Multi-Layer Networks

Tripp Deep Learning F22

UNIVERSITY OF WATERLOO

TODAY'S GOAL

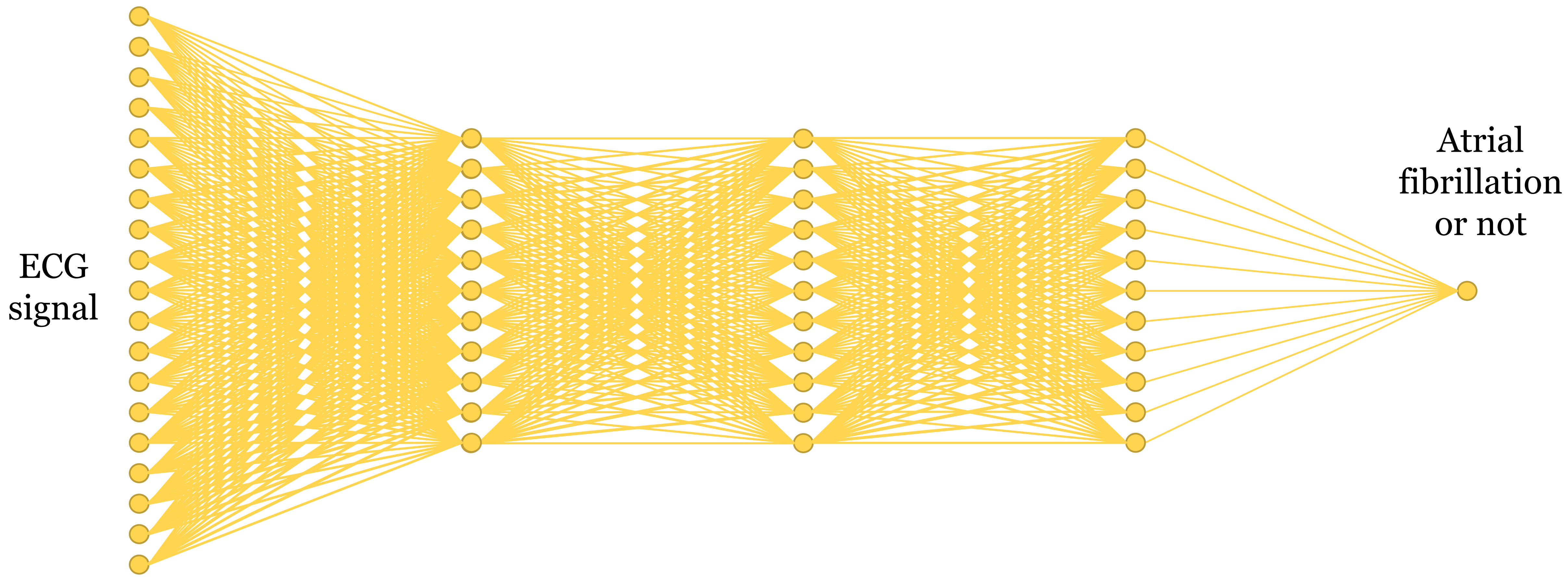
By the end of the class, you should be able to explain how multi-layer networks work and what they are capable of, and you should be able to choose suitable outputs and losses for regression and classification.

Summary

1. Multi-layer feedforward networks make a series of simple but unintuitive calculations
2. Multi-layer networks are composite functions
3. Multi-layer networks break the curse of dimensionality
4. A two-layer network can closely approximate any continuous function on a bounded domain
5. Networks with different weight matrices can perform identical mappings
6. Classification and regression employ different output and loss functions
7. Multi-layer networks are also called multi-layer perceptrons, but are not perceptrons

