Topics:
• Neural Networks
• Backpropagation
• **Assignment 1 out!**
  • Due Feb 7th
  • Start now, start now, start now!
  • Start now, start now, start now!
  • Start now, start now, start now!

• **Piazza**
  • Be active!!!
  • Extra credit!

• **Office hours**
  • Let us know special topic requests (e.g. PS0, Assignment 1, research paper discussion, etc.)
Example with an image with 4 pixels, and 3 classes (cat/dog/ship)

Input image

Stretch pixels into column

\[
\begin{bmatrix}
56 \\
231 \\
24 \\
2
\end{bmatrix}
\]

\[
W = \begin{bmatrix}
0.2 & -0.5 & 0.1 & 2.0 \\
1.5 & 1.3 & 2.1 & 0.0 \\
0 & 0.25 & 0.2 & -0.3
\end{bmatrix}
\]

\[
b = \begin{bmatrix}
56 \\
231 \\
24 \\
2
\end{bmatrix}
\]

\[
\begin{bmatrix}
1.1 \\
3.2 \\
-1.2
\end{bmatrix}
\]

\[
\text{Cat score} = 437.9
\]

\[
\text{Dog score} = 61.95
\]

\[
\text{Ship score} = -96.8
\]

Adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, from CS 231n
We can find the steepest descent direction by computing the **derivative (gradient)**:

\[
f'(a) = \lim_{h \to 0} \frac{f(a + h) - f(a)}{h}
\]

Steepest descent direction is the **negative gradient**

**Intuitively**: Measures how the function changes as the argument \( a \) changes by a small step size

- As step size goes to zero

**In Machine Learning**: Want to know how the **loss function** changes as **weights** are varied

- Can consider each parameter separately by taking **partial derivative** of loss function with respect to that parameter

This idea can be turned into an algorithm (gradient descent)

- Choose a model: $f(x, W) = Wx$
- Choose loss function: $L_i = |y - Wx_i|^2$
- Calculate partial derivative for each parameter: $\frac{\partial L}{\partial w_i}$
- Update the parameters: $w_i = w_i - \frac{\partial L}{\partial w_i}$
- Add learning rate to prevent too big of a step: $w_i = w_i - \alpha \frac{\partial L}{\partial w_i}$
- Repeat (from Step 3)
Often, we only compute the gradients across a small subset of data.

- **Full Batch Gradient Descent**
  \[ L = \frac{1}{N} \sum L(f(x_i, W), y_i) \]

- **Mini-Batch Gradient Descent**
  \[ L = \frac{1}{M} \sum L(f(x_i, W), y_i) \]
  - Where M is a *subset* of data

- We iterate over mini-batches:
  - Get mini-batch, compute loss, compute derivatives, and take a set
Gradient Descent

original $W$

negative gradient direction

http://demonstrations.wolfram.com/VisualizingTheGradientVector/
For some functions, we can analytically derive the partial derivative

**Example:**

<table>
<thead>
<tr>
<th>Function</th>
<th>Loss</th>
</tr>
</thead>
<tbody>
<tr>
<td>$f(w, x_i) = w^T x_i$</td>
<td>$(y_i - w^T x_i)^2$</td>
</tr>
</tbody>
</table>

(Assume $w$ and $x_i$ are column vectors, so same as $w \cdot x_i$)

**Dataset:** $N$ examples (indexed by $k$)

**Update Rule**

$$w_j \leftarrow w_j + 2\eta \sum_{k=1}^{N} \delta_k x_{kj}$$

**Derivation of Update Rule**

$$L = \sum_{k=1}^{N} (y_k - w^T x_k)^2$$

Gradient descent tells us we should update $w$ as follows to minimize $L$:

$$w_j \leftarrow w_j - \eta \frac{\partial L}{\partial w_j}$$

So what’s $\frac{\partial L}{\partial w_j}$?

$$\frac{\partial L}{\partial w_j} = \sum_{k=1}^{N} \frac{\partial}{\partial w_j} (y_k - w^T x_k)^2$$

$$= \sum_{k=1}^{N} 2(y_k - w^T x_k) \frac{\partial}{\partial w_j} (y_k - w^T x_k)$$

$$= -2 \sum_{k=1}^{N} \delta_k \frac{\partial}{\partial w_j} w^T x_k$$

...where...

$$\delta_k = y_k - w^T x_k$$

$$= -2 \sum_{k=1}^{N} \delta_k \frac{\partial}{\partial w_j} \sum_{l=1}^{m} w_l x_{kl}$$

$$= -2 \sum_{k=1}^{N} \delta_k x_{kj}$$
If we add a **non-linearity (sigmoid)**, derivation is more complex

