Contents

1 High Level View

On its core, Machine Learning about is a program that changes from exposure to data (experience) as it is shown in figure 1.

Normally, data and parameters are the input to the model (mathematical function). This outputs a prediction, that we use to update the parameters in such a way that it better fits the data.

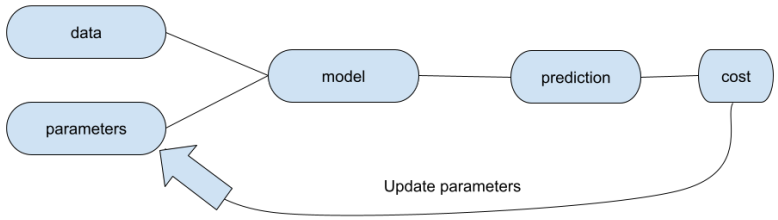


Figure 1: Machine Learning Process

Deep Learning models complex patterns of data. It's particularly useful for non-linear patterns. This opens up a new space of problems to solve. Because Deep Learning is a part of Machine Learning, the previous logic is found neural network diagrams (see Figures 2 and 7). The second image has been simplified (*update* isn't included).

2 The steps

There are two main steps: *forward* and *backward* propagation.

Suppose a mathematical function is given to us:

$$\hat{y}(x_1, x_2) = ax_1 + bx_2 + c$$

where a, b, c are *parameters*.

In **Forward Propagation** we initialize a set of parameters (say $a, b, c = 0$), and use the equation to estimate the real output (y). Both values are input to "$C(y, \hat{y})$" which is small if we're doing well, or large if bad.

So forward propagation is the calculation of $\hat{y}$ and $C(y, \hat{y})$ whatever shape they happen to have.

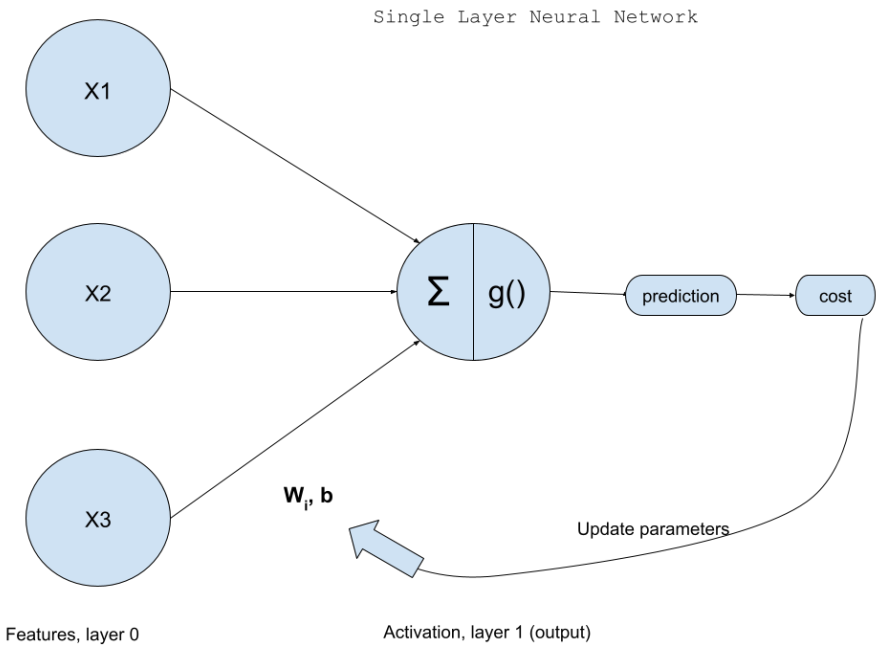


Figure 2: Single Layer Neural Network

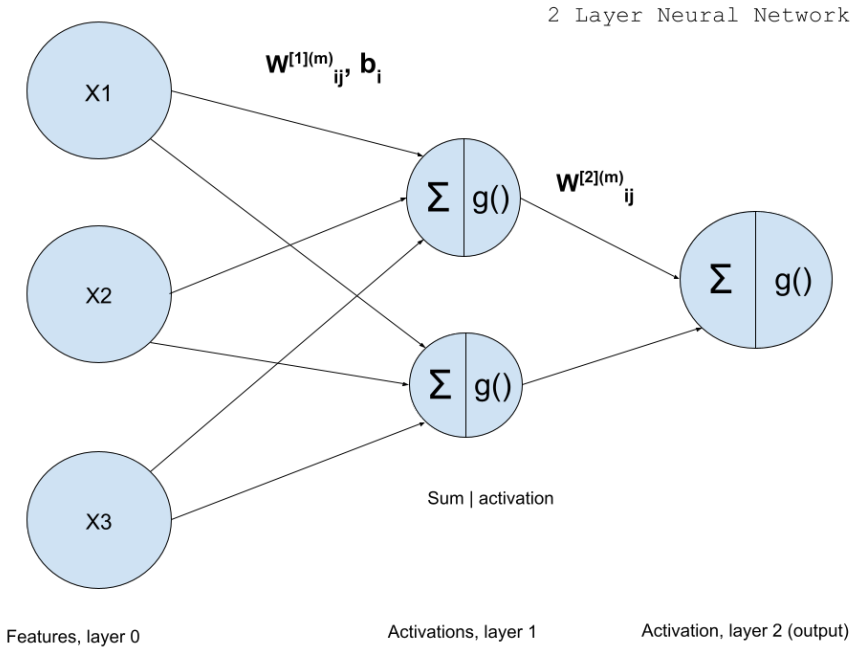


Figure 3: Shallow Neural Network

In the next chapters y, $\hat{y}$ are a and $\hat{a}$; this is just the notation used in ML/DL.

In **Backward Propagation** we minimize C, by differentiation, and find a way to move our function parameters towards the minimum of C. We use *Gradient Descent* for this task.

The simplest neural network is the single layer. Later, hidden layers are included, from 1 (shallow network), to many (deep network).

We will see examples in detail, starting with multivariable linear regression, that is, a previous step to *Deep Learning*.

3 Summary

Calculate $\hat{y}$ which is an estimation of y, compute $C(y, \hat{y})$, and use this to update $\hat{y}$ parameters, iteratively.

4 Example: Multivariable Linear Regression

A linear regression model for 2 features:

$$\begin{aligned} a(x_1, x_2) &= w_1\, x_1 + w_2\, x_2 + b \\ &= \vec{w} \cdot \vec{x} + b \end{aligned}$$

w_1,w_2 control the slopes of this plane, b translates it up and down. $\vec{x}$ is for one sample. Two or more samples are represented as a matrix:

$$[a_1, a_2] = [w_1, w_2] \cdot \begin{bmatrix} x_1 & x_1 \\ x_2 & x_2 \end{bmatrix} + b$$

a_i is the result for each sample (columns in the matrix).

The same than for a line (figure 4) by tuning W and b we can find the best linear fit. The sign of W reflects the line over Y(a) axis, and its magnitude controls the slope; b translates the line over x. Without b the line has to go through the origin.

We need to do forward and backward propagation.

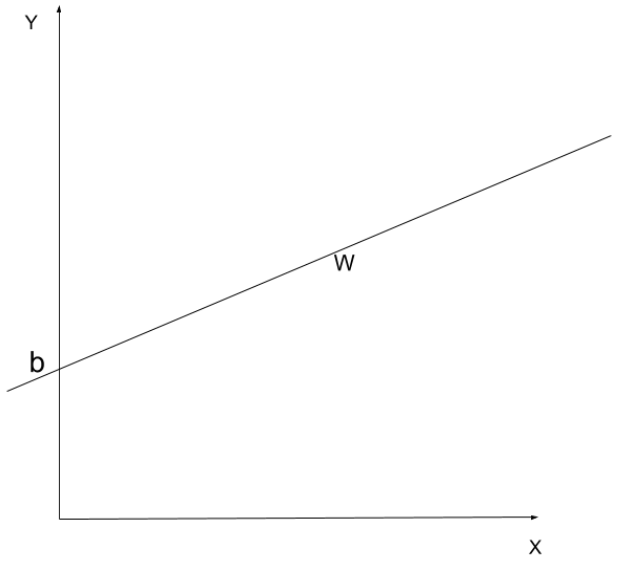


Figure 4: Line plot

4.1 Forward Propagation

The Loss denoted $= Loss(w_1, \ldots, w_n, b)$ or just $L(\vec{w}, b)$ in linear regression is the *Square Error*:

$$\begin{aligned} L_i(\vec{w}, b) &= (a_i - \hat{a})^2 \\ &= (a_i - \vec{w} \cdot \vec{x}_i - b)^2 \end{aligned}$$

$\vec{w} \cdot \vec{x}_i$ can also be denoted $\sum_j w_j \cdot \mathbf{X}_{ji}$. In the first form we multiply the vector w and the column i (dot product).

In Linear Regression, the Cost is the averaged sum of L_i, and it's the *Mean Square Error*:

$$\begin{aligned} C(w_1, w_2, b) &= \frac{1}{2} \sum_{i=0}^{i=2} L_i(w_1, w_2, b) \\ &= \frac{1}{2}([a_1, a_2] - [\hat{a}_1, \hat{a}_2]) \cdot ([a_1, a_2] - [\hat{a}_1, \hat{a}_2]) \\ &= \frac{1}{2}([a_1, a_2] - [w_1, w_2] \cdot \mathbf{X} - b) \cdot ([a_1, a_2] - [w_1, w_2] \cdot \mathbf{X} - b) \quad (1) \end{aligned}$$

$\frac{1}{2}$ is to average over examples. Equation 1 is what we implement into code.

In Python it'd look like:

```
Ap = np.dot(w,X) - b
diff = A-Ap
cost = 1/2*np.dot(diff, diff.T)
```

4.2 Backward Propagation

For a function of a single variable $C(x)$ the derivative at a point x_0 is the slope (figure 5) at x_0.

For many variables, it's a similar process: the total derivative of a function is the sum of partial derivatives.

$$dC(\vec{w}, b) = \frac{\partial C}{\partial w_1} + \frac{\partial C}{\partial w_2} + \frac{\partial C}{\partial b} \quad (2)$$

$$(3)$$

$C(w, b)$ derivative is used to find better weights and bias:

$$\begin{aligned} \vec{w} &= [w_1, w_2] - [\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}] \cdot \alpha \\ \vec{w} &= \vec{w} - \frac{\partial C}{\partial \vec{w}} \cdot \alpha \\ b &= b - \frac{\partial C}{\partial b} \cdot \alpha \end{aligned}$$

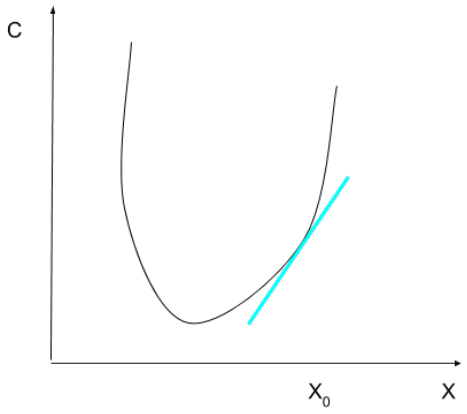


Figure 5: Derivative

Let's get started.

$$dC(\vec{w}, b) = \frac{\partial C}{\partial w_1} + \frac{\partial C}{\partial w_2} + \frac{\partial C}{\partial b} \tag{4}$$

$$= d(\sum_{i=0}^{i=2} L_i) = \sum_{i=0}^{i=2} dL_i(\vec{w}, b) \tag{5}$$

$$= \sum_{i=0}^{i=2} \frac{\partial L_i}{\partial w_1} + \frac{\partial L_i}{\partial w_2} + \frac{\partial L}{\partial b} \tag{6}$$

These equations are important. Equation 5 makes use of linearity property of diff.

We sum over columns (each sample). Now, pick up a column/sample k:

$$L_k(\vec{w}, b) = (a_k - \hat{a}_k)^2$$
$$L_k = A_k(w_1, w_2, b)^2$$
$$dL_k = 2\, A_k(w1, w2, b)\, dA_k$$

A_k is the difference $a_k - \hat{a}_k$. We need dw and db; and for this dA_k:

$$dL_k = 2A_k\,(\frac{\partial A_k}{\partial w_1} + \frac{\partial A_k}{\partial w_2} + \frac{\partial A_k}{\partial b})$$
$$A_k = A_k(w_1, w_2, b)$$
$$= a_k - \vec{w} \cdot \vec{x_k} - b$$
$$= a_k - w_1\, x_{1k} - w_2\, x_{2k} - b$$
$$dA_k = -x_{1k} - x_{2k} - 1$$

where

$$\frac{\partial A_k}{\partial w_1} = -x_{1k} \qquad \frac{\partial A_k}{\partial w_2} = -x_{2k} \qquad \frac{\partial A_k}{\partial b} = -1$$

This can be expressed in compact form:

$$\begin{aligned} dC &= sum(-\frac{1}{2} \times 2\mathbf{X} \cdot \vec{A^T} - \frac{1}{2} \times 2\vec{A} \times \vec{1}) \\ &= \frac{\partial C}{\partial \vec{w}} + \frac{\partial C}{\partial b} \end{aligned}$$

To update the parameters, *Gradient Descent* method is used. We update the vector $\vec{w}$ as follows:

$$\vec{w} = \vec{w} - \frac{\partial C}{\partial \vec{w}} \cdot \alpha \tag{7}$$

$$b = b - \frac{\partial C}{\partial b} \cdot \alpha \tag{8}$$

α is called *learning rate* and we use to tune the derivation. We go against the gradient so the sign is changed to $-$ (it points to max incresing direction otherwise).

In deep neural networks b will explicitly be a vector.

4.2.1 Interpretation

There is no graphical justification as to why we update w and b like that, but we are moving w against (minus) the gradient multiplied by a constant (alpha) called *learning rate*.

The minus sign is because the gradient always points away from the minimum and we want towards it (in one dimension there are only 2 directions).

$$\vec{w} = \vec{w} - \frac{\partial C}{\partial \vec{w}} \, \alpha \tag{9}$$

$$= \vec{w} - [\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \ldots, \frac{\partial C}{\partial w_n}] \, \alpha \tag{10}$$

$$\tag{11}$$

It means we have a vector of corrections for each slope such that the overall cost is decreased. If we take say $\frac{dC}{dw1}$ it is the sum of slopes for each sample, divided by m.

To have more insight and detail, we could take a look at a dataset with just one

feature. Then

$$\begin{aligned} C &= \frac{1}{m}\sum_i (a_i - \hat{a}_i)^2 \\ \frac{\partial C}{dw_1} &= \frac{1}{m}\sum_i \frac{dL_i}{dw_1} \\ &= -\frac{2}{m}\sum_i \mathbf{X}_{1i}(\vec{a}_i - \vec{\hat{a}}_i) \end{aligned}$$

we see the slope depends on the sum of features times errors. Much more can be inspected here.

5 Linear Regression: General Derivation

Only main differences are shown as the process is the same.

The problem is to find w_i, b such that the multidimensional "plane" has small error respect to each datapoint. Then for a new datapoint we will have a trained predictor.

In linear regression, the model is:

$$\begin{aligned} a &= w_1\, x_1 + w_2\, x_2 + \ldots + w_n\, x_n \\ &= \sum_i^n w_i\, x_i + b \\ &= \vec{w}\cdot\vec{x} + b \end{aligned}$$

Here $\vec{x}$ is for one sample. For n samples, it becomes a matrix, we write $\vec{y} = \vec{w}\cdot\mathbf{X} + \vec{b}$. This is represented:

$$[a_1, a_2, \ldots, a_n] = [w_1, w_2, \ldots, w_n] \cdot \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1m} \\ x_{21} & x_{22} & \ldots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{22} & \ldots & x_{nm} \end{bmatrix} + [b_1, b_2, \ldots, b_m]$$

The b_i are all the same number. There are m examples-columns with n features-rows. Hence $[\mathbf{X}] = m \times n$

6 Sigmoid

The derivation is similar to Linear regression. The plot for the equation:

$$\hat{a}(\vec{x}) = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x} + b)}}$$

for one dimension is on figure 6. By tuning w and b we can find the best fit.

The sign of w reflects the line over y axis, the magnitude controls the slope. Again, b translates over x.

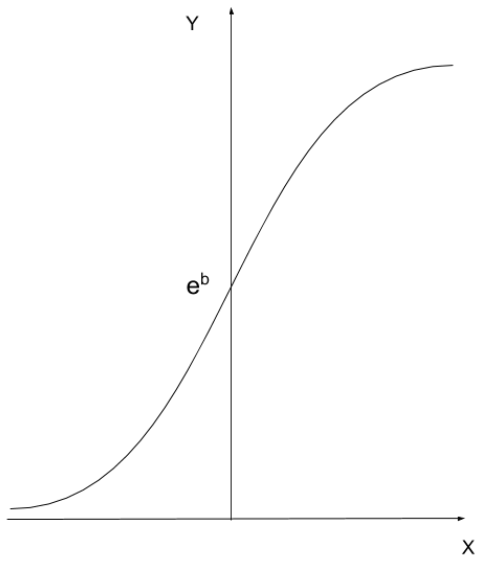


Figure 6: Sigmoid. Mistake: at $x = 0$, $y = (e^{-b} + 1)^{-1}$

6.1 Forward Propagation

- The Loss is the Cross Entropy Loss,
- The activation is *sigmoid*.

$$\hat{a}_i = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x}_i + b)}} \tag{12}$$

$$L_i = a_i \log(\hat{a}_i) + (1 - a_i) \log(1 - \hat{a}_i) \tag{13}$$

$\vec{w} \cdot \vec{x}_i$ is shorthand for $\sum_j w_j \mathbf{X}_{ji}$ (dot product for vector-matrix)

Cost

$$C(\vec{w}, \vec{b}) = -\frac{1}{m} \sum_{i=0}^{m} L_i(\vec{w}, b)$$

This is coded:

```
cost = 1/m*(np.dot(A, np.log(Ap).T)
+ np.dot(1-A, np.log(1-Ap).T))
#or
Cost = 1/m*(np.sum(np.multiply(A, np.log(Ap))
+ np.multiply(1-A, np.log(1-Ap))))
#tested
```

6.2 Backward propagation

$$\frac{\partial L_i}{\partial w_j} = \left(\frac{\partial L_i}{\partial \hat{a}_i} \frac{\partial \hat{a}_i}{\partial z} \right) \frac{\partial z}{\partial w_j} \tag{14}$$

$$\frac{\partial L_i}{\partial b} = \left(\frac{\partial L_i}{\partial \hat{a}_i} \frac{\partial \hat{a}_i}{\partial z} \right) \frac{\partial z}{\partial b} \tag{15}$$

The first two derivatives (enclosed in parens) are the same, let's compute them.

$$\begin{aligned}
\frac{\partial L_i}{\partial \hat{a}_i} &= \frac{a_i}{\hat{a}_i} + \frac{(1 - a_i)}{1 - \hat{a}_i} (-1) \\
\frac{d\hat{a}_i}{dz} &= -\left(\frac{1}{1 + e^{-z}} \right)^{-2} e^{-z}(-1) \\
&= \left(\frac{1}{1 + e^{-z}} \right)^{-2} (e^{-z} + 1 - 1) \\
&= \hat{a}^2 \, (\frac{1}{\hat{a}} - 1) \\
\frac{\partial \hat{a}_i}{\partial z} &= \hat{a}_i \, (1 - \hat{a}_i)
\end{aligned}$$

Hence the derivative is:

$$\begin{aligned}
\frac{\partial L_i}{\partial \hat{a}_i} \frac{\partial \hat{a}_i}{\partial z_i} &= \left[\frac{a_i}{\hat{a}_i} + \frac{(1 - a_i)}{1 - \hat{a}_i} (-1) \right] \hat{a}_i \, (1 - \hat{a}_i) \\
&= a_i - \hat{a}_i \\
\frac{\partial z_i}{\partial w_j} &= \mathbf{X}_{ji} \\
\frac{\partial z_i}{\partial b} &= 1
\end{aligned}$$

We are left with:

$$\frac{\partial L_i}{\partial \hat{a}_i} \frac{\partial \hat{a}_i}{\partial z_i} \frac{\partial z_i}{\partial w_j} = (a_i - \hat{a}_i)\ \mathbf{X}_{ji}$$
$$\frac{\partial L_i}{\partial \hat{a}_i} \frac{\partial \hat{a}_i}{\partial z_i} \frac{\partial z_i}{\partial b} = (a_i - \hat{a}_i)\ 1$$

We then have:

$$w_j = w_j - \frac{\partial C}{\partial w_j}\,\alpha \tag{16}$$
$$= w_j + \frac{1}{m}\sum \frac{\partial L_i}{\partial w_j}\,\alpha \tag{17}$$

But we can compute in general:

$$\vec{w} = \vec{w} - \frac{\partial C}{\partial \vec{w}}\,\alpha \tag{18}$$
$$\vec{w} = \vec{w} - \mathbf{X} \cdot \mathbf{A}^T\,\alpha \tag{19}$$
$$b = b - \frac{\partial C}{\partial b}\,\alpha \tag{20}$$

And this is the exact same result than for linear regression.

7 Results

After experimenting with single layer NNs, the following are a few conclusions.

In general, non-normalized data blows up some calculation. In sigmoid, it is the exponential $wX + b$; in linear, because of A^2 in the cost. Here also the learning rate can be really large (up to 100) and the code optimizes well. With smaller learning rates, we need more cycles.

7.1 Interesting

The backward propagation ends up being the same for both methods (apart from a constant as far as I can see). So it is useful for both methods. The cost is different though.

8 Multiclass Network

This network can be used for multiclass problems, but it's actually used with a *softmax* function instead of *sigmoid*.

Now the results and predictions are column vectors. The cost can be thought now as a row vector of N components, for N classes/labels.

Let's see the computing part:

$$\begin{bmatrix} a_{11} & a_{12} & \dots & a_{1m} \\ a_{12} & a_{22} & \dots & a_{2m} \\ \vdots & \vdots & \vdots & \vdots \\ a_{s1} & a_{s2} & \dots & a_{sm} \end{bmatrix} = \sigma(\begin{bmatrix} w_{11} & w_{12} & \dots & w_{1n} \\ w_{12} & w_{22} & \dots & w_{2n} \\ \vdots & \vdots & \vdots & \vdots \\ w_{sn} & w_{sn} & \dots & w_{sn} \end{bmatrix} \cdot \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{22} & \dots & x_{nm} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_s \end{bmatrix})$$

For $\mathbf{W}$, s dimension is equal to the number of nodes in the output layer, and n to the number of features on the input layer. $\mathbf{B}$ will actually be broadcasted to match the shape of $\mathbf{A}$.

The cost for the j node is:

$$C_j = -\frac{1}{m} \left(\sum_i \mathbf{A}_{ji} \log(\hat{\mathbf{A}}_{ji}) + (1 - \mathbf{A}_{ji}) \log(1 - \hat{\mathbf{A}}_{ji}) \right) \tag{21}$$

which may be coded like this:

```
C = np.sum(
          multiply(A, np.log(Ap))
       + multiply(1-A, np.log(1-Ap)),
       axis=0)
```

The gradients are:

$$\frac{dC}{d\mathbf{W}} = \mathbf{X} \cdot \mathbf{\Delta} \mathbf{A}^T \tag{22}$$

$$\frac{dC}{d\mathbf{B}} = \mathbf{\Delta} \mathbf{A} \tag{23}$$

$\mathbf{A}$ was a vector, now it's turned into a matrix, one row for each node. Same thing for $\mathbf{W}$.

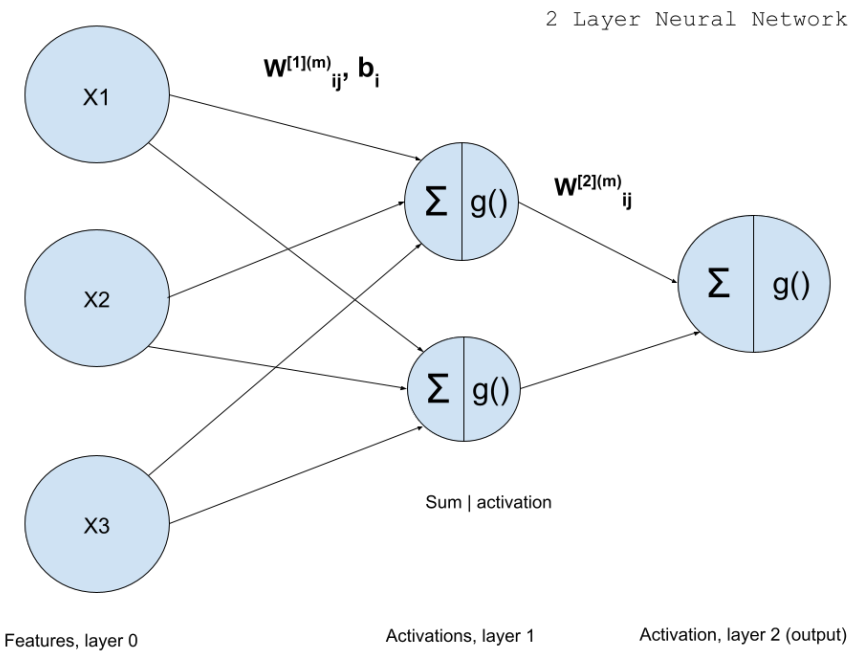


Figure 7: Shallow NN diagram

9 Shallow Neural Network

Standalone regressions approximate data with a single function. Hence they can't represent complex patterns.

We calculate what is shown in 7.

9.1 Forward Propagation

What changes is W, passes from a vector to a matrix. Each row will represent a node.

$$\mathbf{A} = \mathbf{W} \cdot \mathbf{X} + \mathbf{B} \tag{24}$$

$$\begin{bmatrix} a_1^1 & \dots & a_m^1 \\ a_1^2 & \dots & a_m^2 \\ \vdots & \dots & \vdots \\ a_1^s & \dots & a_m^s \end{bmatrix} = \begin{bmatrix} w_1^1 & w_2^1 & \dots & w_n^1 \\ w_1^2 & w_2^2 & \dots & w_n^2 \\ \vdots & \vdots & \ddots & \vdots \\ w_1^s & w_2^s & \dots & w_n^1 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} & \dots & x_{1m} \\ x_{21} & x_{22} & \dots & x_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ x_{n1} & x_{22} & \dots & x_{nm} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

A acts now like X, each column being a sample. Each row belongs to a node. And each activation in the row, is a linear combination of weights and bias from that node, and a sample from X.

