Things we will look at today

- More Backpropagation
- Still more backpropagation
- Quiz at 4:05 PM
To understand, let us just calculate!
Consider example \( \mathbf{x} \); Output for \( \mathbf{x} \) is \( \hat{y} \); Correct Answer is \( y \)

Loss \( L = (y - \hat{y})^2 \)

\[ \hat{y} = \mathbf{x}^T \mathbf{w} = x_1 w_1 + x_2 w_2 + \ldots x_d w_d \]
One Neuron Again

- Want to update $w_i$ (forget closed form solution for a bit!)
- Update rule: $w_i := w_i - \eta \frac{\partial L}{\partial w_i}$
- Now

$$\frac{\partial L}{\partial w_i} = \frac{\partial (\hat{y} - y)^2}{\partial w_i} = 2(\hat{y} - y) \frac{\partial (x_1 w_1 + x_2 w_2 + \ldots x_d w_d)}{\partial w_i}$$
One Neuron Again

We have: \( \frac{\partial L}{\partial w_i} = 2(\hat{y} - y)x_i \)

Update Rule:

\[
w_i := w_i - \eta(\hat{y} - y)x_i = w_i - \eta \delta x_i \quad \text{where} \quad \delta = (\hat{y} - y)
\]

In vector form: \( \mathbf{w} := \mathbf{w} - \eta \delta \mathbf{x} \)

Simple enough! Now let’s graduate ...
$\hat{y} = w_1^{(2)} z_1 + w_2^{(2)} z_2$

$z_1 = \tanh(a_1)$ where $a_1 = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + w_{31}^{(1)} x_3$ likewise for $z_2$
Simple Feedforward Network

- $z_1 = \tanh(a_1)$ where $a_1 = w^{(1)}_{11} x_1 + w^{(1)}_{21} x_2 + w^{(1)}_{31} x_3$
- $z_2 = \tanh(a_2)$ where $a_2 = w^{(1)}_{12} x_1 + w^{(1)}_{22} x_2 + w^{(1)}_{32} x_3$

- Output $\hat{y} = w^{(2)}_1 z_1 + w^{(2)}_2 z_2$; Loss $L = (\hat{y} - y)^2$
- Want to assign credit for the loss $L$ to each weight
Want to find: \( \frac{\partial L}{\partial w_{1}^{(2)}} \) and \( \frac{\partial L}{\partial w_{2}^{(2)}} \)

Consider \( w_{1}^{(2)} \) first

\[
\frac{\partial L}{\partial w_{1}^{(2)}} = \frac{\partial (\hat{y} - y)^2}{\partial w_{1}^{(2)}} = 2(\hat{y} - y) \frac{\partial (w_{1}^{(2)}z_{1} + w_{2}^{(2)}z_{2})}{\partial w_{1}^{(2)}} = 2(\hat{y} - y)z_{1}
\]

Familiar from earlier! Update for \( w_{1}^{(2)} \) would be

\[
w_{1}^{(2)} := w_{1}^{(2)} - \eta \frac{\partial L}{\partial w_{1}^{(2)}} = w_{1}^{(2)} - \eta \delta z_{1} \text{ with } \delta = (\hat{y} - y)
\]

Likewise, for \( w_{2}^{(2)} \) update would be \( w_{2}^{(2)} := w_{2}^{(2)} - \eta \delta z_{2} \)
There are six weights to assign credit for the loss incurred.

Consider $w^{(1)}_{11}$ for an illustration.

Rest are similar.

\[
\frac{\partial L}{\partial w^{(1)}_{11}} = \frac{\partial (\hat{y} - y)^2}{\partial w^{(1)}_{11}} = 2(\hat{y} - y) \frac{\partial (w^{(2)}_1 z_1 + w^{(2)}_2 z_2)}{\partial w^{(21)}_{11}}
\]

Now:
\[
\frac{\partial (w^{(2)}_1 z_1 + w^{(2)}_2 z_2)}{\partial w^{(1)}_{11}} = w^{(2)}_1 \frac{\partial (\tanh(w^{(1)}_{11} x_1 + w^{(1)}_{21} x_2 + w^{(1)}_{31} x_3))}{\partial w^{(1)}_{11}} + 0
\]

