Neural Networks: Design

Shan-Hung Wu
shwu@cs.nthu.edu.tw

Department of Computer Science,
National Tsing Hua University, Taiwan

Machine Learning

Outline

1. **The Basics**
   - Example: Learning the XOR
2. **Training**
   - Back Propagation
3. **Neuron Design**
   - Cost Function & Output Neurons
   - Hidden Neurons
4. **Architecture Design**
   - Architecture Tuning

Outline

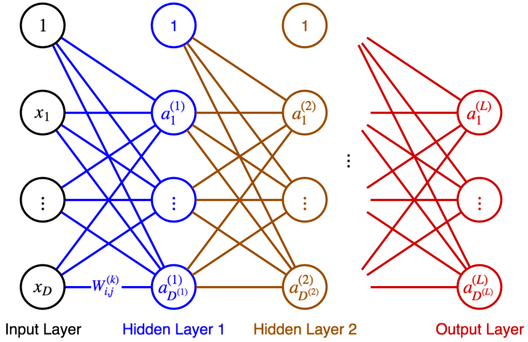
1. **The Basics**
   - Example: Learning the XOR
2. **Training**
   - Back Propagation
3. **Neuron Design**
   - Cost Function & Output Neurons
   - Hidden Neurons
4. **Architecture Design**
   - Architecture Tuning

Model: a Composite Function I

- A ***feedforward neural networks***, or ***multilayer perceptron***, defines a function composition

$$\hat{\boldsymbol{y}} = \boldsymbol{f}^{(L)}(\cdots \boldsymbol{f}^{(2)}(\boldsymbol{f}^{(1)}(\boldsymbol{x};\boldsymbol{\theta}^{(1)});\boldsymbol{\theta}^{(2)});\boldsymbol{\theta}^{(L)})$$

that approximates the target function f^*

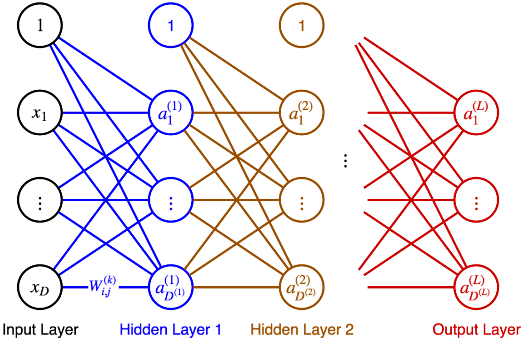


Model: a Composite Function I

- A ***feedforward neural networks***, or ***multilayer perceptron***, defines a function composition

$$\hat{\boldsymbol{y}} = \boldsymbol{f}^{(L)}(\cdots \boldsymbol{f}^{(2)}(\boldsymbol{f}^{(1)}(\boldsymbol{x};\boldsymbol{\theta}^{(1)});\boldsymbol{\theta}^{(2)});\boldsymbol{\theta}^{(L)})$$

  that approximates the target function f^*
- Parameters $\theta^{(1)}, \cdots, \theta^{(L)}$ learned from training set $\mathbb{X}$

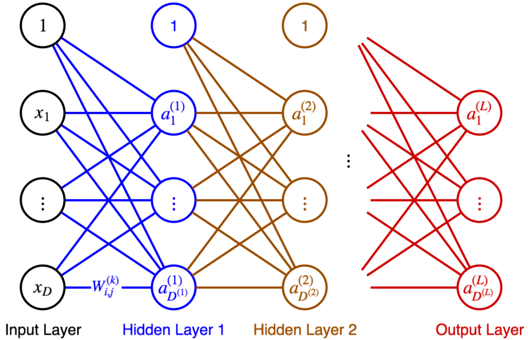


Model: a Composite Function I

- A ***feedforward neural networks***, or ***multilayer perceptron***, defines a function composition

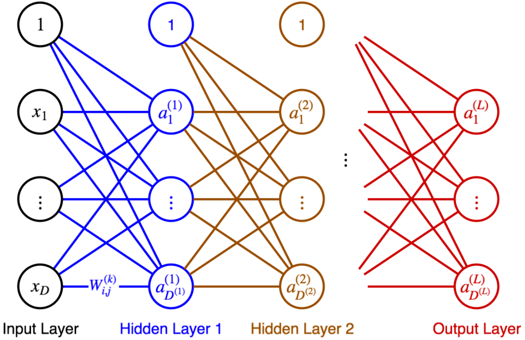
$$\hat{\boldsymbol{y}} = \boldsymbol{f}^{(L)}(\cdots \boldsymbol{f}^{(2)}(\boldsymbol{f}^{(1)}(\boldsymbol{x};\boldsymbol{\theta}^{(1)});\boldsymbol{\theta}^{(2)});\boldsymbol{\theta}^{(L)})$$

  that approximates the target function f^*
- Parameters $\theta^{(1)},\cdots,\theta^{(L)}$ learned from training set $\mathbb{X}$
- "Feedforward" because information flows from input to output



Model: a Composite Function II

- At each layer k, the function $f^{(k)}(\cdot\,;\boldsymbol{W}^{(k)},\boldsymbol{b}^{(k)})$ is ***nonlinear*** and outputs value $\boldsymbol{a}^{(k)} \in \mathbb{R}^{D^{(k)}}$, where
$$\boldsymbol{a}^{(k)} = \mathrm{act}^{(k)}(\boldsymbol{W}^{(k)\top}\boldsymbol{a}^{(k-1)} + \boldsymbol{b}^{(k)})$$
    - $\mathrm{act}^{(\mathrm{i})}(\cdot):\mathbb{R}\to\mathbb{R}$ is an ***activation function*** applied elementwisely

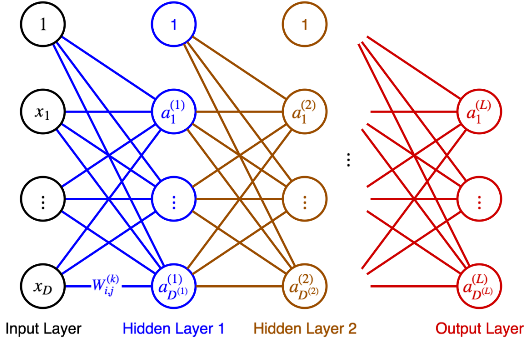


Model: a Composite Function II

- At each layer k, the function $f^{(k)}(\cdot\,;\boldsymbol{W}^{(k)},\boldsymbol{b}^{(k)})$ is ***nonlinear*** and outputs value $\boldsymbol{a}^{(k)} \in \mathbb{R}^{D^{(k)}}$, where

$$\boldsymbol{a}^{(k)} = \text{act}^{(k)}(\boldsymbol{W}^{(k)\top}\boldsymbol{a}^{(k-1)} + \boldsymbol{b}^{(k)})$$

  - $\text{act}^{(i)}(\cdot) : \mathbb{R} \to \mathbb{R}$ is an ***activation function*** applied elementwisely
- Shorthand: $\boldsymbol{a}^{(k)} = \text{act}^{(k)}(\boldsymbol{W}^{(k)\top}\boldsymbol{a}^{(k-1)})$
  - $\boldsymbol{a}^{(k-1)} \in \mathbb{R}^{D^{(k-1)}+1}$, $a_0^{(k-1)} = 1$, and $\boldsymbol{W}^{(k)} \in \mathbb{R}^{(D^{(k-1)}+1)\times D^{(k)}}$

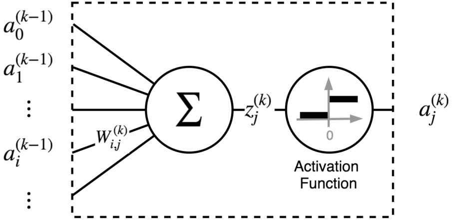


Neurons I

- Each $f_j^{(k)} = \mathrm{act}^{(k)}(\boldsymbol{W}_{:,j}^{(k)\top}\boldsymbol{a}^{(k-1)}) = \mathrm{act}^{(k)}(z_j^{(k)})$ is a ***unit*** (or ***neuron***)

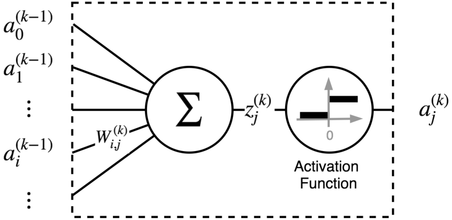
Neurons I

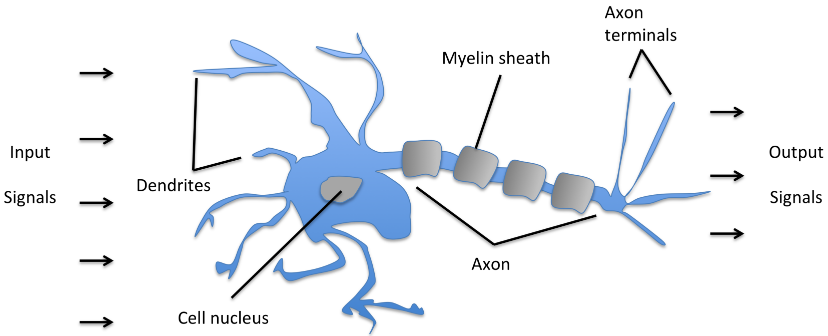
- Each $f_j^{(k)} = \mathrm{act}^{(k)}(\boldsymbol{W}_{:,j}^{(k)\top}\boldsymbol{a}^{(k-1)}) = \mathrm{act}^{(k)}(z_j^{(k)})$ is a ***unit*** (or ***neuron***)
- E.g., the perceptron



Neurons I

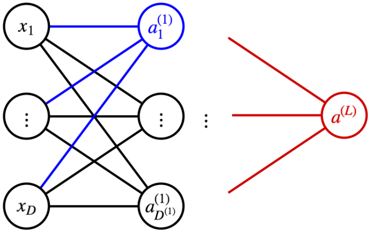
- Each $f_j^{(k)} = \mathrm{act}^{(k)}(\boldsymbol{W}_{:,j}^{(k)\top}\boldsymbol{a}^{(k-1)}) = \mathrm{act}^{(k)}(z_j^{(k)})$ is a ***unit*** (or ***neuron***)
- E.g., the perceptron
- Loosely guided by neuroscience





Neurons II

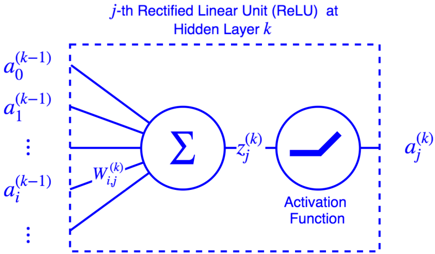
- Modern NN design is mainly guided by mathematical and engineering disciplines. Consider a binary classifier where $y \in \{0, 1\}$:

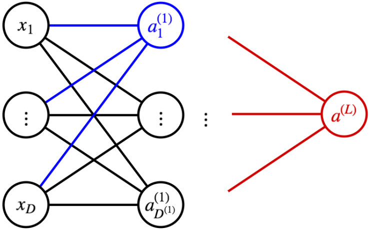


Neurons II

- Modern NN design is mainly guided by mathematical and engineering disciplines. Consider a binary classifier where $y \in \{0, 1\}$:

- Hidden units: $\boldsymbol{a}^{(k)} = \max(0, \boldsymbol{z}^{(k)})$



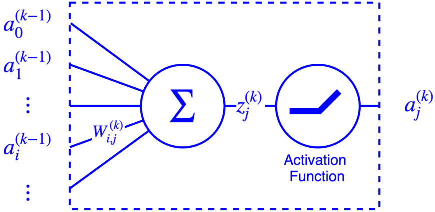


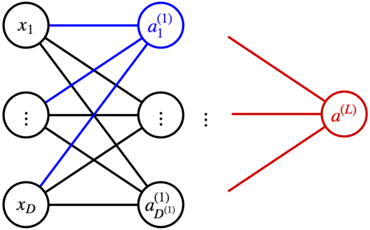
Neurons II

- Modern NN design is mainly guided by mathematical and engineering disciplines. Consider a binary classifier where $y \in \{0,1\}$:

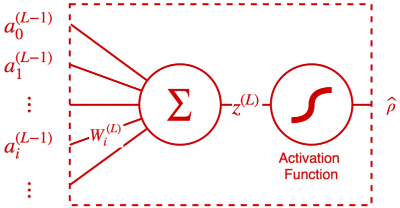
- Hidden units: $\boldsymbol{a}^{(k)} = \max(0, \boldsymbol{z}^{(k)})$
- Output unit: $a^{(L)} = \hat{\rho} = \sigma(z^{(L)})$, assuming $\Pr(\mathrm{y}=1\,|\,\boldsymbol{x}) \sim \mathrm{Bernoulli}(\rho)$
  - Prediction:
    $\hat{y} = 1(\hat{\rho}; \hat{\rho} > 0.5) = 1(z^{(L)}; z^{(L)} > 0)$

j-th Rectified Linear Unit (ReLU) at Hidden Layer k





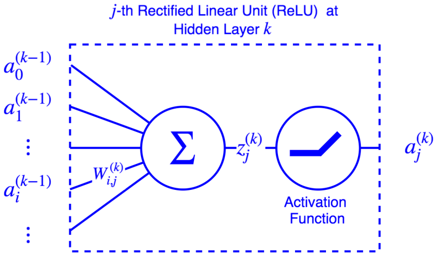
Sigmoid Unit for Bernoulli $P(y \mid x; \rho)$

