PadhAI Week 6: Optimization Algorithms (Part 1)

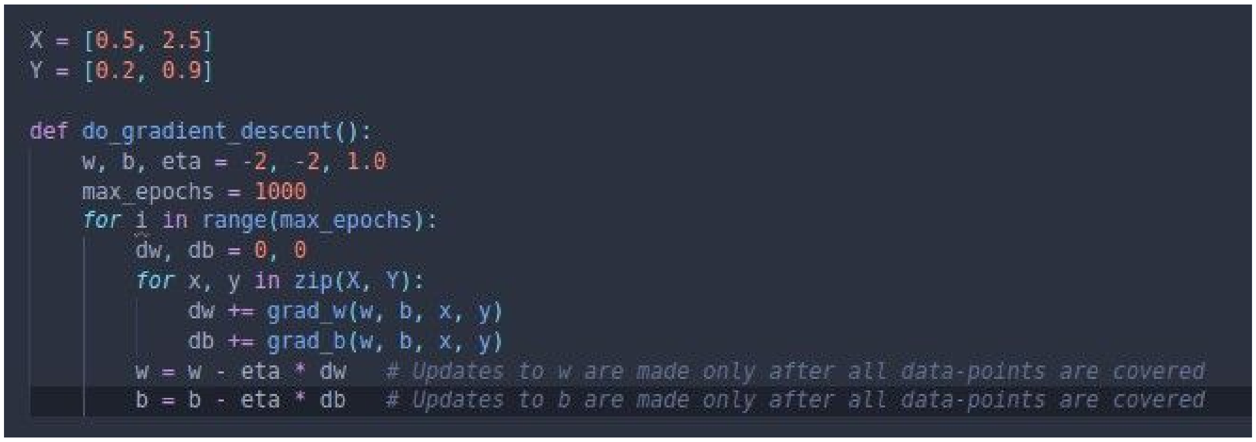
by Manick Vennimalai

6.2: Optimization Algorithms: Part 2

6.2.1: The idea of stochastic and mini-batch gradient descent

How many updates are we making?

