Foundations of Deep Learning

Lecture 01

Neural Networks: Introduction

Aurelien Lucchi

Fall 2024

Section 1

What's Deep Learning?

Coordinate change

Typical classification setting: Given some data $\mathcal{X} \in \mathbb{R}^d$ with some labels, e.g. ± 1 (black or white).

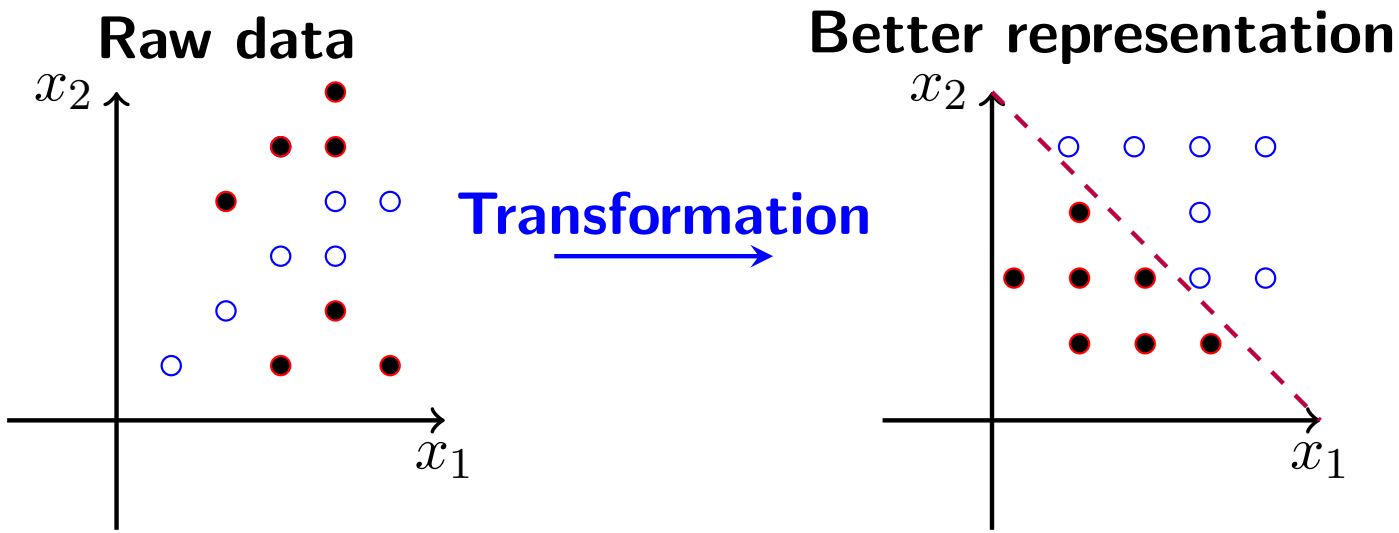
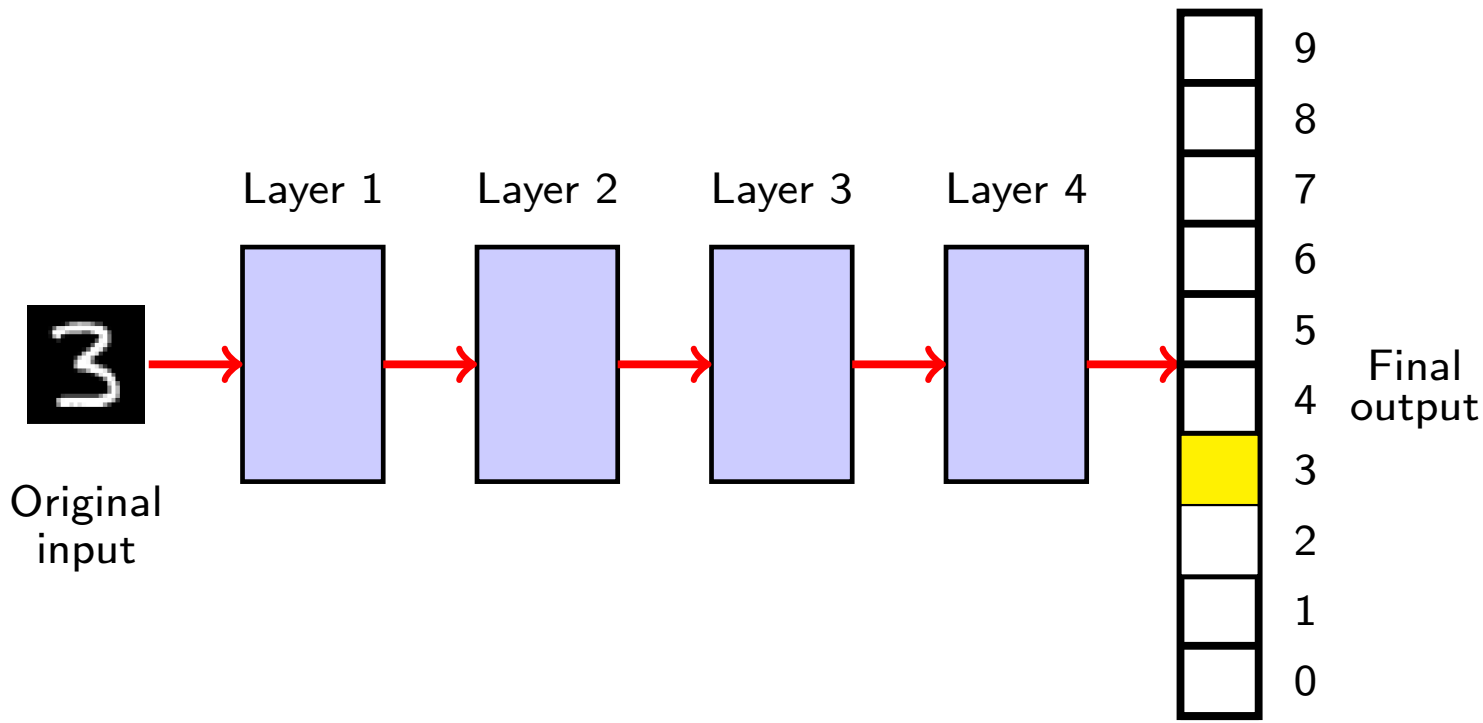


Figure: The left figure shows some raw data that is not linearly separable. A change of coordinate is applied to transform the data as shown in the right figure. The transformed data is linearly separable as shown by the dotted line that separates the two classes.

