Contents

1 High Level View

On its core, Machine Learning about is a program that changes from exposure to data (experience) 1. Deep Learning is a subfield of machine learning, and shares this concept.

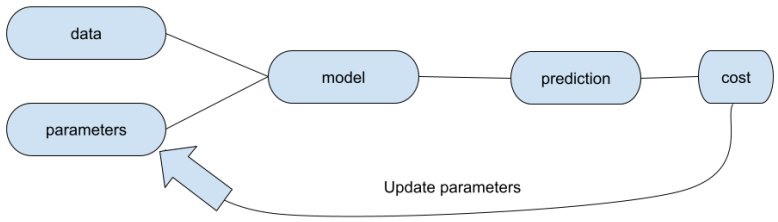


Figure 1: Machine Learning Process

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.

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

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

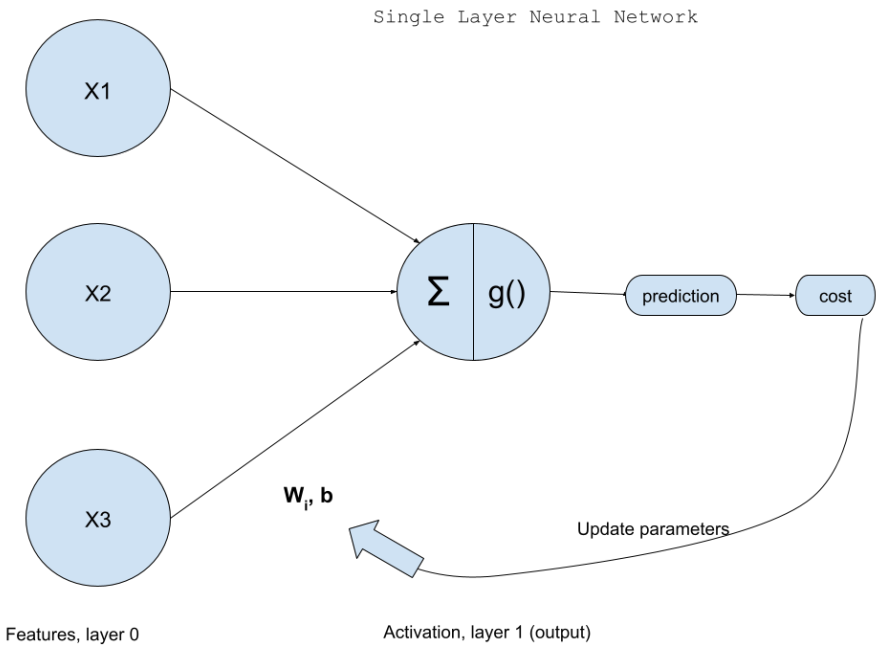


Figure 2: Single Layer Neural Network

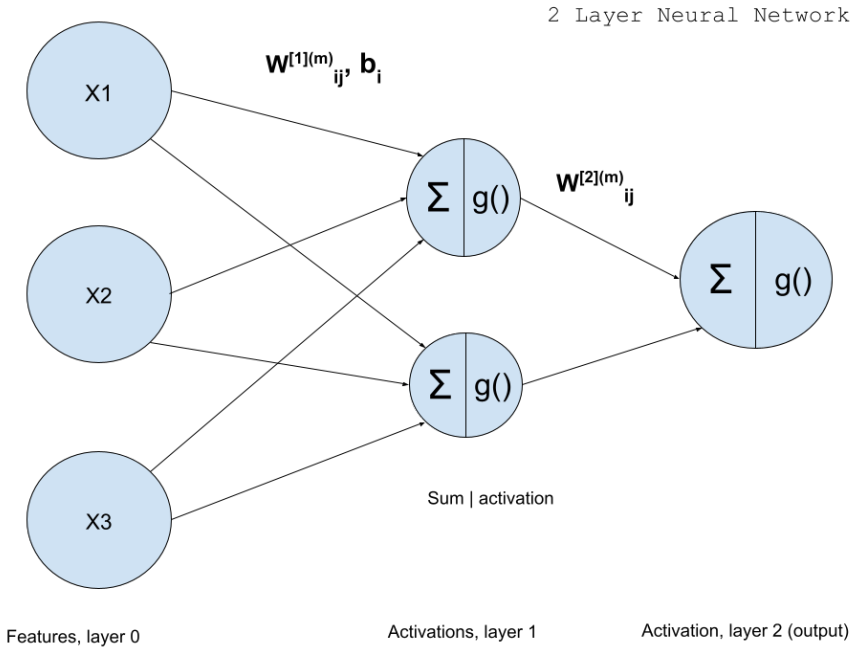


Figure 3: Shallow Neural Network

Basically, we run an estimation, compute C, and use this to update our function's parameters, iteratively.

