AMRITA VISHWA VIDYAPEETHAM
DEEMED TO BE UNIVERSITY

श्रद्धावान् लभते ज्ञानम्

19CSE437
DEEP LEARNING FOR COMPUTER VISION
L-T-P-C: 2-0-3-3

Feed Forward Neural Networks

- **Optimization – Hyper Parameter Tunings**
  - **Gradient Descent Variants**
  - **Gradient Descent Optimizations**

Citation Note: content, of this presentation were inspired by the awesome lectures and the material offered by Prof. ***Mitesh M. Khapra*** *on NPTEL's Deep Learning course*

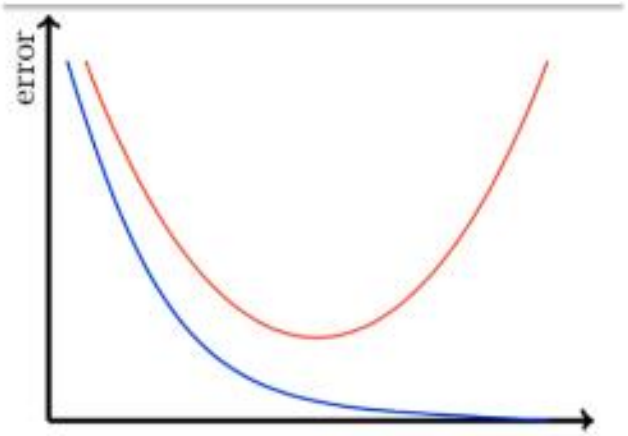
Feed Forward NN - Hyper Parameter Tunings

Network Architectures

- Number of layers
- Number of neurons

Algorithms

- Vanilla/Momentum /Nesterov GD
- AdaGrad
- RMSProp
- Adam



Activation Functions

- tanh (RNNs)
- relu (CNNs, DNNs)
- leaky relu (CNNs)

Strategies

- Batch
- Mini-Batch (32, 64, 128)
- Stochastic
- Learning rate schedule

Regularization

- L2
- Early stopping
- Dataset augmentation
- Drop-out
- Batch Normalizat

Initialization Methods

- *Xavier*
- *He*

Recap-Gradient Descent

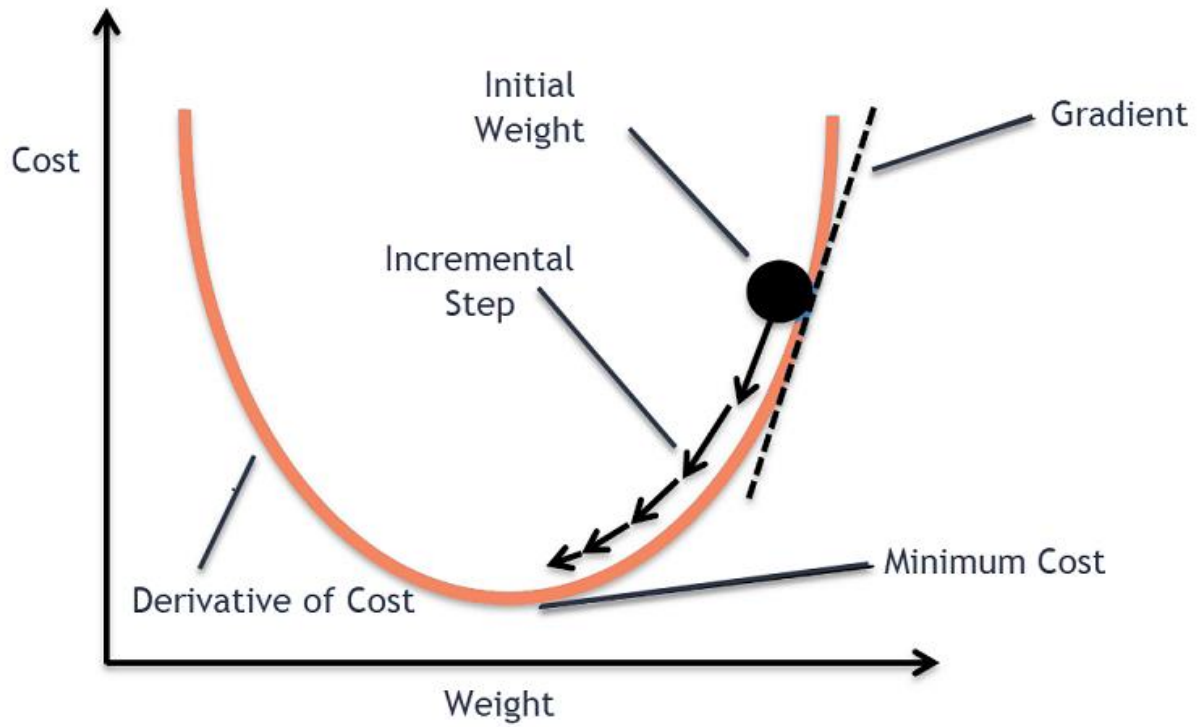
Gradient descent is a way to minimize an objective function parameterized by a model's parameters by updating the parameters in the opposite direction of the gradient of the objective function