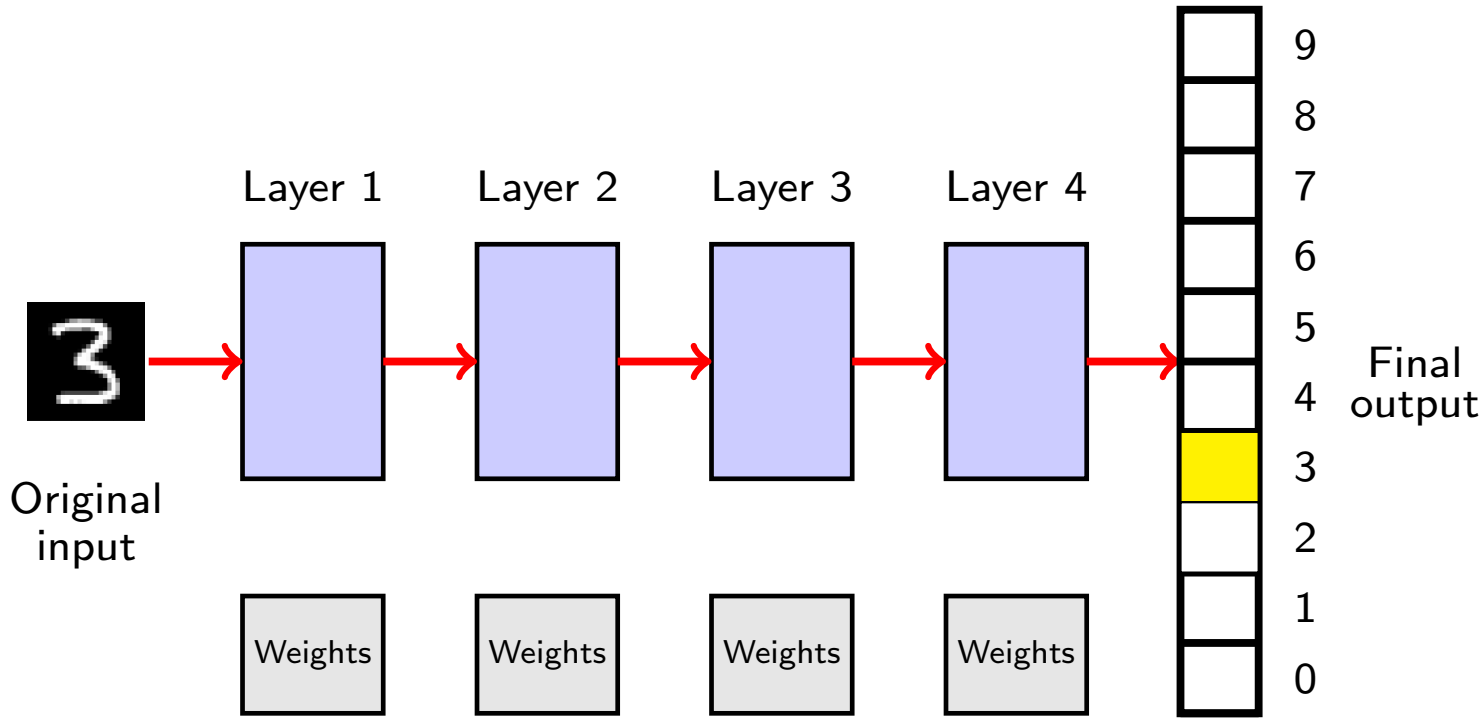
Neural Network

Nested representation: Better representations are obtained by passing the raw data through a series of layers that each apply a coordinate change to the data.



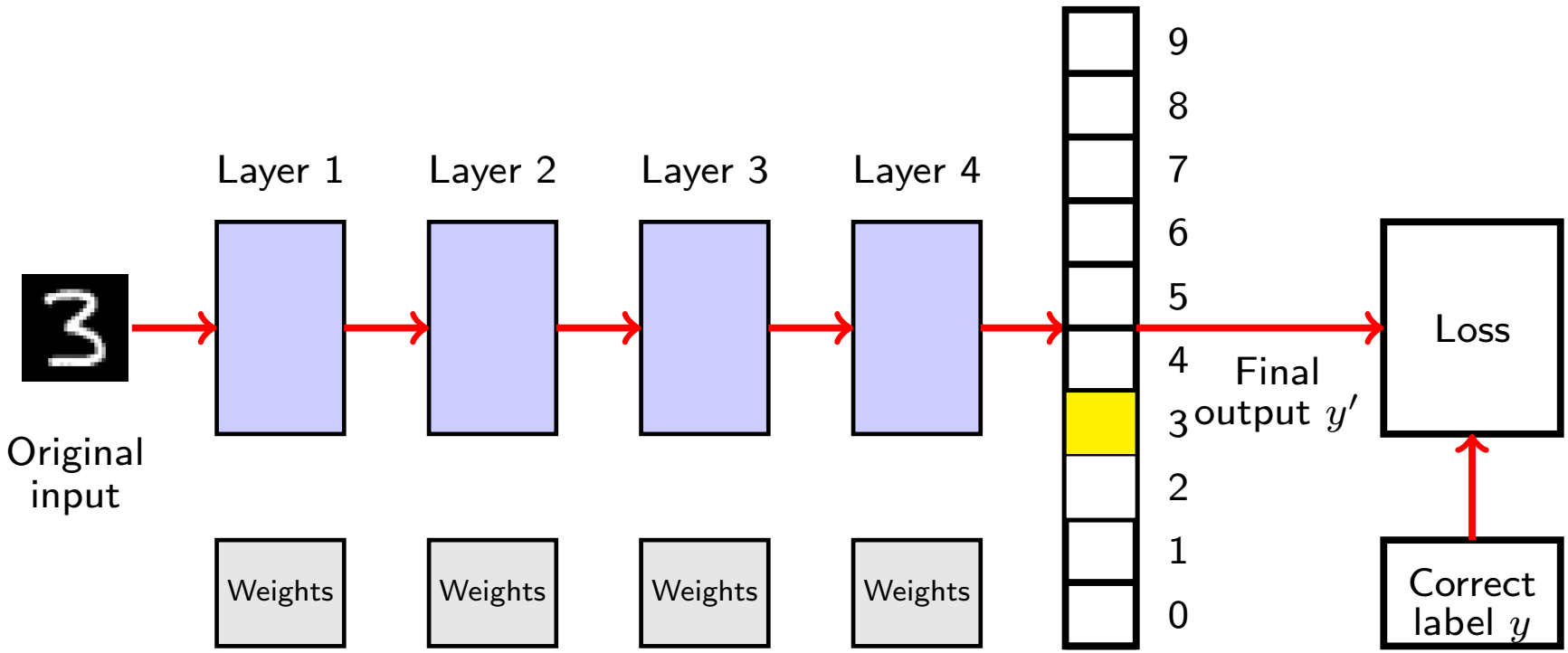
Layer Parameterization

Weights: Each layer has a set of weights that define the transformation applied to the layer input.



Loss function

The weights are learned by minimizing a loss function.



Optimizer

The loss is minimizing using an optimizer, e.g. gradient descent.

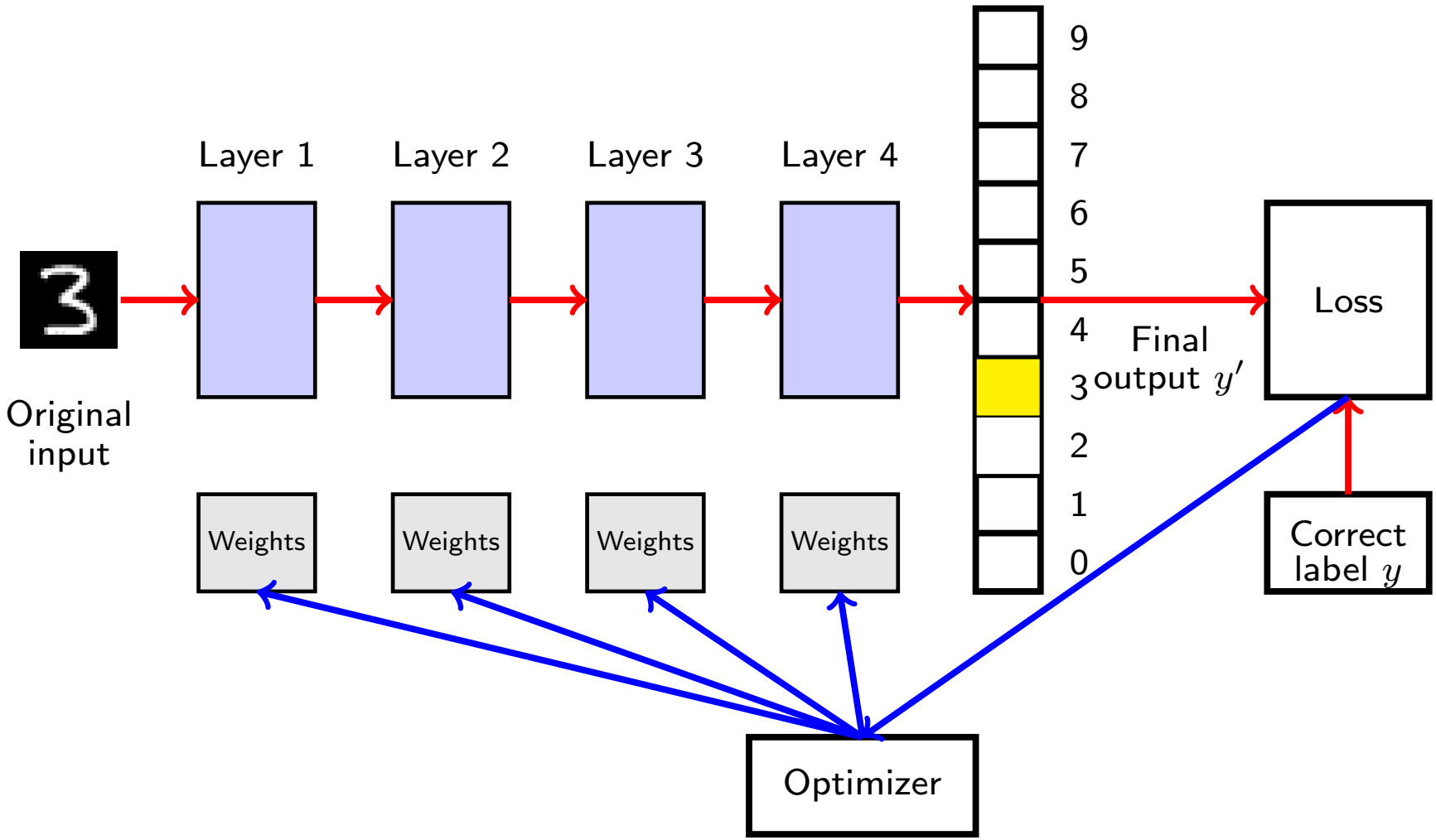
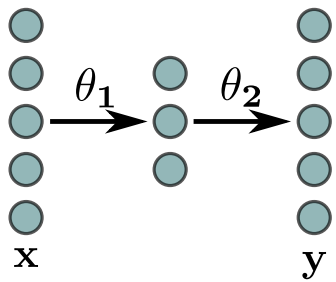


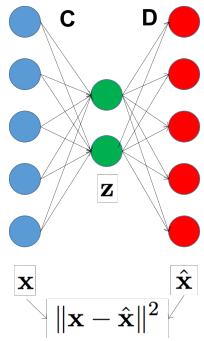
Figure: Optimizer in neural network training.
Figure based on "Deep Learning with Python" by Francois Chollet.

