Deep learning (CSE 40301)
Principles of Deep learning (IE40801)

UNIST AIGS, CSE
Jooyeon Kim, Ph.D.

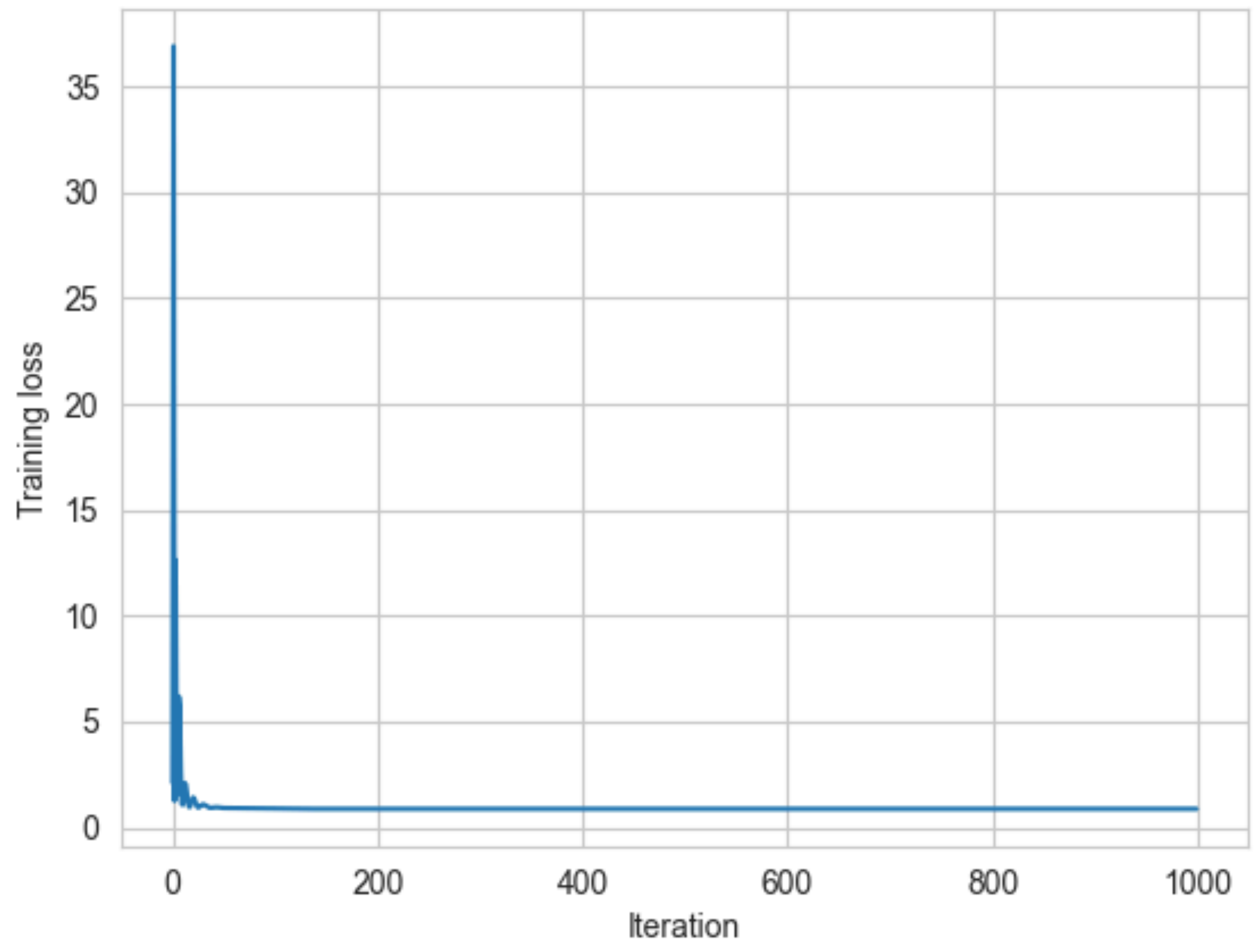
ARTIFICIAL INTELLIGENCE GRADUATE SCHOOL

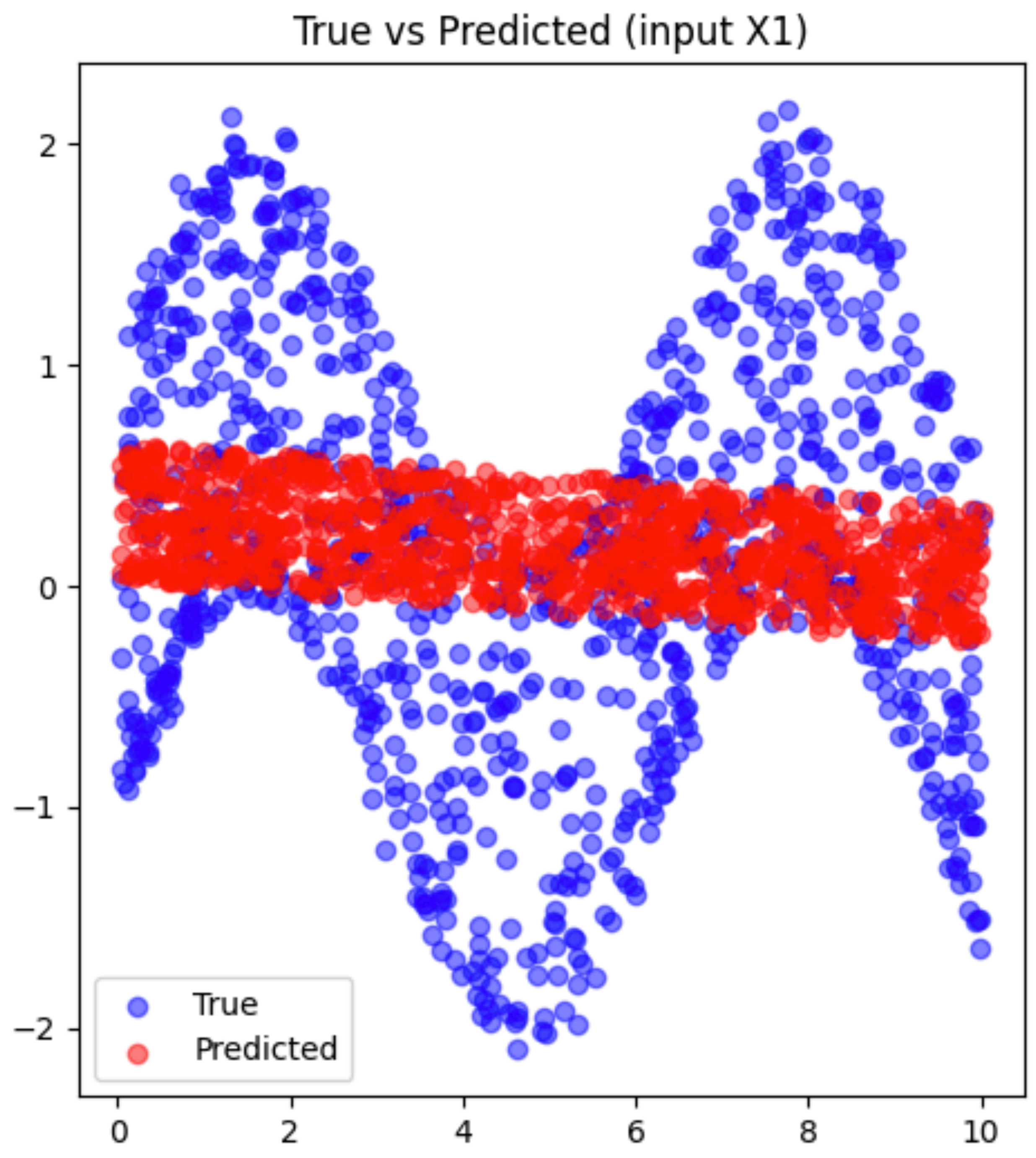
Recap

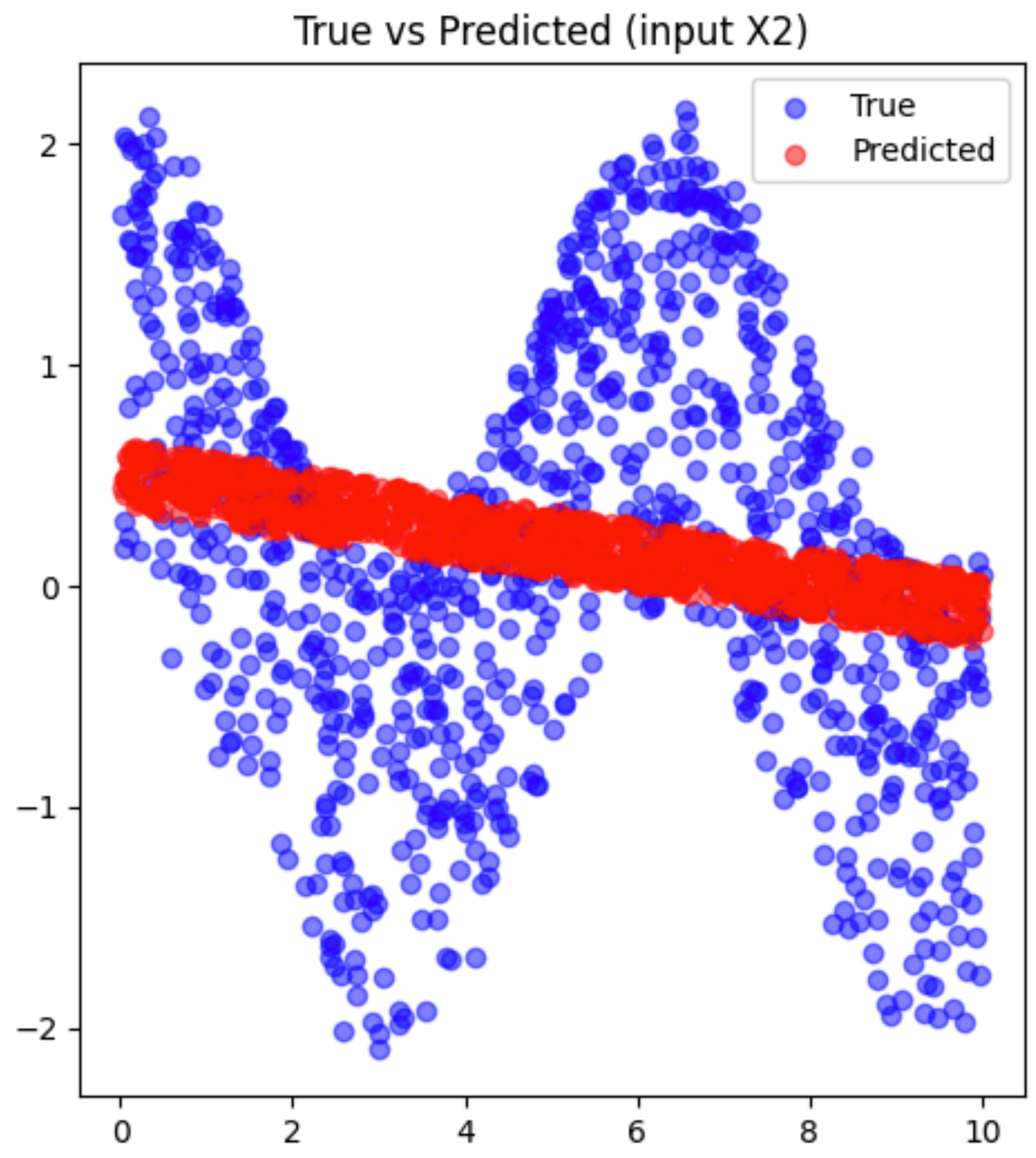
- Deep learning ~= Multi-layer perceptron (MLP) = Feed-forward neural net (FNN)
- MLP is just a spreadings and stackings of perceptrons
- Perceptron is just a linear regression with nonlinearity (activation function)
- In fact, MLP without nonlinearity boils down to linear regression
- -> Deep learning minus nonlinearity = linear regression ☺

Experimental results

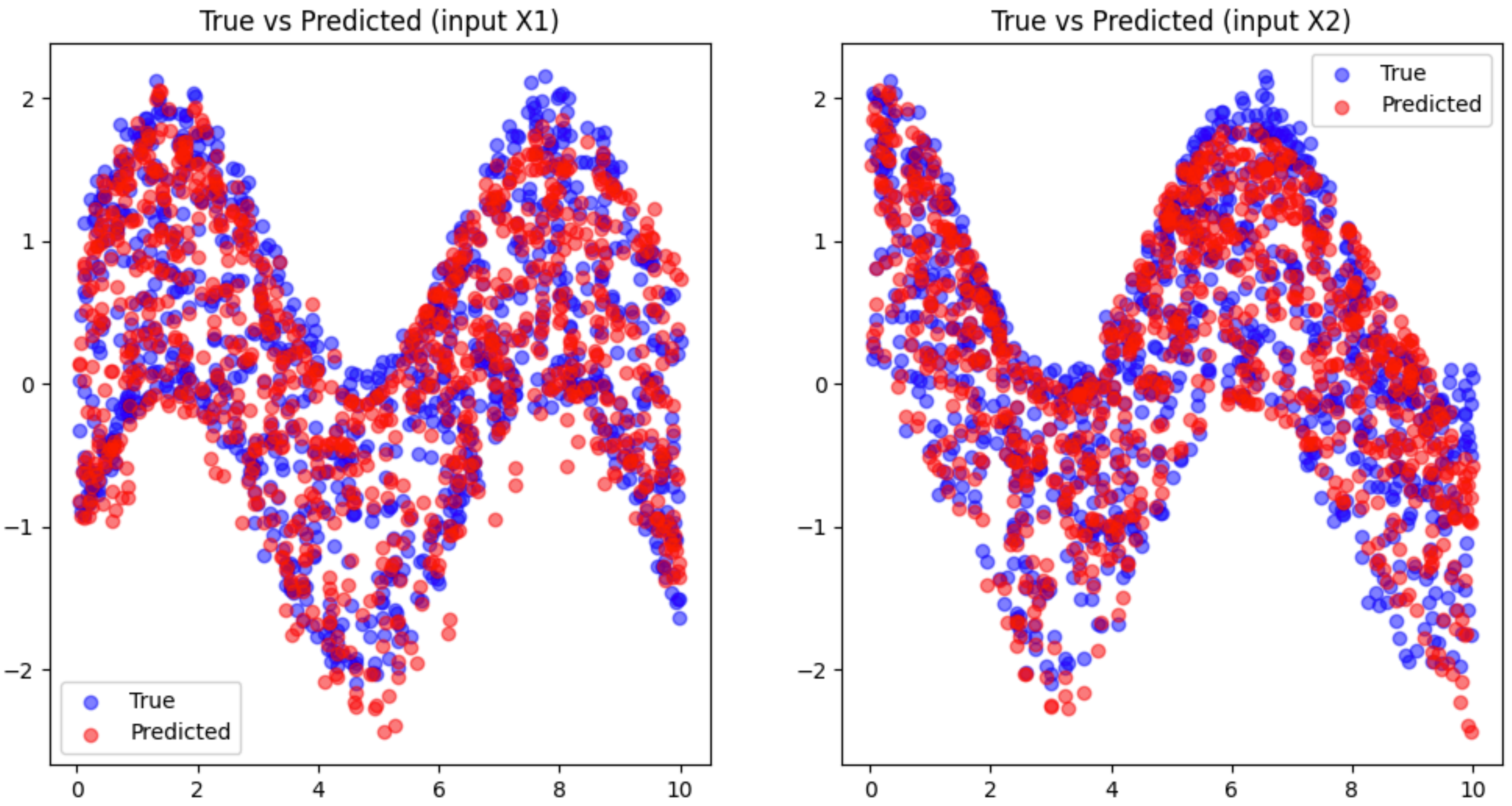
- 2-hidden layer MLP with 128 hidden perceptrons is indeed just a linear regression!



