CSE641 - Deep Learning
Assignment 2 - Part 1
ReadMe File

Submitted By:
Chirag Chawla(MT19089)
Kashish Narang(MT19026)
Saumya Jain(MT19098)

Question 1 Part 1:

Methodology :

We have implemented the back propagation algorithm from scratch. Here are the steps for the implementation :

1. Create the network structure, in our case, we have taken the network structure with three layers : Input, Hidden, Output & the number of neurons in each layer is 784, 32, 10 respectively.

2. Next, we have initialized the weight matrix for the above network with some random values within range -1 to +1.

3. **compute_net_input() :** This method is used to compute the net input for a particular layer which is done by Z_i = X. W $_i^T$ where i is the neuron number for the corresponding layer. So we will evaluate the Z_i s for all the neurons in the corresponding layer.

4. **activation() :** Given input list, this method will perform the activation function (sigmoid or Relu or tanh) which is specified as a parameter of class.

5. **forward() :** This method will take the input, do the complete forward pass and generate the output by following steps :

   Z1 = compute_net_input(1) // 1 is layer number (Hidden Layer)

   O1 = activation(Z1)

   Z2 = compute_net_input(2) // 2 is the layer number (Output Layer)

   O2 = softmax(Z2) // O2 is the output generated by the network.

6. **error() :** Now after the forward pass which predicts the output from the network. Next step is to evaluate the error to get an idea of how our network is working. So this error method will take the predicted output and the actual output as parameters and generate the error by using the cross entropy error formula.

7. **accuracy() :** Instead of error, one more evaluation metric is accuracy. So we have also evaluated the accuracy of predicted values after the forward pass.

8. **gradient() :** Now we have evaluated the error that is occurring in the predicted values with the current weights. Now our task is to back propagate this error to the previous layers and find out the derivative $\partial Loss / \partial W$. So our gradient method will evaluate the gradients.

9. **gradient_descent() :** After the gradients have been evaluated, we have to use these gradients to update the weights by following :

$$W_{new} = W_{old} - \eta * \partial Loss / \partial W$$

So our gradient_descent method will take the gradients as input, and by using these gradients it will update all the weights with respect to corresponding gradients.

10. Repeat the steps 5-9 for multiple epochs.

Preprocessing :

- We have a dataset of 10000 images with 28X28 pixels per image.
- We have converted each image into a 2 - dimensional matrix having dimension 28X28.
- Next, we have flattened this 28X28 2-dimensional matrix into 1-dimension vectors of 784 elements.
- **Normalization :** Now each image consists of values in range 0 to 255 (RGB standard values), so in order to normalize we have divided each value by 255.
- **Weight Initialization :** We have initialized all the weights randomly within range -1 to +1.

Question 1 Part 2:

In this part, we have to implement the five optimizers :

- Gradient Descent with momentum
- NAG
- Adagrad
- RMSprop
- Adam

Methodology :

gradient_descent_momentum() : This method is implementing the optimizer gradient descent with momentum by following :

$$M^{(t)} = M^{(t-1)} - \eta * \partial Loss / \partial W$$
$$W_{new} = W_{old} + M^{(t)}$$

$M^{(t)}$ is initialized to all 0's initially. So this optimizer is basically taking history (as momentum) along with derivatives to move the weights.

NAG() : This method is implementing the NAG optimizer by the following equation :

$$W_{lookahead} = W_{old} - M^{(t-1)}$$
$$M^{(t)} = M^{(t-1)} + \eta * \partial Loss / \partial W_{lookahead}$$
$$W_{new} = W_{old} - M^{(t)}$$

$M^{(t)}$ is initialized to all zero's initially. In NAG, we are re-ordering the terminologies of Gradient descent with momentum. Here, we are first moving by previous history ($M^{(t-1)}$) to reach new position i.e $W_{lookahead}$, then we are taking the derivative of $W_{lookahead}$ and then adjust the weights.

This re-ordering terminology will prevent us from the unwanted oscillation which is caused by the momentum because we are moving by history + gradient which might cross the global minima, then we have to move back, so it causes several oscillations. But NAG will first move by the history and then take the gradient to see the direction here itself. So we are not crossing the global minima by much.

adagrad() : This method is implementing the Adagrad optimizer which basically uses different learning rate for different weights ate each epoch by the following :

$$V^{(t)} = V^{(t-1)} - (\partial Loss / \partial W * \partial Loss / \partial W)$$
$$W_{new} = W_{old} + (\eta / \sqrt{(V^{(t)} + \varepsilon}) * \partial Loss / \partial W$$

RMSprop() : This method is implementing the RMSpropoptimizer which basically uses a controlling parameter β in order to control the square of previous gradients by the following :

$$V^{(t)} = \beta * V^{(t-1)} - [(1 - \beta) * (\partial Loss / \partial W * \partial Loss / \partial W)]$$
$$W_{new} = W_{old} + (\eta / \sqrt{(V^{(t)} + \varepsilon}) * \partial Loss / \partial W$$

adam() : This method is implementing the RMSpropoptimizer which basically uses two controlling parameters $\beta 2$ in order to control the square of previous gradients & $\beta 1$ in order to control the momentum by the following :

$$V^{(t)} = \beta 2 * V^{(t-1)} - [(1 - \beta 2) * (\partial Loss / \partial W * \partial Loss / \partial W)]$$

$$M^{(t)} = \beta 1 * M^{(t-1)} + [(1 - \beta 1) * (\partial Loss / \partial W)]$$

$$W_{new} = W_{old} + (\eta / \sqrt{(V^{(t)} + \varepsilon}) * M^{(t)}$$