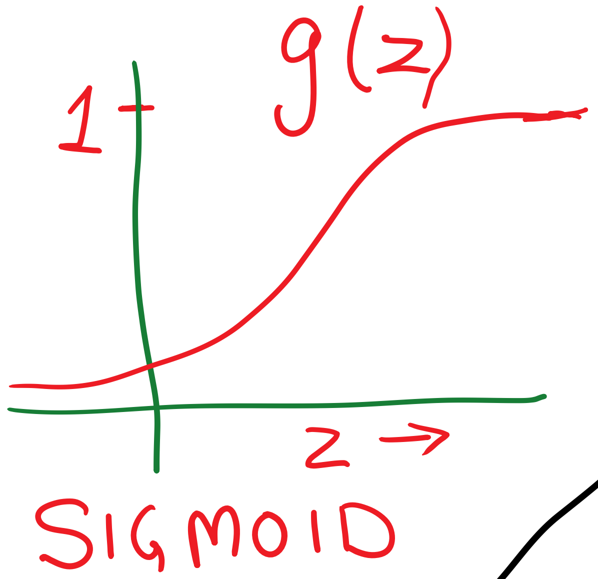
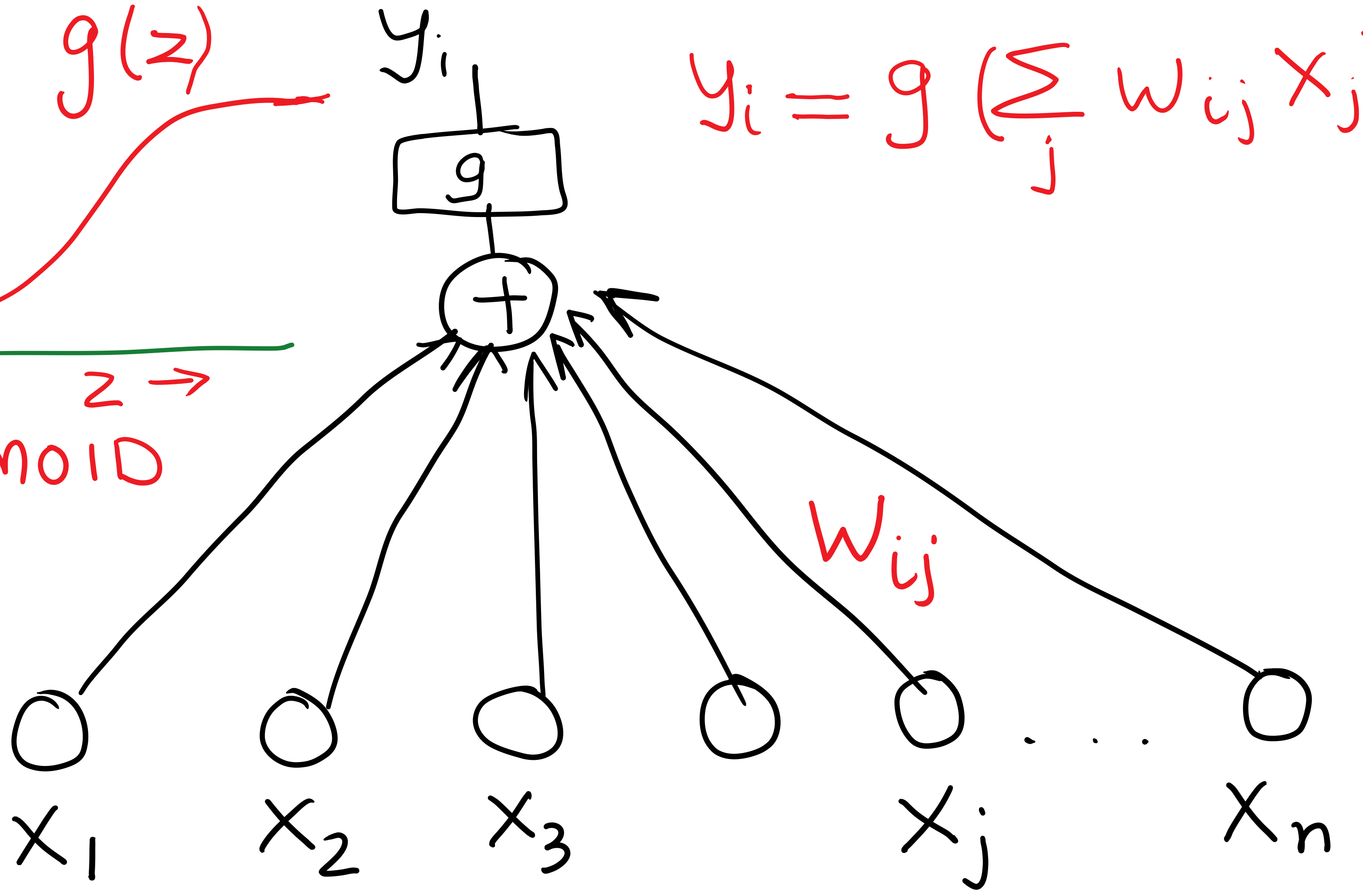
Neural Networks

