# PadhAI Week 3: Sigmoid Neuron & Gradient Descent

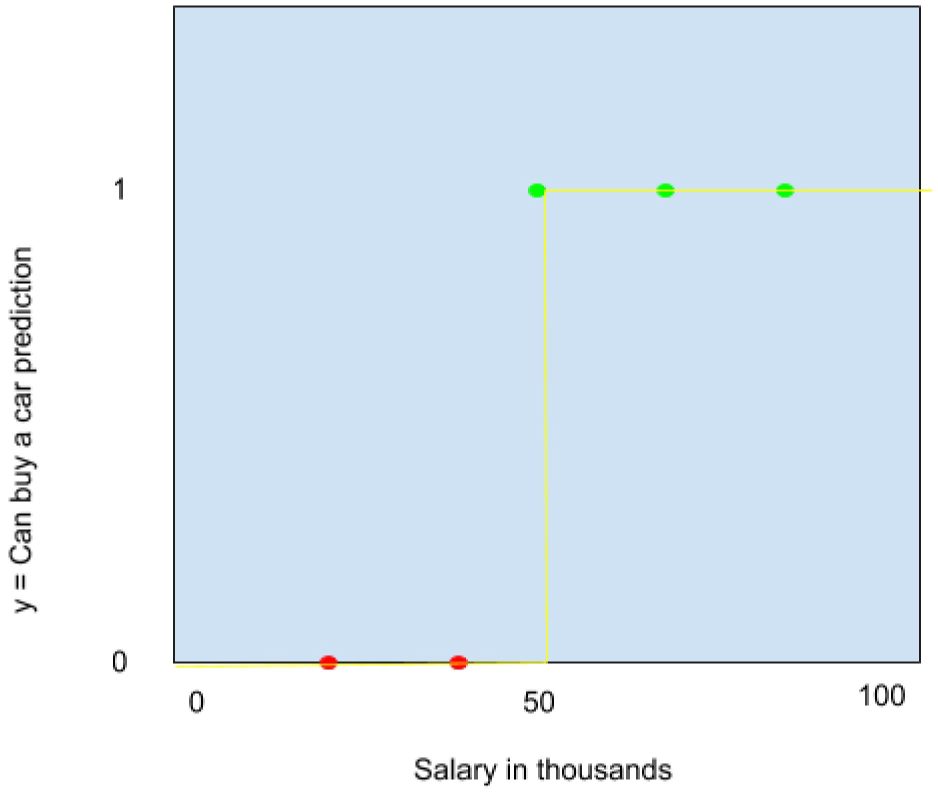
by Manick Vennimalai

## 3.0: Limitations of Perceptron

1. Perceptron model: y = $\Sigma^{n}_{i=1} w_i x_i >= b$
2. Consider the following dataset

| Salary in thousands | Can buy a car? |
|---|---|
| 20 | 0 |
| 30 | 0 |
| 50 | 1 |
| 60 | 1 |
| 70 | 1 |

3. Plotting the perceptron results



4. 
5. The function looks like a step, it has a value(50) beyond which the curve suddenly changes orientation
6. So it divides the input space into two halves with negative on one side and positive on one side
7. This case reproduces in higher dimensions, 2D, 3D etc.
8. It cannot be applied to non-linearly separable data.
9. The function is harsh at the boundary. For eg: 49.9 would be 0 and 50.1 would be 1. In practical real-life scenarios, a much smoother boundary is more applicable.

10. What is the road ahead?
    a. Data: Real inputs 😃
    b. Task: Regression/Classification, Real output 😃
    c. Model: Smooth at boundaries, Non-linear(😃 and ☹ because it's not a very advanced non-linear model)
    d. Loss: $\Sigma_i(y_i - \hat{y}_i)^2$ 😃
    e. Learning: A more generic Learning Algorithm 😃
    f. Evaluation: Accuracy, Root-mean-squared-error

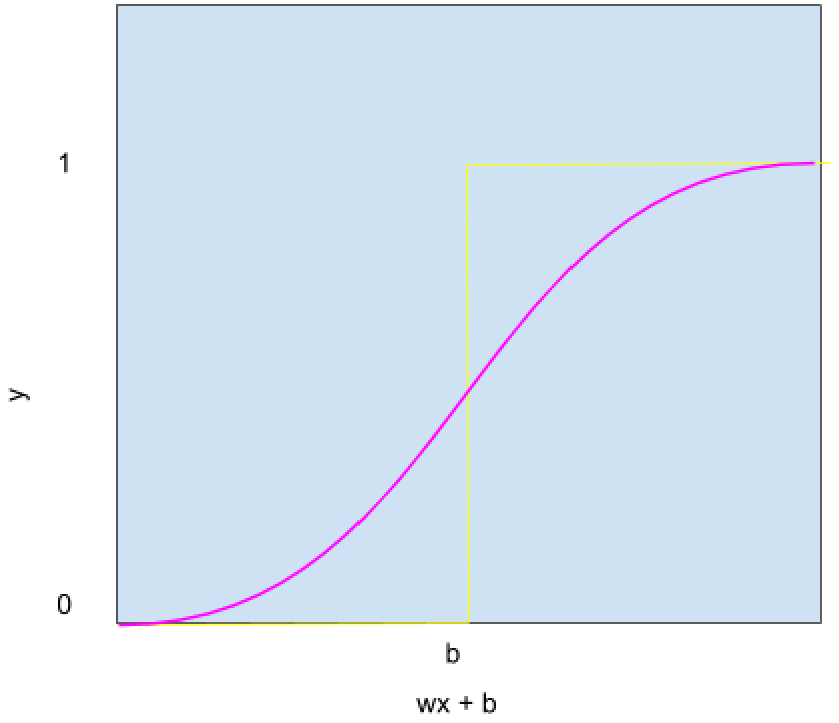
# 3.1: Sigmoid Neuron

## 3.1.1: Sigmoid Model

### 3.1.1a: Model Part 1

Can we have a smoother (not-so-harsh) function?

