Lecture 4b - Loss Functions

Contents

6 Loss Functions

6.1 Recap

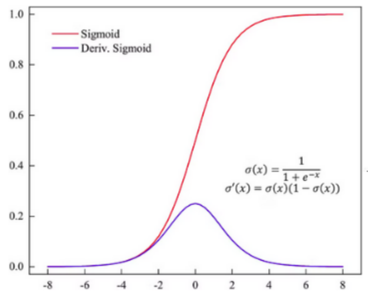
The goal of training a Neural Network is to find the parameters W that minimize the empirical risk:

$$\hat{W} = \arg\min_{W} Loss(W) \tag{1}$$

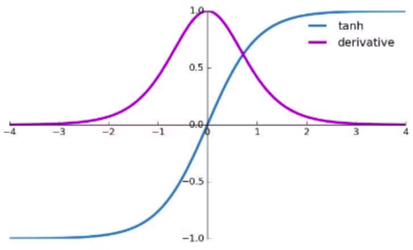
Where $Loss(W)$ can be represented using the below equation:

$$Loss(W) = \frac{1}{N}\sum_{i=1}^{N} div(f(X_i; W), d_i) \tag{2}$$

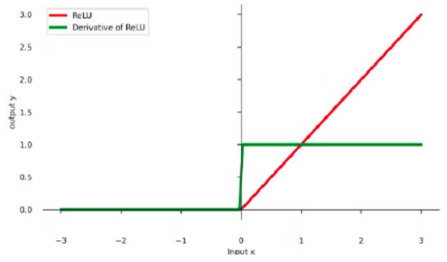
We have also established that the mechanism we'll use to achieve this is the negative gradient i.e. we will initialize the weights, at each point we're going to compute the gradient of the loss function wrt to the weights and take a step against it.
Also, we know that the perceptron first computes an affine function of its inputs and then puts it through an activation function. We prefer continuous graded activation functions which tell how far we were from out boundary otherwise it would gate information.



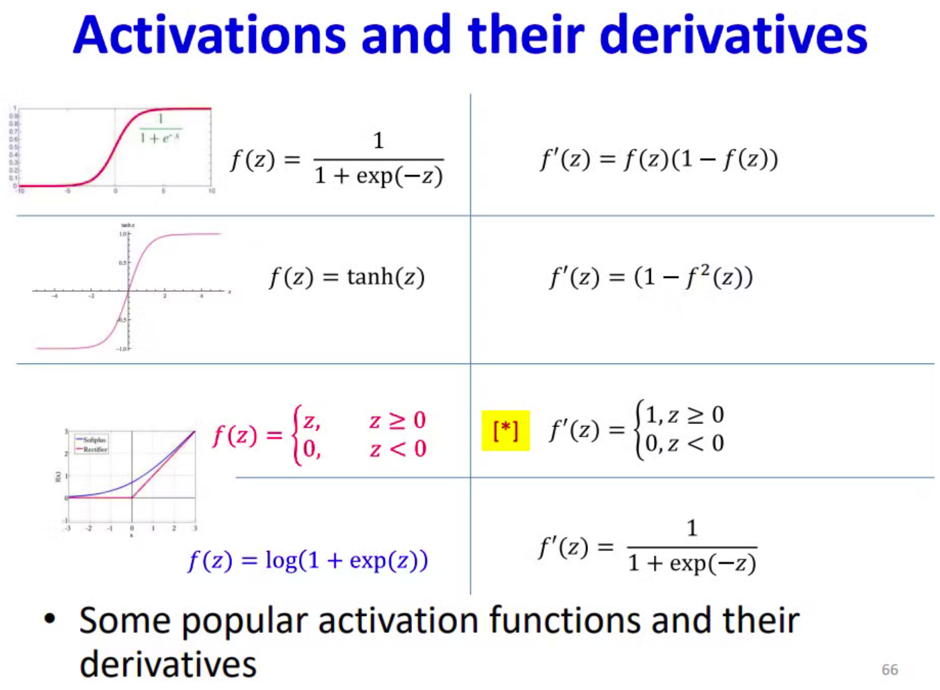
Sigmoid



tanh



ReLU



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

66

6.2 Scalar vs Vector Activations

6.2.1 Scalar Activations

Description: In scalar activations, each neuron receives input, processes it through an affine transformation followed by an activation function, producing an independent output.

