Deep Learning Book

Chapter 6
Deep Feedforward Networks

Botian Shi
botianshi@bit.edu.cn

March 3, 2017

You can download the LaTeX source code of this file from Here.

Feedforward Networks

- A type of neural network
  - *Deep feedforward network*
  - *feedforward neural network*
  - *multilayer perceptron (MLP)*
- For a classifier, $y = f^*(\boldsymbol{x})$ maps an input $\boldsymbol{x}$ to category $\boldsymbol{y}$
- Defines a mapping:

$$\boldsymbol{y} = f(\boldsymbol{x}; \boldsymbol{\theta})$$

- Learns the best approximation of f^* with parameter $\boldsymbol{\theta}$
- Feedforward only, no feedback connections.
- The basis of many applications.

Feedforward networks are ...

1. Extreme important.
2. Stepping stone on the path to recurrent neural networks.

Example: convolutional neural network

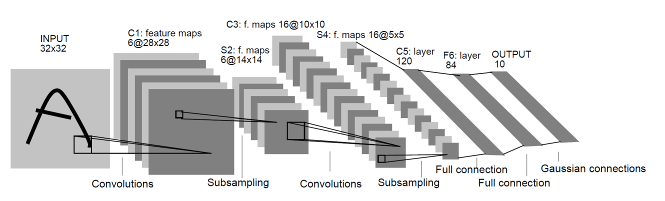


Figure 1: A type of convolutional neural network: LeNet-5 (LeCun et al. [1998])

Multilayer Perceptron

- Why called *networks*?
  - Composing together many different functions.
  - Model is associated with a DAG describing the composition.

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

  - $f^{(1)}$ called first layer of the network
  - $f^{(2)}$ called second layer, and so on
  - $f^{(2)}$ called output layer

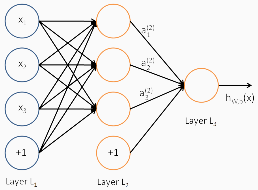


Figure 2:
An example of MLP (from UFLDL)

- During training, we drive $f(x)$ to match $f^*(x)$
- Each example $\boldsymbol{x}$ is accompanied by a label $y \approx f^*(x)$
- At each point $\boldsymbol{x}$, network must produce a value that is close to $\boldsymbol{y}$.
- The learning algorithm must decide how to use those layers to produce the desired output.
- Layers between input and output layer are called *hidden layer*.

The *NEURAL* network

- Inspired by neuroscience.

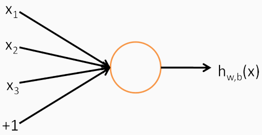


Figure 3: Neuron (from UFLDL)

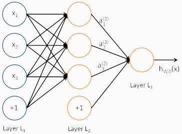


Figure 4: MLP (from UFLDL)

- Each hidden layer is vector-valued. The dimensionality of these hidden layers determines the *width* of the model.
- Each element of the vector may be interpreted as a neuron.
- Layer consist of many *units* that act in parallel, each representing a vector-to-scalar function.
- Each unit resembles a neuron that receives input from many other units and computes its own activation value.

The *NEURAL* network

- The idea of using many layers of vector-valued representation is drawn from neuroscience.
- Modern neural network research $\neq$ perfectly model the brain.
- Feedforward networks $\approx$ function approximation machines.
- Inspired by brain, rather than model a brain.

Understand feedforward networks

- Linear Models

Linear Regression

$$h_\theta(\mathbf{x};\theta) = \theta^T\mathbf{x} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \cdots + \theta_n x_n$$

$$\theta = \arg\min_\theta \mathrm{J}(\theta; X) = \arg\min_\theta \frac{1}{2}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

Logistic Regression

$$h_\theta(\mathbf{x};\theta) = g(\theta^T\mathbf{x}) = \frac{1}{1+e^{-\theta^T\mathbf{x}}}$$

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

General Linear Model

$$p(y;\eta) = b(u)\exp(\eta^T T(y) - a(\eta))$$

Understand feedforward networks

- Efficiently and reliable
- The model capacity is limited to linear functions.
  - The model cannot understand the interaction between any two input variables.
- To extend linear models to represent nonlinear functions of $\boldsymbol{x}$, we can apply the linear model not to $\boldsymbol{x}$ itself but to a transformed input $\phi(\boldsymbol{x})$, where ϕ is a *nonlinear transformation*.
- ϕ provide a set of features describing $\boldsymbol{x}$, or provide a new representation fo $\boldsymbol{x}$

