Lecture 11: Neural Network (Basic Structure, Representation Power)

Tao LIN

SoE, Westlake University

December 2, 2025

Table of Contents

A Brief History of Neural Networks

- **1943:** McCulloch & Pitts propose the first mathematical model of a neuron.

Takeaway The **Deep Learning Revolution** (2012+) was enabled by three key factors: powerful GPUs, massive datasets, and improved algorithms (ReLU, dropout, batch normalization).

A Brief History of Neural Networks

- **1943:** McCulloch & Pitts propose the first mathematical model of a neuron.
- **1958:** Frank Rosenblatt invents the Perceptron (a linear classifier).

Takeaway The **Deep Learning Revolution** (2012+) was enabled by three key factors: powerful GPUs, massive datasets, and improved algorithms (ReLU, dropout, batch normalization).

A Brief History of Neural Networks

- **1943:** McCulloch & Pitts propose the first mathematical model of a neuron.
- **1958:** Frank Rosenblatt invents the Perceptron (a linear classifier).
- **1969:** Minsky & Papert publish "Perceptrons", showing that linear models cannot solve XOR. AI Winter I.

Takeaway The **Deep Learning Revolution** (2012+) was enabled by three key factors: powerful GPUs, massive datasets, and improved algorithms (ReLU, dropout, batch normalization).

A Brief History of Neural Networks

- **1943:** McCulloch & Pitts propose the first mathematical model of a neuron.
- **1958:** Frank Rosenblatt invents the Perceptron (a linear classifier).
- **1969:** Minsky & Papert publish "Perceptrons", showing that linear models cannot solve XOR. AI Winter I.
- **1986:** Rumelhart, Hinton, & Williams popularize Backpropagation, enabling training of multi-layer nets.

Takeaway The **Deep Learning Revolution** (2012+) was enabled by three key factors: powerful GPUs, massive datasets, and improved algorithms (ReLU, dropout, batch normalization).

A Brief History of Neural Networks

- **1943:** McCulloch & Pitts propose the first mathematical model of a neuron.
- **1958:** Frank Rosenblatt invents the Perceptron (a linear classifier).
- **1969:** Minsky & Papert publish "Perceptrons", showing that linear models cannot solve XOR. AI Winter I.
- **1986:** Rumelhart, Hinton, & Williams popularize Backpropagation, enabling training of multi-layer nets.
- **1990s:** SVMs and Random Forests dominate. NNs are seen as hard to train and prone to overfitting.

Takeaway The **Deep Learning Revolution** (2012+) was enabled by three key factors: powerful GPUs, massive datasets, and improved algorithms (ReLU, dropout, batch normalization).

A Brief History of Neural Networks

- **1943:** McCulloch & Pitts propose the first mathematical model of a neuron.
- **1958:** Frank Rosenblatt invents the Perceptron (a linear classifier).
- **1969:** Minsky & Papert publish "Perceptrons", showing that linear models cannot solve XOR. AI Winter I.
- **1986:** Rumelhart, Hinton, & Williams popularize Backpropagation, enabling training of multi-layer nets.
- **1990s:** SVMs and Random Forests dominate. NNs are seen as hard to train and prone to overfitting.
- **2012:** AlexNet wins ImageNet. The Deep Learning revolution begins (GPUs + Big Data + Algorithms).

Takeaway The **Deep Learning Revolution** (2012+) was enabled by three key factors: powerful GPUs, massive datasets, and improved algorithms (ReLU, dropout, batch normalization).

Motivation: Limitations of Linear Models

- We have seen linear models like Logistic Regression:

$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{x}^\top \mathbf{w} + b)$$

Motivation: Limitations of Linear Models

- We have seen linear models like Logistic Regression:

$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{x}^\top \mathbf{w} + b)$$

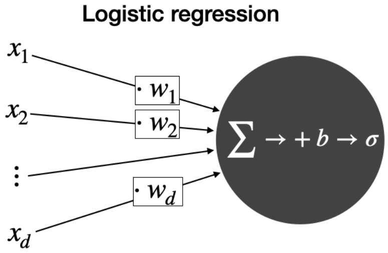
- **Problem:** They can only learn linear decision boundaries.

Motivation: Limitations of Linear Models

- We have seen linear models like Logistic Regression:

$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{x}^\top \mathbf{w} + b)$$

- **Problem:** They can only learn linear decision boundaries.
- **Example:** Consider data where $y = 0$ is inside a unit circle and $y = 1$ is outside.



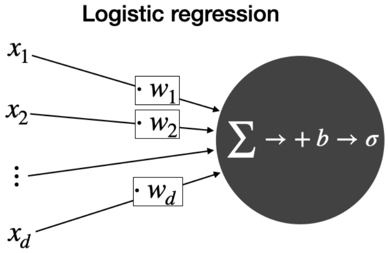
Linear models fail on non-linear data.

Motivation: Limitations of Linear Models

- We have seen linear models like Logistic Regression:

$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{x}^\top \mathbf{w} + b)$$

- **Problem:** They can only learn linear decision boundaries.
- **Example:** Consider data where $y = 0$ is inside a unit circle and $y = 1$ is outside.
- A linear classifier cannot separate this!



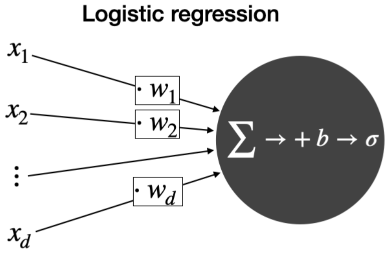
Linear models fail on non-linear data.

Motivation: Limitations of Linear Models

- We have seen linear models like Logistic Regression:

$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{x}^\top \mathbf{w} + b)$$

- **Problem:** They can only learn linear decision boundaries.
- **Example:** Consider data where $y = 0$ is inside a unit circle and $y = 1$ is outside.
- A linear classifier cannot separate this!
- **Old Solution:** Hand-craft features (e.g., $\phi(\mathbf{x}) = [x_1, x_2, x_1^2 + x_2^2]^\top$).



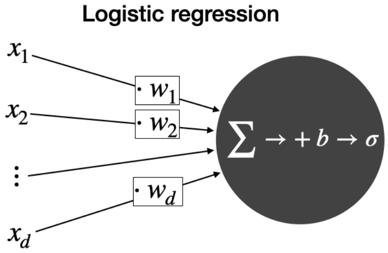
Linear models fail on non-linear data.