MULTI-LAYER FEEDFORWARD NETWORKS MAKE A SERIES OF SIMPLE BUT UNINTUITIVE CALCULATIONS

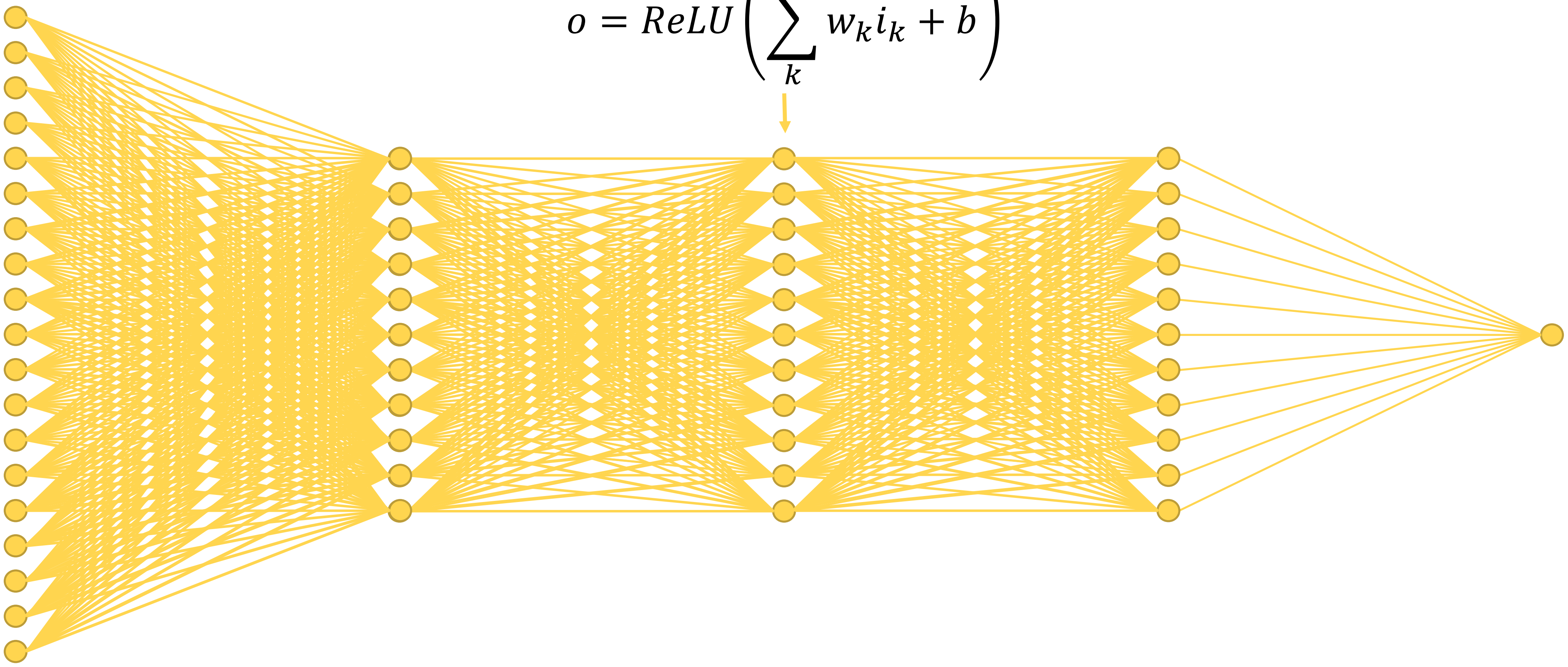
Overall network function is intuitive



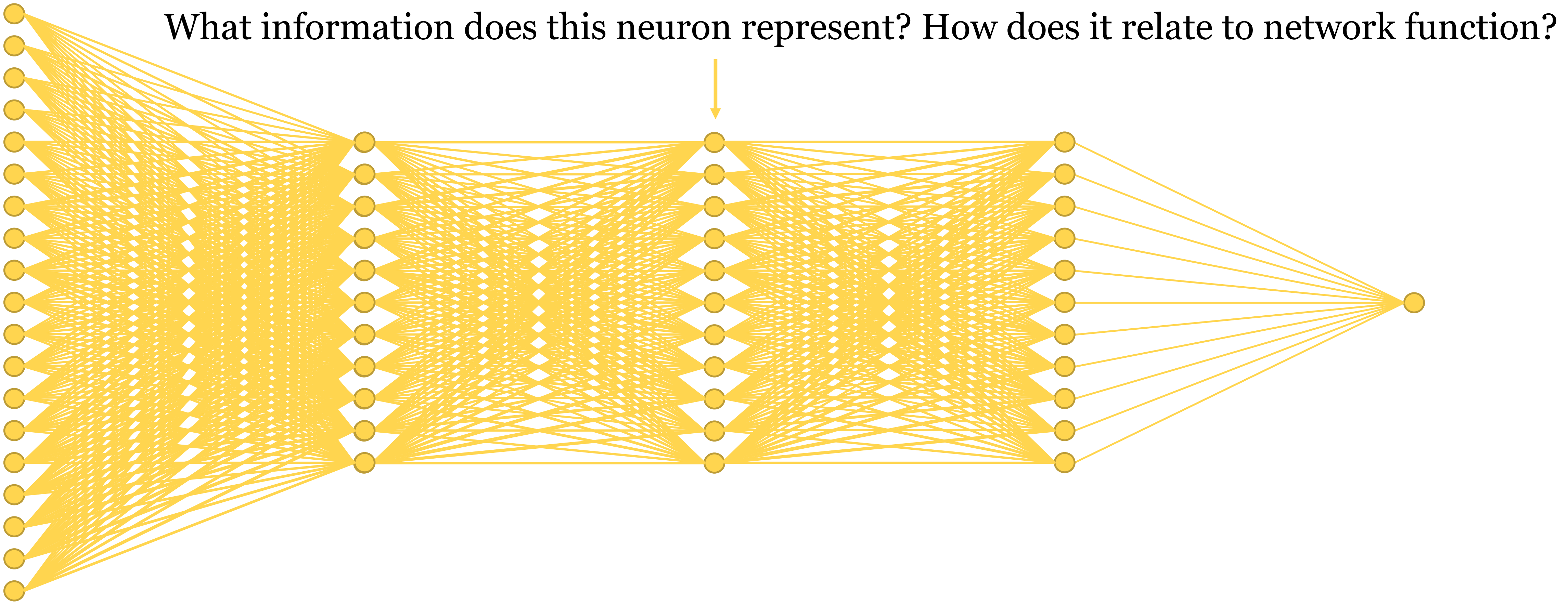
UNIVERSITY OF WATERLOO

How each neuron works is intuitive

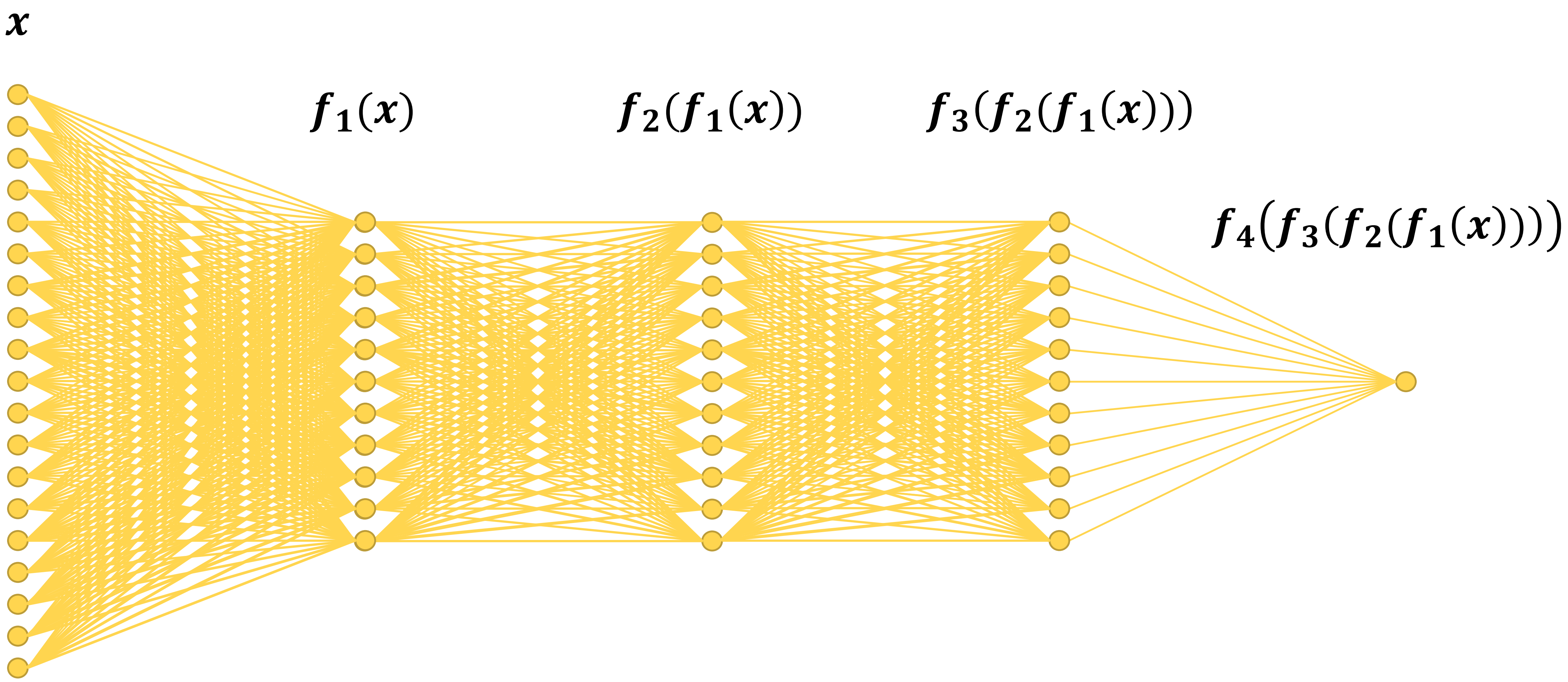
$$o = ReLU\left(\sum_k w_k i_k + b\right)$$

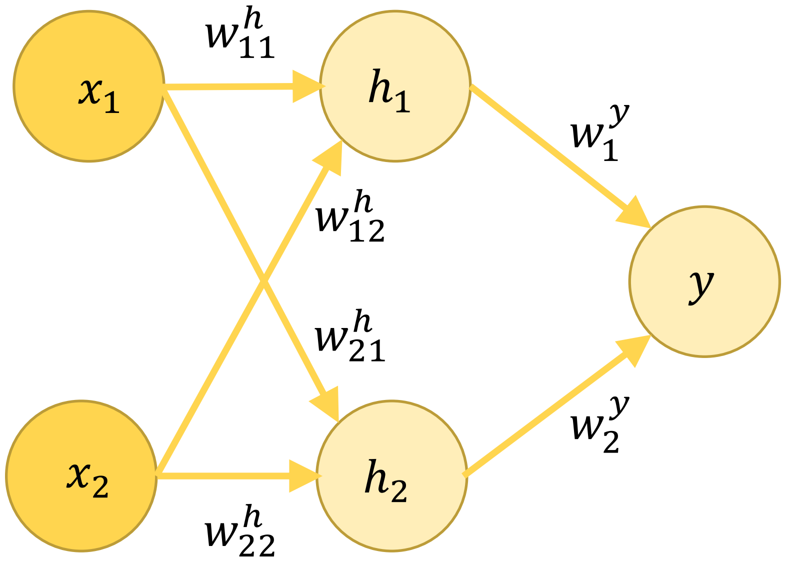


Role of each neuron in network function is opaque



MULTI-LAYER NETWORKS ARE COMPOSITE FUNCTIONS


$\boldsymbol{x}$
$\boldsymbol{f_1(x)}$
$\boldsymbol{f_2(f_1(x))}$
$\boldsymbol{f_3(f_2(f_1(x)))}$
$\boldsymbol{f_4(f_3(f_2(f_1(x))))}$