1. The sigmoid function is provides a smoother, s-shaped curve as opposed to a stepped line.

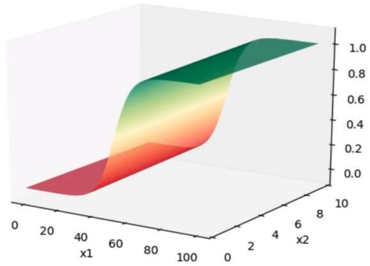


2.
3. The function is defined as y = 1/(1 + exp(-($\Sigma_i w_i x_i$ + b)), where i = 1 to n
4. Substituting different values for ($w^T x$ + b) and y, we will be able to trace out a curve similar to the one drawn above

## 3.1.1b: Model Part 2

What happens when we have two inputs?

1. Consider y = 1/(1 + exp(-($w_1x_1$ + $w_2x_2$ + b))



2. 
3. Substituting different values of w and x would result in the shape as seen in the curve above
4. Whenever $w_1x_1$ + $w_2x_2$ + b = 0, then y = 0.5

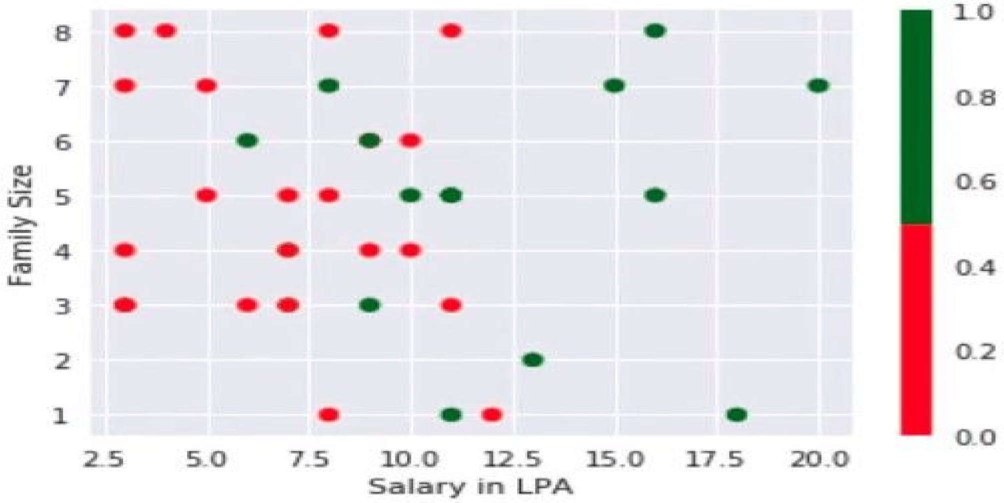
## 3.1.1c: Model Part 3

How does this help when the data is not linearly separable

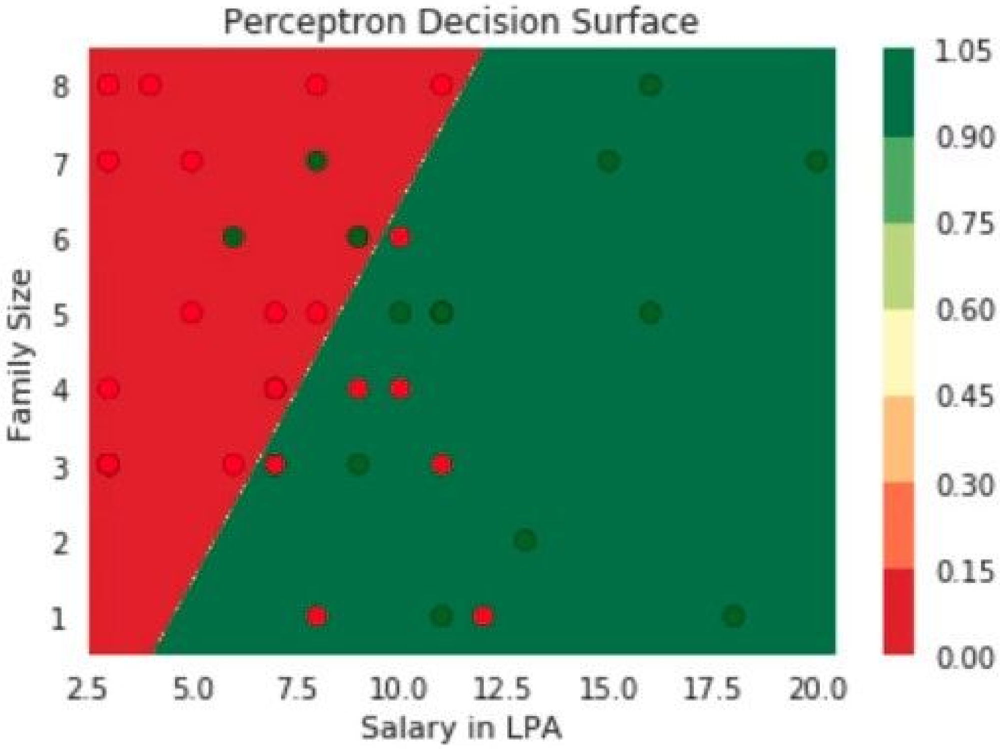
1. y = 1/(1 + exp(-($w^Tx$ + b))
2. Consider the following dataset

| | Salary in LPA | Family Size | Buys Car? |
|---|---|---|---|
| 0 | 11 | 8 | 1 |
| 1 | 20 | 7 | 1 |
| 2 | 4 | 8 | 0 |
| 3 | 8 | 7 | 0 |
| 4 | 11 | 5 | 1 |

