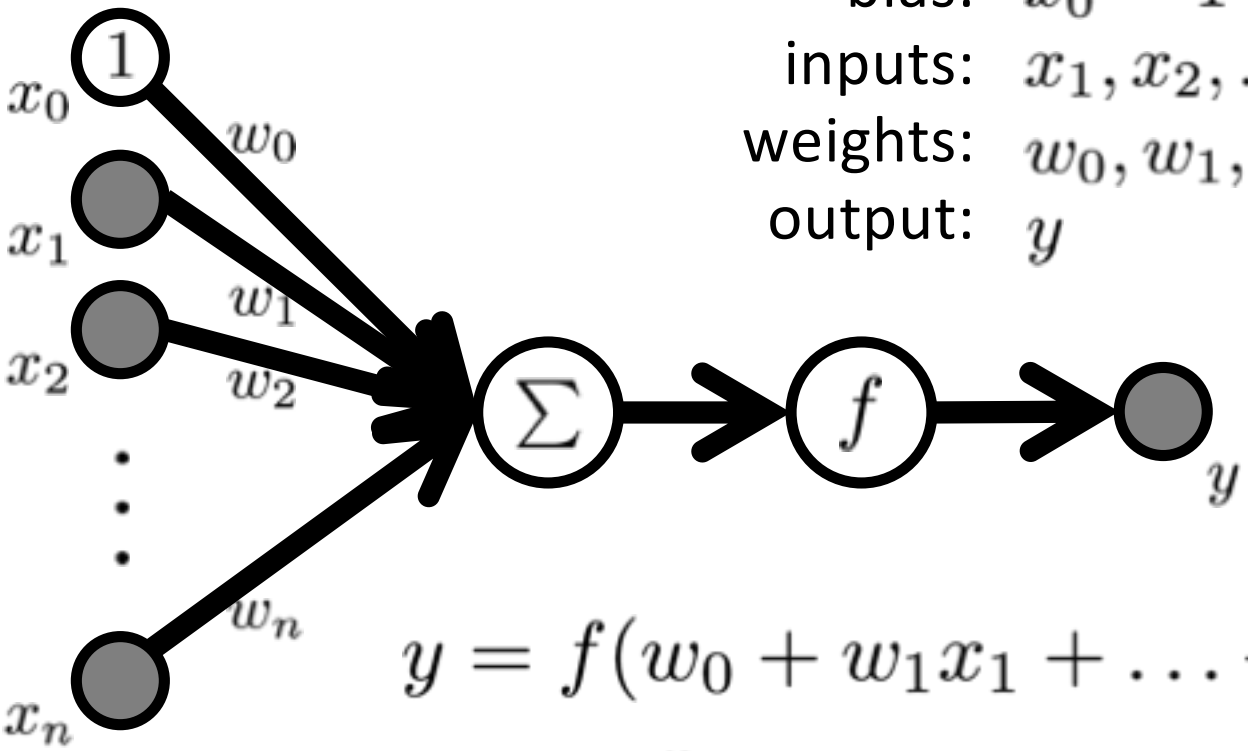
深度学习

第四讲

王胤

Recap: A Single Neuron

bias: $x_0 = 1$
inputs: $x_1, x_2, \ldots, x_n$
weights: $w_0, w_1, \ldots, w_n$
output: y



$y = ?$

$f(\Sigma wx)$

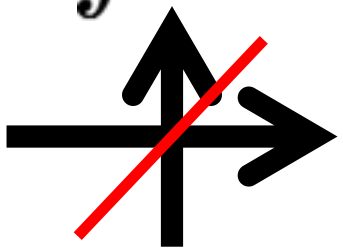
$$y = f(w_0 + w_1 x_1 + \ldots + w_n x_n)$$

$$y = f\left(\sum_{i=0}^{n} w_i x_i\right)$$

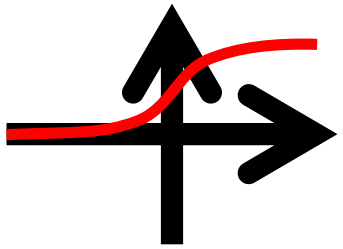
$$y = f(\mathbf{w}^\mathbf{T}\mathbf{x})$$

Recap: Activation function: f

$$y = f(\sum_{i=0}^{n} w_i x_i)$$



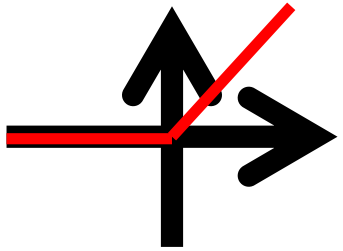
linear: $f(x) = x$



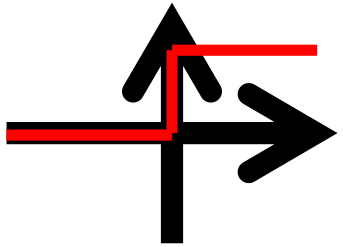
sigmoid: $f(x) = \frac{1}{1+e^{-x}}$

ReLU: $f(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$

Rectified Linear Unit



unit step: $f(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$



Recap: Objective function

How do we measure our error?

training example: $\{\mathbf{x}_j, t_j\}$

target output: t_j

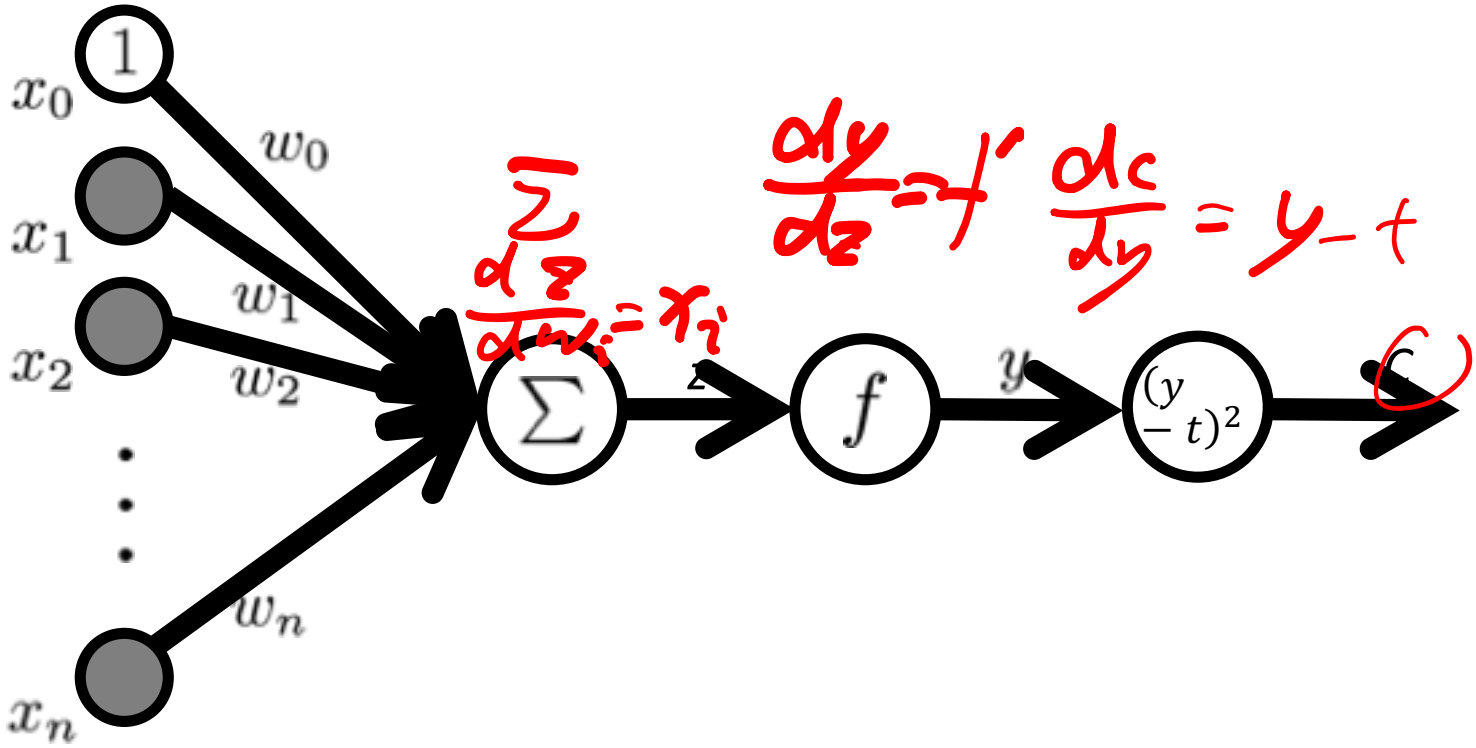
network parameters: ω

$$\min_{\omega} \sum_{j} (y_j - t_j)^2$$

Also called the error or loss function.

Recap: Chain Rule

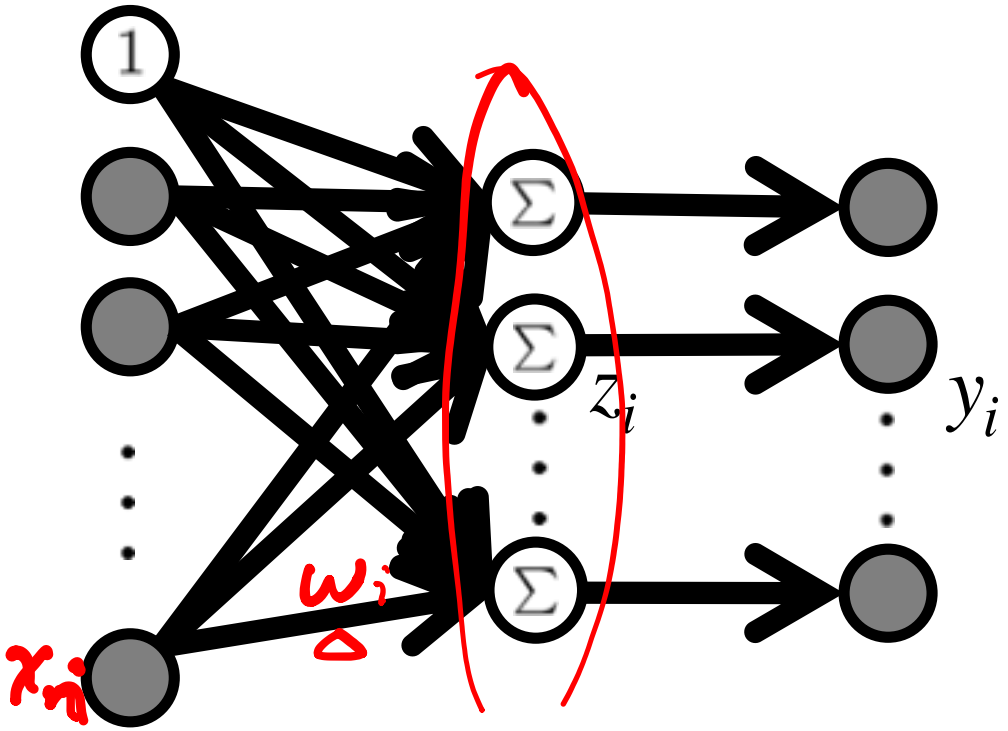
$$\Delta w_i = \alpha (t_j - y_j) f'(net_j) x_{ij}$$



Recap: Function derivatives

		Derivatives
linear:	$f(x) = x$	$f'(x) =$ 1
sigmoid:	$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) =$ $f(x)(1-f(x))$
ReLU:	$f(x) = \begin{cases} x & x \geq 0 \\ 0 & x < 0 \end{cases}$	$f'(x) =$ $\begin{cases} 1 \\ 0 \end{cases}$
unit step:	$f(x) = \begin{cases} 1 & x > 0 \\ 0 & x \leq 0 \end{cases}$	$f'(x) =$ 0

Recap: Softmax



$$y_i = \frac{e^{z_i}}{\sum_{j \in group} e^{z_j}}$$

$$\frac{\partial y_i}{\partial z_i} = y_i (1 - y_i)$$

$$C = -\sum_j t_j \log y_j$$

target value

$$\Delta w_i = \alpha (t_j - y_j) f'(net_j) x_{ij}$$

$$\frac{\partial C}{\partial z_i} =$$

Linear perceptron

- Guaranteed to converge to the minimum square error.
- Given a sufficiently small learning rate.
- Direct solutions exists (no need to iteratively compute solution).

