CAP 4453
Robot Vision
Dr. Gonzalo Vaca-Castaño
gonzalo.vacacastano@ucf.edu
Administrative details

• 1 homework
• 1 project
Credits

• Some slides come directly from:
  • Yosesh Rawat
  • Andrew Ng
Robot Vision

18. Convolutional Neural Networks I
Fully connected networks: Review
A REVIEW

Fully connected Neural network
A.K.A Multi-Layer Perceptron (MLP)

• A deep network is a neural network with many layers
• A neuron in a linear function followed for an activation function
• Activation function must be non-linear
• A loss function measures how close is the created function (network) from a desired output
• The “training” is the process of find parameters (‘weights’) that reduces the loss functions
• Updating the weights as \( w_{\text{new}} = w_{\text{prev}} - \alpha \frac{dJ}{dW} \) reduces the loss
• An algorithm named back-propagation allows to compute \( \frac{dJ}{dW} \) for all the weights of the network in 2 steps: 1 forward, 1 backward
Exercise

\[ \text{Expected output: 1} \]

\[ \text{Div} = \frac{(\hat{y}_i - y_i)^2}{2} \]

\[ - (\hat{y}_i - y_i^{[N]}) \]
Digit classification

• MNIST dataset:
  • 70000 grayscale images of digits scanned.
  • 60000 for training
  • 10000 for testing

• Loss function

\[ J_2(w) = \frac{1}{m} \sum_{\text{train}} (\hat{y}_i - y_i)^2 \]
Digit classification

Typical Problem statement: multiclass classification

- Given, many positive and negative examples (training data),
  - learn all weights such that the network does the desired job
A look in the code

To run this code do:

- import network
- net = network.Network([784, 30, 10])
- net.SGD(training_data, 30, 10, 3.0, test_data=test_data)
A look in code

```
For k = N−1
For l = 1: layer − width
    \[ \frac{\partial \text{Div}}{\partial y_l} = \frac{\partial \text{Div}(Y, d)}{\partial y_l} \]
    \[ \frac{\partial \text{Div}}{\partial y_{l(k)}} = f'_l(x(k)) \frac{\partial \text{Div}}{\partial y_l} \]
    \[ \frac{\partial \text{Div}}{\partial w_{ij}^{(k)}} = y_{l(k)} \frac{\partial \text{Div}}{\partial y_{l(k)}} \]

Initialize: Gradient w.r.t network output
```

```python
def backprop(self, x, y):
    """Return a tuple `(nabla_b, nabla_w)` representing the gradient for the cost function C_X, i.e.,`nabla_b` and `nabla_w` are layer-by-layer lists of `numpy` arrays, similar to `self.biases` and `self.weights`.
    """
    nabla_b = [np.zeros(b.shape) for b in self.biases]
    nabla_w = [np.zeros(w.shape) for w in self.weights]
    # backward pass
    activation = x
    activations = [x] # list to store all the activations, layer by layer
    zs = [] # list to store all the z vectors, layer by layer
    for b, w in zip(self.biases, self.weights):
        z = np.dot(w, activation) + b
        zs.append(z)
        activation = sigmoid(z)
        activations.append(activation)
    # backward pass
    delta = self.cost_derivative(activations[-1], y) * 
          sigmoid_prime(zs[-1])
    nabla_b[-1] = delta
    nabla_w[-1] = np.dot(delta, activations[-1].transpose())
    # Note that the variable l in the loop below is used a little differently to the notation in Chapter 2 of the book. Here, l = 1 means the last layer of neurons, l−1 is the second last layer, and so on. It's a numbering of the scheme in the book, used here to take advantage of the fact that Python can use negative indices in lists.
    for l in xrange(2, self.num_layers):
        zs[-l] = zs[-l-1]
        sp = sigmoid_prime(zs[-l])
        delta = np.dot(delta, sigmoid_prime(zs[-l-1].transpose())) * sp
        nabla_b[-l] = delta
        nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
    return (nabla_b, nabla_w)
```
A look in code
A look in the code