Which is: $w^{(2)}_1 (1 - \tanh^2(a_1)) x_1$ recall $a_1 = \hat{y} - y$

So we have:
\[
\frac{\partial L}{\partial w^{(1)}_{11}} = 2(\hat{y} - y) w^{(2)}_1 (1 - \tanh^2(a_1)) x_1
\]
Next Layer

\[
\frac{\partial L}{\partial w_{11}^{(1)}} = 2(\hat{y} - y)w_{1}^{(2)}(1 - \tanh^2(a_1))x_1
\]

Weight update:
\[
w_{11}^{(1)} := w_{11}^{(1)} - \eta \frac{\partial L}{\partial w_{11}^{(1)}}
\]

Likewise, if we were considering \( w_{22}^{(1)} \), we’d have:

\[
\frac{\partial L}{\partial w_{22}^{(1)}} = 2(\hat{y} - y)w_{2}^{(2)}(1 - \tanh^2(a_2))x_2
\]

Weight update: \( w_{22}^{(1)} := w_{22}^{(1)} - \eta \frac{\partial L}{\partial w_{22}^{(1)}} \)
Let’s clean this up...

- Recall, for top layer: \( \frac{\partial L}{\partial w_i^{(2)}} = (\hat{y} - y)z_i = \delta z_i \) (ignoring 2)

- One can think of this as: \( \frac{\partial L}{\partial w_i^{(2)}} = \delta \underbrace{z_i}_{\text{local input}} \)

- For next layer we had: \( \frac{\partial L}{\partial w_{ij}^{(1)}} = (\hat{y} - y)w_j^{(2)}(1 - \tanh^2(a_j))x_i \)

  Let \( \delta_j = (\hat{y} - y)w_j^{(2)}(1 - \tanh^2(a_j)) = \delta w_j^{(2)}(1 - \tanh^2(a_j)) \)

  (Notice that \( \delta_j \) contains the \( \delta \) term (which is the error!))

- Then: \( \frac{\partial L}{\partial w_{ij}^{(1)}} = \delta_j \underbrace{x_i}_{\text{local input}} \)

- Neat!
Let’s clean this up...

Let’s get a cleaner notation to summarize this.

Let $w_{i \rightarrow j}$ be the weight for the connection FROM node $i$ to node $j$.

Then

$$\frac{\partial L}{\partial w_{i \rightarrow j}} = \delta_j z_i$$

$\delta_j$ is the local error (going from $j$ backwards) and $z_i$ is the local input coming from $i$. 
Credit Assignment: A Graphical Revision

Let’s redraw our toy network with new notation and label nodes.
Credit Assignment: Top Layer

- Local error from 0: \( \delta = (\hat{y} - y) \), local input from 1: \( z_1 \)

\[
\frac{\partial L}{\partial w_{1 \rightarrow 0}} = \delta z_1; \text{ and update } w_{1 \rightarrow 0} := w_{1 \rightarrow 0} - \eta \delta z_1
\]
Credit Assignment: Top Layer

\[ w_5 \rightarrow z_1 \quad w_4 \rightarrow z_1 \quad w_3 \rightarrow z_1 \]

\[ w_2 \rightarrow z_1 \quad w_2 \rightarrow z_2 \]

\[ w_1 \rightarrow z_1 \quad w_1 \rightarrow z_2 \]

\[ x_1 \rightarrow 5 \quad x_2 \rightarrow 4 \quad x_3 \rightarrow 3 \]

\[ z_1 \rightarrow 1 \quad z_2 \rightarrow 2 \]

\[ \hat{y} \]
Local error from 0: $\delta = (\hat{y} - y)$, local input from 2: $z_2$

$$\frac{\partial L}{\partial w_{2\rightarrow0}} = \delta z_2 \text{ and update } w_{2\rightarrow0} := w_{2\rightarrow0} - \eta \delta z_2$$
Credit Assignment: Next Layer

\[
x_1 \overset{5}{\rightarrow} z_1 \overset{1}{\rightarrow} \hat{y}
\]

\[
x_2 \overset{4}{\rightarrow} z_2 \overset{2}{\rightarrow} \hat{y}
\]

\[
x_3 \overset{3}{\rightarrow} \hat{y}
\]
Local error from 1: $\delta_1 = (\delta)(w_{1\rightarrow 0})(1 - \tanh^2(a_1))$, local input from 3: $x_3$

\[
\frac{\partial L}{\partial w_{3\rightarrow 1}} = \delta_1 x_3 \text{ and update } w_{3\rightarrow 1} := w_{3\rightarrow 1} - \eta \delta_1 x_3
\]
Credit Assignment: Next Layer