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

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

3 Example: Multivariable Linear Regression

A linear regression model for 2 features:

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

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).

We need to do forward and backward propagation.

The same than for a line (figure 4) by tuning W and b we can find the best fit. Without b the line has to go through the origin.

The sign of W reflects the line over Y(a) axis, and the magnitude controls the slope. Also b translates over x.

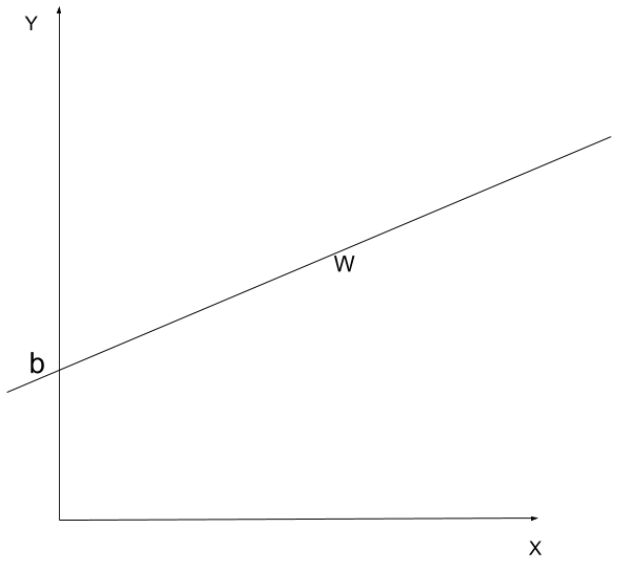


Figure 4: Line plot

3.1 Forward Propagation

The Loss $= Loss(w, b)$ 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. Outliers may be critical on the effect of weights and biases.

In Python it'd look like:

```
A' = np.dot(w,X) - b
diff = A-A'
cost = 1/2*np.dot(diff, diff.T)
```

3.2 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, this is, the ratio $\frac{\Delta C}{\Delta x}$ for very small Δx. It is the equation of a line passing over the point x_0.

Hence C can be approximated at close values of x_0, as $C(x_0 + \Delta x)$. For many variables, it's a similar process.

C has no analytical min, but we can calculate C, dC and update x_0 till we get to the minimum value of C. That's what we do here.

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

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

$$= d(\sum_{i=0}^{i=2} L_i) = \sum_{i=0}^{i=2} dL_i(\vec{w}, b) \quad (3)$$

$$= \sum_{i=0}^{i=2} \frac{\partial L_i}{\partial w_1} + \frac{\partial L_i}{\partial w_2} + \frac{\partial L}{\partial b} \quad (4)$$

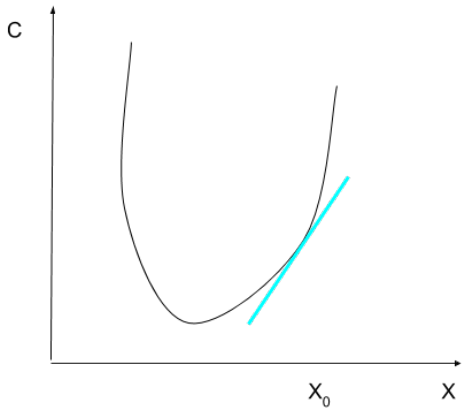


Figure 5: Derivative

These equations general and useful. Take some time to think what they do. Equation 3 makes use of linearity property of diff. We sum over columns (each sample). Now, pick up a column/sample k:

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

We named A_k to the difference $a_i - \hat{a}_i$. So dC was reduced to:

$$\begin{aligned} dC &= \frac{1}{2} \sum_{i=0}^{i=2} dL_i \\ &= \frac{1}{2}\, 2 \sum_{i=0}^{i=2} A_i(w_1, w_2, b)\, dA_i \\ &= \frac{1}{2}\, 2(\nabla_{w,b} \cdot \vec{A}) \cdot \vec{A} \end{aligned}$$

The 2 were purposedly left, as one of those will normally be a different number, which one? We don't actually need dC, but the partials. $\nabla \cdot \vec{A}$ is a fancy way to say $d\vec{A}$ but it's the correct way to write it. The parenthesis are relevant for the computational adaptation of the algorithm.

