Neural Networks - Basics

Introduction

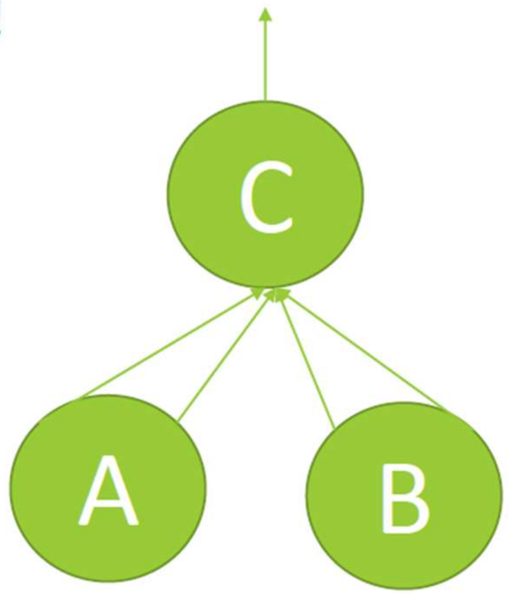
- A neural network is simply a function that takes in an input X and computes an output Y.
- What sets it apart from a function such as, say y = sin(X), is the manner in which it is constructed.
- It is composed as a network of many, very simple elements that together somehow manage to perform complex tasks.
- The term "neural" itself derives from the fact that the simple units that are networked to compose the model are conceptually similar to neurons in the brain, and were in fact originally designed as models for these neurons.

The Biological Inspiration

- Neurons in your cerebral cortex are connected via axons
- A neuron "fires" to the neurons it's connected to, when enough of its input signals are activated.
- Very simple at the individual neuron level – but layers of neurons connected in this way can yield learning behavior.
- Billions of neurons, each with thousands of connections, yields a mind

The First Artificial Neuron

- 1943!!

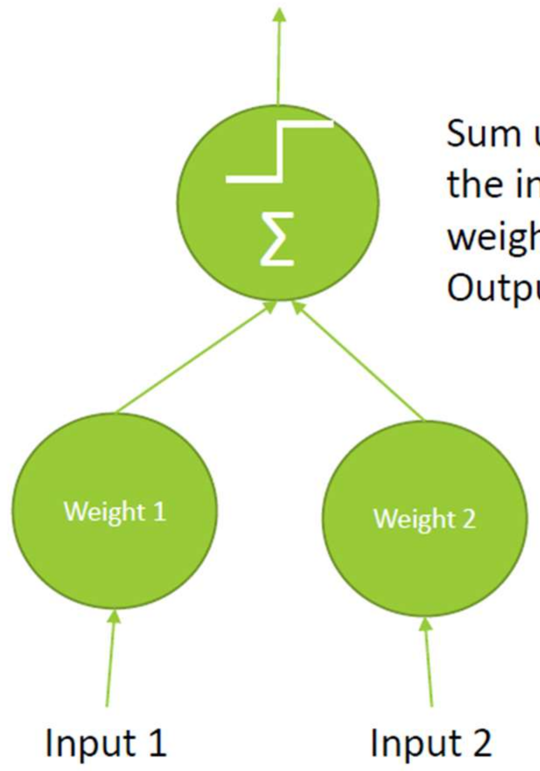


An artificial neuron “fires” if more than N input connections are active.

Depending on the number of connections from each input neuron, and whether a connection activates or suppresses a neuron, you can construct AND, OR, and NOT logical constructs this way.

Linear Threshold Unit (LTU)

- 1957!
- Adds weights to the inputs; output is given by a step function

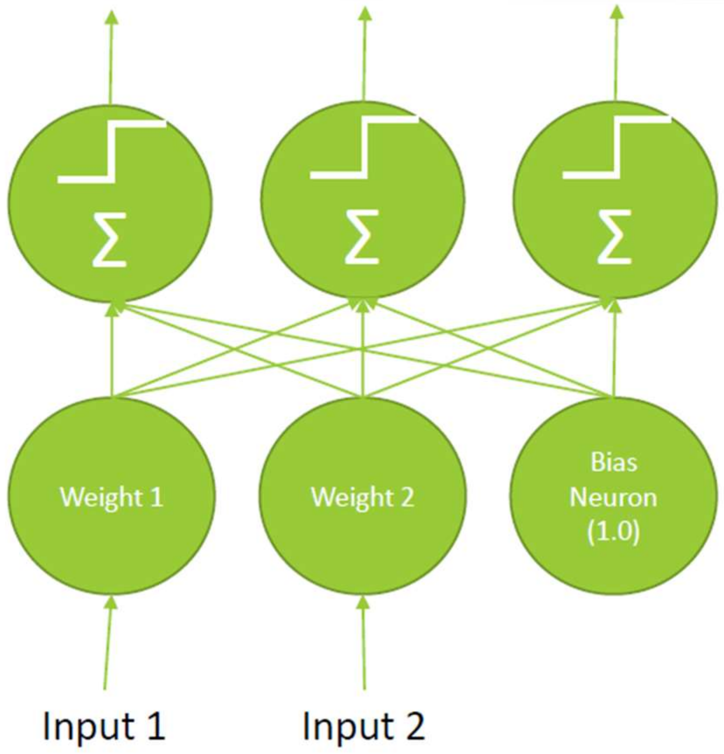


Sum up the products of the inputs and their weights
Output 1 if sum is >= 0

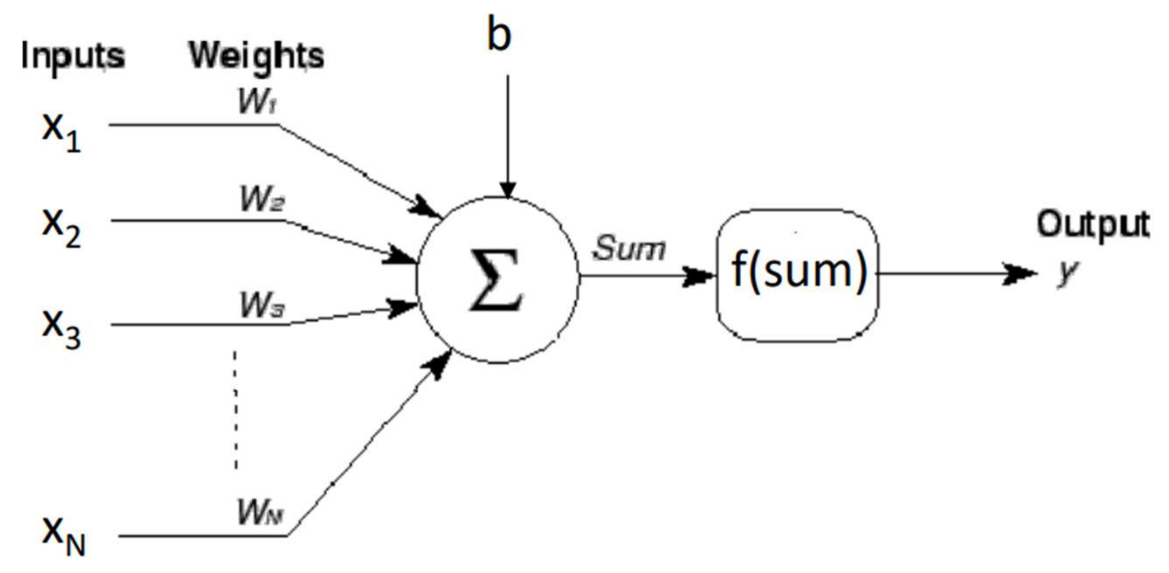
Perceptron

- A layer of LTU's
- A perceptron can learn by reinforcing weights that lead to correct behavior during training
- This too has a biological basis ("cells that fire together, wire together")

Developed by Frank Rosenblatt



Perceptron



The actual computation performed by the perceptron is as follows:

It takes in a set of inputs x_1,x_2, $\cdots$, x_N. It computes an affine combination z given by

$$z = \sum_{i=1}^{N} w_i x_i + b$$

b is the bias

It applies an activation function $a()$ to z to compute the output y as

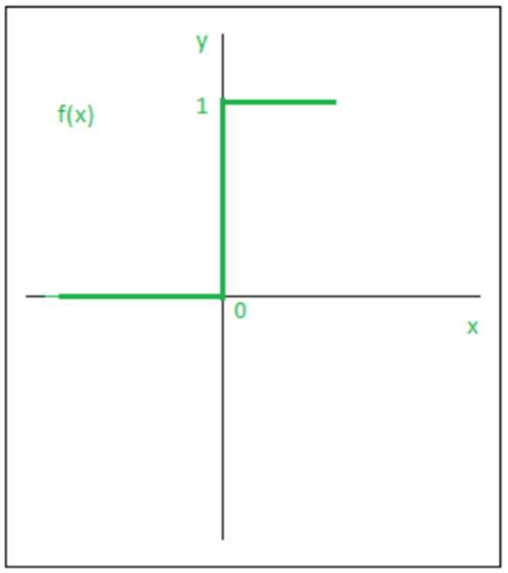
$$y = a(z)$$

Activation Functions

Activation function of a node defines the output of that node given an input or set of inputs. Activation functions decide whether a neuron should be activated or not by calculating the weighted sum and further adding bias to it.

- **Step Function** is one of the simplest kind of activation functions. In this, we consider a threshold value and if the value of net input say **y** is greater than the threshold then the neuron is activated.

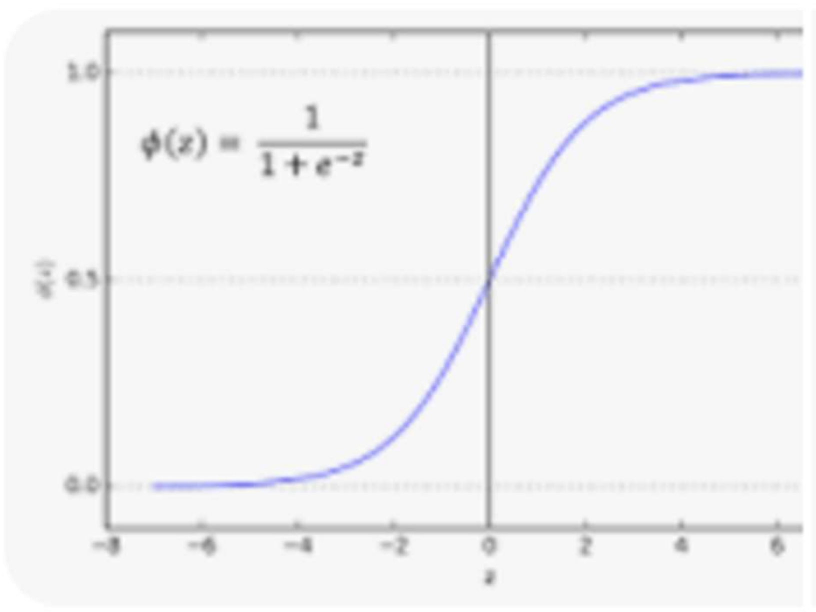
$$f(x) = 1, \text{if x} >= 0$$
$$f(x) = 0, \text{if x} < 0$$



Activation Functions

- **Sigmoid Function** takes any real value as input and outputs values in the range of 0 to 1. The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0

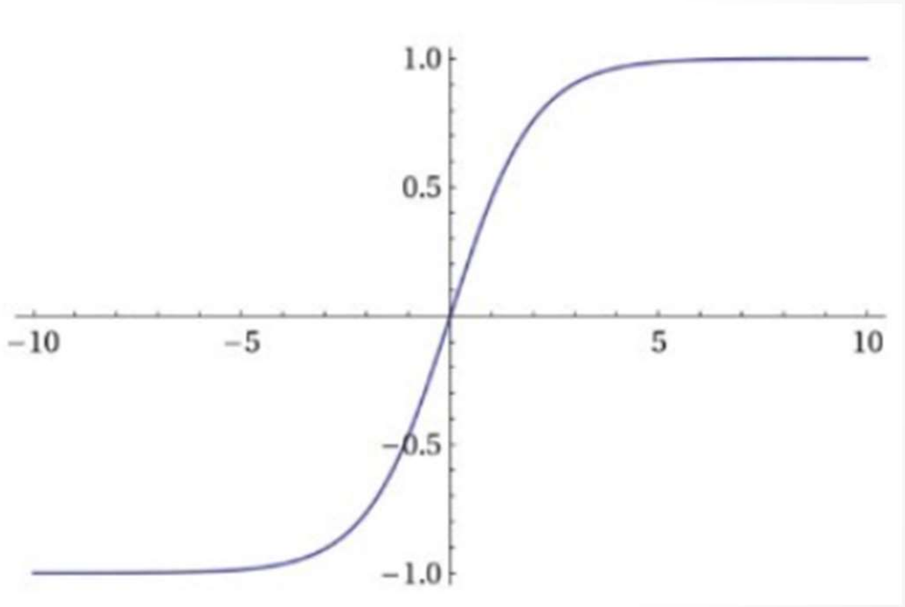
$$S(x) = \frac{1}{1 + e^{-x}}$$



Activation Functions

- **SoftMax** is an activation function that scales numbers/logits into probabilities. It is often used as the last activation function of a neural network to normalize the output of a network to a probability distribution over predicted output classes.

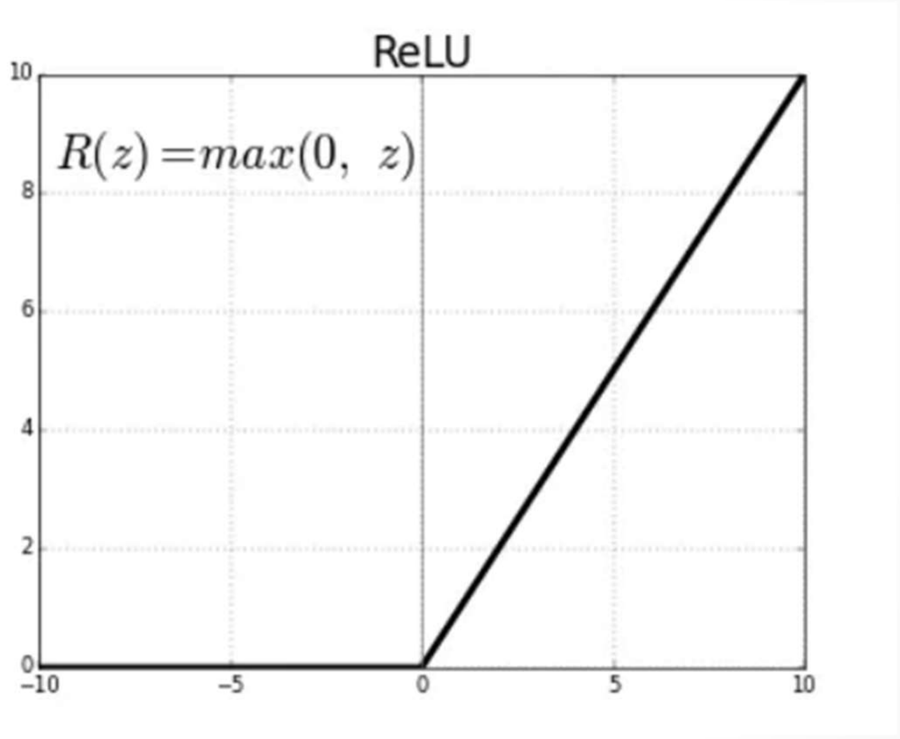
$$S(y)_i = \frac{\exp(y_i)}{\sum_{j=1}^{n} \exp(y_j)}$$



