Lecture 3
Feedforward Networks and Backpropagation

CMSC 35246: Deep Learning

Shubhendu Trivedi
&
Risi Kondor

University of Chicago

April 3, 2017

- Things we will look at today
  - Recap of Logistic Regression

- Things we will look at today
  - Recap of Logistic Regression
  - Going from one neuron to Feedforward Networks

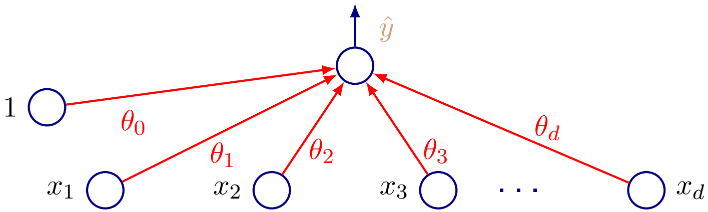
- Things we will look at today
  - Recap of Logistic Regression
  - Going from one neuron to Feedforward Networks
  - Example: Learning XOR

- Things we will look at today
  - Recap of Logistic Regression
  - Going from one neuron to Feedforward Networks
  - Example: Learning XOR
  - Cost Functions, Hidden unit types, output types

- Things we will look at today
  - Recap of Logistic Regression
  - Going from one neuron to Feedforward Networks
  - Example: Learning XOR
  - Cost Functions, Hidden unit types, output types
  - Universality Results and Architectural Considerations

- Things we will look at today
  - Recap of Logistic Regression
  - Going from one neuron to Feedforward Networks
  - Example: Learning XOR
  - Cost Functions, Hidden unit types, output types
  - Universality Results and Architectural Considerations
  - Backpropagation

Recap: The Logistic Function (Single Neuron)



$$p(y = 1|\mathbf{x}) = \frac{1}{1 + \exp(-\theta_0 - \theta^T \mathbf{x})}$$

Likelihood under the Logistic Model

$$p(y_i|\mathbf{x};\theta) = \begin{cases} \sigma(\theta_0 + \theta^T\mathbf{x}_i) \text{ if } y_i = 1 \\ 1 - \sigma(\theta_0 + \theta^T\mathbf{x}_i) \text{ if } y_i = 0 \end{cases}$$

- We can rewrite this as:

Likelihood under the Logistic Model

$$p(y_i|\mathbf{x};\theta) = \begin{cases} \sigma(\theta_0 + \theta^T \mathbf{x}_i) \text{ if } y_i = 1 \\ 1 - \sigma(\theta_0 + \theta^T \mathbf{x}_i) \text{ if } y_i = 0 \end{cases}$$

- We can rewrite this as:

$$p(y_i|\mathbf{x}_i;\theta) = \sigma(\theta_0 + \theta^T \mathbf{x}_i)^{y_i} (1 - \sigma(\theta_0 + \theta^T \mathbf{x}_i))^{1-y_i}$$

Likelihood under the Logistic Model

$$p(y_i|\mathbf{x};\theta) = \begin{cases} \sigma(\theta_0 + \theta^T \mathbf{x}_i) \text{ if } y_i = 1 \\ 1 - \sigma(\theta_0 + \theta^T \mathbf{x}_i) \text{ if } y_i = 0 \end{cases}$$

- We can rewrite this as:

$$p(y_i|\mathbf{x}_i;\theta) = \sigma(\theta_0 + \theta^T \mathbf{x}_i)^{y_i} (1 - \sigma(\theta_0 + \theta^T \mathbf{x}_i))^{1-y_i}$$

- The log-likelihood of θ (cross-entropy!):

$$\log p(Y|X;\theta) = \sum_{i=1}^{N} \log p(y_i|\mathbf{x}_i;\theta)$$

$$= \sum_{i=1}^{N} y_i \log \sigma(\theta_0 + \theta^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\theta_0 + \theta^T \mathbf{x}_i))$$

The Maximum Likelihood Solution

$$\log p(Y|X;\theta) = \sum_{i=1}^{N} y_i \log \sigma(\theta_0 + \theta^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\theta_0 + \theta^T \mathbf{x}_i))$$

The Maximum Likelihood Solution

$$\log p(Y|X;\theta) = \sum_{i=1}^{N} y_i \log \sigma(\theta_0+\theta^T\mathbf{x}_i)+(1-y_i)\log(1-\sigma(\theta_0+\theta^T\mathbf{x}_i))$$

- Setting derivatives to zero:

The Maximum Likelihood Solution

$$\log p(Y|X;\theta) = \sum_{i=1}^{N} y_i \log \sigma(\theta_0 + \theta^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\theta_0 + \theta^T \mathbf{x}_i))$$

- Setting derivatives to zero:

$$\frac{\partial \log p(Y|X;\theta)}{\partial \theta_0} = \sum_{i=1}^{N} (y_i - \sigma(\theta_0 + \theta^T \mathbf{x}_i)) = 0$$

$$\frac{\partial \log p(Y|X;\theta)}{\partial \theta_j} = \sum_{i=1}^{N} (y_i - \sigma(\theta_0 + \theta^T \mathbf{x}_i)) \mathbf{x}_{i,j} = 0$$

The Maximum Likelihood Solution

$$\log p(Y|X;\theta) = \sum_{i=1}^{N} y_i \log \sigma(\theta_0 + \theta^T \mathbf{x}_i) + (1 - y_i) \log(1 - \sigma(\theta_0 + \theta^T \mathbf{x}_i))$$

- Setting derivatives to zero:

$$\frac{\partial \log p(Y|X;\theta)}{\partial \theta_0} = \sum_{i=1}^{N} (y_i - \sigma(\theta_0 + \theta^T \mathbf{x}_i)) = 0$$

$$\frac{\partial \log p(Y|X;\theta)}{\partial \theta_j} = \sum_{i=1}^{N} (y_i - \sigma(\theta_0 + \theta^T \mathbf{x}_i)) \mathbf{x}_{i,j} = 0$$

- Can treat $y_i - p(y_i|\mathbf{x}_i) = y_i - \sigma(\theta_0 + \theta^T \mathbf{x}_i)$ as the prediction error

Finding Maxima

- No closed form solution for the Maximum Likelihood for this model!

Finding Maxima

- No closed form solution for the Maximum Likelihood for this model!
- But $\log p(Y|X;\mathbf{x})$ is jointly concave in all components of θ

Finding Maxima

- No closed form solution for the Maximum Likelihood for this model!
- But $\log p(Y|X;\mathbf{x})$ is jointly concave in all components of θ
- Or, equivalently, the error is convex

Finding Maxima

- No closed form solution for the Maximum Likelihood for this model!
- But $\log p(Y|X;\mathbf{x})$ is jointly concave in all components of θ
- Or, equivalently, the error is convex
- Gradient Descent/ascent (descent on $-\log p(y|\mathbf{x};\theta)$, log loss)

Gradient Descent Solution

- Objective is the average log-loss

Gradient Descent Solution

- Objective is the average log-loss

$$-\frac{1}{N}\sum_{i=1}^{N}\log p(y_i|\mathbf{x}_i;\theta)$$

Gradient Descent Solution

- Objective is the average log-loss

$$-\frac{1}{N}\sum_{i=1}^{N}\log p(y_i|\mathbf{x}_i;\theta)$$

- Gradient update:

$$\theta^{(t+1)} := \theta^t + \frac{\eta_t}{N}\frac{\partial}{\partial\theta}\sum_i \log p(y_i|\mathbf{x}_i;\theta^{(t)})$$

Gradient Descent Solution

- Objective is the average log-loss

$$-\frac{1}{N}\sum_{i=1}^{N}\log p(y_i|\mathbf{x}_i;\theta)$$

- Gradient update:

$$\theta^{(t+1)} := \theta^t + \frac{\eta_t}{N}\frac{\partial}{\partial\theta}\sum_i \log p(y_i|\mathbf{x}_i;\theta^{(t)})$$

- Gradient on one example:

$$\frac{\partial}{\partial\theta}\log p(y_i|\mathbf{x}_i;\theta) = (y_i - \sigma(\theta^T\mathbf{x}_i))\mathbf{x}_i$$

Gradient Descent Solution

- Objective is the average log-loss

$$-\frac{1}{N}\sum_{i=1}^{N}\log p(y_i|\mathbf{x}_i;\theta)$$

- Gradient update:

$$\theta^{(t+1)} := \theta^{t} + \frac{\eta_t}{N}\frac{\partial}{\partial\theta}\sum_{i}\log p(y_i|\mathbf{x}_i;\theta^{(t)})$$

- Gradient on one example:

$$\frac{\partial}{\partial\theta}\log p(y_i|\mathbf{x}_i;\theta) = (y_i - \sigma(\theta^{T}\mathbf{x}_i))\mathbf{x}_i$$

- Above is batch gradient descent

Feedforward Networks

Introduction

- **Goal**: Approximate some unknown ideal function $f^* : \mathcal{X} \to \mathcal{Y}$

Introduction

- **Goal**: Approximate some unknown ideal function $f^* : \mathcal{X} \rightarrow \mathcal{Y}$
- **Ideal classifier**: $y = f^*(\mathbf{x})$ with $\mathbf{x}$ and category y

Introduction

- **Goal**: Approximate some unknown ideal function $f^* : \mathcal{X} \rightarrow \mathcal{Y}$
- **Ideal classifier**: $y = f^*(\mathbf{x})$ with $\mathbf{x}$ and category y
- **Feedforward Network**: Define parametric mapping $y = f(\mathbf{x}, \theta)$

Introduction

- **Goal**: Approximate some unknown ideal function $f^* : \mathcal{X} \to \mathcal{Y}$
- **Ideal classifier**: $y = f^*(\mathbf{x})$ with $\mathbf{x}$ and category y
- **Feedforward Network**: Define parametric mapping $y = f(\mathbf{x}, \theta)$
- Learn parameters θ to get a good approximation to f^* from available sample

Introduction

- **Goal**: Approximate some unknown ideal function $f^* : \mathcal{X} \to \mathcal{Y}$
- **Ideal classifier**: $y = f^*(\mathbf{x})$ with $\mathbf{x}$ and category y
- **Feedforward Network**: Define parametric mapping $y = f(\mathbf{x}, \theta)$
- Learn parameters θ to get a good approximation to f^* from available sample
- **Naming**: Information flow in function evaluation begins at input, flows through intermediate computations (that define the function), to produce the category

Introduction

- **Goal**: Approximate some unknown ideal function $f^*: \mathcal{X} \rightarrow \mathcal{Y}$
- **Ideal classifier**: $y = f^*(\mathbf{x})$ with $\mathbf{x}$ and category y
- **Feedforward Network**: Define parametric mapping $y = f(\mathbf{x}, \theta)$
- Learn parameters θ to get a good approximation to f^* from available sample
- **Naming**: Information flow in function evaluation begins at input, flows through intermediate computations (that define the function), to produce the category
- No feedback connections (Recurrent Networks!)

