Outline

1. What is a Neural Net?
2. Knowledge-Based Design
3. Nonlinearities
4. Error Metric
5. Gradient Descent
Two-Layer Feedforward Neural Network

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

which is decomposed as…

\[ z_\ell = g(b_\ell) \]
\[ \vec{z} = g(\vec{b}) \]
\[ b_\ell = v_{k0} + \sum_{k=1}^{q} v_{\ell k} y_k \]
\[ \vec{b} = V \vec{y} \]
\[ y_k = f(a_k) \]
\[ \vec{y} = f(\vec{a}) \]
\[ a_k = u_{k0} + \sum_{j=1}^{p} u_{kj} x_j \]
\[ \vec{a} = U \vec{x} \]

\( \vec{x} \) is the input vector
A Neural Net is Made Of...

- Linear transformations: \( \vec{a} = U\vec{x}, \vec{b} = V\vec{y} \), one per layer.
- Scalar nonlinearities: \( \vec{y} = f(\vec{a}) \) means that, element-by-element, \( y_k = f(a_k) \) for some nonlinear function \( f(\cdot) \).
- The nonlinearities can all be different, if you want. For today, I’ll assume that all nodes in the first layer use one function \( f(\cdot) \), and all nodes in the second layer use some other function \( g(\cdot) \).
- Networks with more than two layers are called “Deep Neural Networks” (DNN). I won’t talk about them today.

Andrew Barron (1993) proved that combining two layers of linear transforms, with one scalar nonlinearity between them, is enough to model **any** multivariate nonlinear function \( \vec{z} = h(\vec{x}) \).
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 . . .

\[
\max_{h^*(\vec{x}) \in \mathcal{H}} \min_{U, V} E \left[ h(\vec{x}, U, V) - h^*(\vec{x}) \right]^2 \leq O \left\{ \frac{1}{q} \right\}
\]
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. **Nonlinearities.** They come in pairs: the test-time nonlinearity, and the training-time nonlinearity.

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

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

1. What is a Neural Net?
2. Knowledge-Based Design
3. Nonlinearities
4. Error Metric
5. Gradient Descent
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: $z_\ell = \text{sign}(b_\ell)$
Outline

1. What is a Neural Net?
2. Knowledge-Based Design
3. Nonlinearities
4. Error Metric
5. Gradient Descent
The nonlinearities come in pairs: (1) the **test-time nonlinearity** is the one that you use in the output layer of your learned classifier, e.g., in the app on your cell phone (2) the **training-time nonlinearity** is used in the output layer during training, and in the hidden layers during both training and test.

<table>
<thead>
<tr>
<th>Application</th>
<th>Test-Time Output Nonlinearity</th>
<th>Training-Time Output &amp; Hidden Nonlinearity</th>
</tr>
</thead>
<tbody>
<tr>
<td>{0, 1} classification</td>
<td>step</td>
<td>logistic or ReLU</td>
</tr>
<tr>
<td>{-1, +1} classification</td>
<td>signum</td>
<td>tanh</td>
</tr>
<tr>
<td>multinomial classification</td>
<td>argmax</td>
<td>softmax</td>
</tr>
<tr>
<td>regression</td>
<td>linear</td>
<td>(hidden nodes must be nonlinear)</td>
</tr>
</tbody>
</table>
**Step and Logistic nonlinearities**

- **Unit Step**: $g(b) = u(b)$
  
  ![Unit Step Graph](image)

- **Logistic**: $g(b) = \frac{1}{1+e^{-b}}$
  
  ![Logistic Graph](image)

---

**Signum and Tanh nonlinearities**

- **Signum**: $g(b) = \text{sign}(b)$
  
  ![Signum Graph](image)

- **Tanh**: $g(b) = \frac{e^b - e^{-b}}{e^b + e^{-b}}$
  
  ![Tanh Graph](image)
“Linear Nonlinearity” and ReLU

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

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

Argmax and Softmax

**Argmax:**

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

**Softmax:**

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

1. What is a Neural Net?
2. Knowledge-Based Design
3. Nonlinearities
4. Error Metric
5. Gradient Descent
Minimum Mean Squared Error (MMSE)

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

Why would we want to use this metric?

