Lecture 3 - Empirical Risk Minimization (Training the Network)

Contents

4 Empirical Risk Minimization

4.1 Recap

1. There is a minimum depth below which the network size must increase exponentially in order to model the target function
2. If you don't provide the neuron with as many neurons as required it can never model the function
3. With appropriate activation functions, having a deeper network is a lot more effective than having a shallow network

4.2 The Problem of Learning a NN

In a Feed Forward Network there are no loops in the network. The information always flows in 1 direction, It never comes back and loops back to a neuron. The neuron is never going to see that input again in any manner.

4.3 Parameters of the Network

The parameters of a network are the parameters of individual neurons themselves - weights and biases of all the neurons. The extra component which is always 1 represents the bias as a weight term. I can represent the complete set of parameters of a function as the set of weights of all of the neurons which are represented as W in the following expression:

$$Y = f(X; W) \tag{1}$$

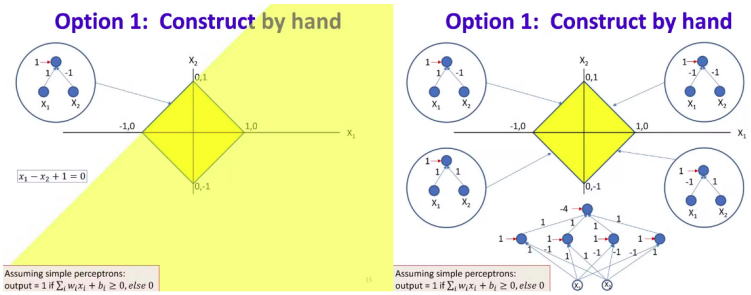
- The entire network is a function which operates on the input X and has parameters W.
- And based on the input X and parameters W, it is going to produce the output Y
- Anything that comes after ";" represents parameters, and before are arguments of the function
- Learning these parameters must be set to appropriate values to get the desired behavior from the network

4.4 How to Compose the Network that performs the desired function?

We know that MLP can represent any function, but how to construct it?

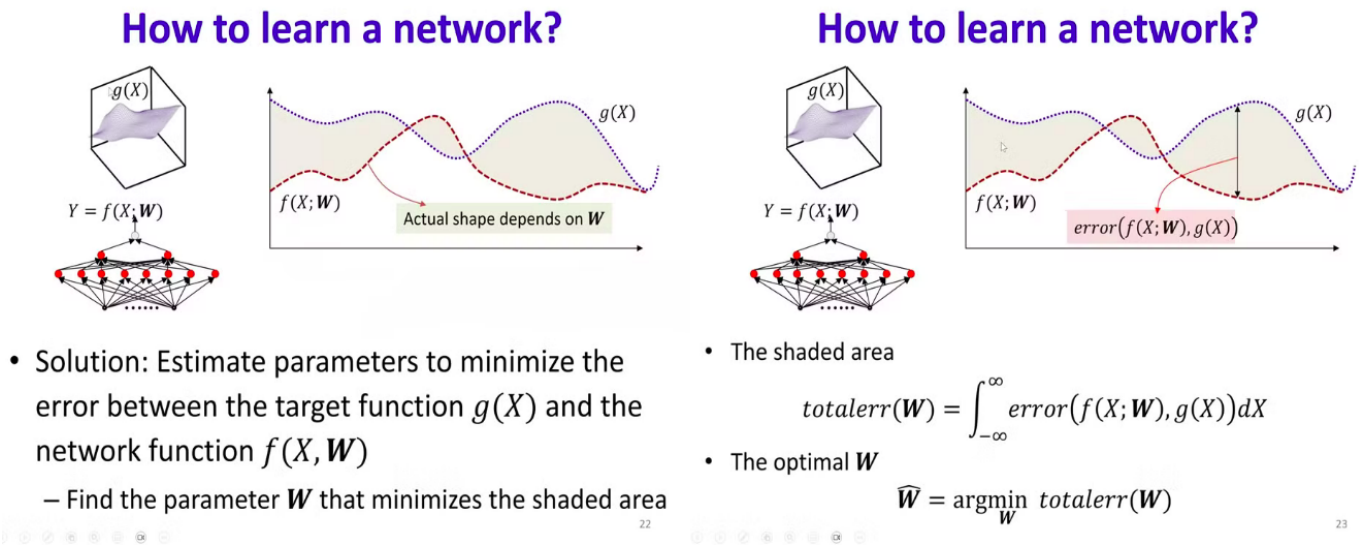
4.4.1 Option 1: Construct by Hand

Given a function, we can Hand craft a network to satisfy it. But this method is not practical

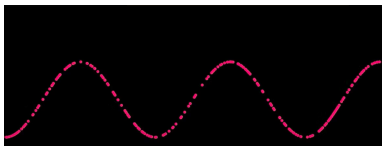


4.4.2 Option 2: Automatic Estimation of MLP

If we're given a function $g(X)$, and we want to build a network that can model it, we have to derive the parameters of the network W such that this network computes this function. For any given setting of W the network is going to compute some function - not necessarily the function we want. And so between the function that the network actually computes and the function that you want the network to compute, there is an error. The shaded area shown is the total error that the network makes over all possible inputs. It is the integral of the error function taken over the entire input space. We want to compute the parameters W such that the shaded area is minimized.