Introduction

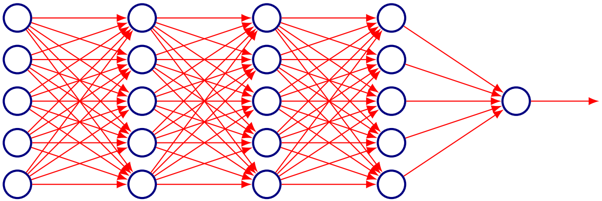
- Function f is a composition of many different functions

Introduction

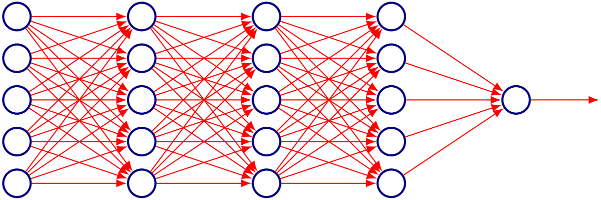
- Function f is a composition of many different functions
- e.g. $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$

Introduction

- Function f is a composition of many different functions
- e.g. $f(\mathbf{x}) = f^{(3)}(f^{(2)}(f^{(1)}(\mathbf{x})))$

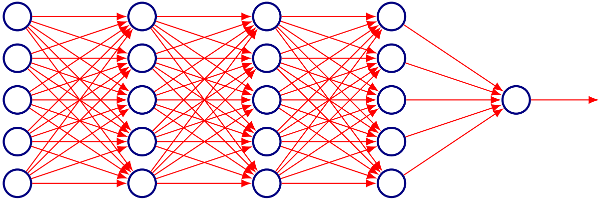


Introduction



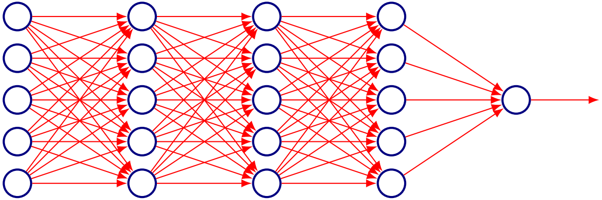
- Function composition can be described by a directed acyclic graph (hence feedforward *networks*)

Introduction



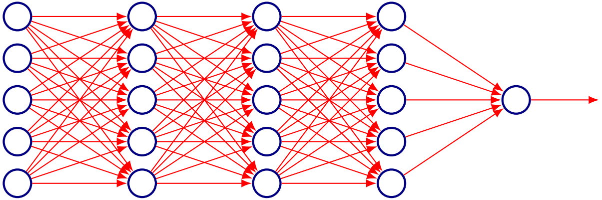
- Function composition can be described by a directed acyclic graph (hence feedforward *networks*)
- $f^{(1)}$ is the first layer, f^2 the second layer and so on.

Introduction



- Function composition can be described by a directed acyclic graph (hence feedforward *networks*)
- $f^{(1)}$ is the first layer, f^2 the second layer and so on.
- *Depth* is the maximum i in the function composition chain

Introduction



- Function composition can be described by a directed acyclic graph (hence feedforward *networks*)
- $f^{(1)}$ is the first layer, f^2 the second layer and so on.
- *Depth* is the maximum i in the function composition chain
- Final layer is called the *output* layer

Introduction

- **Training**: Optimize θ to drive $f(\mathbf{x};\theta)$ closer to $f^*(\mathbf{x})$

Introduction

- **Training**: Optimize θ to drive $f(\mathbf{x};\theta)$ closer to $f^*(\mathbf{x})$
- **Training Data**: f^* evaluated at different $\mathbf{x}$ instances (i.e. expected outputs)

Introduction

- **Training**: Optimize θ to drive $f(\mathbf{x};\theta)$ closer to $f^*(\mathbf{x})$
- **Training Data**: f^* evaluated at different $\mathbf{x}$ instances (i.e. expected outputs)
- Only specifies the output of the *output layers*

Introduction

- **Training**: Optimize θ to drive $f(\mathbf{x};\theta)$ closer to $f^*(\mathbf{x})$
- **Training Data**: f^* evaluated at different $\mathbf{x}$ instances (i.e. expected outputs)
- Only specifies the output of the *output layers*
- Output of intermediate layers is not specified by $\mathcal{D}$, hence the nomenclature *hidden layers*

Introduction

- **Training**: Optimize θ to drive $f(\mathbf{x};\theta)$ closer to $f^*(\mathbf{x})$
- **Training Data**: f^* evaluated at different $\mathbf{x}$ instances (i.e. expected outputs)
- Only specifies the output of the *output layers*
- Output of intermediate layers is not specified by $\mathcal{D}$, hence the nomenclature *hidden layers*
- **Neural**: Choices of $f^{(i)}$'s and layered organization, loosely inspired by neuroscience (first lecture)

Back to Linear Models

- $+$ve: Optimization is convex or closed form!

Back to Linear Models

- $+$ve: Optimization is convex or closed form!
- -ve: Model can't understand interaction between input variables!

Back to Linear Models

- $+$ve: Optimization is convex or closed form!
- -ve: Model can't understand interaction between input variables!
- **Extension**: Do nonlinear transformation $\mathbf{x} \to \phi(\mathbf{x})$; apply linear model to $\phi(\mathbf{x})$

Back to Linear Models

- +ve: Optimization is convex or closed form!
- -ve: Model can't understand interaction between input variables!
- **Extension**: Do nonlinear transformation $\mathbf{x} \to \phi(\mathbf{x})$; apply linear model to $\phi(\mathbf{x})$
- ϕ gives features or a *representation* for $\mathbf{x}$

Back to Linear Models

- +ve: Optimization is convex or closed form!
- -ve: Model can't understand interaction between input variables!
- **Extension**: Do nonlinear transformation $\mathbf{x} \rightarrow \phi(\mathbf{x})$; apply linear model to $\phi(\mathbf{x})$
- ϕ gives features or a *representation* for $\mathbf{x}$
- How do we choose ϕ?

Choosing ϕ

- **Option 1**: Use a generic ϕ

Choosing ϕ

- **Option 1**: Use a generic ϕ
- **Example**: Infinite dimensional ϕ implicitly used by kernel machines with RBF kernel

Choosing ϕ

- **Option 1**: Use a generic ϕ
- **Example**: Infinite dimensional ϕ implicitly used by kernel machines with RBF kernel
- **Positive**: Enough capacity to fit training data

Choosing ϕ

- **Option 1**: Use a generic ϕ
- **Example**: Infinite dimensional ϕ implicitly used by kernel machines with RBF kernel
- **Positive**: Enough capacity to fit training data
- **Negative**: Poor generalization for highly varying f^*

Choosing ϕ

- **Option 1**: Use a generic ϕ
- **Example**: Infinite dimensional ϕ implicitly used by kernel machines with RBF kernel
- **Positive**: Enough capacity to fit training data
- **Negative**: Poor generalization for highly varying f^*
- Prior used: Function is locally smooth.

Choosing ϕ

- **Option 2**: Engineer ϕ for problem

Choosing ϕ

- **Option 2**: Engineer ϕ for problem
- Still convex!

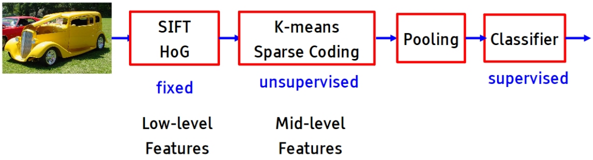


Illustration: Yann LeCun

Choosing ϕ

- **Option 3**: Learn ϕ from data
- Gives up on convexity
- Combines good points of first two approaches: ϕ can be highly generic and the engineering effort can go into architecture

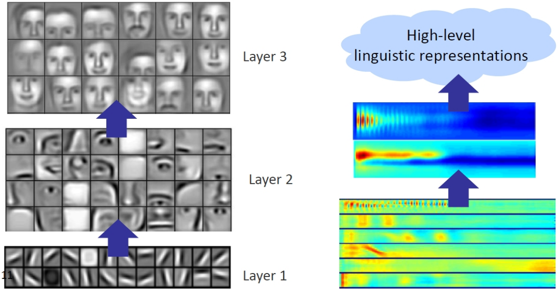


Figure: Honglak Lee

Design Decisions

- Need to choose optimizer, cost function and form of output

Design Decisions

- Need to choose optimizer, cost function and form of output
- Choosing activation functions

Design Decisions

- Need to choose optimizer, cost function and form of output
- Choosing activation functions
- Architecture design (number of layers etc)

Back to XOR

XOR

Exclusive-OR gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

- Let XOR be the target function $f^*(\mathbf{x})$ that we want to learn

XOR

Exclusive-OR gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

- Let XOR be the target function $f^*(\mathbf{x})$ that we want to learn
- We will adapt parameters θ for $f(\mathbf{x};\theta)$ to try and represent f^*

XOR

Exclusive-OR gate

A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

- Let XOR be the target function $f^*(\mathbf{x})$ that we want to learn
- We will adapt parameters θ for $f(\mathbf{x};\theta)$ to try and represent f^*
- Our Data:
  $(X, Y) = \{([0,0]^T, 0), ([0,1]^T, 1), ([1,0]^T, 1), ([1,1]^T, 0)\}$

XOR

- Our Data:
  $(X, Y) = \{([0,0]^T, 0), ([0,1]^T, 1), ([1,0]^T, 1), ([1,1]^T, 0)\}$

XOR

- Our Data:
  $(X, Y) = \{([0,0]^T, 0), ([0,1]^T, 1), ([1,0]^T, 1), ([1,1]^T, 0)\}$
- Not concerned with generalization, only want to fit this data

XOR

- Our Data:
  $(X,Y) = \{([0,0]^T,0),([0,1]^T,1),([1,0]^T,1),([1,1]^T,0)\}$
- Not concerned with generalization, only want to fit this data
- For simplicity consider the squared loss function

XOR

- Our Data:
  $(X, Y) = \{([0,0]^T, 0), ([0,1]^T, 1), ([1,0]^T, 1), ([1,1]^T, 0)\}$
- Not concerned with generalization, only want to fit this data
- For simplicity consider the squared loss function

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

XOR

- Our Data:
  $(X, Y) = \{([0,0]^T, 0), ([0,1]^T, 1), ([1,0]^T, 1), ([1,1]^T, 0)\}$
- Not concerned with generalization, only want to fit this data
- For simplicity consider the squared loss function

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

- Need to choose a form for $f(\mathbf{x}; \theta)$: Consider a linear model with θ being $\mathbf{w}$ and b

XOR

- Our Data:
  $(X, Y) = \{([0,0]^T, 0), ([0,1]^T, 1), ([1,0]^T, 1), ([1,1]^T, 0)\}$
- Not concerned with generalization, only want to fit this data
- For simplicity consider the squared loss function

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

- Need to choose a form for $f(\mathbf{x}; \theta)$: Consider a linear model with θ being $\mathbf{w}$ and b
- Our model $f(\mathbf{x}; \mathbf{w}, b) = \mathbf{x}^T \mathbf{w} + b$

Linear Model

- Recall previous lecture: Normal equations give $\mathbf{w} = 0$ and $b = \frac{1}{2}$

Linear Model

- Recall previous lecture: Normal equations give $\mathbf{w} = 0$ and $b = \frac{1}{2}$
- A linear model is not able to represent XOR, outputs 0.5 everywhere

Linear Model

- Recall previous lecture: Normal equations give $\mathbf{w} = 0$ and $b = \frac{1}{2}$
- A linear model is not able to represent XOR, outputs 0.5 everywhere

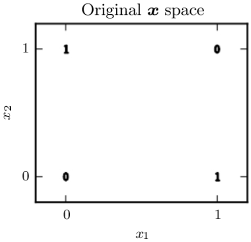


Figure: Goodfellow *et al.*

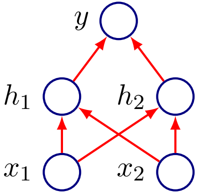
Solving XOR

- How can we solve the XOR problem?