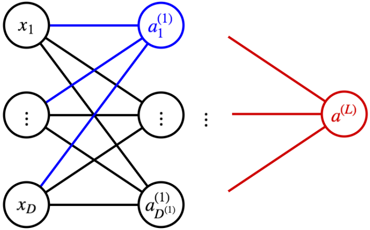


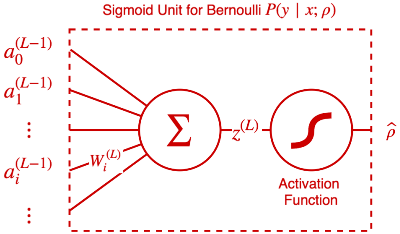
Neurons II

- Modern NN design is mainly guided by mathematical and engineering disciplines. Consider a binary classifier where $y \in \{0,1\}$:

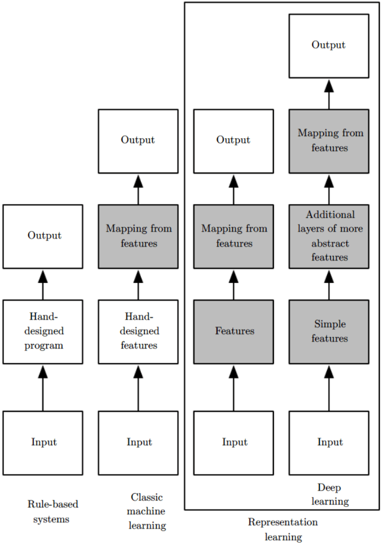
- Hidden units: $\boldsymbol{a}^{(k)} = \max(0, \boldsymbol{z}^{(k)})$
- Output unit: $\boldsymbol{a}^{(L)} = \hat{\rho} = \sigma(\boldsymbol{z}^{(L)})$, assuming $\Pr(\mathrm{y} = 1 \,|\, \boldsymbol{x}) \sim \mathrm{Bernoulli}(\rho)$
  - Prediction: $\hat{y} = 1(\hat{\rho}; \hat{\rho} > 0.5) = 1(z^{(L)}; z^{(L)} > 0)$
  - A logistic regressor with input $\boldsymbol{a}^{(L-1)}$





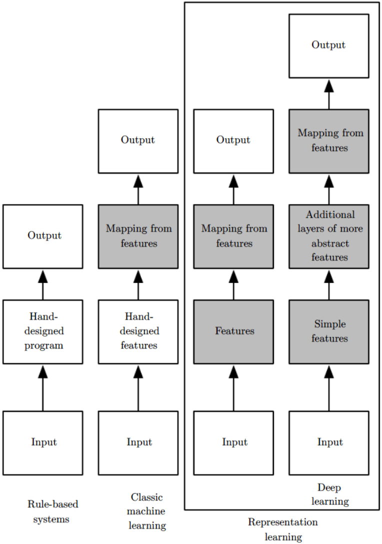


Representation Learning



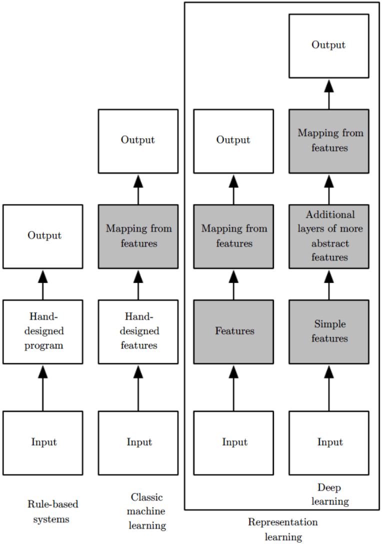
- The outputs $\boldsymbol{a}^{(1)}, \boldsymbol{a}^{(2)}, \cdots, \boldsymbol{a}^{(L-1)}$ of hidden layers $\boldsymbol{f}^{(1)}, \boldsymbol{f}^{(2)}, \cdots, \boldsymbol{f}^{(L-1)}$ are ***distributed representation*** of $\boldsymbol{x}$
  - Nonlinear to input space since $\boldsymbol{f}^{(k)}$'s are nonlinear

Representation Learning



- The outputs $\boldsymbol{a}^{(1)}, \boldsymbol{a}^{(2)}, \cdots, \boldsymbol{a}^{(L-1)}$ of hidden layers $\boldsymbol{f}^{(1)}, \boldsymbol{f}^{(2)}, \cdots, \boldsymbol{f}^{(L-1)}$ are ***distributed representation*** of $\boldsymbol{x}$
  - Nonlinear to input space since $\boldsymbol{f}^{(k)}$'s are nonlinear
  - Usually more abstract at a deeper layer

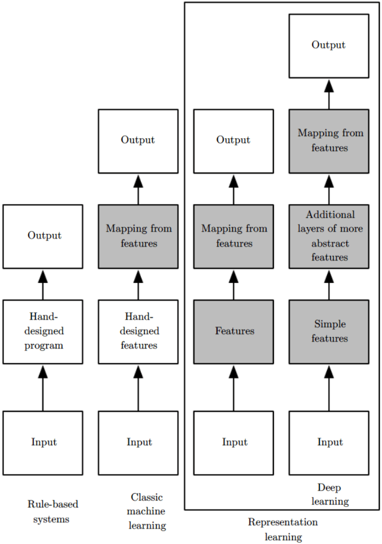
Representation Learning



- The outputs $\boldsymbol{a}^{(1)}, \boldsymbol{a}^{(2)}, \cdots, \boldsymbol{a}^{(L-1)}$ of hidden layers $\boldsymbol{f}^{(1)}, \boldsymbol{f}^{(2)}, \cdots, \boldsymbol{f}^{(L-1)}$ are ***distributed representation*** of $\boldsymbol{x}$
  - Nonlinear to input space since $\boldsymbol{f}^{(k)}$'s are nonlinear
  - Usually more abstract at a deeper layer
- $\boldsymbol{f}^{(L)}$ is the actual prediction function
  - Like in non-linear SVM/polynomial regression, a simple linear function suffices:

$$z^{(L)} = \boldsymbol{W}^{(L)} \boldsymbol{a}^{(L-1)}$$

Representation Learning



- The outputs $\boldsymbol{a}^{(1)}, \boldsymbol{a}^{(2)}, \cdots, \boldsymbol{a}^{(L-1)}$ of hidden layers $\boldsymbol{f}^{(1)}, \boldsymbol{f}^{(2)}, \cdots, \boldsymbol{f}^{(L-1)}$ are ***distributed representation*** of $\boldsymbol{x}$
  - Nonlinear to input space since $\boldsymbol{f}^{(k)}$'s are nonlinear
  - Usually more abstract at a deeper layer
- $\boldsymbol{f}^{(L)}$ is the actual prediction function
  - Like in non-linear SVM/polynomial regression, a simple linear function suffices:

$$\boldsymbol{z}^{(L)} = \boldsymbol{W}^{(L)}\boldsymbol{a}^{(L-1)}$$

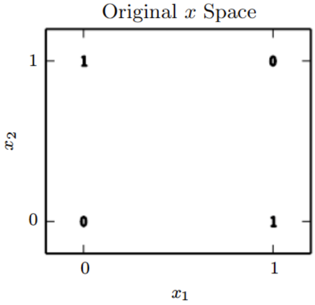
  - $\mathrm{act}^{(L)}(\cdot)$ just "normalizes" $\boldsymbol{z}^{(L)}$ to give $\hat{\rho} \in (0,1)$

Outline

1. **The Basics**
   - Example: Learning the XOR
2. **Training**
   - Back Propagation
3. **Neuron Design**
   - Cost Function & Output Neurons
   - Hidden Neurons
4. **Architecture Design**
   - Architecture Tuning

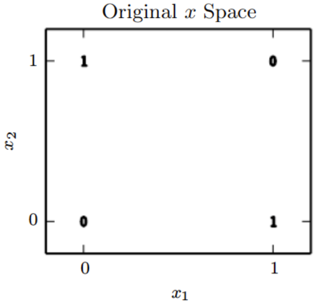
Learning the XOR I

- Why ReLUs learn nonlinear (and better) representation?



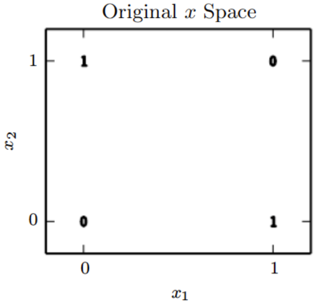
Learning the XOR I

- Why ReLUs learn nonlinear (and better) representation?
- Let's learn XOR (f^*) in a binary classification task
  - $\boldsymbol{x} \in \mathbb{R}^2$ and $y \in \{0,1\}$



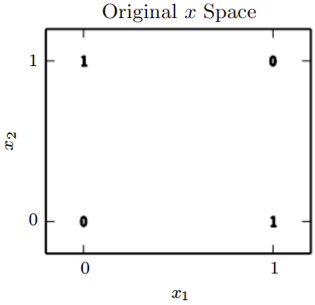
Learning the XOR I

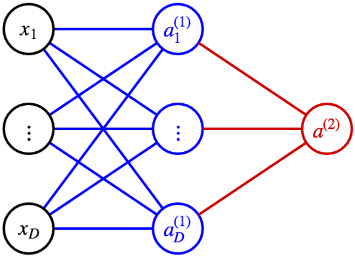
- Why ReLUs learn nonlinear (and better) representation?
- Let's learn XOR (f^*) in a binary classification task
  - $\boldsymbol{x} \in \mathbb{R}^2$ and $y \in \{0,1\}$
  - Nonlinear, so cannot be learned by linear models



Learning the XOR I

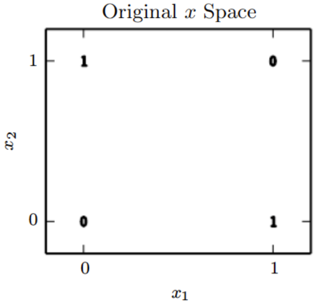
- Why ReLUs learn nonlinear (and better) representation?
- Let's learn XOR (f^*) in a binary classification task
  - $\boldsymbol{x} \in \mathbb{R}^2$ and $y \in \{0,1\}$
  - Nonlinear, so cannot be learned by linear models
- Consider an NN with 1 hidden layer:
  - $\boldsymbol{a}^{(1)} = \max(0, \boldsymbol{W}^{(1)\top}\boldsymbol{x})$
  - $a^{(2)} = \hat{\rho} = \sigma(\boldsymbol{w}^{(2)\top}\boldsymbol{a}^{(1)})$
  - Prediction: $1(\hat{\rho}; \hat{\rho} > 0.5)$

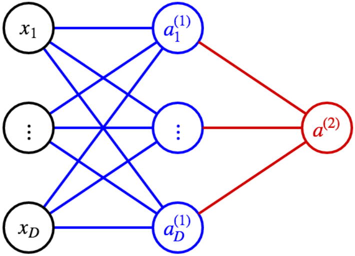




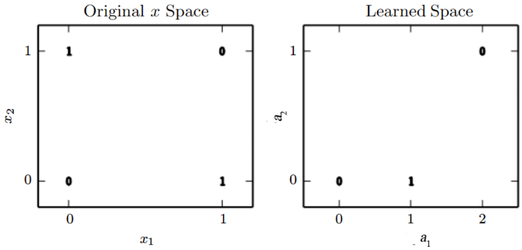
Learning the XOR I

- Why ReLUs learn nonlinear (and better) representation?
- Let's learn XOR (f^*) in a binary classification task
  - $\boldsymbol{x} \in \mathbb{R}^2$ and $y \in \{0,1\}$
  - Nonlinear, so cannot be learned by linear models
- Consider an NN with 1 hidden layer:
  - $\boldsymbol{a}^{(1)} = \max(0, \boldsymbol{W}^{(1)\top}\boldsymbol{x})$
  - $a^{(2)} = \hat{\rho} = \sigma(\boldsymbol{w}^{(2)\top}\boldsymbol{a}^{(1)})$
  - Prediction: $1(\hat{\rho}; \hat{\rho} > 0.5)$
- Learns XOR by "merging" data points first



