Lecture 19
Deep Learning and Computer Vision
Who Am I?

• Hi, I’m Shubhang Desai
  o BS + MS from Stanford CS
  o Moved to Seattle ~2.5 years ago

• Applied Scientist at Microsoft, on Ink AI Team
  o Spearheaded deep learning handwriting recognizer (HWR)
  o Working on HWR, ink analysis, Copilot features
  o Both image and sequence modelling tasks

• Passionate about teaching
  o CS 131 (Comp. Vision), CS 230 (Deep Learning), CS 21SI (AI + Social Good) @ Stanford
  o CSE 493G1 (Deep Learning) @ UW
Plan for Today

• Deep dive on backpropagation

• Convolution-based classification algorithm

• Deep convolutional neural networks

• Vision transformers
Backpropagation Deep Dive
Refresher: Gradient Descent

Iteratively moving neural network weights in the direction of the gradient to minimize loss:

How do we actually compute this?
Refresher: Chain Rule

“Derivative of outside of inside equals derivative of outside times derivative of inside”

\[
\frac{d}{dx} f(g(x)) = \frac{d}{dg(x)} f(g(x)) \times \frac{d}{dx} g(x)
\]
2-Layer MLP

\[ x_1 \rightarrow z_1 \rightarrow a_1 \rightarrow f \rightarrow L \]

\[ x_2 \rightarrow z_1 \rightarrow a_1 \rightarrow f \rightarrow L \]

\[ x_3 \rightarrow z_2 \rightarrow a_2 \rightarrow f \rightarrow L \]

\[ x_4 \rightarrow z_2 \rightarrow a_2 \rightarrow f \rightarrow L \]
\[ \hat{y} = w_2 \sigma(w_1 x) \text{ and } L = f(\hat{y}) \]
2-Layer MLP Equations and Gradients

\[ L = f(\hat{y}), \text{ where } \hat{y} = w_2 A, \ A = \sigma(z), \text{ and } z = w_1 x \]

\[ \frac{dL}{dw_1} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dw_1} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dA} \frac{dA}{dw_1} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dA} \frac{dA}{dz} \frac{dz}{dw_1} \]

\[ \frac{dL}{dw_2} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dw_2} \]

Notice that this value is used in both gradient computations!
\[ \hat{y} = w_2 \sigma(w_1 x) \text{ and } L = f(\hat{y}) \]

\[
\frac{dL}{dw_1} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dA} \frac{dA}{dz} \frac{dz}{dw_1} \\
\frac{dL}{dw_2} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dw_2}
\]
The Backpropagation Algorithm

• Treat entire network as a computational graph, each computation as a node

• We can independently compute local gradient at each node given node inputs

• Accumulate gradients from back (loss) to front (weights) using chain rule (simple multiplication!)
Summation

\[ z = \sum_i x_i, L = f(Z) \]

\[ \frac{\partial z}{\partial x_i} = 1 \]

\[ \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x_i} = \frac{\partial L}{\partial z} \]
Multiplication

\[ z = \prod_{i} x_i, L = f(Z) \]
\[ \frac{\partial z}{\partial x_i} = \frac{z}{x_i} \]
\[ \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x_i} = \frac{\partial L}{\partial z} \frac{z}{x_i} \]
Min/Max

\[ z = \max_i x_i, L = f(Z) \]
\[ \frac{\partial z}{\partial x_i} = \mathbf{1}[x_i = z] \]
\[ \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x_i} = \frac{\partial L}{\partial z} \mathbf{1}[x_i = z] \]

Diagram:
- \( x_1 = 4 \)
- \( x_2 = 3 \)
- \( x_3 = 0 \)
- \( \frac{dL}{dz} \)
- \( \max \)
Sigmoid

\[ z = \sigma(x), L = f(Z) \]
\[ \frac{\partial z}{\partial x_i} = z(1 - z) \]
\[ \frac{\partial L}{\partial x_i} = \frac{\partial L}{\partial z} \frac{\partial z}{\partial x_i} = \frac{\partial L}{\partial z} z(1 - z) \]
\[ L = (x + y)z \]

\[
\frac{dL}{dz} = (1)(3) = 3 \\
\frac{dL}{dy} = (1)(-4)(1) = -4 \\
\frac{dL}{dx} = (1)(-4)(1) = -4
\]
\( \hat{y} = w_2 \sigma(w_1 x) \) and \( L = f(\hat{y}) \)
\[ L = f(w_2\sigma(w_1x + b_1) + b_2) \quad \text{and} \quad f = ||\hat{y} - y||^2 \]