Solving XOR

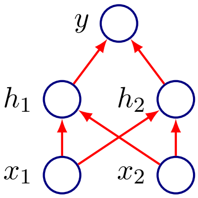
- How can we solve the XOR problem?
- Idea: Learn a different feature space in which a linear model will work

Solving XOR



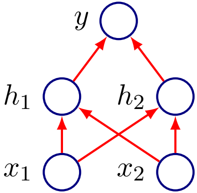
- Define a feedforward network with a vector of hidden units $\mathbf{h}$ computed by $f^{(1)}(\mathbf{x}; W, c)$

Solving XOR



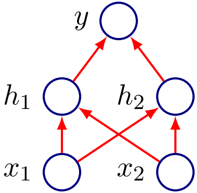
- Define a feedforward network with a vector of hidden units $\mathbf{h}$ computed by $f^{(1)}(\mathbf{x}; W, c)$
- Use hidden unit values as input for a second layer i.e. to compute output $y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$

Solving XOR



- Define a feedforward network with a vector of hidden units $\mathbf{h}$ computed by $f^{(1)}(\mathbf{x}; W, c)$
- Use hidden unit values as input for a second layer i.e. to compute output $y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$
- Complete model: $f(\mathbf{x}; W, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x}))$

Solving XOR



- Define a feedforward network with a vector of hidden units $\mathbf{h}$ computed by $f^{(1)}(\mathbf{x}; W, c)$
- Use hidden unit values as input for a second layer i.e. to compute output $y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$
- Complete model: $f(\mathbf{x}; W, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x}))$
- What should be $f^{(1)}$? Can it be linear?

Solving XOR

- Let us consider a non-linear activation $g(z) = \max\{0, z\}$

Solving XOR

- Let us consider a non-linear activation $g(z) = \max\{0, z\}$
- Our complete network model:

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

Solving XOR

- Let us consider a non-linear activation $g(z) = \max\{0, z\}$
- Our complete network model:

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

- Note: The activation above is applied element-wise

A Solution

- Let

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

A Solution

- Let
$$W = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}, b = 0$$
- Our design matrix is:
$$X = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

A Solution

- Compute the first layer output, by first calculating XW

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

A Solution

- Compute the first layer output, by first calculating XW

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

- Find $XW + \mathbf{c}$

A Solution

- Compute the first layer output, by first calculating XW

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

- Find $XW + \mathbf{c}$

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

A Solution

- Compute the first layer output, by first calculating XW

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

- Find $XW + \mathbf{c}$

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

- Note: Ignore the type mismatch

A Solution

- Next step: Rectify output

$$\max\{0, XW + \mathbf{c}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

A Solution

- Next step: Rectify output

$$\max\{0, XW + \mathbf{c}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

- Finally compute $\mathbf{w}^T \max\{0, XW + \mathbf{c}\} + b$

A Solution

- Next step: Rectify output

$$\max\{0, XW + \mathbf{c}\} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{bmatrix}$$

- Finally compute $\mathbf{w}^T \max\{0, XW + \mathbf{c}\} + b$

$$\begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

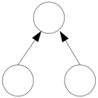
- Able to correctly classify every example in the set

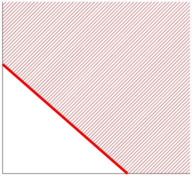
- Able to correctly classify every example in the set
- This is a hand coded; demonstrative example, hence clean

- Able to correctly classify every example in the set
- This is a hand coded; demonstrative example, hence clean
- For more complicated functions, we will proceed by using gradient based learning

An Aside:

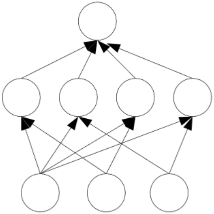
1 layer of trainable weights

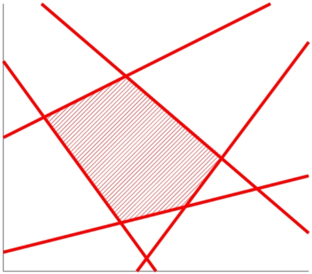




separating hyperplane

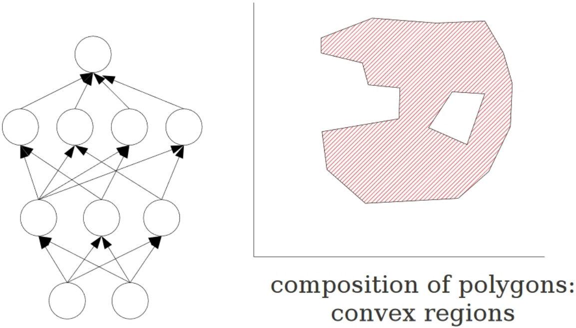
An Aside:





convex polygon region

An Aside:



- Designing and Training a Neural Network is not much different from training any other Machine Learning model with gradient descent

- Designing and Training a Neural Network is not much different from training any other Machine Learning model with gradient descent
- Largest difference: Most interesting loss functions become non-convex

- Designing and Training a Neural Network is not much different from training any other Machine Learning model with gradient descent
- Largest difference: Most interesting loss functions become non-convex
- Unlike in convex optimization, no convergence guarantees

- Designing and Training a Neural Network is not much different from training any other Machine Learning model with gradient descent
- Largest difference: Most interesting loss functions become non-convex
- Unlike in convex optimization, no convergence guarantees
- To apply gradient descent: Need to specify cost function, and output representation

Cost Functions

Cost Functions

- Choice similar to parameteric models from earlier: Define a distribution $p(\mathbf{y}|\mathbf{x};\theta)$ and use principle of maximum likelihood

Cost Functions

- Choice similar to parameteric models from earlier: Define a distribution $p(\mathbf{y}|\mathbf{x};\theta)$ and use principle of maximum likelihood
- We can just use cross entropy between training data and the model's predictions as the cost function:

Cost Functions

- Choice similar to parameteric models from earlier: Define a distribution $p(\mathbf{y}|\mathbf{x};\theta)$ and use principle of maximum likelihood
- We can just use cross entropy between training data and the model's predictions as the cost function:

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

Cost Functions

- Choice similar to parameteric models from earlier: Define a distribution $p(\mathbf{y}|\mathbf{x};\theta)$ and use principle of maximum likelihood
- We can just use cross entropy between training data and the model's predictions as the cost function:

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

- Specific form changes depending on form of $\log p_{model}$

Cost Functions

- Choice similar to parameteric models from earlier: Define a distribution $p(\mathbf{y}|\mathbf{x};\theta)$ and use principle of maximum likelihood
- We can just use cross entropy between training data and the model's predictions as the cost function:

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

- Specific form changes depending on form of $\log p_{model}$
- Example: If $p_{model}(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x};\theta), I)$, then we recover:

Cost Functions

- Choice similar to parameteric models from earlier: Define a distribution $p(\mathbf{y}|\mathbf{x};\theta)$ and use principle of maximum likelihood
- We can just use cross entropy between training data and the model's predictions as the cost function:

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

- Specific form changes depending on form of $\log p_{model}$
- Example: If $p_{model}(\mathbf{y}|\mathbf{x}) = \mathcal{N}(\mathbf{y}; f(\mathbf{x};\theta), I)$, then we recover:

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

Cost Functions

- Advantage: Need to specify $p(\mathbf{y}|\mathbf{x})$, and automatically get a cost function $\log p(\mathbf{y}|\mathbf{x})$

Cost Functions

- Advantage: Need to specify $p(\mathbf{y}|\mathbf{x})$, and automatically get a cost function $\log p(\mathbf{y}|\mathbf{x})$
- Choice of output units is very important for choice of cost function

Output Units

Linear Units

- Given features h, a layer of linear output units gives:

$$\hat{y} = W^T h + b$$

Linear Units

- Given features h, a layer of linear output units gives:

$$\hat{y} = W^T h + b$$

- Often used to produce the mean of a conditional Gaussian distribution:

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

Linear Units

- Given features h, a layer of linear output units gives:

$$\hat{y} = W^T h + b$$

- Often used to produce the mean of a conditional Gaussian distribution:

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

- Maximizing log-likelihood $\implies$ minimizing squared error

Sigmoid Units

- Task: Predict a binary variable y

Sigmoid Units

- Task: Predict a binary variable y
- Use a sigmoid unit:

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

Sigmoid Units

- Task: Predict a binary variable y
- Use a sigmoid unit:

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

- Cost:

$$J(\theta) = -\log p(y|\mathbf{x}) = -\log \sigma((2y-1)(\mathbf{w}^T \mathbf{h} + b))$$

Sigmoid Units

- Task: Predict a binary variable y
- Use a sigmoid unit:

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

- Cost:

$$J(\theta) = -\log p(y|\mathbf{x}) = -\log \sigma((2y-1)(\mathbf{w}^T\mathbf{h} + b))$$

- Positive: Only saturates when model already has right answer i.e. when $y = 1$ and $(\mathbf{w}^T\mathbf{h} + b)$ is very positive and vice versa

Sigmoid Units

- Task: Predict a binary variable y
- Use a sigmoid unit:

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

- Cost:

$$J(\theta) = -\log p(y|\mathbf{x}) = -\log \sigma((2y-1)(\mathbf{w}^T\mathbf{h} + b))$$

- Positive: Only saturates when model already has right answer i.e. when $y = 1$ and $(\mathbf{w}^T\mathbf{h} + b)$ is very positive and vice versa
- When $(\mathbf{w}^T\mathbf{h} + b)$ has wrong sign, a good gradient is returned