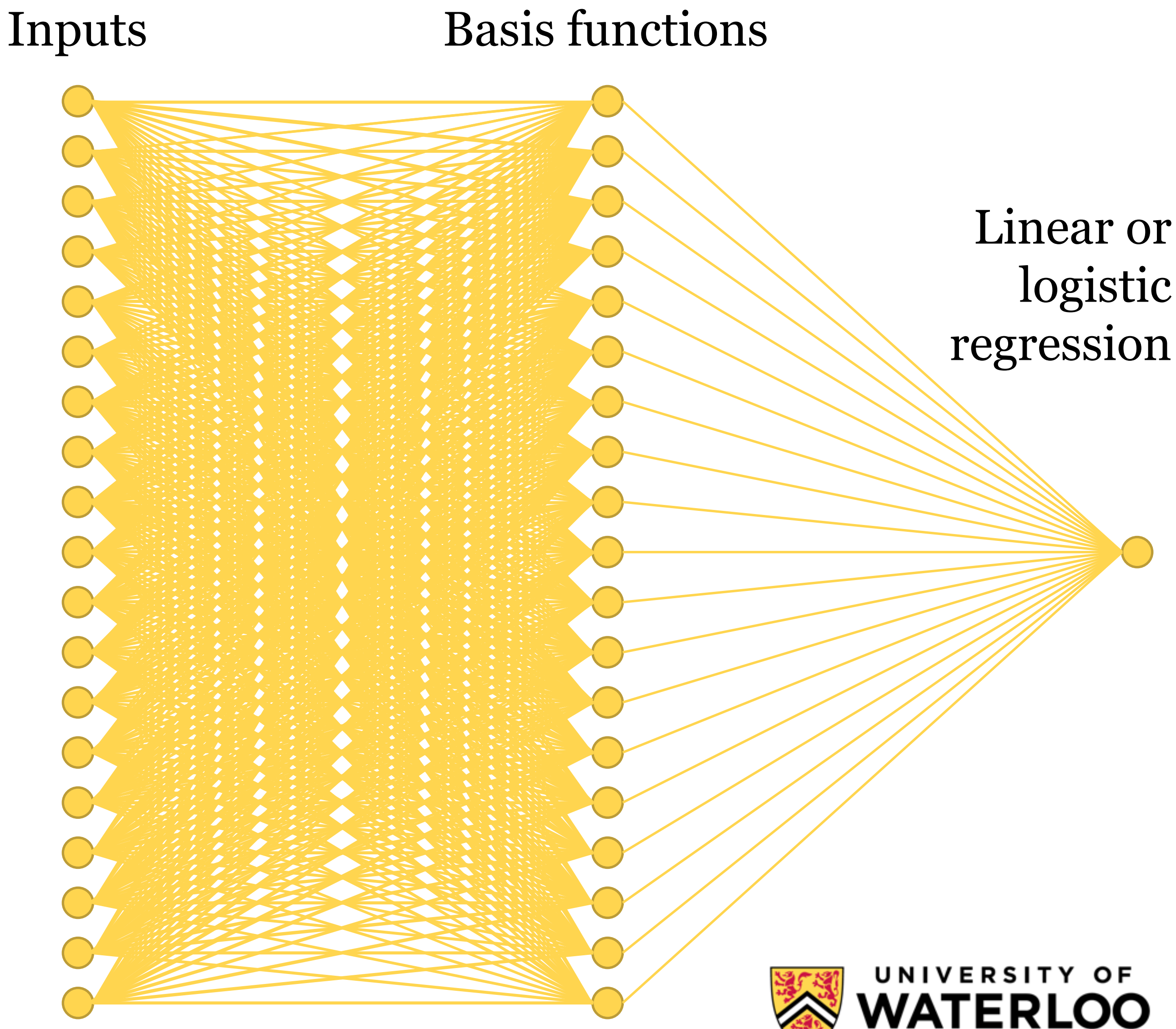
$$y = g_y(w_1^y h_1 + w_2^y h_2 + b_y)$$

$$= g_y(w_1^y g_h(w_{11}^h x_1 + w_{12}^h x_2 + b_{h1}) + w_2^y g_h(w_{21}^h x_1 + w_{22}^h x_2 + +b_{h2}) + b_y)$$

MULTI-LAYER NETWORKS BREAK THE CURSE OF DIMENSIONALITY

Recall: Curse of dimensionality

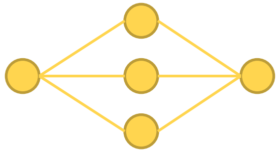
- Multi-layer neural networks take advantage of this by adapting basis functions so that
  - Regions of variation correspond to regions over which input typically varies
  - Directions of variation correspond to directions over which output typically varies

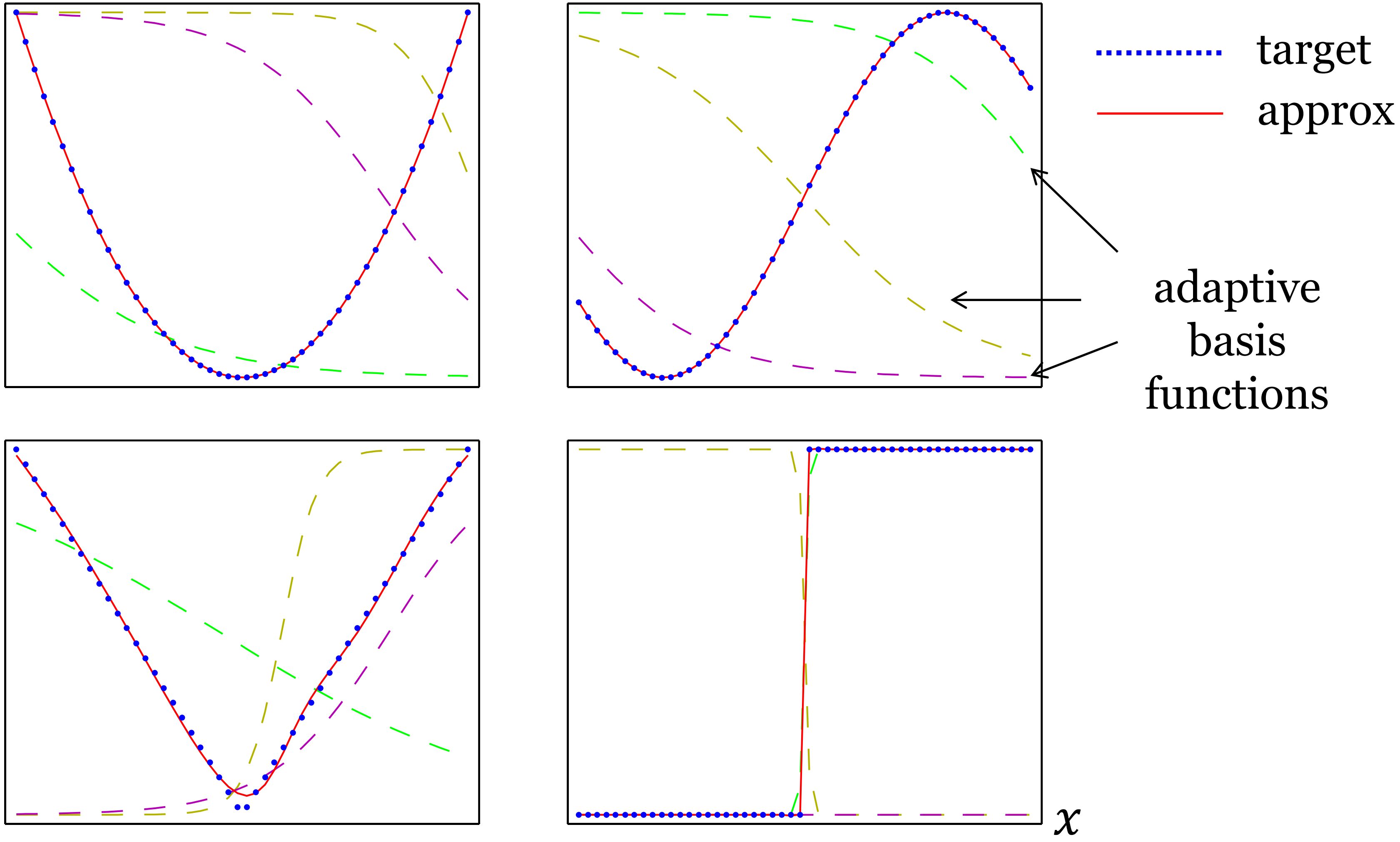


Basis functions adapt to the task

- Example: Approximating different functions with three sigmoid hidden units and a linear output unit
- The hidden units adapt to the task