Syllabus for Remaining of this Lecture

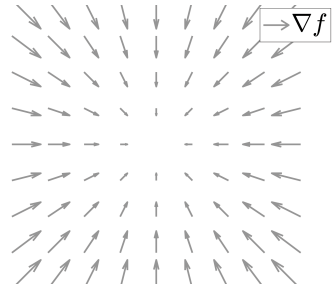
Part 1:



(a) Linear network



(b) Linear autoencoder



(c) Gradient dynamics

Part 2: Adding non-linearities.

Section 2

Deep Linear Networks

Subsection 1

Basic components: Linear Units

Abstract Unit

- Main idea: define a complex system from its elementary building blocks
- In the case of (deep) neural networks: **computational units** or neurons
- Complicated functions obtained through **nested composition**

Abstract Unit

- Main idea: define a complex system from its elementary building blocks
- In the case of (deep) neural networks: **computational units** or neurons
- Complicated functions obtained through **nested composition**
- Mathematically, such units are – at the highest level of abstraction – real-valued functions

$$\boxed{f : \mathbb{R}^n \times \mathbb{R}^d \to \mathbb{R}, \quad (\mathbf{x}, \boldsymbol{\theta}) \mapsto f(\mathbf{x}; \boldsymbol{\theta}) .} \qquad \text{(unit)}$$

Linear Functions

Linear Unit

$$g(\mathbf{x};\boldsymbol{\theta}) = \mathbf{x}\cdot\boldsymbol{\theta}\,, \quad \mathbf{x}\cdot\boldsymbol{\theta} \equiv \sum_{i=1}^{n} x_i\theta_i \qquad \text{(linear unit)}$$

(output = additively weighted combination of inputs)

Affine Unit

$$g(\mathbf{x};\boldsymbol{\theta},b) = \mathbf{x}\cdot\boldsymbol{\theta} + b = \begin{pmatrix}\mathbf{x}\\1\end{pmatrix}\cdot\begin{pmatrix}\boldsymbol{\theta}\\b\end{pmatrix}. \qquad \text{(affine unit)}$$

Linear Maps and Layers

Linear unit defines:

1. **Direction of change** via $\boldsymbol{\theta}/\|\boldsymbol{\theta}\|$
2. **Rate of change** via $\|\boldsymbol{\theta}\|$

Linear Maps and Layers

Linear unit defines:

1. **Direction of change** via $\boldsymbol{\theta}/\|\boldsymbol{\theta}\|$
2. **Rate of change** via $\|\boldsymbol{\theta}\|$

To create more expressive functions, we can use multiple units arranged in a **linear layer**, resulting in

$$L : \mathbb{R}^n \to \mathbb{R}^m, \quad L(\mathbf{x}; \Theta) = \Theta\mathbf{x}, \qquad \text{(linear map)}$$

$$\Theta = \begin{pmatrix} \boldsymbol{\theta}_1^\top \\ \dots \\ \boldsymbol{\theta}_m^\top \end{pmatrix} = \begin{pmatrix} - & \boldsymbol{\theta}_1^\top & - \\ & \dots & \\ - & \boldsymbol{\theta}_m^\top & - \end{pmatrix}, \quad \Theta \in \mathbb{R}^{m \times n}$$

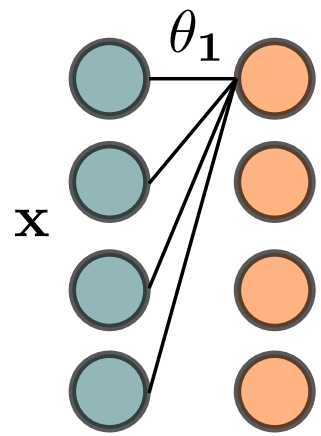
Note by convention that $\boldsymbol{\theta}_i$ is a column vector, so $\boldsymbol{\theta}_i^\top$ is a row vector.

Linear Maps and Layers

Note that the layer weight matrix Θ is related to the parameters associated with single units via

$$g_j(\mathbf{x}; \boldsymbol{\theta}_j) = \mathbf{x} \cdot \boldsymbol{\theta}_j, \quad \Theta = \begin{pmatrix} \boldsymbol{\theta}_1^\top \\ \dots \\ \boldsymbol{\theta}_m^\top \end{pmatrix}, \quad \text{so that} \quad \Theta\mathbf{x} = \begin{pmatrix} \mathbf{x} \cdot \boldsymbol{\theta}_1 \\ \dots \\ \mathbf{x} \cdot \boldsymbol{\theta}_m \end{pmatrix}.$$

- increasing layer width m: extract many features simultaneously
- every unit g_j can tune to a different direction $\boldsymbol{\theta}_j$
- however: **constant rate of change** is too restrictive (!)



Subsection 2

Linearity: A Deeper Look

Superposition Principle

- A more foundational way to define linearity from **superposition principle**:
- $f : \mathbb{R}^n \to \mathbb{R}^m$ is linear, if

$$f(\alpha \mathbf{x}) = \alpha f(\mathbf{x}) \quad \forall \mathbf{x} \in \mathbb{R}^n, \alpha \in \mathbb{R} \qquad \text{(homogeneity)}$$

$$\text{and} \quad f(\mathbf{x} + \mathbf{y}) = f(\mathbf{x}) + f(\mathbf{y}) \quad \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n . \qquad \text{(additivity)}$$

- f is **linear** iff

$$f(\alpha \mathbf{x} + \beta \mathbf{y}) = \alpha f(\mathbf{x}) + \beta f(\mathbf{y}), \quad \forall \mathbf{x}, \mathbf{y} \in \mathbb{R}^n, \quad \forall \alpha, \beta \in \mathbb{R}.$$

- **Affine functions**: ... not necessarily $f(\mathbf{0}) = \mathbf{0}$

Representation via Matrices

We can link the above abstract definition back to the parameterized view of linear maps via (weight) matrices Θ.

In the Euclidian basis $\{\mathbf{e}_1, \ldots, \mathbf{e}_n\}$, a vector $\mathbf{x}$ can (trivially) be written as $\mathbf{x} = \sum_i x_i \mathbf{e}_i$.

This directly leads to the following result.

f is linear $\iff$ $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$ for some unique $\mathbf{A} \in \mathbb{R}^{n \times m}$.

Compositionality

Deep compositional networks rely on the ability to **increase expressivity with compositional depth.**

Composing linear functions does **not** lead to any such gains:

Theorem 1
Let f, g be (matching) linear functions. Then $g \circ f$ is linear.

Compositionality

Deep compositional networks rely on the ability to **increase expressivity with compositional depth.**

Composing linear functions does **not** lead to any such gains:

Theorem 1
Let f, g be (matching) linear functions. Then $g \circ f$ is linear.

Proof.

$$
\begin{aligned}
(g \circ f)(\alpha \mathbf{x}) &= g(f(\alpha \mathbf{x})) = g(\alpha f(\mathbf{x})) = \alpha g(f(\mathbf{x})) = \alpha (g \circ f)(\mathbf{x}) \\
(g \circ f)(\mathbf{x} + \mathbf{y}) &= g(f(\mathbf{x}) + f(\mathbf{y})) = g(f(\mathbf{x})) + g(f(\mathbf{y})) \\
&= (g \circ f)(\mathbf{x}) + (g \circ f)(\mathbf{y})
\end{aligned}
$$

Compositionality with matrices

Let $f(\mathbf{x}) = \mathbf{A}\mathbf{x}$, $g(\mathbf{z}) = \mathbf{B}\mathbf{z}$, then

$$(g \circ f)(\mathbf{x}) = \mathbf{C}\mathbf{x}, \quad \mathbf{C} = \mathbf{B}\mathbf{A}$$

.

