Optimizer Algorithms in Neural Networks

Abstract:

In almost every machine learning problem or neural network problem, we have to predict something based on the learning which is basically the training which we do on the dataset. So, the training is based on the weights which is a learnable parameter which keeps on changing until it predicts the accurate output.

The cost function (or) error function which is the difference between the actual and the predicted values needs to be minimized in order for the model to predict the correct output. So, in order to minimize the error function, we use different optimizers which helps in the minimization (or) reaching the global optimum of the function.

Team #7:

Nishchal M N - PES1201701523
Ravichandra K G - PES1201701581

Problem description:

We have used the MNIST data set of handwritten digits, from which we have developed a binary classifier for digits 0 and 1. We have implemented the neural network from scratch in python with the help of modules like numpy, matplotlib, etc.

The architecture contains input of 28*28 image which is vectorized to 1D with 784 input neurons, followed by 256 neurons hidden layer with Elu activation functions, followed by 128 neurons hidden layer with tanh activation function, and finally an output neuron with sigmoid activation function which gives the classification of the input image as either digit 0 or 1.

Optimizers:

1. Gradient Descent

Formula for the weight updation is given by,

$$\vec{w} = \vec{w} + \Delta\vec{w}, \text{ where } \Delta\vec{w} = -\eta * \nabla E(\vec{w})$$

$$w_i = w_i + \Delta w_i, \text{ where } \Delta w_i = -\eta * \partial E/\partial w_i$$

Where,

E is Error function or loss function given as:

$\Delta w_i = \eta \sum (t_d - o_d) x_{id}$

Where,

t_d - target output

o_d - generated output

w is weight associated with input

η is learning rate, determines the size of steps to reach the minimum gradient value, Value should be low

□E(w) is gradient of error function E(w) w.r.t weight w

Gradient descent algorithm updates the parameters by moving in the direction opposite to the gradient of the objective function with respect to the network parameters.

Parameter updation:

- Initialize the parameters w and b randomly and iterate over all the observations in the given data.
- Calculate the value of the gradient for each parameter.
- Update the value of each parameter based on its gradient value.
- Continue doing step 2 and 3 till loss function gets minimum value.

2. Stochastic Gradient Descent

The normal gradient descent update weights after summing over all the training examples in the training set.

Stochastic Gradient Descent(SGD) performs a parameter update for each training example. It is much faster than gradient descent. It performs one update at a time. Error function or loss function E is given as:

$$E(w) = ½ (t_d - o_d)^2$$

Every point is making the greedy decision and updates the parameters best suitable for that point. We can reduce the problem of oscillations is by considering a subset of data or batch of data and update the parameters after calculating the derivatives for each point within that batch.

3. Momentum

Formula for momentum gradient descent is given as:

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

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

Here we include the history component v_t for updating weights, v_t stores all previous gradient movements till the current time t.
v_t is given as:

$$v_t = \gamma \cdot v_{t-1} + \eta \nabla w_t = \gamma^{t-1} \cdot \eta \nabla w_1 + \gamma^{t-2} \cdot \eta \nabla w_1 + \ldots + \eta \nabla w_t$$

The momentum term γ is usually set to 0.9 or a similar value.

It leads to faster and stable convergence and reduces oscillations. The momentum term γ increases for dimensions whose gradients point in the same directions and reduces updates for dimensions whose gradients change directions. This means it does parameter updates only for relevant examples. This reduces the unnecessary parameter updates which leads to faster and stable convergence and reduced oscillations.

4. Nesterov Accelerated Gradient

In momentum based gradient descent as we reach the minima i.e the lowest point on the curve, the momentum is pretty high and it doesn't know how to slow down at that point due to the high momentum which could cause it to miss the minima entirely and continue moving up. This problem can be solved by using Nesterov Accelerated Gradient.

NAG Update rule is given as:

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

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

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

Here we are making one movement in the direction of gradient history and another in the direction of the current gradient because of this two-step movement we are overshooting the minima. Instead of moving two steps at a time, Move a little in the direction of gradient history, compute the gradient at that point and update the parameters

5. Adagrad

AdaGrad — Gradient Descent with Adaptive Learning Rate

It simply allows the learning Rate -η to adapt based on the parameters. So it makes big updates for infrequent parameters and small updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data. It uses a different learning Rate for every parameter θ at a time step based on the past gradients which were computed for that parameter.

If a particular parameter is getting updated frequently that means the derivative is not zero most of the time, for such parameters we want to have a smaller learning rate in contrast if we have a sparse parameter which will be off (zero) most of the times that mean the derivative will be zero most of the time. For such feature whenever this feature is on (not zero) we want to boost the gradient update with a higher learning rate.

