Machine Learning

Neural Networks

(slides from Domingos, Pardo, others)
Human Brain
Neurons
Spike (= a brief pulse)
Human Learning

- Number of neurons: $\sim 10^{11}$
- Connections per neuron: $\sim 10^3$ to $10^5$
- Neuron switching time: $\sim 0.001$ second
- Scene recognition time: $\sim 0.1$ second

100 inference steps doesn’t seem much
Machine Learning Abstraction
Artificial Neural Networks

• Typically, machine learning ANNs are very artificial, ignoring:
  – Time
  – Space
  – Biological learning processes

• More realistic neural models exist
  – Hodgkin & Huxley (1952) won a Nobel prize for theirs (in 1963)

• Nonetheless, very artificial ANNs have been useful in many ML applications
Perceptrons

• The “first wave” in neural networks
• Big in the 1960’s
  – McCulloch & Pitts (1943), Woodrow & Hoff (1960), Rosenblatt (1962)
Perceptrons

• Problem def:
  – Let $f$ be a target function from $\mathcal{X} = <x_1, x_2, ...>$ where $x_i \in \{0, 1\}$
    to
    $y \in \{0, 1\}$
  – Given training data $\{(X_1, y_1), (X_2, y_2)...\}$
    • Learn $h(X)$, an approximation of $f(X)$
A single perceptron

\[
\sigma = \begin{cases} 
1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
0 & \text{else}
\end{cases}
\]

Inputs

\[x_0\] Bias \((x_0 = 1, \text{always})\)

\[x_1, x_2, x_3, x_4, x_5\]

\[w_1, w_2, w_3, w_4, w_5, w_0\]
Logical Operators

**AND**

\[ \sigma = \begin{cases} 
1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
0 & \text{else} 
\end{cases} \]

**OR**

\[ \sigma = \begin{cases} 
1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
0 & \text{else} 
\end{cases} \]

**NOT**

\[ \sigma = \begin{cases} 
1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
0 & \text{else} 
\end{cases} \]
Learning Weights

- Perceptron Training Rule
- Gradient Descent
- (other approaches: Genetic Algorithms)

\[
\sigma = \begin{cases} 
1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
0 & \text{else}
\end{cases}
\]
Perceptron Training Rule

- Weights modified for each training example
- Update Rule:

\[ w_i \leftarrow w_i + \Delta w_i \]

where

\[ \Delta w_i = \eta (t - o) x_i \]
Perception Training for NOT

\[ \sigma = \begin{cases} 
1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
0 & \text{else}
\end{cases} \]

\[ w_i \leftarrow w_i + \Delta w_i \]

\[ \Delta w_i = \eta (t - o) x_i \]

Work

Start

End

Bryan Pardo, Machine Learning: EECS 349 Fall 2009
What weights make XOR?

- No combination of weights works
- Perceptrons can only represent linearly separable functions
Linear Separability
Linear Separability
Linear Separability

\[ x_1 + \frac{1}{\sqrt{2}} x_2 \]

\text{XOR}
Perceptron Training Rule

• Converges to the correct classification IF
  – Cases are linearly separable
  – Learning rate is slow enough
  – Proved by Minsky and Papert in 1969

Killed widespread interest in perceptrons till the 80’s
\[ \sigma = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 & \text{else} \end{cases} \]
What’s wrong with perceptrons?

• You can always plug multiple perceptrons together to calculate any function.
• BUT...who decides what the weights are?
  – Assignment of error to parental inputs becomes a problem....
Perceptrons use a step function

\[ \sigma = \begin{cases} 1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\ 0 & \text{else} \end{cases} \]

- Small changes in inputs -> either no change or large change in output.
Solution: Differentiable Function

\[ \sigma = \sum_{i=0}^{n} w_i x_i \]

- Varying any input a little creates a perceptible change in the output
- We can now characterize how error changes \( w_i \) even in multi-layer case
Measuring error for linear units

- Output Function

\[ \sigma (\hat{x}) = \hat{w} \cdot \hat{x} \]