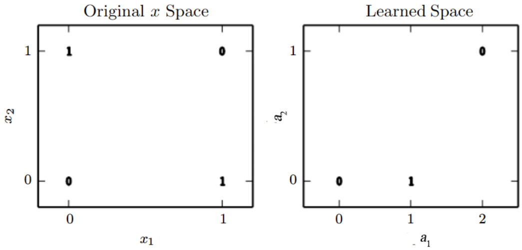


Learning the XOR II



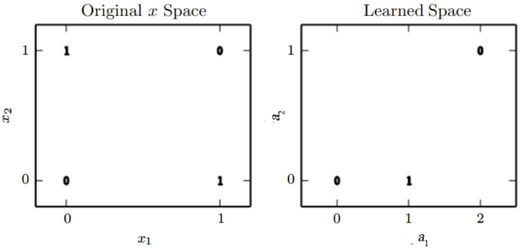
- $\boldsymbol{X} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \in \mathbb{R}^{N\times(1+D)}$, $\boldsymbol{W}^{(1)} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \\ 0 & -1 \end{bmatrix}$, $\boldsymbol{w}^{(2)} = \begin{bmatrix} -1 \\ 2 \\ -4 \end{bmatrix}$
- $\hat{\boldsymbol{y}} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix} = 1(\sigma([\ 1 \quad \max(0, \boldsymbol{X}\boldsymbol{W}^{(1)})\ ]\boldsymbol{w}^{(2)}) > 0.5)$

Latent Representation $A^{(1)}$



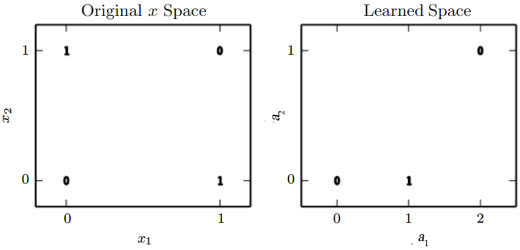
- $XW^{(1)} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 1 & 1 \end{bmatrix} \begin{bmatrix} 0 & -1 \\ 1 & 1 \\ 1 & 1 \end{bmatrix} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$
- $A^{(1)} = [\ 1 \quad \max(0, XW^{(1)})\ ] = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 2 & 1 \end{bmatrix}$

Output Distribution $\boldsymbol{a}^{(2)}$



- $\boldsymbol{a}^{(2)} = \sigma(\boldsymbol{A}^{(1)}\boldsymbol{w}^{(2)}) = \sigma\left(\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 2 & 1 \end{bmatrix}\begin{bmatrix} -1 \\ 2 \\ -4 \end{bmatrix}\right) = \sigma\left(\begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}\right)$
- $\hat{\boldsymbol{y}} = 1(\boldsymbol{a}^{(2)} > 0.5) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$

Output Distribution $\boldsymbol{a}^{(2)}$



- $\boldsymbol{a}^{(2)} = \sigma(\boldsymbol{A}^{(1)}\boldsymbol{w}^{(2)}) = \sigma\left(\begin{bmatrix} 1 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 1 & 0 \\ 1 & 2 & 1 \end{bmatrix}\begin{bmatrix} -1 \\ 2 \\ -4 \end{bmatrix}\right) = \sigma\left(\begin{bmatrix} -1 \\ 1 \\ 1 \\ -1 \end{bmatrix}\right)$
- $\hat{\boldsymbol{y}} = 1(\boldsymbol{a}^{(2)} > 0.5) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$
- But how to train $\boldsymbol{W}^{(1)}$ and $\boldsymbol{w}^{(2)}$ from examples?

Outline

1. The Basics
   - Example: Learning the XOR
2. Training
   - Back Propagation
3. Neuron Design
   - Cost Function & Output Neurons
   - Hidden Neurons
4. Architecture Design
   - Architecture Tuning

Training an NN

- Given examples: $\mathbb{X} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{N}$
- How to learn parameters $\Theta = \{\boldsymbol{W}^{(1)}, \cdots, \boldsymbol{W}^{(L)}\}$?

Training an NN

- Given examples: $\mathbb{X} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{N}$
- How to learn parameters $\Theta = \{\boldsymbol{W}^{(1)}, \cdots, \boldsymbol{W}^{(L)}\}$?
- Most NNs are trained using the ***maximum likelihood*** by default (assuming i.i.d examples):

$$\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X} \,|\, \Theta)$$

Training an NN

- Given examples: $\mathbb{X} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{N}$
- How to learn parameters $\Theta = \{\boldsymbol{W}^{(1)}, \cdots, \boldsymbol{W}^{(L)}\}$?
- Most NNs are trained using the ***maximum likelihood*** by default (assuming i.i.d examples):

$$
\begin{aligned}
&\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X} \,|\, \Theta) \\
&\quad = \arg\min_{\Theta} -\log \mathrm{P}(\mathbb{X} \,|\, \Theta)
\end{aligned}
$$

Training an NN

- Given examples: $\mathbb{X} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{N}$
- How to learn parameters $\Theta = \{\boldsymbol{W}^{(1)}, \cdots, \boldsymbol{W}^{(L)}\}$?
- Most NNs are trained using the ***maximum likelihood*** by default (assuming i.i.d examples):

$$
\begin{aligned}
&\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X} \,|\, \Theta) \\
&\quad = \arg\min_{\Theta} -\log \mathrm{P}(\mathbb{X} \,|\, \Theta) \\
&\quad = \arg\min_{\Theta} \textstyle\sum_i -\log \mathrm{P}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \,|\, \Theta)
\end{aligned}
$$

Training an NN

- Given examples: $\mathbb{X} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{N}$
- How to learn parameters $\Theta = \{\boldsymbol{W}^{(1)}, \cdots, \boldsymbol{W}^{(L)}\}$?
- Most NNs are trained using the ***maximum likelihood*** by default (assuming i.i.d examples):

$$
\begin{aligned}
&\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X} \,|\, \Theta) \\
&\quad = \arg\min_{\Theta} - \log \mathrm{P}(\mathbb{X} \,|\, \Theta) \\
&\quad = \arg\min_{\Theta} \textstyle\sum_i - \log \mathrm{P}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \,|\, \Theta) \\
&\quad = \arg\min_{\Theta} \textstyle\sum_i [- \log \mathrm{P}(\boldsymbol{y}^{(i)} \,|\, \boldsymbol{x}^{(i)}, \Theta) - \log \mathrm{P}(\boldsymbol{x}^{(i)} \,|\, \Theta)]
\end{aligned}
$$

Training an NN

- Given examples: $\mathbb{X} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{N}$
- How to learn parameters $\Theta = \{\boldsymbol{W}^{(1)}, \cdots, \boldsymbol{W}^{(L)}\}$?
- Most NNs are trained using the ***maximum likelihood*** by default (assuming i.i.d examples):

$$
\begin{aligned}
&\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X} \,|\, \Theta) \\
&\quad = \arg\min_{\Theta} -\log \mathrm{P}(\mathbb{X} \,|\, \Theta) \\
&\quad = \arg\min_{\Theta} \textstyle\sum_i -\log \mathrm{P}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \,|\, \Theta) \\
&\quad = \arg\min_{\Theta} \textstyle\sum_i [-\log \mathrm{P}(\boldsymbol{y}^{(i)} \,|\, \boldsymbol{x}^{(i)}, \Theta) - \log \mathrm{P}(\boldsymbol{x}^{(i)} \,|\, \Theta)] \\
&\quad = \arg\min_{\Theta} \textstyle\sum_i -\log \mathrm{P}(\boldsymbol{y}^{(i)} \,|\, \boldsymbol{x}^{(i)}, \Theta) \\
&\quad = \arg\min_{\Theta} \textstyle\sum_i C^{(i)}(\Theta)
\end{aligned}
$$

Training an NN

- Given examples: $\mathbb{X} = \{(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)})\}_{i=1}^{N}$
- How to learn parameters $\Theta = \{\boldsymbol{W}^{(1)}, \cdots, \boldsymbol{W}^{(L)}\}$?
- Most NNs are trained using the ***maximum likelihood*** by default (assuming i.i.d examples):

$$\begin{aligned}
&\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X} \mid \Theta) \\
&\quad = \arg\min_{\Theta} -\log \mathrm{P}(\mathbb{X} \mid \Theta) \\
&\quad = \arg\min_{\Theta} \sum_i -\log \mathrm{P}(\boldsymbol{x}^{(i)}, \boldsymbol{y}^{(i)} \mid \Theta) \\
&\quad = \arg\min_{\Theta} \sum_i [-\log \mathrm{P}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}, \Theta) - \log \mathrm{P}(\boldsymbol{x}^{(i)} \mid \Theta)] \\
&\quad = \arg\min_{\Theta} \sum_i -\log \mathrm{P}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}, \Theta) \\
&\quad = \arg\min_{\Theta} \sum_i C^{(i)}(\Theta)
\end{aligned}$$

- The minimizer $\hat{\Theta}$ is an unbiased estimator of "true" Θ^*
  - Good for large N

Example: Binary Classification

- $\Pr(\mathrm{y} = 1 \,|\, \boldsymbol{x}) \sim \mathrm{Bernoulli}(\rho)$, where $\boldsymbol{x} \in \mathbb{R}^D$ and $y \in \{0, 1\}$
- $a^{(L)} = \hat{\rho} = \sigma(z^{(L)})$ the predicted distribution

Example: Binary Classification

- $\Pr(\mathrm{y}=1\,|\,\boldsymbol{x}) \sim \mathrm{Bernoulli}(\rho)$, where $\boldsymbol{x} \in \mathbb{R}^D$ and $y \in \{0,1\}$
- $a^{(L)} = \hat{\rho} = \sigma(z^{(L)})$ the predicted distribution

- The cost function $C^{(i)}(\Theta)$ can be written as:

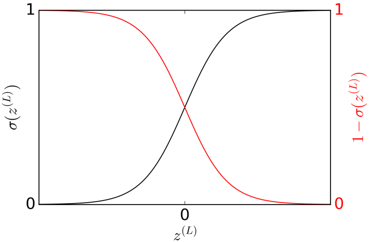
$$
\begin{aligned}
&C^{(i)}(\Theta) \\
&\quad = -\log \mathrm{P}(\boldsymbol{y}^{(i)}\,|\,\boldsymbol{x}^{(i)};\Theta) \\
&\quad = -\log[(a^{(L)})^{y^{(i)}}(1-a^{(L)})^{1-y^{(i)}}] \\
&\quad = -\log[\sigma(z^{(L)})^{y^{(i)}}(1-\sigma(z^{(L)}))^{1-y^{(i)}}]
\end{aligned}
$$

Example: Binary Classification

- $\Pr(y = 1 \,|\, \boldsymbol{x}) \sim \text{Bernoulli}(\rho)$, where $\boldsymbol{x} \in \mathbb{R}^D$ and $y \in \{0, 1\}$
- $a^{(L)} = \hat{\rho} = \sigma(z^{(L)})$ the predicted distribution

- The cost function $C^{(i)}(\Theta)$ can be written as:

$$
\begin{aligned}
C^{(i)}(\Theta) & \\
&= -\log \mathrm{P}(\boldsymbol{y}^{(i)} \,|\, \boldsymbol{x}^{(i)}; \Theta) \\
&= -\log[(a^{(L)})^{y^{(i)}} (1 - a^{(L)})^{1-y^{(i)}}] \\
&= -\log[\sigma(z^{(L)})^{y^{(i)}} (1 - \sigma(z^{(L)}))^{1-y^{(i)}}] \\
&= -\log[\sigma((2y^{(i)} - 1) z^{(L)})]
\end{aligned}
$$

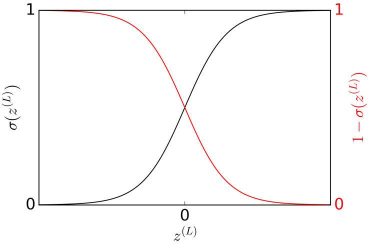


Example: Binary Classification

- $\Pr(\mathrm{y}=1\,|\,\boldsymbol{x}) \sim \mathrm{Bernoulli}(\rho)$, where $\boldsymbol{x} \in \mathbb{R}^D$ and $y \in \{0,1\}$
- $a^{(L)} = \hat{\rho} = \sigma(z^{(L)})$ the predicted distribution

- The cost function $C^{(i)}(\Theta)$ can be written as:

$$
\begin{aligned}
& C^{(i)}(\Theta) \\
& \quad = -\log \mathrm{P}(\boldsymbol{y}^{(i)}\,|\,\boldsymbol{x}^{(i)};\Theta) \\
& \quad = -\log[(a^{(L)})^{y^{(i)}}(1-a^{(L)})^{1-y^{(i)}}] \\
& \quad = -\log[\sigma(z^{(L)})^{y^{(i)}}(1-\sigma(z^{(L)}))^{1-y^{(i)}}] \\
& \quad = -\log[\sigma((2y^{(i)}-1)z^{(L)})] \\
& \quad = \zeta((1-2y^{(i)})z^{(L)})
\end{aligned}
$$



  - $\zeta(\cdot)$ is the softplus function

Optimization Algorithm

- Most NNs use ***SGD*** to solve the problem $\arg\min_{\Theta} \sum_i C^{(i)}(\Theta)$

(Mini-Batched) Stochastic Gradient Descent (SGD)

Initialize $\Theta^{(0)}$ randomly;
Repeat until convergence {
  Randomly partition the training set $\mathbb{X}$ into ***minibatches*** of size M;
  $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \nabla_{\Theta} \sum_{i=1}^{M} C^{(i)}(\Theta^{(t)})$;
}

Optimization Algorithm

- Most NNs use ***SGD*** to solve the problem $\arg\min_{\Theta} \sum_i C^{(i)}(\Theta)$
  - Fast convergence in time [1]

