THE HONG KONG POLYTECHNIC UNIVERSITY
香港理工大學

# DSAI5207 Modern Deep Learning

## Multilayer Perceptrons (MLP)

Xingyi Yang
Department of Data Science and Artificial Intelligence
Hong Kong Polytechnic University

Opening Minds • Shaping the Future
啟迪思維 • 成就未來

# Agenda for Today

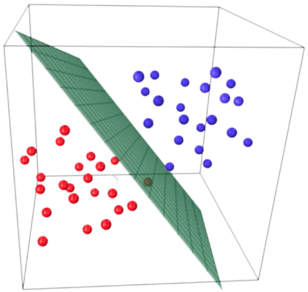
**MLP** ~**50min** → **Break 10m** → **Backprop 1h** → **Break 10m** → **Tutorial & Coding 40m**

# Contents

# Recap: Linear Classification

- Recall the simple linear classifier (Perceptron):

$$y = f(\mathbf{w}^T\mathbf{x} + b)$$



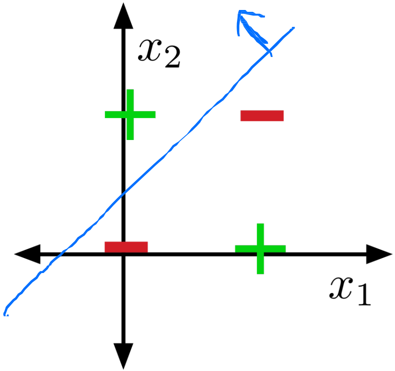
- It defines a linear decision boundary (hyperplane) in the input space.
- **Question:** Is a single linear unit enough for all problems?

# Example I: The XOR Problem

Consider the **XOR logic function**:

| $x_1$ | $x_2$ | $y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Can we separate the classes 0 and 1 with a **single straight line**?



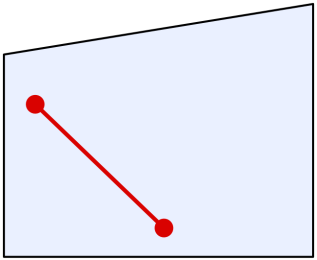

**No!** XOR is not linearly separable.

# The XOR Problem are not Linear Separable

## Convex Sets

A set $S$ is convex if any line segment connecting points in $S$ lies entirely within $S$.

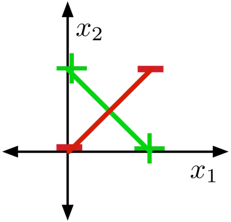
$$\mathbf{x}_1, \mathbf{x}_2 \in S \implies \lambda\mathbf{x}_1 + (1-\lambda)\mathbf{x}_2 \in S, \quad \forall\lambda \in [0, 1]$$



# The XOR Problem are not Linear Separable

Let's prove that XOR is not linearly separable

- Linear classifiers divide space into two **half-spaces**, which are convex sets.

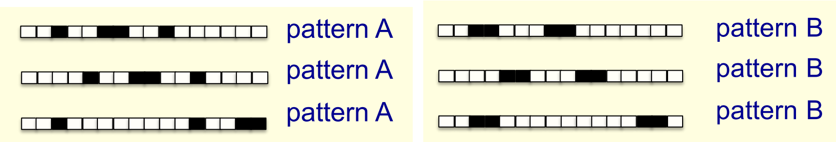



  — If positive examples $(0, 1)$ and $(1, 0)$ are in the positive half-space, the line connecting them must be too.
  — If negative examples $(0, 0)$ and $(1, 1)$ are in the negative half-space, their connecting line must be too.
  — These two lines **intersect**. intersection can't lie in both half-spaces. **Contradiction**!!!!

# Example II: Translation Invariance

Binary 16D vectors (white $= 0$, black $= 1$).

**Goal:** Distinguish patterns A and B under any cyclic translation.




**Key Insight:**

Linear classifier $y = \sum w_i x_i$ uses fixed per-pixel weights

$\Rightarrow$ Shifting the image $x$ activates different weights $w$. $\rightarrow$ output changes

$\Rightarrow$ To be invariant ($y$ stays same), all weights $w_i$ must be equal

$\Rightarrow$ If all weights are equal, the model **only counts the number of black pixels**

$\Rightarrow$ Patterns A and B (same 1 count) are indistinguishable.

**Conclusion:** Linear classifiers **cannot be translation invariant**!

# Overcoming Limits: Feature Maps

- We can solve **XOR** by mapping input to a higher-dimensional space $\psi(\mathbf{x})$.

# Overcoming Limits: Feature Maps

- We can solve **XOR** by mapping input to a higher-dimensional space $\psi(\mathbf{x})$.
- Example: $\psi(\mathbf{x}) = [x_1, x_2, x_1 x_2]^T$.

