Gradient Descent

Repeat until Convergence (Aims to minimize the cost function of an Algorithm)

Definitions:

1. In mathematics, **gradient descent** (also often called **steepest descent**) is a first-order iterative optimization algorithm for finding a local minimum of a differentiable function. The idea is to take repeated steps in the opposite direction of the gradient (or approximate gradient) of the function at the current point, because this is the direction of steepest descent. Conversely, stepping in the direction of the gradient will lead to a local maximum of that function; the procedure is then known as *gradient ascent*.(Wikipedia)

2. Gradient descent is a fundamental optimization method that operates by iteratively adjusting the parameters of a model in the direction of the steepest descent of the objective function. By computing the gradient, which represents the direction of maximum increase of the function, and negating it, the algorithm seeks to find the minimum of the function through a series of parameter updates.

3. Gradient descent is an optimization algorithm which is commonly-used to train machine learning models and neural networks. Training data helps these models learn over time, and the cost function within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates. Until the function is close to or equal to zero, the model will continue to adjust its parameters to yield the smallest possible error. Once machine learning models are optimized for accuracy, they can be powerful tools for artificial intelligence (AI) and computer science applications.(IBM)

4. Gradient Descent reduces the mistake of an algorithm in a speed(in technical terms - Learning rate) by adjusting some parameters. This works like children's parents.

There are three variants of gradient descent , which differ in how much data we use to compute the gradient of the objective function.
Depending on the *amount of data*, we make a trade-off between the *accuracy* of the parameters update and the *time* it takes to perform an update.

Types

- Batch Gradient Descent
- Stochastic Gradient Descent
- Mini-Batch Gradient Descent

Batch Gradient Descent:

Definition - Batch Gradient Descent (BGD) is an optimization algorithm commonly used to minimize the cost function in machine learning models. It updates the model parameters based on the average gradients of the entire training dataset.

How works?...

1. Initialize the model parameters randomly or with some specific values.
2. Set the learning rate (alpha) and the maximum number of iterations (epochs).
3. Repeat the following steps for each epoch:

   a. Initialize the gradients of the model parameters to zero.

   b. For each training example (x, y) in the dataset:

      i. Compute the gradients of the cost function with respect to each model parameter using the current model's parameters and the current training example.

      ii. Accumulate the gradients by adding them to the initialized gradients.

   c. Update the model parameters using the accumulated gradients and the learning rate:
      new_parameter = old_parameter - (learning_rate * accumulated_gradient)

   d. Repeat steps b and c until you have processed all the training examples.

4. Repeat the above process for the specified number of epochs or until convergence.

Note: In Batch Gradient Descent, the entire dataset is used to compute the gradients in each iteration, which can be computationally expensive for large datasets. However, it provides a stable and deterministic convergence as it considers the average gradient over the entire dataset.

Pseudo-code

```
# Initialize model parameters
initialize_parameters()

# Set hyperparameters
learning_rate = 0.01
num_epochs = 5

for epoch in range(num_epochs):
    # Initialize gradients
    initialize_gradients()

    for (x, y) in training_dataset:
        # Compute gradients for each training example
        gradients = compute_gradients(x, y)

        # Accumulate gradients
        accumulate_gradients(gradients)

    # Update model parameters using the accumulated gradients
    update_parameters(learning_rate)
```

Stochastic Gradient Descent (SGD):

Stochastic Gradient Descent (SGD) is an optimization algorithm commonly used to train machine learning models, especially in the context of deep learning and neural networks. Unlike traditional Gradient Descent, which updates the model parameters based on the average gradient of the entire dataset, SGD updates the parameters after processing each individual data point (or a small batch of data points). This makes it more computationally efficient and allows it to handle large datasets.

