Neural Network

Announcements

Recap on Boosting

Boosting iteratively learns a new classifier, and add it to the ensemble

Initialize $H_1 = h_1 \in \mathscr{H}$

For t = 1 …

Recap on Boosting

Boosting iteratively learns a new classifier, and add it to the ensemble

Initialize $H_1 = h_1 \in \mathscr{H}$

For t = 1 …

Denote $\hat{\mathbf{y}} = \left[H_t(x_1), H_t(x_2), \ldots, H_t(x_n)\right]^\top \in \mathbb{R}^n$

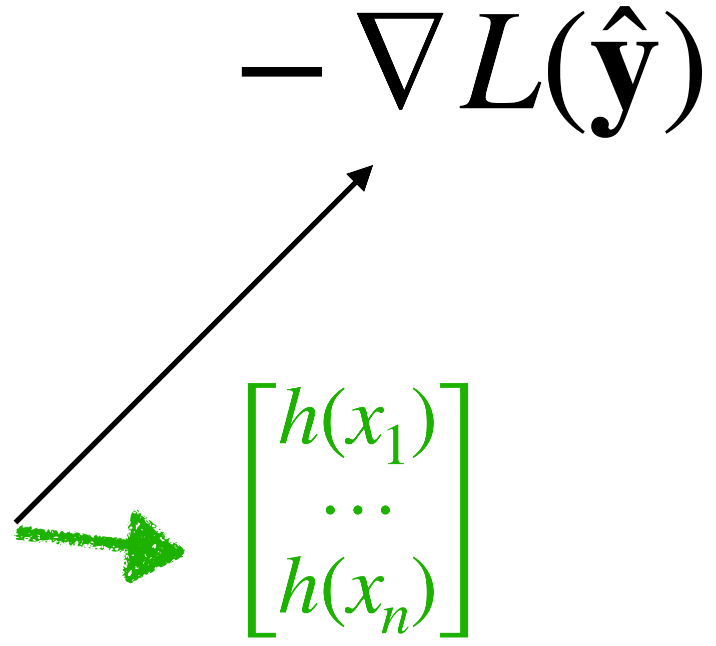
Recap on Boosting

Boosting iteratively learns a new classifier, and add it to the ensemble

Initialize $H_1 = h_1 \in \mathscr{H}$

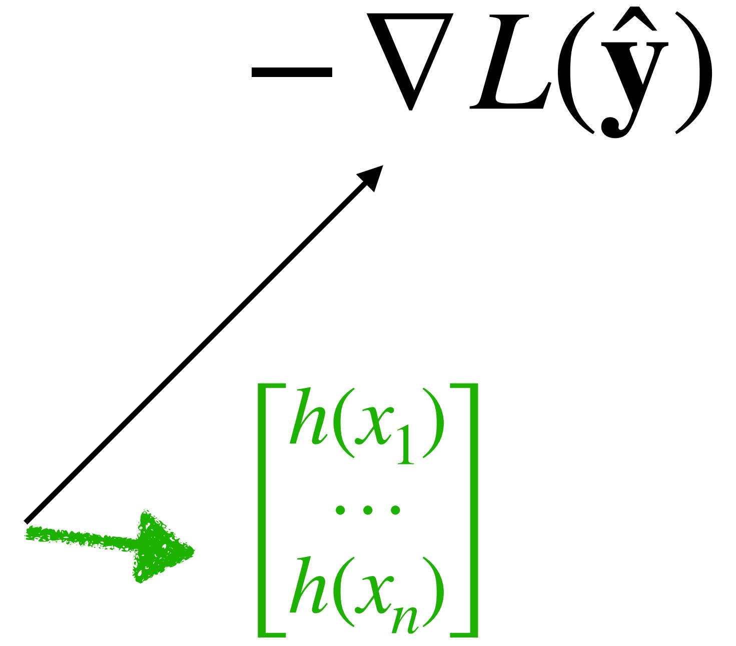
For t = 1 …

Denote $\hat{\mathbf{y}} = \left[H_t(x_1), H_t(x_2), \ldots, H_t(x_n)\right]^\top \in \mathbb{R}^n$



Recap on Boosting

Boosting iteratively learns a new classifier, and add it to the ensemble



Initialize $H_1 = h_1 \in \mathscr{H}$

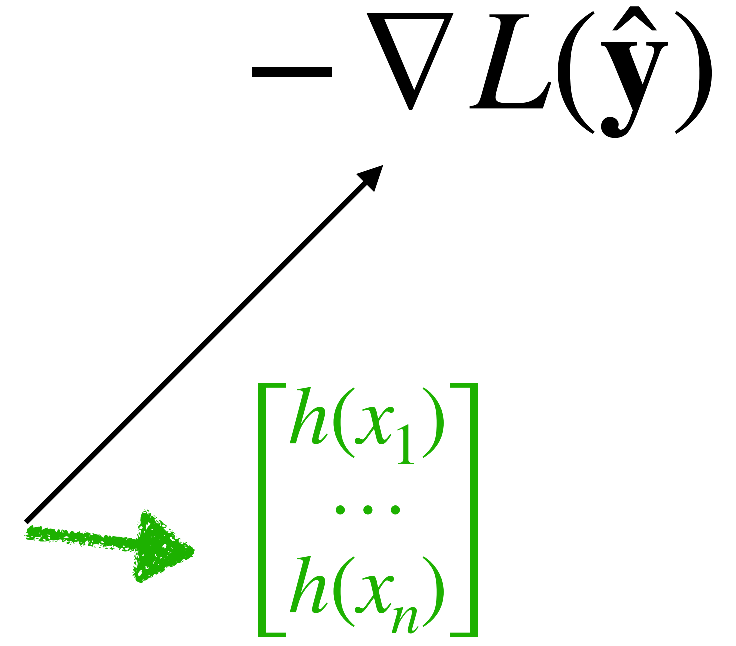
For t = 1 …

Denote $\hat{\mathbf{y}} = \left[H_t(x_1), H_t(x_2), \ldots, H_t(x_n)\right]^\top \in \mathbb{R}^n$

Solve the optimization problem: $h_{t+1} = \arg\max_{h \in \mathscr{H}} \left\langle \begin{bmatrix} h(x_1) \\ \cdots \\ h(x_n) \end{bmatrix}, -\nabla L(\hat{\mathbf{y}}) \right\rangle$

Recap on Boosting

Boosting iteratively learns a new classifier, and add it to the ensemble



Initialize $H_1 = h_1 \in \mathscr{H}$

For t = 1 …

Denote $\hat{\mathbf{y}} = \left[H_t(x_1), H_t(x_2), \ldots, H_t(x_n)\right]^\top \in \mathbb{R}^n$

Solve the optimization problem: $h_{t+1} = \arg\max_{h \in \mathscr{H}} \left\langle \begin{bmatrix} h(x_1) \\ \cdots \\ h(x_n) \end{bmatrix}, -\nabla L(\hat{\mathbf{y}}) \right\rangle$

$$H_{t+1} = H_t + \alpha h_{t+1}$$

Recap on AdaBoost

Adaboost follows this framework with $\ell(\hat{y}, y) = \exp(-\hat{y} \cdot y)$

1. Create a new weighted dataset:

Recap on AdaBoost

Adaboost follows this framework with $\ell(\hat{y}, y) = \exp(-\hat{y} \cdot y)$

1. Create a new weighted dataset:

For each x_i, compute $p_i \propto \exp(-\hat{y}_i \cdot y_i)$

Recap on AdaBoost

Adaboost follows this framework with $\ell(\hat{y}, y) = \exp(-\hat{y} \cdot y)$

1. Create a new weighted dataset:

For each x_i, compute $p_i \propto \exp(-\hat{y}_i \cdot y_i)$

Binary classification: $h_{t+1} = \arg\min_{h \in \mathscr{H}} \sum_i p_i \cdot \mathbf{1}\{h(x_i) \neq y_i\}$

Recap on AdaBoost

Adaboost follows this framework with $\ell(\hat{y}, y) = \exp(-\hat{y} \cdot y)$

1. Create a new weighted dataset:

For each x_i, compute $p_i \propto \exp(-\hat{y}_i \cdot y_i)$

Binary classification: $h_{t+1} = \arg\min_{h \in \mathscr{H}} \sum_i p_i \cdot \mathbf{1}\{h(x_i) \neq y_i\}$

2. Add new learner to the ensemble:

Recap on AdaBoost

Adaboost follows this framework with $\ell(\hat{y}, y) = \exp(-\hat{y} \cdot y)$

1. Create a new weighted dataset:

For each x_i, compute $p_i \propto \exp(-\hat{y}_i \cdot y_i)$

Binary classification: $h_{t+1} = \arg\min_{h \in \mathscr{H}} \sum_i p_i \cdot \mathbf{1}\{h(x_i) \neq y_i\}$

2. Add new learner to the ensemble:

$$H_{t+1} = H_t + \frac{1}{2} \ln \frac{1 - \epsilon}{\epsilon} \cdot h_{t+1}$$

Outline of Today

1. Analysis of Boosting

2. Multilayer feedforward Neural Network

3. Training a neural network

The definition of Weak learning

Each weaker learning optimizes its own data:

$$\widetilde{\mathscr{D}} = \{p_i, x_i, y_i\}, \text{ where } \sum_i p_i = 1, p_i \geq 0, \forall i$$

$$h_{t+1} = \arg\min_{h \in \mathscr{H}} \sum_{i=1}^{n} p_i \cdot \mathbf{1}(h(x_i) \neq y_i)$$

The definition of Weak learning

Each weaker learning optimizes its own data:

$$\widetilde{\mathscr{D}} = \{p_i, x_i, y_i\}, \text{ where } \sum_i p_i = 1, p_i \geq 0, \forall i$$

$$h_{t+1} = \arg\min_{h \in \mathscr{H}} \sum_{i=1}^{n} p_i \cdot \mathbf{1}(h(x_i) \neq y_i)$$

Assume that weaker learner's loss $\epsilon := \sum_{i=1}^{n} p_i \mathbf{1}\{h_{t+1}(x_i) \neq y_i\} \leq \frac{1}{2} - \gamma, \;\; \gamma > 0$

The definition of Weak learning

Each weaker learning optimizes its own data:

$$\widetilde{\mathscr{D}} = \{p_i, x_i, y_i\}, \text{ where } \sum_i p_i = 1, p_i \geq 0, \forall i$$

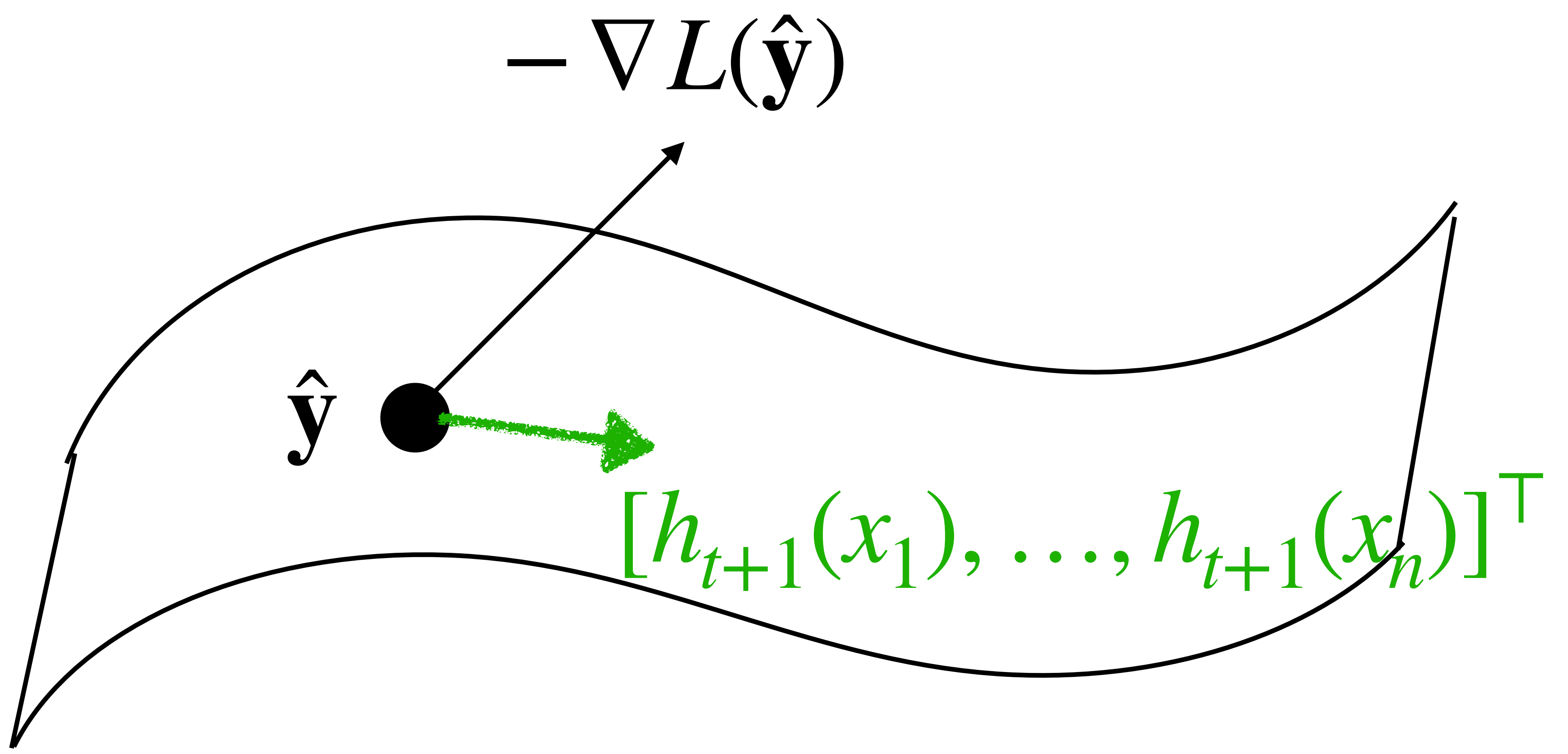
$$h_{t+1} = \arg\min_{h \in \mathscr{H}} \sum_{i=1}^{n} p_i \cdot \mathbf{1}(h(x_i) \neq y_i)$$

Assume that weaker learner's loss $\epsilon := \sum_{i=1}^{n} p_i \mathbf{1}\{h_{t+1}(x_i) \neq y_i\} \leq \frac{1}{2} - \gamma, \ \ \gamma > 0$

Q: assume $\mathscr{H}$ is symmetric, i.e., $h \in \mathscr{H}$ iff $-h \in \mathscr{H}$, why does the above always hold?

Weaker learnability implies approximating gradient well

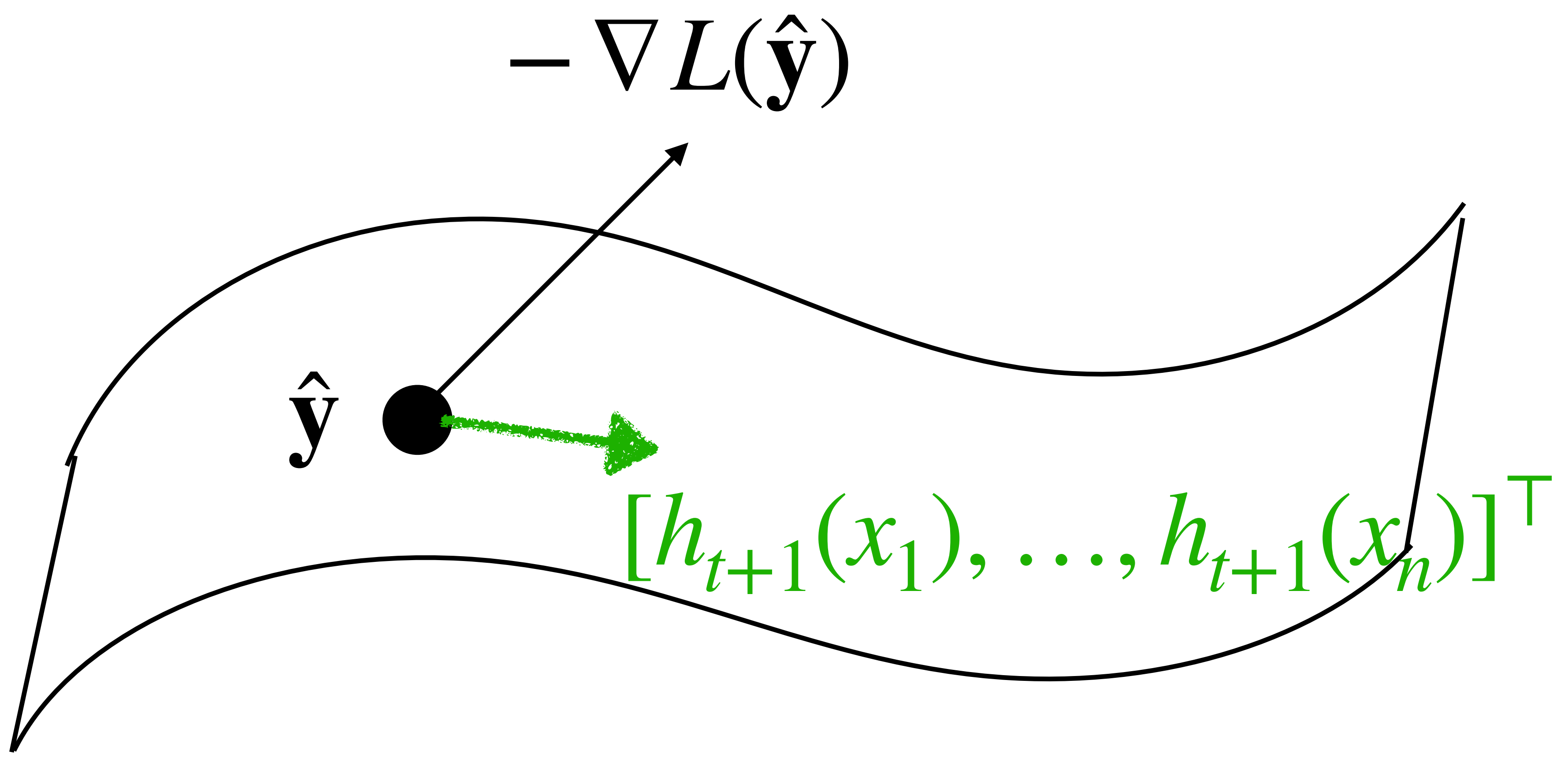
Assume that weaker learner's loss $\epsilon := \sum_{i=1}^{n} p_i \mathbf{1}\{h_{t+1}(x_i) \neq y_i\} \leq \frac{1}{2} - \gamma, \;\; \gamma > 0$



Weaker learnability implies approximating gradient well

Assume that weaker learner's loss $\epsilon := \sum_{i=1}^{n} p_i \mathbf{1}\{h_{t+1}(x_i) \neq y_i\} \leq \frac{1}{2} - \gamma, \;\; \gamma > 0$

$$(-\nabla L(\hat{\mathbf{y}}))^\top \begin{bmatrix} h_{t+1}(x_1) \\ \cdots \\ h_{h+1}(x_n) \end{bmatrix}$$

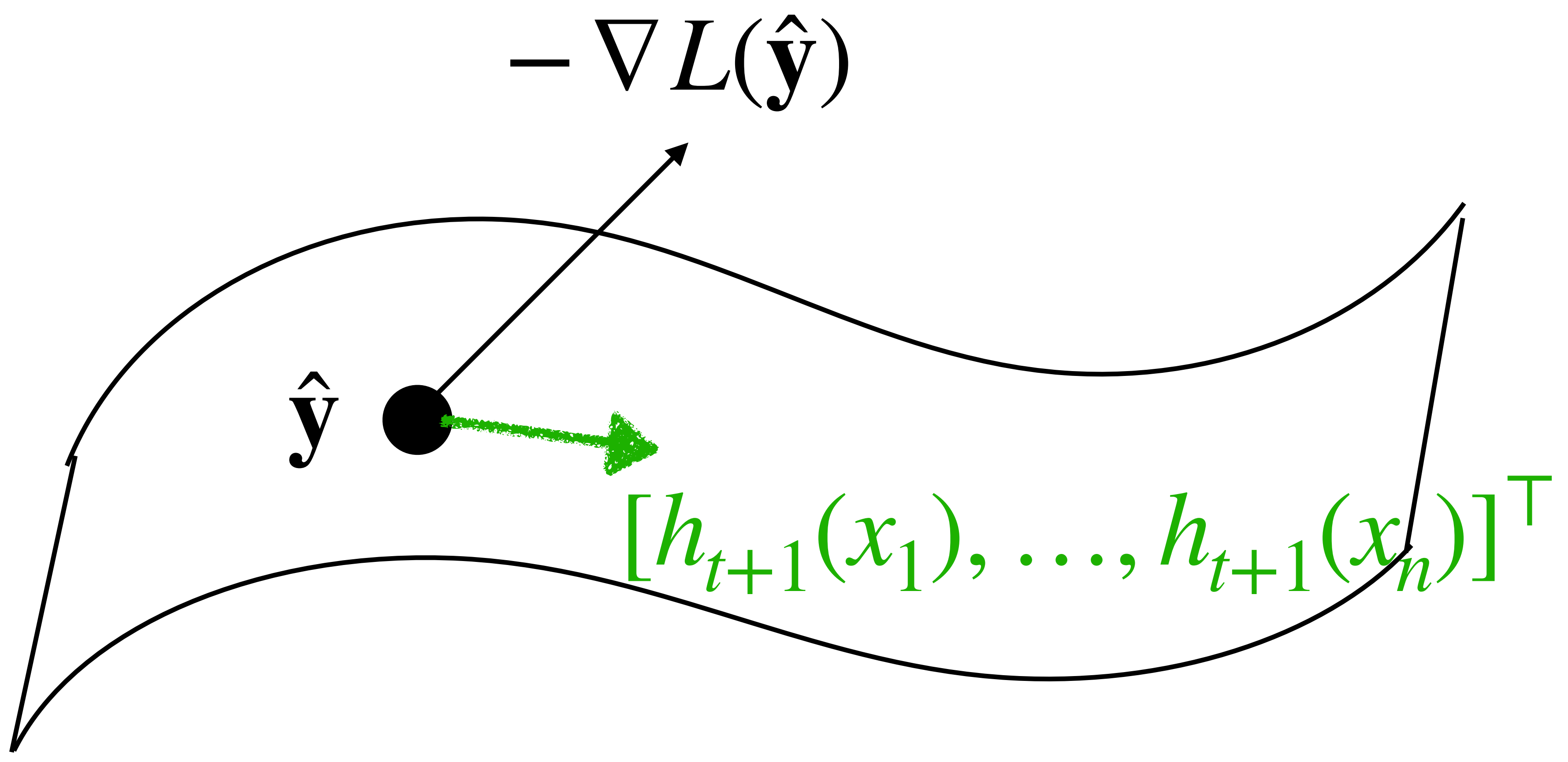


Weaker learnability implies approximating gradient well

Assume that weaker learner's loss $\epsilon := \sum_{i=1}^{n} p_i \mathbf{1}\{h_{t+1}(x_i) \neq y_i\} \leq \frac{1}{2} - \gamma, \;\; \gamma > 0$

$$(-\nabla L(\hat{\mathbf{y}}))^\top \begin{bmatrix} h_{t+1}(x_1) \\ \cdots \\ h_{h+1}(x_n) \end{bmatrix}$$

$$\geq (\sum_{j=1}^{n} |w_j|) 2\gamma > 0$$

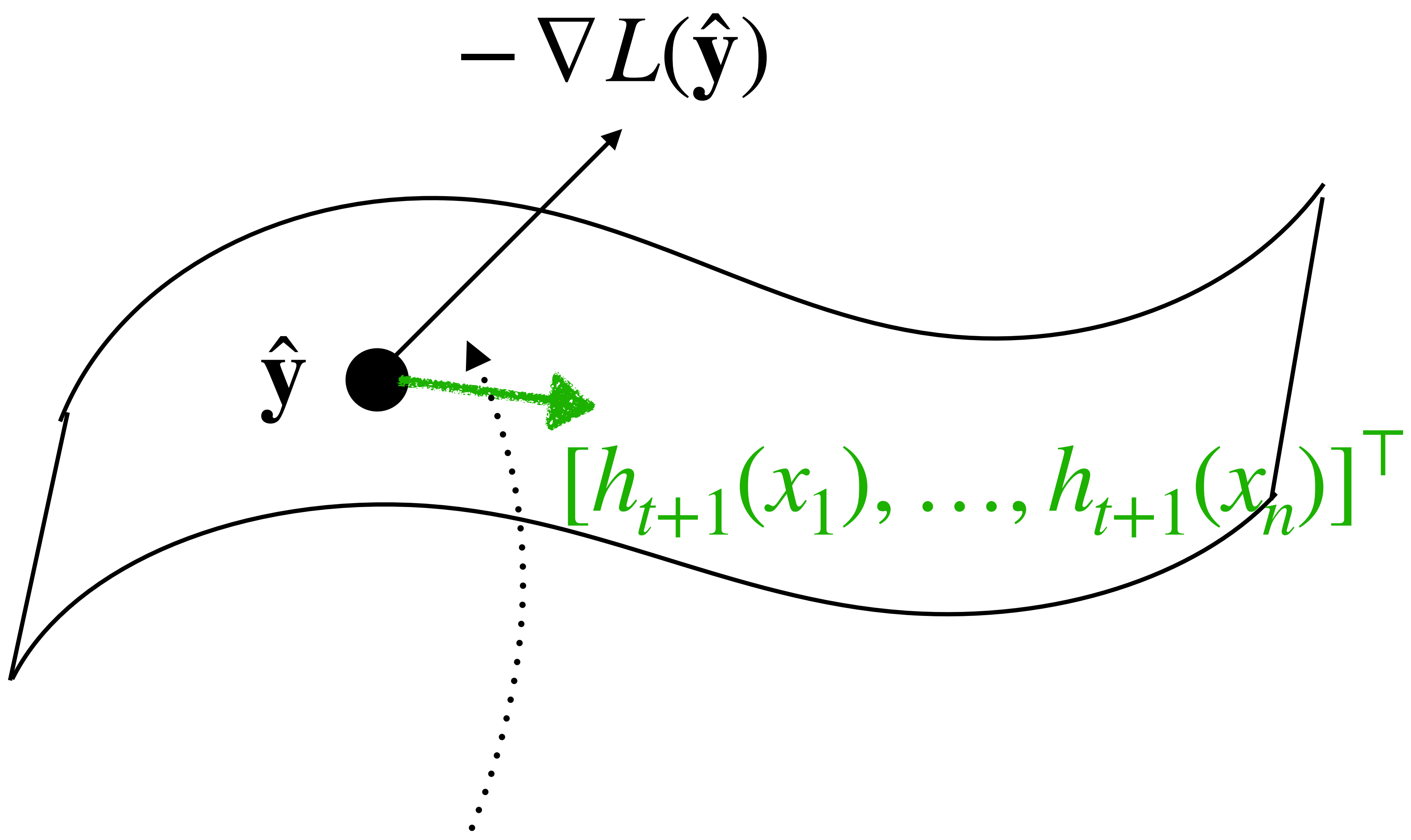


Weaker learnability implies approximating gradient well

Assume that weaker learner's loss $\epsilon := \sum_{i=1}^{n} p_i \mathbf{1}\{h_{t+1}(x_i) \neq y_i\} \leq \frac{1}{2} - \gamma, \;\; \gamma > 0$

$$(-\nabla L(\hat{\mathbf{y}}))^\top \begin{bmatrix} h_{t+1}(x_1) \\ \cdots \\ h_{h+1}(x_n) \end{bmatrix}$$

$$\geq (\sum_{j=1}^{n} |w_j|) 2\gamma > 0$$



Within 90 degree, so
improve the objective!