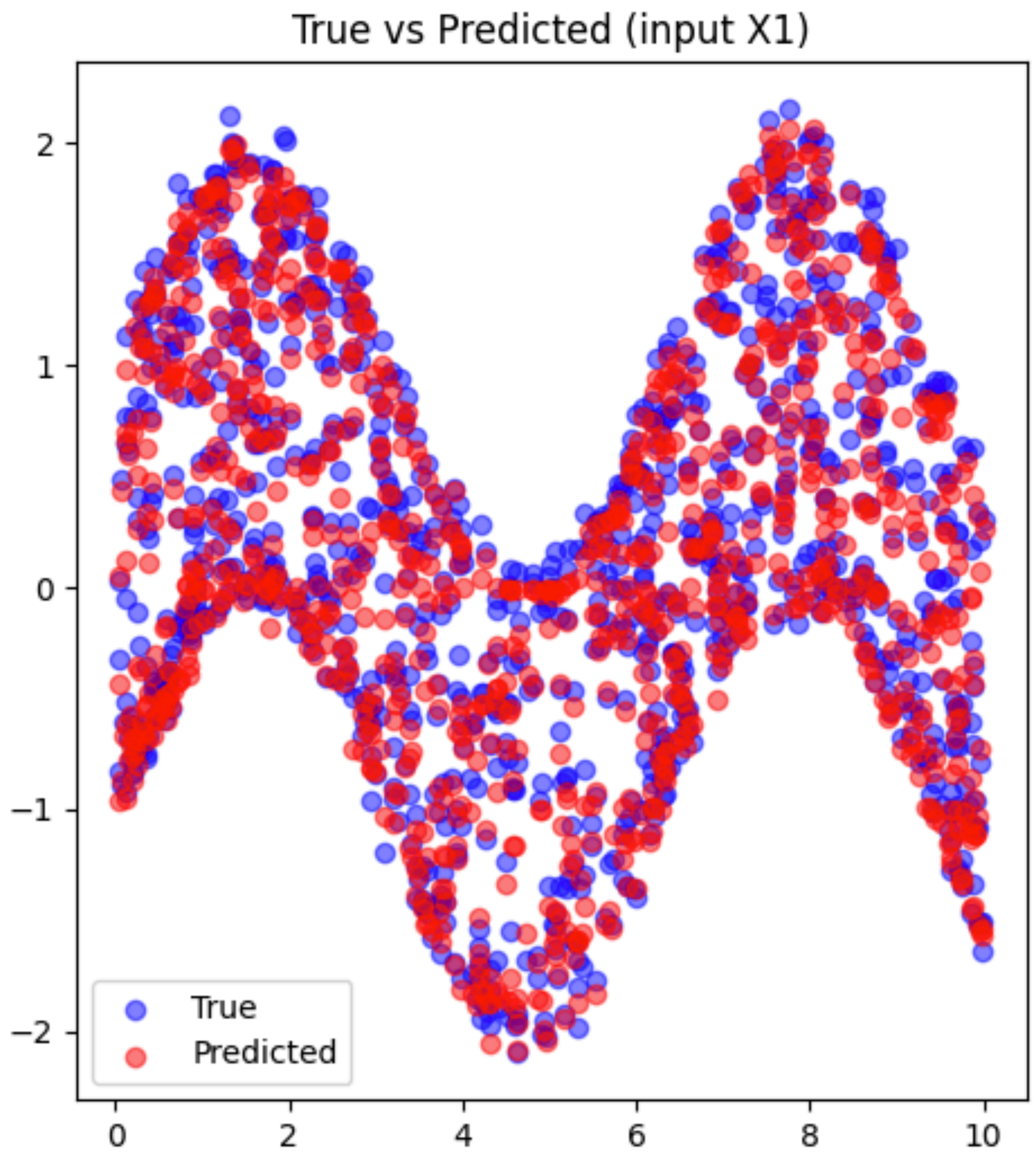


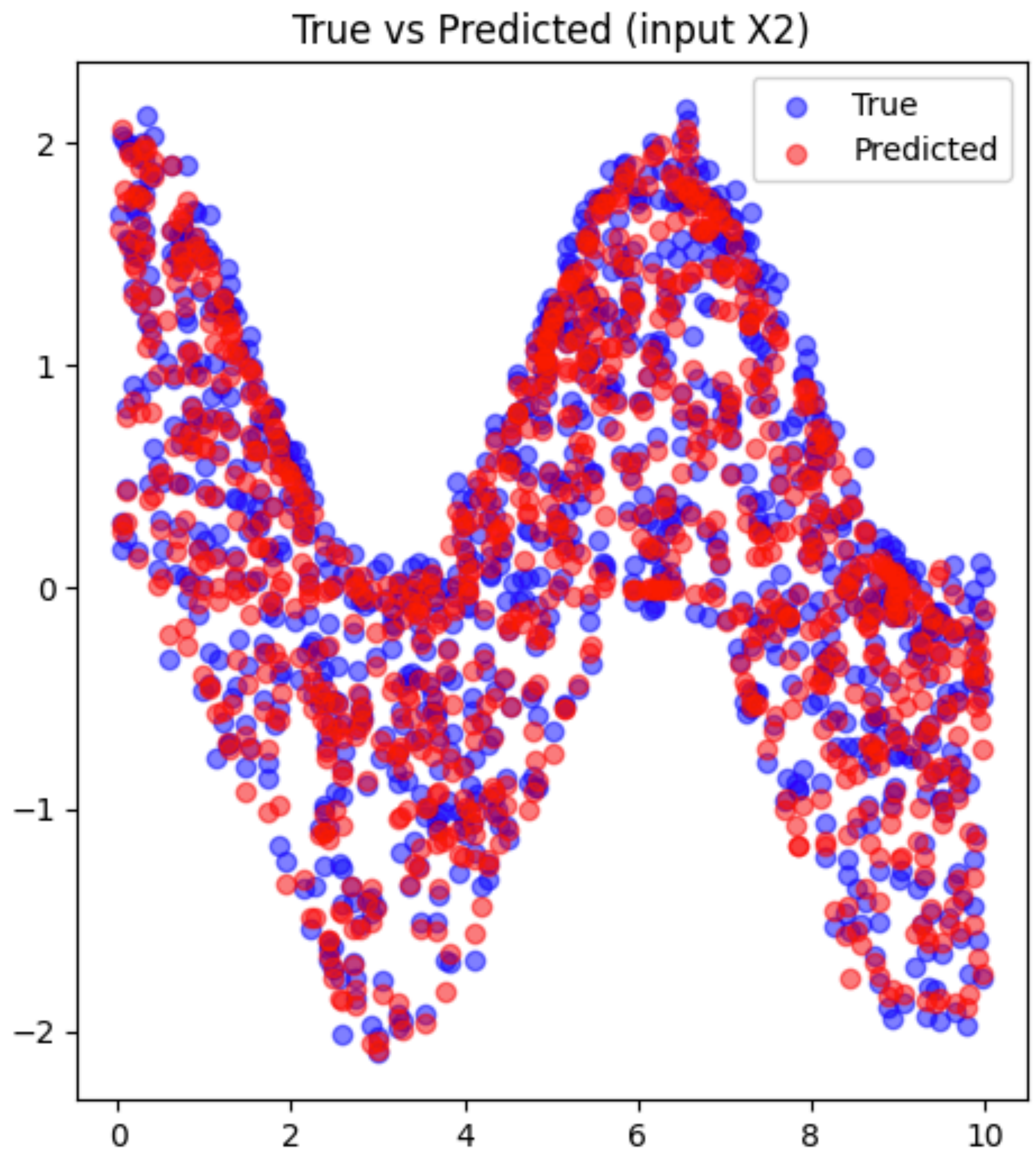


W/ Sigmoid


True vs Predicted (input X1)
True vs Predicted (input X2)
True
Predicted
2
1
0
−1
−2
0
2
4
6
8
10

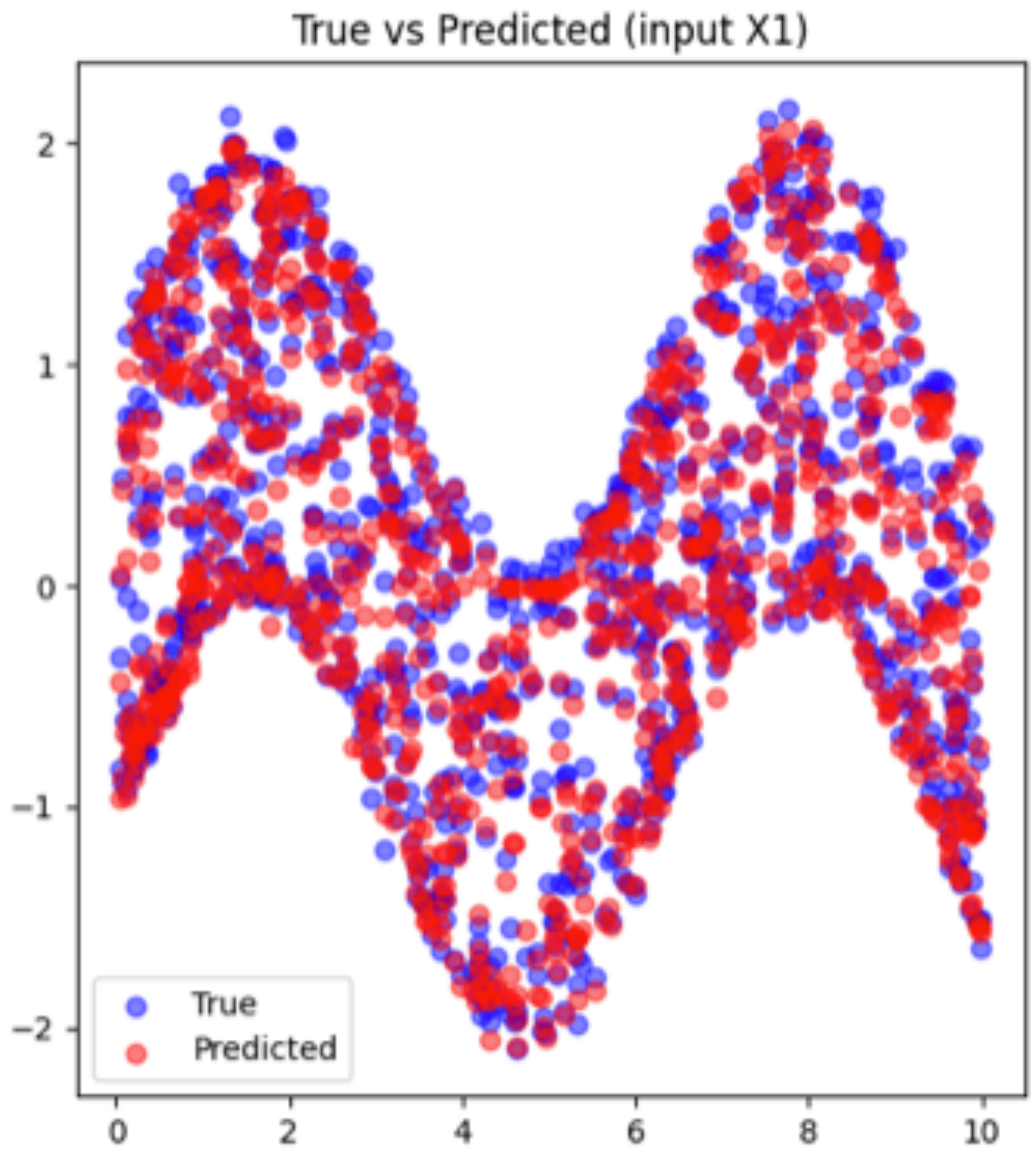
W/ ReLU


True vs Predicted (input X1)
True
Predicted
0
2
4
6
8
10
2
1
0
−1
−2


True vs Predicted (input X2)
True
Predicted
0
2
4
6
8
10
2
1
0
−1
−2

MLPs

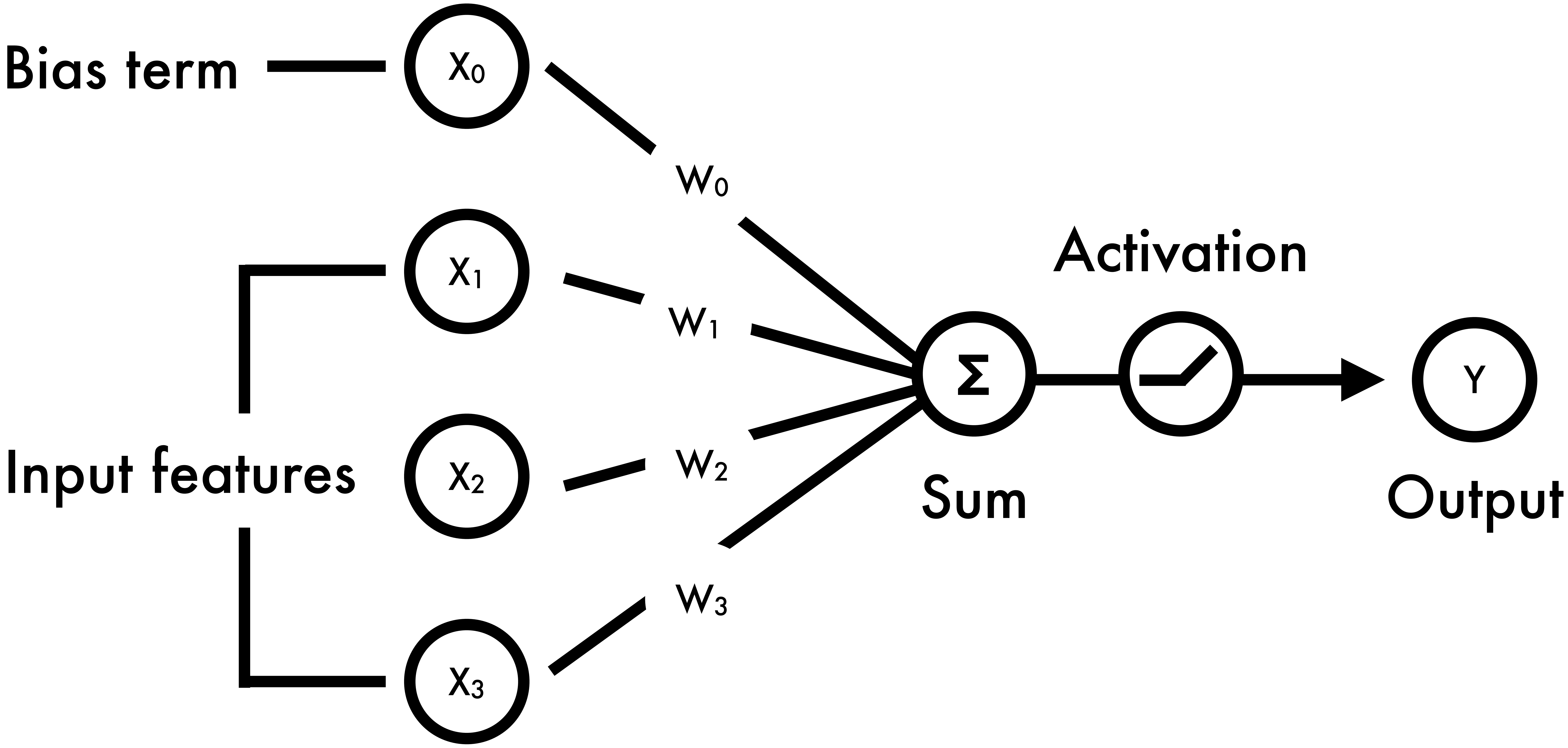
- MLP is capable of fitting the sine function (——>>)
  - 2D input and output
  - Bounded 0 ~ 10
  - 1,000 datapoints
- In fact, a large enough MLP can fit an arbitrary, extremely complex functions!
  - Universal approximation theorem
- With slight modifications, it can model images, sequences, time-series data, … !!
  - Using CNNs, RNNs, transformers, …
  - (Which mostly reduces to MLPs anyways ☺)



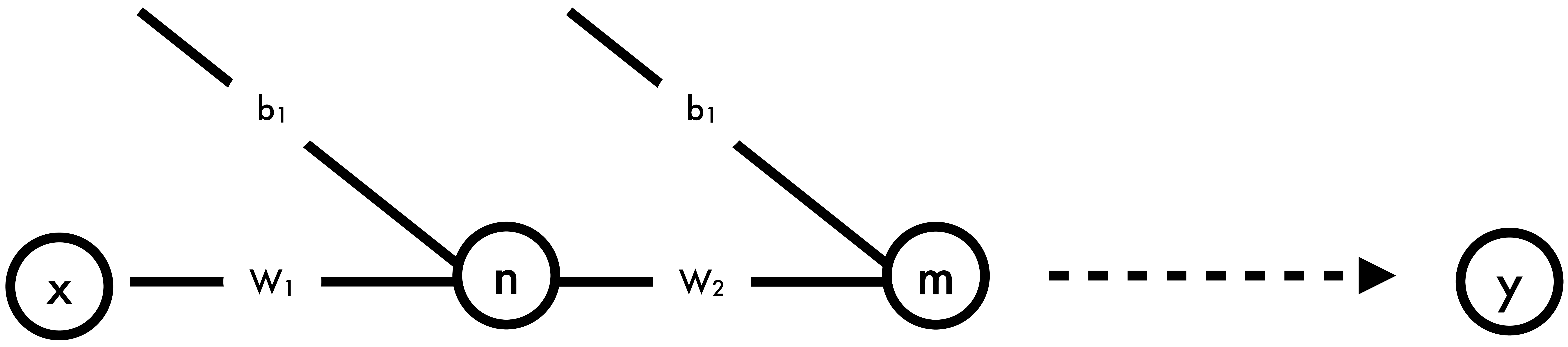
MLP: What's the issue then?

- No closed form solution!
- Q: MLP without nonlinearity: Do we have a closed-form solution for this setting?
- We need to use some sort of gradient descent to iteratively update all parameters in MLP
- Backpropagation = gradient descent + chain rule

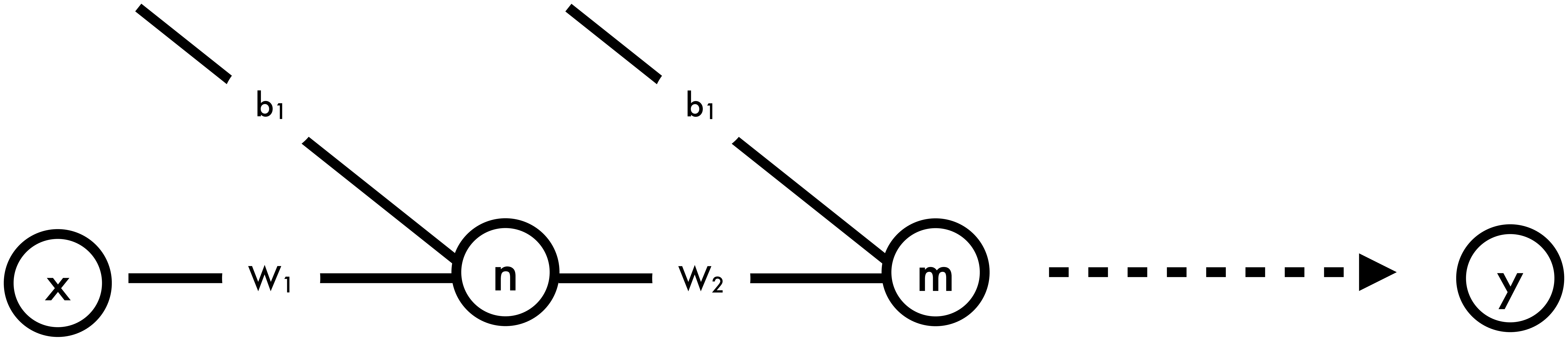
Remember this?



