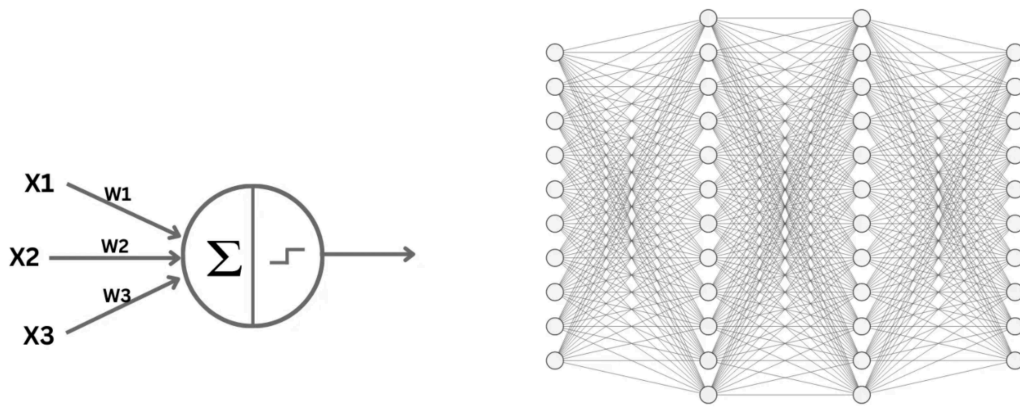


## Basics of deep learning

*Universal approximation theorems* state that a single hidden-layer neural network  $f$  with sufficiently many neurons and a non-linear activation function  $\sigma$  can approximate any continuous function to arbitrary accuracy (given enough data). Thus, artificial neural networks serve as "universal approximators" and form the basis of the most widely used models in deep learning today.

Let  $x \in \mathbb{R}^d$  denote a feature vector of a learning instance and let  $y$  denote its class label. In the case of classification, the labels  $y \in \{0, 1\}$  or  $y \in \{\text{foo}, \text{bar}, \text{baz}, \dots\}$  are categorical, whereas in regression, the labels  $y \in \mathbb{R}$  are real-valued.



### Single-layer perceptron

A *single-layer perceptron* (SLP) is mathematically expressed as

$$f(x) = \sigma(w^T x + b),$$

where  $w \in \mathbb{R}^d$  is the learnable weight vector and  $b \in \mathbb{R}$  is the bias term. Then, the learning objective is to approximate the true label  $y$  as

$$y \approx f(x).$$

### Multilayer perceptron

A *multilayer perceptron* (MLP) with  $L - 1$  hidden layers is mathematically expressed as

$$x^{(0)} = x,$$

$$x^{(l+1)} = \sigma(W^{(l)}x^{(l)} + b^{(l)}) \text{ for } \forall l \in \{0, \dots, L-2\},$$

$$f(x) = \sigma(w^T x^{(L-1)} + b),$$

where  $W^{(l)} \in \mathbb{R}^{d \times d}$  are the learnable weight matrices and  $b^{(l)} \in \mathbb{R}^d$  are the bias vectors.

Note that the bias vectors  $b^{(l)}$  can be absorbed by appending a constant 1 to the hidden representations  $x^{(l)}$ . The last column of each weight matrix  $W^{(l)}$  then effectively acts as the corresponding bias vector.

For simplicity, the above expressions assume that the input features  $x$  and hidden representations  $x^{(l)}$  are all  $d$ -dimensional vectors. In practice, the dimensionality of the representations can, for instance, decrease in subsequent layers.

## Non-linear activation

Non-linear *activation functions*  $\sigma$  increase the expressivity of artificial neural networks, enabling them to learn or approximate non-linear mappings. The most popular choices include

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

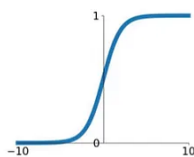
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \text{ and}$$

$$\text{ReLU}(x) = \max(0, x).$$

For additional examples of activation functions, see [PyTorch](#).

