Feedforward Neural Networks and Backpropagation

Changyou Chen

Department of Computer Science and Engineering
Universitpy at Buffalo, SUNY
changyou@buffalo.edu

February 12, 2019

Deep Feedforward Neural Networks

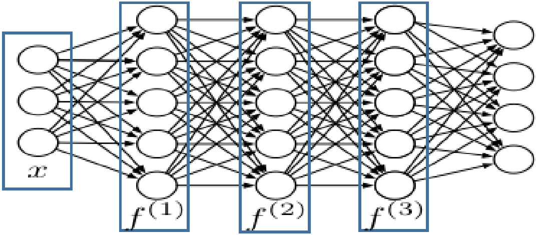
Deep Feedforward Neural Networks

1. Also called:
    - Feedforward neural networks (FNN).
    - Multilayer Perceptrons (MLP).
2. Goal: approximate some (nonlinear) function f
    - *e.g.*, a classifier $y = f(\mathbf{x})$ maps an input $\mathbf{x}$ to a category y.
3. The mapping f is parameterized by θ, thus written as $f(\mathbf{x}; \theta)$.
    - The goal is to learn θ from the data that results in the best function approximation.

How to defined f? and how to learn θ?

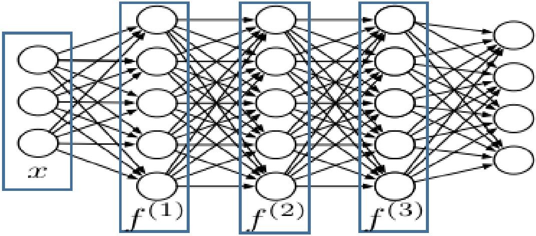
Feedforward Networks

1. It is called Feedforward because
   - information flows through function being evaluated from $\mathbf{x}$ through intermediate computations, and finally to output y.
2. Called networks because they are composed of many different functions.
3. Model is associated with a directed acyclic graph describing how functions are composed, *e.g.*, functions $f^{(1)}$, $f^{(2)}$, $f^{(3)}$ connected in a chain to form $f(\mathbf{x}) = f^{(3)}\left(f^{(2)}\left(f^{(1)}(\mathbf{x})\right)\right) \triangleq f^{(3)} \circ f^{(2)} \circ f^{(1)}(\mathbf{x})$.
   - $f^{(1)}$ is called the first layer of the network, $f^{(2)}$ is the second layer, *etc*.



Terminology

1. Overall length of the chain is the depth of the model.
2. The name deep learning arises from this terminology.
3. Final layer of a feedforward network is called the output layer.
4. All layers between input and output are called hidden layers (units) because they are not from the data:
   - Hidden layers are usually deterministic.
   - Sometimes become stochastic in deep generative models for more flexible modeling ability.



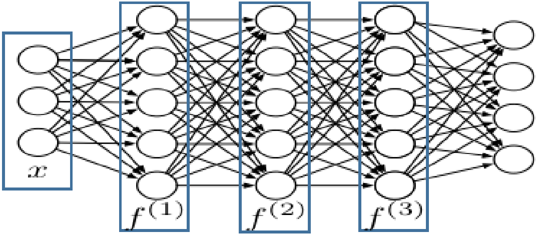
An Example of FNN

1. Define $f^{(i)}(\tilde{\mathbf{x}})$ as

$$f^{(i)}(\tilde{\mathbf{x}}) = \sigma\left({\mathbf{W}^{(i)}}^T \tilde{\mathbf{x}} + \mathbf{b}^{(i)}\right)$$

$$\sigma(\mathbf{a})_k \triangleq \frac{1}{1 + e^{-a_k}}$$

2. Nonlinearity comes in via the sigmoid function.



Using FNN to Solve the XOR Problem

1. XOR: an operation on binary-variable inputs x_1 and x_2:
   - When exactly one value equals 1 it returns 1, otherwise it returns 0:

$$x_1 \oplus x_2 \triangleq \begin{cases} 1 & \text{if } x_1 \neq x_2 \\ 0 & \text{otherwise} \end{cases}$$

   - Target function is $f^*([x_1, x_2]) \triangleq x_1 \oplus x_2$ that we want to learn.
   - Our model is $f([x_1, x_2]; \boldsymbol{\theta})$, where we learn parameters $\boldsymbol{\theta}$ to make f similar to f^*.
2. Not concerned with statistical generalization:
   - Perform correctly on four training points: $X = [0, 0]^T, [0, 1]^T, [1, 0]^T, [1, 1]^T$.
   - Challenge is to fit the training set:
     - ★ we want $f([0, 0]^T; \boldsymbol{\theta}) = f([1, 1]^T; \boldsymbol{\theta}) = 0$ and $f([0, 1]^T; \boldsymbol{\theta}) = f([1, 0]^T; \boldsymbol{\theta}) = 1$, which is nonlinear.

