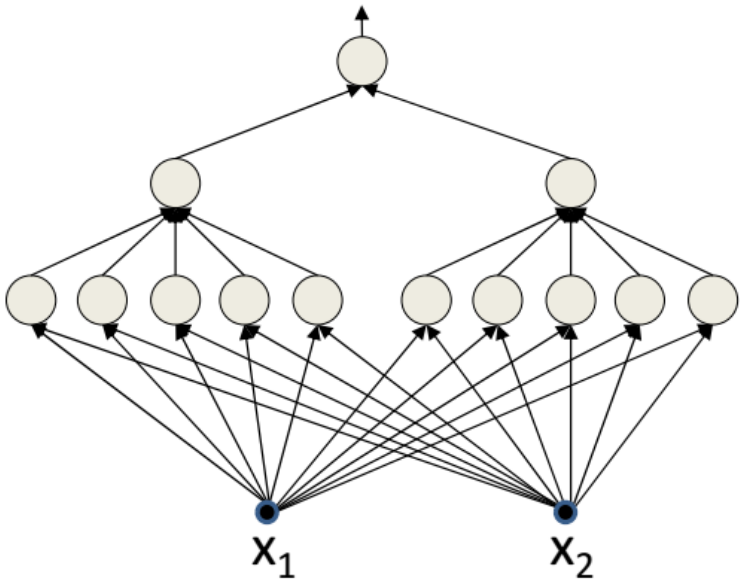
COMP417 Lecture 6

Learning the Network: Backpropagation Part 1

Dr. Hend Dawood

Learning a multilayer perceptron



Training data only specifies input and output of network

Intermediate outputs (outputs of individual neurons) are not specified

- Training this network using the perceptron rule is a combinatorial optimization problem
- We don't know the outputs of the individual intermediate neurons in the network for any training input
- **Must also determine the correct output for *each* neuron for *every* training instance**
- **At least exponential (in inputs) time complexity!!!!!!**

Learning a multilayer perceptron

- Perceptron learning rules cannot directly be used to learn an MLP
  - Exponential complexity of assigning intermediate labels
    - Even worse when classes are not actually separable

Learning a multilayer perceptron

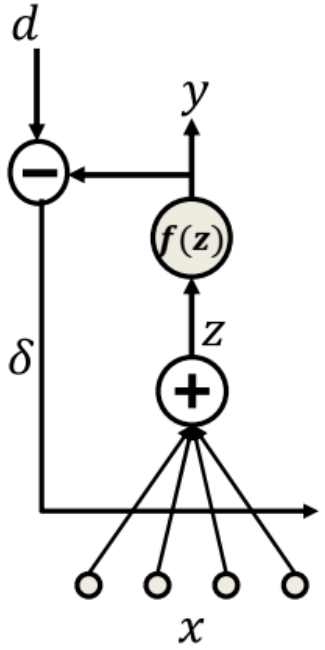
- "Learning" a network = learning the weights and biases to compute a target function
  - Will require a network with sufficient "capacity"
- In practice, we learn networks by "fitting" them to match the input-output relation of "training" instances drawn from the target function
- A linear decision boundary can be learned by a single perceptron (with a threshold-function activation) in linear time if classes are linearly separable
- Non-linear decision boundaries require networks of perceptrons
- Training an MLP with threshold-function activation perceptrons will require knowledge of the input-output relation for every training instance, for *every* perceptron in the network
  - These must be determined as part of training
  - For threshold activations, this is an NP-complexity combinatorial optimization problem

Generalized delta rule

- For any differentiable activation function the following update rule is used

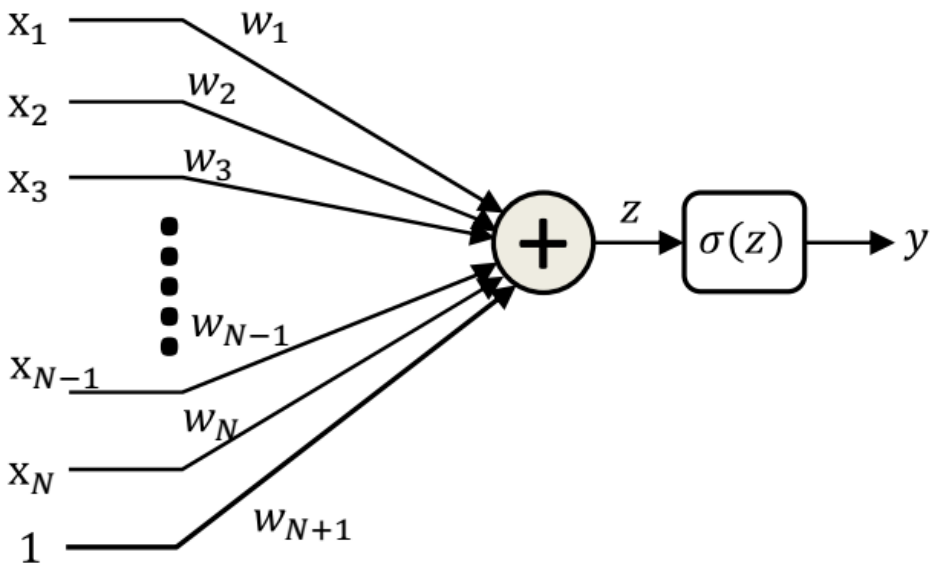
$$\delta = d - y$$

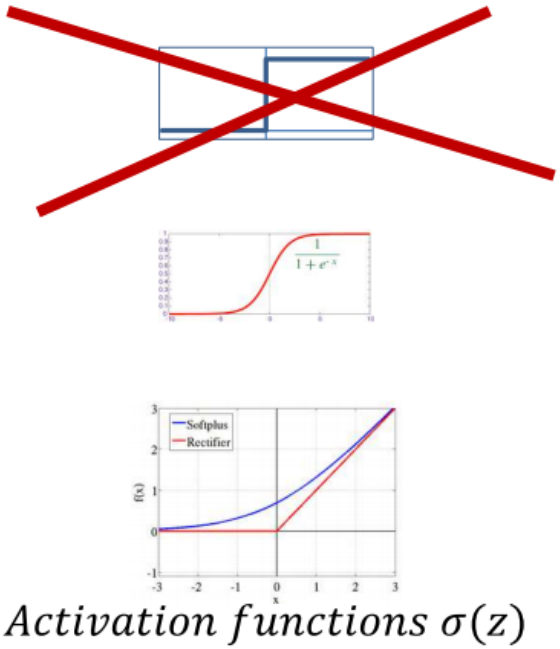
$$w_i = w_i + \eta \delta f'(z) x_i$$



- This is the famous Widrow-Hoff update rule
  - Lookahead: Note that this is *exactly* backpropagation in multilayer nets if we let $f(z)$ represent the entire network between z and y
- It is possibly the most-used update rule in machine learning and signal processing
  - Variants of it appear in almost every problem

Differentiable activation

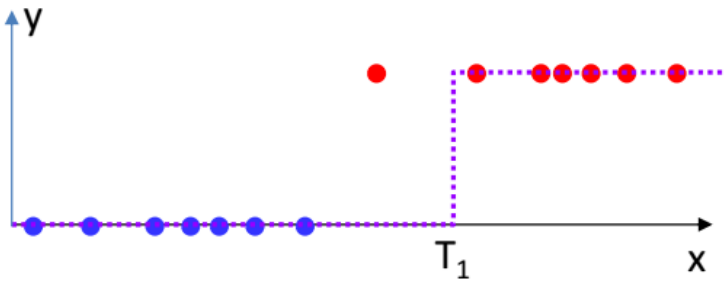


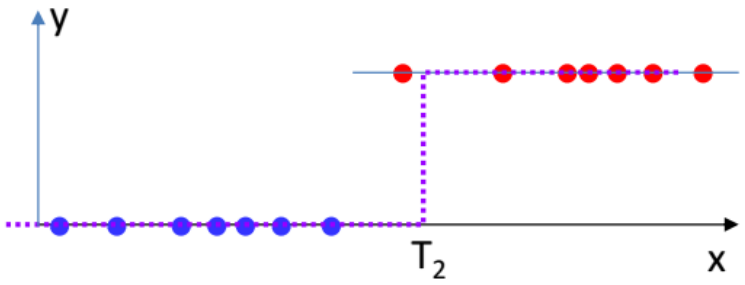


Activation functions $\sigma(z)$

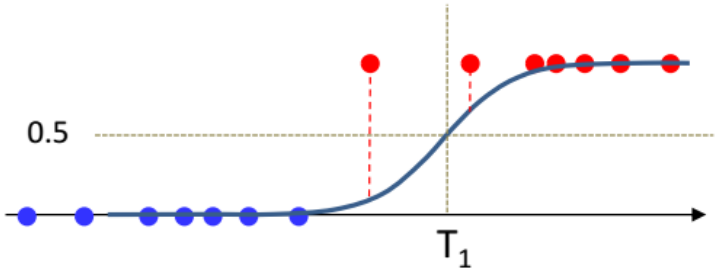
- Let's make the neuron differentiable, *with non-zero derivatives over much of the input space*
  - Small changes in weight can result in non-negligible changes in output
  - This enables us to estimate the parameters using gradient descent techniques..