The learning rate η determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley

Steps

1.Compute the slope (gradient) that is the first-order derivative of the function at the current point

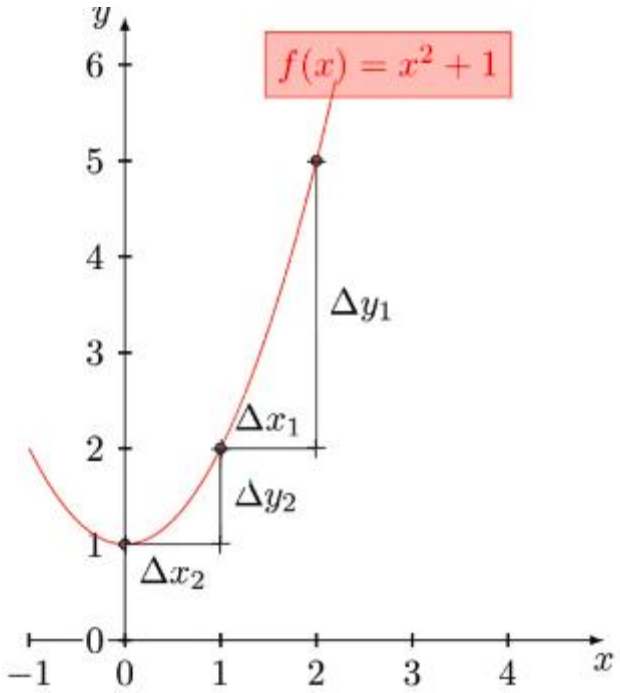
2.Move-in the opposite direction of the slope increase from the current point by the computed amount



$$w_{t+1} = w_t - \eta \Delta w_t$$
$$b_{t+1} = b_t - \eta \Delta b_t$$
$$where\ \Delta w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}\ at\ w=w_t,b=b_t, \Delta b_t = \frac{\partial \mathscr{L}(w,b)}{\partial b}\ atw=w_t,b=b_t$$