$x \quad h_i \quad \hat{y} = \sum w_i h_i + b$



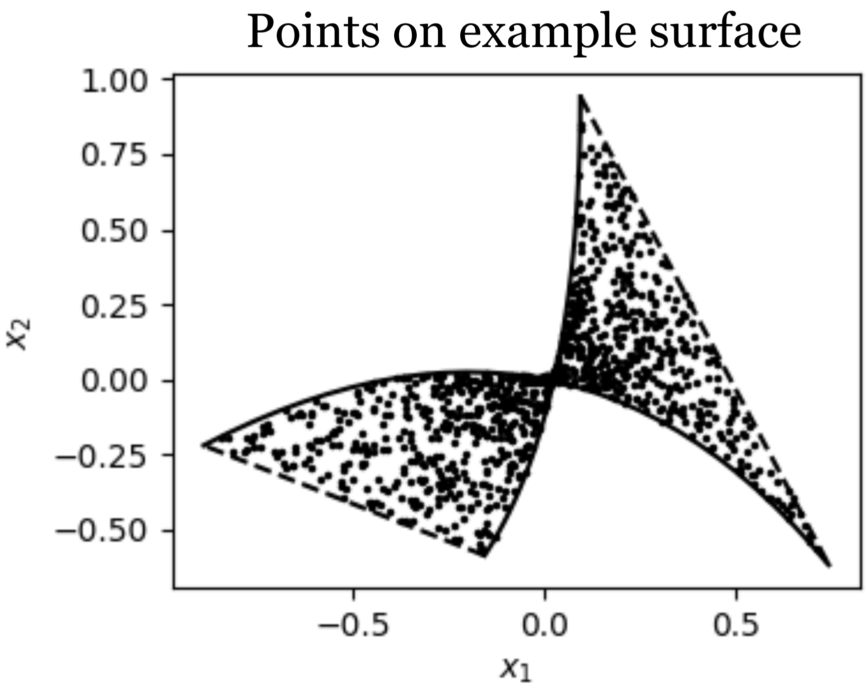


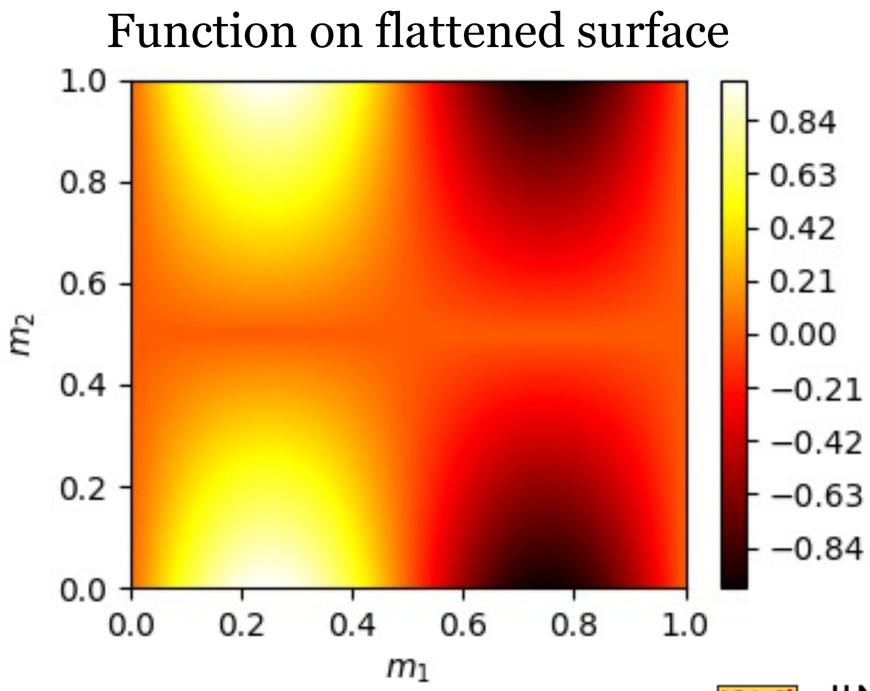
Bishop, Pattern Recognition & Machine Learning

UNIVERSITY OF WATERLOO

Fewer basis functions are needed if they are adaptive

- Example: Nonlinear function on randomly bent 2D surface in 5D space
- Different parts of the surface can overlap in lower-dimensional projections, so all the dimensions should be considered

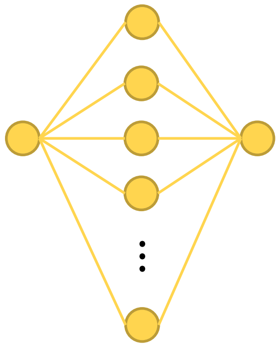


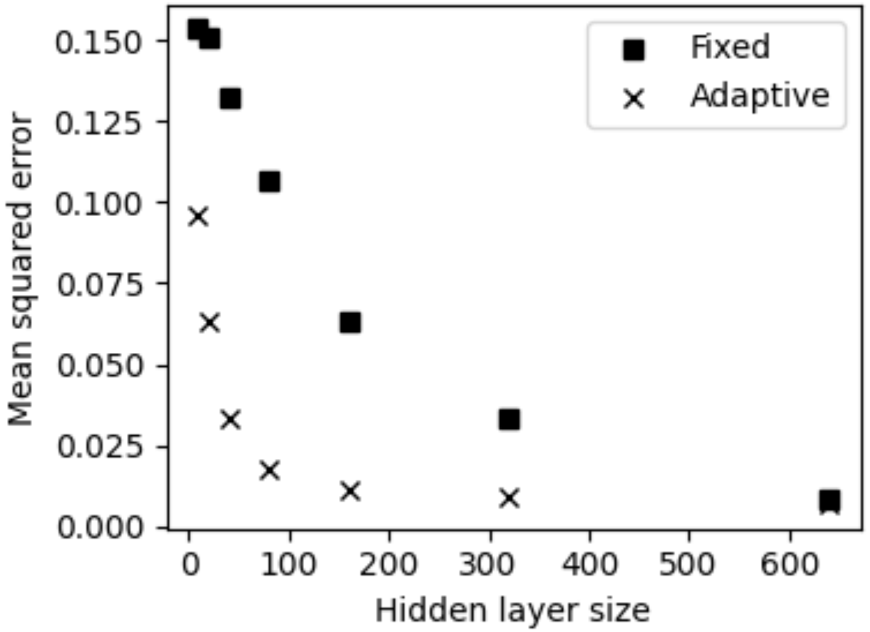


Fewer basis functions are needed if they are adaptive

- Two-layer networks with n ReLU neurons in the hidden layer

$$x \qquad h_i \qquad \hat{y} = \sum w_i h_i + b$$





UNIVERSITY OF WATERLOO

A TWO-LAYER NETWORK CAN CLOSELY APPROXIMATE ANY CONTINUOUS FUNCTION ON A BOUNDED DOMAIN

Universal approximation

- In principle a network with a single hidden layer can approximate any continuous function with arbitrary precision, provided it has enough hidden units
- Proofs allow for an unbounded number of hidden units (although there has also been separate work to establish bounds)
- The number of hidden units required may be impractical
- This result doesn't guarantee that a given algorithm can learn the approximation

Universal approximation

Two-layer networks of the form,

$$\hat{y} = \boldsymbol{w}_y^T\left[g(w_i^T \boldsymbol{x} + b_i)\right] + b_y$$

where g is a scalar nonlinearity that can be a sigmoid or ReLU (or many other continuous functions) can approximate any function over a bounded domain to arbitrary precision, i.e.,

$$|\hat{y} - y| < \varepsilon$$

for all $\boldsymbol{x}$ in the domain and any choice of ε.

