CSC 561: Neural Networks and Deep Learning

Learning (part 1)

Marco Alvarez

Department of Computer Science and Statistics
University of Rhode Island

Spring 2025

THINK BIG WE DO℠

Neural Networks
Learning the network: Part 1

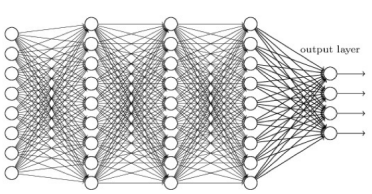
11-785, Spring 2025

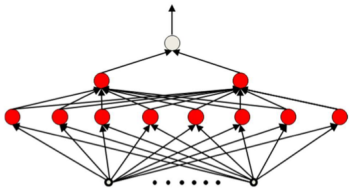
Lecture 3

Topics for the day

- The problem of learning
- The perceptron rule for learning individual perceptrons
  - And its inapplicability to multi-layer perceptrons
- Greedy solutions for classification networks: ADALINE and MADALINE
- Learning through Empirical Risk Minimization
- Intro to function optimization and gradient descent

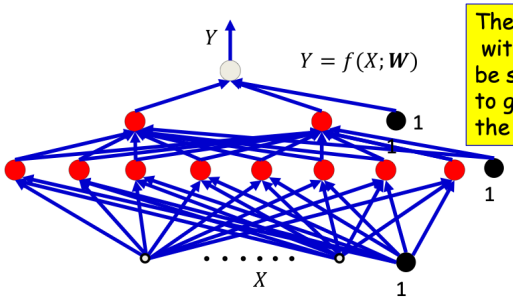
First: the structure of the network





- We will assume a *feed-forward* network
  - No loops: Neuron outputs do not feed back to their inputs directly or indirectly
  - Loopy networks are a future topic
- **Part of the design of a network: The architecture**
  - How many layers/neurons, which neuron connects to which and how, etc.
- For now, assume the architecture of the network is capable of representing the needed function

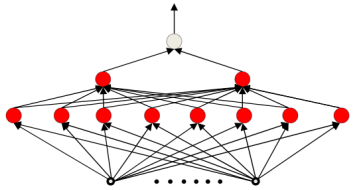
What we learn: The parameters of the network

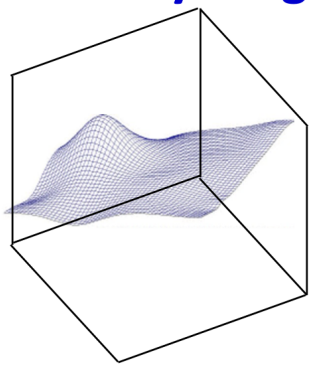


The network is a function f() with parameters W which must be set to the appropriate values to get the desired behavior from the net

- **Given:** the architecture of the network
- The parameters of the network: The weights and biases
  - The weights associated with the blue arrows in the picture
- ***Learning the network* : Determining the values of these parameters such that the network computes the desired function**

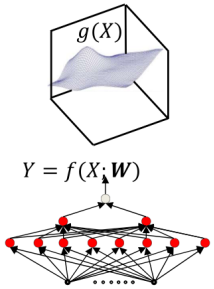
The MLP *can* represent anything

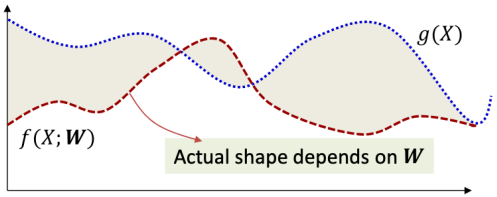




- The MLP *can be constructed* to represent anything
- But *how* do we construct it?

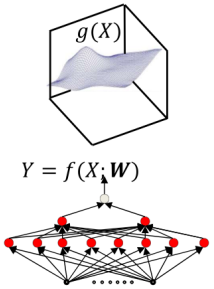
How to learn a network?

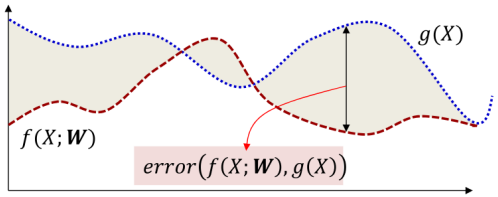




- Solution: Estimate parameters to minimize the error between the target function $g(X)$ and the network function $f(X, \boldsymbol{W})$
  - Find the parameter $\boldsymbol{W}$ that minimizes the shaded area

How to learn a network?