Issue with Gradient Descent optimisation

vanilla gradient descent



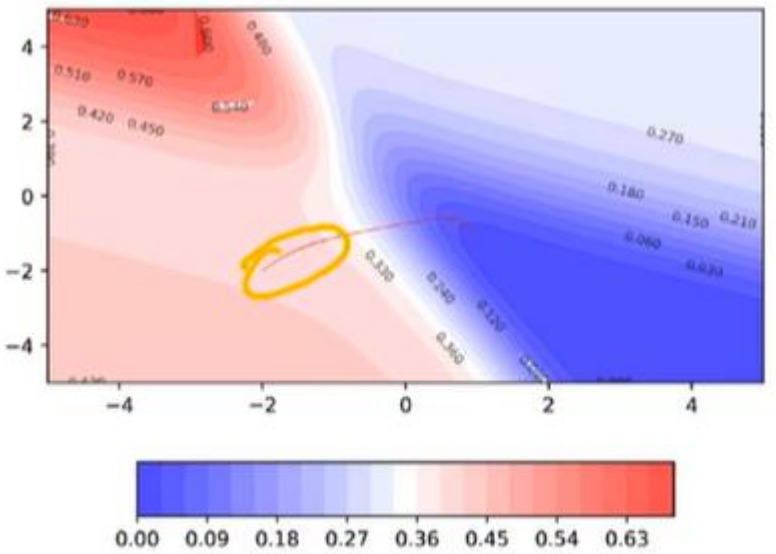
Gradient Descent Update Rule

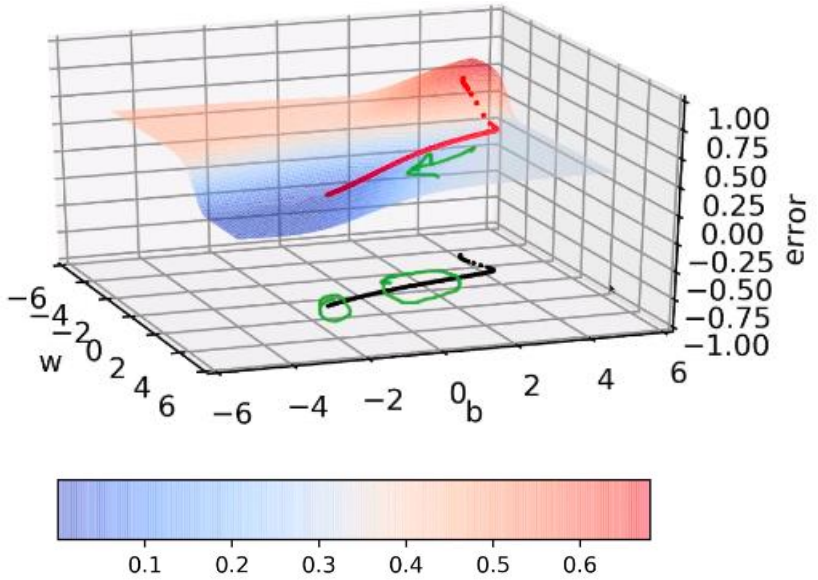
$$w = w - \eta \frac{\partial \mathcal{L}(w)}{\partial w}$$

Issues

It takes a lot of time to navigate regions having gentle slope (because the gradient in these regions is very small)

2D contour Maps-3D surface on 2D map





Momentum-Based Gradient Descent

Intuition : *If I am repeatedly being asked to move in the same direction, then I should probably gain some confidence and start taking bigger steps in that direction. Just as a ball gains momentum while rolling down a slope.*

- In addition to the current update, we also look at the history of updates.
- You can see that the current update is proportional to not just the present gradient but also gradients of previous steps, although their contribution reduces every time step by γ(gamma) times. And that is how we boost the magnitude of the update at gentle regions.

