Neural Networks
Learning the network: Backprop

11-785, Spring 2018
Lecture 4
Recap: The MLP *can* represent any function

- The MLP *can be constructed* to represent anything
- But *how* do we construct it?
Recap: How to learn the function

- By minimizing expected error

\[ \mathbf{\bar{W}} = \text{argmin}_{\mathbf{W}} \int_{\mathbf{X}} \text{div}(f(X;\mathbf{W}), g(X))P(X)dX \]

\[ = \text{argmin}_{\mathbf{W}} E[\text{div}(f(X;\mathbf{W}), g(X))] \]
Recap: Sampling the function

- \( g(X) \) is unknown, so sample it
  - Basically, get input-output pairs for a number of samples of input \( X_i \)
    - Many samples \((X_i, d_i)\), where \( d_i = g(X_i) + \text{noise} \)
      - Good sampling: the samples of \( X \) will be drawn from \( P(X) \)
  - Estimate function from the samples
The *Empirical* risk

- The *expected* error is the average error over the entire input space

\[
E[\text{div}(f(X; W), g(X))] = \int_X \text{div}(f(X; W), g(X))P(X)dX
\]

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

\[
E[\text{div}(f(X; W), g(X))] \approx \frac{1}{T} \sum_{i=1}^{T} \text{div}(f(X_i; W), d_i)
\]
Empirical Risk Minimization

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

- Error on the \(i\)-th instance: \(\text{div}(f(X_i; W), d_i)\)
- Empirical average error on all training data:

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

Estimate the parameters to minimize the empirical estimate of expected error

\[\bar{W} = \arg\min_W Err(W)\]

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

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

• Minimize the following function

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

w.r.t $W$

• This is problem of function minimization
  – An instance of optimization
• A CRASH COURSE ON FUNCTION OPTIMIZATION
Finding the minimum of a scalar function of a multi-variate 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 gradient equation equals to zero

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

2. Compute the Hessian Matrix $\nabla^2 f(X)$ 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 $\nabla f(X) = 0$
  - 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
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 convergence criteria

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

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

- Or

\[ \left\| \nabla f(x^k) \right\| < 2 \]
Overall Gradient Descent Algorithm

- Initialize:
  - $x^0$
  - $k = 0$

- While $|f(x^{k+1}) - f(x^k)| > \varepsilon$
  - $x^{k+1} = x^k - \eta^k \nabla f(x^k)^T$
  - $k = k + 1$
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

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

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?

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

w.r.t \(W\)

- This is problem of function minimization
  - An instance of optimization
Problem Setup: Things to define

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

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

w.r.t \(W\)

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

What are these input-output pairs?

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

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

• What are these input-output pairs?

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

• What is the divergence $div()$?

• 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

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

w.r.t $W$

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

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
- Generic terminology
  - We will refer to the inputs as the *input units*
    - No neurons here – the “input units” are just the inputs
  - We refer to the outputs as the output units
  - Intermediate units are “hidden” units
Individual neurons operate on a set of inputs and produce a single output.

- **Standard setup:** A differentiable activation function applied the sum of weighted inputs and a bias.
  \[
  y = f \left( \sum_i w_i x_i + b \right)
  \]

- More generally: *any* differentiable function
  \[
  y = f(x_1, x_2, ..., x_N; W)
  \]
The individual neurons

- Individual neurons operate on a set of inputs and produce a single output
  - **Standard setup:** A differentiable activation function applied the sum of weighted inputs and a bias
    \[ y = f \left( \sum_i w_i x_i + b \right) \]
  - More generally: *any* differentiable function
    \[ y = f (x_1, x_2, ..., x_N; W) \]

We will assume this unless otherwise specified

Parameters are weights \( w_i \) and bias \( b \)
Activations and their derivatives

- Some popular activation functions and their derivatives

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

\[ f(z) = \tanh(z) \]
\[ f'(z) = (1 - f^2(z)) \]

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

\[ f(z) = \log(1 + \exp(z)) \]
\[ f'(z) = \frac{1}{1 + \exp(-z)} \]
Vector Activations