(Mini-Batched) Stochastic Gradient Descent (SGD)

Initialize $\Theta^{(0)}$ randomly;
Repeat until convergence {
  Randomly partition the training set $\mathbb{X}$ into ***minibatches*** of size M;
  $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \nabla_{\Theta} \sum_{i=1}^{M} C^{(i)}(\Theta^{(t)})$;
}

Optimization Algorithm

- Most NNs use ***SGD*** to solve the problem $\arg\min_{\Theta} \sum_i C^{(i)}(\Theta)$
  - Fast convergence in time [1]
  - Supports (GPU-based) parallelism

(Mini-Batched) Stochastic Gradient Descent (SGD)

Initialize $\Theta^{(0)}$ randomly;
Repeat until convergence {
  Randomly partition the training set $\mathbb{X}$ into ***minibatches*** of size M;
  $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \nabla_{\Theta} \sum_{i=1}^{M} C^{(i)}(\Theta^{(t)})$;
}

Optimization Algorithm

- Most NNs use ***SGD*** to solve the problem $\arg\min_{\Theta} \sum_i C^{(i)}(\Theta)$
  - Fast convergence in time [1]
  - Supports (GPU-based) parallelism
  - Supports online learning

(Mini-Batched) Stochastic Gradient Descent (SGD)

Initialize $\Theta^{(0)}$ randomly;
Repeat until convergence {
  Randomly partition the training set $\mathbb{X}$ into ***minibatches*** of size M;
  $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \nabla_{\Theta} \sum_{i=1}^{M} C^{(i)}(\Theta^{(t)})$;
}

Optimization Algorithm

- Most NNs use ***SGD*** to solve the problem $\arg\min_{\Theta}\sum_i C^{(i)}(\Theta)$
  - Fast convergence in time [1]
  - Supports (GPU-based) parallelism
  - Supports online learning
  - Easy to implement

(Mini-Batched) Stochastic Gradient Descent (SGD)

Initialize $\Theta^{(0)}$ randomly;
Repeat until convergence {
  Randomly partition the training set $\mathbb{X}$ into ***minibatches*** of size M;
  $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \nabla_{\Theta} \sum_{i=1}^{M} C^{(i)}(\Theta^{(t)})$;
}

Optimization Algorithm

- Most NNs use ***SGD*** to solve the problem $\arg\min_{\Theta} \sum_i C^{(i)}(\Theta)$
  - Fast convergence in time [1]
  - Supports (GPU-based) parallelism
  - Supports online learning
  - Easy to implement

(Mini-Batched) Stochastic Gradient Descent (SGD)

Initialize $\Theta^{(0)}$ randomly;
Repeat until convergence {
  Randomly partition the training set $\mathbb{X}$ into ***minibatches*** of size M;
  $\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \nabla_{\Theta} \sum_{i=1}^{M} C^{(i)}(\Theta^{(t)})$;
}

- How to compute $\nabla_{\Theta} \sum_i C^{(i)}(\Theta^{(t)})$ efficiently?
  - There could be a huge number of $W_{i,j}^{(k)}$'s in Θ

Outline

1 The Basics
- Example: Learning the XOR

2 Training
- Back Propagation

3 Neuron Design
- Cost Function & Output Neurons
- Hidden Neurons

4 Architecture Design
- Architecture Tuning

Back Propagation

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \nabla_{\Theta} \sum_{n=1}^{M} C^{(n)}(\Theta^{(t)})$$

- We have $\nabla_{\Theta} \sum_n C^{(n)}(\Theta^{(t)}) = \sum_n \nabla_{\Theta} C^{(n)}(\Theta^{(t)})$

Back Propagation

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \nabla_{\Theta} \sum_{n=1}^{M} C^{(n)}(\Theta^{(t)})$$

- We have $\nabla_{\Theta} \sum_n C^{(n)}(\Theta^{(t)}) = \sum_n \nabla_{\Theta} C^{(n)}(\Theta^{(t)})$
- Let $c^{(n)} = C^{(n)}(\Theta^{(t)})$, our goal is to evaluate

$$\frac{\partial c^{(n)}}{\partial W_{i,j}^{(k)}}$$

  for all i, j, k, and n

Back Propagation

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \nabla_{\Theta} \sum_{n=1}^{M} C^{(n)}(\Theta^{(t)})$$

- We have $\nabla_{\Theta} \sum_n C^{(n)}(\Theta^{(t)}) = \sum_n \nabla_{\Theta} C^{(n)}(\Theta^{(t)})$
- Let $c^{(n)} = C^{(n)}(\Theta^{(t)})$, our goal is to evaluate

  $$\frac{\partial c^{(n)}}{\partial W_{i,j}^{(k)}}$$

  for all i, j, k, and n
- ***Back propagation*** (or simply ***backprop***) is an efficient way to evaluate multiple partial derivatives at once
  - Assuming the partial derivatives share some common evaluation steps

Back Propagation

$$\Theta^{(t+1)} \leftarrow \Theta^{(t)} - \eta \nabla_{\Theta} \sum_{n=1}^{M} C^{(n)}(\Theta^{(t)})$$

- We have $\nabla_{\Theta} \sum_n C^{(n)}(\Theta^{(t)}) = \sum_n \nabla_{\Theta} C^{(n)}(\Theta^{(t)})$
- Let $c^{(n)} = C^{(n)}(\Theta^{(t)})$, our goal is to evaluate

$$\frac{\partial c^{(n)}}{\partial W_{i,j}^{(k)}}$$

  for all i, j, k, and n
- ***Back propagation*** (or simply ***backprop***) is an efficient way to evaluate multiple partial derivatives at once
  - Assuming the partial derivatives share some common evaluation steps
- By the chain rule, we have

$$\frac{\partial c^{(n)}}{\partial W_{i,j}^{(k)}} = \frac{\partial c^{(n)}}{\partial z_j^{(k)}} \cdot \frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$$

Forward Pass

- The second term: $\frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$

Forward Pass

- The second term: $\frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$
- When $k = 1$, we have $z_j^{(1)} = \sum_i W_{i,j}^{(1)} x_i^{(n)}$ and

$$\frac{\partial z_j^{(1)}}{\partial W_{i,j}^{(1)}} = x_i^{(n)}$$

Forward Pass

- The second term: $\frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$
- When $k = 1$, we have $z_j^{(1)} = \sum_i W_{i,j}^{(1)} x_i^{(n)}$ and

$$\frac{\partial z_j^{(1)}}{\partial W_{i,j}^{(1)}} = x_i^{(n)}$$

- Otherwise $(k > 1)$, we have $z_j^{(k)} = \sum_i W_{i,j}^{(k)} a_i^{(k-1)}$ and

$$\frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}} = a_i^{(k-1)}$$

Forward Pass

- The second term: $\frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$
- When $k = 1$, we have $z_j^{(1)} = \sum_i W_{i,j}^{(1)} x_i^{(n)}$ and

$$\frac{\partial z_j^{(1)}}{\partial W_{i,j}^{(1)}} = x_i^{(n)}$$

- Otherwise $(k > 1)$, we have $z_j^{(k)} = \sum_i W_{i,j}^{(k)} a_i^{(k-1)}$ and

$$\frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}} = a_i^{(k-1)}$$

- We can get the second terms of all $\frac{\partial c^{(n)}}{\partial W_{i,j}^{(k)}}$'s starting from the ***most shallow*** layer

Backward Pass I

- Conversely, we can get the first terms of $\frac{\partial c^{(n)}}{\partial W_{i,j}^{(k)}} = \frac{\partial c^{(n)}}{\partial z_j^{(k)}} \cdot \frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$ starting from the ***deepest*** layer

Backward Pass I

- Conversely, we can get the first terms of $\frac{\partial c^{(n)}}{\partial W_{i,j}^{(k)}} = \frac{\partial c^{(n)}}{\partial z_j^{(k)}} \cdot \frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$ starting from the ***deepest*** layer
- Define ***error signal*** $\delta_j^{(k)} = \frac{\partial c^{(n)}}{\partial z_j^{(k)}}$

Backward Pass I

- Conversely, we can get the first terms of $\frac{\partial c^{(n)}}{\partial W_{i,j}^{(k)}} = \frac{\partial c^{(n)}}{\partial z_j^{(k)}} \cdot \frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$ starting from the ***deepest*** layer
- Define ***error signal*** $\delta_j^{(k)} = \frac{\partial c^{(n)}}{\partial z_j^{(k)}}$
- When $k = L$, the evaluation varies from task to task
  - Depending on the definition of functions $\mathrm{act}^{(L)}$ and $C^{(n)}$

Backward Pass I

- Conversely, we can get the first terms of $\frac{\partial c^{(n)}}{\partial W_{i,j}^{(k)}} = \frac{\partial c^{(n)}}{\partial z_j^{(k)}} \cdot \frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$ starting from the ***deepest*** layer
- Define ***error signal*** $\delta_j^{(k)} = \frac{\partial c^{(n)}}{\partial z_j^{(k)}}$
- When $k = L$, the evaluation varies from task to task
  - Depending on the definition of functions $\text{act}^{(L)}$ and $C^{(n)}$
- E.g., in binary classification, we have:

$$\delta^{(L)} = \frac{\partial c^{(n)}}{\partial z^{(L)}} = \frac{\partial \zeta((1-2y^{(n)})z^{(L)})}{\partial z^{(L)}} = \sigma((1-2y^{(n)})z^{(L)}) \cdot (1-2y^{(n)})$$

Backward Pass II

- When $k < L$, we have

$$\delta_j^{(k)} = \frac{\partial c^{(n)}}{\partial z_j^{(k)}} = \frac{\partial c^{(n)}}{\partial a_j^{(k)}} \cdot \frac{\partial a_j^{(k)}}{\partial z_j^{(k)}} = \frac{\partial c^{(n)}}{\partial a_j^{(k)}} \cdot \mathrm{act}'(z_j^{(k)})$$

Backward Pass II

- When $k < L$, we have

$$\begin{aligned}\delta_j^{(k)} &= \frac{\partial c^{(n)}}{\partial z_j^{(k)}} = \frac{\partial c^{(n)}}{\partial a_j^{(k)}} \cdot \frac{\partial a_j^{(k)}}{\partial z_j^{(k)}} = \frac{\partial c^{(n)}}{\partial a_j^{(k)}} \cdot \mathrm{act}'(z_j^{(k)}) \\ &= \left(\sum_s \frac{\partial c^{(n)}}{\partial z_s^{(k+1)}} \cdot \frac{\partial z_s^{(k+1)}}{\partial a_j^{(k)}} \right) \mathrm{act}'(z_j^{(k)})\end{aligned}$$

Theorem (Chain Rule)

Let $\boldsymbol{g} : \mathbb{R} \to \mathbb{R}^d$ *and* $f : \mathbb{R}^d \to \mathbb{R}$*, then f*

$$(f \circ \boldsymbol{g})'(x) = f'(\boldsymbol{g}(x))\boldsymbol{g}'(x) = \nabla f(\boldsymbol{g}(x))^\top \begin{bmatrix} g_1'(x) \\ \vdots \\ g_d'(x) \end{bmatrix}.$$

Backward Pass II

- When $k < L$, we have

$$\begin{aligned}\delta_j^{(k)} &= \frac{\partial c^{(n)}}{\partial z_j^{(k)}} = \frac{\partial c^{(n)}}{\partial a_j^{(k)}} \cdot \frac{\partial a_j^{(k)}}{\partial z_j^{(k)}} = \frac{\partial c^{(n)}}{\partial a_j^{(k)}} \cdot \mathrm{act}'(z_j^{(k)}) \\ &= \left(\sum_s \frac{\partial c^{(n)}}{\partial z_s^{(k+1)}} \cdot \frac{\partial z_s^{(k+1)}}{\partial a_j^{(k)}} \right) \mathrm{act}'(z_j^{(k)}) \\ &= \left(\sum_s \delta_s^{(k+1)} \cdot \frac{\partial \sum_i W_{i,s}^{(k+1)} a_i^{(k)}}{\partial a_j^{(k)}} \right) \mathrm{act}'(z_j^{(k)})\end{aligned}$$

Theorem (Chain Rule)

Let $\boldsymbol{g} : \mathbb{R} \to \mathbb{R}^d$ and $f : \mathbb{R}^d \to \mathbb{R}$, then f

$$(f \circ \boldsymbol{g})'(x) = f'(\boldsymbol{g}(x))\boldsymbol{g}'(x) = \nabla f(\boldsymbol{g}(x))^\top \begin{bmatrix} g_1'(x) \\ \vdots \\ g_d'(x) \end{bmatrix}.$$