The simplest MLP

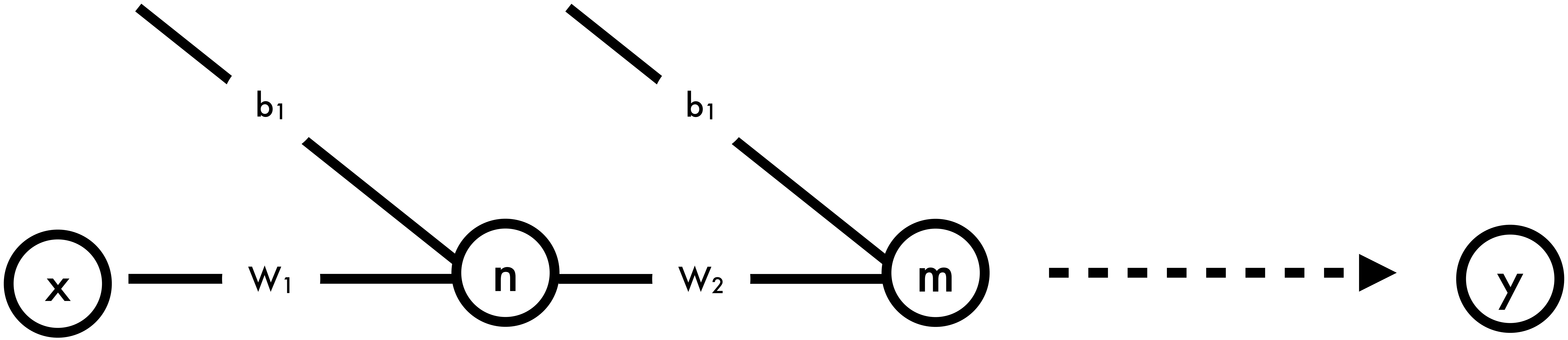


The simplest MLP



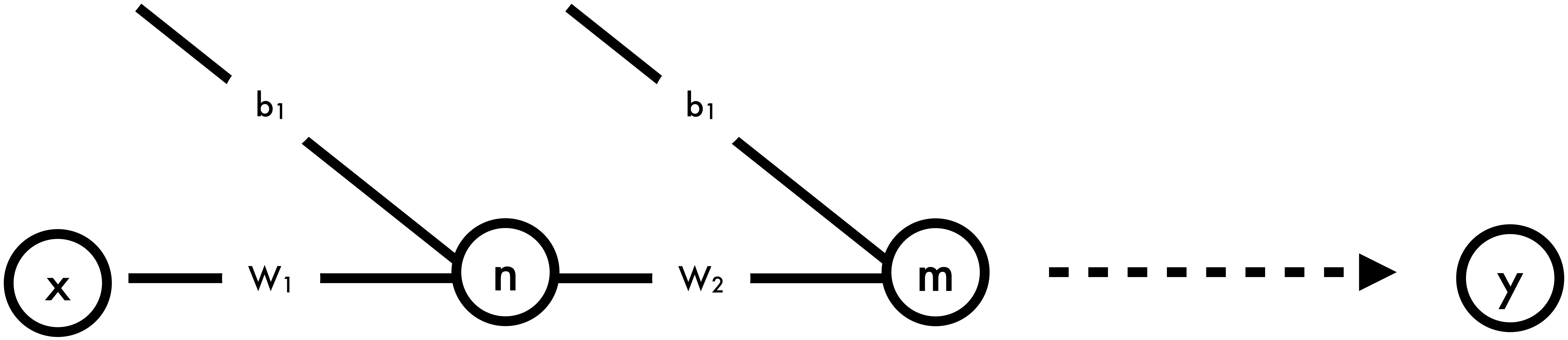
- Input x, output y
  - Both are given!
  - There will be multiple tuples of (x, y), i.e., the number of datapoints

The simplest MLP



- Input x, output y
  - Both are given!
  - There will be multiple tuples of (x, y), i.e., the number of datapoints
- We want to find the optimal values w1, w2, b1, b2, …

The simplest MLP

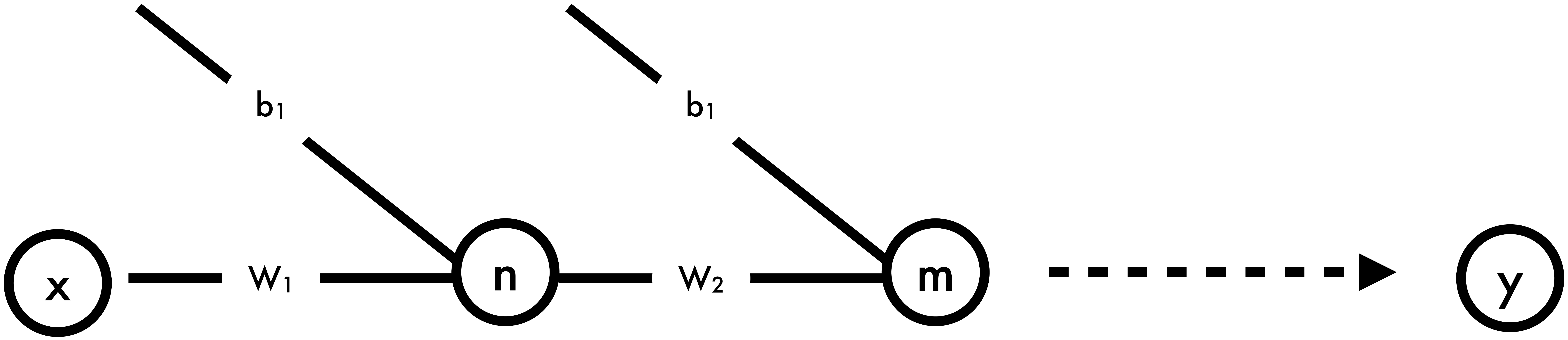


- Input x, output y
  - Both are given!
  - There will be multiple tuples of (x, y), i.e., the number of datapoints
- We want to find the optimal values w1, w2, b1, b2, …
- m fits y: regression

The simplest MLP

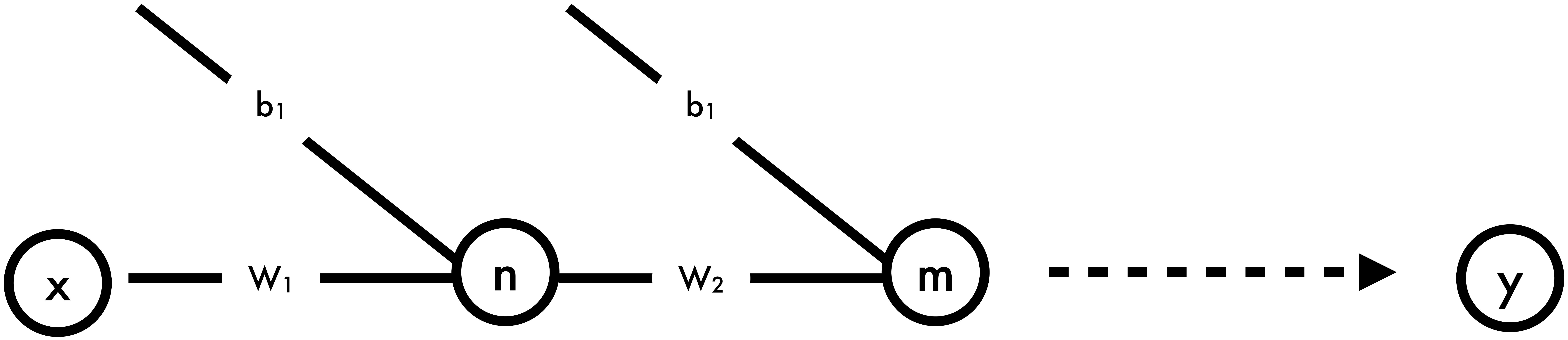
- If you know how to do backprop on this simple example, you can do backprops for ALL MLPs !
- 1,000 hidden layers, 1,000 neurons per each layer
- For billions of datapoints
- For regression and for classification
- For any kind of loss functions
- For any kind of non-linearities
  - As long as they are differentiable

The simplest MLP



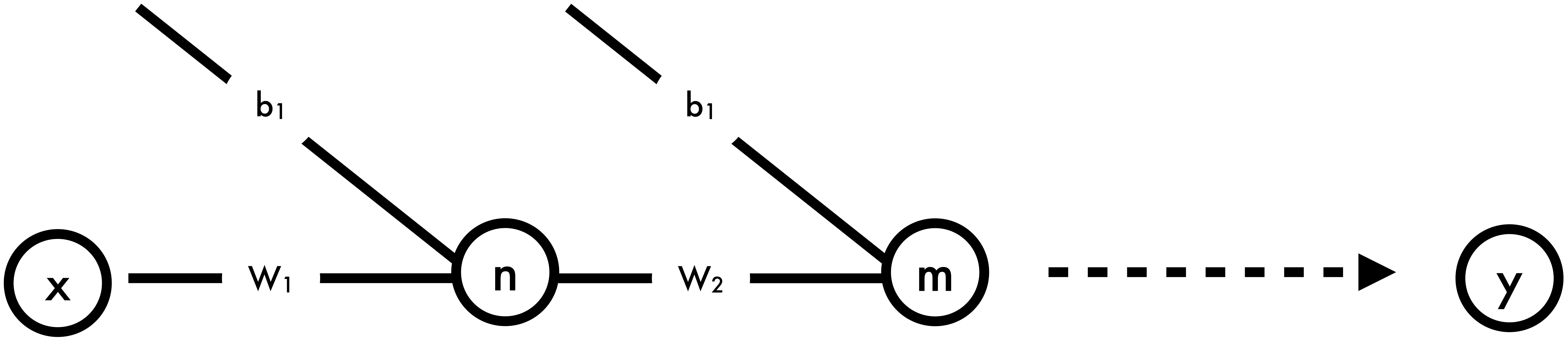
- Loss L = (y - m)^2
- m = \sigmoid(w2 n + b2)
- n = \sigmoid(w1 n + b1)

The simplest MLP



- Loss L = (y - m)^2
- m = \sigmoid(z2)
- z2 = w2 n + b2
- n = \sigmoid(w1 n + b1)

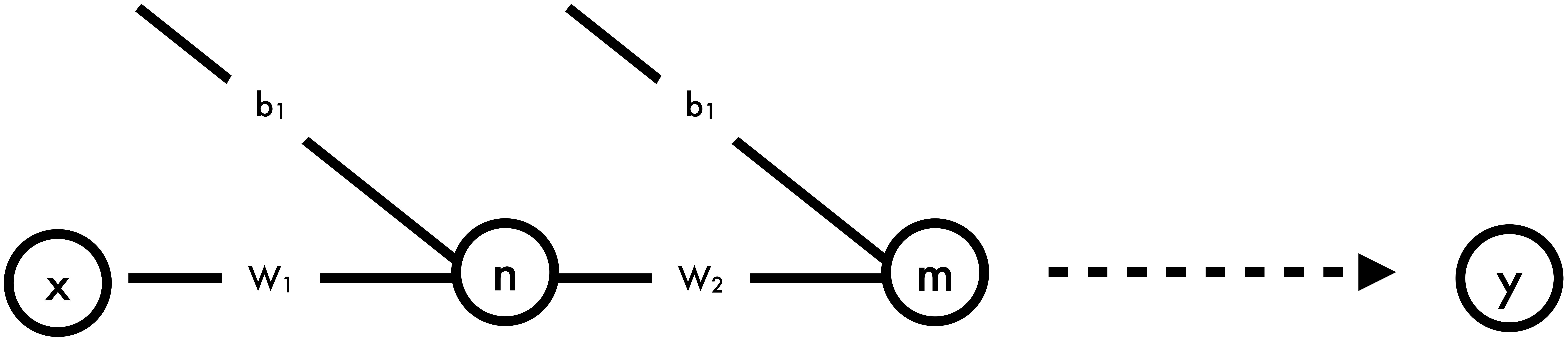
The simplest MLP



- Loss L = (y - m)^2
- m = \sigmoid(z2)
- z2 = w2 n + b2
- n = \sigmoid(w1 n + b1)

- For w2, we need to compute
  - dL / dw2
- Gradient descent:
  - w2 = w2 - \eta * (dL / dw2)
  - \eta is a learning rate (fixed)

The simplest MLP

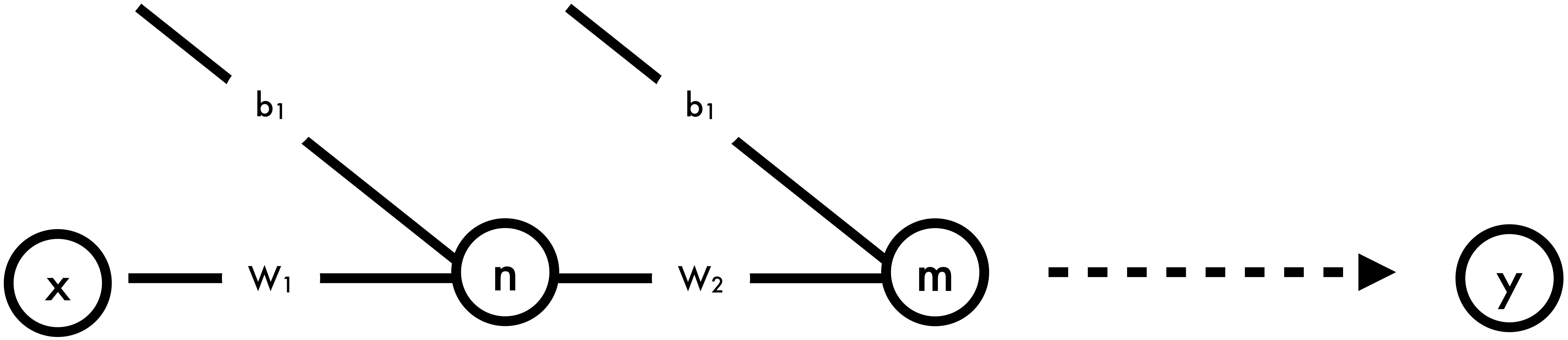


- Loss L = (y - m)^2
- m = \sigmoid(z2)
- z2 = w2 n + b2
- n = \sigmoid(w1 n + b1)

- For w2, we need to compute
  - dL / dw2
- Gradient descent:
  - w2 = w2 - \eta * (dL / dw2)
  - \eta is a learning rate (fixed)

- dL / dw2 =>
- dL / dm =>
- dm / dz2 * dL / dm
- dz2 / dw2 * dm / dz2 * dL / dm

The simplest MLP



- Loss L = (y - m)^2
- m = \sigmoid(z2)
- z2 = w2 n + b2
- n = \sigmoid(w1 n + b1)

- dz2 / dw2 * dm / dz2 * dL / dm

n

\sigma\prime(m)
= \sigma(m) * (1 - \sigma(m))

2 (y - m)

The simplest MLP

$$\frac{\partial L}{\partial w_2} = \frac{\partial z_2}{\partial w_2} \cdot \frac{\partial m}{\partial z_2} \cdot \frac{\partial L}{\partial m} \quad \cdots (1)$$

$n \qquad \sigma'(m) \qquad 2(y-m)$

$= \sigma(m)(1-\sigma(m))$

$$\frac{\partial L}{\partial b_2} = \frac{\partial z_2}{\partial b_2} \cdot \frac{\partial m}{\partial z_2} \cdot \frac{\partial L}{\partial m} \qquad * \text{ from } (1)$$

$1 \qquad$ same* $\qquad$ same*

$$\frac{\partial L}{\partial w_1} = \frac{\partial z_1}{\partial w_1} \cdot \frac{\partial n}{\partial z_1} \cdot \frac{\partial z_2}{\partial n} \cdot \frac{\partial m}{\partial z_2} \cdot \frac{\partial L}{\partial m}$$

$x \qquad \sigma'(n) \qquad w_2 \qquad \sigma'(m) \qquad 2(y-m)$

$$\frac{\partial L}{\partial b_1} = \text{省略}$$

The simplest MLP

$$\frac{\partial L}{\partial w_1} = \frac{\partial z_1}{\partial w_1} \cdot \frac{\partial n}{\partial z_1} \cdot \frac{\partial z_2}{\partial n} \cdot \frac{\partial m}{\partial z_2} \cdot \frac{\partial L}{\partial m}$$

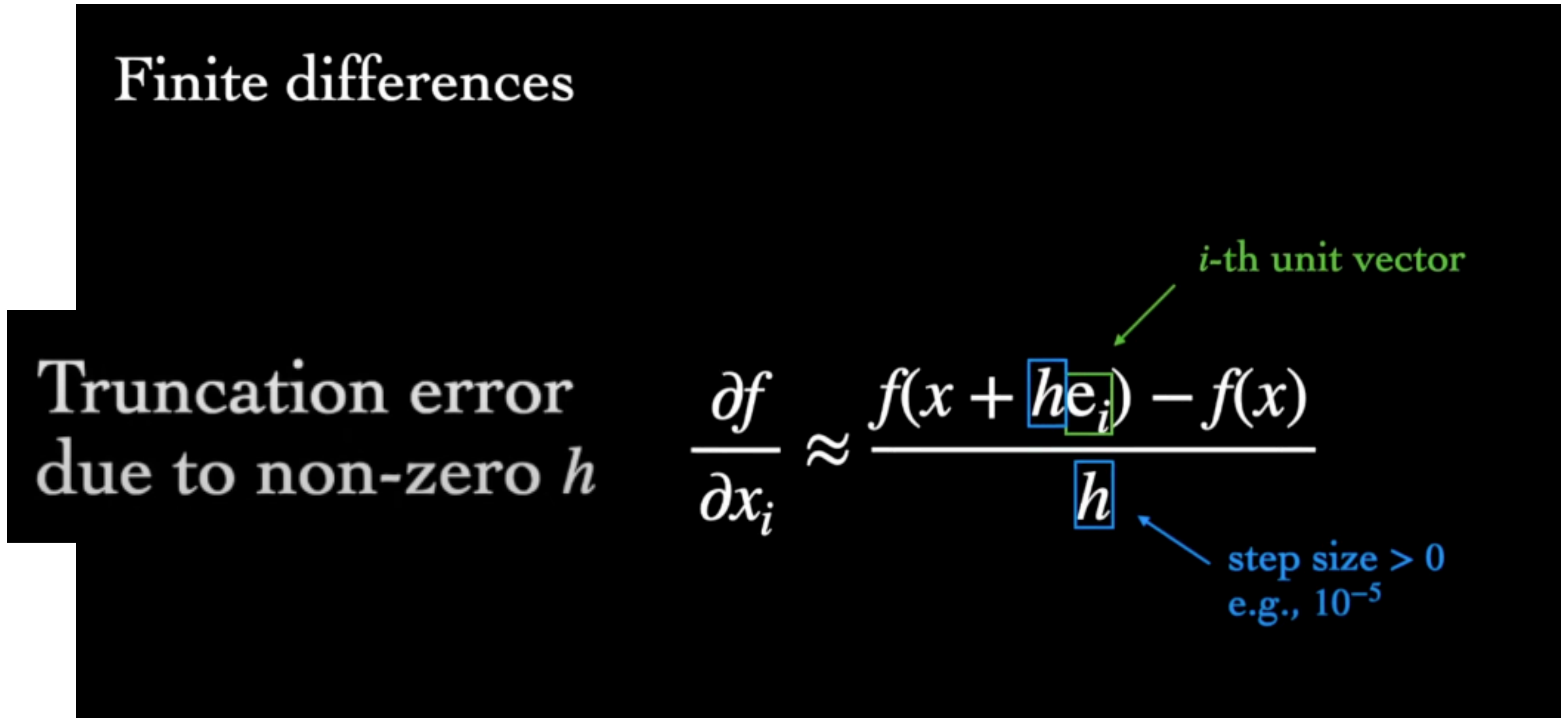
$$x \quad \sigma'(n) \quad w_2 \quad \sigma'(m) \quad 2(y-m)$$

