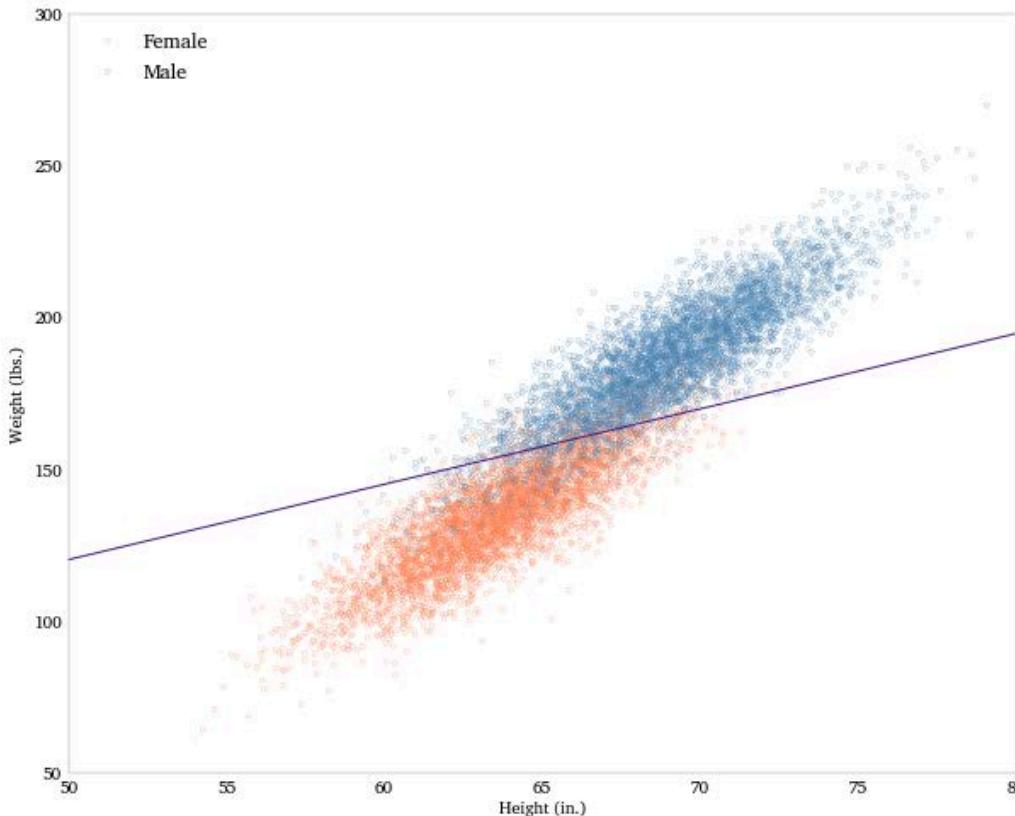


# Multi-Layer Perceptrons

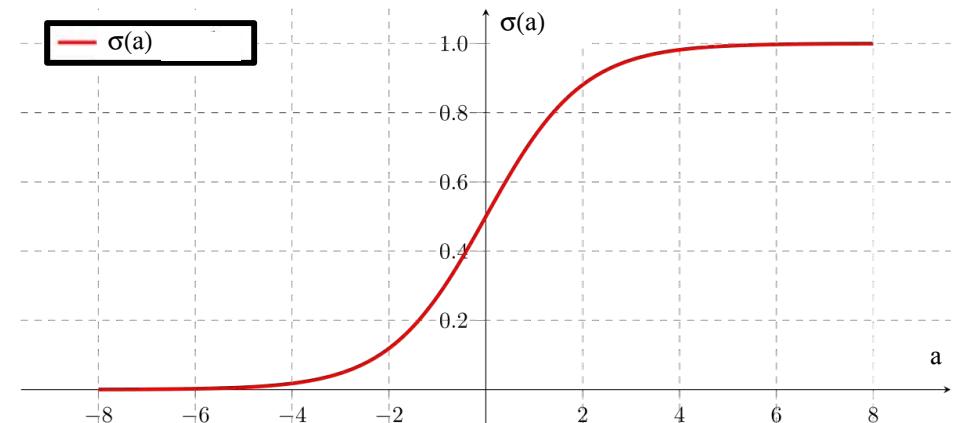
Pascal Fua  
IC-CVLab

# Reminder: Logistic Regression



$$y(\mathbf{x}; \tilde{\mathbf{w}}) = \sigma(\tilde{\mathbf{w}} \cdot \tilde{\mathbf{x}})$$

$$= \frac{1}{1 + \exp(-\tilde{\mathbf{w}} \cdot \tilde{\mathbf{x}})}$$

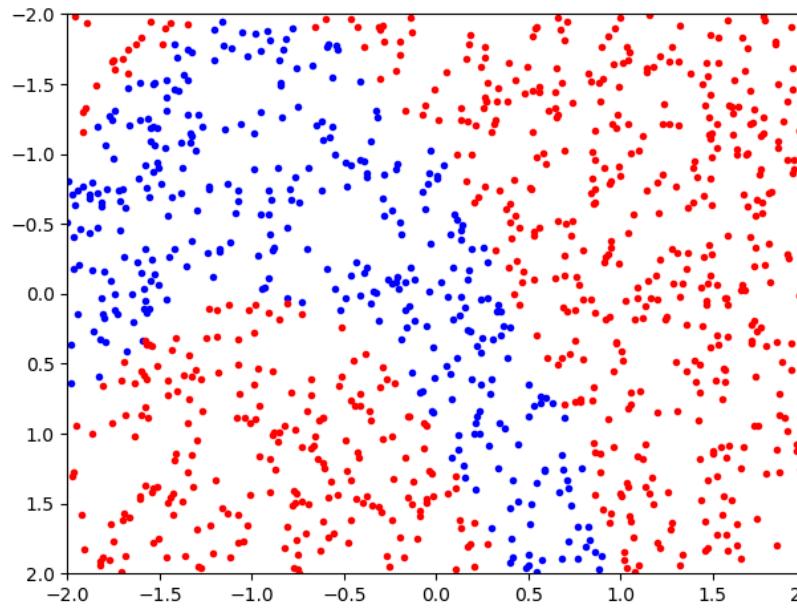
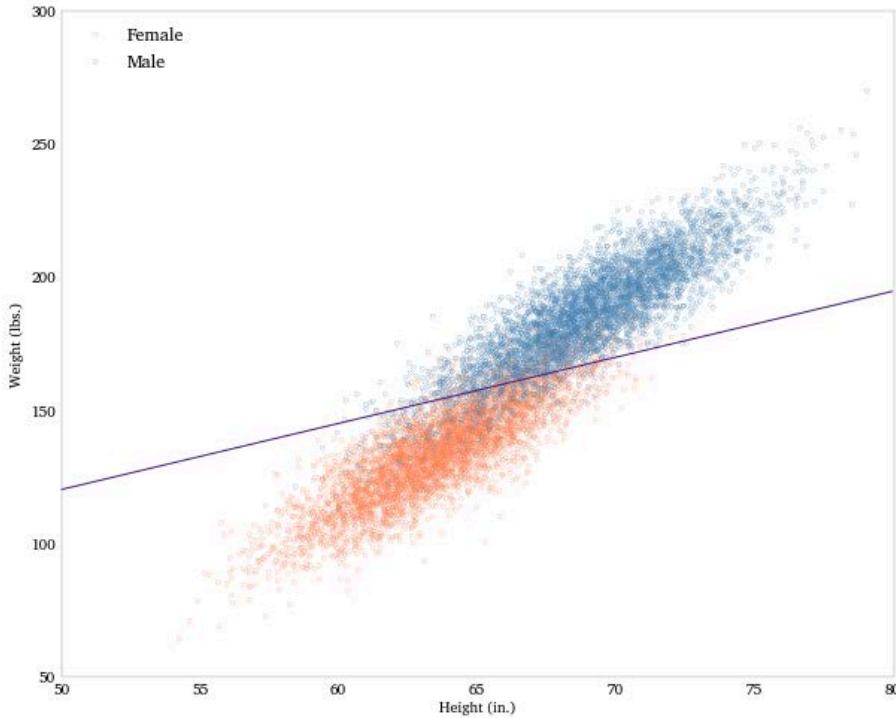


Given a **training** set  $\{(\mathbf{x}_n, t_n)_{1 \leq n \leq N}\}$  minimize

$$-\sum_n (t_n \ln y(\mathbf{x}_n) + (1 - t_n) \ln(1 - y(\mathbf{x}_n)))$$

with respect to  $\tilde{\mathbf{w}}$ .

# Reminder: Non Separable Distribution

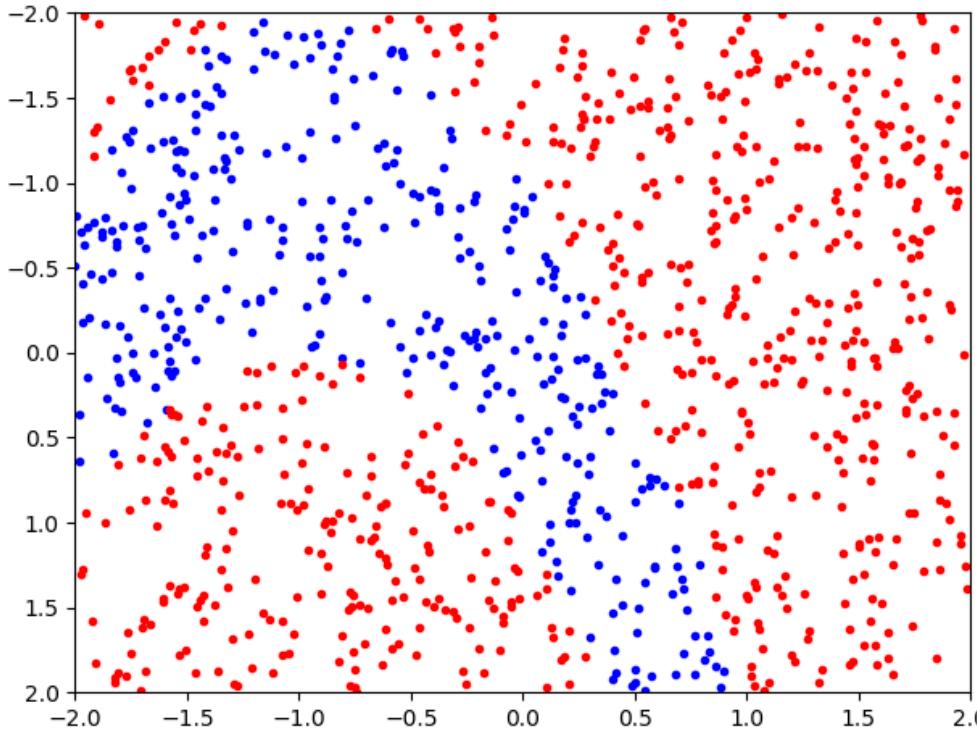


Positive:  $100(x_2 - x_1^2)^2 + (1 - x_1)^2 < 0.5$

Negative: Otherwise

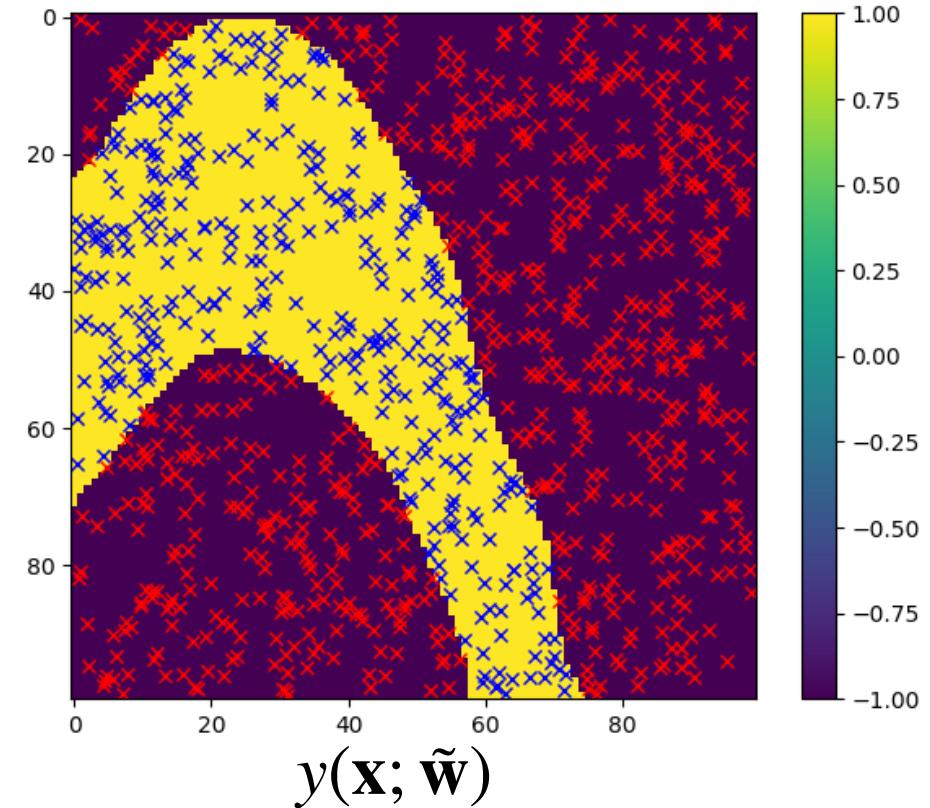
- Logistic regression can handle a few outliers.
- But not a complex non-linear boundary.

# Reminder: Non Separable Distribution



Positive:  $100(x_2 - x_1^2)^2 + (1 - x_1)^2 < 0.5$

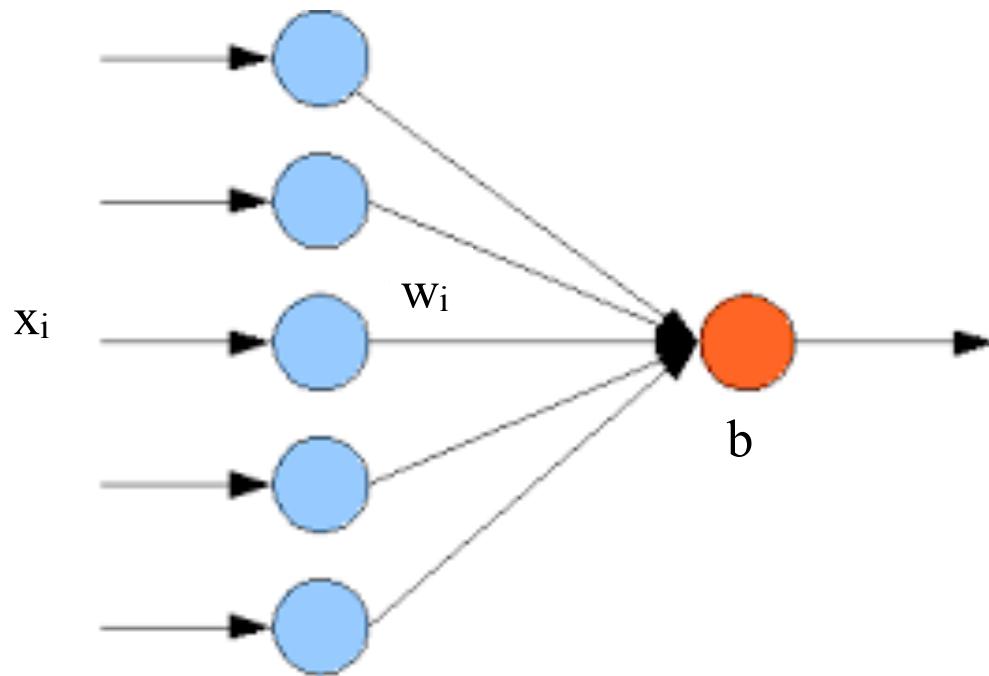
Negative: Otherwise



How can we learn a function  $y$  such that  $y(\mathbf{x}; \tilde{\mathbf{w}})$  is close to 1 for positive samples and close to 0 or -1 for negative ones:

- AdaBoost: Use several hyperplanes.
- SVMs: Map to a higher dimension.
- Neural Nets: Map to a higher dimension and use lots of hyperplanes.

# Reformulating Logistic Regression

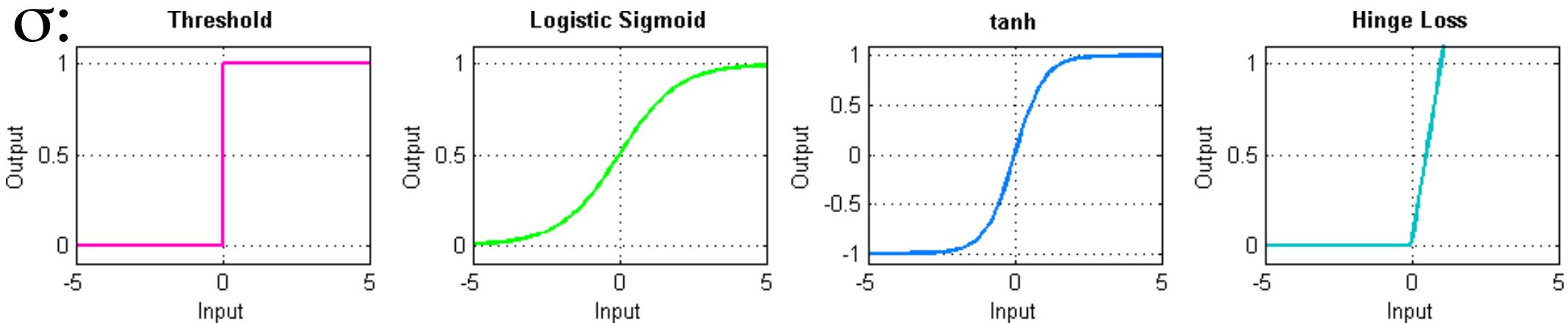


$$y(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

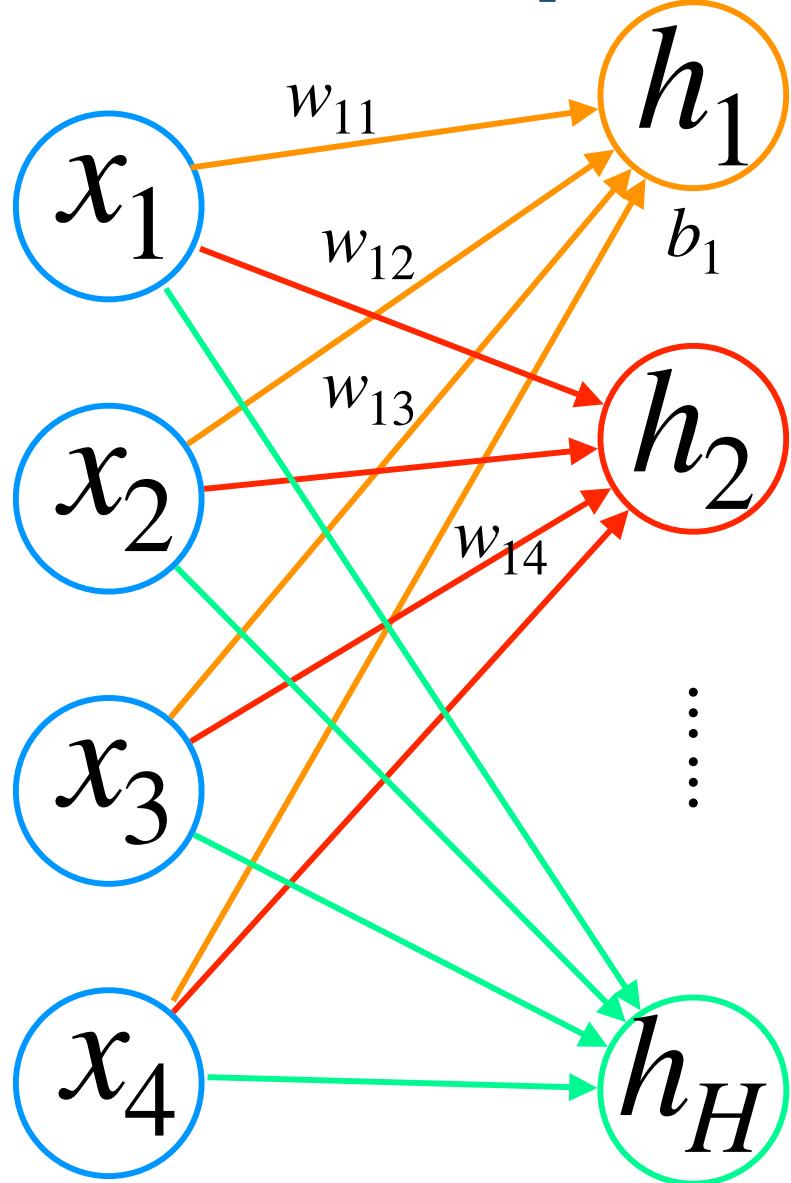
$$\mathbf{x} = [x_1, x_2, \dots, x_n]^T$$

$$\mathbf{w} = [w_1, w_2, \dots, w_n]^T$$

$\sigma$ :



# Repeating the Process



$$h_1 = \sigma(\mathbf{w}_1 \cdot \mathbf{x} + b_1)$$

$$\mathbf{w}_1 = [w_{11}, w_{12}, w_{13}, w_{14}]^T$$

$$h_2 = \sigma(\mathbf{w}_2 \cdot \mathbf{x} + b_2)$$

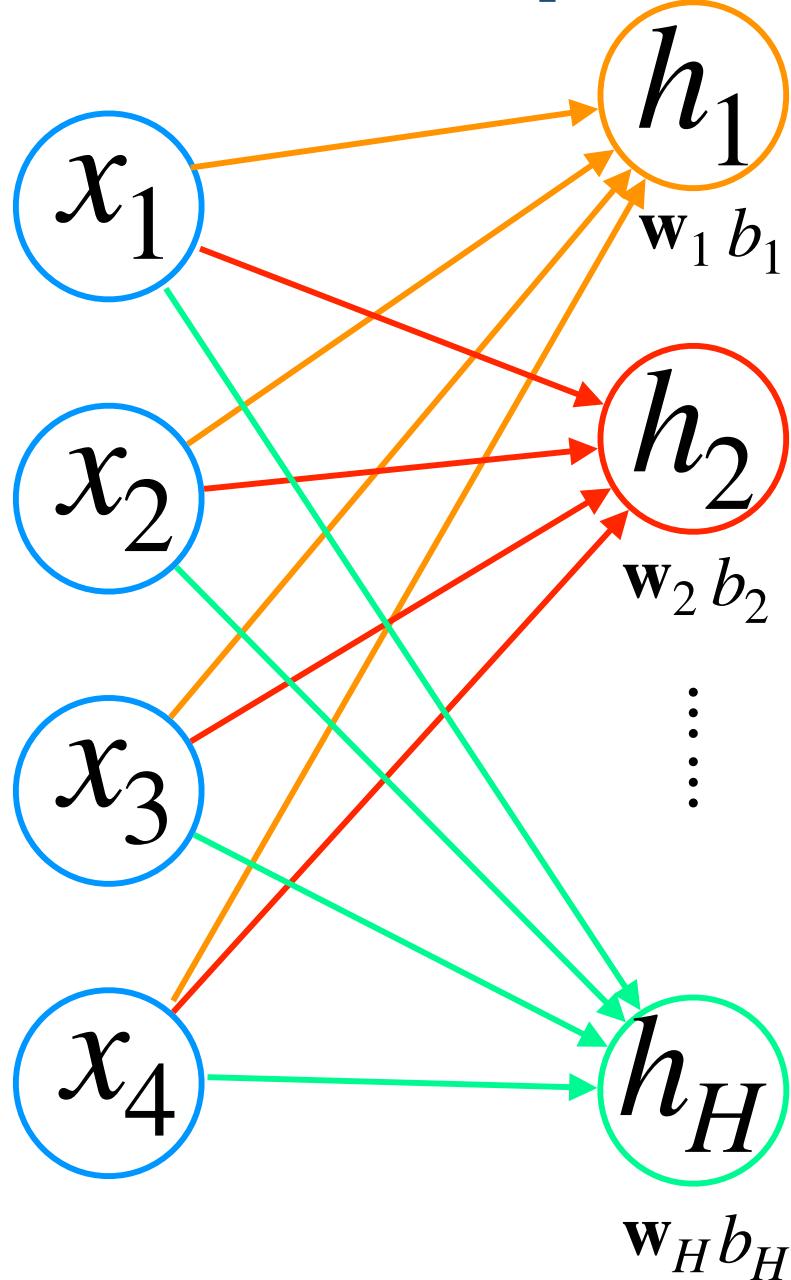
$$\mathbf{w}_2 = [w_{21}, w_{22}, w_{23}, w_{24}]^T$$

⋮  
⋮

$$h_H = \sigma(\mathbf{w}_H \cdot \mathbf{x} + b_H)$$

$$\mathbf{w}_H = [w_{H1}, w_{H2}, w_{H3}, w_{H4}]^T$$

# Repeating the Process

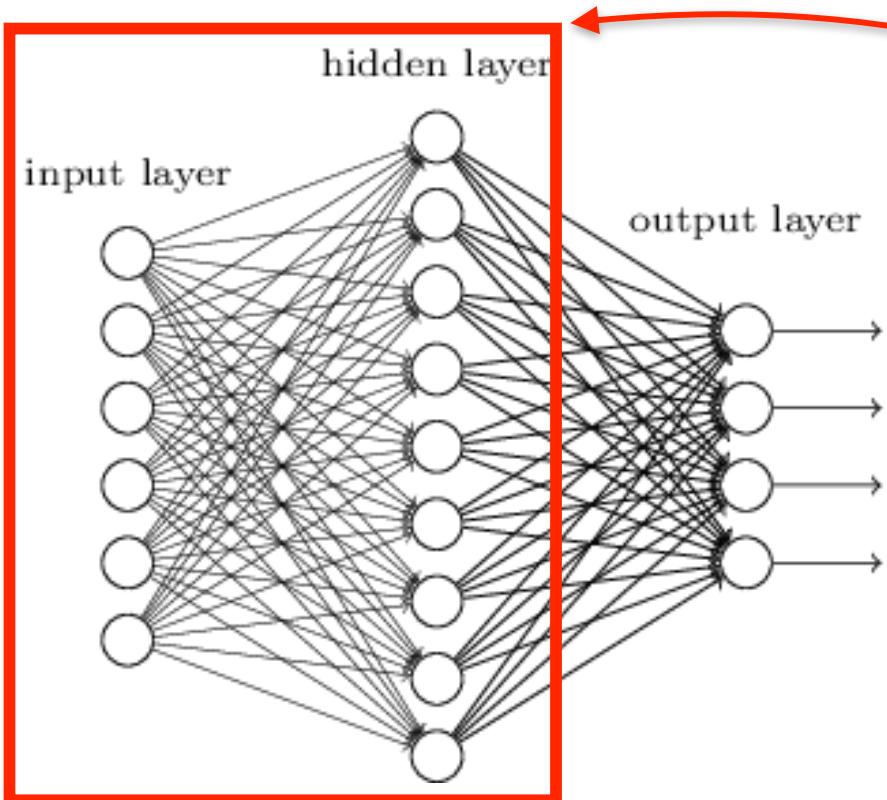


$$\mathbf{h} = \sigma(\mathbf{W}\mathbf{x} + \mathbf{b}) ,$$

$$\text{with } \mathbf{W} = \begin{bmatrix} \mathbf{w}_1 \\ \mathbf{w}_2 \\ \vdots \\ \mathbf{w}_H \end{bmatrix}$$

$$\text{and } \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_H \end{bmatrix} .$$

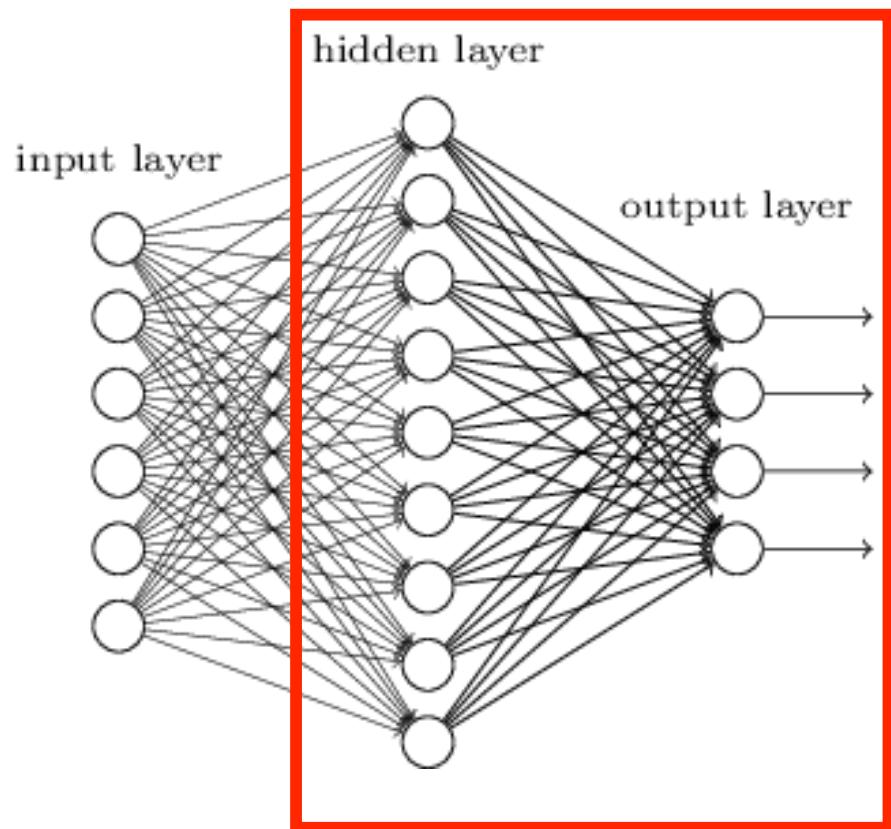
# Multi-Layer Perceptron (MLP)



$$\begin{aligned} \mathbf{h} &= \sigma_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) \\ \mathbf{y} &= \sigma_2(\mathbf{W}_2 \mathbf{h} + \mathbf{b}_2) \end{aligned}$$

- The process can be repeated several times to create a vector  $\mathbf{h}$ .

# Multi-Layer Perceptron (MLP)



$$\begin{aligned}\mathbf{h} &= \sigma_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) \\ \mathbf{y} &= \sigma_2(\mathbf{W}_2 \mathbf{h} + \mathbf{b}_2)\end{aligned}$$

- The process can be repeated several times to create a vector  $\mathbf{h}$ .
- It can then be done again to produce an output  $\mathbf{y}$ .

—> This output is a **differentiable** function of the weights.

# Binary Case

- Let the training set be  $\{(\mathbf{x}_n, t_n)_{1 \leq n \leq N}\}$  where  $t_n \in \{0, 1\}$  is the class label and let us consider a neural net with a 1D output.
- We write

$$y_n = \sigma(\mathbf{w}_2(\sigma(\mathbf{W}_1 \mathbf{x}_n + \mathbf{b}_1)) + \mathbf{b}_2) \in [0, 1]$$

w<sub>2</sub> is a vector in this case

- We want to minimize the binary cross entropy

$$E(\mathbf{W}_1, \mathbf{w}_2, \mathbf{b}_1, \mathbf{b}_2) = \frac{1}{N} \sum_{n=1}^N E_n(\mathbf{W}_1, \mathbf{w}_2, \mathbf{b}_1, \mathbf{b}_2) ,$$

$$E_n(\mathbf{W}_1, \mathbf{w}_2, \mathbf{b}_1, \mathbf{b}_2) = - (t_n \ln(y_n) + (1 - t_n) \ln(1 - y_n)) ,$$

with respect to the coefficients of  $\mathbf{W}_1$ ,  $\mathbf{w}_2$ ,  $\mathbf{b}_1$ , and  $\mathbf{b}_2$ .

- E is a differentiable function and this can be done using a gradient-based technique.

# Multi-Class Case

Let the training set be  $\{(\mathbf{x}_n, [t_n^1, \dots, t_n^K])_{1 \leq n \leq N}\}$  where  $t_n^k \in \{0,1\}$  is the probability that sample  $\mathbf{x}_n$  belongs to class k.

- We write

$$\mathbf{y}_n = \sigma(\mathbf{W}_2(\sigma(\mathbf{W}_1 \mathbf{x}_n + \mathbf{b}_1)) + \mathbf{b}_2) \in R^K$$

$$p_n^k = \frac{\exp(\mathbf{y}_n[k])}{\sum_j \exp(\mathbf{y}_n[j])}$$

- We want to minimize the cross entropy

$$E(\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = \frac{1}{N} \sum_{n=1}^N E_n(\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) ,$$

$$E_n(\mathbf{W}_1, \mathbf{W}_2, \mathbf{b}_1, \mathbf{b}_2) = - \sum t_n^k \ln(p_n^k) ,$$

with respect to the coefficients of  $\mathbf{W}_1$ ,  $\mathbf{W}_2$ ,  $\mathbf{b}_1$ , and  $\mathbf{b}_2$ .

# Optional: PyTorch Translation (1)

```
class MLP(nn.Module):
```

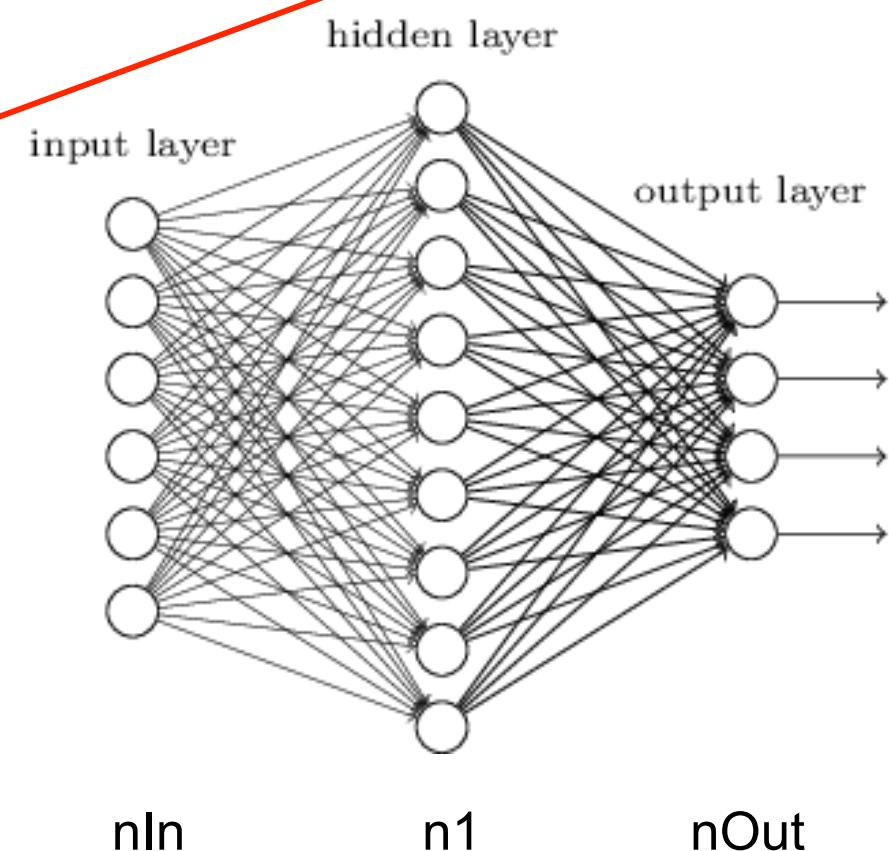
$\mathbf{W}_1$  is an  $nIn \times n_1$  matrix.

$\mathbf{W}_2$  is an  $n_1 \times nOut$  matrix.

```
def __init__(self,n1=10,nIn=2,nOut=1):  
    self.l1 = nn.Linear(nIn,n1)  
    self.l2 = nn.Linear(n1,nOut)
```

```
def forward(self,x):  
    h = sigm(self.l1(x))  
    return sigm(self.l2(h))
```

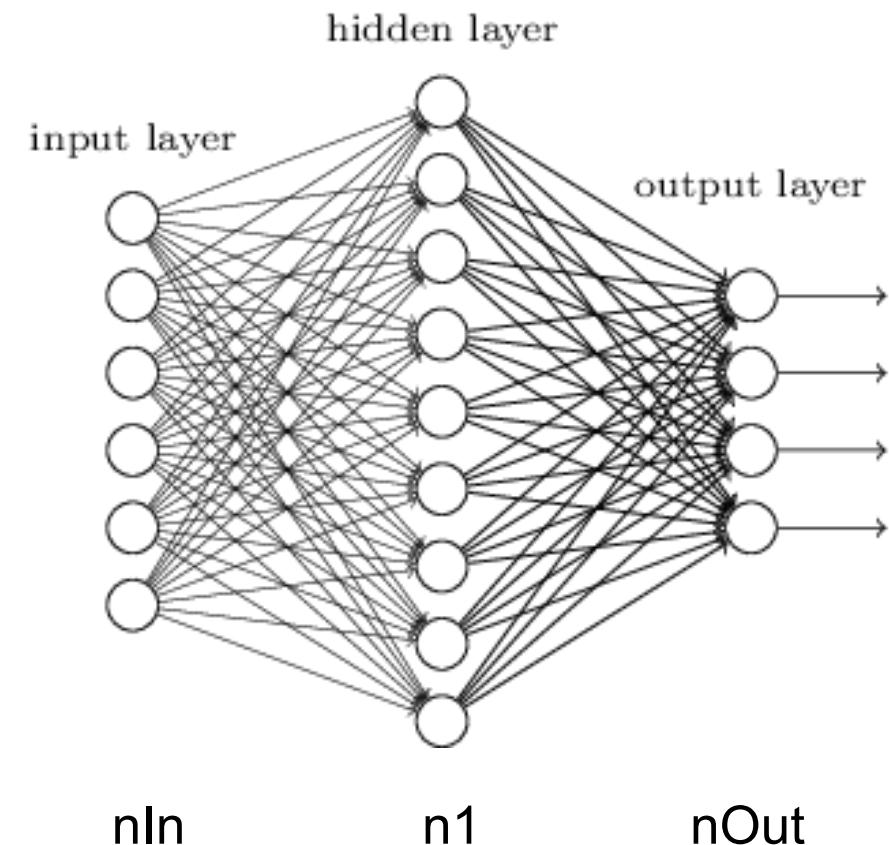
Return  $\sigma(\mathbf{W}_2(\sigma(\mathbf{W}_1\mathbf{x}_n + \mathbf{b}_1)) + \mathbf{b}_2)$



# Optional: PyTorch Translation (2)

```
class MLP(nn.Module):  
  
    def __init__(self,n1=10,nIn=2,nOut=1):  
        self.l1 = nn.Linear(nIn,n1)  
        self.l2 = nn.Linear(n1,nOut)  
  
    def forward(self,x):  
        h = sigm(self.l1(x))  
        return self.l2(h)  
  
    def loss(self,x,target):  
        loss_fn = torch.nn.CrossEntropyLoss()  
        output = self(x)  
        return loss_fn(output,target)
```

$$\text{Return} = - \sum t_n^k \ln(p_n^k)$$



# More Compact Notation

$$\mathbf{h} = \sigma_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$

$$\mathbf{y} = \sigma_2(\mathbf{W}_2 \mathbf{h} + \mathbf{b}_2)$$



$$\mathbf{h} = \sigma_1([\mathbf{W}_1 | \mathbf{b}_1] \begin{bmatrix} \mathbf{x} \\ 1 \end{bmatrix})$$

$$\mathbf{y} = \sigma_2([\mathbf{W}_2 | \mathbf{b}_2] \begin{bmatrix} \mathbf{h} \\ 1 \end{bmatrix})$$

$$\mathbf{w} = [\mathbf{w}_1 | \mathbf{b}_1 | \mathbf{w}_2 | \mathbf{b}_2]$$

$$E(\mathbf{w}) = \sum_{n=1}^N E_n(\mathbf{w})$$

$\mathbf{w}_n$ : Matrix  $\mathbf{W}_n$  represented by a 1D vector.

# More Compact Formulation

- Given a training set be  $\{(\mathbf{x}_n, [t_n^1, \dots, t_n^K])_{1 \leq n \leq N}\}$  where  $t_n^k \in \{0,1\}$  is the probability that sample  $\mathbf{x}_n$  belongs to class k, we write:

$$\mathbf{y}_n = f_{\mathbf{w}}(\mathbf{x}_n) \in R^K$$

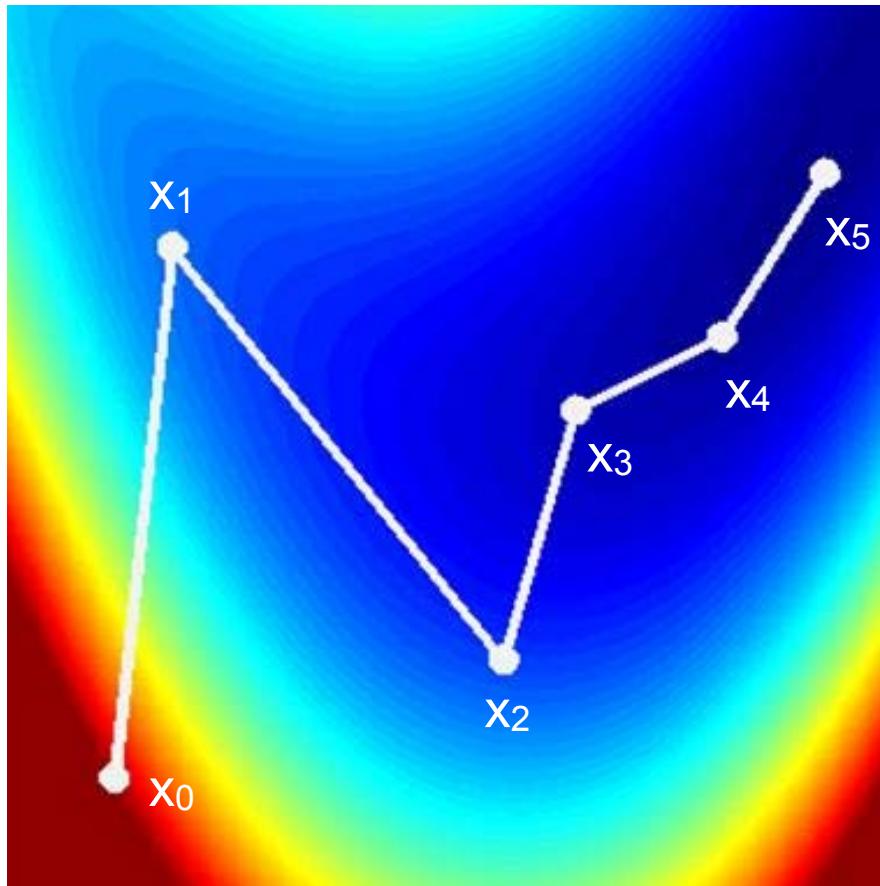
$$p_n^k = \frac{\exp(\mathbf{y}_n[k])}{\sum_j \exp(\mathbf{y}_n[j])}$$

- We want to minimize the cross entropy

$$E(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N E_n(\mathbf{w}) ,$$

$$E_n(\mathbf{w}) = - \sum_{k=1}^K t_n^k \ln(p_n^k) ,$$

# Reminder: Gradient Descent



$$f(\mathbf{x} + d\mathbf{x}) \approx f(\mathbf{x}) + \nabla f \cdot d\mathbf{x}$$

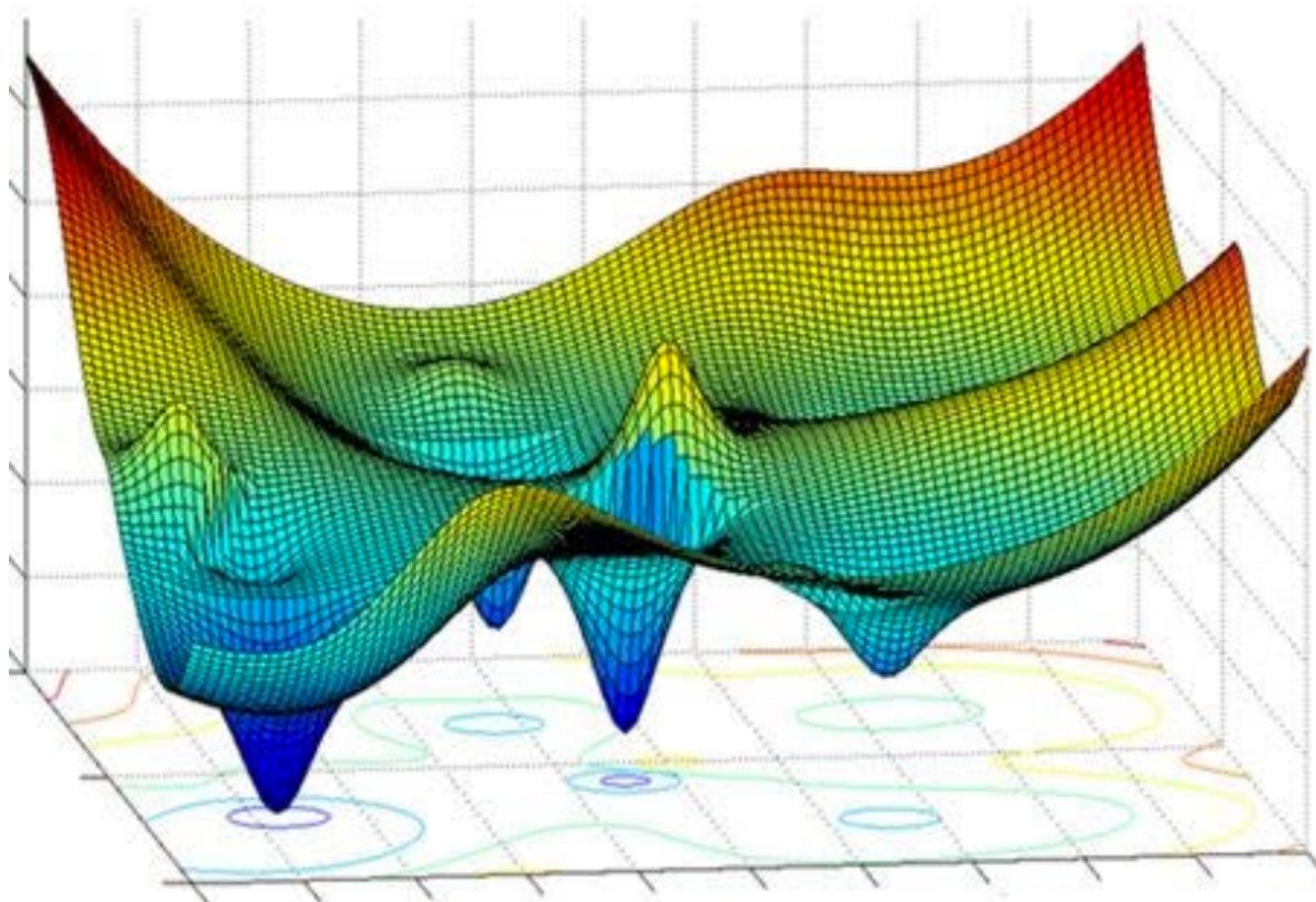
$$d\mathbf{x} = -\eta \nabla f$$

$$\Rightarrow f(\mathbf{x} + d\mathbf{x}) \approx f(\mathbf{x}) - \eta \|\nabla f\|^2 < f(\mathbf{x})$$

$$\boxed{\mathbf{x}^{\tau+1} = \mathbf{x}^\tau - \eta \nabla f(\mathbf{x}^\tau)}$$

—>  $\eta$  is known as the learning rate and must be carefully chosen.