Backward Pass II

- When $k < L$, we have

$$\begin{aligned}
\delta_j^{(k)} &= \frac{\partial c^{(n)}}{\partial z_j^{(k)}} = \frac{\partial c^{(n)}}{\partial a_j^{(k)}} \cdot \frac{\partial a_j^{(k)}}{\partial z_j^{(k)}} = \frac{\partial c^{(n)}}{\partial a_j^{(k)}} \cdot \mathrm{act}'(z_j^{(k)}) \\
&= \left(\sum_s \frac{\partial c^{(n)}}{\partial z_s^{(k+1)}} \cdot \frac{\partial z_s^{(k+1)}}{\partial a_j^{(k)}} \right) \mathrm{act}'(z_j^{(k)}) \\
&= \left(\sum_s \delta_s^{(k+1)} \cdot \frac{\partial \sum_i W_{i,s}^{(k+1)} a_i^{(k)}}{\partial a_j^{(k)}} \right) \mathrm{act}'(z_j^{(k)}) \\
&= \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)} \right) \mathrm{act}'(z_j^{(k)})
\end{aligned}$$

Theorem (Chain Rule)

Let $\boldsymbol{g} : \mathbb{R} \to \mathbb{R}^d$ and $f : \mathbb{R}^d \to \mathbb{R}$, then f

$$(f \circ \boldsymbol{g})'(x) = f'(\boldsymbol{g}(x))\boldsymbol{g}'(x) = \nabla f(\boldsymbol{g}(x))^\top \begin{bmatrix} g_1'(x) \\ \vdots \\ g_d'(x) \end{bmatrix}.$$

Backward Pass III

$$\delta_j^{(k)} = \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)}\right) \mathrm{act}'(z_j^{(k)})$$

- We can evaluate all $\delta_j^{(k)}$'s starting from the deepest layer

Backward Pass III

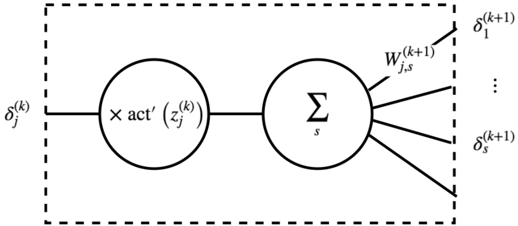
$$\delta_j^{(k)} = \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)}\right) \mathrm{act}'(z_j^{(k)})$$

- We can evaluate all $\delta_j^{(k)}$'s starting from the deepest layer
- The information propagate along a new kind of feedforward network:

Backward Pass III

$$\delta_j^{(k)} = \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)}\right) \text{act}'(z_j^{(k)})$$

- We can evaluate all $\delta_j^{(k)}$'s starting from the deepest layer
- The information propagate along a new kind of feedforward network:



Backprop Algorithm (Minibatch Size $M = 1$)

Input: $(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})$ and $\Theta^{(t)}$

Forward pass:

$\boldsymbol{a}^{(0)} \leftarrow \begin{bmatrix} 1 & \boldsymbol{x}^{(n)} \end{bmatrix}^\top$;

for $k \leftarrow 1$ *to* L **do**

$\quad \boldsymbol{z}^{(k)} \leftarrow \boldsymbol{W}^{(k)\top} \boldsymbol{a}^{(k-1)}$;

$\quad \boldsymbol{a}^{(k)} \leftarrow \mathrm{act}(\boldsymbol{z}^{(k)})$;

end

Backward pass:

Compute error signal $\delta^{(L)}$ (e.g., $(1-2y^{(n)})\sigma((1-2y^{(n)})z^{(L)})$ in binary classification)

for $k \leftarrow L-1$ *to* 1 **do**

$\quad \boldsymbol{\delta}^{(k)} \leftarrow \mathrm{act}'(\boldsymbol{z}^{(k)}) \odot (\boldsymbol{W}^{(k+1)} \boldsymbol{\delta}^{(k+1)})$;

end

Return $\frac{\partial c^{(n)}}{\partial \boldsymbol{W}^{(k)}} = \boldsymbol{a}^{(k-1)} \otimes \boldsymbol{\delta}^{(k)}$ for all k

Backprop Algorithm (Minibatch Size $M > 1$)

Input: $\{(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})\}_{n=1}^{M}$ and $\Theta^{(t)}$

Forward pass:

$\boldsymbol{A}^{(0)} \leftarrow \left[\begin{array}{ccc} \boldsymbol{a}^{(0,1)} & \cdots & \boldsymbol{a}^{(0,M)} \end{array} \right]^{\top}$;

for $k \leftarrow 1$ *to* L **do**

$\quad \boldsymbol{Z}^{(k)} \leftarrow \boldsymbol{A}^{(k-1)} \boldsymbol{W}^{(k)}$;

$\quad \boldsymbol{A}^{(k)} \leftarrow \mathrm{act}(\boldsymbol{Z}^{(k)})$;

end

Backward pass:

Compute error signals

$$\Delta^{(L)} = \left[\begin{array}{ccc} \boldsymbol{\delta}^{(L,0)} & \cdots & \boldsymbol{\delta}^{(L,M)} \end{array} \right]^{\top}$$

for $k \leftarrow L-1$ *to* 1 **do**

$\quad \Delta^{(k)} \leftarrow \mathrm{act}'(\boldsymbol{Z}^{(k)}) \odot (\Delta^{(k+1)} \boldsymbol{W}^{(k+1)\top})$;

end

Return $\frac{\partial c^{(n)}}{\partial \boldsymbol{W}^{(k)}} = \sum_{n=1}^{M} \boldsymbol{a}^{(k-1,n)} \otimes \boldsymbol{\delta}^{(k,n)}$ for all k

Backprop Algorithm (Minibatch Size $M > 1$)

Input: $\{(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})\}_{n=1}^{M}$ and $\Theta^{(t)}$

Forward pass:

$\boldsymbol{A}^{(0)} \leftarrow \begin{bmatrix} \boldsymbol{a}^{(0,1)} & \cdots & \boldsymbol{a}^{(0,M)} \end{bmatrix}^\top$;

for $k \leftarrow 1$ *to* L **do**

$\quad \boldsymbol{Z}^{(k)} \leftarrow \boldsymbol{A}^{(k-1)} \boldsymbol{W}^{(k)}$;

$\quad \boldsymbol{A}^{(k)} \leftarrow \mathrm{act}(\boldsymbol{Z}^{(k)})$;

end

Backward pass:

Compute error signals

$$\Delta^{(L)} = \begin{bmatrix} \boldsymbol{\delta}^{(L,0)} & \cdots & \boldsymbol{\delta}^{(L,M)} \end{bmatrix}^\top$$

for $k \leftarrow L-1$ *to* 1 **do**

$\quad \Delta^{(k)} \leftarrow \mathrm{act}'(\boldsymbol{Z}^{(k)}) \odot (\Delta^{(k+1)} \boldsymbol{W}^{(k+1)\top})$;

end

Return $\frac{\partial c^{(n)}}{\partial \boldsymbol{W}^{(k)}} = \sum_{n=1}^{M} \boldsymbol{a}^{(k-1,n)} \otimes \boldsymbol{\delta}^{(k,n)}$ for all k

- Speed up with GPUs?

Backprop Algorithm (Minibatch Size $M > 1$)

Input: $\{(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})\}_{n=1}^{M}$ and $\Theta^{(t)}$
Forward pass:
$\boldsymbol{A}^{(0)} \leftarrow \left[\begin{array}{ccc} \boldsymbol{a}^{(0,1)} & \cdots & \boldsymbol{a}^{(0,M)} \end{array}\right]^\top$;
for $k \leftarrow 1$ *to* L **do**
$\quad \boldsymbol{Z}^{(k)} \leftarrow \boldsymbol{A}^{(k-1)} \boldsymbol{W}^{(k)}$;
$\quad \boldsymbol{A}^{(k)} \leftarrow \mathrm{act}(\boldsymbol{Z}^{(k)})$;
end
Backward pass:
Compute error signals
$$\Delta^{(L)} = \left[\begin{array}{ccc} \delta^{(L,0)} & \cdots & \delta^{(L,M)} \end{array}\right]^\top$$
for $k \leftarrow L-1$ *to* 1 **do**
$\quad \Delta^{(k)} \leftarrow \mathrm{act}'(\boldsymbol{Z}^{(k)}) \odot (\Delta^{(k+1)} \boldsymbol{W}^{(k+1)\top})$;
end
Return $\frac{\partial c^{(n)}}{\partial \boldsymbol{W}^{(k)}} = \sum_{n=1}^{M} \boldsymbol{a}^{(k-1,n)} \otimes \delta^{(k,n)}$ for all k

- Speed up with GPUs?
- Large width ($D^{(k)}$) at each layer

Backprop Algorithm (Minibatch Size $M > 1$)

Input: $\{(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})\}_{n=1}^{M}$ and $\Theta^{(t)}$

Forward pass:

$$\boldsymbol{A}^{(0)} \leftarrow \begin{bmatrix} \boldsymbol{a}^{(0,1)} & \cdots & \boldsymbol{a}^{(0,M)} \end{bmatrix}^\top;$$

for $k \leftarrow 1$ *to* L **do**

$\quad \boldsymbol{Z}^{(k)} \leftarrow \boldsymbol{A}^{(k-1)} \boldsymbol{W}^{(k)}$;

$\quad \boldsymbol{A}^{(k)} \leftarrow \mathrm{act}(\boldsymbol{Z}^{(k)})$;

end

Backward pass:

Compute error signals

$$\Delta^{(L)} = \begin{bmatrix} \delta^{(L,0)} & \cdots & \delta^{(L,M)} \end{bmatrix}^\top$$

for $k \leftarrow L-1$ *to* 1 **do**

$\quad \Delta^{(k)} \leftarrow \mathrm{act}'(\boldsymbol{Z}^{(k)}) \odot (\Delta^{(k+1)} \boldsymbol{W}^{(k+1)\top})$;

end

Return $\frac{\partial c^{(n)}}{\partial \boldsymbol{W}^{(k)}} = \sum_{n=1}^{M} \boldsymbol{a}^{(k-1,n)} \otimes \delta^{(k,n)}$ for all k

- Speed up with GPUs?
- Large width ($D^{(k)}$) at each layer
- Large batch size

Outline

1. The Basics
   - Example: Learning the XOR
2. Training
   - Back Propagation
3. **Neuron Design**
   - Cost Function & Output Neurons
   - Hidden Neurons
4. Architecture Design
   - Architecture Tuning

Neuron Design

- The design of modern neurons is largely influenced by how an NN is trained

Neuron Design

- The design of modern neurons is largely influenced by how an NN is trained
- Maximum likelihood principle:

$$\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X} \,|\, \Theta) = \arg\min_{\Theta} \sum_i -\log \mathrm{P}(\boldsymbol{y}^{(i)} \,|\, \boldsymbol{x}^{(i)}, \Theta)$$

  - Universal cost function

Neuron Design

- The design of modern neurons is largely influenced by how an NN is trained
- Maximum likelihood principle:

$$\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X} \mid \Theta) = \arg\min_{\Theta} \sum_i -\log \mathrm{P}(\boldsymbol{y}^{(i)} \mid \boldsymbol{x}^{(i)}, \Theta)$$

  - Universal cost function
  - Different output units for different $\mathrm{P}(\mathbf{y} \mid \mathbf{x})$

Neuron Design

- The design of modern neurons is largely influenced by how an NN is trained
- Maximum likelihood principle:

$$\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X}\,|\,\Theta) = \arg\min_{\Theta} \sum_i -\log \mathrm{P}(\boldsymbol{y}^{(i)}\,|\,\boldsymbol{x}^{(i)}, \Theta)$$

  - Universal cost function
  - Different output units for different $\mathrm{P}(\mathbf{y}\,|\,\mathbf{x})$
- Gradient-based optimization:
  - During SGD, the gradient

$$\frac{\partial c^{(n)}}{\partial W_{i,j}^{(k)}} = \frac{\partial c^{(n)}}{\partial z_j^{(k)}} \cdot \frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}} = \delta_j^{(k)} \frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$$

  should be sufficiently large before we get a satisfactory NN

Outline

1. The Basics
   - Example: Learning the XOR
2. Training
   - Back Propagation
3. **Neuron Design**
   - **Cost Function & Output Neurons**
   - Hidden Neurons
4. Architecture Design
   - Architecture Tuning

Negative Log Likelihood and Cross Entropy

- The cost function of most NNs:

$$\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X} \,|\, \Theta) = \arg\min_{\Theta} \sum_i -\log \mathrm{P}(\boldsymbol{y}^{(i)} \,|\, \boldsymbol{x}^{(i)}, \Theta)$$

Negative Log Likelihood and Cross Entropy

- The cost function of most NNs:

$$\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X} \,|\, \Theta) = \arg\min_{\Theta} \sum_i -\log \mathrm{P}(\boldsymbol{y}^{(i)} \,|\, \boldsymbol{x}^{(i)}, \Theta)$$

- For NNs that output an entire distribution $\hat{\mathrm{P}}(\mathbf{y} \,|\, \mathbf{x})$, the problem can be equivalently described as minimizing the ***cross entropy*** (or KL divergence) from $\hat{\mathrm{P}}$ to the empirical distribution of data:

$$\arg\min_{\hat{\mathrm{P}}} -\mathrm{E}_{(\mathbf{x},\mathbf{y})\sim \mathrm{Empirical}(\mathbb{X})} \left[\log \hat{\mathrm{P}}(\mathbf{y} \,|\, \mathbf{x})\right]$$

