Optimization Algorithms

Gradient Descent (GD), Momentum Based GD, Nesterov Accelerated GD, Stochastic GD, AdaGrad, RMSProp, Adam

Learning Parameters : Gradient Descent

Goal

Find a better way of traversing the error surface so that we can reach the minimum value quickly without resorting to brute force search!

Gradient Descent Rule

- The direction u that we intend to move in should be at 180° w.r.t. the gradient
- In other words, move in a direction opposite to the gradient

Parameter Update Equations

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

$$b_{t+1} = b_t - \eta \nabla b_t$$

$$\text{where, } \nabla w_t = \frac{\partial \mathscr{L}(w,b)}{\partial w}_{at\ w = w_t,\, b = b_t}, \nabla b_t = \frac{\partial \mathscr{L}(w,b)}{\partial b}_{at\ w = w_t,\, b = b_t}$$

So we now have a more principled way of moving in the *w*-*b* plane than our “guess work” algorithm

- Let's create an algorithm from this rule ...

```
Algorithm 1: gradient_descent()
$t \leftarrow 0$;
$max\_iterations \leftarrow 1000$;
while $t < max\_iterations$ do
    $w_{t+1} \leftarrow w_t - \eta \nabla w_t$;
    $b_{t+1} \leftarrow b_t - \eta \nabla b_t$;
end
```

- To see this algorithm in practice let us first derive ∇w and ∇b for our toy neural network

Momentum based Gradient Descent

Some observations about gradient descent

- It takes a lot of time to navigate regions having a gentle slope
- This is because the gradient in these regions is very small
- Can we do something better ?
- Yes, let's take a look at '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

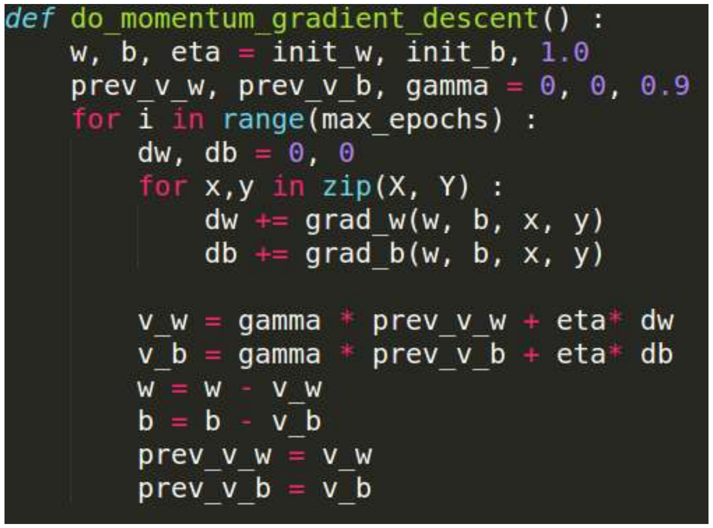
Update rule for momentum based gradient descent

$$\begin{aligned} update_t &= \gamma \cdot update_{t-1} + \eta \nabla w_t \\ w_{t+1} &= w_t - update_t \end{aligned}$$

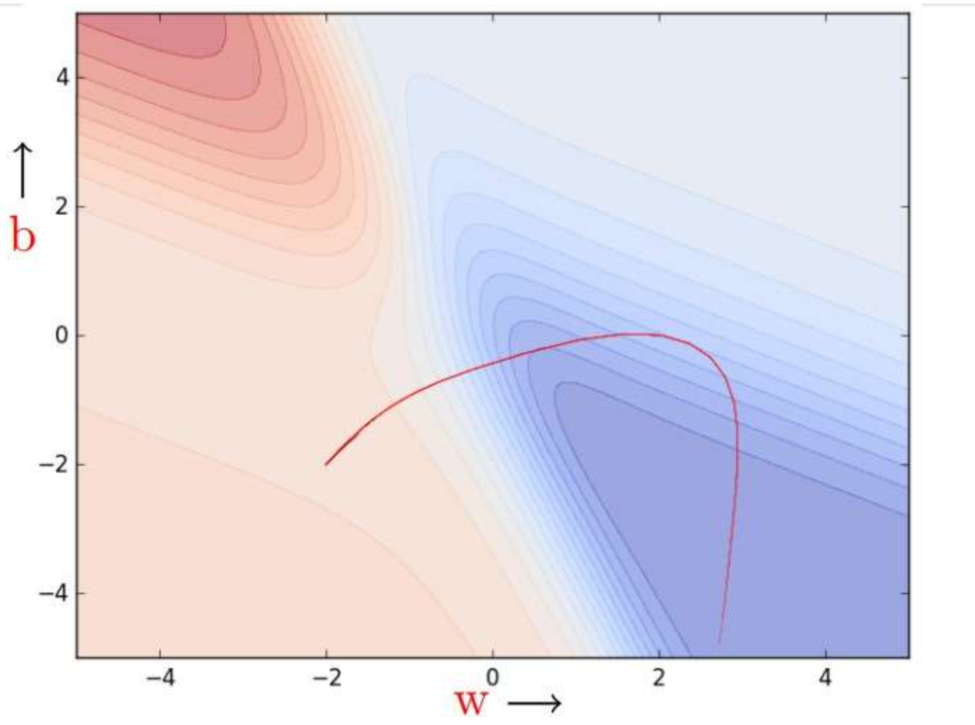
- In addition to the current update, also look at the history of updates.

$$update_t = \gamma \cdot update_{t-1} + \eta\nabla w_t$$
$$w_{t+1} = w_t - update_t$$

$$\begin{aligned}
update_0 &= 0 \\
update_1 &= \gamma \cdot update_0 + \eta\nabla w_1 = \eta\nabla w_1 \\
update_2 &= \gamma \cdot update_1 + \eta\nabla w_2 = \gamma \cdot \eta\nabla w_1 + \eta\nabla w_2 \\
update_3 &= \gamma \cdot update_2 + \eta\nabla w_3 = \gamma(\gamma \cdot \eta\nabla w_1 + \eta\nabla w_2) + \eta\nabla w_3 \\
&= \gamma \cdot update_2 + \eta\nabla w_3 = \gamma^2 \cdot \eta\nabla w_1 + \gamma \cdot \eta\nabla w_2 + \eta\nabla w_3 \\
update_4 &= \gamma \cdot update_3 + \eta\nabla w_4 = \gamma^3 \cdot \eta\nabla w_1 + \gamma^2 \cdot \eta\nabla w_2 + \gamma \cdot \eta\nabla w_3 + \eta\nabla w_4 \\
&\vdots \\
update_t &= \gamma \cdot update_{t-1} + \eta\nabla w_t = \gamma^{t-1} \cdot \eta\nabla w_1 + \gamma^{t-2} \cdot \eta\nabla w_1 + \ldots + \eta\nabla w_t
\end{aligned}$$



```
def do_momentum_gradient_descent() :
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    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)

        v_w = gamma * prev_v_w + eta* dw
        v_b = gamma * prev_v_b + eta* db
        w = w - v_w
        b = b - v_b
        prev_v_w = v_w
        prev_v_b = v_b
```



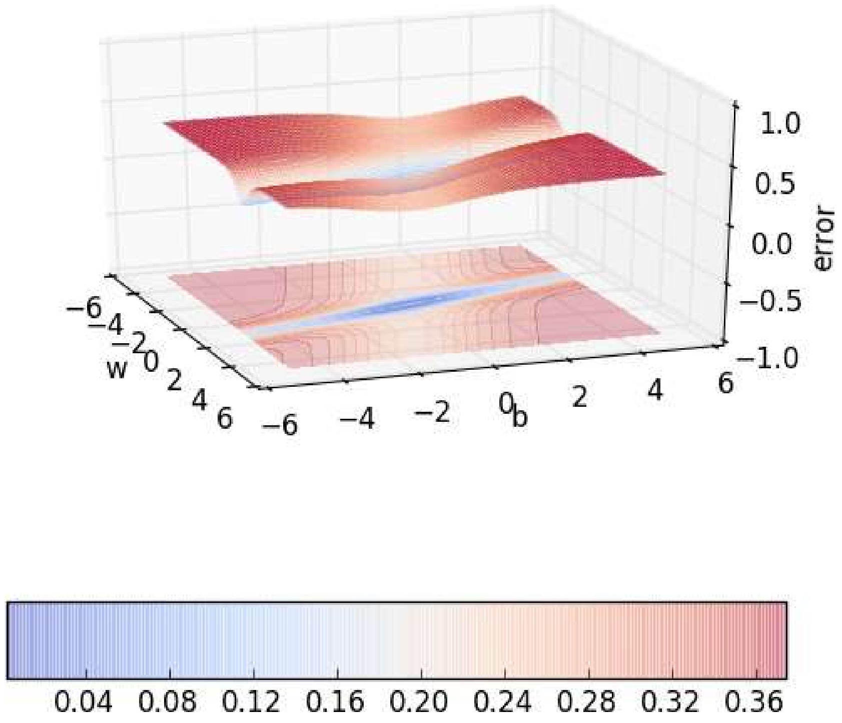
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$$

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

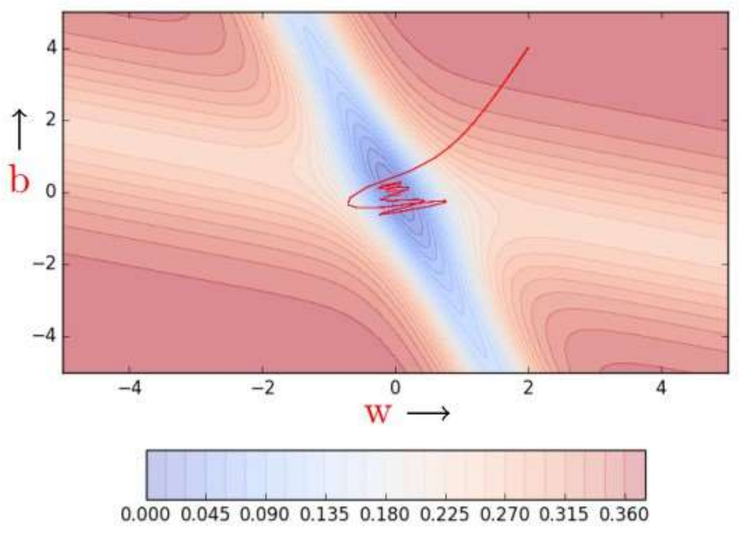
Some observations and questions

- Even in the regions having gentle slopes, momentum based gradient descent is able to take large steps because the momentum carries it along
- Is moving fast always good? Would there be a situation where momentum would cause us to run pass our goal?
- Let us change our input data so that we end up with a different error surface and then see what happens ...

