CS 4774 Machine Learning

Neural Networks

Yangfeng Ji

Information and Language Processing Lab
Department of Computer Science
University of Virginia

Overview

1. From Perceptrons to MLPs
2. From Logistic Regression to Neural Networks
3. Expressive Power of Neural Networks
4. Learning Neural Networks
5. Computation Graph

From Perceptrons to MLPs

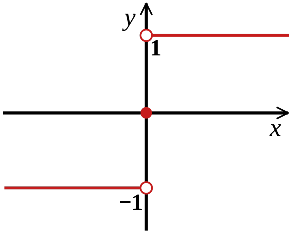
Perceptrons

- $\mathcal{X} = \mathbb{R}^d$
- $\mathcal{Y} = \{-1, +1\}$
- Halfspace hypothesis class

$$\mathcal{H}_{\text{half}} = \{\text{sign}(\langle \boldsymbol{w}, \boldsymbol{x} \rangle) : \boldsymbol{w} \in \mathbb{R}^d\} \tag{1}$$

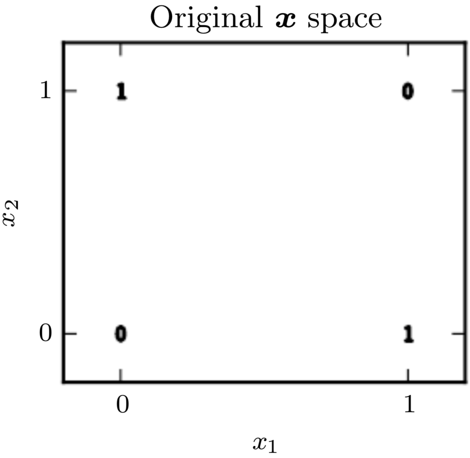
which is an infinite hypothesis space.

The sign function $y = \text{sign}(x)$ is defined as



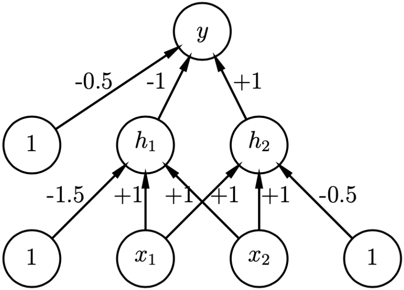
The XOR Problem

$$y = x_1 \oplus x_2 \tag{2}$$



A Multi-Layer Perceptron

The problem can be solved by stacking three perceptrons together, for example,

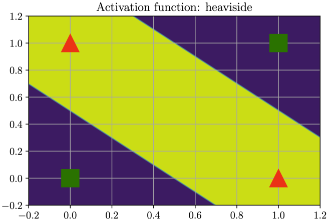


The new model is called Multi-Layer Perceptron (MLP).

Geometric Interpretation

The previous MLP can be write in the mathematical form as

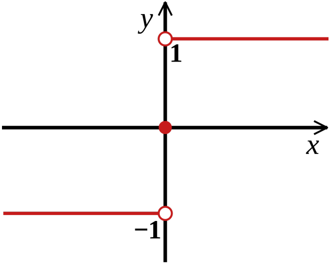
$$
\begin{aligned}
h_1 &= \text{sign}(x_1 + x_2 - 1.5) && (3)\\
h_2 &= \text{sign}(x_1 + x_2 - 0.5) && (4)\\
y &= \text{sign}(-h_1 + h_2 - 0.5) && (5)
\end{aligned}
$$



- Each h_i defines a classifier by deviding the input space into two half-spaces
- Equation 3 forms a non-linear classifier by combining two linear classifiers together

What about Learning?

- Although the previous classifier is simple and intuitive, learning the parameters are not easy, because function $\text{sign}(\cdot)$ is non-differentiable!