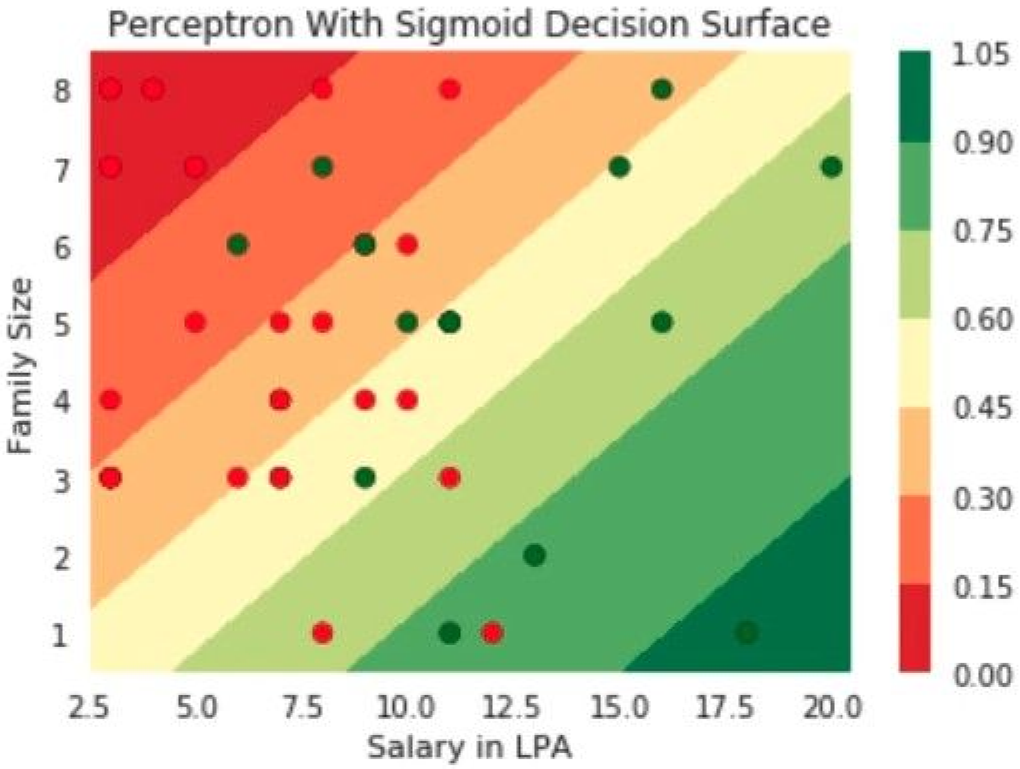
3. The dataset is visualised



4. Decision Boundary: Perceptron



5. Decision boundary: Perceptron with sigmoid. (Not optimised to separate outputs efficiently)



6. Here even the sigmoid function doesn't effectively separate the outputs.
7. We must play around with different values of w and b to find the best fit
8. This can be done with the learning algorithm

### 3.1.1d: Model Part 4

How does the function behave if we change w and b

1. **w**: (controls the slope)
   a. Negative w, negative slope, mirrored s-shape, becomes more harsh(vertical/less smooth) the more negative it goes
   b. Positive w, positive slope, normal s-shape, becomes more harsh(vertical/less smooth) the more positive it goes
2. **b**: (controls the midpoint)
   a. y = 1/(1 + exp(-(wx + b)) = ½ (for w=1.00, b = -5)
   b. exp(-(wx + b)) = 1
   c. wx + b = 0
   d. x = -b/w (As b becomes more -ve, boundary moves more to the right +ve, and vice versa)

## 3.1.2: Sigmoid Data and Tasks

What kind of data and tasks can Sigmoid Neuron process

1. Here, the Sigmoid neuron can process data similar to the Perceptron, the difference being the output is real valued, from 0 to 1.
2. This allows us to perform regression: Where we predict y as a continuous value, being some function applied to x,
3. $\hat{y} = f(x)$, where f() is the sigmoid function in this case
4. Here is a sample, similar to perceptron except for real values output y.

| | phone 1 | phone 2 | phone 3 | phone 4 | phone 5 | phone 6 | phone 7 | phone 8 | phone 9 |
|---|---|---|---|---|---|---|---|---|---|
| Launch (within 6 months) $x_1$ | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| Weight (g) $x_2$ | 151 | 180 | 160 | 205 | 162 | 182 | 138 | 185 | 170 |
| Screen Size (< 5.9in) $x_3$ | 5.8 | 6.18 | 5.84 | 6.2 | 5.9 | 6.26 | 4.7 | 6.41 | 5.5 |
| Dual sim $x_4$ | 1 | 1 | 0 | 0 | 0 | 1 | 0 | 1 | 0 |
| Internal mem(>= 64gb, 4gb ram) $x_5$ | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| NFC $x_6$ | 0 | 1 | 1 | 0 | 1 | 0 | 1 | 1 | 1 |
| Radio $x_7$ | 1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| Battery (mAh) $x_8$ | 3060 | 3500 | 3060 | 5000 | 3000 | 4000 | 1960 | 3700 | 3260 |
| Price? (k) $x_9$ | 15k | 32k | 25k | 18k | 14k | 12k | 35k | 42k | 44k |
| Liked (y) | 0.6 | 0.31 | 0.55 | 0.23 | 0.8 | 0.75 | 0.16 | 0.59 | 0.40 |

## 3.1.3: Sigmoid Loss Function

What is the loss function used for this model

1. Here the squared-error function is used **loss =** $\Sigma_i(\mathbf{y}_i - \hat{y}_i)^2$. Another useful loss function is called the cross entropy function.
2. Consider the following data

| $x_1$ | $x_2$ | y | ŷ |
|---|---|---|---|
| 1 | 1 | 0.5 | 0.6 |
| 2 | 1 | 0.8 | 0.7 |
| 1 | 2 | 0.2 | 0.2 |
| 2 | 2 | 0.9 | 0.5 |

