ECE417: Neural Networks

Mark Hasegawa-Johnson

University of Illinois

2015/04/19
Two-Layer Feedforward Neural Network

\[ \vec{z} = h(\vec{x}, U, V) \]

\[ z_\ell = g(b_\ell) \quad \vec{z} = g(\vec{b}) \]

\[ b_\ell = v_{k0} + \sum_{k=1}^{q} v_{\ell k} y_k \quad \vec{b} = V\vec{y} \]

\[ y_k = f(a_k) \quad \vec{y} = f(\vec{a}) \]

\[ a_k = u_{k0} + \sum_{j=1}^{p} u_{kj} x_j \quad \vec{a} = U\vec{x} \]

\[ \vec{x} \text{ is the input vector} \]
Neural Network = Universal Approximator

Assume...

- Linear Output Nodes: \( g(b) = b \)
- Smoothly Nonlinear Hidden Nodes: \( f'(a) = \frac{df}{da} \) finite
- Smooth Target Function: \( \vec{z} = h(\vec{x}, U, V) \) approximates \( \vec{\zeta} = h^*(\vec{x}) \in \mathcal{H} \), where \( \mathcal{H} \) is some class of sufficiently smooth functions of \( \vec{x} \) (functions whose Fourier transform has a first moment less than some finite number \( C \))
- There are \( q \) hidden nodes, \( y_k, 1 \leq k \leq q \)
- The input vectors are distributed with some probability density function, \( p(\vec{x}) \), over which we can compute expected values.

Then (Barron, 1993) showed that...

\[
\max_{h^*(\vec{x}) \in \mathcal{H}} \min_{U, V} E \left[ \left| h(\vec{x}, U, V) - h^*(\vec{x}) \right|^2 \right] \leq O \left\{ \frac{1}{q} \right\}
\]
Neural Network Problems: Outline of Remainder of this Talk

1. **Knowledge-Based Design.** Given $U, V, f, g$, what kind of function is $h(\vec{x}, U, V)$? Can we draw $\vec{z}$ as a function of $\vec{x}$? Can we heuristically choose $U$ and $V$ so that $\vec{z}$ looks kinda like $\vec{\zeta}$?

2. **Error Metric.** In what way should $\vec{z} = h(\vec{x})$ be “similar to” $\vec{\zeta} = h^*(\vec{x})$?

3. **Local Optimization: Gradient Descent with Back-Propagation.** Given an initial $U, V$, how do I find $\hat{U}, \hat{V}$ that more closely approximate $\vec{\zeta}$?

4. **Global Optimization: Simulated Annealing.** How do I find the globally optimum values of $U$ and $V$?
Outline

1. Intro
2. Knowledge-Based Design
3. Error Metric
4. Gradient Descent
5. Simulated Annealing
6. Lab Review
7. Conclusions
Synapse, First Layer: \( a_k = u_{k0} + \sum_{j=1}^{2} u_{kj} x_j \)
Axon, First Layer: $y_k = \tanh(a_k)$
Synapse, Second Layer: \( b_\ell = v_{\ell 0} + \sum_{k=1}^{2} v_{\ell k} y_k \)
Axon, Second Layer: \( z_\ell = \text{sign}(b_\ell) \)
Step and Logistic nonlinearities

Unit Step: \( g(b) = u(b) \)

Logistic: \( g(b) = \frac{1}{1+e^{-b}} \)

Signum and Tanh nonlinearities

Signum: \( g(b) = \text{sign}(b) \)

Tanh: \( g(b) = \frac{e^b - e^{-b}}{e^b + e^{-b}} \)
**“Linear Nonlinearity” and ReLU**

**Linear:** \( g(b) = b \)

**ReLU:** \( g(b) = \max(0, b) \)

**Max and Softmax**

**Max:**

\[
z_\ell = \begin{cases} 
1 & b_\ell = \max_m b_m \\
0 & \text{otherwise} 
\end{cases}
\]

**Softmax:**

\[
z_\ell = \frac{e^{b_\ell}}{\sum_m e^{b_m}}
\]
Outline

1. Intro
2. Knowledge-Based Design
3. Error Metric
4. Gradient Descent
5. Simulated Annealing
6. Lab Review
7. Conclusions
Error Metric: How should \( h(\vec{x}) \) be “similar to” \( h^*(\vec{x}) \)?