- \( \frac{dL}{db_2} = 2(\hat{y} - y) \)
- \( \frac{dL}{dw_2} = 2(\hat{y} - y)A^T \)
- \( \frac{dL}{db_1} = \left(2w_2^T(\hat{y} - y)\right)(A)(1 - A) \)
- \( \frac{dL}{dw_1} = \left(2w_2^T(\hat{y} - y)\right)(A)(1 - A)x^T \)
Convolution-Based Classifier
Recall convolutions…

\[
f[n, m] \ast h[n, m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k, l] h[n - k, m - l]
\]
Recall convolutions...

\[ f[n, m] \ast h[n, m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k, l] h[n - k, m - l] \]
Recall convolutions...

\[ f[n, m] \ast h[n, m] = \sum_{k=\infty}^{\infty} \sum_{l=\infty}^{\infty} f[k, l] h[n-k, m-l] \]
Recall convolutions...

\[
f[n, m] \ast h[n, m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k, l] h[n - k, m - l]
\]
Recall convolutions...

\[ f[n, m] \ast h[n, m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k, l] h[n - k, m - l] \]
Recall convolutions...

\[ f[n, m] \ast h[n, m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k, l] h[n - k, m - l] \]
Why they are useful

Allow us to find **interesting insights/features** from images!
Recall Image Classification…

Allow us to use features to put *images in categories*!
Convolution-Based Classification System

Convolution = Image -> Features

Classification Algorithm = Features -> Category
Convolution-Based Classification System

Convolution = Image -> Features

Classification Algorithm = Features -> Category

Can we put them together?
Convolution + Learned Linear Layer

Input Image → 3x3 Conv Op → flatten → Linear Classifier → argmax $c_{\hat{y}}$

Input Label $\gamma$ → Classification Output $\hat{y}$ → Loss Function $CE$ → Loss Value $L$
Convolution + Learned Linear Layer

What convolution filter should we use?

- Input Image
- Feature Extractor
- 3x3 Conv Op
- Flatten
- Linear Classifier
- Prediction \( \hat{y} \)
- Classification Output
- Loss Function \( CE \)
- Loss Value \( L \)
- Input Label
- argmax
- \( C_x \)
Can we learn it like we learn the linear layer?

(Learned?) Convolution + Learned Linear Layer

Input Image → 3x3 Conv Op → flatten → Linear Classifier → argmax

Input Label → Prediction $\hat{y}$ → Classification Output

Loss Function → $CE$ → Loss Value

$L$
Calculating Local Gradients

When computations are treated as nodes, all derivatives depend only on inputs to that node.

So, we can cache the initial computation and reuse!
Backprop Graph to Layer-Based Pseudocode

```python
class Linear:
    def __init__(self):
        self.cache = {}

    def forward(self, W, x):
        self.cache['x'] = x
        return np.dot(W, x)

    def backward(self, dout):
        x = self.cache['x']
        return np.matmul(dout, x.T)

class SCELoss:
    def __init__(self):
        self.cache = {}

    def forward(self, z, y):
        self.cache['z'] = z
        self.cache['y'] = y
        return sce(z, y)

    def backward(self):
        z = self.cache['z']
        y = self.cache['y']
        return sm(z) - y
```

Shubhang Desai, Ranjay Krishna, Jieyu Zhang
Model Definition + Training Pseudocode

model = Sequential([Linear, SCELoss])

for i in {0,...,num_epochs}:
    for X, y in data:
        L = model.forward(X, y)
        gradients = model.backprop(L)
        model.update_weights(gradients)
How about a learnable conv layer right here?!

```python
model = Sequential([Linear, SCELoss])

for i in {0,...,num_epochs}:
    for X, y in data:
        L = model.forward(X)
        gradients = model.backprop(L)
        model.update_weights(gradients)
```
It’s Just a Linear Layer in a Double For-Loop!