Motivation: Limitations of Linear Models

- We have seen linear models like Logistic Regression:

$$p(y = 1|\mathbf{x}) = \sigma(\mathbf{x}^\top \mathbf{w} + b)$$

- **Problem:** They can only learn linear decision boundaries.
- **Example:** Consider data where $y = 0$ is inside a unit circle and $y = 1$ is outside.
- A linear classifier cannot separate this!
- **Old Solution:** Hand-craft features (e.g., $\phi(\mathbf{x}) = [x_1, x_2, x_1^2 + x_2^2]^\top$).
- **New Solution:** Learn the features automatically using a Neural Network.



Linear models fail on non-linear data.

Table of Contents

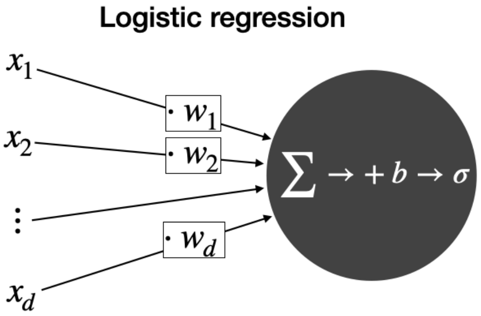
The Basic Building Block: The Neuron

A single neuron j in layer l computes:

$$z_j^{(l)} = \sum_i w_{i,j}^{(l)} x_i^{(l-1)} + b_j^{(l)}$$
$$x_j^{(l)} = \phi(z_j^{(l)})$$

Vectorized form: $\mathbf{x}^{(l)} = \phi(\mathbf{W}^{(l)\top}\mathbf{x}^{(l-1)} + \mathbf{b}^{(l)})$.

- $w_{i,j}^{(l)}$: Weights from layer $l-1$ to l.
- $b_j^{(l)}$: Bias term.
- $\phi(\cdot)$: Activation Function.

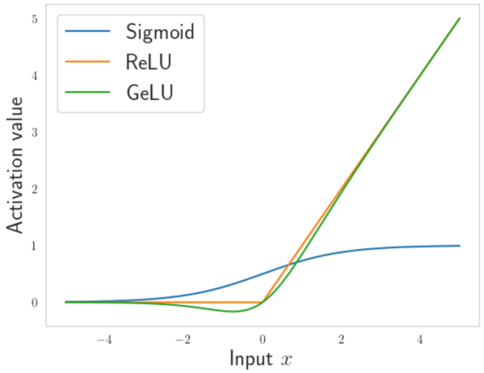


Crucial Insight ϕ must be **non-linear**. If ϕ were linear, the deep network would collapse into a single linear transformation:

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

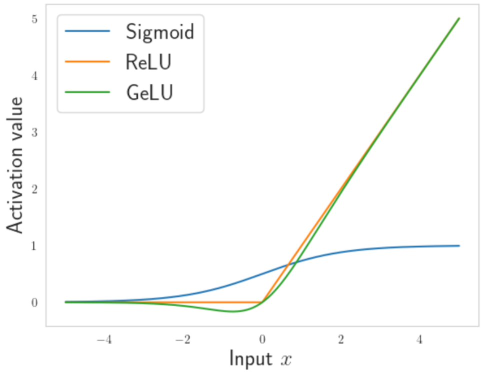
Common Activation Functions

- **Sigmoid:** $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Squashes output to $(0, 1)$.
  - Interpretable as probability.
  - **Issue:** Vanishing gradients for large $|z|$.



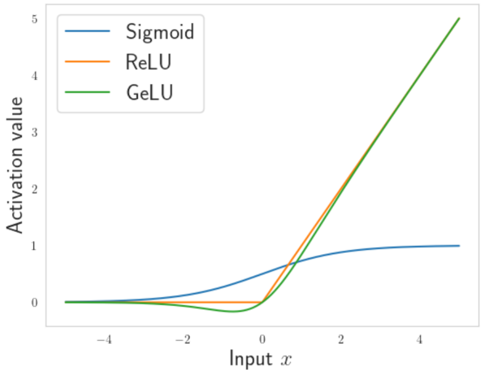
Common Activation Functions

- **Sigmoid:** $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Squashes output to $(0, 1)$.
  - Interpretable as probability.
  - **Issue:** Vanishing gradients for large $|z|$.
- **ReLU:** ReLU$(z) = \max(0, z)$
  - Default choice for modern deep learning.
  - Efficient, avoids vanishing gradients for $z > 0$.

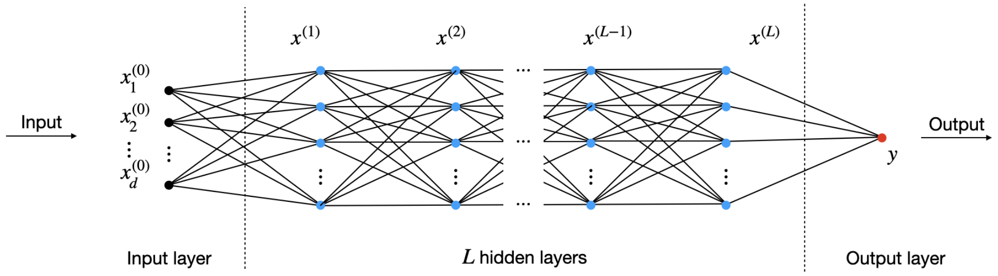


Common Activation Functions

- **Sigmoid:** $\sigma(z) = \frac{1}{1+e^{-z}}$
  - Squashes output to $(0, 1)$.
  - Interpretable as probability.
  - **Issue:** Vanishing gradients for large $|z|$.
- **ReLU:** ReLU$(z) = \max(0, z)$
  - Default choice for modern deep learning.
  - Efficient, avoids vanishing gradients for $z > 0$.
- **Others:** Tanh, Leaky ReLU, GeLU, Swish...



Multi-Layer Perceptron (MLP)



- **Input Layer:** Raw features $\mathbf{x} \in \mathbb{R}^D$.
- **Hidden Layers:** Learn abstract representations.
  - $f : \mathbb{R}^D \to \mathbb{R}^K$.
  - Replaces the "domain expert" feature engineering.
- **Output Layer:** Final prediction (e.g., linear classifier on top of learned features).

