Lecture #7: Intro to neural networks

Slides by Artem Dembitskiy

Goals/Agenda

- From logistic regression to neural networks (NNs)
- Why use NNs?
- How to train NNs?
- Intuition behind backpropagation
- NNs for materials science

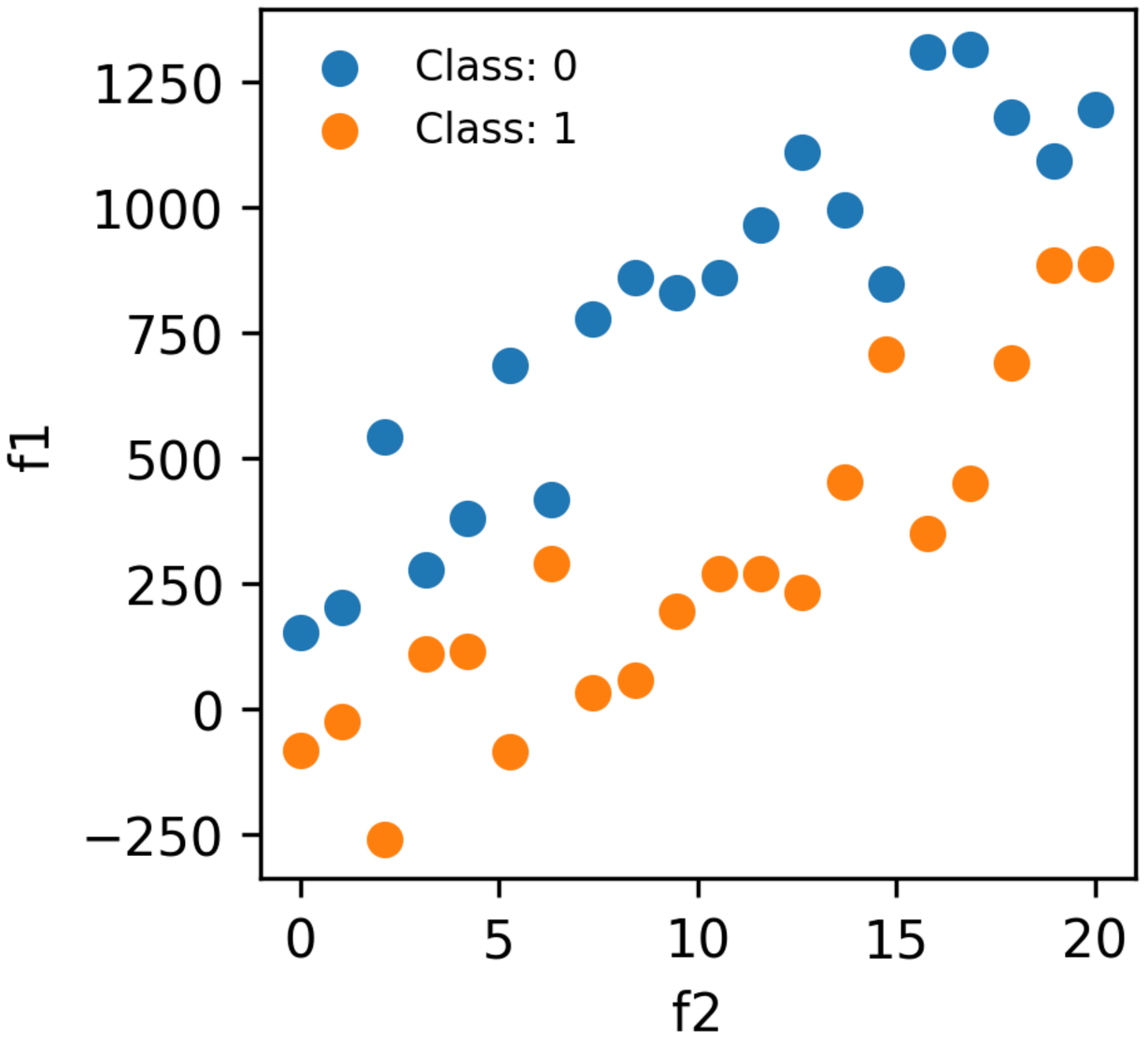
Part #1: Logistic regression

Data samples:

$\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_i, y_i), \ldots, (\boldsymbol{x}_n, y_n)\}$

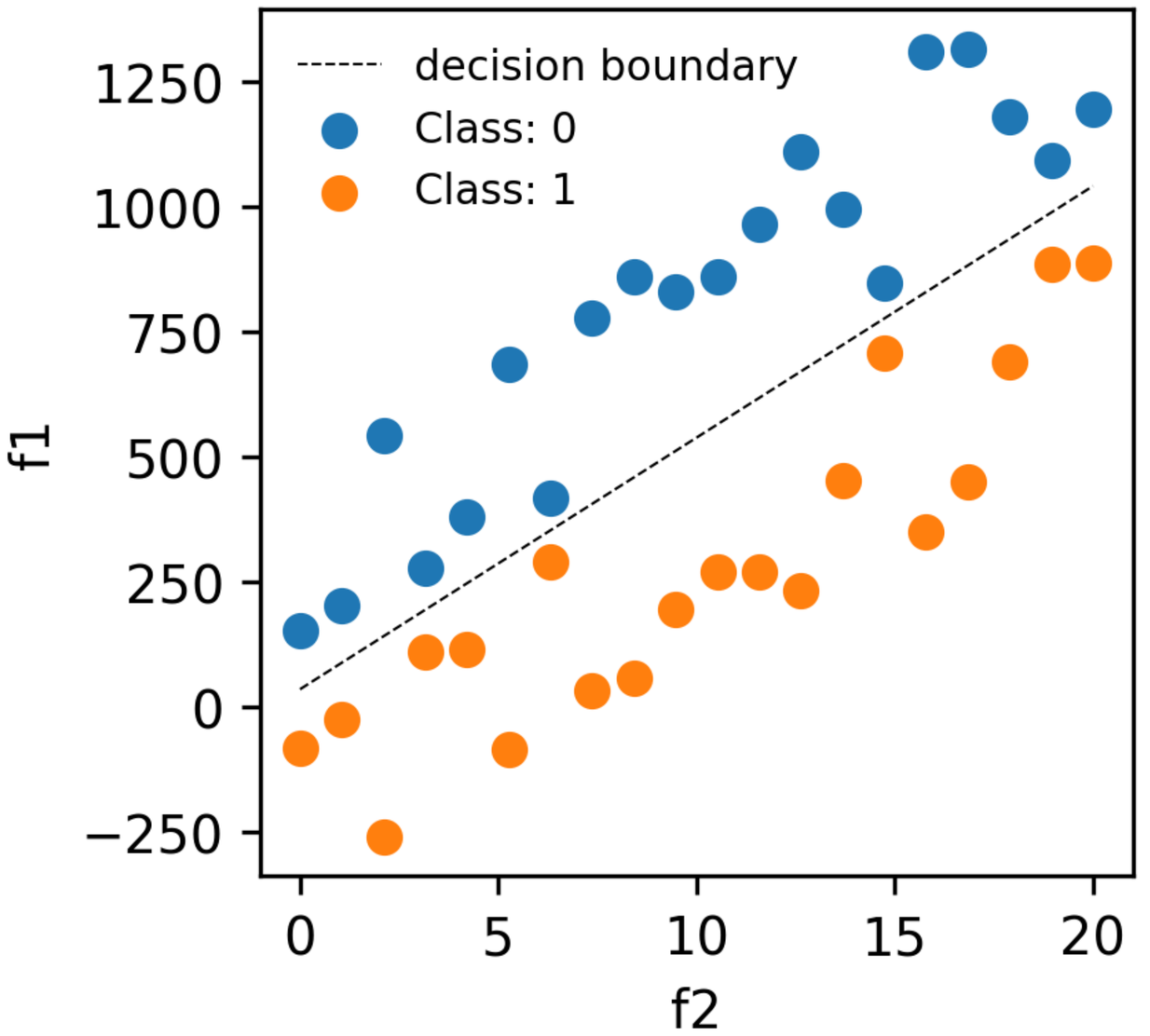
y_i -class of a sample (0 or 1)

$\boldsymbol{x}_i = (x_i^1, \ldots, x_i^d)$ - $1 \; x \; d$ vector of input features



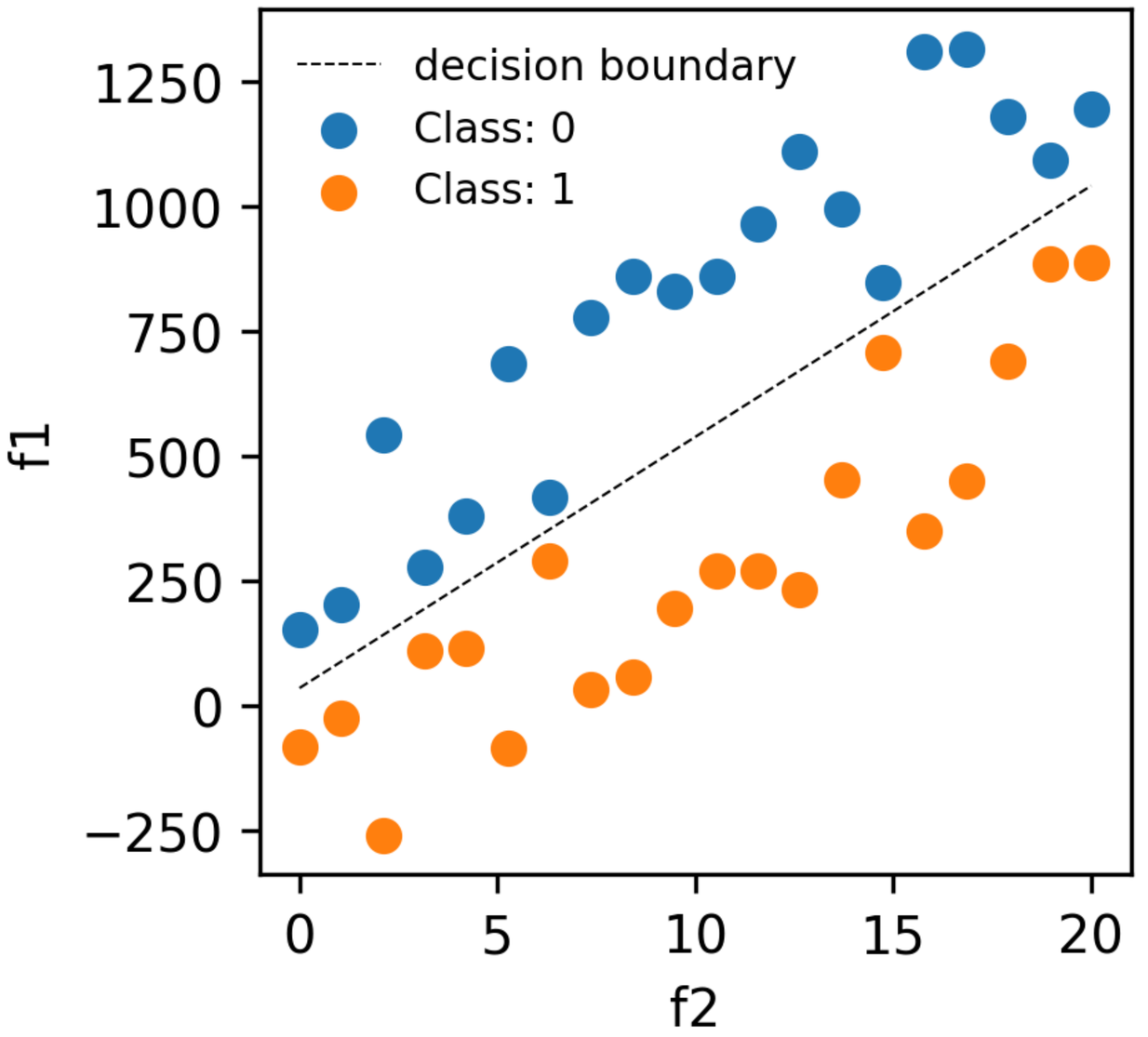
We can draw a line that separates these two groups

