Backpropagation + Deep Learning
Reminders

• Homework 4: Logistic Regression
  – Out: Wed, Sep. 25
  – Due: Fri, Oct. 11 at 11:59pm

• Homework 5: Neural Networks
  – Out: Fri, Oct. 11
  – Due: Fri, Oct. 25 at 11:59pm

• Today’s In-Class Poll

• Exam Viewing
<table>
<thead>
<tr>
<th>Q:</th>
<th>Do I need to know Matrix Calculus to derive the backprop algorithms used in this class?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>No. We’ve carefully constructed our assignments so that you do <strong>not</strong> need to know Matrix Calculus.</td>
</tr>
</tbody>
</table>

That said, it’s kind of handy.
Matrix Calculus

<table>
<thead>
<tr>
<th>Types of Derivatives</th>
<th>scalar</th>
<th>vector</th>
<th>matrix</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>$\frac{\partial y}{\partial x}$</td>
<td>$\frac{\partial y}{\partial x}$</td>
<td>$\frac{\partial Y}{\partial x}$</td>
</tr>
<tr>
<td>vector</td>
<td>$\frac{\partial y}{\partial x}$</td>
<td>$\frac{\partial y}{\partial x}$</td>
<td>$\frac{\partial Y}{\partial x}$</td>
</tr>
<tr>
<td>matrix</td>
<td>$\frac{\partial y}{\partial X}$</td>
<td>$\frac{\partial y}{\partial X}$</td>
<td>$\frac{\partial Y}{\partial X}$</td>
</tr>
</tbody>
</table>

Let $y, x \in \mathbb{R}$ be scalars, $y \in \mathbb{R}^M$ and $x \in \mathbb{R}^P$ be vectors, and $Y \in \mathbb{R}^{M \times N}$ and $X \in \mathbb{R}^{P \times Q}$ be matrices.
## Matrix Calculus

<table>
<thead>
<tr>
<th>Types of Derivatives</th>
<th>scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>scalar</strong></td>
<td>$\frac{\partial y}{\partial x} = \left[ \frac{\partial y}{\partial x} \right]$</td>
</tr>
<tr>
<td><strong>vector</strong></td>
<td>$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \ \frac{\partial y}{\partial x_2} \ \vdots \ \frac{\partial y}{\partial x_P} \end{bmatrix}$</td>
</tr>
<tr>
<td><strong>matrix</strong></td>
<td>$\frac{\partial y}{\partial X} = \begin{bmatrix} \frac{\partial y}{\partial X_{11}} &amp; \frac{\partial y}{\partial X_{12}} &amp; \cdots &amp; \frac{\partial y}{\partial X_{1Q}} \ \frac{\partial y}{\partial X_{21}} &amp; \frac{\partial y}{\partial X_{22}} &amp; \cdots &amp; \frac{\partial y}{\partial X_{2Q}} \ \vdots &amp; \vdots &amp; \ddots &amp; \vdots \ \frac{\partial y}{\partial X_{P1}} &amp; \frac{\partial y}{\partial X_{P2}} &amp; \cdots &amp; \frac{\partial y}{\partial X_{PQ}} \end{bmatrix}$</td>
</tr>
</tbody>
</table>
### Matrix Calculus

<table>
<thead>
<tr>
<th>Types of Derivatives</th>
<th>scalar</th>
<th>vector</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>scalar</strong></td>
<td>$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x} \end{bmatrix}$</td>
<td>$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x} &amp; \frac{\partial y_2}{\partial x} &amp; \cdots &amp; \frac{\partial y_N}{\partial x} \end{bmatrix}$</td>
</tr>
<tr>
<td><strong>vector</strong></td>
<td>$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y}{\partial x_1} \ \frac{\partial y}{\partial x_2} \ \vdots \ \frac{\partial y}{\partial x_P} \end{bmatrix}$</td>
<td>$\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} &amp; \frac{\partial y_2}{\partial x_1} &amp; \cdots &amp; \frac{\partial y_N}{\partial x_1} \ \frac{\partial y_1}{\partial x_2} &amp; \frac{\partial y_2}{\partial x_2} &amp; \cdots &amp; \frac{\partial y_N}{\partial x_2} \ \vdots &amp; \vdots &amp; \ddots &amp; \vdots \ \frac{\partial y_1}{\partial x_P} &amp; \frac{\partial y_2}{\partial x_P} &amp; \cdots &amp; \frac{\partial y_N}{\partial x_P} \end{bmatrix}$</td>
</tr>
</tbody>
</table>
Matrix Calculus

Common Vector Derivatives

Let $\frac{df(x)}{dx} = \nabla_x f(x)$ be the vector derivative of $f$, $B \in \mathbb{R}^{m \times m}$, $x \in \mathbb{R}^m$

Scalar Derivative

- $f(x) \Rightarrow \frac{df}{dx}$
- $bx \Rightarrow b$
- $xb \Rightarrow b$
- $x^2 \Rightarrow 2x$
- $bx^2 \Rightarrow 2bx$

Vector Derivative

- $f(x) \Rightarrow \frac{df}{dx}$
- $x^TB \Rightarrow B$
- $x^Tb \Rightarrow b$
- $x^Tx \Rightarrow 2x$
- $x^TBx \Rightarrow 2Bx$
- $B$ symmetric
Question:
Suppose \( y = g(u) \) and \( u = h(x) \)

Which of the following is the correct definition of the chain rule?

Answer:
Recall:

\[
\frac{\partial y}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} \\ \frac{\partial y_2}{\partial x_2} \\ \vdots \\ \frac{\partial y_N}{\partial x_N} \end{bmatrix} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_N}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_N}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_P} & \frac{\partial y_2}{\partial x_P} & \cdots & \frac{\partial y_N}{\partial x_P} \end{bmatrix} \frac{\partial u}{\partial x} = \begin{bmatrix} \frac{\partial y_1}{\partial u} \\ \frac{\partial y_2}{\partial u} \\ \vdots \\ \frac{\partial y_N}{\partial u} \end{bmatrix} \frac{\partial u}{\partial x}
\]

A. \( \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x} \)

B. \( \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x} \)

C. \( \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x} \)

D. \( \frac{\partial y}{\partial x} = \frac{\partial y}{\partial u} \frac{\partial u}{\partial x} \)

E. \( (\frac{\partial y}{\partial u})^T \frac{\partial u}{\partial x} \)

F. None of the above
Algorithm

BACKPROPAGATION
Differentiation Quiz #1:
Suppose $x = 2$ and $z = 3$, what are $dy/dx$ and $dy/dz$ for the function below? Round your answer to the nearest integer.

$$y = \exp(xz) + \frac{xz}{\log(x)} + \frac{\sin(\log(x))}{xz}$$
Backprop Exists

\[ y = \frac{\log(x) + \exp(xz) + xz + \sinh(z)}{x} \]

Forward Computation

- \( x = 2 \), \( z = 3 \)
- \( a = xz \)
- \( b = \log(x) \)
- \( c = \sin(z) \)
- \( d = \exp(a) \)
- \( e = \sqrt{c} \)
- \( f = \frac{e}{d} \)
- \( y = f + c + f \)

Update:

\[ g_y = \frac{\partial y}{\partial y} = 1 \]

\[ g_x = \frac{\partial y}{\partial x} = \frac{e}{d} + \frac{\partial f}{\partial f} \frac{\partial f}{\partial x} \]

\[ g_z = \frac{\partial y}{\partial z} = \frac{\partial f}{\partial f} \frac{\partial f}{\partial z} + \frac{\partial f}{\partial e} \frac{\partial e}{\partial z} - g_e \frac{\partial z}{\partial z} + g_e \sinh(z) \]

\[ g_x = g_x(x) + g_x(\frac{1}{z}) \]
Chalkboard

- SGD for Neural Network
- Example: Backpropagation for Neural Network
Automatic Differentiation – Reverse Mode (aka. Backpropagation)

