Backpropagation is the central algorithm in training neural networks.

- It’s is an algorithm for computing gradients.
- Really it’s an instance of reverse mode automatic differentiation, which is much more broadly applicable than just neural networks.
  - This is “just” a clever and efficient use of the Chain Rule for derivatives.
  - We’ll see how to implement an automatic differentiation system in this tutorial.
Recap: Gradient Descent

- **Recall:** Gradient descent moves opposite the gradient (the direction of steepest descent)

- We want to compute the cost gradient $dJ/dw$, which is the vector of partial derivatives.
  - This is the average of $dL/dw$ over all the training examples, so in this lecture we focus on computing $dL/dw$. 
Recall: if \( f(x) \) and \( x(t) \) are univariate functions, then

\[
\frac{d}{dt} f(x(t)) = \frac{df}{dx} \frac{dx}{dt}.
\]
Recall: Univariate logistic least squares model

\[
z = wx + b \\
y = \sigma(z) \\
\mathcal{L} = \frac{1}{2}(y - t)^2
\]

Let’s compute the loss derivatives.
Univariate Chain Rule

How you would have done it in calculus class

\[ L = \frac{1}{2} (\sigma(wx + b) - t)^2 \]

\[ \frac{\partial L}{\partial w} = \frac{\partial}{\partial w} \left[ \frac{1}{2} (\sigma(wx + b) - t)^2 \right] \]

\[ = \frac{1}{2} \frac{\partial}{\partial w} (\sigma(wx + b) - t)^2 \]

\[ = (\sigma(wx + b) - t) \frac{\partial}{\partial w} (\sigma(wx + b) - t) \]

\[ = (\sigma(wx + b) - t)^\prime (wx + b) \frac{\partial}{\partial w} (wx + b) \]

\[ = (\sigma(wx + b) - t)^\prime(wx + b)x \]

\[ \frac{\partial L}{\partial b} = \frac{\partial}{\partial b} \left[ \frac{1}{2} (\sigma(wx + b) - t)^2 \right] \]

\[ = \frac{1}{2} \frac{\partial}{\partial b} (\sigma(wx + b) - t)^2 \]

\[ = (\sigma(wx + b) - t) \frac{\partial}{\partial b} (\sigma(wx + b) - t) \]

\[ = (\sigma(wx + b) - t)^\prime(wx + b) \frac{\partial}{\partial b} (wx + b) \]

\[ = (\sigma(wx + b) - t)^\prime(wx + b) \]

What are the disadvantages of this approach?
Univariate Chain Rule

A more structured way to do it

Computing the loss:

\[ z = wx + b \]
\[ y = \sigma(z) \]
\[ \mathcal{L} = \frac{1}{2} (y - t)^2 \]

Computing the derivatives:

\[ \frac{d\mathcal{L}}{dy} = y - t \]
\[ \frac{d\mathcal{L}}{dz} = \frac{d\mathcal{L}}{dy} \sigma'(z) \]
\[ \frac{\partial \mathcal{L}}{\partial w} = \frac{d\mathcal{L}}{dz} x \]
\[ \frac{\partial \mathcal{L}}{\partial b} = \frac{d\mathcal{L}}{dz} \]

Remember, the goal isn’t to obtain closed-form solutions, but to be able to write a program that efficiently computes the derivatives.
Univariate Chain Rule

- We can diagram out the computations using a computation graph.
- The nodes represent all the inputs and computed quantities, and the edges represent which nodes are computed directly as a function of which other nodes.

```
Compute Loss

x
w
b

z
y
L

Compute Derivatives
```
Univariate Chain Rule

A slightly more convenient notation:

- Use $\bar{y}$ to denote the derivative $d\mathcal{L}/dy$, sometimes called the error signal.
- This emphasizes that the error signals are just values our program is computing (rather than a mathematical operation).
- This is not a standard notation, but I couldn’t find another one that I liked.

