OPTIM

THIS BOOK

This book is written (typed) by Ari, who hails from the South and has keen interests in Computer Science, Biology, and Tamil Literature. Occasionally, he updates his website, where you can reach out to him.

https://arihara-sudhan.github.io

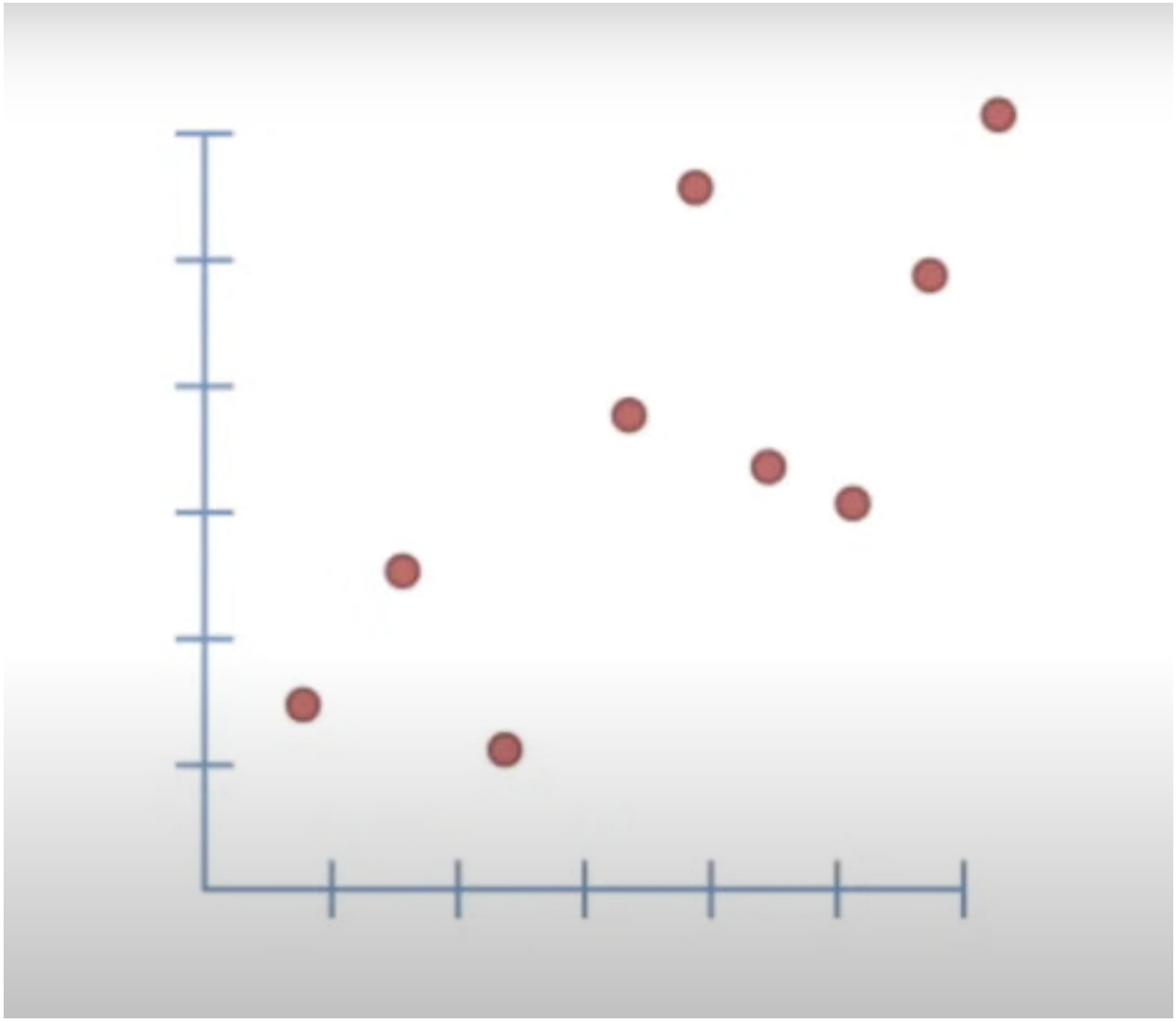
PREREQUISITES

To get the most out of this book, you should be familiar with some basics, which you can learn from the following book.

