Machine Learning

Neural Networks
Human Brain
Neurons
Input-Output Transformation

Input Spikes

Dendrites
Cell body (soma)

Axon hillock
Myelinated axon

Output Spike

Spikes

Spike (= a brief pulse)

Graded EPSP
(Excitatory Post-Synaptic Potential)

Conducted all-or-none spike
(conduction of spike to next cell)

Trigger:
all-or-none spike initiated
### Human Learning

- Number of neurons: \( \sim 10^{11} \)
- Connections per neuron: \( \sim 10^4 \) 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

- Dendrites
- Axon
- Synapses
- Nodes
- Synapses (weights)

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

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

Bias \((x_0 = 1,\text{always})\)
Logical Operators

AND

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

NOT

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

OR

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

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

• Weights modified for each training example
• Update Rule:

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

where

\[ \Delta w_i = \eta(t - o)x_i \]
What weights make XOR?

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

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

XOR

\[ x_1 \quad x_2 \]
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
The XOR gate can be described by the following equation:

\[
\sigma = \begin{cases} 
1 & \text{if } \sum_{i=0}^{n} w_i x_i > 0 \\
0 & \text{else}
\end{cases}
\]
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....
  – This is because of the threshold....
    • Who contributed the error?
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 \(\Rightarrow\) 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**

\[ \sigma(\vec{x}) = \vec{W} \cdot \vec{x} \]

- **Error Measure:**

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

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**

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

- **Linear**

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

- **Sigmoid (Logistic)**

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

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

Output

Prediction

“Probability of being Alive”

0.6
Neural Network Model

Independent variables

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

Weights

- Age: 0.6
- Gender: 0.6
- Stage: 0.6

Hidden Layer

- Summation of weights: 0.6

Output

- “Probability of being alive”: 0.6

Prediction

Dependent variable
Getting an answer from a NN

Inputs

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

Weights

- Independent variables
- Weights
- Hidden Layer
- Weights

Output

- “Probability of being Alive”
- 0.6

Prediction
Getting an answer from a NN

**Independent variables**

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

**Weights**

- Age: 0.2
- Gender: 0.3
- Stage: 0.2

**Hidden Layer**

- Summation: 0.5
- Weight: 0.8

**Dependent variable**

**Prediction**

- Probability of being alive: 0.6

“Probability of being alive”
Getting an answer from a NN

**Inputs**

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

**Independent variables**

- **Weights**
  - Weight to Age: 0.6
  - Weight to Gender: 0.1
  - Weight to Stage: 0.3

**Hidden Layer**

- **Weights**
  - Weight to Hidden Layer: 0.5

**Output**

- **Weight to Output**: 0.8
- **Prediction**: 0.6
- **“Probability of being Alive”**: 0.6

**Dependent variable**

- **Prediction**
Minimizing the Error

- Error surface
- initial error
- negative derivative
- final error
- local minimum
- initial\( w \) \( \rightarrow \) trained\( w \)
- positive change
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
The Backpropagation Algorithm

For each input training example, \( \langle \tilde{x}, \tilde{t} \rangle \)

1. Input instance \( \tilde{x} \) 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**

- 0.6
- 0.1
- 0.3
- 0.7
- 0.2

**Hidden Layer**

- 0.5

**Output**

- 0.6

"Probability of being Alive"

**Independent variables**

**Dependent variable**

**Prediction**
The fine print

• Don’t implement back-propagation
  – Use a package
  – Better 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
  – Local minima
Minimizing the Error

Error surface

local minimum

Error surface

initial error

negative derivative

final error

positive change

$w^\text{initial}$, $w^\text{trained}$
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

\[ \text{error} \]

\[ \text{error}_a \]

\[ \text{error}_b \]

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

Stopping criterion

Epochs

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

\( b = \text{training set} \)
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
Summary of Neural Networks

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
• Hidden Layer Representations
• Hopfield Nets
• Neural Networks on Silicon
• Neural Network language models
Incremental vs. Batch Mode

**Incremental mode** Gradient Descent:  
Do until satisfied  
- For each training example \( d \) in \( D \)  
  1. Compute the gradient \( \nabla E_d[\vec{w}] \)  
  2. \( \vec{w} \leftarrow \vec{w} - \eta \nabla E_d[\vec{w}] \)