Linear output nodes:

**Minimum Mean Squared Error (MMSE)**

\[
U^*, V^* = \text{arg min } E_n = \text{arg min } \frac{1}{n} \sum_{i=1}^{n} |\vec{\zeta}_i - \vec{z}(x_i)|^2
\]

**MMSE Solution:** \( \vec{z} = E \left[ \vec{\zeta} | \vec{x} \right] \)

If the training samples \((\vec{x}_i, \vec{\zeta}_i)\) are i.i.d., then

\[
E_\infty = E \left[ |\vec{\zeta} - \vec{z}|^2 \right]
\]

\(E_\infty\) is minimized by

\[
\vec{z}_{MMSE}(\vec{x}) = E \left[ \vec{\zeta} | \vec{x} \right]
\]
Error Metric: How should $h(\vec{x})$ be “similar to” $h^*(\vec{x})$?

Logistic output nodes:

**Binary target vector**

Suppose

$$\zeta_\ell = \begin{cases} 
1 & \text{with probability } P_\ell(\vec{x}) \\
0 & \text{with probability } 1 - P_\ell(\vec{x})
\end{cases}$$

and suppose $0 \leq z_\ell \leq 1$, e.g., logistic output nodes.

**MMSE Solution**: $z_\ell = \Pr \{\zeta_\ell = 1|\vec{x}\}$

$$E[\zeta_\ell|\vec{x}] = 1 \cdot P_\ell(\vec{x}) + 0 \cdot (1 - P_\ell(\vec{x})) = P_\ell(\vec{x})$$

So the MMSE neural network solution is

$$z_{\ell, MMSE}(\vec{x}) = P_\ell(\vec{x})$$
Error Metric: How should $h(\vec{x})$ be “similar to” $h^*(\vec{x})$?

Softmax output nodes:

One-Hot Vector, MKLD Solution: $z_\ell = \Pr \{\zeta_\ell = 1 | \vec{x} \}$

- Suppose $\vec{\zeta}_i$ is a “one hot” vector, i.e., only one element is “hot” ($\zeta_{\ell(i)} = 1$), all others are “cold” ($\zeta_{mi} = 0$, $m \neq \ell(i)$).
- MMSE will approach the solution $z_\ell = \Pr \{\zeta_\ell = 1 | \vec{x} \}$, but there’s no guarantee that it’s a correctly normalized pmf ($\sum z_\ell = 1$) until it has fully converged.
- MKLD also approaches $z_\ell = \Pr \{\zeta_\ell = 1 | \vec{x} \}$, and guarantees that $\sum z_\ell = 1$. MKLD is also more computationally efficient, if $\vec{\zeta}$ is a one-hot vector.

MKLD = Minimum Kullback-Leibler Distortion

$$D_n = \frac{1}{n} \sum_{i=1}^{n} \sum_{\ell=1}^{r} \zeta_{\ell i} \log \left( \frac{\zeta_{\ell i}}{z_{\ell i}} \right) = -\frac{1}{n} \sum_{i=1}^{n} \log z_{\ell(i),i}$$
Error Metrics Summarized

- Use MSE to achieve $\vec{z} = E \left[ \vec{\zeta} | \vec{x} \right]$. That’s almost always what you want.
- If $\vec{\zeta}$ is a one-hot vector, then use KLD (with a softmax nonlinearity on the output nodes) to guarantee that $\vec{z}$ is a properly normalized probability mass function, and for better computational efficiency.
- If $\zeta_\ell$ is binary, but not necessarily one-hot, then use MSE (with a logistic nonlinearity) to achieve $z_\ell = \Pr \left\{ \zeta_\ell = 1 | \vec{x} \right\}$.
- If $\zeta_\ell$ is signed binary ($\zeta_\ell \in \{-1, +1\}$, then use MSE (with a tanh nonlinearity) to achieve $z_\ell = E \left[ \zeta_\ell | \vec{x} \right]$.

After you’re done training, you can make your cell phone app more efficient by throwing away the uncertainty:

- Replace softmax output nodes with max
- Replace logistic output nodes with unit-step
- Replace tanh output nodes with signum
Outline

1. Intro
2. Knowledge-Based Design
3. Error Metric
4. Gradient Descent
5. Simulated Annealing
6. Lab Review
7. Conclusions
Gradient Descent = Local Optimization

Neural Net Error Surface (Schematic)
Gradient Descent = Local Optimization

Given an initial $U$, $V$, find $\hat{U}$, $\hat{V}$ with lower error.

$$\hat{u}_{kj} = u_{kj} - \eta \frac{\partial E_n}{\partial u_{kj}}$$

$$\hat{v}_{\ell k} = v_{\ell k} - \eta \frac{\partial E_n}{\partial v_{\ell k}}$$