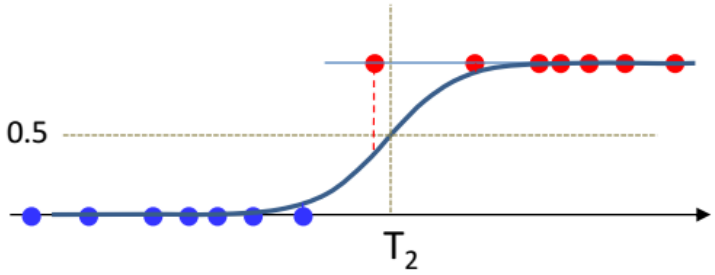
Differentiable Mismatch function





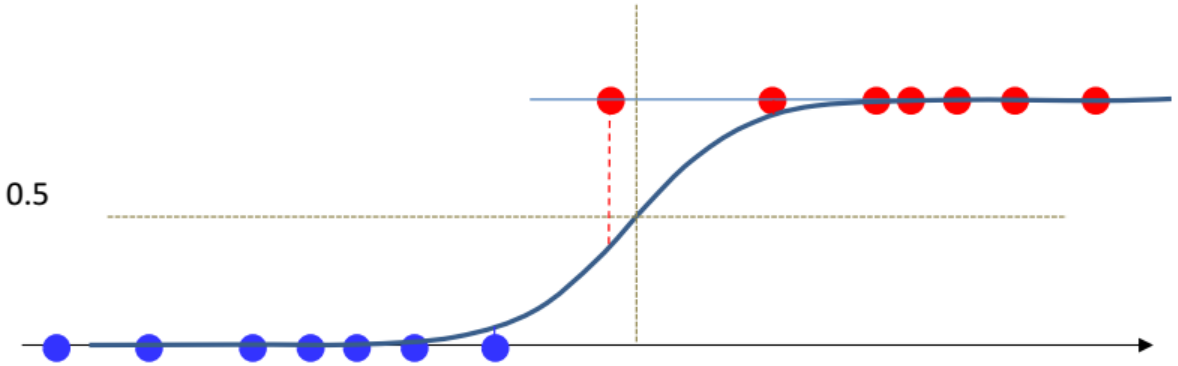
- Threshold activation: shifting the threshold from T_1 to T_2 does not change classification error
  - Does not indicate if moving the threshold left was good or not





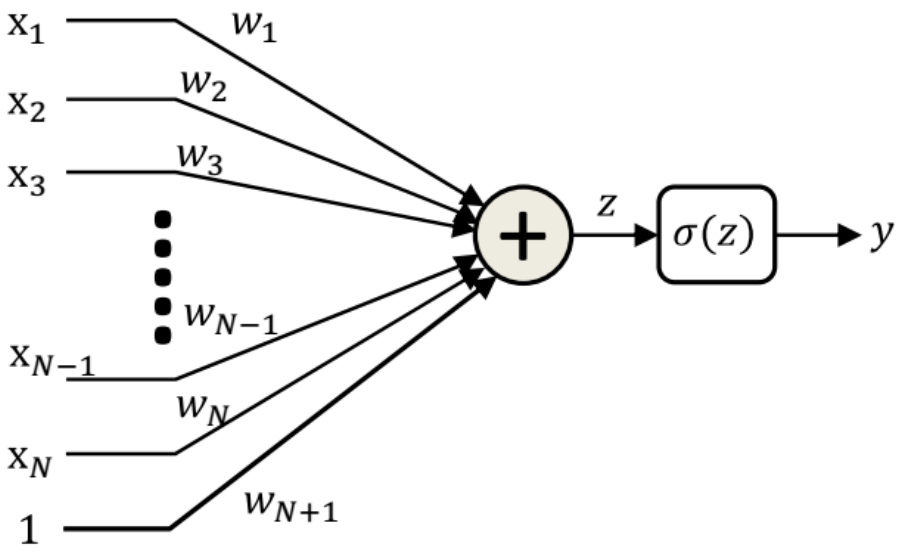
- Smooth, continuously varying activation: Classification based on whether the output is greater than 0.5 or less
  - Quantify *how much* the output differs from the desired target value (0 or 1)
  - Moving the function left or right changes this quantity, even if the classification error itself doesn't change

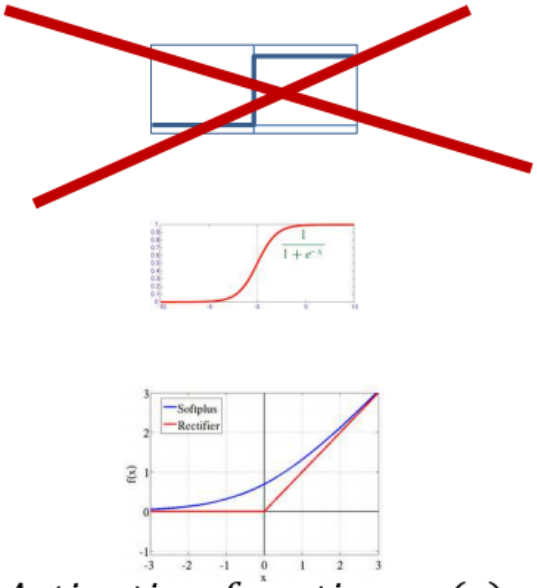
The two key requirements for learnability



- Continuously varying activation
  - Differentiable
- Continuously varying error function
  - Also differentiable

Continuous Activations

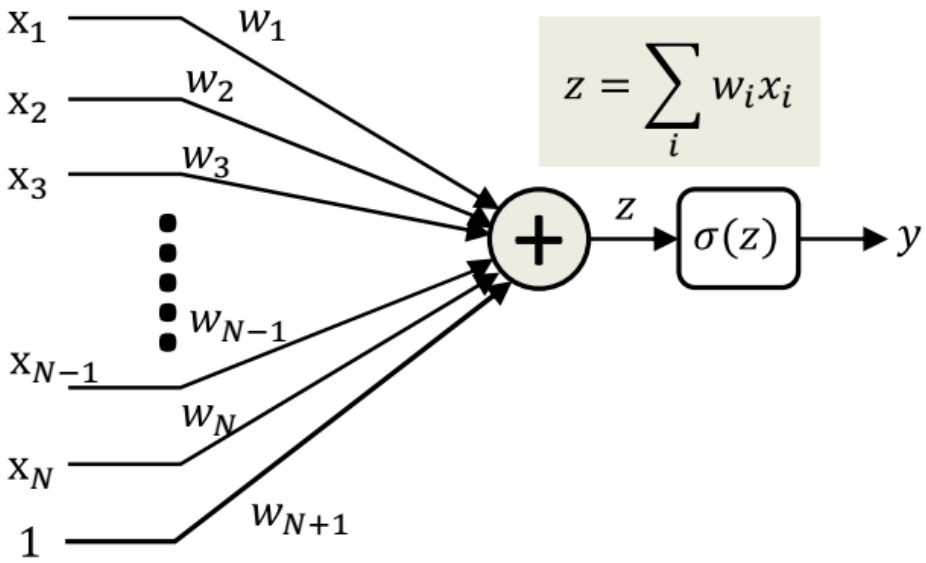




Activation functions $\sigma(z)$

- Replace the threshold activation with continuous graded activations
  - E.g. RELU, softplus, sigmoid etc.
- The activations are *differentiable* almost everywhere
  - Have derivatives almost everywhere
  - And have "subderivatives" at non-differentiable corners
    - Bounds on the derivative that can substitute for derivatives in our setting

