Neural Networks
Learning the network: Part 2

11-785, Fall 2020
Lecture 4
Recap: Universal approximators

- Neural networks are universal approximators
  - Can approximate any function
    - Provided they have sufficient architecture
  - We have to determine the weights and biases to make them model the function
Recap: Approach

- Define a divergence $\text{div}(f, d)$ between the actual output $f$ and desired output $d$ of the network
  - Must be differentiable: can quantify how much a miniscule change of $f$ changes $\text{div}(f, d)$
- Make all neuronal activations $\sigma(z)$ differentiable
  - Differentiable: can quantify how much a miniscule change of $z$ changes $\sigma(z)$
- Differentiability – enables us to determine if a small change in any parameter of the network is increasing or decreasing $\text{div}(f, d)$
  - Will let us optimize the network
Recap: The expected divergence

- Minimize the expected “divergence” between the output of the net and the desired function over the input space

\[ \hat{W} = \arg\min_W E[\text{div}(f(X; W), d(X))] \]
Recap: Empirical Risk Minimization

- **Problem**: Computing the expected divergence $E[\text{div}(f(X; W), d(X))]$ requires knowledge of $d(X)$ at all $X$ which we will not have.

- **Solution**: Approximate it by the average divergence over a large number of “training” samples $(X, d(X))$ drawn from $P(X)$

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

- Estimate the parameters to minimize this “loss” instead

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

• Given a training set of input-output pairs 
  \((X_1, d_1), (X_2, d_2), \ldots, (X_N, d_N)\)

• Minimize the following function
  \[
  \text{Loss}(W) = \frac{1}{N} \sum_i \text{div}(f(X_i; W), d_i)
  \]
  w.r.t \(W\)

• This is problem of function minimization
  – An instance of optimization
• A CRASH COURSE ON FUNCTION OPTIMIZATION
A brief note on derivatives...

- A derivative of a function at any point tells us how much a minute increment to the argument of the function will increment the value of the function.
  - For any $y = f(x)$, expressed as a multiplier $\alpha$ to a tiny increment $\Delta x$ to obtain the increments $\Delta y$ to the output $\Delta y = \alpha \Delta x$.
  - Based on the fact that at a fine enough resolution, any smooth, continuous function is locally linear at any point.
Scalar function of scalar argument

- When $x$ and $y$ are scalar
  \[ y = f(x) \]
  - Derivative:
    \[ \Delta y = \alpha \Delta x \]
  - Often represented (using somewhat inaccurate notation) as \( \frac{dy}{dx} \)
  - Or alternately (and more reasonably) as \( f'(x) \)
Multivariate scalar function:
Scalar function of vector argument

\[ \Delta y = \alpha \Delta x \]

- Giving us that \( \alpha \) is a row vector: \( \alpha = [\alpha_1 \ldots \alpha_D] \)
  \[ \Delta y = \alpha_1 \Delta x_1 + \alpha_2 \Delta x_2 + \cdots + \alpha_D \Delta x_D \]
- The partial derivative \( \alpha_i \) gives us how \( y \) increments when only \( x_i \) is incremented
- Often represented as \( \frac{\partial y}{\partial x_i} \)
  \[ \Delta y = \frac{\partial y}{\partial x_1} \Delta x_1 + \frac{\partial y}{\partial x_2} \Delta x_2 + \cdots + \frac{\partial y}{\partial x_D} \Delta x_D \]
Multivariate scalar function:
Scalar function of *vector* argument

\[ \Delta y = \nabla_x y \Delta x \]

- Where

\[ \nabla_x y = \begin{bmatrix} \frac{\partial y}{\partial x_1} & \ldots & \frac{\partial y}{\partial x_D} \end{bmatrix} \]

- You may be more familiar with the term “gradient” which is actually defined as the transpose of the derivative

Note: \( \Delta x \) is now a vector

\[ \Delta x = \begin{bmatrix} \Delta x_1 \\ \vdots \\ \Delta x_D \end{bmatrix} \]
Caveat about following slides

• The following slides speak of optimizing a function w.r.t a variable “x”

• This is only mathematical notation. In our actual network optimization problem we would be optimizing w.r.t. network weights “w”

• To reiterate – “x” in the slides represents the variable that we’re optimizing a function over and not the input to a neural network

• Do not get confused!
• General problem of optimization: find the value of $x$ where $f(x)$ is minimum
Finding the minimum of a function

- Find the value $x$ at which $f'(x) = 0$
  - Solve

\[
\frac{df(x)}{dx} = 0
\]

- The solution is a “turning point”
  - Derivatives go from positive to negative or vice versa at this point
- But is it a minimum?
Turning Points

• Both maxima and minima have zero derivative
• Both are turning points
Derivatives of a curve

• Both *maxima* and *minima* are turning points
• Both *maxima* and *minima* have zero derivative
Derivative of the derivative of the curve

• Both maxima and minima are turning points
• Both maxima and minima have zero derivative

• The second derivative $f''(x)$ is –ve at maxima and +ve at minima!
Solution: Finding the minimum or maximum of a function

- Find the value $x$ at which $f'(x) = 0$: Solve
  \[ \frac{df(x)}{dx} = 0 \]

- The solution $x_{soln}$ is a **turning point**

- Check the double derivative at $x_{soln}$: compute
  \[ f''(x_{soln}) = \frac{df'(x_{soln})}{dx} \]

- If $f''(x_{soln})$ is positive $x_{soln}$ is a minimum, otherwise it is a maximum
A note on derivatives of functions of single variable

• All locations with zero derivative are *critical* points
  – These can be local maxima, local minima, or inflection points

• The *second* derivative is
  – Positive (or 0) at minima
  – Negative (or 0) at maxima
  – Zero at inflection points
A note on derivatives of functions of single variable

- All locations with zero derivative are *critical* points
  - These can be local maxima, local minima, or inflection points

- The *second* derivative is
  - $\geq 0$ at minima
  - $\leq 0$ at maxima
  - Zero at inflection points

- It’s a little more complicated for functions of multiple variables..
What about functions of multiple variables?

• The optimum point is still “turning” point
  – Shifting in any direction will increase the value
  – For smooth functions, miniscule shifts will not result in any change at all
• We must find a point where shifting in any direction by a microscopic amount will not change the value of the function
A brief note on derivatives of multivariate functions
The Gradient of a scalar function

- The **derivative** $\nabla_X f(X)$ of a scalar function $f(X)$ of a multi-variate input $X$ is a multiplicative factor that gives us the change in $f(X)$ for tiny variations in $X$

  $$df(X) = \nabla_X f(X)dX$$

- The **gradient** is the transpose of the derivative $\nabla_X f(X)^T$
Gradients of scalar functions with multivariate inputs

- Consider $f(X) = f(x_1, x_2, ..., x_n)$

$$\nabla_X f(X) = \begin{bmatrix} \frac{\partial f(X)}{\partial x_1} & \frac{\partial f(X)}{\partial x_2} & \cdots & \frac{\partial f(X)}{\partial x_n} \end{bmatrix}$$

- Relation:

$$df(X) = \nabla_X f(X) dX$$

This is a vector inner product. To understand its behavior let's consider a well-known property of inner products.
A well-known vector property

\[ u^T v = |u||v|\cos \theta \]

- The inner product between two vectors of fixed lengths is maximum when the two vectors are aligned
  - i.e. when \( \theta = 0 \)
Properties of Gradient

- \( df(X) = \nabla_X f(X) dX \)
- For an increment \( dX \) of any given length \( df(X) \) is max if \( dX \) is aligned with \( \nabla_X f(X)^T \)
  - The function \( f(X) \) increases most rapidly if the input increment \( dX \) is exactly in the direction of \( \nabla_X f(X)^T \)
- The gradient is the direction of fastest increase in \( f(X) \)
Gradient

Gradient vector $\nabla_X f (X)^T$
Gradient

Moving in this direction increases $f(X)$ fastest

Gradient vector $\nabla_X f(X)^T$
Gradient

Moving in this direction increases $f(X)$ fastest

$\nabla_x f(X)^T$

Moving in this direction decreases $f(X)$ fastest

Gradient vector $\nabla_x f(X)^T$
Gradient

Gradient here is 0

Gradient here is 0
Properties of Gradient: 2

- The gradient vector $\nabla_x f(X)^T$ is perpendicular to the level curve.
The Hessian

- The Hessian of a function $f(x_1, x_2, \ldots, x_n)$ is given by the second derivative

$$\nabla^2_x f(x_1, \ldots, x_n) := \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1} & \frac{\partial^2 f}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 f}{\partial x_n^2} \end{bmatrix}$$
Returning to direct optimization...
Finding the minimum of a scalar function of a multivariate input

• The optimum point is a turning point – the gradient will be 0
Unconstrained Minimization of function (Multivariate)

1. Solve for the $X$ where the derivative (or gradient) equals to zero

$$\nabla_X f(X) = 0$$

2. Compute the Hessian Matrix $\nabla^2_X f(X)$ at the candidate solution and verify that
   - Hessian is positive definite (eigenvalues positive) -> to identify local minima
   - Hessian is negative definite (eigenvalues negative) -> to identify local maxima
Closed Form Solutions are not always available

- Often it is not possible to simply solve $\nabla_x f(X) = 0$
  - The function to minimize/maximize may have an intractable form
- In these situations, iterative solutions are used
  - Begin with a “guess” for the optimal $X$ and refine it iteratively until the correct value is obtained
Iterative solutions

- Start from an initial guess $X_0$ for the optimal $X$
- Update the guess towards a (hopefully) “better” value of $f(X)$
- Stop when $f(X)$ no longer decreases

Problems:
- Which direction to step in
- How big must the steps be

$f(X)$

$x_0 \ x_1 \ x_2 \ x_3 \ x_4 \ x_5$
The Approach of Gradient Descent

- Iterative solution:
  - Start at some point
  - Find direction in which to shift this point to decrease error
    - This can be found from the derivative of the function
      - A negative derivative $\rightarrow$ moving right decreases error
      - A positive derivative $\rightarrow$ moving left decreases error
  - Shift point in this direction
The Approach of Gradient Descent

- Iterative solution: Trivial algorithm
  - Initialize $x^0$
  - While $f'(x^k) \neq 0$
    - If $\text{sign}(f'(x^k))$ is positive:
      \[ x^{k+1} = x^k - \text{step} \]
    - Else
      \[ x^{k+1} = x^k + \text{step} \]
  - What must step be to ensure we actually get to the optimum?
The Approach of Gradient Descent