Questions and discussions

1. MLP without nonlinearity: Do we have a closed-form solution for this setting?

2. Derivative of sigmoid function $\sigma(x)$=1 / (1+e^{−x})

3. PyTorch, TensorFlow, ….: automatic differentiation??
   1. “Numerical”
   2. “Symbolic” differentiation — Expression swell
   3. “automatic”



4. Different optimization methods?
   1. “Optimizers”
   2. SGD, RMSProp, Adam, AdamW, …

5. Derivative of Cross entropy, …

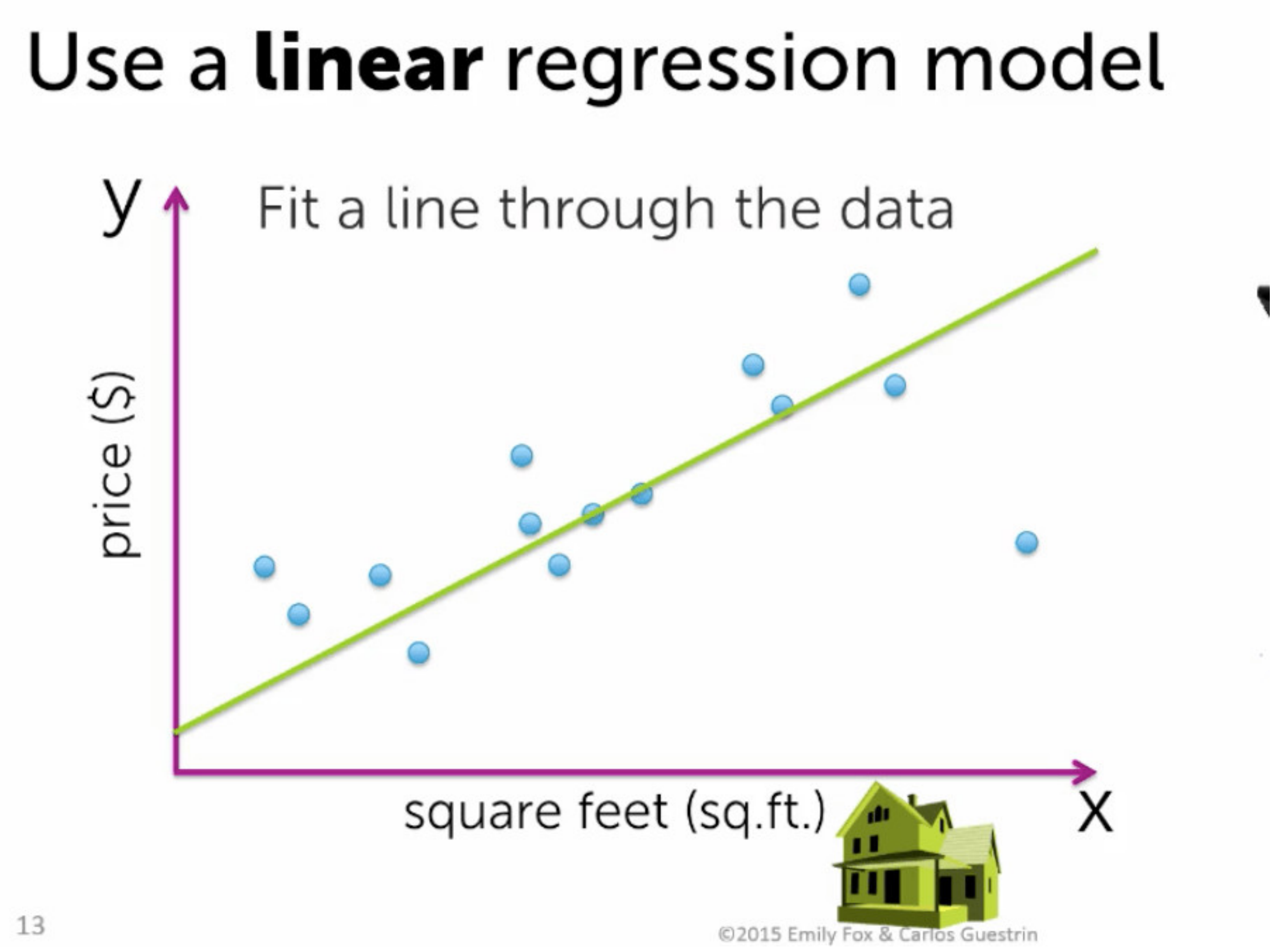
https://www.youtube.com/watch?v=wG_nF1awSSY

Questions and discussions

1. Multiple datapoints???
2. MLP without nonlinearity: Do we have a closed-form solution for this setting?
3. Derivative of sigmoid function $\sigma(x)$=1 / (1+e^{−x})
4. PyTorch, TensorFlow, ….: automatic differentiation??
5. Different optimization methods?
6. Derivative of Cross entropy, …
7. Local optimum, overfitting
8. Bias and variance, …

Brief history of deep learning

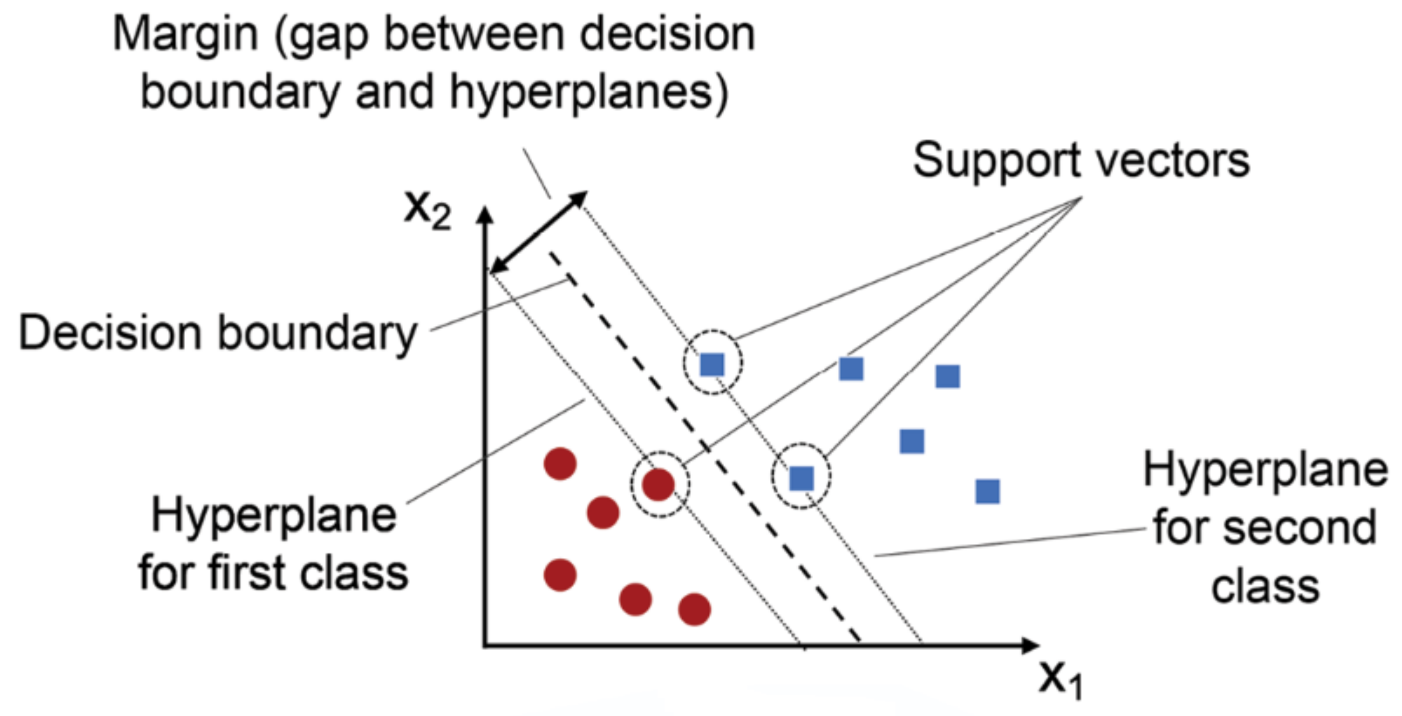
- <u>**Linear regression** (Legendre & Gauss, 1805) and **logistic regression**</u>
  - Simplest forms of regression or classification to model the relationship between input variables and a continuous output or a label
  - Example: Predicting house prices based on features like area, number of rooms.
  - Example: Predicting if an email is spam or not.
  - More statistics than AI (ML)



* https://www.linkedin.com/pulse/predicting-house-prices-using-linear-regression-along-muhammet-ergenc

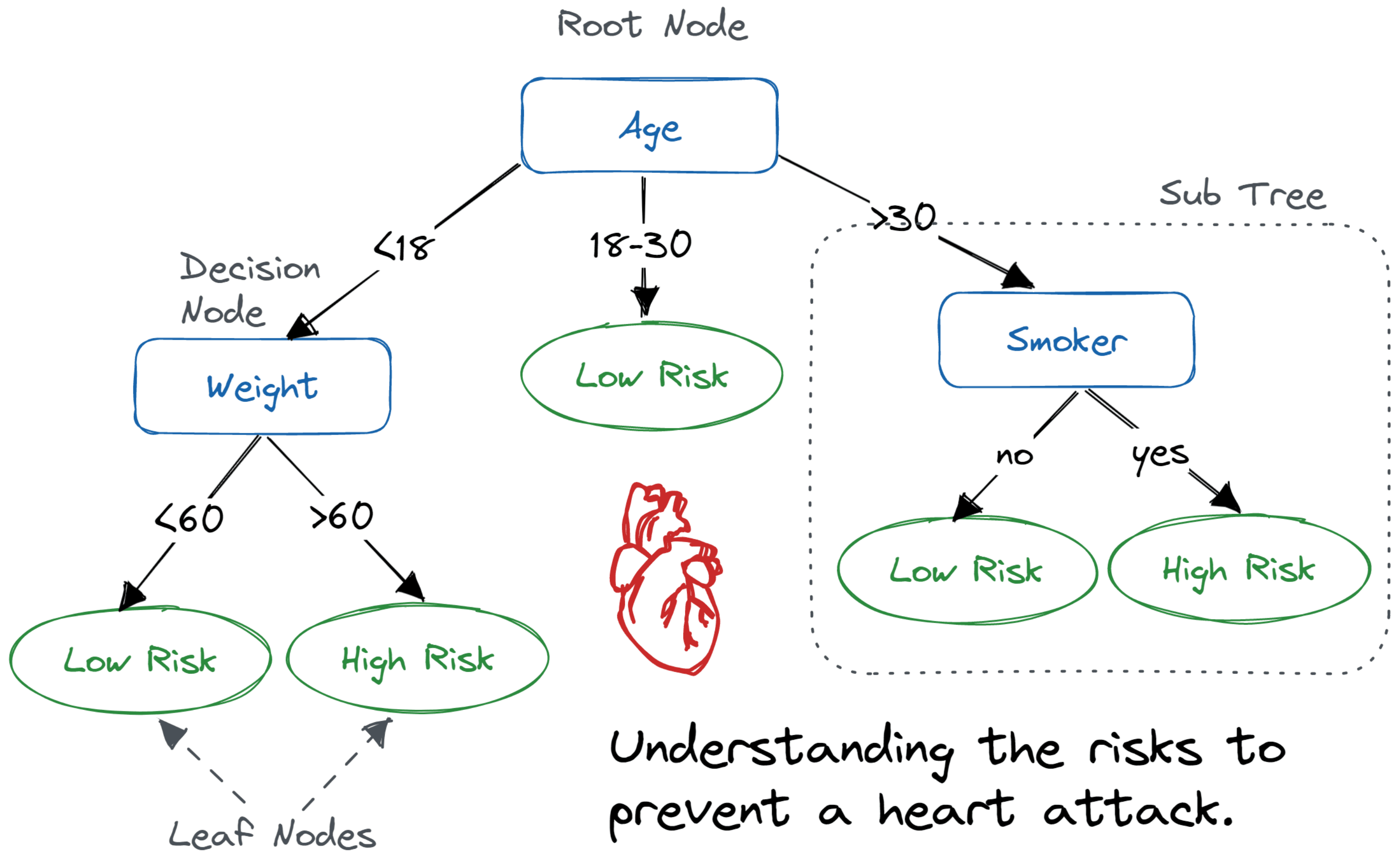
Brief history of deep learning

- **Support Vector Machines** (SVM) (Vladimir Vapnik, 1990s)
  - A powerful classifier that finds the hyperplane separating different classes with the maximum margin
  - Used in image classification, handwriting recognition, etc.



https://vitalflux.com/classification-model-svm-classifier-python-example/

- **Decision Trees and Ensembles**
  - Tree-based methods and advancements in ensemble methods like Random Forests and Gradient Boosting
  - Limitations 1: Heavy reliance on handcrafted features.
  - Limitations 2: Poor scalability and limited ability to handle large, complex data like images or raw text.



https://www.datacamp.com/tutorial/decision-tree-classification-python

Before the deep learning take-off

- Before 2012
  - SVM-variants were the model-of-choice !
  - With handcrafted features such as SIFT (Scale-Invariant Feature Transform) or HOG (Histogram of Oriented Gradients)
- Since 2012
  - AlexNet breakthrough in 2012
  - Dramatically changed the landscape
    - reducing error rates by a significant margin
    - surpassed traditional methods in virtually every domain,
    - from image classification to natural language processing.

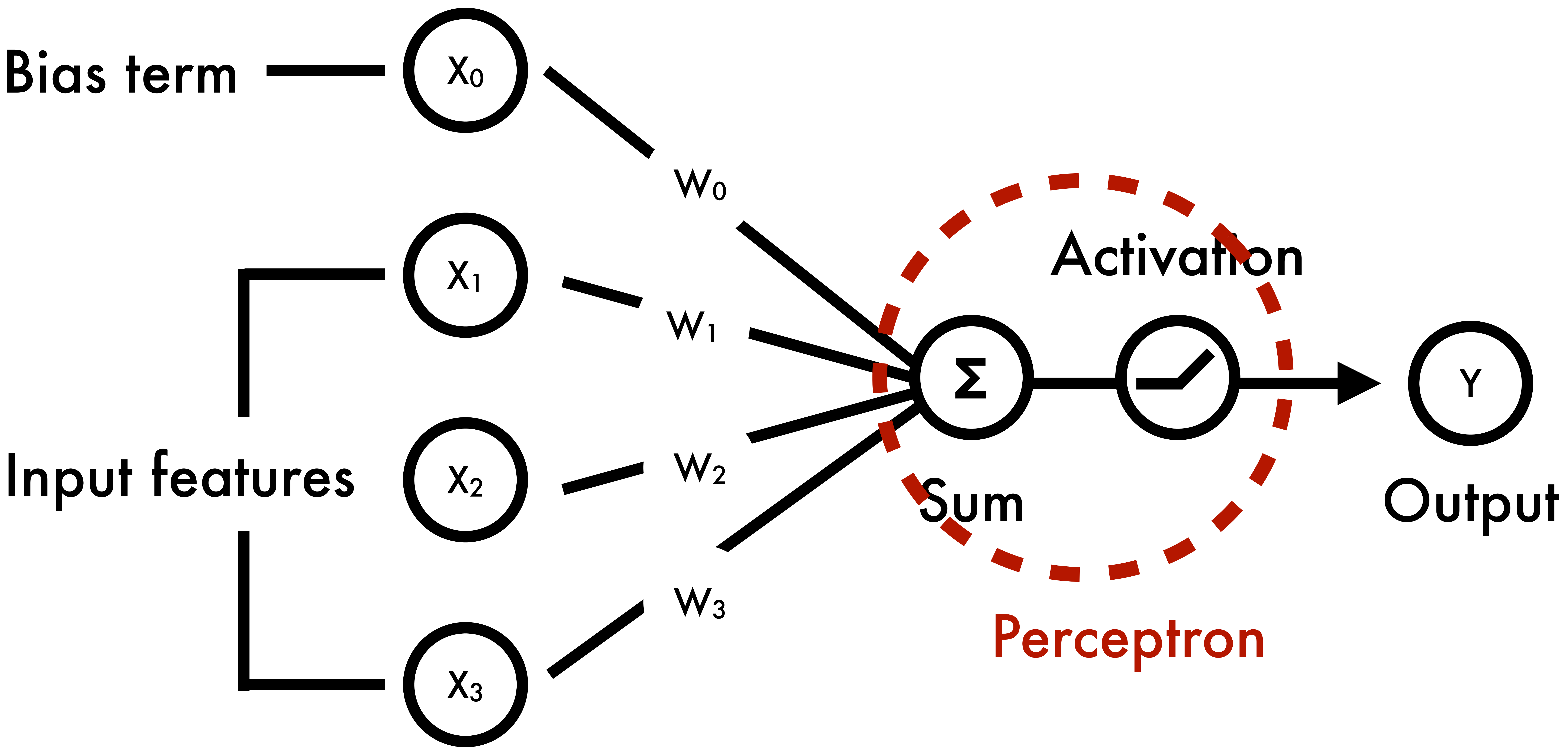
ImageNet Leaderboard: Pre-Deep Learning & Early Deep Learning Models

Year	Model	Top-5 Error Rate	Notes
2010	SIFT + SVM	28.2%	SVM + handcrafted features (e.g., SIFT, HOG). First ImageNet competition.
2011	Improved SIFT + Fisher Vectors	25.7%	Handcrafted features with Fisher vectors. Top-performing non-deep model.
2011	Deep Learning (LeCun's Lab)	~26%	Early CNN approach from LeCun's lab. Limited computational power.
2011	Handcrafted + Shallow CNNs	~25%	Small CNNs combined with handcrafted features, still underperforming.
2012	AlexNet (Deep CNN)	16.4%	First deep learning breakthrough. Massive performance leap (8.5% gain).
2013	ZFNet (Deep CNN)	14.8%	Improved on AlexNet with better architecture (deconvolution visualization).
2014	GoogLeNet (Inception Network)	6.7%	Introduced the Inception module, made networks deeper and more efficient.
2015	ResNet (Deep Residual Network)	3.6%	Introduced residual connections to solve vanishing gradients.
2016	ResNet-152	3.0%	Extended ResNet to 152 layers, further reducing error.