Formal Convergence of AdaBoost

Then after T iterations, for the original exp loss, we have

$$\frac{1}{n}\sum_{i=1}^{n}\exp(-H_T(x_i)\cdot y_i) \leq n(1-4\gamma^2)^{T/2}$$

(Proof in lecture note, optional)

Formal Convergence of AdaBoost

Then after T iterations, for the original exp loss, we have

$$\frac{1}{n}\sum_{i=1}^{n}\exp(-H_T(x_i)\cdot y_i) \leq n(1-4\gamma^2)^{T/2}$$

Note zero-one loss is upper bounded by exponential loss

(Proof in lecture note, optional)

Formal Convergence of AdaBoost

Then after T iterations, for the original exp loss, we have

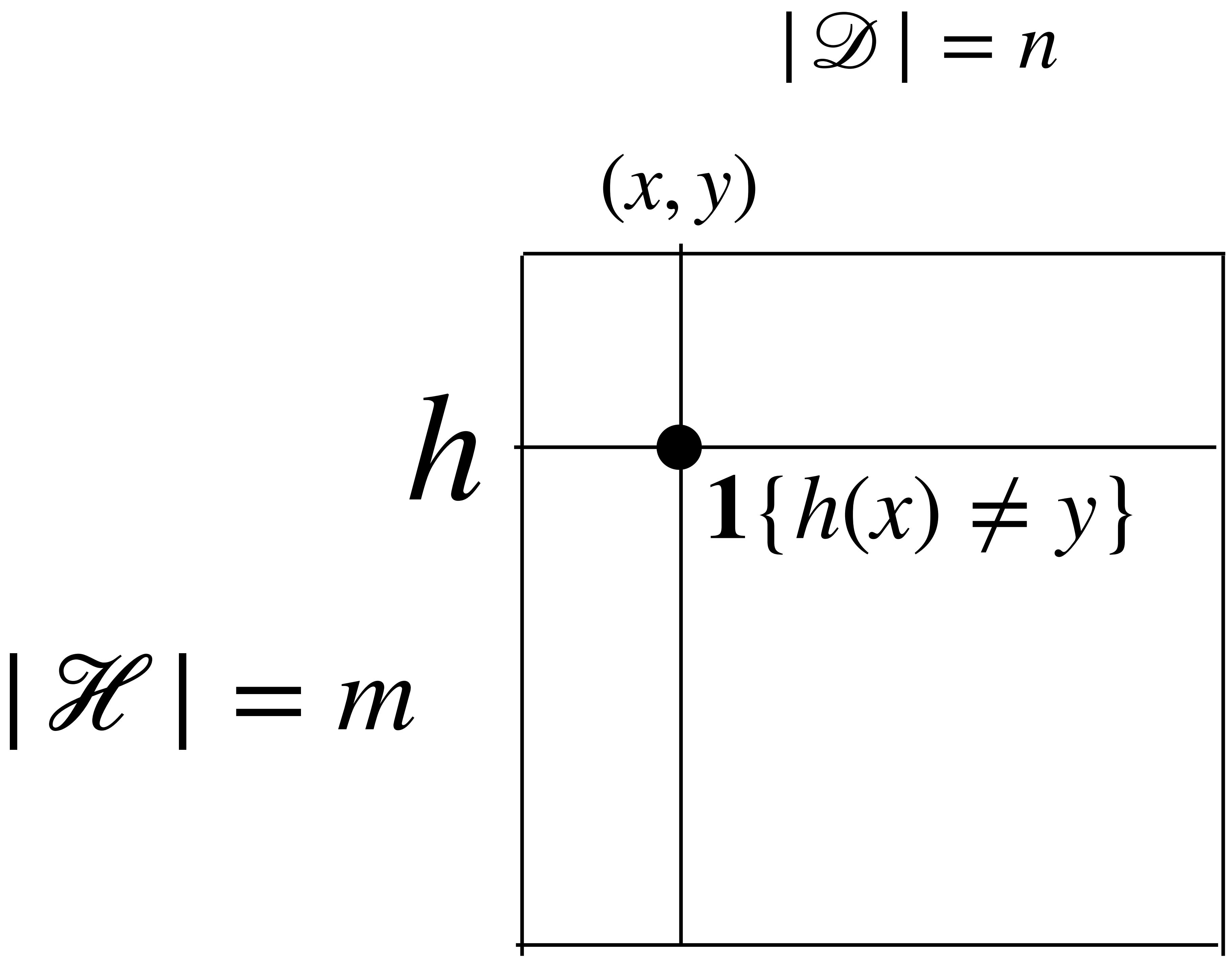
$$\frac{1}{n}\sum_{i=1}^{n}\exp(-H_T(x_i)\cdot y_i) \leq n(1-4\gamma^2)^{T/2}$$

Note zero-one loss is upper bounded by exponential loss

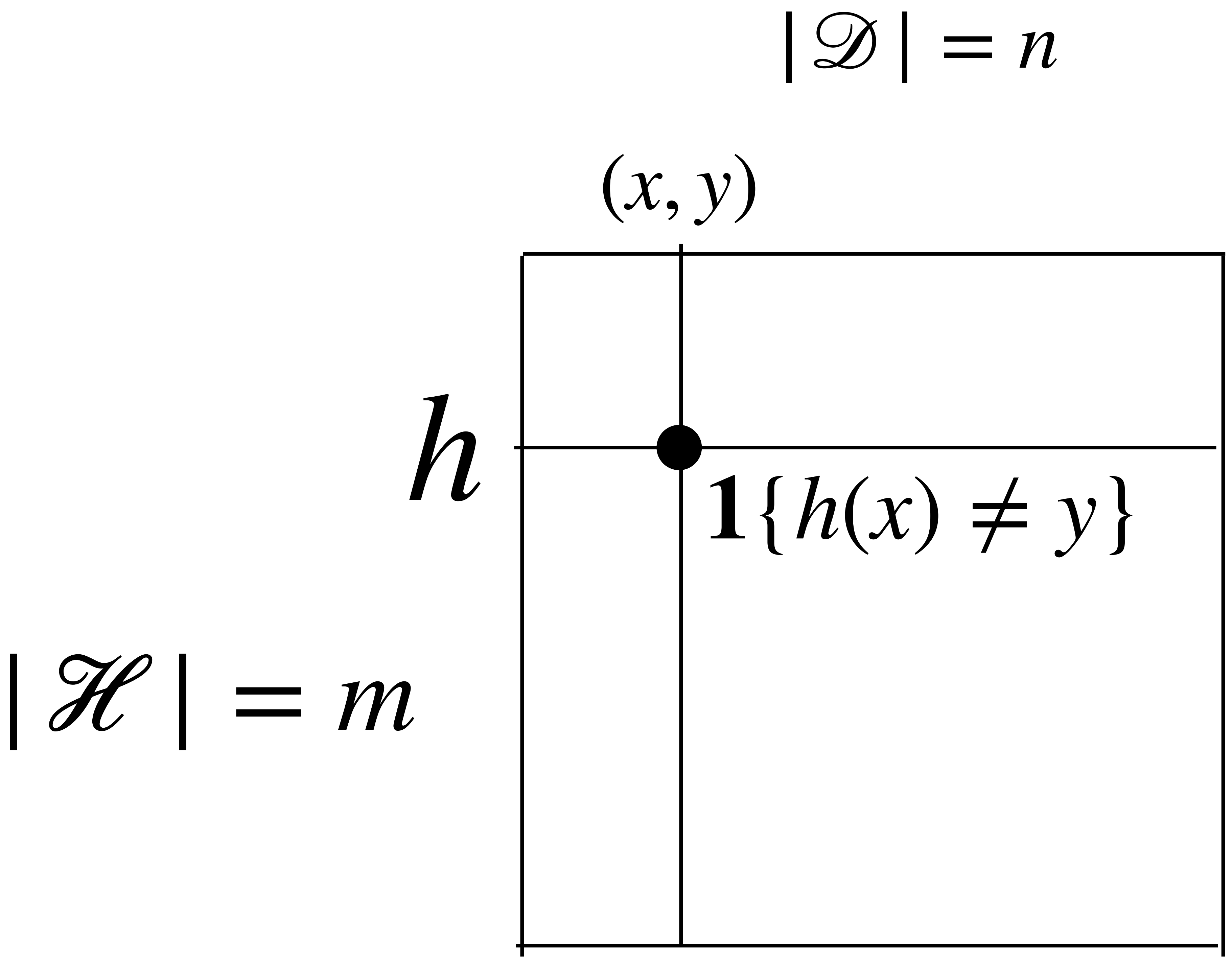
$$\frac{1}{n}\sum_{i=1}^{n}\mathbf{1}\{\mathrm{sign}(H_T(x_i)) \neq y_i\} \leq \frac{1}{n}\sum_{i=1}^{n}\exp(-H_T(x_i)\cdot y_i) \leq n(1-4\gamma^2)^{T/2}$$

(Proof in lecture note, optional)

Thinking about Boosting via two player zero sum game



Thinking about Boosting via two player zero sum game



$|\mathscr{D}| = n$

(x, y)

h

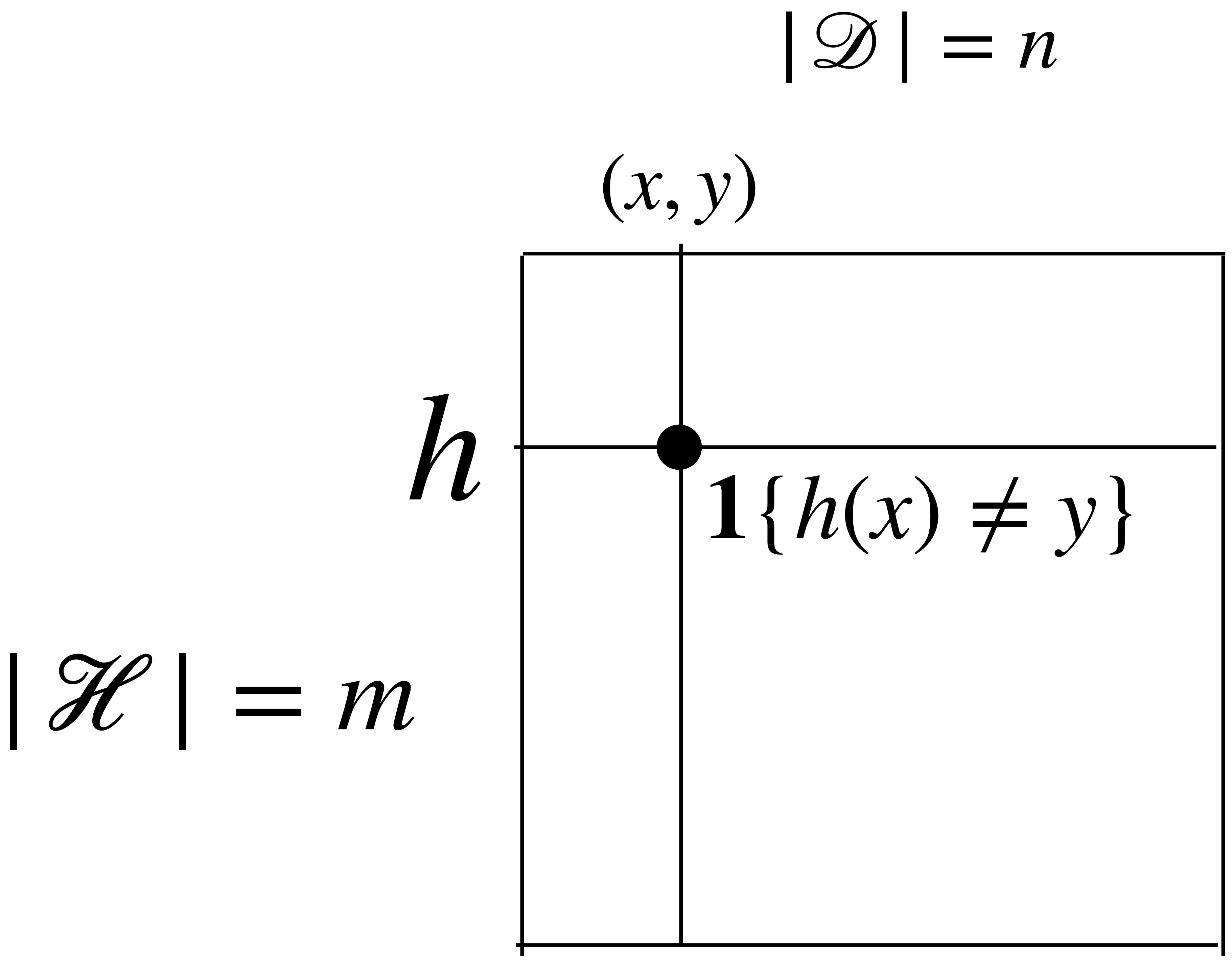
$\mathbf{1}\{h(x) \neq y\}$

$|\mathscr{H}| = m$

Row player plays hypothesis $h \in \mathscr{H}$

Column player plays example (x, y)

Thinking about Boosting via two player zero sum game

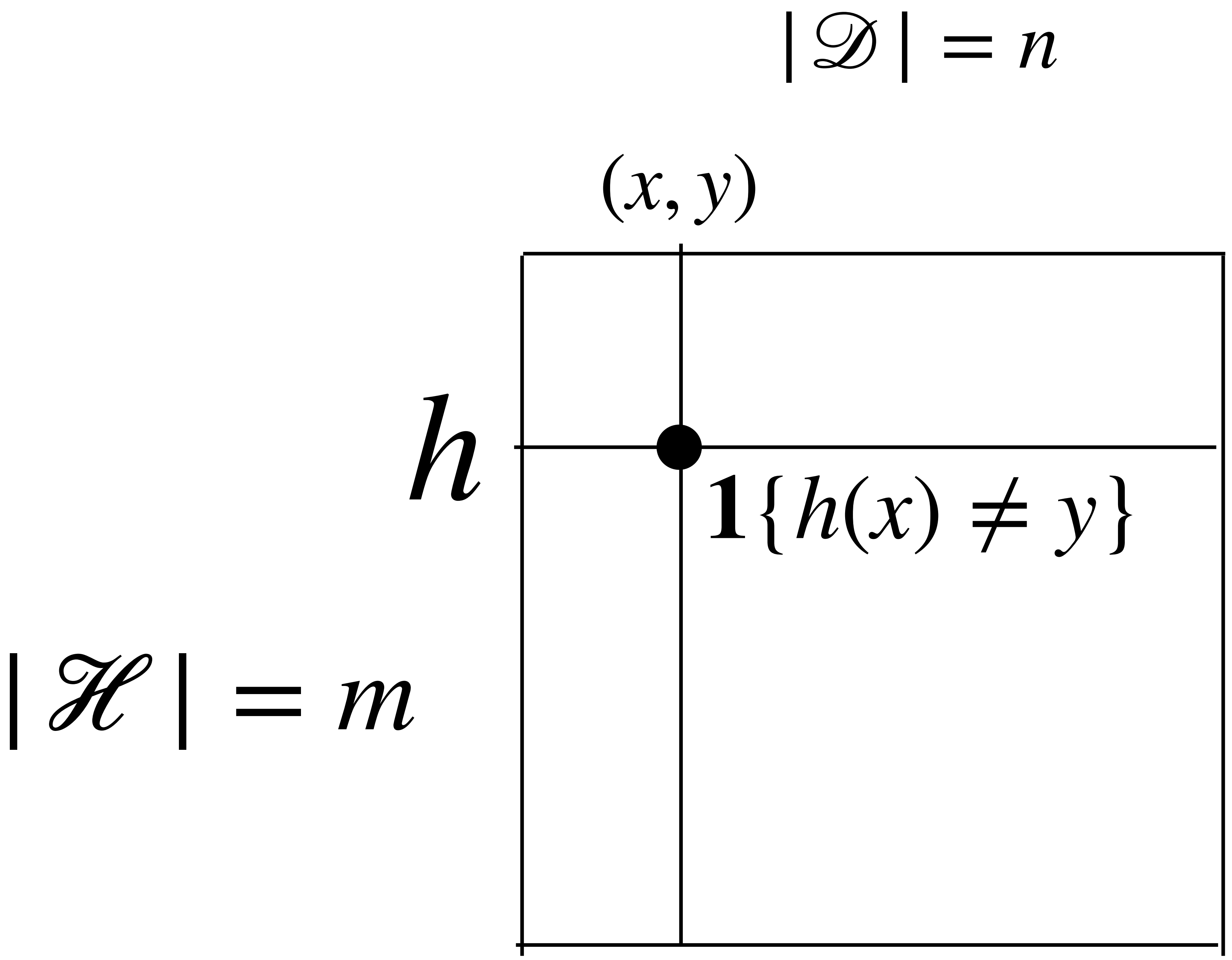


Row player plays hypothesis $h \in \mathscr{H}$

Column player plays example (x, y)

Row player gets loss $\mathbf{1}\{h(x) \neq y\}$

Thinking about Boosting via two player zero sum game



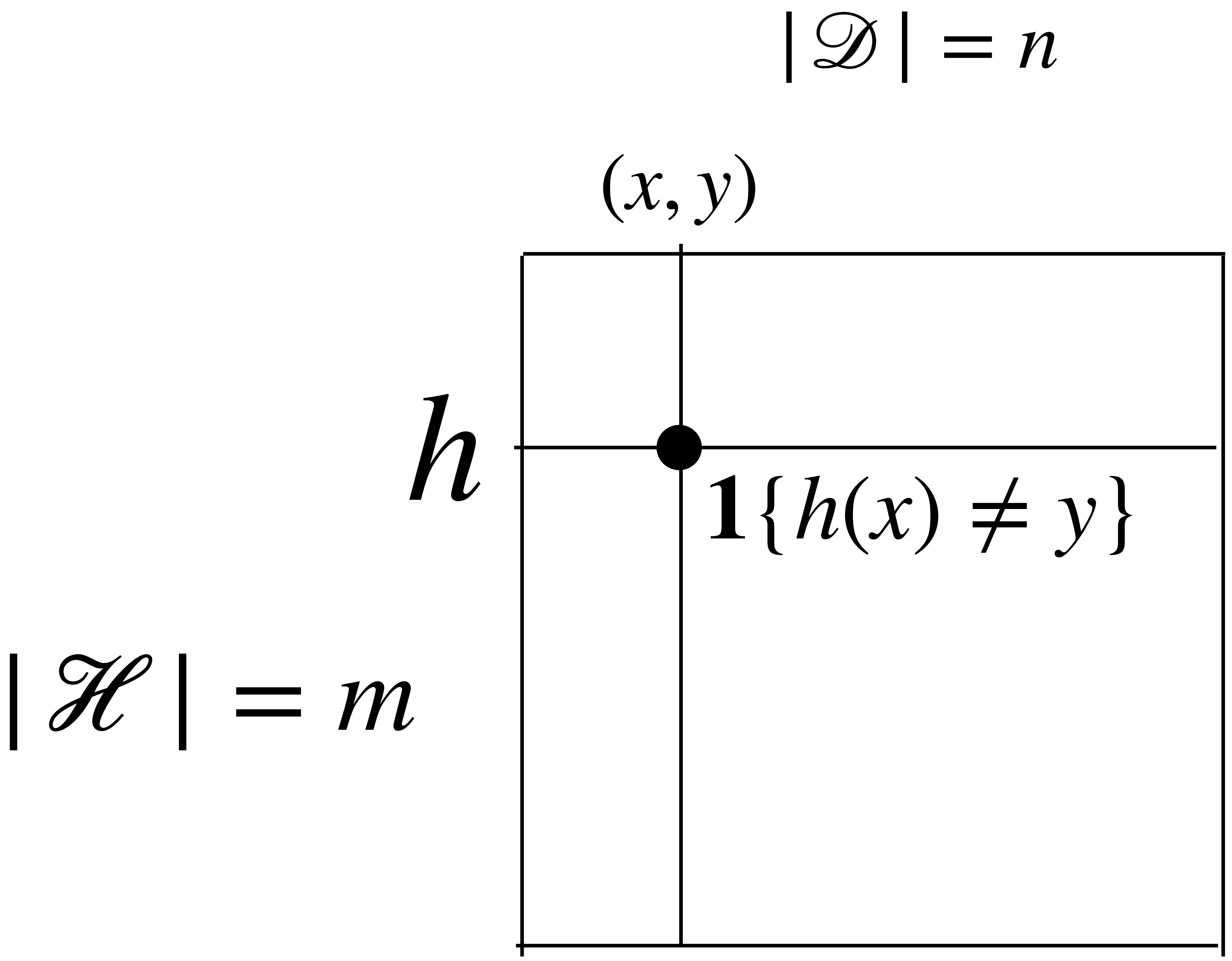
Row player plays hypothesis $h \in \mathscr{H}$

Column player plays example (x, y)

Row player gets loss $\mathbf{1}\{h(x) \neq y\}$

Column player gets loss $-\mathbf{1}\{h(x) \neq y\}$

Thinking about Boosting via two player zero sum game



Row player plays hypothesis $h \in \mathscr{H}$

Column player plays example (x, y)

Row player gets loss $\mathbf{1}\{h(x) \neq y\}$

Column player gets loss $-\mathbf{1}\{h(x) \neq y\}$

Boosting can be understood as running some specific algorithm to find the Nash equilibrium of the game

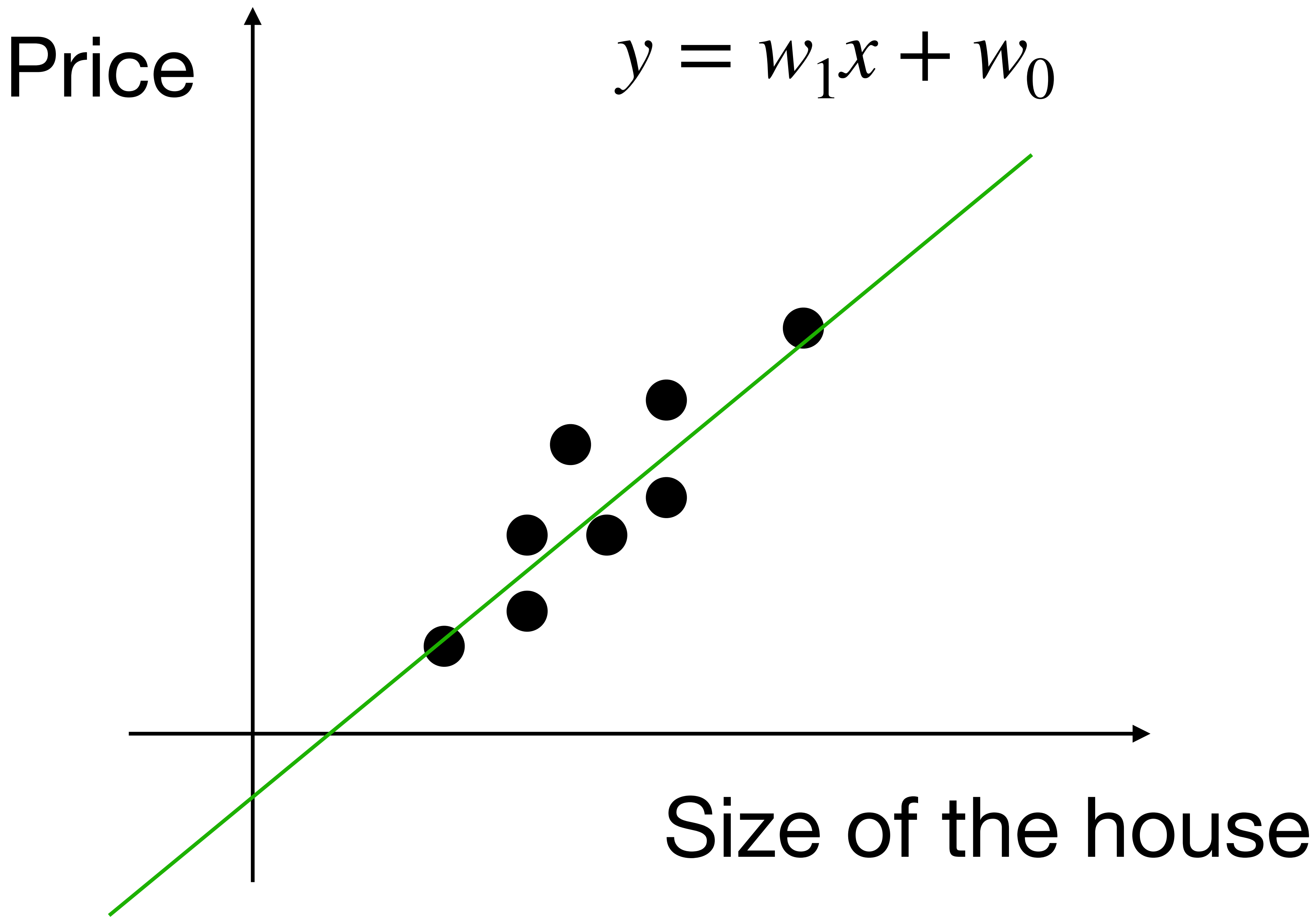
Outline of Today

1. Analysis of Boosting

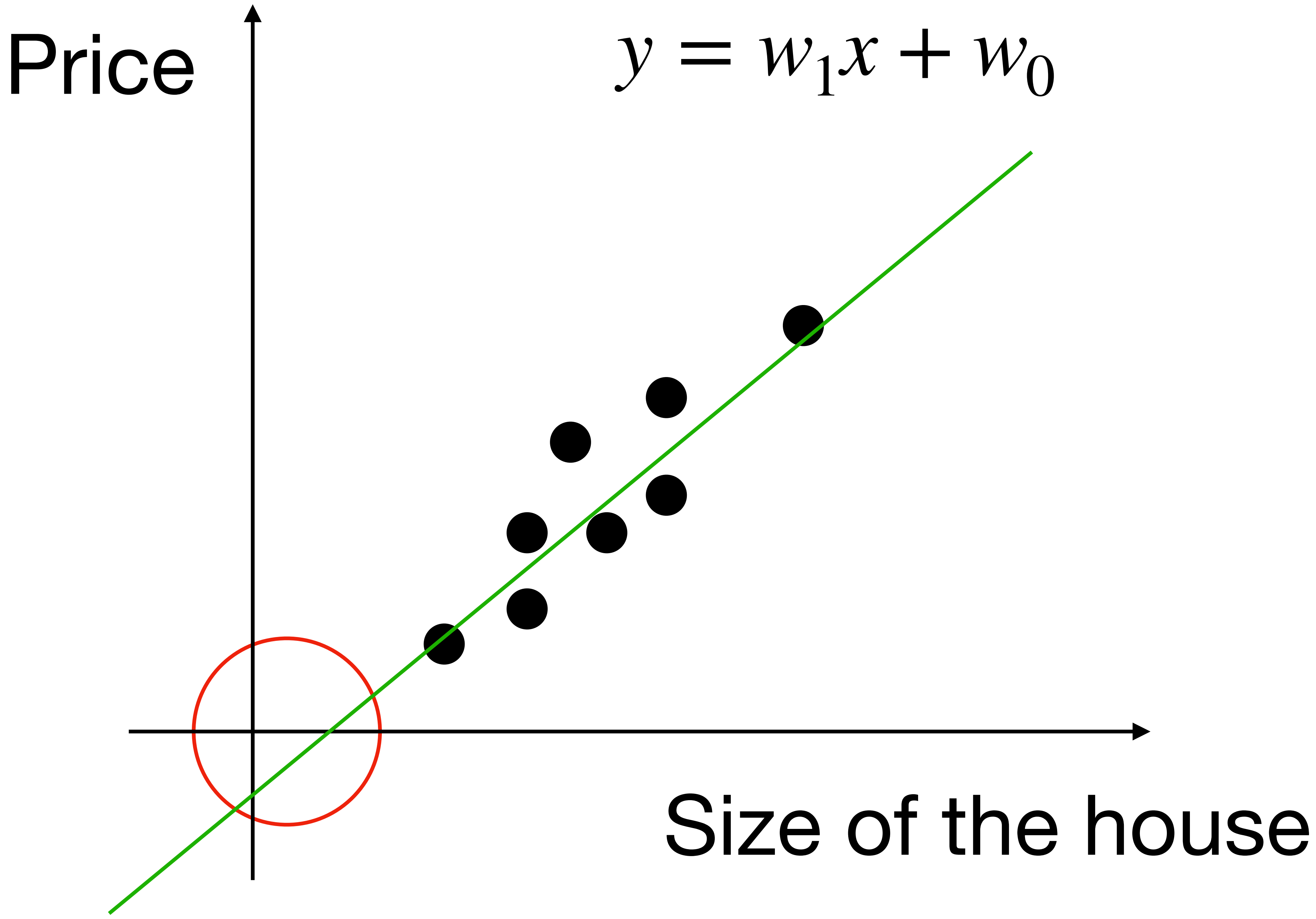
2. Multilayer feedforward Neural Network