Softmax Units

- Need to produce a vector $\hat{\mathbf{y}}$ with $\hat{y}_i = p(y = i|\mathbf{x})$

Softmax Units

- Need to produce a vector $\hat{\mathbf{y}}$ with $\hat{y}_i = p(y = i|\mathbf{x})$
- Linear layer first produces unnormalized log probabilities: $\mathbf{z} = W^T\mathbf{h} + \mathbf{b}$

Softmax Units

- Need to produce a vector $\hat{\mathbf{y}}$ with $\hat{y}_i = p(y = i|\mathbf{x})$
- Linear layer first produces unnormalized log probabilities: $\mathbf{z} = W^T\mathbf{h} + \mathbf{b}$
- Softmax:

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

Softmax Units

- Need to produce a vector $\hat{\mathbf{y}}$ with $\hat{y}_i = p(y = i|\mathbf{x})$
- Linear layer first produces unnormalized log probabilities: $\mathbf{z} = W^T\mathbf{h} + \mathbf{b}$
- Softmax:

$$\text{softmax}(\mathbf{z})_i = \frac{\exp(z_i)}{\sum_j \exp(z_j)}$$

- Log of the softmax (since we wish to maximize $p(y = i; \mathbf{z})$):

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

Benefits

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

Benefits

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

- z_i term never saturates, making learning easier

Benefits

$$\log \mathsf{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

- z_i term never saturates, making learning easier
- Maximizing log-likelihood encourages z_i to be pushed up, while encouraging all $\mathbf{z}$ to be pushed down (Softmax encourages competition)

Benefits

$$\log \mathsf{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

- z_i term never saturates, making learning easier
- Maximizing log-likelihood encourages z_i to be pushed up, while encouraging all $\mathbf{z}$ to be pushed down (Softmax encourages competition)
- More intuition: Think of $\log \sum_j \exp(z_j) \approx \max_j z_j$ (why?)

Benefits

$$\log \mathsf{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

- z_i term never saturates, making learning easier
- Maximizing log-likelihood encourages z_i to be pushed up, while encouraging all $\mathbf{z}$ to be pushed down (Softmax encourages competition)
- More intuition: Think of $\log \sum_j \exp(z_j) \approx \max_j z_j$ (why?)
- log-likelihood cost function ($\sim z_i - \max_j z_j$) strongly penalizes the most active *incorrect* prediction

Benefits

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

- z_i term never saturates, making learning easier
- Maximizing log-likelihood encourages z_i to be pushed up, while encouraging all $\mathbf{z}$ to be pushed down (Softmax encourages competition)
- More intuition: Think of $\log \sum_j \exp(z_j) \approx \max_j z_j$ (why?)
- log-likelihood cost function ($\sim z_i - \max_j z_j$) strongly penalizes the most active *incorrect* prediction
- If model already has correct answer then $\log \sum_j \exp(z_j) \approx \max_j z_j$ and z_i will roughly cancel out

Benefits

$$\log \text{softmax}(\mathbf{z})_i = z_i - \log \sum_j \exp(z_j)$$

- z_i term never saturates, making learning easier
- Maximizing log-likelihood encourages z_i to be pushed up, while encouraging all $\mathbf{z}$ to be pushed down (Softmax encourages competition)
- More intuition: Think of $\log \sum_j \exp(z_j) \approx \max_j z_j$ (why?)
- log-likelihood cost function ($\sim z_i - \max_j z_j$) strongly penalizes the most active *incorrect* prediction
- If model already has correct answer then $\log \sum_j \exp(z_j) \approx \max_j z_j$ and z_i will roughly cancel out
- Progress of learning is dominated by incorrectly classified examples

Hidden Units

Hidden Units

- Accept input $\mathbf{x}$

Hidden Units

- Accept input $\mathbf{x} \rightarrow$ compute affine transformation $\mathbf{z} = W^T\mathbf{x} + \mathbf{b}$

Hidden Units

- Accept input $\mathbf{x}$ $\rightarrow$ compute affine transformation $\mathbf{z} = W^T\mathbf{x} + \mathbf{b}$ $\rightarrow$ apply *elementwise* non-linear function $g(z)$

Hidden Units

- Accept input $\mathbf{x}$ $\rightarrow$ compute affine transformation $\mathbf{z} = W^T\mathbf{x} + \mathbf{b}$ $\rightarrow$ apply *elementwise* non-linear function $g(z)$ $\rightarrow$ obtain output $g(\mathbf{z})$

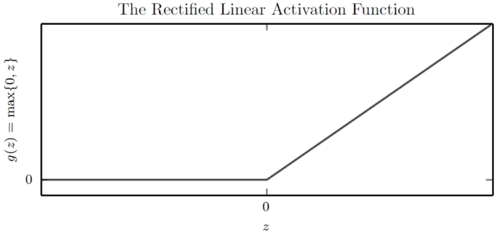
Hidden Units

- Accept input $\mathbf{x} \rightarrow$ compute affine transformation $\mathbf{z} = W^T\mathbf{x} + \mathbf{b} \rightarrow$ apply *elementwise* non-linear function $g(z)$ $\rightarrow$ obtain output $g(\mathbf{z})$
- Choices for g?

Hidden Units

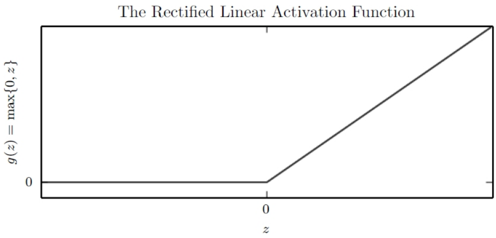
- Accept input $\mathbf{x}$ $\rightarrow$ compute affine transformation $\mathbf{z} = W^T\mathbf{x} + \mathbf{b}$ $\rightarrow$ apply *elementwise* non-linear function $g(z)$ $\rightarrow$ obtain output $g(\mathbf{z})$
- Choices for g?
- Design of Hidden units is an active area of research

Rectified Linear Units



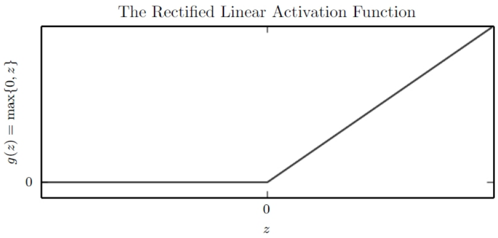
- Activation function: $g(z) = \max\{0, z\}$ with $z \in \mathbb{R}$

Rectified Linear Units



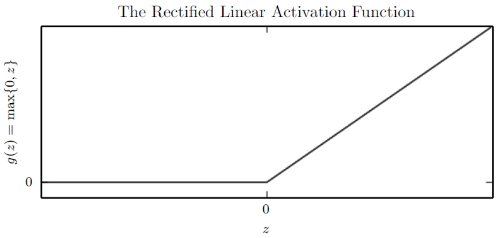
- Activation function: $g(z) = \max\{0, z\}$ with $z \in \mathbb{R}$
- On top of a affine transformation $\max\{0, W\mathbf{x} + \mathbf{b}\}$

Rectified Linear Units



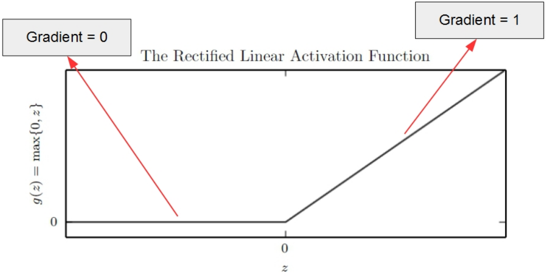
- Activation function: $g(z) = \max\{0, z\}$ with $z \in \mathbb{R}$
- On top of a affine transformation $\max\{0, W\mathbf{x} + \mathbf{b}\}$
- Two layer network: First layer $\max\{0, W_1^T\mathbf{x} + \mathbf{b}_1\}$

Rectified Linear Units



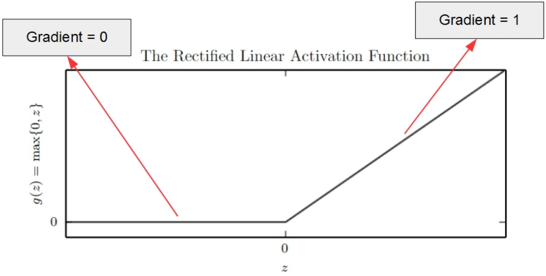
- Activation function: $g(z) = \max\{0, z\}$ with $z \in \mathbb{R}$
- On top of a affine transformation $\max\{0, W\mathbf{x} + \mathbf{b}\}$
- Two layer network: First layer $\max\{0, W_1^T\mathbf{x} + \mathbf{b}_1\}$
- Second layer: $W_2^T \max\{0, W_1^T\mathbf{x} + \mathbf{b}_1\} + \mathbf{b}_2$

Rectified Linear Units



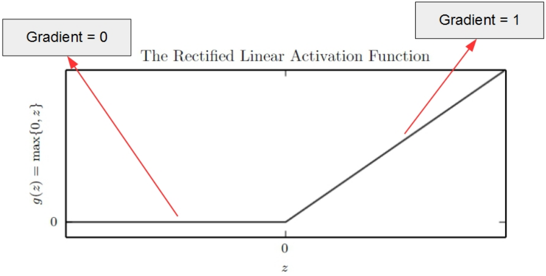
- Similar to linear units. Easy to optimize!

Rectified Linear Units



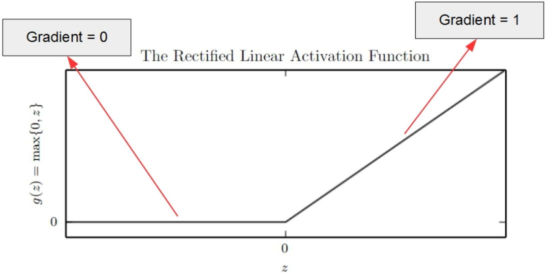
- Similar to linear units. Easy to optimize!
- Give large and *consistent* gradients when active

Rectified Linear Units



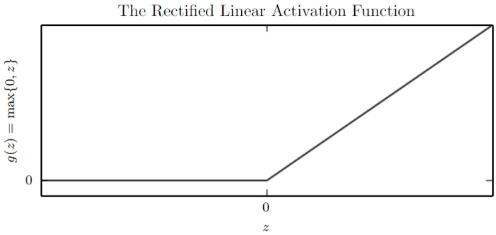
- Similar to linear units. Easy to optimize!
- Give large and *consistent* gradients when active
- **Good practice:** Initialize $\mathbf{b}$ to a small positive value (e.g. 0.1)

Rectified Linear Units



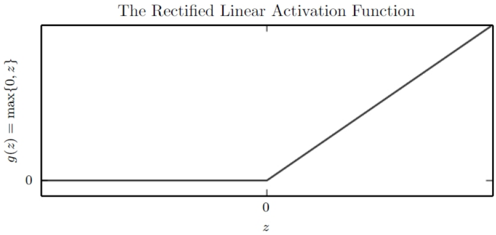
- Similar to linear units. Easy to optimize!
- Give large and *consistent* gradients when active
- **Good practice:** Initialize $\mathbf{b}$ to a small positive value (e.g. 0.1)
- Ensures units are initially active for most inputs and derivatives can pass through

Rectified Linear Units



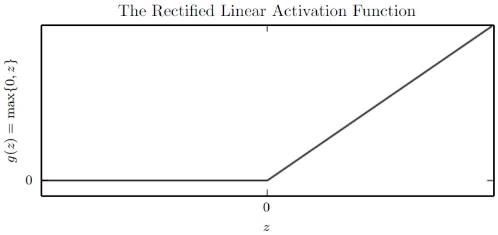
- Not everywhere differentiable. Is this a problem?