Forward Computation
1. Write an **algorithm** for evaluating the function \( y = f(x) \). The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the “**computation graph**”)
2. Visit each node in **topological order**.
   For variable \( u_i \) with inputs \( v_1, \ldots, v_N \)
   - a. Compute \( u_i = g_i(v_1, \ldots, v_N) \)
   - b. Store the result at the node

Backward Computation (Version A)
1. **Initialize** \( \frac{dy}{dy} = 1 \).
2. Visit each node \( v_j \) in **reverse topological order**.
   Let \( u_1, \ldots, u_M \) denote all the nodes with \( v_j \) as an input
   Assuming that \( y = h(u) = h(u_1, \ldots, u_M) \)
   and \( u = g(v) \) or equivalently \( u_i = g_i(v_1, \ldots, v_j, \ldots, v_N) \) for all \( i \)
   - a. We already know \( \frac{dy}{du_i} \) for all \( i \)
   - b. Compute \( \frac{dy}{dv_j} \) as below (Choice of algorithm ensures computing \( (\frac{du_i}{dv_j}) \) is easy)
   \[
   \frac{dy}{dv_j} = \sum_{i=1}^{M} \frac{dy}{du_i} \cdot \frac{du_i}{dv_j}
   \]

Return partial derivatives \( \frac{dy}{du_i} \) for all variables
Automatic Differentiation – Reverse Mode (aka. Backpropagation)

Forward Computation
1. Write an **algorithm** for evaluating the function \( y = f(x) \). The algorithm defines a **directed acyclic graph**, where each variable is a node (i.e. the “**computation graph**”)
2. Visit each node in **topological order**.
   For variable \( u_i \) with inputs \( v_1, \ldots, v_N \)
   a. Compute \( u_i = g_i(v_1, \ldots, v_N) \)
   b. Store the result at the node

Backward Computation (Version B)
1. **Initialize** all partial derivatives \( \frac{dy}{du_j} \) to 0 and \( \frac{dy}{dy} = 1 \).
2. Visit each node in **reverse topological order**.
   For variable \( u_i = g_i(v_1, \ldots, v_N) \)
   a. We already know \( \frac{dy}{du_i} \)
   b. Increment \( \frac{dy}{dv_j} \) by \( (\frac{dy}{du_i})(\frac{du_i}{dv_j}) \)
      \( \text{(Choice of algorithm ensures computing} \ (\frac{du_i}{dv_j}) \text{is easy)} \)

**Return** partial derivatives \( \frac{dy}{du_i} \) for all variables
Why is the backpropagation algorithm efficient?

1. Reuses **computation from the forward pass** in the backward pass

2. Reuses **partial derivatives** throughout the backward pass (*but only if the algorithm reuses shared computation in the forward pass*)

(Key idea: partial derivatives in the backward pass should be thought of as variables stored for reuse)
Algorithm 1: Stochastic Gradient Descent (SGD)

1: procedure SGD(Training data $\mathcal{D}$, test data $\mathcal{D}_t$)
2:   Initialize parameters $\alpha, \beta$
3:   for $e \in \{1, 2, \ldots, E\}$ do
4:     for $(x, y) \in \mathcal{D}$ do
5:         Compute neural network layers:
6:         $o = \text{object}(x, a, b, z, \hat{y}, J) = \text{NNFORWARD}(x, y, \alpha, \beta)$
7:     Compute gradients via backprop:
8:         $g_\alpha = \nabla_\alpha J$
9:         $g_\beta = \nabla_\beta J$ = $\text{NNBACKWARD}(x, y, \alpha, \beta, o)$
10:    Update parameters:
11:       $\alpha \leftarrow \alpha - \gamma g_\alpha$
12:       $\beta \leftarrow \beta - \gamma g_\beta$
13:    Evaluate training mean cross-entropy $J_D(\alpha, \beta)$
14:    Evaluate test mean cross-entropy $J_{D_t}(\alpha, \beta)$
15: return parameters $\alpha, \beta$
Simple Example: The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

Forward

$J = \cos(u)$

$u = u_1 + u_2$

$u_1 = \sin(t)$

$u_2 = 3t$

$t = x^2$
Simple Example: The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

<table>
<thead>
<tr>
<th>Forward</th>
<th>Backward</th>
</tr>
</thead>
<tbody>
<tr>
<td>$J = \cos(u)$</td>
<td>$\frac{dJ}{du} = \cos(u)$</td>
</tr>
<tr>
<td>$u = u_1 + u_2$</td>
<td>$\frac{dJ}{du_1} = \frac{dJ}{du}, \quad \frac{du}{du_1} = 1$</td>
</tr>
<tr>
<td>$u_1 = \sin(t)$</td>
<td>$\frac{dJ}{dt} = \frac{dJ}{du_1} \frac{du_1}{dt}, \quad \frac{du_1}{dt} = \cos(t)$</td>
</tr>
<tr>
<td>$u_2 = 3t$</td>
<td>$\frac{dJ}{dt} = \frac{dJ}{du_2} \frac{du_2}{dt}, \quad \frac{du_2}{dt} = 3$</td>
</tr>
<tr>
<td>$t = x^2$</td>
<td>$\frac{dJ}{dx} = \frac{dJ}{dt} \frac{dt}{dx}, \quad \frac{dt}{dx} = 2x$</td>
</tr>
</tbody>
</table>
Case 1: Logistic Regression

Forward

\[ J = y^* \log y + (1 - y^*) \log(1 - y) \]

\[ y = \frac{1}{1 + \exp(-a)} \]

\[ a = \sum_{j=0}^{D} \theta_j x_j \]

Backward

\[ \frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1} \]

\[ \frac{dJ}{da} = \frac{dJ}{dy} \frac{dy}{da}, \quad \frac{dy}{da} = \frac{\exp(-a)}{(\exp(-a) + 1)^2} \]

\[ \frac{dJ}{d\theta_j} = \frac{dJ}{da} \frac{da}{d\theta_j}, \quad \frac{da}{d\theta_j} = x_j \]

\[ \frac{dJ}{dx_j} = \frac{dJ}{da} \frac{da}{dx_j}, \quad \frac{da}{dx_j} = \theta_j \]
Training

Backpropagation

\( J = \frac{1}{2} (y - y_d)^2 \)

\( y = \frac{1}{1 + e^{-b}} \)

\( b = \sum_{j=0}^{D} \beta_j z_j \)

\( z_j = \frac{1}{1 + e^{-a_j}}, \forall j \)

\( a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \forall j \)

\( x_i, \forall i \)
Training

Backpropagation

\[
\text{(F) Loss} \\
J = \frac{1}{2}(y - y^*)^2
\]

\[
\text{(E) Output (sigmoid)} \\
y = \frac{1}{1+\exp(-b)}
\]

\[
\text{(D) Output (linear)} \\
b = \sum_{j=0}^{D} \beta_j z_j
\]

\[
\text{(C) Hidden (sigmoid)} \\
z_j = \frac{1}{1+\exp(-a_j)}, \quad \forall j
\]

\[
\text{(B) Hidden (linear)} \\
a_j = \sum_{i=0}^{M} \alpha_{ji} x_i, \quad \forall j
\]

\[
\text{(A) Input} \\
\text{Given } x_i, \quad \forall i
\]
Case 2: Neural Network

Forward
\[ J = y^* \log y + (1 - y^*) \log(1 - y) \]
\[ y = \frac{1}{1 + \exp(-b)} \]
\[ b = \sum_{j=0}^{D} \beta_j z_j \]
\[ z_j = \frac{1}{1 + \exp(-a_j)} \]
\[ a_j = \sum_{i=0}^{M} \alpha_{ji} x_i \]