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

First, one can derive that: \( \sigma'(x) = \sigma(x)(1 - \sigma(x)) \)

\[ f(x) = \sigma\left(\sum_k w_k x_k\right) \]

\[ L = \sum_i \left(y_i - \sigma\left(\sum_k w_k x_{ik}\right)\right)^2 \]

\[ \frac{\partial L}{\partial w_j} = \sum_i 2 \left(y_i - \sigma\left(\sum_k w_k x_{ik}\right)\right) \sigma'\left(\sum_k w_k x_{ik}\right) \frac{\partial}{\partial w_j} \sum_k w_k x_{ik} \]

\[ = \sum_i -2 \left(y_i - \sigma\left(\sum_k w_k x_{ik}\right)\right) \sigma'\left(\sum_k w_k x_{ik}\right) \frac{\partial}{\partial w_j} \sum_k w_k x_{ik} \]

\[ = \sum_i -2 \delta_i \sigma(d_i)(1 - \sigma(d_i)) x_{ij} \]

where \( \delta_i = y_i - f(x_i) \quad d_i = \sum w_k x_{ik} \)

**The sigmoid perception update rule:**

\[ w_j \leftarrow w_j + 2\eta \sum_{k=1}^N \frac{\partial L}{\partial w_j} x_{ij} \]

where \( \delta_i = y_i - \sigma_i \)

---

**Adding a Non-Linear Function**
A **linear classifier** can be broken down into:

- **Input**
- A function of the input
- A loss function

It’s all just one function that can be decomposed into building blocks.

**What Does a Linear Classifier Consist of?**

- **Input**: $X$
- **Model**: $w \cdot x \rightarrow u \rightarrow \frac{1}{1 + e^{-u}} \rightarrow p$
- **Loss Function**: $-\log(p) \rightarrow L$
The same two-layered neural network corresponds to adding another weight matrix

- We will prefer the linear algebra view, but use some terminology from neural networks (& biology)

\[
\begin{align*}
\mathbf{x} & \quad \mathbf{W}_1 \quad \mathbf{W}_2 \\
& = \\
& f(\mathbf{x}, \mathbf{W}_1, \mathbf{W}_2) = \sigma(\mathbf{W}_2 \sigma(\mathbf{W}_1 \mathbf{x}))
\end{align*}
\]

Figure adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Large (deep) networks can be built by adding more and more layers.

Three-layered neural networks can represent any function.

- The number of nodes could grow unreasonably (exponential or worse) with respect to the complexity of the function.

We will show them without edges:

\[ f(x, W_1, W_2, W_3) = \sigma(W_2 \sigma(W_1 x)) \]

---

*Figure adapted from slides by Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n*
Demo

- [http://playground.tensorflow.org](http://playground.tensorflow.org)
Computation Graphs
Functions can be made \textbf{arbitrarily complex} (subject to memory and computational limits), e.g.:

\[ f(x, W) = \sigma(W_5 \sigma(W_4 \sigma(W_3 \sigma(W_2 \sigma(W_1 x)))) \]

We can use \textbf{any type of differentiable function (layer)} we want!

- At the end, \textbf{add the loss function}

Composition can have \textbf{some structure}
The world is **compositional**!

We want our **model** to reflect this

Empirical and theoretical evidence that it makes **learning complex functions** easier

Note that **prior state of art engineered features** often had this compositionality as well

- Pixels -> edges -> object parts -> objects

![Compositionality Diagram](image-url)
- We are learning **complex models** with significant amount of parameters (millions or billions)
- How do we compute the gradients of the **loss** (at the end) with respect to **internal** parameters?
- Intuitively, want to understand how **small changes** in weight deep inside are **propagated** to affect the **loss function** at the end

![Diagram of computing gradients in complex function](https://via.placeholder.com/150)

$$\frac{\partial L}{\partial w_i}$$

**Computing Gradients in Complex Function**
Given a library of simple functions

\[ \sin(x) \quad \log(x) \quad \cos(x) \quad x^3 \quad \exp(x) \]

Compose into a complicate function

\[ -\log \left( \frac{1}{1 + e^{-w \cdot x}} \right) \]

Adapted from slides by: Marc'Aurelio Ranzato, Yann LeCun
To develop a general algorithm for this, we will view the function as a **computation graph**

Graph can be any **directed acyclic graph (DAG)**

- Modules must be differentiable to support gradient computations for gradient descent

