Contents

Chapter 1

Introduction to neural networks

1.1 High Level View

On its core, Machine Learning about is a program that changes from exposure to data (experience).

Data and parameters are the input to a mathematical function. This outputs a prediction, that we later use to update the parameters. The process is iterative.

Deep Learning models complex patterns of data. It's particularly useful for non-linear patterns, and opens up a new space of problems to solve. Because Deep Learning is a part of Machine Learning, the previous logic is found deep network diagrams.

Examples of Network Diagrams can be found in section 5.1.

1.1.1 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.

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*.

1.1.2 Summary

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

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

1.2 Multivariable Linear Regression

A linear regression model for 2 features:

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

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 1.1) 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.

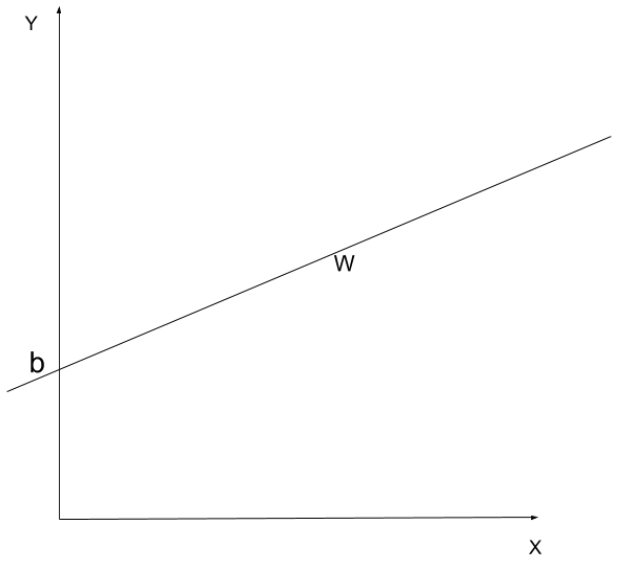


Figure 1.1: Line plot

1.2.1 Forward Propagation

A general network scheme is shown in 5.2. Calculations are built following that network scheme.

- The activation is a linear function:

$$\hat{\vec{a}} = \vec{w} \cdot \mathbf{X} + \vec{b}$$

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

$$L_i(\vec{w}, b) = (a_i - \hat{a})^2 = (a_i - \sum_j w_j \mathbf{X}_{ji} - b)^2$$

  $\sum_j w_j \, \mathbf{X}_{ji}$ is the dot product.

- The cost is the averaged sum of L_i, and it's the *Mean Square Error*:

$$\begin{aligned} C(w_1, w_2, b) &= \frac{1}{2} \sum_{i=0}^{i=2} L_i(w_1, w_2, b) \\ &= \frac{1}{2}([a_1, a_2] - [\hat{a}_1, \hat{a}_2]) \cdot ([a_1, a_2] - [\hat{a}_1, \hat{a}_2]) \\ &= \frac{1}{2}([a_1, a_2] - [w_1, w_2] \cdot \mathbf{X} - b) \cdot ([a_1, a_2] - [w_1, w_2] \cdot \mathbf{X} - b) \end{aligned} \tag{1.1}$$