**Batch mode** Gradient Descent:  
Do until satisfied  
1. Compute the gradient \( \nabla E_D[\vec{w}] \)  
2. \( \vec{w} \leftarrow \vec{w} - \eta \nabla E_D[\vec{w}] \)  
   \[ E_D[\vec{w}] \equiv \frac{1}{2} \sum_{d \in D} (t_d - o_d)^2 \]
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
- Hidden Layer Representations
- Hopfield Nets
- 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>100000000</td>
<td>10000000</td>
</tr>
<tr>
<td>010000000</td>
<td>01000000</td>
</tr>
<tr>
<td>001000000</td>
<td>00100000</td>
</tr>
<tr>
<td>000100000</td>
<td>00010000</td>
</tr>
<tr>
<td>000010000</td>
<td>00001000</td>
</tr>
<tr>
<td>000001000</td>
<td>00000100</td>
</tr>
<tr>
<td>000000100</td>
<td>00000010</td>
</tr>
<tr>
<td>000000010</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>10000000</td>
<td>→ 0.89 0.04 0.08</td>
<td>→ 10000000</td>
</tr>
<tr>
<td>01000000</td>
<td>→ 0.01 0.11 0.88</td>
<td>→ 01000000</td>
</tr>
<tr>
<td>00100000</td>
<td>→ 0.01 0.97 0.27</td>
<td>→ 00100000</td>
</tr>
<tr>
<td>00010000</td>
<td>→ 0.99 0.97 0.71</td>
<td>→ 00010000</td>
</tr>
<tr>
<td>00001000</td>
<td>→ 0.03 0.05 0.02</td>
<td>→ 00001000</td>
</tr>
<tr>
<td>00000100</td>
<td>→ 0.22 0.99 0.99</td>
<td>→ 00000100</td>
</tr>
<tr>
<td>00000010</td>
<td>→ 0.80 0.01 0.98</td>
<td>→ 00000010</td>
</tr>
<tr>
<td>00000001</td>
<td>→ 0.60 0.94 0.01</td>
<td>→ 00000001</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
Advanced Topics in Neural Nets

- Batch Move vs. incremental
- Hidden Layer Representations
- Hopfield Nets
- Neural Networks on Silicon
- Neural Network language models
Advanced Topics in Neural Nets

• Batch Move vs. incremental
• Hidden Layer Representations
• 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
Luminance Adaptation
Comparison with Mammal Data

• Real:

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

Kareem A Zaghoul¹ and Kwabena Boahen²,³
General NN learning in silicon?

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

• Interest has 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:

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

\[ = P(w_j \mid h_j) \]
Neural Network

Input: $w_{j-n+1}, w_{j-n+2}, w_{j-1}$

Probability estimation:

- $p_1 = P(w_j=1|h_j)$
- $p_i = P(w_j=i|h_j)$
- $p_N = P(w_j=N|h_j)$

Discrete representation: indices in wordlist
Continuous representation: $P$ dimensional vectors

LM probabilities for all words
Optimizations

• **Key idea** – learn simultaneously:
  – vector representations of each word (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>Px alone</td>
<td>100.5</td>
<td>100.1</td>
<td>99.5</td>
<td>94.5</td>
</tr>
<tr>
<td>interpol.</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.

• Almost no difference...
Design Decisions (2)

- Word representation (# of dimensions)

- They chose 120
Comparison vs. state of the art

<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>-</td>
<td>2h40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>14h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>9h40</td>
</tr>
<tr>
<td></td>
<td></td>
<td>12h</td>
</tr>
<tr>
<td></td>
<td></td>
<td>$3 \times 12h$</td>
</tr>
<tr>
<td>Perplexity (NN LM alone)</td>
<td>103.0</td>
<td>97.5</td>
</tr>
<tr>
<td></td>
<td>84.0</td>
<td>80.0</td>
</tr>
<tr>
<td></td>
<td>76.5</td>
<td></td>
</tr>
<tr>
<td>Perplexity (interpolated LMs)</td>
<td>70.2</td>
<td>67.6</td>
</tr>
<tr>
<td></td>
<td>67.9</td>
<td>66.7</td>
</tr>
<tr>
<td></td>
<td>66.5</td>
<td></td>
</tr>
<tr>
<td></td>
<td>65.9</td>
<td></td>
</tr>
<tr>
<td>Word error rate (interpolated LMs)</td>
<td><strong>14.24%</strong></td>
<td>14.02%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.88%</td>
</tr>
<tr>
<td></td>
<td></td>
<td>13.81%</td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>13.75%</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>13.61%</strong></td>
</tr>
</tbody>
</table>

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