Deep Learning Book

Chapter 6
Deep Feedforward Networks

Botian Shi
botianshi@bit.edu.cn

March 7, 2017

You can download the LaTeX source code of this file from Here.

Feedforward Networks

- A type of neural network
  - *Deep feedforward network*
  - *Feedforward neural network*
  - *Multilayer perceptron (MLP)*
- For a classifier, $y = f^*(x)$ maps an input x to category y
- Defines a mapping:

$$y = f(x; \theta)$$

- Learns the best approximation of f^* with parameter θ
- Feedforward only, no feedback connections.
- The basis of many applications.

Feedforward Networks

- A type of neural network
  - *Deep feedforward network*
  - *Feedforward neural network*
  - *Multilayer perceptron (MLP)*
- For a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to category $\boldsymbol{y}$
- Defines a mapping:

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$$

- Learns the best approximation of f^* with parameter $\boldsymbol{\theta}$
- Feedforward only, no feedback connections.
- The basis of many applications.

Feedforward Networks

- A type of neural network
  - *Deep feedforward network*
  - *Feedforward neural network*
  - *Multilayer perceptron (MLP)*
- For a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to category $\boldsymbol{y}$
- Defines a mapping:

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$$

- Learns the best approximation of f^* with parameter $\boldsymbol{\theta}$
- Feedforward only, no feedback connections.
- The basis of many applications.

Feedforward Networks

- A type of neural network
  - *Deep feedforward network*
  - *Feedforward neural network*
  - *Multilayer perceptron (MLP)*
- For a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to category $\boldsymbol{y}$
- Defines a mapping:

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$$

- Learns the best approximation of f^* with parameter $\boldsymbol{\theta}$
- Feedforward only, no feedback connections.
- The basis of many applications.

Feedforward Networks

- A type of neural network
  - *Deep feedforward network*
  - *Feedforward neural network*
  - *Multilayer perceptron (MLP)*
- For a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to category $\boldsymbol{y}$
- Defines a mapping:

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$$

- Learns the best approximation of f^* with parameter $\boldsymbol{\theta}$
- Feedforward only, no feedback connections.
- The basis of many applications.

Feedforward networks are ...

1. Extreme important.
2. Stepping stone on the path to recurrent neural networks.

Example: convolutional neural network

Figure 1: A type of convolutional neural network: LeNet-5 (LeCun et al. [1998])

Feedforward networks are ...

1. Extreme important.
2. Stepping stone on the path to recurrent neural networks.

Example: convolutional neural network

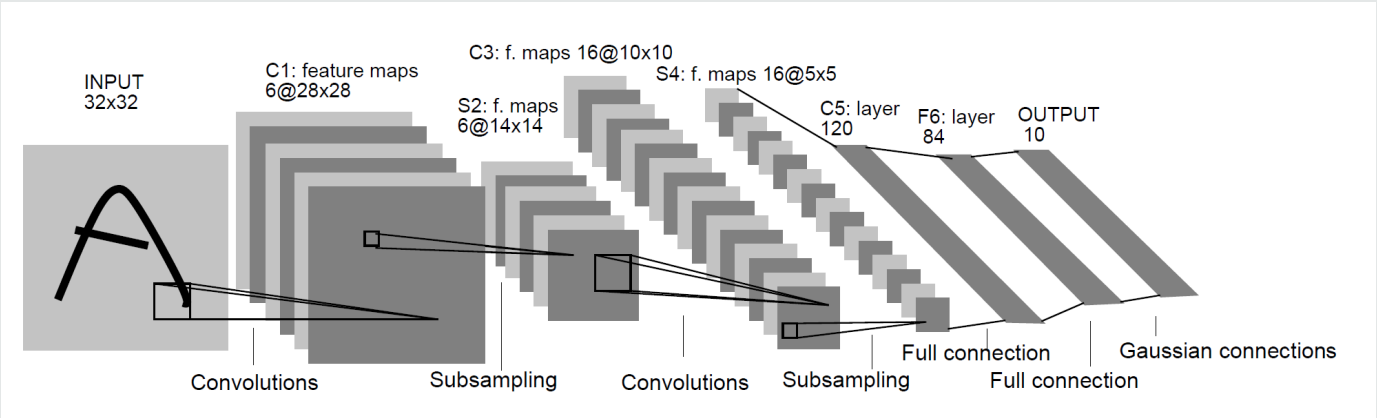


Figure 1: A type of convolutional neural network: LeNet-5 (LeCun et al. [1998])

Multilayer Perceptron

- Why called *networks*?
  - Composing together many different functions.
  - Model is associated with a DAG describing the composition:

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

  - $f^{(1)}$ called first layer of the network
  - $f^{(2)}$ called second layer, and so on
  - $f^{(2)}$ called output layer

Figure 2:
An example of MLP (from UFLDL)

- During training, we drive $f(x)$ to match $f^*(x)$
- Each example x is accompanied by a label$y \approx f^*(x)$
- At each point x, network must produce a value that is close to y.
- The learning algorithm must decide how to use those layers to produce the desired output.
- Layers between input and output layer are called *hidden layer*.

Multilayer Perceptron

- Why called *networks*?
  - Composing together many different functions.
  - Model is associated with a DAG describing the composition:

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

  - $f^{(1)}$ called first layer of the network
  - $f^{(2)}$ called second layer, and so on
  - $f^{(2)}$ called output layer

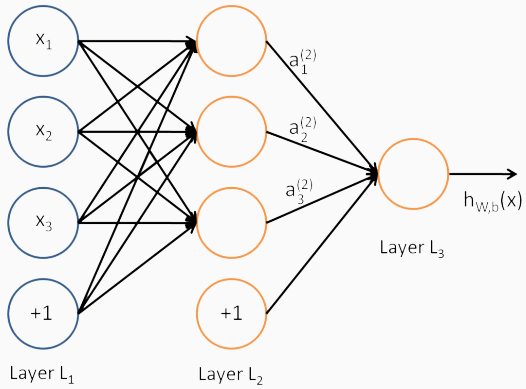


Figure 2:
An example of MLP (from UFLDL)

- During training, we drive $f(x)$ to match $f^*(x)$
- Each example x is accompanied by a label $y \approx f^*(x)$
- At each point x, network must produce a value that is close to y.
- The learning algorithm must decide how to use those layers to produce the desired output.
- Layers between input and output layer are called *hidden layer*.

Multilayer Perceptron

- Why called *networks*?
  - Composing together many different functions.
  - Model is associated with a DAG describing the composition:

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

  - $f^{(1)}$ called first layer of the network
  - $f^{(2)}$ called second layer, and so on
  - $f^{(2)}$ called output layer

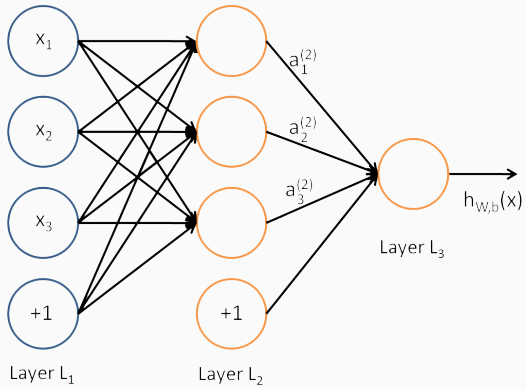


Figure 2:
An example of MLP (from UFLDL)

- During training, we drive $f(x)$ to match $f^*(x)$
- Each example x is accompanied by a label $y \approx f^*(x)$
- At each point x, network must produce a value that is close to y.
- The learning algorithm must decide how to use those layers to produce the desired output.
- Layers between input and output layer are called *hidden layer*.

Multilayer Perceptron

- Why called *networks*?
  - Composing together many different functions.
  - Model is associated with a DAG describing the composition:

$$f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}\left(x\right)\right)\right)$$

  - $f^{(1)}$ called first layer of the network
  - $f^{(2)}$ called second layer, and so on
  - $f^{(2)}$ called output layer

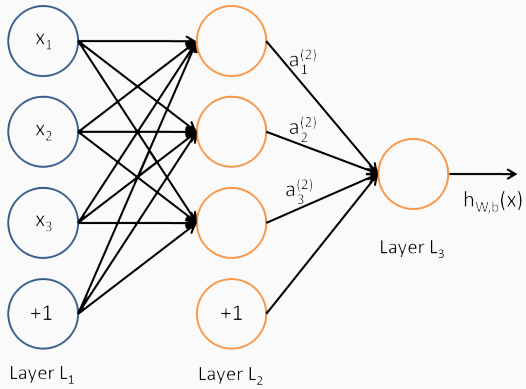


Figure 2:
An example of MLP (from UFLDL)

- During training, we drive $f(x)$ to match $f^*(x)$
- Each example x is accompanied by a label $y \approx f^*(x)$
- At each point x, network must produce a value that is close to y.
- The learning algorithm must decide how to use those layers to produce the desired output.
- Layers between input and output layer are called *hidden layer*.

Multilayer Perceptron

- Why called *networks*?
  - Composing together many different functions.
  - Model is associated with a DAG describing the composition:

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

  - $f^{(1)}$ called first layer of the network
  - $f^{(2)}$ called second layer, and so on
  - $f^{(2)}$ called output layer

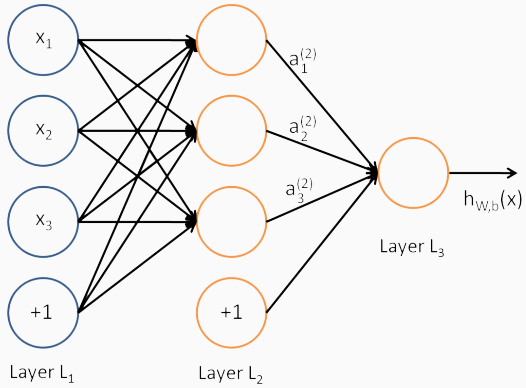


Figure 2:
An example of MLP (from UFLDL)

- During training, we drive $f(x)$ to match $f^*(x)$
- Each example x is accompanied by a label $y \approx f^*(x)$
- At each point x, network must produce a value that is close to y.
- The learning algorithm must decide how to use those layers to produce the desired output.
- Layers between input and output layer are called *hidden layer*.

Multilayer Perceptron

- Why called *networks*?
  - Composing together many different functions.
  - Model is associated with a DAG describing the composition:

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

  - $f^{(1)}$ called first layer of the network
  - $f^{(2)}$ called second layer, and so on
  - $f^{(2)}$ called output layer

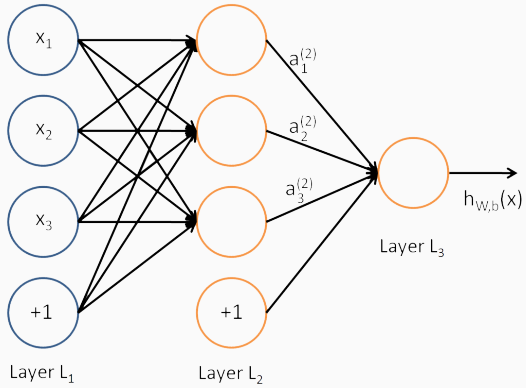


Figure 2:
An example of MLP (from UFLDL)

- During training, we drive $f(x)$ to match $f^*(x)$
- Each example x is accompanied by a label $y \approx f^*(x)$
- At each point x, network must produce a value that is close to y.
- The learning algorithm must decide how to use those layers to produce the desired output.
- Layers between input and output layer are called *hidden layer*.

Multilayer Perceptron

- Why called *networks*?
  - Composing together many different functions.
  - Model is associated with a DAG describing the composition:

$$f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}\left(x\right)\right)\right)$$

  - $f^{(1)}$ called first layer of the network
  - $f^{(2)}$ called second layer, and so on
  - $f^{(2)}$ called output layer

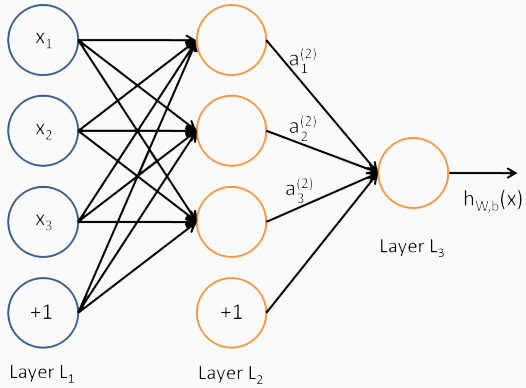


Figure 2:
An example of MLP (from UFLDL)

- During training, we drive $f(x)$ to match $f^*(x)$
- Each example x is accompanied by a label$y \approx f^*(x)$
- At each point x, network must produce a value that is close to y.
- The learning algorithm must decide how to use those layers to produce the desired output.
- Layers between input and output layer are called *hidden layer*.

Multilayer Perceptron

- Why called *networks*?
  - Composing together many different functions.
  - Model is associated with a DAG describing the composition:

$$f(x) = f^{(3)}\left(f^{(2)}\left(f^{(1)}\left(x\right)\right)\right)$$

  - $f^{(1)}$ called first layer of the network
  - $f^{(2)}$ called second layer, and so on
  - $f^{(2)}$ called output layer

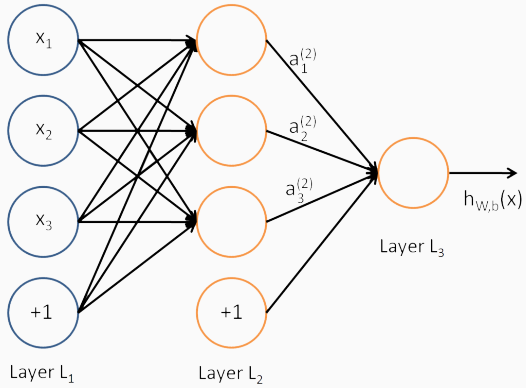


Figure 2:
An example of MLP (from UFLDL)

- During training, we drive $f(x)$ to match $f^*(x)$
- Each example x is accompanied by a label$y \approx f^*(x)$
- At each point x, network must produce a value that is close to y.
- The learning algorithm must decide how to use those layers to produce the desired output.
- Layers between input and output layer are called *hidden layer*.

Multilayer Perceptron

- Why called *networks*?
  - Composing together many different functions.
  - Model is associated with a DAG describing the composition:

$$f(x) = f^{(3)}(f^{(2)}(f^{(1)}(x)))$$

  - $f^{(1)}$ called first layer of the network
  - $f^{(2)}$ called second layer, and so on
  - $f^{(2)}$ called output layer

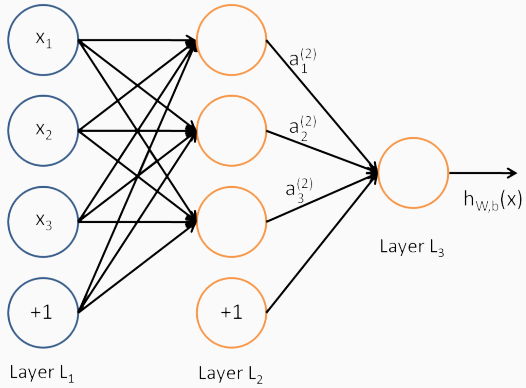


Figure 2:
An example of MLP (from UFLDL)

- During training, we drive $f(x)$ to match $f^*(x)$
- Each example x is accompanied by a label$y \approx f^*(x)$
- At each point x, network must produce a value that is close to y.
- The learning algorithm must decide how to use those layers to produce the desired output.
- Layers between input and output layer are called *hidden layer*.

The *NEURAL* network

- Inspired by neuroscience.

Figure 3: Neuron (from UFLDL)

Figure 4: MLP (from UFLDL)

- Each hidden layer is vector-valued. The dimensionality of these hidden layers determines the *width* of the model.
- Each element of the vector may be interpreted as a neuron.
- Each unit resembles a neuron that receives input from many other units and computes its own activation value.
- Layer consist of many *units* that act in parallel, each representing a vector-to-scalar function.

The *NEURAL* network

- Inspired by neuroscience.

Figure 3: Neuron (from UFLDL)

Figure 4: MLP (from UFLDL)

- Each hidden layer is vector-valued. The dimensionality of these hidden layers determines the *width* of the model.
- Each element of the vector may be interpreted as a neuron.
- Each unit resembles a neuron that receives input from many other units and computes its own activation value.
- Layer consist of many *units* that act in parallel, each representing a vector-to-scalar function.

The *NEURAL* network

- Inspired by neuroscience.