Why? Scalability!

- More data
  - The availability of massive datasets (e.g., ImageNet) allowed deep learning models to generalize better by learning from a wide variety of examples
- Larger model
  - Deep learning models, especially deep neural networks, have a large number of parameters, enabling them to capture more complex patterns and relationships in the data
- More compute
  - The rise of powerful GPUs and specialized hardware (e.g., TPUs) made it feasible to train large models efficiently, overcoming previous computational barriers

Perceptron


Bias term
X0
W0
X1
W1
Input features
X2
W2
X3
W3
Activation
Σ
Sum
Y
Output
Perceptron

Perceptron ≈ Linear regression

Linear Regression in Matrix Form:

$$\mathbf{Y} = \mathbf{X}\mathbf{w} + \epsilon$$

Where:

- $\mathbf{Y} \in \mathbb{R}^{n \times 1}$: The vector of target values (outputs) with n samples.
- $\mathbf{X} \in \mathbb{R}^{n \times d}$: The matrix of input features (with n samples and d features). Each row is a feature vector corresponding to a data sample.
- $\mathbf{w} \in \mathbb{R}^{d \times 1}$: The vector of weights (coefficients) that we aim to learn.
- $\epsilon \in \mathbb{R}^{n \times 1}$: The vector of error terms (residuals).

Perceptron ≈ Linear regression

Linear Regression in Matrix Form:

$$\mathbf{Y} = \mathbf{X}\mathbf{w} + \epsilon$$

Where:

- $\mathbf{Y} \in \mathbb{R}^{n \times 1}$: The vector of target values (outputs) with n samples.
- $\mathbf{X} \in \mathbb{R}^{n \times d}$: The matrix of input features (with n samples and d features). Each row is a feature vector corresponding to a data sample.
- $\mathbf{w} \in \mathbb{R}^{d \times 1}$: The vector of weights (coefficients) that we aim to learn.
- $\epsilon \in \mathbb{R}^{n \times 1}$: The vector of error terms (residuals).

- **Input matrix $\mathbf{X}$:**

$$\mathbf{X} = \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1d} \\ x_{21} & x_{22} & \dots & x_{2d} \\ \vdots & \vdots & \dots & \vdots \\ x_{n1} & x_{n2} & \dots & x_{nd} \end{bmatrix}$$

  Where each row represents a sample, and each column represents a feature.

- **Weight vector $\mathbf{w}$:**

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_d \end{bmatrix}$$

  These are the coefficients for each feature.

- **Target vector $\mathbf{Y}$:**

$$\mathbf{Y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}$$

Closed-form solution using SSE

$$SSE = ||\mathbf{y} - \mathbf{X}\mathbf{w}||^2$$

$$SSE = (\mathbf{y} - \mathbf{X}\mathbf{w})^\top(\mathbf{y} - \mathbf{X}\mathbf{w})$$

$$SSE = \mathbf{y}^\top\mathbf{y} - 2\mathbf{y}^\top\mathbf{X}\mathbf{w} + \mathbf{w}^\top\mathbf{X}^\top\mathbf{X}\mathbf{w}$$

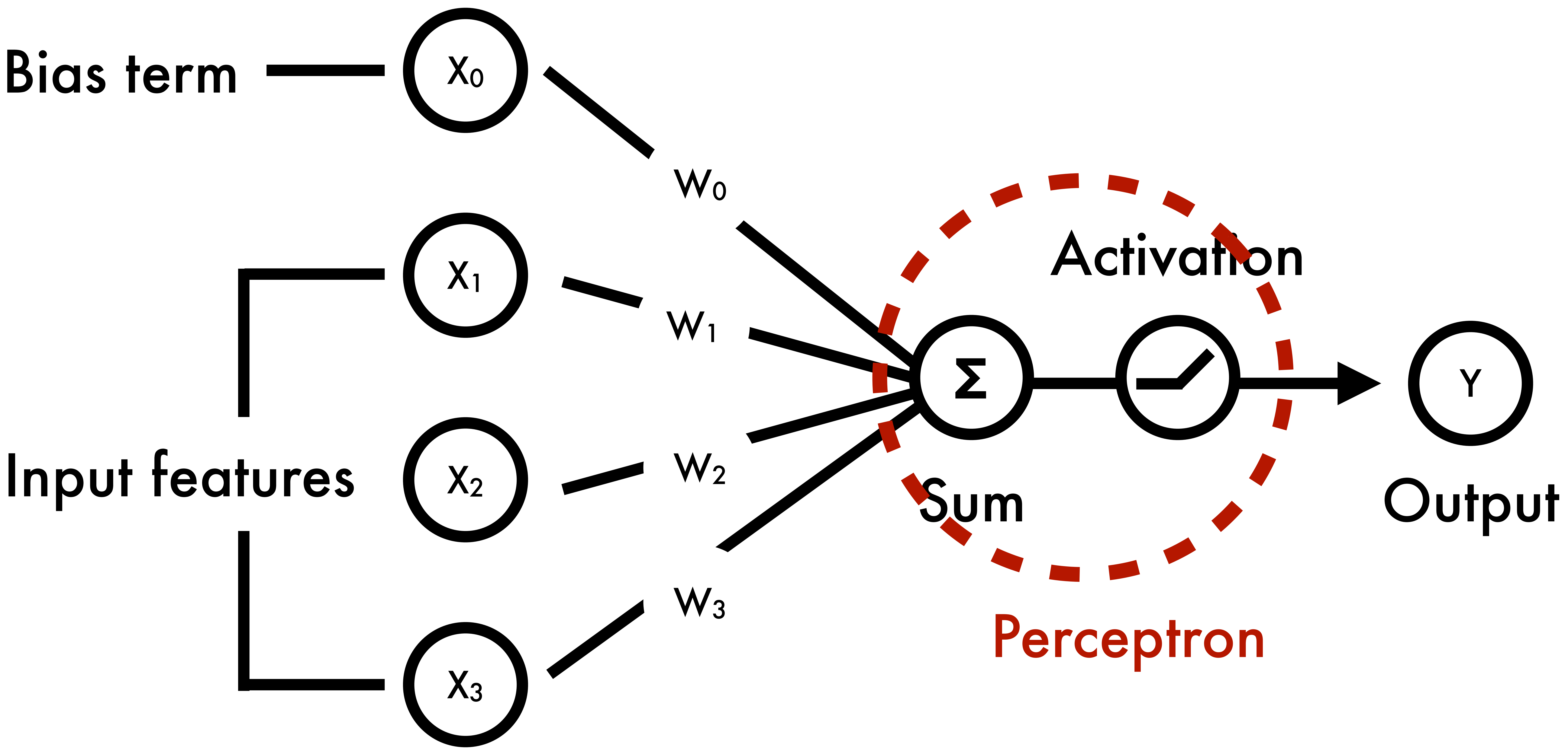
$$\frac{\partial SSE}{\partial \mathbf{w}} = -2\mathbf{X}^\top\mathbf{y} + 2\mathbf{X}^\top\mathbf{X}\mathbf{w} = 0$$

$$\mathbf{X}^\top\mathbf{X}\mathbf{w} = \mathbf{X}^\top\mathbf{y}$$

$$\mathbf{w} = (\mathbf{X}^\top\mathbf{X})^{-1}\mathbf{X}^\top\mathbf{y}$$

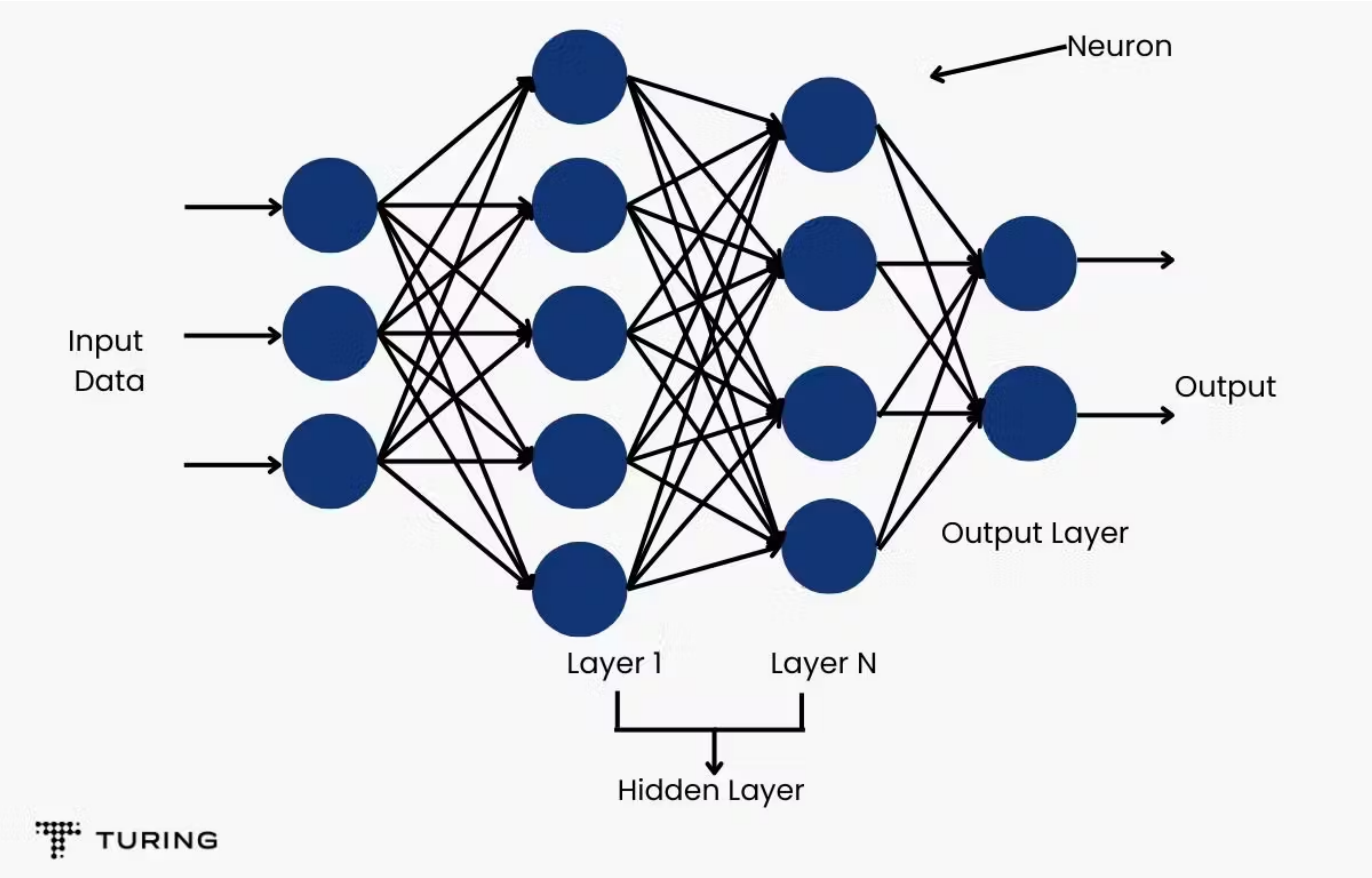
Pseudo-inverse

Perceptron


Bias term
X_0
W_0
X_1
W_1
Input features
X_2
W_2
X_3
W_3
Activation
Σ
Sum
Y
Output
Perceptron

Multi Layer Perceptron

- Also known as feed-forward network (FNN)
- **Spread** multiple perceptrons horizontally
  - -> (Hidden) layers
- and
- **Stack** multiple layers vertically
- That’s it!!!
- Works for both regression and classification
  - And much more actually…



https://www.turing.com/kb/explanation-of-deep-neural-network-multilayer-perceptron-deep-q-network

MLP ≈ Deep learning

- Understanding how MLP works means that you know (almost) everything about deep learning!!!
- By spreading and stacking hundreds of thousands of (even millions of) perceptrons,
- By feeding an enormous amount of data,
- By adopting a simple learning algorithm, e.g., back propagation,
- Deep learning has been, is, and will be achieving some amazing things
    - Regression, classification
    - Generative models
    - Deep reinforcement learning algorithms

Wait...

- Linear regression ≈ Perceptron
- Perceptron -> MLP
- MLP ≈ Deep learning

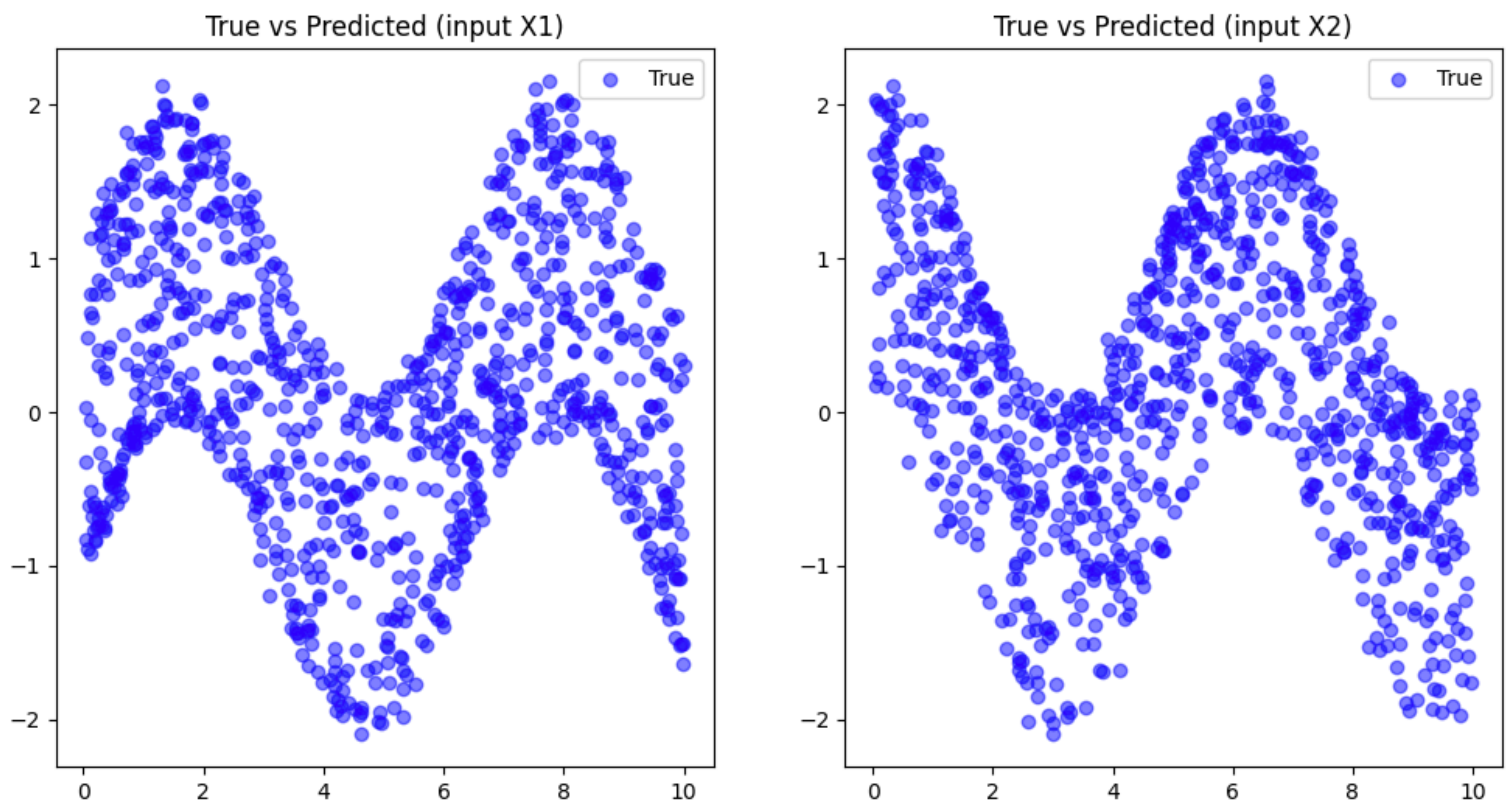
- Deep learning eventually reduces to a perceptron?!

Wait…

- Y = WX + B
- = W(WX + B) + B
- = W(W(WX + B) + B)
- = WX + B
- Thus, eventually, spreading and stacking multiple perceptrons means absolutely, nothing
- It’s still just a linear projection!