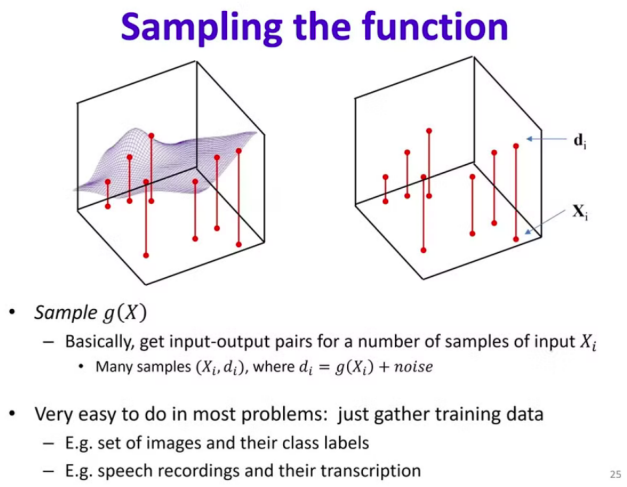
4.4.3 But $g(X)$ is unknown



More generally we dont know what the function is - which the network is supposed to model. If we know the function we dont need the network - we can just compute it. Instead of the function $g(X)$ we have samples (training data) - values of $g(X)$ for certain inputs. We must approximate a function that fits these data points and allow us to accurately predict outputs given inputs that are

not in our dataset - curve fitting. The neural network is also a function and must target a target function.

$$f(x) \approx NN(x) \tag{2}$$



From these dots we have to determine the network parameters that compute the complete function. We must learn the entire function just from these few samples (training examples)

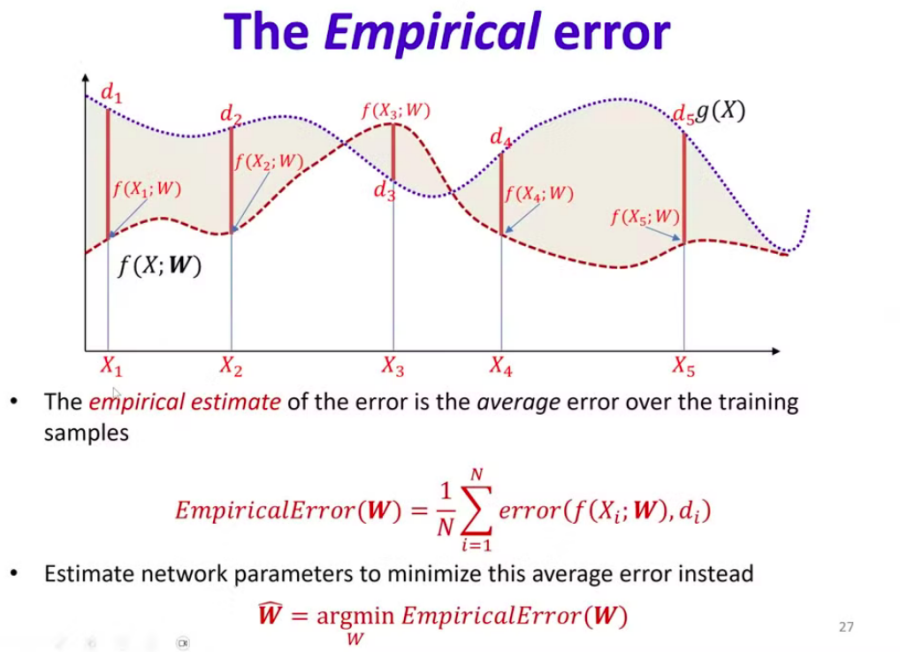
4.5 Empirical Error

Also we don't actually have the total error because we don't know $g(X)$. Instead we have a certain number of samples where we know the target value, and at these samples we also know the actual output of the network itself. The average of these errors over all of the training samples - This is the empirical error. `This is a proxy for the total error`. We want to find the network parameters that fit the training points exactly (The network parameters which minimize the empirical error). If you learn the network to predict the function exactly at these training points, It will somehow also figure out the correct function.

4.6 Empirical Classification Error

Below is a Binary Classification Example:

- Assuming target function we want to model is the green line - but we dont know it
- The current set of W gives us this blue dotted line - it is giving 2 errors
- Empirical error = what fraction of inputs it is making the error on
- d_i = desired function; N = number of samples
- Training is going to find the W which minimizes the empirical error
- Trying to minimize the count of misclassifications - minimize FP and FN



4.7 The Perceptron Rule for Learning Individual Perceptron

A single perceptron computes an affine function of its inputs, which is a `linear combination` of the input values and their corresponding weights. The result of this combination is passed through a threshold function:

- If the output is non-negative, the perceptron returns 1 (classifying one group).
- If the output is negative, it returns 0 (classifying the other group).
- The decision boundary is therefore a hyperplane that separates the two classes.

4.7.1 Affine vs Linear Functions

An affine function is like a linear function but does not necessarily pass through the origin. It's represented as $W^T X + b$, where b is a bias term. To convert an affine function to a linear function, you can add a bias by introducing an extra input with a value of 1. This moves the function into a higher dimension, allowing the decision boundary to shift.