random Initialization

Feed forward ‘a’ thru all the layers

A Epoch is when all the training data has been used to update weights

A minibatch is a subset of all the data used to obtain a ‘quick’ weight updates

If there is test data perform evaluation
A look in the code

```python
def update_mini_batch(self, mini_batch, eta):
    """Update the network's weights and biases by applying
    gradient descent using backpropagation to a single mini batch.
    The 'mini_batch' is a list of tuples '((x, y)', 'mba' is the learning rate.'"
    nablaba = [np.zeros(b.shape) for b in self.biases]
    nablawa = [np.zeros(w.shape) for w in self.weights]
    for x, y in mini_batch:
        delta_nablaba, delta_nablawa = self.backprop(x, y)
        nablaba = [nb + db for nb, db in zip(nablaba, delta_nablaba)]
        nablawa = [nw + dw for nw, dw in zip(nablawa, delta_nablawa)]
    self.weights = [w-(eta/len(mini_batch))*nw
                    for w, nw in zip(self.weights, nablawa)]
    self.biases = [b-(eta/len(mini_batch))*nb
                   for b, nb in zip(self.biases, nablaba)]

def evaluate(self, test_data):
    """Return the number of test inputs for which the neural
    network outputs the correct result. Note that the neural
    network's output is assumed to be the index of whichever
    neuron in the final layer has the highest activation.'""
    test_results = [(np.argmax(self.feedforward(x)), y) for (x, y) in test_data]
    return sum(int(x == y) for (x, y) in test_results)
```

Add errors from all the training data from the mini-batch

Update the weights
Initial guess
Epoch 1

9109/10000
Epoch 2

9277/10000
Epoch 4

9308/10000
Other epochs

Epoch 0: 9109 / 10000
Epoch 1: 9277 / 10000
Epoch 2: 9237 / 10000
Epoch 3: 9248 / 10000
Epoch 4: 9308 / 10000
Epoch 5: 9340 / 10000
Epoch 6: 9317 / 10000
Epoch 7: 9360 / 10000
Epoch 8: 9368 / 10000
Epoch 9: 9315 / 10000
Epoch 10: 9365 / 10000
Epoch 11: 9331 / 10000
Epoch 12: 9379 / 10000
Epoch 13: 9432 / 10000
Epoch 14: 9382 / 10000
Epoch 15: 9408 / 10000
Epoch 16: 9429 / 10000
Epoch 17: 9393 / 10000
Epoch 18: 9444 / 10000
Epoch 19: 9438 / 10000
A REVIEW

Story so far

- Neural nets can be trained via gradient descent that minimizes a loss function

- Backpropagation can be used to derive the derivatives of the loss

- Backprop *is not guaranteed* to find a “true” solution, even if it exists, and lies within the capacity of the network to model
  - The optimum for the loss function may not be the “true” solution

- For large networks, the loss function may have a large number of unpleasant saddle points
  - Which backpropagation may find
Convergence of gradient descent

- An iterative algorithm is said to converge to a solution if the value updates arrive at a fixed point
  - Where the gradient is 0 and further updates do not change the estimate

- The algorithm may not actually converge
  - It may jitter around the local minimum
  - It may even diverge

- Conditions for convergence?
Learning rate
A closer look at the convergence problem

• With dimension-independent learning rates, the solution will converge smoothly in some directions, but oscillate or diverge in others

• **Proposal:**
  – Keep track of oscillations
  – Emphasize steps in directions that converge smoothly
  – Shrink steps in directions that bounce around.
Momentum Update

- The momentum method maintains a running average of all gradients until the *current* step
  \[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Err}(W^{(k-1)}) \]
  \[ W^{(k)} = W^{(k-1)} + \Delta W^{(k)} \]
  - Typical $\beta$ value is 0.9

- The running average steps
  - Get longer in directions where gradient stays in the same sign
  - Become shorter in directions where the sign keeps flipping
Nestorov’s Accelerated Gradient

- Nestorov’s method

\[ \Delta W^{(k)} = \beta \Delta W^{(k-1)} - \eta \nabla_W \text{Err}(W^{(k)} + \beta \Delta W^{(k-1)}) \]

\[ W^{(k)} = W^{(k-1)} + \Delta W^{(k)} \]

Momentum methods emphasize directions of steady improvement are demonstrably superior to other methods
Other popular optimizers

RMSprop

RMSprop is an unpublished, adaptive learning rate method proposed by Geoff Hinton in Lecture 6e of his Coursera Class.