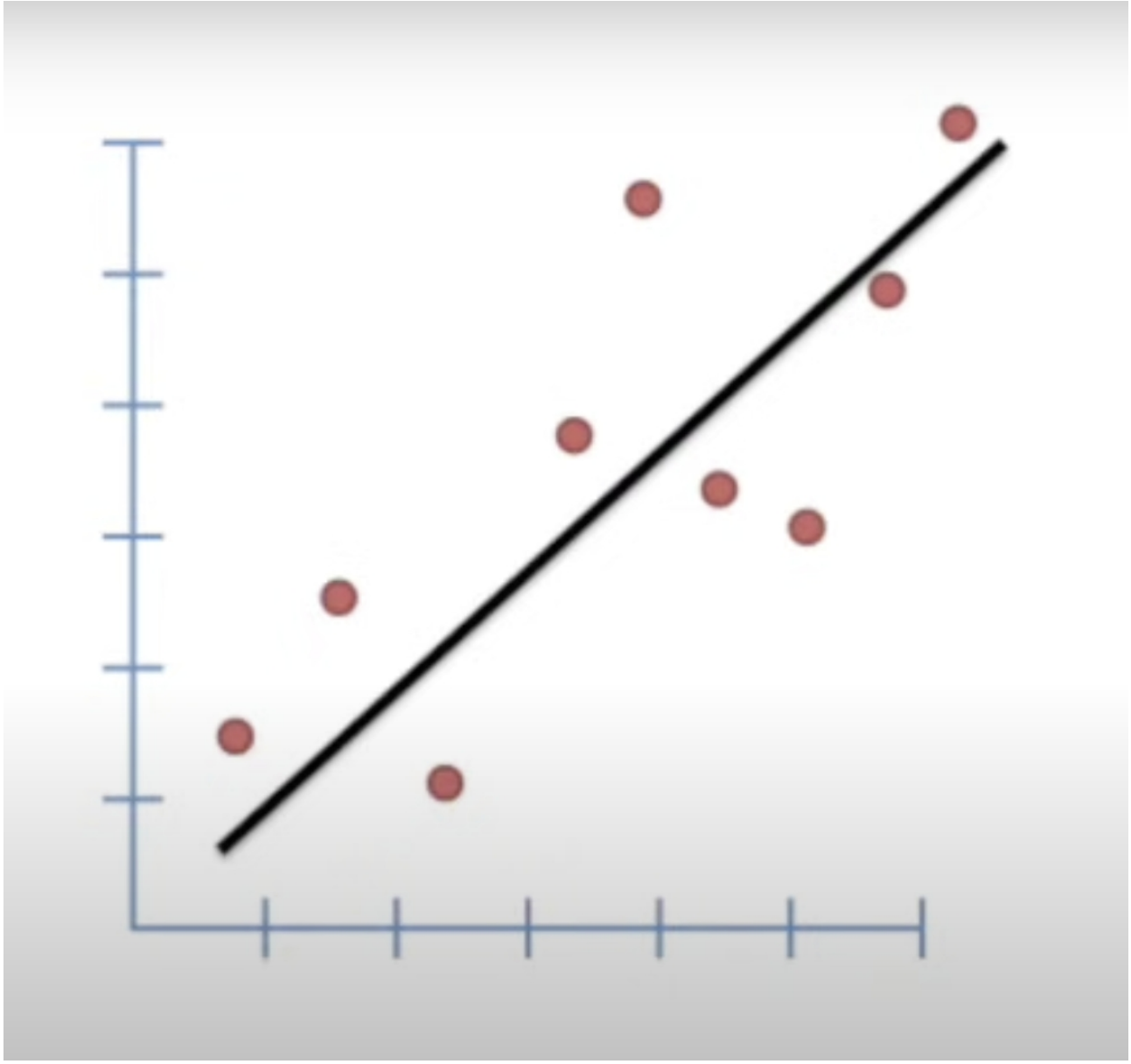
MLP

FITTING A LINE TO DATA

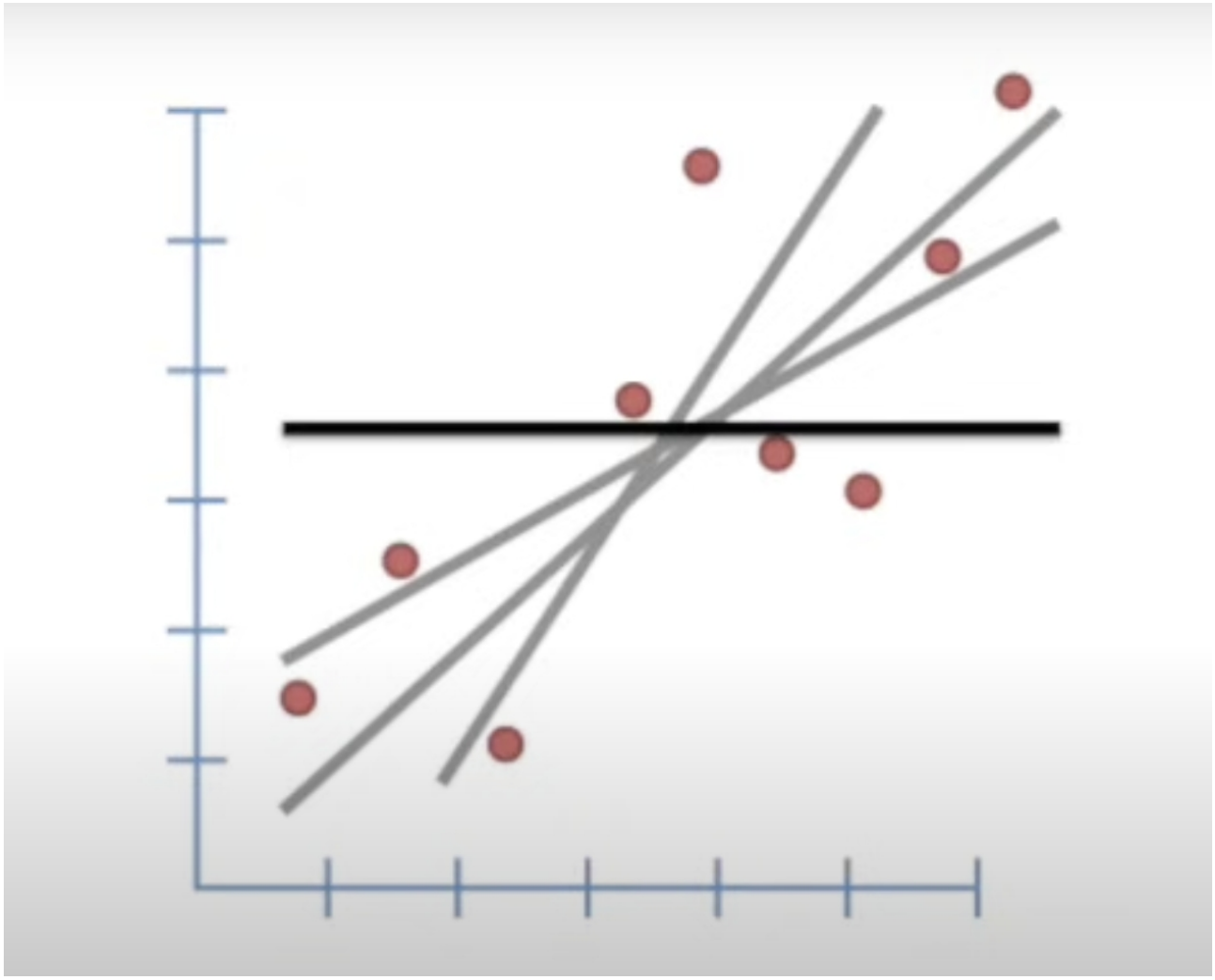
Assume we do have a data distribution as shown below.