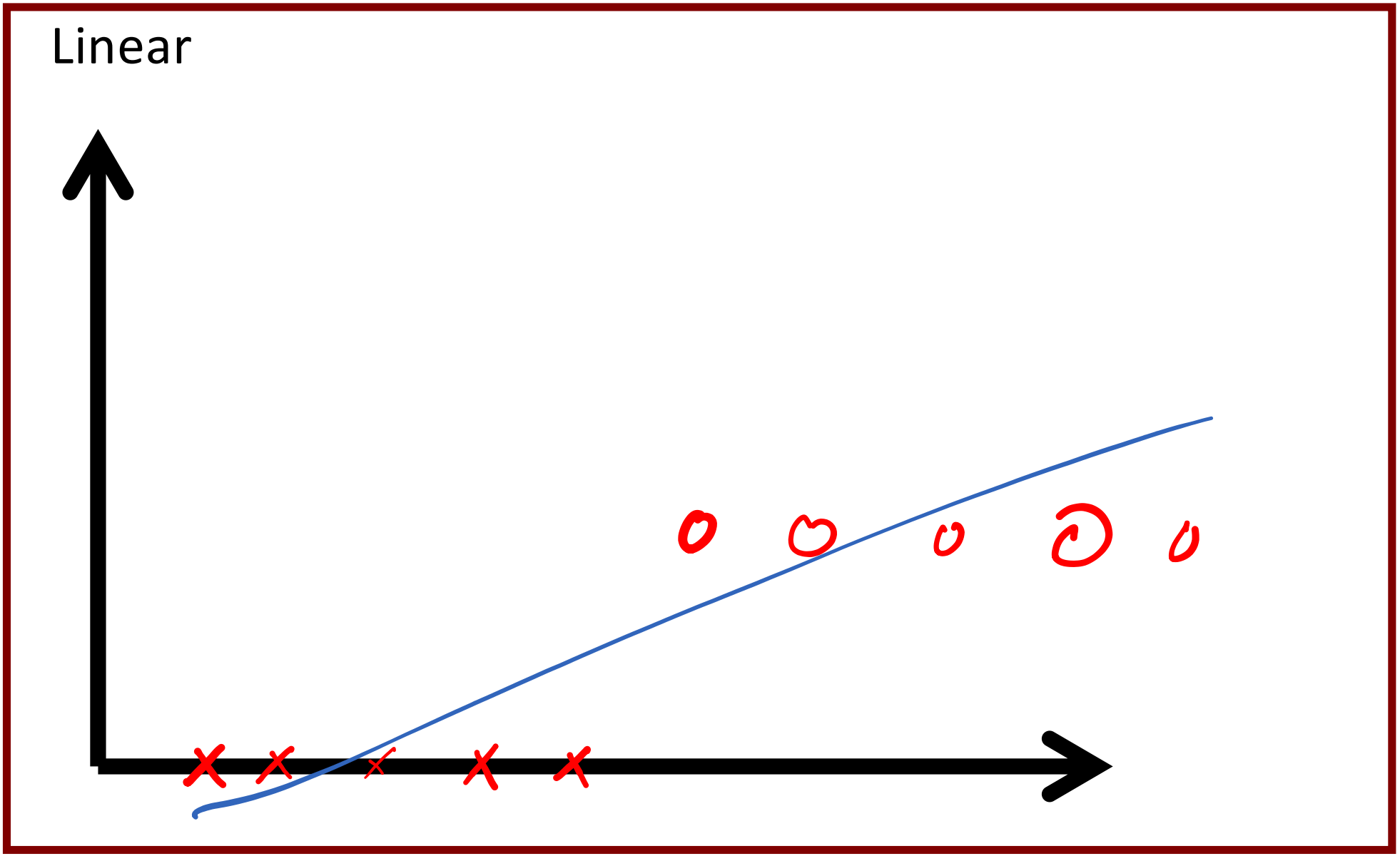
$$f(x) = x$$

$$\Delta w_i = \alpha(t_j - y_j) f'(net_j) x_{ij}$$

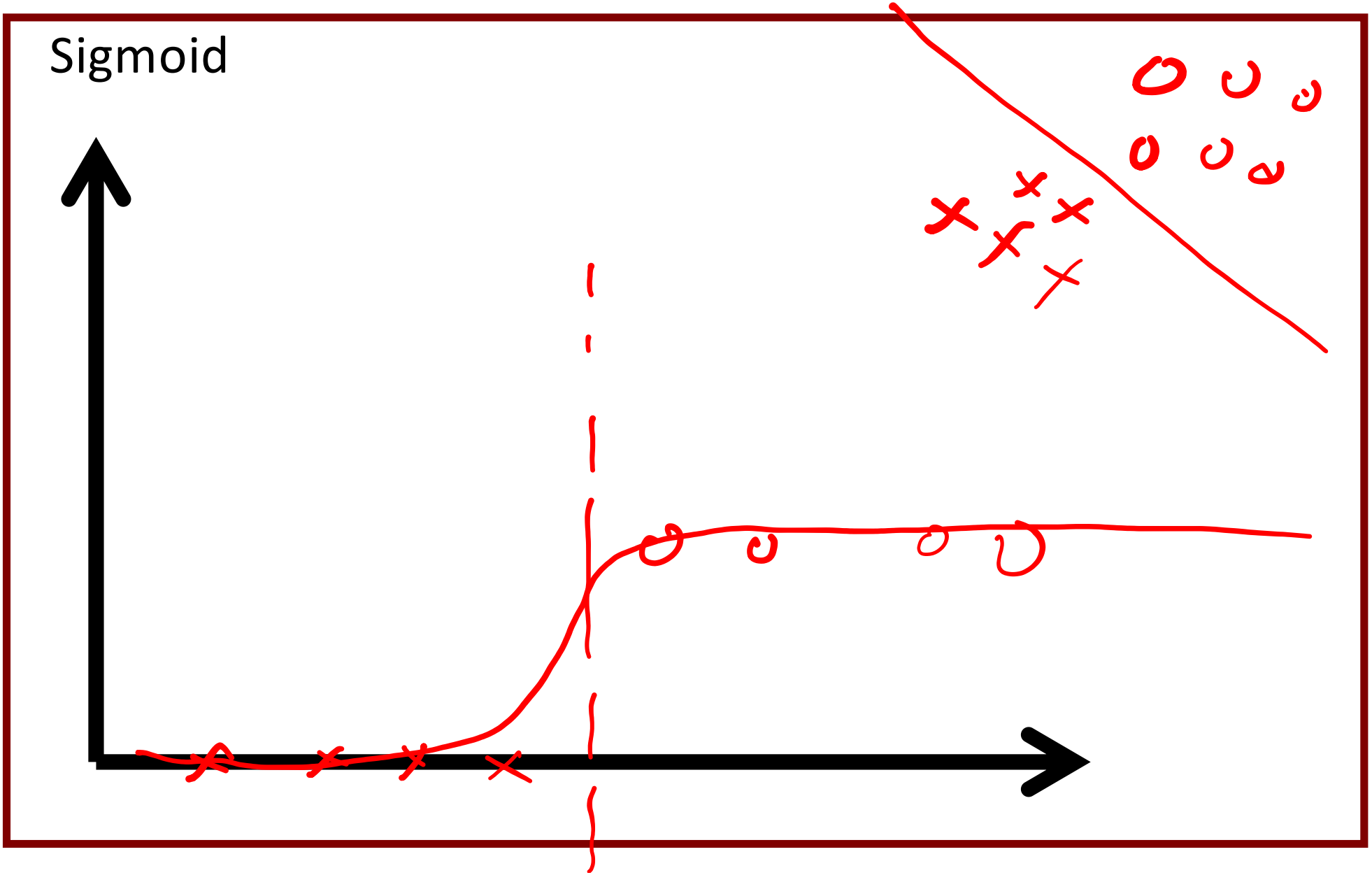
$$f'(net_j) = 1$$

$$\Delta w_i = \alpha(t_j - y_j) x_{ij}$$

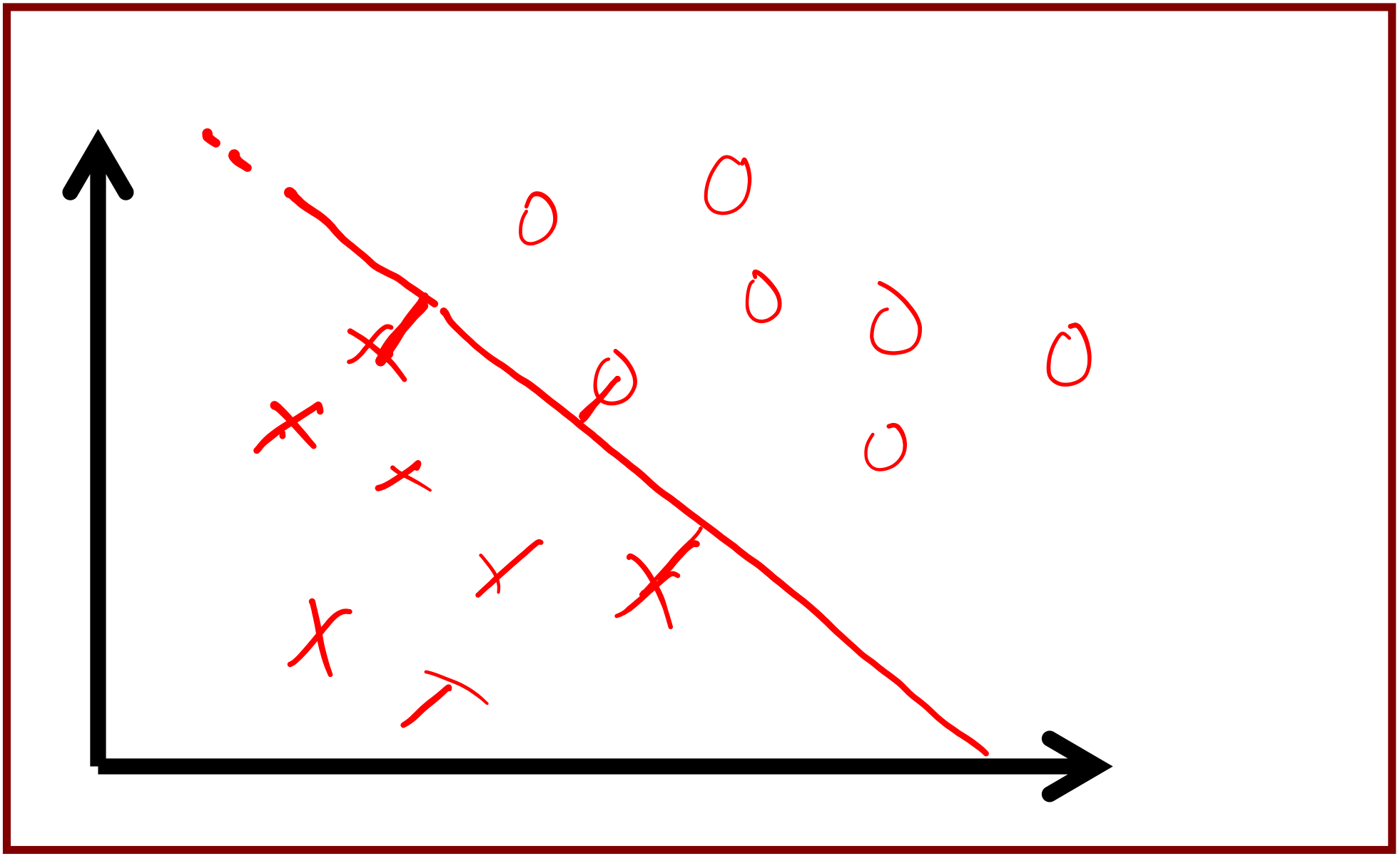
Effect of activation function



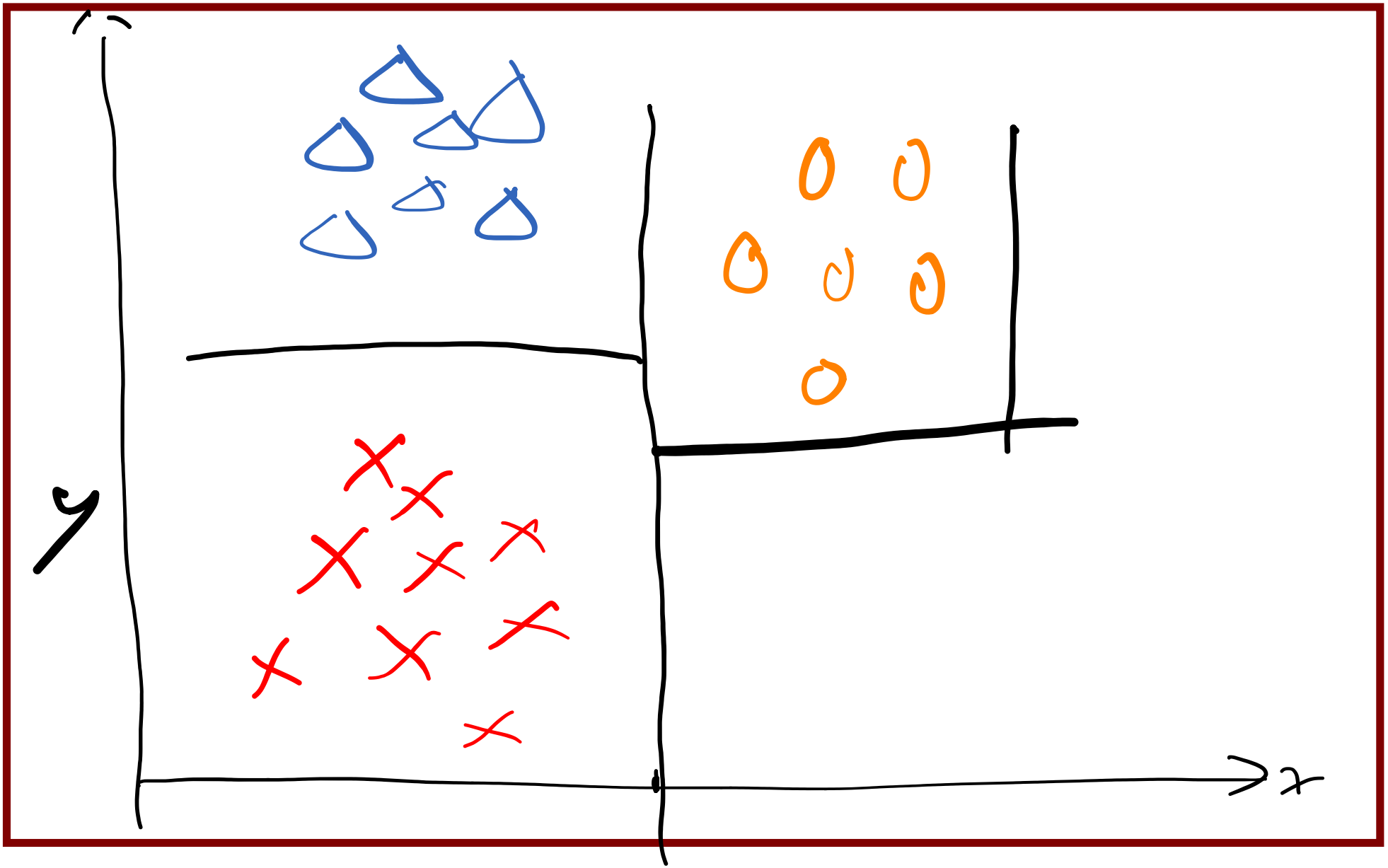
Effect of activation function



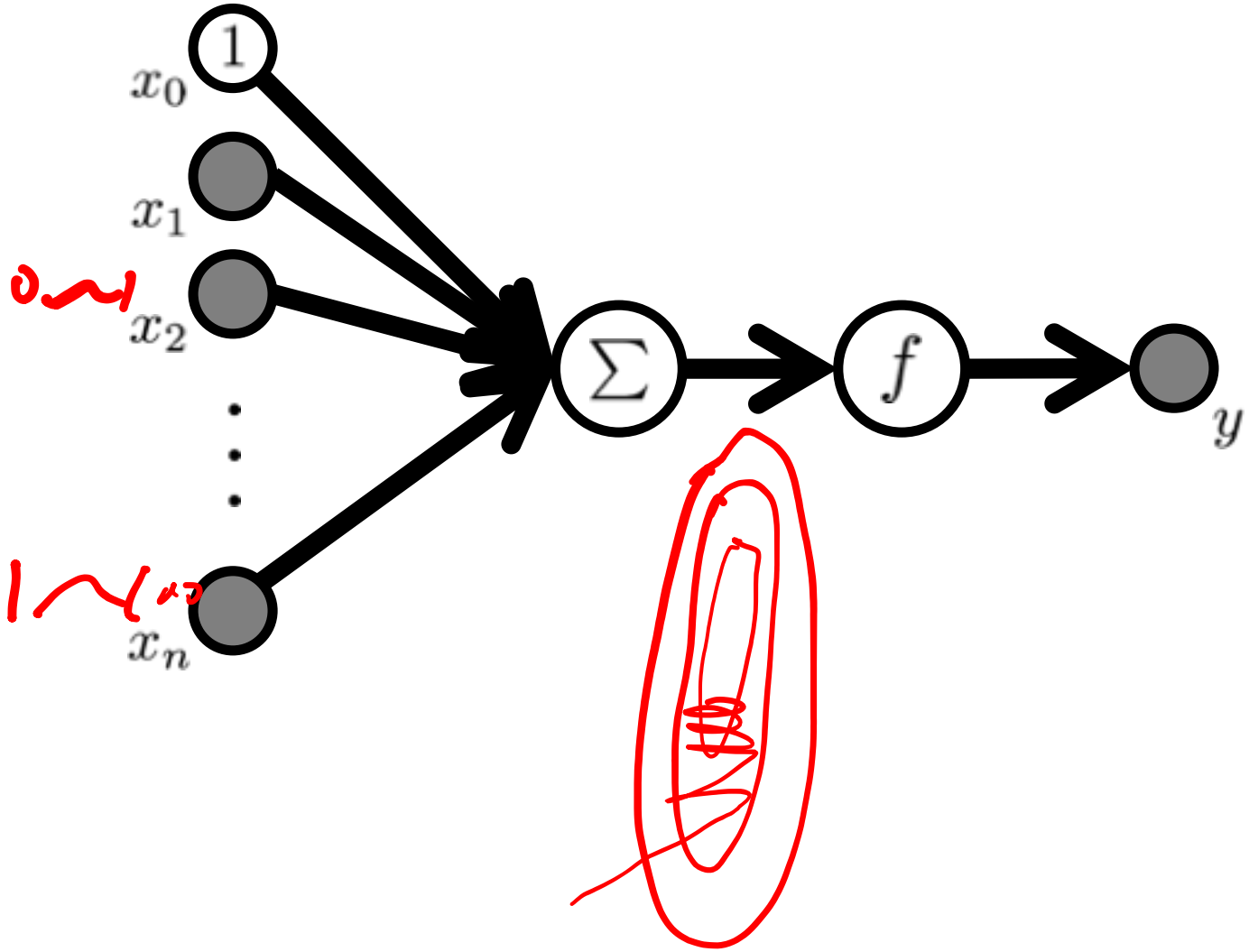
SVM (Support vector machine)