Momentum based Gradient Descent Update Rule

$$v_t = \gamma * v_{t-1} + \eta \nabla w_t$$

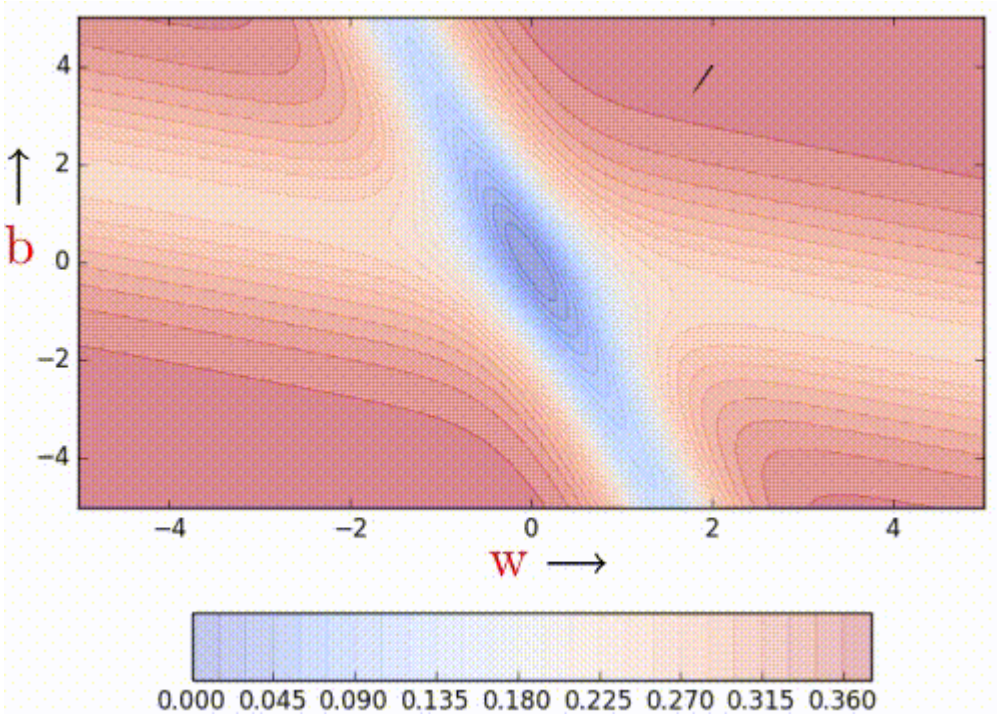
$$w_{t+1} = w_t - v_t$$

Gradient Descent Update Rule

$$w_{t+1} = w_t - \eta \nabla w_t$$

$$\begin{aligned}
v_0 &= 0 \\
v_1 &= \gamma \cdot v_0 + \eta \nabla w_1 = \eta \nabla w_1 \\
v_2 &= \gamma \cdot v_1 + \eta \nabla w_2 = \gamma \cdot \eta \nabla w_1 + \eta \nabla w_2 \\
v_3 &= \gamma \cdot v_2 + \eta \nabla w_3 = \gamma(\gamma \cdot \eta \nabla w_1 + \eta \nabla w_2) + \eta \nabla w_3 \\
&= \gamma \cdot v_2 + \eta \nabla w_3 = \gamma^2 \cdot \eta \nabla w_1 + \gamma \cdot \eta \nabla w_2 + \eta \nabla w_3 \\
v_4 &= \gamma \cdot v_3 + \eta \nabla w_4 = \gamma^3 \cdot \eta \nabla w_1 + \gamma^2 \cdot \eta \nabla w_2 + \gamma \cdot \eta \nabla w_3 + \eta \nabla w_4 \\
&\vdots \\
v_t &= \gamma \cdot v_{t-1} + \eta \nabla w_t = \gamma^{t-1} \cdot \eta \nabla w_1 + \gamma^{t-2} \cdot \eta \nabla w_1 + \ldots + \eta \nabla w_t
\end{aligned}$$

Advantage and disadvantage of momentum based GD



momentum would cause us to overshoot and run past our goal!

We can observe that momentum-based gradient descent oscillates in and out of the minima valley as the momentum carries it out of the valley. This makes us take a lot of U-turns before finally converging. **Despite these U-turns, it still converges faster than vanilla gradient descent**

It is evident that even in the regions having gentle slopes, momentum-based gradient descent can take substantial steps because the momentum carries it along.

Can we do something to reduce the oscillations/U-turns? Yes, Nesterov Accelerated Gradient Descent helps us do just that.

Nesterov Accelerated Gradient Descent (NAG)

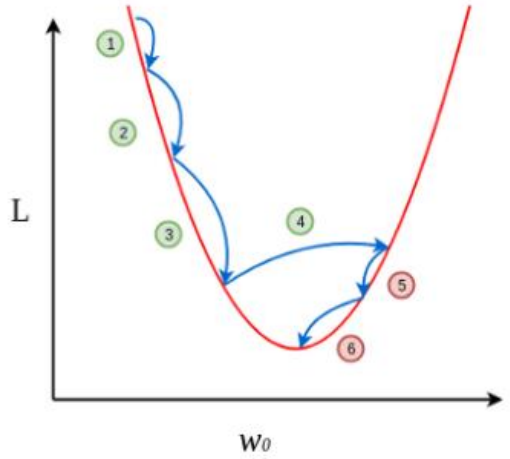
Intuition : *Look ahead before you leap!* → Effectively look ahead by calculating the gradient not w.r.t. to our current parameters θ but w.r.t. the approximate future position of our parameters:

NAG Update Rule

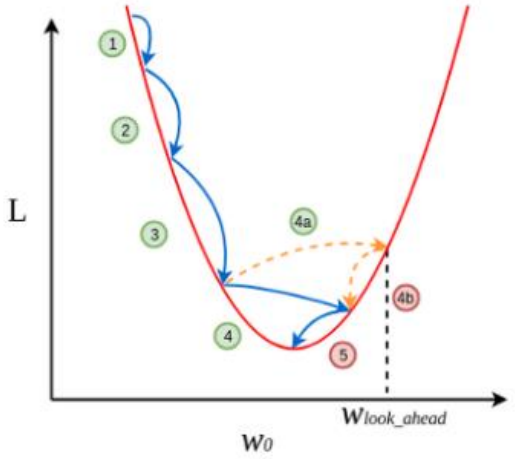
$$w_{temp} = w_t - \gamma * v_{t-1}$$

$$w_{t+1} = w_{temp} - \eta \nabla w_{temp}$$

$$v_t = \gamma * v_{t-1} + \eta \nabla w_{temp}$$



(a) Momentum-Based Gradient Descent



(b) Nesterov Accelerated Gradient Descent

Calculate the gradient at the partially updated value w_{temp} instead of calculating using the current value w_t

NAG's case, every update happens in two steps — first, a partial update, where we get to the *look_ahead* point and then the final update which is negative

This negative final update slightly reduces the overall magnitude of the update, still resulting in an overshoot but a smaller one when compared to the vanilla momentum-based gradient descent

Vanilla Gradient Descent

```
if self.algo == 'GD':
  for i in range(epochs):
    dw, db = 0, 0
    for x, y in zip(X, Y):
      dw += self.grad_w(x, y)
      db += self.grad_b(x, y)
    self.w -= eta * dw / X.shape[0]
    self.b -= eta * db / X.shape[0]
    self.append_log()
```

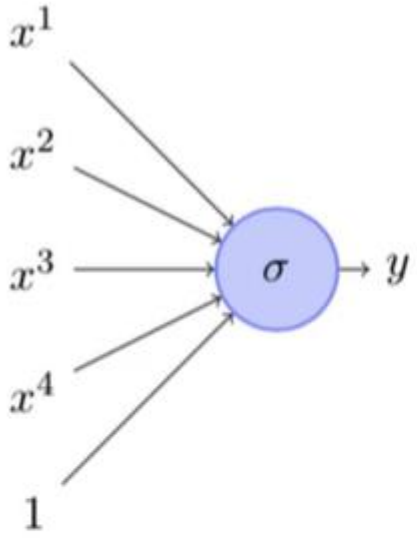
Momentum Gradient Descent

```
elif self.algo == 'Momentum':
  v_w, v_b = 0, 0
  for i in range(epochs):
    dw, db = 0, 0
    for x, y in zip(X, Y):
      dw += self.grad_w(x, y)
      db += self.grad_b(x, y)
    v_w = gamma * v_w + eta * dw
    v_b = gamma * v_b + eta * db
    self.w = self.w - v_w
    self.b = self.b - v_b
    self.append_log()
```

NAG

```
elif self.algo == 'NAG':
  v_w, v_b = 0, 0
  for i in range(epochs):
    dw, db = 0, 0
    v_w = gamma * v_w
    v_b = gamma * v_b
    for x, y in zip(X, Y):
      dw += self.grad_w(x, y, self.w - v_w, self.b - v_b)
      db += self.grad_b(x, y, self.w - v_w, self.b - v_b)
    v_w = v_w + eta * dw
    v_b = v_b + eta * db
    self.w = self.w - v_w
    self.b = self.b - v_b
    self.append_log()
```

Why adaptive learning rate needed



$$y = f(x) = \frac{1}{1+e^{-(\mathbf{w}\cdot\mathbf{x}+b)}}$$

$$\mathbf{x} = \{x^1, x^2, x^3, x^4\}$$

$$\mathbf{w} = \{w^1, w^2, w^3, w^4\}$$

$$\nabla w_1^1 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^1$$
$$\nabla w^2 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^2$$

$$\nabla w \propto x$$

Updated $w \propto \nabla w$ $\quad w_{t+1} = w_t - \eta \Delta w_t$

Can we have a different learning rate for each parameter which takes care of the frequency of features ?