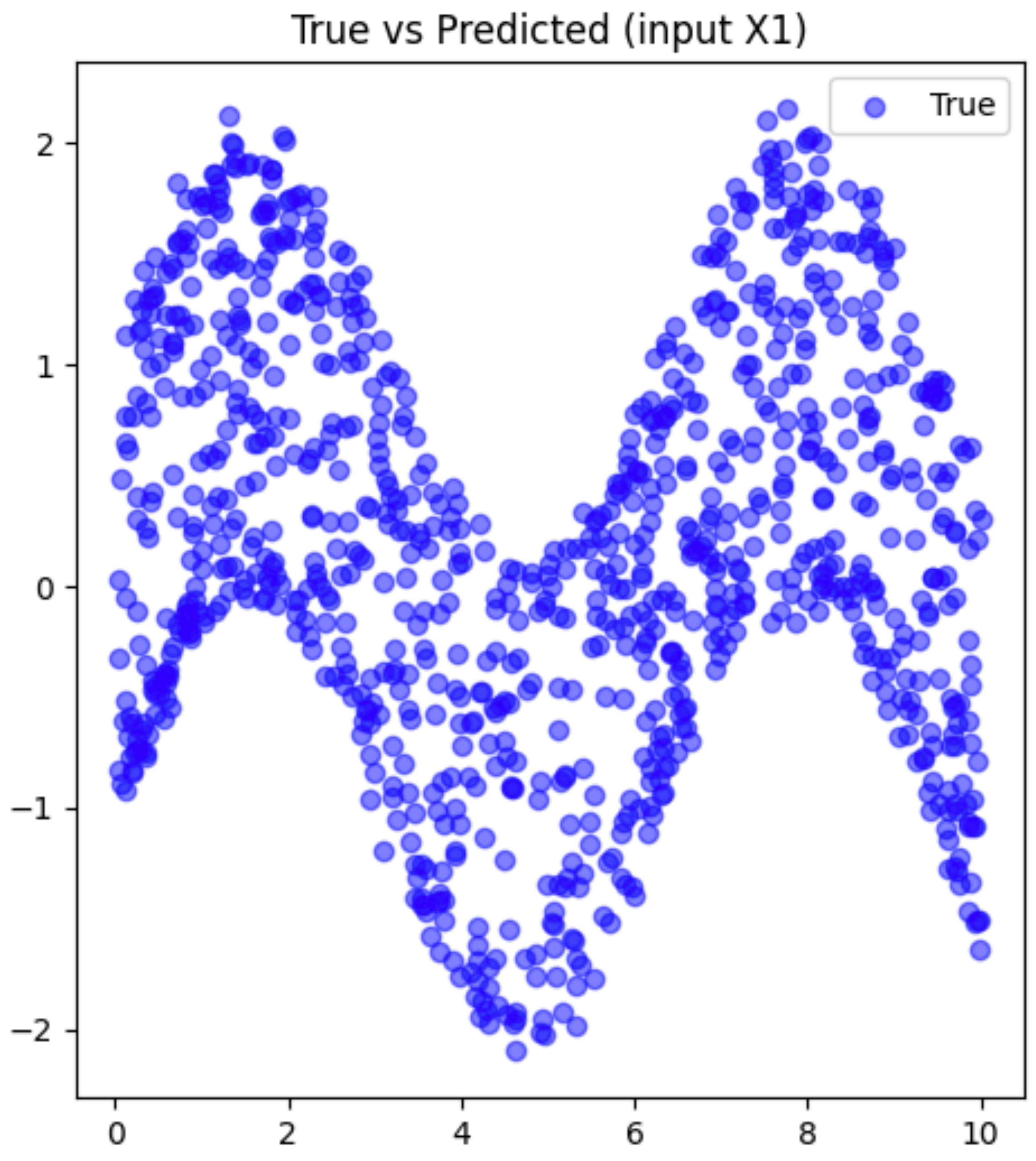
Experiment

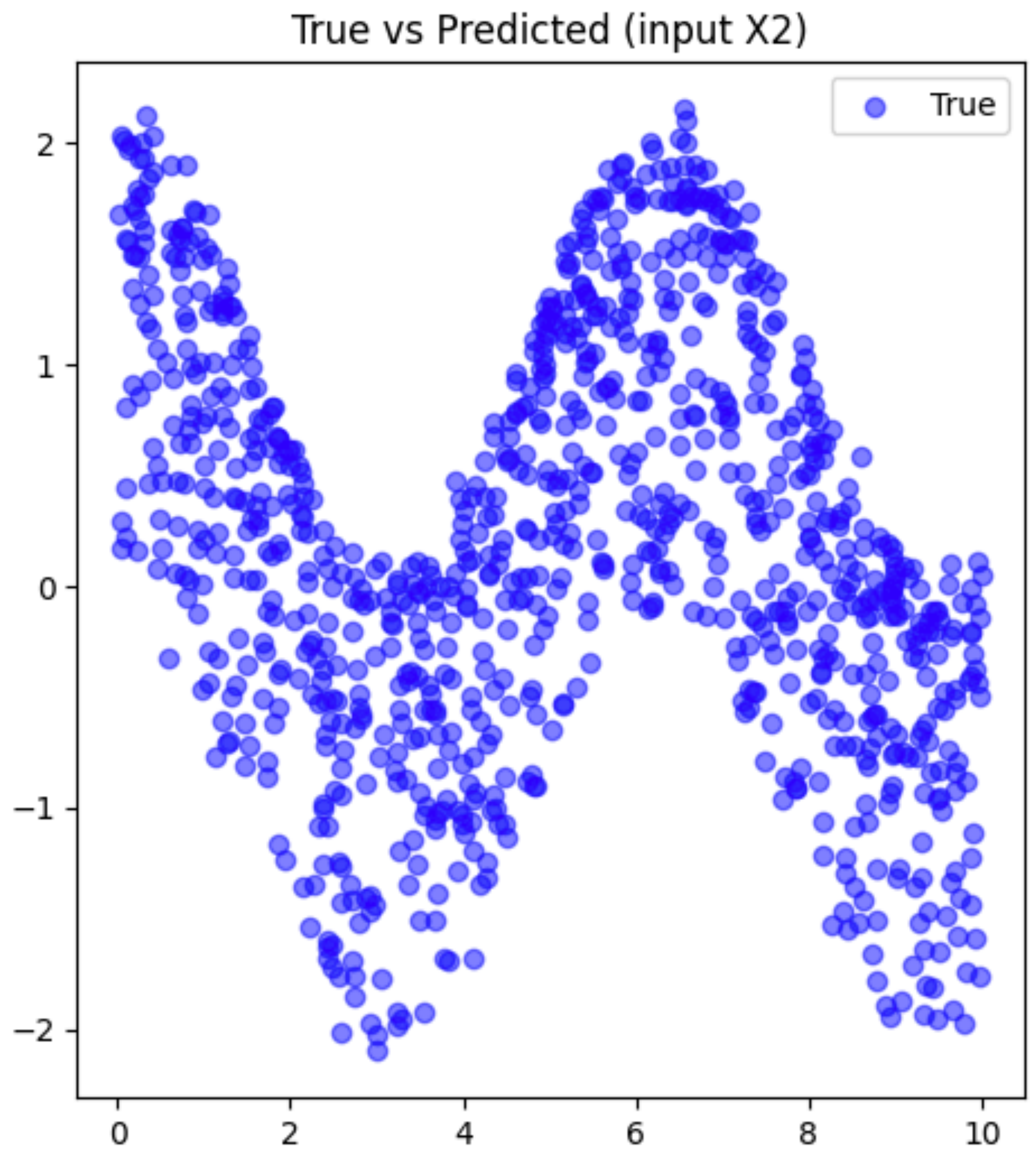
```
# Generate random inputs (X) and target outputs (Y)
X = np.random.rand(n_samples, 2) * 10  # inputs in range [0, 10]
Y = np.sin(X[:, 0]) + np.cos(X[:, 1]) + np.random.randn(n_samples) * 0.1  # target with some noise
```



Experiment

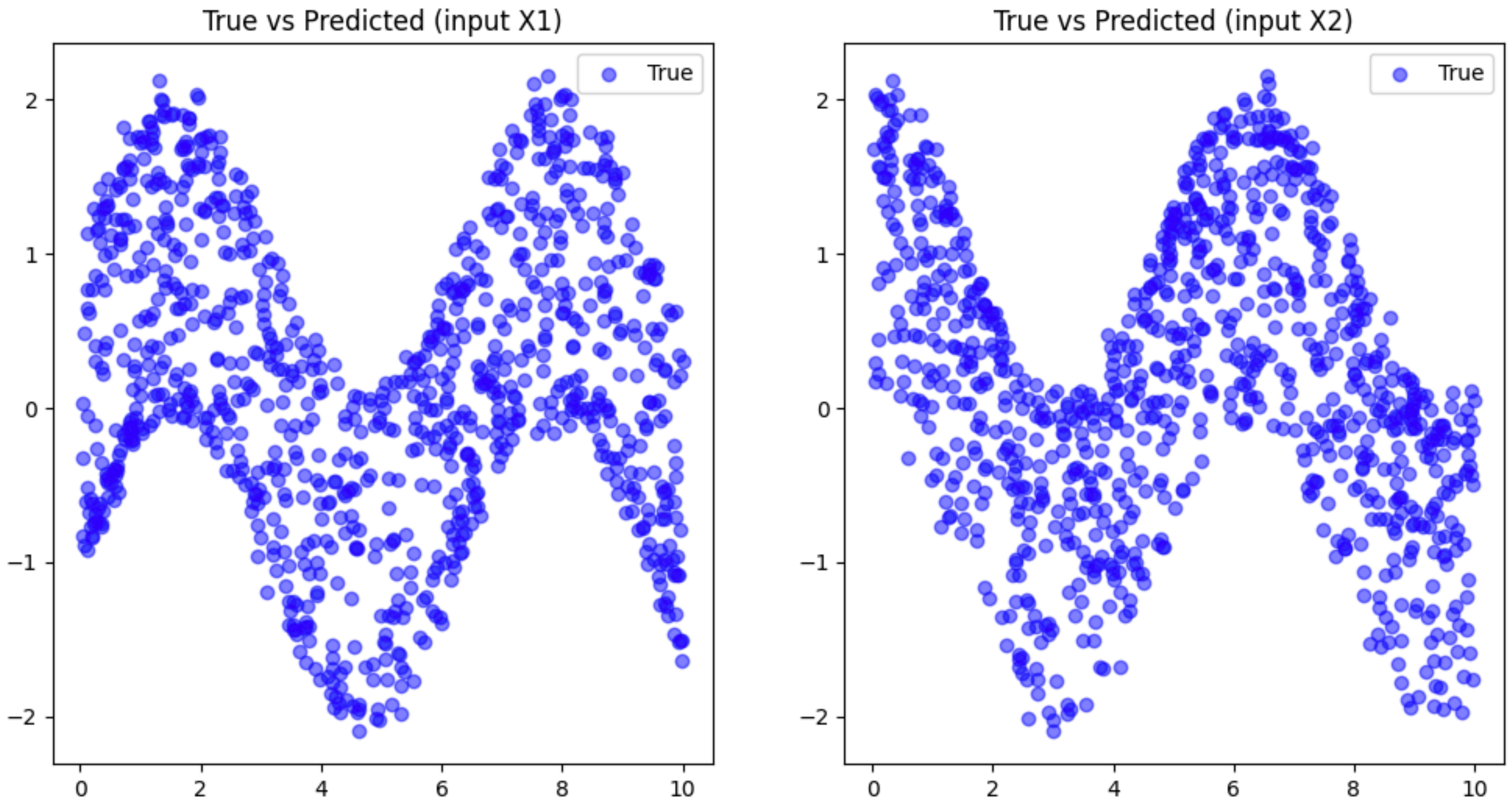
- 1000 training datapoints
- Input dimension: 2 (2 features)
- 2 hidden layers
- A little bit of noise
- 64 perceptrons for each hidden layer





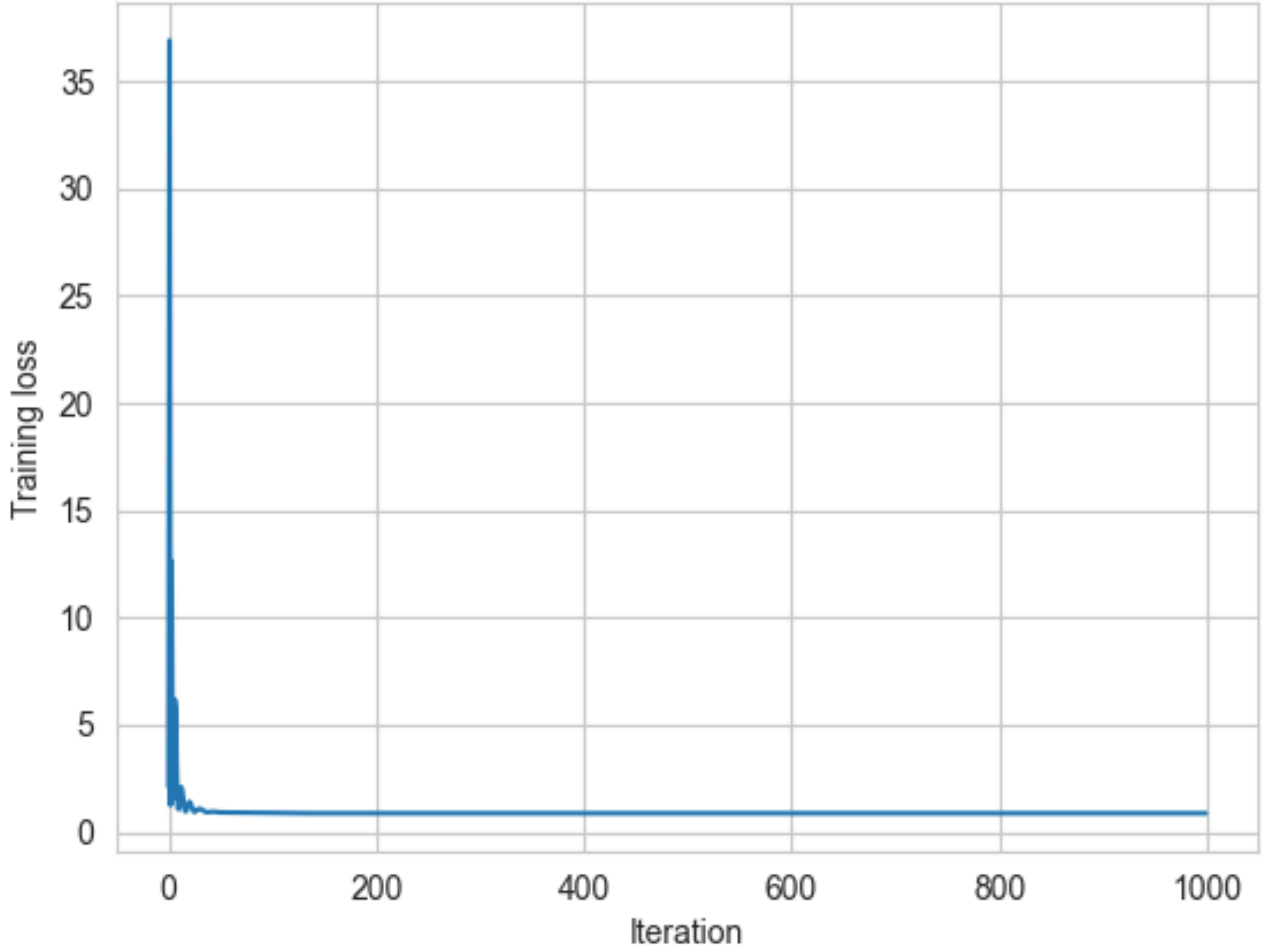
Experiment

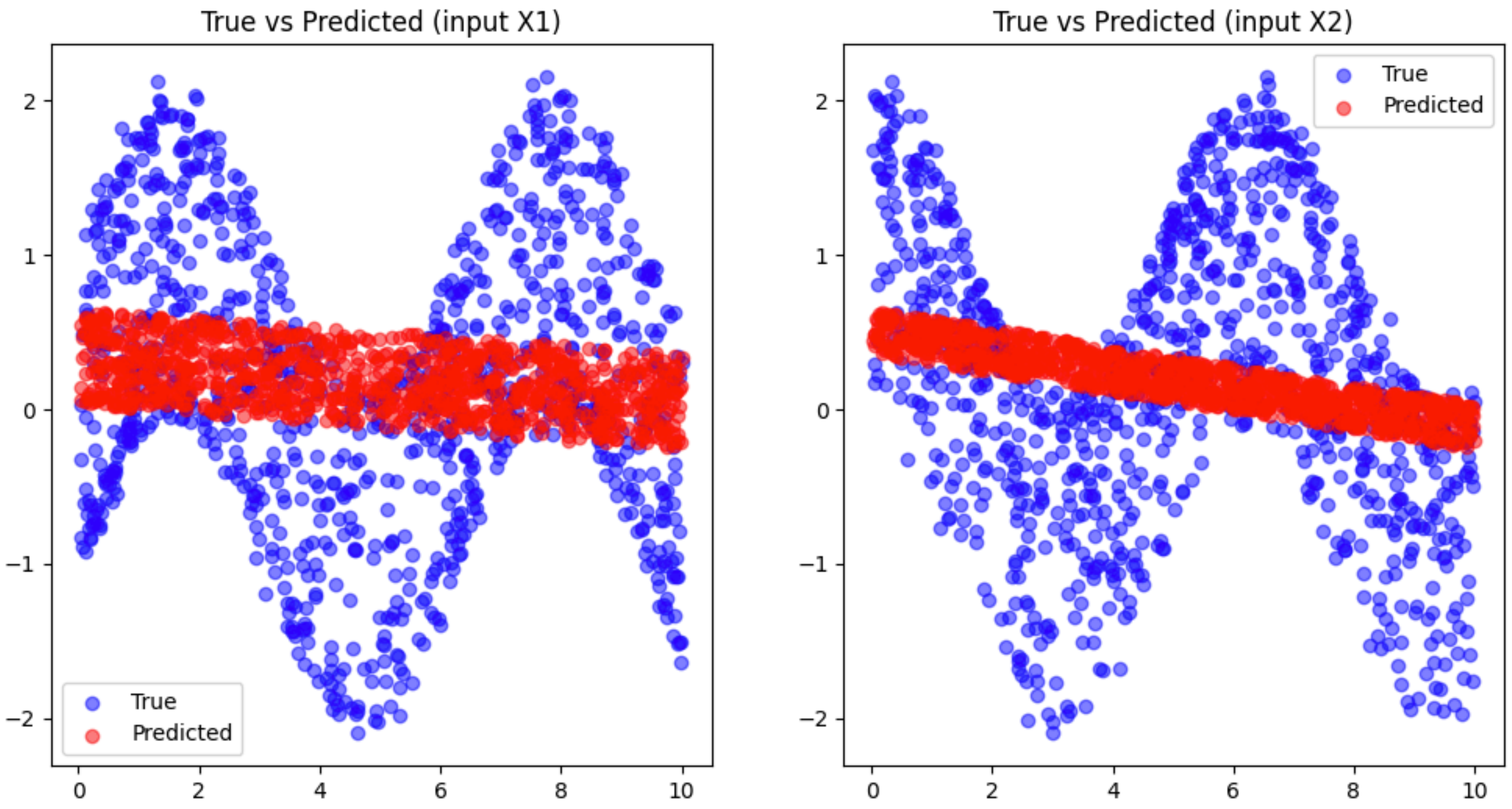
- 1000 training datapoints
- Input dimension: 2 (2 features)
- 2 hidden layers
- A little bit of noise
- 64 perceptrons for each hidden layer
- For a deep learning model of this size, fitting 1,000 2-dimensional datapoints is a piece of cake!