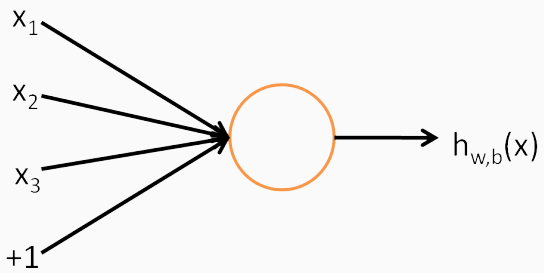


Figure 3: Neuron (from UFLDL)

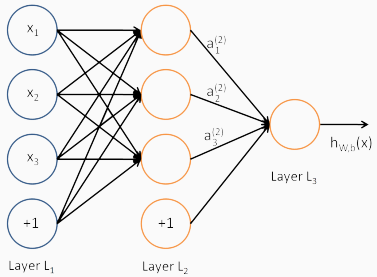


Figure 4: MLP (from UFLDL)

- Each hidden layer is vector-valued. The dimensionality of these hidden layers determines the *width* of the model.
- Each element of the vector may be interpreted as a neuron.
- Each unit resembles a neuron that receives input from many other units and computes its own activation value.
- Layer consist of many *units* that act in parallel, each representing a vector-to-scalar function.

The *NEURAL* network

- Inspired by neuroscience.

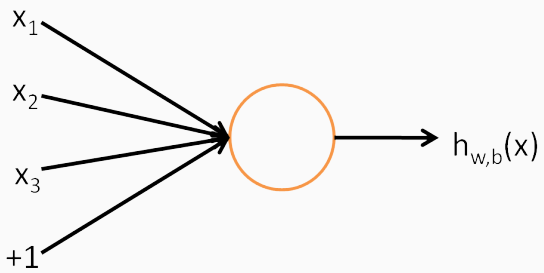


Figure 3: Neuron (from UFLDL)

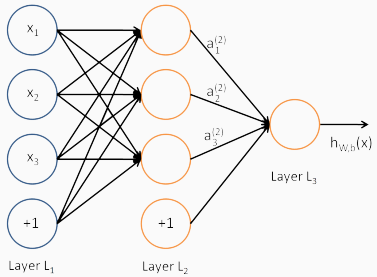


Figure 4: MLP (from UFLDL)

- Each hidden layer is vector-valued. The dimensionality of these hidden layers determines the *width* of the model.
- Each element of the vector may be interpreted as a neuron.
- Each unit resembles a neuron that receives input from many other units and computes its own activation value.
- Layer consist of many *units* that act in parallel, each representing a vector-to-scalar function.

The *NEURAL* network

- Inspired by neuroscience.

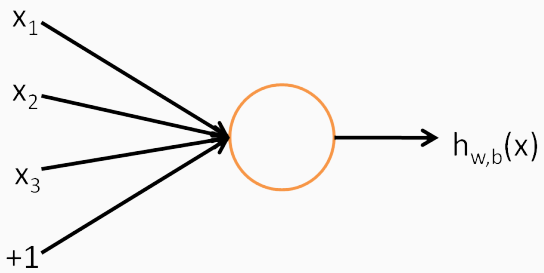


Figure 3: Neuron (from UFLDL)

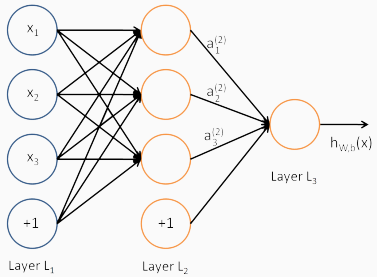


Figure 4: MLP (from UFLDL)

- Each hidden layer is vector-valued. The dimensionality of these hidden layers determines the *width* of the model.
- Each element of the vector may be interpreted as a neuron.
- Each unit resembles a neuron that receives input from many other units and computes its own activation value.
- Layer consist of many *units* that act in parallel, each representing a vector-to-scalar function.

The *NEURAL* network

- Inspired by neuroscience.

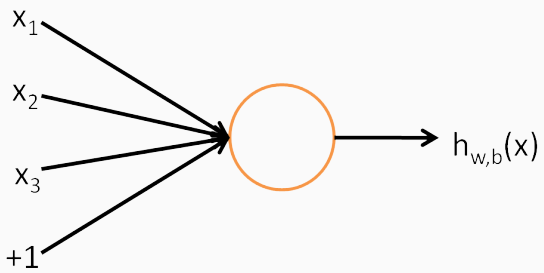


Figure 3: Neuron (from UFLDL)

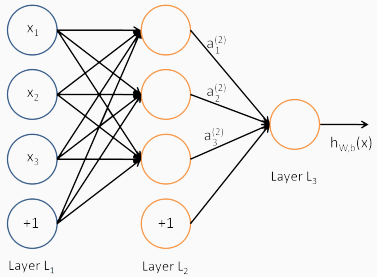


Figure 4: MLP (from UFLDL)

- Each hidden layer is vector-valued. The dimensionality of these hidden layers determines the *width* of the model.
- Each element of the vector may be interpreted as a neuron.
- Each unit resembles a neuron that receives input from many other units and computes its own activation value.
- Layer consist of many *units* that act in parallel, each representing a vector-to-scalar function.

The *NEURAL* network

- Inspired by neuroscience.

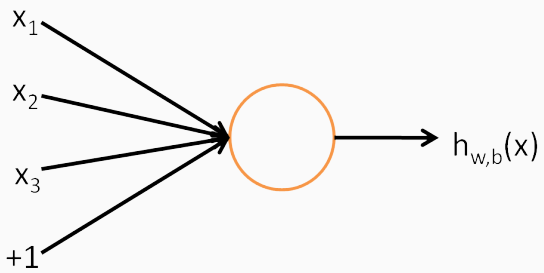


Figure 3: Neuron (from UFLDL)

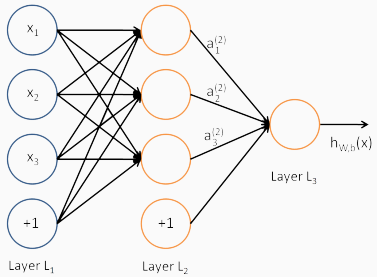


Figure 4: MLP (from UFLDL)

- Each hidden layer is vector-valued. The dimensionality of these hidden layers determines the *width* of the model.
- Each element of the vector may be interpreted as a neuron.
- Each unit resembles a neuron that receives input from many other units and computes its own activation value.
- Layer consist of many *units* that act in parallel, each representing a vector-to-scalar function.

The *NEURAL* network

- The idea of using many layers of vector-valued representation is drawn from neuroscience.
- Modern neural network research $\neq$ perfectly model the brain.
- Feedforward networks $\approx$ function approximation machines.
- Inspired by brain, rather than model a brain.

The *NEURAL* network

- The idea of using many layers of vector-valued representation is drawn from neuroscience.
- Modern neural network research $\neq$ perfectly model the brain.
- Feedforward networks $\approx$ function approximation machines.
- Inspired by brain, rather than model a brain.

The *NEURAL* network

- The idea of using many layers of vector-valued representation is drawn from neuroscience.
- Modern neural network research $\neq$ perfectly model the brain.
- Feedforward networks $\approx$ function approximation machines.
- Inspired by brain, rather than model a brain.

The *NEURAL* network

- The idea of using many layers of vector-valued representation is drawn from neuroscience.
- Modern neural network research $\neq$ perfectly model the brain.
- Feedforward networks $\approx$ function approximation machines.
- Inspired by brain, rather than model a brain.

The *NEURAL* network

- The idea of using many layers of vector-valued representation is drawn from neuroscience.
- Modern neural network research $\neq$ perfectly model the brain.
- Feedforward networks $\approx$ function approximation machines.
- Inspired by brain, rather than model a brain.

Understand feedforward networks

- Linear Models

Linear Regression

$$h_\theta(x;\theta) = \theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

$$\theta = \arg\min_\theta \mathrm{J}(\theta; X) = \arg\min_\theta \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Logistic Regression

$$h_\theta(x;\theta) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$$

$$g(z) = \frac{1}{1+e^{-z}}$$

General Linear Model

$$p(y;\eta) = b(u)\exp(\eta^T T(y) - a(\eta))$$

Understand feedforward networks

- Linear Models

Linear Regression

$$h_\theta(x;\theta) = \theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

$$\theta = \arg\min_\theta \mathrm{J}(\theta; X) = \arg\min_\theta \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Logistic Regression

$$h_\theta(x;\theta) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$$

$$g(z) = \frac{1}{1+e^{-z}}$$

General Linear Model

$$p(y;\eta) = b(u)\exp(\eta^T T(y) - a(\eta))$$

Understand feedforward networks

- Linear Models

Linear Regression

$$h_\theta(x;\theta) = \theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

$$\theta = \arg\min_\theta J(\theta; X) = \arg\min_\theta \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Logistic Regression

$$h_\theta(x;\theta) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

General Linear Model

$$p(y;\eta) = b(u) \exp(\eta^T T(y) - a(\eta))$$

Understand feedforward networks

- Linear Models

Linear Regression

$$h_\theta(x;\theta) = \theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

$$\theta = \arg\min_\theta J(\theta; X) = \arg\min_\theta \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Logistic Regression

$$h_\theta(x;\theta) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$$

$$g(z) = \frac{1}{1+e^{-z}}$$

General Linear Model

$$p(y;\eta) = b(u)\exp(\eta^T T(y) - a(\eta))$$

Understand feedforward networks

- Linear Models

Linear Regression

$$h_\theta(x;\theta) = \theta^T x = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

$$\theta = \arg\min_\theta J(\theta; X) = \arg\min_\theta \frac{1}{2} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$

Logistic Regression

$$h_\theta(x;\theta) = g(\theta^T x) = \frac{1}{1 + e^{-\theta^T x}}$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

General Linear Model

$$p(y;\eta) = b(u) \exp(\eta^T T(y) - a(\eta))$$

Understand feedforward networks

- Advantage and disadvantage
  - 🙂 Efficiently and reliable
  - 😕 The model capacity is limited to linear functions: The model cannot understand the interaction between any two input variables.
- To extend linear models to represent nonlinear functions of $\boldsymbol{x}$, we can apply the linear model not to $\boldsymbol{x}$ itself but to a transformed input $\phi(\boldsymbol{x})$, where ϕ is a *nonlinear transformation*.
- ϕ provide a set of features describing $\boldsymbol{x}$, or provide a new representation fo $\boldsymbol{x}$

Understand feedforward networks

- Advantage and disadvantage
  - 🙂 Efficiently and reliable
  - 😕 The model capacity is limited to linear functions: The model cannot understand the interaction between any two input variables.
- To extend linear models to represent nonlinear functions of $\boldsymbol{x}$, we can apply the linear model not to $\boldsymbol{x}$ itself but to a transformed input $\phi(\boldsymbol{x})$, where ϕ is a *nonlinear transformation*.
- ϕ provide a set of features describing $\boldsymbol{x}$, or provide a new representation fo $\boldsymbol{x}$

Understand feedforward networks

- Advantage and disadvantage
  - 🙂 Efficiently and reliable
  - 😕 The model capacity is limited to linear functions: The model cannot understand the interaction between any two input variables.
- To extend linear models to represent nonlinear functions of $\boldsymbol{x}$, we can apply the linear model not to $\boldsymbol{x}$ itself but to a transformed input $\phi(\boldsymbol{x})$, where ϕ is a *nonlinear transformation*.
- ϕ provide a set of features describing $\boldsymbol{x}$, or provide a new representation fo $\boldsymbol{x}$

Understand feedforward networks

- How to choose ϕ?
  1. Use a very generic ϕ, e.g., infinite-dimensional ϕ.
     - 🙂 High dimension $\Leftrightarrow$ enough capacity to fit the training set.
     - 🙁 High dimension $\Leftrightarrow$ poor generalization capacity.
     - 🙁 No-free-lunch: Runge phenomenon; Gibbs phenomenon.
  2. Manually engineer ϕ.
     - Require human effort for each separate task.
     - Need practitioners specializing in different domains.
  3. Deep learning.
     - Strategy: learn a ϕ automatically.
     - $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$
     - We have parameters $\boldsymbol{\theta}$ to learn ϕ from a broad class of functions.
     - We have parameters $\boldsymbol{w}$ to map from $\phi(\boldsymbol{x})$ to the desired output.
     - Here in example, ϕ defining a hidden layer.
     - Deep learning is not simply a "*deep*" neural network. The ϕ is crucial!

Understand feedforward networks

- How to choose ϕ?
  1. Use a very generic ϕ, e.g., infinite-dimensional ϕ.
     - 🙂 High dimension $\Leftrightarrow$ enough capacity to fit the training set.
     - 🙁 High dimension $\Leftrightarrow$ poor generalization capacity.
     - 🙁 No-free-lunch: Runge phenomenon; Gibbs phenomenon.
  2. Manually engineer ϕ.
     - Require human effort for each separate task.
     - Need practitioners specializing in different domains.
  3. Deep learning.
     - Strategy: learn a ϕ automatically.
     - $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$
     - We have parameters $\boldsymbol{\theta}$ to learn ϕ from a broad class of functions.
     - We have parameters $\boldsymbol{w}$ to map from $\phi(\boldsymbol{x})$ to the desired output.
     - Here in example, ϕ defining a hidden layer.
     - Deep learning is not simply a "*deep*" neural network. The ϕ is crucial!

Understand feedforward networks

- How to choose ϕ?
  1. Use a very generic ϕ, e.g., infinite-dimensional ϕ.
     - 🙂 High dimension $\Leftrightarrow$ enough capacity to fit the training set.
     - 🙁 High dimension $\Leftrightarrow$ poor generalization capacity.
     - 🙁 No-free-lunch: Runge phenomenon; Gibbs phenomenon.
  2. Manually engineer ϕ.
     - Require human effort for each separate task.
     - Need practitioners specializing in different domains.
  3. Deep learning.
     - Strategy: learn a ϕ automatically.
     - $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$
     - We have parameters $\boldsymbol{\theta}$ to learn ϕ from a broad class of functions.
     - We have parameters $\boldsymbol{w}$ to map from $\phi(\boldsymbol{x})$ to the desired output.
     - Here in example, ϕ defining a hidden layer.
     - Deep learning is not simply a "*deep*" neural network. The ϕ is crucial!

Understand feedforward networks

- How to choose ϕ?
  1. Use a very generic ϕ, e.g., infinite-dimensional ϕ.
     - 🙂 High dimension $\Leftrightarrow$ enough capacity to fit the training set.
     - 🙁 High dimension $\Leftrightarrow$ poor generalization capacity.
     - 🙁 No-free-lunch: Runge phenomenon; Gibbs phenomenon.
  2. Manually engineer ϕ.
     - Require human effort for each separate task.
     - Need practitioners specializing in different domains.
  3. Deep learning.
     - Strategy: learn a ϕ automatically.
     - $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$
     - We have parameters $\boldsymbol{\theta}$ to learn ϕ from a broad class of functions.
     - We have parameters $\boldsymbol{w}$ to map from $\phi(\boldsymbol{x})$ to the desired output.
     - Here in example, ϕ defining a hidden layer.
     - Deep learning is not simply a "*deep*" neural network. The ϕ is crucial!

Understand feedforward networks

- How to choose ϕ?
  1. Use a very generic ϕ, e.g., infinite-dimensional ϕ.
     - 🙂 High dimension $\Leftrightarrow$ enough capacity to fit the training set.
     - 🙁 High dimension $\Leftrightarrow$ poor generalization capacity.
     - 🙁 No-free-lunch: Runge phenomenon; Gibbs phenomenon.
  2. Manually engineer ϕ.
     - Require human effort for each separate task.
     - Need practitioners specializing in different domains.
  3. Deep learning.
     - Strategy: learn a ϕ automatically.
     - $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$
     - We have parameters $\boldsymbol{\theta}$ to learn ϕ from a broad class of functions.
     - We have parameters $\boldsymbol{w}$ to map from $\phi(\boldsymbol{x})$ to the desired output.
     - Here in example, ϕ defining a hidden layer.
     - Deep learning is not simply a "*deep*" neural network. The ϕ is crucial!

Understand feedforward networks

- How to choose ϕ?
  1. Use a very generic ϕ, e.g., infinite-dimensional ϕ.
     - 🙂 High dimension $\Leftrightarrow$ enough capacity to fit the training set.
     - 🙁 High dimension $\Leftrightarrow$ poor generalization capacity.
     - 🙁 No-free-lunch: Runge phenomenon; Gibbs phenomenon.
  2. Manually engineer ϕ.
     - Require human effort for each separate task.
     - Need practitioners specializing in different domains.
  3. Deep learning.
     - Strategy: learn a ϕ automatically.
     - $y = f(\boldsymbol{x}; \boldsymbol{\theta}, \boldsymbol{w}) = \phi(\boldsymbol{x}; \boldsymbol{\theta})^T \boldsymbol{w}$
     - We have parameters $\boldsymbol{\theta}$ to learn ϕ from a broad class of functions.
     - We have parameters $\boldsymbol{w}$ to map from $\phi(\boldsymbol{x})$ to the desired output.
     - Here in example, ϕ defining a hidden layer.
     - Deep learning is not simply a "*deep*" neural network. The ϕ is crucial!