Negative Log Likelihood and Cross Entropy

- The cost function of most NNs:

$$\arg\max_{\Theta} \log \mathrm{P}(\mathbb{X}\,|\,\Theta) = \arg\min_{\Theta} \sum_i -\log \mathrm{P}(\boldsymbol{y}^{(i)}\,|\,\boldsymbol{x}^{(i)}, \Theta)$$

- For NNs that output an entire distribution $\hat{\mathrm{P}}(\mathbf{y}\,|\,\mathbf{x})$, the problem can be equivalently described as minimizing the ***cross entropy*** (or KL divergence) from $\hat{\mathrm{P}}$ to the empirical distribution of data:

$$\arg\min_{\hat{\mathrm{P}}} -\mathrm{E}_{(\mathbf{x},\mathbf{y})\sim \mathrm{Empirical}(\mathbb{X})}\left[\log \hat{\mathrm{P}}(\mathbf{y}\,|\,\mathbf{x})\right]$$

- Provides a consistent way to define output units

Sigmoid Units for Bernoulli Output Distributions

- In binary classification, we assuming $\mathrm{P}(\mathrm{y}=1\,|\,\boldsymbol{x}) \sim \mathrm{Bernoulli}(\rho)$
  - $y \in \{0,1\}$ and $\rho \in (0,1)$

Sigmoid Units for Bernoulli Output Distributions

- In binary classification, we assuming $\mathrm{P}(\mathrm{y}=1\,|\,\boldsymbol{x}) \sim \mathrm{Bernoulli}(\rho)$
  - $y \in \{0,1\}$ and $\rho \in (0,1)$
- Sigmoid output unit:

$$a^{(L)} = \hat{\rho} = \sigma(z^{(L)}) = \frac{\exp(z^{(L)})}{\exp(z^{(L)})+1}$$

Sigmoid Units for Bernoulli Output Distributions

- In binary classification, we assuming $\mathrm{P}(\mathrm{y}=1\,|\,\boldsymbol{x}) \sim \mathrm{Bernoulli}(\rho)$
  - $y \in \{0,1\}$ and $\rho \in (0,1)$
- Sigmoid output unit:

$$a^{(L)} = \hat{\rho} = \sigma(z^{(L)}) = \frac{\exp(z^{(L)})}{\exp(z^{(L)})+1}$$

- $\delta^{(L)} = \frac{\partial c^{(n)}}{\partial z^{(L)}} = \frac{\partial -\log \hat{\mathrm{P}}(y^{(n)}\,|\,\boldsymbol{x}^{(n)};\Theta)}{\partial z^{(L)}} = (1-2y^{(n)})\sigma((1-2y^{(n)})z^{(L)})$
  - Close to 0 only when $y^{(n)}=1$ and $z^{(L)}$ is large positive;

Sigmoid Units for Bernoulli Output Distributions

- In binary classification, we assuming $\mathrm{P}(\mathrm{y}=1\,|\,\boldsymbol{x}) \sim \mathrm{Bernoulli}(\rho)$
  - $y \in \{0,1\}$ and $\rho \in (0,1)$
- Sigmoid output unit:

$$a^{(L)} = \hat{\rho} = \sigma(z^{(L)}) = \frac{\exp(z^{(L)})}{\exp(z^{(L)})+1}$$

- $\delta^{(L)} = \frac{\partial c^{(n)}}{\partial z^{(L)}} = \frac{\partial -\log \hat{\mathrm{P}}(y^{(n)}\,|\,\boldsymbol{x}^{(n)};\Theta)}{\partial z^{(L)}} = (1-2y^{(n)})\sigma((1-2y^{(n)})z^{(L)})$
  - Close to 0 only when $y^{(n)}=1$ and $z^{(L)}$ is large positive; or $y^{(n)}=0$ and $z^{(L)}$ is small negative

Sigmoid Units for Bernoulli Output Distributions

- In binary classification, we assuming $\mathrm{P}(\mathrm{y}=1\,|\,\boldsymbol{x}) \sim \mathrm{Bernoulli}(\rho)$
  - $y \in \{0,1\}$ and $\rho \in (0,1)$
- Sigmoid output unit:

$$a^{(L)} = \hat{\rho} = \sigma(z^{(L)}) = \frac{\exp(z^{(L)})}{\exp(z^{(L)})+1}$$

- $\delta^{(L)} = \frac{\partial c^{(n)}}{\partial z^{(L)}} = \frac{\partial - \log \hat{\mathrm{P}}(y^{(n)}\,|\,\boldsymbol{x}^{(n)};\Theta)}{\partial z^{(L)}} = (1-2y^{(n)})\sigma((1-2y^{(n)})z^{(L)})$
  - Close to 0 only when $y^{(n)}=1$ and $z^{(L)}$ is large positive; or $y^{(n)}=0$ and $z^{(L)}$ is small negative
- The loss $c^{(n)}$ saturates (becomes flat) only when $\hat{\rho}$ is "correct"

Softmax Units for Categorical Output Distributions I

- In multiclass classification, we can assume that $\mathrm{P}(\mathbf{y}\,|\,\mathbf{x}) \sim \mathrm{Categorical}(\rho)$, where $\boldsymbol{y}, \rho \in \mathbb{R}^K$ and $\mathbf{1}^\top \rho = 1$

Softmax Units for Categorical Output Distributions I

- In multiclass classification, we can assume that $\mathrm{P}(\mathbf{y}\,|\,\mathbf{x}) \sim \mathrm{Categorical}(\rho)$, where $\boldsymbol{y}, \rho \in \mathbb{R}^K$ and $\mathbf{1}^\top \rho = 1$
- ***Softmax*** units:

$$a_j^{(L)} = \hat{\rho}_j = \mathrm{sofmax}(\boldsymbol{z}^{(L)})_j = \frac{\exp(z_j^{(L)})}{\sum_{i=1}^{K} \exp(z_i^{(L)})}$$

Softmax Units for Categorical Output Distributions I

- In multiclass classification, we can assume that $\mathrm{P}(\mathbf{y}\,|\,\mathbf{x}) \sim \mathrm{Categorical}(\boldsymbol{\rho})$, where $\boldsymbol{y}, \boldsymbol{\rho} \in \mathbb{R}^K$ and $\mathbf{1}^\top \boldsymbol{\rho} = 1$
- ***Softmax*** units:

$$a_j^{(L)} = \hat{\rho}_j = \mathrm{sofmax}(\boldsymbol{z}^{(L)})_j = \frac{\exp(z_j^{(L)})}{\sum_{i=1}^{K} \exp(z_i^{(L)})}$$

- Actually, to define a Categorical distribution, we only need $\rho_1, \cdots, \rho_{K-1}$ ($\rho_K = 1 - \sum_{i=1}^{K-1} \rho_i$ can be discarded)

Softmax Units for Categorical Output Distributions I

- In multiclass classification, we can assume that $\mathrm{P}(\mathbf{y}\,|\,\mathbf{x}) \sim \mathrm{Categorical}(\boldsymbol{\rho})$, where $\boldsymbol{y}, \boldsymbol{\rho} \in \mathbb{R}^K$ and $\mathbf{1}^\top \boldsymbol{\rho} = 1$
- ***Softmax*** units:

$$a_j^{(L)} = \hat{\rho}_j = \mathrm{sofmax}(\boldsymbol{z}^{(L)})_j = \frac{\exp(z_j^{(L)})}{\sum_{i=1}^{K} \exp(z_i^{(L)})}$$

- Actually, to define a Categorical distribution, we only need $\rho_1, \cdots, \rho_{K-1}$ ($\rho_K = 1 - \sum_{i=1}^{K-1} \rho_i$ can be discarded)
- We can alternatively define $K-1$ output units (discarding $a_K^{(L)} = \hat{\rho}_K = 1$):

$$a_j^{(L)} = \hat{\rho}_j = \frac{\exp(z_j^{(L)})}{\sum_{i=1}^{K-1} \exp(z_i^{(L)}) + 1}$$

that is a direct generalization of σ in binary classification

Softmax Units for Categorical Output Distributions I

- In multiclass classification, we can assume that $\mathrm{P}(\mathbf{y}\,|\,\mathbf{x}) \sim \mathrm{Categorical}(\boldsymbol{\rho})$, where $\boldsymbol{y}, \boldsymbol{\rho} \in \mathbb{R}^K$ and $\mathbf{1}^\top \boldsymbol{\rho} = 1$
- ***Softmax*** units:

$$a_j^{(L)} = \hat{\rho}_j = \mathrm{sofmax}(\boldsymbol{z}^{(L)})_j = \frac{\exp(z_j^{(L)})}{\sum_{i=1}^K \exp(z_i^{(L)})}$$

- Actually, to define a Categorical distribution, we only need $\rho_1, \cdots, \rho_{K-1}$ ($\rho_K = 1 - \sum_{i=1}^{K-1} \rho_i$ can be discarded)
- We can alternatively define $K-1$ output units (discarding $a_K^{(L)} = \hat{\rho}_K = 1$):

$$a_j^{(L)} = \hat{\rho}_j = \frac{\exp(z_j^{(L)})}{\sum_{i=1}^{K-1} \exp(z_i^{(L)}) + 1}$$

  that is a direct generalization of σ in binary classification
- In practice, the two versions make little difference

Softmax Units for Categorical Output Distributions II

- Now we have

$$\delta_j^{(L)} = \frac{\partial c^{(n)}}{\partial z_j^{(L)}} = \frac{\partial - \log \hat{\mathrm{P}}(y^{(n)} \,|\, \boldsymbol{x}^{(n)}; \Theta)}{\partial z_j^{(L)}} = \frac{\partial - \log \left(\prod_i \hat{\rho}_i^{1(y^{(n)}; y^{(n)} = i)} \right)}{\partial z_j^{(L)}}$$

Softmax Units for Categorical Output Distributions II

- Now we have

$$\delta_j^{(L)} = \frac{\partial c^{(n)}}{\partial z_j^{(L)}} = \frac{\partial - \log \hat{\mathrm{P}}(y^{(n)} \,|\, \boldsymbol{x}^{(n)}; \Theta)}{\partial z_j^{(L)}} = \frac{\partial - \log \left(\prod_i \hat{\rho}_i^{1(y^{(n)}; y^{(n)} = i)} \right)}{\partial z_j^{(L)}}$$

- If $y^{(n)} = j$, then $\delta_j^{(L)} = -\frac{\partial \log \hat{\rho}_j}{\partial z_j^{(L)}} = -\frac{1}{\hat{\rho}_j} \left(\hat{\rho}_j - \hat{\rho}_j^2 \right) = \hat{\rho}_j - 1$

Softmax Units for Categorical Output Distributions II

- Now we have

$$\delta_j^{(L)} = \frac{\partial c^{(n)}}{\partial z_j^{(L)}} = \frac{\partial - \log \hat{\mathrm{P}}(y^{(n)} \,|\, \boldsymbol{x}^{(n)}; \Theta)}{\partial z_j^{(L)}} = \frac{\partial - \log \left(\prod_i \hat{\rho}_i^{1(y^{(n)}; y^{(n)} = i)} \right)}{\partial z_j^{(L)}}$$

- If $y^{(n)} = j$, then $\delta_j^{(L)} = -\frac{\partial \log \hat{\rho}_j}{\partial z_j^{(L)}} = -\frac{1}{\hat{\rho}_j} \left(\hat{\rho}_j - \hat{\rho}_j^2 \right) = \hat{\rho}_j - 1$
    - $\delta_j^{(L)}$ is close to 0 only when $\hat{\rho}_j$ is "correct"
    - In this case, $z_j^{(L)}$ dominates among all $z_i^{(L)}$'s

Softmax Units for Categorical Output Distributions II

- Now we have

$$\delta_j^{(L)} = \frac{\partial c^{(n)}}{\partial z_j^{(L)}} = \frac{\partial - \log \hat{\mathrm{P}}(y^{(n)} \,|\, \boldsymbol{x}^{(n)}; \Theta)}{\partial z_j^{(L)}} = \frac{\partial - \log \left(\prod_i \hat{\rho}_i^{1(y^{(n)}; y^{(n)}=i)} \right)}{\partial z_j^{(L)}}$$

- If $y^{(n)} = j$, then $\delta_j^{(L)} = -\frac{\partial \log \hat{\rho}_j}{\partial z_j^{(L)}} = -\frac{1}{\hat{\rho}_j} \left(\hat{\rho}_j - \hat{\rho}_j^2 \right) = \hat{\rho}_j - 1$
  - $\delta_j^{(L)}$ is close to 0 only when $\hat{\rho}_j$ is "correct"
  - In this case, $z_j^{(L)}$ dominates among all $z_i^{(L)}$'s
- If $y^{(n)} = i \neq j$, then $\delta_j^{(L)} = -\frac{\partial \log \hat{\rho}_i}{\partial z_j^{(L)}} = -\frac{1}{\hat{\rho}_i} (-\hat{\rho}_i \hat{\rho}_j) = \hat{\rho}_j$