RMSprop and Adadelta have both been developed independently around the same time stemming from the need to resolve Adagrad’s radically diminishing learning rates. RMSprop in fact is identical to the first update vector of Adadelta that we derived above:

\[
E[g^2] = 0.9E[g^2]_{t-1} + 0.1g^2 \\
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t} + \epsilon} g_t
\]

RMSprop as well divides the learning rate by an exponentially decaying average of squared gradients. Hinton suggests \( \gamma \) to be set to 0.9, while a good default value for the learning rate \( \eta \) is 0.001.

Adam

Adaptive Moment Estimation (Adam) [14] is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients \( v_t \) like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients \( m_t \), similar to momentum. Whereas momentum can be seen as a ball running down a slope, Adam behaves like a heavy ball with friction, which thus prefers flat minima in the error surface. [15] We compute the decaying averages of past and past squared gradients \( m_t \) and \( v_t \) respectively as follows:

\[
m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \\
v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2
\]

\( m_t \) and \( v_t \) are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As \( m_t \) and \( v_t \) are initialized as vectors of 0’s, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. \( \beta_1 \) and \( \beta_2 \) are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

\[
\tilde{m}_t = \frac{m_t}{1 - \beta_1^t} \\
\tilde{v}_t = \frac{v_t}{1 - \beta_2^t}
\]

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

\[
\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\tilde{v}_t} + \epsilon} \tilde{m}_t.
\]

The authors propose default values of 0.9 for \( \beta_1 \), 0.999 for \( \beta_2 \), and \( 10^{-8} \) for \( \epsilon \). They show empirically that Adam works well in practice and compares favorably to other
Other omitted tricks

**REGULARIZATION**

- **Batch normalization**
  - Batch Normalization | What is Batch Normalization in Deep Learning (analyticsvidhya.com)
  - Batch normalization - Wikipedia

- **Regularization**
  
  \[ L(W_1, W_2, \ldots, W_K) = \frac{1}{T} \sum_t \text{Div}(Y_t, d_t) + \frac{1}{2} \lambda \sum_k \|W_k\|_2^2 \]

  - Batch mode:
    \[ \Delta W_k = \frac{1}{T} \sum_t \nabla W_k \text{Div}(Y_t, d_t)^T + \lambda W_k \]

- **Dropout**: During training, for each input, at each iteration turn off” each neuron with a probability 1-\(a\)

- **Data augmentation**
Chain of assumptions in ML

→ Fit training set well on cost function
  (≈ Human level performance)
  
→ Fit dev set well on cost function
  
→ Fit test set well on cost function

→ Performs well in real world
  (Happy cat piε off users.)
Train/dev/test sets

Data

- Training set
  - Hold-out cross validation
  - Development set "dev"

- Test set

Proportion:
- 70/30/10
- 100 - 1000 - 10000

Big data:
- 1000,000

Accuracy:
- 98/1/1/1
- 99.5% (≈ 25/25/10/1)
Outline

• What is a CNN (convolutional Neural Network)
• Image Classification
  • AlexNet: Network structure
    • Dropout, RELU
  • NN as feature vector
  • More recent networks:
    • VGG
    • ResNet
• Domain adaptation
  • Transfer learning, fine-tuning
  • Example: Python detection
References

- https://www.cs.cmu.edu/~bhiksha/courses/deeplearning/Fall.2015/
- Coursera (Deep learning specialization)
Convolutional Neural Networks
A problem

- Will an MLP that recognizes the left image as a flower also recognize the one on the right as a flower?
The need for *shift invariance*
Convolutional Neural Network (CNN)

- A class of Neural Networks
  - Takes image as input (mostly)
  - Make predictions about the input image
Neural Network vs CNN

• Image as input in neural network
  • Size of feature vector = HxWxC
  • For 256x256 RGB image
    • 196608 dimensions

• CNN - Special type of neural network
  • Operate with volume of data
  • Weight sharing in form of kernels