Perceptrons with differentiable activation functions



$$z = \sum_i w_i x_i$$

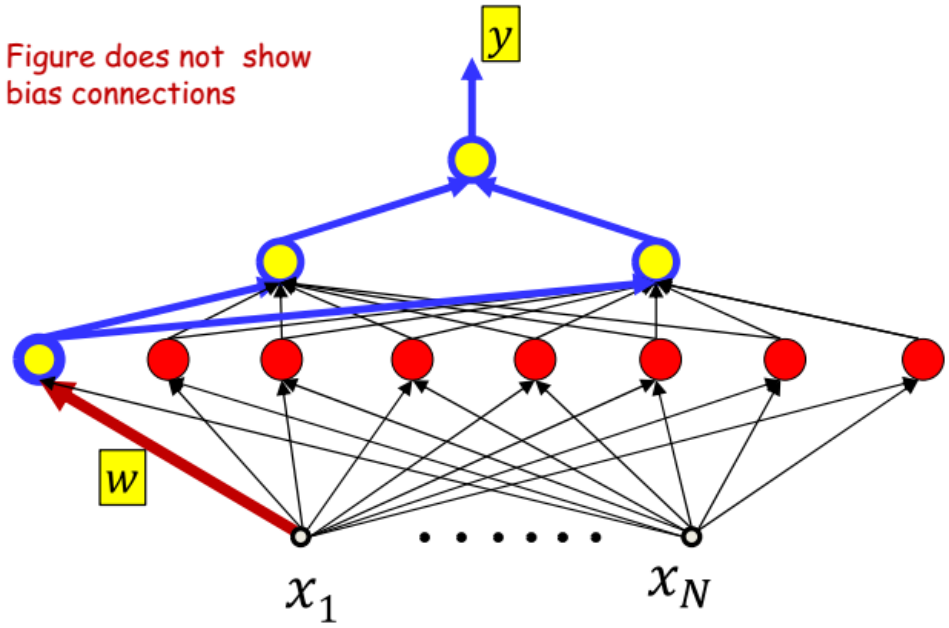
$$\frac{dy}{dz} = \sigma'(z)$$

$$\frac{dy}{dw_i} = \frac{dy}{dz}\frac{dz}{dw_i} = \sigma'(z)x_i$$

$$\frac{dy}{dx_i} = \frac{dy}{dz}\frac{dz}{dx_i} = \sigma'(z)w_i$$

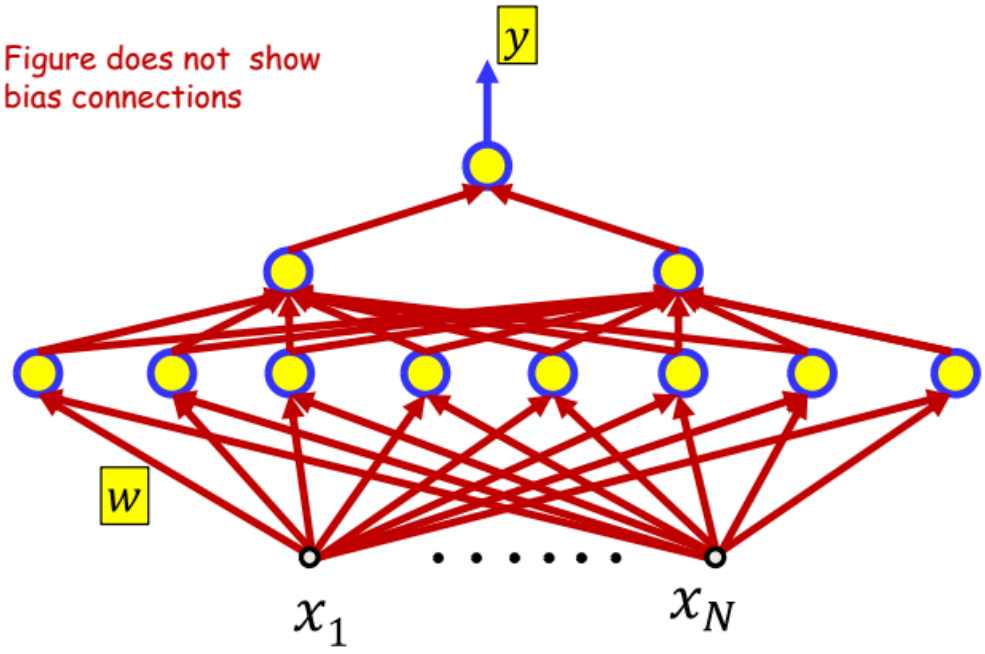
- **$\sigma(z)$ is a differentiable function of z**
  - $\frac{d\sigma(z)}{dz}$ is well-defined and finite for all z
- **Using the chain rule, y is a differentiable function of both inputs x_i and weights w_i**
- **This means that we can compute the change in the output for *small* changes in either the input or the weights**

Overall network is differentiable



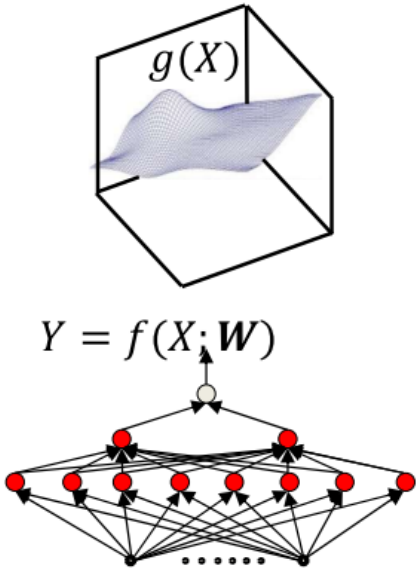
- Every individual perceptron is differentiable w.r.t its inputs and its weights (including "bias" weight)
  - Small changes in the parameters result in measurable changes in output
- Using the chain rule can compute how small perturbations of a parameter change the output of the network
  - The network output is differentiable with respect to the parameter

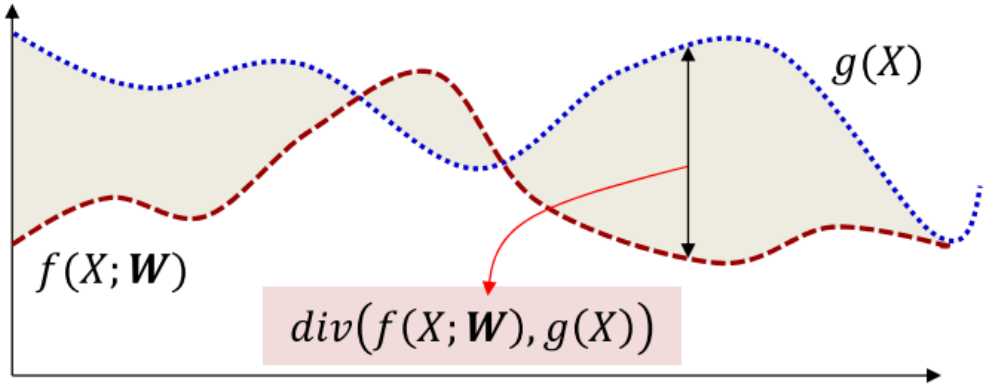
Overall function is differentiable



- By extension, the overall function is differentiable w.r.t every parameter in the network
  - We can compute how small changes in the parameters change the output
    - For non-threshold activations the derivative are finite and generally non-zero
- We will derive the actual derivatives using the chain rule later

The error function

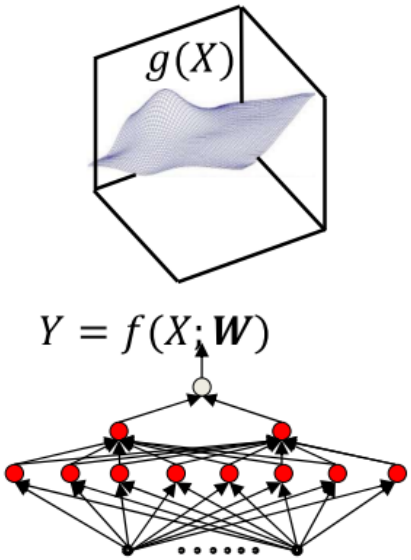


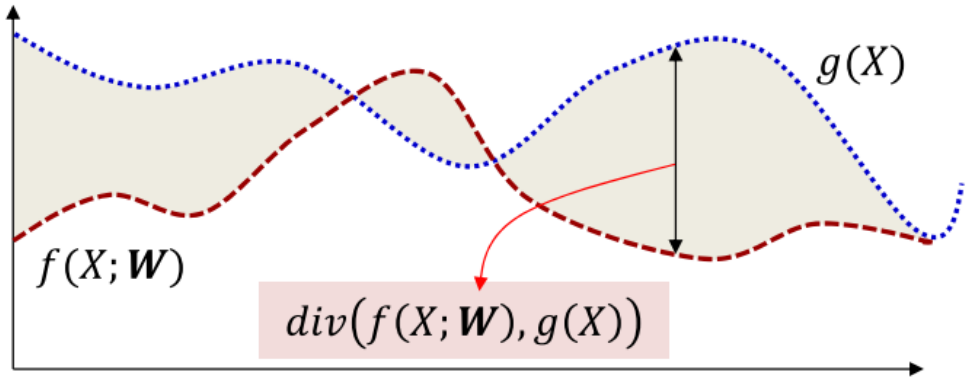


- Define a *divergence* function $div\big(f(X; \boldsymbol{W}), g(X)\big)$ with the following properties
  - $div\big(f(X; \boldsymbol{W}), g(X)\big) = 0$ if $f(X; \boldsymbol{W}) = g(X)$
  - $div\big(f(X; \boldsymbol{W}), g(X)\big) > 0$ if $f(X; \boldsymbol{W}) \neq g(X)$
  - $div(f, g)$ is differentiable with respect to f