3. Training a neural network

Linear Regression Revisit

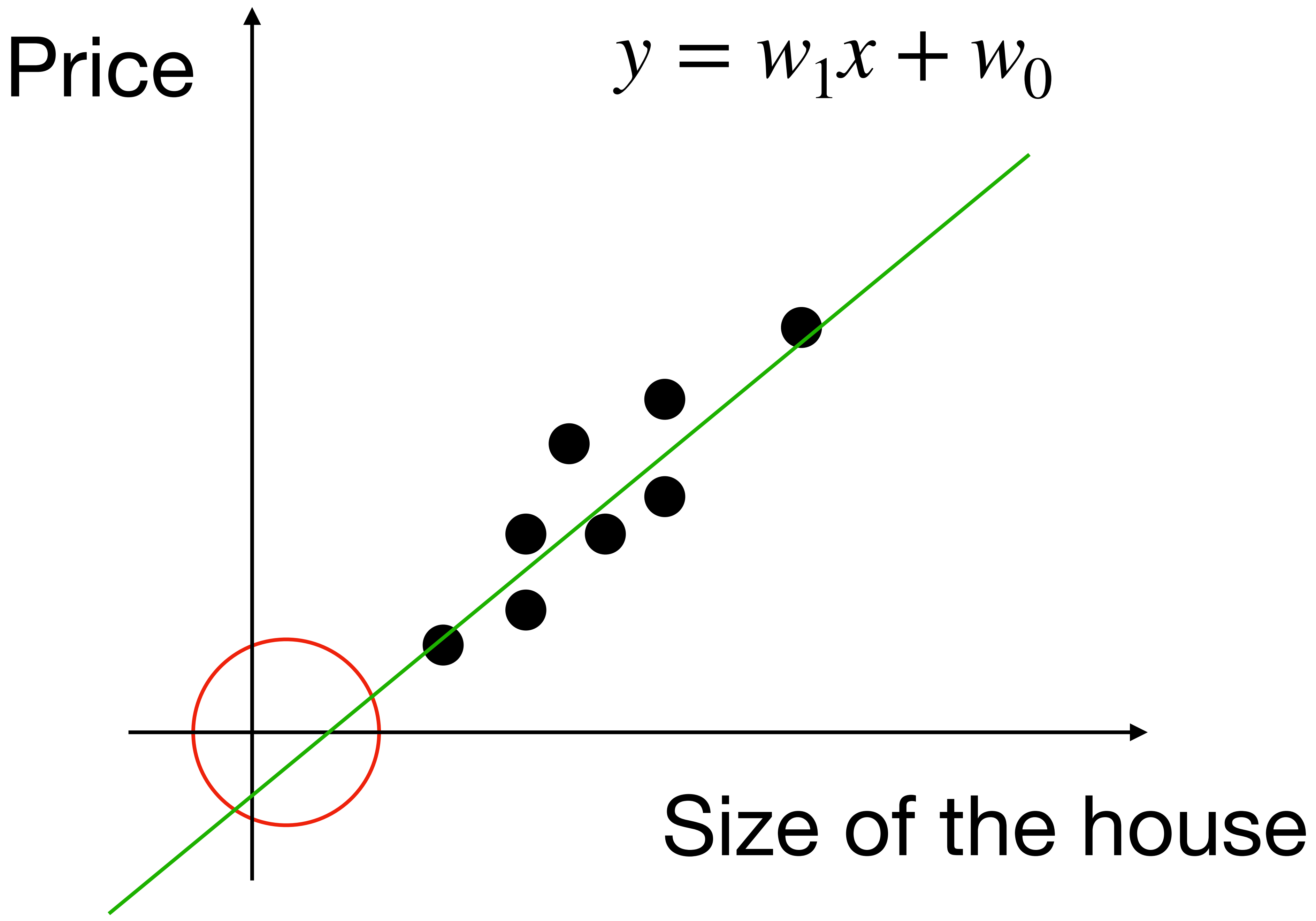


Linear Regression Revisit



Negative part does not make too much sense

Linear Regression Revisit

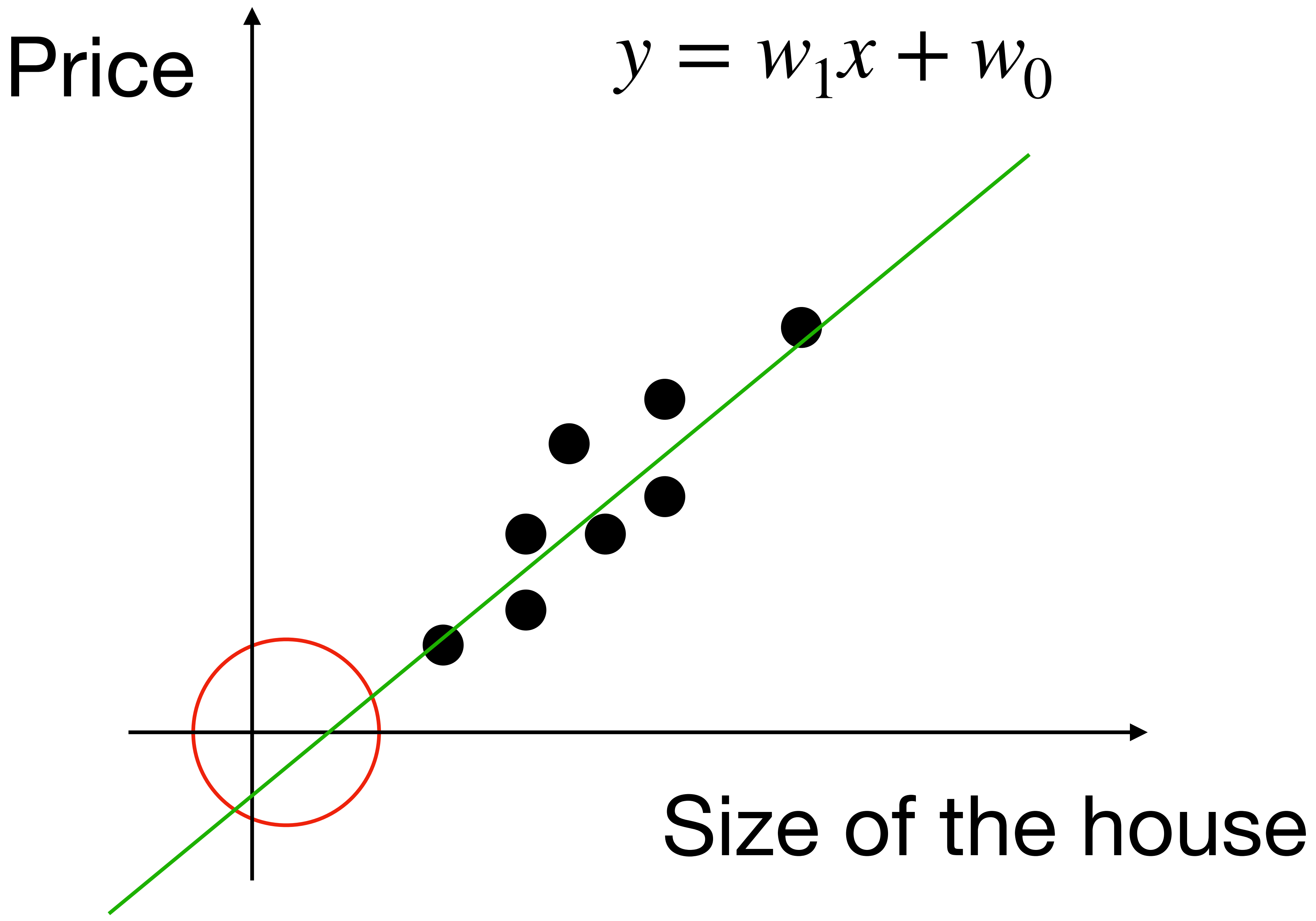


Negative part does not make too much sense

We can fix this with a simple nonlinear function

$$y = \max\{w_1 x + w_0, 0\}$$

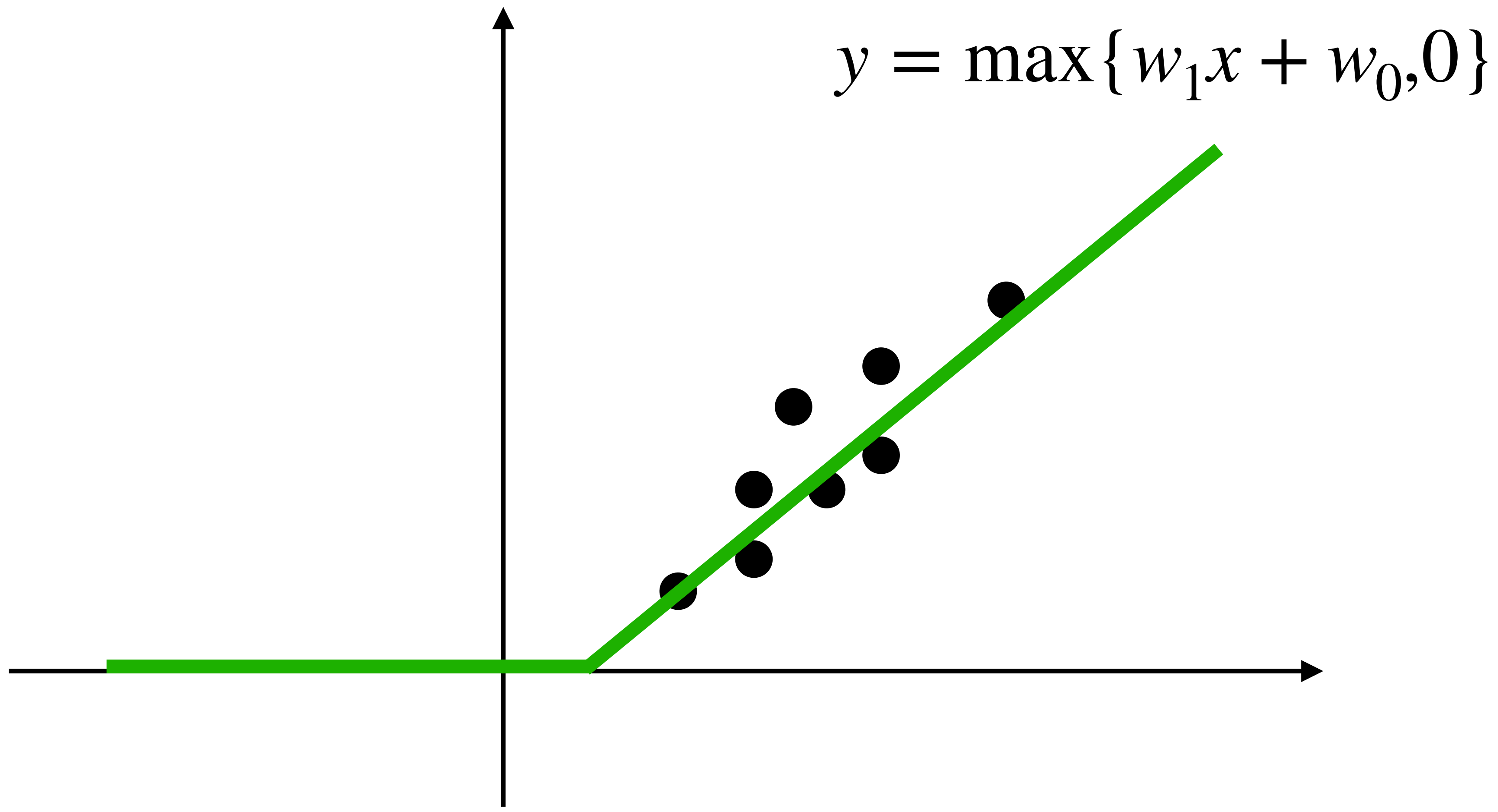
Linear Regression Revisit



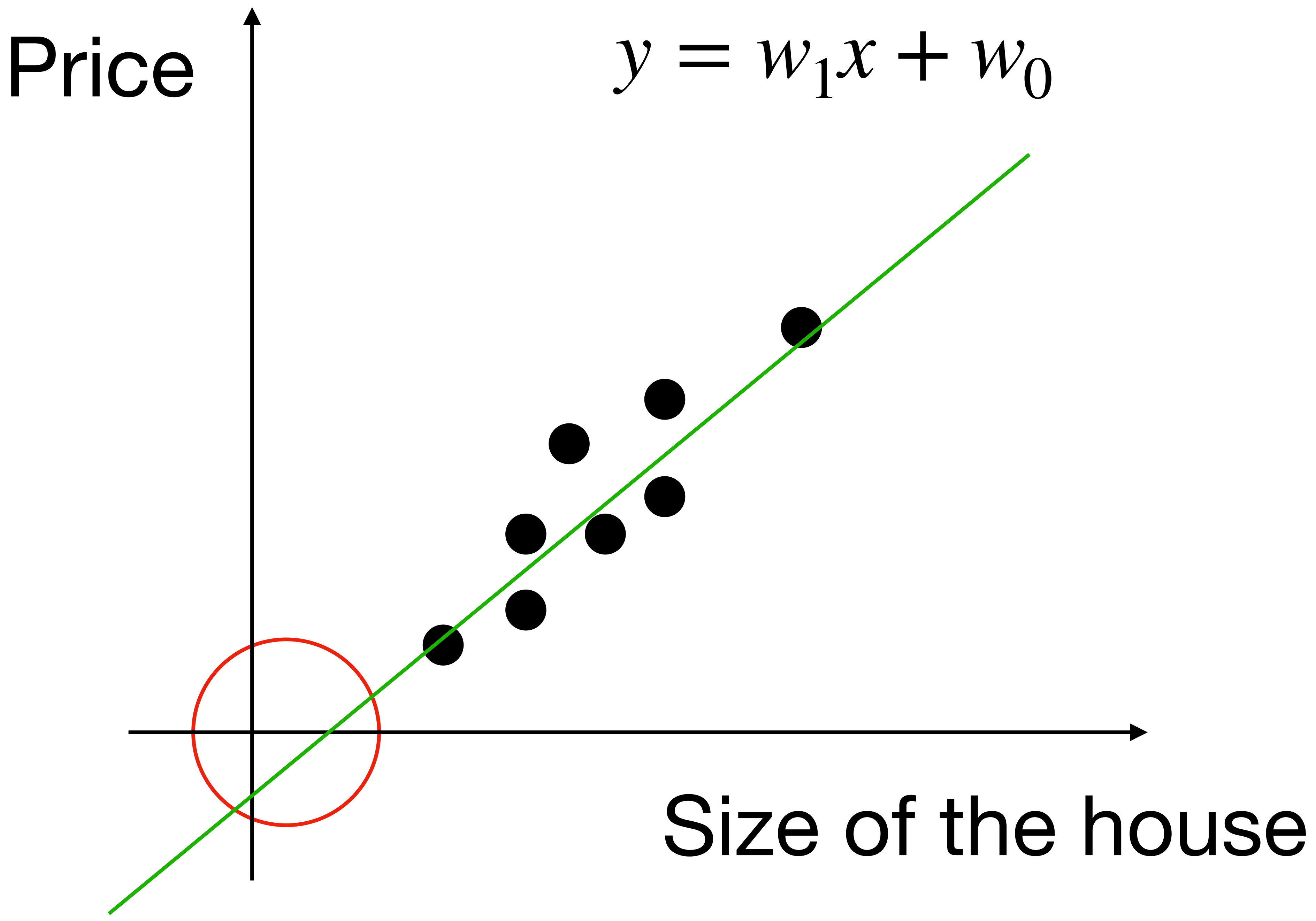
Negative part does not make too much sense

We can fix this with a simple nonlinear function

$$y = \max\{w_1x + w_0, 0\}$$



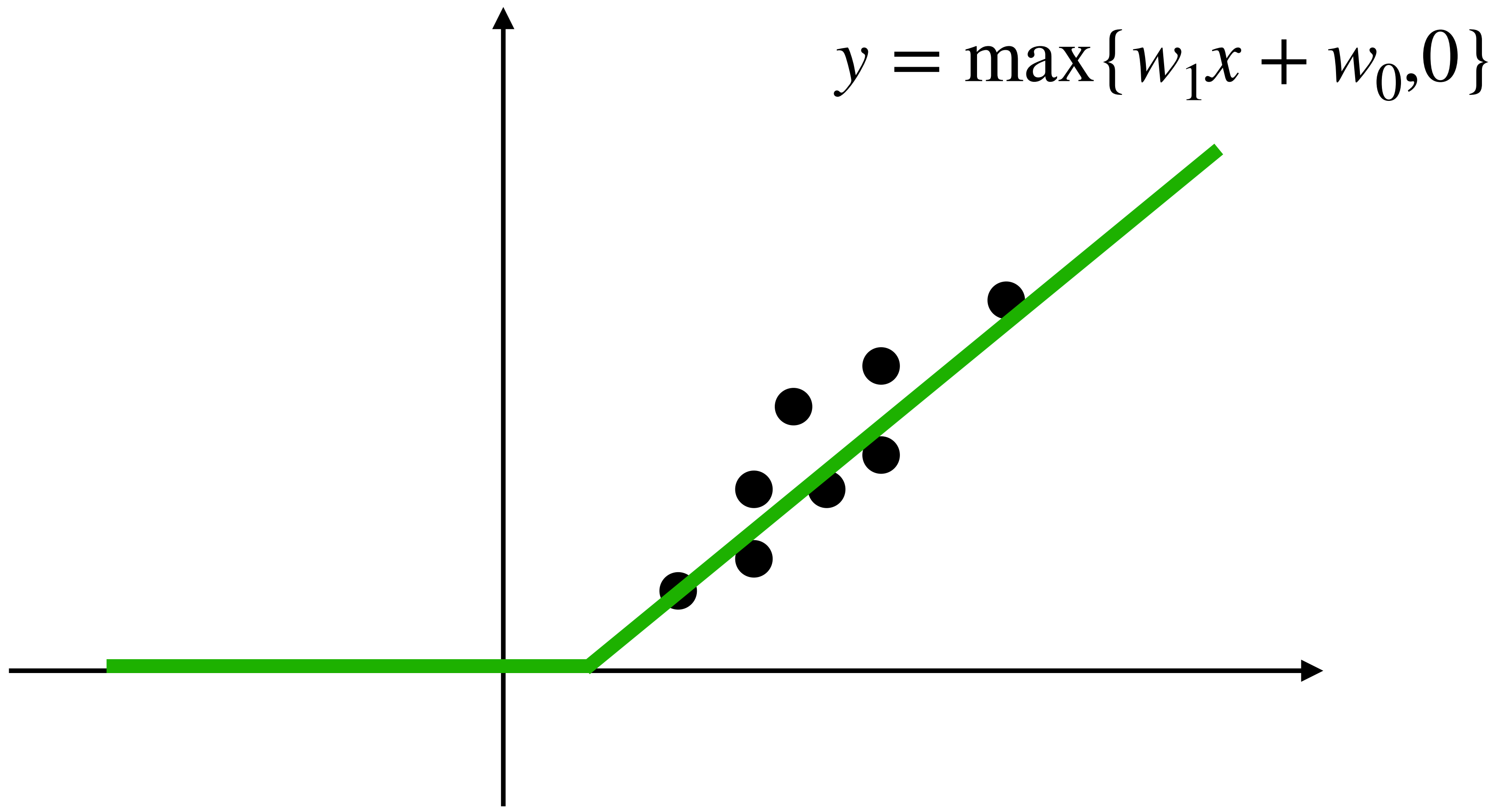
Linear Regression Revisit



Negative part does not make too much sense

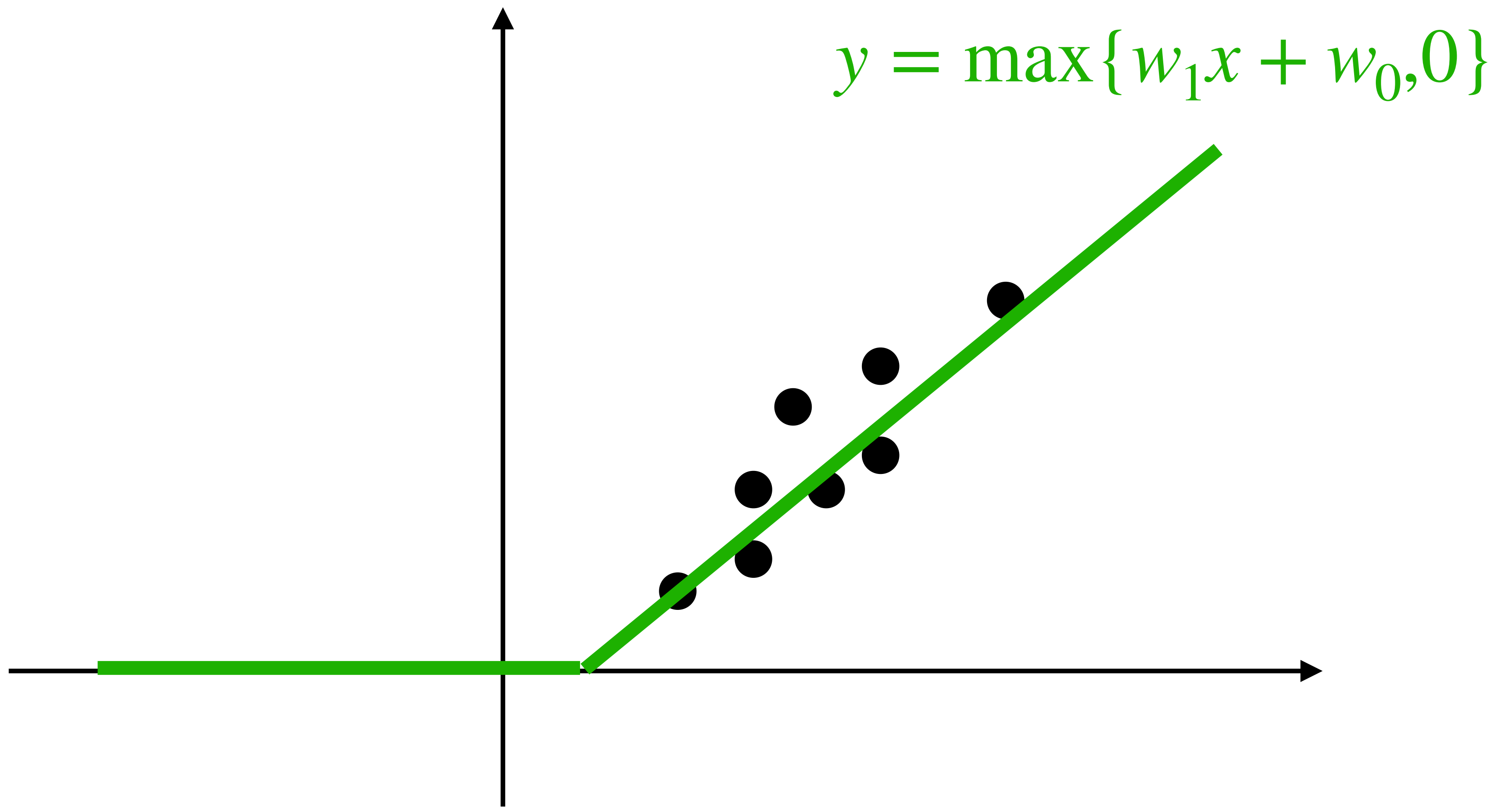
We can fix this with a simple nonlinear function

$$y = \max\{w_1x + w_0, 0\}$$

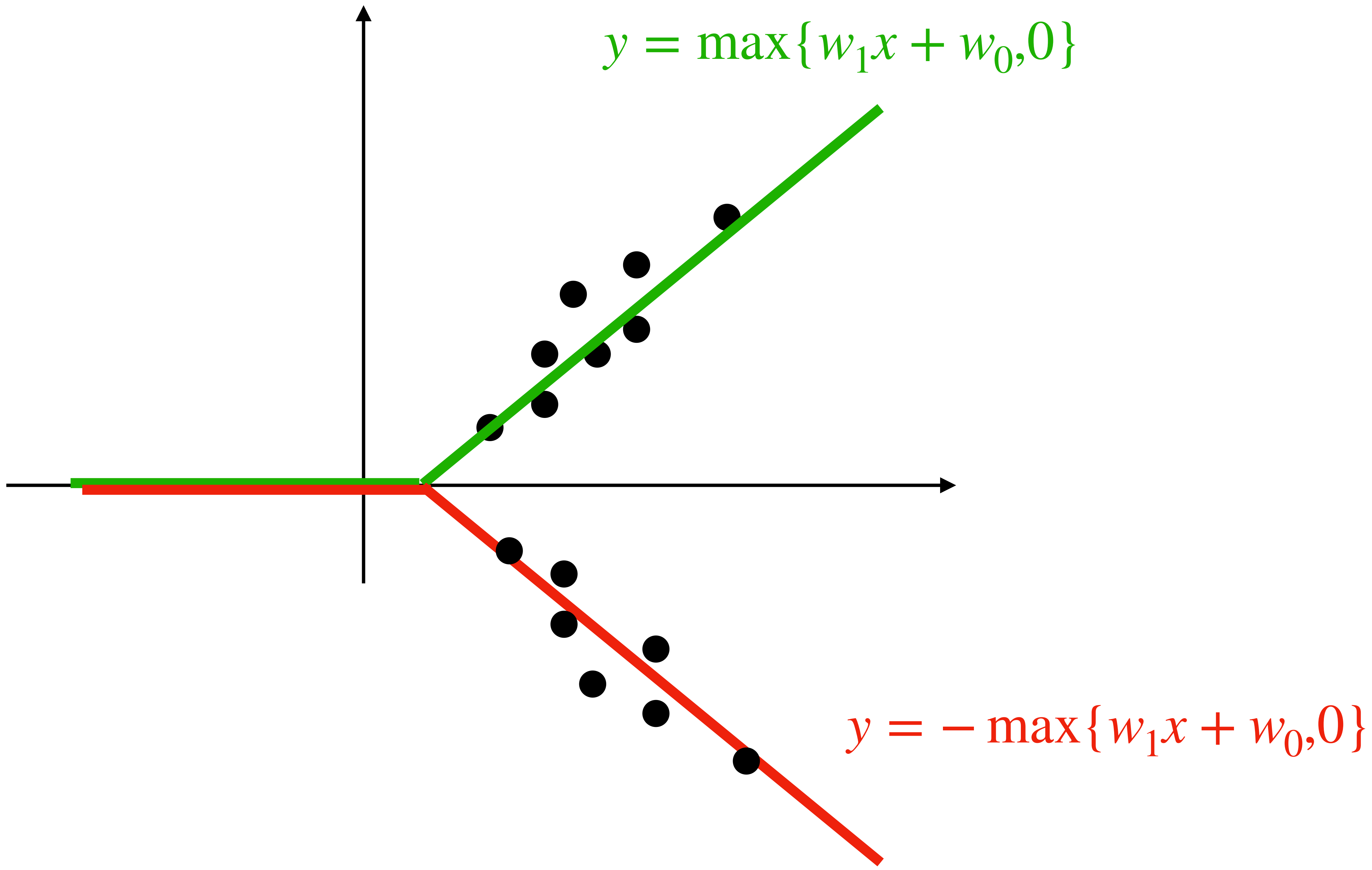


rectified linear unit (ReLU)