Jitendra Malik

CS 189

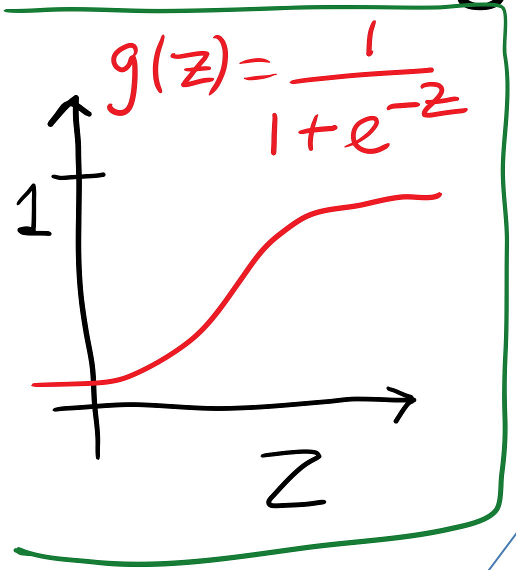
Mathematical Abstraction

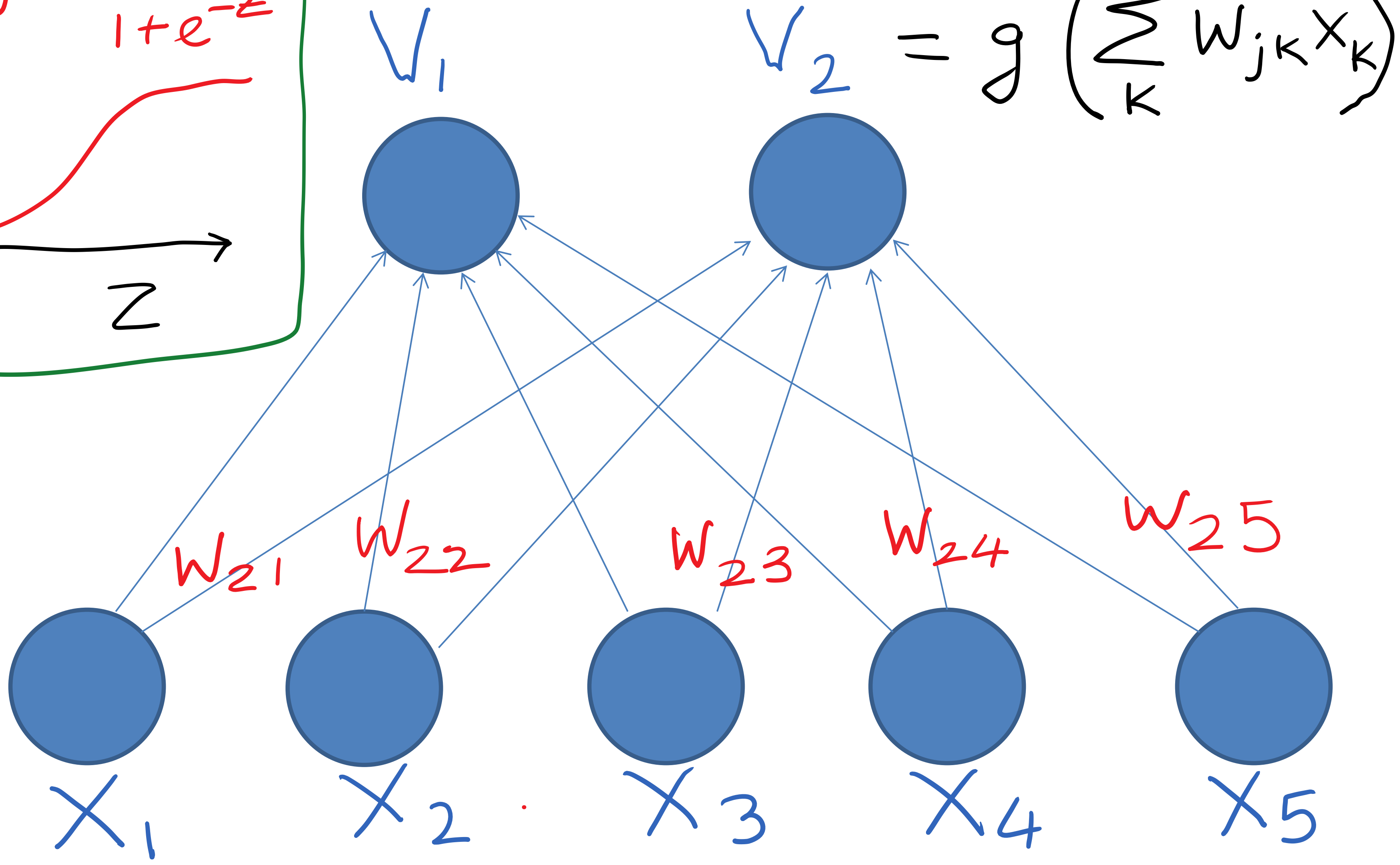




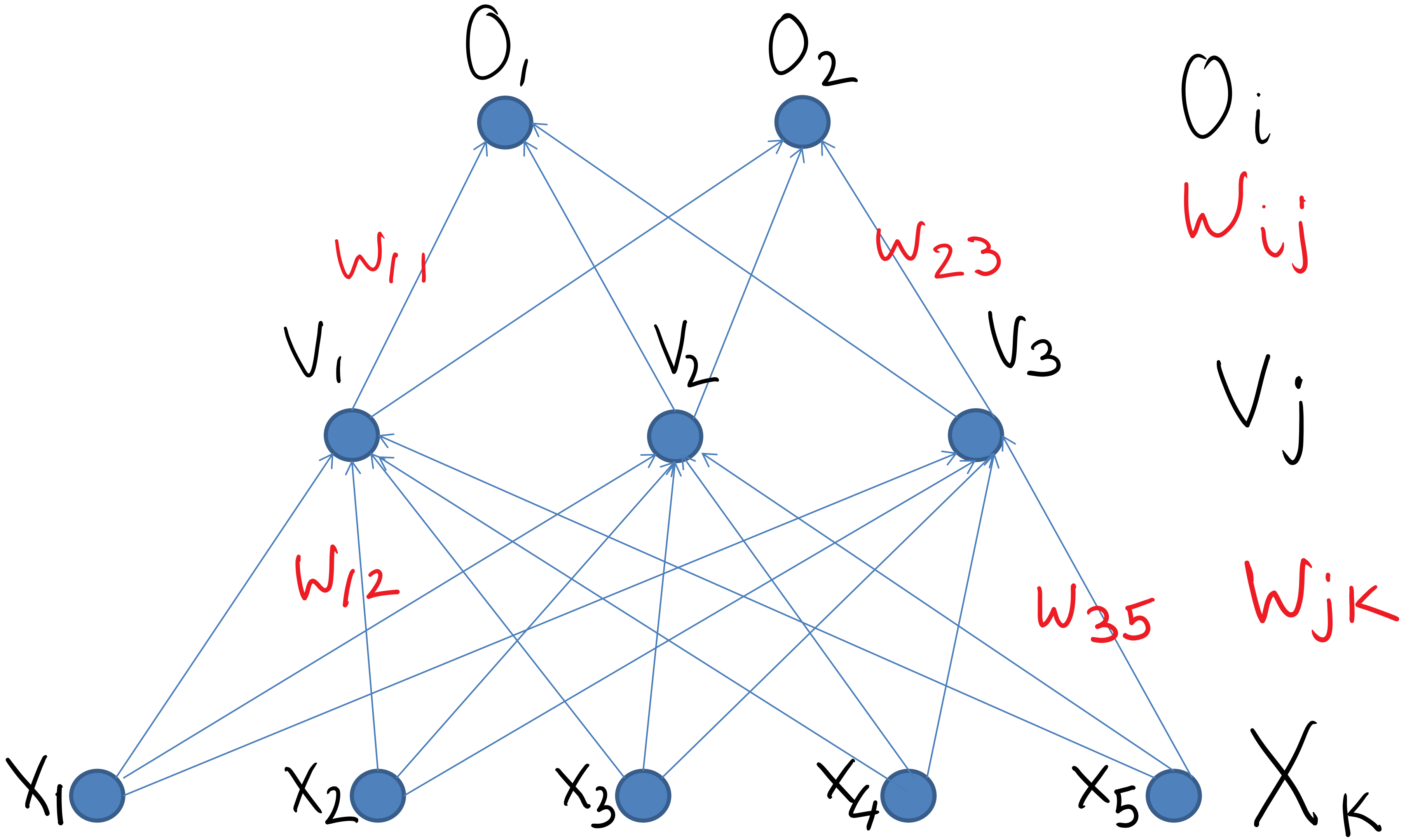
$$y_i = g\left(\sum_j w_{ij} x_j\right)$$

Single layer neural network

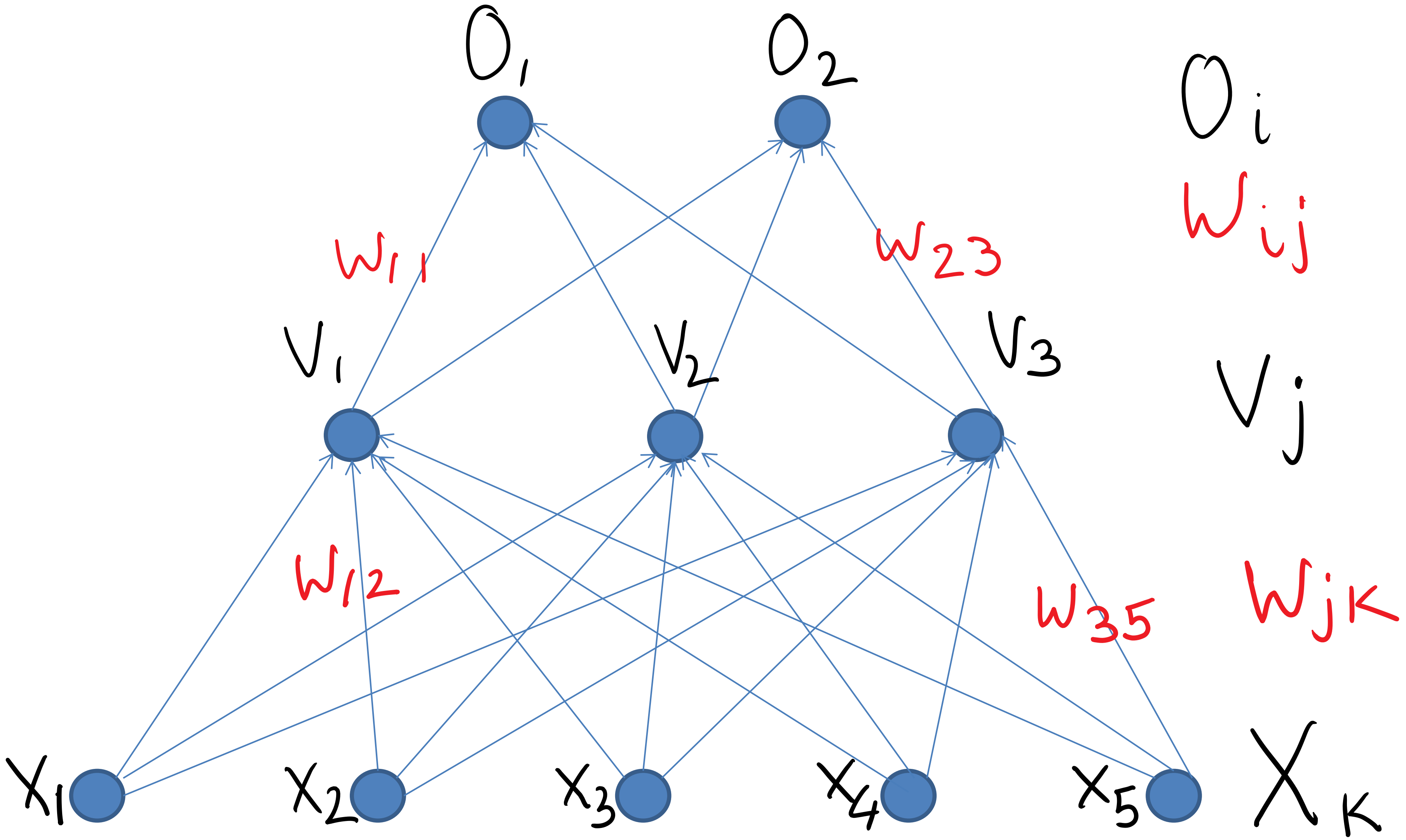




Two layer neural network



$$O_i = g\left(\sum_j W_{ij}\, g\left(\sum_k W_{jk}\, x_k\right)\right)$$



Recall from multivariable calculus that the *gradient* of f is the vector

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_d} \end{bmatrix}.$$

The gradient is the direction which leads to a maximal increase of f. Similarly, the negative gradient is the direction of *steepest descent*. Gradient descent uses this fact to construct an algorithm: at every step, compute the gradient and follow that direction to minimize f.