Composing linear functions amounts to multiplying matrices, the result of which will be yet another matrix.

Subsection 3

Linear Autoencoder

Autoencoder

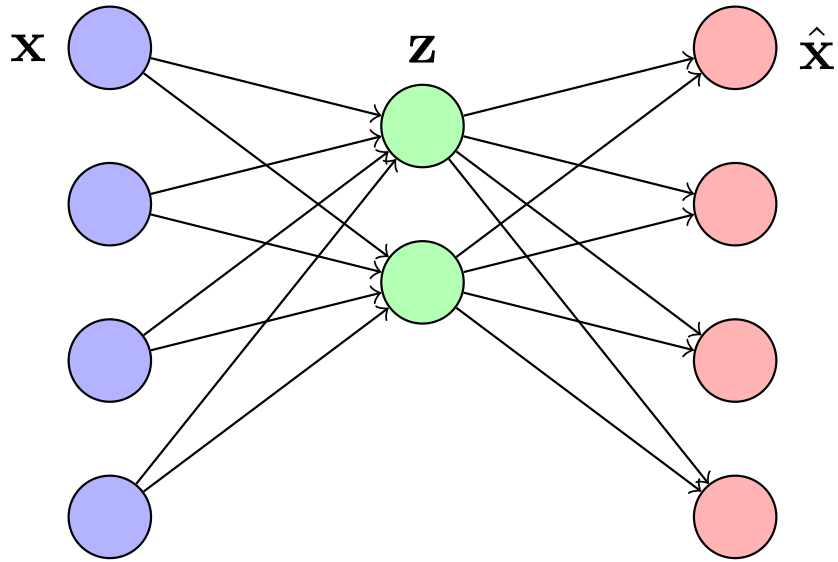
Compress data through a bottleneck layer

$$\boxed{\mathbf{x} \mapsto \mathbf{z} \mapsto \mathbf{y}, \quad \mathbf{z} = \mathbf{C}\mathbf{x},\ \mathbf{y} = \mathbf{D}\mathbf{z}, \quad \mathbf{C}, \mathbf{D}^\top \in \mathbb{R}^{m \times d},}$$

where

$$\boxed{m < d, \ \ \ell(\theta) = \frac{1}{2}\|\mathbf{x} - \mathbf{y}\|^2 = \frac{1}{2}\|\mathbf{x} - \mathbf{D}\mathbf{C}\mathbf{x}\|^2 \qquad \theta = (\mathbf{C}, \mathbf{D})}$$

Autoencoder: Diagram



Goal: minimize $\|\mathbf{x} - \hat{\mathbf{x}}\|^2$

How should you choose C and D to be "optimal"?

Reconstruction Error

Let $\boldsymbol{\theta} = (\mathbf{C}, \mathbf{D})$ and define a loss $L(\boldsymbol{\theta})$ to minimize to reconstruct the input at the output

$$\boxed{L(\boldsymbol{\theta}) = \frac{1}{2n} \sum_{i=1}^{n} \|\mathbf{x}_i - \mathbf{DCx}_i\|^2} \qquad \text{(loss)}$$

In matrix notation:

$$\boxed{\mathbf{X} = [\mathbf{x}_1 \dots \mathbf{x}_n] = \begin{pmatrix} | & & | \\ \mathbf{x}_1 & \cdots & \mathbf{x}_n \\ | & & | \end{pmatrix}, \ \mathbf{Y} = [\mathbf{y}_1 \dots \mathbf{y}_n] = \mathbf{DCX}}$$

$$\boxed{L(\boldsymbol{\theta}) = \frac{1}{2} \|\mathbf{X} - \mathbf{Y}\|_F^2} \qquad \text{(loss-frobenius)}$$

Autoencoder: Goal

- Essentially: learning the identity map, $\mathbf{DC} \approx \mathbf{I}$!
- But (1): trying to learn the identity with regard to a specific pattern distribution, represented by a sample
- But (2): need to find an efficient lossy compression scheme

Autoencoder: Gradients

Autoencoder: Gradients

Recall $\ell(\theta) = \frac{1}{2}\|\mathbf{x} - \underbrace{\mathbf{DCx}}_{=\mathbf{y}}\|^2 = \frac{1}{2}\sum_{j=1}^{d}(\mathbf{x} - \mathbf{DCx})_j^2$

Stochastic (per datapoint) gradients:

$$\Rightarrow \boxed{\begin{aligned} \frac{\partial \ell(\theta)}{\partial C_{ki}} &= \sum_{j=1}^{d}(y_j - x_j)\frac{\partial y_j}{\partial C_{ki}} = x_i \sum_{j=1}^{d} D_{jk}(y_j - x_j) \\ \frac{\partial \ell(\theta)}{\partial D_{jk}} &= \sum_{i=1}^{d}(y_i - x_i)\frac{\partial y_i}{\partial D_{jk}} = (y_j - x_j)\sum_{i=1}^{d} C_{ki}x_i \end{aligned}}$$

Autoencoder: Gradients

Alternatively

$$\boxed{\partial_{\mathbf{A}} f := \left(\frac{\partial f}{\partial A_{ij}}\right)_{ij}} \qquad \text{(gradient layout)}$$

One can then write elegantly

$$\Rightarrow \boxed{\begin{aligned} \frac{\partial \ell(\theta)}{\partial \mathbf{C}} &= \mathbf{D}^\top(\mathbf{y}-\mathbf{x})\mathbf{x}^\top \in \mathbb{R}^{m\times d}, \\ \frac{\partial \ell(\theta)}{\partial \mathbf{D}} &= (\mathbf{y}-\mathbf{x})\mathbf{x}^\top\mathbf{C}^\top \in \mathbb{R}^{d\times m} \end{aligned}}$$

Rank Constraint

$$\Rightarrow \boxed{\operatorname{rank}(\mathbf{DC}) \leq \min\{\operatorname{rank}(\mathbf{C}), \operatorname{rank}(\mathbf{D})\} \leq m < d\,.}$$

which implies

$$\Rightarrow \boxed{\mathbf{Y} = \mathbf{DCX}, \quad \operatorname{rank}(\mathbf{Y}) \leq m}$$

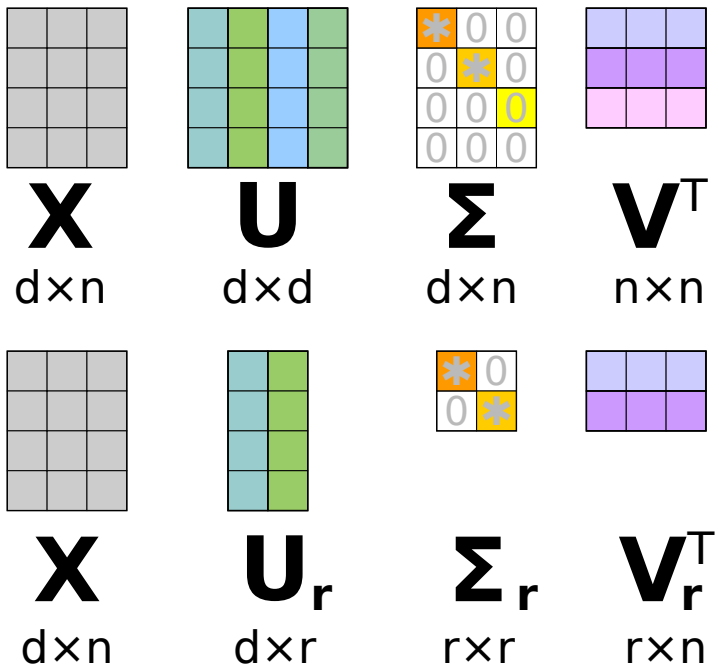
Singular Value Decomposition

Factorization of a matrix $\mathbf{X} \in \mathbb{R}^{d \times n}$ such that:

$$\boxed{\begin{aligned} \mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top, \quad & \mathbf{U} \in \mathbb{R}^{d \times d}, \mathbf{V} \in \mathbb{R}^{n \times n}, \text{ orthogonal}, \\ & \mathbf{\Sigma} = \operatorname{diag}(\sigma_1, \ldots, \sigma_{\min\{n,d\}}) \end{aligned}} \qquad \text{(SVD)}$$

Reduced/Truncated SVD

$$\mathbf{X}_r = \mathbf{U}_r \mathbf{\Sigma}_r \mathbf{V}_r^\top, \ \mathbf{\Sigma} = \mathrm{diag}(\sigma_1, \ldots, \sigma_r) \in \mathbb{R}^{r \times r}, \quad \text{(reduced SVD)}$$
$$\mathbf{U}_r = [\mathbf{u}_1 \ldots \mathbf{u}_r], \ \mathbf{V}_r = [\mathbf{v}_1 \ldots \mathbf{v}_r]$$



(Note that various terminologies are used in different ways: thin, compact, reduced)

Eckart-Young Theorem

Theorem 2

The rank-r matrix $\mathbf{X}_r$, obtained from the truncated singular value decomposition is such that:

$$\Rightarrow \boxed{\|\mathbf{X} - \mathbf{X}_r\|_F = \min_{\text{rank}(\mathbf{Y}) \leq r} \|\mathbf{X} - \mathbf{Y}\|_F} \qquad \text{(Eckart-Young)}$$

Optimal Linear Autoencoder

How should you choose C and D to be "optimal"?

