Deep Learning
Feed-Forward Neural Networks
Backpropagation

Razvan C. Bunescu
Department of Computer Science @ CCI
rbunescu@uncc.edu
Neuron Function

- **Algebraic interpretation:**
  - The output of the neuron is a **linear combination** of inputs from other neurons, rescaled by the synaptic weights.
    - weights $w_i$ correspond to the synaptic weights (activating or inhibiting).
    - summation corresponds to combination of signals in the soma.
  - It is often transformed through a monotonic **activation function**.
Activation Functions

**unit step** \( f(z) = \begin{cases} 
0 & \text{if } z < 0 \\
1 & \text{if } z \geq 0 
\end{cases} \)

Perceptron

**logistic** \( f(z) = \frac{1}{1 + e^{-z}} \)

Logistic Neuron

**ReLU** \( f(z) = \begin{cases} 
0 & \text{if } z < 0 \\
z & \text{if } z \geq 0 
\end{cases} \)

Rectified Linear Unit

\( f(z) = \text{ramp}(z) = \max(0, z) \)
Perceptron vs. Logistic Neuron

- **Logistic neuron = Logistic regression:**
  - At inference time, same decision function as perceptron, for binary classification with equal misclassification costs (prove it):
    \[
    \hat{y}(x) = \begin{cases} 
    1 & \text{if } w^T x > 0 \\
    0 & \text{otherwise}
    \end{cases}
    \]
  - **Perceptron** cannot represent the XOR function:
    - **Logistic neuron, ReLU, Tanh** have the same limitation.

- How can we use (logistic) neurons to achieve better representational power?
Universal Approximation Theorem


- Let $\sigma$ be a nonconstant, bounded, and monotonically-increasing continuous function;
- Let $I_m$ denote the m-dimensional unit hypercube $[0, 1]^m$;
- Let $C(I_m)$ denote the space of continuous functions on $I_m$;

**Theorem**: Given any function $f \in C(I_m)$ and $\varepsilon > 0$, there exist an integer $N$ and real constants $\alpha_i, b_i \in \mathbb{R}, \mathbf{w}_i \in \mathbb{R}^m$, where $i = 1, ..., N$, such that:

$$\left| F(x) - f(x) \right| < \varepsilon, \quad \forall x \in I_m$$

where

$$F(x) = \sum_{i=1}^{N} \alpha_i \sigma(\mathbf{w}_i^T x + b_i)$$

Universal Approximation Theorem


\[ F(x) = \sum_{i=1}^{N} \alpha_i \sigma(w_i^T x + b_i) \]

\[ |F(x) - f(x)| < \varepsilon, \forall x \in I_m \]

\( m = 3, \ N = 3 \)

\( x = [x_1, x_2, x_3] \)

\( w_i = [w_{i1}, w_{i2}, w_{i3}] \)
Neural Network Model

- Put together many neurons in layers, such that the output of a neuron on layer $l$ can be the input of another neuron on layer $l + 1$:
Feed-Forward Neural Networks
The Importance of Representation

http://www.deeplearningbook.org
From Cartesian to Polar Coordinates

- **Manually engineered:**
  \[
  r = \sqrt{x^2 + y^2} \\
  \theta = \tan^{-1} \left| \frac{y}{x} \right| \text{ (first quadrant)}
  \]

- **Learned from data:**

  Fully connected layers: linear transformation $W + \text{element-wise nonlinearity } f \rightarrow f(Wx)$
Representation Learning: Images

https://www.datarobot.com/blog/a-primer-on-deep-learning/
Representation Learning: Images

https://www.datarobot.com/blog/a-primer-on-deep-learning/
A Rapidly Evolving Field

• Used to think that training deep networks requires **greedy layer-wise pretraining**:
  - Unsupervised learning of representations with **auto-encoders** (2012).

• Better random **weight initialization** schemes now allow training deep networks from scratch.

• **Batch normalization** allows for training even deeper models (2014).
  - Replaced by the simpler **Layer Normalization** (2016).

• **Residual learning** allows training arbitrarily deep networks (2015).