Training a neural network

Goal: Find W such that O_i is as close as possible to y_i (desired output)

Approach:
- Define loss function $\mathcal{L}(W)$
- Compute $\nabla_W \mathcal{L}$
- $W_{new} \leftarrow W_{old} - \eta \nabla_W \mathcal{L}$

Training a single layer neural network

- For binary classification, a good choice of loss function is the cross entropy. For regression, use squared error

$$L = -\sum_{\text{input data}} \left(y_i \ln O_i + (1-y_i)\ln(1-O_i) \right)$$

- We model the activation function *g* as a sigmoid

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

- Finding w reduces to logistic regression!

We can use STOCHASTIC GRADIENT DESCENT.

Training a multi layer neural network

- We compute the gradient with respect to all the weights: from input to hidden layer, and hidden layer to output layer. This requires computing the partial derivative of the error with respect to each weight Wij
- The complexity of computing the gradient in the naïve version is quadratic in the number of weights. The back-propagation algorithm is a trick that enables it to be computed in linear time.

The Chain Rule from Multivariable Calculus

Given $u(x, y) = x^2 + 2y$ where $x(r, t) = r\sin(t)$ and $y(r,t) = \sin^2(t)$,

determine the value of $\partial u / \partial r$ and $\partial u / \partial t$ using the chain rule.

The Chain Rule from Multivariable Calculus

Given $u(x, y) = x^2 + 2y$ where $x(r, t) = r \sin(t)$ and $y(r,t) = \sin^2(t)$, determine the value of $\partial u / \partial r$ and $\partial u / \partial t$ using the chain

$$\frac{\partial u}{\partial r} = \frac{\partial u}{\partial x}\frac{\partial x}{\partial r} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial r} = (2x)(\sin(t)) + (2)(0) = 2r\sin^2(t),$$

and

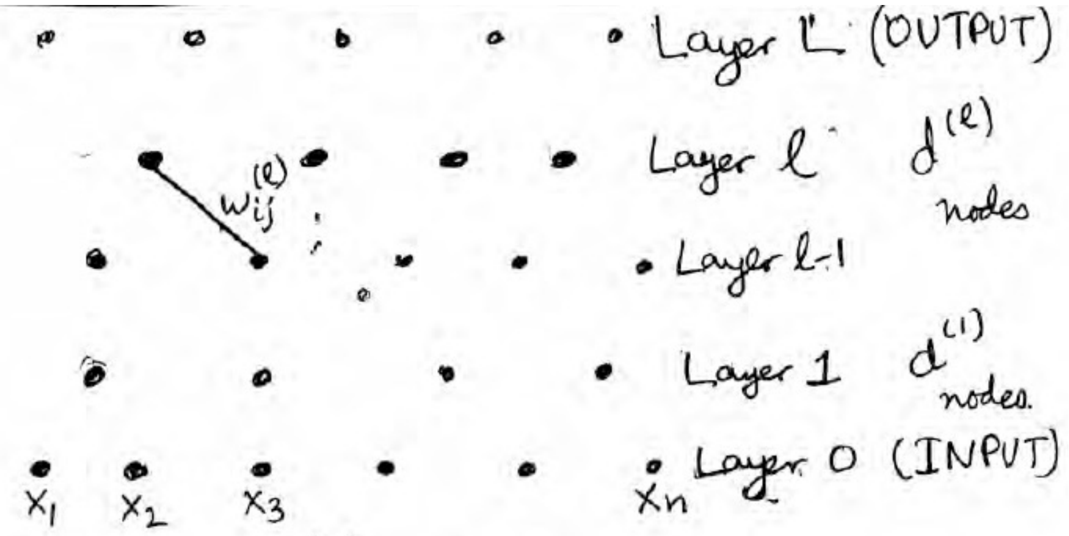
$$\begin{aligned}\frac{\partial u}{\partial t} &= \frac{\partial u}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial u}{\partial y}\frac{\partial y}{\partial t}\\ &= (2x)(r\cos(t)) + (2)(2\sin(t)\cos(t))\\ &= (2r\sin(t))(r\cos(t)) + 4\sin(t)\cos(t)\\ &= 2(r^2+2)\sin(t)\cos(t)\\ &= (r^2+2)\sin(2t).\end{aligned}$$

Our goal

- To compute the partial derivative of error with respect to Wij
- We will work this out separately for connections feeding into the output layer and then for any other layer.
- It turns out to be useful to define two variables Sj (the summed input to a node j) δj (the partial derivative of error w. r. t Sj)

The Notation

Use superscripts to keep track of layers



Output of neuron j in layer l

$$x_j^{(l)} = g\left(\sum_{i=0}^{d^{(l-1)}} w_{ij}^{(l)} x_i^{(l-1)}\right)$$

$$= g\left(s_j^{(l)}\right)$$

$s_j^{(l)}$ is the weighted input to node j in layer l

$w_{ij}^{(l)}$ is weight from node i (in layer l-1) to node j (in layer l)