If the training samples \((\vec{x}_i, \vec{\zeta}_i)\) are i.i.d., then in the limit as the number of training tokens goes to infinity,

\[ h(\vec{x}) \to E \left[ \vec{\zeta} | \vec{x} \right] \]
Error Metric: MMSE for Binary Target Vector

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.

Why does MMSE make sense for binary targets?

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

So the MMSE neural network solution is

\[ h(\vec{x}) \rightarrow E \left[ \zeta | \vec{x} \right] = P_\ell(\vec{x}) \]
Encoding the Neural Net Output using a “One-Hot Vector”

- Suppose $\vec{\zeta}_i$ is a “one hot” vector, i.e., only one element is “hot” ($\zeta_{\ell(i),i} = 1$), all others are “cold” ($\zeta_{mi} = 0, m \neq \ell(i)$).
- Training logistic output nodes with MMSE training will approach the solution $z_\ell = \Pr\{\zeta_\ell = 1|x\}$, but there’s no guarantee that it’s a correctly normalized pmf ($\sum z_\ell = 1$) until it has fully converged.
- Softmax output nodes guarantee that $\sum z_\ell = 1$.

Softmax output nodes

$$z_\ell = \frac{e^{b_\ell}}{\sum_m e^{b_m}}$$
The softmax nonlinearity is “matched” to an error criterion called “cross-entropy,” in the sense that its derivative can be simplified to have a very, very simple form.

- $\zeta_{\ell,i}$ is the true reference probability that observation $\vec{x}_i$ is of class $\ell$. In most cases, this “reference probability” is either 0 or 1 (one-hot).
- $z_{\ell,i}$ is the neural network’s hypothesis about the probability that $\vec{x}_i$ is of class $\ell$. The softmax function constrains this to be $0 \leq z_{\ell,i} \leq 1$ and $\sum_\ell z_{\ell,i} = 1$.

The average cross-entropy between these two distributions is

$$E = -\frac{1}{n} \sum_{i=1}^n \sum_\ell \zeta_{\ell,i} \log z_{\ell,i}$$
Suppose token $\tilde{x}_i$ is of class $\ell^*$, meaning that $\zeta_{\ell^*,i} = 1$, and all others are zero. Then cross-entropy is just the neural net's estimate of the negative log probability of the correct class:

$$E = -\frac{1}{n} \sum_{i=1}^{n} \log z_{\ell^*,i}$$

In other words, $E$ is the average of the negative log probability of each training token:

$$E = -\frac{1}{n} \sum_{i=1}^{n} E_i, \quad E_i = - \log z_{\ell^*,i}$$
Cross-Entropy is matched to softmax

Now let’s plug in the softmax:

\[ E_i = -\log z_{\ell^*,i}, \quad z_{\ell^*,i} = \frac{e^{b_{\ell^*,i}}}{\sum_k e^{b_{ki}}} \]

Its gradient with respect to the softmax inputs, \( b_{mi} \), is

\[
\frac{\partial E_i}{\partial b_{mi}} = -\frac{1}{z_{\ell^*,i}} \frac{\partial z_{\ell^*,i}}{\partial b_{mi}} = \begin{cases} 
-\frac{1}{z_{\ell^*,i}} \left( \frac{e^{b_{\ell^*,i}}}{\sum_k e^{b_{ki}}} - \left( \frac{e^{b_{\ell^*,i}}}{\sum_k e^{b_{ki}}} \right)^2 \right) & m = \ell^* \\
-\frac{1}{z_{\ell^*,i}} \left( -\frac{e^{b_{\ell^*,i}} e^{b_{mi}}}{\left( \sum_k e^{b_{ki}} \right)^2} \right) & m \neq \ell^* 
\end{cases}
\]

\[ = z_{mi} - \zeta_{mi} \]
Error Metrics Summarized

- Use MSE to achieve $\vec{z} = E [\vec{\zeta} | \vec{x}]$. That’s almost always what you want.