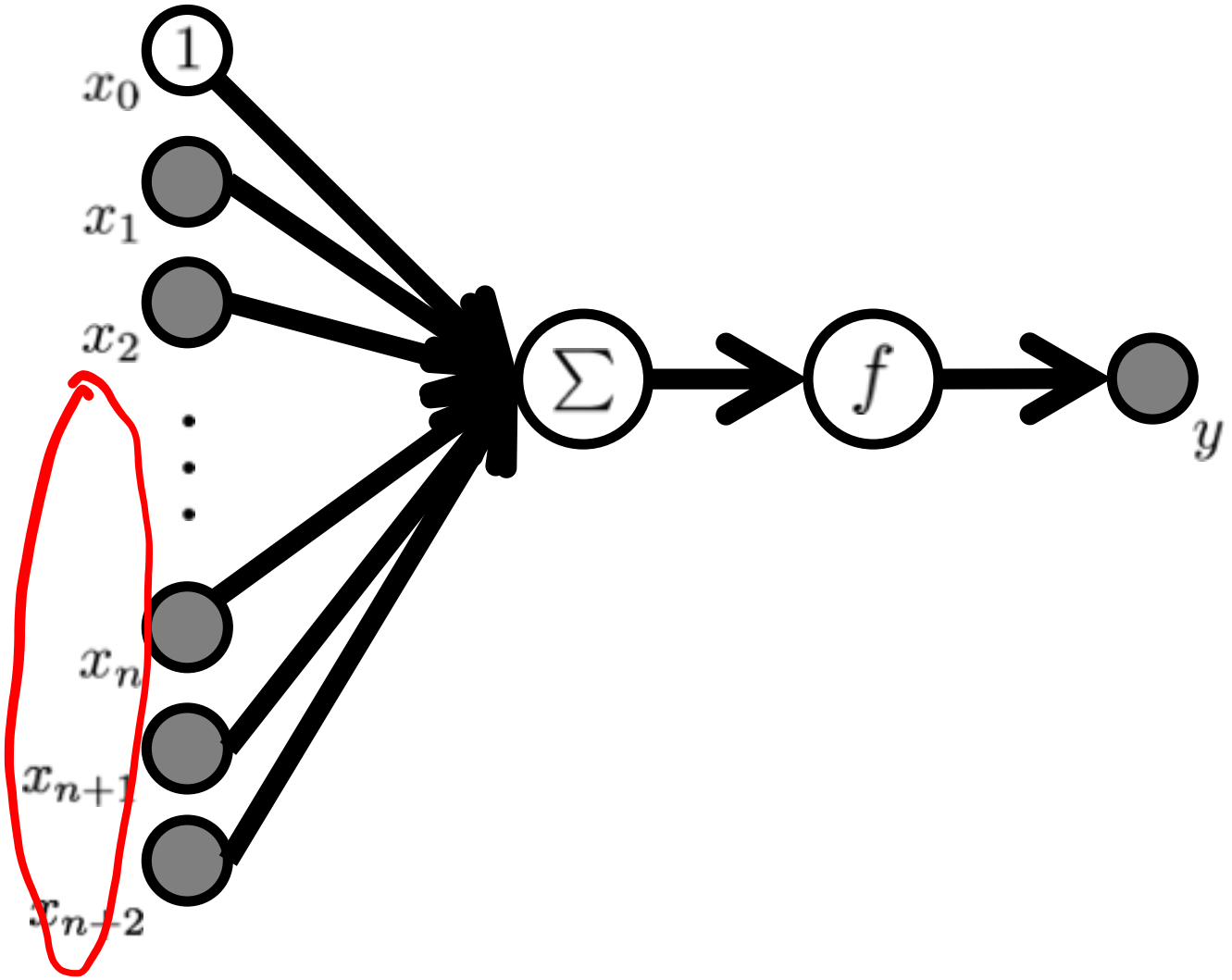
Decision tree



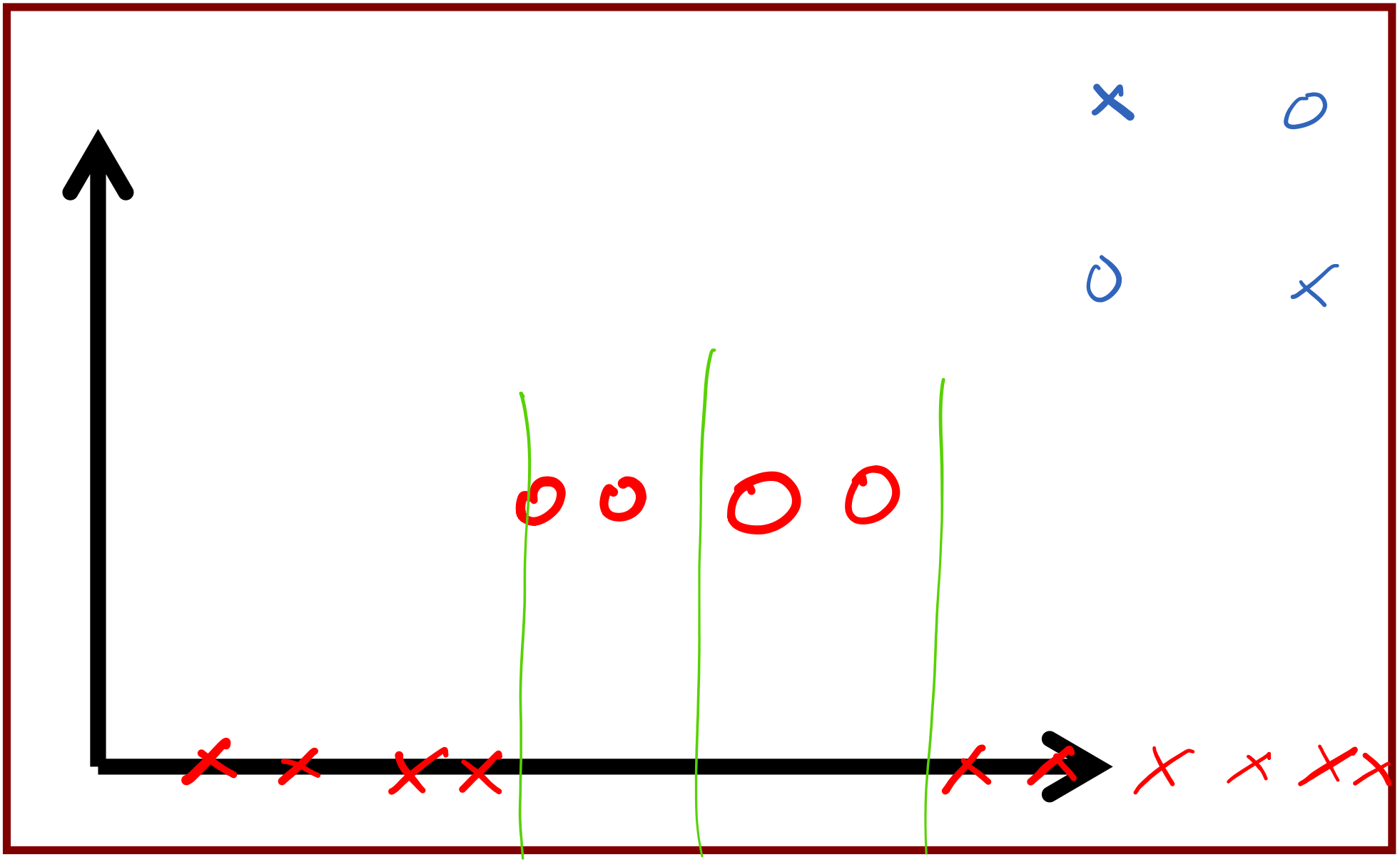
Inputs: Variance



Inputs: Redundancy



Limits of perceptrons (XOR)



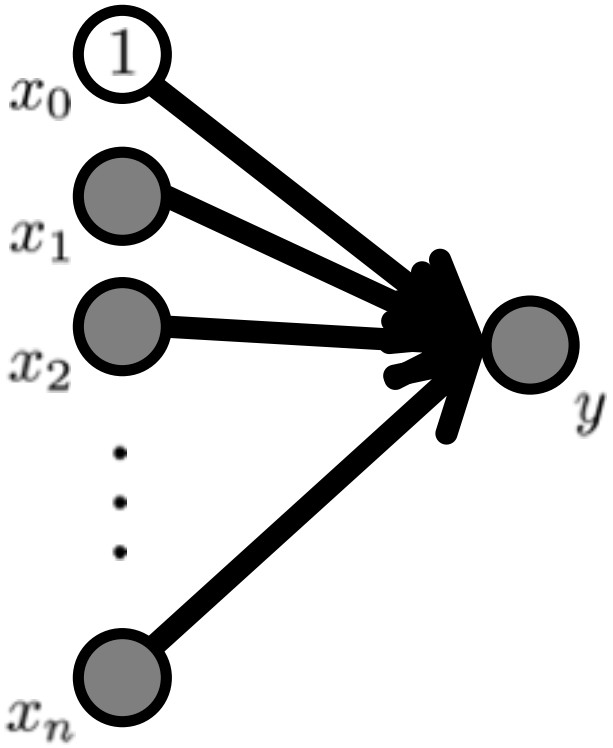
Linear perceptron

bias: $x_0 = 1$

inputs: $x_1, x_2, \ldots, x_n$

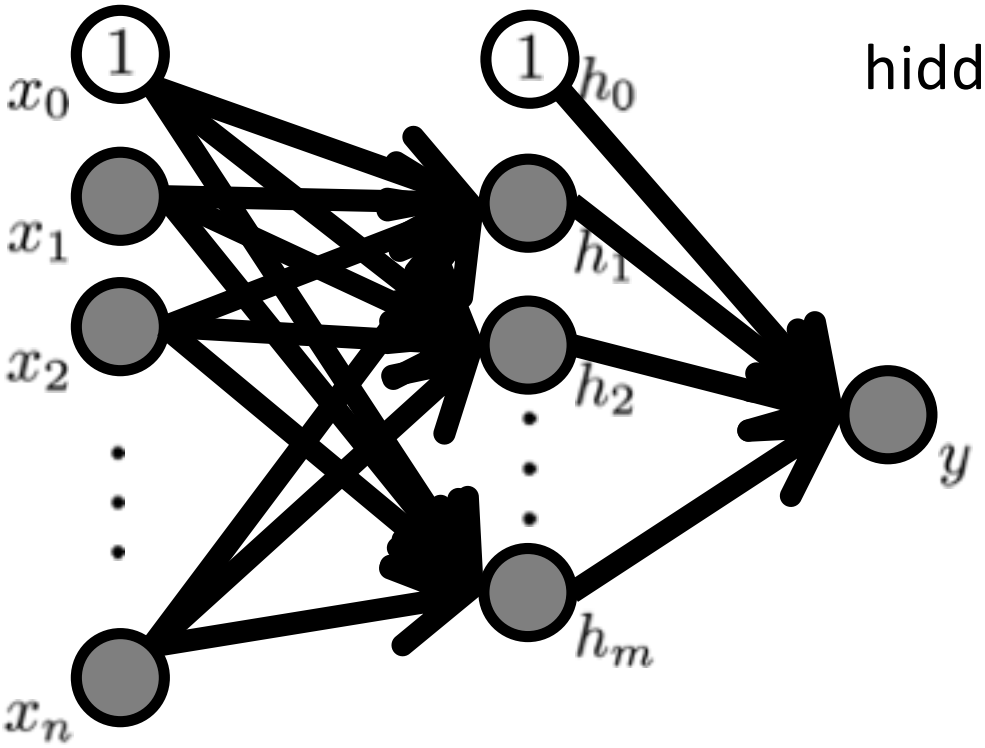
weights: w_i

output: y



$$y = \sum_{i=0}^{n} w_i x_i$$

Linear MLPs



bias: $x_0 = 1$

inputs: $x_1, x_2, \ldots, x_n$

weights: w_{ij}

hidden units: $h_1, h_2, \ldots, h_m$

output: y

$$h_j = \sum_{i=0}^{n} w_{i,j} x_i$$

$$y = \sum_{j=0}^{m} w_j h_j$$

XOR solved?

$$h_j = \sum_{i=0}^{n} w_{i,j} x_i \qquad y = \sum_{j=0}^{m} w_j h_j$$

$$y = \sum_j w_j h_j$$

$$= \sum_j w_j \sum_i w_{i,j} x_i$$

$$= \sum_i \sum_j w_j w_{i,j} x_i$$

$$= \sum_i w_i^* x_i \qquad w_i^* = \sum_j w_j w_{i,j}$$

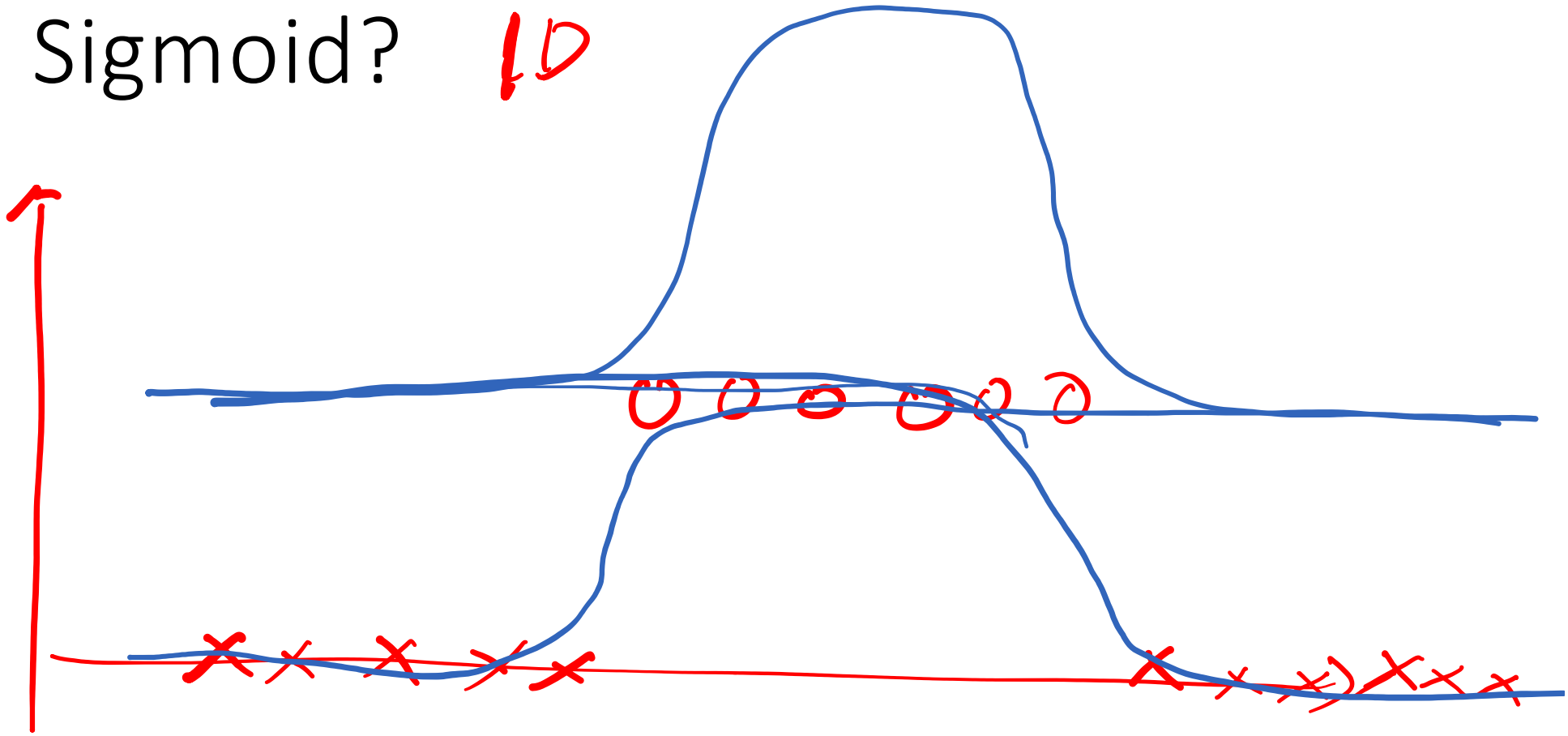
$$y = \sum_{j=0}^{m} w_j \sum_{i=0}^{n} w_{i,j} x_i$$