Computing the loss:

$$z = wx + b$$
$$y = \sigma(z)$$
$$\mathcal{L} = \frac{1}{2}(y - t)^2$$

Computing the derivatives:

$$\bar{y} = y - t$$
$$\bar{z} = \bar{y} \sigma'(z)$$
$$\bar{w} = \bar{z} x$$
$$\bar{b} = \bar{z}$$
Problem: What if the computation graph has fan-out > 1? This requires the multivariate Chain Rule!

$L_2$-Regularized regression

\[ z = wx + b \]
\[ y = \sigma(z) \]
\[ \mathcal{L} = \frac{1}{2} (y - t)^2 \]
\[ \mathcal{R} = \frac{1}{2} w^2 \]
\[ \mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \mathcal{R} \]

Multiclass logistic regression

\[ z_\ell = \sum_j w_{\ell j} x_j + b_\ell \]
\[ y_k = \frac{e^{z_k}}{\sum_\ell e^{z_\ell}} \]
\[ \mathcal{L} = -\sum_k t_k \log y_k \]
In the context of backpropagation:

\[
\frac{df}{dt} = \frac{\partial f}{\partial x} \frac{dx}{dt} + \frac{\partial f}{\partial y} \frac{dy}{dt}
\]

In our notation:

\[
\bar{t} = \bar{x} \frac{dx}{dt} + \bar{y} \frac{dy}{dt}
\]
**Backpropagation**

**Full backpropagation algorithm:**

Let $v_1, \ldots, v_N$ be a topological ordering of the computation graph (i.e. parents come before children.)

$v_N$ denotes the variable we’re trying to compute derivatives of (e.g. loss).

- **forward pass**
  
  For $i = 1, \ldots, N$
  
  Compute $v_i$ as a function of $\text{Pa}(v_i)$

- **backward pass**
  
  For $i = N - 1, \ldots, 1$
  
  $\overline{v_i} = \sum_{j \in \text{Ch}(v_i)} \overline{v_j} \frac{\partial v_j}{\partial v_i}$

$\overline{v_N} = 1$
**Backpropagation**

**Example:** univariate logistic least squares regression

Forward pass:

\[ z = wx + b \]
\[ y = \sigma(z) \]
\[ \mathcal{L} = \frac{1}{2} (y - t)^2 \]
\[ \mathcal{R} = \frac{1}{2} w^2 \]
\[ \mathcal{L}_{\text{reg}} = \mathcal{L} + \lambda \mathcal{R} \]

Backward pass:

\[ \bar{\mathcal{L}}_{\text{reg}} = 1 \]
\[ \bar{\mathcal{R}} = \bar{\mathcal{L}}_{\text{reg}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{R}} = \bar{\mathcal{L}}_{\text{reg}} \lambda \]
\[ \bar{\mathcal{L}} = \bar{\mathcal{L}}_{\text{reg}} \frac{d\mathcal{L}_{\text{reg}}}{d\mathcal{L}} = \bar{\mathcal{L}}_{\text{reg}} \]
\[ \bar{y} = \bar{\mathcal{L}} \frac{d\mathcal{L}}{dy} = \bar{\mathcal{L}} (y - t) \]
\[ \bar{z} = \bar{y} \frac{dy}{dz} = \bar{y} \sigma'(z) \]
\[ \bar{w} = \bar{z} \frac{dz}{dw} + \bar{\mathcal{R}} \frac{d\mathcal{R}}{dw} = \bar{z} x + \bar{\mathcal{R}} w \]
\[ \bar{b} = \bar{z} \frac{dz}{db} = \bar{z} \]
Backpropagation

Multilayer Perceptron (multiple outputs):

Forward pass:

\[ z_i = \sum_j w_{i:j}^{(1)} x_j + b_i^{(1)} \]

\[ h_i = \sigma(z_i) \]

\[ y_k = \sum_i w_{k:i}^{(2)} h_i + b_k^{(2)} \]

\[ \mathcal{L} = \frac{1}{2} \sum_k (y_k - t_k)^2 \]