• We can also have neurons that have \textit{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

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

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

\[ y = \frac{\exp(z_i)}{\sum_j \exp(z_j)} \]
Multiplicative combination: Can be viewed as a case of vector activations

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

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

\[ y_i = \prod_l (z_l)^{\alpha_{li}} \]

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

- We assume a “layered” network for simplicity
  - We will refer to the inputs as the *input layer*
    - No neurons here – the “layer” simply refers to inputs
  - We refer to the outputs as the output layer
  - Intermediate layers are “hidden” layers
In a layered network, each layer of perceptrons can be viewed as a single vector activation.
Notation

• 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 $b_j^{(k)}$
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?

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

w.r.t $W$

• This is problem of function minimization
  – An instance of optimization
• Given a training set of input-output pairs \((X_1, d_1), (X_2, d_2), \ldots, (X_T, d_T)\)
• \(X_n = [x_{n1}, x_{n2}, \ldots, x_{nD}]\) is the nth input vector
• \(d_n = [d_{n1}, d_{n2}, \ldots, d_{nL}]\) is the nth desired output
• \(Y_n = [y_{n1}, y_{n2}, \ldots, y_{nL}]\) is the nth vector of actual outputs of the network
• 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)
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.
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 an feature value $X$ may occur for both classes, but with different probabilities
    - Is differentiable
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:
  \[
  \begin{bmatrix}
  \text{cat} & \text{dog} & \text{camel} & \text{hat} & \text{flower}
  \end{bmatrix}^T
  \]
• 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 outputs
  - 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

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

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

- This can be viewed as the probability \( y_i = P(class = i|X) \)
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? Is this a digit in the first place?
Typical Problem statement: binary classification

- Given, many positive and negative examples (training data),
  - learn all weights such that the network does the desired job
Typical Problem statement: multiclass classification

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

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

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

- This is a problem of function minimization
  - An instance of optimization
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 cross entropy 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

$$\frac{dDiv(Y, d)}{dY} = \begin{cases} 
\frac{1}{Y} & \text{if } d = 1 \\
\frac{1}{1 - Y} & \text{if } d = 0 
\end{cases}$$
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 cross-entropy between the desired one-hot output and actual output:

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

- Derivative

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

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

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

• The error on the \(i^{th}\) instance is \(\text{div}(Y_i, d_i)\)

• The total error
  \[
  \text{Err} = \frac{1}{T} \sum_i \text{div}(Y_i, d_i)
  \]

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

- In order to minimize any function $f(x)$ w.r.t. $x$

- Initialize:
  - $x^0$
  - $k = 0$

- While $|f(x^{k+1}) - f(x^k)| > \varepsilon$
  - $x^{k+1} = x^k - \eta^k \nabla f(x^k)^T$
  - $k = k + 1$
Recap: Gradient Descent Algorithm

• In order to minimize any function $f(x)$ w.r.t. $x$

• Initialize:
  - $x^0$
  - $k = 0$

• While $|f(x^{k+1}) - f(x^k)| > \varepsilon$
  - For every component $i$
    • $x_i^{k+1} = x_i^k - \eta^k \frac{df}{dx_i}$
  - $k = k + 1$

Explicitly stating it by component
Training Neural Nets through Gradient Descent

Total training error:

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

- Gradient descent algorithm:
- Initialize all weights and biases \( \{w_{ij}\} \)
  - Using the extended notation: the bias is also a weight
- Do:
  - For every layer \( k \) for all \( i, j \), update:
    - \( w_{i,j}^{(k)} = w_{i,j}^{(k)} - \eta \frac{dErr}{dw_{i,j}^{(k)}} \)
- Until \( Err \) has converged
Training Neural Nets through Gradient Descent

Total training error:

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

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

Total training error:

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

- Computing the derivative

Total derivative:

\[ \frac{dErr}{dw_{i,j}^{(k)}} = \frac{1}{T} \sum_t \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}} \]
Training by gradient descent

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

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

Total training error:

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

Total derivative:

\[
\frac{dErr}{dw_{i,j}^{(k)}} = \frac{1}{T} \sum_t \frac{d\text{Div}(Y_t, d_t)}{dw_{i,j}^{(k)}}
\]

- 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 \( \Delta x_1, \Delta x_2, ..., \Delta x_M \)
\[ \Delta y \approx \frac{\partial y}{\partial x_1} \Delta x_1 + \frac{\partial y}{\partial x_2} \Delta x_2 + \cdots + \frac{\partial y}{\partial x_M} \Delta x_M \]
Calculus Refresher: Chain rule

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

\[
\frac{dy}{dx} = \frac{\partial y}{\partial g(x)} \frac{dg(x)}{dx}
\]

Check - we can confirm that:

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

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

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

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

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

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

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

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

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

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

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

• How to compute \( \frac{d\text{Div}(Y,d)}{d w^{(k)}_{i,j}} \)
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 and activations shown
- The overall function is differentiable w.r.t every weight, bias and input
Computing the derivative for a single input

• Aim: compute derivative of $Div(Y, d)$ w.r.t. each of the weights

• But first, let's label all our variables and activation functions
Computing the derivative for a single input

\[\n\begin{align*}
W_1,1 & : w_1^{(1)} \\
W_1,2 & : w_1^{(2)} \\
W_2,1 & : w_2^{(1)} \\
W_2,2 & : w_2^{(2)} \\
W_3,1 & : w_3^{(1)} \\
W_3,2 & : w_3^{(2)}
\end{align*}\n
\[\begin{align*}
Z_1^{(1)} & : z_1^{(1)} \\
Z_1^{(2)} & : z_1^{(2)} \\
Z_1^{(3)} & : z_1^{(3)} \\
y_1^{(1)} & : y_1^{(1)} \\
y_1^{(2)} & : y_1^{(2)} \\
y_1^{(3)} & : y_1^{(3)} \\
y_2^{(1)} & : y_2^{(1)} \\
y_2^{(2)} & : y_2^{(2)} \\
y_2^{(3)} & : y_2^{(3)}
\end{align*}\n
\[\begin{align*}
f_1(.) & : f_1(.) \\
f_2(.) & : f_2(.) \\
f_3(.) & : f_3(.)
\end{align*}\n
\[\begin{align*}
W_1,1 & : w_1^{(3)} \\
W_1,2 & : w_1^{(4)} \\
W_2,1 & : w_2^{(3)} \\
W_2,2 & : w_2^{(4)} \\
W_3,1 & : w_3^{(3)} \\
W_3,2 & : w_3^{(4)}
\end{align*}\n
\[\begin{align*}
Z_1^{(3)} & : z_1^{(3)} \\
Z_1^{(4)} & : z_1^{(4)}
\end{align*}\n
\[\begin{align*}
Y & : y
\end{align*}\n
\[\begin{align*}
d & : d
\end{align*}\n
\[\begin{align*}
Div & : \text{Div}
\end{align*}\n
100
Computing the gradient

• What is: \( \frac{dDiv(Y,d)}{dw_{i,j}^{(k)}} \)

– Derive on board?
Computing the gradient

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

• Derive on board?

• Note: computation of the derivative requires intermediate and final output values of the network in response to the input
• The network again
Gradients: Local Computation

- Redrawn
- Separately label input and output of each node
Forward Computation

\[ Z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i \]

Assuming \( w_{0j}^{(1)} = b_j^{(1)} \) and \( x_0 = 1 \)
Forward Computation

\[ Z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i \]

\[ Z_j^{(k)} = \sum_i w_{ij}^{(k)} y_j^{(k-1)} \]

