Lecture 5
Machine Learning

Neural Networks

Many presentation Ideas are due to Andrew NG
• Motivation
  • Non linear discriminant functions
• Introduction to Neural Networks
  • Inspiration from Biology
  • History
• Perceptron
• Multilayer Perceptron
• Practical Tips for Implementation
Need for Non-Linear Discriminant

- Previous lecture studied linear discriminant
- Works for linearly (or almost) separable cases
- Many problems are far from linearly separable
  - underfitting with linear model

\[ g(x) = w_0 + w_1 x_1 + w_2 x_2 \]
Need for Non-Linear Discriminant

- Can use other discriminant functions, like quadratics
  \[ g(x) = w_0 + w_1 x_1 + w_2 x_2 + w_{12} x_1 x_2 + w_{11} x_1^2 + w_{22} x_2^2 \]

- Methodology is almost the same as in the linear case:
  - \[ f(x) = \text{sign}(w_0 + w_1 x_1 + w_2 x_2 + w_{12} x_1 x_2 + w_{11} x_1^2 + w_{22} x_2^2) \]
  - \[ z = \begin{bmatrix} 1 & x_1 & x_2 & x_1 x_2 & x_1^2 & x_2^2 \end{bmatrix} \]
  - \[ a = \begin{bmatrix} w_0 & w_1 & w_2 & w_{12} & w_{11} & w_{22} \end{bmatrix} \]
  - “normalization”: multiply negative class samples by -1
  - gradient descent to minimize Perceptron objective function
  \[ J_p(a) = \sum_{z \in Z(a)} (-a^t z) \]
Need for Non-Linear Discriminant

- May need highly non-linear decision boundaries
- This would require too many high order polynomial terms to fit

\[ g(x) = w_0 + w_1 x_1 + w_2 x_2 + \]
\[ + w_{12} x_1 x_2 + w_{11} x_1^2 + w_{22} x_2^2 + \]
\[ + w_{111} x_1^3 + w_{112} x_1^2 x_2 + w_{122} x_1 x_2^2 + w_{222} x_2^3 + \]
\[ + \text{even more terms of degree 4} \]
\[ + \text{super many terms of degree } k \]

- For \( n \) features, there \( O(n^k) \) polynomial terms of degree \( k \)
- Many real world problems are modeled with hundreds and even thousands features
- \( 100^{10} \) is too large of function to deal with
Neural Networks

- Neural Networks correspond to some discriminant function $g_{\text{NN}}(x)$
- Can carve out arbitrarily complex decision boundaries without requiring so many terms as polynomial functions
- Neural Nets were inspired by research in how human brain works
- But also proved to be quite successful in practice
- Are used nowadays successfully for a wide variety of applications
  - took some time to get them to work
  - now used by US post for postal code recognition
Neural Nets: Character Recognition


Yann LeCun et. al.
Brain vs. Computer

- usually one very fast processor
- high reliability
- designed to solve logic and arithmetic problems
- absolute precision
- can solve a gazillion arithmetic and logic problems in an hour

- huge number of parallel but relatively slow and unreliable processors
- not perfectly precise, not perfectly reliable
- evolved (in a large part) for pattern recognition
- learns to solve various PR problems

seek inspiration for classification from human brain
One Learning Algorithm Hypothesis

- Brain does many different things
- Seems like it runs many different “programs”
- Seems we have to write tons of different programs to mimic brain

- Hypothesis: there is a single underlying learning algorithm shared by different parts of the brain

- Evidence from neuro-rewiring experiments
  - Cut the wire from ear to auditory cortex
  - Route signal from eyes to the auditory cortex
  - Auditory cortex learns to see
    - animals will eventually learn to perform a variety of object recognition tasks

- There are other similar rewiring experiments

[Roe et al, 1992]
Seeing with Tongue

• Scientists use the amazing ability of the brain to learn to retrain brain tissue