Understand feedforward networks

- Why deep learning?
  1. Parametrize the representation as $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
  2. We can use optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation.
  3. Capture the benefit of first and second approach.
     - Being highly generic: using a very broad family $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
     - Human practitioners can encode their knowledge by designing families $\phi(\boldsymbol{x}; \boldsymbol{\theta})$.
     - Human designer only needs to find the right general function family rather than finding precisely the right function.
- The general principle of deep learning is to improve models by **learning feature representation**.
- Feedforward networks are the application of this principle to learning deterministic mappings from $\boldsymbol{x}$ to $\boldsymbol{y}$ that lack feedback connections.

Understand feedforward networks

- Why deep learning?
  1. Parametrize the representation as $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
  2. We can use optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation.
  3. Capture the benefit of first and second approach.
     - Being highly generic: using a very broad family $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
     - Human practitioners can encode their knowledge by designing families $\phi(\boldsymbol{x}; \boldsymbol{\theta})$.
     - Human designer only needs to find the right general function family rather than finding precisely the right function.
- The general principle of deep learning is to improve models by **learning feature representation**.
- Feedforward networks are the application of this principle to learning deterministic mappings from $\boldsymbol{x}$ to $\boldsymbol{y}$ that lack feedback connections.

Understand feedforward networks

- Why deep learning?
  1. Parametrize the representation as $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
  2. We can use optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation.
  3. Capture the benefit of first and second approach.
     - Being highly generic: using a very broad family $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
     - Human practitioners can encode their knowledge by designing families $\phi(\boldsymbol{x}; \boldsymbol{\theta})$.
     - Human designer only needs to find the right general function family rather than finding precisely the right function.
- The general principle of deep learning is to improve models by **learning feature representation**.
- Feedforward networks are the application of this principle to learning deterministic mappings from $\boldsymbol{x}$ to $\boldsymbol{y}$ that lack feedback connections.

Understand feedforward networks

- Why deep learning?
  1. Parametrize the representation as $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
  2. We can use optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation.
  3. Capture the benefit of first and second approach.
     - Being highly generic: using a very broad family $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
     - Human practitioners can encode their knowledge by designing families $\phi(\boldsymbol{x}; \boldsymbol{\theta})$.
     - Human designer only needs to find the right general function family rather than finding precisely the right function.
- The general principle of deep learning is to improve models by **learning feature representation**.
- Feedforward networks are the application of this principle to learning deterministic mappings from $\boldsymbol{x}$ to $\boldsymbol{y}$ that lack feedback connections.

Understand feedforward networks

- Why deep learning?
  1. Parametrize the representation as $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
  2. We can use optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation.
  3. Capture the benefit of first and second approach.
     - Being highly generic: using a very broad family $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
     - Human practitioners can encode their knowledge by designing families $\phi(\boldsymbol{x}; \boldsymbol{\theta})$.
     - Human designer only needs to find the right general function family rather than finding precisely the right function.
- The general principle of deep learning is to improve models by **learning feature representation**.
- Feedforward networks are the application of this principle to learning deterministic mappings from $\boldsymbol{x}$ to $\boldsymbol{y}$ that lack feedback connections.

Understand feedforward networks

- Why deep learning?
  1. Parametrize the representation as $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
  2. We can use optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation.
  3. Capture the benefit of first and second approach.
     - Being highly generic: using a very broad family $\phi(\boldsymbol{x}; \boldsymbol{\theta})$
     - Human practitioners can encode their knowledge by designing families $\phi(\boldsymbol{x}; \boldsymbol{\theta})$.
     - Human designer only needs to find the right general function family rather than finding precisely the right function.
- The general principle of deep learning is to improve models by **learning feature representation**.
- Feedforward networks are the application of this principle to learning deterministic mappings from $\boldsymbol{x}$ to $\boldsymbol{y}$ that lack feedback connections.

Deploy a feedforward network

Training a feedforward network requires the same design decisions for linear model:

- Define the form of input and output units.
- Design the architecture of the network
  1. How many layers the network should contain.
  2. How these networks should be connected to each other.
  3. How many units should be in each layer.
- Choose activation functions for hidden layers.
- Define a cost function
- Choose an optimizer (gradient descent algorithm and its modern generalizations) to train the network.
- Use back-propagation algorithm to compute the gradients of complicated functions.

Deploy a feedforward network

Training a feedforward network requires the same design decisions for linear model:

- Define the form of input and output units.
- Design the architecture of the network
  1. How many layers the network should contain.
  2. How these networks should be connected to each other.
  3. How many units should be in each layer.
- Choose activation functions for hidden layers.
- Define a cost function
- Choose an optimizer (gradient descent algorithm and its modern generalizations) to train the network.
- Use back-propagation algorithm to compute the gradients of complicated functions.

Deploy a feedforward network

Training a feedforward network requires the same design decisions for linear model:

- Define the form of input and output units.
- Design the architecture of the network
  1. How many layers the network should contain.
  2. How these networks should be connected to each other.
  3. How many units should be in each layer.
- Choose activation functions for hidden layers.
- Define a cost function
- Choose an optimizer (gradient descent algorithm and its modern generalizations) to train the network.
- Use back-propagation algorithm to compute the gradients of complicated functions.

Deploy a feedforward network

Training a feedforward network requires the same design decisions for linear model:

- Define the form of input and output units.
- Design the architecture of the network
  1. How many layers the network should contain.
  2. How these networks should be connected to each other.
  3. How many units should be in each layer.
- Choose activation functions for hidden layers.
- Define a cost function
- Choose an optimizer (gradient descent algorithm and its modern generalizations) to train the network.
- Use back-propagation algorithm to compute the gradients of complicated functions.

Deploy a feedforward network

Training a feedforward network requires the same design decisions for linear model:

- Define the form of input and output units.
- Design the architecture of the network
  1. How many layers the network should contain.
  2. How these networks should be connected to each other.
  3. How many units should be in each layer.
- Choose activation functions for hidden layers.
- Define a cost function
- Choose an optimizer (gradient descent algorithm and its modern generalizations) to train the network.
- Use back-propagation algorithm to compute the gradients of complicated functions.

Deploy a feedforward network

Training a feedforward network requires the same design decisions for linear model:

- Define the form of input and output units.
- Design the architecture of the network
  1. How many layers the network should contain.
  2. How these networks should be connected to each other.
  3. How many units should be in each layer.
- Choose activation functions for hidden layers.
- Define a cost function
- Choose an optimizer (gradient descent algorithm and its modern generalizations) to train the network.
- Use back-propagation algorithm to compute the gradients of complicated functions.

Deploy a feedforward network

Training a feedforward network requires the same design decisions for linear model:

- Define the form of input and output units.
- Design the architecture of the network
  1. How many layers the network should contain.
  2. How these networks should be connected to each other.
  3. How many units should be in each layer.
- Choose activation functions for hidden layers.
- Define a cost function
- Choose an optimizer (gradient descent algorithm and its modern generalizations) to train the network.
- Use back-propagation algorithm to compute the gradients of complicated functions.

Deploy a feedforward network

Training a feedforward network requires the same design decisions for linear model:

- Define the form of input and output units.
- Design the architecture of the network
  1. How many layers the network should contain.
  2. How these networks should be connected to each other.
  3. How many units should be in each layer.
- Choose activation functions for hidden layers.
- Define a cost function
- Choose an optimizer (gradient descent algorithm and its modern generalizations) to train the network.
- Use back-propagation algorithm to compute the gradients of complicated functions.

Deploy a feedforward network

Training a feedforward network requires the same design decisions for linear model:

- Define the form of input and output units.
- Design the architecture of the network
  1. How many layers the network should contain.
  2. How these networks should be connected to each other.
  3. How many units should be in each layer.
- Choose activation functions for hidden layers.
- Define a cost function
- Choose an optimizer (gradient descent algorithm and its modern generalizations) to train the network.
- Use back-propagation algorithm to compute the gradients of complicated functions.

Deploy a feedforward network

Training a feedforward network requires the same design decisions for linear model:

- Define the form of input and output units.
- Design the architecture of the network
  1. How many layers the network should contain.
  2. How these networks should be connected to each other.
  3. How many units should be in each layer.
- Choose activation functions for hidden layers.
- Define a cost function
- Choose an optimizer (gradient descent algorithm and its modern generalizations) to train the network.
- Use back-propagation algorithm to compute the gradients of complicated functions.

Example: Learning XOR

- The XOR function is an operation on two binary values, x_1 and x_2.
- Target: $y = f^*(\boldsymbol{x})$
- Model: $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- Learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*
- Regression problem
- use mean squared error(MSE) loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{x \in \mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}))^2$$

- Suppose our model is:

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T \boldsymbol{w} + b$$

- After solving the equations, we obtain $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}; f(\boldsymbol{x}) = \frac{1}{2}; J(\boldsymbol{\theta}) = \frac{1}{4}$
- We will never achieve 100% accuracy. Why?

Table 1: XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5: XOR problem

Example: Learning XOR

- The XOR function is an operation on two binary values, x_1 and x_2.
- Target: $y = f^*(\boldsymbol{x})$
- Model: $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- Learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*
- Regression problem
- use mean squared error(MSE) loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{x \in \mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}))^2$$

- Suppose our model is:

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T \boldsymbol{w} + b$$

- After solving the equations, we obtain $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}; f(\boldsymbol{x}) = \frac{1}{2}; J(\boldsymbol{\theta}) = \frac{1}{4}$
- We will never achieve 100% accuracy. Why?

Table 1: XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5: XOR problem

Example: Learning XOR

- The XOR function is an operation on two binary values, x_1 and x_2.
- Target: $y = f^*(\boldsymbol{x})$
- Model: $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- Learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*
- Regression problem
- use mean squared error(MSE) loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{x \in \mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}))^2$$

- Suppose our model is:

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T \boldsymbol{w} + b$$

- After solving the equations, we obtain $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}; f(\boldsymbol{x}) = \frac{1}{2}; J(\boldsymbol{\theta}) = \frac{1}{4}$
- We will never achieve 100% accuracy. Why?

Table 1: XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5: XOR problem

Example: Learning XOR

- The XOR function is an operation on two binary values, x_1 and x_2.
- Target: $y = f^*(\boldsymbol{x})$
- Model: $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- Learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*
- Regression problem
- use mean squared error(MSE) loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{x \in \mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}))^2$$

- Suppose our model is:

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T \boldsymbol{w} + b$$

- After solving the equations, we obtain $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}; f(\boldsymbol{x}) = \frac{1}{2}; J(\boldsymbol{\theta}) = \frac{1}{4}$
- We will never achieve 100% accuracy. Why?

Table 1: XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5: XOR problem

Example: Learning XOR

- The XOR function is an operation on two binary values, x_1 and x_2.
- Target: $y = f^*(\boldsymbol{x})$
- Model: $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- Learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*
- Regression problem
- use mean squared error(MSE) loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{x \in \mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}))^2$$

Table 1: XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5: XOR problem

- Suppose our model is:

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T \boldsymbol{w} + b$$

- After solving the equations, we obtain $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}; f(\boldsymbol{x}) = \frac{1}{2}; J(\boldsymbol{\theta}) = \frac{1}{4}$
- We will never achieve 100% accuracy. Why?

Example: Learning XOR

- The XOR function is an operation on two binary values, x_1 and x_2.
- Target: $y = f^*(\boldsymbol{x})$
- Model: $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- Learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*
- Regression problem
- use mean squared error(MSE) loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{x \in \mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}))^2$$

Table 1: XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5: XOR problem

- Suppose our model is:

$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T \boldsymbol{w} + b$$

- After solving the equations, we obtain $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}; f(\boldsymbol{x}) = \frac{1}{2}; J(\boldsymbol{\theta}) = \frac{1}{4}$
- We will never achieve 100% accuracy. Why?

Example: Learning XOR

- The XOR function is an operation on two binary values, x_1 and x_2.
- Target: $y = f^*(\boldsymbol{x})$
- Model: $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- Learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*
- Regression problem
- use mean squared error(MSE) loss function:

$$J(\boldsymbol{\theta}) = \frac{1}{4} \sum_{x \in \mathbb{X}} (f^*(\boldsymbol{x}) - f(\boldsymbol{x}; \boldsymbol{\theta}))^2$$

- Suppose our model is:

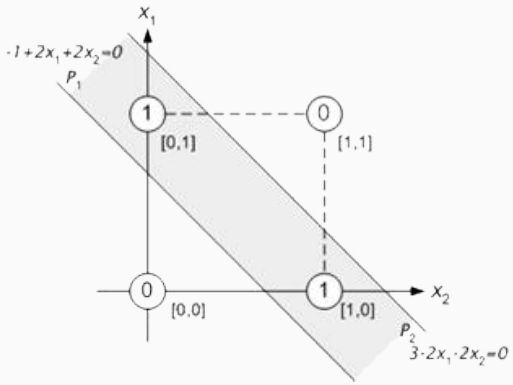
$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T \boldsymbol{w} + b$$

- After solving the equations, we obtain $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}$; $f(\boldsymbol{x}) = \frac{1}{2}$; $J(\boldsymbol{\theta}) = \frac{1}{4}$
- We will never achieve 100% accuracy. Why?

Table 1: XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

Figure 5: XOR problem



Solution of XOR problem

- Why linear model cannot deal with XOR problem?
- Solution? Nonlinear or transform space!

Figures from colah's blog

- One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.
- Non-linear models: Neural Networks, SVM, etc.

Solution of XOR problem

- Why linear model cannot deal with XOR problem?
- Solution? Nonlinear or transform space!

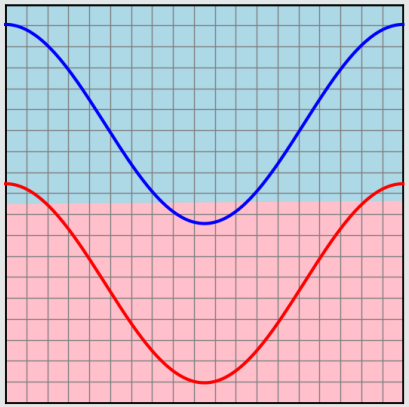
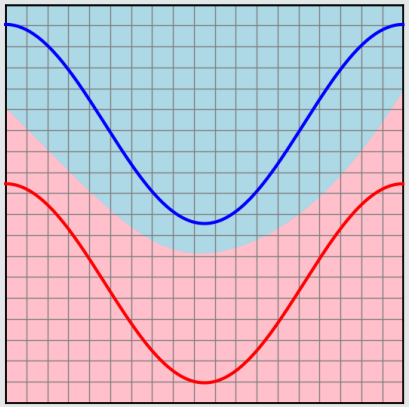
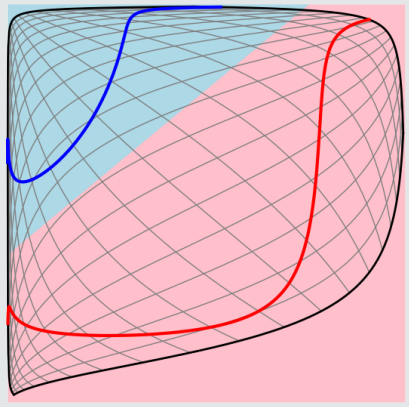
Figures from colah's blog

- One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.
- Non-linear models: Neural Networks, SVM, etc.

Solution of XOR problem

- Why linear model cannot deal with XOR problem?
- Solution? Nonlinear or transform space!

Figures from colah's blog

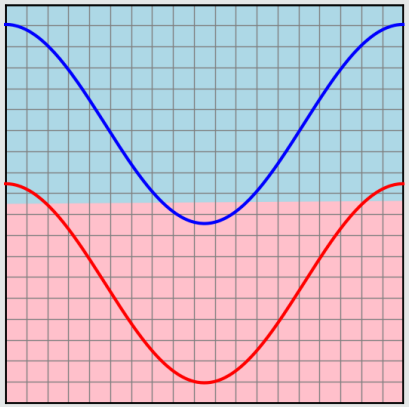
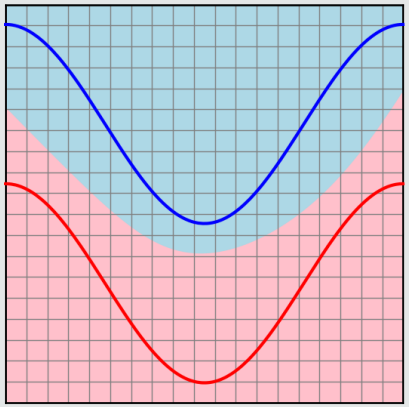
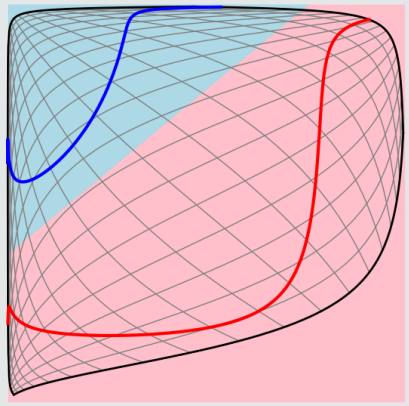




- One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.
- Non-linear models: Neural Networks, SVM, etc.