First Try: Linear Model does not Fit

1. Treat it as regression with MSE loss function

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

   - If $f(\mathbf{x};\theta)$ perfectly matches $f^*(\mathbf{x})$, it has the minimum loss (zero loss).

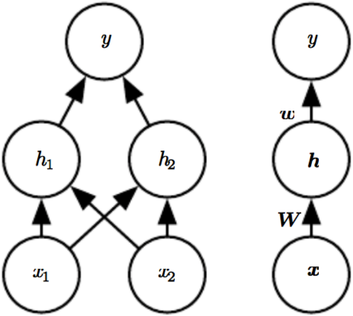
2. Consider a linear model with $\theta = \{\mathbf{w}, \mathbf{b}\}$ such that

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

3. Minimize $J(\theta)$ to get a close-form solution:
   - Differentiate J w.r.t. $\mathbf{w}$ and $\mathbf{b}$ to obtain $\mathbf{w} = 0$ and $\mathbf{b} = 1/2$.
   - Thus $f(\mathbf{x};\theta) = 1/2$, which simply outputs 0.5 everywhere $\Rightarrow$ not equal to $f^*(\mathbf{x})$.

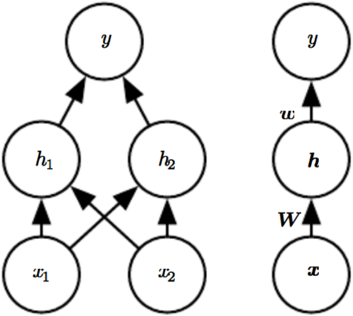
Feedforward Network for XOR

1. Use a simple feedforward network with one hidden layer **h** containing two units.
2. Matrix **W** describes mapping from **x** to **h**.
3. Vector **w** describes mapping from **h** to **y**;
4. Intercept parameters **b** are omitted for simplicity.



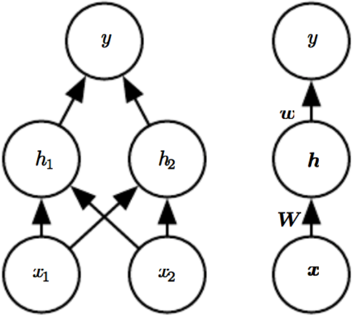
Model Specification

1. Layer 1 (hidden): $\mathbf{h} = f^{(1)}(\mathbf{x}; \mathbf{W}, \mathbf{c})$.
2. Layer 2 (output): $y = f^{(2)}(\mathbf{h}; \mathbf{w}, b)$.
3. Complete model: $f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, b) = f^{(2)}(f^{(1)}(\mathbf{x}))$.



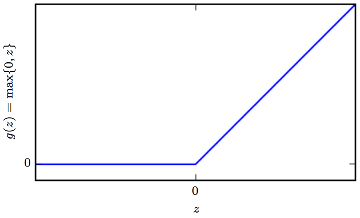
Linear vs. Nonlinear Functions

1. If we choose $f^{(1)}$ and $f^{(2)}$ to be linear functions, *e.g.*, $f^{(1)}(\mathbf{x}) = \mathbf{W}^T\mathbf{x}$ and $f^{(2)}(\mathbf{h}) = \mathbf{w}^T\mathbf{h}$, the completer model is still linear: $f(\mathbf{x}) = \mathbf{w}^T\mathbf{W}^T\mathbf{x}$.
2. Since linear is insufficient, we must use a nonlinear function to describe the features.
3. We use the strategy of neural networks by using a nonlinear activation function g: $\mathbf{h} = g(\mathbf{W}^T\mathbf{x} + \mathbf{c})$.



Activation Functions

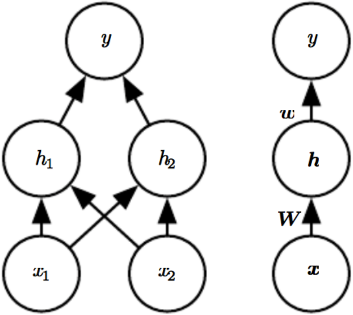
1. Activation function g is typically chosen to be applied element-wise $h_i = g(W_{:,i}^T \mathbf{x} + c_i)$.
2. Default activation function: $g(\mathbf{z}) = \max\{0, \mathbf{z}\}$.
   - Called Rectified Linear Unit (ReLU).
   - Applying ReLU to the output of a linear transformation yields a nonlinear transformation.
   - Function remains close to linear
     - preserve properties that make linear models easy to optimize with gradient-based methods
     - preserve many properties that make linear models generalize well