Pseudo code

```
function stochastic_gradient_descent(data, learning_rate, num_epochs, batch_size, initial_parameters):
    parameters = initial_parameters
    num_samples = length(data)

    for epoch in range(num_epochs):
        # Shuffle the data to randomize the order of samples in each epoch
        shuffle(data)

        for batch_start in range(0, num_samples, batch_size):
            # Extract a mini-batch of data points
            batch_data = data[batch_start: batch_start + batch_size]

            # Compute the gradient of the loss function with respect to the parameters
            gradient = compute_gradient(parameters, batch_data)

            # Update the parameters using the gradient and learning rate
            parameters = parameters - learning_rate * gradient

    return parameters
```

Batch VS Stochastic Gradient Descent:

Data Processing Approach:
Batch gradient descent computes the gradient of the cost function with respect to the model parameters using the entire training dataset in each iteration. Stochastic gradient descent, on the other hand, computes the gradient using only a single training example or a small subset of examples in each iteration.

Convergence Speed:
Batch gradient descent takes longer to converge since it computes the gradient using the entire training dataset in each iteration. Stochastic gradient descent, on the other hand, can converge faster since it updates the model parameters after processing each example, which can lead to faster convergence.

Convergence Accuracy:
Batch gradient descent is more accurate since it computes the gradient using the entire training dataset. Stochastic gradient descent, on the other hand, can be less accurate since it computes the gradient using a subset of examples, which can introduce more noise and variance in the gradient estimate.

Computation and Memory Requirements:
Batch gradient descent requires more computation and memory since it needs to process the entire training dataset in each iteration. Stochastic gradient descent, on the other hand, requires less computation and memory since it only needs to process a single example or a small subset of examples in each iteration.

Optimization of Non-Convex Functions:
Stochastic gradient descent is more suitable for optimizing non-convex functions since it can escape local minima and find the global minimum. Batch gradient descent, on the other hand, can get stuck in local minima.

In summary, batch gradient descent is more accurate but slower, while stochastic gradient descent is faster but less accurate. The choice of algorithm depends on the specific problem, dataset, and computational resources available.

Mini Batch Gradient Descent:

Mini Batch Gradient Descent is considered to be the cross-over between GD and SGD. In this approach instead of iterating through the entire dataset or one observation, we split the dataset into small subsets (batches) and compute the gradients for each batch.

How works?.....
1. Divide the dataset into mini-batches: The training dataset is divided into smaller subsets of fixed size. These subsets are called mini-batches. The size of the mini-batch is typically a power of 2, like 32, 64, or 128, but it can be adjusted based on hardware constraints and dataset size.

2. Initialize model parameters: Initialize the model's parameters (weights and biases) randomly or using some predefined values.

3. Choose a learning rate: The learning rate is a hyperparameter that controls the step size during parameter updates. It determines how big a step the algorithm takes in the direction of the gradient.

4. Iterate through mini-batches: For each iteration, the algorithm processes one mini-batch at a time.

5. Compute gradients: Compute the gradient of the cost function with respect to the model parameters for the current mini-batch. This involves performing forward propagation to calculate the predictions and then backward propagation to compute the gradients.

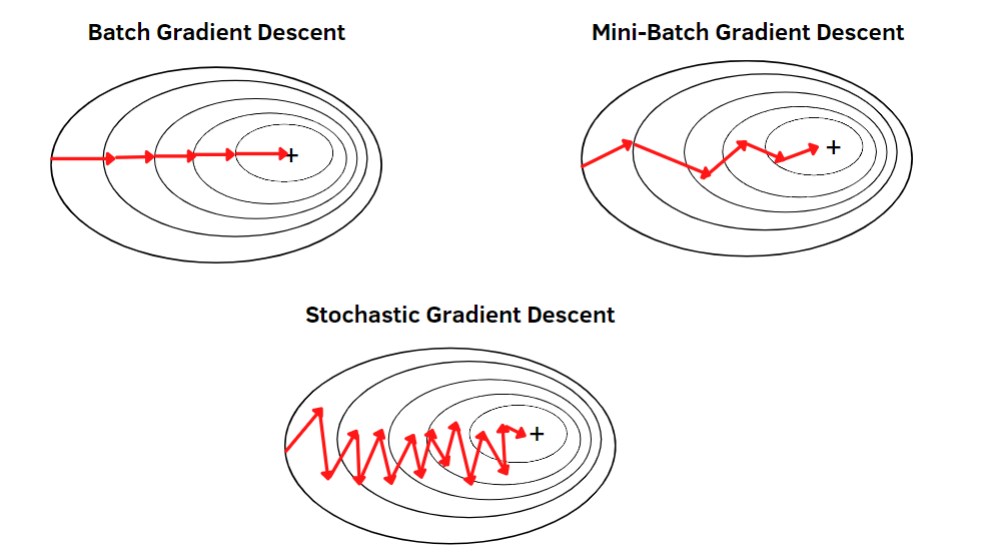
6. Update parameters: Update the model parameters using the computed gradients and the learning rate. The update equation typically looks like: parameter = parameter - learning_rate * gradient.