$$y = \sum_{i=0}^{n} \sum_{j=0}^{m} w_j w_{i,j} x_i$$

$$w_i^* = \sum_{j=0}^{m} w_j w_{i,j}$$

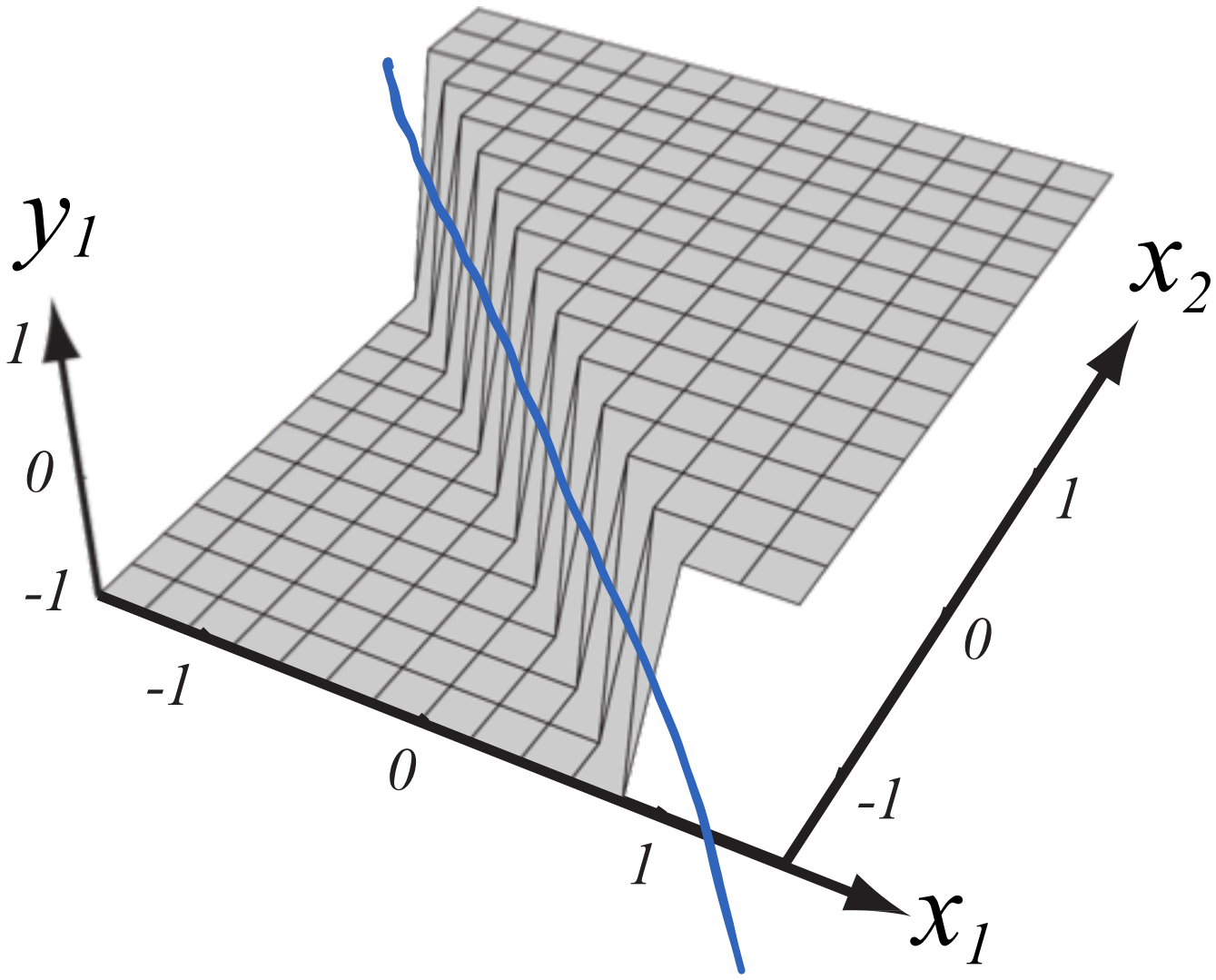
$$y = \sum_{i=0}^{n} w_i^* x_i$$

Sigmoid?

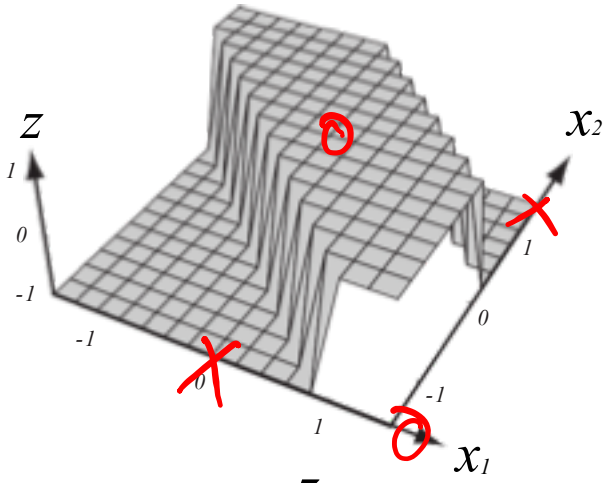


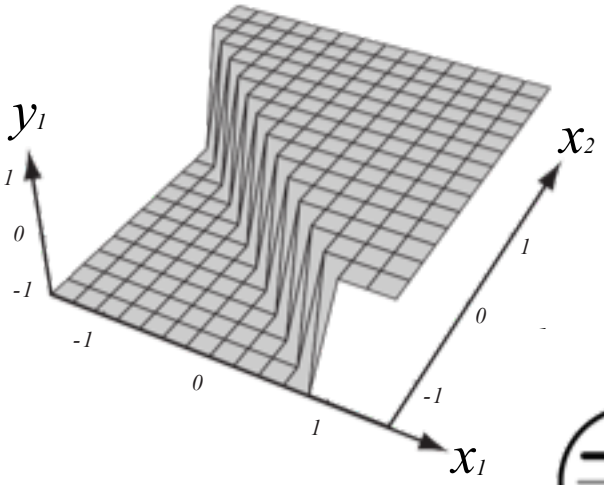
Non-linear Activation

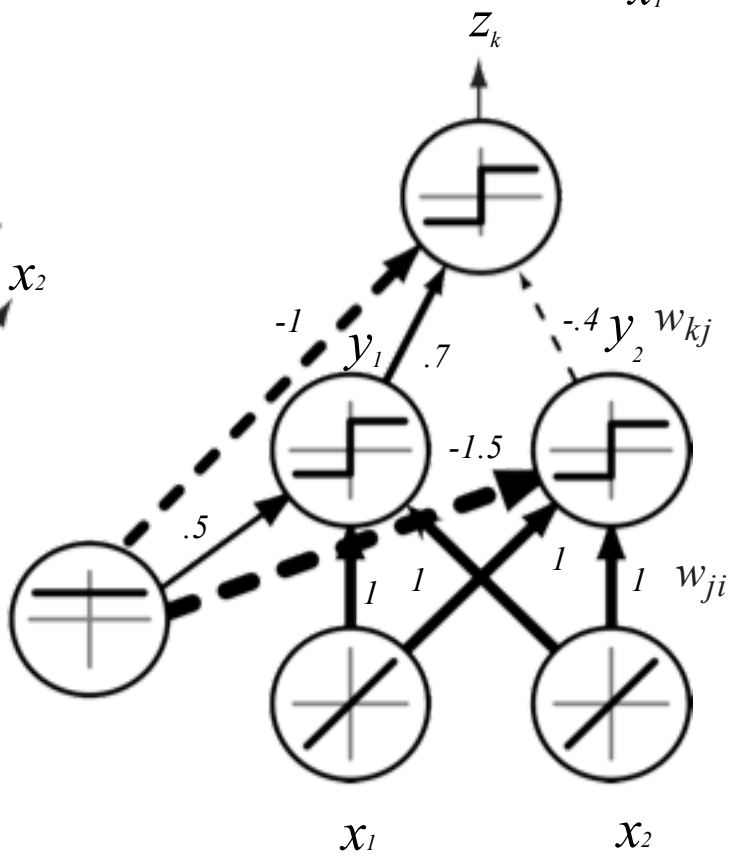
$w^T x$

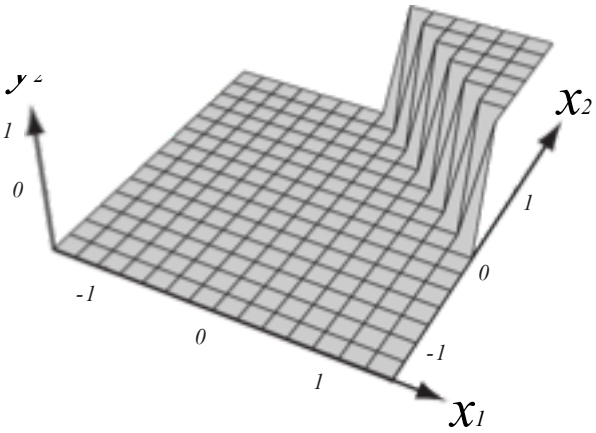


Single Hidden Layer

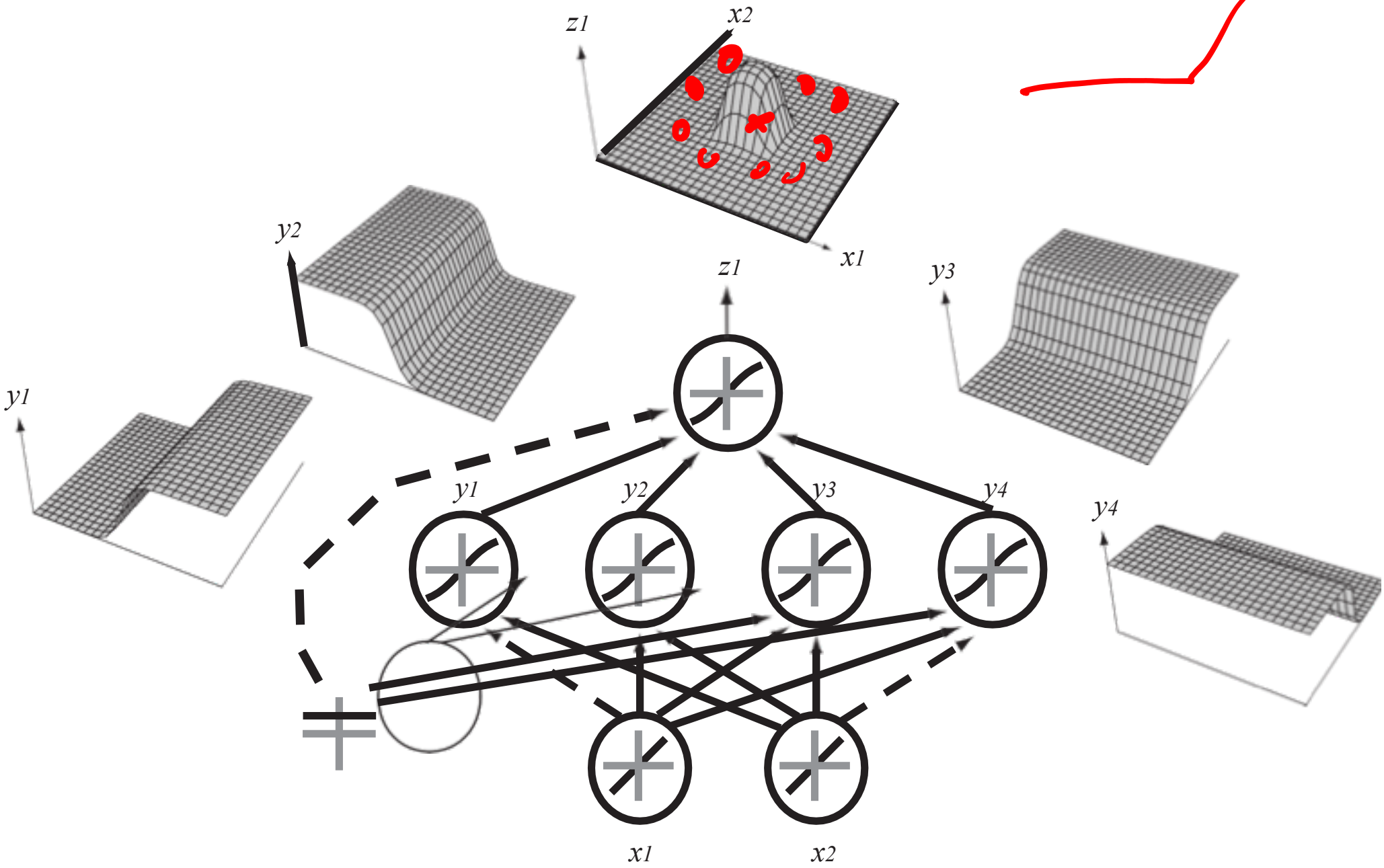






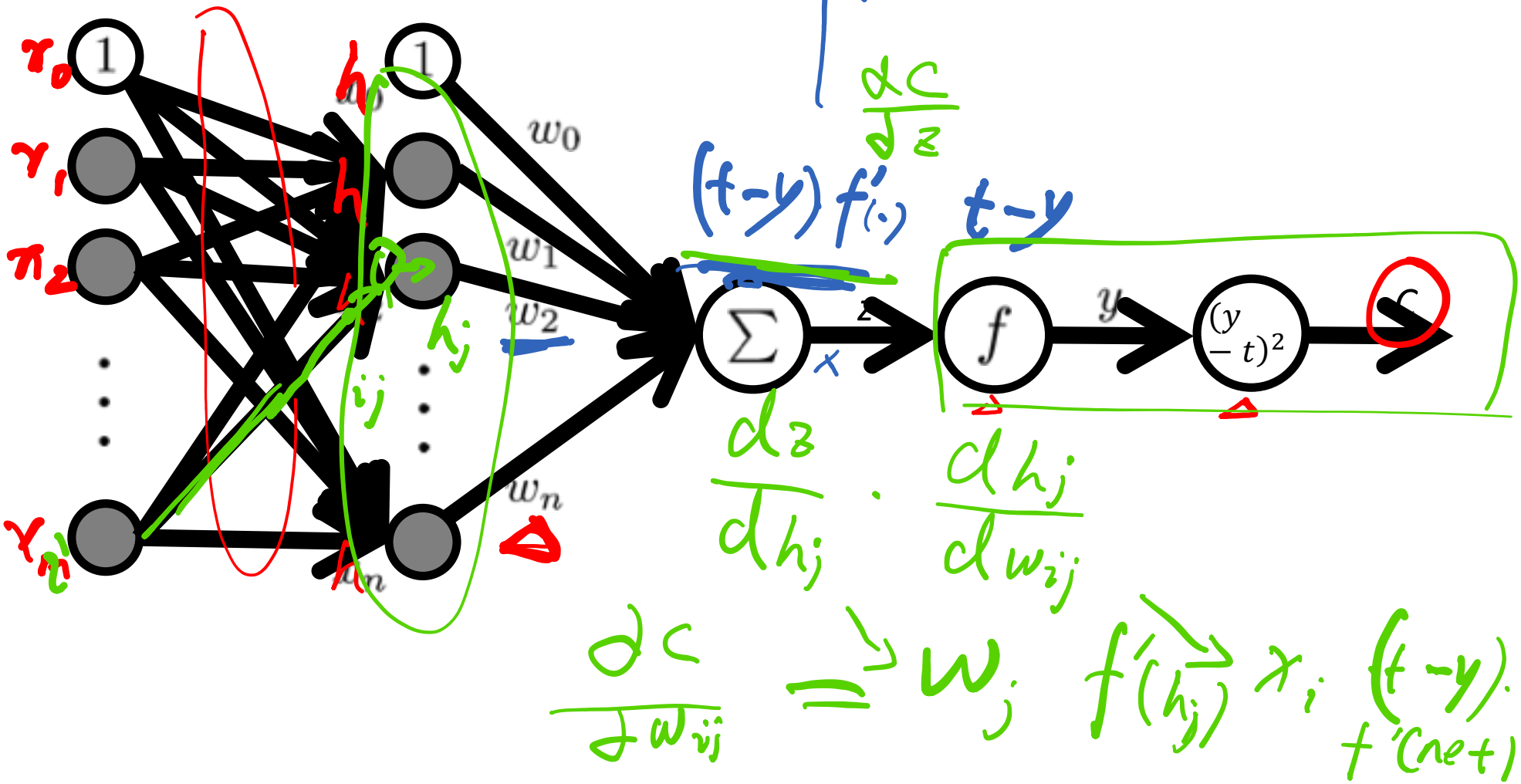


Single Hidden Layer



Multiple Layer

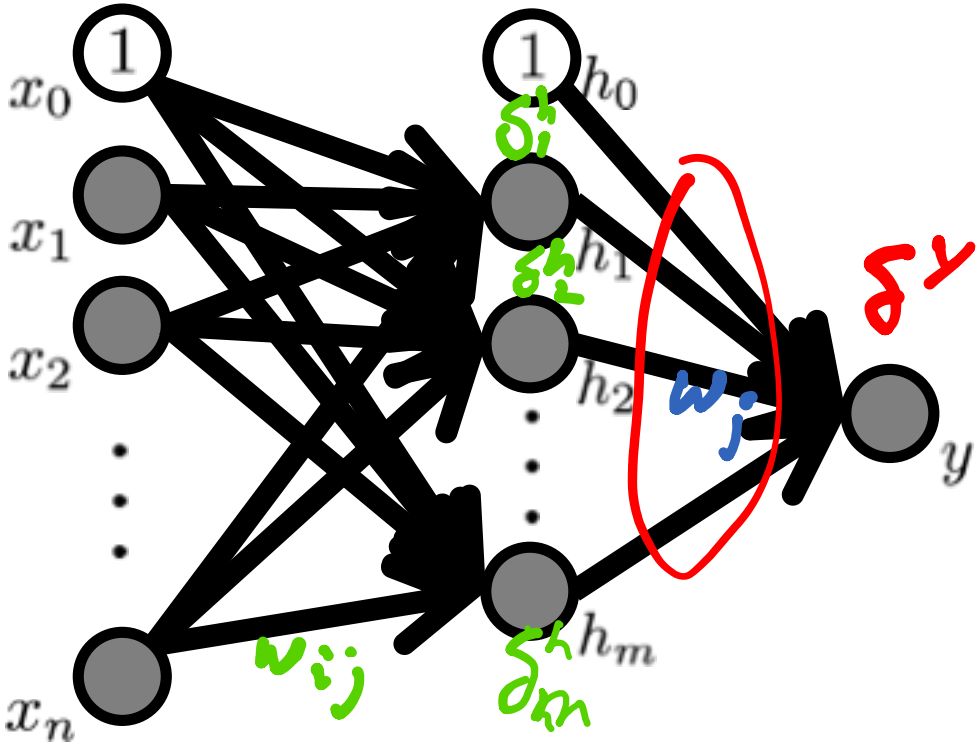
$$\Delta w_i = \alpha(t_j - y_j) f'(net_j) x_{ij}$$