Assuming \( w_{0j}^{(k)} = b_j^{(k)} \) and \( y_0^{(k-1)} = 1 \)
Forward Computation

\[
Z_j^{(k)} = \sum_i w_{ij}^{(k)} y_j^{(k-1)}
\]

\[
y_j^{(k)} = f_k \left( Z_j^{(k)} \right)
\]

\[
Z_j^{(1)} = \sum_i w_{ij}^{(1)} x_i
\]
Forward Computation

\[
\begin{align*}
\mathbf{y}_{(0)} &= \mathbf{x} \\
\mathbf{z}_{(k-1)} &= \mathbf{y}_{(k-1)} \\
\mathbf{z}_{(k)} &= \mathbf{f}_k \\
\mathbf{y}_{(k)} &= \mathbf{z}_{(k-1)} \\
\mathbf{z}_{(N-1)} &= \mathbf{f}_{N-1} \\
\mathbf{y}_{(N-1)} &= \mathbf{z}_{(N-1)} \\
\mathbf{y}_{(N)} &= \mathbf{f}_N \\
\mathbf{z}_{(N)} &= \mathbf{y}_{(N)} \\
\mathbf{y}_{(k)} &= f_k \left( \mathbf{z}_{(k)} \right)
\end{align*}
\]

ITERATE FOR \( k = 1:N \)

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

\[
\mathbf{z}_{j}^{(k)} = \sum_{i} w_{ij}^{(k)} \mathbf{y}_{i}^{(k-1)}
\]

\[
\mathbf{y}_{j}^{(k)} = f_k \left( \mathbf{z}_{j}^{(k)} \right)
\]
Forward “Pass”

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

• Set:
  
  - $D_0 = D$, is the width of the $0^{th}$ (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 $j = 1 \ldots D_k$
    
    $D_k$ is the size of the $k$th layer
    
    - $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)}$
    
    - $y_j^{(k)} = f_k \left(z_j^{(k)}\right)$

• Output:

  - $Y = y_j^{(N)}, j = 1 \ldots D_N$
Gradients: Backward Computation

\[
\text{Div}(Y, d)
\]
Gradients: Backward Computation

\[
\frac{\partial \text{Div}(Y, d)}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}}
\]
Gradients: Backward Computation

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

\[ \frac{\partial \text{Div}}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial \text{Div}}{\partial y_i} = f_N'(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}} \]
Gradients: Backward Computation

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

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

\( z_i^{(N)} \) computed during the forward pass

\( y^{(k)} \)

\( f_k \)

\( z^{(k)} \)

\( y^{(k-1)} \)

\( z^{(k-1)} \)
Gradients: Backward Computation

Derivative of the activation function of Nth layer

\[
\frac{\partial \text{Div}(Y, d)}{\partial z_i^{(N)}} = \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}} \frac{\partial \text{Div}}{\partial Y_i} = f_N' \left( z_i^{(N)} \right) \frac{\partial \text{Div}}{\partial y_i^{(N)}}
\]
Gradients: Backward Computation

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

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

\[
\frac{\partial \text{Div}}{\partial z_i^{(N)}} = f'_N(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}}
\]

Because:

\[
\frac{\partial z_i^{(N)}}{\partial y_i^{(N-1)}} = w_{ij}^{(N)}
\]
Gradients: Backward Computation

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

\[ \frac{\partial \text{Div}}{\partial z_i^{(N)}} = f'_N \left( z_i^{(N)} \right) \frac{\partial \text{Div}}{\partial y_i^{(N)}} \]

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

computed during the forward pass
Gradients: Backward Computation

\[
\frac{\partial \text{Div} \left( Y, d \right)}{\partial y_{i}^{(N)}} = f_{N}' \left( z_{i}^{(N)} \right) \frac{\partial \text{Div} \left( Y, d \right)}{\partial y_{i}^{(N)}}
\]

\[
\frac{\partial \text{Div} \left( Y, d \right)}{\partial y_{i}^{(k-1)}} = \sum_{j} w_{ij}^{(k)} \frac{\partial \text{Div} \left( Y, d \right)}{\partial z_{j}^{(k)}} = \sum_{j} w_{ij}^{(k)} \frac{\partial \text{Div} \left( Y, d \right)}{\partial z_{j}^{(k)}}
\]
Gradients: Backward Computation

\[
\begin{align*}
\frac{\partial \text{Div}}{\partial Y_i} &= \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}} \\
\frac{\partial \text{Div}}{\partial z_i^{(N)}} &= f_N'(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y_i^{(N)}} \\
\frac{\partial \text{Div}}{\partial y_i^{(k-1)}} &= \sum_j w_{ij}^{(k)} \frac{\partial \text{Div}}{\partial z_j^{(k)}}
\end{align*}
\]

