CS 2770: Computer Vision

Neural Networks

Prof. Adriana Kovashka
University of Pittsburgh
January 19, 2017
Plan for the next few lectures

Why (convolutional) neural networks?

Neural network basics
  • Architecture
  • Biological inspiration
  • Loss functions
  • Optimization / gradient descent
  • Training with backpropagation

Convolutional neural networks (CNNs)
  • Special operations
  • Common architectures

Understanding CNNs
  • Visualization
  • Synthesis / style transfer
  • Breaking CNNs

Practical matters
  • Tips and tricks for training
  • Transfer learning
  • Software packages
Why (convolutional) neural networks?

Obtained state of the art performance on many problems…

Most papers in CVPR 2016 use deep learning

Razavian et al., CVPR 2014 Workshops
ImageNet Challenge 2012

[Deng et al. CVPR 2009]

- ~14 million labeled images, 20k classes
- Images gathered from Internet
- Human labels via Amazon Turk
- Challenge: 1.2 million training images, 1000 classes

ImageNet Challenge 2012

- AlexNet: Similar framework to LeCun’98 but:
  - Bigger model (7 hidden layers, 650,000 units, 60,000,000 params)
  - More data (10⁶ vs. 10³ images)
  - GPU implementation (50x speedup over CPU)
    - Trained on two GPUs for a week
  - Better regularization for training (DropOut)


Adapted from Lana Lazebnik
ImageNet Challenge 2012

Krizhevsky et al. -- 16.4% error (top-5)
Next best (non-convnet) – 26.2% error
Object detection system overview. Our system (1) takes an input image, (2) extracts around 2000 bottom-up region proposals, (3) computes features for each proposal using a large convolutional neural network (CNN), and then (4) classifies each region using class-specific linear SVMs. **R-CNN achieves a mean average precision (mAP) of 53.7% on PASCAL VOC 2010.** For comparison, Uijlings et al. (2013) report 35.1% mAP using the same region proposals, but with a spatial pyramid and bag-of-visual-words approach. The popular deformable part models perform at 33.4%.

Object Detection

<table>
<thead>
<tr>
<th>Method</th>
<th>VOC 2007</th>
<th>VOC 2010</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPM (2011)</td>
<td>33.7</td>
<td>29.6</td>
</tr>
<tr>
<td>Regionlets (2013)</td>
<td>41.7</td>
<td>39.7</td>
</tr>
<tr>
<td>R-CNN (2014, AlexNet)</td>
<td>54.2</td>
<td>50.2</td>
</tr>
<tr>
<td>R-CNN + bbox reg (AlexNet)</td>
<td>58.5</td>
<td>53.7</td>
</tr>
<tr>
<td>R-CNN (VGG-16)</td>
<td>66.0</td>
<td>62.9</td>
</tr>
</tbody>
</table>
## Object Detection

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Training Time:</td>
<td>84 hours</td>
<td>9.5 hours</td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>8.8x</td>
</tr>
<tr>
<td>Test time per image</td>
<td>47 seconds</td>
<td>0.32 seconds</td>
</tr>
<tr>
<td>(Speedup)</td>
<td>1x</td>
<td>146x</td>
</tr>
<tr>
<td>Better!</td>
<td>66.0</td>
<td>66.9</td>
</tr>
</tbody>
</table>

Using VGG-16 CNN on Pascal VOC 2007 dataset

Adapted from Andrej Karpathy
Beyond classification

Detection
Segmentation
Regression
Pose estimation
Synthesis

and many more…

Adapted from Jia-bin Huang
What are CNNs?

- Convolutional neural networks are a type of \textit{neural network}.
- The neural network includes layers that perform special operations.
- Used in vision, but to a lesser extent also in NLP, biomedical, etc.
- Often they are \textit{deep}. 
Deep neural network

Figure from http://neuralnetworksanddeeplearning.com/chap5.html
Traditional Recognition Approach

- Features are key to recent progress in recognition, but research shows they’re flawed…
- Where next? Better classifiers? Or keep building more features?

Adapted from Lana Lazebnik
What about learning the features?