Initially, the weights and biases makes a line as in the following figure.



Now, We want to check whether this is a perfect fit or not. A perfect fit makes the loss low. So, it is the place for experiments. We can tweak and tune the parameters and check with different lines.

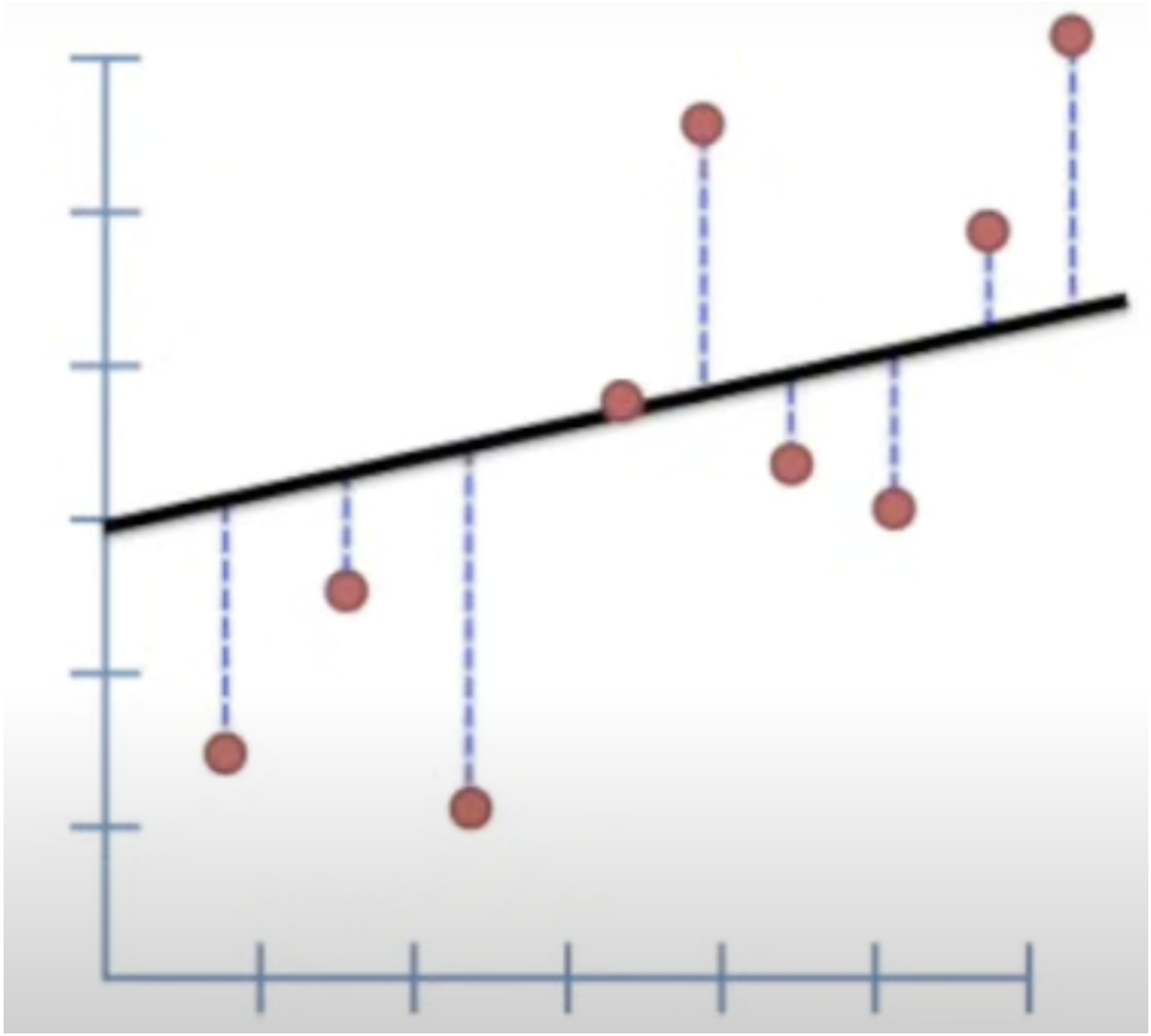


Let’s consider that horizontal line (no slope). What could be the distance to the data points from this line? Since the Y magnitude represents bias (intercept), let’s measure the difference with accordance to that.

$$(b-y_1)^2+(b-y_2)^2+(b-y_3)^2+(b-y_4)^2+(b-y_5)^2+(b-y_6)^2+(b-y_7)^2+(b-y_8)^2+(b-y_9)^2$$