Table of Contents

Barron's Universal Approximation Theorem

Let $f : \mathbb{R}^d \to \mathbb{R}$ and define $\tilde{f}(\omega) = \int_{\mathbb{R}^d} f(\mathbf{x}) e^{-i\omega^\top \mathbf{x}} d\mathbf{x}$, its Fourier transform.

Assumption: $\int_{\mathbb{R}^d} |\omega||\tilde{f}(\omega)|\, d\omega \leq C$ (smoothness assumption)

Claim: For all $n \geq 1$ and $r > 0$, there exists a function f_n of the form

$$f_n(\mathbf{x}) = \sum_{j=1}^{n} c_j \phi(\mathbf{x}^\top \mathbf{w}_j + b_j) + c_0$$

such that

$$\int_{|\mathbf{x}| \leq r} (f(\mathbf{x}) - f_n(\mathbf{x}))^2 \, d\mathbf{x} \leq \frac{(2Cr)^2}{n}$$

A. R. Barron, *Universal approximation bounds for superpositions of a sigmoidal function*, IEEE Trans. Inf. Theory, 1993

All Sufficiently Smooth Functions Can Be Approximated

$$\int_{|\mathbf{x}|\leq r} (f(\mathbf{x}) - f_n(\mathbf{x}))^2 \, d\mathbf{x} \leq \frac{(2Cr)^2}{n} \tag{1}$$

- **The more neurons allowed, the smaller the error.**
  - Error decreases as $\mathcal{O}(1/n)$ — very fast convergence!

Takeaway One-hidden-layer neural networks can approximate *any* smooth function arbitrarily well, with error decreasing as $\mathcal{O}(1/n)$. This is the theoretical foundation for deep learning!

All Sufficiently Smooth Functions Can Be Approximated

$$\int_{|\mathbf{x}|\leq r} (f(\mathbf{x}) - f_n(\mathbf{x}))^2 \, d\mathbf{x} \leq \frac{(2Cr)^2}{n} \tag{1}$$

- **The more neurons allowed, the smaller the error.**
  - Error decreases as $\mathcal{O}(1/n)$ — very fast convergence!
- **The smoother the function (the smaller C), the smaller the error.**
  - The constant C measures the "complexity" of f in the frequency domain.

Takeaway One-hidden-layer neural networks can approximate *any* smooth function arbitrarily well, with error decreasing as $\mathcal{O}(1/n)$. This is the theoretical foundation for deep learning!

All Sufficiently Smooth Functions Can Be Approximated

$$\int_{|\mathbf{x}|\leq r} (f(\mathbf{x}) - f_n(\mathbf{x}))^2 \, d\mathbf{x} \leq \frac{(2Cr)^2}{n} \tag{1}$$

- **The more neurons allowed, the smaller the error.**
  - Error decreases as $\mathcal{O}(1/n)$ — very fast convergence!
- **The smoother the function (the smaller C), the smaller the error.**
  - The constant C measures the "complexity" of f in the frequency domain.
- **The larger the domain (the larger r), the greater the error.**
  - Need more neurons to approximate over larger regions.

Takeaway One-hidden-layer neural networks can approximate *any* smooth function arbitrarily well, with error decreasing as $\mathcal{O}(1/n)$. This is the theoretical foundation for deep learning!

All Sufficiently Smooth Functions Can Be Approximated

$$\int_{|\mathbf{x}|\leq r} (f(\mathbf{x}) - f_n(\mathbf{x}))^2 \, d\mathbf{x} \leq \frac{(2Cr)^2}{n} \tag{1}$$

- **The more neurons allowed, the smaller the error.**
  - Error decreases as $\mathcal{O}(1/n)$ — very fast convergence!
- **The smoother the function (the smaller C), the smaller the error.**
  - The constant C measures the "complexity" of f in the frequency domain.
- **The larger the domain (the larger r), the greater the error.**
  - Need more neurons to approximate over larger regions.
- **Approximation is in average (in L^2-norm).**
  - Not pointwise — some points may have larger errors.

Takeaway One-hidden-layer neural networks can approximate *any* smooth function arbitrarily well, with error decreasing as $\mathcal{O}(1/n)$. This is the theoretical foundation for deep learning!

All Sufficiently Smooth Functions Can Be Approximated

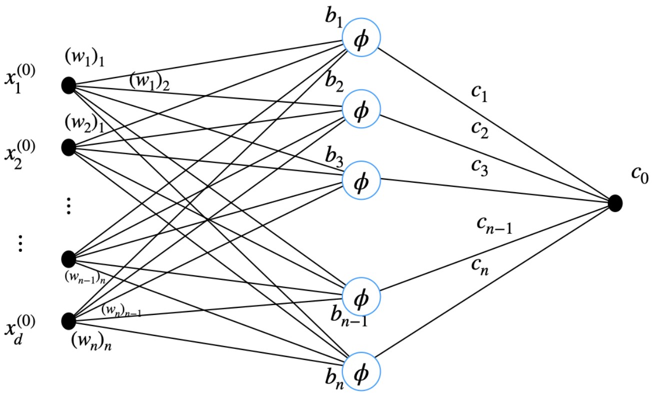
$$\int_{|\mathbf{x}|\le r} (f(\mathbf{x}) - f_n(\mathbf{x}))^2 \, d\mathbf{x} \le \frac{(2Cr)^2}{n} \tag{1}$$

- **The more neurons allowed, the smaller the error.**
  - Error decreases as $\mathcal{O}(1/n)$ — very fast convergence!
- **The smoother the function (the smaller C), the smaller the error.**
  - The constant C measures the "complexity" of f in the frequency domain.
- **The larger the domain (the larger r), the greater the error.**
  - Need more neurons to approximate over larger regions.
- **Approximation is in average (in L^2-norm).**
  - Not pointwise — some points may have larger errors.
- **Applicable for any "sigmoid-like" activation function.**
  - Any ϕ with $\lim_{x\to-\infty} \phi(x) = 0$ and $\lim_{x\to\infty} \phi(x) = 1$.

Takeaway One-hidden-layer neural networks can approximate *any* smooth function arbitrarily well, with error decreasing as $\mathcal{O}(1/n)$. This is the theoretical foundation for deep learning!

