Introduction to Machine Learning

Foundations and Applications

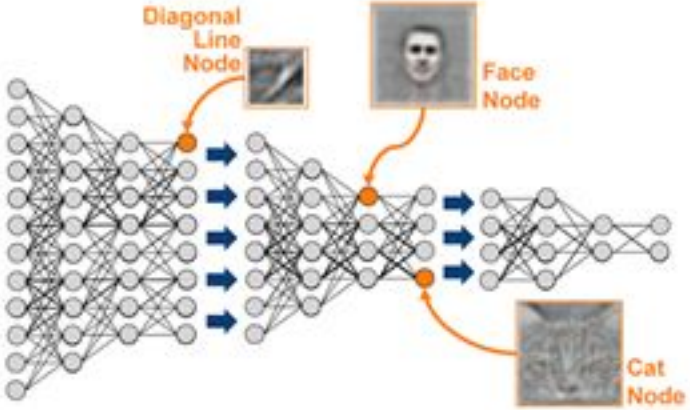
Paul J. Atzberger

University of California
Santa Barbara

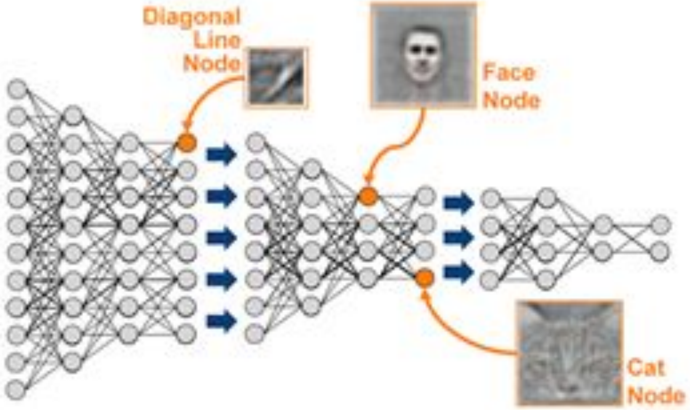
Deep Learning and Neural Networks

Deep Learning Overview

Machine Learning: Typical task is to try to learn a function $\mathrm{h}(\boldsymbol{x};\theta)$ from data $S = \{(x_i, y_i)\}$ that approximates y = f(x).



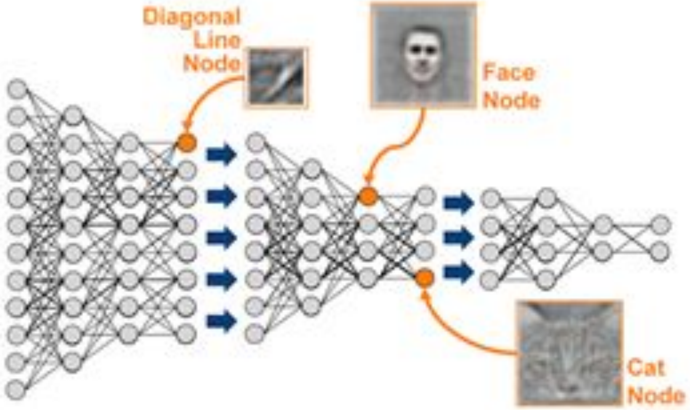
Deep Learning Overview



Machine Learning: Typical task is to try to learn a function $\mathrm{h}(\boldsymbol{x};\theta)$ from data $S = \{(x_i, y_i)\}$ that approximates y = f(x).

One approach: For regression use linear hypotheses $h(\boldsymbol{x}; W, b) = \boldsymbol{x}^T W + b$ or for classification $h(\boldsymbol{x}; W, b) = \mathrm{sign}(x^T W + b)$, [convex optimization problems].

Deep Learning Overview

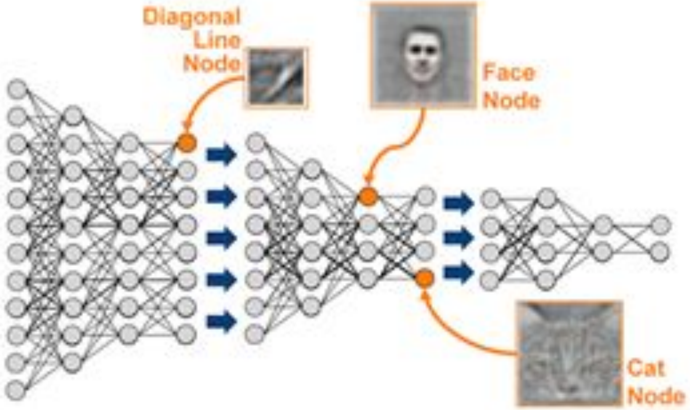


Machine Learning: Typical task is to try to learn a function $\mathrm{h}(\boldsymbol{x};\theta)$ from data $S = \{(x_i, y_i)\}$ that approximates $y = f(x)$.

One approach: For regression use linear hypotheses $h(\boldsymbol{x}; W, b) = \boldsymbol{x}^T W + b$ or for classification $h(\boldsymbol{x}; W, b) = \mathrm{sign}(x^T W + b)$, [convex optimization problems].

Non-linear case: We can use **kernel trick** with feature map $\mathbf{z} = \phi(\boldsymbol{x})$ and use linear methods in feature space $h(\boldsymbol{x}; W, b) = \phi(x)^T W + b$.

Deep Learning Overview



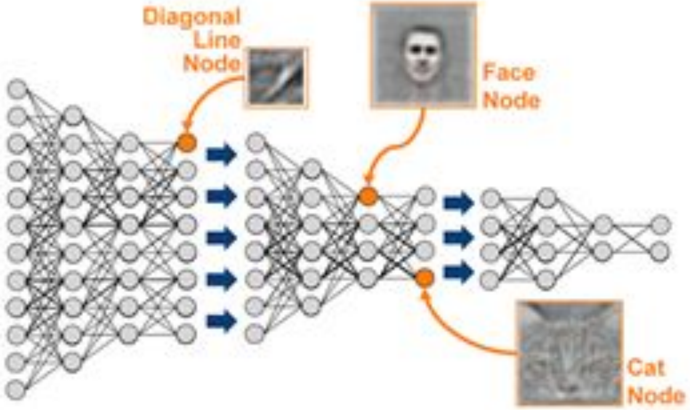
Machine Learning: Typical task is to try to learn a function $\mathrm{h}(\boldsymbol{x};\theta)$ from data $S = \{(x_i, y_i)\}$ that approximates y = f(x).

One approach: For regression use linear hypotheses $h(\boldsymbol{x}; W, b) = \boldsymbol{x}^T W + b$ or for classification $h(\boldsymbol{x}; W, b) = \mathrm{sign}(x^T W + b)$, [convex optimization problems].

Non-linear case: We can use **kernel trick** with feature map $\mathbf{z} = \phi(\boldsymbol{x})$ and use linear methods in feature space $h(\boldsymbol{x}; W, b) = \phi(x)^T W + b$.

Design of kernel $\mathrm{k}(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ incorporates prior knowledge and traditionally was done manually by experts (computer vision, speech recognition, natural language processing).

Deep Learning Overview



Machine Learning: Typical task is to try to learn a function $\mathrm{h}(\boldsymbol{x};\theta)$ from data $S = \{(x_i, y_i)\}$ that approximates y = f(x).

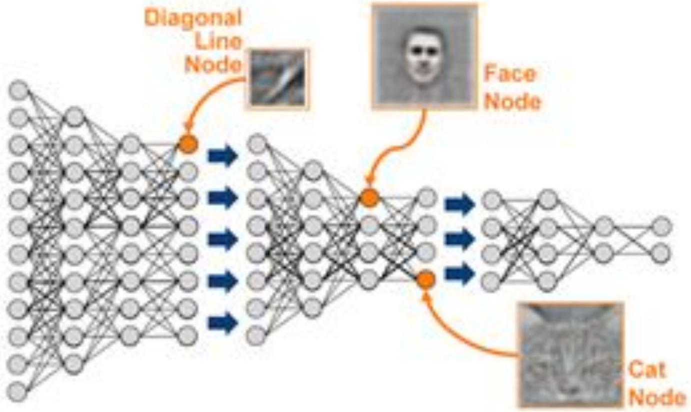
One approach: For regression use linear hypotheses $h(\boldsymbol{x}; W, b) = \boldsymbol{x}^T W + b$ or for classification $h(\boldsymbol{x}; W, b) = \mathrm{sign}(x^T W + b)$, [convex optimization problems].

Non-linear case: We can use **kernel trick** with feature map $\mathbf{z} = \phi(\boldsymbol{x})$ and use linear methods in feature space $h(\boldsymbol{x}; W, b) = \phi(x)^T W + b$.

Design of kernel $\mathrm{k}(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ incorporates prior knowledge and traditionally was done manually by experts (computer vision, speech recognition, natural language processing).

Data-driven approach: **Try to learn the feature map $\boldsymbol{\phi}$ from the data!** Find best parameters θ over some class of feature maps $\phi(\boldsymbol{x};\theta)$. Choice of feature map class can incorporate prior knowledge.

Deep Learning Overview



Machine Learning: Typical task is to try to learn a function $\mathrm{h}(\boldsymbol{x};\theta)$ from data $S = \{(x_i, y_i)\}$ that approximates y = f(x).

One approach: For regression use linear hypotheses $h(\boldsymbol{x}; W, b) = \boldsymbol{x}^T W + b$ or for classification $h(\boldsymbol{x}; W, b) = \mathrm{sign}(x^T W + b)$, [convex optimization problems].

Non-linear case: We can use **kernel trick** with feature map $\mathbf{z} = \phi(\boldsymbol{x})$ and use linear methods in feature space $h(\boldsymbol{x}; W, b) = \phi(x)^T W + b$.

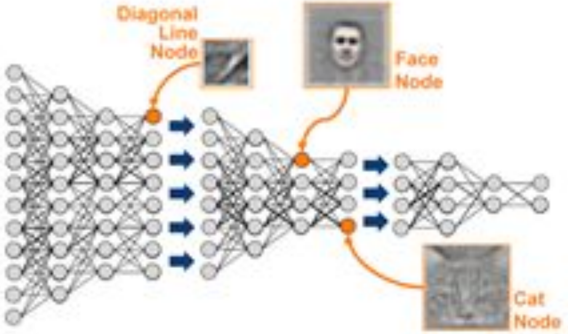
Design of kernel $\mathrm{k}(x_i, x_j) = \phi(x_i)^T \phi(x_j)$ incorporates prior knowledge and traditionally was done manually by experts (computer vision, speech recognition, natural language processing).

Data-driven approach: **Try to learn the feature map ϕ from the data!** Find best parameters θ over some class of feature maps $\phi(\boldsymbol{x};\theta)$. Choice of feature map class can incorporate prior knowledge.

Deep Learning: Many functions we try to learn in applications often can be approximated well by a composition of functions: $\mathrm{y} = \mathrm{f}(\boldsymbol{x}) = f^{(L)}\left(f^{(L-1)}\left(\ldots f^{(1)}(\boldsymbol{x}) \ldots\right)\right)$, where $f^{(k)}\colon \mathbb{R}^{N_k} \to \mathbb{R}^{N_{k+1}}$.

Deep Learning Overview

Data-driven approaches: **Try to learn the feature map ϕ from the data!** Find best parameters θ over some class of feature maps $\phi(\boldsymbol{x};\theta)$. Choice of feature map class can incorporate prior knowledge.

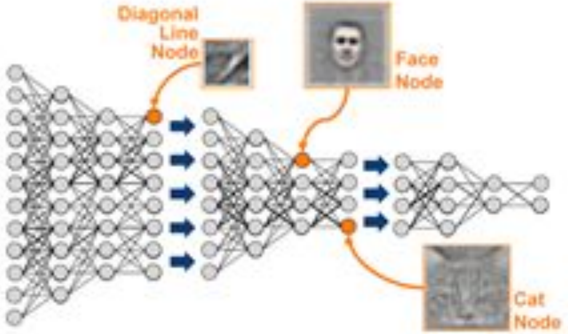


Deep Learning Overview

Data-driven approaches: **Try to learn the feature map ϕ from the data!** Find best parameters θ over some class of feature maps $\phi(\boldsymbol{x};\theta)$. Choice of feature map class can incorporate prior knowledge.

Deep Learning: Many functions in applications can be approximated well by a composition of functions:

$\mathrm{y} = \mathrm{f}(\boldsymbol{x}) = f^{(L)}\left(f^{(L-1)}\left(\ldots f^{(1)}(\boldsymbol{x})\ldots\right)\right)$, where $f^{(k)}\colon \mathbb{R}^{N_k} \to \mathbb{R}^{N_{k+1}}$.

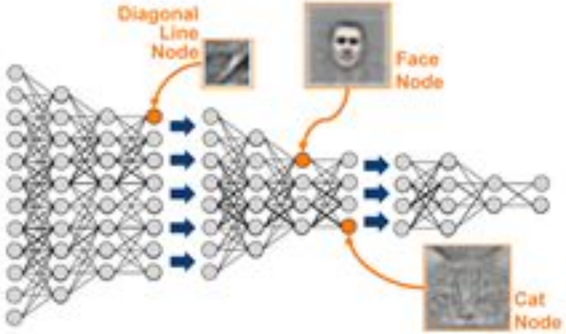


Deep Learning Overview

Data-driven approaches: **Try to learn the feature map ϕ from the data!** Find best parameters θ over some class of feature maps $\phi(\boldsymbol{x};\theta)$. Choice of feature map class can incorporate prior knowledge.

Deep Learning: Many functions in applications can be approximated well by a composition of functions:

$\mathrm{y} = \mathrm{f}(\boldsymbol{x}) = f^{(L)}\left(f^{(L-1)}\left(\ldots f^{(1)}(\boldsymbol{x}) \ldots\right)\right)$, where $f^{(k)}\colon \mathbb{R}^{N_k} \to \mathbb{R}^{N_{k+1}}$.



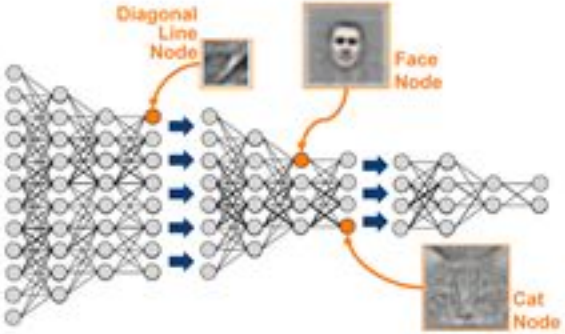
Ex: Regression $\mathrm{y} = |\cos(x_1 x_2) + 1|$ would have $f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)\right)$ with $f^{(1)}(x_1, x_2) = x_1 x_2$ and $f^{(2)}\left(h^{(1)}\right) = \cos\left(h^{(1)}\right) + 1$ and $f^{(3)}\left(h^{(2)}\right) = \mathrm{abs}\left(h^{(2)}\right)$.

Deep Learning Overview

Data-driven approaches: **Try to learn the feature map ϕ from the data!** Find best parameters θ over some class of feature maps $\phi(\boldsymbol{x};\theta)$. Choice of feature map class can incorporate prior knowledge.

Deep Learning: Many functions in applications can be approximated well by a composition of functions:

$\mathrm{y} = \mathrm{f}(\boldsymbol{x}) = f^{(L)}\left(f^{(L-1)}\left(\ldots f^{(1)}(\boldsymbol{x})\ldots\right)\right)$, where $f^{(k)}\colon \mathbb{R}^{N_k} \to \mathbb{R}^{N_{k+1}}$.



Ex: Regression $\mathrm{y} = |\cos(x_1 x_2) + 1|$ would have $f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)\right)$ with $f^{(1)}(x_1, x_2) = x_1 x_2$ and $f^{(2)}\left(h^{(1)}\right) = \cos\left(h^{(1)}\right) + 1$ and $f^{(3)}\left(h^{(2)}\right) = \mathrm{abs}\left(h^{(2)}\right)$.

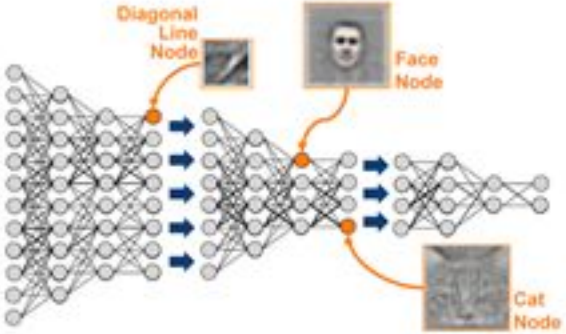
Generally, we can think about $\mathrm{y} = \mathrm{f}(\boldsymbol{x}) = f^{(L)}\left(f^{(L-1)}\left(\ldots f^{(1)}(\boldsymbol{x})\ldots\right)\right)$ in terms of components as

$$\mathrm{y} = f(x_1, x_2, \ldots, x_N) = f^{(L)}\left(f_1^{(L-1)}\left(z_{i_1}, z_{i_2}, \ldots, z_{i_{N1}}\right), f_2^{(L-1)}\left(z_{j_1}, z_{j_2}, \ldots, z_{j_{N2}}\right), \ldots, f_{n2}^{(L-1)}\left(z_{k_1}, z_{k_2}, \ldots, z_{k_{Nn2}}\right)\right),\ z = f^{(L-2)}$$

Deep Learning Overview

Data-driven approaches: **Try to learn the feature map ϕ from the data!** Find best parameters θ over some class of feature maps $\phi(\boldsymbol{x};\theta)$. Choice of feature map class can incorporate prior knowledge.

Deep Learning: Many functions in applications can be approximated well by a composition of functions: $y = f(\boldsymbol{x}) = f^{(L)}\left(f^{(L-1)}\left(\dots f^{(1)}(\boldsymbol{x})\dots\right)\right)$, where $f^{(k)}\colon \mathbb{R}^{N_k} \to \mathbb{R}^{N_{k+1}}$.



Ex: Regression $y = |\cos(x_1 x_2) + 1|$ would have $f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)\right)$ with $f^{(1)}(x_1, x_2) = x_1 x_2$ and $f^{(2)}\left(h^{(1)}\right) = \cos\left(h^{(1)}\right) + 1$ and $f^{(3)}\left(h^{(2)}\right) = \mathrm{abs}\left(h^{(2)}\right)$.

Generally, we can think about $y = f(\boldsymbol{x}) = f^{(L)}\left(f^{(L-1)}\left(\dots f^{(1)}(\boldsymbol{x})\dots\right)\right)$ in terms of components as

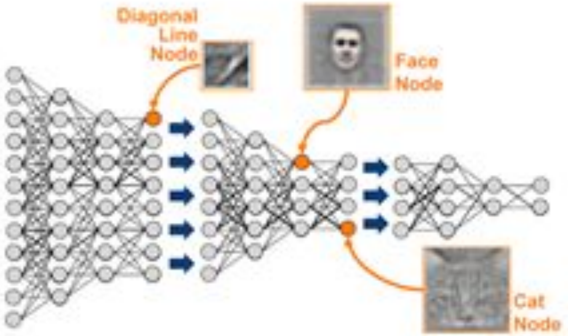
$$y = f(x_1, x_2, \dots, x_N) = f^{(L)}\left(f_1^{(L-1)}\left(z_{i_1}, z_{i_2}, \dots, z_{i_{N1}}\right), f_2^{(L-1)}\left(z_{j_1}, z_{j_2}, \dots, z_{j_{N2}}\right), \dots, f_{n2}^{(L-1)}\left(z_{k_1}, z_{k_2}, \dots, z_{k_{Nn2}}\right)\right), z = f^{(L-2)}$$

Concepts represented by hierarchy of distributed features. For instance: position of robotic actuator from angles in arm, identity of person from parts of the face, meaning of a sentence from phrases/words.

Deep Learning Overview

Data-driven approaches: **Try to learn the feature map ϕ from the data!** Find best parameters θ over some class of feature maps $\phi(\boldsymbol{x};\theta)$. Choice of feature map class can incorporate prior knowledge.

Deep Learning: Many functions in applications can be approximated well by a composition of functions:
$\mathrm{y} = \mathrm{f}(\boldsymbol{x}) = f^{(L)}\left(f^{(L-1)}\left(\ldots f^{(1)}(\boldsymbol{x})\ldots\right)\right)$, where $f^{(k)}\colon \mathbb{R}^{N_k} \to \mathbb{R}^{N_{k+1}}$.



Ex: Regression $\mathrm{y} = |\cos(x_1 x_2) + 1|$ would have $f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(\boldsymbol{x})\right)\right)$ with $f^{(1)}(x_1, x_2) = x_1 x_2$ and $f^{(2)}\left(h^{(1)}\right) = \cos\left(h^{(1)}\right) + 1$ and $f^{(3)}\left(h^{(2)}\right) = \mathrm{abs}\left(h^{(2)}\right)$.

Generally, we can think about $\mathrm{y} = \mathrm{f}(\boldsymbol{x}) = f^{(L)}\left(f^{(L-1)}\left(\ldots f^{(1)}(\boldsymbol{x})\ldots\right)\right)$ in terms of components as

$$\mathrm{y} = f(x_1, x_2, \ldots, x_N) = f^{(L)}\left(f_1^{(L-1)}\left(z_{i_1}, z_{i_2}, \ldots, z_{i_{N1}}\right), f_2^{(L-1)}\left(z_{j_1}, z_{j_2}, \ldots, z_{j_{N2}}\right), \ldots, f_{n2}^{(L-1)}\left(z_{k_1}, z_{k_2}, \ldots, z_{k_{Nn2}}\right)\right), z = f^{(L-2)}$$

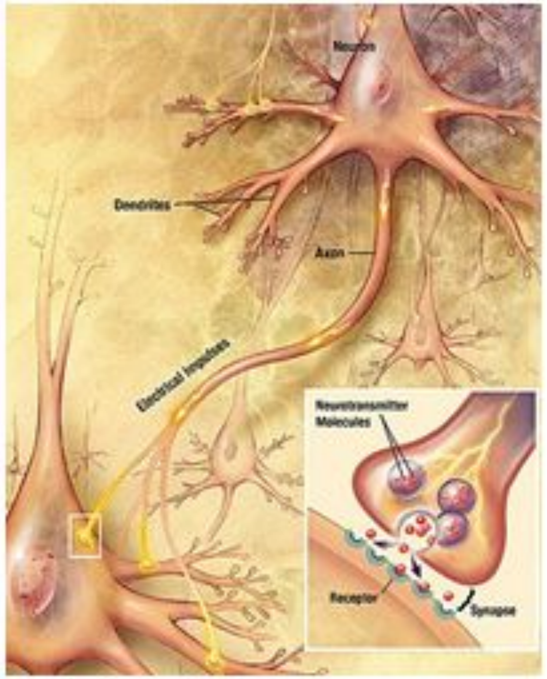
Concepts represented by hierarchy of distributed features. For instance: position of robotic actuator from angles in arm, identity of person from parts of the face, meaning of a sentence from phrases/words.

View as building up function from "hidden units" that detect particular features of input **z** as $h^{(k)} = f^{(k)}(\boldsymbol{z})$. Popular way to do this is to use Artificial Neural Networks (ANNs), $h^{(k)} = g(\boldsymbol{z}^T W + b)$.

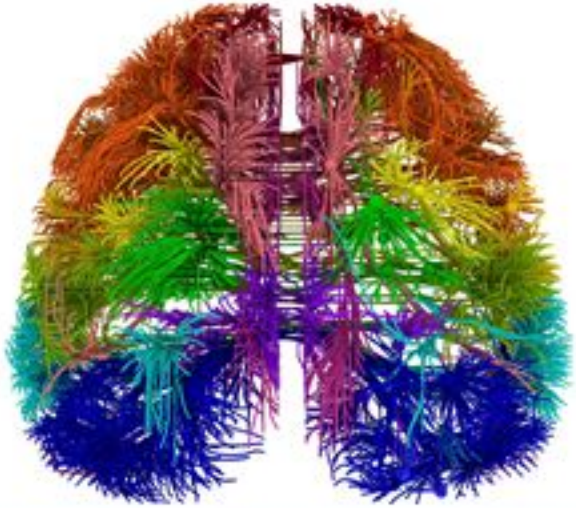
What are Artificial Neural Networks?

Biological neurons process information by firing to excite or inhibit neighbors:
action potentials → voltage-gated ion channels / neurotransmitters → collective neural activity.

Neuron: Axons and Dendrites



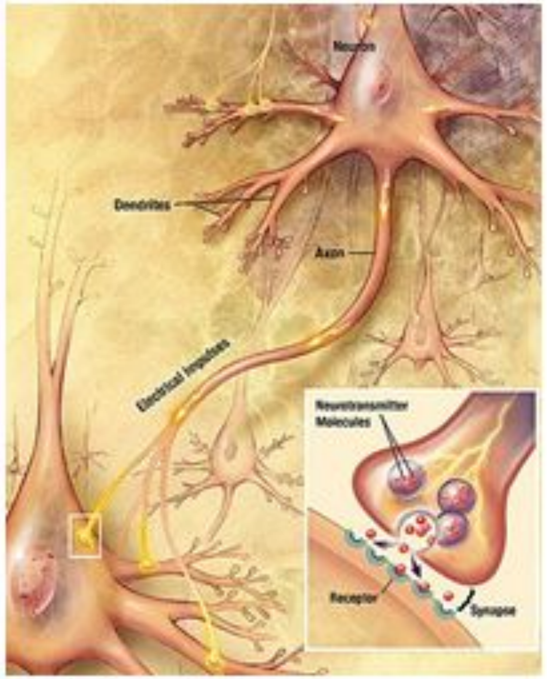
Human Connectome Project



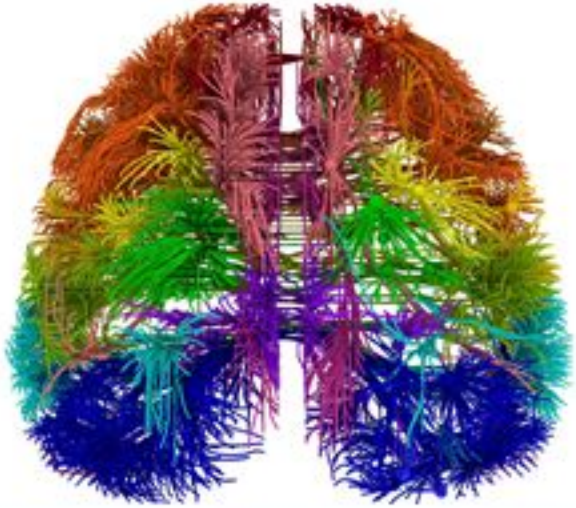
What are Artificial Neural Networks?

Biological neurons process information by firing to excite or inhibit neighbors:
action potentials → voltage-gated ion channels / neurotransmitters → collective neural activity.

Neuron: Axons and Dendrites



Human Connectome Project

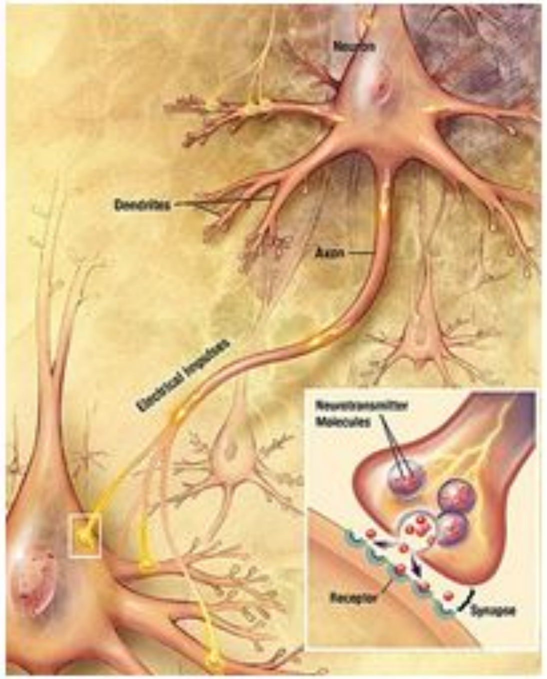


NN's are engineering tools inspired by nature. **Not a model of real neurons!**

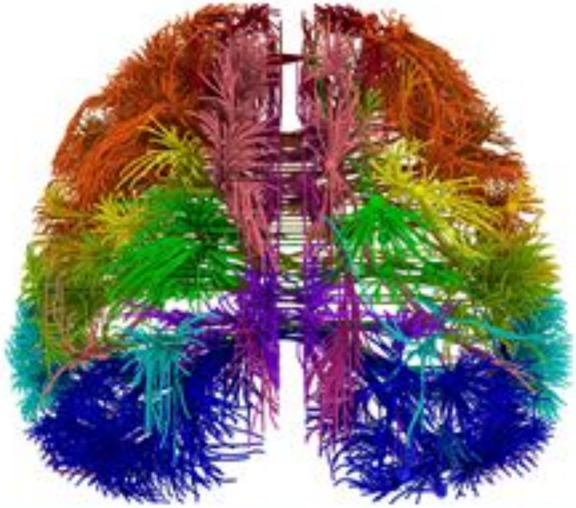
What are Artificial Neural Networks?

Biological neurons process information by firing to excite or inhibit neighbors: action potentials → voltage-gated ion channels / neurotransmitters → collective neural activity.

Neuron: Axons and Dendrites



Human Connectome Project



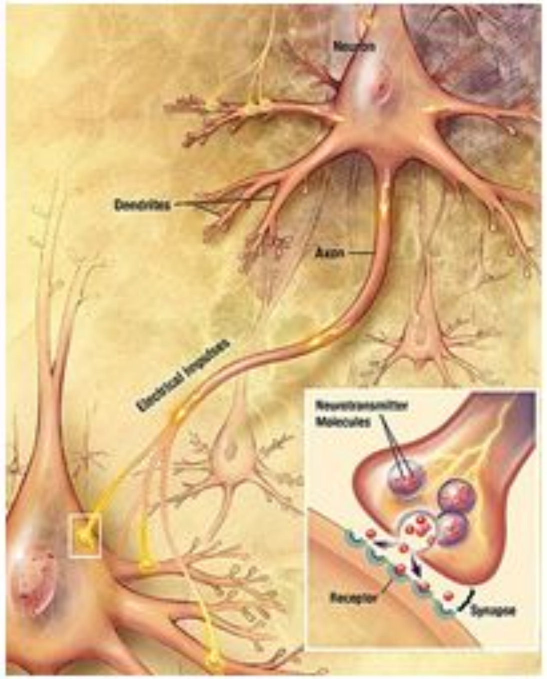
NN's are engineering tools inspired by nature. **Not a model of real neurons!** Biology is more complex / temporal dynamics, refraction / other relevant factors.

What are Artificial Neural Networks?