We need dw and db; and for this dA_k:

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

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:

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

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

$$\vec{w} = \vec{w} - \frac{\partial C}{\partial \vec{w}} \cdot \alpha \tag{5}$$

$$b = b - \frac{\partial C}{\partial b} \cdot \alpha \tag{6}$$

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 Linear Regression: General Derivation

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:

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

There are m examples-columns with n features-rows. Hence $[\mathbf{X}] = m \times n$

4.1 Forward Propagation

Take the k column. The Loss $= L(w, b)$ in linear regression is the *Square Error*:

$$\begin{aligned} L_k(\vec{w}, b) &= (a_k - \hat{a}_k)^2 \\ &= (a_k - \sum_{j=0}^{n} w_j \cdot \mathbf{X}_{jk} - b)^2 \end{aligned}$$

The cost in any method/model measures how well it's doing with the current parameters. In Linear Regression, it is the averaged sum of L_k, and it's the *Mean Square Error*:

$$\begin{aligned} C(\vec{w}, \vec{b}) &= \frac{1}{m} \sum_{i=0}^{m} L_i(\vec{w}, b) \\ &= \frac{1}{m} (\vec{a}_i - \vec{\hat{a}}_i) \cdot (\vec{a}_i - \vec{\hat{a}}_i) \\ &= \frac{1}{m} (\vec{a} - \vec{w} \cdot \mathbf{X} - \vec{b}) \cdot (\vec{a} - \vec{w} \cdot \mathbf{X} - \vec{b}) \end{aligned} \tag{7}$$

$C(\vec{w})$ is a way to denote C depends on all the variables in $\vec{w}$. *m* is the number of samples. C is computed from 7.

In Python it'd look like:

```
Y' = np.dot(w,X) - b
diff = Y-Y'
cost = 1/m*np.dot(diff, diff.T)
```

4.2 Backward Propagation

The cost is a "bowl", we will reach the global minimum (or close). We use the Cost derivative to find the better weight and bias.

$$
\begin{aligned}
dC(\vec{w},\vec{b}) &= \frac{dC}{d\vec{w}} + \frac{dC}{db} \\
&= \frac{1}{m} d(\sum_{0}^{m} L_i) \\
&= \frac{1}{m} \sum_{i=0}^{m} dL_i(\vec{w}, b_i) \qquad (8) \\
&= \frac{1}{m} \sum_{i=0}^{m} \frac{dL_i}{d\vec{w}} + \frac{dL_i}{db} \\
&= \frac{1}{m} \sum_{i=0}^{m} \frac{dL_i}{dw_1} + \ldots + \frac{dL_i}{dw_n} + \frac{dL}{db}
\end{aligned}
$$

Take a particular column k: Equation 8 makes use of linearity property of diff. We sum over columns (each sample). Now, pick up a column/sample k:

$$
\begin{aligned}
L_k(\vec{w}, b) &= (a_k - \hat{a_k})^2 \\
&= A_k(w_1, w_2, \ldots, w_n, b)^2 \\
dL_k &= 2\, A_k(w_1, w_2, \ldots, w_n, b) \cdot dA_k
\end{aligned}
$$

We named A_k to the difference $a_k - \hat{a}_k$.

$$
\begin{aligned}
A_k &= a_k - \sum_j w_j \cdot \mathbf{X}_{jk} - b \\
&= a_k - w_1\, x_{1k} \ldots - w_n\, x_{nk} - b
\end{aligned}
$$

$$
\begin{aligned}
dA_k &= \frac{\partial A}{\partial w_1} + \ldots + \frac{\partial A}{\partial w_n} + \frac{\partial A}{\partial b} \\
&= -x_{1k} - x_{2k} \ldots - x_{nk} - 1 \qquad (9)
\end{aligned}
$$

From equation 9, it follows:

$$
\frac{\partial A_k}{\partial w_1} = -x_{1k} \quad \ldots \quad \frac{\partial A_k}{\partial w_n} = -x_{nk} \qquad \frac{\partial A_k}{\partial b} = -1
$$

This can be expressed in compact form:

$$
\begin{aligned}
dC &= sum(-\frac{1}{m}\, 2\, \mathbf{A} \cdot \mathbf{X^T} - \frac{1}{m}\, 2\mathbf{A}\, 1) \qquad (10) \\
&= \frac{\partial C}{\partial \vec{w}} + \frac{\partial C}{\partial b}
\end{aligned}
$$

We don't need the sum. We also found each partial derivative.

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

$$\vec{w} = \vec{w} - \frac{\partial C}{\partial \vec{w}}\,\alpha \tag{11}$$

$$b = b - \frac{\partial C}{\partial b}\,\alpha \tag{12}$$

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).

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).

The meaning of equation 11 is important.

$$\vec{w} = \vec{w} - \frac{\partial C}{\partial \vec{w}}\,\alpha \tag{13}$$

$$= \vec{w} - [\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \ldots, \frac{\partial C}{\partial w_n}]\,\alpha \tag{14}$$

$$\tag{15}$$

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

$$C = \frac{1}{m}\sum_i (a_i - \hat{a}_i)^2 \tag{16}$$

$$\frac{\partial C}{dw_1} = \frac{1}{m}\sum_i \frac{dL_i}{dw_1} \tag{17}$$

$$= \frac{1}{m}\sum_i 2(a_i - \hat{a}_i)_i\,(-x_i) \tag{18}$$

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

This is almost the same result we get in the *sigmoid*, just multiplied by 2. Because the 2 can be thought as inside the learning rate, we can use the exact same backpropagation for both methods!

As it will be described, the forward propagation is different (but very simple).

5 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.

From now, the output is named a and $\hat{a}$.

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

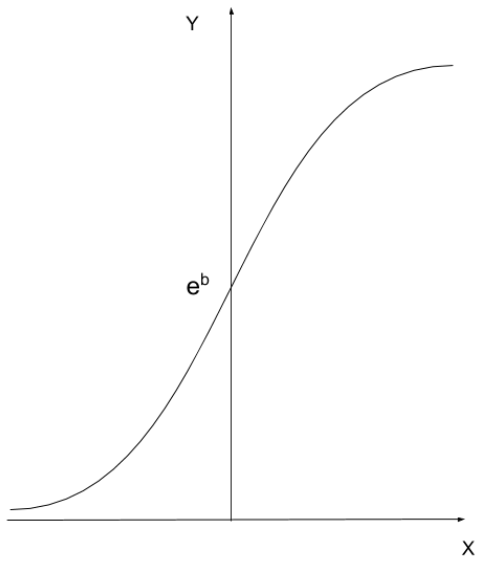


Figure 6: Sigmoid plot

5.1 Forward Propagation

This step is similar, but with new Loss and $\hat{a}$. The Loss is the Cross Entropy Loss.

$$\hat{a}_i = \frac{1}{1 + e^{-(\vec{w} \cdot \vec{x}_i + b)}} \tag{19}$$

$$L_i = a_i \log(\hat{a}_i) + (1 - a_i) \log(1 - \hat{a}_i) \tag{20}$$

Again $\vec{w} \cdot \vec{x}_i$ is shorthand for $\sum_j w_j \mathbf{X}_{ji}$

We calculate cost again:

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

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, yields same number :-)
```

5.2 Backward propagation

$$\frac{\partial L_i}{\partial \vec{w}} = \left(\frac{\partial L_i}{\partial \hat{a}_i} \frac{\partial \hat{a}_i}{\partial z} \right) \frac{\partial z}{\partial \vec{w}} \tag{21}$$

$$\frac{\partial L_i}{\partial b} = \left(\frac{\partial L_i}{\partial \hat{a}_i} \frac{\partial \hat{a}_i}{\partial z} \right) \frac{\partial z}{\partial b} \tag{22}$$

The first two derivatives (enclosed in parens) are the same, let's compute them right away. There is no need for textual explanation. i

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

Hence the derivative is:

$$\begin{aligned}\frac{\partial L_i}{\partial \hat{a}_i}\frac{\partial \hat{a}}{\partial z} &= \left[\frac{a_i}{\hat{a}_i} + \frac{(1-a_i)}{1-\hat{a}}(-1)\right]\hat{a}_i(1-\hat{a}_i)\\ &= a_i - \hat{a}_i\\ \frac{\partial z_i}{\partial w} &= x_i\\ \frac{\partial z_i}{\partial b} &= 1\end{aligned}$$

We are left with:

$$\begin{aligned}\frac{\partial L_i}{\partial \hat{a}_i}\frac{\partial \hat{a}}{\partial z}\frac{\partial \hat{z}}{\partial \vec{w}} &= (a_i - \hat{a}_i)\,\vec{x}_i\\ \frac{\partial L_i}{\partial \hat{a}_i}\frac{\partial \hat{a}}{\partial z}\frac{\partial \hat{z}}{\partial b} &= (a_i - \hat{a}_i)\,1\end{aligned}$$

$\vec{x}_i$ are all the features of sample i, not just one. In code we need an update-all vector, this is implemented: $\frac{1}{m}\,np.dot(A - Ap, X)$

So part of the derivative turns out to be simple to calculate. Using dw, db we can update values:

$$w_j = w_j - \frac{\partial C}{\partial w_j}\alpha \tag{23}$$

$$= w_j + \frac{1}{m}\sum\frac{\partial L_i}{\partial w_j}\alpha \tag{24}$$

$$\vec{w} = \vec{w} - \frac{\partial C}{\partial \vec{w}}\alpha \tag{25}$$

$$b = b - \frac{\partial C}{\partial b}\alpha \tag{26}$$

When we compute the process involves 24, and this is the exact same result than for linear regression.

6 Results

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

In general, non-normalized data explodes some calculation. In sigmoid, it explodes the exponential $wX + b$; in linear, because of A^2 in the cost.

In sigmoid example, the learning rate can be really big like 100 and the code optimizes well (but it's an easy dataset). With smaller learning rates, we need more cycles.

6.1 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.

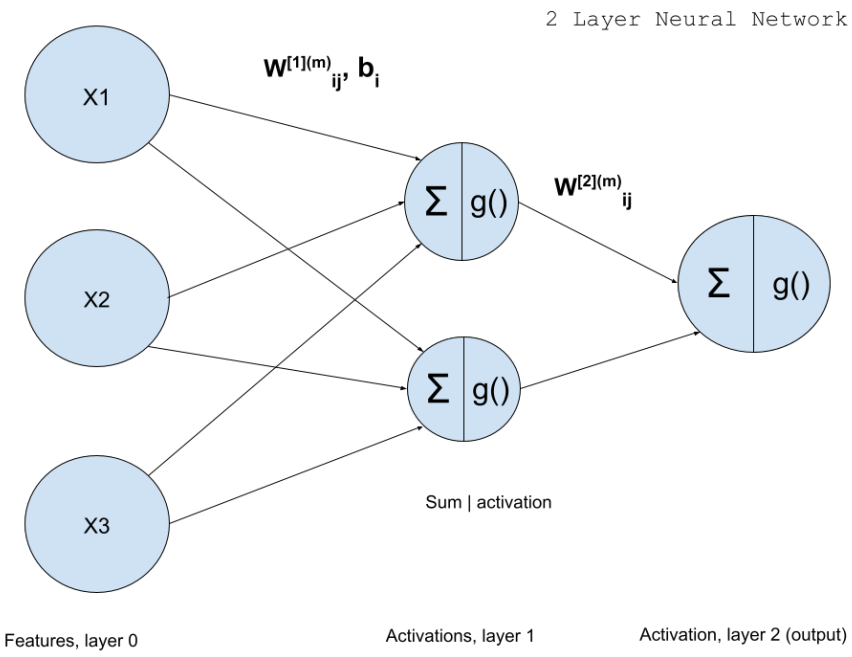


Figure 7: Shallow NN diagram

7 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.

7.1 Forward Propagation

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

$$\mathbf{A} = \mathbf{W} \cdot \mathbf{X} + \mathbf{B} \tag{27}$$

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

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).

7.2 Implement Forward Propagation

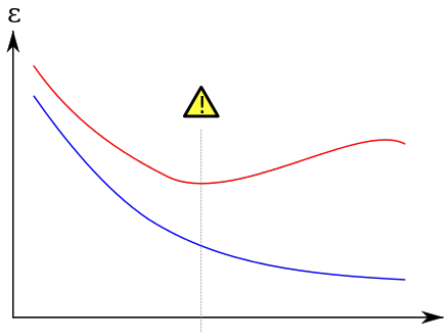
In the code, we implement equation 27 several times. The most important piece is:

```
# 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.

8 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.