• Seeing with tongue
  • BrainPort Technology
  • Camera connected to a tongue array sensor
  • Pictures are “painted” on the tongue
    • Bright pixels correspond to high voltage
    • Gray pixels correspond to medium voltage
    • Black pixels correspond to no voltage
  • Learning takes from 2-10 hours
  • Some users describe experience resembling a low resolution version of vision they once had
    • able to recognize high contrast object, their location, movement
One Learning Algorithm Hypothesis

- Experimental evidence that we can plug any sensor to any part of the brain, and brain can learn how to deal with it
- Since the same physical piece of brain tissue can process sight, sound, etc.
- Maybe there is one learning algorithm can process sight, sound, etc.
- Maybe we need to figure out and implement an algorithm that approximates what the brain does
- Neural Networks were developed as a simulation of networks of neurons in human brain
Neuron: Basic Brain Processor

- Neurons (or nerve cells) are special cells that process and transmit information by electrical signaling
  - in brain and also spinal cord
- Human brain has around $10^{11}$ neurons
- A neuron connects to other neurons to form a network
- Each neuron cell communicates to anywhere from 1000 to 10,000 other neurons
Neuron: Main Components

• **cell body**
  • computational unit

• **dendrites**
  • “input wires”, receive inputs from other neurons
  • a neuron may have thousands of dendrites, usually short

• **axon**
  • “output wire”, sends signal to other neurons
  • single long structure (up to 1 meter)
  • splits in possibly thousands branches at the end, “axon terminals”
Neurons in Action (Simplified Picture)

- Cell body collects and processes signals from other neurons through dendrites.
- If there the strength of incoming signals is large enough, the cell body sends an electricity pulse (a spike) to its axon.
- Its axon, in turn, connects to dendrites of other neurons, transmitting spikes to other neurons.
- This is the process by which all human thought, sensing, action, etc. happens.
Artificial Neural Network (ANN) History: Birth

- 1943, famous paper by W. McCulloch (neurophysiologist) and W. Pitts (mathematician)
  - Using only math and algorithms, constructed a model of how neural network may work
  - Showed it is possible to construct any computable function with their network
  - Was it possible to make a model of thoughts of a human being?
  - Can be considered to be the birth of AI

- 1949, D. Hebb, introduced the first (purely psychologial) theory of learning
  - Brain learns at tasks through life, thereby it goes through tremendous changes
  - If two neurons fire together, they strengthen each other’s responses and are likely to fire together in the future
ANN History: First Successes

• 1958, F. Rosenblatt,
  • perceptron, oldest neural network still in use today
    • that’s what we studied in lecture on linear classifiers
  • Algorithm to train the perceptron network
  • Built in hardware
  • Proved convergence in linearly separable case

• 1959, B. Widrow and M. Hoff
  • Madaline
  • First ANN applied to real problem
    • eliminates echoes in phone lines
Early success lead to a lot of claims which were not fulfilled.

1969, M. Minsky and S. Pappert
- Book "Perceptrons"
- Proved that perceptrons can learn only linearly separable classes
- In particular cannot learn very simple XOR function
- Conjectured that multilayer neural networks also limited by linearly separable functions

No funding and almost no research (at least in North America) in 1970’s as the result of 2 things above
ANN History: Revival

- Revival of ANN in 1980’s
- 1982, J. Hopfield
  - New kind of networks (Hopfield’s networks)
  - Not just model of how human brain might work, but also how to create useful devices
    - Implements associative memory
- 1982 joint US-Japanese conference on ANN
  - US worries that it will stay behind
- Many examples of multilayer NN appear
- 1986, re-discovery of backpropagation algorithm by Werbos, Rumelhart, Hinton and Ronald Williams
  - Allows a network to learn not linearly separable classes
Artificial Neural Nets (ANN): Perceptron

- Linear classifier $f(x) = \text{sign}(w^T x + w_0)$ is a single neuron “net”
- Input layer units output features, except bias outputs ‘1’
- Output layer unit applies $\text{sign}()$ or some other function $h()$
- $h()$ is also called an activation function
Multilayer Neural Network (MNN)