Properties:

- Individual operation: Each neuron operates independently.
- Local influence: Changing one activation does not affect others.

Equations: For a neuron j in layer l,

$$z_j^{(l)} = \sum_i w_{ij}^{(l)} x_i + b_j^{(l)}$$

$$y_j^{(l)} = f(z_j^{(l)})$$

6.2.2 Vector Activations

Description: In vector activations, the output of neurons are interdependent; modifying one affects others in the layer.

Common Type: Softmax

- Converts outputs z into a probability distribution.
- All outputs are non-negative and sum to 1.

Equations:

$$Z_i = \sum_j W_{ij} x_j + b_i$$

$$Y = \text{Softmax}(Z) = \frac{e^{Z_i}}{\sum_k e^{Z_k}}$$

Implications:

- Interaction: Changes in weights affect all outputs due to the normalization step.
- Derivative relation: The derivative of an output with respect to another is non-zero, indicating dependency.

6.3 Input, Target Output, and Actual Output: Vector Notation

Description: Neural networks operate on vector inputs and outputs, especially in layers with multiple neurons.

Notation:

$$X_n = [x_{n1}, x_{n2}, \ldots, x_{nI}]^T$$
$$d_n = [d_{n1}, d_{n2}, \ldots, d_{nD}]^T$$
$$Y_n = [y_{n1}, y_{n2}, \ldots, y_{nL}]^T$$

Where:

- X_n is the input vector for the nth instance.
- d_n is the desired output vector for the nth instance.
- Y_n is the actual output vector from the network.

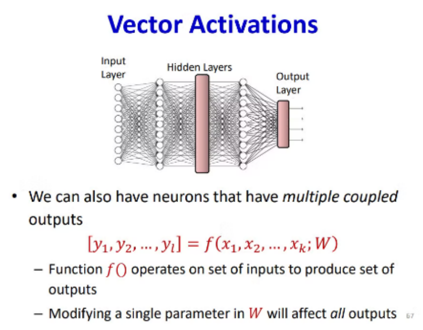
Usage:

- Each input instance X_i corresponds to a target output d_i.
- The network learns to approximate d_i from X_i through training.

The output activation is usually a sigmoid function. It is viewed as:

$$\text{Probability } P(Y = 1|X) \text{ of class value 1} \tag{3}$$

The above equation means, for actual data, in general a feature value X may occur for both classes, but with different probabilities. The output from the sigmoid is the probability of the target class.



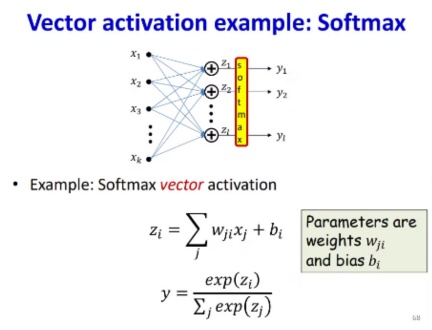
Vector Activations

Input Layer
Hidden Layers
Output Layer

- We can also have neurons that have *multiple coupled* outputs

$$[y_1, y_2, \dots, y_l] = f(x_1, x_2, \dots, x_k; W)$$

  - Function $f()$ operates on set of inputs to produce set of outputs
  - Modifying a single parameter in W will affect *all* outputs



Vector activation example: Softmax

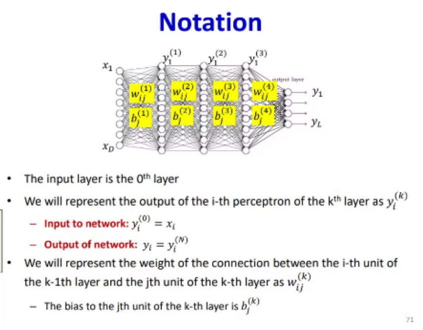
x_1
x_2
x_3
x_k
z_1
z_2
z_l
softmax
y_1
y_2
y_l

- Example: Softmax *vector* activation

$$z_i = \sum_j w_{ji} x_j + b_i$$

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

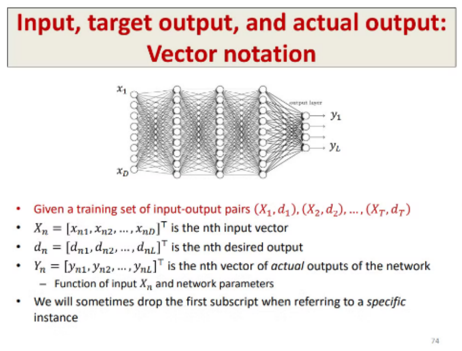
Parameters are weights w_{ji} and bias b_i