$$\Delta w_j = \alpha h_j (t - y) f'(net)$$

$$= \alpha h_j \delta^y$$

$\delta^y = \frac{\partial C}{\partial y}$

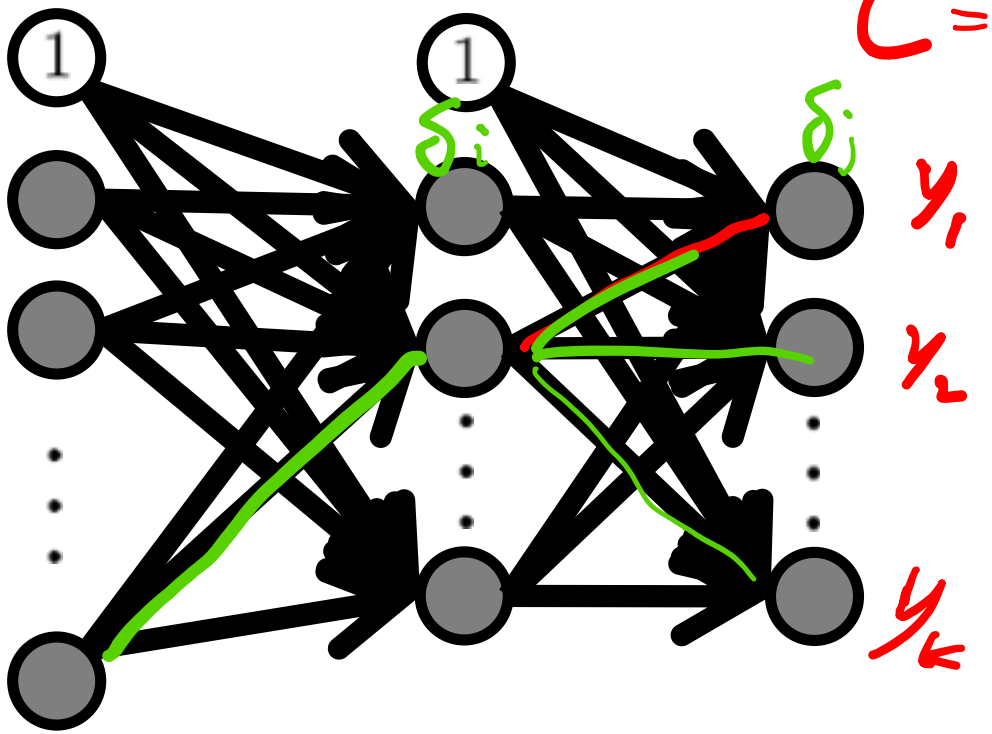


$$\Delta w_{ij} = \alpha x_i w_j \delta^y f'(net_j)$$

$$= \alpha x_i \delta_j^h$$

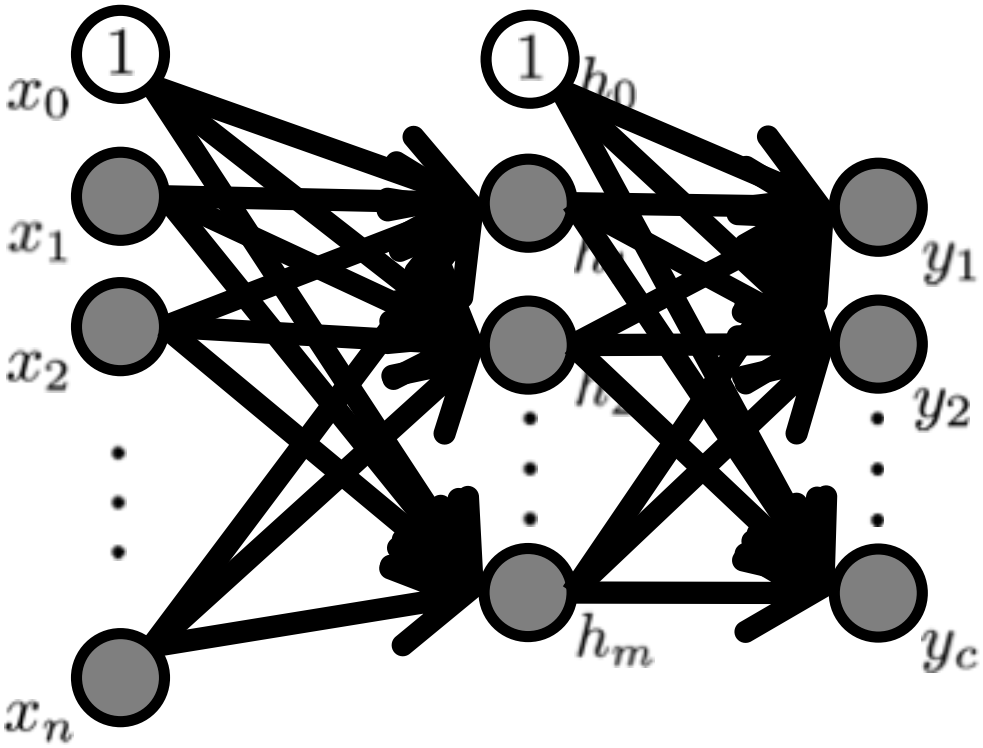
$\frac{\partial C}{\partial h_j}$

In general: $\delta_i = f'(net_i) \sum_j w_{ij} \delta_j$



$C = \sum_k (y_k - t_k)^2$

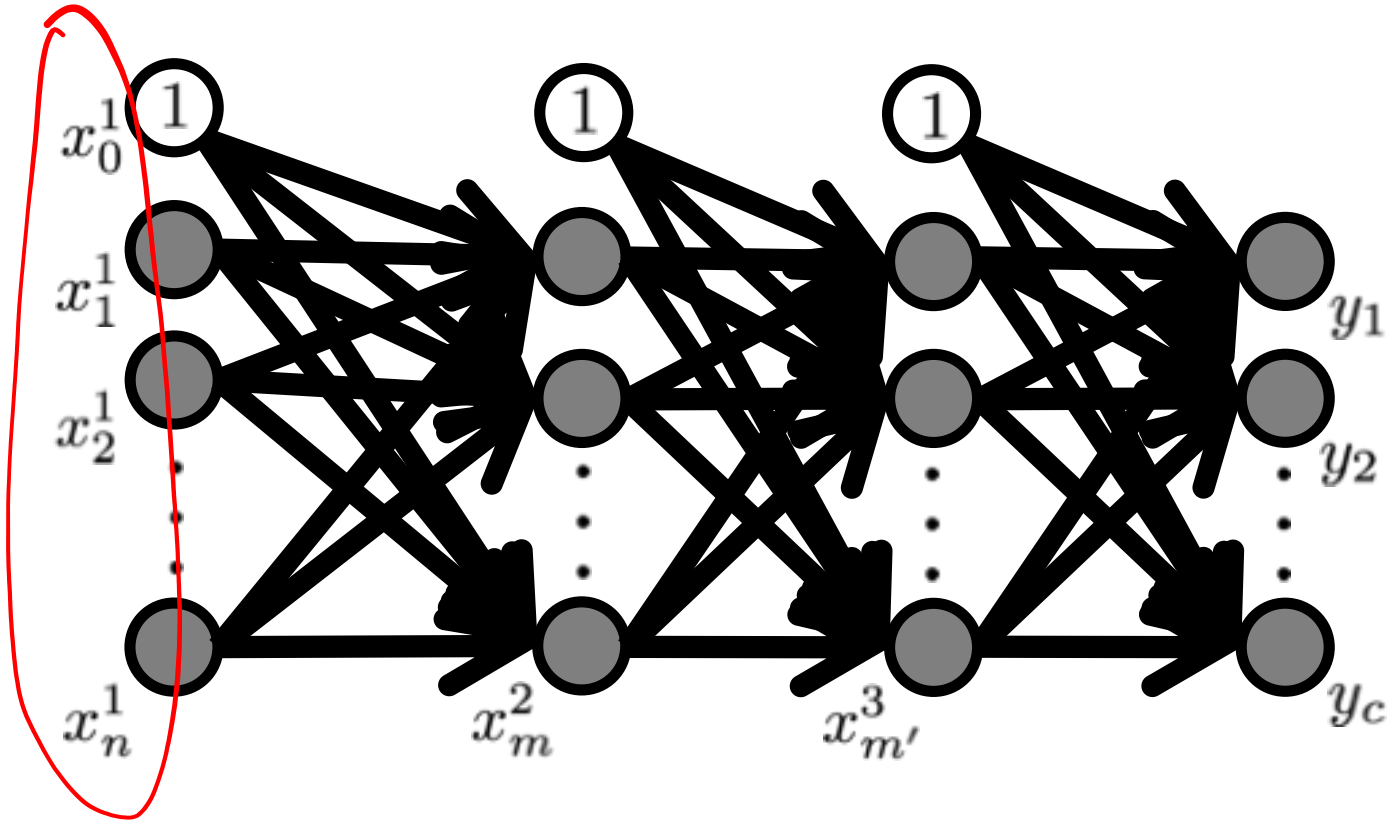
Multiple output



$$y_k = f(\sum_{j=0}^{m} w_{jk} h_j)$$

Putting it all together (forward pass)

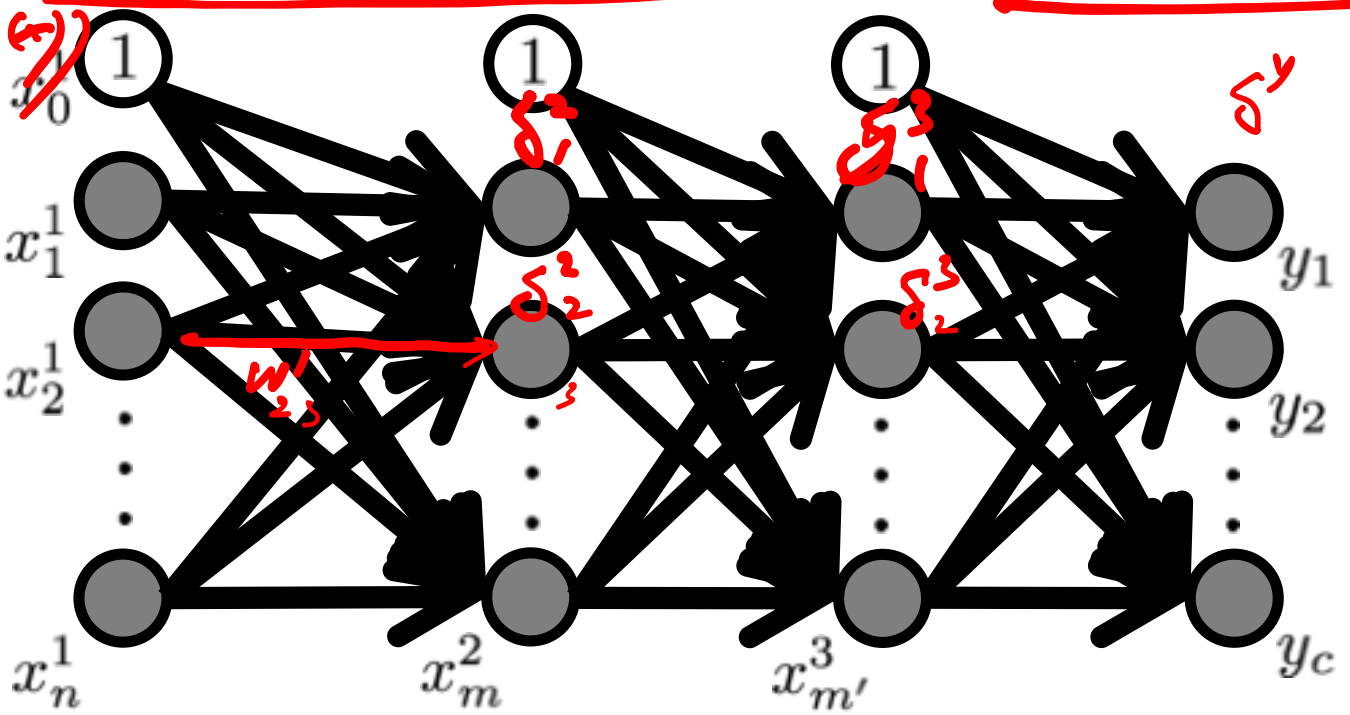
$$x_k^{i+1} = f\left(\sum_{j=0} w_{jk}^i x_j^i\right) \qquad y_k = f\left(\sum_{j=0} w_{jk}^3 x_j^3\right)$$



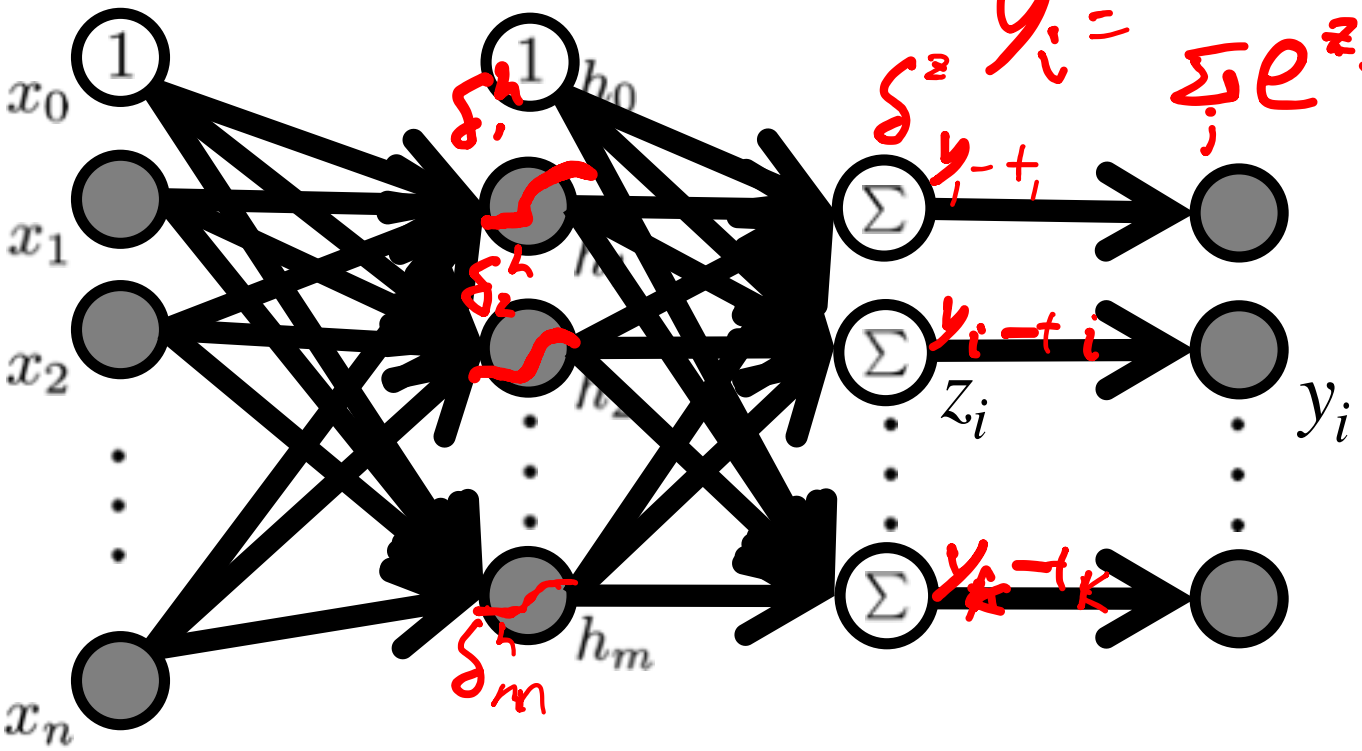
Putting it all together (backward pass)

$$\delta_k^y = (t_k - y_k) f'(net_k^y) \qquad \Delta w_{jk}^y = \alpha x_j^3 \delta_k^y$$

$$\delta_j^{i-1} = f'(net_j^{i-1}) \sum_k w_{jk}^i \delta_k^i \qquad \Delta w_{jk}^i = \alpha x_j^i \delta_k^i$$



Multiclass output: Softmax



How many hidden units should I use?