- Error Measure:

\[ E (\hat{w}) = \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \]

- (data, target value, linear unit output)
Gradient Descent

Gradient:

\[ \nabla E[\vec{w}] \equiv \left[ \frac{\partial E}{\partial w_0}, \frac{\partial E}{\partial w_1}, \ldots, \frac{\partial E}{\partial w_n} \right] \]

Training rule:

\[ \Delta \vec{w} = -\eta \nabla E[\vec{w}] \]

\[ \Delta w_i = -\eta \frac{\partial E}{\partial w_i} \]
Gradient Descent Rule

\[
\frac{\partial E}{\partial w_i} \equiv \frac{\partial}{\partial w_i} \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2
\]

\[
= \sum_{d \in D} (t_d - o_d)(-x_{i,d})
\]

Update Rule:

\[
w_i \leftarrow w_i + \eta \sum_{d \in D} (t_d - o_d)x_{i,d}
\]
Gradient Descent for Multiple Layers

\[ \sigma = \sum_{i=0}^{n} w_i x_i \]

We can compute:

\[ \frac{\partial E}{\partial w_{ij}} = \ldots \]
Gradient Descent vs. Perceptrons

- **Perceptron Rule & Threshold Units**
  - Learner converges on an answer ONLY IF data is linearly separable
  - Can’t assign proper error to parent nodes

- **Gradient Descent**
  - (locally) Minimizes error even if examples are not linearly separable
  - Works for multi-layer networks
    - But... *linear units* only make linear decision surfaces (can’t learn XOR even with many layers)
  - And the step function isn’t differentiable...
A compromise function

- **Perceptron**

  \[
  \text{output} = \begin{cases} 
  1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
  0 & \text{else}
  \end{cases}
  \]

- **Linear**

  \[
  \text{output} = \text{net} = \sum_{i=0}^{n} w_i x_i
  \]

- **Sigmoid (Logistic)**

  \[
  \text{output} = \sigma(\text{net}) = \frac{1}{1 + e^{-\text{net}}}
  \]
The sigmoid (logistic) unit

- Has differentiable function
  - Allows gradient descent
- Can be used to learn non-linear functions

\[ \sigma = \frac{1}{1 + e^{-\sum_{i=0}^{n} w_i x_i}} \]
Logistic function

Inputs

- **Age**: 34
- **Gender**: 1
- **Stage**: 4

Coefficients

- **Age**: 0.5
- **Gender**: 0.4
- **Stage**: 0.8

Output

- **Prediction**: 0.6

“Probability of being Alive”

Independent variables

Prediction

$$\sigma = \frac{1}{1 + e^{-\sum_{i=0}^{n} w_i x_i}}$$
Neural Network Model

**Inputs**

- **Age**: 34
- **Gender**: 2
- **Stage**: 4

**Weights**

- Stage: 0.6, 0.2, 0.1
- Gender: 0.3, 0.7, 0.2
- Age: 0.6, 0.4

**Hidden Layer**

- Weighted sum of inputs: 0.6 + 0.2 + 0.1 = 1.9
- Weighted sum of gender: 0.3 + 0.7 + 0.2 = 1.2
- Weighted sum of stage: 0.6 + 0.4 = 1.0

**Output**

- “Probability of being Alive”: 0.6

**Dependent variable**

**Prediction**
Getting an answer from a NN

Inputs

* Age: 34
* Gender: 2
* Stage: 4

Weights

* Independent variables
  * Age: 0.6
  * Gender: 0.1
  * Stage: 0.7

Hidden Layer

* Weights
  * Probability of being Alive: 0.6

Output

* Prediction
  * "Probability of being Alive"
Getting an answer from a NN

Independent variables

<table>
<thead>
<tr>
<th></th>
<th>Weights</th>
<th>Hidden Layer</th>
<th>Weights</th>
<th>Dependent variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>Age</td>
<td>34</td>
<td>.2</td>
<td>.5</td>
<td>Prediction</td>
</tr>
<tr>
<td>Gender</td>
<td>2</td>
<td>.3</td>
<td>.8</td>
<td></td>
</tr>
<tr>
<td>Stage</td>
<td>4</td>
<td>.2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