- First hidden unit outputs: \( h(...)=h\left(w_0+w_1x_1+w_2x_2+w_3x_3\right) \)
- Second hidden unit outputs: \( h(...)=h\left(w_0+w_1x_1+w_2x_2+w_3x_3\right) \)
- Network corresponds to classifier \( f(x) = h\left( w\cdot h(...) + w\cdot h(...) \right) \)
- More complex than Perceptron, more complex boundaries
MNN Small Example

layer 1: input  layer 2: hidden  layer 3: output

1

x_1

3

6

7

x_2

3

5

4

2

7

• Let activation function h() = sign()
• MNN Corresponds to classifier

f(x) = \text{sign}(4 \cdot h(...) + 2 \cdot h(...) + 7) = \text{sign}(4 \cdot \text{sign}(3x_1 + 5x_2) + 2 \cdot \text{sign}(6 + 3x_2) + 7)

• MNN terminology: computing f(x) is called \textit{feed forward operation}
  • graphically, function is computed from left to right
  • Edge weights are learned through training
MNN: Multiple Classes

- 3 classes, 2 features, 1 hidden layer
  - 3 input units, one for each feature
  - 3 output units, one for each class
  - 2 hidden units
  - 1 bias unit, usually drawn in layer 1
**MNN: General Structure**

- **Input layer** (layer 1)
- **Hidden layer** (layer 2)
- **Output layer** (layer 3)

### **Classification**:

- If \( f_1(x) \) is largest, decide class 1
- If \( f_2(x) \) is largest, decide class 2
- If \( f_3(x) \) is largest, decide class 3

- \( f(x) = [f_1(x), f_2(x), f_3(x)] \) is multi-dimensional

\[
h(...)=f_1(x) \quad h(...)=f_2(x) \quad h(...)=f_3(x)\]
MNN: General Structure

- **Input layer:** $d$ features, $d$ input units
- **Output layer:** $m$ classes, $m$ output units
- **Hidden layer:** how many units?
MNN: General Structure

- Can have more than 1 hidden layer
  - $i$th layer connects to $(i+1)$th layer
    - except bias unit can connect to any layer
    - can have different number of units in each hidden layer

- First output unit outputs:
  $$h(\ldots) = h( w \cdot h(\ldots) + w ) = h( w \cdot h( w \cdot h(\ldots) + w \cdot h(\ldots) ) + w \cdot h(\ldots) ) + w$$
MNN: Activation Function

- \( h() = \text{sign}() \) is discontinuous, not good for gradient descent

- Instead can use continuous sigmoid function

- Or another differentiable function

- Can even use different activation functions at different layers/units

- From now, assume \( h() \) is a differentiable function
MNN: Overview

• A neural network corresponds to a classifier \( f(x,w) \) that can be rather complex
  • complexity depends on the number of hidden layers/units
  • \( f(x,w) \) is a composition of many functions
    • easier to visualize as a network
    • notation gets ugly

• To train neural network, just as before
  • formulate an objective function \( J(w) \)
  • optimize it with gradient descent
  • That’s all!
  • Except we need quite a few slides to write down details due to complexity of \( f(x,w) \)
Expressive Power of MNN

• Every continuous function from input to output can be implemented with enough hidden units, 1 hidden layer, and proper \textit{nonlinear} activation functions
  • easy to show that with linear activation function, multilayer neural network is equivalent to perceptron

• This is more of theoretical than practical interest
  • Proof is not constructive (does not tell how construct MNN)
  • Even if constructive, would be of no use, we do not know the desired function, our goal is to learn it through the samples
  • But this result gives confidence that we are on the right track
    • MNN is general (expressive) enough to construct any required decision boundaries, unlike the Perceptron
• Perceptron (single layer neural net)