The Function f_n is a One-Hidden-Layer NN with n Nodes

$$f_n(\mathbf{x}) = \sum_{j=1}^{n} c_j \phi(\mathbf{x}^\top \mathbf{w}_j + b_j) + c_0 = \mathbf{c}^\top \phi(\mathbf{W}^\top \mathbf{x} + \mathbf{b}) + c_0$$

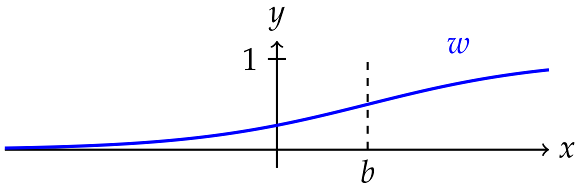


- **Input:** $\mathbf{x} \in \mathbb{R}^d$
- **Weights:** $\mathbf{w}_j \in \mathbb{R}^d$
- **Biases:** $b_j \in \mathbb{R}$
- **Output weights:** $c_j \in \mathbb{R}$
- **Activation:** ϕ (sigmoid-like)
- **Output:** scalar $f_n(\mathbf{x})$

Intuition: Constructing Functions (Step 1)

How can we build any function using Sigmoids?

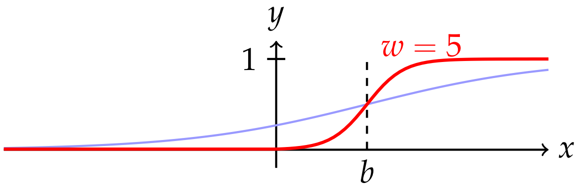
- Consider a neuron with weight w and bias $-wb$: $y = \sigma(w(x - b))$.



Intuition: Constructing Functions (Step 1)

How can we build any function using Sigmoids?

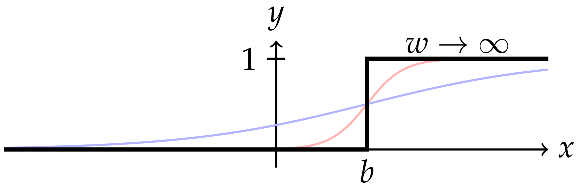
- Consider a neuron with weight w and bias $-wb$: $y = \sigma(w(x-b))$.
- As $w \to \infty$, the sigmoid becomes a Step Function at $x = b$.



Intuition: Constructing Functions (Step 1)

How can we build any function using Sigmoids?

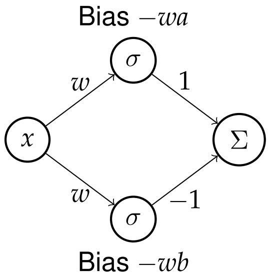
- Consider a neuron with weight w and bias $-wb$: $y = \sigma(w(x - b))$.
- As $w \to \infty$, the sigmoid becomes a Step Function at $x = b$.

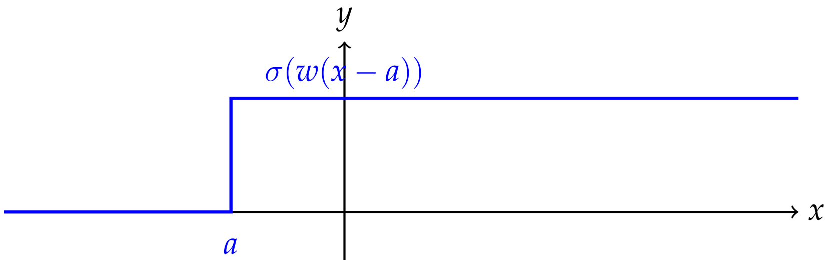


Intuition: Constructing Functions (Step 2)

- We can subtract two step functions to create a Rectangle (or "bump"):

$$\text{Bump}(x) \approx \sigma(w(x-a)) - \sigma(w(x-b))$$



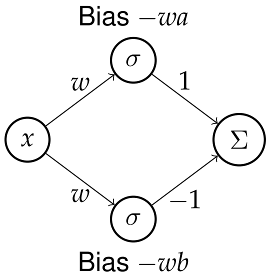


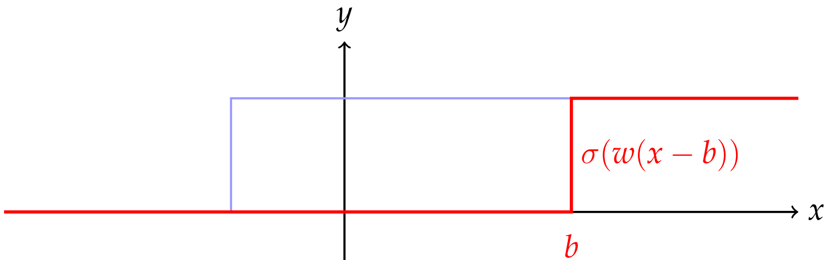
Intuition: Constructing Functions (Step 2)

- We can subtract two step functions to create a Rectangle (or "bump"):

$$\text{Bump}(x) \approx \sigma(w(x-a)) - \sigma(w(x-b))$$

- This requires **2 neurons** in the hidden layer.



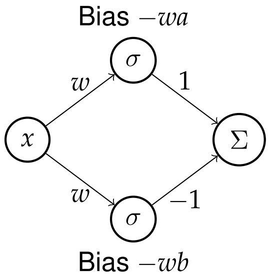


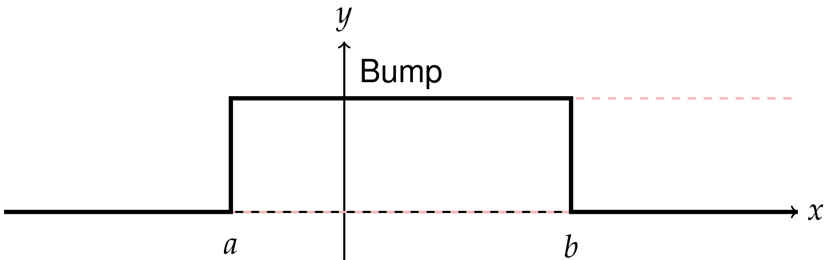
Intuition: Constructing Functions (Step 2)

- We can subtract two step functions to create a Rectangle (or "bump"):

$$\text{Bump}(x) \approx \sigma(w(x-a)) - \sigma(w(x-b))$$

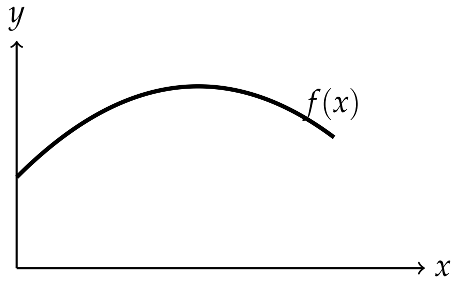
- This requires **2 neurons** in the hidden layer.
- We can control the width $(b-a)$ and position of the bump.