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

\[
\begin{align*}
\text{Div}(Y,d)
\end{align*}
\]
Gradients: Backward Computation

Initialize: Gradient w.r.t. network output

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

For \( k = N \ldots 1 \)

For \( i = 1: \text{layer} - \text{width} \)

\[ \frac{\partial \text{Div}}{\partial z_i^{(k)}} = f_k'(z_i^{(k)}) \frac{\partial \text{Div}}{\partial y_i^{(k)}} \]

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

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

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

• Output layer (N) :
  – For $i = 1 \ldots D_N$
    \[
    \frac{\partial \text{Div}}{\partial y_i} = \frac{\partial \text{Div}(Y,d)}{\partial y^{(N)}_i}
    \]
    \[
    \frac{\partial \text{Div}}{\partial z_i^{(k)}} = \frac{\partial \text{Div}}{\partial y^{(k)}_i} \frac{\partial y^{(k)}_i}{\partial z^{(k)}_i}
    \]

• For layer $k = N - 1 \text{ down to } 0$
  – For $i = 1 \ldots D_k$
    \[
    \frac{\partial \text{Div}}{\partial y^{(k)}_i} = \sum_j w^{(k+1)}_{ij} \frac{\partial \text{Div}}{\partial z^{(k+1)}_j}
    \]
    \[
    \frac{\partial \text{Div}}{\partial z^{(k)}_i} = \frac{\partial \text{Div}}{\partial y^{(k)}_i} \frac{\partial y^{(k)}_i}{\partial z^{(k)}_i}
    \]
    \[
    \frac{\partial \text{Div}}{\partial w^{(k+1)}_{ji}} = y^{(k)}_j \frac{\partial \text{Div}}{\partial z^{(k+1)}_i} \quad \text{for } j = 1 \ldots D_{k-1}
    \]
Backward Pass

• Output layer (N):
  – For $i = 1 \ldots D_N$
    
    \[
    \frac{\partial \text{Div}}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y_i^{(N)}}
    \]

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

• For layer $k = N - 1 \text{ down to } 0$
  – For $i = 1 \ldots D_k$

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

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

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

Called “Backpropagation” because the derivative of the error 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 $0^{th}$ (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 $j = 1 \ldots D_k$
    • $z_j^{(k)} = \sum_{i=0}^{N_k} w_{i,j}^{(k)} y_i^{(k-1)}$
    • $y_j^{(k)} = f_k \left( z_j^{(k)} \right)$

• 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. Outputs of neurons only combine through weighted addition
  3. Activations are actually differentiable
     - All of these conditions are frequently not applicable
- Not discussed 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, $y_1 \ldots y_M$

$$\begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_M^{(k)} \end{bmatrix} = f\begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_D^{(k)} \end{bmatrix}$$
“Influence” diagram

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.
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 \( y_j^{(k)} \)
Training by BackProp

- Initialize all weights \( (W^{(1)}, W^{(2)}, \ldots, W^{(K)}) \)
- Do:
  - Initialize \( Err = 0 \); For all \( i, j, k \), initialize \( \frac{dErr}{dw_{i,j}^{(k)}} = 0 \)
  - For all \( t = 1 : T \) (Loop over training instances)
    - **Forward pass:** Compute
      - Output \( Y_t \)
      - \( Err += Div(Y_t, d_t) \)
    - **Backward pass:** For all \( i, j, k \):
      - Compute \( \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}} \)
      - Compute \( \frac{dErr}{dw_{i,j}^{(k)}} += \frac{dDiv(Y_t, d_t)}{dw_{i,j}^{(k)}} \)
  - For all \( i, j, k \), update:
    \[
    w_{i,j}^{(k)} = w_{i,j}^{(k)} - \eta \frac{dErr}{T dw_{i,j}^{(k)}}
    \]
- Until \( Err \) 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
  – On slides, please read
  – This is what is actually used in any real system
  – Will appear in quiz
Vector formulation