Understand feedforward networks

- How to choose ϕ?
  1. Use a very generic ϕ, e.g., infinite-dimensional ϕ.
     - 🙂 High dimension $\Leftrightarrow$ enough capacity to fit the training set.
     - 🙁 High dimension $\Leftrightarrow$ poor generalization capacity.
     - 🙁 More is less: Runge phenomenon; Gibbs phenomenon.
  2. Manually engineer ϕ.
     - Require human effort for each separate task.
     - Need practitioners specializing in different domains.
  3. Deep learning.
     - Strategy: learn a ϕ.
     - $y = f(x; \boldsymbol{\theta}, w) = \phi(x; \boldsymbol{\theta})^T w$
     - We have parameters θ to learn ϕ from a broad class of functions.
     - We have parameters $\boldsymbol{w}$ to map from $\phi(\boldsymbol{x})$ to the desired output.
     - Here in example, ϕ defining a hidden layer.
     - Deep learning is not simply a *deep* neural network. The ϕ is crucial!

Understand feedforward networks

- Why deep learning?
  1. Parametrize the representation as $\phi(x; \boldsymbol{\theta})$
  2. We can use optimization algorithm to find the $\boldsymbol{\theta}$ that corresponds to a good representation.
  3. Capture the benefit of first and second approach.
     - Being highly generic: using a very broad family $\phi(x; \boldsymbol{\theta})$
     - Human practitioners can encode their knowledge by designing families $\phi(x; \boldsymbol{\theta})$.
     - Human designer only needs to find the right general function family rather than finding precisely the right function.
- The general principle of deep learning is to improve models by learning feature representation.
- Feedforward networks are the application of this principle to learning deterministic mappings from x to y that lack feedback connections.

Deploy a feedforward network

Training a feedforward network requires the same design decisions for linear model:

- Define the form of input and output units.
- Designthe architecture of the network
  1. How many layers the network should contain
  2. How these networks should be connected to each other
  3. How many units should be in each layer.
- Choose activation functions for hidden layers.
- Define a cost function
- Choose an optimizer (gradient descent algorithm and its modern generalizations) to train the network.
- Use back-propagation algorithm to compute the gradients of complicated functions.

Example: Learning XOR

- The XOR function is an operation on two binary values, x_1 and x_2.
- Target: $y = f^*(\boldsymbol{x})$
- Model: $y = f(\boldsymbol{x}; \boldsymbol{\theta})$
- Learning algorithm will adapt the parameters $\boldsymbol{\theta}$ to make f as similar as possible to f^*
- Regression problem
- use mean squared error(MSE) loss function:

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

- Suppose our model is:

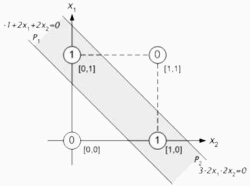
$$f(\boldsymbol{x}; \boldsymbol{w}, b) = \boldsymbol{x}^T \boldsymbol{w} + b$$

- After solving the equations, we obtain $\boldsymbol{w} = \boldsymbol{0}$ and $b = \frac{1}{2}$; $f(\boldsymbol{x}) = \frac{1}{2}$; $J(\boldsymbol{\theta}) = \frac{1}{4}$
- We will never achieve 100% accuracy. Why?

Table 1: XOR

x_1	x_2	y
0	0	0
0	1	1
1	0	1
1	1	0

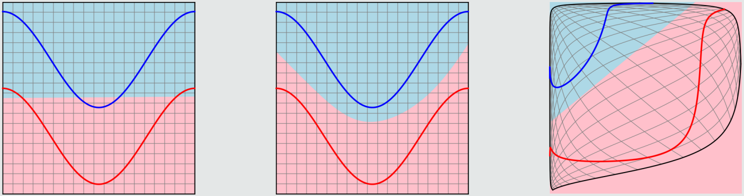
Figure 5: XOR problem



Solution of XOR problem

- Why linear model cannot deal with XOR problem?
- Solution? Nonlinear or transform space!

Figures from colah's blog

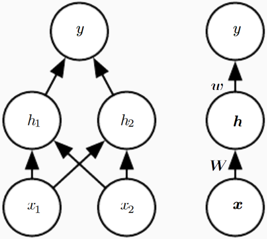


- One way to solve this problem is to use a model that learns a different feature space in which a linear model is able to represent the solution.
- Non-linear models: Neural Networks, SVM, etc.