Classes are linearly separable



This line is called a decision boundary

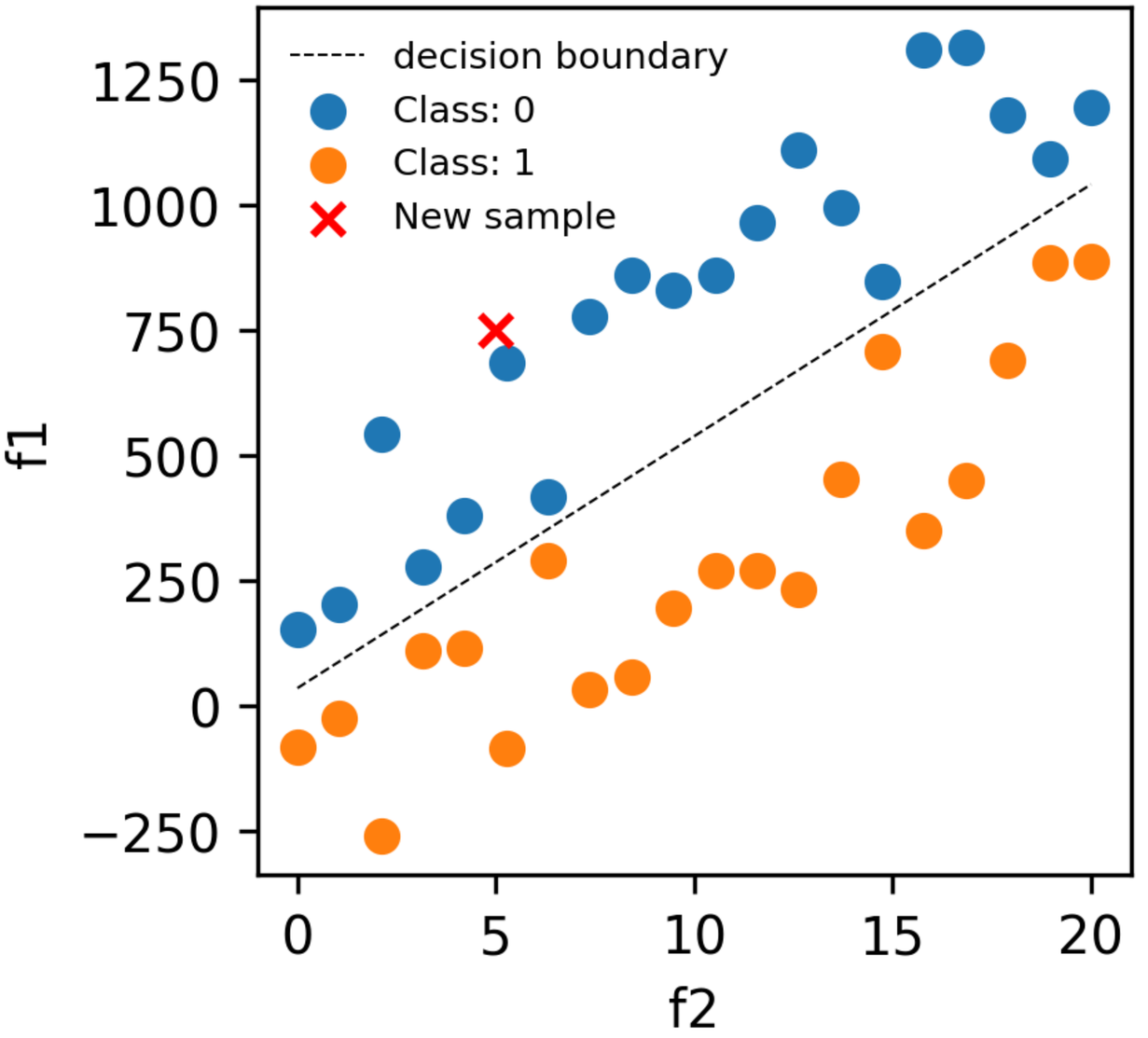
$$w_1 \cdot x_1 + w_2 \cdot x_2 + b = 0$$



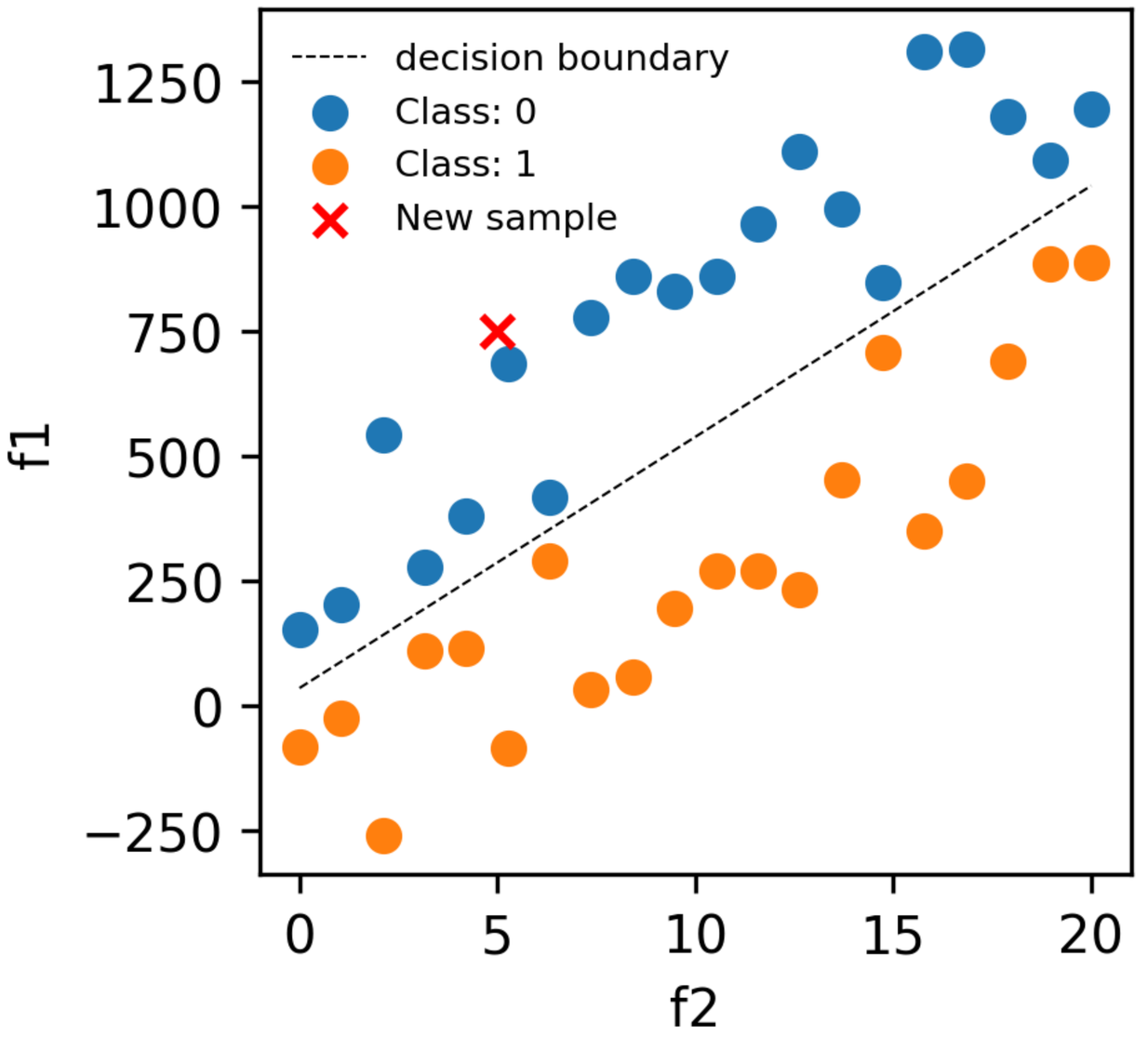
Let z be the score metric that helps answer the question "does this new sample belong to class 1?"

$$z = w_1 \cdot x_1 + w_2 \cdot x_2 + b$$

Note: $z = 0$ is a decision boundary



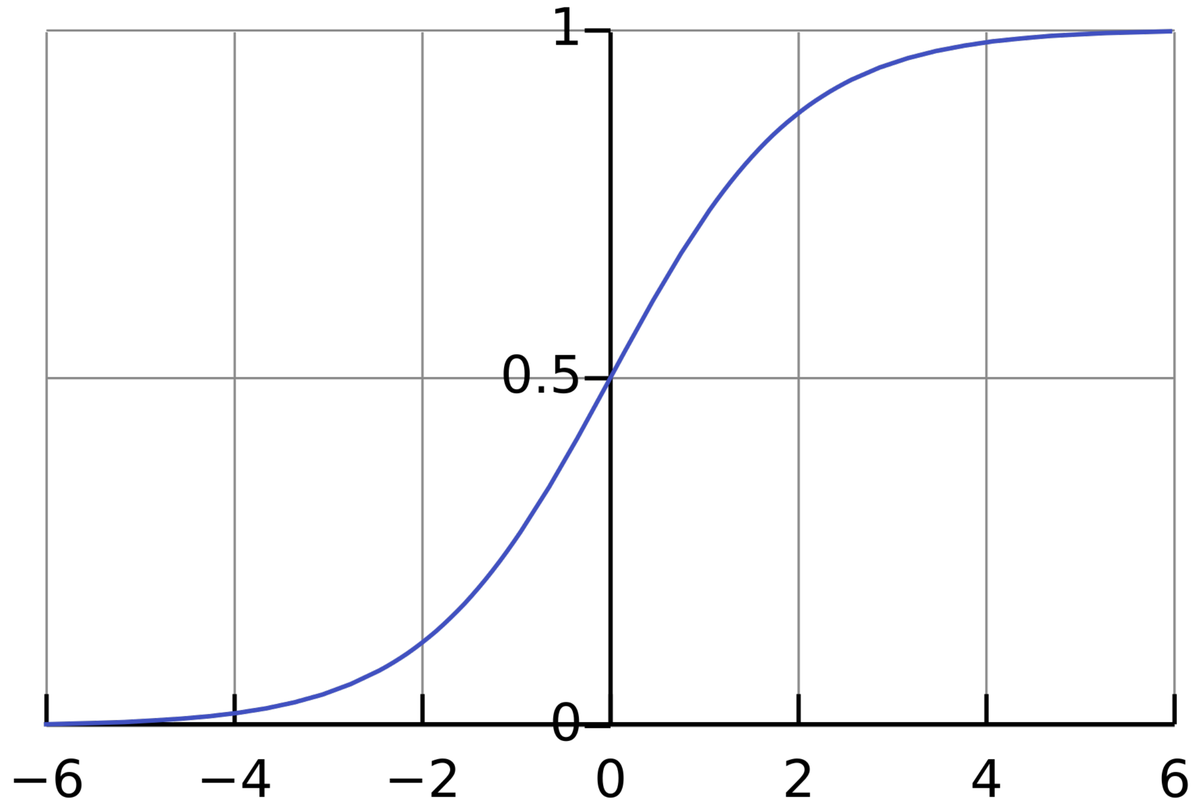
- In principle, z can take any value between $-\infty$ and ∞
- We can map this value to the range of [0, 1] using some non-linear function
- And interpret the result as a probability



Logistic function

$$f(z) = \frac{1}{1 + \exp(-z)}, \text{ where } z = \mathbf{x}\mathbf{W} + b$$

- Tells us how likely the point is to be in Class 1
- i.e. the probability



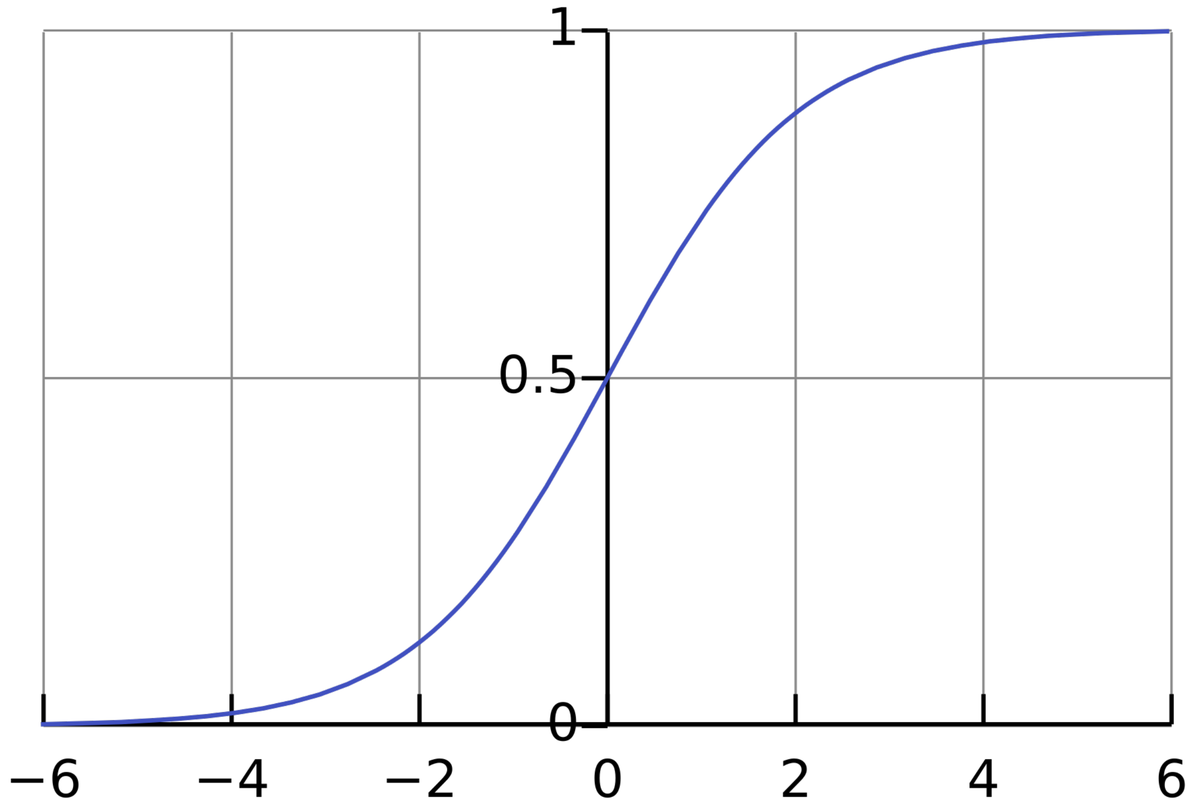
Logistic function

$$f(z) = \frac{1}{1 + \exp(-z)}, \text{ where } z = \mathbf{x}\mathbf{W} + b$$

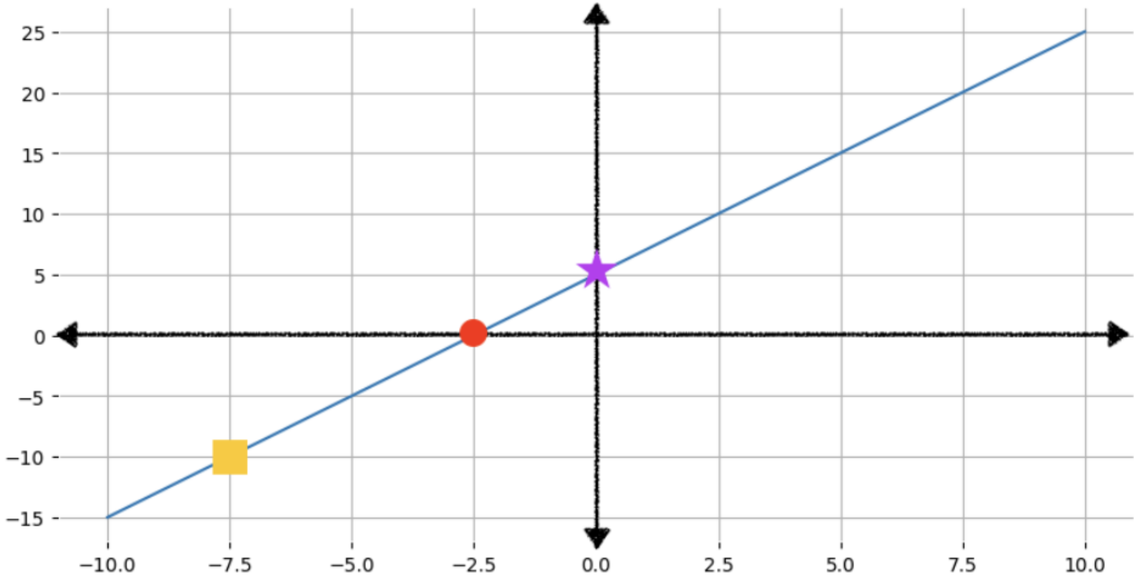
Post-processing step -> get class prediction

$$t(f(z)) = \begin{cases} 1 & \text{if } f(z) \geq 0.5 \\ 0 & \text{if } f(z) < 0.5 \end{cases}$$