$\eta =$Learning Rate

- If $\eta$ too large, gradient descent won’t converge. If too small, convergence is slow. Usually we pick $\eta \approx 0.001$ and cross our fingers.
- Second-order methods like L-BFGS choose an optimal $\eta$ at each step, so they’re MUCH faster.
Computing the Gradient

OK, let’s compute the gradient of $E_n$ with respect to the $V$ matrix. Remember that $V$ enters the neural net computation as $b_{\ell i} = \sum_k v_{\ell k} y_{ki}$, and then $z$ depends on $b$ somehow. So . . .

$$\frac{\partial E_n}{\partial v_{\ell k}} = \sum_{i=1}^{n} \left( \frac{\partial E_n}{\partial b_{\ell i}} \right) \left( \frac{\partial b_{\ell i}}{\partial v_{\ell k}} \right)$$

$$= \sum_{i=1}^{n} \epsilon_{\ell i} y_{ki}$$

where the last line only works if we define $\epsilon_{\ell i}$ in a useful way:

Back-Propagated Error

$$\epsilon_{\ell i} = \frac{\partial E_n}{\partial b_{\ell i}} = \frac{2}{n} (z_{\ell i} - \zeta_{\ell i}) g'(b_{\ell i})$$

where $g'(b) = \frac{\partial g}{\partial b}$.
\[ \vec{z} = h(\vec{x}, U, V) \]

\[ z_\ell = g(b_\ell) \quad \tilde{z} = g(\tilde{b}) \]

\[ b_\ell = v_{k0} + \sum_{k=1}^{q} v_{\ell k} y_k \quad \tilde{b} = V \tilde{y} \]

\[ y_k = f(a_k) \quad \tilde{y} = f(\tilde{a}) \]

\[ a_k = u_{k0} + \sum_{j=1}^{p} u_{kj} x_j \quad \tilde{a} = U \tilde{x} \]

\[ \vec{x} \text{ is the input vector} \]

**Back-Propagating to the First Layer**

\[
\frac{\partial E_n}{\partial u_{kj}} = \sum_{i=1}^{n} \left( \frac{\partial E_n}{\partial a_{ki}} \right) \left( \frac{\partial a_{ki}}{\partial u_{kj}} \right) = \sum_{i=1}^{n} \delta_{ki} x_{ji}
\]

where...

\[
\delta_{ki} = \frac{\partial E_n}{\partial a_{ki}} = \sum_{\ell=1}^{r} \epsilon_{\ell i} v_{\ell k} f'(a_{ki})
\]
The Back-Propagation Algorithm

\[ \hat{V} = V - \eta \nabla_V E_n, \quad \hat{U} = U - \eta \nabla_U E_n \]

\[ \nabla_V E_n = EY^T, \quad \nabla_U E_n = DX^T \]

\[ Y = [\vec{y_1}, \ldots, \vec{y_n}], \quad X = [\vec{x_1}, \ldots, \vec{x_n}] \]

\[ E = [\vec{\epsilon_1}, \ldots, \vec{\epsilon_n}], \quad D = [\vec{\delta_1}, \ldots, \vec{\delta_n}] \]

\[ \vec{\epsilon}_i = \frac{2}{n} g'(\vec{b_i}) \odot (\vec{z}_i - \vec{\zeta_i}), \quad \vec{\delta}_i = f'(\vec{a_i}) \odot V^T \vec{\epsilon}_i \]

...where \( \odot \) means element-wise multiplication of two vectors; \( g'(\vec{b}) \) and \( f'(\vec{a}) \) are element-wise derivatives of the \( g(\cdot) \) and \( f(\cdot) \) nonlinearities.
Derivatives of the Nonlinearities

**Logistic**

Logistic: \( g(b) = \frac{1}{1 + e^{-b}} \)

Logistic Derivative: \( g'(b) = g(b)(1-g(b)) \)

**Tanh**

Tanh: \( g(b) = \frac{e^b - e^{-b}}{e^b + e^{-b}} \)

Tanh Derivative: \( g'(b) = (1-g^2(b)) \)

**ReLU**

ReLU: \( g(b) = \max(0, b) \)

Unit Step: \( g(b) = u(b) \)
Outline

1. Intro
2. Knowledge-Based Design
3. Error Metric
4. Gradient Descent
5. Simulated Annealing
6. Lab Review
7. Conclusions
Simulated Annealing: How can we find the globally optimum $U, V$?