• Iterative solution: Trivial algorithm
  ▪ Initialize $x^0$
  ▪ While $f'(x^k) \neq 0$
    \[
x^{k+1} = x^k - \text{sign} \left( f'(x^k) \right) \cdot \text{step}
    \]
• Identical to previous algorithm
The Approach of Gradient Descent

- Iterative solution: Trivial algorithm
  - Initialize $x^0$
  - While $f'(x^k) \neq 0$
    $x^{k+1} = x^k - \eta^k f'(x^k)$
  - $\eta^k$ is the “step size”
Gradient descent/ascent (multivariate)

• The gradient descent/ascent method to find the minimum or maximum of a function $f$ iteratively
  – To find a maximum move in the direction of the gradient
    $$x^{k+1} = x^k + \eta^k \nabla_x f(x^k)^T$$
  – To find a minimum move exactly opposite the direction of the gradient
    $$x^{k+1} = x^k - \eta^k \nabla_x f(x^k)^T$$

• Many solutions for step size $\eta^k$
Gradient descent convergence criteria

- The gradient descent algorithm converges when one of the following criteria is satisfied

\[ |f(x^{k+1}) - f(x^k)| < \varepsilon_1 \]

- Or

\[ \|\nabla_x f(x^k)\| < \varepsilon_2 \]
Overall Gradient Descent Algorithm

• Initialize:
  ▪ $x^0$
  ▪ $k = 0$

• do
  ▪ $x^{k+1} = x^k - \eta^k \nabla_x f(x^k)^T$
  ▪ $k = k + 1$

• while $|f(x^{k+1}) - f(x^k)| > \varepsilon$
Convergence of Gradient Descent

- For appropriate step size, for convex (bowl-shaped) functions gradient descent will always find the minimum.

- For non-convex functions it will find a local minimum or an inflection point.
• Returning to our problem..
Problem Statement

• Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$

• Minimize the following function

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

w.r.t $W$

• This is problem of function minimization
  – An instance of optimization
Preliminaries

• Before we proceed: the problem setup
Problem Setup: Things to define

• Given a training set of input-output pairs \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)

• What are these input-output pairs?

\[
\text{Loss}(W) = \frac{1}{T} \sum_i \text{div}(f(X_i; W), d_i)
\]
Problem Setup: Things to define

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), ..., (X_T, d_T)$

- What are these input-output pairs?

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

- What is $f()$ and what are its parameters $W$?
Problem Setup: Things to define

• Given a training set of input-output pairs 
  \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)

• What are these input-output pairs?

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

What is the divergence \(\text{div}()\)?

What is \(f()\) and what are its parameters \(W\)?
Problem Setup: Things to define

• Given a training set of input-output pairs
  \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)

• Minimize the following function

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

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
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 combination of the inputs
    \[
    y = f\left(\sum_i w_i x_i + b\right)
    \]
  - More generally: *any* differentiable function
    \[
    y = f(x_1, x_2, \ldots, x_N; W)
    \]
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 combination of the input
    \[ y = f \left( \sum_i w_i x_i + b \right) \]
  - 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 \( b \)
### Activations and their derivatives

<table>
<thead>
<tr>
<th>Function</th>
<th>Derivative</th>
</tr>
</thead>
<tbody>
<tr>
<td>( f(z) = \frac{1}{1 + \exp(-z)} )</td>
<td>( f'(z) = f(z)(1 - f(z)) )</td>
</tr>
<tr>
<td>( f(z) = \tanh(z) )</td>
<td>( f'(z) = (1 - f^2(z)) )</td>
</tr>
<tr>
<td>( f(z) = \begin{cases} z, &amp; z \geq 0 \ 0, &amp; z &lt; 0 \end{cases} )</td>
<td>( f'(z) = \begin{cases} 1, &amp; z \geq 0 \ 0, &amp; z &lt; 0 \end{cases} )</td>
</tr>
<tr>
<td>( f(z) = \log(1 + \exp(z)) )</td>
<td>( f'(z) = \frac{1}{1 + \exp(-z)} )</td>
</tr>
</tbody>
</table>

- Some popular activation functions and their derivatives
Vector Activations

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

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

– 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

- 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 \(b_i\)
Multiplicative combination: Can be viewed as a case of vector activations

\[ z_i = \sum_j w_{ji}x_j + b_i \]

\[ y_i = \prod_l (z_i)^{a_{li}} \]

- Parameters are weights \( w_{ji} \) and bias \( b_i \)

- A layer of multiplicative combination is a special case of vector activation
• In a layered network, each layer of perceptrons can be viewed as a single vector activation
**Notation**

- The input layer is the 0\(^{th}\) layer
- We will represent the output of the \(i\)-th perceptron of the \(k\)^{th} 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 \(j\)th unit of the \(k\)-th layer as \(w_{ij}^{(k)}\)
  - The bias to the \(j\)th unit of the \(k\)-th layer is \(b_{j}^{(k)}\)
Problem Setup: Things to define

• Given a training set of input-output pairs 
  \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)

• Minimize the following function

\[
\text{Loss}(W) = \frac{1}{T} \sum \text{div}(f(X_i; W), d_i)
\]

What is \(f()\) and what are its parameters \(W\)?
Problem Setup: Things to define

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$
- What are these input-output pairs?

$\text{Loss}(W) = \frac{1}{T} \sum_i \text{div}(f(X_i; W), d_i)$
Given a training set of input-output pairs \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)

- \(X_n = [x_{n1}, x_{n2}, \ldots, x_{nD}]^T\) is the nth input vector
- \(d_n = [d_{n1}, d_{n2}, \ldots, d_{nL}]^T\) is the nth desired output
- \(Y_n = [y_{n1}, y_{n2}, \ldots, y_{nL}]^T\) 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

- 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

• If the desired output is real-valued, no special tricks are necessary
  – Scalar Output: single output neuron
    • \( d = \text{scalar} \) (real value)
  – Vector Output: as many output neurons as the dimension of the desired output
    • \( d = [d_1 \; d_2 \; \ldots \; d_L] \) (vector of real values)
• If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
  – 1 = Yes it’s a cat
  – 0 = No it’s not a cat.
Representing the output

• If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output

• Output activation: Typically a sigmoid
  – Viewed as the probability \( P(Y = 1|X) \) of class value 1
    • Indicating the fact that for actual data, in general a feature value \( X \) may occur for both classes, but with different probabilities
    • Is differentiable
Representing the output

- If the desired output is binary (is this a cat or not), use a simple 1/0 representation of the desired output
  - 1 = Yes it’s a cat
  - 0 = No it’s not a cat.

- Sometimes represented by two outputs, one representing the desired output, the other representing the negation of the desired output
  - Yes: $\rightarrow [1 \ 0]$
  - No: $\rightarrow [0 \ 1]$

- The output explicitly becomes a 2-output softmax
**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:
  \[
  \text{[cat dog camel hat flower]}^T
  \]
- 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*.
Multi-class networks

- 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 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.
Multi-class classification: Output

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

\[ z_i = \sum_j w^{(n)}_{ji} y^{(n-1)}_j \]

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

- This can be viewed as the probability \( y_i = P(\text{class} = i | X) \)
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

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

Training data

\[
\begin{align*}
(5, 0) & \quad (2, 1) \\
(2, 1) & \quad (4, 0) \\
(0, 0) & \quad (2, 1)
\end{align*}
\]

Input: vector of pixel values

Output: sigmoid
Typical Problem statement: multiclass classification

- Given, many positive and negative examples (training data),
  - learn all weights such that the network does the desired job
Problem Setup: Things to define

- Given a training set of input-output pairs \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)

- Minimize the following function

\[
\text{Loss}(W) = \frac{1}{T} \sum_{i} \text{div}(f(X_i; W), d_i)
\]

What is the divergence \(\text{div}()\)?
Problem Setup: Things to define

- Given a training set of input-output pairs $(X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)$

- Minimize the following function

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

What is the divergence $\text{div}()$?

Note: For $\text{Loss}(W)$ to be differentiable w.r.t $W$, $\text{div}()$ must be differentiable.
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

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

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

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

- Minimum when $d = Y$
KL vs L2

- 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 \textit{not} equal to 0 at the minimum
  - It is 0 for L2, though

\[
L2(Y, d) = (y - d)^2 \quad \text{KL}(Y, d) = -d \log Y - (1 - d) \log (1 - Y)
\]
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:

\[
Div(Y, d) = -d \log Y - (1 - d) \log(1 - Y)
\]

- Minimum when \( d = Y \)

- Derivative

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

Note: when \( y = d \) the derivative is not 0

Even though \( div() = 0 \) (minimum) when \( y = d \)
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 $[y_1, y_2, ... ]$
- The KL divergence between the desired one-hot output and actual output:

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

- Note $\sum_i d_i \log d_i = 0$ for one-hot $d \Rightarrow Div(Y, d) = -\sum_i d_i \log y_i$

- Derivative

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

$$\nabla_Y Div(Y, d) = \begin{bmatrix} 0 & 0 & ... & -\frac{1}{y_c} & ... & 0 & 0 \end{bmatrix}$$

The slope is negative w.r.t. $y_c$

Indicates increasing $y_c$ will reduce 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 $[y_1, y_2, ...]$
- The KL divergence between the desired one-hot output and actual output:

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

- Derivative

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

$$\nabla_Y Div(Y, d) = \begin{bmatrix} 0 & 0 & \ldots & -\frac{1}{y_c} & \ldots & 0 & 0 \end{bmatrix}$$

The slope is negative w.r.t. $y_c$
Indicates increasing $y_c$ will reduce divergence

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

Even though $\text{div}(\cdot) = 0$ (minimum) when $y = d$
KL divergence vs cross entropy

• KL divergence between $d$ and $y$:

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

• Cross-entropy between $d$ and $y$:

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

• The cross entropy is merely the KL - entropy of $d$

$$Xent(Y, d) = KL(Y, d) - \sum_i d_i \log d_i = KL(Y, d) - H(d)$$

• The $W$ that minimizes cross-entropy will minimize the KL divergence
  – since $d$ is the desired output and does not depend on the network, $H(d)$ does not depend on the net
  – In fact, for one-hot $d$, $H(d) = 0$ (and KL = Xent)

• We will generally minimize to the cross-entropy loss rather than the KL divergence
  – The Xent is not a divergence, and although it attains its minimum when $y = d$, its minimum value is not 0
“Label smoothing”