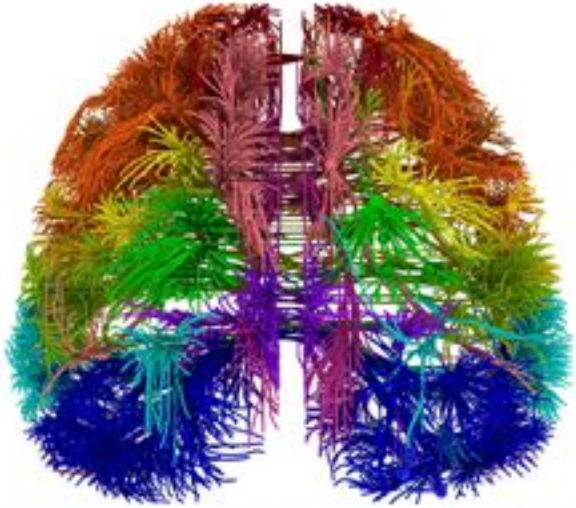
Biological neurons process information by firing to excite or inhibit neighbors:
action potentials → voltage-gated ion channels / neurotransmitters → collective neural activity.

Winged Flight

Neuron: Axons and Dendrites



Human Connectome Project



NN's are engineering tools inspired by nature. **Not a model of real neurons!**
Biology is more complex / temporal dynamics, refraction / other relevant factors.

What are Artificial Neural Networks?

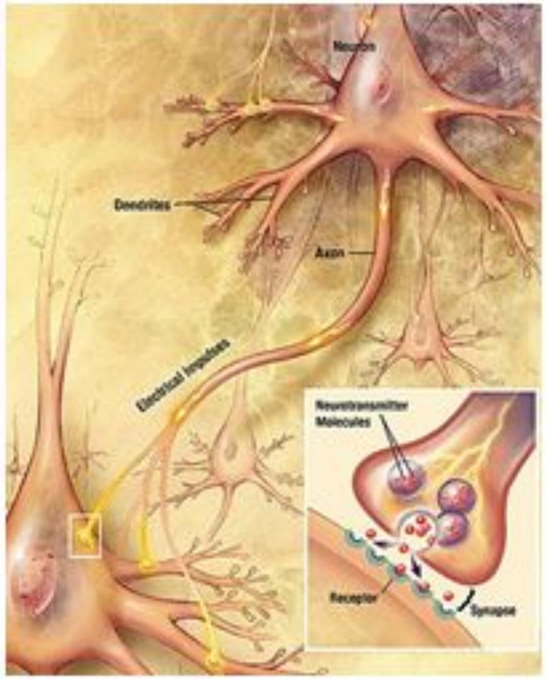
Biological neurons process information by firing to excite or inhibit neighbors:
action potentials → voltage-gated ion channels / neurotransmitters → collective neural activity.

Winged Flight

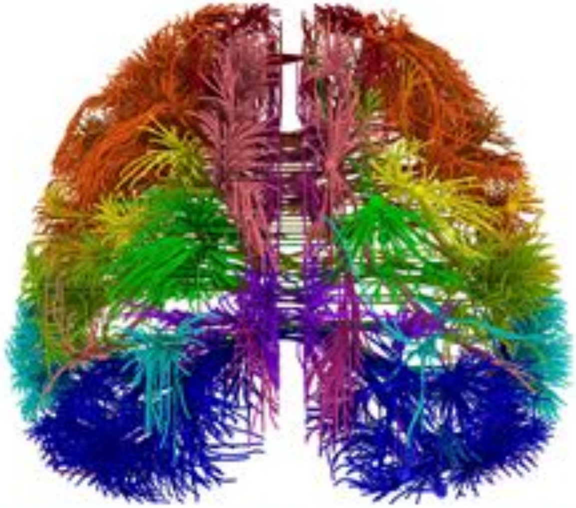
inspiration →

Wright Brothers

Neuron: Axons and Dendrites



Human Connectome Project

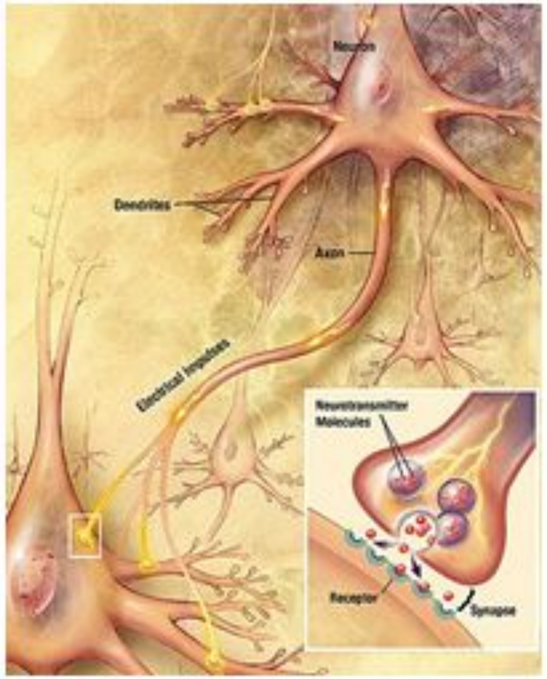


NN's are engineering tools inspired by nature. **Not a model of real neurons!**
Biology is more complex / temporal dynamics, refraction / other relevant factors.

What are Artificial Neural Networks?

Biological neurons process information by firing to excite or inhibit neighbors:
action potentials → voltage-gated ion channels / neurotransmitters → collective neural activity.

Neuron: Axons and Dendrites



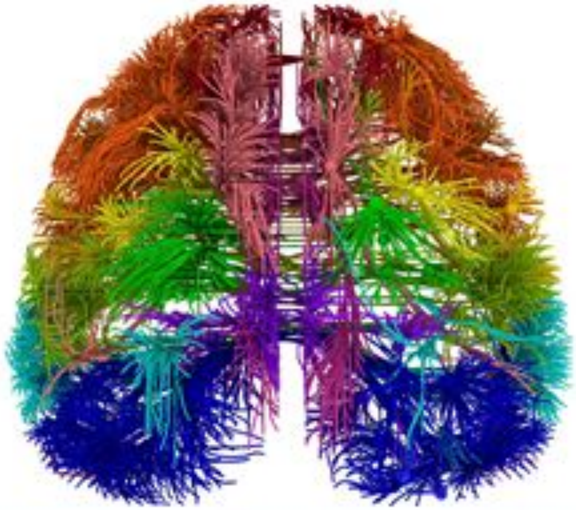
Winged Flight

inspiration →

Wright Brothers

Modern Airplanes

Human Connectome Project



NN's are engineering tools inspired by nature. **Not a model of real neurons!**
Biology is more complex / temporal dynamics, refraction / other relevant factors.

What are Artificial Neural Networks?

Biological neurons process information by firing to excite or inhibit neighbors:
action potentials → voltage-gated ion channels / neurotransmitters → collective neural activity.

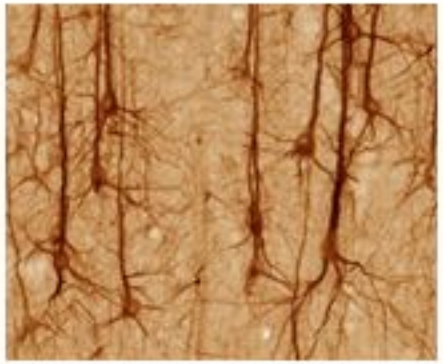
Winged Flight

inspiration →

Wright Brothers

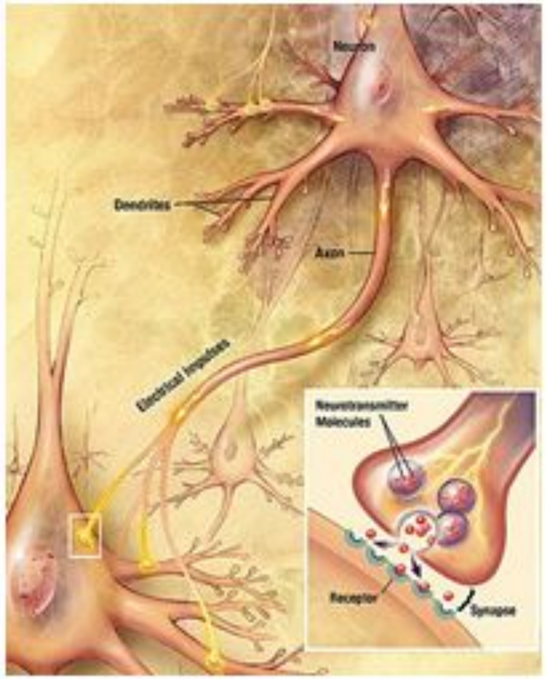
Modern Airplanes

Neural Information Processing

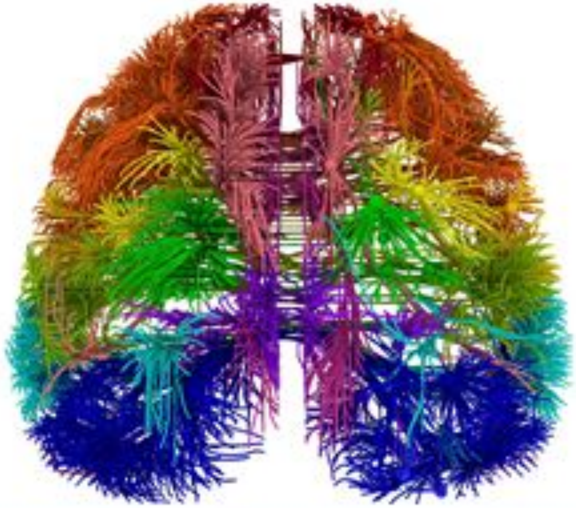


NN's are engineering tools inspired by nature. **Not a model of real neurons!**
Biology is more complex / temporal dynamics, refraction / other relevant factors.

Neuron: Axons and Dendrites



Human Connectome Project



What are Artificial Neural Networks?

Biological neurons process information by firing to excite or inhibit neighbors:
action potentials → voltage-gated ion channels / neurotransmitters → collective neural activity.

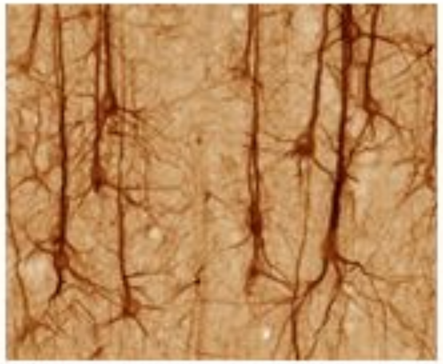
Winged Flight

inspiration →

Wright Brothers

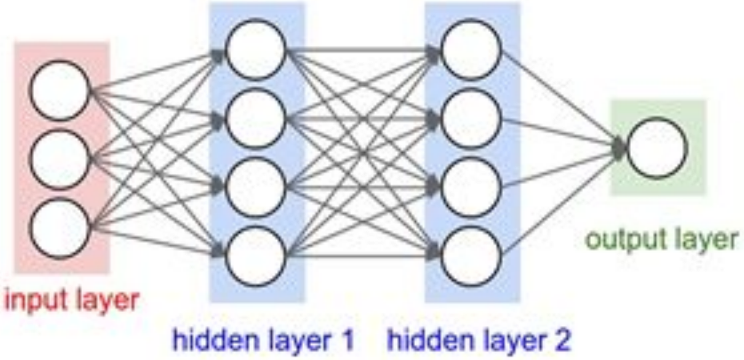
Modern Airplanes

Neural Information Processing



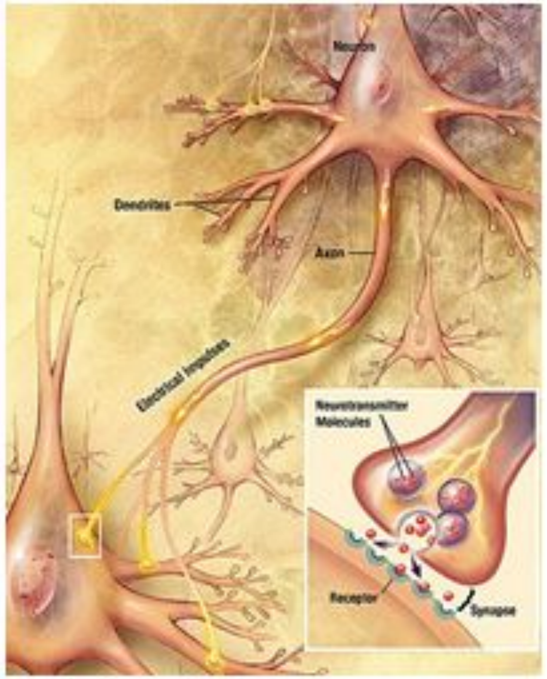
inspiration →

Artificial Neural Network (ANN)

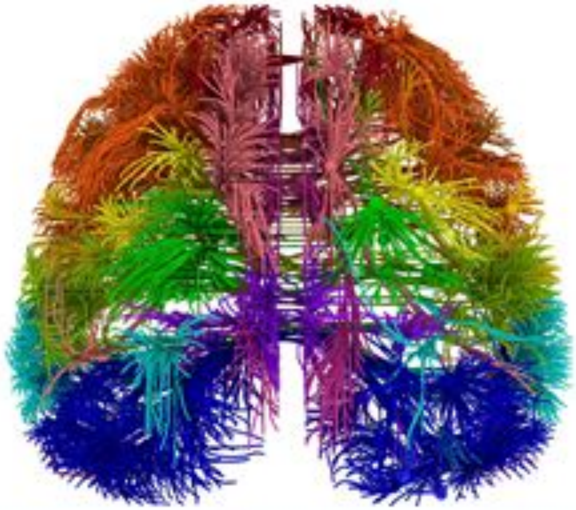


NN's are engineering tools inspired by nature. **Not a model of real neurons!**
Biology is more complex / temporal dynamics, refraction / other relevant factors.

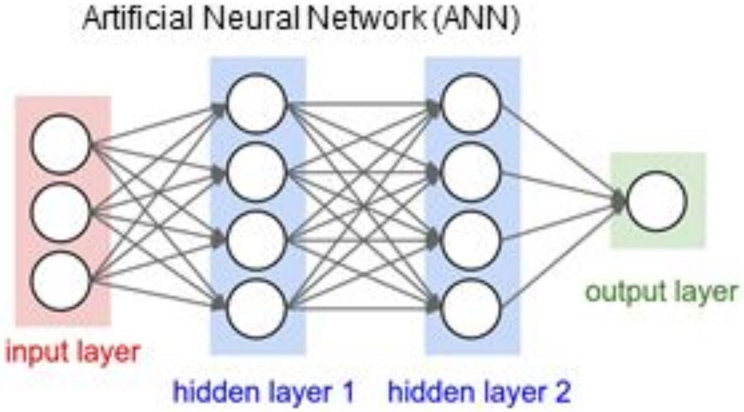
Neuron: Axons and Dendrites



Human Connectome Project

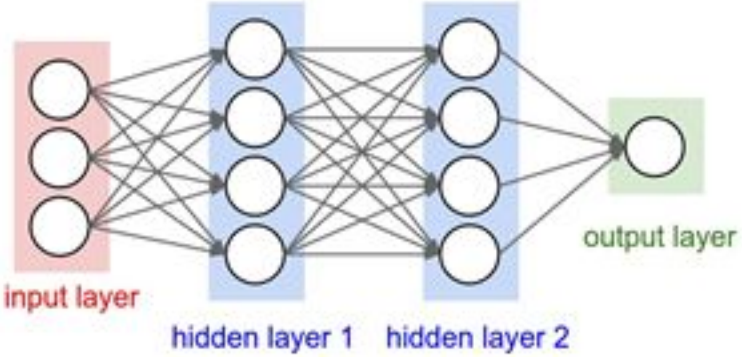


What are Neural Networks (NNs)?



What are Neural Networks (NNs)?

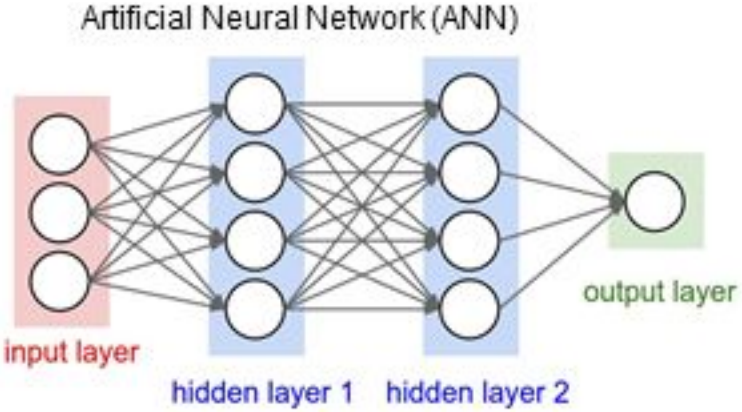
Artificial Neural Network (ANN)

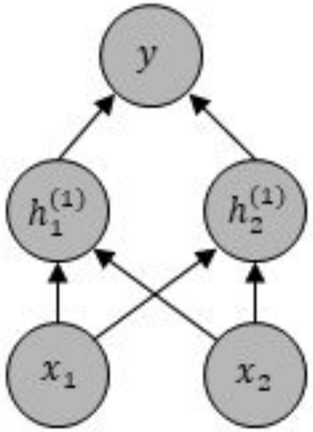


NN Transform: $y = f(\boldsymbol{x};\theta)$ where f is obtained by compositions $f^{(L)}\left(f^{(L-1)}(\ldots f^{(1)}(\boldsymbol{x})\ldots)\right)$ with $\boldsymbol{h}^k = f^{(k)}(\boldsymbol{h}^{k-1}) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + b^k)$.

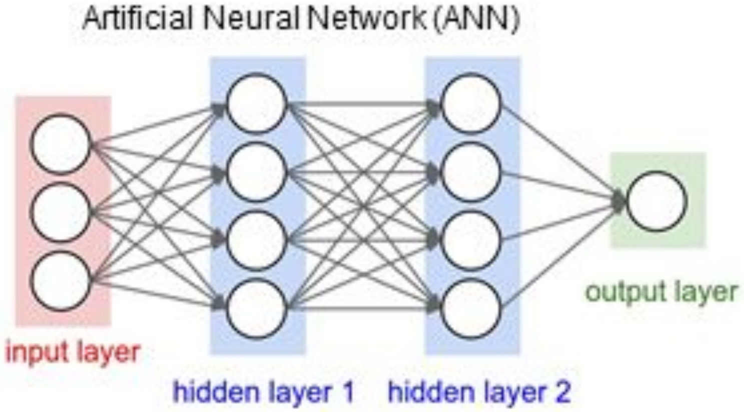
What are Neural Networks (NNs)?

NN Transform: $y = f(\boldsymbol{x};\theta)$ where f is obtained by compositions $f^{(L)}\left(f^{(L-1)}(\ldots f^{(1)}(\boldsymbol{x})\ldots)\right)$ with $\boldsymbol{h}^k = f^{(k)}\left(\boldsymbol{h}^{k-1}\right) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + b^k)$.





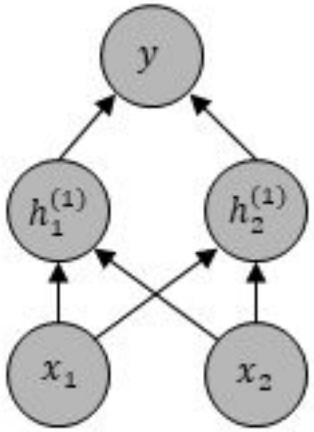
What are Neural Networks (NNs)?



NN Transform: $y = f(\boldsymbol{x};\theta)$ where f is obtained by compositions $f^{(L)}\left(f^{(L-1)}(\dots f^{(1)}(\boldsymbol{x})\dots)\right)$ with $\boldsymbol{h}^k = f^{(k)}(\boldsymbol{h}^{k-1}) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + b^k)$.

The $g(z)$ is called the **activation function.** Common choices include

- **Sigmoid σ**: $g(z) = 1/(1 + e^{-z})$.
- **Rectified Linear Unit (ReLu)**: $g(z) = \max(0, z)$.



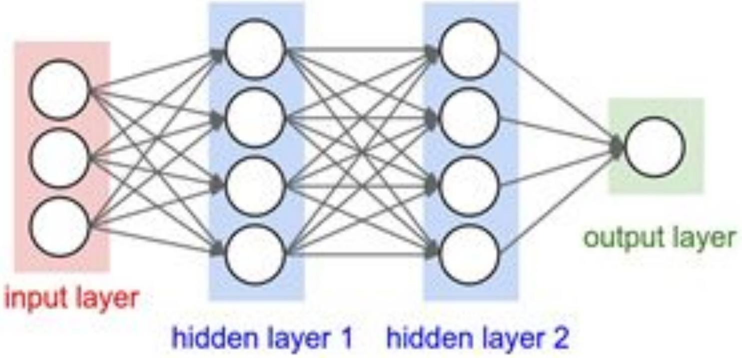
What are Neural Networks (NNs)?

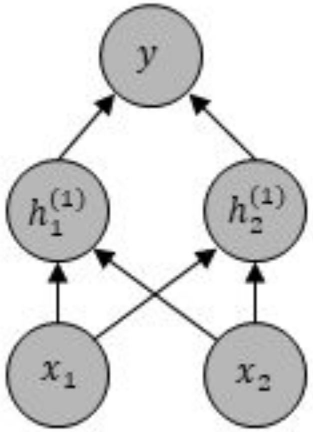
NN Transform: $y = f(\boldsymbol{x};\theta)$ where f is obtained by compositions $f^{(L)}\left(f^{(L-1)}\left(\ldots f^{(1)}\left(\boldsymbol{x}\right)\ldots\right)\right)$ with $\boldsymbol{h}^k = f^{(k)}\left(\boldsymbol{h}^{k-1}\right) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + b^k)$.

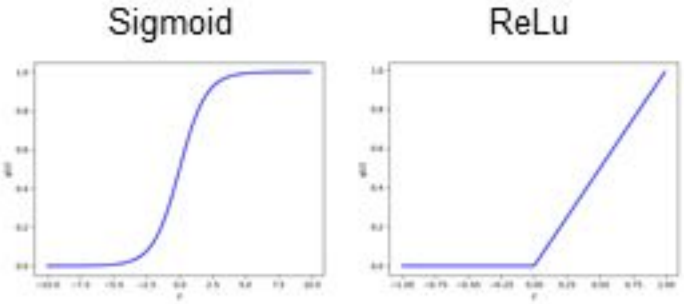
The $g(z)$ is called the **activation function.** Common choices include

- **Sigmoid σ**: $g(z) = 1/(1 + e^{-z})$.
- **Rectified Linear Unit (ReLu)**: $g(z) = \max(0, z)$.

Artificial Neural Network (ANN)







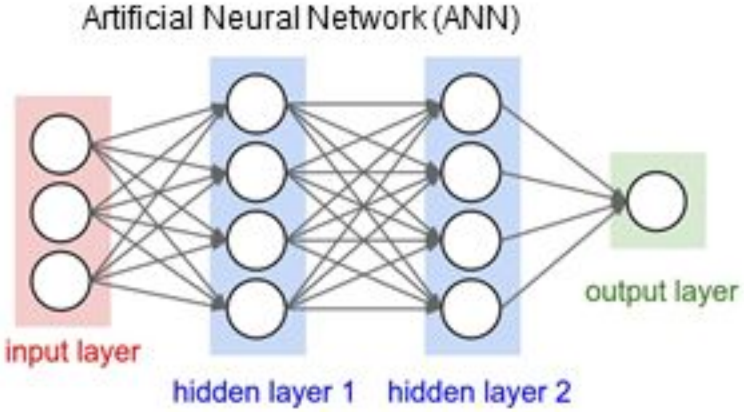
What are Neural Networks (NNs)?

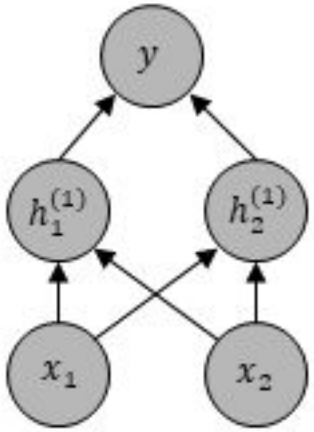
NN Transform: $y = f(\boldsymbol{x};\theta)$ where f is obtained by compositions $f^{(L)}\left(f^{(L-1)}(\ldots f^{(1)}(\boldsymbol{x})\ldots)\right)$ with $\boldsymbol{h}^k = f^{(k)}\left(\boldsymbol{h}^{k-1}\right) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + b^k)$.

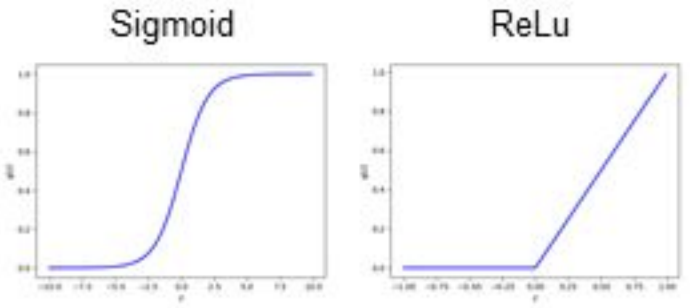
The $g(z)$ is called the **activation function.** Common choices include

- **Sigmoid σ**: $g(z) = 1/(1 + e^{-z})$.
- **Rectified Linear Unit (ReLu)**: $g(z) = \max(0,z)$.

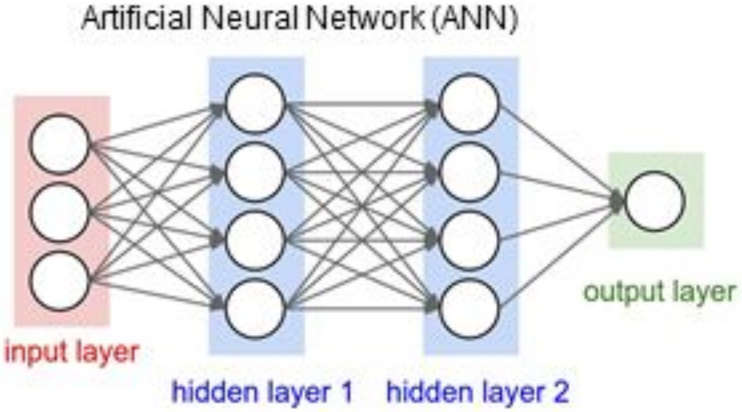
Find best weights W and **bias** b for each layer to minimize some **loss** $\ell(.,.)$.







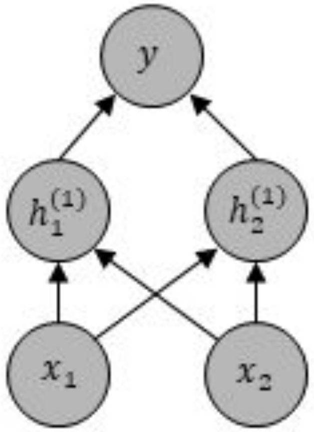
What are Neural Networks (NNs)?



NN Transform: $y = f(\boldsymbol{x};\theta)$ where f is obtained by compositions $f^{(L)}\left(f^{(L-1)}(\dots f^{(1)}(\boldsymbol{x})\dots)\right)$ with $\boldsymbol{h}^k = f^{(k)}(\boldsymbol{h}^{k-1}) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + b^k)$.

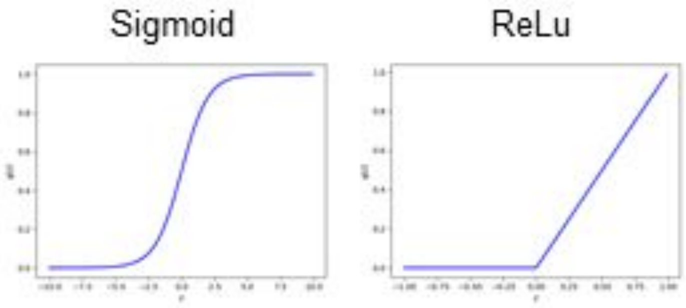
The $g(z)$ is called the **activation function.** Common choices include

- **Sigmoid σ:** $g(z) = 1/(1 + e^{-z})$.
- **Rectified Linear Unit (ReLu):** $g(z) = \max(0, z)$.

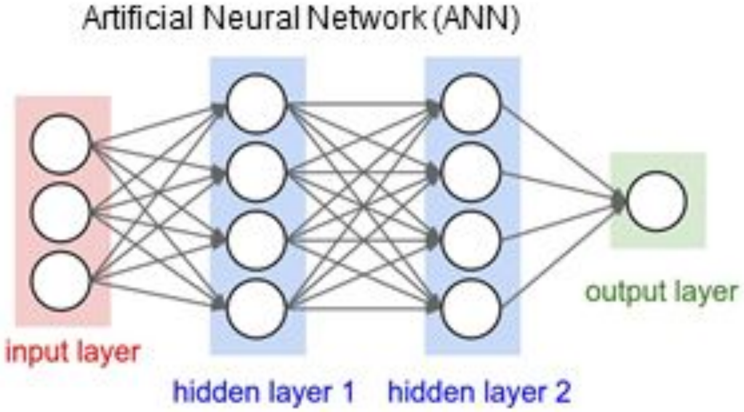


Find best **weights** W and **bias** b for each layer to minimize some **loss** $\ell(.,.)$.

Optimization non-convex so use local gradient learning methods.



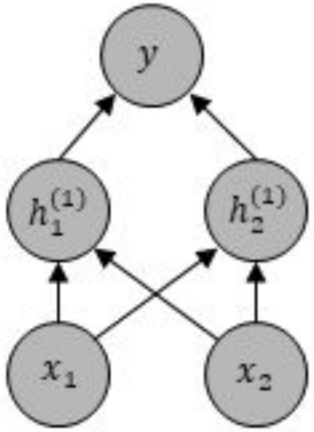
What are Neural Networks (NNs)?



NN Transform: $y = f(\boldsymbol{x};\theta)$ where f is obtained by compositions $f^{(L)}\left(f^{(L-1)}(\ldots f^{(1)}(\boldsymbol{x})\ldots)\right)$ with $\boldsymbol{h}^k = f^{(k)}\left(\boldsymbol{h}^{k-1}\right) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + b^k)$.

The $g(z)$ is called the **activation function.** Common choices include

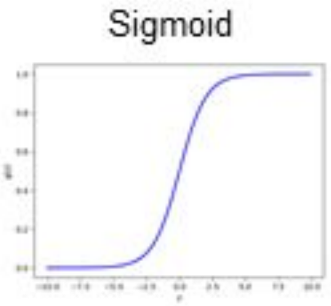
- **Sigmoid σ:** $g(z) = 1/(1 + e^{-z})$.
- **Rectified Linear Unit (ReLu):** $g(z) = \max(0, z)$.

