# Introduction to Deep Learning

## Tutorial 2

Gabriel Deza
Department of Industrial Engineering
Tel Aviv University

November 11th & 13th, 2025

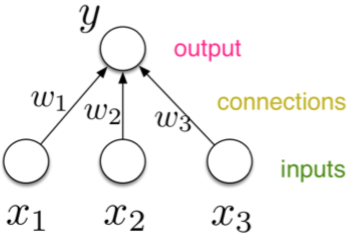
# Overview

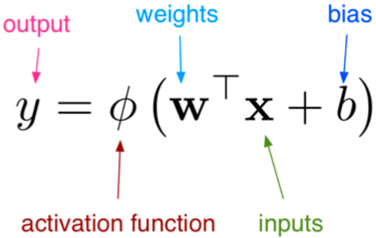
**Last tutorial**: review of linear models, gradient descent, PyTorch.

**Today**:

- Multilayer Perceptron
- Activation functions
- Backprop
- Regularization techniques

# Limits of linear models



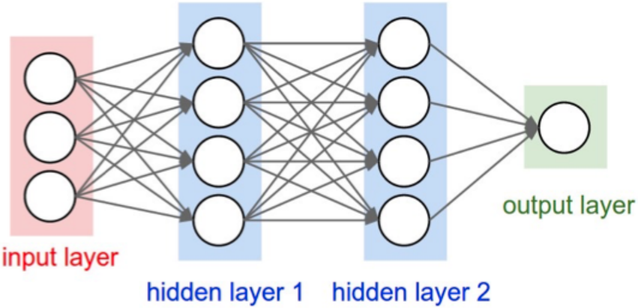




Can view logistic and linear regression as a single unit.

In lecture, we saw it was not possible to model XOR as it is not linearly separable.

Connecting many such units in a network $\rightarrow$ more expressive power.

# Multilayer Perceptron



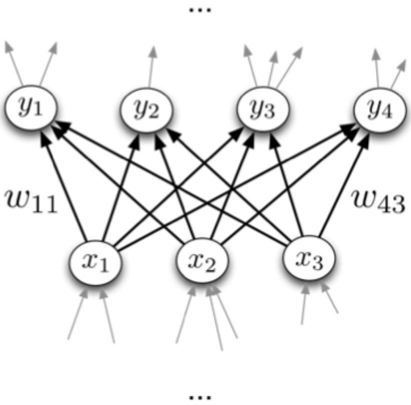

- Directed acylic graph (DAG)
- Also known as feed forward neural network (no cycles)
- Layers:
  - 1 input layer to feed in input features.
  - *h* hidden layers to compute advanced features.
  - 1 output layer to produce interpretable output.

# MLP

- Each layer connects $N$ input units to $M$ output units. When all input units are connected to output units $\rightarrow$ Fully connected layer.
- Inputs/outputs of a layer is not the same as inputs/outputs to the network.

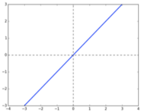
$$\boldsymbol{y} = f(\boldsymbol{x}) = \phi(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b})$$

- Why do we need activation functions $\phi$? What if $\phi(x) = x$?
- Multilayer feed-forward neural nets with nonlinear activation functions are universal approximators: they can approximate any function arbitrarily well.



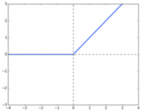
# Activation Functions

- Non-linear activation functions allow us to go beyond an affine transform.
- Multilayer feed-forward neural nets with nonlinear activation functions are universal approximators: they can approximate any continuous function arbitrarily well.



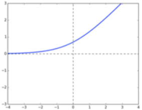
**Linear**

$y = z$



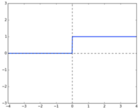
**Rectified Linear Unit (ReLU)**

$y = \max(0, z)$



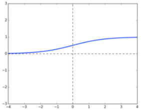
**Soft ReLU**

$y = \log 1 + e^z$



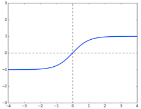
**Hard Threshold**

$$y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$



**Logistic**

$$y = \frac{1}{1 + e^{-z}}$$



**Hyperbolic Tangent (tanh)**

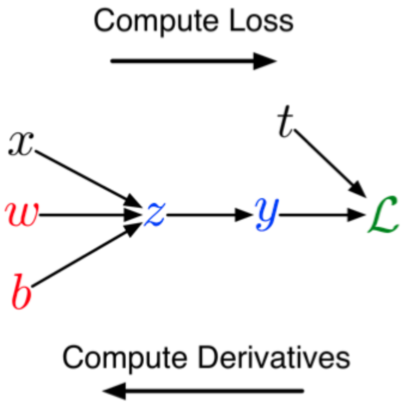
$$y = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

