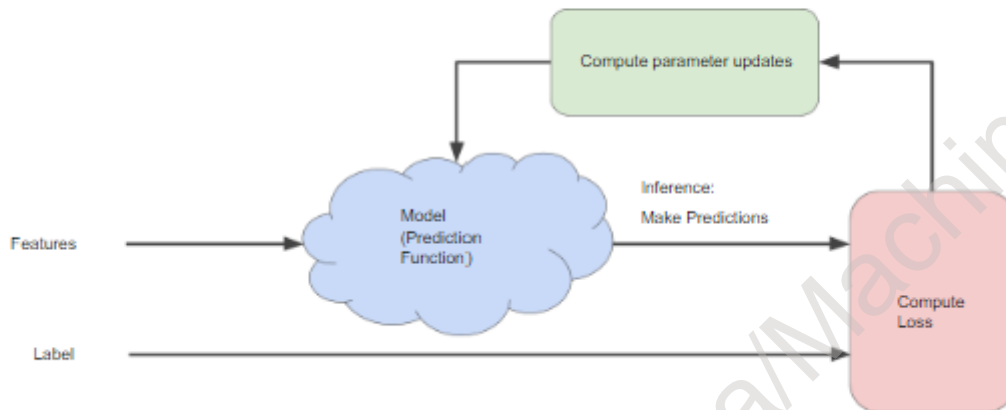


ML Concepts - Reducing Loss

So how do we reduce loss?

Block Diagram of Gradient Descent



- In the above diagram, an iterative approach is used to train models. This means that the process involves repeating a series of steps, making adjustments along the way to improve the model's performance.
- Within this iterative process, special attention is given to the "Model (Prediction Function)" part. This is because it contains various complexities that need to be addressed during the training process.
- Iterative strategies are commonly used in machine learning because they work well with large datasets. They allow the model to adapt and improve over time.
- The "model" is a key component. It takes one or more features as input and produces a prediction as output. For simplicity, let's consider a model that takes only one feature and returns one prediction.

$$y' = b + w_1 x_1$$

- Two parameters, denoted as b and w_i , need initial values. For linear regression problems, the specific starting values are not critical. In this example, we set them to 0.
- Let's say the first feature value is 10. This value is plugged into the prediction function to get an initial prediction.
- The "Compute Loss" part involves using a loss function. In this case, suppose we use the squared loss function. The loss function takes two inputs:
 - Prediction: The model's output based on the features (x)
 - Correct Label: The actual label corresponding to the features (x)
- The loss function quantifies the difference between the model's prediction and the actual label. It helps guide the model towards making better predictions.
- In the Compute step, the machine learning system evaluates the value of the loss function and calculates new values for b and w_i .

- The learning process continues in a loop. The model's predictions are refined, and new parameter values are calculated. This continues until the algorithm finds parameter values that result in the lowest possible loss.
- Typically, you keep iterating until the overall loss stops changing significantly or changes extremely slowly. This indicates that the model has converged to a stable state.

What is weight initialization?