# Reminder: Local Minima



The result depends critically on the starting point and is very likely to be closest local minimum, which is not usually the global one.

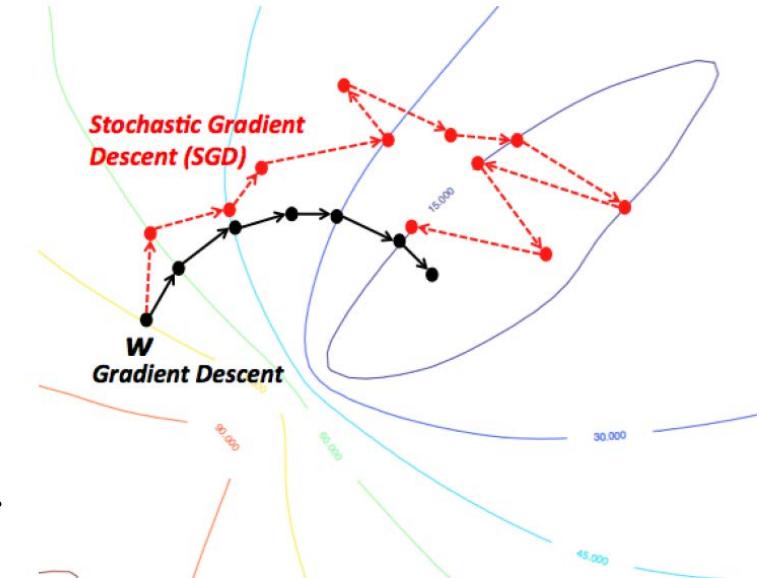
# Stochastic Gradient Descent

$$E(\mathbf{w}) = \sum_{n=1}^N E_n(\mathbf{w})$$

Gradient descent:  $\mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \eta \sum_{n=1}^N \nabla E_n(\mathbf{w}^\tau)$ .

Stochastic descent:  $\mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \eta \sum_{n \in B^\tau} \nabla E_n(\mathbf{w}^\tau)$ ,

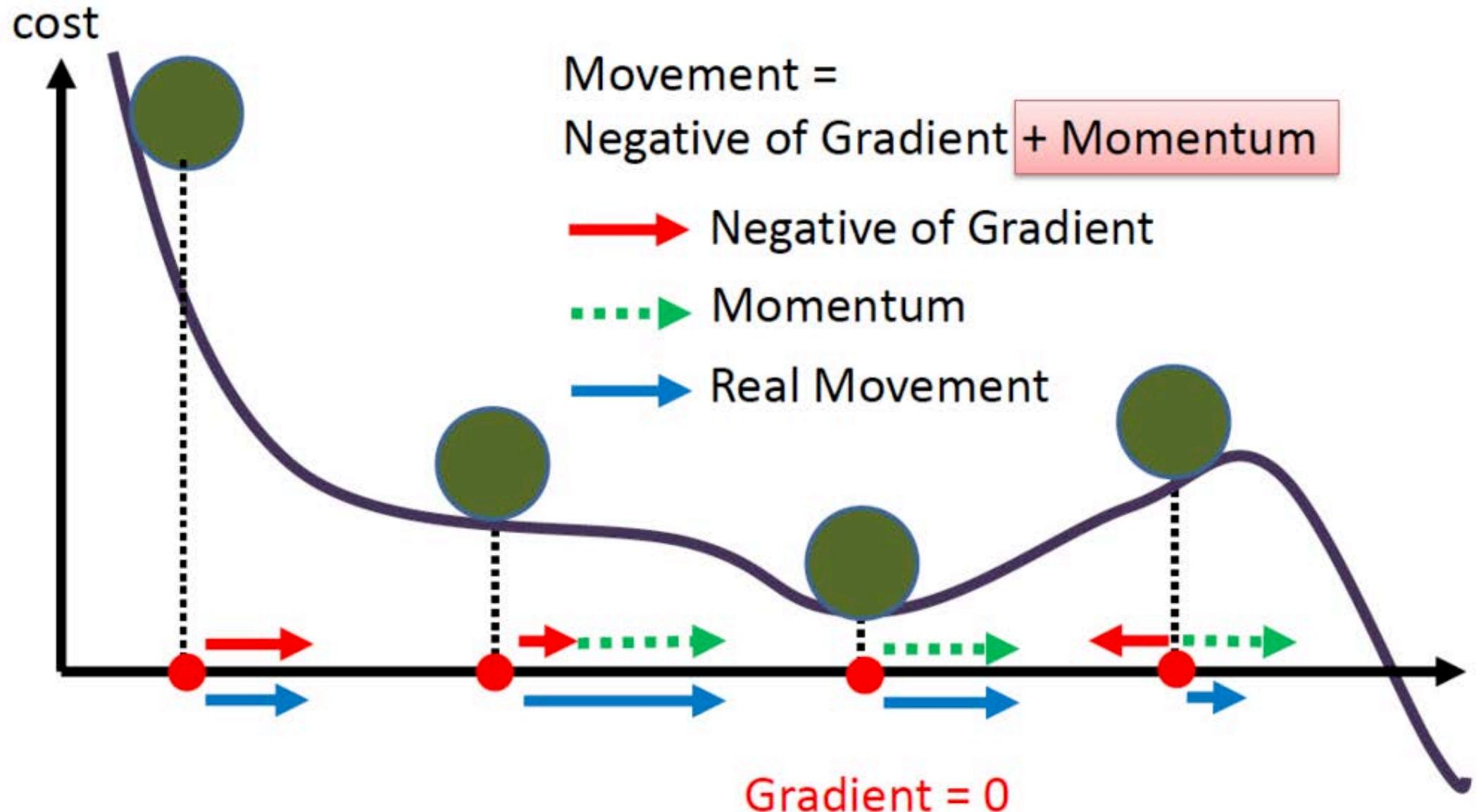
where  $B^\tau$  represents a different randomly chosen set of indices at each iteration, also known as a mini-batch.



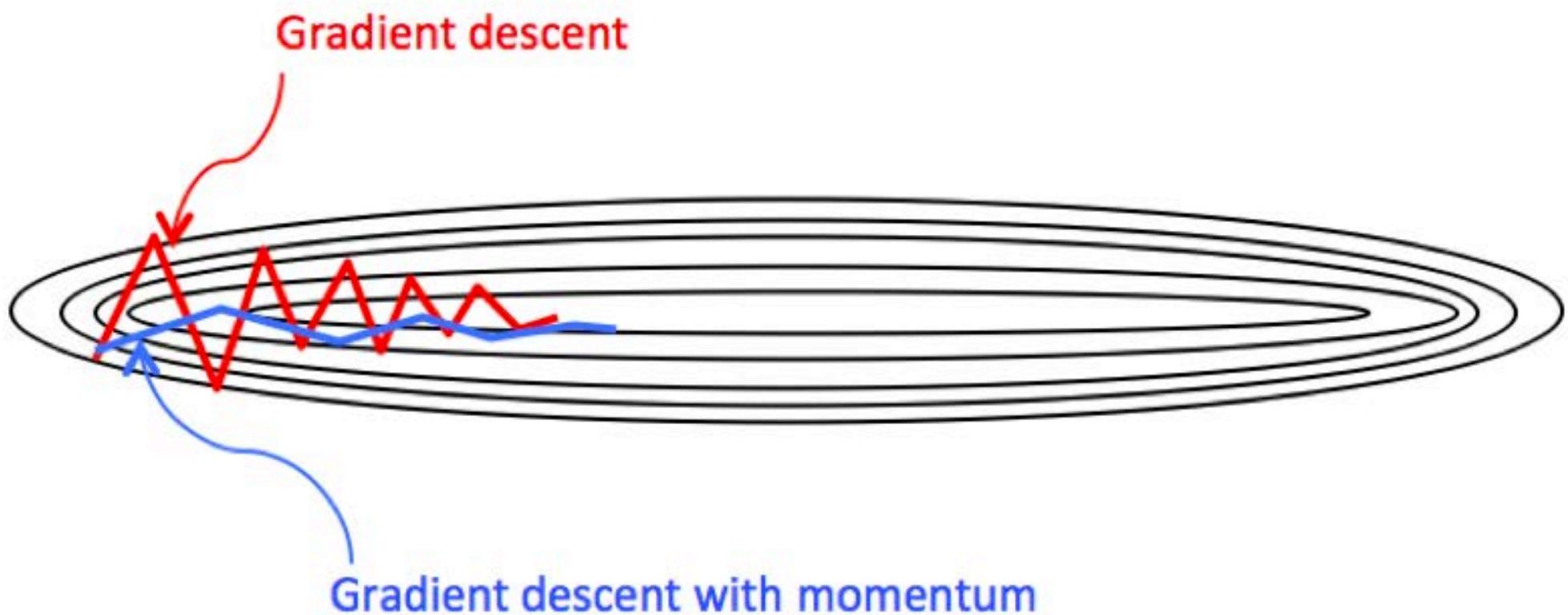
## Randomly choosing batches

- helps reduce the chances of falling into local minima,
- makes the computation possible on GPUs even when dealing with LARGE databases.

# Escaping Local Minima



# Improved Gradient Descent



# Adaptive Moment Estimation

$$\mathbf{g}_{\tau+1} = \sum_{n \in B^\tau} \nabla E_n(\mathbf{w}^\tau)$$

Minibatch gradient.

$$\mathbf{m}_{\tau+1} = \beta_1 \mathbf{m}_\tau + (1 - \beta_1) \mathbf{g}_{\tau+1}$$

Mean gradient.

$$\mathbf{v}_{\tau+1} = \beta_2 \mathbf{v}_\tau + (1 - \beta_2) \mathbf{g}_{\tau+1}^2$$

Mean gradient squared.

$$\hat{\mathbf{m}}_{\tau+1} = \frac{\mathbf{m}_{\tau+1}}{1 - \beta_1^t}$$

Corrective factor.

$$\hat{\mathbf{v}}_{\tau+1} = \frac{\mathbf{v}_{\tau+1}}{1 - \beta_2^t}$$

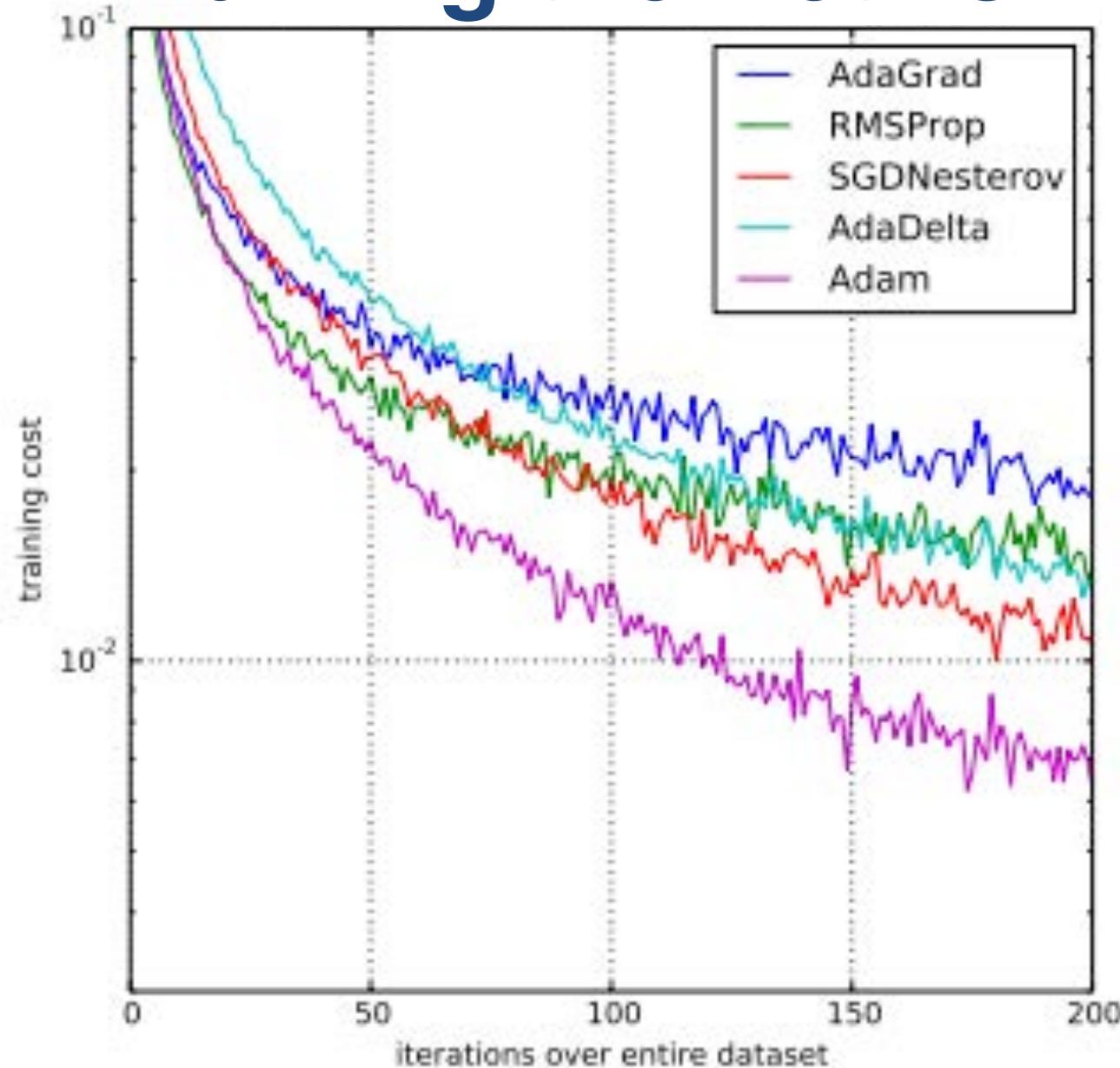
Corrective factor.

$$\mathbf{w}_{\tau+1} = \mathbf{w}_\tau - \alpha \frac{\hat{\mathbf{m}}_{\tau+1}}{\sqrt{\hat{\mathbf{v}}_{\tau+1} + \epsilon}}$$

Gradient step.

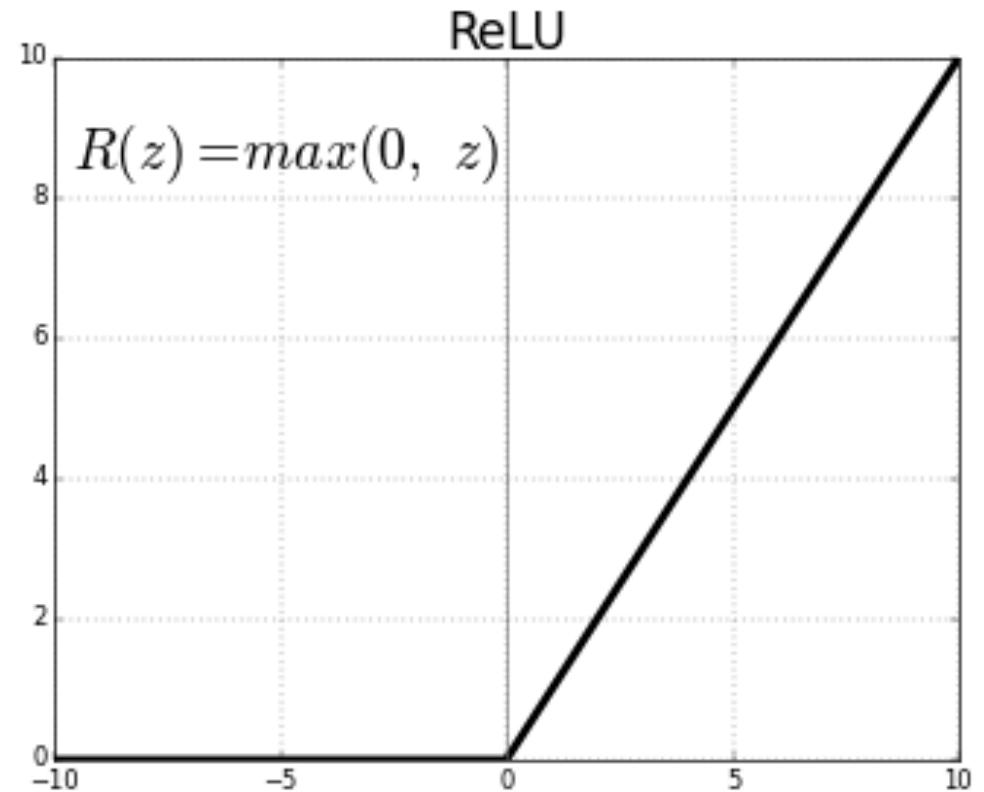
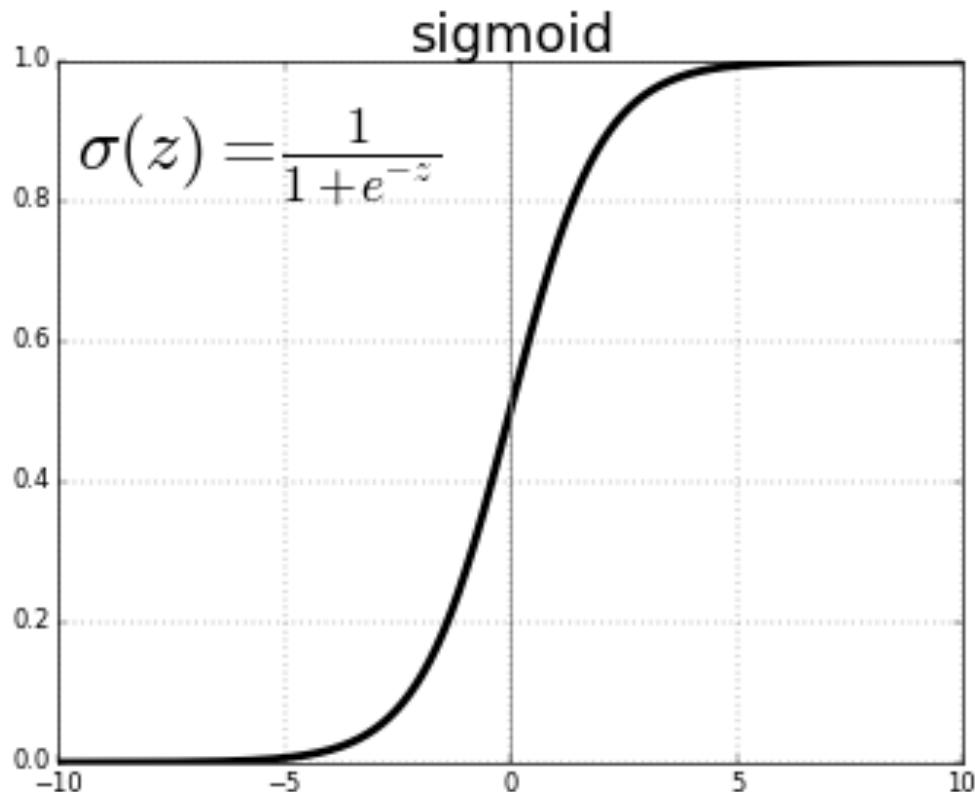
$$\begin{aligned}\mathbf{m}_0 &= \mathbf{v}_0 = 0, \\ \beta_1 &= 0.9, \\ \beta_2 &= 0.999, \\ \alpha &= 0.001, \\ \epsilon &= 10^{-8}\end{aligned}$$

# Training the Network



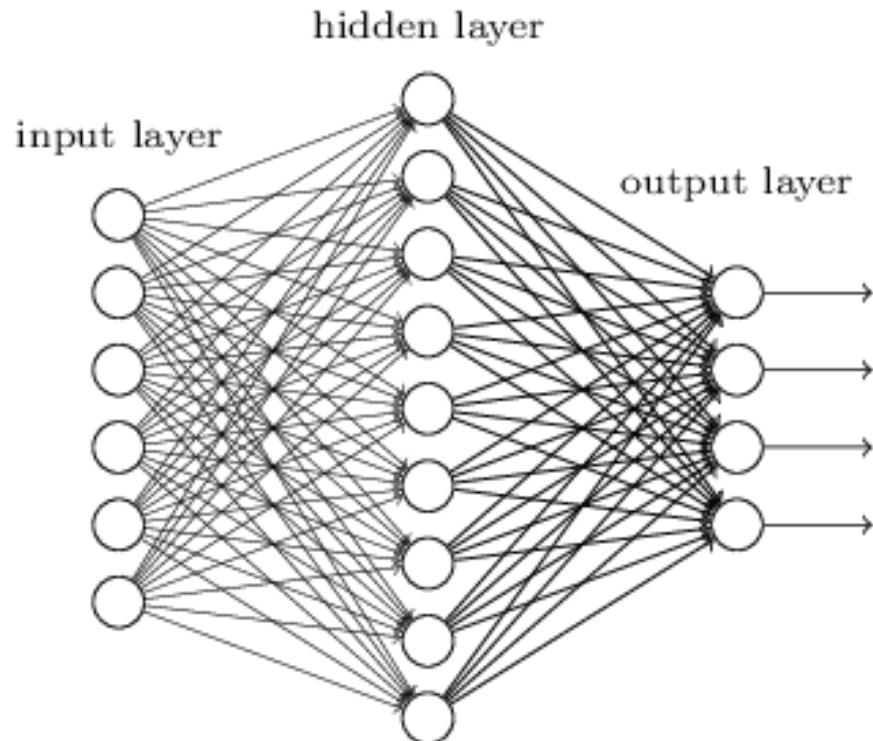
- The loss decreases over time but not monotonically.
- It can take a very long time to converge.

# Sigmoid vs ReLu



- One problem with the sigmoid function is that when the argument is not close to zero the gradients vanish.
- Empirically, replacing the Sigmoid by ReLu has significantly boosted performance in many cases.

# MLP with ReLU

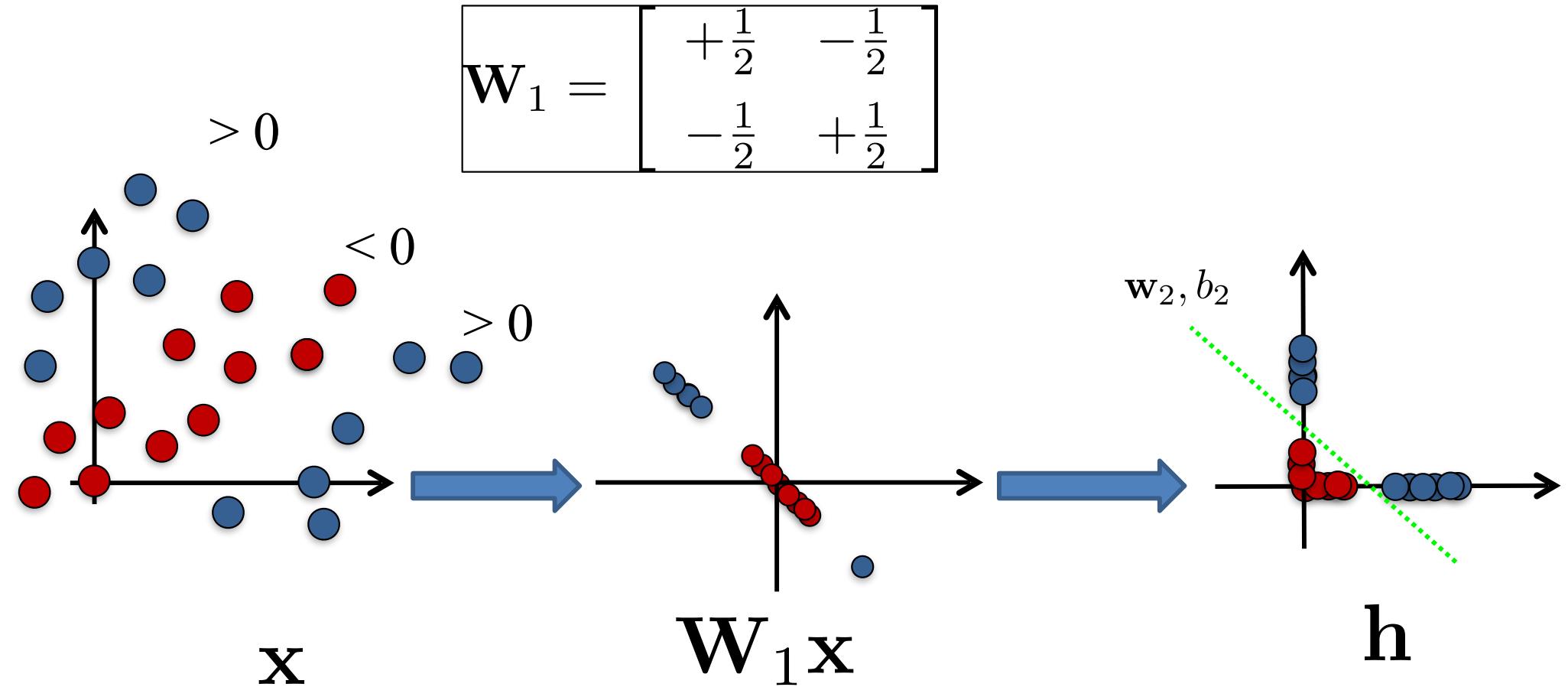


$$\begin{aligned}\mathbf{h} &= \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) \\ \mathbf{y} &= \sigma(\mathbf{W}_2 \mathbf{h} + \mathbf{b}_2)\end{aligned}$$

$$\sigma(\mathbf{x}) = \max(0, \mathbf{x}).$$

- Each node defines a hyperplane.
- The resulting function is piecewise linear affine and continuous.

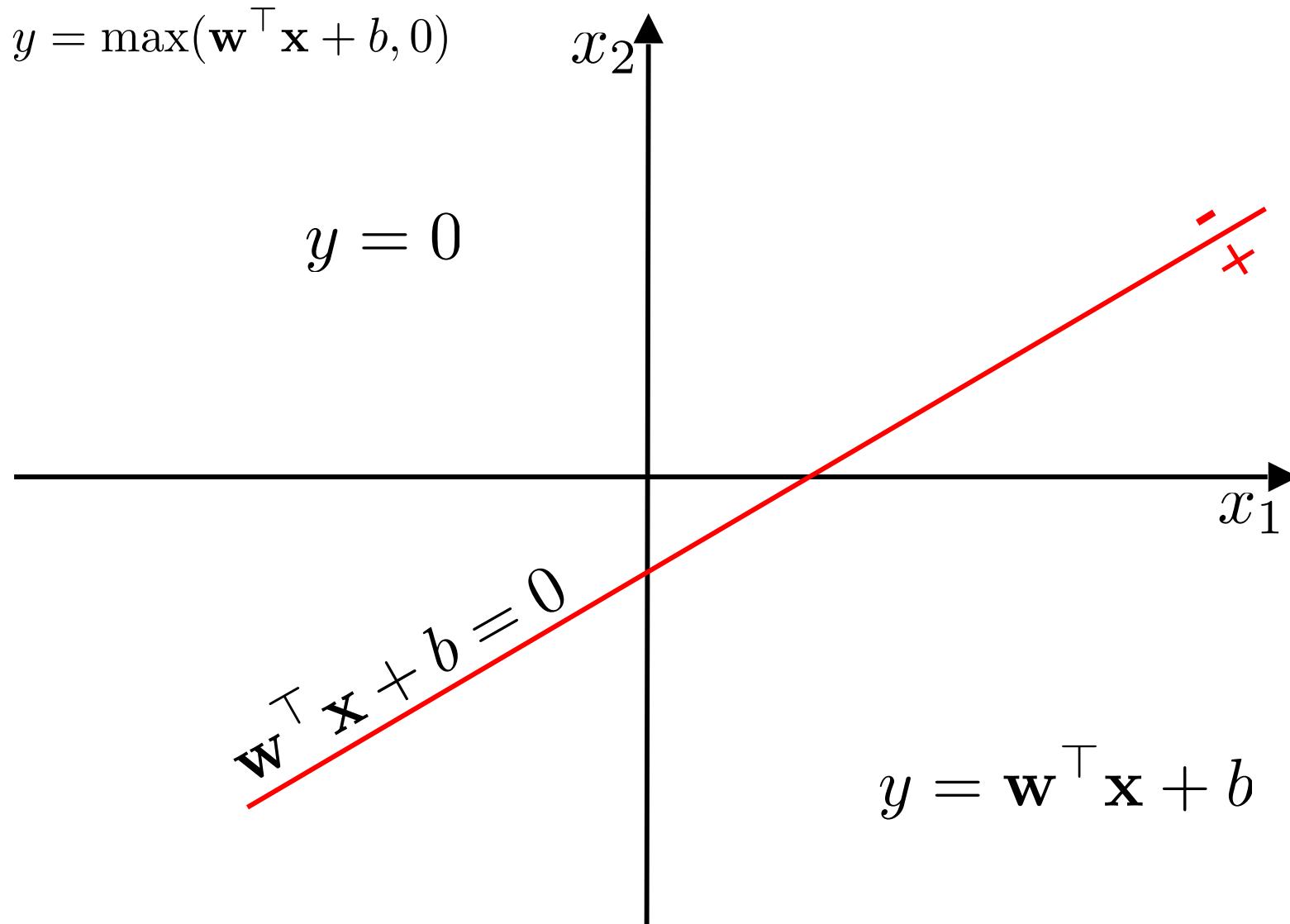
# ReLU Behavior



$$\mathbf{h} = \text{ReLU}(\mathbf{W}_1 \mathbf{x})$$

$$\mathbf{y} = \mathbf{w}_2^T \mathbf{h} + b_2$$

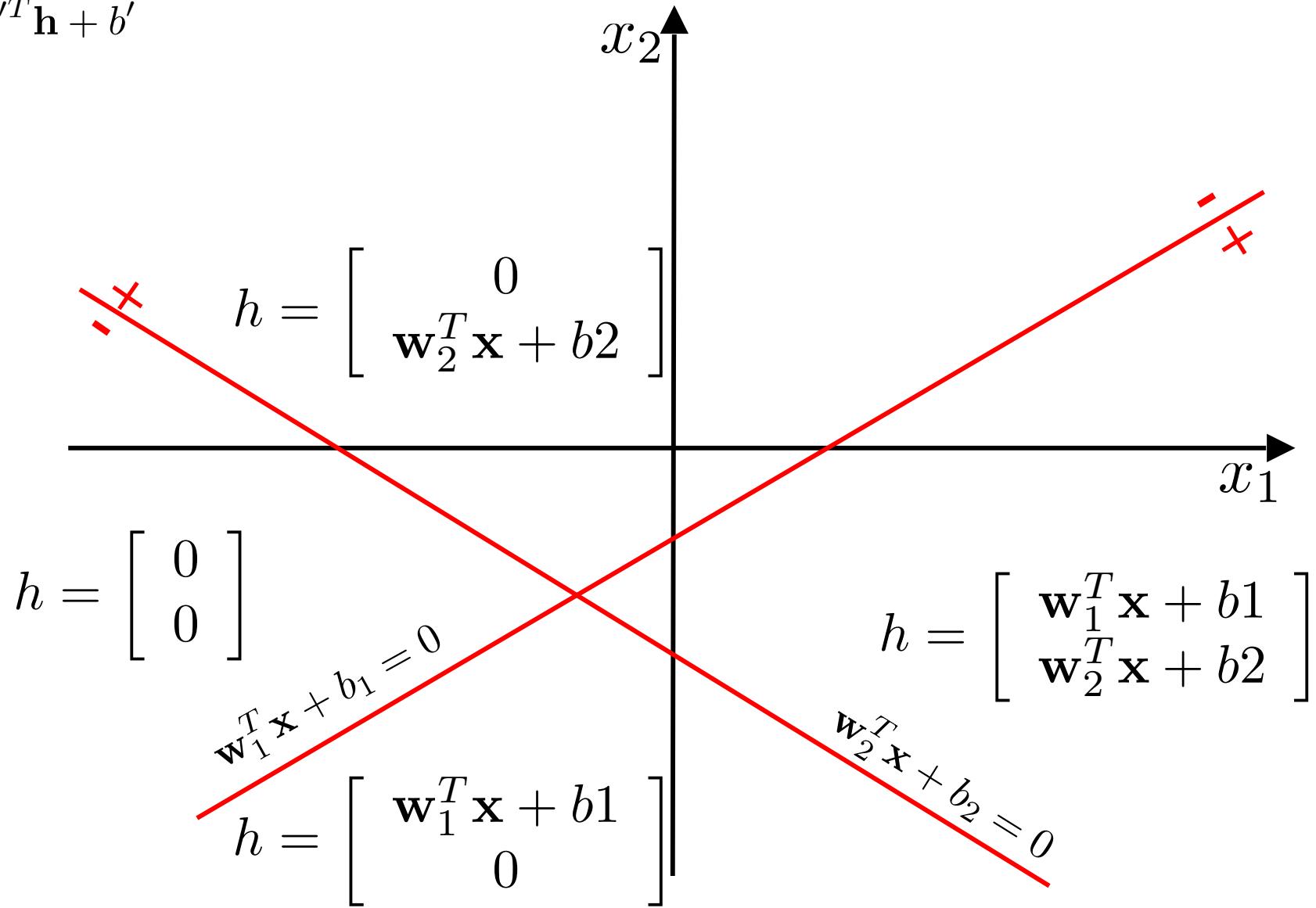
# One Single Hyperplane



# Two Hyperplanes

$\mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0)$  with  $\mathbf{W} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \end{bmatrix}$  and  $\mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$

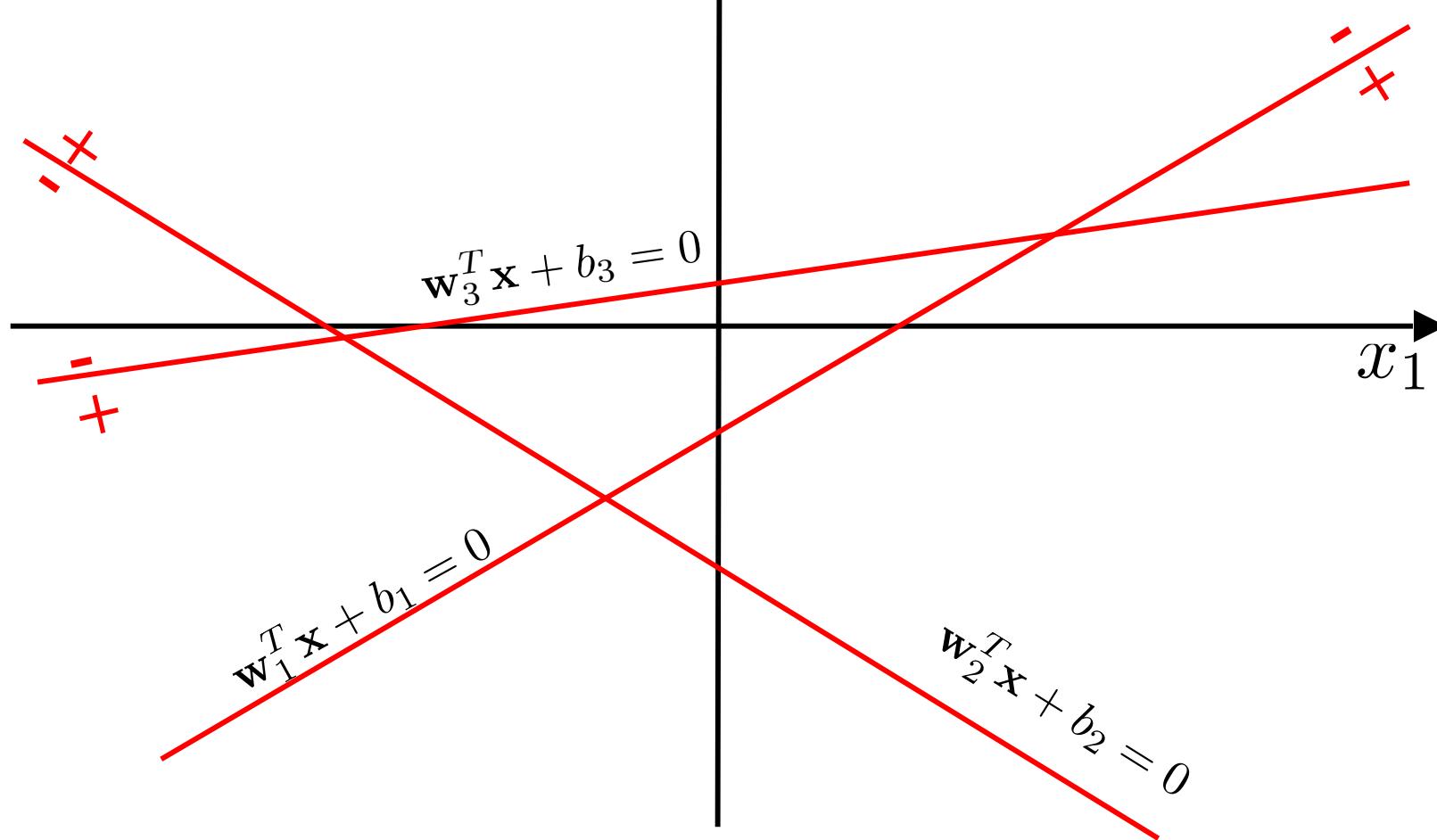
$$y = \mathbf{w}'^T \mathbf{h} + b'$$



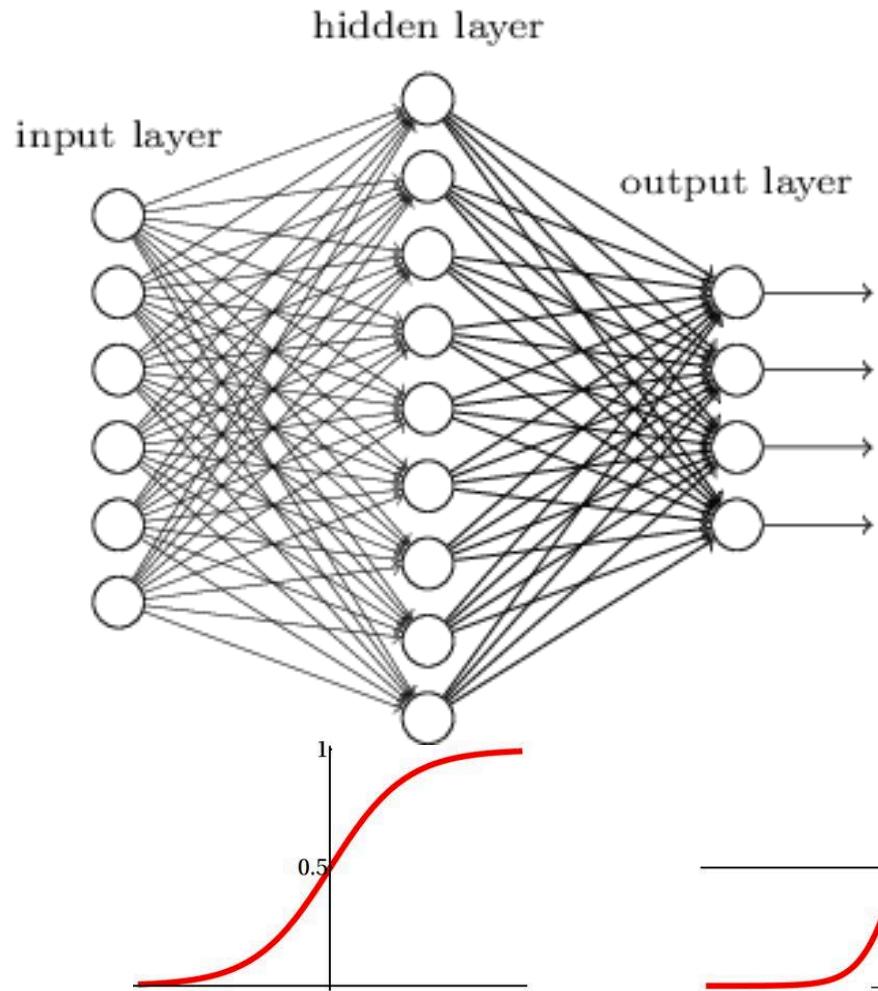
# Three Hyperplanes

$$\begin{cases} \mathbf{h} = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \\ y = \mathbf{w}'^\top \mathbf{h} \end{cases}$$

with  $\dim(\mathbf{h}) = 3$



# MLP with Sigmoid or Tanh



$$\mathbf{h} = \sigma(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1)$$
$$\mathbf{y} = \sigma(\mathbf{W}_2 \mathbf{h} + \mathbf{b}_2)$$

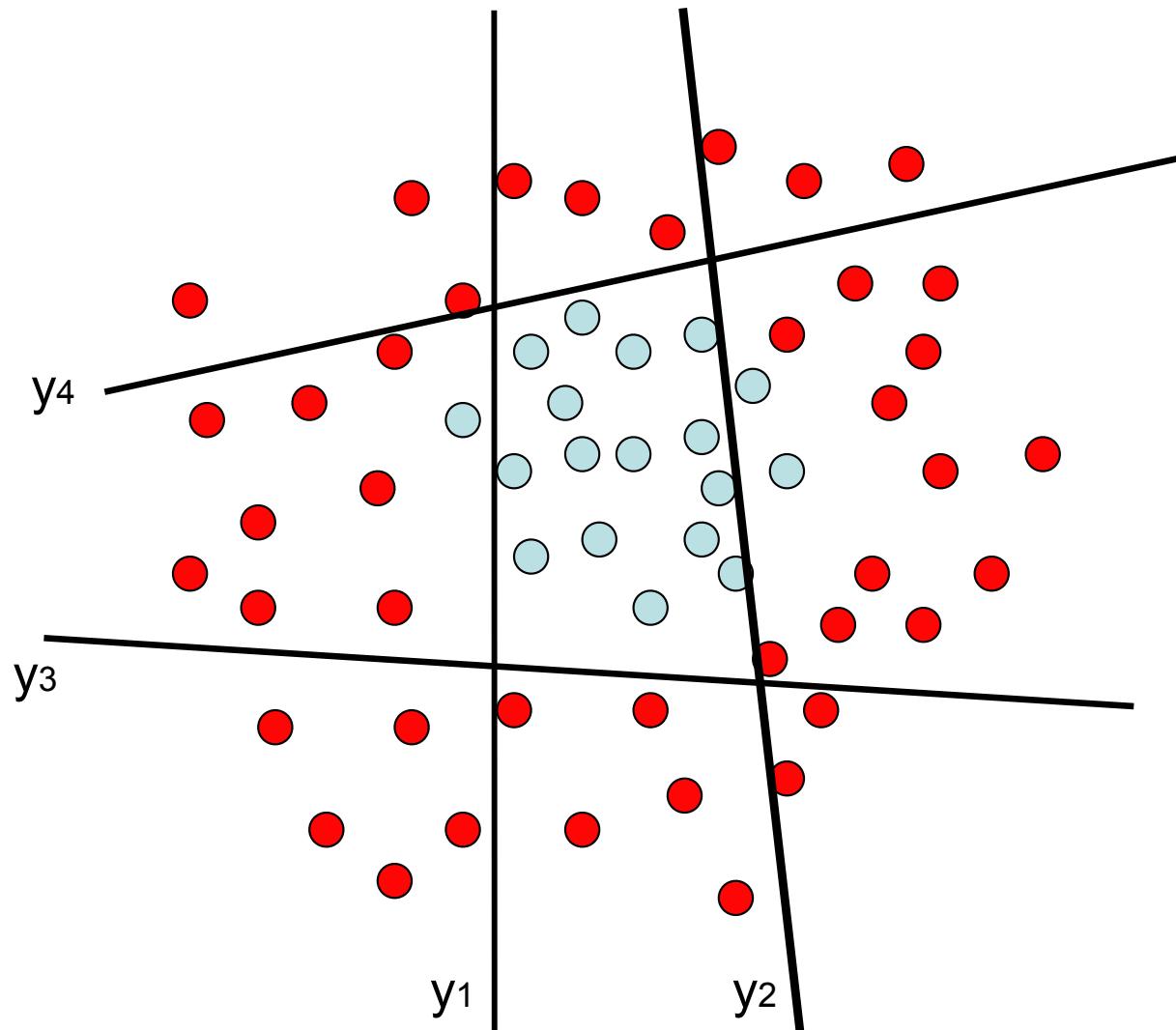
sigm:  $\sigma(x) = \frac{1}{1 + \exp(-x)}$

tanh:  $\sigma(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$

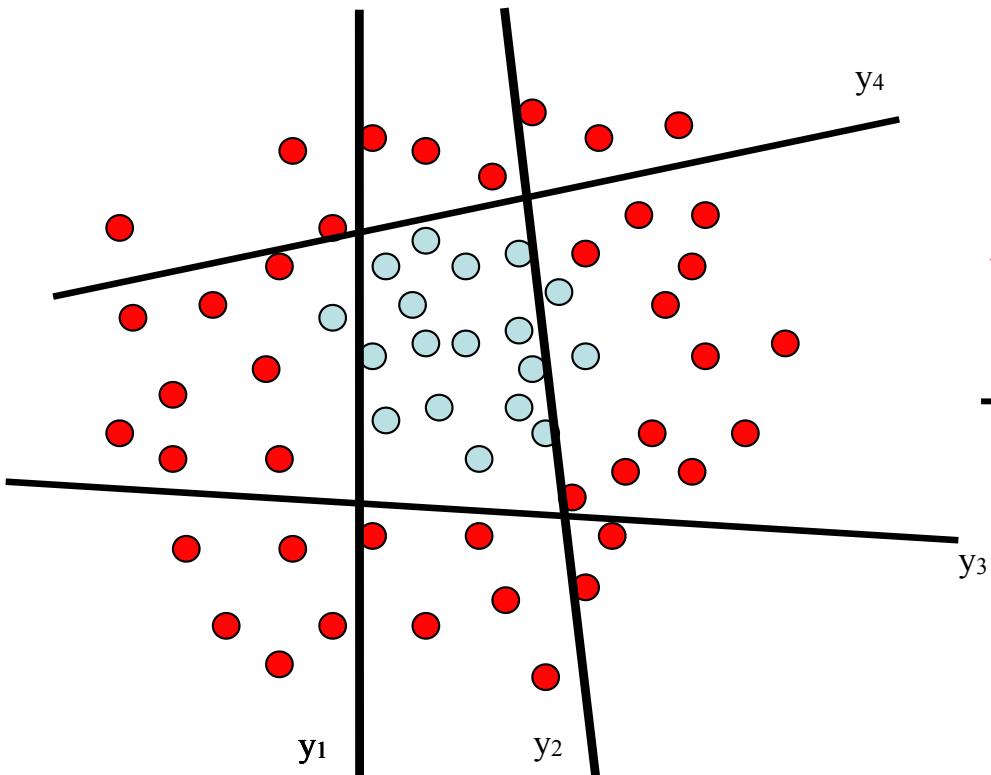
- Each node defines a hyperplane.
- The resulting function is continuously differentiable.

