Connectionist Models of Learning

Neural Networks
Characterized by:

- A large number of very simple neuronlike processing elements.
- A large number of weighted connections between the elements.
- Highly parallel, distributed control.
- An emphasis on learning internal representations automatically.

Slide CS472 – Machine Learning II 1

Why Neural Nets?
Solving problems under the constraints similar to those of the brain may lead to solutions to AI problems that would otherwise be overlooked.

- Individual neurons operate very slowly.  
  massively parallel algorithms 
- Neurons are failure-prone devices.  
  distributed representations 
- Neurons promote approximate matching.  
  less brittle 

Slide CS472 – Machine Learning II 2
Neural Networks

Rich history, starting in the early forties.
(McCulloch and Pitts 1943)

Two views:

- Modeling the brain.
- “Just” representation of complex functions.
  (Continuous; contrast decision trees.)

Much progress on both fronts.

Drawn interests from: Neuro-science, Cognitive science, AI, Physic, Statistics, and CS / EE.
Learning, Massive Parallelism

There is evidence of learning — plasticity — at synapses.

Complexity arises out of connectivity: Neurons perform highly parallel computation.

Idea: collection of simple cells leads to complex behavior: thought, action, and consciousness . . .

<table>
<thead>
<tr>
<th>Computational Resources in Computers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Computer</strong></td>
</tr>
<tr>
<td>1 CPU, $10^5$ gates</td>
</tr>
<tr>
<td>$10^6$ bits RAM, $10^{10}$ bits disk</td>
</tr>
<tr>
<td>$10^{-8}$ sec</td>
</tr>
<tr>
<td>$10^9$ bits/sec</td>
</tr>
<tr>
<td>$10^5$</td>
</tr>
<tr>
<td><strong>Human Brain</strong></td>
</tr>
<tr>
<td>$10^{11}$ neurons</td>
</tr>
<tr>
<td>$10^{11}$ neurons, $10^{14}$ synapses</td>
</tr>
<tr>
<td>$10^{-3}$ sec</td>
</tr>
<tr>
<td>$10^{14}$ bits/sec</td>
</tr>
<tr>
<td>$10^{14}$</td>
</tr>
</tbody>
</table>

Slide CS472 – Machine Learning II 5

Slide CS472 – Machine Learning II 6
Artificial Neural Networks

\[ a_i = g(in_i) \]

Slide CS472 – Machine Learning II 7

Activation Functions

(a) Step function  (b) Sign function  (c) Sigmoid function

Slide CS472 – Machine Learning II 8
Example

\[ \sum_{i=1}^{n} w_i \]

AND

OR

NOT

Slide CS472 – Machine Learning II 9

Can Simulate Boolean Gates

Slide CS472 – Machine Learning II 10
Perceptron Learning Algorithm

Remarkable learning algorithm: (Rosenblatt 1960)
if function can be represented by perceptron,
then learning algorithm is guaranteed to quickly converge
to the hidden function!

Enormous popularity in early to mid 60’s

But analysis by Minsky and Papert (1969)
showed certain simple functions cannot be represented
(Boolean XOR)
Killed the field! (and possibly Rosenblatt (rumored)).
Linearly Separable Functions Only

(a) $I_1$ and $I_2$

(b) $I_1$ or $I_2$

(c) $I_1$ xor $I_2$

Learning Threshold Values

Slide CS472 – Machine Learning II 13

Slide CS472 – Machine Learning II 14
Perceptron Learning

A perceptron can learn any linearly separable function, given
enough enough training examples.

Key idea: adjust weights till all examples correct.
Update weights repeatedly (epochs) for each example.

High Level Algorithm

function Neural-Network-Learning(examples) returns network

\[ \text{network} \leftarrow \text{a network with randomly assigned weights} \]

repeat
  for each e in examples do
    O ← Neural-Network-Output(network, e)
    T ← the observed output values from e
    update the weights in network based on e, O, and T
  end
until all examples correctly predicted or stopping criterion is reached
return network

Slide CS472 – Machine Learning II 15

Slide CS472 – Machine Learning II 16
Weight Update Function

Single output $O$; target output for example $T$.

