Neural Networks

Michael

School of Mathematics and Statistics, UCD

School of Economics, University of Nottingham

1 Review of Logistic Regression

Since neural network models employ the *softmax* function, which is exactly the transformation used in the multilogit model, we will review logistic regression first before presenting a very simple neural network model.

Now, given a feature vector $x \in \mathbb{R}^m$, we can set the linear regression function[1]

$$z = w^T x + b$$

where $w \in \mathbb{R}^m$ is the coefficient vector (or weights) and b is the bias (or the constant factor). The logistic regression then is built upon the activation function $\sigma(z)$, which is usually chosen to be the *sigmoid* function:

$$\sigma(z) = 1/(1 + e^{-z})$$

In the context of binary classification in neural networks, the cross-entropy loss function $L(z, y)$ is defined as follows:

$$L(z, y) = -\left[y \log(\hat{y}) + (1 - y)\log(1 - \hat{y})\right],$$

where $\hat{y} = \sigma(z)$. The loss function is derived by maximizing the likelihood. Based on this loss function, the cost function $J(w, b)$ for the whole model, is defined as follows:

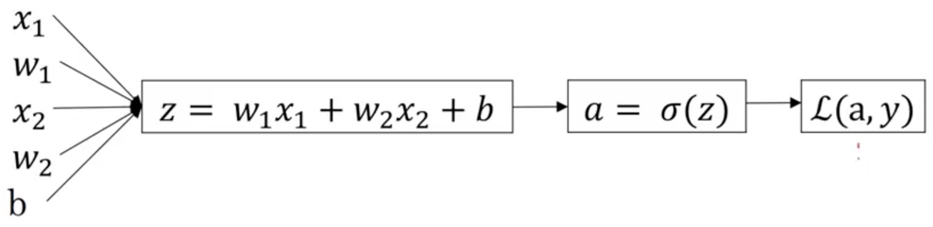
$$J(w, b) = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)\right]$$

To estimate the coefficients, we deploy the gradient descent method again, and have the following update rules:

$$\begin{aligned} w :=& w - \alpha \frac{\partial J(w)}{\partial w} \\ b :=& b - \alpha \frac{\partial J(w)}{\partial b} \end{aligned}$$

[1]By the way, you can also write $x^T w$, it's just the different notation. Also, Andrew Ng likes to use m to denote the sample size and n to denote the number of features. I prefer to using n to denote sample size.

Put them together, we can have the following flow chart, where we replace $\hat{y}$ with a.



Based on the above chart, we should have

- $\frac{\partial L}{\hat{y}} = \frac{\partial L}{\partial f(x)} = \frac{\partial L}{a} = -\frac{y}{a} + \frac{1-y}{1-a}$
- $\frac{\partial a}{\partial z} = a(1-a)$
- according to the chain rule: $dz = \frac{\partial L}{\partial z} = \frac{dL}{da}\frac{da}{dz} = a - y$

Assume we only have two features, we should have

$$\frac{\partial L}{\partial w_1} = x_1 dz; \quad \frac{\partial L}{\partial w_2} = x_2 dz$$

Algorithm 1: Logistic Regression: Gradient Descent (one iteration)

```
Result: Updated $w$ and $b$
initialization $J = 0; w = 0; dw_1 = 0; dw_2 = 0; db = 0;$
for $i = 1$ to $N$ do
    $z_i = w^T x_i + b;$
    $a_i = \sigma(z_i);$
    $J_i + = -[y_i \log a_i + (1 - y_i)\log(1 - a_i);$
    $dz_i = a_i - y_i$ ;
    for $j = 1$ to $m$ do
        $dw_i^j = x_i^j dz_i$ ;
    end
    $dw_i + = dw_i;$
    $db = +dz_i$
end
$J/ = N; dw/ = N; db/ = N;$
$w = w - \alpha\ dw;$
$b = b - \alpha\ db$
```

The reason for presenting the above algorithm is to emphasize that one *should use all information from the data matrix $N \times m$, and we need two loops without doing vectorization.*

Now, if we vectorize the dw, then we would have the following algorithm:

Algorithm 2: Logistic Regression: Gradient Descent (one iteration)

```
**Result:** Updated $w$ and $b$
initialization $J = 0; w = 0; dw = np.zeros([m, 1]); db = 0;$
**for** $i = 1$ *to* $N$ **do**
    $z_i = w^T x_i + b;$
    $a_i = \sigma(z_i);$
    $J_i+ = -[y_i \log a_i + (1 - y_i) \log(1 - a_i);$
    $dz_i = a_i - y_i$ ;
    $dw_i+ = x_i dz_i;$
    $db = +dz_i$
**end**
$J/ = N; dw/ = N; db/ = N;$
$w = w - \alpha\ dw;$
$b = b - \alpha\ db$
```