# Reminder: AdaBoost

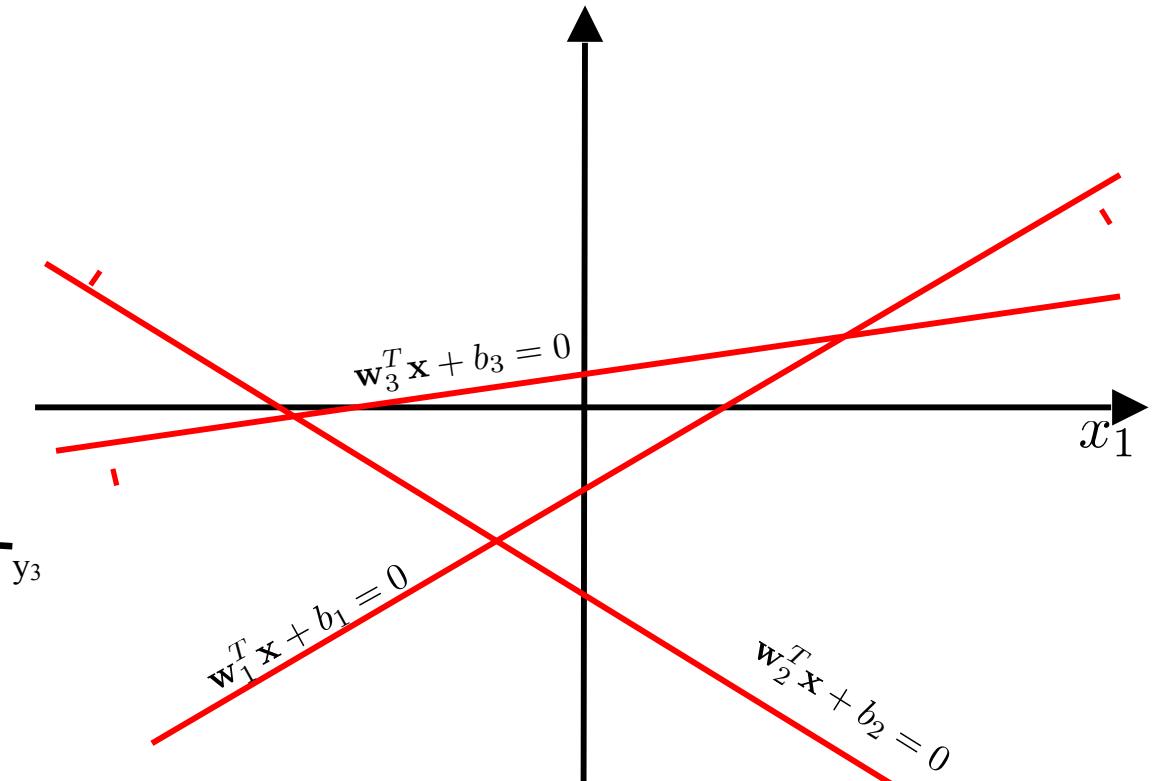
$$y(\mathbf{x}) = \alpha_1 y_1(\mathbf{x}) + \alpha_2 y_2(\mathbf{x}) + \alpha_3 y_3(\mathbf{x}) + \alpha_4 y_4(\mathbf{x})$$



# AdaBoost vs MLP



AdaBoost

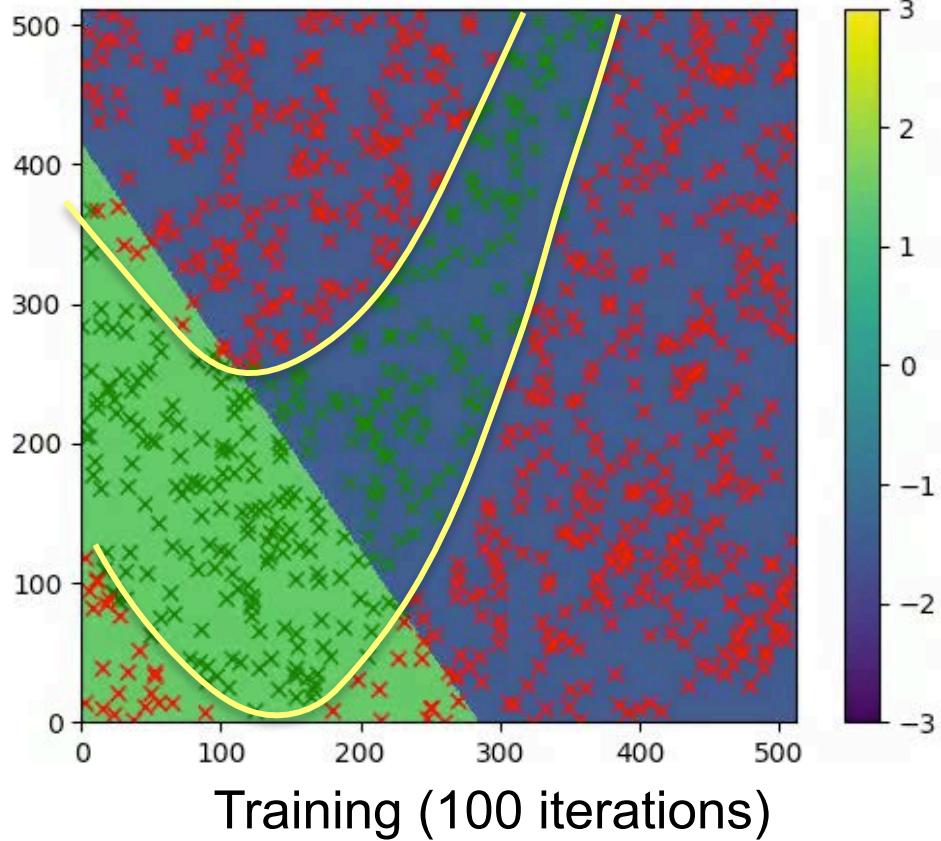


MLP

Both methods find a set of hyperplanes:

- One at a time for a AdaBoost.
- All together for MLPs.

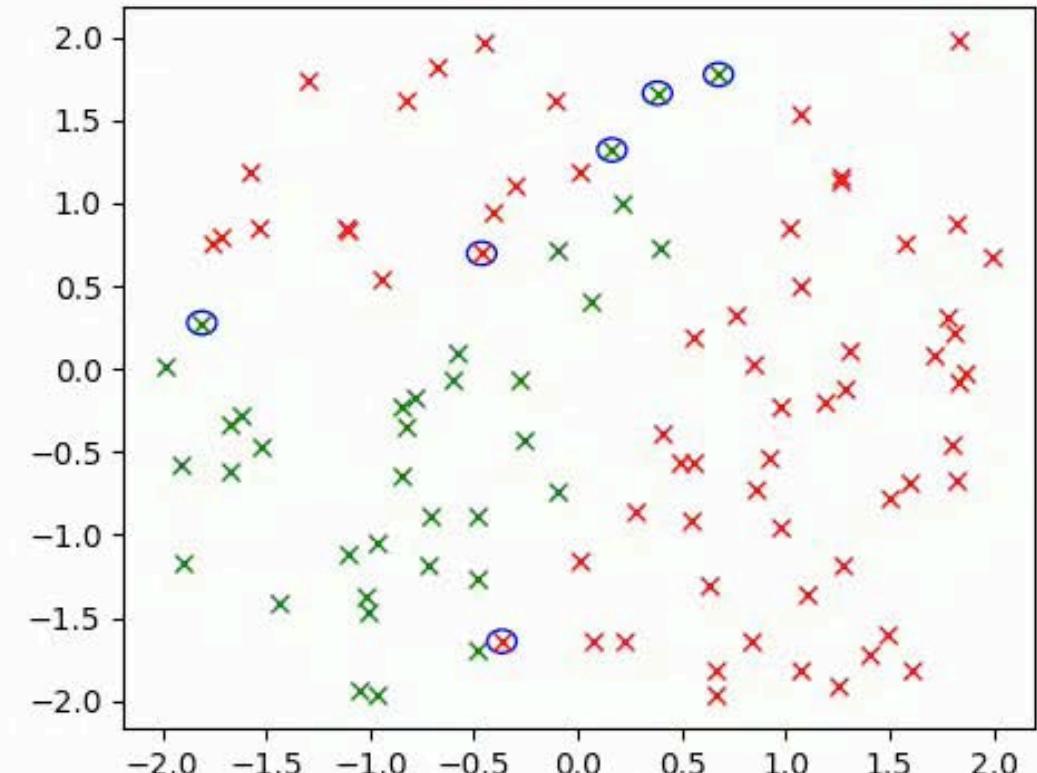
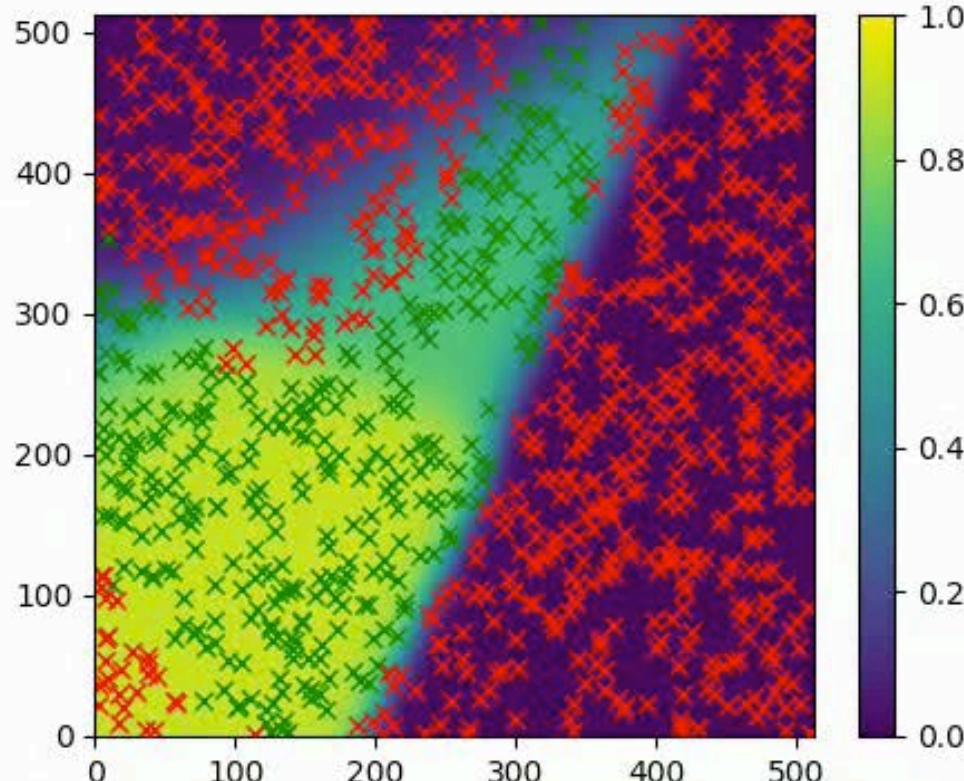
# Rosenbrock using Adaboost



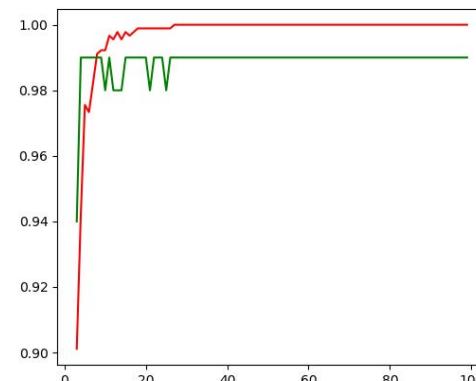
$$r(x, y) = 100 * (y - x^2)^2 + (1 - x)^2$$
$$f(x, y) = \begin{cases} -1 & \text{if } r(x, y) < T \\ 1 & \text{otherwise} \end{cases}$$

- Adaboost adds one linear classifier at a time.
- MLP works with a fixed number of classifiers and optimizes them all at the same time.

# Rosenbrock using a MLP



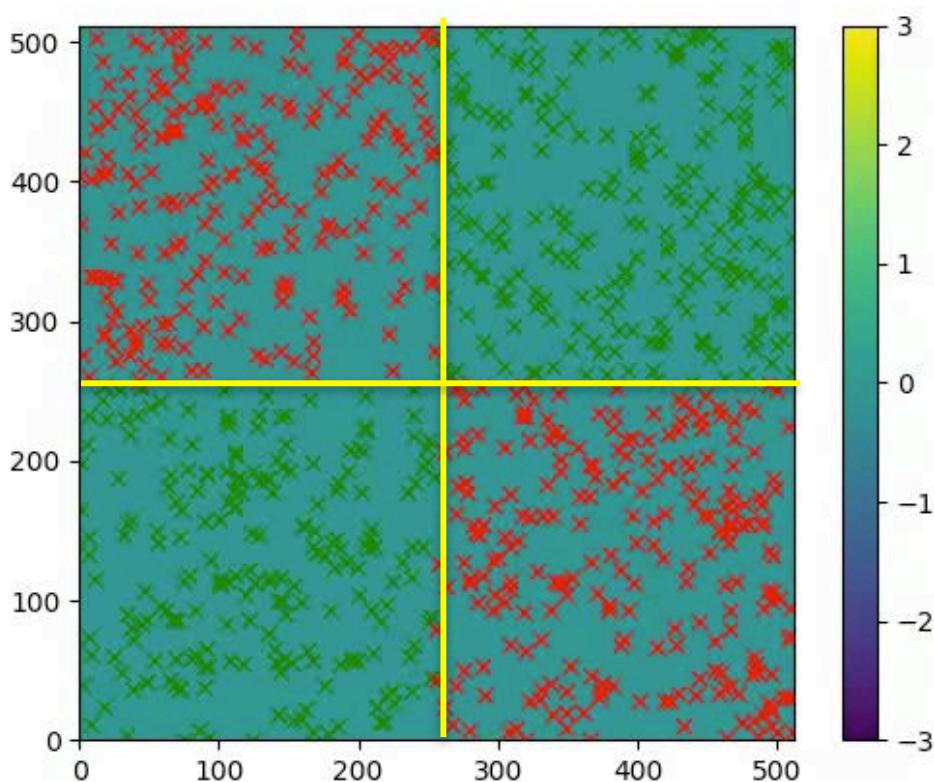
One hidden layer:  $3 < n < 100$



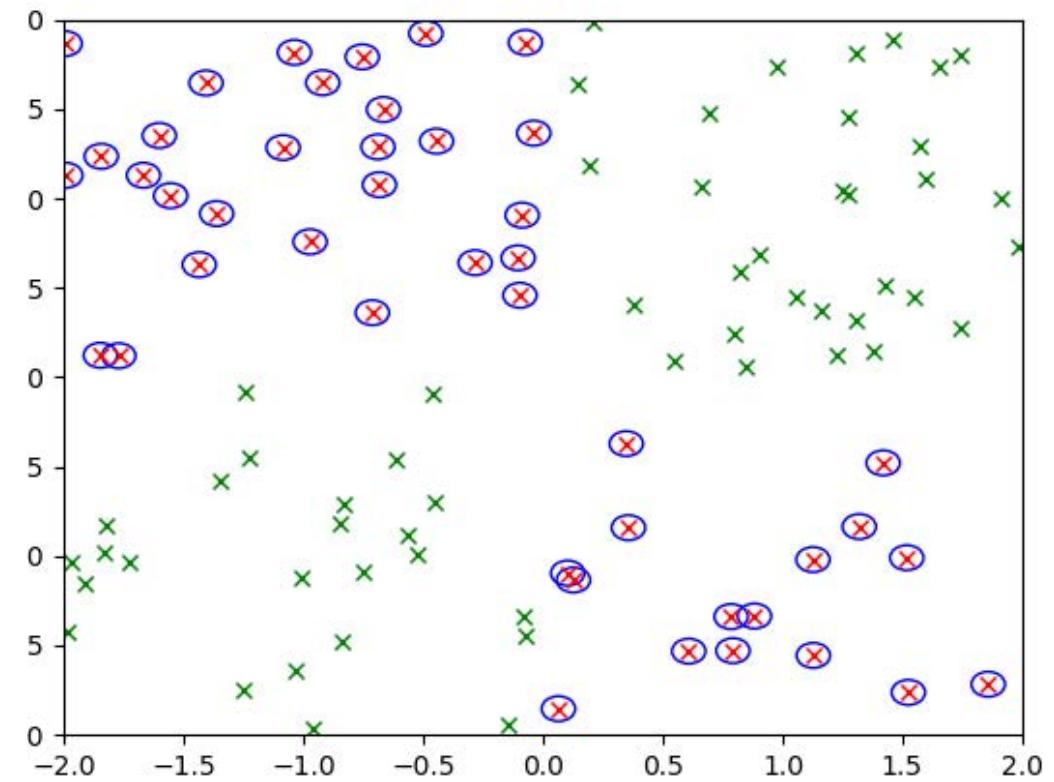
Accuracy as a function of  $n$ :

- Adaboost
- MLP

# Checker Board using AdaBoost



Training (100 iterations)

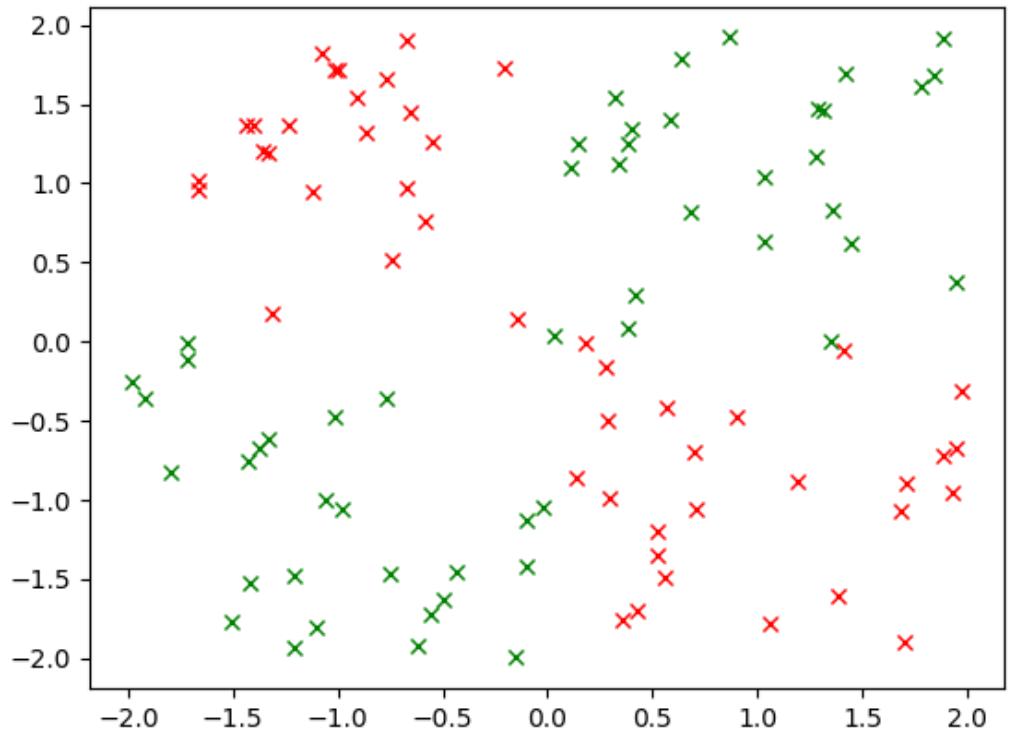
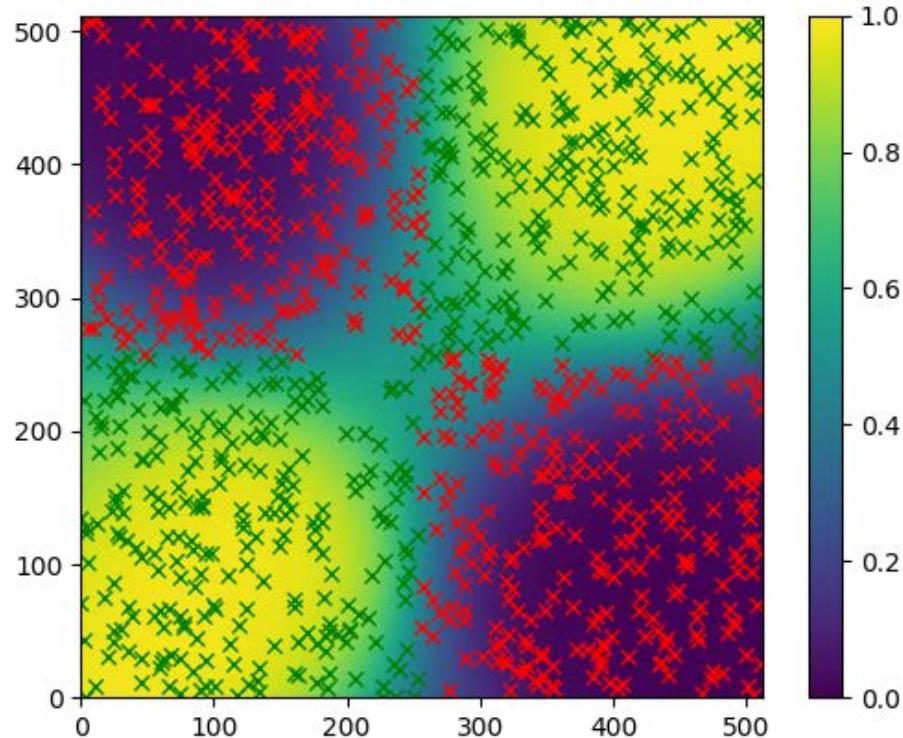


Validation (56% accuracy)

Individual weak classifiers cannot do better than chance.

→ AdaBoost with linear weak classifiers fails in this example.

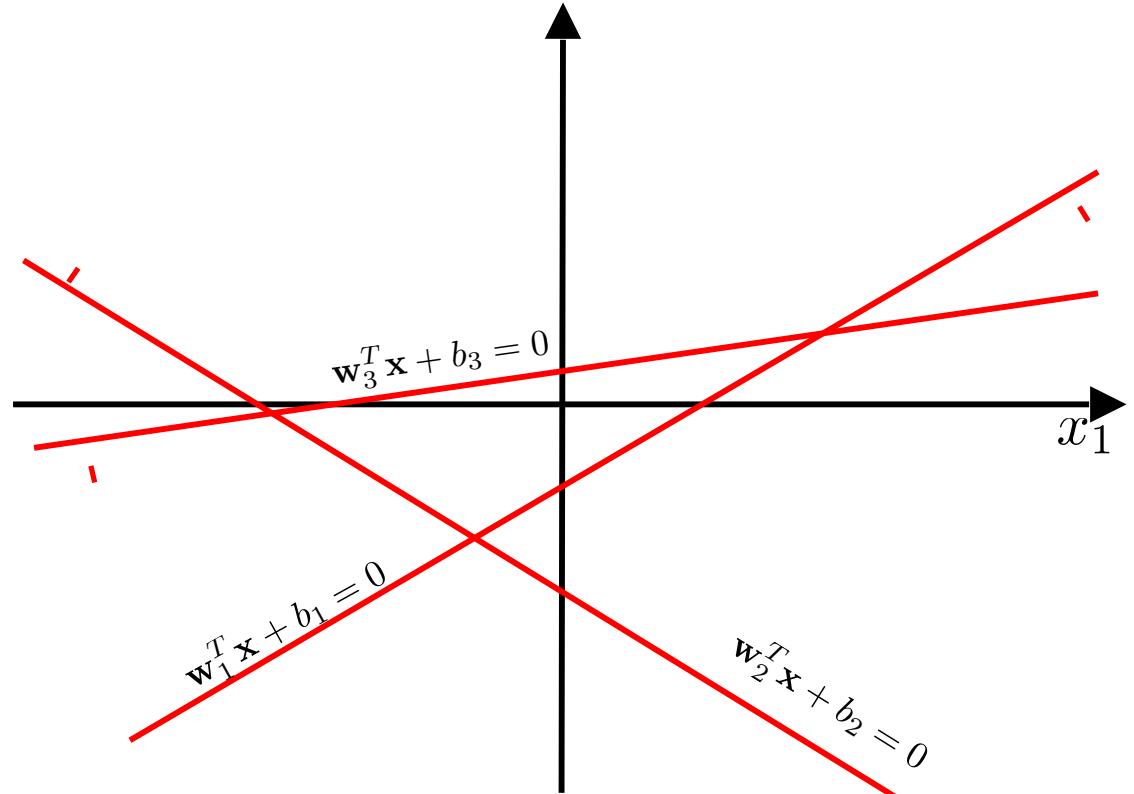
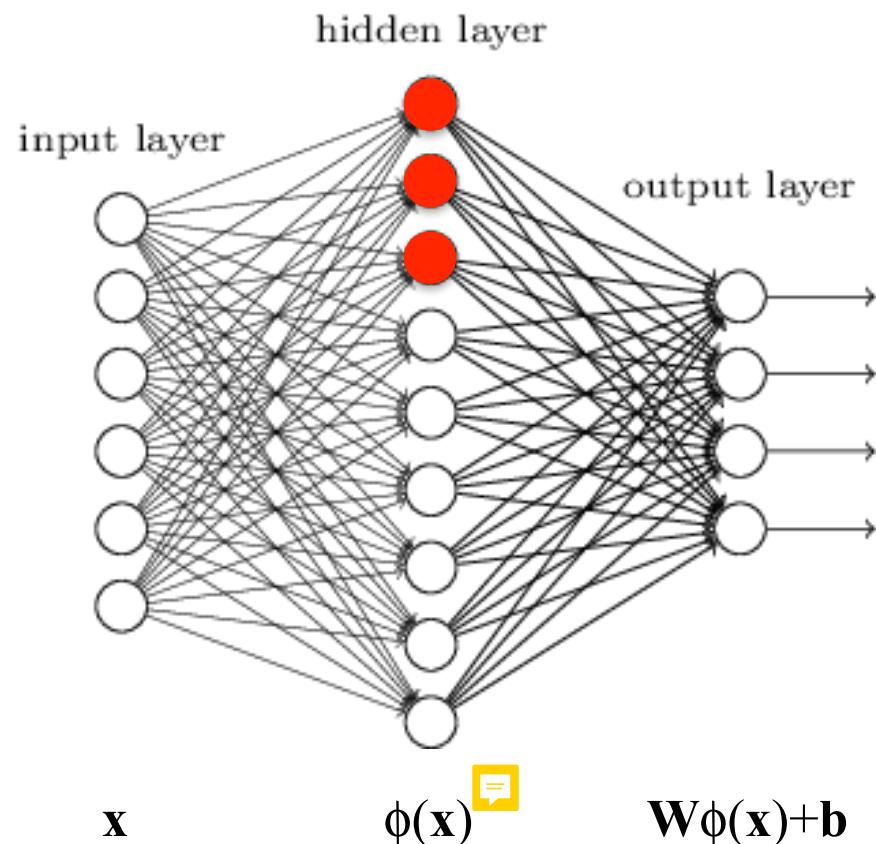
# Checkerboard using a MLP



One hidden layer: n=10

MLPs solve the problem by using several hyperplanes at the same time.  
→ They succeed in this example.

# SVM vs MLP

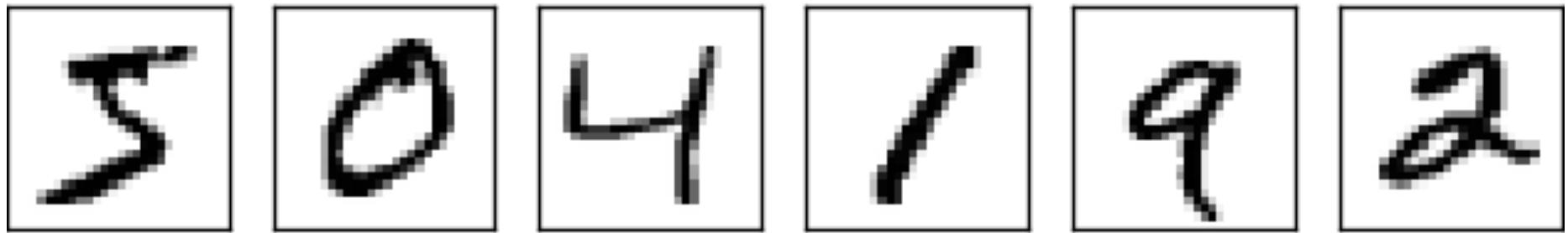


Both methods

- create a high-dimensional feature vector,
- define a classifier on that feature vector.

But the form of  $\phi$  is not defined a priori in a MLP.

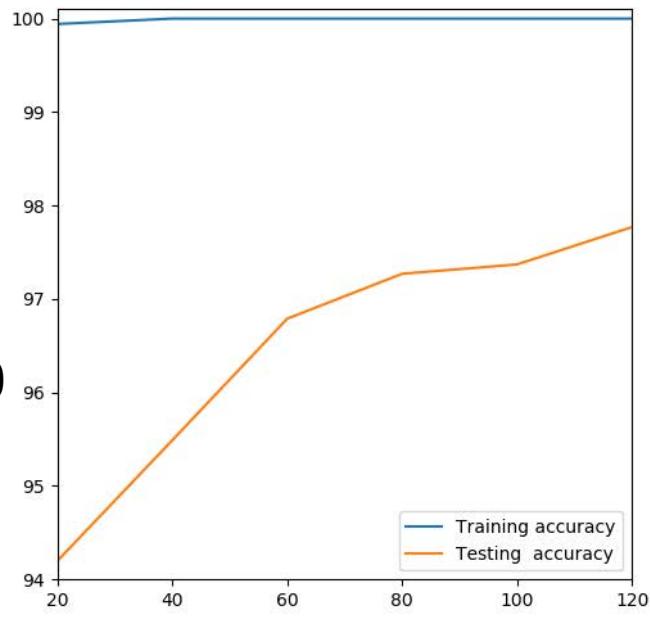
# Reminder: MNIST



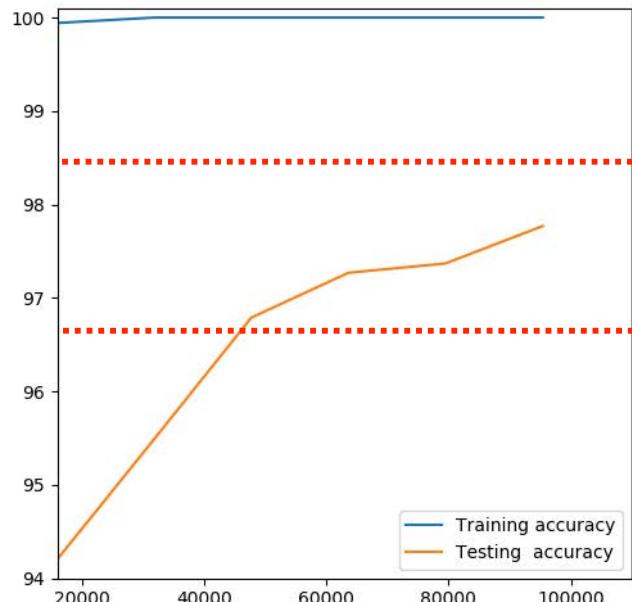
- The network takes as input 28x28 images represented as 784D vectors.
- The output is a 10D vector giving the probability of the image representing any of the 10 digits.
- There are 50'000 training pairs of images and the corresponding label, 10'000 validation pairs, and 5'000 testing pairs.

# MNIST Results

nIn = 784  
nOut = 10  
 $20 < n1 < 120$



- Deep nets have **many** parameters.
- This has long been a major problem.  
—> Was eventually solved by using GPUs.



SVM: 98.6

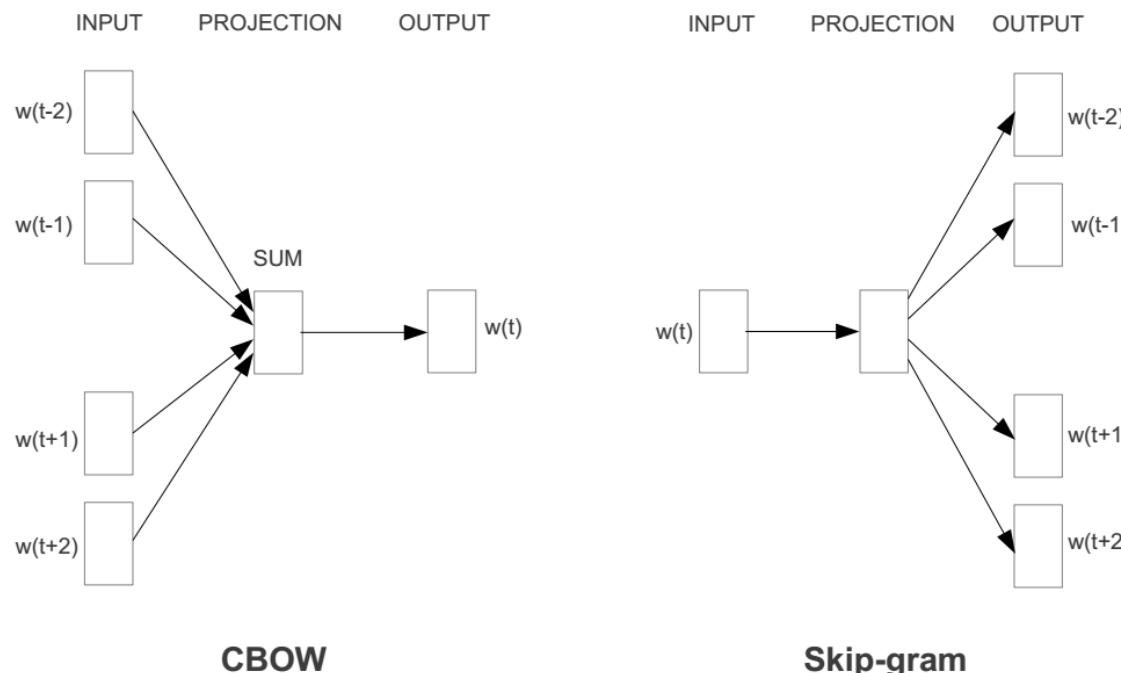
Knn: 96.8

- Around 2005, SVMs were often felt to be superior to neural nets.
- This is no longer the case ....

# Optional: Converting Words to Vectors

- How similar is **pizza** to **pasta**?
- How related is **pizza** to **Italy**?
- Representing words as vectors allows for easy computation of similarity.
- Makes it possible to use the Machine Learning techniques we have discussed.
- Exploit the theory that similar words tend to occur in similar context.

# Optional: Words in Context (*word2vec*)

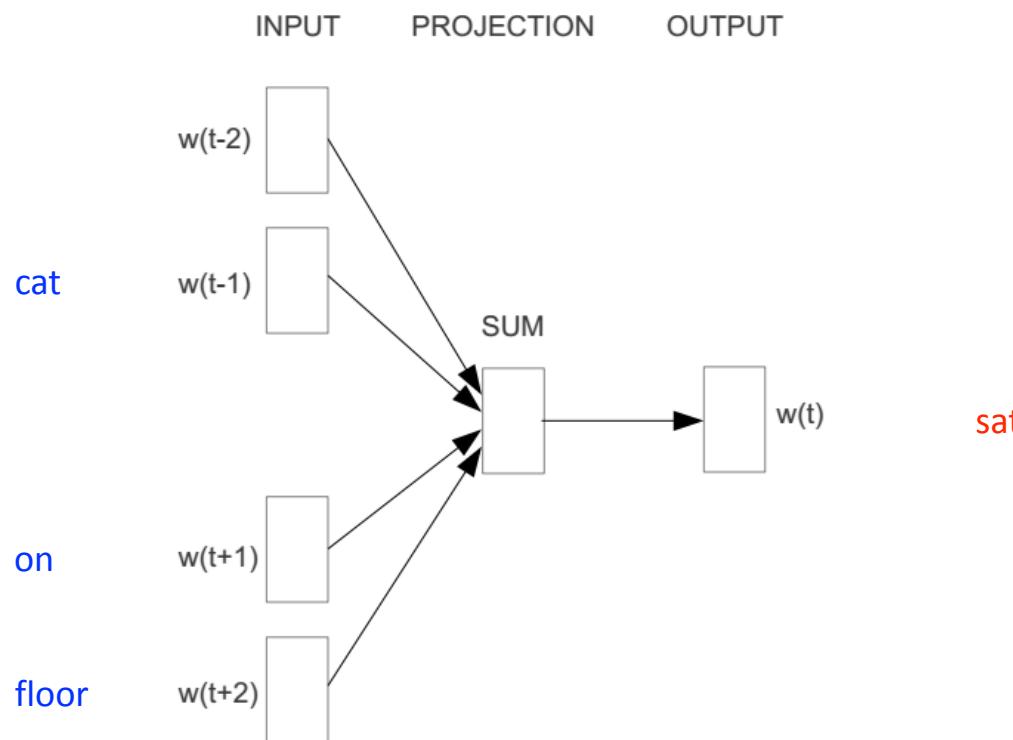


Two basic neural network models:

- Continuous Bag of Word (CBOW). Use a window of words to predict the middle one.
- Skip-gram (SG). Use a word to predict the surrounding ones in a window.

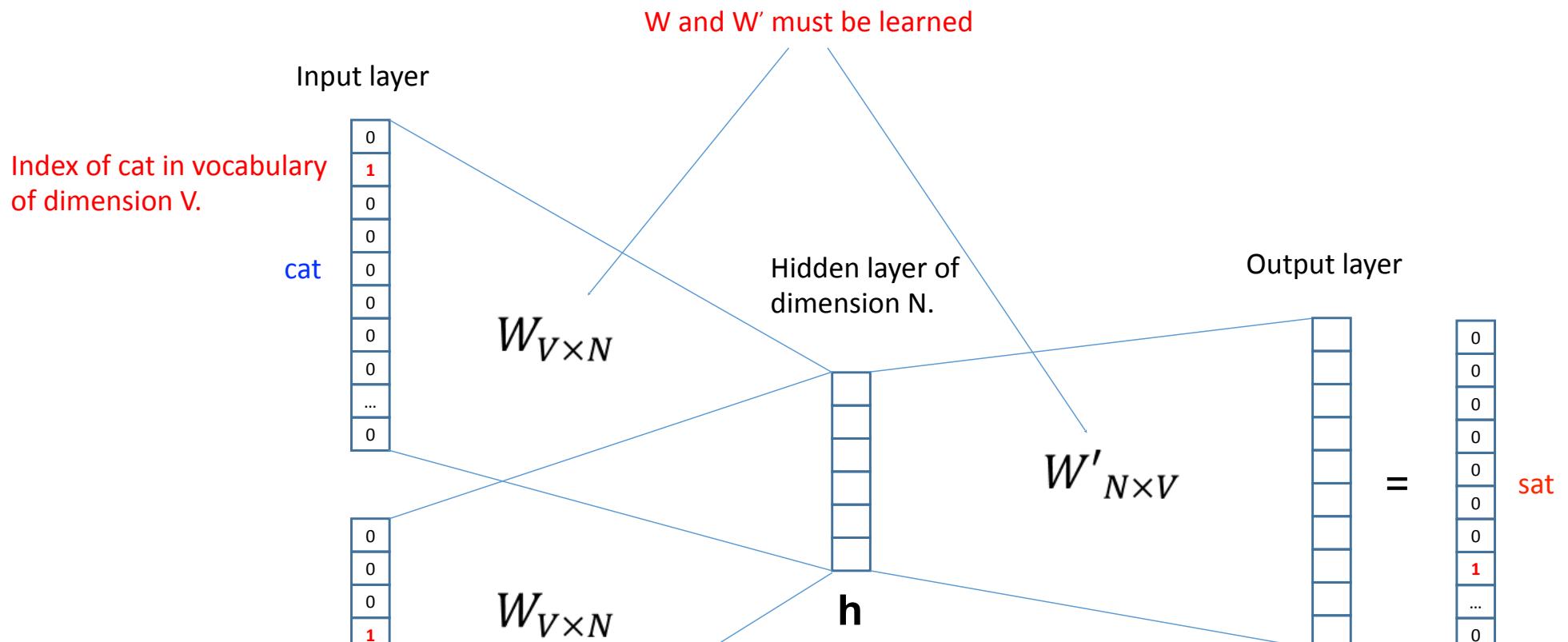
# Optional: Continuous Bag of Words

The cat sat on the floor.