- In this case, the error is high on either side of the minima valley
- Could momentum be detrimental in such cases... let's see....



- Momentum based gradient descent oscillates in and out of the minima valley as the momentum carries it out of the valley
- Takes a lot of *u*-turns before finally converging
- Despite these *u*-turns it still converges faster than vanilla gradient descent
- After 100 iterations momentum based method has reached an error of 0.00001 whereas vanilla gradient descent is still stuck at an error of 0.36



Nesterov Accelerated Gradient Descent (NAG)

Question

- Can we do something to reduce these oscillations ?
- Yes, let's look at Nesterov accelerated gradient

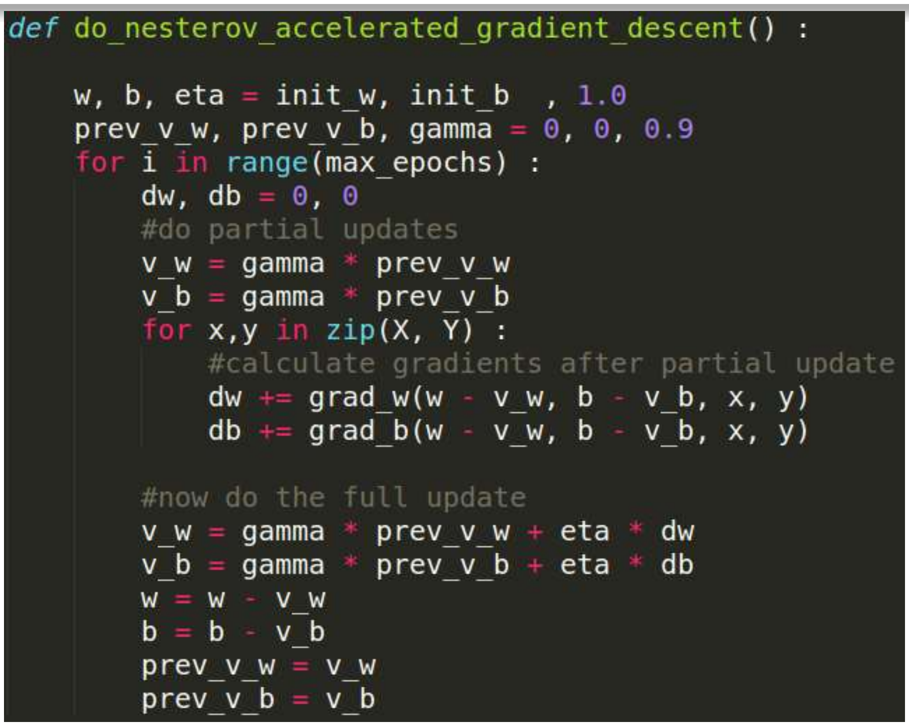
Intuition

- Look before you leap
- Recall that $update_t = \gamma \cdot update_{t-1} + \eta \nabla w_t$
- So we know that we are going to move by at least by $\gamma \cdot update_{t-1}$ and then a bit more by $\eta \nabla w_t$
- Why not calculate the gradient (∇w_{look_ahead}) at this partially updated value of w ($w_{look\_ahead} = w_t - \gamma \cdot update_{t-1}$) instead of calculating it using the current value w_t

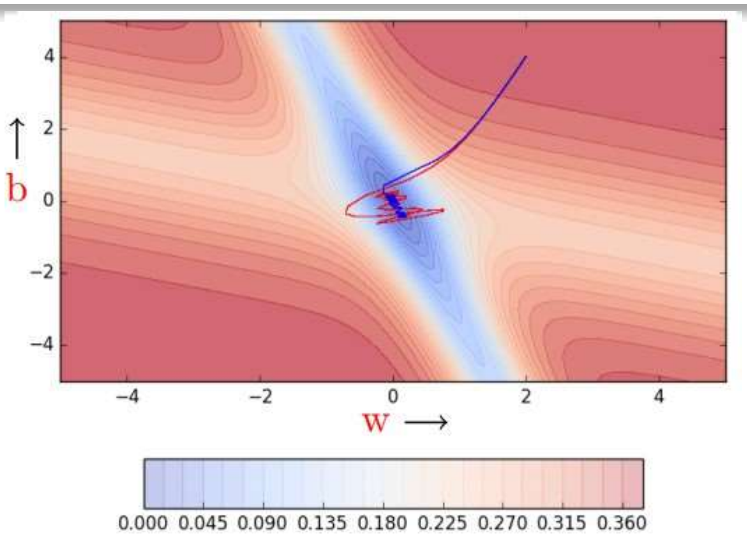
Update rule for NAG

$$\begin{aligned} w_{look\_ahead} &= w_t - \gamma \cdot update_{t-1} \\ update_t &= \gamma \cdot update_{t-1} + \eta \nabla w_{look\_ahead} \\ w_{t+1} &= w_t - update_t \end{aligned}$$

We will have similar update rule for b_t



```
def do_nesterov_accelerated_gradient_descent() :

    w, b, eta = init_w, init_b  , 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs) :
        dw, db = 0, 0
        #do partial updates
        v_w = gamma * prev_v_w
        v_b = gamma * prev_v_b
        for x,y in zip(X, Y) :
            #calculate gradients after partial update
            dw += grad_w(w - v_w, b - v_b, x, y)
            db += grad_b(w - v_w, b - v_b, x, y)

        #now do the full update
        v_w = gamma * prev_v_w + eta * dw
        v_b = gamma * prev_v_b + eta * db
        w = w - v_w
        b = b - v_b
        prev_v_w = v_w
        prev_v_b = v_b
```

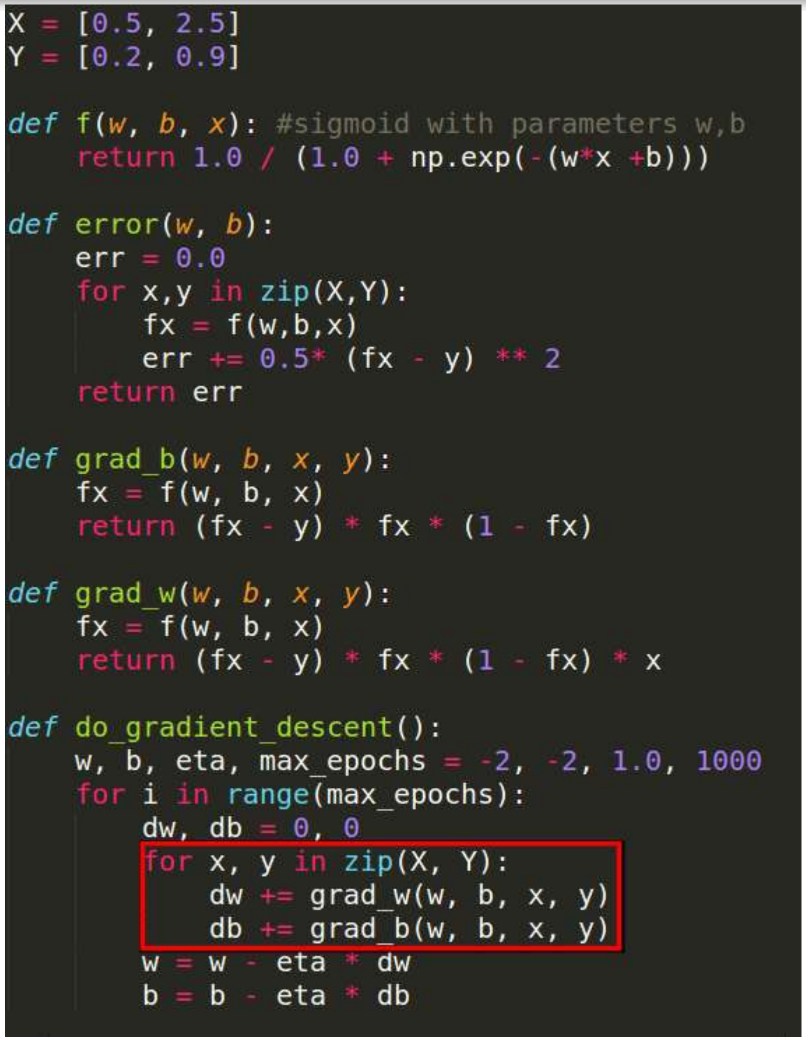


$$w_{look\_ahead} = w_t - \gamma \cdot update_{t-1}$$
$$update_t = \gamma \cdot update_{t-1} + \eta \nabla w_{look\_ahead}$$
$$w_{t+1} = w_t - update_t$$