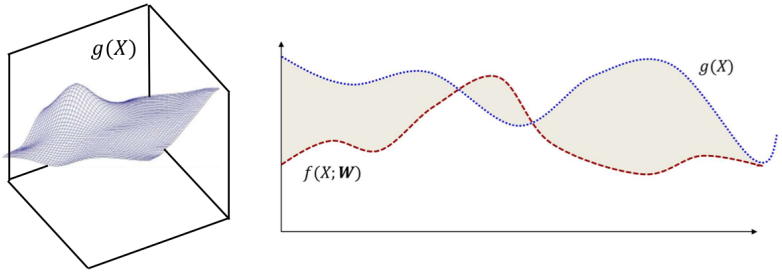
- The shaded area

$$totalerr(\boldsymbol{W}) = \int_{-\infty}^{\infty} error\big(f(X; \boldsymbol{W}), g(X)\big) dX$$

- The optimal $\boldsymbol{W}$

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

Problem: $g(X)$ is unknown

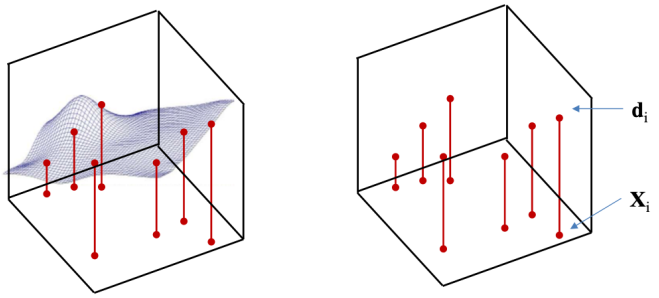


- Function $g(X)$ must be fully specified in order to compute

$$\int_{-\infty}^{\infty} error\big(f(X;\boldsymbol{W}), g(X)\big)dX$$

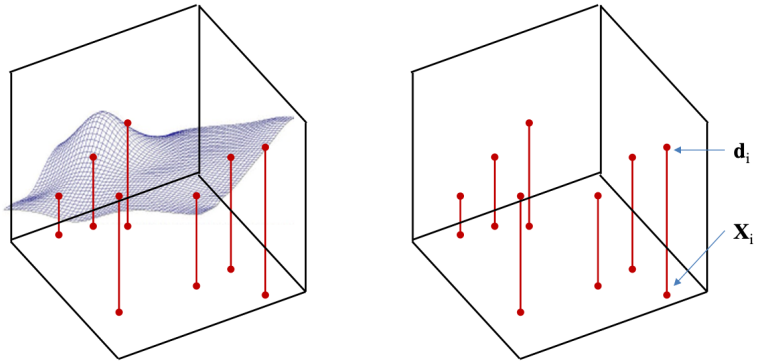
  - Known *everywhere,* i.e. for *every* input X
- **In practice we will not have such specification**

Sampling the function



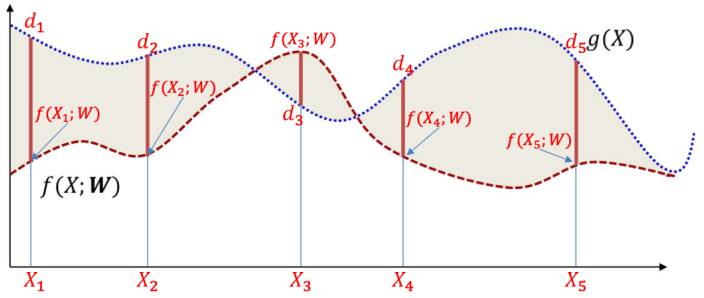
- *Sample* $g(X)$
  - Basically, get input-output pairs for a number of samples of input X_i
    - Many samples (X_i, d_i), where $d_i = g(X_i) + noise$
- Very easy to do in most problems: just gather training data
  - E.g. set of images and their class labels
  - E.g. speech recordings and their transcription

Drawing samples



- We must ***learn*** the *entire* function from these few examples
  - The "training" samples

The *Empirical* error



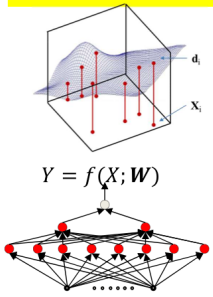
- The *empirical estimate* of the error is the *average* error over the training samples