- It is sometimes useful to set the target output to $[\epsilon \ \epsilon \ \cdots (1-(K-1)\epsilon) \ \cdots \ \epsilon \ \epsilon \ \epsilon]$ with the value $1-(K-1)\epsilon$ in the $c$-th position (for class $c$) and $\epsilon$ elsewhere for some small $\epsilon$
  - “Label smoothing” -- aids gradient descent
- The KL divergence remains:

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

- Derivative

$$\frac{dDiv(Y, d)}{dY_i} = \begin{cases} 1 - (K-1)\epsilon & \text{for the } c \text{th component} \\ \frac{y_c}{\epsilon} & \frac{\epsilon}{y_i} \text{for remaining components} \end{cases}$$
“Label smoothing”

- It is sometimes useful to set the target output to \([\epsilon \ \epsilon \ ... \ (1 - (K - 1)\epsilon) \ ... \ \epsilon \ \epsilon \ \epsilon]\) with the value \(1 - (K - 1)\epsilon\) in the \(c\)-th position (for class \(c\)) and \(\epsilon\) elsewhere for some small \(\epsilon\)
  - “Label smoothing” -- aids gradient descent
- The KL divergence remains:
  \[
  \text{Div}(Y, d) = \sum_i d_i \log d_i - \sum_i d_i \log y_i
  \]
- Derivative
  \[
  \frac{d\text{Div}(Y, d)}{dY_i} = \begin{cases} 
  -\frac{1 - (K - 1)\epsilon}{y_c} & \text{for the } c\text{-th component} \\
  -\frac{\epsilon}{y_i} & \text{for remaining components}
  \end{cases}
  \]

Negative derivatives encourage increasing the probabilities of all classes, including incorrect classes!
(Seems wrong, no?)
Problem Setup: Things to define

• Given a training set of input-output pairs
  \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)

• Minimize the following function

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

ALL TERMS HAVE BEEN DEFINED
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
Problem Setup

• Given a training set of input-output pairs 
  \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)

• The divergence on the \(i^{th}\) instance is \(\text{div}(Y_i, d_i)\)
  
  \[-Y_i = f(X_i; W)\]

• The loss

\[
\text{Loss} = \frac{1}{T} \sum_i \text{div}(Y_i, d_i)
\]

• Minimize \(\text{Loss}\) w.r.t \(\{w_{ij}^{(k)}, b_j^{(k)}\}\)
Recap: Gradient Descent Algorithm

- **Initialize:**
  - $W^0$
  - $k = 0$

- **do**
  - $W^{k+1} = W^k - \eta^k \nabla L(W^k)^T$
  - $k = k + 1$

- **while** $|L(W^k) - L(W^{k-1})| > \varepsilon$

To minimize any function $L(W)$ w.r.t $W$
Recap: Gradient Descent Algorithm

• In order to minimize $L(W)$ w.r.t. $W$

• Initialize:
  – $W^0$
  – $k = 0$

• do
  – For every component $i$
    • $W_i^{k+1} = W_i^k - \eta^k \frac{\partial L}{\partial W_i}$
    – $k = k + 1$
  • while $|L(W^k) - L(W^{k-1})| > \epsilon$
### Training Neural Nets through Gradient Descent

#### Total training Loss:

\[
Loss = \frac{1}{T} \sum_t \text{Div}(Y_t, d_t)
\]

- **Gradient descent algorithm:**
- **Initialize all weights and biases** \( \{w^{(k)}_{i,j}\} \)
  - Using the extended notation: the bias is also a weight
- **Do:**
  - For every layer \( k \) for all \( i, j \), update:
    - \( w^{(k)}_{i,j} = w^{(k)}_{i,j} - \eta \frac{dLoss}{dw^{(k)}_{i,j}} \)
- Until **Loss** has converged

---

*Assuming the bias is also represented as a weight*
Training Neural Nets through Gradient Descent

Total training Loss:

\[ \text{Loss} = \frac{1}{T} \sum_{t} \text{Div}(Y_t, d_t) \]

- Gradient descent algorithm:
  - Initialize all weights \( \{ w_{ij}^{(k)} \} \)
  - Do:
    - For every layer \( k \) for all \( i, j \), update:
      - \( w_{i,j}^{(k)} = w_{i,j}^{(k)} - \eta \frac{d\text{Loss}}{dw_{i,j}^{(k)}} \)
  - Until \( Err \) has converged

Assuming the bias is also represented as a weight
The derivative

Total training Loss:

\[ \text{Loss} = \frac{1}{T} \sum_{t} \text{Div}(Y_t, d_t) \]

• Computing the derivative

Total derivative:

\[ \frac{d\text{Loss}}{dW_{i,j}^{(k)}} = \frac{1}{T} \sum_{t} \frac{d\text{Div}(Y_t, d_t)}{dW_{i,j}^{(k)}} \]
Training by gradient descent

- Initialize all weights \( \{ w_{ij}^{(k)} \} \)

- Do:
  - For all \( i, j, k \), initialize \( \frac{dLoss}{dw_{i,j}^{(k)}} = 0 \)
  - For all \( t = 1: T \)
    - For every layer \( k \) for all \( i, j \):
      - Compute \( \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}} \)
      - \( \frac{dLoss}{dw_{i,j}^{(k)}} = \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}} \)
    - For every layer \( k \) for all \( i, j \):
      \[
      w_{i,j}^{(k)} = w_{i,j}^{(k)} - \frac{\eta}{T} \frac{dLoss}{dw_{i,j}^{(k)}}
      \]
  - Until \( Err \) has converged
The derivative

Total training Loss:

\[
Loss = \frac{1}{T} \sum_t Div(Y_t, d_t)
\]

Total derivative:

\[
\frac{dLoss}{dw_{i,j}^{(k)}} = \frac{1}{T} \sum_t \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}}
\]

- So we must first figure out how to compute the derivative of divergences of individual training inputs
Calculus Refresher: Basic rules of calculus

For any differentiable function
\[ y = f(x) \]
with derivative
\[ \frac{dy}{dx} \]
the following must hold for sufficiently small \( \Delta x \)
\[ \Delta y \approx \frac{dy}{dx} \Delta x \]

For any differentiable function
\[ y = f(x_1, x_2, ..., x_M) \]
with partial derivatives
\[ \frac{\partial y}{\partial x_1}, \frac{\partial y}{\partial x_2}, ..., \frac{\partial y}{\partial x_M} \]
the following must hold for sufficiently small \( \Delta x_1, \Delta x_2, ..., \Delta x_M \)
\[ \Delta y \approx \frac{\partial y}{\partial x_1} \Delta x_1 + \frac{\partial y}{\partial x_2} \Delta x_2 + \cdots + \frac{\partial y}{\partial x_M} \Delta x_M \]

Both by the definition
\[ \Delta y = \nabla_x f \Delta x \]
Calculus Refresher: Chain rule

For any nested function $y = f(g(x))$

$$\frac{dy}{dx} = \frac{df}{dg(x)} \frac{dg(x)}{dx}$$

Check - we can confirm that:

$z = g(x) \Rightarrow \Delta z = \frac{dg(x)}{dx} \Delta x$

$y = f(z) \Rightarrow \Delta y = \frac{df}{dz} \Delta z = \frac{df}{dg(x)} \frac{dg(x)}{dx} \Delta x$
Calculus Refresher: Distributed Chain rule

\[ y = f(g_1(x), g_1(x), ..., g_M(x)) \]

\[
\frac{dy}{dx} = \frac{\partial f}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)} \frac{dg_2(x)}{dx} + ... + \frac{\partial f}{\partial g_M(x)} \frac{dg_M(x)}{dx}
\]

Check: \[ \Delta y = \frac{dy}{dx} \Delta x \]

Let \[ z_i = g_i(x) \]

\[ \Delta y = \frac{\partial f}{\partial z_1} \Delta z_1 + \frac{\partial f}{\partial z_2} \Delta z_2 + ... + \frac{\partial f}{\partial z_M} \Delta z_M \]

\[ \Delta y = \frac{\partial f}{\partial z_1} \frac{dz_1}{dx} \Delta x + \frac{\partial f}{\partial z_2} \frac{dz_2}{dx} \Delta x + ... + \frac{\partial f}{\partial z_M} \frac{dz_M}{dx} \Delta x \]

\[ \Delta y = \left( \frac{\partial f}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)} \frac{dg_2(x)}{dx} + ... + \frac{\partial f}{\partial g_M(x)} \frac{dg_M(x)}{dx} \right) \Delta x \]
Calculus Refresher: Distributed Chain rule

\[ y = f(g_1(x), g_2(x), \ldots, g_M(x)) \]

\[
\frac{dy}{dx} = \frac{\partial f}{\partial g_1(x)} \frac{dg_1(x)}{dx} + \frac{\partial f}{\partial g_2(x)} \frac{dg_2(x)}{dx} + \cdots + \frac{\partial f}{\partial g_M(x)} \frac{dg_M(x)}{dx}
\]

Check: \[ \Delta y = \frac{dy}{dx} \Delta x \]

\[
\Delta y = \frac{\partial f}{\partial g_1(x)} \Delta g_1(x) + \frac{\partial f}{\partial g_2(x)} \Delta g_2(x) + \cdots + \frac{\partial f}{\partial g_M(x)} \Delta g_M(x)
\]

\[
\Delta y = \left( \frac{\partial f}{\partial g_1(x)} \frac{dg_1(x)}{dx} \Delta x + \frac{\partial f}{\partial g_2(x)} \frac{dg_2(x)}{dx} \Delta x + \cdots + \frac{\partial f}{\partial g_M(x)} \frac{dg_M(x)}{dx} \Delta x \right) \Delta x
\]
Distributed Chain Rule: Influence Diagram

\[ y = f(g_1(x), g_1(x), ..., g_M(x)) \]

- \( x \) affects \( y \) through each of \( g_1 \ldots g_M \)
Distributed Chain Rule: Influence Diagram

\[ dy = \sum_i dz_i \]

- Small perturbations in \( x \) cause small perturbations in each of \( g_1 \) ... \( g_M \), each of which individually additively perturbs \( y \)
Returning to our problem

• How to compute \( \frac{d \text{Div}(Y,d)}{d w_{i,j}^{(k)}} \)
A first closer look at the network

• Showing a tiny 2-input network for illustration
  – Actual network would have many more neurons and inputs
A first closer look at the network

• Showing a tiny 2-input network for illustration
  – Actual network would have many more neurons and inputs
• Explicitly separating the weighted sum of inputs from the activation
A first closer look at the network

- Showing a tiny 2-input network for illustration
  - Actual network would have many more neurons and inputs
- Expanded **with all weights shown**
- Lets label the other variables too...
Computing the derivative for a single input

\[ f_1(x) = \mathbf{1}^T \mathbf{w}_1 + \mathbf{1}^T \mathbf{w}_2 + \mathbf{1}^T \mathbf{w}_3 + \mathbf{1}^T \mathbf{w}_4 \]

\[ f_2(x) = f_1(x) + \mathbf{1}^T \mathbf{w}_5 + \mathbf{1}^T \mathbf{w}_6 + \mathbf{1}^T \mathbf{w}_7 + \mathbf{1}^T \mathbf{w}_8 \]

\[ f_3(x) = f_2(x) + \mathbf{1}^T \mathbf{w}_9 + \mathbf{1}^T \mathbf{w}_{10} + \mathbf{1}^T \mathbf{w}_{11} + \mathbf{1}^T \mathbf{w}_{12} \]

\[ y = f_3(x) \]
Computing the derivative for a single input

What is: \( \frac{d \text{Div}(Y,d)}{d w_{i,j}^{(k)}} \)
Computing the gradient

- Note: computation of the derivative \( \frac{d\text{Div}(Y,d)}{dw_{i,j}^{(k)}} \) requires intermediate and final output values of the network in response to the input.
The “forward pass”

We will refer to the process of computing the output from an input as the forward pass.