4.7.2 Perceptron Learning Rule

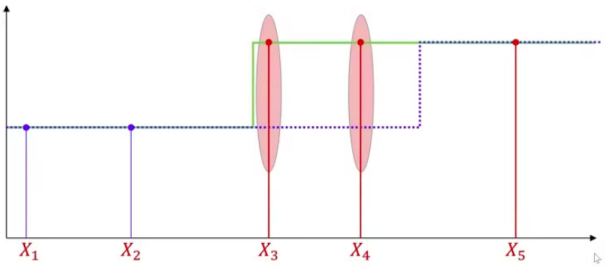
In training, the perceptron doesn't "know" the function in advance—it is only given input data and corresponding output labels (ground truth). The perceptron updates its weights using the perceptron learning rule: If the perceptron makes a wrong prediction, adjust the weights by moving in the direction that would have given the correct classification. If the prediction is correct, no adjustment is made.

Summary

- "Learning" a neural network == determining the parameters of the network (weights and biases) required for it to model a desired function
- The network must have sufficient capacity to model the function
- Ideally, we would like to optimize the network to represent the desired function everywhere
- However this requires knowledge of the function everywhere
- Instead, we draw "input-output" training instances from the function and estimate network parameters to "fit" the input-output relation at these instances

- Which of the following are true regarding how to compose a network to approximate a given function?
  - **The network architecture must have sufficient capacity to model the function**
  - **The network is actually a parametric function, whose parameters are its weights and biases**
  - **The parameters must be learned to best approximate the target function**
  - The parameters can be perfectly learned from just a few training samples of the target function, even if the actual target function is unknown.

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

33

4.7.3 Affine to Linear Function Transformation

- By introducing the bias term, we effectively work in a higher-dimensional space. The decision boundary can now pass through the origin in this higher space, transforming the problem from affine to linear.

- This allows us to define a hyperplane as the decision boundary, which separates the two classes.

The equation of the hyperplane is given by:

$$\sum_{i=1}^{N+1} w_i X_i = 0 \tag{3}$$

or in vector form:

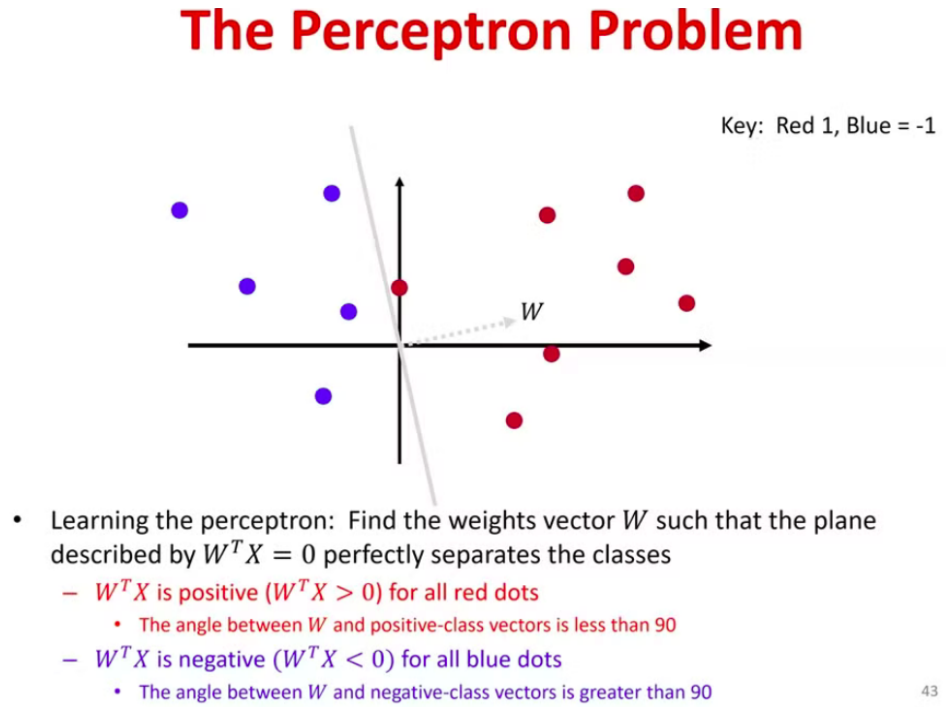
$$W^T X = 0 \tag{4}$$

where: - W is the weight vector, - X is the input vector.

This equation represents all the points XX that lie on the hyperplane, i.e., the decision boundary. The perceptron's task is to find the weight vector WW such that this boundary correctly separates the two classes.

4.7.4 Geometric Interpretation

The hyperplane is orthogonal (perpendicular) to the weight vector W. Points on one side of the hyperplane (classified as 1) will have a positive dot product with WW, while points on the other side (classified as 0) will have a negative dot product. The dot product $W^T X$ indicates whether a point is on the same side as the weight vector. If the angle between W and X is less than 90°, $W^T X$ is positive; otherwise, it's negative.



4.8 Finding W (Perceptron Learning Rule for Individual Perceptrons)

The goal of the perceptron learning algorithm is to find the optimal weight vector W such that the empirical error is minimized. This can be achieved using the **Online Perceptron Rule** developed by Rosenblatt. Here's how it works:

1. **Initialize** W: The weight vector W is initialized, often with small random values.
2. **Update** W: Each time a training instance is incorrectly classified, the weight vector is updated. The goal is to adjust W incrementally, ensuring that the decision boundary becomes optimal. The optimal decision boundary will place training points at the maximum possible distance, and will be **90 degrees** to either the input vector x or the weight vector W.