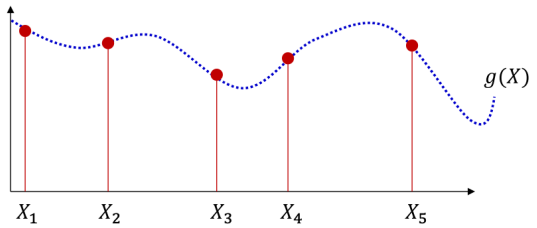
$$EmpiricalError(\boldsymbol{W}) = \frac{1}{N}\sum_{i=1}^{N} error(f(X_i;\boldsymbol{W}), d_i)$$

- Estimate network parameters to minimize this average error instead

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

Learning the function from training samples





- Aim: Find the network parameters that "fit" the training points exactly

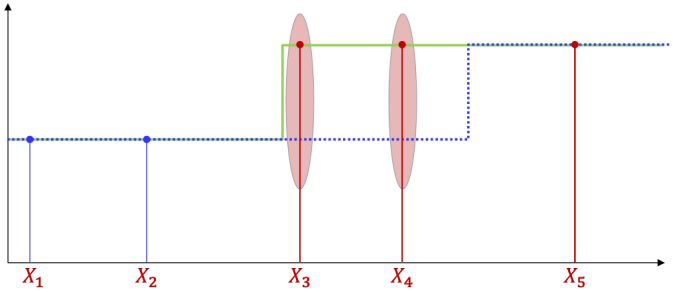
$$\boldsymbol{W}: EmpiricalError(\boldsymbol{W}) = 0$$

  - Assuming network architecture is sufficient for such a fit
  - Assuming unique output d at any $\mathbf{X}$
- And hopefully the resulting function is also correct where we *don't* have training samples

Let's begin with a simple task

- Learning a *classifier*
  - Simpler than regressions
- This was among the earliest problems addressed using MLPs
- Specifically, consider *binary* classification
  - Generalizes to multi-class

The *Empirical Classification* error

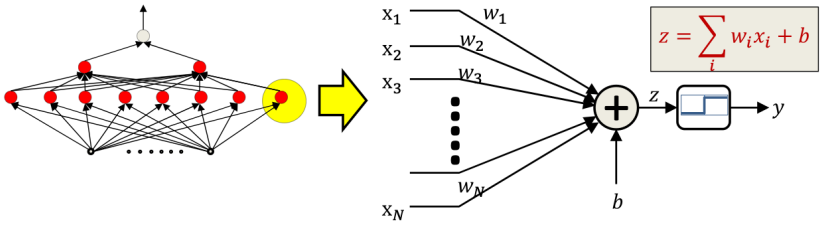


- The obvious error metric in a classifier is binary
  - The classifier is either right (error=0) or wrong (error=1)
    - Either $f(X; \boldsymbol{W}) = d$, or $f(X; \boldsymbol{W}) \neq d$

$$EmpiricalError(\boldsymbol{W}) = \frac{1}{N}\sum_{i=1}^{N} 1(f(X_i; \boldsymbol{W}) \neq d_i)$$

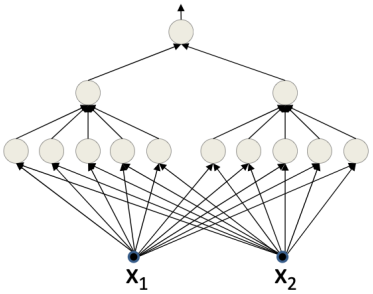
- **Learning the classifier:** Minimizing the count of misclassifications

History: The original MLP



- The original MLP as proposed by Rosenblatt: a network of threshold units
  - But how do you train it?
    - Given only "training" instances of input-output pairs