Notation

x_1
x_D
$y_1^{(1)}$
$y_1^{(2)}$
$y_1^{(3)}$
$w_{ij}^{(1)}$
$w_{ij}^{(2)}$
$w_{ij}^{(3)}$
$w_{ij}^{(4)}$
$b_j^{(1)}$
$b_j^{(2)}$
$b_j^{(3)}$
$b_j^{(4)}$
output layer
y_1
y_L

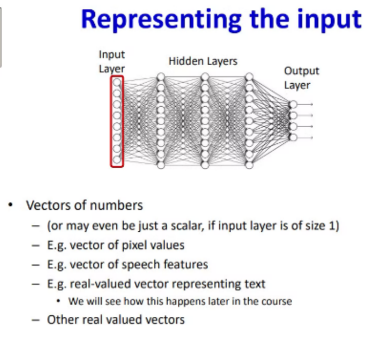
- The input layer is the 0th layer
- We will represent the output of the i-th perceptron of the kth layer as $y_i^{(k)}$
  - **Input to network:** $y_i^{(0)} = x_i$
  - **Output of network:** $y_i = y_i^{(N)}$
- We will represent the weight of the connection between the i-th unit of the k-1th layer and the jth unit of the k-th layer as $w_{ij}^{(k)}$
  - The bias to the jth unit of the k-th layer is $b_j^{(k)}$



Input, target output, and actual output: Vector notation

x_1
x_D
output layer
y_1
y_L

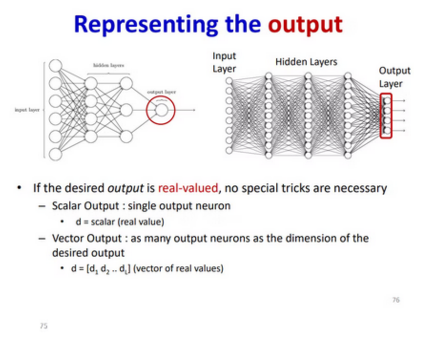
- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \dots, (X_T, d_T)$
- $X_n = [x_{n1}, x_{n2}, \dots, x_{nD}]^\top$ is the nth input vector
- $d_n = [d_{n1}, d_{n2}, \dots, d_{nL}]^\top$ is the nth desired output
- $Y_n = [y_{n1}, y_{n2}, \dots, y_{nL}]^\top$ is the nth vector of *actual* outputs of the network
  - Function of input X_n and network parameters
- We will sometimes drop the first subscript when referring to a *specific* instance



Representing the input

Input Layer
Hidden Layers
Output Layer

- Vectors of numbers
  - (or may even be just a scalar, if input layer is of size 1)
  - E.g. vector of pixel values
  - E.g. vector of speech features
  - E.g. real-valued vector representing text
    - We will see how this happens later in the course
  - Other real valued vectors



Representing the output

input layer
hidden layers
output layer
Input Layer
Hidden Layers
Output Layer

- If the desired *output* is real-valued, no special tricks are necessary
  - Scalar Output : single output neuron
    - d = scalar (real value)
  - Vector Output : as many output neurons as the dimension of the desired output
    - d = [d_1 d_2 .. d_L] (vector of real values)

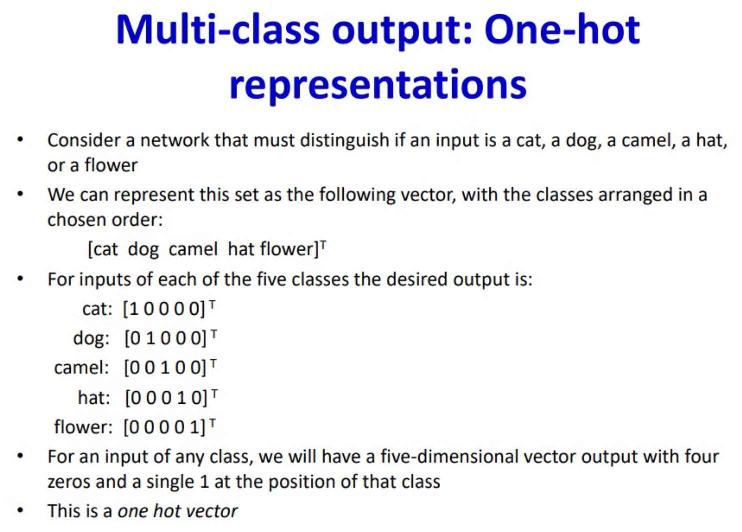
6.3.1 Multi-class Classification

Overview: In scenarios requiring classification among multiple classes, both input X and output d are vectors.

