# Feedforward Neural Networks

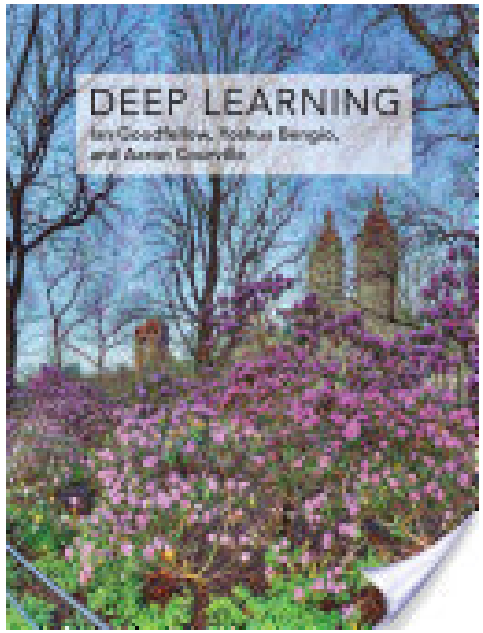
Yagmur Gizem Cinar, Eric Gaussier

AMA, LIG, Univ. Grenoble Alpes

17 March 2017

# Reference Book

Deep Learning
Ian Goodfellow and Yoshua Bengio and Aaron Courville
MIT Press
2016



# Table of Contents

# Multilayer Perceptrons

Feedforward Neural Networks, Deep feedforward Networks

## Goal

to approximate function $f^*$

$$y = f^*(\mathbf{x}) \tag{1}$$

- Classification $y \in \{c_1, c_2, \ldots c_K\}$
- Regression $y \in \mathbb{R}$

## A feedforward network

$$y = f(\mathbf{x}; \theta) \tag{2}$$

**Feedforward**: $\mathbf{x}$ through $f$ and finally $y$
No feedback connections as **recurrent neural network**

# Multilayer Perceptrons

Feedforward Neural Networks

- network: composing different functions
- a directed acyclic graph
- e.g. $f^{(1)}$, $f^{(2)}$, and $f^{(3)}$

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

  $f^{(1)}$ is $1^{\text{st}}$ layer
  $f^{(2)}$ is $2^{\text{nd}}$ layer
- final layer is called **output layer**
- other layers are called **hidden layers**
- length of the chain is the **depth** of the network
- **width** is the dimensionality of the hidden layers

# Feedforward Neural Networks

loosely inspired by Neuroscience

- Many **units** acts at the same time
- Each unit receives from many other units and computes its own activation
- MLP as a function approximation machines designed to generalize well
- Linear models
  - \+ fit efficiently and reliable with convex optimization
  - \- limited to linear function

One way to obtain nonlinearity is a mapping $\phi$ can be learned with deep learning

$$y = f(\mathbf{x}; \theta, \mathbf{w}) = \phi(\mathbf{x}; \theta)^{\mathsf{T}} \mathbf{w} \tag{3}$$

- $\theta$ parameters of $\phi$
- $\mathbf{w} \in \mathbb{R}^n$ parameters of desired map from $\phi(\mathbf{x})$ to $y$

# Example: Learning XOR

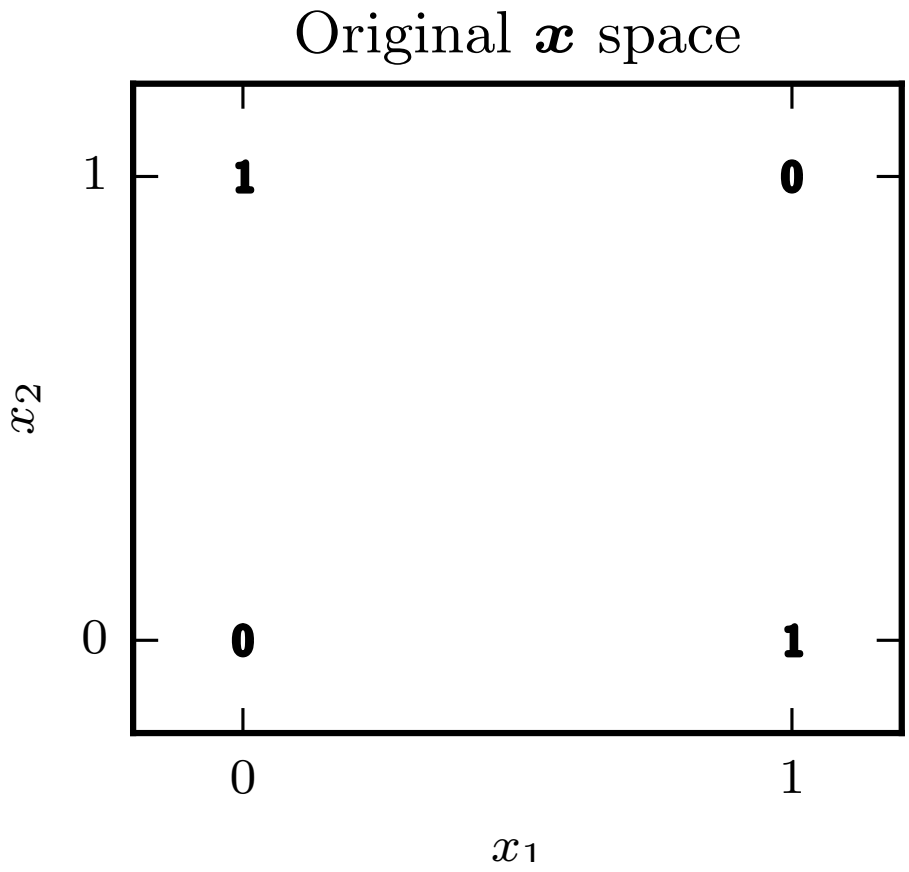



Figure 1: XOR in **x** space[1].