- Learn a *feature hierarchy* all the way from pixels to classifier
- Each layer extracts features from the output of previous layer
- Train all layers jointly
“Shallow” vs. “deep” architectures

Traditional recognition: “Shallow” architecture

Image/Video Pixels → Hand-designed feature extraction → Trainable classifier → Object Class

Deep learning: “Deep” architecture

Image/Video Pixels → Layer 1 → ... → Layer N → Simple classifier → Object Class
Neural network definition

**Figure 5.1** Network diagram for the two-layer neural network corresponding to (5.7). The input, hidden, and output variables are represented by nodes, and the weight parameters are represented by links between the nodes, in which the bias parameters are denoted by links coming from additional input and hidden variables \( x_0 \) and \( z_0 \). Arrows denote the direction of information flow through the network during forward propagation.

- **Activations:**

  \[
  a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}
  \]

- **Nonlinear activation function** \( h \) (e.g. sigmoid, tanh):

  \[
  z_j = h(a_j) \quad \text{tanh} \, x = \frac{\sinh x}{\cosh x} = \frac{e^x - e^{-x}}{e^x + e^{-x}}
  \]

Recall SVM: \( w^T x + b \)
Neural network definition

- Layer 2
  \[ a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \]

- Layer 3 (final)
  \[ a_k = \]

- Outputs
  (binary) \[ y_k = \sigma(a_k) = \frac{1}{1 + \exp(-a_k)} \]
  (multiclass) \[ y_k = \frac{\exp(a_k)}{\sum_j \exp(a_j)} \]

- Finally:
  (binary) \[ y_k(x, w) = \sigma\left( \sum_{j=1}^{M} w_{kj}^{(2)} h\left( \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \right) + w_{k0}^{(2)} \right) \]
Activation functions

**Sigmoid**

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

**tanh**

\[ \tanh(x) \]

**ReLU**

\[ \max(0, x) \]

**Leaky ReLU**

\[ \max(0.1x, x) \]

**Maxout**

\[ \max(w_1^T x + b_1, w_2^T x + b_2) \]

**ELU**

\[ f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha (\exp(x) - 1) & \text{if } x \leq 0 \end{cases} \]

Andrej Karpathy
A multi-layer neural network

- **Nonlinear classifier**
- Can approximate any continuous function to arbitrary accuracy given sufficiently many hidden units
Inspiration: Neuron cells

- Neurons
  - accept information from multiple inputs,
  - transmit information to other neurons.
- Multiply inputs by weights along edges
- Apply some function to the set of inputs at each node
- If output of function over threshold, neuron “fires”
A neuron

Input

Weights

$x_1$

$w_1$

$x_2$

$w_2$

$x_3$

$w_3$

$\cdots$

$w_d$

$x_d$

Output: $\sigma(w \cdot x + b)$

Sigmoid function:

$$\sigma(t) = \frac{1}{1 + e^{-t}}$$
Multilayer networks

- Cascade neurons together
- Output from one layer is the input to the next
- Each layer has its own sets of weights
Feed-forward networks

- Predictions are fed forward through the network to classify
Feed-forward networks

• Predictions are fed forward through the network to classify
Feed-forward networks

- Predictions are fed forward through the network to classify
Feed-forward networks

- Predictions are fed forward through the network to classify
Feed-forward networks

- Predictions are fed forward through the network to classify
Feed-forward networks

- Predictions are fed forward through the network to classify
Deep neural networks

• Lots of hidden layers
• Depth = power (usually)
How do we train them?

• The goal is to iteratively find such a set of weights that allow the activations/outputs to match the desired output

• We want to minimize a *loss function*

• The loss function is a function of the weights in the network

• For now let’s simplify and assume there’s a single layer of weights in the network
Classification goal

Example dataset: CIFAR-10
- 10 labels
- 50,000 training images
- each image is 32x32x3
- 10,000 test images.
Classification scores

\[ f(x, W) = Wx \]

\[ f(x, W) \rightarrow 10 \text{ numbers, indicating class scores} \]

[32x32x3] array of numbers 0...1 (3072 numbers total)
Linear classifier

\[ f(x, W) = Wx + (b) \]