  $\frac{1}{2}$ is to average over samples. Equation 1.1 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)
```

1.2.2 Backward Propagation

For a function of a single variable $C(x)$ the derivative at a point x_0 is the slope (figure 1.2) at x_0.

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

$$dC(\vec{w}, b) = \frac{\partial C}{\partial w_1} + \frac{\partial C}{\partial w_2} + \frac{\partial C}{\partial b} \tag{1.2}$$

$$\tag{1.3}$$

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

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

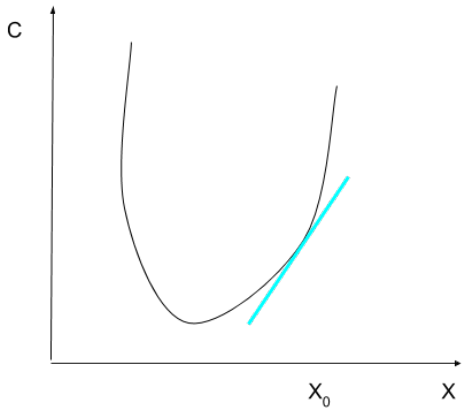


Figure 1.2: Derivative

Let's get started.

$$dC(\vec{w}, b) = \frac{\partial C}{\partial w_1} + \frac{\partial C}{\partial w_2} + \frac{\partial C}{\partial b} \tag{1.4}$$

$$= d(\sum_{i=0}^{i=2} L_i) = \sum_{i=0}^{i=2} dL_i(\vec{w}, b) \tag{1.5}$$

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

These equations are important. Equation 1.4 is always the starting point in any method. Equation 1.5 makes use of linearity property of diff.

What we look for is:

$$\frac{\partial C}{\partial w_1} = c \sum_i \frac{\partial L_i}{\partial w_1} = c \sum_i \frac{\partial L_i}{\partial A_i} \frac{A_i}{\partial w_1}$$

$$\vdots$$

$$\frac{\partial C}{\partial w_n} = c \sum_i \frac{\partial L_i}{\partial w_n} = c \sum_i \frac{\partial L_i}{\partial A_i} \frac{A_i}{\partial w_n} \tag{1.6}$$

(Remember, updating w, b, minimizing C, fits line to data. That's why we are deriving). Last equality uses the chain rule $df(g(x)) = \frac{df}{dg}\frac{dg}{dx}$.

$$L_k(\vec{w}, b) = (a_k - \hat{a}_k)^2$$

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

$$\frac{\partial L_k}{\partial w_j} = 2A_k \frac{\partial(a_k - w_j \mathbf{X}_{jk} - b)}{\partial w_j}$$
$$\sum_k \frac{\partial L_k}{\partial w_j} = -2 \sum_k A_k \mathbf{X}_{jk}$$

Replacing results in equations 1.6. For example:

$$\frac{\partial C}{\partial w_1} = \frac{1}{2} 2 \sum_i \mathbf{X}_{1i} (a_i - \hat{a}_i)$$
$$= \frac{1}{2} 2 \sum_i \mathbf{X}_{1i} A_i$$

Finally, using *Gradient Descent* method, we update vectors:

$$\vec{w} = \vec{w} - \frac{\partial C}{\partial \vec{w}} \alpha$$
$$= \vec{w} + \frac{1}{2} \times 2\mathbf{X} \cdot \vec{A^T} \alpha \tag{1.7}$$
$$b = b - \frac{\partial C}{\partial b} \cdot \alpha$$
$$= b + \frac{1}{2} \times 2\vec{A} \times \vec{1} \alpha \tag{1.8}$$

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 increasing direction otherwise).

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).

$$\vec{w} = \vec{w} - \frac{\partial C}{\partial \vec{w}} \alpha \tag{1.9}$$
$$= \vec{w} - \left[\frac{\partial C}{\partial w_1}, \frac{\partial C}{\partial w_2}, \ldots, \frac{\partial C}{\partial w_n} \right] \alpha \tag{1.10}$$

It means we have a vector of corrections for each slope such that the overall cost is decreased. If we take say $\frac{dC}{dw_1}$ it is the sum of slopes for each sample, divided by m.

The slope depends on the sum of features times errors, as it was found:

$$\frac{\partial C}{dw_1} = -\frac{2}{m}\sum_i \mathbf{X}_{1i}(\vec{a}_i - \vec{\hat{a}}_i)$$

1.2.3 Many features and samples

For a larger problem, the only difference is the size of the matrices.

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.

As it was shown, for linear regression the model is:

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

Here $\vec{x}$ is for one sample. For n samples, it becomes a matrix, we write $\vec{\hat{a}} = \vec{w}\cdot\mathbf{X} + \vec{b}$. 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 $[\mathbf{X}] = m \times n$

The results are exactly the same than the previous section.

1.3 Mulvariate Binary Regression

Also known as *logistic regression*. The activation is:

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

plot for 1 feature on figure 1.3. 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.

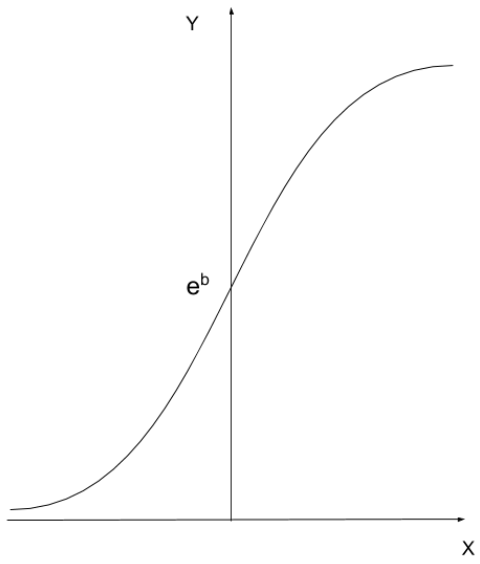


Figure 1.3: Sigmoid. Mistake: at $x = 0$, $y = (e^{-b} + 1)^{-1}$

1.3.1 Forward Propagation

The network schema can be found in 5.2. It is used to build the calculations:

- The activation is a *sigmoid* function:

$$\hat{a}_i = \frac{1}{1 + e^{-(\sum_j w_j \mathbf{X}_{ji} + b)}}$$

- The Loss is the Cross Entropy Loss:

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

- The cost:

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

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
```