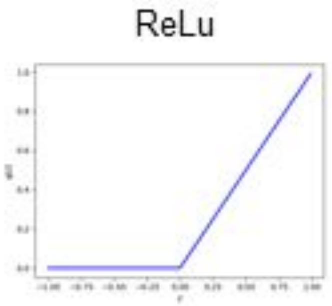


Find best weights W and **bias** b for each layer to minimize some **loss** $\ell(.,.)$.

Optimization non-convex so use local gradient learning methods.

Universal approximation of any smooth function y = f(x) just **two layers sufficient** for broad class of g(z). **However,** may need many hidden units in layer.





What are Neural Networks (NNs)?

NN Transform: $y = f(\boldsymbol{x};\theta)$ where f is obtained by compositions $f^{(L)}\left(f^{(L-1)}(\ldots f^{(1)}(\boldsymbol{x})\ldots)\right)$ with $\boldsymbol{h}^k = f^{(k)}(\boldsymbol{h}^{k-1}) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + b^k)$.

The $g(z)$ is called the **activation function.** Common choices include

- **Sigmoid σ:** $g(z) = 1/(1+e^{-z})$.
- **Rectified Linear Unit (ReLu):** $g(z) = \max(0,z)$.

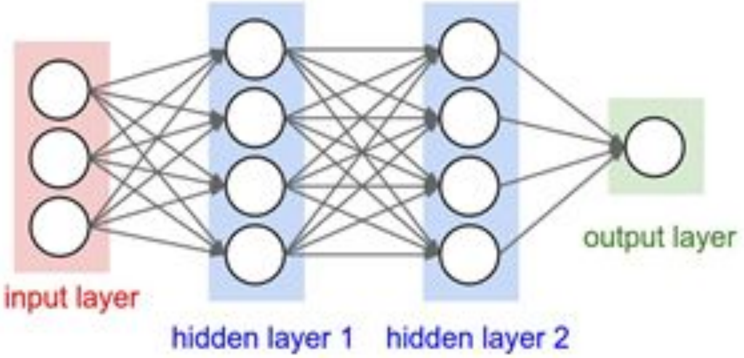
Find best weights W and **bias** b for each layer to minimize some **loss** $\ell(.,.)$.

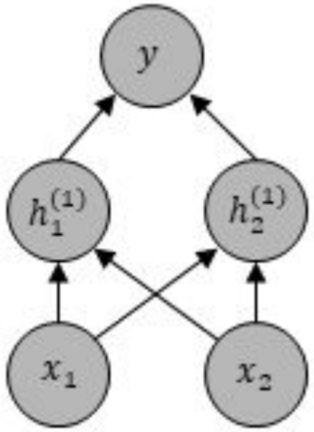
Optimization non-convex so use local gradient learning methods.

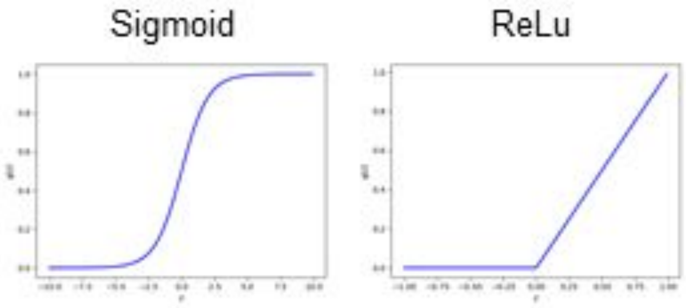
Universal approximation of any smooth function y = f(x) just **two layers sufficient** for broad class of g(z). **However,** may need many hidden units in layer.

Deep architectures often less hidden units needed (symmetries). However, training can be more challenging for deep architectures.

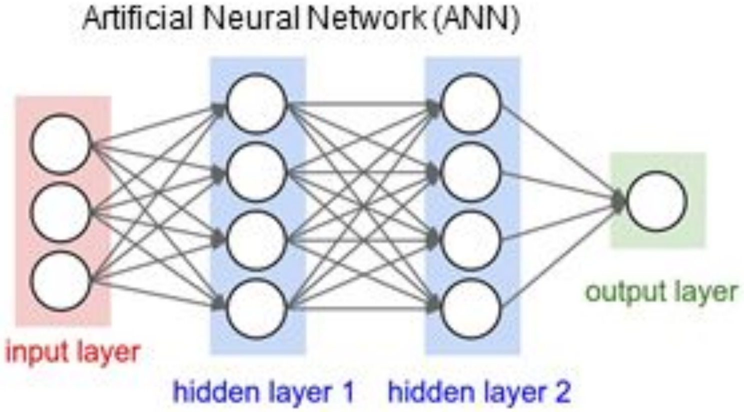
Artificial Neural Network (ANN)







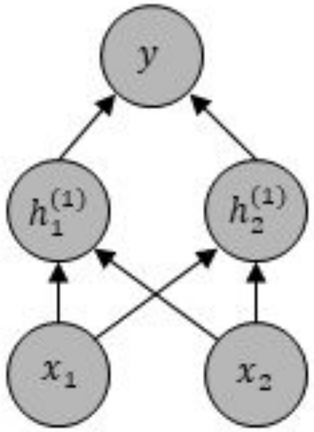
What are Neural Networks (NNs)?



NN Transform: $y = f(\boldsymbol{x};\theta)$ where f is obtained by compositions $f^{(L)}\left(f^{(L-1)}(\ldots f^{(1)}(\boldsymbol{x})\ldots)\right)$ with $\boldsymbol{h}^k = f^{(k)}\left(\boldsymbol{h}^{k-1}\right) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + b^k)$.

The $g(z)$ is called the **activation function.** Common choices include

- **Sigmoid σ:** $g(z) = 1/(1 + e^{-z})$.
- **Rectified Linear Unit (ReLu):** $g(z) = \max(0, z)$.

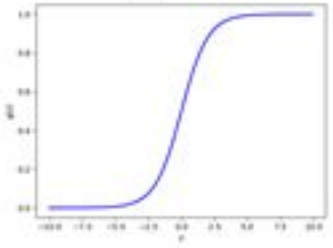


Find best weights W and **bias** b for each layer to minimize some **loss** $\ell(.,.)$.

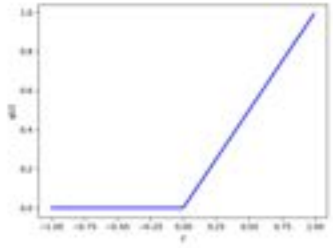
Optimization non-convex so use local gradient learning methods.

Universal approximation of any smooth function y = f(x) just **two layers sufficient** for broad class of g(z). **However,** may need many hidden units in layer.

Sigmoid



ReLu



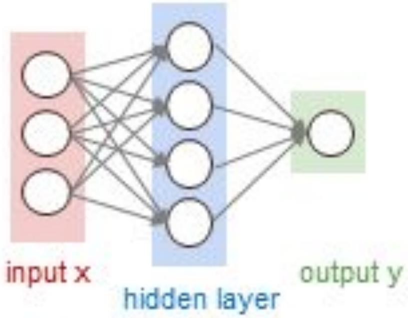
Deep architectures often less hidden units needed (symmetries). However, training can be more challenging for deep architectures.

Many possible choices for network architectures, depth, activation functions.

Universal Approximation Theorem

For many activation functions g(z) just two layers is sufficient for universal approximation of any continuous function y = f(x) on a compact set.

Neural Network: 1-Hidden Layer



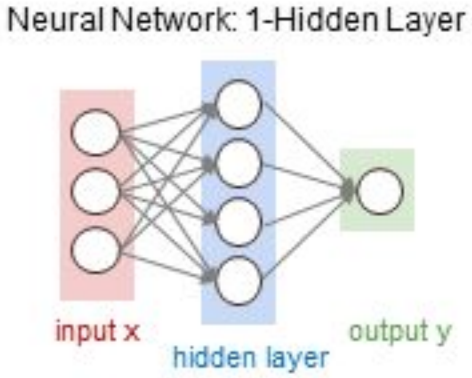
I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem

For many activation functions g(z) just two layers is sufficient for universal approximation of any continuous function y = f(x) on a compact set.

Definition: We say a subspace V **zeros-out** a measure μ. If for all $v \in V$, $\int v(x) d\mu(x) = 0$ holds then the measure must be zero $\mu \equiv 0$.

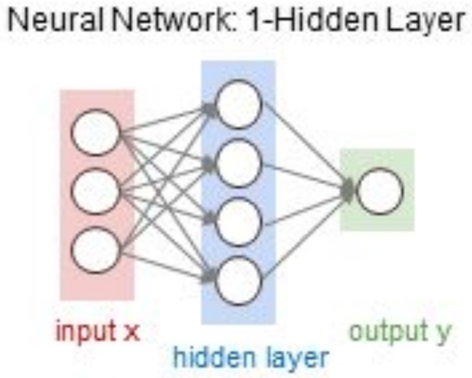
Definition: We say a function g(z) is **discriminatory** if for all $\boldsymbol{w}, b,$ we have $\int g(\boldsymbol{w}^T \boldsymbol{x} + b)\, d\mu(x) = 0$ then the measure must be zero $\mu \equiv 0$.



I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem

For many activation functions g(z) just two layers is sufficient for universal approximation of any continuous function y = f(x) on a compact set.



Definition: We say a subspace V **zeros-out** a measure μ. If for all $v \in V$, $\int v(x)d\mu(x) = 0$ holds then the measure must be zero $\mu \equiv 0$.

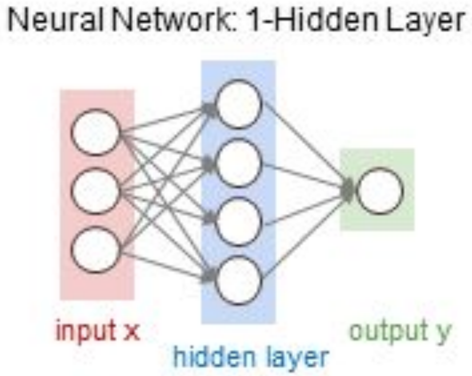
Definition: We say a function g(z) is **discriminatory** if for all $\boldsymbol{w}, b,$ we have $\int g(\boldsymbol{w}^T\boldsymbol{x} + b)\,d\mu(x) = 0$ then the measure must be zero $\mu \equiv 0$.

Lemma: The g(z) is discriminatory for Borel measures μ iff the subspace $V = \left\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j\, g\left(\boldsymbol{w}_j^T\boldsymbol{x} + b_j\right), N \in \mathbb{N}\right\}$ zeros-out the measures μ.

I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem

For many activation functions g(z) just two layers is sufficient for universal approximation of any continuous function y = f(x) on a compact set.



Definition: We say a subspace V **zeros-out** a measure μ. If for all $v \in V$, $\int v(x) d\mu(x) = 0$ holds then the measure must be zero $\mu \equiv 0$.

Definition: We say a function g(z) is **discriminatory** if for all $\mathbf{w}, b,$ we have $\int g(\mathbf{w}^T \mathbf{x} + b)\, d\mu(x) = 0$ then the measure must be zero $\mu \equiv 0$.

Lemma: The g(z) is discriminatory for Borel measures μ iff the subspace $V = \left\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j g\left(\mathbf{w}_j^T \mathbf{x} + b_j\right), N \in \mathbb{N}\right\}$ zeros-out the measures μ.

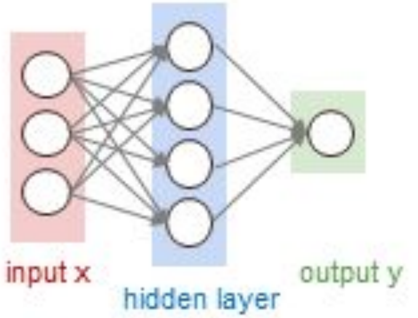
Theorem (Cybenko 1989): Let g(z) be a continuous activation function that generates a subspace $V = \left\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j g\left(\mathbf{w}_j^T \mathbf{x} + b_j\right), N \in \mathbb{N}\right\}$ that zeros-out all Borel measures on I_n, then V is dense in the space of continuous functions $C(\mathrm{I}_n)$.

I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem

For many activation functions g(z) just two layers is sufficient for universal approximation of any continuous function y = f(x) on a compact set.

Neural Network: 1-Hidden Layer



Definition: We say a subspace V **zeros-out** a measure μ. If for all $v \in V$, $\int v(x)\,d\mu(x) = 0$ holds then the measure must be zero $\mu \equiv 0$.

Definition: We say a function g(z) is **discriminatory** if for all $\mathbf{w}, b,$ we have $\int g(\mathbf{w}^T\mathbf{x} + b)\,d\mu(x) = 0$ then the measure must be zero $\mu \equiv 0$.

Lemma: The g(z) is discriminatory for Borel measures μ iff the subspace $V = \left\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j\, g\left(\mathbf{w}_j^T\mathbf{x} + b_j\right), N \in \mathbb{N}\right\}$ zeros-out the measures μ.

Theorem (Cybenko 1989): Let g(z) be a continuous activation function that generates a subspace $V = \left\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j\, g\left(\mathbf{w}_j^T\mathbf{x} + b_j\right), N \in \mathbb{N}\right\}$ that zeros-out all Borel measures on I_n, then V is dense in the space of continuous functions $C(\mathrm{I}_n)$.

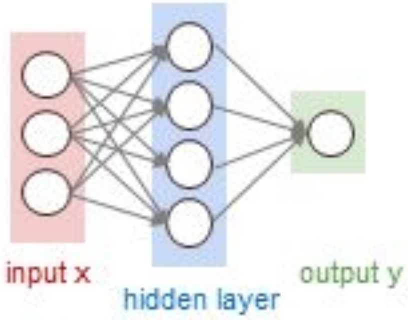
Remark: For any function $\mathrm{f} \in C(\mathrm{I}_n)$ and $\varepsilon > 0$ there exists $\mathrm{q} \in V$ with N and weights $\alpha_j, \mathbf{w}_j^T, b_j$ such that $|f(x) - q(x)| < \epsilon$ for all $\mathrm{x} \in \mathrm{I}_n$.

I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem

Theorem (Cybenko 1989): Let g(z) be a continuous activation function that generates a subspace $V =$ in the space$\left\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j g\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right), N \in \mathbb{N}\right\}$ that zeros-out all Borel measures on I_n, then V is dense of continuous functions $C(\mathrm{I}_n)$.

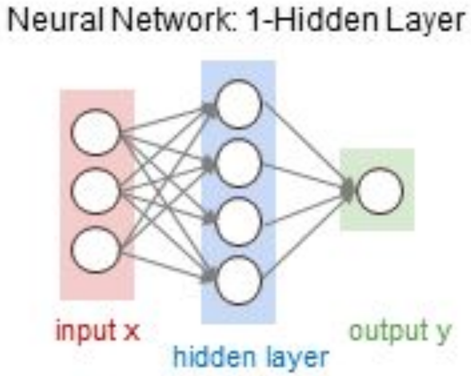
Neural Network: 1-Hidden Layer



I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem

Theorem (Cybenko 1989): Let g(z) be a continuous activation function that generates a subspace $V =$ in the space $\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j g\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right), N \in \mathbb{N}\}$ that zeros-out all Borel measures on I_n, then V is dense of continuous functions $C(\mathrm{I}_n)$.

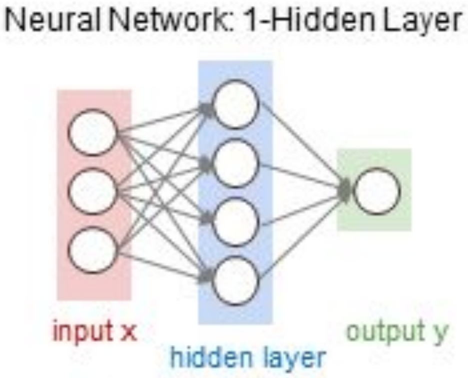


Remark: For any function $\mathrm{f} \in C(\mathrm{I}_n)$ and $\varepsilon > 0$ there exists $\mathrm{q} \in V$ with N and weights $\alpha_j, \boldsymbol{w}_j^T, b_j$ such that $|f(x) - q(x)| < \epsilon$ for all $\mathrm{x} \in \mathrm{I}_n$.

I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem

Theorem (Cybenko 1989): Let g(z) be a continuous activation function that generates a subspace $V =$ in the space$\left\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j g\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right), N \in \mathbb{N}\right\}$ that zeros-out all Borel measures on I_n, then V is dense of continuous functions $C(\mathrm{I}_n)$.


Neural Network: 1-Hidden Layer

Remark: For any function $\mathrm{f} \in C(\mathrm{I}_n)$ and $\varepsilon > 0$ there exists $\mathrm{q} \in V$ with N and weights $\alpha_j, \boldsymbol{w}_j^T, b_j$ such that $|f(x) - q(x)| < \epsilon$ for all $\mathrm{x} \in \mathrm{I}_n$.

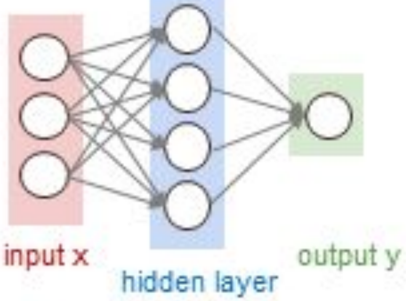
Example: Linear/affine activation functions are ***not discrimantory***, $g(z) = c_1 z + c_2$.

I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem

Theorem (Cybenko 1989): Let g(z) be a continuous activation function that generates a subspace $V =$ in the space$\left\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j g\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right), N \in \mathbb{N}\right\}$ that zeros-out all Borel measures on I_n, then V is dense of continuous functions $C(\mathrm{I}_n)$.

Neural Network: 1-Hidden Layer



Remark: For any function $\mathrm{f} \in C(\mathrm{I}_n)$ and $\varepsilon > 0$ there exists $\mathrm{q} \in V$ with N and weights $\alpha_j, \boldsymbol{w}_j^T, b_j$ such that $|f(x) - q(x)| < \epsilon$ for all $\mathrm{x} \in \mathrm{I}_n$.

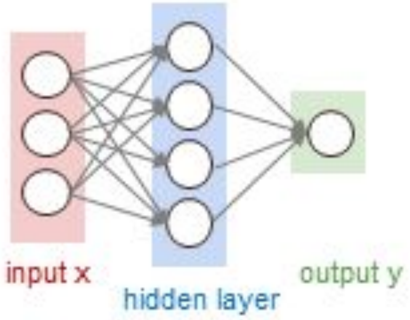
Example: Linear/affine activation functions are ***not discrimantory***, $g(z) = c_1 z + c_2$.

Follows from: $\mathrm{v}(x) = \sum_{j=1}^{N} \alpha_j g\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right) = \sum_{j=1}^{N} \alpha_j \left(c_1\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right) + c_2\right) = \boldsymbol{w}^T \boldsymbol{x} + b$, so $0 = \int v(x) d\mu(x) = \int (\boldsymbol{w}^T \boldsymbol{x} + b) d\mu(x) = \boldsymbol{w}^T \int \boldsymbol{x} d\mu(x) + b \int d\mu(x)$.

I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem

Theorem (Cybenko 1989): Let g(z) be a continuous activation function that generates a subspace $V =$ in the space$\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j g\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right), N \in \mathbb{N}\}$ that zeros-out all Borel measures on I_n, then V is dense of continuous functions $C(\mathrm{I}_n)$.


Neural Network: 1-Hidden Layer

Remark: For any function $\mathrm{f} \in C(\mathrm{I}_n)$ and $\varepsilon > 0$ there exists $\mathrm{q} \in V$ with N and weights $\alpha_j, \boldsymbol{w}_j^T, b_j$ such that $|f(x) - q(x)| < \epsilon$ for all $\mathrm{x} \in \mathrm{I}_n$.

Example: Linear/affine activation functions are ***not discrimantory***, $g(z) = c_1 z + c_2$.

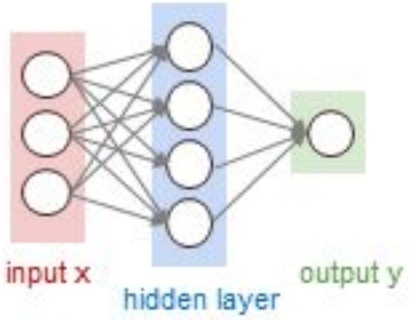
Follows from: $\mathrm{v}(x) = \sum_{j=1}^{N} \alpha_j g\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right) = \sum_{j=1}^{N} \alpha_j \left(c_1\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right) + c_2\right) = \boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x} + b$, so $0 = \int v(x) d\mu(x) = \int (\boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x} + b) d\mu(x) = \boldsymbol{w}^{\boldsymbol{T}} \int \boldsymbol{x} d\mu(x) + b \int d\mu(x)$. This can be made to hold if we can find a Borel measure μ so that both $\int d\mu(x) = 0$ and $\int \boldsymbol{x} d\mu(x) = 0$.

I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem

Theorem (Cybenko 1989): Let g(z) be a continuous activation function that generates a subspace $V =$ in the space $\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j g\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right), N \in \mathbb{N}\}$ that zeros-out all Borel measures on I_n, then V is dense of continuous functions $C(\mathrm{I}_n)$.

Neural Network: 1-Hidden Layer



Remark: For any function $\mathrm{f} \in C(\mathrm{I}_n)$ and $\varepsilon > 0$ there exists $\mathrm{q} \in V$ with N and weights $\alpha_j, \boldsymbol{w}_j^T, b_j$ such that $|f(x) - q(x)| < \epsilon$ for all $\mathrm{x} \in \mathrm{I}_n$.

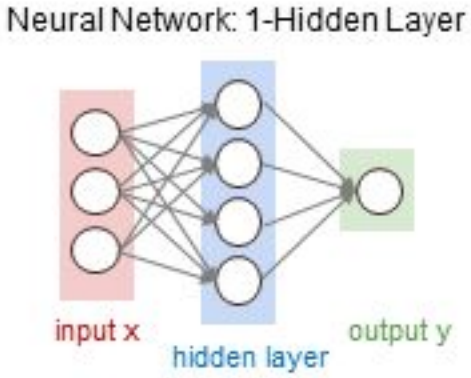
Example: Linear/affine activation functions are ***not discrimantory***, $g(z) = c_1 z + c_2$.

Follows from: $\mathrm{v}(x) = \sum_{j=1}^{N} \alpha_j g\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right) = \sum_{j=1}^{N} \alpha_j \left(c_1\left(\boldsymbol{w}_j^T \boldsymbol{x} + b_j\right) + c_2\right) = \boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x} + b$, so $0 = \int v(x) d\mu(x) = \int (\boldsymbol{w}^{\boldsymbol{T}} \boldsymbol{x} + b) d\mu(x) = \boldsymbol{w}^{\boldsymbol{T}} \int \boldsymbol{x} d\mu(x) + b \int d\mu(x)$. This can be made to hold if we can find a Borel measure μ so that both $\int d\mu(x) = 0$ and $\int \boldsymbol{x} d\mu(x) = 0$. Let $\mu(x) = a_1 \delta(x_1 - r_1) + a_2 \delta(x_1 - r_2) + a_3 \delta(x_1 - r_2) = -\delta(x_1) + 2\delta\left(x_1 - \frac{1}{2}\right) - \delta(x_1 - 1)$.

I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem

Theorem (Cybenko 1989): Let g(z) be a continuous activation function that generates a subspace $V =$ in the space$\left\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j g\left(\mathbf{w}_j^T \mathbf{x} + b_j\right), N \in \mathbb{N}\right\}$ that zeros-out all Borel measures on I_n, then V is dense of continuous functions $C(\mathrm{I}_n)$.



Remark: For any function $f \in C(\mathrm{I}_n)$ and $\varepsilon > 0$ there exists $q \in V$ with N and weights $\alpha_j, \mathbf{w}_j^T, b_j$ such that $|f(x) - q(x)| < \epsilon$ for all $x \in \mathrm{I}_n$.

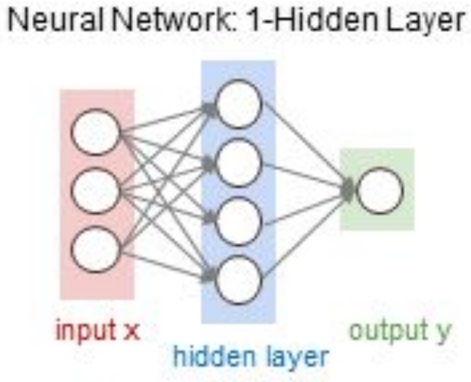
Example: Linear/affine activation functions are ***not discrimantory***, $g(z) = c_1 z + c_2$.

Follows from: $\mathrm{v}(x) = \sum_{j=1}^{N} \alpha_j g\left(\mathbf{w}_j^T \mathbf{x} + b_j\right) = \sum_{j=1}^{N} \alpha_j \left(c_1\left(\mathbf{w}_j^T \mathbf{x} + b_j\right) + c_2\right) = \mathbf{w}^T \mathbf{x} + b$, so $0 = \int v(x) d\mu(x) = \int (\mathbf{w}^T \mathbf{x} + b) d\mu(x) = \mathbf{w}^T \int \mathbf{x} d\mu(x) + b \int d\mu(x)$. This can be made to hold if we can find a Borel measure μ so that both $\int d\mu(x) = 0$ and $\int \mathbf{x} d\mu(x) = 0$. Let $\mu(x) = a_1 \delta(x_1 - r_1) + a_2 \delta(x_1 - r_2) + a_3 \delta(x_1 - r_2) = -\delta(x_1) + 2\delta\left(x_1 - \frac{1}{2}\right) - \delta(x_1 - 1)$.

Example: ReLU activations generate subspace that zeros-out Borel measures, so ReLU-NN's have the universal approximation property.

I_n is the unit cube in $\mathbb{R}^n$

Universal Approximation Theorem



Theorem (Cybenko 1989): Let g(z) be a continuous activation function that generates a subspace $V =$ in the space$\{q \mid q(x) = \sum_{j=1}^{N} \alpha_j g(\mathbf{w}_j^T \mathbf{x} + b_j), N \in \mathbb{N}\}$ that zeros-out all Borel measures on I_n, then V is dense of continuous functions $C(\mathrm{I}_n)$.

Remark: For any function $\mathrm{f} \in C(\mathrm{I}_n)$ and $\varepsilon > 0$ there exists $\mathrm{q} \in V$ with N and weights $\alpha_j, \mathbf{w}_j^T, b_j$ such that $|f(x) - q(x)| < \epsilon$ for all $\mathrm{x} \in \mathrm{I}_n$.

Example: Linear/affine activation functions are ***not discrimantory***, $g(z) = c_1 z + c_2$.

Follows from: $\mathrm{v}(x) = \sum_{j=1}^{N} \alpha_j g(\mathbf{w}_j^T \mathbf{x} + b_j) = \sum_{j=1}^{N} \alpha_j (c_1(\mathbf{w}_j^T \mathbf{x} + b_j) + c_2) = \mathbf{w}^T \mathbf{x} + b$, so $0 = \int v(x) d\mu(x) = \int (\mathbf{w}^T \mathbf{x} + b) d\mu(x) = \mathbf{w}^T \int \mathbf{x} d\mu(x) + b \int d\mu(x)$. This can be made to hold if we can find a Borel measure μ so that both $\int d\mu(x) = 0$ and $\int \mathbf{x} d\mu(x) = 0$. Let $\mu(x) = a_1 \delta(x_1 - r_1) + a_2 \delta(x_1 - r_2) + a_3 \delta(x_1 - r_2) = -\delta(x_1) + 2\delta\left(x_1 - \frac{1}{2}\right) - \delta(x_1 - 1)$.

Example: ReLU activations generate subspace that zeros-out Borel measures, so ReLU-NN's have the universal approximation property.

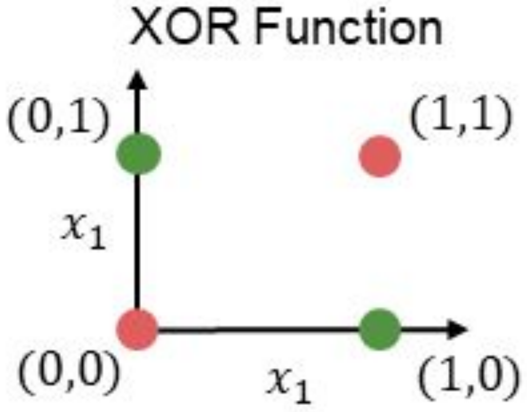
ReLU-NN's include all pieces-wise linear approximations, realizable with enough hidden nodes, also provides another way to prove the universal approximation.

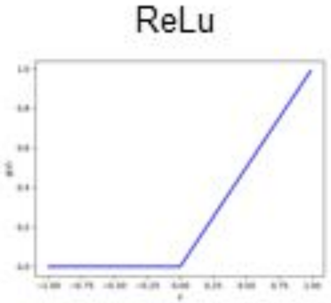
I_n is the unit cube in $\mathbb{R}^n$

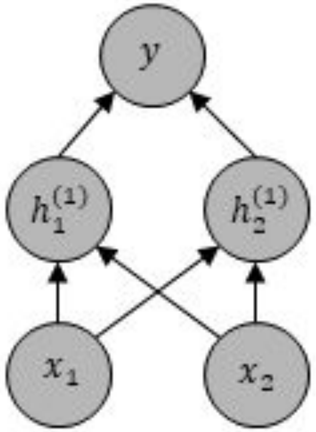
Neural Networks: Example

NN Transform: $y = f(\boldsymbol{x};\theta)$ from compositions $\boldsymbol{h}^k = f^{(k)}\left(\boldsymbol{h}^{k-1}\right) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + \ b^k)$.

Example: Compute the XOR function $y = f(\boldsymbol{x}) = x_1 \oplus x_2$.







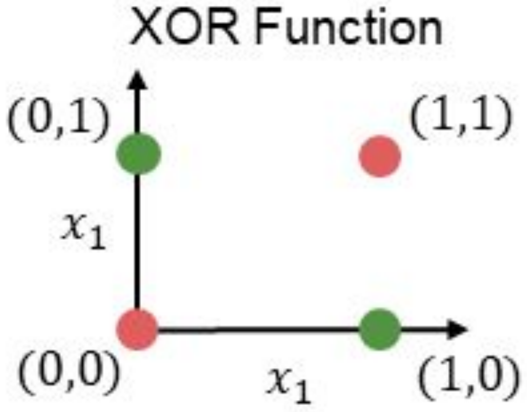
Neural Networks: Example

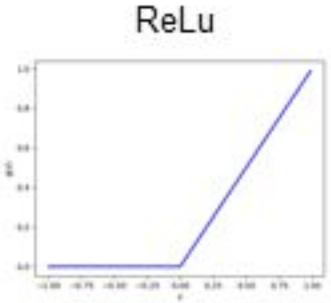
NN Transform: $y = f(\boldsymbol{x};\theta)$ from compositions $\boldsymbol{h}^k = f^{(k)}\left(\boldsymbol{h}^{k-1}\right) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + \ b^k)$.