**sat** should be predicted from the words around it.

# Optional: Window of Size 2



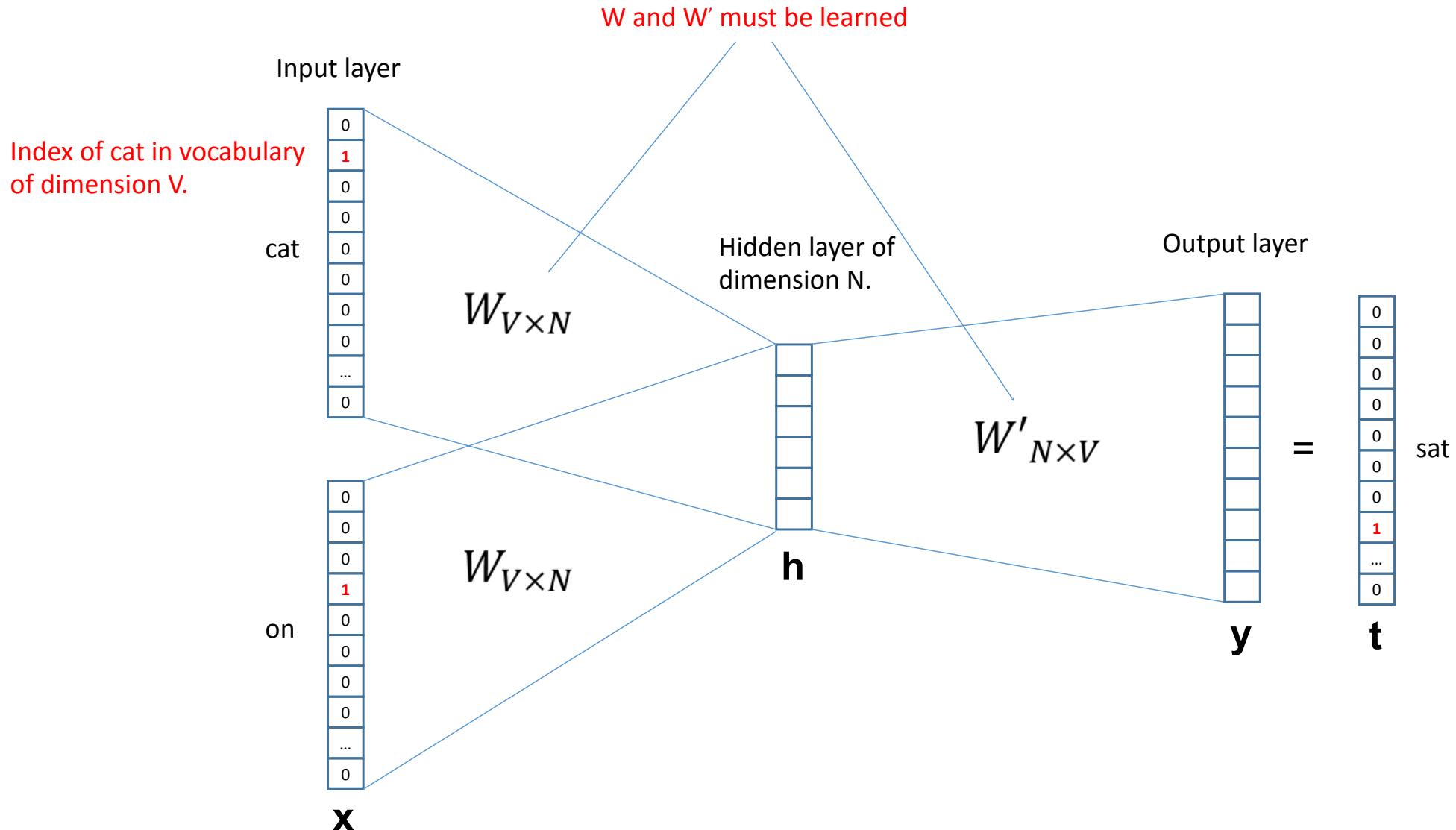
$$\mathbf{v}_{cat} = \mathbf{W}\mathbf{x}_{cat}$$

$$\mathbf{v}_{on} = \mathbf{W}\mathbf{x}_{on}$$

$$\mathbf{h} = \frac{\mathbf{v}_{cat} + \mathbf{v}_{on}}{2}$$

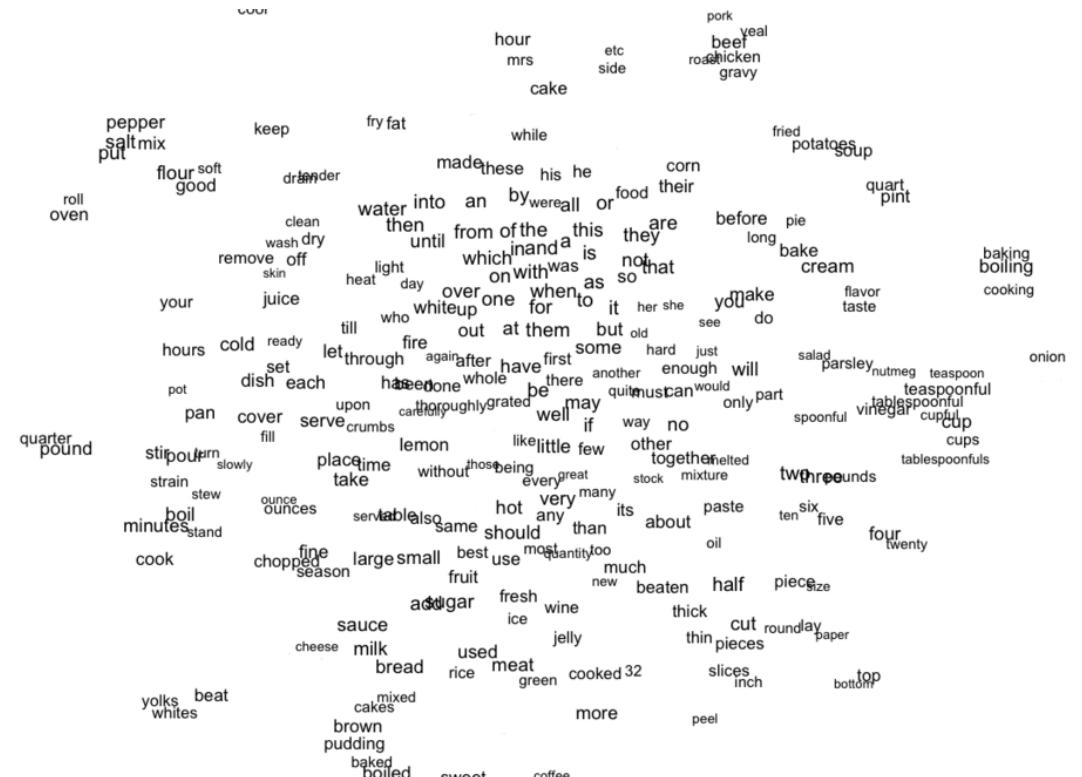
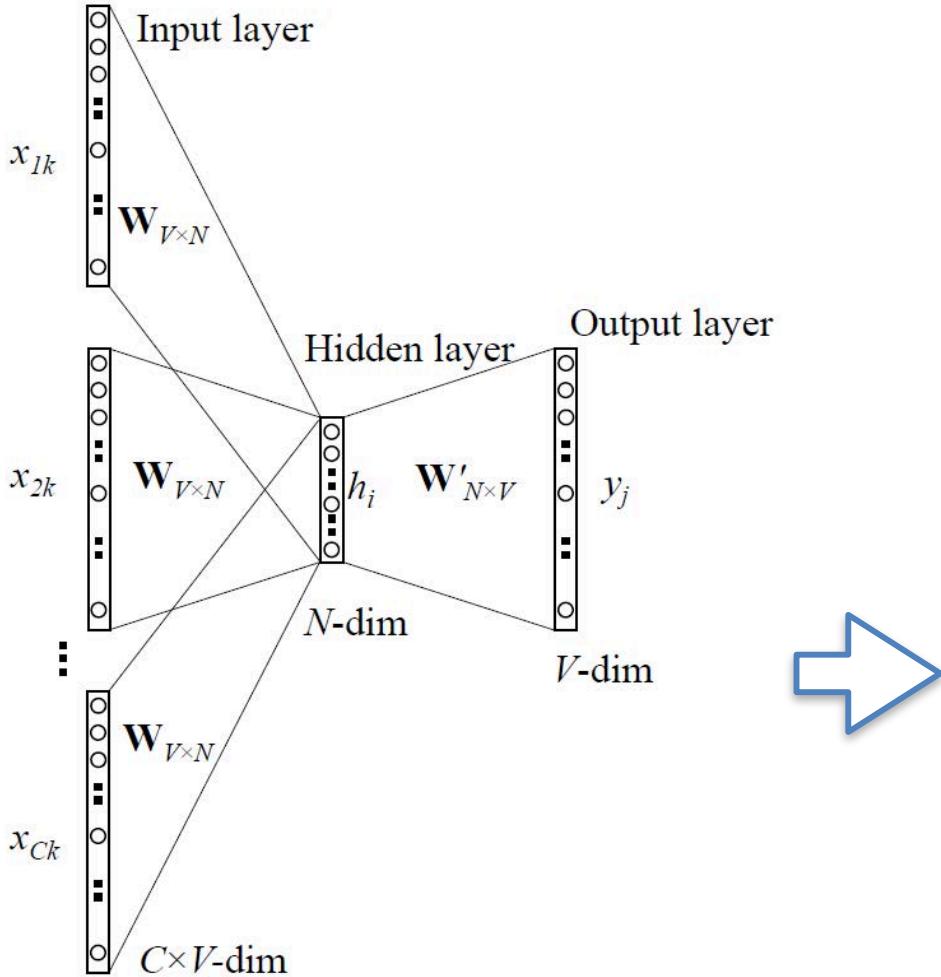
$$\mathbf{y} = \text{softmax}(\mathbf{W}'\mathbf{h})$$

# Optional: Window of Size 2



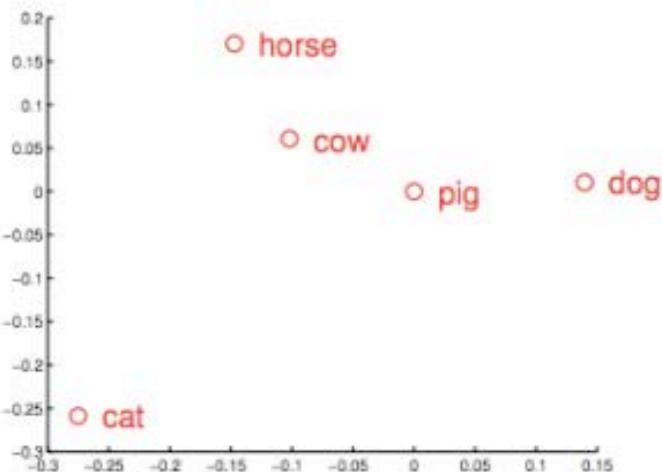
Once the training is complete, the  $W$  matrices associate to each word a vector of dimension  $N$ .

# Optional: Larger Windows

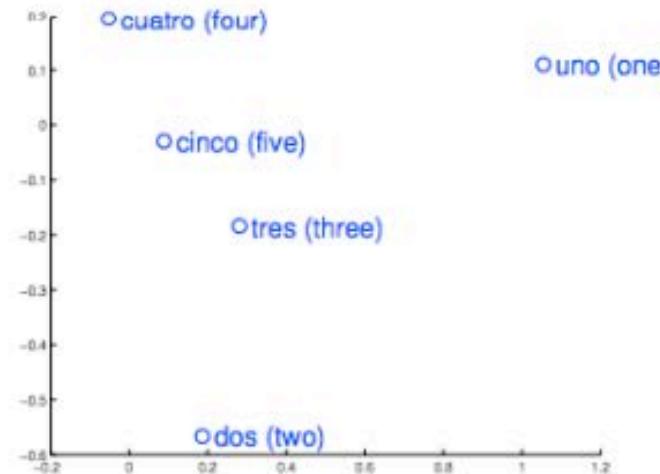


- Once the training is complete, the  $\mathbf{W}$  matrices associate to each word a vector of dimension  $N$ .
- The distances of between these vectors is highly correlated to the similarity of the corresponding words.

# Optional: Geometry of Words



English



Spanish

# Optional: Some Results

Training on Google News Vocab:

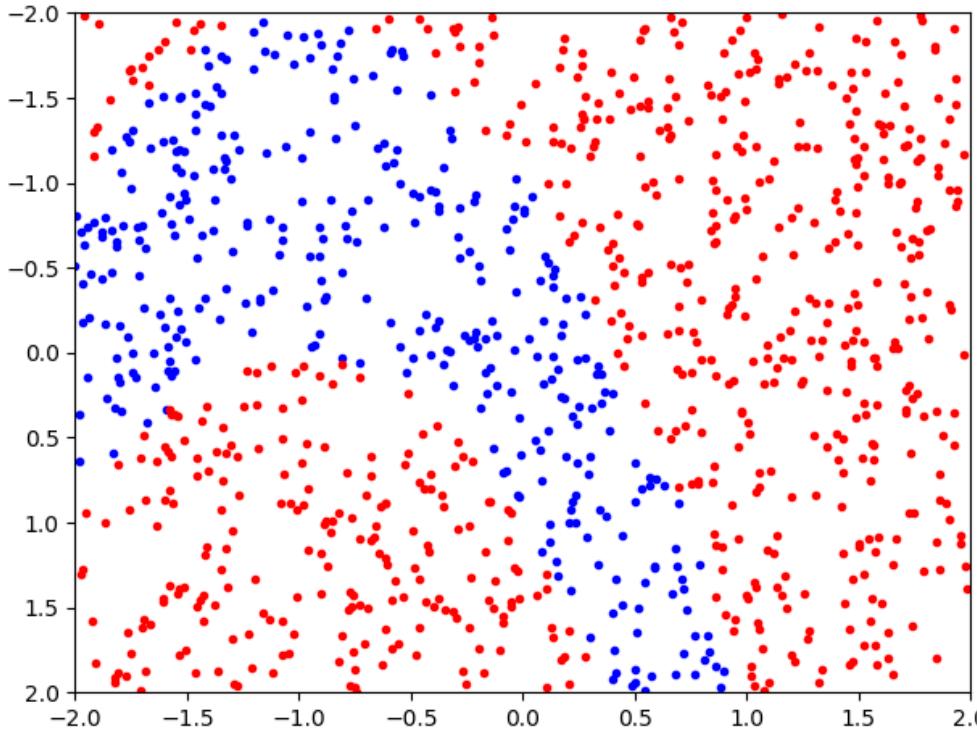
- Beijing is to China what Bern is to Switzerland

Testing:

- king:queen man:[woman,attempted,abduction,girl]
- knee:leg elbow:[forearm,arm,ulna\_bone]
- love:indifference fear:[apathy,callousmess,timidity,helplessness]

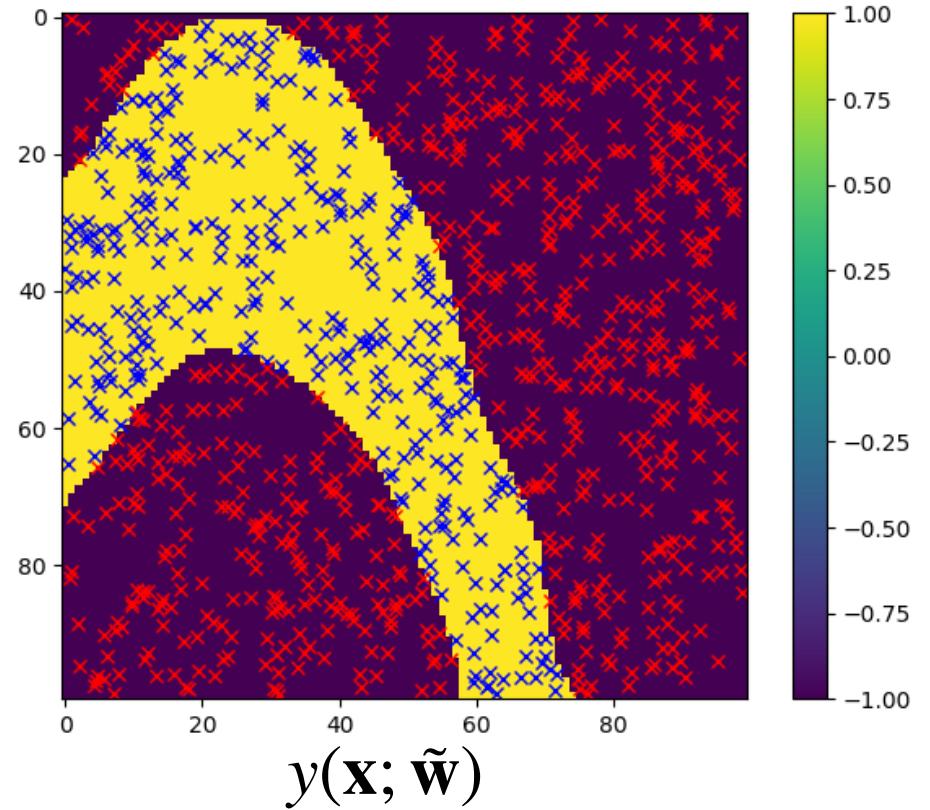
Not bad given that word2vec knows **nothing** about syntax, about the world, or even about logic!!

# From Classification to Regression



Positive:  $100(x_2 - x_1^2)^2 + (1 - x_1)^2 < 0.5$

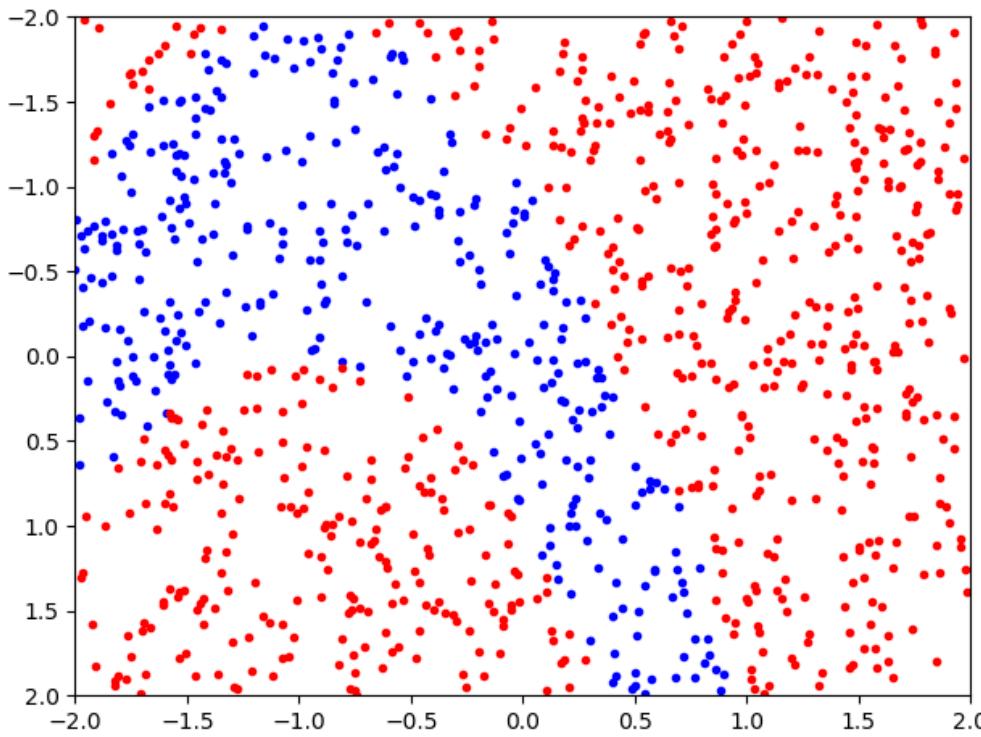
Negative: Otherwise



- The deep network implements the function  $f_{\tilde{\mathbf{w}}} = y(\cdot; \tilde{\mathbf{w}})$ .
- Ideally, we would like  $f_{\tilde{\mathbf{w}}}(\mathbf{x})$  to be approximately equal to the probability that  $\mathbf{x}$  belongs to the positive class.

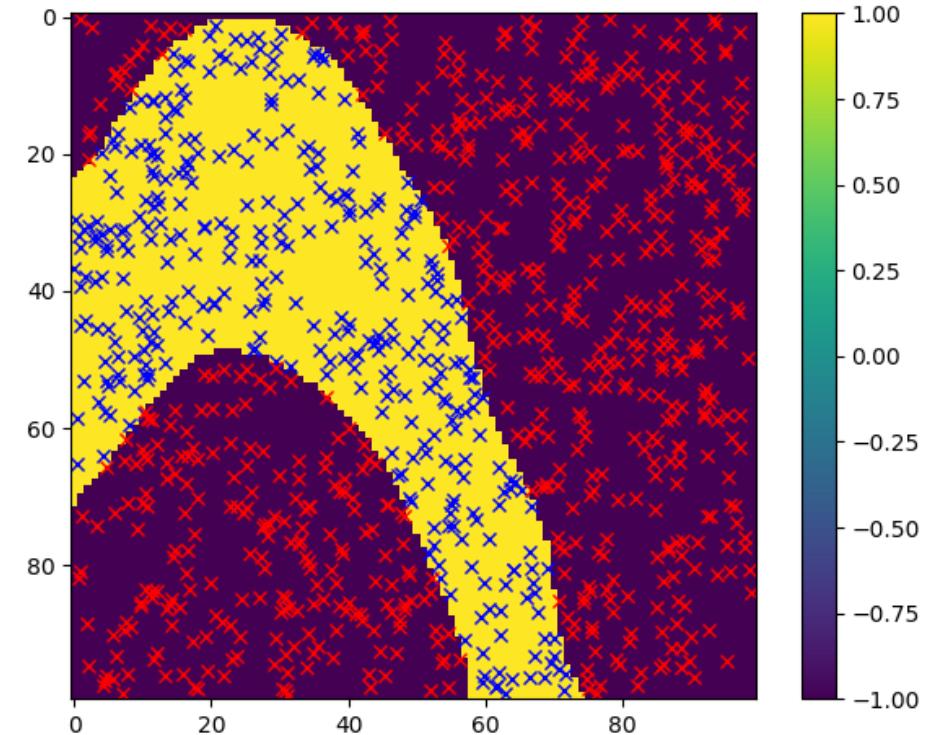
—> This raises a theoretical question: If  $f_{\tilde{\mathbf{w}}}$  were known, how well could a network approximate it?

# Rosenbrock Regression



Positive:  $100(x_2 - x_1^2)^2 + (1 - x_1)^2 < 0.5$

Negative: Otherwise

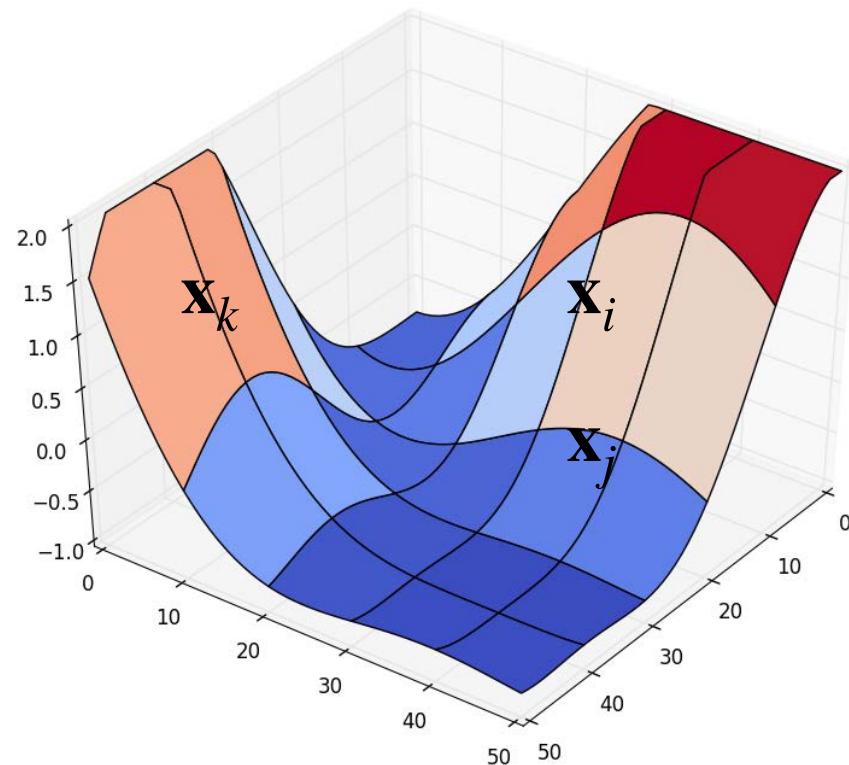


$y(\mathbf{x}; \tilde{\mathbf{w}})$  is now a non-linear function implemented by the network.

**Problem statement:** Find  $\tilde{\mathbf{w}}$  such that

$$y(\mathbf{x}; \tilde{\mathbf{w}}) \approx 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$

# Non-Linear Regression



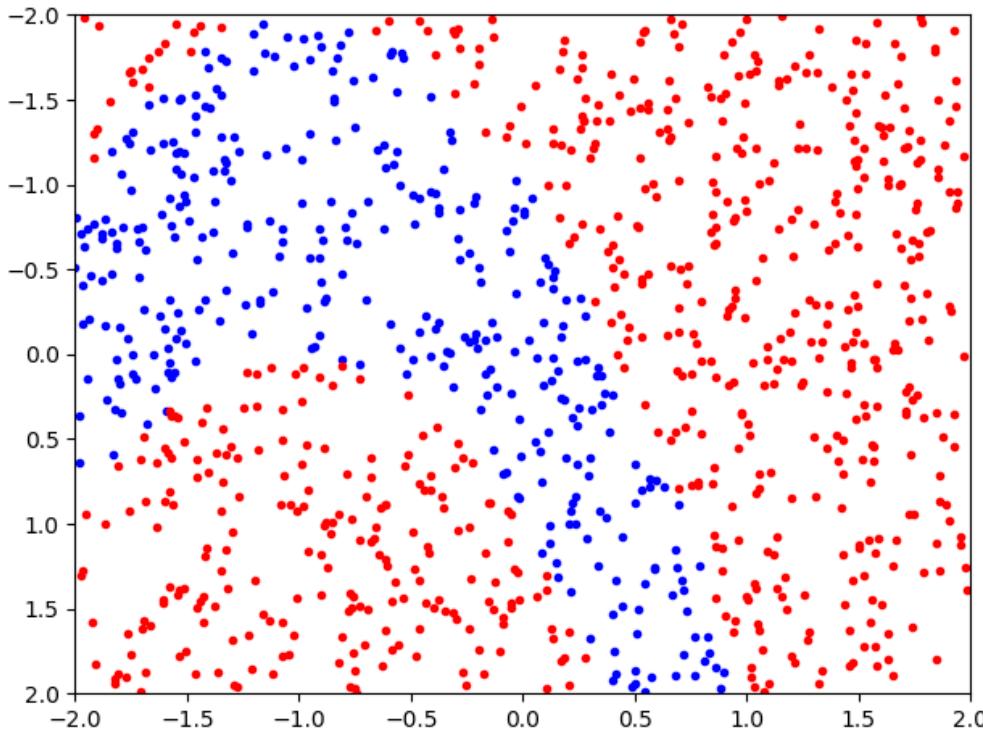
$$z = f(\mathbf{x})$$

**Problem statement:** Given  $(\{\mathbf{x}_1, z_1\}, \dots, \{\mathbf{x}_n, z_n\})$ , minimize

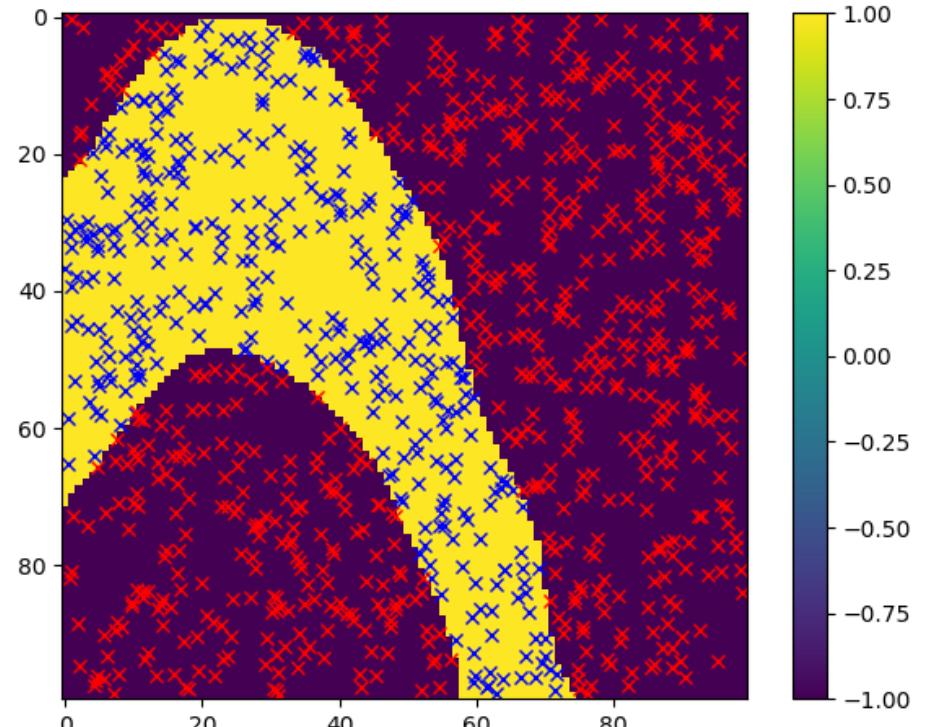
$$\sum_i (z_i - f(\mathbf{x}_i, \tilde{\mathbf{w}}))^2$$

w.r.t.  $\tilde{\mathbf{w}}$ .

# Classification / Regression



Positive:  $f(\mathbf{x}) < 0.5$   
Negative: Otherwise

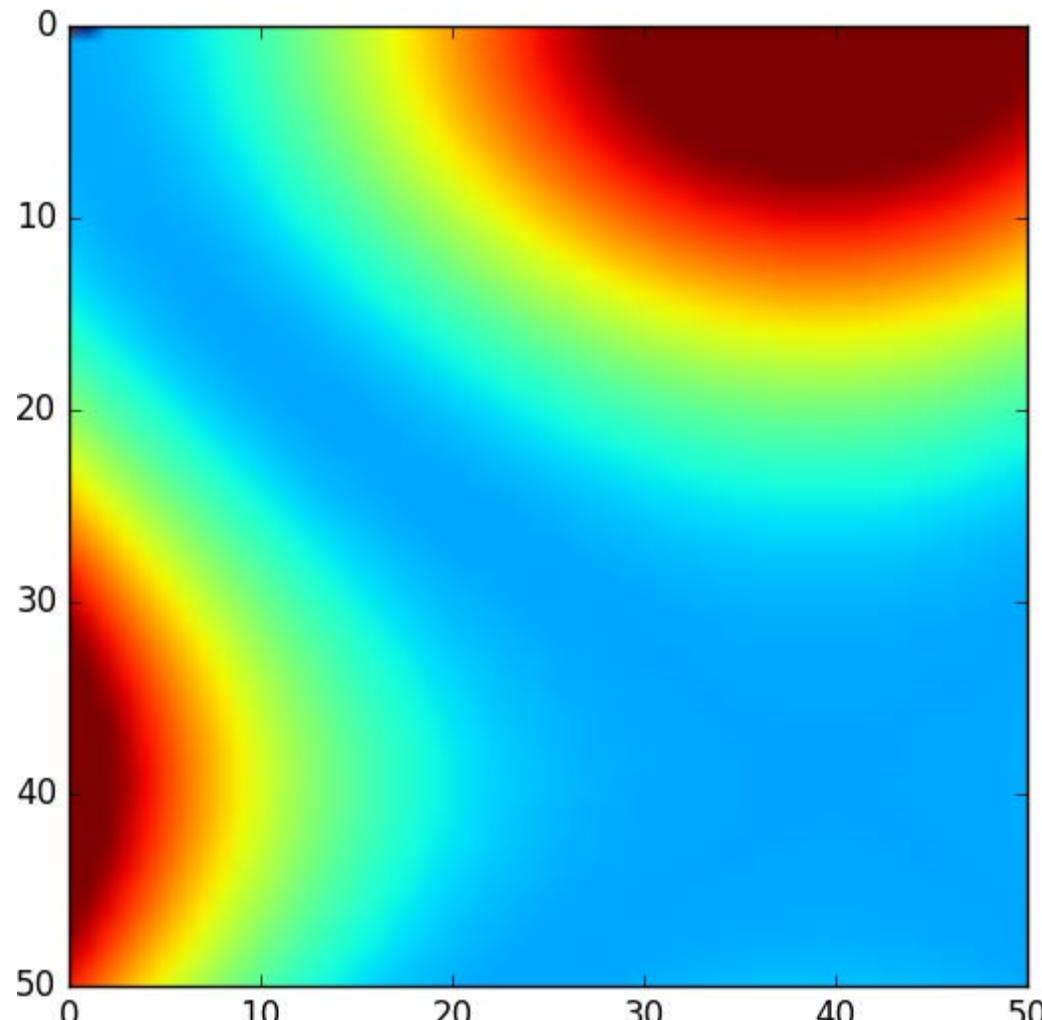


$y(\mathbf{x}; \tilde{\mathbf{w}})$  is now a non-linear function implemented by the network.

Classification can be understood as finding  $\tilde{\mathbf{w}}$  such that

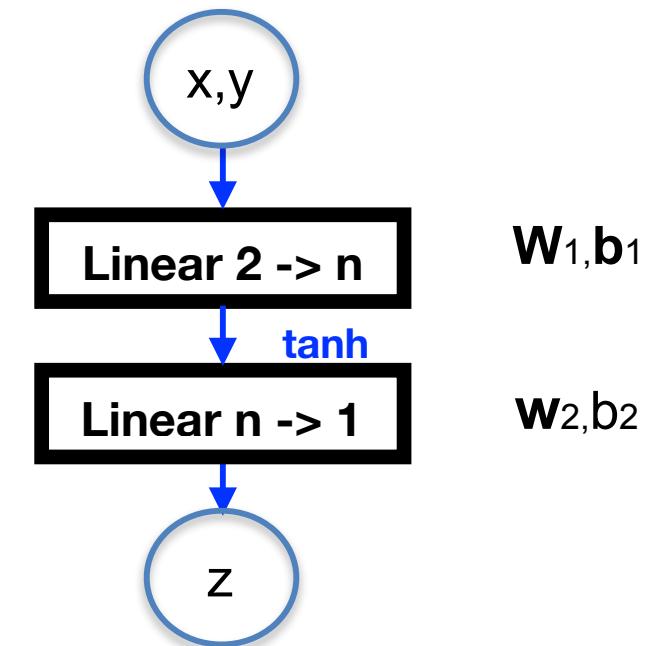
$$y(\mathbf{x}; \tilde{\mathbf{w}}) \approx f(x)$$

# Regressing the Rosenbrock Function



$$z = f(\mathbf{x})$$

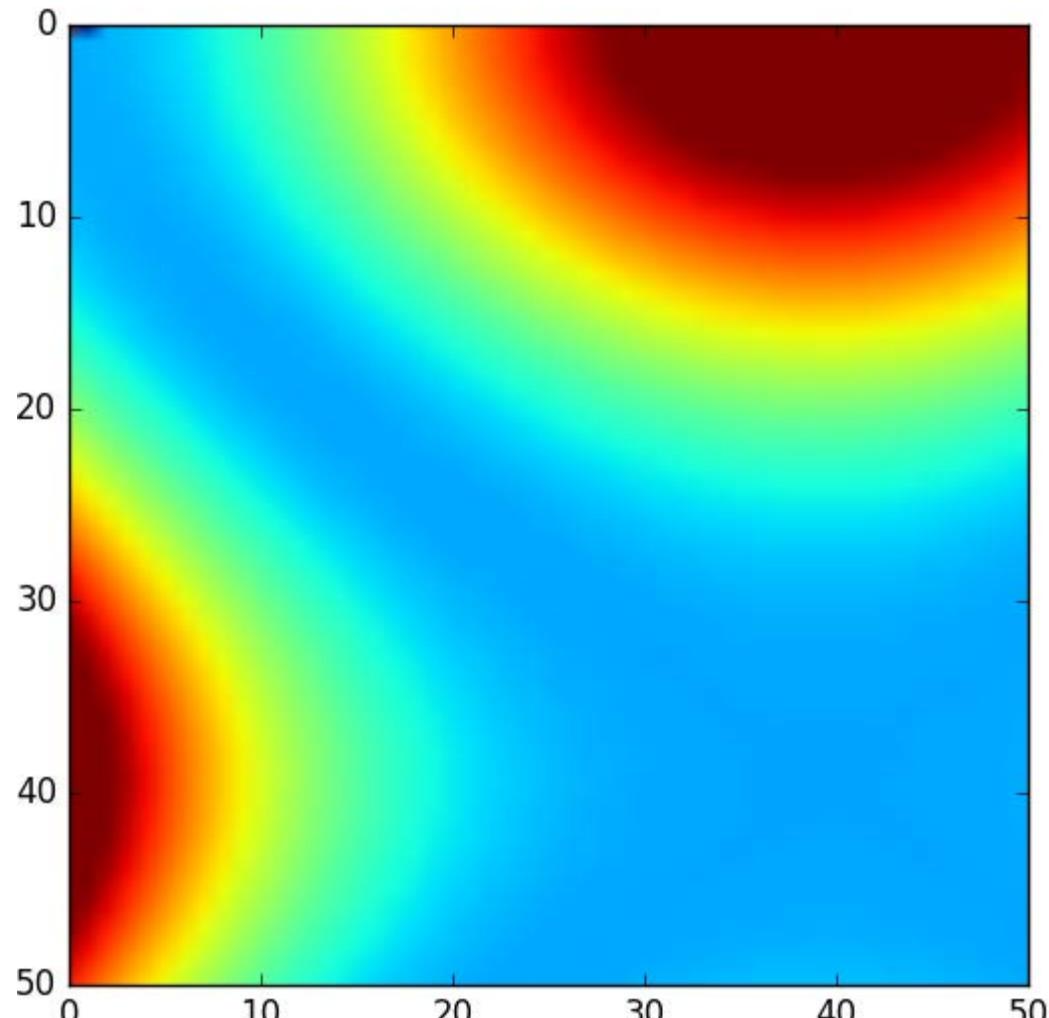
$$= 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$



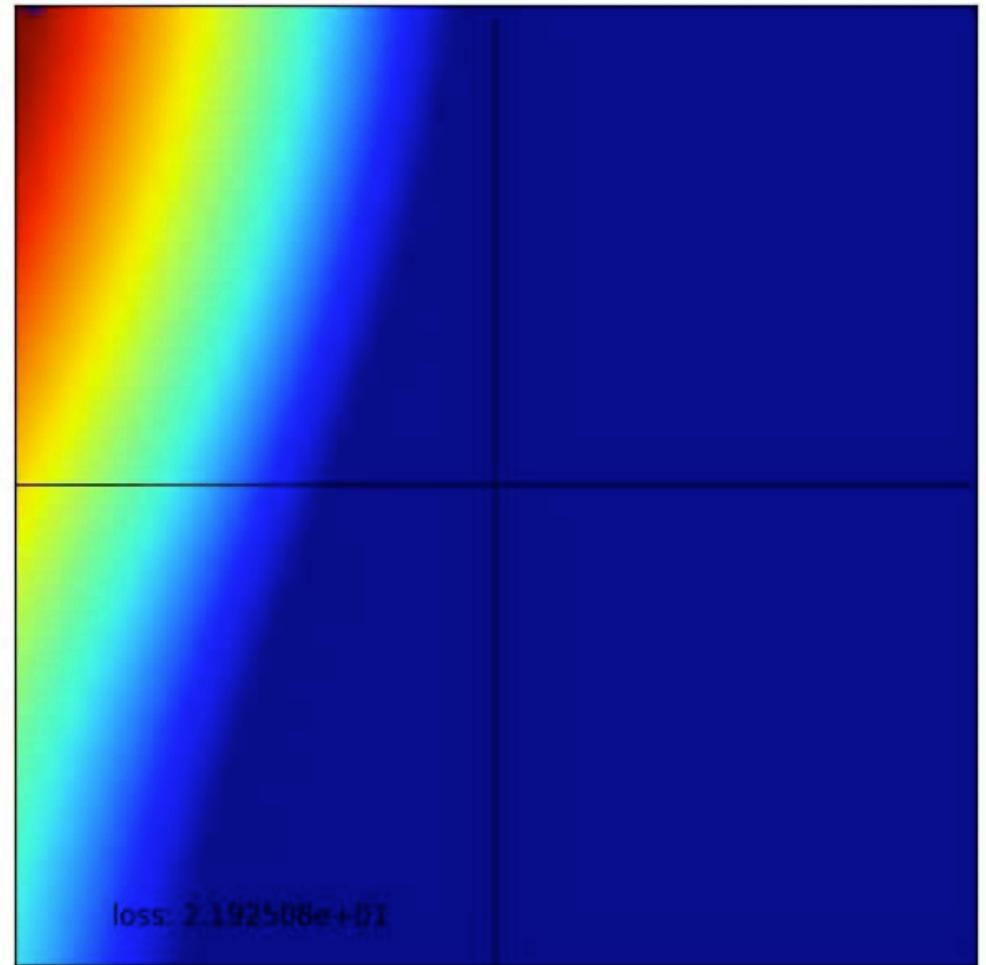
$$\begin{aligned} z &= f(x, y) \\ &= \mathbf{w}_2 \sigma(\mathbf{W}_1 \begin{bmatrix} x \\ y \end{bmatrix} + \mathbf{b}_1) + b_2 \end{aligned}$$

Minimize  $\sum_i (z_i - f(x_i, y_i))^2$ ,  
with respect to  $\mathbf{W}_1, \mathbf{w}_2, b_x, b_y, b_z$ .

# Regressing the Rosenbrock Function



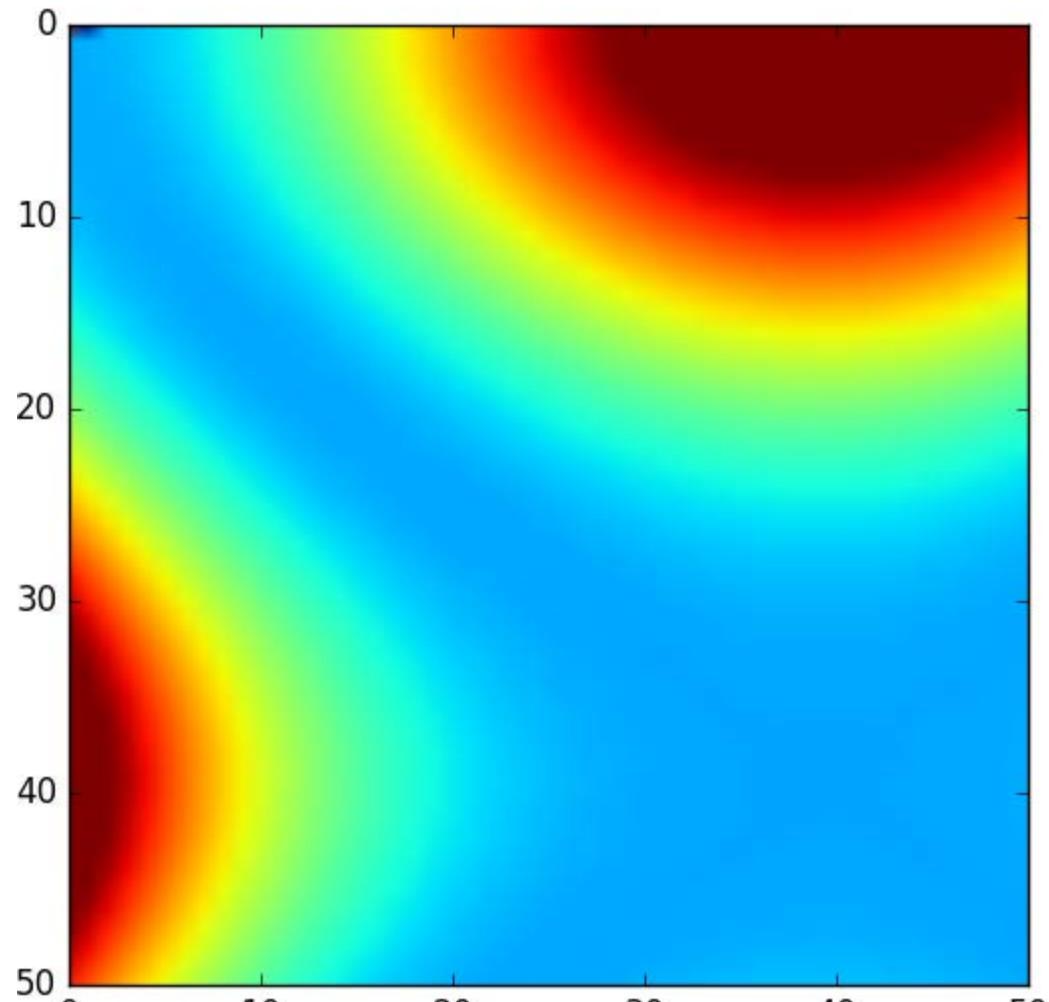
$$z = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$



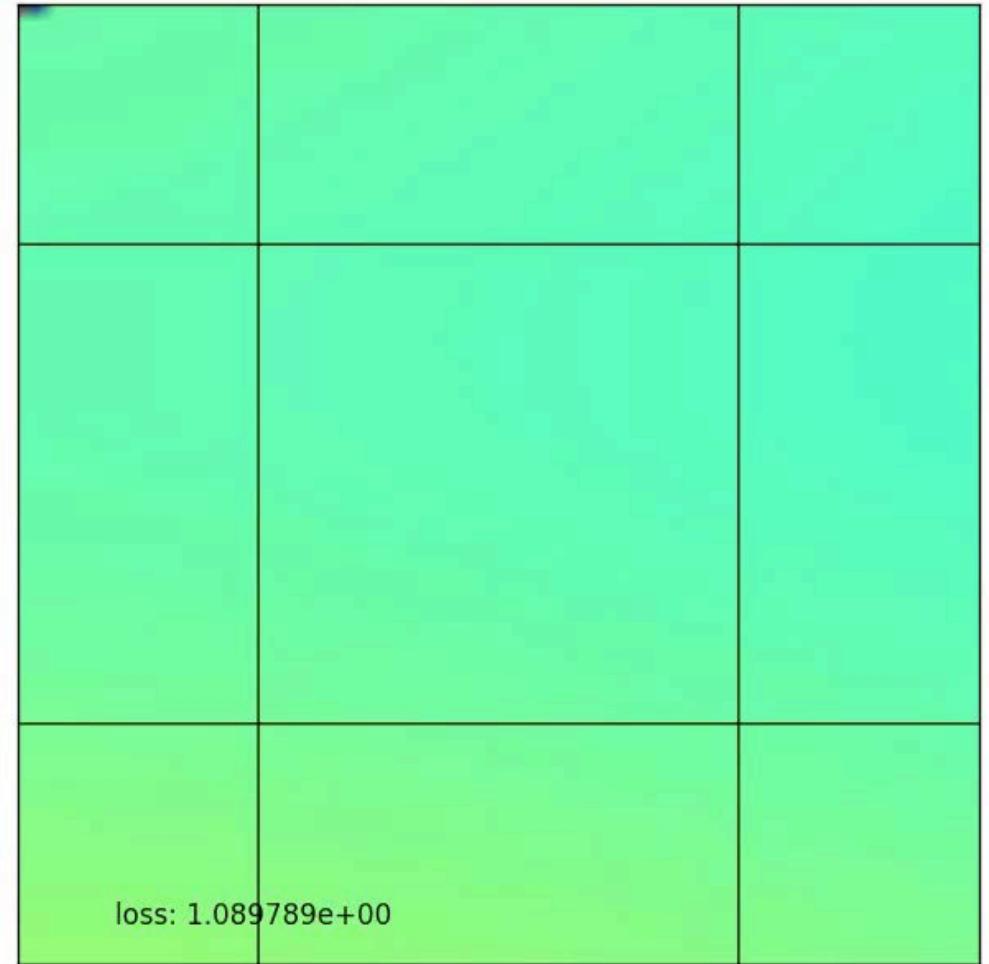
3-node hidden layer

→ 3 nodes is not quite enough.

# Regressing the Rosenbrock Function



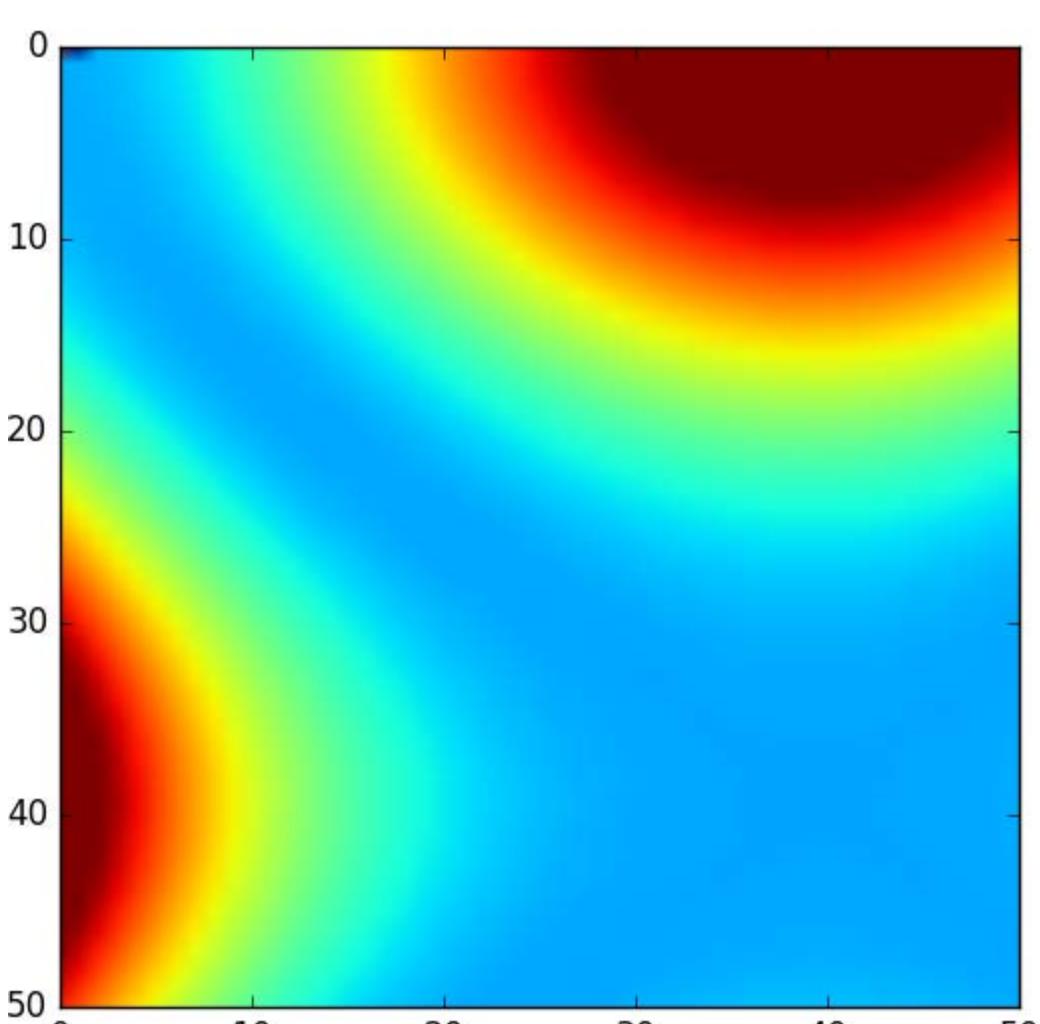
$$z = 100(x_2 - x_1^2)^2 + (1 - x_1)^2$$



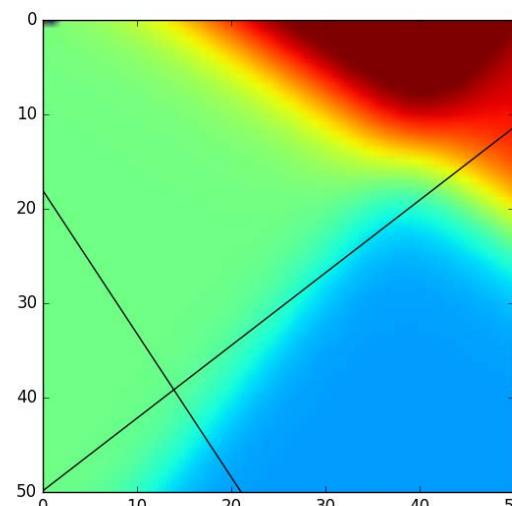
4-node hidden layer

→ 4 nodes is better.