[32x32x3] array of numbers 0...1

10 numbers, indicating class scores

parameters, or “weights”
Linear classifier

Example with an image with 4 pixels, and 3 classes (cat/dog/ship)
Linear classifier

Going forward: Loss function/Optimization

<table>
<thead>
<tr>
<th>Category</th>
<th>Score 1</th>
<th>Score 2</th>
<th>Score 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>airplane</td>
<td>-3.45</td>
<td>-0.51</td>
<td>3.42</td>
</tr>
<tr>
<td>automobile</td>
<td>-8.87</td>
<td>6.04</td>
<td>4.64</td>
</tr>
<tr>
<td>bird</td>
<td>0.09</td>
<td>5.31</td>
<td>2.65</td>
</tr>
<tr>
<td>cat</td>
<td>2.9</td>
<td>-4.22</td>
<td>5.1</td>
</tr>
<tr>
<td>deer</td>
<td>4.48</td>
<td>-4.19</td>
<td>2.64</td>
</tr>
<tr>
<td>dog</td>
<td>8.02</td>
<td>3.58</td>
<td>5.55</td>
</tr>
<tr>
<td>frog</td>
<td>3.78</td>
<td>4.49</td>
<td>-4.34</td>
</tr>
<tr>
<td>horse</td>
<td>1.06</td>
<td>-4.37</td>
<td>-1.5</td>
</tr>
<tr>
<td>ship</td>
<td>-0.36</td>
<td>-2.09</td>
<td>-4.79</td>
</tr>
<tr>
<td>truck</td>
<td>-0.72</td>
<td>-2.93</td>
<td>6.14</td>
</tr>
</tbody>
</table>

TODO:

1. Define a **loss function** that quantifies our unhappiness with the scores across the training data.

2. Come up with a way of efficiently finding the parameters that minimize the loss function. *(optimization)*
### Linear classifier

Suppose: 3 training examples, 3 classes. With some $W$ the scores $f(x, W) = Wx$ are:

<table>
<thead>
<tr>
<th></th>
<th>cat</th>
<th>car</th>
<th>frog</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>3.2</td>
<td>1.3</td>
<td>2.2</td>
</tr>
<tr>
<td></td>
<td>5.1</td>
<td>4.9</td>
<td>2.5</td>
</tr>
<tr>
<td></td>
<td>-1.7</td>
<td>2.0</td>
<td>-3.1</td>
</tr>
</tbody>
</table>
Suppose: 3 training examples, 3 classes. With some $W$ the scores $f(x, W) = Wx$ are:

<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>

For the cat example:

<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>

**Multiclass SVM loss:**

Given an example $(x_i, y_i)$ where $x_i$ is the image and $y_i$ is the (integer) label, and using the shorthand for the scores vector: $s = f(x_i, W)$, the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$
Linear classifier: SVM loss

Suppose: 3 training examples, 3 classes.
With some $W$ the scores $f(x, W) = Wx$ are:

cat | 3.2 | 1.3 | 2.2
--- | --- | --- | ---
car | 5.1 | 4.9 | 2.5
frog | -1.7 | 2.0 | -3.1

**Multiclass SVM loss:**

Given an example $(x_i, y_i)$ where $x_i$ is the image and $y_i$ is the (integer) label, and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$= \max(0, 5.1 - 3.2 + 1) + \max(0, -1.7 - 3.2 + 1)$$

$$= \max(0, 2.9) + \max(0, -3.9)$$

$$= 2.9 + 0$$

$$= 2.9$$

Andrej Karpathy
Linear classifier: SVM loss

Suppose: 3 training examples, 3 classes. With some $W$ the scores $f(x, W) = Wx$ are:

Multiclass SVM loss:

Given an example $(x_i, y_i)$ where $x_i$ is the image and $y_i$ is the (integer) label, and using the shorthand for the scores vector: $s = f(x_i, W)$, the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$= \max(0, 1.3 - 4.9 + 1) + \max(0, 2.0 - 4.9 + 1)$$

$$= \max(0, -2.6) + \max(0, -1.9)$$

$$= 0 + 0$$

$$= 0$$
Linear classifier: SVM loss

Suppose: 3 training examples, 3 classes. With some $W$ the scores $f(x, W) = Wx$ are:

<table>
<thead>
<tr>
<th></th>
<th>cat</th>
<th>car</th>
<th>frog</th>
</tr>
</thead>
<tbody>
<tr>
<td>score</td>
<td>3.2</td>
<td>5.1</td>
<td>-1.7</td>
</tr>
<tr>
<td>label</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>2.0</td>
<td></td>
</tr>
<tr>
<td>Loss</td>
<td>2.9</td>
<td>0</td>
<td>12.9</td>
</tr>
</tbody>
</table>

Multiclass SVM loss:

Given an example $(x_i, y_i)$ where $x_i$ is the image and $y_i$ is the (integer) label, and using the shorthand for the scores vector: $s = f(x_i, W)$, the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$= \max(0, 2.2 - (-3.1) + 1) + \max(0, 2.5 - (-3.1) + 1)$$
$$= \max(0, 5.3 + 1) + \max(0, 5.6 + 1)$$
$$= 6.3 + 6.6$$
$$= 12.9$$

Adapted from Andrej Karpathy
### Linear classifier: SVM loss

Suppose: 3 training examples, 3 classes. With some $W$ the scores $f(x, W) = Wx$ are:

<table>
<thead>
<tr>
<th></th>
<th>cat</th>
<th>car</th>
<th>frog</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>1.3</td>
<td>2.2</td>
<td></td>
</tr>
<tr>
<td>5.1</td>
<td>4.9</td>
<td>2.5</td>
<td></td>
</tr>
<tr>
<td>-1.7</td>
<td>2.0</td>
<td>-3.1</td>
<td></td>
</tr>
</tbody>
</table>

**Losses:**

|       | 2.9  | 0    | 12.9 |

### Multiclass SVM loss:

Given an example $(x_i, y_i)$ where $x_i$ is the image and $y_i$ is the (integer) label, and using the shorthand for the scores vector: $s = f(x_i, W)$

the SVM loss has the form:

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

and the full training loss is the mean over all examples in the training data:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i$$

$$L = \frac{(2.9 + 0 + 12.9)}{3} = 15.8 / 3 = 5.3$$

Adapted from Andrej Karpathy
Linear classifier: SVM loss

\[ f(x, W) = Wx \]

\[ L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) \]
Linear classifier: SVM loss

Weight Regularization

$$L = \frac{1}{N} \sum_{i=1}^{N} \sum_{j \neq y_i} \max(0, f(x_i; W)_j - f(x_i; W)_{y_i} + 1) + \lambda R(W)$$

$\lambda = \text{regularization strength (hyperparameter)}$

In common use:
L2 regularization

$R(W) = \sum_k \sum_l W_{k,l}^2$

L1 regularization

$R(W) = \sum_k \sum_l |W_{k,l}|$

Dropout (will see later)

Adapted from Andrej Karpathy
scores = unnormalized log probabilities of the classes.

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

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

Want to maximize the log likelihood, or (for a loss function) to minimize the negative log likelihood of the correct class:

\[ L_i = - \log P(Y = y_i | X = x_i) \]
Another loss: Softmax

\[ L_i = - \log \left( \frac{e^{s_{y_i}}}{\sum_j e^{s_j}} \right) \]

<table>
<thead>
<tr>
<th></th>
<th>Unnormalized Log Probabilities</th>
<th>Probabilities</th>
</tr>
</thead>
<tbody>
<tr>
<td>cat</td>
<td>3.2</td>
<td>0.13</td>
</tr>
<tr>
<td>car</td>
<td>5.1</td>
<td>0.87</td>
</tr>
<tr>
<td>frog</td>
<td>-1.7</td>
<td>0.00</td>
</tr>
</tbody>
</table>

\[ L_i = -\log(0.13) = 0.89 \]

Adapted from Andrej Karpathy
How to minimize the loss function?
How to minimize the loss function?