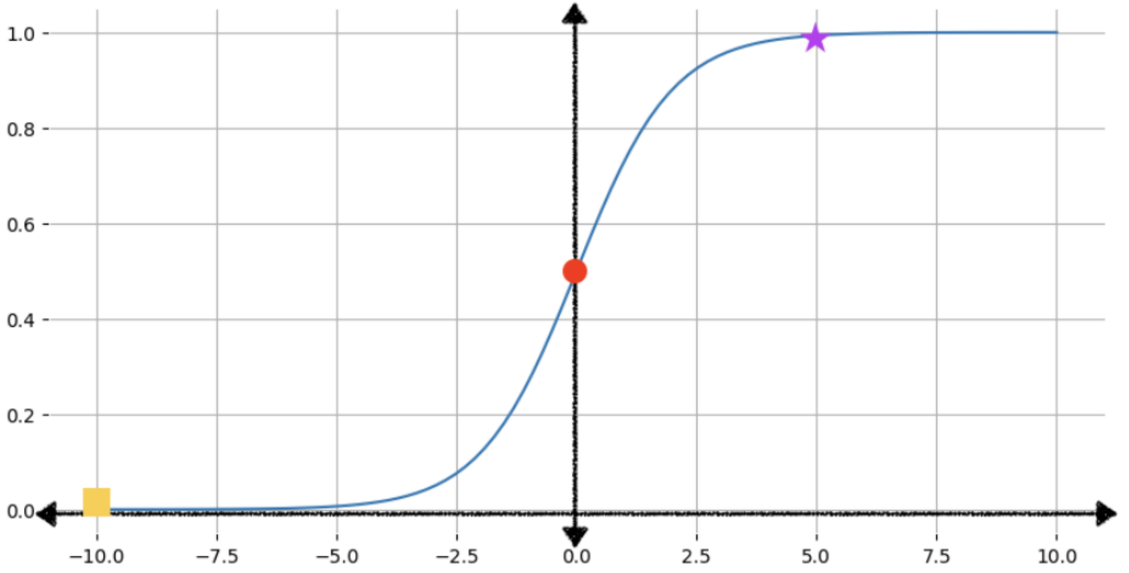
(You can use other than 0.5 threshold)



$z = 2x + 5$


25
20
15
10
5
0
−5
−10
−15
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

$y' = 1 / (1 + e^{-z})$


1.0
0.8
0.6
0.4
0.2
0.0
−10.0
−7.5
−5.0
−2.5
0.0
2.5
5.0
7.5
10.0

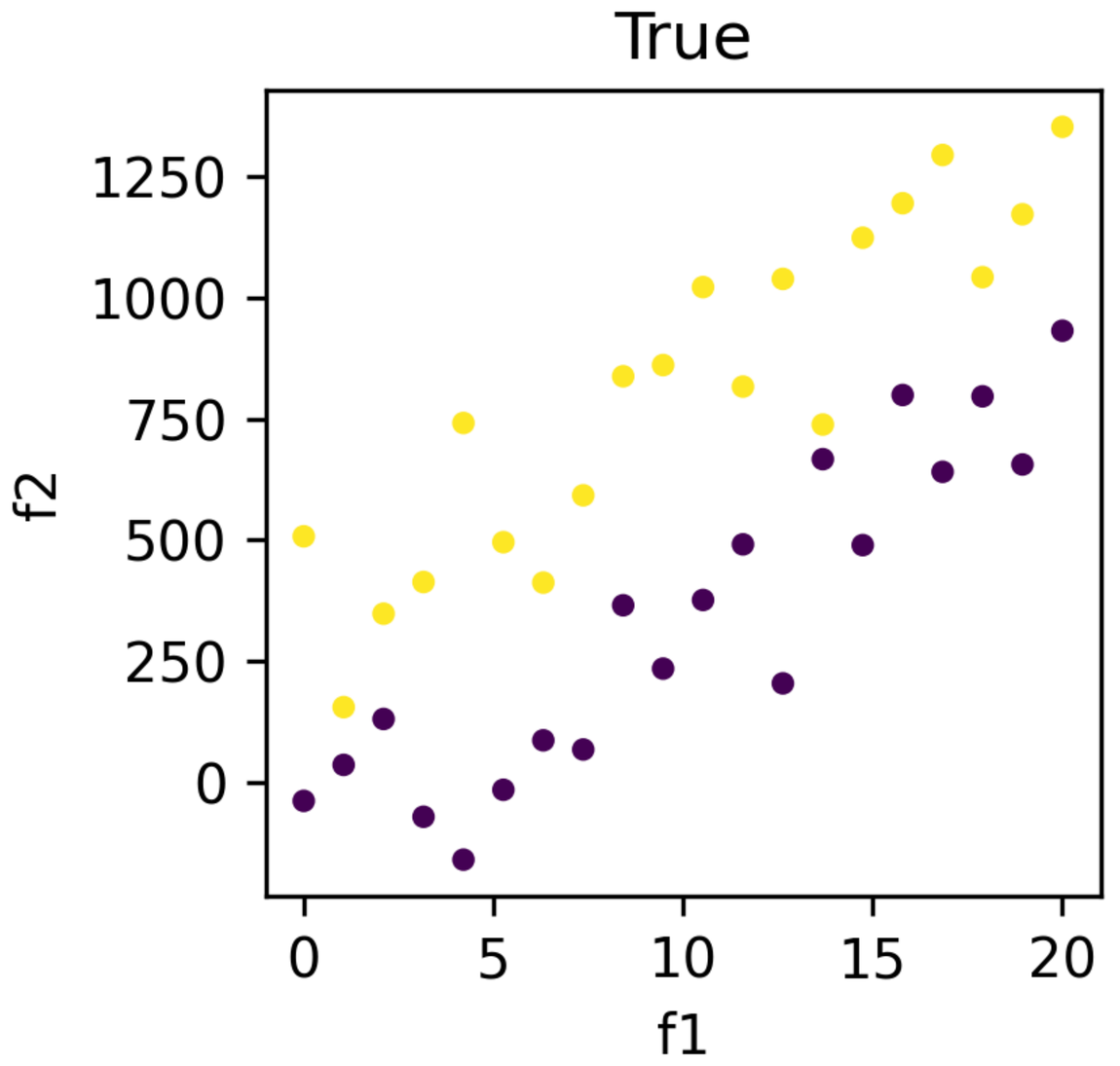
Loss: Cross-entropy loss (Log loss)

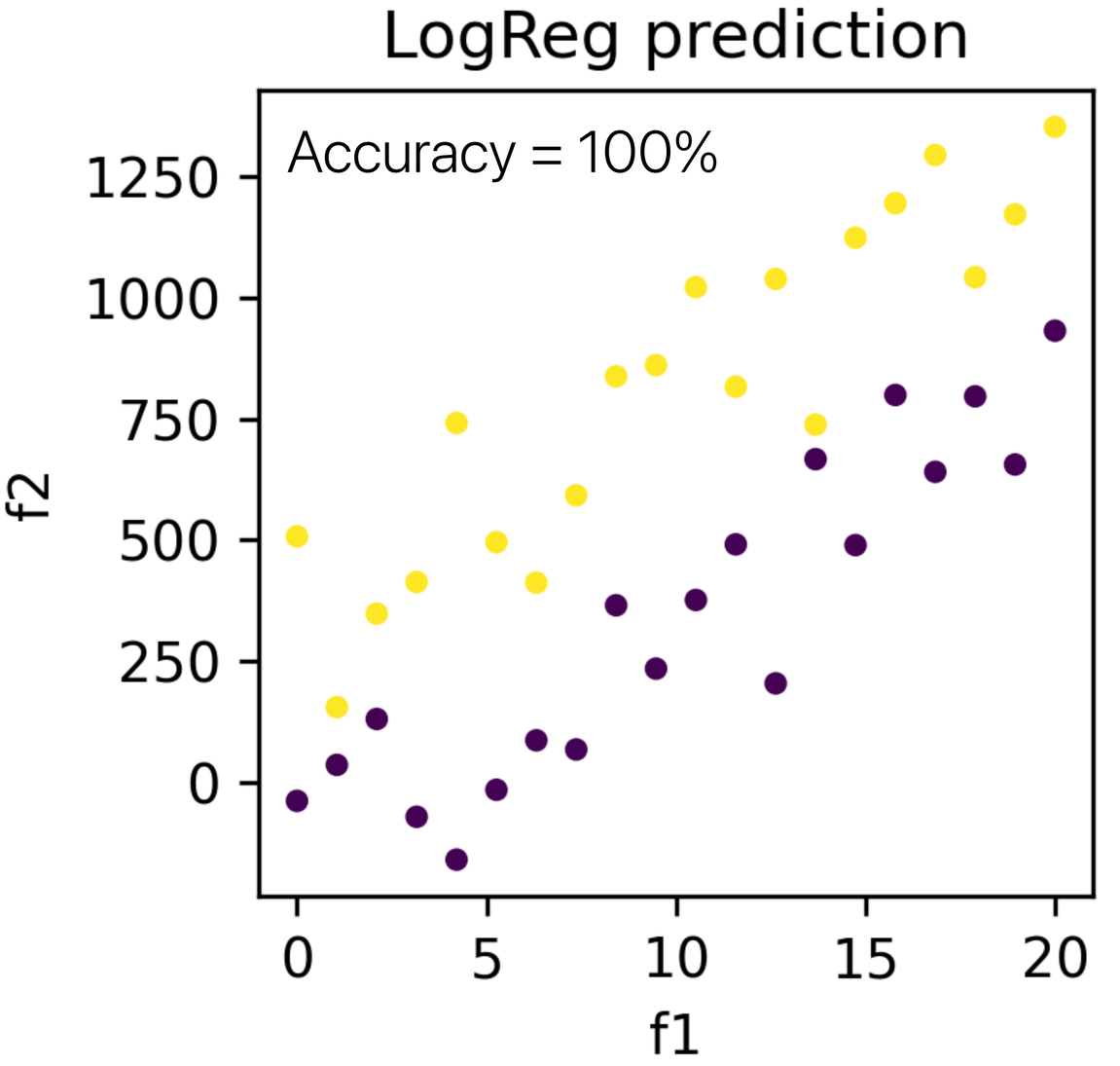
$$J(\mathbf{y}, \hat{\mathbf{y}}) = -\frac{1}{m}\sum_{i=1}^{m}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

"...the logarithm of the magnitude of the change, rather than just the distance from data to prediction."