A **training algorithm** will then process this graph, **one module at a time**

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*
Directed Acyclic Graphs (DAGs)

• Exactly what the name suggests
  – Directed edges
  – No (directed) cycles
  – Underlying undirected cycles okay
Directed Acyclic Graphs (DAGs)

• Concept
  – Topological Ordering
Directed Acyclic Graphs (DAGs)
\[ f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2) \]
-\log\left(\frac{1}{1 + e^{-w \cdot x}}\right)

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Backpropagation
Given this computation graph, the training algorithm will:

- Calculate the current model’s outputs (called the **forward pass**)
- Calculate the gradients for each module (called the **backward pass**)

Backward pass is a recursive algorithm that:

- Starts at **loss function** where we know how to calculate the gradients
- Progresses back through the modules
- Ends in the **input layer** where we do not need gradients (no parameters)

This algorithm is called **backpropagation**
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Layer 1 → Layer 2 → Layer 3

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Layer 1 → [Green Arrow] → Layer 3

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*
Note that we must store the intermediate outputs of all layers!

- This is because we will need them to compute the gradients (the gradient equations will have terms with the output values in them).

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
In the **backward pass**, we seek to calculate the gradients of the loss with respect to the module’s parameters

- Assume that we have the gradient of the loss with respect to the **module’s outputs** (given to us by upstream module)
- We will also pass the gradient of the loss with respect to the **module’s inputs**
  - This is not required for update the module’s weights, but passes the gradients back to the previous module

### Problem:

- We can compute local gradients: \( \{ \frac{\partial h^\ell}{\partial h^{\ell-1}}, \frac{\partial h^\ell}{\partial W} \} \)
- We are given: \( \frac{\partial L}{\partial h^\ell} \)
- Compute: \( \{ \frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W} \} \)

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*
We can compute **local gradients**: \( \left\{ \frac{\partial h^\ell}{\partial h^{\ell-1}}, \frac{\partial h^\ell}{\partial W} \right\} \)

This is just the **derivative of our function** with respect to its parameters and inputs!

**Example:**

If \( h^\ell = Wh^{\ell-1} \)

then \( \frac{\partial h^\ell}{\partial h^{\ell-1}} = W \)

and \( \frac{\partial h_i^\ell}{\partial w_i} = h^{\ell-1, T} \)
We want to compute:
\[ \left\{ \frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W} \right\} \]

We will use the *chain rule* to do this:

**Chain Rule:**
\[ \frac{\partial z}{\partial x} = \frac{\partial z}{\partial y} \cdot \frac{\partial y}{\partial x} \]
We will use the **chain rule** to compute: \( \{ \frac{\partial L}{\partial h^{\ell-1}}, \frac{\partial L}{\partial W} \} \)

**Gradient of loss w.r.t. inputs:** \( \frac{\partial L}{\partial h^{\ell-1}} = \frac{\partial L}{\partial h^{\ell}} \cdot \frac{\partial h^{\ell}}{\partial h^{\ell-1}} \)

**Gradient of loss w.r.t. weights:** \( \frac{\partial L}{\partial W} = \frac{\partial L}{\partial h^{\ell}} \cdot \frac{\partial h^{\ell}}{\partial W} \)

---

**Computing the Gradients of Loss**

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Step 2: Compute Gradients wrt parameters: **Backward Pass**

Layer 1 → Layer 2 → Backward Arrow

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Step 2: Compute Gradients wrt parameters: **Backward Pass**

Layer 1 → **Green Arrow** → Layer 3

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
Neural Network Training

Step 1: Compute Loss on Mini-Batch: **Forward Pass**

Step 2: Compute Gradients wrt parameters: **Backward Pass**

Layer 1

Layer 2

Layer 3

Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun
**Step 1:** Compute Loss on Mini-Batch: *Forward Pass*

**Step 2:** Compute Gradients wrt parameters: *Backward Pass*

**Step 3:** Use gradient to update *all parameters* at the end

Backpropagation is the application of gradient descent to a computation graph via the chain rule!

$$w_i = w_i - \alpha \frac{\partial L}{\partial w_i}$$

*Adapted from figure by Marc'Aurelio Ranzato, Yann LeCun*
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

Want: \( \frac{\partial f}{\partial x}, \ \frac{\partial f}{\partial y}, \ \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \(x = -2, y = 5, z = -4\)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \(\frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z}\)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

E.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

\( e.g. \ x = -2, \ y = 5, \ z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, y = 5, z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)