# Overcoming Limits: Feature Maps

- We can solve **XOR** by mapping input to a higher-dimensional space $\psi(\mathbf{x})$.
- Example: $\psi(\mathbf{x}) = [x_1, x_2, x_1 x_2]^T$.

# Overcoming Limits: Feature Maps

- We can solve **XOR** by mapping input to a higher-dimensional space $\psi(\mathbf{x})$.
- Example: $\psi(\mathbf{x}) = [x_1, x_2, x_1 x_2]^T$.

| **Input** | | **Feature Space** | | | **Tgt** |
|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_1$ | $x_2$ | $\psi_3$ | $t$ |
| 0 | 0 | 0 | 0 | **0** | 0 |
| 0 | 1 | 0 | 1 | **0** | 1 |
| 1 | 0 | 1 | 0 | **0** | 1 |
| 1 | 1 | 1 | 1 | **1** | 0 |

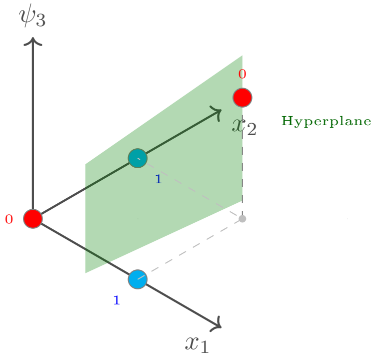
**Insight:** In 3D, the Red points are *above* the separator, while Blue points are *below* it.

# Overcoming Limits: Feature Maps

- We can solve **XOR** by mapping input to a higher-dimensional space $\psi(\mathbf{x})$.
- Example: $\psi(\mathbf{x}) = [x_1, x_2, x_1 x_2]^T$.

| **Input** | | **Feature Space** | | | **Tgt** |
|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_1$ | $x_2$ | $\psi_3$ | $t$ |
| 0 | 0 | 0 | 0 | **0** | 0 |
| 0 | 1 | 0 | 1 | **0** | 1 |
| 1 | 0 | 1 | 0 | **0** | 1 |
| 1 | 1 | 1 | 1 | **1** | 0 |

**Insight:** In 3D, the Red points are *above* the separator, while Blue points are *below* it.

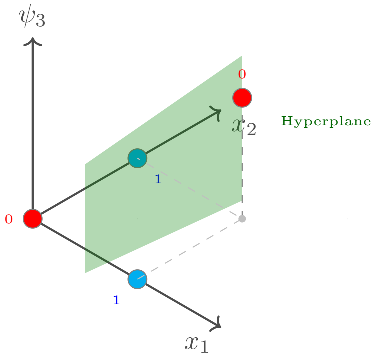


# Overcoming Limits: Feature Maps

- We can solve **XOR** by mapping input to a higher-dimensional space $\psi(\mathbf{x})$.
- Example: $\psi(\mathbf{x}) = [x_1, x_2, x_1 x_2]^T$.

| Input | | Feature Space | | | Tgt |
|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_1$ | $x_2$ | $\psi_3$ | $t$ |
| 0 | 0 | 0 | 0 | **0** | 0 |
| 0 | 1 | 0 | 1 | **0** | 1 |
| 1 | 0 | 1 | 0 | **0** | 1 |
| 1 | 1 | 1 | 1 | **1** | 0 |

**Insight:** In 3D, the Red points are *above* the separator, while Blue points are *below* it.

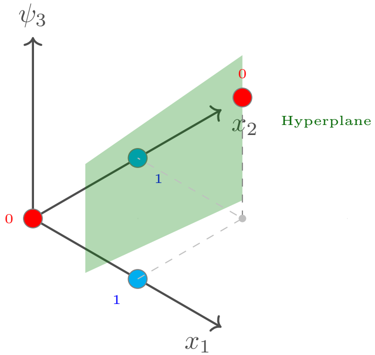



- In this new space, the problem is linearly separable.

# Overcoming Limits: Feature Maps

- We can solve **XOR** by mapping input to a higher-dimensional space $\psi(\mathbf{x})$.
- Example: $\psi(\mathbf{x}) = [x_1, x_2, x_1 x_2]^T$.

| **Input** | | **Feature Space** | | | **Tgt** |
|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_1$ | $x_2$ | $\psi_3$ | $t$ |
| 0 | 0 | 0 | 0 | **0** | 0 |
| 0 | 1 | 0 | 1 | **0** | 1 |
| 1 | 0 | 1 | 0 | **0** | 1 |
| 1 | 1 | 1 | 1 | **1** | 0 |




**Insight:** In 3D, the Red points are *above* the separator, while Blue points are *below* it.

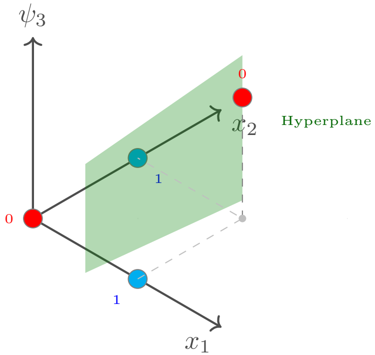
- In this new space, the problem is linearly separable.
- **Problem:** Hand-engineering features $\psi(\mathbf{x})$ is difficult and domain-specific (e.g., for pixels in images).