1.3.2 Backward propagation

$$\frac{\partial L_i}{\partial w_j} = \left(\frac{\partial L_i}{\partial \hat{a}_i} \frac{\partial \hat{a}_i}{\partial z_i}\right) \frac{\partial z_i}{\partial w_j} \tag{1.11}$$

$$\frac{\partial L_i}{\partial b} = \left(\frac{\partial L_i}{\partial \hat{a}_i} \frac{\partial \hat{a}_i}{\partial z_i}\right) \frac{\partial z_i}{\partial b} \tag{1.12}$$

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

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

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:

$$\begin{aligned} w_j &= w_j - \alpha\frac{\partial C}{\partial w_j} \\ &= w_j + \frac{\alpha}{m}\sum\frac{\partial L_i}{\partial w_j} \\ &= w_j + \frac{\alpha}{m}\sum_i (a_i - \hat{a}_i)\mathbf{X}_{ji} \end{aligned}$$

But we can compute in general:

$$\begin{aligned} \vec{w} &= \vec{w} - \frac{\alpha}{m}\frac{\partial C}{\partial \vec{w}} \\ &= \vec{w} + \frac{\alpha}{m}\mathbf{X}\cdot\mathbf{\Delta}\mathbf{A}^T \qquad (1.13) \\ b &= b - \frac{\alpha}{m}\frac{\partial C}{\partial b} \\ &= b + \frac{\alpha}{m}\mathbf{\Delta}\mathbf{A} \qquad (1.14) \end{aligned}$$

The same result than for linear regression.

1.4 Results

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.

The backward propagation ends up being the same for both methods (apart from a constant). Hence it is useful for both methods. The cost is different though.

Chapter 2

More complex networks

2.1 Multiclass Network

This network can be used for multinomial problems (classification).

The prediction for a sample a is a column vector of s nodes. The cost is a row vector of n components, for n different labels.