By Theorem 2, choose $\mathbf{C} = \mathbf{U}_m^\top$, $\mathbf{D} = \mathbf{U}_m$:

$$\Rightarrow \boxed{\mathbf{DCX} = \mathbf{X}_m}$$

Proof.

$$\begin{aligned}
\mathbf{DCX} &= \mathbf{DCU\Sigma V}^\top \\
&= \mathbf{U}_m(\mathbf{U}_m^\top \mathbf{U})\mathbf{\Sigma V}^\top \\
&= \mathbf{U}_m \mathbf{I}_m \mathbf{\Sigma V}^\top \\
&= \mathbf{U}_m \mathbf{\Sigma}_m \mathbf{V}_m^\top \\
&= \mathbf{X}_m
\end{aligned}$$

□

Subsection 4

Deep Linear Gradients

Matrix Differentiation

Gradient of the matrix Frobenius norm: For a matrix $\mathbf{X} \in \mathbb{R}^{m \times n}$:

$$\nabla_{\mathbf{X}} \|\mathbf{X}\|_F^2 = 2\mathbf{X}$$

Derivation:

$$\nabla_{x_{ij}} \|\mathbf{X}\|_F^2 = \nabla_{x_{ij}} \left(\sum_{i=1}^{m} \sum_{j=1}^{n} x_{ij}^2 \right) = 2x_{ij}.$$

Gradient of the Trace of a Matrix Product

Gradient of the trace of a matrix product:

$$\nabla_{\mathbf{X}} \text{tr}(\mathbf{AX}) = \mathbf{A}^\top,$$
$$\nabla_{\mathbf{X}} \text{tr}(\mathbf{AX}^\top) = \mathbf{A}.$$

Gradient of Quadratic Form

Gradient of the quadratic form (symmetric $\mathbf{A}$): For a symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ and vector $\mathbf{x} \in \mathbb{R}^n$:

$$\nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = 2\mathbf{A}\mathbf{x}$$

Gradient of the quadratic form (non-symmetric $\mathbf{A}$): For a non-symmetric matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ and vector $\mathbf{x} \in \mathbb{R}^n$:

$$\nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = (\mathbf{A} + \mathbf{A}^\top)\mathbf{x}$$

Gradient of the quadratic form with respect to $\mathbf{A}$: For a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ and vector $\mathbf{x} \in \mathbb{R}^n$:

$$\nabla_{\mathbf{A}}(\mathbf{x}^\top \mathbf{A}\mathbf{x}) = \mathbf{x}\mathbf{x}^\top$$

Gradient of More Complex Quadratic Forms

Gradient of $\mathbf{x}^\top \mathbf{A}^\top \mathbf{A}\mathbf{x}$: For a matrix $\mathbf{A} \in \mathbb{R}^{n \times n}$ and vector $\mathbf{x} \in \mathbb{R}^n$:

$$\nabla_{\mathbf{x}}(\mathbf{x}^\top \mathbf{A}^\top \mathbf{A}\mathbf{x}) = 2\mathbf{A}^\top \mathbf{A}\mathbf{x}$$

Gradient of $(\mathbf{y} - \mathbf{A}\mathbf{x})^\top (\mathbf{y} - \mathbf{A}\mathbf{x})$ with respect to $\mathbf{A}$:

$$\frac{\partial}{\partial \mathbf{A}}(\mathbf{y} - \mathbf{A}\mathbf{x})^\top (\mathbf{y} - \mathbf{A}\mathbf{x}) = -2(\mathbf{y} - \mathbf{A}\mathbf{x})\mathbf{x}^\top.$$

Neural Network Gradients

Network Architecture: The input-output map is defined as:

$$\mathbf{y} = \mathbf{B}\mathbf{A}\mathbf{x}$$

where $\mathbf{x} \in \mathbb{R}^d, \mathbf{y} \in \mathbb{R}^k, \mathbf{A} \in \mathbb{R}^{m \times d}, \mathbf{B} \in \mathbb{R}^{k \times m}$.

Loss Function for the Neural Network

Training Objective: Given a set of n training examples $(\mathbf{x}_i, \mathbf{y}_i)_{i=1}^n$, the loss function is:

$$L(\mathbf{B}, \mathbf{A}) = \frac{1}{2} \sum_{i=1}^{n} \|\mathbf{y}_i - \mathbf{B}\mathbf{A}\mathbf{x}_i\|^2.$$

Gradient of the Loss w.r.t $\mathbf{A}$

Gradient of the loss with respect to $\mathbf{A}$:

$$\frac{\partial L(\mathbf{B}, \mathbf{A})}{\partial \mathbf{A}} = -\mathbf{B}^\top(\Sigma_{xy} - \mathbf{B}\mathbf{A}\Sigma_{xx}),$$

where Σ_{xx} and Σ_{xy} are input and input-output correlation matrices.

Derivation (I/III)

In order to calculate the gradients, let's first expand the above expression:

$$
\begin{aligned}
L(\mathbf{B}, \mathbf{A}) &= \frac{1}{2} \sum_{i=1}^{n} \|\mathbf{y}_i - \mathbf{B}\mathbf{A}\mathbf{x}_i\|^2 \\
&= \frac{1}{2} \sum_{i=1}^{n} (\mathbf{y}_i - \mathbf{B}\mathbf{A}\mathbf{x}_i)^\top (\mathbf{y}_i - \mathbf{B}\mathbf{A}\mathbf{x}_i) \\
&= \frac{1}{2} \sum_{i=1}^{n} \mathbf{y}_i^\top \mathbf{y}_i - \mathbf{y}_i^\top \mathbf{B}\mathbf{A}\mathbf{x}_i - (\mathbf{B}\mathbf{A}\mathbf{x}_i)^\top \mathbf{y}_i + (\mathbf{B}\mathbf{A}\mathbf{x}_i)^\top (\mathbf{B}\mathbf{A}\mathbf{x}_i) \\
&= \frac{1}{2} \sum_{i=1}^{n} \mathbf{y}_i^\top \mathbf{y}_i - \mathbf{y}_i^\top \mathbf{B}\mathbf{A}\mathbf{x}_i - \mathbf{x}_i^\top \mathbf{A}^\top \mathbf{B}^\top \mathbf{y}_i + \mathbf{x}_i^\top \mathbf{A}^\top \mathbf{B}^\top \mathbf{B}\mathbf{A}\mathbf{x}_i.
\end{aligned}
$$

Derivation (II/III)

Recall the following equalities:

$$\frac{\partial \mathbf{a}^\top \mathbf{X}\mathbf{b}}{\partial \mathbf{X}} = \mathbf{a}\mathbf{b}^\top$$
$$\frac{\partial \mathbf{a}^\top \mathbf{X}^\top \mathbf{D}\mathbf{X}\mathbf{b}}{\partial \mathbf{X}} = \mathbf{D}^\top \mathbf{X}\mathbf{a}\mathbf{b}^\top + \mathbf{D}\mathbf{X}\mathbf{b}\mathbf{a}^\top,$$

therefore we have

$$\frac{\partial \mathbf{y}_i^\top \mathbf{B}\mathbf{A}\mathbf{x}_i}{\partial \mathbf{A}} = \mathbf{B}^\top \mathbf{y}_i \mathbf{x}_i^\top$$
$$\frac{\partial \mathbf{x}_i^\top \mathbf{A}^\top \mathbf{B}^\top \mathbf{y}_i}{\partial \mathbf{A}} = \mathbf{x}_i \mathbf{y}_i^\top \mathbf{B}$$
$$\begin{aligned}\frac{\partial \mathbf{x}_i^\top \mathbf{A}^\top \mathbf{B}^\top \mathbf{B}\mathbf{A}\mathbf{x}_i}{\partial \mathbf{A}} &= \mathbf{B}\mathbf{B}^\top \mathbf{A}\mathbf{x}_i \mathbf{x}_i^\top + \mathbf{B}^\top \mathbf{B}\mathbf{A}\mathbf{x}_i \mathbf{x}_i^\top \\ &= 2\mathbf{B}\mathbf{B}^\top \mathbf{A}\mathbf{x}_i \mathbf{x}_i^\top.\end{aligned}$$