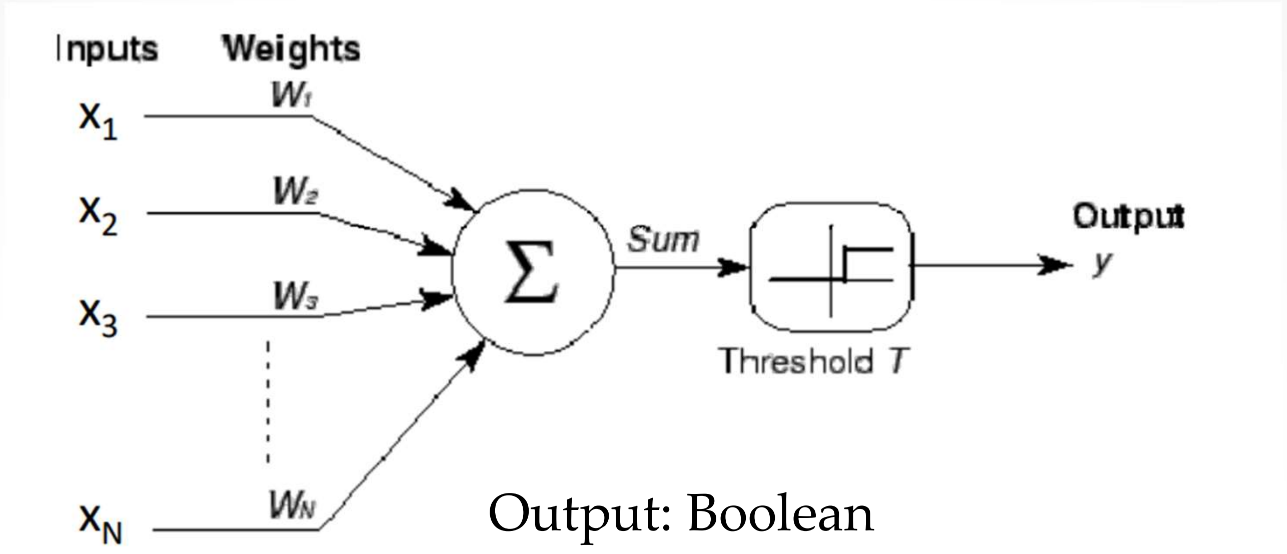
Activation Functions

- The rectified linear activation function or **ReLU** is a **non-linear** function or **piecewise linear** function that will output the input directly if it is positive, otherwise, it will output zero.