1. Let us consider vanilla gradient descent



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

def do_gradient_descent():
    w, b, eta = -2, -2, 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   # Updates to w are made only after all data-points are covered
        b = b - eta * db   # Updates to b are made only after all data-points are covered
```

2. From the above image, we can see that we make one update(w,b) for one pass/epoch over the data
3. It can be exemplified as follows
    a. Consider a training set with 1 million data points
    b. With Gradient Descent, we calculate the derivatives for each of these points
    c. Once we're done, we update the parameters
    d. Thus, we pass over all 1 million points to make a single update to w & b
    e. It can also be called **batch gradient descent**, as the entire dataset is used as a single batch
4. However, we can choose to make an approximation based on looking at a smaller portion(batch) of the data points instead of analysing the whole dataset each time.
5. This is called **mini-batch gradient descent** and can be described as follows
    a. Consider a training set of 1 million data points
    b. Select a batch size of 100 data points
    c. What this means is that, every batch, the algorithm calculates all of the 100 derivatives and updates the parameters
    d. Thus, passing over all 1 million data points results in 10000 updates to w & b.
6. **Stochastic gradient descent** is when the batch size is 1, i.e. an update to the parameters after each single data point
7. One key thing to note is that both stochastic and mini-batch gradient descent are approximations of the true derivative obtained by batch gradient descent.
8. However it is advantageous as it allows is to make updates faster and achieve quicker progress.

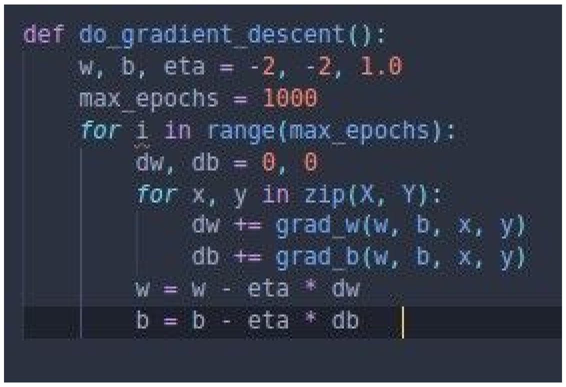
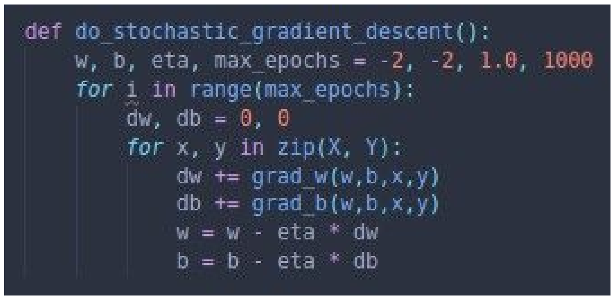
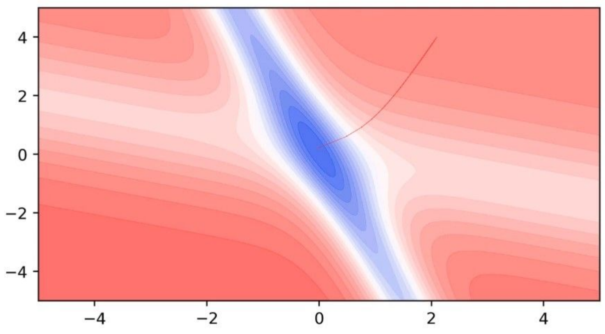
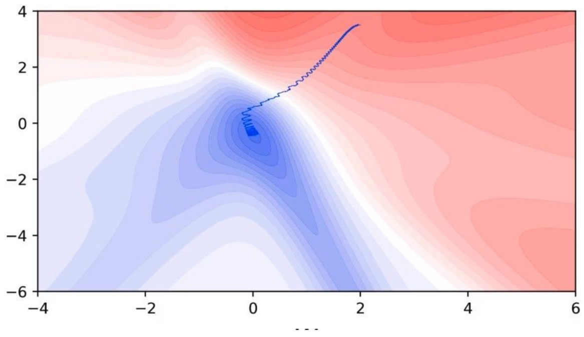
PadhAI Week 6: Optimization Algorithms (Part 1)

by Manick Vennimalai

6.2.1a: Running stochastic gradient descent

Can we make stochastic updates

1. Let's do a side by side comparison of batch GD and stochastic GD

Batch GD	Stochastic GD
	
	

```python
def do_gradient_descent():
    w, b, eta = -2, -2, 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 * db
```

