Machine learning

BY YOUJUN HU

yjhu@ipp.cas.cn

1 Introduction

Artificial intelligence (AI) research has tried many different approaches since its founding. In the first decades of the 21st century, the AI research is dominated by highly mathematical optimization machine learning (ML), which has proved successful in solving many challenging real life problems.

Many problems in AI can be solved theoretically by searching through many possible solutions: Reasoning can be reduced to performing a search. Simple exhaustive searches are rarely sufficient for most real-world problems. The solution, for many problems, is to use "heuristics" or "rules of thumb" that prioritize choices in favor of those more likely to reach a goal. A very different kind of search came to prominence in the 1990s, based on the mathematical theory of optimization. Modern machine learning is based on these methods. Instead, of using detailed explanations to guide the search, it uses a combination of[1]: (a) general architectures; (b) trying trillions of possibilities, guided by simple ideas (like gradient descent) for improvement; and (c) the ability to recognize progress (by defining a objective function).

I am interested in applying machine learning to problems in computational physics problems that traditional numerical methods can not easily handle either because of its computational costs being too high or its traditional algorithms are too complicated to easily implement.

Enrico Fermi once criticized the complexity of a model (that contains many free parameters) by quoting Johnny von Neumann "With four parameters I can fit an elephant, and with five I can make him wiggle his trunk". What Fermi implies is that it is easy to fit existing data and what is important is to have a model with predicting capability (fitting data not seen yet). The artificial neural network method tackles this difficulty by increasing the number of free parameters to millions, with the hope of obtaining predicting capability.

2 Neural network

Neural networks consists of multiple layers of interconnected nodes (neurons), each having a weight for a connection, a bias, and an activation function. Each layer build upon the previous layer. This progression of computations through the network is called forward propagation. Another process called backpropagation uses algorithms which moves backwards through the layers to efficiently compute the partial derivatives of the objective function with respect to the weights and biases. Combining the forward and backward propagation, we can calculate errors in predictions and then adjusts the weights and biases using the gradient descent method. This process is called training/learning.

2.1 Node (neuron or unit), weight, bias, and activation

As is shown in Fig. 1, we use w^l_{jk} to denote the weight for the connection from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer. Use b^l_j to denote the bias of the j^{th} neuron in the l^{th} layer.

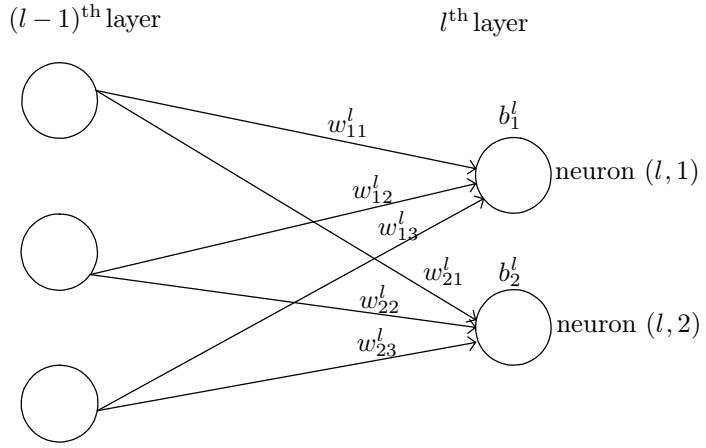


Figure 1. Definition of layers, neurons, weights, and biases in a neural network. The j^{th} neuron in the l^{th} layer is referred to as neuron (l, j)

We use a_j^l to denote the output (activation) of the j^{th} neuron in l^{th} layer. A neural network model assumes that a_j^l is related to the a^{l-1} (output of the previous layer) via

$$a_j^l = \sigma\Big(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\Big), \tag{1}$$

where the summation is over all neurons in the $(l-1)^{\text{th}}$ layer and σ is a function called activation function which can take various forms, e.g., step function,

$$\sigma(z) = \begin{cases} 1 \text{ if } z \geqslant 0 \\ 0 \quad \text{else} \end{cases}, \tag{2}$$

rectified linear unit (ReLU),

$$\sigma(z) = \max(0, z), \tag{3}$$

and sigmoid ("S"-shaped curve, also called logistic function)

$$\sigma(z) = \frac{1}{1+\exp(-z)}. \tag{4}$$

For natation ease, define z_j^l by

$$z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l, \tag{5}$$

which can be interpreted as an weighted input to the neuron (l, j), then Eq. (1) is written as

$$a_j^l = \sigma(z_j^l). \tag{6}$$

In matrix form, Eq. (5) is written as

$$z^l = w^l a^{l-1} + b^l, \tag{7}$$

where w^l is a $J \times K$ matrix, z^l and b^l are column vectors of length J, a^{l-1} is a column vector of length K, where J and K are the number of neurons in the l^{th} layer and $(l-1)^{\text{th}}$ layer, respectively.