- The divergence function quantifies mismatch between the network output and target function
  - For classification, this is usually not the classification error but a proxy to it

How to learn a network



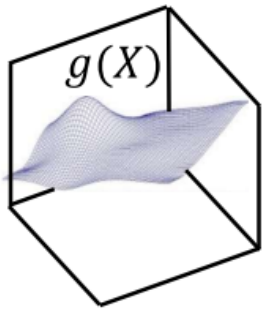


- When $f(X; \boldsymbol{W})$ has the capacity to exactly represent $g(X)$

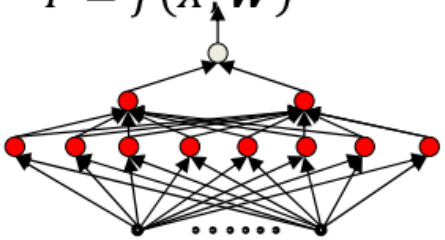
$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}} \int_X div\big(f(X; W), g(X)\big) dX$$

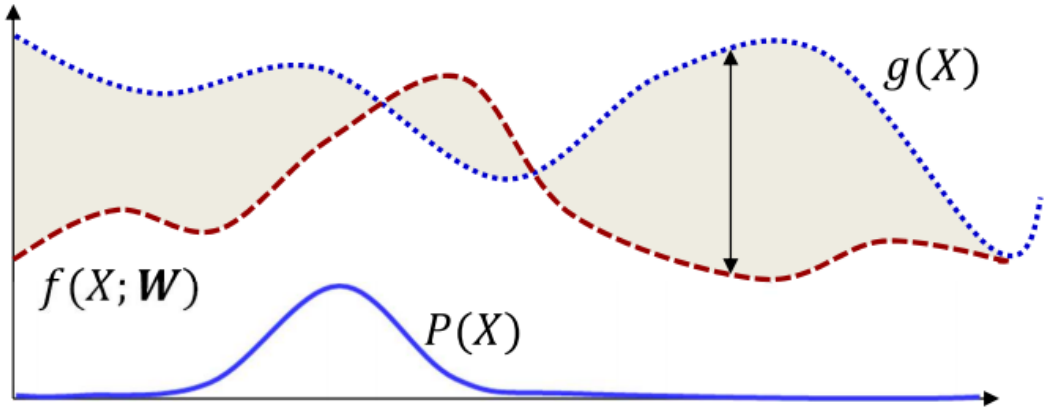
- div() is a divergence function that goes to zero when $f(X; W) = g(X)$

Minimizing expected divergence



$Y = f(X; \boldsymbol{W})$



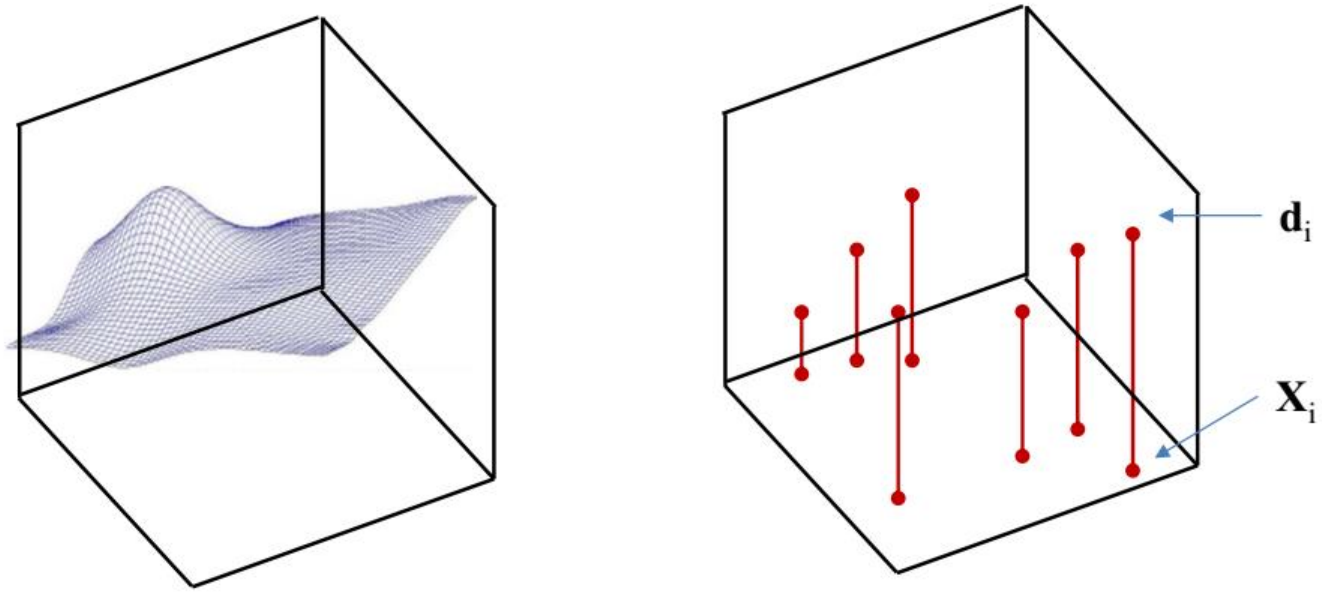


- More generally, assuming X is a random variable

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}} \int_X div\big(f(X;W), g(X)\big) P(X) dX$$

$$= \underset{W}{\operatorname{argmin}}\, E\big[div\big(f(X;W), g(X)\big)\big]$$

The Empirical risk



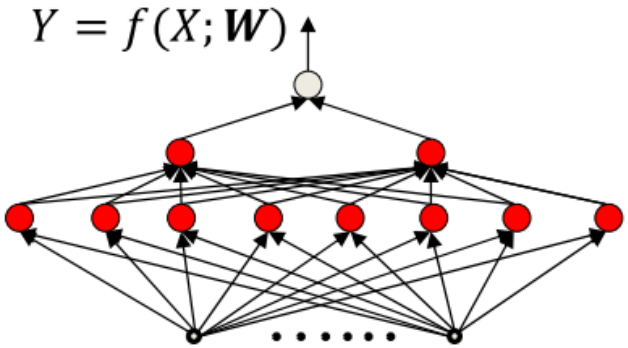
- The *expected* divergence (or risk) is the average divergence over the entire input space

$$E\big[div\big(f(X;W),g(X)\big)\big] = \int_X div\big(f(X;W),g(X)\big)P(X)dX$$

- The *empirical estimate* of the expected risk is the *average* divergence over the samples

$$E\big[div\big(f(X;W),g(X)\big)\big] \approx \frac{1}{N}\sum_{i=1}^{N} div(f(X_i;W),d_i)$$

Training the network: Empirical Risk Minimization



- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$
  - Quantification of error on the i[th] instance: $div(f(X_i; W), d_i)$
  - Empirical average divergence (Empirical Risk) on all training data:

$$Loss(W) = \frac{1}{N} \sum_i div(f(X_i; W), d_i)$$

- Estimate the parameters to minimize the empirical estimate of expected divergence (empiricial risk)

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}}\, Loss(W)$$

  - I.e. minimize the *empirical risk* over the drawn samples

Empirical Risk Minimization

$$Y = f(X; \boldsymbol{W})$$

Note : Its really a measure of error, but using standard terminology, we will call it a "Loss"

Note 2: The empirical risk $Loss(W)$ is only an empirical approximation to the true risk $E\left[div\left(f(X;W), g(X)\right)\right]$ which is our *actual* minimization objective

Note 3: For a *given training set* the loss is only a function of W

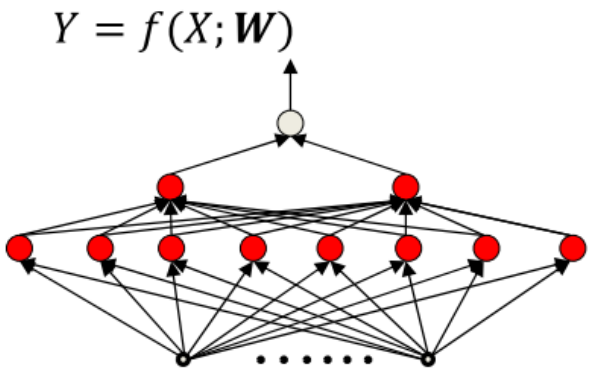
$$Loss(W) = \frac{1}{N}\sum_{i} div(f(X_i; W), d_i)$$

- Estimate the parameters to minimize the empirical estimate of expected error

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}}\, Loss(W)$$

  - I.e. minimize the *empirical error* over the drawn samples

Empirical Risk Minimization



This is an instance of function minimization (optimization)

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$
  - Error on the i-th instance: $div(f(X_i; W), d_i)$
  - Empirical average error on all training data:

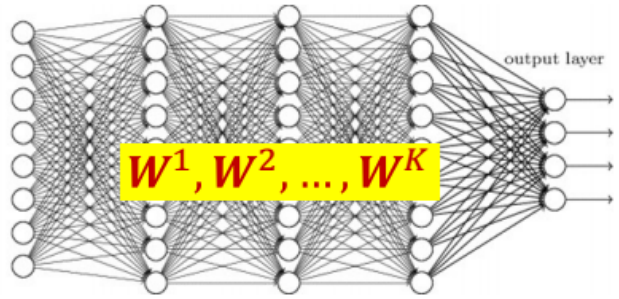
$$Loss(W) = \frac{1}{T}\sum_i div(f(X_i; W), d_i)$$

- Estimate the parameters to minimize the empirical estimate of expected error

$$\widehat{\boldsymbol{W}} = \underset{W}{\operatorname{argmin}}\, Loss(W)$$

  - I.e. minimize the *empirical error* over the drawn samples

ERM for neural networks



Actual output of network:

$$Y_i = net\left(X_i; \left\{w_{i,j}^k \forall i,j,k\right\}\right) = net(X_i; W^1, W^2, \ldots, W^K)$$

Desired output of network: d_i

Error on i-th training input: $Div(Y_i, d_i; W^1, W^2, \ldots, W^K)$

Average training error(loss):

$$Loss(W^1, W^2, \ldots, W^K) = \frac{1}{N}\sum_{i=1}^{N} Div(Y_i, d_i; W^1, W^2, \ldots, W^K)$$

- What is the exact form of Div()? More on this later

- Optimize network parameters to minimize the total error over all training inputs

Problem Statement

- Given a training set of input-output pairs $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_N, \boldsymbol{d}_N)$
- Minimize the following function

$$Loss(W) = \frac{1}{N}\sum_i div(f(X_i; W), d_i)$$

w.r.t W

- This is problem of function minimization
  - An instance of optimization

Problem Setup

Problem Setup: Things to define

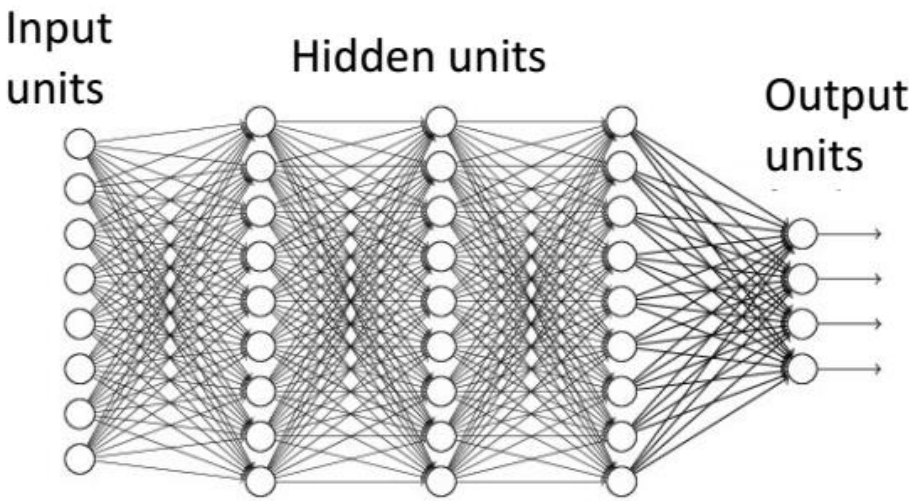
- Given a training set of input-output pairs
  $(\boldsymbol{X}_1, \boldsymbol{d}_1), (\boldsymbol{X}_2, \boldsymbol{d}_2), \ldots, (\boldsymbol{X}_T, \boldsymbol{d}_T)$
- What are these input-output pairs?

$$Loss(W) = \frac{1}{T}\sum_i div(f(X_i; W), d_i)$$

What is the divergence div()?

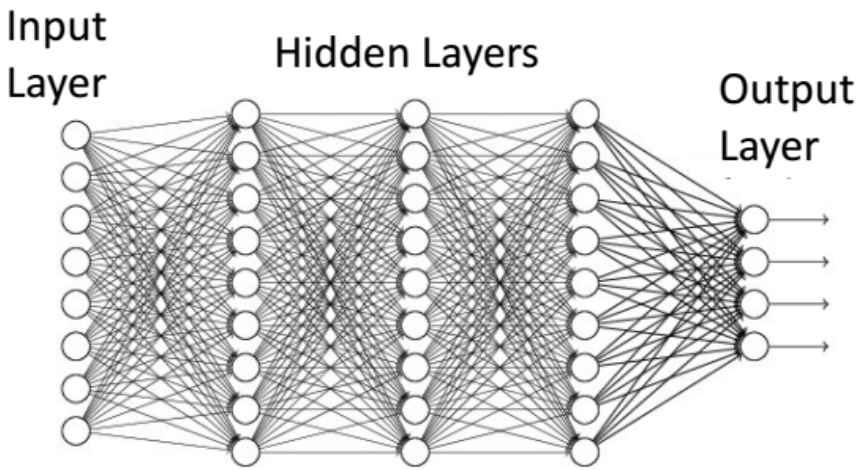
What is f() and what are its parameters W?

What is f()? Typical network



- Multi-layer perceptron
- A *directed* network with a set of inputs and outputs
  - No loops

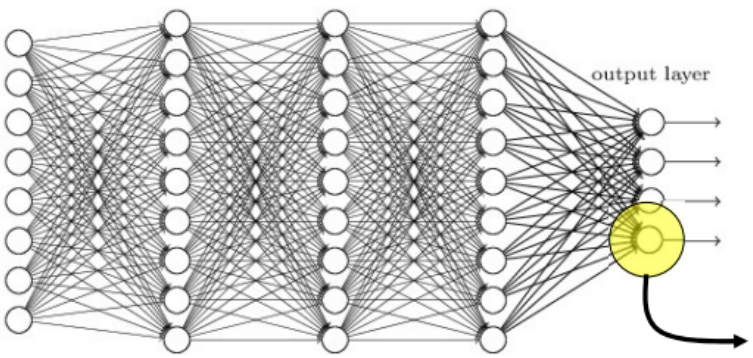
What is f()? Typical network

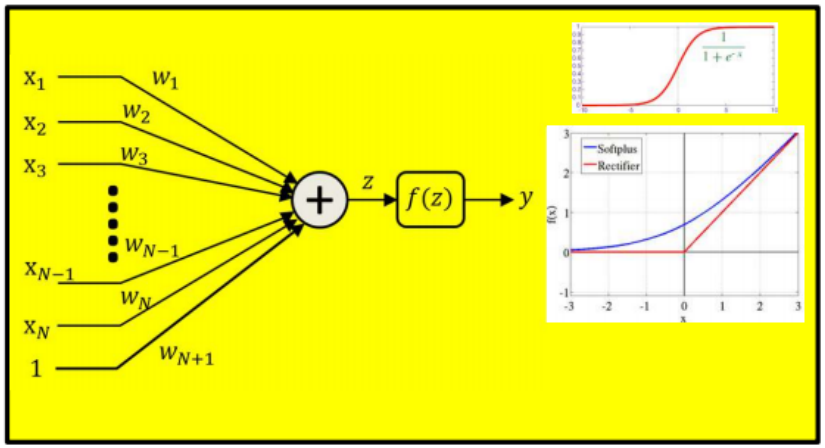


- We assume a "layered" network for simplicity
  - Each "layer" of neurons only gets inputs from the earlier layer(s) and outputs signals only to later layer(s)
  - We will refer to the inputs as the ***input layer***
    - No neurons here – the "layer" simply refers to inputs
  - We refer to the outputs as the ***output layer***
  - Intermediate layers are ***"hidden" layers***

What is f()? Typical network

The individual neurons





- Individual neurons operate on a set of inputs and produce a single output
  - **Standard setup:** A continuous activation function applied to an affine function of the inputs

$$y = f\left(\sum_i w_i x_i + b\right)$$

  - More generally: *any* differentiable function

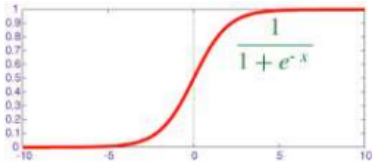
$$y = f(x_1, x_2, \ldots, x_N; W)$$

We will assume this unless otherwise specified

Parameters are weights w_i and bias b

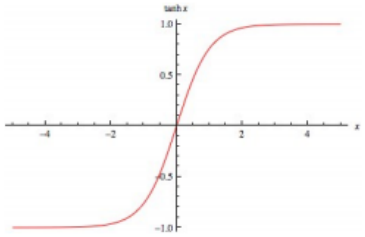
What is f()? Typical network

Activations and their derivatives



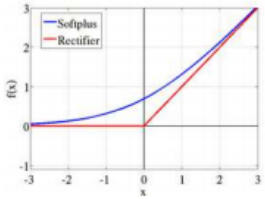
$$f(z) = \frac{1}{1 + \exp(-z)}$$

$$f'(z) = f(z)(1 - f(z))$$



$$f(z) = \tanh(z)$$

$$f'(z) = (1 - f^2(z))$$



$$f(z) = \begin{cases} z, & z \geq 0 \\ 0, & z < 0 \end{cases}$$

[*]