   Positive Instances

   For a correctly classified **positive** instance, the optimal weight vector W^* is equal to the input vector x, i.e.:

   $$W^* = x$$

   This means the weight vector W^* aligns with the input vector of the training point.

   Negative Instances

   For a **negative** instance, the optimal weight vector is the opposite of the input vector, i.e.:

   $$W^* = -x$$

   The weight vector points in the opposite direction of the input.

4.8.1 Misclassified Instances

When a training instance is **misclassified**, we adjust the weight vector W based on whether the instance is positive or negative:

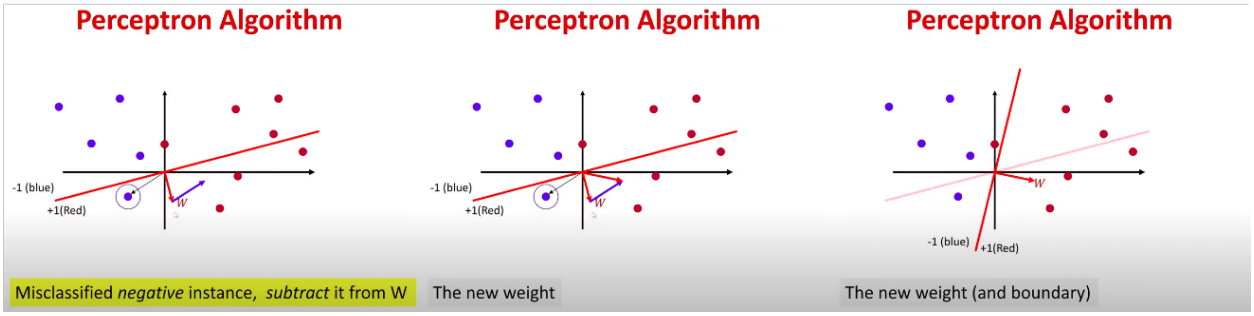
- For a **positive** misclassified instance, we add X_i to W.
- For a **negative** misclassified instance, we subtract X_i from W.

4.8.2 Iterating Over Training Instances

1. **Initialize** W randomly.
2. The algorithm starts by classifying all instances in the dataset. Initially, instances below the decision boundary are classified as positive and those above as negative.
3. The algorithm cycles through each instance, checking if it is correctly classified.
4. If a misclassification is found:
   - For a **positive** instance, add $+X$ to the current weight vector.

- For a **negative** instance, add $-X$ to the current weight vector.

After adjusting the weight vector, a new decision boundary is created. The process then repeats by checking all training instances again with this updated boundary.



4.8.3 Convergence of the Perceptron Algorithm

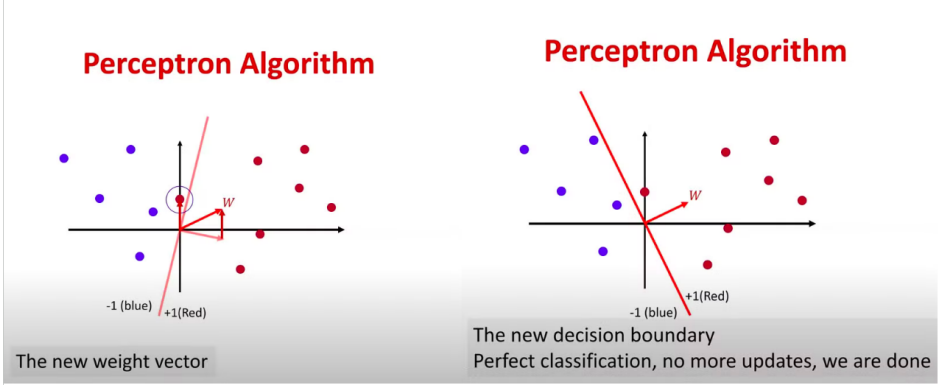
The perceptron learning rule is based on the principle that the optimal weight vector for any given instance either points directly **toward** it (positive instance) or directly **away** from it (negative instance).

- If the two classes are linearly separable by a hyperplane, the algorithm is guaranteed to converge in a finite number of steps.
- However, if the classes are not linearly separable, the algorithm will never converge to a solution, as no perfect hyperplane exists that can separate the two classes.

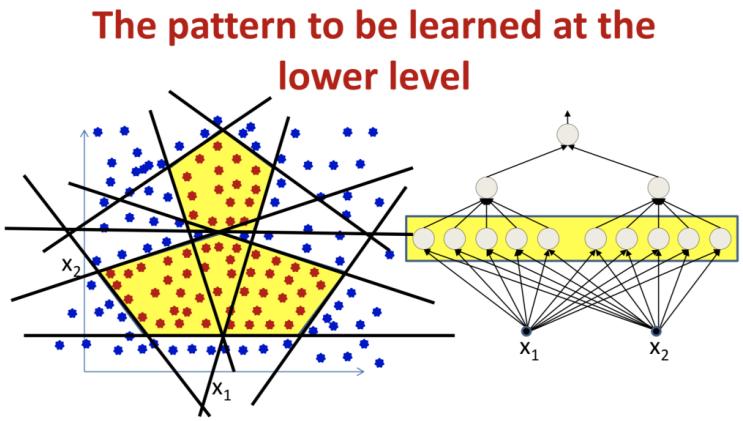
This iterative approach continues until all training instances are correctly classified or until the algorithm fails to converge due to non-linearly separable data.

