CS 6375 Machine Learning: Artificial Neural Networks (ANNs)

Instructor: Yang Liu

Slides adopted from Prof. Spears*

*Math significant portions of this lecture are taken from *Artificial Intelligence: A Modern Approach* by Russell and Norvig, as well as the course textbook on ML by Mitchell.

---

**Math Review**

**Dot product:**

\[ \mathbf{v} = [v_1, v_2, \ldots, v_n]; \quad \mathbf{w} = [w_1, w_2, \ldots, w_n] \]

\[ \mathbf{v} \cdot \mathbf{w} = v_1 w_1 + v_2 w_2 + \ldots + v_n w_n \]

**Partial derivative:**

\[ \frac{\partial f (v_1, v_2, \ldots, v_n)}{\partial v_i} \text{ Hold all } v_j \text{ where } j \neq i \text{ fixed, then take the derivative.} \]

**Chain rule for derivatives:**

*(the chain rule can also be applied if these are partial derivatives)*

\[ \frac{df (g (x))}{dx} = \frac{df}{dg} \frac{dg}{dx} \]
Gradient Descent

- The gradient points in the direction of steepest ascent of the function $f$.

- Use gradient descent to find variables minimizing objective function, rather than trying to find close form solution

$$ w_i = w_i + \Delta w_i $$

Artificial Neural Network

- Can be considered as simplified mathematical models of brain-like systems and they function as parallel distributed computing networks.

- Started >50 years ago. There’s a rebirth in interest in neural computing in 1980s (developed mathematic foundations, learning algorithms)

- Recently `deep learning` is a hot topic

- Used in vision, speech, decision making, signal processors.
Multi-layer Feedforward Neural Network

Properties of ANNs

- Many neuron-like threshold switching units.
- Many weighted interconnections between units.
- Highly parallel, distributed computation.
- Weights are tuned automatically (training).
- Especially useful for learning complex functions with continuous-valued outputs and large numbers of noisy inputs, which is the type that logic-based techniques have difficulty with.
Digit Recognition Example

ANNs as Simple Computing Elements

- Each unit (node) receives signals from its input links and computes a new activation level that it sends along all output links.
- Computation is split into two steps:
  - $i_n = \sum W_{j,i} x_j = \bar{W} \cdot \bar{x}$ (vector dot product), the linear step
  - $x_i = g(i_n)$, the nonlinear step.
Purpose of the Activation Function $g$

- We want the unit to be “active” (near +1) when the “right” inputs are given.
- We want the unit to be “inactive” (near 0) when the “wrong” inputs are given.
- It’s preferable for $g$ to be nonlinear. Otherwise, the entire neural network collapses into a simple linear function.
Possibilities for \( g \)

Step (threshold) function

\[
\text{step}(x) = \begin{cases} 
1, & \text{if } x > \text{threshold} \\
0, & \text{if } x \leq \text{threshold}
\end{cases}
\]

(in picture above, threshold = 0)

Sign function

\[
\text{sign}(x) = \begin{cases} 
+1, & \text{if } x > 0 \\
-1, & \text{if } x \leq 0
\end{cases}
\]

Sigmoid (logistic or squashing) function

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

(“soft” threshold)

Adding an extra input \( x_0 = -1 \) and weight \( W_{0,j} = t \) (called the bias weight) is equivalent to having a threshold at \( t \). This way we can always assume a 0 threshold.

Using a Bias Weight to Standardize the Threshold

\[
W_1x_1 + W_2x_2 > t
\]

\[
W_1x_1 + W_2x_2 - t > 0
\]
Implementing AND

Assume Boolean (0/1) input values...

\[ o(x_1, x_2) = 1 \text{ if } -1.5 + x_1 + x_2 > 0 \]
\[ = 0 \text{ otherwise} \]

Implementing OR

\[ o(x_1, x_2) = 1 \text{ if } -0.5 + x_1 + x_2 > 0 \]
\[ = 0 \text{ otherwise} \]
Implementing NOT

\[ o(x_1) = 1 \quad \text{if} \quad 0.5 - x_1 > 0 \]
\[ = 0 \quad \text{otherwise} \]

Implementing More Complex Boolean Functions

\[ o(x_1) = 1 \quad \text{if} \quad 0.5 - x_1 > 0 \]
\[ = 0 \quad \text{otherwise} \]
Types of ANNs

- **Feedforward**: Links are unidirectional, and there are no cycles, i.e., the network is a directed acyclic graph (DAG). Units are arranged in layers, and each unit is linked only to units in the next layer.