This process can be done iteratively, with many layers.

- Each *row in the is associated with a single node* in the neural network.
- Each *column is instead refered to a sample* of X going through each node, so it's a *different linear combination of the features*.

To look at detailed take a hidden layer with two nodes, and two initial input features.

$$\begin{bmatrix} a_1^1 & a_2^1 \\ a_1^2 & a_2^2 \end{bmatrix} = \begin{bmatrix} w_1^1 & w_2^1 \\ w_1^2 & w_2^2 \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

They compute:

$$\begin{aligned} a_1^1 &= w_1^1\, x_{11} + w_2^1\, x_{21} + b_1 \\ a_1^2 &= w_1^2\, x_{11} + w_2^2\, x_{21} + b_2 \\ a_2^1 &= w_1^1\, x_{12} + w_2^1\, x_{22} + b_1 \\ a_2^2 &= w_1^2\, x_{12} + w_2^2\, x_{22} + b_2 \end{aligned}$$

The upper number is the neuron, lower number is the sample. If each of those is input to a linear function the result is $r_1 = c_1\, a_1^1 + c_2\, a_1^2 + b$ and $r_2 = c_1\, a_2^1 + c_2\, a_2^2 + b$ this is the same than using a single neuron. The same thing happens if we have any other linear piece like a sigmoid. Hence *linear functions aren't normally used in hidden layers*.

But the whole reasoning is useful for other hidden layers.

Luckily, each activation is input to a function, hence the whole matrix is passed through a function (easy to do in python).

9.2 Implement Forward Propagation

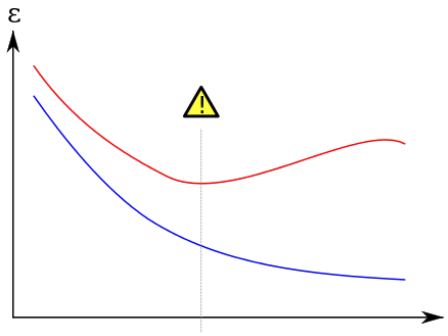
In the code, we implement equation 24 several times. The most important piece is:

```
# arch is an array describing the architecture:
arch = [4,3,2,1] #has 4 layers with those nodes.
def fp(X, arch):
    W,B = initialize(arch[0], X.shape[0]) #nodes x features
    A = predict(W,B,X,np.tanh) # 2, m
    for layer in arch[1:-1]:
        W,B = initialize(arch[layer], A.shape[0])
        A = predict(W,B,A,np.tanh) # 2, m
    W,B = initialize(arch[-1], A.shape[0]) #nodes x features
    return predict(W,B,A,sigmoid)
```

Sanity check: W, b aren't initialized as zeros, but if they were and the output is a sigmoid, the result has to be 0.69.

10 Related Concepts

Overfitting: Occurs when the cost in the training dataset decreases but it increases on the test dataset. The model starts to *memorize* data, and does not *generalize*.



Training error: blue; validation error: red, both as a function of the number of training cycles. The best predictive and fitted model would be where the validation error has its global minimum.

Underfitting: It occurs when the model or algorithm does not fit the data enough. It could be a bad model (too simple, or just not the right fit), or a lack of training, etc.

Classification v Regression: A classification model is one which attempts to predict a class, or category. That is, it's predicting from a number of discrete possibilities, such as "dog" or "cat." A regression model is one which attempts to predict one or more numeric quantities, such as a temperature or a location. Which one we use depends on the nature of the variables.

Cross Validation: The goal of cross-validation is to test the model's ability to predict new data that was not used in estimating it, in order to flag problems like overfitting or selection bias, and to give an insight on how the model will generalize to an independent dataset.

Why a CNN? It's the current state-of-the-art approach to creating computer vision models.