CSI 436/536
Introduction to Machine Learning

Basic Neural Networks

Professor Siwei Lyu
Computer Science
University at Albany, State University of New York
biological neural networks

Brains consist of 10 to 14 types, 10 to 14 synapses, and cycle times of 1ms–10ms. Signals are noisy "spike trains" of electrical potential.

Axon
Cell body or Soma
Dendrite
Synapse
Axon from another cell
Axonal arborization
Nucleus

Chapter 20, Section 5

Visual field
Left
Right

Left side olfaction
Verbal memory

Right side olfaction
Memory for shapes

Motor speech
Stereognosis, right side
Right hand skills (e.g. writing)

Hearing (right ear preference)
Understanding language
Mathematical ability

Stereognosis, left side
Hearing (left ear preference)
Musical ability

Recognition of forms, faces and body image

Left hemisphere
Right hemisphere

Left visual field
Right visual field
artificial neural network

Output is a "squashed" linear function of the inputs:

\[ a_i = g(\sum_j W_{j,i} a_j) \]

A gross oversimplification of real neurons, but its purpose is to develop understanding of what networks of simple units can do.
an artificial neuron

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

- logistic function \( \sigma(x) = (1+e^{-x})^{-1} \)
- hyperbolic tangent function: \( \tanh(x) = 2\sigma(x) - 1 \)
- rectified linear unit function: \( \text{relu}(x) = \max(x,0) \)
- Soft-max function:
- Leaky ReLU:
Perceptron

- single layer feedforward ANN is known as a Perceptron and only produces a linear classifier

- online training algorithm
  - first developed by McClum & Pitts in the 1950s
  - online stochastic gradient descent algorithm for single layer NN with squared loss
  - convergence is theoretically guaranteed
Training multi-layered NN

- An algorithm known as back propagation (BP), and developed by Rumdblardt et.al. & LeCun in early 1980s
  - treat NN as a parametric function from input to output
  - use training data (input-output pairs) to perform supervise training
  - minimize training error (measured by a loss function) with regards to the NN
  - dynamic programming computation of the gradient to the parameter
terminology

- Computing graph (DAG)
  - input layer
  - input weights
  - hidden layers
  - activation
  - hidden weights
- output layer

$F_{\text{feed-forward network}} = \text{a parameterized family of nonlinear functions:}$

$\begin{align*}
a_5 &= g(W_{3,5} \cdot a_3 + W_{4,5} \cdot a_4) \\
   &= g(W_{3,5} \cdot g(W_{1,3} \cdot a_1 + W_{2,3} \cdot a_2)) + W_{4,5} \cdot g(W_{1,4} \cdot a_1 + W_{2,4} \cdot a_2)
\end{align*}$

Adjusting weights changes the function: doing learning this way!
feedforward neural network

- feed-forward network = a parameterized family of nonlinear functions
- adjusting weights changes the function: this is how NN is trained

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

- the key is the chain rule in calculus
back propagation (BP)

- formulation of an n-layer MLP
  - x: input data, y: target variable,
    for i=1,…n
  - $h_i$: i-th layer output of the network (vector)
  - $w_i$: network weight of i-th layer (matrix)
- network structure: $h_i = g(w_i^T h_{i-1})$ for i=1,…n
- loss function: $L(y,x_n)$:
  - $l_2$ loss, log likelihood, cross-entropy, etc
- training objective $L(w_1…w_n) = \sum_{k=1}^{m} L(y^{(k)}, x_n^{(k)})$
Optimization by gradient

- The learning objective is
  \[ \min_{w_1 \ldots w_n} L(w_1 \ldots w_n) \]
- We perform gradient descent based algorithm
  - Initializing \( W^{(0)} \)
  - Iterate until convergence
    \[ W^{(t)} = W^{(t-1)} - \eta_t \nabla L(W^{(t-1)}) \]
  - \( \nabla L(W^{(t-1)}) \) is the gradient of loss function wrt network parameter
  - \( \eta_t \) is the step size
  - This algorithm will converge to a local minimum of the learning objective
Computing gradient

