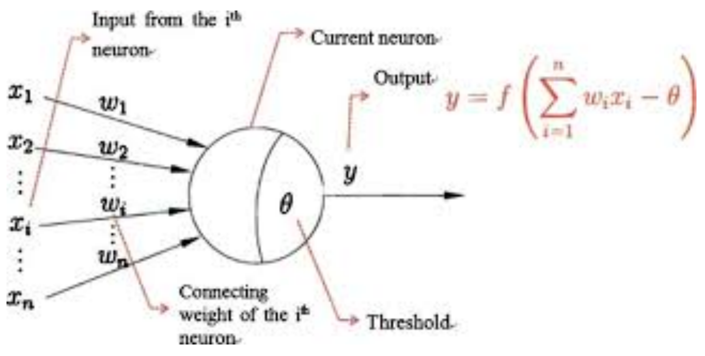
Neural Networks

A neural network is a supervised learning method. It can be applied to both regression and classification problems. **The core idea is to extract linear combinations of the inputs as derived features and then model the target as a nonlinear function of these features (feature learning)**.

1. Activation Function



Input from the i^th neuron
Current neuron
Output
$y = f\left(\sum_{i=1}^{n} w_i x_i - \theta\right)$
x_1
x_2
x_i
x_n
w_1
w_2
w_i
w_n
θ
y
Connecting weight of the i^th neuron
Threshold

The neuron used in neural networks takes linear combinations of the inputs in and if the value is greater than the threshold, the neuron would apply a nonlinear function on the combinations. The nonlinear function is called **activation function**. Activation function actually decides whether a neuron should be activated or not. Just like what our brain does. There are a lot of widely used activation functions.

- Step function

  $$sgn(x) = \begin{cases} 1 \; x \geq 0 \\ 0 \; x < 0 \end{cases}$$

  This one really does a perfect job on activation. However, it does not have continuity and smoothness that we usually want (for later back-propagation, it is very important!). Therefore, we have to come up with other functions to somehow approximate it.

- Linear function $A = cx$

  In this function, activation is proportional to input. One thing to point out is that when we are using gradient decent for training, the deriative of this activation function is always constant. And if we make all activation functions in our neural network to be linear, the whole network is just a linear model.

- Sigmoid function $A = \frac{1}{1+e^{-x}}$

  Sigmoid function is one of the most widely used activation functions today. That's because

  - Its range is between 0 and 1 which means it won't blow up the activations.
  - It could turn the model to be nonlinear.

  However, it also has a drawback that when the Y values tend to respond very less or more to changes in X, the graident at that region is going to be small. This would lead to the problem of vanishing gradient.

- Tanh function $A = \frac{2}{1+e^{-2x}} - 1 = 2sigmoid(2x) - 1$

Its bound is (-1,1) so no worries of activations blowing up. And one point to mention is that the gradient is stronger for tanh than sigmoid but of course it also has the chance of vanishing gradient.

- Relu function $A = max(0, x)$

  Relu is a nonlinear activation function which has the range of [0, inf). This means it can blow up the activation. However, it has a BIG advantage—the sparsity of the activation. For sigmoid or tanh function, all neurons would be activated in some way which brings complexity in computation but for relu function, only a fewer neurons are activated and the network would be lighter.

  Relu is very popular in CNN but it does have dying neuron problem because the gradient of $x = 0$ in relu function would be 0. It could cause learning stall.

In practise, which one to use is a parameter to be tuned in neural networks.

Ref: https://medium.com/the-theory-of-everything/understanding-activation-functions-in-neural-networks-9491262884e0

2. Back Propagation

Now it seems that we have a neural network composed of a lot of neurons but how to train the network and then get the weights? The answer is back propagation.

- Derived features Z_m are obtained by applying the *activation function* σ to linear combinations of the inputs:

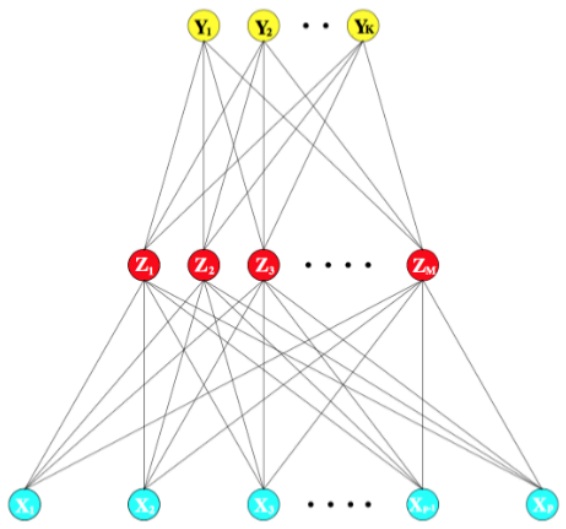
$$Z_m = \sigma(\alpha_{0m} + \alpha_m^T X), m = 1, \ldots, M.$$

- The target Y_k (or T_k in the figure) is modeled as a function of linear combinations of the Z_m:

$$T_k = \beta_{0k} + \beta_k^T Z, \quad k = 1, \ldots, K.$$

- The output function $g_k(T)$ allows a final transformation of the vector of outputs T:

$$f_k(X) = g_k(T), \quad k = 1, \ldots, K.$$



Schematic of a single hidden layer, feed-forward neural network

3 / 18

Suppose that we have a simple neural network with one hidde layer like above. α_{0m}, α_m , β_{0k} and β_k are weights. We got $M(p+1)$ weights for α set and $K(M+1)$ weights for β set.