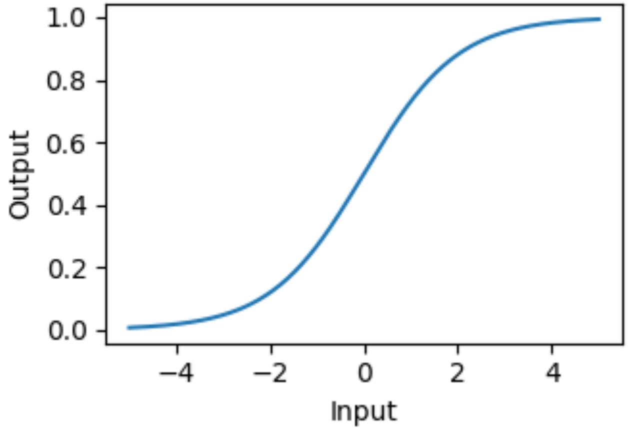
Influence of sigmoid hidden units with one input

- Let's get a sense of how this can be
- Suppose we have a two-layer network G with one input x, n sigmoid hidden units and one linear output,

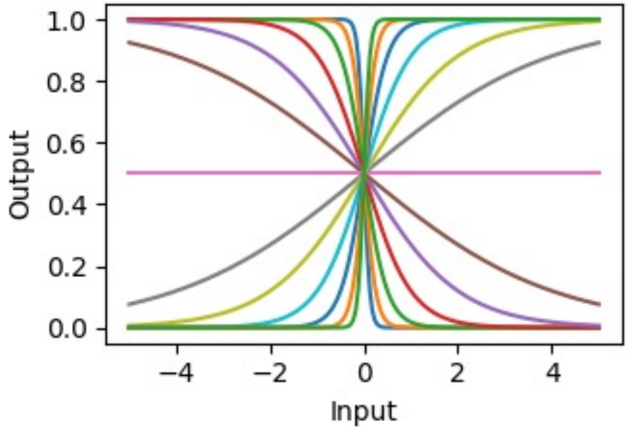
$$\hat{y} = G(x)$$
$$= \sum_{i=1}^{n} w_{yi}\left(1 + e^{-(w_{ix}x + b_i)}\right)^{-1} + b_y$$

- Then $\hat{y}$ is a sum of functions that can vary in slope, offset, and height, as illustrated in these figures