Training Set: Composed of pairs (X_i, d_i) where X_i are input features and d_i are labeled outputs.

Objective: To train the network such that it can predict the output vector d corresponding to the input vector X.

6.4 Multi-class Outputs



Multi-class output: One-hot representations

- Consider a network that must distinguish if an input is a cat, a dog, a camel, a hat, or a flower
- We can represent this set as the following vector, with the classes arranged in a chosen order:
  [cat dog camel hat flower]^T
- For inputs of each of the five classes the desired output is:
  cat: [1 0 0 0 0]^T
  dog: [0 1 0 0 0]^T
  camel: [0 0 1 0 0]^T
  hat: [0 0 0 1 0]^T
  flower: [0 0 0 0 1]^T
- For an input of any class, we will have a five-dimensional vector output with four zeros and a single 1 at the position of that class
- This is a *one hot vector*

6.4.1 One-Hot Encoding

- All classes are arranged in an order, and to represent a single class, a vector is created with as many components as the number of classes. - This vector has a value of 1 corresponding to the class being represented and 0 elsewhere. This is known as a **one-hot vector**.

$$\text{For example:} \quad \text{Cat} = [1,0,0], \quad \text{Dog} = [0,1,0], \quad \text{Bird} = [0,0,1]$$

- The desired output d is a multi-component vector with as many components as the number of classes.

6.5 Probability Output and Softmax Activation

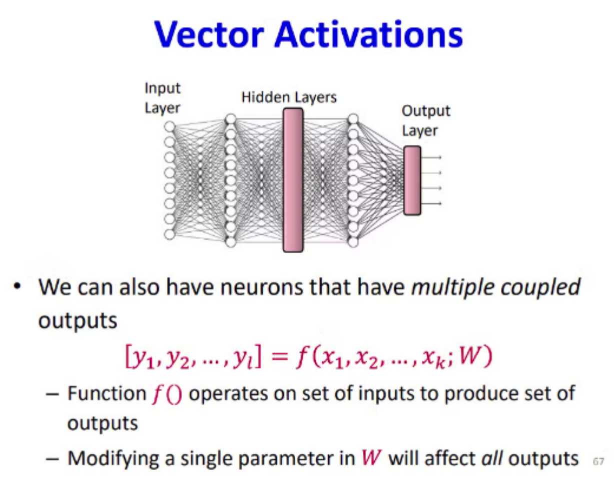
- Ideally, when given a picture of a cat, we want the network to output a 1 for the correct class (cat) and 0 for the others. - In practice, the output is a **probability vector**, where each component represents the predicted probability of each class.

- **Softmax Activation**: - For multi-class classification, we use a softmax activation at the output. - The softmax function exponentiates the affine transformation output (which ensures all values are positive) and normalizes it (ensuring they sum to 1).

$$Y_j = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

Where:

- z_j is the affine transformation (input to the softmax),
- Y_j is the probability assigned to class j,



Vector Activations

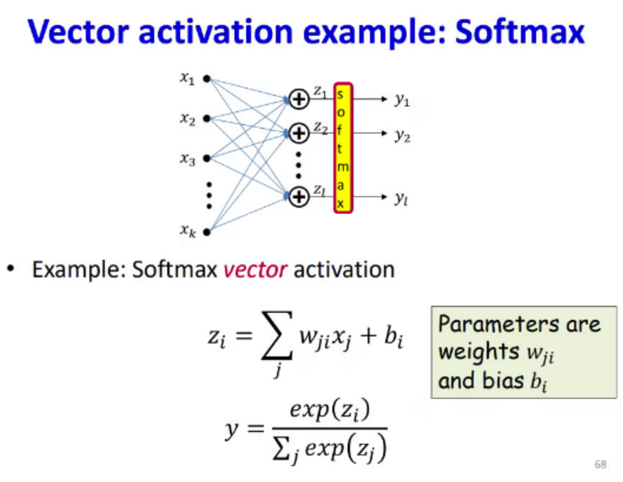
Input Layer
Hidden Layers
Output Layer

- We can also have neurons that have *multiple coupled* outputs

$$[y_1, y_2, \dots, y_l] = f(x_1, x_2, \dots, x_k; W)$$

  - Function $f()$ operates on set of inputs to produce set of outputs
  - Modifying a single parameter in W will affect *all* outputs

