Deep Learning book, by Ian Goodfellow, Yoshua Bengio and Aaron Courville

Chapter 6 :Deep Feedforward Networks

Benoit Massé   Dionyssos Kounades-Bastian

Linear regression (and classification)

Input vector $\mathbf{x}$

Output vector $\mathbf{y}$

Parameters Weight $\mathbf{W}$ and bias $\mathbf{b}$

$$\text{Prediction}: \ \mathbf{y} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$$

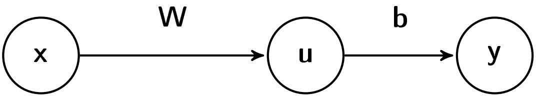
Linear regression (and classification)

Input vector $\mathbf{x}$

Output vector $\mathbf{y}$

Parameters Weight $\mathbf{W}$ and bias $\mathbf{b}$

$$\text{Prediction}: \mathbf{y} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$$



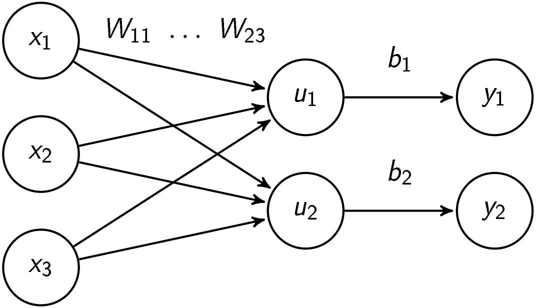
Linear regression (and classification)

Input vector $\mathbf{x}$

Output vector $\mathbf{y}$

Parameters Weight $\mathbf{W}$ and bias $\mathbf{b}$

$$\text{Prediction}: \mathbf{y} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$$



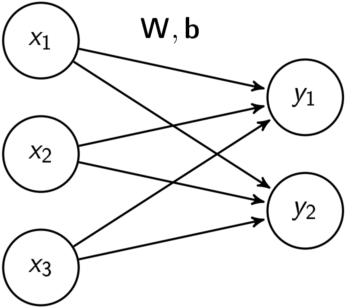
Linear regression (and classification)

Input vector $\mathbf{x}$

Output vector $\mathbf{y}$

Parameters Weight $\mathbf{W}$ and bias $\mathbf{b}$

$$\text{Prediction} : \mathbf{y} = \mathbf{W}^\top \mathbf{x} + \mathbf{b}$$



Linear regression (and classification)

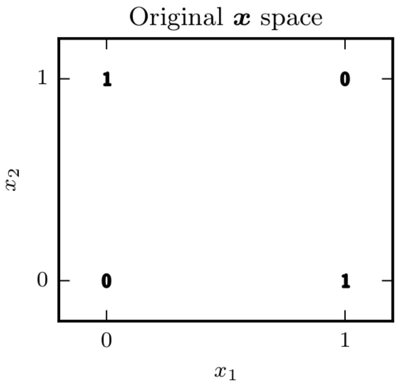
Advantages

- Easy to use
- Easy to train, low risk of overfitting

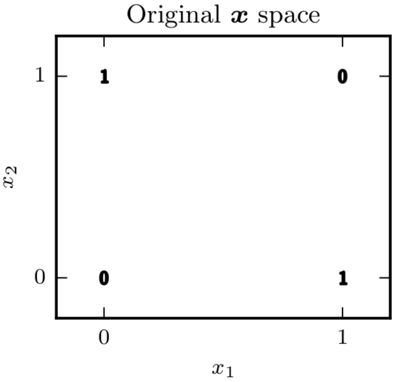
Drawbacks

- Some problems are inherently non-linear

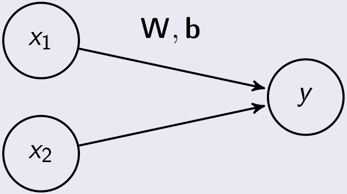
Solving XOR



Solving XOR



Linear regressor

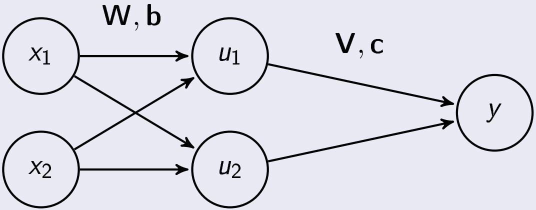


There is no value for $\mathbf{W}$ and $\mathbf{b}$ such that $\forall (x_1, x_2) \in \{0, 1\}^2$

$$\mathbf{W}^\top \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + \mathbf{b} = xor(x_1, x_2)$$