# Overcoming Limits: Feature Maps

- We can solve **XOR** by mapping input to a higher-dimensional space $\psi(\mathbf{x})$.
- Example: $\psi(\mathbf{x}) = [x_1, x_2, x_1 x_2]^T$.

| **Input** | | **Feature Space** | | | **Tgt** |
|---|---|---|---|---|---|
| $x_1$ | $x_2$ | $x_1$ | $x_2$ | $\psi_3$ | $t$ |
| 0 | 0 | 0 | 0 | **0** | 0 |
| 0 | 1 | 0 | 1 | **0** | 1 |
| 1 | 0 | 1 | 0 | **0** | 1 |
| 1 | 1 | 1 | 1 | **1** | 0 |

**Insight:** In 3D, the Red points are *above* the separator, while Blue points are *below* it.

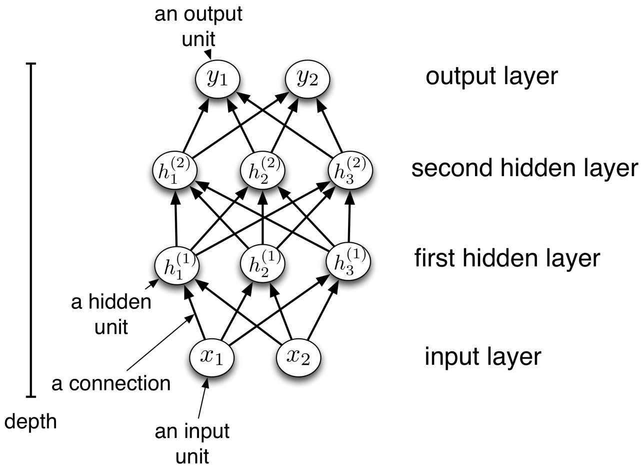



- In this new space, the problem is linearly separable.
- **Problem:** Hand-engineering features $\psi(\mathbf{x})$ is difficult and domain-specific (e.g., for pixels in images).
- **Solution:** Learn the non-linear mapping $\psi$ automatically! $\rightarrow$ **Multiple Layer of Neural Networks**.

# Contents

# Multilayer Perceptrons (MLPs)

- An MLP is a **Directed Acyclic Graph (DAG)** of units, organized in layers.
- Also known as a **Feed-Forward Network**.
- **Architecture:**
  1. **Input Layer:** Raw data $\mathbf{x}$
  2. **Hidden Layers:** Learn intermediate representations $\mathbf{h}^{(l)}$
  3. **Output Layer:** Predictions $\mathbf{y}$



# Fully Connected Layers

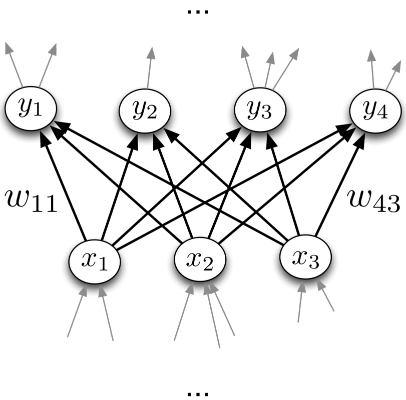
- In the simplest case, all input units are connected to all output units. We call this a **fully connected layer**.

**Layer $l$ Computation**

For every layer $l = 1 \dots L$:

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)} \qquad \text{(Linear)}$$

$$\mathbf{h}^{(l)} = \phi(\mathbf{z}^{(l)}) \qquad \text{(Non-linear)}$$




- $\mathbf{W}^{(l)}$: Weight matrix (learnable parameters).
- $\mathbf{b}^{(l)}$: Bias vector.
- $\phi$: Activation function (applied element-wise).

# Why Non-linearities?

- What if we stack linear layers without $\phi$?

$$\mathbf{y} = \mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} = \mathbf{W}'\mathbf{x}$$

# Why Non-linearities?

- What if we stack linear layers without $\phi$?

$$\mathbf{y} = \mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} = \mathbf{W}'\mathbf{x}$$

- **Crucial Insight**

  The composition of linear functions is just another linear function.

# Why Non-linearities?

- What if we stack linear layers without $\phi$?

$$\mathbf{y} = \mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} = \mathbf{W}'\mathbf{x}$$

- **Crucial Insight**

  The composition of linear functions is just another linear function.

- Deep linear networks have no more expressive power than a single linear layer!