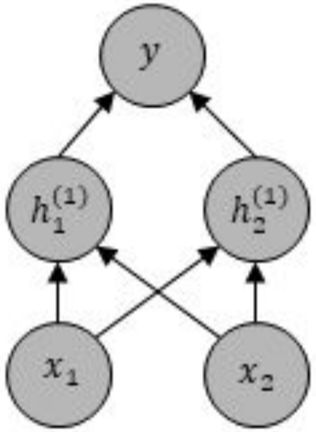
Example: Compute the XOR function $y = f(\boldsymbol{x}) = x_1 \oplus x_2$.

Linear models are insufficient $h(x) = \mathrm{sign}(\boldsymbol{x}^{\boldsymbol{T}}W \ + \ b)$, no choice W, b works.

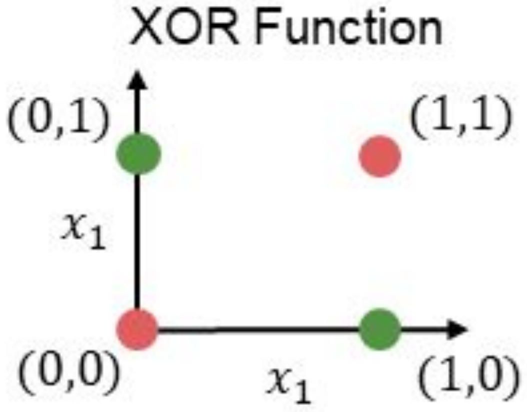
Non-linearity important to approximate functions such as XOR.







Neural Networks: Example



NN Transform: $y = f(\boldsymbol{x};\theta)$ from compositions $\boldsymbol{h}^k = f^{(k)}\left(\boldsymbol{h}^{k-1}\right) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + \ b^k)$.

Example: Compute the XOR function $y = f(\boldsymbol{x}) = x_1 \oplus x_2$.

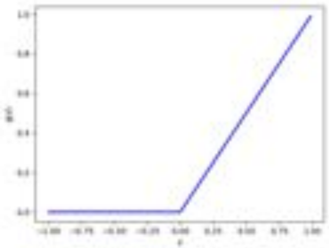
Linear models are insufficient $h(x) = \mathrm{sign}(\boldsymbol{x}^T W \ + \ b)$, no choice W, b works.

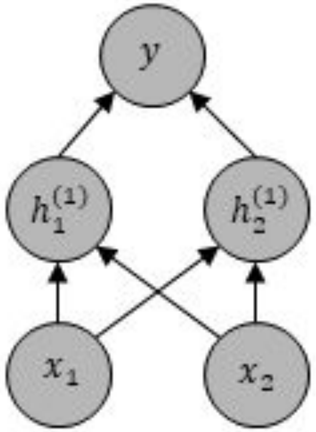
Non-linearity important to approximate functions such as XOR.

NN with two layers using non-linear ReLU activation g yields

$$\tilde{y} = f(\boldsymbol{x};\theta) = \big(\max\big(0, \boldsymbol{x}^T W^{(1)} + b^{(1)}\big\}\big)^T W^{(2)} + b^{(2)}$$

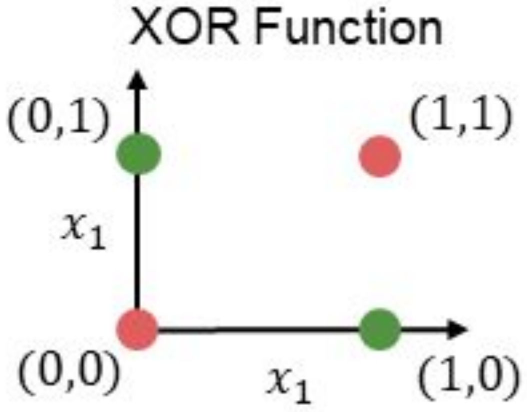
ReLu





Find parameters $W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$ to try to obtain correct classification $y = \mathrm{sign}(\tilde{y})$.

Neural Networks: Example



NN Transform: $y = f(\boldsymbol{x};\theta)$ from compositions $\boldsymbol{h}^k = f^{(k)}\left(\boldsymbol{h}^{k-1}\right) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + \ b^k)$.

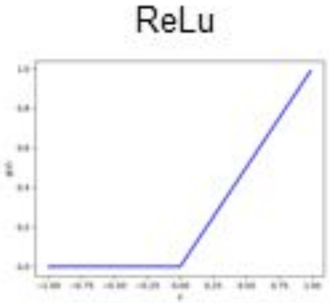
Example: Compute the XOR function $y = f(\boldsymbol{x}) = x_1 \oplus x_2$.

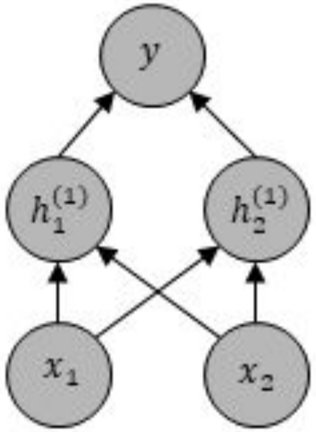
Linear models are insufficient $h(x) = \mathrm{sign}(\boldsymbol{x}^T W \ + \ b)$, no choice W, b works.

Non-linearity important to approximate functions such as XOR.

NN with two layers using non-linear ReLU activation g yields

$$\tilde{y} = f(\boldsymbol{x};\theta) = \left(\max\left(0, \boldsymbol{x}^T W^{(1)} + b^{(1)}\right\}\right)^T W^{(2)} + b^{(2)}$$

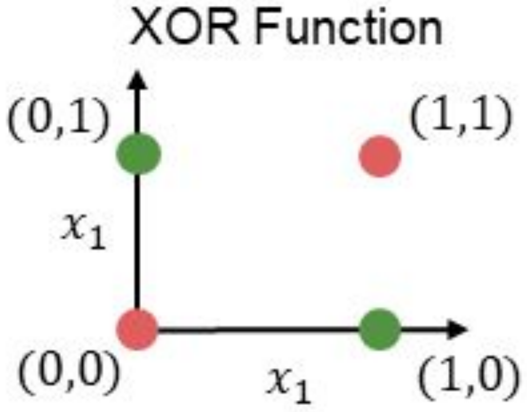




Find parameters $W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$ to try to obtain correct classification $y = \mathrm{sign}(\tilde{y})$.

$$W^{(1)} = \begin{bmatrix} +1 & +1 \\ +1 & +1 \end{bmatrix},\ b^{(1)} = [0,-1],\ W^{(2)} = \begin{bmatrix} 2 \\ -4 \end{bmatrix},\ b^{(2)} = [-1].$$

Neural Networks: Example

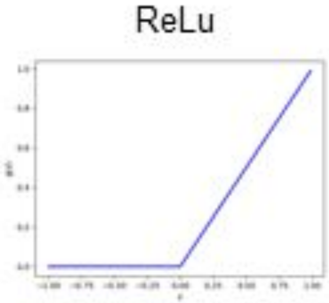


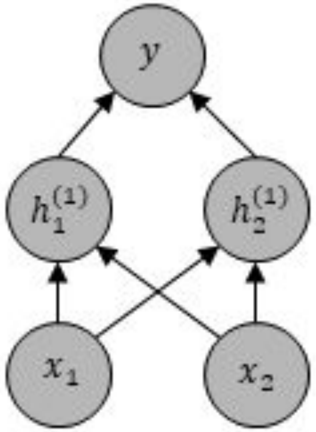
NN Transform: $y = f(\boldsymbol{x};\theta)$ from compositions $\boldsymbol{h}^k = f^{(k)}(\boldsymbol{h}^{k-1}) = g(\boldsymbol{h}^{k-1}\mathrm{W}^k + b^k)$.

Example: Compute the XOR function $y = f(\boldsymbol{x}) = x_1 \oplus x_2$.

Linear models are insufficient $h(x) = \mathrm{sign}(\boldsymbol{x}^T W + b)$, no choice W, b works.

Non-linearity important to approximate functions such as XOR.





NN with two layers using non-linear ReLU activation g yields

$$\tilde{y} = f(\boldsymbol{x};\theta) = \left(\max\left(0, \boldsymbol{x}^T W^{(1)} + b^{(1)}\}\right)\right)^T W^{(2)} + b^{(2)}$$

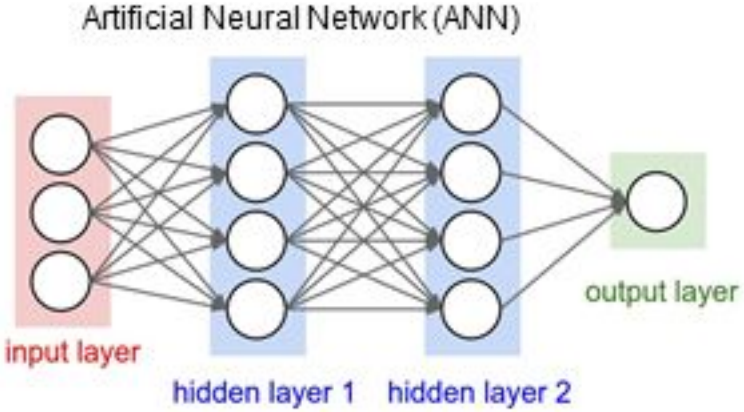
Find parameters $W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)}$ to try to obtain correct classification $y = \mathrm{sign}(\tilde{y})$.

$$W^{(1)} = \begin{bmatrix} +1 & +1 \\ +1 & +1 \end{bmatrix},\ b^{(1)} = [0,-1],\ W^{(2)} = \begin{bmatrix} 2 \\ -4 \end{bmatrix},\ b^{(2)} = [-1].$$

In general, we need methods to learn from data such weights to minimize a loss function.

Learning with Neural Networks

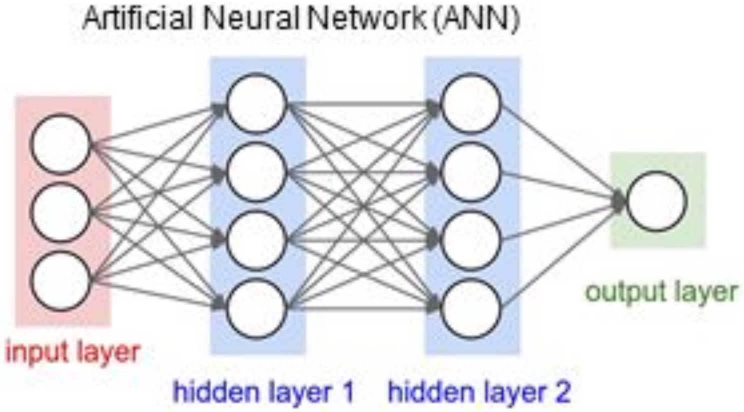
How do we find weights of Neural Networks?



Optimization Problem:

$$\min_{\theta} L(\theta;\{\boldsymbol{x}_i, y_i\}) = E_{\boldsymbol{x},y\sim\tilde{D}_{data}}\left[\ell\big(y, f(\boldsymbol{x};\theta)\big)\right]$$

How do we find weights of Neural Networks?

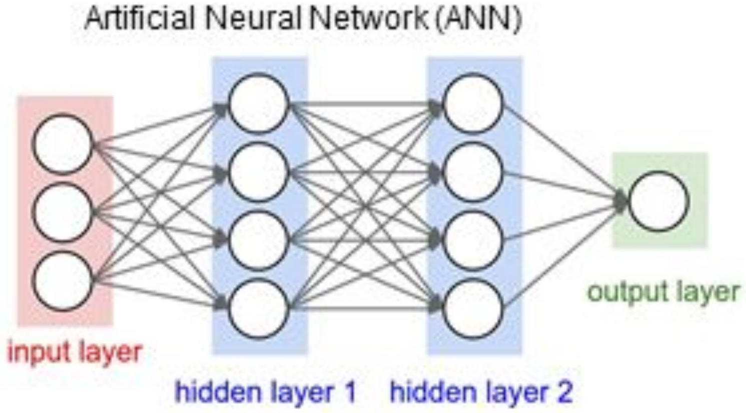


Optimization Problem:

$$\min_{\theta} L(\theta; \{\boldsymbol{x}_i, y_i\}) = E_{\boldsymbol{x},y \sim \tilde{D}_{data}} \left[\ell\left(y, f(\boldsymbol{x};\theta)\right)\right]$$

Non-convex problems typically have non-unique solutions and many local minima.

How do we find weights of Neural Networks?



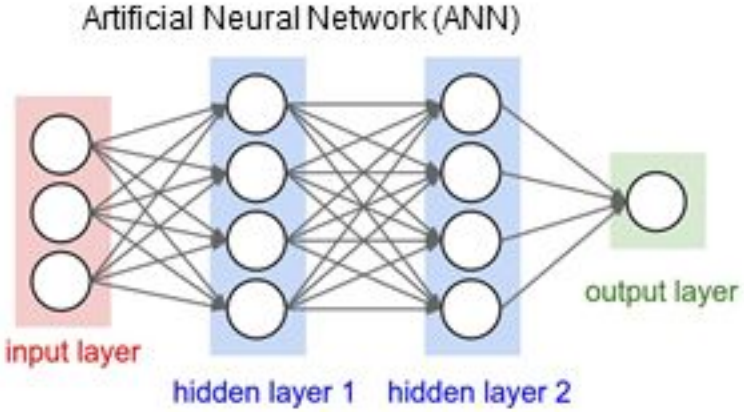
Optimization Problem:

$$\min_{\theta} L(\theta; \{\boldsymbol{x}_i, y_i\}) = E_{\boldsymbol{x},y \sim \tilde{D}_{data}} \left[\ell\left(y, f(\boldsymbol{x};\theta)\right)\right]$$

Non-convex problems typically have non-unique solutions and many local minima.

Goal is to find sets of parameters with small loss. Gradient-based methods can be used.

How do we find weights of Neural Networks?



Optimization Problem:

$$\min_{\theta} L(\theta;\{\boldsymbol{x}_i,y_i\}) = E_{\boldsymbol{x},y\sim\tilde{\mathcal{D}}_{data}}\left[\ell\big(y,f(\boldsymbol{x};\theta)\big)\right]$$

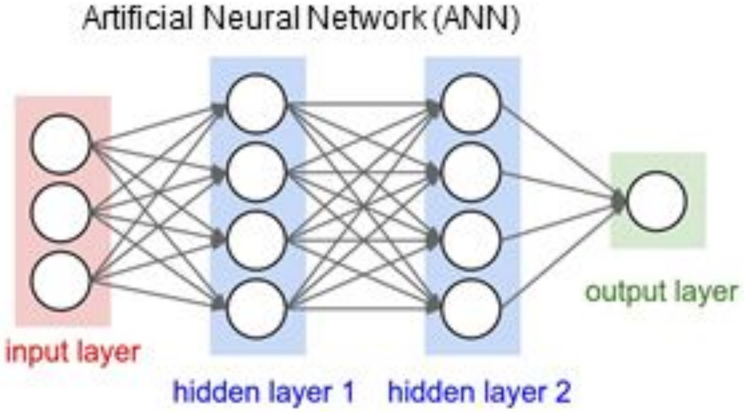
Non-convex problems typically have non-unique solutions and many local minima.

Goal is to find sets of parameters with small loss. Gradient-based methods can be used.

Stochastic Gradient Descent:

$\theta^{n+1} = \theta^n - \alpha\nabla_\theta Q^n(\theta^n)$, with $Q^n(\theta^n) = Q^n(\boldsymbol{X};\theta^n) = \frac{1}{m_b}\sum_{k=1}^{m_b}\ell\left(y_{i_k}, f\left(\boldsymbol{x}_{\boldsymbol{i_k}};\theta^n\right)\right)$.

How do we find weights of Neural Networks?



Optimization Problem:

$$\min_{\theta} L(\theta; \{\boldsymbol{x}_i, y_i\}) = E_{\boldsymbol{x},y \sim \tilde{\mathcal{D}}_{data}} \left[\ell\left(y, f(\boldsymbol{x}; \theta)\right)\right]$$

Non-convex problems typically have non-unique solutions and many local minima.

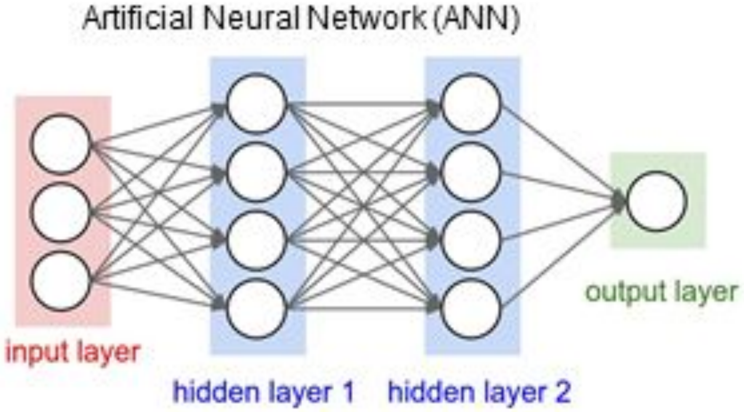
Goal is to find sets of parameters with small loss. Gradient-based methods can be used.

Stochastic Gradient Descent:

$$\theta^{n+1} = \theta^n - \alpha \nabla_\theta Q^n(\theta^n), \text{ with } Q^n(\theta^n) = Q^n(\boldsymbol{X}; \theta^n) = \frac{1}{m_b} \sum_{k=1}^{m_b} \ell\left(y_{i_k}, f\left(\boldsymbol{x}_{\boldsymbol{i_k}}; \theta^n\right)\right).$$

A subset (batch) of the available data is used of size m_b to estimate expected loss, (provides regularization).

How do we find weights of Neural Networks?



Optimization Problem:

$$\min_{\theta} L(\theta; \{\boldsymbol{x}_i, y_i\}) = E_{\boldsymbol{x},y \sim \tilde{\mathcal{D}}_{data}} \left[\ell\big(y, f(\boldsymbol{x}; \theta)\big)\right]$$

Non-convex problems typically have non-unique solutions and many local minima.

Goal is to find sets of parameters with small loss. Gradient-based methods can be used.

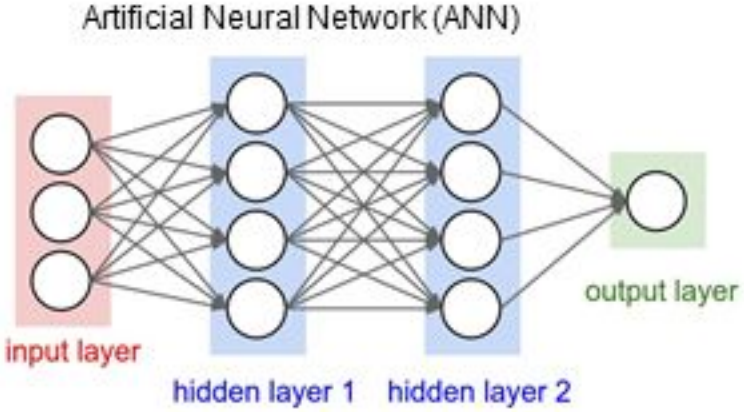
Stochastic Gradient Descent:

$\theta^{n+1} = \theta^n - \alpha \nabla_\theta Q^n(\theta^n)$, with $Q^n(\theta^n) = Q^n(\boldsymbol{X}; \theta^n) = \frac{1}{m_b} \sum_{k=1}^{m_b} \ell\left(y_{i_k}, f\left(\boldsymbol{x}_{\boldsymbol{i_k}}; \theta^n\right)\right)$.

A subset (batch) of the available data is used of size m_b to estimate expected loss, (provides regularization).

Still needs computation of gradients $\nabla_\theta f(x; \theta)$.

How do we find weights of Neural Networks?



Optimization Problem:

$$\min_{\theta} L(\theta;\{\boldsymbol{x}_i, y_i\}) = E_{\boldsymbol{x},y\sim\tilde{D}_{data}}\left[\ell\big(y, f(\boldsymbol{x};\theta)\big)\right]$$

Non-convex problems typically have non-unique solutions and many local minima.

Goal is to find sets of parameters with small loss. Gradient-based methods can be used.

Stochastic Gradient Descent:

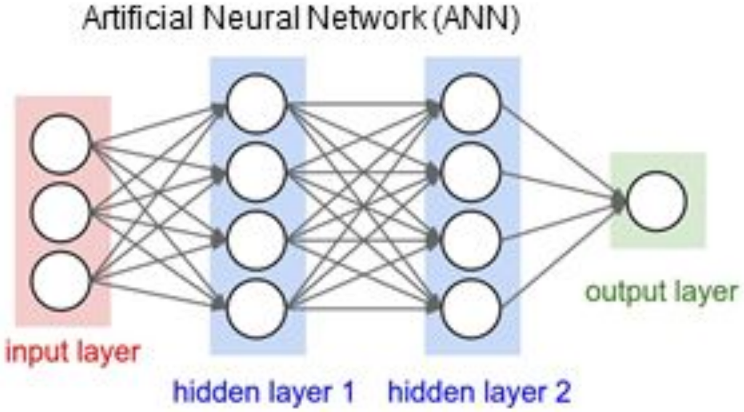
$\theta^{n+1} = \theta^n - \alpha\nabla_\theta Q^n(\theta^n)$, with $Q^n(\theta^n) = Q^n(\boldsymbol{X};\theta^n) = \frac{1}{m_b}\sum_{k=1}^{m_b}\ell\left(y_{i_k}, f\left(\boldsymbol{x}_{\boldsymbol{i_k}};\theta^n\right)\right)$.

A subset (batch) of the available data is used of size m_b to estimate expected loss, (provides regularization).

Still needs computation of gradients $\nabla_\theta f(x;\theta)$.

Analytically straight-forward to compute by chain-rule, but naïve evaluation is **computationally expensive.**

How do we find weights of Neural Networks?



Optimization Problem:

$$\min_{\theta} L(\theta; \{\boldsymbol{x}_i, y_i\}) = E_{\boldsymbol{x},y \sim \tilde{\mathcal{D}}_{data}} \left[\ell\left(y, f(\boldsymbol{x}; \theta)\right)\right]$$

Non-convex problems typically have non-unique solutions and many local minima.

Goal is to find sets of parameters with small loss. Gradient-based methods can be used.

Stochastic Gradient Descent:

$$\theta^{n+1} = \theta^n - \alpha \nabla_\theta Q^n(\theta^n), \text{ with } Q^n(\theta^n) = Q^n(\boldsymbol{X}; \theta^n) = \frac{1}{m_b} \sum_{k=1}^{m_b} \ell\left(y_{i_k}, f\left(\boldsymbol{x}_{\boldsymbol{i_k}}; \theta^n\right)\right).$$

A subset (batch) of the available data is used of size m_b to estimate expected loss, (provides regularization).

Still needs computation of gradients $\nabla_\theta f(x; \theta)$.

Analytically straight-forward to compute by chain-rule, but naïve evaluation is **computationally expensive.**

Automatic differentiation used in practice called **back-propagation.**

Computational Graphs and Back-Propagation

Computational Graphs

Optimization methods often need gradients $\nabla_\theta f(\boldsymbol{X};\theta)$.

Symbolic representations of $f(\theta)$ useful for automatic differentiation to obtain $\nabla_\theta f$.

Computational graph represents function evaluation in terms of more basic operations.

Computational Graphs

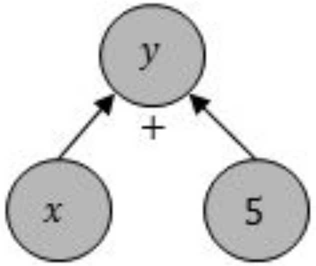
Optimization methods often need gradients $\nabla_\theta f(\boldsymbol{X};\theta)$.

Symbolic representations of $f(\theta)$ useful for automatic differentiation to obtain $\nabla_\theta f$.

Computational graph represents function evaluation in terms of more basic operations.

Example: $\mathrm{y} = f(x) = \mathrm{x} + 5$.

Computational Graphs



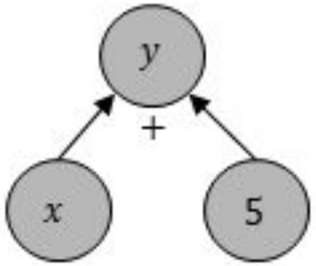
Optimization methods often need gradients $\nabla_\theta f(\boldsymbol{X};\theta)$.

Symbolic representations of $f(\theta)$ useful for automatic differentiation to obtain $\nabla_\theta f$.

Computational graph represents function evaluation in terms of more basic operations.

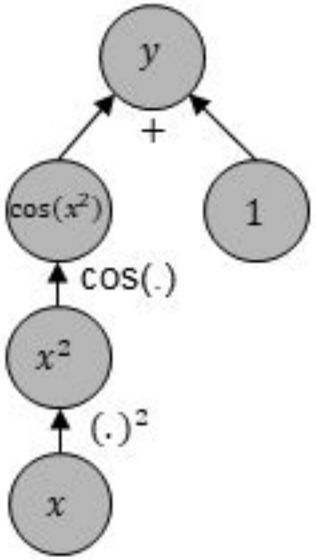
Example: $\mathrm{y} = f(x) = \mathrm{x} + 5$.

Computational Graphs



Optimization methods often need gradients $\nabla_\theta f(\boldsymbol{X}; \theta)$.

Symbolic representations of $f(\theta)$ useful for automatic differentiation to obtain $\nabla_\theta f$.

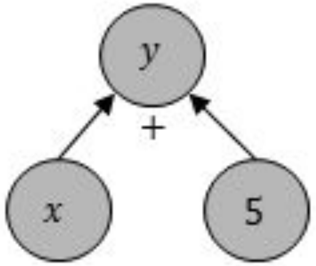


Computational graph represents function evaluation in terms of more basic operations.

Example: $\mathrm{y} = f(x) = \mathrm{x} + 5.$

Example: $\mathrm{y} = f(x) = \cos(x^2) + 1.$

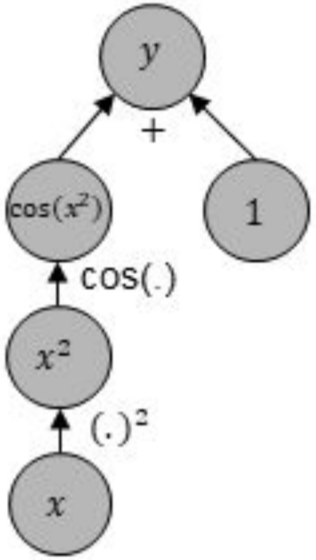
Computational Graphs



Optimization methods often need gradients $\nabla_\theta f(\boldsymbol{X};\theta)$.

Symbolic representations of $f(\theta)$ useful for automatic differentiation to obtain $\nabla_\theta f$.

Computational graph represents function evaluation in terms of more basic operations.



Example: $\mathrm{y} = f(x) = \mathrm{x} + 5.$

Example: $\mathrm{y} = f(x) = \cos(x^2) + 1.$

Example: $y = f(x_1, x_2) = w_1^{(2)} g(w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2) + w_2^{(2)} g(w_{2,1}^{(1)} x_1 + w_{2,2}^{(1)} x_2)$

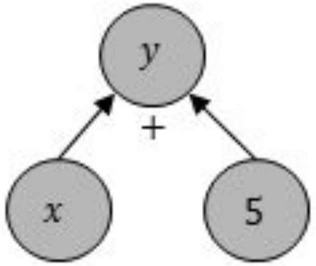
Computational Graphs

Optimization methods often need gradients $\nabla_\theta f(\boldsymbol{X};\theta)$.

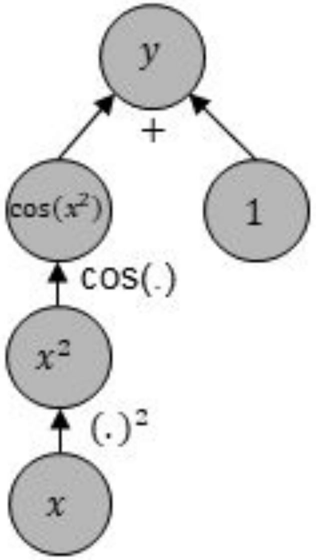
Symbolic representations of $f(\theta)$ useful for automatic differentiation to obtain $\nabla_\theta f$.

Computational graph represents function evaluation in terms of more basic operations.

Example: $\mathrm{y} = f(x) = \mathrm{x} + 5.$

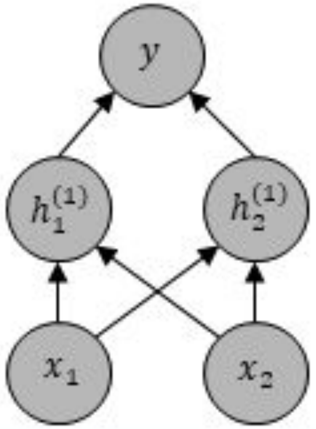


Example: $\mathrm{y} = f(x) = \cos(x^2) + 1.$



Example: $y = f(x_1, x_2) = w_1^{(2)} g(w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2) + w_2^{(2)} g(w_{2,1}^{(1)} x_1 + w_{2,2}^{(1)} x_2)$

$$y = w_1^{(2)} h_1^{(1)} + w_2^{(2)} h_2^{(1)}$$



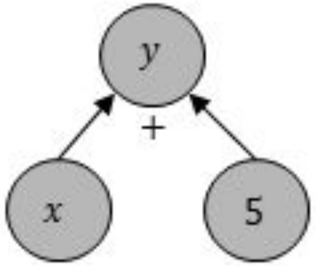
Computational Graphs

Optimization methods often need gradients $\nabla_\theta f(\boldsymbol{X};\theta)$.

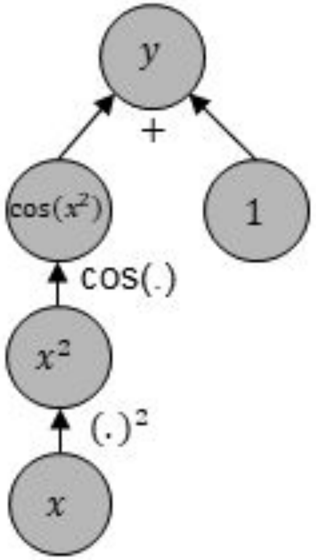
Symbolic representations of $f(\theta)$ useful for automatic differentiation to obtain $\nabla_\theta f$.

Computational graph represents function evaluation in terms of more basic operations.