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

# Example: Learning XOR

- $\mathbb{X} = \{[0,0]^\mathsf{T}, [0,1]^\mathsf{T}, [1,0]^\mathsf{T}, [1,1]^\mathsf{T}\}$
- XOR is not linearly separable
- XOR target function $y = f^*(\mathbf{x})$
- model function $y = f(\mathbf{x}; \theta)$
- XOR MSE loss function

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

- If model is a linear single-layer with one unit

$$f(\mathbf{x}; \theta) = \mathbf{x}^\mathsf{T} \mathbf{w} + b$$

# Example: Learning XOR

A single-layer with one hidden unit also called perceptron:

$$f(\mathbf{x};\theta) = \mathbf{x}^{\mathsf{T}}\mathbf{w} + b$$

- cannot separate XOR

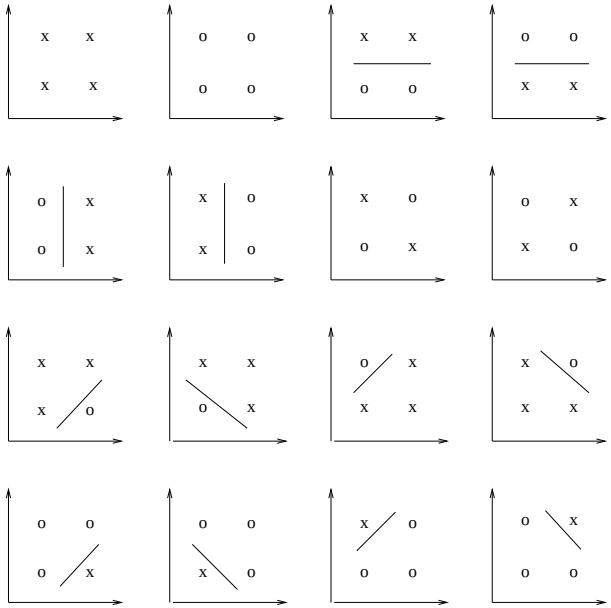


Figure 2: XOR is not linearly separable[2].

[2] Johan Suykens. *Lecture notes in Artificial Neural Networks*. 2015.

# Example: Learning XOR

A single layer with two hidden units

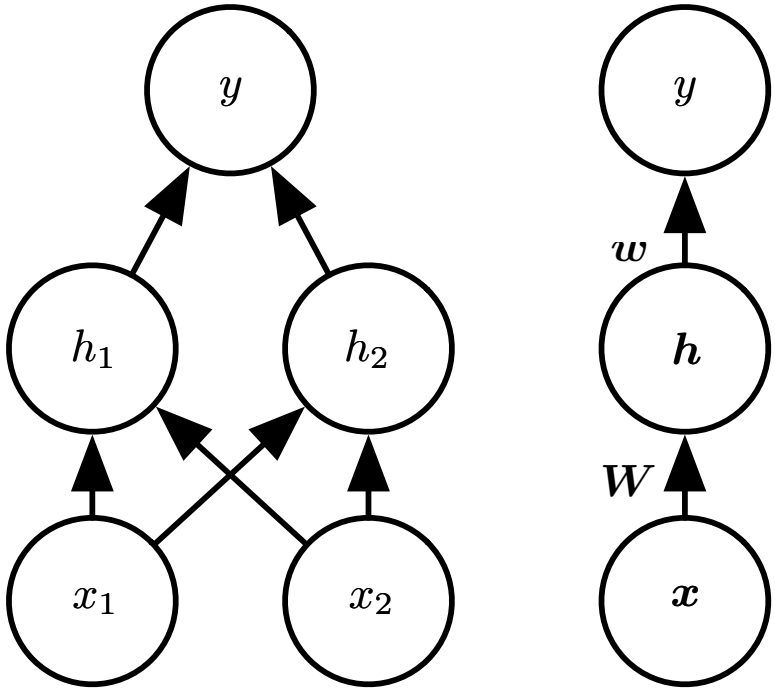



Figure 3: Network diagrams[3].

[3] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

# Example: Learning XOR

One hidden layer with two hidden units

$$\mathbf{h} = f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$$
$$y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$$
$$f(\mathbf{x}, \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x}))$$

$\mathbf{W}$ and $\mathbf{w}$ weights of a linear transformation
$b$ and $\mathbf{c}$ biases

$$f^{(1)}(\mathbf{x}) = \mathbf{W}^{\mathsf{T}}\mathbf{x}$$
$$f^{(2)}(\mathbf{h}) = \mathbf{h}^{\mathsf{T}}\mathbf{w}$$
$$f(\mathbf{x}) = \mathbf{w}^{\mathsf{T}}\mathbf{W}^{\mathsf{T}}\mathbf{x}$$

(intercept/bias terms ignored)

# Example: Learning XOR

For a nonlinearity: **activation function** $g$

$$\mathbf{h} = g(\mathbf{W}^{\mathsf{T}}\mathbf{x} + c)$$

A rectified linear unit (ReLU) is the activation function for many feedforward networks