Solution of XOR problem

- Why linear model cannot deal with XOR problem?
- Solution? Nonlinear or transform space!

Figures from colah's blog

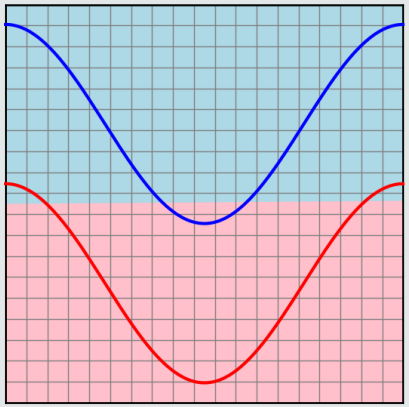
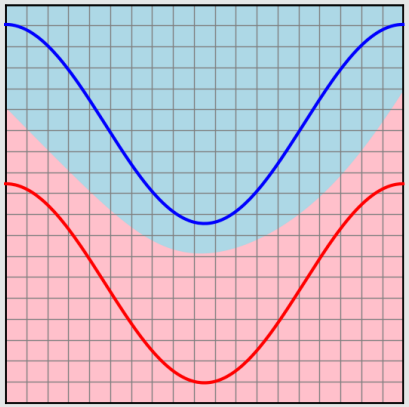
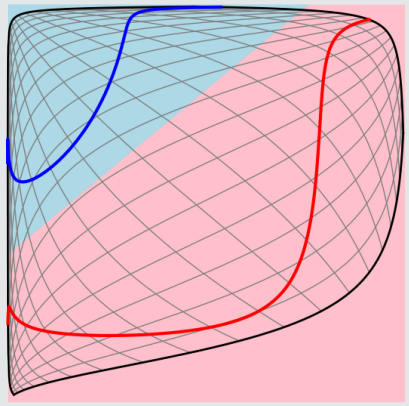




- One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.
- Non-linear models: Neural Networks, SVM, etc.

Solution of XOR problem

- Why linear model cannot deal with XOR problem?
- Solution? Nonlinear or transform space!

Figures from colah's blog

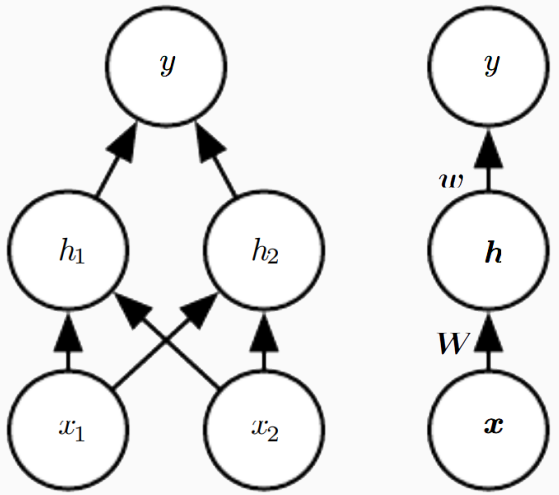




- One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.
- Non-linear models: Neural Networks, SVM, etc.

A simple feedforward network

- A vector of hidden units $\boldsymbol{h}$ that are computed by a function $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$
- The values of these hidden units are then used as the input for a second layer.
- The output layer is still just a linear regression model, but now it is applied to $\boldsymbol{h}$ rather than to $\boldsymbol{x}$.
- The network now contains two functions chained together: $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$.
- the complete model is: $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$

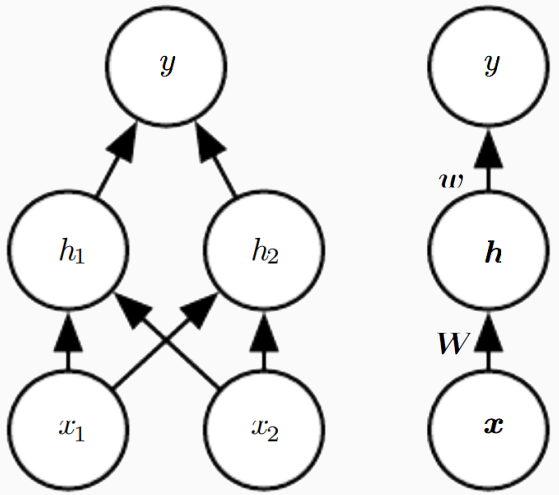
Figure 6: A feedforward network with one hidden layer and two hidden units



A simple feedforward network

- A vector of hidden units $\boldsymbol{h}$ that are computed by a function $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$
- The values of these hidden units are then used as the input for a second layer.
- The output layer is still just a linear regression model, but now it is applied to $\boldsymbol{h}$ rather than to $\boldsymbol{x}$.
- The network now contains two functions chained together: $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$.
- the complete model is: $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$

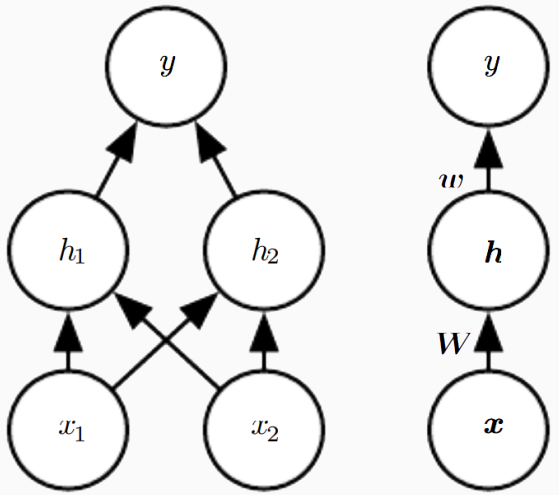
Figure 6: A feedforward network with one hidden layer and two hidden units



A simple feedforward network

- A vector of hidden units $\boldsymbol{h}$ that are computed by a function $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$
- The values of these hidden units are then used as the input for a second layer.
- The output layer is still just a linear regression model, but now it is applied to $\boldsymbol{h}$ rather than to $\boldsymbol{x}$.
- The network now contains two functions chained together: $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$.
- the complete model is: $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$

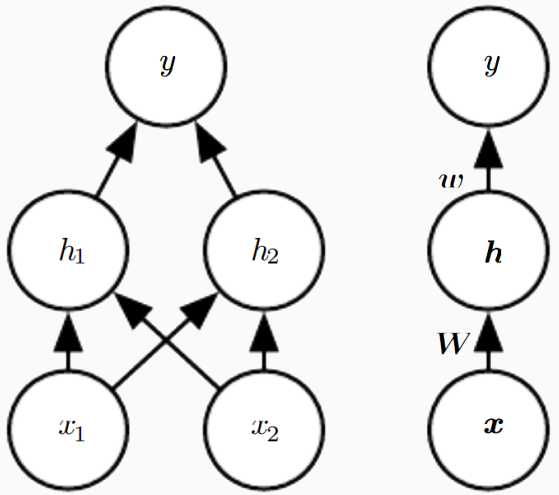
Figure 6: A feedforward network with one hidden layer and two hidden units



A simple feedforward network

- A vector of hidden units $\boldsymbol{h}$ that are computed by a function $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$
- The values of these hidden units are then used as the input for a second layer.
- The output layer is still just a linear regression model, but now it is applied to $\boldsymbol{h}$ rather than to $\boldsymbol{x}$.
- The network now contains two functions chained together: $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$.
- the complete model is: $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$

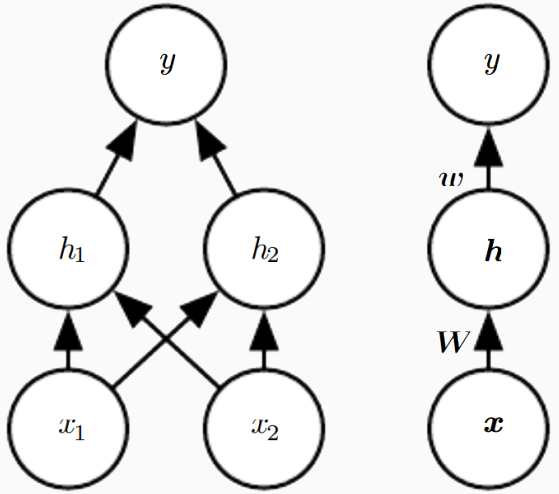
Figure 6: A feedforward network with one hidden layer and two hidden units



A simple feedforward network

- $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$
- What function should $f^{(1)}$ compute? We may be tempting to make $f^{(1)}$be linear as well?
- Unfortunately, if $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input.
- suppose $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^T\boldsymbol{x}$ and $f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^T\boldsymbol{w}$. Then $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{W}^T\boldsymbol{x}$. We could represent this function as $f(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{w}'$ where $\boldsymbol{w}' = \boldsymbol{W}\boldsymbol{w}$
- Clearly, we must use a nonlinear function to describe the features.

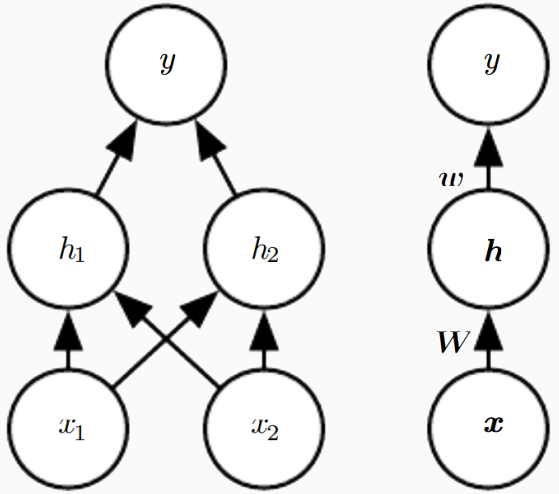
Figure 6: A feedforward network with one hidden layer and two hidden units



A simple feedforward network

- $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$
- What function should $f^{(1)}$ compute? We may be tempting to make $f^{(1)}$be linear as well?
- Unfortunately, if $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input.
- suppose $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^T\boldsymbol{x}$ and $f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^T\boldsymbol{w}$. Then $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{W}^T\boldsymbol{x}$. We could represent this function as $f(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{w}'$ where $\boldsymbol{w}' = \boldsymbol{W}\boldsymbol{w}$
- Clearly, we must use a nonlinear function to describe the features.

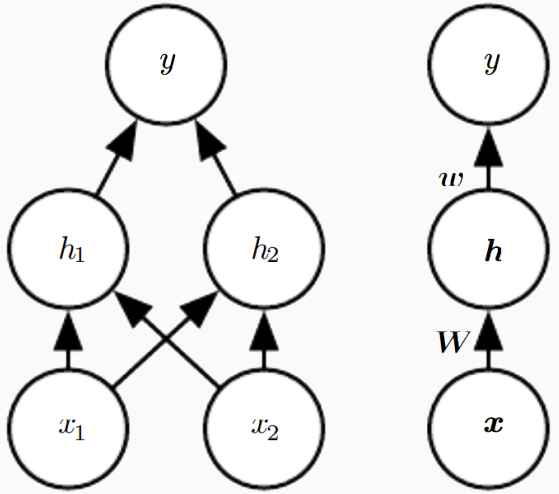
Figure 6: A feedforward network with one hidden layer and two hidden units



A simple feedforward network

- $h = f^{(1)}(x; W, c)$ and $y = f^{(2)}(h; w, b)$
- What function should $f^{(1)}$ compute? We may be tempting to make $f^{(1)}$be linear as well?
- Unfortunately, if $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input.
- suppose $f^{(1)}(x) = W^T x$ and $f^{(2)}(h) = h^T w$. Then $f(x) = w^T W^T x$. We could represent this function as $f(x) = x^T w'$ where $w' = Ww$
- Clearly, we must use a nonlinear function to describe the features.

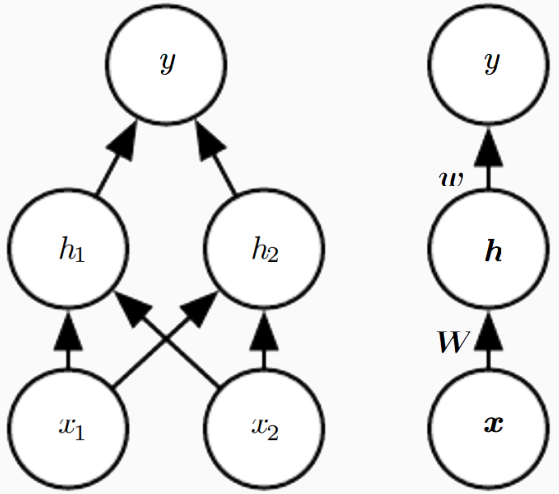
Figure 6: A feedforward network with one hidden layer and two hidden units



A simple feedforward network

- $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$
- What function should $f^{(1)}$ compute? We may be tempting to make $f^{(1)}$be linear as well?
- Unfortunately, if $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input.
- suppose $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^T\boldsymbol{x}$ and $f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^T\boldsymbol{w}$. Then $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{W}^T\boldsymbol{x}$. We could represent this function as $f(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{w}'$ where $\boldsymbol{w}' = \boldsymbol{W}\boldsymbol{w}$
- Clearly, we must use a nonlinear function to describe the features.

Figure 6: A feedforward network with one hidden layer and two hidden units



Activation Function

- Most neural networks describe the features using an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an *activation function*.
- We define: $\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c})$, where $\boldsymbol{W}$ provides the weights of a linear transformation and $\boldsymbol{c}$ the biases.
- The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(\boldsymbol{x}^T\boldsymbol{W}^{:,i} + c_i)$.
- In modern neural networks, the default recommendation is to use the *rectified linear unit* or ReLU (Jarrett et al. [2009], Nair and Hinton [2010], Glorot et al. [2011]) defined by the activation function $g(z) = \max\{0, z\}$

Activation Function

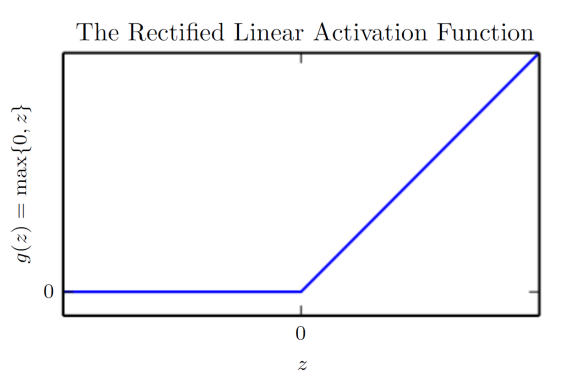
- Most neural networks describe the features using an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an *activation function*.
- We define: $\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c})$, where $\boldsymbol{W}$ provides the weights of a linear transformation and $\boldsymbol{c}$ the biases.
- The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(\boldsymbol{x}^T\boldsymbol{W}^{:,i} + c_i)$.
- In modern neural networks, the default recommendation is to use the *rectified linear unit* or ReLU (Jarrett et al. [2009], Nair and Hinton [2010], Glorot et al. [2011]) defined by the activation function $g(z) = \max\{0, z\}$

Activation Function

- Most neural networks describe the features using an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an *activation function*.
- We define: $\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c})$, where $\boldsymbol{W}$ provides the weights of a linear transformation and $\boldsymbol{c}$ the biases.
- The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(\boldsymbol{x}^T\boldsymbol{W}^{:,i} + c_i)$.
- In modern neural networks, the default recommendation is to use the *rectified linear unit* or ReLU (Jarrett et al. [2009], Nair and Hinton [2010], Glorot et al. [2011]) defined by the activation function $g(z) = \max\{0, z\}$

Activation Function

- Most neural networks describe the features using an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an *activation function*.
- We define: $\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c})$, where $\boldsymbol{W}$ provides the weights of a linear transformation and $\boldsymbol{c}$ the biases.
- The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(\boldsymbol{x}^T\boldsymbol{W}^{:,i} + c_i)$.
- In modern neural networks, the default recommendation is to use the *rectified linear unit* or ReLU (Jarrett et al. [2009], Nair and Hinton [2010], Glorot et al. [2011]) defined by the activation function $g(z) = \max\{0, z\}$



Solution of XOR problem

We can now specify our complete network as

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T \max\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{x}\} + b$$

We can now specify a solution to the XOR problem. Let

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$

Let $\boldsymbol{X}$ be the design matrix containing all four points in the binary input space:

$$x = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Solution of XOR problem

We can now specify our complete network as

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T \max\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{x}\} + b$$

We can now specify a solution to the XOR problem. Let

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$

Let $\boldsymbol{X}$ be the design matrix containing all four points in the binary input space:

$$x = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Solution of XOR problem

We can now specify our complete network as

$$f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = \boldsymbol{w}^T \max\{0, \boldsymbol{W}^T \boldsymbol{x} + \boldsymbol{x}\} + b$$

We can now specify a solution to the XOR problem. Let

$$\boldsymbol{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \boldsymbol{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \boldsymbol{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$

Let $\boldsymbol{X}$ be the design matrix containing all four points in the binary input space:

$$\boldsymbol{x} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Solution of XOR problem

The first step in the neural network is to multiply the input matrix by the first layer's weight matrix:

$$XW = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

Next, we add the bias vector c, to obtain:

$$XW + c = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

Solution of XOR problem

The first step in the neural network is to multiply the input matrix by the first layer's weight matrix:

$$XW = \begin{bmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{bmatrix}$$

Next, we add the bias vector $\boldsymbol{c}$, to obtain:

$$XW + \boldsymbol{c} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

Solution of XOR problem

To finish computing the value of $\boldsymbol{h}$ for each example, we apply the rectified linear transformation and then, we can use a linear model to solve the problem. We finish by multiplying by the weight vector $\boldsymbol{w}$

$$\mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{c}) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 0 \\ 2 & 1 \end{bmatrix}; \qquad \mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{c}) \times \boldsymbol{w} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix};$$

The neural network has obtained the correct answer for every example in the batch.