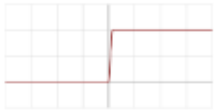
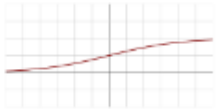
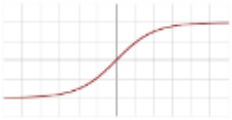
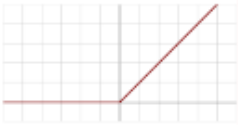
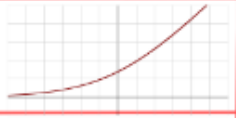
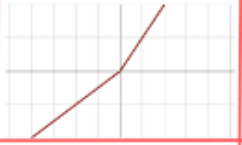
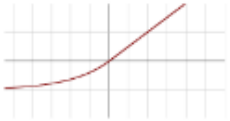
$$f'(z) = \begin{cases} 1, z \geq 0 \\ 0, z < 0 \end{cases}$$

$$f(z) = \log(1 + \exp(z))$$

$$f'(z) = \frac{1}{1 + \exp(-z)}$$

- Some popular activation functions and their derivatives

What is f()? Typical network

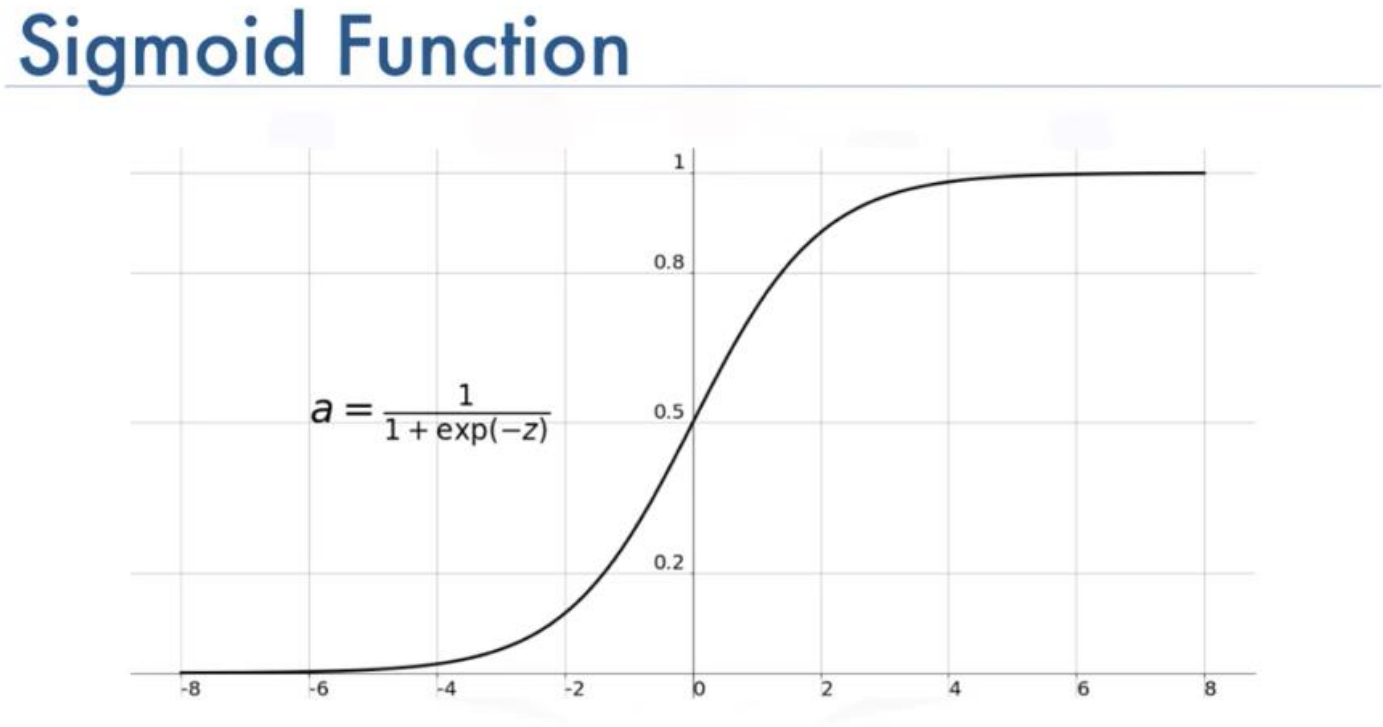
ACTIVATION FUNCTION	PLOT	EQUATION	DERIVATIVE	RANGE
Linear		$f(x) = x$	$f'(x) = 1$	$(-\infty, \infty)$
Binary Step		$f(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x \neq 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$\{0, 1\}$
Sigmoid		$f(x) = \sigma(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$	$(0, 1)$
Hyperbolic Tangent(tanh)		$f(x) = \tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$	$f'(x) = 1 - f(x)^2$	$(-1, 1)$
Rectified Linear Unit(ReLU)		$f(x) = \begin{cases} 0 & \text{if } x<0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$[0, \infty)$
Softplus		$f(x) = \ln(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$	$(0, \infty)$
Leaky ReLU		$f(x) = \begin{cases} 0.01x & \text{if } x < 0 \\ x & \text{if } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0.01 & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ \text{undefined} & \text{if } x = 0 \end{cases}$	$(-\infty, \infty)$
Exponential Linear Unit(ELU)		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{if } x \leq 0 \\ x & \text{if } x > 0 \end{cases}$	$f'(x) = \begin{cases} \alpha e^x & \text{if } x < 0 \\ 1 & \text{if } x > 0 \\ 1 & \text{if } x = 0 \text{ and } \alpha = 1 \end{cases}$	$(-\alpha, \infty)$

What is f()? Typical network

Sigmoid / Logistic Activation Function

This function takes any real value as input and outputs values in the range of 0 to 1.

The larger the input (more positive), the closer the output value will be to 1.0, whereas the smaller the input (more negative), the closer the output will be to 0.0, as shown below.



What is f()? Typical network

Mathematically it can be represented as:

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

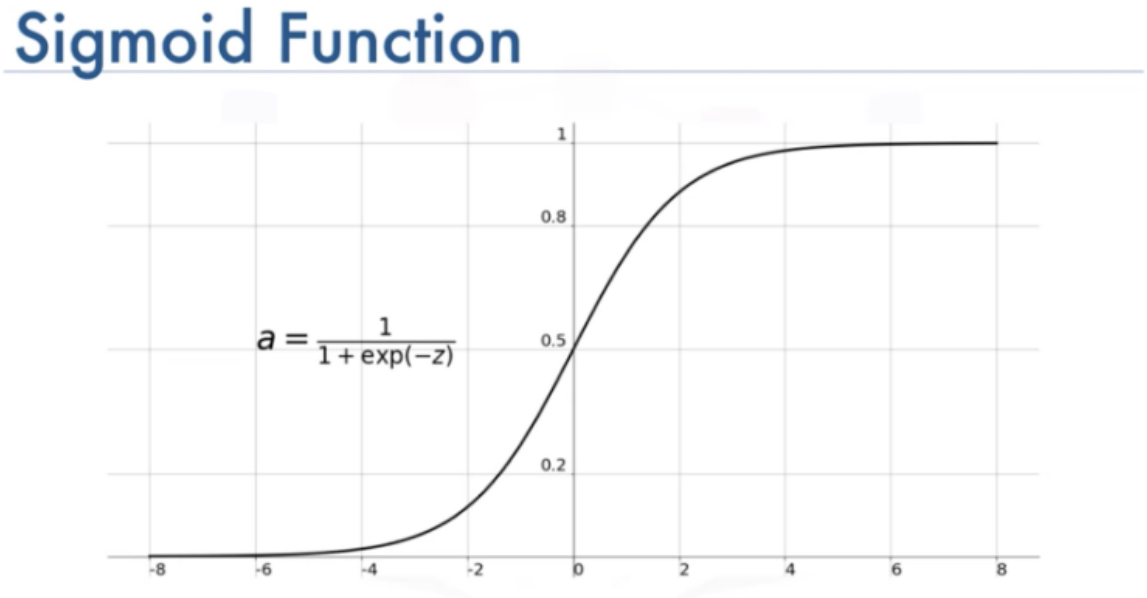
Here's why sigmoid/logistic activation function is one of the most widely used functions:

- It is commonly used for models where we have to predict the probability as an output. Since probability of anything exists only between the range of 0 and 1, sigmoid is the right choice because of its range.
- The function is differentiable and provides a smooth gradient, i.e., preventing jumps in output values. This is represented by an S-shape of the sigmoid activation function.