Overfitting

Train/Validation/Test Sets:

Train: Data used to train the network.

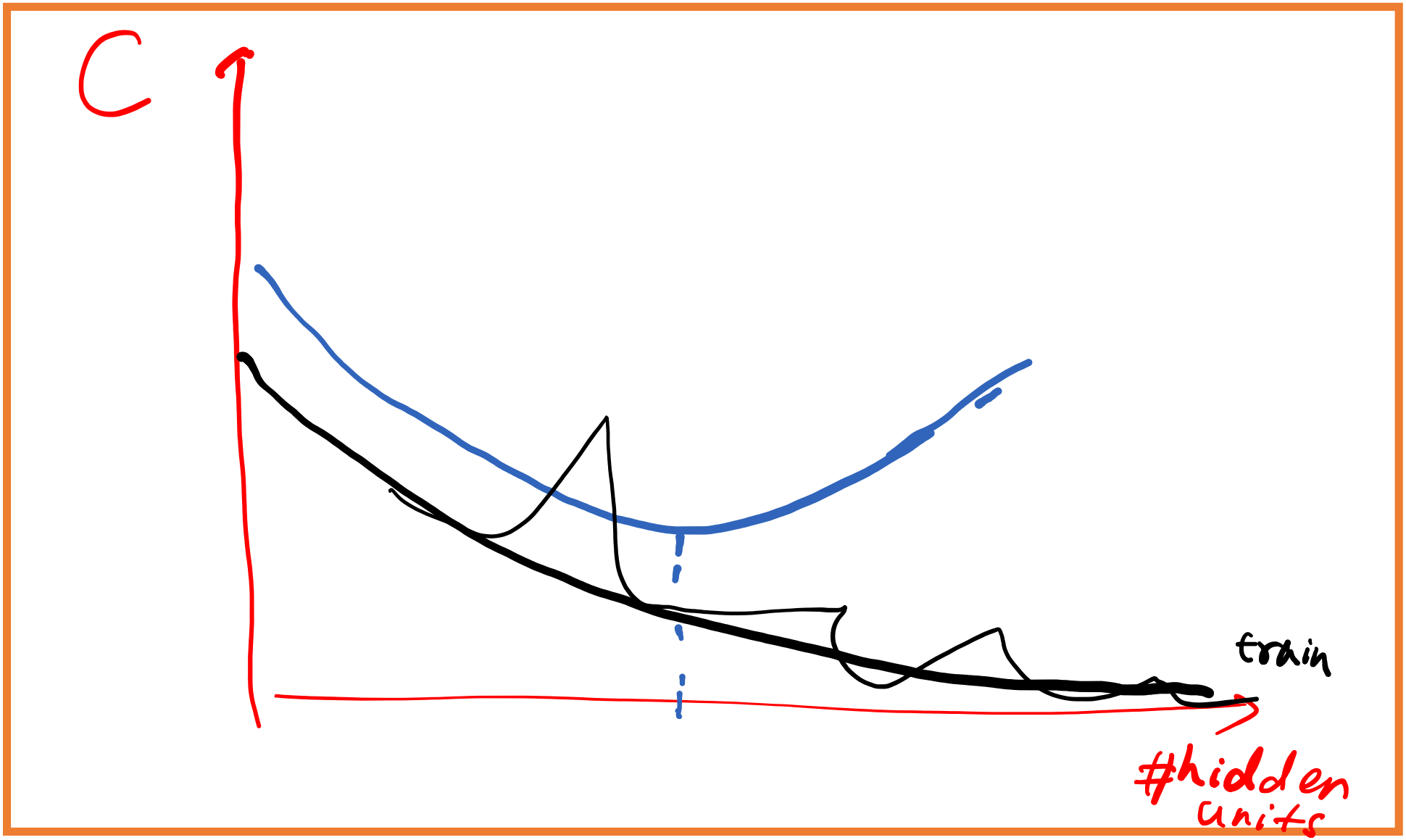
Validation: Data used to determine whether training is overfitting

Test: The data used when in production

Train vs. Validation

- Randomly sample dataset to create train and validation datasets.
- You may want to do several random splits to get a sense of the variance in your results (cross-validation).
- Only train on train!
- For best results validation should be similar to test (but not the same!)

Overfitting (looking at the errors)



Training set size

- The more training examples you have, the larger your network can be without overfitting.
- 1,000 to 10,000 training examples per class.

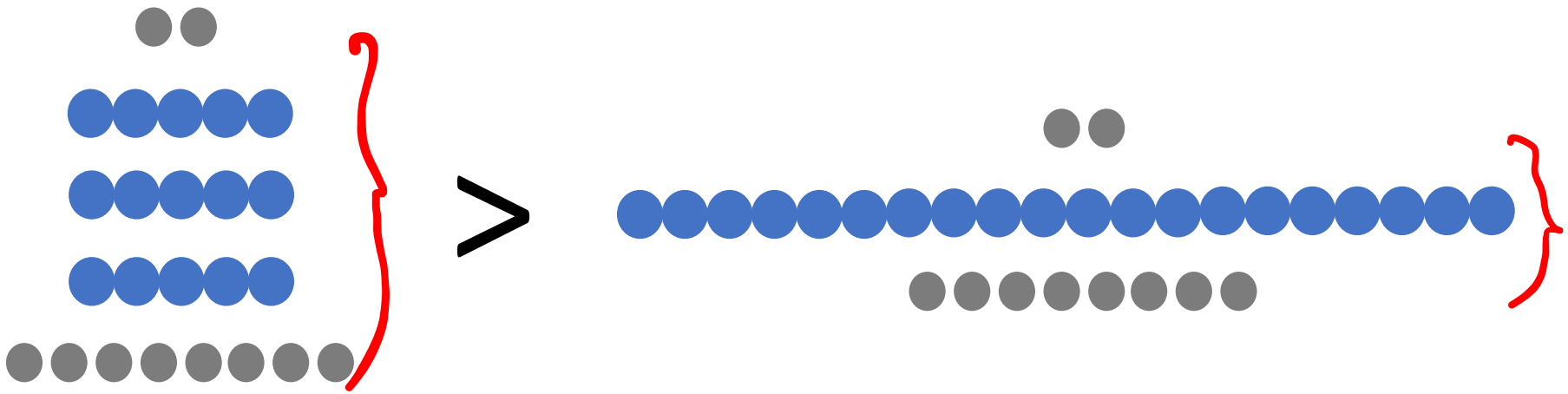
Rules of thumb

- The number of hidden units should be between the number of inputs and outputs.
- The number of hidden units should be 2/3 the number of input plus the number of output.
- Start with a smaller number of hidden units and increase their number.
- ...

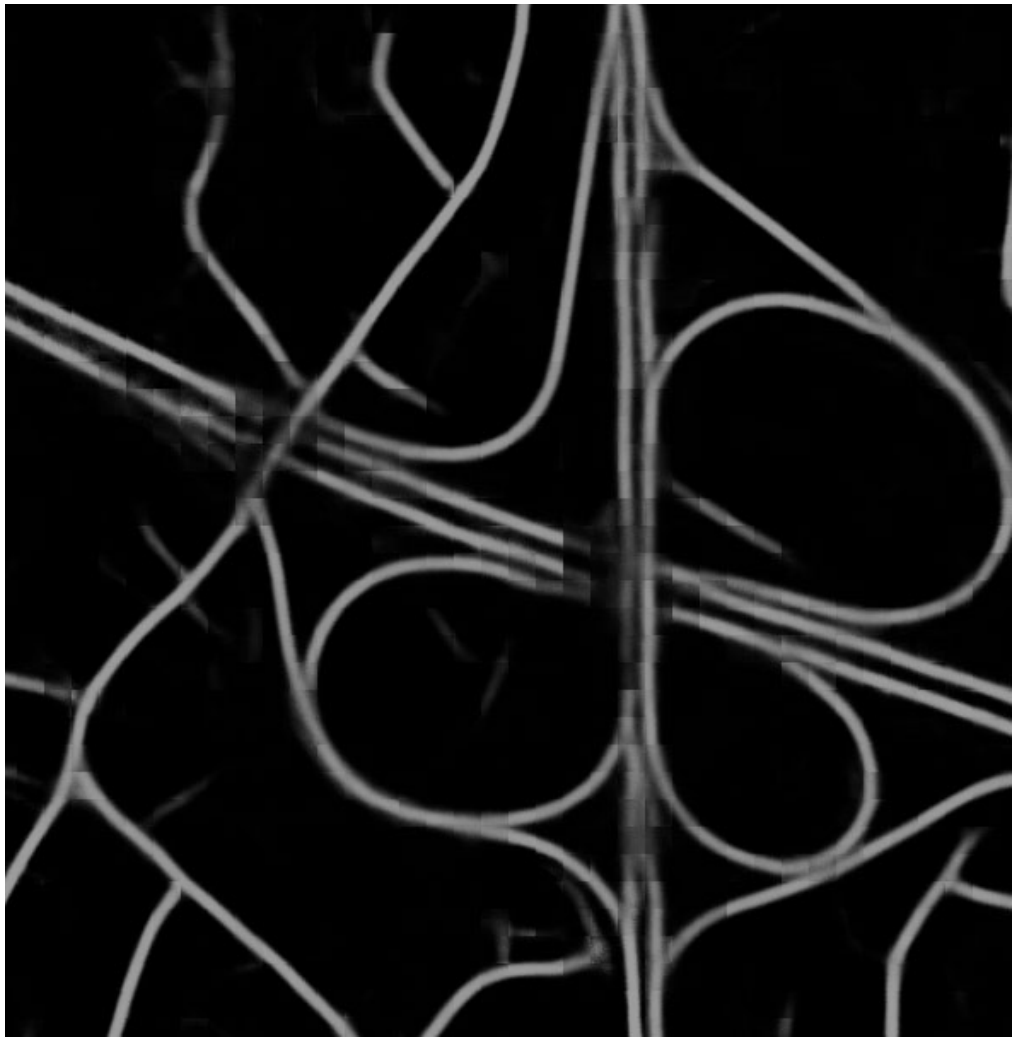
- Try several different numbers and see!