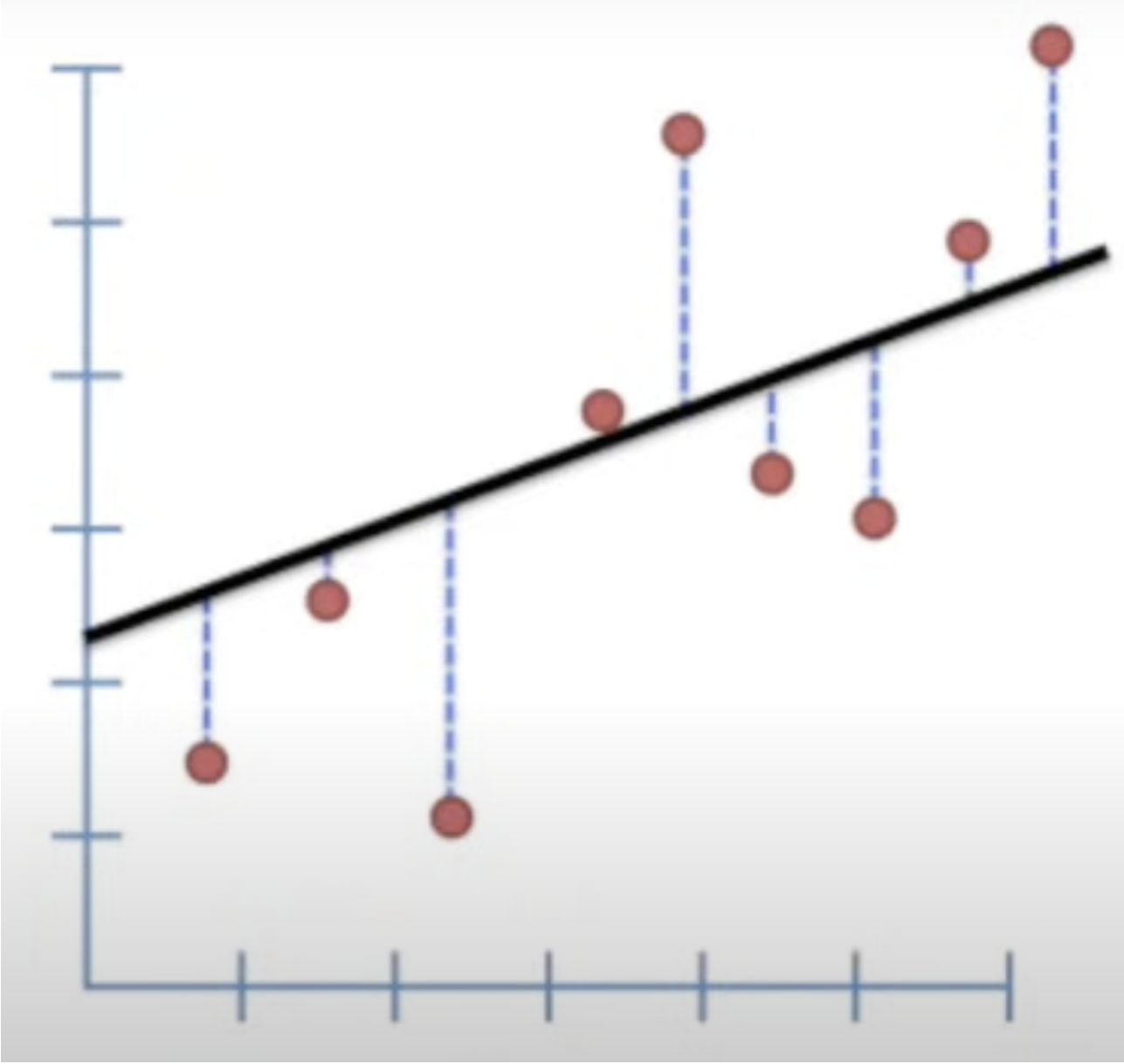
The answer for this is 25.

Let us tilt the line and try once again.



Now, the sum is a way lesser than the previous one. It is 19. I would like to make this sum even more lesser.

So, let’s tweak the parameters again.



Now, the sum is 14. Yes! It could be a better fit than the all previous ones.

This method of finding the appropriate parameter values for the smallest sum of squares is called, “Least Squares”. However, The expression in general mathematical term would be like the following.

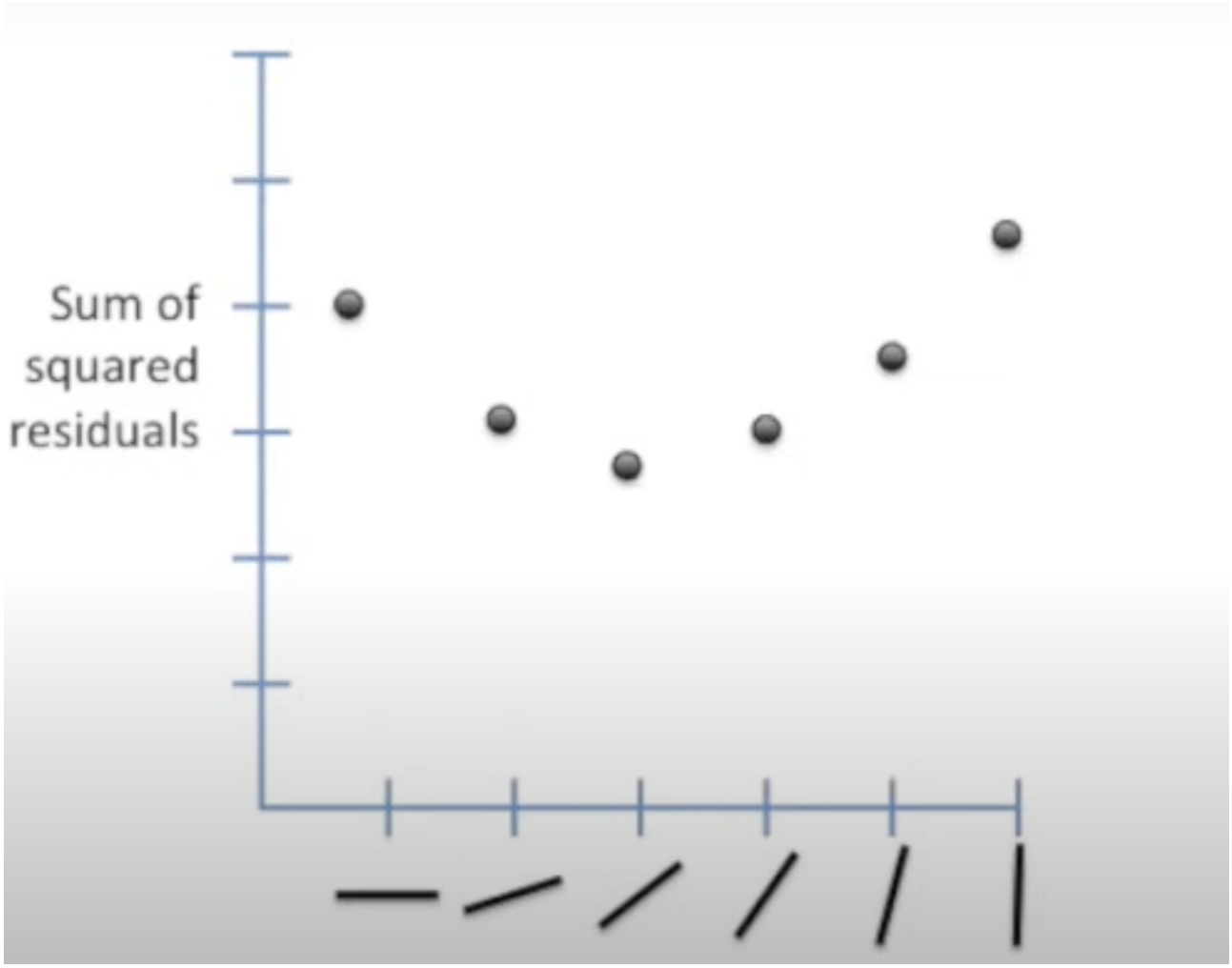
$$\text{Sum of squared residuals} = ((a^*x_1 + b) - y_1)^2 + ((a^*x_2 + b) - y_2)^2 + \ldots$$

We find values for a (weight) and b (intercept). Now, how do we find the optimal line? How do we fit the line to data?