Observations about NAG

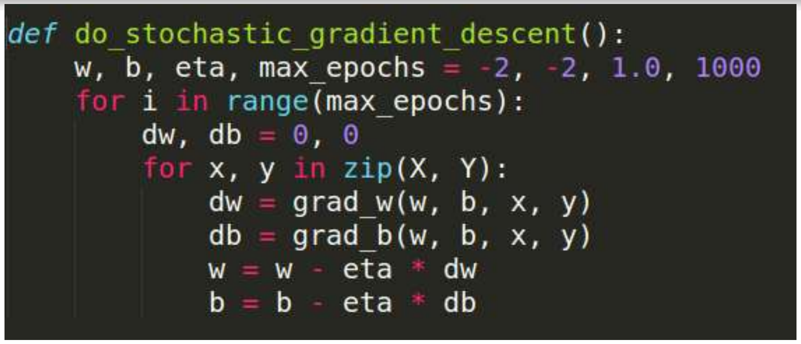
- Looking ahead helps NAG in correcting its course quicker than momentum based gradient descent
- Hence the oscillations are smaller and the chances of escaping the minima valley also smaller

Stochastic And Mini-Batch Gradient Descent



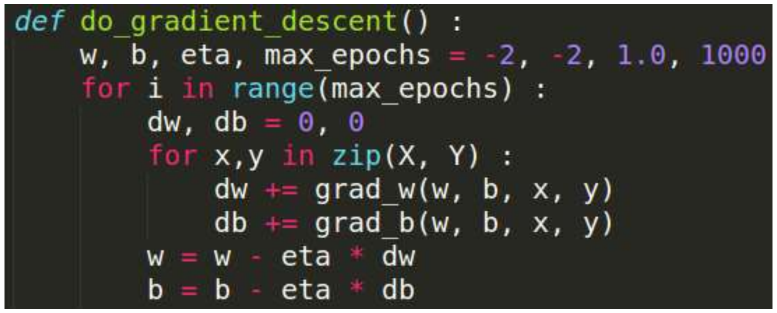
```python
X = [0.5, 2.5]
Y = [0.2, 0.9]

def f(w, b, x): #sigmoid with parameters w,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, max_epochs = -2, -2, 1.0, 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 * db
```

- Notice that the algorithm goes over the entire data once before updating the parameters
- Why? Because this is the true gradient of the loss as derived earlier (sum of the gradients of the losses corresponding to each data point)
- No approximation. Hence, theoretical guarantees hold (in other words each step guarantees that the loss will decrease)
- What's the flipside? Imagine we have a million points in the training data. To make 1 update to *w, b* the algorithm makes a million calculations. Obviously very slow!!
- Can we do something better ? Yes, let's look at stochastic gradient descent



```python
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 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 * db
```

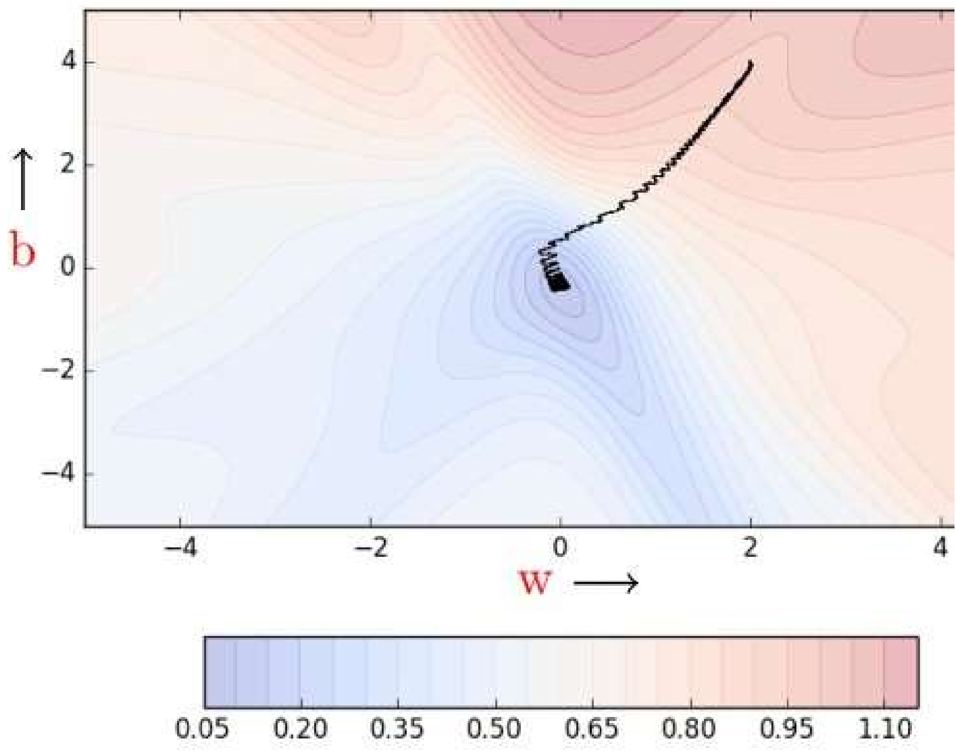
- Stochastic because we are estimating the total gradient based on a single data point. Almost like tossing a coin only once and estimating P(heads).



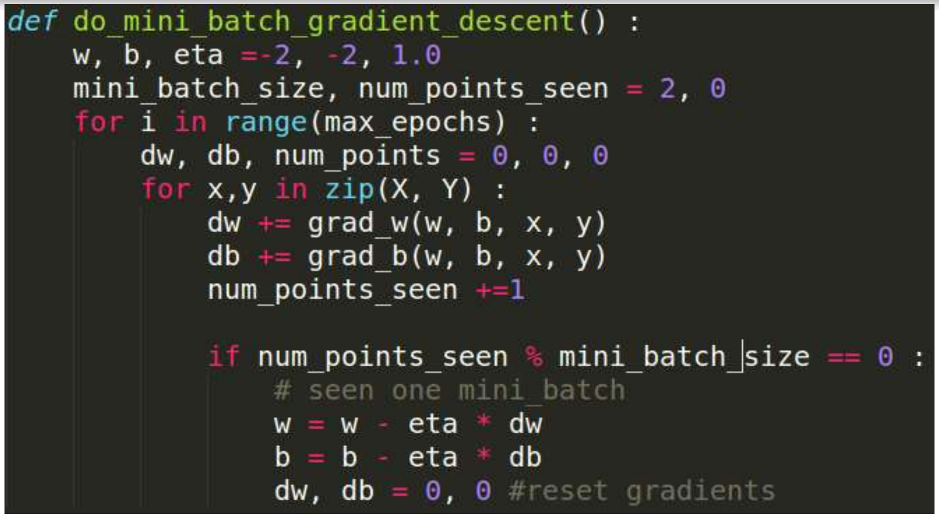
```python
def do_gradient_descent() :
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
        w = w - eta * dw
        b = b - eta * db
```

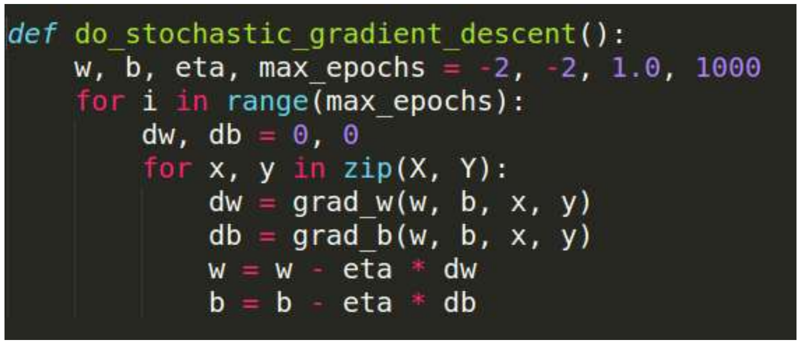
- Notice that the algorithm updates the parameters for every single data point
- Now if we have a million data points we will make a million updates in each epoch (1 epoch = 1 pass over the data; 1 step = 1 update)
- What is the flipside ? It is an approximate (rather stochastic) gradient
- No guarantee that each step will decrease the loss
- Let's see this algorithm in action when we have a few data points

- We see many oscillations. Why ? Because we are making greedy decisions.
- Each point is trying to push the parameters in a direction most favorable to it (without being aware of how this affects other points)
- A parameter update which is locally favorable to one point may harm other points (its almost as if the data points are competing with each other)
- Indeed we see that there is no guarantee that each local greedy move reduces the global error
- Can we reduce the oscillations by improving our stochastic estimates of the gradient (currently estimated from just 1 data point at a time)



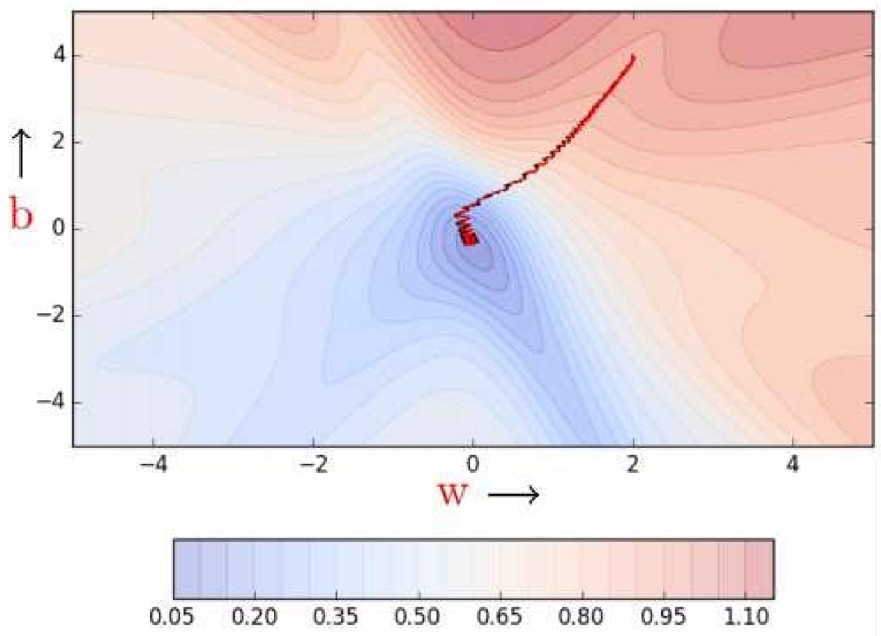
- Yes, let's look at mini-batch gradient descent


```
def do_mini_batch_gradient_descent() :
    w, b, eta =-2, -2, 1.0
    mini_batch_size, num_points_seen = 2, 0
    for i in range(max_epochs) :
        dw, db, num_points = 0, 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)
            num_points_seen +=1

            if num_points_seen % mini_batch_size == 0 :
                # seen one mini_batch
                w = w - eta * dw
                b = b - eta * db
                dw, db = 0, 0 #reset gradients
```


```
def do_stochastic_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 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 * db
```