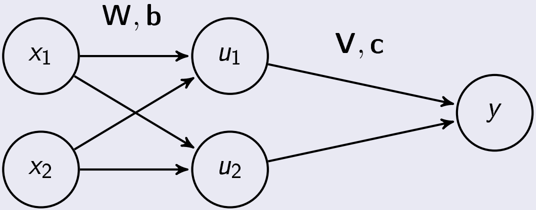
Solving XOR

What about... ?



Solving XOR

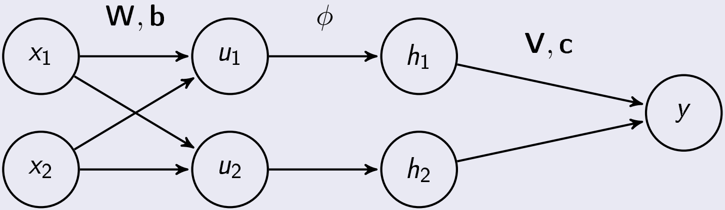
What about... ?



Strictly equivalent :
The composition of two linear operation is still a linear operation

Solving XOR

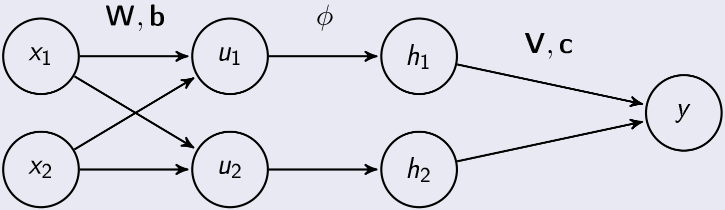
And about... ?



In which $\phi(x) = max\{0, x\}$

Solving XOR

And about... ?



In which $\phi(x) = max\{0, x\}$

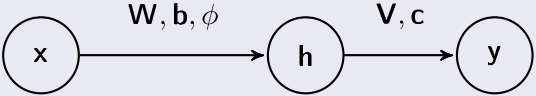
It is possible !

With $\mathbf{W} = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}$, $\mathbf{b} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$, $\mathbf{V} = \begin{pmatrix} 1 \\ -2 \end{pmatrix}$ and $\mathbf{c} = 0$,

$$\mathbf{V}\phi(\mathbf{W}\mathbf{x} + \mathbf{b}) = xor(x_1, x_2)$$

Neural network with one hidden layer

Compact representation



Neural network

Hidden layer with non-linearity

→ can represent broader class of function

Universal approximation theorem

Theorem

A neural network with one hidden layer can approximate any continuous function

More formally, given a continuous function $f : C_n \mapsto \mathbb{R}^m$ where C_n is a compact subset of $\mathbb{R}^n$,

$$\forall \varepsilon, \exists f_{NN}^{\varepsilon} : \mathbf{x} \to \sum_{i=1}^{K} \mathbf{v}_i \phi(\mathbf{w}_i{}^{\top}\mathbf{x} + b_i) + \mathbf{c}$$

such that

$$\forall x \in C_n, \; ||f(x) - f_{NN}^{\varepsilon}(x)|| < \varepsilon$$

Problems

Obtaining the network

The universal theorem gives no information about HOW to obtain such a network

- Size of the hidden layer $\mathbf{h}$
- Values of $\mathbf{W}$ and $\mathbf{b}$

Problems

Obtaining the network

The universal theorem gives no information about HOW to obtain such a network

- Size of the hidden layer $\mathbf{h}$
- Values of $\mathbf{W}$ and $\mathbf{b}$

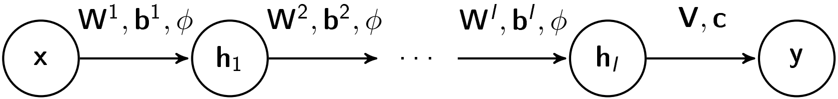
Using the network

Even if we find a way to obtain the network, the size of the hidden layer may be prohibitively large.