Now, let $X \in \mathbb{R}^{N\times m}$, which means that we have N rows of dataset with m features. Then, we can construct

$$Z = Xw + b$$

where w is the $m \times 1$ coefficient vector. I found it very interesting that people who are doing computer science like the following notation (especially Andrew Ng).

$$X \in \mathbb{R}^{m\times N}, \quad w \in \mathbb{R}^m$$
$$Z = w^T X + b$$

With the full vectorization, our algorithm becomes:

Algorithm 3: Logistic Regression: Gradient Descent (1000 iterations)

```
**Result:** Updated $w$ and $b$
initialization $J = 0; w = np.zeros([m, 1]); b = 0;$
**for** *i in range(1000)* **do**
    $Z = Xw + b;$
    $A = \sigma(X);$
    $dZ = A - Y$ ;
    $J = -\frac{1}{N}[Y * \log(A) + (1 - Y) * \log(1 - A)];$
    $dw = \frac{1}{N} X^T dZ;$
    $db = \frac{1}{N} np.sum(dZ, axis = 0);$
    $w_i = w_i - \alpha\ dw;$
    $b_i = b_i - \alpha\ db;$
**end**
```

Comment

In my notes `Logistic Regression`, I said: *Although we could use gradient descent algorithm to estimate coefficients. It turns out that gradient descent is not efficient. Hence, people turned to Newton-Raphson algorithm for maximising our loglikelihood function.* Hope you will not be confused.

The following code is to implement gradient descent method for linear regression, which is almost same with logistic regression one, except for the sigmoid function in logistic regression.

```
# Define gradient descent Function
def gradient_descent(x, y, theta, convergence,
                     learn_rate=0.1, iterations=100):
    '''
    Implementation of Gradient Descent Algorithm
    Input
    ------
    x : dataset of dependent variables, an n by m matrix
    y : a vector of independent variable, an n by 1 vector
    theta : parameters (or estimations), m by 1 vetor
    convergence : convergence rate measures when loop should stop
    learning rate : default value = 0.1
    interations : the maximum of interation
    Output
    -------
    theta: the convergent parameters
    cost_history : cost vector
    theta_history : trace of theta updating
    '''

    n = x.shape[0]
    m = x.shape[1]
    cost_history = np.zeros([iterations, 1])
    theta_history = np.zeros([iterations, m])
    current_theta = theta
    it = 0
    initial_converg = 10
    while initial_converg > convergence and it <= iterations:
        yhat = x @ current_theta
        theta_update = current_theta-(1/n)*learn_rate*x.transpose() @ (yhat
                                                                  -y)
        theta_history[it, :] = theta_update.transpose()
        cost_history[it, :] = square_loss(x, y, theta)
        it += 1
        initial_converg = np.max(np.abs(theta_update - current_theta))
        current_theta = theta_update

    return current_theta, cost_history, theta_history
```

2 Neural Network

2.1 Intuition and Representation

The central idea of neural network is to extract linear combinations of the inputs as derived features, and then model the target as a nonlinear function of these features. The result is a powerful learning method, with widespread applications in many fields. In this section, we will discuss the most widely used "vanilla" neural net, sometimes called the single hidden layer back-propagation network, or single layer perceptron. There has been a great deal of *hype* surrounding neural networks, making them seem magical and mysterious. As we make clear in this section, they are just nonlinear statistical models based on the logistic regression we have reviewed.