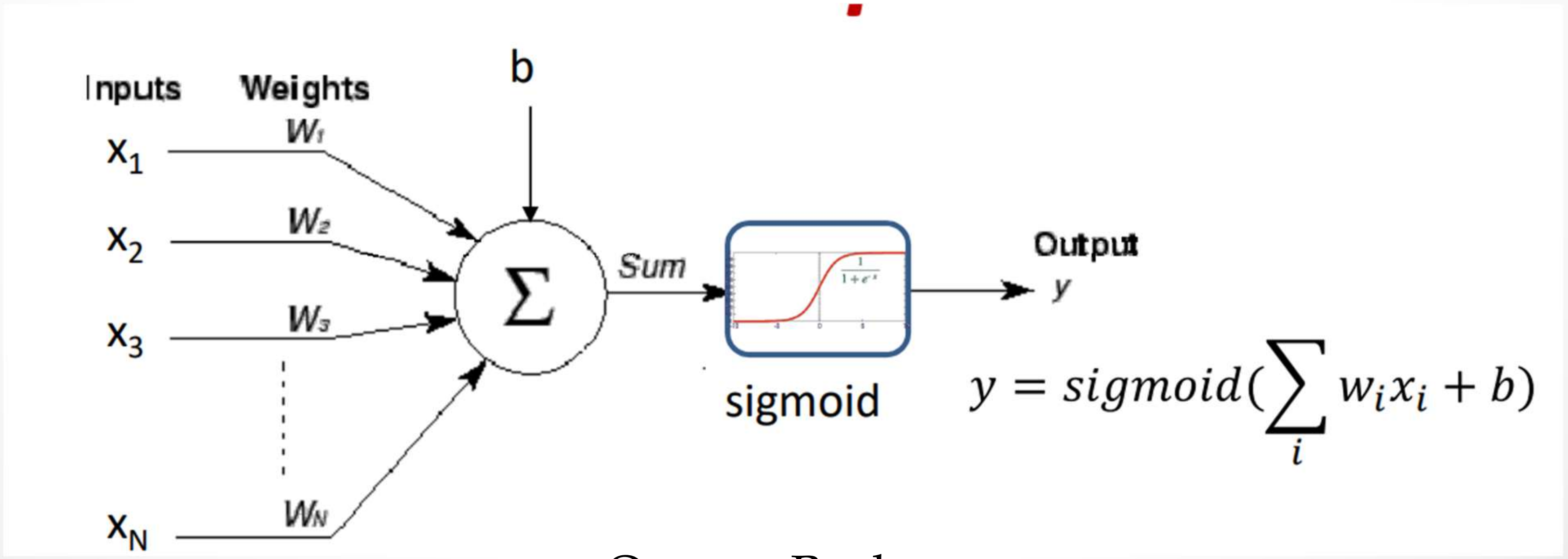
  $f(x) = \max(0,x)$



Perceptron Outputs



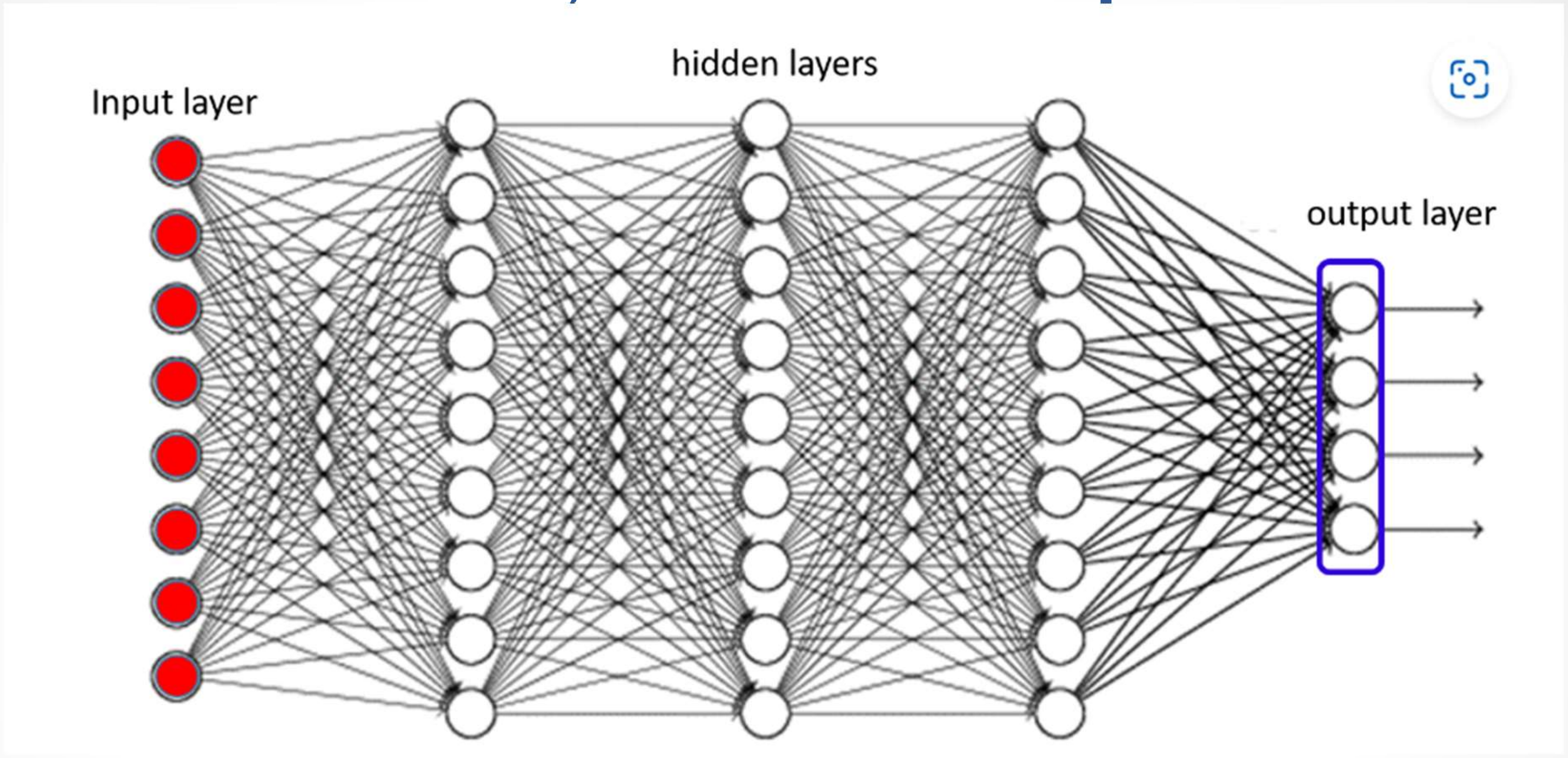
Output: Boolean



$$y = sigmoid(\sum_i w_i x_i + b)$$

Output: Real

Multi-layer Perceptron



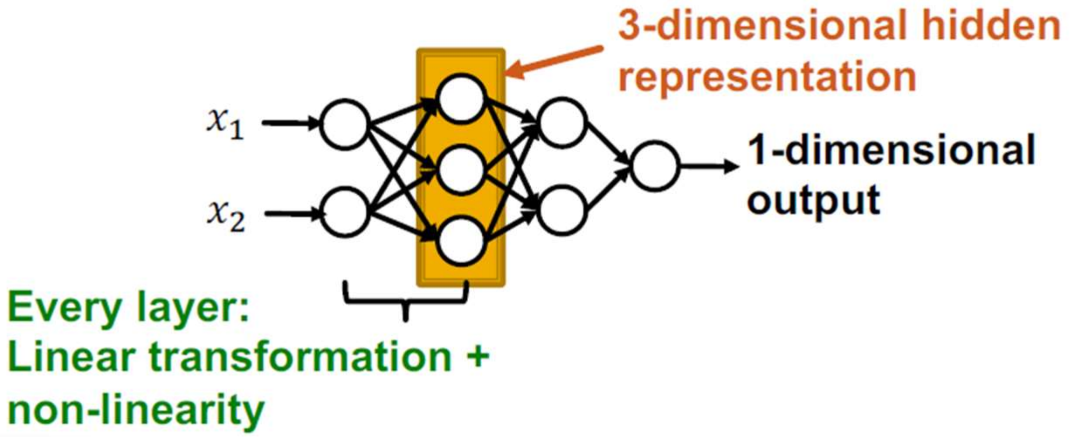
- The MLP must take in an input (comprising components $x_1, \cdots, x_N$) and compute the output (comprising components $y_1, \cdots, y_M$).
- The computation of the MLP is preformed sequentially from the neurons closest to the input, progressing until the neurons at the output, in such a manner that when each neuron is evaluated, all of the values it requires as input are already evaluated.

Multi-layer Perceptron

- **Each layer of MLP combines linear transformation and non-linearity:**

$$\boldsymbol{x}^{(l+1)} = \sigma(W_l \boldsymbol{x}^{(l)} + b^l)$$

  - where W_l is weight matrix that transforms hidden representation at layer l to layer $l+1$
  - b^l is bias at layer l, and is added to the linear transformation of $\boldsymbol{x}$
  - σ is non-linearity function (e.g., sigmod)
- Suppose $\boldsymbol{x}$ is 2-dimensional, with entries x_1 and x_2



Note: Adding the bias term provides more flexibility and better generalization to the neural network model

Training

- To ensure that the output is correct we must train the neural network.
- The network has a number of parameters: the weights and the biases.
- The behavior of the network changes according to their value.
- “Training” the network is the process of learning these values, such that the network performs it tasks correctly.
- The actual training process is an iterative process, in which we begin with initial estimates for all the parameters (which may be randomly set).
- These estimates are then iteratively refined such that the network outputs (mostly) correct answers to a set of “training” instances for which the answers are known.

Training Iterative Process

- In **Forward Propagation** each training instance x$_i$ is passed through the network to obtain the output y_i

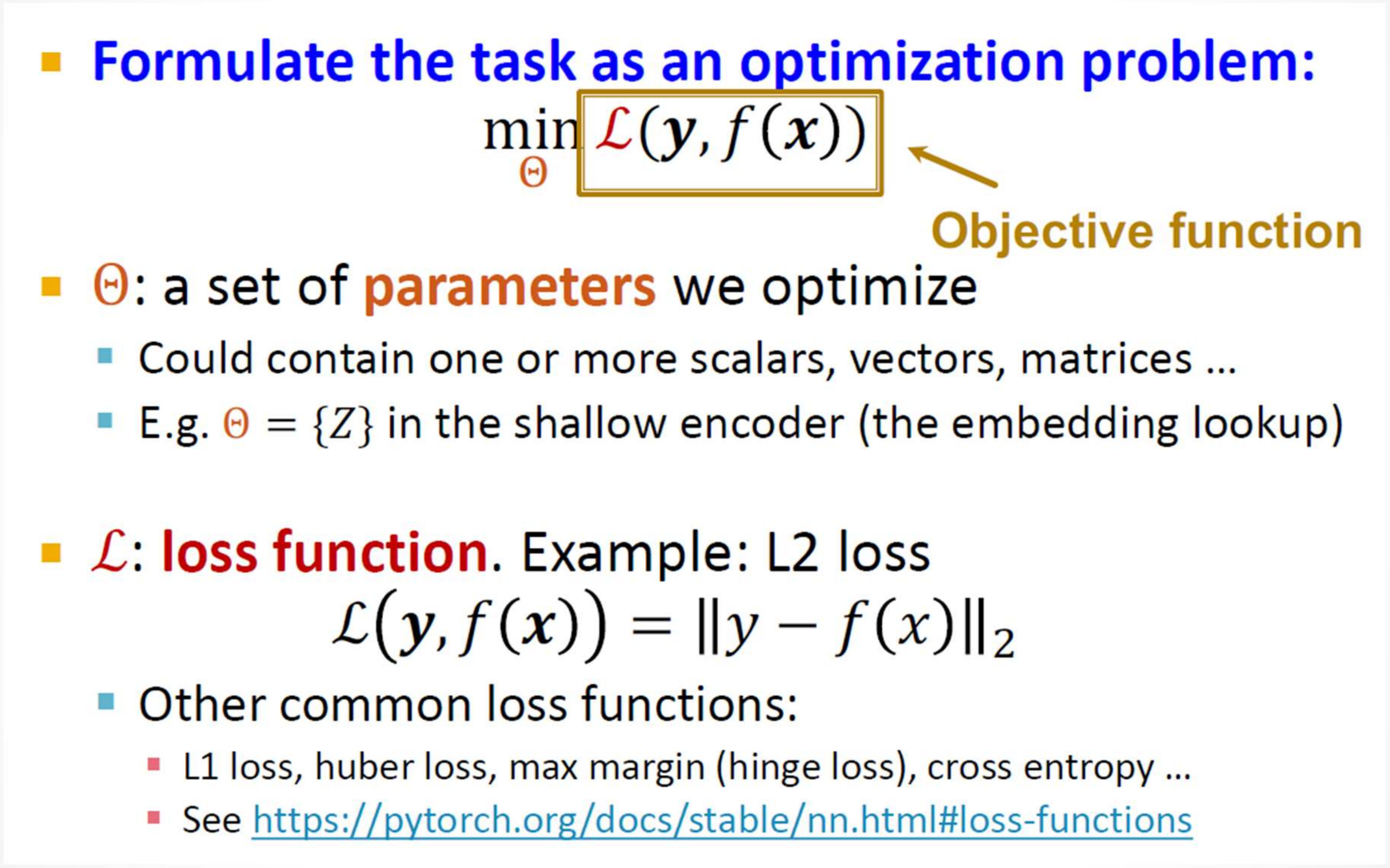
- A **loss function** is a function that compares the target and predicted output values
  - Measures how well the neural network models the training data

- **Back propagation** is a process involved in the training
  - For each training instance, we compute *backward* from the output layer of the network to the input layer
  - It involves taking the error rate of a forward propagation and feeding this loss backward through the neural network layers to fine-tune the weights.