We will illustrate the forward pass in the following slides.
The “forward pass”

\[ y^{(0)} = x \]

Setting \( y_i^{(0)} = x_i \) for notational convenience

Assuming \( w^{(k)}_{0j} = b^{(k)}_j \) and \( y^{(k)}_0 = 1 \) -- assuming the bias is a weight and extending the output of every layer by a constant 1, to account for the biases
The “forward pass”

\[ y^{(0)} = x \]

\[ z^{(1)} = f_1(x) \]

\[ y^{(1)} = f_2(z^{(1)}) \]

\[ z^{(2)} = f_3(y^{(1)}) \]

\[ y^{(2)} = f_4(z^{(2)}) \]

\[ z^{(3)} = f_5(y^{(2)}) \]

\[ y^{(3)} = f_6(z^{(3)}) \]

\[ z^{(N-1)} = f_{N-1}(y^{(N-1)}) \]

\[ y^{(N)} = f_N(z^{(N)}) \]

\[ z_1^{(1)} = \sum_i w_{i1}^{(1)} y_i^{(0)} \]
The “forward pass”

\[ y^{(0)} = x \]

\[ z^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)} \]
$y^{(0)} = x$

$z^{(1)} = \sum_{i} w_{ij}^{(1)} y^{(0)}_i$  

$y_j^{(1)} = f_1(z_j^{(1)})$
\( y^{(0)} = x \)

\[
z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)} \quad y_j^{(1)} = f_1(z_j^{(1)})
\]

\[
z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}
\]
\[ y^{(0)} = x \]

\[ z_j^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)} \]

\[ y_j^{(1)} = f_1 \left( z_j^{(1)} \right) \]

\[ z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)} \]

\[ y_j^{(2)} = f_2 \left( z_j^{(2)} \right) \]
$y^{(0)} = x$

$z^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)}$

$y_j^{(1)} = f_1 \left( z_j^{(1)} \right)$

$z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)}$

$y_j^{(2)} = f_2 \left( z_j^{(2)} \right)$

$z_j^{(3)} = \sum_i w_{ij}^{(3)} y_i^{(2)}$

$y_j^{(3)} = f_3 \left( z_j^{(3)} \right)$
\( y^{(0)} = x \)

\[ z^{(1)} = \sum_i w_{ij}^{(1)} y_i^{(0)} \]

\[ y_j^{(1)} = f_1 \left( z_j^{(1)} \right) \]

\[ z_j^{(2)} = \sum_i w_{ij}^{(2)} y_i^{(1)} \]

\[ y_j^{(2)} = f_2 \left( z_j^{(2)} \right) \]

\[ z_j^{(3)} = \sum_i w_{ij}^{(3)} y_i^{(2)} \]

\[ y_j^{(3)} = f_3 \left( z_j^{(3)} \right) \]

\[ \cdots \]

118
$y^{(0)} = x$

$y_j^{(N-1)} = f_{N-1}(z_j^{(N-1)})$

$z_j^{(N)} = \sum_i w_{ij}^{(N)} y_i^{(N-1)}$

$y^{(N)} = f_N(z^{(N)})$
Forward Computation

ITERATE FOR \( k = 1:N \)

for \( j = 1:\text{layer-width} \)

\[
y_i^{(0)} = x_i
\]

\[
z_j^{(k)} = \sum_i w_{ij} y_i^{(k-1)}
\]

\[
y_j^{(k)} = f_k(z_j^{(k)})
\]
Forward “Pass”

• Input: $D$ dimensional vector $\mathbf{x} = [x_j, \ j = 1 \ldots D]$

• Set:
  - $D_0 = D$, is the width of the 0th (input) layer
  - $y_j^{(0)} = x_j, \ j = 1 \ldots D$; $y_0^{(k=1 \ldots N)} = x_0 = 1$

• For layer $k = 1 \ldots N$
  - For $j = 1 \ldots D_k$
    - $z_j^{(k)} = \sum_{i=0}^{D_{k-1}} w_{i,j}^{(k)} y_i^{(k-1)}$
    - $y_j^{(k)} = f_k(z_j^{(k)})$

• Output:
  - $Y = y_j^{(N)}, j = 1 \ldots D_N$
Computing derivatives

We have computed all these intermediate values in the forward computation.

We must remember them - we will need them to compute the derivatives.
Computing derivatives

First, we compute the divergence between the output of the net $y = y^{(N)}$ and the desired output $d$.
We then compute $\nabla_{Y^{(N)}} \text{div}(\cdot)$ the derivative of the divergence w.r.t. the final output of the network $y^{(N)}$. 

$$y^{(0)} = x$$
We then compute $\nabla_{Y^{(N)}} \text{div}(.)$ the derivative of the divergence w.r.t. the final output of the network $y^{(N)}$

We then compute $\nabla_{z^{(N)}} \text{div}(.)$ the derivative of the divergence w.r.t. the pre-activation affine combination $z^{(N)}$ using the chain rule
Computing derivatives

Continuing on, we will compute $\nabla_{W^{(N)}} \text{div}(\cdot)$ the derivative of the divergence with respect to the weights of the connections to the output layer.
Continuing on, we will compute $\nabla_{W^{(N)}} \text{div}(\cdot)$ the derivative of the divergence with respect to the weights of the connections to the output layer.

Then continue with the chain rule to compute $\nabla_{Y^{(N-1)}} \text{div}(\cdot)$ the derivative of the divergence w.r.t. the output of the N-1th layer.
Computing derivatives

We continue our way backwards in the order shown

\[ \nabla_{z^{(N-1)}} \text{div}(\cdot) \]
We continue our way backwards in the order shown.

\[ \nabla_{W^{(N-1)}} \text{div}(.) \]
We continue our way backwards in the order shown

\[ \nabla_{\mathbf{y}^{(N-2)}} \text{div}(\cdot) \]
We continue our way backwards in the order shown.

\[ \nabla_{Z^{(N-2)}} \text{div}(.) \]
We continue our way backwards in the order shown

$$\nabla_{Y(1)} \text{div}(.)$$
We continue our way backwards in the order shown.

\[ \nabla_{z^{(1)}} \text{div}(. \ ) \]
We continue our way backwards in the order shown.

\[ y^{(0)} = x \]

\[ f_1 \]

\[ f_{N-2} \]

\[ f_{N-1} \]

\[ f_N \]

\[ z^{(1)} \]

\[ z^{(N-2)} \]

\[ z^{(N-1)} \]

\[ z^{(N)} \]

\[ y^{(1)} \]

\[ y^{(N-2)} \]

\[ y^{(N-1)} \]

\[ y^{(N)} \]

\[ \nabla_{W^{(1)}} \text{div}(\cdot) \]
Backward Gradient Computation

• Lets actually see the math..
Computing derivatives

\[ y^{(0)} = x \]

\[ y^{(1)} \]

\[ z^{(1)} \]

\[ y^{(N-2)} \]

\[ z^{(N-2)} \]

\[ z^{(N-1)} \]

\[ y^{(N-1)} \]

\[ z^{(N)} \]

\[ y^{(N)} \]

\[ \text{Div}(Y,d) \]

\[ \text{div()} \]

\[ d \]
Computing derivatives

The derivative w.r.t the actual output of the final layer of the network is simply the derivative w.r.t to the output of the network

\[
\frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}} = \frac{\partial \text{Div}(Y, d)}{\partial y_i}
\]
Computing derivatives

\[ y^{(0)} = x \]
Computing derivatives

\[ y^{(0)} = x \]

\[ y^{(1)} \quad z^{(1)} \]

\[ y^{(N)} \quad z^{(N)} \]

\[ \text{Div}(Y, d) \]

\[ \frac{\partial \text{Div}}{\partial z_1^{(N)}} = \frac{\partial y_1^{(N)}}{\partial z_1^{(N)}} \frac{\partial \text{Div}}{\partial y_1^{(N)}} \]

Already computed
Computing derivatives

\[ y^{(0)} = x \]

Derivative of activation function

\[ \frac{\partial \text{Div}}{\partial z_1^{(N)}} = \frac{\partial y_1^{(N)}}{\partial z_1^{(N)}} \cdot \frac{\partial \text{Div}}{\partial y_1^{(N)}} \]

Derivative (Div(Y,d))
Computing derivatives

$$y^{(0)} = x$$

Div(Y, d)

Derivative of activation function
Computed in forward pass
Computing derivatives

$\frac{\partial Div}{\partial z_1^{(N)}} = f'_N \left( z_1^{(N)} \right) \frac{\partial Div}{\partial y_1^{(N)}}$