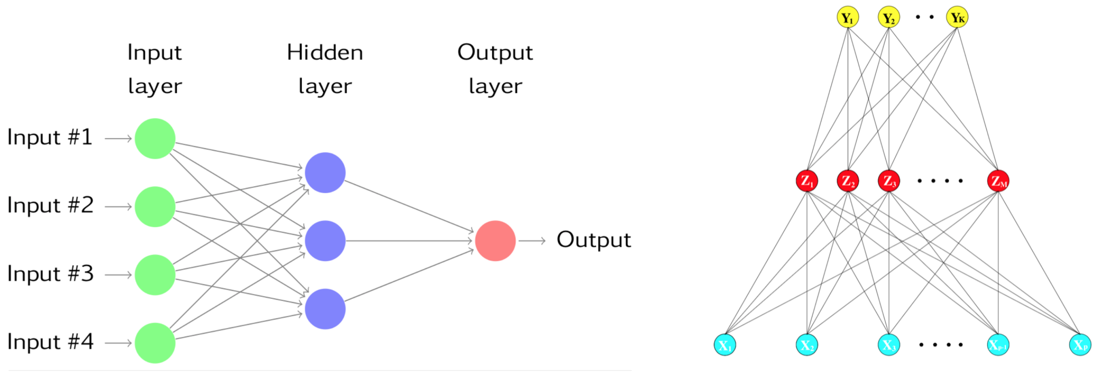


Figure 1: Visualisation of Neural Network: a single hidden layer

Again, we will use matrix to explain the mathematical model. Let $X \in \mathbb{R}^m \times \mathbb{R}^N$, which is a $m \times N$ matrix. The feature size is m and the sample size is N. Let $w \in \mathbb{R}^m$ be the coefficient vector (or weights) and b is the bias (or constant factor). Instead of constructing one logistic regression, we would construct $\mathcal{M}$ logistic regression models for the *hidden layer*.

First, we set the linear regression function:

$$w_i^T X + b_i, \quad i = 1, \cdots \mathcal{M}$$

We could still employ the sigmoid function as the activation function. However, in neural network models, we would introduce more activation functions as follows:

- Sigmoid function: $\sigma(z) = \frac{1}{1+e^{-z}}$;
- Tanh function: $\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$;
- ReLu (rectifier linear unit): $ReLU(z) = \max(0, z)$;
- Leaky ReLU: $LReLU(z) = \max(0.1z, z)$;
- and many other functions

I did not *compare and contrast* above different activation functions, but you should know their differences and have a good judgement on when you should apply which.

Choosing one activation function, suppose we use sigmoid function $\sigma(z)$, then the hidden layer should be:

$$Z_i = \sigma(w_i^T X + b_i), \quad i = 1, \cdots, \mathcal{M}$$

Once, we calculated each Z_i, we stack them vertically. Then, we should get a vector $Z \in \mathbb{R}^{\mathcal{M}} \times \mathbb{R}^N$, which is an $\mathcal{M} \times N$ matrix. If we also stack the coefficients vector and bias vertically, then we should have[2]:

- W^T is a $\mathcal{M} \times m$
- B is a $\mathcal{M} \times 1$ vector

Then the transformation for the hidden layer can be represented in the following matrix operation:

$$Z = \sigma(W^T X + B'), \quad Z \in \mathcal{M} \times N$$

where B' is a $\mathcal{M} \times N$ matrix by repeating column vector B N times.

Dimension Illustration

The reason we use the hidden layer is for reducing the dimension of features. Recall the dimension of one image in classification. Suppose we have a $(64, 64, 3)$ picture, then the vector size becomes $64 \times 64 \times 3 = 12,288$, which gives the *feature size.* Suppose the same size is 1000, in which we have 1000 images, then the dataset set dimension becomes:

- Dimension of dataset: $12,288 \times 1000$.

Normally, you wish you could reduce the feature size as $12,288$ is just too big. This is the first reason that we construct the hidden layers. Suppose we let $\mathcal{M} = 60$, then after the transformation, out dataset becomes:

- Dimension of hidden layer: 60×1000

, which is much easier to handle now.

Now, with $Z = \mathcal{M} \times N$, we deploy the logistic regression again, either for binary classification or K classes classification. Suppose, we will do K classes classification, we continue to build up the linear regression with $Z = \mathcal{M} \times N$,

$$T_k = \beta_k^T Z + \tau_k, \quad k = 1, \cdots K,$$

where β is a $\mathcal{M} \times 1$ vector and τ is the bias for the hidden layer. Sending the above linear regression into the activation function, we can obtain the final estimations:

$$Y_k = \sigma(T_k), \quad k = 1, \cdots, K$$

If we stack vectors in the above model, we should have

$$Y = \begin{cases} \sigma(T) = \sigma(\beta^T Z + \tau) & \text{if } K = 2, \text{ then } \beta \in 1 \times \mathcal{M} \\ \texttt{softmax}(T) = \max_{i \in K} \frac{\sigma(T_i)}{\sum_{i=1}^K \sigma(T_i)} & \text{if } K \geq 3, \text{ then } \beta \in K \times \mathcal{M} \end{cases}$$

where β is a $K \times \mathcal{M}$ matrix and τ is $K \times N$ vector[3], and Y is a $1 \times N$ vector that gives

[2]Be careful, we are stacking w^T here

[3]Again, repeating the column vector N times

the final estimations.