• Attention-based **Transformers** replace RNNs and CNNs in NLP (2018):
Neural Network Model

- Put together many neurons in layers, such that the output of a neuron can be the input of another:
$n_l=3$ is the number of layers.

- $L_1$ is the input layer, $L_3$ is the output layer.

- $(W, b) = (W^{(1)}, b^{(1)}, W^{(2)}, b^{(2)})$ are the parameters:
  - $W^{(l)}_{ij}$ is the weight of the connection between unit $j$ in layer $l$ and unit $i$ in layer $l+1$.
  - $b^{(l)}_i$ is the bias associated unit unit $i$ in layer $l+1$.

- $a^{(l)}_i$ is the activation of unit $i$ in layer $l$, e.g. $a^{(1)}_i = x_i$ and $a^{(3)}_1 = h_{W,b}(x)$. 
Inference: Forward Propagation

- The activations in the hidden layer are:

\[
\begin{align*}
a_1^{(2)} &= f(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + W_{13}^{(1)} x_3 + b_1^{(1)}) \\
a_2^{(2)} &= f(W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + W_{23}^{(1)} x_3 + b_2^{(1)}) \\
a_3^{(2)} &= f(W_{31}^{(1)} x_1 + W_{32}^{(1)} x_2 + W_{33}^{(1)} x_3 + b_3^{(1)})
\end{align*}
\]

- The activations in the output layer are:

\[
h_{W,b}(x) = a_1^{(3)} = f(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} + b_1^{(2)})
\]

- Compressed notation:

\[
a_i^{(l)} = f(z_i^{(l)}) \quad \text{where} \quad z_i^{(2)} = \sum_{j=1}^{n} W_{ij}^{(1)} x_j + b_i^{(1)}
\]
Forward Propagation

- Forward propagation (unrolled):

\[
\begin{align*}
    a_1^{(2)} &= f(W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 + W_{13}^{(1)} x_3 + b_1^{(1)}) \\
    a_2^{(2)} &= f(W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 + W_{23}^{(1)} x_3 + b_2^{(1)}) \\
    a_3^{(2)} &= f(W_{31}^{(1)} x_1 + W_{32}^{(1)} x_2 + W_{33}^{(1)} x_3 + b_3^{(1)}) \\
    h_{W,b}(x) &= a_1^{(3)} = f(W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} + b_1^{(2)})
\end{align*}
\]

- Forward propagation (compressed):

- Element-wise application:

\[f(z) = [f(z_1), f(z_2), f(z_3)]\]
Forward Propagation

• Forward propagation (compressed):

\[
\begin{align*}
  z^{(2)} &= W^{(1)}x + b^{(1)} \\
  a^{(2)} &= f(z^{(2)}) \\
  z^{(3)} &= W^{(2)}a^{(2)} + b^{(2)} \\
  h_{W,b}(x) &= a^{(3)} = f(z^{(3)})
\end{align*}
\]

• Composed of two \textit{forward propagation steps}:

\[
\begin{align*}
  z^{(l+1)} &= W^{(l)}a^{(l)} + b^{(l)} \\
  a^{(l+1)} &= f(z^{(l+1)})
\end{align*}
\]
Multiple Hidden Units, Multiple Outputs

• Write down the forward propagation steps for:

\[ h_{w,b}(x) \]
ReLU and Generalizations

• It has become more common to use piecewise linear activation functions for hidden units:
  – **ReLU**: the rectifier activation $g(z) = \max\{0, z\}$.
  – **Absolute value ReLU**: $g(z) = |z|$.
  – **Maxout**: $g(a_1, \ldots, a_k) = \max\{a_1, \ldots, a_k\}$.
    • needs $k$ weight vectors instead of 1.
  – **Leaky ReLU**: $g(a) = \max\{0, a\} + \alpha \min(0, a)$.