Backward
\[ \frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1} \]
\[ \frac{dJ}{db} = \frac{dJ}{dy} \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2} \]
\[ \frac{dJ}{d\beta_j} = \frac{dJ}{db} \frac{db}{d\beta_j} = z_j \]
\[ \frac{dJ}{dz_j} = \frac{dJ}{db} \frac{db}{dz_j} = \beta_j \]
\[ \frac{dJ}{da_j} = \frac{dJ}{dz_j} \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2} \]
\[ \frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j} \frac{da_j}{d\alpha_{ji}} = x_i \]
\[ \frac{dJ}{dx_i} = \sum_{j=0}^{D} \frac{dJ}{da_j} \frac{da_j}{dx_i} = \alpha_{ji} \]
### Case 2: Neural Network

#### Forward

- **Loss**
  \[ J = y^* \log y + (1 - y^*) \log(1 - y) \]

- **Sigmoid**
  \[ y = \frac{1}{1 + \exp(-b)} \]

- **Linear**
  \[ b = \sum_{j=0}^{D} \beta_j z_j \]

- **Sigmoid**
  \[ z_j = \frac{1}{1 + \exp(-a_j)} \]

- **Linear**
  \[ a_j = \sum_{i=0}^{M} \alpha_{ji} x_i \]

#### Backward

- **Loss**
  \[ \frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1} \]

- **Sigmoid**
  \[ \frac{dJ}{db} = \frac{dJ}{dy} \frac{dy}{db}, \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2} \]

- **Linear**
  \[ \frac{dJ}{d\beta_j} = \frac{dJ}{db} \frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j \]

  \[ \frac{dJ}{dz_j} = \frac{dJ}{db} \frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j \]

- **Sigmoid**
  \[ \frac{dJ}{da_j} = \frac{dJ}{dz_j} \frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2} \]

- **Linear**
  \[ \frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j} \frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i \]

  \[ \frac{dJ}{dx_i} = \sum_{j=0}^{D} \frac{dJ}{da_j} \frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \alpha_{ji} \]
First suppose that

\[ s = \frac{1}{1 + \exp(-b)} \]  \hspace{1cm} (1)

To obtain the simplified form of the derivative of a sigmoid.

\[ \frac{ds}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2} \]  \hspace{1cm} (2)

\[ = \frac{\exp(-b) + 1 - 1}{(\exp(-b) + 1 + 1 - 1)^2} \]  \hspace{1cm} (3)

\[ = \frac{\exp(-b) + 1 - 1}{(\exp(-b) + 1)^2} \]  \hspace{1cm} (4)

\[ = \frac{\exp(-b) + 1}{(\exp(-b) + 1)^2} - \frac{1}{(\exp(-b) + 1)^2} \]  \hspace{1cm} (5)

\[ = \frac{1}{(\exp(-b) + 1)} - \frac{1}{(\exp(-b) + 1)^2} \]  \hspace{1cm} (6)

\[ = \frac{1}{(\exp(-b) + 1)} - \left( \frac{1}{(\exp(-b) + 1)} \frac{1}{(\exp(-b) + 1)} \right) \]  \hspace{1cm} (7)

\[ = \frac{1}{(\exp(-b) + 1)} \left( 1 - \frac{1}{(\exp(-b) + 1)} \right) \]  \hspace{1cm} (8)

\[ = s(1 - s) \]  \hspace{1cm} (9)
### Case 2: Neural Network

#### Forward

**Loss**

\[ J = y^* \log y + (1 - y^*) \log(1 - y) \]

**Sigmoid**

\[ y = \frac{1}{1 + \exp(-b)} \]

\[ b = \sum_{j=0}^{D} \beta_j z_j \]

**Sigmoid**

\[ z_j = \frac{1}{1 + \exp(-a_j)} \]

**Linear**

\[ a_j = \sum_{i=0}^{M} \alpha_{ji} x_i \]

#### Backward

\[ \frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1} \]

\[ \frac{dJ}{db} = \frac{dJ}{dy} \frac{dy}{db} \]

\[ \frac{dy}{db} = \frac{\exp(-b)}{(\exp(-b) + 1)^2} \]

\[ \frac{dJ}{d\beta_j} = \frac{dJ}{db} \frac{db}{d\beta_j}, \quad \frac{db}{d\beta_j} = z_j \]

\[ \frac{dJ}{dz_j} = \frac{dJ}{db} \frac{db}{dz_j}, \quad \frac{db}{dz_j} = \beta_j \]

\[ \frac{dJ}{da_j} = \frac{dJ}{dz_j} \frac{dz_j}{da_j}, \quad \frac{dz_j}{da_j} = \frac{\exp(-a_j)}{(\exp(-a_j) + 1)^2} \]

\[ \frac{dJ}{dx_i} = \sum_{j=0}^{D} \frac{dJ}{da_j} \frac{da_j}{dx_i}, \quad \frac{dJ}{dx_i} = x_i \]

\[ \frac{dJ}{dx_i} = \sum_{j=0}^{D} \frac{dJ}{da_j} \frac{da_j}{dx_i}, \quad \frac{dJ}{dx_i} = \alpha_{ji} \]
Case 2:

**Forward**

- **Loss**
  \[ J = y^* \log y + (1 - y^*) \log(1 - y) \]

- **Sigmoid**
  \[ y = \frac{1}{1 + \exp(-b)} \]

- **Linear**
  \[ b = \sum_{j=0}^{D} \beta_j z_j \]

- **Sigmoid**
  \[ z_j = \frac{1}{1 + \exp(-a_j)} \]

- **Linear**
  \[ a_j = \sum_{i=0}^{M} \alpha_{ji} x_i \]

**Backward**

- **Loss**
  \[ \frac{dJ}{dy} = \frac{y^*}{y} + \frac{(1 - y^*)}{y - 1} \]

- **Sigmoid**
  \[ \frac{dJ}{db} = \frac{dy}{db} - \frac{dy}{db} = y(1 - y) \]

- **Linear**
  \[ \frac{dJ}{d\beta_j} = \frac{dJ}{db} \frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j \]

- **Sigmoid**
  \[ \frac{dJ}{dz_j} = \frac{dJ}{db} \frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j \]

- **Linear**
  \[ \frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j} \frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i \]

\[ \frac{dJ}{dx_i} = \sum_{j=0}^{D} \frac{dJ}{da_j} \frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \alpha_{ji} \]
Backpropagation

Training

Automatic Differentiation – Reverse Mode (aka. Backpropagation)

**Forward Computation**
1. Write an **algorithm** for evaluating the function $y = f(x)$. The algorithm defines a directed acyclic graph, where each variable is a node (i.e. the “computation graph”)
2. Visit each node in **topological order**. For variable $u_i$ with inputs $v_1, \ldots, v_N$
   a. Compute $u_i = g_i(v_1, \ldots, v_N)$
   b. Store the result at the node

**Backward Computation (Version A)**
1. **Initialize** $dy/du = 1$.
2. Visit each node $v_j$ in **reverse topological order**. Let $u_1, \ldots, u_M$ denote all the nodes with $v_j$ as an input
   Assuming that $y = h(u) = h(u_1, \ldots, u_M)$ and $u = g(v)$ or equivalently $u_i = g_i(v_1, \ldots, v_j, \ldots, v_N)$ for all $i$
   a. We already know $dy/du_i$ for all $i$
   b. Compute $dy/dv_j$ as below (Choice of algorithm ensures computing $(du_i/dv_j)$ is easy)

\[
\frac{dy}{dv_j} = \sum_{i=1}^{M} \frac{dy}{du_i} \frac{du_i}{dv_j}
\]

**Return** partial derivatives $dy/du_i$ for all variables
Automatic Differentiation – Reverse Mode (aka. Backpropagation)

Forward Computation
1. Write an algorithm for evaluating the function \( y = f(x) \). The algorithm defines a directed acyclic graph, where each variable is a node (i.e. the “computation graph”)
2. Visit each node in topological order.
   For variable \( u_i \) with inputs \( v_1, \ldots, v_N \)
   a. Compute \( u_i = g_i(v_1, \ldots, v_N) \)
   b. Store the result at the node