Neural Network as Optimization

- **Supervised learning:** we are given input x, and the goal is to predict label y
- **Input x can be:**
  - Vectors of real numbers
  - Sequences (natural language)
  - Matrices (images)
  - Graphs (potentially with node and edge features)
- **We formulate the task as an optimization problem**

Neural Network as Optimization



- **Formulate the task as an optimization problem:**

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

Objective function

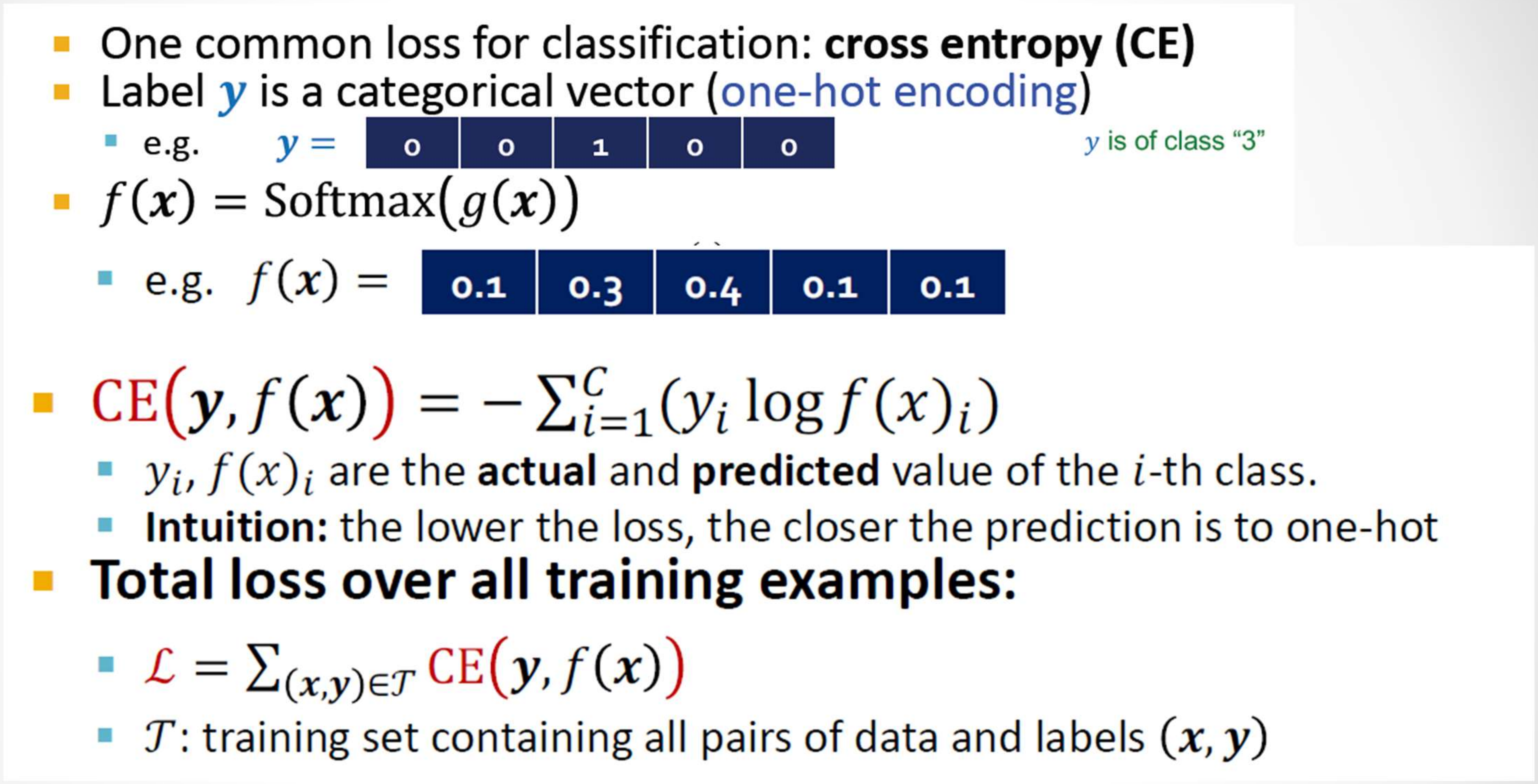
- Θ: a set of **parameters** we optimize
  - Could contain one or more scalars, vectors, matrices …
  - E.g. $\Theta = \{Z\}$ in the shallow encoder (the embedding lookup)

- $\mathcal{L}$: **loss function**. Example: L2 loss

$$\mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x})) = \|y - f(x)\|_2$$

  - Other common loss functions:
    - L1 loss, huber loss, max margin (hinge loss), cross entropy …
    - See https://pytorch.org/docs/stable/nn.html#loss-functions

Loss Function: Example



- One common loss for classification: **cross entropy (CE)**
- Label $\boldsymbol{y}$ is a categorical vector (one-hot encoding)
  - e.g. $\boldsymbol{y} =$

0	0	1	0	0

y is of class "3"

- $f(\boldsymbol{x}) = \mathrm{Softmax}(g(\boldsymbol{x}))$
  - e.g. $f(\boldsymbol{x}) =$

0.1	0.3	0.4	0.1	0.1

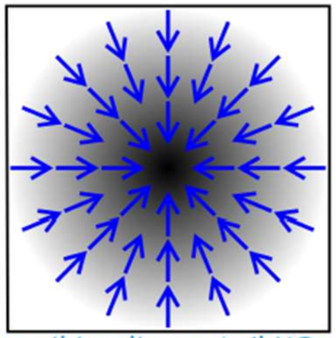
- $\mathrm{CE}(\boldsymbol{y}, f(\boldsymbol{x})) = -\sum_{i=1}^{C}(y_i \log f(x)_i)$
  - $y_i, f(x)_i$ are the **actual** and **predicted** value of the i-th class.
  - **Intuition:** the lower the loss, the closer the prediction is to one-hot
- **Total loss over all training examples:**
  - $\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\mathcal{T}} \mathrm{CE}(\boldsymbol{y}, f(\boldsymbol{x}))$
  - $\mathcal{T}$: training set containing all pairs of data and labels $(\boldsymbol{x}, \boldsymbol{y})$

Neural Network as Optimization

- **How to optimize the objective function?**
- **Gradient vector:** Direction and rate of fastest increase

Partial derivative

$$\nabla_{\Theta}\mathcal{L} = \left(\frac{\partial \mathcal{L}}{\partial \Theta_1}, \frac{\partial \mathcal{L}}{\partial \Theta_2}, \ldots\right)$$