In real data some features will be sparse and some dense.
We want η to be aggressive in case of sparse features and less aggressive in case of dense features- adaptive learning rate (I don't mind learning slow when feature is ON all the time. But when feature is ON less frequently then learn aggressively when it is ON)

Adagrad

Advantage

- Parameters corresponding to sparse features get better updates

Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

Intuition: Decay the learning rate for parameters in proportion to their update history (fewer updates, lesser decay)

Performing smaller updates (i.e. low learning rates) for parameters associated with frequently occurring features, and larger updates (i.e. high learning rates) for parameters associated with infrequent features. For this reason, it is well-suited for dealing with sparse data

- Denominator will be less for sparse features (History taken) and will be high for dense features. Hence high fraction for sparse features – learn aggressive. Low fraction for dense features- learn less aggressive

$$\frac{\eta}{\sqrt{(v_t)} + \epsilon}$$

Disadvantage

- The learning rate decays very aggressively as the denominator grows (not good for parameters corresponding to dense features)

RMSProp

- Adagrad got stuck when it was close to convergence (it was no longer able to move in the vertical (b) direction because of the decayed learning rate)
- RMSProp overcomes this problem by being less aggressive on the decay

Intuition: Why not decay the denominator and prevent its rapid growth ?

RMSProp

$$v_t = \beta * v_{t-1} + (1-\beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

Adagrad

$$v_t = v_{t-1} + (\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

- RMSprop resolve Adagrad's radically diminishing learning rates.
- It divides the learning rate by an exponentially decaying average of squared gradients

RMSprop

$V_0 = 0$

$V_1 = 0.1(\nabla\omega_1)^2$

$V_2 = \beta V_1 + (1-\beta)\nabla\omega_2^2$

$= (0.9)(0.1)\nabla\omega_1^2 + 0.1\nabla\omega_2^2$

$V_3 = (0.9)^2(0.1)\nabla\omega_1^2 + (0.9)(0.1)\nabla\omega_2^2$

$V_4 = (0.9)^3(0.1)\nabla\omega_1^2 + (0.9)^2(0.1)\nabla\omega_2^2$

$+ (0.9)(0.1)\nabla\omega_3^2 + (0.1)\nabla\omega_4^2$

Adagrad

$V_4 = \nabla\omega_1^2 + \nabla\omega_2^2 + \nabla\omega_3^2 + \nabla\omega_4^2$

RMSProp Will prevent blowup of denominator for dense features

Adam

In addition to storing an exponentially decaying average of past squared gradients vtvt like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients mtmt, similar to momentum.