3. Loss = $\Sigma^4_i(y_i - \hat{y}_i)^2 = 0.18$

4. This also works if y is boolean valued

| $x_1$ | $x_2$ | y | ŷ |
|---|---|---|---|
| 1 | 1 | 1 | 0.6 |
| 2 | 1 | 1 | 0.7 |
| 1 | 2 | 0 | 0.2 |
| 2 | 2 | 0 | 0.5 |

5. Loss = $(1 - 0.6)^2 + (1 - 0.7)^2 + (0 - 0.2)^2 + (0 - 0.5)^2$
6. The interesting thing to note here is that in sigmoid neuron, each individual points contribute differently to the overall loss. Some points are more correct than others and some are more wrong than others.
7. Whereas in Perceptron, it was either right or wrong, no degrees of correctness or wrongness.

# 3.2: Learning Algorithm (Gradient Descent)

## 3.2.1: Intro to Learning Algorithm

**Learning Objective:** In sigmoid neuron or any other model, we have parameters that influence the way the output is predicted (ie, the way the curve is drawn). Changing these parameters changes the curve. The objective of a learning algorithm is to determine the values for these parameters such that the overall loss of the model over the training data is minimized.

Steps (for sigmoid neuron)
1. Initialize w and b (random initialization)
2. Iterate over the data
    a. Update w and b for every iteration
3. End when satisfied (ie, loss is minimized)

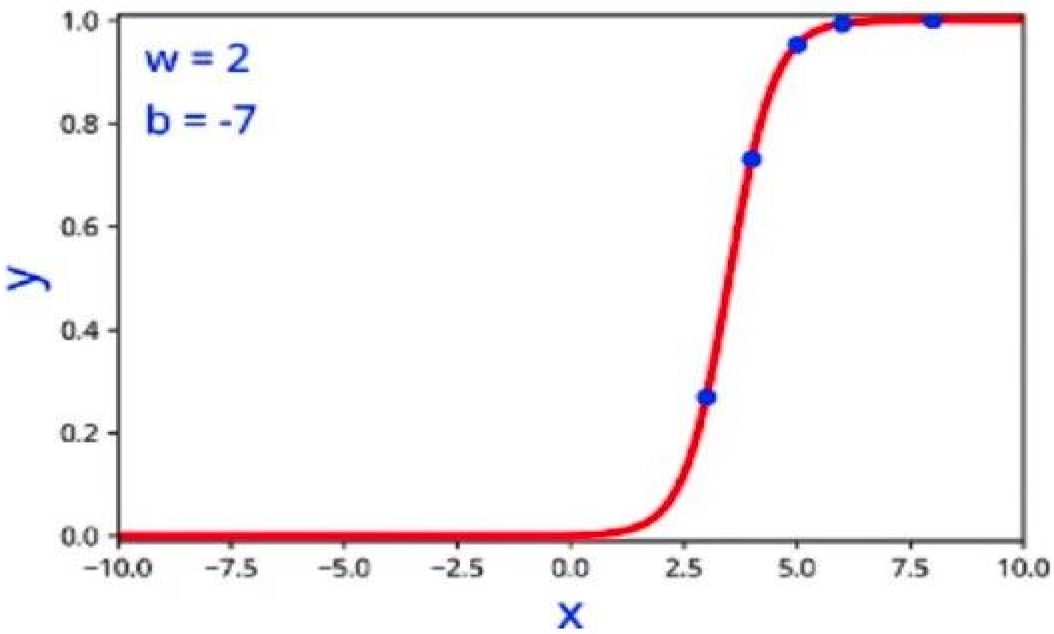
## 3.2.2: Learning by guessing

Can we try to estimate w,b by using some guess work?

1. Steps:
   a. **Initialise:** w,b to 0
   b. **Iterate over Data:** guess_and_update($x_i$)
      i. w = w + ∆w
      ii. b = b + ∆b
      iii. Here, ∆w and ∆b are the amounts we change w and b, by pure guess-work. We need to design a function to replace the guess-work.
   **c. Till satisfied**
2. Consider the following dataset

| **I/P** | **O/P** |
|---|---|
| 2 | 0.047 |
| 3 | 0.268 |
| 4 | 0.73 |
| 5 | 0.952 |
| 8 | 0.999 |

3. Manually change the slope w and the midpoint b till it looks to fit the data, then perform fine-tuning to match the training examples as closely as possible



4. We have guessed, by trial and error and found that w=2 and b=-7 fits the training data best.
5. This is only possible in lower dimensions, 1D or 2D, and becomes much harder as more features are involved.

