Neural Networks
Learning the network: Backprop
part 2
11-785, Spring 2020
Lecture 4
Computing the gradient

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

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

ITERATE FOR \( k = 1:N \)

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

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

\[ y_j^{(k)} = f_k(z_j^{(k)}) \]
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 $j = 1 \ldots D_k$
    - $z_j^{(k)} = \sum_{i=0}^{D_{k-1}} w_{i,j}^{(k)} y_i^{(k-1)}$
    - $y_j^{(k)} = f_k\left(z_j^{(k)}\right)$
- Output:
  - $Y = y_j^{(N)}, \ j = 1 \ldots D_N$
**Gradients: Backward Computation**

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

**Initialize: Gradient w.r.t network output**

\[
\frac{\partial \text{Div}}{\partial y_i} = \frac{\partial \text{Div}(Y, d)}{\partial y^{(N)}_i} \\
\frac{\partial \text{Div}}{\partial z_i^{(N)}} = f'_k(z_i^{(N)}) \frac{\partial \text{Div}}{\partial y^{(N)}_i}
\]

**For** \( k = N - 1 \ldots 0 \)

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

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

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

$$y_i^{(k)} = f\left(z_i^{(k)}\right)$$

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

$$\begin{bmatrix} y_1^{(k)} \\ y_2^{(k)} \\ \vdots \\ y_M^{(k)} \end{bmatrix} = f\left(\begin{bmatrix} Z_1^{(k)} \\ Z_2^{(k)} \\ \vdots \\ Z_D^{(k)} \end{bmatrix}\right)$$
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

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

• 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. Regardless of the number of outputs:

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

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

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

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

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

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

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

Forward:

\[ o_i^{(k)} = y_j^{(k-1)} y_l^{(k-1)} \]

Backward:

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

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

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

• Some types of networks have multiplicative combination
A layer of multiplicative combination is a special case of vector activation.

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

\[ y_i^{(k)} = z_{2i-1}^{(k)} z_{2i}^{(k)} \]
Multiplicative combination: Can be viewed as a case of vector activations

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

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

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

\[ \frac{\partial y_i^{(k)}}{\partial z_j^{(k)}} = \alpha_{ji}^{(k)} (z_j^{(k)})^{\alpha_{ji}^{(k)} - 1} \prod_{l \neq j} (z_l^{(k)})^{\alpha_{li}^{(k)}} \]

\[ \frac{\partial \text{Div}}{\partial z_j^{(k)}} = \sum_i \frac{\partial \text{Div}}{\partial z_j^{(k)}} \frac{\partial y_i^{(k)}}{\partial z_j^{(k)}} \]
Gradients: Backward Computation

For $k = N\ldots1$
For $i = 1:\text{layer width}$

If layer has vector activation

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

Else if activation is scalar

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

\[
\frac{\partial \text{Div}}{\partial w_{ij}^{(k)}} = y_i^{(k-1)} \frac{\partial \text{Div}}{\partial z_j^{(k)}}
\]
Backward Pass for softmax output layer

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

- **For layer** $k = N - 1$ *down to 0*
  - For $i = 1 \ldots D_k$
    - $\frac{\partial \text{Div}}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$
    - $\frac{\partial \text{Div}}{\partial z_i^{(k)}} = f'_k \left( z_i^{(k)} \right) \frac{\partial \text{Div}}{\partial y_i^{(k)}}$
    - $\frac{\partial \text{Div}}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \frac{\partial \text{Div}}{\partial z_i^{(k+1)}}$ for $j = 1 \ldots D_{k+1}$
Special Case 3: 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 $v$ such that

$$(f(x) - f(x_0)) \geq v^T (x - x_0)$$

- 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 $i = 1 \ldots D_N$
    - $\frac{\partial \text{Div}}{\partial y_i} = \frac{\partial \text{Div}(Y,d)}{\partial y_i^{(N)}}$
    - $\frac{\partial \text{Div}}{\partial z_i^{(N)}} = \frac{\partial \text{Div}}{\partial y_i^{(N)}} \frac{\partial y_i^{(N)}}{\partial z_i^{(N)}}$ OR $\sum_j \frac{\partial \text{Div}}{\partial y_j^{(N)}} \frac{\partial y_j^{(N)}}{\partial z_i^{(N)}}$ (vector activation)

• For layer $k = N - 1$ down to 0
  - For $i = 1 \ldots D_k$
    - $\frac{\partial \text{Div}}{\partial y_i^{(k)}} = \sum_j w_{ij}^{(k+1)} \frac{\partial \text{Div}}{\partial z_j^{(k+1)}}$
    - $\frac{\partial \text{Div}}{\partial z_i^{(k)}} = \frac{\partial \text{Div}}{\partial y_i^{(k)}} \frac{\partial y_i^{(k)}}{\partial z_i^{(k)}}$ OR $\sum_j \frac{\partial \text{Div}}{\partial y_j^{(k)}} \frac{\partial y_j^{(k)}}{\partial z_i^{(k)}}$ (vector activation)
    - $\frac{\partial \text{Div}}{\partial w_{ji}^{(k+1)}} = y_j^{(k)} \frac{\partial \text{Div}}{\partial z_i^{(k+1)}}$ for $j = 1 \ldots D_{k+1}$