Backward pass:

\[ \overline{\mathcal{L}} = 1 \]

\[ \overline{y_k} = \overline{\mathcal{L}} (y_k - t_k) \]

\[ \overline{w_{ki}^{(2)}} = \overline{y_k} h_i \]

\[ \overline{b_k^{(2)}} = \overline{y_k} \]

\[ \overline{h_i} = \sum_k \overline{y_k} w_{ki}^{(2)} \]

\[ \overline{z_i} = \overline{h_i} \sigma'(z_i) \]

\[ \overline{w_{ij}^{(1)}} = \overline{z_i} x_j \]

\[ \overline{b_i^{(1)}} = \overline{z_i} \]
Computation graphs showing individual units are cumbersome.

As you might have guessed, we typically draw graphs over the vectorized variables.

We pass messages back analogous to the ones for scalar-valued nodes.
Vector Form

- Consider this computation graph:

  ![Computation Graph]

- Backprop rules:

  \[
  z_j = \sum_k y_k \frac{\partial y_k}{\partial z_j}
  \]

  \[
  z = \frac{\partial y^\top}{\partial z} y,
  \]

  where \( \frac{\partial y}{\partial z} \) is the Jacobian matrix:

  \[
  \frac{\partial y}{\partial z} = \begin{pmatrix}
  \frac{\partial y_1}{\partial z_1} & \cdots & \frac{\partial y_1}{\partial z_n} \\
  \vdots & \ddots & \vdots \\
  \frac{\partial y_m}{\partial z_1} & \cdots & \frac{\partial y_m}{\partial z_n}
  \end{pmatrix}
  \]
Vector Form

Examples

- **Matrix-vector product**

  \[ z = Wx \quad \frac{\partial z}{\partial x} = W \quad \bar{x} = W^\top \bar{z} \]

- **Elementwise operations**

  \[ y = \exp(z) \quad \frac{\partial y}{\partial z} = \begin{pmatrix} \exp(z_1) & 0 \\ \vdots & \ddots \\ 0 & \exp(z_D) \end{pmatrix} \quad \bar{z} = \exp(z) \circ \bar{y} \]

- **Note:** we never explicitly construct the Jacobian. It’s usually simpler and more efficient to compute the VJP directly.
Full backpropagation algorithm (vector form):

Let \( v_1, \ldots, v_N \) be a **topological ordering** of the computation graph (i.e. parents come before children.) \( v_N \) denotes the variable we’re trying to compute derivatives of (e.g. loss). It’s a scalar, which we can treat as a 1-D vector.

- **forward pass**
  - For \( i = 1, \ldots, N \)
    - Compute \( v_i \) as a function of \( \text{Pa}(v_i) \)
  - \( \overline{v_N} = 1 \)

- **backward pass**
  - For \( i = N - 1, \ldots, 1 \)
    - \( \overline{v_i} = \sum_{j \in \text{Ch}(v_i)} \frac{\partial v_i}{\partial v_i}^\top \overline{v_j} \)
MLP example in vectorized form:

Forward pass:

\[ z = W^{(1)}x + b^{(1)} \]
\[ h = \sigma(z) \]
\[ y = W^{(2)}h + b^{(2)} \]
\[ \mathcal{L} = \frac{1}{2} \| t - y \|^2 \]

Backward pass:

\[ \mathcal{L} = 1 \]
\[ \bar{y} = \mathcal{L}(y - t) \]
\[ \mathcal{W}^{(2)} = \bar{y}h^\top \]
\[ \bar{b}^{(2)} = \bar{y} \]
\[ \bar{h} = \mathcal{W}^{(2)}\top \bar{y} \]
\[ \bar{z} = \bar{h} \circ \sigma'(z) \]
\[ \mathcal{W}^{(1)} = \bar{z}x^\top \]
\[ \bar{b}^{(1)} = \bar{z} \]
Some Thoughts

- Backprop is used to train the overwhelming majority of neural nets today.
  - Even optimization algorithms much fancier than gradient descent (e.g. second-order methods) use backprop to compute the gradients.