67



Vector activation example: Softmax

x_1
x_2
x_3
x_k
z_1
z_2
z_l
softmax
y_1
y_2
y_l

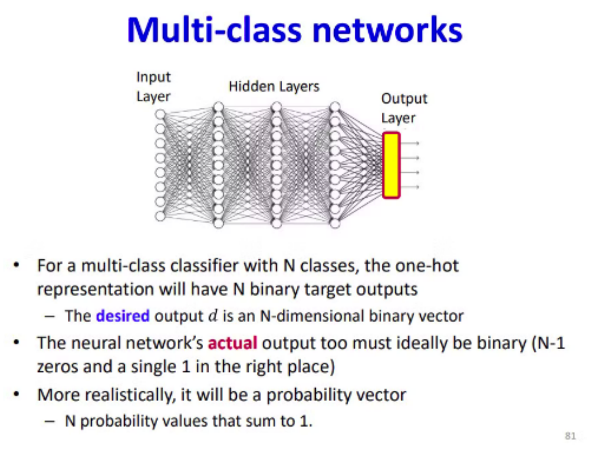
- Example: Softmax *vector* activation

$$z_i = \sum_j w_{ji} x_j + b_i$$

Parameters are weights w_{ji} and bias b_i

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

68

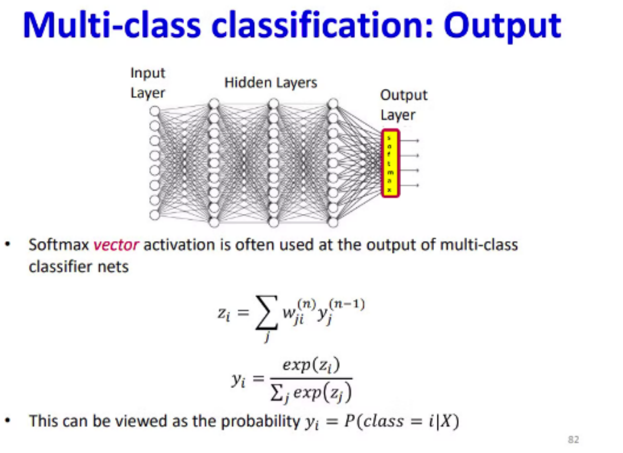


Multi-class networks

Input Layer
Hidden Layers
Output Layer

- For a multi-class classifier with N classes, the one-hot representation will have N binary target outputs
  - The **desired** output d is an N-dimensional binary vector
- The neural network's **actual** output too must ideally be binary (N-1 zeros and a single 1 in the right place)
- More realistically, it will be a probability vector
  - N probability values that sum to 1.

81



Multi-class classification: Output

Input Layer
Hidden Layers
Output Layer
softmax

- Softmax *vector* activation is often used at the output of multi-class classifier nets

$$z_i = \sum_j w_{ji}^{(n)} y_j^{(n-1)}$$

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

- This can be viewed as the probability $y_i = P(class = i|X)$

82

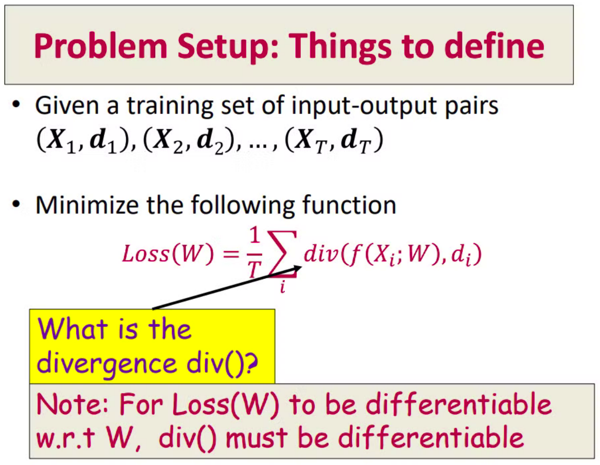
- The denominator is the sum of the exponentials of all classes to normalize the probabilities.