• Arbitrarily complex decision regions
• Even not contiguous
Nonlinear Decision Boundary: Example

- Start with two Perceptrons, $h() = \text{sign}()$

\[-x_1 + x_2 - 1 > 0 \Rightarrow \text{class 1}\]

\[-x_1 + x_2 - 3 > 0 \Rightarrow \text{class 1}\]
Nonlinear Decision Boundary: Example

- Now combine them into a 3 layer NN
MNN: Modes of Operation

• For Neural Networks, due to historical reasons, training and testing stages have special names
  • **Backpropagation (or training)**
    Minimize objective function with gradient descent
  • **Feedforward (or testing)**
**MNN: Notation for Edge Weights**

- $w_{pj}^k$ is edge weight from unit $p$ in layer $k-1$ to unit $j$ in layer $k$
- $w_{0j}^k$ is edge weight from bias unit to unit $j$ in layer $k$
- $w_j^k$ is all weights to unit $j$ in layer $k$, i.e. $w_{0j}^k, w_{1j}^k, \ldots, w_{N(k-1)j}^k$
  - $N(k)$ is the number of units in layer $k$, excluding the bias unit
• Denote the output of unit \( j \) in layer \( k \) as \( z_{j}^{k} \)

• For the input layer \((k=1)\), \( z_{0}^{1} = 1 \) and \( z_{j}^{1} = x_{j}, j \neq 0 \)

• For all other layers, \((k > 1)\), \( z_{j}^{k} = h(\ldots) \)

• Convenient to set \( z_{0}^{k} = 1 \) for all \( k \)

• Set \( z^{k} = [z_{0}^{k}, z_{1}^{k}, \ldots, z_{N(k)}^{k}] \)
• Net activation at unit $j$ in layer $k > 1$ is the sum of inputs

$$a_j^k = \sum_{p=1}^{N_{k-1}} z_{p}^{k-1} w_{pj}^k + w_{0j}^k = \sum_{p=0}^{N_{k-1}} z_{p}^{k-1} w_{pj}^k = z_{1}^{k-1} \cdot w_{j}^k$$

$$a_1^2 = z_{0}^1 w_{01}^2 + z_{1}^1 w_{11}^2 + z_{2}^1 w_{21}^2$$

• For $k > 1$, $z_j^k = h(a_j^k)$
MNN: Class Representation

- **m** class problem, let Neural Net have **t** layers
- Let \( x^i \) be a example of class \( c \)
- It is convenient to denote its label as \( y^i = \begin{bmatrix} 1 \\ \vdots \\ 0 \end{bmatrix} \)
- Recall that \( z^t_c \) is the output of unit \( c \) in layer \( t \) (output layer)

\[
\begin{bmatrix}
    z^t_1 \\
    \vdots \\
    z^t_c \\
    \vdots \\
    z^t_m
\end{bmatrix}
\]

- \( f(x) = z^t = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \)  
  If \( x^i \) is of class \( c \), want \( z^t = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \)
Training MNN: Objective Function

• Want to minimize difference between $y_i$ and $f(x_i)$
• Use squared difference
• Let $w$ be all edge weights in MNN collected in one vector
• Error on one example $x_i$: $J_i(w) = \frac{1}{2} \sum_{c=1}^{m} (f_c(x_i) - y_c^i)^2$
• Error on all examples: $J(w) = \frac{1}{2} \sum_{i=1}^{n} \sum_{c=1}^{m} (f_c(x_i) - y_c^i)^2$

Gradient descent:

initialize $w$ to random
choose $\varepsilon, \alpha$
while $\alpha ||\nabla J(w)|| > \varepsilon$

$w = w - \alpha \nabla J(w)$
Training MNN: Single Sample

- For simplicity, first consider error for one example $x^i$
  \[
  J_i(w) = \frac{1}{2} \|y^i - f(x^i)\|^2 = \frac{1}{2} \sum_{c=1}^{m} (f_c(x^i) - y^i_c)^2
  \]
  - $f_c(x^i)$ depends on $w$
  - $y^i$ is independent of $w$