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

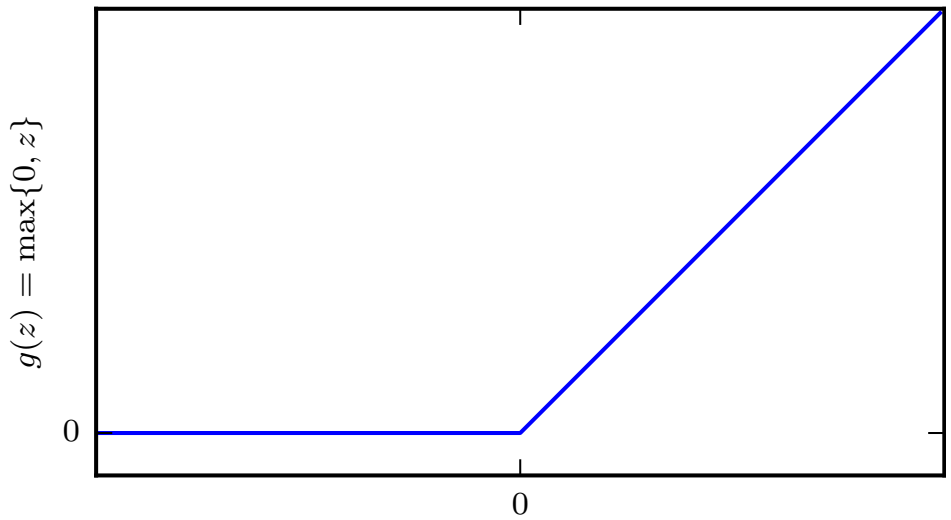



Figure 4: ReLU activation function[4].

# Example: Learning XOR

Complete network

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

Let

$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{c} = \begin{bmatrix} 0 \\ -1 \end{bmatrix}$$

$$\mathbf{w} = \begin{bmatrix} 1 \\ -2 \end{bmatrix}$$

$$b = 0$$

# Example: Learning XOR

Design matrix $\mathbf{X}$

$$\mathbf{X} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

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

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

# Example: Learning XOR

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

$$\mathbf{w}^{\mathsf{T}} \max\{0, \mathbf{XW} + \mathbf{c}\} + b = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

# Gradient-Based Learning

- In real life billions of model parameters
- Gradient-based optimization algorithm provide solution with little error
- Nonlinearity leads to a nonconvex loss function
- Trained by iterative gradient-based optimizers
- Global convergence is not guaranteed
- Sensitive to initialization of parameters
  - initialize weights with small random values
  - bias can be 0 or small positive values e.g. 0.1

# Gradient-Based Learning

Cost function:

$$J(\mathbf{w}, b) = -\mathbb{E}_{\mathbf{x},y \ \hat{p}_{\text{data}}} \log p_{\text{model}}(y|\mathbf{x})$$

Mostly negative log-likelihood as a cost function

So, minimizing the cost leads to maximum likelihood estimation

Cross entropy between the training data and model's prediction as a cost function

Typically total cost composed of cross entropy and regularization

# Gradient-Based Learning

Cost functions

- mean squared error (MSE)

$$f^* = \underset{f}{\operatorname{argmin}} \, \mathbb{E}_{x,y \ p_{\text{data}}} ||y - f(\mathbf{x})||^2$$

- Mean absolute error (MAE)

$$f^* = \underset{f}{\operatorname{argmin}} \, \mathbb{E}_{x,y \ p_{\text{data}}} ||y - f(\mathbf{x})||_1$$

# Gradient-Based Learning

Output units

The choice of output function determines the cross entropy

| Output Type | Output Distribution | Output Layer | Cost Function |
|---|---|---|---|
| Binary | Bernoulli | Sigmoid | Binary cross-entropy |
| Discrete | Multinoulli | Softmax | Discrete cross-entropy |
| Continuous | Gaussian | Linear | Gaussian cross-entropy (MSE) |
| Continuous | Mixture of Gaussian | Mixture Density | Cross-entropy |
| Continuous | Arbitrary | See part III: GAN, VAE, FVBN | Various |

Figure 5: Output units[5].

[5] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*

# Hidden Units

Hidden units are composed of

- input vectors $\mathbf{x}$
- computing an affine transformation $z = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$
- element-wise nonlinear function $g(z)$

Some activation functions $g(z)$ are not differentiable at all points
e.g. ReLU not differentiable at $z = 0$
But still perform well in practice

# Hidden Units Activation Function

Why perform well in practice?

- Training algorithms do not usually reach to the global minimum (nonconvex) but reduce it significantly

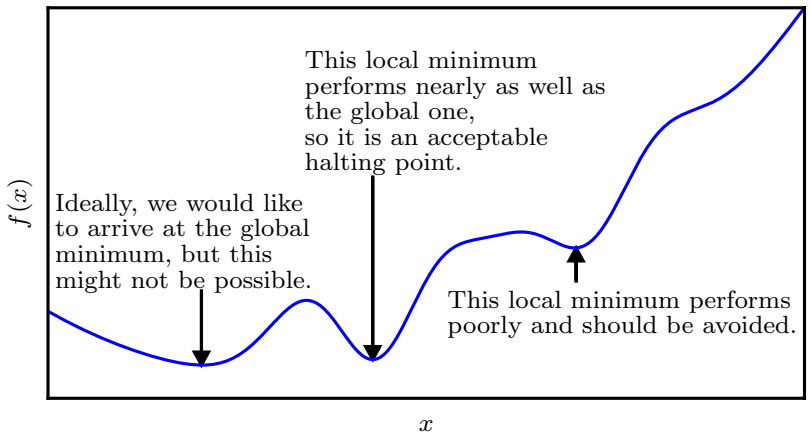



Figure 6: Approximate optimization[6].