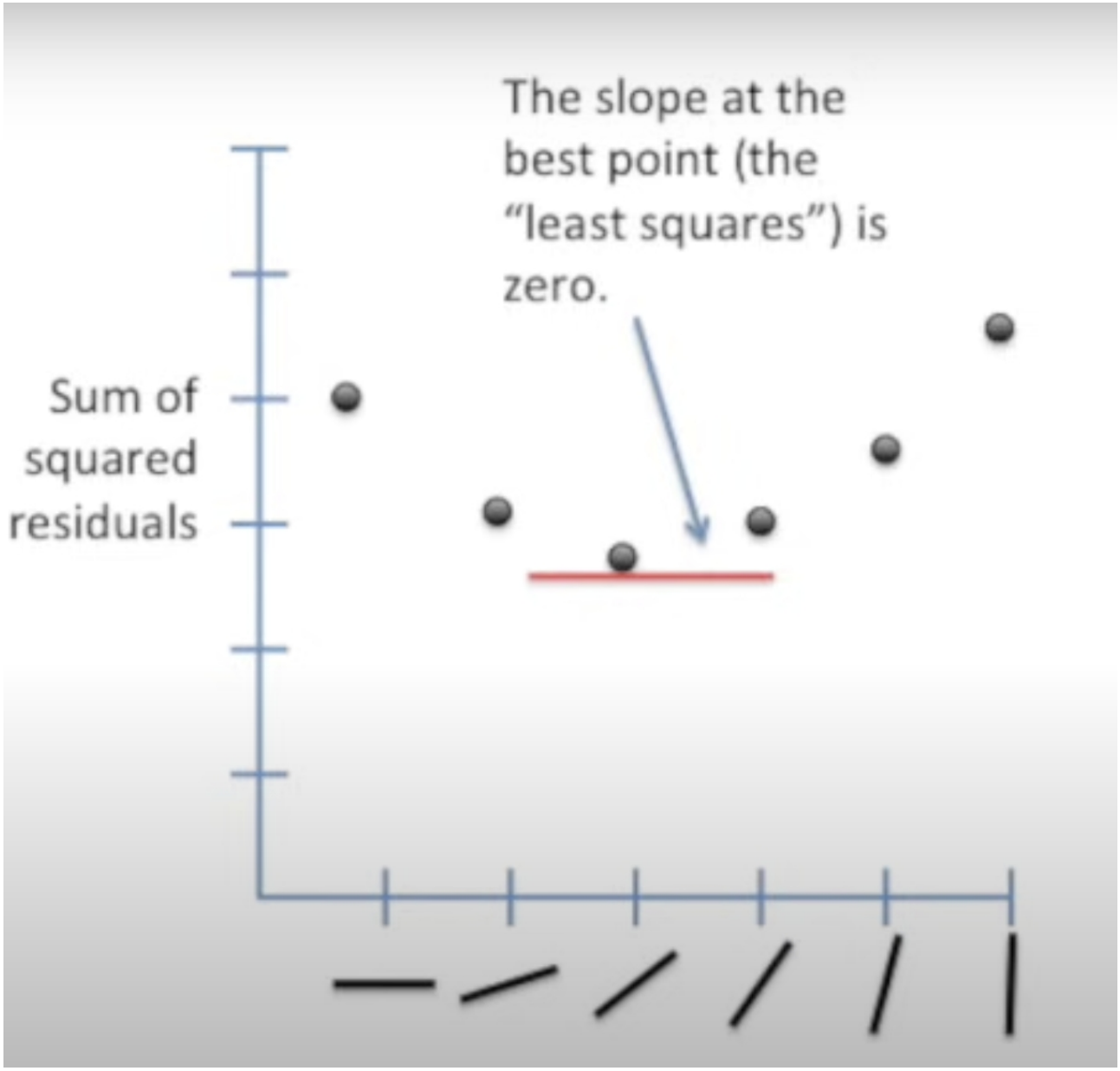
GRADIENT

The gradient refers to the vector of partial derivatives of a function with respect to its parameters. It's like the slope of the function at a certain point, and it tells us the direction and rate at which the function's value increases or decreases. In one dimension, Imagine a hill with a path leading up to the peak. If Ari stands at a certain point on the path, the slope tells him whether he is going up or down. A positive slope means he is going up, a negative slope means he is going down, and a slope of zero means he is at a flat point (possibly a peak or a valley). In calculus, this slope is the derivative of the function.

If he wants to find the minimum point of a function (like reaching the valley), he would look at how the slope changes and move opposite to it. In machine learning, our functions are often defined over many dimensions (like many features in a dataset). The gradient in this multi-dimensional space tells us the direction of steepest ascent. It is represented as a vector of partial derivatives, where each element represents the rate of change along one dimension. If we want to minimize a function (say, reduce error in a model), we take a step opposite to the gradient direction, because this leads us downhill toward lower values.



In the figure above, for the first chosen parameters, it is so steepest! For the second also, it is somewhat steepest.. But for the middle(3rd) one, it seems to be less steep.



As we have discussed there in "MLP" book, it is all about derivatives. Taking derivative at a point tells us how steepy that point is.