- *Solution*: replace $\text{sign}(\cdot)$ function with the Sigmoid function $\sigma(\cdot)$
  - For example, $h_1 = \sigma(w_1 x_1 + w_2 x_2)$
  - In other words, transform each perceptron classifier to a logistic regresion classifier

From Logistic Regression to Neural Networks

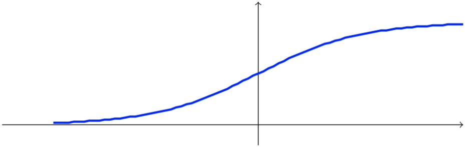
Logistic Regression

- An unified form for $y \in \{-1, +1\}$

$$p(Y = +1 \mid \boldsymbol{x}) = \frac{1}{1 + \exp(-\langle \boldsymbol{w}, \boldsymbol{x} \rangle)} \tag{6}$$

- The sigmoid function $\sigma(a)$ with $a \in \mathbb{R}$

$$\sigma(a) = \frac{1}{1 + \exp(-a)} \tag{7}$$

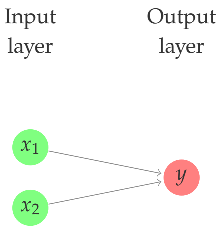


Graphical Representation

- A specific example of LR

$$p(Y = 1 \mid \boldsymbol{x}) = \sigma(\sum_{j=1}^{2} w_j \boldsymbol{x}_{\cdot,j}) \tag{8}$$

- The graphical representation of this LR model is



From LR to Neural Networks

Build upon logistic regression, a simple neural network can be constructed as

$$z_k = \sigma\left(\sum_{j=1}^{d} w_{k,j}^{(1)} x_{\cdot,j}\right) \quad k \in [K] \tag{9}$$

$$P(y = 1 \mid \boldsymbol{x}) = \sigma\left(\sum_{k=1}^{K} w_k^{(o)} z_k\right) \tag{10}$$

- $\boldsymbol{x} \in \mathbb{R}^d$: d-dimensional input
- $y \in \{-1, +1\}$ (binary classification problem)
- $\{w_{k,i}^{(1)}\}$ and $\{w_k^{(o)}\}$ are two sets of the parameters, and
- K is the number of hidden units, each of them has the same form as a LR.

Mathematical Formulation

- Element-wise formulation

$$z_k = \sigma(\sum_{j=1}^{d} w_{k,j}^{(1)} x_{\cdot,j}) \quad k \in [K] \tag{11}$$

$$P(y = +1 \mid \boldsymbol{x}) = \sigma(\sum_{k=1}^{K} w_k^{(o)} z_k) \tag{12}$$

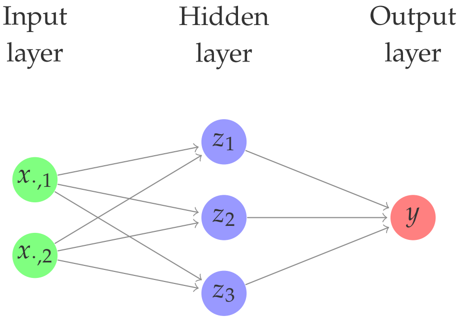
- Matrix-vector formulation

$$\boldsymbol{z} = \sigma(\mathbf{W}^{(1)} \boldsymbol{x}) \tag{13}$$

$$P(y = +1 \mid \boldsymbol{x}) = \sigma((\boldsymbol{w}^{(o)})^\top \boldsymbol{z}) \tag{14}$$

where $\mathbf{W}^{(1)} \in \mathbb{R}^{K \times d}$ and $\mathbf{w}^{(o)} \in \mathbb{R}^K$

Graphical Representation



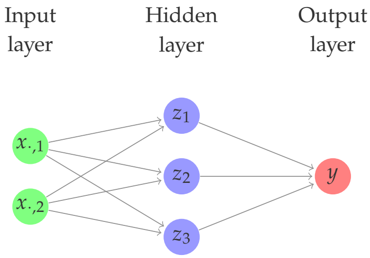
- Depth: 2 (two-layer neural network)
- Width: 3 (the maximal number of units in each layer)

Demo for solve the XOR problem

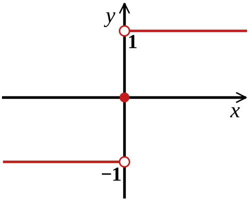
Hypothesis Space

The hypothesis space of neural networks is usually defined by the architecture of the network, which includes

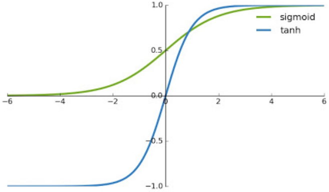
- the nodes in the network,
- the connections in the network, and
- the activation function (e.g., σ, tanh)