Derivation (III/III)

Note that the first two derivatives are equal (since $\mathbf{y}_i^\top \mathbf{B}\mathbf{A}\mathbf{x}_i = \mathbf{x}_i^\top \mathbf{A}^\top \mathbf{B}^\top \mathbf{y}_i$), thus

$$\begin{aligned}\frac{\partial L(\mathbf{B},\mathbf{A})}{\partial \mathbf{A}} &= -\sum_{i=1}^{n} \mathbf{B}^\top \mathbf{y}_i \mathbf{x}_i^\top + \sum_{i=1}^{n} \mathbf{B}^\top \mathbf{B}\mathbf{A}\mathbf{x}_i \mathbf{x}_i^\top \\ &= -\mathbf{B}^\top (\Sigma_{xy} - \mathbf{B}\mathbf{A}\Sigma_{xx}),\end{aligned}$$

where $\Sigma_{xx} = \sum_{i=1}^{n} \mathbf{x}_i \mathbf{x}_i^\top$ is the input correlation matrix and $\Sigma_{xy} = \sum_{i=1}^{n} \mathbf{y}_i \mathbf{x}_i^\top$ is the input-output correlation matrix.

Gradient of the Loss w.r.t B

Gradient of the loss with respect to B:

$$\frac{\partial L}{\partial \mathbf{B}} = -(\Sigma_{xy} - \mathbf{B}\mathbf{A}\Sigma_{xx})\mathbf{A}^\top.$$

Section 4

Non-linear Activations

Subsection 1

Ridge Functions

Definition and Level Sets

Ridge functions: generalize linear functions in just the right way to allow for powerful compositionality.

Simple idea: compose linear (or affine) function with a non-linear (continuous) function ϕ, also called **activation function**

$$\boxed{f = \phi \circ g, \quad f(\mathbf{x}; \boldsymbol{\theta}) = \phi(\mathbf{x} \cdot \boldsymbol{\theta})} \qquad \text{(ridge unit)}$$

Rate of Change

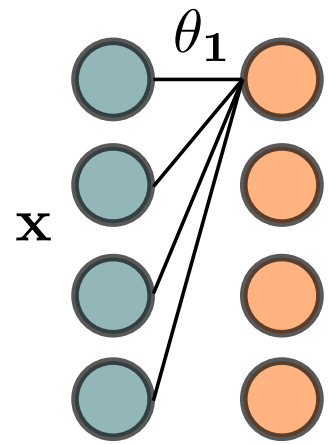
Ridge functions have a **variable rate of change**, governed by ϕ or rather ϕ' (in case ϕ is differentiable).

$$\Rightarrow \boxed{\|\nabla_{\mathbf{x}} f(\mathbf{x};\boldsymbol{\theta})\| = |\phi'(\mathbf{x}\cdot\boldsymbol{\theta})|\ \|\boldsymbol{\theta}\|}$$

Ridge units can easily be arranged in layers. We will then write, extending ϕ to vectors,

$$\boxed{F(\mathbf{x};\Theta) = \phi(\Theta\mathbf{x}), \quad \phi(\mathbf{z}) = \begin{pmatrix} \phi(z_1) \\ \dots \\ \phi(z_m) \end{pmatrix}}$$

(ridge layer)



Parameter Gradients

Parameter update directions for ridge functions are always in the **direction of the input vector**.

For any loss function ℓ, the chain rule yields

$$\Rightarrow \boxed{\nabla_{\boldsymbol{\theta}}(\ell \circ \phi)(\mathbf{x} \cdot \boldsymbol{\theta}) = (\ell \circ \phi)'(\mathbf{x} \cdot \boldsymbol{\theta})\, \mathbf{x}\,.}$$

This is an inheritance of the way ridge functions generalize linear functions.

Subsection 2

Threshold Units

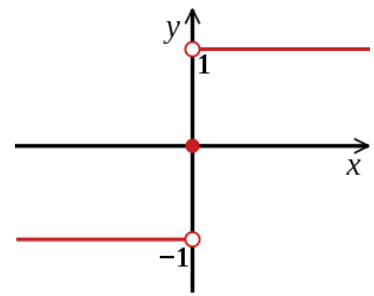
Heavyside or Sign Units

A **linear threshold unit** is a function that uses the Heavyside or sign function as activation and produces Boolean outputs.

$$f^H(\mathbf{x};\boldsymbol{\theta}) = H(\mathbf{x}\cdot\boldsymbol{\theta}),\ H(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{else} \end{cases}$$ (threshold unit)

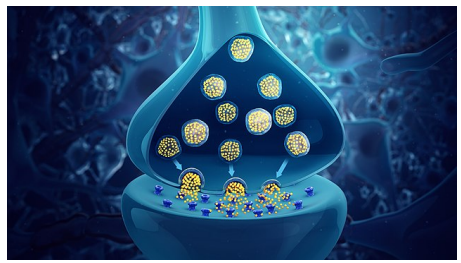
Equivalently, one can also map to $\{-1, 1\}$,

$$f^{\pm}(\mathbf{x};\boldsymbol{\theta}) = \text{sign}(\mathbf{x}\cdot\boldsymbol{\theta})\,.$$ (sign unit)



Learning with Threshold Units

Threshold units $=$ **cartoon version of a biological neuron**.



$\Longrightarrow$ If activation exceeds a threshold, the unit 'fires'.

Major drawback: H has a jump at 0

$\Longrightarrow$ no derivative information to guide the adaptation of the parameters

$\Longrightarrow$ rules out gradient-based methods.

Subsection 3

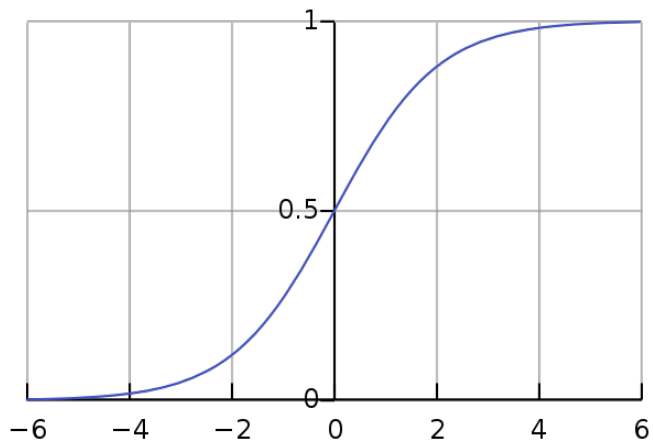
Sigmoid Units

Logistic Unit

Goal: retain some characteristics of the threshold unit, in particular the limits $f(\mathbf{x}, \boldsymbol{\theta}) \to (0, 1)$ as $\mathbf{x} \cdot \boldsymbol{\theta} \to (-\infty, \infty)$, but to **define a smooth, monotone function**.

One such choice is the **logistic Unit**

$$\boxed{\sigma(\mathbf{x} \cdot \boldsymbol{\theta}) = \frac{1}{1 + \exp[-\mathbf{x} \cdot \boldsymbol{\theta}]}.} \qquad \text{(logistic unit)}$$



Logistic Unit: Derivatives

Proposition 3

The logistic function has derivative

$$\Rightarrow \boxed{\sigma'(z) = \sigma(z)(1 - \sigma(z)) = \sigma(z)\sigma(-z)\,.}$$

Proof: With the quotient rule,

$$\begin{aligned}\sigma'(z) &= e^{-z}(1 + e^{-z})^{-2} = \frac{1}{1 + e^{-z}} \frac{e^{-z}}{1 + e^{-z}} \\ &= \sigma(z)(1 - \sigma(z))\,.\end{aligned}$$

This implies that all higher-order derivatives are polynomials in σ and hence σ is smooth:

$$\Rightarrow \boxed{\sigma \in C^\infty} \tag{1}$$

Hyperbolic Tangent

Another popular choice of activation function is the hyperbolic tangent, which is **simple transformation of the logistic function**

$$\Rightarrow \boxed{\tanh(z) := \frac{e^z - e^{-z}}{e^z + e^{-z}} = 2\sigma(2z) - 1,}$$

where the identity follows from:

$$2\sigma(2z) - 1 = \frac{2}{1 + e^{-2z}} - \frac{1 + e^{-2z}}{1 + e^{-2z}} = \frac{1 - e^{-2z}}{1 + e^{-2z}} = \frac{e^z - e^{-z}}{e^z + e^{-z}}\,.$$

As the range of the $\tanh$ function $R = (-1; 1)$ is symmetric around zero, it is sometimes preferred.