BATCH GRADIENT DESCENT

Batch Gradient Descent or Gradient Descent is an optimization algorithm used to minimize the loss function of a model. It works by iteratively adjusting the model's parameters (weights and biases) in the direction opposite to the gradient of the loss function. This helps reduce the loss and improves the model's performance. It calculates the gradient of the loss function with respect to each parameter (weight or bias). Following is the parameter updating rule.

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta)$$

$\nabla_\theta J(\theta)$ is the gradient of the loss function with respect to the parameters.

STOCHASTIC GD

Gradient Descent (GD) has several drawbacks, including the risk of getting stuck in local minima or saddle points, especially in non-convex functions, which can lead to suboptimal solutions. It is sensitive to the learning rate, requiring careful tuning to avoid slow convergence or instability.

Stochastic Gradient Descent is a variant of Gradient Descent where the parameters are updated using only a single training example at a time, rather than the entire dataset. SGD is more frequent and faster, but the updates are noisier and less stable. Over time, this noise can help the algorithm escape local minima and explore more of the solution space.

$$\theta = \theta - \alpha \cdot \nabla_\theta J(\theta; x^{(i)}, y^{(i)})$$

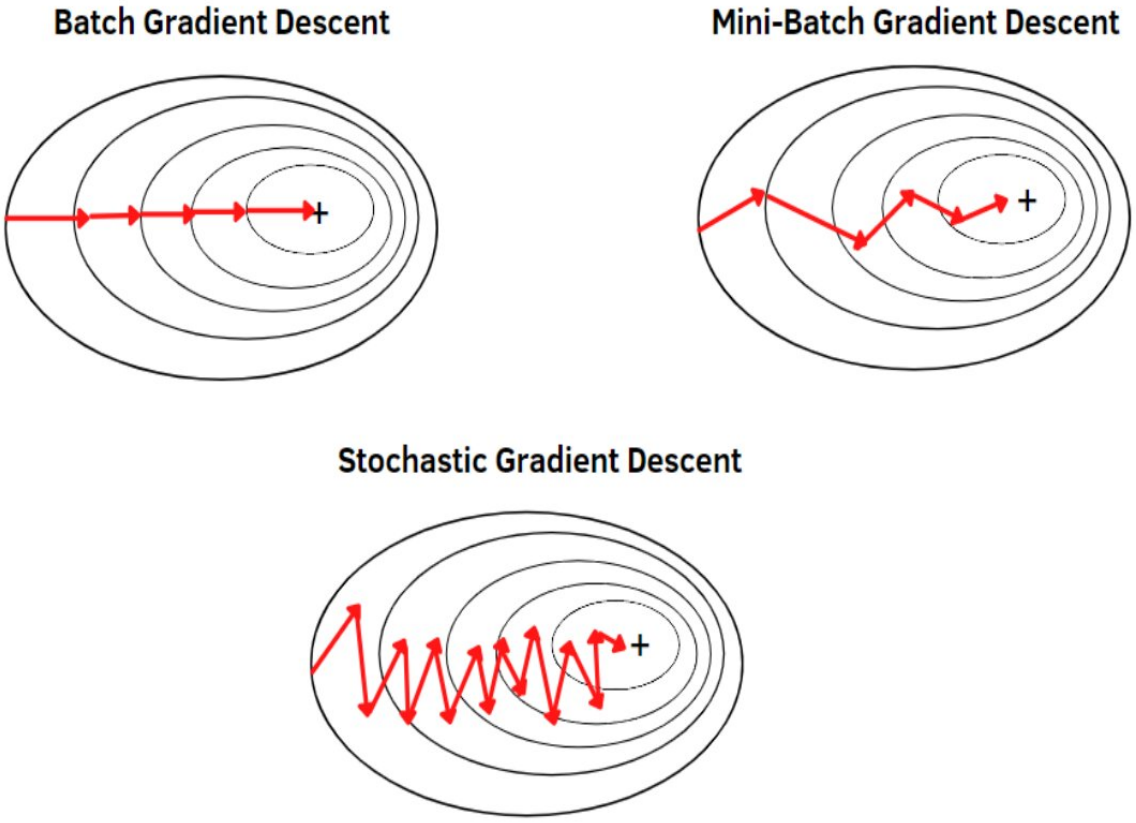
MINI BATCH GD

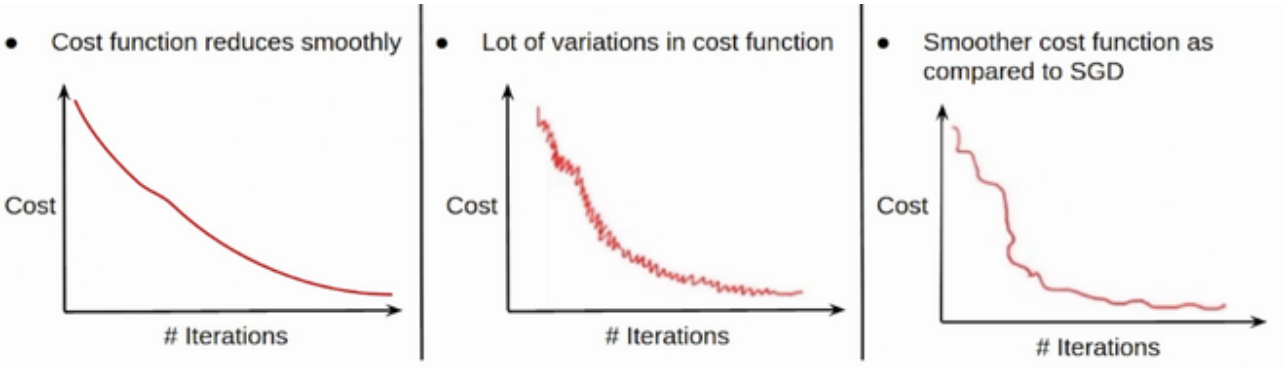
Mini-Batch Gradient Descent is a compromise between Batch Gradient Descent and Stochastic Gradient Descent. In this approach, instead of using the entire dataset or just a single example, we compute the gradient using a small batch of training examples. We split the training data into small batches (e.g., 32 or 64 samples per batch), and then we calculate the gradient for each batch and update the parameters accordingly. This reduces the computational cost compared to Batch Gradient Descent, while still providing more stable update.