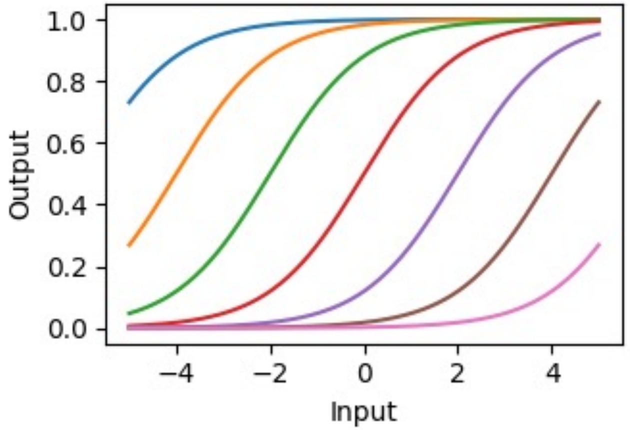
$(1 + e^{-x})^{-1}$



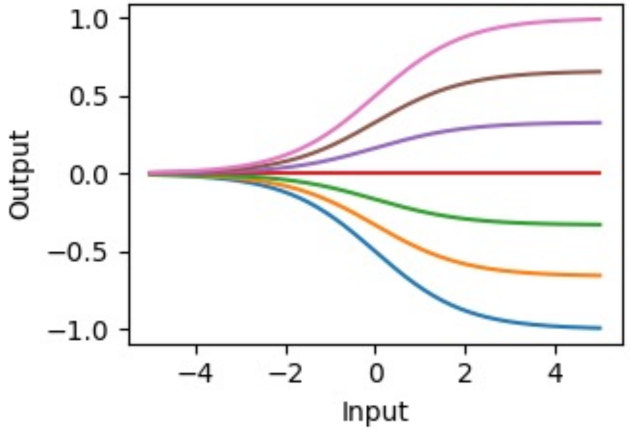
$(1 + e^{-w_{ix}x})^{-1}$



$\left(1 + e^{-(x + b_i)}\right)^{-1}$

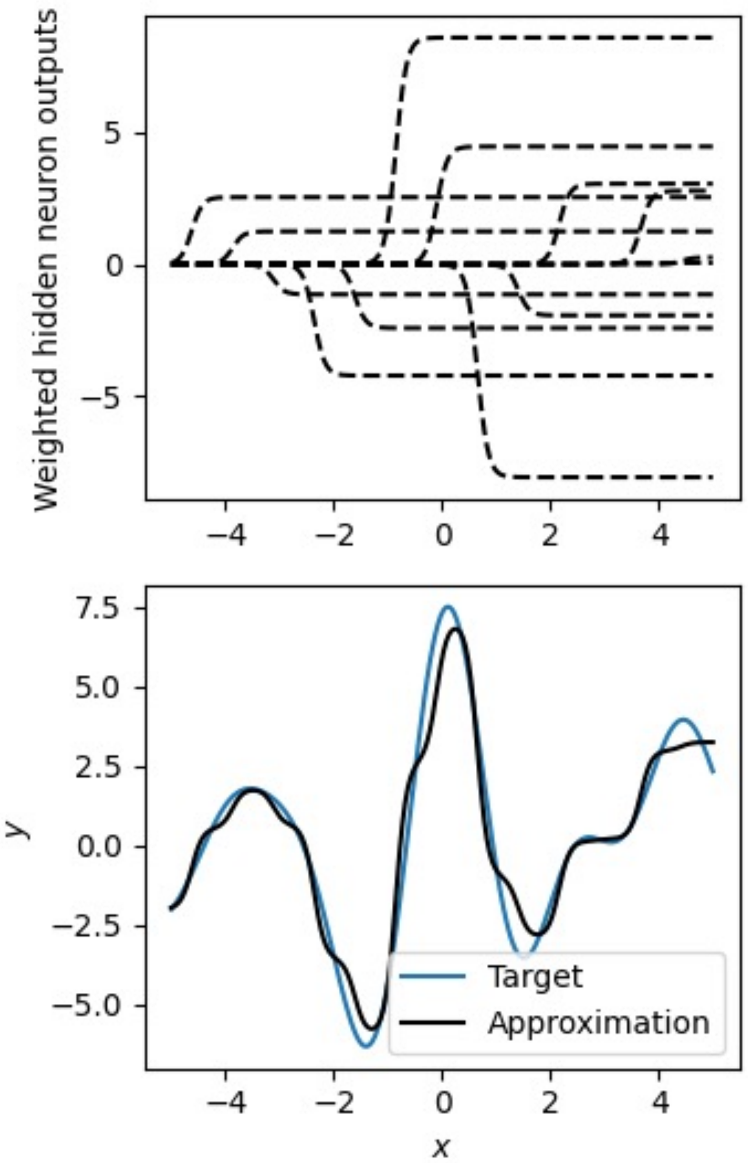


$w_{yi}(1 + e^{-x})^{-1}$



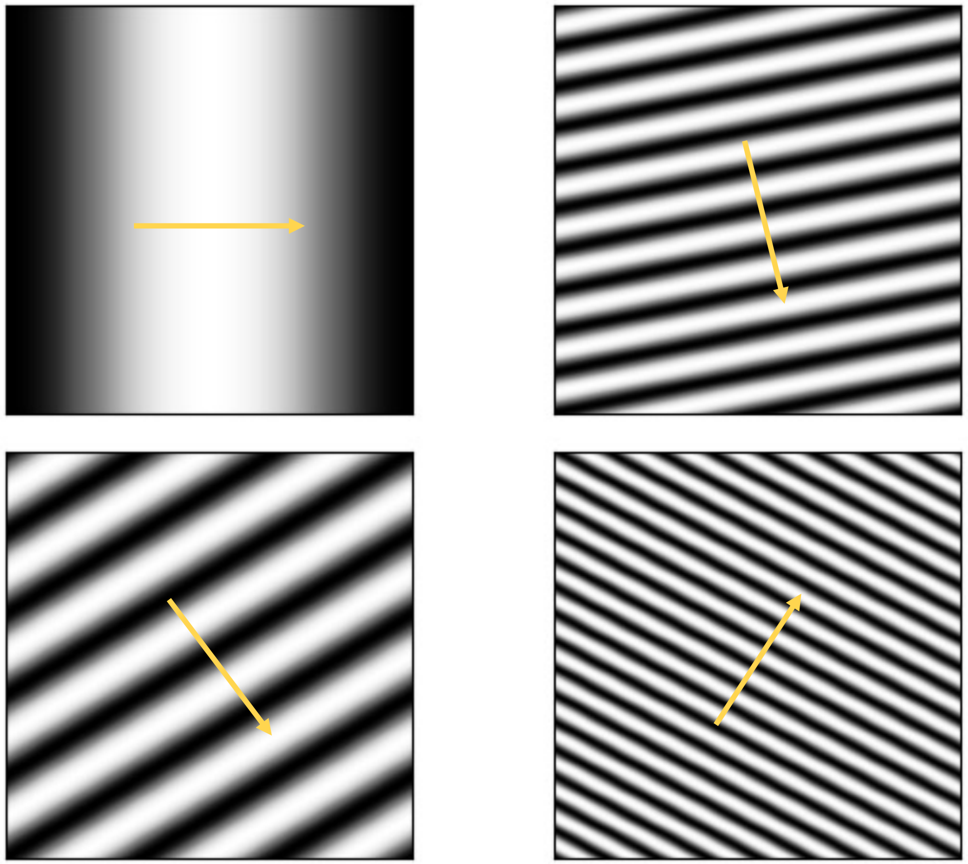
Approximating a 1D function in steps

- To approximate a function $y = f(x)$ on the interval $[x_a, x_b]$ so that $|\hat{y} - y| \leq \varepsilon$:
    1. Choose a step size Δx and choose a weight w_{ix} large enough that the sigmoid changes most of the way from 0 to 1 over a span of Δx
    2. Set the bias to $b_y = f(x_a)$
    3. For each x_i from x_a to x_b in steps of Δx:
        1. Add a hidden neuron with $w_{yi} = f(x_i + \Delta x) - f(x_i)$ and $b_i = -w_{ix}(x_i + \Delta x/2)$
- This will make $\hat{y}$ approximate y, perhaps poorly. To make $|\hat{y} - y| \leq \varepsilon$ increase w_{ix} and decrease Δx as needed



Extending to multiple input dimensions

- Approximate the desired function with a multi-dimensional Fourier series
  - Use a finite but arbitrarily large number of frequency components
  - Use real-valued form (e.g., amplitude-phase)
- Approximate each component as before
  - Each component is a sinusoidal function of the projection onto a vector $\boldsymbol{v}$ in the input space
  - Approximate each sinusoid with sigmoid functions that point along $\boldsymbol{v}$, specifically $\left(1+e^{-a\boldsymbol{v}^T\boldsymbol{x}+b}\right)^{-1}$

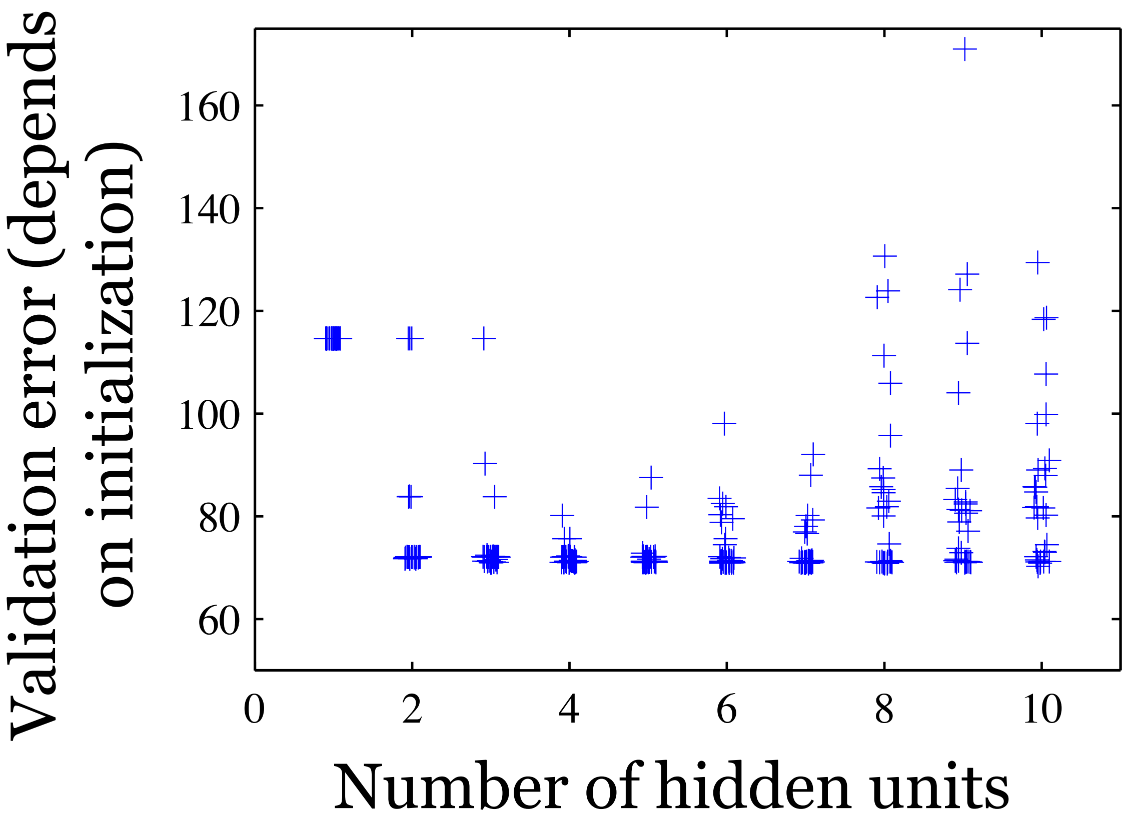


Proofs

- For a discussion of various proofs: Scarselli, F., & Tsoi, A. C. (1998). Universal approximation using feedforward neural networks: A survey of some existing methods, and some new results. *Neural Networks*, 11(1), 15-37.

Large hidden layer increases risk of overfitting

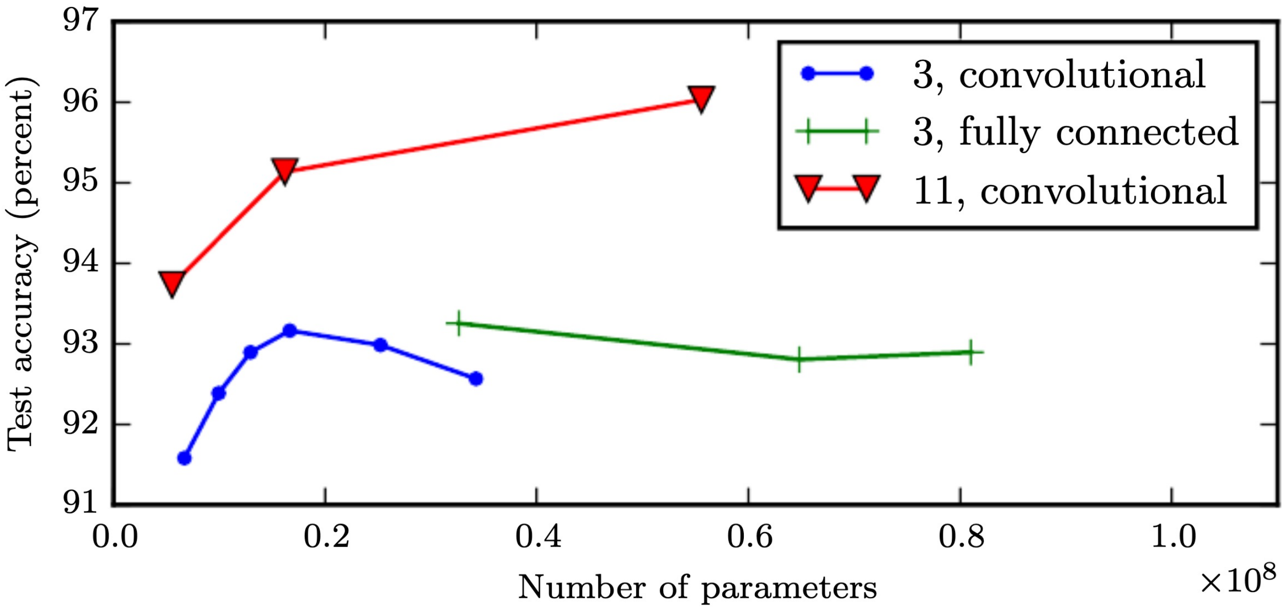
- These are results of training small networks with 30 sets of random initial weights for each hidden-layer size



Goodfellow, Bengio & Courville, *Deep Learning*

Depth

- Deeper networks can represent some kinds of functions more efficiently, particularly functions that are compositions of simpler functions
- Choosing a deep architecture is consistent with the belief that the desired mapping is in this category
- Empirically, greater depth results in better generalization in a wide range of tasks