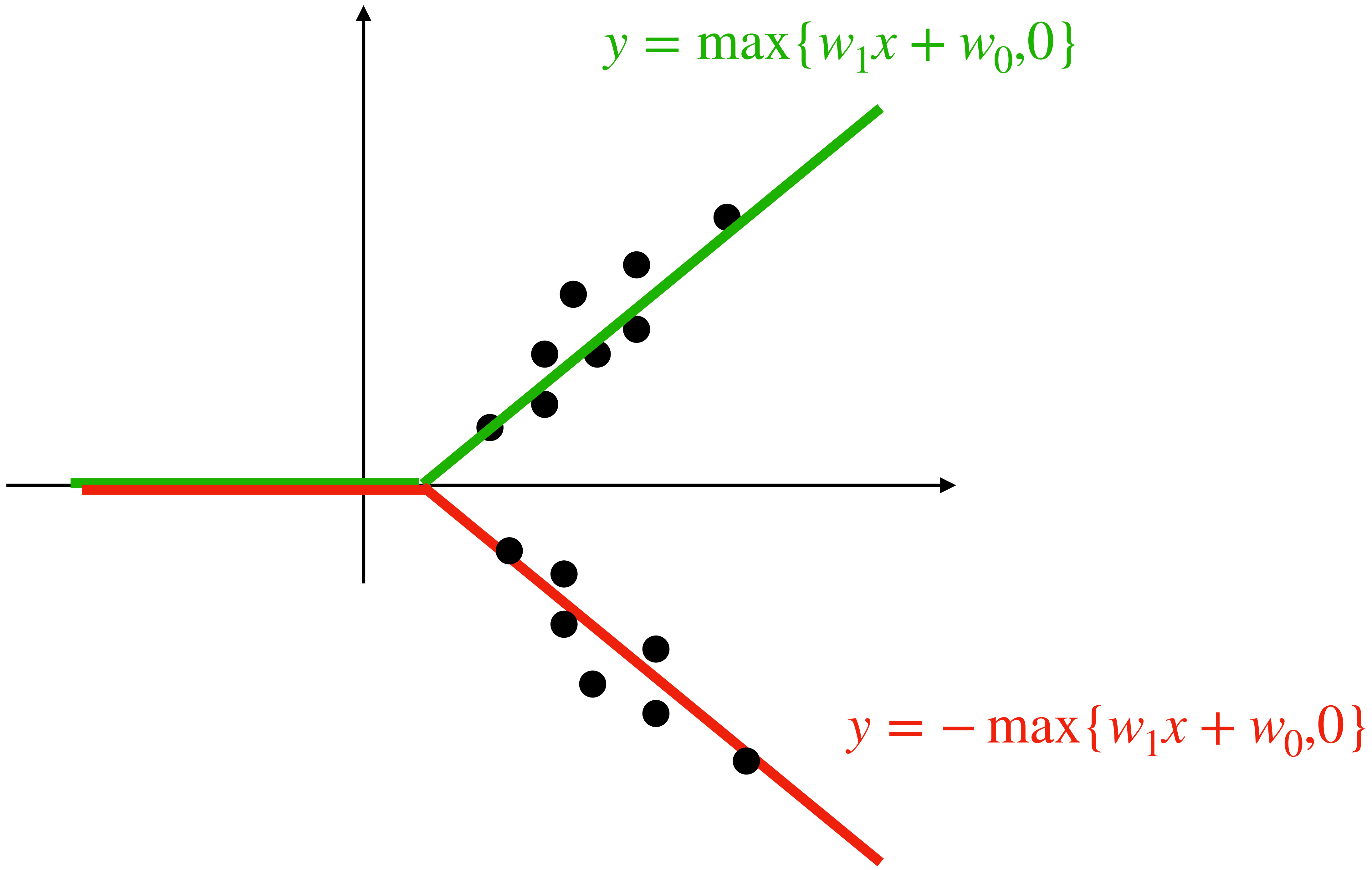
A single neuron network



A single neuron network

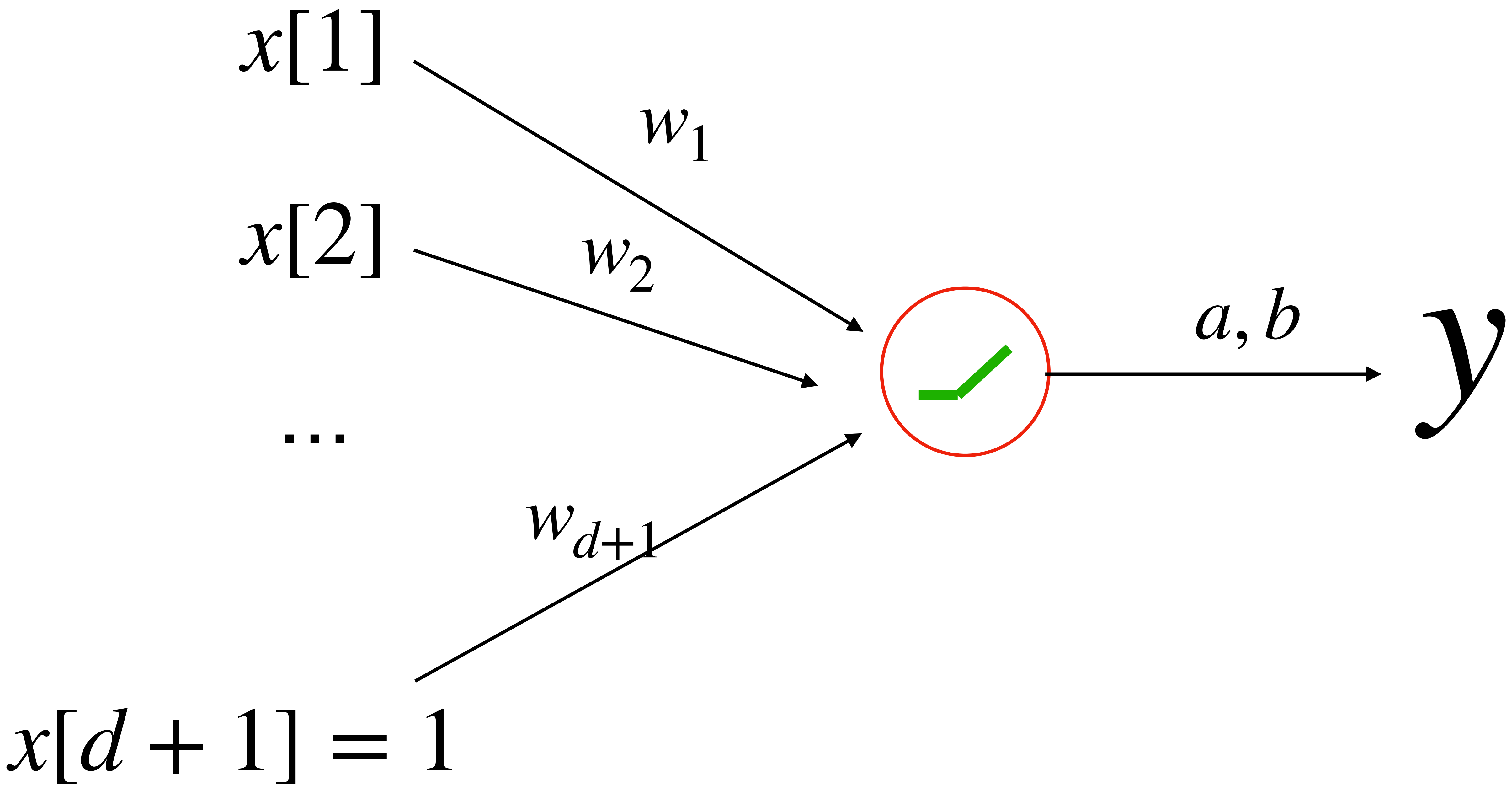


A single neuron network

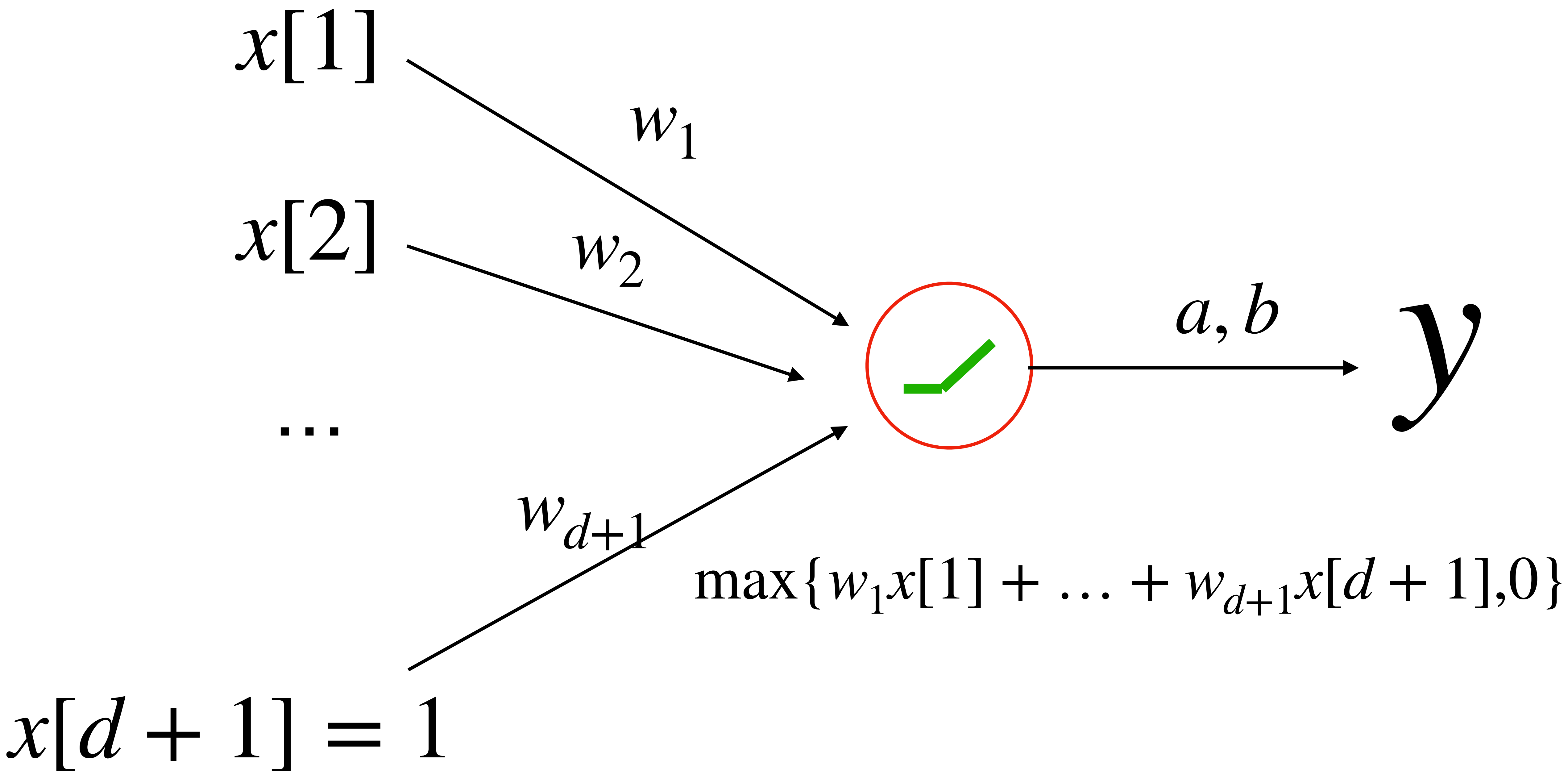


$$y = a\max\{w_1x + w_0, 0\} + b$$

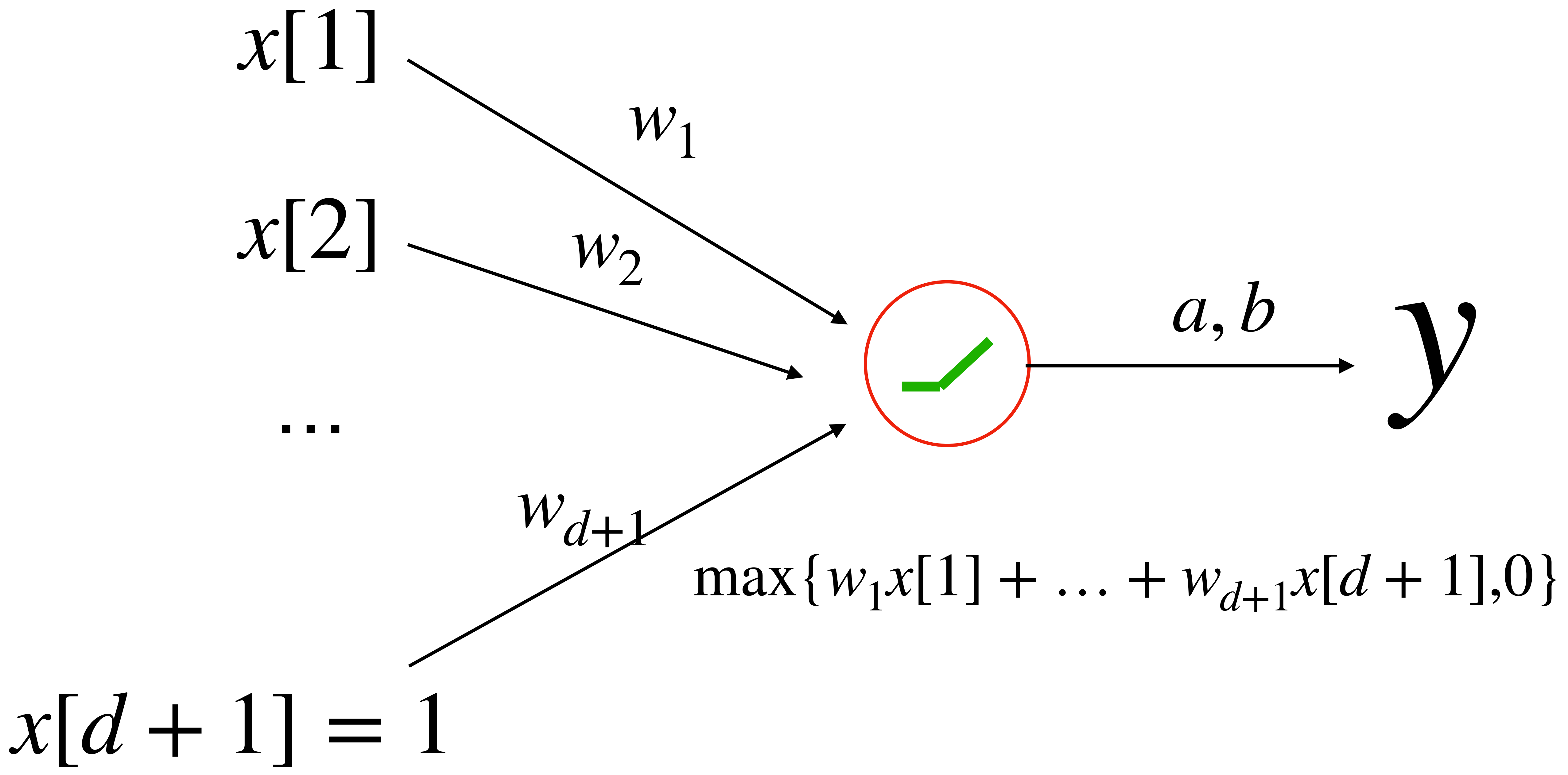
A single neuron network



A single neuron network

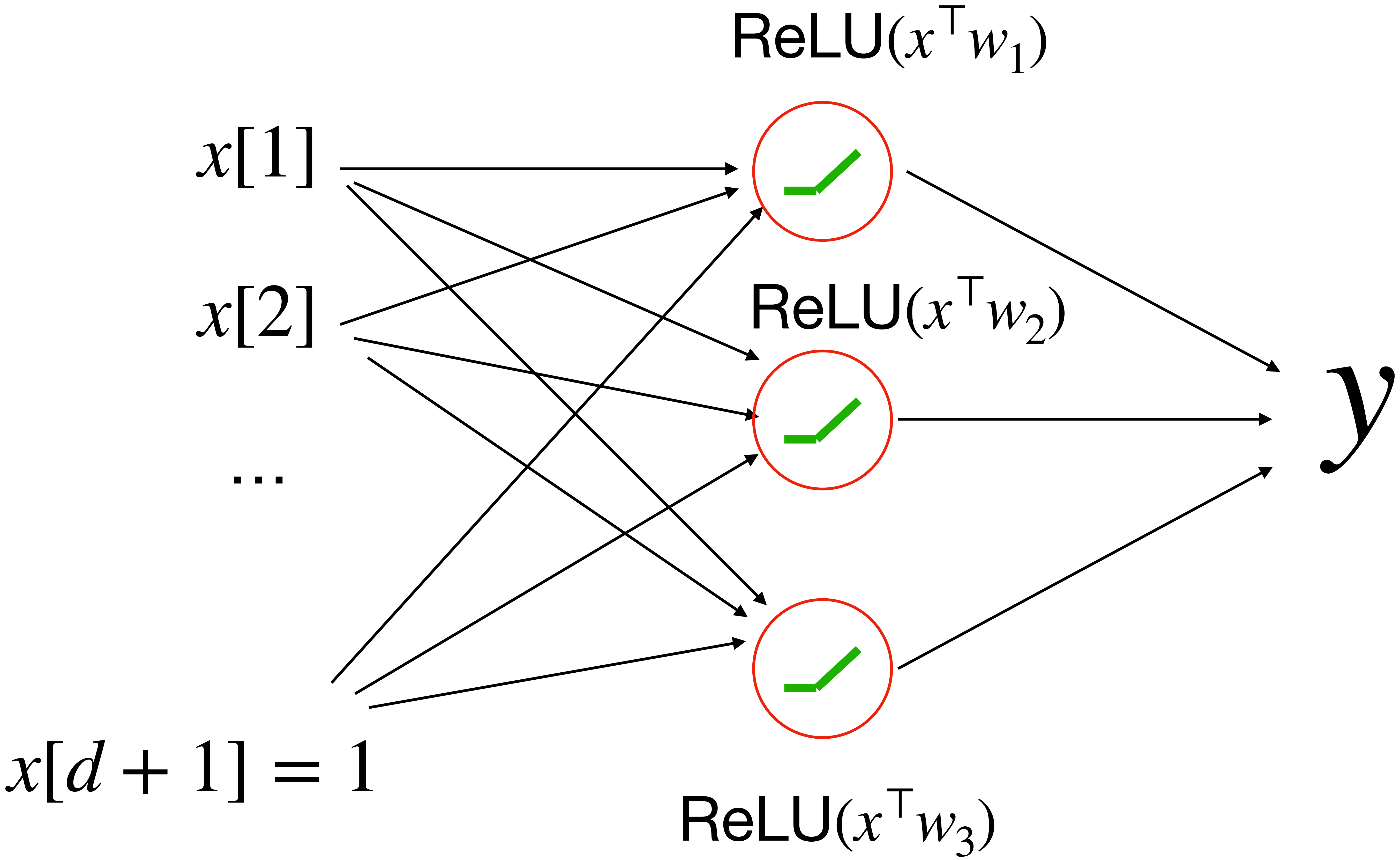


A single neuron network



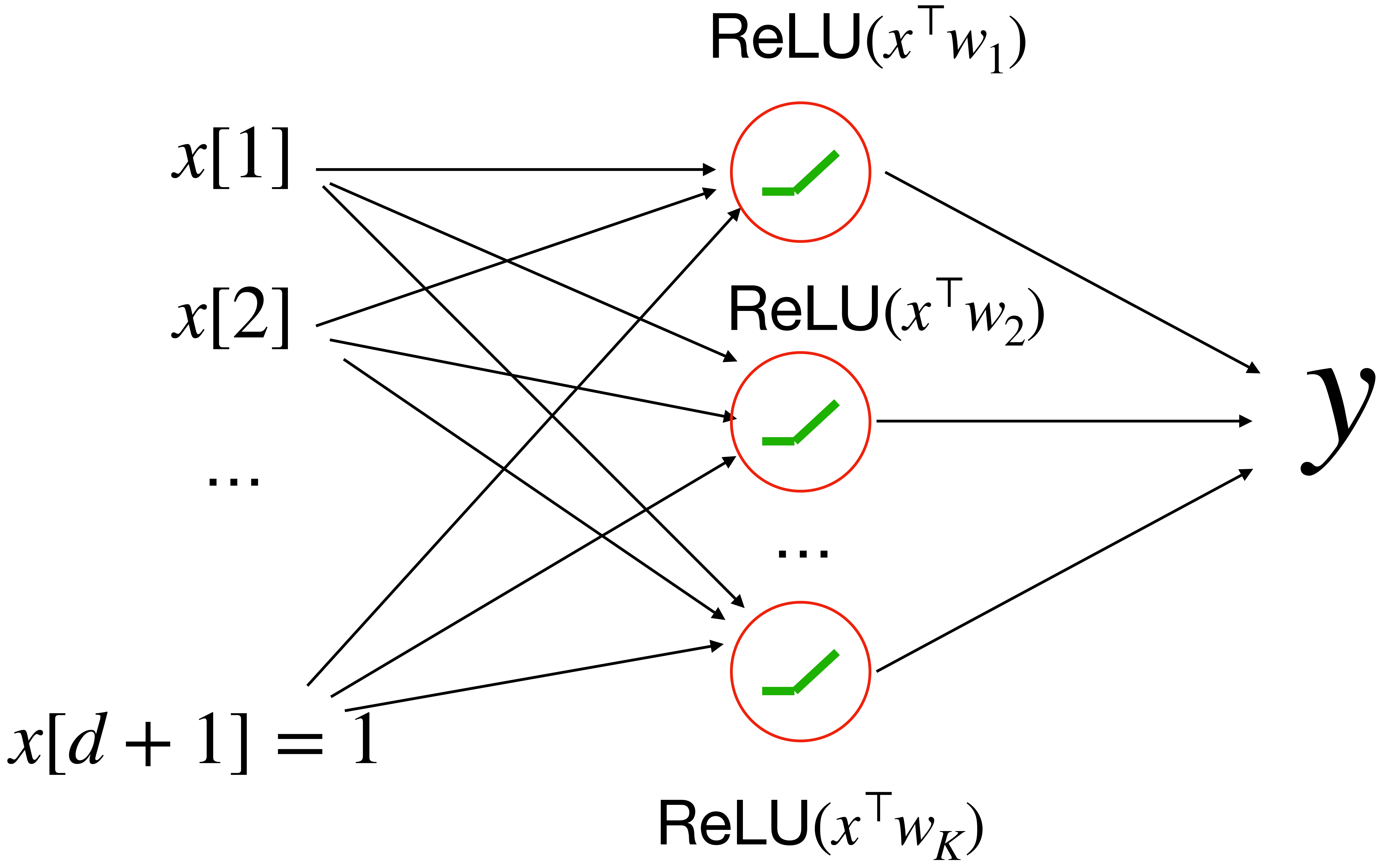
$$y = a\mathrm{ReLU}(w^\top x) + b$$

Let us stack multiple neurons together

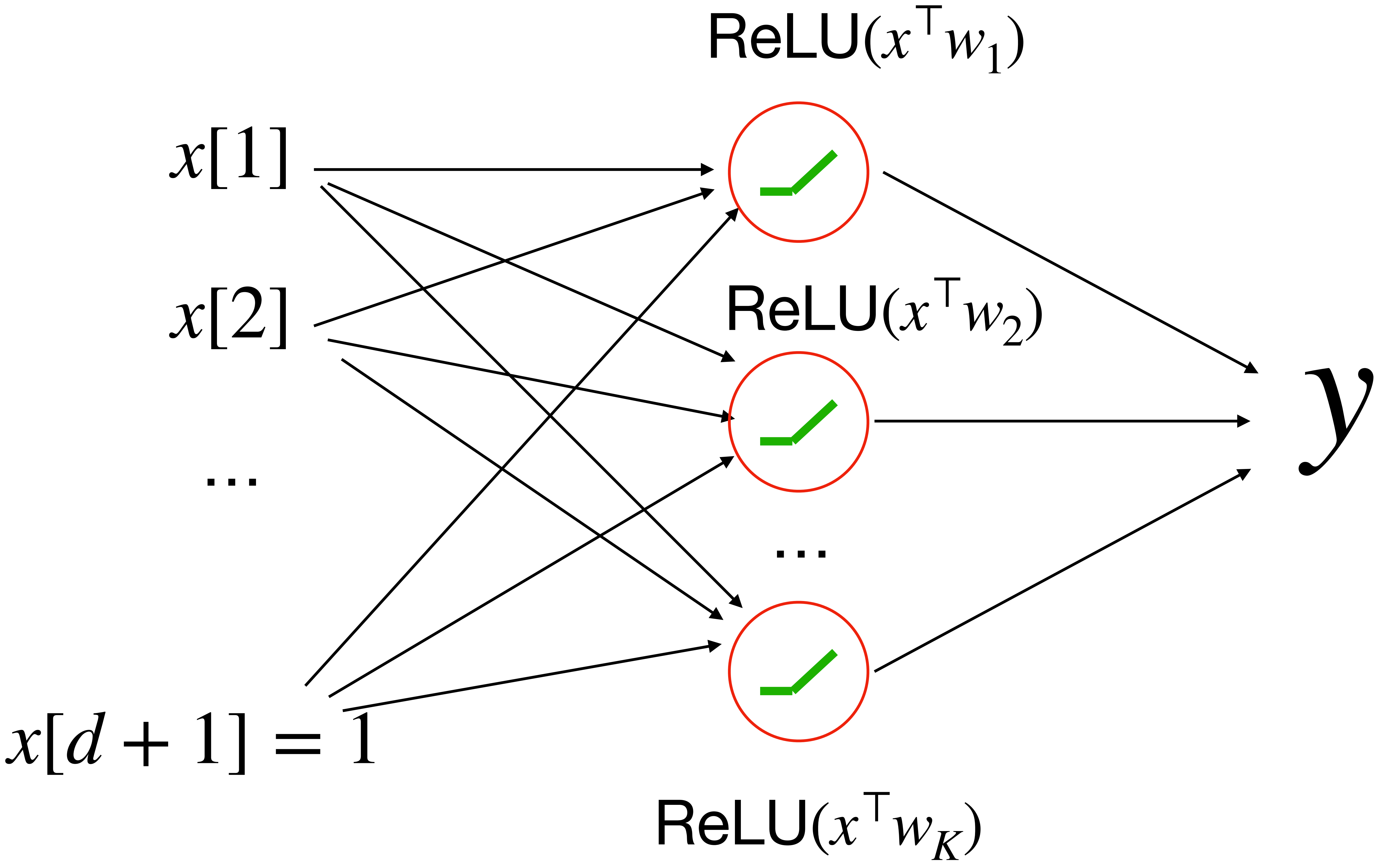


$$y = \sum_{i=1}^{3} a_i \mathrm{ReLU}(x^\top w_i) + b$$

Let us stack multiple neurons together



Let us stack multiple neurons together



$\text{ReLU}(x^\top w_1)$

$x[1]$

$x[2]$

$\text{ReLU}(x^\top w_2)$

$\cdots$

y

$\cdots$

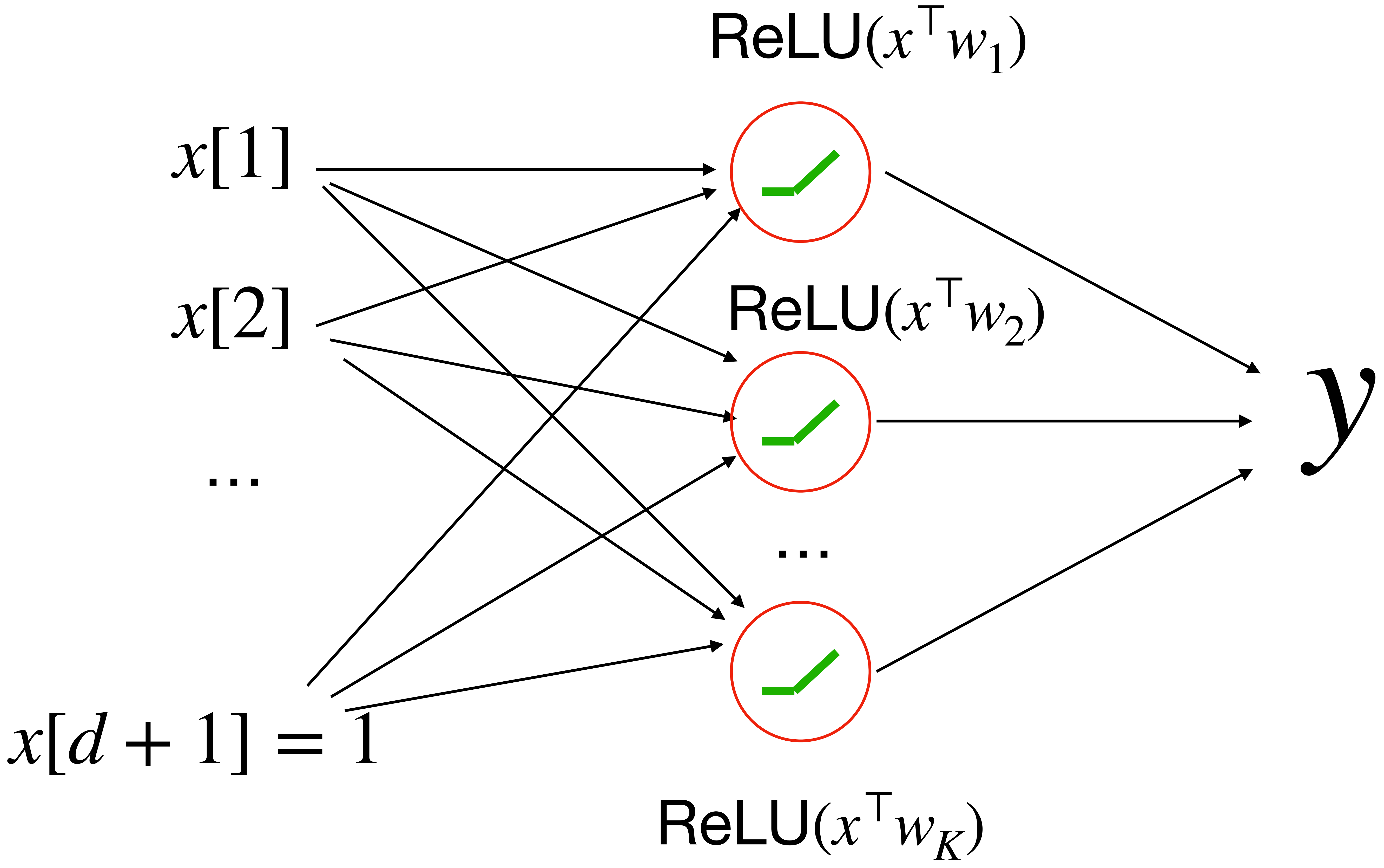
$x[d+1] = 1$

$\text{ReLU}(x^\top w_K)$

Vectorized form:

$$\text{Define } W = \begin{bmatrix} (w_1)^\top \\ \cdots \\ (w_K)^\top \end{bmatrix} \in \mathbb{R}^{K\times d}$$

Let us stack multiple neurons together

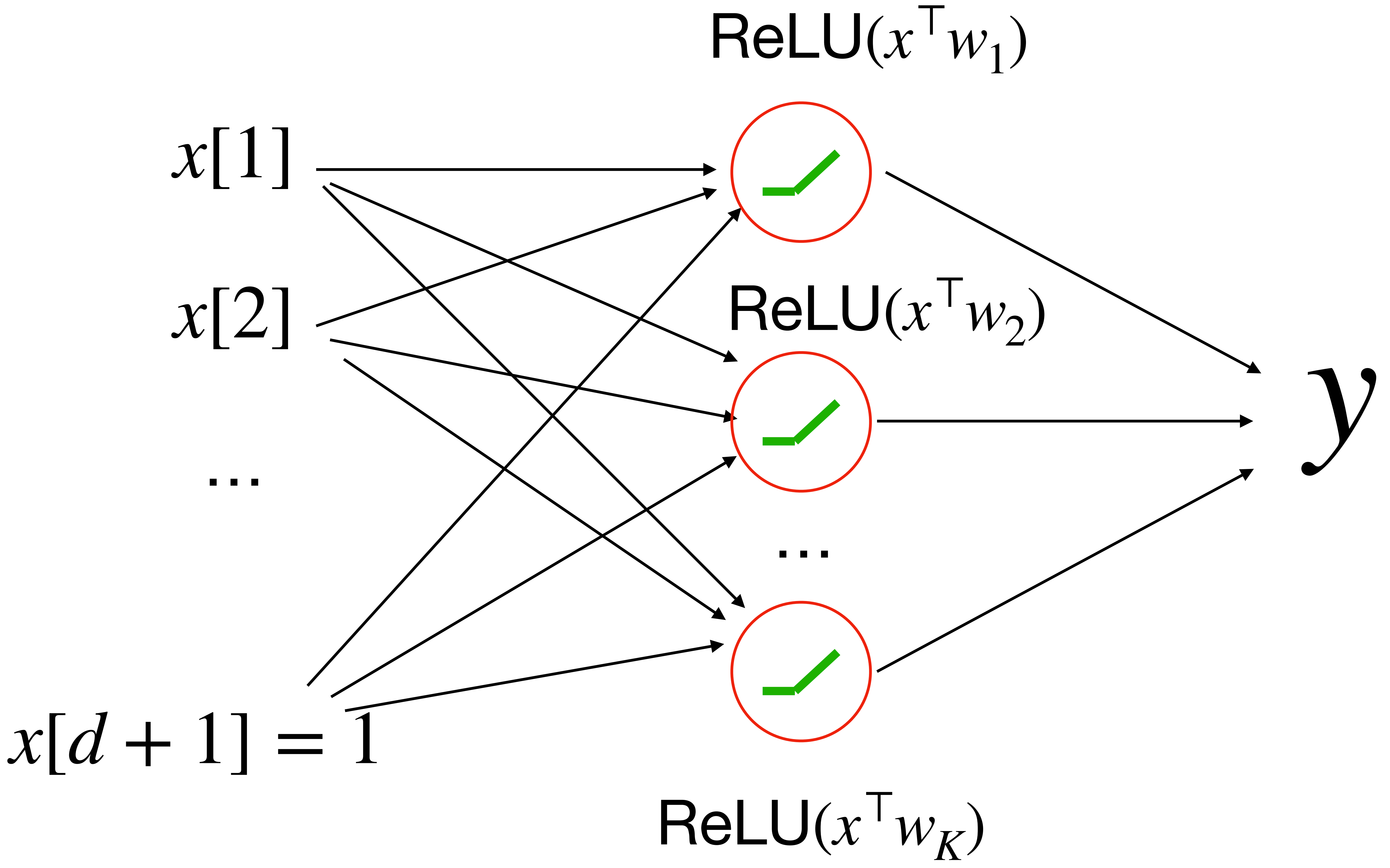


Vectorized form:

$$\text{Define } W = \begin{bmatrix} (w_1)^\top \\ \cdots \\ (w_K)^\top \end{bmatrix} \in \mathbb{R}^{K \times d}$$

$$\alpha = [a_1, \ldots, a_K]^\top$$

Let us stack multiple neurons together



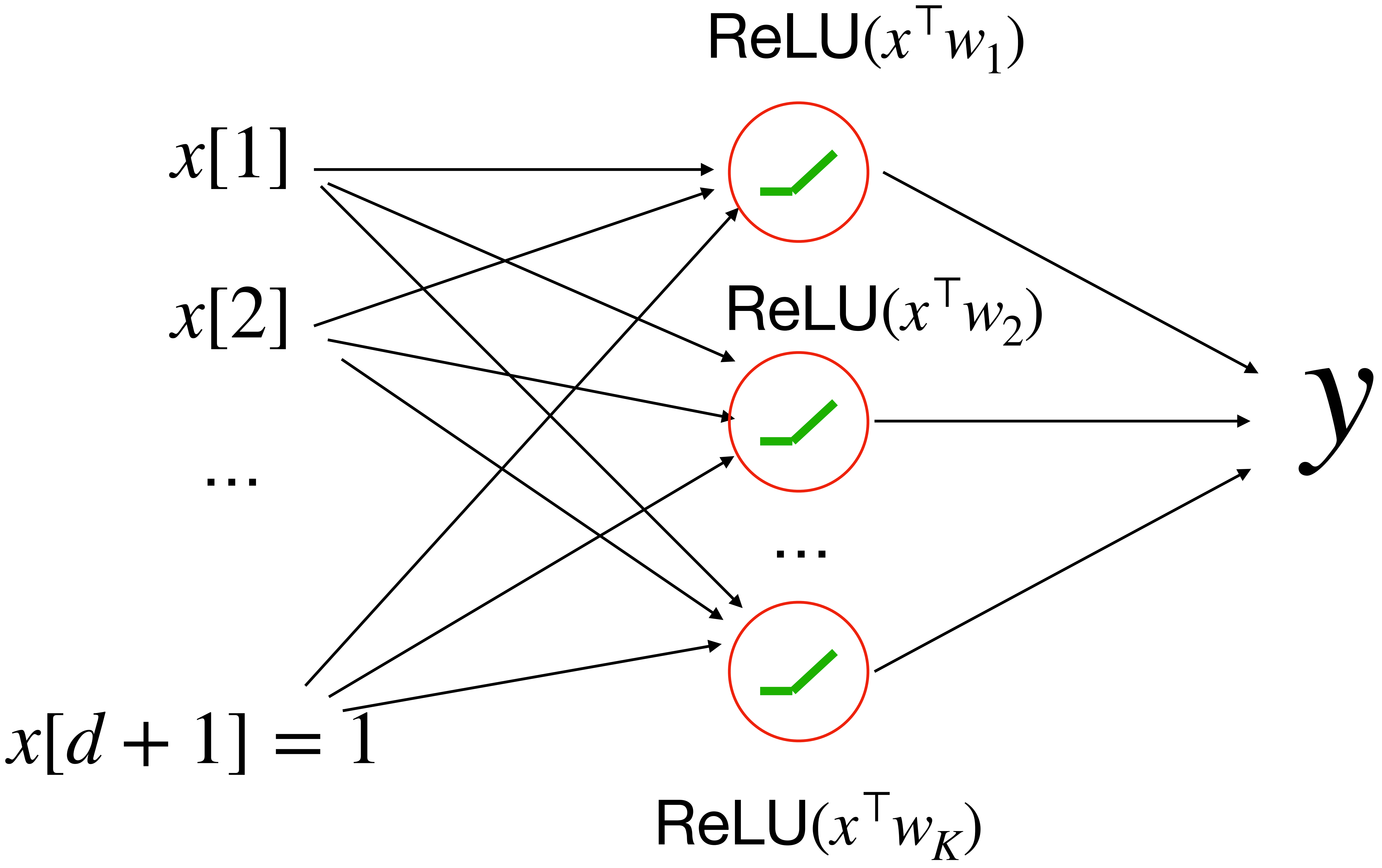
Vectorized form:

Define $W = \begin{bmatrix} (w_1)^\top \\ \cdots \\ (w_K)^\top \end{bmatrix} \in \mathbb{R}^{K \times d}$

$$\alpha = [a_1, \ldots, a_K]^\top$$

$$y = \alpha^\top \left(\text{ReLU}(Wx)\right) + b$$

Let us stack multiple neurons together



Vectorized form:

$$\text{Define } W = \begin{bmatrix} (w_1)^\top \\ \cdots \\ (w_K)^\top \end{bmatrix} \in \mathbb{R}^{K \times d}$$

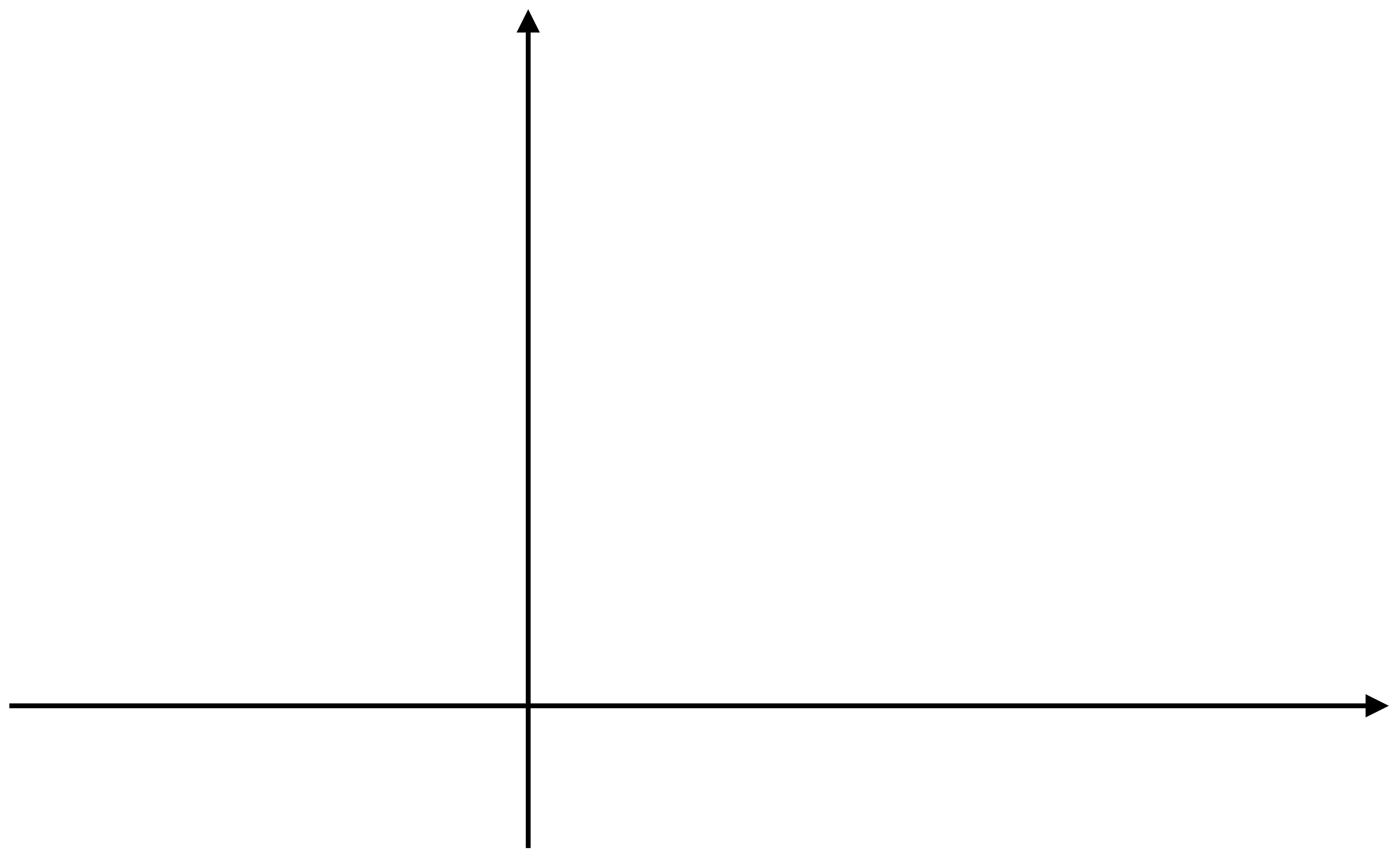
$$\alpha = [a_1, \ldots, a_K]^\top$$

$$y = \alpha^\top \boxed{\left(\text{ReLU}(Wx)\right)} + b$$

Learnable feature $\phi(x)$

What does a neural network approximate

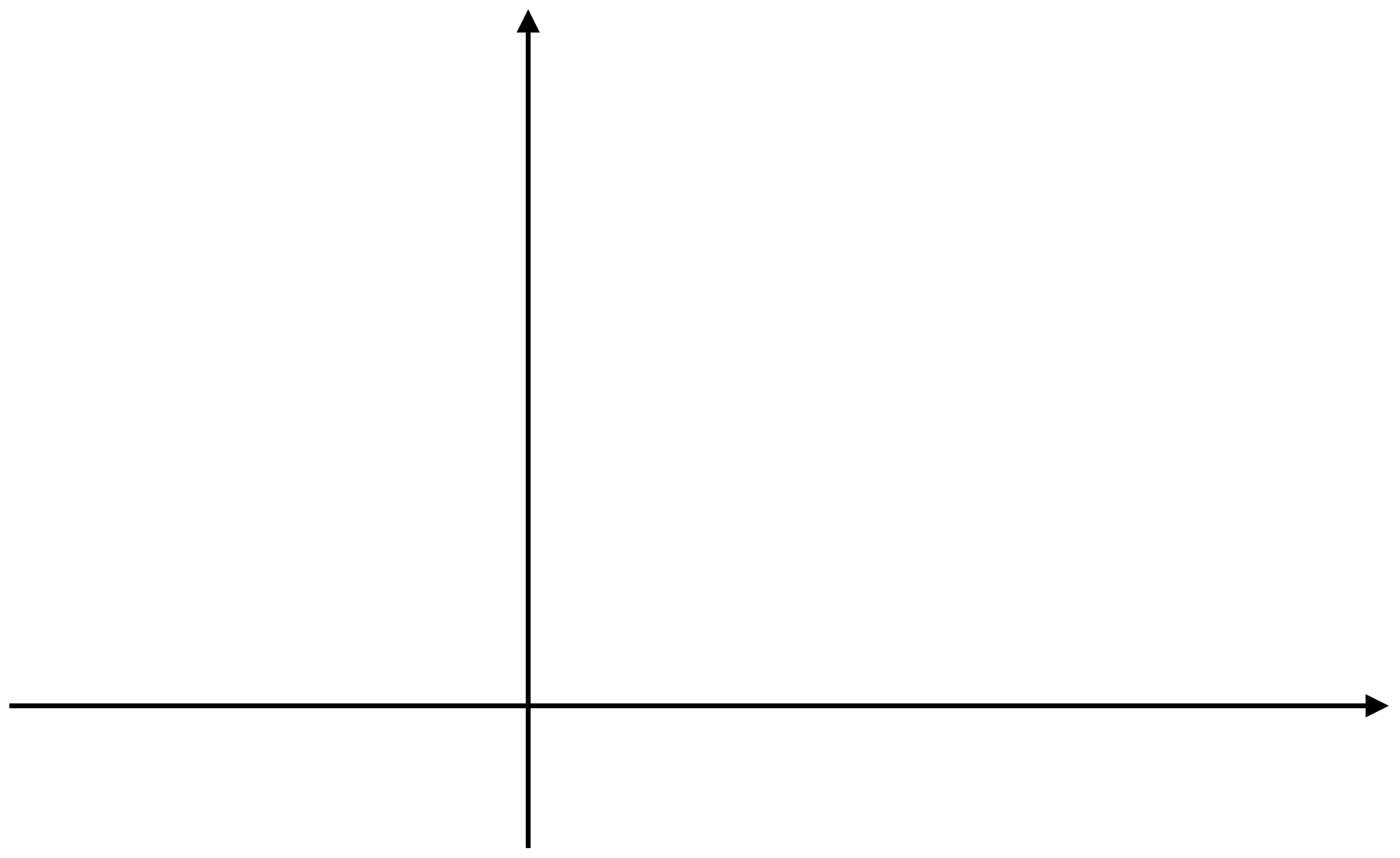
$$y = \alpha^\top \left(\mathrm{ReLU}(Wx)\right) + b$$