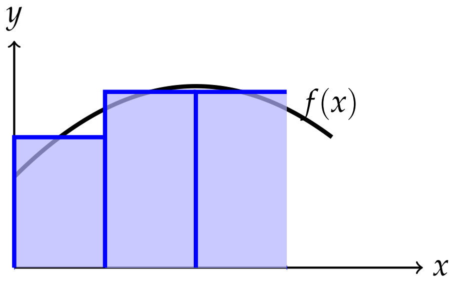
Intuition: Constructing Functions (Step 3)

- Any continuous function can be approximated by a sum of rectangles (Riemann Sum).



Intuition: Constructing Functions (Step 3)

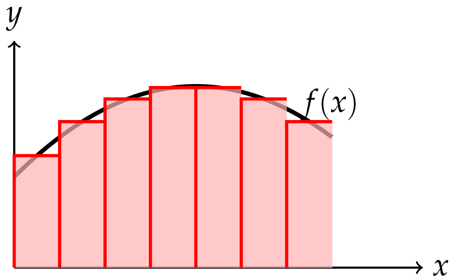
- Any continuous function can be approximated by a sum of rectangles (Riemann Sum).
- Since a NN can build any rectangle, it can sum them up to approximate the function.



$N = 3$

Intuition: Constructing Functions (Step 3)

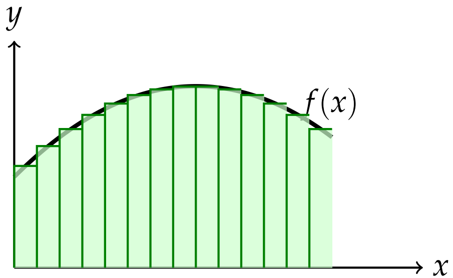
- Any continuous function can be approximated by a sum of rectangles (Riemann Sum).
- Since a NN can build any rectangle, it can sum them up to approximate the function.
- **Conclusion:** A 1-hidden layer NN with $2K$ nodes can approximate a function represented by K Riemann rectangles.



$N = 7$

Intuition: Constructing Functions (Step 3)

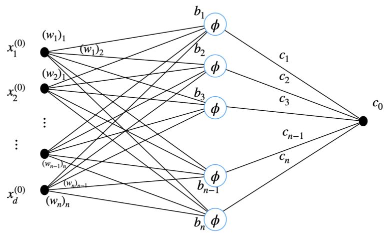
- Any continuous function can be approximated by a sum of rectangles (Riemann Sum).
- Since a NN can build any rectangle, it can sum them up to approximate the function.
- **Conclusion:** A 1-hidden layer NN with $2K$ nodes can approximate a function represented by K Riemann rectangles.



$N = 14$

Conclusion in the 1D Case

1. Approximate the function in the Riemann sense by a sum of k rectangles.
2. Represent each rectangle using **two nodes** in the hidden layer.
3. Compute the sum of all nodes in the hidden layer (considering appropriate weights and signs) to get the final output.



Remarks:

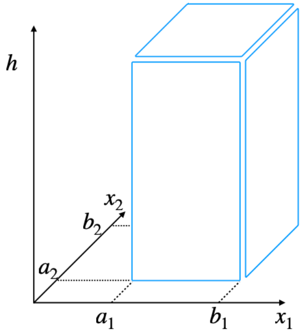
- The same intuition applies to any sigmoid-like function.
- This is an intuitive explanation, not a quantitative one.
- The weights w must be large to mimic the step function.

Takeaway **1D Construction:** Any continuous 1D function can be approximated using a one-hidden-layer network with $2k$ sigmoid neurons (for k rectangles). **The key:** sigmoids approximate step functions!

Larger Dimension: $d = 2$

Extension to 2D:

- Approximate the function by 2D box functions.
- Constructing a 2D box is harder than 1D. We need to intersect two infinite "strips".



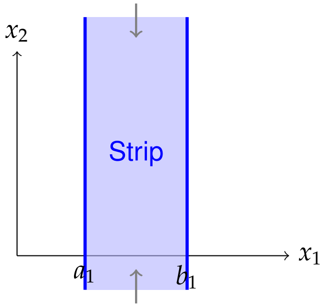
Step 1: Create Infinite Strip in x_1 Direction

$$(x_1, x_2) \mapsto \phi(w(x_1 - a_1)) - \phi(w(x_1 - b_1)) \tag{2}$$

Result:

- Value ≈ 1 when $a_1 \leq x_1 \leq b_1$
- Value ≈ 0 otherwise
- Unbounded in x_2 direction

Network: 2 sigmoid neurons in hidden layer.



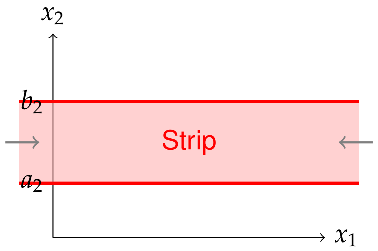
Step 2: Create Infinite Strip in x_2 Direction

$$(x_1, x_2) \mapsto \phi(w(x_2 - a_2)) - \phi(w(x_2 - b_2)) \tag{3}$$

Result:

- Value ≈ 1 when $a_2 \leq x_2 \leq b_2$
- Value ≈ 0 otherwise
- Unbounded in x_1 direction

Network: 2 more sigmoid neurons (total: 4 in hidden layer).



Step 3: Combine Strips to Form a Cross

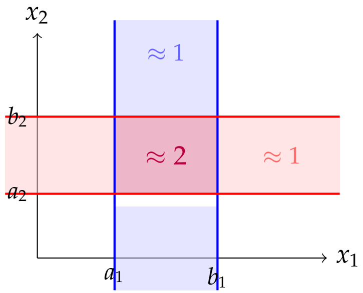
$$(x_1, x_2) \mapsto \phi(w(x_1 - a_1)) - \phi(w(x_1 - b_1)) + \phi(w(x_2 - a_2)) - \phi(w(x_2 - b_2)) \tag{4}$$

Result:

- Value ≈ 2 in rectangle (both strips active).
- Value ≈ 1 in arms (one strip active).
- Value ≈ 0 elsewhere.

