Neural networks

Slides adapted from Stuart Russell
Brains

$10^{11}$ neurons of > 20 types, $10^{14}$ synapses, 1ms–10ms cycle time
Signals are noisy “spike trains” of electrical potential
McCulloch–Pitts “unit”

Output is a “squashed” linear function of the inputs:

\[ a_i \leftarrow g(in_i) = g(\sum j W_{j,i} a_j) \]

A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do.
(a) is a **step function** or **threshold function**

(b) is a **sigmoid** function \(1/(1 + e^{-x})\)

Changing the bias weight \(W_{0,i}\) moves the threshold location
Network structures

Feed-forward networks:
- single-layer perceptrons
- multi-layer perceptrons

Feed-forward networks implement functions, have no internal state

Recurrent networks:
- recurrent neural nets have directed cycles with delays
  ⇒ have internal state (like flip-flops), can oscillate etc.
Feed-forward example

Feed-forward network = a parameterized family of nonlinear functions:

\[ a_5 = g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \]
\[ = g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)) \]

Adjusting weights changes the function: do learning this way!
Single-layer perceptrons

Adjusting weights moves the location, orientation, and steepness of cliff
Expressiveness of perceptrons

Consider a perceptron with $g = \text{step function}$ (Rosenblatt, 1957, 1960). Represents a **linear separator** in input space:

$$\sum_j W_j x_j > 0 \quad \text{or} \quad W \cdot x > 0$$

Can represent AND, OR, NOT, majority, etc.:

- **AND**
  - $W_0 = 1.5$
  - $W_1 = 1$
  - $W_2 = 1$

- **OR**
  - $W_0 = 0.5$
  - $W_1 = 1$
  - $W_2 = 1$

- **NOT**
  - $W_0 = -0.5$
  - $W_1 = -1$

But not XOR:

(a) $x_1 \text{ and } x_2$
(b) $x_1 \text{ or } x_2$
(c) $x_1 \text{ xor } x_2$

Slides adapted from Stuart Russell
Multilayer perceptrons

Layers are usually fully connected; numbers of hidden units typically chosen by hand

- Output units: $a_i$
- Hidden units: $a_j$
- Input units: $a_k$

$W_{j,i}$

$W_{k,j}$

Slides adapted from Stuart Russell
Expressiveness of MLPs

All continuous functions w/ 2 layers, all functions w/ 3 layers

Combine two opposite-facing threshold functions to make a ridge
Combine two perpendicular ridges to make a bump
Add bumps of various sizes and locations to fit any surface
Proof requires exponentially many hidden units
Back-propagation learning

At each epoch, sum gradient updates for all examples and apply

Training curve for 100 restaurant examples: finds exact fit

Total error on training set vs. Number of epochs

Typical problems: slow convergence, local minima
Handwritten digit recognition

3-nearest-neighbor = 2.4% error
400–300–10 unit MLP = 1.6% error
LeNet (1998): 768–192–30–10 unit MLP = 0.9% error
SVMs: ≈ 0.6% error

Current best: 0.24% error (committee of convolutional nets)
Example: ALVINN

steering direction

[Pomerleau, 1995]
Backpropagation

Slides adapted from Kyunghyun Cho
Learning as an Optimization

Ultimately, learning is \((mostly)\)

\[
\theta = \arg \min_{\theta} \frac{1}{N} \sum_{n=1}^{N} c((x_n, y_n) \mid \theta) + \lambda \Omega(\theta, D),
\]

where \(c((x, y) \mid \theta)\) is a per-sample cost function.
Gradient Descent

Gradient-descent Algorithm:

$$\theta^t = \theta^{t-1} - \eta \nabla L(\theta^{t-1})$$

where, in our case,

$$L(\theta) = \frac{1}{N} \sum_{n=1}^{N} l((x_n, y_n) \mid \theta).$$

Let us assume that $$\Omega(\theta, D) = 0.$$
Stochastic Gradient Descent

Often, it is too costly to compute $C(\theta)$ due to a large training set.

Stochastic gradient descent algorithm:

$$\theta^t = \theta^{t-1} - \eta^t \nabla l \left( (x', y') \mid \theta^{t-1} \right),$$

where $(x', y')$ is a randomly chosen sample from $D$, and

$$\sum_{t=1}^{\infty} \eta^t \rightarrow \infty \text{ and } \sum_{t=1}^{\infty} (\eta^t)^2 < \infty.$$

Let us assume that $\Omega(\theta, D) = 0$. 
Almost there . . .