# Why Non-linearities?

- What if we stack linear layers without $\phi$?

$$\mathbf{y} = \mathbf{W}_2(\mathbf{W}_1\mathbf{x}) = (\mathbf{W}_2\mathbf{W}_1)\mathbf{x} = \mathbf{W}'\mathbf{x}$$

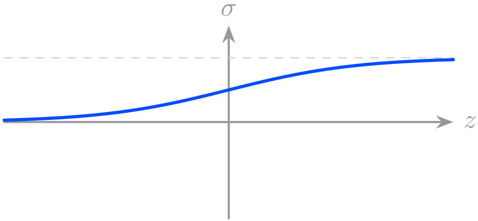
- **Crucial Insight**

  The composition of linear functions is just another linear function.

- Deep linear networks have no more expressive power than a single linear layer!
- **Activation functions** break linearity, allowing us to approximate curved boundaries and complex manifolds.

# Common Activation Functions (1)

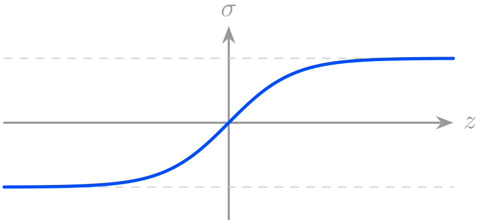
## Sigmoid




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

+ Interpretation: Probability
– Vanishing gradients
– Output not zero-centered

## Tanh

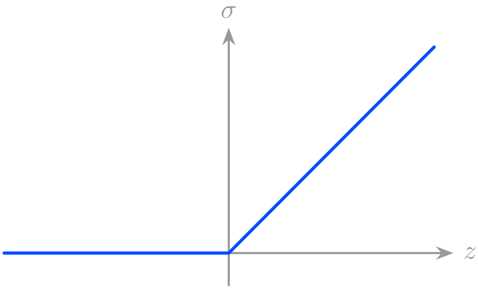



$$\sigma(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

+ Zero-centered output
+ Stronger gradients than Sigmoid
– Still saturates (vanishing grad)

# Common Activation Functions (2)

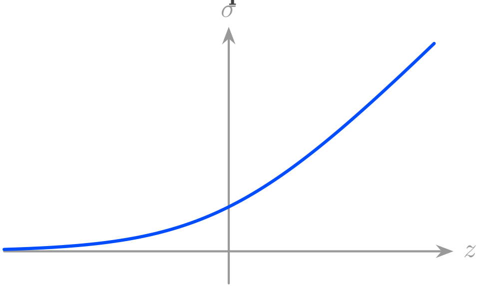
**ReLU**




$$\sigma(z) = \max(0, z)$$

+ Efficient; Sparse activations
+ No vanishing grad ($z > 0$)
– "Dead" neurons ($z < 0$)

**Softplus**

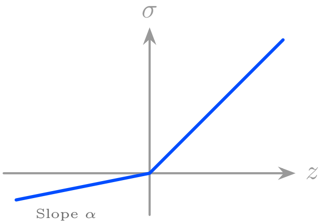



$$\sigma(z) = \ln(1 + e^z)$$

+ Smooth approx. of ReLU
+ Differentiable everywhere
– Computationally slower

# Modern Activation Functions (3)

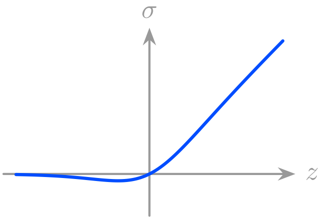
## Leaky ReLU




$$\sigma(z) = \max(\alpha z, z)$$

+ Fixes "Dead ReLU"
**Note:** $\alpha$ is small
**Use:** GANs, Deep MLPs

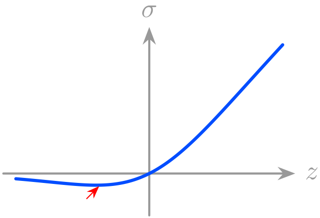
## GELU




$$\sigma(z) \approx z\Phi(z)$$

**Note:** $\Phi(z)$ is CDF of Normal
**Core:** **GPT, BERT, ViT**
+ Probabilistic
+ Optimization aid

## Swish / SiLU



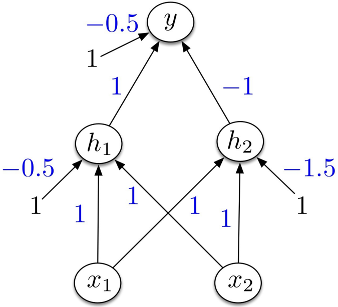

$$\sigma(z) = z \cdot \sigma_s(z)$$

**Note:** $\sigma_s(z)$ is Sigmoid
**Core:** **EfficientNet, YOLO**
+ Non-monotonic dip
+ Self-gating

# Solving XOR with an MLP

We can design a network that computes XOR *exactly* using hard threshold function:

$$\sigma(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



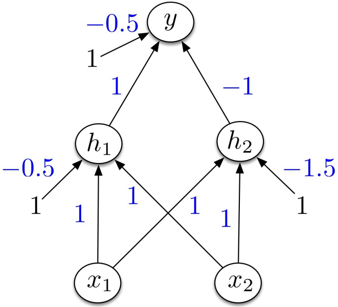

## Logic Circuits

- $h_1$ acts as **AND**$=x_1 \wedge x_2$: $\sigma(x_1 + x_2 - 1.5)$

# Solving XOR with an MLP

We can design a network that computes XOR *exactly* using hard threshold function:

$$\sigma(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



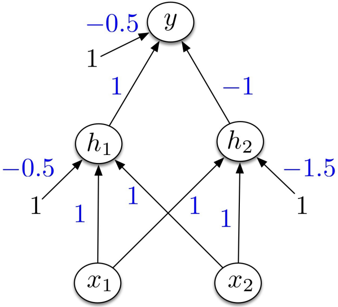

**Logic Circuits**

- $h_1$ acts as **AND**$= x_1 \wedge x_2$: $\sigma(x_1 + x_2 - 1.5)$
- $h_2$ acts as **OR**$= x_1 \vee x_2$: $\sigma(x_1 + x_2 - 0.5)$

# Solving XOR with an MLP

We can design a network that computes XOR *exactly* using hard threshold function:

$$\sigma(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



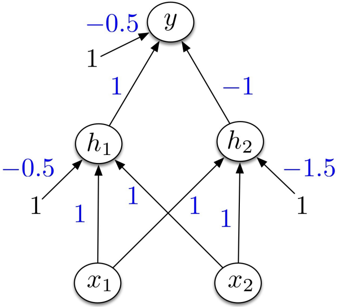

## Logic Circuits

- $h_1$ acts as **AND**$= x_1 \wedge x_2$: $\sigma(x_1 + x_2 - 1.5)$
- $h_2$ acts as **OR**$= x_1 \vee x_2$: $\sigma(x_1 + x_2 - 0.5)$
- $y$ acts as **XOR**$= (x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$: $\sigma(h_2 - h_1 - 0.5)$

# Solving XOR with an MLP

We can design a network that computes XOR *exactly* using hard threshold function:

$$\sigma(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$



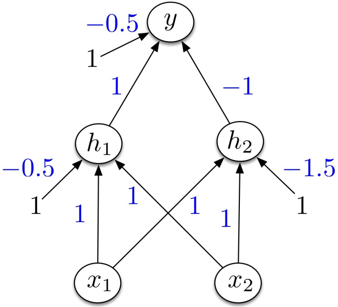

**Logic Circuits**

- $h_1$ acts as **AND**$=x_1 \wedge x_2$: $\sigma(x_1 + x_2 - 1.5)$
- $h_2$ acts as **OR**$=x_1 \vee x_2$: $\sigma(x_1 + x_2 - 0.5)$
- $y$ acts as **XOR**$=(x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$: $\sigma(h_2 - h_1 - 0.5)$

# Solving XOR with an MLP

We can design a network that computes XOR *exactly* using hard threshold function:

$$\sigma(z) = \begin{cases} 0 & \text{if } z < 0 \\ 1 & \text{if } z \geq 0 \end{cases}$$




## Logic Circuits

- $h_1$ acts as **AND**$=x_1 \wedge x_2$: $\sigma(x_1 + x_2 - 1.5)$
- $h_2$ acts as **OR**$=x_1 \vee x_2$: $\sigma(x_1 + x_2 - 0.5)$
- $y$ acts as **XOR**$=(x_1 \vee x_2) \wedge \neg(x_1 \wedge x_2)$: $\sigma(h_2 - h_1 - 0.5)$

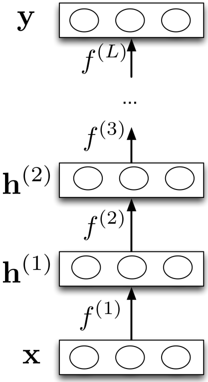
**Trace for input $(1, 1)$:**

$$h_1 = \sigma(1 + 1 - 1.5) = \sigma(0.5) = \mathbf{1}$$
$$h_2 = \sigma(1 + 1 - 0.5) = \sigma(1.5) = \mathbf{1}$$
$$y = \sigma(1 - 1 - 0.5) = \sigma(-0.5) = \mathbf{0}$$

# MLP as Feature Learning

Each layer computes a function, so the network computes a composition of functions:

$$\mathbf{h}^{(1)} = f^{(1)}(\mathbf{x})$$
$$\mathbf{h}^{(2)} = f^{(2)}(\mathbf{h}^{(1)})$$
$$\vdots$$
$$\mathbf{y} = f^{(L)}(\mathbf{h}^{(L-1)})$$