- If $\vec{\zeta}$ is a one-hot vector, then use Cross-Entropy (with a softmax nonlinearity on the output nodes) to guarantee that $\vec{z}$ is a properly normalized probability mass function, and because it gives you the amazingly easy formula
  \[
  \frac{\partial E_i}{\partial b_{mi}} = z_{mi} - \zeta_{mi}.
  \]

- If $\zeta_\ell$ is binary, but not necessarily one-hot, then use MSE (with a logistic nonlinearity) to achieve $z_\ell = \Pr \{ \zeta_\ell = 1 | \vec{x} \}$. 
Outline

1. What is a Neural Net?
2. Knowledge-Based Design
3. Nonlinearities
4. Error Metric
5. Gradient Descent
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.

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

$$\hat{v}_{lk} = v_{lk} - \eta \frac{\partial E}{\partial v_{lk}}$$

$\eta = $Learning Rate

- If $\eta$ too large, gradient descent won't converge. If too small, convergence is slow. Usually we pick $\eta \approx 0.001$, then see whether it converges or not; if not, we tweak $\eta$ and then try again.

- Second-order methods like Newton’s algorithm, L-BFGS, ADAM, and Hessian-free optimization choose an optimal $\eta$ at each step, so they’re MUCH faster.
Computing the Gradient

\[ E = \frac{1}{n} \sum_{i=1}^{n} E_i, \quad E_i = \text{cross-entropy or MMSE} \]

\[ \frac{\partial E}{\partial v_{\ell k}} = \frac{1}{n} \sum_{i=1}^{n} \left( \frac{\partial E}{\partial b_{\ell i}} \right) \left( \frac{\partial b_{\ell i}}{\partial v_{\ell k}} \right) = \frac{1}{n} \sum_{i=1}^{n} \epsilon_{\ell i} y_{ki} \]

where I've used one thing you already know, and one new definition. Here's the thing you already know:

\[ b_{\ell i} = \sum_{k} v_{\ell k} y_{ki}, \quad \text{therefore} \quad \frac{\partial b_{\ell i}}{\partial v_{\ell k}} = y_{ki} \]

Here's the new definition:

\[ \epsilon_{\ell i} = \frac{\partial E_i}{\partial b_{\ell i}} = \left\{ \begin{array}{ll} z_{\ell i} - \zeta_{\ell i} & \text{Cross-Entropy with Softmax} \\ (z_{\ell i} - \zeta_{\ell i})g'(b_{\ell i}) & \text{MMSE with Nonlinearity } g(b) \end{array} \right. \]
Forward Propagation and Back-Propagation

\[
\frac{\partial E}{\partial v_{\ell k}} = \frac{1}{n} \sum_{i=1}^{n} \epsilon_{\ell i} y_{ki}
\]

- First, \( y_{ji} \) and \( z_{\ell i} \) are generated from \( \vec{x}_i \) in the forward pass.
- Then \( \epsilon_{\ell i} \) is generated from \( z_{\ell i} - \zeta_{\ell i} \) in the back-propagation.


\( g'(b) \): Derivatives of the Nonlinearities

**Logistic**

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

**Tanh**

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

**ReLU**

\[
\text{ReLU: } g(b) = \max(0, b)
\]
\[ \vec{z} = h(\vec{x}, U, V) \]

which is decomposed as...

\[ z_\ell = g(b_\ell) \quad \vec{z} = g(\vec{b}) \]
\[ b_\ell = v_{k0} + \sum_{k=1}^{q} v_{\ell k} y_k \quad \vec{b} = V \vec{y} \]
\[ y_k = f(a_k) \quad \vec{y} = f(\vec{a}) \]
\[ a_k = u_{k0} + \sum_{j=1}^{p} u_{kj} x_j \quad \vec{a} = U \vec{x} \]

\( \vec{x} \) is the input vector

---

**Back-Propagating to the First Layer**

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

where...

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

\[
\frac{\partial E}{\partial v_{\ell k}} = \frac{1}{n} \sum_{i=1}^{n} \epsilon_{\ell i} y_{ki}
\]

\[
\frac{\partial E}{\partial u_{kj}} = \frac{1}{n} \sum_{i=1}^{n} \delta_{ki} x_{ji}
\]

- First, \( y_{ji} \) and \( z_{\ell i} \) are generated from \( \vec{x}_i \) in the forward pass.
- Then \( \epsilon_{\ell i} \) and \( \delta_{ki} \) are generated from \( z_{\ell i} - \zeta_{\ell i} \) in the back-propagation.