Problem: Unwanted infinite arms! **Solution:**

Threshold at $c \in (1, 2]$.



Step 4: Thresholding (Construction vs. Existence)

Thresholding Operation:

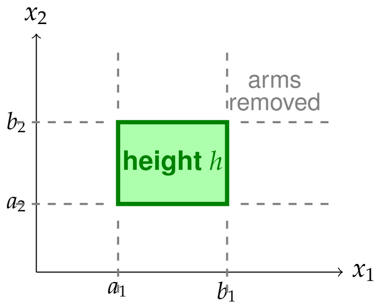
- Compose cross with $\mathbf{1}_{x>c}$ for $c \in (1, 2]$.
- We approximate $\mathbf{1}_{x>c}$ using a steep sigmoid:

$$\phi(w \cdot (x - c))$$

- **Implementation:** This effectively uses a **2nd hidden layer** to clean up the shape.

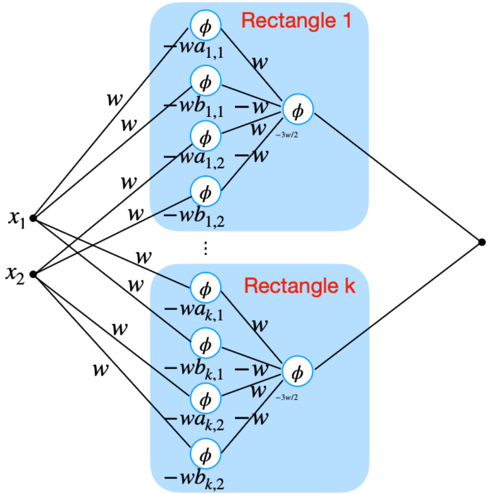
Result: Clean rectangle!

$$f(\mathbf{x}) \approx \begin{cases} h & \text{if } \mathbf{x} \in \text{Box} \\ 0 & \text{otherwise} \end{cases}$$

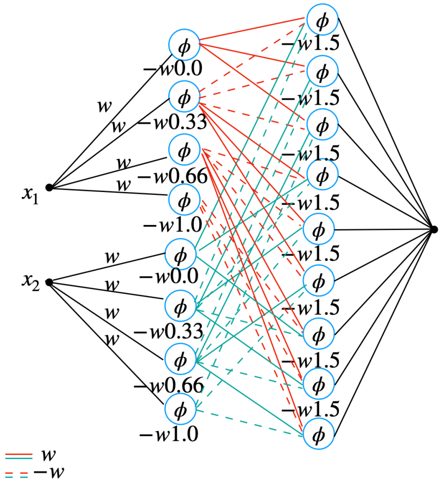


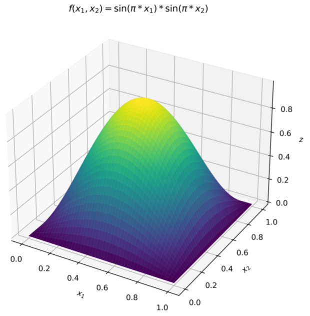
Deeper Networks: Efficiency

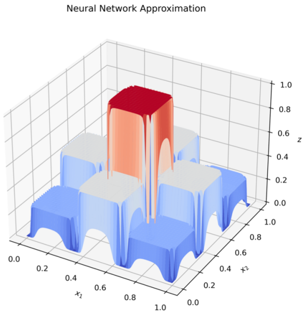
- While 1 layer is *sufficient* (Barron), it may require exponentially many neurons.
- Deep networks (2+ layers) can construct complex shapes (like clean rectangles) more efficiently.
- We can implement multiple rectangles with 2 hidden layers to approximate complex 2D functions.



Example: Approximate $f(\mathbf{x}) = \sin(\pi x_1) \cdot \sin(\pi x_2)$ with 9 Rectangles

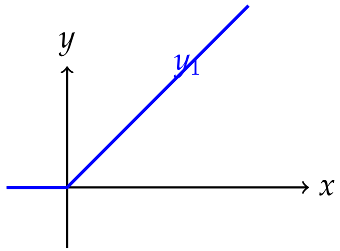






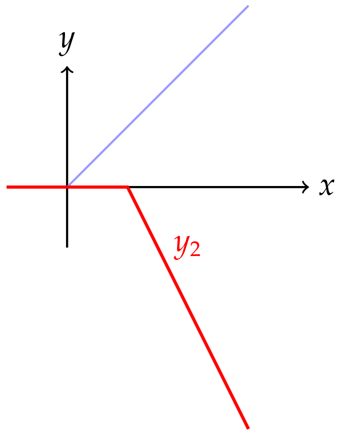
Approximation with ReLUs

- What about ReLU? $\phi(x) = \max(0, x)$.



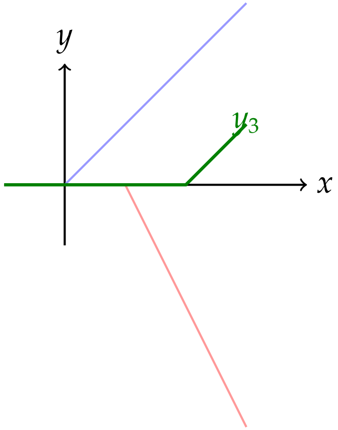
Approximation with ReLUs

- What about ReLU? $\phi(x) = \max(0, x)$.
- A ReLU unit introduces a "kink" (change in slope).



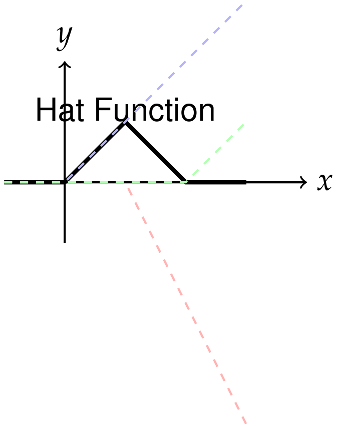
Approximation with ReLUs

- What about ReLU? $\phi(x) = \max(0, x)$.
- A ReLU unit introduces a "kink" (change in slope).
- Sum of ReLUs $\rightarrow$ Piecewise Linear (PWL) Function.