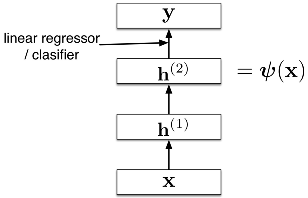



Or more generally: $\mathbf{y} = f^{(L)} \circ f^{(L-1)} \circ \cdots \circ f^{(1)}(\mathbf{x})$

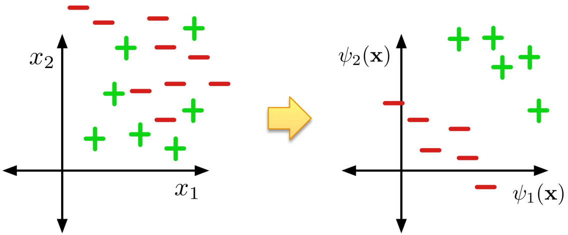
- Each layer transforms the input into a new, more abstract representation.
- The final layer produces the output based on these learned features.

# MLP as Feature Learning

- Neural nets can be viewed as a way of learning features:



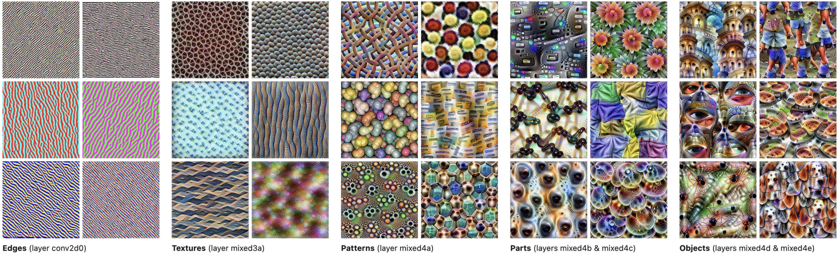

- The goal:



# MLP as Feature Learning: Image Classification

**Idea:** When multiple layers are stacked, the network learns hierarchical features:

- **Lower layers:** Detect simple patterns such as edges.
- **Middle layers:** Combine edges into parts.
- **Higher layers:** Assemble parts into objects.




Example: GoogLeNet trained on ImageNet reveals how features evolve from edges to complex objects using *DeepDream*.

# Designing the Output Layer

The choice of output function $\phi_{out}$ and Loss function $\mathcal{L}$ depends on the task:

| **Task** | **Target** $t$ | **Output Unit** | **Loss Function** |
|---|---|---|---|
| Regression | $t \in \mathbb{R}$ | Linear (Identity) | Squared Error |
| Binary Class. | $t \in \{0, 1\}$ | Sigmoid $\sigma(z)$ | Binary Cross-Entropy |
| Multi-class | $t \in \{1 \dots K\}$ | **Softmax** | **Cross-Entropy** |

# Expressive Power

**Universal Approximation Theorem**

A feed-forward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$ to arbitrary accuracy.

# Expressive Power

**Universal Approximation Theorem**

A feed-forward neural network with a single hidden layer containing a finite number of neurons can approximate any continuous function on compact subsets of $\mathbb{R}^n$ to arbitrary accuracy.

- **Implication:** Neural networks are universal function approximators.
- **Caveat:** The theorem says a network *exists*, but doesn't say how to find it (optimization) or how large it needs to be (efficiency).
- Various activation sufficient for approximation.
- We often prefer **deep** (more layers) networks over wide ones for better generalization and parameter efficiency.

# Contents

# Myth vs. Reality

## Myth

"Deep Learning replaced MLPs with ConvNets and Transformers."

# Myth vs. Reality

## Myth

"Deep Learning replaced MLPs with ConvNets and Transformers."

- **Reality: MLPs never left**. They just changed their name.
- In modern architectures (Transformers, NeRFs), MLPs typically constitute **2/3 of the total parameters**.
- In **NeRF**, the entire scene is an MLP.

Self-Attention

**MLP / FFN**
(Feed Forward)

Layer Norm

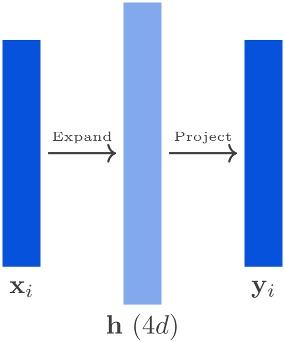
← In LLMs (GPT, Llama), this block runs on *every token* and consumes the majority of compute!

# Role 1: The "Channel Mixer" (Transformers)

In Transformers (e.g., ChatGPT), we distinguish two types of operations:

1. **Token Mixing (Attention)**
   - Mixes information *between* different words in the sequence.
   - "Looking at other words"

2. **Channel Mixing (MLP)**
   - Mixes information *within* a single word embedding.
   - Processing the features extracted by attention.
   - Applied **position-wise** (independently to every token).




$$\mathrm{FFN}(x) = \phi(x\mathbf{W}_1 + \mathbf{b}_1)\mathbf{W}_2 + \mathbf{b}_2$$

# Role 2: Knowledge Storage (Key-Value Memories)

**The Hypothesis (Geva et al., 2021)**

Transformer MLPs function as **associative memories**, storing factual knowledge in their weights.