From the NN diagram, we derive the computations:

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

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:

$$C_j = -\frac{1}{m} \left(\sum_i \mathbf{A}_{ji} \log(\hat{\mathbf{A}}_{ji}) + (1 - \mathbf{A}_{ji}) \log(1 - \hat{\mathbf{A}}_{ji}) \right) \tag{2.1}$$

coded:

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

The gradients are:

$$\frac{dC}{d\mathbf{W}} = -\frac{1}{m} \mathbf{\Delta A} \cdot \mathbf{X}^T \tag{2.2}$$

$$\frac{dC}{d\mathbf{B}} = -\frac{1}{m} \mathbf{\Delta A} \tag{2.3}$$

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

2.2 Shallow Neural Network

A shallow network adds a hidden layer. This helps to model more complex data patterns respect to standalone linear or logistic regression. The next calculations follow the network on figure 5.3.

2.2.1 Forward Propagation

- activation tanh hidden layer,
- activation sigmoid output layer,
- cost is a number given by log-loss

$\mathbf{W}$, passes from a vector to a matrix. Each row will represent a node.

$$\mathbf{A}^{[i]} = \mathbf{W}^{[i]} \cdot \mathbf{A}^{[i-1]} + \mathbf{B}^{[i]} \tag{2.4}$$

$$\begin{bmatrix} a_{11}^{[i]} & \dots & a_{1m}^{[i]} \\ \vdots & \ddots & \vdots \\ a_{s1}^{[i]} & \dots & a_{sm}^{[i]} \end{bmatrix} = \begin{bmatrix} w_{11}^{[i]} & \dots & w_{1n}^{[i]} \\ \vdots & \vdots & \ddots \\ w_{s1}^{[i]} & \dots & w_{sn}^{[i]} \end{bmatrix} \begin{bmatrix} a_{11}^{[i-1]} & \dots & a_{1m}^{[i-1]} \\ \vdots & \ddots & \vdots \\ a_{n1}^{[i-1]} & \dots & a_{nm}^{[i-1]} \end{bmatrix} + \begin{bmatrix} b_1^{[i]} \\ \vdots \\ b_s^{[i]} \end{bmatrix}$$

For any $\mathbf{A}$, each column correlates to a sample. Each row is always a node, for $\mathbf{A}^{[\mathbf{i-1}]}$ this is the number of features.

And each output activation in the row, is a linear combination of weights and bias from that node.

Why non-linear

The activation in the hidden layer can not be linear. Take a hidden layer with two nodes, and two initial input features.

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

First row in the output matrix is for node 1. First term is for first sample, second for second sample. Second row are computations for node 2.

So the first column is:

$$\begin{aligned} a_{11} &= w_{11}\, x_{11} + w_{12}\, x_{21} + b_1 \\ a_{21} &= w_{21}\, x_{11} + w_{22}\, x_{21} + b_2 \end{aligned}$$

When is input to another linear function the result is

$$\begin{aligned} r_1 &= c_1\, a_{11} + c_2\, a_{21} + b_1 \\ &= c\, x_{12} + c'\, x_{21} + b_1 \end{aligned}$$

This is the same than using a single linear neuron! The same thing happens if we a sigmoid. Hence *linear functions aren't normally used in hidden layers.*

2.2.2 Implement Forward Propagation

In the code, we implement equation 2.4 several times. The most important piece is:

```
def fp(A0,m,nodes=[4,3,2,1],hfn=np.tanh,ofn=sigmoid):
    """
    A0, matrix of samples
    nodes: architecture, list of nodes per layer
    hfn: hidden layer function
    ofn: output layer function
    returns row vector/matrix of predictions
    """
    W,B = initialize(nodes[0], A0.shape[0])
    Ap = predict(W,B,A0,m,hfn) # nodesL x samples
    for l in range(len(nodes[1:-1])):
        W,B = initialize(nodes[l+1], nodes[l])
        Ap = predict(W,B,Ap,m,hfn) # nodes x samples
    W,B = initialize(nodes[-1], Ap.shape[0]) #nodes x nodes_prev
    return predict(W,B,Ap,m,ofn)
```

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 (tested on scripts).