\[ \hat{y} \rightarrow w_1 \rightarrow z_1 \rightarrow w_2 \rightarrow z_2 \rightarrow w_3 \rightarrow 1 \]

\[ x_1 \rightarrow 5 \rightarrow w_5 \rightarrow z_1 \]

\[ x_2 \rightarrow 4 \rightarrow w_4 \rightarrow z_2 \]

\[ x_3 \rightarrow 3 \rightarrow w_3 \rightarrow 1 \]
Credit Assignment: Next Layer
Local error from 2: \( \delta_2 = (\delta)(w_{2 \rightarrow 0})(1 - \tanh^2(a_2)) \), local input from 4: \( x_2 \)

\[
\frac{\partial L}{\partial w_{4 \rightarrow 2}} = \delta_2 x_2 \quad \text{and update} \quad w_{4 \rightarrow 2} := w_{4 \rightarrow 2} - \eta \delta_2 x_2
\]
Let’s Vectorize

• Let $W^{(2)} = \begin{bmatrix} w_{1\sim0} \\ w_{2\sim0} \end{bmatrix}$ (ignore that $W^{(2)}$ is a vector and hence more appropriate to use $w^{(2)}$)

• Let $W^{(1)} = \begin{bmatrix} w_{5\sim1} & w_{5\sim2} \\ w_{4\sim1} & w_{4\sim2} \\ w_{3\sim1} & w_{3\sim2} \end{bmatrix}$

• Let $Z^{(1)} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}$ and $Z^{(2)} = \begin{bmatrix} z_1 \\ z_2 \end{bmatrix}$
Feedforward Computation

1. Compute $A^{(1)} = Z^{(1)^T} W^{(1)}$
2. Applying element-wise non-linearity $Z^{(2)} = \tanh A^{(1)}$
3. Compute Output $\hat{y} = Z^{(2)^T} W^{(2)}$
4. Compute Loss on example $(\hat{y} - y)^2$
1. Top: Compute $\delta$

2. Gradient w.r.t $W^{(2)} = \delta Z^{(2)}$

3. Compute $\delta_1 = (W^{(2)^T}\delta) \odot (1 - \tanh(A^{(1)})^2)$

   Notes: (a): $\odot$ is Hadamard product. (b) have written $W^{(2)^T}\delta$ as $\delta$ can be a vector when there are multiple outputs

4. Gradient w.r.t $W^{(1)} = \delta_1 Z^{(1)}$

5. Update $W^{(2)} := W^{(2)} - \eta \delta Z^{(2)}$

6. Update $W^{(1)} := W^{(1)} - \eta \delta_1 Z^{(1)}$

7. All the dimensionalities nicely check out!
So Far

Backpropagation in the context of neural networks is all about assigning credit (or blame!) for error incurred to the weights

- We follow the path from the output (where we have an error signal) to the edge we want to consider
- We find the $\delta$s from the top to the edge concerned by using the chain rule
- Once we have the partial derivative, we can write the update rule for that weight
What did we miss?

- Exercise: What if there are multiple outputs? (look at slide from last class)
- Another exercise: Add bias neurons. What changes?
- As we go down the network, notice that we need previous $\delta$s
- If we recompute them each time, it can blow up!
- Need to book-keep derivatives as we go down the network and reuse them
A General View of Backpropagation
Some redundancy in upcoming slides, but redundancy can be good!
An Aside

- Backpropagation only refers to the method for computing the gradient.
- This is used with another algorithm such as SGD for learning using the gradient.
- Next: Computing gradient $\nabla_x f(x, y)$ for arbitrary $f$.
- $x$ is the set of variables whose derivatives are desired.
- Often we require the gradient of the cost $J(\theta)$ with respect to parameters $\theta$ i.e $\nabla_\theta J(\theta)$.
- Note: We restrict to case where $f$ has a single output.
- First: Move to more precise computational graph language!
Computational Graphs

- Formalize computation as graphs
- **Nodes** indicate variables (scalar, vector, tensor or another variable)
- **Operations** are simple functions of one or more variables
- Our graph language comes with a set of **allowable** operations
- Examples:
\[ z = xy \]

- Graph uses \( \times \) operation for the computation
Logistic Regression

Computes $\hat{y} = \sigma(x^T w + b)$
$H = \max\{0, XW + b\}$

MM is matrix multiplication and Rc is ReLU activation
Back to backprop: Chain Rule