In 1-dimension, the derivative of a function:

\[
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}
\]

In multiple dimensions, the **gradient** is the vector of (partial derivatives).
current $W$: 

\[
[0.34, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33,\ldots]
\]

loss 1.25347

gradient $dW$: 

\[
[?, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?, \\
?,\ldots]
\]
<table>
<thead>
<tr>
<th>current W:</th>
<th>( W + h ) (first dim):</th>
<th>gradient dW:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, \ldots]</td>
<td>[0.34 + 0.0001, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, \ldots]</td>
<td>[?, ?, ?, ?, ?, ?, ?, ?, ?, \ldots]</td>
</tr>
<tr>
<td>loss 1.25347</td>
<td>loss 1.25322</td>
<td></td>
</tr>
</tbody>
</table>
**current W:**

\[
\begin{align*}
0.34, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33, \\
\text{loss 1.25347}
\end{align*}
\]

**W + h (first dim):**

\[
\begin{align*}
0.34 + 0.0001, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33, \\
\text{loss 1.25322}
\end{align*}
\]

**gradient dW:**

\[
\begin{align*}
-2.5, \\
?, \\
?, \\
\text{(1.25322 - 1.25347)/0.0001} = -2.5
\end{align*}
\]
<table>
<thead>
<tr>
<th>current W:</th>
<th>$W + h$ (second dim):</th>
<th>gradient $dW$:</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]</td>
<td>[0.34, -1.11 + 0.0001, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33,...]</td>
<td>[-2.5, ?, ?, ?, ?, ?, ?, ?, ?, ...,?]</td>
</tr>
</tbody>
</table>

**loss 1.25347** | **loss 1.25353**
### current W:

| 0.34,  |
| -1.11, |
| 0.78,  |
| 0.12,  |
| 0.55,  |
| 2.81,  |
| -3.1,  |
| -1.5,  |
| 0.33,...] |

**loss 1.25347**

### W + h (second dim):

| 0.34, |
| -1.11 + **0.0001** , |
| 0.78, |
| 0.12, |
| 0.55, |
| 2.81, |
| -3.1, |
| -1.5, |
| 0.33,...] |

**loss 1.25353**

### gradient dW:

| [-2.5, |
| **0.6,** |
| ?, |
| ?, |

\[
\frac{df(x)}{dx} = \lim_{h \to 0} \frac{f(x + h) - f(x)}{h}
\]

\[
(1.25353 - 1.25347)/0.0001 = 0.6
\]

Andrej Karpathy
current W: [0.34, -1.11, 0.78, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, ...]  
W + h (third dim): [0.34, -1.11, 0.78 + 0.0001, 0.12, 0.55, 2.81, -3.1, -1.5, 0.33, ...]  
gradient dW: [-2.5, 0.6, ?, ?, ?, ?, ?, ?, ?]  
loss 1.25347
This is silly. The loss is just a function of $W$:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$
This is silly. The loss is just a function of $W$:

$$L = \frac{1}{N} \sum_{i=1}^{N} L_i + \sum_k W_k^2$$

$$L_i = \sum_{j \neq y_i} \max(0, s_j - s_{y_i} + 1)$$

$$s = f(x; W) = Wx$$

want $\nabla_W L$

Calculus

$$\nabla_W L = ...$$
current W:

\[
\begin{bmatrix}
0.34, \\
-1.11, \\
0.78, \\
0.12, \\
0.55, \\
2.81, \\
-3.1, \\
-1.5, \\
0.33, \\
\end{bmatrix}
\]

loss 1.25347

dW = ...

(some function data and W)

gradien dW:

\[
\begin{bmatrix}
-2.5, \\
0.6, \\
0, \\
0.2, \\
0.7, \\
-0.5, \\
1.1, \\
1.3, \\
-2.1, \\
\end{bmatrix}
\]
Loss gradients

- Denoted as (diff notations): \( \frac{\partial E}{\partial w^{(1)}_{ji}} \) \( \nabla_w L \)

- i.e. how does the loss change as a function of the weights

- We want to change the weights in such a way that makes the loss decrease as fast as possible
Gradient descent

• We’ll update weights
• Move in direction opposite to gradient:

\[ w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)}) \]
Gradient descent

• Iteratively *subtract* the gradient with respect to the model parameters (\(w\))
• I.e. we’re moving in a direction opposite to the gradient of the loss
• I.e. we’re moving towards *smaller* loss
Mini-batch gradient descent