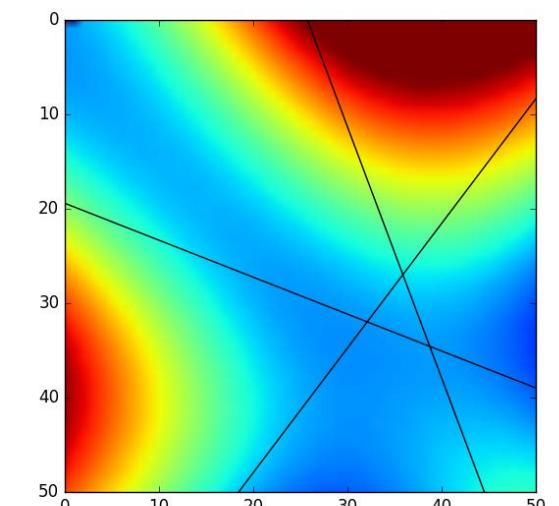
# Accuracy as a Function of the Number of Nodes



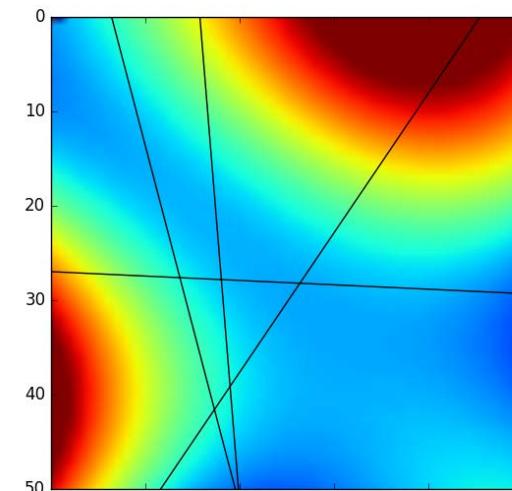
$$z = 100 * (y - x^2)^2 + (1 - x)^2$$



2 nodes -> loss 3.02e-01



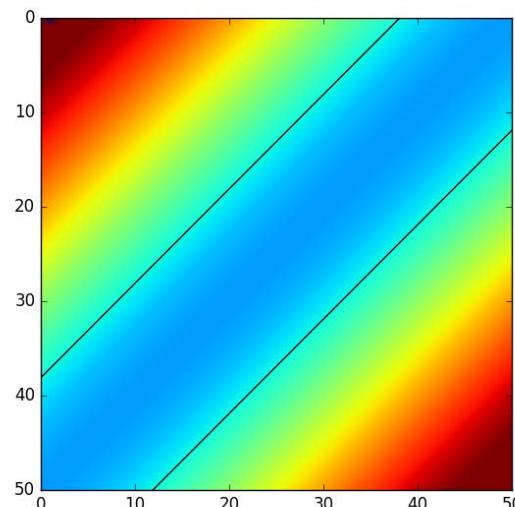
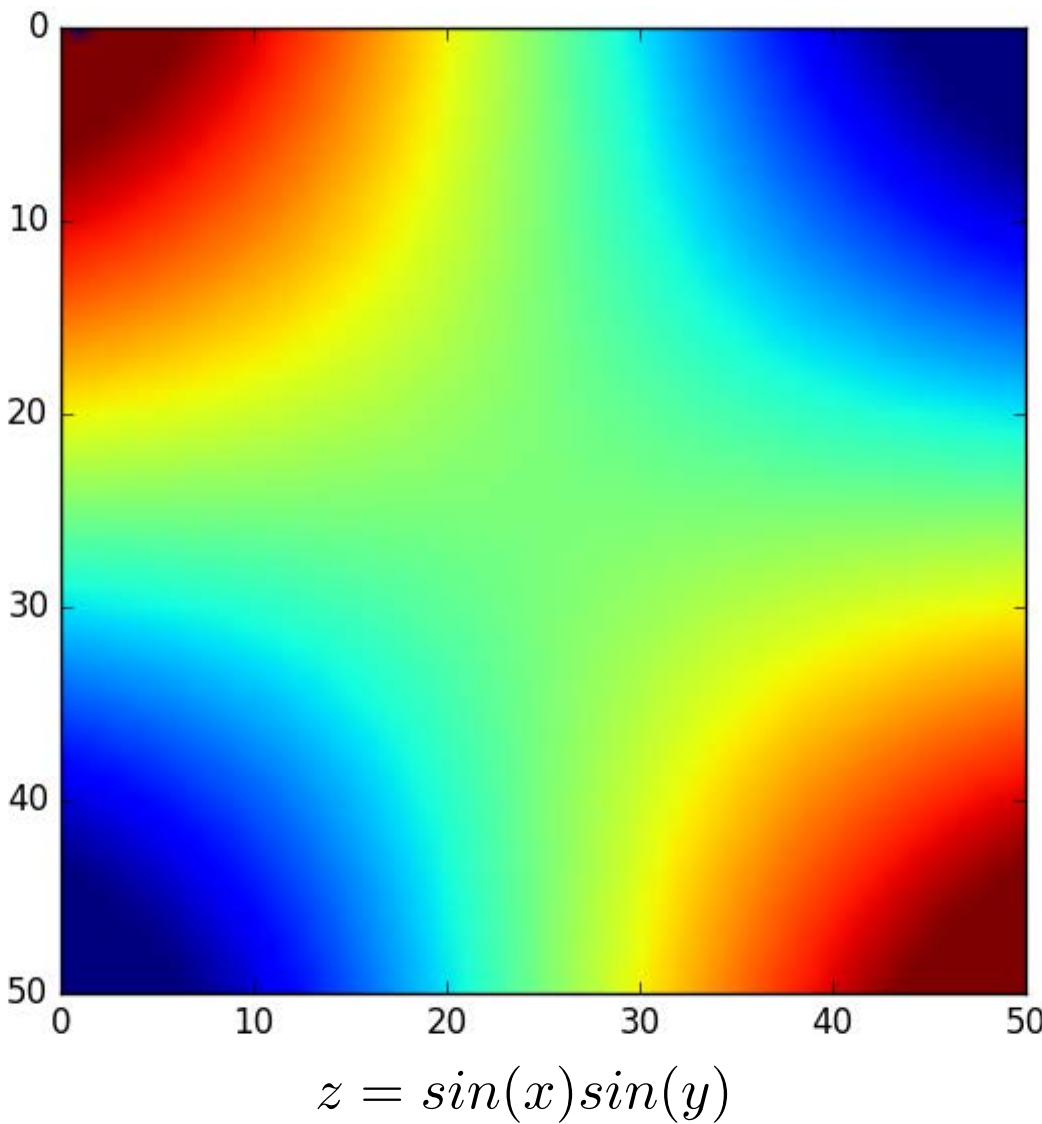
3 nodes -> loss 2.08e-02



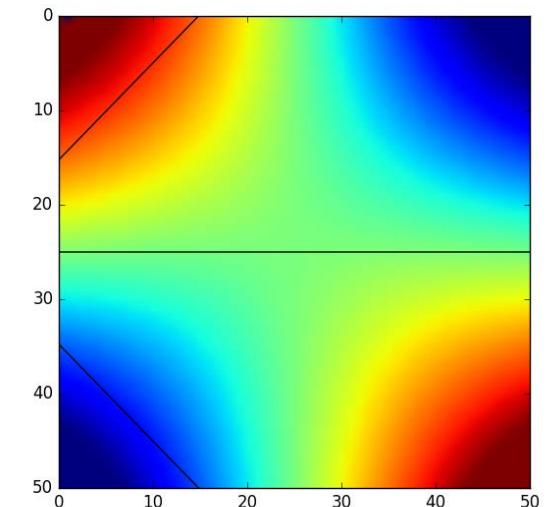
4 nodes -> loss 8.27e-03

→ The more nodes the more accurate the approximation.

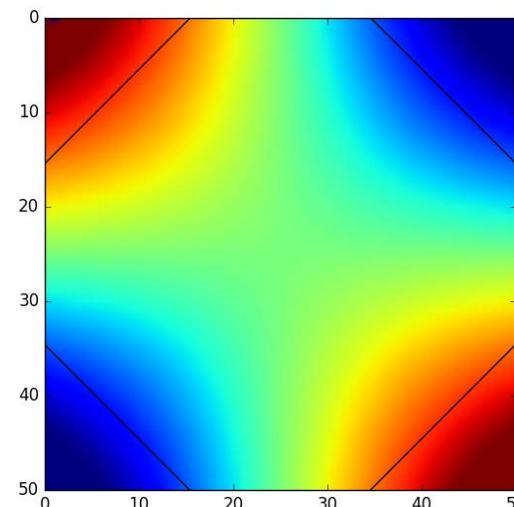
# Accuracy as a Function of the Number of Nodes



2 nodes -> loss 2.61e-01



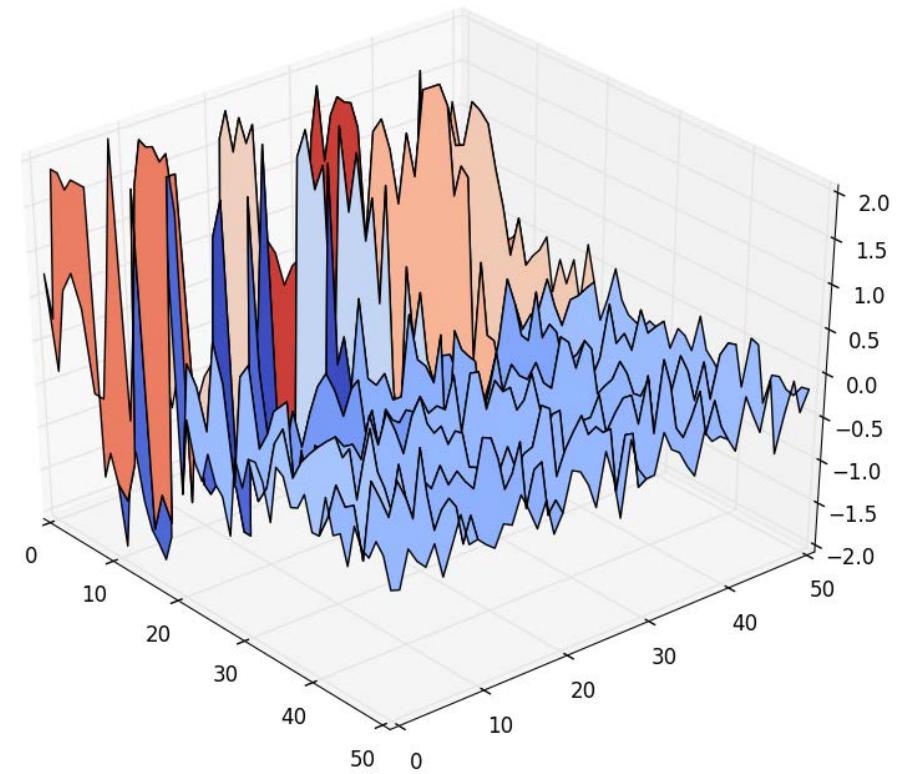
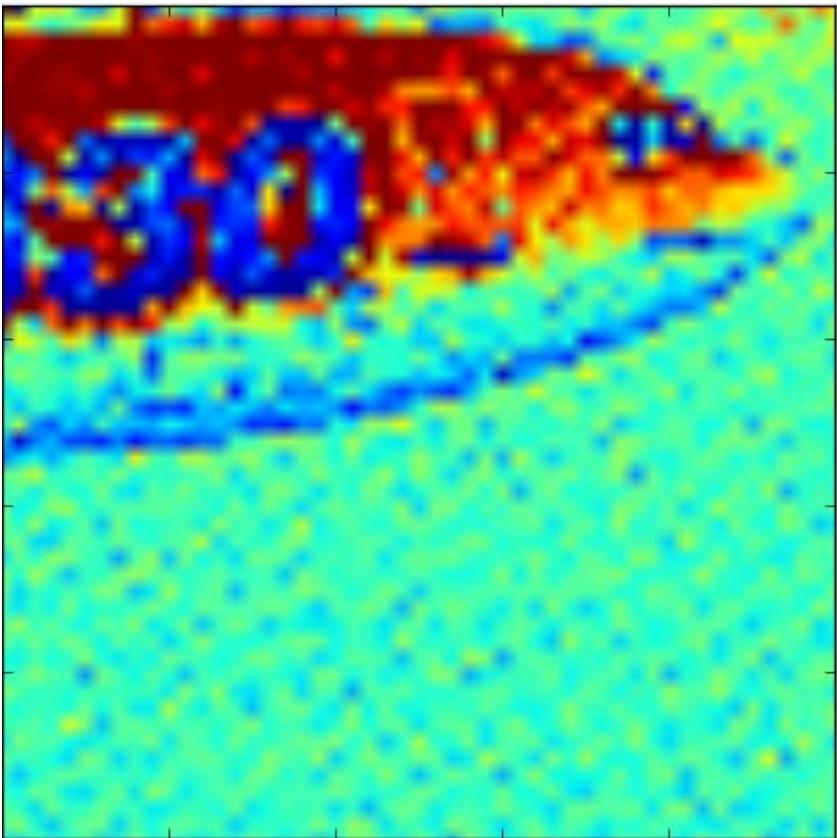
3 nodes -> loss 2.51e-04



4 nodes -> loss 3.07e-07

→ The more nodes the more accurate the approximation.

# Image as a 3D Surface

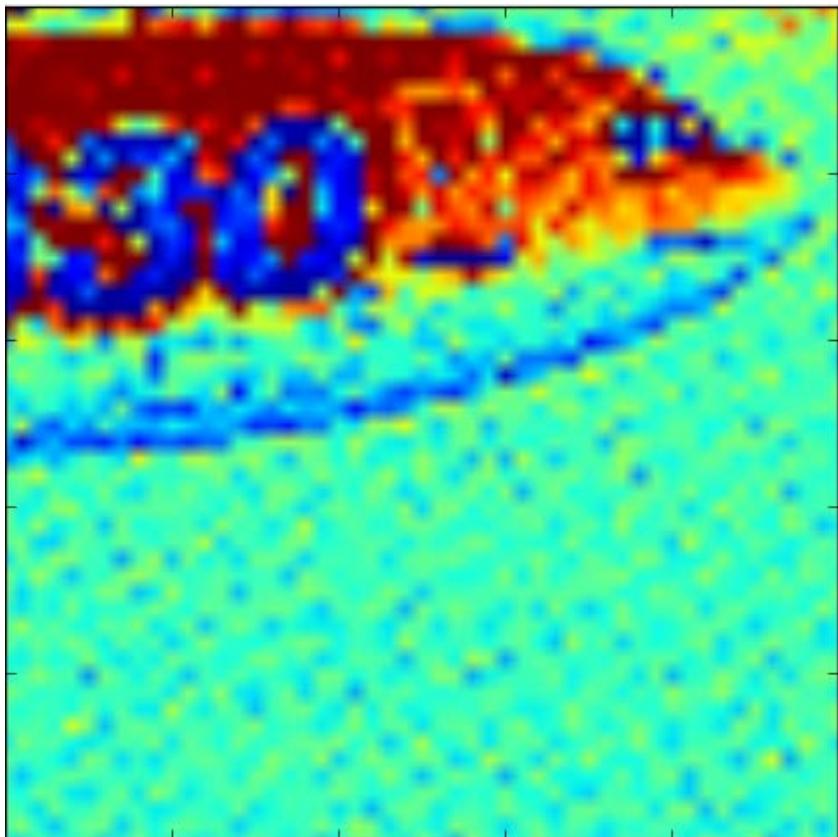


$$I = f(x, y)$$

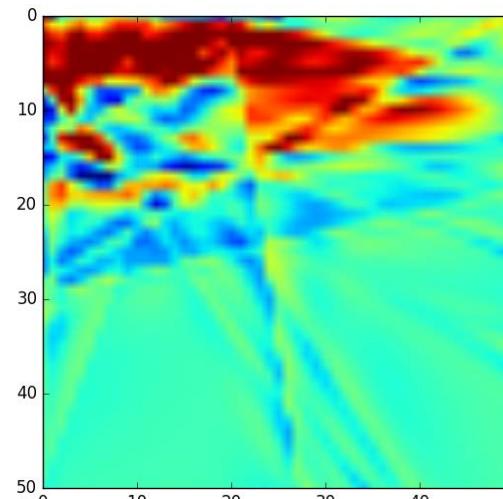
We treat the image as a surface:

- The intensity is the z coordinate.
- It is shown in false color on the left.
- The corresponding surface is complex.

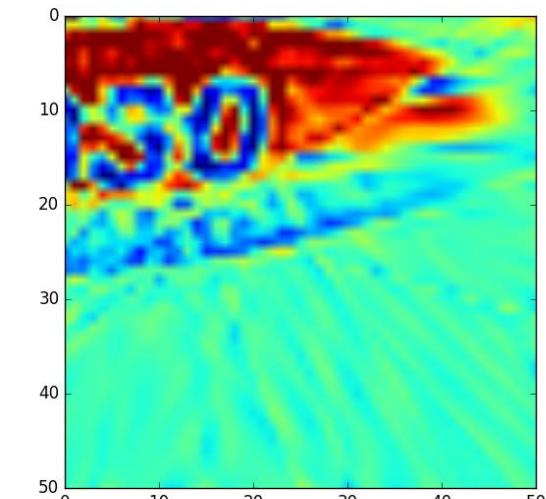
# More Complex Surface



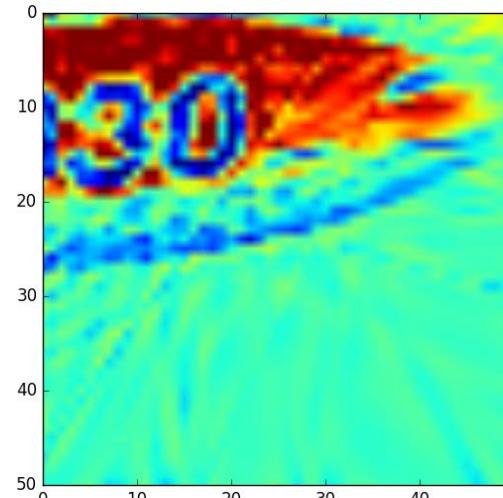
$$I = f(x, y)$$



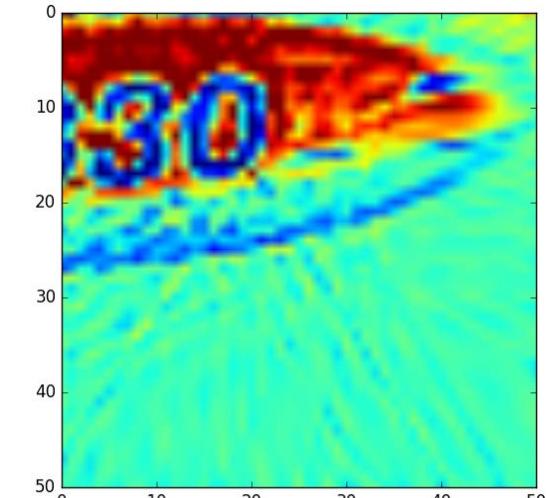
50 nodes -> loss 3.65e-01



100 nodes -> loss 2.50e-01



125 nodes -> loss 2.40e-01



300 nodes -> loss 1.92e-01

→ The more nodes the more accurate the approximation.

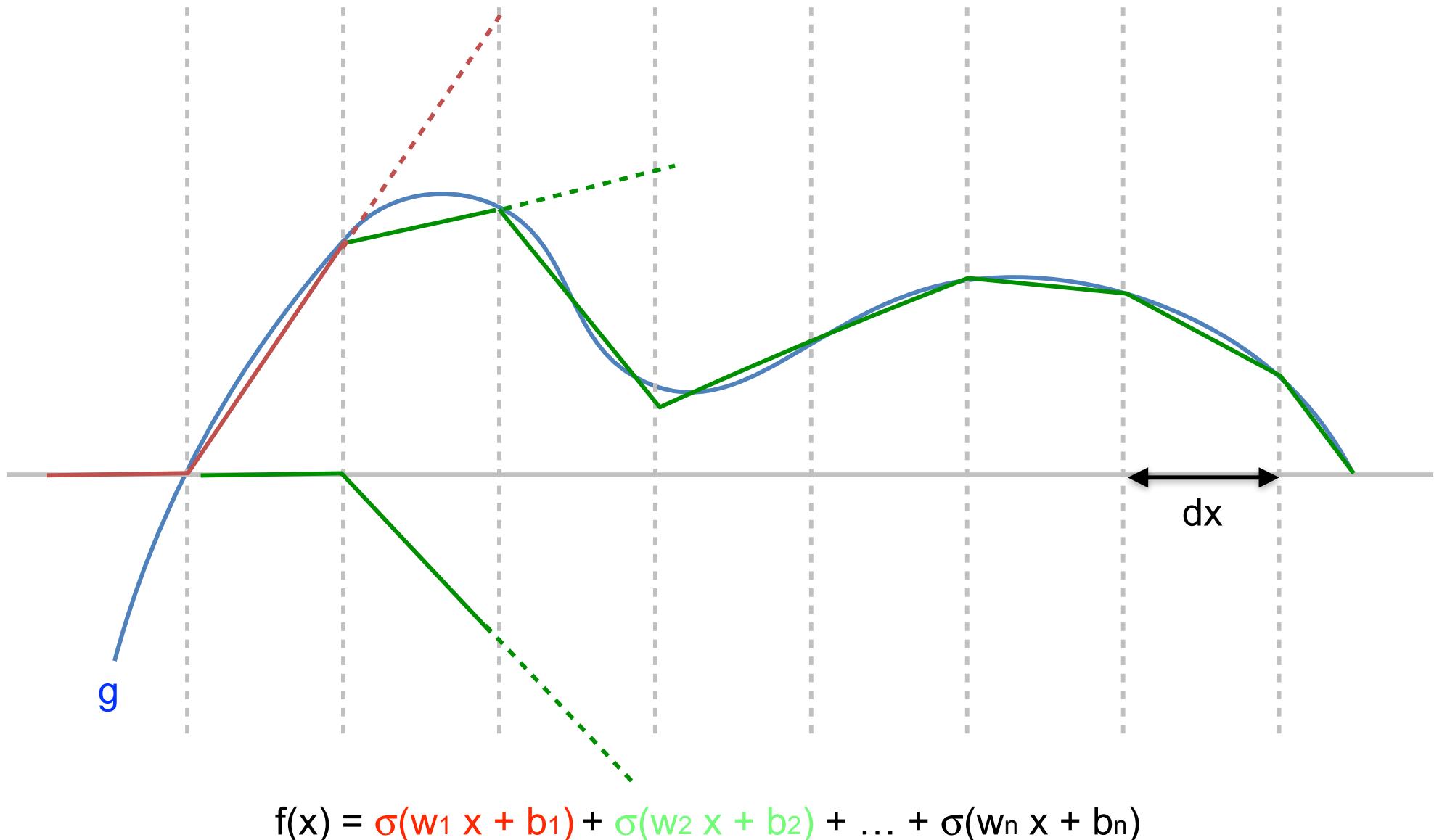
# Universal Approximation Theorem

A feedforward network with a linear output layer and at least one hidden layer with any 'squashing' activation function (e.g. logistic sigmoid) can approximate any Borel measurable function (from one finite-dimensional space to another) with any desired nonzero error.

Any continuous function on a closed and bounded set of  $R^n$  is Borel-measurable.

—> In theory, any reasonable function can be approximated by a one-hidden layer network as long as it is continuous.

# Universal Approximation Theorem in 1D

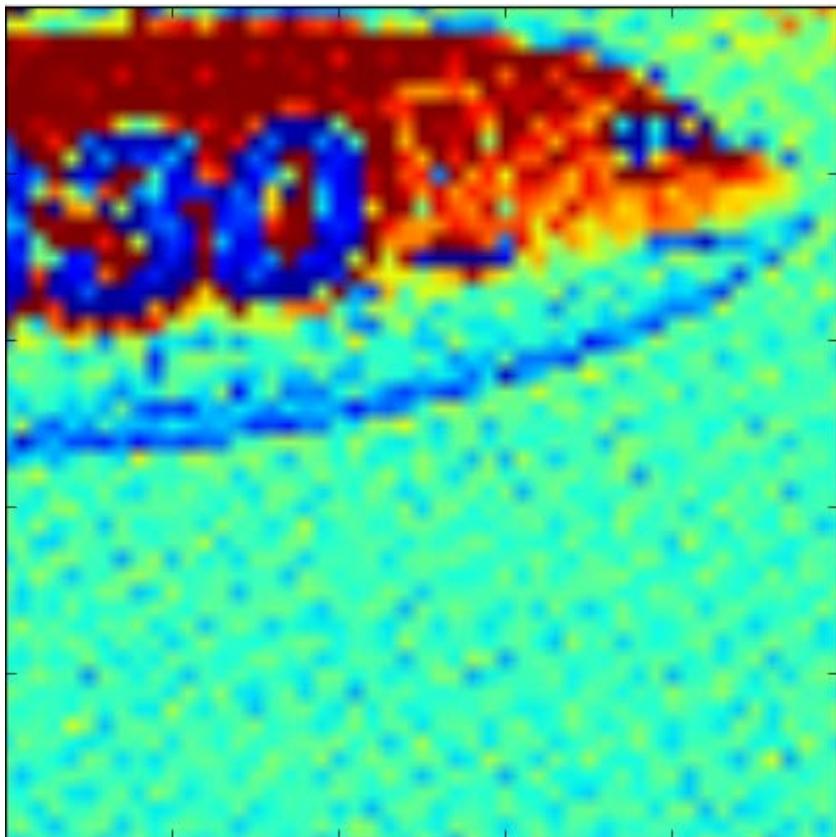


When  $dx \rightarrow 0$ ,  $f \rightarrow g$ .

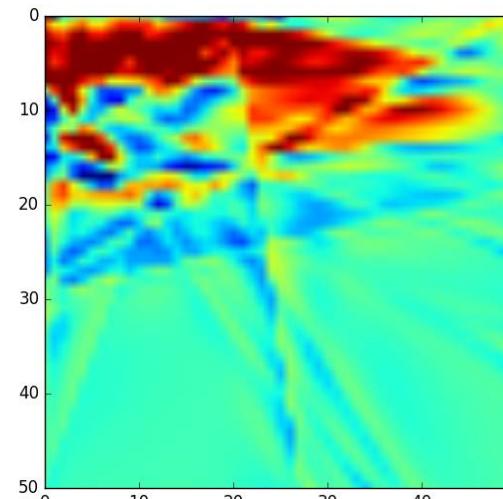
# Universal Approximation Theorem in nD

- The sine function can be approximated arbitrarily well.
- According to Fourier analysis, continuous functions from  $R^n$  into R can be approximated arbitrarily well by a weighted sum of sine functions.

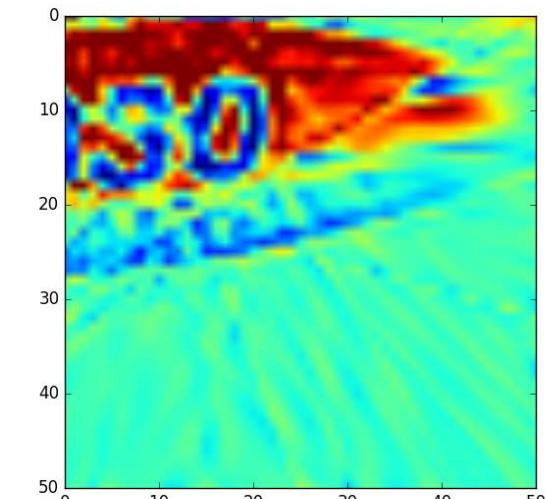
# More Complex Surface



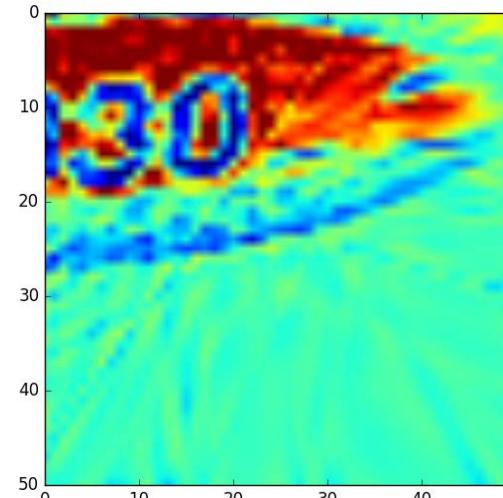
$$I = f(x, y)$$



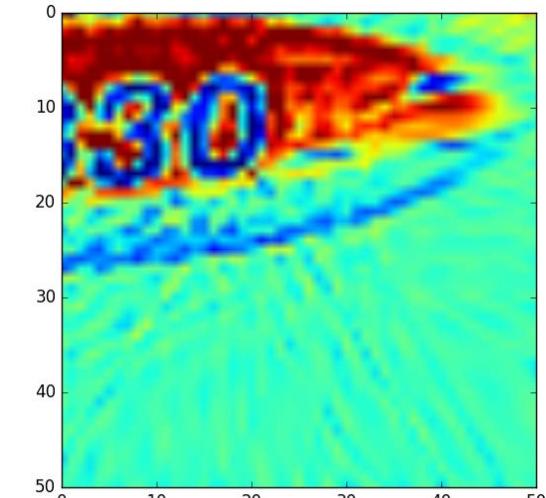
50 nodes -> loss 3.65e-01



100 nodes -> loss 2.50e-01



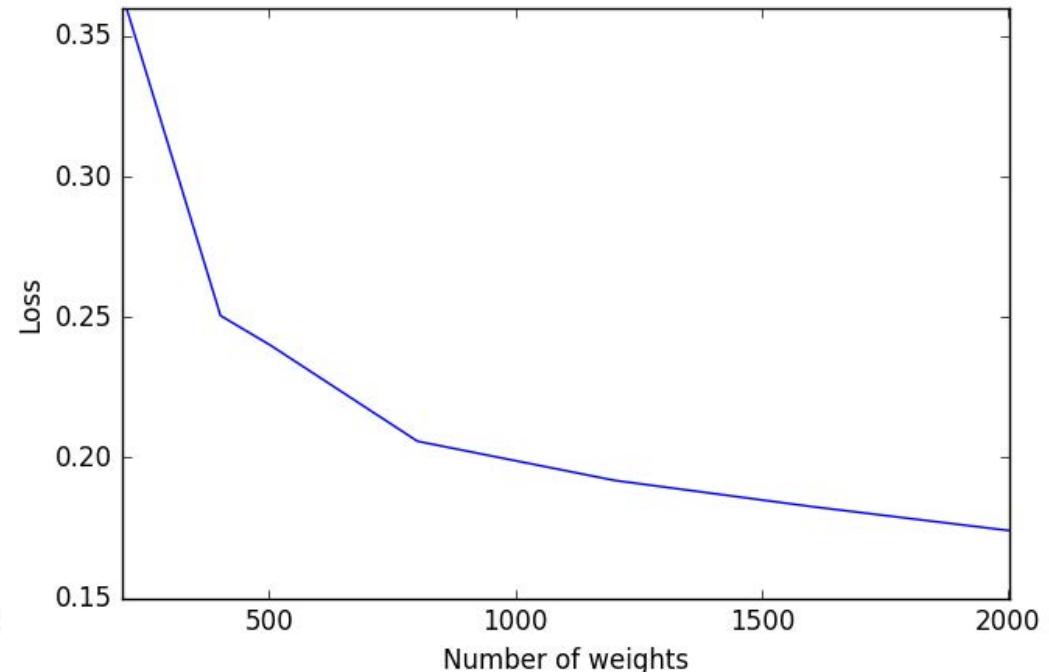
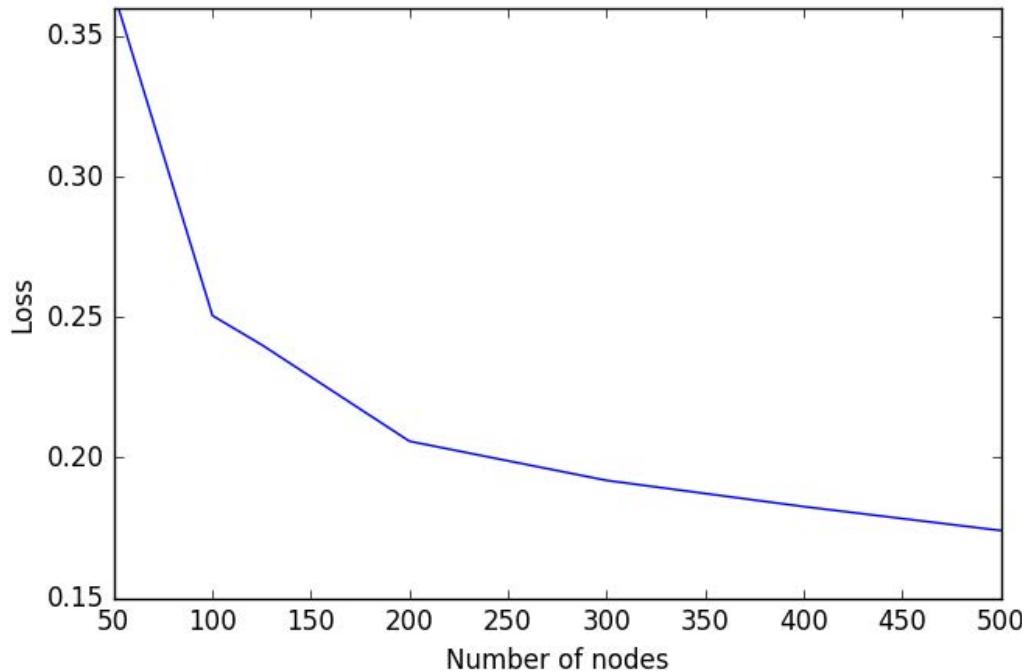
125 nodes -> loss 2.40e-01



300 nodes -> loss 1.92e-01

→ The more nodes the more accurate the approximation.

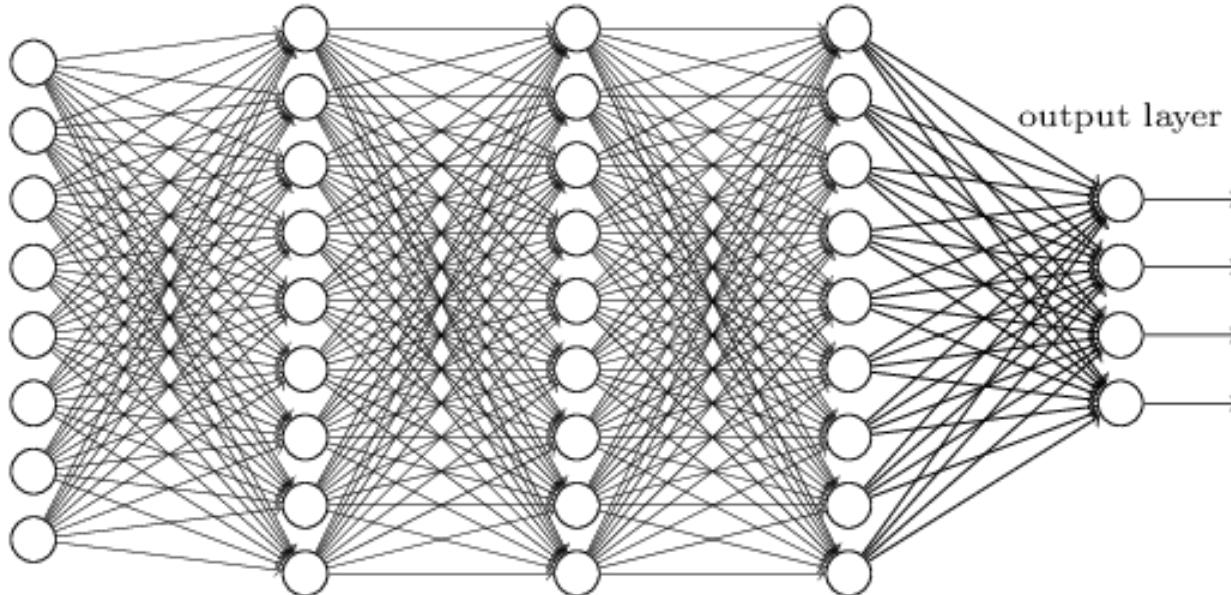
# In Practice



- It may take an exponentially large number of parameters for a good approximation.
  - The optimization problem becomes increasingly difficult.
- The one hidden layer perceptron may not converge to the best solution!

# From MLP to Deep Learning

input layer      hidden layer 1    hidden layer 2    hidden layer 3



$$\begin{aligned} \mathbf{h}_1 &= \sigma_1(\mathbf{W}_1 \mathbf{x}_1 + \mathbf{b}_1) \\ \mathbf{h}_2 &= \sigma_2(\mathbf{W}_2 \mathbf{h}_2 + \mathbf{b}_2) \\ &\dots \\ \mathbf{y} &= \sigma_n(\mathbf{W}_n \mathbf{h}_n + \mathbf{b}_n) \end{aligned}$$

- MLPs can have more than one hidden layer.
- Their descriptive power increases with the number of layers.

# PyTorch Translation

```
class MLP(nn.Module):
```

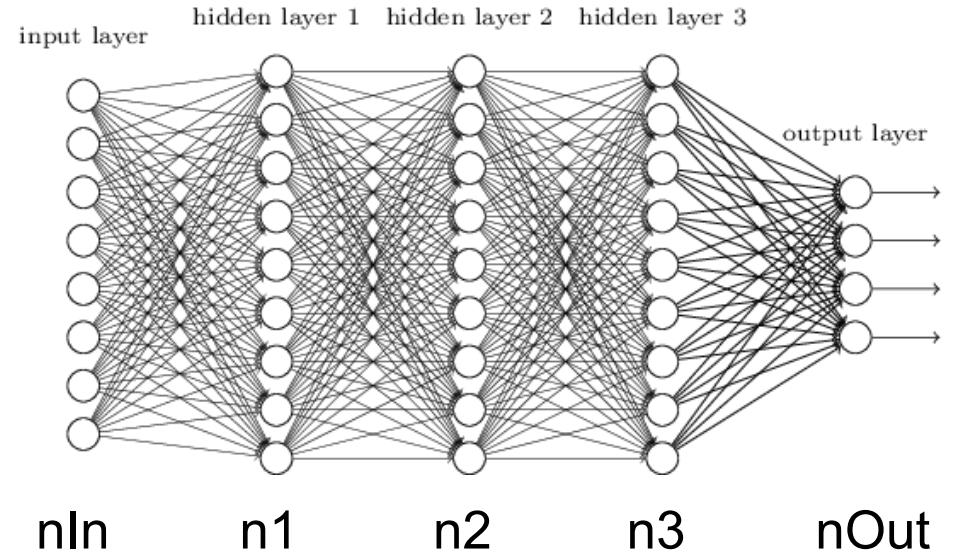
```
    def __init__(self,n1=10,n2=10,n3=10,nIn=2,nOut=1):
        self.l1 = nn.Linear(nIn,n1)
        self.l2 = nn.Linear(n1,n2)
        self.l3 = nn.Linear(n2,n3)
        self.l4 = nn.Linear(n3,nOut)
```

```
    def forward(self,x):
```

```
        h1 = sigm(self.l1(x))
        h2 = sigm(self.l2(h1))
        h3 = sigm(self.l3(h2))
        return(self.l4(h3))
```

```
    def loss(self,x,target):
```

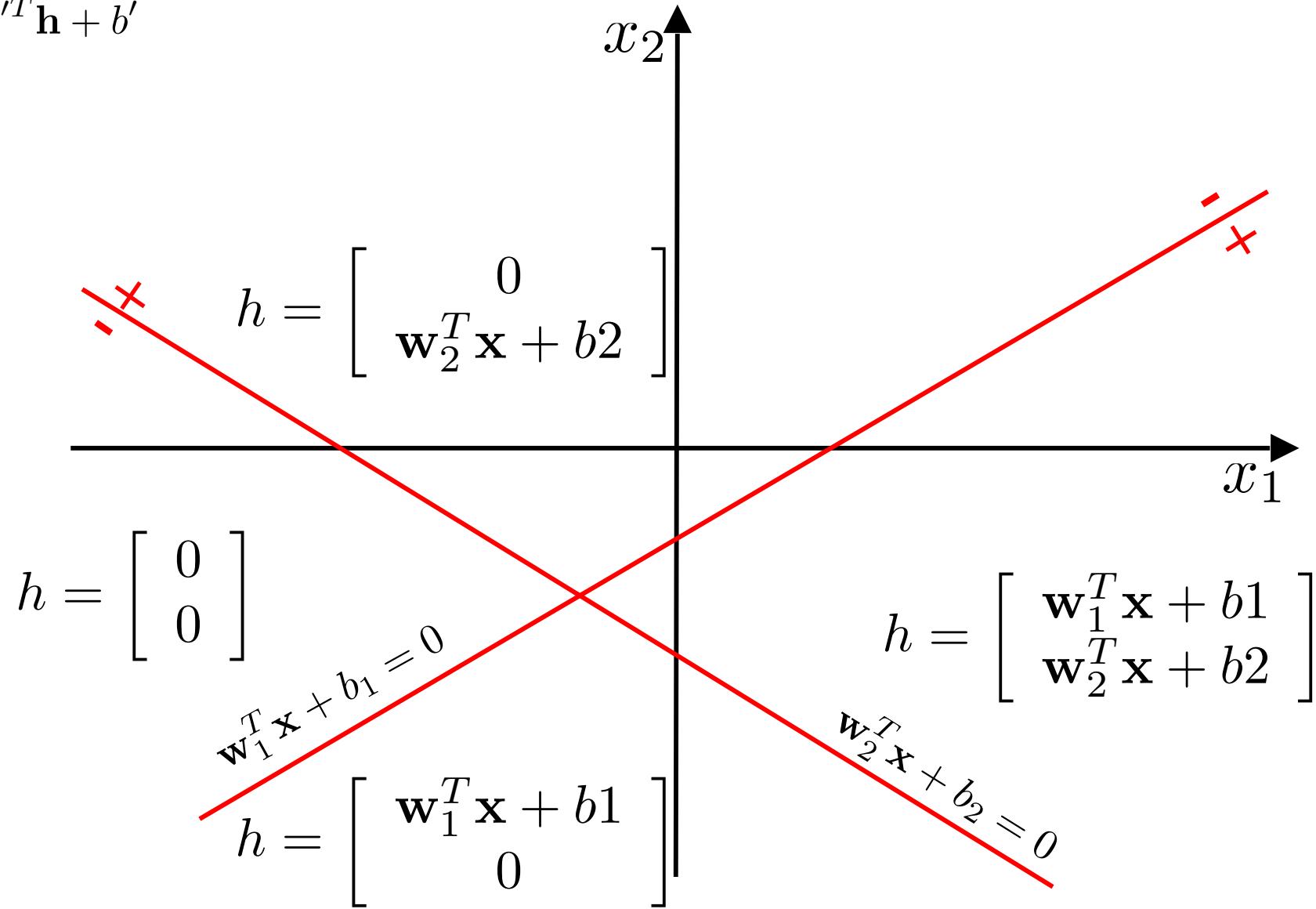
```
        loss_fn = torch.nn.CrossEntropyLoss()
        output = self(x)
        return loss_fn(output,target)
```



# One Layer: Two Hyperplanes

$$h = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0) \text{ with } \mathbf{W} = \begin{bmatrix} \mathbf{w}_1^T \\ \mathbf{w}_2^T \end{bmatrix} \text{ and } \mathbf{b} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

$$y = \mathbf{w}'^T \mathbf{h} + b'$$

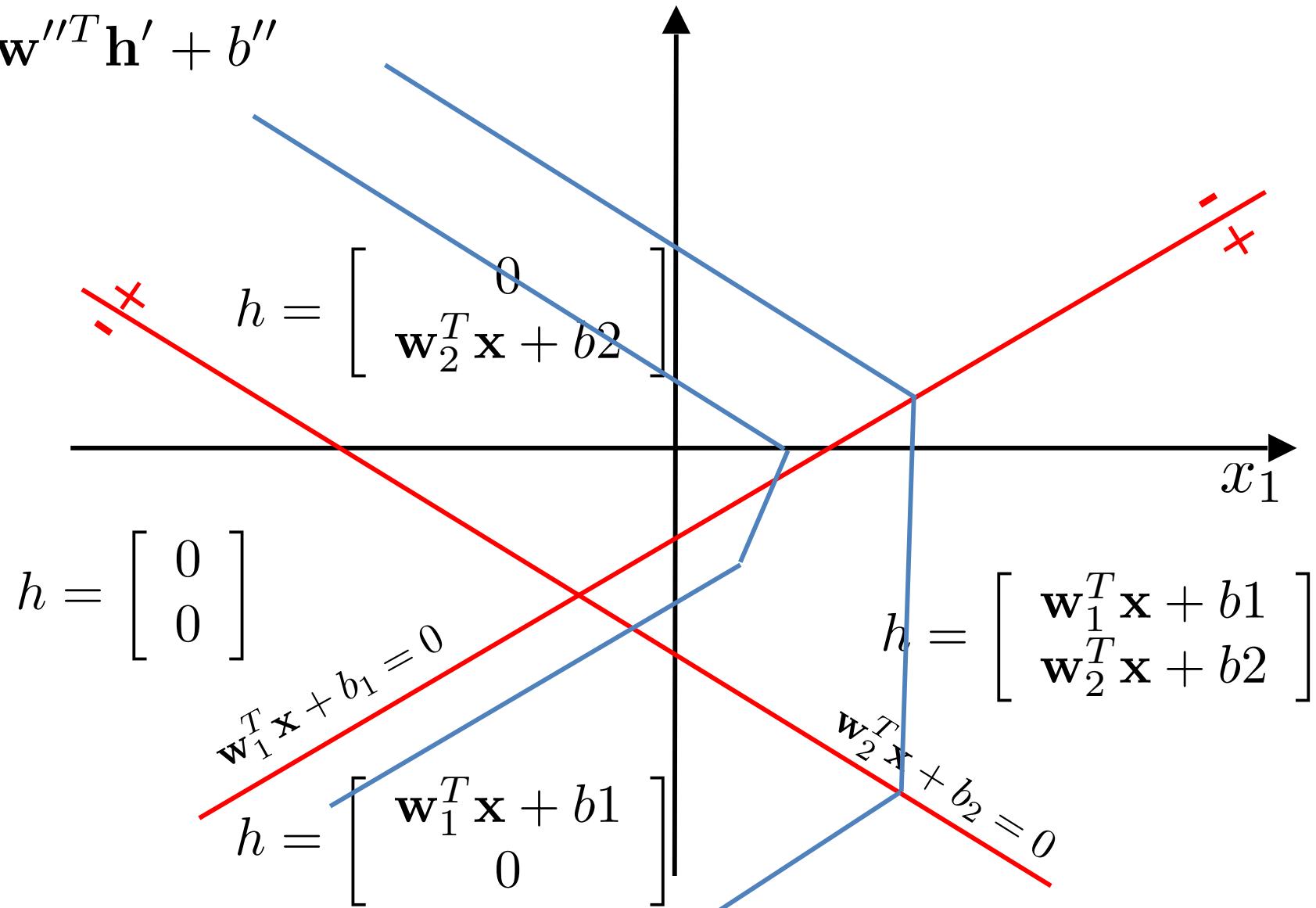


# Two Layers: Two Hyperplanes

$$h = \max(\mathbf{W}\mathbf{x} + \mathbf{b}, 0)$$

$$h' = \max(\mathbf{W}'\mathbf{h} + \mathbf{b}', 0)$$

$$y = \mathbf{w}''^T \mathbf{h}' + b''$$



# Multi Layer Perceptrons

The function learned by a DNN using either ReLU, Sigmoid, or Tanh operators is:



- piecewise affine or smooth;
- continuous because it is a composition of continuous functions.