Chain rule:

\[ \frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} \]

Upstream gradient  Local gradient
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[
\begin{align*}
q &= x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \\
f &= qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q
\end{align*}
\]

Want: \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)

Chain rule:

\[
\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y}
\]
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \ y = 5, \ z = -4 \)

\[ q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1 \]

\[ f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q \]

Want: \( \frac{\partial f}{\partial x}, \ \frac{\partial f}{\partial y}, \ \frac{\partial f}{\partial z} \)

**Chain rule:**

\[ \frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} \]

Upstream gradient  
Local gradient
Backpropagation: a simple example

\[ f(x, y, z) = (x + y)z \]

e.g. \( x = -2, \, y = 5, \, z = -4 \)

\[
q = x + y \quad \frac{\partial q}{\partial x} = 1, \quad \frac{\partial q}{\partial y} = 1
\]

\[
f = qz \quad \frac{\partial f}{\partial q} = z, \quad \frac{\partial f}{\partial z} = q
\]

Want:

\[
\frac{\partial f}{\partial x}, \quad \frac{\partial f}{\partial y}, \quad \frac{\partial f}{\partial z}
\]

Chain rule:

\[
\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \cdot \frac{\partial q}{\partial x}
\]

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Backpropagation: a simple example
Backpropagation: a simple example
Patterns in backward flow
Patterns in backward flow

Q: What is an add gate?
Patterns in backward flow

**add** gate: gradient distributor
Patterns in backward flow

**add gate**: gradient distributor

**Q**: What is a **max** gate?
Patterns in backward flow

**add** gate: gradient distributor

**max** gate: gradient router
**Patterns in backward flow**

**add gate:** gradient distributor

**max gate:** gradient router

**Q:** What is a **mul gate?**
Patterns in backward flow

- **add** gate: gradient distributor
- **max** gate: gradient router
- **mul** gate: gradient switcher
Gradients add at branches
Duality in Fprop and Bprop
Deep Learning = Differentiable Programming

• Computation = Graph
  – Input = Data + Parameters
  – Output = Loss
  – Scheduling = Topological ordering

• What do we need to do?
  – Generic code for representing the graph of modules
  – Specify modules (both forward and backward function)
Modularized implementation: forward / backward API

Graph (or Net) object (rough pseudo code)

```python
class ComputationalGraph(object):
    #...
    def forward(inputs):
        # 1. [pass inputs to input gates...]
        # 2. forward the computational graph:
        for gate in self.graph.nodes_topologically_sorted():
            gate.forward()
        return loss # the final gate in the graph outputs the loss
    def backward():
        for gate in reversed(self.graph.nodes_topologically_sorted()):
            gate.backward() # little piece of backprop (chain rule applied)
        return inputs_gradients
```
Modularized implementation: forward / backward API

(x, y, z are scalars)
Modularized implementation: forward / backward API

\[(x, y, z \text{ are scalars})\]

```python
class MultiplyGate(object):
    def forward(x, y):
        z = x*y
        self.x = x  # must keep these around!
        self.y = y
        return z

    def backward(dz):
        dx = self.y * dz  # [dz/dx \* dL/dz]
        dy = self.x * dz  # [dz/dy \* dL/dz]
        return [dx, dy]
```

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
### Example: Caffe layers

<table>
<thead>
<tr>
<th>Layer Name</th>
<th>Description</th>
<th>Date Created</th>
</tr>
</thead>
<tbody>
<tr>
<td>AlexNet</td>
<td>A convolutional neural network used for image classification tasks.</td>
<td>x year ago</td>
</tr>
<tr>
<td>YOLO</td>
<td>A popular real-time object detection system.</td>
<td>x year ago</td>
</tr>
<tr>
<td>ResNet</td>
<td>A deep learning architecture that achieved state-of-the-art image classification results.</td>
<td>x year ago</td>
</tr>
<tr>
<td>VGG</td>
<td>Another convolutional neural network architecture known for achieving high performance on ImageNet.</td>
<td>x year ago</td>
</tr>
</tbody>
</table>

*Note: Caffe is licensed under BSD 2-Clause.*
Caffe Sigmoid Layer

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

\[(1 - \sigma(x)) \sigma(x) \] * top_diff (chain rule)
Backpropagation and Automatic Differentiation
Backpropagation does not really spell out how to **efficiently** carry out the necessary computations.

But the idea can be applied to any **directed acyclic graph (DAG)**.

- Graph represents an **ordering constraining** which paths must be calculated first.

Given an ordering, we can then iterate from the last module backwards, **applying the chain rule**.