$y^{(0)} = x$

$\begin{array}{c}
\text{Div}(Y,d) \\
\end{array}$
Computing derivatives

\[ y^{(0)} = x \]

\[ y^{(1)} \]

\[ z^{(1)} \]

\[ f_1 \]

\[ f_1 \]

\[ f_1 \]

\[ f_1 \]

\[ 1 \]

\[ 1 \]

\[ f_{N-2} \]

\[ f_{N-2} \]

\[ f_{N-2} \]

\[ f_{N-2} \]

\[ 1 \]

\[ 1 \]

\[ f_{N-1} \]

\[ f_{N-1} \]

\[ f_{N-1} \]

\[ f_{N-1} \]

\[ 1 \]

\[ 1 \]

\[ f_N \]

\[ f_N \]

\[ f_N \]

\[ f_N \]

\[ d \]

\[ \text{Div}(Y,d) \]

\[ \frac{\partial \text{Div}}{\partial z_i^{(N)}} = f_N' \left( z_i^{(N)} \right) \frac{\partial \text{Div}}{\partial y_i^{(N)}} \]
Computing derivatives

\[ y^{(0)} = x \]

\[ y^{(1)} \]

\[ z^{(1)} \]

\[ f_1 \]

\[ f_{N-2} \]

\[ f_{N-1} \]

\[ y^{(N-2)} \]

\[ z^{(N-2)} \]

\[ z^{(N-1)} \]

\[ y^{(N-1)} \]

\[ y^{(N)} \]

\[ z^{(N)} \]

\[ \text{Div}(Y,d) \]

\[ \frac{\partial \text{Div}}{\partial w_{11}^{(N)}} = \frac{\partial z_1^{(N)}}{\partial w_{11}^{(N)}} \frac{\partial \text{Div}}{\partial z_1^{(N)}} \]
Computing derivatives

\[ y^{(0)} = x \]

\[ z^{(1)} \]

\[ y^{(1)} \]

\[ f_1 \]

\[ f_1 \]

\[ f_1 \]

\[ f_1 \]

\[ f_{N-2} \]

\[ f_{N-2} \]

\[ f_{N-2} \]

\[ f_{N-2} \]

\[ f_{N-1} \]

\[ f_{N-1} \]

\[ f_{N-1} \]

\[ f_{N-1} \]

\[ f_{N-1} \]

\[ f_{N-1} \]

\[ f_{N} \]

\[ f_{N} \]

\[ f_{N} \]

\[ y^{(N-2)} \]

\[ z^{(N-2)} \]

\[ z^{(N-1)} \]

\[ y^{(N-1)} \]

\[ z^{(N)} \]

\[ y^{(N)} \]

\[ \text{Div}(Y,d) \]

\[ \nabla \]

\[ d \]

\[ \frac{\partial \text{Div}}{\partial w_{11}^{(N)}} = \frac{\partial z_1^{(N)}}{\partial w_{11}^{(N)}} \frac{\partial \text{Div}}{\partial z_1^{(N)}} \]

Just computed
Computing derivatives

$y^{(0)} = x$

Because

$z^{(N)}_{1} = w^{(N)}_{11} y^{(N-1)}_{1} + \text{other terms}$
Computing derivatives

\[ y^{(0)} = x \]

\[ \frac{\partial \text{Div}}{\partial w_{11}^{(N)}} = \frac{\partial z_1^{(N)}}{\partial w_{11}^{(N)}} \frac{\partial \text{Div}}{\partial z_1^{(N)}} \]

Because
\[ z_1^{(N)} = w_{11}^{(N)} y_1^{(N-1)} + \text{other terms} \]

Computed in forward pass
Computing derivatives

\[ y^{(0)} = x \]

\[ y^{(1)} \]

\[ z^{(1)} \]

\[ y^{(N)} \]

\[ z^{(N)} \]

\[ y^{(N-2)} \]

\[ z^{(N-2)} \]

\[ z^{(N-1)} \]

\[ y^{(N-1)} \]

\[ \frac{\partial \text{Div}}{\partial w_{11}^{(N)}} = y_1^{(N-1)} \frac{\partial \text{Div}}{\partial z_1^{(N)}} \]
Computing derivatives

\[ y^{(0)} = x \]

\[ z^{(1)} \quad y^{(1)} \]

For the bias term \( y_0^{(N-1)} = 1 \)

\[ \frac{\partial \text{Div}}{\partial w_{ij}^{(N)}} = y_i^{(N-1)} \frac{\partial \text{Div}}{\partial z_j^{(N)}} \]
Computing derivatives

\[ y^{(0)} = x \]

\[ \partial \text{Div}_Y(x, d) = \sum_j \frac{\partial z_j^{(N)}}{\partial y_1^{(N-1)}} \frac{\partial \text{Div}_Y(x, d)}{\partial z_j^{(N)}} \]
Computing derivatives

\( y^{(0)} = x \)

\[ \frac{\partial \text{Div}}{\partial y^{(N-1)}} = \sum_j \frac{\partial z_j^{(N)}}{\partial y^{(N-1)}} \frac{\partial \text{Div}}{\partial z_j^{(N)}} \]

Already computed
Computing derivatives

\[ y^{(0)} = x \]

\[ y^{(1)} \]

\[ f_1 \]

\[ z^{(1)} \]

\[ y^{(1)} \]

\[ f_1 \]

\[ f_{N-2} \]

\[ f_{N-1} \]

\[ y^{(N-2)} \]

\[ z^{(N-2)} \]

\[ z^{(N-1)} \]

\[ y^{(N-1)} \]

\[ y^{(N)} \]

\[ z^{(N)} \]

Because

\[ z_j^{(N)} = w_{1j}^{(N)} y_1^{(N-1)} + \text{other terms} \]

\[ \frac{\partial \text{Div}}{\partial y_1^{(N-1)}} = \sum_{j} \frac{\partial z_j^{(N)}}{\partial y_1^{(N-1)}} \frac{\partial \text{Div}}{\partial z_j^{(N)}} \]
Computing derivatives

$y^{(0)} = x$

$z^{(1)}$ $y^{(1)}$

$z^{(N-2)}$ $y^{(N-2)}$ $z^{(N-1)}$

$y^{(N-1)}$

$y^{(N)}$

$z^{(N)}$

$\frac{\partial \text{Div}}{\partial y_1^{(N-1)}} = \sum_j w_{1j}^{(N)} \frac{\partial \text{Div}}{\partial z_j^{(N)}}$

$\text{Div}(Y,d)$

$d$
Computing derivatives

\[ y^{(0)} = x \]

\[ \frac{\partial \text{Div}}{\partial y_{(N-1)^{(N-1)}}} = \sum_j w_{ij}^{(N)} \frac{\partial \text{Div}}{\partial z_j^{(N)}} \]
Computing derivatives

\[ y^{(0)} = x \]

We continue our way backwards in the order shown

\[
\frac{\partial \text{Div}}{\partial z_{(N-1)}^{(N-1)}} = f'_{N-1} \left( z_{i}^{(N-1)} \right) \frac{\partial \text{Div}}{\partial y_{i}^{(N-1)}}
\]
We continue our way backwards in the order shown

\[
\frac{\partial \text{Div}}{\partial w_{ij}^{(N-1)}} = y_i^{(N-2)} \frac{\partial \text{Div}}{\partial z_j^{(N-1)}}
\]

For the bias term \(y_0^{(N-2)} = 1\)
We continue our way backwards in the order shown

$$\frac{\partial \text{Div}}{\partial y_i^{(N-2)}} = \sum_j w_{ij}^{(N-1)} \frac{\partial \text{Div}}{\partial z_j^{(N-1)}}$$
We continue our way backwards in the order shown.

\[
\frac{\partial \text{Div}}{\partial z^{(N-2)}_i} = f'_{N-2} \left( z^{(N-2)}_i \right) \frac{\partial \text{Div}}{\partial y^{(N-2)}_i}
\]
We continue our way backwards in the order shown.

\[
\frac{\partial \text{Div}}{\partial y^{(1)}_1} = \sum_j w_{ij}^{(2)} \frac{\partial \text{Div}}{\partial z^{(2)}_j}
\]
We continue our way backwards in the order shown.

\[
\frac{\partial \text{Div}}{\partial z_i^{(1)}} = f_1' \left( z_i^{(1)} \right) \frac{\partial \text{Div}}{\partial y_i^{(1)}}
\]
We continue our way backwards in the order shown

\[
\frac{\partial \text{Div}}{\partial w_{ij}^{(1)}} = y_i^{(0)} \frac{\partial \text{Div}}{\partial z_j^{(1)}}
\]
Gradients: Backward Computation

$\frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}} = \frac{\partial \text{Div}(Y, d)}{\partial y_i}$

$\frac{\partial \text{Div}}{\partial z_i^{(N)}} = f'_k(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}}$

For $k = N - 1 \ldots 0$

For $i = 1$: layer width

$\frac{\partial \text{Div}}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$

$\frac{\partial \text{Div}}{\partial z_i^{(k)}} = f'_k(z_i^{(k)}) \frac{\partial \text{Div}}{\partial y_i^{(k)}}$

$\forall j \frac{\partial \text{Div}}{\partial w_{ij}^{(k+1)}} = y_i^{(k)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$

Figure assumes, but does not show the “1” bias nodes.
Backward Pass

• Output layer (N):
  – For $i = 1 \ldots D_N$
    - $\frac{\partial Div}{\partial y_i^{(N)}} = \frac{\partial Div(Y,d)}{\partial y_i}$
    - $\frac{\partial Div}{\partial z_i^{(N)}} = \frac{\partial Div}{\partial y_i^{(N)}} f'_N(z_i^{(N)})$

• For layer $k = N - 1$ down to 1
  – For $i = 1 \ldots D_k$
    - $\frac{\partial Div}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial Div}{\partial z_j^{(k+1)}}$
    - $\frac{\partial Div}{\partial z_i^{(k)}} = \frac{\partial Div}{\partial y_i^{(k)}} f'_k(z_i^{(k)})$
    - $\frac{\partial Div}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \frac{\partial D}{\partial z_i^{(k+1)}}$ for $j = 1 \ldots D_k$
    - $\frac{\partial Div}{\partial w_{ji}^{(1)}} = y_j^{(0)} \frac{\partial Div}{\partial z_i^{(1)}}$ for $j = 1 \ldots D_0
Backward Pass

- Output layer (N):
  - For $i = 1 \ldots D_N$
    - \[ \frac{\partial D_{\text{Div}}}{\partial y_i^{(N)}} = \frac{\partial D_{\text{Div}}(y,d)}{\partial y_i} \]
    - \[ \frac{\partial D_{\text{Div}}}{\partial z_i^{(N)}} = \frac{\partial D_{\text{Div}}}{\partial y_i^{(N)}} f'_N \left( z_i^{(N)} \right) \]