**The Mechanism:**

**1. Pattern Detection (The Key)**

- The input $\mathbf{x}$ acts as a **query**.
- It scans the rows of $\mathbf{W}_1$ to find a matching pattern (e.g., "Capital" + "France").
- *Math:* $k = \sigma(\mathbf{x} \cdot \mathbf{W}_1^T)$

# Role 2: Knowledge Storage (Key-Value Memories)

**The Hypothesis (Geva et al., 2021)**

Transformer MLPs function as **associative memories**, storing factual knowledge in their weights.

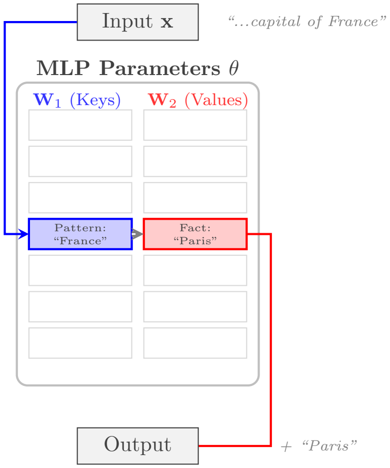
**The Mechanism:**

**1. Pattern Detection (The Key)**

- The input $\mathbf{x}$ acts as a **query**.
- It scans the rows of $\mathbf{W}_1$ to find a matching pattern (e.g., "Capital" + "France").
- *Math:* $k = \sigma(\mathbf{x} \cdot \mathbf{W}_1^T)$

**2. Fact Retrieval (The Value)**

- The active neuron "unlocks" the corresponding row in $\mathbf{W}_2$.
- The network retrieves the stored vector (e.g., "Paris").
- *Math:* $\mathbf{y} = k \cdot \mathbf{W}_2$



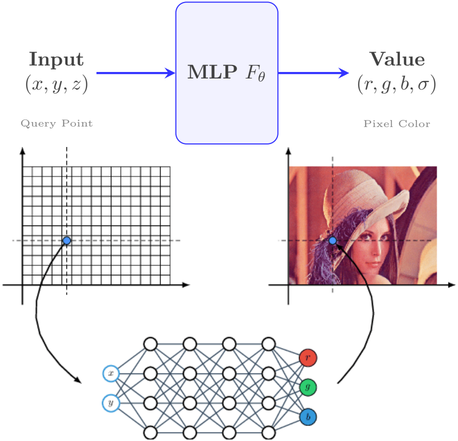
# Role 3: Implicit Neural Representations (INRs)

**Concept: Functions as Data**

MLPs are not just for *processing* data; they can **be** the data.

**Coordinate-based MLPs:**

- Instead of storing data in a discrete grid (pixels, voxels), we represent it as a continuous function $F_\theta$.




→ Infinite Resolution, Continuously Differentiable

# Role 3: Implicit Neural Representations (INRs)

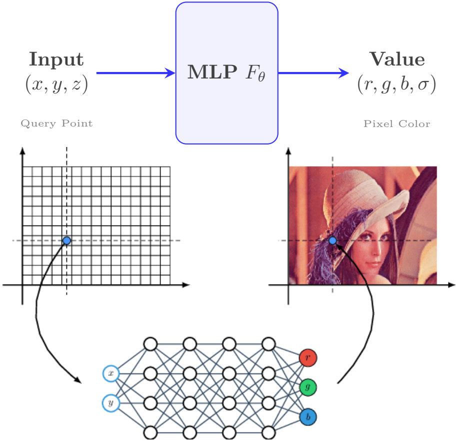
**Concept: Functions as Data**

MLPs are not just for *processing* data; they can **be** the data.

**Coordinate-based MLPs:**

- Instead of storing data in a discrete grid (pixels, voxels), we represent it as a continuous function $F_\theta$.
- $F_\theta$ as a **coordinate-wise look-up table**, mapping a position $\mathbf{x}$ to a value:

$$F_\theta(\mathbf{x}) \approx \text{Signal}(\mathbf{x})$$




→ Infinite Resolution, Continuously Differentiable

# Role 3: Implicit Neural Representations (INRs)

**Concept: Functions as Data**

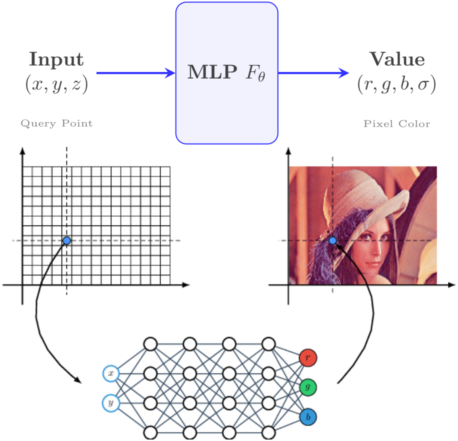
MLPs are not just for *processing* data; they can **be** the data.