What does a neural network approximate

$$y = \alpha^{\top}\left(\text{ReLU}(Wx)\right) + b$$

It’s a pieces wise linear functions

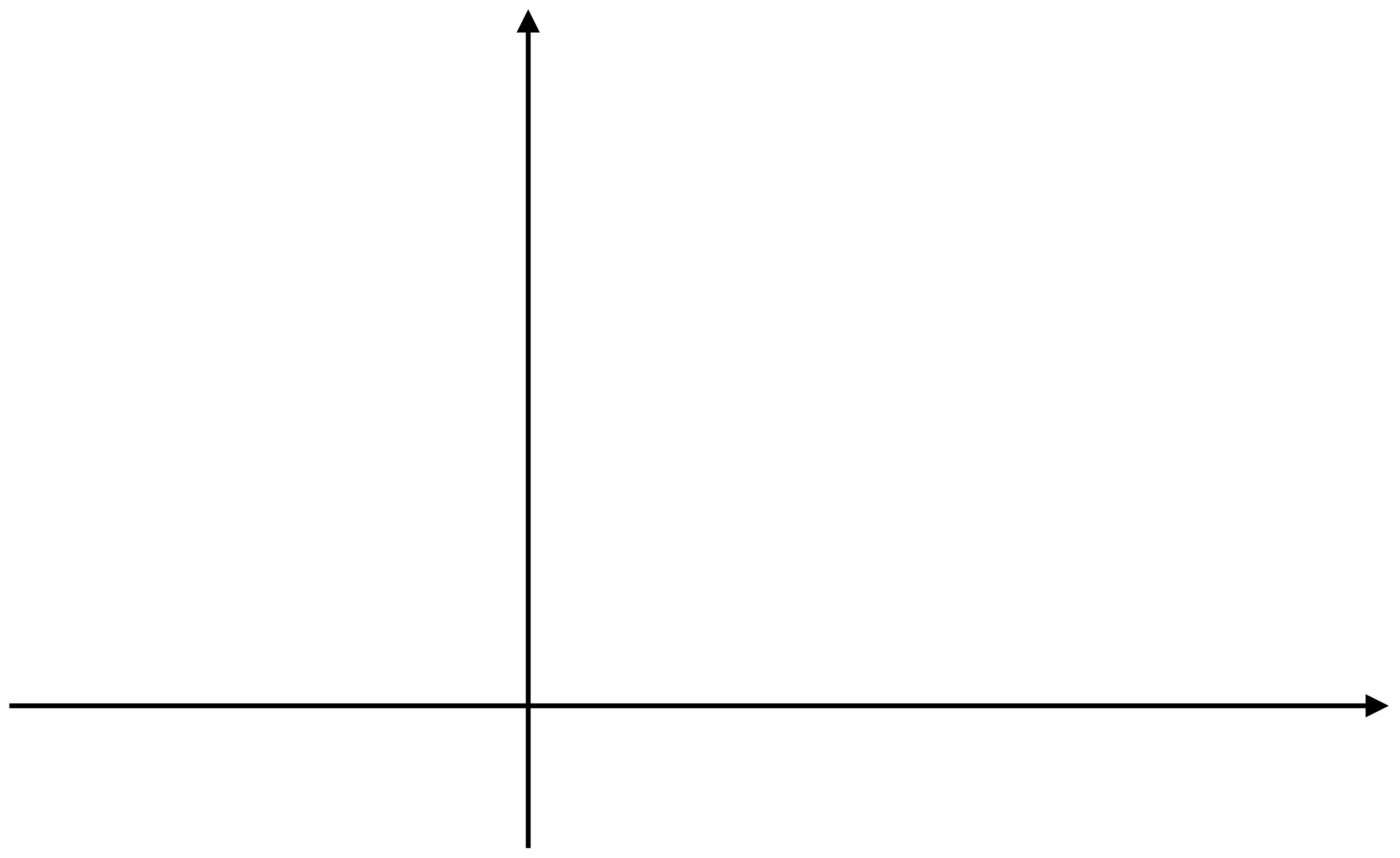


What does a neural network approximate

$$y = \alpha^\top \left(\text{ReLU}(Wx)\right) + b$$

It’s a pieces wise linear functions

Consider $d = 1$ case (and assume $b = 0$):



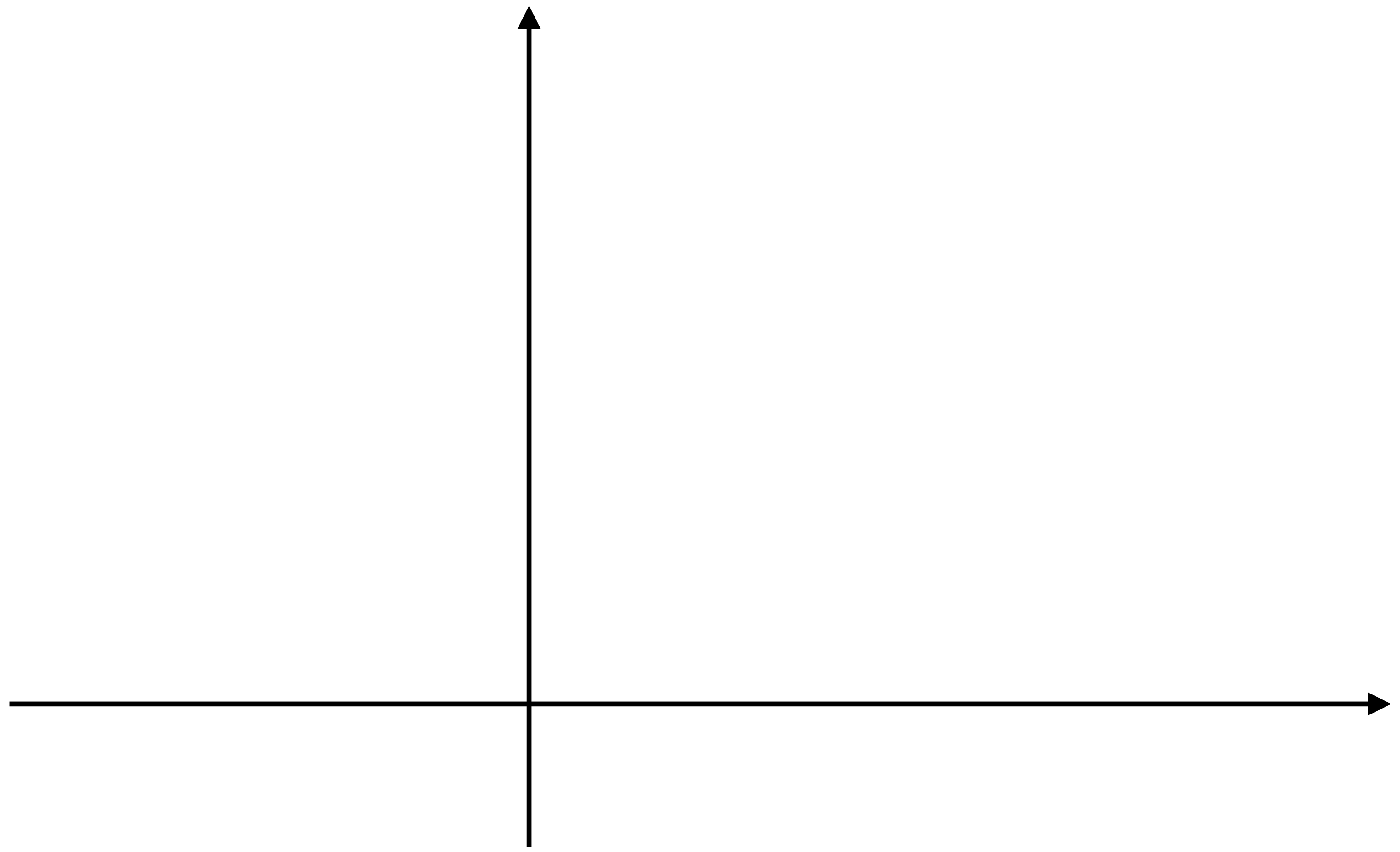
What does a neural network approximate

$$y = \alpha^\top \left(\text{ReLU}(Wx)\right) + b$$

It’s a pieces wise linear functions

Consider $d = 1$ case (and assume $b = 0$):

$K = 1 : y = a_1 \max\{w_1 x + c_1, 0\}$



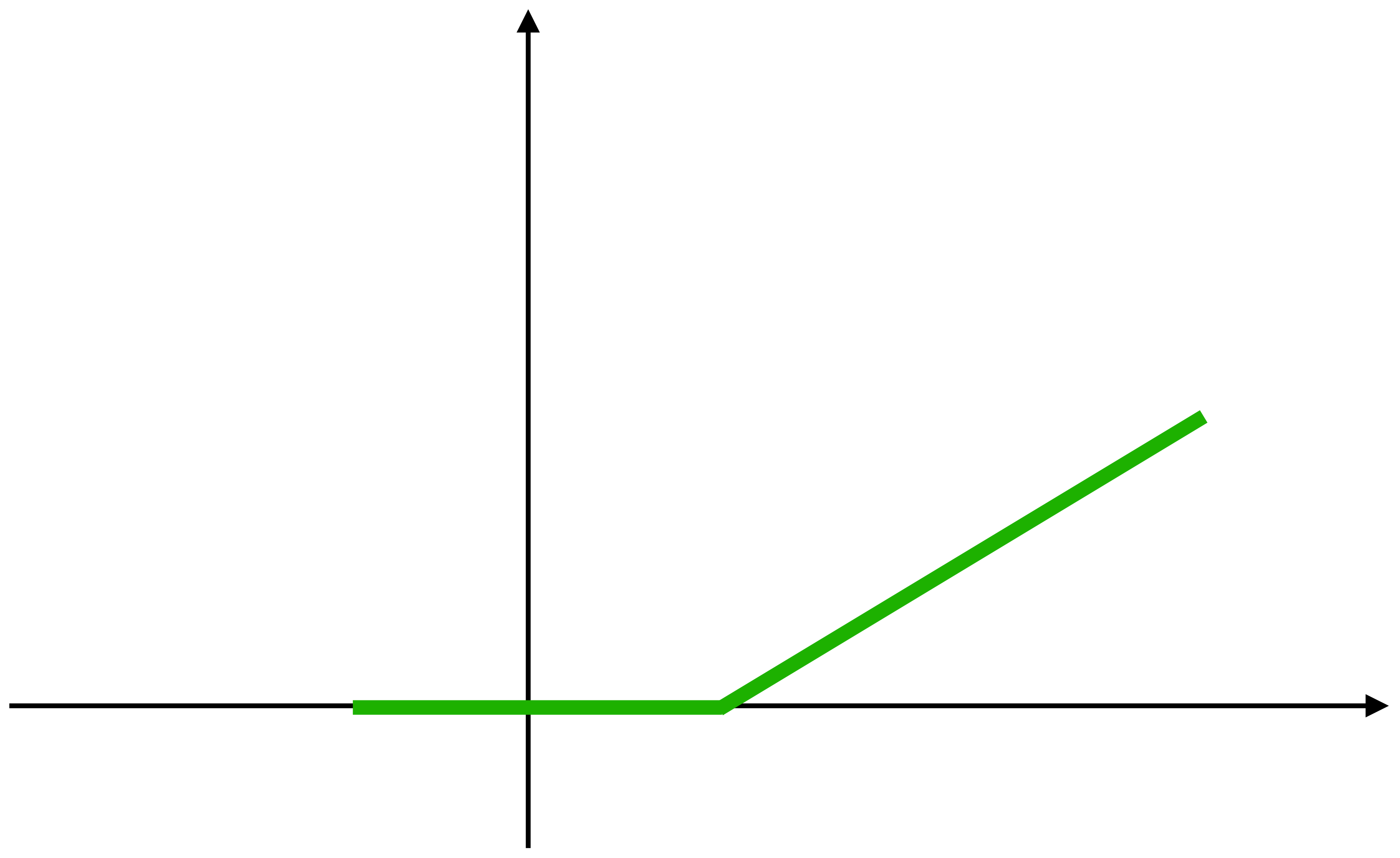
What does a neural network approximate

$$y = \alpha^\top \left(\text{ReLU}(Wx)\right) + b$$

It’s a pieces wise linear functions

Consider $d = 1$ case (and assume $b = 0$):

$K = 1 : y = a_1 \max\{w_1 x + c_1, 0\}$



What does a neural network approximate

$$y = \alpha^\top \left(\text{ReLU}(Wx)\right) + b$$

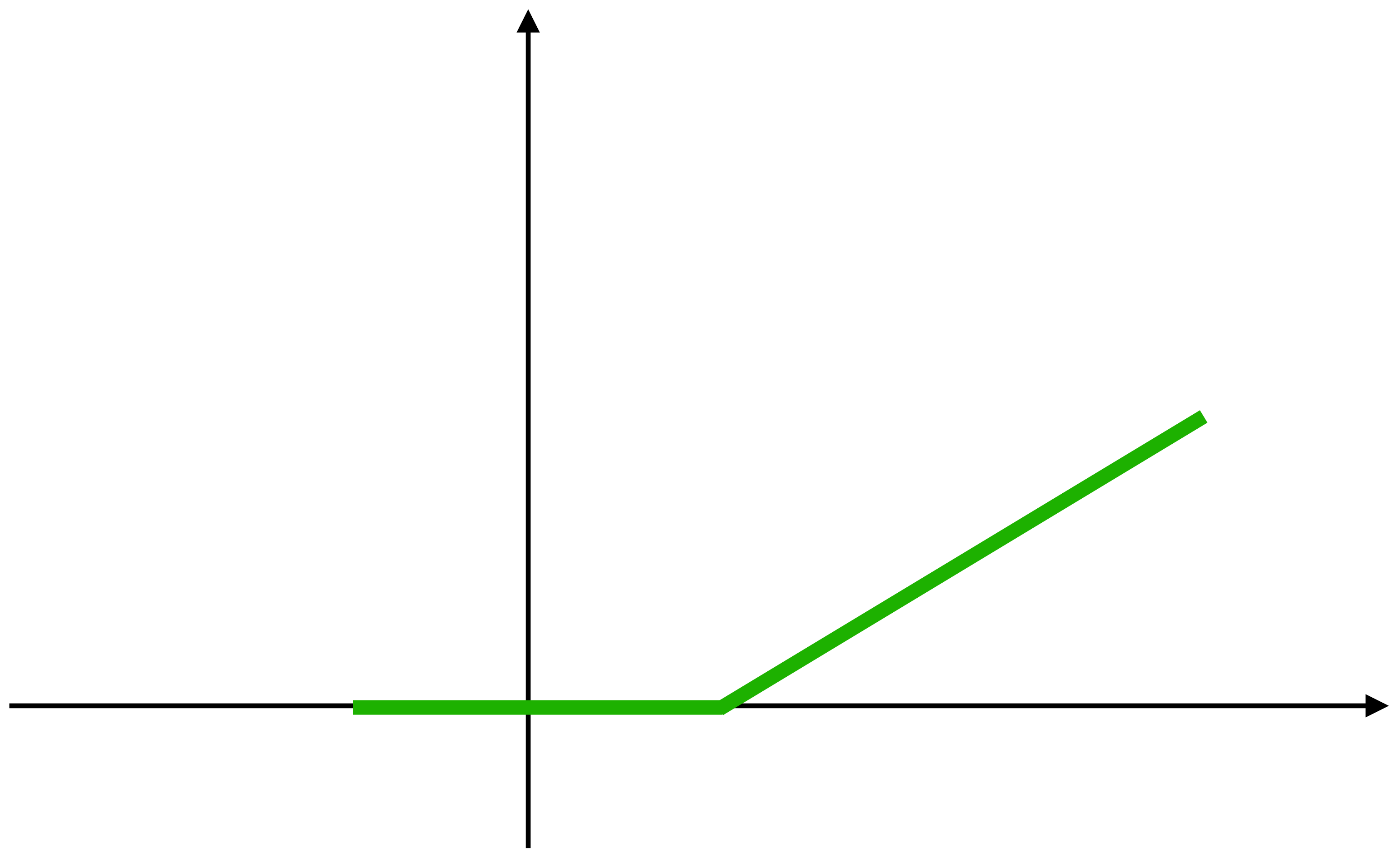
It’s a pieces wise linear functions

Consider $d = 1$ case (and assume $b = 0$):

$$K = 1 : y = a_1 \max\{w_1 x + c_1, 0\}$$

$$K = 2 : y = a_1 \max\{w_1 x + c_1, 0\}$$

$$+a_2 \max\{w_2 x + c_2, 0\}$$



What does a neural network approximate

$$y = \alpha^\top \left(\mathrm{ReLU}(Wx)\right) + b$$

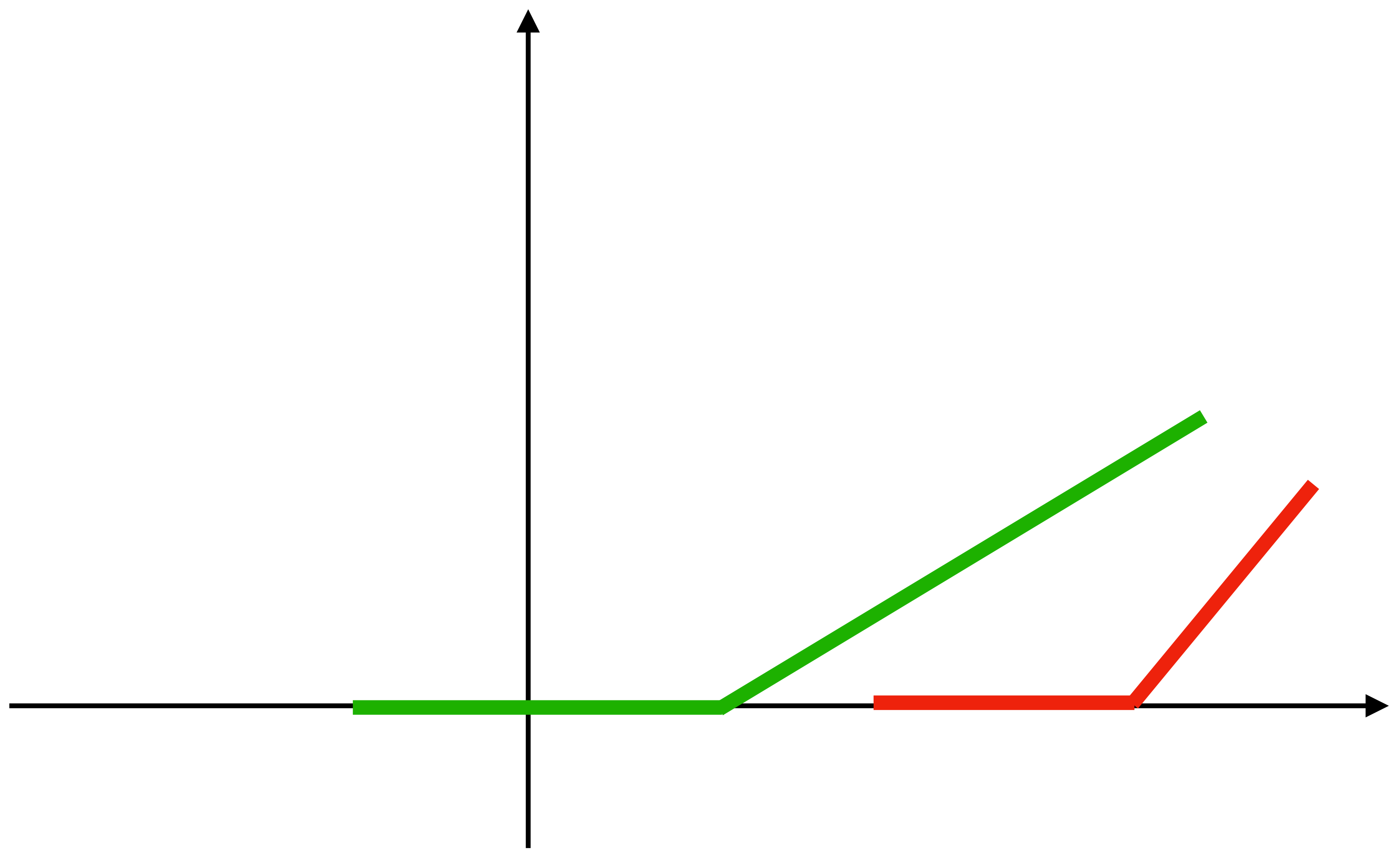
It's a pieces wise linear functions

Consider $d = 1$ case (and assume $b = 0$):

$$K = 1 : y = a_1 \max\{w_1 x + c_1, 0\}$$

$$K = 2 : y = a_1 \max\{w_1 x + c_1, 0\}$$

$$+ a_2 \max\{w_2 x + c_2, 0\}$$



What does a neural network approximate

$$y = \alpha^\top \left(\text{ReLU}(Wx)\right) + b$$

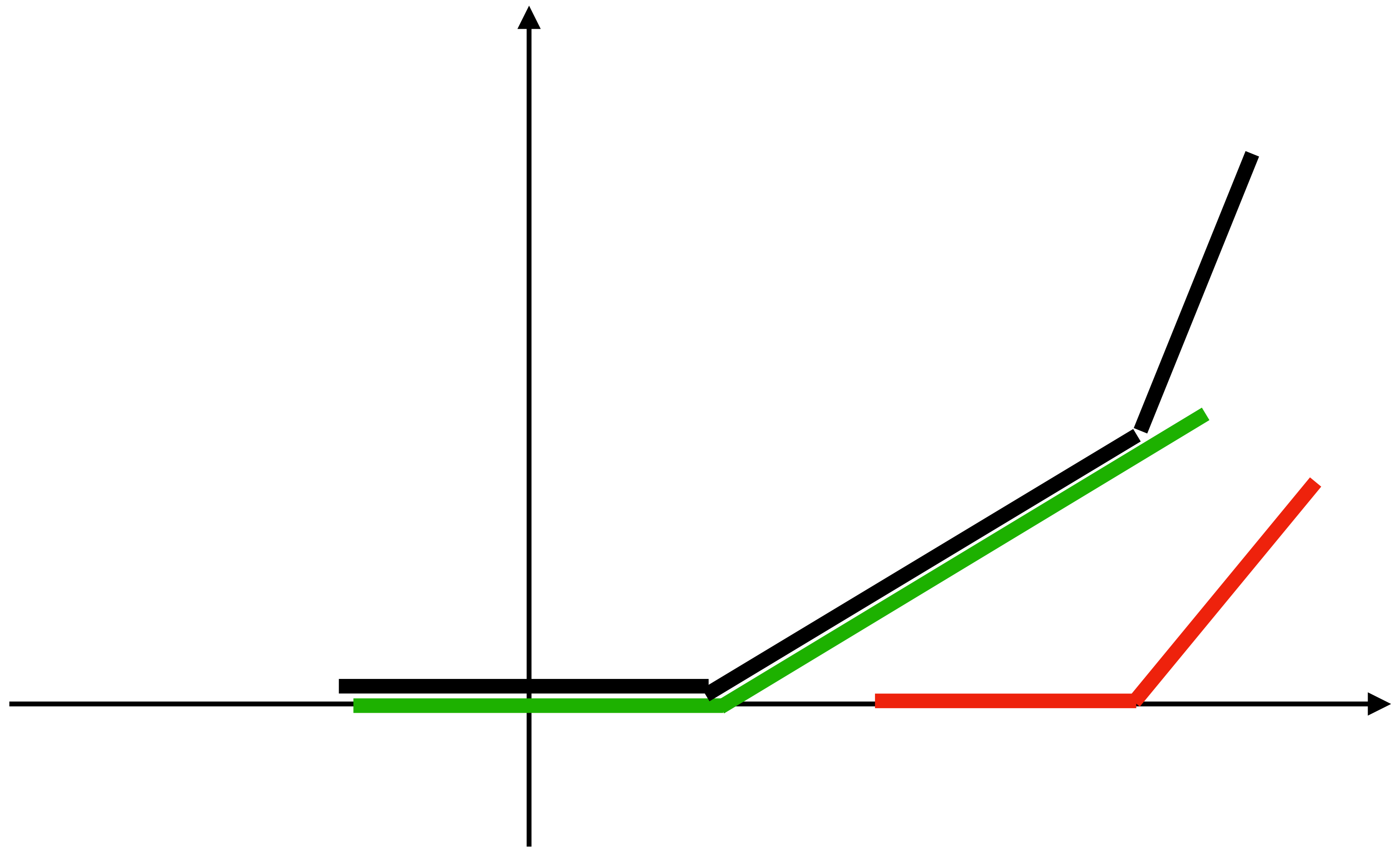
It’s a pieces wise linear functions

Consider $d = 1$ case (and assume $b = 0$):

$$K = 1 : y = a_1 \max\{w_1 x + c_1, 0\}$$

$$K = 2 : y = a_1 \max\{w_1 x + c_1, 0\}$$

$$+ a_2 \max\{w_2 x + c_2, 0\}$$



What does a neural network approximate

$$y = \alpha^\top \left(\text{ReLU}(Wx)\right) + b$$

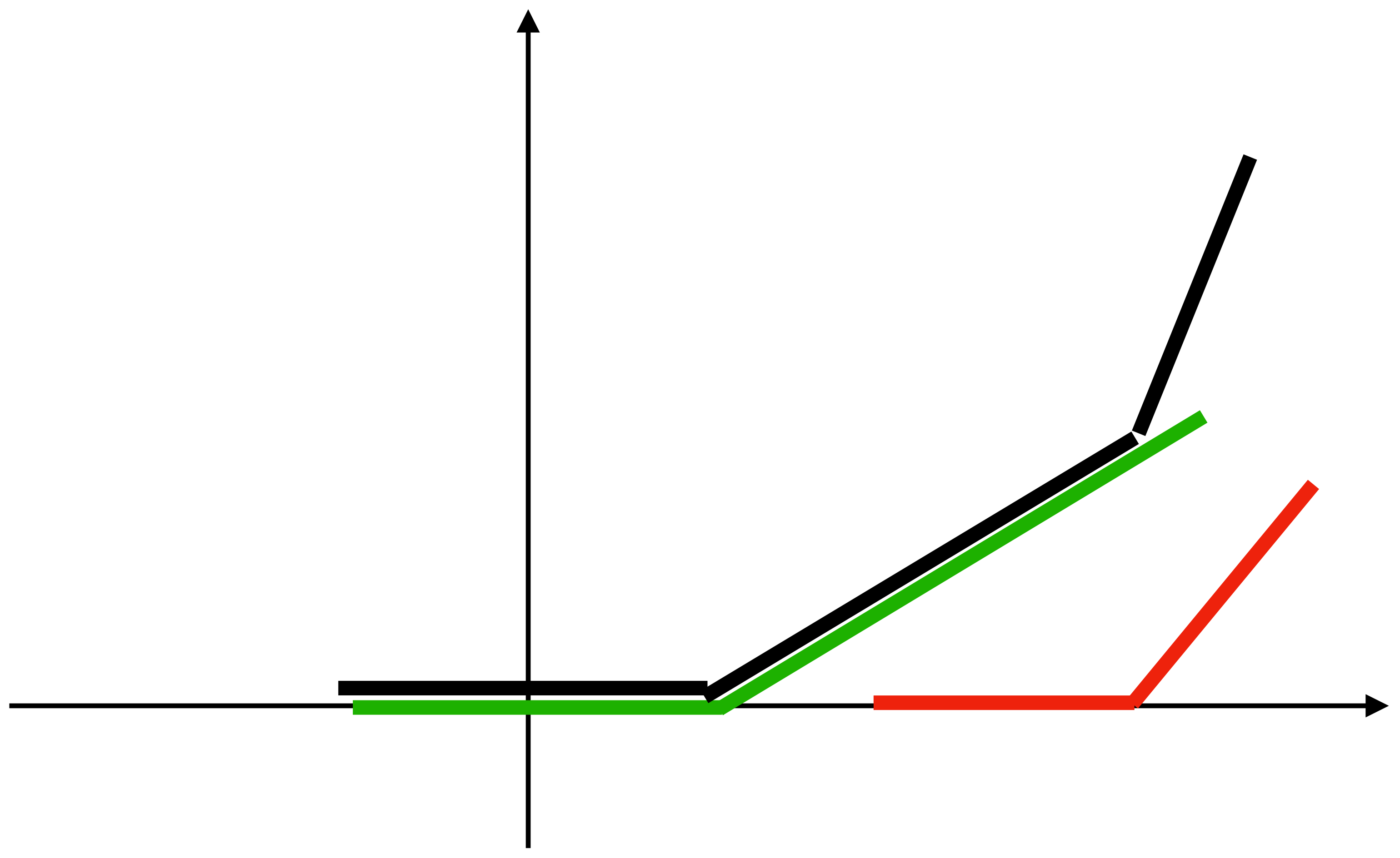
It's a pieces wise linear functions

Consider $d = 1$ case (and assume $b = 0$):

$$K = 1 : y = a_1 \max\{w_1 x + c_1, 0\}$$

$$K = 2 : y = a_1 \max\{w_1 x + c_1, 0\}$$
$$+ a_2 \max\{w_2 x + c_2, 0\}$$

$$K = 3 : y = a_1 \max\{w_1 x + c_1, 0\}$$
$$+ a_2 \max\{w_2 x + c_2, 0\}$$
$$+ a_3 \max\{w_3 x + c_3, 0\}$$