“Probability of being Alive”

Output 0.6
Getting an answer from a NN

**Independent variables**
- Age: 34
- Gender: 1
- Stage: 4

**Weights**
- Age: 0.6
- Gender: 0.1, 0.3, 0.7
- Stage: 0.2

**Hidden Layer**
- Weight: 0.5
- Weight: 0.8

**Output**
- “Probability of being Alive”: 0.6

**Prediction**
- Dependent variable
Minimizing the Error

Error surface

local minimum

initial error

negative derivative

final error

positive change

$w_{\text{initial}}, w_{\text{trained}}$
Differentiability is key!

• Sigmoid is easy to differentiate

\[
\frac{\partial \sigma(y)}{\partial y} = \sigma(y) \cdot (1 - \sigma(y))
\]

• For gradient descent on multiple layers, a little dynamic programming can help:
  – Compute errors at each output node
  – Use these to compute errors at each hidden node
  – Use these to compute errors at each input node
For each input training example, \( \langle \vec{x}, \vec{t} \rangle \)

1. Input instance \( \vec{x} \) to the network and compute the output \( o_u \) for every unit \( u \) in the network.

2. For each output unit \( k \), calculate its error term \( \delta_k \)

\[
\delta_k \leftarrow o_k (1 - o_k)(t_k - o_k)
\]

3. For each hidden unit \( h \), calculate its error term \( \delta_h \)

\[
\delta_h \leftarrow o_h (1 - o_h) \sum_{k \in \text{outputs}} w_{hk} \delta_k
\]

4. Update each network weight \( w_{ji} \)

\[
w_{ji} \leftarrow w_{ji} + \eta \delta_k x_{ji}
\]
Learning Weights

Inputs

Age 34
Gender 1
Stage 4

Weights

Hidden Layer

Output

“Probability of being Alive”

0.6

Independent variables

Weights

Dependent variable

Prediction

Weights
The fine print

• Don’t implement back-propagation
  – Use a package
  – Second-order or variable step-size optimization techniques exist

• Feature normalization
  – Typical to normalize inputs to lie in \([0,1]\)
    • (and outputs must be normalized)

• Problems with NN training:
  – Slow training times (though, getting better)
  – Local minima
Minimizing the Error

Error surface

initial error

negative derivative

final error

local minimum

$w_{\text{initial}}$, $w_{\text{trained}}$

positive change
Expressive Power of ANNs

• Universal Function Approximator:
  – Given enough hidden units, can approximate *any* continuous function $f$

• Need 2+ hidden units to learn XOR

• Why not use millions of hidden units?
  – Efficiency (training is slow)
  – Overfitting
Overfitting

Real Distribution

Overfitted Model
Combating Overfitting in Neural Nets

• Many techniques

• Two popular ones:
  – Early Stopping
    • Use “a lot” of hidden units
    • Just don’t over-train
  – Cross-validation
    • Test different architectures to choose “right” number of hidden units
Early Stopping

Overfitted model

\( a = \text{validation set} \)

\( b = \text{training set} \)

\( \text{Stopping criterion} \)

\( \min (\Delta \text{error}) \)

\( \text{error}_a \)

\( \text{error}_b \)

Epochs

error
Cross-validation

• Cross-validation: general-purpose technique for model selection
  – E.g., “how many hidden units should I use?”

• More extensive version of validation-set approach.
Cross-validation

- Break training set into $k$ sets
- For each model $M$
  - For $i=1\ldots k$
    - Train $M$ on all but set $i$
    - Test on set $i$
- Output $M$ with highest average test score, trained on full training set
Summary of Neural Networks

When are Neural Networks useful?

- Instances represented by attribute-value pairs
  - Particularly when attributes are real valued
- The target function is
  - Discrete-valued
  - Real-valued
  - Vector-valued
- Training examples may contain errors
- Fast evaluation times are necessary

When not?

- Fast training times are necessary
- Understandability of the function is required
Non-linear regression technique that is trained with gradient descent.