Softmax Units for Categorical Output Distributions II

- Now we have

$$\delta_j^{(L)} = \frac{\partial c^{(n)}}{\partial z_j^{(L)}} = \frac{\partial - \log \hat{\mathrm{P}}(y^{(n)} \,|\, \boldsymbol{x}^{(n)}; \Theta)}{\partial z_j^{(L)}} = \frac{\partial - \log \left(\prod_i \hat{\rho}_i^{1(y^{(n)}; y^{(n)}=i)} \right)}{\partial z_j^{(L)}}$$

- If $y^{(n)} = j$, then $\delta_j^{(L)} = -\frac{\partial \log \hat{\rho}_j}{\partial z_j^{(L)}} = -\frac{1}{\hat{\rho}_j}\left(\hat{\rho}_j - \hat{\rho}_j^2\right) = \hat{\rho}_j - 1$
  - $\delta_j^{(L)}$ is close to 0 only when $\hat{\rho}_j$ is "correct"
  - In this case, $z_j^{(L)}$ dominates among all $z_i^{(L)}$'s
- If $y^{(n)} = i \neq j$, then $\delta_j^{(L)} = -\frac{\partial \log \hat{\rho}_i}{\partial z_j^{(L)}} = -\frac{1}{\hat{\rho}_i}(-\hat{\rho}_i \hat{\rho}_j) = \hat{\rho}_j$
  - Again, close to 0 only when $\hat{\rho}_j$ is "correct"

Linear Units for Gaussian Means

- An NN can also output just one conditional statistic of $\mathbf{y}$ given $\boldsymbol{x}$

Linear Units for Gaussian Means

- An NN can also output just one conditional statistic of $\mathbf{y}$ given $\boldsymbol{x}$
- For example, we can assume $\mathrm{P}(\mathbf{y}\,|\,\mathbf{x}) \sim \mathcal{N}(\mu, \Sigma)$ for regression

Linear Units for Gaussian Means

- An NN can also output just one conditional statistic of $\mathbf{y}$ given $\boldsymbol{x}$
- For example, we can assume $\mathrm{P}(\mathbf{y}\,|\,\mathbf{x}) \sim \mathcal{N}(\mu, \Sigma)$ for regression
- How to design output neurons if we want to predict the mean $\hat{\mu}$?

Linear Units for Gaussian Means

- An NN can also output just one conditional statistic of $\mathbf{y}$ given $\boldsymbol{x}$
- For example, we can assume $\mathrm{P}(\mathbf{y}\,|\,\mathbf{x}) \sim \mathcal{N}(\mu, \Sigma)$ for regression
- How to design output neurons if we want to predict the mean $\hat{\mu}$?
- ***Linear*** units:

$$\boldsymbol{a}^{(L)} = \hat{\mu} = \boldsymbol{z}^{(L)}$$

Linear Units for Gaussian Means

- An NN can also output just one conditional statistic of $\mathbf{y}$ given $\boldsymbol{x}$
- For example, we can assume $\mathrm{P}(\mathbf{y}\,|\,\mathbf{x}) \sim \mathcal{N}(\mu, \Sigma)$ for regression
- How to design output neurons if we want to predict the mean $\hat{\mu}$?
- ***Linear*** units:
$$\boldsymbol{a}^{(L)} = \hat{\mu} = \boldsymbol{z}^{(L)}$$
- We have
$$\delta^{(L)} = \frac{\partial c^{(n)}}{\partial \boldsymbol{z}^{(L)}} = \frac{\partial - \log \mathcal{N}(\boldsymbol{y}^{(n)}; \hat{\mu}, \Sigma)}{\partial \boldsymbol{z}^{(L)}}$$
- Let $\Sigma = \boldsymbol{I}$, maximizing the log-likelihood is equivalent to minimizing the SSE/MSE
  - $\delta^{(L)} = \partial \|\boldsymbol{y}^{(n)} - \boldsymbol{z}^{(L)}\|^2 / \partial \boldsymbol{z}^{(L)}$ (see linear regression)

Linear Units for Gaussian Means

- An NN can also output just one conditional statistic of $\mathbf{y}$ given $\boldsymbol{x}$
- For example, we can assume $\mathrm{P}(\mathbf{y}\,|\,\mathbf{x}) \sim \mathcal{N}(\mu, \Sigma)$ for regression
- How to design output neurons if we want to predict the mean $\hat{\mu}$?
- ***Linear*** units:

$$\boldsymbol{a}^{(L)} = \hat{\mu} = \boldsymbol{z}^{(L)}$$

- We have

$$\delta^{(L)} = \frac{\partial c^{(n)}}{\partial \boldsymbol{z}^{(L)}} = \frac{\partial - \log \mathcal{N}(\boldsymbol{y}^{(n)}; \hat{\mu}, \Sigma)}{\partial \boldsymbol{z}^{(L)}}$$

- Let $\Sigma = \boldsymbol{I}$, maximizing the log-likelihood is equivalent to minimizing the SSE/MSE
  - $\delta^{(L)} = \partial \|\boldsymbol{y}^{(n)} - \boldsymbol{z}^{(L)}\|^2 / \partial \boldsymbol{z}^{(L)}$ (see linear regression)
- Linear units do not saturate, so they pose little difficulty for gradient based optimization

Outline

1. The Basics
   - Example: Learning the XOR
2. Training
   - Back Propagation
3. Neuron Design
   - Cost Function & Output Neurons
   - Hidden Neurons
4. Architecture Design
   - Architecture Tuning

Design Considerations

- Most units differ from each other only in activation functions:

$$\boldsymbol{a}^{(k)} = \mathrm{act}(\boldsymbol{z}^{(k)}) = \mathrm{act}(\boldsymbol{W}^{(k)\top}\boldsymbol{a}^{(k-1)})$$

Design Considerations

- Most units differ from each other only in activation functions:

$$\boldsymbol{a}^{(k)} = \mathrm{act}(\boldsymbol{z}^{(k)}) = \mathrm{act}(\boldsymbol{W}^{(k)\top}\boldsymbol{a}^{(k-1)})$$

- Why use ReLU as default hidden units?
  - $\mathrm{act}(\boldsymbol{z}^{(k)}) = \max(0, \boldsymbol{z}^{(k)})$

Design Considerations

- Most units differ from each other only in activation functions:

$$\boldsymbol{a}^{(k)} = \mathrm{act}(\boldsymbol{z}^{(k)}) = \mathrm{act}(\boldsymbol{W}^{(k)\top}\boldsymbol{a}^{(k-1)})$$

- Why use ReLU as default hidden units?
  - $\mathrm{act}(\boldsymbol{z}^{(k)}) = \max(0, \boldsymbol{z}^{(k)})$
- Why not, for example, use Sigmoid as hidden units?

Vanishing Gradient Problem

- In backward pass of Backprop:

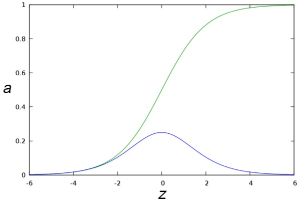
$$\delta_j^{(k)} = \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)} \right) \mathrm{act}'(z_j^{(k)})$$

Vanishing Gradient Problem

- In backward pass of Backprop:

$$\delta_j^{(k)} = \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)} \right) \mathrm{act}'(z_j^{(k)})$$

- If $\mathrm{act}'(\cdot) = \sigma'(\cdot) < 1$, then $\delta_j^{(k)}$ becomes smaller and smaller during backward pass

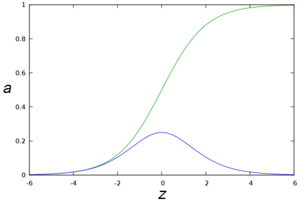


Vanishing Gradient Problem

- In backward pass of Backprop:

$$\delta_j^{(k)} = \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)}\right) \mathrm{act}'(z_j^{(k)})$$

- If $\mathrm{act}'(\cdot) = \sigma'(\cdot) < 1$, then $\delta_j^{(k)}$ becomes smaller and smaller during backward pass
- The surface of cost function becomes very flat at shallow layers

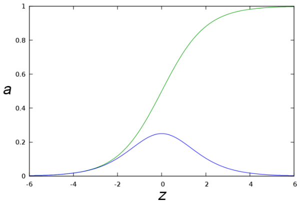


Vanishing Gradient Problem

- In backward pass of Backprop:

$$\delta_j^{(k)} = \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)} \right) \mathrm{act}'(z_j^{(k)})$$

- If $\mathrm{act}'(\cdot) = \sigma'(\cdot) < 1$, then $\delta_j^{(k)}$ becomes smaller and smaller during backward pass
- The surface of cost function becomes very flat at shallow layers
- Slows down the learning speed of entire network
  - Weights at deeper layers depend on those in shallow ones

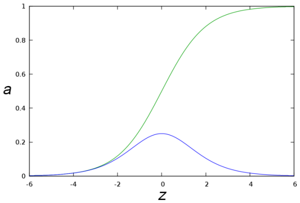


Vanishing Gradient Problem

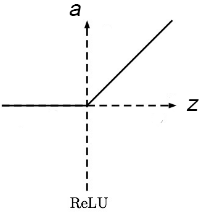
- In backward pass of Backprop:

$$\delta_j^{(k)} = \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)}\right) \mathrm{act}'(z_j^{(k)})$$

- If $\mathrm{act}'(\cdot) = \sigma'(\cdot) < 1$, then $\delta_j^{(k)}$ becomes smaller and smaller during backward pass
- The surface of cost function becomes very flat at shallow layers
- Slows down the learning speed of entire network
  - Weights at deeper layers depend on those in shallow ones
- Numeric problems, e.g., underflow

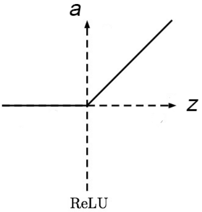


ReLU I



$$\mathrm{act}'(z^{(k)}) = \begin{cases} 1, & \text{if } z^{(k)} > 0 \\ 0, & \text{otherwise} \end{cases}$$

ReLU I

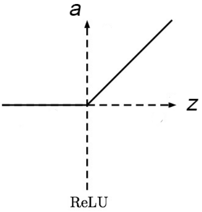


$$\mathrm{act}'(z^{(k)}) = \begin{cases} 1, & \text{if } z^{(k)} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- No vanishing gradients

$$\delta_j^{(k)} = \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)} \right) \mathrm{act}'(z_j^{(k)})$$

ReLU I



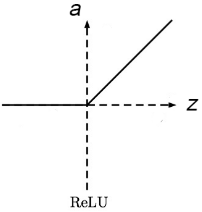
$$\text{act}'(z^{(k)}) = \begin{cases} 1, & \text{if } z^{(k)} > 0 \\ 0, & \text{otherwise} \end{cases}$$

- No vanishing gradients

$$\delta_j^{(k)} = \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)} \right) \text{act}'(z_j^{(k)})$$

- What if $z^{(k)} = 0$?

ReLU I



$$\mathrm{act}'(z^{(k)}) = \begin{cases} 1, & \text{if } z^{(k)} > 0 \\ 0, & \text{otherwise} \end{cases}$$

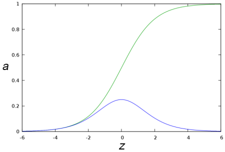
- No vanishing gradients

$$\delta_j^{(k)} = \left(\sum_s \delta_s^{(k+1)} \cdot W_{j,s}^{(k+1)}\right) \mathrm{act}'(z_j^{(k)})$$

- What if $z^{(k)} = 0$?
- In practice, we usually assign 1 or 0 randomly
  - Floating points are not precise anyway

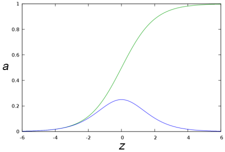
ReLU II

- Why piecewise linear?
  - To avoid vanishing gradient, we can modify $\sigma(\cdot)$ to make it steeper at middle such that $\sigma'(\cdot) > 1$



ReLU II

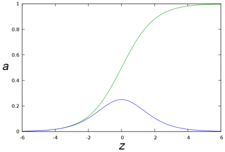
- Why piecewise linear?
  - To avoid vanishing gradient, we can modify $\sigma(\cdot)$ to make it steeper at middle such that $\sigma'(\cdot) > 1$



- The second derivative $\mathrm{ReLU}''(\cdot)$ is 0 everywhere
  - Eliminates the second-order effects and makes the gradient-based optimization more useful (than, e.g., Newton methods)

ReLU II

- Why piecewise linear?
  - To avoid vanishing gradient, we can modify $\sigma(\cdot)$ to make it steeper at middle such that $\sigma'(\cdot) > 1$

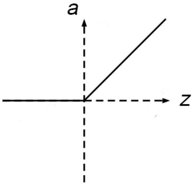


- The second derivative $\mathrm{ReLU}''(\cdot)$ is 0 everywhere
  - Eliminates the second-order effects and makes the gradient-based optimization more useful (than, e.g., Newton methods)
- Problem: for neurons with $\delta_j^{(k)} = 0$, theirs weights $\boldsymbol{W}_{:,j}^{(k)}$ will ***not*** be updated

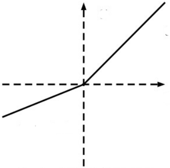
$$\frac{\partial c^{(n)}}{\partial W_{i,j}^{(k)}} = \delta^{(k)} \frac{\partial z_j^{(k)}}{\partial W_{i,j}^{(k)}}$$

- Improvement?