- Nondifferentiable at a small number of points
- Implementation return 1 for nondifferentiable inputs

[6] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

# Rectified Linear units

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

+ Easy to optimize, close to linear units
+ Derivatives through ReLU remain large when active
+ Derivative is 1 when active
+ Second order derivative is 0 almost everywhere
+ Derivative more useful without second-order effects
− ReLU cannot learn via gradient when activation is 0

Typically applied to affine transformation

$$\mathbf{h} = g(\mathbf{W}^\top \mathbf{x} + c)$$

- small positive $b$ e.g. $b = 0.1$ enables to allow derivatives to pass
  $->$ generalized ReLUs to have gradient everywhere

# Generalizations of Rectified Linear Units

- 3 generalizations based on nonzero slope $\alpha_i$ when $z_i < 0$

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

- Absolute value rectification $\alpha_i = -1$ and $g(z) = |z|$
- Leaky ReLU $\alpha_i$ a small value like 0.01
- Parametric ReLU (PReLU) $\alpha_i$ is a learnable parameter

# Maxout units

- Maxout units divide $z$ into groups of $k$ values
- For each output is the maximum element of these groups

$$g(\mathbf{z})_i = \max_{j \in \mathbb{G}^{(i)}} z_j$$

- where $\mathbb{G}^{(i)}$ is set of indices for group $i$, $\{(i-1)k+1, \ldots, ik\}$
- Maxout can learn piecewise linear convex function up to $k$ pieces
- Maxout generalize rectified units further
- Requires more regularization compared to rectified units
- Maxout with number of elements in group and large number of examples can work without regularization
- Next layer can get $k$ times smaller

# Sigmoidal Activation Functions

- Logistic sigmoid $\sigma$

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

- Hyperbolic tangent tanh

$$\tanh(z) = \frac{1 - \exp(-2z)}{1 + \exp(-2z)}$$

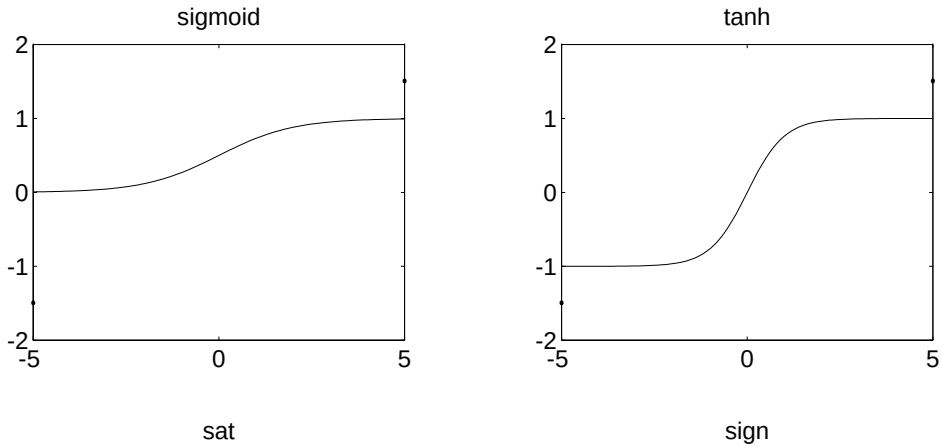



Figure 7: Sigmoid $\sigma$ and *tanh* activation functions[7].

[7] Johan Suykens. *Lecture notes in Artificial Neural Networks*. 2015.

# Gradient-Based Learning

Sigmoidal units

- $\tanh(z) = 2\sigma(2z) - 1$
- sigmoid predict that a binary variable is 1
- Sigmoidal units saturates large and gradient-based learning difficult
- *tanh* is more preferable than $\sigma$ for hidden units
- Compatible with the gradient-based learning with a cost function that can undo the saturation as output units
- are more common in recurrent networks, many probabilistic model and autoencoders

# Architecture Design

Architecture

- Overall network structure
- How many units
- How to connect to each other
- Layer is organized groups of units
- Mostly in a chain structure

$$\mathbf{h}^{(1)} = g^{(1)}(\mathbf{W}^{(1)\mathsf{T}}\mathbf{x} + \mathbf{b}^{(1)})$$
$$\mathbf{h}^{(2)} = g^{(2)}(\mathbf{W}^{(2)\mathsf{T}}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$

# Architecture Design

Architecture

- Main design choices
  - depth of the network
  - width of each layer
- Deeper networks
  - fewer units per layer
  - fewer parameters
  - tend to be harder to optimize
- Ideal network architecture via experimentation guided by monitoring the validation error

# Architecture Design

- Universal approximation theorem
  - A feedforward network with a linear output
  - At least one hidden layer with squashing activation function with enough number of hidden units
  - can approximate any continuous function on a closed and bounded subset of $\mathbb{R}^n$ with any desired (nonzero) error
- MLP able to represent function of interest though learning not guaranteed by the training
  - optimization algorithm might fail to find the corresponding parameter values
  - overfitting

# Architecture Design

- Universal approximation theorem and Depth
  - A feedforward network with one layer can represent any function being infeasible large
  - Deeper models reduce the number of units and can reduce the generalization error
- The number of linear regions carved out by a deep rectifier network[8]

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

where input dimension $d$, depth $l$, units per hidden layer $n$

- Number of linear regions for a maxout network with k filters per unit

$$O\left(k^{(l-1)+d}\right)$$

[8] Guido Montúfar et al. "On the Number of Linear Regions of Deep Neural Networks". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 2924–2932.