- Notice that the algorithm updates the parameters after it sees *mini_batch_size* number of data points
- The stochastic estimates are now slightly better
- Let's see this algorithm in action when we have k = 2

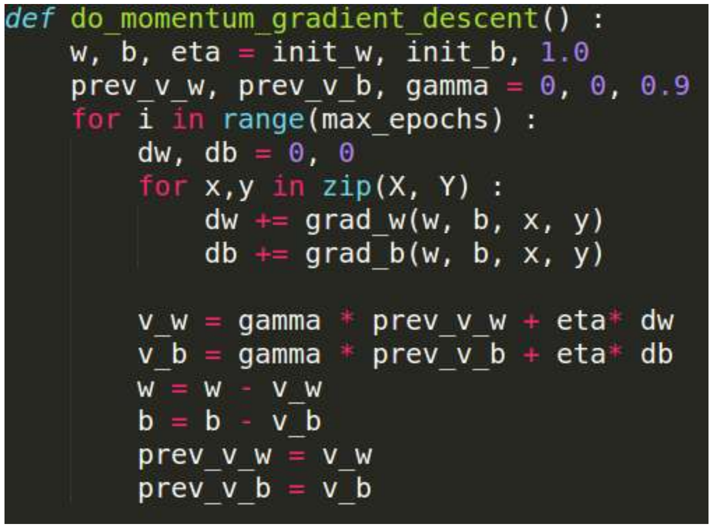
- Even with a batch size of k=2 the oscillations have reduced slightly. Why ?
- Because we now have slightly better estimates of the gradient [analogy: we are now tossing the coin k=2 times to estimate P(heads)]
- The higher the value of k the more accurate are the estimates
- In practice, typical values of k are 16, 32, 64
- Of course, there are still oscillations and they will always be there as long as we are using an approximate gradient as opposed to the true gradient



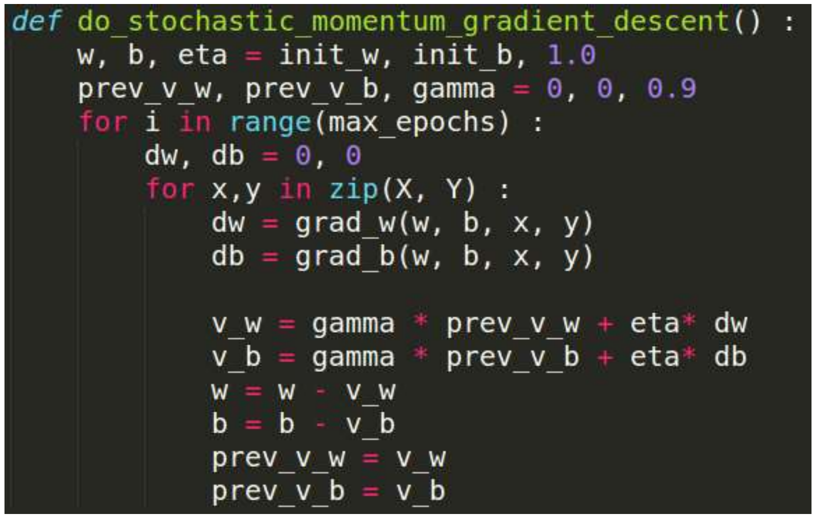
Some things to remember ....

- 1 epoch = one pass over the entire data
- 1 step = one update of the parameters
- N = number of data points
- B = Mini batch size

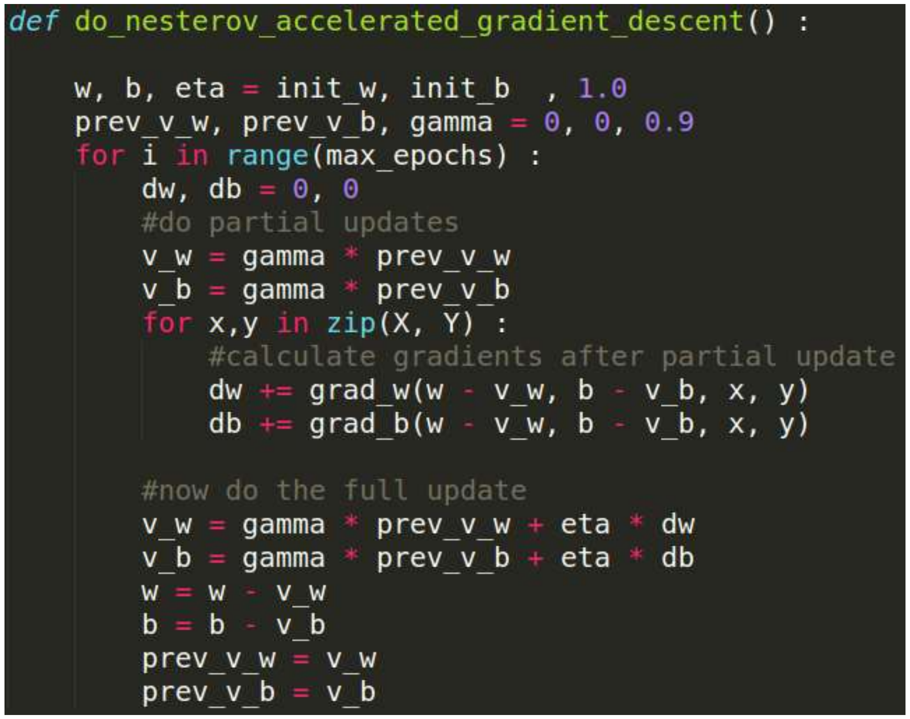
Algorithm	# of steps in 1 epoch
Vanilla (Batch) Gradient Descent	1
Stochastic Gradient Descent	N
Mini-Batch Gradient Descent	$\frac{N}{B}$



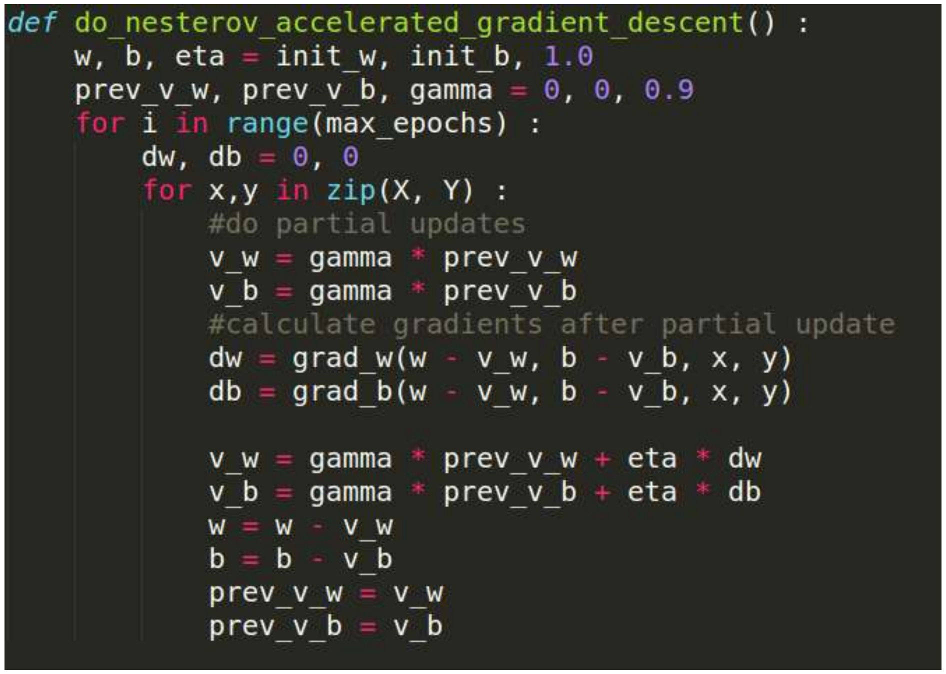
```python
def do_momentum_gradient_descent() :
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)

        v_w = gamma * prev_v_w + eta* dw
        v_b = gamma * prev_v_b + eta* db
        w = w - v_w
        b = b - v_b
        prev_v_w = v_w
        prev_v_b = v_b
```



```python
def do_stochastic_momentum_gradient_descent() :
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    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)

            v_w = gamma * prev_v_w + eta* dw
            v_b = gamma * prev_v_b + eta* db
            w = w - v_w
            b = b - v_b
            prev_v_w = v_w
            prev_v_b = v_b
```