- We will store, for each node, its **gradient outputs for efficient computation**.
- We will do this **automatically** by computing backwards function for primitives and as you write code, express the function with them.

This is called reverse-mode **automatic differentiation**.
Computation = Graph

- Input = Data + Parameters
- Output = Loss
- Scheduling = Topological ordering

Auto-Diff

- A family of algorithms for implementing chain-rule on computation graphs

Deep Learning = Differentiable Programming
We want to find the partial derivative of output $f$ (output) with respect to all intermediate variables

- Assign intermediate variables

Simplify notation:
Denote bar as: $\bar{a}_3 = \frac{\partial f}{\partial a_3}$

- Start at end and move backward

Example

$$f(x_1, x_2) = x_1x_2 + \sin(x_2)$$
\[ f(x_1, x_2) = x_1x_2 + \sin(x_2) \]

\[ a_3 = \frac{\partial f}{\partial a_3} = 1 \]

\[ a_1 = \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial (a_1 + a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3} 1 = a_3 \]

\[ a_2 = \frac{\partial f}{\partial a_2} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_2} = a_3 \]

\[ x_2^{P1} = \frac{\partial f}{\partial a_1} \frac{\partial a_1}{\partial x_2} = a_1 \cos(x_2) \]

\[ x_2^{P2} = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2} \frac{\partial (x_1x_2)}{\partial x_2} = a_2x_1 \]

\[ x_1 = \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_1} = a_2x_2 \]

Gradients from multiple paths summed
Patterns of Gradient Flow: Addition

\[ f(x_1, x_2) = x_1x_2 + \sin(x_2) \]

\[ \frac{\partial f}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_1} = \frac{\partial f}{\partial a_3} \frac{\partial (a_1 + a_2)}{\partial a_1} = \frac{\partial f}{\partial a_3} 1 = \overline{a}_3 \]

\[ \frac{\partial f}{\partial a_2} = \frac{\partial f}{\partial a_3} \frac{\partial a_3}{\partial a_2} = \overline{a}_3 \]

Addition operation distributes gradients along all paths!
Multiplication operation is a gradient switcher (multiplies it by the values of the other term)

\[
\begin{align*}
\overline{x_2} &= \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_2} = \frac{\partial f}{\partial a_2} \frac{\partial (x_1x_2)}{\partial x_2} = a_2 x_1 \\
\overline{x_1} &= \frac{\partial f}{\partial a_2} \frac{\partial a_2}{\partial x_1} = a_2 x_2
\end{align*}
\]
Several other patterns as well, e.g.:

Max operation **selects** which path to push the gradients through

- Gradient flows along the path that was “selected” to be max
- This information must be recorded in the forward pass

The **flow of gradients** is one of the **most important aspects** in deep neural networks

- If gradients **do not flow backwards properly**, learning slows or stops!
Key idea is to **explicitly store computation graph in memory and corresponding gradient functions**

- **Nodes** broken down to **basic primitive computations** (addition, multiplication, log, etc.) for which corresponding derivative is known

\[
\overline{x_2} = \frac{\partial f}{\partial a_1} \frac{\partial a_1}{\partial x_2} = \overline{a_1} \cos(x_2)
\]
Note that we can also do **forward mode** automatic differentiation.

Start from **inputs** and propagate gradients forward.

Complexity is proportional to input size.

- However, in most cases our **inputs** (images) are large and **outputs** (loss) are small.

Mathematical expression:

\[
\dot{w}_3 = \dot{w}_1 + \dot{w}_2 \\
\dot{w}_1 = \cos(x_1) \dot{x}_1 \\
\dot{w}_2 = \dot{x}_1 x_2 + x_1 \dot{x}_2
\]
A graph is created on the fly

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 20))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 20))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```

(Note above)
Back-propagation uses the dynamically built graph

```python
from torch.autograd import Variable

x = Variable(torch.randn(1, 20))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 20))

i2h = torch.mm(W_x, x.t())
h2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
next_h.backward(torch.ones(1, 20))
```
Convolutional network (AlexNet)
Neural Turing Machine

input image

loss

Figure reproduced with permission from a Twitter post by Andrej Karpathy.
Computation graphs are **not limited to mathematical functions**!

Can have **control flows** (if statements, loops) and **backpropagate** through algorithms!

Can be done **dynamically** so that **gradients are computed**, then **nodes are added**, repeat

**Differentiable programming**

---

**Power of Automatic Differentiation**

Adapted from figure by Andrej Karpathy