Source: http://cs231n.github.io
What is a convolution

Example 5x5 image with binary pixels

Example 3x3 filter

bias

\[ z(i,j) = \sum_{k=1}^{3} \sum_{l=1}^{3} f(k,l)I(i + l, j + k) + b \]

- Scanning an image with a “filter”
  - Note: a filter is really just a perceptron, with weights and a bias
What is a convolution

- Scanning an image with a “filter”
  - At each location, the “filter and the underlying map values are multiplied component wise, and the products are added along with the bias
The “Stride” between adjacent scanned locations need not be 1

• Scanning an image with a “filter”
  – The filter may proceed by *more* than 1 pixel at a time
  – E.g. with a “hop” of two pixels per shift
The “Stride” between adjacent scanned locations need not be 1

- Scanning an image with a “filter”
  - The filter may proceed by more than 1 pixel at a time
  - E.g. with a “hop” of two pixels per shift
The “Stride” between adjacent scanned locations need not be 1

- Scanning an image with a “filter”
  - The filter may proceed by *more* than 1 pixel at a time
  - E.g. with a “hop” of *two* pixels per shift
Extending to multiple input maps

\[
z(i, j) = \sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S, l, n}(p, k, l) Y_{p}(i + l, j + k) + b
\]

- The computation of the convolutive map at any location sums the convolutive outputs at all planes
Extending to multiple input maps

\[
z(i, j) = \sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p, k, l) Y_p(i + l, j + k) + b
\]

- The computation of the convolutive map at any location **sums** the convolutive outputs *at all planes*
Extending to multiple input maps

\[ z(i, j) = \sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{s,l,n}(p, k, l) Y_p(i + l, j + k) + b \]

- The computation of the convolutive map at any location sums the convolutive outputs at all planes
Extending to multiple input maps

\[ z(i, j) = \sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p, k, l) Y_p(i + l, j + k) + b \]

- The computation of the convolutive map at any location *sums* the convolutive outputs *at all planes*
Extending to multiple input maps

\[ z(i, j) = \sum_{p} \sum_{k=1}^{L} \sum_{l=1}^{L} w_{S,l,n}(p, k, l)Y_{p}(i + l, j + k) + b \]

- The computation of the convolutive map at any location sums the convolutive outputs at all planes.
The size of the convolution

- Image size: $N \times N$
- Filter: $M \times M$
- Stride: $S$
- Output size (each side) = $\lceil(N - M)/S\rceil + 1$
  - Assuming you’re not allowed to go beyond the edge of the input
Convolutional Network

• Convolution network is a sequence of these layers

![Diagram showing a convolutional network with 6 5x5x3 filters](image)

3

32

32

6 5x5x3 filters
Convolutional Network

- Convolution network is a sequence of these layers

6 5x5x3 filters
Parameters

32x32x3 image → Convolution layer → Activation maps

6 3x3x3 kernels – 6x3x3x3x3 parameters = 162
2D Convolution - dimensions

7x7 map

3x3 filter

Output activation map 5x5
Output size
N-F+1
(7 – 3 + 1) = 5

N – input size
F – filter size
Stride

7x7 map

3x3 filter

Filter applied with stride 2
Stride

7x7 map

3x3 filter

Filter applied with stride 2
Stride

7x7 map

3x3 filter

Filter applied with stride 2

Activation map size 3x3
Output size
\((7-3)/2 + 1 = 3\)

\((N-F)/S + 1\)
Stride

7x7 map

3x3 filter

Filter applied with stride 3
Stride

7x7 map

3x3 filter

Filter applied with stride 3

Cannot cover perfectly

Not all parameters will fit
Stride

7x7 map

3x3 filter
Output size \((N-F)/S + 1\)
\(N = 7, F = 3\)

Stride 1
\((7-3)/1 + 1 \Rightarrow 5\)

Stride 2
\((7-3)/2 + 1 \Rightarrow 3\)

Stride 3
\((7-3)/3 + 1 \Rightarrow 2.33\)
Solution

• Zero-pad the input
  – Pad the input image/map all around
  – Pad as symmetrically as possible, such that...
  – For stride 1, the result of the convolution is the same size as the original image
**Padding**

- Zero padding in the input

For 7x7 input and 3x3 filter

If we have padding of one pixel

Output

7x7

Size (recall \((N-F)/S+1\))

\((N-F+2P)/S + 1\)
Padding

• Zero padding in the input

Common to see,
(F-1)/2 padding with stride 1 to preserve the map size

\[ N = \frac{(N-F+2P)}{S} + 1 \]
\[ \Rightarrow (N-1)S = N-F+2P \]
\[ \Rightarrow P = \frac{(F-1)}{2} \]
Why convolution?