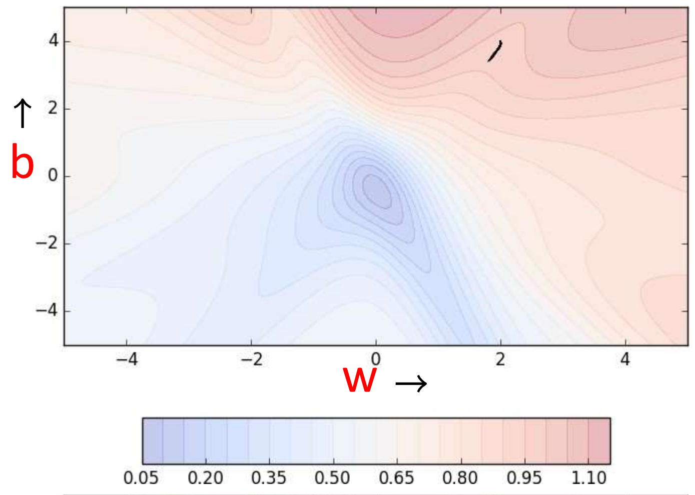


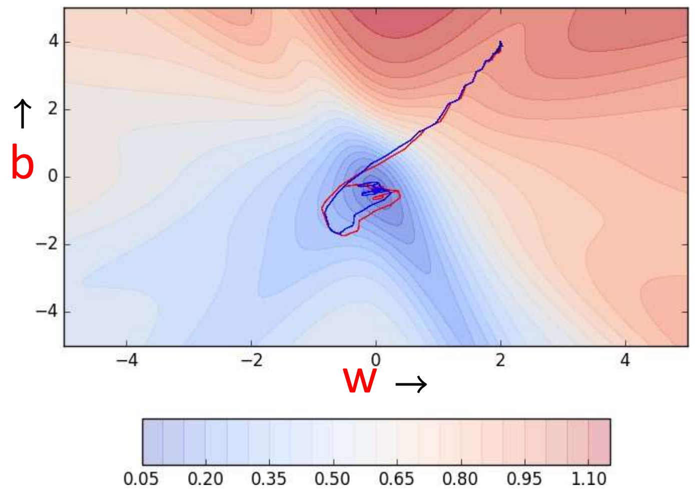
```
def do_nesterov_accelerated_gradient_descent() :

    w, b, eta = init_w, init_b  , 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs) :
        dw, db = 0, 0
        #do partial updates
        v_w = gamma * prev_v_w
        v_b = gamma * prev_v_b
        for x,y in zip(X, Y) :
            #calculate gradients after partial update
            dw += grad_w(w - v_w, b - v_b, x, y)
            db += grad_b(w - v_w, b - v_b, x, y)

        #now do the full update
        v_w = gamma * prev_v_w + eta * dw
        v_b = gamma * prev_v_b + eta * db
        w = w - v_w
        b = b - v_b
        prev_v_w = v_w
        prev_v_b = v_b
```



```
def do_nesterov_accelerated_gradient_descent() :
    w, b, eta = init_w, init_b, 1.0
    prev_v_w, prev_v_b, gamma = 0, 0, 0.9
    for i in range(max_epochs) :
        dw, db = 0, 0
        for x,y in zip(X, Y) :
            #do partial updates
            v_w = gamma * prev_v_w
            v_b = gamma * prev_v_b
            #calculate gradients after partial update
            dw = grad_w(w - v_w, b - v_b, x, y)
            db = grad_b(w - v_w, b - v_b, x, y)

            v_w = gamma * prev_v_w + eta * dw
            v_b = gamma * prev_v_b + eta * db
            w = w - v_w
            b = b - v_b
            prev_v_w = v_w
            prev_v_b = v_b
```

- While the stochastic versions of both Momentum [red] and NAG [blue] exhibit oscillations the relative advantage of NAG over Momentum still holds (i.e., NAG takes relatively shorter u-turns)
- Further both of them are faster than stochastic gradient descent (after 60 steps, stochastic gradient descent [black - top figure] still exhibits a very high error whereas NAG and Momentum are close to convergence)





Tips for Adjusting learning Rate and Momentum

Before moving on to advanced optimization algorithms let us revisit the problem of learning rate in gradient descent

- One could argue that we could have solved the problem of navigating gentle slopes by setting the learning rate high (i.e., blow up the small gradient by multiplying it with a large η)
- Let us see what happens if we set the learning rate to 10
- On the regions which have a steep slope, the already large gradient blows up further
- It would be good to have a learning rate which could adjust to the gradient ... we will see a few such algorithms soon

includegraphics[scale=0.38]images/mo

Tips for initial learning rate ?

- Tune learning rate [Try different values on a log scale: 0.0001, 0.001, 0.01, 0.1. 1.0]
- Run a few epochs with each of these and figure out a learning rate which works best
- Now do a finer search around this value [for example, if the best learning rate was 0.1 then now try some values around it: 0.05, 0.2, 0.3]
- Disclaimer: these are just heuristics ... no clear winner strategy

Tips for annealing learning rate

- **Step Decay:**
  - Halve the learning rate after every 5 epochs or
  - Halve the learning rate after an epoch if the validation error is more than what it was at the end of the previous epoch
- **Exponential Decay:** $\eta = \eta_0^{-kt}$ where η_0 and k are hyperparameters and t is the step number
- **1/t Decay:** $\eta = \frac{\eta_0}{1+kt}$ where η_0 and k are hyperparameters and t is the step number

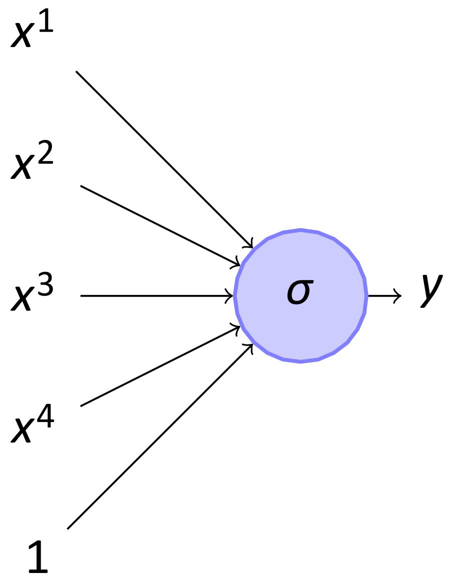
Tips for momentum

- The following schedule was suggested by Sutskever *et. al.*, 2013

$$\gamma_t = min(1 - 2^{-1-log_2(\lfloor t/250 \rfloor+1)}, \gamma_{max})$$

where, γ_{max} was chosen from {0.999, 0.995, 0.99, 0.9, 0}

Gradient Descent with Adaptive Learning Rate



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

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

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

- Given this network, it should be easy to see that given a single point ($\mathbf{x}$, y)...
- $\nabla w^1 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^1$
- $\nabla w^2 = (f(\mathbf{x}) - y) * f(\mathbf{x}) * (1 - f(\mathbf{x})) * x^2$ ... so on
- If there are n points, we can just sum the gradients over all the n points to get the total gradient
- What happens if the feature x^2 is very sparse? (*i.e.*, if its value is 0 for most inputs)
- ∇w^2 will be 0 for most inputs (see formula) and hence w^2 will not get enough updates
- If x^2 happens to be sparse as well as important we would want to take the updates to w^2 more seriously
- Can we have a different learning rate for each parameter which takes care of the frequency of features ?

Intuition

- Decay the learning rate for parameters in proportion to their update history (more updates means more decay)

Update rule for Adagrad

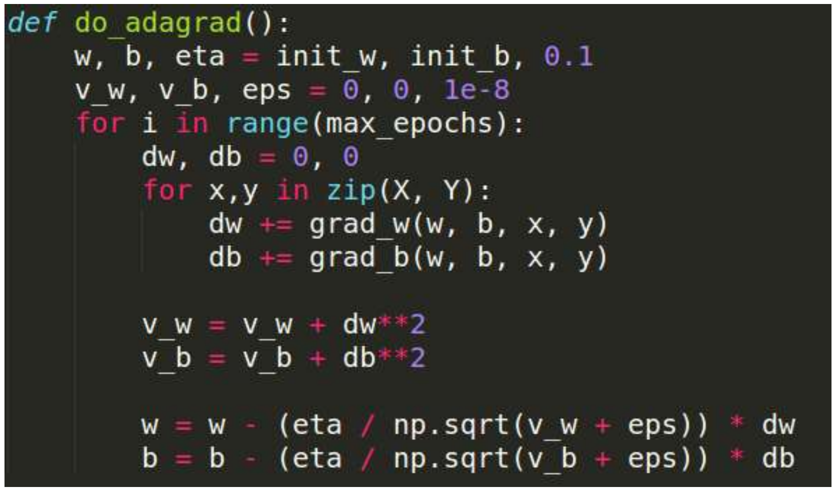
$$v_t = v_{t-1} + (\nabla w_t)^2$$

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

... and a similar set of equations for b_t

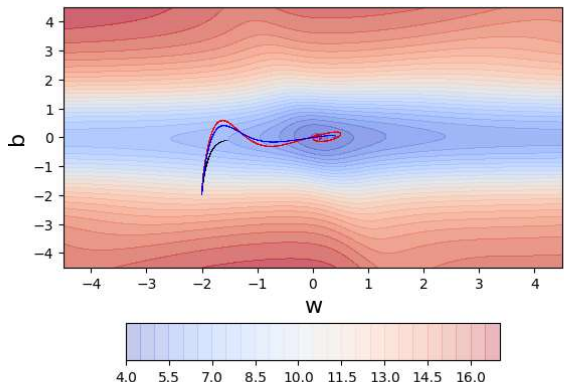
It accumulates the squared gradients of each parameter and scales the learning rate inversely proportional to the square root of this accumulated value.

- To see this in action we need to first create some data where one of the features is sparse
- How would we do this in our toy network ? Take some time to think about it
- Well, our network has just two parameters (w and b). Of these, the input/feature corresponding to b is always on (so can't really make it sparse)
- The only option is to make x sparse
- **Solution:** We created 100 random (x, y) pairs and then for roughly 80% of these pairs we set x to 0 thereby, making the feature for w sparse



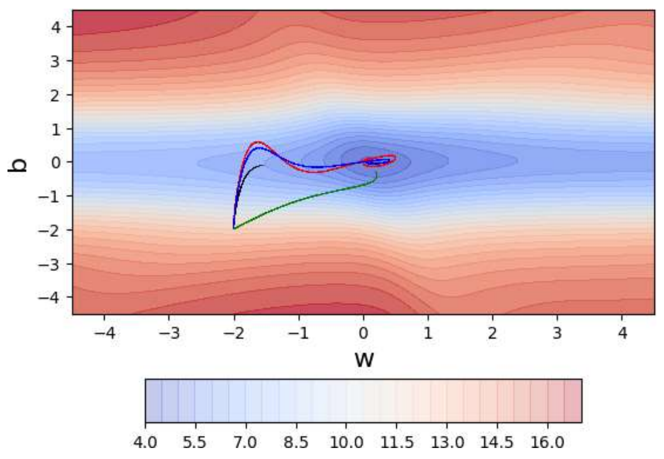
```
def do_adagrad():
    w, b, eta = init_w, init_b, 0.1
    v_w, v_b, eps = 0, 0, 1e-8
    for i in range(max_epochs):
        dw, db = 0, 0
        for x,y in zip(X, Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)

        v_w = v_w + dw**2
        v_b = v_b + db**2

        w = w - (eta / np.sqrt(v_w + eps)) * dw
        b = b - (eta / np.sqrt(v_b + eps)) * db
```

- GD (black), momentum (red) and NAG (blue)
- There is something interesting that these 3 algorithms are doing for this dataset. Can you spot it?
- Initially, all three algorithms are moving mainly along the vertical (b) axis and there is very little movement along the horizontal (w) axis
- Why? Because in our data, the feature corresponding to w is sparse and hence w undergoes very few updates ...on the other hand b is very dense and undergoes many updates
- Such sparsity is very common in large neural networks containing 1000s of input features and hence we need to address it



- Let's see what Adagrad does....

- By using a parameter specific learning rate it ensures that despite sparsity w gets a higher learning rate and hence larger updates
- Further, it also ensures that if b undergoes a lot of updates its effective learning rate decreases because of the growing denominator
- In practice, this does not work so well if we remove the square root from the denominator (something to ponder about)
- What's the flipside? over time the effective learning rate for b will decay to an extent that there will be no further updates to b
- Can we avoid this?



Intuition

- Adagrad decays the learning rate very aggressively (as the denominator grows)
- As a result after a while the frequent parameters will start receiving very small updates because of the decayed learning rate
- To avoid this why not decay the denominator and prevent its rapid growth

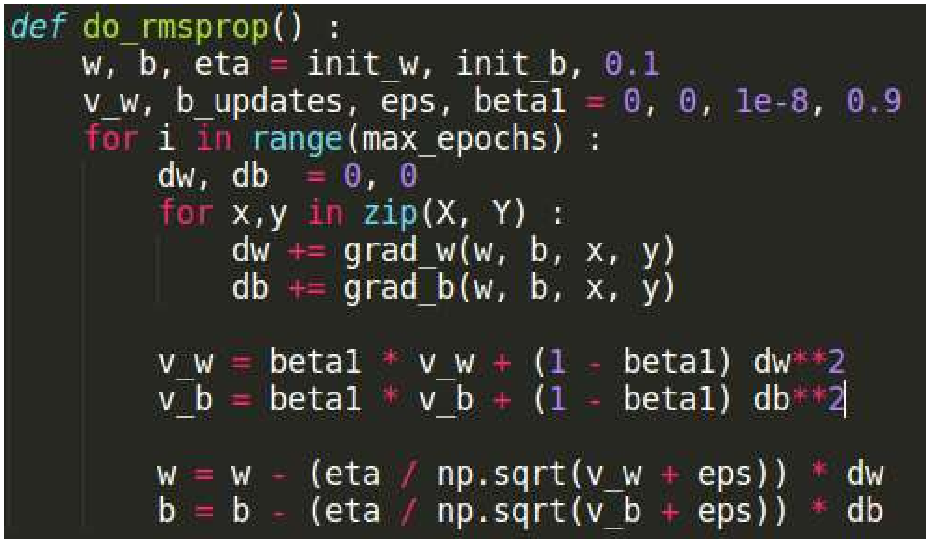
Update rule for RMSProp

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

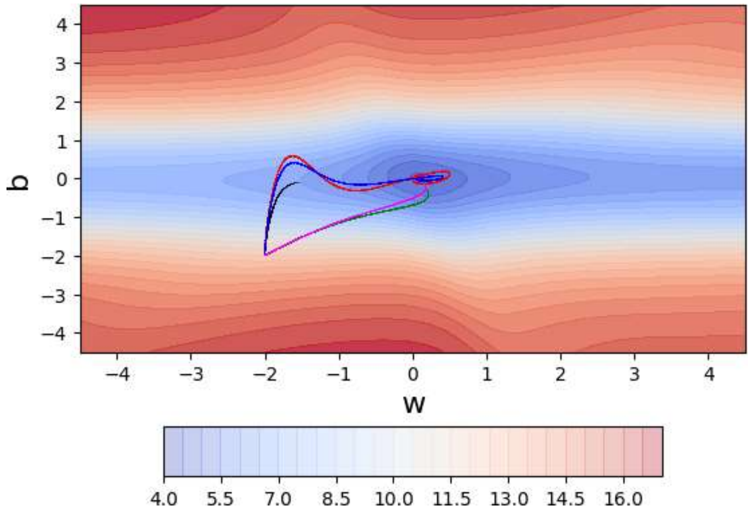
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t + \epsilon}} * \nabla w_t$$

... and a similar set of equations for b_t

decayed average of past squared gradients, ensuring that frequently updated parameters have smaller learning rates and less frequently updated ones have larger learning rates



```
def do_rmsprop() :
    w, b, eta = init_w, init_b, 0.1
    v_w, b_updates, eps, beta1 = 0, 0, 1e-8, 0.9
    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)

        v_w = beta1 * v_w + (1 - beta1) dw**2
        v_b = beta1 * v_b + (1 - beta1) db**2

        w = w - (eta / np.sqrt(v_w + eps)) * dw
        b = b - (eta / np.sqrt(v_b + eps)) * db
```



- 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

- Do everything that RMSProp does to solve the decay problem of Adagrad
- Plus use a cumulative history of the gradients
- In practice, $\beta_1 = 0.9$ and $\beta_2 = 0.999$

Update rule for Adam

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

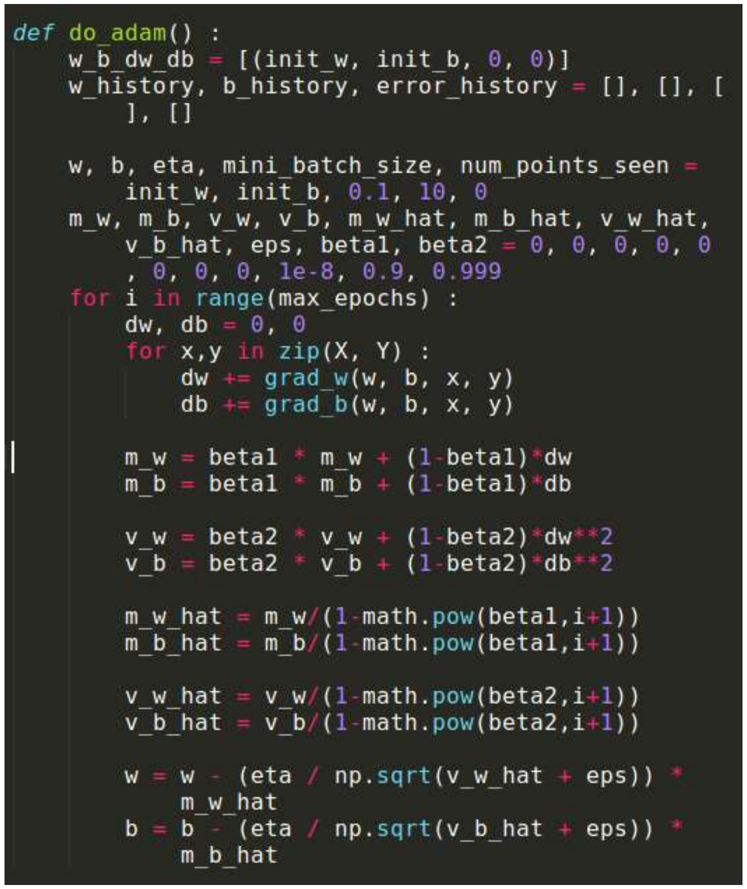
$$v_t = \beta_2 * v_{t-1} + (1-\beta_2) * (\nabla w_t)^2$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v_t} = \frac{v_t}{1-\beta_2^t}$$

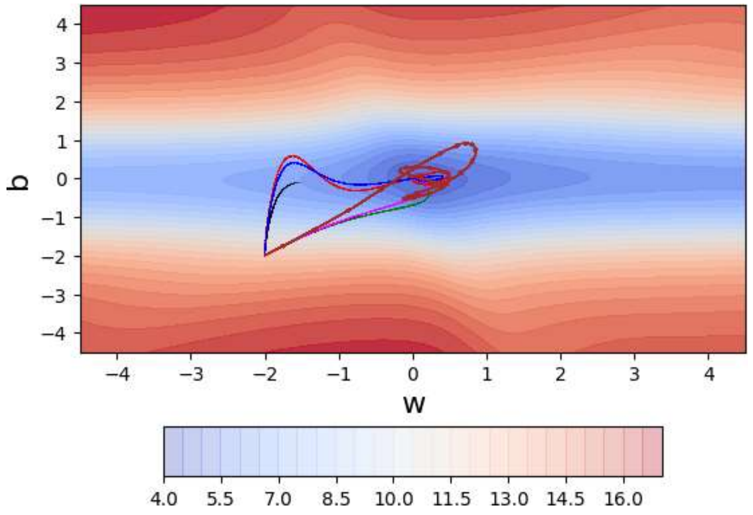
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v_t} + \epsilon}} * \hat{m}_t$$

... and a similar set of equations for b_t

Adagrad accumulates the squared gradients of each parameter. Parameters with larger accumulated gradients get smaller updates (smaller learning rates), while parameters with smaller accumulated gradients get larger updates (larger learning rates).