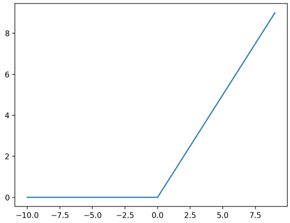
Other Activation Functions



(a) Sign function



(b) Tanh function



(c) ReLU function [Jarrett et al., 2009]

Expressive Power of Neural Networks

Two-layer NNs with Sign Function

Consider a neural network defined by the following functions

$$
\begin{aligned}
z_k &= \mathrm{sign}(\sum_{j=1}^{d} w_{k,j}^{(1)} x_{\cdot,j}) \quad k \in [K] && (15) \\
h(\boldsymbol{x}) &= \mathrm{sign}(\sum_{k=1}^{K} w_k^{(o)} z_k) && (16)
\end{aligned}
$$

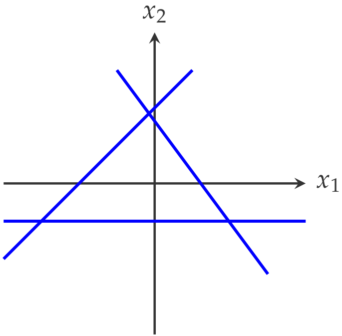
where $\mathrm{sign}(a)$ is the sign function.

$h(\boldsymbol{x})$ can be rewritten as

$$
h(\boldsymbol{x}) = \mathrm{sign}\left(\sum_{k=1}^{K} w_k^{(o)} \cdot \mathrm{sign}(\sum_{j=1}^{d} w_{k,i}^{(1)} x_{\cdot,j})\right) \tag{17}
$$

Decision Boundary

$h(\boldsymbol{x})$ is defined by a combination of K linear predictors



Similar conclusion applies to other activation functions. [Demo]

[Shalev-Shwartz and Ben-David, 2014, Page 274]

Universal Approximation Theorem

Restrict the inputs $x_{\cdot,j} \in \{-1, +1\} \forall j \in [d]$ as binary

Universal Approximation Theorem

For every d, there exists a two-layer neural network (Equations 15 – 16), such that this hypothesis space contains all functions from $\{-1, +1\}^d$ to $\{-1, +1\}$

- The minimal size of network that satisfies the theorem is exponential in d
- Similar results hold for σ as the activation function

[Shalev-Shwartz and Ben-David, 2014, Section 20.3]

Learning Neural Networks

Neural Network Predictions

Consider a binary classification problem with $\mathcal{Y} = \{-1, +1\}$,

- A two-layer neural network gives the following prediction as

$$P(Y = +1 \mid \boldsymbol{x}) = \sigma\left((\boldsymbol{w}^{(o)})^{\mathsf{T}} \sigma(\mathbf{W}^{(1)} \boldsymbol{x})\right) \tag{18}$$

  where $\{\boldsymbol{w}^{(o)}, \mathbf{W}^{(1)}\}$ are the parameters

- Assume the ground-truth label is y, let's introduce an empirical distribution

$$q(Y = y' \mid \boldsymbol{x}) = \delta(y', y) = \begin{cases} 1 & y' = y \\ 0 & y' \neq y \end{cases} \tag{19}$$

Cross Entropy

Given one data point, The loss function of a neural network is usually defined as the cross entropy of the prediction distribution p and the empirical distribution p

$$\begin{aligned} H(q,p) &= -q(Y=+1 \mid \boldsymbol{x}) \log p(Y=+1 \mid \boldsymbol{x}) \\ &\quad -q(Y=-1 \mid \boldsymbol{x}) \log p(Y=-1 \mid \boldsymbol{x}) \end{aligned} \tag{20}$$

Since q is defined with a Delta function, Depending on y, we have

$$H(q,p) = \begin{cases} -\log p(Y=+1 \mid \boldsymbol{x}) & Y=+1 \\ -\log p(Y=-1 \mid \boldsymbol{x}) & Y=-1 \end{cases} \tag{21}$$

It is equivalent to the negative log-likelihood (NLL) function used in learning LR.

ERM

- Given a set of training example $S = \{(\boldsymbol{x}_i, y_i)\}_{i=1}^m$, the loss function is defined as

$$L(\boldsymbol{\theta}) = -\sum_{i=1}^{m} \log p(y_i \mid \boldsymbol{x}_i) \tag{22}$$

where $\boldsymbol{\theta}$ indicates all the parameters in a network.