Metrics: Accuracy, recall, precision, and related metrics

$$\text{Accuracy} = \frac{\text{correct classifications}}{\text{total classifications}} = \frac{TP + TN}{TP + TN + FP + FN}$$


True
1250
1000
750
500
250
0
f2
0
5
10
15
20
f1


LogReg prediction
Accuracy = 100%
1250
1000
750
500
250
0
f2
0
5
10
15
20
f1

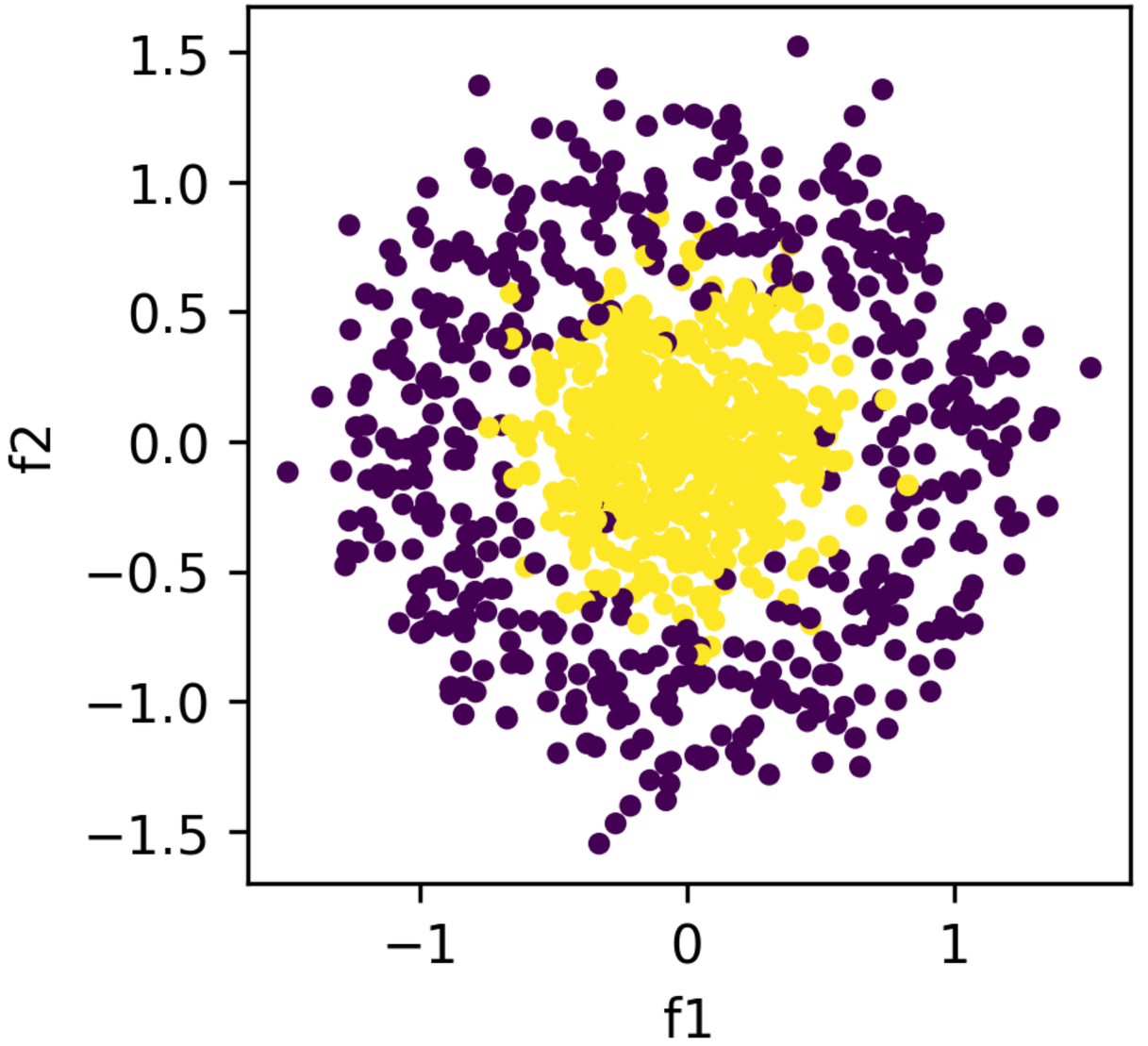
Part #2: Neural nets

Data samples:

$\{(\boldsymbol{x}_1, y_1), \ldots, (\boldsymbol{x}_i, y_i), \ldots, (\boldsymbol{x}_n, y_n)\}$

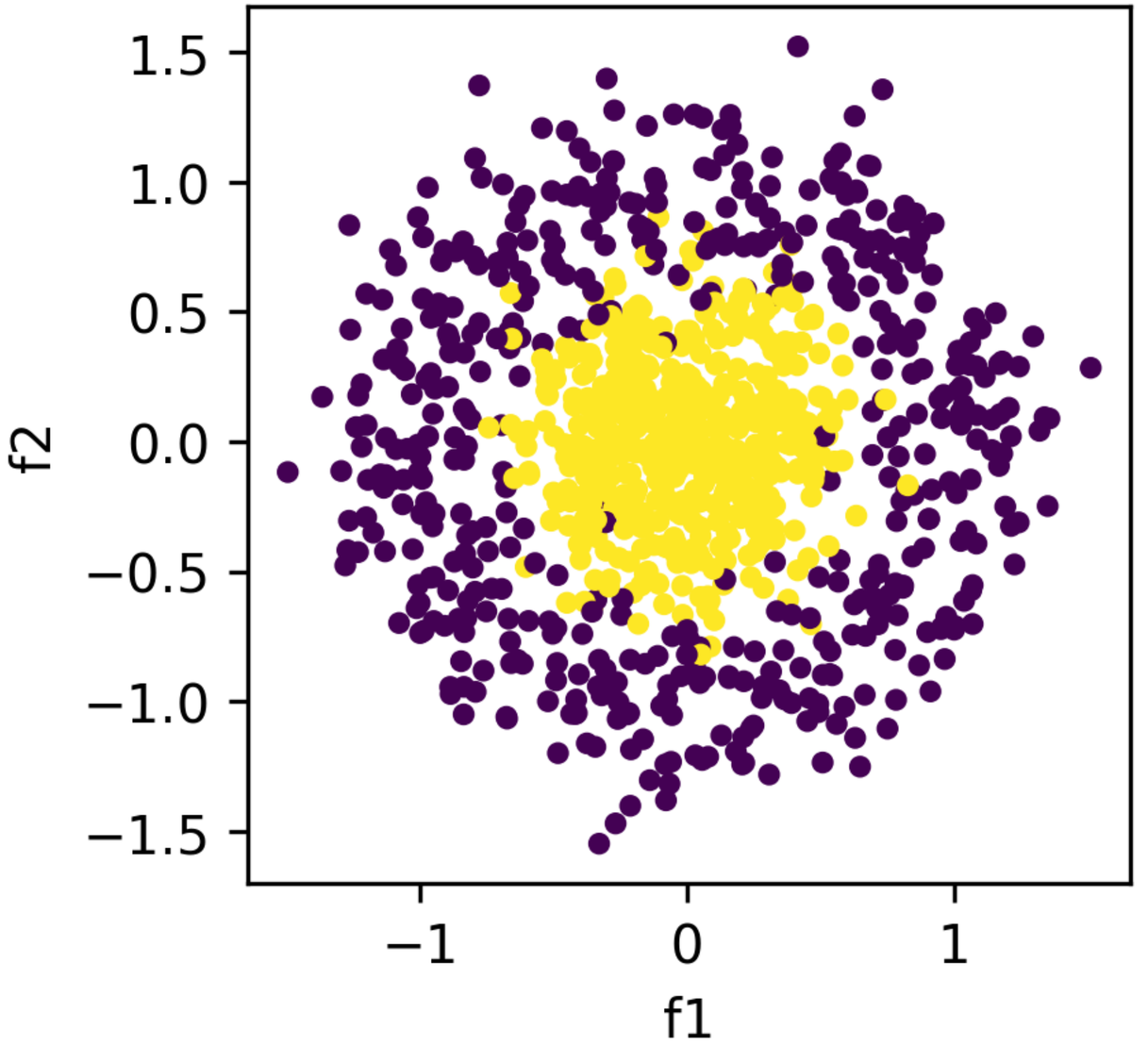
y_i -class of a sample (0 or 1)

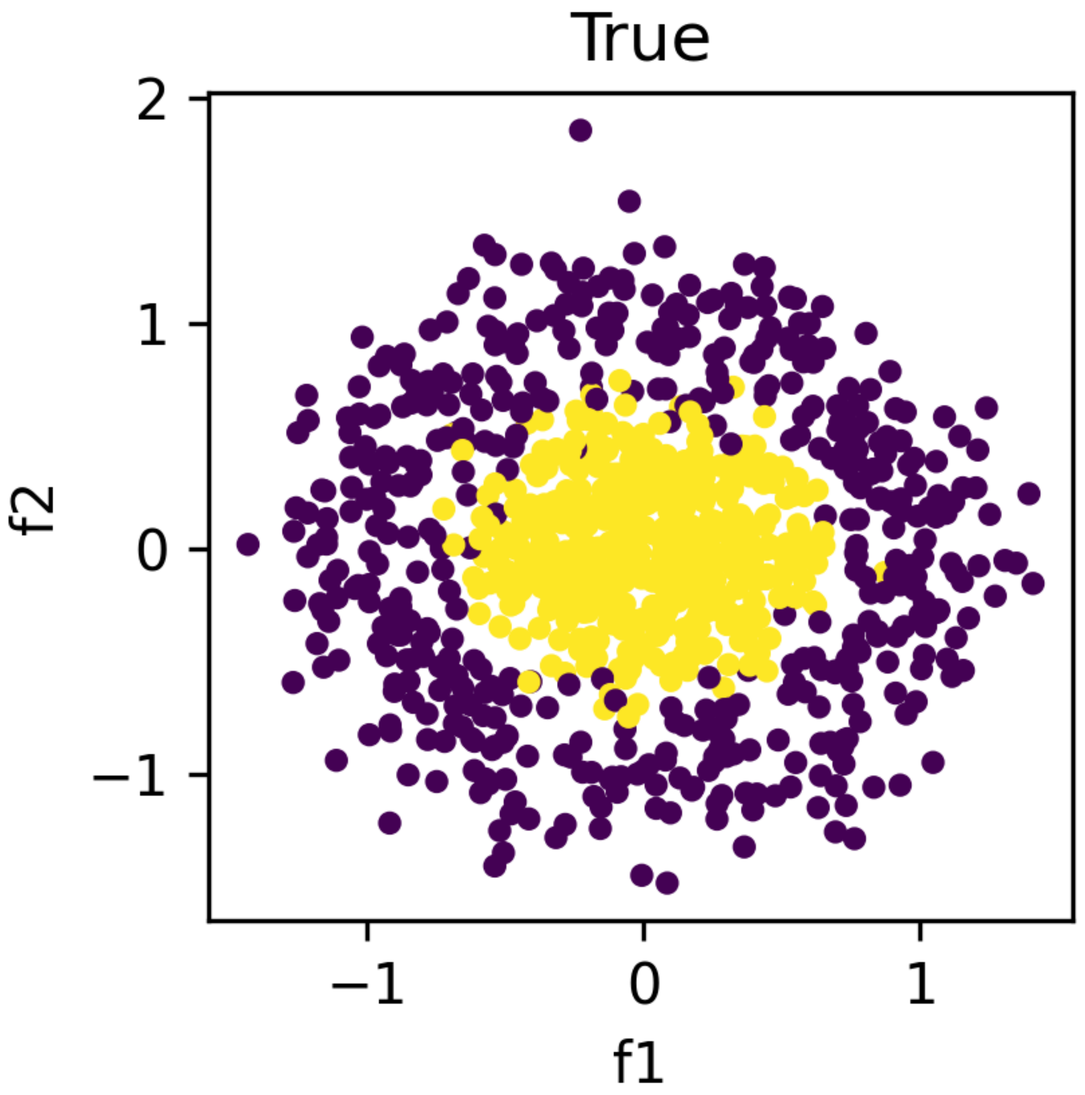
$\boldsymbol{x}_i = (x_i^1, \ldots, x_i^d)$ - $1 \; x \; d$ vector of input features

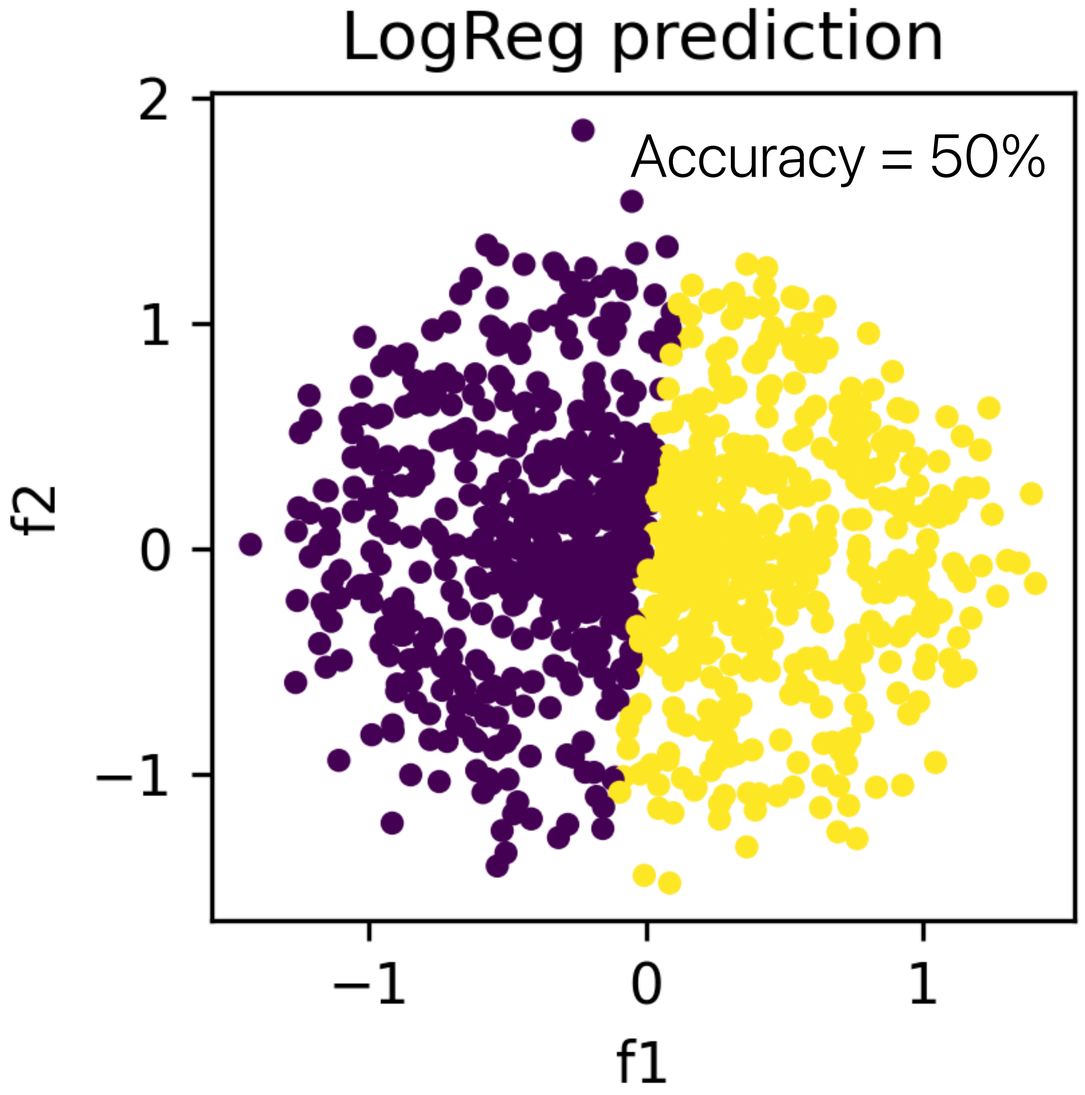


Can we draw a line to separate the classes?