Goodfellow, Bengio & Courville, *Deep Learning (pg. 197)*

NETWORKS WITH DIFFERENT WEIGHT MATRICES CAN PERFORM IDENTICAL MAPPINGS

Invariance to parameter changes

- For any input-output function that a network may perform with a certain set of parameters, there are many other sets of parameters that implement *exactly* the same function
- In particular, swapping the input and output weights and the bias of a hidden unit with those of another hidden in the same layer, with the same nonlinearity, does not affect the output
- Some networks have additional invariances, for example
  - With tanh hidden-layer activation functions, tanh(-a)=-tanh(a), so changing the signs of both the input and output weights together has no effect
  - With multi-head attention in transformers, the heads are interchangeable
  - With ReLU hidden-layer activation functions, some neurons can become "dead" (unresponsive to any input), so continuous changes in their parameters within some neighbourhood have no effect
  - With ReLU hidden-layer activation functions, multiplying all the weights and biases of one layer by α and dividing the weights of the next layer by α has no effect

Invariance to parameter changes

- This means that there isn't a single global minimum of the loss
- For any minimum, there are **many** equivalent minima
    - E.g., for a network with a single hidden layer with ten units, at least 10! = 3,628,800 equally optimal minima

UNIVERSITY OF WATERLOO

CLASSIFICATION AND REGRESSION EMPLOY DIFFERENT OUTPUT AND LOSS FUNCTIONS

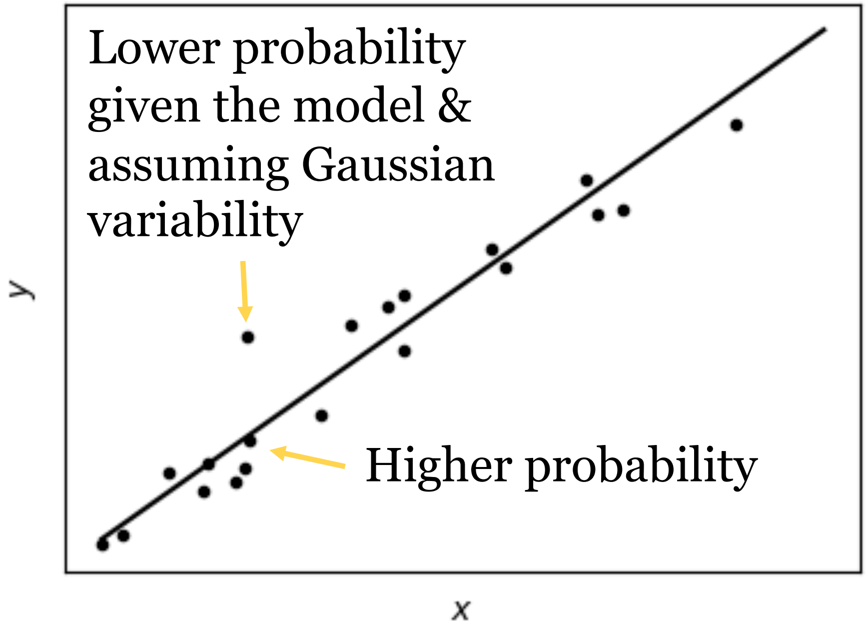
Using negative log likelihood for loss

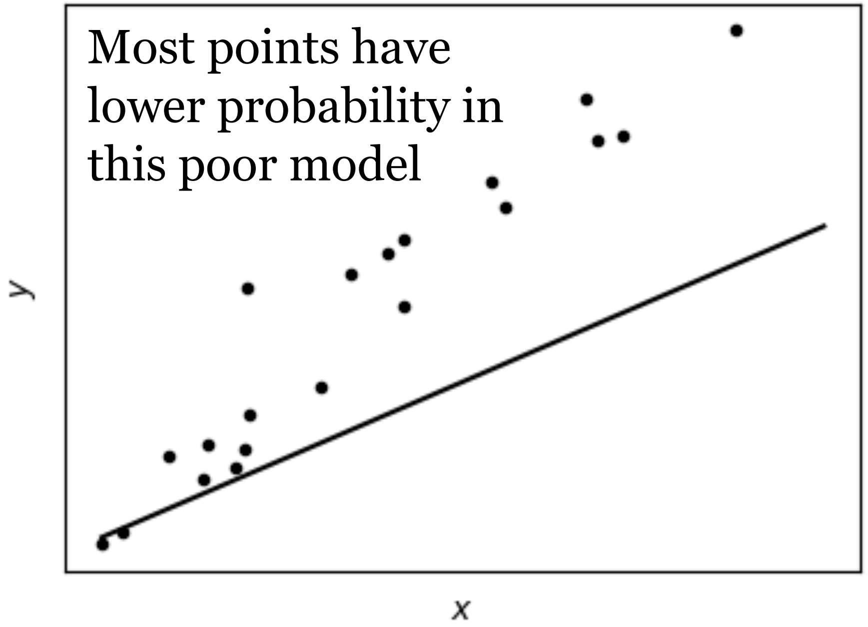
We want to perform maximum likelihood estimation of the network parameters, $\boldsymbol{\theta}$. In other words, we adjust the parameters to maximize the probability of the target given the input and the parameters,

$$p(\mathbf{y}|\boldsymbol{\theta}) = \prod_{n=1}^{N} p(y_n|\boldsymbol{\theta}),$$

over N points of training data. This is equivalent to minimizing the negative log likelihood,

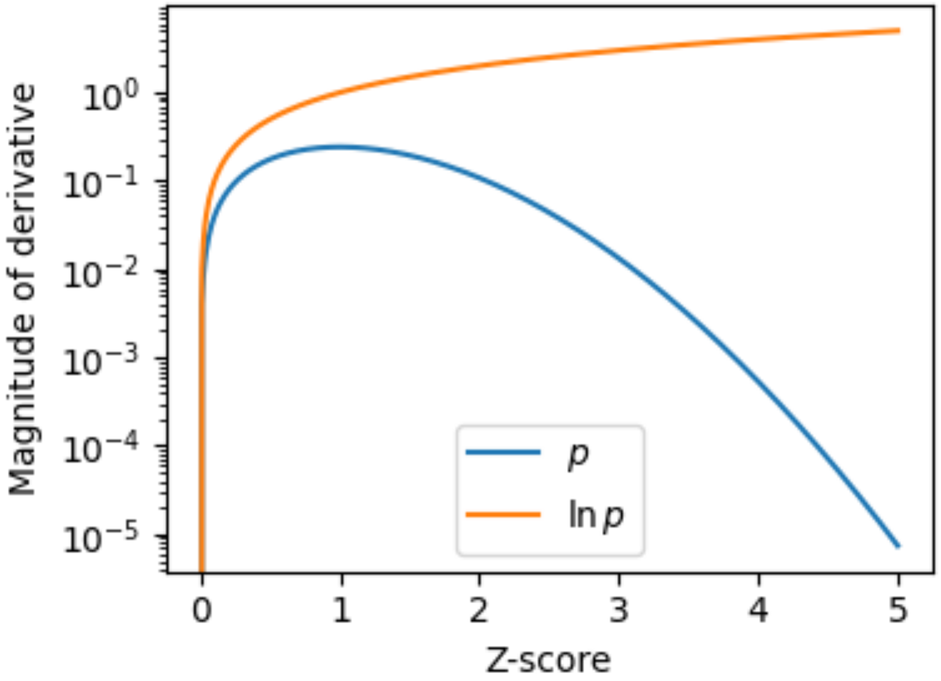
$$L(\boldsymbol{\theta}) = -\ln p(\mathbf{y}|\boldsymbol{\theta}) = -\sum_{n=1}^{N} \ln p(y_n|\boldsymbol{\theta}).$$





Using negative log likelihood for loss

- While they have the same minima, negative log probability is better than negative probability for several reasons:
  - It is a sum rather than a product, so we can do gradient descent on parts of it at a time (minibatches) and these actions combine properly
  - Multiplying multiple small probabilities can underflow the floating-point representation, whereas their log takes on a better-behaved range of values
  - It's useful to take gradient steps that scale with the gradient of the –ve log probability, whereas doing this with probabilities would result in very small steps far from the optimum (plot is for Gaussian distribution)



UNIVERSITY OF WATERLOO

Using negative log likelihood for loss

Example: In a binary classification network with one output, the target is $y \in \{0,1\}$ and we want the network's output $\hat{y}$ to be the probability that $y = 1$. Over N examples in the training dataset, the likelihood is,

$$p(\boldsymbol{y}|\boldsymbol{\theta}) = \prod_{n=1}^{N} \hat{y}_n^{y_n} (1 - \hat{y}_n)^{1-y_n},$$

The negative log-likelihood is,

$$L(\boldsymbol{\theta}) = -\ln p(\boldsymbol{y}|\boldsymbol{\theta}) = -\sum_{n=1}^{N} y_n \ln \hat{y}_n + (1 - y_n) \ln(1 - \hat{y}_n)\,,$$

the binary cross-entropy loss that we have already seen.