In this example, we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here.

Instead, a *gradient-based optimization algorithm* can find parameters that produce very little error.

Solution of XOR problem

To finish computing the value of $\boldsymbol{h}$ for each example, we apply the rectified linear transformation and then, we can use a linear model to solve the problem. We finish by multiplying by the weight vector $\boldsymbol{w}$

$$\mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{c}) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 0 \\ 2 & 1 \end{bmatrix}; \qquad \mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{c}) \times \boldsymbol{w} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix};$$

The neural network has obtained the correct answer for every example in the batch.

In this example, we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here.

Instead, a *gradient-based optimization algorithm* can find parameters that produce very little error.

Solution of XOR problem

To finish computing the value of $\boldsymbol{h}$ for each example, we apply the rectified linear transformation and then, we can use a linear model to solve the problem. We finish by multiplying by the weight vector $\boldsymbol{w}$

$$\mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{c}) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 0 \\ 2 & 1 \end{bmatrix}; \qquad \mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{c}) \times \boldsymbol{w} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix};$$

The neural network has obtained the correct answer for every example in the batch.

In this example, we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here.

Instead, a *gradient-based optimization algorithm* can find parameters that produce very little error.

Solution of XOR problem

To finish computing the value of $\boldsymbol{h}$ for each example, we apply the rectified linear transformation and then, we can use a linear model to solve the problem. We finish by multiplying by the weight vector $\boldsymbol{w}$

$$\mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{c}) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 0 \\ 2 & 1 \end{bmatrix}; \qquad \mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{c}) \times \boldsymbol{w} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix};$$

The neural network has obtained the correct answer for every example in the batch.

In this example, we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here.

Instead, a *gradient-based optimization algorithm* can find parameters that produce very little error.

Gradient-Based Learning

- Use *Gradient descent* algorithm to train a neural network.
- There are some difference between linear models we have seen so far and the neural network.
- Nonlinearity $\Longrightarrow$ non-convex loss functions.
- Iterative training, gradient-based optimization.
- Drive the cost function to a very low value, rather than the linear equation solvers (global convergence guarantee).
- Convex optimization converges starting from any initial parameters (in theory).
- *Stochastic gradient descent* applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- **Initialize all weights to small random values and the biases may be initialized to zero or to small positive values.**

Gradient-Based Learning

- Use *Gradient descent* algorithm to train a neural network.
- There are some difference between linear models we have seen so far and the neural network.
- Nonlinearity $\Longrightarrow$ non-convex loss functions.
- Iterative training, gradient-based optimization.
- Drive the cost function to a very low value, rather than the linear equation solvers (global convergence guarantee).
- Convex optimization converges starting from any initial parameters (in theory).
- *Stochastic gradient descent* applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- **Initialize all weights to small random values and the biases may be initialized to zero or to small positive values.**

Gradient-Based Learning

- Use *Gradient descent* algorithm to train a neural network.
- There are some difference between linear models we have seen so far and the neural network.
- Nonlinearity $\Longrightarrow$ non-convex loss functions.
- Iterative training, gradient-based optimization.
- Drive the cost function to a very low value, rather than the linear equation solvers (global convergence guarantee).
- Convex optimization converges starting from any initial parameters (in theory).
- *Stochastic gradient descent* applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- **Initialize all weights to small random values and the biases may be initialized to zero or to small positive values.**

Gradient-Based Learning

- Use *Gradient descent* algorithm to train a neural network.
- There are some difference between linear models we have seen so far and the neural network.
- Nonlinearity $\Longrightarrow$ non-convex loss functions.
- Iterative training, gradient-based optimization.
- Drive the cost function to a very low value, rather than the linear equation solvers (global convergence guarantee).
- Convex optimization converges starting from any initial parameters (in theory).
- *Stochastic gradient descent* applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- **Initialize all weights to small random values and the biases may be initialized to zero or to small positive values.**

Gradient-Based Learning

- Use *Gradient descent* algorithm to train a neural network.
- There are some difference between linear models we have seen so far and the neural network.
- Nonlinearity $\Longrightarrow$ non-convex loss functions.
- Iterative training, gradient-based optimization.
- Drive the cost function to a very low value, rather than the linear equation solvers (global convergence guarantee).
- Convex optimization converges starting from any initial parameters (in theory).
- *Stochastic gradient descent* applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- Initialize all weights to small random values and the biases may be initialized to zero or to small positive values.

Gradient-Based Learning

- Use *Gradient descent* algorithm to train a neural network.
- There are some difference between linear models we have seen so far and the neural network.
- Nonlinearity $\Longrightarrow$ non-convex loss functions.
- Iterative training, gradient-based optimization.
- Drive the cost function to a very low value, rather than the linear equation solvers (global convergence guarantee).
- Convex optimization converges starting from any initial parameters (in theory).
- *Stochastic gradient descent* applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- Initialize all weights to small random values and the biases may be initialized to zero or to small positive values.

Gradient-Based Learning

- Use *Gradient descent* algorithm to train a neural network.
- There are some difference between linear models we have seen so far and the neural network.
- Nonlinearity $\Longrightarrow$ non-convex loss functions.
- Iterative training, gradient-based optimization.
- Drive the cost function to a very low value, rather than the linear equation solvers (global convergence guarantee).
- Convex optimization converges starting from any initial parameters (in theory).
- *Stochastic gradient descent* applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- Initialize all weights to small random values and the biases may be initialized to zero or to small positive values.

Gradient-Based Learning

- Use *Gradient descent* algorithm to train a neural network.
- There are some difference between linear models we have seen so far and the neural network.
- Nonlinearity $\Longrightarrow$ non-convex loss functions.
- Iterative training, gradient-based optimization.
- Drive the cost function to a very low value, rather than the linear equation solvers (global convergence guarantee).
- Convex optimization converges starting from any initial parameters (in theory).
- *Stochastic gradient descent* applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- **Initialize all weights to small random values and the biases may be initialized to zero or to small positive values**.

Cost Functions

- The cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.
- Parametric model defines a distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ and we simply use the principle of maximum likelihood.
- The total cost function = primary cost functions + regularization term.
- Regularization term: weight decay.

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(W_{ji}^{(l)}\right)^2$$

- More advanced regularization strategies for neural networks will be describe in next chapter.

Cost Functions

- The cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.
- Parametric model defines a distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ and we simply use the principle of maximum likelihood.
- The total cost function = primary cost functions + regularization term.
- Regularization term: weight decay.

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(W_{ji}^{(l)}\right)^2$$

- More advanced regularization strategies for neural networks will be describe in next chapter.

Cost Functions

- The cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.
- Parametric model defines a distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ and we simply use the principle of maximum likelihood.
- The total cost function = primary cost functions + regularization term.
- Regularization term: weight decay.

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(W_{ji}^{(l)}\right)^2$$

- More advanced regularization strategies for neural networks will be describe in next chapter.

Cost Functions

- The cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.
- Parametric model defines a distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ and we simply use the principle of maximum likelihood.
- The total cost function = primary cost functions + regularization term.
- Regularization term: weight decay.

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(W_{ji}^{(l)}\right)^2$$

- More advanced regularization strategies for neural networks will be describe in next chapter.

Cost Functions

- The cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.
- Parametric model defines a distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ and we simply use the principle of maximum likelihood.
- The total cost function $=$ primary cost functions $+$ regularization term.
- Regularization term: weight decay.

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{2}\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2\right)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(W_{ji}^{(l)}\right)^2$$

- More advanced regularization strategies for neural networks will be describe in next chapter.

For your information:
Cross-entropy cost function in logistic regression with 1 unit

- We can use MSE to be the cost function:

$$C = \frac{1}{2}(y-a)^2 \text{ where } a = \sigma(z) \text{ and } z = WX+b$$

- The gradient of the cost function C:

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$

- Then update parameters:

$$w \leftarrow w - \eta\frac{\partial C}{\partial w} = w - \eta \times a \times \sigma'(z)$$

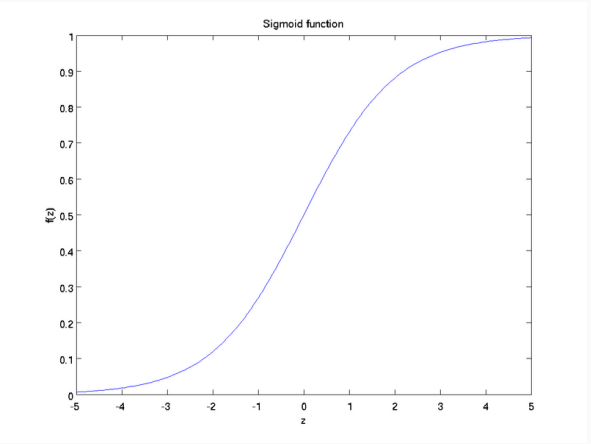


Figure 7: Sigmoid Function (from UFLDL)

For your information:
Cross-entropy cost function in logistic regression with 1 unit

- We can use MSE to be the cost function:

$$C = \frac{1}{2}(y-a)^2 \text{ where } a = \sigma(z) \text{ and } z = WX+b$$

- The gradient of the cost function C:

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$

- Then update parameters:

$$w \leftarrow w - \eta\frac{\partial C}{\partial w} = w - \eta \times a \times \sigma'(z)$$

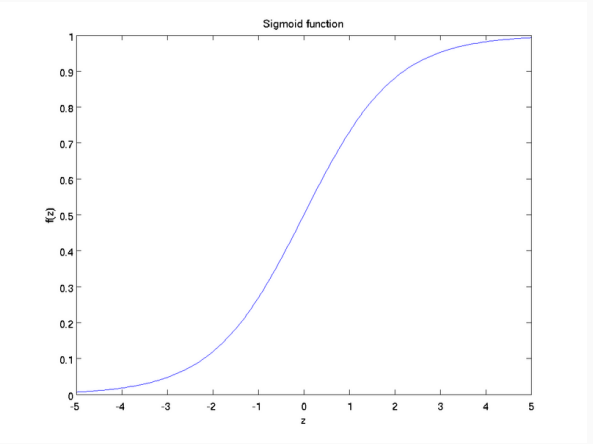


Figure 7: Sigmoid Function (from UFLDL)

For your information:
Cross-entropy cost function in logistic regression with 1 unit

- We can use MSE to be the cost function:

$$C = \frac{1}{2}(y-a)^2 \text{ where } a = \sigma(z) \text{ and } z = WX+b$$

- The gradient of the cost function C:

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$

- Then update parameters:

$$w \leftarrow w - \eta\frac{\partial C}{\partial w} = w - \eta \times a \times \sigma'(z)$$

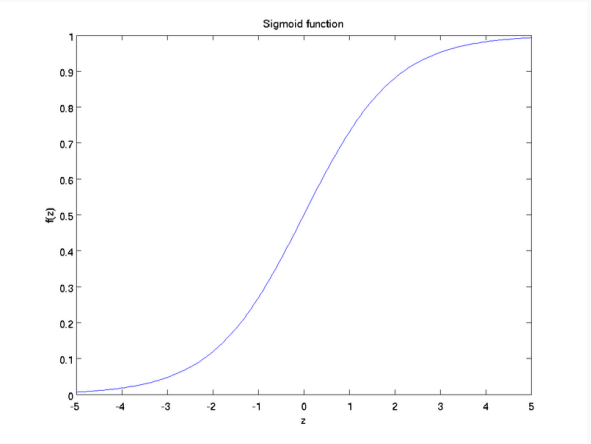


Figure 7: Sigmoid Function (from UFLDL)

For your information:
Cross-entropy cost function in logistic regression with 1 unit

- Cross-entropy cost function:

$$C = -\frac{1}{n} \sum_x [y \ln a + (1-y) \ln(1-a)]$$

- The gradient of the cost function C:

$$\frac{\partial C}{\partial w} = \frac{1}{n} \sum_x x_j (\sigma(z) - y)$$

For your information:
Cross-entropy cost function in logistic regression with 1 unit

- Cross-entropy cost function:

$$C = -\frac{1}{n} \sum_x [y \ln a + (1-y) \ln(1-a)]$$

- The gradient of the cost function C:

$$\frac{\partial C}{\partial w} = \frac{1}{n} \sum_x x_j (\sigma(z) - y)$$

Learning Conditional Distributions with Maximum Likelihood

- Most modern neural networks are trained using maximum likelihood.
- This means that the cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data an the model distribution:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{x,y\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{y}|\boldsymbol{x})$$

- The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{model}$. For example, if $p_{model}(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x};\boldsymbol{\theta}), I)$, the we recover the mean squared error cost,

$$J(\theta) = \frac{1}{2}\mathbb{E}_{x,y\sim\hat{p}_{data}} \|\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})\|^2 + \text{const}$$

- An advantage of this approach of deriving the cost function from maximum likelihood is that it removes the burden of designing cost functions for each model. Specifying a model $p(\boldsymbol{y}|\boldsymbol{x})$ automatically determines a cost function $\log p(\boldsymbol{y}|\boldsymbol{x})$.

Learning Conditional Distributions with Maximum Likelihood

- Most modern neural networks are trained using maximum likelihood.
- This means that the cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data an the model distribution:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{x,y\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{y}|\boldsymbol{x})$$

- The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{model}$. For example, if $p_{model}(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x};\boldsymbol{\theta}), I)$, the we recover the mean squared error cost,

$$J(\theta) = \frac{1}{2}\mathbb{E}_{x,y\sim\hat{p}_{data}} \|\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})\|^2 + \text{const}$$

- An advantage of this approach of deriving the cost function from maximum likelihood is that it removes the burden of designing cost functions for each model. Specifying a model $p(\boldsymbol{y}|\boldsymbol{x})$ automatically determines a cost function $\log p(\boldsymbol{y}|\boldsymbol{x})$.

Learning Conditional Distributions with Maximum Likelihood

- Most modern neural networks are trained using maximum likelihood.
- This means that the cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data an the model distribution:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{x,y\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{y}|\boldsymbol{x})$$

- The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{model}$. For example, if $p_{model}(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x};\boldsymbol{\theta}), I)$, the we recover the mean squared error cost,

$$J(\theta) = \frac{1}{2}\mathbb{E}_{x,y\sim\hat{p}_{data}} \|\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})\|^2 + \text{const}$$

- An advantage of this approach of deriving the cost function from maximum likelihood is that it removes the burden of designing cost functions for each model. Specifying a model $p(\boldsymbol{y}|\boldsymbol{x})$ automatically determines a cost function $\log p(\boldsymbol{y}|\boldsymbol{x})$.

Learning Conditional Distributions with Maximum Likelihood

- Most modern neural networks are trained using maximum likelihood.
- This means that the cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data an the model distribution:

$$J(\boldsymbol{\theta}) = -\mathbb{E}_{x,y\sim\hat{p}_{data}} \log p_{model}(\boldsymbol{y}|\boldsymbol{x})$$

- The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{model}$. For example, if $p_{model}(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x};\boldsymbol{\theta}), I)$, the we recover the mean squared error cost,

$$J(\theta) = \frac{1}{2}\mathbb{E}_{x,y\sim\hat{p}_{data}} \|\boldsymbol{y} - f(\boldsymbol{x};\boldsymbol{\theta})\|^2 + \text{const}$$

- An advantage of this approach of deriving the cost function from maximum likelihood is that it removes the burden of designing cost functions for each model. Specifying a model $p(\boldsymbol{y}|\boldsymbol{x})$ automatically determines a cost function $\log p(\boldsymbol{y}|\boldsymbol{x})$.