```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
```

2. Some of the advantages of Stochastic GD are
   a. Quicker updates
   b. Many updates in one pass of the data
3. Some of the disadvantages of Stochastic GD are
   a. Approximate(stochastic) gradient
   b. Almost like tossing a coin once and computing P(heads)
4. From the Gradient descent visualization, we can see that it oscillates during movement. However, this oscillation is different from Momentum GD or NAG.
5. In stochastic GD, the oscillations are due to redirection after every point, as every point behaves as an individual greedy entity influencing w & b, thus leading to fluctuations right from the start.
6. In MGD or NAG, the oscillations appear the value approaches the minima as a result of overshooting the intended destination.

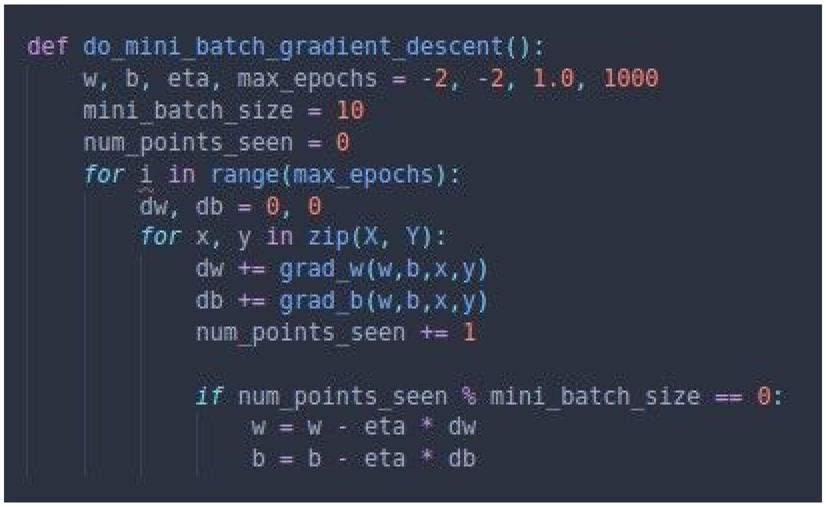
PadhAI Week 6: Optimization Algorithms (Part 1)

by Manick Vennimalai

6.2.1b: Running mini-batch gradient descent

Doesn't it make sense to use more than one point or a mini-batch of points?

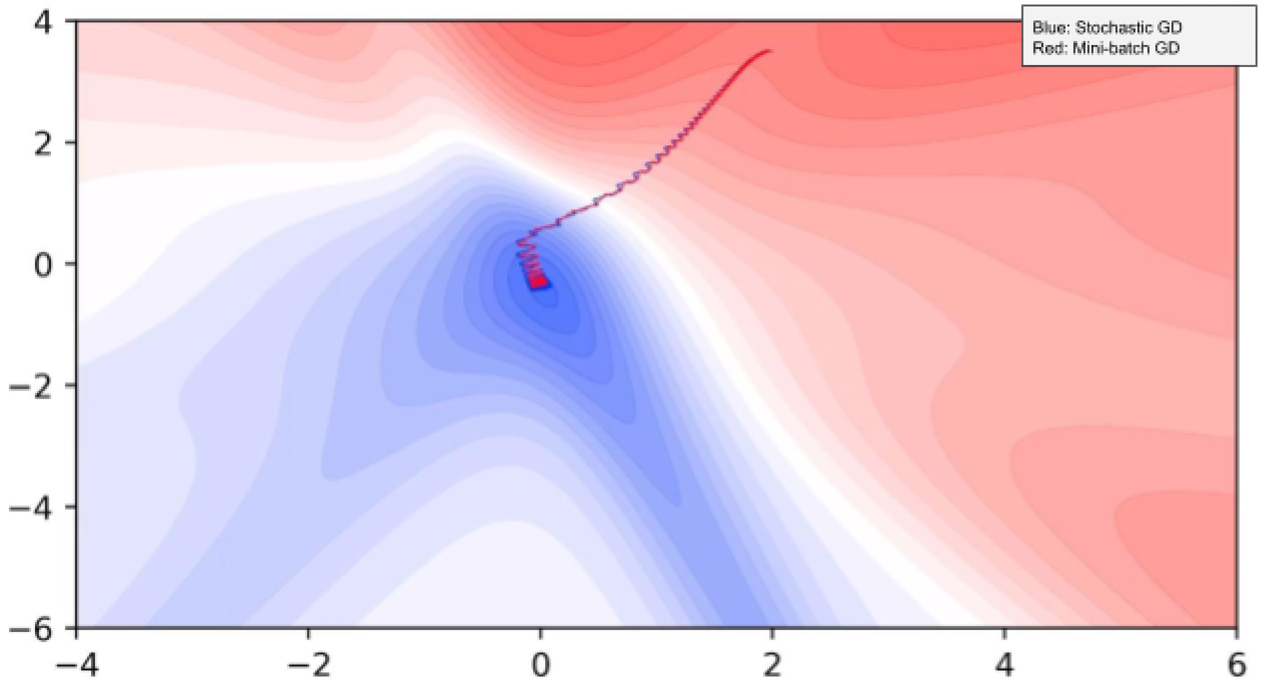
1. Let's look at the python implementation of mini-batch GD