- Compute partial derivatives w.r.t. $w_{pj}^k$ for all $k, p, j$
- Suppose have $t$ layers
  \[
  f_c(x^i) = z^t_c = h(a^t_c) = h(z^{t-1} \cdot w^t_c)
  \]
Training MNN: Single Sample

- For derivation, we use:
  \[ J_i(w) = \frac{1}{2} \sum_{c=1}^{m} (f_c(x^i) - y_c^i)^2 \]
  \[ f_c(x^i) = h(a_c^t) = h(z^{t-1} \cdot w_c^t) \]

- For weights \( w_{pj}^t \) to the output layer \( t \):
  \[ \frac{\partial}{\partial w_{pj}^t} J(w) = (f_j(x^i) - y_j^i) \frac{\partial}{\partial w_{pj}^t} (f_j(x^i) - y_j^i) \]

  \[ \frac{\partial}{\partial w_{pj}^t} (f_j(x^i) - y_j^i) = h'(a_j^t) z_p^{t-1} \]

- Therefore,
  \[ \frac{\partial}{\partial w_{pj}^t} J_i(w) = (f_j(x^i) - y_j^i) h'(a_j^t) z_p^{t-1} \]

  - both \( h'(a_j^t) \) and \( z_p^{t-1} \) depend on \( x^i \). For simpler notation, we don’t make this dependence explicit.
Training MNN: Single Sample

- For a layer $k$, compute partial derivatives w.r.t. $w^k_{pj}$
- Gets complex, since have lots of function compositions
- Will give the rest of derivatives
- First define $e^k_j$, the error attributed to unit $j$ in layer $k$:
  - For layer $t$ (output): $e^t_j = (f_j(x^i) - y^i_j)$
  - For layers $k < t$: $e^k_j = \sum_{c=1}^{N(k+1)} e^{k+1}_c h'(a^{k+1}_c) w^{k+1}_{jc}$
  - Thus for $2 \leq k \leq t$: $\frac{\partial}{\partial w^k_{pj}} J_i(w) = e^k_j h'(a^k_j) z^{k-1}_p$
MNN Training: Multiple Samples

- Error on one example $x^i$:  
  $$ J_i(w) = \frac{1}{2} \sum_{c=1}^{m} \left( f_c(x^i) - y_c^i \right)^2 $$

  $$ \frac{\partial}{\partial w^k_{pj}} J_i(w) = e_j^k h'(a_j^k) z_{p}^{k-1} $$

- Error on all examples:  
  $$ J(w) = \frac{1}{2} \sum_{i=1}^{n} \sum_{c=1}^{m} \left( f_c(x^i) - y_c^i \right)^2 $$

  $$ \frac{\partial}{\partial w^k_{pj}} J(w) = \sum_{i=1}^{n} e_j^k h'(a_j^k) z_{p}^{k-1} $$
Training Protocols

• **Batch Protocol**
  - true gradient descent
  - weights are updated only after all examples are processed
  - might be slow to converge

• **Single Sample Protocol**
  - examples are chosen randomly from the training set
  - weights are updated after every example
  - converges faster than batch, but maybe to an inferior solution

• **Mini-Batch protocol**
  - In between batch and single sample protocols
  - choose sets (batches) of examples
  - Update weights after each batch

