Lecture 7
Machine Learning
Neural Networks

Many presentation ideas are due to Andrew NG
Outline

- Motivation
  - Non linear discriminant functions
- Introduction to Neural Networks
  - Inspiration from Biology
  - History
- Perceptron: 1 layer Neural Network
- Multilayer Neural Networks
  - also called Artificial Neural Network (ANN), perceptron (MLP), Feedforward Neural Network
- Training Neural Networks
  - backpropagation algorithm
  - practical tips for training
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
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
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
1958, F. Rosenblatt, Cornell University
- Perceptron, oldest neural network
  - studied in lecture on linear classifiers
- Algorithm to train the Perceptron
- Built in hardware to recognize digits images
- Proved convergence in linearly separable case
- Early success led to a lot of claims which were not fulfilled
- New York Times reports that perceptron is "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence."
ANN History: Stagnation

- 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
• Revival of ANN in early 1980
• 1986, (re)discovery of backpropagation algorithm by Werbos, Rumelhart, Hinton and Ronald Williams
  • Allows training a MLP
• Many examples of multilayer Neural Networks appear
• 1998, Convolutional network (convnet) by Y. Lecun for digit recognition, very successful
• 1990’s: research in NN move slowly again
  • Networks with multiple layers are hard to train well (except convnet for digit recognition)
  • SVM becomes popular, works better
Deep networks are inspired by brain architecture.

Until now, no success at training them, except convnet.

2006-now: deep networks are trained successfully

- massive training data becomes available
- better hardware: fast training on GPU
- better training algorithms for network training when there are many hidden layers
  - unsupervised learning of features, helps when training data is limited

Break through papers


Industry: Facebook, Google, Microsoft, etc.
Linear classifier $f(x) = \text{sign}(w^t x + w_0)$ is a single neuron “net”

- Input layer units emits features, except bias emits “1”
- Output layer unit applies $h(t) = \text{sign}(t)$
- $h(t)$ is also called an activation function
Multilayer Neural Network

- First hidden unit outputs: $h(w_0+w_1x_1+w_2x_2)$
- Second hidden unit outputs: $h(w_0+w_1x_1+w_2x_2)$
- Network implements classifier: $f(x) = h(wh(\cdot)+wh(\cdot))$
- More complex boundaries than Perceptron
• Implements classifier

\[ f(x) = \text{sign}(4h(\cdot)+2h(\cdot) + 7) \]

\[ = \text{sign}(4 \cdot \text{sign}(3x_1+5x_2)+2 \cdot \text{sign}(6+3x_2) + 7) \]

• Computing \( f(x) \) is called \textit{feed forward operation}
  • graphically, function is computed from left to right

• Edge weights are learned through training
• 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, can draw in layer 1, or each layer has one
Multilayer NN: General Structure

- **Input layer**
- **Hidden layer**
- **Output layer**

- $f(x)$ is multi-dimensional

**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
Multilayer NN: General Structure

- **Input layer**: \(d\) features, \(d\) input units
- **Output layer**: \(m\) classes, \(m\) output units
- **Hidden layer**: how many units?
  - more units correspond to more complex classifiers
Multilayer NN: General Structure

- Can have many hidden layers
- **Feed forward** structure
  - $i$th layer connects to $(i+1)$th layer
  - except bias unit can connect to any layer
  - or, alternatively each layer can have its own bias unit
• NN corresponds to rather complex classifier $f(x, w)$
  • complexity depends on the number of hidden layers/units
  • $f(x, w)$ is a composition of many functions
    • easier to visualize as a network rather than write out the functions

• To train NN, just as before
  • formulate per-sample loss function $L(w)$
  • optimize it with gradient descent
    • lots of heuristics to get gradient descent work well enough
• Every continuous function from input to output can be implemented with enough hidden units, 1 hidden layer, and proper *nonlinear* activation functions
  • easy to show that with linear activation function, multilayer neural network is equivalent to perceptron
• More of theoretical than practical interest
  • do not know the desired function in the first place, our goal is to learn it through the samples
  • but this result gives confidence that we are on the right track
    • multilayer NN is general (expressive) enough to construct any required decision boundaries, unlike the Perceptron
Multilayer NN: Decision Boundaries

• Perceptron (single layer neural net)

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

\(-x_1 + x_2 - 1 > 0 \Rightarrow \text{class 1}\)

\(x_1 - x_2 - 3 > 0 \Rightarrow \text{class 1}\)
Multilayer NN: Nonlinear Boundary Example