Backward Computation (Version B)
1. Initialize all partial derivatives \( \frac{dy}{du_j} \) to 0 and \( \frac{dy}{dy} = 1 \).
2. Visit each node in reverse topological order.
   For variable \( u_i = g_i(v_1, \ldots, v_N) \)
   a. We already know \( \frac{dy}{du_i} \)
   b. Increment \( \frac{dy}{dv_j} \) by \( (\frac{dy}{du_i})(\frac{du_i}{dv_j}) \)
   (Choice of algorithm ensures computing \( \frac{du_i}{dv_j} \) is easy)

Return partial derivatives \( \frac{dy}{du_i} \) for all variables
Example: 1-Hidden Layer Neural Network

Algorithm 1: Stochastic Gradient Descent (SGD)

1: procedure SGD(Training data $\mathcal{D}$, test data $\mathcal{D}_t$)
2: Initialize parameters $\alpha, \beta$
3: for $e \in \{1, 2, \ldots, E\}$ do
4:   for $(x, y) \in \mathcal{D}$ do
5:     Compute neural network layers:
6:     $o = \text{object}(x, a, b, z, \hat{y}, J) = \text{NNFORWARD}(x, y, \alpha, \beta)$
7:     Compute gradients via backprop:
8:     $g_\alpha = \nabla_\alpha J$
9:     $g_\beta = \nabla_\beta J$
10: return $\text{NNBACKWARD}(x, y, \alpha, \beta, o)$
11: Update parameters:
12:    $\alpha \leftarrow \alpha - \gamma g_\alpha$
13:    $\beta \leftarrow \beta - \gamma g_\beta$
14: Evaluate training mean cross-entropy $J_D(\alpha, \beta)$
15: Evaluate test mean cross-entropy $J_{D_t}(\alpha, \beta)$
16: return parameters $\alpha, \beta$
1. Given training data:
\[ \{x_i, y_i\}_{i=1}^{N} \]

2. Choose each of the following:
- Decision function
  \[ \hat{y} = f_{\theta}(x_i) \]
- Loss function
  \[ \ell(\hat{y}, y_i) \in \mathbb{R} \]

3. Define goal:

4. Train with SGD:
   \[ \theta^{(t)} \leftarrow \theta^{(t)} - \eta t \nabla \ell(f_{\theta}(x_i), y_i) \]

**Gradients**

**Backpropagation** can compute this gradient!

And it’s a **special case of a more general algorithm** called reverse-mode automatic differentiation that can compute the gradient of any differentiable function efficiently!
OTHER APPROACHES TO DIFFERENTIATION
The centered finite difference approximation is:

\[
\frac{\partial}{\partial \theta_i} J(\theta) \approx \frac{(J(\theta + \epsilon \cdot d_i) - J(\theta - \epsilon \cdot d_i))}{2\epsilon}
\]  

where \(d_i\) is a 1-hot vector consisting of all zeros except for the \(i\)th entry of \(d_i\), which has value 1.

Notes:

- Suffers from issues of floating point precision, in practice
- Typically only appropriate to use on small examples with an appropriately chosen epsilon
Differentiation Quiz #1:
Suppose $x = 2$ and $z = 3$, what are $dy/dx$ and $dy/dz$ for the function below? Round your answer to the nearest integer.

$$y = \exp(xz) + \frac{xz}{\log(x)} + \frac{\sin(\log(x))}{xz}$$

**Answer:** Answers below are in the form $[dy/dx, dy/dz]$

A. [42, -72]  
B. [72, -42]  
C. [100, 127]  
D. [127, 100]  
E. [1208, 810]  
F. [810, 1208]  
G. [1505, 94]  
H. [94, 1505]
Differentiation Quiz #2:
A neural network with 2 hidden layers can be written as:

\[ y = \sigma(\beta^T \sigma((\alpha^{(2)})^T \sigma((\alpha^{(1)})^T x))) \]

where \( y \in \mathbb{R}, x \in \mathbb{R}^{D^{(0)}}, \beta \in \mathbb{R}^{D^{(2)}} \) and \( \alpha^{(i)} \) is a \( D^{(i)} \times D^{(i-1)} \) matrix. Nonlinear functions are applied elementwise:

\[ \sigma(a) = [\sigma(a_1), \ldots, \sigma(a_K)]^T \]

Let \( \sigma \) be sigmoid: \( \sigma(a) = \frac{1}{1+e^{x-p-a}} \)

What is \( \frac{\partial y}{\partial \beta_j} \) and \( \frac{\partial y}{\partial \alpha_{j}^{(i)}} \) for all \( i, j \).
Summary

1. **Neural Networks**
   - provide a way of learning features
   - are highly nonlinear prediction functions
   - (can be) a highly parallel network of logistic regression classifiers
   - discover useful hidden representations of the input

2. **Backpropagation**
   - provides an efficient way to compute gradients
   - is a special case of reverse-mode automatic differentiation
Backprop Objectives

You should be able to...

• Construct a computation graph for a function as specified by an algorithm
• Carry out the backpropagation on an arbitrary computation graph
• Construct a computation graph for a neural network, identifying all the given and intermediate quantities that are relevant
• Instantiate the backpropagation algorithm for a neural network
• Instantiate an optimization method (e.g. SGD) and a regularizer (e.g. L2) when the parameters of a model are comprised of several matrices corresponding to different layers of a neural network
• Apply the empirical risk minimization framework to learn a neural network
• Use the finite difference method to evaluate the gradient of a function
• Identify when the gradient of a function can be computed at all and when it can be computed efficiently
DEEP LEARNING
Deep Learning Outline

• **Background: Computer Vision**
  – Image Classification
  – ILSVRC 2010 - 2016
  – Traditional Feature Extraction Methods
  – Convolution as Feature Extraction

• **Convolutional Neural Networks (CNNs)**
  – Learning Feature Abstractions
  – Common CNN Layers:
    • Convolutional Layer
    • Max-Pooling Layer
    • Fully-connected Layer (w/tensor input)
    • Softmax Layer
    • ReLU Layer
  – Background: Subgradient
  – Architecture: LeNet
  – Architecture: AlexNet

• **Training a CNN**
  – SGD for CNNs
  – Backpropagation for CNNs
Why is everyone talking about Deep Learning?

• Because a lot of money is invested in it...
  – DeepMind: Acquired by Google for $400 million
  – DNNResearch: Three person startup (including Geoff Hinton) acquired by Google for unknown price tag
  – Enlitic, Ersatz, MetaMind, Nervana, Skylab: Deep Learning startups commanding millions of VC dollars

• Because it made the front page of the New York Times
Why is everyone talking about Deep Learning?