Output Units

- Any kind of neural network unit that may be used as an output can also be used as a hidden unit.
- Here, we focus on the use of these units as outputs of the model, but in principle they can be used internally as well.
- Throughout this section, we suppose that the feedforward network provides a set of hidden features defined by $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$
- The role of the output layers is then to provide some additional transformation from the features to complete the task that the network must perform.

Output Units

- Any kind of neural network unit that may be used as an output can also be used as a hidden unit.
- Here, we focus on the use of these units as outputs of the model, but in principle they can be used internally as well.
- Throughout this section, we suppose that the feedforward network provides a set of hidden features defined by $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$
- The role of the output layers is then to provide some additional transformation from the features to complete the task that the network must perform.

Output Units

- Any kind of neural network unit that may be used as an output can also be used as a hidden unit.
- Here, we focus on the use of these units as outputs of the model, but in principle they can be used internally as well.
- Throughout this section, we suppose that the feedforward network provides a set of hidden features defined by $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$
- The role of the output layers is then to provide some additional transformation from the features to complete the task that the network must perform.

Output Units

- Any kind of neural network unit that may be used as an output can also be used as a hidden unit.
- Here, we focus on the use of these units as outputs of the model, but in principle they can be used internally as well.
- Throughout this section, we suppose that the feedforward network provides a set of hidden features defined by $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$
- The role of the output layers is then to provide some additional transformation from the features to complete the task that the network must perform.

Output Units - Linear Units

- One kind of output unit is based on an affine transformation with no nonlinearity. (Linear Units)
- Given features h, a layer of linear output units produces a vector $\hat{y} = W^T h + b$
- Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

$$p(y|x) = \mathcal{N}(y; \hat{y}, I)$$

- Maximizing the log-likelihood is equivalent to minimizing the mean squared error.
- Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms.

Output Units - Linear Units

- One kind of output unit is based on an affine transformation with no nonlinearity. (Linear Units)
- Given features h, a layer of linear output units produces a vector $\hat{y} = W^T h + b$
- Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

$$p(y|x) = \mathcal{N}(y; \hat{y}, I)$$

- Maximizing the log-likelihood is equivalent to minimizing the mean squared error.
- Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms.

Output Units - Linear Units

- One kind of output unit is based on an affine transformation with no nonlinearity. (Linear Units)
- Given features h, a layer of linear output units produces a vector $\hat{y} = W^T h + b$
- Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

$$p(y|x) = \mathcal{N}(y; \hat{y}, I)$$

- Maximizing the log-likelihood is equivalent to minimizing the mean squared error.
- Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms.

Output Units - Linear Units

- One kind of output unit is based on an affine transformation with no nonlinearity. (Linear Units)
- Given features h, a layer of linear output units produces a vector $\hat{y} = W^T h + b$
- Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

$$p(y|x) = \mathcal{N}(y; \hat{y}, I)$$

- Maximizing the log-likelihood is equivalent to minimizing the mean squared error.
- Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms.

Output Units - Linear Units

- One kind of output unit is based on an affine transformation with no nonlinearity. (Linear Units)
- Given features h, a layer of linear output units produces a vector $\hat{y} = W^T h + b$
- Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

$$p(y|x) = \mathcal{N}(y; \hat{y}, I)$$

- Maximizing the log-likelihood is equivalent to minimizing the mean squared error.
- Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms.

Output Units - Linear Units

- One kind of output unit is based on an affine transformation with no nonlinearity. (Linear Units)
- Given features h, a layer of linear output units produces a vector $\hat{y} = W^T h + b$
- Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

$$p(y|x) = \mathcal{N}(y; \hat{y}, I)$$

- Maximizing the log-likelihood is equivalent to minimizing the mean squared error.
- Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms.

Output Units - Sigmoid Units

- Many tasks require predicting the value of a binary variable y.
- Classification problems with two classes can be cast in this form.
- The neural net needs to predict only $P(y = 1|\boldsymbol{x})$.
- A valid probability must lie in the interval $[0, 1]$.

$$P(y = 1|\boldsymbol{x}) = \max\left\{0, min\left\{1, \boldsymbol{w}^T\boldsymbol{h} + b\right\}\right\}$$

- But we would not be able to train it very effectively with gradient descent.
- Any time that $\boldsymbol{w}^T\boldsymbol{h} + b$ strayed outside the unit interval, the gradient of would be 0.
- So we use sigmoid output units combined with maximum likelihood.

Output Units - Sigmoid Units

- Many tasks require predicting the value of a binary variable y.
- Classification problems with two classes can be cast in this form.
- The neural net needs to predict only $P(y = 1|\boldsymbol{x})$.
- A valid probability must lie in the interval $[0, 1]$.

$$P(y = 1|\boldsymbol{x}) = \max\left\{0, min\left\{1, \boldsymbol{w}^T\boldsymbol{h} + b\right\}\right\}$$

- But we would not be able to train it very effectively with gradient descent.
- Any time that $\boldsymbol{w}^T\boldsymbol{h} + b$ strayed outside the unit interval, the gradient of would be 0.
- So we use sigmoid output units combined with maximum likelihood.

Output Units - Sigmoid Units

- Many tasks require predicting the value of a binary variable y.
- Classification problems with two classes can be cast in this form.
- The neural net needs to predict only $P(y = 1|\boldsymbol{x})$.
- A valid probability must lie in the interval $[0, 1]$.

$$P(y = 1|\boldsymbol{x}) = \max\left\{0, min\left\{1, \boldsymbol{w}^T\boldsymbol{h} + b\right\}\right\}$$

- But we would not be able to train it very effectively with gradient descent.
- Any time that $\boldsymbol{w}^T\boldsymbol{h} + b$ strayed outside the unit interval, the gradient of would be 0.
- So we use sigmoid output units combined with maximum likelihood.

Output Units - Sigmoid Units

- Many tasks require predicting the value of a binary variable y.
- Classification problems with two classes can be cast in this form.
- The neural net needs to predict only $P(y = 1|\boldsymbol{x})$.
- A valid probability must lie in the interval $[0, 1]$.

$$P(y = 1|\boldsymbol{x}) = \max\left\{0, min\left\{1, \boldsymbol{w}^T\boldsymbol{h} + b\right\}\right\}$$

- But we would not be able to train it very effectively with gradient descent.
- Any time that $\boldsymbol{w}^T\boldsymbol{h} + b$ strayed outside the unit interval, the gradient of would be 0.
- So we use sigmoid output units combined with maximum likelihood.

Output Units - Sigmoid Units

- Many tasks require predicting the value of a binary variable y.
- Classification problems with two classes can be cast in this form.
- The neural net needs to predict only $P(y = 1|\boldsymbol{x})$.
- A valid probability must lie in the interval $[0, 1]$.

$$P(y = 1|\boldsymbol{x}) = \max\left\{0, min\left\{1, \boldsymbol{w}^T\boldsymbol{h} + b\right\}\right\}$$

- But we would not be able to train it very effectively with gradient descent.
- Any time that $\boldsymbol{w}^T\boldsymbol{h} + b$ strayed outside the unit interval, the gradient of would be 0.
- So we use sigmoid output units combined with maximum likelihood.

Output Units - Sigmoid Units

- Many tasks require predicting the value of a binary variable y.
- Classification problems with two classes can be cast in this form.
- The neural net needs to predict only $P(y = 1|\boldsymbol{x})$.
- A valid probability must lie in the interval $[0, 1]$.

$$P(y = 1|\boldsymbol{x}) = \max\left\{0, min\left\{1, \boldsymbol{w}^T\boldsymbol{h} + b\right\}\right\}$$

- But we would not be able to train it very effectively with gradient descent.
- Any time that $\boldsymbol{w}^T\boldsymbol{h} + b$ strayed outside the unit interval, the gradient of would be 0.
- So we use sigmoid output units combined with maximum likelihood.

Output Units - Sigmoid Units

- A sigmoid output unit is defined by:

$$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{h} + b)$$

- The sigmoid function is:

$$\sigma(h) = \frac{1}{1 + e^{-h}}$$

- We can think of the sigmoid output unit as having two components.
  1. A linear layer to compute $z = \boldsymbol{w}^T \boldsymbol{h} + b$
  2. Convert z into a probability by the sigmoid activation function.

Output Units - Sigmoid Units

- A sigmoid output unit is defined by:

$$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{h} + b)$$

- The sigmoid function is:

$$\sigma(h) = \frac{1}{1 + e^{-h}}$$

- We can think of the sigmoid output unit as having two components.
  1. A linear layer to compute $z = \boldsymbol{w}^T \boldsymbol{h} + b$
  2. Convert z into a probability by the sigmoid activation function.

Output Units - Sigmoid Units

- A sigmoid output unit is defined by:

$$\hat{y} = \sigma(\boldsymbol{w}^T \boldsymbol{h} + b)$$

- The sigmoid function is:

$$\sigma(h) = \frac{1}{1 + e^{-h}}$$

- We can think of the sigmoid output unit as having two components.
  1. A linear layer to compute $z = \boldsymbol{w}^T \boldsymbol{h} + b$
  2. Convert z into a probability by the sigmoid activation function.

Output Units - Softmax Units

- Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes.
- To generalize to the case of a discrete variable with n values, we now need to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i|\boldsymbol{x})$.
- We require not only that each element of $\hat{y}_i$ between 0 and 1, but also that the entire vector sums to 1.

$$\text{Softmax}(\boldsymbol{z})_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

- Defining $P(y = 1|\boldsymbol{x}) = \sigma(z)$ is equivalent to defining $P(y = 1|\boldsymbol{x}) = softmax(\boldsymbol{z})_1$ with a two-dimensional $\boldsymbol{z}$ and $z_1 = 0$.

Output Units - Softmax Units

- Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes.
- To generalize to the case of a discrete variable with n values, we now need to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i|\boldsymbol{x})$.
- We require not only that each element of $\hat{y}_i$ between 0 and 1, but also that the entire vector sums to 1.

$$\text{Softmax}(\boldsymbol{z})_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

- Defining $P(y = 1|\boldsymbol{x}) = \sigma(z)$ is equivalent to defining $P(y = 1|\boldsymbol{x}) = softmax(\boldsymbol{z})_1$ with a two-dimensional $\boldsymbol{z}$ and $z_1 = 0$.

Output Units - Softmax Units

- Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes.
- To generalize to the case of a discrete variable with n values, we now need to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i|\boldsymbol{x})$.
- We require not only that each element of $\hat{y}_i$ between 0 and 1, but also that the entire vector sums to 1.

$$\text{Softmax}(\boldsymbol{z})_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

- Defining $P(y = 1|\boldsymbol{x}) = \sigma(z)$ is equivalent to defining $P(y = 1|\boldsymbol{x}) = softmax(\boldsymbol{z})_1$ with a two-dimensional $\boldsymbol{z}$ and $z_1 = 0$.

Output Units - Softmax Units

- Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes.
- To generalize to the case of a discrete variable with n values, we now need to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i|\boldsymbol{x})$.
- We require not only that each element of $\hat{y}_i$ between 0 and 1, but also that the entire vector sums to 1.

$$\text{Softmax}(\boldsymbol{z})_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

- Defining $P(y = 1|\boldsymbol{x}) = \sigma(z)$ is equivalent to defining $P(y = 1|\boldsymbol{x}) = softmax(\boldsymbol{z})_1$ with a two-dimensional $\boldsymbol{z}$ and $z_1 = 0$.

Output Units - Softmax Units

- Softmax functions are most often used as the output of a classifier, to represent the probability distribution over n different classes.
- To generalize to the case of a discrete variable with n values, we now need to produce a vector $\hat{\boldsymbol{y}}$, with $\hat{y}_i = P(y = i|\boldsymbol{x})$.
- We require not only that each element of $\hat{y}_i$ between 0 and 1, but also that the entire vector sums to 1.

$$\text{Softmax}(\boldsymbol{z})_i = \frac{exp(z_i)}{\sum_j exp(z_j)}$$

- Defining $P(y = 1|\boldsymbol{x}) = \sigma(z)$ is equivalent to defining $P(y = 1|\boldsymbol{x}) = softmax(\boldsymbol{z})_1$ with a two-dimensional $\boldsymbol{z}$ and $z_1 = 0$.

Output Units - Softmax Units

From a neuroscientific point of view, it is interesting to think of the softmax as a way to create a form of competition between the units that participate in it: the softmax outputs always sum to 1 so an increase in the value of one unit necessarily corresponds to a decreases in the value of others.

This is analogous to the lateral inhibition that is believed to exist between nearby neurons in the cortex. At the extreme it becomes a form of *winner – take – all* (one of the outputs is nearly 1 and the others are nearly 0).

Hidden Units
Retified Linear Units and Their Generalizations

- Easy to optimize: similar to linear units

$$g(z) = \max\{0, z\}$$

- A variety of generalizations of ReLU guarantee that they receive gradient everywhere.
- Three generalizations of rectified linear units are based on using a non-zero slope α_i when $z_i < 0$:

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

  1. *Absolute value rectification* (Jarrett et al. [2009]) fixes $\alpha_i = -1$ to obtain $g(z) = |z|$
  2. A leaky ReLU (Maas et al. [2013]) fixes α_i to a small value like 0.01
  3. A *parametric* ReLU or PReLU (He et al. [2015]) treats α_i as a learnable parameter.

- Maxout (Goodfellow et al. [2013])

Hidden Units
Retified Linear Units and Their Generalizations

- Easy to optimize: similar to linear units

$$g(z) = \max\{0, z\}$$

- A variety of generalizations of ReLU guarantee that they receive gradient everywhere.
- Three generalizations of rectified linear units are based on using a non-zero slope α_i when $z_i < 0$:

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

  1. *Absolute value rectification* (Jarrett et al. [2009]) fixes $\alpha_i = -1$ to obtain $g(z) = |z|$
  2. A leaky ReLU (Maas et al. [2013]) fixes α_i to a small value like 0.01
  3. A *parametric* ReLU or PReLU (He et al. [2015]) treats α_i as a learnable parameter.
- Maxout (Goodfellow et al. [2013])

Hidden Units
Retified Linear Units and Their Generalizations

- Easy to optimize: similar to linear units

$$g(z) = \max\{0, z\}$$

- A variety of generalizations of ReLU guarantee that they receive gradient everywhere.
- Three generalizations of rectified linear units are based on using a non-zero slope α_i when $z_i < 0$:

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

1. *Absolute value rectification* (Jarrett et al. [2009]) fixes $\alpha_i = -1$ to obtain $g(z) = |z|$
2. A leaky ReLU (Maas et al. [2013]) fixes α_i to a small value like 0.01
3. A *parametric* ReLU or PReLU (He et al. [2015]) treats α_i as a learnable parameter.

- Maxout (Goodfellow et al. [2013])

Hidden Units
Retified Linear Units and Their Generalizations

- Easy to optimize: similar to linear units

$$g(z) = \max\{0, z\}$$

- A variety of generalizations of ReLU guarantee that they receive gradient everywhere.
- Three generalizations of rectified linear units are based on using a non-zero slope α_i when $z_i < 0$:

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

1. *Absolute value rectification* (Jarrett et al. [2009]) fixes $\alpha_i = -1$ to obtain $g(z) = |z|$
2. A leaky ReLU (Maas et al. [2013]) fixes α_i to a small value like 0.01
3. A *parametric* ReLU or PReLU (He et al. [2015]) treats α_i as a learnable parameter.

- Maxout (Goodfellow et al. [2013])

Hidden Units
Retified Linear Units and Their Generalizations

- Easy to optimize: similar to linear units

$$g(z) = \max\{0, z\}$$

- A variety of generalizations of ReLU guarantee that they receive gradient everywhere.
- Three generalizations of rectified linear units are based on using a non-zero slope α_i when $z_i < 0$:

$$h_i = g(\boldsymbol{z}, \boldsymbol{\alpha})_i = \max(0, z_i) + \alpha_i \min(0, z_i)$$

  1. *Absolute value rectification* (Jarrett et al. [2009]) fixes $\alpha_i = -1$ to obtain $g(z) = |z|$
  2. A leaky ReLU (Maas et al. [2013]) fixes α_i to a small value like 0.01
  3. A *parametric* ReLU or PReLU (He et al. [2015]) treats α_i as a learnable parameter.

