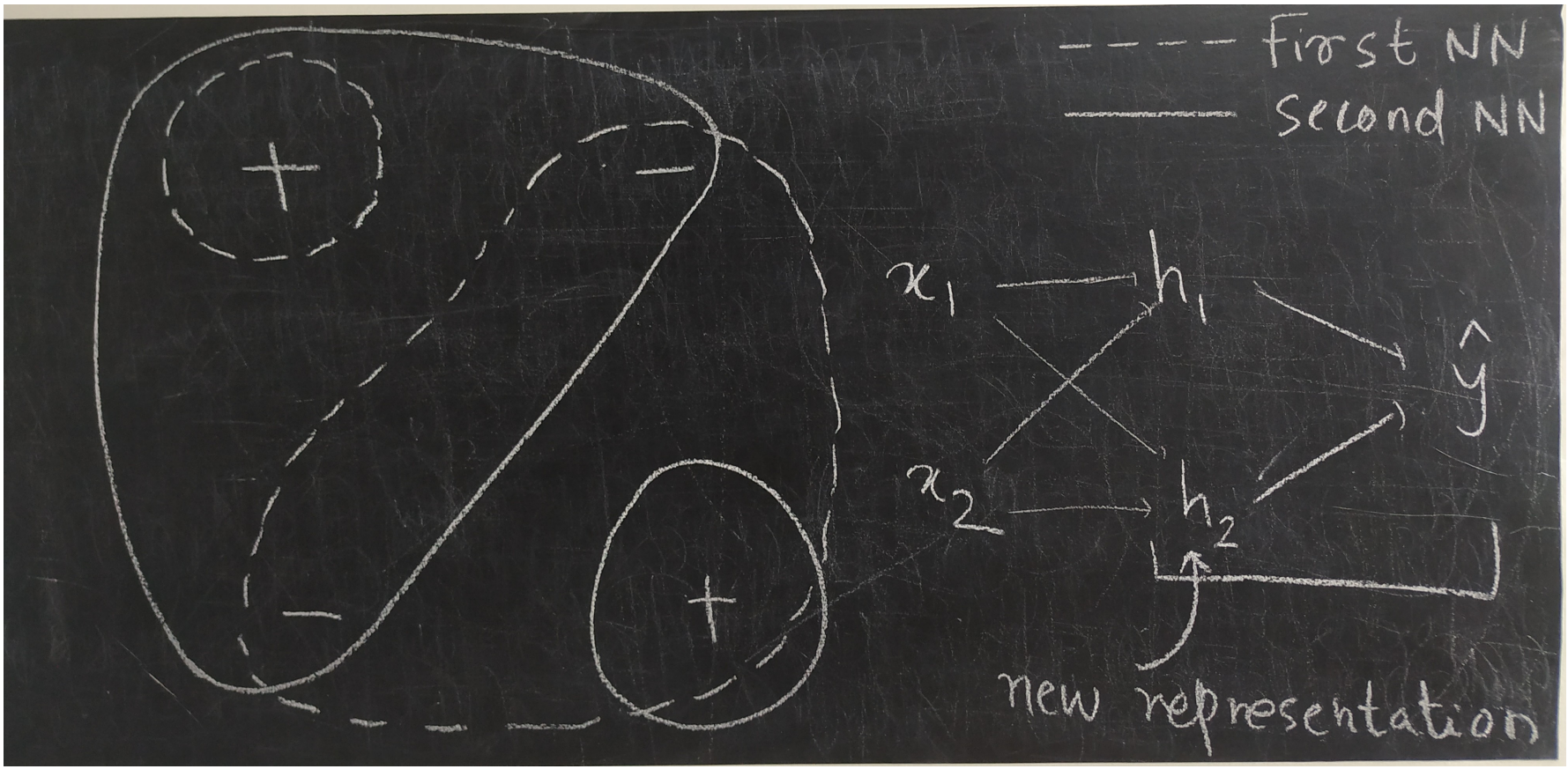


Multilayer NNs I

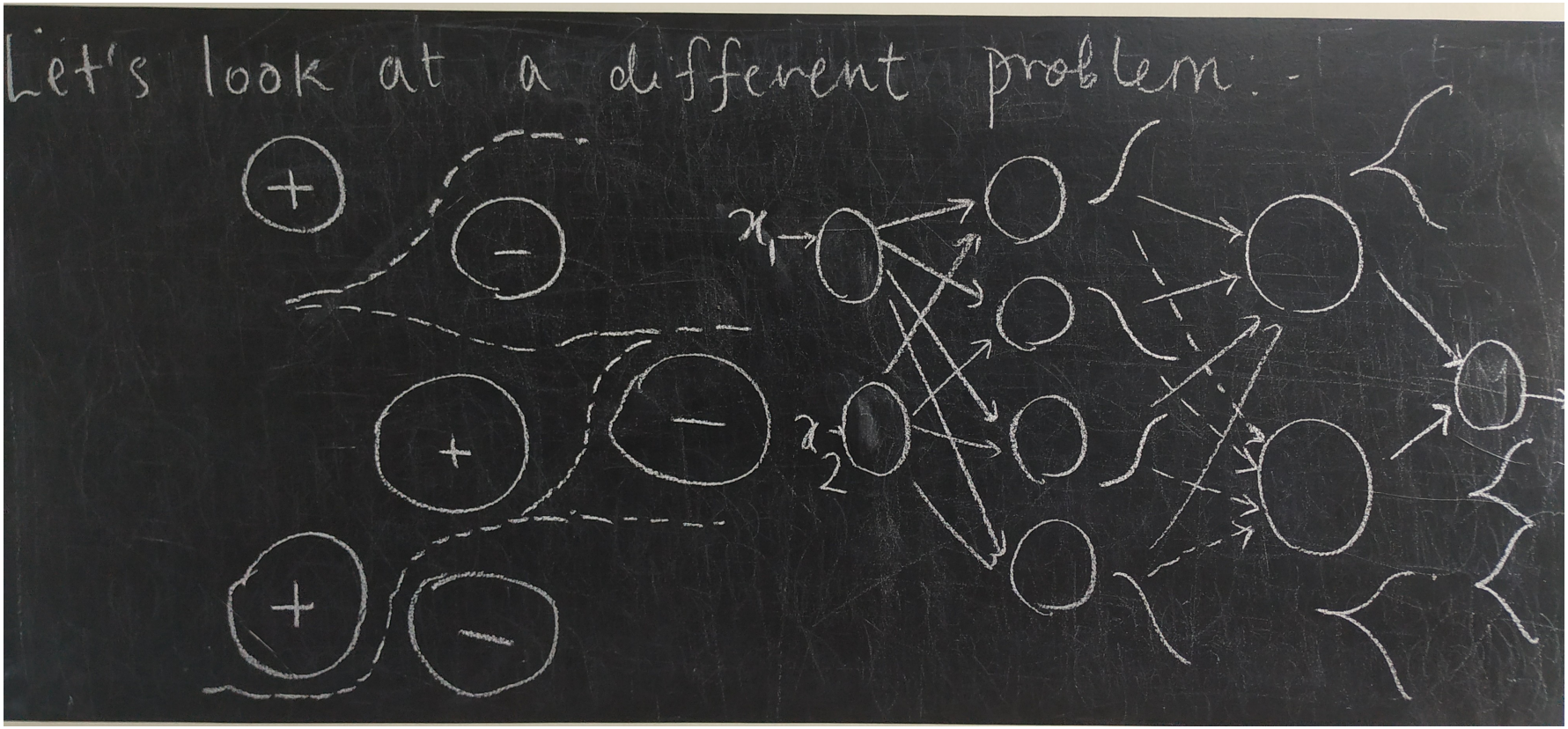
Muliple levels of transformation of features using layers:



Multilayer NNs II

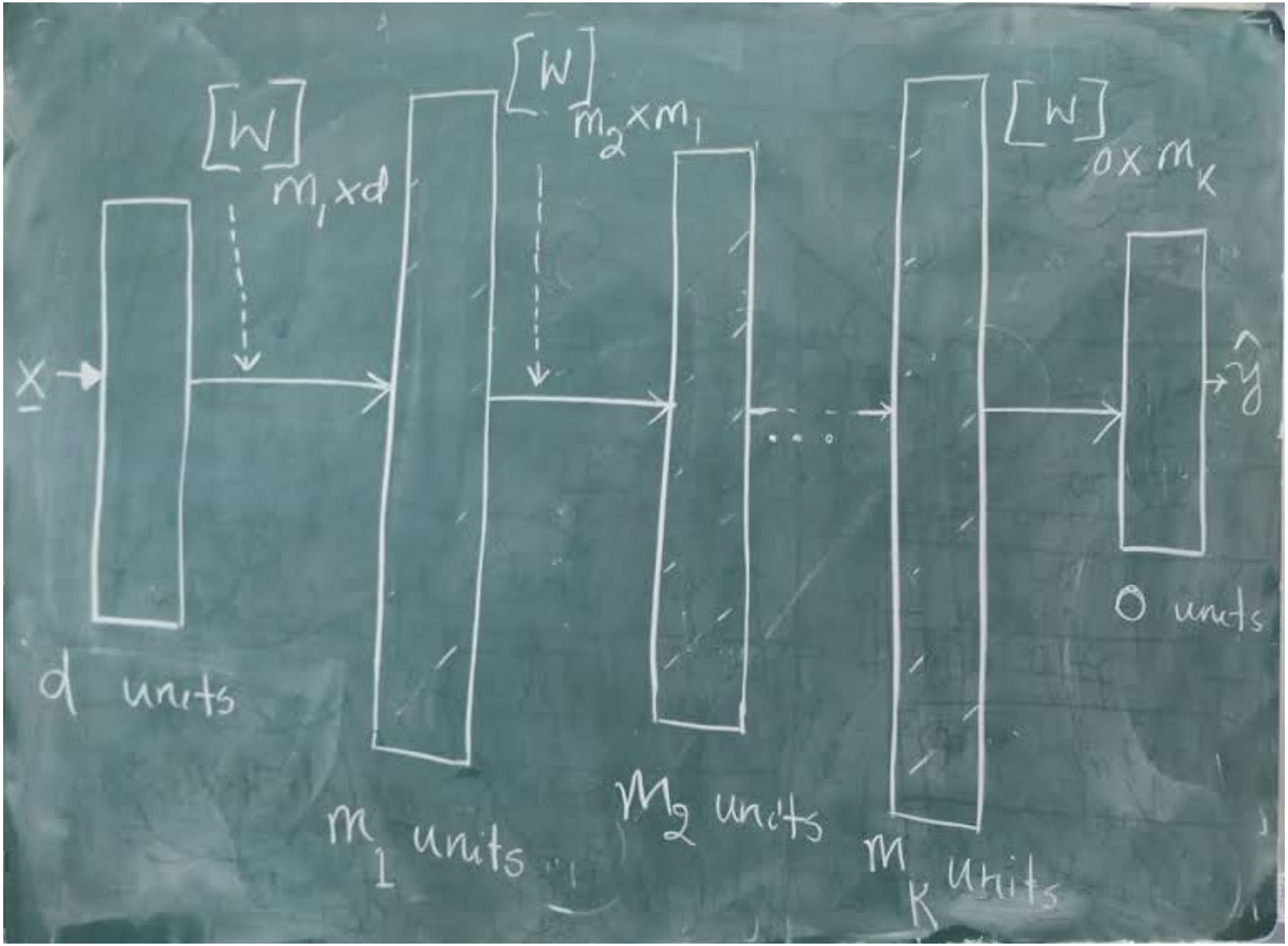


Multilayer NNs III



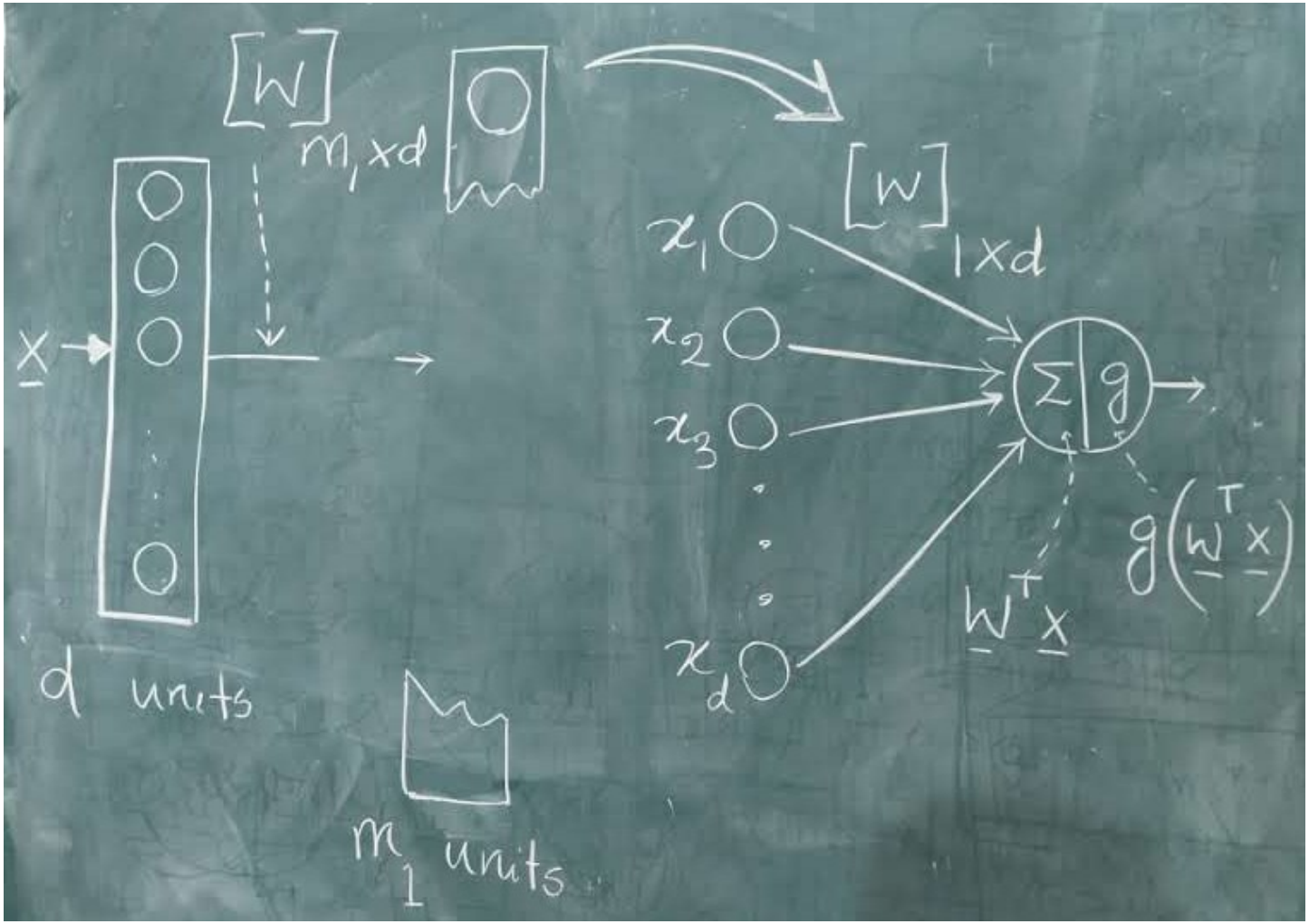
A neural network that is based on the this idea is 'Multilayer Perceptron (MLP)'.

Multilayer NNs IV



Multilayer NNs V