How do we compute the gradient efficiently for neural networks?
Backpropagation Algorithm – (1) Forward Pass

Forward Computation:

\[ L(f(h_1(x_1, x_2, \theta_{h_1}), h_2(x_1, x_2, \theta_{h_2}), \theta_f), y) \]

Multilayer Perceptron with a single hidden layer:

\[ L(x, y, \theta) = \frac{1}{2} (y - U^T \phi(W^T x))^2 \]
Chain rule of derivatives:

\[
\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial f} \frac{\partial f}{\partial x_1} = \frac{\partial L}{\partial f} \left( \frac{\partial f}{\partial h_1} \frac{\partial h_1}{\partial x_1} + \frac{\partial f}{\partial h_2} \frac{\partial h_2}{\partial x_1} \right)
\]
Local derivatives are *shared*:

\[
\frac{\partial L}{\partial x_1} = \frac{\partial L}{\partial f} \left( \frac{\partial f}{\partial h_1} \frac{\partial h_1}{\partial x_1} + \frac{\partial f}{\partial h_2} \frac{\partial h_2}{\partial x_1} \right) \\
\frac{\partial L}{\partial x_2} = \frac{\partial L}{\partial f} \left( \frac{\partial f}{\partial h_1} \frac{\partial h_1}{\partial x_2} + \frac{\partial f}{\partial h_2} \frac{\partial h_2}{\partial x_2} \right)
\]
Each node computes

- **Forward**: \( h(a_1, a_2, \ldots, a_q) \)
- **Backward**: \( \frac{\partial h}{\partial a_1}, \frac{\partial h}{\partial a_2}, \ldots, \frac{\partial h}{\partial a_q} \)
Backpropagation Algorithm – Requirements

Each node computes a differentiable function\(^1\)

Directed Acyclic Graph\(^2\)

---

\(^1\)Well...?

\(^2\)Well...?
Generalized approach to computing partial derivatives

As long as your neural network fits the requirements, you do *not* need to derive the derivatives yourself!

- Theano, Torch, …
Convolutional Neural Networks

(First without the brain stuff)
Convolution Layer

32x32x3 image

32 height

32 width

3 depth
Convolution Layer

32x32x3 image

5x5x3 filter

Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”
Convolutions are a way to extract features from images.

**Convolution Layer**

- **32x32x3 image**
- **5x5x3 filter**

Filters always extend the full depth of the input volume.

**Convolve** the filter with the image i.e. “slide over the image spatially, computing dot products”.

Fei-Fei Li & Andrej Karpathy & Justin Johnson  Lecture 7 - 12   27 Jan 2016
Convolution Layer

32x32x3 image
5x5x3 filter $w$

1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. $5*5*3 = 75$-dimensional dot product + bias)

$w^T x + b$
Convolution Layer

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations

activation map
Convolution Layer

consider a second, green filter

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations

activation maps
For example, if we had 6 5x5 filters, we’ll get 6 separate activation maps:

We stack these up to get a “new image” of size 28x28x6!
Preview: ConvNet is a sequence of Convolution Layers, interspersed with activation functions.

CONV, ReLU
e.g. 6
5x5x3 filters
**Preview:** ConvNet is a sequence of Convolutional Layers, interspersed with activation functions.

```
3 32
CONV, ReLU
  e.g. 6 5x5x3 filters

6 28
CONV, ReLU
  e.g. 10 5x5x6 filters

10 24
CONV, ReLU
```

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

Hubel & Weisel

Topographical mapping

Featural hierarchy

Hyper-complex cells
Complex cells
Simple cells

High level
Mid level
Low level

[From recent Yann LeCun slides]
We call the layer convolutional because it is related to convolution of two signals:

\[ f[x,y] \ast g[x,y] = \sum_{n_1=-\infty}^{\infty} \sum_{n_2=-\infty}^{\infty} f[n_1,n_2] \cdot g[x-n_1,y-n_2] \]

elementwise multiplication and sum of a filter and the signal (image)
A closer look at spatial dimensions:

32x32x3 image
5x5x3 filter

convolve (slide) over all spatial locations
A closer look at spatial dimensions:

7x7 input (spatially)
assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially) 
assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially)
assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially)
assume 3x3 filter
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter

=> 5x5 output
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 2
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 2
A closer look at spatial dimensions:

7x7 input (spatially) assume 3x3 filter applied with stride 2 => 3x3 output!
Pooling layer

- makes the representations smaller and more manageable
- operates over each activation map independently:
MAX 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>

max pool with 2x2 filters and stride 2

<table>
<thead>
<tr>
<th>6</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
What is a word embedding?

Suppose you have a dictionary of words.

The $i^{th}$ word in the dictionary is represented by an embedding:

$$w_i \in \mathbb{R}^d$$

i.e. a $d$-dimensional vector, which is learnt!

- $d$ typically in the range 50 to 1000.
- Similar words should have similar embeddings (share latent features).
- Embeddings can also be applied to symbols as well as words (e.g. Freebase nodes and edges).
- Discuss later: can also have embeddings of phrases, sentences, documents, or even other modalities such as images.
Learning an Embedding Space

Example of Embedding of 115 Countries (Bordes et al., ’11)
How well can we do with a simple CNN?

Collobert-Weston style CNN with pre-trained embeddings from word2vec