```python
def do_mini_batch_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    mini_batch_size = 10
    num_points_seen = 0
    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)
            num_points_seen += 1

            if num_points_seen % mini_batch_size == 0:
                w = w - eta * dw
                b = b - eta * db
```

2. Now let us look at the 2D visualisation of mini-batch superimposed over stochastic GD



3. Here, we can observe that even though the plot oscillates for mini-batch GD, it is still considerably less than with stochastic GD, evidenced by the red plot lying entirely within the blue plot.
4. As we increase the batch size, the stability of the curve also improves, resulting in better estimates of the gradient
5. Recommended batch size is 32, 64, 128 etc.
6. The higher the batch size (k), the more accurate the estimates.

PadhAI Week 6: Optimization Algorithms (Part 1)

by Manick Vennimalai

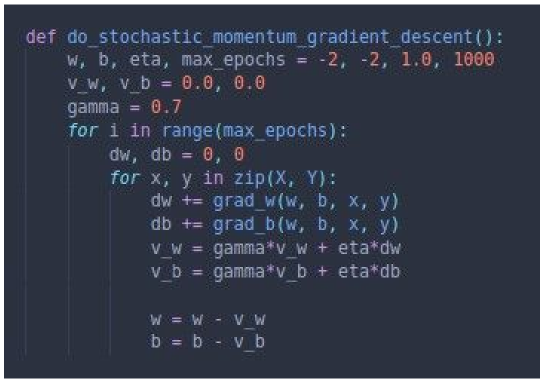
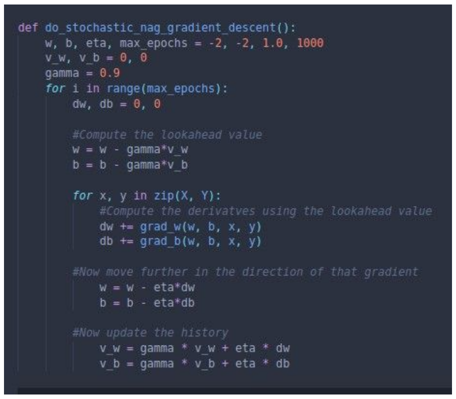
6.2.2: Epochs and Steps

What is an epoch and what is a step?

1. Let us go over the definitions of an epoch and a step
   a. 1 epoch = one pass over the entire data
   b. 1 step = one update of the parameters
   c. N = number of data points
   d. B = mini-batch size
2. Let's analyse the algorithms using epochs and steps

Algorithm	Number of steps in one epoch
Batch Gradient Descent	1
Stochastic Gradient Descent	N
Mini-Batch Gradient Descent	N/B

3. Let's look at stochastic version of NAG and Momentum based GD

Stochastic Momentum GD	Stochastic NAG
	

```
def do_stochastic_momentum_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    v_w, v_b = 0.0, 0.0
    gamma = 0.7
    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*v_w + eta*dw
            v_b = gamma*v_b + eta*db

            w = w - v_w
            b = b - v_b
```

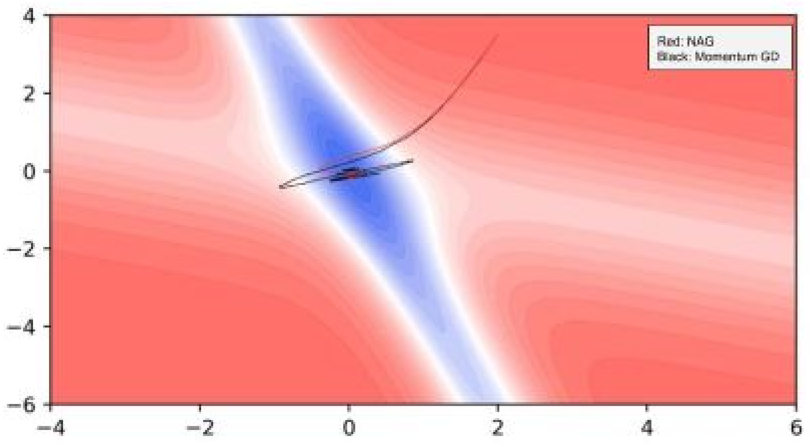
```
def do_stochastic_nag_gradient_descent():
    w, b, eta, max_epochs = -2, -2, 1.0, 1000
    v_w, v_b = 0, 0
    gamma = 0.9
    for i in range(max_epochs):
        dw, db = 0, 0

        #Compute the lookahead value
        w = w - gamma*v_w
        b = b - gamma*v_b

        for x, y in zip(X, Y):
            #Compute the derivatves using the lookahead value
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)

        #Now move further in the direction of that gradient
            w = w - eta*dw
            b = b - eta*db