- **Recurrent**: Links can form arbitrary topologies. Cycles can implement memory. Behavior can become unstable, oscillatory, or chaotic.

A Feedforward Network Topology

- **Multilayer feedforward network**

\[
\begin{align*}
h_j &= 1 \text{ if } \sum_k W_{kj} x_k > 0 \quad & h_j &= 0 \text{ otherwise} \\
o_i &= 1 \text{ if } \sum_j W_{ji} h_j > 0 \quad & o_i &= 0 \text{ otherwise}
\end{align*}
\]

assuming a step function

Layer of input units
Layer of hidden units
Layer of output units
Learning in ANNS

- The topology (nodes and connections) of the network is often assumed to be fixed. The weights are learned/updated.
- If the topology is not assumed to be fixed, then genetic algorithms can be used to learn it.

Perceptrons: Single-layer networks (no hidden layer)
Perceptrons

- **Perceptrons** are single-layer feedforward networks
- Each output unit is independent of the others
- **Can assume a single output unit**
- Activation of the output unit is calculated by:

\[ o = g \left( \sum_j W_j x_j \right) \]

where \( x_j \) is the activation of input unit \( j \), and we assume an additional weight and input to represent the threshold.

Expressive Limits of Perceptrons

- Already seen perceptrons for AND, OR, NOT.
- Can the XOR function be represented by a perceptron?

\[ o(x_1, x_2) = -w_0 + 0 \cdot w_1 + 0 \cdot w_2 \leq 0 \]
\[ -w_0 + 0 \cdot w_1 + 1 \cdot w_2 > 0 \]
\[ -w_0 + 1 \cdot w_1 + 0 \cdot w_2 > 0 \]
\[ -w_0 + 1 \cdot w_1 + 1 \cdot w_2 \leq 0 \]

There is no assignment of values to \( w_0, w_1 \) and \( w_2 \) that satisfies above inequalities. **XOR cannot be represented!**
What Can be Represented Using Perceptrons?

Representation Theorem: 1-layer feedforward networks (perceptrons) can only represent linearly separable functions. That is, the decision surface separating positive from negative examples has to be a plane. Examples: AND, OR, NOT.

Perceptron Learning

- We do not necessarily assume here that the perceptrons have threshold units. They may have sigmoid units.
- What is the space of hypotheses being searched during learning?
  - The set of all possible weight vectors.
- The goal is to adjust the network weights to minimize the error on the training set.
  - The error is computed by comparing the output of the ANN with the correct class of the training examples.
  - Error is a function of the weights, use gradient descent to update weights: $W_i = W_i + \Delta W_i$
Perceptron Learning

- The algorithm consists of running the training examples through the ANN one at a time, adjusting the weights slightly after each example [note: or adjust weights using all the examples, or some examples]
- Each cycle through the set of all examples is called an epoch.
- Epochs are typically repeated until the weight changes are below some small $\epsilon$. In other words, ANN training typically requires multiple training epochs over the same set of training examples.

Definition of Error:
Sum of Squared Errors

$$E = \frac{1}{2} \sum_{\text{examples}} (t - o)^2$$

Here, $t$ is the correct (desired) output and $o$ is the actual output of the neural net.

Introduce $Err$ to simplify the math on the following slides:

$$Err = (t - o)$$

$$E = \frac{1}{2} (t - o)^2 = \frac{1}{2} Err^2$$

assume one example for now
Reduction of Squared Error

Gradient descent reduces the squared error by calculating the partial derivative of $E$ with respect to each weight:

$$
\nabla E = \frac{\partial E}{\partial W_j} = \frac{\partial E}{\partial \text{Err}} \times \frac{\partial \text{Err}}{\partial W_j} = \text{Err} \times \frac{\partial \text{Err}}{\partial W_j}
$$

This is called “in”

$$
= \text{Err} \times \frac{\partial}{\partial W_j} \left( t - g \left( \sum_{i=0}^{n} W_i x_i \right) \right)
$$

The weight is updated by $\eta$ times this gradient of error $\nabla E$ in weight space.