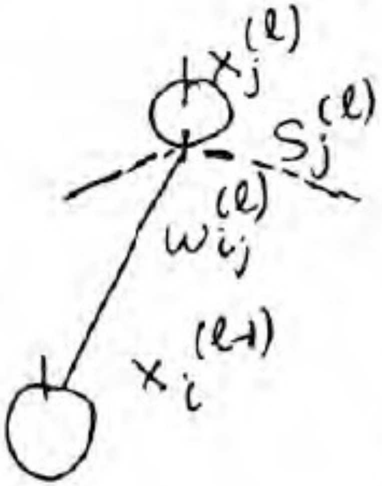
Defining δ

Partial derivative of error with respect to s, the summed input

Error on example $(\underset{\sim}{x}_n, y_n)$ is $e(h(\underset{\sim}{x}_n), y_n) = e(w)$

We use h to denote the function computed by the neural network.

$$\nabla e(w) = \frac{\partial e(w)}{\partial w_{ij}^{(l)}}$$



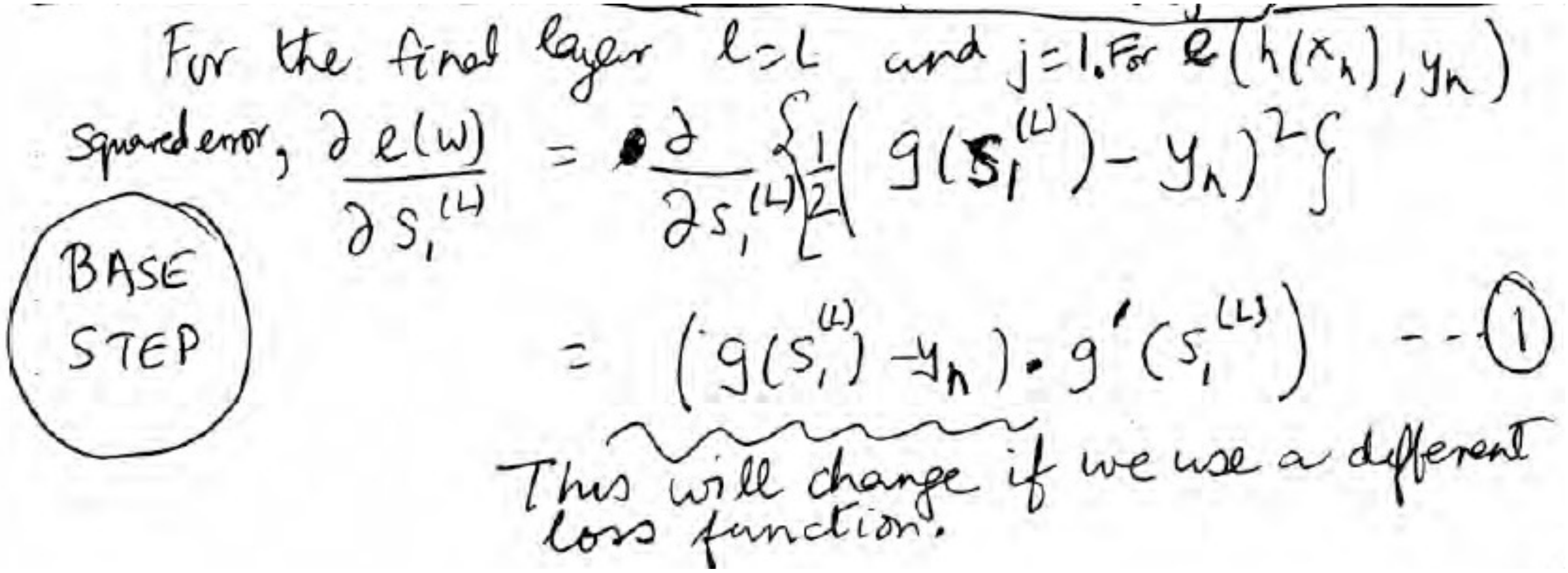
$$\frac{\partial e(w)}{\partial w_{ij}^{(l)}} = \frac{\partial e(w)}{\partial s_j^{(l)}} \times \frac{\partial s_j^{(l)}}{\partial w_{ij}^{(l)}}$$

$$= \delta_j^{(l)} \qquad x_i^{(l-1)}$$

$$\boxed{\text{Define } \delta_j^{(l)} = \frac{\partial e(w)}{\partial s_j^{(l)}}}$$

Computing δ for neuron in final layer

(this example is for squared loss, as in regression)



For the final layer $l = L$ and $j = 1$. For $e(h(x_n), y_n)$ squared error,

BASE STEP

$$\frac{\partial e(w)}{\partial s_1^{(L)}} = \frac{\partial}{\partial s_1^{(L)}} \left\{ \frac{1}{2} \left(g(s_1^{(L)}) - y_n \right)^2 \right\}$$

$$= \underbrace{\left(g(s_1^{(L)}) - y_n \right)}_{} \cdot g'(s_1^{(L)}) \quad \text{- -} \; (1)$$

This will change if we use a different loss function.

Computing δ for neuron in intermediate layer

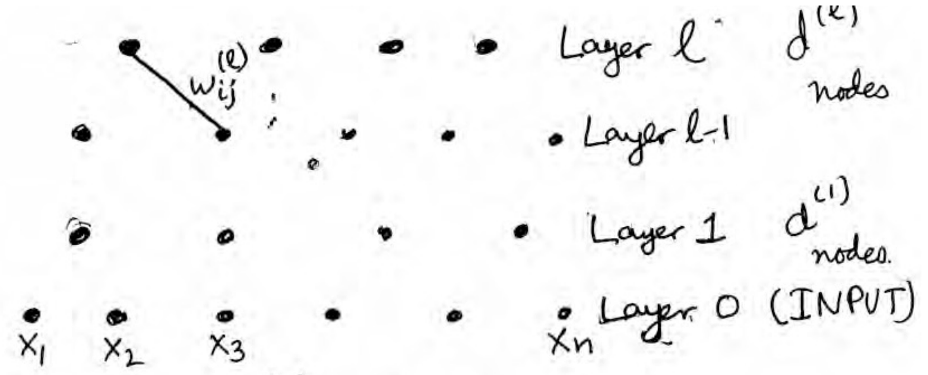
This is just the chain rule from calculus