Visualising a single neuron in a hidden layer:



Multilayer NNs VI

Some difficulties concerning the structure of an MLP:

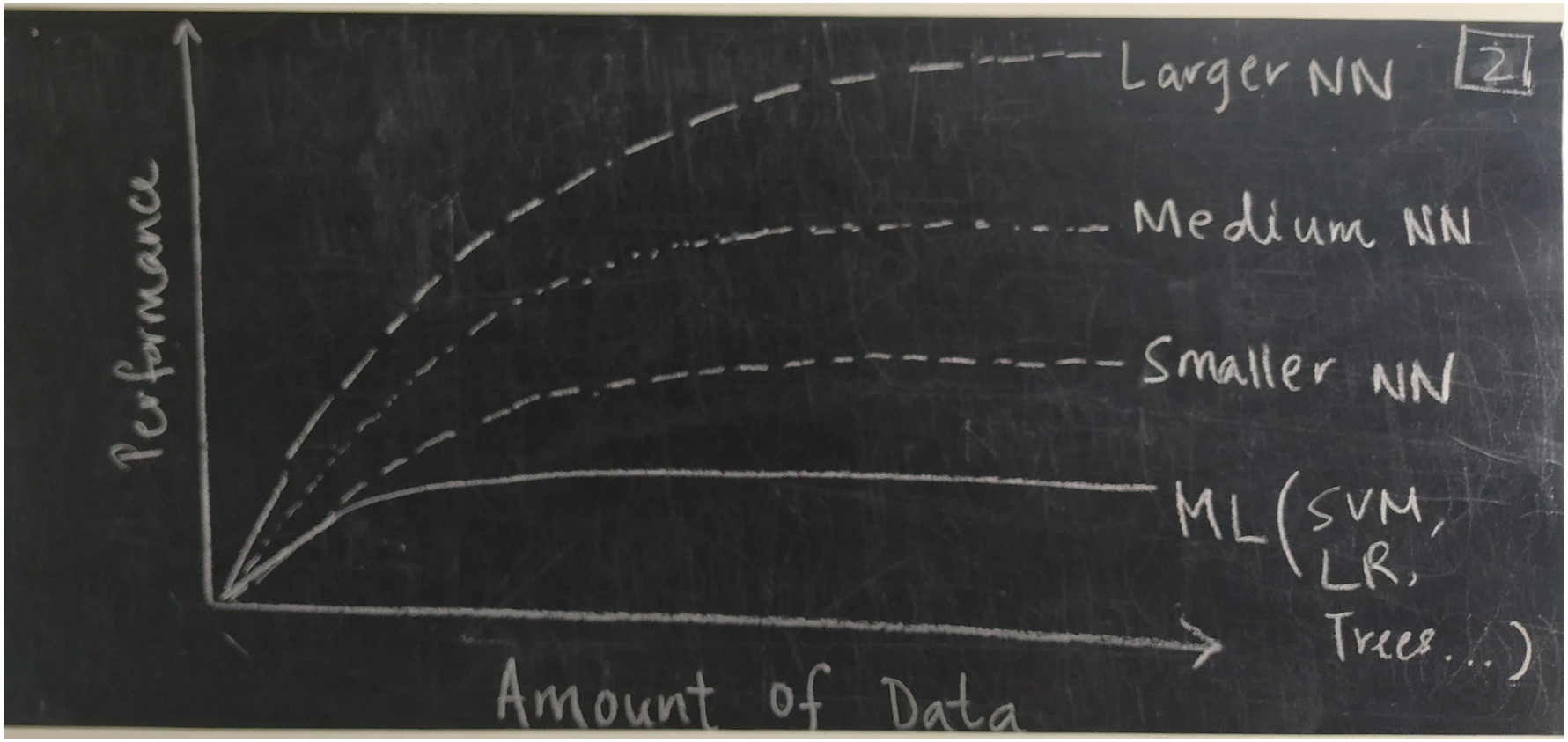
1. How many hidden layers for a given problem?
2. How many hidden units per layer?
3. What activation function to choose?