$\Rightarrow$ the network computes a *piecewise linear function* (up to the output activation function).
ReLU vs. Sigmoid and Tanh

- Sigmoid and Tanh saturate for values not close to 0:
  - “kill” gradients, bad behavior for gradient-based learning.
- ReLU does not saturate for values > 0:
  - greatly accelerates learning, fast implementation.
  - fragile during training and can “die”, due to 0 gradient:
    - initialize all $b$’s to a small, positive value, e.g. 0.1.
ReLU vs. Softplus

- Softplus $g(z) = \ln(1+e^z)$ is a smooth version of the rectifier.
  - Saturates less than ReLU, yet ReLU still does better [Glorot, 2011].
Learning: Backpropagation for Regression

- Regularized sum of squares error:
  
  \[ J(W, b, x, y) = \frac{1}{2} \| h_{W,b}(x) - y \|^2 \]

  \[ J(W, b) = \frac{1}{m} \sum_{k=1}^{m} J(W, b, x^{(k)}, y^{(k)}) + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_{l+1}} \sum_{j=1}^{s_l} (W_{ij}^{(l)})^2 \]

- Gradient:
  
  \[
  \frac{\partial J(W, b)}{\partial W_{ij}^{(l)}} = \frac{1}{m} \sum_{k=1}^{m} \frac{\partial J(W, b, x^{(k)}, y^{(k)})}{\partial W_{ij}^{(l)}} + \lambda W_{ij}^{(l)}
  \]

  \[
  \frac{\partial J(W, b)}{\partial b_{i}^{(l)}} = \frac{1}{m} \sum_{k=1}^{m} \frac{\partial J(W, b, x^{(k)}, y^{(k)})}{\partial b_{i}^{(l)}}
  \]
Backpropagation for Regression

- Need to compute the gradient of the squared error with respect to a single training example \((x, y)\):

\[
J(W, b, x, y) = \frac{1}{2} \|h_{W,b}(x) - y\|^2 = \frac{1}{2} \|a^{(n_l)} - y\|^2
\]

\[
\frac{\partial J}{\partial W^{(l)}_{ij}} = ? \quad \frac{\partial J}{\partial b^{(l)}_i} = ?
\]
Learning: Regression vs. Classification

• **Regression** => *loss* = squared error:

\[
J(W, b, x, y) = \frac{1}{2} \| h_{w,b}(x) - y \|^2
\]

• **Classification** => *loss* = negative log-likelihood:

\[
J(W, b, x, y) = -\ln p(y|W, b, x)
\]

• Need to compute the gradient of the loss with respect to parameters *W*, *b*:

\[
\frac{\partial J}{\partial W_{ij}} = ? \quad \frac{\partial J}{\partial b_i} = ?
\]
NN Learning: Softmax Regression

- Consider layer $n_l$ to be the input to the softmax layer i.e. softmax output layer is $n_l+1$.

- Softmax weights stored in matrix $W^{(n_l)}$.

- $K$ classes $\Rightarrow W^{(n_l)} = \begin{bmatrix} -w_1^T & - \\ -w_2^T & - \\ \vdots & \vdots \\ -w_K^T & - \end{bmatrix}$
NN Learning: Softmax Regression

- Softmax output is \( a^{(n_l+1)} = a^{(2+1)} = \text{softmax}(z^{(n_l+1)}) = \text{softmax}(z^{(2+1)}) \)

For homework: \( n_l = 2 \)

\( a^{(1)} = x \) \( a^{(2)} \)

Softmax input

Softmax logits

Softmax weights \( W^{(n_l)} \)

Cross-entropy

\[ J(a^{(n_l+1)}, y) \]
Optional Material
Learning: Backpropagation

- Regularized sum of squares error:

\[
J(W, b, x, y) = \frac{1}{2} \| h_{W,b}(x) - y \|^2
\]

\[
J(W, b) = \frac{1}{m} \sum_{k=1}^{m} J(W, b, x^{(k)}, y^{(k)}) + \frac{\lambda}{2} \sum_{l=1}^{n_l-1} \sum_{i=1}^{s_{l+1}} \sum_{j=1}^{s_l} (W_{ij}^{(l)})^2
\]