- Convolutional neural networks are, in fact, equivalent to *scanning* with an MLP
  - Just run the entire MLP on each block separately, and combine results
    - As opposed to scanning (convolving) the picture with individual neurons/filters
    - Even computationally, the number of operations in both computations is identical
      - The neocognitron in fact views it equivalently to a scan
  - So why convolutions?
Convolutional Neural Networks

$K_1$ total filters
Filter size: $L \times L \times 3$

Small enough to capture fine features (particularly important for scaled-down images)

$1 \times 1$ image

- Input is convolved with a set of $K_1$ filters
  - Typically $K_1$ is a power of 2, e.g. 2, 4, 8, 16, 32, ...
  - Filters are typically 5x5, 3x3, or even 1x1

What on earth is this?
The 1x1 filter

• A 1x1 filter is simply a perceptron that operates over the *depth* of the map, but has no spatial extent
  – Takes one pixel from each of the maps (at a given location) as input
Convolutional Neural Network (CNN)

- A class of Neural Networks
  - Takes image as input (mostly)
  - Make predictions about the input image
Convolutional Neural Networks

- Input is convolved with a set of $K_1$ filters
  - Typically $K_1$ is a power of 2, e.g. 2, 4, 8, 16, 32,..
  - Better notation: Filters are typically 5x5(x3), 3x3(x3), or even 1x1(x3)
  - Typical stride: 1 or 2

Parameters to choose: $K_1$, $L$ and $S$
1. Number of filters $K_1$
2. Size of filters $L \times L \times 3 + \text{bias}$
3. Stride of convolution $S$

Total number of parameters: $K_1(3L^2 + 1)$
The convolution operation results in a convolution map
An *Activation* is finally applied to every entry in the map
Convolutional Neural Networks

First convolutional layer: Several convolutional filters
   - Filters are “3-D” (third dimension is color)
   - Convolution followed typically by a RELU activation

Each filter creates a single 2-D output map
Convolutional Neural Networks

Filter size: \( L \times L \times 3 \)

The layer pools \( P \times P \) blocks of \( Y \) into a single value. It employs a stride \( D \) between adjacent blocks.

\[
U_m^{(1)}(i,j) = \max_{k \in [(i-1)D+1, iD], \ l \in [(j-1)D+1, jD]} Y_m^{(1)}(k,l)
\]

- **First downsampling layer:** From each \( P \times P \) block of each map, *pool* down to a single value.
  - For max pooling, during training keep track of which position had the highest value.
Convolutional Neural Networks

$W_m: 3 \times L \times L$
$m = 1 \ldots K_1$

$W_m: K_1 \times L_2 \times L_2$
$m = 1 \ldots K_2$

$Y^{(1)}_{K_1}$

$K_1 \times I \times I$

$K_1 \times [I/D] \times [I/D]$

Pool: $P \times P(D)$

$z_m^{(n)}(i, j) = \sum_{r=1}^{K_{n-1}} \sum_{k=1}^{L_n} \sum_{l=1}^{L_n} w_m^{(n)}(r, k, l) U_r^{(n-1)}(i + k, j + l) + b_m^{(n)}$

$Y_m^{(n)}(i, j) = f\left(z_m^{(n)}(i, j)\right)$

- **Second convolutional layer**: $K_2$ 3-D filters resulting in $K_2$ 2-D maps
Convolutional Neural Network (CNN)

- A class of Neural Networks
  - Takes image as input (mostly)
  - Make predictions about the input image
The other component
Downsampling/Pooling