Multilayer NNs VII

Hidden layers and units:

1. The hidden layers behave like feature detectors.
2. Early layers detect low-level features, and later layers detect high-level features.
3. Both these parameters can be considered as hyperparameters and they can be tuned using cross-validation method.
4. These parameters influence the performance of the model.

Multilayer NNs VIII



Multilayer NNs IX

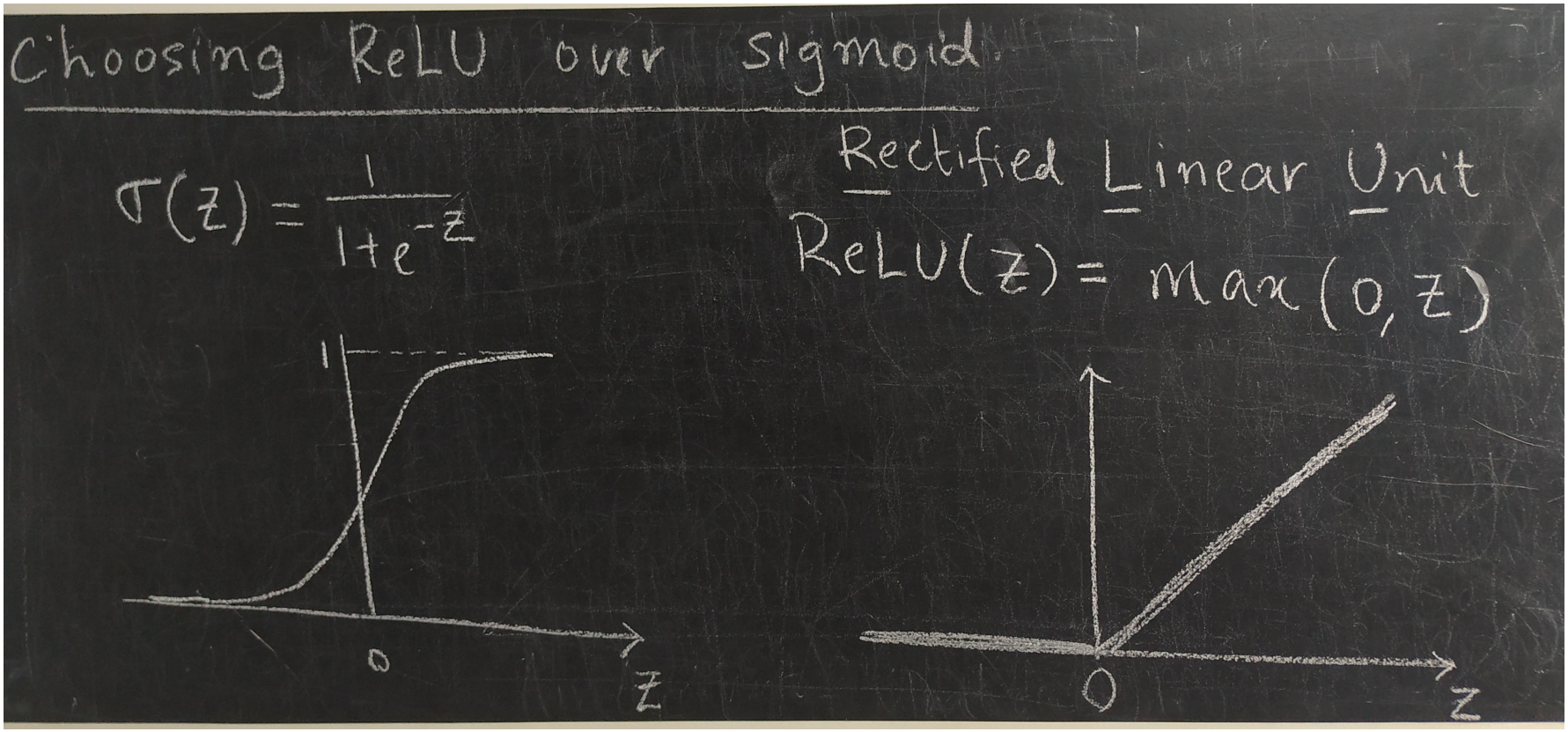
Activation function:

- No activation for inputs
- Output activation depends on the target domain:

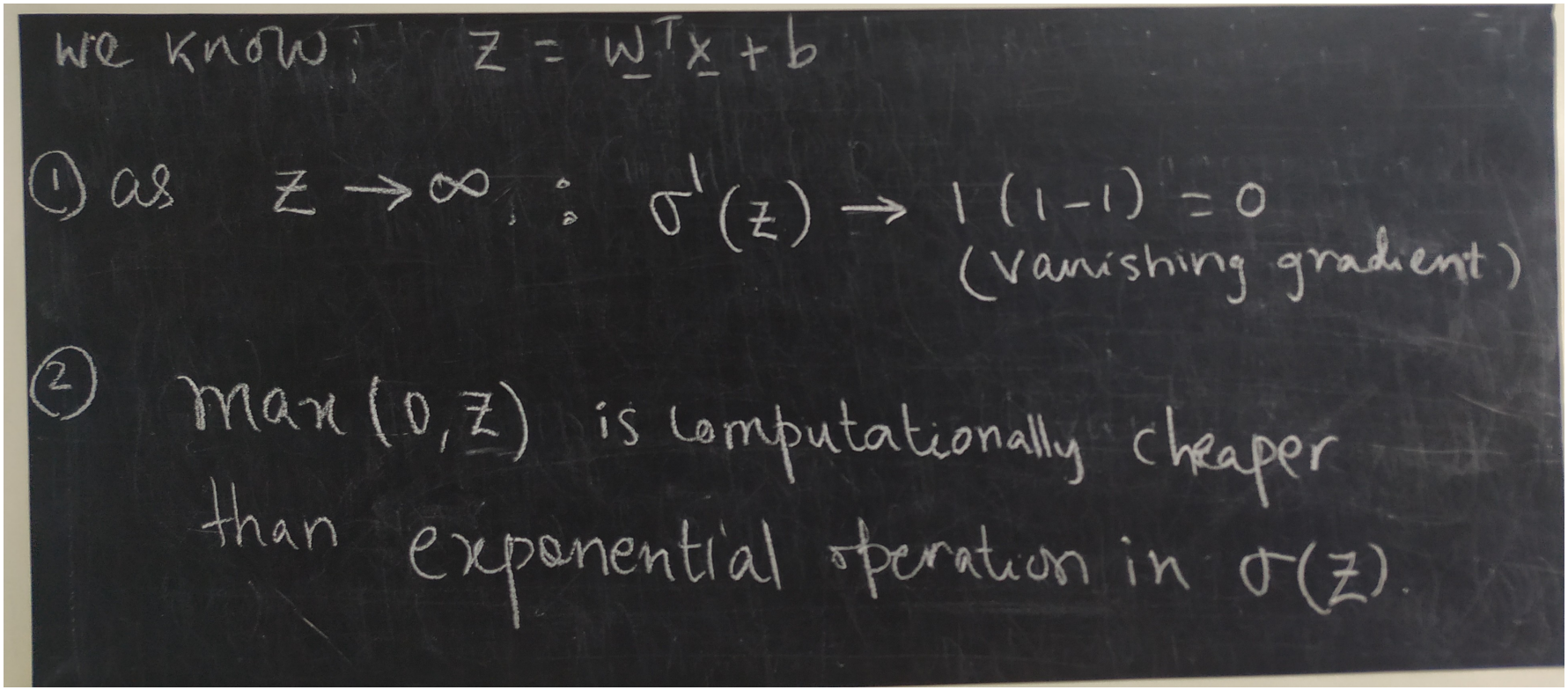
Problem	# of output units	$g(\cdot)$
$y \in \mathbb{R}$	1	linear
$y \in \{0, 1\}$	1	sigmoid
$y \in \{0, 1, \ldots, c\}$	c	softmax

- Hidden unit activations are non-linear:
  - sigmoid
  - tanh
  - ReLU and its variants

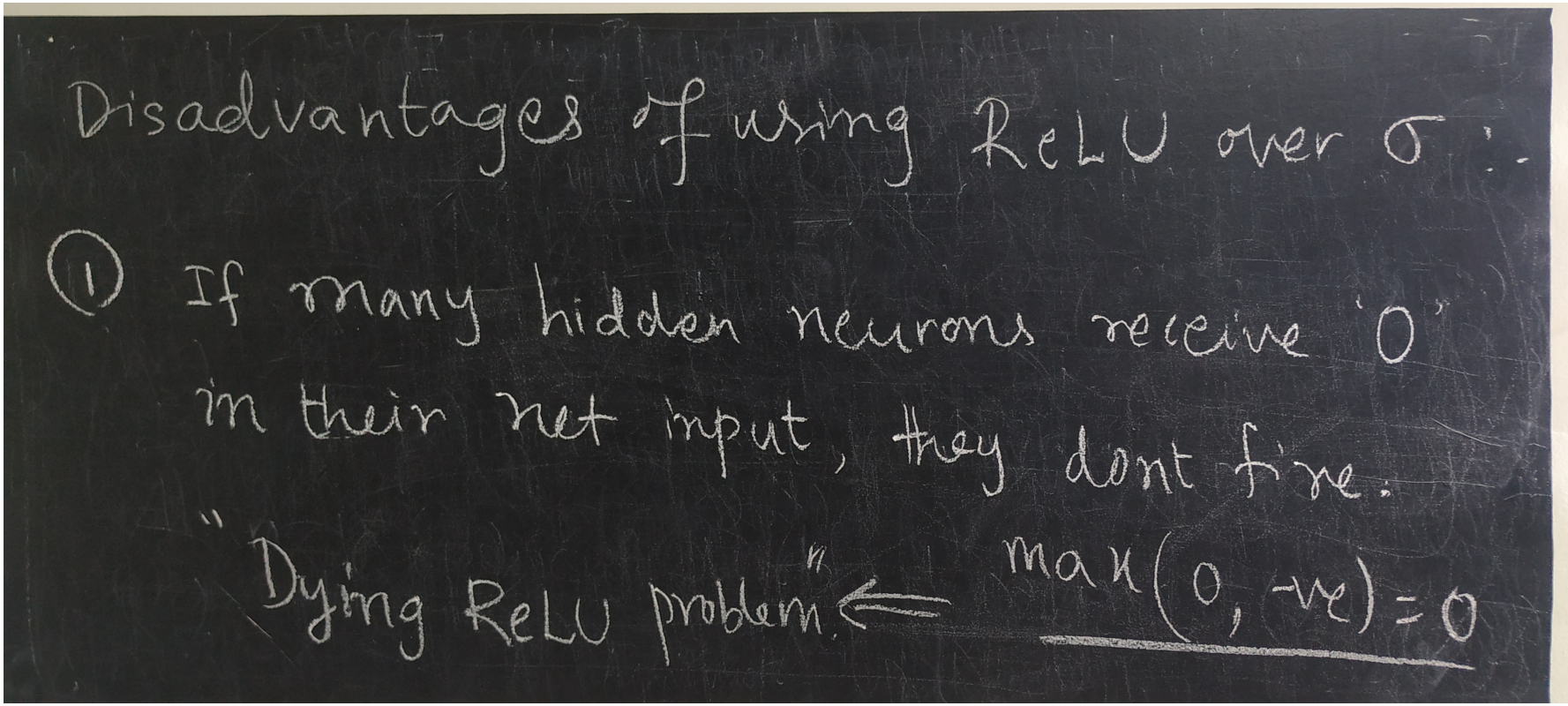
ReLU activation:



Multilayer NNs XI



Multilayer NNs XII



Two variants of ReLU function:

Multilayer NNs XIII

- Leaky ReLU:

$$\text{LeakyReLU}(z) = \max(0.01z, z)$$

- Parametric ReLU:

$$\text{pReLU}(z) = \max(\alpha z, z)$$

Multilayer NNs XIV

Weights:

The synaptic weights (**w**s) for each layer of an MLP are optimised using gradient descent. The procedure is popularly known as 'Backpropagation'.