- For example, $\boldsymbol{\theta} = \{\boldsymbol{w}^{(o)}, \mathbf{W}^{(1)}\}$, for the previously defined two-layer neural network
- Just like learning a LR, we can use gradient-based learning algorithm

Gradient-based Learning

A simple scratch of gradient-based learning[1]

1. Compute the gradient of $\boldsymbol{\theta}$, $\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}$
2. Update the parameter with the gradient

$$\boldsymbol{\theta}^{(\text{new})} \leftarrow \boldsymbol{\theta}^{(\text{old})} - \eta \cdot \left.\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right|_{\boldsymbol{\theta}=\boldsymbol{\theta}^{(\text{old})}} \tag{23}$$

   where η is the learning rate

3. Go back step 1 until it converges

[1] More detail will be discussed in the next lecture

Gradient Computation

Consider the two-layer neural network with one training example $(\boldsymbol{x}, y)$, to further simplify the computation, we assume $y = +1$

$$\log p(y \mid \boldsymbol{x}) = \log \sigma \left((\boldsymbol{w}^{(o)})^\mathsf{T} \sigma(\mathbf{W}^{(1)} \boldsymbol{x}) \right) \tag{24}$$

The gradient with respect to $\boldsymbol{w}^{(o)}$ is

$$\begin{aligned} \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{w}^{(o)}} &= -\frac{\partial \log \sigma\left(\cdot\right)}{\partial \sigma\left(\cdot\right)} \cdot \frac{\partial \sigma\left((\boldsymbol{w}^{(o)})^\mathsf{T} \sigma(\mathbf{W}^{(1)} \boldsymbol{x})\right)}{\partial (\boldsymbol{w}^{(o)})^\mathsf{T} \sigma(\mathbf{W}^{(1)} \boldsymbol{x})} \cdot \frac{\partial (\boldsymbol{w}^{(o)})^\mathsf{T} \sigma(\mathbf{W}^{(1)} \boldsymbol{x})}{\partial \boldsymbol{w}^{(o)}} \\ &= -\left\{ 1 - \sigma\left((\boldsymbol{w}^{(o)})^\mathsf{T} \sigma(\mathbf{W}^{(1)} \boldsymbol{x}) \right) \right\} \cdot \sigma(\mathbf{W}^{(1)} \boldsymbol{x}) \end{aligned} \tag{25}$$

which is in the similar form as the LR updating equation.

Gradient Computation (II)

The gradient with respect to $W^{(1)}$ is

$$\begin{aligned}\frac{\partial L(\boldsymbol{\theta})}{\partial \mathbf{W}^{(1)}} \quad = \quad & -\frac{\partial \log \sigma\left(\cdot\right)}{\partial \sigma\left(\cdot\right)} \cdot \frac{\partial \sigma\left((\boldsymbol{w}^{(o)})^{\mathsf{T}} \sigma(\mathbf{W}^{(1)} \boldsymbol{x})\right)}{\partial (\boldsymbol{w}^{(o)})^{\mathsf{T}} \sigma(\mathbf{W}^{(1)} \boldsymbol{x})} \\ & \cdot \frac{\partial (\boldsymbol{w}^{(o)})^{\mathsf{T}} \sigma(\mathbf{W}^{(1)} \boldsymbol{x})}{\partial \sigma(\mathbf{W}^{(1)} \boldsymbol{x})} \cdot \frac{\partial \sigma(\mathbf{W}^{(1)} \boldsymbol{x})}{\partial \mathbf{W}^{(1)} \boldsymbol{x}} \cdot \frac{\partial \mathbf{W}^{(1)} \boldsymbol{x}}{\partial \mathbf{W}^{(1)}} \end{aligned} \tag{26}$$

- Both of them are the applications of the chain rule in calculus plus some derivatives of basic functions
- In the literature of neural networks, it is called the back-propagation algorithm [Rumelhart et al., 1986]