- Convolution (and activation) layers are followed intermittently by “downsampling” (or “pooling”) layers
  - Often, they alternate with convolution, though this is not necessary
Pooling

- Makes the representations smaller
- Operates over each activation map independently
Recall: Max pooling

- Max pooling selects the largest from a pool of elements
- Pooling is performed by “scanning” the input
Pooling

- Kernel size
- Stride

Single depth slice

max pool with 2x2 filters and stride 2

6 8
3 4
Alternative to Max pooling: Mean Pooling

Single depth slice

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Mean pool with 2x2 filters and stride 2

<table>
<thead>
<tr>
<th>3.25</th>
<th>5.25</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Convolutional Neural Network (CNN)

- A class of Neural Networks
  - Takes image as input (mostly)
  - Make predictions about the input image
Convolutional Neural Networks

\[ W_m : 3 \times L \times L \]
\[ m = 1 \ldots K_1 \]

\[ W_m : K_1 \times L_2 \times L_2 \]
\[ m = 1 \ldots K_2 \]

\[ K_1 \times [l/D] \times [l/D] \]

\[ K_1 \times I \times I \]

Pool: \( P \times P(D) \)

\[ Y_K^{(1)} \]

\[ K_1 \]

\[ U_K^{(1)} \]

\[ K_1 \]

\[ Y_K^{(2)} \]

\[ K_2 \]

\[ U_m^{(n)}(i,j) = Y_m^{(n)}(P_m^{(n)}(i,j)) \]

\[ p_m^{(n)}(i,j) = \operatorname{argmax}_{k \in ([i-1)d+1, id], \ i \in ([j-1)d+1, fd]} Y_m^{(n)}(k,l) \]

- Second convolutional layer: \( K_2 \) 3-D filters resulting in \( K_2 \) 2-D maps
- Second pooling layer: \( K_2 \) Pooling operations: outcome \( K_2 \) reduced 2D maps
Convolutional Neural Networks

$W_m: 3 \times L \times L$
$m = 1 \ldots K_1$

$W_m: K_1 \times L_2 \times L_2$
$m = 1 \ldots K_2$

$K_1 \times I \times I$
$K_1 \times [I/D] \times [I/D]$

$Y_{K_1}^{(1)}$
$U_{K_1}^{(1)}$
$Y_{K_2}^{(2)}$

Pool: $P \times P(D)$

• This continues for several layers until the final convolved output is fed to an MLP
Parameters to choose (design choices)

- Number of convolutional and downsampling layers
  - And arrangement (order in which they follow one another)

- For each convolution layer:
  - Number of filters $K_i$
  - Spatial extent of filter $L_i \times L_i$
    - The “depth” of the filter is fixed by the number of filters in the previous layer $K_{i-1}$
  - The stride $S_i$

- For each downsampling/pooling layer:
  - Spatial extent of filter $P_i \times P_i$
  - The stride $D_i$

- For the final MLP:
  - Number of layers, and number of neurons in each layer
Binary classification

• Target class present or not?
  • Single output
  • Two outputs
Multi-class

- One prediction for each class
Softmax activation

\[
\text{scores} = \text{unnormalized log probabilities of the classes.}
\]

\[
P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \quad \text{where} \quad s = f(x_i; W)
\]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>3.2</td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
</tr>
</tbody>
</table>

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Softmax activation

scores = unnormalized log probabilities of the classes.

\[ P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \]

where \[ s = f(x_i; W) \]

cat 3.2

car 5.1

frog -1.7

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Softmax activation

scores = unnormalized log probabilities of the classes.

\[ P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \]

where

\[ s = f(x_i; W) \]

<table>
<thead>
<tr>
<th>Class</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>3.2</td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
</tr>
</tbody>
</table>

\[ \exp \rightarrow 24.5 \]

\[ \exp \rightarrow 164.0 \]

\[ \exp \rightarrow 0.18 \]

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Softmax activation

scores = unnormalized log probabilities of the classes.

\[ P(Y = k | X = x_i) = \frac{e^{s_k}}{\sum_j e^{s_j}} \]

where \[ s = f(x_i; W) \]

cat  \[ 3.2 \]
      \[ 5.1 \]
      \[ -1.7 \]

car  \[ \text{exp} \]

24.5
164.0
0.18

normalize

0.13
0.87
0.00

probabilities

Slide Credit: Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n
Multi-label

- Multiple classes can be active
- Softmax will not work
- Use sigmoid activation
Why not correlation neural network?