• In classic gradient descent, we compute the gradient from the loss for all training examples
• Could also only use *some* of the data for each gradient update
• We cycle through all the training examples multiple times
• Each time we’ve cycled through all of them once is called an ‘epoch’
• Allows faster training (e.g. on GPUs), parallelization
Learning rate selection

The effects of step size (or “learning rate”)
Gradient descent in multi-layer nets

• We’ll update weights
• Move in direction opposite to gradient:

\[ w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)}) \]

• How to update the weights at all layers?
• Answer: backpropagation of error from higher layers to lower layers
Backpropagation

More info:
https://www.youtube.com/watch?v=QWfmCyLEQ8U&list=PL16j5WbGpaM0_Tj8CRmurZ8Kk1gEBc7fg&index=4

Andrej Karpathy
Backpropagation: Graphic example

First calculate error of output units and use this to change the top layer of weights.

\[ \delta_k = y_k - t_k \]

Update weights into \( j \)

\[ \frac{\partial E}{\partial w_{k,j}^{(2)}} = \delta_k z_j \]

\[ w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)}) \]

Adapted from Ray Mooney, equations from Chris Bishop
Backpropagation: Graphic example

Next calculate error for hidden units based on errors on the output units it feeds into.

\[ \delta_k = y_k - t_k \]

\[ \delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{k,j} \delta_k \]
Finally update bottom layer of weights based on errors calculated for hidden units.

\[
\delta_j = (1 - z_j^2) \sum_{k=1}^{K} w_{kj} \delta_k
\]

Update weights into \( i \)

\[
\frac{\partial E}{\partial w_{ji}^{(1)}} = \delta_j x_i
\]

\[
w^{(\tau+1)} = w^{(\tau)} - \eta \nabla E(w^{(\tau)})
\]

Adapted from Ray Mooney, equations from Chris Bishop
Comments on training algorithm

- Not guaranteed to converge to zero training error, may converge to local optima or oscillate indefinitely.
- However, in practice, does converge to low error for many large networks on real data.
- Thousands of epochs (epoch = network sees all training data once) may be required, hours or days to train.
- To avoid local-minima problems, run several trials starting with different random weights (*random restarts*), and take results of trial with lowest training set error.
- May be hard to set learning rate and to select number of hidden units and layers.
- Neural networks had fallen out of fashion in 90s, early 2000s; back with a new name and significantly improved performance (deep networks trained with dropout and lots of data).
Over-training prevention

- Running too many epochs can result in over-fitting.

- Keep a hold-out validation set and test accuracy on it after every epoch. Stop training when additional epochs actually increase validation error.

Adapted from Ray Mooney
Determining best number of hidden units

- Too few hidden units prevents the network from adequately fitting the data.
- Too many hidden units can result in over-fitting.

- Use internal cross-validation to empirically determine an optimal number of hidden units.
A note on training

- The more weights you need to learn, the more data you need.
- That’s why with a deeper network, you need more data for training than for a shallower network.
- That’s why if you have sparse data, you only train the last few layers of a deep net.

Set these to the already learned weights from another network.
Learn these on your own task.
Effect of number of neurons

more neurons = more capacity

Andrej Karpathy
Effect of regularization

Do not use size of neural network as a regularizer. Use stronger regularization instead:

\[ \lambda = 0.001 \quad \lambda = 0.01 \quad \lambda = 0.1 \]

(you can play with this demo over at ConvNetJS: http://cs.stanford.edu/people/karpathy/convnetjs/demo/classify2d.html)
Hidden unit interpretation

• Trained hidden units can be seen as newly constructed features that make the target concept linearly separable in the transformed space.

• On many real domains, hidden units can be interpreted as representing meaningful features such as vowel detectors or edge detectors, etc.

• However, the hidden layer can also become a distributed representation of the input in which each individual unit is not easily interpretable as a meaningful feature.
Summary

• We use deep neural networks because of their strong performance in practice
• Feed-forward network architecture
• Training deep neural nets
  • We need an objective function that measures and guides us towards good performance
  • We need a way to minimize the loss function: stochastic gradient descent
  • We need backpropagation to propagate error towards all layers and change weights at those layers
• Practices for preventing overfitting