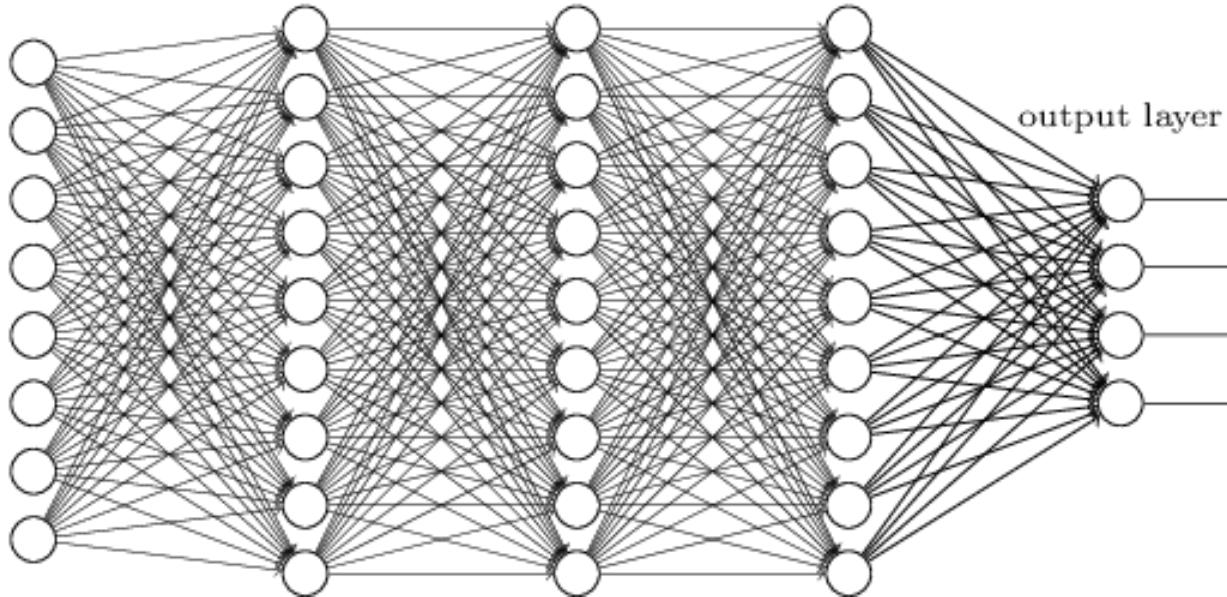
Each region created by a layer is split into smaller regions:

- Their boundaries are correlated in a complex way.
- Their descriptive power is **larger** than shallow networks for the **same number** of parameters.

shallow : poco profunda

# Deep Learning

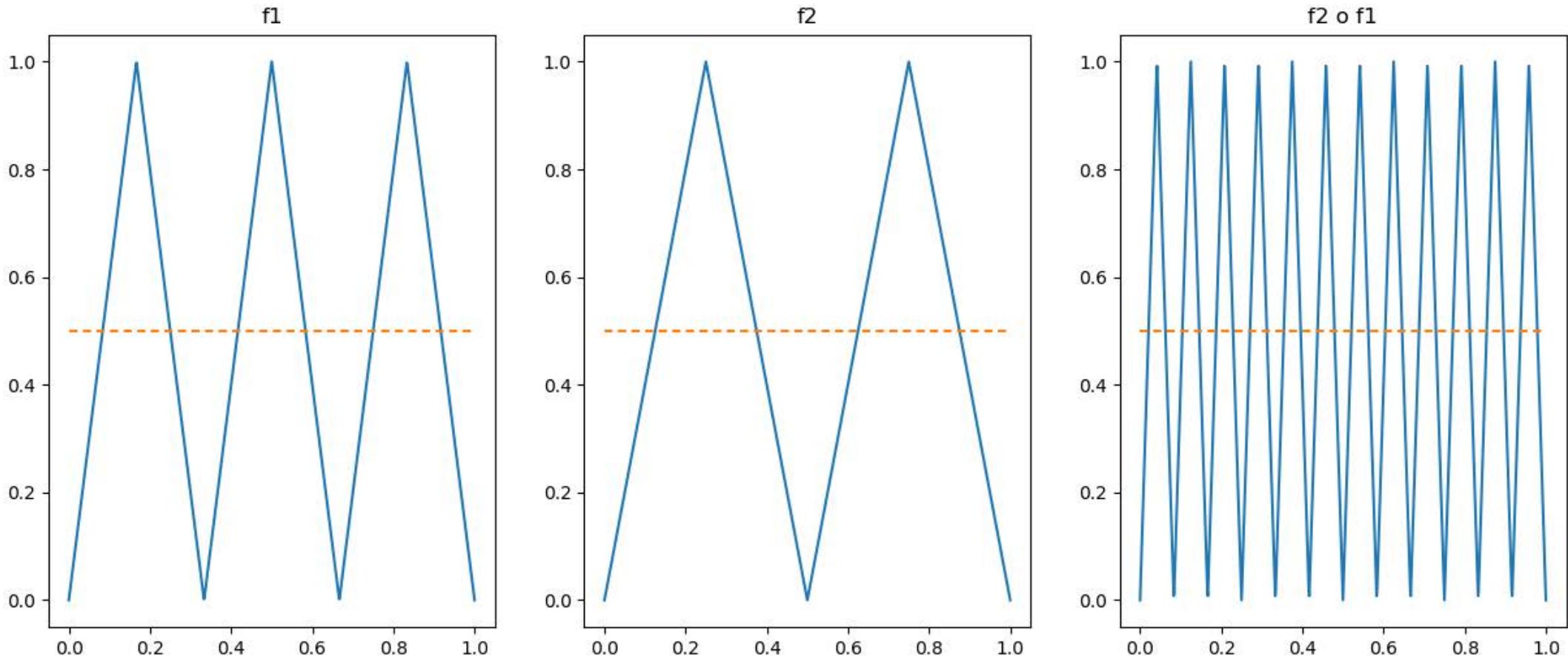
input layer      hidden layer 1    hidden layer 2    hidden layer 3



$$\begin{aligned} \mathbf{h}_1 &= \sigma_1(\mathbf{W}_1 \mathbf{x}_1 + \mathbf{b}_1) \\ \mathbf{h}_2 &= \sigma_2(\mathbf{W}_2 \mathbf{h}_2 + \mathbf{b}_2) \\ &\dots \\ \mathbf{y} &= \sigma_n(\mathbf{W}_n \mathbf{h}_n + \mathbf{b}_n) \end{aligned}$$

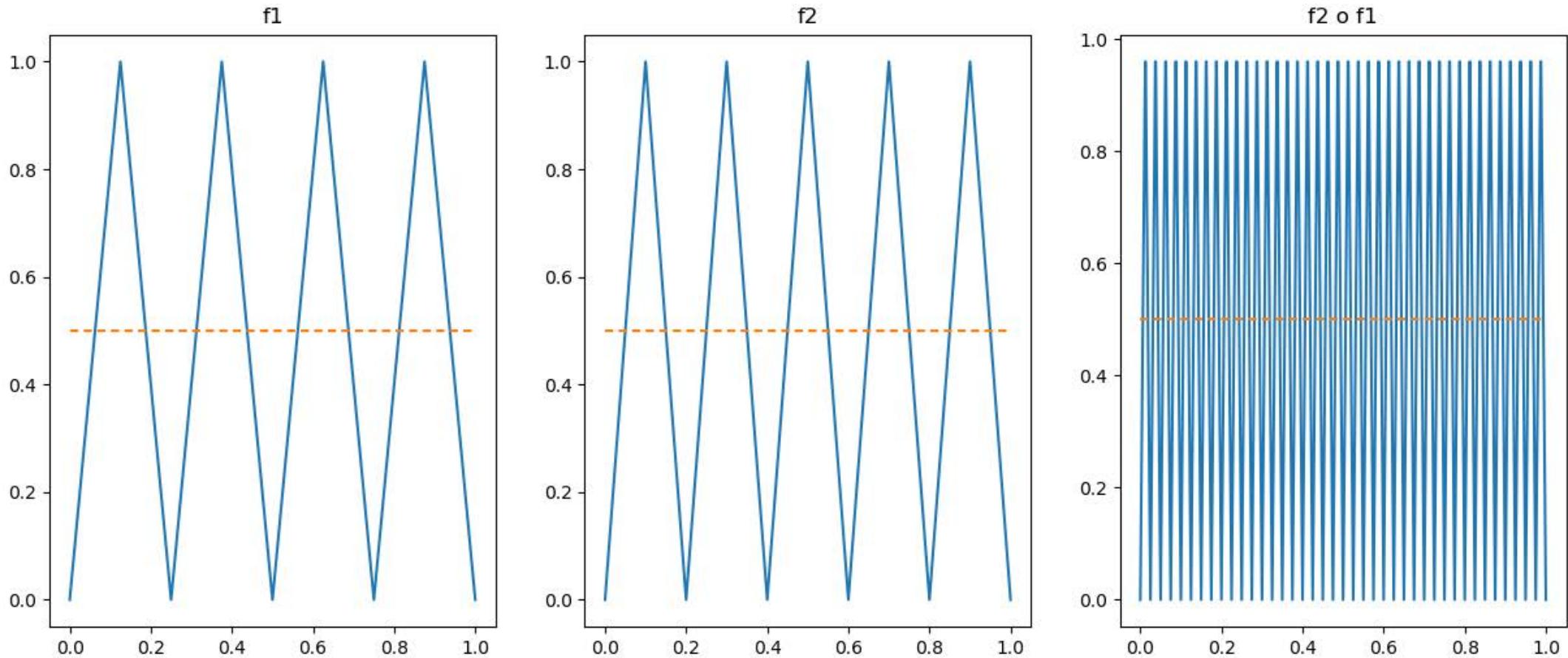
- MLPs can have more than one hidden layer.
- Their descriptive power increases with the number of layers.
- In the case of a 1D signal, it is roughly proportional to  $\prod_n w_n$  where  $w_n$  is the width of layer n.

# The Power of Composition



- $f_1(\cdot)$  has  $n_1=3$  peaks.
- $f_2(\cdot)$  has  $n_2=2$  peaks.
- $f_2(f_1(\cdot))$  has  $2n_1n_2=12$  peaks.

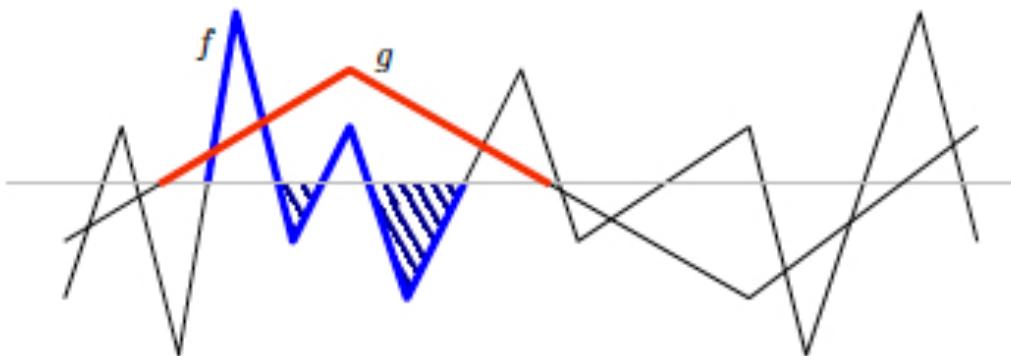
# The Power of Composition



- $f_1(\cdot)$  has  $n_1=4$  peaks.
- $f_2(\cdot)$  has  $n_2=5$  peaks.
- $f_2(f_1(\cdot))$  has  $2n_1n_2=40$  peaks.

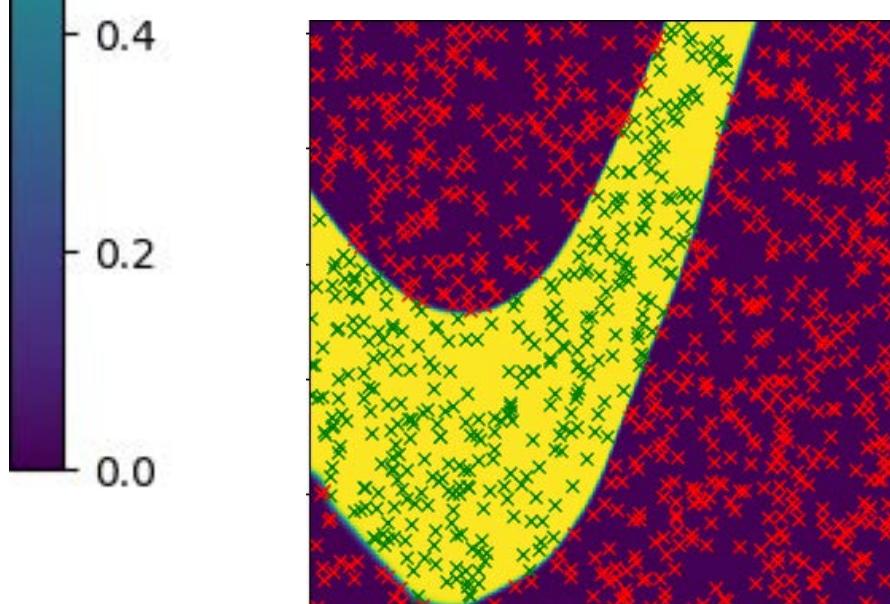
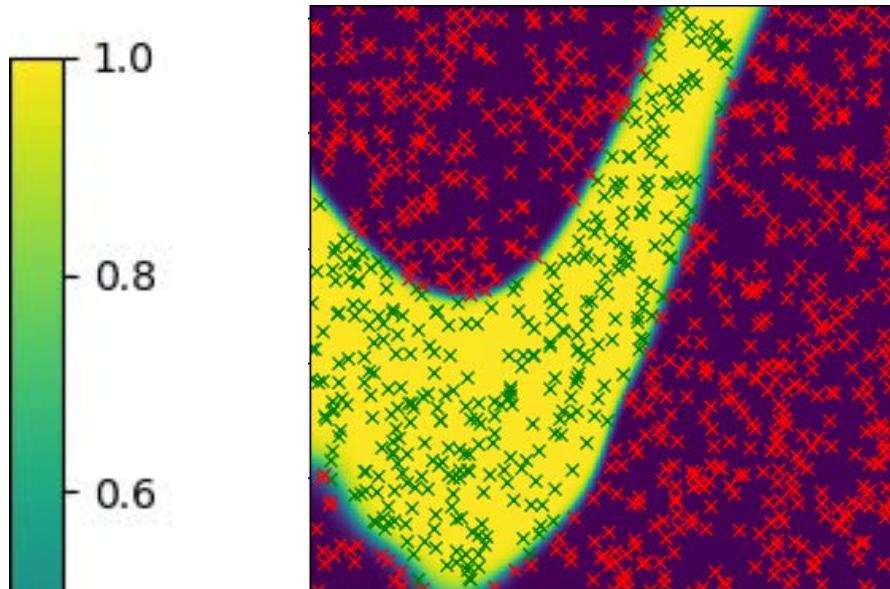
→ Descriptive power is proportional to  $n_1n_2$

# Optional: Proof Sketch

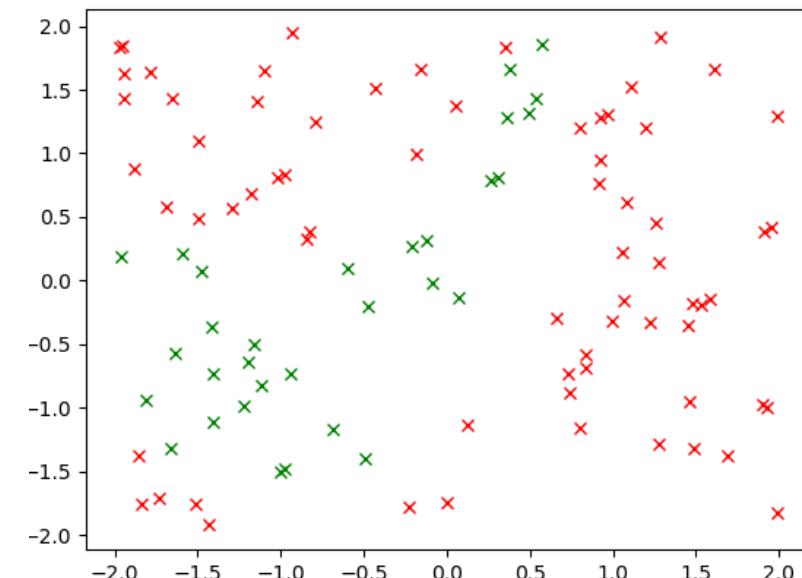
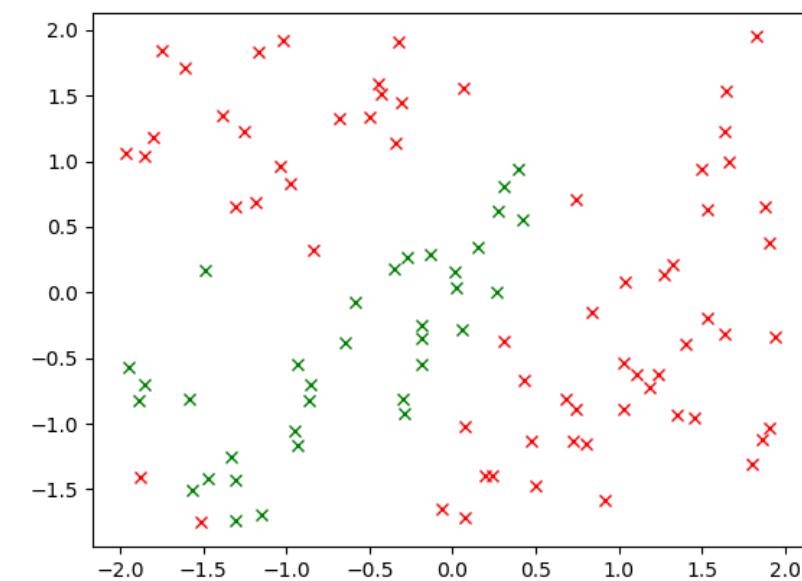


- Functions with few oscillations do not approximate well functions that have many.
- Networks can depict functions that have as many as  $\prod_n^{W_n}$  oscillations, where  $w_n$  is the width of layer n.
  - Functions computed by networks with few layers have few oscillations.
  - Functions computed by deeper networks have many more oscillations.

# Back to Non-Linear Classification

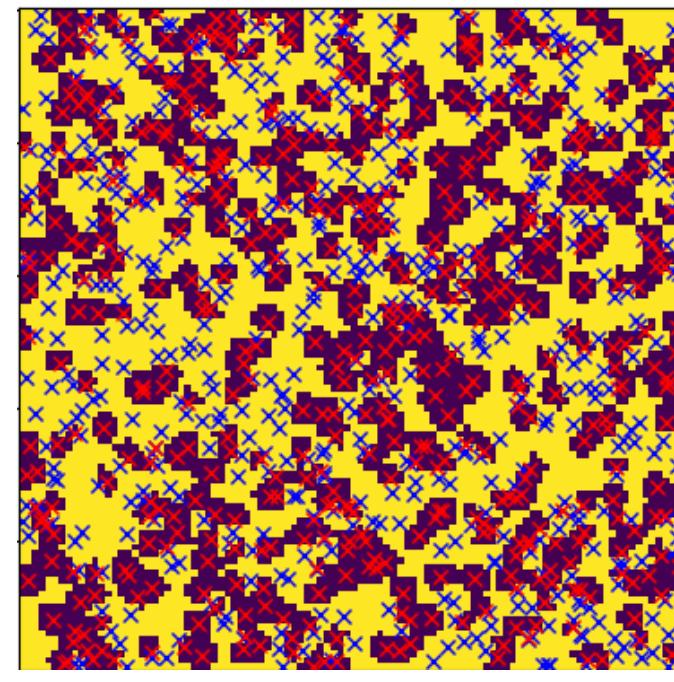
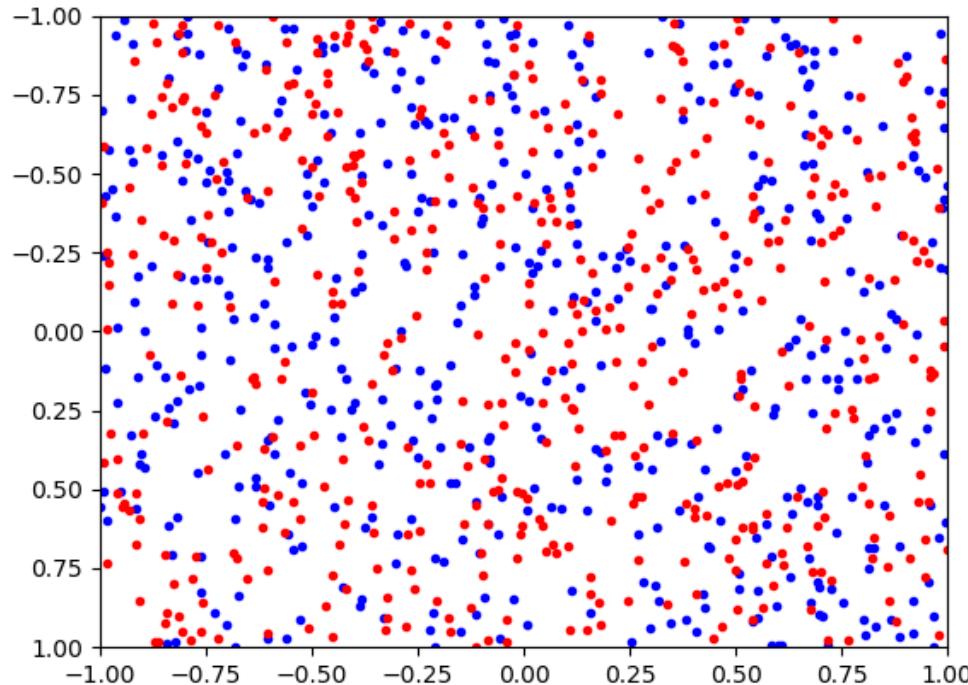


One hidden layer:  $n=100$ , 401 weights.



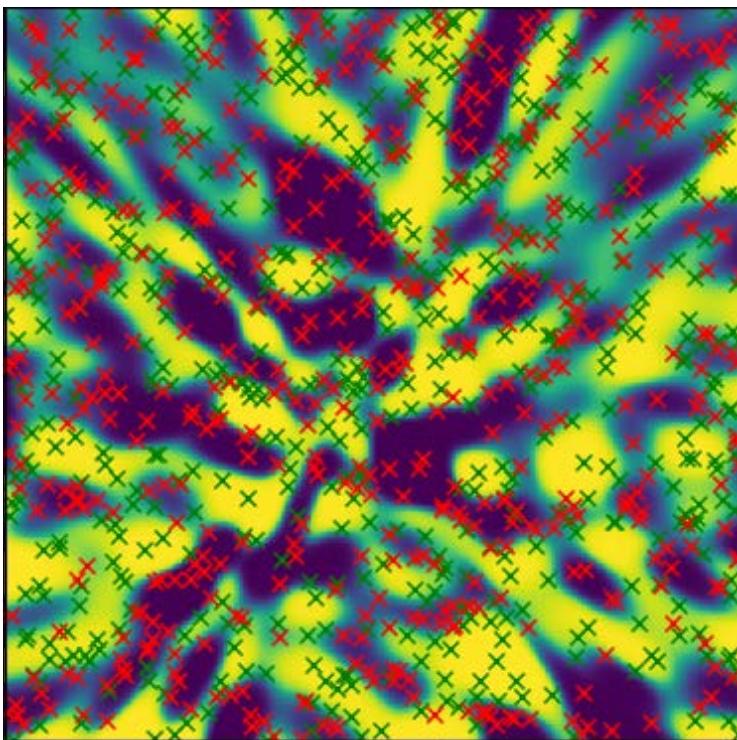
Two hidden layers:  $n_1=10$ ,  $n_2=10$ , 151 weights: Better defined boundaries with far fewer weights.

# Randomly Distributed Points and RBF Kernels

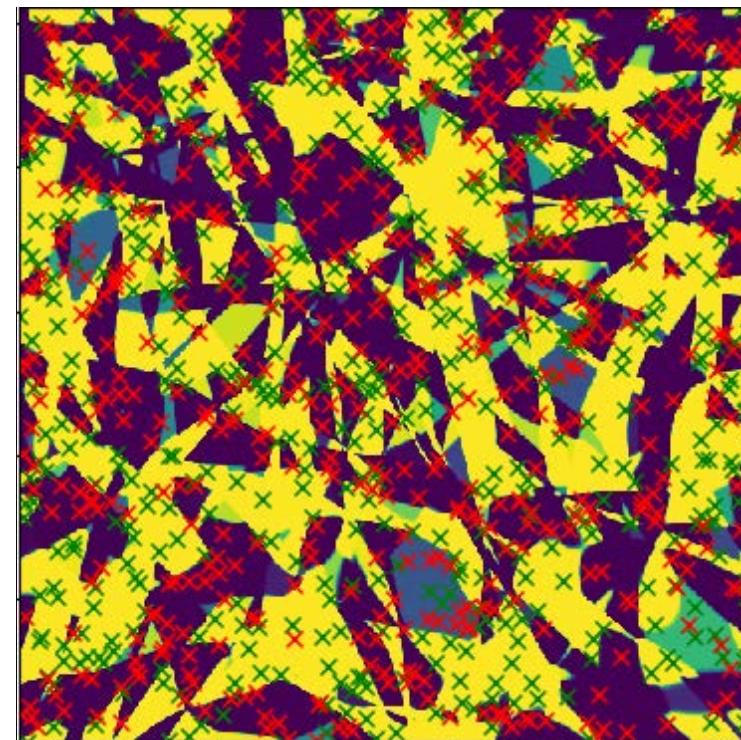


Rbf,  $\gamma = 200$

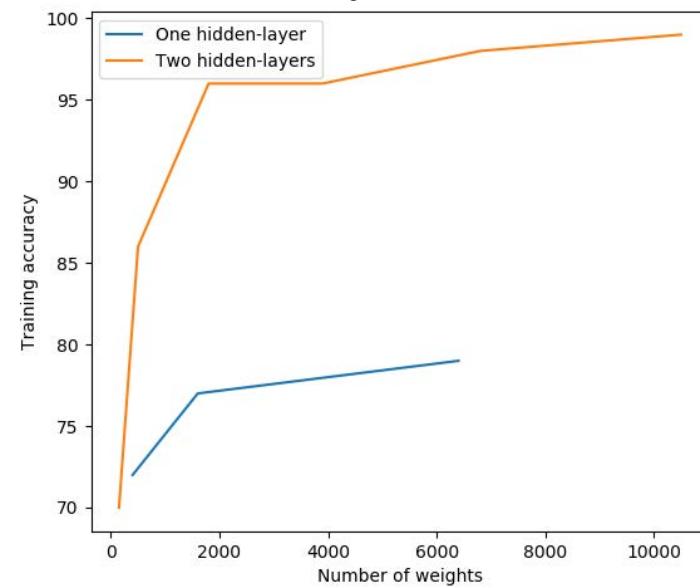
$n=100, 72\%$   
 $n=400, 77\%$   
 $n=1600, 79\%$



One hidden layer



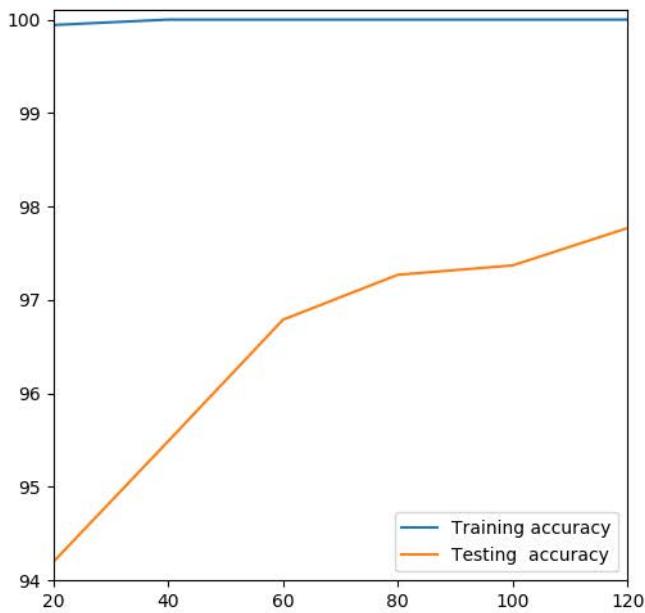
Two hidden layers



Empirical observation: The two layer networks converge faster and better.

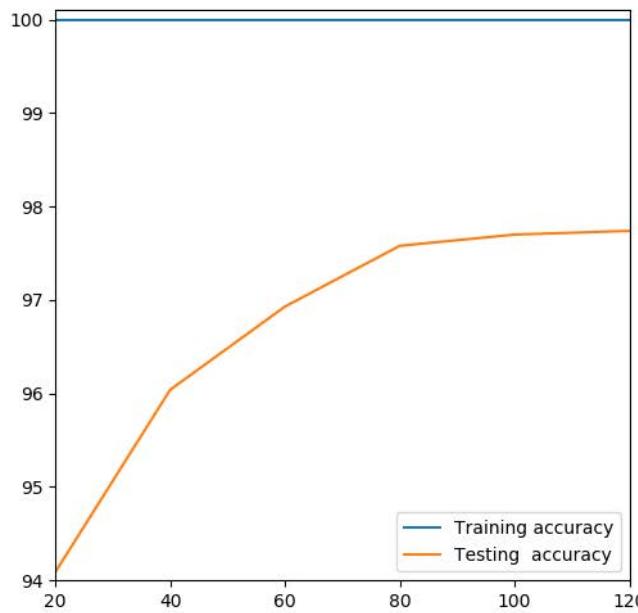
# MNIST Results

$nIn = 784$   
 $nOut = 10$   
 $20 < n1 < 120$



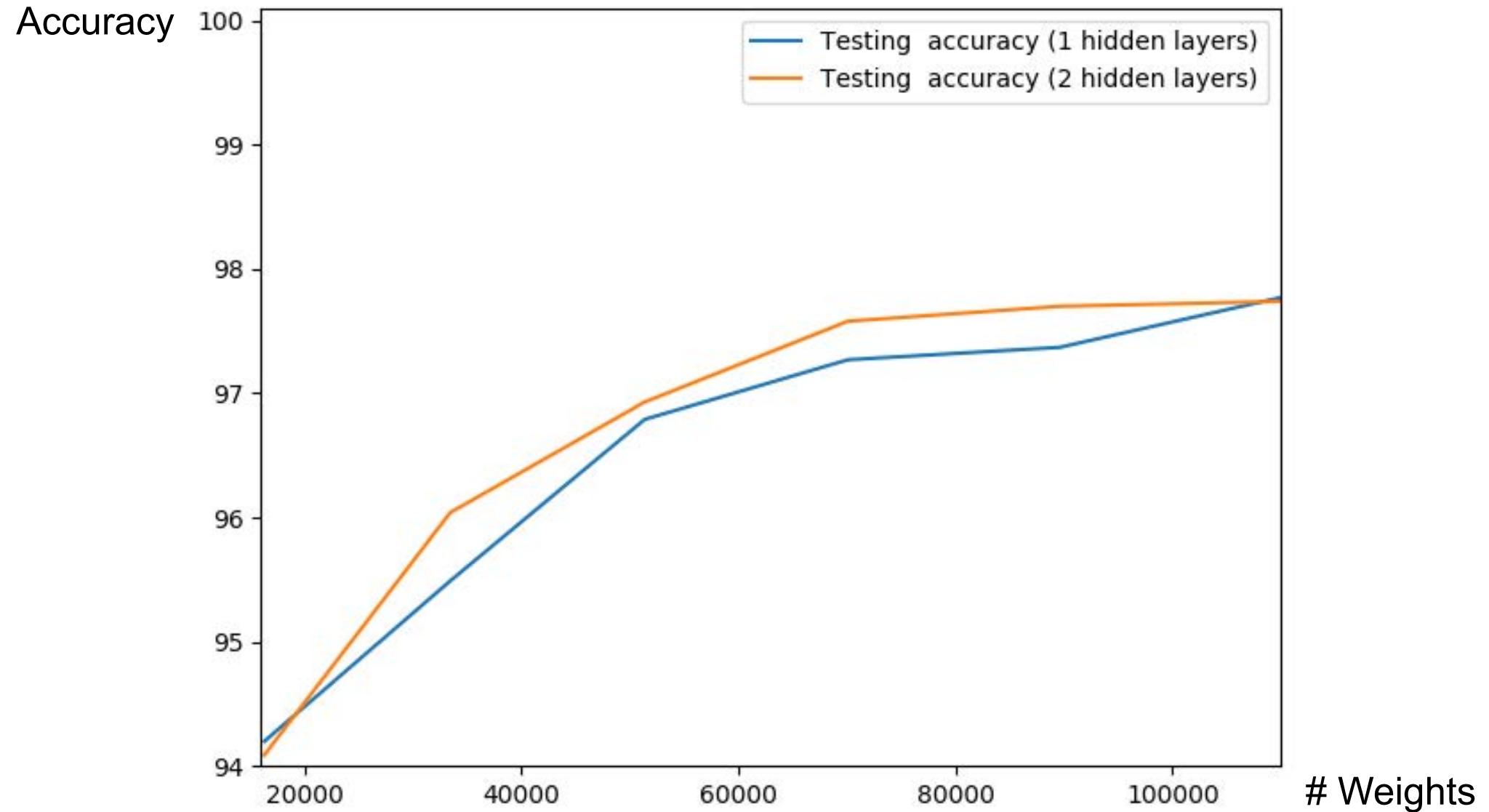
One Layer MLP

$nIn = 784$   
 $nOut = 10$   
 $20 < n1=n2 < 120$



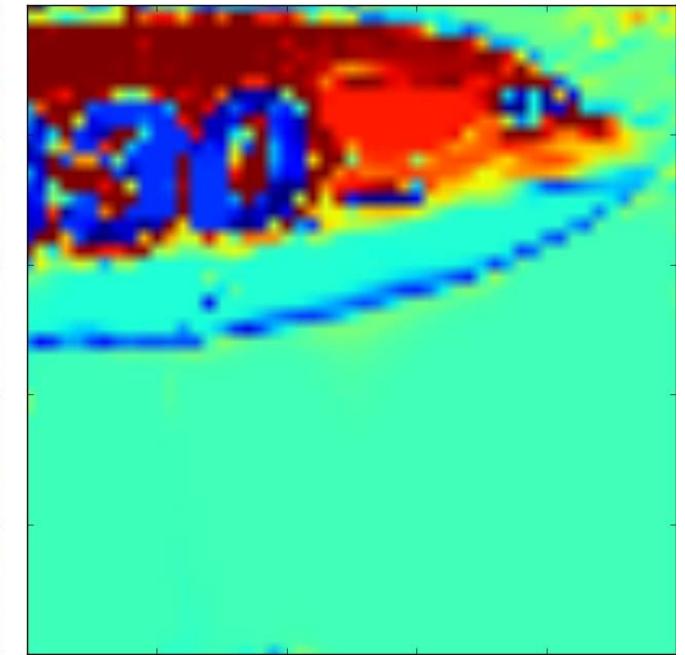
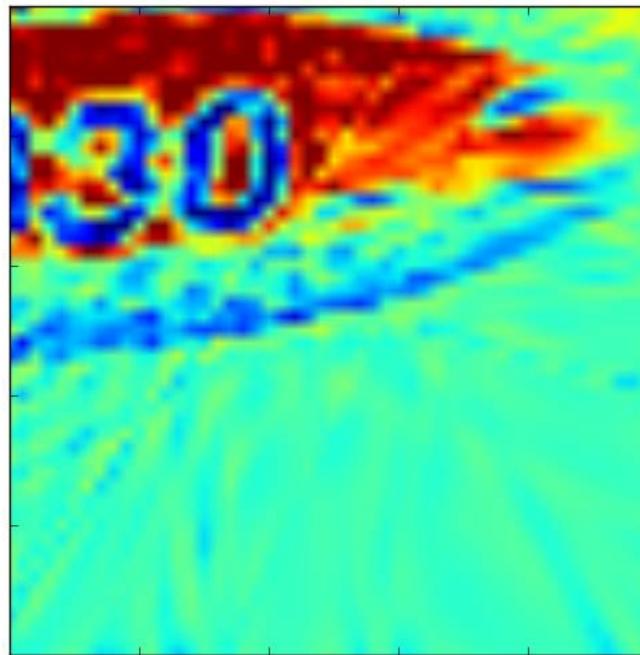
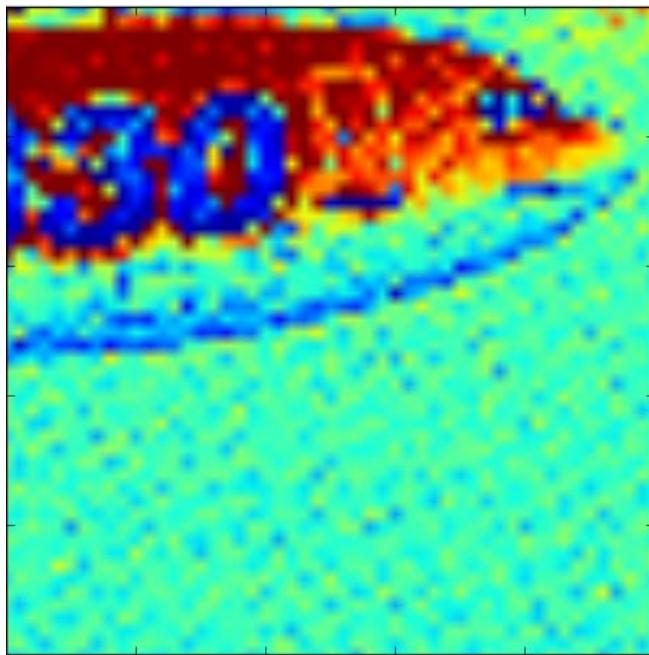
Two-Layer MLP

# Testing Accuracy on MNIST



- Two-layer MLP yield better accuracy with fewer weights initially.
- One-layer MLP Eventually catches up.

# Second Layer for Regression



$$I = f(x, y)$$

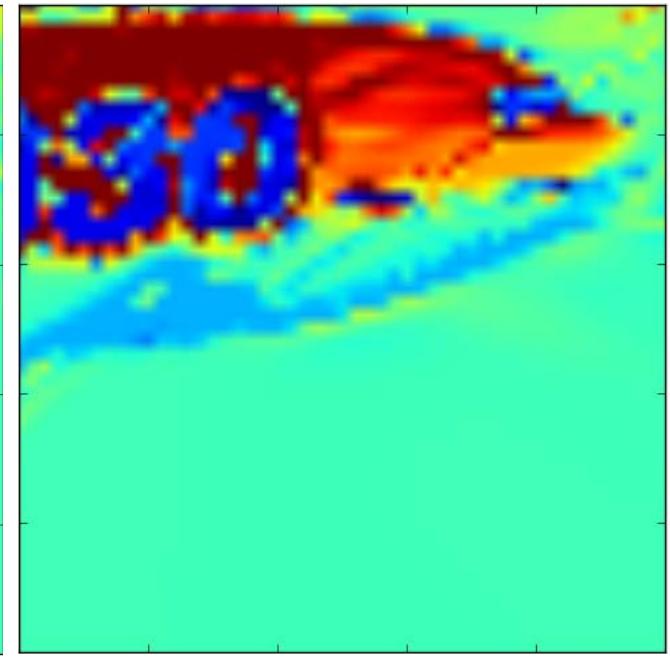
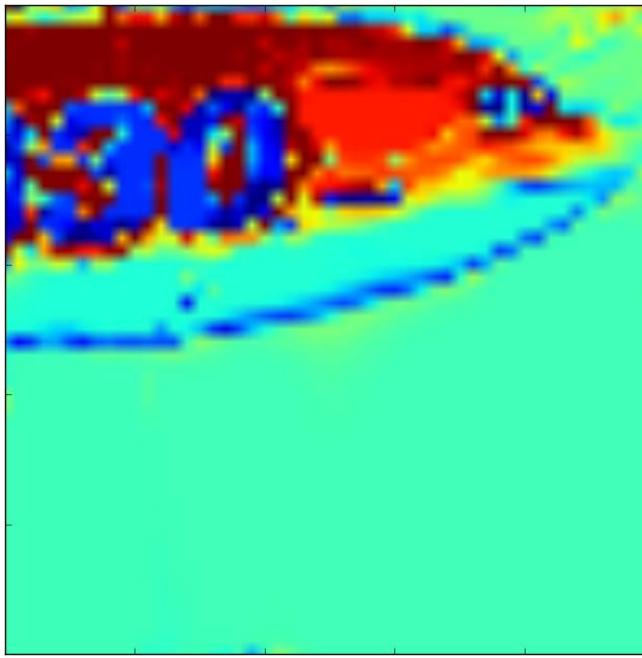
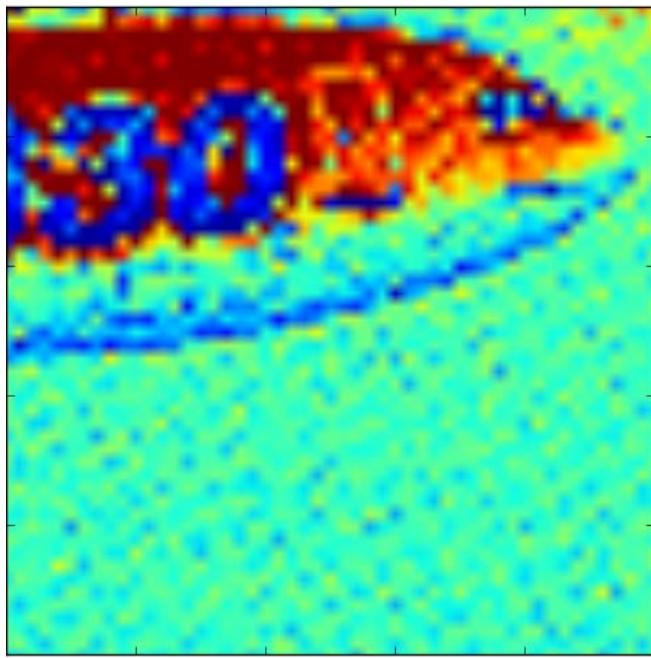
1 Layer: 125 nodes -> loss 2.40e-01

2 Layers: 20 nodes -> loss 8.31e-02

501 weights in both cases

→ The two-layer MLP yields a better approximation with the same number of weights.