- Despite its practical success, backprop is believed to be neurally implausible.
  - No evidence for biological signals analogous to error derivatives.
  - All the biologically plausible alternatives we know about learn much more slowly (on computers).
  - So how on earth does the brain learn?
> **Automatic differentiation (autodiff)** refers to a general way of taking a program which computes a value, and automatically constructing a procedure for computing derivatives of that value.
>   - Today, we focus on reverse mode autodiff. There is also a forward mode, which is for computing directional derivatives.

> **Backpropagation** is the special case of autodiff applied to neural nets
>   - But in machine learning, we often use backprop synonymously with autodiff

> **Autograd** is the name of a particular autodiff package.
>   - But lots of people, including the PyTorch developers, got confused and started using “autograd” to mean “autodiff”
What Autodiff Is Not: Finite Differences

- We often use finite differences to check our gradient calculations.
  - One-sided version:
    \[
    \frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i, \ldots, x_N)}{h}
    \]
  - Two-sided version:
    \[
    \frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) \approx \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i - h, \ldots, x_N)}{2h}
    \]
Autodiff Is Not: Finite Differences

- Autodiff is not finite differences.
  - Finite differences are expensive, since you need to do a forward pass for each derivative.
  - It also induces huge numerical error.
  - Normally, we only use it for testing.

- Autodiff is both efficient (linear in the cost of computing the value) and numerically stable.
Autodiff Is Not: Symbolic Differentiation

- Autodiff is not symbolic differentiation (e.g. Mathematica).
  - Symbolic differentiation can result in complex and redundant expressions.
  - Mathematica’s derivatives for one layer of soft ReLU (univariate case):

\[
D[\log(1 + \exp(w \cdot x + b)), w] = \frac{e^{b+w \cdot x} \cdot w}{1 + e^{b+w \cdot x}}
\]

- Derivatives for two layers of soft ReLU:

\[
D[\log(1 + \exp(w2 \cdot \log(1 + \exp(w1 \cdot x + b1)) + b2)), w1] = \frac{e^{b1+b2+w1 x+w2 \log[1+e^{b1+w1 x}]} \cdot w2 \cdot x}{\left(1 + e^{b1+w1 x}\right) \left(1 + e^{b2+w2 \log[1+e^{b1+w1 x}]\right)}
\]

- There might not be a convenient formula for the derivatives.
- The goal of autodiff is not a formula, but a procedure for computing derivatives.
Recall how we computed the derivatives of logistic least squares regression. An autodiff system should transform the left-hand side into the right-hand side.

**Computing the loss:**

\[
\begin{align*}
  z &= wx + b \\
  y &= \sigma(z) \\
  \mathcal{L} &= \frac{1}{2} (y - t)^2
\end{align*}
\]

**Computing the derivatives:**

\[
\begin{align*}
  \overline{\mathcal{L}} &= 1 \\
  \overline{y} &= y - t \\
  \overline{z} &= \overline{y} \sigma'(z) \\
  \overline{w} &= \overline{z} x \\
  \overline{b} &= \overline{z}
\end{align*}
\]
What Autodiff Is

• An autodiff system will convert the program into a sequence of primitive operations (ops) which have specified routines for computing derivatives.

• In this representation, backprop can be done in a completely mechanical way.