2.2 Model training:back-propagation

The neural network model has unknown parameters, often called *weights*, and we seek values for them that make the model fit the training data well. We denote the complete set of weights by θ, which consists of

- W^T and B, total number is $\mathcal{M}(m+1)$
- β^T and τ, total number is $K(\mathcal{M}+1)$

Suppose y is the labeled result and $f_k(x)$ is the estimated result. Then, for regression, we use sum-of-squared errors as our measure of fit (error function)

$$R(\theta) = \sum_{k=1} \sum_{i=1}^{N} (y_{ik} - f_k(x_i))^2$$

For classification we use either squared error or cross-entropy:

$$R(\theta) = -\sum_{i=1}^{N} \sum_{k=1}^{K} y_{ik} \log f_k(x_i)$$

and the corresponding classifier is $G(x) = \arg\max_k f_k(x)$. With the softmax function and the cross-entropy error function, the neural network model is exactly a linear logistic regression model in the hidden units, and all the parameters are estimated by maximum likelihood.

The generic approach to minimize $R(\theta)$ is by gradient descent, called *back-propagation* in this setting. Because of the compositional form of the model, the gradient can be easily derived using the chain rule for differentiation. This can be computed by a forward and backward sweep over the network, keeping track only of quantities local to each unit.

2.2.1 Binary Case

In the first section, we reviewed the logistic regression. Now, assume that we are still doing the binary classification. Then, we can have the following loss function for the last layer:

$$J(W, B, \beta, \tau) = -\frac{1}{N} L(\hat{y}, y)$$

In the neural network model, we need apply gradient descent to estimate a bunch of weights. Since the last step is just the logistic regression, then we can have the following update rule:

$$\begin{aligned} \beta :=& \beta - \alpha \frac{\partial J}{\partial \beta} \\ \tau :=& \tau - \alpha \frac{\partial J}{\partial \tau} \end{aligned}$$

To update β and τ, the algorithm is exactly same with the one we described in `Algorithm 3`.

The tricky part in neural network model is to update W and B as we need use *chain rule* to get the derivatives. Before we present the derivatives, let's organize our equations in the model first.

Starting from the original dataset X (an $m \times N$ matrix), we have[4]:

$$\begin{aligned} Z &= \sigma(W^T X + B'), \quad W \in m \times \mathcal{M}, B' \in \mathcal{M} \times 1 \\ T &= \beta^T Z + \tau, \quad Z \in \mathcal{M} \times N \\ Y &= \sigma(T) \\ L(Y, y) &= -[y \log(Y) + (1-y)\log(1-Y)] \end{aligned}$$

For binary classification, We can have

$$\begin{aligned} d\beta &= \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial T}\frac{\partial T}{\partial \beta} = dT \cdot X^T \\ dT &= Y - y \quad d\beta = dT \cdot Z^T \end{aligned}$$

To check whether the above derivation is right or not, we can check the dimension:

$$dT = Y - y \in 1 \times N, Z^T \in N \times \mathcal{M}, \quad \Rightarrow \quad \beta^T \in 1 \times \mathcal{M} \text{ or } \beta \in \mathcal{M} \times 1$$
$$d\tau = dT = Y - y$$

Now, when need do differentiation further, and obtain

$$\begin{aligned} dW &= \frac{\partial L}{\partial Y}\frac{\partial Y}{\partial T}\frac{\partial T}{\partial Z}\frac{\partial Z}{\partial W} \\ dW &= \texttt{element-wise}(\beta(Y-T) * [\sigma'(W^T X + B')])X^T \\ dB' &= \texttt{element-wise}(\beta(Y-T) * [\sigma'(W^T X + B')]) \qquad (dB' \in \mathcal{M} \times N) \end{aligned}$$

Since we use the loss function L in the above derivations, we need adjust our algorithms by the factor of sample size N as once we replace L with J, then we should have

$$\begin{aligned} dT &= Y - y \\ d\beta &= \frac{1}{N} dT Z^T \\ dB' &= \frac{1}{N}\texttt{np.sum}(dT, \texttt{axis=1}, \texttt{keepdims=True}) \\ A &= \texttt{element-wise}(\beta(Y-T) * [\sigma'(W^T X + B')]) \\ dW &= \frac{1}{N} A X^T \\ dB' &= \frac{1}{N}\texttt{np.sum}(A, \texttt{axis=1}, \texttt{keepdims=True})) \end{aligned}$$

[4]Y is the estimation, y is the value from the data.

2.3 Practical issues and coding techniques

There is one thing that I have to mention, which is the initialisation of weights. Rather than initialising the weights W be the matrix of zeros, we need initialise them the random values.

When you do programming, it is better to divide a big problem into small ones. For instance, we will build up the logistic regression model, which should include:

- initialize parameters
- estimate parameters by doing optimization
- predict based on the model