Classes are linearly non-separable

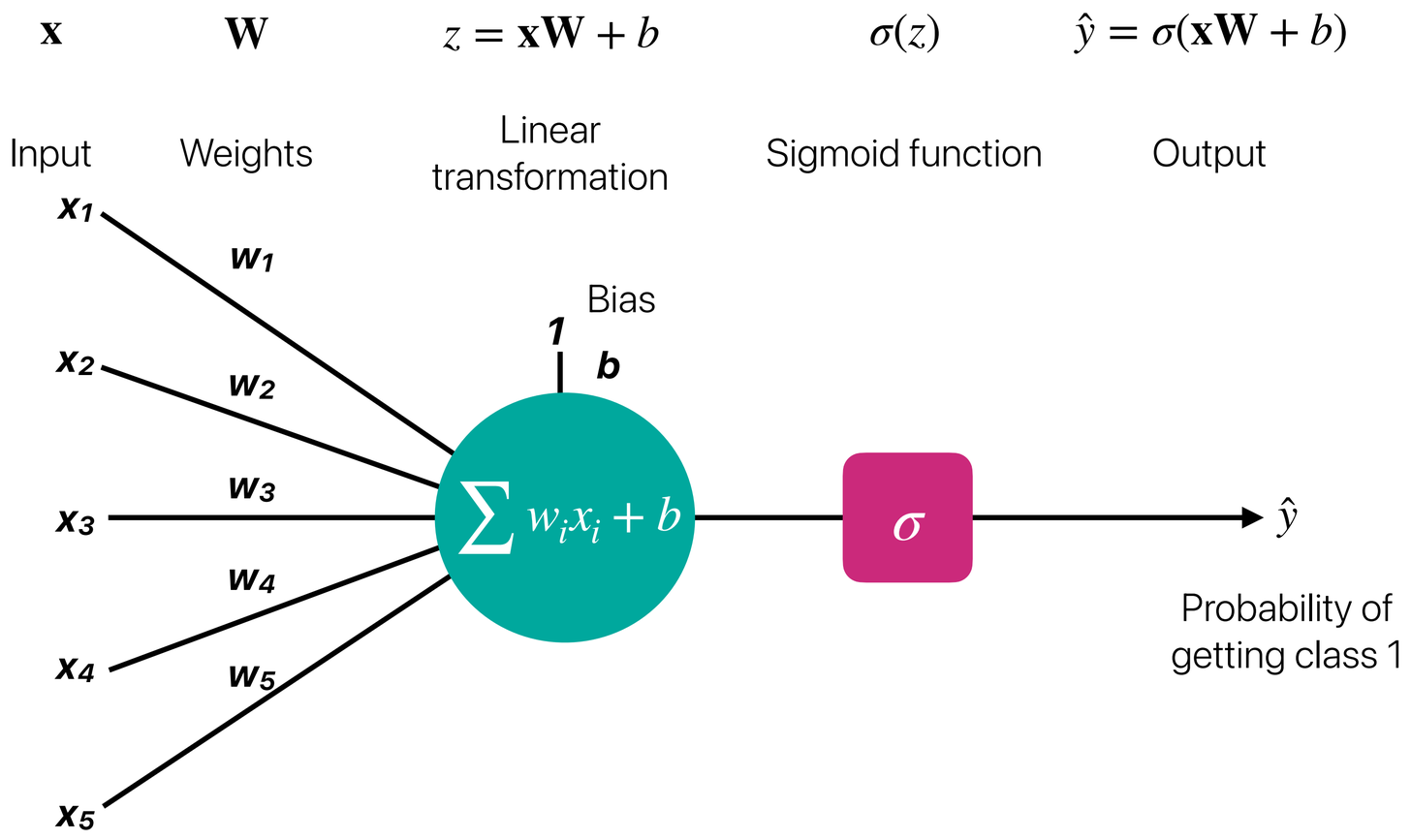



True
2
1
0
−1
f2
−1
0
1
f1

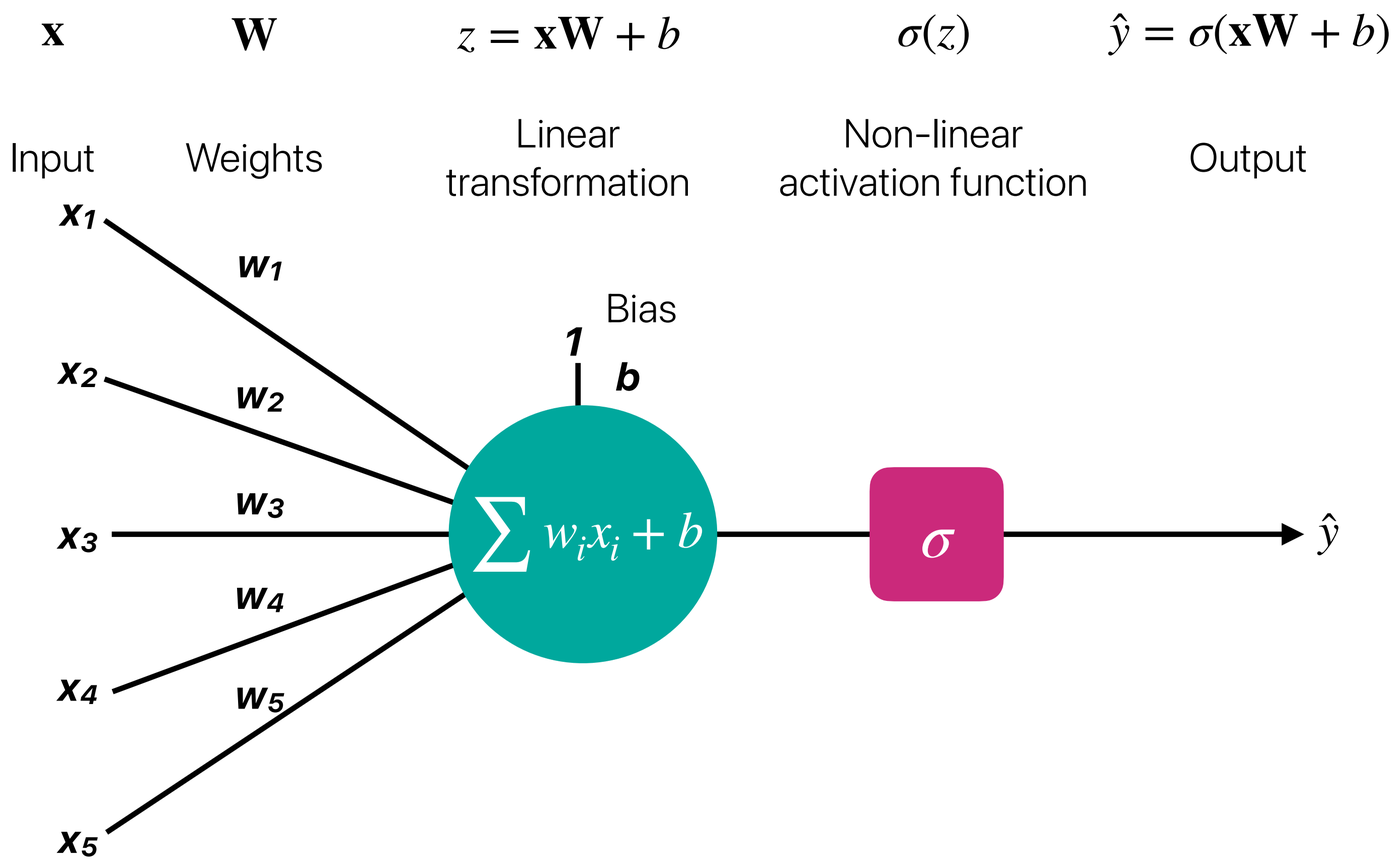

LogReg prediction
Accuracy = 50%
2
1
0
−1
f2
−1
0
1
f1

That’s why we need neural nets

Let's represent a LogReg in a following way

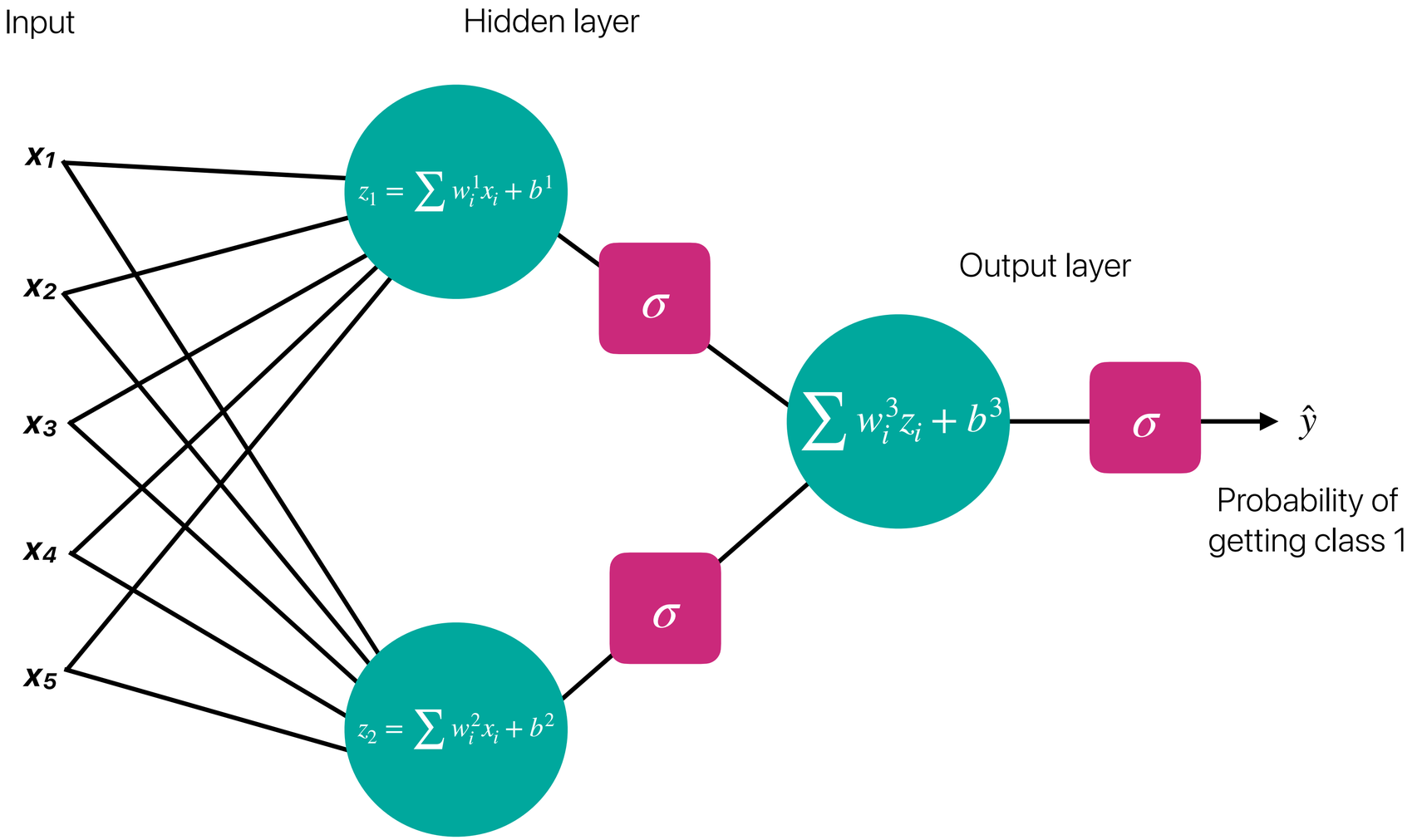


This unit is called a perceptron



One hidden layer multilayer perceptron

- Input layer
  - Features
- Hidden layer
  - Weighted sum
  - Activation
- Output layer
  - Weighted sum
  - Sigmoid
- Post-processing
  - Convert probabilities to labels