Example: $\mathrm{y} = f(x) = \mathrm{x} + 5.$

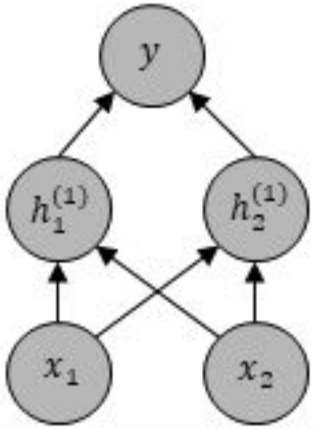


Example: $\mathrm{y} = f(x) = \cos(x^2) + 1.$



Example: $y = f(x_1, x_2) = w_1^{(2)} g(w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2) + w_2^{(2)} g(w_{2,1}^{(1)} x_1 + w_{2,2}^{(1)} x_2)$

$y = w_1^{(2)} h_1^{(1)} + w_2^{(2)} h_2^{(1)}$



Gradient can be computed provided we know how to differentiate result of each operation in terms of contributing terms.

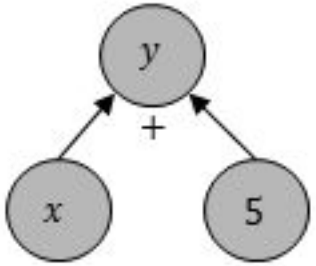
Computational Graphs

Optimization methods often need gradients $\nabla_\theta f(X;\theta)$.

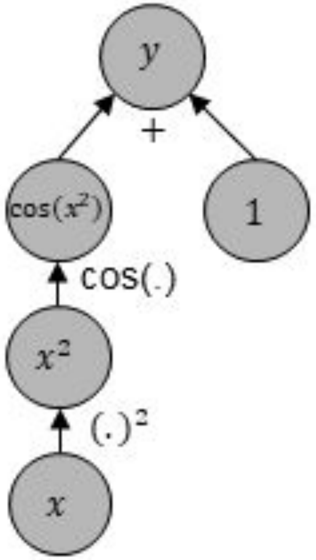
Symbolic representations of $f(\theta)$ useful for automatic differentiation to obtain $\nabla_\theta f$.

Computational graph represents function evaluation in terms of more basic operations.

Example: $y = f(x) = x + 5$.

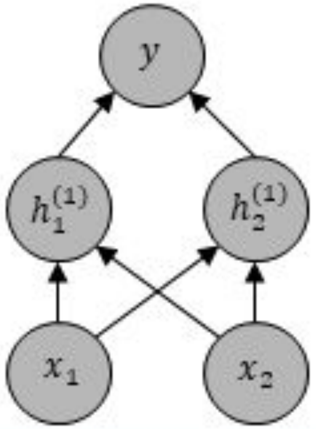


Example: $y = f(x) = \cos(x^2) + 1$.



Example: $y = f(x_1, x_2) = w_1^{(2)} g(w_{1,1}^{(1)} x_1 + w_{1,2}^{(1)} x_2) + w_2^{(2)} g(w_{2,1}^{(1)} x_1 + w_{2,2}^{(1)} x_2)$

$y = w_1^{(2)} h_1^{(1)} + w_2^{(2)} h_2^{(1)}$



Gradient can be computed provided we know how to differentiate result of each operation in terms of contributing terms.

Function derivatives can then be built up using the **chain-rule of calculus.**

Chain-Rule

Chain-Rule of Calculus:

$$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right), \qquad \boldsymbol{u}^{(n)} = f\left(f\left(\dots f\left(\boldsymbol{u}^{(1)}\right)\dots\right)\right)$$

$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Chain-Rule

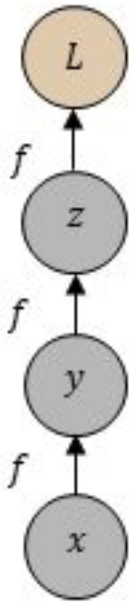
Chain-Rule of Calculus:

$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right), \qquad \boldsymbol{u}^{(n)} = f\left(f\left(\ldots f\left(\boldsymbol{u}^{(1)}\right)\ldots\right)\right)$

$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Example: $L = f(z),\ z = f(y), y = f(x)$ we then have $L = f\left(f\left(f(x)\right)\right)$

Chain-Rule



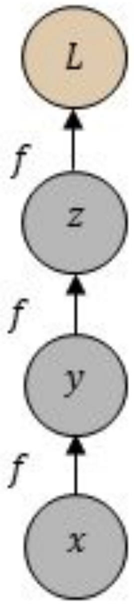
Chain-Rule of Calculus:

$$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right), \qquad \boldsymbol{u}^{(n)} = f\left(f\left(\dots f\left(\boldsymbol{u}^{(1)}\right)\dots\right)\right)$$

$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Example: $L = f(z),\ z = f(y), y = f(x)$ we then have $L = f\left(f\left(f(x)\right)\right)$

Chain-Rule



Chain-Rule of Calculus:

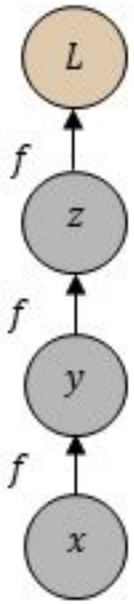
$$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right), \qquad \boldsymbol{u}^{(n)} = f\left(f\left(\ldots f\left(\boldsymbol{u}^{(1)}\right)\ldots\right)\right)$$

$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}}\frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Example: $L = f(z),\ z = f(y), y = f(x)$ we then have $L = f\left(f\left(f(x)\right)\right)$

$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$, can also be expressed as (i) $\frac{\partial L}{\partial x} = \mathrm{f}'(\mathrm{f}(\mathrm{f}(\mathrm{x}))\mathrm{f}'(\mathrm{f}(\mathrm{x}))\mathrm{f}'(\mathrm{x})$ or (ii) $\frac{\partial L}{\partial x} = \mathrm{f}'(\mathrm{z})\mathrm{f}'(\mathrm{y})\mathrm{f}'(\mathrm{x})$.

Chain-Rule



Chain-Rule of Calculus:

$$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right), \qquad \boldsymbol{u}^{(n)} = f\left(f\left(\ldots f\left(\boldsymbol{u}^{(1)}\right)\ldots\right)\right)$$

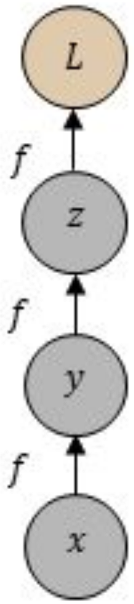
$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Example: $L = f(z),\ z = f(y), y = f(x)$ we then have $L = f\left(f\left(f(x)\right)\right)$

$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$, can also be expressed as (i) $\frac{\partial L}{\partial x} = \mathrm{f}'(\mathrm{f}(\mathrm{f}(\mathrm{x}))\mathrm{f}'(\mathrm{f}(\mathrm{x}))\mathrm{f}'(\mathrm{x})$ or (ii) $\frac{\partial L}{\partial x} = \mathrm{f}'(\mathrm{z})\mathrm{f}'(\mathrm{y})\mathrm{f}'(\mathrm{x})$.

First expression (i) “naive” chain-rule can require many function evaluations (exponential number some cases).

Chain-Rule



Chain-Rule of Calculus:

$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right), \qquad \boldsymbol{u}^{(n)} = f\left(f\left(\ldots f\left(\boldsymbol{u}^{(1)}\right)\ldots\right)\right)$

$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

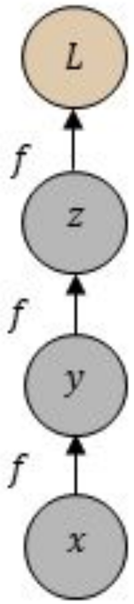
Example: $L = f(z),\ z = f(y), y = f(x)$ we then have $L = f\left(f\left(f(x)\right)\right)$

$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$, can also be expressed as (i) $\frac{\partial L}{\partial x} = \mathrm{f}'(\mathrm{f}(\mathrm{f}(\mathrm{x}))\mathrm{f}'(\mathrm{f}(\mathrm{x}))\mathrm{f}'(\mathrm{x})$ or (ii) $\frac{\partial L}{\partial x} = \mathrm{f}'(\mathrm{z})\mathrm{f}'(\mathrm{y})\mathrm{f}'(\mathrm{x})$.

First expression (i) "naive" chain-rule can require many function evaluations (exponential number some cases).

Second expression (ii) composite chain-rule reuses previous functional evaluations (cost in memory).

Chain-Rule



Chain-Rule of Calculus:

$$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right), \qquad \boldsymbol{u}^{(n)} = f\left(f\left(\dots f\left(\boldsymbol{u}^{(1)}\right)\dots\right)\right)$$

$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Example: $L = f(z),\ z = f(y), y = f(x)$ we then have $L = f\left(f\left(f(x)\right)\right)$

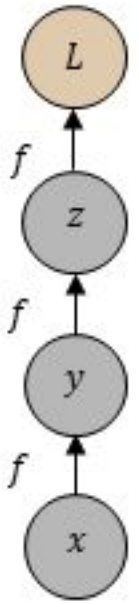
$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$, can also be expressed as (i) $\frac{\partial L}{\partial x} = \mathrm{f}'(\mathrm{f}(\mathrm{f}(\mathrm{x}))\mathrm{f}'(\mathrm{f}(\mathrm{x}))\mathrm{f}'(\mathrm{x})$ or (ii) $\frac{\partial L}{\partial x} = \mathrm{f}'(\mathrm{z})\mathrm{f}'(\mathrm{y})\mathrm{f}'(\mathrm{x})$.

First expression (i) "naive" chain-rule can require many function evaluations (exponential number some cases).

Second expression (ii) composite chain-rule reuses previous functional evaluations (cost in memory).

Computational can either assemble product to evaluate or store symbolic representation.

Chain-Rule



Chain-Rule of Calculus:

$$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right), \qquad \boldsymbol{u}^{(n)} = f\left(f\left(\dots f\left(\boldsymbol{u}^{(1)}\right)\dots\right)\right)$$

$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Example: $L = f(z),\ z = f(y), y = f(x)$ we then have $L = f\left(f\left(f(x)\right)\right)$

$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z}\frac{\partial z}{\partial y}\frac{\partial y}{\partial x}$, can also be expressed as (i) $\frac{\partial L}{\partial x} = \mathrm{f}'(\mathrm{f}(\mathrm{f}(\mathrm{x}))\mathrm{f}'(\mathrm{f}(\mathrm{x}))\mathrm{f}'(\mathrm{x})$ or (ii) $\frac{\partial L}{\partial x} = \mathrm{f}'(\mathrm{z})\mathrm{f}'(\mathrm{y})\mathrm{f}'(\mathrm{x})$.

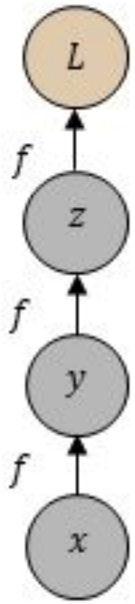
First expression (i) "naive" chain-rule can require many function evaluations (exponential number some cases).

Second expression (ii) composite chain-rule reuses previous functional evaluations (cost in memory).

Computational can either assemble product to evaluate or store symbolic representation.

Advantages of (i) when memory storage issues, otherwise (ii) is usually preferred.

Back-Propagation Method



Back-Propagation Method:

$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right)$, $\boldsymbol{u}^{(n)} = f\left(f\left(\ldots f\left(\boldsymbol{u}^{(1)}\right)\ldots\right)\right)$

$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \sum_{i:\, j \in \mathrm{Pa}\left(u^{(i)}\right)} \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Back-Propagation Method

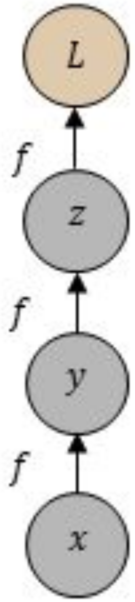
Back-Propagation Method:

$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right)$, $\boldsymbol{u}^{(n)} = f\left(f\left(\dots f\left(\boldsymbol{u}^{(1)}\right)\dots\right)\right)$

$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \sum_{i:\, j \in \mathrm{Pa}(u^{(i)})} \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Algorithm I (Forward-Pass):

```
Input: $x^{(1)}, x^{(2)}, \dots, x^{(n_I)}$
  for $k = 1,2,\dots,n_I$
    $u^{(k)} \leftarrow x^{(k)}$
  for $m = n_I + 1, \dots, n$
    $\mathbb{U}^{(m)} \leftarrow \{u^{(j)} \mid j \in \mathrm{Pa}(u^{(m)})\}$
    $u^{(m)} \leftarrow f^{(m)}(\mathbb{U}^{(m)})$
Output: $u^{(n)}, \{u^{(i)}\}_{i=1}^{n}$.
```



Back-Propagation Method

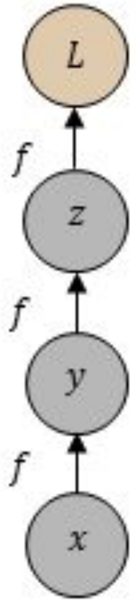
Back-Propagation Method:

$$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right),\ \boldsymbol{u}^{(n)} = f\left(f\left(\ldots f\left(\boldsymbol{u}^{(1)}\right)\ldots\right)\right)$$

$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \sum_{i:\, j\, \in \mathrm{Pa}\left(u^{(i)}\right)} \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Algorithm I (Forward-Pass):

```
Input: $x^{(1)}, x^{(2)}, \ldots, x^{(n_I)}$
  for $k = 1,2,\ldots,n_I$
    $u^{(k)} \leftarrow x^{(k)}$
  for $m = n_I + 1,\ldots,n$
    $\mathbb{U}^{(m)} \leftarrow \{u^{(j)} \mid j \in \mathrm{Pa}(u^{(m)})\}$
    $u^{(m)} \leftarrow f^{(m)}(\mathbb{U}^{(m)})$
Output: $u^{(n)}, \{u^{(i)}\}_{i=1}^{n}$.
```



Algorithm I computes the functional evaluations $\{u^{(i)}\}_{i=1}^{n}$.

Back-Propagation Method

Back-Propagation Method:

$$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right),\ \boldsymbol{u}^{(n)} = f\left(f\left(\dots f\left(\boldsymbol{u}^{(1)}\right)\dots\right)\right)$$

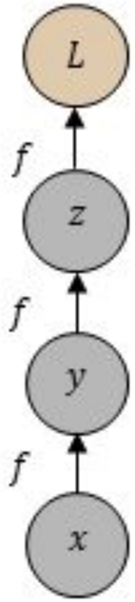
$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \sum_{i:\, j\, \in \mathrm{Pa}\left(u^{(i)}\right)} \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Algorithm I (Forward-Pass):

```
Input: $x^{(1)}, x^{(2)}, \dots, x^{(n_I)}$
  for $k = 1,2,\dots,n_I$
    $u^{(k)} \leftarrow x^{(k)}$
  for $m = n_I + 1,\dots,n$
    $\mathbb{U}^{(m)} \leftarrow \{u^{(j)} \mid j \in \mathrm{Pa}(u^{(m)})\}$
    $u^{(m)} \leftarrow f^{(m)}(\mathbb{U}^{(m)})$
Output: $u^{(n)}, \{u^{(i)}\}_{i=1}^{n}$.
```

Algorithm II (Backward-Pass):

```
Input: $u^{(m)}, m = 1,\dots n$.
grad_table[$u^{(n)}$] ← 1
For $j = n-1,\dots 1$
    grad_table[$u^{(j)}$] ← $\sum_{i:\, j\, \in \mathrm{Pa}(u^{(i)})}$ grad_table[$u^{(i)}$] $\frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$
Output: $\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(k)}},\ k = 1,\dots n_I$.
```



Algorithm I computes the functional evaluations $\{u^{(i)}\}_{i=1}^{n}$.

Back-Propagation Method

Back-Propagation Method:

$$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right),\ \boldsymbol{u}^{(n)} = f\left(f\left(\dots f\left(\boldsymbol{u}^{(1)}\right)\dots\right)\right)$$

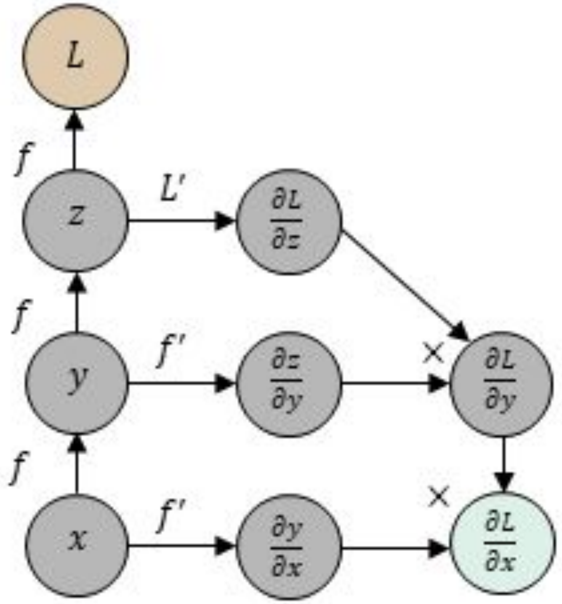
$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \sum_{i:\, j \in \mathrm{Pa}(u^{(i)})} \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Algorithm I (Forward-Pass):

```
Input: $x^{(1)}, x^{(2)}, \dots, x^{(n_I)}$
  for $k = 1,2,\dots,n_I$
    $u^{(k)} \leftarrow x^{(k)}$
  for $m = n_I + 1,\dots,n$
    $\mathbb{U}^{(m)} \leftarrow \{u^{(j)} \mid j \in \mathrm{Pa}(u^{(m)})\}$
    $u^{(m)} \leftarrow f^{(m)}(\mathbb{U}^{(m)})$
Output: $u^{(n)}, \{u^{(i)}\}_{i=1}^{n}$.
```

Algorithm II (Backward-Pass):

```
Input: $u^{(m)}, m = 1,\dots n$.
grad_table$[u^{(n)}] \leftarrow 1$
For $j = n-1,\dots 1$
    grad_table$[u^{(j)}] \leftarrow \sum_{i:\, j \in \mathrm{Pa}(u^{(i)})}$ grad_table$[u^{(i)}] \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$
Output: $\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(k)}},\ k = 1,\dots n_I$.
```



Algorithm I computes the functional evaluations $\{u^{(i)}\}_{i=1}^{n}$.

Back-Propagation Method

Back-Propagation Method:

$$\boldsymbol{u}^{(k)} = f\left(\boldsymbol{u}^{(k-1)}\right),\ \boldsymbol{u}^{(n)} = f\left(f\left(\dots f\left(\boldsymbol{u}^{(1)}\right)\dots\right)\right)$$

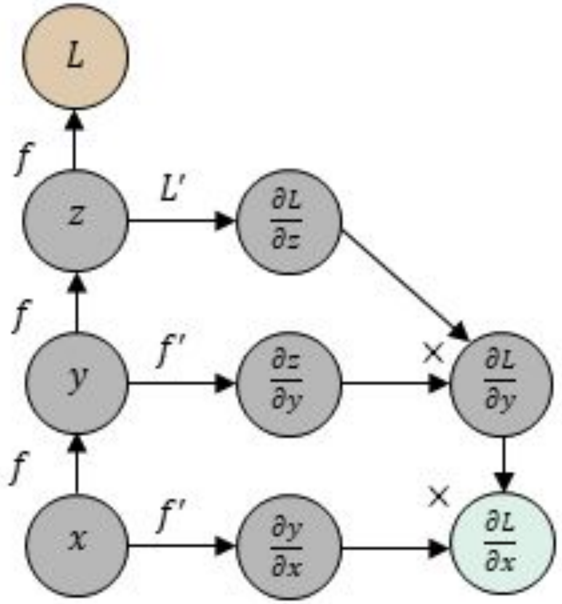
$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \sum_{i:\, j \in \mathrm{Pa}(u^{(i)})} \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Algorithm I (Forward-Pass):

```
Input: $x^{(1)}, x^{(2)}, \dots, x^{(n_I)}$
  for $k = 1,2,\dots,n_I$
    $u^{(k)} \leftarrow x^{(k)}$
  for $m = n_I + 1,\dots,n$
    $\mathbb{U}^{(m)} \leftarrow \{u^{(j)} \mid j \in \mathrm{Pa}(u^{(m)})\}$
    $u^{(m)} \leftarrow f^{(m)}(\mathbb{U}^{(m)})$
Output: $u^{(n)}, \{u^{(i)}\}_{i=1}^{n}$.
```

Algorithm II (Backward-Pass):

```
Input: $u^{(m)}, m = 1,\dots n$.
grad_table$[u^{(n)}] \leftarrow 1$
For $j = n-1,\dots 1$
    grad_table$[u^{(j)}] \leftarrow \sum_{i:\, j \in \mathrm{Pa}(u^{(i)})}$grad_table$[u^{(i)}]\frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$
Output: $\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(k)}},\ k = 1,\dots n_I$.
```



Algorithm I computes the functional evaluations $\{u^{(i)}\}_{i=1}^{n}$.

Algorithm II maintains at each stage: grad_table$[u^{(j)}] = \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}}$.

Back-Propagation Method

Back-Propagation Method:

$$\boldsymbol{u}^{(k)} = f(\boldsymbol{u}^{(k-1)}),\ \boldsymbol{u}^{(n)} = f\left(f(\dots f(\boldsymbol{u}^{(1)})\dots)\right)$$

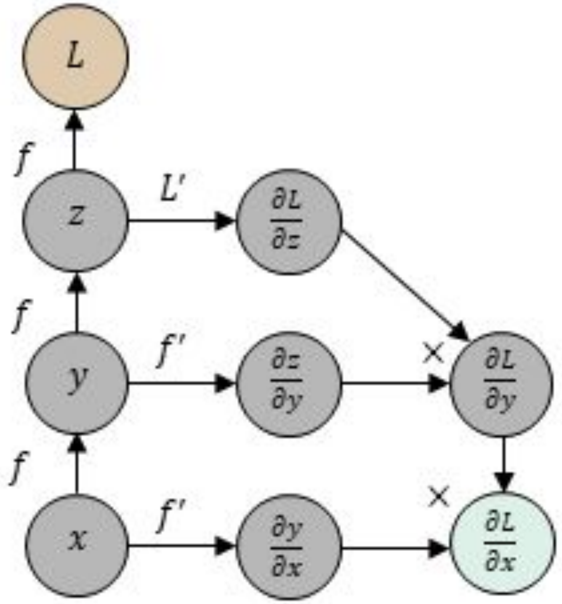
$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \sum_{i:\, j\, \in \mathrm{Pa}(u^{(i)})} \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Algorithm I (Forward-Pass):

```
Input: $x^{(1)}, x^{(2)}, \dots, x^{(n_I)}$
  for $k = 1,2,\dots,n_I$
    $u^{(k)} \leftarrow x^{(k)}$
  for $m = n_I + 1, \dots, n$
    $\mathbb{U}^{(m)} \leftarrow \{u^{(j)} \mid j \in \mathrm{Pa}(u^{(m)})\}$
    $u^{(m)} \leftarrow f^{(m)}(\mathbb{U}^{(m)})$
Output: $u^{(n)}, \{u^{(i)}\}_{i=1}^{n}$.
```

Algorithm II (Backward-Pass):

```
Input: $u^{(m)}, m = 1, \dots n$.
grad_table$[u^{(n)}] \leftarrow 1$
For $j = n-1, \dots 1$
    grad_table$[u^{(j)}] \leftarrow \sum_{i:\, j\, \in \mathrm{Pa}(u^{(i)})}$grad_table$[u^{(i)}]\frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$
Output: $\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(k)}}$, $k = 1, \dots n_I$.
```



Algorithm I computes the functional evaluations $\{u^{(i)}\}_{i=1}^{n}$.

Algorithm II maintains at each stage: grad_table$[u^{(j)}] = \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}}$.

Back-Propagation consists of the two steps (i) forward pass of algorithm I followed by (ii) backward pass of algorithm II.

Back-Propagation Method

Back-Propagation Method:

$$\boldsymbol{u}^{(k)} = f(\boldsymbol{u}^{(k-1)}),\ \boldsymbol{u}^{(n)} = f\left(f(\dots f(\boldsymbol{u}^{(1)})\dots)\right)$$

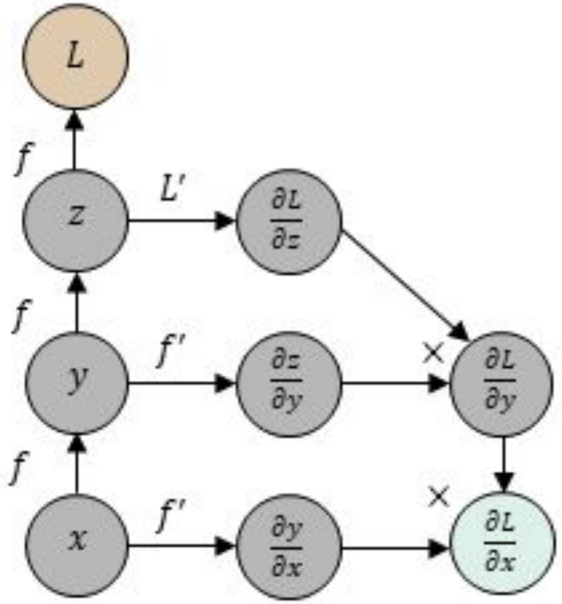
$$\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}} = \sum_{i:\, j \in \mathrm{Pa}(u^{(i)})} \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(i)}} \frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$$

Algorithm I (Forward-Pass):

```
Input: $x^{(1)}, x^{(2)}, \dots, x^{(n_I)}$
  for $k = 1,2,\dots,n_I$
    $u^{(k)} \leftarrow x^{(k)}$
  for $m = n_I + 1,\dots,n$
    $\mathbb{U}^{(m)} \leftarrow \{u^{(j)} \mid j \in \mathrm{Pa}(u^{(m)})\}$
    $u^{(m)} \leftarrow f^{(m)}(\mathbb{U}^{(m)})$
Output: $u^{(n)}, \{u^{(i)}\}_{i=1}^{n}$.
```

Algorithm II (Backward-Pass):

```
Input: $u^{(m)}, m = 1,\dots n$.
grad_table$[u^{(n)}] \leftarrow 1$
For $j = n-1,\dots 1$
    grad_table$[u^{(j)}] \leftarrow \sum_{i:\, j \in \mathrm{Pa}(u^{(i)})}$grad_table$[u^{(i)}]\frac{\partial \boldsymbol{u}^{(i)}}{\partial \boldsymbol{u}^{(j)}}$
Output: $\frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(k)}},\ k = 1,\dots n_I$.
```



Algorithm I computes the functional evaluations $\{u^{(i)}\}_{i=1}^{n}$.

Algorithm II maintains at each stage: grad_table$[u^{(j)}] = \frac{\partial \boldsymbol{u}^{(n)}}{\partial \boldsymbol{u}^{(j)}}$.

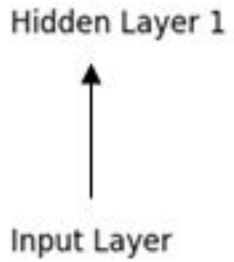
Back-Propagation consists of the two steps (i) forward pass of algorithm I followed by (ii) backward pass of algorithm II.

Parallelized versions and other variants also used for efficiency.

Feed-Forward Neural Networks (FFNNs)
Basic Examples of NN's

Feed-Forward Neural Networks

Neural network architecture with one processing layer feeding forward into the next processing layer.

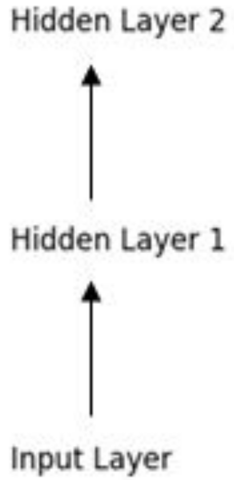


Hidden Layer 1

Input Layer

Feed-Forward Neural Networks

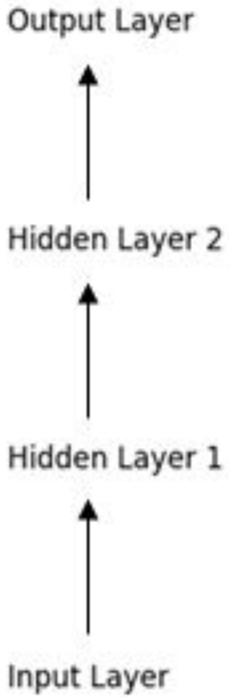
Neural network architecture with one processing layer feeding forward into the next processing layer.



Feed-Forward Neural Networks

Neural network architecture with one processing layer feeding forward into the next processing layer.

Neural Network Architecture



Output Layer

Hidden Layer 2

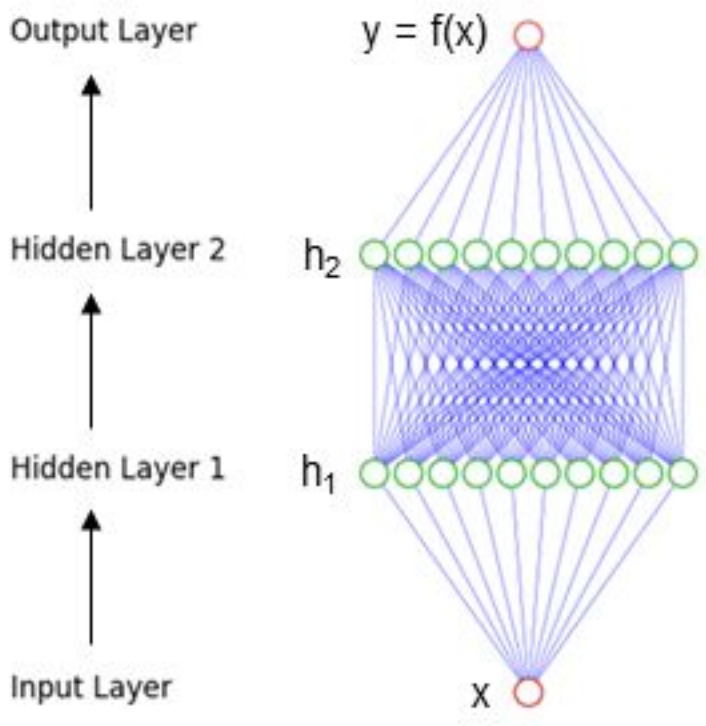
Hidden Layer 1

Input Layer

Feed-Forward Neural Networks

Neural network architecture with one processing layer feeding forward into the next processing layer.