• **Online Protocol**
  - each example is presented only once, weights update after each example presentation
  - used if number of examples is large and does not fit in memory
  - should be avoided when possible
**MNN Training: Single Sample**

initialize $w$ to small random numbers
choose $\varepsilon$, $\alpha$

```latex
\textbf{while} $\alpha||\nabla J(w)|| > \varepsilon$

\textbf{for} $i = 1$ to $n$

\begin{align*}
    r &= \text{random index from } \{1, 2, \ldots, n\} \\
    \text{delta}_{pjk} &= 0 \quad \forall \ p, j, k \\
    e_j^t &= (f_j(x^r) - y_j^r) \quad \forall j \\
    \text{for } k = t \text{ to } 2 \\
    \text{delta}_{pjk} &= \text{delta}_{pjk} - e_j^k h'(a_j^k) z_p^{k-1} \\
    e_j^{k-1} &= \sum_{c=1}^{N(k)} e_c^k h'(a_c^k) w_{jc}^k \quad \forall j \\
    w_{pj}^k &= w_{pj}^k + \text{delta}_{pjk} \quad \forall \ p, j, k
\end{align*}
```
MNN Training: Batch

initialize \( w \) to small random numbers
choose \( \varepsilon, \alpha \)

while \( \alpha||\nabla J(w)|| > \varepsilon \)
    for \( i = 1 \) to \( n \)
        \( \delta_{pjk} = 0 \quad \forall \ p,j,k \)
        \( e^t_j = (f_j(x^i) - y^i_j) \quad \forall j \)
    for \( k = t \) to \( 2 \)
        \( \delta_{pjk} = \delta_{pjk} - e^k_j h'(a^k_j) z^{k-1}_p \)
        \( e^{k-1}_j = \sum_{c=1}^{N(k)} e^k_c h'(a^k_c) w^k_{jc} \quad \forall j \)
    \( w^k_{pj} = w^k_{pj} + \delta_{pjk} \quad \forall p,j,k \)
BackPropagation of Errors

• In MNN terminology, training is called backpropagation
• errors computed (propagated) backwards from the output to the input layer

\[
\text{while } \alpha||\nabla J(w)|| > \varepsilon \\
\text{for } i = 1 \text{ to } n \\
\delta_{pjk} = 0 \quad \forall \ p,j,k \\
e^t_j = \left(y^r_j - f_j(x^r)\right) \quad \forall j \quad \text{first last layer errors computed} \\
\text{for } k = t \text{ to } 2 \\
\delta_{pjk} = \delta_{pjk} - e^k_j h'(a^k_j) z^{k-1}_p \\
e^{k-1}_j = \sum_{c=1}^{N(k)} e^k_c h'(a^k_c) w^k_{jc} \quad \forall j \\
w^k_{pj} = w^k_{pj} + \delta_{pjk} \quad \forall p,j,k
\]
MNN Training

• Important: weights should be initialized to random nonzero numbers

\[
\frac{\partial}{\partial w_{pj}^k} J_i(w) = -e_j^k h'(a_j^k) z_p^{k-1}
\]

\[
e_j^k = \sum_{c=1}^{N(k+1)} e_c^{k+1} h'(a_c^{k+1}) w_{jc}^{k+1}
\]

• if \( w_{jc}^k = 0 \), errors \( e_j^k \) are zero for layers \( k < t \)

• weights in layers \( k < t \) will not be updated
MNN Training: How long to Train?

- **Large training error:**
  - Random decision regions in the beginning - underfit

- **Small training error:**
  - Decision regions improve with time

- **Zero training error:**
  - Decision regions fit training data perfectly - overfit

Can learn when to stop training through validation
MNN as Non-Linear Feature Mapping

- MNN can be interpreted as first mapping input features to new features
- Then applying Perceptron (linear classifier) to the new features
MNN as Non-Linear Feature Mapping

This part implements Perceptron (linear classifier)
MNN as Non-Linear Feature Mapping

This part implements mapping to new features $y$. 
MNN as Nonlinear Feature Mapping

- Consider 3 layer NN example we saw previously:

non linearly separable in the original feature space

linearly separable in the new feature space
Neural Network Demo

- [http://www.youtube.com/watch?v=nIRGz1GEzgI](http://www.youtube.com/watch?v=nIRGz1GEzgI)

RoboSight - Neural Network Camera
To avoid overfitting, it is recommended to keep weights small.