- Backpropagation computes the chain rule, in a manner that is highly efficient.
- Let $f, g : \mathbb{R} \rightarrow \mathbb{R}$
- Suppose $y = g(x)$ and $z = f(y) = f(g(x))$
- Chain rule:
  $$\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$$
Chain rule: $\frac{dz}{dx} = \frac{dz}{dy} \frac{dy}{dx}$
Multiple Paths:
\[ \frac{dz}{dx} = \frac{dz}{dy_1} \frac{dy_1}{dx} + \frac{dz}{dy_2} \frac{dy_2}{dx} \]
Multiple Paths: \[ \frac{dz}{dx} = \sum_{j} \frac{dz}{dy_j} \frac{dy_j}{dx} \]
Chain Rule

- Consider \( x \in \mathbb{R}^m, y \in \mathbb{R}^n \)
- Let \( g : \mathbb{R}^m \rightarrow \mathbb{R}^n \) and \( f : \mathbb{R}^n \rightarrow \mathbb{R} \)
- Suppose \( y = g(x) \) and \( z = f(y) \), then

\[
\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_i}
\]

- In vector notation:

\[
\begin{pmatrix}
\frac{\partial z}{\partial x_1} \\
\vdots \\
\frac{\partial z}{\partial x_m}
\end{pmatrix} = \begin{pmatrix}
\sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_1} \\
\vdots \\
\sum_j \frac{\partial z}{\partial y_j} \frac{\partial y_j}{\partial x_m}
\end{pmatrix} = \nabla_x z = \left( \frac{\partial y}{\partial x} \right)^T \nabla_y z
\]
Chain Rule

$$\nabla_x z = \left( \frac{\partial y}{\partial x} \right)^T \nabla_y z$$

- $\left( \frac{\partial y}{\partial x} \right)$ is the $n \times m$ Jacobian matrix of $g$
- **Gradient** of $x$ is a multiplication of a Jacobian matrix $\left( \frac{\partial y}{\partial x} \right)$ with a vector i.e. the gradient $\nabla_y z$
- Backpropagation consists of applying such Jacobian-gradient products to each operation in the computational graph
- In general this need not only apply to vectors, but can apply to tensors w.l.o.g
Chain Rule

- We can of course also write this in terms of tensors.
- Let the gradient of $z$ with respect to a tensor $X$ be $\nabla_X z$.
- If $Y = g(X)$ and $z = f(Y)$, then:

$$\nabla_X z = \sum_j (\nabla_X Y_j) \frac{\partial z}{\partial Y_j}$$
Recursive Application in a Computational Graph

- Writing an algebraic expression for the gradient of a scalar with respect to any node in the computational graph that produced that scalar is straightforward using the chain-rule.
- Let for some node $x$ the successors be: $\{y_1, y_2, \ldots, y_n\}$
- Node: Computation result
- Edge: Computation dependency

$$\frac{dz}{dx} = \sum_{i=1}^{n} \frac{dz}{dy_i} \frac{dy_i}{dx}$$
Flow Graph (for previous slide)
Recursive Application in a Computational Graph

- **Fpropagation**: Visit nodes in the order after a topological sort
- Compute the value of each node given its ancestors
- **Bpropagation**: Output gradient = 1
- Now visit nodes in reverse order
- Compute gradient with respect to each node using gradient with respect to successors
- **Successors of** $x$ **in previous slide** $\{y_1, y_2, \ldots, y_n\}$:

$$\frac{dz}{dx} = \sum_{i=1}^{n} \frac{dz}{dy_i} \frac{dy_i}{dx}$$
Automatic Differentiation

- Computation of the gradient can be automatically inferred from the symbolic expression of fprop.
- Every node type needs to know:
  - How to compute its output.
  - How to compute its gradients with respect to its inputs given the gradient w.r.t its outputs.
- Makes for rapid prototyping.
To train we want to compute $\nabla_{W(1)} J$ and $\nabla_{W(2)} J$.

Two paths lead backwards from $J$ to weights: Through cross entropy and through regularization cost.
Weight decay cost is relatively simple: Will always contribute $2\lambda W^{(i)}$ to gradient on $W^{(i)}$.

Two paths lead backwards from $J$ to weights: Through cross entropy and through regularization cost.
In this approach backpropagation never accesses any numerical values.
Instead it just adds nodes to the graph that describe how to compute derivatives.
A graph evaluation engine will then do the actual computation.
Approach taken by Theano and TensorFlow.
Next time

- Optimization Methods for Deep Neural Networks