Define error: $Err = T - O$

Now, just move weights in right direction!

- If error is positive, then need to increase $O$.
- Each input unit $j$ contributes $W_j I_j$ to total input.
  - if $I_j$ is positive, increasing $W_j$ tends to increase $O$
  - if $I_j$ is negative, decreasing $W_j$ tends to increase $O$

So, use: $W_j \leftarrow W_j + \alpha \times I_j \times Err$

Perceptron learning rule (Rosenblatt 1960). $\alpha$ is learning rate.

Slide CS472 – Machine Learning II 17

Rule is intuitively correct.

**Gradient descent through weight space.**

Surprise is **proof** of convergence.

Weight space has **no local minima**.

With enough examples, it will find the function.

(provided $\alpha$ not too large)

Explains early popularity.

Slide CS472 – Machine Learning II 18
Consider learning the logical “or” function.
Our examples are:

<table>
<thead>
<tr>
<th>example</th>
<th>x_1</th>
<th>x_2</th>
<th>x_3</th>
<th>l</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>-1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>1</td>
<td>-1</td>
<td>1</td>
</tr>
</tbody>
</table>

Slide CS472 – Machine Learning II 19

We’ll use a single perceptron with three inputs 
(x_1, x_2, x_3) and single output (l).
Learning rate of 0.5.
Note artificial input x_3 fixed at -1.

[Step-by-step walk-through is in separate handout.]
Slide CS472 – Machine Learning II 21

Slide CS472 – Machine Learning II 22
Complex Feedforward Nets for Classification

Feedforward, layered, fully connected

Backpropagation Procedure

1. Present all training instances. For each one,
   (a) Calculate actual output. (forward pass)
   (b) Compute the weight changes. (backward pass)
      i. Calculate error at output nodes. Compute adjustment to weights from hidden layer to output layer accordingly.
      ii. Calculate error at hidden layer. Compute adjustment to weights from initial layer to hidden layer accordingly.
2. Add up weight changes and change the weights.
Requires a Smooth Threshold Function

Because backpropagation updates all weights simultaneously, stair-step threshold function won’t work.

Adjusting the Weights

Make a large change to a weight, \( w \), if the change leads to a large reduction in the errors observed at the output nodes.

\[
d = \text{desired value at output nodes} \\
o = \text{actual value at output nodes} \\
\text{error} = d - o
\]
Adjusting the Weights

Let change in $w_{i\rightarrow j}$ be proportional to

- the slope of the threshold function at $j$
- the output at node $i$
- degree of error at $j$ (benefit)
  
  $\beta_z = d_z - o_z$
  
  $\beta_j = \sum_k w_{j\rightarrow k} o_k (1 - o_k) \beta_k$

- learning rate $r$

Change to $w_{i\rightarrow j}$ should be proportional to $o_i o_j (1 - o_j) \beta_j$.

Slide CS472 – Machine Learning II 27

The Backpropagation Procedure

Pick a rate parameter $r$.

Until performance is satisfactory,

For each training instance,
- Compute the resulting output.
- Compute $\beta = d_z - o_z$ for nodes in the output layer.
- Compute $\beta = \sum_k w_{j\rightarrow k} o_k (1 - o_k) \beta_k$ for all other nodes.
- Compute weight changes for all weights using

  $\Delta w_{i\rightarrow j} = r \cdot o_i o_j (1 - o_j) \beta_j$

Add up weight changes for all training instances, and change the weights.

Slide CS472 – Machine Learning II 28
Backpropagation Algorithm (Mitchell)

Initialize all weights to small random numbers. Until satisfied, do

For each training example, do

- Input the training example to the network and compute the network outputs
- For each output unit $k$
  \[ \delta_k \leftarrow o_k(1 - o_k)(t_k - o_k) \]
- For each hidden unit $h$
  \[ \delta_h \leftarrow o_h(1 - o_h) \sum_{k \in \text{outputs}} w_{h,k} \delta_k \]
- Update each network weight $w_{i,j}$
  \[ w_{i,j} \leftarrow w_{i,j} + \Delta w_{i,j} \text{ where } \Delta w_{i,j} = \eta \delta_j x_{i,j} \]