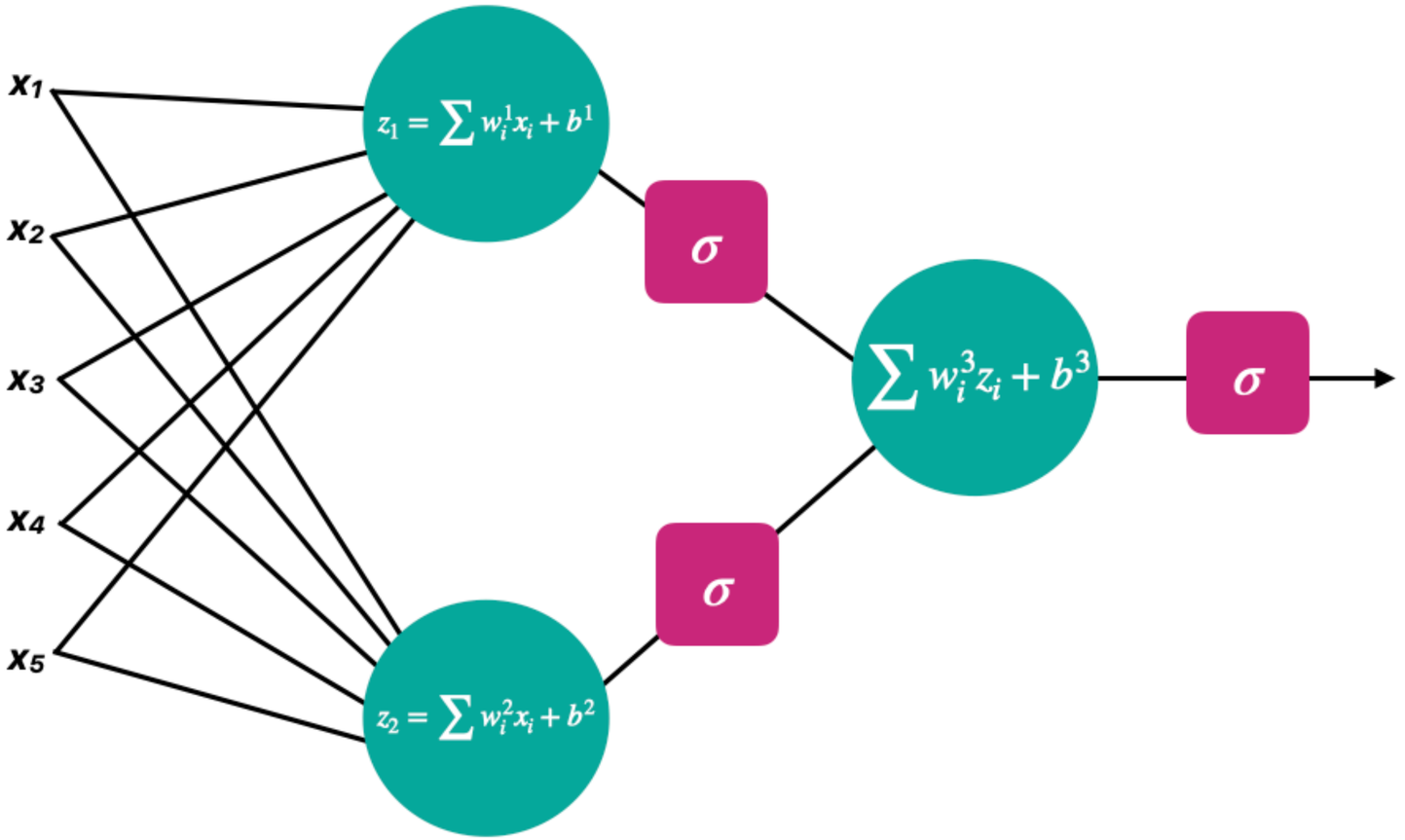
Matrix form

$$\begin{bmatrix} z_1 \\ z_2 \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1k} \\ w_{21} & w_{22} & \cdots & w_{2k} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_k \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$\mathbf{z} = W\vec{x} + \vec{b}$$

$$h = \sigma(W\vec{x} + \vec{b})$$

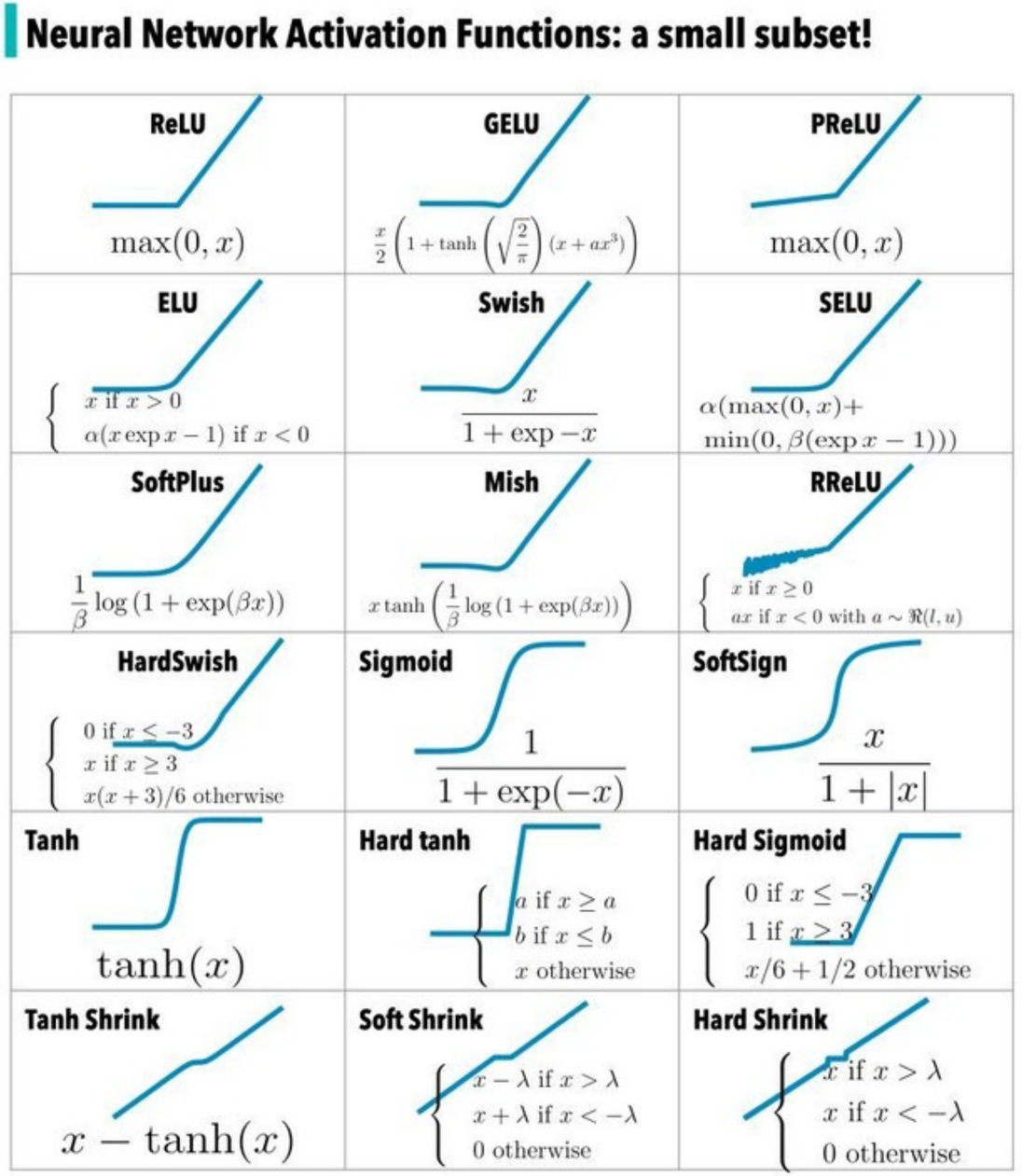
*"The output of each layer is the product of its weight matrix and the input vector plus its bias vector, all wrapped in a non-linear activation function."

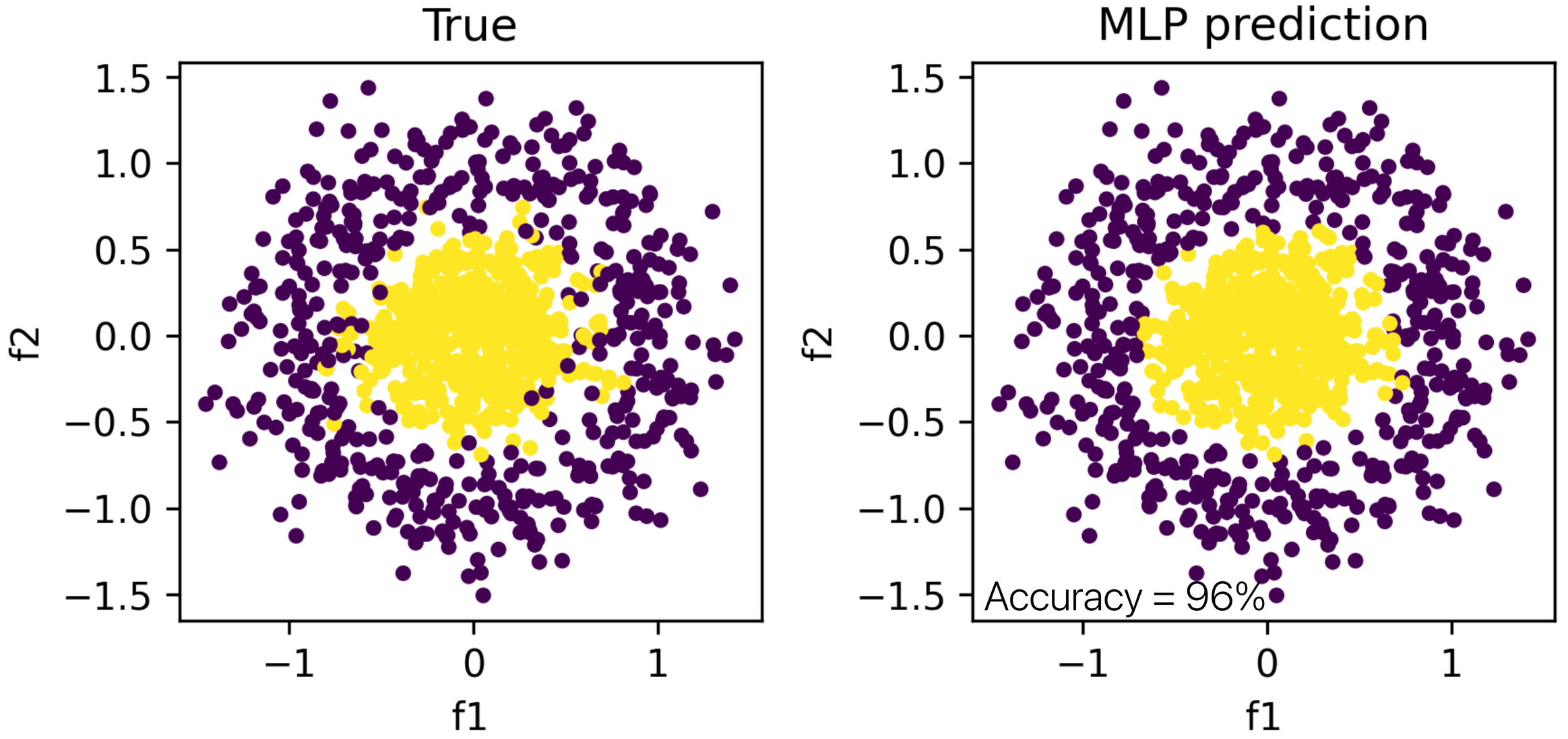


*An Introduction to Neural Networks by Satchit Chatterji

Non-linear coupling between the layers is the key

...that enables neural networks to be universal function approximators and learn complex, non-linear patterns.




True
MLP prediction
f1
f2
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−1
0
1
Accuracy = 96%

Universal Approximation Theorem states that certain models, most notably feedforward neural networks with a single hidden layer, can approximate any continuous function to an arbitrary degree of accuracy, given enough neurons in the hidden layer.

How to train NNs: Backpropagation and chain rule

Consider a simple example with a scalar feature

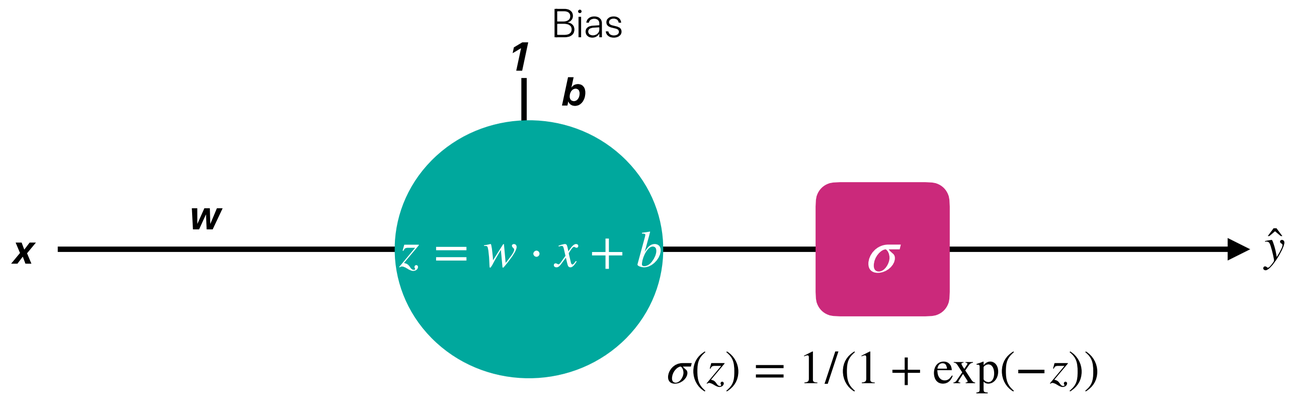
We want to find the best parameters of our model using gradient descent, i.e. update weights as follows:

$$w = w - \alpha \cdot \frac{\partial E}{\partial w}$$

$$b = b - \alpha \cdot \frac{\partial E}{\partial b}$$

Where $E = \frac{1}{2}(\hat{y} - y)^2$ is the MSE loss function

$\mathbf{x}$	$\mathbf{W}$	$z = \mathbf{x}\mathbf{W} + b$	$\sigma(z)$	$\hat{y} = \sigma(\mathbf{x}\mathbf{W} + b)$
Input	Weights	Linear transformation	Non-linear activation function	Output