Computation Graph

Forward Operations

Consider the example of a two-layer neural network

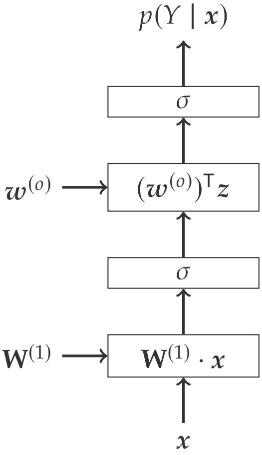
$$P(Y = +1 \mid \boldsymbol{x}) = \sigma\left((\boldsymbol{w}^{(o)})^{\mathsf{T}}\sigma(\mathbf{W}^{(1)}\boldsymbol{x})\right) \tag{27}$$

A neural network is a composition of some basic functions and operations. For example

- $\sigma(\cdot)$
- matrix transpose $(\boldsymbol{w}^{(o)})^{\mathsf{T}}$
- matrix-vector multiplication $\mathbf{W}^{(1)}\boldsymbol{x}$

Forward Graph

The computation graph of the two-layer neural network[2]



[2]For simplicity, the transpose operation is ignored from the graph

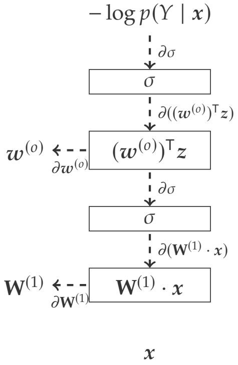
Backward Operations

Similarly, the gradient of neural network parameters are computed with a series of backward operations associated with the derivative of some basic function. For example

- $\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x))$
- $\frac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{a}$
- $\frac{\partial \log(x)}{\partial x} = \frac{1}{x}$
- $\frac{\partial \mathbf{W}\boldsymbol{x}}{\partial \boldsymbol{x}} = \begin{bmatrix} \boldsymbol{x}^\top \\ \vdots \\ \boldsymbol{x}^\top \end{bmatrix}$

Backward Graph

With the chain rule, gradient of the loss function with respect to any parameter can be computed backward step-by-step along the path



Basic Operators

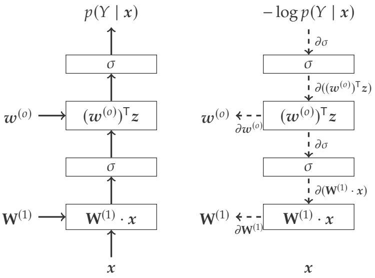
Every basic operator need to be re-implemented, so it can be attached to the computation graph, and also have the forward/backward functions. For example

```
class Add(Function):
    @staticmethod
    def forward(ctx, t1, t2):
        return add_zip(t1, t2)

    @staticmethod
    def backward(ctx, grad_output):
        return grad_output, grad_output
```

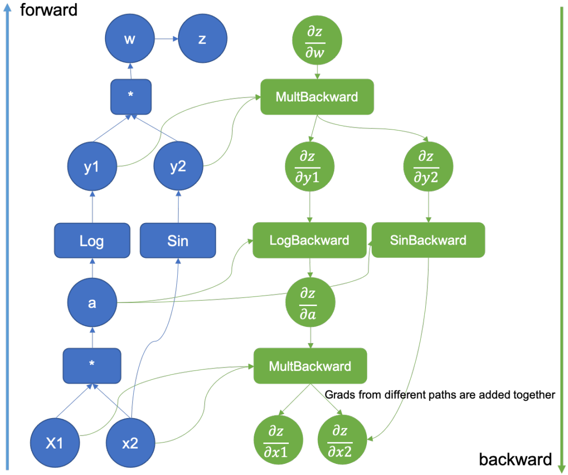
Computation Graph

Perform the forward/backward step with a graph of basic operations (e.g., PyTorch, Tensorflow)



- Modular implementation: implement each module with its forward/backward operations together
- Automatic differentiation: automatically run with the backward step

Another Computation Graph



Link

Reference

Jarrett, K., Kavukcuoglu, K., Ranzato, M., and LeCun, Y. (2009).
What is the best multi-stage architecture for object recognition?
In *Proceedings of the 12th International Conference on Computer Vision*, pages 2146–2153. IEEE.

LeCun, Y. (2020).
Self-supervised learning.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986).
Learning representations by back-propagating errors.
Nature, 323(6088):533–536.

Shalev-Shwartz, S. and Ben-David, S. (2014).
Understanding machine learning: From theory to algorithms.
Cambridge university press.