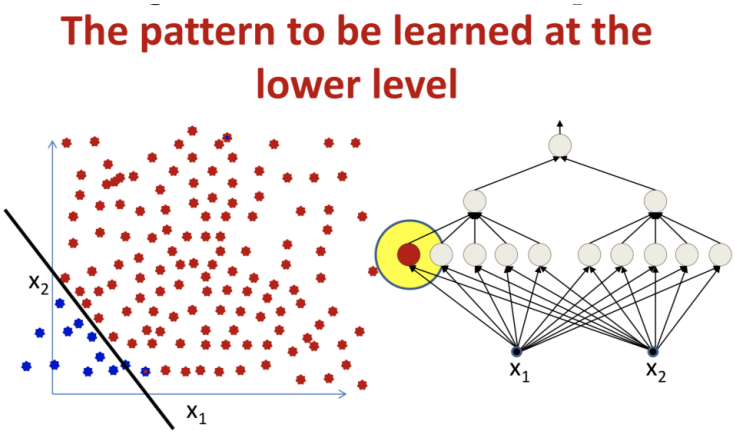
4.9 Perceptron Learning Rule for Complex Tasks

Each decision line is a two-class linear classifier of a single perceptron, but needs to change the label of some samples. Positive class: red dot, negative class: blue dot (2 variables, 10 decision lines, 2 pentagons for summation). Each decision line is a two-class linear classifier of a single perceptron,



but needs to change the label of some samples. So we Turn the blue sample on the right side of the decision line into red and start training. How do I change the label? This also requires training! In theory, for each decision line, you need to try each method of changing the blue point to red, get different decision faces, and finally choose a correct one. Therefore, in the process of training, it is also necessary to train at the same time, how to modify the rules of the label color for each decision line. Computationally, this is an exponential search operation, NP problem, it is impossible to obtain the optimal solution on the calculation. But there are 2 greedy algorithms that can try to get suboptimal solutions, namely Adaline and Madaline. Each neuron must be trained to output all samples, including the output of the modified sample.





Summary

So it is difficult to train the network according to the perceptron rules above

To create a correct decision boundary for non-linearly separable data, we might need to relabel the data in various ways. If there are N training instances, there are 2^N possible ways of relabeling them. For each relabeling, a boundary is learned, and we aim to find the one that minimizes classification error.

In practice, the perceptrons in the network aren't already trained, so the relabeling process must be repeated for each neuron. This creates an exponential complexity problem, both in terms of the number of neurons and the number of inputs, making it impractical. ADALINE (Adaptive Linear

Neuron) and MADALINE (Multiple ADALINEs) were proposed as greedy algorithms that address some of these challenges by learning in a stepwise, simpler manner.

4.10 Solution - Differentiable Activation Functions

4.10.1 Why is it difficult to train a network with perceptron rules and the threshold function?

Answer

- The perceptron rule combined with the threshold function presents a problem because the function does not provide continuous feedback on weight changes. It only updates when the input makes a large enough jump to cross the threshold, preventing the network from making small, incremental improvements.
- This is why differentiable activation functions (like sigmoid or ReLU) are favored in modern neural networks—they provide smooth feedback and allow for more efficient training through gradient-based methods.

- **Lack of Gradient Information**
  - The threshold function is essentially a step function. It outputs a binary value, typically 0 or 1, based on whether the input crosses a certain threshold.
  - Because it has no slope (its derivative is either zero or undefined), a small change in the input (e.g., weights or inputs to the neuron) does not produce any meaningful change in the output. Therefore, the model doesn't provide any feedback on how to adjust the weights for better accuracy, unless the change is significant enough to cross the threshold.
- **Delayed Feedback**
  - In networks that rely on the threshold function, a small adjustment to the weight w doesn't immediately influence the output y. This is because the step function either "steps" from 0 to 1 (or vice versa), but only when a large enough change happens.
  - As a result, training becomes difficult because we don't receive incremental feedback as the weight changes. For example, if the weight w changes slightly, there might be no immediate effect on the output y. The network would only recognize an effect when the change is large enough to pass the threshold, making it hard to fine-tune the model.
- **Optimization Challenge**
  - In neural networks, weight updates typically rely on gradient information—i.e., how sensitive the output is to changes in the weights. The step function lacks a gradient because it is either "on" or "off."
  - Without this gradient, gradient-based optimization methods like gradient descent cannot be applied effectively. This makes it hard for the network to know how to optimize the weights in the direction that improves classification performance.

4.10.2 Differentiable Activation Functions

Traditional perceptrons use a threshold activation function, which is non-differentiable and has a zero derivative almost everywhere. In order to use gradient-based optimization techniques, such as gradient descent, the activation function must be differentiable. This allows us to estimate parameters effectively. Graded activation functions (e.g., sigmoid, tanh, ReLU) help compute how small changes in input z affect the output y.