Chain rule reminder:

$$y = f(g(x))$$

$$\frac{dy}{dx} = \frac{df}{dg} \cdot \frac{dg}{dx}$$

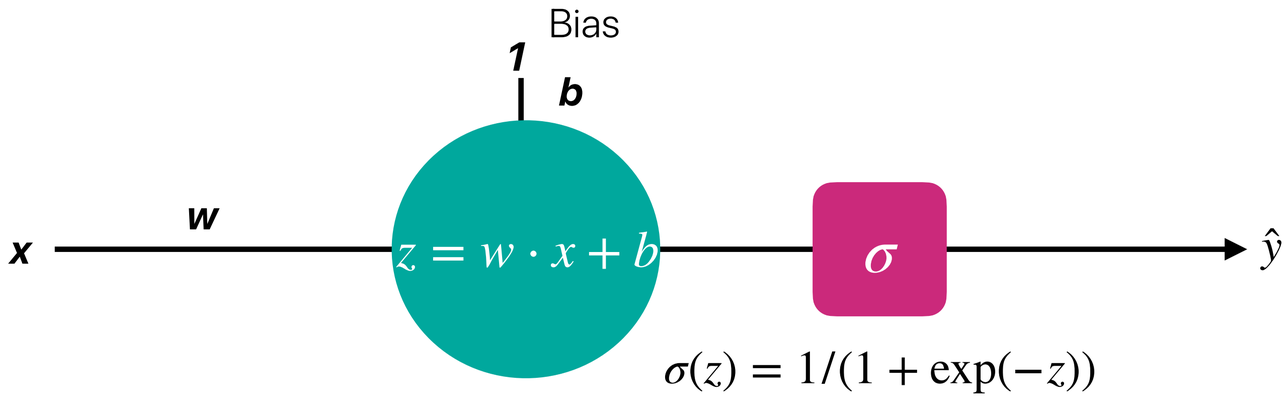
$\mathbf{x}$	$\mathbf{W}$	$z = \mathbf{x}\mathbf{W} + b$	$\sigma(z)$	$\hat{y} = \sigma(\mathbf{x}\mathbf{W} + b)$
Input	Weights	Linear transformation	Non-linear activation function	Output

According to the chain rule

$$E = \frac{1}{2}(\hat{y}(z(w)) - y)^2$$

$$\frac{\partial E}{\partial w} = \frac{\partial E(\hat{y}(z(w)))}{\partial w}$$

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w}$$



We are propagating gradients back from the last (output) layer to the first hidden layer

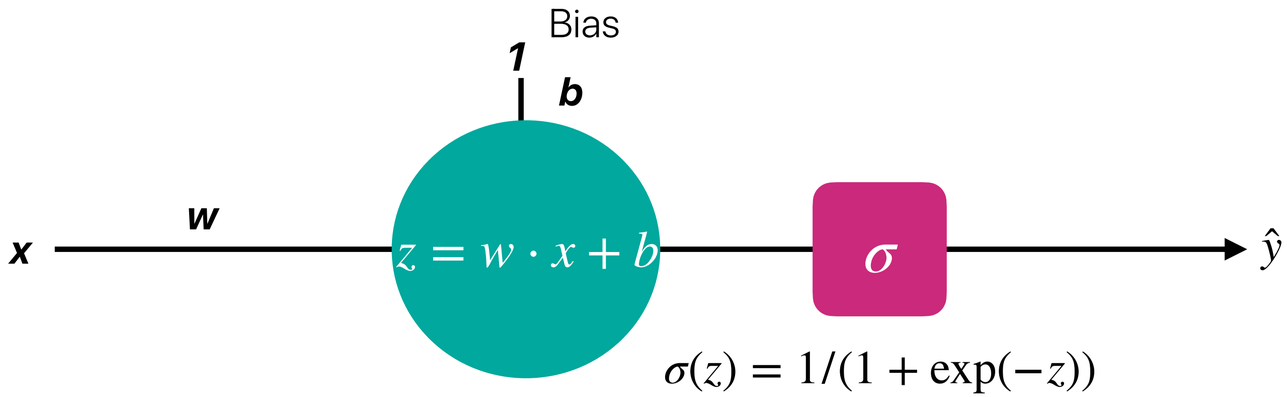
$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w}$$

$$\frac{\partial E}{\partial y} = \hat{y} - y$$

$$\frac{\partial y}{\partial z} = \frac{\partial(\frac{1}{1+exp(-z)})}{\partial z} = -\frac{exp(-z)}{1+exp(-z)} = \sigma(z) \cdot (1 - \sigma(z))$$

$$\frac{\partial z}{\partial w} = x$$

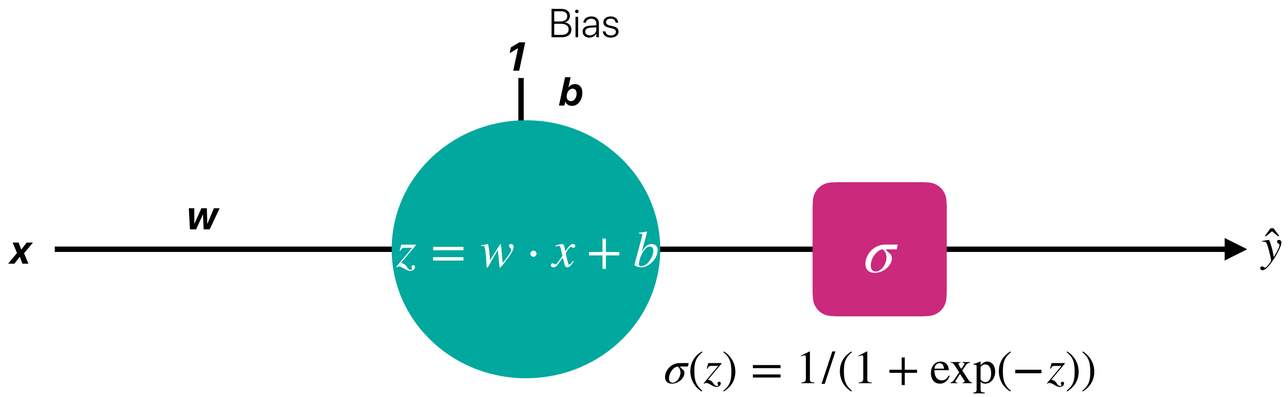
$\mathbf{x}$	$\mathbf{W}$	$z = \mathbf{x}\mathbf{W} + b$	$\sigma(z)$	$\hat{y} = \sigma(\mathbf{x}\mathbf{W} + b)$
Input	Weights	Linear transformation	Non-linear activation function	Output



$\mathbf{x}$	$\mathbf{W}$	$z = \mathbf{x}\mathbf{W} + b$	$\sigma(z)$	$\hat{y} = \sigma(\mathbf{x}\mathbf{W} + b)$
Input	Weights	Linear transformation	Non-linear activation function	Output

$$\frac{\partial E}{\partial w} = \frac{\partial E}{\partial y} \cdot \frac{\partial y}{\partial z} \cdot \frac{\partial z}{\partial w}$$

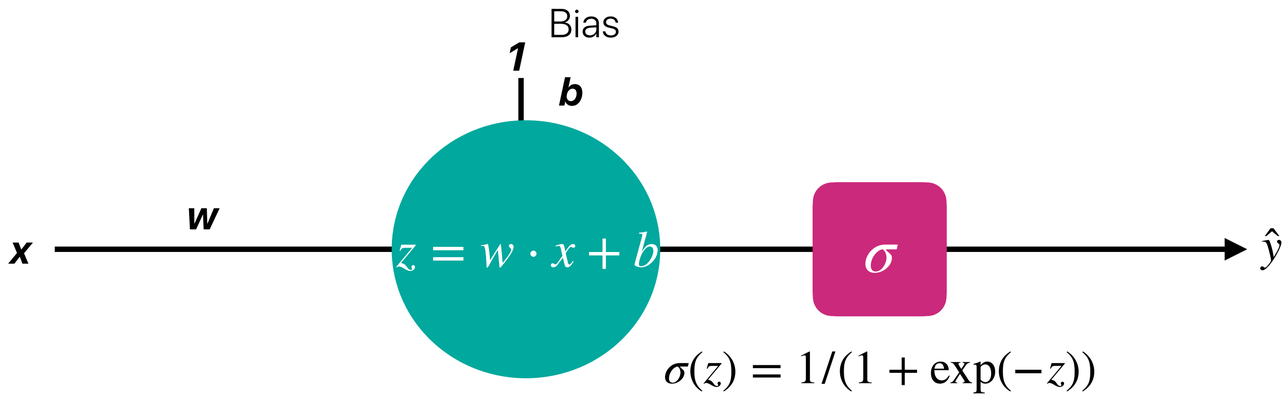
$$\frac{\partial E}{\partial w} = (\hat{y} - y) \cdot \sigma(z)(1 - \sigma(z)) \cdot x$$



$\mathbf{x}$	$\mathbf{W}$	$z = \mathbf{x}\mathbf{W} + b$	$\sigma(z)$	$\hat{y} = \sigma(\mathbf{x}\mathbf{W} + b)$
Input	Weights	Linear transformation	Non-linear activation function	Output

Similarly

$$\frac{\partial E}{\partial b} = (\hat{y} - y) \cdot \sigma(z)(1 - \sigma(z))$$



Let's rewrite MLP as follows:

$\text{MLP}_N(\mathbf{x}) = \sigma_N\Big(\mathbf{W}_N \cdots \sigma_2\Big(\mathbf{W}_2 \cdot \sigma_1\Big(\mathbf{W}_1 \cdot \mathbf{x} + \mathbf{b}_1\Big) + \mathbf{b}_2\Big) \cdots + \mathbf{b}_N\Big)$ - just a fancy (composite) function, isn't?

Or even simpler:

$\text{MLP}_N(\mathbf{x}) = L_N(L_{N-1}(\cdots L_2(L_1(\mathbf{X}, \mathbf{W}_1^*), \mathbf{W}_2^*)\cdots), \mathbf{W}_n^*)$, (* symbol specifies that W matrix includes bias term)

$$\text{MLP}_N(\mathbf{x}) = L_N(L_{N-1}(\cdots L_2(L_1(\mathbf{X}, \mathbf{W}_1^*), \mathbf{W}_2^*)\cdots), \mathbf{W}_n^*)$$

$\mathbf{x}$ - input features, $\mathbf{y}$ - true labels

Let:

$$\mathbf{h}_0 = \mathbf{x}$$

$$\mathbf{h}_k = \sigma_k\left(\mathbf{W}_k \cdot \mathbf{h}_{k-1} + \mathbf{b}_k\right) \quad \text{for } k = 1,2,\ldots,L$$

$$\hat{\mathbf{y}} = \mathbf{h}_L$$

Feed forward:

$$\mathbf{h_0} -> \mathbf{h_1} -> \ldots -> \mathbf{h_l}$$

Backward propagation:

$$\frac{\partial E}{\partial \mathbf{W}_k} = \frac{\partial E}{\partial \mathbf{h}_N} \cdot \frac{\partial \mathbf{h}_N}{\partial \mathbf{h}_{N-1}} \cdots \cdot \frac{\partial \mathbf{h}_{k+1}}{\partial \mathbf{h}_k} \cdot \frac{\partial \mathbf{h}_k}{\partial \mathbf{W}_k}$$

For a more detailed explanation consider:

– Backpropagation lecture by Prof. Roger Grosse

– Deep learning handout lecture by Prof. Evgeny Burnaev

– Neural networks handout lecture by Prof. Evgeny Burnaev

– https://en.wikipedia.org/wiki/Backpropagation

– https://www.qwertee.io/blog/an-introduction-to-backpropagation/

(Approximate) Training loop

1. Randomly initialize the model's parameters
2. For a specified number of epochs:
    A. Shuffle the training dataset $\{\mathbf{x_i}, y_i\}$
    B. Split training dataset into non-overlapping mini-batches
    C. For each mini-batch in the dataset
        I. Forward pass: Compute predictions for the batch $\hat{y} = \text{MLP}(x)$
        II. Compute the average loss over the batch: loss $= E(\hat{y}, y)$
        III. Compute gradients via backpropagation
        IV. Update the weights using the gradients: $w_{ij}^k = w_{ij}^k - \alpha \cdot \frac{\partial E}{\partial w_{ij}}$