Learning a *multilayer* perceptron

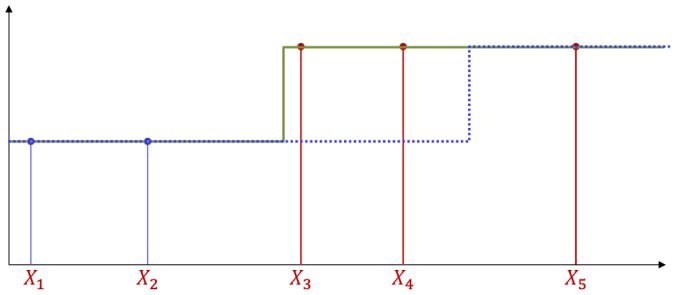


Training data only specifies input and output of network

Intermediate outputs (outputs of individual neurons) are not specified

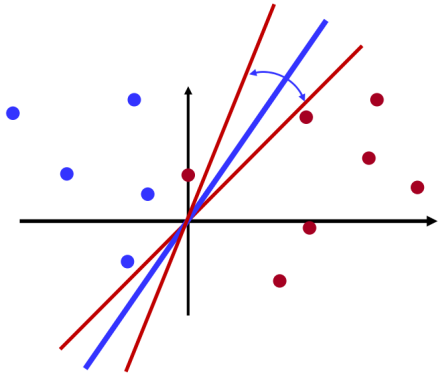
- Training this network using the perceptron rule is a combinatorial optimization problem
- We don't know the outputs of the individual intermediate neurons in the network for any training input
- **Must also determine the correct output for *each* neuron for *every* training instance**
- **At least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

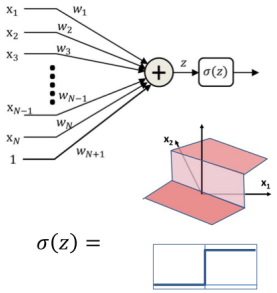
The problem



- Our binary error metric is not useful
  - To improve the classifier we must move the blue dotted line left
  - But if we move it only slightly, moving it either right or left results in no change in error

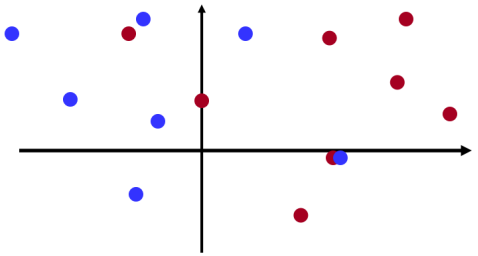
Why this problem?





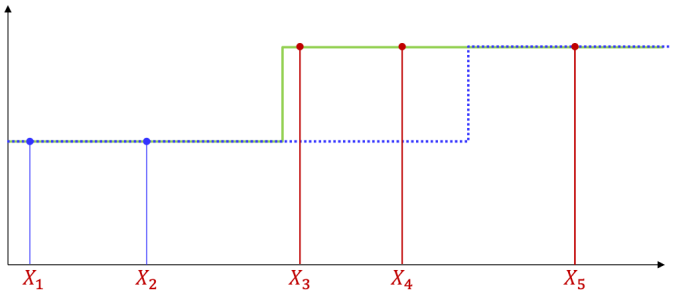
- The perceptron is a flat function with zero derivative everywhere, except at 0 where it is non-differentiable
  - You can vary the weights a *lot* without changing the error
  - There is no indication of which direction to change the weights to reduce error

A second problem: What we *actually* model



- Real-life data are rarely clean
  - Not linearly separable
  - Rosenblatt's perceptron learning rule wouldn't work in the first place

The solution

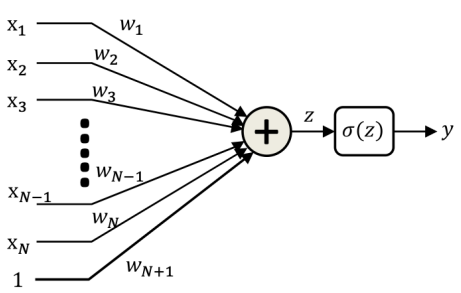


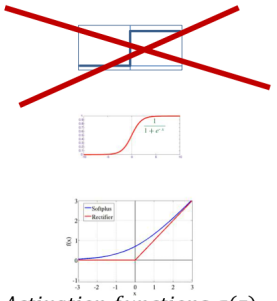
- Change our way of computing the mismatch such that modifying the classifier slightly lets us know if we are going the right way or not
  - This requires changing both, our activation functions, and the manner in which we evaluate the mismatch between the classifier output and the target output
  - Our mismatch function will now not actually count errors, but a *proxy* for it

99

1

Solution: Differentiable activation





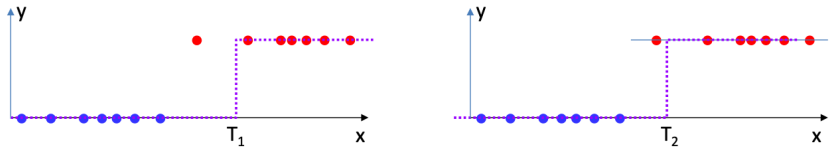
Activation functions $\sigma(z)$

- 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..