Training using Backpropagation I

- Training a single-layer (Input–Output) neural network is straightforward because the loss function is a direct function of weights **w**s. Gradient computation is easy.
- For multi-layer networks, the loss is a complicated composition function of the weights in earlier layers.
- The gradient of a composition function is computed using the backpropagation procedure.
- The backpropagation procedure leverages the chain rule of differential calculus, which *computes the error gradients* in terms of *summations of local-gradient products* over the *various paths from a node to the output*.
- There can be many paths from input to outputs. Dynamic programming is useful to compute sub-solutions.
- Backprop has two phases: forward phase and backward phase.

Training using Backpropagation II

Forward phase:

- computes the output values for every nodes in the network
- computes the local derivatives at various nodes

Backward phase:

- accumulates the products of the local derivative values over all paths from the node to the output.
- These accumulated gradients are used to update the weights.

Training using Backpropagation III

(Based on: Neural Networks and Learning Machine by Simon Haykin)
There are some changes in notations:

Notation	Meaning
n	iterator over instances
$\mathbf{x}(n)$	nth input instance
$d(n)$	true output for $\mathbf{x}(n)$
$y_j(n)$	output* from a neuron j

*If the neuron is in output layer: it is the computed output. If it is in the hidden layer, it is an output of that layer and it becomes an input to the next layer.

Training using Backpropagation IV

Loss or error function:

- Consider an MLP with an input layer of source nodes, a set of hidden layers and an output layer that may consist of a set of output neurons.
- Let $\mathcal{D} = \{\mathbf{x}(n), \mathbf{d}(n)\}_{n=1}^{N}$ denote the training samples used to train the network.
- Let $y_j(n)$ denote the function signal produced at the output of neuron j at the output layer by the stimulus $\mathbf{x}(n)$ applied at the input layer.
- Correspondingly, the error signal produced at the output of neuron j is defined by

$$e_j(n) = d_j(n) - y_j(n) \tag{1}$$

where $d_j(n)$ is the jth element of the desired-response vector $\mathbf{d}(n)$.

Training using Backpropagation V

- The instantaneous error energy can be computed as

$$\mathcal{E}_j(n) = \frac{1}{2} e_j^2(n) \tag{2}$$

- Summing the error-energy contributions of all the neurons in the output layer, we can express the total instantaneous error energy of the whole network as

$$\mathcal{E}(n) = \sum_{j \in C} \mathcal{E}_j(n) \tag{3}$$

$$= \frac{1}{2} \sum_{j \in C} e_j^2(n) \tag{4}$$

where C is the set of all neurons in the output layer.

Training using Backpropagation VI

- Consider the output neuron j:

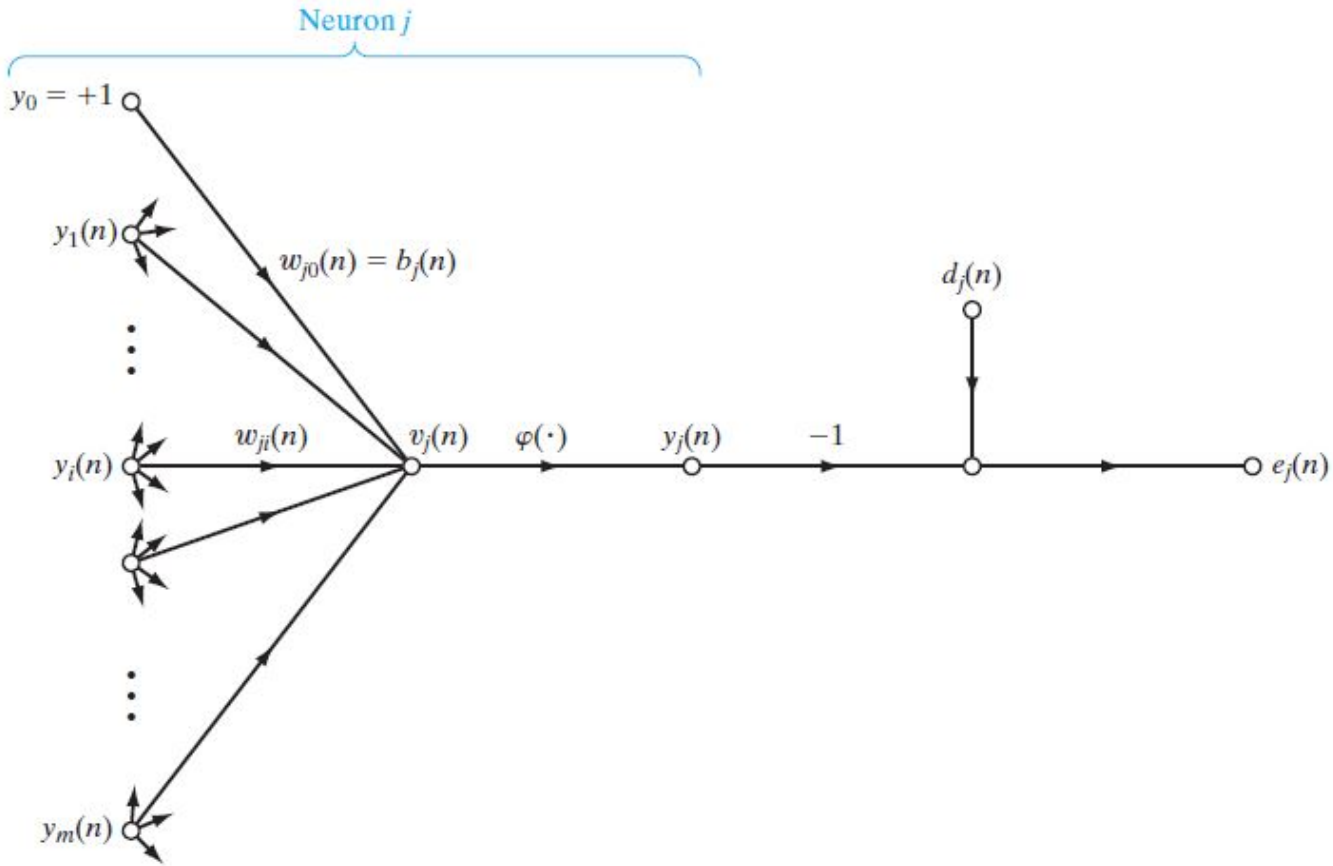


Figure: Signal-flow graph for output node j

Training using Backpropagation VII

- Let focus on the backpropagation procedure developed with SGD for free parameter tuning.
- From the figure: The output neuron j is fed by the function signals produced by a layer of neurons to its left.
- The induced local field $v_j(n)$ produced at the input of the activation function associated with neuron j is therefore