Implement weight decay after each weight update:

\[ w_{\text{new}} = w_{\text{new}}(1-\beta), \ 0 < \beta < 1 \]

Additional benefit is that “unused” weights grow small and may be eliminated altogether:

- a weight is “unused” if it is left almost unchanged by the backpropagation algorithm.
Practical Tips for BP: Momentum

• Gradient descent finds only a local minima
• Momentum: popular method to avoid local minima and speed up descent in flat (plateau) regions
• Add temporal average direction in which weights have been moving recently
• Previous direction: $\Delta w^t = w^t - w^{t-1}$
• Weight update rule with momentum:

$$w^{t+1} = w^t + (1-\beta) \left[ \alpha \frac{\partial J}{\partial w} \right] + \beta \Delta w^{t-1}$$

- steepest descent direction
- previous direction
Practical Tips for BP: Activation Function

- Gradient descent works with any differentiable $h$, however some choices are better
- Desirable properties for $h$:
  - Nonlinearity to express nonlinear decision boundaries
  - Saturation, that is $h$ has minimum and maximum values
    - Keeps weights bounded, thus training time is reduced
  - Monotonicity so that activation function itself does not introduce additional local minima
  - Linearity for a small values, so that network can produce linear model, if data supports it
  - Antisymmetry, that is $h(-1) = -h(1)$, leads to faster learning
Practical Tips for BP: Activation Function

- Sigmoid function $h$ satisfies all of the properties

$$h(q) = a \frac{e^{b \cdot q} - e^{-b \cdot q}}{e^{b \cdot q} + e^{-b \cdot q}}$$

- Good parameter choices are $a = 1.716$, $b = 2/3$
- Asymptotic values $\pm 1.716$
  - bigger than our labels, which are 1
  - If asymptotic values were smaller than 1, training error will not be small
- Linear range is roughly for $-1 < q < 1$
Practical Tips for BP: Normalization

• Features should be normalized for faster convergence
• Suppose we measure fish length in meters and weight in grams
  • Typical sample [length = 0.5, weight = 3000]
  • Feature length will be almost ignored
  • If length is in fact important, learning will be very slow
• Any normalization we looked at before (lecture on kNN) will do
  • Test samples should be normalized exactly as the training samples
Practical Tips: Initializing Weights

• Depends on the activation function
• Rule of thumb for commonly used sigmoid function
  • recall that $N(k)$ is the number of units in layer $k$
  • for layer $k$, choose weights from the range at random
    \[-\frac{1}{\sqrt{N(k)}} < w_{pj}^k < \frac{1}{\sqrt{N(k)}}\]
As any gradient descent algorithm, backpropagation depends on the learning rate $\alpha$

- Rule of thumb $\alpha = 0.1$
- However can adjust $\alpha$ at the training time
- The objective function $J(w)$ should decrease during gradient descent
  - If $J(w)$ oscillates, $\alpha$ is too large, decrease it
  - If $J(w)$ goes down but very slowly, $\alpha$ is too small, increase it
Practical Tips: Number of Hidden Layers

• Network with 1 hidden layer has the same expressive power as with several hidden layers
• Having more than 1 hidden layer may result in faster learning and less hidden units
• However, networks with more than 1 hidden layer are more prone to stuck in a local minima
Practical Tips for BP: Number of Hidden Units

- Number of hidden units determines the expressive power of the network
  - Too small may not be sufficient to learn complex decision boundaries
  - Too large may overfit the training data
- Sometimes recommended that
  - number of hidden units is larger than the number of input units
  - number of hidden units is the same in all hidden layers
- Can choose number of hidden units through validation
Concluding Remarks

• Advantages
  • MNN can learn complex mappings from inputs to outputs, based only on the training samples
  • Easy to incorporate a lot of heuristics

• Disadvantages
  • It is a “black box”, i.e. it is difficult to analyze and predict its behavior
  • May take a long time to train
  • May get trapped in a bad local minima
  • A lot of tricks for best implementation