What does a neural network approximate

$$y = \alpha^\top \left(\text{ReLU}(Wx)\right) + b$$

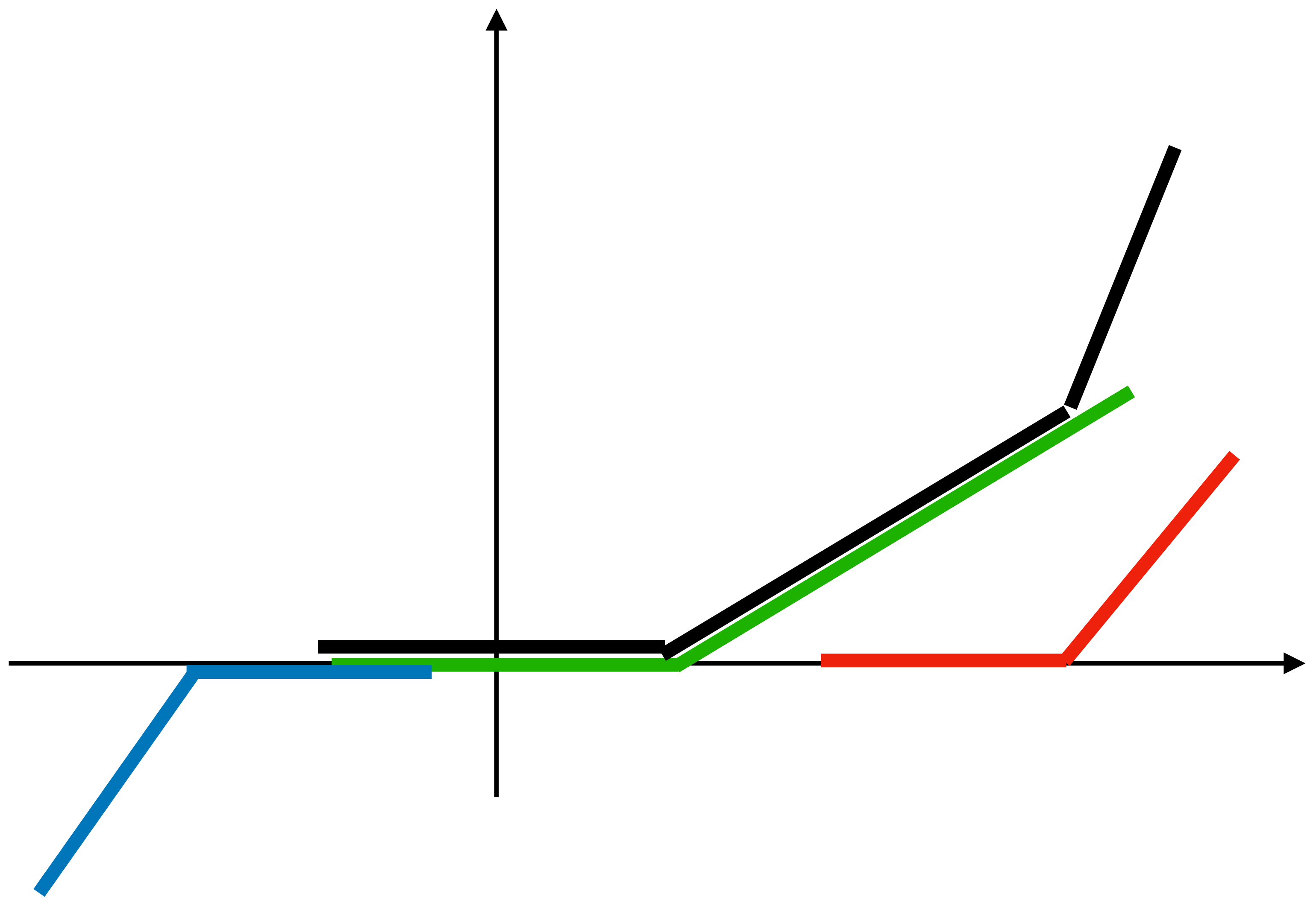
It’s a pieces wise linear functions

Consider $d = 1$ case (and assume $b = 0$):

$$K = 1 : y = a_1 \max\{w_1 x + c_1, 0\}$$

$$K = 2 : y = a_1 \max\{w_1 x + c_1, 0\}$$
$$+a_2 \max\{w_2 x + c_2, 0\}$$

$$K = 3 : y = a_1 \max\{w_1 x + c_1, 0\}$$
$$+a_2 \max\{w_2 x + c_2, 0\}$$
$$+a_3 \max\{w_3 x + c_3, 0\}$$



What does a neural network approximate

$$y = \alpha^\top \left(\text{ReLU}(Wx)\right) + b$$

It's a pieces wise linear functions

Consider $d = 1$ case (and assume $b = 0$):

$K = 1 : y = a_1 \max\{w_1 x + c_1, 0\}$

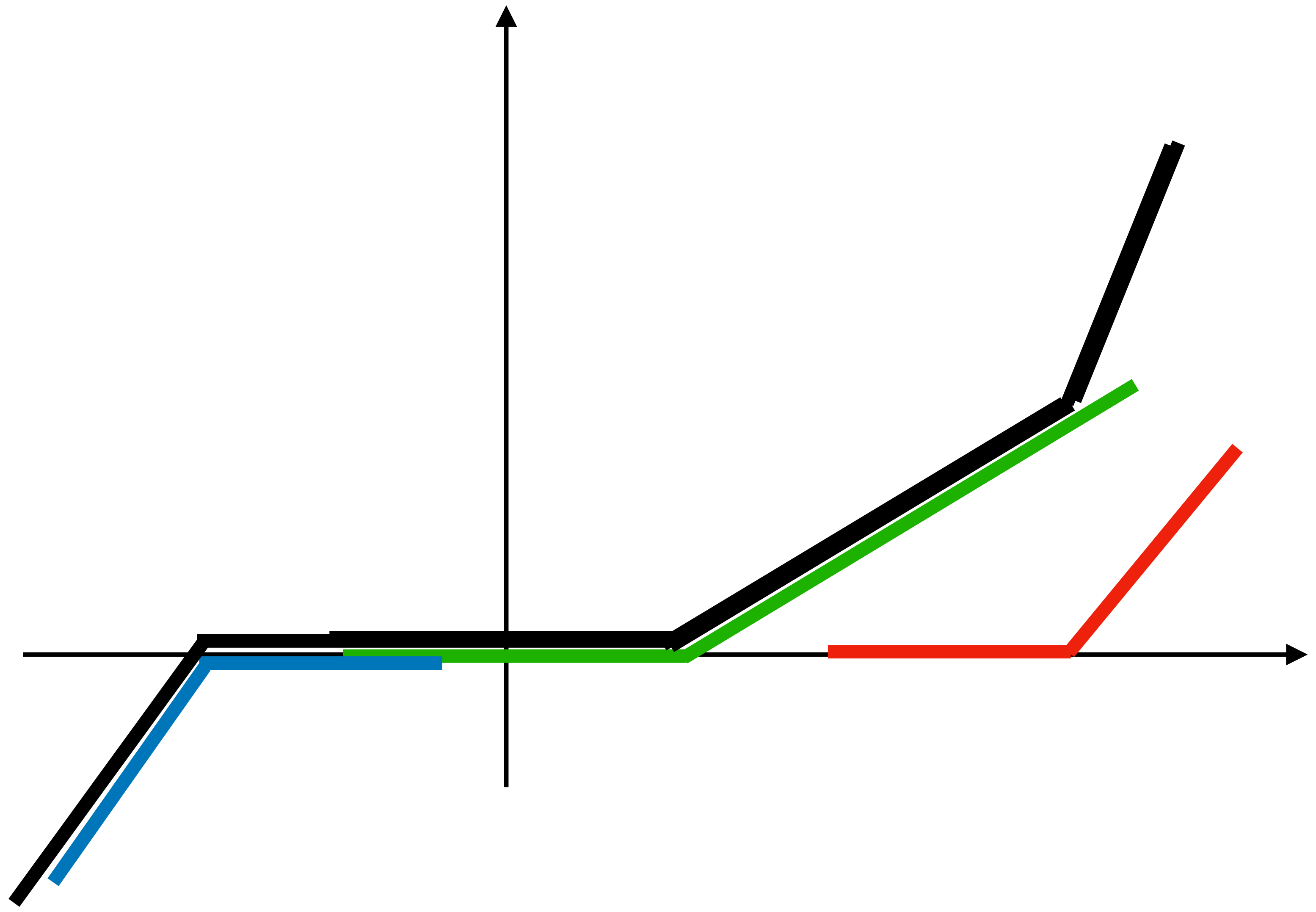
$K = 2 : y = a_1 \max\{w_1 x + c_1, 0\}$

$+a_2 \max\{w_2 x + c_2, 0\}$

$K = 3 : y = a_1 \max\{w_1 x + c_1, 0\}$

$+a_2 \max\{w_2 x + c_2, 0\}$

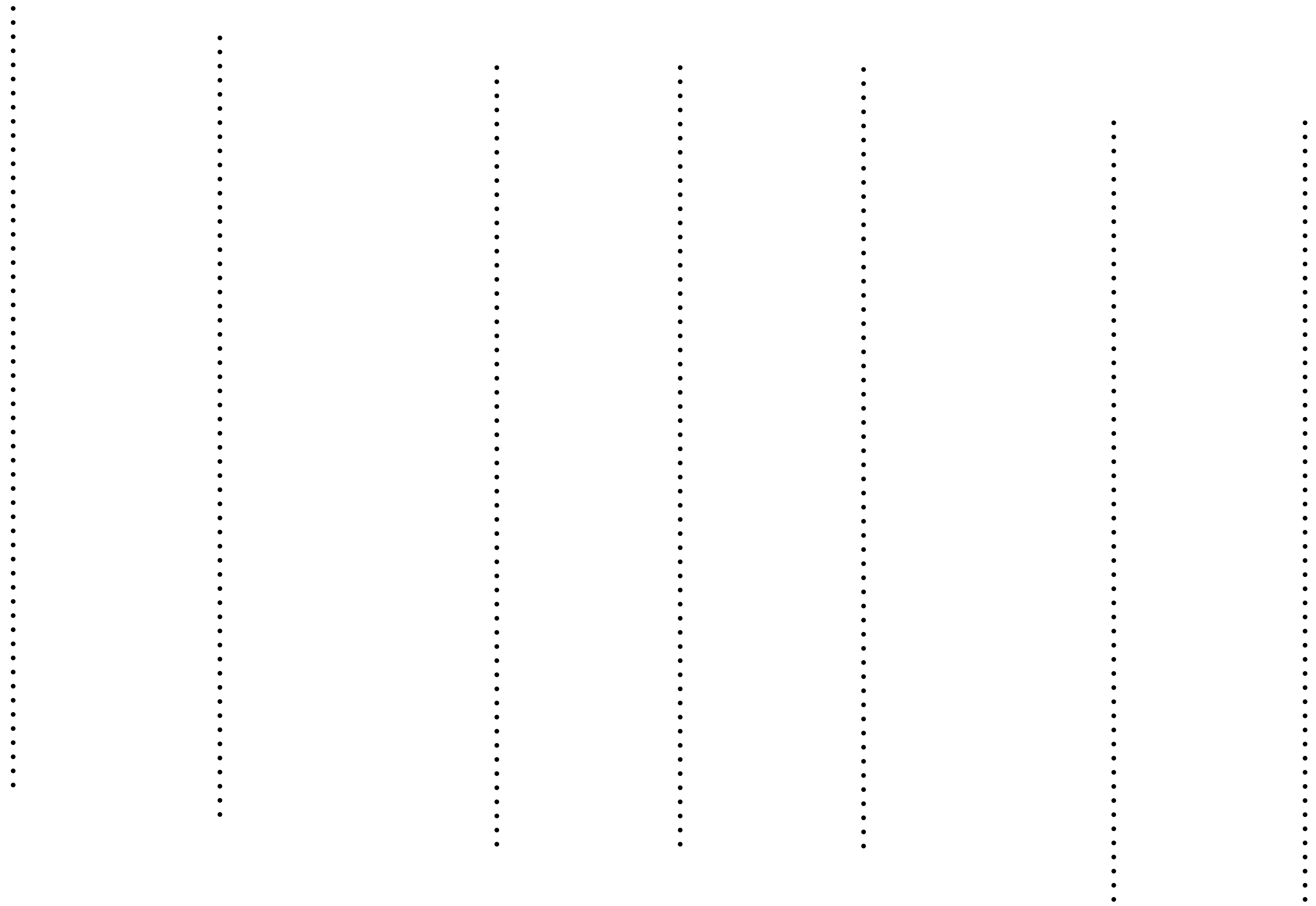
$+a_3 \max\{w_3 x + c_3, 0\}$



What does a neural network approximate

$$y = \alpha^\top \left(\text{ReLU}(Wx)\right) + b$$

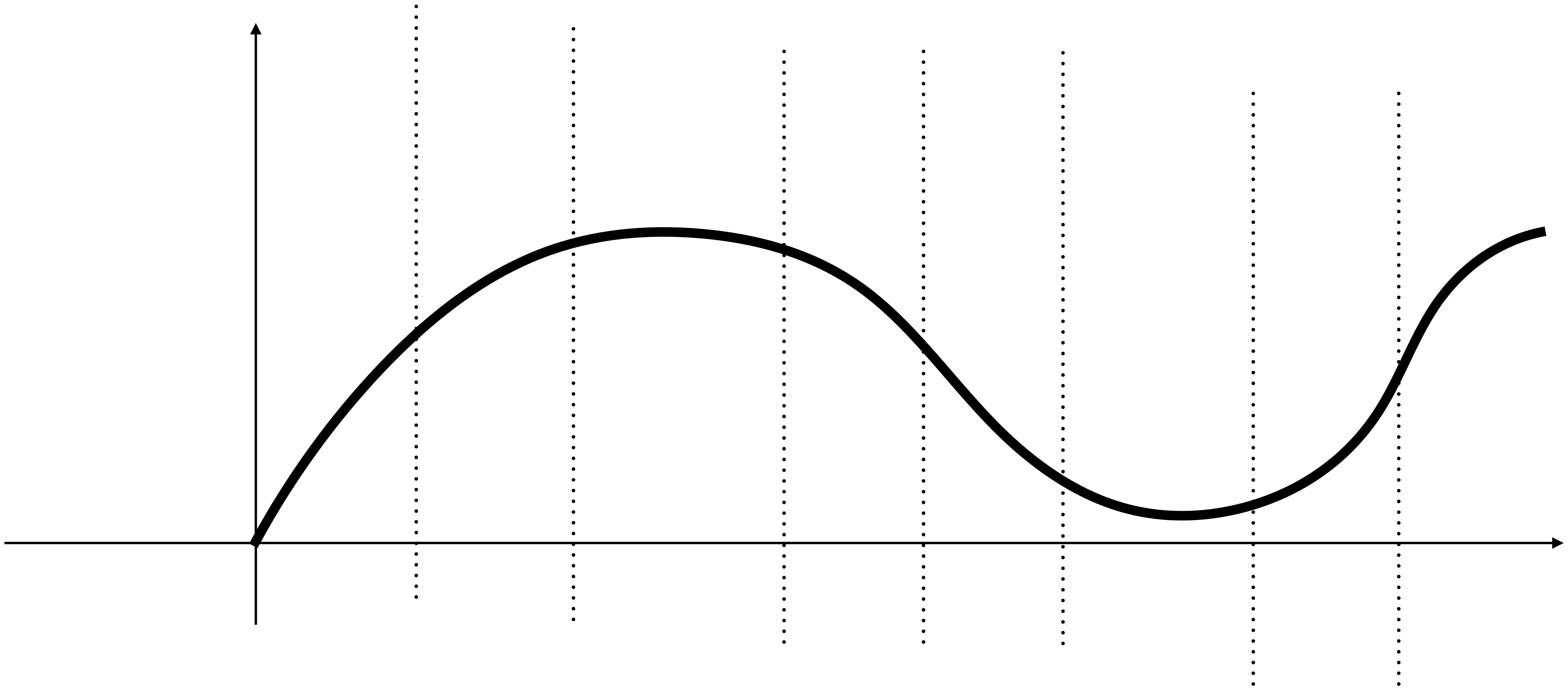
Claim: a wide enough one layer NN can approximate any smooth functions



What does a neural network approximate

$$y = \alpha^\top \left(\mathrm{ReLU}(Wx)\right) + b$$

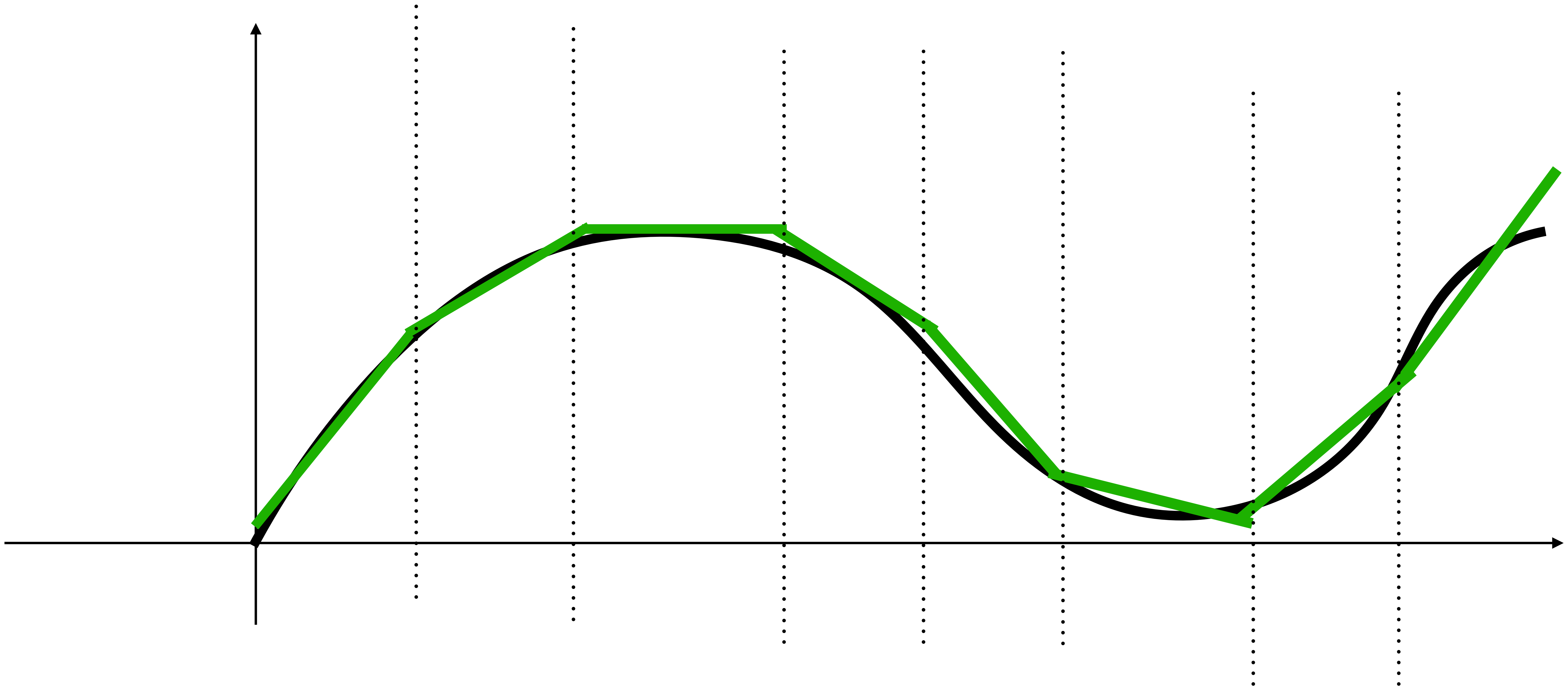
Claim: a wide enough one layer NN can approximate any smooth functions



What does a neural network approximate

$$y = \alpha^{\top}\left(\text{ReLU}(Wx)\right) + b$$

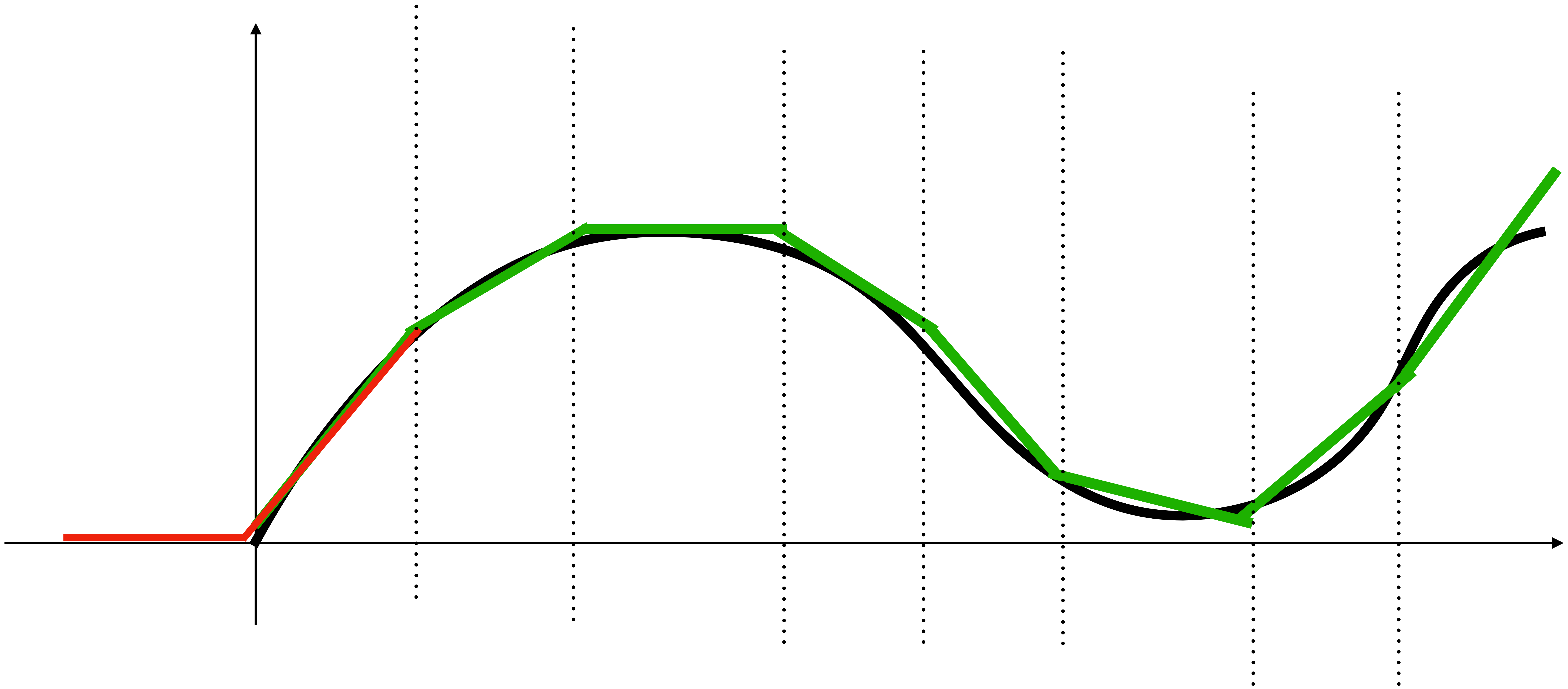
Claim: a wide enough one layer NN can approximate any smooth functions



What does a neural network approximate

$$y = \alpha^\top \left(\text{ReLU}(Wx)\right) + b$$

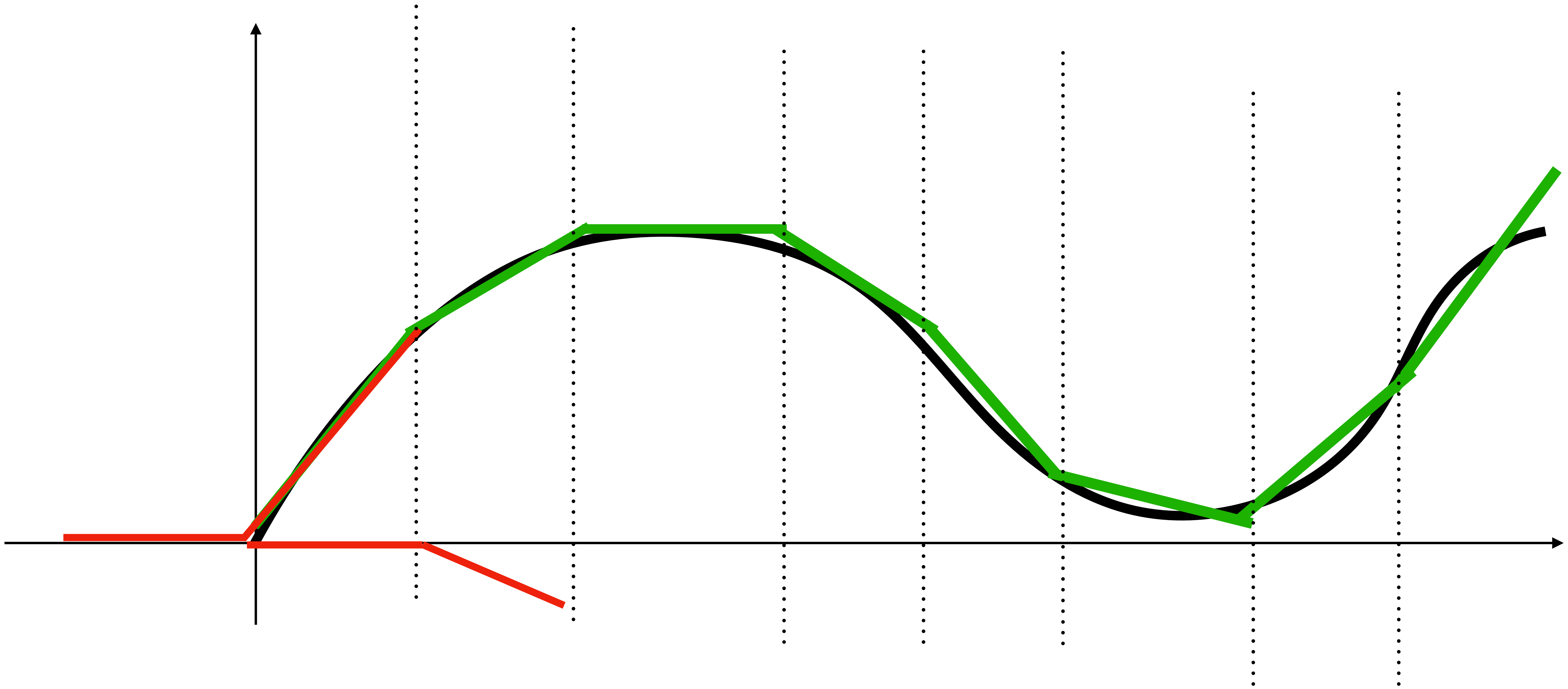
Claim: a wide enough one layer NN can approximate any smooth functions



What does a neural network approximate

$$y = \alpha^{\top}\left(\mathrm{ReLU}(Wx)\right) + b$$

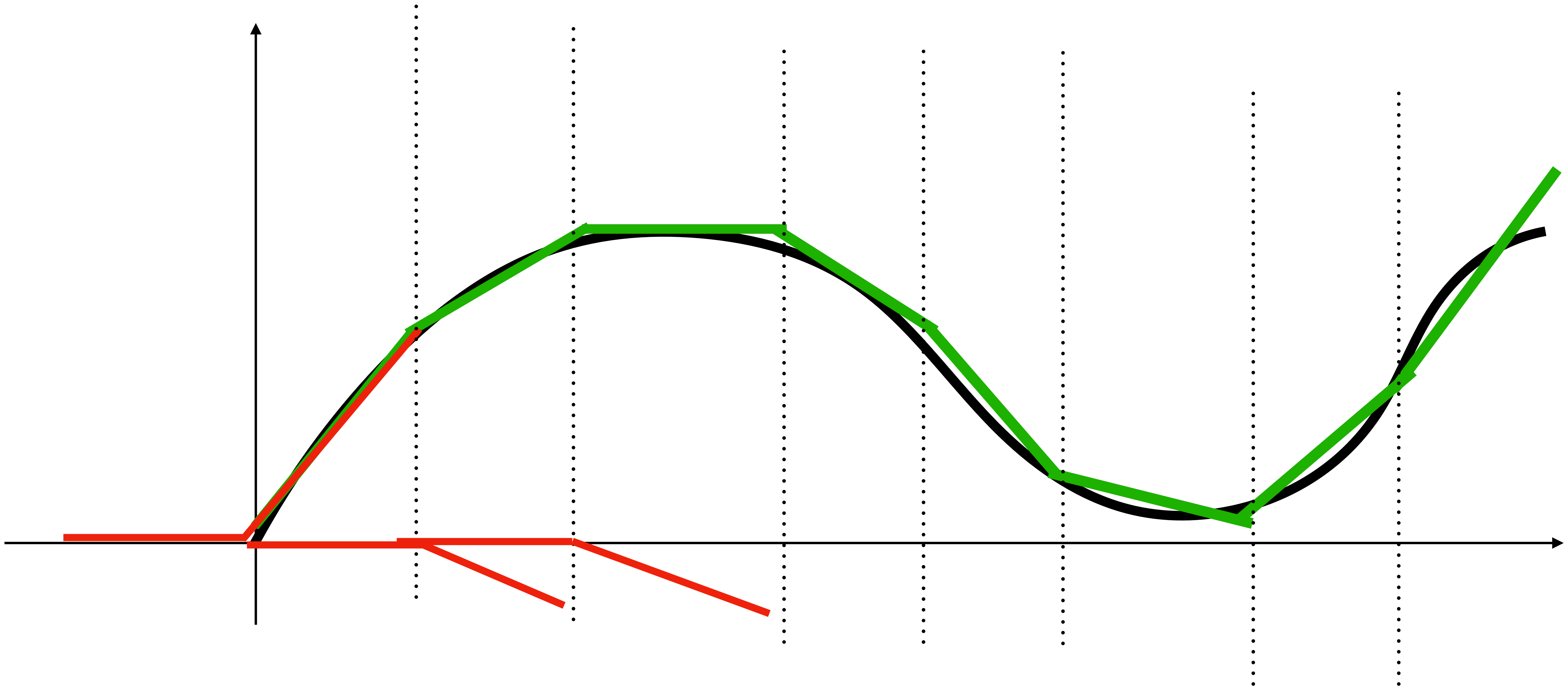
Claim: a wide enough one layer NN can approximate any smooth functions



What does a neural network approximate

$$y = \alpha^\top \left(\mathrm{ReLU}(Wx)\right) + b$$

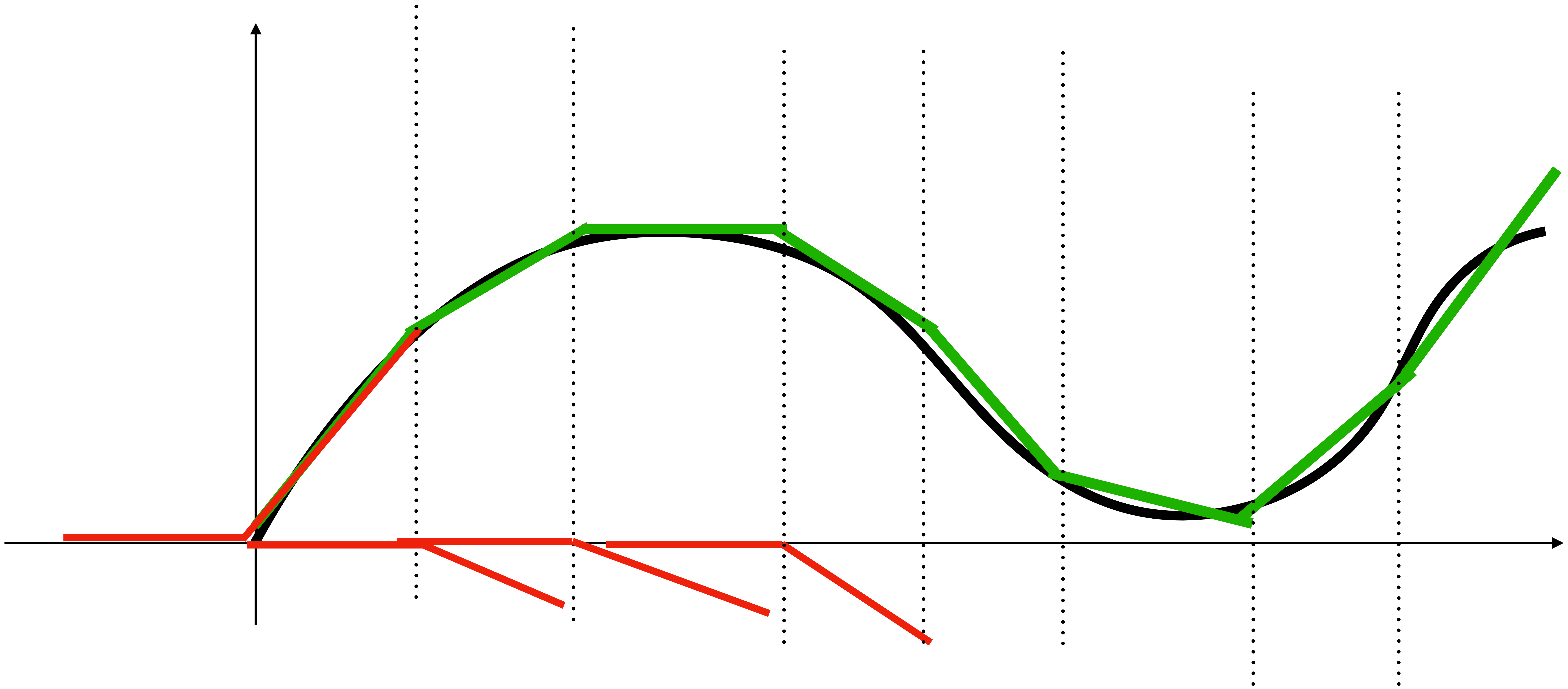
Claim: a wide enough one layer NN can approximate any smooth functions