        #Now update the history
            v_w = gamma * v_w + eta * dw
            v_b = gamma * v_b + eta * db
```



4. Since there is a history component, NAG and Momentum GD have slightly smoother oscillations.

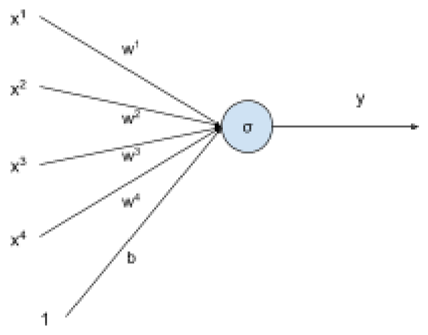
PadhAI Week 6: Optimization Algorithms (Part 1)

by Manick Vennimalai

6.2.3: Why do we need an adaptive learning rate?

Why do we need an adaptive learning rate for every feature?

1. Consider input data with 4 features being processed through a sigmoid neuron



2. Here $y = f(x) = \frac{1}{1+e^{-(w.x+b)}}$
   a. $x = \{x^1, x^2, x^3, x^4\}$
   b. $w = \{w^1, w^2, w^3, w^4\}$
3. From our gradient formula, we know that the value of the input feature plays a role in the gradient calculation i.e. $\nabla w^n = (f(x) - y) * f(x) * (1 - f(x)) * x^n$
4. In real world scenarios, many features in the data are **sparse**, i.e. they take on a 0 value for most of the training inputs. Therefore, the derivatives corresponding to these 0 valued points are also 0, and the weight update is going to be 0.
5. To aid these sparse features, a larger learning rate can be applied to the **non-zero valued points** of these sparse features.
6. Example:
   a. Consider a subject at college that you are taught for only 5 minutes a day
   b. For those 5 minutes, maximising your attention span would allow for maximum knowledge retention
   c. In this case, the 5-minute subject would be a sparse feature i.e. a feature that does not occur very often in the training data
   d. And the attention span would be our learning rate. A high learning rate for the sparse features allows us to maximise the learning (weight updation) we get from it.
7. Conversely, **dense** features are those with non-zero values for most of the data points. They must be dealt with by using a lower learning rate.
8. Can we have a different learning rate for each parameter(weights) which takes care of the frequency(sparsity/density) of features?

6.2.4: Introducing Adagrad

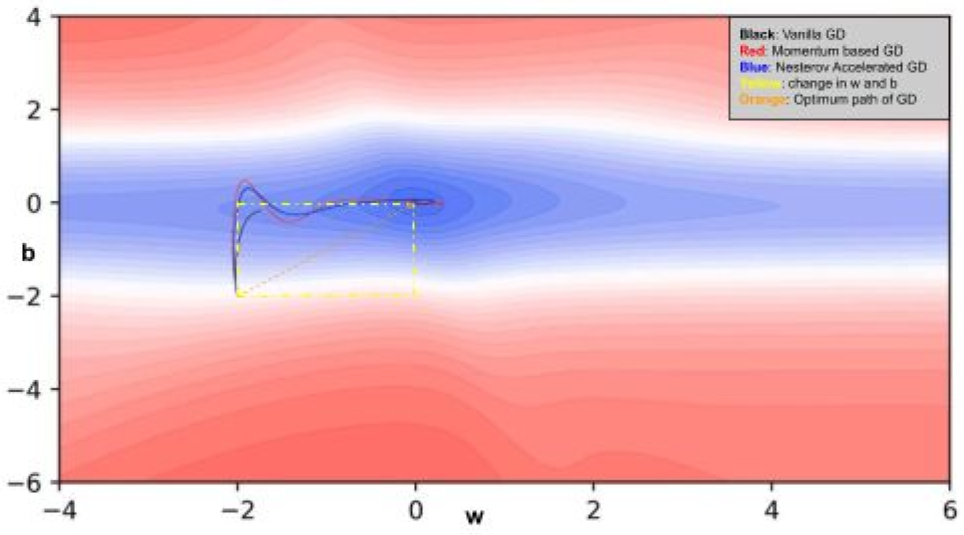
How do we convert the adaptive learning rate intuition into an equation?