Rectified Linear Units



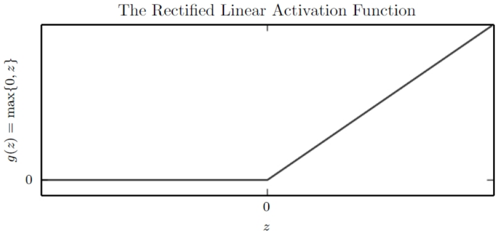
- Not everywhere differentiable. Is this a problem?
  - In practice not a problem. Return one sided derivatives at $z = 0$

Rectified Linear Units



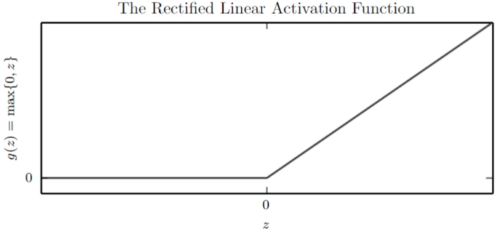
- Not everywhere differentiable. Is this a problem?
  - In practice not a problem. Return one sided derivatives at $z = 0$
  - Gradient based optimization is subject to numerical error anyway

Rectified Linear Units



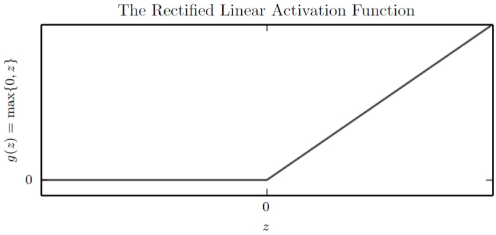
- Positives:

Rectified Linear Units



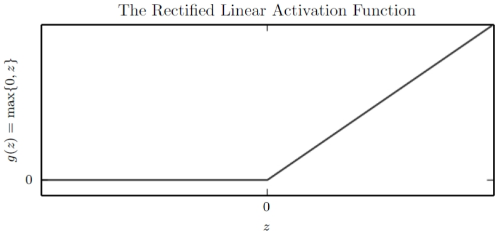
- Positives:
  - Gives large and *consistent* gradients (does not saturate) when active

Rectified Linear Units



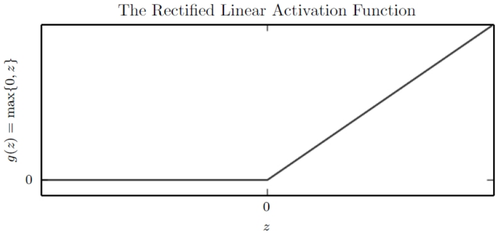
- Positives:
  - Gives large and *consistent* gradients (does not saturate) when active
  - Efficient to optimize, converges much faster than sigmoid or tanh

Rectified Linear Units



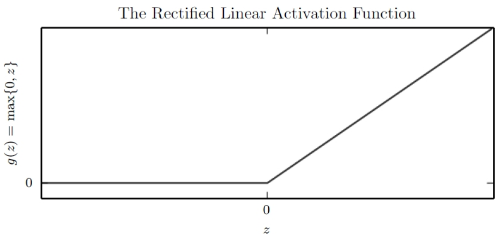
- Positives:
  - Gives large and *consistent* gradients (does not saturate) when active
  - Efficient to optimize, converges much faster than sigmoid or tanh
- Negatives:

Rectified Linear Units



- Positives:
  - Gives large and *consistent* gradients (does not saturate) when active
  - Efficient to optimize, converges much faster than sigmoid or tanh
- Negatives:
  - Non zero centered output

Rectified Linear Units



- Positives:
  - Gives large and *consistent* gradients (does not saturate) when active
  - Efficient to optimize, converges much faster than sigmoid or tanh
- Negatives:
  - Non zero centered output
  - Units "die" i.e. when inactive they will never update

Generalized Rectified Linear Units

- Get a non-zero slope when $z_i < 0$

Generalized Rectified Linear Units

- Get a non-zero slope when $z_i < 0$
- $g(z, a)_i = \max\{0, z_i\} + a_i \min\{0, z_i\}$

Generalized Rectified Linear Units

- Get a non-zero slope when $z_i < 0$
- $g(z, a)_i = \max\{0, z_i\} + a_i \min\{0, z_i\}$
  - **Absolute value rectification:** (Jarret *et al*, 2009) $a_i = 1$ gives $g(z) = |z|$

Generalized Rectified Linear Units

- Get a non-zero slope when $z_i < 0$
- $g(z, a)_i = \max\{0, z_i\} + a_i \min\{0, z_i\}$
  - **Absolute value rectification:** (Jarret *et al*, 2009) $a_i = 1$ gives $g(z) = |z|$
  - **Leaky ReLU:** (Maas *et al.*, 2013) Fix a_i to a small value e.g. 0.01

Generalized Rectified Linear Units

- Get a non-zero slope when $z_i < 0$
- $g(z, a)_i = \max\{0, z_i\} + a_i \min\{0, z_i\}$
  - **Absolute value rectification:** (Jarret *et al*, 2009) $a_i = 1$ gives $g(z) = |z|$
  - **Leaky ReLU:** (Maas *et al.*, 2013) Fix a_i to a small value e.g. 0.01
  - **Parametric ReLU:** (He *et al.*, 2015) Learn a_i

Generalized Rectified Linear Units

- Get a non-zero slope when $z_i < 0$
- $g(z, a)_i = \max\{0, z_i\} + a_i \min\{0, z_i\}$
  - **Absolute value rectification:** (Jarret *et al*, 2009) $a_i = 1$ gives $g(z) = |z|$
  - **Leaky ReLU:** (Maas *et al.*, 2013) Fix a_i to a small value e.g. 0.01
  - **Parametric ReLU:** (He *et al.*, 2015) Learn a_i
  - **Randomized ReLU:** (Xu *et al.*, 2015) Sample a_i from a fixed range during training, fix during testing

Generalized Rectified Linear Units

- Get a non-zero slope when $z_i < 0$
- $g(z, a)_i = \max\{0, z_i\} + a_i \min\{0, z_i\}$
  - **Absolute value rectification:** (Jarret *et al*, 2009) $a_i = 1$ gives $g(z) = |z|$
  - **Leaky ReLU:** (Maas *et al.*, 2013) Fix a_i to a small value e.g. 0.01
  - **Parametric ReLU:** (He *et al.*, 2015) Learn a_i
  - **Randomized ReLU:** (Xu *et al.*, 2015) Sample a_i from a fixed range during training, fix during testing
  - ....

Generalized Rectified Linear Units

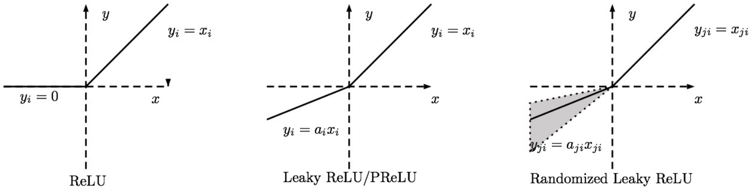


Figure: Xu *et al.* "Empirical Evaluation of Rectified Activations in Convolutional Network"

Exponential Linear Units (ELUs)

$$g(z) = \begin{cases} z \text{ if } z > 0 \\ \alpha(\exp z - 1) \text{ if } z \leq 0 \end{cases}$$

Exponential Linear Units (ELUs)

$$g(z) = \begin{cases} z \text{ if } z > 0 \\ \alpha(\exp z - 1) \text{ if } z \leq 0 \end{cases}$$

- All the benefits of ReLU + does not get killed

Exponential Linear Units (ELUs)

$$g(z) = \begin{cases} z \text{ if } z > 0 \\ \alpha(\exp z - 1) \text{ if } z \leq 0 \end{cases}$$

- All the benefits of ReLU + does not get killed
- Problem: Need to exponentiate

Exponential Linear Units (ELUs)

$$g(z) = \begin{cases} z \text{ if } z > 0 \\ \alpha(\exp z - 1) \text{ if } z \leq 0 \end{cases}$$

- All the benefits of ReLU + does not get killed
- Problem: Need to exponentiate

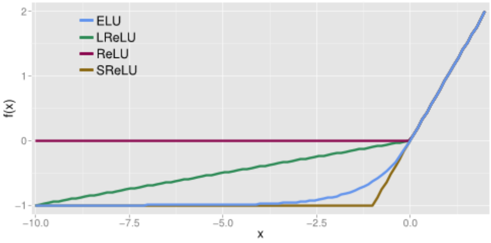


Figure: Clevert *et al.* "Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)", 2016

Maxout Units

- Generalizes ReLUs further but does not fit into the (dot product $\rightarrow$ nonlinearity) mold

Maxout Units

- Generalizes ReLUs further but does not fit into the (dot product $\rightarrow$ nonlinearity) mold
- Instead of applying an element-wise function $g(z)$, divide vector $\mathbf{z}$ into k groups (more parameters!)