- Gradient:

\[
\frac{\partial J(W, b)}{\partial W_{ij}^{(l)}} = \frac{1}{m} \sum_{k=1}^{m} \frac{\partial J(W, b, x^{(k)}, y^{(k)})}{\partial W_{ij}^{(l)}} + \lambda W_{ij}^{(l)}
\]

\[
\frac{\partial J(W, b)}{\partial b_i^{(l)}} = \frac{1}{m} \sum_{k=1}^{m} \frac{\partial J(W, b, x^{(k)}, y^{(k)})}{\partial b_i^{(l)}}
\]
Univariate Chain Rule for Differentiation

• Univariate Chain Rule:

\[ f = f \circ g \circ h = f(g(h(x))) \]

\[ \frac{\partial f}{\partial x} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial h} \frac{\partial h}{\partial x} \]

• Example:

\[ f(g(x)) = 2g(x)^2 - 3g(x) + 1 \]

\[ g(x) = x^3 + 2x \]
Multivariate Chain Rule for Differentiation

- Multivariate Chain Rule:
  
  \[ f = f(g_1(x), g_2(x), ..., g_n(x)) \]

  \[ \frac{\partial f}{\partial x} = \sum_{i=1}^{n} \frac{\partial f}{\partial g_i} \frac{\partial g_i}{\partial x} \]

- Example:
  
  \[ f(g_1(x), g_2(x)) = 2g_1(x)^2 - 3g_1(x)g_2(x) + 1 \]

  \[ g_1(x) = 3x \]

  \[ g_2(x) = x^2 + 2x \]
Backpropagation: $W_{ij}^{(l)}$

- $J$ depends on $W_{ij}^{(l)}$ only through $a_i^{(l+1)}$, which depends on $W_{ij}^{(l)}$ only through $z_i^{(l+1)}$.

$$J(W, b, x, y) = \frac{1}{2} \left\| a^{(n_f)} - y \right\|^2$$

$$a_i^{(l+1)} = f(z_i^{(l+1)})$$

$$z_i^{(l+1)} = \sum_{j=1}^{s_l} W_{ij}^{(l)} a_j^{(l)} + b_i^{(l)}$$
Backpropagation: $b_i^{(l)}$

- $J$ depends on $b_i^{(l)}$ only through $a_i^{(l+1)}$, which depends on $b_i^{(l)}$ only through $z_i^{(l+1)}$.

\[
J(W, b, x, y) = \frac{1}{2} \| a^{(n_l)} - y \|^2
\]

\[
a_i^{(l+1)} = f(z_i^{(l+1)})
\]

\[
z_i^{(l+1)} = \sum_{j=1}^{s_l} W_{ij}^{(l)} a_j^{(l)} + b_i^{(l)}
\]
Backpropagation: $W_{ij}^{(l)}$ and $b_i^{(l)}$

\[
\frac{\partial J}{\partial W_{ij}^{(l)}} = \frac{\partial J}{\partial a_i^{(l+1)}} \times \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \times \frac{\partial z_i^{(l+1)}}{\partial W_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)}
\]

How to compute $\delta_i^{(l+1)}$ for all layers $l$?

\[
\frac{\partial J}{\partial b_i^{(l)}} = \frac{\partial J}{\partial a_i^{(l+1)}} \times \frac{\partial a_i^{(l+1)}}{\partial z_i^{(l+1)}} \times \frac{\partial z_i^{(l+1)}}{\partial b_i^{(l)}} = \delta_i^{(l+1)}
\]

\[
\delta_i^{(l+1)} = \delta_i^{(l+1)} + 1
\]
Backpropagation: $\delta_i^{(l)}$

$$
\delta_i^{(l)} = \frac{\partial J}{\partial a_i^{(l)}} \times \frac{\partial a_i^{(l)}}{\partial z_i^{(l)}} = \frac{\partial J}{\partial a_i^{(l)}} \times f'(z_i^{(l)})
$$

- $J$ depends on $a_i^{(l)}$ only through $a_1^{(l+1)}$, $a_2^{(l+1)}$, ...