Specifying the Network using ReLU

1. Activation: $g(\mathbf{z}) = \max\{0, \mathbf{z}\}$ for the first layer.
2. The completer model:

$$f(\mathbf{x}; \mathbf{W}, \mathbf{c}, \mathbf{w}, \mathbf{b}) = f^{(2)}(f^{(1)}(\mathbf{x})) = \mathbf{w}^T \max\{0, \mathbf{W}^T \mathbf{x} + \mathbf{c}\} + \mathbf{b}$$



XOR Solution

1. The model can be learned by gradient (introduced later). For now let's guess the following solution

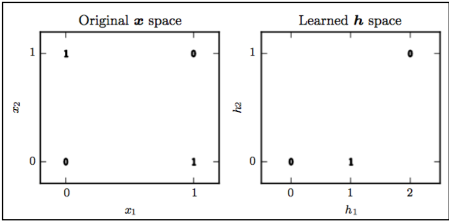
$$\mathbf{W} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \mathbf{c} = \begin{bmatrix} 0 & -1 \end{bmatrix}, \mathbf{w} = \begin{bmatrix} 1 & -2 \end{bmatrix}, b = 0$$

$$f\left(\begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}; \{\mathbf{W}, \mathbf{c}, \mathbf{w}, b\}\right) = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix}$$

Perfectly match!

Learned representation for XOR

1. Two points that must have output 1 have been collapsed into one in the latent space, *i.e.*, points $x = [0, 1]^T$ and $x = [1, 0]^T$ have been mapped into $h = [0, 1]^T$.
2. Data in the latent can be described in a linear model.

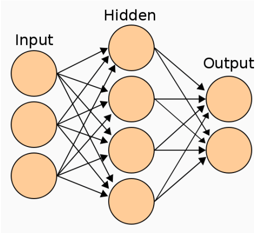


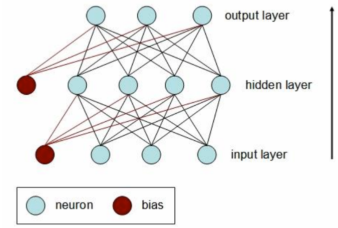
Summary

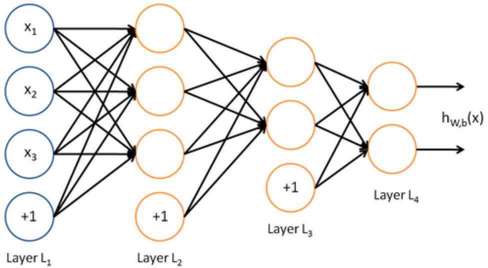
1. In the above FNN, we simply specified its solution, and showed that it achieves zero error.
2. In real situations, there might be billions of parameters and billions of training examples,
   - one cannot simply guess the solution
3. Instead gradient-descent-based optimization is used to find parameters that produce very little error:
   - detailed later.

Hidden Units

Hidden Units in Neural Networks







What a Hidden unit does?

1. Accepts a vector of inputs $\mathbf{x}$ and computes an affine transformation $\mathbf{z} = \mathbf{W}^T \mathbf{x} + \mathbf{b}$.
2. Computes an element-wise non-linear function $g(\mathbf{z})$.
3. Most hidden units are distinguished from each other by the choice of activation function $g(\mathbf{z})$, *e.g.*, ReLU, sigmoid and tanh, and *etc*.

Choice of Hidden Unit

1. ReLu is a popular default choice.
2. Design of hidden units is an active research area that does not have many definitive guiding theoretical principles.
3. Design process is trial and error.

Is Differentiability necessary?

1. Some hidden units are not differentiable at some input points:
   - Rectified Linear Unit $g(\mathbf{z}) = \max\{0, \mathbf{z}\}$ is not differentiable at $\mathbf{z} = 0$.
2. May seem like it invalidates for use in gradient-based learning, however, it performs well in practice.
3. Generally required non-differentiability at only a finite number of points:
   - otherwise there is no gradient backpropagated.

Rectified Linear Unit

1. Activation function $g(\mathbf{z}) = \max\{0, \mathbf{z}\}$
   - easy to optimize due to similarity with linear units.
2. Usually used on top of an affine transformation $\mathbf{h} = g(\mathbf{W}^T\mathbf{x} + \mathbf{b})$.
3. Good practice to set all elements of b to a small value such as 0.1
   - makes it likely that ReLU will be initially active for most training samples and allow derivatives to pass through.

Three generalizations of ReLU

1. Three methods based on using a non-zero slope α_i when $z_i < 0$:

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

- Absolute-value rectification:
  - fix $\alpha_i = -1$ to obtain $g(\mathbf{z}) = |\,\mathbf{z}\,|$.
- Leaky ReLU:
  - fixes α_i to a small value like 0.01.
- Parametric ReLU or PReLU:
  - treats α_i as a parameter.