- Combine two Perceptrons into a 3 layer NN
Multilayer NN as Non-Linear Feature Mapping

- **Interpretation**
  - 1 hidden layer maps input features to new features
  - next layer then applies linear classifier to the new features
Multilayer NN as Non-Linear Feature Mapping

this part implements Perceptron (linear classifier)
Multilayer NN as Non-Linear Feature Mapping

this part implements mapping to new features $y$
Multilayer NN as Non-Linear 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
Multi Layer NN: Activation Function

- \( h() = \text{sign}() \) does not work for gradient descent

- Can use \text{tanh} or \text{sigmoid} function

- Rectified Linear (ReLu) popular recently
  - gradients do not saturate for positive half-interval
  - but have to be careful with learning rate, otherwise many units can become “dead”, i.e. always output 0
Due to historical reasons, training and testing stages have special names

- **Backpropagation (or training)**
  Minimize objective function with gradient descent

- **Feedforward (or testing)**
Multilayer NN: Matrix Notation

- Recall matrix notation for linear classifier

\[
\begin{pmatrix}
2 & 4 & -7 \\
9 & -3 & 2 \\
4 & 5 & 2 \\
2 & -7 & 1
\end{pmatrix}
\begin{pmatrix}
1 \\
7 \\
4
\end{pmatrix}
= 
\begin{pmatrix}
2 \\
-4 \\
47 \\
-43
\end{pmatrix}
\]

- Full picture, ignoring bias weights

\[
\begin{align*}
&\mathbf{w}_1^T \mathbf{x} \\
&\mathbf{w}_2^T \mathbf{x} \\
&\mathbf{w}_3^T \mathbf{x} \\
&\mathbf{w}_4^T \mathbf{x}
\end{align*}
\]

\[W\mathbf{x}\]

- This is subpart of neural network
- Need to add activation function
Multilayer NN: Matrix Notation

- Full picture

\[ Wx \]

- This is subpart of neural network

\[ h(\mathbf{w}_1^T \mathbf{x}) \]

- Need activation function \( h \) in Neural Network

\[
\begin{pmatrix}
4 & -8 & -3 & 2
\end{pmatrix}
\]

\[
\begin{pmatrix}
2 & 5 & -7 & 1
\end{pmatrix}
\]

\[
\begin{pmatrix}
1
\end{pmatrix}
\]

\[ h = \begin{pmatrix}
* \\
* \\
* \\
* 
\end{pmatrix} \]
**Multilayer NN: Matrix Notation**

- Use similar notation for NN

\[ h = h(Wx) \]

\[ h = h(Wh) \]

\[ W = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \]

\[ h = \begin{bmatrix} * \\ * \end{bmatrix} \]

\[ h = \begin{bmatrix} * \\ * \end{bmatrix} \]

\[ W = \begin{bmatrix} 7 & 4 \\ 8 & 9 \\ 1 & 3 \end{bmatrix} \]
Multilayer NN: Matrix Notation

• Instead of color, use superscripts

\[ h^1 = h(W^1 x) \]
\[ h^2 = h(W^2 h^1) \]

\[ h^1 = \begin{bmatrix} * \\ * \end{bmatrix} \]
\[ h^2 = \begin{bmatrix} * \\ * \end{bmatrix} \]

\[ W^1 = \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \]
\[ W^2 = \begin{bmatrix} 7 & 4 \\ 8 & 9 \\ 1 & 3 \end{bmatrix} \]
Multilayer NN: Matrix Notation

- Add bias weights, also as vectors

\[ h^1 = h(W^1 x + b^1) \quad h^2 = h(W^2 h^1 + b^2) \]

\[
\begin{align*}
\mathbf{W}^1 &= \begin{bmatrix} 1 & 3 & 5 \\ 2 & 4 & 6 \end{bmatrix} \\
\mathbf{W}^2 &= \begin{bmatrix} 7 & 4 \\ 8 & 9 \\ 1 & 3 \end{bmatrix} \\
\mathbf{b}^1 &= \begin{bmatrix} \ast \\ \ast \end{bmatrix} \\
\mathbf{b}^2 &= \begin{bmatrix} \ast \\ \ast \\ \ast \end{bmatrix}
\end{align*}
\]
Multilayer NN: Vector Notation for Next Layer

\[
\begin{align*}
\mathbf{x}_1 \rightarrow \mathbf{h} & \rightarrow \mathbf{h} \rightarrow \mathbf{h} \\
\mathbf{x}_2 \rightarrow \mathbf{h} & \rightarrow \mathbf{h} \rightarrow \mathbf{h}
\end{align*}
\]

\[\mathbf{x} \rightarrow \mathbf{h} (\mathbf{W}^1 \mathbf{x} + \mathbf{b}^1) \rightarrow \mathbf{h}^1 \rightarrow \mathbf{h} (\mathbf{W}^2 \mathbf{h}^1 + \mathbf{b}^2) \rightarrow \mathbf{h}^2\]