The input layer is where data inputs are provided, and the output layer is where the final prediction is made. The input and output layers of a deep neural network are called visible layers. The layers between the input layer and output layer are called hidden layers. Note that the input layer is usually not considered as a layer of the network since it does not involve any computation. In tensorflow, layers refer to the computing layers (i.e., hidden layers and the output layer, not including the input layer). The activation function of each layer can be different. The activation function of the output layer is often chosen as None, ReLU, logistic/tanh, and is usually different from those used in the hidden layers. Here "None" means activation $\sigma(z) = z$.

2.2 Objective function

Define an objective function (can be called loss or cost function depending on contexts) by

$$C(w,b)\equiv\frac{1}{2n}\sum_x \|y(x)-a^L\|^2, \tag{8}$$

where w and b denotes the collection of all weights and biases in the network, n is the total number of training examples x, the summation is over all the training examples, $y(x)$ is the desired output from the network (i.e., correct answer) when x is the input, and a^L is the actual output from the output layer of the network and is a function of w, b, and x. Note that y and a^L are vectors (with number of elements being the number of neurons in the output layer) and $\|\ldots\|$ denotes the vector norm. Explicitly writing out the vector norm, Eq. (8) is written as

$$C(w,b)\equiv\frac{1}{2n}\sum_x \sum_{j=1}^{N_L} (y_j(x)-a_j^L)^2, \tag{9}$$

where N_L is the number of neurons in the output layer.

The cost function is the average error of the approximate solution away from the desired exact solution. The goal of a learning algorithm is to find weights and biases that minimize the cost function. A method of minimizing the cost function over (w,b) is the gradient descent method:

$$w_{jk}^l \to w_{jk}^l - \eta\frac{\partial C}{\partial w_{jk}^l}, \tag{10}$$

$$b_j^l \to b_j^l - \eta\frac{\partial C}{\partial b_j^l}, \tag{11}$$

where η is called learning rate, which should be positive.

In using the gradient descent method, we need to compute the partial derivatives $\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^l$. Next we will discuss how to compute them.

2.3 Gradients of objective function

Note that the loss function involves an average over all the training examples. Denote the loss function for a specific training example by C_x, i.e.,

$$C_x=\frac{1}{2}\sum_{j=1}^{N_L} (y_j(x)-a_j^L)^2, \tag{12}$$

then expression (9) is written as

$$C=\frac{1}{n}\sum_x C_x, \tag{13}$$

Then the partial derivatives $\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^l$ can be written as the sum of $\partial C_x/\partial w_{jk}^l$ and $\partial C_x/\partial b_j^l$, i.e.,

$$\frac{\partial C}{\partial w_{jk}^l}=\frac{1}{n}\sum_x \frac{\partial C_x}{\partial w_{jk}^l}, \tag{14}$$

$$\frac{\partial C}{\partial b_j^l}=\frac{1}{n}\sum_x \frac{\partial C_x}{\partial b_j^l}. \tag{15}$$

The above formulas indicate that, once $\partial C_x/\partial w_{jk}^l$ and $\partial C_x/\partial b_j^l$ are known, obtaining $\partial C/\partial w_{jk}^l$ and $\partial C/\partial b_j^l$ is trivial, i.e., just averaging them. Therefore, we will focus on C_x (i.e., the cost function for a fixed training example) and discuss how to compute $\partial C_x/\partial w_{jk}^l$ and $\partial C_x/\partial b_j^l$.

In practice, we do not sum over all the training examples. Instead, we average the derivative over a small number (say 16) of training examples (a mini batch) and use these approximate derivatives to advance a step. For the next step, we stochastically change to using a different mini batch. This is called stochastic gradient descent (SGD) method.

2.4 Back-propagating algorithm

The cost function C_x is a function of weights and biases of all neurons (the input x and output $y(x)$ are fixed parameters). For a specific neuron (l, j), its weights and biases enter C_x via the combination $z_j^l = \sum_k w_{jk}^l a_k^{l-1} + b_j^l$. Then it is useful to define the following partial derivative:

$$\delta_j^l \equiv \frac{\partial C_x}{\partial z_j^l}, \tag{16}$$

where the partial derivative are taken with fixed weights and biases for all neurons except neuron (l, j). Note that the a_k^{l-1} appearing in the expression of z_j^l does not depend on w_{jk}^l or b_j^l. It only depends on the weights and biases in the layers $\leqslant (l-1)$, which are all fixed when taking the derivative in expression (16). δ_j^l defined in expression (16) is often called the error of neuron (l, j).