Deep neural network

Why Deep ?

Let's stack l hidden layers one after the other; l is called the length of the network.



Properties of DNN

- The universal approximation theorem also apply
- Some functions can be approximated by a DNN with N hidden unit, and would require $\mathcal{O}(e^N)$ hidden units to be represented by a shallow network.

Summary

Comparison

- Linear classifier
  - − Limited representational power
  - + Simple
- Shallow Neural network (Exactly one hidden layer)
  - + Unlimited representational power
  - − Sometimes prohibitively wide
- Deep Neural network
  - + Unlimited representational power
  - + Relatively small number of hidden units needed

Summary

Comparison

- Linear classifier
  - − Limited representational power
  - + Simple
- Shallow Neural network (Exactly one hidden layer)
  - + Unlimited representational power
  - − Sometimes prohibitively wide
- Deep Neural network
  - + Unlimited representational power
  - + Relatively small number of hidden units needed

Remaining problem

How to get this DNN ?

On the path of getting my own DNN

Hyperparameters

First, we need to define the architecture of the DNN

- The depth l
- The size of the hidden layers $n_1, \dots, n_l$
- The activation function ϕ
- The output unit

On the path of getting my own DNN

Hyperparameters

First, we need to define the architecture of the DNN

- The depth l
- The size of the hidden layers $n_1, \ldots, n_l$
- The activation function ϕ
- The output unit

Parameters

When the architecture is defined, we need to train the DNN

- $\mathbf{W}^1, \mathbf{b}^1, \ldots, \mathbf{W}^l, \mathbf{b}^l$

Hyperparameters

- The depth l
- The size of the hidden layers $n_1, \ldots, n_l$
  - ➩ Strongly depend on the problem to solve

Hyperparameters

- The depth l
- The size of the hidden layers $n_1, \ldots, n_l$
  - ➪ Strongly depend on the problem to solve
- The activation function ϕ
  - ➪ ReLU $g : x \mapsto max\{0, x\}$
  - ➪ Sigmoid $\sigma : x \mapsto (1 + e^{-x})^{-1}$
  - ➪ Many others : tanh, RBF, softplus...

Hyperparameters

- The depth l
- The size of the hidden layers $n_1, \ldots, n_l$
  - ⇨ Strongly depend on the problem to solve
- The activation function ϕ
  - ⇨ ReLU $g : x \mapsto max\{0, x\}$
  - ⇨ Sigmoid $\sigma : x \mapsto (1 + e^{-x})^{-1}$
  - ⇨ Many others : tanh, RBF, softplus...
- The output unit
  - ⇨ Linear output $\mathbb{E}[\mathbf{y}] = \mathbf{V}^\top \mathbf{h}_l + \mathbf{c}$
    - For regression with Gaussian distribution $y \sim \mathcal{N}(\mathbb{E}[\mathbf{y}], \mathbf{I})$
  - ⇨ Sigmoid output $\hat{y} = \sigma(\mathbf{w}^\top \mathbf{h}_l + b)$
    - For classification with Bernouilli distribution $P(y = 1|\mathbf{x}) = \hat{y}$

Parameters Training

Objective

Let's define $\boldsymbol{\theta} = (\mathbf{W}^1, \mathbf{b}^1, \dots, \mathbf{W}^l, \mathbf{b}^l)$.
We suppose we have a set of inputs $\mathbf{X} = (\mathbf{x}^1, \dots, \mathbf{x}^N)$ and a set of expected outputs $\mathbf{Y} = (\mathbf{y}^1, \dots, \mathbf{y}^N)$. The goal is to find a neural network f_{NN} such that

$$\forall i, \;\; f_{NN}(x^i, \boldsymbol{\theta}) \simeq \mathbf{y}^i.$$

Parameters Training

Cost function

To evaluate the error that our current network makes, let's define a cost function $\mathcal{L}(\mathbf{X}, \mathbf{Y}, \boldsymbol{\theta})$. The goal becomes to find

$$\underset{\boldsymbol{\theta}}{\operatorname{argmin}}\, \mathcal{L}(\mathbf{X}, \mathbf{Y}, \boldsymbol{\theta})$$

Loss function

Should represent a combination of the distances between every $\mathbf{y}^i$ and the corresponding $f_{NN}(\mathbf{x}^i, \boldsymbol{\theta})$

- Mean square error (rare)
- Cross-entropy

Parameters Training

Find the minimum

The basic idea consists in computing $\hat{\boldsymbol{\theta}}$ such that

$$\nabla_{\boldsymbol{\theta}} \mathcal{L}(\mathbf{X}, \mathbf{Y}, \hat{\boldsymbol{\theta}}) = \mathbf{0}.$$

This is difficult to solve analytically *e.g.* when $\boldsymbol{\theta}$ have millions of degrees of freedom.

Gradient descent

Let's use a numerical way to optimize $\boldsymbol{\theta}$, called the **gradient descent** (section 4.3). The idea is that

$$f(\boldsymbol{\theta} - \varepsilon \mathbf{u}) \simeq f(\boldsymbol{\theta}) - \varepsilon \mathbf{u}^\top \nabla f(\boldsymbol{\theta})$$

So if we take $\mathbf{u} = \nabla f(\boldsymbol{\theta})$, we have $\mathbf{u}^\top \mathbf{u} > 0$ and then

$$f(\boldsymbol{\theta} - \varepsilon \mathbf{u}) \simeq f(\boldsymbol{\theta}) - \varepsilon \mathbf{u}^\top \mathbf{u} < f(\boldsymbol{\theta}).$$

If f is a function to minimize, we have an update rule that improves our estimate.

Parameters Training

Gradient descent algorithm

1. Have an estimate $\hat{\boldsymbol{\theta}}$ of the parameters
2. Compute $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\mathbf{X}, \mathbf{Y}, \hat{\boldsymbol{\theta}})$
3. Update $\hat{\boldsymbol{\theta}} \longleftarrow \hat{\boldsymbol{\theta}} - \varepsilon\nabla_{\boldsymbol{\theta}}\mathcal{L}$
4. Repeat step 2-3 until $\nabla_{\boldsymbol{\theta}}\mathcal{L} <$ threshold

Parameters Training

Gradient descent algorithm

1. Have an estimate $\hat{\boldsymbol{\theta}}$ of the parameters
2. Compute $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\mathbf{X}, \mathbf{Y}, \hat{\boldsymbol{\theta}})$
3. Update $\hat{\boldsymbol{\theta}} \longleftarrow \hat{\boldsymbol{\theta}} - \varepsilon \nabla_{\boldsymbol{\theta}}\mathcal{L}$
4. Repeat step 2-3 until $\nabla_{\boldsymbol{\theta}}\mathcal{L} < \text{threshold}$

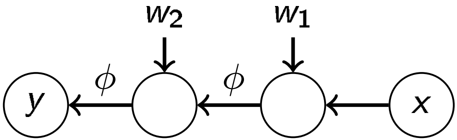
Problem

How to estimate efficiently $\nabla_{\boldsymbol{\theta}}\mathcal{L}(\mathbf{X}, \mathbf{Y}, \hat{\boldsymbol{\theta}})$?