# Backpropagation

- Gradient descent (GD) updates parameters using the cost gradient:

$$\mathbf{W} \leftarrow \mathbf{W} - \alpha \nabla \mathcal{J}(\mathbf{W})$$

- The cost gradient is the average of $\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\mathbf{w}}$ across all training examples.
- Backpropagation efficiently computes these gradients in neural networks. It relies on:
    - Chain rule
    - Computation graphs



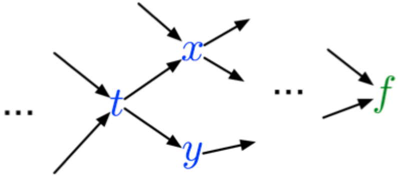
# Backpropagation

- Suppose we have a function $f(x, y)$ and functions $x(t)$ and $y(t)$. (All the variables here are scalar-valued.) Then

Mathematical expressions to be evaluated

$$\frac{\mathrm{d}f}{\mathrm{d}t} = \frac{\partial f}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial f}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

Values already computed by our program



# Backpropagation

**Full backpropagation algorithm:**
Let $v_1, \ldots, v_N$ be a topological ordering of the computation graph (i.e. parents come before children).
$v_N$ denotes the variable we're trying to compute derivatives of (e.g. the loss).

**forward pass**

$$\text{For } i = 1, \ldots, N: \quad \text{Compute } v_i \text{ as a function of } \mathrm{Pa}(v_i)$$

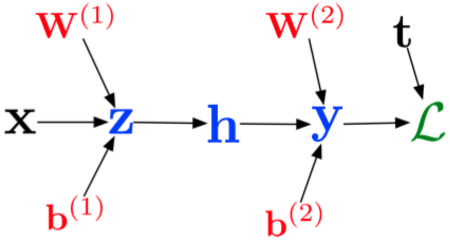
**backward pass**

$$\bar{v}_N = 1$$

$$\text{For } i = N-1, \ldots, 1: \quad \bar{v}_i = \sum_{j \in \mathrm{Ch}(v_i)} \bar{v}_j \frac{\partial v_j}{\partial v_i}$$

# MLP Example

**MLP example in vectorized form:**




**Forward pass:**

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\mathbf{h} = \sigma(\mathbf{z})$$
$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$
$$\mathcal{L} = \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2$$

**Backward pass:**

$$\overline{\mathcal{L}} = 1$$
$$\overline{\mathbf{y}} = \overline{\mathcal{L}}(\mathbf{y} - \mathbf{t})$$
$$\overline{\mathbf{W}^{(2)}} = \overline{\mathbf{y}}\mathbf{h}^\top$$
$$\overline{\mathbf{b}^{(2)}} = \overline{\mathbf{y}}$$
$$\overline{\mathbf{h}} = (\mathbf{W}^{(2)})^\top\overline{\mathbf{y}}$$
$$\overline{\mathbf{z}} = \overline{\mathbf{h}} \odot \sigma'(\mathbf{z})$$
$$\overline{\mathbf{W}^{(1)}} = \overline{\mathbf{z}}\mathbf{x}^\top$$
$$\overline{\mathbf{b}^{(1)}} = \overline{\mathbf{z}}$$

# Putting it all together

- Initialize model parameters $\boldsymbol{W}$ randomly*
- **Batch Gradient Descent (GD):**
  - At iteration $k$, compute the cost (average loss) over the entire dataset.

$$\boldsymbol{W}^{(k+1)} = \boldsymbol{W}^{(k)} - \alpha \nabla \mathcal{J}(\boldsymbol{W}^{(k)})$$

- **Stochastic Gradient Descent (SGD):**
  - Randomly select a mini-batch (small subset of the training data).
  - At iteration $k$, perform GD update with cost averaged over the mini-batch. Update.
  - **Intuition:** noisy approximation of the full gradient.
  - **Tip:** subsample without replacement.
  - **Epoch:** enough iterations to cover the entire dataset once.

* In practice, use smarter initialization (e.g. Xavier, He).

# Code: Training MLP using PyTorch

Notebook time :)

https://colab.research.google.com/drive/1DaOKVM_t5S4zXCHY--_OHFAdqbye2IwN?us