7. Repeat: Continue iterating through mini-batches and updating the parameters until convergence or reaching a predefined number of iterations.

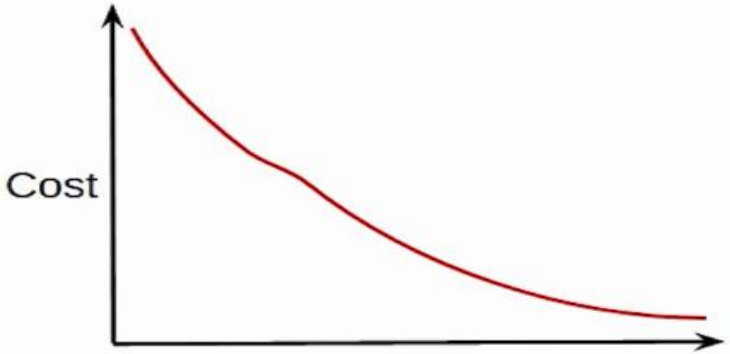
Pseudo code:

```
function mini_batch_gradient_descent(X, y, learning_rate, batch_size, epochs):
    Initialize model parameters randomly or using predefined values
    total_samples = number of training samples
    total_batches = total_samples // batch_size

    for epoch in range(epochs):
        shuffle(X, y)  # Shuffle the training data at the beginning of each epoch

        for batch in range(total_batches):
            start_idx = batch * batch_size
            end_idx = (batch + 1) * batch_size
            X_batch = X[start_idx:end_idx]
            y_batch = y[start_idx:end_idx]

            # Forward propagation: calculate predictions for the mini-batch
            predictions = forward_propagation(X_batch)

            # Compute the loss/cost for the mini-batch
            loss = calculate_loss(predictions, y_batch)

            # Backward propagation: compute gradients
            gradients = backward_propagation(X_batch, y_batch, predictions)

            # Update model parameters using gradients and learning rate
            update_parameters(gradients, learning_rate)

        # Optionally, you can compute the total loss for the epoch for monitoring purposes
        total_loss = calculate_total_loss(X, y)

    return model_parameters
```

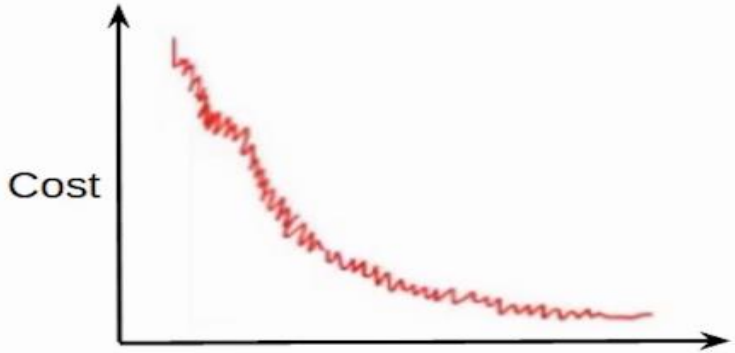
Let's Visualize:



- Cost function reduces smoothly



- Lot of variations in cost function



- Smoother cost function as compared to SGD