Slide CS472 – Machine Learning II 29
Learning Hidden Layer Representations

<table>
<thead>
<tr>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>1000000</td>
</tr>
<tr>
<td>0100000</td>
<td>0100000</td>
</tr>
<tr>
<td>0010000</td>
<td>0010000</td>
</tr>
<tr>
<td>0001000</td>
<td>0001000</td>
</tr>
<tr>
<td>0000100</td>
<td>0000100</td>
</tr>
<tr>
<td>0000010</td>
<td>0000010</td>
</tr>
<tr>
<td>0000001</td>
<td>0000001</td>
</tr>
</tbody>
</table>

Slide CS472 – Machine Learning II 31

Learning Hidden Layer Representations

<table>
<thead>
<tr>
<th>Input</th>
<th>Hidden Values</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000000</td>
<td>.89 .04 .08</td>
<td>1000000</td>
</tr>
<tr>
<td>0100000</td>
<td>.01 .11 .88</td>
<td>0100000</td>
</tr>
<tr>
<td>0010000</td>
<td>.01 .97 .27</td>
<td>0010000</td>
</tr>
<tr>
<td>0001000</td>
<td>.99 .97 .71</td>
<td>0001000</td>
</tr>
<tr>
<td>0000100</td>
<td>.03 .05 .02</td>
<td>0000100</td>
</tr>
<tr>
<td>0000010</td>
<td>.22 .99 .99</td>
<td>0000010</td>
</tr>
<tr>
<td>0000001</td>
<td>.80 .01 .98</td>
<td>0000001</td>
</tr>
<tr>
<td></td>
<td>.60 .94 .01</td>
<td></td>
</tr>
</tbody>
</table>

Slide CS472 – Machine Learning II 32
Hidden Units

- **Hidden units** are nodes that are situated between the input nodes and the output nodes.
- Hidden units allow a network to learn non-linear functions.
- Hidden units allow the network to represent combinations of the input features.
- Given too many hidden units, a neural net will simply memorize the input patterns.
- Given too few hidden units, the network may not be able to represent all of the necessary generalizations.

Slide CS472 – Machine Learning II 33

When to Consider Neural Networks

- Input is high-dimensional discrete or real-valued (e.g. raw sensor input)
- Output is discrete, real-valued, or a vector of values
- Possibly noisy data
- Form of target function is unknown
- Human readability of result is unimportant

Slide CS472 – Machine Learning II 34
More on Backpropagation

- Gradient descent over entire network weight vector
- Easily generalized to arbitrary directed graphs
- Will find a local, not necessarily global error minimum
  - In practice, often works well (can run multiple times)
- Minimizes error over training examples
  - Will it generalize well to subsequent examples?
- Training can take thousands of iterations → slow!
- Using network after training is very fast

Expressive Capabilities of ANNs

Boolean functions:
- Every boolean function can be represented by network with single hidden layer
- but might require exponential (in number of inputs) hidden units

Continuous functions:
- Every bounded continuous function can be approximated with arbitrarily small error, by network with one hidden layer [Cybenko 1989; Hornik et al. 1989]

Any function can be approximated to arbitrary accuracy by a network with two hidden layers [Cybenko 1988].
Momentum

$$\Delta w_{ij}(t + 1) = r o_i o_j (1 - o_j) \beta_j + \alpha [w_{ij}(t) - w_{ij}(t - 1)]$$

- A momentum factor, $\alpha$, makes the $n^{th}$ weight change partially dependent on the $(n - 1)^{th}$ weight change. $\alpha$ ranges between 0 and 1.
- Momentum tends to keep the weight moving in the same direction, thereby improving convergence.
- Tends to increase the step size in regions where the gradient is unchanging, speeding convergence.
- Tends to avoid getting caught in small local minima and in oscillations about local minima.

Slide CS472 – Machine Learning II 37

How many training pairs are needed?