- Maxout (Goodfellow et al. [2013])

Hidden Units
Logistic Sigmoid and Hyperbolic Tangent

- Prior to the introduction of rectified linear units, most neural networks used the logistic sigmoid activation function

$$g(z) = \sigma(z)$$

- Or the hyperbolic tangent activation function

$$g(z) = \tanh(z)$$

- These activation functions are closely related because $\tanh(z) = 2\sigma(2z) - 1$

Other Hidden Units

- *Radial basis function* or RBF unit

$$h_i = \exp\left(-\frac{1}{\sigma_i^2}\|\boldsymbol{W}_{:,i} - x\|^2\right)$$

- *Softplus* (Dugas et al. [2001])

$$g(a) = \xi(a) = \log(1 + e^a)$$

- *Hard* tanh (Collobert [2004])

$$g(a) = \max(-1, \min(1, a))$$

Architecture Design

- Another key design consideration for neural networks is determining the architecture.
- The word *architecture* refers to the overall structure of the network:
  1. How many units it should have
  2. How these units should be connected to each other.
- Most networks are organized into groups of units called layers.
- Most network architecture arrange these layers in a chain structure.
- In this structure, the layers are given by

$$h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)})$$

$$h^{(2)} = g^{(2)}(W^{(1)T}h^{(1)} + b^{(2)})$$

  and so on.
- Deeper networks often are able to far fewer units per layer and far fewer parameters and often generalize to the test set, but are also often harder to optimize.

Architecture Design

- Another key design consideration for neural networks is determining the architecture.
- The word *architecture* refers to the overall structure of the network:
  1. How many units it should have
  2. How these units should be connected to each other.
- Most networks are organized into groups of units called layers.
- Most network architecture arrange these layers in a chain structure.
- In this structure, the layers are given by

$$h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)})$$

$$h^{(2)} = g^{(2)}(W^{(1)T}h^{(1)} + b^{(2)})$$

  and so on.
- Deeper networks often are able to far fewer units per layer and far fewer parameters and often generalize to the test set, but are also often harder to optimize.

Architecture Design

- Another key design consideration for neural networks is determining the architecture.
- The word *architecture* refers to the overall structure of the network:
  1. How many units it should have
  2. How these units should be connected to each other.
- Most networks are organized into groups of units called layers.
- Most network architecture arrange these layers in a chain structure.
- In this structure, the layers are given by

$$h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)})$$

$$h^{(2)} = g^{(2)}(W^{(1)T}h^{(1)} + b^{(2)})$$

  and so on.
- Deeper networks often are able to far fewer units per layer and far fewer parameters and often generalize to the test set, but are also often harder to optimize.

Architecture Design

- Another key design consideration for neural networks is determining the architecture.
- The word *architecture* refers to the overall structure of the network:
  1. How many units it should have
  2. How these units should be connected to each other.
- Most networks are organized into groups of units called layers.
- Most network architecture arrange these layers in a chain structure.
- In this structure, the layers are given by

$$h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)})$$

$$h^{(2)} = g^{(2)}(W^{(1)T}h^{(1)} + b^{(2)})$$

  and so on.
- Deeper networks often are able to far fewer units per layer and far fewer parameters and often generalize to the test set, but are also often harder to optimize.

Architecture Design

- Another key design consideration for neural networks is determining the architecture.
- The word *architecture* refers to the overall structure of the network:
  1. How many units it should have
  2. How these units should be connected to each other.
- Most networks are organized into groups of units called layers.
- Most network architecture arrange these layers in a chain structure.
- In this structure, the layers are given by

$$h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)})$$

$$h^{(2)} = g^{(2)}(W^{(1)T}h^{(1)} + b^{(2)})$$

  and so on.
- Deeper networks often are able to far fewer units per layer and far fewer parameters and often generalize to the test set, but are also often harder to optimize.

Architecture Design

- Another key design consideration for neural networks is determining the architecture.
- The word *architecture* refers to the overall structure of the network:
  1. How many units it should have
  2. How these units should be connected to each other.
- Most networks are organized into groups of units called layers.
- Most network architecture arrange these layers in a chain structure.
- In this structure, the layers are given by

$$h^{(1)} = g^{(1)}(W^{(1)T}x + b^{(1)})$$

$$h^{(2)} = g^{(2)}(W^{(1)T}h^{(1)} + b^{(2)})$$

  and so on.
- Deeper networks often are able to far fewer units per layer and far fewer parameters and often generalize to the test set, but are also often harder to optimize.

Universal Approximation Properties and Depth

- The reason why linear models are easy to train is their loss functions are convex.
- Unfortunately, we often want to learn nonlinear functions.
- Fortunately, feedforward networks with hidden layers provide a universal approximation framework: *universal approximation theorem* (Hornik et al. [1989], Cybenko [1989])

Universal Approximation Properties and Depth

- The reason why linear models are easy to train is their loss functions are convex.
- Unfortunately, we often want to learn nonlinear functions.
- Fortunately, feedforward networks with hidden layers provide a universal approximation framework: *universal approximation theorem* (Hornik et al. [1989], Cybenko [1989])

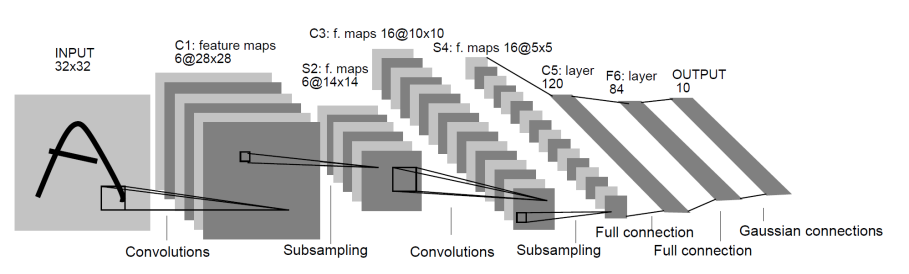
Universal Approximation Properties and Depth

- The reason why linear models are easy to train is their loss functions are convex.
- Unfortunately, we often want to learn nonlinear functions.
- Fortunately, feedforward networks with hidden layers provide a universal approximation framework: *universal approximation theorem* (Hornik et al. [1989], Cybenko [1989])

Other Architectural Considerations

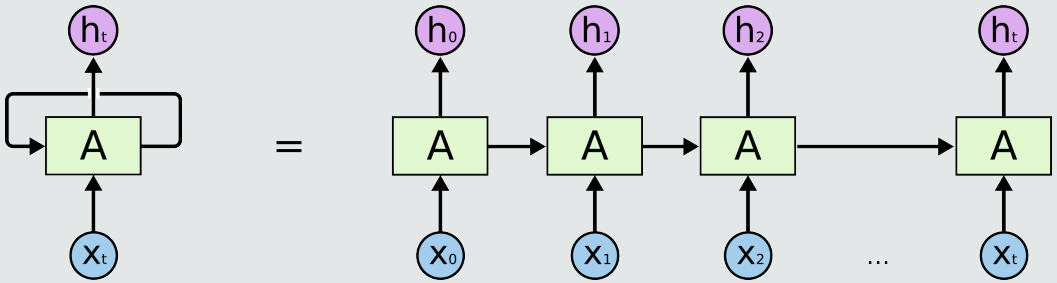
- Convolutional Networks for computer vision

Convolutional Network (LeNet-5 LeCun et al. [1998])



- Recurrent Neural Networks for sequence modeling

Recurrent Neural Network



Back-Propagation and Other Differentiation Algorithms

- input $\boldsymbol{x}$ provide information $\rightarrow$ hidden layers $\rightarrow$ an output $\hat{\boldsymbol{y}}$
- This process called *forward propagation*.
- During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$
- Computing an analytical expression for gradient: computationally expensive.
- The *back-propagation* algorithm (Rumelhard and Williams [1986]) allows the information from the cost to then flow backward through the network, in order to compute the gradient.
- The gradient we require is the gradient of the cost function with respect to the parameters

$$\nabla_{\boldsymbol{\theta}} \mathrm{J}(\boldsymbol{\theta})$$

Back-Propagation and Other Differentiation Algorithms

- input $\boldsymbol{x}$ provide information $\rightarrow$ hidden layers $\rightarrow$ an output $\hat{\boldsymbol{y}}$
- This process called *forward propagation*.
- During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$
- Computing an analytical expression for gradient: computationally expensive.
- The *back-propagation* algorithm (Rumelhard and Williams [1986]) allows the information from the cost to then flow backward through the network, in order to compute the gradient.
- The gradient we require is the gradient of the cost function with respect to the parameters

$$\nabla_{\theta} \mathrm{J}(\theta)$$

Back-Propagation and Other Differentiation Algorithms

- input $\boldsymbol{x}$ provide information $\rightarrow$ hidden layers $\rightarrow$ an output $\hat{\boldsymbol{y}}$
- This process called *forward propagation*.
- During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$
- Computing an analytical expression for gradient: computationally expensive.
- The *back-propagation* algorithm (Rumelhard and Williams [1986]) allows the information from the cost to then flow backward through the network, in order to compute the gradient.
- The gradient we require is the gradient of the cost function with respect to the parameters

$$\nabla_{\theta} \mathrm{J}(\theta)$$

Back-Propagation and Other Differentiation Algorithms

- input $\boldsymbol{x}$ provide information $\rightarrow$ hidden layers $\rightarrow$ an output $\hat{\boldsymbol{y}}$
- This process called *forward propagation*.
- During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$
- Computing an analytical expression for gradient: computationally expensive.
- The *back-propagation* algorithm (Rumelhard and Williams [1986]) allows the information from the cost to then flow backward through the network, in order to compute the gradient.
- The gradient we require is the gradient of the cost function with respect to the parameters

$$\nabla_{\theta} \mathrm{J}(\theta)$$

Back-Propagation and Other Differentiation Algorithms

- input $\boldsymbol{x}$ provide information $\rightarrow$ hidden layers $\rightarrow$ an output $\hat{\boldsymbol{y}}$
- This process called *forward propagation*.
- During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$
- Computing an analytical expression for gradient: computationally expensive.
- The *back-propagation* algorithm (Rumelhard and Williams [1986]) allows the information from the cost to then flow backward through the network, in order to compute the gradient.
- The gradient we require is the gradient of the cost function with respect to the parameters

$$\nabla_{\theta} \mathrm{J}(\theta)$$

Back-Propagation and Other Differentiation Algorithms

- input $\boldsymbol{x}$ provide information $\rightarrow$ hidden layers $\rightarrow$ an output $\hat{\boldsymbol{y}}$
- This process called *forward propagation*.
- During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$
- Computing an analytical expression for gradient: computationally expensive.
- The *back-propagation* algorithm (Rumelhard and Williams [1986]) allows the information from the cost to then flow backward through the network, in order to compute the gradient.
- The gradient we require is the gradient of the cost function with respect to the parameters

$$\nabla_{\theta} \mathrm{J}(\theta)$$

Back-Propagation and Other Differentiation Algorithms

- input $\boldsymbol{x}$ provide information $\rightarrow$ hidden layers $\rightarrow$ an output $\hat{\boldsymbol{y}}$
- This process called *forward propagation*.
- During training, forward propagation can continue onward until it produces a scalar cost $J(\theta)$
- Computing an analytical expression for gradient: computationally expensive.
- The *back-propagation* algorithm (Rumelhard and Williams [1986]) allows the information from the cost to then flow backward through the network, in order to compute the gradient.
- The gradient we require is the gradient of the cost function with respect to the parameters

$$\nabla_{\boldsymbol{\theta}} \mathrm{J}(\boldsymbol{\theta})$$

Chain Rule of Calculus

- To compute the derivatives of composite function
- Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

$$y = g(x) \text{ and } z = f(y) = f(g(x))$$

- The chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- Generalize beyond the scalar case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m to \mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$ then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

Chain Rule of Calculus

- To compute the derivatives of composite function
- Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

$$y = g(x) \text{ and } z = f(y) = f(g(x))$$

- The chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- Generalize beyond the scalar case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m to \mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$ then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

Chain Rule of Calculus

- To compute the derivatives of composite function
- Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

$$y = g(x) \text{ and } z = f(y) = f(g(x))$$

- The chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- Generalize beyond the scalar case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m to \mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$ then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

Chain Rule of Calculus

- To compute the derivatives of composite function
- Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

$$y = g(x) \text{ and } z = f(y) = f(g(x))$$

- The chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- Generalize beyond the scalar case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m to \mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$ then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

Chain Rule of Calculus

- To compute the derivatives of composite function
- Back-propagation is an algorithm that computes the chain rule, with a specific order of operations that is highly efficient.

$$y = g(x) \text{ and } z = f(y) = f(g(x))$$

- The chain rule states that

$$\frac{dz}{dx} = \frac{dz}{dy}\frac{dy}{dx}$$

- Generalize beyond the scalar case. Suppose that $\boldsymbol{x} \in \mathbb{R}^m, \boldsymbol{y} \in \mathbb{R}^n$, g maps from $\mathbb{R}^m to \mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $\boldsymbol{y} = g(\boldsymbol{x})$ and $z = f(\boldsymbol{y})$ then

$$\frac{\partial z}{\partial x_i} = \sum_j \frac{\partial z}{\partial y_j}\frac{\partial y_j}{\partial x_i}$$

Recursively Applying the Chain Rule to Obtain Backprob

- Using the chain rule is straightforward.
- However, actually evaluating that expression in a computer introduces some extra considerations
- Repeat computation of subexpressions.
- Store these subexpression and reuse.

Recursively Applying the Chain Rule to Obtain Backprob

- Using the chain rule is straightforward.
- However, actually evaluating that expression in a computer introduces some extra considerations
- Repeat computation of subexpressions.
- Store these subexpression and reuse.

Recursively Applying the Chain Rule to Obtain Backprob

- Using the chain rule is straightforward.
- However, actually evaluating that expression in a computer introduces some extra considerations
- Repeat computation of subexpressions.
- Store these subexpression and reuse.

Recursively Applying the Chain Rule to Obtain Backprob

- Using the chain rule is straightforward.
- However, actually evaluating that expression in a computer introduces some extra considerations
- Repeat computation of subexpressions.
- Store these subexpression and reuse.

Forward Propagation

Algorithm 1 Forward Propagation

Require: Network depth, l
Require: $W^{(i)}, i \in \{1, \ldots, l\}$, the weight matrix of the model
Require: $b^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: x, the input to process
Require: y, the target output

$h^{(0)} = x$
for $k = 1, \ldots, l$ **do**
  $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
  $h^{(l)} = f(a^{(k)})$
end for
$\hat{y} = h^{(l)}$
$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Forward Propagation

Algorithm 2 Forward Propagation

Require: Network depth, l

Require: $W^{(i)}, i \in \{1, \ldots, l\}$, the weight matrix of the model

Require: $b^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model

Require: x, the input to process

Require: y, the target output

$h^{(0)} = x$

for $k = 1, \ldots, l$ **do**

$a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$

$h^{(l)} = f(a^{(k)})$

end for

$\hat{y} = h^{(l)}$

$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Forward Propagation

Algorithm 3 Forward Propagation

Require: Network depth, l

Require: $W^{(i)}, i \in \{1, \ldots, l\}$, the weight matrix of the model

Require: $b^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model

Require: x, the input to process

Require: y, the target output

$h^{(0)} = x$

for $k = 1, \ldots, l$ **do**

$a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$

$h^{(l)} = f(a^{(k)})$

end for

$\hat{y} = h^{(l)}$

$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Forward Propagation

Algorithm 4 Forward Propagation

Require: Network depth, l

Require: $W^{(i)}, i \in \{1, \ldots, l\}$, the weight matrix of the model

Require: $b^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model

Require: x, the input to process

Require: y, the target output

$h^{(0)} = x$

for $k = 1, \ldots, l$ **do**

$a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$

$h^{(l)} = f(a^{(k)})$

end for

$\hat{y} = h^{(l)}$

$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Forward Propagation

Algorithm 5 Forward Propagation

Require: Network depth, l

Require: $W^{(i)}, i \in \{1, \ldots, l\}$, the weight matrix of the model

Require: $b^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model

Require: x, the input to process

Require: y, the target output

$h^{(0)} = x$

for $k = 1, \ldots, l$ **do**

$a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$

$h^{(l)} = f(a^{(k)})$

end for

$\hat{y} = h^{(l)}$

$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Forward Propagation

Algorithm 6 Forward Propagation

Require: Network depth, l
Require: $W^{(i)}, i \in \{1, \ldots, l\}$, the weight matrix of the model
Require: $b^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: x, the input to process
Require: y, the target output

$h^{(0)} = x$
for $k = 1, \ldots, l$ do
  $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
  $h^{(l)} = f(a^{(k)})$
end for
$\hat{y} = h^{(l)}$
$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Forward Propagation

Algorithm 7 Forward Propagation

Require: Network depth, l
Require: $W^{(i)}, i \in \{1, \ldots, l\}$, the weight matrix of the model
Require: $b^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: x, the input to process
Require: y, the target output