The network will have as many outputs as there are classes.

Ideally If I'm giving a pic of a cat, I want the output to be a 1 for 1 class and 0 else but that's not the case. Realistically it outputs a probability vector which has the probability of all of the classes. Soft-max activation gave a probability vector. So for multi-class classification we use a soft-max activation at the output. It exponentiates the affine term (which ensures it's always positive) and then normalizes it (which ensures they all sum to 1). In multi-class classification the network has as many outputs as the number of classes.

6.6 Divergence in the Loss Function: L2 and KL

The divergence needs to be differentiable as we want to know how much changing the output of the network changes the divergence. Which we need to know how to modify the network parameters to decrease the loss.



Problem Setup: Things to define

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

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

What is the divergence div()?

Note: For Loss(W) to be differentiable w.r.t W, div() must be differentiable

6.6.1 L2 Divergence

- The network outputs a vector Y, and the desired vector is d. - The L2 divergence, also known as mean squared error, is:

$$\text{L2}(Y, d) = \frac{1}{2}\sum_i (Y_i - d_i)^2$$

- The $\frac{1}{2}$ is used because the derivative of the squared distance includes a factor of 2, which cancels out in optimization. - L2 Divergence is used in regression tasks.

- **No Activation for Real-Value Outputs:** In regression tasks, we typically don't apply a final activation function, leaving real-valued outputs.

- The derivative of the L2 loss with respect to the output is:

$$\frac{\partial}{\partial Y}\text{L2}(Y, d) = Y - d$$

- This shows the error between the actual network output and the desired output. If $Y < d$, the derivative is negative, encouraging an increase in Y to reduce the error, and vice versa if $Y > d$.

6.6.2 Kullback-Leibler (KL) Divergence

- For classification tasks, we use KL divergence to measure the difference between the predicted probability distribution Y and the target one-hot vector d.

$$\mathrm{KL}(d||Y) = \sum_i d_i \log\left(\frac{d_i}{Y_i}\right)$$

For classification we use KL divergence. In classification the desired output d is either 1 or 0, and the output from the network Y is a probability. In this case I can think of my one hot vector as an extreme case of a probability vector. Now I have 2 probability vectors. And KL Divergence quantifies the difference between 2 probability vectors. The first term is $-dlogY$, because Y is going to have a probability value between 0 and 1, and log of that would be negative. - and - cancel each other out. `Divergence must always be positive. min value =0`. If the desired output is 1 and y is less than 1, I should be increase y to reduce error and the sign of the derivative is negative. Hence $-\frac{1}{Y}$ will always be negative when the desired output is 1. If the desired output is 0, y should be reduced to decrease error. The derivative is always positive.

Since d is a one-hot vector, this reduces to:

$$\mathrm{KL}(d||Y) = -\sum_i d_i \log Y_i$$

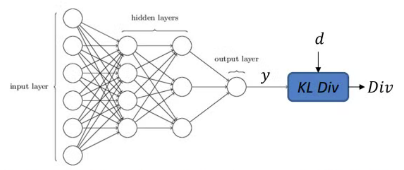
- The first term is $-d_i \log Y_i$, which penalizes incorrect predictions, and ensures the divergence is always positive.

- The derivative of the KL divergence with respect to the output Y is:

$$\frac{\partial}{\partial Y_i}\mathrm{KL}(d||Y) = \frac{-d_i}{Y_i}$$

- When the desired output d_i is 1, the derivative is negative, encouraging an increase in Y_i to reduce the error. - When $d_i = 0$, we want to reduce Y_i, and the derivative will be positive.