Question: How important is the biological metaphor?
Advanced Topics in Neural Nets

- Batch Move vs. incremental
- Auto-Encoders
- Deep Nets (briefly)
- Neural Networks on Silicon
- Neural Network language models
Incremental vs. Batch Mode

**Incremental mode** Gradient Descent:
Do until satisfied

1. For each training example $d$ in $D$
   1. Compute the gradient $\nabla E_d[\mathbf{w}]$
   2. $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_d[\mathbf{w}]$

**Batch mode** Gradient Descent:
Do until satisfied

1. Compute the gradient $\nabla E_D[\mathbf{w}]$
2. $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla E_D[\mathbf{w}]$

$$E_D[\mathbf{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2$$
Incremental vs. Batch Mode

• In Batch Mode we minimize:

\[ E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \]

• Same as computing: \[ \Delta \vec{w}_D = \sum_{d \in D} \Delta \vec{w}_d \]

• Then setting \[ \vec{w} \leftarrow \vec{w} + \Delta \vec{w}_D \]
Advanced Topics in Neural Nets

- Batch Move vs. incremental
- Auto-Encoders
- Deep Nets (briefly)
- Neural Networks on Silicon
- Neural Network language models
Hidden Layer Representations

- Input->Hidden Layer mapping:
  - representation of input vectors tailored to the task

- Can also be exploited for dimensionality reduction
  - Form of unsupervised learning in which we output a “more compact” representation of input vectors
  - \(<x_1, ..., x_n> \rightarrow <x'_1, ..., x'_m>\) where \(m < n\)
  - Useful for visualization, problem simplification, data compression, etc.
Dimensionality Reduction

Model:

Function to learn:

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000000</td>
<td>10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>00000001</td>
</tr>
</tbody>
</table>
Dimensionality Reduction: Example

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden Values</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>100000000</td>
<td>→ .89 .04 .08</td>
<td>→ 100000000</td>
</tr>
<tr>
<td>010000000</td>
<td>→ .01 .11 .88</td>
<td>→ 010000000</td>
</tr>
<tr>
<td>001000000</td>
<td>→ .01 .97 .27</td>
<td>→ 001000000</td>
</tr>
<tr>
<td>000100000</td>
<td>→ .99 .97 .71</td>
<td>→ 000100000</td>
</tr>
<tr>
<td>000010000</td>
<td>→ .03 .05 .02</td>
<td>→ 000010000</td>
</tr>
<tr>
<td>000001000</td>
<td>→ .22 .99 .99</td>
<td>→ 000001000</td>
</tr>
<tr>
<td>000000100</td>
<td>→ .80 .01 .98</td>
<td>→ 000000100</td>
</tr>
<tr>
<td>000000010</td>
<td>→ .60 .94 .01</td>
<td>→ 000000010</td>
</tr>
<tr>
<td>000000001</td>
<td>→ .60 .94 .01</td>
<td>→ 000000001</td>
</tr>
</tbody>
</table>
Dimensionality Reduction: Example

Sum of squared errors for each output unit
Dimensionality Reduction: Example
Dimensionality Reduction: Example

Weights from inputs to one hidden unit

Graph showing weights from inputs to one hidden unit over time, with values ranging from -5 to 4 on the y-axis and from 0 to 2500 on the x-axis.
Advanced Topics in Neural Nets

- Batch Move vs. incremental
- Auto-encoders
- Deep Nets (briefly)
- Neural Networks on Silicon
- Neural Network language models
Restricted Boltzman Machine

2 layers (hidden & input) of Boolean nodes
Nodes only connected to the other layer
• Setting the hidden nodes to a vector of values updates the visible nodes...and vice versa
Auto-encoders vs. RBMs?