2.2.3 Backward propagation

The quantities to compute are:

- C, dC
- dL
- w', b'

Chapter 3

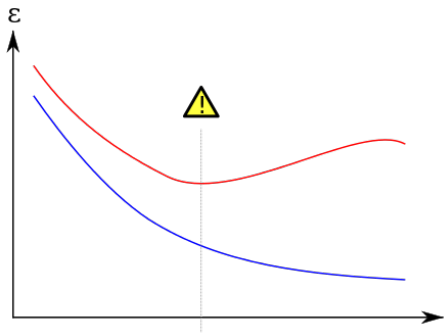
Deep Networks

Chapter 4

Concepts

4.1 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.

4.2 Math: Basic intuitions

Sometimes we forget mathematics is pure representation. We could wake up and come up with a symbolic system to represent things or processes. Languages, chemistry, mathematics, all belong to this symbolic realm of *representation*.

Matrices can be thought as standalone entities, with rules and operations that are defined in some way. When representing physics, the rules have to be coherent with reality, and that's how we test a matrix is correct.

As a side note, it's a curiosity for us how maths can be so well suited to represent phenomena in the world in such a condensed way. There is an essay from a Nobel Prize winner The Unreasonable Effectiveness of Mathematics in the Natural Sciences.

Another way to see matrices - apart from a rectangle of elements - is as an extension of equations. Indeed, they come up as a handy notation of sets of equations (no need if there is just one equation). This seems a good starting point. Summation, multiplication, derivation and operations defined for equations are supposed to exist, or have some analogous.

4.2.1 A concrete example

We are told that the relation mass of flour (kilograms) to volume (in litres) of water for a particular recipe is 2 plus 3. Hence $F = 2\,W + 3$

A thousand recipes containing Flour and Water in different proportions is needed for a dinner. There are 1000 equations similar to the one above $F = k\,W + b$. k, c being constants.

This can be represented as a matrix:

$$\begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_n \end{bmatrix} = \begin{bmatrix} k_1 & 0 & \dots & 0_n \\ 0 & k_2 & \dots & 0_n \\ \vdots & & \dots & \vdots \\ 0 & 0 & 0 & k_n \end{bmatrix} \begin{bmatrix} W_1 \\ W_2 \\ \vdots \\ W_n \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix}$$

What would mean to find $d\mathbf{F}$? It has to be defined. It is the derivative of each equation. The equation i^{th} is $f_i(w_i) = w_i\,k_i + b_i$, then $df_i = \frac{df_i}{dw_i} = k_i$. In this case, we are tempted to write: the result $d\mathbf{F} = [k_1, k_2, \dots, k_n]$. But actually, the best notation is the matrix of coefficients written above. Hence $d\mathbf{F} = \mathbf{k}$.

What would happen if each f_i is in a exponent, for example? $f_i(w_i) = e^{w_i\,k_i + b_i}$. How are the 1000 equations represented? Just $e^{\mathbf{F}}$ would indicate that. What is the derivative?

$$\frac{df}{dw} = e^{w_i\,k_i + b_i}\,k_i$$

. Then

$$\frac{d\mathbf{F}}{dW} = \mathbf{k}\, e^{\mathbf{F}}$$

In this way many derivatives can be found. For example $d\mathbf{x}^T\mathbf{x} = 2\mathbf{x}$. Because $\mathbf{x}^T\mathbf{x}$ is a representation of a set of equations of this kind: x^2. A list of useful derivatives.