- For layer $k = N - 1$ downto 1
  - For $i = 1 \ldots D_k$
    - \[ \frac{\partial D_{\text{Div}}}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial D_{\text{Div}}}{\partial z_j^{(k+1)}} \]
    - \[ \frac{\partial D_{\text{Div}}}{\partial z_i^{(k)}} = \frac{\partial D_{\text{Div}}}{\partial y_i^{(k)}} f'_k \left( z_i^{(k)} \right) \]
    - \[ \frac{\partial D_i}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \frac{\partial D_{\text{Div}}}{\partial z_i^{(k+1)}} \text{ for } j = 1 \ldots D_k \]
    - \[ \frac{\partial D_{\text{Div}}}{\partial w_{ji}^{(1)}} = y_j^{(0)} \frac{\partial D_{\text{Div}}}{\partial z_i^{(1)}} \text{ for } j = 1 \ldots D_0 \]
Using notation $\dot{y} = \frac{\partial Div(Y,d)}{\partial y}$ etc (overdot represents derivative of $Div$ w.r.t variable)

- **Output layer (N):**
  - For $i = 1 \ldots D_N$
    - $\dot{y}_i^{(N)} = \frac{\partial Div}{\partial y_i}$
    - $\dot{z}_i^{(N)} = \dot{y}_i^{(N)} f'_N(z_i^{(N)})$

- **For layer $k = N - 1$ down to 1**
  - For $i = 1 \ldots D_k$
    - $\dot{y}_i^{(k)} = \sum_j w_{ij}^{(k+1)} \dot{z}_j^{(k+1)}$
    - $\dot{z}_i^{(k)} = \dot{y}_i^{(k)} f'_k(z_i^{(k)})$

  - $\frac{\partial Div}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \dot{z}_i^{(k+1)}$ for $j = 1 \ldots D_k$

  - $\frac{\partial Div}{\partial w_{ji}^{(1)}} = y_j^{(0)} \dot{z}_i^{(1)}$ for $j = 1 \ldots D_0$

Called “Backpropagation” because the derivative of the loss is propagated “backwards” through the network.

Very analogous to the forward pass:

- Backward weighted combination of next layer
- Backward equivalent of activation
For comparison: the forward pass again

- Input: \( D \) dimensional vector \( \mathbf{x} = [x_j, \ j = 1 \ldots D] \)
- Set:
  - \( D_0 = D \), is the width of the 0th (input) layer
  - \( y_j^{(0)} = x_j, \ j = 1 \ldots D; \quad y_0^{(k=1\ldots N)} = x_0 = 1 \)
- For layer \( k = 1 \ldots N \)
  - For \( j = 1 \ldots D_k \)
    - \( z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)} \)
    - \( y_j^{(k)} = f_k\left(z_j^{(k)}\right) \)
- Output:
  - \( Y = y_j^{(N)}, j = 1 \ldots D_N \)
Special cases

• Have assumed so far that
  1. The computation of the output of one neuron does not directly affect computation of other neurons in the same (or previous) layers
  2. Inputs to neurons only combine through weighted addition
  3. Activations are actually differentiable
     – All of these conditions are frequently not applicable

• Will not discuss all of these in class, but explained in slides
  – Will appear in quiz. Please read the slides
Special Case 1. Vector activations

• Vector activations: all outputs are functions of all inputs
Special Case 1. Vector activations

Scalar activation: Modifying a $z_i$ only changes corresponding $y_i$

$y_i^{(k)} = f(z_i^{(k)})$

Vector activation: Modifying a $z_i$ potentially changes all, $y_1 \ldots y_M$

$\begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_M^{(k)} \end{bmatrix} = f(\begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_D^{(k)} \end{bmatrix})$
Scalar activation: Each $z_i$ influences one $y_i$

Vector activation: Each $z_i$ influences all, $y_1 \ldots y_M$
The number of outputs

- Note: The number of outputs ($y^{(k)}$) need not be the same as the number of inputs ($z^{(k)}$)
  - May be more or fewer
Scalar Activation: Derivative rule

In the case of scalar activation functions, the derivative of the error w.r.t to the input to the unit is a simple product of derivatives.

\[
\frac{\partial Div}{\partial z_i^{(k)}} = \frac{\partial Div}{\partial y_i^{(k)}} \frac{dy_i^{(k)}}{dz_i^{(k)}}
\]
Derivatives of vector activation

For vector activations the derivative of the error w.r.t. to any input is a sum of partial derivatives:

\[
\frac{\partial \text{Div}}{\partial z_i^{(k)}} = \sum_j \frac{\partial \text{Div}}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}}
\]

Note: derivatives of scalar activations are just a special case of vector activations:

\[
\frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} = 0 \text{ for } i \neq j
\]

- For vector activations the derivative of the error w.r.t. to any input is a sum of partial derivatives:
  - Regardless of the number of outputs \(y_j^{(k)}\)
Example Vector Activation: Softmax

\[ y^{(k)}_i = \frac{\exp(z^{(k)}_i)}{\sum_j \exp(z^{(k)}_j)} \]
Example Vector Activation: Softmax

\[ y_i^{(k)} = \frac{\exp(z_i^{(k)})}{\sum_j \exp(z_j^{(k)})} \]

\[ \frac{\partial Div}{\partial z_i^{(k)}} = \sum_j \frac{\partial Div}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} \]
Example Vector Activation: Softmax

\[ y_i^{(k)} = \frac{\exp(z_i^{(k)})}{\sum_j \exp(z_j^{(k)})} \]

\[ \frac{\partial \text{Div}}{\partial z_i^{(k)}} = \sum_j \frac{\partial \text{Div}}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} \]

\[ \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} = \begin{cases} y_i^{(k)} \left(1 - y_i^{(k)} \right) & \text{if } i = j \\ -y_i^{(k)} y_j^{(k)} & \text{if } i \neq j \end{cases} \]
Example Vector Activation: Softmax

\[ y_i^{(k)} = \frac{\exp(z_i^{(k)})}{\sum_j \exp(z_j^{(k)})} \]

\[ \frac{\partial \text{Div}}{\partial z_i^{(k)}} = \sum_j \frac{\partial \text{Div}}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} \]

\[ \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} = \begin{cases} y_i^{(k)} (1 - y_i^{(k)}) & \text{if } i = j \\ -y_i^{(k)} y_j^{(k)} & \text{if } i \neq j \end{cases} \]

\[ \frac{\partial \text{Div}}{\partial z_i^{(k)}} = \sum_j \frac{\partial \text{Div}}{\partial y_j^{(k)}} y_i^{(k)} (\delta_{ij} - y_j^{(k)}) \]

- For future reference
- \( \delta_{ij} \) is the Kronecker delta: \( \delta_{ij} = 1 \) if \( i = j \), \( 0 \) if \( i \neq j \)
Backward Pass for \textit{softmax output layer}

- **Output layer (N):**
  - For $i = 1 \ldots D_N$
    - $\frac{\partial \text{Div}}{\partial y_i^{(N)}} = \frac{\partial \text{Div}(y,d)}{\partial y_i}$
    - $\frac{\partial \text{Div}}{\partial z_i^{(N)}} = \sum_j \frac{\partial \text{Div}(y,d)}{\partial y_j^{(N)}} y_i^{(N)} \left( \delta_{ij} - y_j^{(N)} \right)$

- **For layer $k = N - 1$ downto 1**
  - For $i = 1 \ldots D_k$
    - $\frac{\partial \text{Div}}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$
    - $\frac{\partial \text{Div}}{\partial z_i^{(k)}} = f'_k \left( z_i^{(k)} \right) \frac{\partial \text{Div}}{\partial y_i^{(k)}}$
    - $\frac{\partial \text{Div}}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \frac{\partial \text{Div}}{\partial z_i^{(k+1)}}$ for $j = 1 \ldots D_k$
    - $\frac{\partial \text{Div}}{\partial w_{ji}^{(1)}} = y_j^{(0)} \frac{\partial \text{Div}}{\partial z_i^{(1)}}$ for $j = 1 \ldots D_0$
Special cases

• Examples of vector activations and other special cases on slides
  – Please look up
  – Will appear in quiz!
• In reality the vector combinations can be anything
  – E.g. linear combinations, polynomials, logistic (softmax), etc.

\[
\begin{bmatrix}
  y_1^{(k)} \\
  y_2^{(k)} \\
  \vdots \\
  y_M^{(k)} \\
\end{bmatrix} = f \left( \begin{bmatrix}
  Z_1^{(k)} \\
  Z_2^{(k)} \\
  \vdots \\
  Z_D^{(k)} \\
\end{bmatrix} \right)
\]
Special Case 2: Multiplicative networks

Some types of networks have *multiplicative* combination
  – In contrast to the *additive* combination we have seen so far

Seen in networks such as LSTMs, GRUs, attention models, etc.

Forward: \( O_i^{(k)} = y_j^{(k-1)} y_l^{(k-1)} \)
Backpropagation: Multiplicative Networks

Forward:

$$o_i^{(k)} = y_j^{(k-1)} y_l^{(k-1)}$$

Backward:

$$\frac{\partial \text{Div}}{\partial o_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$$

$$\frac{\partial \text{Div}}{\partial y_j^{(k-1)}} = \frac{\partial o_i^{(k)}}{\partial y_j^{(k-1)}} \frac{\partial \text{Div}}{\partial o_i^{(k)}} = y_l^{(k-1)} \frac{\partial \text{Div}}{\partial o_i^{(k)}}$$

$$\frac{\partial \text{Div}}{\partial y_l^{(k-1)}} = y_j^{(k-1)} \frac{\partial \text{Div}}{\partial o_i^{(k)}}$$

• Some types of networks have multiplicative combination
Multiplicative combination as a case of vector activations

A layer of multiplicative combination is a special case of vector activation

\[ Z_i^{(k)} = y_i^{(k-1)} \]

\[ y_i^{(k)} = Z_{2i-1}^{(k)} Z_{2i}^{(k)} \]

• A layer of multiplicative combination is a special case of vector activation
Multiplicative combination: Can be viewed as a case of vector activations

- A layer of multiplicative combination is a special case of vector activation

\[
\begin{align*}
  z_i^{(k)} &= \sum_j w_{ji}^{(k)} y_j^{(k-1)} \\
  y_i^{(k)} &= \prod_l (z_i^{(k)})^{\alpha_{li}^{(k)}} \\
  \frac{\partial y_i^{(k)}}{\partial z_j^{(k)}} &= \alpha_{ji}^{(k)} \left( z_j^{(k)} \right)^{\alpha_{ji}^{(k)}} - 1 \prod_{l \neq j} (z_l^{(k)})^{\alpha_{li}^{(k)}} \\
  \frac{\partial \text{Div}}{\partial z_j^{(k)}} &= \sum_i \frac{\partial \text{Div}}{\partial z_j^{(k)}} \frac{\partial y_i^{(k)}}{\partial z_j^{(k)}}
\end{align*}
\]
Gradients: Backward Computation