- Similar
- Auto-encoder (AE) goal is to reconstruct input in two steps, input->hidden->output
- RBM defines a probability distribution over $P(x)$
  - Goal is to assign high likelihood to the observed training examples
  - Determining likelihood of a given $x$ actually requires summing over all possible settings of hidden nodes, rather than just computing a single activation as in AE
  - Take EECS 395/495 Probabilistic Graphical Models to learn more
Deep Belief Nets

- A stack of RBNS
- Trained bottom to top with Contrastive Divergence
- Trained AGAIN with supervised training (similar to backprop in MLPs)
Advanced Topics in Neural Nets

- Batch Move vs. incremental
- Auto-Encoders
- Hopfield Nets
- Neural Networks on Silicon
- Neural Network language models
Neural Networks on Silicon

- Currently:

  * Simulation of continuous device physics (neural networks)
  * Digital computation (thresholding)
  * Continuous device physics (voltage)

Why not skip this?
Example: Silicon Retina

Simulates function of biological retina

Single-transistor synapses adapt to luminance, temporal contrast

Modeling retina directly on chip => requires 100x less power!
Example: Silicon Retina

- Synapses modeled with single transistors
Comparison with Mammal Data

- **Real:**

- **Artificial:**
A silicon retina that reproduces signals in the optic nerve

Kareem A Zaghoul and Kwabena Boahen
General NN learning in silicon?

• *Seems* less in-vogue than in late 90s

• In early 2000s, interest turned somewhat to implementing Bayesian techniques in analog silicon
Advanced Topics in Neural Nets

- Batch Move vs. incremental
- Hidden Layer Representations
- Hopfield Nets
- Neural Networks on Silicon
- Neural Network language models
Neural Network Language Models

• Statistical Language Modeling:
  – Predict probability of next word in sequence

I was headed to Madrid , ___

\[ P(___ = \text{“Spain”}) = 0.5, \]
\[ P(___ = \text{“but”}) = 0.2, \text{ etc.} \]

• Used in speech recognition, machine translation, (recently) information extraction
Formally

• Estimate:

\[ P(w_j \mid w_{j-1}, w_{j-2}, \ldots, w_{j-n+1}) \]

\[ = P(w_j \mid h_j) \]
Optimizations

• Key idea – learn simultaneously:
  – vector representations of each word (here 120 dim)
  – predictor of next word. based on previous vectors

• Short-lists
  – Much complexity in hidden->output layer
    • Number of possible next words is large
  – Only predict a *subset* of words
    • Use a standard probabilistic model for the rest
Design Decisions (1)

- Number of hidden units

<table>
<thead>
<tr>
<th>size</th>
<th>400</th>
<th>500</th>
<th>600</th>
<th>1000*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tr. time</td>
<td>11h20</td>
<td>13h50</td>
<td>16h15</td>
<td>11+16h</td>
</tr>
<tr>
<td>Pα alone</td>
<td>100.5</td>
<td>100.1</td>
<td>99.5</td>
<td>94.5</td>
</tr>
<tr>
<td>interp.</td>
<td>68.3</td>
<td>68.3</td>
<td>68.2</td>
<td>68.0</td>
</tr>
<tr>
<td>Werr</td>
<td>13.99%</td>
<td>13.97%</td>
<td>13.96%</td>
<td>13.92%</td>
</tr>
</tbody>
</table>

* Interpolation of networks with 400 and 600 hidden units.
Design Decisions (2)

- Word representation (# of dimensions)
  - They chose 120
Comparison vs. state of the art

• Circa 2005

<table>
<thead>
<tr>
<th></th>
<th>Back-off LM</th>
<th>Neural Network LM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Training data [#words]</td>
<td>600M</td>
<td>4M</td>
</tr>
<tr>
<td>Training time [h/epoch]</td>
<td>2h40</td>
<td>14h</td>
</tr>
<tr>
<td>Perplexity (NN LM alone)</td>
<td>-</td>
<td>103.0</td>
</tr>
<tr>
<td>Perplexity (interpolated LMs)</td>
<td>70.2</td>
<td>67.6</td>
</tr>
<tr>
<td>Word error rate (interpolated LMs)</td>
<td><strong>14.24%</strong></td>
<td>14.02%</td>
</tr>
</tbody>
</table>

* By resampling different random parts at the beginning of each epoch.