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

\[
\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_D \end{bmatrix} \quad \mathbf{z}_k = \begin{bmatrix} z_1^{(k)} \\ z_2^{(k)} \\ \vdots \\ z_{D_k}^{(k)} \end{bmatrix} \quad \mathbf{y}_k = \begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_{D_k}^{(k)} \end{bmatrix} \\
\mathbf{W}_k = \begin{bmatrix} w_{11}^{(k)} & w_{21}^{(k)} & \cdots & w_{D_k1}^{(k)} \\ w_{12}^{(k)} & w_{22}^{(k)} & \cdots & w_{D_k2}^{(k)} \\ \vdots & \vdots & \ddots & \vdots \\ w_{1D_k1}^{(k)} & w_{1D_k2}^{(k)} & \cdots & w_{1D_kD_k}^{(k)} \end{bmatrix} \quad \mathbf{b}_k = \begin{bmatrix} b_1^{(k)} \\ b_2^{(k)} \\ \vdots \\ b_{D_k+1}^{(k)} \end{bmatrix}
\]
Vector formulation

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

$$
z_k = W_k y_{k-1} + b_k \\
y_k = f_k(z_k)
$$

$$
W_k = \begin{bmatrix}
  w_{11}^{(k)} & w_{12}^{(k)} & \cdots & w_{1D_k}^{(k)} \\
  w_{21}^{(k)} & w_{22}^{(k)} & \cdots & w_{2D_k}^{(k)} \\
  \vdots & \vdots & \ddots & \vdots \\
  w_{D1}^{(k)} & w_{D2}^{(k)} & \cdots & w_{DD_k}^{(k)}
\end{bmatrix} \\
b_k = \begin{bmatrix}
  b_1^{(k)} \\
  b_2^{(k)} \\
  \vdots \\
  b_{Dk+1}^{(k)}
\end{bmatrix}
$$
The forward pass: Evaluating the network

$y_0 = x$
The forward pass

\[ z_1 = W_1x + b_1 \]
The forward pass

The Complete computation

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

\[
\begin{align*}
\vec{z}_2 &= \mathbf{W}_2 \vec{y}_1 + \vec{b}_2 \\
y_1 &= f_1(\mathbf{W}_1 \vec{x} + \vec{b}_1)
\end{align*}
\]
The forward pass

\[ y_2 = f_2(W_2f_1(W_1x + b_1) + b_2) \]

The Complete computation
The forward pass

\[ y_2 = f_2(W_2f_1(W_1x + b_1) + b_2) \]
The forward pass

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

Forward pass:

1. **Initialize**
   
   \[ y_0 = x \]

2. **For k = 1 to N:**
   
   \[ z_k = W_k y_{k-1} + b_k \]
   
   \[ y_k = f_k(z_k) \]

3. **Output**
   
   \[ Y = y_N \]
The Forward Pass

• Set $y_0 = x$

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

• Output:
  \[ Y = y_N \]
The backward pass

- The network is a nested function

\[ Y = f_N(\mathbf{W}_N f_{N-1}(\ldots f_2(\mathbf{W}_2 f_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) \ldots) + \mathbf{b}_N) \]

- The error for any \( \mathbf{x} \) is also a nested function

\[ \text{Div}(Y, d) = \text{Div}(f_N(\mathbf{W}_N f_{N-1}(\ldots f_2(\mathbf{W}_2 f_1(\mathbf{W}_1 \mathbf{x} + \mathbf{b}_1) + \mathbf{b}_2) \ldots) + \mathbf{b}_N), d) \]
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

\[\textbf{y} = f(\textbf{z})\]

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

Check:

\[\Delta \textbf{y} = J_y(\textbf{z})\Delta \textbf{z}\]
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

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

\[ J_y(z) = \begin{bmatrix} f'(y_1) & 0 & \cdots & 0 \\ 0 & f'(y_2) & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & f'(y_M) \end{bmatrix} \]