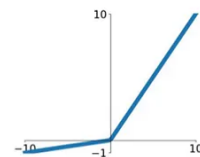
### Sigmoid

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



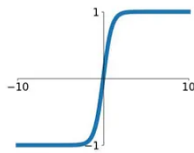
### Leaky ReLU

$$\max(0.1x, x)$$



### tanh

$$\tanh(x)$$

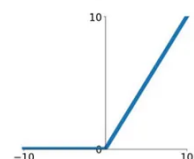


### Maxout

$$\max(w_1^T x + b_1, w_2^T x + b_2)$$

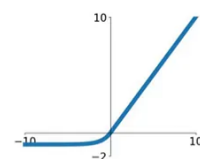
### ReLU

$$\max(0, x)$$



### ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



## Loss functions

The learning objective can be formulated as an optimization problem that minimizes a chosen *loss function*  $\mathcal{L}$  over  $n$  training instances  $(\mathbf{x}, \mathbf{y}) = \{(x_i, y_i)\}_{i=1}^n$ . More precisely,

$$\min_{\Theta} \mathcal{L}(\mathbf{y}, f(\mathbf{x})),$$

where  $\Theta$  denotes the set of learnable parameters (e.g., weight matrices).

## Regression

In the case of regression, where  $\mathbf{y} \in \mathbb{R}^n$ , a standard choice is the  $L_2$  loss (also known as *MSE*), defined as

$$\mathcal{L}(\mathbf{y}, f(\mathbf{x})) = \|\mathbf{y} - f(\mathbf{x})\|_2^2.$$

Other common choices include the  $L_1$  loss (also known as *MAE*), Huber loss, and various others (for details, see [PyTorch](#)).

## Binary classification

In the case of binary classification, where  $\mathbf{y} \in \{0, 1\}^n$ , a standard choice is the binary cross-entropy loss (*BCE*), defined as

$$\mathcal{L}(\mathbf{y}, f(\mathbf{x})) = - \sum_{i=1}^n y_i \log f(x_i) + (1 - y_i) \log(1 - f(x_i)),$$

where  $f(\mathbf{x})$  is the result of a sigmoid activation in the final layer.

## Multi-class classification

In the general case of multi-class classification over  $C$  classes, where  $\mathbf{y} \in \{0, 1\}^{n \times C}$  are one-hot encoded labels, a standard choice is again the cross-entropy loss (*CE*), defined as

$$\mathcal{L}(\mathbf{y}, f(\mathbf{x})) = - \sum_{i=1}^n \sum_{c=1}^C y_{ic} \log f(x_i)_c,$$

where  $f(x_i)_c$  is the output of the softmax activation for class  $c$  in the last layer.

Other common choices include Huber loss, focal loss, and various others (for details, see [PyTorch](#)).

## Machine learning as optimization

Modern deep neural networks are trained using an optimization procedure known as *gradient descent*. Since computing the full gradient of the loss function  $\nabla_{\Theta} \mathcal{L}$  over the entire dataset is computationally prohibitive, training instead relies on *stochastic gradient descent*,

which estimates the gradient using a randomly sampled (mini)batch of training instances. (Computing the full gradient is too slow on a CPU and too memory-intensive for a GPU.) However, stochastic gradient descent is an unbiased estimator of the full gradient, but there is no guarantee on the rate of convergence.

A single update of stochastic gradient descent on a minibatch is called an "iteration," while a complete pass over the entire dataset is referred to as an "epoch." In practice, only a few epochs are typically applied for architecture search and hyperparameter tuning, whereas training a final model for production may require hundreds of epochs or more (depending on the number of training instances).

1. **Sampling:** sample a minibatch of training instances  $(\mathbf{x}, \mathbf{y})$ .
2. **Forward pass:** compute the loss function  $\mathcal{L}(\mathbf{y}, f(\mathbf{x}))$  over the minibatch.
3. **Differentiation:** compute gradients

$$\nabla_{\Theta} \mathcal{L} = \frac{\partial \mathcal{L}}{\partial \Theta}$$

using automatic differentiation in *PyTorch*, which applies the chain rule to efficiently compute gradients.

4. **Back-propagation:** update the learning parameters

$$\Theta \leftarrow \Theta - \eta \nabla_{\Theta} \mathcal{L}$$

via gradient descent, where  $\eta$  is the learning rate, which can be adjusted dynamically by an optimization algorithm such as *Adam* (for details, see [PyTorch](#)).

**Repeat:** continue the steps above until convergence. In theory, this means reaching a point where  $\nabla_{\Theta} \mathcal{L} = 0$ . In practice, training usually stops once performance on the validation set ceases to improve or the specified maximum number of epochs has been reached.