Sequence of primitive operations:

Original program:

\[ z = wx + b \]
\[ y = \frac{1}{1 + \exp(-z)} \]
\[ L = \frac{1}{2} (y - t)^2 \]

\[ t_1 = wx \]
\[ z = t_1 + b \]
\[ t_3 = -z \]
\[ t_4 = \exp(t_3) \]
\[ t_5 = 1 + t_4 \]
\[ y = 1/t_5 \]
\[ t_6 = y - t \]
\[ t_7 = t_6^2 \]
\[ L = t_7/2 \]
import autograd.numpy as np
from autograd import grad

def sigmoid(x):
    return 0.5*(np.tanh(x) + 1)

def logistic_predictions(weights, inputs):
    # Outputs probability of a label being true according to logistic model.
    return sigmoid(np.dot(inputs, weights))

def training_loss(weights):
    # Training loss is the negative log-likelihood of the training labels.
    preds = logistic_predictions(weights, inputs)
    label_probabilities = preds * targets + (1 - preds) * (1 - targets)
    return -np.sum(np.log(label_probabilities))

... (load the data) ...

# Define a function that returns gradients of training loss using Autograd.
training_gradient_fun = grad(training_loss)

# Optimize weights using gradient descent.
weights = np.array([[0.0, 0.0, 0.0]])
print "Initial loss:", training_loss(weights)
for i in xrange(100):
    weights -= training_gradient_fun(weights) * 0.01
print "Trained loss:", training_loss(weights)
Autograd

- The rest of this tutorial covers how Autograd is implemented.
- Source code for the original Autograd package:
  - https://github.com/HIPS/autograd
- Autodidact, a pedagogical implementation of Autograd — you are encouraged to read the code.
  - https://github.com/mattjj/autodidact
  - Thanks to Matt Johnson for providing this!
Building the Computation Graph

Most autodiff systems, including Autograd, explicitly construct the computation graph.

- Some frameworks like TensorFlow provide mini-languages for building computation graphs directly. Disadvantage: need to learn a totally new API.
- Autograd instead builds them by tracing the forward pass computation, allowing for an interface nearly indistinguishable from NumPy.

The Node class (defined in tracer.py) represents a node of the computation graph. It has attributes:

- value, the actual value computed on a particular set of inputs
- fun, the primitive operation defining the node
- args and kwargs, the arguments the op was called with
- parents, the parent Nodes
Building the Computation Graph

- Autograd’s fake NumPy module provides primitive ops which look and feel like NumPy functions, but secretly build the computation graph.
- They wrap around NumPy functions:

```
Node a
value: a
function: F
parents: [x]
```

```
primitive

Node b
value: b
function: anp.sum
parents: [a]
```
Building the Computation Graph

Example:

def logistic(z):
    return 1. / (1. + np.exp(-z))

# that is equivalent to:
def logistic2(z):
    return np.reciprocal(np.add(1, np.exp(np.negative(z)))))

z = 1.5
y = logistic(z)
Recap: Vector-Jacobian Products

- Recall: the Jacobian is the matrix of partial derivatives:

\[ J = \frac{\partial y}{\partial x} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_n} \end{pmatrix} \]

- The backprop equation (single child node) can be written as a vector-Jacobian product (VJP):

\[ \bar{x}_j = \sum_i \bar{y}_i \frac{\partial y_i}{\partial x_j} \quad \bar{x} = \bar{y}^\top J \]

- That gives a row vector. We can treat it as a column vector by taking

\[ \bar{x} = J^\top \bar{y} \]
Recap: Vector-Jacobian Products

Examples

- **Matrix-vector product**

  \[ z = Wx \quad J = W \quad \bar{x} = W^\top \bar{z} \]

- **Elementwise operations**

  \[ y = \exp(z) \quad J = \begin{pmatrix} \exp(z_1) & 0 \\ \vdots & \ddots \\ 0 & \exp(z_D) \end{pmatrix} \quad \bar{z} = \exp(z) \circ \bar{y} \]