- Gradient descent finds a local optimum. The $\hat{U}, \hat{V}$ you end up with depends on the $U, V$ you started with.
- How can you find the **global optimum** of a non-convex error function?
- The answer: Add randomness to the search, in such a way that...

\[ P(\text{reach global optimum}) \xrightarrow{t \to \infty} 1 \]
Take a random step. If it goes downhill, do it.
- Take a random step. If it goes downhill, do it.
- If it goes uphill, SOMETIMES do it.
- Take a random step. If it goes downhill, do it.
- If it goes uphill, SOMETIMES do it.
- Uphill steps become less probable as $t \to \infty$
Simulated Annealing: Algorithm

FOR $t = 1$ TO $\infty$, DO

1. Set $\hat{U} = U + \text{RANDOM}$

2. If your random step caused the error to decrease ($E_n(\hat{U}) < E_n(U)$), then set $U = \hat{U}$
   (prefer to go downhill)

3. Else set $U = \hat{U}$ with probability $P$
   (... but sometimes go uphill!)
   
   $P = \exp\left(-\frac{E_n(\hat{U}) - E_n(U)}{\text{Temperature}}\right)$
   (Small steps uphill are more probable than big steps uphill.)
   
   Temperature $= T_{\text{max}} / \log(t + 1)$
   (Uphill steps become less probable as $t \to \infty$.)

4. Whenever you reach a local optimum ($U$ is better than both the preceding and following time steps), check to see if it’s better than all preceding local optima; if so, remember it.
Convergence Properties of Simulated Annealing

(Hajek, 1985) proved that, if we start out in a “valley” that is separated from the global optimum by a “ridge” of height $T_{\max}$, and if the temperature at time $t$ is $T(t)$, then simulated annealing converges in probability to the global optimum if

$$\sum_{t=1}^{\infty} \exp \left( - \frac{T_{\max}}{T(t)} \right) = +\infty$$

For example, this condition is satisfied if

$$T(t) = \frac{T_{\max}}{\log(t + 1)}$$
Outline

1. Intro
2. Knowledge-Based Design
3. Error Metric
4. Gradient Descent
5. Simulated Annealing
6. Lab Review
7. Conclusions
Here’s the dataset

WS15 ANN Lab, Reference Labels, ER=0
You’ll have to plot it many times, so I recommend writing a plot function

```matlab
function ER = nnplot(X,Z,ZETA,STRING,fignum)
    [p,n]=size(X);
    ER=sum(ZETA.*Z<0)/n;
    figure(fignum);
    plot(X(1,Z<0),X(2,Z<0),'r.',X(1,Z>0),X(2,Z>0),'b.);
    title(sprintf('WS15 ANN Lab, %s, ER=%g',STRING,ER));
```
Knowledge-based design: set each row of $\mathbf{U}$ to be a line segment, $u_0 + u_1 x_1 + u_2 x_2 = 0$, on the decision boundary. $u_0$ is an arbitrary scale factor; $u_0 = -20$ makes the tanh work well.

$[x1,x2]=\text{ginput}(2)$;
$u0=-20; \ % \text{Arbitrary scale factor}$
$u = -\text{inv}([x1,x2]) *[u0;u0];$
$\mathbf{U}(1,:) = [u0,u(1),u(2)];$
Check your math by plotting $x_2 = -\frac{u_0}{u_2} - \frac{u_1}{u_2}x_1$

\begin{verbatim}
nnplot(X,ZETA,ZETA,'Reference Labels',1); hold on; plot([0,1],[-u(0)/u(2)]+[0,-u(1)/u(2)],'g-'); hold off;
\end{verbatim}
Here are 3 such segments, mapping out the lowest curve:

```matlab
for m=1:3,
    plot([0 1],-U(m,1)/U(m,3)+[0,-U(m,2)/U(m,3)]);
end
```
(1) Reflect through \( x_2 = -0.75 \), and (2) Shift upward:

\[
\begin{align*}
U_{\text{foo}} &= [U; U(:,1)-1.5*U(:,3), U(:,2), -U(:,3)]; \\
U_{\text{bar}} &= [U_{\text{foo}}; U_{\text{foo}}-[0.5*U_{\text{foo}}(:,3), \text{zeros}(6,2)]]; \\
U &= [U_{\text{bar}}; U_{\text{bar}}-[U_{\text{bar}}(:,3), \text{zeros}(12,2)]]; \\
\end{align*}
\]
nnclassify.m: Error Rate = 14%

function [Z,Y]=nnclassify(X,U,V)
Y = tanh(U*[ones(1,n); X]);
Z = tanh(V*[ones(1,n); Y]);
**nnbackprop.m: Error Rate = 2.8%**

```matlab
function [EPSILON,DELTA]=nnbackprop(X,Y,Z,ZETA,V)
EPSILON = 2* (1-Z.^2) .* (Z-ZETA);
DELTA = (1-Y.^2) .* (V(:,2:(q+1))' * EPSILON);
```
But with random initialization: Error Rate = 28%