Inverse of Logistic Function

Finally, we also derive the inverse of the logistic function for later use

$$\Rightarrow \boxed{\sigma^{-1}(t) = \ln \frac{t}{1-t},} \qquad \text{(inverse logistic)}$$

The validity of this formula can be checked via:

$$\sigma\left(\ln \frac{t}{1-t}\right) = \frac{1}{1+\frac{1-t}{t}} = t\,.$$

Subsection 4

Softmax

Softmax: Definition

A generalization of logistic units for **multiclass classification** (k classes).

Defines class conditional probabilities via

$$\boxed{\sigma_i^{\max}(\mathbf{x};\Theta) = \frac{\exp[\mathbf{x}\cdot\boldsymbol{\theta}_i]}{\sum_{j=1}^{k}\exp[\mathbf{x}\cdot\boldsymbol{\theta}_j]}, \quad \Theta = [\boldsymbol{\theta}_1,\ldots,\boldsymbol{\theta}_k]} \quad \text{(softmax)}$$

Properties:

- Non-negative
- Normalized: $\sum_{i=1}^{k}\sigma_i^{\max}(\mathbf{x};\Theta) = 1$

Softmax: Comments

1) Over-parameterization: one can subtract any (constant) vector
$\boldsymbol{\theta}_i' = \boldsymbol{\theta}_i + \Delta\boldsymbol{\theta} \qquad \forall i$

$\Longrightarrow$ can be avoided by setting $\boldsymbol{\theta}_k = \mathbf{0}$.

2) Sigmoid unit recovered for $k = 2$

$\Rightarrow$ soft-max with $\boldsymbol{\theta}_1, \boldsymbol{\theta}_2 \quad \Longleftrightarrow \quad$ sigmoid with $\boldsymbol{\theta} = \boldsymbol{\theta}_1 - \boldsymbol{\theta}_2$

$$\sigma_1^{\text{max}}(\mathbf{x}) = \frac{e^{\mathbf{x}\cdot\boldsymbol{\theta}_1}}{e^{\mathbf{x}\cdot\boldsymbol{\theta}_1} + e^{\mathbf{x}\cdot\boldsymbol{\theta}_2}} = \frac{e^{\mathbf{x}\cdot(\boldsymbol{\theta}_1-\boldsymbol{\theta}_1)}}{e^{x\cdot(\boldsymbol{\theta}_1-\boldsymbol{\theta}_1)} + e^{\mathbf{x}\cdot(\boldsymbol{\theta}_2-\boldsymbol{\theta}_1)}}$$
$$= \frac{1}{1 + e^{-\mathbf{x}\cdot\boldsymbol{\theta}}} = \sigma(\mathbf{x})$$

Softmax + Cross-entropy

Softmax is typically coupled with cross entropy loss.

Encode class information in one-hot vector, $\mathbf{y} \in \{\mathbf{e}_1, \ldots, \mathbf{e}_k\}$, then:

$$\boxed{\ell(\mathbf{x}, \mathbf{y}; \Theta) = -\mathbf{y} \cdot \ln \sigma^{\mathsf{max}}(\mathbf{x}; \Theta)}$$

Alternatively: separate out the normalization

$$\ell(\mathbf{x}, \mathbf{y}; \Theta) = -\sum_{i=1}^{k} y_i \, \mathbf{x} \cdot \boldsymbol{\theta}_i + \ln \left(\sum_{i=1}^{k} \exp[\mathbf{x} \cdot \boldsymbol{\theta}_i] \right) .$$

leading to the gradient formula

$$\Rightarrow \boxed{\nabla_{\boldsymbol{\theta}_i} \ell(\mathbf{x}, \mathbf{y}; \Theta) = (\sigma_i^{\mathsf{max}} - y_i)\mathbf{x}}$$

Subsection 5

Multilayer Perceptron

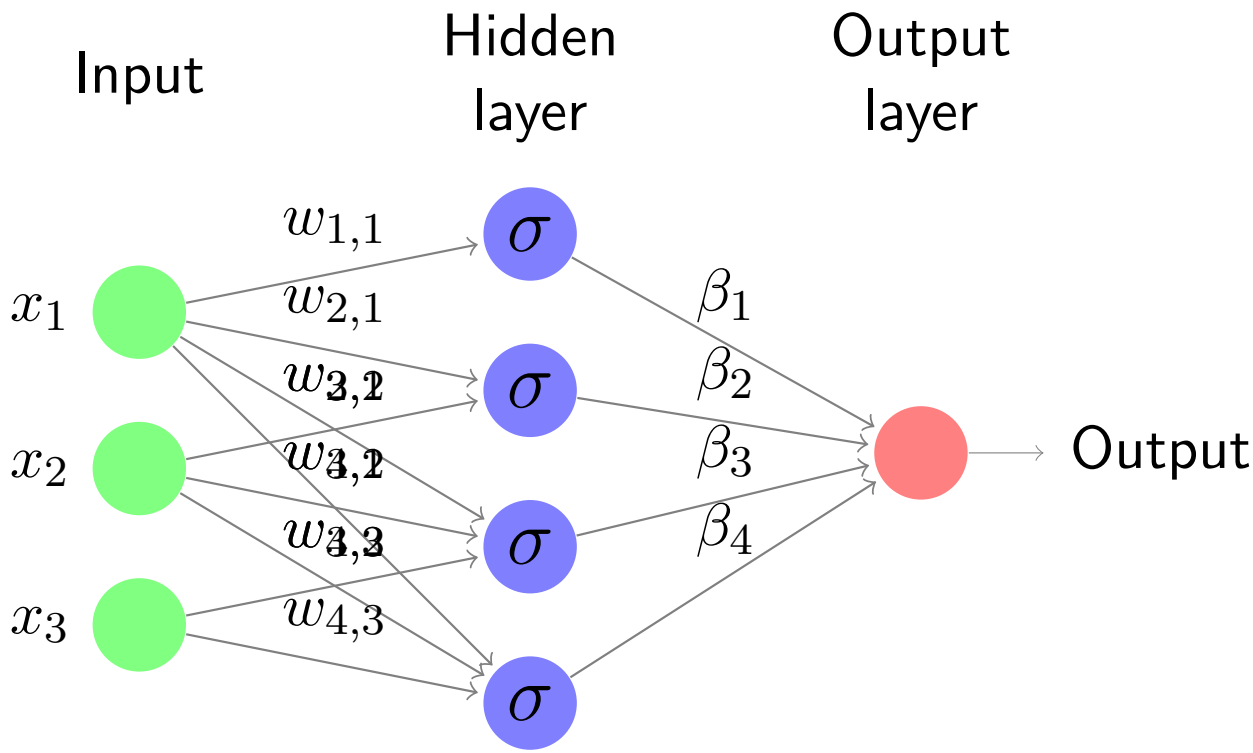
MLP: Architecture

- Three-layer reference architecture with
  - n inputs
  - one hidden layer of m sigmoid units
  - one real-valued output (could be more)

$$f(\mathbf{x}) := f(\mathbf{x}; \boldsymbol{\beta}, \mathbf{w}) = \sum_{j=1}^{m} \frac{\beta_j}{1 + \exp[-\mathbf{w}_j \cdot \mathbf{x}]}.$$

- Each unit computes a weighted combination of inputs and applies a logistic activation function.
- These activations (in range $(0; 1)$) are then summed up with weights $\boldsymbol{\beta}$.