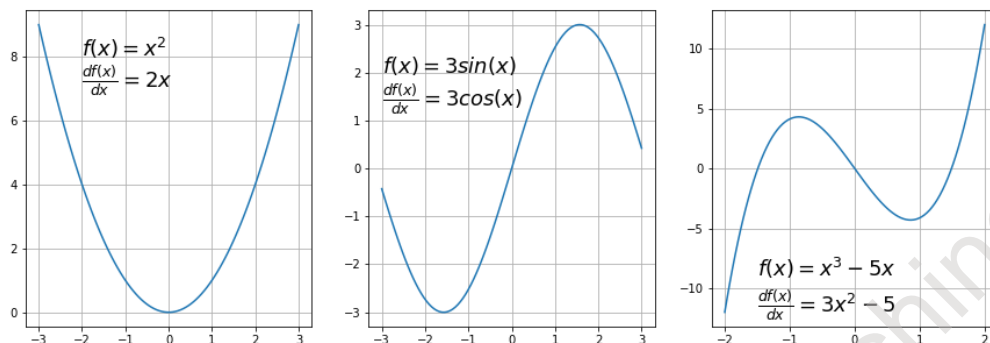
- Weight initialization refers to the method of setting initial weights before training. The choice of initial weights can significantly impact the learning process and the final performance of the model. It matters because:
 - the choice of initial weights can influence the path the optimization algorithm takes during training. Poorly selected initial weights might result in slow convergence or getting stuck in local minima.
 - Also different weight initializations can lead to different final models even if trained on the same data. The initial weights can influence the model's ability to generalize to unseen data.
- **For convex problems:** In convex problems, the shape of the loss function is akin to a bowl, having a single minimum. Therefore, the initial weights, even if set as all zeros, don't impact the optimization process significantly due to the single minimum point in the loss landscape.
- **Convex nature:** The convex nature implies that regardless of the starting point, the optimization process converges to the global minimum, making weight initialization less critical in these scenarios.
- **Non-convex problems:** Contrarily, in non-convex problems such as those encountered in neural networks, the loss landscape resembles an egg crate, with multiple minimum points.
- **Multiple minimums:** The presence of multiple local minimums in non-convex problems indicates that the choice of initial weights greatly influences the optimization process. It can lead the model to converge towards different local minima based on the initial weight values.

What is Gradient Descent?

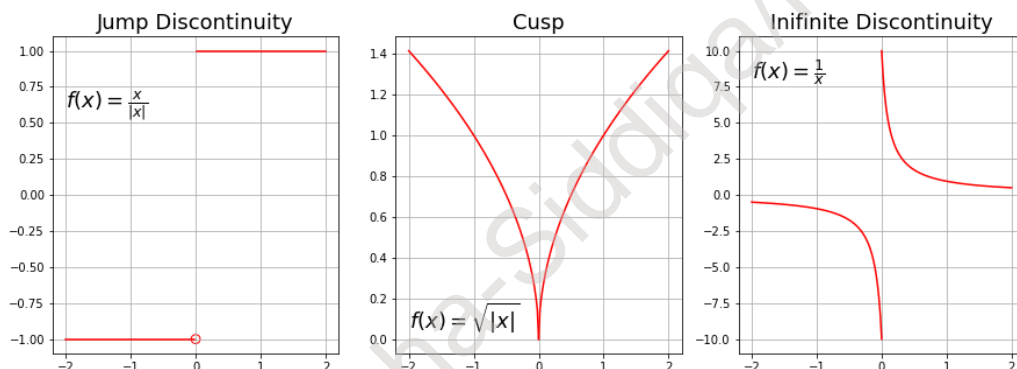
- Gradient descent is an optimization algorithm widely employed for training machine learning models and neural networks.
- Training data is crucial for allowing these models to learn and improve their performance over time.
- The cost function in gradient descent serves as a measure of accuracy, assessing how well the model is performing with each round of parameter updates.
- The objective of gradient descent is to minimize the cost function, aiming for it to be as close to zero as possible.
- The algorithm iteratively adjusts the model's parameters to achieve the smallest possible error, based on the feedback provided by the cost function.
- Once machine learning models are optimized for accuracy, they become powerful tools with a wide range of applications in artificial intelligence (AI) and computer science.
- Gradient descent algorithm does not work for all functions. There are two specific requirements. A function has to be:

1. Differentiable

- First, what does it mean it has to be **differentiable**? If a function is differentiable it has a derivative for each point in its domain — not all functions meet these criteria. First, let's see some examples of functions meeting this criterion:



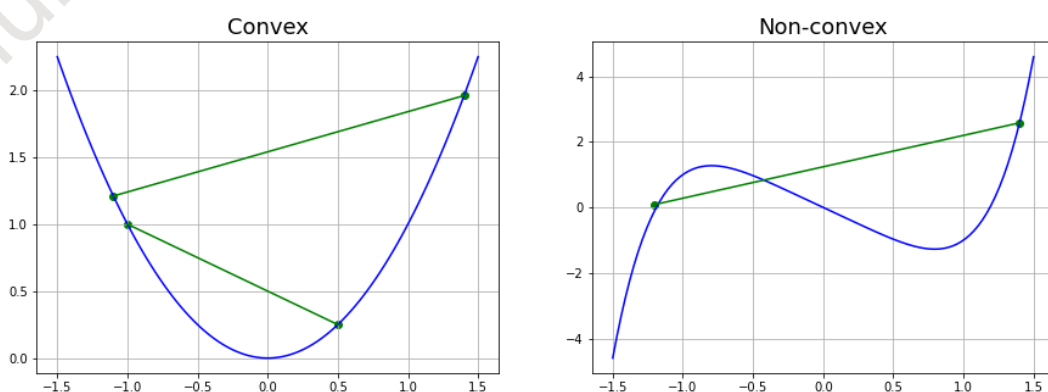
Examples of differentiable functions



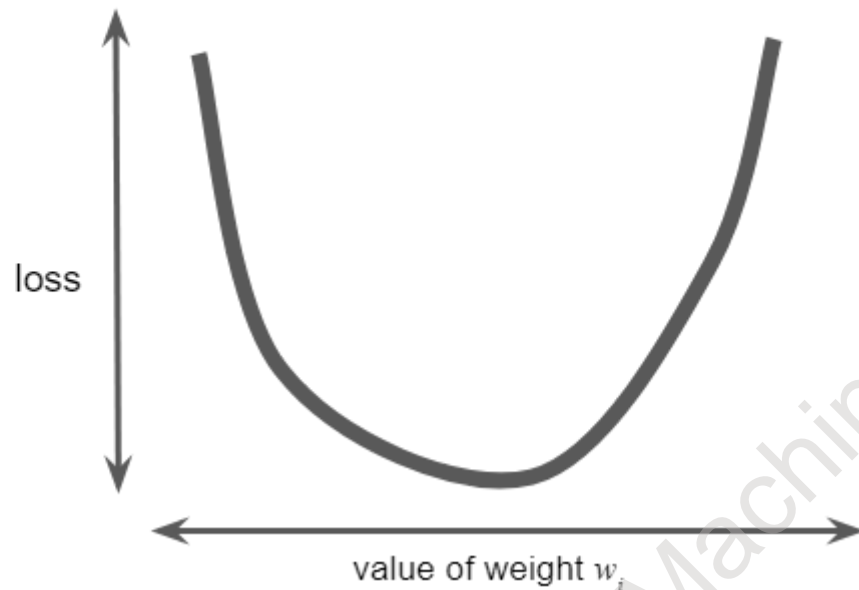
Typical non-differentiable functions have a step a cusp or a discontinuity:

2. Convex:

- For a univariate function, this means that the line segment connecting two function's points lays on or above its curve (it does not cross it). If it does it means that it has a local minimum which is not a global one.
- Another way to check mathematically if a univariate function is convex is to calculate the second derivative and check if its value is always bigger than 0.



***How does Gradient Descent work?**



Step 1: Starting Point

- Begin by selecting an arbitrary starting point for evaluating performance.

Step 2: Calculating the Derivative (Slope)

- Calculate the derivative (slope) at this starting point.

Step 3: Observing Slope Steepness

- The slope indicates how steep the curve is, guiding updates to the model's parameters (weights and bias).

Step 4: Minimizing the Cost Function

- Aim to minimize the cost function, which measures the error between predicted and actual values, similar to finding the line of best fit in linear regression.

Step 5: Direction and Learning Rate

- Consider two factors: direction and learning rate.
- Direction shows where to make adjustments, and learning rate determines the size of the steps towards the minimum.

Step 6: Adjusting Parameters

- Utilize the derivative and learning rate to update the model's parameters, gradually reducing the error.

Step 7: Convergence Point

- Continue the process, with the slope gradually decreasing until it reaches the lowest point on the curve, known as the point of convergence.

Step 8: Learning Rate Considerations

- Choose the learning rate carefully. A high rate allows for larger steps but may overshoot the minimum, while a low rate offers more precision but might require more iterations.

Step 9: Cost (Loss) Function

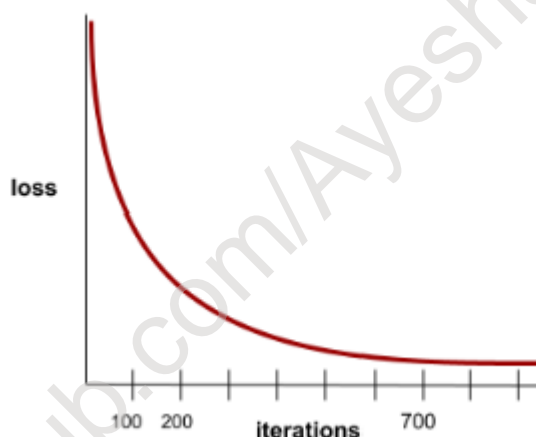
- Measure the error between actual and predicted values at the current position.
- Guide the model's adjustments to minimize error and find the local or global minimum.

Step 10: Continual Iteration

- Iterate, moving in the direction of steepest descent (negative gradient), until the cost function is close to zero, indicating that the model has learned as much as it can.

*Explain Convergence.

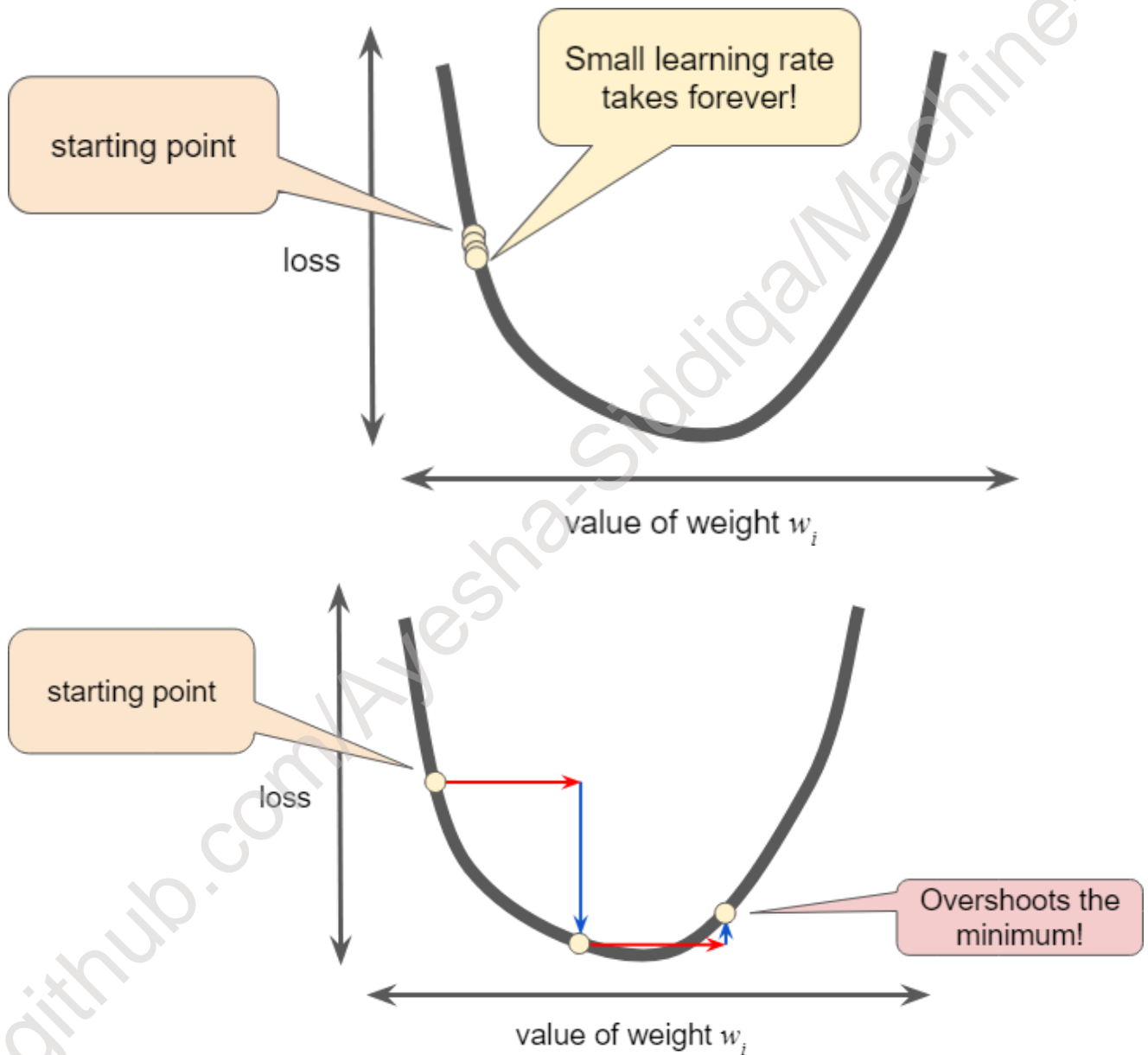
- It's a state where the loss values change very little or not at all with each iteration.
- Think of it as a point where the model's performance levels off and doesn't get significantly better.
- When the loss curve starts to flatten out, it suggests that the model is converging and when further training won't significantly improve its performance.

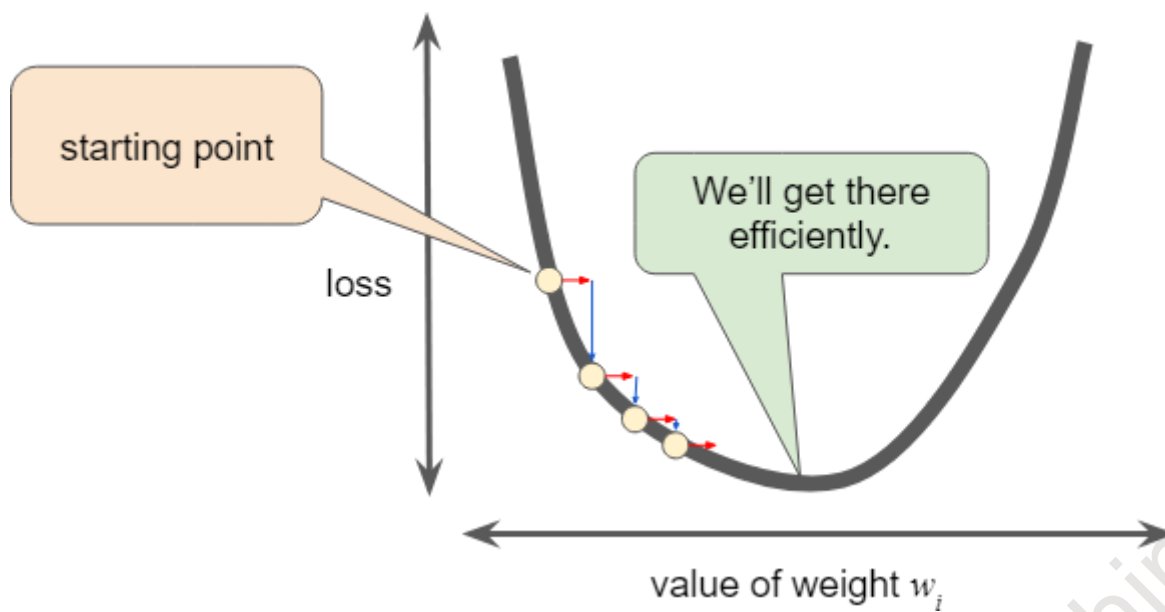


What is the role of learning rate?

- **Hyperparameters** are the adjustable settings that programmers tune during successive runs of training a model.
- Learning rate is one such hyperparameter. It's a floating-point number that tells the gradient descent algorithm how strongly to adjust weights and biases on each iteration.
- **Adjusting Learning Rate:** For instance, you might set the learning rate to 0.01 for one training session. If you find it too high, you might change it to 0.003 for the next session.

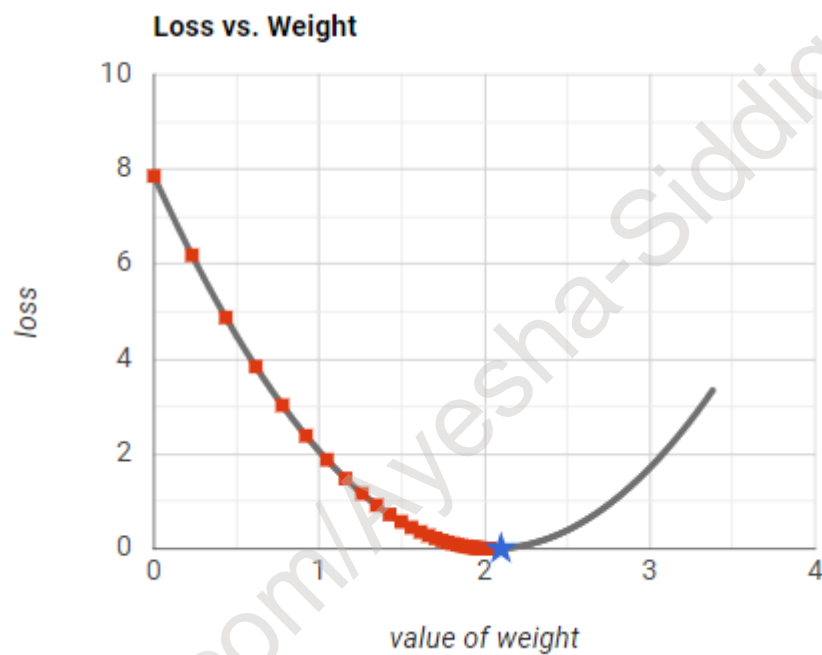
- During each iteration, the gradient descent algorithm multiplies the learning rate by the gradient, yielding the gradient step. Setting it too low can lead to prolonged training, while setting it too high can impede convergence. If the learning rate is too small, the learning progress is extremely slow towards the minimum point. Conversely, if the learning rate is too large, the points may bounce unpredictably and struggle to converge.
- There's an optimal learning rate for each regression problem, depending on the flatness of the loss function.
- **Ideal Learning Rate:** In one dimension, the ideal learning rate is the inverse of the second derivative of the function. In two or more dimensions, it's the inverse of the Hessian (matrix of second partial derivatives).



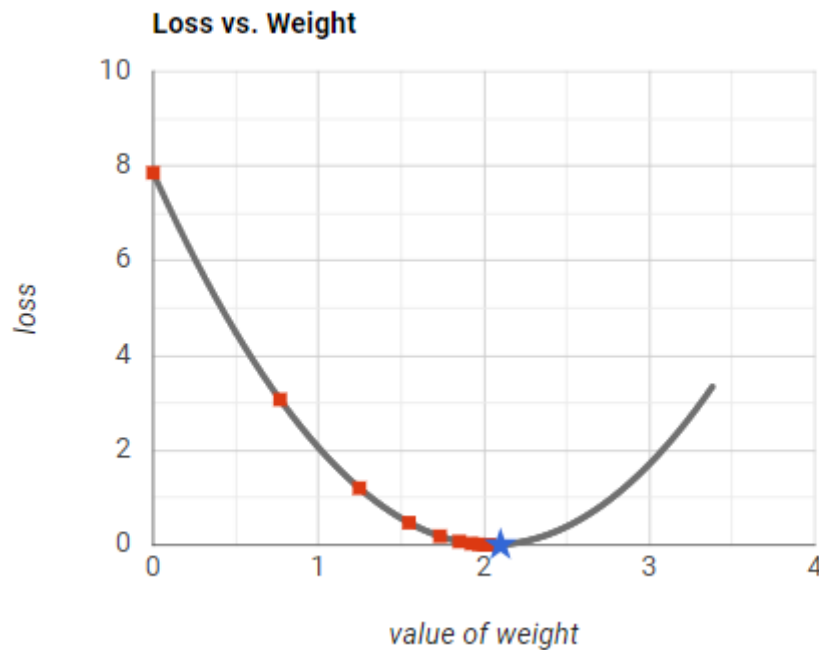


Now let's visualize how many steps it actually takes to reach the min with different learning rates

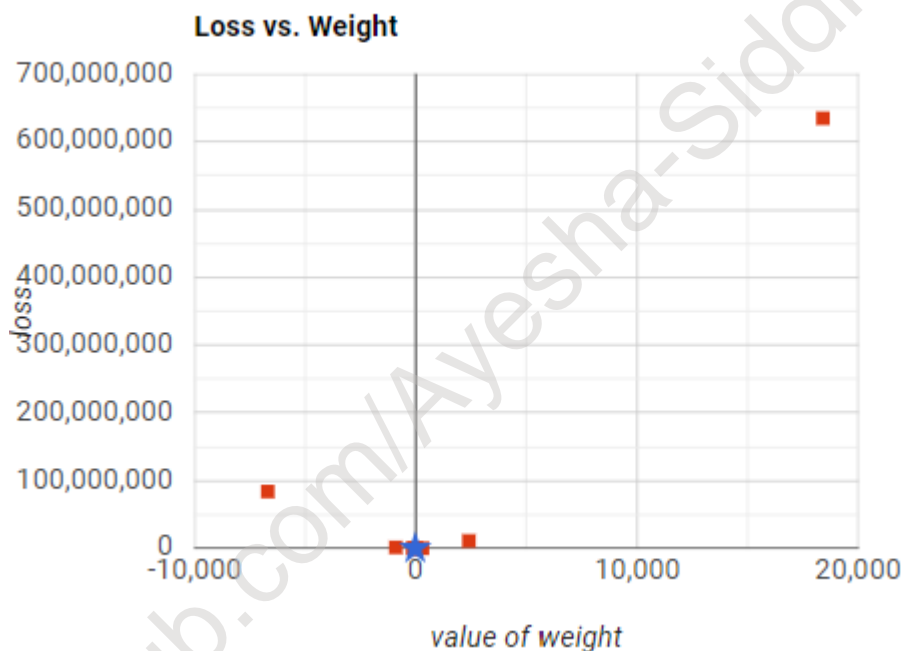
Learning rate = 0.03, steps taken = 40



Learning rate = 0.10, steps taken = 11



Learning rate = 1.0. Gradient Descent will never reach the minimum.




How do Stochastic Gradient Descent (SGD) and Mini-Batch Gradient Descent differ in their approaches to updating model parameters during training in machine learning?

- **SGD (Stochastic Gradient Descent):** This optimization algorithm involves computing and applying gradients to the model's parameters based on one example at a time from the training dataset. It's efficient but can be noisy due to frequent updates influenced by individual samples. However, it helps avoid unnecessary computations over the entire dataset and often converges faster in large datasets.

- **Mini-Batch Gradient Descent:** This approach lies between the extremes of SGD and full-batch gradient descent. It involves computing gradients on batches of data samples (typically ranging from 10 to 1000). Instead of considering one sample at a time, the loss and gradients are averaged over these batches, reducing the noise associated with SGD while maintaining the computational efficiency compared to full-batch gradient descent.
- **Efficiency in Computing Gradients:** Instead of computing the gradient over the entire dataset, both SGD and mini-batch gradient descent compute gradients over smaller subsets, making it computationally efficient. This approach works well in practice and is more feasible than processing the entire dataset at every iteration.

When performing gradient descent on a large data set, which of the following batch sizes will likely be more efficient?

A small batch or even a batch of one example (SGD). 

Amazingly enough, performing gradient descent on a small batch or even a batch of one example is usually more efficient than the full batch. After all, finding the gradient of one example is far cheaper than finding the gradient of millions of examples. To ensure a good representative sample, the algorithm scoops up another random small batch (or batch of one) on every iteration.

Correct answer.

The full batch. 