• It could be
  • Deep learning libraries actually implement correlation

• Correlation relates to convolution via a 180deg rotation
  • When we learn kernels, we could easily learn them flipped
Digit classification
Learning phases

Training

Images

Labels

Image Features

Training

Trained classifier

Testing

Image not in training set

Image Features

Apply classifier

Prediction

Slide credit: D. Hoiem and L. Lazebnik
General CNN architecture

End to end learning!
Learning the network

- Parameters to be learned:
  - The weights of the neurons in the final MLP
  - The (weights and biases of the) filters for every convolutional layer
Backpropagation: Final flat layers

- Backpropagation from the flat MLP requires special consideration of
  - The pooling layers (particularly Maxout)
  - The shared computation in the convolution layers
Training Issues

• Standard convergence issues
  – Solution: RMS prop or other momentum-style algorithms
  – Other tricks such as batch normalization

• The number of parameters can quickly become very large

• Insufficient training data to train well
  – Solution: Data augmentation
Data Augmentation

- **Original data**
- **Augmented data**

- rotation: uniformly chosen random angle between 0° and 360°
- translation: random translation between -10 and 10 pixels
- rescaling: random scaling with scale factor between 1/1.6 and 1.6 (log-uniform)
- flipping: yes or no (bernoulli)
- shearing: random shearing with angle between -20° and 20°
- stretching: random stretching with stretch factor between 1/1.3 and 1.3 (log-uniform)
https://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html
AlexNet

- 1.2 million high-resolution images from ImageNet LSVRC-2010 contest
- 1000 different classes (softmax layer)
- NN configuration
  - NN contains 60 million parameters and 650,000 neurons,
  - 5 convolutional layers, some of which are followed by max-pooling layers
  - 3 fully-connected layers

Krizhevsky et. al.

- Input: 227x227x3 images
- Conv1: 96 11x11 filters, stride 4, no zeropad
- Pool1: 3x3 filters, stride 2
- “Normalization” layer [Unnecessary]
- Conv2: 256 5x5 filters, stride 2, zero pad
- Pool2: 3x3, stride 2
- Normalization layer [Unnecessary]
- Conv3: 384 3x3, stride 1, zeropad
- Conv4: 384 3x3, stride 1, zeropad
- Conv5: 256 3x3, stride 1, zeropad
- Pool3: 3x3, stride 2
- FC: 3 layers,
  - 4096 neurons, 4096 neurons, 1000 output neurons
AlexNet: Network Size

- Input: 227x227x3 images
- First layer (CONV1): 96 11x11 filters applied at stride 4
- What is the output volume size? \((227-11)/4+1 = 55\)
- What is the number of parameters? \(11 \times 11 \times 3 \times 96 = 35K\)
AlexNet: Network Size

- After CONV1: 55x55x96
- Second layer (POOL1): 3x3 filters applied at stride 2
- What is the output volume size? \( (55-3)/2+1 = 27 \)
- What is the number of parameters in this layer? 0
AlexNet: Network Size

- After POOL1: 27x27x96
- Third layer (NORM1): Normalization
- What is the output volume size? 27x27x96
AlexNet: Network Size

1. [227x227x3] INPUT
2. [55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
3. [27x27x96] MAX POOL1: 3x3 filters at stride 2
4. [27x27x96] NORM1: Normalization layer
5. [27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
6. [13x13x256] MAX POOL2: 3x3 filters at stride 2
7. [13x13x256] NORM2: Normalization layer
8. [13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
9. [13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
10. [13x13x384] CONV5: 256 3x3 filters at stride 1, pad 1
11. [6x6x256] MAX POOL3: 3x3 filters at stride 2
12. [4096] FC6: 4096 neurons
13. [4096] FC7: 4096 neurons
14. [1000] FC8: 1000 neurons (class scores)
Alexnet: Total parameters