# Architecture Design

Empirically deeper networks generalize better

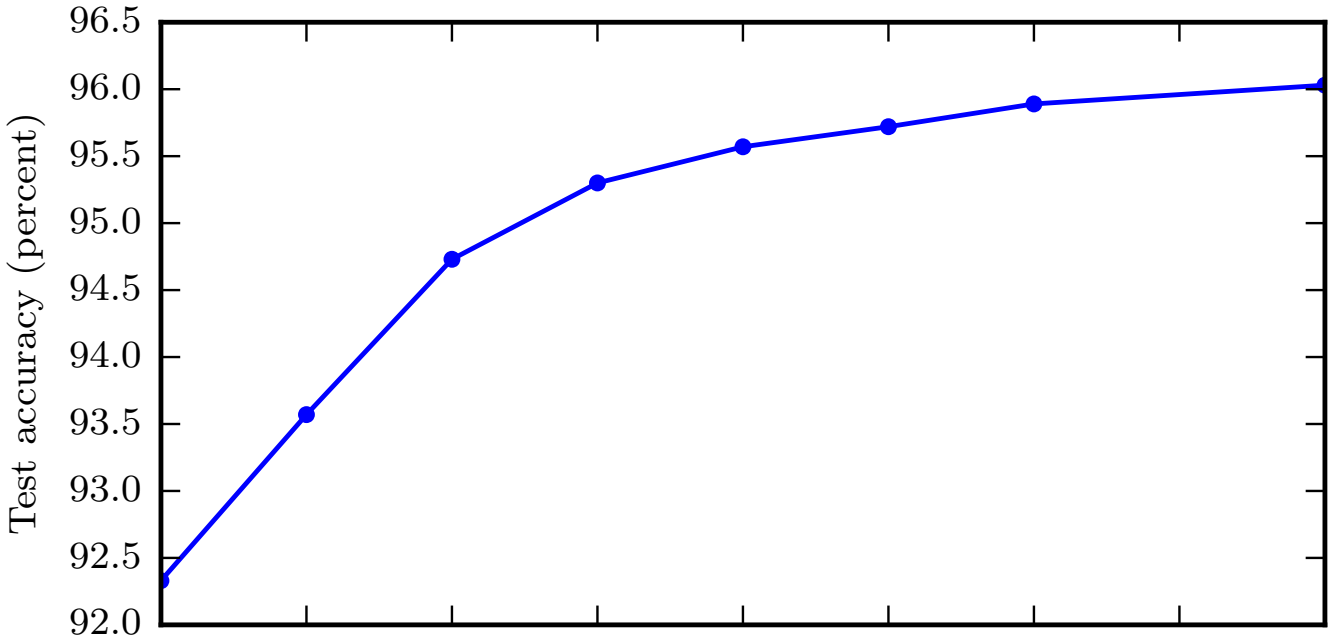



Figure 8: Effect of depth[9].