Using the chain rule, we can propagate these changes through the network to adjust weights in the direction that minimizes error.

For example:

$$z = \sum_i w_i x_i$$

$$\frac{dy}{dz} \text{ tells how much a small change in } z \text{ changes } y$$

$$\frac{dz}{dw} \text{ tells how small changes in } w \text{ affect } z$$

Chaining these derivatives helps compute how changes in weights w affect the output y, which allows us to adjust weights during learning.

4.11 Learning through Empirical Risk Minimization

In modern neural networks, the error function is also made differentiable, and instead of counting misclassifications, a divergence function is defined that can be minimized. This divergence is integrated to obtain the total error, focusing on regions of the input space that are likely to be seen during inference.

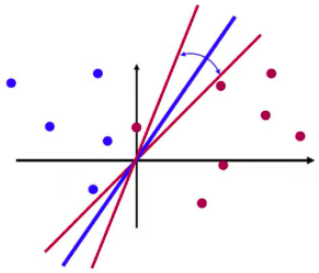
The integral of this divergence, weighted by the probability of seeing each input x, is the expected value of the divergence, which gives us the empirical risk estimate.

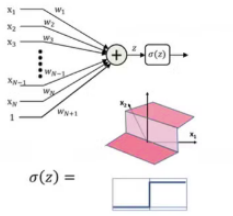
In essence:

- Empirical risk minimization seeks to minimize the expected divergence across all training samples, creating a proxy for the true risk.

This approach is critical for training modern neural networks using gradient-based optimization techniques and makes learning from complex, non-linear boundaries feasible.

Why this problem?

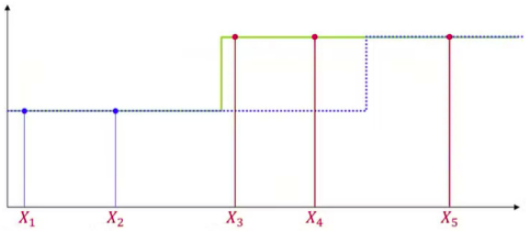




- The perceptron is a flat function with zero derivative everywhere, except at 0 where it is non-differentiable
  - You can vary the weights a *lot* without changing the error
  - There is no indication of which direction to change the weights to reduce error

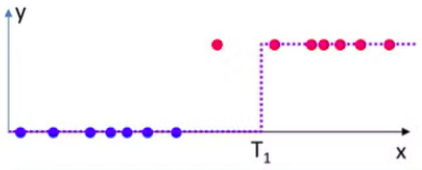
97

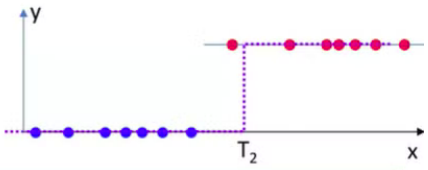
The solution



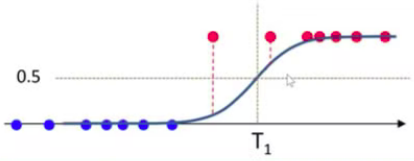
- Change our way of computing the mismatch such that modifying the classifier slightly lets us know if we are going the right way or not
  - This requires changing both, our activation functions, and the manner in which we evaluate the mismatch between the classifier output and the target output
  -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