What is f()? Typical network

Softmax Function

Before exploring the ins and outs of the Softmax activation function, we should focus on its building block—the sigmoid/logistic activation function that works on calculating probability values.



The output of the sigmoid function was in the range of 0 to 1, which can be thought of as probability.

But—

This function faces certain problems.

What is f()? Typical network

Let's suppose we have five output values of 0.8, 0.9, 0.7, 0.8, and 0.6, respectively. How can we move forward with it?

The answer is: We can't.

The above values don't make sense as the sum of all the classes/output probabilities should be equal to 1.

You see, the Softmax function is described as a combination of multiple sigmoids.

It calculates the relative probabilities. Similar to the sigmoid/logistic activation function, the SoftMax function returns the probability of each class.

It is most commonly used as an activation function for the last layer of the neural network in the case of multi-class classification.

Mathematically it can be represented as:

$$softmax(z_i) = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

Softmax Function

What is f()? Typical network

Let's go over a simple example together.

Assume that you have three classes, meaning that there would be three neurons in the output layer. Now, suppose that your output from the neurons is [1.8, 0.9, 0.68].

Applying the softmax function over these values to give a probabilistic view will result in the following outcome: [0.58, 0.23, 0.19].

The function returns 1 for the largest probability index while it returns 0 for the other two array indexes. Here, giving full weight to index 0 and no weight to index 1 and index 2. So the output would be the class corresponding to the 1st neuron(index 0) out of three.

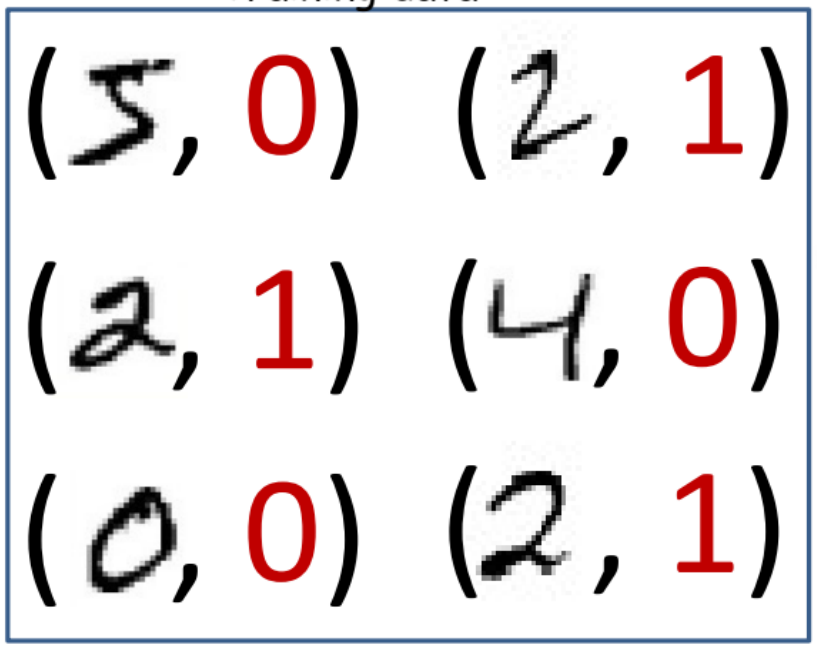
You can see now how softmax activation function make things easy for multi-class classification problems.

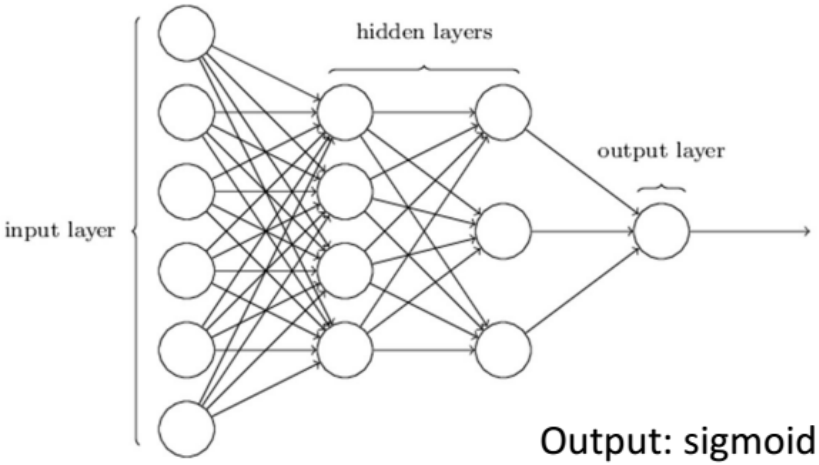
Inputs and outputs: Typical Problem Statement

- We are given a number of "training" data instances
- E.g. images of digits, along with information about which digit the image represents
- Tasks:
  - Binary recognition: Is this a "2" or not
  - Multi-class recognition: Which digit is this?

Typical Problem statement: binary classification

Training data



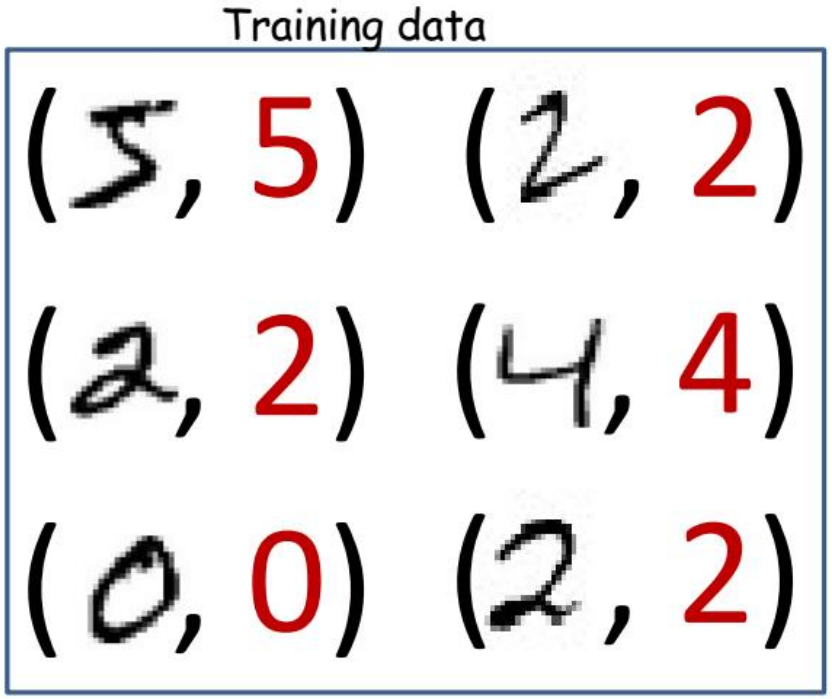


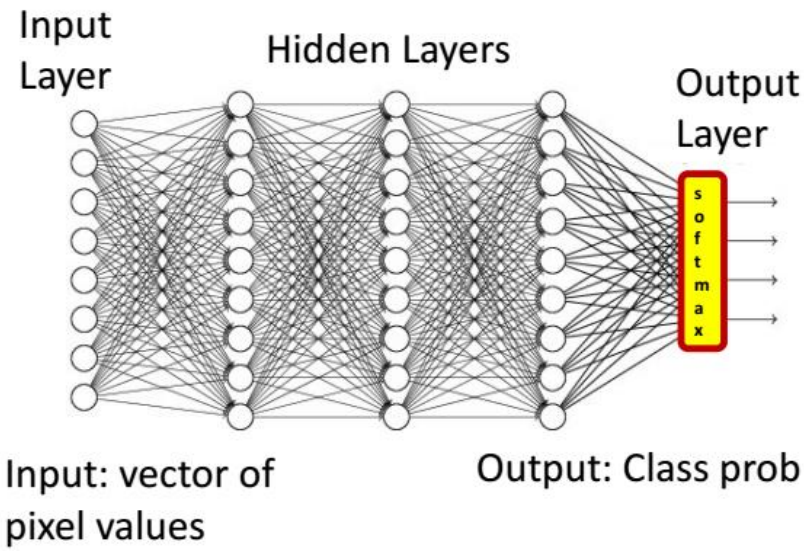
Output: sigmoid

Input: vector of pixel values

- Given, many positive and negative examples (training data),
  - learn all weights such that the network does the desired job

Typical Problem statement: multiclass classification

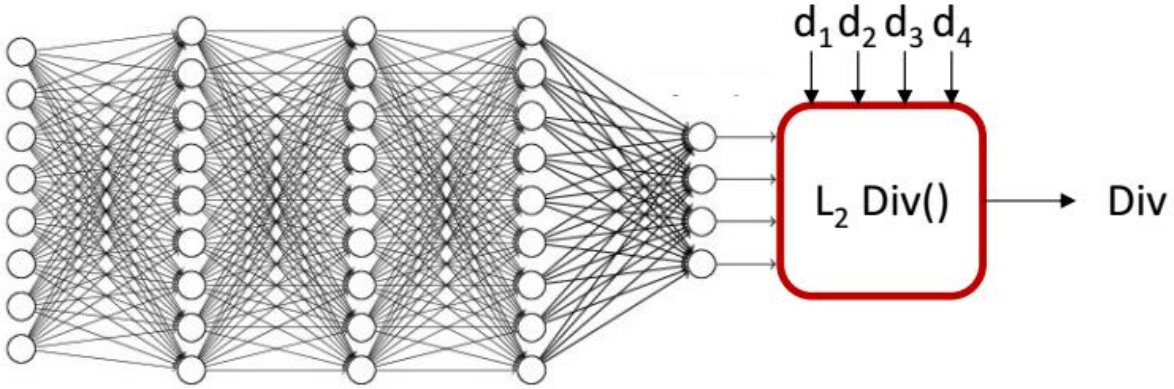




- Given, many positive and negative examples (training data),
  - learn all weights such that the network does the desired job

What is the divergence div()?