Leaky/Parametric ReLU



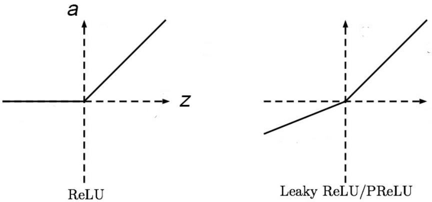
ReLU


Leaky ReLU/PReLU

$$\mathrm{act}(z^{(k)}) = \max(\alpha \cdot z^{(k)}, z^{(k)}),$$

for some $\alpha \in \mathbb{R}$

Leaky/Parametric ReLU

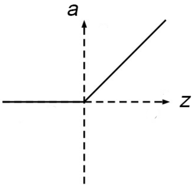


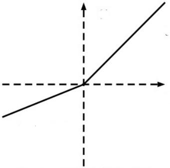
$$\mathrm{act}(z^{(k)}) = \max(\alpha \cdot z^{(k)}, z^{(k)}),$$

for some $\alpha \in \mathbb{R}$

- ***Leaky ReLU***: α is set in advance (fixed during training)
  - Usually a small value
  - Or domain-specific

Leaky/Parametric ReLU





$$\mathrm{act}(z^{(k)}) = \max(\alpha \cdot z^{(k)}, z^{(k)}),$$

for some $\alpha \in \mathbb{R}$

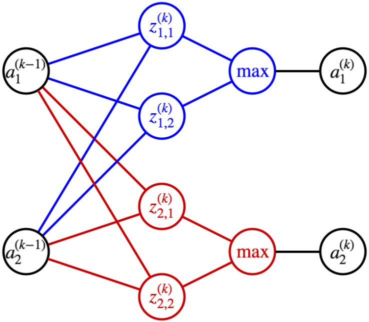
- ***Leaky ReLU***: α is set in advance (fixed during training)
  - Usually a small value
  - Or domain-specific
- Example: ***absolute value rectification*** $\alpha = -1$
  - Used for object recognition from images
  - Seek features that are invariant under a polarity reversal of the input illumination
- ***Parametric ReLU*** (PReLU): α learned automatically by gradient descent

Maxout Units I

- ***Maxout units*** generalize ReLU variants further:

$$\mathrm{act}(\boldsymbol{z}^{(k)})_j = \max_s z_{j,s}$$

  - $\boldsymbol{a}^{(k-1)}$ is linearly mapped to multiple groups of $z_{j,:}^{(k)}$'s

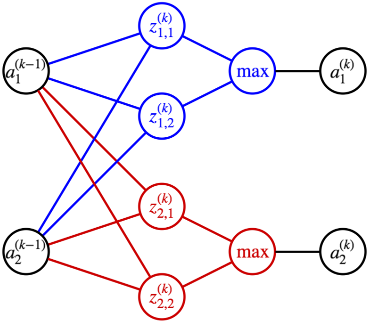


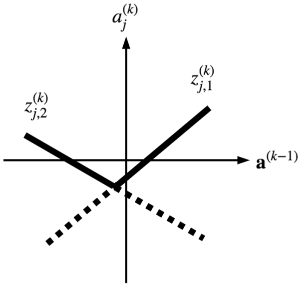
Maxout Units I

- ***Maxout units*** generalize ReLU variants further:

$$\mathrm{act}(\boldsymbol{z}^{(k)})_j = \max_s z_{j,s}$$

  - $\boldsymbol{a}^{(k-1)}$ is linearly mapped to multiple groups of $z_{j,:}^{(k)}$'s
- Learns a piecewise linear, convex activation function automatically
  - Covers both leaky ReLU and PReLU



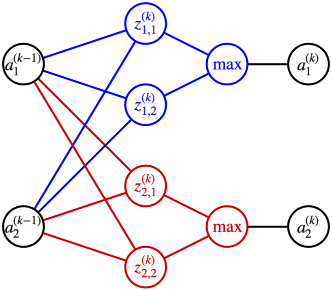


Maxout Units II

- How to train an NN with maxout units?

Maxout Units II

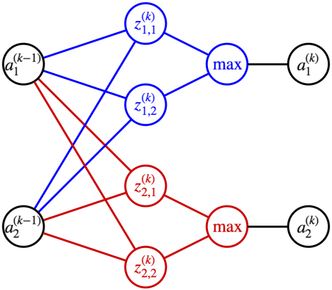
- How to train an NN with maxout units?
- Given a training example $(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})$, update the weights that corresponds to the ***winning*** $z_{j,s}^{(k)}$'s for this example



Maxout Units II

- How to train an NN with maxout units?

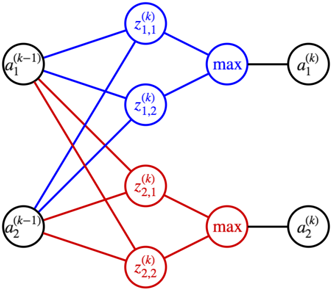
- Given a training example $(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})$, update the weights that corresponds to the ***winning*** $z_{j,s}^{(k)}$'s for this example
- Different examples may update different parts of the network



Maxout Units II

- How to train an NN with maxout units?

- Given a training example $(\boldsymbol{x}^{(n)}, \boldsymbol{y}^{(n)})$, update the weights that corresponds to the ***winning*** $z_{j,s}^{(k)}$'s for this example
- Different examples may update different parts of the network



- Offers some "redundancy" that helps to resist the ***catastrophic forgetting*** phenomenon [2]
  - An NN may forget how to perform tasks that they were trained on in the past

Maxout Units III

- Cons?

Maxout Units III

- Cons?
- Each maxout unit is now parametrized by multiple weight vectors instead of just one

Maxout Units III

- Cons?
- Each maxout unit is now parametrized by multiple weight vectors instead of just one
- Typically requires more training data
- Otherwise, regularization is needed

Outline

1. The Basics
   - Example: Learning the XOR
2. Training
   - Back Propagation
3. Neuron Design
   - Cost Function & Output Neurons
   - Hidden Neurons
4. **Architecture Design**
   - Architecture Tuning

Architecture Design

- Thin-and-deep or fat-and-shallow?

Architecture Design

- Thin-and-deep or fat-and-shallow?

Theorem (Universal Approximation Theorem [4, 5])

A feedforward network with at least one hidden layer can approximate any continuous function (on a closed and bounded subset of $\mathbb{R}^D$) or any function mapping from a finite dimensional discrete space to another.

- In short, a feedforward network with a single layer is sufficient to represent any function

Architecture Design

- Thin-and-deep or fat-and-shallow?

Theorem (Universal Approximation Theorem [4, 5])

A feedforward network with at least one hidden layer can approximate any continuous function (on a closed and bounded subset of $\mathbb{R}^D$) or any function mapping from a finite dimensional discrete space to another.

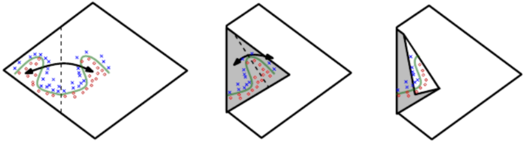
- In short, a feedforward network with a single layer is sufficient to represent any function
- Why going deep?

Exponential Gain in Number of Hidden Units

- Functions representable with a deep rectifier NN require an exponential number of hidden units in a shallow NN [6]

Exponential Gain in Number of Hidden Units

- Functions representable with a deep rectifier NN require an exponential number of hidden units in a shallow NN [6]
- Example: an NN with absolute value rectification units



- Each hidden unit specifies where to fold the input space in order to create mirror responses (on both sides of the absolute value)
- By composing these folding operations, we obtain an exponentially large number of piecewise linear regions which can capture all kinds of regular (e.g., repeating) patterns

Deep ReLU Networks

- Activation-constant regions vs. output values [3]

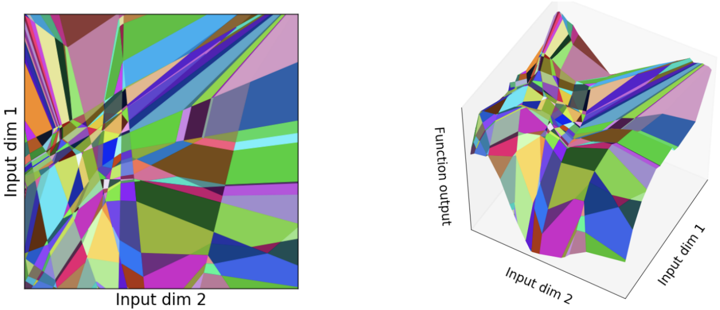


Figure 2: Function defined by a ReLU network of depth 5 and width 8 at initialization. Left: Partition of the input space into regions, on each of which the activation pattern of neurons is constant. Right: the function computed by the network, which is linear on each activation region.

Architecture as Prior Knowledge

- Choosing a deep model also encodes a very general belief that the function we want to learn should involve composition of several simpler functions

Architecture as Prior Knowledge

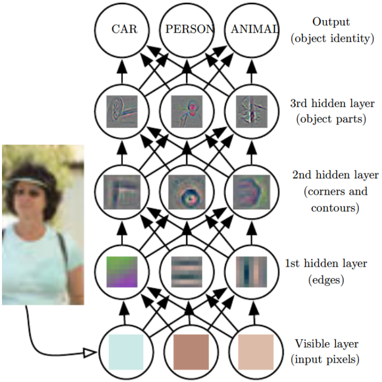
- Choosing a deep model also encodes a very general belief that the function we want to learn should involve composition of several simpler functions
  - If valid, deep NNs give ***better generalizability***

Architecture as Prior Knowledge

- Choosing a deep model also encodes a very general belief that the function we want to learn should involve composition of several simpler functions
  - If valid, deep NNs give ***better generalizability***
- When is the assumption valid?

Architecture as Prior Knowledge

- Choosing a deep model also encodes a very general belief that the function we want to learn should involve composition of several simpler functions
  - If valid, deep NNs give ***better generalizability***
- When is the assumption valid? E.g., image recognition, natural language processing, etc.



Outline

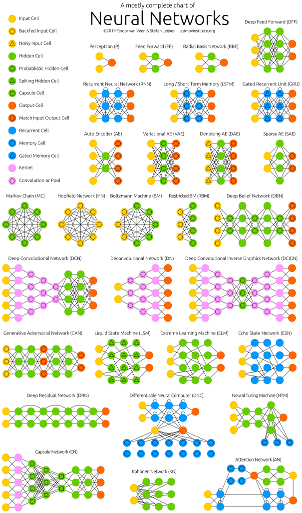
1. The Basics
   - Example: Learning the XOR
2. Training
   - Back Propagation
3. Neuron Design
   - Cost Function & Output Neurons
   - Hidden Neurons
4. **Architecture Design**
   - **Architecture Tuning**

Hyperparameters

- Width & depth
  - Can be determined via cross validation

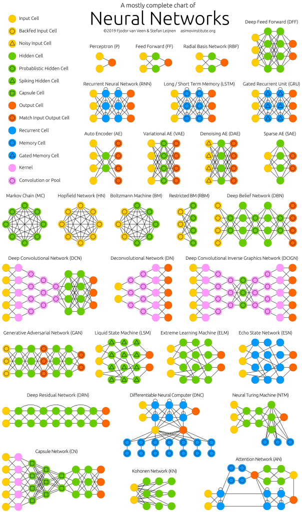
Hyperparameters

- Width & depth
  - Can be determined via cross validation
- Types of neurons & their wiring
  - Usually model a domain-specific prior
  - Validated via ablation study



Hyperparameters

- Width & depth
  - Can be determined via cross validation
- Types of neurons & their wiring
  - Usually model a domain-specific prior
  - Validated via ablation study
- Auto ML
  - The process of automating the above



Reference I

[1] Léon Bottou.
Large-scale machine learning with stochastic gradient descent.
In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[2] Ian J Goodfellow, Mehdi Mirza, Da Xiao, Aaron Courville, and Yoshua Bengio.
An empirical investigation of catastrophic forgetting in gradient-based neural networks.
arXiv preprint arXiv:1312.6211, 2013.

[3] Boris Hanin and David Rolnick.
Deep relu networks have surprisingly few activation patterns.
Advances in neural information processing systems, 32, 2019.

[4] Kurt Hornik, Maxwell Stinchcombe, and Halbert White.
Multilayer feedforward networks are universal approximators.
Neural networks, 2(5):359–366, 1989.

Reference II

[5] Moshe Leshno, Vladimir Ya Lin, Allan Pinkus, and Shimon Schocken.
Multilayer feedforward networks with a nonpolynomial activation function can approximate any function.
Neural networks, 6(6):861–867, 1993.

[6] Guido F Montufar, Razvan Pascanu, Kyunghyun Cho, and Yoshua Bengio.
On the number of linear regions of deep neural networks.
In *Advances in neural information processing systems*, pages 2924–2932, 2014.