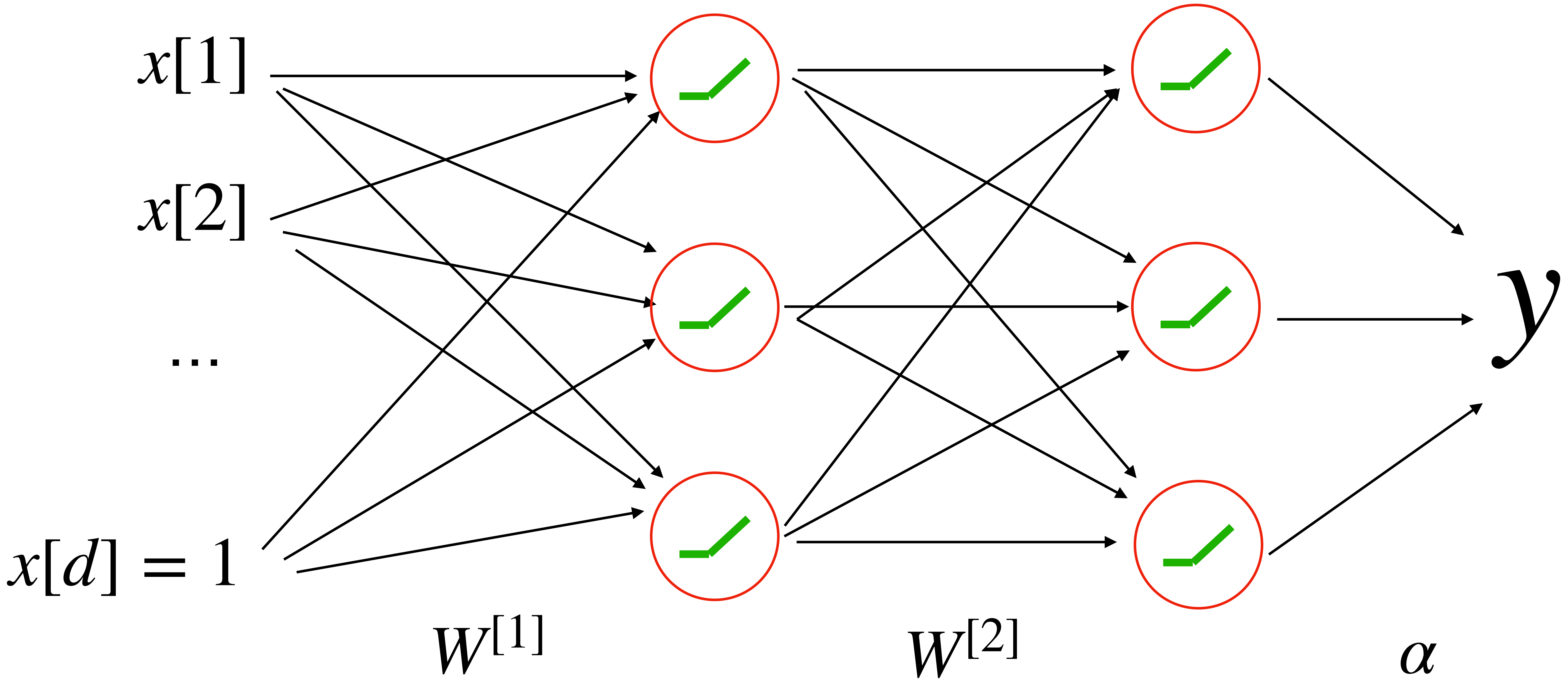
What does a neural network approximate

$$y = \alpha^\top \left(\mathrm{ReLU}(Wx)\right) + b$$

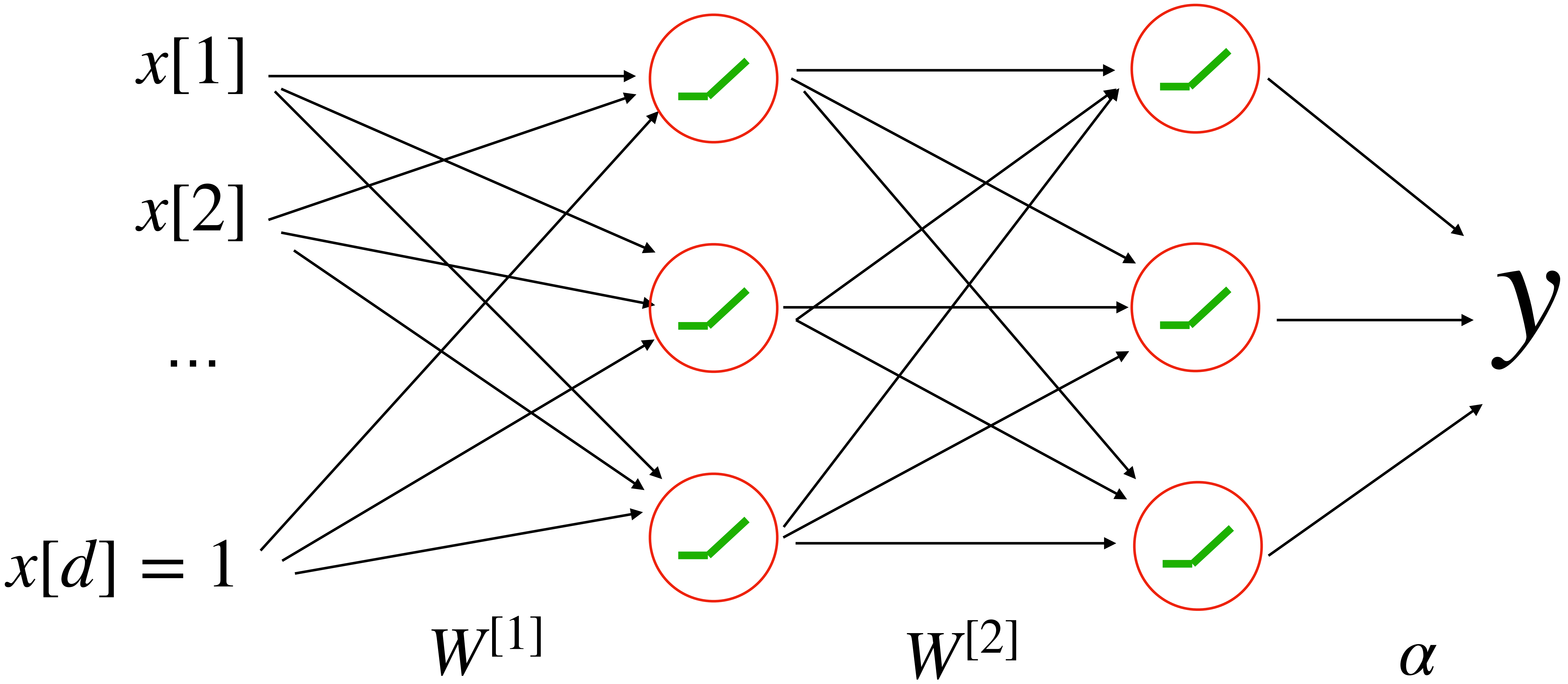
Claim: a wide enough one layer NN can approximate any smooth functions



A multi-layer fully connected neural network

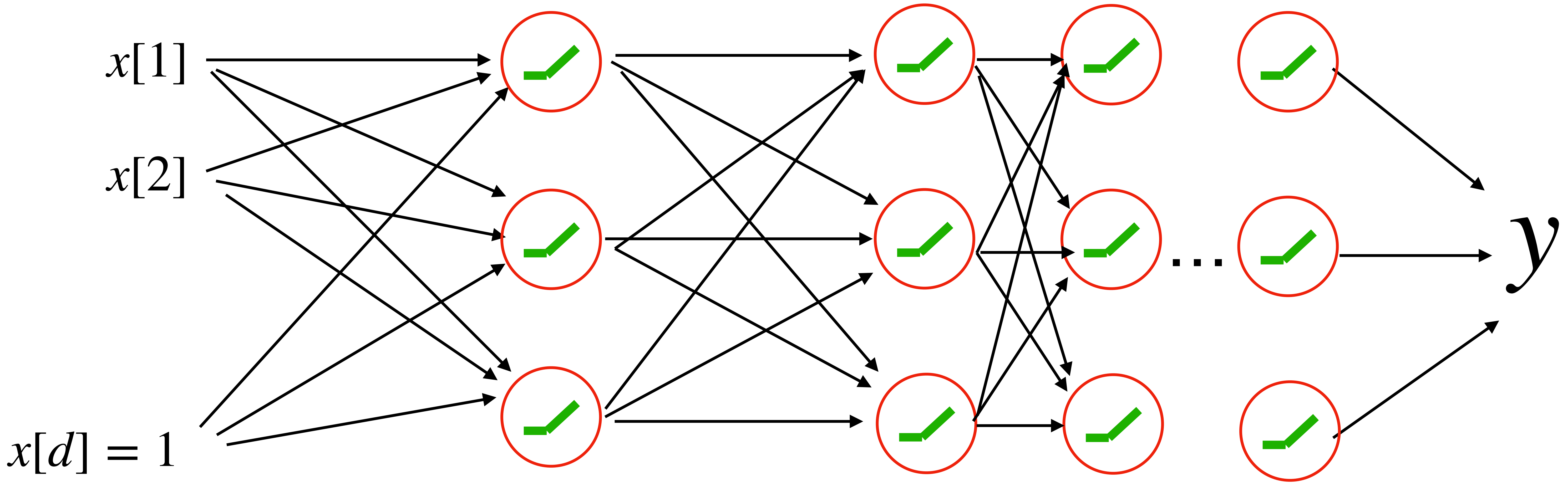


A multi-layer fully connected neural network

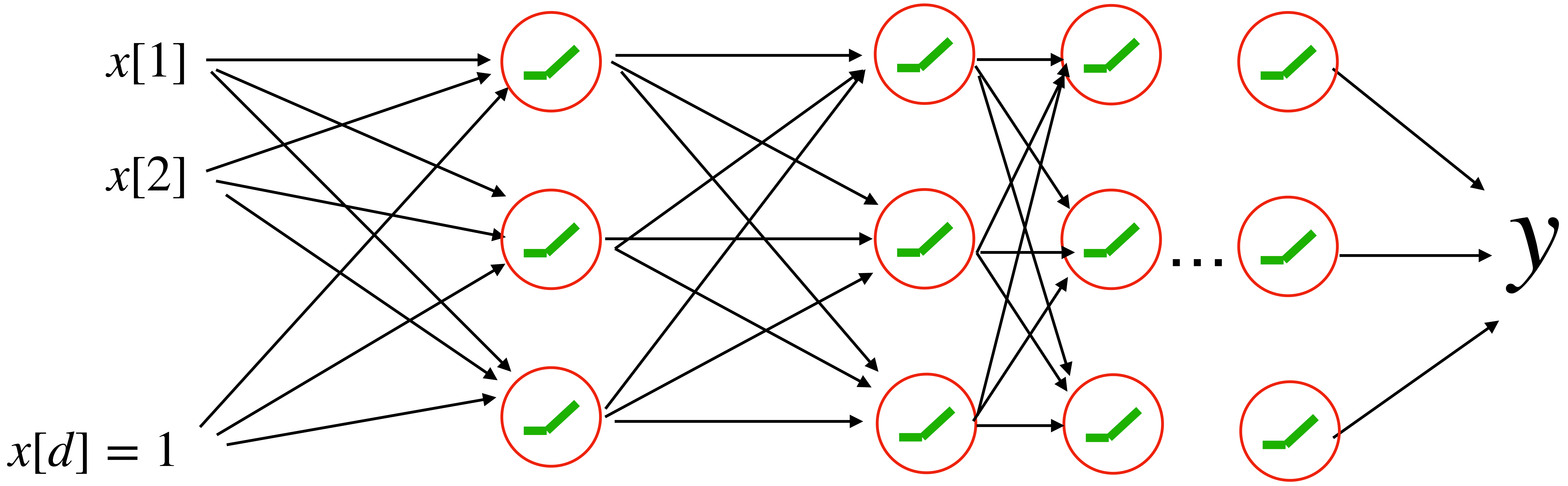


$$y = \alpha^{\top}\text{ReLU}\left(W^{[2]}\text{ReLU}\left(W^{[1]}x\right)\right) + b$$

A multi-layer fully connected neural network

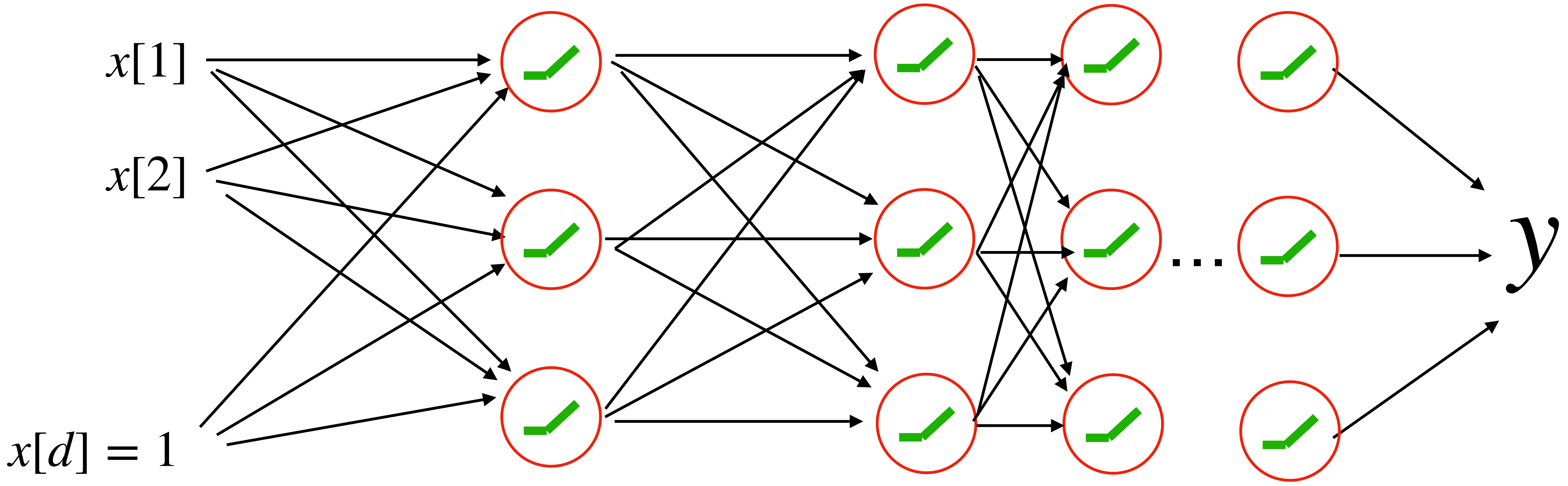


A multi-layer fully connected neural network



Define it by a forward pass:

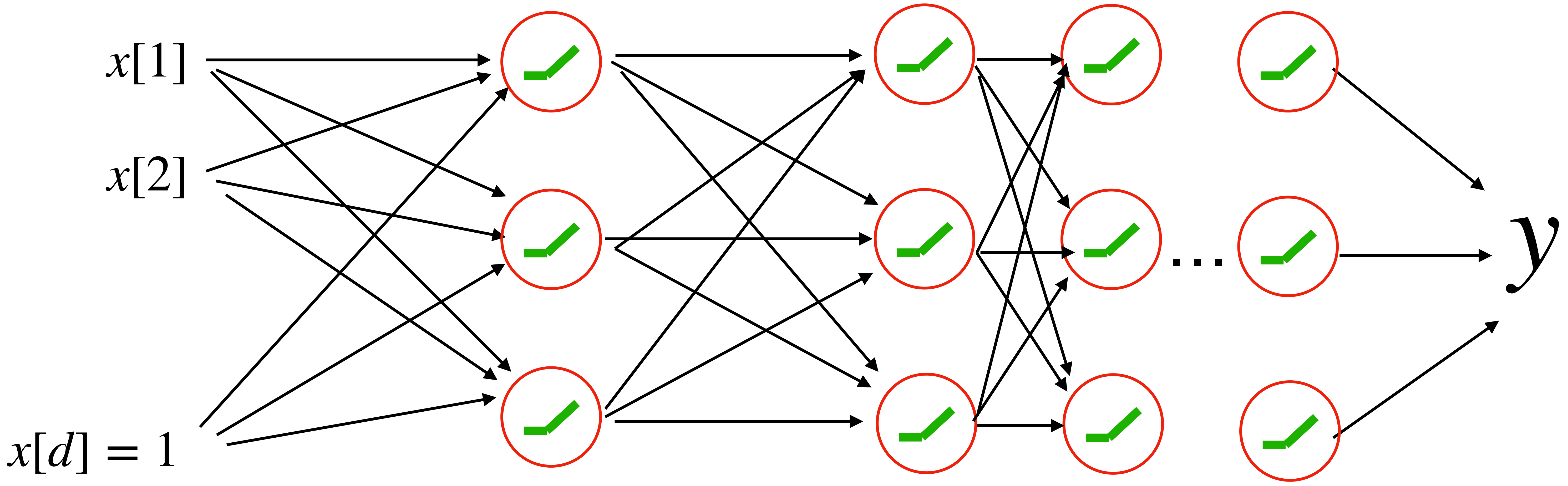
A multi-layer fully connected neural network



Define it by a forward pass:

$$z^{[1]} = x$$

A multi-layer fully connected neural network

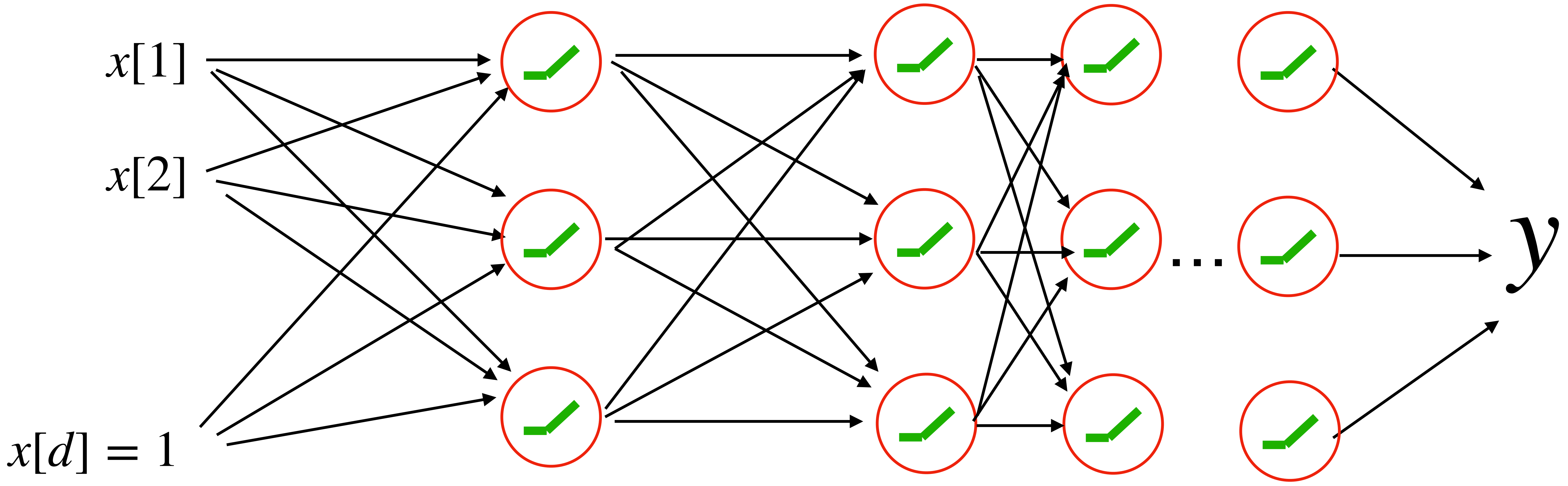


Define it by a forward pass:

$$z^{[1]} = x$$

For t = 1 to T-1:

A multi-layer fully connected neural network



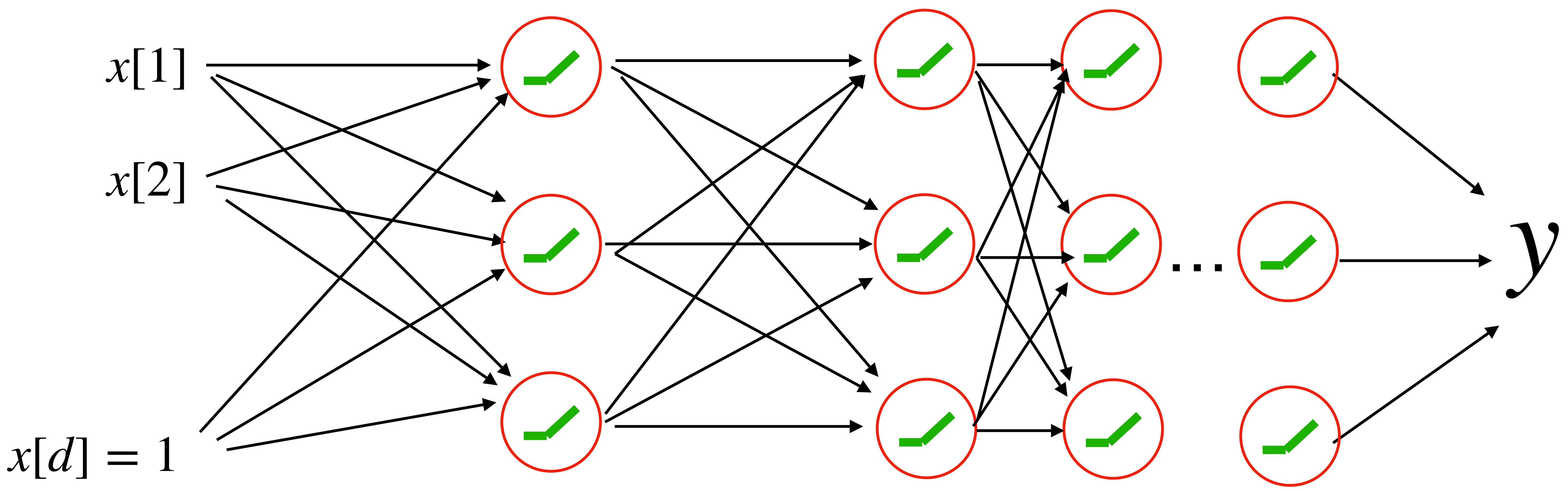
Define it by a forward pass:

$$z^{[1]} = x$$

For t = 1 to T-1:

$$z^{[t+1]} = \text{ReLU}\left(W^{[t]} z^{t}\right)$$

A multi-layer fully connected neural network



Define it by a forward pass:

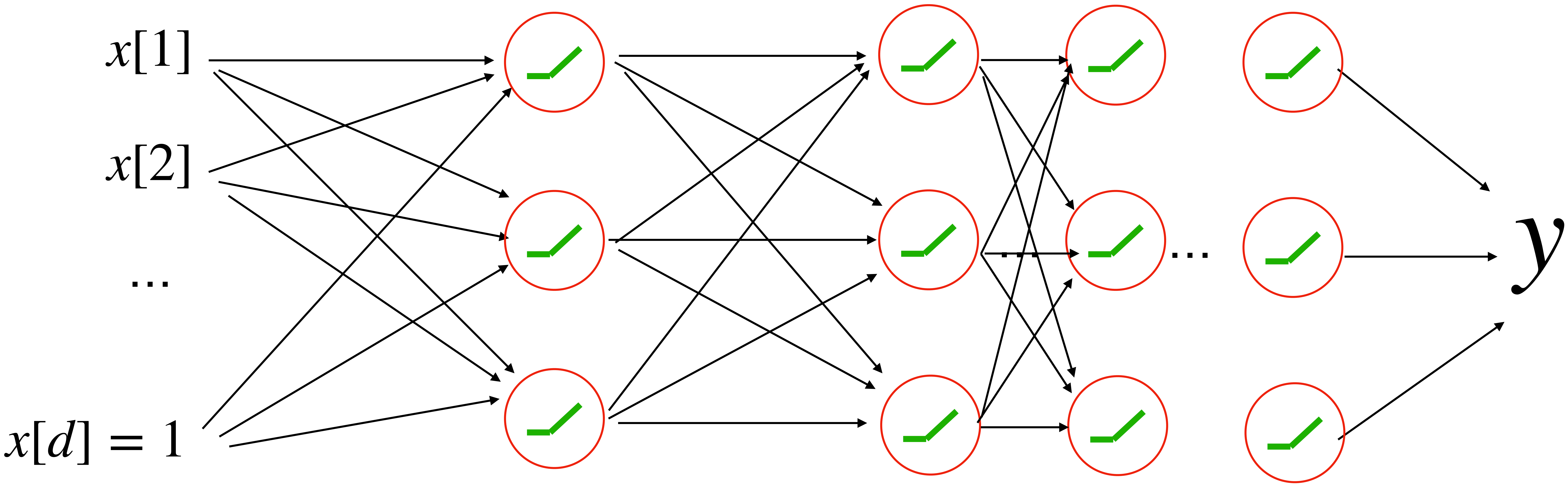
$$z^{[1]} = x$$

For t = 1 to T-1:

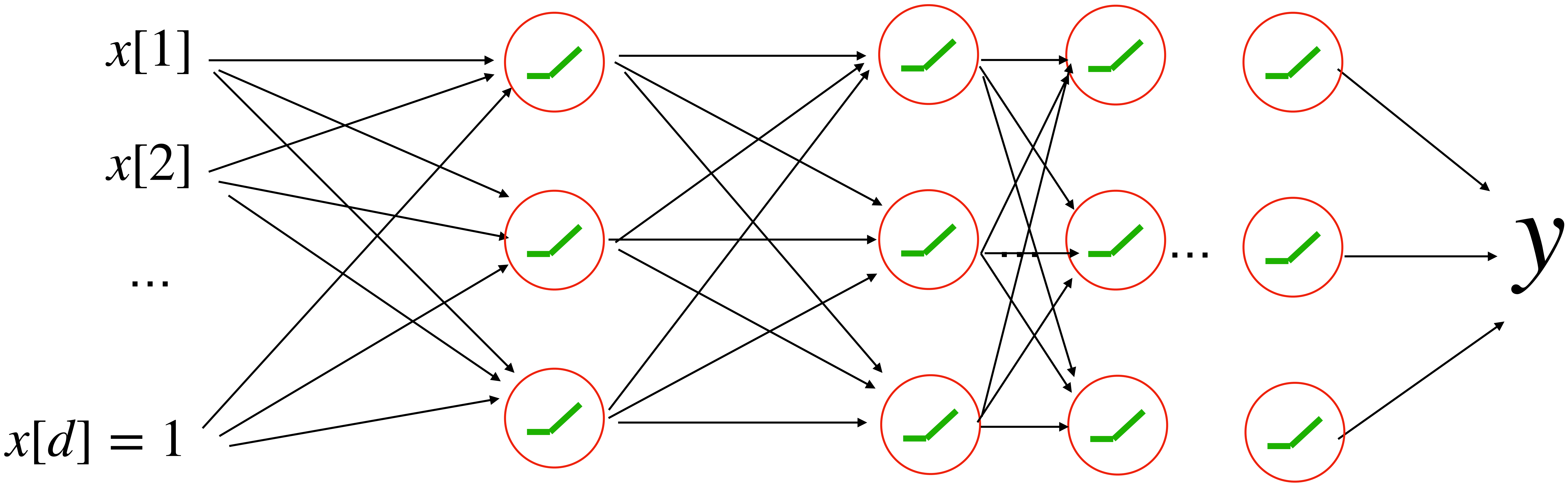
$$z^{[t+1]} = \text{ReLU}\left(W^{[t]} z^{t}\right)$$

$$y = \alpha^\top z^{[T]} + b$$

The benefits of going deep



The benefits of going deep



Allows us to represent complicated functions without making NN too wide

Outline of Today

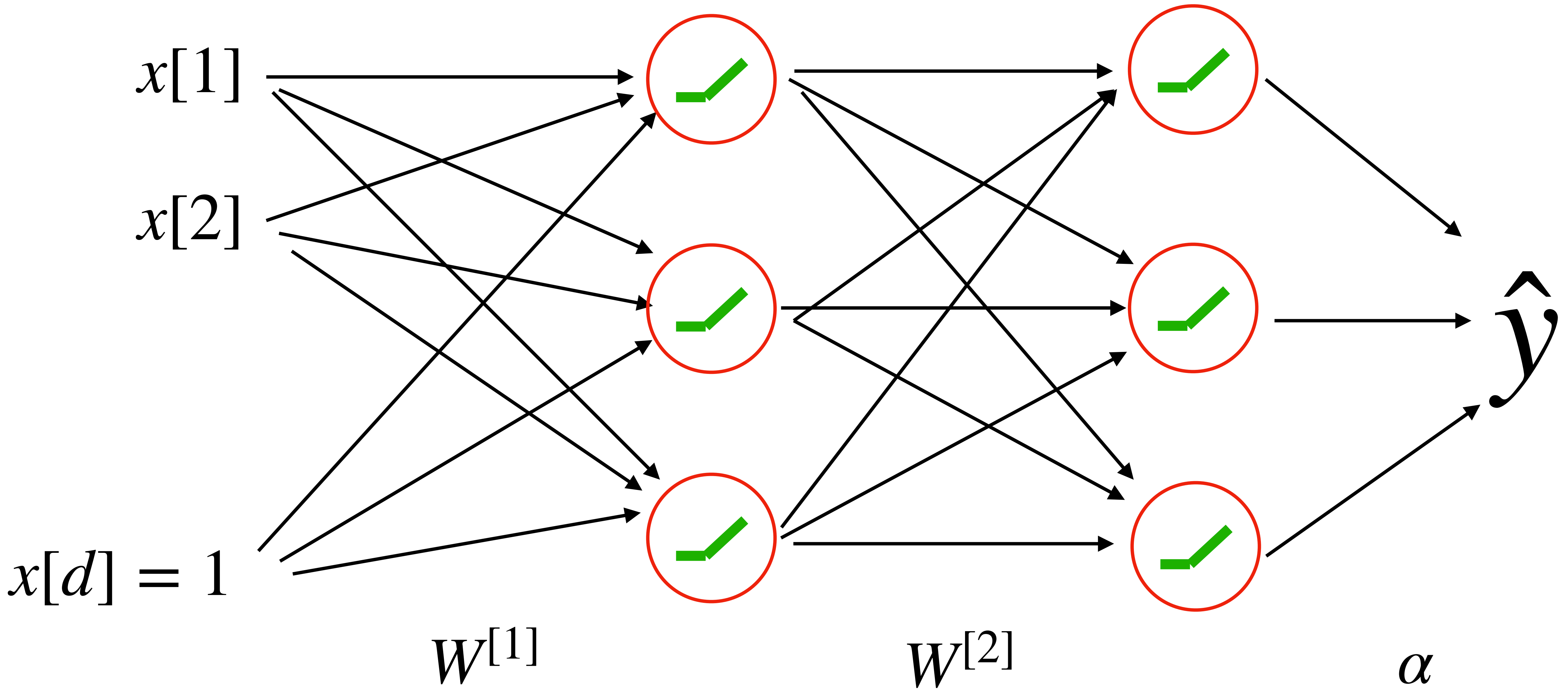
1. Analysis of Boosting

2. Multilayer feedforward Neural Network

3. Training a neural network

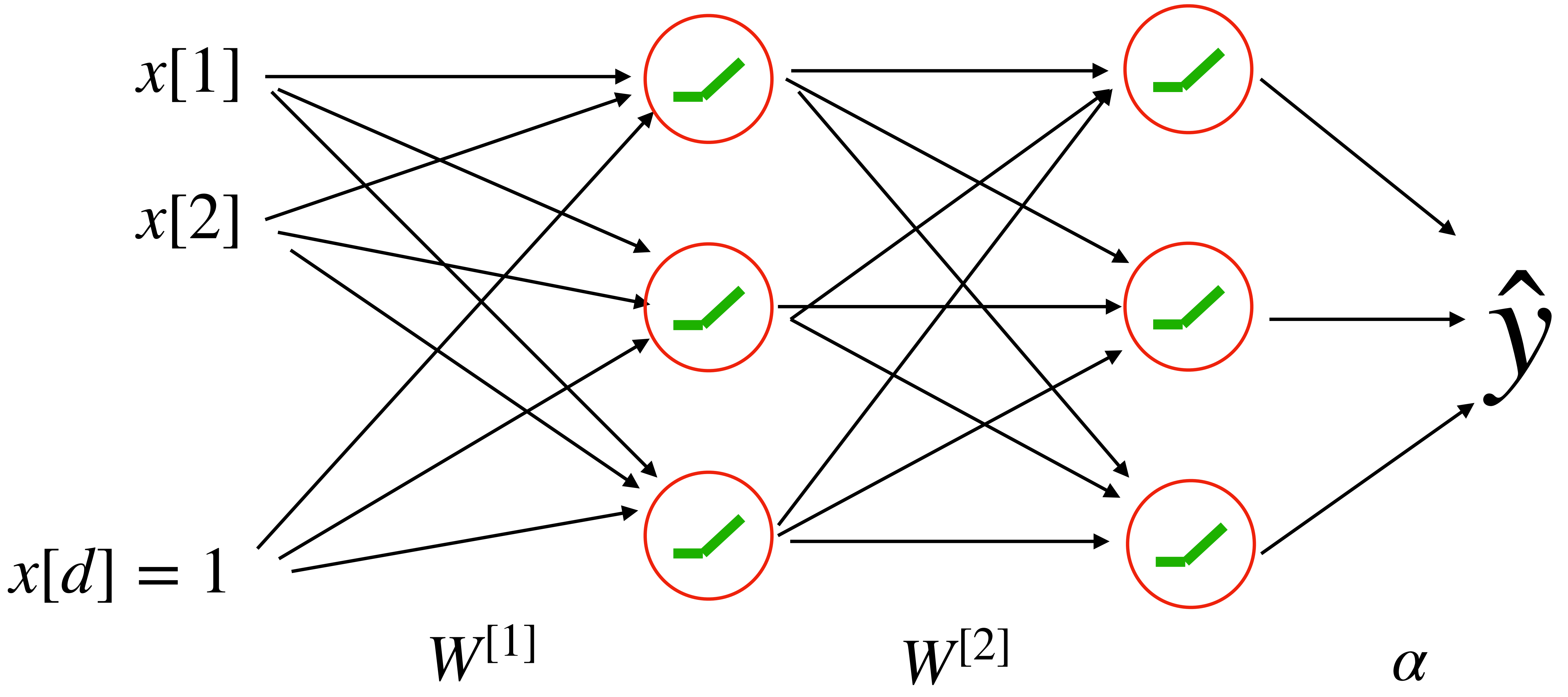
Training neural network via SGD

$$h(x) := \alpha^\top \text{ReLU}\left(W^{[2]}\text{ReLU}\left(W^{[1]}x\right)\right) + b$$