Deep learning:

– Has won numerous pattern recognition competitions
– Does so with minimal feature engineering

This wasn’t always the case!
Since 1980s: Form of models hasn’t changed much, but lots of new tricks...
– More hidden units
– Better (online) optimization
– New nonlinear functions (ReLUs)
– Faster computers (CPUs and GPUs)
BACKGROUND: COMPUTER VISION
Example: Image Classification

• ImageNet LSVRC-2011 contest:
  – **Dataset**: 1.2 million labeled images, 1000 classes
  – **Task**: Given a new image, label it with the correct class
  – **Multiclass** classification problem
• Examples from http://image-net.org/
Bird

Warm-blooded egg-laying vertebrates characterized by feathers and forelimbs modified as wings

- marine animal, marine creature, sea animal, sea creature (1)
- scavenger (1)
- biped (0)
- predator, predatory animal (1)
- larva (49)
- acrodont (0)
- feeder (0)
- sturt (0)
- chordate (3087)
  - tunicate, urochordate, urochord (6)
  - cephalochordate (1)
  - vertebrate, craniate (3077)
    - mammal, mammalian (1169)
  - bird (871)
    - dickybird, dicky-bird, dickybird, dicky-bird (0)
    - cock (1)
    - hen (0)
    - nester (0)
    - night bird (1)
    - bird of passage (0)
    - protoavis (0)
    - archaeopteryx, archaeopteryx, Archaeopteryx lithographi
      - Sinornis (0)
    - ibero-mesornis (0)
    - archaeornis (0)
    - ratite, ratite bird, flightless bird (10)
    - carinate, carinate bird, flying bird (0)
    - passerine, passeriform bird (279)
    - nonpasserine bird (0)
    - bird of prey, raptor, raptorial bird (80)
    - gallinaceous bird, gallinaceous (114)
German iris, *Iris kochii*

Iris of northern Italy having deep blue-purple flowers; similar to but smaller than *Iris germanica*

- halophyte (0)
- succulent (39)
- cultivar (0)
- cultivated plant (0)
- weed (54)
- evergreen, evergreen plant (0)
- deciduous plant (0)
- vine (272)
- creeper (0)
- woody plant, ligneous plant (1868)
- geophyte (0)
- desert plant, xerophyte, xerophytic plant, xerophile, xerophilic mesophyte, mesophytic plant (0)
- aquatic plant, water plant, hydrophyte, hydrophytic plant (11)
- tuberous plant (0)
- bulbous plant (179)
- iridaceous plant (27)
  - iris, flag, fleur-de-lis, sword lily (19)
    - bearded iris (4)
      - Florentine iris, orris, *Iris germanica* florentina, *Iris germanica* (0)
      - German iris, *Iris germanica* (0)
      - German iris, *Iris kochii* (0)
      - Dalmatian iris, *Iris pallida* (0)
    - beardless iris (4)
      - bulbous iris (0)
      - dwarf iris, *Iris cristata* (0)
      - stinking iris, gladdon, gladdon iris, stinking gladwyn, Persian iris, *Iris persica* (0)
      - yellow iris, *Iris pseudacorus* (0)
      - dwarf iris, vernal iris, *Iris verna* (0)
      - blue flag, *Iris versicolor* (0)
Court, courtyard

An area wholly or partly surrounded by walls or buildings; "the house was built around an inner court"
Feature Engineering for CV

Edge detection (Canny)

Corner Detection (Harris)

Scale Invariant Feature Transform (SIFT)

Figures from http://opencv.org

Figure from Lowe (1999) and Lowe (2004)
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

The first convolutional layer takes as input the raw pixel values of the 224×224×3 image and filters it with 48 kernels of size 5×5×3. The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size 5×5×48.

The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers. The third convolutional layer has 384 kernels of size 3×3×256 connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size 3×3×192, and the fifth convolutional layer has 256 kernels of size 3×3×192. The fully-connected layers have 4096 neurons each.

4 Reducing Overfitting

Our neural network architecture has 60 million parameters. Although the 1000 classes of ILSVRC make each training example impose 10 bits of constraint on the mapping from image to label, this turns out to be insufficient to learn so many parameters without considerable overfitting. Below, we describe the two primary ways in which we combat overfitting.

4.1 Data Augmentation

The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations (e.g., [25, 4, 5]). We employ two distinct forms of data augmentation, both of which allow transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk. In our implementation, the transformed images are generated in Python code on the CPU while the GPU is training on the previous batch of images. So these data augmentation schemes are, in effect, computationally free.

The first form of data augmentation consists of generating image translations and horizontal reflections. We do this by extracting random 224×224 patches (and their horizontal reflections) from the 256×256 images and training our network on these extracted patches. This increases the size of our training set by a factor of 2048, though the resulting training examples are, of course, highly interdependent. Without this scheme, our network suffers from substantial overfitting, which would have forced us to use much smaller networks. At test time, the network makes a prediction by extracting five 224×224 patches (the four corner patches and the center patch) as well as their horizontal reflections (hence ten patches in all), and averaging the predictions made by the network's softmax layer on the ten patches.

The second form of data augmentation consists of altering the intensities of the RGB channels in training images. Specifically, we perform PCA on the set of RGB pixel values throughout the ImageNet training set. To each training image, we add multiples of the found principal components.

This is the reason why the input images in Figure 2 are 224×224×3-dimensional.
CNNs for Image Recognition

革命深度

深度 ImageNet 分类 top-5 错误率（%）

ILSVRC'15 ResNet 3.57
ILSVRC'14 GoogleNet 6.7
ILSVRC'14 VGG 7.3
ILSVRC'13 11.7
ILSVRC'12 AlexNet 16.4
ILSVRC'11 25.8
ILSVRC'10 28.2


滑动来自 Kaiming He
CONVOLUTION
What’s a convolution?

• Basic idea:
  – Pick a 3x3 matrix $F$ of weights
  – Slide this over an image and compute the “inner product” (similarity) of $F$ and the corresponding field of the image, and replace the pixel in the center of the field with the output of the inner product operation

• Key point:
  – Different convolutions extract different types of low-level “features” from an image
  – All that we need to vary to generate these different features is the weights of $F$

---

**Example:** 1 input channel, 1 output channel

![Convolution Example](image)

$y_{11} = \alpha_{11} x_{11} + \alpha_{12} x_{12} + \alpha_{21} x_{21} + \alpha_{22} x_{22} + \alpha_0$

$y_{12} = \alpha_{11} x_{12} + \alpha_{12} x_{13} + \alpha_{21} x_{22} + \alpha_{22} x_{23} + \alpha_0$

$y_{21} = \alpha_{11} x_{21} + \alpha_{12} x_{22} + \alpha_{21} x_{31} + \alpha_{22} x_{32} + \alpha_0$

$y_{22} = \alpha_{11} x_{22} + \alpha_{12} x_{23} + \alpha_{21} x_{32} + \alpha_{22} x_{33} + \alpha_0$

---

Slide adapted from William Cohen
A **convolution matrix** is used in image processing for tasks such as edge detection, blurring, sharpening, etc.
Background: Image Processing

A **convolution matrix** is used in image processing for tasks such as edge detection, blurring, sharpening, etc.

**Input Image**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Convolution**

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

**Convolved Image**

<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Background: Image Processing

A **convolution matrix** is used in image processing for tasks such as edge detection, blurring, sharpening, etc.

Input Image

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Convolution

<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
A convolution matrix is used in image processing for tasks such as edge detection, blurring, sharpening, etc.
Background: Image Processing

A **convolution matrix** is used in image processing for tasks such as edge detection, blurring, sharpening, etc.

**Input Image**

```
0 0 0 0 0 0 0 0
0 1 1 1 1 1 0 0
0 1 0 0 1 0 0 0
0 1 0 1 0 0 0 0
0 1 1 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

**Convolution**