Maxout Units

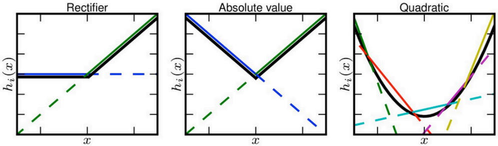
1. Instead of applying element-wise function $g(\mathbf{z})$ as in ReLU, maxout units divide $\mathbf{z}$ into k-groups.
2. Each maxout unit then outputs the maximum element of one of these groups:

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

   where $G(i)$ is the set of indices into the inputs for group i.
3. This provides a way of learning a piecewise linear function that responds to multiple directions in the input x space.
4. Computationally more expensive than ReLU, not as widely used.

Maxout as Learning Activation

1. A maxout unit can learn piecewise linear, convex function with upto k pieces
   - with large enough k, approximate any convex function

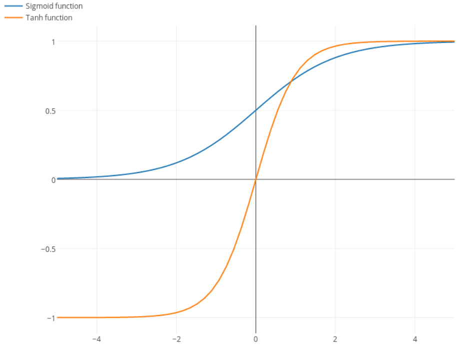


2. A maxout layer with two pieces can learn to implement the same function of the input as a traditional layer using ReLU or its generalizations.

Logistic Sigmoid

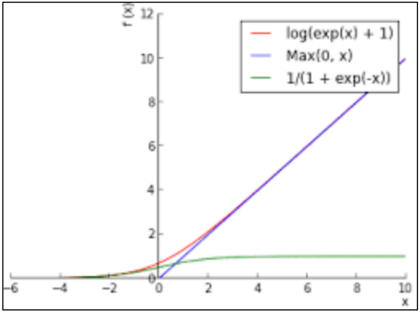
1. Prior to ReLU, most neural networks used logistic sigmoid activation: $g(\mathbf{z}) = \sigma(\mathbf{z}) \triangleq \frac{1}{1+e^{-\mathbf{z}}} \in [0, 1]$.
2. Or the hyperbolic tangent

$$g(\mathbf{z}) = tanh(\mathbf{z}) = \frac{\sinh(\mathbf{z})}{\cosh(\mathbf{z})} = \frac{e^{\mathbf{z}} - e^{-\mathbf{z}}}{e^{\mathbf{z}} + e^{-\mathbf{z}}} = 2\sigma(2\mathbf{z}) - 1 \in [-1, 1]$$



Sigmoid Saturation

1. Sigmoid saturates across most of domain:
   - saturate to 1 when **z** is very positive and 0 when **z** is very negative.
   - output sensitive to input when **z** is near 0.
   - saturation makes gradient-learning difficult, *e.g.*, difficult to set the step size for the whole range.
2. ReLU and Softplus ($\log(\exp(x) + 1)$, smooth version of ReLU) increase (almost linearly) for input >0:
   - easier for gradient learning



Sigmoid vs tanh Activation

1. Hyperbolic tangent typically performs better than logistic sigmoid.
2. It resembles the identity function more closely

$$tanh(0) = 0, \text{ while } \sigma(0) = 1/2$$

3. *tanh* still saturates when $\mathbf{z}$ is very positive or negative.
4. Since tanh is similar to identity near 0, training a deep neural network $\hat{y} = \mathbf{w}^T \tanh(\mathbf{U}^T \tanh(\mathbf{V}^T \mathbf{x}))$ resembles training a linear model $\hat{y} = \mathbf{w}^T \mathbf{U}^T \mathbf{V}^T \mathbf{x}$, as long as the activations can be kept small, thus learning is easier.
5. Sigmoid units still useful in other types of networks such as recurrent neural networks despite the saturation.

Other Hidden Units

1. Many other types of hidden units possible, but used less frequently
   - FNN with $\mathbf{h} = \sin(\mathbf{W}\mathbf{x} + \mathbf{b})$, obtained error rate similar to that using tanh*, approximately 2% error rate on MNIST for a one- or two-layer FNN.
   - Radial Basis: $h_i = \exp\left(\frac{1}{\sigma^2} \| \mathbf{W}_{:,i} - \mathbf{x} \|^2\right)$.
   - Softplus: $g(\mathbf{z}) = \log(1 + e^{\mathbf{z}})$, smooth version of the ReLU.
   - Hard tanh: shaped similar to tanh and the rectifier but it is bounded:

$$g(\mathbf{z}) = \max\left(-1, \min(1, \mathbf{z})\right)$$

* G. Parascandolo & H. Huttunen & T. Virtanen, 2017