Training neural network via SGD

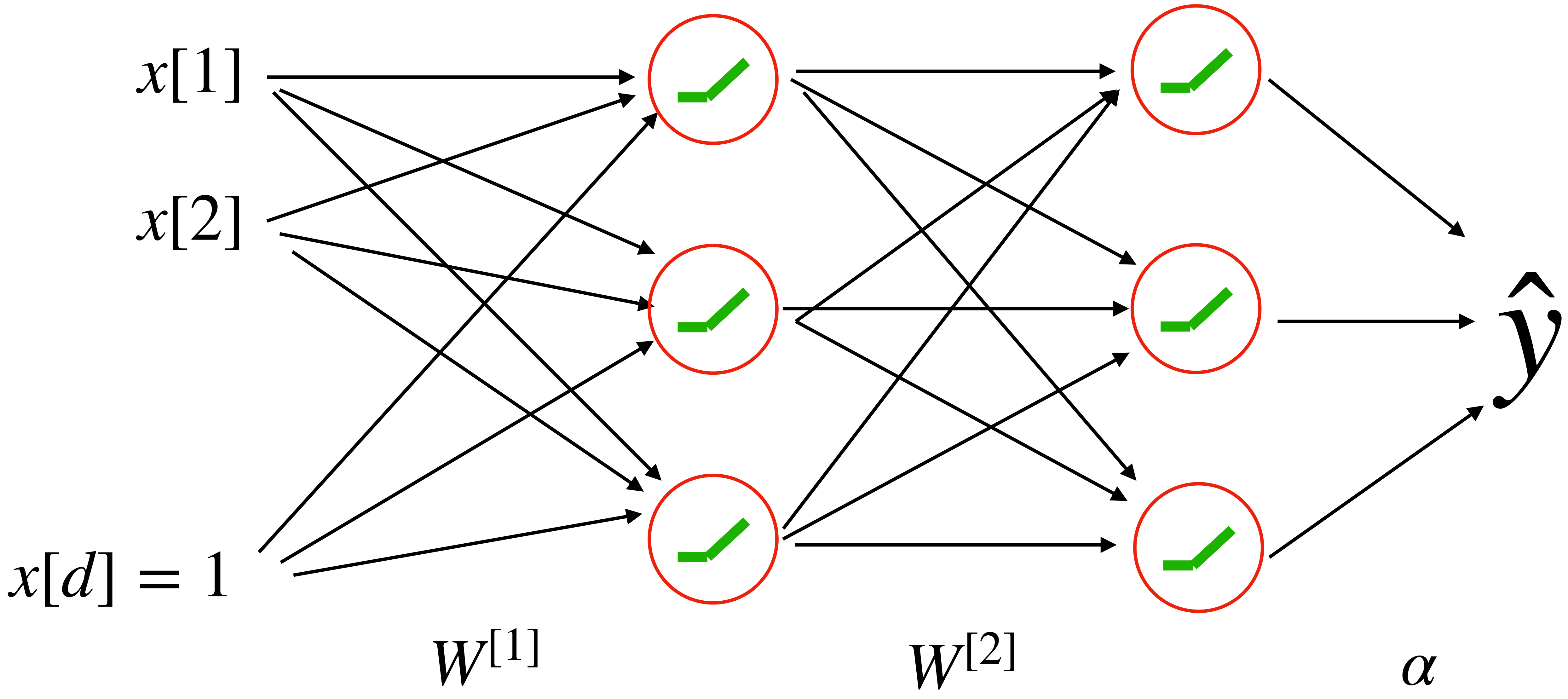
$$h(x) := \alpha^\top \text{ReLU}\left(W^{[2]}\text{ReLU}\left(W^{[1]}x\right)\right) + b$$



Let $\ell(h(x), y)$ be any differentiable loss function

Training neural network via SGD

$$h(x) := \alpha^\top \text{ReLU}\left(W^{[2]}\text{ReLU}\left(W^{[1]}x\right)\right) + b$$



Let $\ell(h(x), y)$ be any differentiable loss function

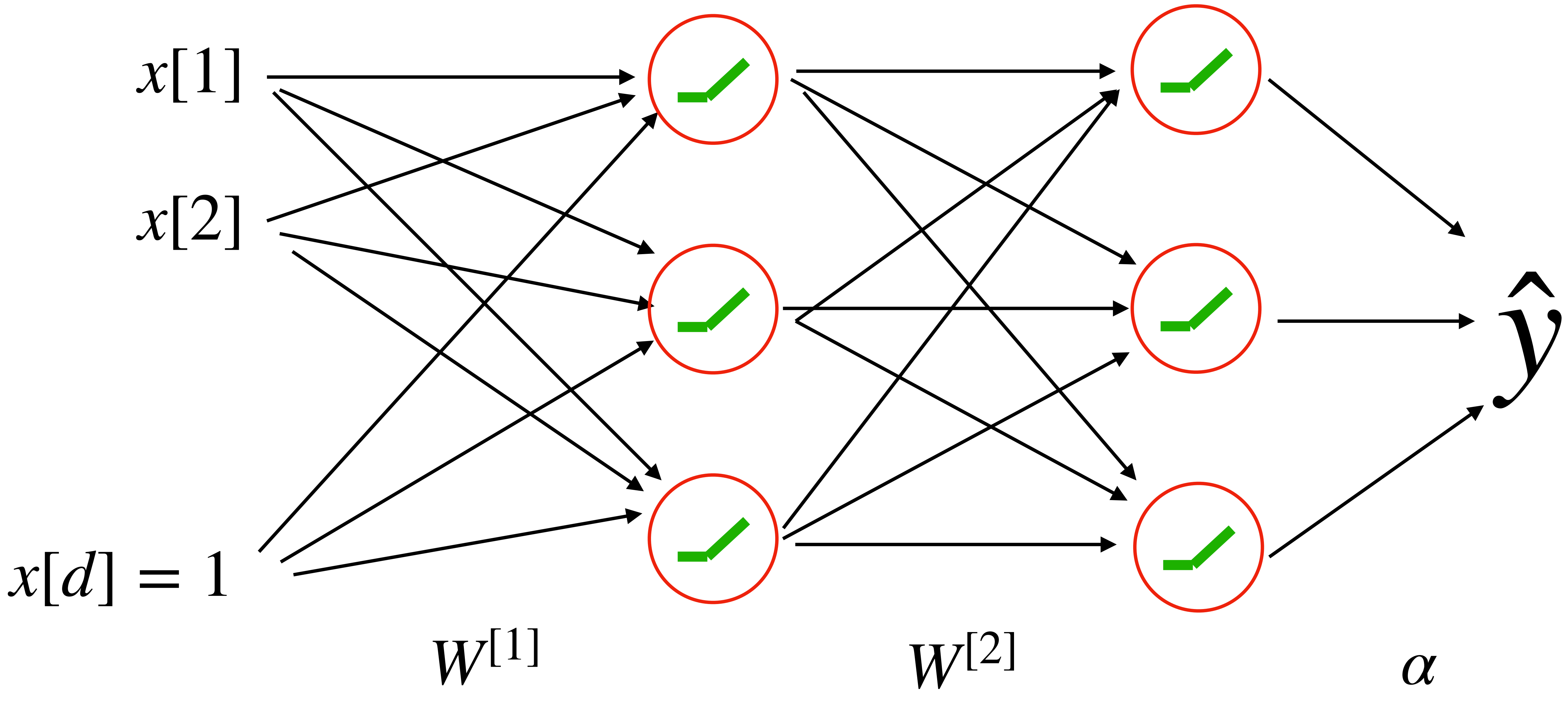
Compute gradients:

$$\frac{\partial \ell(h(x), y)}{\partial W^{[1]}} \qquad \frac{\partial \ell(h(x), y)}{\partial W^{[2]}}$$

$$\frac{\partial \ell(h(x), y)}{\alpha} \qquad \frac{\partial \ell(h(x), y)}{b}$$

Training neural network via SGD

$$h(x) := \alpha^{\top}\text{ReLU}\left(W^{[2]}\text{ReLU}\left(W^{[1]}x\right)\right) + b$$



Let $\ell(h(x), y)$ be any differentiable loss function

Compute gradients:

$$\frac{\partial \ell(h(x), y)}{\partial W^{[1]}} \qquad \frac{\partial \ell(h(x), y)}{\partial W^{[2]}}$$

$$\frac{\partial \ell(h(x), y)}{\alpha} \qquad \frac{\partial \ell(h(x), y)}{b}$$

(Next lecture: backpropagation for computing gradients)

Training neural network via SGD

Mini-batch Stochastic gradient descent

$\theta = [W^{[1]}, W^{[2]}, \alpha, b]$

```
For epoc $t = 1$ to $T$:
```

Training neural network via SGD

Mini-batch Stochastic gradient descent

$\theta = [W^{[1]}, W^{[2]}, \alpha, b]$

For epoc $t = 1$ to T: // go through dataset multiple times

Training neural network via SGD

Mini-batch Stochastic gradient descent

```
$\theta = [W^{[1]}, W^{[2]}, \alpha, b]$
For epoc $t = 1$ to $T$:        // go through dataset multiple times
    Randomly shuffle the data
```

Training neural network via SGD

Mini-batch Stochastic gradient descent

```
$\theta = [W^{[1]}, W^{[2]}, \alpha, b]$
For epoc $t = 1$ to $T$:    // go through dataset multiple times
    Randomly shuffle the data    // important (unbiased estimate of the true gradient)
```

Training neural network via SGD

Mini-batch Stochastic gradient descent

```
$\theta = [W^{[1]}, W^{[2]}, \alpha, b]$
For epoc $t = 1$ to $T$:    // go through dataset multiple times
    Randomly shuffle the data    // important (unbiased estimate of the true gradient)
    Split the data into $n/B$ many batches $\mathscr{D}_i$, each w/ size B
```

Training neural network via SGD

Mini-batch Stochastic gradient descent

```
$\theta = [W^{[1]}, W^{[2]}, \alpha, b]$
For epoc $t = 1$ to $T$:    // go through dataset multiple times
    Randomly shuffle the data    // important (unbiased estimate of the true gradient)
    Split the data into $n/B$ many batches $\mathscr{D}_i$, each w/ size B
    For  j = 1 to $n/B$
```

Training neural network via SGD

Mini-batch Stochastic gradient descent

```
$\theta = [W^{[1]}, W^{[2]}, \alpha, b]$
For epoc $t = 1$ to $T$:    // go through dataset multiple times
    Randomly shuffle the data    // important (unbiased estimate of the true gradient)
    Split the data into $n/B$ many batches $\mathscr{D}_i$, each w/ size B
    For j = 1 to $n/B$
        Mini-batch gradient $g = \sum_{x,y \in \mathscr{D}_i} \nabla_\theta \ell(h_\theta(x), y)/B$
```

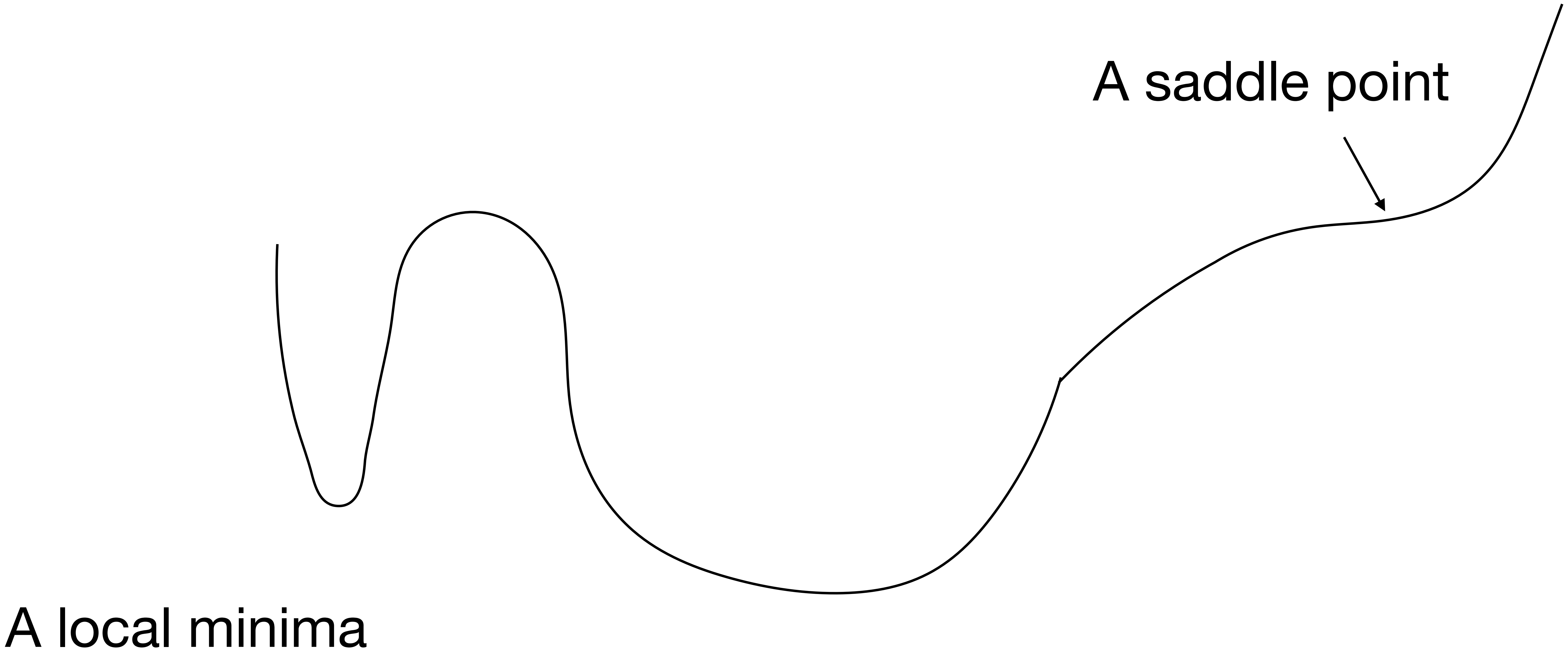
Training neural network via SGD

Mini-batch Stochastic gradient descent

```
$\theta = [W^{[1]}, W^{[2]}, \alpha, b]$
For epoc $t = 1$ to $T$:                                        // go through dataset multiple times
    Randomly shuffle the data                                   // important (unbiased estimate of the true gradient)
    Split the data into $n/B$ many batches $\mathcal{D}_i$, each w/ size B
    For j = 1 to $n/B$
        Mini-batch gradient $g = \sum_{x,y \in \mathcal{D}_i} \nabla_\theta \ell(h_\theta(x), y)/B$
        $\theta = \theta - \eta g$
```

Training neural network via SGD

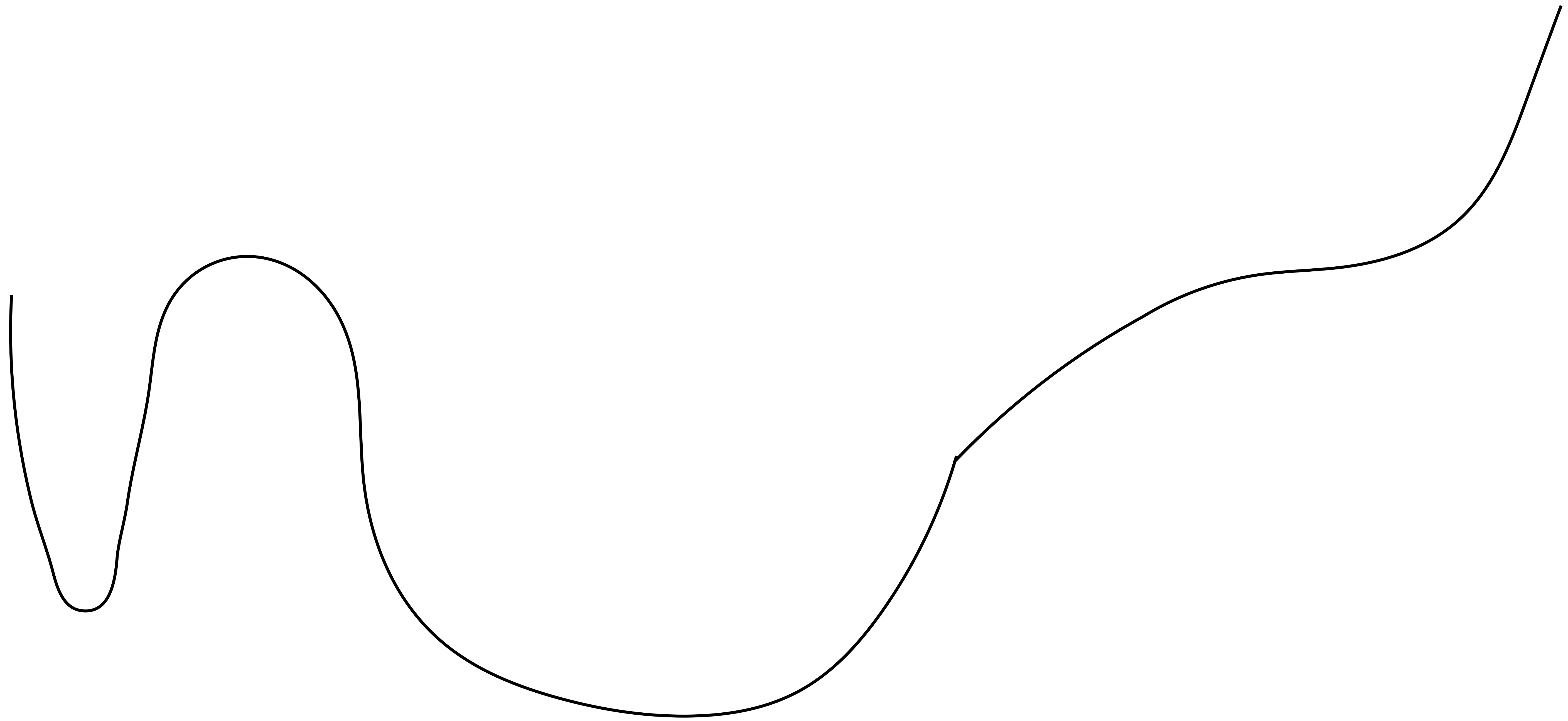
SGD helps avoiding local minima and saddle point



Training neural network via SGD

SGD tends to converge to a flat region

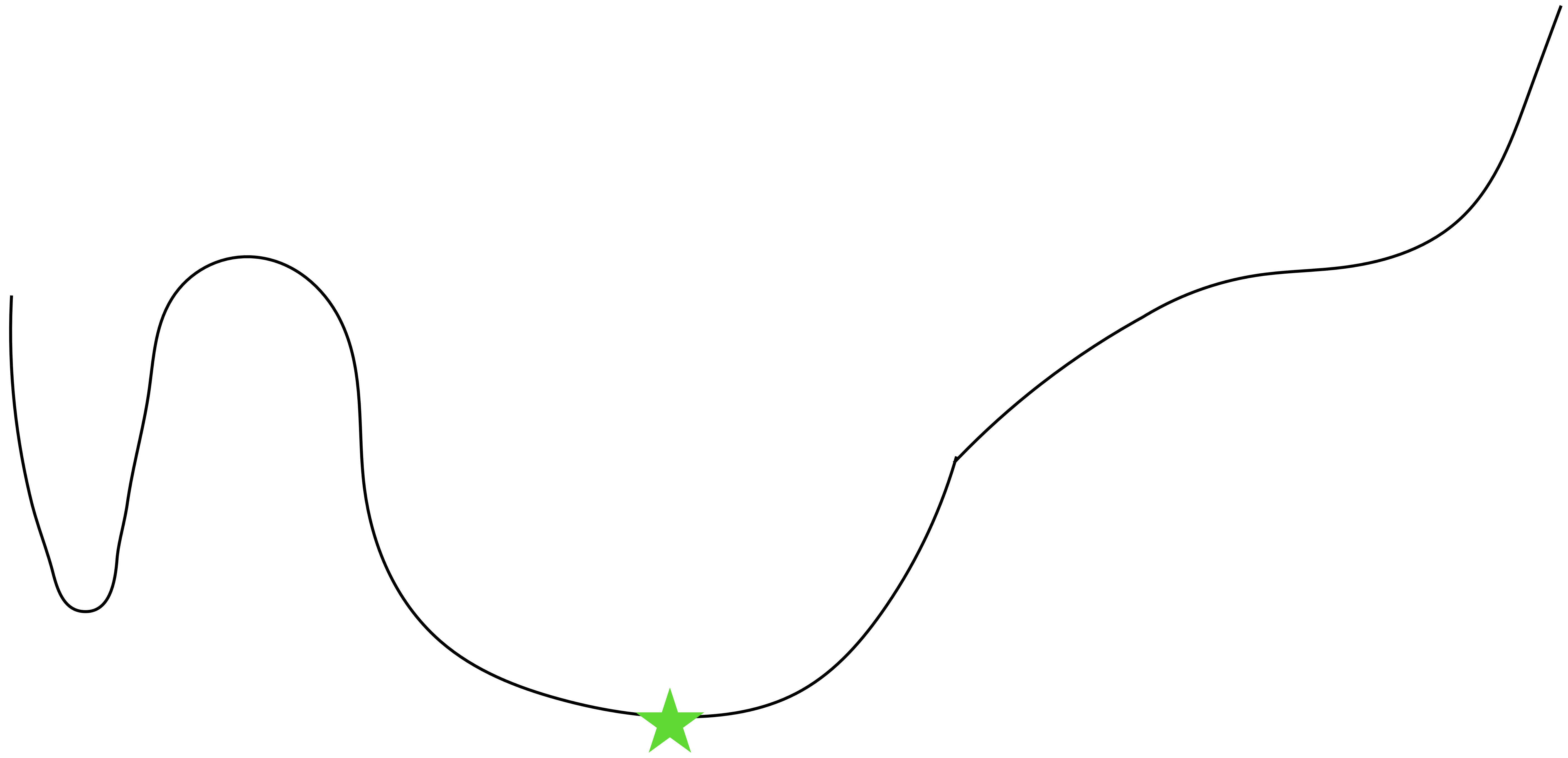
Training loss



Training neural network via SGD

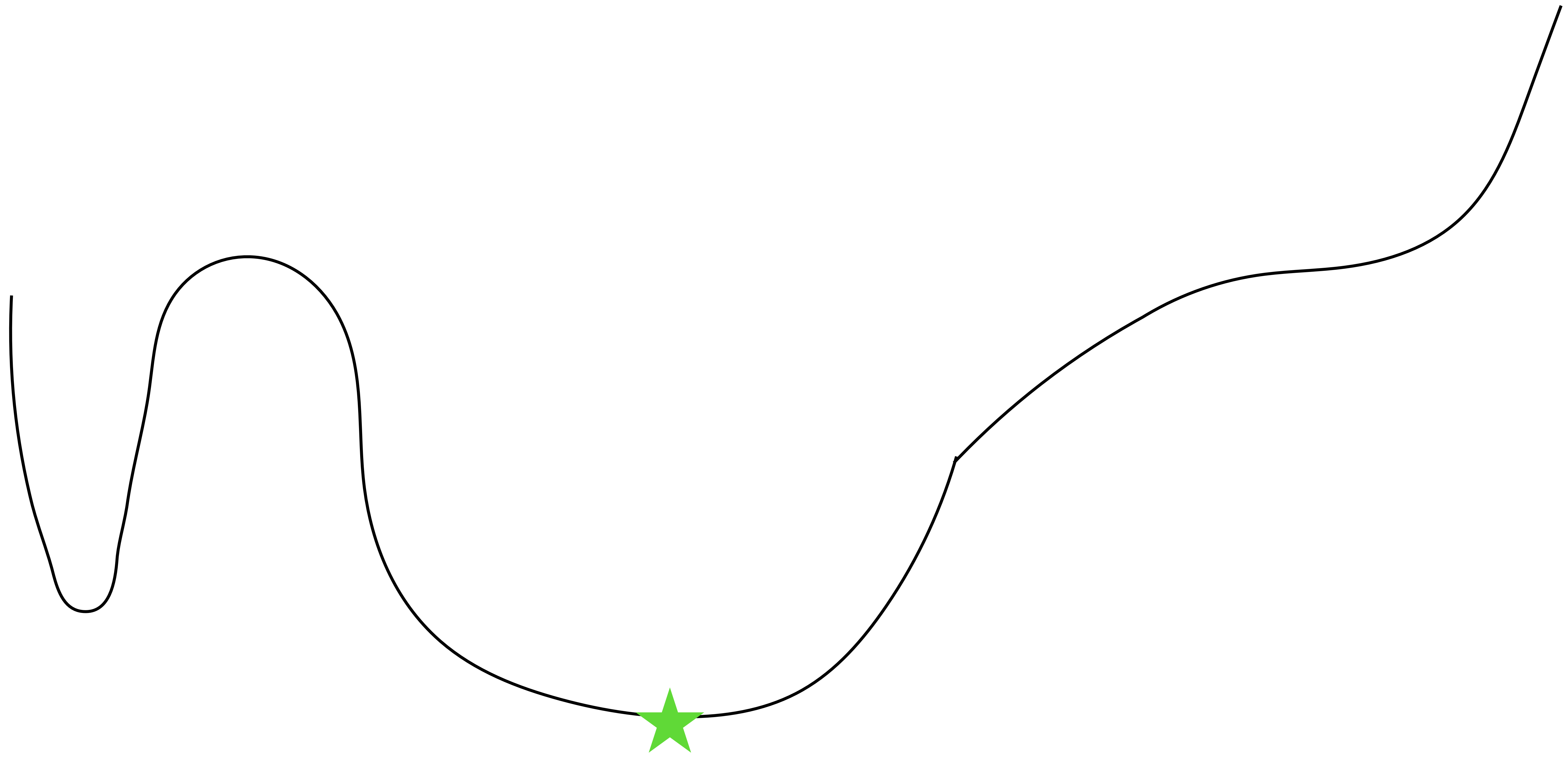
SGD tends to converge to a flat region

Training loss



Training neural network via SGD

SGD tends to converge to a flat region

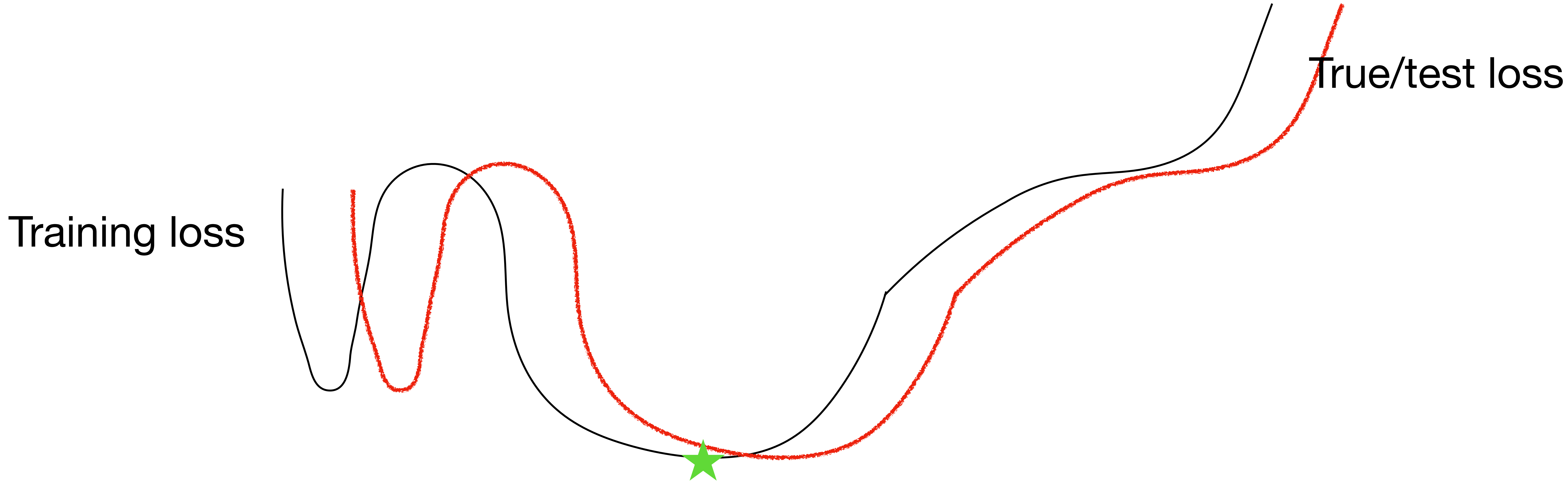


Training loss

A flat local minima solution can help generalizes better to test data

Training neural network via SGD

SGD tends to converge to a flat region



A flat local minima solution can help generalizes better to test data

Connecting neural network with kernels

Consider a NN $f(x; \theta)$

(the Neural Tangent Kernel theorem)

Connecting neural network with kernels

Consider a NN $f(x;\theta)$

Let's do a first order Taylor expansion around initialization θ_0

$$f(x;\theta) \approx f(x;\theta_0) + \nabla_\theta f(x;\theta_0)^\top (\theta - \theta_0)$$

(the Neural Tangent Kernel theorem)

Connecting neural network with kernels

Consider a NN $f(x;\theta)$

Let's do a first order Taylor expansion around initialization θ_0

$$f(x;\theta) \approx f(x;\theta_0) + \nabla_\theta f(x;\theta_0)^\top(\theta - \theta_0)$$

feature $\phi(x)$

(the Neural Tangent Kernel theorem)

Connecting neural network with kernels

Consider a NN $f(x;\theta)$

Let’s do a first order Taylor expansion around initialization θ_0

$$f(x;\theta) \approx f(x;\theta_0) + \nabla_\theta f(x;\theta_0)^\top(\theta - \theta_0)$$

feature $\phi(x)$

$$K(x,x') = \phi(x)^\top \phi(x')$$

(the Neural Tangent Kernel theorem)

Connecting neural network with kernels

Consider a NN $f(x;\theta)$

Let's do a first order Taylor expansion around initialization θ_0

$$f(x;\theta) \approx f(x;\theta_0) + \underbrace{\nabla_\theta f(x;\theta_0)^\top}_{\text{feature } \phi(x)}(\theta - \theta_0)$$

$$K(x,x') = \phi(x)^\top \phi(x')$$

If NN training does not move θ to far away from θ_0, this is behaving like kernel regression

(the Neural Tangent Kernel theorem)

Summary for today

1. Neural network is universal function approximation

2. SGD is important for training neural networks

Next lecture: backpropagation