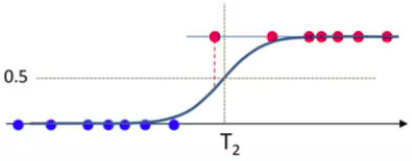
Differentiable Mismatch function





- 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

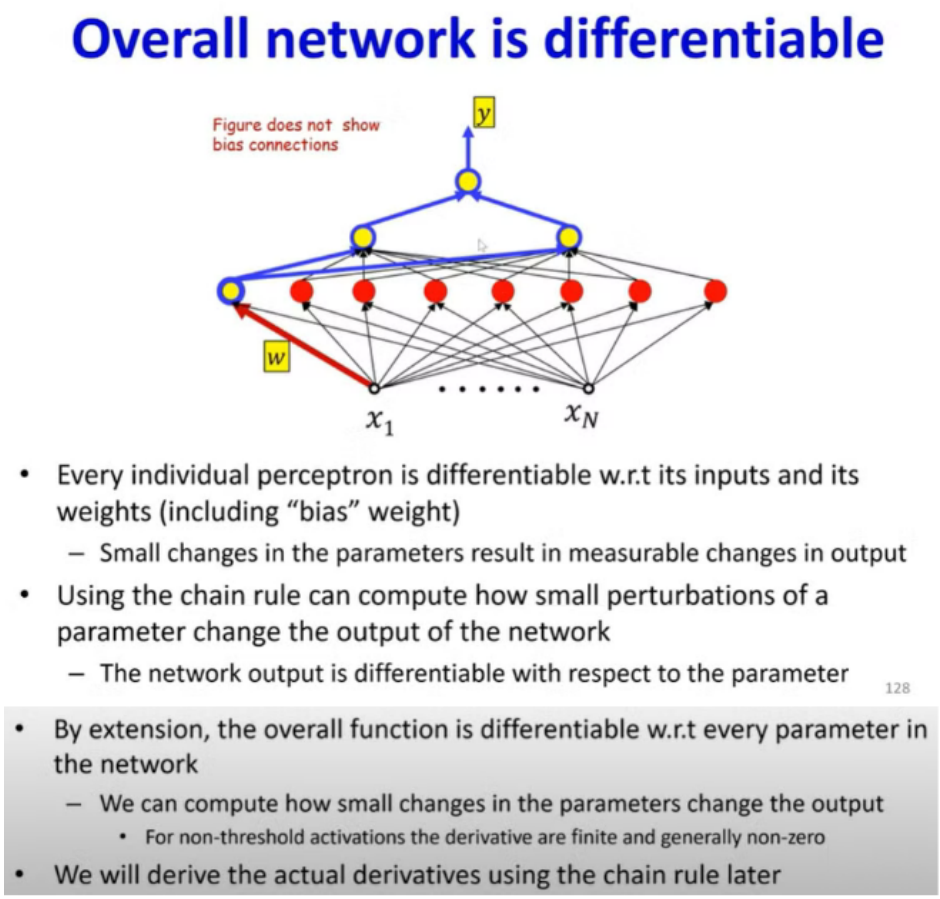
102

- Which of the following are true of the threshold activation
  - **Increasing (or decreasing) the threshold will not change the overall classification error unless the threshold moves past a misclassified training sample**
  - **We cannot know if a change (increase of decrease) of the threshold moves it in the correct direction that will result in a net decrease in classification error**
  - The derivative of the classification error with respect to the threshold gives us an indication of whether to increase or decrease the threshold

- Which of the following are true of the continuous activation (sigmoid)
  - **Shifting the function left or right will not change the overall classification error unless the crossover point (where the function crosses 0.5) moves past a misclassified training sample**
  - **Shifting the function will change the total distance of the value of the function from its target value at the training instances**
  - **The derivative of the total distance with respect to the shift of the function gives us an indication of which direction to shift the function to improve classification error**

2 Key Requirements for Learnability

1. Continuously Varying Function (Differentiable)
2. Continuously Varying Error Function (Differentiable)



4.12 Overall Network Differentiability

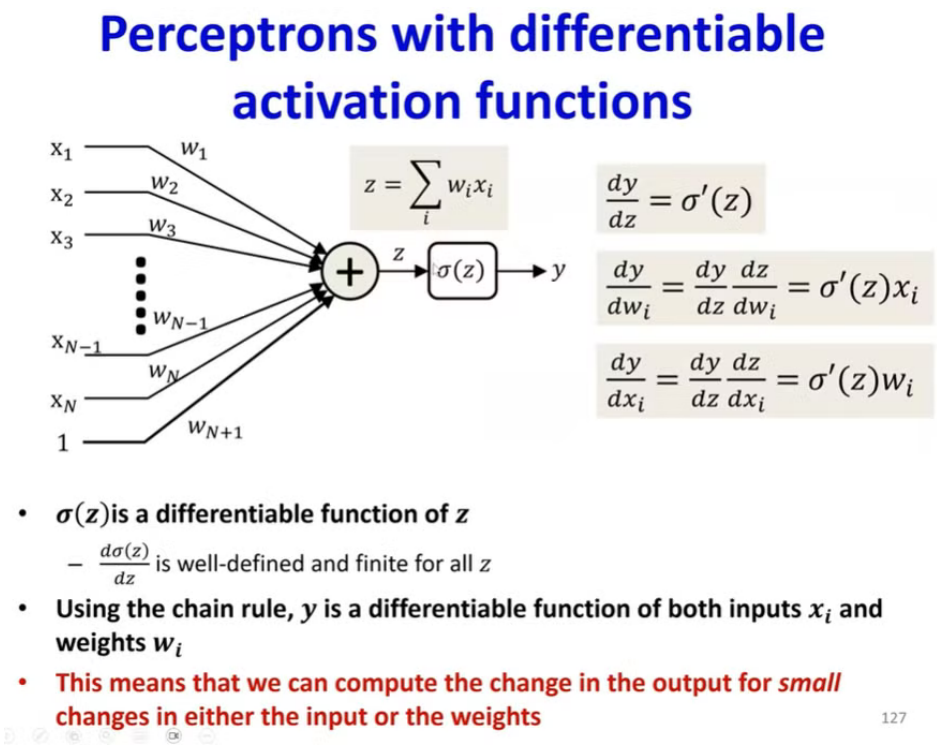
Key Points:

- **Perceptron Differentiability**: Every perceptron in a neural network is differentiable with respect to:
  - **Inputs**: The changes in inputs lead to measurable changes in the output.
  - **Weights**: The small changes in weights, including the bias weight, result in output changes.

Chain Rule Application:

- **Chain Rule**: This rule helps calculate how small changes (perturbations) in a parameter (weights, bias) affect the network's output.

- **Differentiability of the Network**: The overall network becomes differentiable with respect to each parameter due to the differentiability of individual perceptrons.



Conclusion:

- The entire function of the network is differentiable concerning every parameter, allowing us to compute derivatives, which will be derived later using the chain rule.