➩ Back-propagation algorithm

Back-propagation for Parameter Learning

- Consider the architecture:



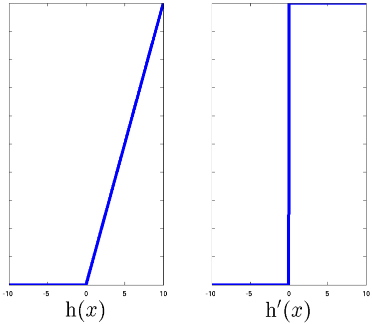
- with function:

$$y = \phi\big(w_2\phi(w_1 x)\big),$$

- some training pairs $T = \{\hat{x}_n, \hat{y}_n\}_{n=1}^{N}$, and
- an activation-function $\phi()$.
- Learn w_1, w_2 so that: Feeding $\hat{x}_n$ results $\hat{y}_n$.

Prerequisite: differentiable activation function

- For learning to be possible $\phi()$ has to be differentiable.
- Let $\phi'(x) = \frac{\partial \phi(x)}{\partial x}$ denote the derivative of $\phi(x)$.
- For example when $\phi(x) = \mathrm{Relu}(x)$ we have:



Gradient-based Learning

- Minimize the loss function $\mathcal{L}(w_1, w_2, T)$.
- We will learn the weights by iterating:

$$\begin{bmatrix} w_1 \\ w_2 \end{bmatrix}^{\text{updated}} = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} - \gamma \begin{bmatrix} \frac{\partial \mathcal{L}}{\partial w_1} \\ \frac{\partial \mathcal{L}}{\partial w_2} \end{bmatrix}, \tag{1}$$

- $\mathcal{L}$ is the loss function (must be differentiable): In detail is $\mathcal{L}(w_1, w_2, T)$ and we want to compute the gradient(s) at w_1, w_2.
- γ is the learning rate (a scalar typically known).

Back-propagation

- Calculate intermediate values on all units:

1. $a = w_1 \hat{x}_n$.
2. $b = \phi(w_1 \hat{x}_n)$.
3. $c = w_2 \phi(w_1 \hat{x}_n)$.
4. $d = \phi\big(w_2 \phi(w_1 \hat{x}_n)\big)$.
5. $\mathcal{L}(d) = \mathcal{L}\left(\phi\big(w_2 \phi(w_1 \hat{x}_n)\big)\right)$.

- The partial derivatives are:

6. $\frac{\partial \mathcal{L}(d)}{\partial d} = \mathcal{L}'(d)$.
7. $\frac{\partial d}{\partial c} = \phi'(w_2 \phi(w_1 \hat{x}_n))$.
8. $\frac{\partial c}{\partial b} = w_2$.
9. $\frac{\partial b}{\partial a} = \phi'(w_1 \hat{x}_n)$.

Calculating the Gradients I

- Apply chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{\partial b}\frac{\partial b}{\partial a}\frac{\partial a}{w_1},$$

$$\frac{\partial \mathcal{L}(d)}{\partial w_2} = \frac{\partial \mathcal{L}(d)}{\partial d}\frac{\partial d}{\partial c}\frac{\partial c}{w_2}.$$

Calculating the Gradients I

- Apply chain rule:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial d} \frac{\partial d}{\partial c} \frac{\partial c}{\partial b} \frac{\partial b}{\partial a} \frac{\partial a}{w_1},$$

$$\frac{\partial \mathcal{L}(d)}{\partial w_2} = \frac{\partial \mathcal{L}(d)}{\partial d} \frac{\partial d}{\partial c} \frac{\partial c}{w_2}.$$

- Start the calculation from left-to-right.
- We propage the gradients (partial products) from the last layer towards the input.

Calculating the Gradients

- And because we have N training pairs:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}(d_n)}{\partial d_n} \frac{\partial d_n}{\partial c_n} \frac{\partial c_n}{\partial b_n} \frac{\partial b_n}{\partial a_n} \frac{\partial a_n}{w_1},$$

$$\frac{\partial \mathcal{L}}{\partial w_2} = \sum_{n=1}^{N} \frac{\partial \mathcal{L}(d_n)}{\partial d_n} \frac{\partial d_n}{\partial c_n} \frac{\partial c_n}{w_2}.$$

Thank you !