For binary classifier



- For binary classifier with scalar output, Y ∈ (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

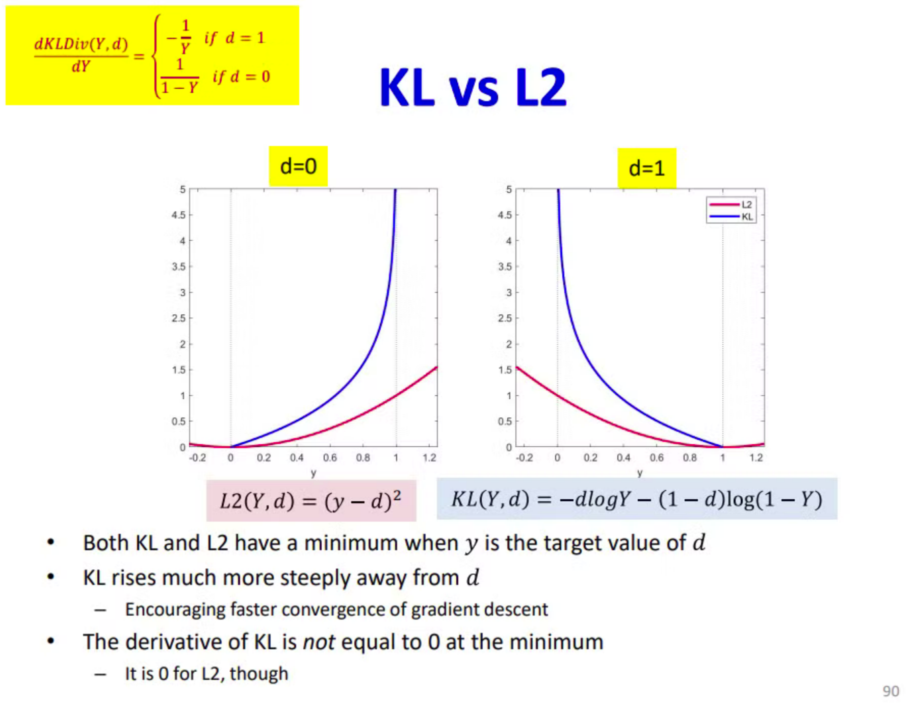
$$Div(Y,d) = -dlogY - (1-d)\log(1-Y)$$

  - Minimum when $d = Y$

- Derivative

$$\frac{dDiv(Y,d)}{dY} = \begin{cases} -\frac{1}{Y} & if\ d = 1 \\ \frac{1}{1-Y} & if\ d = 0 \end{cases}$$

6.7 KL vs L2 Loss



$$\frac{dKLDiv(Y,d)}{dY} = \begin{cases} -\frac{1}{Y} & if\ d = 1 \\ \frac{1}{1-Y} & if\ d = 0 \end{cases}$$

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
- The derivative of KL is *not* equal to 0 at the minimum
  - It is 0 for L2, though

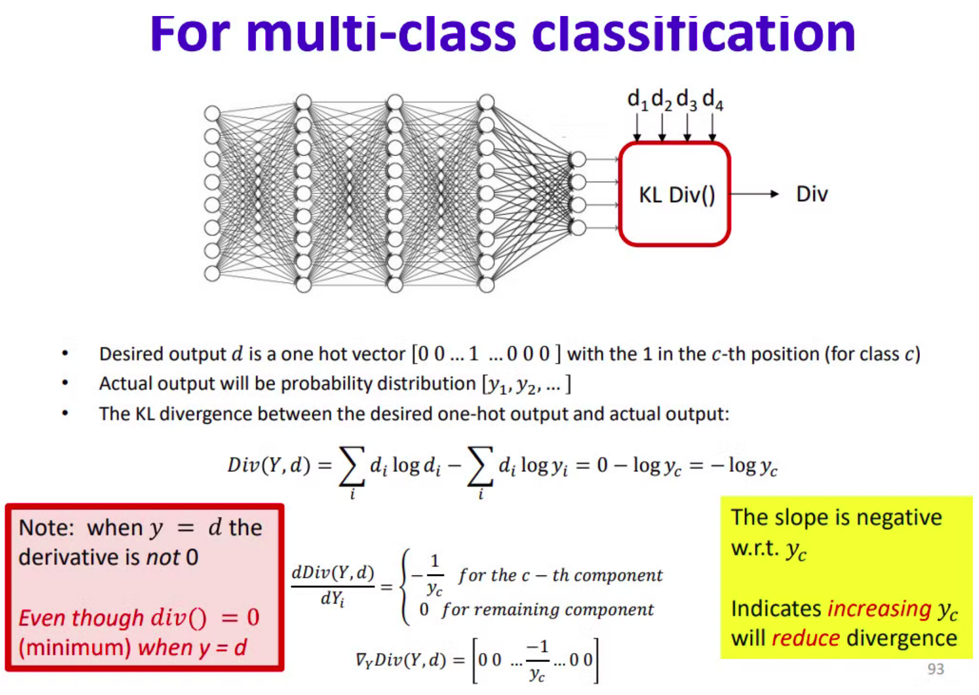
90

In classification problems, KL divergence has properties that make it more suitable than L2. - When the network's prediction is completely wrong, KL divergence forces the output to change quickly to correct the error, while L2 divergence would not. However, at the optimal point, KL divergence does not have a zero gradient, unlike L2.