$h^{(0)} = x$
for $k = 1, \ldots, l$ do
  $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
  $h^{(l)} = f(a^{(k)})$
end for
$\hat{y} = h^{(l)}$
$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Forward Propagation

Algorithm 8 Forward Propagation

Require: Network depth, l
Require: $W^{(i)}, i \in \{1, \ldots, l\}$, the weight matrix of the model
Require: $b^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: x, the input to process
Require: y, the target output

$h^{(0)} = x$
for $k = 1, \ldots, l$ **do**
  $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
  $h^{(l)} = f(a^{(k)})$
end for
$\hat{y} = h^{(l)}$
$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Forward Propagation

Algorithm 9 Forward Propagation

Require: Network depth, l
Require: $W^{(i)}, i \in \{1, \ldots, l\}$, the weight matrix of the model
Require: $b^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: x, the input to process
Require: y, the target output

$h^{(0)} = x$
for $k = 1, \ldots, l$ **do**
  $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
  $h^{(l)} = f(a^{(k)})$
end for
$\hat{y} = h^{(l)}$
$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Forward Propagation

Algorithm 10 Forward Propagation

Require: Network depth, l
Require: $W^{(i)}, i \in \{1, \ldots, l\}$, the weight matrix of the model
Require: $b^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
Require: x, the input to process
Require: y, the target output

$h^{(0)} = x$
for $k = 1, \ldots, l$ **do**
  $a^{(k)} = b^{(k)} + W^{(k)} h^{(k-1)}$
  $h^{(l)} = f(a^{(k)})$
end for
$\hat{y} = h^{(l)}$
$J = L(\hat{y}, y) + \lambda \Omega(\theta)$

Backward Propagation

Algorithm 11 Backward computation

After the forward computation, compute the gradient on the output layer:

$g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} (\hat{y}, y)$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation:

$g \leftarrow \nabla_{\hat{a}^{(k)}} J = g \otimes f'(a^{(k)})$

Compute gradients on weights and biases (including the regularization term):

$\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \Omega(\theta)$

$\nabla_{W^{(k)}} J = g h^{(k-1)^T} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower level hidden layer's activations:

$g \leftarrow \nabla_{\hat{h}^{(k-1)}} J = W^{(k)^T} g$

end for

Backward Propagation

Algorithm 12 Backward computation

After the forward computation, compute the gradient on the output layer:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} (\hat{\boldsymbol{y}}, y)$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{a}}^{(k)}} J = \boldsymbol{g} \otimes f'(\boldsymbol{a}^{(k)})$

Compute gradients on weights and biases (including the regularization term):

$\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$

$\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\boldsymbol{h}^{(k-1)^T} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower level hidden layer's activations:

$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{h}}^{(k-1)}} J = \boldsymbol{W}^{(k)^T} \boldsymbol{g}$

end for

Backward Propagation

Algorithm 13 Backward computation

After the forward computation, compute the gradient on the output layer:

$g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} (\hat{y}, y)$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation:

$g \leftarrow \nabla_{\hat{a}^{(k)}} J = g \otimes f'(a^{(k)})$

Compute gradients on weights and biases (including the regularization term):

$\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \Omega(\theta)$

$\nabla_{W^{(k)}} J = g h^{(k-1)^T} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower level hidden layer's activations:

$g \leftarrow \nabla_{\hat{h}^{(k-1)}} J = W^{(k)^T} g$

end for

Backward Propagation

Algorithm 14 Backward computation

After the forward computation, compute the gradient on the output layer:

$g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} (\hat{y}, y)$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation:

$g \leftarrow \nabla_{\hat{a}^{(k)}} J = g \otimes f'(a^{(k)})$

Compute gradients on weights and biases (including the regularization term):

$\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \Omega(\theta)$

$\nabla_{W^{(k)}} J = g h^{(k-1)^T} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower level hidden layer's activations:

$g \leftarrow \nabla_{\hat{h}^{(k-1)}} J = W^{(k)^T} g$

end for

Backward Propagation

Algorithm 15 Backward computation

After the forward computation, compute the gradient on the output layer:

$g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} (\hat{y}, y)$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation:

$g \leftarrow \nabla_{\hat{a}^{(k)}} J = g \otimes f'(a^{(k)})$

Compute gradients on weights and biases (including the regularization term):

$\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \Omega(\theta)$

$\nabla_{W^{(k)}} J = g h^{(k-1)T} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower level hidden layer's activations:

$g \leftarrow \nabla_{\hat{h}^{(k-1)}} J = W^{(k)T} g$

end for

Backward Propagation

Algorithm 16 Backward computation

After the forward computation, compute the gradient on the output layer:

$g \leftarrow \nabla_{\hat{y}} J = \nabla_{\hat{y}} (\hat{y}, y)$

for $k = l, l-1, \ldots, 1$ **do**

Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation:

$g \leftarrow \nabla_{\hat{a}^{(k)}} J = g \otimes f'(a^{(k)})$

Compute gradients on weights and biases (including the regularization term):

$\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \Omega(\theta)$

$\nabla_{W^{(k)}} J = g h^{(k-1)T} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$

Propagate the gradients w.r.t. the next lower level hidden layer's activations:

$g \leftarrow \nabla_{\hat{h}^{(k-1)}} J = W^{(k)T} g$

end for

Backward Propagation from UFLDL

- UFLDL (Unsupervised Feature Learning and Deep Learning)
- Backpropagation Algorithm

Backward Propagation from UFLDL

- Suppose we have a fixed training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m training examples.
- For a single training example (x, y), cost function:

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

- Given a training set of m examples, the overall cost function:

$$\begin{aligned} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l + 1} \left(W_{ji}^{(l)} \right)^2 \\ &= \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \| h_{W,b}(x^{(i)}) - y^{(i)} \|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l + 1} \left(W_{ji}^{(l)} \right)^2 \end{aligned}$$

Backward Propagation from UFLDL

- Suppose we have a fixed training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m training examples.
- For a single training example (x, y), cost function:

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

- Given a training set of m examples, the overall cost function:

$$\begin{aligned} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l + 1} \left(W_{ji}^{(l)} \right)^2 \\ &= \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \| h_{W,b}(x^{(i)}) - y^{(i)} \|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l + 1} \left(W_{ji}^{(l)} \right)^2 \end{aligned}$$

Backward Propagation from UFLDL

- Suppose we have a fixed training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m training examples.
- For a single training example (x, y), cost function:

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

- Given a training set of m examples, the overall cost function:

$$\begin{aligned} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l + 1} \left(W_{ji}^{(l)} \right)^2 \\ &= \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \| h_{W,b}(x^{(i)}) - y^{(i)} \|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l + 1} \left(W_{ji}^{(l)} \right)^2 \end{aligned}$$

Backward Propagation from UFLDL

- Suppose we have a fixed training set $\{(x^{(1)}, y^{(1)}), \ldots, (x^{(m)}, y^{(m)})\}$ of m training examples.
- For a single training example (x, y), cost function:

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

- Given a training set of m examples, the overall cost function:

$$\begin{aligned} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} J(W, b; x^{(i)}, y^{(i)}) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l + 1} \left(W_{ji}^{(l)} \right)^2 \\ &= \left[\frac{1}{m} \sum_{i=1}^{m} \left(\frac{1}{2} \| h_{W,b}(x^{(i)}) - y^{(i)} \|^2 \right) \right] + \frac{\lambda}{2} \sum_{l=1}^{n_l - 1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_l + 1} \left(W_{ji}^{(l)} \right)^2 \end{aligned}$$

Backward Propagation from UFLDL

- One iteration of gradient descent updates the parameters W, b as follows:

$$
\begin{aligned}
W_{ij}^{(l)} &= W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \\
b_i^{(l)} &= b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b)
\end{aligned}
$$

- The derivative of the overall cost function $J(W, b)$:

$$
\begin{aligned}
\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)} \\
\frac{\partial}{\partial b_i^{(l)}} J(W, b) &= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W, b; x^{(i)}, y^{(i)})
\end{aligned}
$$

Backward Propagation from UFLDL

- One iteration of gradient descent updates the parameters W, b as follows:

$$
\begin{aligned}
W_{ij}^{(l)} &= W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \\
b_i^{(l)} &= b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b)
\end{aligned}
$$

- The derivative of the overall cost function $J(W, b)$:

$$
\begin{aligned}
\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)} \\
\frac{\partial}{\partial b_i^{(l)}} J(W, b) &= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W, b; x^{(i)}, y^{(i)})
\end{aligned}
$$

Backward Propagation from UFLDL

- One iteration of gradient descent updates the parameters W, b as follows:

$$
\begin{aligned}
W_{ij}^{(l)} &= W_{ij}^{(l)} - \alpha \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) \\
b_i^{(l)} &= b_i^{(l)} - \alpha \frac{\partial}{\partial b_i^{(l)}} J(W, b)
\end{aligned}
$$

- The derivative of the overall cost function $J(W, b)$:

$$
\begin{aligned}
\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b) &= \left[\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x^{(i)}, y^{(i)}) \right] + \lambda W_{ij}^{(l)} \\
\frac{\partial}{\partial b_i^{(l)}} J(W, b) &= \frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial b_i^{(l)}} J(W, b; x^{(i)}, y^{(i)})
\end{aligned}
$$

Backward Propagation from UFLDL

- The intuition behind the backpropagation algorithm is as follows.
  1. Given a training example (x, y): run a "forward pass" to compute all the activations.
  2. Then, for each node i in layer l: compute an "error term" $\delta_i^{(i)}$.
  3. For an output node, we can measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(n_l)}$ (where layer n_l is the output layer).
  4. For hidden nodes, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input.

Backward Propagation from UFLDL

- The intuition behind the backpropagation algorithm is as follows.
  1. Given a training example (x, y): run a "forward pass" to compute all the activations.
  2. Then, for each node i in layer l: compute an "error term" $\delta_i^{(i)}$.
  3. For an output node, we can measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(n_l)}$ (where layer n_l is the output layer).
  4. For hidden nodes, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input.

Backward Propagation from UFLDL

- The intuition behind the backpropagation algorithm is as follows.
  1. Given a training example (x, y): run a "forward pass" to compute all the activations.
  2. Then, for each node i in layer l: compute an "error term" $\delta_i^{(;)}$.
  3. For an output node, we can measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(n_l)}$ (where layer n_l is the output layer).
  4. For hidden nodes, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input.

Backward Propagation from UFLDL

- The intuition behind the backpropagation algorithm is as follows.
  1. Given a training example (x, y): run a "forward pass" to compute all the activations.
  2. Then, for each node i in layer l: compute an "error term" $\delta_i^{(;)}$.
  3. For an output node, we can measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(n_l)}$ (where layer n_l is the output layer).
  4. For hidden nodes, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input.

Backward Propagation from UFLDL

- The intuition behind the backpropagation algorithm is as follows.
  1. Given a training example (x, y): run a "forward pass" to compute all the activations.
  2. Then, for each node i in layer l: compute an "error term" $\delta_i^{(;)}$.
  3. For an output node, we can measure the difference between the network's activation and the true target value, and use that to define $\delta_i^{(n_l)}$ (where layer n_l is the output layer).
  4. For hidden nodes, we will compute $\delta_i^{(l)}$ based on a weighted average of the error terms of the nodes that uses $a_i^{(l)}$ as an input.

Backward Propagation from UFLDL

In detail, here is the backpropagation algorithm:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit i in layer n_l (the output layer), set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, For each node i in layer l, set

$$\delta_i^l = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

4. Compute the desired partial derivatives:

$$\begin{aligned} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) &= a_j^{(l)} \delta_i^{(l+1)} \\ \frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) &= \delta_i^{(l+1)} \end{aligned}$$

Backward Propagation from UFLDL

In detail, here is the backpropagation algorithm:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit i in layer n_l (the output layer), set
$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$
3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, For each node i in layer l, set
$$\delta_i^l = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$
4. Compute the desired partial derivatives:
$$\begin{aligned} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) &= a_j^{(l)} \delta_i^{(l+1)} \\ \frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) &= \delta_i^{(l+1)} \end{aligned}$$

Backward Propagation from UFLDL

In detail, here is the backpropagation algorithm:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit i in layer n_l (the output layer), set

$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, For each node i in layer l, set

$$\delta_i^l = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$

4. Compute the desired partial derivatives:

$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}$$

$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}$$

Backward Propagation from UFLDL

In detail, here is the backpropagation algorithm:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit i in layer n_l (the output layer), set
$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$
3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, For each node i in layer l, set
$$\delta_i^l = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$
4. Compute the desired partial derivatives:
$$\frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) = a_j^{(l)} \delta_i^{(l+1)}$$
$$\frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) = \delta_i^{(l+1)}$$

Backward Propagation from UFLDL

In detail, here is the backpropagation algorithm:

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, and so on up to the output layer L_{n_l}.
2. For each output unit i in layer n_l (the output layer), set
$$\delta_i^{(n_l)} = \frac{\partial}{\partial z_i^{(n_l)}} \frac{1}{2} \|y - h_{W,b}(x)\|^2 = -(y_i - a_i^{(n_l)}) \cdot f'(z_i^{(n_l)})$$
3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, For each node i in layer l, set
$$\delta_i^l = \left(\sum_{j=1}^{s_{l+1}} W_{ji}^{(l)} \delta_j^{(l+1)} \right) f'(z_i^{(l)})$$
4. Compute the desired partial derivatives:
$$\begin{aligned} \frac{\partial}{\partial W_{ij}^{(l)}} J(W, b; x, y) &= a_j^{(l)} \delta_i^{(l+1)} \\ \frac{\partial}{\partial b_i^{(l)}} J(W, b; x, y) &= \delta_i^{(l+1)} \end{aligned}$$

Backward Propagation from UFLDL

We can also re-write the algorithm using matrix-vectorial notation. ("$\bullet$" denote the element-wise product operator)

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}
2. For the output layer (layer n_l), set:

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, set

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

4. Compute the desired partial derivatives:

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T \quad (1)$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)} \quad (2)$$

Backward Propagation from UFLDL

We can also re-write the algorithm using matrix-vectorial notation. ("$\bullet$" denote the element-wise product operator)

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}
2. For the output layer (layer n_l), set:

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, set

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

4. Compute the desired partial derivatives:

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T \quad (1)$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)} \quad (2)$$

Backward Propagation from UFLDL

We can also re-write the algorithm using matrix-vectorial notation. ("$\bullet$" denote the element-wise product operator)

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}
2. For the output layer (layer n_l), set:

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, set

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

4. Compute the desired partial derivatives:

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T \quad (1)$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)} \quad (2)$$

Backward Propagation from UFLDL

We can also re-write the algorithm using matrix-vectorial notation. ("$\bullet$" denote the element-wise product operator)

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}
2. For the output layer (layer n_l), set:

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, set

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

4. Compute the desired partial derivatives:

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T \quad (1)$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)} \quad (2)$$

Backward Propagation from UFLDL

We can also re-write the algorithm using matrix-vectorial notation. ("$\bullet$" denote the element-wise product operator)

1. Perform a feedforward pass, computing the activations for layers L_2, L_3, up to the output layer L_{n_l}
2. For the output layer (layer n_l), set:

$$\delta^{(n_l)} = -(y - a^{(n_l)}) \bullet f'(z^{(n_l)})$$

3. For $l = n_l - 1, n_l - 2, n_l - 3, \ldots, 2$, set

$$\delta^{(l)} = \left((W^{(l)})^T \delta^{(l+1)}\right) \bullet f'(z^{(l)})$$

4. Compute the desired partial derivatives:

$$\nabla_{W^{(l)}} J(W, b; x, y) = \delta^{(l+1)} (a^{(l)})^T \quad (1)$$

$$\nabla_{b^{(l)}} J(W, b; x, y) = \delta^{(l+1)} \quad (2)$$

Q&A

Q&A

References

Ronan Collobert. Large scale machine learning. 2004.

George Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems (MCSS)*, 2 (4):303–314, 1989.

Charles Dugas, Yoshua Bengio, François Bélisle, Claude Nadeau, and René Garcia. Incorporating second-order functional knowledge for better option pricing. *Advances in neural information processing systems*, pages 472–478, 2001.

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

References ii

Ian J Goodfellow, David Warde-Farley, Mehdi Mirza, Aaron C Courville, and Yoshua Bengio. Maxout networks. *ICML (3)*, 28:1319–1327, 2013.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

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

Kevin Jarrett, Koray Kavukcuoglu, Yann LeCun, et al. What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2146–2153. IEEE, 2009.

References iii

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. Rectifier nonlinearities improve neural network acoustic models. In *Proc. ICML*, volume 30, 2013.

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

Hinton G. Rumelhard, D. and R. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–538, 1986.