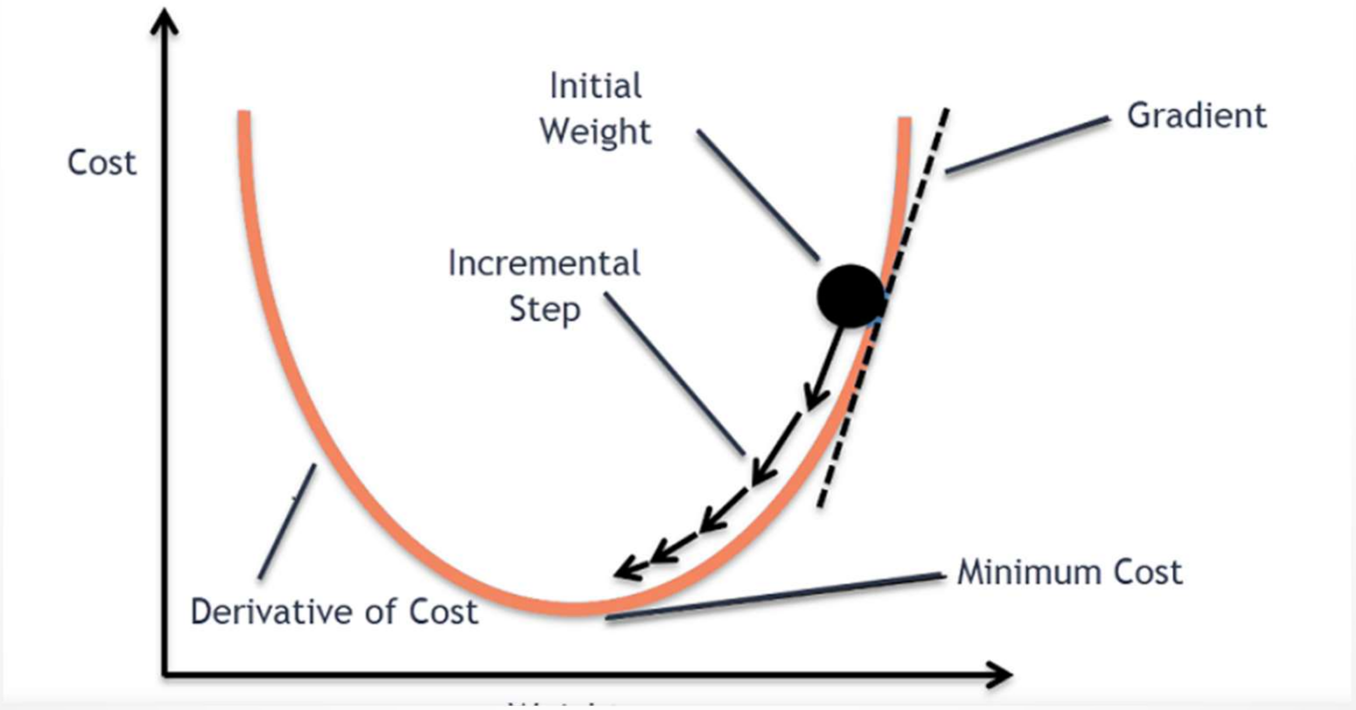


https://en.wikipedia.org/wiki/Gradient

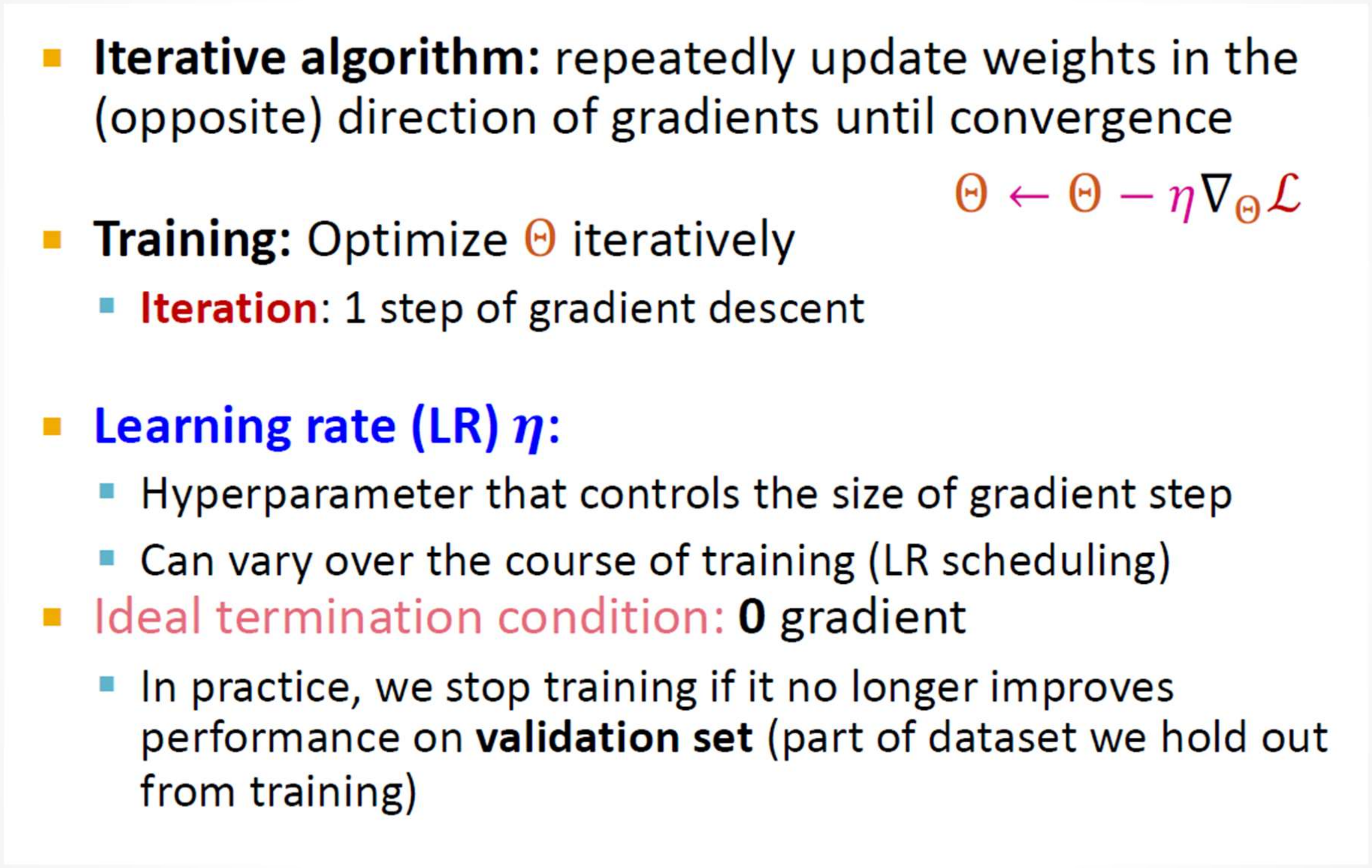
  - Θ_1, Θ_2 … : components of Θ
- Recall **directional derivative** of a multi-variable function (e.g. $\mathcal{L}$) along a given vector represents the instantaneous rate of change of the function along the vector.
- Gradient is the directional derivative in the **direction of largest increase**

Gradient Descent

- **Gradient descent** is an iterative first-order optimisation algorithm used to find a local minimum/maximum of a given function.
- Gradient Descent Algorithm iteratively calculates the next point using gradient at the current position, scales it (by a learning rate) and subtracts obtained value from the current position (makes a step).