1. **Intuition**: Decay the learning rate for parameters in proportion to their update history (fewer updates, lesser decay)
2. The Adagrad (Adaptive Gradient) is an algorithm which satisfies the above intuition
3. Adagrad
   a. $\upsilon_t = \upsilon_{t-1} + (\nabla \omega_t)^2$
      i. This value increments based on the gradient of that particular iteration, i.e. the value of the feature is non-zero.
      ii. In the case of dense features, it increments for most iterations, resulting in a larger υ_t value
      iii. For sparse features, does not increment much as the gradient value is often 0, leading to a lower υ_t value.
   b. $\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{(\upsilon_t)} + \varepsilon} \nabla \omega_t$
      i. The denominator term $\sqrt{(\upsilon_t)}$ serves to regulate the learning rate η
      ii. For dense features, υ_t is larger, $\sqrt{(\upsilon_t)}$ becomes larger thereby lowering η
      iii. For sparse features, υ_t is smaller, $\sqrt{(\upsilon_t)}$ becomes smaller and lowers η to a smaller extent.
      iv. The ε term is added to the denominator $\sqrt{(\upsilon_t)} + \varepsilon$ to **prevent** a **divide-by-zero error** from occurring in the case of very sparse features i.e. where all the data points yield zero up till the measured instance.

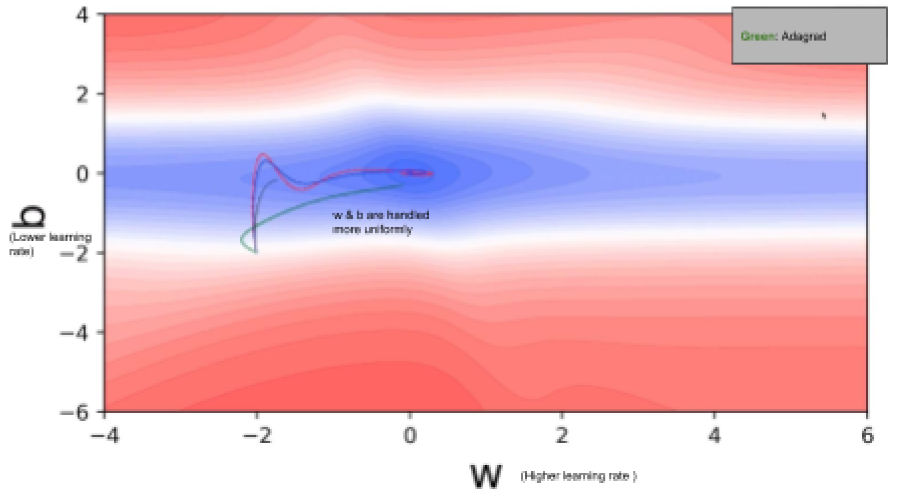
6.2.5: Running and Visualizing Adagrad

Let's compare this to vanilla, momentum based, NAG gradient descent

1. Let's plot the 2D visualisation of vanilla, momentum based, NAG gradient descent



2. Here, w & b behave as two features of the input (x_0, x_1). b is a dense feature and is always a non-zero value. w is deliberately chosen as a sparse feature with 80% of the values as 0.
3. Thus, we would need a higher learning rate for w and a lower learning rate for b, if not, we will end up with sub-optimal paths as shown by the previous 3 types of GD from the figure.
4. Let's look at a visualisation of Adagrad



6.2.6: A limitation of Adagrad

What do we observe?

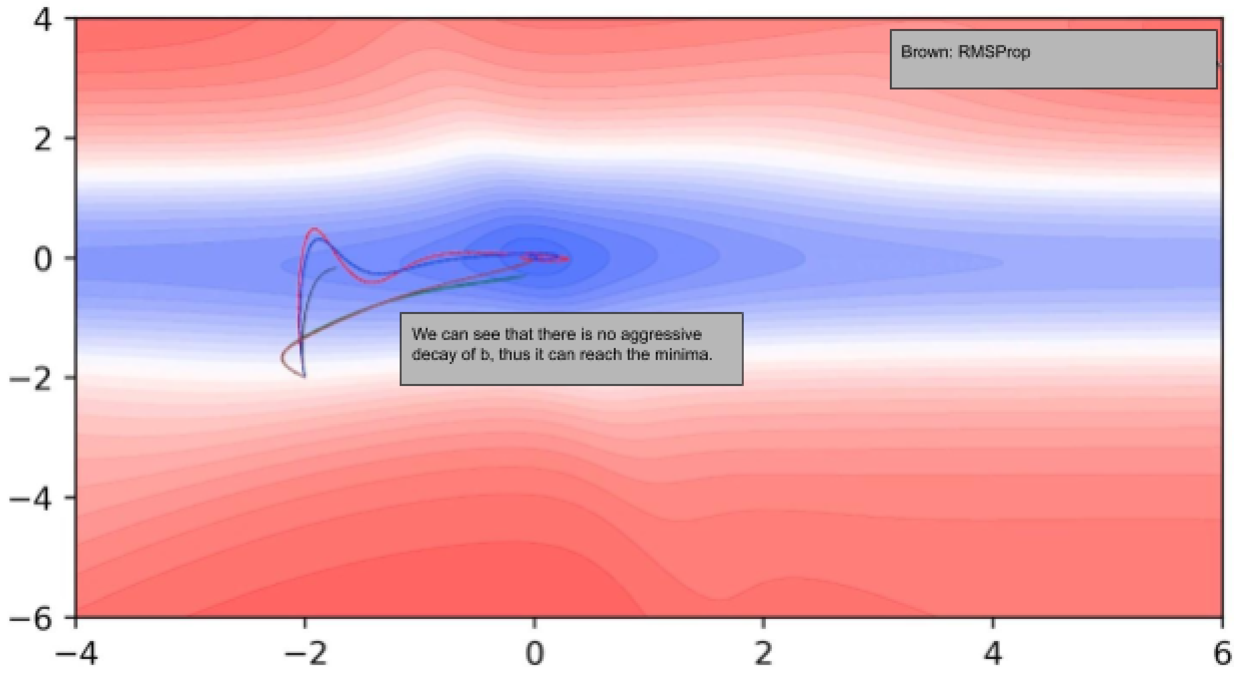