- \( \mathbf{W}^2 \) is a matrix of weights between hidden layer 1 and 2
  - \( \mathbf{W}^2(r,c) \) is weight from unit \( c \) to unit \( r \)
- \( \mathbf{b}^2 \) is a vector of bias weights for second hidden layer
  - \( b^2_r \) is bias weight of unit \( r \) in second layer
- \( \mathbf{h}^2 \) is a vector of second layer outputs
  - \( h^2_r \) is output of unit \( r \) in second layer
Multilayer NN: Vector Notation, all Layers

- $h^3$ is vector from the output layer and it is also called $f(x,W)$
- $h^3 = h(W^3h^2 + b^3)$
  
  $= h(W^3h(W^2h^1 + b^2) + b^3)$
  
  $= h(W^3h(W^2h(W^1x + b^1) + b^2) + b^3)$
Vector Notation, Example

- Assuming sign activation function, draw a NN given by

\[
\begin{align*}
    w^1 &= \begin{bmatrix} 3 & 4 \\ 1 & 2 \\ 9 & 7 \end{bmatrix}, & b^1 &= \begin{bmatrix} 5 \\ 8 \end{bmatrix}, & w^2 &= \begin{bmatrix} 3 & 4 & 1 \\ 5 & 2 & 7 \end{bmatrix}, & b^2 &= \begin{bmatrix} 9 \\ 1 \end{bmatrix}, & w^3 &= \begin{bmatrix} 5 & 3 \end{bmatrix}, & b^3 &= \begin{bmatrix} 6 \end{bmatrix}.
\end{align*}
\]
Multilayer NN: Output Representation

- Output of NN is a vector
- As before, if $x^i$ be sample of class $k$, its label is

$$y^i = \begin{bmatrix} 0 & \cdots & 1 & \cdots & 0 \end{bmatrix} \quad \text{row } k$$

$$f(x^i, W) = \begin{bmatrix} 0 & \cdots & 1 & \cdots & 0 \end{bmatrix} \quad \text{row } k$$

wish to get this output
• Wish to minimize difference between $y^i$ and $f(x^i)$

• Let $W$ be all weights (all matrices $W^t$ and bias vectors $b^t$)

• With squared difference loss

• Squared loss on one example $x^i$:

$$L(x^i, y^i; W) = \left\| f(x^i, W) - y^i \right\|^2 = \sum_{j=1}^{m} (f_j(x^i, W) - y_j^i)^2$$

• For this example, squared loss is $3^2 + 2^2 = 13$

$$f(x^i, W) = \begin{bmatrix} 3 \\ 1 \\ -2 \end{bmatrix} \quad y^i = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$
Training NN: Squared Difference Loss

• Let \( X = x^1, \ldots, x^n \)
  \( Y = y^1, \ldots, y^n \)

• Loss on all examples: \( L(X, Y; W) = \sum_{i=1}^{n} \left\| f(x^i, W) - y^i \right\|^2 \)

• Gradient descent

Initialize \( W \) to random
choose \( \varepsilon, \alpha \)
while \( \alpha \|\nabla L(X, Y; W)\| > \varepsilon \)

\( W = W - \alpha \nabla L(X, Y; W) \)
Training NN: Softmax Loss

- Squared error loss is not recommended for classification
- Softmax is a better loss function, seen before in linear classifier
- First put the output of the network through soft-max

\[
f_k(x) = \frac{\exp(o_k)}{\sum_{j=1}^{m} \exp(o_j)}
\]

\[
o = \begin{bmatrix} 0.6 \\ -1 \\ 5 \\ 8 \\ 4 \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} 0.006 \\ 0.0001 \\ 0.047 \\ 0.94 \\ 0.17 \end{bmatrix} = f(x) = \text{softmax}(o)
\]

- Interpret \( f_k(x) \) as probability of class \( k \)
Training NN: Softmax Loss

• If sample $x$ is of class $k$, the loss is

$$L(x, y; W) = -\log f_k(x)$$

• this loss function is also called $-\log$ loss, cross entropy loss
• minimizing $-\log$ is equivalent to maximizing probability

• Loss on all samples

$$L(X, Y; W) = \sum L(x, y; W)$$
Training NN: -Log Loss Function

- Need to find derivative of $L$ wrt every network weight $w_i$
  \[ \frac{\partial L}{\partial w_i} \]

- After derivative found, according to gradient descent, weight update is
  \[ \Delta w_i = -\alpha \frac{\partial L}{\partial w_i} \]
  - where $\alpha$ is the learning rate

- Update weight
  \[ w_i = w_i + \Delta w_i \]
• How many weights do we have in our network?