# Adding a Third Layer



$$I = f(x, y)$$

2 Layers: 20 nodes -> loss 8.31e-02

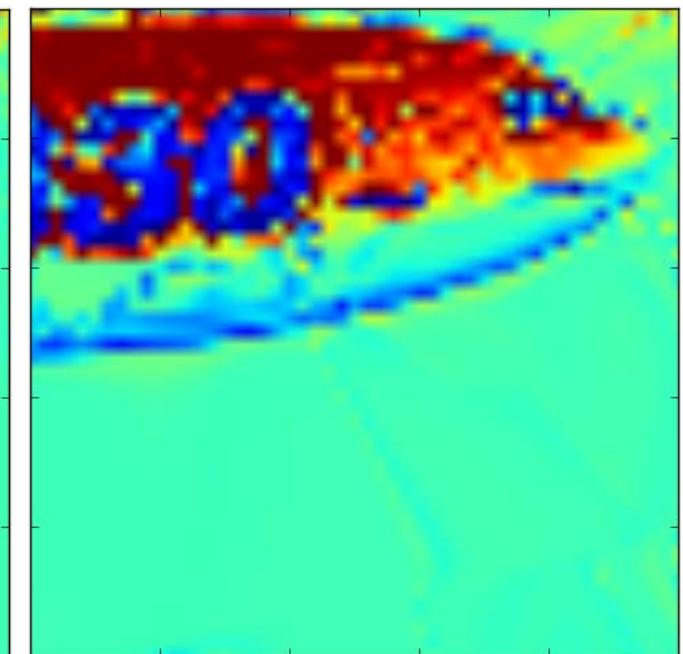
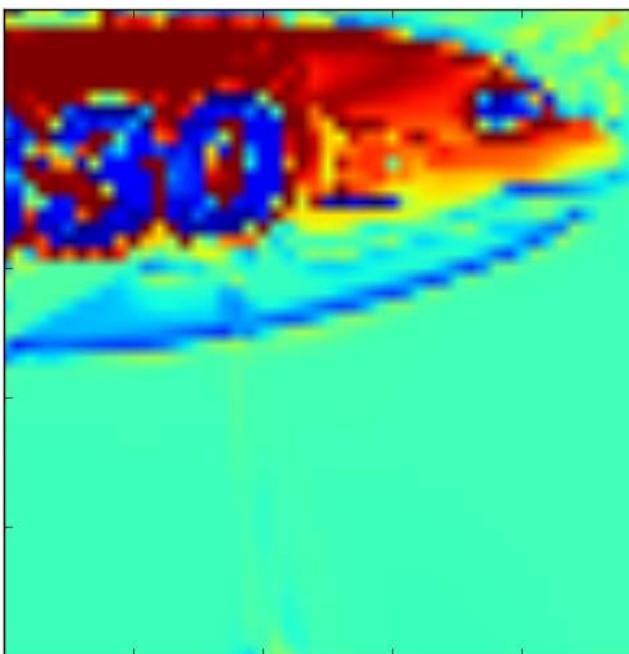
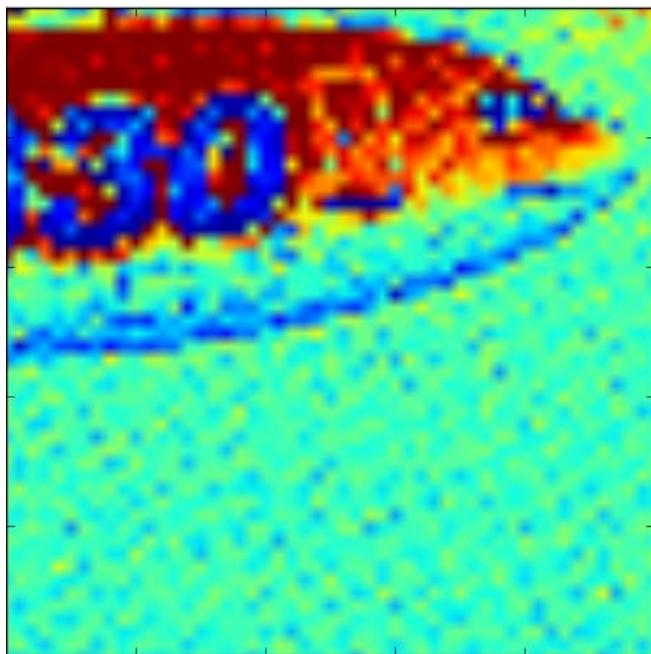
501 weights

3 Layers: 14 nodes -> loss 7.55e-02

477 weights

→ The three-layer MLP does even better but the difference is less striking. Diminishing returns?

# Adding a Third Layer



$$I = f(x, y)$$

3 Layers: 15 nodes -> loss 5.93e-02

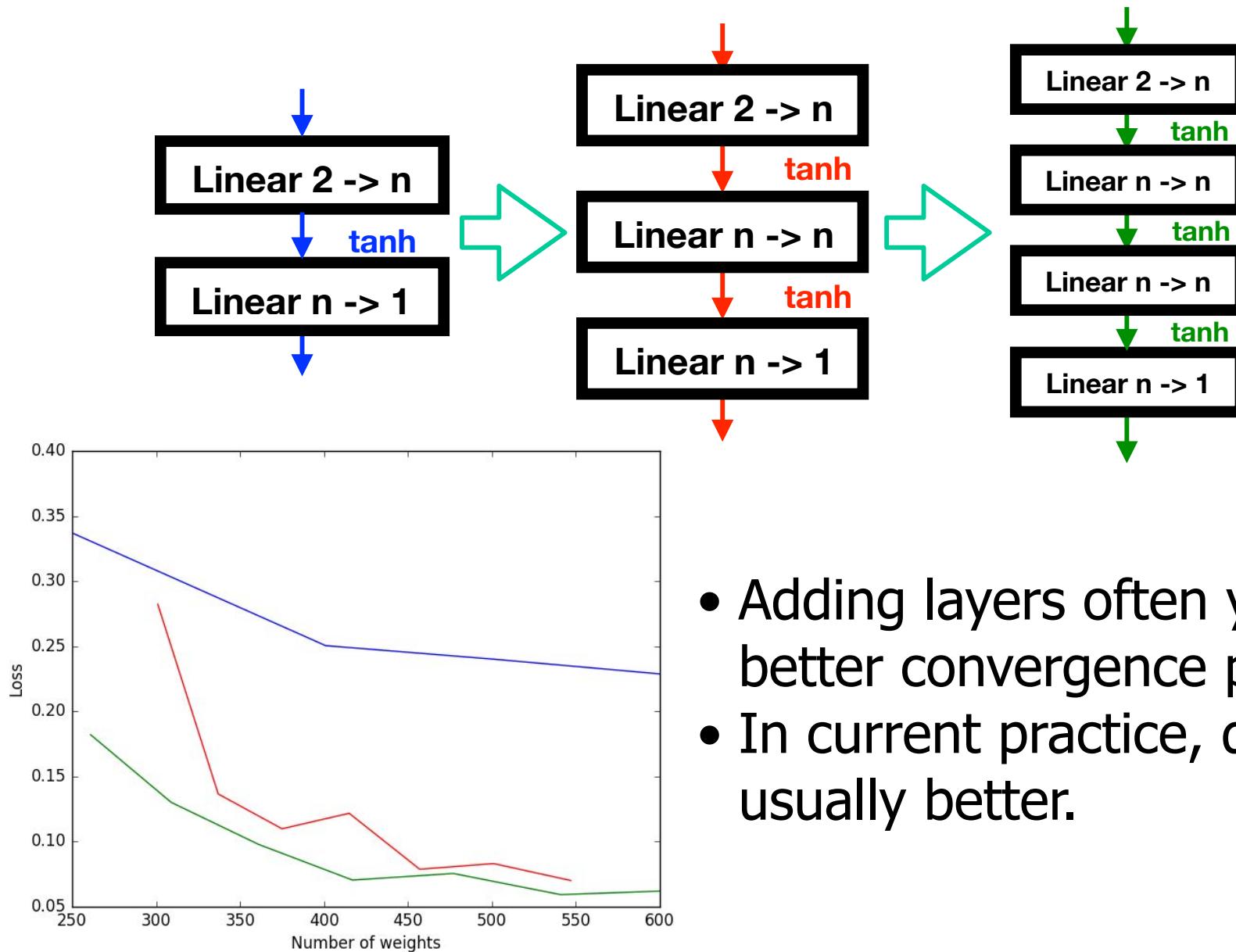
541 weights

3 Layers: 19 nodes -> loss 4.38e-02

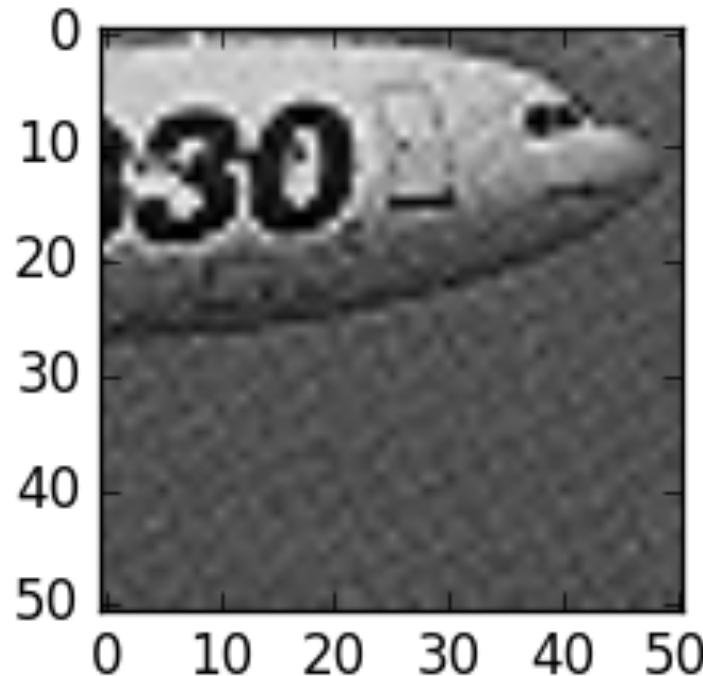
837 weights

→ The three-layer MLP does even better but the difference is less striking. Diminishing returns?

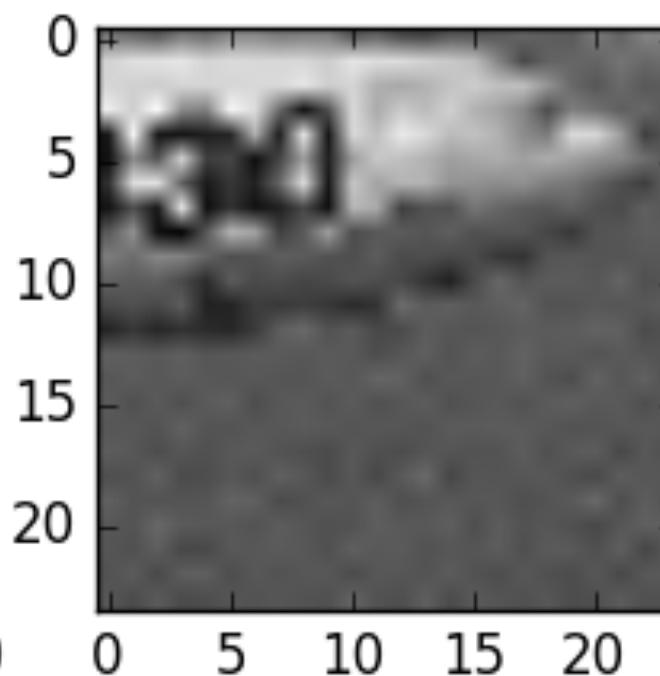
# Multi Layer Perceptrons



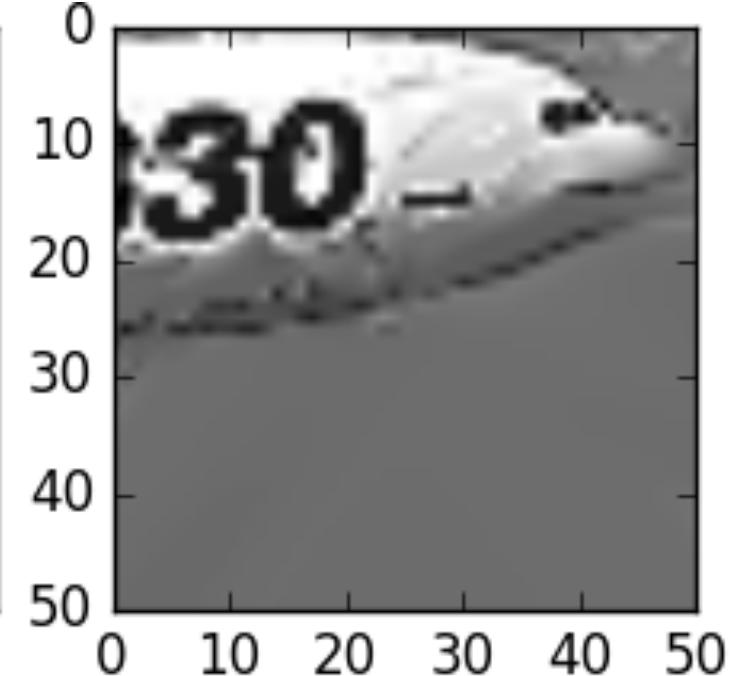
# Other Ways to Interpolate



Original 51x51 image:  
2601 gray level values.

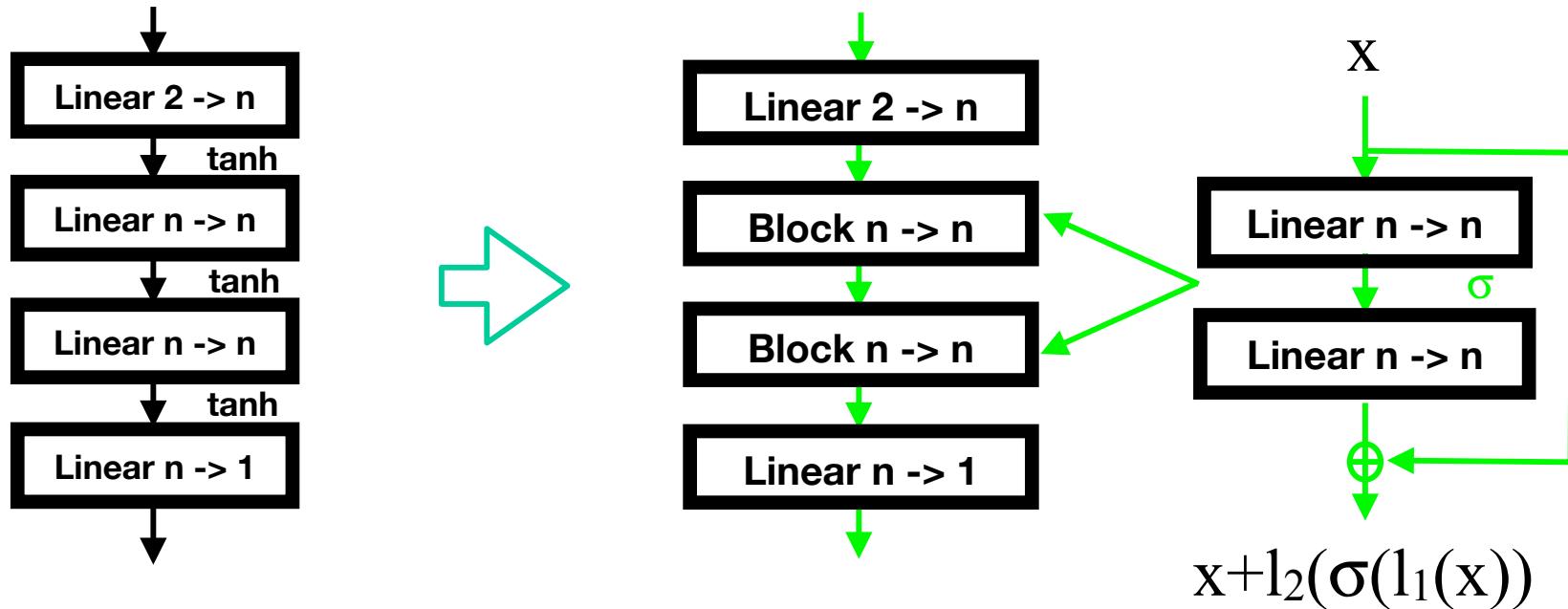


Scaled 24x24 image:  
576 gray level values.



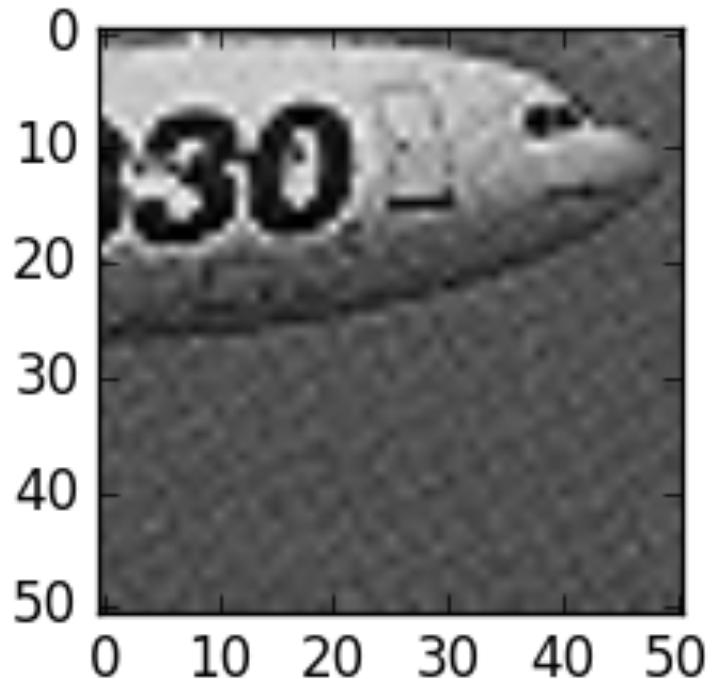
MLP 10/20/10 Interpolation:  
471 weights.

# MLP to ResNet

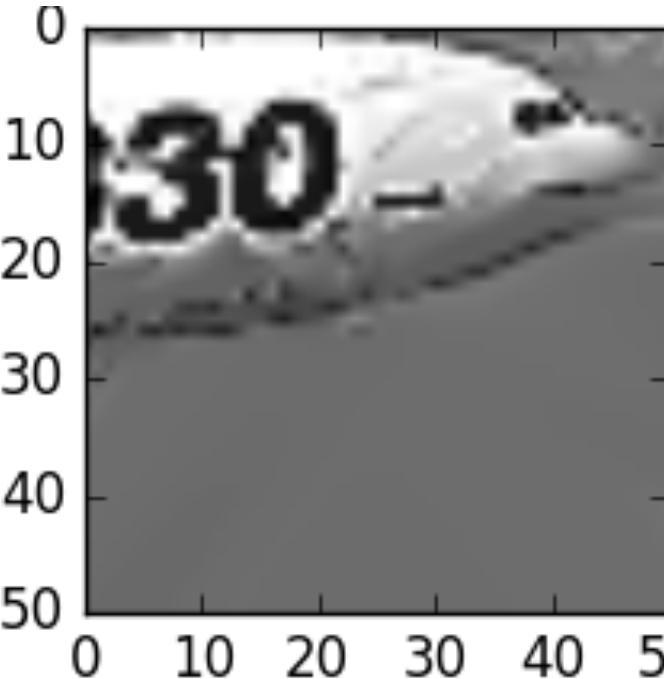


Further improvements in the convergence properties have been obtained by adding a bypass, which allows the final layers to only compute residuals.

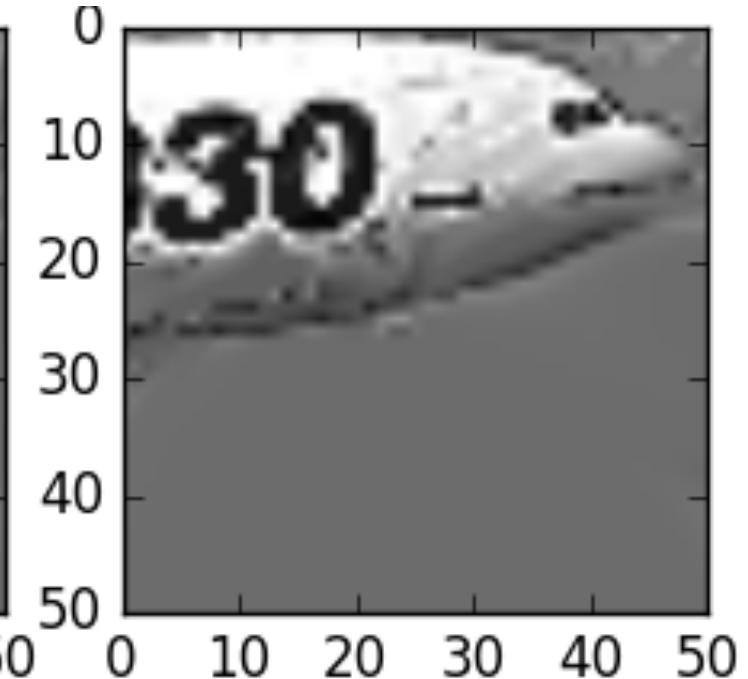
# Adding a ResNet Layer



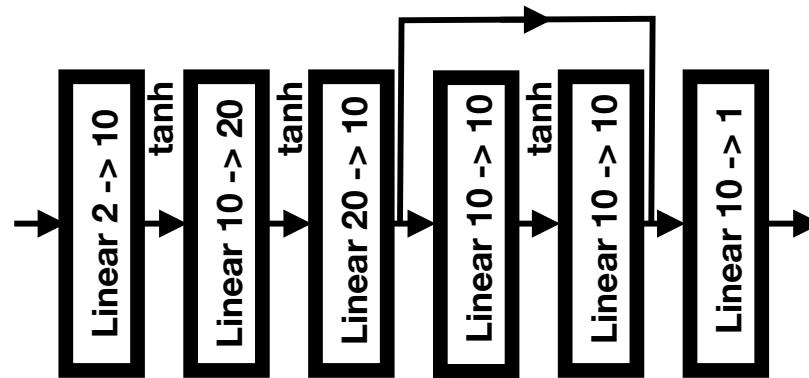
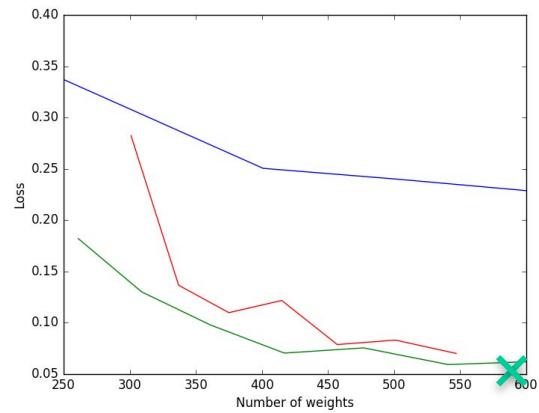
Original 51x51 image:  
2601 gray level values.



MLP 10/20/10 Interpolation:  
471 weights, loss 6.43e-02.

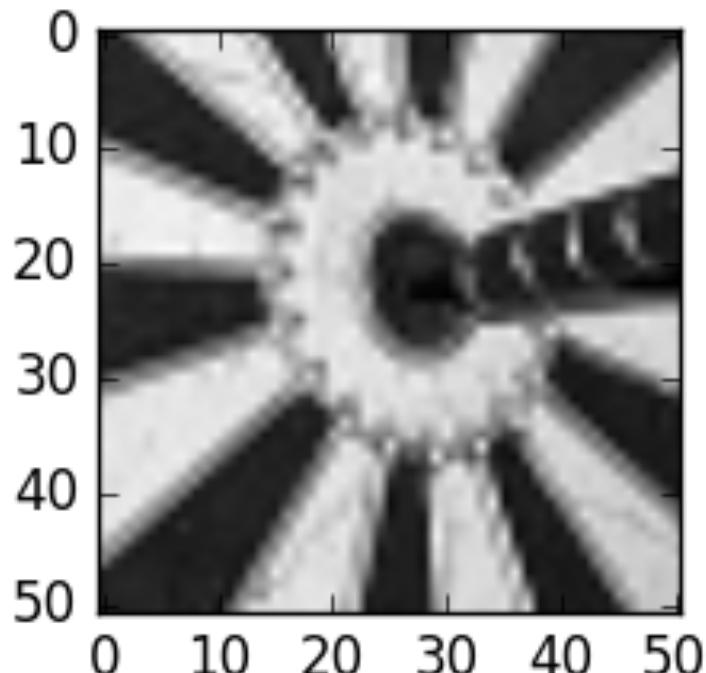


Resnet 10/20/10/10 Interpolation:  
581 weights, loss 5.30e-2.

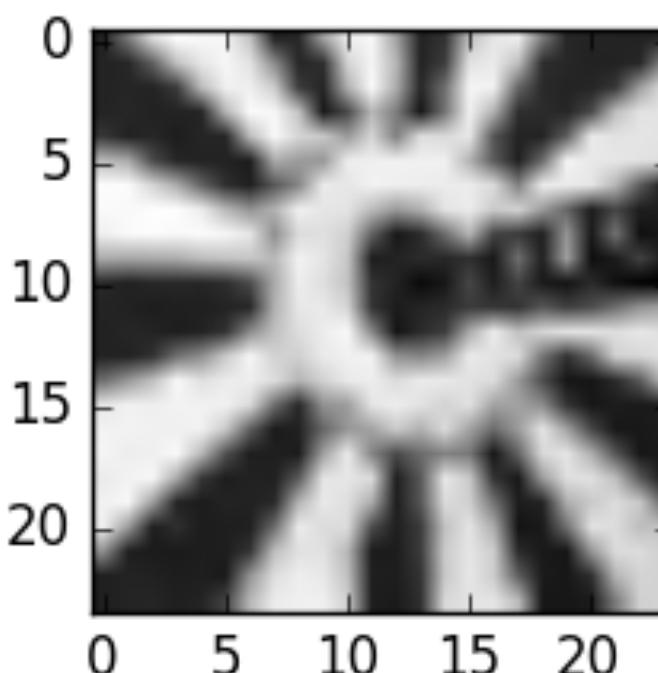


→ Adding a ResNet layer yields a further small improvement.

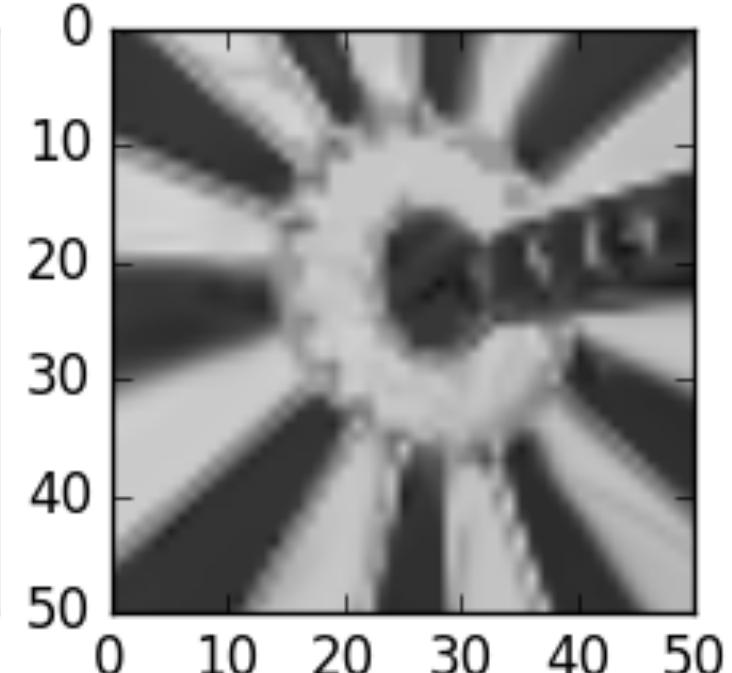
# Adding a ResNet Layer



Original 51x51 image:  
2601 gray level values.



MLP 10/20/10 Interpolation:  
471 weights, loss 1.95e-02.

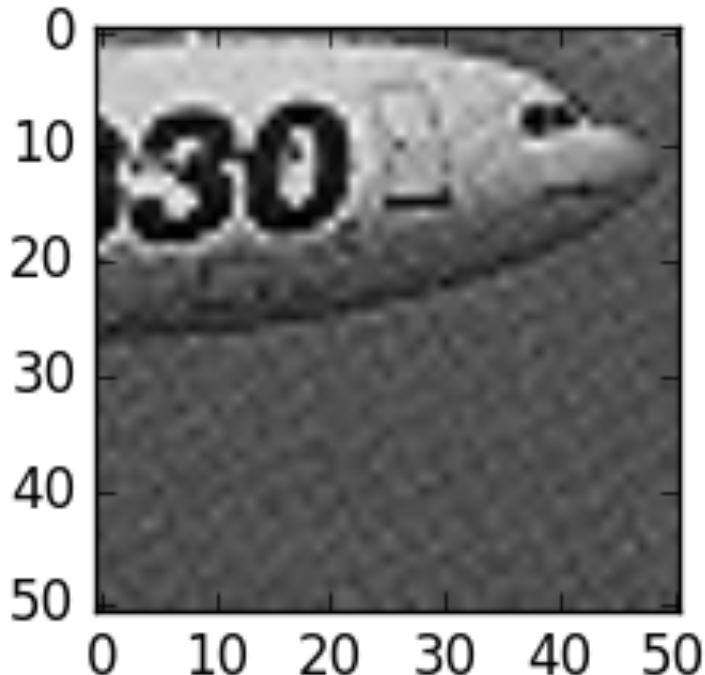


Resnet 10/20/10/10 Interpolation:  
581 weights, loss 1.36e-2.

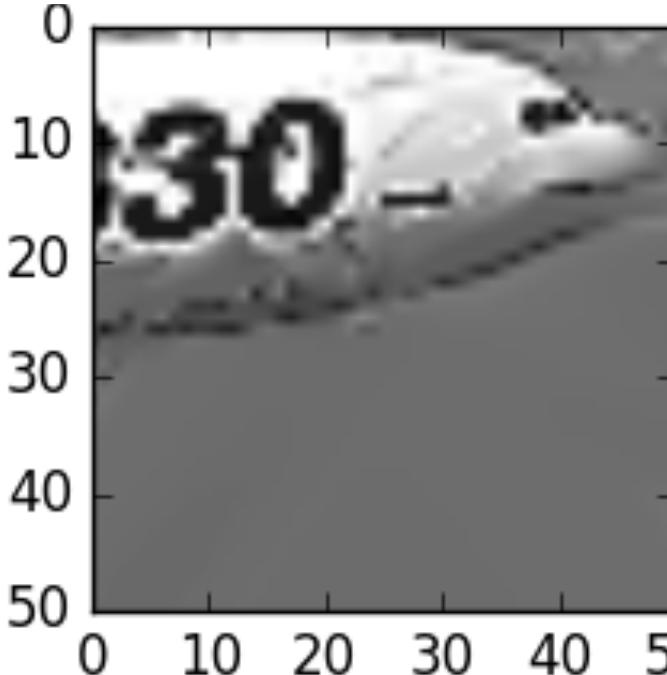
- Relatively small improvement in **this** case.
- We will see a different behavior for large networks.
- The problem is probably too small.

→ Networks can behave very differently for small and large problems!

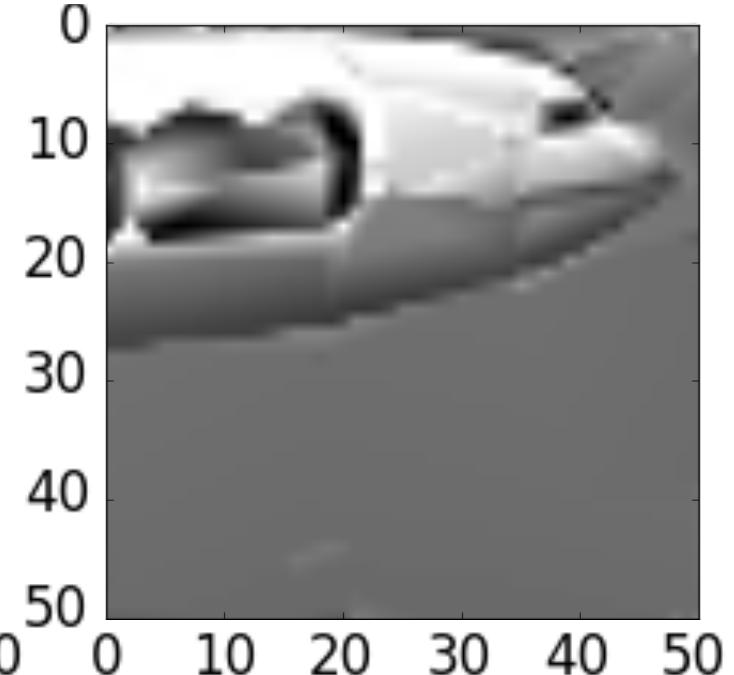
# Tanh vs ReLU



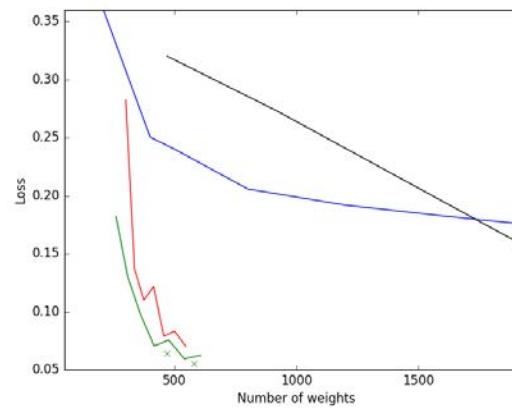
Original 51x51 image:  
2601 gray level values.



MLP 10/20/10 Interpolation:  
Tanh, loss 6.43e-02.



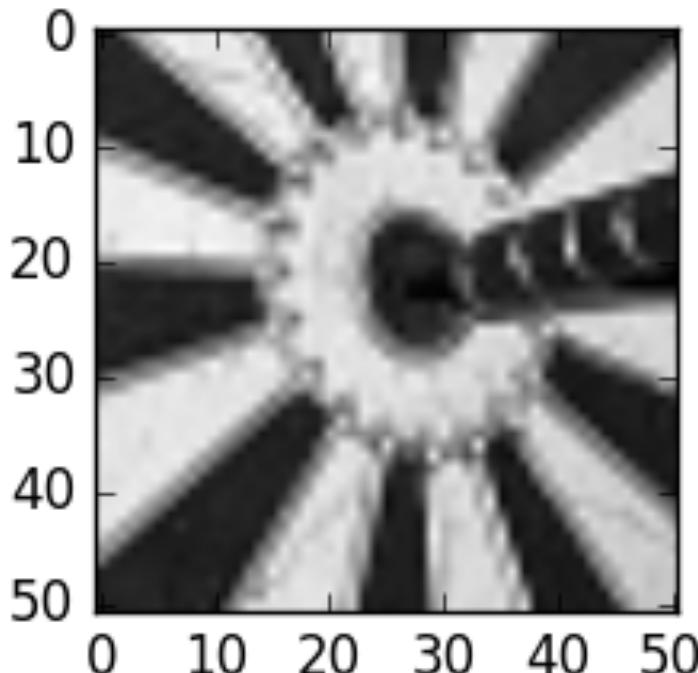
MLP 10/20/10 Interpolation:  
ReLU, loss 3.07e-1.



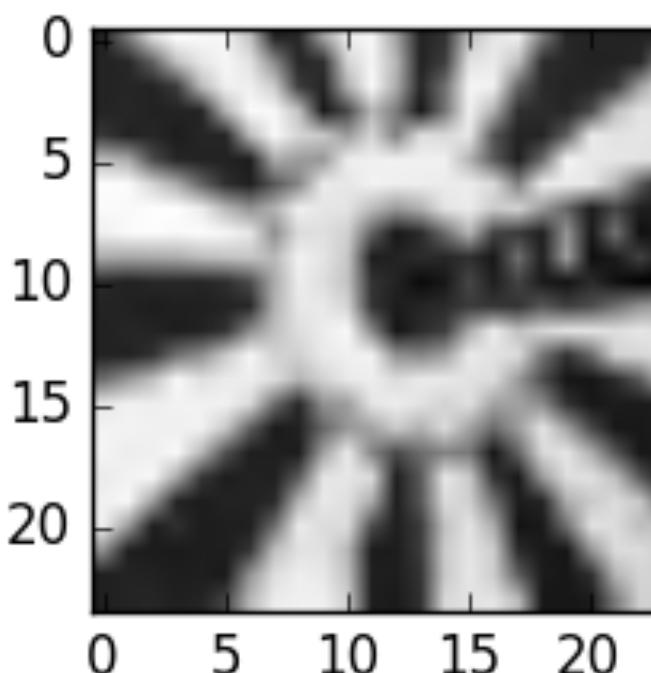
- Blue line: Tanh, 1 layers
- Red line: Tanh, 2 layers
- Green line: Tanh, 3 layers
- Black line: ReLU, 3 layers
- Green 'x': Tanh, 4 layers

→ Tanh works better than ReLU in this case.

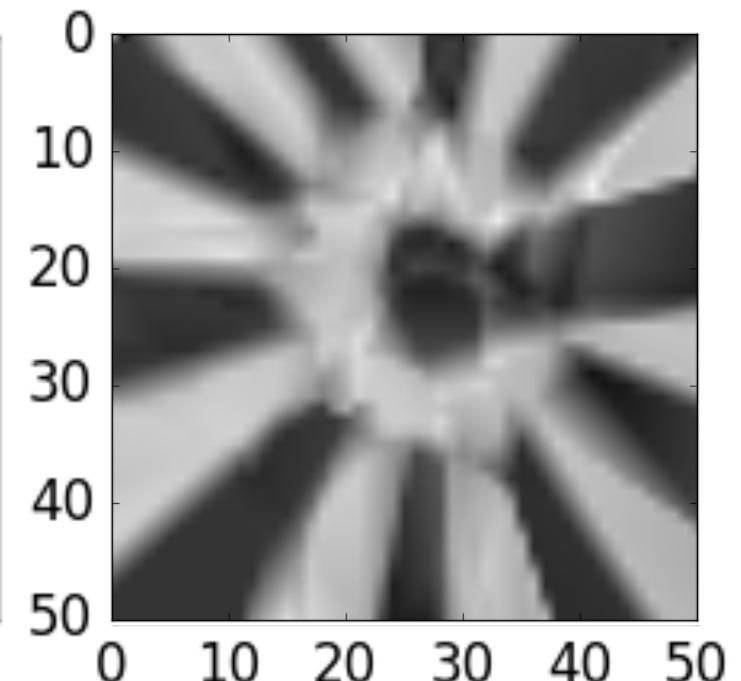
# Tanh vs ReLu



Original 51x51 image:  
2601 gray level values.



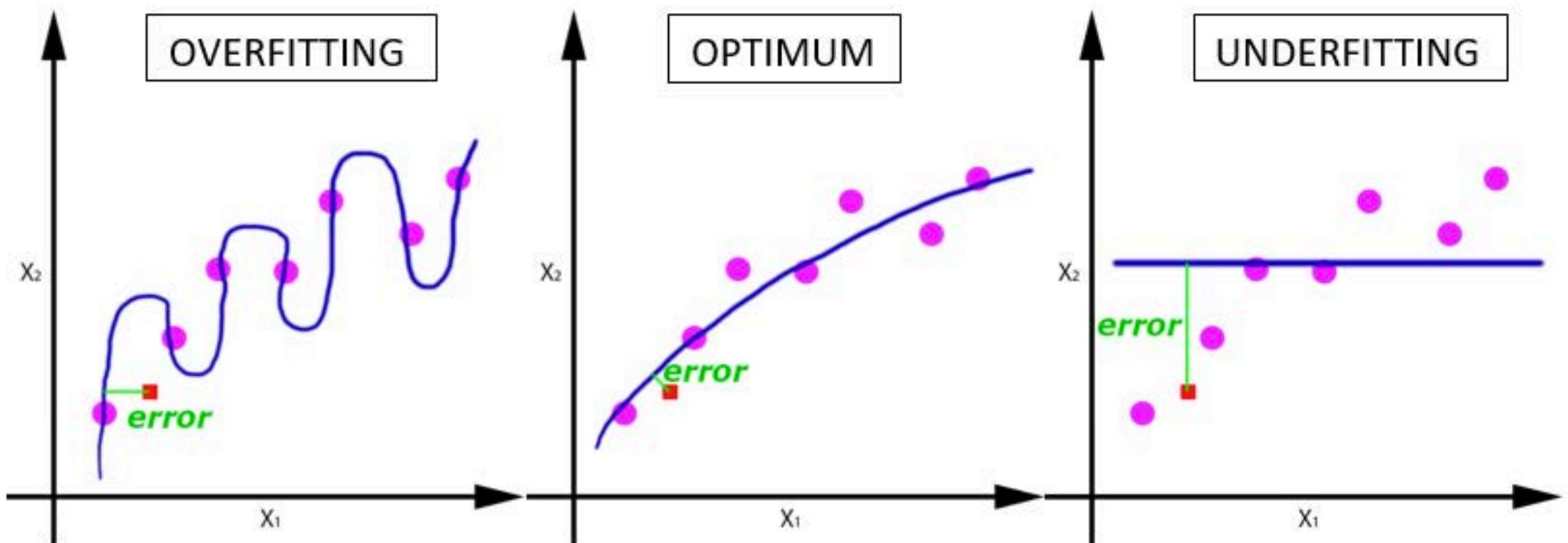
MLP 10/20/10 Interpolation:  
tanh, loss 1.95e-02.



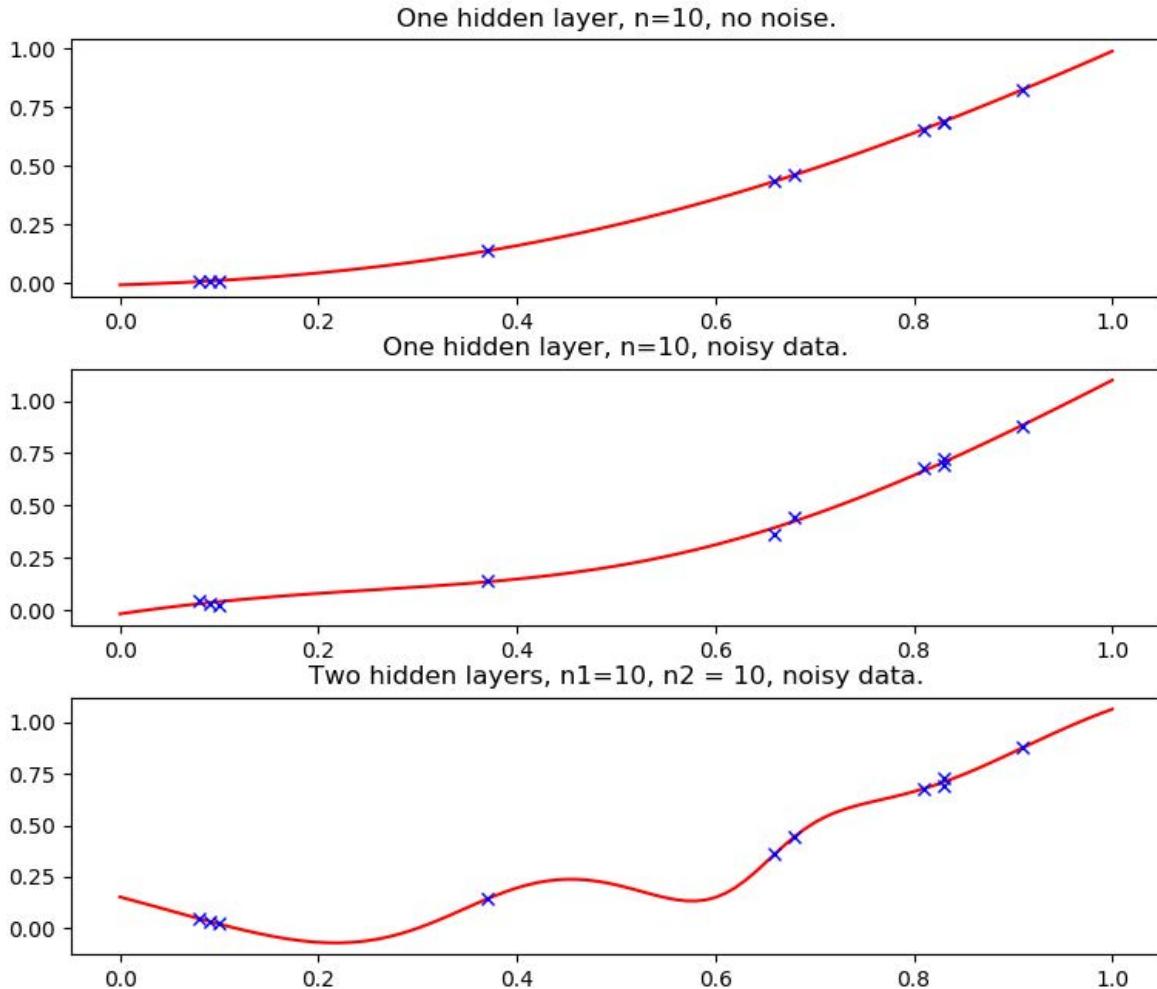
MLP 10/20/10 Interpolation:  
relu, loss 7.21e-2.

- Tanh works better than ReLU in **this** case.
- ReLU is widely credited with eliminating the vanishing gradient problem in large networks.  
→ There is no substitute for experimentation!

# Overfitting vs Underfitting



# Overfitting



- Multi-layer perceptrons have great descriptive power and can approximate almost any reasonable function.
- But they can also overfit.
- There is no truly automated way to set the number and width of the layers.

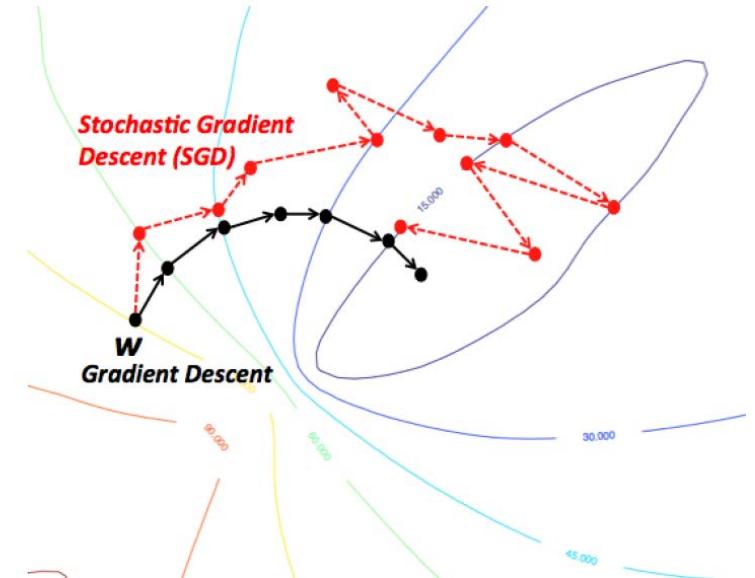
# Reminder: Stochastic Gradient Descent

$$E(\mathbf{w}) = \sum_{n=1}^N E_n(\mathbf{w})$$

Gradient descent:  $\mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \eta \sum_{n=1}^N \nabla E_n(\mathbf{w}^\tau)$ .