\[
f[n, m] \ast h[n, m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k, l] h[n - k, m - l]
\]
Conv Layer Pseudocode

class Conv:
    def __init__(self):
        self.cache = {}

    def forward(self, W, x):
        self.cache['x'] = x
        out = np.zeros(out_shape)
        for i in range(out.shape[0]):
            for j in range(out.shape[1]):
                out[i, j] = np.dot(W, x[i-3:i+3, j-3:j+3])

    def backward(self, dout):
        x = self.cache['x']
        d = np.zeros(W_shape)

        for i in range(dout.shape[0]):
            for j in range(dout.shape[1]):
                d += dout[i, j] * x[i-3:i+3, j-3:j+3]
Adding Conv Layer to Model Definition

```python
layers = [Linear, SCELoss]
model = Sequential(layers)

for i in {0,...,num_epochs}:
    for x, y in data:
        L = model.forward(X)
        gradients = model.backprop(L)
        model.update_weights(gradients)
```
Adding Conv Layer to Model Definition

```python
layers = [Conv, Linear, SCELoss]
model = Sequential(layers)

for i in {0,...,num_epochs}:
    for x, y in data:
        L = model.forward(x)
        gradients = model.backprop(L)
        model.update_weights(gradients)
```
Learned Convolution + Learned Linear Layer

Input Image → 3x3 Conv Layer → flatten → Linear Classifier → argmax $c_\hat{y}$ → Classification Output

Input Label $y$ → $CE$ → $L$ → Loss Function → Loss Value
Deep Convolutional Networks
Stacking Conv Layers in Model Definition

Why just one convolutional layer?

```
layers = [Conv, Linear, SCELoss]
model = Sequential(layers)

for i in {0,...,num_epochs}:
    for x, y in data:
        L = model.forward(x)
        gradients = model.backprop(L)
        model.update_weights(gradients)
```
Stacking Conv Layers in Model Definition

```
layers = [Conv, Conv, Conv, Linear, SCELoss]
model = Sequential(layers)

for i in {0,...,num_epochs}:
    for x, y in data:
        L = model.forward(X)
        gradients = model.backprop(L)
        model.update_weights(gradients)
```
Stacking Conv Layers in Model Definition

And how much control do we have over what happens in each conv layer?

```python
layers = [Conv, Conv, Conv, Linear, SCELoss]
model = Sequential(layers)

for i in {0,...,num_epochs}:
    for x, y in data:
        L = model.forward(X)
        gradients = model.backprop(L)
        model.update_weights(gradients)
```
Right Now: One Filter Per Conv Layer

28×28×3 Image

15×15×3 Filter

14×14×1 Output
New Idea: Multiple Filters Per Conv Layer!

28×28×3 Image

15×15×3×4 Filter

14×14×4 Output
New Idea: Multiple Filters Per Conv Layer!

more output channels
= more filters
= more features we can learn!
Simplified Notation

28 × 28 × 3 → 15 × 15 × 3 × 4 Conv → 14 × 14 × 4
Stacking Wide Convolutions

5 × 5 × 3 × 4 Conv Block

15 × 15 × 4 × 6 Conv Block

8 × 8 × 6 × 8 Conv Block

Flatten + 392 × 10 Linear Layer

32×32×3 Input

28×28×4 Output

14×14×6 Output

7×7×8 Output

1 × 10 Output
Multiple Filters in Conv Layer Pseudocode

class Conv:
    def __init__(self):
        self.cache = {}

    def forward(self, W, x):
        self.cache['x'] = x
        out = np.zeros(out_shape)
        for i in range(out.shape[0]):
            for j in range(out.shape[1]):
                out[i, j] = np.dot(W, x[i-3:i+3, j-3:j+3])

    def backward(self, dout):
        x = self.cache['x']
        d = np.zeros(W_shape)

        for i in range(dout.shape[0]):
            for j in range(dout.shape[1]):
                d += dout[i, j] * x[i-3:i+3, j-3:j+3]
Multiple Filters in Conv Layer Pseudocode

class Conv:
    def __init__(self):
        self.cache = {}

    def forward(self, W, x):
        self.cache['x'] = x
        out = np.zeros(out_shape)
        for i in range(out.shape[0]):
            for j in range(out.shape[1]):
                for f in range(out.shape[2]):
                    out[i, j, f] = np.dot(W[f], x[i-3:i+3, j-3:j+3])

    def backward(self, dout):
        x = self.cache['x']
        d = np.zeros(W_shape)

        for i in range(dout.shape[0]):
            for j in range(dout.shape[1]):
                for f in range(dout.shape[2]):
                    d[f] += dout[i, j, f] * x[i-3:i+3, j-3:j+3]
Fully-Convolutional Neural Network