$\boxed{\text{INDUCTION STEP}}$

$$\delta_i^{(l-1)} = \frac{\partial e(w)}{\partial s_i^{(l-1)}}$$

$$= \sum_j \frac{\partial e(w)}{\partial s_j^{(l)}} \times \frac{\partial s_j}{\partial x_i^{(l-1)}} \times \frac{\partial x_i^{(l-1)}}{\partial s_i^{(l-1)}}$$

Computing δ for neuron in intermediate layer



INDUCTION STE

$$x_j^{(\ell)} = g\left(\sum_{i=0}^{d^{(\ell-1)}} w_{ij}^{(\ell)} x_i^{(\ell-1)}\right) = g\left(s_j^{(\ell)}\right)$$

$s_j^{(\ell)}$ is the weighted input to node j in layer ℓ

$$\delta_i^{(\ell-1)} = \frac{\partial e(w)}{\partial s_i^{(l-1)}}$$

$$= \sum_j \frac{\partial e(w)}{\partial s_j^{(\ell)}} \times \frac{\partial s_j}{\partial x_i^{(\ell-1)}} \times \frac{\partial x_i^{(\ell-1)}}{\partial s_i^{(\ell-1)}}$$

Computing δ for neuron in intermediate layer
Deriving the basic step of "backpropagation"

INDUCTION STEP

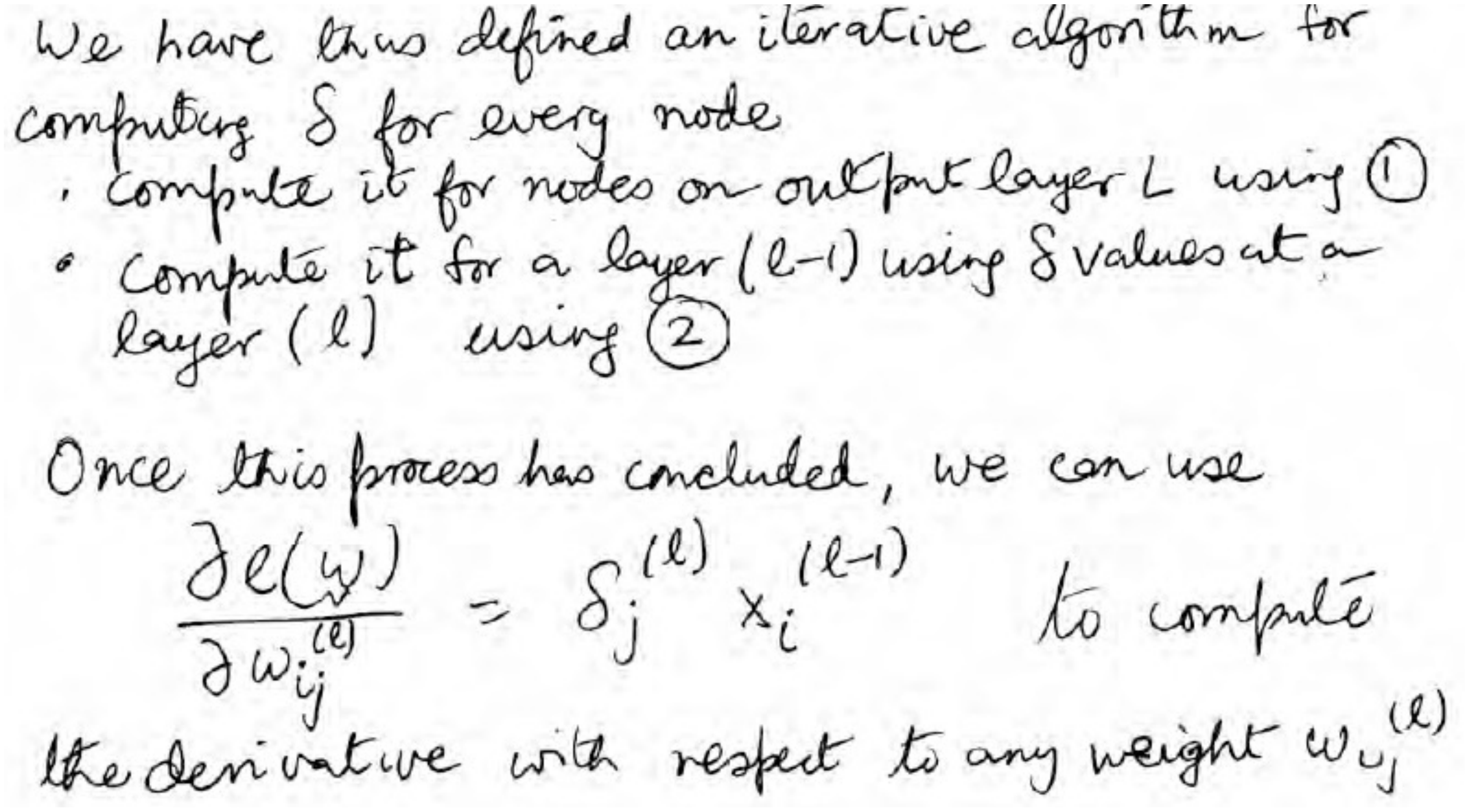
$$\delta_i^{(l-1)} = \frac{\partial e(w)}{\partial s_i^{(l-1)}}$$

$$= \sum_j \frac{\partial e(w)}{\partial s_j^{(l)}} \times \frac{\partial s_j}{\partial x_i^{(l-1)}} \times \frac{\partial x_i^{(l-1)}}{\partial s_i^{(l-1)}}$$

$$\delta_i^{(l-1)} = \sum_j \delta_j^{(l)} \, w_{ij}^{(l)} \, g'(s_i^{(l-1)})$$

$$= g'(s_i^{(l-1)}) \sum_j w_{ij}^{(l)} \delta_j^{(l)} \quad - - - ②$$

The full algorithm



We have thus defined an iterative algorithm for computing δ for every node

- Compute it for nodes on output layer L using ①
- Compute it for a layer $(\ell-1)$ using δ values at a layer (ℓ) using ②