Maxout Units

- Generalizes ReLUs further but does not fit into the (dot product $\rightarrow$ nonlinearity) mold
- Instead of applying an element-wise function $g(z)$, divide vector $\mathbf{z}$ into k groups (more parameters!)
- Output maximum element of one of k groups
  $g(\mathbf{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$

Maxout Units

- Generalizes ReLUs further but does not fit into the (dot product $\rightarrow$ nonlinearity) mold
- Instead of applying an element-wise function $g(z)$, divide vector $\mathbf{z}$ into k groups (more parameters!)
- Output maximum element of one of k groups
  $g(\mathbf{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$
- $g(\mathbf{z})_i = \max\{w_1^T \mathbf{x} + b_1, \ldots, w_k^T \mathbf{x} + b_k\}$
- A maxout unit makes a piecewise linear approximation (with k pieces) to an arbitrary convex function

Maxout Units

- Generalizes ReLUs further but does not fit into the (dot product $\rightarrow$ nonlinearity) mold
- Instead of applying an element-wise function $g(z)$, divide vector $\mathbf{z}$ into k groups (more parameters!)
- Output maximum element of one of k groups
  $g(\mathbf{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$
- $g(\mathbf{z})_i = \max\{w_1^T \mathbf{x} + b_1, \ldots, w_k^T \mathbf{x} + b_k\}$
- A maxout unit makes a piecewise linear approximation (with k pieces) to an arbitrary convex function
- Can be thought of as learning the activation function itself

Maxout Units

- Generalizes ReLUs further but does not fit into the (dot product $\rightarrow$ nonlinearity) mold
- Instead of applying an element-wise function $g(z)$, divide vector $\mathbf{z}$ into k groups (more parameters!)
- Output maximum element of one of k groups
  $g(\mathbf{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$
- $g(\mathbf{z})_i = \max\{w_1^T \mathbf{x} + b_1, \ldots, w_k^T \mathbf{x} + b_k\}$
- A maxout unit makes a piecewise linear approximation (with k pieces) to an arbitrary convex function
- Can be thought of as learning the activation function itself
- With $k = 2$ we CAN recover absolute value rectification, or ReLU or PReLU

Maxout Units

- Generalizes ReLUs further but does not fit into the (dot product $\rightarrow$ nonlinearity) mold
- Instead of applying an element-wise function $g(z)$, divide vector $\mathbf{z}$ into k groups (more parameters!)
- Output maximum element of one of k groups
  $g(\mathbf{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$
- $g(\mathbf{z})_i = \max\{w_1^T \mathbf{x} + b_1, \ldots, w_k^T \mathbf{x} + b_k\}$
- A maxout unit makes a piecewise linear approximation (with k pieces) to an arbitrary convex function
- Can be thought of as learning the activation function itself
- With $k = 2$ we CAN recover absolute value rectification, or ReLU or PReLU
- Each unit parameterized by k weight vectors instead of 1, needs stronger regularization

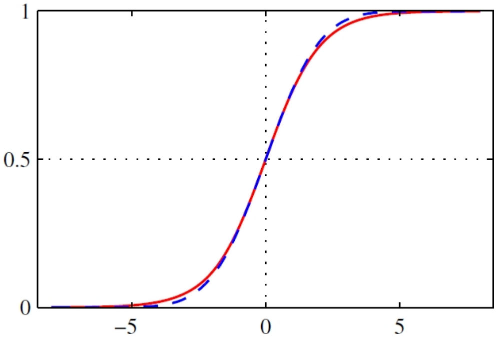
Sigmoid Units

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

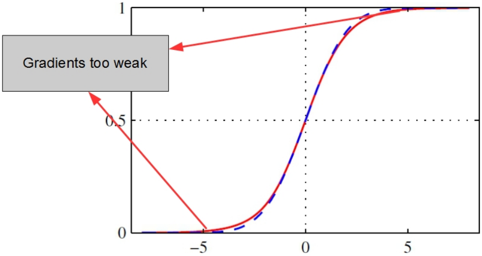
Sigmoid Units

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

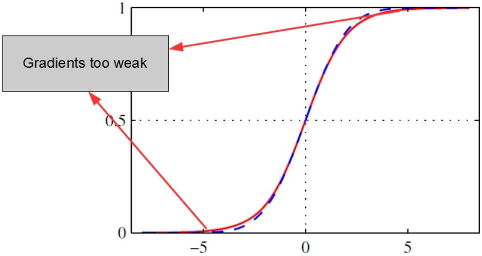
- Squashing type non-linearity: pushes outputs to range $[0, 1]$



Sigmoid Units

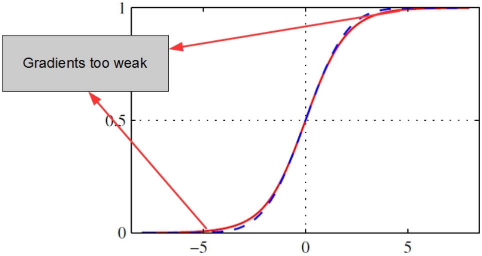


Sigmoid Units



- Problem: Saturate across most of their domain, strongly sensitive only when z is closer to zero

Sigmoid Units



- Problem: Saturate across most of their domain, strongly sensitive only when z is closer to zero
- Saturation makes gradient based learning difficult

Tanh Units

- Related to sigmoid: $g(z) = \tanh(z) = 2\sigma(2z) - 1$

Tanh Units

- Related to sigmoid: $g(z) = \tanh(z) = 2\sigma(2z) - 1$
- Positives: Squashes output to range $[-1, 1]$, outputs are zero-centered

Tanh Units

- Related to sigmoid: $g(z) = \tanh(z) = 2\sigma(2z) - 1$
- Positives: Squashes output to range $[-1, 1]$, outputs are zero-centered
- Negative: Also saturates

Tanh Units

- Related to sigmoid: $g(z) = \tanh(z) = 2\sigma(2z) - 1$
- Positives: Squashes output to range $[-1, 1]$, outputs are zero-centered
- Negative: Also saturates
- Still better than sigmoid as $\hat{y} = \mathbf{w}^T \tanh(U^T \tanh(V^T \mathbf{x}))$ resembles $\hat{y} = \mathbf{w}^T U^T V^T \mathbf{x}$ when activations are small

Other Units

- **Radial Basis Functions:** $g(z)_i = \exp\left(\frac{1}{\sigma_i^2}\|W_{:,i}\mathbf{x}\|^2\right)$

Other Units

- **Radial Basis Functions:** $g(z)_i = \exp\left(\frac{1}{\sigma_i^2}\|W_{:,i}\mathbf{x}\|^2\right)$
- Function is more active as $\mathbf{x}$ approaches a template $W_{:,i}$. Also saturates and is hard to train

Other Units

- **Radial Basis Functions:** $g(z)_i = \exp\left(\frac{1}{\sigma_i^2}\|W_{:,i}\mathbf{x}\|^2\right)$
- Function is more active as $\mathbf{x}$ approaches a template $W_{:,i}$. Also saturates and is hard to train
- **Softplus:** $g(z) = \log(1 + e^z)$. Smooth version of rectifier (Dugas *et al.*, 2001), although differentiable everywhere, empirically performs worse than rectifiers

Other Units

- **Radial Basis Functions:** $g(z)_i = \exp\left(\frac{1}{\sigma_i^2}\|W_{:,i}\mathbf{x}\|^2\right)$
- Function is more active as $\mathbf{x}$ approaches a template $W_{:,i}$. Also saturates and is hard to train
- **Softplus:** $g(z) = \log(1 + e^z)$. Smooth version of rectifier (Dugas *et al.*, 2001), although differentiable everywhere, empirically performs worse than rectifiers
- **Hard Tanh:** $g(z) = \max(-1, \min(1, z))$, like the rectifier, but bounded (Collobert, 2004)

Summary

- In Feedforward Networks don't use Sigmoid

Summary

- In Feedforward Networks don't use Sigmoid
- When a sigmoidal function *must* be used, use $\tanh$

Summary

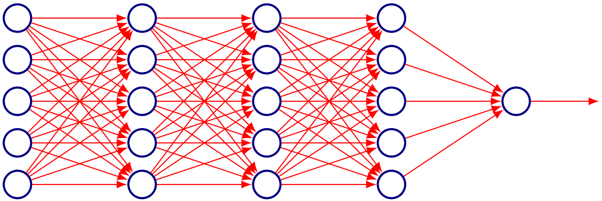
- In Feedforward Networks don't use Sigmoid
- When a sigmoidal function *must* be used, use $\tanh$
- Use ReLU by default, but be careful with learning rates

Summary

- In Feedforward Networks don't use Sigmoid
- When a sigmoidal function *must* be used, use $\tanh$
- Use ReLU by default, but be careful with learning rates
- Try other generalized ReLUs and Maxout for possible improvement

Universality and Depth

Architecture Design



- First layer: $\mathbf{h}^{(1)} = g^{(1)}\left(W^{(1)^T}\mathbf{x} + \mathbf{b}^{(1)}\right)$
- Second layer: $\mathbf{h}^{(2)} = g^{(2)}\left(W^{(2)^T}\mathbf{h}^{(1)} + \mathbf{b}^{(2)}\right)$
- How do we decide *depth*, *width*?
- In theory how many layers *suffice*?

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)
- Implication: Regardless of function we are trying to learn, we know a large MLP can *represent* this function

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)
- Implication: Regardless of function we are trying to learn, we know a large MLP can *represent* this function
- But not guaranteed that our training algorithm will be able to *learn* that function

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)
- Implication: Regardless of function we are trying to learn, we know a large MLP can *represent* this function
- But not guaranteed that our training algorithm will be able to *learn* that function
- Gives no guidance on how large the network will be (exponential size in worst case)

Universality

- Theoretical result [Cybenko, 1989]: 2-layer net with linear output with some squashing non-linearity in hidden units can approximate any continuous function over compact domain to arbitrary accuracy (given enough hidden units!)
- Implication: Regardless of function we are trying to learn, we know a large MLP can *represent* this function
- But not guaranteed that our training algorithm will be able to *learn* that function
- Gives no guidance on how large the network will be (exponential size in worst case)
- Talked of some suggestive results earlier:

One more result:

- (Montufar *et al.*, 2014) Number of linear regions carved out by a deep rectifier network with d inputs, depth l and n units per hidden layer is:

$$O\left(\binom{n}{d}^{d(l-1)} n^d \right)$$

One more result:

- (Montufar *et al.*, 2014) Number of linear regions carved out by a deep rectifier network with d inputs, depth l and n units per hidden layer is:

$$O\left(\binom{n}{d}^{d(l-1)} n^d\right)$$

- Exponential in depth!
- They showed functions representable with a deep rectifier network can require an exponential number of hidden units with a shallow network

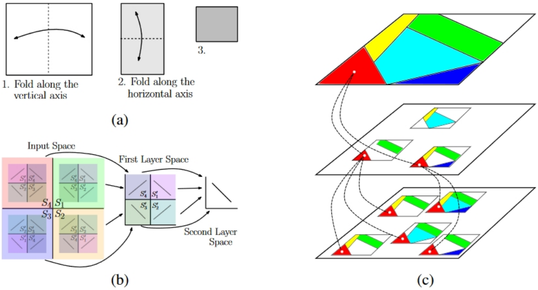


Figure 2: (a) Space folding of 2-D Euclidean space along the two axes. (b) An illustration of how the top-level partitioning (on the right) is replicated to the original input space (left). (c) Identification of regions across the layers of a deep model.