Momentum based Gradient Descent Update Rule

$$v_t = \gamma * v_{t-1} + \eta \nabla w_t$$

$$w_{t+1} = w_t - v_t$$

RMSProp

$$v_t = \beta * v_{t-1} + (1 - \beta)(\nabla w_t)^2$$

$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} \nabla w_t$$

Adam

$$m_t = \beta_1 * v_{t-1} + (1 - \beta_1)(\nabla w_t)$$

$$v_t = \beta_2 * v_{t-1} + (1 - \beta_2)(\nabla w_t)^2$$

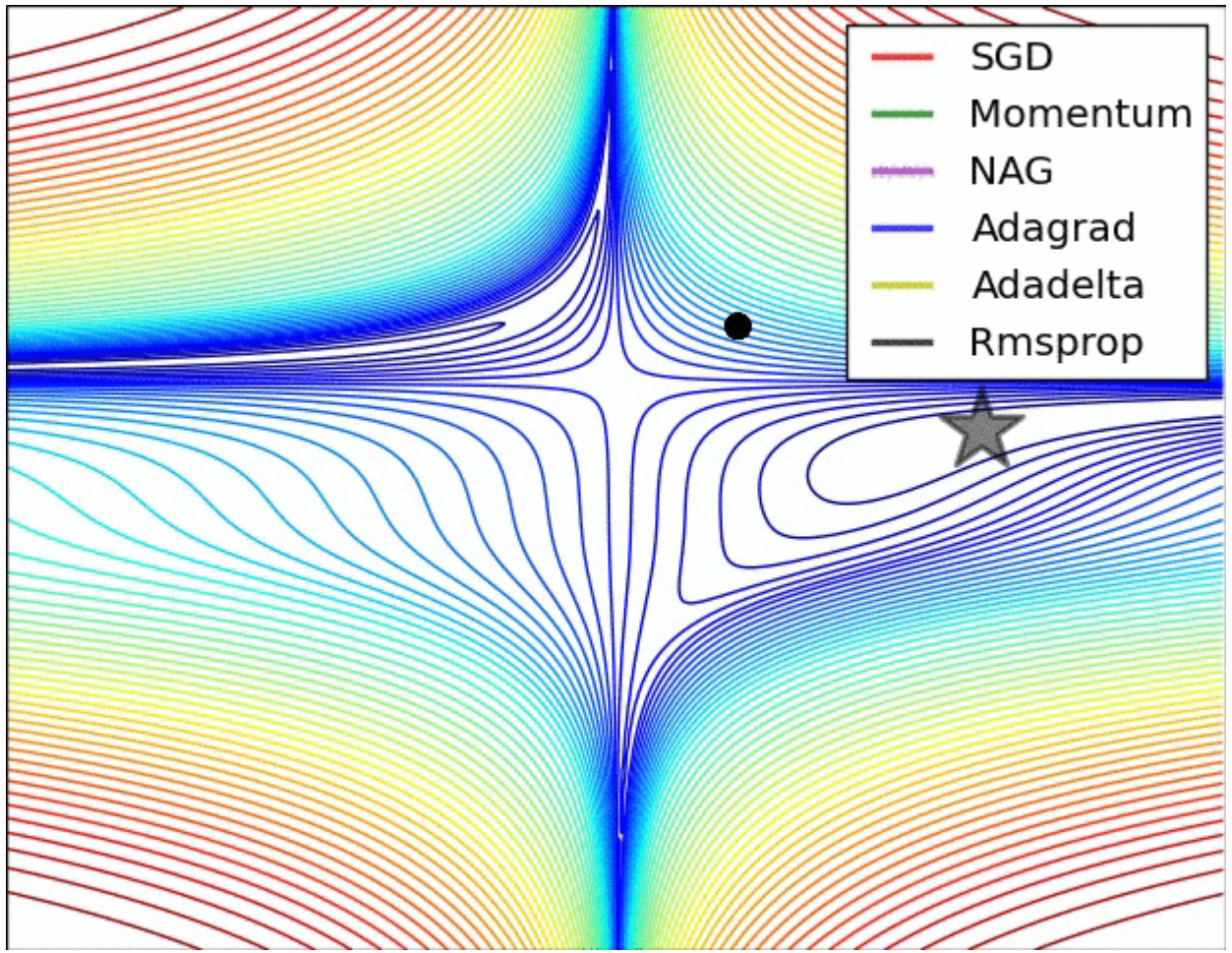
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{(v_t)} + \epsilon} m_t$$

