- Normalisation:

> ```
>   If feature range is different, for example: x1 ranges from (0,1) and x2 from (1,
> 100000) then  you should normalize your dataset.
>   Normalization helps in reaching minimum of  cost function easily (less training
> steps).
> ```

- Mini Batch Gradient Descent

> ```
>   mini-batch size = m, Batch Gradient Descent (too long per iteration)
>   mini-batch size = 1, Stochastic Gradient Descent
> ```

- Stochastic Gradient Descent

> ```
>   Very noisy
>   Use low learning rate
>   Very slow
>   Loose the speeding up due to vectorization in python.
> ```

- Choosing mini-batch size

> m<= 2000, mini-batch size = m
> common range - 64 to 1024

- Mini batch should fit in the CPU/GPU memory to avoid reduced/slow performance
- **Exponenially Weighted Average**
  - $V_t = (1\text{-}B) * V_{t\text{-}1} + theta_t$
- **Bias Correction in Exponenially Weighted Average**
  - $V_t = V_t / (1\text{-}B^t)$
- **Gradient Descent with momentum**

  > ```
  >   Gradient descent takes a lot of steps to converge to the minima.
  >   Therefore, large learnng rate cannot be used in gradient descent to avoid over
  > shooting.
  > ```

  Without momentum -
  $W := W - learningrate * dW$

  With momentum -
  $V_{dw} = B * V_{dw} + (1\text{-}B) * dW$
  $W := W - learningrate * V_{dW}$
  $b := b - learningrate * V_{db}$

  > This leads to small oscillations in vertical direction and large oscillations in horizontal directions

> Hyperparameters - B, learningrate
> Normally, B = 0.9 works well

- **RMSprop**
  *S_{dw} = B * S_{dw} + (1-B) * dw^2 (element wise square)*
  *S_{db} = B * S_{db} + (1-B) * db^2 (element wise square)*

  w := w - learningrate * dw / [sqrt(S_{dw}) + epsilon]
  b := b - learningrate * db / [sqrt(S_{db}) + epsilon]

  Here, epsilon is a small value to avoid dividing by 0

  > db is very large, S_{db} is also very large moreover dw is small and so is S_{dw}.
  > Thus, in above equations, large or small value of dw or db is normalized.
  > This, allows usage of a higher learning rate

- **Adam (Adaptive Moment Estimation) optimization algorithm**
  - Combines RMSprop and momentum
    - **On iteration t:**
      Compute dw, db using current mini batch
      V_{dw} = B_1 * V_{dw} + (1-B_1) * dW
      V_{db} = B_1 * V_{db} + (1-B_1) * db
      S_{dw} = B * S_{dw} + (1-B) * dw^2
      S_{db} = B * S_{db} + (1-B) * db^2
      $V^{corrected}{}_{dw}$ = V_{dw}/(1-$B_1{}^t$)
      $V^{corrected}{}_{db}$ = V_{db}/(1-$B_1{}^t$)
      $S^{corrected}{}_{dw}$ = S_{dw}/(1-$B_2{}^t$)
      $S^{corrected}{}_{db}$ = S_{db}/(1-$B_2{}^t$)
      w := w - learningrate * V_{dw}/ [sqrt(S_{dw}) + epsilon]
      b := b - learningrate * V_{db}/ [sqrt(S_{db}) + epsilon]
    - **Hyperparameters**
      learning rate: to be tuned
      B1: 0.9
      B2: 0.999
      epsilon: 10^{-8}

- **Learning rate decay methods**
  Learning rate can be large initially and gradually slow it down as the algorithm approaches convergence
  - learningrate = (1 / 1 + decayrate * epoch_num) * alpha
  - learningrate = $(0.95)^{epoch\_num}$ * alpha
  - learningrate = k * alpha / sqrt(epoch_num)
  - Manual decay

- Problem of local optima

```
-   If you are training a neral network with lets say, 20000 parameters then it highly unlikely t
```

```
hat the algorithm gets stuck in a local optima.
-   Out of the 20000 directions atleast 1 will have a significant gradient to guide towards minim
a.
-   Such points are called as saddle points
-   Training problem occurs at plateaus where the gradient is very low and training slows down.
```