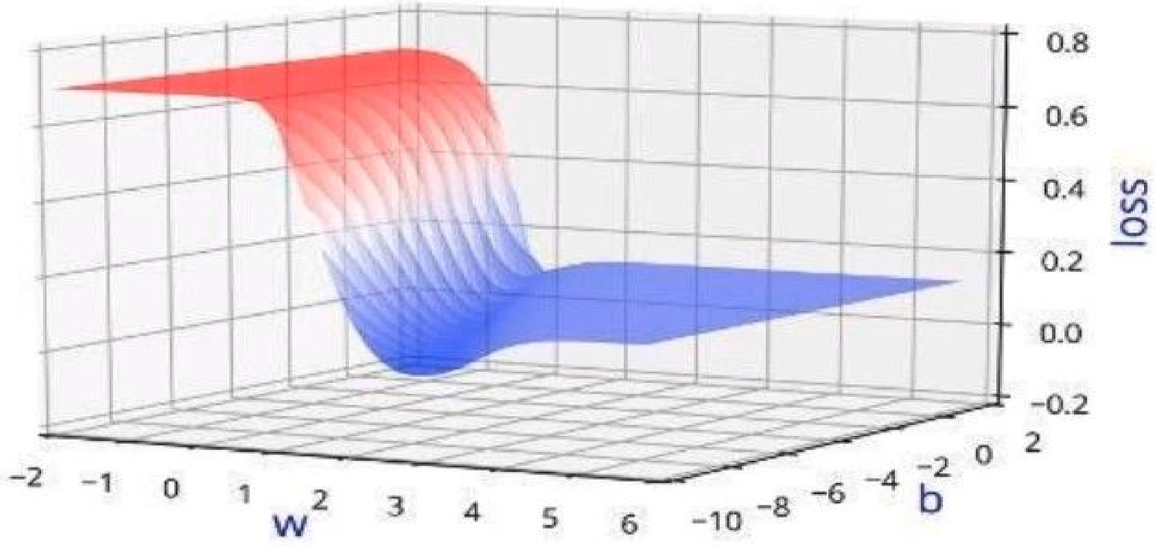
### 3.2.3: Error Surfaces for learning

Can we connect this to the loss function?

1. So far, we have iteratively modified w and b till we reached the values which yielded minimum loss
2. However, instead of a smooth descent from initial value to the minimum, the loss fluctuates each iteration.
3. For eg, for the previously used dataset, loss over each iteration was

| Iteration | w | b | loss | Increase/decrease |
|---|---|---|---|---|
| 1 | 0 | 0 | 0.1609 | - |
| 2 | 1 | 0 | 0.1064 | decrease |
| 3 | 2 | 0 | 0.1210 | increase |
| 4 | 3 | 0 | 0.1217 | increase |
| 5 | 3 | -2 | 0.1215 | decrease |
| 6 | 3 | -9 | 0.0209 | decrease |
| 7 | 2 | -9 | 0.0696 | increase |
| ...final | 2 | -7 | 0 | decrease |

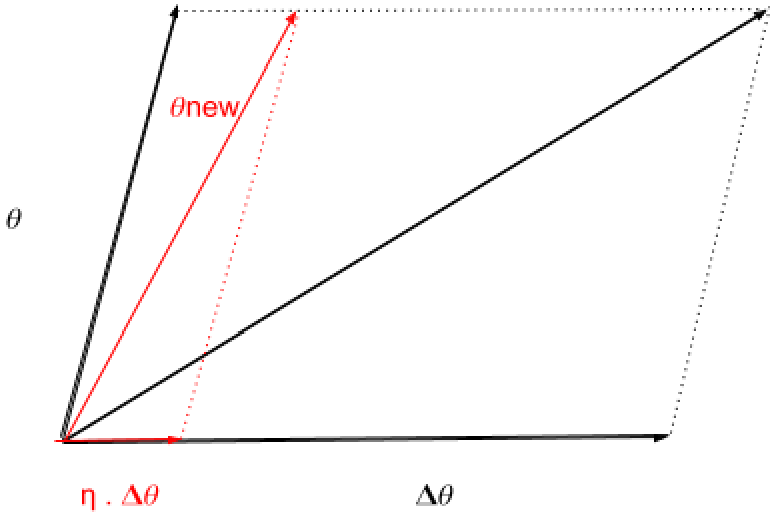
4. Now, this erratic fluctuation is undesirable. We need to use an algorithm that ensures that the loss decreases on every iteration or at the very least, doesn't increase.
5. The following image shows the plot of the loss wrt w and b. The lowest point refers to the minimum loss which corresponds to the ideal parameters of w(2) and b(-7).



## 3.2.4: Mathematical setup for the learning algorithm

What is our aim now?

1. Instead of guessing $\Delta$w and $\Delta$b, we need a principled way of changing w and b based on the loss function.
2. First, let's formulate this more mathematically
   a. $\theta$ = [w, b] (Theta is a vector containing the values of w and b)
   b. $\Delta\theta$ = [$\Delta$w, $\Delta$b] ($\Delta\theta$ is the change vector, the value we change w and b by)
   c. $\theta = \theta + \eta\Delta\theta$ (Where η is the learning rate, which allows for small changes in $\theta$)



   d. We need to compute $\Delta\theta$ such that Loss($\theta_{new}$) < Loss($\theta_{old}$)

## 3.2.5: The math-free version of the learning algorithm

What does the algorithm look like now

1. The learning algorithm
   a. **Initialise:** w, b
   b. **Iterate over data**
      i. Compute ŷ
      ii. Compute Loss(w,b)
      iii. $w_{t+1} = w_t + \eta\Delta w_t$
      iv. $b_{t+1} = b_t + \eta\Delta b_t$
   c. **Till satisfied**
2. Where
   a. model: ŷ = 1/(1 + exp(-(wx + b))
   b. Loss(w,b) = $\Sigma_i(y_i - \hat{y}_i)^2$
   c. $\Delta$w and $\Delta$b are the partial derivatives of Loss(w,b) with respect to w and b respectively.
3. Frameworks like Pytorch and Tensorflow can automatically implement the learning algorithm and return the ideal values of parameters w and b

## 3.2.6: Introducing Taylor Series

Can we get the answer from some basic mathematics.

1. Our aim is
   a. w ⇒ w + ηΔw
   b. Loss(w) > Loss(w + ηΔw)
2. Taylor Series: $f(x+\Delta x) = f(x) + \frac{\{f'(x)\}}{\{1!\}}\Delta x + \frac{\{f''(x)\}}{\{2!\}}(\Delta x^2) + \frac{\{f'''(x)\}}{\{3!\}}(\Delta x^3) + ...$
3. Here, f(x + Δx) is Loss(w + ηΔw) and f(x) is Loss(w)
4. We need to find Δx such that everything after f(x) sums to a negative value, ie, lowering the overall value of f(x + Δx)

## 3.2.7: More intuitions about Taylor Series

Can we get the answer from some basic mathematics?