First plot is when desired output is 0 and second when desired output is 1. In both cases the divergence is minimized when the actual output of the network matches the desired output. We don't use an L2 error in classification problems because: - When the output from the network is opposite to the desired output the L2 error would be infinitely wrong - When you're very wrong KL divergence has a property which forces the output to change very fast in order to fix it. L2 does not have this property. But in exchange we're paying a penalty that at d=0 the blue curve has a slope of 1, and at d=1 the slope =-1. So this means checking if derivative =0 to check if we're at the minimum is not the best way. For KL Divergance at the Optimum tha value of the gradient =1 or -1 - So in Gradient descent we only check if the function is changing or not, and we don't check if the derivative =0.

- **Gradient Behavior in KL:** - In gradient descent, we monitor whether the function is still changing rather than checking if the gradient is zero. - In KL divergence, at the optimal point, the gradient will be ± 1 rather than zero, so stopping conditions are based on changes in the output rather than the gradient reaching zero.

6.8 Multi-Class Classification with KL Divergence



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:

$$Div(Y,d) = \sum_i d_i \log d_i - \sum_i d_i \log y_i = 0 - \log y_c = -\log y_c$$

Note: when $y = d$ the derivative is *not* 0

Even though $div() = 0$ (minimum) when y = d

$$\frac{dDiv(Y,d)}{dY_i} = \begin{cases} -\frac{1}{y_c} & for\ the\ c-th\ component \\ 0 & for\ remaining\ component \end{cases}$$

$$\nabla_Y Div(Y,d) = \left[0\ 0\ \dots \frac{-1}{y_c} \dots 0\ 0\right]$$

The slope is negative w.r.t. y_c

Indicates *increasing* y_c will *reduce* divergence

93

- In multi-class classification, d and Y are probability vectors, where d is an extreme probability vector (one-hot vector). - For non-target classes, $d_i = 0$, and for the target class, $d_c = 1$.

- The KL divergence reduces to:

$$\text{KL}(d||Y) = -\log Y_c$$

Where Y_c is the probability assigned to the correct class. The gradient with respect to Y is 0 for non-target classes and $-\frac{1}{Y_c}$ for the target class.

- **Derivative of KL Divergence with respect to Softmax input:**

Before applying the softmax, we compute an affine value z. The gradient of the KL divergence with respect to the input z to the softmax layer simplifies to:

$$\frac{\partial}{\partial z}\text{KL}(d||Y) = Y - d$$

This is the error between the predicted output and the desired output, which drives the learning process in gradient-based optimization.

6.9 Cross Entropy

- For any model trained by gradient optimization methods, minimizing the cross-entropy between the data distribution and the model distribution gives the same results as minimizing KL-divergence.
- If the gradients are the same, the end result of the optimization will be the same. Cross entropy loss is the same thing as KL divergence off by a constant. The constant is the entropy of the target distribution, which behaves like a constant because the target distribution is fixed. Because the gradients of this constant are 0, this means that it does not affect the optimization procedure as it does not contribute to the parameter updates.
- Note that the entropy of a one-hot distribution is 0, so for one-hot vectors the KL = X-entropy. The reason they are separated in most libraries is because you can compute the X-entropy of a one hot vector slightly faster than the KL since you only need to compute the log-softmax of one of the logits.
- The KL-divergence is nicer as a loss since it will equal 0 when the student network matches the teacher on all labels. In contrast, if we use X-entropy, then the loss will fluctuate even when the student and teacher output the exact same thing, and it will fluctuate according to the batch, as you described.

6.10 Summary

Objective Functions

Notes: The core of the objective function is to define Div(Y,d). Real value output (regression): Euclidean distance

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

Two classification problem: cross entropy

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

Multi-classification problem: cross entropy

$$Div(Y,d) = -\sum_i d_i \log y_i$$

Summary

- 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