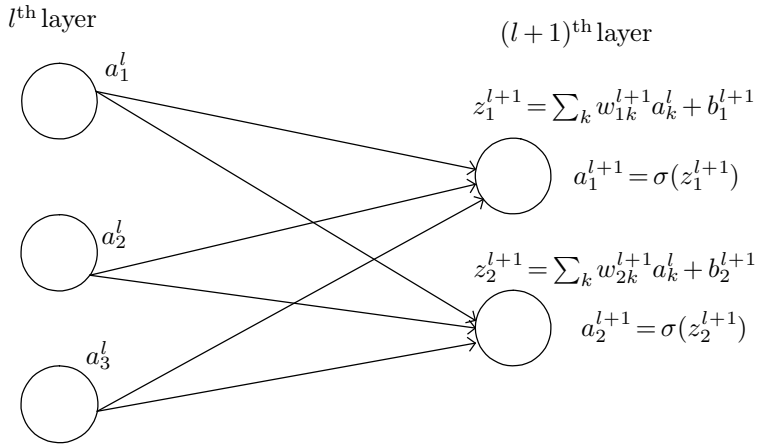
Using the chain rule, $\partial C_x / \partial w_{jk}^l$ and $\partial C_x / \partial b_j^l$ can be expressed in terms of δ_j^l:

$$\frac{\partial C_x}{\partial b_j^l} = \frac{\partial C_x}{\partial z_j^l} \frac{\partial z_j^l}{\partial b_j^l} = \delta_j^l, \tag{17}$$

and

$$\frac{\partial C_x}{\partial w_{jk}^l} = \frac{\partial C_x}{\partial z_j^l} \frac{\partial z_j^l}{\partial w_{jk}^l} = \delta_j^l a_k^{l-1}. \tag{18}$$

Therefore, if δ_j^l is known, it is trivial to compute the gradients needed in the gradient descent method.



For the output layer (L^{th} layer), δ_j^l defined in Eq. (16) is written as

$$\delta_j^L = \frac{\partial C_x}{\partial z_j^L} = \frac{\partial C_x}{\partial a_j^L} \frac{\partial a_j^L}{\partial z_j^L}. \tag{19}$$

The dependence of C_x on a_j^L is explicitly given by Eq. (12), from which the above expression for δ_j^L is written as

$$\delta_j^L = (a_j^L - y_j(x)) \sigma'(z_j^L). \tag{20}$$

Therefore δ_j^L is easy to compute.

Backpropagation is a way of computing δ_j^l for every layer using recurrence relations: the relation between δ^l and δ^{l+1}. Noting how the error is propagating through the network, we know the following identity:

$$\frac{\partial C_x}{\partial z_J^l} d z_J^l = \sum_j \frac{\partial C_x}{\partial z_j^{l+1}} d z_j^{l+1}, \tag{21}$$

with

$$dz_j^{l+1} = w_{jJ}^{l+1} d(a_J^l), \tag{22}$$

i.e.,

$$dz_j^{l+1} = w_{jJ}^{l+1} \sigma'(z_J^l) dz_J^l. \tag{23}$$

Therefore

$$\frac{\partial C_x}{\partial z_J^l} = \sum_j \frac{\partial C_x}{\partial z_j^{l+1}} w_{jJ}^{l+1} \sigma'(z_J^l). \tag{24}$$

i.e.,

$$\delta_J^l = \sum_j \delta_j^{l+1} w_{jJ}^{l+1} \sigma'(z_J^l). \tag{25}$$

Equation (25) gives the recurrence relations of computing δ^l from δ^{l+1}. This is called the back-propagation algorithm. Eq. (25) can be written in the matrix form:

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l), \tag{26}$$

where T stands for matrix transpose, $\odot$ is the element-wise product.

3 misc

4 Least square

In the least square method, the loss function is defined as

$$L = \sum_{i=1}^{n} |\hat{\mathbf{y}}(\mathbf{x}_i) - \mathbf{y}_i|^2, \tag{27}$$

where $\hat{\mathbf{y}}(\mathbf{x}_i)$ is the output of the model for the input $\mathbf{x}_i$, n is the number of data points.