- **Note:** we never explicitly construct the Jacobian. It’s usually simpler and more efficient to compute the VJP directly.
Consider a naïve backprop implementation where the $z$ module needs to compute $\bar{z}$ using the formula:

$$\bar{z} = \frac{\partial r}{\partial z} \bar{r} + \frac{\partial s}{\partial z} \bar{s} + \frac{\partial t}{\partial z} \bar{t}$$

This breaks modularity, since $z$ needs to know how it’s used in the network in order to compute partial derivatives of $r$, $s$, and $t$. 
Backprop as Message Passing:

- Each node receives a bunch of messages from its children, which it aggregates to get its error signal. It then passes messages to its parents.
- Each of these messages is a VJP.
- This formulation provides modularity: each node needs to know how to compute its outgoing messages, i.e. the VJPs corresponding to each of its parents (arguments to the function).
- The implementation of $z$ doesn’t need to know where $\overline{z}$ came from.
Vector-Jacobian Products

- For each primitive operation, we must specify VJPs for each of its arguments. Consider \( y = \exp(x) \).
- This is a function which takes in the output gradient (i.e. \( \bar{y} \)), the answer (\( y \)), and the arguments (\( x \)), and returns the input gradient (\( \bar{x} \)).
- `defvjp` (defined in `core.py`) is a convenience routine for registering VJPs. It just adds them to a dict.
- Examples from `numpy/numpy_vjps.py`

```python
defvjp(negative, lambda g, ans, x: -g)
defvjp(exp, lambda g, ans, x: ans * g)
defvjp(log, lambda g, ans, x: g / x)
defvjp(add, lambda g, ans, x, y : g, lambda g, ans, x, y : g)
defvjp(multiply, lambda g, ans, x, y : y * g, lambda g, ans, x, y : x * g)
defvjp(subtract, lambda g, ans, x, y : g, lambda g, ans, x, y : -g)
```
Backward Pass

- The backwards pass is defined in `core.py`.
- The argument `g` is the error signal for the end node; for us this is always $\overline{L} = 1$.

```python
def backward_pass(g, end_node):
    outgrads = {end_node: g}
    for node in toposort(end_node):
        outgrad = outgrads.pop(node)
        fun, value, args, kwargs, argnums = node.recipe
        for argnum, parent in zip(argnums, node.parents):
            vjp = primitive_vjps[fun][argnum]
            parent_grad = vjp(outgrad, value, *args, **kwargs)
            outgrads[parent] = add_outgrads(outgrads.get(parent), parent_grad)
    return outgrad

def add_outgrads(prev_g, g):
    if prev_g is None:
        return g
    return prev_g + g
```
Backward Pass

- `grad` (in `differential_operators.py`) is just a wrapper around `make_vjp` (in `core.py`) which builds the computation graph and feeds it to `backward_pass`.
- `grad` itself is viewed as a VJP, if we treat $\overrightarrow{L}$ as the $1 \times 1$ matrix with entry 1.

$$\frac{\partial L}{\partial w} = \frac{\partial L}{\partial \overrightarrow{L}}$$

def make_vjp(fun, x):
    
    """Trace the computation to build the computation graph, and return
    a function which implements the backward pass."""
    start_node = Node.new_root()
    end_value, end_node = trace(start_node, fun, x)
    def vjp(g):
        return backward_pass(g, end_node)
    return vjp, end_value

def grad(fun, argnum=0):
    def gradfun(*args, **kwargs):
        unary_fun = lambda x: fun(*subval(args, argnum, x), **kwargs)
        vjp, ans = make_vjp(unary_fun, args[argnum])
        return vjp(np.ones_like(ans))
    return gradfun
Recap

- We saw three main parts to the code:
  - tracing the forward pass to build the computation graph
  - vector-Jacobian products for primitive ops
  - the backwards pass

- Building the computation graph requires fancy NumPy gymnastics, but other two items are basically what I showed you.

- You’re encouraged to read the full code (< 200 lines!) at:
  
  https://github.com/mattjj/autodidact/tree/master/autograd