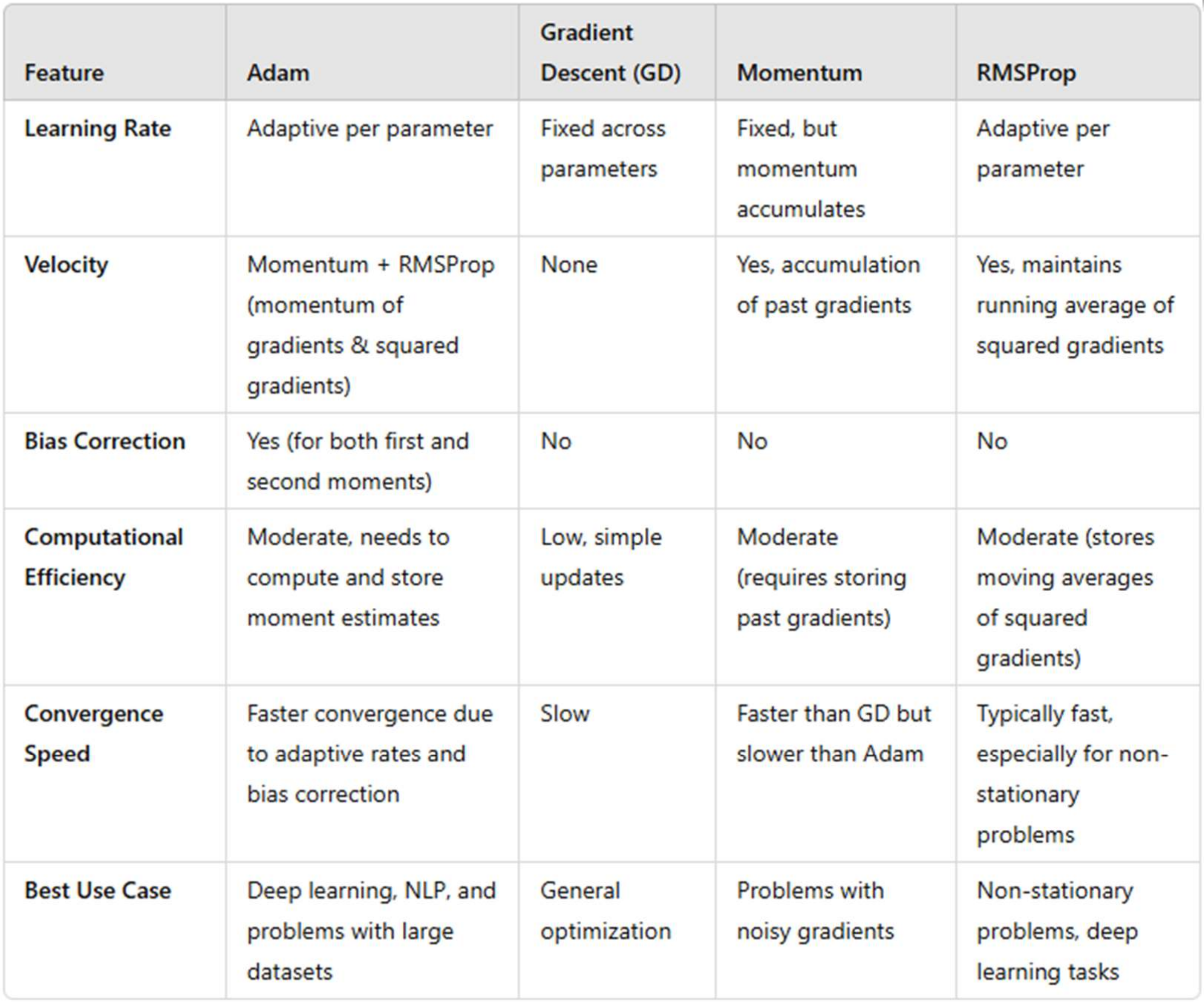
```
def do_adam() :
    w_b_dw_db = [(init_w, init_b, 0, 0)]
    w_history, b_history, error_history = [], [], [
        ], []

    w, b, eta, mini_batch_size, num_points_seen =
        init_w, init_b, 0.1, 10, 0
    m_w, m_b, v_w, v_b, m_w_hat, m_b_hat, v_w_hat,
        v_b_hat, eps, beta1, beta2 = 0, 0, 0, 0, 0
        , 0, 0, 0, 1e-8, 0.9, 0.999
    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)

        m_w = beta1 * m_w + (1-beta1)*dw
        m_b = beta1 * m_b + (1-beta1)*db

        v_w = beta2 * v_w + (1-beta2)*dw**2
        v_b = beta2 * v_b + (1-beta2)*db**2

        m_w_hat = m_w/(1-math.pow(beta1,i+1))
        m_b_hat = m_b/(1-math.pow(beta1,i+1))

        v_w_hat = v_w/(1-math.pow(beta2,i+1))
        v_b_hat = v_b/(1-math.pow(beta2,i+1))

        w = w - (eta / np.sqrt(v_w_hat + eps)) *
            m_w_hat
        b = b - (eta / np.sqrt(v_b_hat + eps)) *
            m_b_hat
```



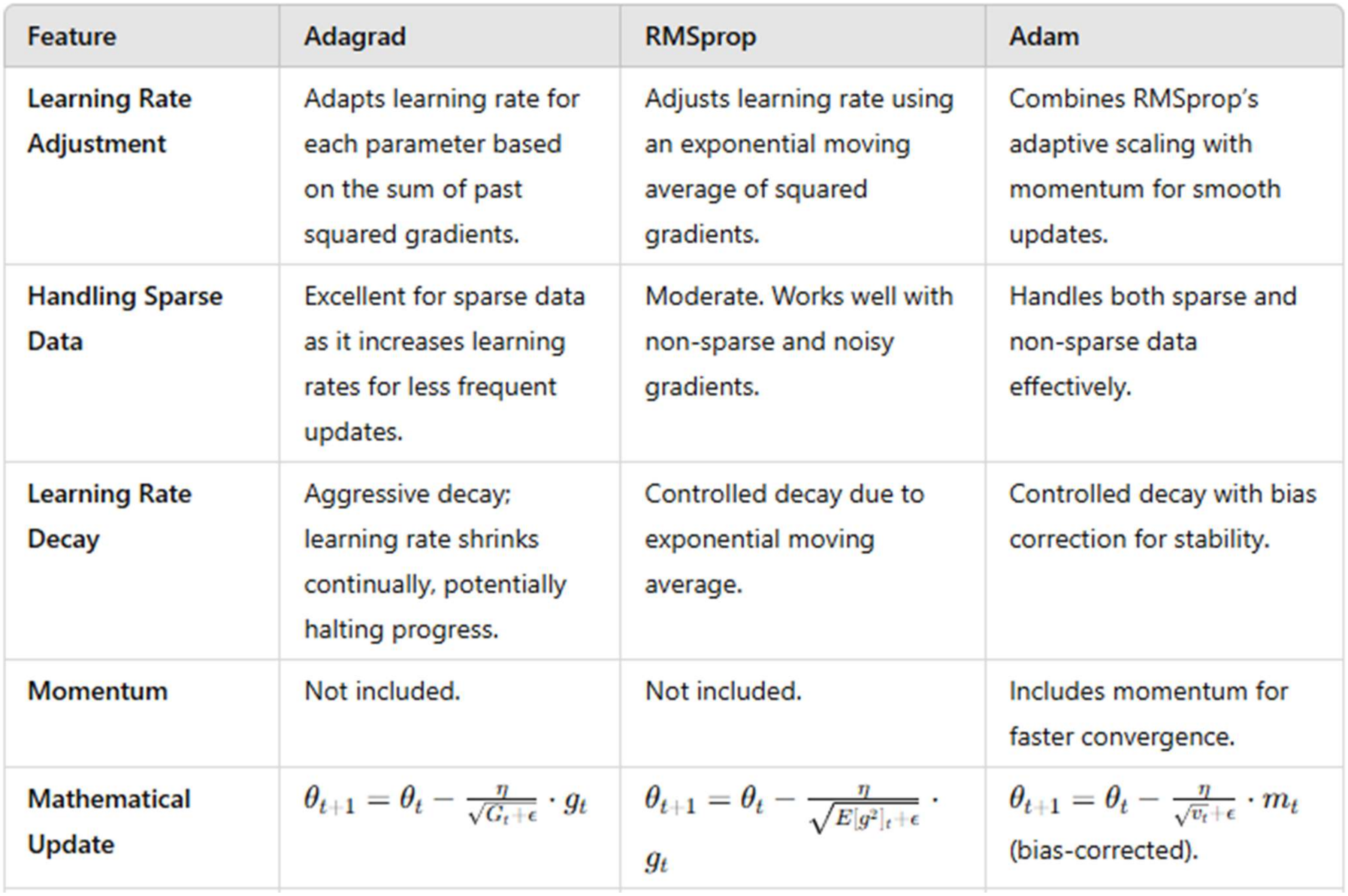
- As expected, taking a cumulative history gives a speed up ...