In the most general case, each data point considers of multiple independent variables and multiple dependent variables $(\mathbf{x}, \mathbf{y})$. In simple cases, each data point has one independent variable and one dependent variable. For example, a data set consists of n data-points (x_i, y_i), $i = 1, ..., n$, where x_i is an independent variable and y_i is a dependent variable whose value is found by observation. The model function has the form $y(x) = f(x, \boldsymbol{\beta})$, where m adjustable parameters are held in the vector $\boldsymbol{\beta}$. A least squar model is called linear if the model comprises a linear combination of the parameters, i.e.,

$$f(x, \boldsymbol{\beta}) = \sum_{j=1}^{m} \beta_j \varphi_j(x), \tag{28}$$

where $\varphi_j(x)$ are basis functions chosen. Letting $X_{ij} = \varphi_j(x_i)$, then the model prediction for input x_i can be written as

$$f_i \equiv f(x_i, \boldsymbol{\beta}) = \sum_{j=1}^{m} X_{ij} \beta_j. \tag{29}$$

For n data points, the above can be written in matrix form:

$$\mathbf{f} = \mathbf{X}\boldsymbol{\beta}, \tag{30}$$

where $\mathbf{f} = (f_1, \dots f_n)^T$.

For linear least-square fitting, we can solve the "normal equation" to get the fitting coefficients. Alternatively, one can use iterative methods, e.g., the gradient descent method, to minimize the mean square error over the coefficients. The following is a complete example in Python:

```
import numpy as np
import matplotlib.pyplot as plt
```

```
class Linear_Regression:
    def __init__(self):
            self.b = [0, 0]

    def predict(self, X):
            Y_pred = self.b[0] + self.b[1]*X
            return Y_pred

    def update_coeffs(self, X, Y, learning_rate):
            Y_pred = self.predict(X)
            m = len(Y)
            self.b[0] = self.b[0] - (learning_rate * ((1/m) *
                                                      np.sum(Y_pred - Y)))
            self.b[1] = self.b[1] - (learning_rate * ((1/m) *
                                                      np.sum((Y_pred - Y) * X)))


regressor = Linear_Regression()
Nd=11
X = np.array([i for i in range(Nd)])
Y = np.array([2*i for i in range(Nd)]) + np.random.uniform(high=5.0,size=Nd)
fig, ax = plt.subplots()
ax.plot(X,Y, 'k.',label='data')
Y_pred = regressor.predict(X)
ax.plot(X, Y_pred, label='Initial fit line')

learning_rate = 0.01
i = 0
while i<100:
    regressor.update_coeffs(X,Y,learning_rate)
    i = i+1

Y_pred = regressor.predict(X)
ax.plot(X, Y_pred, 'b-',label='Final Fit Line')
ax.legend()
plt.show()
```

The loss function is defined by the mean square error, which is not directly used in the above code. Only the partial derivatives of the loss function is directly used.

5 Logistic regression for binary classification

Hypothesis function (the model): Denote the output of the model by $\hat{y}$, which is given by

$$\hat{y} = \sigma(z), \tag{31}$$

where $z = w \cdot x + b$ and σ is the sigmoid function given by Eq. (4). The model is nonlinear in the unknowns w and b.

The loss function is chosen as

$$L = -\frac{1}{m}\sum_{i=1}^{m} [y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)], \tag{32}$$

where y_i is the correct answer of the ith training example (y_i can take only two values, 0 or 1). The value of $\hat{y}_i$ is interpreted as the probability of y being 1.

Because the model function is nonlinear and the loss function is complicated, there is usually no closed-form solution that minimizes the loss function. Iterative methods, such as gradient descent, are needed to solve for w and b. The partial derivatives needed in the gradient desent method can be written as

$$\begin{aligned}
\frac{\partial L}{\partial w} &= -\frac{1}{m}\sum_i \left[y_i \frac{1}{\hat{y}_i}\sigma'(z)x_i - (1-y_i)\frac{1}{(1-\hat{y}_i)}\sigma'(z)x_i \right] \\
&= -\frac{1}{m}\sum_i \left\{ \left[y_i \frac{1}{\hat{y}_i} - (1-y_i)\frac{1}{(1-\hat{y}_i)} \right]\sigma'(z)x_i \right\} \\
&= -\frac{1}{m}\sum_i \left\{ \left[\frac{y_i-\hat{y}_i}{\hat{y}_i(1-\hat{y}_i)} \right]\sigma'(z)x_i \right\} \\
&= -\frac{1}{m}\sum_i \left\{ \left[\frac{y_i-\hat{y}_i}{\hat{y}_i(1-\hat{y}_i)} \right]\sigma(1-\sigma)x_i \right\} \\
&= -\frac{1}{m}\sum_i \left\{ \left[\frac{y_i-\hat{y}_i}{\hat{y}_i(1-\hat{y}_i)} \right]\hat{y}_i(1-\hat{y}_i)x_i \right\} \\
&= -\frac{1}{m}\sum_i \left[(y_i-\hat{y}_i)x_i \right] \qquad (33)
\end{aligned}$$