Once this process has concluded, we can use

$$\frac{\partial e(w)}{\partial w_{ij}^{(\ell)}} = \delta_j^{(\ell)} \; x_i^{(\ell-1)}$$ to compute

the derivative with respect to any weight $w_{ij}^{(\ell)}$

Summary

- Note that we have computed all of the partial derivatives in one backward pass. The time complexity is proportional to number of weights
- We could have done this, one partial derivative at a time, using finite differences. This is a good way to debug your code for backprop. But it costs more, because you need to repeat the computation as many times as you have weights.
- The basic steps in backpropagation can be expressed as matrix multiplications, and GPUs are great at that. This is what enabled the scaling up of deep learning in the 2010s.

Basics of optimization

- Distinction between local and global minima
- Convex functions
- Gradient descent
- Gradient descent enables us to find local minima, but for a convex function, a local minimum is also a global minimum

$$\text{minimize}_w \quad f(w)$$

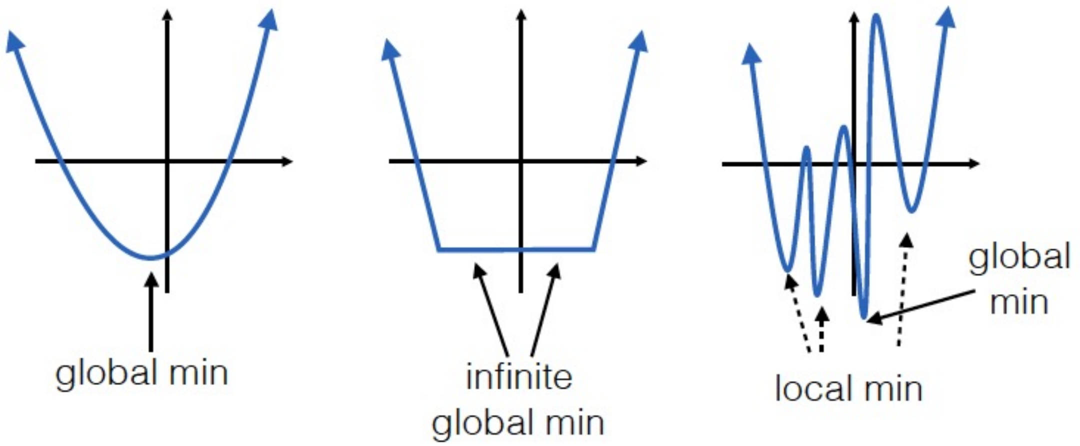
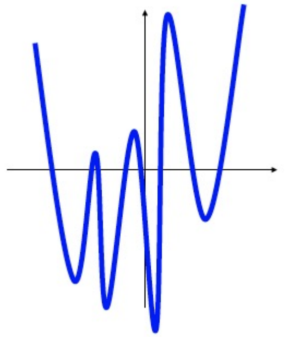


Figure 1: Examples of minima of functions. In the first illustration, there is a unique minimizer. In the second, there are an infinite number of minimizers, but all local minimizers are global minimizers. In the third example, there are many local minimizers that are not global minimizers.

Stationary points for functions of two variables



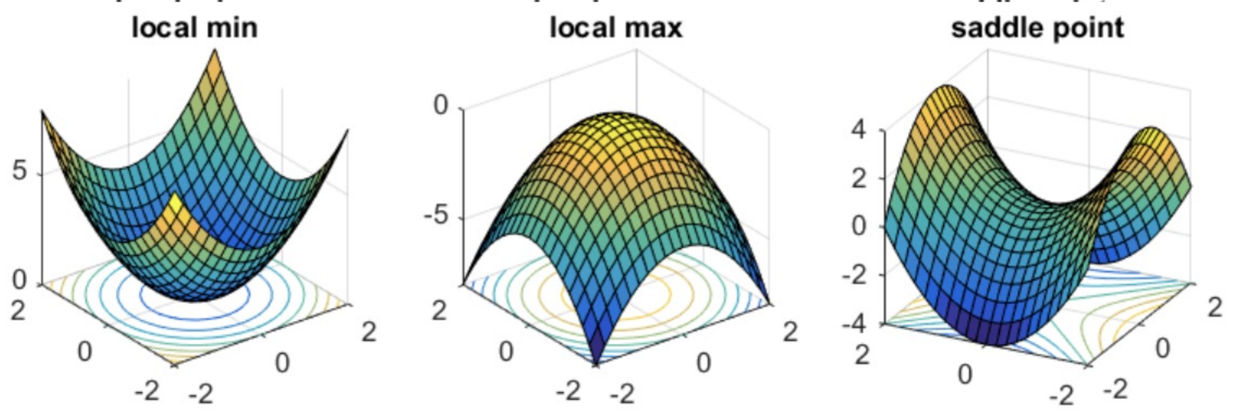
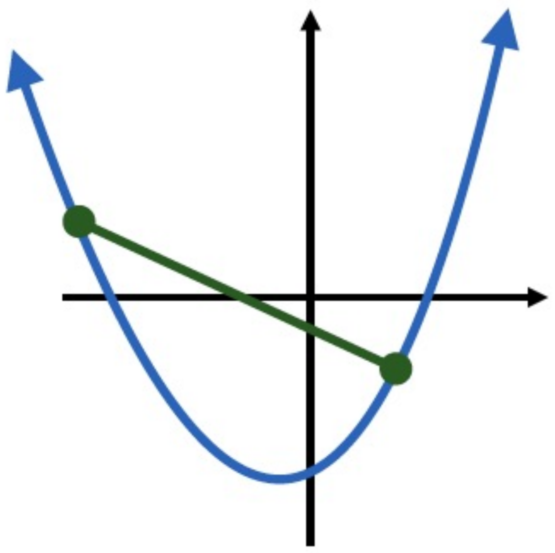
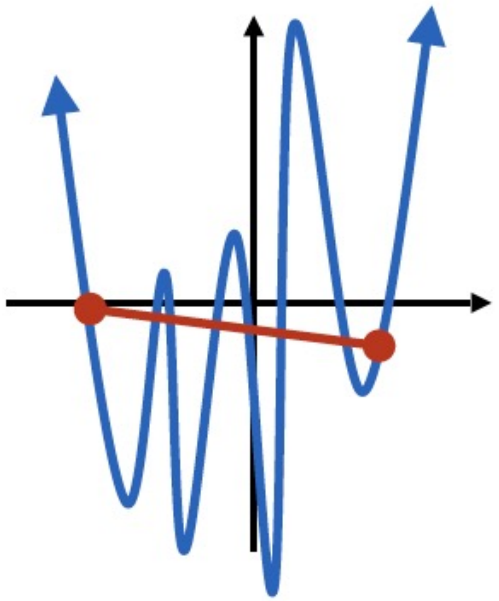


