1. Explain in detail about AdaGrad (Adaptive Gradient Descent) Deep Learning Optimizer

The adaptive gradient descent algorithm is slightly different from other gradient descent algorithms. This is because it uses different learning rates for each iteration. The change in learning rate depends upon the difference in the parameters during training. The more the parameters get changed, the more minor the learning rate changes. This modification is highly beneficial because real-world datasets contain sparse as well as dense features. So it is unfair to have the same value of learning rate for all the features. The Adagrad algorithm uses the below formula to update the weights. Here the alpha(t) denotes the different learning rates at each iteration, n is a constant, and E is a small positive to avoid division by 0.

$$\mathrm{w_t} = \mathrm{w_{t-1}} - \eta'_{\mathrm{t}} \frac{\partial \mathrm{L}}{\partial \mathrm{w(t-1)}} \qquad \eta'_{\mathrm{t}} = \frac{\eta}{\mathrm{sqrt}(\alpha_{\mathrm{t}} + \epsilon)}$$

Adaptative Learning Rate Effect:

An estimate for the uncentered second moment of the objective function's gradient is given by the following expression:

$$v = \frac{1}{t} \sum_{\tau=1}^{t} g_\tau g_\tau^\top$$

which is similar to the definition of matrix G_t, used in AdaGrad's update rule. Note that AdaGrad adapts the learning rate for each parameter proportionally to the inverse of the gradient's variance for every parameter. This leads to the main advantages of AdaGrad:

1. Parameters associated with low-frequency features tend to have larger learning rates than parameters associated with high-frequency features.
2. Step sizes in directions with high gradient variance are lower than the step sizes in directions with low gradient variance. Geometrically, the step sizes tend to decrease proportionally to the curvature of the stochastic objective function.

which favors the convergence rate of the algorithm.

Algorithm:

The general version of the AdaGrad algorithm is presented in the pseudocode below. The update step within the for loop can be modified with the version that uses the diagonal of G_t.

Algorithm 1: AdaGrad general algorithm

η: Stepsize ;
$f(x)$: Stochastic objective function ;
x_1: Initial parameter vector;
for $t = 1$ *to* T **do**
  Evaluate $f_t(x_t)$;
  Get and save g_t ;
  $G_t \leftarrow \sum_{\tau=1}^{t} g_\tau g_\tau^\top$;
  $x_{t+1} \leftarrow x_t - \eta G_t^{-1/2} g_t$;
end
return x_t

AdaGrad — Adaptive Gradient Algorithm:

Intuition:

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

Update Rule for AdaGrad:

$$v_t^w = v_{t-1}^w + (\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t^w + \epsilon}} * \nabla w_t$$

$$v_t^b = v_{t-1}^b + (\nabla b_t)^2$$
$$b_{t+1} = b_t - \frac{\eta}{\sqrt{v_t^b + \epsilon}} * \nabla b_t$$

It is clear from the update rule that history of the gradient is accumulated in ***v***. The smaller the gradient accumulated, the smaller the ***v*** value will be, leading to a bigger learning rate (because ***v*** divides η).

Python Code for AdaGrad:

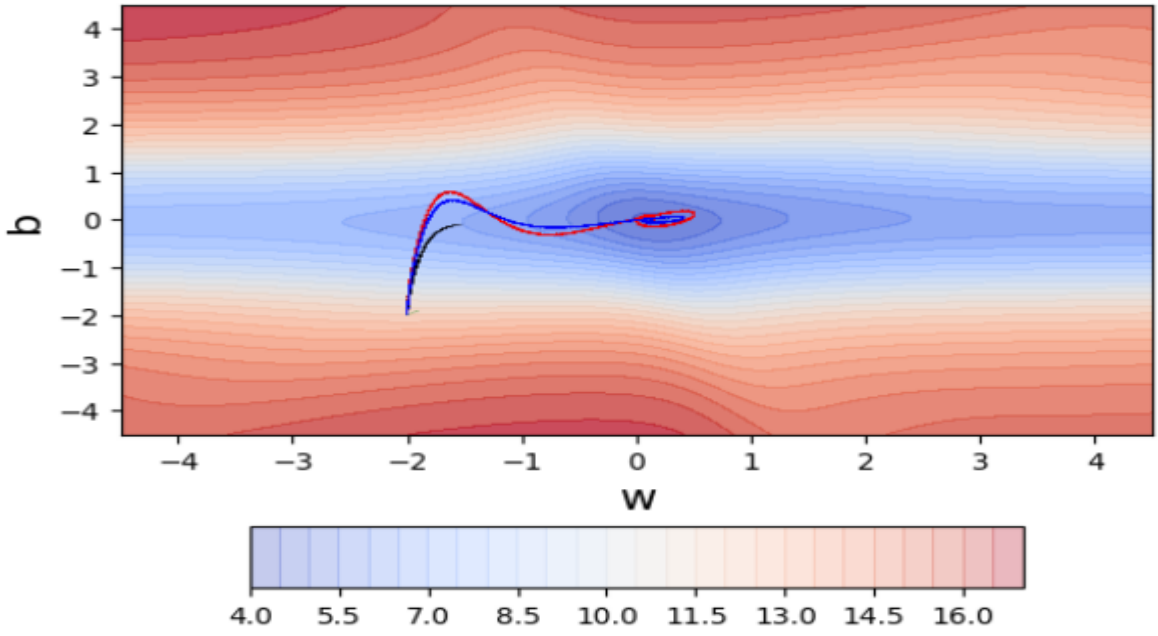
```
def do_adagrad():
    w, b, eta = init_w, init_b, 0.1
    v_w, v_b, eps = 0, 0, 1e-8
    for i in range(max_epochs):
        dw, db = 0, 0
        for x,y in zip(X,Y):
            dw += grad_w(w, b, x, y)
            db += grad_b(w, b, x, y)

        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
```

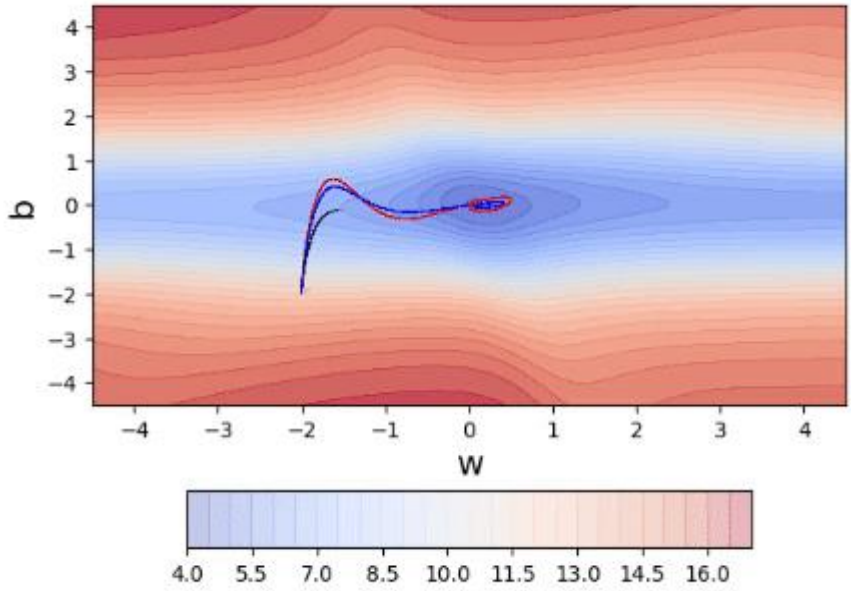
AdaGrad in Action:

To see AdaGrad in action, we need to first create some data where one of the features is sparse. How would we do this to the toy network we used across all parts of the Learning Parameters series? Well, our network has just two parameters (***w*** and ***b***). Of these, the input/feature corresponding to b is always on, so we can't really make it sparse. So the only option is to make x sparse. Which is why we created 100 random *(x,y)* pairs and then roughly 80% of these pairs we set *x* to 0, making the feature for ***w*** sparse.



Vanilla GD(black), momentum (red), NAG (blue)

Before we look at AdaGrad in action, please look at the other 3 optimizers above - vanilla GD(black), momentum (red), and NAG (blue). Something is interesting that these 3 optimizers are doing for this dataset. Can you spot it? Feel free to pause and ponder. **Answer:** 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 us now look at AdaGrad in action.



Voila! By using a parameter-specific learning rate AdaGrad ensures that despite sparsity ***w*** gets a higher learning rate and hence larger updates. Furthermore, 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? RMSProp can!

Advantages:

- **Easy to use**– It's a reasonably straightforward optimization technique and may be applied to various models.
- **No need for manual**– There is no need to manually tune hyperparameters since this optimization method automatically adjusts the learning rate for each parameter.
- **Adaptive learning rate**– Modifies the learning rate for each parameter depending on the parameter's past gradients. This implies that for parameters with big gradients, the learning rate is lowered, while for parameters with small gradients, the learning rate is raised, allowing the algorithm to converge quicker and prevent overshooting the ideal solution.
- **Adaptability to noisy data**– This method provides the ability to smooth out the impacts of noisy data by assigning lesser learning rates to parameters with strong gradients owing to noisy input.

Dis-Advantages:

- The learning rate can become too small over time, hindering convergence
- Requires more memory than SGD due to the need to store past gradients' sums

2. Explain in detail about RMS Prop (Root Mean Square) Deep Learning Optimizer

RMS prop is one of the popular optimizers among deep learning enthusiasts. This is maybe because it hasn't been published but is still very well-known in the community. RMS prop is ideally an extension of the work RPPROP. It resolves the problem of varying gradients. The problem with the gradients is that some of them were small while others may be huge. So, defining a single learning rate might not be the best idea. RPPROP uses the gradient sign, adapting the step size individually for each weight. In this algorithm, the two gradients are first compared for signs. If they have the same sign, we're going in the right direction, increasing the step size by a small fraction. If they have opposite signs, we must decrease the step size. Then we limit the step size and can now go for the weight update.