- Training NN hinges on computing gradient, and we will use the chain rule

\[ \Delta z = \frac{\partial z}{\partial y} \Delta y \]
\[ \Delta y = \frac{\partial y}{\partial x} \Delta x \]
\[ \Delta z = \frac{\partial z}{\partial y_1} \frac{\partial y_1}{\partial x} \Delta x + \frac{\partial z}{\partial y_2} \frac{\partial y_2}{\partial x} \Delta x \]
\[ \frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x} \]
computation graph

- computation graph: a directed acyclic graph
- node: variables (inputs and outputs of neurons)
- edge: dependencies of variables
- \((y_1, \ldots, y_n)\) are children of \(x\)

\[
\frac{\partial z}{\partial x} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x}
\]

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

- BP = gradient descent update, so we need to compute gradient of weights of each layer
- gradient of loss function w.r.t. \( W_i \) using **chain rule**
  \[
  \frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial h_n} \frac{\partial h_n}{\partial h_{n-1}} \cdots \frac{\partial h_{i+1}}{\partial h_i} \frac{\partial h_i}{\partial w_i}
  \]
- recursion \( \frac{\partial L}{\partial h_{i-1}} = \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial h_{i-1}} \)
- \( \frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial x_i} \frac{\partial h_i}{\partial w_i} \)
BP algorithm

- BP algorithm compute $\frac{\partial L}{\partial h_n}$
  [this shows one step in the iteration over all data and until convergence]
  for $i = n:-1:1$ (back propagation)

\[
\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial h_i} \frac{\partial h_i}{\partial w_i} \quad \text{(gradient computation)}
\]

update current value of $w_i$ with $-\eta_t \frac{\partial L}{\partial w_i}$

\[
\frac{\partial I}{\partial h_{i-1}} = \frac{\partial I}{\partial h_i} \frac{\partial h_i}{\partial h_{i-1}} \quad \text{(error propagation)}
\]
**gradient check**

- NN code is difficult to debug
- gradient check is a simple trick to make sure no bug in the implementation
  - implement gradient
  - implement a finite difference computation by looping through the parameters of your network, adding and subtracting a small epsilon ($\sim 10^{-4}$) and estimate derivatives
    \[ g_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2 \times \text{EPSILON}}. \]
    \[ \theta^{(i+)} = \theta + \text{EPSILON} \times \epsilon^i \]
  - compare the two and make sure they are almost the same

**Gradient Checks are Awesome!**
- Allow you to know that there are no bugs in your neural network implementation!
Deriving gradient check

- Taylor expansion
  \[ f(x+\epsilon) = f(x) + \epsilon \nabla f(x) + 0.5 \epsilon^T \nabla^2 f(x) \epsilon + O(\epsilon^3) \]
  \[ f(x-\epsilon) = f(x) - \epsilon \nabla f(x) + 0.5 \epsilon^T \nabla^2 f(x) \epsilon + O(\epsilon^3) \]

- So if we use \( (f(x+\epsilon) - f(x))/\epsilon \) we have second order error, while if we use \( (f(x+\epsilon) - f(x-\epsilon))/2\epsilon \) we only have third order error
drawbacks of BP-trained MLP

- Vanishing gradient
  \[ \frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial h_n} \frac{\partial h_n}{\partial h_{n-1}} \ldots \frac{\partial h_{i+1}}{\partial h_i} \frac{\partial h_i}{\partial w_i} \]

- Note that the gradient will vanish after several layers of back propagation
  - Squashing nonlinearity like sigmoid or tanh reduce the range of the values
  - Multiplying smaller values eventually reduce the update to zero (below numerical precision)

- No NN can be effectively trained up to 3 layers — so not very deep model can be used
- This is one reason NN lost favor in ML in late 1990s