1. The real aim is:
   a. w ⇒ w + ηΔw
   b. b ⇒ b + ηΔb
   c. Loss(w) > Loss(w + ηΔw)
   d. Loss(b) > Loss(b + ηΔb)
   e. Loss(w, b) > Loss(w + ηΔw, b + ηΔb)
   f. Loss($\theta$) > Loss($\theta$ + ηΔ$\theta$)          (where $\theta$ = [w, b])

2. Vectorized Taylor Series: $L(\theta+\eta u) = L(\theta) + \eta * u^T \nabla_\theta L(\theta) + \frac{\eta^2}{2!} * u^T \nabla^2 L(\theta) u + \frac{\eta^3}{3!} * ...$
3. Where, u = Δ$\theta$
4. Here, we know that in practice, η is very small ie (0.001) etc
5. So $\eta^2$ ,$\eta^3$… all end up being negligible, so remove those corresponding terms
6. New Vectorized Taylor Series: $L(\theta+\eta u) \approx L(\theta) + \eta * u^T \nabla_\theta L(\theta)$
   a. Here, $\nabla_\theta$ refers to Gradient w.r.t $\theta$ and it consists of the partial derivatives of L($\theta$) w.r.t w and b, stacked up into a vector
   b. $L(\theta+\eta u) \in \mathbb{R}$
   c. $L(\theta) \in \mathbb{R}$
   d. $\eta \in \mathbb{R}$ $\frac{\partial Loss}{\partial w}$ $\frac{\partial Loss}{\partial b}$
   e. $u^T$( $\nabla_\theta L(\theta)$ ) = Dot product of Δ$\theta$ transposed and the partial derivative vector $\nabla_\theta$ (∈ $\mathbb{R}$)

| | |
|---|---|
| [Δw | Δb] |

| |
|---|
| $\frac{\partial Loss}{\partial w}$ |
| $\frac{\partial Loss}{\partial b}$ |

## 3.2.8: Deriving the Gradient Descent Update rule

How does Taylor series help us arrive at the right answer?

1. For ease of notation, let $\Delta\theta$ = u
2. Then from Taylor series,we have:
   a. $L(\theta+\eta u) = L(\theta) + \eta * u^T \nabla_\theta L(\theta)$
   b. Rearranging: $L(\theta+\eta u) - L(\theta) = \eta * u^T \nabla_\theta L(\theta)$
   c. Note,that the move $\eta u$ would only be favourable if
      i. $L(\theta+\eta u) - L(\theta) < 0$ (i.e. if the new loss is less than the previous loss)
      ii. This implies $u^T \nabla_\theta L(\theta) < 0$
   d. Now we have $u^T \nabla_\theta L(\theta) < 0$
      i. Let $\beta$ be the angle between u and $\nabla_\theta L(\theta)$, then we know that,
      ii. $-1 \leq cos(\beta) = \frac{u^T \nabla_\theta L(\theta)}{||u|| * ||\nabla_\theta L(\theta)||} \leq 1$
      iii. Multiply throughout by k = $||u|| * ||\nabla_\theta L(\theta)||$
      iv. This gives us $-k \leq u^T \nabla_\theta L(\theta) \leq k$
   e. Thus, $L(\theta+\eta u) - L(\theta) = u^T \nabla_\theta L(\theta) = k * cos(\beta)$ will be most negative when $cos(\beta) = -1$, i.e. when β is 180°
3. Gradient Descent Rule
   a. The direction u that we intend to move in should be at 180° w.r.t, the gradient
   b. In other words, move in a direction opposite to the gradient
4. Parameter Update Rule
   a. $w_{t+1} = w_t - \eta\Delta w_t$
   b. $b_{t+1} = b_t + \eta\Delta b_t$
   c. Where $\Delta w_t = \frac{\partial L(w,b)}{\partial w}$ $at\ w = w_t\ ,\ b = b_t$
   d. Where $\Delta b_t = \frac{\partial L(w,b)}{\partial b}$ $at\ w = w_t\ ,\ b = b_t$

## 3.2.9: The complete learning algorithm

How does the algorithm look like now?

1. The algorithm
   a. **Initialise:** w, b randomly
   b. **Iterate over data**
      i. Compute ŷ
      ii. Compute L(w,b)
      iii. $w_{t+1} = w_t - \eta\Delta w_t$
      iv. $b_{t+1} = b_t + \eta\Delta b_t$
      v. Pytorch/Tensorflow have functions to compute $\frac{\delta l}{\delta w}$ $and$ $\frac{\delta l}{\delta b}$
   c. **Till satisfied**
      i. Number of epochs is reached ( ie 1000 passes/epochs)
      ii. Continue till Loss < ε (some defined value)