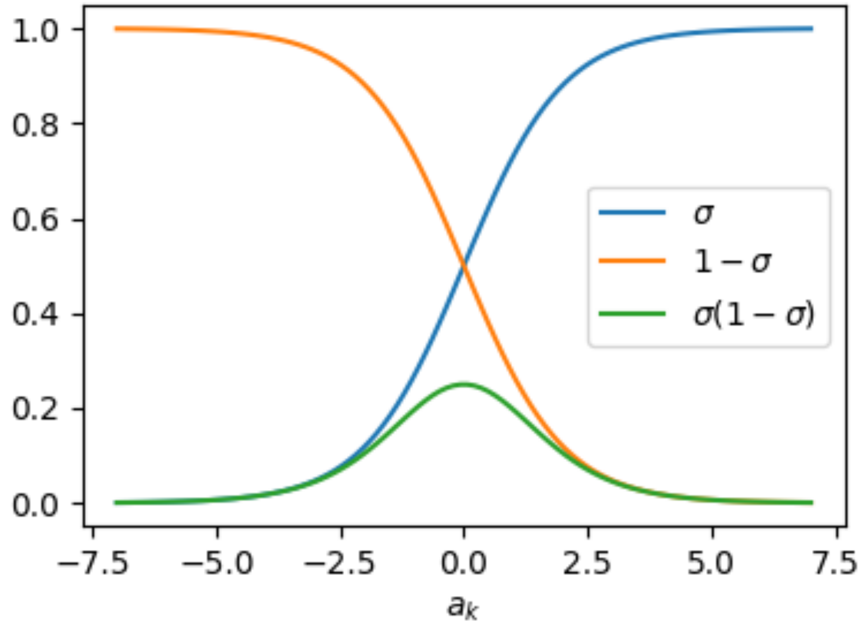
Derivative of the loss for binary classification

For a binary classification network with one output, the output for the n^{th} data point is,

$$\hat{y}_n = \sigma(a_n) = \frac{1}{1 + \exp(-a_n)},$$

where a_n is a linear combination of the activities of the second-last layer.

The derivative of this σ function is $\sigma(1 - \sigma)$.



UNIVERSITY OF WATERLOO

Derivative of the loss for binary classification

The loss for the n^{th} data point is,

$$L = -y_n \ln \hat{y}_n - (1 - y_n) \ln(1 - \hat{y}_n)\,.$$

The derivative of the loss with respect to a_n is,

$$\begin{aligned}\frac{dL}{da_n} &= -\frac{y_n}{\hat{y}_n}\hat{y}_n(1-\hat{y}_n) + \frac{(1-y_n)}{(1-\hat{y}_n)}\hat{y}_n(1-\hat{y}_n) \\ &= -y_n(1-\hat{y}_n) + (1-y_n)\hat{y}_n \\ &= -y_n + y_n\hat{y}_n + \hat{y}_n - y_n\hat{y}_k \\ &= \hat{y}_n - y_n\end{aligned}$$

UNIVERSITY OF WATERLOO

Natural combinations of nonlinearity and loss

Network Purpose	Activation of k[th] output unit	Loss for a single training data point	Gradient
Regression	$y_k = a_k$	$L = \frac{1}{2}\sum_{k=1}^{K}(\hat{y}_k - y_k)^2$	$\frac{\partial L}{\partial a_k} = \hat{y}_k - y_k$
Binary Classification	$y_k = \frac{1}{1 + \exp(-a_k)}$	$L = -\sum_{k=1}^{K} y_k \ln \hat{y}_k + (1 - y_k)\ln(1 - \hat{y}_k)$	$\frac{\partial L}{\partial a_k} = \hat{y}_k - y_k$
Multiclass Classification	$y_k = \frac{\exp(a_k)}{\sum_i \exp(a_i)}$	$L = -\sum_{k=1}^{K} y_k \ln \hat{y}_k$	$\frac{\partial L}{\partial a_k} = \hat{y}_k - y_k$

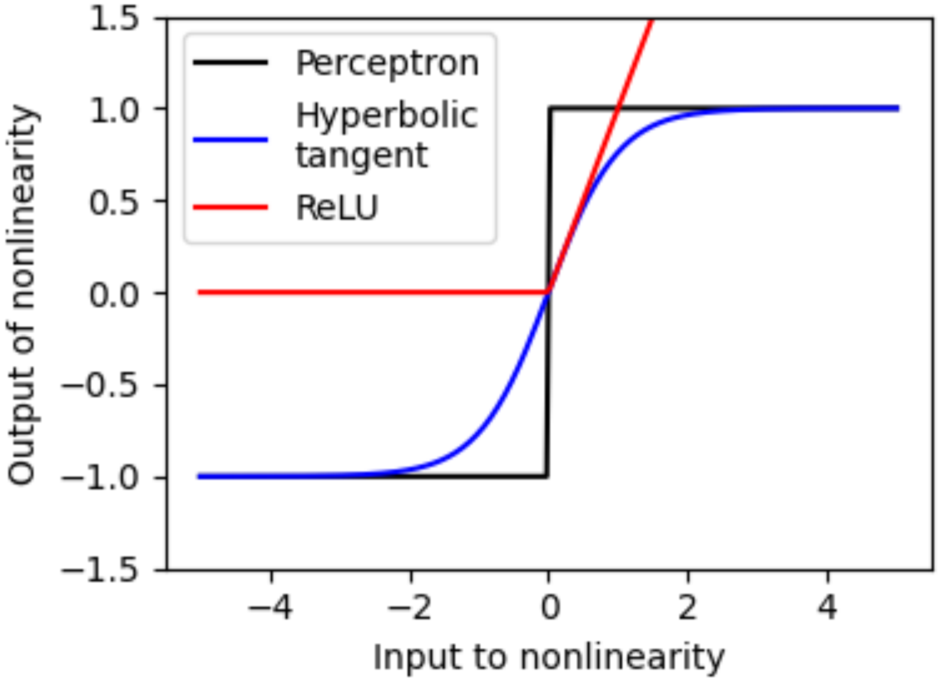
Natural combinations of nonlinearity and loss

- This simple form for the derivative occurs whenever:
  - The target variable is assumed to have a conditional distribution from the exponential family
  - The loss for a training example is the negative log likelihood of the target
  - The activation function is chosen in a certain way (as the inverse of the canonical link function)
- For more detail see Bishop, *Pattern Recognition and Machine Learning*

MULTI-LAYER NETWORKS ARE ALSO CALLED MULTI-LAYER PERCEPTRONS, BUT ARE NOT PERCEPTRONS

Terminology

- Artificial neural networks with at least two layers are often called multi-layer perceptrons
  - E.g., sklearn.neural_network.MLPClassifier
- However, such networks
  - Do not typically use the perceptron activation function (a step function)
  - Do not typically use the perceptron learning rule (they use gradient descent with different losses)
  - Can perform regression as well as classification
- Until about ten years ago, the most common activation function for hidden layers was the hyperbolic tangent, which is like a soft step function, so this made slightly more sense



UNIVERSITY OF WATERLOO

Summary

1. Multi-layer feedforward networks make a series of simple but unintuitive calculations
2. Multi-layer networks are composite functions
3. Multi-layer networks break the curse of dimensionality
4. A two-layer network can closely approximate any continuous function on a bounded domain
5. Networks with different weight matrices can perform identical mappings
6. Classification and regression employ different output and loss functions
7. Multi-layer networks are also called multi-layer perceptrons, but are not perceptrons