Million dollar question: Which algorithm to use in practice

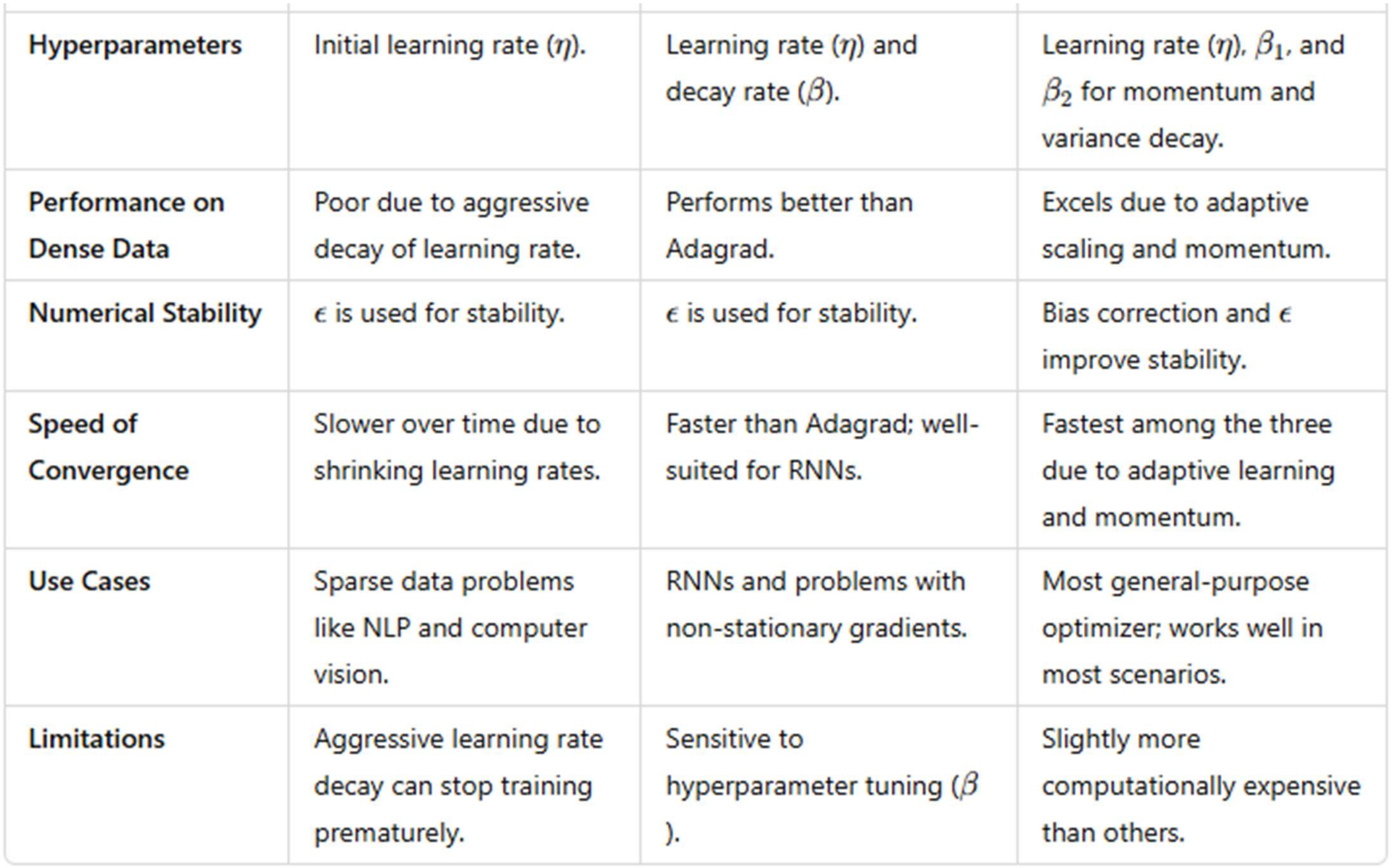
- Adam seems to be more or less the default choice now ($\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 1e-8$)
- Although it is supposed to be robust to initial learning rates, we have observed that for sequence generation problems $\eta = 0.001, 0.0001$ works best
- Having said that, many papers report that SGD with momentum (Nesterov or classical) with a simple annealing learning rate schedule also works well in practice (typically, starting with $\eta = 0.001, 0.0001$ for sequence generation problems)
- Adam might just be the best choice overall!!
- Some recent work suggest that there is a problem with Adam and it will not converge in some cases



Feature	Adam	Gradient Descent (GD)	Momentum	RMSProp
Learning Rate	Adaptive per parameter	Fixed across parameters	Fixed, but momentum accumulates	Adaptive per parameter
Velocity	Momentum + RMSProp (momentum of gradients & squared gradients)	None	Yes, accumulation of past gradients	Yes, maintains running average of squared gradients
Bias Correction	Yes (for both first and second moments)	No	No	No
Computational Efficiency	Moderate, needs to compute and store moment estimates	Low, simple updates	Moderate (requires storing past gradients)	Moderate (stores moving averages of squared gradients)
Convergence Speed	Faster convergence due to adaptive rates and bias correction	Slow	Faster than GD but slower than Adam	Typically fast, especially for non-stationary problems
Best Use Case	Deep learning, NLP, and problems with large datasets	General optimization	Problems with noisy gradients	Non-stationary problems, deep learning tasks



Feature	Adagrad	RMSprop	Adam
Learning Rate Adjustment	Adapts learning rate for each parameter based on the sum of past squared gradients.	Adjusts learning rate using an exponential moving average of squared gradients.	Combines RMSprop's adaptive scaling with momentum for smooth updates.
Handling Sparse Data	Excellent for sparse data as it increases learning rates for less frequent updates.	Moderate. Works well with non-sparse and noisy gradients.	Handles both sparse and non-sparse data effectively.
Learning Rate Decay	Aggressive decay; learning rate shrinks continually, potentially halting progress.	Controlled decay due to exponential moving average.	Controlled decay with bias correction for stability.
Momentum	Not included.	Not included.	Includes momentum for faster convergence.
Mathematical Update	$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \cdot g_t$	$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{E[g^2]_t + \epsilon}} \cdot g_t$	$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t} + \epsilon} \cdot m_t$ (bias-corrected).



Hyperparameters	Initial learning rate (η).	Learning rate (η) and decay rate (β).	Learning rate (η), β_1, and β_2 for momentum and variance decay.
Performance on Dense Data	Poor due to aggressive decay of learning rate.	Performs better than Adagrad.	Excels due to adaptive scaling and momentum.
Numerical Stability	ϵ is used for stability.	ϵ is used for stability.	Bias correction and ϵ improve stability.
Speed of Convergence	Slower over time due to shrinking learning rates.	Faster than Adagrad; well-suited for RNNs.	Fastest among the three due to adaptive learning and momentum.
Use Cases	Sparse data problems like NLP and computer vision.	RNNs and problems with non-stationary gradients.	Most general-purpose optimizer; works well in most scenarios.
Limitations	Aggressive learning rate decay can stop training prematurely.	Sensitive to hyperparameter tuning (β).	Slightly more computationally expensive than others.

Explanation for why we need bias correction in Adam

Update rule for Adam

$$m_t = \beta_1 * m_{t-1} + (1-\beta_1) * \nabla w_t$$
$$v_t = \beta_2 * v_{t-1} + (1-\beta_2) * (\nabla w_t)^2$$
$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$
$$\hat{v_t} = \frac{v_t}{1-\beta_2^t}$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{\hat{v_t} + \epsilon}} * \hat{m}_t$$

- Note that we are taking a running average of the gradients as m_t
- The reason we are doing this is that we don't want to rely too much on the current gradient and instead rely on the overall behaviour of the gradients over many timesteps
- One way of looking at this is that we are interested in the expected value of the gradients and not on a single point estimate computed at time t
- However, instead of computing $E[\nabla w_t]$ we are computing m_t as the exponentially moving average
- Ideally we would want $E[m_t]$ to be equal to $E[\nabla w_t]$
- Let us see if that is the case

- For convenience we will denote ∇w_t as g_t and β_1 as β

$$\begin{aligned}
m_t &= \beta * m_{t-1} + (1-\beta) * g_t \\
m_0 &= 0 \\
m_1 &= \beta m_0 + (1-\beta)g_1 \\
&= (1-\beta)g_1 \\
m_2 &= \beta m_1 + (1-\beta)g_2 \\
&= \beta(1-\beta)g_1 + (1-\beta)g_2 \\
m_3 &= \beta m_2 + (1-\beta)g_3 \\
&= \beta(\beta(1-\beta)g_1 + (1-\beta)g_2) + (1-\beta)g_3 \\
&= \beta^2(1-\beta)g_1 + \beta(1-\beta)g_2 + (1-\beta)g_3 \\
&= (1-\beta)\sum_{i=1}^{3} \beta^{3-i} g_i
\end{aligned}$$

- In general,

$$m_t = (1-\beta)\sum_{i=1}^{t} \beta^{t-i} g_i$$

- So we have, $m_t = (1-\beta)\sum_{i=1}^{t} \beta^{t-i} g_i$
- Taking Expectation on both sides

$$E[m_t] = E[(1-\beta)\sum_{i=1}^{t} \beta^{t-i} g_i]$$

$$E[m_t] = (1-\beta)E[\sum_{i=1}^{t} \beta^{t-i} g_i]$$

$$E[m_t] = (1-\beta)\sum_{i=1}^{t} E[\beta^{t-i} g_i]$$

$$= (1-\beta)\sum_{i=1}^{t} \beta^{t-i} E[g_i]$$

- Assumption: All g_i's come from the same distribution i.e. $E[g_i] = E[g]\ \forall i$

$$E[m_t] = (1-\beta)\sum_{i=1}^{t} (\beta)^{t-i} E[g_i]$$

$$= E[g](1-\beta)\sum_{i=1}^{t} (\beta)^{t-i}$$

$$= E[g](1-\beta)(\beta^{t-1} + \beta^{t-2} + \cdots + \beta^{0})$$

$$= E[g](1-\beta)\frac{1-\beta^{t}}{1-\beta}$$

the last fraction is the sum of a GP with common ratio $= \beta$

$$E[m_t] = E[g](1-\beta^t)$$

$$E[\frac{m_t}{1-\beta^t}] = E[g]$$

$$E[\hat{m}_t] = E[g](\because \frac{m_t}{1-\beta^t} = \hat{m}_t)$$

Hence we apply the bias correction because then the expected value of $\hat{m}_t$ is the same as the expected value of g_t