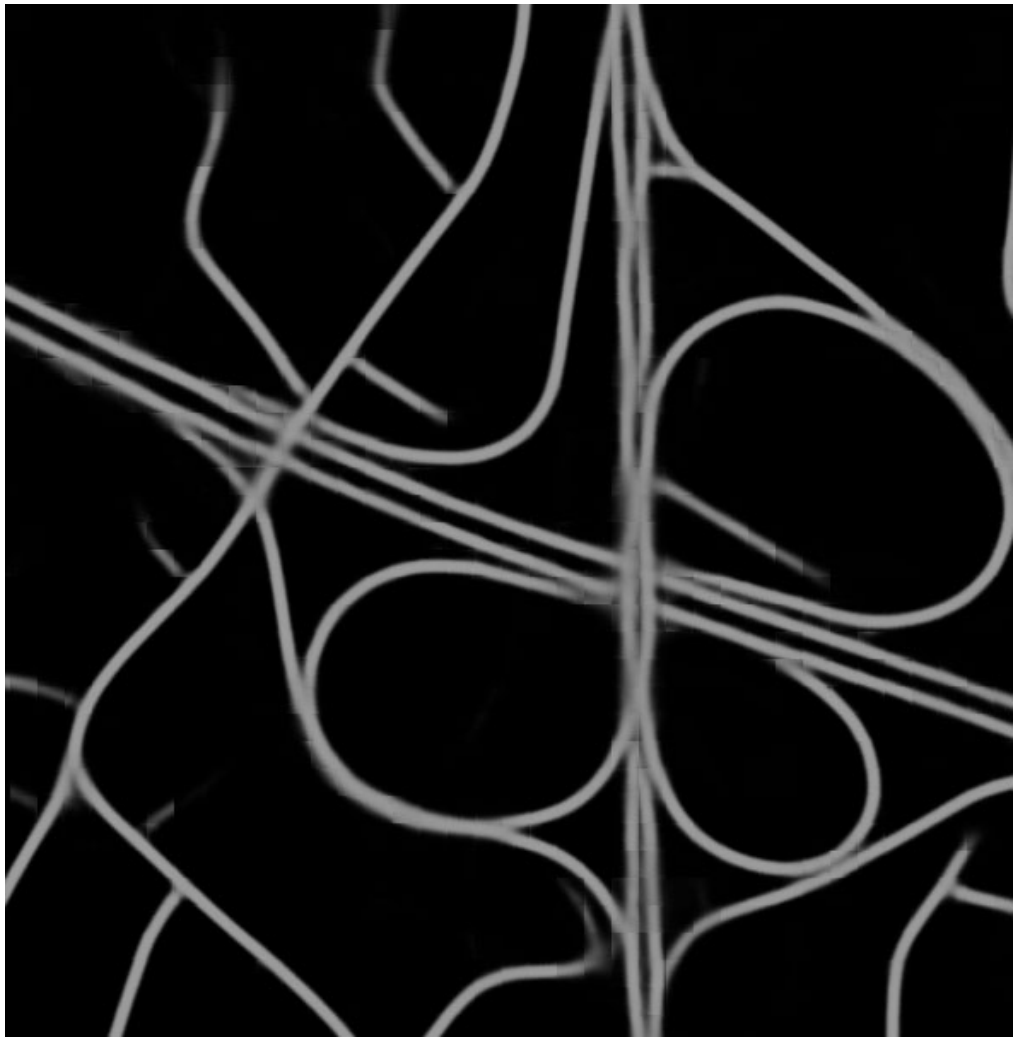
Deep networks

deeper > wider



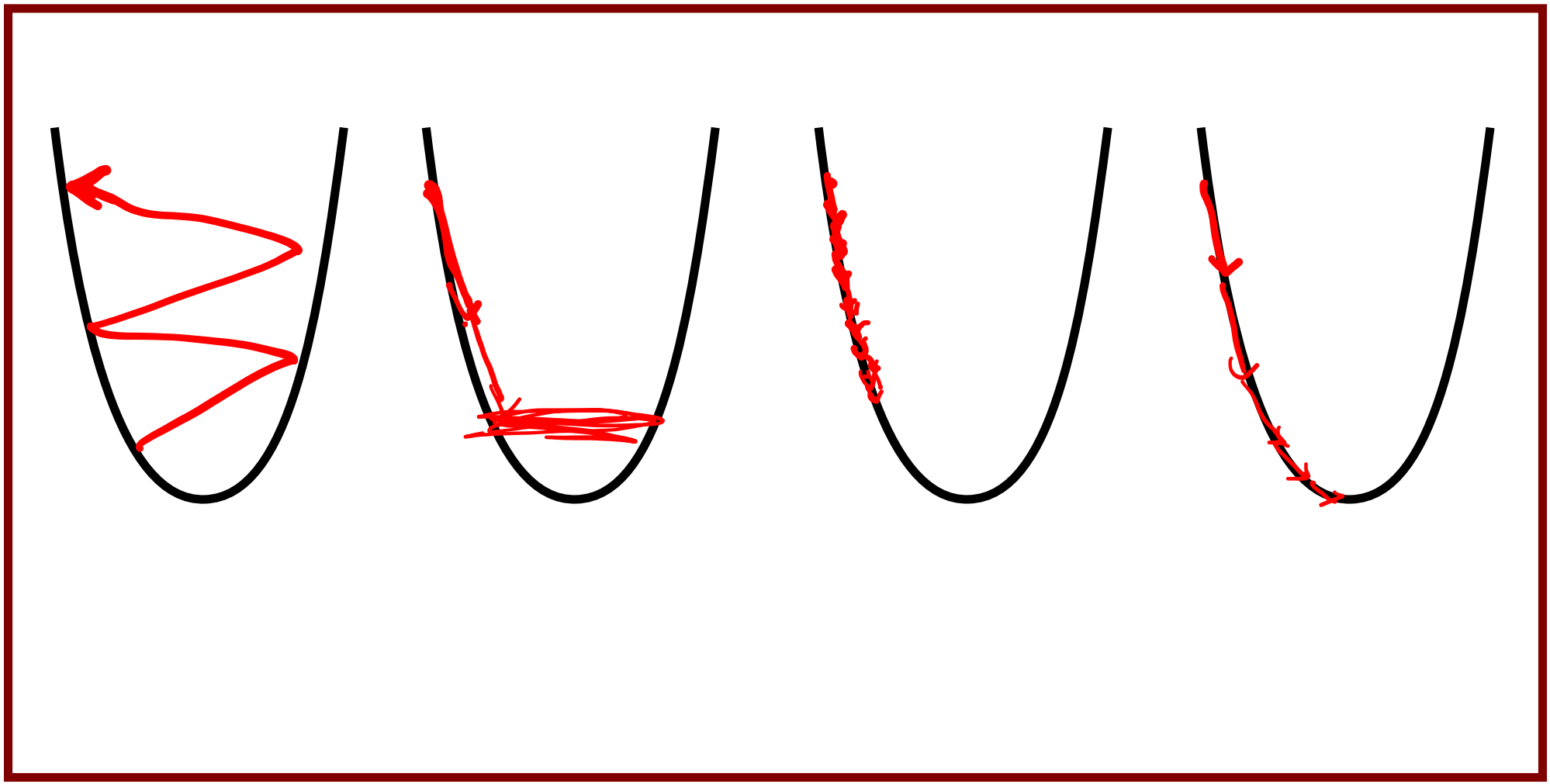
Deeper Neural Net Models



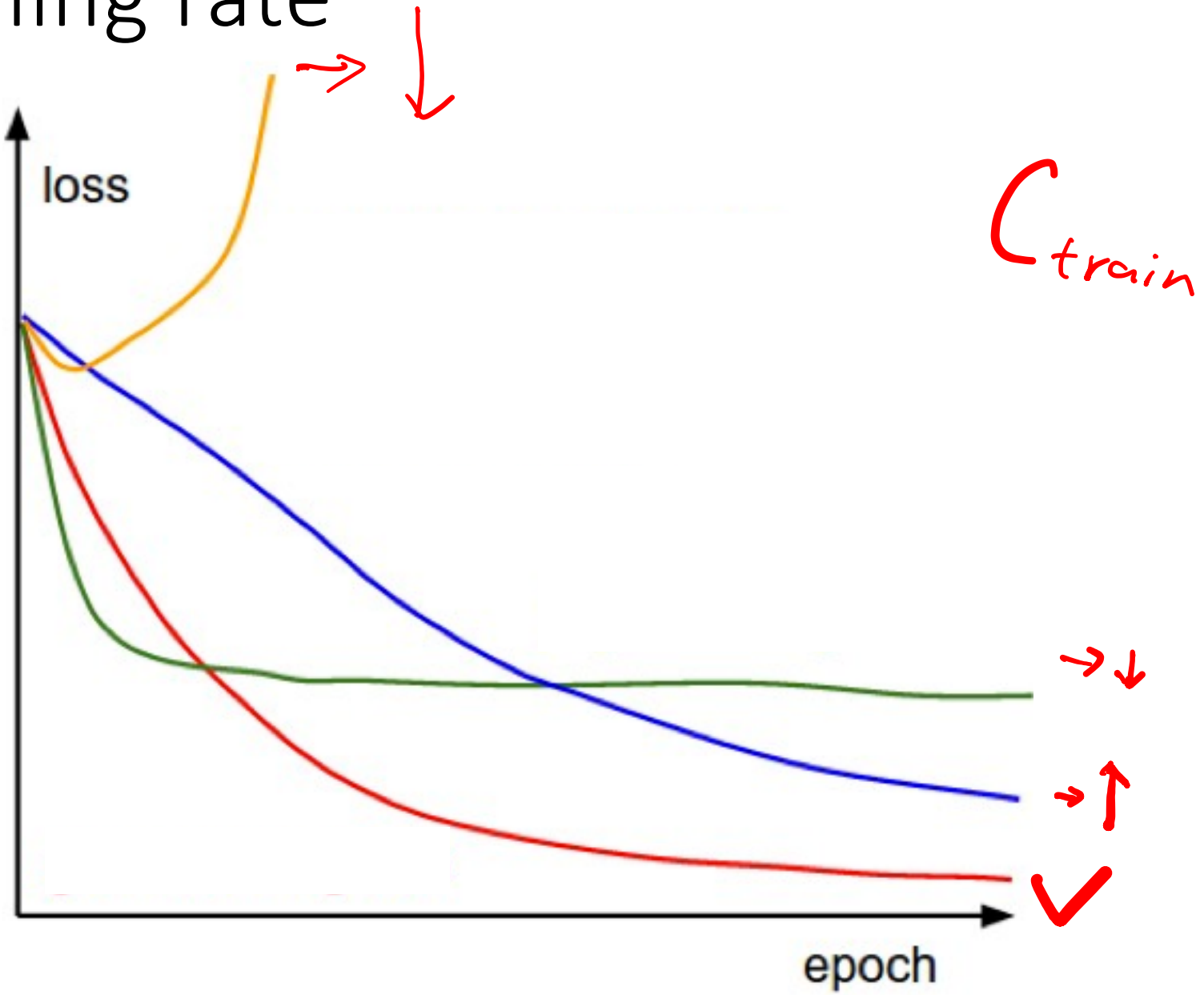


Learning rate

Picking the right learning rate



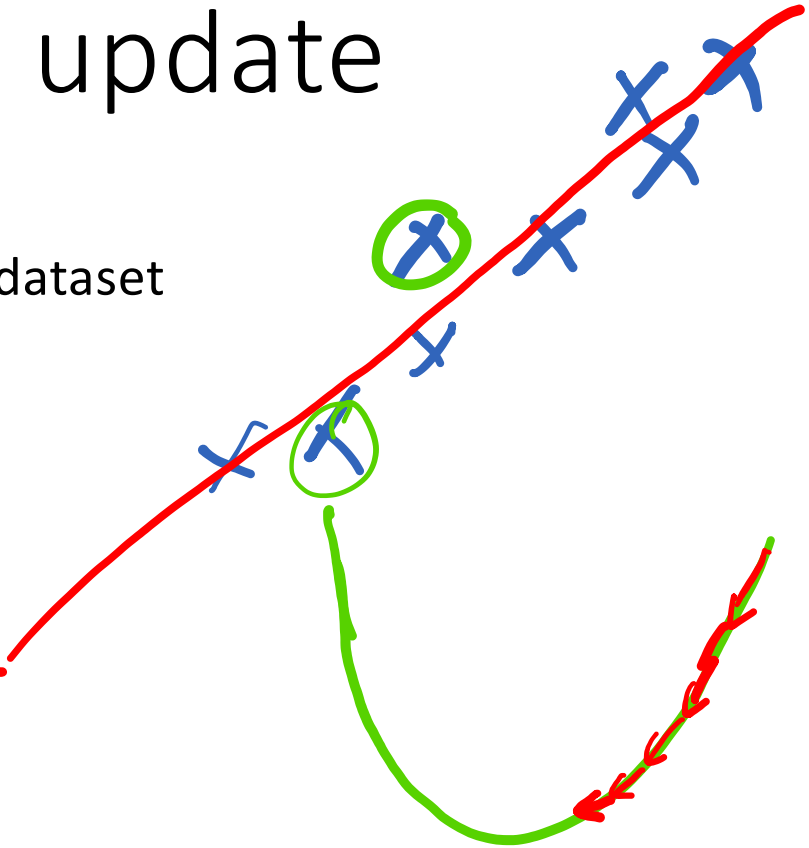
Learning rate



Batch vs. incremental update

Batch (gradient descent):

1. Compute the average $\Delta \mathbf{w}$ over the whole dataset
2. Update weights
3. Repeat until convergence



Incremental (Stochastic Gradient Descent SGD):

1. Compute $\Delta \mathbf{w}$ for a single training example
2. Update weights
3. Repeat until convergence

In practice, weights are typically updated using “mini-batches” that are subsets of the entire dataset. Sample dataset randomly.

Code

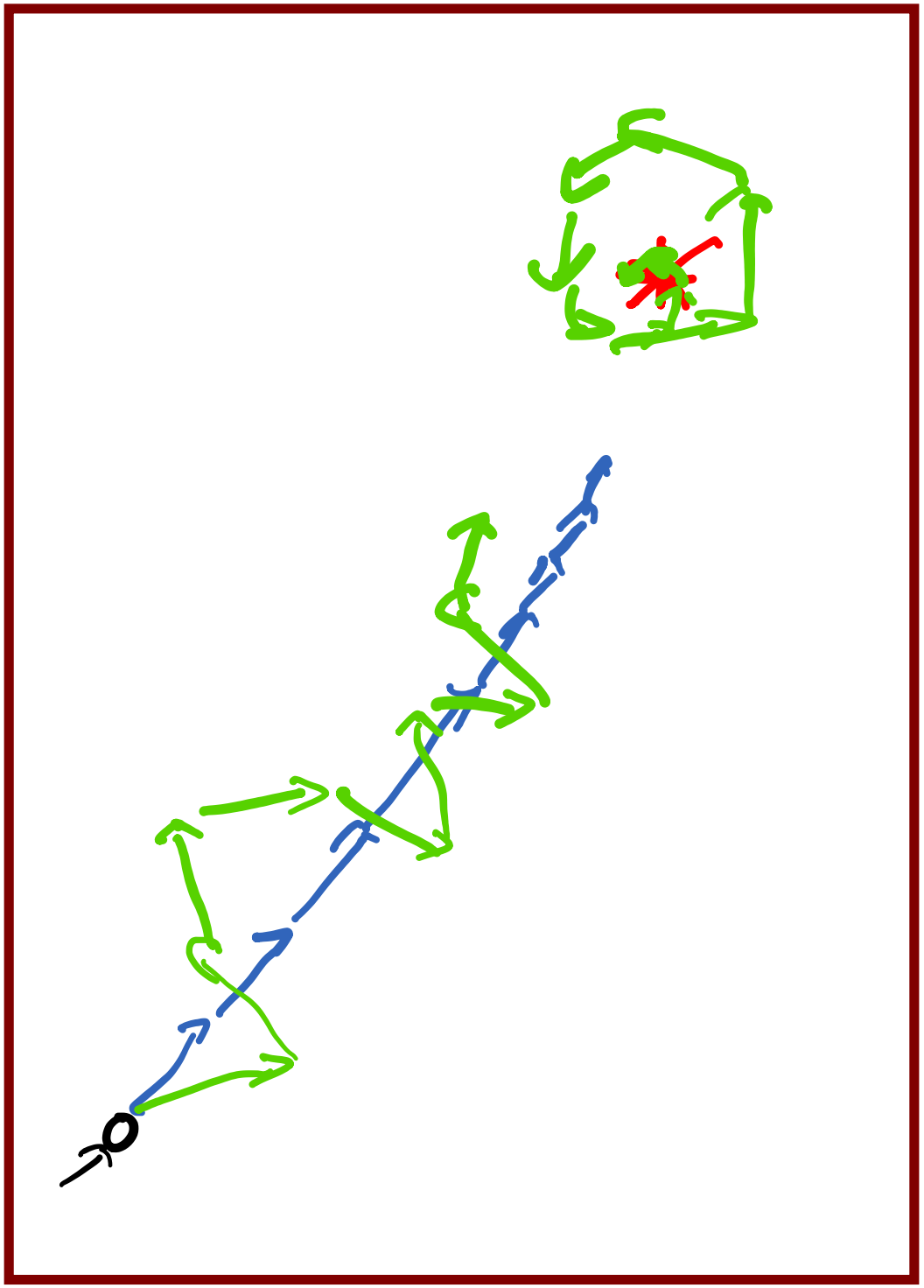
Iteration: One pass through a batch
Epoch: One pass through the training data

```
for(i=0;i<num_iterations;i++) {
        delta_sum = 0;
        for(j=0;j<batch_size;j++) {
                ComputeWeightDeltas(j, delta);
                for(k=0;k<num_Weights;k++)
                        delta_sum[k] += delta[k];
        }
        for(k=0;k<num_Weights;k++)
                weights[k] += learning_rate*delta_sum[k];
}
```