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

# Architecture Design

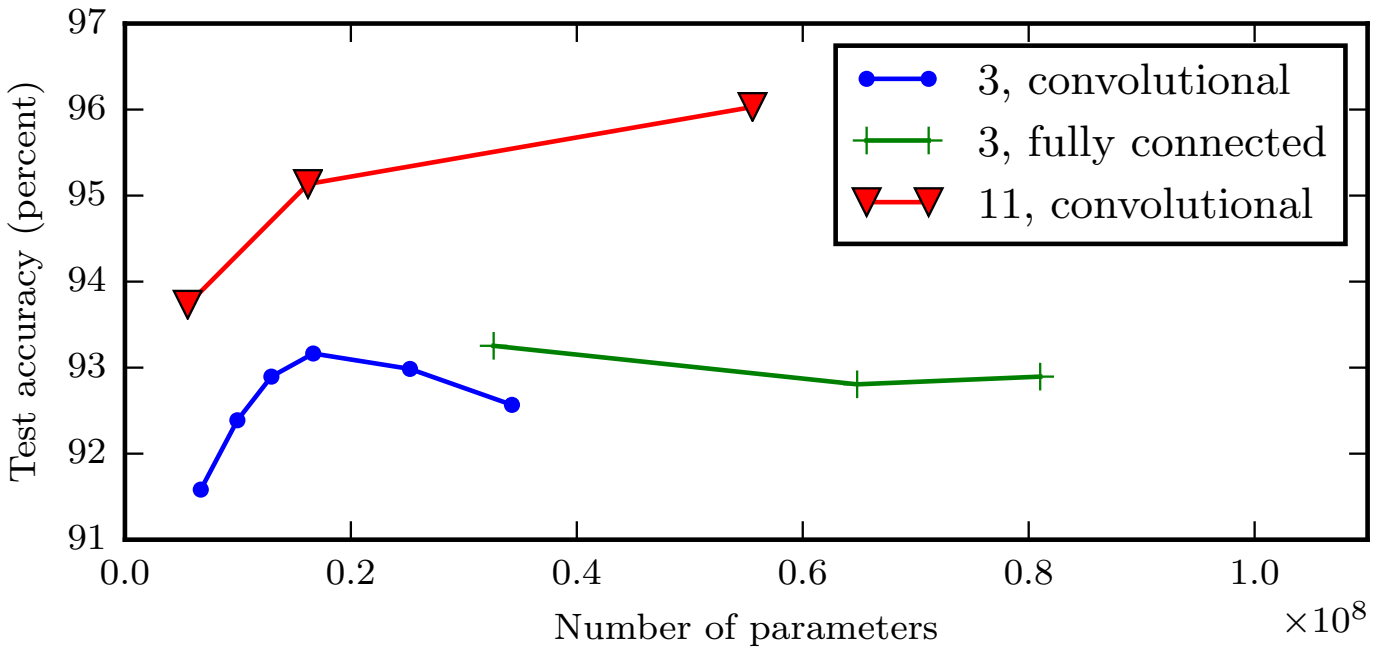



Figure 9: Effect of number of parameters[10].

[10] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

# Back-Propagation

- Forward propagation is flow from $\mathbf{x}$ to $\hat{\mathbf{y}}$
- During training forward propagation continue onward until cost $J(\theta)$
- **backprop** from cost $J(\theta)$ to network backwards to compute the gradient
- Numerical evaluation of analytical gradient expression computationally expensive
- Backprop makes it simple and inexpensive
- Backprop is a method of computing gradient
- Notation
  - $\nabla_{\mathbf{x}} f(\mathbf{x}, \vec{y})$ is the gradient of an arbitrary function $f$
  - $x$ is a set of variable derivatives desired
  - $y$ is input to function derivatives not desired
  - $\nabla_{\theta} J(\theta)$ is gradient of cost function

# Back-propagation

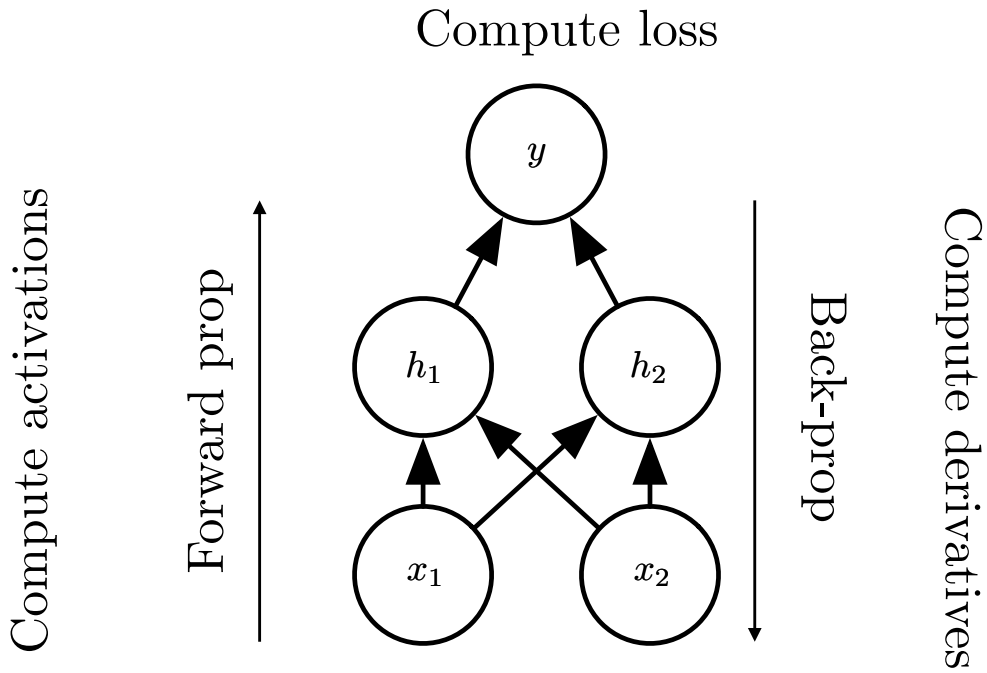



Figure 10: Back-propagation[11].

[11] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

# Back-propagation

Computational Graphs

- Each node a variable
- A variable might be scalar, vector, matrix or tensor
- An **operation** a simple function of one or more variables
- Functions more complex, composed of many operations
- directed edge from $x$ to $y$ indicates $x$ used to calculate $y$