Experimental results

- 2-hidden layer MLP with 128 hidden perceptrons is indeed just a linear regression!

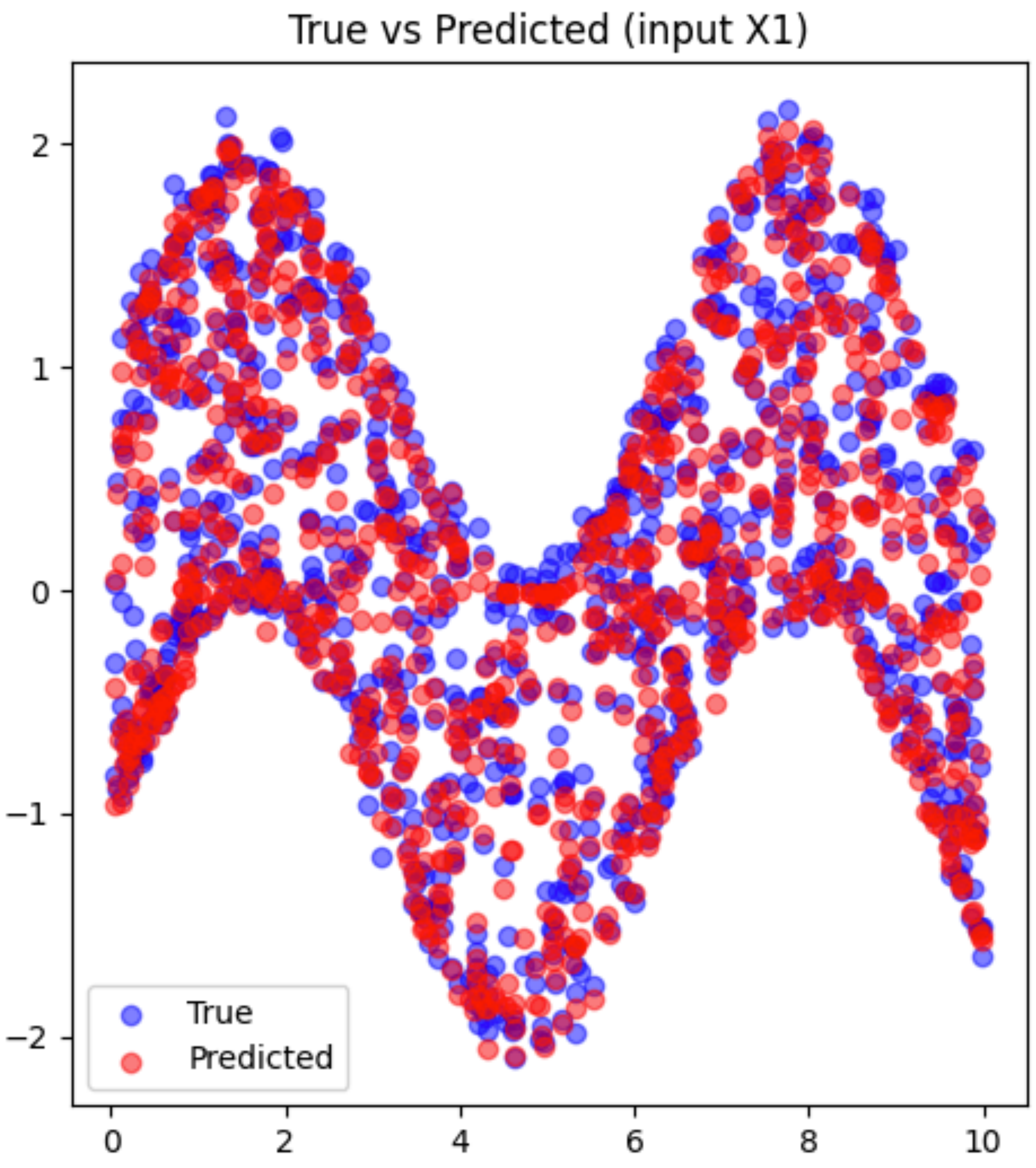


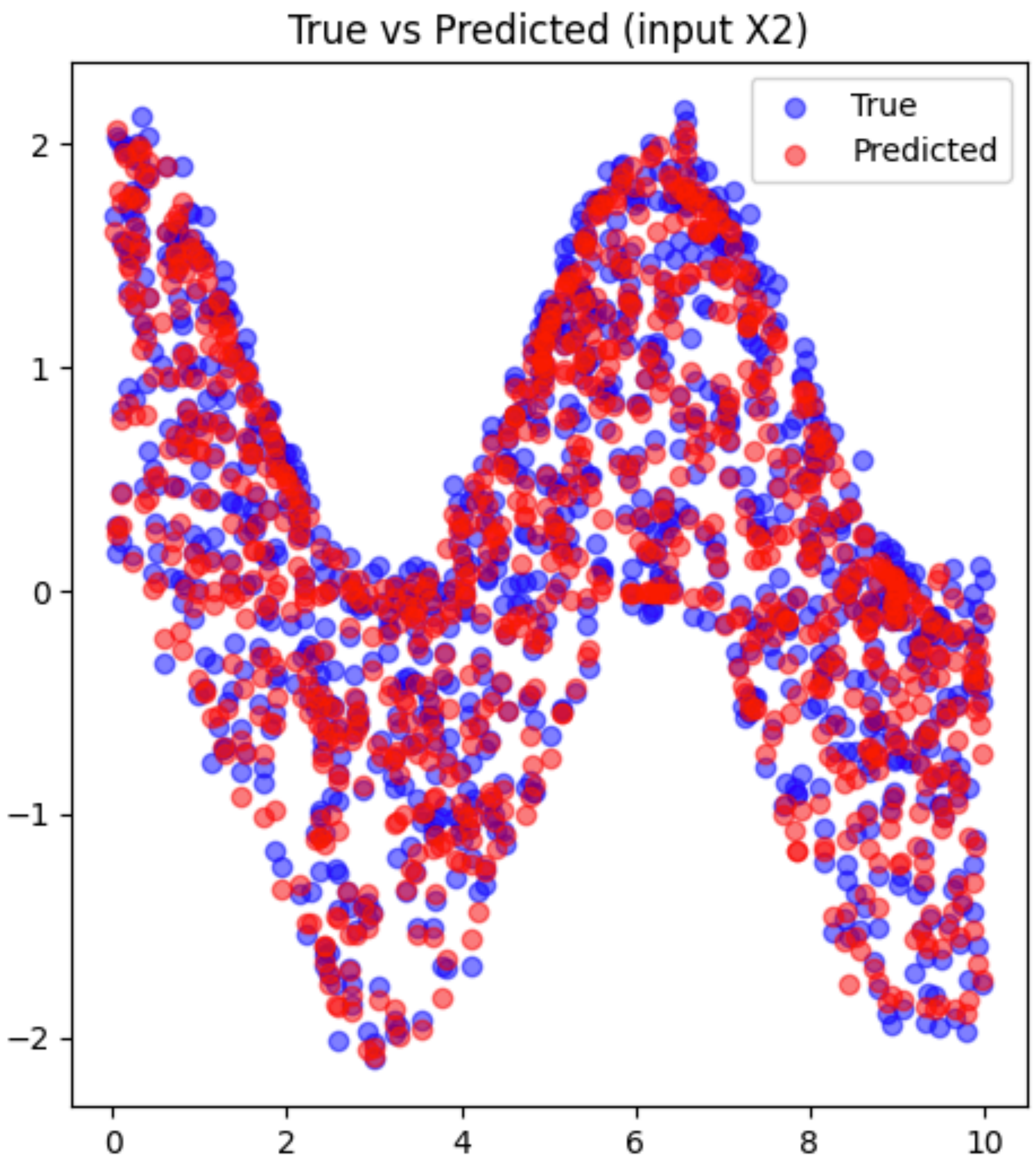


Of course, there are more concerns

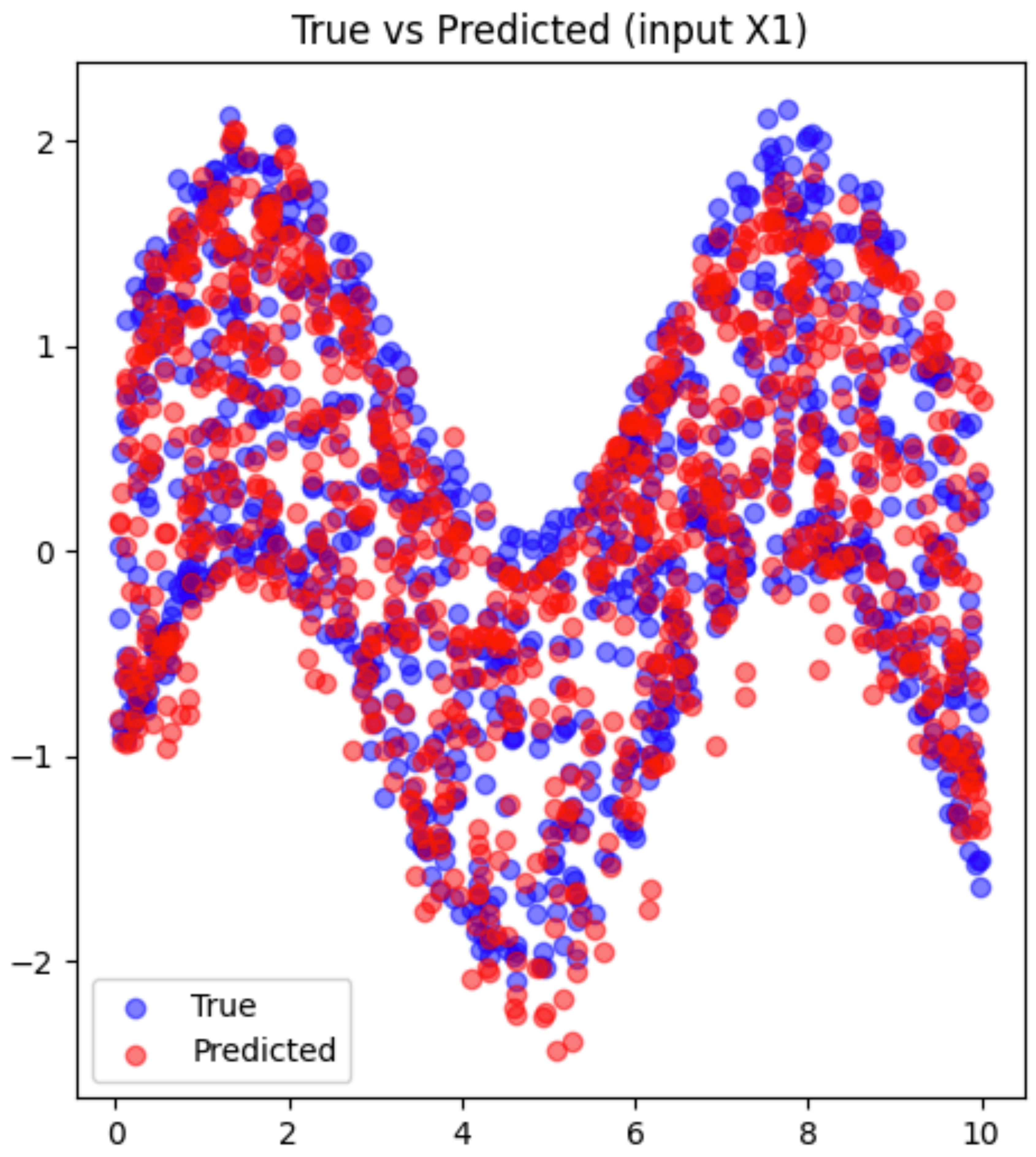
- More data
  - Fit different modalities
    - Sequential data
    - Image data
- Larger model
  - Bigger model is not always the answer
  - Overfitting? Overparameterization?
- More compute
  - Closed-form solution
  - Learning algorithm