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n) \tag{5}$$

where m is the total number of inputs (excluding the bias) applied to neuron j. The synaptic weight w_{j0} (corresponding to the fixed input $y_0 = +1$) equals the bias b_j applied to neuron j. Hence, the function signal $y_j(n)$ appearing at the output of neuron j at iteration n is

$$y_j(n) = \varphi_j(v_j(n)) \tag{6}$$

Training using Backpropagation VIII

- The backpropagation procedure applies a correction $\Delta w_{ji}(n)$ to the synaptic weight $w_{ji}(n)$, which is proportional to the partial derivative $\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}$.
- According to chain rule of derivatives, we can express this gradient as

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathcal{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)} \tag{7}$$

- We can obtain every term at right of the above equation using the available definitions.
- From the definition of instantaneous energy function, we can get

$$\frac{\partial \mathcal{E}(n)}{\partial e_j(n)} = e_j(n) \tag{8}$$

Training using Backpropagation IX

- Similarly,

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \tag{9}$$

- Differentiating $y_j(n) = \varphi_j(v_j(n))$ w.r.t. $v_j(n)$ gives

$$\frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n)) \tag{10}$$

- Finally,

$$\frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n) \tag{11}$$

- So, we can write

$$\frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n) \tag{12}$$

Training using Backpropagation X

- The correction $\Delta w_{ji}(n)$ applies to $w_{ji}(n)$ is defined by the *delta rule*, or

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)} \tag{13}$$

where η is the learning rate parameter of the backpropagation procedure. If we remember the gradient descent, we were taking a step opposite to the direction of the gradient. Therefore, the – sign is used in the delta rule that signify the same in the weight space.

- Now, we can write

$$\Delta w_{ji}(n) = \eta\ e_j(n)\varphi'_j(v_j(n))y_i(n) \tag{14}$$

- The term at the right $e_j(n)\varphi'_j(v_j(n))$ can be written as $\delta_j(n)$ which is called the local gradient at the jth node.

$$\Delta w_{ji}(n) = \eta\ \delta_j(n)y_i(n) \tag{15}$$

Training using Backpropagation XI

- Based on the location of the neuron j, the calculation of the local gradient may be different. In other words, the computation of local gradient is somewhat easier for output layer because a proper error signal is there to back propagate. However, for the a neuron in the hidden layer, there is no such direct error signal.
- However, based on the concept of credit assignment, each hidden neuron share some amount of responsibility towards the computed error at their next layer.
- **Neuron j is an output neuron**: When neuron j is located in the output layer of the network, it is supplied with a desired response of its own. The computation is straightforward for this neuron. (See earlier equations)

Training using Backpropagation XII

- **Neuron j is a hidden neuron**: When neuron j is located in a hidden layer of the network, there is no specified desired response for that neuron.
- The error signal for a hidden neuron would have to be determined recursively and working backwards in terms of the error signals of all the neurons to which that hidden neuron is directly connected.
- See the following figure:

Training using Backpropagation XIII

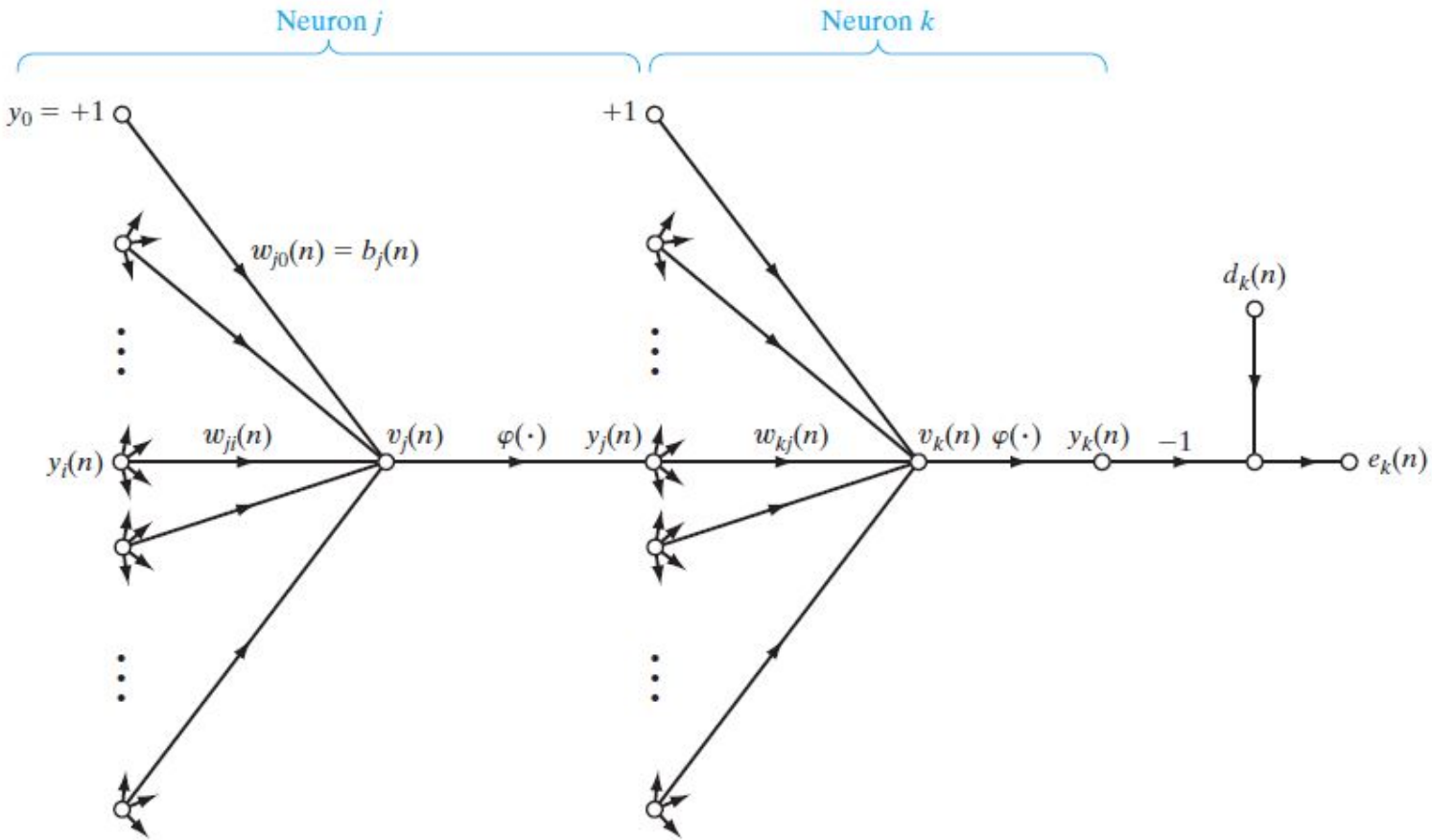


Figure: Output node k connected to hidden neuron j

Training using Backpropagation XIV

- We now redefine the local gradient $\delta_j(n)$ as

$$\delta_j(n) = -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \tag{16}$$

$$= -\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} \varphi'(v_j(n)) \tag{17}$$

where, neuron j is hidden.

- Now to compute $\frac{\partial \mathcal{E}(n)}{\partial y_j(n)}$ we have to use the figure given above:

$$\mathcal{E}(n) = \frac{1}{2} \sum_{k \in C} e_k^2(n) \tag{18}$$

where neuron k is an output node. (This is from our previous equation of $\mathcal{E}(n)$ with just j replaced with k – just avoid the confusion of j now being called the hidden neuron).

Training using Backpropagation XV

- Differentiating above equation w.r.t. $y_j(n)$, we get

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)} \tag{19}$$

- From chain rule for partial derivatives, $\frac{\partial e_k(n)}{\partial y_j(n)}$ can be replaced in the above equation as

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)} \tag{20}$$

- However, we know that $e_k(n) = d_k(n) - y_k(n)$ or

$$e_k(n) = d_k(n) - \varphi(v_k(n)) \tag{21}$$

note, k is the output node here.

Training using Backpropagation XVI

- Hence,

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi'(v_k(n)) \tag{22}$$

- We also note from the above figure that for neuron k, the induced local field is

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \tag{23}$$

where m is the total number of inputs (excluding the bias) applied to the neuron k.

- The synaptic weight $w_{k0}(n)$ is equal to the bias $b_k(n)$ applied to neuron k, and the corresponding input is fixed at the value $+1$. Differentiating $v_k(n)$ with respect to $y_j(n)$ yields

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n) \tag{24}$$

Training using Backpropagation XVII

- Coming back to the term $\frac{\partial \mathcal{E}(n)}{\partial y_j(n)}$, we can now write

$$\frac{\partial \mathcal{E}(n)}{\partial y_j(n)} = -\sum_k e_k \varphi'(v_k(n)) w_{kj}(n) \tag{25}$$

$$= -\sum_k \delta_k(n) w_{kj}(n) \tag{26}$$

- Therefore the local gradient of a hidden neuron j can be obtained by putting the above equation in the local gradient equation as (in Eq.(38))

$$\delta_j(n) = \varphi'_j(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \tag{27}$$

where, neuron j is hidden.

Training using Backpropagation XVIII

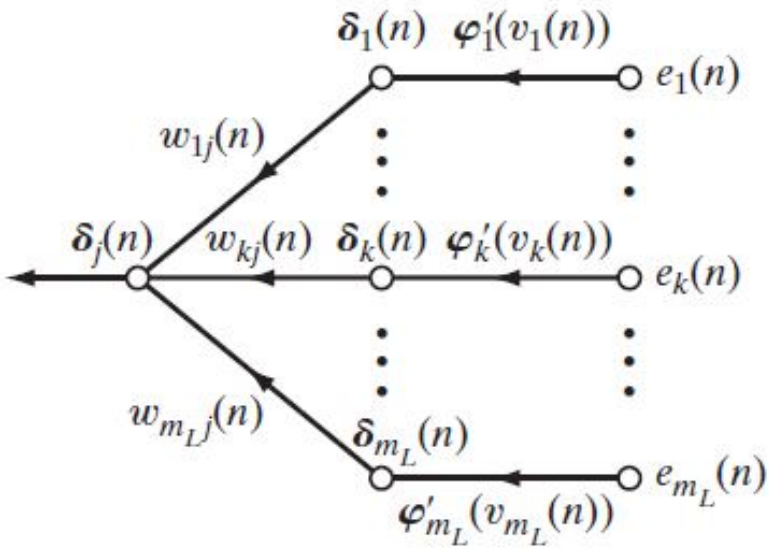


Figure: Error signal-flow graph for hidden neuron

- The outside factor $\varphi_j'(v_j(n))$ involved in the computation of the local gradient $\delta_j(n)$ in above equation depends solely on the activation function associated with hidden neuron j.
- The remaining factor involved in this computation, namely, the summation over k depends on two sets of terms:

Training using Backpropagation XIX

- The first set of terms, the $\delta_k(n)$, requires knowledge of the error signals $e_k(n)$ for all neurons that lie in the layer to the immediate right of hidden neuron j and that are directly connected to neuron j
- The second set of terms, the $w_{kj}(n)$, consists of the synaptic weights associated with these connections.

- Summary of backpropagation procedure:

$$\begin{pmatrix} \text{Weight} \\ \text{correction} \\ \Delta w_{ji}(n) \end{pmatrix} = \begin{pmatrix} \text{learning-} \\ \text{rate parameter} \\ \eta \end{pmatrix} \times \begin{pmatrix} \text{local} \\ \text{gradient} \\ \delta_j(n) \end{pmatrix} \times \begin{pmatrix} \text{input signal} \\ \text{of neuron } j, \\ y_i(n) \end{pmatrix}$$

Figure: Computation of weight correction

- Based on the location of neuron j, values have to be substituted.

Training using Backpropagation XX

- Then, all the weight updates at iteration n are:

$$w_{ji}(n) = w_{ji}(n) + \Delta w_{ji}(n)$$