iii. Continue till Loss(w,b)$_{t+1}$ ≈ Loss(w,b)$_t$

## 3.2.10: Computing Partial Derivatives

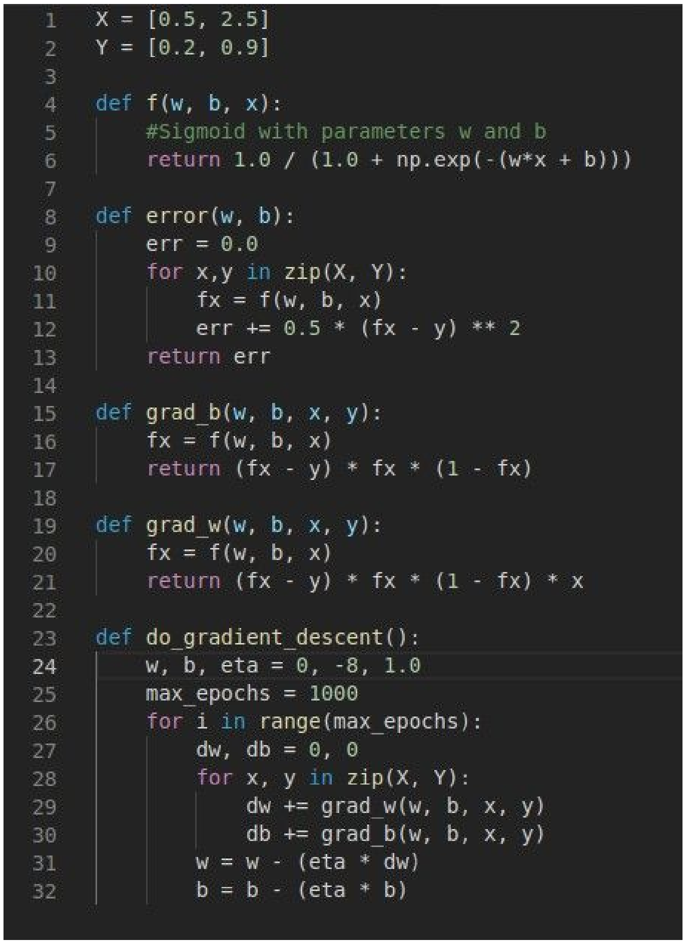
How do I compute Δw and Δb

1. Consider the following example
2. Loss = $\frac{1}{5}\Sigma^5_{i=1}(f(x_i) - y_i)^2$ (where f(x$_i$) refers to the sigmoid function
3. $\Delta w = \frac{\partial L}{\partial w} = \frac{1}{5}\Sigma^5_{i=1}\frac{\delta}{\delta w}(f(x_i) - y_i)^2$
4. Let's consider only one term in this sum
5. $\nabla w = \frac{\delta}{\delta w}[\frac{1}{2} * (f(x) - y)^2]$ (where $\nabla w$ refers to the gradient/partial derivative of L(w,b) w.r.t w)
6. Using chain rule, we expand it
    a. $\nabla w = \frac{1}{2} * [2 * (f(x) - y) * \frac{\delta}{\delta w}(f(x) - y)]$
    b. $\nabla w = (f(x) - y) * \frac{\delta}{\delta w}(f(x))$
    c. $\nabla w = (f(x) - y) * \frac{\delta}{\delta w}(\frac{1}{1+e^{-(wx+b)}})$, Let's look into the derivative of the sigmoid function in detail
        i. $\frac{\delta}{\delta w}(\frac{1}{1+e^{-(wx+b)}})$
        ii. $\frac{-1}{(1+e^{-(wx+b)})^2}\frac{\delta}{\delta w}(e^{-(wx+b)})$
        iii. $\frac{-1}{(1+e^{-(wx+b)})^2} * (e^{-(wx+b)})\frac{\delta}{\delta w}(-(wx + b))$
        iv. $\frac{-1}{(1+e^{-(wx+b)})^2} * (e^{-(wx+b)}) * (-x)$
        v. $\frac{1}{(1+e^{-(wx+b)})} * \frac{(e^{-(wx+b)})}{(1+e^{-(wx+b)})} * (x)$
        vi. $f(x) * (1 - f(x)) * x$
    d. Therefore, $\nabla w = (f(x) - y) * f(x) * (1 - f(x)) * x$
7. For each of the 5 points
    a. $\Delta w = \frac{1}{5}\Sigma^5_{i=1}(f(x_i) - y_i) * f(x_i) * (1 - f(x_i)) * x_i$
    b. Similarly $\Delta b = \frac{1}{5}\Sigma^5_{i=1}(f(x_i) - y_i) * f(x_i) * (1 - f(x_i))$

## 3.2.11: Writing the code

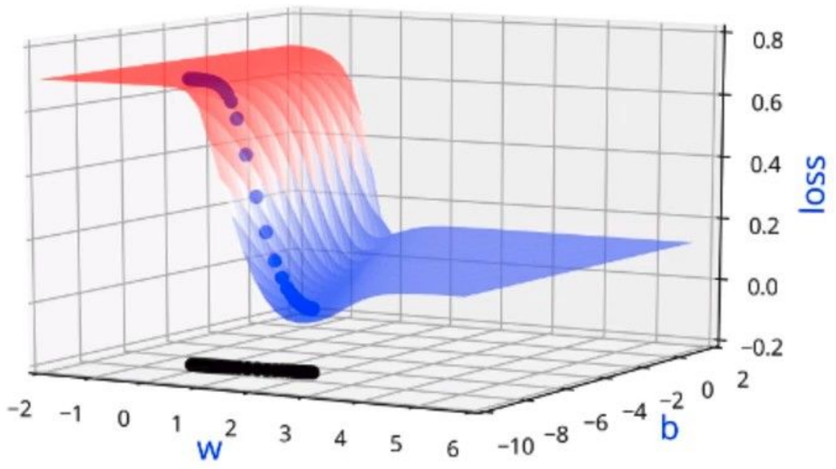
How do we implement this in Python

1. Here is the Python code for Gradient descent



```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x):
    #Sigmoid with parameters w and b
    return 1.0 / (1.0 + np.exp(-(w*x + b)))

def error(w, b):
    err = 0.0
    for x,y in zip(X, Y):
        fx = f(w, b, x)
        err += 0.5 * (fx - y) ** 2
    return err

def grad_b(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx)

def grad_w(w, b, x, y):
    fx = f(w, b, x)
    return (fx - y) * fx * (1 - fx) * x

def do_gradient_descent():
    w, b, eta = 0, -8, 1.0
    max_epochs = 1000
    for i in range(max_epochs):
        dw, db = 0, 0
        for x, y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - (eta * dw)
        b = b - (eta * b)
```