The update rule is as the following:

$$\theta = \theta - \alpha \cdot \nabla_{\theta} J(\theta; X_{batch}, Y_{batch})$$

Where X_{batch} and Y_{batch} are the features and labels of the mini-batch.





LET'S LIGHT ON IT

Let's define the network first.

```python
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt

X = torch.tensor([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])
y = torch.tensor([[0.0], [1.0], [1.0], [0.0]])

class XORModel(nn.Module):
    def __init__(self):
        super(XORModel, self).__init__()
        self.hidden = nn.Linear(2, 4)
        self.output = nn.Linear(4, 1)
        self.sigmoid = nn.Sigmoid()
    def forward(self, x):
        x = torch.relu(self.hidden(x))
        x = self.sigmoid(self.output(x))
        return x

```

The way we pass the data matters here.

```python
def train_with_batch_gd(model, optimizer, X, y, epochs=500, lr=0.1):
    criterion = nn.BCELoss()
    optimizer = optimizer(model.parameters(), lr=lr)
    losses = []
    for epoch in range(epochs):
        outputs = model(X)
        loss = criterion(outputs, y)
        losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return model, losses
```

```python
def train_with_sgd(model, optimizer, X, y, epochs=500, lr=0.1):
    criterion = nn.BCELoss()
    optimizer = optimizer(model.parameters(), lr=lr)
    losses = []
    for epoch in range(epochs):
        for i in range(len(X)):
            x_sample = X[i:i+1]
            y_sample = y[i:i+1]
            outputs = model(x_sample)
            loss = criterion(outputs, y_sample)
            losses.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
    return model, losses

def train_with_mini_batch_gd(model, optimizer, X, y, batch_size=2, epochs=500, lr=0.1):
    criterion = nn.BCELoss()
    optimizer = optimizer(model.parameters(), lr=lr)
    losses = []
    for epoch in range(epochs):
        for i in range(0, len(X), batch_size):
            x_batch = X[i:i+batch_size]
            y_batch = y[i:i+batch_size]
            outputs = model(x_batch)
            loss = criterion(outputs, y_batch)
            losses.append(loss.item())
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()
    return model, losses
```

SGD in PyTorch is more flexible and can work for both Mini-Batch GD and full Batch GD by adjusting the batch_size. The underlying optimization algorithm remains the same (stochastic in nature).

Full Code is HERE

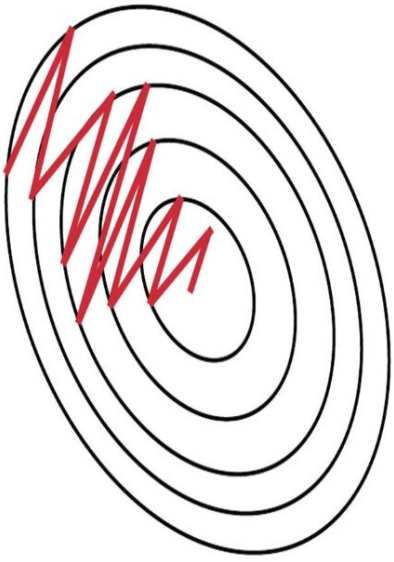
GD WITH MOMENTUM

In basic gradient descent, the weights are updated by moving in the direction of the negative gradient (the direction that decreases the cost function), scaled by the learning rate. The idea of momentum is inspired by physics, where an object in motion tends to continue in its direction unless acted upon by a force. Similarly, momentum helps to "carry" the update in the same direction over time, even if the gradient is small. Instead of just using the gradient at the current step, we use a combination of the current gradient and the previous update.

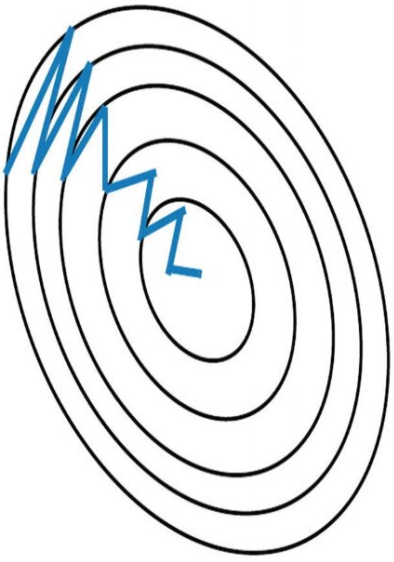
$$v_t = \beta v_{t-1} + (1-\beta)\nabla_\theta J(\theta)$$