5\times 5 \times 3 \times 4
Conv Block

28\times 28 \times 4
Output

15\times 15 \times 4 \times 6
Conv Block

14\times 14 \times 6
Output

8\times 8 \times 6 \times 8
Conv Block

7\times 7 \times 8
Output

7\times 7 \times 8 \times 10
Conv Block

1\times 1 \times 10
Output
Fully-Convolutional Neural Network

```python
layers = [Conv, Conv, Conv, Linear, SCELoss]
model = Sequential(layers)

for i in {0,...,num_epochs}:
    for x, y in data:
        L = model.forward(X)
        gradients = model.backprop(L)
        model.update_weights(gradients)
```
Fully-Convolutional Neural Network

```
layers = [Conv, Conv, Conv, Conv, SCELoss]
model = Sequential(layers)

for i in {0,...,num_epochs}:
    for x, y in data:
        L = model.forward(x)
        gradients = model.backprop(L)
        model.update_weights(gradients)
```
History of ConvNets

LeNet – 1998
History of ConvNets

AlexNet – 2012
History of ConvNets

NiN – 2013
History of ConvNets

Inception Network – 2015
Vision Transformers
Self-Attention: An Alternative to Convolution

- Until now, we have been using convolution as the operation to gain deeper understanding of images
  - Good theoretical grounding: inspired by filters in classical CV
  - Efficient, stackable, and maintains spatial relationships

- Another operation which can efficiently learn spatial features is called self-attention
  - First applied to NLP tasks, eventually applied to CV
  - At each layer, we learn alignments between image patches i.e. how “relevant” is one patch to another
  - We can stack self-attn as as featurizer for arbitrary downstream CV tasks
Self-Attention Background: Sentiment Classifier

Input Sentence: 

what a great movie

Feature Extractor:
embedding

Linear Classifier:
average

Loss Function:
CE

Classification Output:
argmax $C_{\hat{y}}$

Input Label:

$\gamma$

Prediction $\hat{y}$

Loss Value:
$L$

what a great movie
Self-Attention Background: Sentiment Classifier

What (learnable) operation should we put here?

Input Sentence

what a
great
movie

embedding

Feature
Extractor

average

Linear
Classifier

argmax

Classification
Output

Prediction \( \hat{y} \)

\( L \)

\( CE \)

Input Label

Output Sentence

Loss Value

Loss Function

what a
great
movie

Shubhang Desai, Ranjay Krishna, Jieyu Zhang

Lecture 19 - 63

May 28, 2024
Visual Intuition of Self-Attention

Input Sequence: what a movie

Input Vectors:
- w
- a
- m

Self-Attn Weights:
- [a b c]
- [d e f]
- [h i j]

Matmul w/ Input Vecs

Output Vectors:
- w
- a
- m
Visual Intuition of Self-Attention (Reality)

Input Sequence: what a movie

Query Vectors:
- w
- a
- m

Key Vectors:
- w
- a
- m

Value Vectors:
- w
- a
- m

Self-Attn Weights:
- Matmul w/ Key Vecs
- Matmul w/ Value Vecs

Output Vectors:
- w
- a
- m

\[ \begin{array}{ccc}
    \text{Matmul w/ Value Vecs} & \rightarrow & \text{Output Vectors} \\
    \begin{bmatrix}
        w & a & m
    \end{bmatrix} & \rightarrow & \begin{bmatrix}
        w & a & m
    \end{bmatrix}
\end{array} \]

\[ \begin{array}{ccc}
    \text{Matmul w/ Key Vecs} & \rightarrow & \text{Output Vectors} \\
    \begin{bmatrix}
        w & a & m
    \end{bmatrix} & \rightarrow & \begin{bmatrix}
        w & a & m
    \end{bmatrix}
\end{array} \]
Transformer: Stack of Self-Attn (+more) Layers
Vision Transformer: Self-Attn on Img Patches
CLIP: Learning Multi-Model Embedding Space

1. Contrastive pre-training

2. Create dataset classifier from label text

3. Use for zero-shot prediction
Summary

• The popular method to do so in deep learning is backpropagation algorithm
  o Greedily use chain rule to accumulate local gradients from “back to front”

• We can use this principle to create computation “nodes” (layers), including a learnable convolutional layer
  o Foundational idea: create a classifier with a filter-based featurizer

• Convolution is not the only operation in our playbook: we can use an operation borrowed from NLP called self-attention to build a Transformer