# Back-propagation

## Examples of Computational Graphs

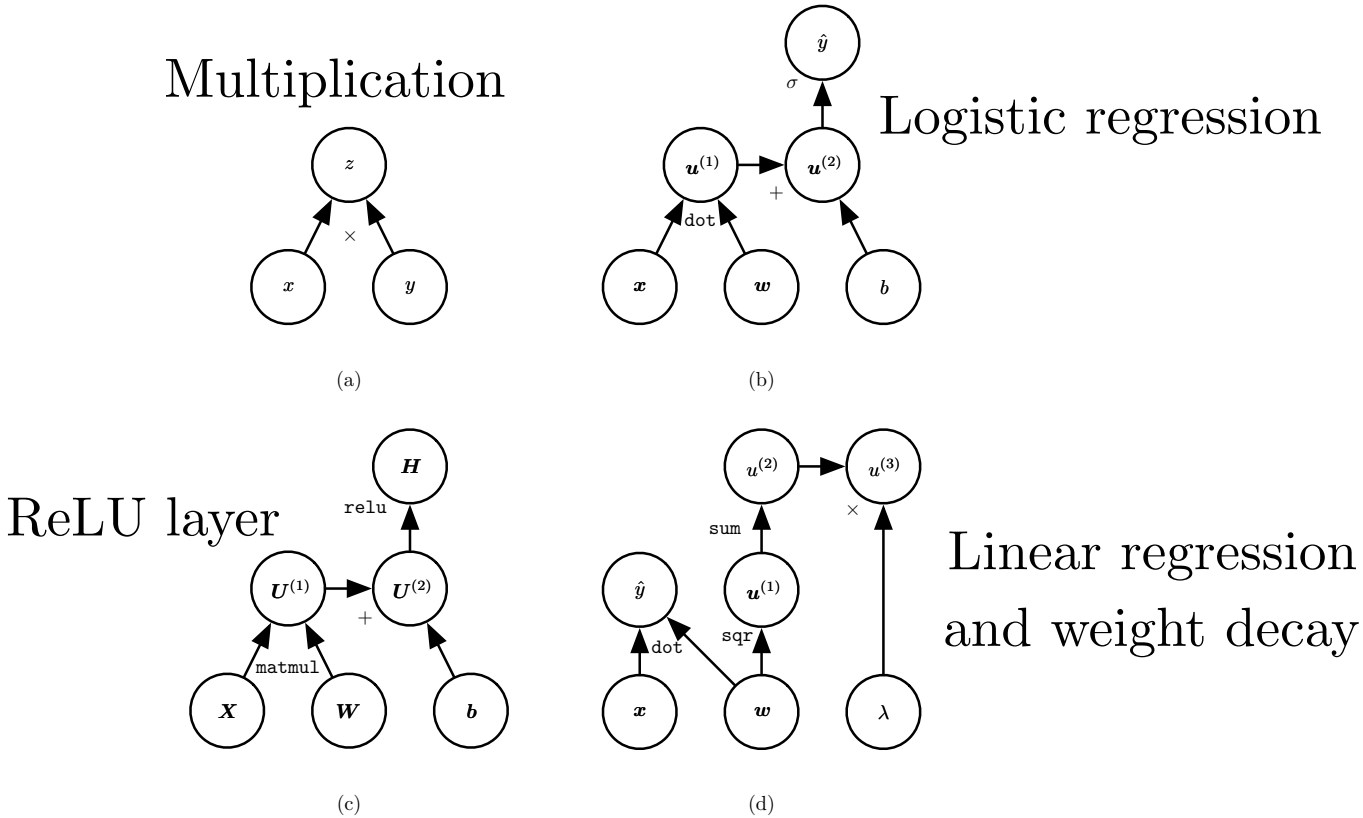



Figure 11: Computation Graphs[12].

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

# Back-propagation

Back-propagation is a chain rule of calculus

- Highly efficient
- $x$ is a real number and $f, g : \mathbb{R} \to \mathbb{R}$, $y = g(x)$ and $z = f(g(x)) = f(y)$
- Chain rule:

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

- For $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, $g : \mathbb{R}^m \to \mathbb{R}^n$ and $f : \mathbb{R}^n \to \mathbb{R}$

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

$$\nabla_x z = \left(\frac{\partial \mathbf{y}}{\partial \mathbf{x}}\right)^{\mathsf{T}} \nabla_{\mathbf{y}} z$$

where $\frac{\partial \mathbf{y}}{\partial \mathbf{x}}$ is $n \times m$ Jacobian matrix of $g$

# Repeated Subexpressions

Input $w \in \mathbb{R}$, $f : \mathbb{R} \to \mathbb{R}$, $x = f(w)$, $y = f(x)$, $z = f(y)$