Approximation with ReLUs

- What about ReLU? $\phi(x) = \max(0, x)$.
- A ReLU unit introduces a "kink" (change in slope).
- Sum of ReLUs $\rightarrow$ Piecewise Linear (PWL) Function.
- Any continuous function can be approximated by a piecewise linear function.



ℓ_2 vs ℓ_∞ Approximations

- **ℓ_2-norm (average approx.):**
    - Measures the average deviation between the true function and the approximation.
    - Barron's result guarantees this for 1-layer nets.

ℓ_2 vs ℓ_∞ Approximations

- **ℓ_2-norm (average approx.):**
  - Measures the average deviation between the true function and the approximation.
  - Barron's result guarantees this for 1-layer nets.
- **ℓ_∞-norm (pointwise approx.):**
  - Measures the **maximum** error at any point: $\sup_x |f(x) - \hat{f}(x)|$.
  - Ensures uniformly small error across the domain.

ℓ_2 vs ℓ_∞ Approximations

- **ℓ_2-norm (average approx.):**
  - Measures the average deviation between the true function and the approximation.
  - Barron's result guarantees this for 1-layer nets.
- **ℓ_∞-norm (pointwise approx.):**
  - Measures the **maximum** error at any point: $\sup_x |f(x) - \hat{f}(x)|$.
  - Ensures uniformly small error across the domain.
- $\Rightarrow$ **Question:** Can we get ℓ_∞ approximation with Neural Networks?
  Yes, by exploiting Piecewise Linear (PWL) functions!

ℓ_∞ Approximation with Piecewise Linear Functions

Def: piecewise linear (PWL) function:

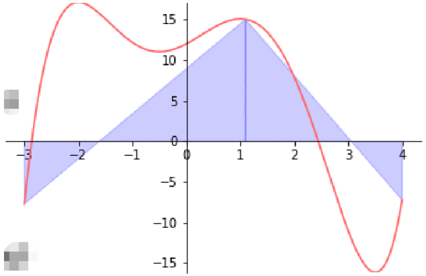
$$q(x) = \sum_{i=1}^{m} (a_i x + b_i) \mathbf{1}_{r_{i-1} \leq x < r_i} ,$$

with $a_i r_i + b_i = a_{i+1} r_i + b_{i+1}$ (continuity).

ℓ_∞**-approximation result** (Shekhtman, 1982):
Let f be a continuous function on $[c, d]$. For all $\varepsilon > 0$, there exists a piecewise linear function q such that:

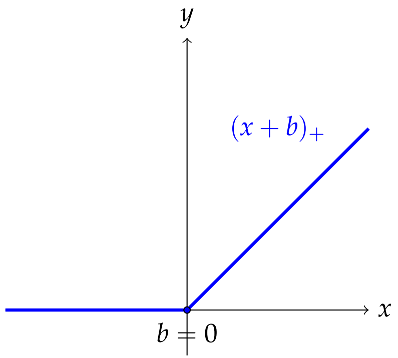
$$\sup_{x \in [c,d]} |f(x) - q(x)| \leq \varepsilon$$

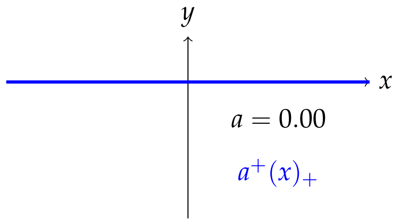
$\Rightarrow$ **Goal: Approximate PWL functions with a ReLU NN.**



ReLU Activation and PWL Functions

Role of the bias and weight in ReLU function $(ax+b)_+ = \max\{0, ax+b\}$:





The bias b determines the position of the kink. **The weight a determines the slope.**

A linear combination of ReLUs $\sum_{i=1}^{m} \tilde{a}_i(x-\tilde{b}_i)_+$ is a piecewise linear function.

Piecewise Linear Functions Can Be Written as Combination of ReLU

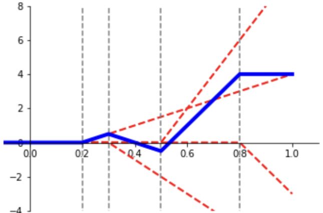
Claim 1: Any PWL $q(x)$ can be rewritten as:

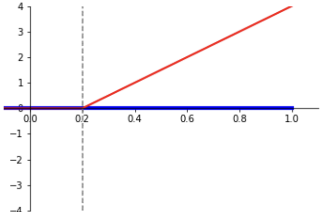
$$q(x) = \tilde{a}_1 x + \tilde{b}_1 + \sum_{i=2}^{m} \tilde{a}_i (x - \tilde{b}_i)_+$$

where $\tilde{a}_1 = a_1, \tilde{b}_1 = b_1, a_i = \sum_{j=1}^{i} \tilde{a}_j$ and $\tilde{b}_i = r_{i-1}$.

Proof sketch:

- How do we get a new segment with slope a starting at $r > \max(\tilde{b}_j)$?
- Get the kink at r by setting $\tilde{b}_{i+1} = r$ and slope by additionally canceling existing slope i.e. $\tilde{a}_{i+1} = a - \sum \tilde{a}_i$.

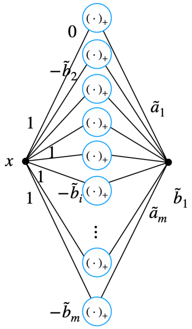




PWL as One-Hidden Layer NN with ReLU

Claim 2: q can be implemented as a one-hidden-layer NN with ReLU activation. Each term corresponds to one node:

- Bias $-\tilde{b}_i$
- Output weight $\tilde{a}_i$



The term $\tilde{a}_1 x + \tilde{b}_1$ corresponds to one node:

- Bias $\tilde{b}_1$: bias of the output node.
- Term $\tilde{a}_1 x = \tilde{a}_1 (x)_+$, assuming $x \geq 0$ (if input domain bounded).

Key Result Any ℓ_∞ approximation via PWL can be implemented with a one-hidden-layer ReLU network!