For $k = N \ldots 1$
For $i = 1: \text{layer width}$

\[
\frac{\partial \text{Div}}{\partial z_{i}^{(k)}} = \sum_{j} \frac{\partial \text{Div}}{\partial y_{j}^{(k)}} \frac{\partial y_{j}^{(k)}}{\partial z_{i}^{(k)}}
\]

If layer has vector activation

\[
\frac{\partial \text{Div}}{\partial y_{i}^{(k-1)}} = \sum_{j} w_{ij}^{(k)} \frac{\partial \text{Div}}{\partial z_{j}^{(k)}}
\]

Else if activation is scalar

\[
\frac{\partial \text{Div}}{\partial w_{ij}^{(k)}} = y_{i}^{(k-1)} \frac{\partial \text{Div}}{\partial z_{j}^{(k)}}
\]
Special Case: Non-differentiable activations

- Activation functions are sometimes not actually differentiable
  - E.g. The RELU (Rectified Linear Unit)
    - And its variants: leaky RELU, randomized leaky RELU
  - E.g. The “max” function
- Must use “subgradients” where available
  - Or “secants”
The subgradient

• A subgradient of a function $f(x)$ at a point $x_0$ is any vector $\nu$ such that
  $$(f(x) - f(x_0)) \geq \nu^T (x - x_0)$$
  - Any direction such that moving in that direction increases the function

• Guaranteed to exist only for convex functions
  - “bowl” shaped functions
  - For non-convex functions, the equivalent concept is a “quasi-secant”

• The subgradient is a direction in which the function is guaranteed to increase
• If the function is differentiable at $x_0$, the subgradient is the gradient
  - The gradient is not always the subgradient though
Subgradients and the RELU

• Can use any subgradient
  – At the differentiable points on the curve, this is the same as the gradient
  – Typically, will use the equation given

$$f'(z) = \begin{cases} 
0, & z < 0 \\
1, & z \geq 0 
\end{cases}$$
Subgradients and the Max

\[ y = \max_j z_j \]

\[ \frac{\partial y}{\partial z_i} = \begin{cases} 
1, & i = \arg\max_j z_j \\
0, & \text{otherwise} 
\end{cases} \]

- Vector equivalent of subgradient
  - 1 w.r.t. the largest incoming input
    - Incremental changes in this input will change the output
  - 0 for the rest
    - Incremental changes to these inputs will not change the output
Subgradients and the Max

- Multiple outputs, each selecting the max of a different subset of inputs
  - Will be seen in convolutional networks
- Gradient for any output:
  - 1 for the specific component that is maximum in corresponding input subset
  - 0 otherwise

\[ y_i = \arg\max_{l \in S_j} z_l \]

\[
\frac{\partial y_j}{\partial z_i} = \begin{cases} 
1, & i = \arg\max_{l \in S_j} z_l \\
0, & \text{otherwise}
\end{cases}
\]
Backward Pass: Recap

• Output layer (N):
  – For \( i = 1 \ldots D_N \):
    
    \[
    \frac{\partial D_i}{\partial y_i^{(N)}} = \frac{\partial D(y,d)}{\partial y_i} \]
    
    \[
    OR \quad \sum_j \frac{\partial D_i}{\partial y_j^{(N)}} \frac{\partial y_j^{(N)}}{\partial y_i^{(N)}} \text{ (vector activation)}
    \]

• For layer \( k = N - 1 \) down to 1
  – For \( i = 1 \ldots D_k \):
    
    \[
    \frac{\partial D_i}{\partial z_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial D_i}{\partial z_j^{(k+1)}} \]
    
    \[
    OR \quad \sum_j \frac{\partial D_i}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}} \text{ (vector activation)}
    \]

  \[
  \frac{\partial D_i}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \frac{\partial D_i}{\partial z_i^{(k+1)}} \text{ for } j = 1 \ldots D_k
  \]

  \[
  - \frac{\partial D_i}{\partial w_{ji}^{(1)}} = y_j^{(0)} \frac{\partial D_i}{\partial z_i^{(1)}} \text{ for } j = 1 \ldots D_0
  \]

These may be subgradients
Overall Approach

• For each data instance
  – **Forward pass**: Pass instance forward through the net. Store all intermediate outputs of all computation.
  – **Backward pass**: Sweep backward through the net, iteratively compute all derivatives w.r.t weights

• Actual loss is the sum of the divergence over all training instances

\[
\text{Loss} = \frac{1}{|\{X\}|} \sum_X \text{Div}(Y(X), d(X))
\]

• Actual gradient is the sum or average of the derivatives computed for each training instance

\[
\nabla_W \text{Loss} = \frac{1}{|\{X\}|} \sum_X \nabla_W \text{Div}(Y(X), d(X)) \quad W \leftarrow W - \eta \nabla_W \text{Loss}^T
\]
Training by BackProp

- Initialize weights $W^{(k)}$ for all layers $k = 1 \ldots K$
- Do: (Gradient descent iterations)
  - Initialize $Loss = 0$; For all $i, j, k$, initialize $\frac{dLoss}{dw_{i,j}^{(k)}} = 0$
  - For all $t = 1: T$ (Iterate over training instances)
    - **Forward pass:** Compute
      - Output $Y_t$
      - $Loss += Div(Y_t, d_t)$
    - **Backward pass:** For all $i, j, k$:
      - Compute $\frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}}$
      - Compute $\frac{dLoss}{dw_{i,j}^{(k)}} += \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}}$
      - For all $i, j, k$, update:
        $$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \frac{\eta}{T} \frac{dLoss}{dw_{i,j}^{(k)}}$$
- Until $Loss$ has converged
Vector formulation

• For layered networks it is generally simpler to think of the process in terms of vector operations
  – Simpler arithmetic
  – Fast matrix libraries make operations much faster

• We can restate the entire process in vector terms
  – This is what is actually used in any real system
Vector formulation

- Arrange all inputs to the network in a vector $\mathbf{x}$
- Arrange the inputs to neurons of the kth layer as a vector $\mathbf{z}_k$
- Arrange the outputs of neurons in the kth layer as a vector $\mathbf{y}_k$
- Arrange the weights to any layer as a matrix $\mathbf{W}_k$
  - Similarly with biases

\[ \mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix}, \quad \mathbf{z}_k = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_D^{(k)} \end{bmatrix}, \quad \mathbf{y}_k = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_D^{(k)} \end{bmatrix} \]
Vector formulation

- The computation of a single layer is easily expressed in matrix notation as (setting $y_0 = x$):

$$z_k = W_k y_{k-1} + b_k \quad \quad y_k = f_k(z_k)$$
The forward pass: Evaluating the network

\[ y_0 = x \]
The forward pass

\[ z_1 = W_1 x + b_1 \]
The forward pass

\[ y_1 = f_1(z_1) \]

The Complete computation

\[ y_1 = f_1(W_1 x + b_1) \]
The forward pass

\[ z_2 = W_2 y_1 + b_2 \]

The Complete computation

\[ y_1 = f_1(W_1 x + b_1) \]
The forward pass

The Complete computation

\[ y_2 = f_2(W_2 f_1(W_1 x + b_1) + b_2) \]
The forward pass

\[ z_N = W_N f_{N-1}(... f_2(W_2 f_1(W_1 x + b_1) + b_2) ... ) + b_N \]
The forward pass

The Complete computation

\[ Y = f_N(W_N f_{N-1}(... f_2(W_2 f_1(W_1 x + b_1) + b_2) ... ) + b_N) \]
Forward pass:

Initialize

For k = 1 to N:

\[ y_k = f_k(z_k) \]

Output

\[ Y = y_N \]
The Forward Pass

• Set $y_0 = x$

• Recursion through layers:
  – For layer $k = 1$ to $N$:
    $$z_k = W_k y_{k-1} + b_k$$
    $$y_k = f_k(z_k)$$

• Output:
  $$Y = y_N$$
The backward pass

- The network is a nested function

\[ Y = f_N (W_N f_{N-1} (\ldots f_2 (W_2 f_1 (W_1 x + b_1) + b_2) \ldots ) + b_N) \]

- The divergence for any \( x \) is also a nested function

\[ \text{Div}(Y, d) = \text{Div}(f_N (W_N f_{N-1} (\ldots f_2 (W_2 f_1 (W_1 x + b_1) + b_2) \ldots ) + b_N), d) \]
Calculus recap 2: The Jacobian

• The derivative of a vector function w.r.t. vector input is called a Jacobian

• It is the matrix of partial derivatives given below

\[
\begin{bmatrix}
  y_1 \\
  y_2 \\
  \vdots \\
  y_M
\end{bmatrix} = f \left( \begin{bmatrix}
  z_1 \\
  z_2 \\
  \vdots \\
  z_D
\end{bmatrix} \right)
\]

Using vector notation

\( \mathbf{y} = f(\mathbf{z}) \)

Check:

\( \Delta \mathbf{y} = J_y(\mathbf{z}) \Delta \mathbf{z} \)
Jacobians can describe the derivatives of neural activations w.r.t their input

For Scalar activations

- Number of outputs is identical to the number of inputs

Jacobian is a diagonal matrix

- Diagonal entries are individual derivatives of outputs w.r.t inputs
- Not showing the superscript “(k)” in equations for brevity

\[
J_y(z) = \begin{bmatrix}
\frac{dy_1}{dz_1} & 0 & \ldots & 0 \\
0 & \frac{dy_2}{dz_2} & \ldots & 0 \\
\ldots & \ldots & \ddots & \ldots \\
0 & 0 & \ldots & \frac{dy_D}{dz_D}
\end{bmatrix}
\]
Jacobians can describe the derivatives of neural activations w.r.t. their input.