Neural Network Architecture



Feed-Forward Neural Networks

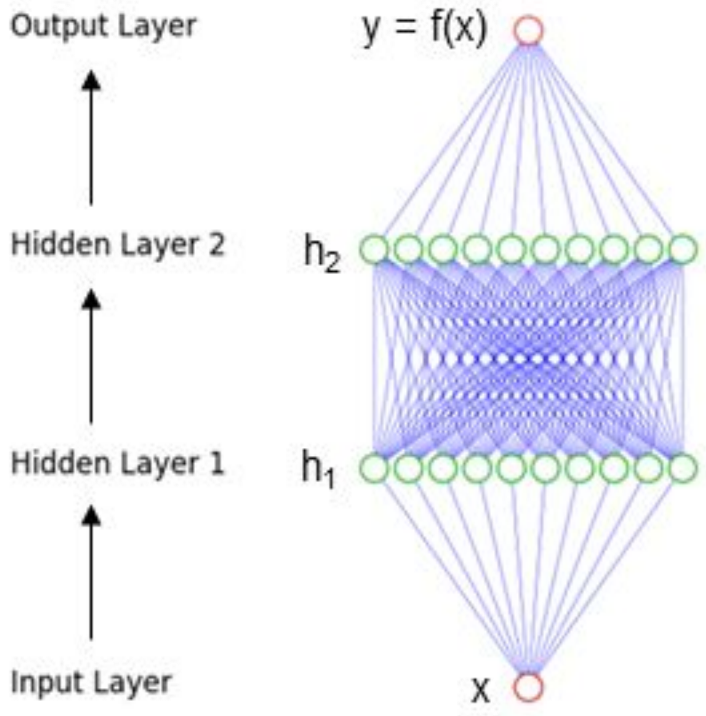
Neural network architecture with one processing layer feeding forward into the next processing layer.

Intermediate hidden processing layers of the form $g(XW + b)$.

Nonlinear transformation by some activation function $g(z)$.

Last processing layer typically is linear $XW + b$.

Neural Network Architecture



Feed-Forward Neural Networks

Neural network architecture with one processing layer feeding forward into the next processing layer.

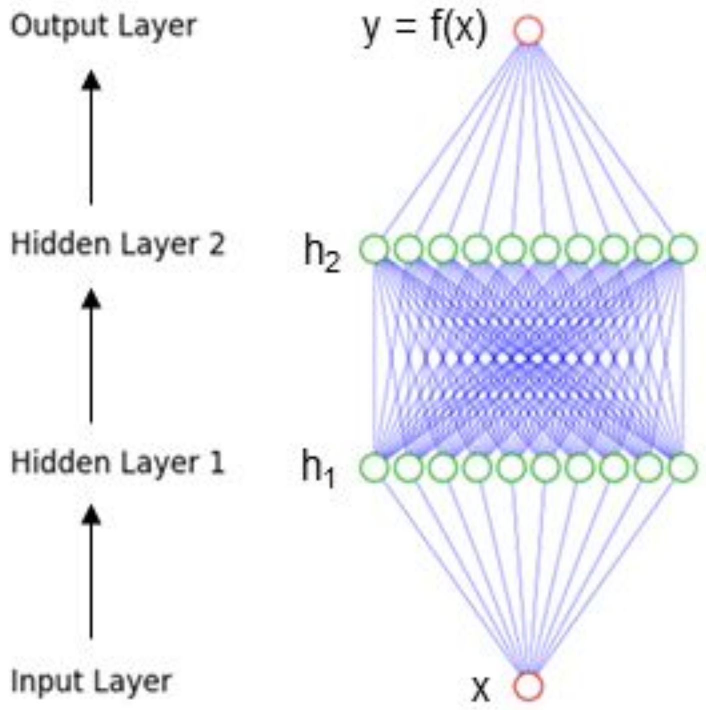
Intermediate hidden processing layers of the form $g(XW + b)$.

Nonlinear transformation by some activation function $g(z)$.

Last processing layer typically is linear $XW + b$.

Feed-Forward Neural Network (FFNN) provide model for $y = f(x)$.

Neural Network Architecture



Feed-Forward Neural Networks

Neural network architecture with one processing layer feeding forward into the next processing layer.

Intermediate hidden processing layers of the form $g(XW + b)$.

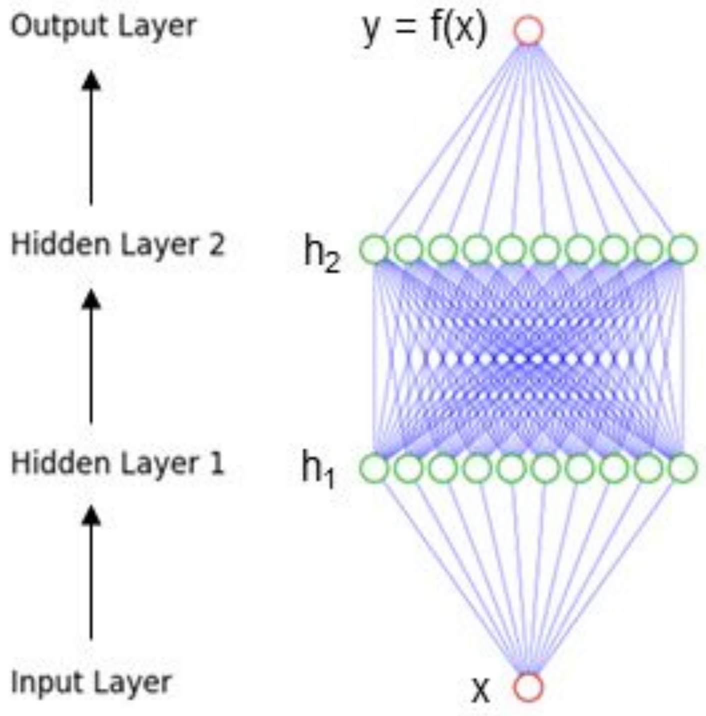
Nonlinear transformation by some activation function $g(z)$.

Last processing layer typically is linear $XW + b$.

Feed-Forward Neural Network (FFNN) provide model for $y = f(x)$.

Learning involves adjusting weights W and bias b of layers.

Neural Network Architecture



Feed-Forward Neural Networks

Neural network architecture with one processing layer feeding forward into the next processing layer.

Intermediate hidden processing layers of the form $g(XW + b)$.

Nonlinear transformation by some activation function $g(z)$.

Last processing layer typically is linear $XW + b$.

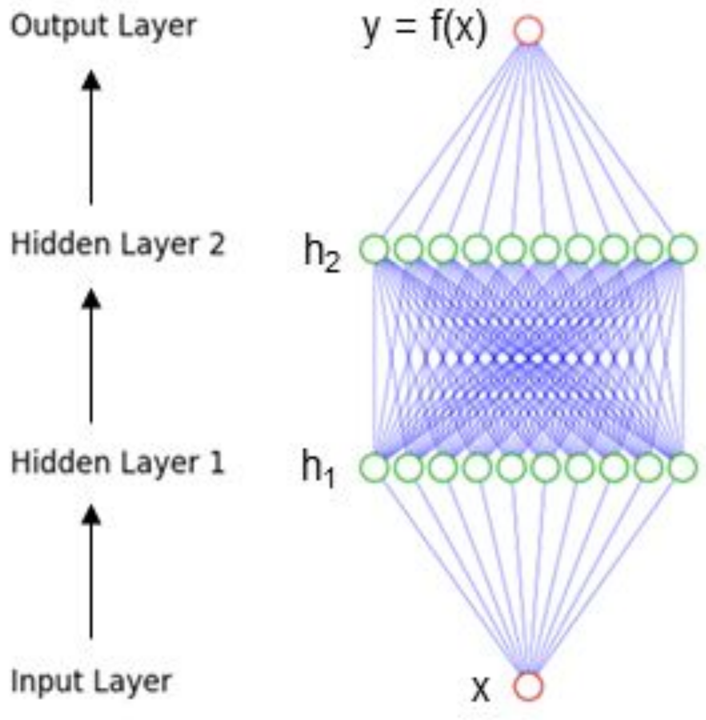
Feed-Forward Neural Network (FFNN) provide model for $y = f(x)$.

Learning involves adjusting weights W and bias b of layers.

Stochastic Gradient Descent (SGD) currently widely used for optimization of weights and bias.

Implicit regularization by choice of batch size and learning rates.

Neural Network Architecture



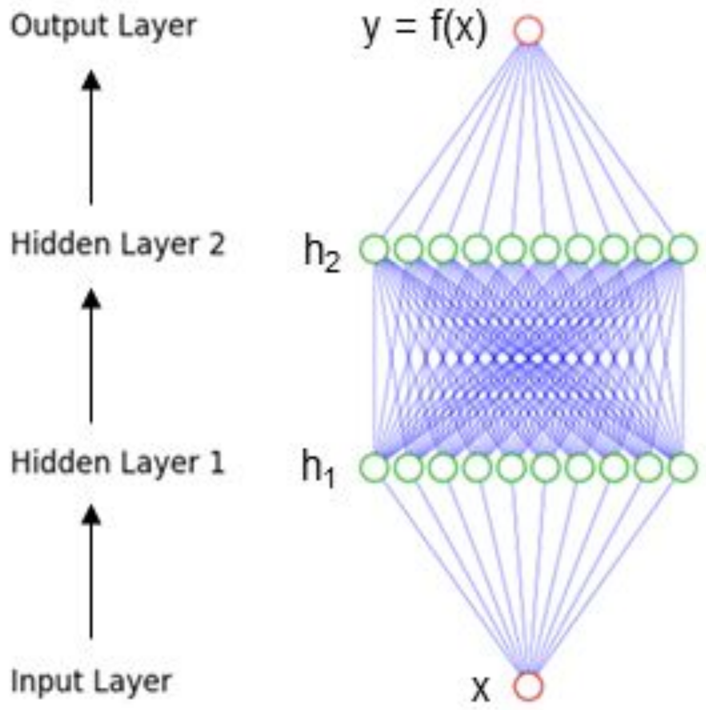
Feed-Forward Neural Networks: Example

Example: Approximate the function $y = \sin(x)$ using FFNN.

Architecture: 2-layers with 10-hidden ReLu nodes per layer.

This NN architecture spans piecewise linear functions (10 nodes).

Neural Network Architecture



Feed-Forward Neural Networks: Example

Example: Approximate the function $y = \sin(x)$ using FFNN.

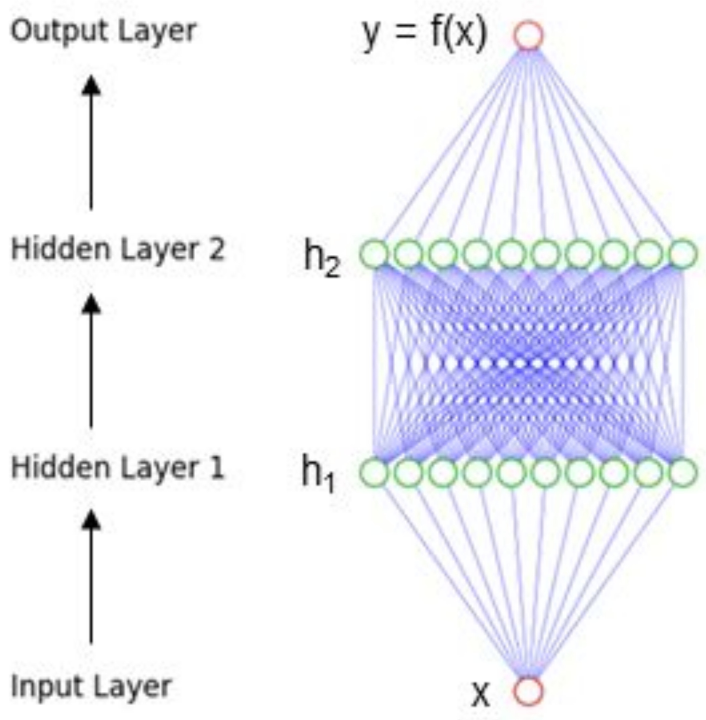
Architecture: 2-layers with 10-hidden ReLu nodes per layer.

This NN architecture spans piecewise linear functions (10 nodes).

Explicitly: $f(X) = g(g(g(X \cdot W^{(1)} + b^{(1)}) \cdot W^{(2)} + b^{(2)}) \cdot W^{(3)} + b^{(3)}$ with $g(z) = \max(0, z)$.

A notion of "loss" required to assess level of success in fit.

Neural Network Architecture



Feed-Forward Neural Networks: Example

Example: Approximate the function $y = \sin(x)$ using FFNN.

Architecture: 2-layers with 10-hidden ReLu nodes per layer.

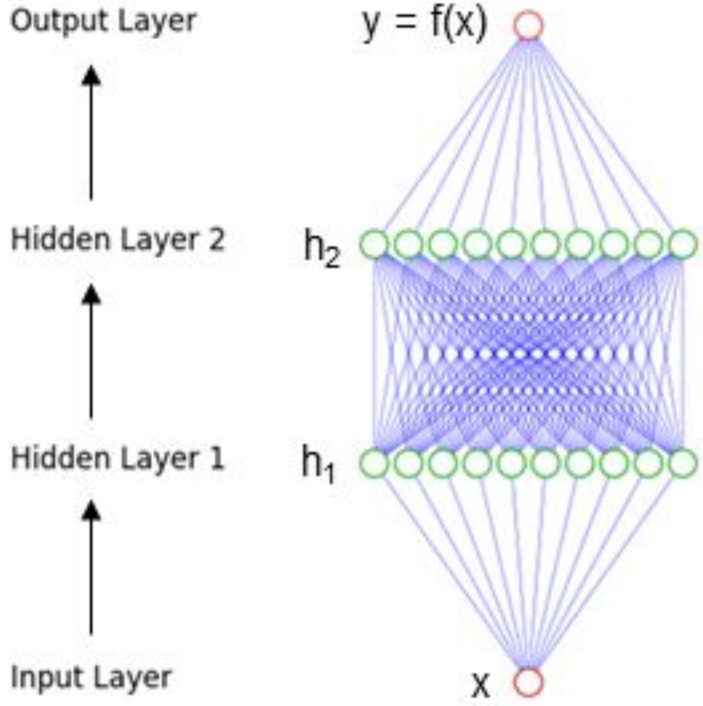
This NN architecture spans piecewise linear functions (10 nodes).

Explicitly: $f(X) = g(g(g(X \cdot W^{(1)} + b^{(1)}) \cdot W^{(2)} + b^{(2)}) \cdot W^{(3)} + b^{(3)}$ with $g(z) = \max(0, z)$.

A notion of "loss" required to assess level of success in fit.

Least-squares loss function $\ell(\{x_i, yi\}) = \sum_i (f(xi) - y_i)^2$.

Neural Network Architecture



Feed-Forward Neural Networks: Example

Example: Approximate the function $y = \sin(x)$ using FFNN.

Architecture: 2-layers with 10-hidden ReLu nodes per layer.

This NN architecture spans piecewise linear functions (10 nodes).

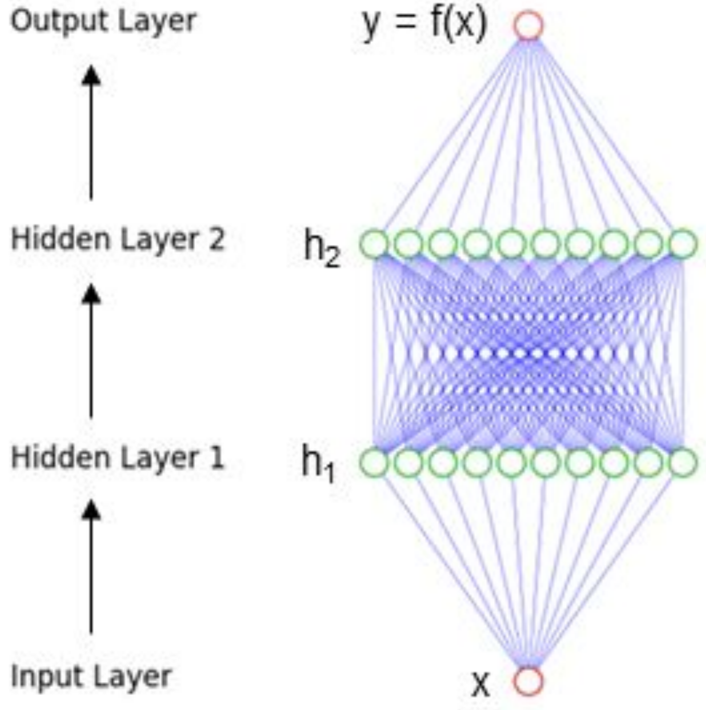
Explicitly: $f(X) = g(g(g(X \cdot W^{(1)} + b^{(1)}) \cdot W^{(2)} + b^{(2)}) \cdot W^{(3)} + b^{(3)}$ with $g(z) = \max(0, z)$.

A notion of "loss" required to assess level of success in fit.

Least-squares loss function $\ell(\{x_i, yi\}) = \sum_i (f(xi) - y_i)^2$.

Learning W,b proceeds by stochastic gradient descent.

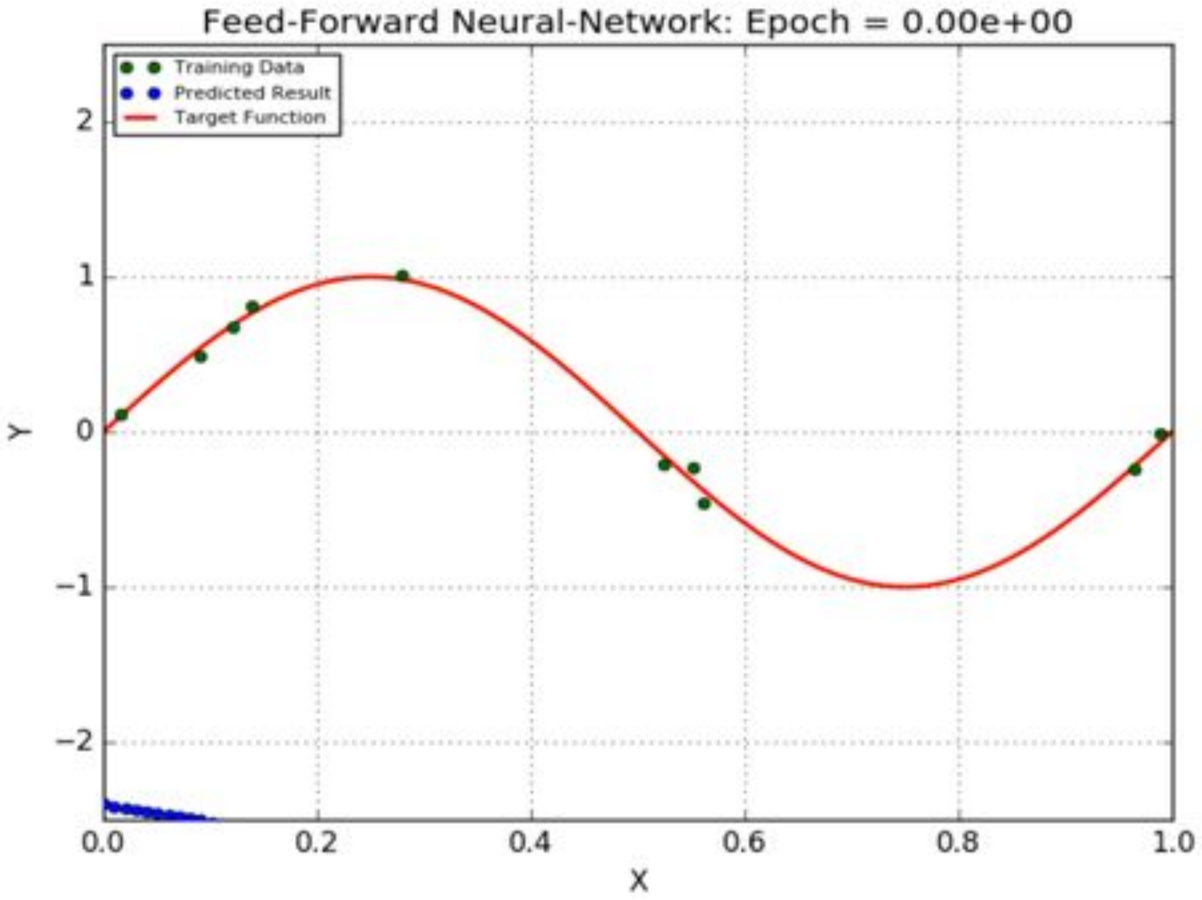
Neural Network Architecture



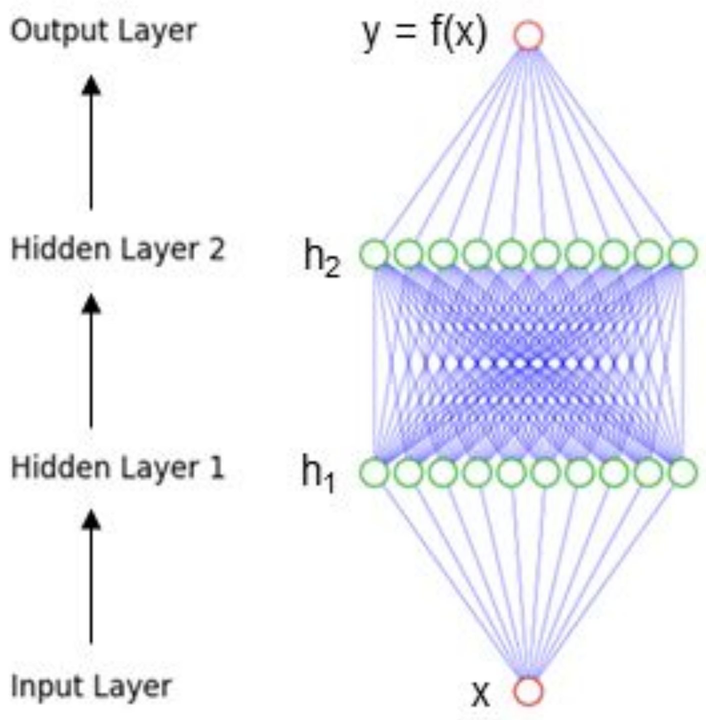
Feed-Forward Neural Networks: Example

Trained FFNN on set of 1000 samples $y_i = \sin(x_i) + \xi_i$.

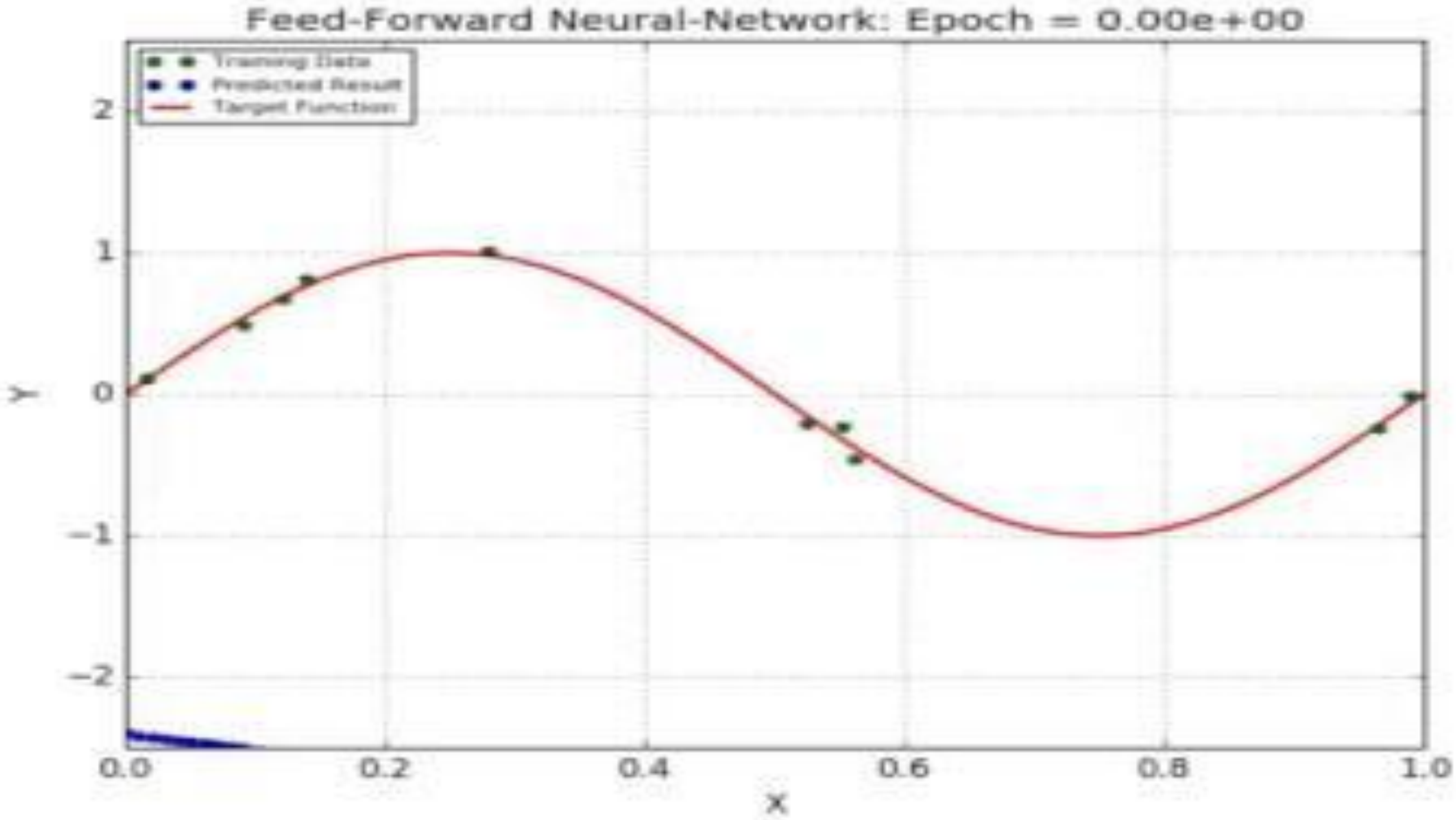
Stochastic Gradient Descent (SGD) used with batch size 10 and learning rate 10^{-4}.



Neural Network Architecture



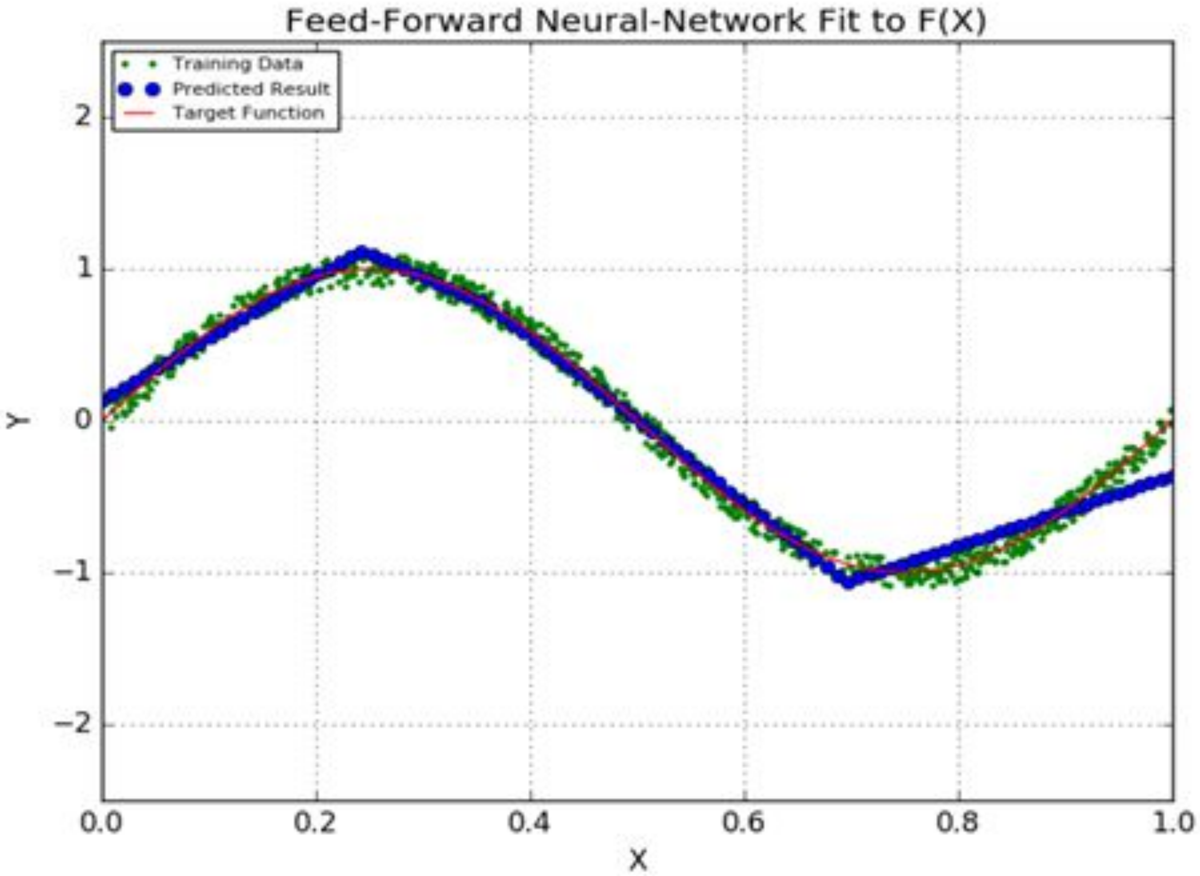
Epoch = 1 step of SGD throughout these examples.


Feed-Forward Neural-Network: Epoch = 0.00e+00
Training Data
Predicted Result
Target Function
Y
2
1
0
−1
−2
X
0.0
0.2
0.4
0.6
0.8
1.0

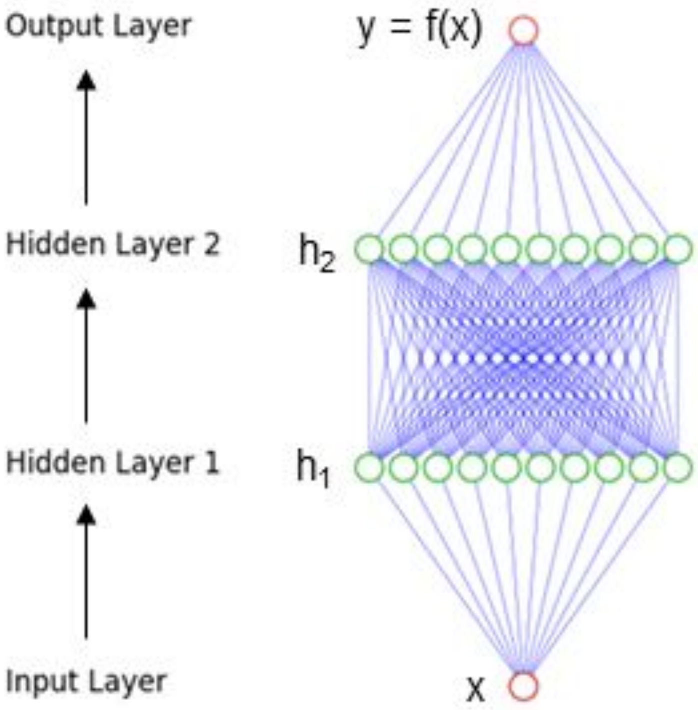
Feed-Forward Neural Networks: Example

Trained FFNN on set of 1000 samples $y_i = \sin(x_i) + \xi_i$.