W/ ReLU

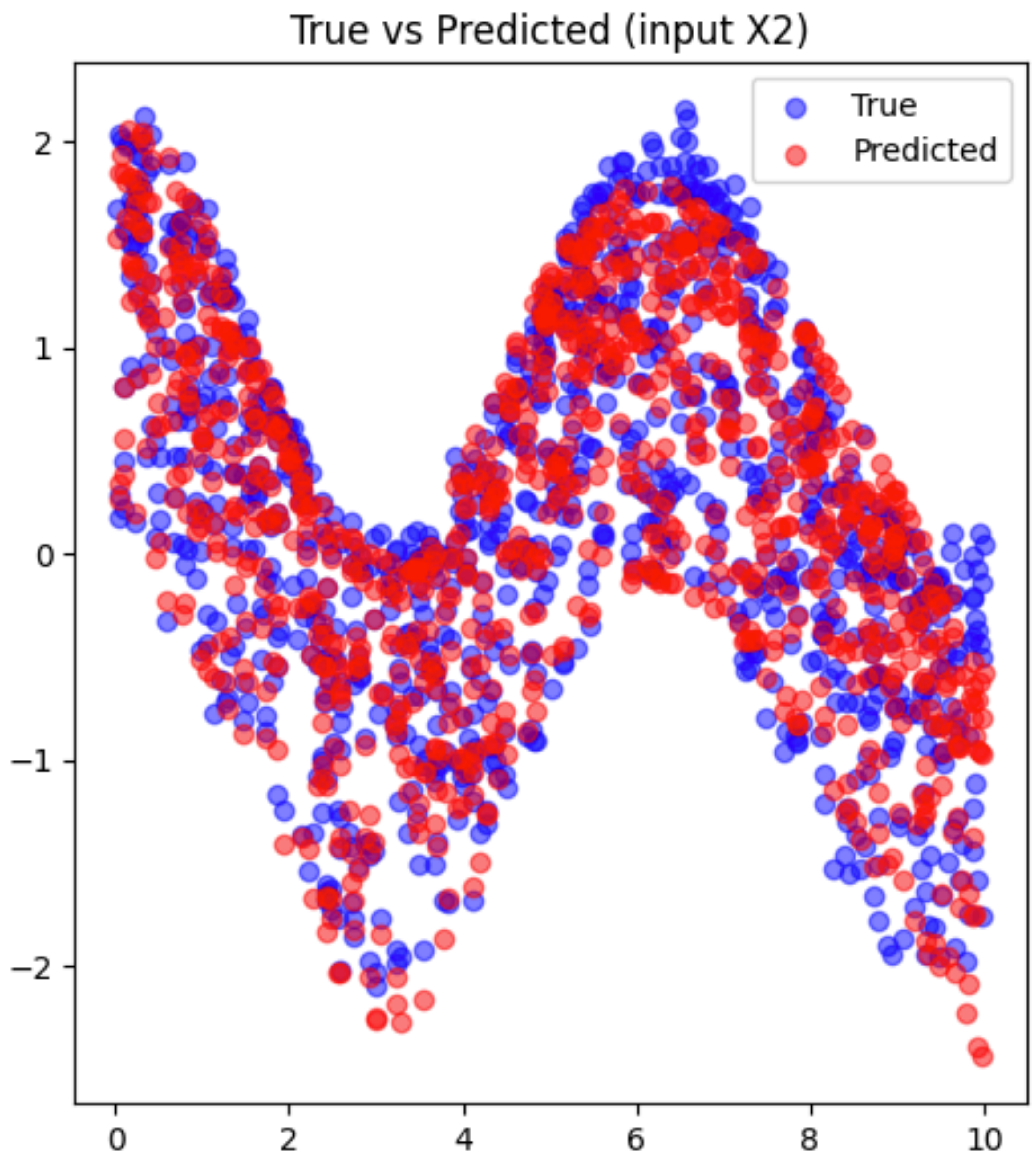

True vs Predicted (input X1)
True
Predicted
2
1
0
−1
−2
0
2
4
6
8
10


True vs Predicted (input X2)
True
Predicted
2
1
0
−1
−2
0
2
4
6
8
10

W/ Sigmoid

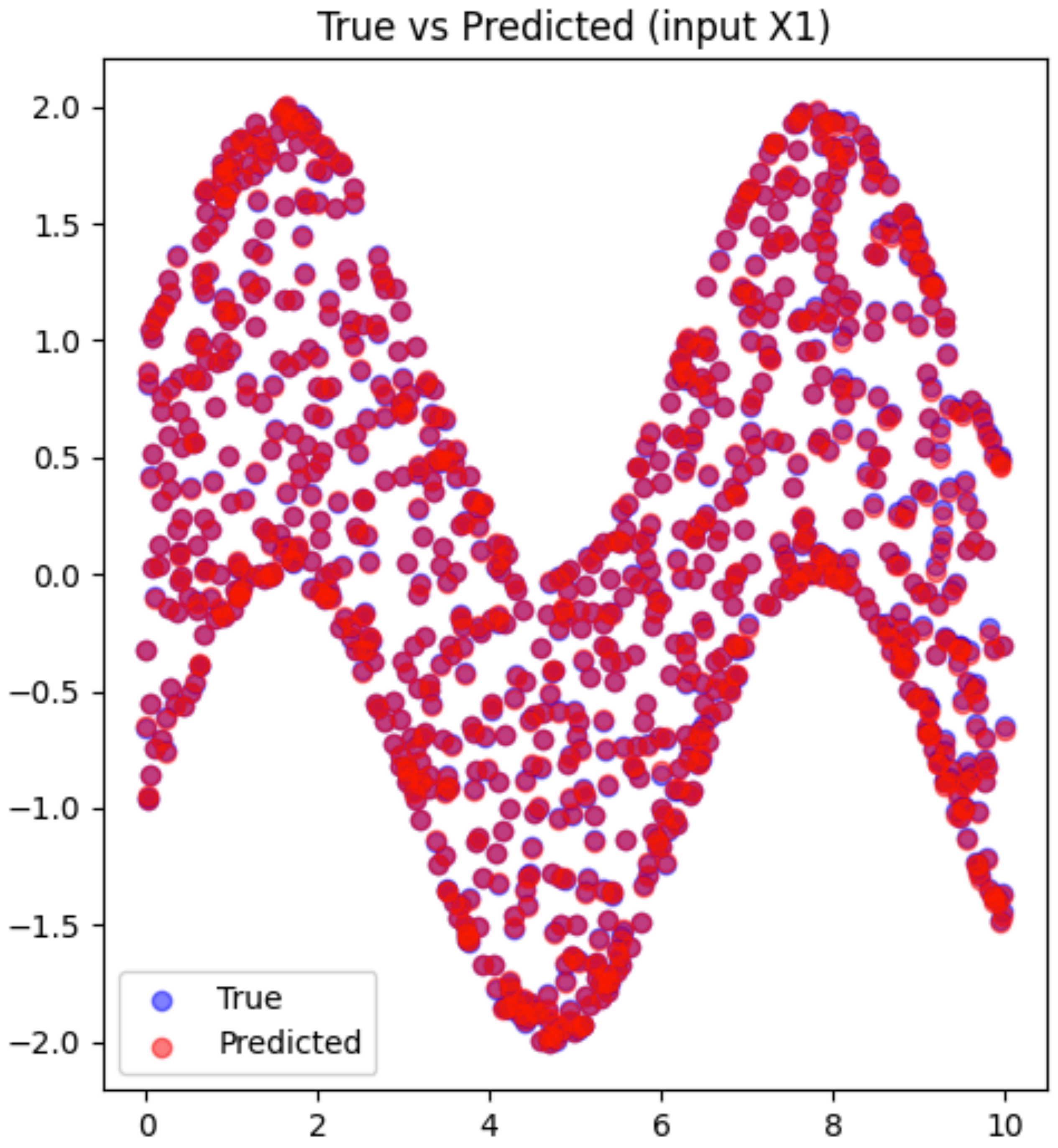


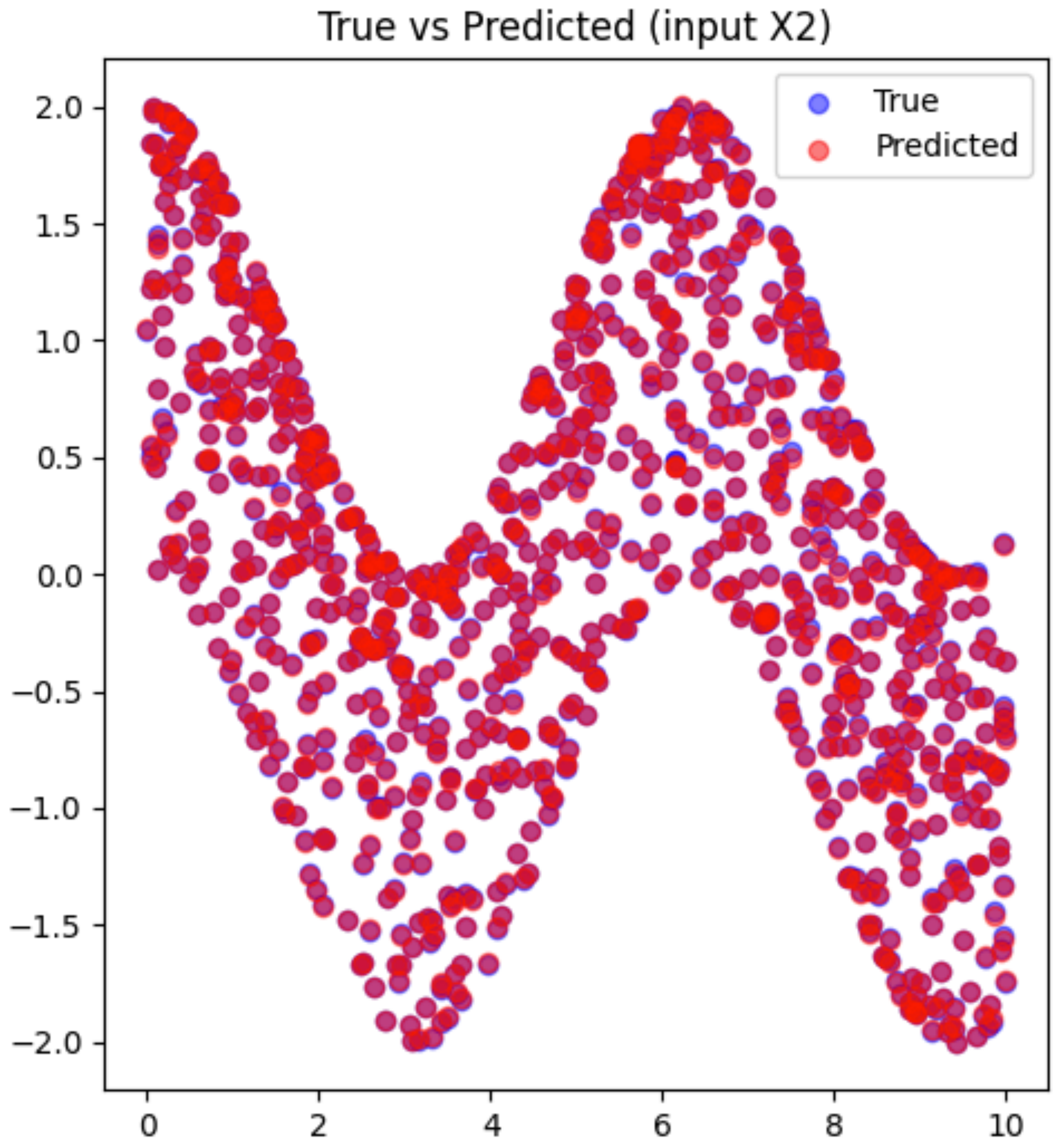
True vs Predicted (input X1)
True
Predicted
2
1
0
−1
−2
0
2
4
6
8
10



True vs Predicted (input X2)
True
Predicted
2
1
0
−1
−2
0
2
4
6
8
10

W/ ReLU, 100K datapoints, 100K iter





MNIST dataset

- Handwritten numbers from 0 to 9
  - Different writing styles
  - Some numbers are hard to be differentiated even for humans
  - Can be downloaded using PyTorch, which is a python’s machine learning library

MNIST dataset

- Handwritten numbers from 0 to 9
  - Different writing styles
  - Some numbers are hard to be differentiated even for humans
  - Can be downloaded using PyTorch, which is a python's machine learning library
-

```
import torch
from torchvision import datasets, transforms
from torchvision.utils import save_image

train_kwargs = {'batch_size': 64}

transform=transforms.Compose([
    transforms.ToTensor(),
    # transforms.Normalize((0.1307,), (0.3081,))
    ])

train_dataset = datasets.MNIST('mnist_data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST('mnist_data', train=False, transform=transform)

print(train_dataset)
print(test_dataset)

train_loader = torch.utils.data.DataLoader(train_dataset, **train_kwargs)
test_loader = torch.utils.data.DataLoader(test_dataset, **train_kwargs)

data = next(iter(train_loader))

print(data[0].shape)
print(data[1], data[1].shape)

save_image(data[0], 'mnist_samples.png')
```

MNIST dataset

- Handwritten numbers from 0 to 9
  - Different writing styles
  - Some numbers are hard to be differentiated even for humans
  - Can be downloaded using PyTorch, which is a python's machine learning library
- 60,000 train set; 10,000 test set
  - 6,000, 1,000 train, test images
  - For each class

```python
import torch
from torchvision import datasets, transforms
from torchvision.utils import save_image

train_kwargs = {'batch_size': 64}

transform=transforms.Compose([
    transforms.ToTensor(),
    # transforms.Normalize((0.1307,), (0.3081,))
    ])

train_dataset = datasets.MNIST('mnist_data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST('mnist_data', train=False, transform=transform)

print(train_dataset)
print(test_dataset)
```

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: mnist_data
    Split: Train
    StandardTransform
Transform: Compose(
               ToTensor()
           )
Dataset MNIST
    Number of datapoints: 10000
    Root location: mnist_data
    Split: Test
    StandardTransform
Transform: Compose(
               ToTensor()
           )
```

MNIST dataset

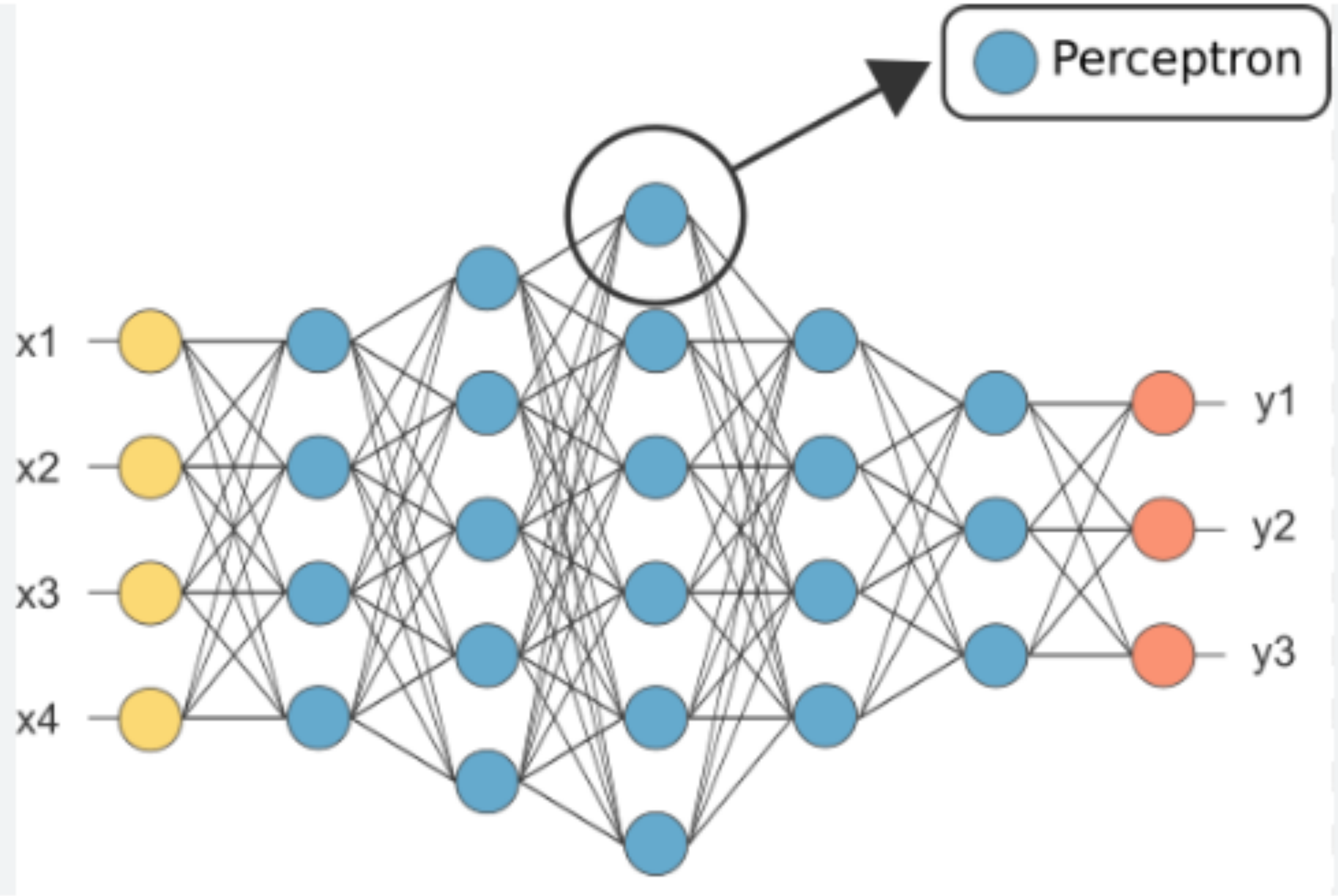
- Handwritten numbers from 0 to 9
  - Different writing styles
  - Some numbers are hard to be differentiated even for humans
  - Can be downloaded using PyTorch, which is a python's machine learning library
- 60,000 train set; 10,000 test set
  - 6,000, 1,000 train, test images
  - For each class
- Each is 28 * 28 black and white image

```python
import torch
from torchvision import datasets, transforms
from torchvision.utils import save_image

train_kwargs = {'batch_size': 64}

transform=transforms.Compose([
    transforms.ToTensor(),
    # transforms.Normalize((0.1307,), (0.3081,))
    ])

train_dataset = datasets.MNIST('mnist_data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST('mnist_data', train=False, transform=transform)

print(train_dataset)
print(test_dataset)

train_loader = torch.utils.data.DataLoader(train_dataset, **train_kwargs)
test_loader = torch.utils.data.DataLoader(test_dataset, **train_kwargs)

data = next(iter(train_loader))

print(data[0].shape)
print(data[1], data[1].shape)
```

```
torch.Size([64, 1, 28, 28])
tensor([5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5, 3, 6, 1, 7, 2, 8, 6, 9, 4, 0, 9, 1,
        1, 2, 4, 3, 2, 7, 3, 8, 6, 9, 0, 5, 6, 0, 7, 6, 1, 8, 7, 9, 3, 9, 8, 5,
        9, 3, 3, 0, 7, 4, 9, 8, 0, 9, 4, 1, 4, 4, 6, 0]) torch.Size([64])
```

The simplest deep learning structure

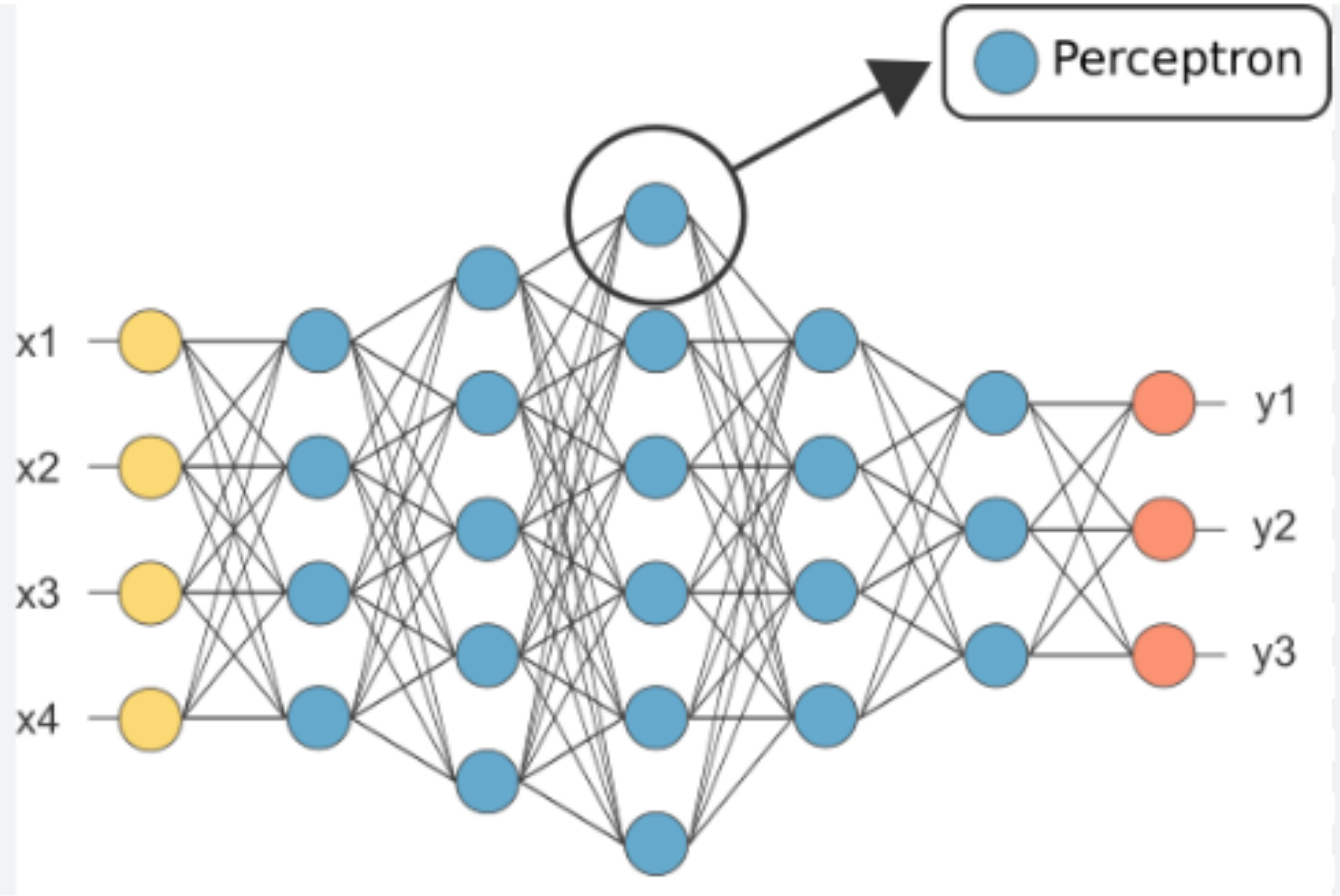
- Feedforward neural network (FNN)
- 3 types of layers:
  - Input (Yellow), hidden (Blue), output (Red)
  - Each layer consists of multiple perceptrons (neurons)
  - Layers-layers are connected with weights
  - Fully-connected:
    - All perceptrons from n th to n+1 th layer are all (fully) connected
- "Learning" is a process of which the weights of the neural networks are being optimized



https://medium.com/@b.terryjack/introduction-to-deep-learning-feed-forward-neural-networks-ffnns-a-k-a-c688d83a309d

The simplest deep learning structure

- Feedforward neural network (FNN)
- 3 types of layers
- "Learning" is a process of which the weights of the neural networks are being optimized
  - All weights and the output values of the perceptrons are scalar values
  - First, only the input perceptrons have legit values
    - Weights are randomly initialized
  - Using the input values as well as all the other (randomized) weights, you compute the output value of the upper-layer perceptrons



https://medium.com/@b.terryjack/introduction-to-deep-learning-feed-forward-neural-networks-ffnns-a-k-a-c688d83a309d