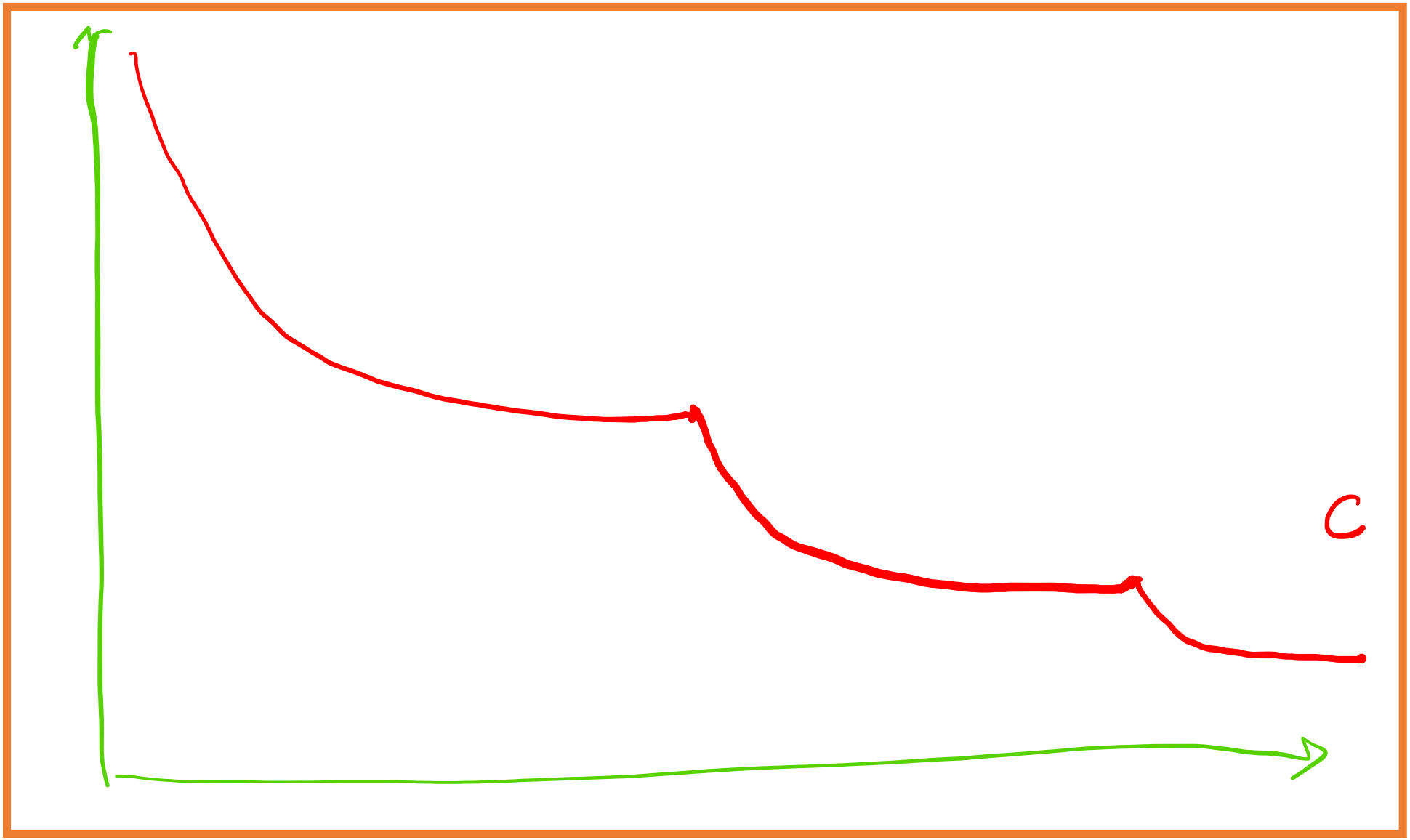
Learning rate

Batch vs. incremental

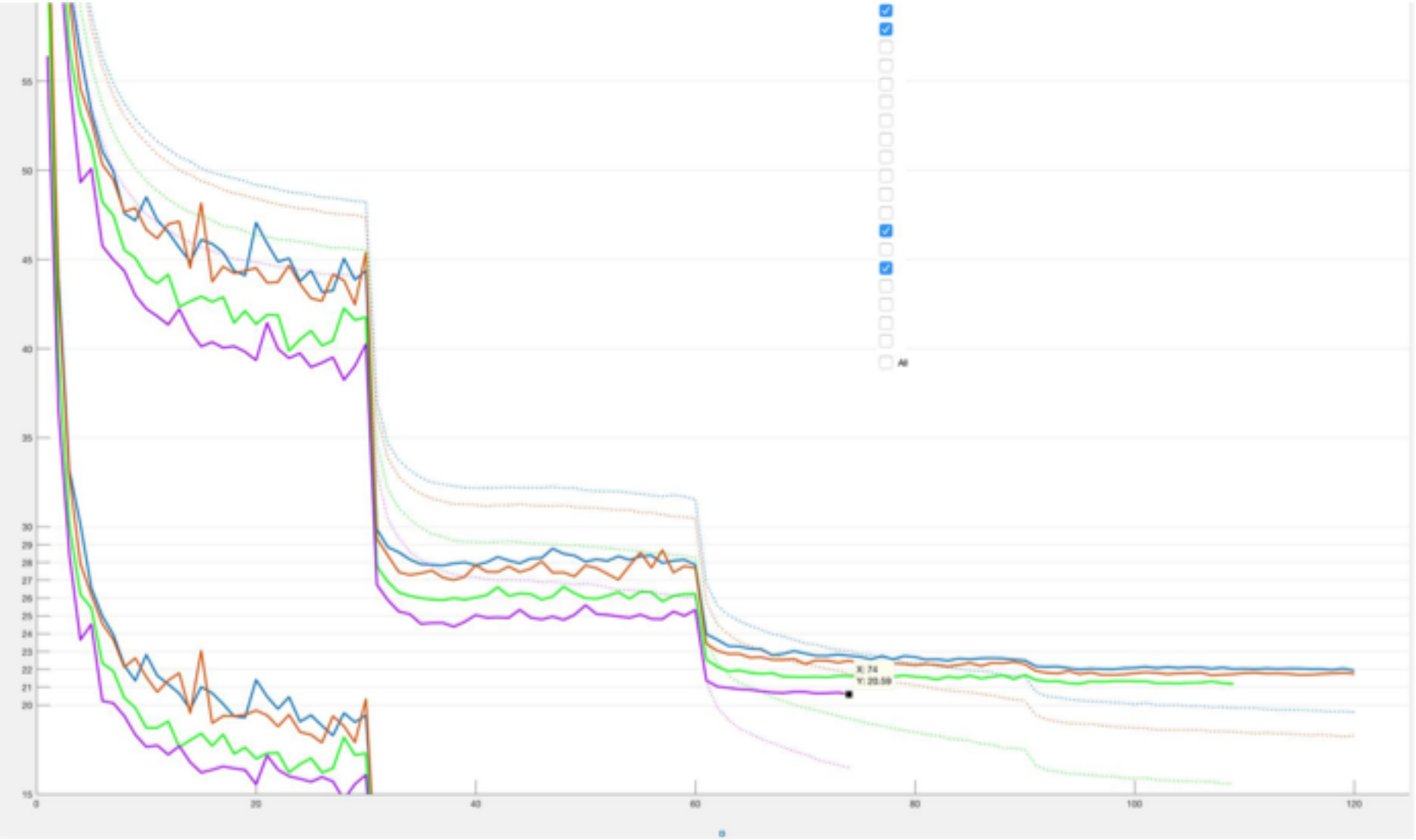
How should the learning rate differ between batch vs. incremental?



Learning rate schedule (stepping down)

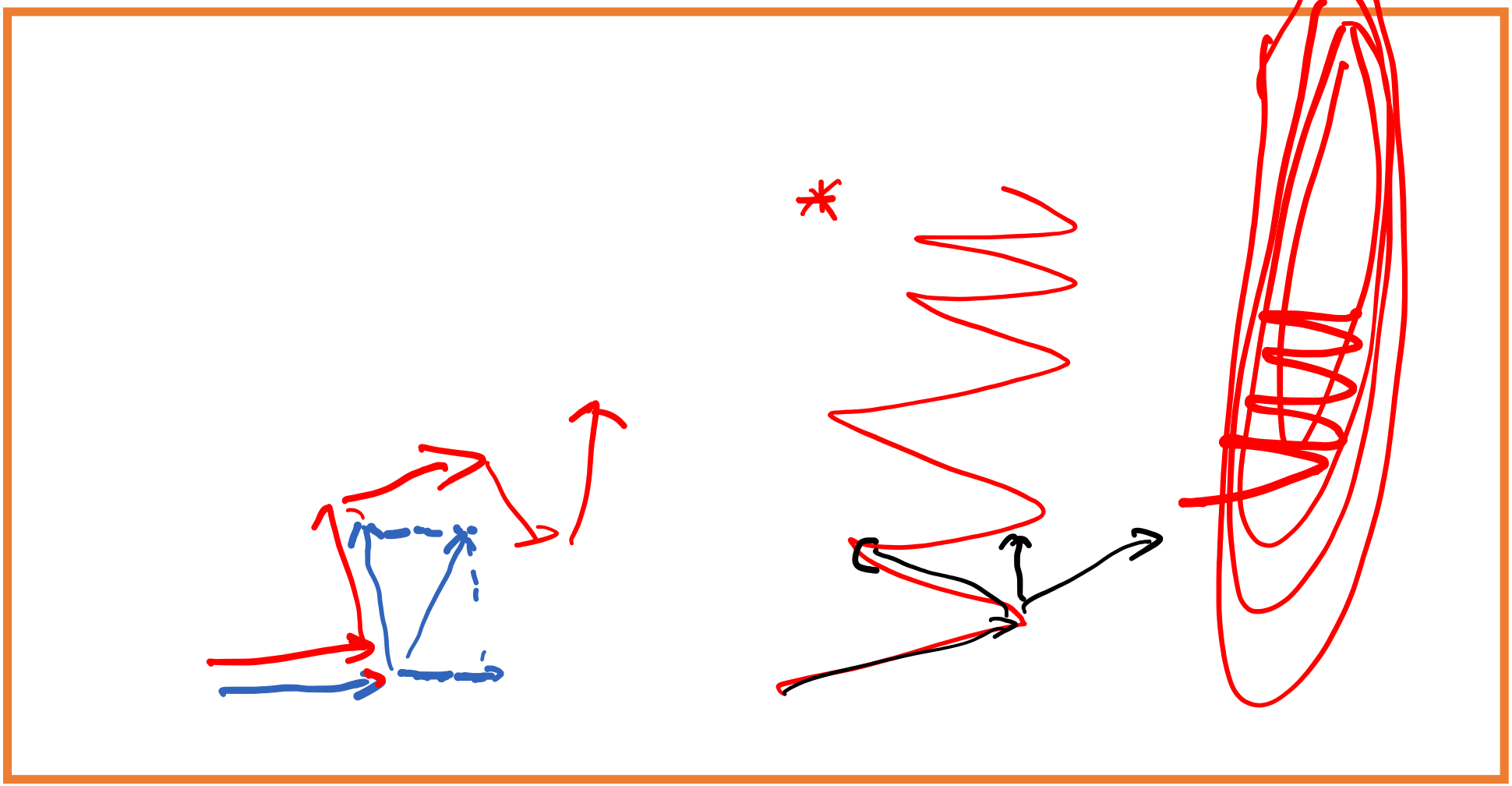


Learning example

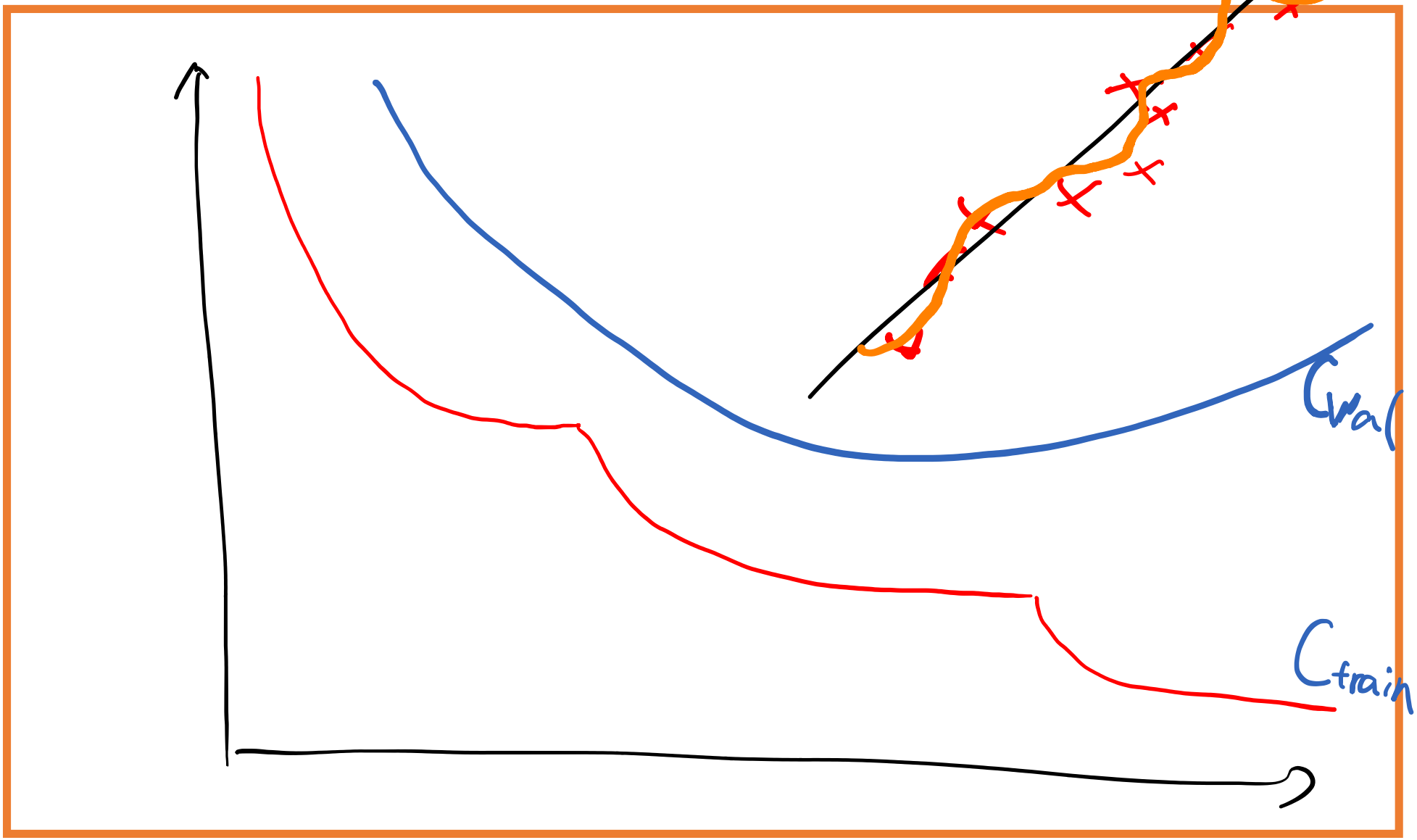


Momentum

$$\Delta w_{ij}(t+1) = \alpha x_i \delta_j^h + \beta \Delta w_{ij}(t)$$



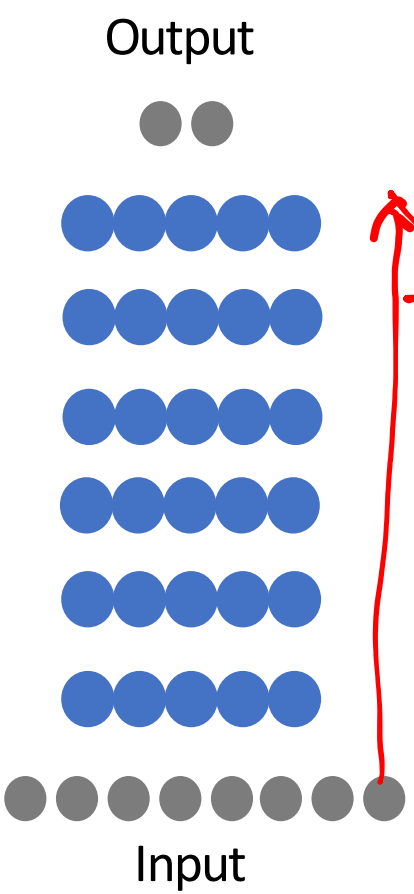
When should you stop learning?



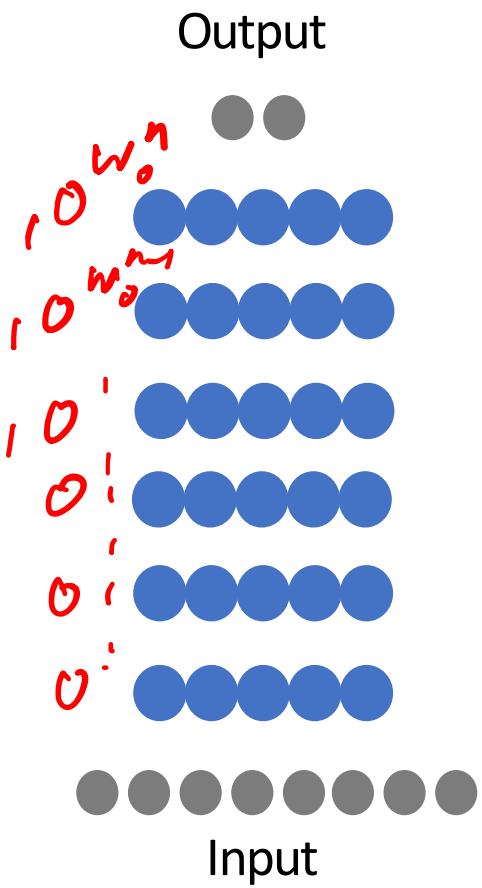
Automatically adjust learning rate

- ADAGRAD, ADADELTA, RMSProp, Adam

ReLU vs. Sigmoid



Weight initialization




Q&A