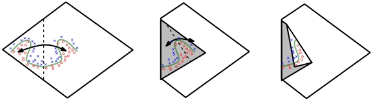


Figure 3: Space folding of 2-D space in a non-trivial way. Note how the folding can potentially identify symmetries in the boundary that it needs to learn.

Figure: Montufar *et al.*, 2014

Advantages of Depth

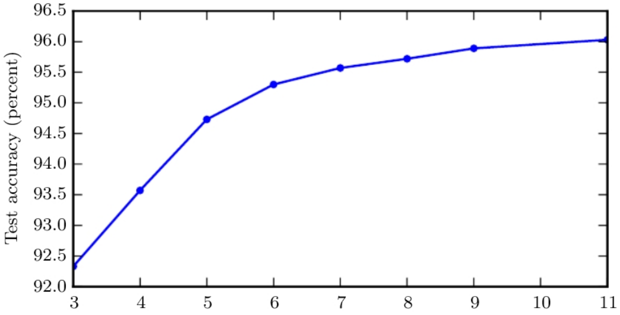
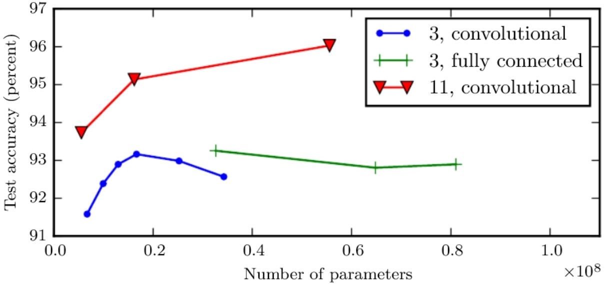


Figure: Goodfellow *et al.*, 2014

Advantages of Depth

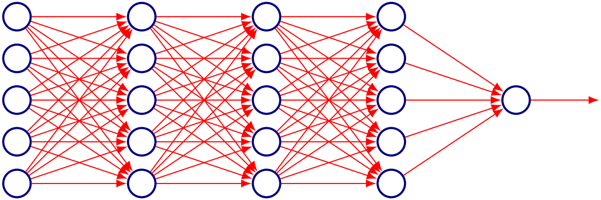


- Control experiments show that other increases to model size don't yield the same effect

Figure: Goodfellow *et al.*, 2014

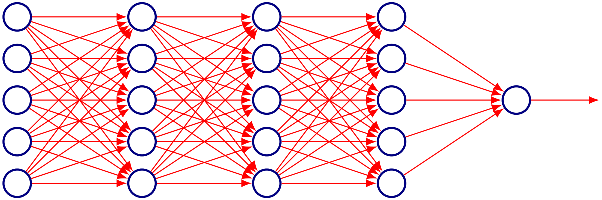
Backpropagation: Introduction

How do we learn weights?



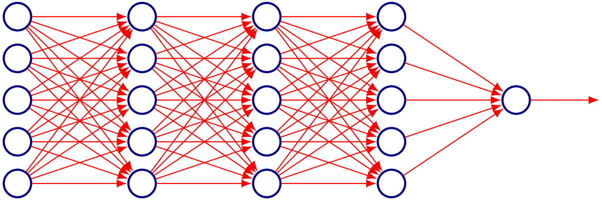
- First Idea: Randomly perturb one weight, see if it improves performance, save the change

How do we learn weights?



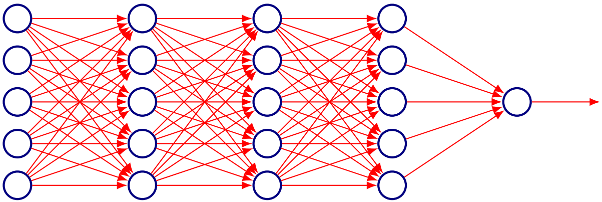
- First Idea: Randomly perturb one weight, see if it improves performance, save the change
- **Very inefficient:** Need to do many passes over a sample set for just one weight change

How do we learn weights?



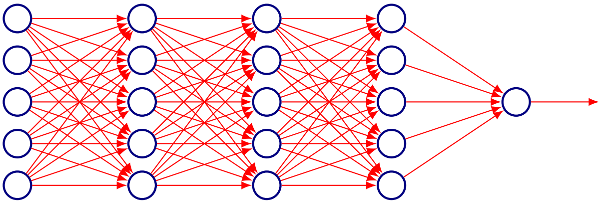
- First Idea: Randomly perturb one weight, see if it improves performance, save the change
- **Very inefficient:** Need to do many passes over a sample set for just one weight change
- What does this remind you of?

How do we learn weights?



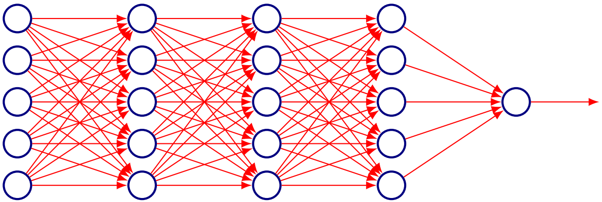
- Another Idea: Perturb all the weights in parallel, and correlate the performance gain with weight changes

How do we learn weights?



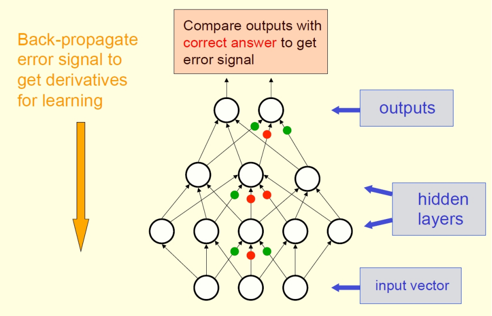
- Another Idea: Perturb all the weights in parallel, and correlate the performance gain with weight changes
- Very hard to implement

How do we learn weights?

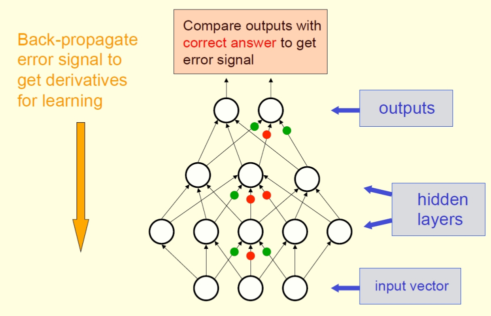


- Another Idea: Perturb all the weights in parallel, and correlate the performance gain with weight changes
- Very hard to implement
- Yet another idea: Only perturb activations (since they are fewer). Still very inefficient.

Backpropagation

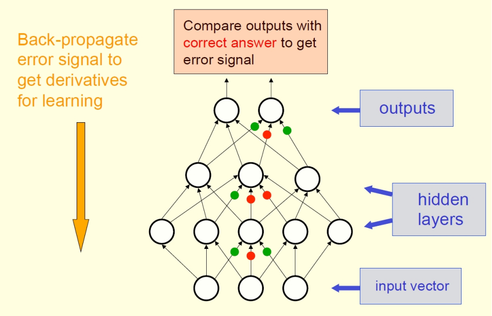


Backpropagation



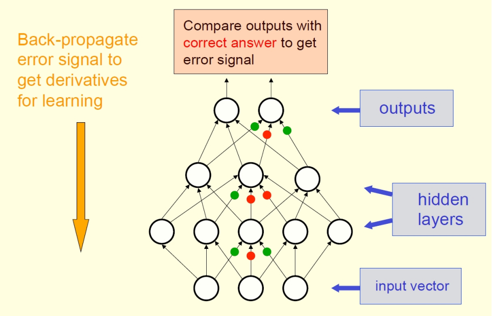
- **Feedforward Propagation**: Accept input x, pass through intermediate stages and obtain output $\hat{y}$

Backpropagation



- **Feedforward Propagation**: Accept input x, pass through intermediate stages and obtain output $\hat{y}$
- **During Training**: Use $\hat{y}$ to compute a scalar cost $J(\theta)$

Backpropagation



- **Feedforward Propagation**: Accept input x, pass through intermediate stages and obtain output $\hat{y}$
- **During Training**: Use $\hat{y}$ to compute a scalar cost $J(\theta)$
- Backpropagation allows information to flow backwards from cost to compute the gradient

Figure: G. E. Hinton

Backpropagation

- From the training data we don't know what the hidden units should do

Backpropagation

- From the training data we don't know what the hidden units should do
- But, we can compute how fast the error changes as we change a hidden activity

Backpropagation

- From the training data we don't know what the hidden units should do
- But, we can compute how fast the error changes as we change a hidden activity
- Use error derivatives w.r.t hidden activities

Backpropagation

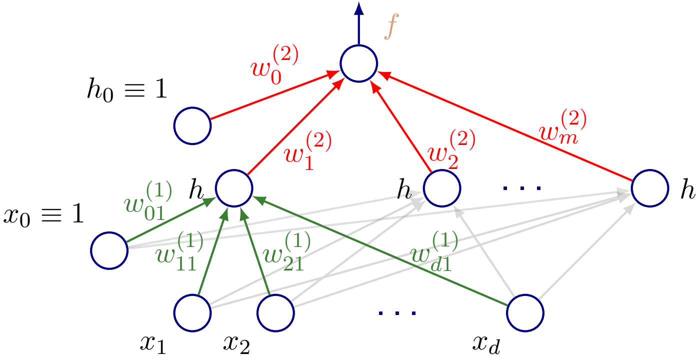
- From the training data we don't know what the hidden units should do
- But, we can compute how fast the error changes as we change a hidden activity
- Use error derivatives w.r.t hidden activities
- Each hidden unit can affect many output units and have separate effects on error – combine these effects

Backpropagation

- From the training data we don't know what the hidden units should do
- But, we can compute how fast the error changes as we change a hidden activity
- Use error derivatives w.r.t hidden activities
- Each hidden unit can affect many output units and have separate effects on error – combine these effects
- Can compute error derivatives for hidden units efficiently (and once we have error derivatives for hidden activities, easy to get error derivatives for weights going in)

Slide: G. E. Hinton

Review: neural networks



- Feedforward operation, from input $\mathbf{x}$ to output $\hat{y}$:

$$\hat{y}(\mathbf{x};\mathbf{w}) = f\left(\sum_{j=1}^{m} w_j^{(2)} h\left(\sum_{i=1}^{d} w_{ij}^{(1)} x_i + w_{0j}^{(1)}\right) + w_0^{(2)}\right)$$

Slide adapted from TTIC 31020, Gregory Shakhnarovich

Training the network

- Error of the network on a training set:

$$L(X;\mathbf{w}) \; = \; \sum_{i=1}^{N} \frac{1}{2} \left(y_i - \hat{y}(\mathbf{x}_i;\mathbf{w})\right)^2$$

Training the network

- Error of the network on a training set:

$$L(X; \mathbf{w}) = \sum_{i=1}^{N} \frac{1}{2} \left(y_i - \hat{y}(\mathbf{x}_i; \mathbf{w})\right)^2$$