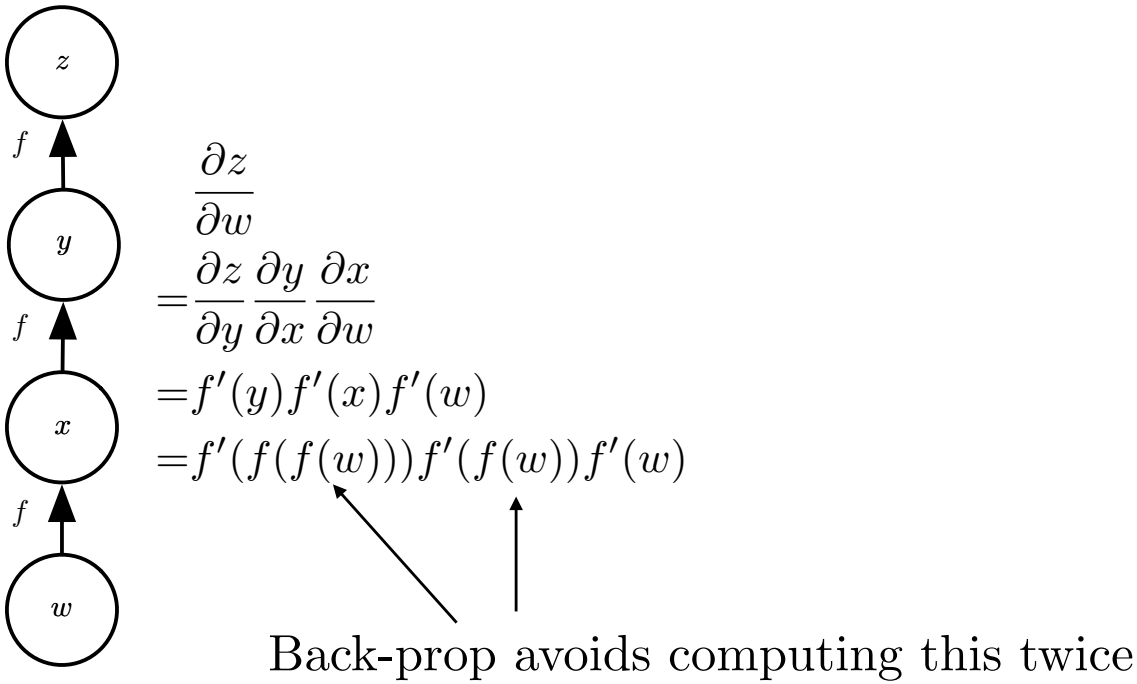



Figure 12: Repeated Subexpressions[13].

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

# Symbol-to-symbol derivatives

Algebraic and graph-based representations are **symbolic representations**
Symbol-to-number differentiation: Torch, Caffe
Symbol-to-symbol differentiation: Theano, Tensorflow

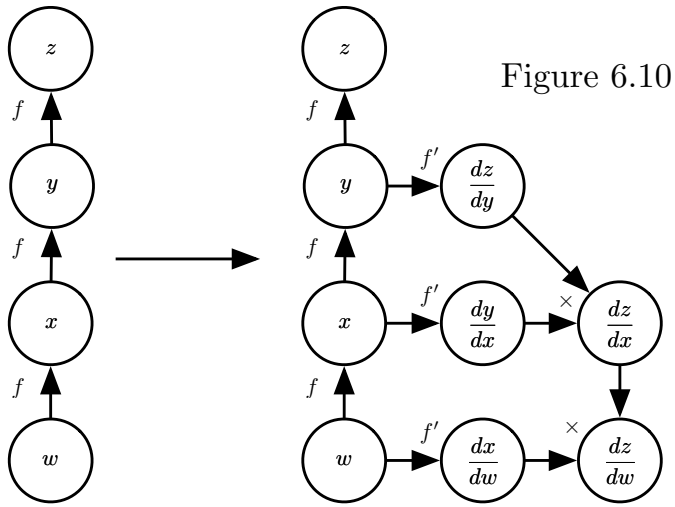



Figure 6.10

Figure 13: Symbol-to-symbol example[14].

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

# Forward Pass Fully Connected MLP

**Require:** Network depth, $l$
**Require:** $\boldsymbol{W}^{(i)}, i \in \{1, \ldots, l\}$, the weight matrices of the model
**Require:** $\boldsymbol{b}^{(i)}, i \in \{1, \ldots, l\}$, the bias parameters of the model
**Require:** $\boldsymbol{x}$, the input to process
**Require:** $\boldsymbol{y}$, the target output
$\boldsymbol{h}^{(0)} = \boldsymbol{x}$
**for** $k = 1, \ldots, l$ **do**
$\boldsymbol{a}^{(k)} = \boldsymbol{b}^{(k)} + \boldsymbol{W}^{(k)} \boldsymbol{h}^{(k-1)}$
$\boldsymbol{h}^{(k)} = f(\boldsymbol{a}^{(k)})$
**end for**
$\hat{\boldsymbol{y}} = \boldsymbol{h}^{(l)}$
$J = L(\hat{\boldsymbol{y}}, \boldsymbol{y}) + \lambda \Omega(\theta)$

Figure 14: Forward Pass Algorithm for a MLP[15].

[15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

# Backward Pass Fully Connected MLP

After the forward computation, compute the gradient on the output layer:
$\boldsymbol{g} \leftarrow \nabla_{\hat{\boldsymbol{y}}} J = \nabla_{\hat{\boldsymbol{y}}} L(\hat{\boldsymbol{y}}, \boldsymbol{y})$
**for** $k = l, l-1, \ldots, 1$ **do**
  Convert the gradient on the layer's output into a gradient into the pre-nonlinearity activation (element-wise multiplication if $f$ is element-wise):
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{a}^{(k)}} J = \boldsymbol{g} \odot f'(\boldsymbol{a}^{(k)})$
  Compute gradients on weights and biases (including the regularization term, where needed):
  $\nabla_{\boldsymbol{b}^{(k)}} J = \boldsymbol{g} + \lambda \nabla_{\boldsymbol{b}^{(k)}} \Omega(\theta)$
  $\nabla_{\boldsymbol{W}^{(k)}} J = \boldsymbol{g}\, \boldsymbol{h}^{(k-1)\top} + \lambda \nabla_{\boldsymbol{W}^{(k)}} \Omega(\theta)$
  Propagate the gradients w.r.t. the next lower-level hidden layer's activations:
  $\boldsymbol{g} \leftarrow \nabla_{\boldsymbol{h}^{(k-1)}} J = \boldsymbol{W}^{(k)\top}\, \boldsymbol{g}$
**end for**

Figure 15: Backward Pass Algorithm for a MLP[16].

[16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

# Next Week Recurrent Neural Networks

Questions?

References

 Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

 Guido Montúfar et al. "On the Number of Linear Regions of Deep Neural Networks". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 2924–2932.

 Johan Suykens. *Lecture notes in Artificial Neural Networks*. 2015.