1. **Advantage**: Parameters corresponding to sparse features get better updates
2. **Disadvantage**: The learning rate decays very aggressively as the denominator grows (not good for parameters corresponding to dense features)

6.2.7: Running and Visualizing RMSProp

Can we overcome aggressively decaying denominators?

1. Intuition: Why not decay the denominator and prevent its rapid growth?
2. We can consider the RMSProp algorithm
   a. $\upsilon_t = \beta * \upsilon_{t-1} + (1-\beta)(\nabla\omega_t)^2$
      i. Here we are taking an exponentially decaying sum
      ii. Let $\beta = 0.9$ and consider the 4th iteration υ_4
      iii. $\upsilon_0 = 0$
      iv. $\upsilon_1 = 0.1(\nabla\omega_1)^2$
      v. $\upsilon_2 = (0.9)(0.1)(\nabla\omega_1)^2 + 0.1(\nabla\omega_2)^2$
      vi. $\upsilon_3 = (0.9)^2(0.1)(\nabla\omega_1)^2 + (0.9)(0.1)(\nabla\omega_2)^2 + 0.1(\nabla\omega_3)^2$
      vii. $\upsilon_4 = (0.9)^3(0.1)(\nabla\omega_1)^2 + (0.9)^2(0.1)(\nabla\omega_2)^2 + (0.9)(0.1)(\nabla\omega_3)^2 + 0.1(\nabla\omega_4)^2$
      viii. We can see from this that our value υ_4 is much smaller than in the case of Adagrad, due the history of the gradients being multiplied by the decay ratio.
      ix. The relative difference between dense and sparse features is still maintained.

   b. $\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{(\upsilon_t)} + \varepsilon}\nabla\omega_t$
      i. This is the same as in Adagrad
3. Let's visualise RMSProp in 2D



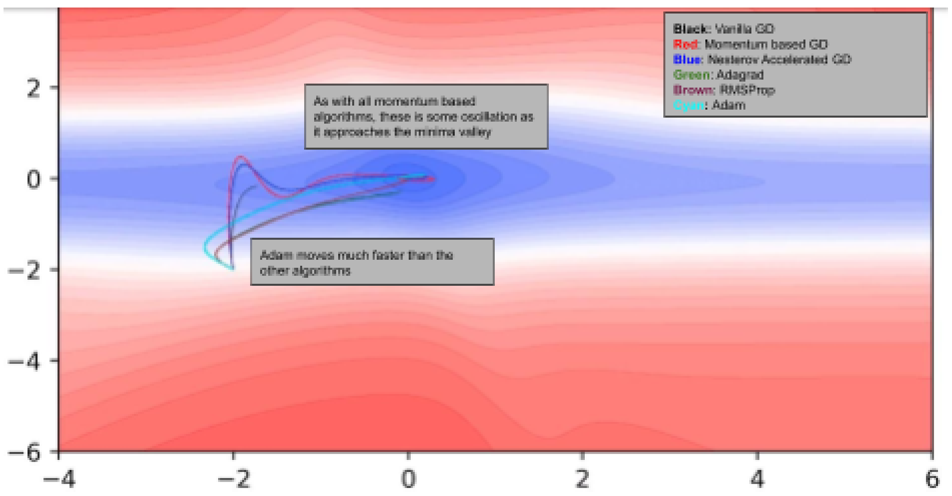
4. Adagrad got stuck when it was close to convergence (it was no longer able to move in the vertical (b) direction because of the decayed learning rate)
5. RMSProp overcomes this problem by being less aggressive on the decay

PadhAI Week 6: Optimization Algorithms (Part 1)

by Manick Vennimalai

6.2.8: Running and Visualizing Adam

Does it make sense to use a cumulative history of gradients?

1. We have already looked at a algorithms that make use of a history term
   a. Momentum based GD: Makes use of the history of the gradients
      i. $\upsilon_t = \gamma * \upsilon_{t-1} + \eta \nabla \omega_t$