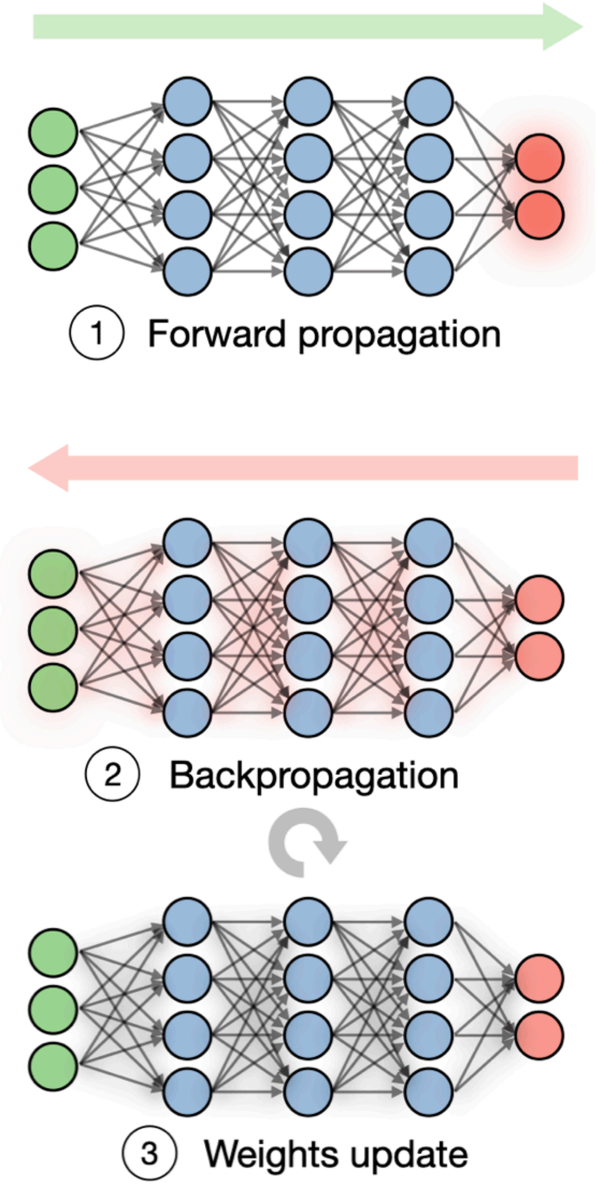
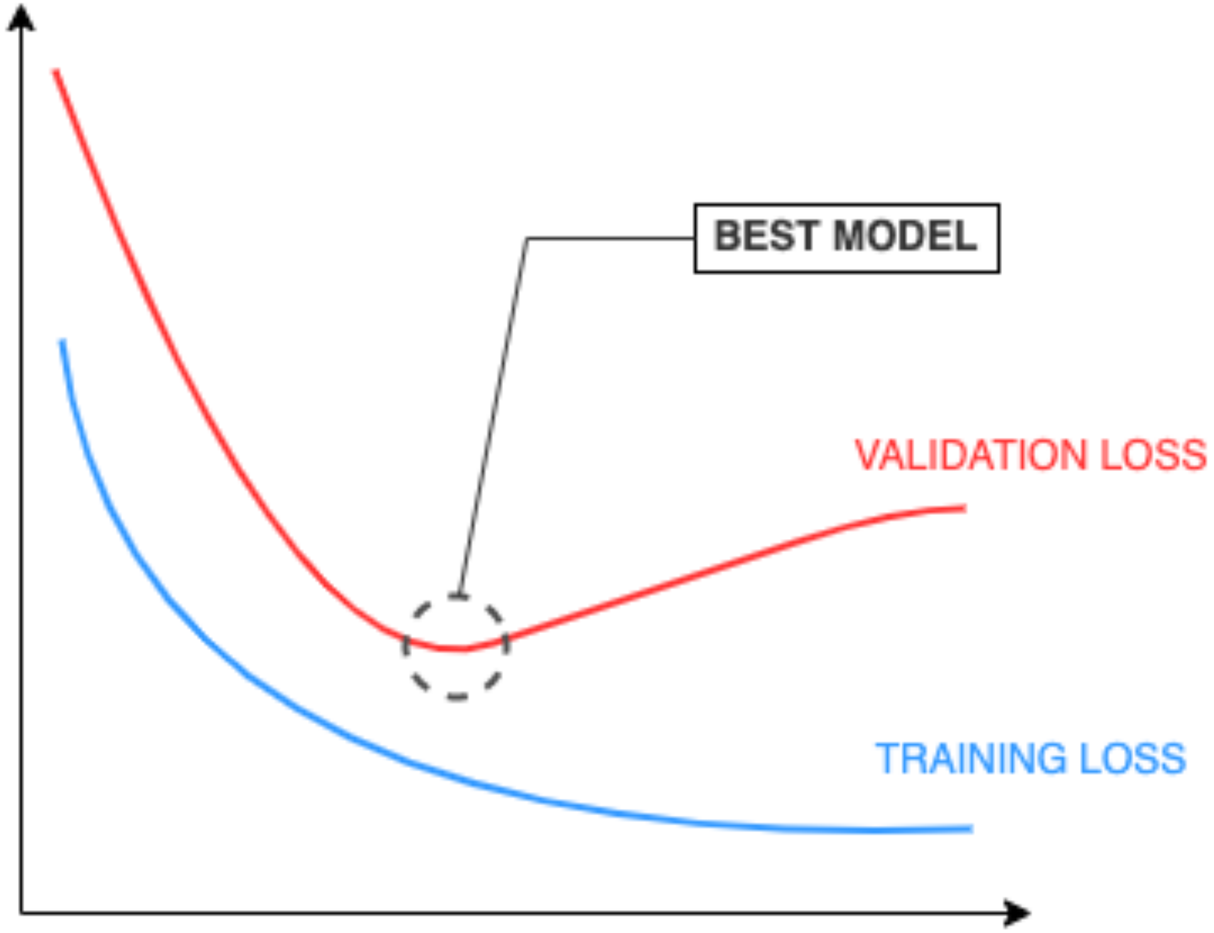


Image source

When to stop training procedure?

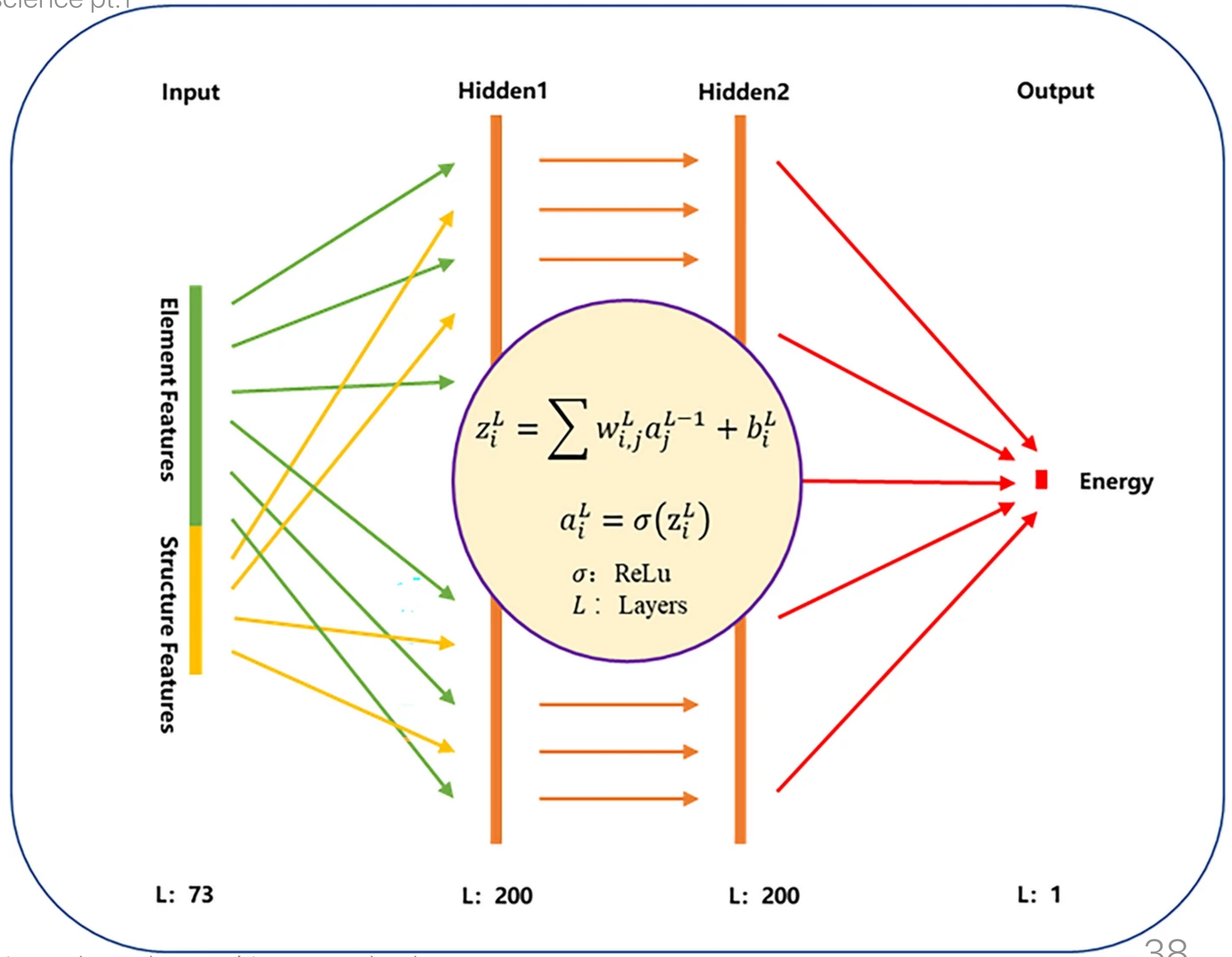
- Model parameters are updated using training data
- Validation data monitors loss on unseen data to determine when to stop
- Best model is then evaluated on test data



Good practices for neural network training: source

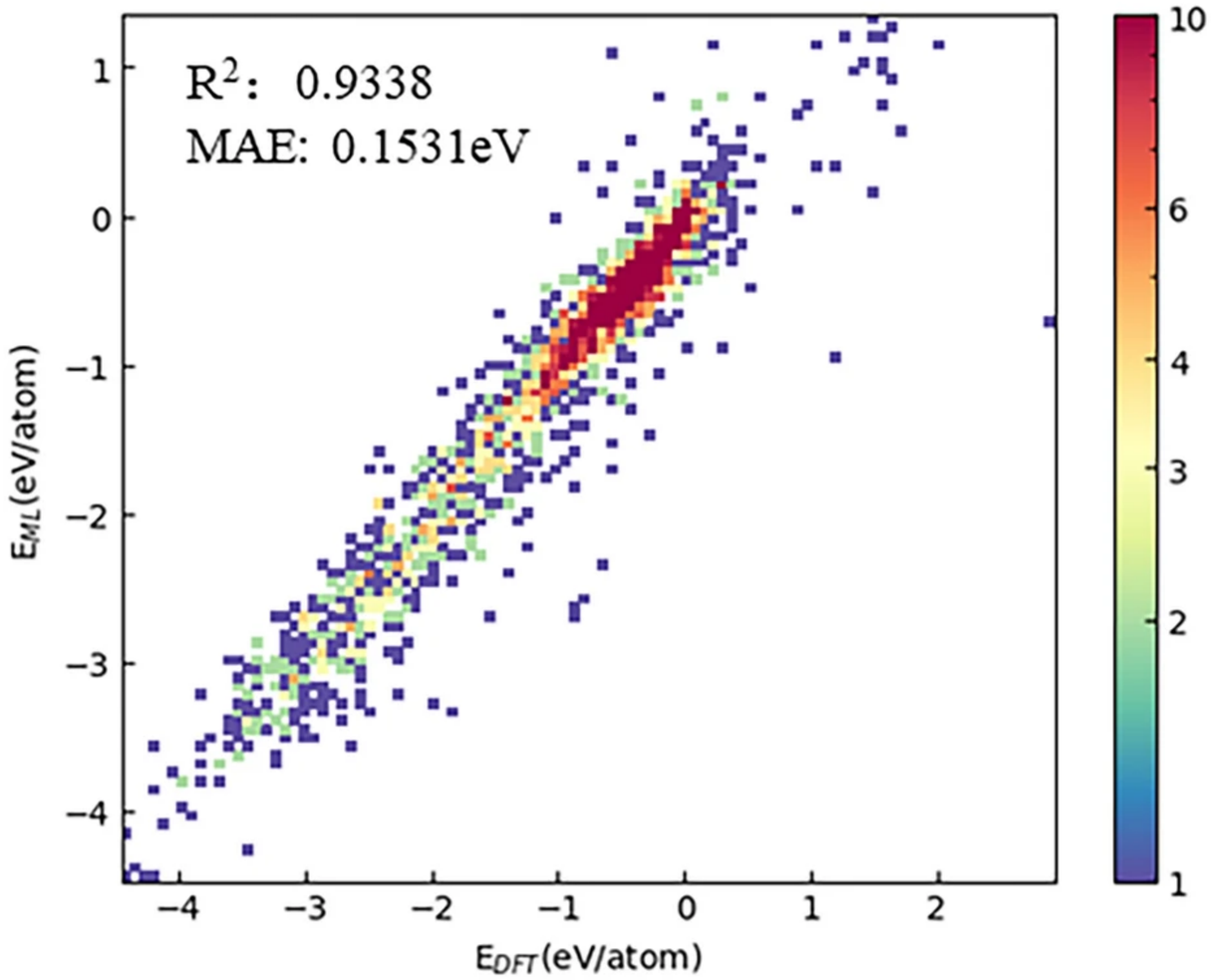
MLP for formation energy

- Input:
  - Aggregated element features
  - Aggregated structure features
- Output:
  - Formation energy per atom



MLP for formation energy

- Input:
  - Aggregated element features
  - Aggregated structure features
- Output:
  - Formation energy per atom



MLP pros/cons

- Pros
  - Universal approximator
  - Versatility -> easy to implement for various tasks (regression/classification)
- Cons
  - Requires large datasets
  - In theory, MLP can capture more complex non-linear relationships compared to e.g. RF
    - In practice, well tuned RF may outperform MLP
  - Black box -> no interpretability
  - Computational complexity

- The Multilayer Perceptron (MLP) is one class of neural network
- These days, Graph Neural Networks (GNNs) are the most popular nets used in computational materials science
- We will discuss GNNs in the next lecture

Resources

- https://uvadlc-notebooks.readthedocs.io/en/latest/tutorial_notebooks/tutorial2/Introduction_to_PyTorch.html
- https://pabloinsente.github.io/the-multilayer-perceptron