- For scalar activations (shorthand notation):
  - Jacobian is a diagonal matrix
  - Diagonal entries are individual derivatives of outputs w.r.t inputs
For *Vector* activations

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

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

\[ z = Wy + b \]

\[ J_z(y) = W \]

- 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 gradient (for scalar $b$) or a Jacobian (for vector $b$).
First compute the gradient of the divergence w.r.t. $Y$. The actual gradient depends on the divergence function.
The backward pass

\[ \nabla_z Div = \nabla_Y Div \cdot \nabla_z Y \]
The backward pass

\[ \nabla_{z_N} \text{Div} = \nabla_Y \text{Div} J_Y(z_N) \]
The backward pass

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

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

\[ \nabla_{y_{N-1}} \text{Div} \]
The backward pass

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

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

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

\[
\begin{align*}
\nabla_{z_{N-1}} \text{Div} &= \nabla_{y_{N-1}} \text{Div} \cdot \nabla_{z_{N-1}} y_{N-1} \\
\nabla_{z_{N-1}} \text{Div} &= \nabla_{y_{N-1}} \text{Div} \cdot \nabla_{z_{N-1}} y_{N-1}
\end{align*}
\]
The backward pass

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

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

\[ W_1, b_1 \]

\[ \vdots \]

\[ \vdots \]

\[ \vdots \]

\[ W_{N-1}, b_{N-1} \]

\[ W_N, b_N \]

\[ d \]

\[ \text{Div} \]

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

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

\[ \nabla y_{N-2} Div = \nabla z_{N-1} Div W_{N-1} \]

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

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

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

\[ \nabla_{W_1} Div = x \nabla_{z_1} Div \]
\[ \nabla_{b_1} Div = \nabla_{z_1} Div \]

In some problems we will also want to compute the derivative w.r.t. the input
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
  – Recursion:
    \[
    \nabla_{z_k} \text{Div} = \nabla_{y_k} \text{Div} J_{y_k}(z_k)
    \]
    \[
    \nabla_{y_{k-1}} \text{Div} = \nabla_{z_k} \text{Div} W_k
    \]
  – Gradient computation:
    \[
    \nabla_{W_k} \text{Div} = y_{k-1} \nabla_{z_k} \text{Div}
    \]
    \[
    \nabla_{b_k} \text{Div} = \nabla_{z_k} \text{Div}
    \]
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
  – Recursion:
    
    $\nabla_{z_k} \text{Div} = \nabla_{y_k} \text{Div} J_{y_k}(z_k)$

    $\nabla_{y_{k-1}} \text{Div} = \nabla_{z_k} \text{Div} W_k$

  – Gradient computation:
    
    $\nabla_{W_k} \text{Div} = y_{k-1} \nabla_{z_k} \text{Div}$

    $\nabla_{b_k} \text{Div} = \nabla_{z_k} \text{Div}$

Note analogy to forward pass
For comparison: The Forward Pass

• Set $y_0 = x$

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

• 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:
  
  – \(Err = 0\)
  
  – For all \(k\), initialize \(\nabla_{W_k} Err = 0, \nabla_{b_k} Err = 0\)
  
  – For all \(t = 1: T\)
    
    • Forward pass: Compute
      
      – Output \(Y(X_t)\)
      
      – Divergence \(Div(Y_t, d_t)\)
      
      – \(Err += Div(Y_t, d_t)\)
    
    • Backward pass: For all \(k\) compute:
      
      – \(\nabla_{W_k} Div(Y_t, d_t); \nabla_{b_k} Div(Y_t, d_t)\)
      
      – \(\nabla_{W_k} Err += \nabla_{W_k} Div(Y_t, d_t); \nabla_{b_k} Err += \nabla_{b_k} Div(Y_t, d_t)\)
    
    – For all \(k\), update:
      
      \[
      W_k = W_k - \frac{\eta}{T} (\nabla_{W_k} Err)^T; \quad b_k = b_k - \frac{\eta}{T} (\nabla_{W_k} Err)^T
      \]
  
• Until \(Err\) has converged
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