4.13 The Error Function

Assume that the objective function = g(X), the function of the multi-layer perceptron network training = $f(X;W)$. Therefore, the ultimate goal of training is to minimize the difference between g(X) and $f(X;W)$. In the two-dimensional plane, the training network parameter W , the area enclosed by the function curves of $g(X)$ and $f(X;W)$ is minimized.
Key Concepts:

- **Target and Output Functions**:

$$g(X) \quad \text{(Target function)} \quad \text{and} \quad f(X;W) \quad \text{(Output function of the network)}$$

Divergence Function:

- **Definition**: A divergence function $div(f(X;W), g(X))$ is used to quantify the difference between the network output and the target function.
  $div()$ represents the divergence (gap) between functions; the area enclosed by product dispersion.

Properties:

- If $f(X;W) = g(X)$, the divergence is zero.
- If $f(X;W) \neq g(X)$, the divergence is greater than zero.
- The divergence is differentiable with respect to f.

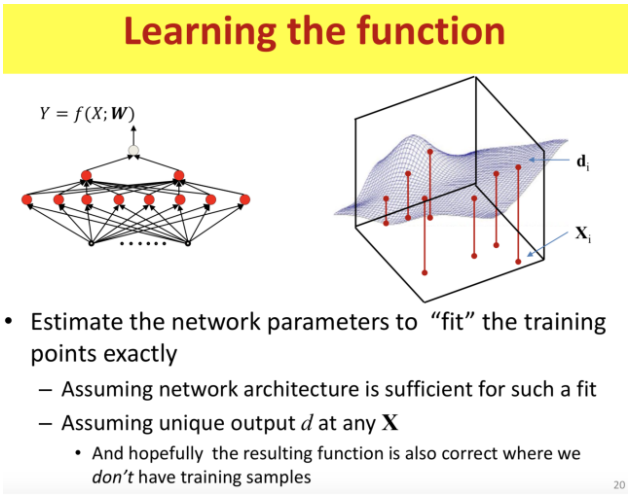
Minimization of Divergence:

- **Minimization Goal**: When $f(X;W)$ can exactly represent $g(X)$, we aim to minimize the divergence:
$$\hat{W} = \arg\min_W \int_X div(f(X;W), g(X))dX$$
- This means finding the weights W that minimize the divergence across the entire input space.

4.14 Expected Divergence (Risk) and Empirical Risk

When the range of the variable x is limited, it is meaningful to discuss the minimum expected error. When x occurs more frequently in certain intervals, more attention needs to be given to the region. Therefore, more accurately, the training network minimizes the expected error weighted for the probability of occurrence of x.
Expected Divergence:

- **Definition**: The expected divergence or **risk** quantifies the average divergence between the output and target functions over the entire input space:
$$E[div(f(X;W), g(X))] = \int_X div(f(X;W), g(X))P(X)dX$$
where $P(X)$ is the probability distribution of the inputs.
The above equation represents "the expected divergence (error) over the input space."



Empirical Estimate:

- Since we often do not have access to the true distribution $P(X)$, we estimate the expected risk empirically (average divergance) over the training samples

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

  where d_i is the target output for the i-th sample.

4.15 Training the Network: Empirical Risk Minimization

Loss Function:

- **Empirical Risk**: The empirical risk is the average of the divergence over all training data, referred to as the **Loss** function:

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

Minimization:

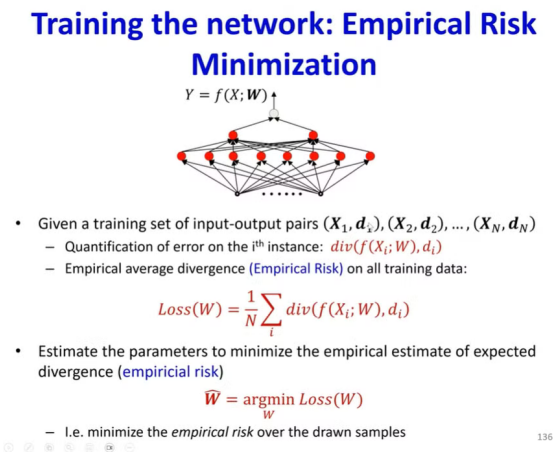
- The goal of training is to find the parameters W that minimize the empirical risk:

$$\hat{W} = \arg\min_{W} Loss(W)$$

- This involves adjusting W to minimize the empirical estimate of the expected divergence.

Notes:

1. **Loss Terminology**: While it is a measure of error, it is referred to as a **Loss** in standard terminology.
2. **True vs Empirical Risk**: The **Empirical Risk** is an approximation of the true risk $E[div(f(X;W), g(X))]$.
3. **Loss as a Function of** W: For a given training set, the loss depends solely on the parameters W.



4.16 Empirical Risk Minimization Summary

Summary

Learning networks of threshold-activation perceptrons requires solving a hard combinatorial-optimization problem

- Because we cannot compute the influence of small changes to the parameters on the overall error

Instead we use continuous activation functions with non-zero derivatives to enables us to estimate network parameters

- This makes the output of the network differentiable w.r.t every parameter in the network
- The logistic activation perceptron actually computes the a posteriori probability of the output given the input

We define differentiable divergence between the output of the network and the desired output for the training instances

- And a total error, which is the average divergence over all training instances

We optimize network parameters to minimize this error **Empirical risk minimization**. This is an instance of function minimization

xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx