**Regular Linear regression Model:**

1. **Data Collection**: After loading and describing the dataset, we must encode the non-numerical features (d-type = object) into numerical before applying the regression algorithm.
2. **Data Normalization**: The goal of normalization is to bring all the features into a similar range so that no single feature dominates the learning process or causes numerical issues. Common normalization techniques include Min-Max scaling and Z-score normalization (Standardization).

* **Min-Max Scaling**: It scales the data to a fixed range, usually between 0 and 1. The formula for Min-Max scaling is:
  + X\_normalized = (X - X\_min) / (X\_max - X\_min)

1. **Data Scandalization**: is a method of normalization that transforms the data in such a way that the mean becomes 0 and the standard deviation becomes 1. This method ensures that our data have been normalized correctly or not.

* Z-score Normalization (Standardization): It transforms the data to have a mean of 0 and a standard deviation of 1.
  + X\_normalized = (X - mean(X)) / std(X)

1. **Data split**: the process of dividing the dataset into two separate subsets: the training set and the testing set, the reason for this split is to mimic real-world scenarios where the model needs to make predictions on data it has never seen before. If we use the entire dataset for training, the model might become overly specialized and fail to generalize to new, unseen data.
   * **Training Set**: This portion of the data (usually around 70-80% of the dataset) is used to train the linear regression model. During the training process, the model learns the relationships between the input features and the target variable by adjusting its parameters (slope and intercept) based on the training data.
   * **Testing (Validation) Set**: This portion of the data (usually around 20-30% of the dataset) is used to evaluate the performance of the trained model. After the model is trained, it is tested on this independent dataset to see how well it can make predictions on unseen data. The model's predictions are compared with the actual target values from the testing set, and various evaluation metrics (e.g., Mean Squared Error, R-squared, etc.) are calculated to measure the model's performance.



1. A black symbols with a plus and a cross

   Description automatically generatedLinea regression formula is:

* **x:** input data points
* **y:** predicted value, dependent variable (supervised learning)

The model gets the best-fit regression line by finding the best m, c values.

* **m:** bias or slope of the regression line
* **c:** intercept, shows the point where the estimated regression line crosses the 𝑦 axis

1. Cost Function (J) : As explained above our goal is to find a regression line or the best fit line which has the least difference (error/residual) between the predicted value and the actual value. This is where the cost function comes into the picture as we use the cost function extensively to calculate the values of ( c, m) to reach the best value that minimizes the error between predicted y value (y^) and true y value (y).

We calculate the cost function using the MSE (Mean Squared Error) formula-

Given our simple linear equation y = c + m\*x, we can calculate MSE as: formula,

A black text on a white background

Description automatically generatedA mathematical equation with numbers and symbols

Description automatically generated

* 𝑁 is the total number of observations (data points)
* yᵢ is the actual value of an observation and y^ is the predicted value
* J is the cost function which is the mean squared error in this case

**NOTE**: Lower MSE indicates that the model's predictions are closer to the actual values, which is a desirable outcome.

1. after fitting the best linear regression line using the cost function, in this step we need to calculate m & c for the above formula (y = c + m\*x), for calculating m we use this formula:

A black symbols on a white background

Description automatically generated

Where is the mean of x and is the mean of y for example:

A table with numbers and a arrow

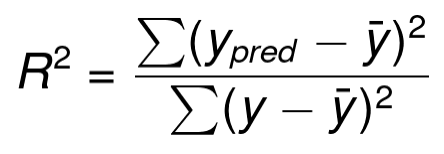
Description automatically generated

so moving ahead, according to the formula of ‘m’, what we’re gonna do is calculate (x-x̅ )& (y-y̅) for each data point in our very simple dataset.

A white sheet with black and white text

Description automatically generated

1. Now it’s time to measure how good our model is. For this, we will calculate the R-squared value and evaluate our linear regression model, R-squared value is a statistical measure of how close the data are to the fitted regression line, Here’s how we calculate R-squared value:



Where ypred is the predicted y value and y̅ is the mean and y is the actual value

A white sheet with black and blue text

Description automatically generated

The higher the R-squared value the better our model performance will be. So as the R-squared value gradually increases, the distance of actual points from the regression line decreases, and the performance of the model increases.

**Gradient decent Linear regression Model:**

1. Initially let m = 0 and c = 0. Let L be our learning rate. This controls how much the value of m changes with each step. L could be a small value like 0.0001 for good accuracy.
2. Calculate the partial derivative of the loss function with respect to m, and plug in the current values of x, y, m and c in it to obtain the derivative value D.

A math equations with numbers and symbols

Description automatically generated

1. Dₘ is the value of the partial derivative with respect to m. Similarly let’s find the partial derivative with respect to c, Dc :

A mathematical equation with numbers and symbols

Description automatically generated

1. Now we update the current value of m and c using the following equation:

A math equations on a white background

Description automatically generated

1. We repeat this process until our loss function is a very small value or ideally 0 (which means 0 error or 100% accuracy). The value of m and c that we are left with now will be the optimum values.

A screenshot of a computer program

Description automatically generated

**Mini batch Gradient decent on Linear regression Model**

Mini-batch gradient descent combines the benefits of both batch gradient descent and stochastic gradient descent (SGD) to efficiently update the model parameters during training.

Here's how mini-batch gradient descent works:

* **Batch Gradient Descent**: In batch gradient descent, we update the model parameters (weights and biases) using the gradients of the loss function with respect to the entire training dataset. This means that we compute the gradients for all the training examples and then update the parameters accordingly. While this ensures more stable updates and convergence, it can be computationally expensive, especially for large datasets.

In Batch Gradient Descent, all the training data is taken into consideration to take a single step. We take the average of the gradients of all the training examples and then use that mean gradient to update our parameters. So that’s just one step of gradient descent in one epoch.

A graph with a dotted line

Description automatically generated

* **Stochastic Gradient Descent (SGD)**: In stochastic gradient descent, we update the model parameters after each individual training example. This leads to more frequent updates and faster convergence but can also result in noisy updates, making the optimization process more erratic and less stable.

This algorithm tackles the problem of having huge database since batch algorithm will calculate the gradients of all the 5 million examples, in Stochastic Gradient Descent (SGD), we consider just one example at a time to take a single step. We do the following steps in one epoch for SGD:

* + Take an example.
  + Feed it to Neural Network.
  + Calculate it’s gradient.
  + Use the gradient we calculated in step 3 to update the weights.
  + Repeat steps 1–4 for all the examples in training dataset.

A graph of a number of gradients

Description automatically generatedSince we are considering just one example at a time the cost will fluctuate over the training examples and it will not necessarily decrease. But in the long run, you will see the cost decreasing with fluctuations.

**Mini-batch Gradient Descent**: Mini-batch gradient descent strikes a balance between batch gradient descent and stochastic gradient descent. Instead of updating the model parameters using the gradients of the entire dataset or just one example, mini-batch gradient descent divides the dataset into smaller subsets called mini-batches.

Each mini-batch contains a predefined number of training examples (commonly a power of 2, like 32, 64, or 128). The size of the mini-batch is a hyperparameter chosen by the user.

The model parameters are updated based on the average gradients of the loss function computed on the current mini-batch. This allows mini-batch gradient descent to benefit from more stable updates than SGD while being computationally more efficient than batch gradient descent.

The steps of mini-batch gradient descent are as follows:

1. **Shuffle the Training Data**: Before starting each epoch (a full pass through the training data), shuffle the training examples randomly. This ensures that each mini-batch contains diverse samples and reduces the risk of getting stuck in local optima.
2. **Pick a Mini-batch**: Split the shuffled training data into mini-batches of equal size (except for the last one, which may have fewer examples). The size of the mini-batch is a hyperparameter chosen by the user.
3. **Feed to Neural Network**: For each mini-batch, feed the mini-batch into the neural network for forward propagation. The neural network processes each example in the mini-batch through its layers, calculating the activations of each neuron and producing predictions for the target variable.
4. **Compute Loss**: Compute the loss (or cost) function for the mini-batch. The loss function measures the difference between the predicted values and the actual target values for the examples in the mini batch.
5. **Calculate Gradients**: Perform backward propagation (also known as backpropagation) to calculate the gradients of the loss function with respect to the model parameters (weights and biases).
6. **Update Model Parameters**: Using the gradients obtained in the previous step, update the model parameters (weights and biases) of the neural network.
7. **Repeat**: Repeat steps 2-6 for all mini batches in the training data. Each mini-batch updates the model parameters, and the process continues until all mini-batches have been processed.

A graph with a line drawn on it

Description automatically generated

Differences &Advantages

**Batch Gradient Decent:**

* The graph of cost vs epochs is also quite smooth because we are averaging over all the gradients of training data for a single step. The cost keeps on decreasing over the epochs.

**Stochastic Gradient Descent:**

* Since we are considering just one example at a time the cost will fluctuate over the training examples and it will not necessarily decrease. But in the long run, you will see the cost decreasing with fluctuations.
* Also, because the cost is so fluctuating, it will never reach the minima, but it will keep dancing around it.
* SGD can be used for larger datasets. It converges faster when the dataset is large as it causes updates to the parameters more frequently.

**Mini-Batch Gradient Decent:**

* Just like SGD, the average cost over the epochs in mini-batch gradient descent fluctuates because we are averaging a small number of examples at a time.
* when we are using the mini-batch gradient descent we are updating our parameters frequently as well as we can use vectorized implementation for faster computations.

**Momentum, Adagrad and Adam Optimizers**

**Momentum**

The problem with gradient descent is that the weight update at a moment (t), It doesn’t take into account the past steps taken while traversing the cost space.

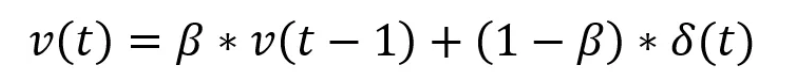
Momentum is an optimization algorithm that aims to accelerate gradient descent in the relevant direction.

It introduces a "velocity" term that accumulates the gradients over time. The idea is to add a fraction of the previous velocity to the current update step.

Algorithm steps:

* Initialize the velocity for each parameter as zero.
* Compute the gradients for the current mini batch.
* Update the velocity using a momentum factor (usually denoted by β) and the current gradients.
* Update the parameters using the velocity.
* Repeat the process for each mini batch.

In our case of a sequence of gradients, the new weight update equation at iteration t becomes:



* 𝓥(t): is the new weight update done at iteration t
* **β** is the coefficient that represents the degree of weighting increase, a constant smoothing factor between 0 and 1. A lower β discounts older observations faster.
* **𝛿(t)**: is the gradient at iteration t

A math equations with numbers and symbols

Description automatically generated

Therefore, the equation becomes for calculating V(n):

A math equation with a triangle and triangle and square and triangle with a triangle and a triangle with a triangle and a triangle with a triangle and a triangle with a triangle and a triangle with a triangle

Description automatically generated

**Note**: In many texts, you might find **(1-β)** replaced with **η** the learning rate.

Even though momentum with gradient descent converges better and faster, it still doesn’t resolve all the problems. First, the hyperparameter η (learning rate) has to be tuned manually. Second, in some cases, where, even if the learning rate is low, the momentum term and the current gradient can alone drive and cause oscillations.

The Learning rate problem can be further resolved by using other variations of Gradient Descent like Adaptive Gradient (**Adagrad**)

**Adagrad (Adaptive Gradient algorithm)**

we saw how learning rate(η) affects the convergence. Setting the learning rate too high can cause oscillations around minima and setting it too low, slows the convergence. Learning Rate(η) in Gradient Descent and its variations like Momentum is a hyper-parameter which needs to be tuned manually for all the features.

A black emoji with a white background

Description automatically generated

When we use the above equation for updating weights in a neural net

* Learning rate is the same for all the features.
* Learning rate is the same at all the places in the cost space.

Adagrad (Adaptive Gradient Algorithm) is an adaptive learning rate optimization algorithm. It adjusts the learning rate for each parameter based on the historical gradients of that parameter. Parameters that have larger gradients will have a smaller learning rate, and parameters with smaller gradients will have a larger learning rate.

Algorithm steps:

* Initialize a per-parameter learning rate accumulator as zero.
* Compute the gradients for the current mini batch.
* Accumulate the squares of the gradients in the learning rate accumulator.
* Update the parameters using the adaptive learning rate, which is the original learning rate divided by the square root of the accumulated gradients.
* Repeat the process for each mini batch.

So, the new proxy for update frequency 𝓥(t) becomes.

A math equations on a white background

Description automatically generated

Adagrad automatically adapts the learning rates, allowing for more substantial updates for rare parameters and smaller updates for frequent parameters. It is well-suited for sparse data and tasks where different features have different frequencies in the data.

**Adam Optimizer**

Adam (Adaptive Moment Estimation) combines the concepts of momentum and adaptive learning rates from Momentum and Adagrad, respectively. It uses both the first-order moment (mean) and the second-order moment (uncentered variance) of the gradients to compute adaptive learning rates for each parameter.

Algorithm steps:

* Initialize the first and second moment accumulators for each parameter as zero.
* Compute the gradients for the current mini batch.
* Update the first and second moment accumulators.
* Compute the bias-corrected estimates of the first and second moments.
* Update the parameters using the adaptive learning rates, which are the original learning rate multiplied by the first moment divided by the square root of the second moment.
* Repeat the process for each mini batch.

For updating the weights with an adaptive learning rate at iteration t, first, we need to calculate the first and second moment given by the following formulae,

**— — GD with Momentum (1st)**

**— — Adagrad(2nd)**

The corrected and is given by,

Therefore, the new weight will be updated using the formula,

The initial value of n is to be tuned for better results.

Adam is relatively easy to configure where the default configuration parameters do well on most problems. It is proposed to have default values of β1=0.9, β2 = 0.999 and ε =10E-8. Studies show that Adam works well in practice, in comparison to other adaptive learning algorithms.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Algorithm** | **Problem Solved** | **Importance & Use Cases** | **When to Use** | **Problems it Doesn't Solve** | **When Not to Use** |
| **Gradient Descent Linear Regression** | Minimizing the Cost Function in Linear Regression | Simplest optimization method for linear regression. | When dealing with small to medium-sized datasets and linear models. | Not suitable for non-linear models. | Avoid for complex non-linear data, use for linear problems only. |
| **Mini Batch Gradient Descent** | Faster Convergence, Better Generalization | Effective for large datasets; reduces noise in parameter updates. | Preferred for large-scale datasets and neural network training. | Can still get stuck in local minima. | Avoid if computation resources are limited, use smaller datasets. |
| **Momentum** | Accelerating Convergence | Helps overcome the problem of slow convergence. | When dealing with deep learning models or saddle point issues. | May overshoot and oscillate around minima. | Avoid for simple models as it may not show significant improvement. |
| **Adagrad** | Adaptive Learning Rates | Adjusts learning rates for each parameter. | Well-suited for sparse data and problems with varying gradients. | May accumulate squared gradients, causing the learning rate to shrink excessively. | Avoid for very deep models as the learning rate may become too small. |
| **Adam** | Adaptive Moment Estimation | Combination of Momentum and Adagrad, balances the benefits. | Widely used for deep learning tasks due to its effectiveness. | More complex than other algorithms. | Avoid for simpler models or small datasets due to higher complexity. |

**Gradient Descent Linear Regression:**

**Problem Solved**: Minimizing the Cost Function in Linear Regression.

**How It Works**: Gradient Descent is an iterative optimization algorithm used to find the minimum of the cost function in linear regression. It calculates the gradient of the cost function with respect to the model parameters and updates the parameters in the opposite direction of the gradient to minimize the cost. This process continues iteratively until convergence, resulting in the optimal parameters for the linear regression model.

**Mini Batch Gradient Descent**:

**Problem Solved**: Faster Convergence, Better Generalization.

**How It Works**: Mini Batch Gradient Descent is a variation of Gradient Descent that updates the model parameters using a small batch of data instead of the entire dataset. It provides faster convergence and better generalization by reducing the noise in parameter updates. The algorithm computes the gradient for each mini-batch and updates the parameters accordingly, allowing for efficient optimization, especially in large datasets and neural network training.

**Momentum**:

**Problem Solved**: Accelerating Convergence.

**How It Works**: Momentum is an optimization technique that introduces a moving average of past gradients to accelerate convergence. It helps the algorithm to continue moving in the same direction, overcoming slow convergence. By accumulating the past gradients, Momentum gains momentum in the parameter updates and helps avoid getting stuck in shallow local minima.

**Adagrad**:

**Problem Solved**: Adaptive Learning Rates.

**How It Works**: Adagrad adapts the learning rate for each parameter based on the historical gradient information. It gives more weight to less frequently updated parameters and less weight to frequently updated parameters. This adaptive learning rate allows Adagrad to handle sparse data and problems with varying gradients effectively. However, over time, it may accumulate squared gradients, causing the learning rate to shrink excessively, which can be problematic for very deep models.

**Adam**:

**Problem Solved**: Adaptive Moment Estimation.

**How It Works**: Adam combines the benefits of Momentum and Adagrad by maintaining a moving average of past gradients and their squares. It balances the adaptive learning rates and momentum to efficiently optimize parameters. The algorithm uses biased estimates of the first and second moments of the gradients, making it particularly effective for deep learning tasks. However, Adam is more complex than other algorithms and may not show significant improvements for simpler models or small datasets due to its higher complexity.