The learning rate, $\eta$, is typically set to a small value such as 0.1

Weight Updates

On the previous slide, we showed how to calculate the weight updates:

$$
W_j \leftarrow W_j + \Delta W_j \quad \text{where} \quad \Delta W_j = \eta \times \text{Err} \times g'(\text{in}) \times x_j
$$

IF the error is positive, we need to increase the output $o$ to make it closer to $t$. Adding $\Delta W_j$ does this (because $\text{Err}$ is positive in this case).

IF the error is negative, we need to decrease the output $o$ to make it closer to $t$. Adding $\Delta W_j$ does this (because $\text{Err}$ is negative in this case).

In both cases, we add $\Delta W_j$, which reduces the error.
General Perceptron Learning Algorithm

- Inputs: training set \( \{(x_1, x_2, \ldots, x_n, t)\} \)

- Method
  - Randomly initialize weights \( W_j \), -0.5 \( \leq W_j \leq 0.5 \)
  - Repeat for several epochs until convergence:
    - For each example in training set do:
      \[
      in \leftarrow \left( \sum_{k=0}^{n} W_k x_k \right)
      \]
  - Calculate network output \( o = g(in) \)
  - Adjust weights:
    \[
    Err \leftarrow t - g(in)
    W_j \leftarrow W_j + \eta \times Err \times g'(in) \times x_j
    \]

Convergence

- Convergence to correct target concept guaranteed if:
  - Training data linearly separable
  - \( \eta \) is sufficiently small

- Why does it work?
  - For perceptrons, the error surface in weight space has a single global minimum and no local minima. Gradient descent is guaranteed to find the global minimum, provided the learning rate is not so big that you overshoot it.
A Simpler Version

- For **threshold** functions (step, sign), the g function is not differentiable. In this case the factor g'(in) is omitted from the weight update, and the algorithm becomes the following…
The Perceptron Learning Algorithm with a **Threshold** Function $g$

- **Inputs**: training set $\{(x_1, x_2, \ldots, x_n, t)\}$
- **Method**
  - Randomly initialize weights $W_j$
  - Repeat for several epochs until convergence:
    - For each example:
      - Calculate network output $o = g(\text{in})$.
      - Adjust weights:

Stochastic Gradient Descent

- The algorithm described is **online learning** (incremental learning)
- **Original gradient descent**:
  - minimizing error for the entire training set
  - get average weight update
  - learning is slow for large training set
- **Stochastic gradient descent** uses mini-batch:
  - use a small number of randomly chosen instances (extreme case is online learning, just use one example a time)
Multi-Layer Artificial Neural Networks

Learning in Multi-layer Networks

- Multi-layer feedforward networks are trainable by backpropagation (i.e., using the delta training rule) provided the activation function \( g \) is a differentiable function.
  - Threshold units don't qualify, but the “logistic” (also called “sigmoid” or “squashing”) function does.
Weight Adjustment with Backprop

- Learning is similar to perceptrons:
  - Example inputs are presented to the network. If the network computes an output vector that matches the target, nothing is done. (Note that we generalize now to multiple output nodes.)
  - If there is a difference between output and target (i.e., an error), then the weights are adjusted to reduce this error.
  - The key is to assess the blame for the error and divide it among the contributing weights.