A simple feedforward network

- A vector of hidden units $\boldsymbol{h}$ that are computed by a function $f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$
- The values of these hidden units are then used as the input for a second layer.
- The second layer is the output layer of the network.
- The output layer is still just a linear regression model, but now it is applied to $\boldsymbol{h}$ rather than to $\boldsymbol{x}$.
- The network now contains two functions chained together: $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$.
- the complete model is: $f(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c}, \boldsymbol{w}, b) = f^{(2)}(f^{(1)}(\boldsymbol{x}))$

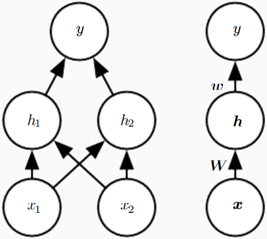
Figure 6: A feedforward network with one hidden layer and two hidden units



A simple feedforward network

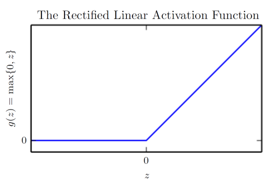
- $\boldsymbol{h} = f^{(1)}(\boldsymbol{x}; \boldsymbol{W}, \boldsymbol{c})$ and $y = f^{(2)}(\boldsymbol{h}; \boldsymbol{w}, b)$
- What function should $f^{(1)}$ compute? We may be tempting to make $f^{(1)}$be linear as well?
- Unfortunately, if $f^{(1)}$ were linear, then the feedforward network as a whole would remain a linear function of its input.
- suppose $f^{(1)}(\boldsymbol{x}) = \boldsymbol{W}^T\boldsymbol{x}$ and $f^{(2)}(\boldsymbol{h}) = \boldsymbol{h}^T\boldsymbol{w}$. Then $f(\boldsymbol{x}) = \boldsymbol{w}^T\boldsymbol{W}^T\boldsymbol{x}$. We could represent this function as $f(\boldsymbol{x}) = \boldsymbol{x}^T\boldsymbol{w'}$ where $\boldsymbol{w'} = \boldsymbol{W}\boldsymbol{w}$
- Clearly, we must use a nonlinear function to describe the features.

Figure 6: A feedforward network with one hidden layer and two hidden units



Activation Function

- Most neural networks describe the features using an affine transformation controlled by learned parameters, followed by a fixed, nonlinear function called an *activation function*.
- We define: $\boldsymbol{h} = g(\boldsymbol{W}^T\boldsymbol{x} + \boldsymbol{c})$, where $\boldsymbol{W}$ provides the weights of a linear transformation and $\boldsymbol{c}$ the biases.
- The activation function g is typically chosen to be a function that is applied element-wise, with $h_i = g(\boldsymbol{x}^T\boldsymbol{W}^{:,i} + c_i)$.
- In modern neural networks, the default recommendation is to use the *rectified linear unit* or ReLU (Jarrett et al. [2009], Nair and Hinton [2010], Glorot et al. [2011]) defined by the activation function $g(z) = \max\{0, z\}$



Solution of XOR problem

We can now specify our complete network as

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

We can now specify a solution to the XOR problem. Let

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

Let $\boldsymbol{X}$ be the design matrix containing all four points in the binary input space:

$$\boldsymbol{x} = \begin{bmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{bmatrix}$$

Solution of XOR problem

The first step in the neural network is to multiply the input matrix by the first layer's weight matrix:

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

Next, we add the bias vector c, to obtain:

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

Solution of XOR problem

To finish computing the value of $\boldsymbol{h}$ for each example, we apply the rectified linear transformation and then, we can use a linear model to solve the problem. We finish by multiplying by the weight vector $\boldsymbol{w}$

$$\mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{c}) = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 2 & 0 \\ 2 & 1 \end{bmatrix}; \qquad \mathrm{ReLU}(\boldsymbol{X}\boldsymbol{W} + \boldsymbol{c}) \times \boldsymbol{w} = \begin{bmatrix} 0 \\ 1 \\ 1 \\ 0 \end{bmatrix};$$

The neural network has obtained the correct answer for every example in the batch.

In this example, we simply specified the solution, then showed that it obtained zero error. In a real situation, there might be billions of model parameters and billions of training examples, so one cannot simply guess the solution as we did here.

Instead, a *gradient-based optimization algorithm* can find parameters that produce very little error.

Gradient-Based Learning

- Use *Gradient descent* algorithm to train a neural network.
- There are some difference between linear models we have seen so far and the neural network.
- Nonlinearity $\Longrightarrow$ non-convex loss functions.
- Iterative training, gradient-based optimization:
- Drive the cost function to a very low value, rather than the linear equation solvers (global convergence guarantee).
- Convex optimization converges starting from any initial parameters (in theory).
- *Stochastic gradient descent* applied to non-convex loss functions has no such convergence guarantee, and is sensitive to the values of the initial parameters.
- **Initialize all weights to small random values and the biases may be initialized to zero or to small positive values**.