      ii. $\omega_{t+1} = \omega_t - \upsilon_t$
      iii. Here, history is used to calculate the current update
   b. RMSProp: Makes use of the history of the square of the gradients
      i. $\upsilon_t = \beta * \upsilon_{t-1} + (1-\beta)(\nabla \omega_t)^2$
      ii. $\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{(\upsilon_t)} + \varepsilon} \nabla \omega_t$
      iii. Here, history is used to adjust the learning-rate
   c. Can we combine these two ideas?
   d. Yes, in the form of Adam, which uses both of those history terms
2. Adam
   a. $m_t = \beta_1 * \upsilon_{t-1} + (1-\beta_1)(\nabla \omega_t)$
      i. This is very similar to the history that Momentum based GD maintains
      ii. It's a running sum of all the updates done
   b. $\upsilon_t = \beta_2 * \upsilon_{t-1} + (1-\beta_2)(\nabla \omega_t)^2$
      i. This is similar to the history that RMSProp maintains
      ii. It is used to regulate the learning-rate
   c. $\omega_{t+1} = \omega_t - \frac{\eta}{\sqrt{(\upsilon_t)} + \varepsilon} m_t$
      i. Here, the first history m_t is used to make the update, ensuring that the history of derivatives is used to calculate the current update
      ii. The second derivative v_t is used to regulate the learning rate based on density or sparsity of the feature
   d. In addition to the above points, Adam performs bias correction by using the following equations
      i. $m_t = \frac{m_t}{1-\beta^t_1}$
      ii. $\upsilon_t = \frac{\upsilon_t}{1-\beta^t_1}$
      iii. It ensure that the training is smoother and also prevents erratic updates in beginning of training.
3. 

6.2.9: Summary

Which algorithm do we use in practice?

1. **Algorithms**:
   a. GD
   b. Momentum based GD
   c. Nesterov Accelerated GD
   d. AdaGrad
   e. RMSProp
   f. Adam
2. **Strategies**:
   a. Batch
   b. Mini-Batch (32, 64, 128)
   c. Stochastic
3. Up till this point, we have been using GD with Batch update
4. In practice, Adam with Mini-Batch is the most popular choice
5. However, GD with Stochastic update is also used along with the use of special strategies to adjust the learning rate.