Gradient Descent

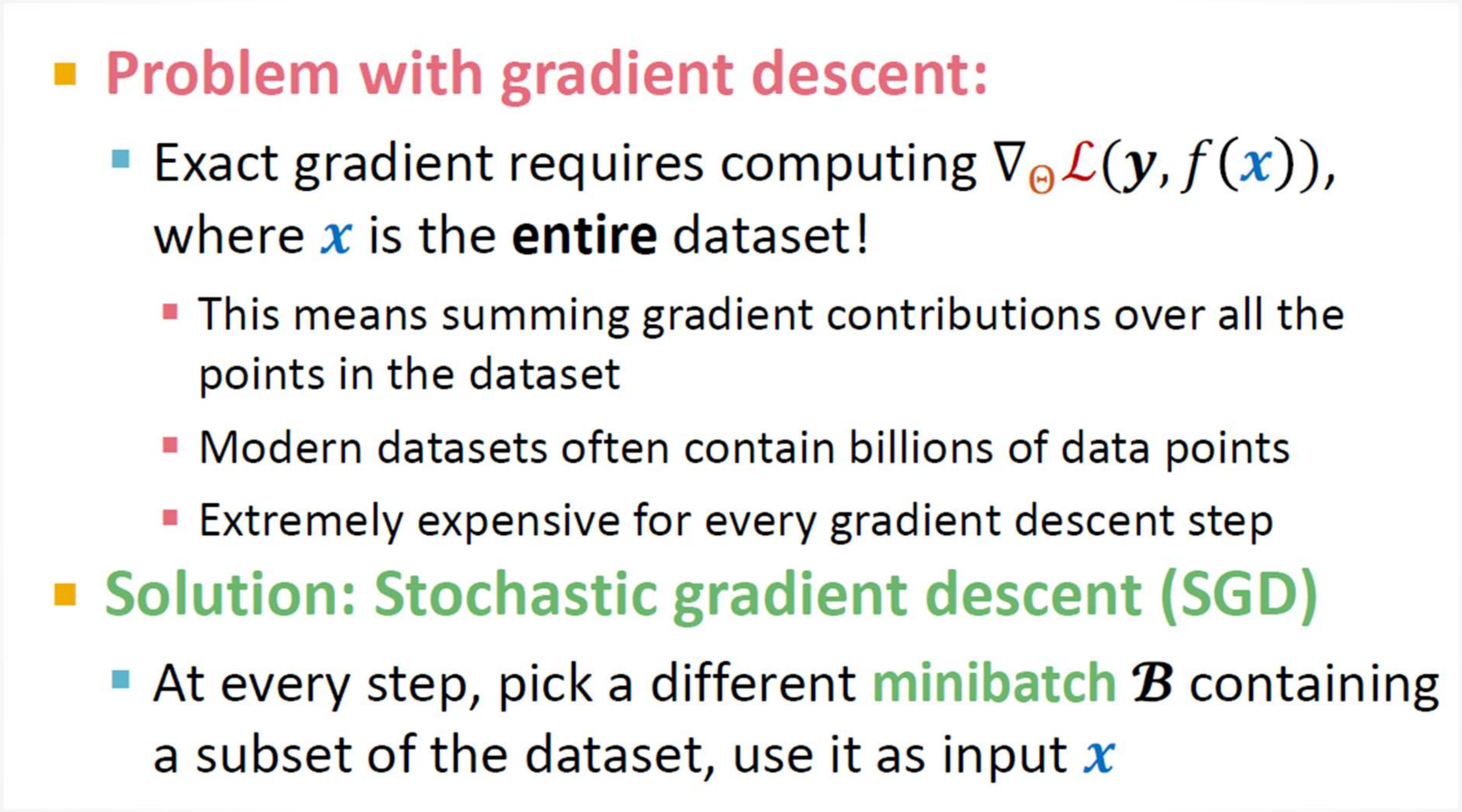


- **Iterative algorithm:** repeatedly update weights in the (opposite) direction of gradients until convergence

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

- **Training:** Optimize Θ iteratively
  - **Iteration**: 1 step of gradient descent
- **Learning rate (LR) η:**
  - Hyperparameter that controls the size of gradient step
  - Can vary over the course of training (LR scheduling)
- Ideal termination condition: **0** gradient
  - In practice, we stop training if it no longer improves performance on **validation set** (part of dataset we hold out from training)

Stochastic Gradient Descent

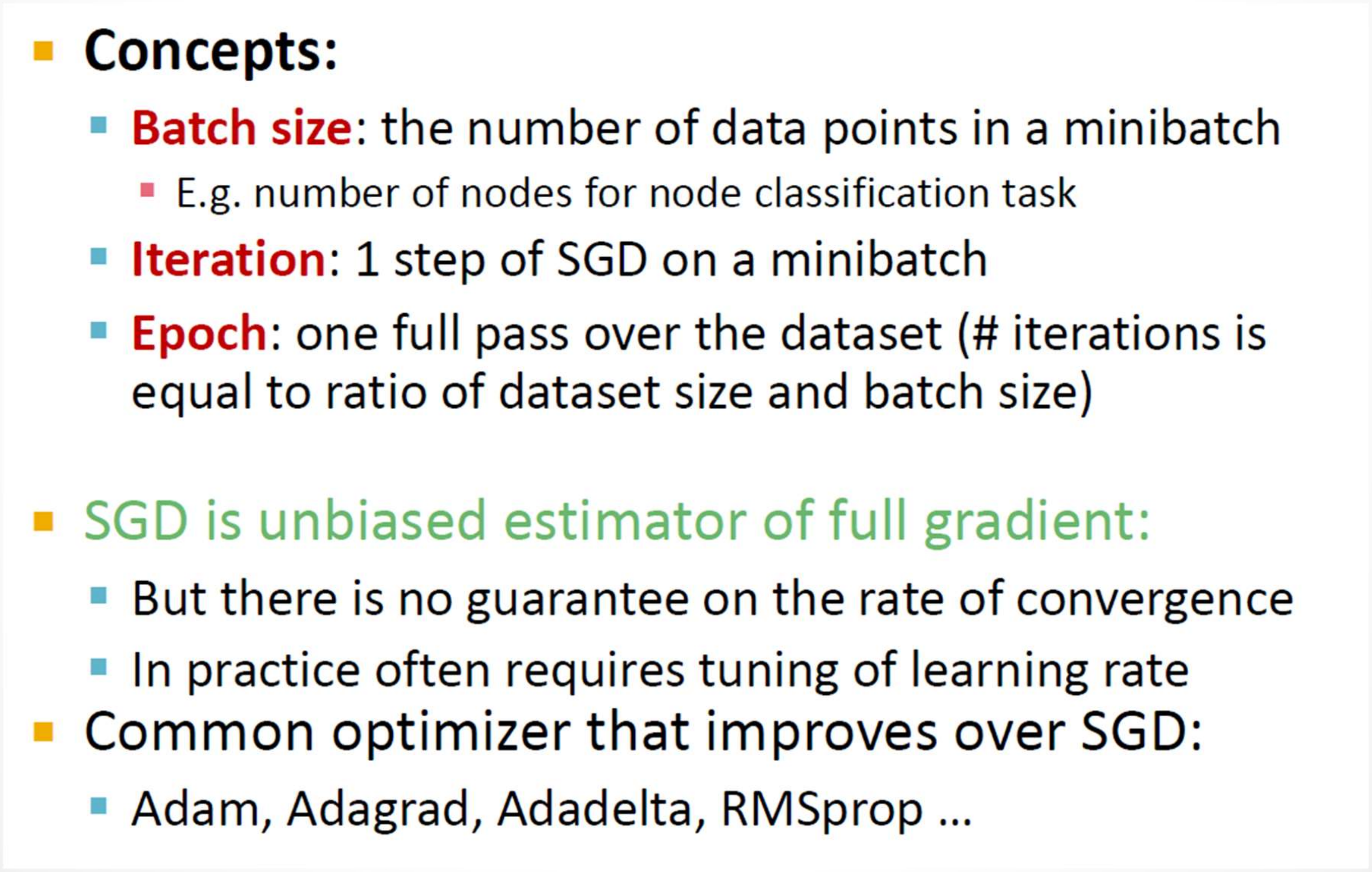


- **Problem with gradient descent:**
  - Exact gradient requires computing $\nabla_{\Theta}\mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$, where $\boldsymbol{x}$ is the **entire** dataset!
    - This means summing gradient contributions over all the points in the dataset
    - Modern datasets often contain billions of data points
    - Extremely expensive for every gradient descent step
- **Solution: Stochastic gradient descent (SGD)**
  - At every step, pick a different minibatch $\boldsymbol{\mathcal{B}}$ containing a subset of the dataset, use it as input $\boldsymbol{x}$

Stochastic Gradient Descent

- Stochastic Gradient Descent (SGD) is a variant of the Gradient Descent algorithm
- It addresses the computational inefficiency of traditional Gradient Descent methods when dealing with large datasets in machine learning projects.
- In SGD, instead of using the entire dataset for each iteration, only a single random training example (or a small batch) is selected to calculate the gradient and update the model parameters.
- This random selection introduces randomness into the optimization process, hence the term “stochastic” in stochastic Gradient Descent
- The advantage of using SGD is its computational efficiency, especially when dealing with large datasets.
- By using a single example or a small batch, the computational cost per iteration is significantly reduced compared to traditional Gradient Descent methods that require processing the entire dataset.