![Diagram of neural network](image)
Backpropagation: $\delta_i^{(l)}$

- $J$ depends on $a_i^{(l)}$ only through $a_1^{(l+1)}$, $a_2^{(l+1)}$, ...

\[
\frac{\partial J}{\partial a_i^{(l)}} = \sum_{j=1}^{s_{l+1}} \left( \frac{\partial J}{\partial a_j^{(l+1)}} \times \frac{\partial a_j^{(l+1)}}{\partial a_i^{(l)}} \right) = \sum_{j=1}^{s_{l+1}} \frac{\partial J}{\partial a_j^{(l+1)}} \times \frac{\partial a_j^{(l+1)}}{\partial z_j^{(l+1)}} \times \frac{\partial z_j^{(l+1)}}{\partial a_i^{(l)}} \times \delta_j^{(l+1)} \times W_{ji}^{(l)}
\]

- Therefore, $\delta_i^{(l)}$ can be computed as:

\[
\delta_i^{(l)} = \frac{\partial J}{\partial a_i^{(l)}} \times f'(z_i^{(l)}) = \left( \sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) \times f'(z_i^{(l)})
\]
Backpropagation: $\delta_i^{(l)}$

- Start computing $\delta$’s for the output layer:

\[
\delta_i^{(n_l)} = \frac{\partial J}{\partial a_i^{(n_l)}} \times \frac{\partial a_i^{(n_l)}}{\partial z_i^{(n_l)}} = \frac{\partial J}{\partial a_i^{(n_l)}} \times f'(z_i^{(n_l)})
\]

\[
J = \frac{1}{2} \|a^{(n_l)} - y\|^2 \Rightarrow \frac{\partial J}{\partial a_i^{(n_l)}} = (a_i^{(n_l)} - y_i)
\]

\[
\delta_i^{(n_l)} = (a_i^{(n_l)} - y_i) \times f'(z_i^{(n_l)})
\]
Backpropagation Algorithm

1. Feedforward pass on \( x \) to compute activations \( a_i^{(l)} \)

2. For each output unit \( i \) compute:
   \[
   \delta_i^{(n_l)} = (a_i^{(n_l)} - y_i) \times f'(z_i^{(n_l)})
   \]

3. For \( l = n_{l-1}, n_{l-2}, n_{l-3}, \ldots, 2 \) compute:
   \[
   \delta_i^{(l)} = \left( \sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) \times f'(z_i^{(l)})
   \]

4. Compute the partial derivatives of the cost \( J(W, b, x, y) \)
   \[
   \frac{\partial J}{\partial W_{ij}^{(l)}} = a_j^{(l)} \delta_i^{(l+1)} \quad \frac{\partial J}{\partial b_i^{(l)}} = \delta_i^{(l+1)}
   \]
Backpropagation Algorithm: Vectorization for 1 Example

1. Feedforward pass on $x$ to compute activations $a_i^{(l)}$
2. For last layer compute:
   \[ \delta^{(n_l)} = (a^{(n_l)} - y) \cdot f'(z^{(n_l)}) \]
3. For $l = n_l-1, n_l-2, n_l-3, \ldots, 2$ compute:
   \[ \delta^{(l)} = \left( (W^{(l)})^T \delta^{(l+1)} \right) \cdot f'(z^{(l)}) \]
4. Compute the partial derivatives of the cost $J(W, b, x, y)$
   \[ \nabla_{W^{(l)}} J = \delta^{(l+1)} (a^{(l)})^T \quad \nabla_{b^{(l)}} J = \delta^{(l+1)} \]
Backpropagation Algorithm: Vectorization for Dataset of m Examples

1. Feedforward pass on X to compute activations $a_i^{(l)}$

2. For last layer compute:
   \[ \delta^{(n_l)} = (a^{(n_l)} - y) \cdot f'(z^{(n_l)}) \]

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$ compute:
   \[ \delta^{(l)} = \left( (W^{(l)})^T \delta^{(l+1)} \right) \cdot f'(z^{(l)}) \]

4. Compute the partial derivatives of the cost $J(W, b, x, y)$
   \[ \nabla_{w^{(l)}} J = \delta^{(l+1)} (a^{(l)})^T / m \ 
   \nabla_{b^{(l)}} J = \delta^{(l+1)}.\text{col\_avg()} \]
Backpropagation: Softmax Regression

• Consider layer $n_l$ to be the input to the softmax layer i.e. softmax output layer is $n_l+1$.