For scalar activations (shorthand notation):

- Jacobian is a diagonal matrix
- Diagonal entries are individual derivatives of outputs w.r.t. inputs

\[ y_i = f(z_i) \]

\[
J_y(z) = \begin{bmatrix}
    f'(z_1) & 0 & \cdots & 0 \\
    0 & f'(z_2) & \cdots & 0 \\
    \vdots & \vdots & \ddots & \vdots \\
    0 & 0 & \cdots & f'(z_M)
\end{bmatrix}
\]
For Vector activations

- Jacobian is a full matrix
  - Entries are partial derivatives of individual outputs w.r.t individual inputs

\[
J_y(z) = \begin{bmatrix}
\frac{\partial y_1}{\partial z_1} & \frac{\partial y_1}{\partial z_2} & \cdots & \frac{\partial y_1}{\partial z_D} \\
\frac{\partial y_2}{\partial z_1} & \frac{\partial y_2}{\partial z_2} & \cdots & \frac{\partial y_2}{\partial z_D} \\
\vdots & \vdots & \ddots & \vdots \\
\frac{\partial y_M}{\partial z_1} & \frac{\partial y_M}{\partial z_2} & \cdots & \frac{\partial y_M}{\partial z_D}
\end{bmatrix}
\]
Special case: Affine functions

\[ z = W y + b \]

\[ J_z(y) = W \]

• Matrix \( W \) and bias \( b \) operating on vector \( y \) to produce vector \( z \)
• The Jacobian of \( z \) w.r.t \( y \) is simply the matrix \( W \)
Vector derivatives: Chain rule

- We can define a chain rule for Jacobians
- For vector functions of vector inputs:

\[ y = f(g(x)) \]

\[ z = g(x) \]

\[ y = f(z) \]

\[ J_y(x) = J_y(z)J_z(x) \]

Check

\[ \Delta z = J_z(x)\Delta x \]

\[ \Delta y = J_y(z)\Delta z \]

\[ \Delta y = J_y(z)J_z(x)\Delta x = J_y(x)\Delta x \]

Note the order: The derivative of the outer function comes first
Vector derivatives: Chain rule

- The chain rule can combine Jacobians and Gradients
- For scalar functions of vector inputs ($g()$ is vector):

\[
D = f(g(x))
\]

\[
z = g(x)
\]

\[
D = f(z)
\]

\[
\nabla_x D = \nabla_z (D) J_z(x)
\]

Check

\[
\Delta z = J_z(x) \Delta x
\]

\[
\Delta D = \nabla_z (D) \Delta z
\]

\[
\Delta D = \nabla_z (D) J_z(x) \Delta x = \nabla_x D \Delta x
\]

Note the order: The derivative of the outer function comes first
Special Case

- Scalar functions of Affine functions

\[ D = f(Wy + b) \]
\[ z = Wy + b \]
\[ D = f(z) \]

\[ \nabla_y D = \nabla_z(D)W \]
\[ \nabla_b D = \nabla_z(D) \]
\[ \nabla_W D = y\nabla_z(D) \]

Note reversal of order. This is in fact a simplification of a product of tensor terms that occur in the right order.
The backward pass

In the following slides we will also be using the notation $\nabla_z Y$ to represent the Jacobian $J_Y(z)$ to explicitly illustrate the chain rule.

In general $\nabla_a b$ represents a derivative of $b$ w.r.t. $a$ and could be a the transposed gradient (for scalar $b$) or a Jacobian (for vector $b$).
First compute the derivative of the divergence w.r.t. $Y$. The actual derivative depends on the divergence function.

N.B: The gradient is the transpose of the derivative.
The backward pass

\[ \nabla_{z_N} Div = \nabla_Y Div \cdot \nabla_{z_N} Y \]

Already computed  New term
The backward pass

\[ \nabla_{z_N} \text{Div} = \nabla_Y \text{Div} \cdot J_Y(z_N) \]

Already computed \hspace{1cm} \text{New term}
The backward pass

\[ \nabla_{y_{N-1}} Div = \nabla_{z_N} Div \cdot \nabla_{y_{N-1}} z_N \]

Already computed  New term
The backward pass

\[ \nabla_{y_{N-1}} Div = \nabla_{z_N} Div W_N \]

Already computed  New term

220
The backward pass

\[ \nabla_{y_{N-1}} \text{Div} = \nabla_{z_N} \text{Div} W_N \]

\[ \nabla_{w_N} \text{Div} = y_{N-1} \nabla_{z_N} \text{Div} \]

\[ \nabla_{b_N} \text{Div} = \nabla_{z_N} \text{Div} \]
The backward pass

\[ \nabla_{z_{N-1}} \text{Div} = \nabla_{y_{N-1}} \text{Div} \cdot \nabla_{z_{N-1}} y_{N-1} \]

Already computed  New term
The Jacobian will be a diagonal matrix for scalar activations

\[ \nabla_{z_{N-1}} \text{Div} = \nabla_{y_{N-1}} \text{Div} \, J_{y_{N-1}}(z_{N-1}) \]

The Jacobian will be a diagonal matrix for scalar activations
The backward pass

\[ \nabla_{y_{N-2}} Div = \nabla_{z_{N-1}} Div \cdot \nabla_{y_{N-2}} z_{N-1} \]
The backward pass

\[ \nabla_{y_{N-2}} \text{Div} = \nabla_{z_{N-1}} \text{Div} \mathbf{W}_{N-1} \]
The backward pass

\[ \nabla_{y_{N-2}} \text{Div} = \nabla_{z_{N-1}} \text{Div} \cdot W_{N-1} \]

\[ \nabla_{w_{N-1}} \text{Div} = y_{N-2} \nabla_{z_{N-1}} \text{Div} \]

\[ \nabla_{b_{N-1}} \text{Div} = \nabla_{z_{N-1}} \text{Div} \]
The backward pass

\[ \nabla_{z_1} \text{Div} = \nabla_{y_1} \text{Div} J_{y_1}(z_1) \]
The backward pass

In some problems we will also want to compute the derivative w.r.t. the input.

\[ \nabla_{W_1} \text{Div} = x \nabla_{z_1} \text{Div} \]
\[ \nabla_{b_1} \text{Div} = \nabla_{z_1} \text{Div} \]
The Backward Pass

• Set $y_N = Y$, $y_0 = x$

• Initialize: Compute $\nabla_{y_N} Div = \nabla_Y Div$

• For layer $k = N$ downto 1:
  – Compute $J_{y_k}(z_k)$
    • Will require intermediate values computed in the forward pass
  – Backward recursion step:
    \[
    \nabla_{z_k} Div = \nabla_{y_k} Div J_{y_k}(z_k)
    \]
    \[
    \nabla_{y_{k-1}} Div = \nabla_{z_k} Div W_k
    \]
  – Gradient computation:
    \[
    \nabla_{w_k} Div = y_{k-1} \nabla_{z_k} Div
    \]
    \[
    \nabla_{b_k} Div = \nabla_{z_k} Div
    \]
The Backward Pass

• Set $y_N = Y$, $y_0 = x$
• Initialize: Compute $\nabla_{y_N} \text{Div} = \nabla_Y \text{Div}$

• For layer $k = N$ downto 1:
  – Compute $J_{y_k}(z_k)$
    • Will require intermediate values computed in the forward pass
  – Backward recursion step: [Note analogy to forward pass]
    \[
    \nabla_{z_k} \text{Div} = \nabla_{y_k} \text{Div} \cdot J_{y_k}(z_k) \\
    \nabla_{y_{k-1}} \text{Div} = \nabla_{z_k} \text{Div} \cdot W_k
    \]
  – Gradient computation:
    \[
    \nabla_{W_k} \text{Div} = y_{k-1} \nabla_{z_k} \text{Div} \\
    \nabla_{b_k} \text{Div} = \nabla_{z_k} \text{Div}
    \]
For comparison: The Forward Pass

• Set $y_0 = x$

• For layer $k = 1$ to $N$:
  – Forward recursion step:
    \[ z_k = W_k y_{k-1} + b_k \]
    \[ y_k = f_k(z_k) \]

• Output:
  \[ Y = y_N \]
Neural network training algorithm

- Initialize all weights and biases \((W_1, b_1, W_2, b_2, \ldots, W_N, b_N)\)
- Do:
  - \(Loss = 0\)
  - For all \(k\), initialize \(\nabla_{W_k} Loss = 0, \nabla_{b_k} Loss = 0\)
  - For all \(t = 1:T\)  # Loop through training instances
    - Forward pass: Compute
      - Output \(Y(X_t)\)
      - Divergence \(\text{Div}(Y_t, d_t)\)
      - \(Loss += \text{Div}(Y_t, d_t)\)
    - Backward pass: For all \(k\) compute:
      - \(\nabla_{y_k} \text{Div} = \nabla_{z_{k+1}} \text{Div} W_{k+1}\)
      - \(\nabla_{z_k} \text{Div} = \nabla_{y_k} \text{Div} J_y(z_k)\)
      - \(\nabla_{W_k} \text{Div}(Y_t, d_t) = y_{k-1} \nabla_{z_k} \text{Div}; \ nabla_{b_k} \text{Div}(Y_t, d_t) = \nabla_{z_k} \text{Div}\)
      - \(\nabla_{W_k} Loss += \nabla_{W_k} \text{Div}(Y_t, d_t); \ nabla_{b_k} Loss += \nabla_{b_k} \text{Div}(Y_t, d_t)\)
    - For all \(k\), update:
      \[
      W_k = W_k - \frac{\eta}{T} \left(\nabla_{W_k} \text{Loss}\right)^T; \quad b_k = b_k - \frac{\eta}{T} \left(\nabla_{W_k} \text{Loss}\right)^T
      \]
  - Until \(Loss\) has converged
Setting up for digit recognition

- Simple Problem: Recognizing “2” or “not 2”
- Single output with sigmoid activation
  - $Y \in (0,1)$
  - $d$ is either 0 or 1
- Use KL divergence
- Backpropagation to learn network parameters

Training data

- $(5, 0)$, $(2, 1)$
- $(2, 1)$, $(4, 0)$
- $(0, 0)$, $(2, 1)$
- $(3, 0)$, $(1, 1)$
- $(1, 0)$, $(3, 1)$

Sigmoid output neuron
Recognizing the digit

• More complex problem: Recognizing digit
• Network with 10 (or 11) outputs
  – First ten outputs correspond to the ten digits
    • Optional 11th is for none of the above
• Softmax output layer:
  – Ideal output: One of the outputs goes to 1, the others go to 0
• Backpropagation with KL divergence to learn network
Story so far

• Neural networks must be trained to minimize the average divergence between the output of the network and the desired output over a set of training instances, with respect to network parameters.

• Minimization is performed using gradient descent

• Gradients (derivatives) of the divergence (for any individual instance) w.r.t. network parameters can be computed using backpropagation
  – Which requires a “forward” pass of inference followed by a “backward” pass of gradient computation

• The computed gradients can be incorporated into gradient descent
Issues

• Convergence: How well does it learn
  – And how can we improve it
• How well will it generalize (outside training data)
• What does the output really mean?
• *Etc.*
Next up

• Convergence and generalization