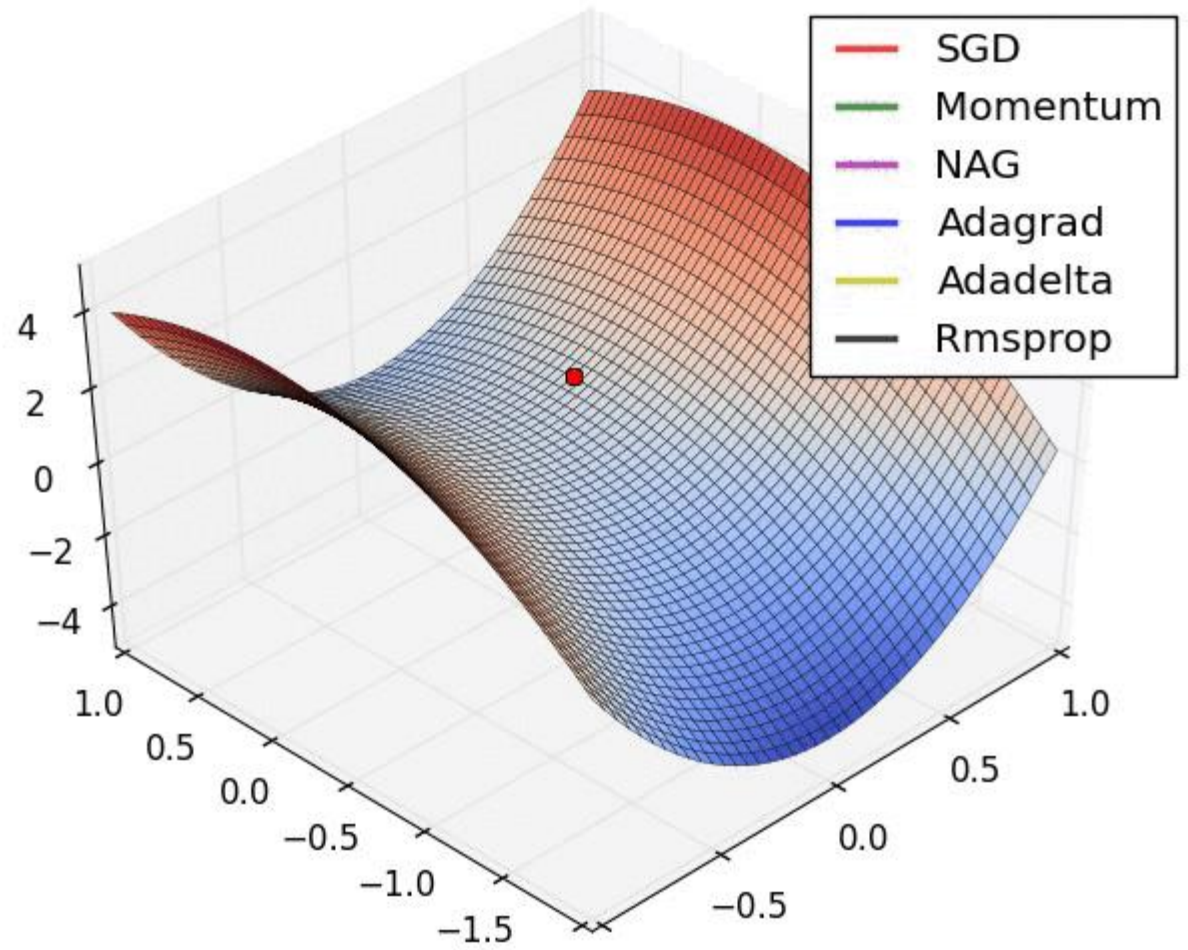
$$m_t = \frac{m_t}{1 - \beta_1^t}$$

$$v_t = \frac{v_t}{1 - \beta_2^t}$$

Combines the advantage of momentum and RMSprop

Comparison- Visualisation





```
class SN:

  def __init__(self, w_init, b_init, algo):
    self.w = w_init
    self.b = b_init
    self.w_h = []
    self.b_h = []
    self.e_h = []
    self.algo = algo

  def sigmoid(self, x, w=None, b=None):
    if w is None:
      w = self.w
    if b is None:
      b = self.b
    return 1. / (1. + np.exp(-(w*x + b)))
```

```
def error(self, X, Y, w=None, b=None):
  if w is None:
    w = self.w
  if b is None:
    b = self.b
  err = 0
  for x, y in zip(X, Y):
    err += 0.5 * (self.sigmoid(x, w, b) - y) ** 2
  return err

def grad_w(self, x, y, w=None, b=None):
  if w is None:
    w = self.w
  if b is None:
    b = self.b
  y_pred = self.sigmoid(x, w, b)
  return (y_pred - y) * y_pred * (1 - y_pred) * x

def grad_b(self, x, y, w=None, b=None):
  if w is None:
    w = self.w
  if b is None:
    b = self.b
  y_pred = self.sigmoid(x, w, b)
  return (y_pred - y) * y_pred * (1 - y_pred)
```

```
def fit(self, X, Y,
        epochs=100, eta=0.01, gamma=0.9, mini_batch_size=100, eps=1e-8,
        beta=0.9, beta1=0.9, beta2=0.9
       ):
  self.w_h = []
  self.b_h = []
  self.e_h = []
  self.X = X
  self.Y = Y

  if self.algo == 'GD':
    for i in range(epochs):
      dw, db = 0, 0
      for x, y in zip(X, Y):
        dw += self.grad_w(x, y)
        db += self.grad_b(x, y)
      self.w -= eta * dw / X.shape[0]
      self.b -= eta * db / X.shape[0]
      self.append_log()
```

```
def append_log(self):
  self.w_h.append(self.w)
  self.b_h.append(self.b)
  self.e_h.append(self.error(self.X, self.Y))
```

Namah Shivaya