Weights are in matrices $W^1, W^2, ..., W^L$
• And in matrices $b^1, b^2, ..., b^L$
Computing Derivatives: Small Example

- Small network $f(x,y,z) = (x+y)z$
- Rewrite using
  - $q = x + y$
  - $f(x,y,z) = qz$
- each node does one operation

```
x

q=x+y

y

f=qz

z
```
Computing Derivatives: Small Example

- Small network \( f(x, y, z) = (x+y)z \)
- Rewrite using
  - \( q = x + y \)
  - \( f(x, y, z) = qz \)
- Example of computing \( f(-2, 5, -4) \)

\[
\begin{align*}
q &= x + y \\
f &= qz
\end{align*}
\]
Computing Derivatives: Small Example

- Small network \( f(x,y,z) = (x+y)z \)
- Rewrite using \( q = x + y \) \( \implies \) \( f(x,y,z) = qz \)
- Want \( \frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \)
- Compute \( \frac{\partial f}{\partial q} \) from the end backwards
  - for each edge, with respect to the main variable at edge origin
  - using chain rule with respect to the variable at edge end, if needed

\[
\frac{\partial f}{\partial x} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial x} = -4
\]
\[
\frac{\partial f}{\partial y} = \frac{\partial f}{\partial q} \frac{\partial q}{\partial y} = -4
\]
\[
\frac{\partial f}{\partial z} = q = 3
\]
\[
\frac{\partial f}{\partial q} = z = -4
\]
\[
\frac{\partial f}{\partial f} = 1
\]
\[
\frac{\partial f}{\partial q} = 3
\]
\[
\frac{\partial f}{\partial f} = -12
\]
Computing Derivatives: Chain of Chain Rule

- Compute $\frac{\partial d}{\partial \partial}$ from the end backwards
  - for each edge, with respect to the main variable at edge origin
  - using chain rule with respect to the variable at edge end, if needed

$\frac{\partial d}{\partial a} = \frac{\partial d}{\partial b} \frac{\partial b}{\partial a}$

$\frac{\partial d}{\partial b} = \frac{\partial d}{\partial c} \frac{\partial c}{\partial b}$

$\frac{\partial d}{\partial c}$

prev local prev local local

example: if $h(c) = c^2$, then $\frac{\partial d}{\partial c} = \frac{\partial h}{\partial c} = 2c$
Computing Derivatives Backwards

- Have loss function $L(o)$
- Need derivatives for all $\frac{\partial L}{\partial w}$, $\frac{\partial L}{\partial b}$
- Will compute derivatives from end to front, backwards
- On the way will also compute intermediate derivatives $\frac{\partial L}{\partial h}$
Computing Derivatives: Look at One Node

- Simplified view at a network node
  - inputs $x, y$ come in
  - node computes some function $h(x, y)$
Computing Derivatives: Look at One Node

- At each network node
  - inputs $x, y$ come in
  - nodes computes activation function $h(x, y)$
- Have loss function $L(\cdot)$

\[
\frac{\partial L}{\partial x} \quad \text{already computed} \quad \frac{\partial L}{\partial h}
\]
Computing Derivatives: Look at One Node

- Need $\frac{\partial L}{\partial x}, \frac{\partial L}{\partial y}$
- Easy to compute local node derivatives $\frac{\partial h}{\partial x}, \frac{\partial h}{\partial y}$

\[
\frac{\partial L}{\partial x} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial x}
\]

\[
\frac{\partial L}{\partial y} = \frac{\partial L}{\partial h} \frac{\partial h}{\partial y}
\]

$h(x, y)$

already computed

$\frac{\partial L}{\partial h}$
Computing Derivatives: Look at One Node

- More complete view at a network node
  - inputs \(x, y\) come in, get multiplied by weight \(w\) and \(v\)
  - node computes function \(h(wx, vy)\)
  - node output \(h\) gets multiplied by \(u\)
To be concrete, let $h(i,j) = i + j$
Computing Derivatives: Look at One Node