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

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

\[
W \leftarrow W - \eta \nabla_W \text{Loss}^T
\]
Training by BackProp

- Initialize weights $\mathbf{W}^{(k)}$ for all layers $k = 1 \ldots K$
- Do:
  - Initialize $\text{Loss} = 0$; For all $i, j, k$, initialize $\frac{d\text{Loss}}{dw_{i,j}^{(k)}} = 0$
  - For all $t = 1: T$ (Loop over training instances)
    - **Forward pass:** Compute
      - Output $\mathbf{Y}_t$
      - $\text{Loss} += \text{Div}(\mathbf{Y}_t, d_t)$
    - **Backward pass:** For all $i, j, k$:
      - Compute $\frac{d\text{Div}(\mathbf{Y}_t, d_t)}{dw_{i,j}^{(k)}}$
      - Compute $\frac{d\text{Loss}}{dw_{i,j}^{(k)}} += \frac{d\text{Div}(\mathbf{Y}_t, d_t)}{dw_{i,j}^{(k)}}$
  - For all $i, j, k$, update:
    $$w_{i,j}^{(k)} = w_{i,j}^{(k)} - \eta \frac{\text{Loss}}{T} \frac{d\text{Loss}}{dw_{i,j}^{(k)}}$$
- Until $\text{Loss}$ 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
  – 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 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
The computation of a single layer is easily expressed in matrix notation as (setting $y_0 = x$):

$$z_k = W_k y_{k-1} + b_k$$

$$y_k = f_k(z_k)$$
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

\[ z_2 = W_2y_1 + b_2 \]

The Complete computation

\[ y_1 = f_1(W_1x + b_1) \]
The forward pass

\[ y_2 = f_2(z_2) \]

The Complete computation

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

The Complete computation

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

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

Initialize

\[ y_0 = x \]

For \( k = 1 \) to \( N \):

\[ z_k = W_k y_{k-1} + b_k \]

\[ y_k = f_k(z_k) \]

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 error for any \( x \) is also a nested function

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

\[
y = f(z)
\]

Check:

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

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

\[ z = W y + b \]

\[ J_z(y) = W \]

• 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:

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

Check

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

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

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.
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 gradient of the divergence w.r.t. $Y$. The actual gradient depends on the divergence function.
The backward pass

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

Already computed  New term
The backward pass

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

Already computed  New term
The backward pass

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

- Already computed
- New term

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

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

Already computed  New term
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 backward pass

The Jacobian will be a diagonal matrix for scalar activations

$$\nabla_{z_{N-1}} Div = \nabla_{y_{N-1}} Div J_{y_{N-1}}(z_{N-1})$$
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

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

\[ \begin{align*}
\nabla_{w_{N-1}} \text{Div} &= y_{N-2} \nabla_{z_{N-1}} \text{Div} \\
\nabla_{b_{N-1}} \text{Div} &= \nabla_{z_{N-1}} \text{Div}
\end{align*} \]
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

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

\[ \nabla_{b_1} Div = \nabla_{z_1} Div \]
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:
    \[
    \nabla_{z_k} \text{Div} = \nabla_{y_k} \text{Div} J_{y_k}(z_k) \\
    \nabla_{y_{k-1}} \text{Div} = \nabla_{z_k} \text{Div} \mathbf{W}_k
    \]
  – Gradient computation:
    \[
    \nabla_{\mathbf{W}_k} \text{Div} = y_{k-1} \nabla_{z_k} \text{Div} \\
    \nabla_{b_k} \text{Div} = \nabla_{z_k} \text{Div}
    \]
The Backward Pass

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

- For layer $k = N$ downto 1:
  - Compute $J_{y_k}(z_k)$
    - Will require intermediate values computed in the forward pass
  - Backward recursion step: Note analogy to forward pass
    \[
    \nabla_{z_k} Div = \nabla_{y_k} Div J_{y_k}(z_k)
    \]
    \[
    \nabla_{y_{k-1}} Div = \nabla_{z_k} Div W_k
    \]
  - Gradient computation:
    \[
    \nabla_{W_k} Div = y_{k-1} \nabla_{z_k} Div
    \]
    \[
    \nabla_{b_k} Div = \nabla_{z_k} Div
    \]
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 \((W_1, b_1, W_2, b_2, ..., W_N, b_N)\)
- 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:
      - \(\nabla_{y_k} \text{Div} = \nabla_{z_{k+1}} \text{Div} W_{k+1}\)
      - \(\nabla_{z_k} \text{Div} = \nabla_{y_k} \text{Div} j_k(z_k)\)
      - \(\nabla_{W_k} \text{Div}(Y_t, d_t) = y_{k-1} \nabla_{z_k} \text{Div}; \ nabla_{b_k} \text{Div}(Y_t, d_t) = \nabla_{z_k} \text{Div}\)
      - \(\nabla_{W_k} Loss + = \nabla_{W_k} \text{Div}(Y_t, d_t); \ nabla_{b_k} Loss + = \nabla_{b_k} \text{Div}(Y_t, d_t)\)
    - For all \(k\), update:
      \[
      W_k = W_k - \frac{\eta}{T} (\nabla_{W_k} Loss)^T; \quad b_k = b_k - \frac{\eta}{T} (\nabla_{W_k} Loss)^T
      \]
- Until \(Loss\) has converged
### Setting up for digit recognition

**Training data**

<table>
<thead>
<tr>
<th>5, 0</th>
<th>2, 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2, 1</td>
<td>4, 0</td>
</tr>
<tr>
<td>0, 0</td>
<td>2, 1</td>
</tr>
</tbody>
</table>

- 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

![Neural network diagram](image)
Recognizing the digit

Training data

- 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
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