• Softmax weights stored in matrix $W^{(n_l)}$.

• K classes $\Rightarrow \ W^{(n_l)} = \begin{bmatrix} -w_1^T & - \\ -w_2^T & - \\ \vdots & \vdots \\ -w_K^T & - \end{bmatrix}$
Backpropagation: Softmax Regression

- Softmax output is $a^{(n_l+1)} = \text{softmax}(z^{(n_l+1)})$

$$x \rightarrow a^{(1)} \leftarrow \text{Softmax input}$$

$$\text{Softmax weights } W^{(n_l)}$$

$$a^{(n_l)} \rightarrow z^{(n_l+1)} \leftarrow \text{Softmax logits}$$

$$\vdots$$

$$a^{(n_l)} \rightarrow z^{(n_l+1)}$$

$$J(a^{(n_l+1)}, y)$$

$$x$$

$$a^{(1)}$$

$$\text{Cross-entropy}$$
Backpropagation Algorithm: Softmax (1)

1. Feedforward pass on \( x \) to compute activations \( a^{(l)} \) for layers \( l = 1, 2, \ldots, n_l \).

2. Compute softmax outputs \( a^{(n_l+1)} \) and objective \( J(a^{(n_l+1)}, y) \).

3. Let \( y = [\delta_1(y), \delta_2(y), \ldots, \delta_K(y)]^T \) be the one-hot vector representation for label \( y \).

4. Compute gradient with respect to softmax weights:

\[
\frac{\partial J}{\partial W^{(n_l)}} = (a^{(n_l+1)} - y)a^{(n_l)T}
\]
Backpropagation Algorithm: Softmax (2)

5. Compute gradient with respect to softmax inputs:

\[ \delta^{(n_l)} = (W^{(n_l)})^T (a^{(n_{l+1})} - y) \odot f'(z^{(n_l)}) \]

\[ \frac{\partial J}{\partial a^{(n_l)}} \]

6. For \( l = n_l-1, n_l-2, n_l-3, \ldots, 2 \) compute:

\[ \delta^{(l)} = \left( (W^{(l)})^T \delta^{(l+1)} \right) \cdot f'(z^{(l)}) \]

7. Compute the partial derivatives of the cost \( J(W, b, x, y) \)

\[ \nabla_{W^{(l)}} J = \delta^{(l+1)} (a^{(l)})^T \]

\[ \nabla_{b^{(l)}} J = \delta^{(l+1)} \]
Backpropagation Algorithm: Softmax for 1 Example

1. For softmax layer, compute:
   \[ \delta^{(n_l+1)} = (\mathbf{a}^{(n_l+1)} - \mathbf{y}) \]

2. For \( l = n_l, n_l-2, n_l-3, \ldots, 2 \) compute:
   \[ \delta^{(l)} = \left( (W^{(l)})^T \delta^{(l+1)} \right) \cdot f'(z^{(l)}) \]

3. Compute the partial derivatives of the cost \( J(W, b, x, y) \)
   \[ \nabla_{W^{(l)}} J = \delta^{(l+1)} (a^{(l)})^T \quad \nabla_{b^{(l)}} J = \delta^{(l+1)} \]
Backpropagation Algorithm: Softmax for Dataset of $m$ Examples

1. For softmax layer, compute:
$$\delta^{(n_l+1)} = (a^{(n_l+1)} - y)$$

2. For $l = n_l, n_l-1, n_l-2, \ldots, 2$ compute:
$$\delta^{(l)} = \left( \left( W^{(l)} \right)^T \delta^{(l+1)} \right) \cdot f'(z^{(l)})$$

3. Compute the partial derivatives of the cost $J(W, b, x, y)$
$$\nabla_{W^{(l)}} J = \delta^{(l+1)} \left( a^{(l)} \right)^T / m \quad \nabla_{b^{(l)}} J = \delta^{(l+1)} . \text{col\_avg}()$$
Backpropagation: Logistic Regression