Stochastic descent:  $\mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \eta \sum_{n \in B^\tau} \nabla E_n(\mathbf{w}^\tau)$ ,

where  $B^\tau$  represents a different randomly chosen set of indices at each iteration, also known as a mini-batch.

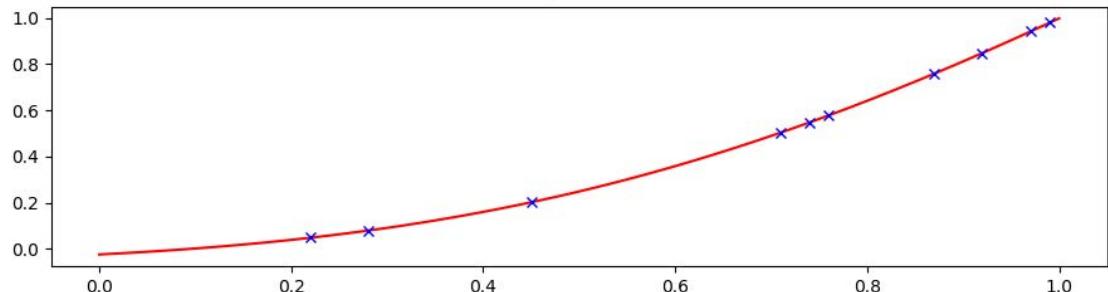


## Randomly choosing batches

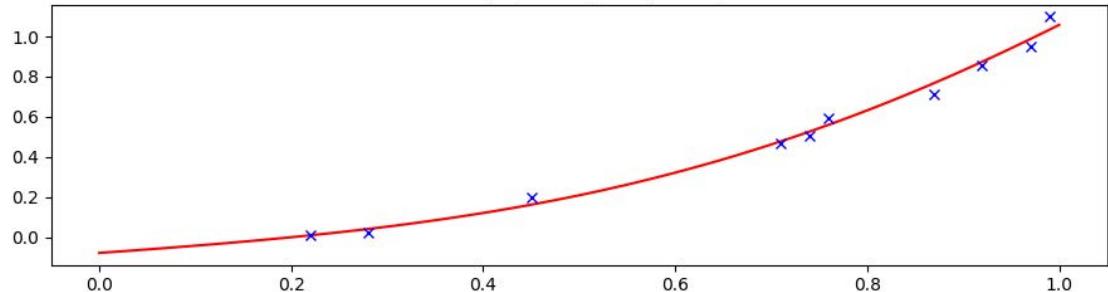
- helps reduce the chances of falling into local minima,
- makes the computation possible on GPUs even when dealing with LARGE databases.

# Using Mini-Batches

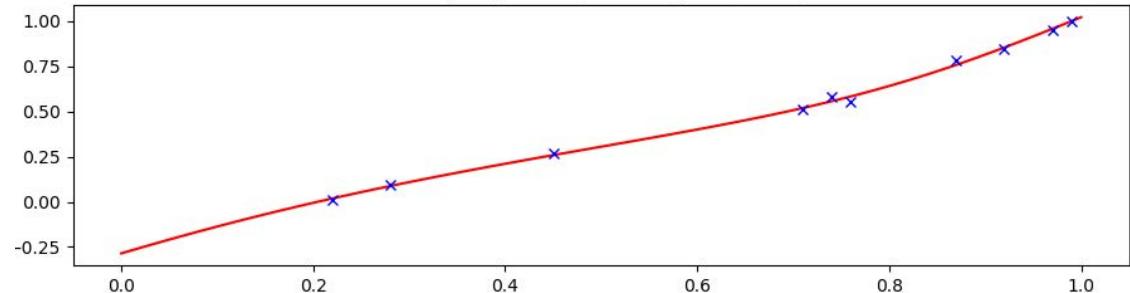
One hidden layer,  $n=10$ , no noise.



One hidden layer,  $n=10$ , noisy data, batches.

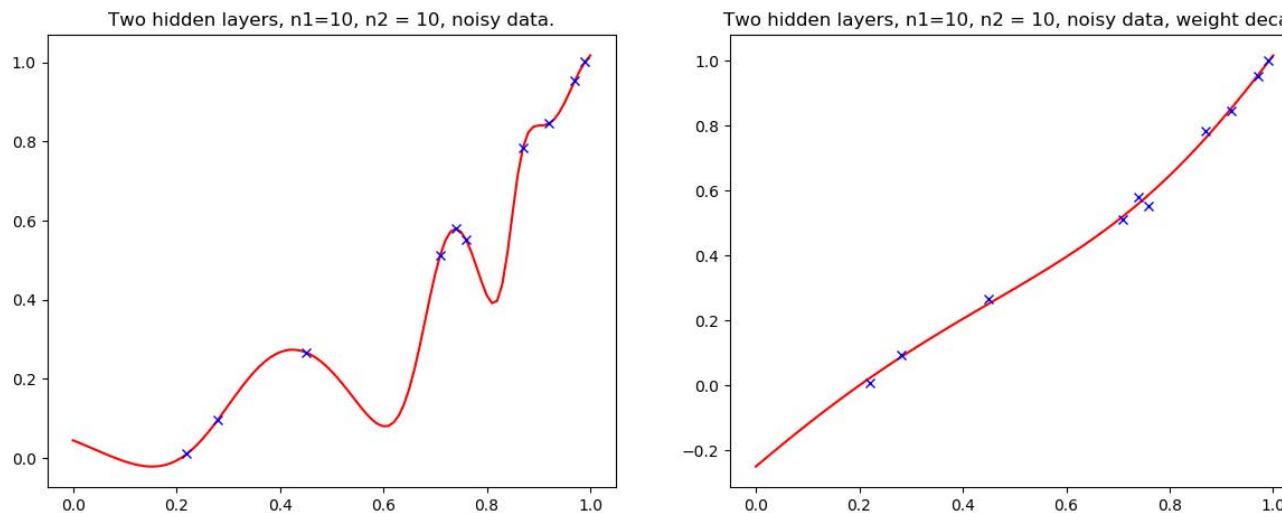
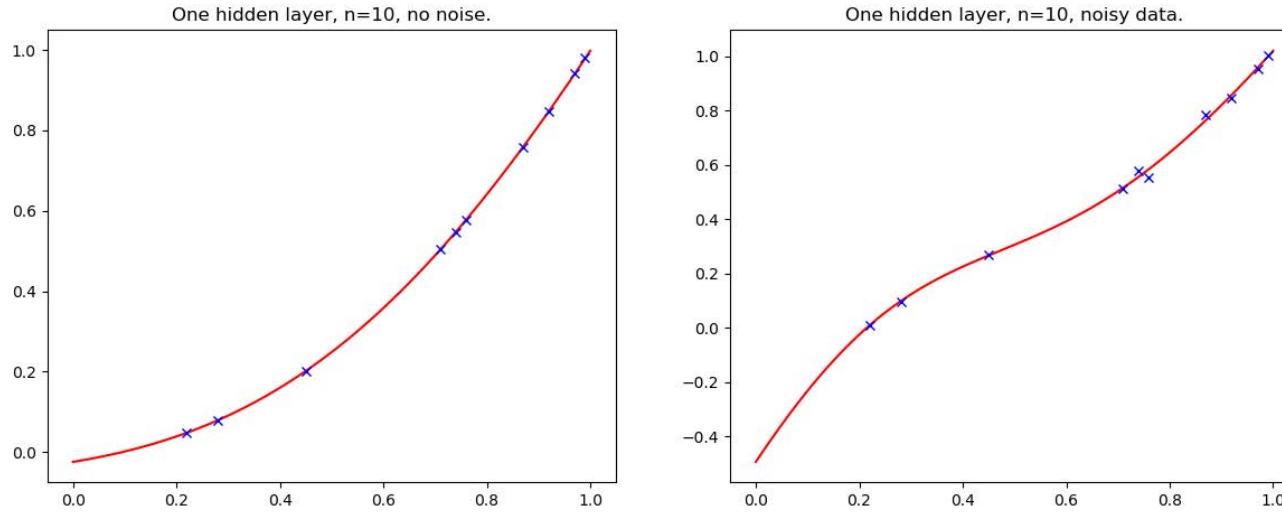
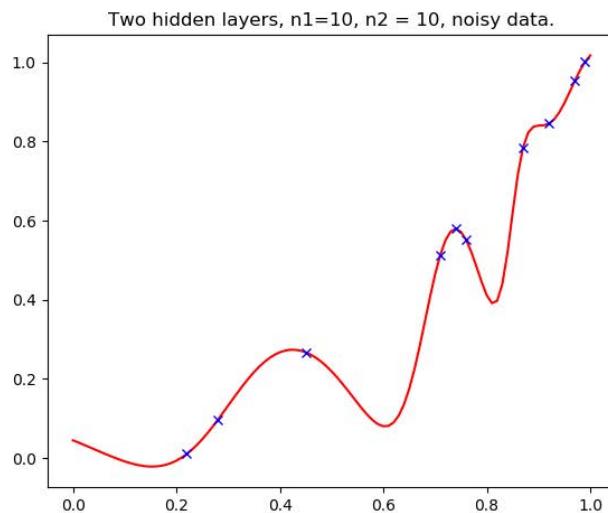
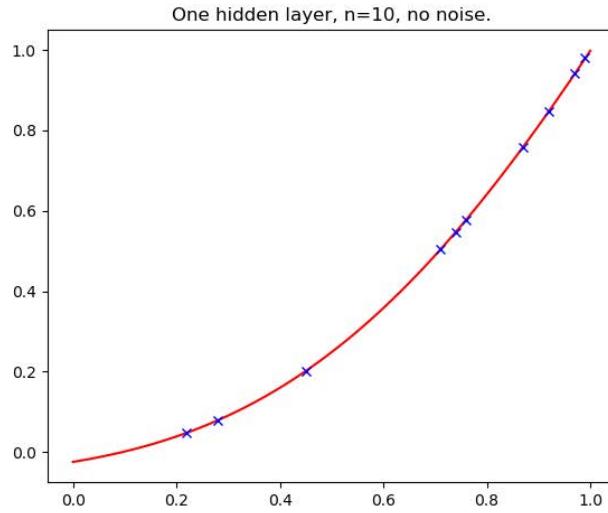


Two hidden layers,  $n_1=10$ ,  $n_2 = 10$ , noisy data, batches.



- The element of randomness prevents overfitting.
- No principled way to set the size of the mini-batches to achieve this result.

# Weight Decay

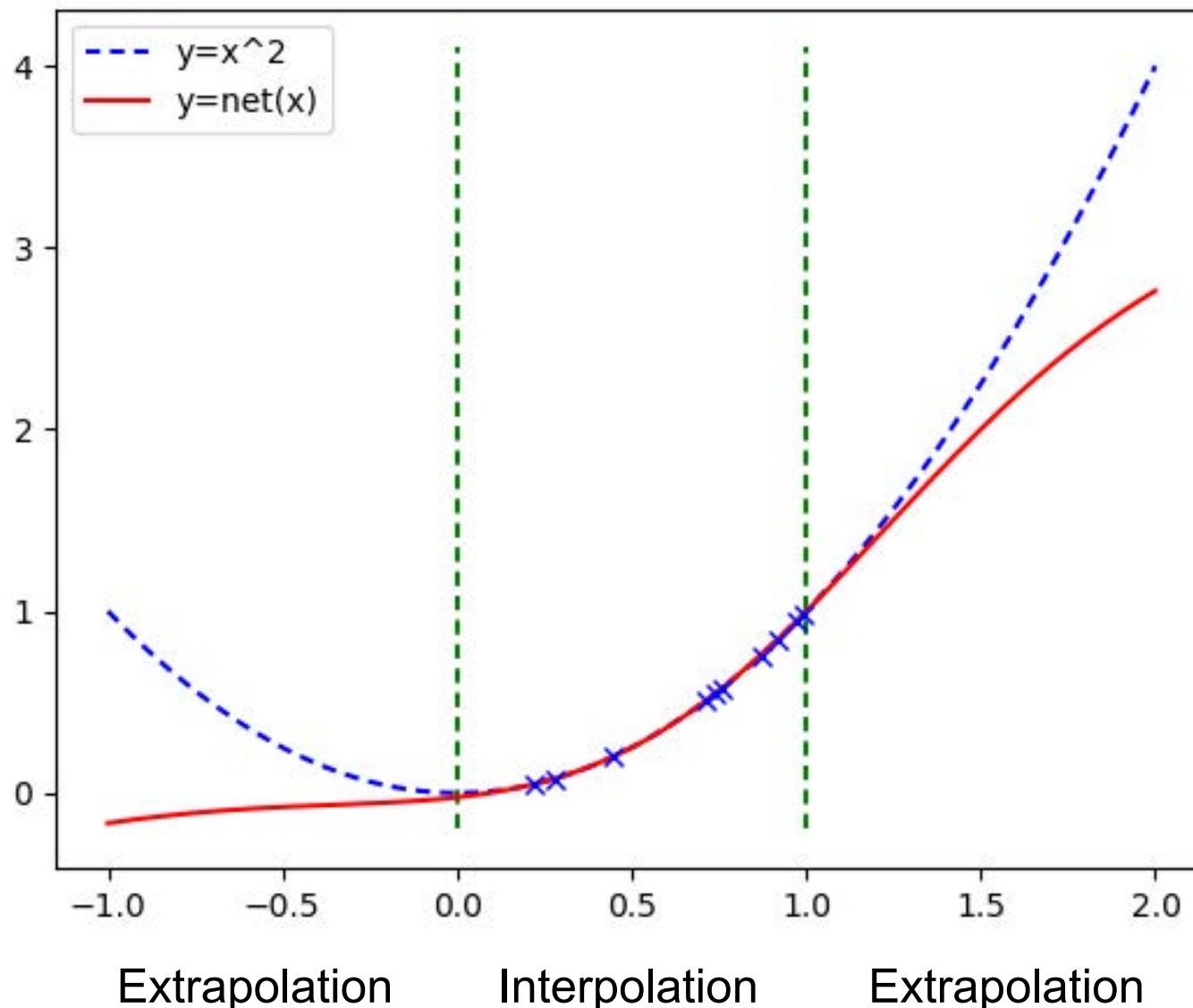


$$E(\mathbf{w}) = \sum_{n=1}^N E_n(\mathbf{w}) + \lambda \|\mathbf{w}\|^2$$

How large should  $\lambda$  be?

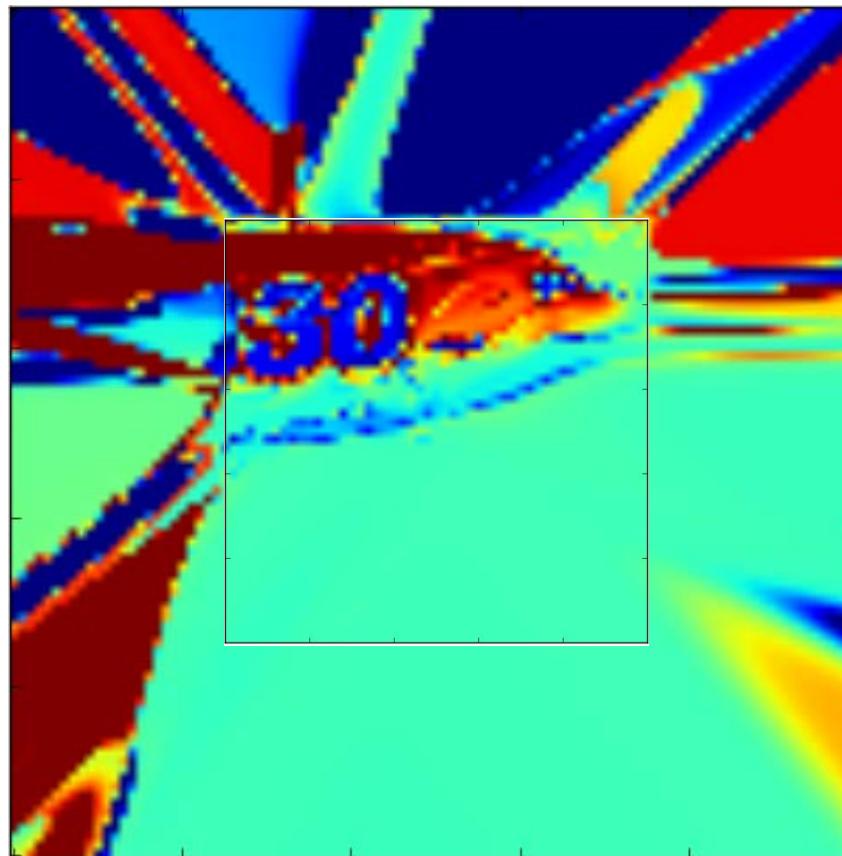
→ Use validation data as usual.

# Interpolation vs Extrapolation



→ Perceptrons do **not** extrapolate well

# Interpolation vs Extrapolation



→ Perceptrons do **not** extrapolate well

# Reminder: Learning the Weights

- Given a training set be  $\{(\mathbf{x}_n, [t_n^1, \dots, t_n^K])_{1 \leq n \leq N}\}$  where  $t_n^k \in \{0,1\}$  is the probability that sample  $\mathbf{x}_n$  belongs to class k, we write:

$$\mathbf{y}_n = f_{\mathbf{w}}(\mathbf{x}_n) \in R^K$$

$$p_n^k = \frac{\exp(\mathbf{y}_n[k])}{\sum_j \exp(\mathbf{y}_n[j])}$$

- We want to minimize the cross entropy

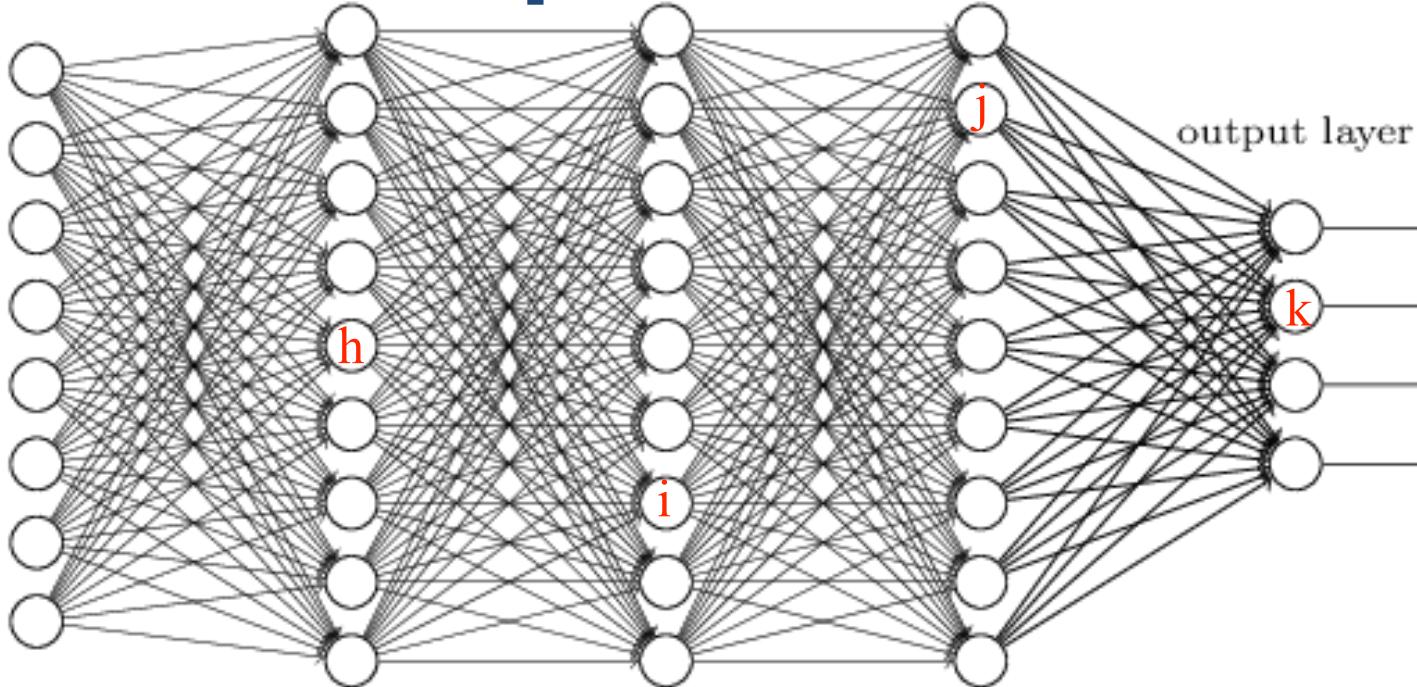
$$E(\mathbf{w}) = \frac{1}{N} \sum_{n=1}^N E_n(\mathbf{w}) ,$$

$$E_n(\mathbf{w}) = - \sum t_n^k \ln(p_n^k) ,$$

with respect to the coefficients of  $\mathbf{w}$ .

# Deep Stochastic Gradient

input layer



output layer

$$E_n(\mathbf{w}) = - \sum_{k=1}^K t_n^k \ln(p_n^k), \\ = L_n(a_1, \dots, a_K)$$

$$z_h = \sigma(a_h)$$

$$z_i = \sigma(a_i)$$

$$z_j = \sigma(a_j)$$

$$y_k = a_k$$

$x_l$

$$a_h = \sum_l w_{hl} x_l \quad a_i = \sum_h w_{ih} z_h \quad a_j = \sum_i w_{ji} z_i \quad a_k = \sum_j w_{kj} z_j$$

$$E(\mathbf{w}) = \sum_{n=1}^N E_n(\mathbf{w})$$

$$\mathbf{w} = [w_{ji}]$$

$$\mathbf{w}^{\tau+1} = \mathbf{w}^\tau - \eta \sum_{n \in B^\tau} \nabla E_n(\mathbf{w}^\tau)$$

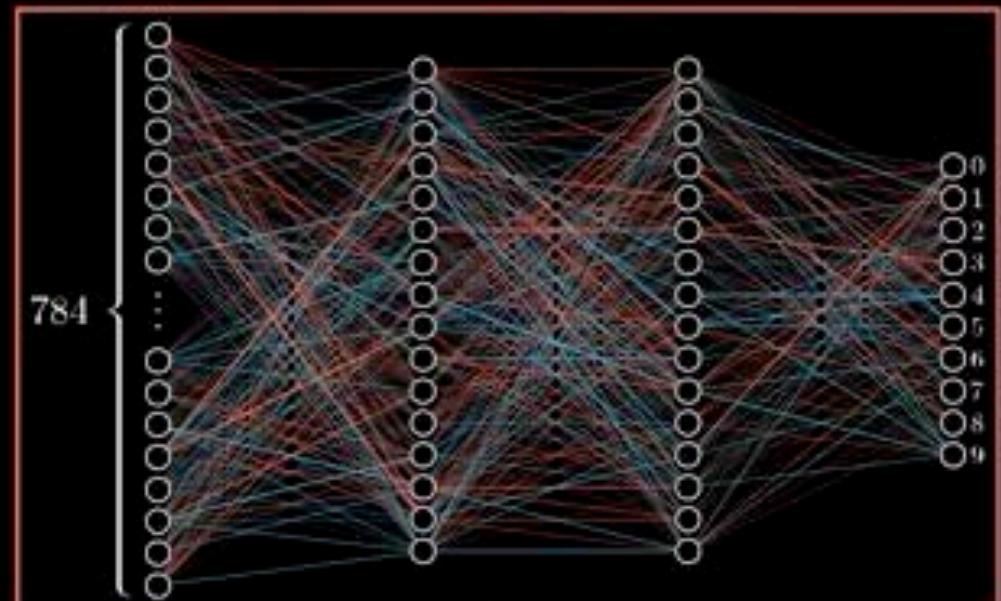
$$\nabla E_n = \left[ \frac{\delta E_n}{\delta w_{ji}} \right]$$

But how do we compute all these derivatives?

# Back Propagation

$$-\nabla C(\dots) = \begin{bmatrix} 0.16 \\ 0.72 \\ -0.93 \\ \vdots \\ 0.04 \\ 1.64 \\ 1.52 \end{bmatrix}$$

All weights  
and biases

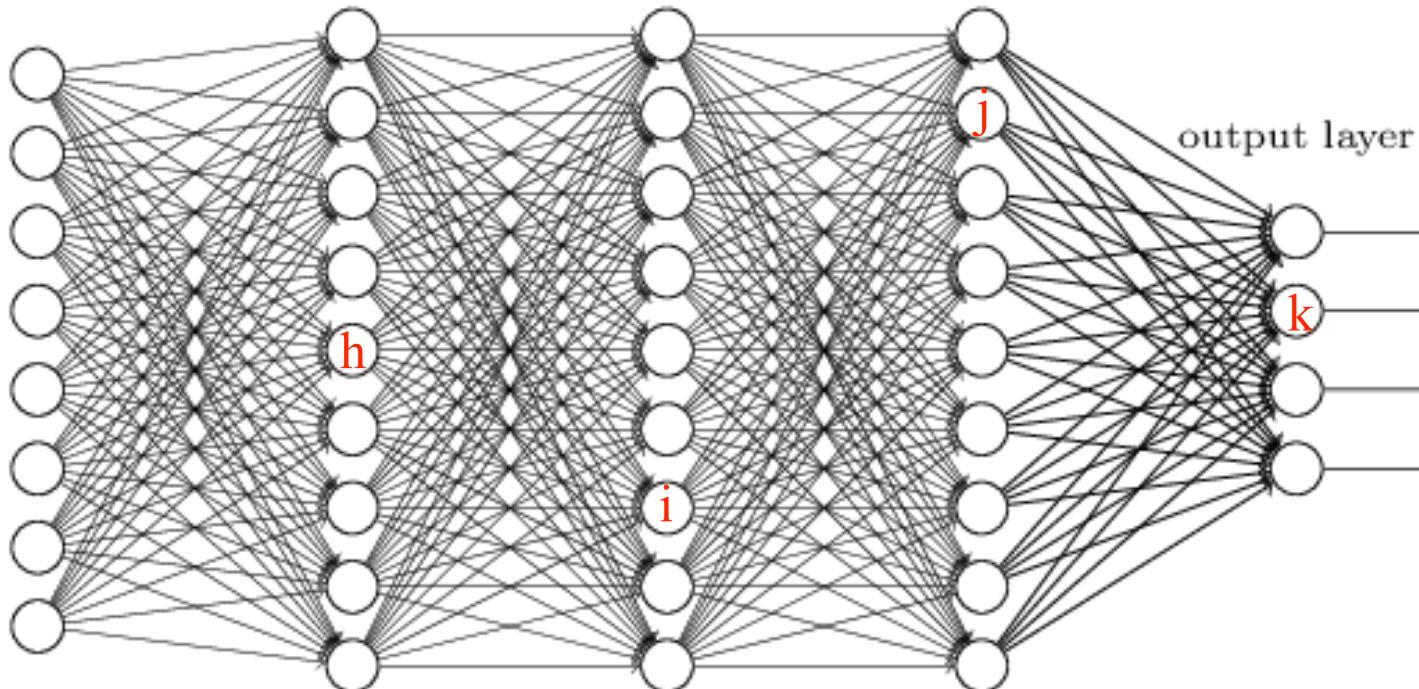


$$C(w_0, w_1, \dots, w_{13,001}) = 2.85$$

3Blue1Brown: <https://www.youtube.com/watch?v=Ilq3gGewQ5U>

# Partial Derivatives

input layer



$$E_n = L_n(a_1, \dots, a_K)$$

$$z_h = \sigma(a_h)$$

$$z_i = \sigma(a_i)$$

$$z_j = \sigma(a_j)$$

$$y_k = a_k$$

$x_l$

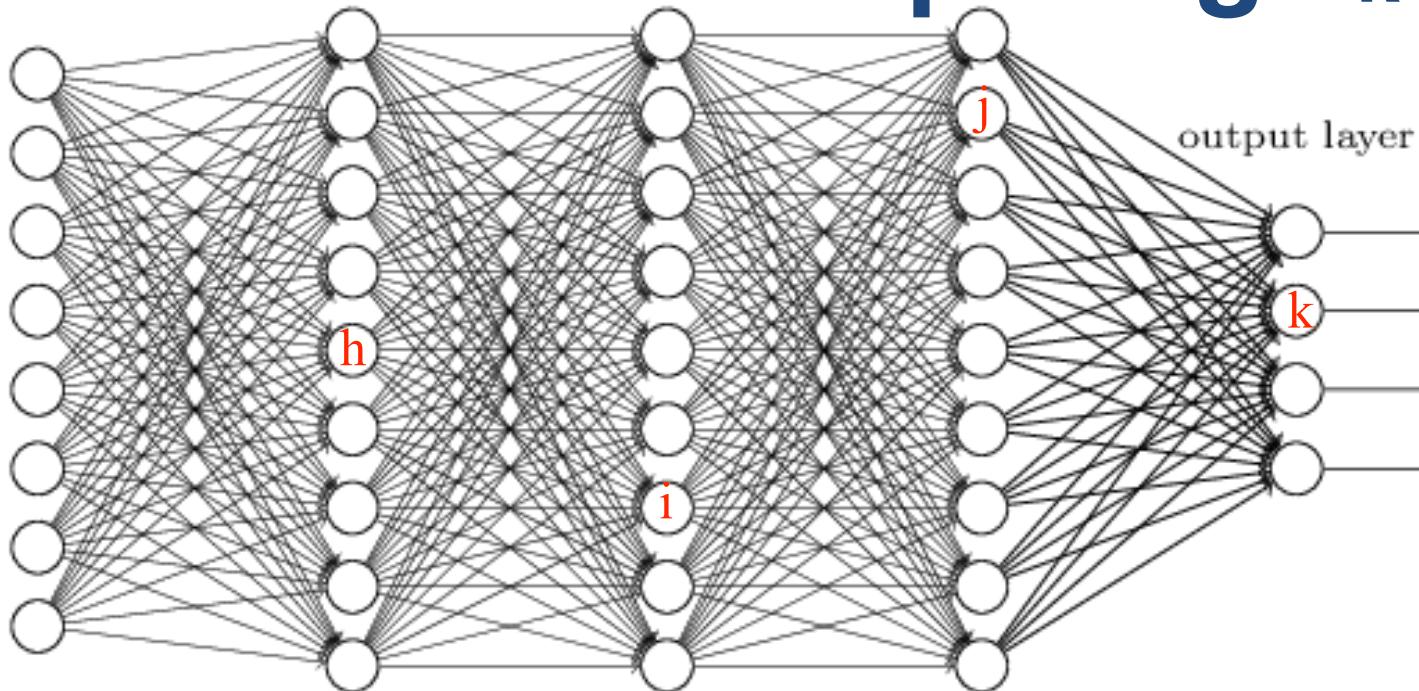
$$a_h = \sum_l w_{hl} x_l \quad a_i = \sum_h w_{ih} z_h \quad a_j = \sum_i w_{ji} z_i \quad a_k = \sum_j w_{kj} z_j$$

$$\begin{aligned} \frac{\delta E_n}{\delta w_{ji}} &= \frac{\delta E_n}{\delta a_j} \frac{\delta a_j}{\delta w_{ji}} \\ &= \delta_j z_i \end{aligned}$$

$$\text{with } \delta_j \equiv \frac{\delta E_n}{\delta a_j}$$

# Computing $\delta_k$

input layer



output layer

$$E_n = L_n(a_1, \dots, a_K)$$

$$z_h = \sigma(a_h)$$

$$z_i = \sigma(a_i)$$

$$z_j = \sigma(a_j)$$

$$y_k = a_k$$

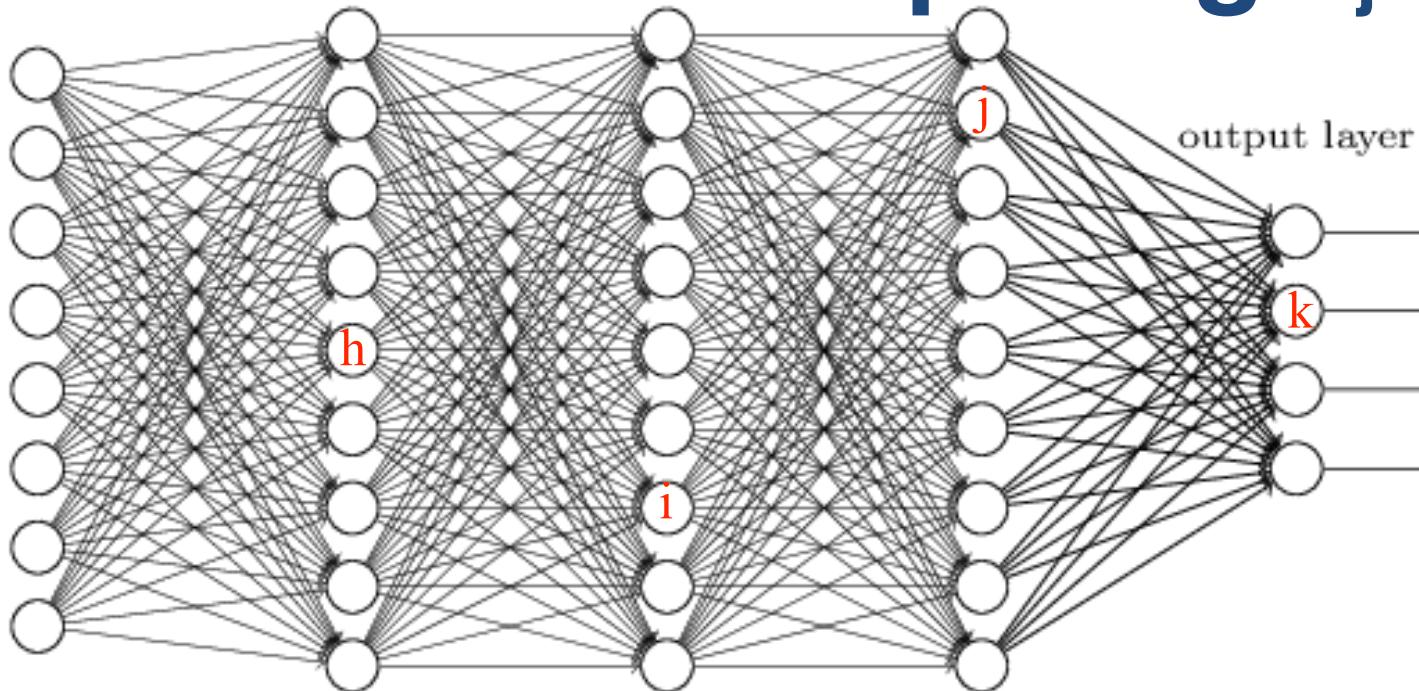
$x_l$

$$a_h = \sum_l w_{hl} x_l \quad a_i = \sum_h w_{ih} z_h \quad a_j = \sum_i w_{ji} z_i \quad a_k = \sum_j w_{kj} z_j$$

Output layer:  $\delta_k = \frac{\delta E_n}{\delta a_k}$  No  $w_{ij}$  here.

# Computing $\delta_j$

input layer



$$E_n = L_n(a_1, \dots, a_K)$$

$$z_h = \sigma(a_h)$$

$$z_i = \sigma(a_i)$$

$$z_j = \sigma(a_j)$$

$x_l$

$$a_h = \sum_l w_{hl} x_l \quad a_i = \sum_h w_{ih} z_h \quad a_j = \sum_i w_{ji} z_i \quad y_k = \sum_j w_{kj} z_j$$

Other layers:  $\delta_j = \frac{\delta E_n}{\delta a_j} = \sum_k \frac{\delta E_n}{\delta a_k} \frac{\delta a_k}{\delta a_j}$

$$= \sigma'(a_j) \sum_k w_{kj} \delta_k$$

$$a_k = \sum_j w_{kj} \sigma(a_j)$$

# Back Propagation

Forward pass:

$$\forall h, a_h = \sum_l w_{hl} x_l, z_h = \sigma(a_h)$$

$$\forall i, a_i = \sum_h w_{ih} z_h, z_i = \sigma(a_i)$$

$$\forall j, a_j = \sum_i w_{ji} z_i, z_j = \sigma(a_j)$$

$$\forall k, a_k = \sum_j w_{kj} z_j$$

Backward pass:

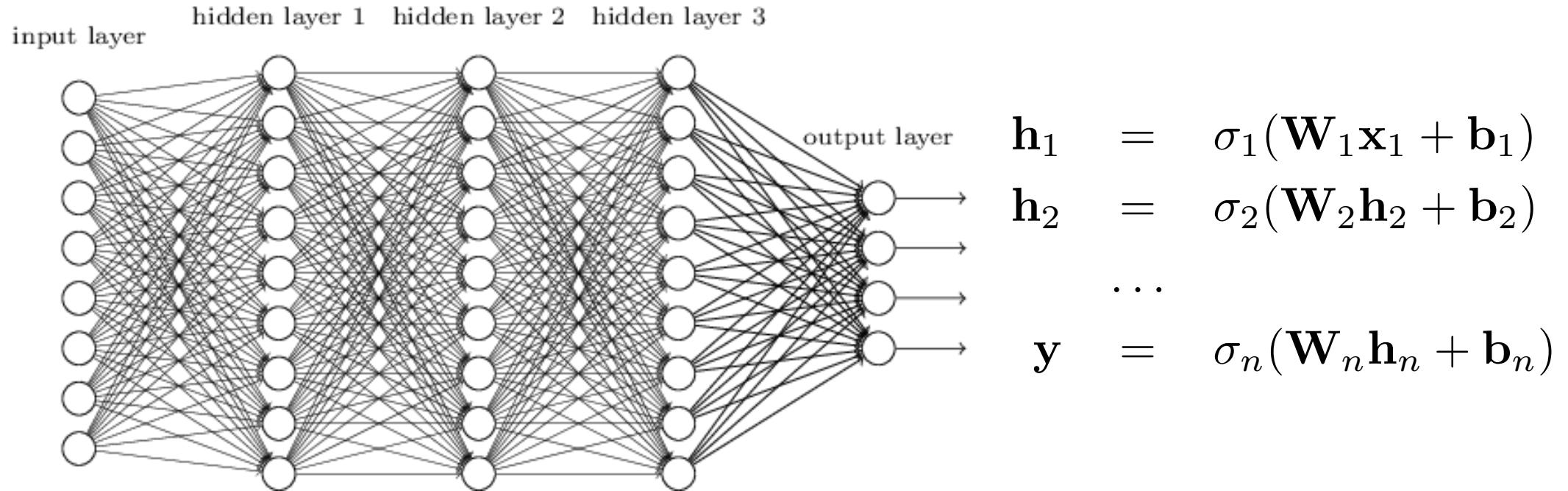
$$\forall k, \delta_k = \frac{\delta E_n}{\delta a_k}$$

$$\forall j, \delta_j = \sigma'(a_j) \sum_k w_{kj} \delta_k$$

$$\forall i, \delta_i = \sigma'(a_i) \sum_j w_{ji} \delta_j$$

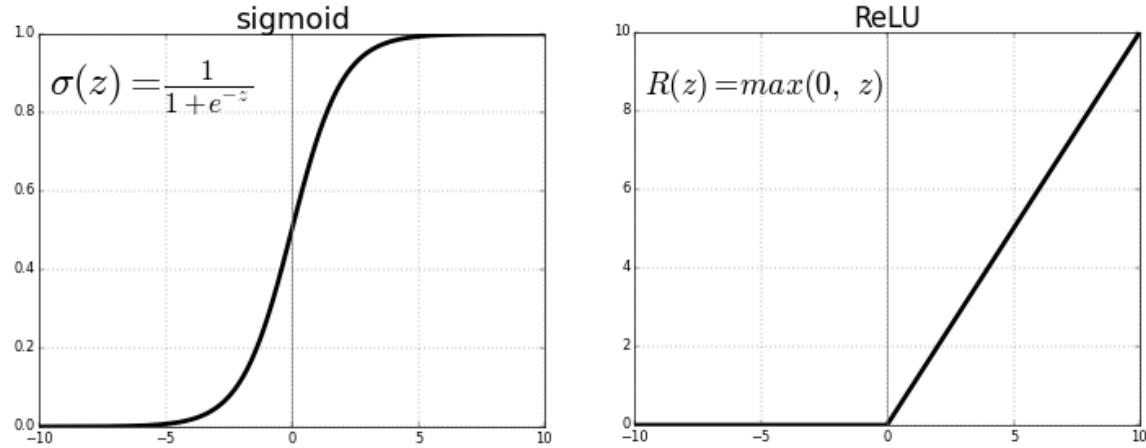
$$\forall h, \delta_h = \sigma'(a_h) \sum_j w_{ih} \delta_h$$

# Back Propagation



- Both the loss and its derivatives can be computed using simple and regular operations.
- Can be implemented on GPUs and runs much faster than on CPUs.

# Vanishing and Exploding Gradients



$$\forall i, \delta_i = \sigma'(a_i) \sum_j w_{ji} \delta_j$$

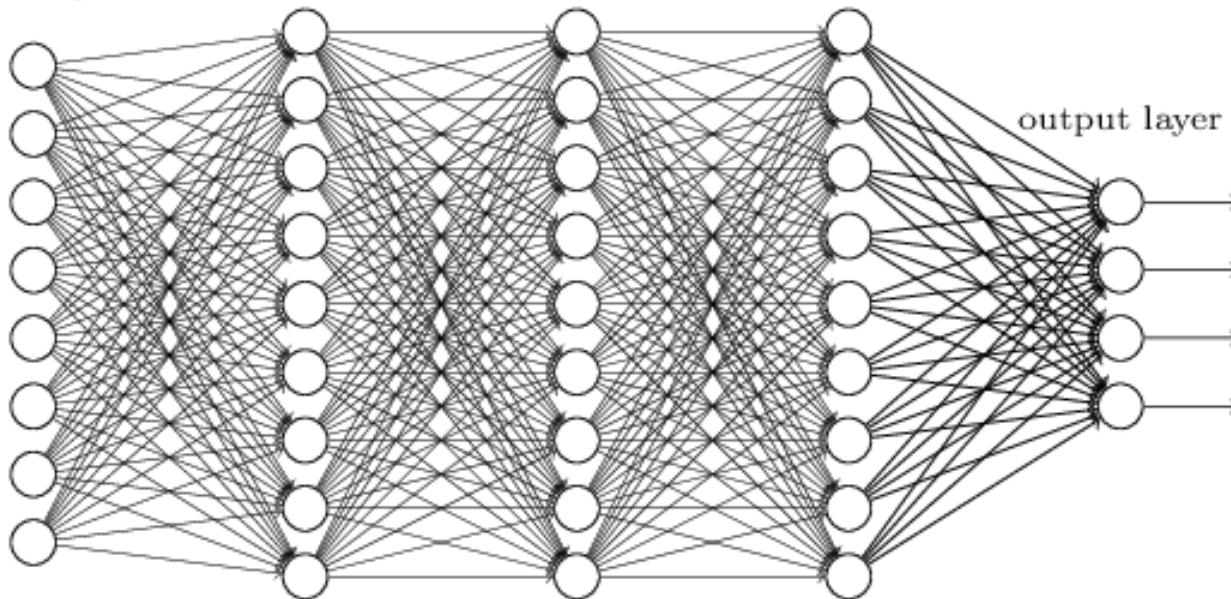
In a very deep network, the derivatives at all levels are multiplied by each other:

- As  $\sigma(a_i)$  approaches zero, so does  $\delta_i$  and the gradients in the lower layers are lost. —> Vanishing gradients.
- If the  $\delta_i$  become large in several consecutive layers, their product can become exponentially large —> Exploding gradients.

The problem can be mitigated by using ReLU, clipping the gradients, relying on gated or resnet-type architectures, ...

# Back Propagation Works!

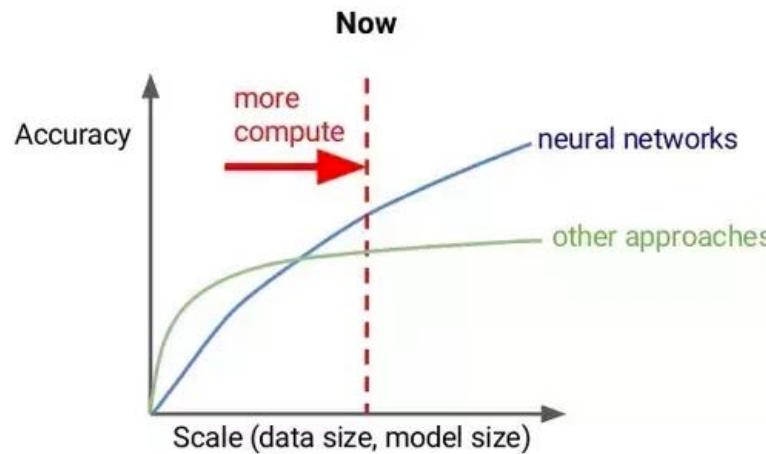
input layer      hidden layer 1    hidden layer 2    hidden layer 3



$$\begin{aligned} \mathbf{h}_1 &= \sigma_1(\mathbf{W}_1 \mathbf{x}_1 + \mathbf{b}_1) \\ \mathbf{h}_2 &= \sigma_2(\mathbf{W}_2 \mathbf{h}_2 + \mathbf{b}_2) \\ &\dots \\ \mathbf{y} &= \sigma_n(\mathbf{W}_n \mathbf{h}_n + \mathbf{b}_n) \end{aligned}$$

- Both the loss and its derivatives can be computed using simple and regular operations.
- Can be implemented on GPUs and runs much faster than on CPUs.
- Vanishing and exploding gradients can be handled through a number of methods, the simplest of which is to use reLU to introduce the non-linearities.

# MLPs in Short



- Deep Neural Networks can handle huge training databases.
  - When the objective function can be minimized, the results are outstanding.
  - There are failure cases and performance is hard to predict.
- > Many questions are still open and there is much theoretical work left to do.