This is a difficult question and depends on the problem, the training examples, and the network architecture. But a good rule of thumb is:

$$\frac{W}{P} = e$$

where $W$ = # weights; $P$ = # training pairs; $e$ = error rate

For example, for $e = 0.1$, a net with 80 weights will require 800 training patterns to be assured of getting 90% of the test patterns correct (assuming it got 95% of the training patterns correct).

Slide CS472 – Machine Learning II 38
How long should you train the net?

- The goal is to achieve a balance between correct responses for the training patterns and correct responses for new patterns. (That is, a balance between memorization and generalization.)

- If you train the net for too long, then you run the risk of overfitting.

- In general, the network is trained until it reaches an acceptable error rate (e.g., 95%).

Overfitting in ANNs

![Error versus weight updates (example 1)](error_vs_weight_updates_example.png)

Slide CS472 – Machine Learning II 39

Slide CS472 – Machine Learning II 40
Overfitting in ANNs

![Error versus weight updates (example 2)](image)

Slide CS472 – Machine Learning II 41

Implementing Backprop – Design Decisions

1. Choice of $r$
2. Network architecture
   (a) How many hidden layers? how many hidden units per layer?
   (b) How should the units be connected? (Fully? Partial? Use domain knowledge?)
3. Stopping criterion – when should training stop?

Slide CS472 – Machine Learning II 42
Recurrent Networks

![Diagram of Recurrent Networks](https://via.placeholder.com/150)

Reinforcement Learning

So far, we had a well-defined set of training examples.

What if feedback is not so clear?

E.g., when playing a game, only after many actions final result: win, loss, or draw.

Issue: learning via **delayed rewards / delayed feedback**

Main success: Tesauro’s backgammon player (TD Gammon)

Start from random play; millions of games

World-level performance (changed game itself)
Imagine agent wandering around in environment.

How does it learn utility values of each state?

(i.e., what are good / bad states? avoid bad ones...)

Reinforcement learning will tell us how! Variations:

- environment accessible or inaccessible
- have model of environment and effects of action...or not
- rewards in terminal states only; or in any state
- agent can be passive (watch) or active (explore)

---

Reinforcement Learning for Backgammon

In backgammon: states = boards.

Only clear feedback in final states (win/loss).

We want to know utility of the other states.

Intuitively: utility = chance of winning

At first, we only know this for the end states.

Reinforcement learning: computes for intermediate states. Play by moving to maximum utility states!

back to simplified world...
Passive Learning in a Known, Accessible Environment

Agent just wanders from state to state.
Each transition is made with a fixed probability.

Initially: only two known reward positions:
- State (4,2) — a loss / poison / reward $-1$ (utility)
- State (4,3) — a win / food / reward $+1$ (utility)

How does the agent learn about the utility, i.e.,
**expected value**, of the other states?
Three strategies:
(a) “Sampling” (Adaptive control theory)
   Naive updating
(b) “Calculation” / “Equation solving”
   Adaptive dynamic programming
(c) “in between (a) and (b)”
   Temporal Difference Learning — TD learning used for backgammon
Naive updating (LMS approach)

Widrow and Hoff [1960]

(a) “Sampling” — agent makes random runs through environment; collect statistics on final payoff for each state (e.g. when at (2,3), how often do you reach +1 vs. -1?)

Learning algorithm keeps a running average for each state. Provably converges to true expected values (utilities).

function LMS-UPDATE(U, e, percepts, M, N) returns an updated U

if TERMINAL?[e] then reward-to-go ← 0
for each e, in percepts (starting at end) do
    reward-to-go ← reward-to-go + REWARD[e]
    U[STATE[e]] ← RUNNING-AVERAGE(U[STATE[e]], reward-to-go, N[STATE[e]])
end
Utility estimates

Number of epochs

RMS error in utility

Number of epochs
Problems

Ignores structure of transitions which impose strong additional constraints.

*The actual utility of a state is constrained to be the probability-weighted average of its successors’ utilities, plus its own reward.*

Main effect: *slow convergence.*

Example where LMS does poorly
Adaptive dynamic programming

Consider $U(3, 3)$

From figure we see that

$$U(3, 3) = 0.33 \times U(4, 3) + 0.33 \times U(2, 3) + 0.33 \times U(3, 2)$$

$$= 0.33 \times 1.0 + 0.33 \times 0.0886 + 0.33 \times -0.4430$$

$$= 0.2152$$

Check e.g. $U(3, 1)$ yourself.

