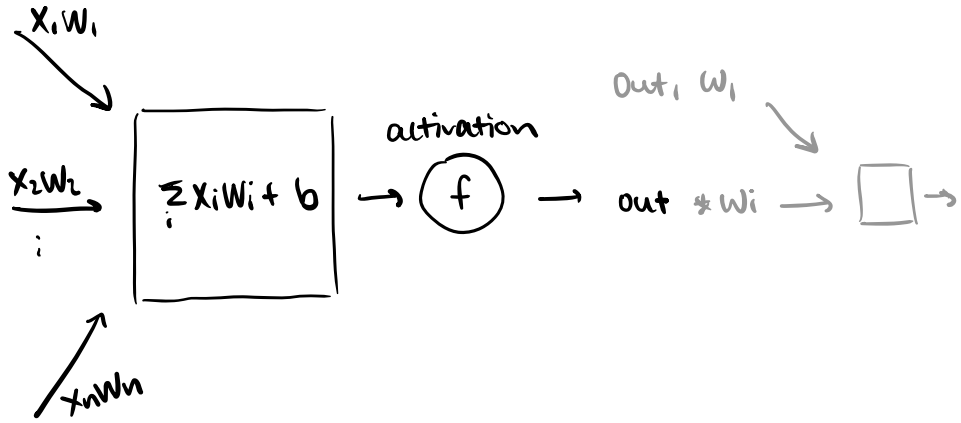
Neuron



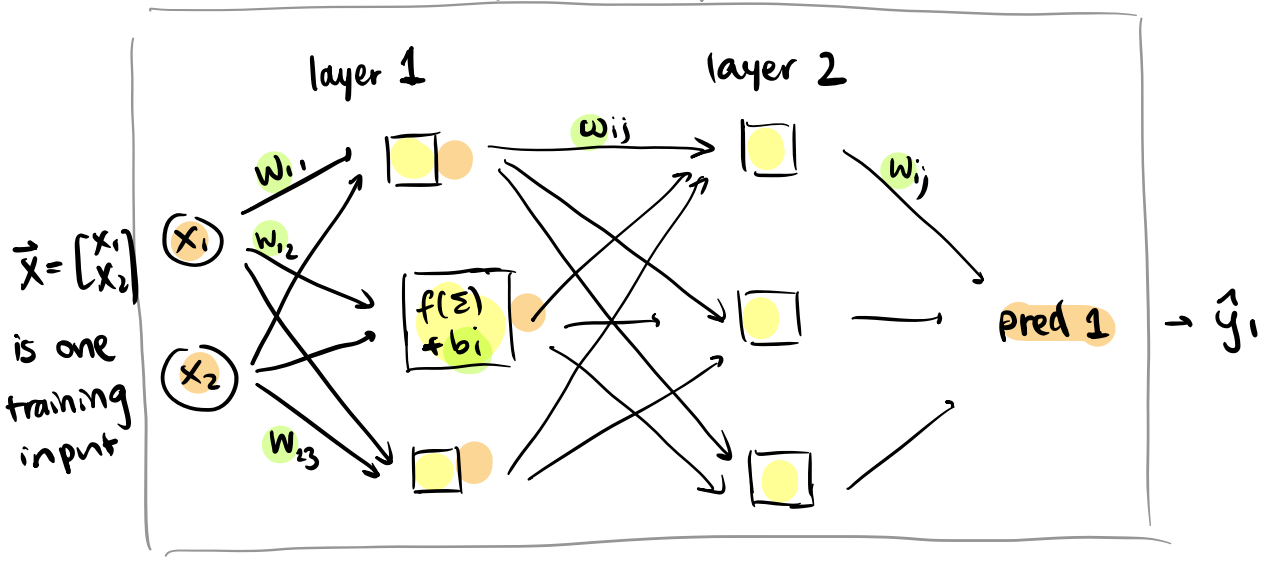
w_i, b are initiated to be a random value in $[-1, 1]$

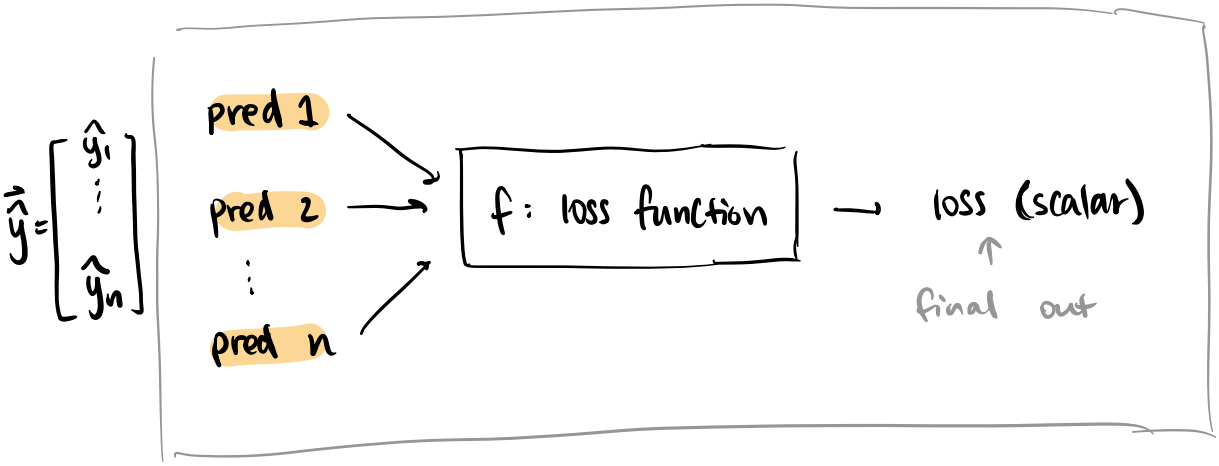
n_{in} = dimension of x_i and w_i

A layer

- a list of neurons
- # neurons we have = n_{out}
- Evaluate each neuron independently
- n_{in}, n_{out} = # neuron in this & next layer

Multi layer Perceptron (MLP)





- all w_{ij} are different and initialized between $[-1, 1]$
- x_i are scalars. Input $\vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix}$
- each layer has an output that's used as inputs for the next layer.
- loss function compares $\vec{\hat{y}}$ with $\vec{y}$ and ensures loss is always positive.
- Backward prop starts at the very end (loss)

What do neuro-nets do?

- When fed in training data pairs $\vec{x}$, y, we want to find weights (all W_{ij}) that help us predict y given $\vec{x}$
- Training input = a list of $\vec{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_i \\ \vdots \\ x_n \end{bmatrix}$ $\Rightarrow$ a matrix
- Each $\vec{x}$ have a different set of weights & biases, all of these are parameters we optimize over! (so we'd find gradients for all of them & step)

Loss Function

- $L = \Sigma \left[f(y_{predicted}, y_{actual}) \text{ for all } y \right]$
  f should return positive value only!
- L is where we start the backpropagation, we want the gradients of all weights and biases
- weights & biases are parameters we tune. Store them in one array.
- We want to minimize lost so we move a small step opposite the direction of the gradient and iterate the process. step size = learning rate

- **Gradient Descent**: Forward pass, backward pass, update parameters