- Generally, no closed-form solution;
  resort to gradient descent

Training the network

- Error of the network on a training set:

$$L(X;\mathbf{w}) \;=\; \sum_{i=1}^{N} \frac{1}{2}\left(y_i - \hat{y}(\mathbf{x}_i;\mathbf{w})\right)^2$$

- Generally, no closed-form solution;
  resort to gradient descent
- Need to evaluate derivative of L on a single example

Training the network

- Error of the network on a training set:

$$L(X;\mathbf{w}) \;=\; \sum_{i=1}^{N} \frac{1}{2}\left(y_i - \hat{y}(\mathbf{x}_i;\mathbf{w})\right)^2$$

- Generally, no closed-form solution;
  resort to gradient descent
- Need to evaluate derivative of L on a single example
- Let's start with a simple linear model $\hat{y} = \sum_j w_j x_{ij}$:

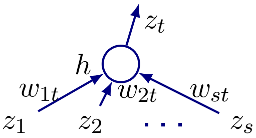
$$\frac{\partial L(\mathbf{x}_i)}{\partial w_j} \;=\; \underbrace{(\hat{y}_i - y_i)}_{\text{error}} x_{ij}.$$

Slide adapted from TTIC 31020, Gregory Shakhnarovich

Backpropagation

- General unit activation in a multilayer network:

$$z_t = h\left(\sum_j w_{jt} z_j\right)$$

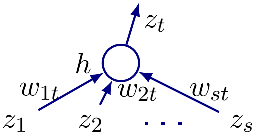


- Forward propagation: calculate for each unit $a_t = \sum_j w_{jt} z_j$

Backpropagation

- General unit activation in a multilayer network:

$$z_t = h\left(\sum_j w_{jt} z_j\right)$$

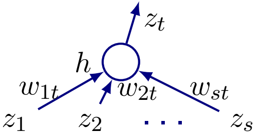


- Forward propagation: calculate for each unit $a_t = \sum_j w_{jt} z_j$
- The loss L depends on w_{jt} only through a_t:

Backpropagation

- General unit activation in a multilayer network:

$$z_t = h\left(\sum_j w_{jt} z_j\right)$$



- Forward propagation: calculate for each unit $a_t = \sum_j w_{jt} z_j$
- The loss L depends on w_{jt} only through a_t:

$$\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t}\frac{\partial a_t}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j$$

Slide adapted from TTIC 31020, Gregory Shakhnarovich

Backpropagation

$$\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j \qquad \frac{\partial L}{\partial w_{jt}} = \underbrace{\frac{\partial L}{\partial a_t}}_{\delta_t} z_j$$

Backpropagation

$$\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j \qquad \frac{\partial L}{\partial w_{jt}} = \underbrace{\frac{\partial L}{\partial a_t}}_{\delta_t} z_j$$

- Output unit with linear activation: $\delta_t = \hat{y} - y$

Backpropagation

$$\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j \qquad \frac{\partial L}{\partial w_{jt}} = \underbrace{\frac{\partial L}{\partial a_t}}_{\delta_t} z_j$$

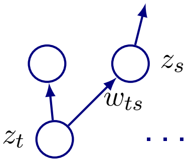
- Output unit with linear activation: $\delta_t = \hat{y} - y$
- Hidden unit $z_t = h(a_t)$ which sends inputs to units S:

Backpropagation

$$\frac{\partial L}{\partial w_{jt}} = \frac{\partial L}{\partial a_t} z_j \qquad \frac{\partial L}{\partial w_{jt}} = \underbrace{\frac{\partial L}{\partial a_t}}_{\delta_t} z_j$$

- Output unit with linear activation: $\delta_t = \hat{y} - y$
- Hidden unit $z_t = h(a_t)$ which sends inputs to units S:

$$\delta_t = \sum_{s \in S} \frac{\partial L}{\partial a_s} \frac{\partial a_s}{\partial a_t}$$
$$= h'(a_t) \sum_{s \in S} w_{ts} \delta_s$$



$$a_s = \sum_{j : j \to s} w_{js} h(a_j)$$

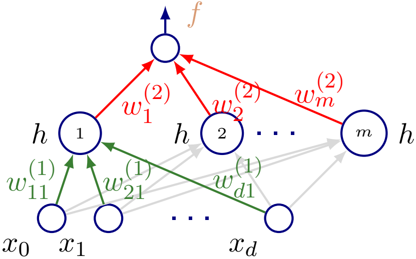
Slide adapted from TTIC 31020, Gregory Shakhnarovich

Backpropagation: example

- Output: $f(a) = a$
- Hidden:

$$h(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}},$$

$$h'(a) = 1 - h(a)^2.$$



- Given example $\mathbf{x}$, feed-forward inputs:

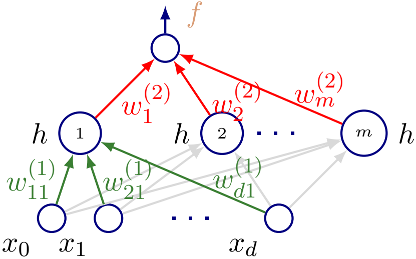
$$\text{input to hidden: } a_j = \sum_{i=0}^{d} w_{ij}^{(1)} x_i,$$

Backpropagation: example

- Output: $f(a) = a$
- Hidden:

$$h(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}},$$

$$h'(a) = 1 - h(a)^2.$$



- Given example $\mathbf{x}$, feed-forward inputs:

$$\text{input to hidden: } a_j = \sum_{i=0}^{d} w_{ij}^{(1)} x_i,$$

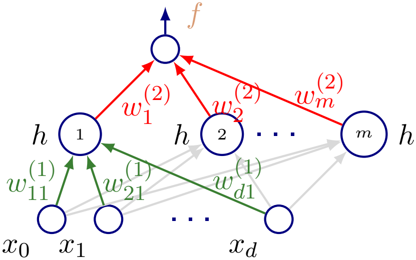
$$\text{hidden output: } z_j = \tanh(a_j),$$

Backpropagation: example

- Output: $f(a) = a$
- Hidden:

$$h(a) = \tanh(a) = \frac{e^a - e^{-a}}{e^a + e^{-a}},$$

$$h'(a) = 1 - h(a)^2.$$



- Given example $\mathbf{x}$, feed-forward inputs:

$$\text{input to hidden: } a_j = \sum_{i=0}^{d} w_{ij}^{(1)} x_i,$$

$$\text{hidden output: } z_j = \tanh(a_j),$$

$$\text{net output: } \hat{y} = a = \sum_{j=0}^{m} w_j^{(2)} z_j.$$

Backpropagation: example

$$a_j = \sum_{i=0}^{d} w_{ij}^{(1)} x_i, \quad z_j = \tanh(a_j), \quad \hat{y} = a = \sum_{j=0}^{m} w_j^{(2)} z_j.$$

- Error on example $\mathbf{x}$: $L = \frac{1}{2}(y - \hat{y})^2$.

Backpropagation: example

$$a_j = \sum_{i=0}^{d} w_{ij}^{(1)} x_i, \quad z_j = \tanh(a_j), \quad \hat{y} = a = \sum_{j=0}^{m} w_j^{(2)} z_j.$$

- Error on example $\mathbf{x}$: $L = \frac{1}{2}(y - \hat{y})^2$.
- Output unit: $\delta = \frac{\partial L}{\partial a} = y - \hat{y}$.

Backpropagation: example

$$a_j = \sum_{i=0}^{d} w_{ij}^{(1)} x_i, \quad z_j = \tanh(a_j), \quad \hat{y} = a = \sum_{j=0}^{m} w_j^{(2)} z_j.$$

- Error on example $\mathbf{x}$: $L = \frac{1}{2}(y - \hat{y})^2$.
- Output unit: $\delta = \frac{\partial L}{\partial a} = y - \hat{y}$.
- Next, compute δs for the hidden units:

$$\delta_j = (1 - z_j)^2 w_j^{(2)} \delta$$

Backpropagation: example

$$a_j = \sum_{i=0}^{d} w_{ij}^{(1)} x_i, \quad z_j = \tanh(a_j), \quad \hat{y} = a = \sum_{j=0}^{m} w_j^{(2)} z_j.$$

- Error on example $\mathbf{x}$: $L = \frac{1}{2}(y - \hat{y})^2$.
- Output unit: $\delta = \frac{\partial L}{\partial a} = y - \hat{y}$.
- Next, compute δs for the hidden units:

$$\delta_j = (1 - z_j)^2 w_j^{(2)} \delta$$

- Derivatives w.r.t. weights:

$$\frac{\partial L}{\partial w_{ij}^{(1)}} = \delta_j x_i, \qquad \frac{\partial L}{\partial w_j^{(2)}} = \delta z_j.$$

Backpropagation: example

$$a_j = \sum_{i=0}^{d} w_{ij}^{(1)} x_i, \quad z_j = \tanh(a_j), \quad \hat{y} = a = \sum_{j=0}^{m} w_j^{(2)} z_j.$$

- Error on example $\mathbf{x}$: $L = \frac{1}{2}(y - \hat{y})^2$.
- Output unit: $\delta = \frac{\partial L}{\partial a} = y - \hat{y}$.
- Next, compute δs for the hidden units:

$$\delta_j = (1 - z_j)^2 w_j^{(2)} \delta$$

- Derivatives w.r.t. weights:

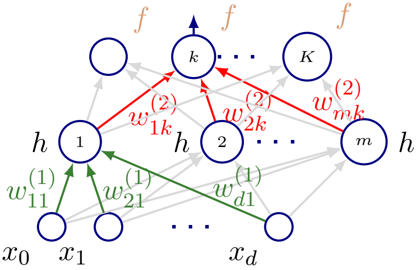
$$\frac{\partial L}{\partial w_{ij}^{(1)}} = \delta_j x_i, \qquad \frac{\partial L}{\partial w_j^{(2)}} = \delta z_j.$$

- Update weights: $w_j \leftarrow w_j - \eta \delta z_j$ and $w_{ij}^{(1)} \leftarrow w_{ij}^{(1)} - \eta \delta_j x_i$. η is called the weight decay

Multidimensional output

- Loss on example $(\mathbf{x}, \mathbf{y})$:

$$\frac{1}{2}\sum_{k=1}^{K}(y_k - \hat{y}_k)^2$$



- Now, for each output unit $\delta_k = y_k - \hat{y}_k$;
- For hidden unit j,

$$\delta_j = (1 - z_j)^2 \sum_{k=1}^{K} w_{jk}^{(2)} \delta_k.$$

Next time

- More Backpropagation

Next time

- More Backpropagation
- Start with Regularization in Neural Networks

Next time

- More Backpropagation
- Start with Regularization in Neural Networks
- Quiz