2. This is how the algorithm works



## 3.2.12: Dealing with more than 2 parameters

What happens when we have more than 2 parameters

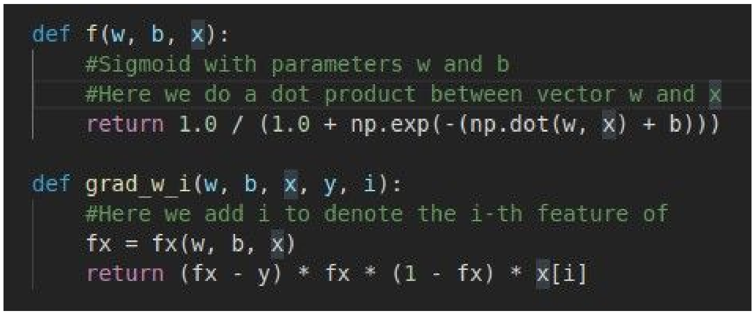
1. Consider the following dataset

| ER_visits | Narcotics | Pain | TotalVisits |
|---|---|---|---|
| 0 | 2 | 6 | 11 |
| 1 | 1 | 4 | 25 |
| 0 | 0 | 5 | 10 |
| 1 | 3 | 5 | 7 |

2. Then,
    a. $z = \Sigma^n_{i=1} w_i x_i$
    b. Or z = ($w_1$ * ER_visits) + ($w_2$ * Narcotics) + ($w_3$ * Pain) + ($w_4$ * TotalVisits) + b
    c. $\hat{y} = \frac{1}{1+e^{-z}}$
3. So the algorithm is as follows
    a. **Initialise:** $w_1$, $w_2$, $w_3$, $w_4$ and b randomly
    b. **Iterate over data**
        i. Compute ŷ
        ii. Compute L(w,b)
        iii. $w_1 = w_1 - \eta\Delta w_1$
        iv. $w_2 = w_2 - \eta\Delta w_2$
        v. $w_3 = w_3 - \eta\Delta w_3$
        vi. $w_4 = w_4 - \eta\Delta w_4$
        vii. $b = b + \eta\Delta b$
        viii. Where $\Delta w_j = \Sigma^m_{i=1}(\hat{y} - y) * (\hat{y}) * (1 - \hat{y}) * x_{ij}$
    c. **Till satisfied**

i. Number of epochs is reached ( ie 1000 passes/epochs)
ii. Continue till Loss < ε (some defined value)
iii. Continue till $Loss(w,b)_{t+1} \approx Loss(w,b)_t$

d. A few of the functions from the code also change, namely



```
def f(w, b, x):
    #Sigmoid with parameters w and b
    #Here we do a dot product between vector w and x
    return 1.0 / (1.0 + np.exp(-(np.dot(w, x) + b)))

def grad_w_i(w, b, x, y, i):
    #Here we add i to denote the i-th feature of
    fx = fx(w, b, x)
    return (fx - y) * fx * (1 - fx) * x[i]
```

e. The function do_grad_descent also changes, but we will figure it out in the practical implementation

## 3.2.13: Sigmoid Evaluation

How do you check the performance

1. Consider the following test data

| | phone1 | phone2 | phone3 | phone4 |
|---|---|---|---|---|
| Launch (within 6 months) $x_1$ | 1 | 0 | 0 | 1 |
| Weight (g) $x_2$ | 0.2 | 0.73 | 0.6 | 0.8 |
| Screen Size (< 5.9in) $x_3$ | 0.2 | 0.7 | 0.8 | 0.9 |
| Dual sim $x_4$ | 0 | 1 | 0 | 0 |
| Internal mem(>= 64gb, 4gb ram) $x_5$ | 1 | 0 | 0 | 0 |
| NFC $x_6$ | 0 | 0 | 1 | 0 |
| Radio $x_7$ | 1 | 1 | 1 | 0 |
| Battery (mAh) $x_8$ | 0.83 | 0.96 | 0.9 | 0.2 |
| Price? (k) $x_9$ | 0.34 | 0.4 | 0.6 | 0.1 |
| Liked (y) | 0.17 | 0.67 | 0.9 | 0.3 |
| Predicted($\hat{y}$) | 0.24 | 0.67 | 0.9 | 0.3 |

2. Calculate the Root Mean Square Error
3. $RMSE = \sqrt{\frac{1}{n}\Sigma^{n}{}_{i=1}(y-\hat{y})^2}$
4. Here, RMSE = 0.311, the smaller the better
5. For classification problems, set a threshold ε, such that
   a. $(y|\,\hat{y} < \varepsilon) = 0$

b. $(y|\hat{y} >= \varepsilon) = 0$

## 3.4: Summary

Let's compare MP Neuron, Perceptron and Sigmoid Neuron

| | Data | Task | Model | Loss | Learning | Evaluation |
|---|---|---|---|---|---|---|
| MP Neuron | {0,1} | Binary Classification | $g(x) = \Sigma^{n}_{i=1} x_i$<br>$y = 1$ if $g(x) >= b$<br>$y = 0$ otherwise | Loss = $\Sigma_i (y_i \mathrel{!=} \hat{y}_i)$ | Brute Force Search | Accuracy |
| Perceptron | Real Inputs | Binary Classification | $y = 1$ if $\Sigma^{n}_{i=1} w_i x_i >= b$<br>$y = 0$ otherwise | Loss = $\Sigma_i (y_i - \hat{y}_i)^2$ | Perceptron Learning Algorithm | Accuracy |
| Sigmoid | Real Inputs | Classification/ Regression | $y = \frac{1}{1 + e^{-(w^T x + b)}}$ | Loss = $\Sigma_i (y_i - \hat{y}_i)^2$ | Gradient Descent | Accuracy/ RMSE |

Fin