Figure 3: Examples of stationary points.


convex


nonconvex

- A function f is *convex* if for all w_1, w_2 in $\mathbb{R}^d$ and $t \in [0, 1]$,

$$f(tw_1 + (1-t)w_2) \leq tf(w_1) + (1-t)f(w_2)$$

Recall from multivariable calculus that the *gradient* of f is the vector

$$\nabla f(x) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ \frac{\partial f}{\partial x_2} \\ \vdots \\ \frac{\partial f}{\partial x_d} \end{bmatrix}.$$

The gradient is the direction which leads to a maximal increase of f. Similarly, the negative gradient is the direction of *steepest descent*. Gradient descent uses this fact to construct an algorithm: at every step, compute the gradient and follow that direction to minimize f.

Gradient Descent

The gradient descent method follows the simple algorithmic procedure:

1. Choose $x_0 \in \mathbb{R}^d$ and set $k = 0$
2. Choose $t_k > 0$ and set $x_{k+1} = x_k - t_k \nabla f(x_k)$ and $k = k + 1$,
3. Repeat 2 until converged.

Two ways to characterize a “descent direction”

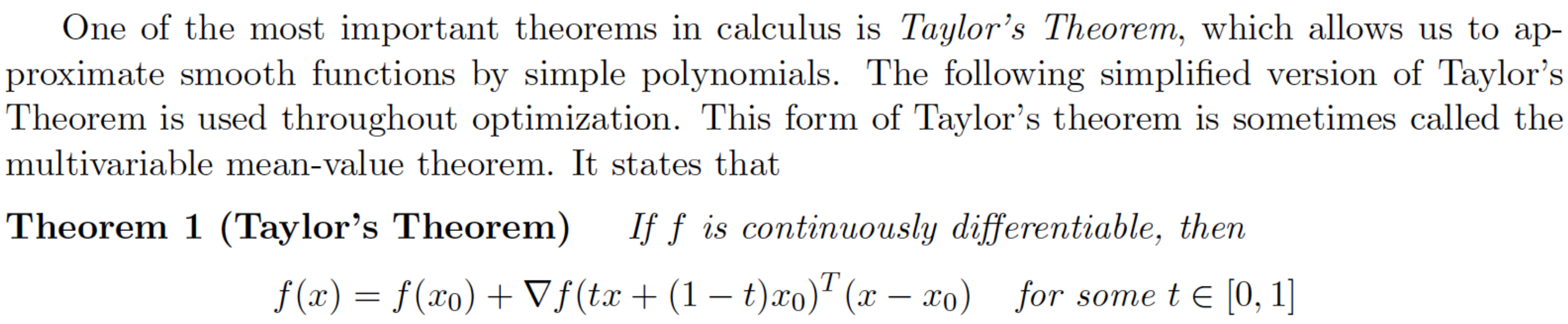
- v is a *descent direction* for f at x_0 if $f(x_0 + tv) < f(x_0)$ for some $t > 0$.

For continuously differentiable functions, it’s easy to tell if v is a descent direction: if $v^T \nabla f(x_0) < 0$ then v is a descent direction.

To see this note that by Taylor’s theorem, $f(x_0 + tv) = f(x_0) + t\nabla f(x_0 + \tilde{t}v)^T v$ for some $\tilde{t} \in [0, t]$. By continuity, if t is small, we’ll have $\nabla f(x_0 + \tilde{t}v)^T v < 0$. Therefore $f(x_0 + tv) < f(x_0)$ and v is a descent direction.

Note that among all directions with unit norm, the *steepest descent* possible is given when we move in the direction of the negative gradient.

Taylor’s theorem from multivariable calculus



One of the most important theorems in calculus is *Taylor’s Theorem*, which allows us to approximate smooth functions by simple polynomials. The following simplified version of Taylor’s Theorem is used throughout optimization. This form of Taylor’s theorem is sometimes called the multivariable mean-value theorem. It states that

Theorem 1 (Taylor’s Theorem) *If f is continuously differentiable, then*

$$f(x) = f(x_0) + \nabla f(tx + (1-t)x_0)^T (x - x_0) \quad \textit{for some } t \in [0,1]$$

$$f(x_0 + tv) = f(x_0) + t\nabla f(x_0 + \tilde{t}v)^T v$$