Stochastic Gradient Descent (SGD) used with batch size 10 and learning rate 10^{-4}.



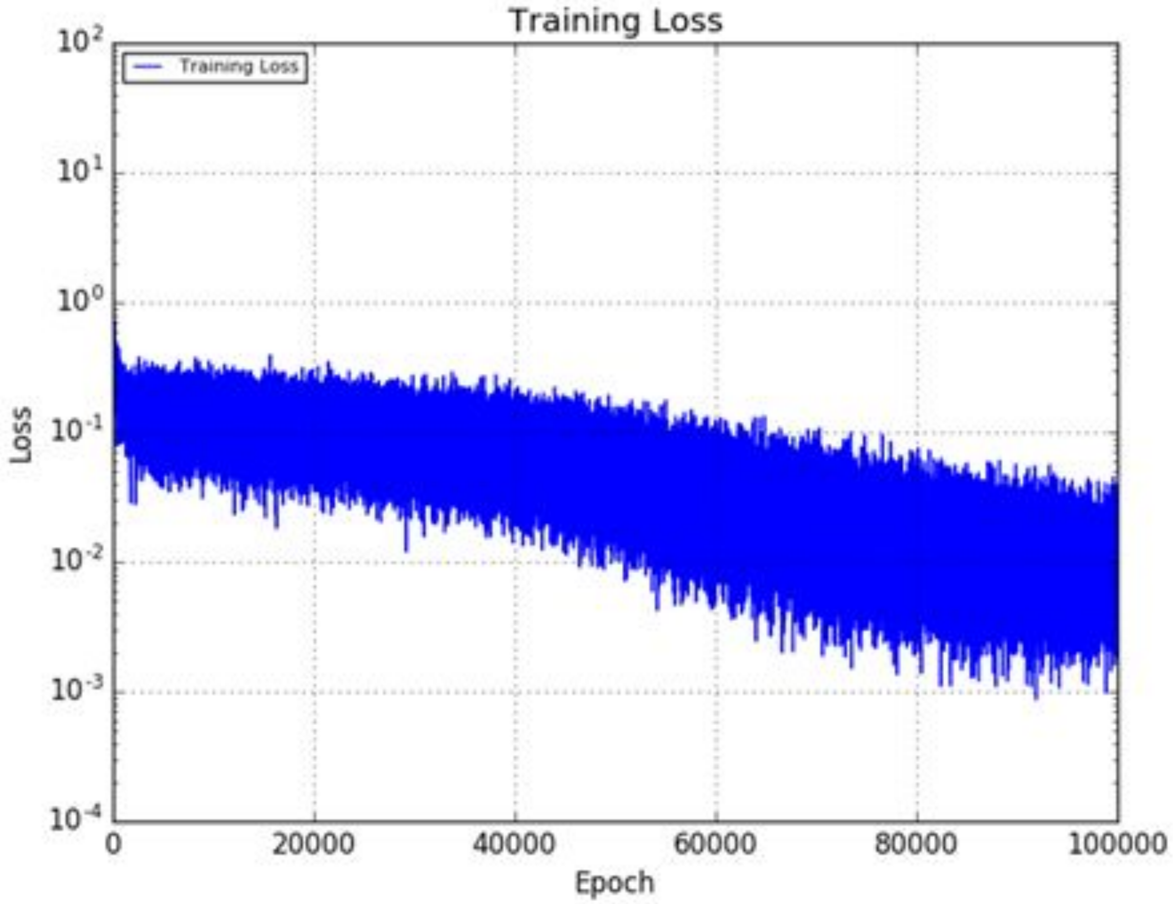
Neural Network Architecture



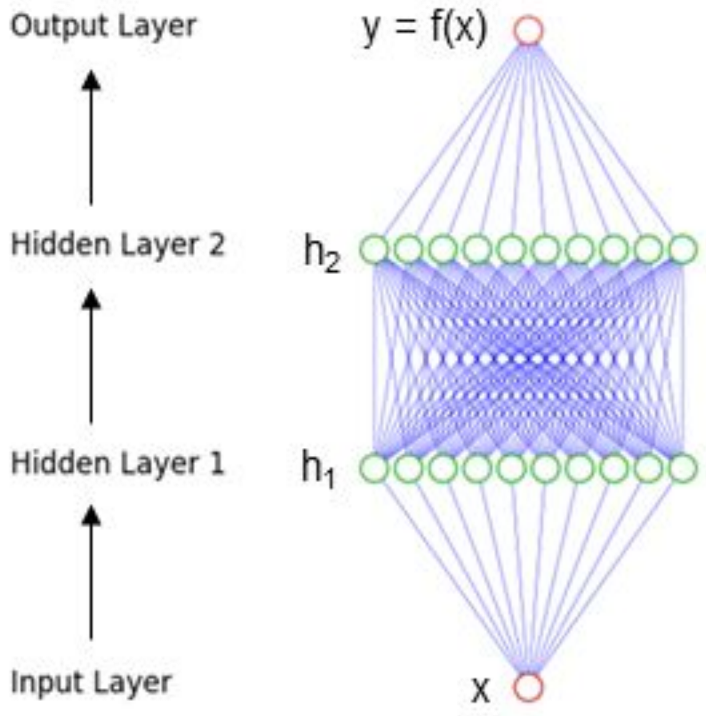
Feed-Forward Neural Networks: Example

Trained FFNN on set of 1000 samples $y_i = \sin(x_i) + \xi_i$.

Stochastic Gradient Descent (SGD) used with batch size 10 and learning rate 10^{-4}.



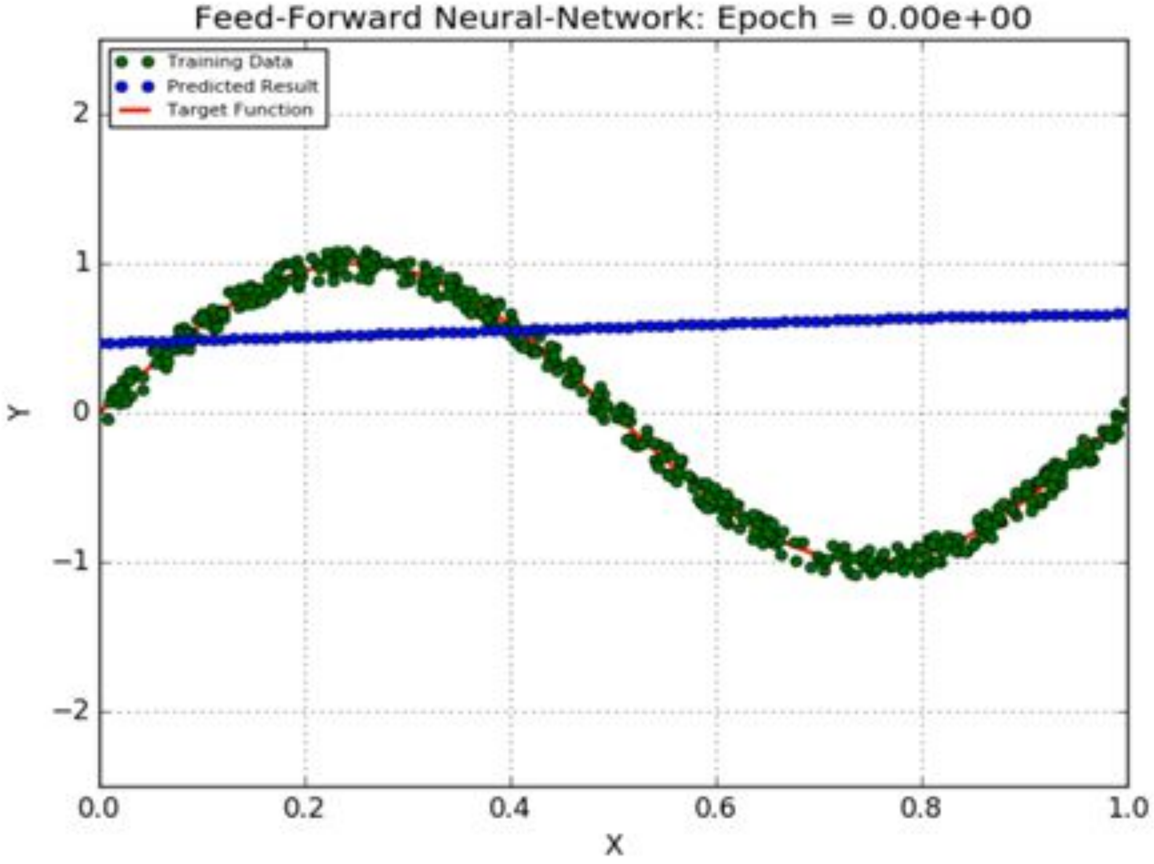
Neural Network Architecture



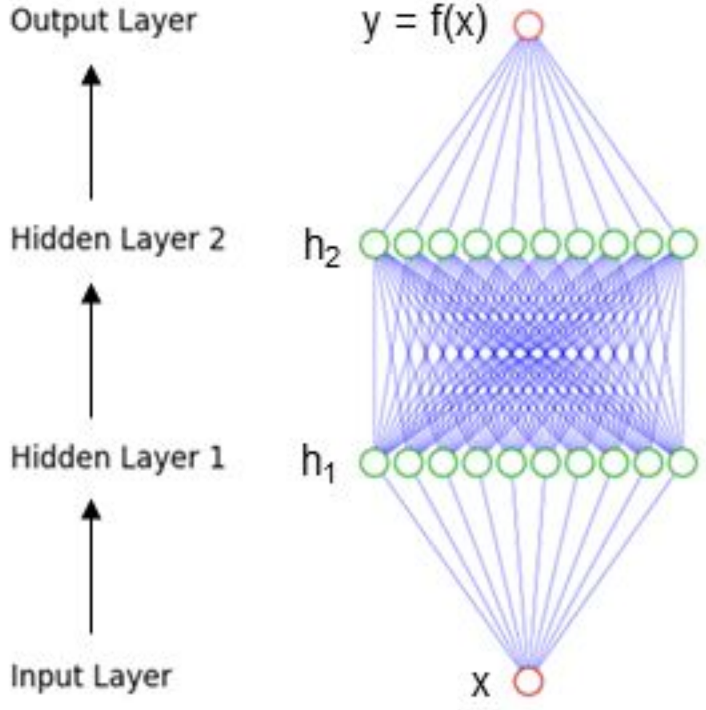
Feed-Forward Neural Networks: Example

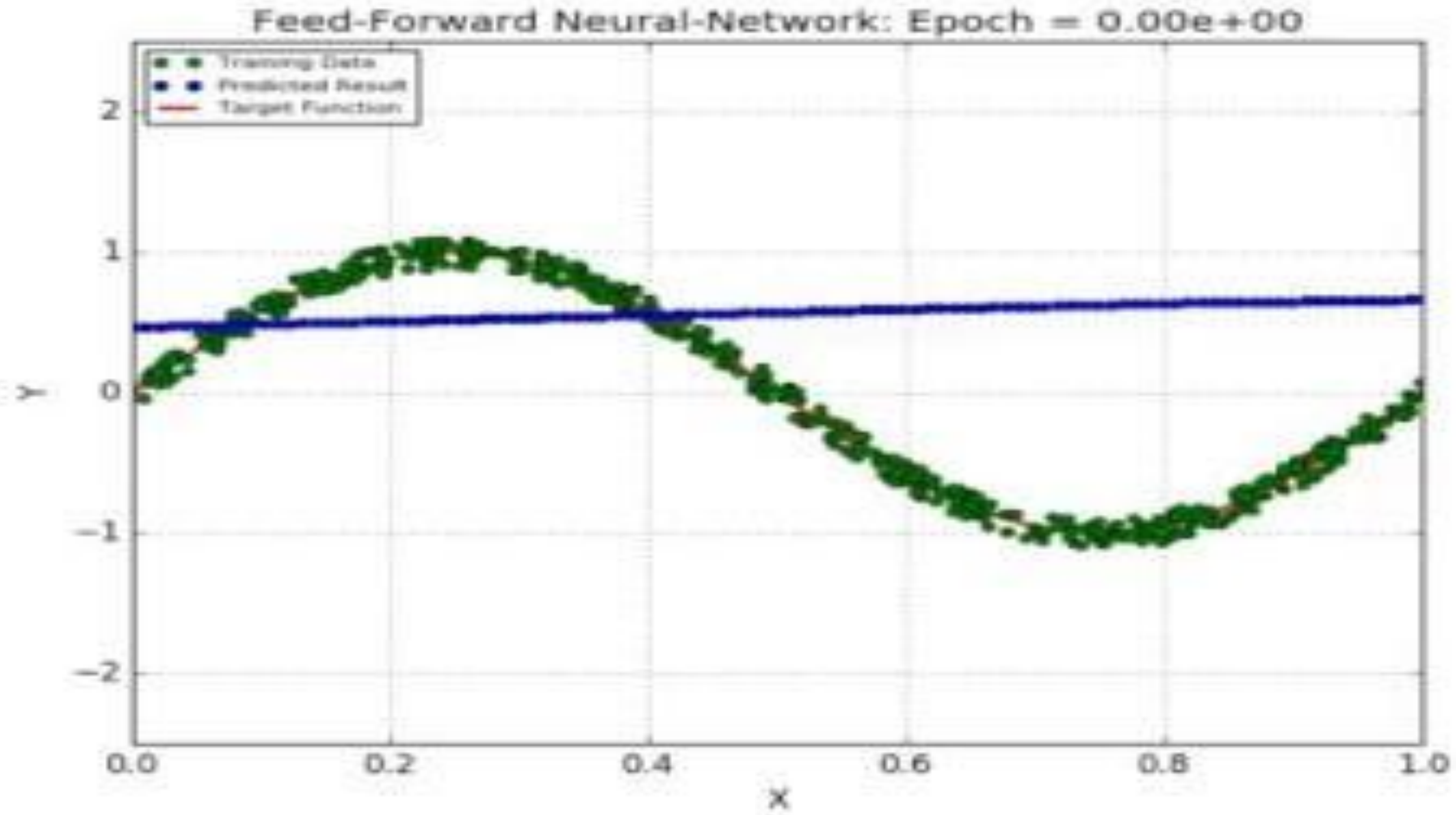
Trained FFNN on set of 1000 samples $y_i = \sin(x_i) + \xi_i$.

Stochastic Gradient Descent (SGD) used with batch size 500 and learning rate 10^{-4}.



Neural Network Architecture

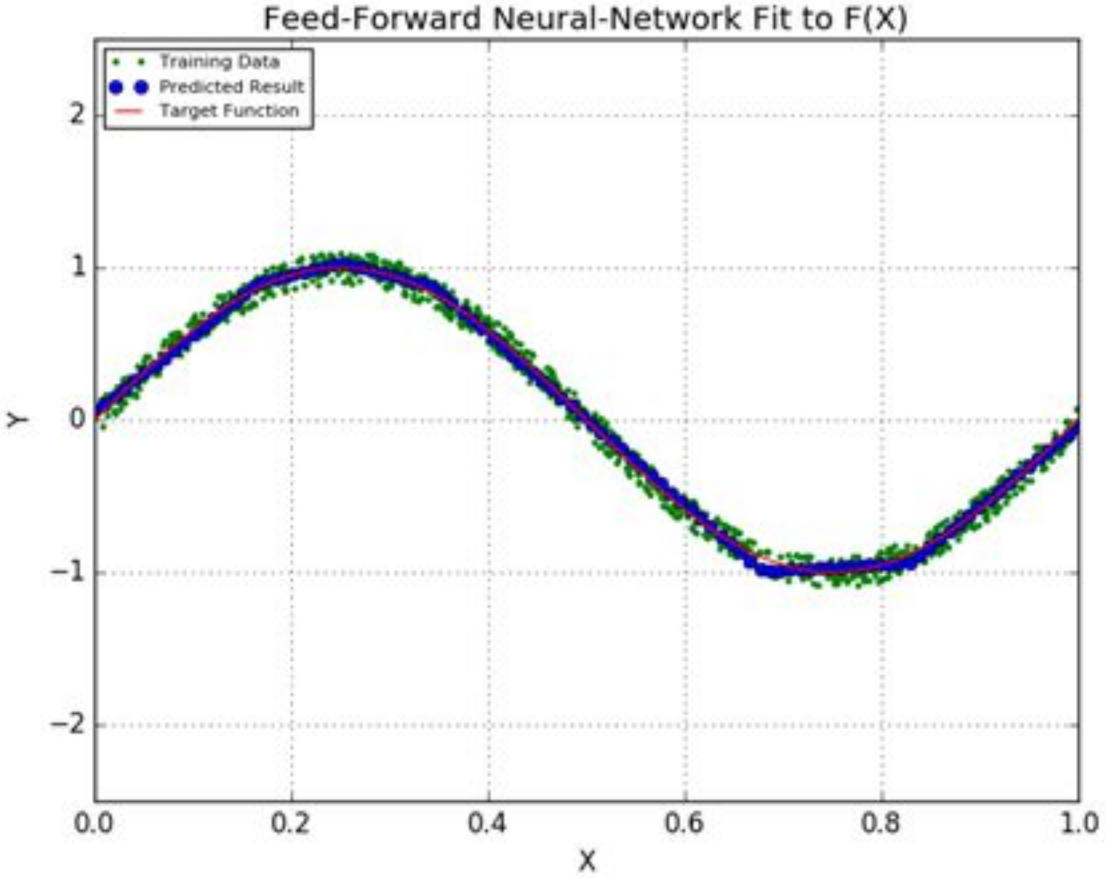



Feed-Forward Neural-Network: Epoch = 0.00e+00
Training Data
Predicted Result
Target Function
Y
2
1
0
−1
−2
X
0.0
0.2
0.4
0.6
0.8
1.0

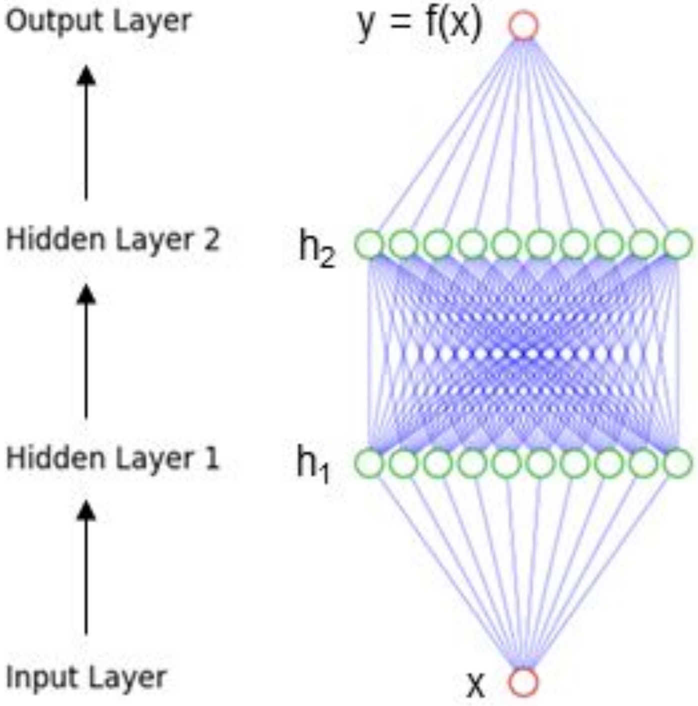
Feed-Forward Neural Networks: Example

Trained FFNN on set of 1000 samples $y_i = \sin(x_i) + \xi_i$.

Stochastic Gradient Descent (SGD) used with batch size 500 and learning rate 10^{-4}.



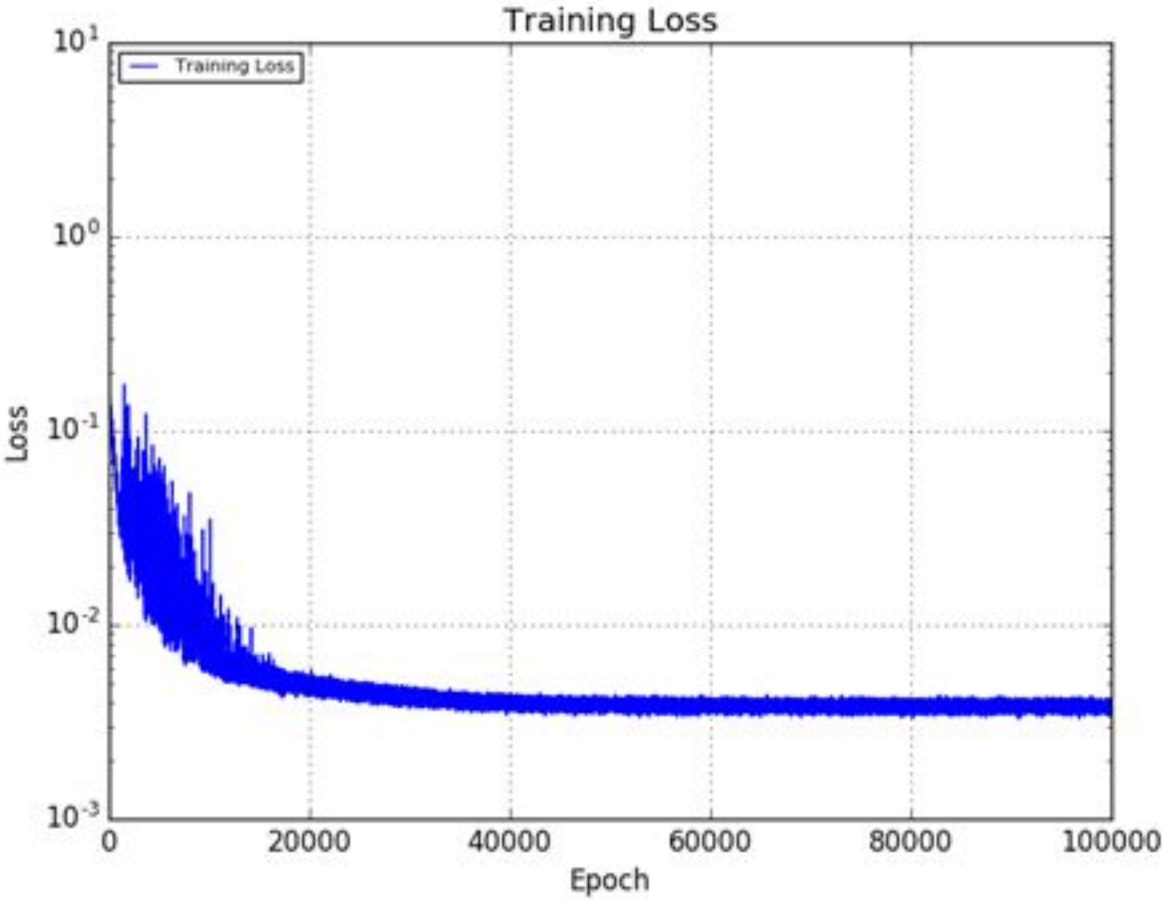
Neural Network Architecture



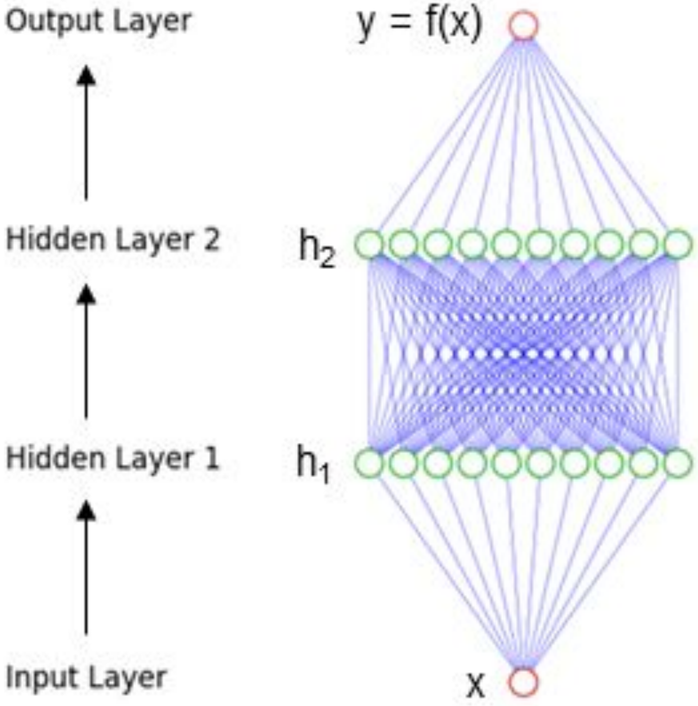
Feed-Forward Neural Networks: Example

Trained FFNN on set of 1000 samples $y_i = \sin(x_i) + \xi_i$.

Stochastic Gradient Descent (SGD) used with batch size 500 and learning rate 10^{-4}.



Neural Network Architecture



Additional fine-tuning of hyper-parameters should be done to enhance efficiency of training.

Feed-Forward Neural Networks: Example

Example: Approximate the function $y = \sin(6\pi x) + 2x^2$ using FFNN.

Architecture: 2-layers with 100-hidden ReLu nodes per layer.

This NN architecture spans piecewise linear functions (100 nodes).

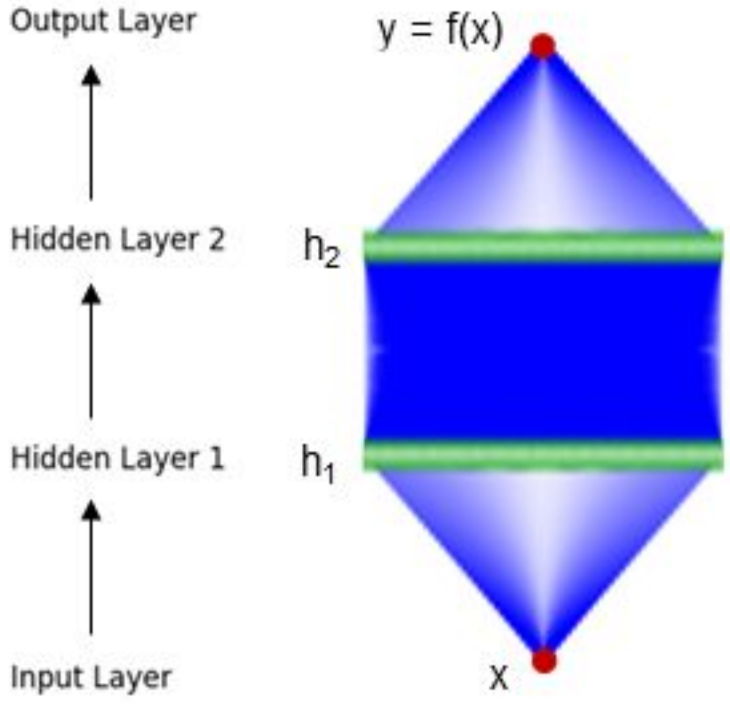
Explicitly: $f(X) = g(g(g(X \cdot W^{(1)} + b^{(1)}) \cdot W^{(2)} + b^{(2)}) \cdot W^{(3)} + b^{(3)}$
with $g(z) = \max(0, z)$.

A notion of "loss" required to assess level of success in fit.

Least-squares loss function $\ell(\{x_i, yi\}) = \sum_i (f(xi) - y_i)^2$.

Learning W,b proceeds by stochastic gradient descent.

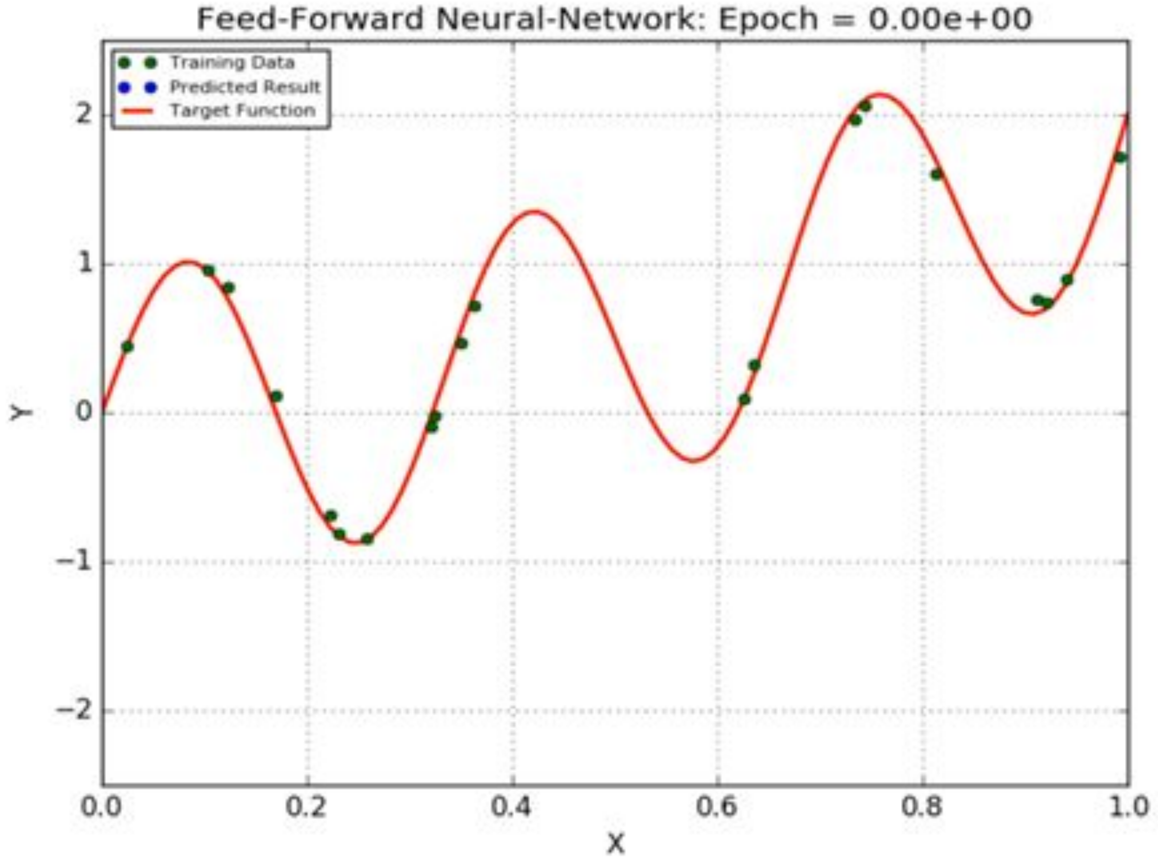
Neural Network Architecture



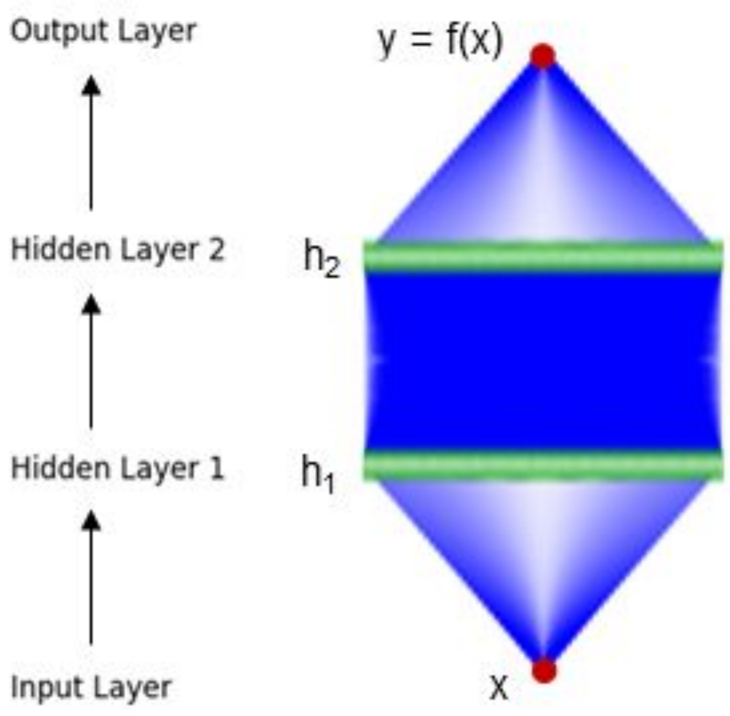
Feed-Forward Neural Networks: Example

Trained FFNN on set of 1000 samples $y_i = \sin(6\pi x_i) + 2x^2 + \xi_i$.