- Break into more computational nodes
  - all computation happens inside nodes, not on edges

\[
\begin{align*}
wx & \rightarrow h(wx, vy) \rightarrow uh \\
v & \rightarrow vy \\
w & \rightarrow a = wx \\
x & \rightarrow b = vy \\
y & \\
h &= a + b \\
c &= uh
\end{align*}
\]

- \( h(i,j) = i + j \)
Computing Derivatives: Look at One Node

- Some of these partial derivatives are intermediate
  - their values will not be used for gradient descent
Computing Derivatives: Look at One Node

\[
\frac{\partial L}{\partial w} = \frac{\partial L}{\partial a} \frac{\partial a}{\partial w} = \frac{\partial L}{\partial a}
\]

\[
\frac{\partial L}{\partial a} \frac{\partial a}{\partial x} = \frac{\partial L}{\partial x} \frac{\partial x}{\partial a}
\]

\[
\frac{\partial L}{\partial h} = \frac{\partial L}{\partial c} \frac{\partial c}{\partial h} \frac{\partial h}{\partial c} = 2u = 4
\]

\[
\frac{\partial L}{\partial c} = 2
\]

\[
\frac{\partial L}{\partial y} = \frac{\partial L}{\partial v} \frac{\partial v}{\partial y} = \frac{\partial L}{\partial v}
\]

Example when \( w = 1, x = 2, v = 3, y = 4, u = 2 \), \( \frac{\partial L}{\partial c} = 2 \)
• Each node is responsible for one function
• To compute $\exp(1/x)$
Computing Derivatives: Vector Notation

- Inputs and outputs are often vectors and/or matrices

- $h(a)$ is a function from $\mathbb{R}^n$ to $\mathbb{R}^m$
- Chain rule generalizes to vector and matrix functions
- Will not derive it, but will give you the end result
Computing Derivatives: Vector Notation

- Assume loss $L$ is a scalar
  - if not, can do derivation for each component independently

- Assume $W$, $X$, and $h$ are matrices
  - subsumes the case when they are vectors as well

\[
\frac{\partial L}{\partial X} = W^T \frac{\partial L}{\partial h}
\]

\[
\frac{\partial L}{\partial W} = \frac{\partial L}{\partial h} X^T
\]

- $\frac{\partial L}{\partial h}$ is a matrix of partial derivatives of the same shape as $W$
• First compute $\frac{\partial L}{\partial o}$

• Under quadratic loss

$$\frac{\partial L}{\partial o} = f(x) - y$$

• Under softmax loss

$$\frac{\partial L}{\partial o} = \text{softmax}(f(x)) - y$$
Computing Derivatives: Vector Notation

- Let vector $a^3 = W^3 h^2 + b^3$

\[
\frac{\partial L}{\partial W^3} = \text{diag}(h'(a^3)) \frac{\partial L}{\partial o} (h^2)^T
\]

\[
\frac{\partial L}{\partial h^2} = \text{diag}(h'(a^3))(W^3)^T \frac{\partial L}{\partial o}
\]

\[
\frac{\partial L}{\partial b^3} = \text{diag}(h'(a^3)) \frac{\partial L}{\partial o}
\]
Computing Derivatives: Vector Notation

- Continue computing backwards
- Let vector $a^2 = W^2h^1 + b^2$

\[
\frac{\partial L}{\partial W^2} = \text{diag}(h'(a^2)) \frac{\partial L}{\partial h^2} (h^1)^T
\]

\[
\frac{\partial L}{\partial h^1} = \text{diag}(h'(a^2))(W^2)^T \frac{\partial L}{\partial h^2}
\]

\[
\frac{\partial L}{\partial b^2} = \text{diag}(h'(a^2)) \frac{\partial L}{\partial h^2}
\]
• Continue computing backwards
• Let vector $a^1 = W^1 x^1 + b^1$

\[
\frac{\partial L}{\partial W^1} = \text{diag}(h'(a^1)) \frac{\partial L}{\partial h^1} x^T
\]

\[
\frac{\partial L}{\partial b^1} = \text{diag}(h'(a^1)) \frac{\partial L}{\partial h^1}
\]
Training Protocols

• Batch Protocol
  • full gradient descent
  • weights are updated only after all examples are processed
  • might be very slow to train

• Single Sample Protocol
  • examples are chosen randomly from the training set
  • weights are updated after every example
  • weighs get changed faster than batch, less stable
  • One iteration over all samples (in random order) is called an epoch