Chapter 5

Figures

5.1 Figures

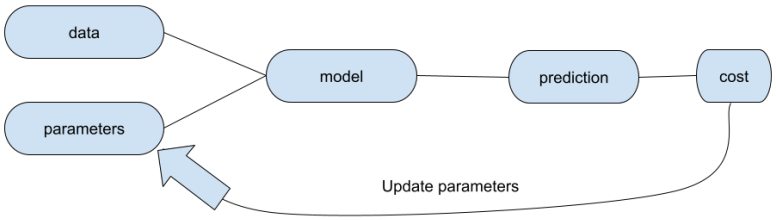


Figure 5.1: Machine Learning Process

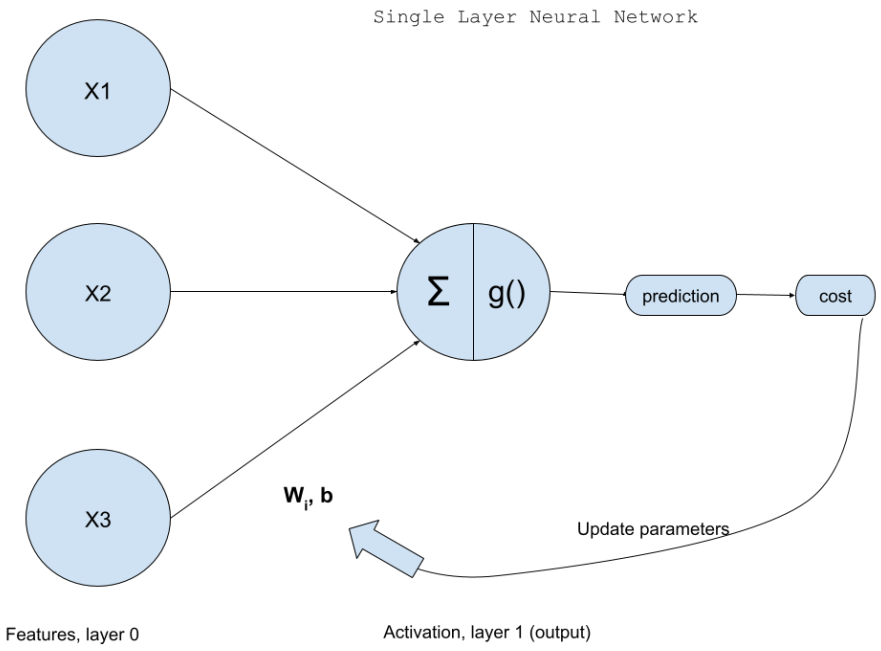


Figure 5.2: Single Layer Neural Network

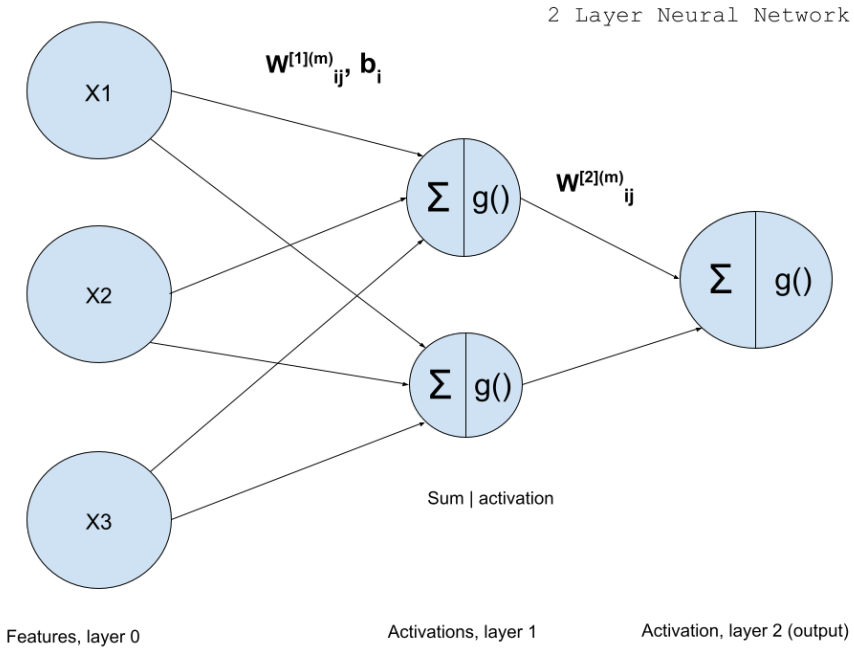


Figure 5.3: Shallow Neural Network