**Coordinate-based MLPs:**

- Instead of storing data in a discrete grid (pixels, voxels), we represent it as a continuous function $F_\theta$.
- $F_\theta$ as a **coordinate-wise look-up table**, mapping a position $\mathbf{x}$ to a value:

$$F_\theta(\mathbf{x}) \approx \text{Signal}(\mathbf{x})$$

- The network weights $\theta$ act as the **compression storage**.




$\rightarrow$ Infinite Resolution, Continuously Differentiable

# Role 3: Implicit Neural Representations (INRs)

**Concept: Functions as Data**

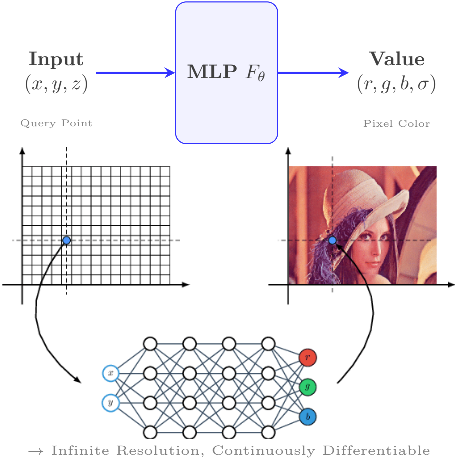
MLPs are not just for *processing* data; they can **be** the data.

**Coordinate-based MLPs:**

- Instead of storing data in a discrete grid (pixels, voxels), we represent it as a continuous function $F_\theta$.
- $F_\theta$ as a **coordinate-wise look-up table**, mapping a position $\mathbf{x}$ to a value:

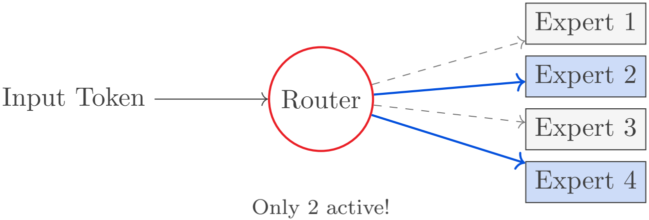
$$F_\theta(\mathbf{x}) \approx \text{Signal}(\mathbf{x})$$

- The network weights $\theta$ act as the **compression storage**.
- **Example:** neural radiance fields (NeRFs) for 3D rendering.



# Role 4: Mixture of Experts (MoE)

- As models grow, dense MLPs become too expensive to compute.
- **Solution: Mixture of Experts (MoE)**.
- Replace one giant MLP with $N$ smaller "expert" MLPs.
- A **Router** network selects only the top-$k$ (e.g., 2) experts for each token.




**Impact:** Massive capacity (params) with low inference cost (FLOPs). Used in **Mixtral**, **GPT-4**.

# Summary

1. **Linear Models** are limited (XOR problem, no compositionality).
2. **MLPs** introduce non-linearity via activation functions ($\sigma$), allowing universal approximation.
3. **Deep Learning** leverages depth to learn hierarchical features efficiently.
4. **Modern Usage:** MLPs are the workhorse of Transformers (Channel Mixing) and NeRFs.

# Next Up: Backpropagation

### The Big Question

The Universal Approximation Theorem guarantees that there exists a set of weights $\mathbf{W}^*$ that approximates our function $f(x)$.

**But how do we find $\mathbf{W}^*$?**

# Next Up: Backpropagation

## The Big Question

The Universal Approximation Theorem guarantees that there exists a set of weights $\mathbf{W}^*$ that approximates our function $f(x)$.

**But how do we find $\mathbf{W}^*$?**

We treat this as an **Optimization Problem**:

1. Define a **Loss Function** $\mathcal{L}(\mathbf{y}, \mathbf{t})$ that measures "badness".
2. Minimize this loss w.r.t parameters $\theta = \{\mathbf{W}, \mathbf{b}\}$.
3. Use **Gradient Descent**: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$.

# Next Up: Backpropagation

**The Big Question**

The Universal Approximation Theorem guarantees that there exists a set of weights $\mathbf{W}^*$ that approximates our function $f(x)$.

**But how do we find $\mathbf{W}^*$?**

We treat this as an **Optimization Problem**:

1. Define a **Loss Function** $\mathcal{L}(\mathbf{y}, \mathbf{t})$ that measures "badness".
2. Minimize this loss w.r.t parameters $\theta = \{\mathbf{W}, \mathbf{b}\}$.
3. Use **Gradient Descent**: $\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}$.

To do this, we will invent **Backpropagation** for neural nets.

See how to do backprop in next part!