• 650K neurons
• 60M parameters
• 630M connections

• Testing: Multi-crop
  – Classify different shifts of the image and vote over the lot!
Learning magic in Alexnet

- Activations were RELU
  - Made a large difference in convergence
- “Dropout” – 0.5 (in FC layers only)
- Large amount of data augmentation
- SGD with mini batch size 128
- Momentum, with momentum factor 0.9
- L2 weight decay 5e-4
- Learning rate: 0.01, decreased by 10 every time validation accuracy plateaus
- Evaluated using: Validation accuracy

\[ L(w, b) + \frac{\lambda}{2} \|w\|^2 \]

- Final top-5 error: 18.2% with a single net, 15.4% using an ensemble of 7 networks
  - Lowest prior error using conventional classifiers: > 25%
Figure 3: 96 convolutional kernels of size $11 \times 11 \times 3$ learned by the first convolutional layer on the $224 \times 224 \times 3$ input images. The top 48 kernels were learned on GPU 1 while the bottom 48 kernels were learned on GPU 2. See Section 6.1 for details.

The net actually learns features!

Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5).

ZFNet

ZFNet Architecture

- **Zeiler and Fergus 2013**
- Same as Alexnet except:
  - 7x7 input-layer filters with stride 2
  - 3 conv layers are 512, 1024, 512
  - Error went down from 15.4% → 14.8%
    - Combining multiple models as before
Visualizing Convolution

Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]
VGGNet

- Simonyan and Zisserman, 2014
- *Only* used 3x3 filters, stride 1, pad 1
- *Only* used 2x2 pooling filters, stride 2

- Tried a large number of architectures.
- Finally obtained 7.3% top-5 error using 13 conv layers and 3 FC layers
  - Combining 7 classifiers
  - Subsequent to paper, reduced error to 6.8% using only two classifiers
- Final arch: 64 conv, 64 conv, 64 pool,
  128 conv, 128 conv,
  128 pool,
  256 conv, 256 conv, 256 conv,
  256 pool,
  512 conv, 512 conv, 512 conv,
  512 pool,
  512 conv, 512 conv, 512 conv,
  512 pool,
  FC with 4096, 4096, 1000
- ~140 million parameters in all! Madness!

### ConvNet Configuration

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>A-LRN</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>11 weight layers</td>
<td>11 weight layers</td>
<td>13 weight layers</td>
<td>16 weight layers</td>
<td>16 weight layers</td>
<td>19 weight layers</td>
</tr>
<tr>
<td>conv3-64</td>
<td>conv3-64</td>
<td>LRN</td>
<td>conv3-64</td>
<td>conv3-64</td>
<td>conv3-64</td>
<td>conv3-64</td>
</tr>
<tr>
<td>maxpool</td>
<td>conv3-128</td>
<td>conv3-128</td>
<td>conv3-128</td>
<td>conv3-128</td>
<td>conv3-128</td>
<td>conv3-128</td>
</tr>
<tr>
<td>maxpool</td>
<td>conv3-256</td>
<td>conv3-256</td>
<td>conv3-256</td>
<td>conv3-256</td>
<td>conv3-256</td>
<td>conv3-256</td>
</tr>
<tr>
<td>maxpool</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
</tr>
<tr>
<td>maxpool</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
</tr>
<tr>
<td>maxpool</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
</tr>
<tr>
<td>maxpool</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
<td>conv3-512</td>
</tr>
<tr>
<td>maxpool</td>
<td>FC-4096</td>
<td>FC-4096</td>
<td>FC-1000</td>
<td>soft-max</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Googlenet: Inception

- Multiple filter sizes simultaneously
- Details irrelevant; error $\Rightarrow 6.7\%$
  - Using only 5 million parameters, thanks to average pooling
Imagenet

- Resnet: 2015
  - Current top-5 error: < 3.5%
  - Over 150 layers, with “skip” connections.
Resnet details for the curious..

- Last layer before addition must have the same number of filters as the input to the module
- Batch normalization after each convolution
- SGD + momentum (0.9)
- Learning rate 0.1, divide by 10 (batch norm lets you use larger learning rate)
- Mini batch 256
- Weight decay 1e-5
- **No pooling in Resnet**
Questions?