Cost Functions

- The cost functions for neural networks are more or less the same as those for other parametric models, such as linear models.
- Parametric model defines a distribution $p(\boldsymbol{y}|\boldsymbol{x};\boldsymbol{\theta})$ and we simply use the principle of maximum likelihood.
- Cross-entropy cost function.
- The total cost function $=$ primary cost functions $+$ regularization term.
- Regularization term: weight decay.

$$J(W,b) = \left[\frac{1}{m}\sum_{i=1}^{m}(\frac{1}{2}\|h_{W,b}(x^{(i)}) - y^{(i)}\|^2)\right] + \frac{\lambda}{2}\sum_{l=1}^{n_l-1}\sum_{i=1}^{s_l}\sum_{j=1}^{s_l+1}\left(W_{ji}^{(l)}\right)^2$$

- More advanced regularization strategies for neural networks will be describe in next chapter.

For your information: Cross-entropy cost function in logistic regression with 1 unit

- We can use MSE to be the cost function:

$$C = \frac{(y-a)^2}{2} \text{ where } a = \sigma(z) \text{ and } z = WX+b$$

- The gradient of the cost function C:

$$\frac{\partial C}{\partial w} = (a-y)\sigma'(z)x = a\sigma'(z)$$

- Then update parameters:

$$w \leftarrow w - \eta\frac{\partial C}{\partial w} = w - \eta \times a \times \sigma'(z)$$

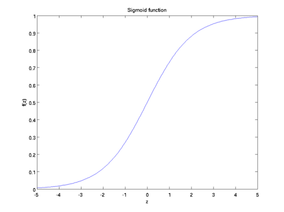


Figure 7: Sigmoid Function (from UFLDL)

For your information: Cross-entropy cost function in logistic regression with 1 unit

- Cross-entropy cost function:

$$C = -\frac{1}{n} \sum_{x} \left[y \ln a + (1 - y) \ln(1 - a) \right]$$

- The gradient of the cost function C:

$$\frac{\partial C}{\partial w} = \frac{1}{n} \sum_{x} x_j (\sigma(z) - y)$$

Learning Conditional Distributions with Maximum Likelihood

- Most modern neural networks are trained using maximum likelihood.
- This means that the cost function is simply the negative log-likelihood, equivalently described as the cross-entropy between the training data an the model distribution:

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

- The specific form of the cost function changes from model to model, depending on the specific form of $\log p_{model}$. For example, if $p_{model}(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\boldsymbol{y}; f(\boldsymbol{x};\boldsymbol{\theta}), I)$, the we recover the mean squared error cost,

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

- An advantage of this approach of deriving the cost function from maximum likelihood is that it removes the burden of designing cost functions for each model. Specifying a model $p(\boldsymbol{y}|\boldsymbol{x})$ automatically determines a cost function $\log p(\boldsymbol{y}|\boldsymbol{x})$.

Output Units

- Any kind of neural network unit that may be used as an output can also be used as a hidden unit.
- Here, we focus on the use of these units as outputs of the model, but in principle they can be used internally as well.
- Throughout this section, we suppose that the feedforward network provides a set of hidden features defined by $\boldsymbol{h} = f(\boldsymbol{x}; \boldsymbol{\theta})$
- The role of the output layers is then to provide some additional transformation from the features to complete the task that the network must perform.

Output Units
Linear Units for Gaussian Output Distributions

- One kind of output unit is based on an affine transformation with no nonlinearity. (Linear Units)
- Given features h, a layer of linear output units produces a vector $\hat{y} = W^T h + b$
- Linear output layers are often used to produce the mean of a conditional Gaussian distribution:

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

- Maximizing the log-likelihood is equivalent to minimizing the mean squared error.
- Because linear units do not saturate, they pose little difficulty for gradient-based optimization algorithms.

Output Units
Sigmoid Units for Bernoulli Output Distributions

- Many tasks require predicting the value of a binary variable y.
- Classification problems with two classes can be cast in this form.
- The neural net needs to predict only $P(y = 1|\boldsymbol{x})$.
- A valid probability must lie in the interval $[0, 1]$.

$$P(y = 1|\boldsymbol{x}) = \max\left\{0, min\left\{1, \boldsymbol{w}^T\boldsymbol{h} + b\right\}\right\}$$

- But we would not be able to train it very effectively with gradient descent.
- Any time that $\boldsymbol{w}^T\boldsymbol{h} + b$ strayed outside the unit interval, the gradient of would be 0.
- So we use sigmoid output units combined with maximum likelihood.

- A sigmoid output unit is defined by

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

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

References

Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

Kevin Jarrett, Koray Kavukcuoglu, Yann LeCun, et al. What is the best multi-stage architecture for object recognition? In *Computer Vision, 2009 IEEE 12th International Conference on*, pages 2146–2153. IEEE, 2009.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

References II

Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.