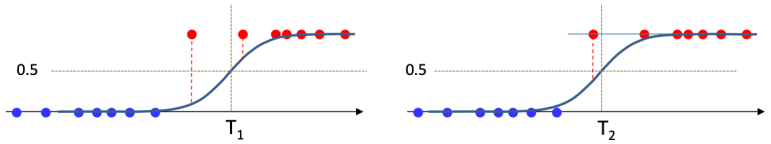
100

2

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

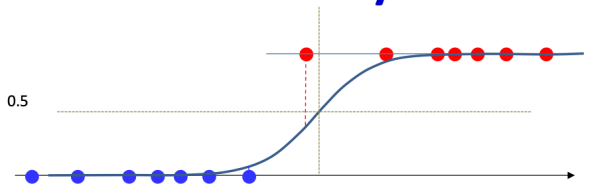


- Smooth, continuously varying activation: Classification based on whether the output is greater than 0.5 or less
  - Quantify *how much* the output differs from the desired target value (0 or 1)
  - Moving the function left or right changes this quantity, even if the classification error itself doesn't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

101

3

The two key requirements for learnability

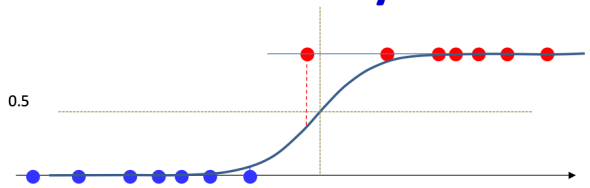


- Continuously varying activation
  - Differentiable
- Continuously varying error function
  - Also differentiable

104

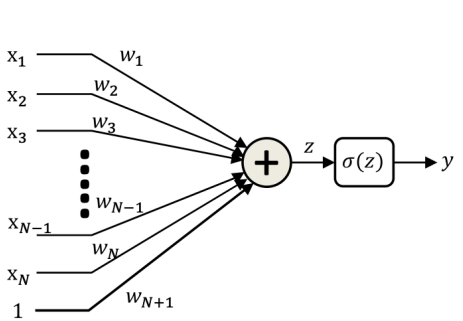
4

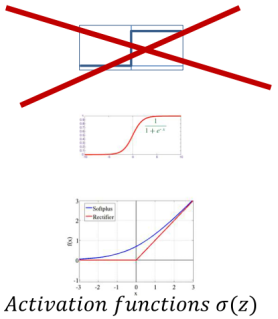
The two key requirements for learnability



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

Continuous Activations

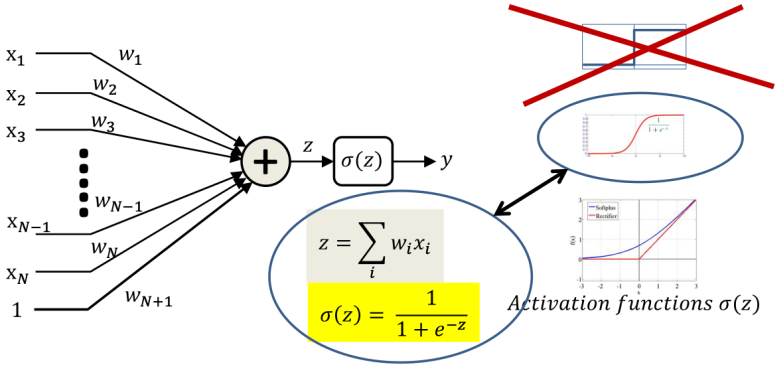




Activation functions $\sigma(z)$

- 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
    - More on these later

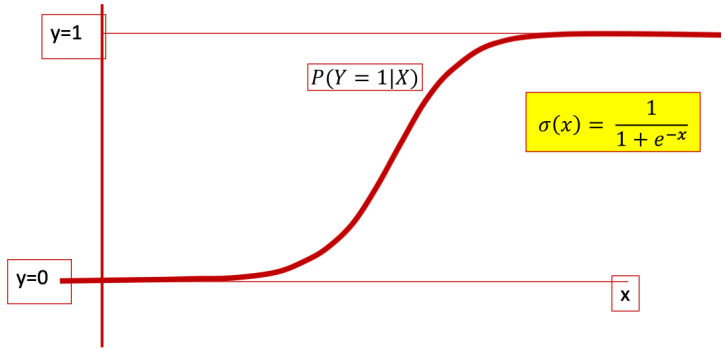
The sigmoid activation is special



Activation functions $\sigma(z)$

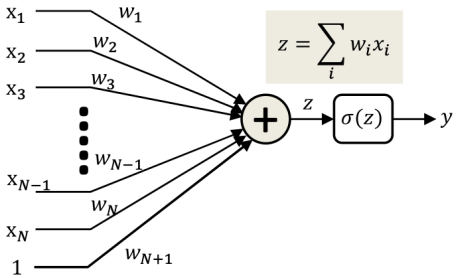
- This particular one has a nice interpretation
- It can be interpreted as $P(y = 1|x)$