Proof of the Equivalent Formulation

Show: The two representations are equivalent:

$$q(x) = \sum_{i=1}^{m} (a_i x + b_i) \mathbf{1}_{r_{i-1} \le x < r_i}$$

$$r(x) = \tilde{a}_1 x + \tilde{b}_1 + \sum_{i=2}^{m} \tilde{a}_i (x - \tilde{b}_i)_+$$

where $\tilde{a}_1 = a_1$, $\tilde{b}_1 = b_1$, $a_i = \sum_{j=1}^{i} \tilde{a}_j$, and $\tilde{b}_i = r_{i-1}$.

- **For $x \in [0, r_1]$:**

$$(\tilde{a}_1, \tilde{b}_1) = (a_1, b_1) \quad \Longrightarrow \quad q(x) = a_1 x + b_1 = \tilde{a}_1 x + \tilde{b}_1 = r(x)$$

(because $\tilde{b}_2 = r_1$, so no ReLU terms activate)

- **For $x \in [r_1, r_2]$:**

$$\begin{aligned} r(x) &= \tilde{a}_1 x + \tilde{b}_1 + \tilde{a}_2 (x - r_1)_+ \\ &= a_1 x + b_1 + (a_2 - a_1)(x - r_1) = a_2 x + b_1 - (a_2 - a_1) r_1 = q(x) \end{aligned}$$

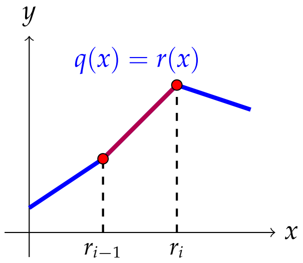
Note: $r'(x) = a_2$ and $r(r_1) = q(r_1)$, so $r(x) = q(x)$ for $x \in [r_1, r_2]$.

Proof by Induction

Induction Hypothesis: Assume $r(x) = q(x)$ for $x \in [0, r_{i-1}]$. **Induction Step:** Show $r(x) = q(x)$ for $x \in [r_{i-1}, r_i]$.

For $x \in [r_{i-1}, r_i]$:

$$\begin{aligned} r(x) &= \tilde{a}_1 x + \tilde{b}_1 + \sum_{j=2}^{m} \tilde{a}_j (x - \tilde{b}_j)_+ \\ &= \tilde{a}_1 x + \tilde{b}_1 + \sum_{j=2}^{i} \tilde{a}_j (x - \tilde{b}_j) = \sum_{j=1}^{i} \tilde{a}_j x + \dots \end{aligned}$$



The segment in red shows the interval $[r_{i-1}, r_i]$.

Thus:

- $r'(x) = \sum_{j=1}^{i} \tilde{a}_j = a_i$ (correct slope)
- $r(r_{i-1}) = q(r_{i-1})$ (correct starting point)

$\implies r(x) = q(x)$ for $x \in [r_{i-1}, r_i]$

Key insight: Two affine functions with the same starting point and slope are identical.

Table of Contents

The Learning Problem

Given a dataset $\mathcal{D} = \{(\mathbf{x}_n, y_n)\}_{n=1}^{N}$, we want to find parameters $\theta = \{\mathbf{W}^{(l)}, \mathbf{b}^{(l)}\}$ that minimize a loss function:

$$\min_{\theta} \mathcal{L}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \ell(y_n, f(\mathbf{x}_n; \theta))$$

- **Regression:** MSE Loss $\ell(y, \hat{y}) = (y - \hat{y})^2$.
- **Classification:** Cross-Entropy Loss $\ell(y, \hat{y}) = -\sum_k \mathbf{1}(y = k) \log \hat{y}_k$.

Backpropagation

- We use Gradient Descent (or SGD) to optimize θ:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

Takeaway **Backpropagation** makes training deep networks practical by efficiently computing gradients via the chain rule. Modern frameworks provide **automatic differentiation**, making implementation straightforward!

Backpropagation

- We use Gradient Descent (or SGD) to optimize θ:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

- How to compute gradients $\nabla_\theta \mathcal{L}$ for a deep net?

Takeaway **Backpropagation** makes training deep networks practical by efficiently computing gradients via the chain rule. Modern frameworks provide **automatic differentiation**, making implementation straightforward!

Backpropagation

- We use Gradient Descent (or SGD) to optimize θ:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

- How to compute gradients $\nabla_\theta \mathcal{L}$ for a deep net?
- **Backpropagation:** Efficient application of the Chain Rule.
  - **Forward Pass:** Compute activations layer by layer.
  - **Backward Pass:** Propagate error signals (δ) from output to input.

Takeaway **Backpropagation** makes training deep networks practical by efficiently computing gradients via the chain rule. Modern frameworks provide **automatic differentiation**, making implementation straightforward!

Backpropagation

- We use Gradient Descent (or SGD) to optimize θ:

$$\theta \leftarrow \theta - \eta \nabla_\theta \mathcal{L}(\theta)$$

- How to compute gradients $\nabla_\theta \mathcal{L}$ for a deep net?
- **Backpropagation:** Efficient application of the Chain Rule.
  - **Forward Pass:** Compute activations layer by layer.
  - **Backward Pass:** Propagate error signals (δ) from output to input.
- Modern frameworks (PyTorch, TensorFlow) do this automatically (Autodiff).

Takeaway **Backpropagation** makes training deep networks practical by efficiently computing gradients via the chain rule. Modern frameworks provide **automatic differentiation**, making implementation straightforward!

Summary

- **Structure:** NNs are composed of layers of neurons.

$$\text{Linear Transform}(\mathbf{W}\mathbf{x} + \mathbf{b}) \rightarrow \text{Non-linear Activation}(\phi) \rightarrow \dots$$

  Non-linearity is essential for learning complex patterns!

- **Representation Power:** NNs are Universal Approximators.
  - **Barron's Theorem:** One hidden layer can approximate any smooth function with $\mathcal{O}(1/n)$ error.
  - **1D:** Sigmoids create bumps $\rightarrow$ Riemann sums.
  - **2D:** Deeper networks (2 layers) are more efficient for constructing clean shapes.
  - **ReLU:** Equivalent to piecewise linear functions (ℓ_∞ approx).

- **Training:** Optimized via SGD using gradients computed by Backpropagation.

Final Takeaway Neural networks combine **universal approximation capability** with **efficient training** via backpropagation. This makes them the most powerful and practical machine learning models for complex, high-dimensional data!