Stochastic Gradient Descent (SGD) used with batch size 20 and learning rate 10^{-4}.



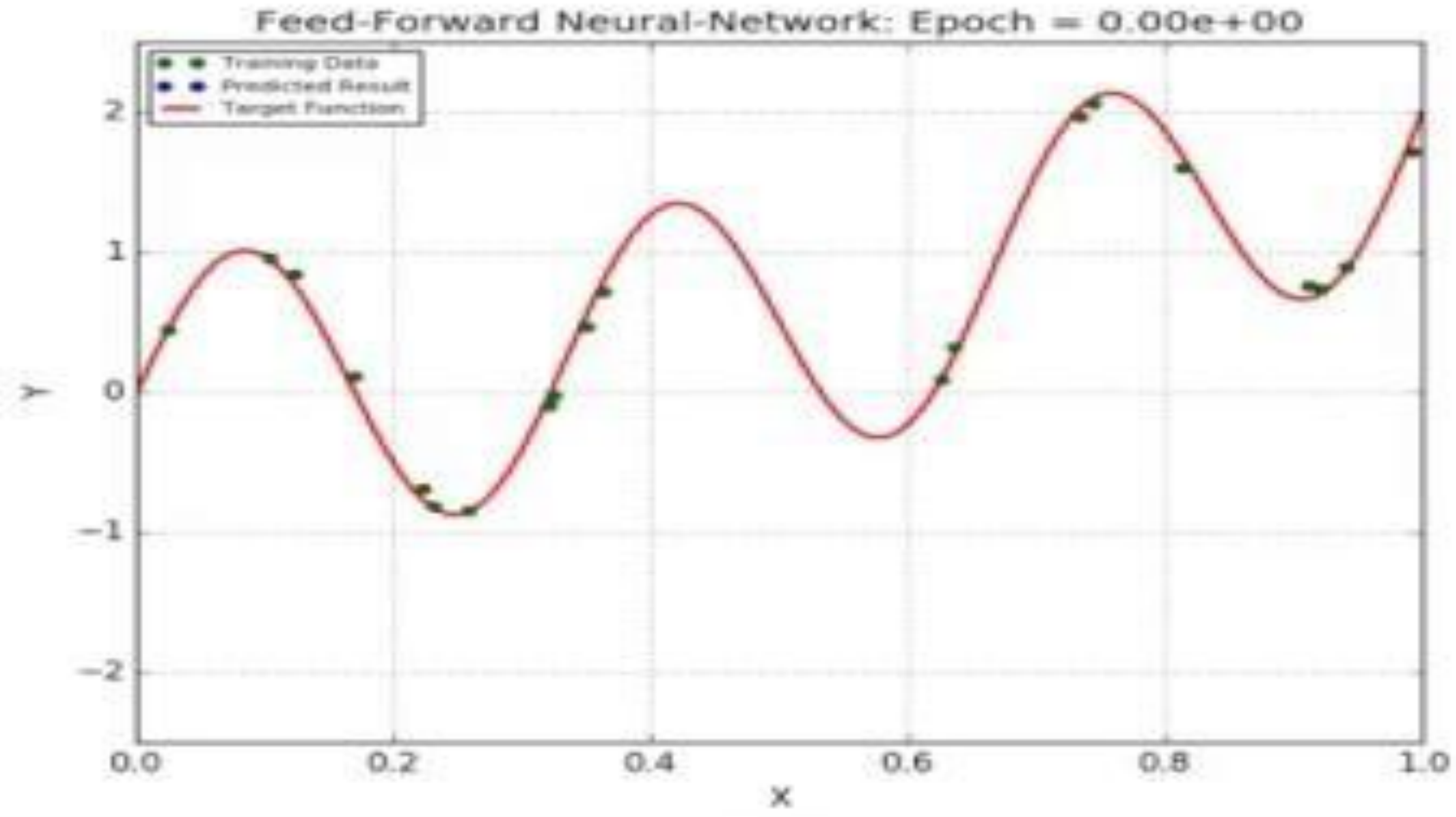
Neural Network Architecture


Output Layer

Hidden Layer 2

Hidden Layer 1

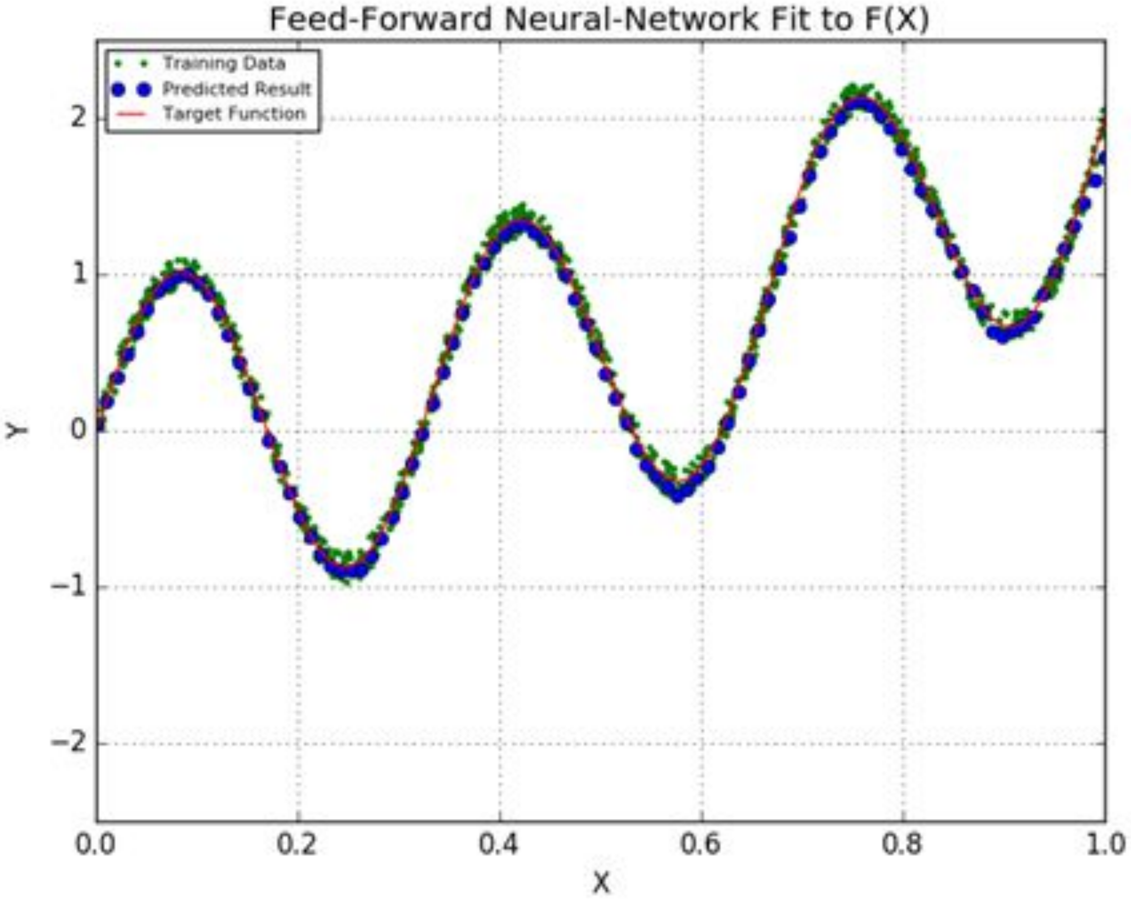
Input Layer


Feed-Forward Neural-Network: Epoch = 0.00e+00
Training Data
Predicted Result
Target Function
y
x
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0

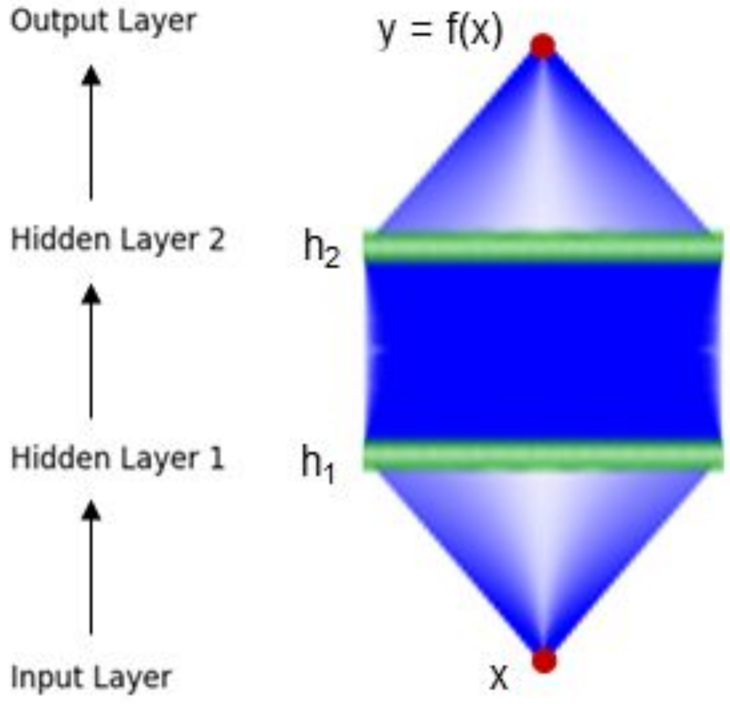
Feed-Forward Neural Networks: Example

Trained FFNN on set of 1000 samples $y_i = \sin(6\pi x_i) + 2x^2 + \xi_i$.

Stochastic Gradient Descent (SGD) used with batch size 20 and learning rate 10^{-4}.



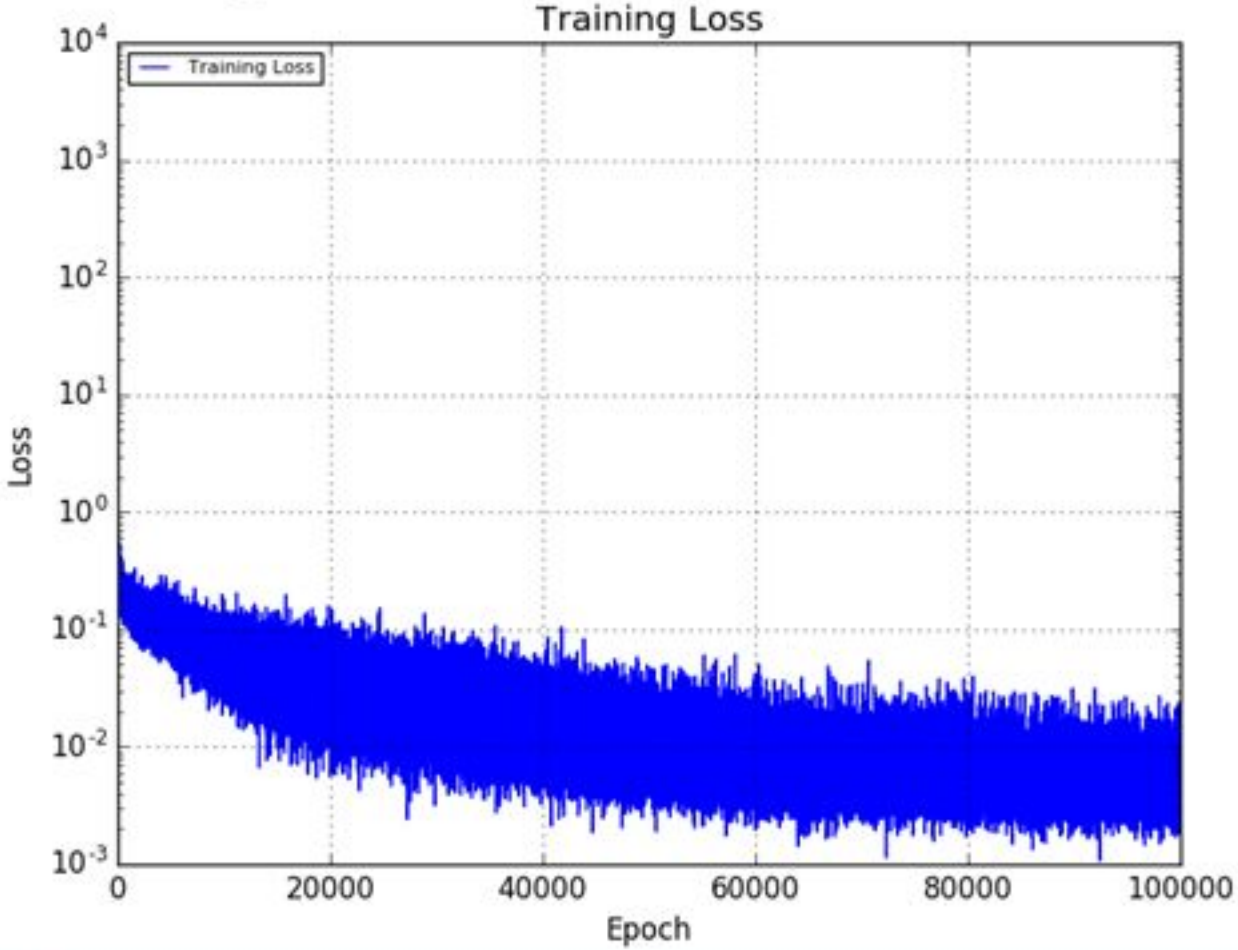
Neural Network Architecture



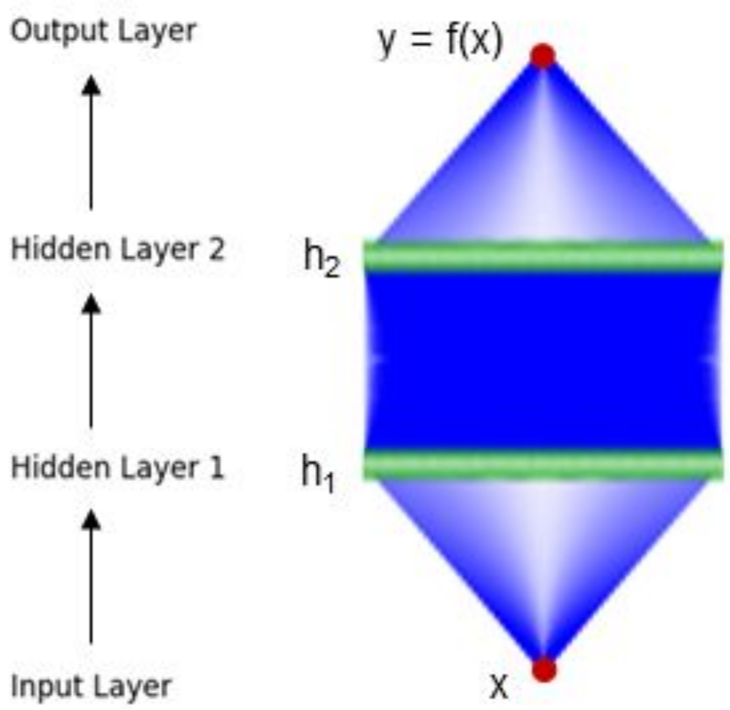
Feed-Forward Neural Networks: Example

Trained FFNN on set of 1000 samples $y_i = \sin(6\pi x_i) + 2x^2 + \xi_i$.

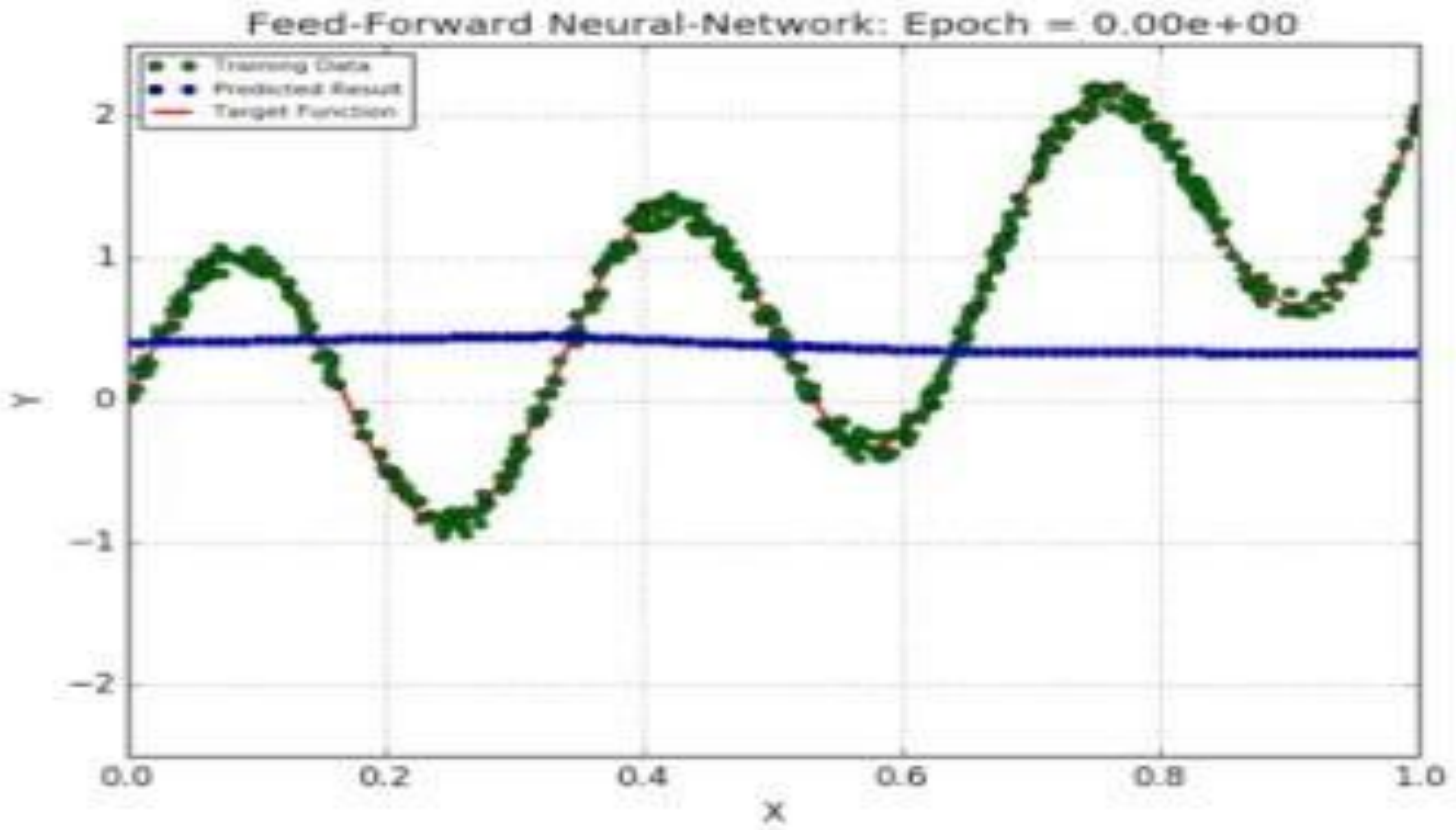
Stochastic Gradient Descent (SGD) used with batch size 20 and learning rate 10^{-4}.



Neural Network Architecture



Additional fine-tuning of hyper-parameters should be done to enhance efficiency and robustness of training.


Feed-Forward Neural-Network: Epoch = 0.00e+00
Training Data
Predicted Result
Target Function
Y
2
1
0
−1
−2
0.0
0.2
0.4
0.6
0.8
1.0
X

Summary

Neural Networks: Summary

Deep Neural Network (DNN)

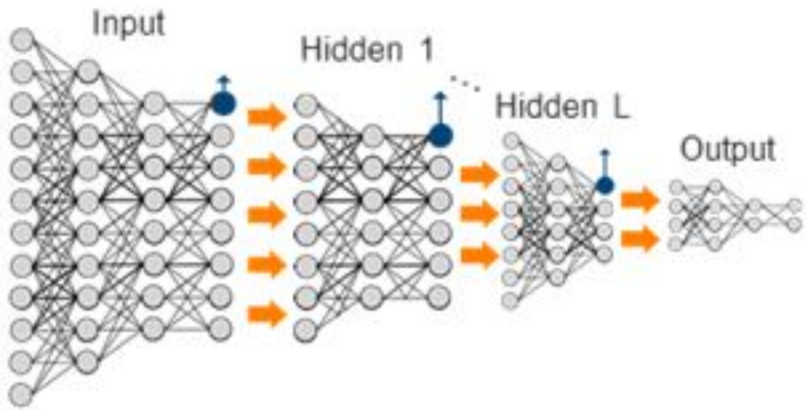
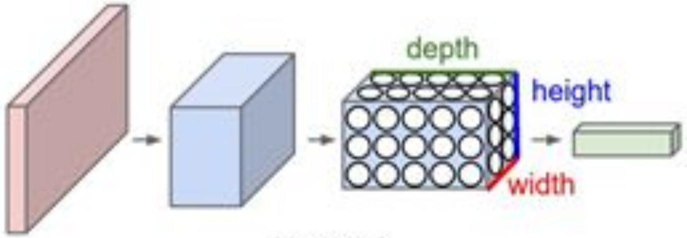
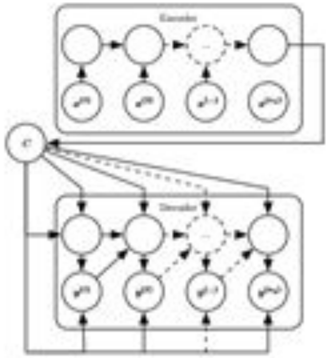


Image Classifier: Convolutional Neural Network (CNN)



LeCun 1989

Natural Language Processing
Recurrent Neural Network (RNN)



- **Neural Networks** are providing state-of-the-art results in many fields: (computer vision, natural language processing, reinforcement learning).

Alpha-Go

Google's Self-Driving Car

Neural Networks: Summary

Deep Neural Network (DNN)

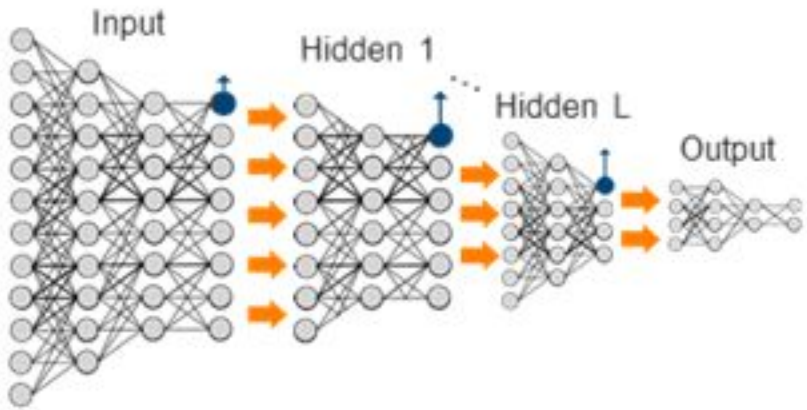
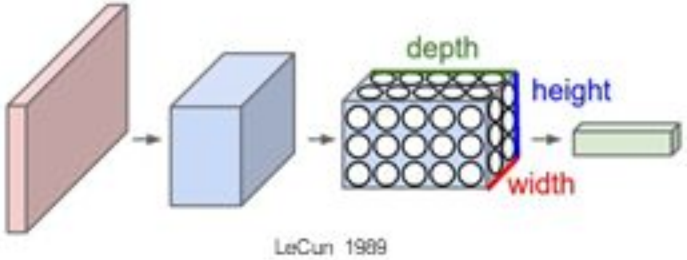
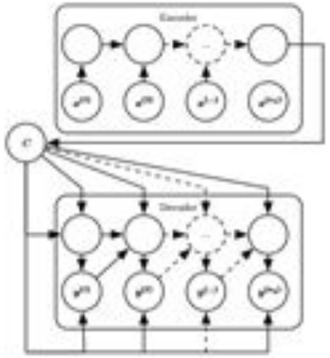


Image Classifier: Convolutional Neural Network (CNN)



Natural Language Processing
Recurrent Neural Network (RNN)



- **Neural Networks** are providing state-of-the-art results in many fields: (computer vision, natural language processing, reinforcement learning).
- **Powerful approximation properties:** target functions approximated by compositions, perform well in high dimensional spaces, many variants.

Alpha-Go

Google's Self-Driving Car

Neural Networks: Summary

Deep Neural Network (DNN)

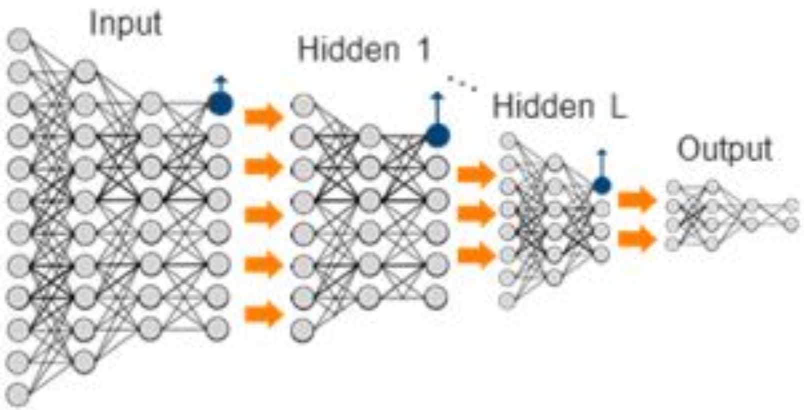
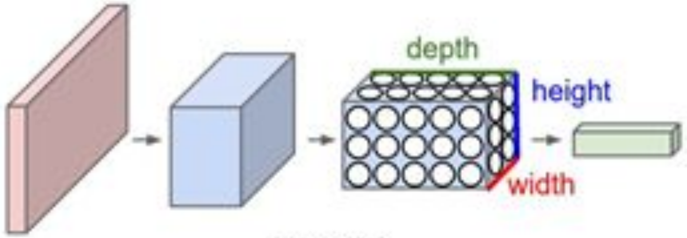
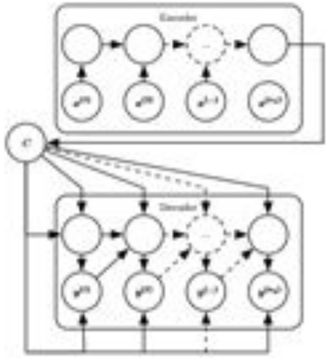


Image Classifier: Convolutional Neural Network (CNN)



LeCun 1989

Natural Language Processing
Recurrent Neural Network (RNN)



- **Neural Networks** are providing state-of-the-art results in many fields: (computer vision, natural language processing, reinforcement learning).

- **Powerful approximation properties:** target functions approximated by compositions, perform well in high dimensional spaces, many variants.

- **With appropriate learning protocols,** despite richness of NN's, seems they can be well-enough regularized to not overfit the training data.

Alpha-Go

Google's Self-Driving Car

Neural Networks: Summary

Deep Neural Network (DNN)

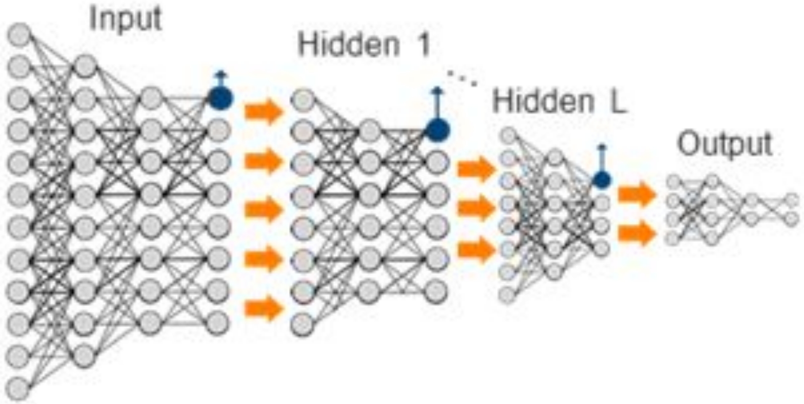
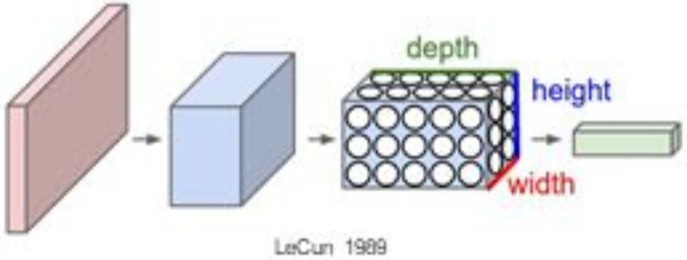
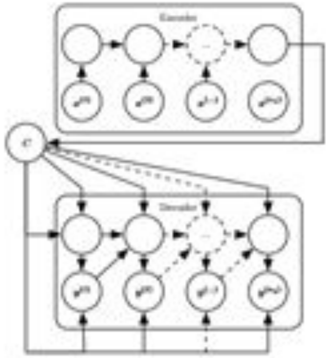


Image Classifier: Convolutional Neural Network (CNN)



Natural Language Processing
Recurrent Neural Network (RNN)



- **Neural Networks** are providing state-of-the-art results in many fields: (computer vision, natural language processing, reinforcement learning).
- **Powerful approximation properties:** target functions approximated by compositions, perform well in high dimensional spaces, many variants.
- **With appropriate learning protocols,** despite richness of NN's, seems they can be well-enough regularized to not overfit the training data.
- **Current research to better understand NN's**: choice of architectures, training protocols, approximation properties, reliability, interpretability, …

Alpha-Go

Google's Self-Driving Car