```
0 0 0 0 0 0 0 0
0 3 2 2 3 1 0 0
0 2 0 2 1 0 0 0
0 2 2 1 0 0 0 0
0 3 1 0 0 0 0 0
0 1 0 0 0 0 0 0
```

**Convolved Image**
Background: Image Processing

A **convolution matrix** is used in image processing for tasks such as edge detection, blurring, sharpening, etc.

![Convolution Matrix Illustration]

- **Input Image**
- **Convolution**
- **Convolved Image**
Background: Image Processing

A convolution matrix is used in image processing for tasks such as edge detection, blurring, sharpening, etc.
Background: Image Processing

A **convolution matrix** is used in image processing for tasks such as edge detection, blurring, sharpening, etc.
A **convolution matrix** is used in image processing for tasks such as edge detection, blurring, sharpening, etc.
Background: Image Processing

A **convolution matrix** is used in image processing for tasks such as edge detection, blurring, sharpening, etc.

<table>
<thead>
<tr>
<th>Input Image</th>
<th>Convolved Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0</td>
<td>3 2 2 3 1</td>
</tr>
<tr>
<td>0 1 1 1</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 1</td>
<td></td>
</tr>
<tr>
<td>0 1 1 0</td>
<td></td>
</tr>
<tr>
<td>0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

Convolution
A convolution matrix is used in image processing for tasks such as edge detection, blurring, sharpening, etc.
Background: Image Processing

A convolution matrix is used in image processing for tasks such as edge detection, blurring, sharpening, etc.

**Input Image**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Convolution**

**Convolved Image**

<table>
<thead>
<tr>
<th>3</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A **convolution matrix** is used in image processing for tasks such as edge detection, blurring, sharpening, etc.
Background: Image Processing

A **convolution matrix** is used in image processing for tasks such as edge detection, blurring, sharpening, etc.

<table>
<thead>
<tr>
<th>Input Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 1 1 1 1 1 0 0</td>
</tr>
<tr>
<td>0 1 0 0 1 0 0 0</td>
</tr>
<tr>
<td>0 1 0 1 0 0 0 0</td>
</tr>
<tr>
<td>0 1 1 0 0 0 0 0</td>
</tr>
<tr>
<td>0 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Identity Convolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0</td>
</tr>
<tr>
<td>0 1 0</td>
</tr>
<tr>
<td>0 0 0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Convoluted Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 1 1 1 1 1 1 1</td>
</tr>
<tr>
<td>1 0 0 1 1 0 0 0</td>
</tr>
<tr>
<td>1 0 1 0 0 0 0 0</td>
</tr>
<tr>
<td>1 1 0 0 0 0 0 0</td>
</tr>
<tr>
<td>1 0 0 0 0 0 0 0</td>
</tr>
</tbody>
</table>
Background: Image Processing

A convolution matrix is used in image processing for tasks such as edge detection, blurring, sharpening, etc.

Input Image

![Input Image](image)

Convolved Image

![Convolved Image](image)

Blurring Convolution

![Blurring Convolution](image)
What’s a convolution?

http://matlabtricks.com/post-5/3x3-convolution-kernels-with-online-demo
What’s a convolution?

http://matlabtricks.com/post-5/3x3-convolution-kernels-with-online-demo
What’s a convolution?

http://matlabtricks.com/post-5/3x3-convolution-kernels-with-online-demo
What’s a convolution?

http://matlabtricks.com/post-5/3x3-convolution-kernels-with-online-demo
What’s a convolution?

http://matlabtricks.com/post-5/3x3-convolution-kernels-with-online-demo
What’s a convolution?

http://matlabtricks.com/post-5/3x3-convolution-kernels-with-online-demo

Slide from William Cohen
What’s a convolution?

• Basic idea:
  – Pick a 3x3 matrix $F$ of weights
  – Slide this over an image and compute the “inner product” (similarity) of $F$ and the corresponding field of the image, and replace the pixel in the center of the field with the output of the inner product operation

• Key point:
  – Different convolutions extract different types of low-level “features” from an image
  – All that we need to vary to generate these different features is the weights of $F$
Downsampling

- Suppose we use a convolution with stride 2
- Only 9 patches visited in input, so only 9 pixels in output

Input Image

```
1 1 1 1 1 1 0
1 0 0 1 0 0 0
1 0 1 0 0 0 0
1 1 0 0 0 0 0
1 1 0 0 0 0 0
1 0 0 0 0 0 0
0 0 0 0 0 0 0
```

Convolution

```
1 1
1 1
```

Convolved Image
Downsampling

- Suppose we use a convolution with stride 2
- Only 9 patches visited in input, so only 9 pixels in output
Downsampling

• Suppose we use a convolution with stride 2
• Only 9 patches visited in input, so only 9 pixels in output
Downsampling

• Suppose we use a convolution with stride 2
• Only 9 patches visited in input, so only 9 pixels in output
Downsampling

- Suppose we use a convolution with stride 2
- Only 9 patches visited in input, so only 9 pixels in output
Downsampling

- Suppose we use a convolution with stride 2
- Only 9 patches visited in input, so only 9 pixels in output
Downsampling

• Suppose we use a convolution with stride 2
• Only 9 patches visited in input, so only 9 pixels in output

\[
\begin{array}{ccccccc}
1 & 1 & 1 & 1 & 1 & 0 \\
1 & 0 & 0 & 1 & 0 & 0 \\
1 & 0 & 1 & 0 & 0 & 0 \\
1 & 1 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{cccc}
3 & 3 & 1 \\
3 & 1 & 0 \\
\end{array}
\]
Downsampling

• Suppose we use a convolution with stride 2
• Only 9 patches visited in input, so only 9 pixels in output
Downsampling

- Suppose we use a convolution with stride 2
- Only 9 patches visited in input, so only 9 pixels in output
Downsampling

• Suppose we use a convolution with stride 2
• Only 9 patches visited in input, so only 9 pixels in output
CONVOLUTIONAL NEURAL NETS
1. Given training data: \[ \{ x_i, y_i \}_{i=1}^{N} \]

2. Choose each of these:
   - Decision function
     \[ \hat{y} = f_\theta(x_i) \]
   - Loss function
     \[ \ell(\hat{y}, y_i) \in \mathbb{R} \]

3. Define goal:
   \[ \theta^* = \arg\min_{\theta} \sum_{i=1}^{N} \ell(f_\theta(x_i), y_i) \]

4. Train with SGD:
   (take small steps opposite the gradient)
   \[ \theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla \ell(f_\theta(x_i), y_i) \]
1. Given training data:

2. Choose each of these:

   - Decision function
   - Loss function

3. Define goal:

4. Train with SGD:
   
   \[ \theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla \ell(f_\theta(x_i), y_i) \]

- Convolutional Neural Networks (CNNs) provide another form of **decision function**
- Let’s see what they look like…
Convolutional Neural Network (CNN)

- Typical layers include:
  - Convolutional layer
  - Max-pooling layer
  - Fully-connected (Linear) layer
  - ReLU layer (or some other nonlinear activation function)
  - Softmax
- These can be arranged into arbitrarily deep topologies

Architecture #1: LeNet-5

Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.
Convolutional Layer

**CNN key idea:**
Treat convolution matrix as parameters and learn them!

<table>
<thead>
<tr>
<th>Input Image</th>
<th>Learned Convolution</th>
<th>Convolved Image</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td>θ_{11} θ_{12} θ_{13}</td>
<td>.4 .5 .5 .5 .4</td>
</tr>
<tr>
<td>0 1 1 1 1 1 0 0</td>
<td>θ_{21} θ_{22} θ_{23}</td>
<td>.4 .2 .3 .6 .3</td>
</tr>
<tr>
<td>0 1 0 0 1 0 0 0</td>
<td>θ_{31} θ_{32} θ_{33}</td>
<td>.5 .4 .4 .2 .1</td>
</tr>
<tr>
<td>0 1 0 1 0 0 0 0</td>
<td></td>
<td>.5 .6 .2 .1 0</td>
</tr>
<tr>
<td>0 1 1 0 0 0 0 0</td>
<td></td>
<td>.4 .3 .1 0 0</td>
</tr>
<tr>
<td>0 1 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0 0 0 0 0 0 0 0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Downsampling by Averaging

- Downsampling by averaging **used to be** a common approach
- This is a special case of convolution where the weights are fixed to a uniform distribution
- The example below uses a stride of 2

**Input Image**

```
1 1 1 1 1 0
1 0 0 1 0 0
1 0 1 0 0 0
1 1 0 0 0 0
1 0 0 0 0 0
0 0 0 0 0 0
```

**Convolution**