Weight Adjustment with Backprop

- The error term \((t - o)\) is known for the units in the output layer. To adjust the weights between the hidden and the output layer, we can use the gradient descent rule as done for perceptrons.
- To adjust weights between the input and hidden layer, we need some way of estimating the errors made by the hidden units.
Let’s Rename Nodes for Backprop

- Multilayer feedforward network

Learning the Weights Between Hidden and Output

Recall the perceptron learning rule:

$$ W_j \leftarrow W_j + \eta \times Err \times g'(in) \times x_j $$

This is what we use, but modify the notation:

- We add subscript $i$ for output node $i$

$$ \delta_i = Err_i \times g'(in_i) $$

$$ W_{ji} \leftarrow W_{ji} + \eta \times a_j \times \delta_i $$
Error Back-Propagation to Learn Weights Between Input and Hidden

- **Key idea**: each hidden node is responsible for some fraction of the error in each of the output nodes. This fraction equals the strength of the connection (weight) between the hidden node and the output node.

\[ \delta_j = \text{error at hidden node } j = g'(i_j) \sum_{i \in \text{outputs}} W_{ji} \delta_i \]

where \( \delta_i \) is the defined error at output node \( i \).

(previous slide)

Can also use chain rule to calculate derivative

Learning Between Input and Hidden

The update rule is now the standard, with notation adjusted to suit the situation:

\[ W_{kj} \leftarrow W_{kj} + \eta \times a_k \times \delta_j \]
The Backpropagation Algorithm for Three-Layer Networks with Sigmoid Units

- Initialize all weights in the network to small random numbers.
- Until weights converge (may take thousands of iterations) do
  - For each training example
    - Compute network output vector \( \mathbf{o} \) (by doing feedforward)
    - For each output unit \( i \) do
      - \( \delta_i = o_i(1 - o_i)(t_i - o_i) \)
    - For each hidden unit \( j \) do
      - \( \delta_j = o_j(1 - o_j) \sum_{i \in \text{outputs}} W_{ji} \delta_i \)
    - Update each network weight, hidden layer to output
      - \( \forall j, i, \ W_{ji} \leftarrow W_{ji} + \eta \times a_j \times \delta_i \)
    - Update network weight from each input \( k \) to hidden \( j \)
      - \( \forall k, j, \ W_{kj} \leftarrow W_{kj} + \eta \times a_k \times \delta_j \)

Weight Learning Issues

- When to terminate backprob? Overfitting and early stopping
  - After a fixed number of iterations
  - When training error less than some threshold
  - Hold out data error starts to go up
- Converge to local minimum
- Learning rate
  - Convergence sensitive to learning rate
  - Learning can be rather slow
Overfitting Issues

- NNs have the same overfitting problem as any of the other techniques
- Use small networks (reduce model complexity)
- Early stopping:
  - Train on training data set
  - At each step, evaluate performance on an independent validation test set
  - Stop when the error on the validation data is minimum

Overfitting Issues

- Regularization
  - Change the objective function
  \[
  C = \frac{1}{2N} \sum_i (t_i - o_i)^2 + \gamma \frac{1}{2N} \sum_{j=1}^D w_j^2
  \]
  \[
  \text{original cost } C_0
  \]
  \[
  \frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\gamma}{N} w
  \]
Expressive Capabilities of ANNs

- Every Boolean function can be expressed with an ANN with one hidden layer. But might require exponential (in the number of inputs) number of hidden units.
- Every bounded continuous function can be approximated with arbitrarily small error by an ANN with one hidden layer (Cybenko, 1989; Hornik et al., 1989).
- Any function can be approximated to arbitrary accuracy by a network with 2 hidden layers (Cybenko, 1988).

XOR

- Not implementable using linear perceptron
- Can you use a neural network with one hidden layer and two hidden nodes to implement XOR?
When Neural Networks Are Appropriate

- Input is high-dimensional discrete or real-valued
- Possibly noisy data
- Long training times are acceptable, but rapid classification is required
- Form of target function unknown
- Human readability of learned function not important

Deep Neural Nets

- A lot of success recently in various tasks (speech recognition, computer vision)
  - Pretraining weights, fine tuning
  - Dropout to avoid overfitting
  - etc.
Summary

Key Concepts:
- Gradient descent for training
- Backpropagation for general networks

Good:
- "simple" framework
- Direct procedure for training (gradient descent…)
- Convergence guarantees in the linear case

Not so good:
- Some parameters (learning rate, etc.)
- Need to design the architecture of the network, requires a substantial amount of engineering in designing the network
- Training can be very slow and can get stuck in local minima