Using

$$\frac{d\sigma}{dz} = \frac{1}{(1+\exp(-z))^2}\exp(-z) = \sigma^2\exp(-z) = \sigma(1-\sigma) \qquad (34)$$

The above formula is simplified as

$$\begin{aligned}
\frac{\partial L}{\partial w} &= -\frac{1}{m}\sum_i \left\{ \left[\frac{y_i-\hat{y}_i}{\hat{y}_i(1-\hat{y}_i)} \right]\sigma(1-\sigma)x_i \right\} \\
&= -\frac{1}{m}\sum_i \left\{ \left[\frac{y_i-\hat{y}_i}{\hat{y}_i(1-\hat{y}_i)} \right]\hat{y}_i(1-\hat{y}_i)x_i \right\} \qquad (35) \\
&= -\frac{1}{m}\sum_i \left[(y_i-\hat{y}_i)x_i \right]
\end{aligned}$$

Similary, we obtain

$$\frac{\partial L}{\partial w} = -\frac{1}{m}\sum_i (y_i-\hat{y}_i). \qquad (36)$$

5.1 Automatic differentiation

Forward:

```
class Expression:
    def __add__(self, other):
        return Plus(self, other)
    def __mul__(self, other):
        return Multiply(self,other)
class Variable(Expression):
    def __init__(self,value):
        self.value = value
    def evaluate(self):
        return self.value
    def derive(self, v):
        return 1 if self == v else 0

class Plus(Expression):
    def __init__(self,exp1,exp2):
        self.a = exp1
```

```
        self.b = exp2
    def evaluate(self):
        return self.a.evaluate() + self.b.evaluate()
    def derive(self,exp):
        return self.a.derive(exp) + self.b.derive(exp)

class Multiply(Expression):
    def __init__(self,exp1,exp2):
        self.a = exp1
        self.b = exp2
    def evaluate(self):
        return self.a.evaluate() * self.b.evaluate()
    def derive(self,exp):
        return (self.a.derive(exp)*self.b.evaluate()
               +self.b.derive(exp)*self.a.evaluate())

# Example: derivatives of z = x * (x + y) + y * y at (x, y) = (2, 3)
x = Variable(2)
y = Variable(3)
z = x * (x + y) + y * y
print(z.evaluate()) # z, Output: 19
print(z.derive(x)) # dz/dx,  Output: 7
print(z.derive(y)) # dz/dy,  Output: 8
```

This simple example illustrates many concepts:
* Class
* Operator overloading
* Subclass, inheritance
* Polymorphism
* Recursion

The following is the backword (or reverse) method. This code is a little harder to understand than the the forward method.

```
class Expression:
    def __add__(self, other):
        return Plus(self, other)
    def __mul__(self, other):
        return Multiply(self, other)

class Variable(Expression):
    def __init__(self, value):
        self.value = value
        self.partial = 0
    def evaluate(self):
        pass
    def derive(self, seed):
        self.partial += seed

class Plus(Expression):
    def __init__(self, exp1, exp2):
        self.a = exp1
        self.b = exp2
        self.value = None
    def evaluate(self):
        self.a.evaluate()
```

```
        self.b.evaluate()
        self.value = self.a.value + self.b.value
    def derive(self, seed):
        self.a.derive(seed)
        self.b.derive(seed)

class Multiply(Expression):
    def __init__(self, exp1, exp2):
        self.a = exp1
        self.b = exp2
        self.value = None
    def evaluate(self):
        self.a.evaluate()
        self.b.evaluate()
        self.value = self.a.value * self.b.value
    def derive(self, seed):
        self.a.derive(self.b.value * seed)
        self.b.derive(self.a.value * seed)

# Example: Finding the partials of z = x * (x + y) + y * y at (x, y) = (2, 3)
x = Variable(2)
y = Variable(3)
z = x * (x + y) + y * y
z.evaluate(); print("z =", z.value)  # Output 19
z.derive(1)
print(x.partial)  # dz/dx Output:  7
print(y.partial)  # dz/dy Output:  8
```

Bibliography

[1] Michael A Nielsen. *Neural networks and deep learning*, volume 25. Determination Press: http://neuralnetworksanddeeplearning.com/, 2015.