```
1/4 1/4
1/4 1/4
```

**Convolved Image**

```
3/4 3/4 1/4
3/4 1/4 0
1/4 0 0
```
Max-Pooling

- Max-pooling is another (common) form of downsampling
- Instead of averaging, we take the max value within the same range as the equivalently-sized convolution
- The example below uses a stride of 2

\[ y_{ij} = \max(x_{ij}, x_{i,j+1}, x_{i+1,j}, x_{i+1,j+1}) \]
TRAINING CNNS
A Recipe for Machine Learning

1. Given training data:
\[ \{x_i, y_i\}_{i=1}^N \]

2. Choose each of these:
   - Decision function
     \[ \hat{y} = f_{\theta}(x_i) \]
   - Loss function
     \[ \ell(\hat{y}, y_i) \in \mathbb{R} \]

3. Define goal:
\[ \theta^* = \arg \min_{\theta} \sum_{i=1}^{N} \ell(f_{\theta}(x_i), y_i) \]

4. Train with SGD:
   (take small steps opposite the gradient)
\[ \theta^{(t+1)} = \theta^{(t)} - \eta_t \nabla \ell(f_{\theta}(x_i), y_i) \]
1. Given training data:
\[ \{x_i, y_i\}_{i=1}^{N} \]

2. Choose each of these:
   - Decision function
   \[ \hat{y} = f_{\theta}(x_i) \]

   - Loss function
   \[ \ell(\hat{y}, y_i) \in \mathbb{R} \]

3. Define goal:
\[ \sum_{i=1}^{N} N(y_i) \]

4. Train with SGD:
   (take small steps opposite the gradient)
\[ \theta(t+1) = \theta(t) - \eta_t \nabla \ell(f_{\theta}(x_i), y_i) \]

Q: Now that we have the CNN as a decision function, how do we compute the gradient?

A: Backpropagation of course!
SGD for CNNs

**Example Architecture:**

Given $\hat{x}, y^*$

\[
J = l(y, y^*)
\]

\[
y = \text{softmax}(z^{(5)})
\]

\[
z^{(5)} = \text{linear}(z^{(4)}, W)
\]

\[
z^{(4)} = \text{relu}(z^{(3)})
\]

\[
z^{(3)} = \text{conv}(z^{(2)}, \beta)
\]

\[
z^{(2)} = \text{max-pool}(z^{(1)})
\]

\[
z^{(1)} = \text{conv}(\hat{x}, \alpha)
\]

**Parameters:**

$\hat{\theta} = [\alpha, \beta, W]$

**SGD:**

1. **Initialize $\hat{\theta}$**
2. **While not converged:**
   - Sample $i \in \{1, ..., N\}$
   - **Forward:** $Y = h_{\theta}(x^{(i)})$, $J_i(\theta) = l(y, y^*)$
   - **Backward:** $\nabla_\theta J_i(\theta) = ...$
   - **Update:** $\hat{\theta} \leftarrow \hat{\theta} - \gamma \nabla_\theta J_i(\theta)$
LAYERS OF A CNN
Common CNN Layers

Whiteboard

- ReLU Layer
- Background: Subgradient
- Fully-connected Layer (w/tensor input)
- Softmax Layer
- Convolutional Layer
- Max-Pooling Layer
ReLU Layer

Input: \( \hat{x} \in \mathbb{R}^k \)  
Output: \( \hat{y} \in \mathbb{R}^k \)

**Forward:**  
\[
\hat{y} = \sigma(\hat{x}) \quad \text{element-wise}
\]
\[
\sigma(a) = \max(0, a)
\]

**Backward:**  
\[
\frac{dJ}{dx_i} = \frac{dJ}{dy_i} \frac{dy_i}{dx_i}
\]

where \( \frac{dy_i}{dx_i} = \begin{cases} 
1 & \text{if } x_i > 0 \\
0 & \text{otherwise}
\end{cases} \)
Softmax Layer

Input: $\hat{x} \in \mathbb{R}^k$
Output: $\hat{y} \in \mathbb{R}^k$

Forward:
$y_i = \frac{\exp(x_i)}{\sum_{k=1}^{K} \exp(x_k)}$

Backward:
$\frac{dS}{dx_j} = \sum_{i=1}^{K} \frac{dS}{dy_i} \frac{dy_i}{dx_j}$

where $\frac{dy_i}{dx_j} = \begin{cases} y_i(1-y_i) & \text{if } i=j \\ -y_i y_j & \text{otherwise} \end{cases}$
Fully-Connected Layer

- Suppose input is a 3D Tensor: $X = X_{(c \times h \times w)}$

- Stretch out into a long vector: $X = [x_1, \ldots, x_{(c \times h \times w)}]$

- Then standard linear layer:

  $y = \alpha^T x + \alpha_0$ where $\alpha \in \mathbb{R}^{A \times B}$

  $|x| = A, |y| = B$
Convolutional Layer

**Ex. 1 input channel, 1 output channel**

<table>
<thead>
<tr>
<th>Input</th>
<th>Conv</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{11}$, $x_{12}$, $x_{13}$</td>
<td>$\alpha_{11}$, $\alpha_{12}$</td>
<td>$y_{11}$, $y_{12}$</td>
</tr>
<tr>
<td>$x_{21}$, $x_{22}$, $x_{23}$</td>
<td>$\alpha_{21}$, $\alpha_{22}$</td>
<td>$y_{21}$, $y_{22}$</td>
</tr>
<tr>
<td>$x_{31}$, $x_{32}$, $x_{33}$</td>
<td></td>
<td>$y_{31}$, $y_{32}$</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
y_{11} &= \alpha_{11} x_{11} + \alpha_{12} x_{12} + \alpha_{21} x_{21} + \alpha_{22} x_{22} + \alpha_0 \\
y_{12} &= \alpha_{11} x_{12} + \alpha_{12} x_{13} + \alpha_{21} x_{22} + \alpha_{22} x_{23} + \alpha_0 \\
y_{21} &= \alpha_{11} x_{21} + \alpha_{12} x_{22} + \alpha_{21} x_{31} + \alpha_{22} x_{32} + \alpha_0 \\
y_{22} &= \alpha_{11} x_{22} + \alpha_{12} x_{23} + \alpha_{21} x_{32} + \alpha_{22} x_{33} + \alpha_0
\end{align*}
\]

**Ex. 1 input channel, 2 output channels**

<table>
<thead>
<tr>
<th>Input</th>
<th>Conv #1</th>
<th>Output #1</th>
<th>Conv #2</th>
<th>Output #2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_{11}$, $x_{12}$, $x_{13}$</td>
<td>$\alpha_{11}$, $\alpha_{12}$</td>
<td>$y_{11}^{(1)}$, $y_{12}^{(1)}$</td>
<td>$\alpha_{11}$, $\alpha_{12}$</td>
<td>$y_{11}^{(2)}$, $y_{12}^{(2)}$</td>
</tr>
<tr>
<td>$x_{21}$, $x_{22}$, $x_{23}$</td>
<td>$\alpha_{21}$, $\alpha_{22}$</td>
<td>$y_{21}^{(1)}$, $y_{22}^{(1)}$</td>
<td>$\alpha_{21}$, $\alpha_{22}$</td>
<td>$y_{21}^{(2)}$, $y_{22}^{(2)}$</td>
</tr>
<tr>
<td>$x_{31}$, $x_{32}$, $x_{33}$</td>
<td></td>
<td>$y_{31}^{(1)}$, $y_{32}^{(1)}$</td>
<td></td>
<td>$y_{31}^{(2)}$, $y_{32}^{(2)}$</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
y_{11}^{(1)} &= \alpha_{11} x_{11} + \alpha_{12} x_{12} + \alpha_{21} x_{21} + \alpha_{22} x_{22} + \alpha_0 \\
y_{12}^{(1)} &= \alpha_{11} x_{12} + \alpha_{12} x_{13} + \alpha_{21} x_{22} + \alpha_{22} x_{23} + \alpha_0 \\
y_{21}^{(1)} &= \alpha_{11} x_{21} + \alpha_{12} x_{22} + \alpha_{21} x_{31} + \alpha_{22} x_{32} + \alpha_0 \\
y_{22}^{(1)} &= \alpha_{11} x_{22} + \alpha_{12} x_{23} + \alpha_{21} x_{32} + \alpha_{22} x_{33} + \alpha_0 \\
y_{11}^{(2)} &= \alpha_{11} x_{11} + \alpha_{12} x_{12} + \alpha_{21} x_{21} + \alpha_{22} x_{22} + \alpha_0 \\
y_{12}^{(2)} &= \alpha_{11} x_{12} + \alpha_{12} x_{13} + \alpha_{21} x_{22} + \alpha_{22} x_{23} + \alpha_0 \\
y_{21}^{(2)} &= \alpha_{11} x_{21} + \alpha_{12} x_{22} + \alpha_{21} x_{31} + \alpha_{22} x_{32} + \alpha_0 \\
y_{22}^{(2)} &= \alpha_{11} x_{22} + \alpha_{12} x_{23} + \alpha_{21} x_{32} + \alpha_{22} x_{33} + \alpha_0
\end{align*}
\]
Convolutional Layer

Example: \( C^I \) input channels, \( C^O \) output channels

Input \( \rightarrow \) Conv #1 \( \rightarrow \) Output

- The slice is output from \( i \)th convolution matrix.
- \( H^O = \left( H^I + 2p - K \right) / s + 1 \)
- \( W^O = \left( W^I + 2p - K \right) / s + 1 \)

where
- \( p \) = # pixels of padding on input
- \( K \) = size of conv. matrix
- \( s \) = stride length

Forward:

\[