Slide CS472 – Machine Learning II 57

Utilities follow basic laws of probabilities:

write down equations; solve for unknowns.

Utilities follow from:

$$U(i) = R(i) + \sum_j M_{i,j} U(j) \quad (\ast)$$

(note: $i, j$ over states.)

$R(i)$ is the reward associated with being in state $i$.

(often non-zero for only a few end states)

$M_{i,j}$ is the probability of transition from state $i$ to $j$.

Slide CS472 – Machine Learning II 58
Dynamic programming style methods can be used to solve the set of equations.

Major drawback: number of equations and number of unknowns.

E.g. for backgammon: roughly $10^{50}$ equations with $10^{50}$ unknowns. Infeasibly large.

Temporal difference learning

Combine “sampling” with “calculation”

Or stated differently: TD-learning uses a sampling approach to solve the set of equations.

*Consider the transitions, observed by a wandering agent.*

*Use the observed transitions to adjust the utilities of the observed states to bring them closer to the constraint equations.*
Temporal difference learning

When observing a transition from $i$ to $j$,
   bring $U(i)$ value closer to that of $U(j)$
Use update rule:

$$U(i) \leftarrow U(i) + \alpha(R(i) + U(j) - U(i)) \quad (**)$$

$\alpha$ is the learning rate parameter
rule is called the temporal-difference or TD
   equation (because we take the difference in utilities between successive states).

Slide CS472 – Machine Learning II 61

---

function TD-UPDATE($U, e, \text{percepts}, M, N$) returns the utility table $U$

if TERMINAL[$e$] then
    $U[\text{STATE}[e]] \leftarrow \text{RUNNING-AVERAGE}(U[\text{STATE}[e]], \text{REWARD}[e], N[\text{STATE}[e]])$
else if \text{percepts} contains more than one element then
    $e' \leftarrow$ the penultimate element of \text{percepts}
    $i, j \leftarrow \text{STATE}[e'], \text{STATE}[e]$
    $U[i] \leftarrow U[i] + \alpha(N[i]) \text{REWARD}[e'] + U[j] - U[i]$

Slide CS472 – Machine Learning II 62
At first blush, the rule:
\[ U(i) \leftarrow U(i) + \alpha(R(i) + U(j) - U(i)) \] (**) may appear to be a bad way to solve/approximate:
\[ U(i) = R(i) + \sum_j M_{i,j} U(j) \] (*)

Note that (**) brings \( U(i) \) closer to \( U(j) \) but
in (*) we really want the **weighted** average
over the neighboring states!

Issue resolves itself, because over time, we **sample**
from the transitions out of \( i \). So, successive applications
of (**) average over neighboring states.
(keep \( \alpha \) appropriately small)

---

**Performance**

Runs noisier than Naive Updating (averaging),
but smaller error.
In our 4x3 world, we get a root-mean-square error of less
than 0.07 after 1000 examples.

Also, note that compared to Adaptive Dynamic Programming
we only deal with observed states during sample runs.
I.e., in backgammon consider only a few hundreds of thousands
of states out of \( 10^{30} \). Represent utility function
**implicitly** (no table) in neural network.
Slide CS472 – Machine Learning II 65

Slide CS472 – Machine Learning II 66
Reinforcement learning is a very rich area of study.

*In some sense, touches on much of the core of AI.*

“How does an agent learn to take the right actions in its environment?”

In general, pick action that leads to state with highest utility as learned so far.

---

**Extensions**

— **Active learning** — exploration.

   now and then make new (non utility optimizing move)

— **Learning action-value functions**

   \(Q(a, i)\) denotes value of taking action \(a\) in state \(i\)

   we have: \(U(i) = \max_a Q(a, i)\)

— **Generalization in reinforcement learning**

   Use implicit representation of utility function
   e.g. a neural network as in backgammon.

   Input nodes encode board position;
       activation of output node gives utility.