Examples of divergence functions



- For real-valued output vectors, the (scaled) L_2 divergence is popular

$$Div(Y,d) = \frac{1}{2}\|Y-d\|^2 = \frac{1}{2}\sum_i (y_i - d_i)^2$$

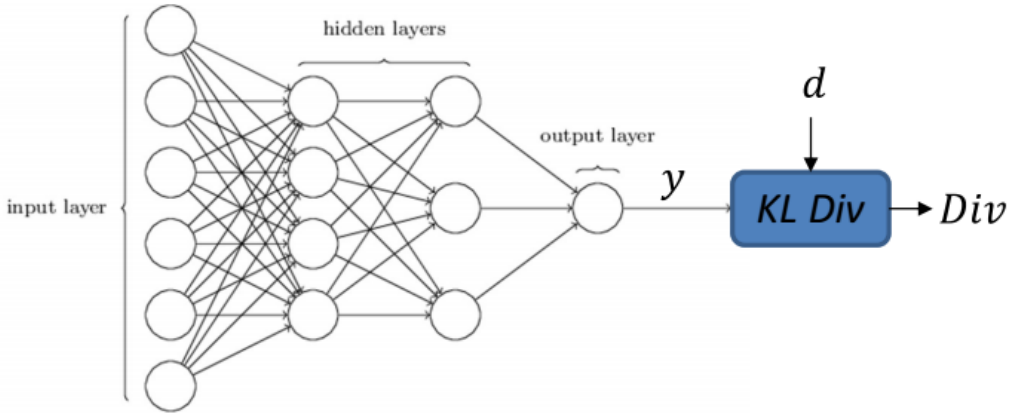
  - Squared Euclidean distance between true and desired output
  - Note: this is differentiable

$$\frac{dDiv(Y,d)}{dy_i} = (y_i - d_i)$$

$$\nabla_Y Div(Y,d) = [y_1 - d_1, y_2 - d_2, \dots]$$

What is the divergence div()?

For binary classifier



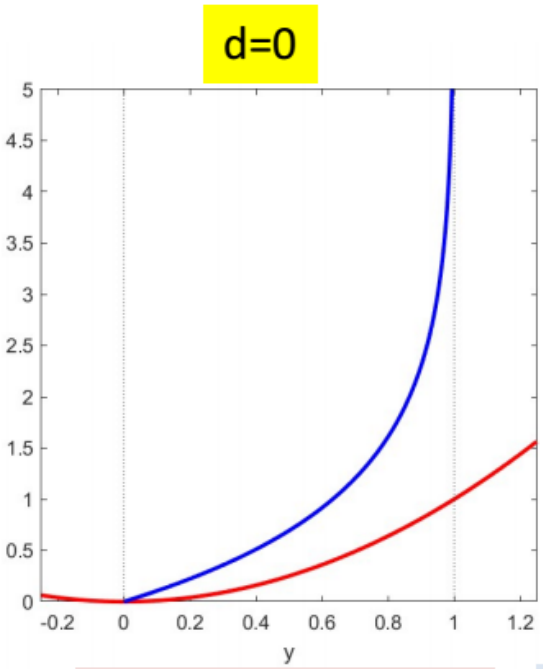
- For binary classifier with scalar output, $Y \in (0,1)$, d is $0/1$, the Kullback Leibler (KL) divergence between the probability distribution $[Y, 1-Y]$ and the ideal output probability $[d, 1-d]$ is popular

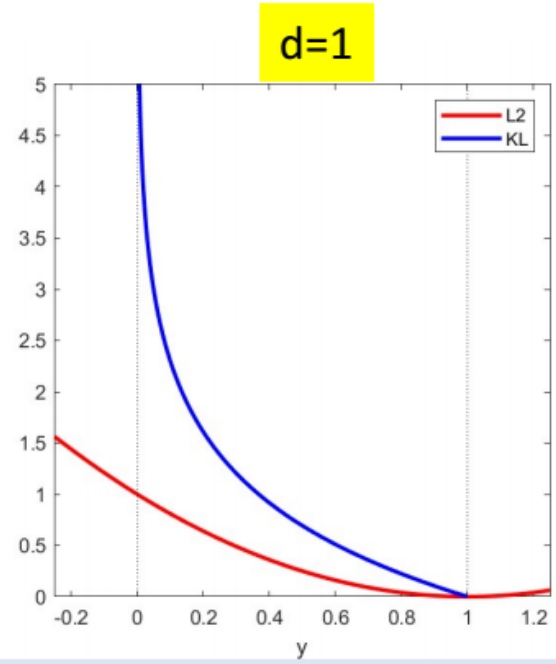
$$Div(Y,d) = -dlogY - (1-d)\log(1-Y)$$

  - Minimum when $d = Y$

What is the divergence div()?

KL vs L2





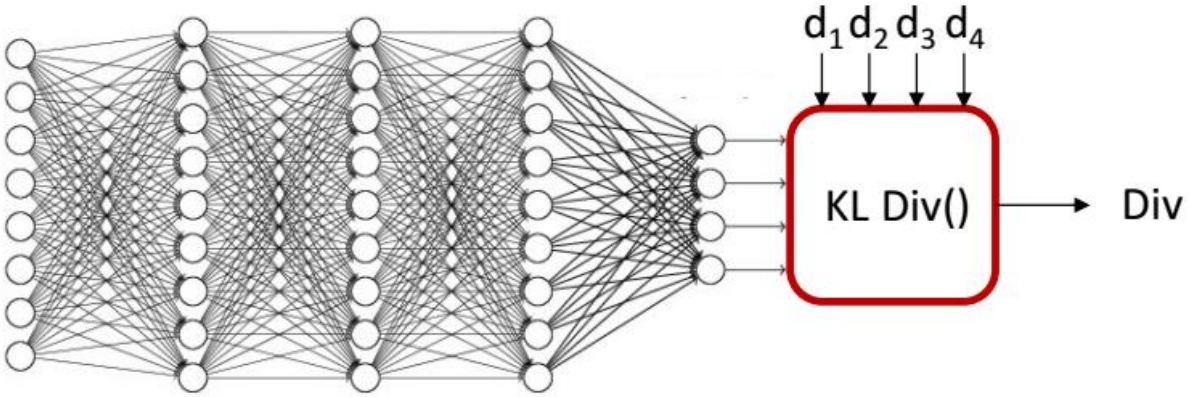
$L2(Y,d) = (y-d)^2$

$KL(Y,d) = -dlogY - (1-d)\log(1-Y)$

- Both KL and L2 have a minimum when y is the target value of d
- KL rises much more steeply away from d
  - Encouraging faster convergence of gradient descent

What is the divergence div()?

For multi-class classification



- Desired output d is a one hot vector [0 0 … 1 … 0 0 0] with the 1 in the c-th position (for class c)
- Actual output will be probability distribution $[y_1, y_2, \dots]$
- The KL divergence between the desired one-hot output and actual output:

$$Div(Y, d) = \sum_i d_i \log \frac{d_i}{y_i} = \sum_i d_i \log d_i - \sum_i d_i \log y_i = -\log y_c$$

Story so far

- Neural nets are universal approximators
- Neural networks are trained to approximate functions by adjusting their parameters to minimize the average divergence between their actual output and the desired output at a set of "training instances"
  - Input-output samples from the function to be learned
  - The average divergence is the "Loss" to be minimized
- To train them, several terms must be defined
  - The network itself
  - The manner in which inputs are represented as numbers
  - The manner in which outputs are represented as numbers
    - As numeric vectors for real predictions
    - As one-hot vectors for classification functions
  - The divergence function that computes the error between actual and desired outputs
    - L2 divergence for real-valued predictions
    - KL divergence for classifiers