Update rule is given as:

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

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

Here we are dividing the learning rate with the history of gradient value until that point. Non-sparse features will have a large history value because they would be getting frequent updates, by dividing the learning rate with the large history, the effective learning rate would be very small. In the case of sparse features, gradient history value would be very less which leads to large effective learning rate.

The main benefit of Adagrad is that we don’t need to manually tune the learning Rate. Most implementations use a default value of 0.01 and leave it at that.

Disadvantage:
Its main weakness is that its learning rate-η is always Decreasing and decaying due to the accumulation of each squared Gradients in the denominator.

This problem of Decaying learning Rate is Rectified in another algorithm called AdaDelta.

6. RMSprop

RMSProp — Root Mean Square Propagation

Update rule is given as:

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

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

RMSProp uses the intuition of decay the denominator and prevent the rapid growth to prevent the rapid growth of the denominator for dense variables so the effective learning rate doesn’t become close to zero.

Here history of gradients is calculated using an exponential decaying average unlike the sum of gradients in AdaGrad, which helps to prevent the rapid growth of the denominator for dense features.
Since the denominator is not decaying as aggressively as AdaGrad, RMSProp is able to move in the direction of convergence.

7. Adam

The name Adam is derived from adaptive moment estimation.

Adaptive Moment Estimation (Adam) is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients like AdaDelta ,Adam also keeps an exponentially decaying average of past gradients

In Momentum based gradient descent we are use the cumulative history of gradients to move faster in the gentle surfaces and we have seen RMSProp which is also using history to decay the denominator and prevent its rapid growth. The way these algorithms uses the history is different, In Momentum gradient descent, we use history to compute the current update whereas in case of RMSProp history was used to adjust the learning rate.

Update rule is given as:

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

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

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

Adam maintains two histories, ‘m□’ similar to the history used in Momentum GD and ‘v□’ similar to the history used in RMSProp.

Adam bias correction. It equations for this is given as ‘m□’ and ‘v□’,

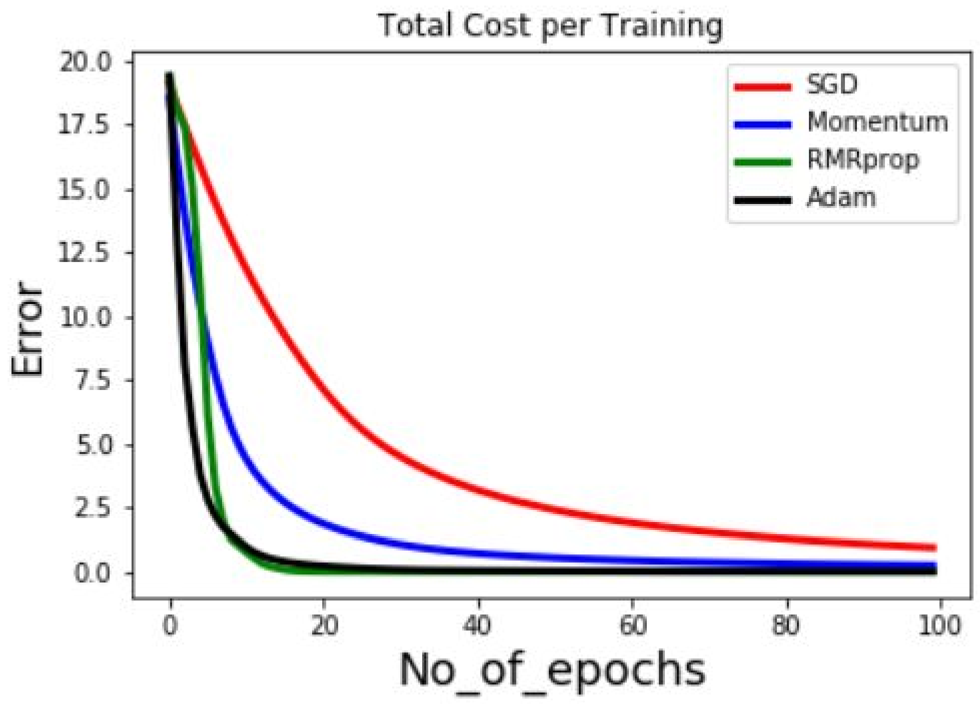
$$m_t = \frac{m_t}{1 - \beta_1^t}$$

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

Bias correction ensures that at the beginning of the training updates don’t behave in a weird manner. The key point in Adam is that it combines the advantages of Momentum GD (moving faster in gentle regions) and RMSProp GD (adjusting learning rate).

Results:

In the reverse order of the optimizers explained, they take very few steps to reach the global minimum of the error function. Hence faster in convergence.



Conclusion:

Though the choice of the optimizer functions depends upon the problem being solved. By far, Adam optimizer has proved to be the best optimizer in most of the neural network architectures developed.