Urand = [0.02*randn(q,p+1)];
Vrand = [0.02*randn(r,q+1)];
[Uc,Vc] = nndescent(X,ZETA,Urand,Vrand,0.1,1000);
[Zc,Yc] = nnclassify(X,Uc,Vc);
WS15 ANN Lab, Simulated Annealing, ER=0.051
nnanneal.m: Error Rate = 5.1%

function [Es,Us,Vs] = nnanneal(X,ZETA,U0,V0,ETA,T)
for t=1:T,
    U1=U0+randn(q,p+1); V1=V0+randn(r,q+1);
    ER1 = sum(nnclassify(X,U1,V1).*ZETA<0)/n;
    if ER1 < ER0,
        U0=U1;V0=V1;ER0=ER1;
    else
        P = exp(-(ER1-ER0)*log(t+1)/ridge);
        if rand() < P,
            U0=U1;V0=V1;ER0=ER1;
        end
    end
end
Here's one that Amit tried based on my mistaken early draft of the instructions for this lab. Error Rate: 28%

\[
\text{temperature} = \frac{\text{ridge}}{\sqrt{t}};
\]

instead of the correct form,

\[
\text{temperature} = \frac{\text{ridge}}{\log(t+1)};
\]
...and Amit solved it using Geometric Annealing. Error Rate: 0.67%

- Smaller random steps: \( \Delta U \sim \mathcal{N}(0, 1e^{-4}) \) instead of \( \mathcal{N}(0, 1) \), and only one weight at a time instead of all weights at once

- Geometric annealing: temperature cools geometrically \( T(t) = \alpha T(t - 1) \) rather than logarithmically \( T(t) = c / \log(t + 1) \)
Simulated Annealing: More Results

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>$c$ or $\alpha$</th>
<th>$t$</th>
<th>Error Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hajek Cooling ($T = c / \log(t + 1)$)</td>
<td>1, $10^{-4}$</td>
<td>52356</td>
<td>5.1%</td>
</tr>
<tr>
<td>Geometric Annealing ($T(t) = \alpha T(t - 1)$)</td>
<td>0.7, 0.8, 0.9</td>
<td>500</td>
<td>0.43%, 0.40%, 0.80%</td>
</tr>
</tbody>
</table>
More Comments on Simulated Annealing

- Gaussian random walk results in very large weights
  - I fought this using the `mod` operator, to map weights back to the range $[-25, 25]$
  - I suspect it matters, but I'm not sure
- Every time you reach a new low error,
  - Store it, and its associated weights, in case you never find it again, and
  - Print it on the screen (using `disp` and `sprintf`) so you can see how your code is doing
- Simulated annealing can take a really long time.
Real-World Randomness: Stochastic Gradient Descent (SGD)

- SGD is the following algorithm. For $t=1:T$,
  1. Randomly choose a small subset of your training data (a **minibatch**: strictly speaking, SGD is minibatch size of $m=1$, but practical minibatches are typically $m \sim 100$)
  2. Perform a complete backprop iteration using the minibatch.

- Advantage of SGD over Simulated Annealing: computational complexity
  - Instead of introducing randomness with a random weight update ($\mathcal{O}\{n\}$), we introduce randomness by randomly sampling the dataset ($\mathcal{O}\{m\}$)
  - Matters a lot when $n$ is large

- Disadvantage of SGD over Simulated Annealing: It’s not theoretically proven to converge to a global optimum
  - ...but it works in practice, if training dataset is big enough.
Conclusions

- Back-prop.
  - You need to know how to do it.
  - ...but back-prop is only useful if you start from a good initial set of weights, or if you have good randomness.

- Knowledge-based initialization
  - Sometimes, it helps if you understand what you’re doing.

- Stochastic search.
  - Simulated annealing: guaranteed performance, high complexity.
  - Stochastic gradient descent: not guaranteed, but low complexity. Incidentally, I haven’t tried it yet on hard2d.txt; if you try it, please tell me how it works.

Confucius Says... Local optimization makes a good idea better.