y_{ij}^{(k)} = x_0^{(k)} + \sum_{c=1}^{C^I} \sum_{q=1}^{K} \sum_{r=1}^{K} x_r^{(k)} X_{mn}^{(c)} \quad \text{where} \quad m = s(i-1) + q, \quad n = s(j-1) + r
\]

Backward:

\[
\frac{dJ}{dw_0^{(k)}} = \sum_i \sum_j \frac{dJ}{dy_{ij}^{(k)}} \frac{dy_{ij}^{(k)}}{dalpha_0^{(k)}}
\]

\[
\frac{dJ}{dx_r^{(k)}} = \sum_i \sum_j \frac{dJ}{dy_{ij}^{(k)}} \frac{dy_{ij}^{(k)}}{dalpha_r^{(k)}}
\]

\[
\frac{dJ}{dx_{mn}^{(c)}} = \sum_i \sum_j \frac{dJ}{dy_{ij}^{(k)}} \frac{dy_{ij}^{(k)}}{dx_{mn}^{(c)}}
\]

Just save calculations.
Max-Pooling Layer

Ex: 1 input channel, 1 output channel, stride of 1

\[
\begin{align*}
\text{Input} & : \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix} \\
\text{Pool Size} & : 2x2 \\
\text{Output} & : \begin{bmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{bmatrix} \\
\end{align*}
\]

\[
\begin{align*}
y_{11} &= \max \left( x_{11}, \ x_{12}, \ x_{21}, \ x_{22} \right) \\
y_{12} &= \max \left( x_{12}, \ x_{13}, \ x_{22}, \ x_{23} \right) \\
y_{21} &= \max \left( x_{21}, \ x_{22}, \ x_{31}, \ x_{32} \right) \\
y_{22} &= \max \left( x_{22}, \ x_{23}, \ x_{32}, \ x_{33} \right) \\
\end{align*}
\]
Max-Pooling Layer

Forward:
\[ Y_{ij}^{(k)} = \max_{q \in \{1, \ldots, k^3\}} X_{m,n}^{(k)} \]
where \( m = s(i-1) + q \), \( n = s(j-1) + r \).

Backward:
\[ \frac{dJ}{dx_{mn}^{(k)}} = \sum_{i,j} \frac{dJ}{dy_{ij}^{(k)}} \frac{dy_{ij}^{(k)}}{dx_{mn}^{(k)}} \]

Subderivatives:

- Max() is not differentiable, but subdifferentiable.
- There are a set of derivatives and we can just choose one for SGD.

\[ y = \max(a,b) \]
\[ \Rightarrow \frac{dJ}{da} = \frac{dJ}{dy} \frac{dy}{da} \]
where \( \frac{dy}{da} = \begin{cases} 1 & \text{if } a > b \\ 0 & \text{otherwise} \end{cases} \)
Convolutional Neural Network (CNN)

- Typical layers include:
  - Convolutional layer
  - Max-pooling layer
  - Fully-connected (Linear) layer
  - ReLU layer (or some other nonlinear activation function)
  - Softmax

- These can be arranged into arbitrarily deep topologies

**Architecture #1: LeNet-5**

Fig. 2. Architecture of LeNet-5, a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network’s input is 150,528-dimensional, and the number of neurons in the network’s remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

The second convolutional layer takes as input the (response-normalized and pooled) output of the first convolutional layer and filters it with 256 kernels of size $5 \times 5 \times 48$. The third, fourth, and fifth convolutional layers are connected to one another without any intervening pooling or normalization layers. The third convolutional layer has 384 kernels of size $3 \times 3 \times 256$ connected to the (normalized, pooled) outputs of the second convolutional layer. The fourth convolutional layer has 384 kernels of size $3 \times 3 \times 192$, and the fifth convolutional layer has 256 kernels of size $3 \times 3 \times 192$. The fully-connected layers have 4096 neurons each.

4 Reducing Overfitting

Our neural network architecture has 60 million parameters. Although the 1000 classes of ILSVRC make each training example impose 10 bits of constraint on the mapping from image to label, this turns out to be insufficient to learn so many parameters without considerable overfitting. Below, we describe the two primary ways in which we combat overfitting.

4.1 Data Augmentation

The easiest and most common method to reduce overfitting on image data is to artificially enlarge the dataset using label-preserving transformations (e.g., [25, 4, 5]). We employ two distinct forms of data augmentation, both of which allow transformed images to be produced from the original images with very little computation, so the transformed images do not need to be stored on disk. In our implementation, the transformed images are generated in Python code on the CPU while the GPU is training on the previous batch of images. So these data augmentation schemes are, in effect, computationally free.

The first form of data augmentation consists of generating image translations and horizontal reflections. We do this by extracting random $224 \times 224$ patches (and their horizontal reflections) from the $256 \times 256$ images and training our network on these extracted patches. This increases the size of our training set by a factor of 2048, though the resulting training examples are, of course, highly interdependent. Without this scheme, our network suffers from substantial overfitting, which would have forced us to use much smaller networks. At test time, the network makes a prediction by extracting five $224 \times 224$ patches (the four corner patches and the center patch) as well as their horizontal reflections (hence ten patches in all), and averaging the predictions made by the network’s softmax layer on the ten patches.

The second form of data augmentation consists of altering the intensities of the RGB channels in training images. Specifically, we perform PCA on the set of RGB pixel values throughout the ImageNet training set. To each training image, we add multiples of the found principal components.

4 This is the reason why the input images in Figure 2 are $224 \times 224$-dimensional.
CNNs for Image Recognition

Revolution of Depth

ImageNet Classification top-5 error (%)

CNN VISUALIZATIONS
3D Visualization of CNN

http://scs.ryerson.ca/~aharley/vis/conv/
Convolution of a Color Image

- Color images consist of 3 floats per pixel for RGB (red, green blue) color values
- Convolution must also be 3-dimensional

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations

activation map

Figure from Fei-Fei Li & Andrej Karpathy & Justin Johnson (CS231N)
Animation of 3D Convolution

http://cs231n.github.io/convolutional-networks/

Figure from Fei-Fei Li & Andrej Karpathy & Justin Johnson (CS231N)
MNIST Digit Recognition with CNNs (in your browser)

https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html
CNN Summary

CNNs

– Are used for all aspects of computer vision, and have won numerous pattern recognition competitions
– Able learn interpretable features at different levels of abstraction
– Typically, consist of convolution layers, pooling layers, nonlinearities, and fully connected layers

Other Resources:

– Readings on course website
– Andrej Karpathy, CS231n Notes
  http://cs231n.github.io/convolutional-networks/