The problem with RPPROP is that it doesn't work well with large datasets and when we want to perform mini-batch updates. So, achieving the robustness of RPPROP and the efficiency of mini-batches simultaneously was the main motivation behind the rise of RMS prop. RMS prop is an advancement in AdaGrad optimizer as it reduces the monotonically decreasing learning rate.

RMS Prop Formula

The algorithm mainly focuses on accelerating the optimization process by decreasing the number of function evaluations to reach the local minimum. The algorithm keeps the moving average of squared gradients for every weight and divides the gradient by the square root of the mean square.

$$v(w,t) := \gamma v(w,t-1) + (1-\gamma)(\nabla Q_i(w))^2$$

where gamma is the forgetting factor. Weights are updated by the below formula

$$w := w - \frac{\eta}{\sqrt{v(w,t)}} \nabla Q_i(w)$$

In simpler terms, if there exists a parameter due to which the cost function oscillates a lot, we want to penalize the update of this parameter. Suppose you built a model to classify a variety of fishes. The model relies on the factor 'color' mainly to differentiate between the fishes. Due to this, it makes a lot of errors. What RMS Prop does is, penalize the parameter 'color' so that it can rely on other features too. This prevents the algorithm from adapting too quickly to changes in the parameter 'color' compared to other parameters. This algorithm has several benefits as compared to earlier versions of gradient descent algorithms. The algorithm converges quickly and requires less tuning than gradient descent algorithms and their variants.

The problem with RMS Prop is that the learning rate has to be defined manually, and the suggested value doesn't work for every application.

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:

$$v_t^w = \beta * v_{t-1}^w + (1-\beta)(\nabla w_t)^2$$
$$w_{t+1} = w_t - \frac{\eta}{\sqrt{v_t^w + \epsilon}} * \nabla w_t$$

$$v_t^b = \beta * v_{t-1}^b + (1-\beta)(\nabla b_t)^2$$
$$b_{t+1} = b_t - \frac{\eta}{\sqrt{v_t^b + \epsilon}} * \nabla b_t$$

Everything is very similar to AdaGrad, except now we decay the denominator as well.

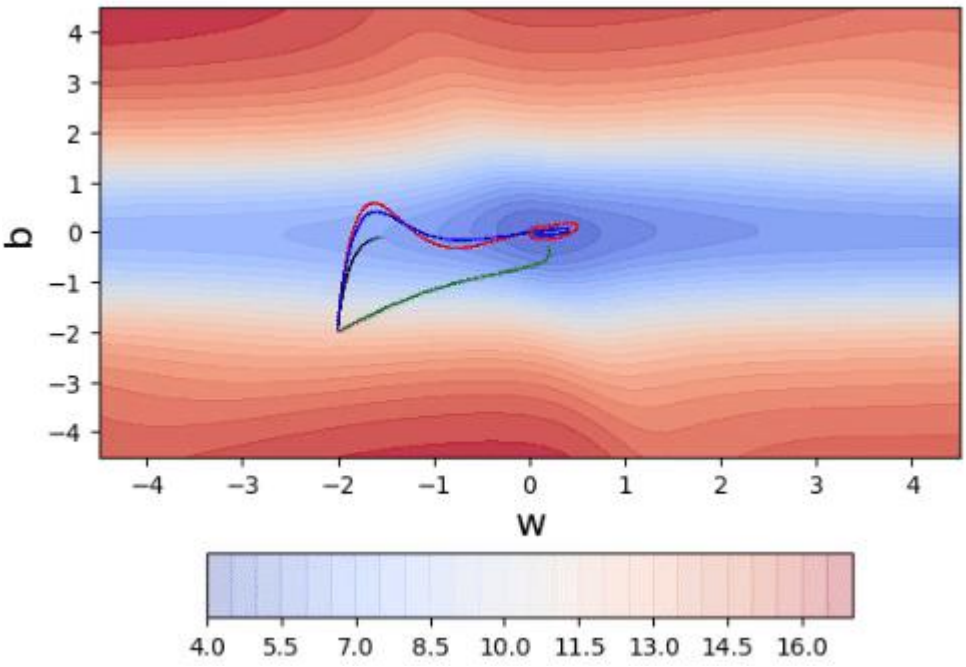
Python Code for RMSProp:

```
def do_rmsprop():
    w, b, eta = init_w, init_b, 0.1
    v_w, v_b, beta, eps = 0, 0, 0.9, 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 = beta * v_w + (1 - beta) * dw**2
        v_b = beta * v_b + (1 - beta) * db**2

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

RMSProp in Action:



Vanilla GD(black), momentum (red), NAG (blue), AdaGrad (magenta)

Advantages:

- Adapts the learning rate for each parameter based on an exponentially decaying average of past squared gradients, making it suitable for deep neural networks with many layers.
- Performs well in various domains

Disadvantages:

- May converge slowly compared to other optimizers like Adam
- Not suitable for problems with very sparse data

Deep Learning Assignment

Y20AIT449