The logistic regression model



- Class 1 becomes increasingly probable going left to right
  - Very typical in many problems

Perceptrons with differentiable activation functions



$$z = \sum_i w_i x_i$$

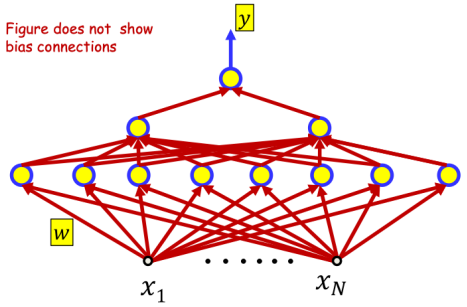
$$\frac{dy}{dz} = \sigma'(z)$$

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

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

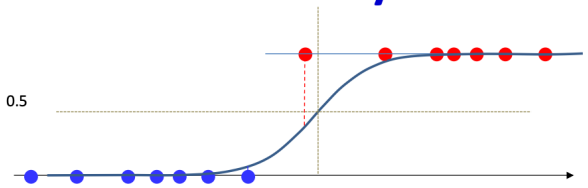
- $\sigma(z)$ **is a differentiable function of** z
  - $\frac{d\sigma(z)}{dz}$ is well-defined and finite for all z
- **Using the chain rule,** y **is a differentiable function of both inputs** x_i **and weights** w_i
- **This means that we can compute the change in the output for *small* changes in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

Overall function is differentiable



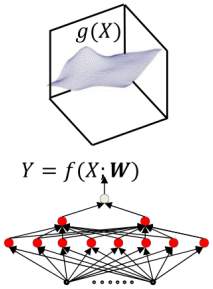
- By extension, the overall function is differentiable w.r.t every parameter in the network
  - We can compute how small changes in the parameters change the output
    - For non-threshold activations the derivative are finite and generally non-zero
- We will derive the actual derivatives using the chain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

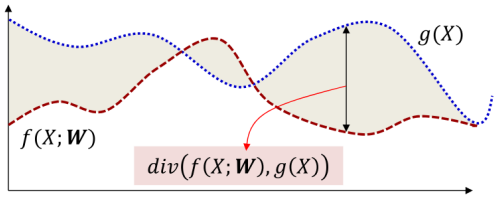
The two key requirements for learnability



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

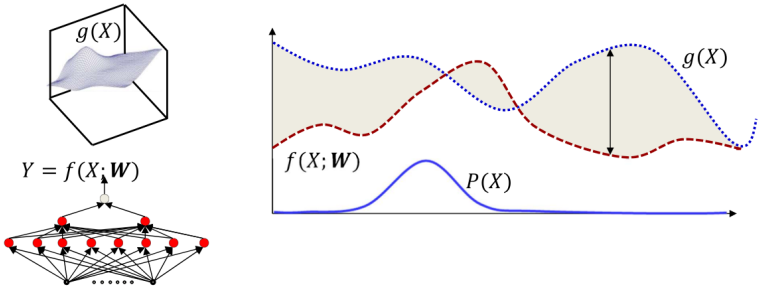
The error function





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

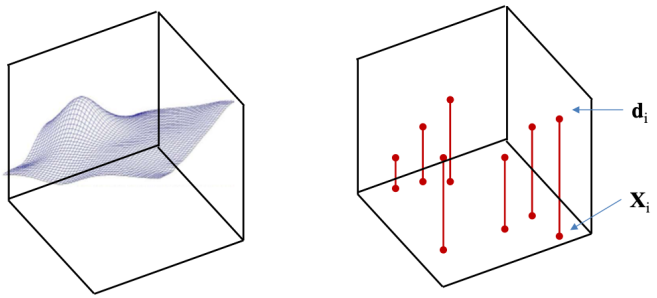
Minimizing *expected* divergence



- 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



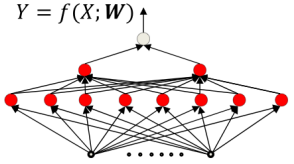
- The *expected* divergence (or risk) is the average divergence over the entire input space

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

- 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

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