MLP: Sketch


Input
Hidden layer
Output layer
x_1
x_2
x_3
$w_{1,1}$
$w_{2,1}$
$w_{3,1}$
$w_{4,1}$
$w_{3,2}$
$w_{4,2}$
$w_{3,3}$
$w_{4,3}$
σ
σ
σ
σ
β_1
β_2
β_3
β_4
Output

MLP: Learning

$$\frac{\partial \frac{1}{2}(f(\mathbf{x}) - y)^2}{\partial \beta_j} = \frac{f(\mathbf{x}) - y}{1 + \exp[-\mathbf{w}_j \cdot \mathbf{x}]},$$

$$\frac{\partial \frac{1}{2}(f(\mathbf{x}) - y)^2}{\partial w_{ji}} = \frac{f(\mathbf{x}) - y}{1 + \exp[-\mathbf{w}_j \cdot \mathbf{x}]} \cdot \frac{\beta_j x_i}{1 + \exp[\mathbf{w}_j \cdot \mathbf{x}]}.$$

- Forward pass: computes hidden activities and output
- Re-combined with a few more operations to get derivatives
- Updates via stochastic gradient descent (SGD)

$$w \leftarrow w - \eta \frac{\partial \frac{1}{2}(f(\mathbf{x}) - y)^2}{\partial w}, \quad w \in \{\beta_j, w_{ji}\},$$

with some step size $\eta > 0$.

Code: Forward Pass

```
from scipy.special import expit #sigmoid
class MLP3():
  def __init__(self,n,m):
    self.theta
      = np.random.uniform(-1/n,1/n,(m,n)) # mxn
    self.beta
      = np.random.uniform(-1/m,1/m,(1,m)) # 1xm

def forward(self, x):
  net_in   = np.dot(self.theta,x)         # mxn*nxs=mxs
  f['hid'] = expit(net_in)                # mxs
  f['out'] = np.dot(self.beta,f['hid']) # 1xm*mxs=1xs
  return f
```

Code: Gradient Pass

```
def gradient(self, f, x, y):

  # beta gradients
  delta     = (f['out']-y)                  # 1xs
  g['bet'] = np.dot(delta,f['hid'].T) # 1xs*sxm=1xm

  # theta gradients
  hid       = f['hid']              # hidden activities
  hid_sv    = hid-hid*hid           # sensitivities
  outer     = np.outer(self.beta.T,delta)# mx1*1xs=mxs
  g['tet'] = np.dot((outer*hid_sv),x.T) # mxs*sxn=mxn

  return g
```

Subsection 6

Backpropagation

Introduction to Backpropagation

- Backpropagation is a supervised learning algorithm used for training artificial neural networks.
- It calculates the gradient of the loss function with respect to each weight by the chain rule, iterating backward from the output to the input layer.
- The main goal is to minimize the error by adjusting the weights.

Example Network Structure

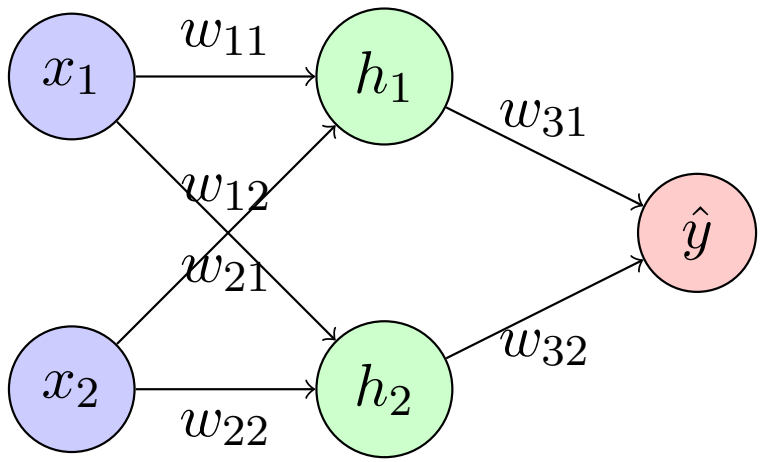


Figure: A simple neural network with input layer, one hidden layer, and output layer.

Forward Pass

- Calculate the input to the hidden layer neurons:

$$\text{net}_{h_1} = w_{11}x_1 + w_{21}x_2$$

$$\text{net}_{h_2} = w_{12}x_1 + w_{22}x_2$$

- Calculate the output of the hidden layer neurons using an activation function σ:

$$h_1 = \sigma(\text{net}_{h_1})$$

$$h_2 = \sigma(\text{net}_{h_2})$$

- Calculate the input to the output neuron:

$$\text{net}_{o1} = w_{31}h_1 + w_{32}h_2$$

- Calculate the output neuron:

$$\hat{y} = \sigma(\text{net}_{o1})$$

Loss Calculation

- Compute the loss (error) using a loss function, e.g., Mean Squared Error (MSE):

$$L(y, \hat{y}) = \frac{1}{2}(y - \hat{y})^2$$

- Here, y is the true label, and $\hat{y}$ is the predicted output.

Gradient of the Loss with Respect to Output

We start by computing the gradient of the loss with respect to the output:

$$\frac{\partial L}{\partial \hat{y}} = \hat{y} - y.$$

Gradient with Respect to Weights (Hidden to Output)

Then, we compute the gradient of the loss with respect to the weights connecting the hidden layer to the output layer.

$$\frac{\partial L}{\partial w_{31}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \text{net}_{o1}} \cdot \frac{\partial \text{net}_{o1}}{\partial w_{31}} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot h_1$$

$$\frac{\partial L}{\partial w_{32}} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \text{net}_{o1}} \cdot \frac{\partial \text{net}_{o1}}{\partial w_{32}} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot h_2$$

Gradient with Respect to Hidden Layer Neurons

The next step is to compute the gradient of the loss with respect to the hidden layer neurons:

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \text{net}_{o1}} \cdot \frac{\partial \text{net}_{o1}}{\partial h_1} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot w_{31}$$

$$\frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial \text{net}_{o1}} \cdot \frac{\partial \text{net}_{o1}}{\partial h_2} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot w_{32}$$

Gradient with Respect to Weights (Input to Hidden)

We then compute the gradient of the loss with respect to the weights connecting the input layer to the hidden layer. This also involves the derivative of the activation function σ applied to net_{h_1} and net_{h_2}:

$$\frac{\partial L}{\partial w_{11}} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial \text{net}_{h_1}} \cdot \frac{\partial \text{net}_{h_1}}{\partial w_{11}} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot w_{31} \cdot \sigma'(\text{net}_{h_1}) \cdot x_1$$

$$\frac{\partial L}{\partial w_{21}} = \frac{\partial L}{\partial h_1} \cdot \frac{\partial h_1}{\partial \text{net}_{h_1}} \cdot \frac{\partial \text{net}_{h_1}}{\partial w_{21}} = (\hat{y} - y) \cdot \sigma'(\text{net}_{o1}) \cdot w_{31} \cdot \sigma'(\text{net}_{h_1}) \cdot x_2$$

Illustration Backpropagation

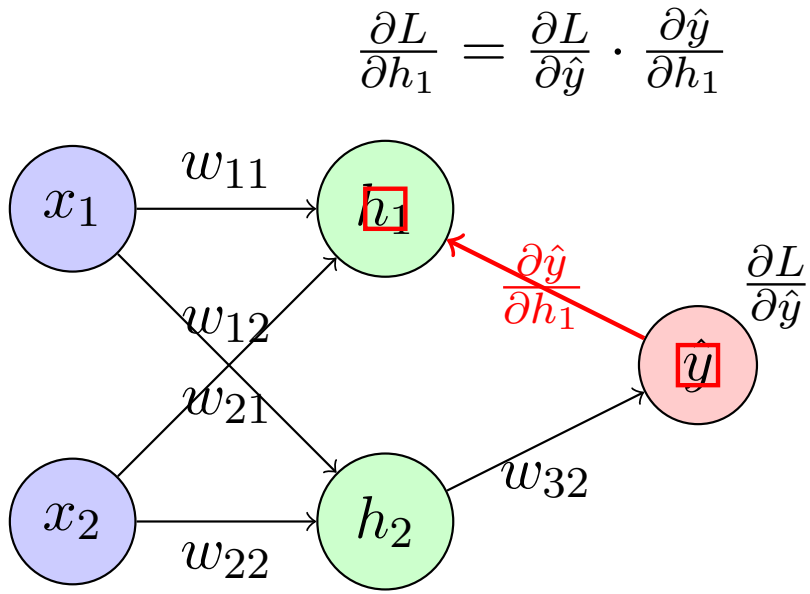


Figure: Illustration of the backward pass in the neural network. The red gradients are backpropagated gradients.

Weight Update

- Update the weights using gradient descent:

$$w_{ij} := w_{ij} - \eta \cdot \frac{\partial L}{\partial w_{ij}}$$

- Here, η is the learning rate.
- Repeat the forward and backward passes for multiple iterations (epochs) until the loss converges.