Minibatch SGD



- **Concepts:**
  - **Batch size**: the number of data points in a minibatch
    - E.g. number of nodes for node classification task
  - **Iteration**: 1 step of SGD on a minibatch
  - **Epoch**: one full pass over the dataset (# iterations is equal to ratio of dataset size and batch size)
- SGD is unbiased estimator of full gradient:
  - But there is no guarantee on the rate of convergence
  - In practice often requires tuning of learning rate
- Common optimizer that improves over SGD:
  - Adam, Adagrad, Adadelta, RMSprop …

Neural Network Function

- **Objective:** $\min_{\Theta} \mathcal{L}(\boldsymbol{y}, f(\boldsymbol{x}))$
- In deep learning, the function f can be very complex
- To start simple, consider linear function
$$f(\boldsymbol{x}) = W \cdot \boldsymbol{x}, \qquad \Theta = \{\mathrm{W}\}$$
- If f returns a scalar, then W is a learnable **vector**
$$\nabla_W f = \left(\frac{\partial f}{\partial w_1}, \frac{\partial f}{\partial w_2}, \frac{\partial f}{\partial w_3} \ldots\right)$$
- If f returns a vector, then W is the **weight matrix**
$$\nabla_W f = W^T$$
Jacobian matrix of f

Back Propagation

- How about a more complex function:

$$f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x}), \qquad \Theta = \{W_1, W_2\}$$

- Recall **chain rule**:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

In other words:

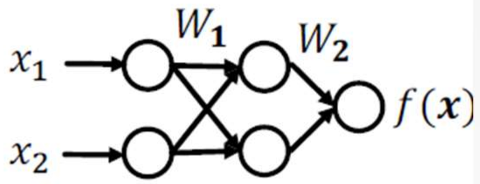
$$f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x})$$
$$h(x) = W_1\boldsymbol{x}$$
$$g(z) = W_2 z$$

- E.g. $\nabla_{\boldsymbol{x}} f = \frac{\partial f}{\partial(W_1\boldsymbol{x})} \cdot \frac{\partial(W_1\boldsymbol{x})}{\partial\boldsymbol{x}}$

- **Back-propagation**: Use of **chain rule** to propagate gradients of intermediate steps, and finally obtain gradient of $\mathcal{L}$ w.r.t. Θ

Back Propagation: Example

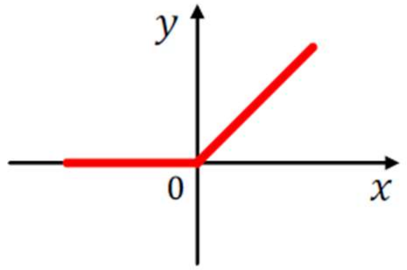
- **Example:** Simple 2-layer linear network
- $f(\boldsymbol{x}) = g(h(x)) = W_2(W_1\boldsymbol{x})$

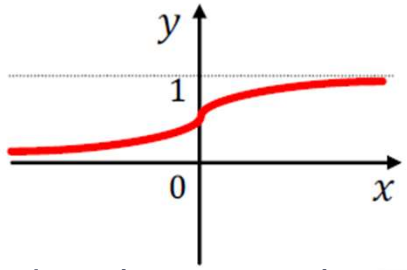


- $\mathcal{L} = \sum_{(\boldsymbol{x},\boldsymbol{y})\in\boldsymbol{\mathcal{B}}} \left\|(\boldsymbol{y}, -f(\boldsymbol{x}))\right\|_2$ sums the L2 loss in a minibatch $\boldsymbol{\mathcal{B}}$
- **Hidden layer:** intermediate representation for input $\boldsymbol{x}$
  - Here we use $h(x) = W_1\boldsymbol{x}$ to denote the hidden layer
  - $f(\boldsymbol{x}) = W_2 h(x)$

Non-linearity

- Note that in $f(\boldsymbol{x}) = W_2(W_1\boldsymbol{x})$, W_2W_1 is another matrix (vector, if we do binary classification)
- Hence $f(\boldsymbol{x})$ is still linear w.r.t. $\boldsymbol{x}$ no matter how many weight matrices we compose
- **Introduce non-linearity:**
  - **Rectified linear unit (ReLU)**
    $ReLU(x) = \max(x, 0)$
  - **Sigmoid**
    $$\sigma(x) = \frac{1}{1 + e^{-x}}$$





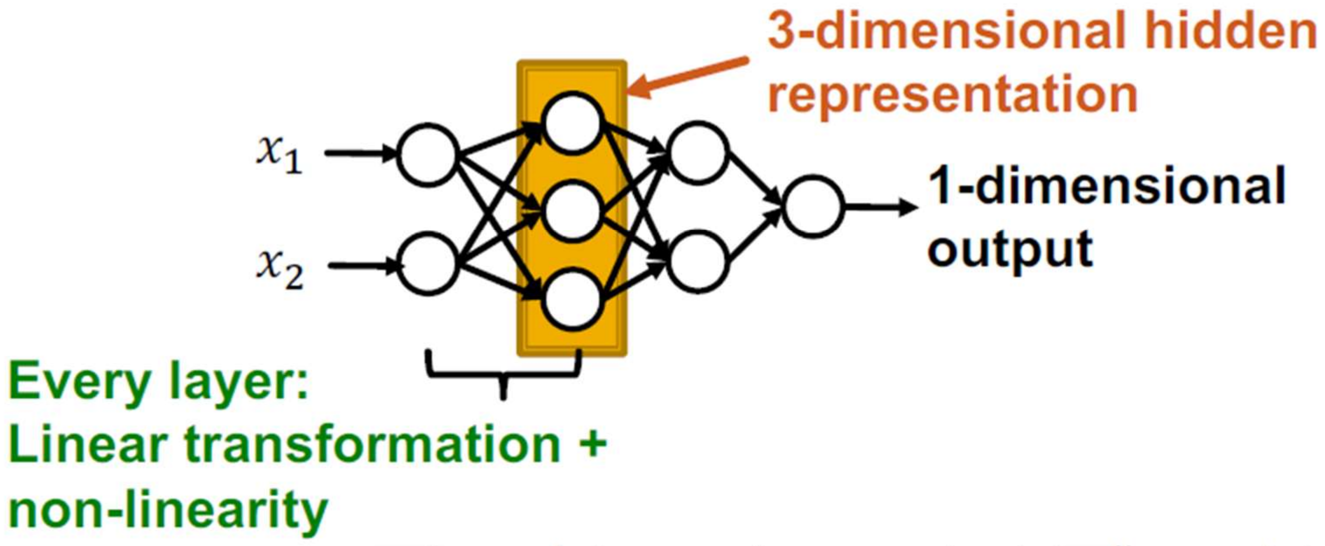
This nonlinearity allows neural networks to develop complex representations and functions based on the inputs that would not be possible with a simple linear model.

Multi-layer Perceptron (MLP)

- **Each layer of MLP combines linear transformation and non-linearity:**

$$\boldsymbol{x}^{(l+1)} = \sigma(W_l \boldsymbol{x}^{(l)} + b^l)$$

  - where W_l is weight matrix that transforms hidden representation at layer l to layer $l+1$
  - b^l is bias at layer l, and is added to the linear transformation of $\boldsymbol{x}$
  - σ is non-linearity function (e.g., sigmod)
- Suppose $\boldsymbol{x}$ is 2-dimensional, with entries x_1 and x_2



Summary

- **Objective function:**

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

- f can be a simple linear layer, an MLP, or other neural networks (e.g., a GNN later)
- Sample a minibatch of input $\boldsymbol{x}$
- **Forward propagation:** compute $\mathcal{L}$ given $\boldsymbol{x}$
- **Back-propagation:** obtain gradient $\nabla_{\Theta}\mathcal{L}$ using a chain rule
- Use **stochastic gradient descent (SGD)** to optimize for Θ over many iterations

Limitations - MLP

- When you have data that doesn’t neatly align into columns
  - Images that you want to find features within
  - Machine translation
  - Sentence classification
  - Graphs