As we usually do, if we're handling a regression problem, the loss would be square error loss $L = \Sigma_{i=1}^{n}\Sigma_{k=1}^{K}(y_{ik} - f_k(x_i))^2$ while for classification problem, the loss would be cross entropy $L = -\Sigma_{i=1}^{n}\Sigma_{k=1}^{K}y_{ik}logf_k(x_i)$. Here, we'll use squared error loss for next deriation.

Firstly, we know that

$$\begin{aligned} f_k(x_i) &= g_k(\beta_{0k} + \beta_k^T Z_i) = g_k(\beta_{0k} + \Sigma_{m=1}^{M}\beta_{km}\sigma(\alpha_{0m} + \alpha_m^T x_i)) \\ L &= \Sigma_{i=1}^{n}L_i = \Sigma_{i=1}^{n}\Sigma_{k=1}^{K}(y_{ik} - f_k(x_i))^2 \end{aligned}$$

Therefore, we could get the derivatives based on chain rule

$$\begin{aligned} \frac{\partial L_i}{\partial \beta_{km}} &= \frac{\partial L_i}{\partial f_k(x_i)}\frac{\partial f_k(x_i)}{\partial \beta_{km}} = -2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)z_{mi} \\ \frac{\partial L_i}{\partial \alpha_{ml}} &= \Sigma_{k=1}^{K}\frac{\partial L_i}{\partial f_k(x_i)}\frac{\partial f_k(x_i)}{\partial \alpha_{ml}} = -\Sigma_{k=1}^{K}2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i)x_{il} \end{aligned}$$

Then we could denote

$$\begin{aligned} \delta_{ki} &= -2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i) \\ s_{mi} &= -\Sigma_{k=1}^{K}2(y_{ik} - f_k(x_i))g_k'(\beta_k^T z_i)\beta_{km}\sigma'(\alpha_m^T x_i) \end{aligned}$$

Then

$$\begin{aligned} \frac{\partial L_i}{\partial \beta_{km}} &= \delta_{ki}z_{mi} \\ \frac{\partial L_i}{\partial \alpha_{ml}} &= s_{mi}x_{il} \end{aligned}$$

where $s_{mi} = \sigma'(\alpha_m^T x_i)\Sigma_{k=1}^{K}\beta_{km}\delta_{ki}$

And then we'll use gradient decent to iterate over the weights

$$\begin{aligned} \beta_{km}^{(r+1)} &= \beta_{km}^{(r)} - \gamma\Sigma_{i=1}^{N}\frac{\partial L_i}{\partial \beta_{km}^{(r)}} \\ \alpha_{ml}^{(r+1)} &= \alpha_{ml}^{(r)} - \gamma\Sigma_{i=1}^{N}\frac{\partial L_i}{\partial \alpha_{ml}^{(r)}} \end{aligned}$$

They are called the back-propagation equations. The updates can be implemented with a two-pass algorithm:

- forward pass: fix weights, compute the predicted values $\hat{f}_k(x_i)$
- backward pass: calculate δ_{ki} and back progagated to s_{mi}. Then use both sets of errors to compute the gradients.

Good to see: https://juejin.im/post/5c13b00d518825314143675c

3. Batch Learning of Gradient Decent

Notice that, when we are using gradient decent to get solution of weights, all samples are considered which could make the training process slow and inefficient. That's why we need a alternative algorithm called batch learning to update these weights. Stochastic gradient decent and minibatch stochastic gradient decent are two batch learning version of graident decent.

SGD

Stochastic gradient decent picks 1 sample randomly at each iteration to update the parameters. That means

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma \frac{\partial L_i}{\partial \beta_{km}^{(r)}}$$
$$\alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} - \gamma \frac{\partial L_i}{\partial \alpha_{ml}^{(r)}}$$

Compared with gradient decent, SGD would be faster and closer to the global minimal because of randomness.

Minibatch SGD

Minibatch stochastic gradient decent is actually a middle way of SGD and GD. Minibatch SGD picks a subset of sample randomly at each iteration.

$$\beta_{km}^{(r+1)} = \beta_{km}^{(r)} - \gamma \frac{1}{b} \Sigma_{i=1}^{b} \frac{\partial L_i}{\partial \beta_{km}^{(r)}}$$
$$\alpha_{ml}^{(r+1)} = \alpha_{ml}^{(r)} - \gamma \frac{1}{b} \Sigma_{i=1}^{b} \frac{\partial L_i}{\partial \alpha_{ml}^{(r)}}$$

Minibatch SGD has a lower variance than SGD and it converges faster than GD. Therefore, it is widely used in training.

4. Additionals

We've already covered some basic concepts about classic neural networks (so called dense neural network). There're some other variations in this model.

- **CNN**: Convolutional neural network is popular in image processing problems. It has two properties: local features detecting and weight-sharing.
- **RNN**: Recurrent neural network does a great job on text processing because of its 'memory'. Nowadays, some advanced neural networks, such as LSTM, GPT2, BERT are all based on RNN architecture.