• Mini Batch
  • Divide data in batches, and update weights after processing each batch
  • Middle ground between single sample and batch protocols
  • Helps to prevent over-fitting in practice, think of it as “noisy” gradient
  • allows CPU/GPU memory hierarchy to be exploited so that it trains much faster than single-sample in terms of wall-clock time
  • One iteration over all mini-batches is called an epoch
Regularization

- Larger networks are more prone to overfitting
Regularization

- Can control overfitting by using network with less units
- Better if control overfitting by adding weight regularization $\frac{\lambda}{2}||W||^2$ to the loss function

- During gradient descent, subtract $\lambda w$ from each weight $w$
  - intuitively, implements weight decay
Small model vs. Big Model + Regularize
Ensembles of Neural Networks

- Train multiple independent models, average their predictions
- Improvements are more dramatic with higher model variety
- Few approaches to forming an ensemble
  - **Same model, different initializations**
    - train multiple models with the best set of hyperparameters (found through cross validation) but with different random initialization.
    - drawback is that the variety is only due to initialization
  - **Top models discovered during cross-validation**
    - Use cross-validation to determine the best hyperparameters, then pick the top few
    - Improves ensemble variety but has the danger of including suboptimal models
    - practical, does not require additional retraining of models after cross-validation
  - **Different checkpoints of a single model**
    - Take different “checkpoints” of a single network over time
    - Lacks variety, but very cheap
  - **Running average of parameters during training**
    - Maintain a second copy of the network’s weights in memory that maintains an exponentially decaying sum of previous weights during training
    - This way you’re averaging the state of the network over last several iterations
Practical Training Tips: Initialization

• Initialization parameters for \( W \)
  • do not set all the parameters \( W \) equal
    • all units compute the same output, gradient descent updates are the same
  • can initialize \( W \) to small random numbers
  • if using RELU, better initialize with \( \text{randn}(n) \frac{2}{\sqrt{n}} \), where \( n \) is number of inputs to the unit

• Biases \( b \) usually initialized to 0
  • with ReLU often initialize to small positive number, like 0.1
Practical Training Tips: Learning Rate

• Loss $L(w)$ should decrease during gradient descent
  • If $L(w)$ oscillates, $\alpha$ is too large, decrease it
  • If $L(w)$ goes down but very slowly, $\alpha$ is too small, increase it
• Typically cross-validate learning rates from $10^{-2}$ to $10^{-5}$
• Helps to adjust $\alpha$ at the training time, especially for many layered (deep) networks
  • Step decay
    • reduce learning rate by some factor every few epochs
    • i.e. by a factor 0.5 every 5 epochs, or by 0.1 every 20 epochs
• Exponential decay
  • $\alpha=\alpha_0 e^{-kt}\alpha$, where $\alpha_0, k$ are hyperparameters and $t$ is epoch number
• 1/$t$ decay
  • $\alpha=\alpha_0/(1+kt)$ where $\alpha_0, k$ are hyperparameters and $t$ is epoch number
• Err on the side of slower decay, if time budget allows
Practical Training Tips: Validation/Training Accuracy

- Track number of epoch vs. validation/training accuracy

![Graph showing training and validation accuracy over epochs]

- Not much overfitting, increase network capacity?
- Strong overfitting, increase regularization?
Practical Training Tips: Momentum

• Add temporal average direction in which weights have been moving recently
• Parameter vector will build up velocity in direction that has consistent gradient
• Helps avoid local minima and speed up descent in flat (plateau) regions
• Previous direction: $\Delta w^t = w^t - w^{t-1}$
• Weight update rule with momentum
  • common to set $\beta \in (0.6, 0.9)$, also can cross-validate

$$w^{t+1} = w^t + (1 - \beta) \nabla L(w^t) + \beta \Delta w^{t-1}$$

steepest descent direction
previously direction
Practical Training Tips: Normalization

- Features should be normalized for faster convergence.
- Suppose fish length is 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 will do.
  - Test samples should be normalized exactly as training samples.
- Images are already roughly normalized.
  - Intensity/color are in the range [0, 255].
  - Usually subtract mean image from training data, zero-centers data.
    - Mean computed on training data only.
    - Subtracted from test data as well.
Training NN: How Many Epochs?

- 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

- Learn when to stop training through validation
Other Practical Training Tips

- Before training on full dataset, make sure can overfit on a small portion of the data
  - turn regularization off
- Search hyperparameters on coarse scale for a few epoch, and then on finer scale for more epochs
  - random search might be better than grid search