$$\theta = \theta - \eta v_t$$

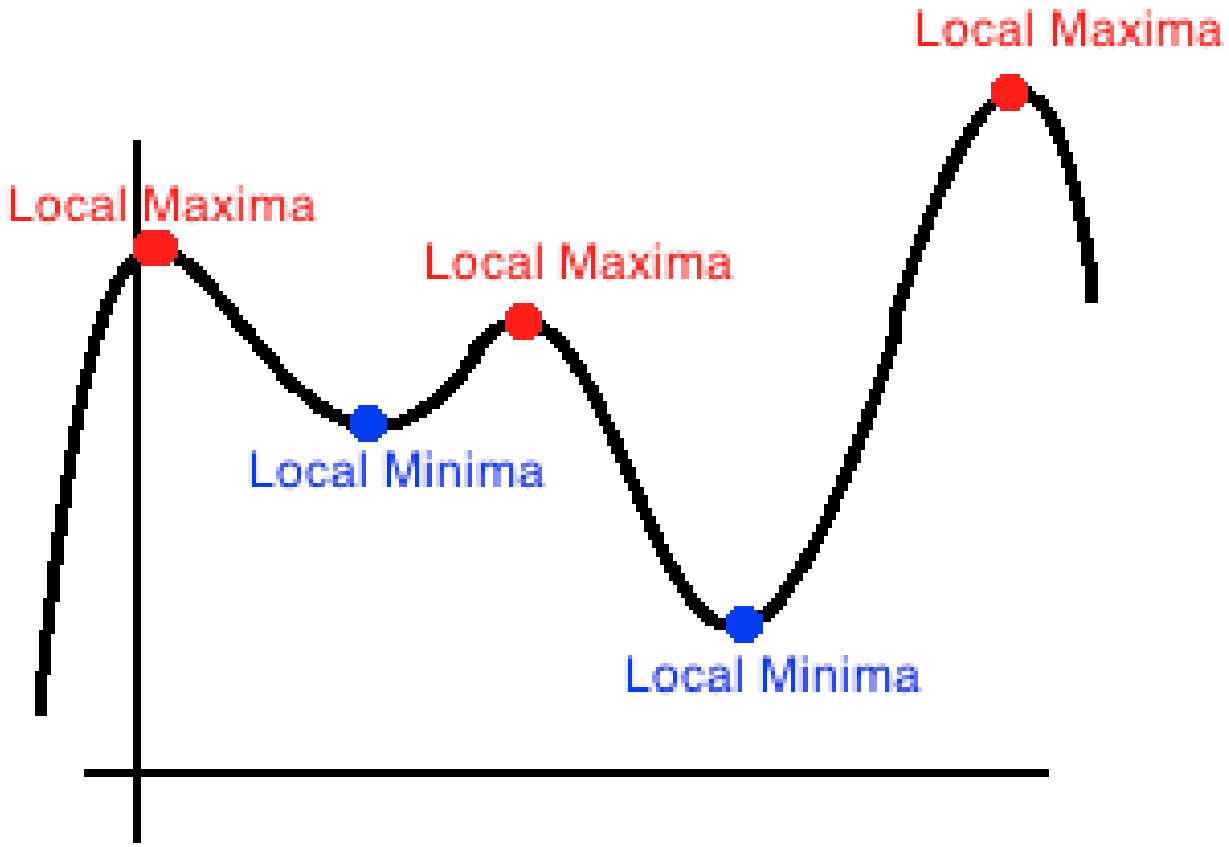
The velocity, v is the exponentially weighted moving average of the past gradients. It accumulates the gradient's influence over time. β is the momentum coefficient which determines how much of the previous velocity is retained. A typical value for β is between 0.9 and 0.99. By incorporating the past gradients, momentum smooths out the noisy fluctuations, leading to more stable convergence. Momentum helps the algorithm accelerate in the direction of consistent gradients and dampen oscillations, especially in areas with steep gradients.



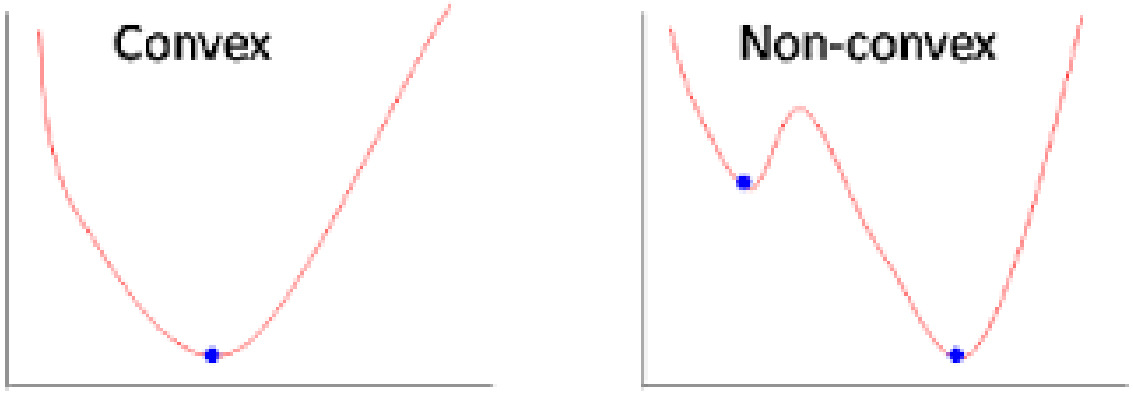
Stochastic Gradient Descent **without** Momentum



Stochastic Gradient Descent **with** Momentum



Gradient descent with momentum helps overcome the challenge of getting stuck in local minima by adding a momentum term to the standard gradient descent update rule. There's no absolute guarantee that momentum-based gradient descent will find the global minimum, especially in complex, non-convex landscapes.



NESTEROV ACCELARATED

Nesterov Accelerated Gradient is a modification of traditional momentum-based gradient descent where, instead of just adding momentum, it anticipates the direction of the next step by looking slightly ahead in the parameter space. In standard gradient descent with momentum, the update step simply accumulates the previous gradients. But in NAG, the gradient is computed at a look-ahead position, meaning we compute the gradient not at the current position θ but at θ+βv, where v is the momentum term.

$$v_t = \beta v_{t-1} + \alpha \nabla J(\theta + \beta v_{t-1})$$

$$\theta = \theta - v_t$$

While this can help with escaping shallow local minima or speeding up convergence, it does not inherently guarantee that you'll avoid deep or well-formed local minima. The challenge with local minima is inherent in the non-convex nature of the optimization landscape.

Gradient Descent Update Rule

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

Momentum based Gradient Descent Update Rule

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

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

ADAGRAD

Adaptive Gradient Algorithm adapts the learning rate for each parameter individually based on how frequently it is updated. Each parameter has its own learning rate that changes over time. Parameters updated frequently receive lower learning rates, while those updated infrequently get higher learning rates. This adjustment helps Adagrad handle sparse features or parameters well, making it useful in cases where some features (or weights) rarely occur in the data. Adagrad keeps track of the sum of squares of the gradients for each parameter over time. This is called the accumulated squared gradient.

Adagrad

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

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

In Adagrad, there happens learning rate decay. Adagrad calculates the learning rate for each parameter based on the cumulative sum of squared gradients for that parameter. As training progresses, v_t becomes very large because it continuously accumulates squared gradient values, causing Square root of v_t to increase significantly over time. When it grows, learning rate becomes smaller and smaller.

RMS PROP

Root Mean Square Propagation is an adaptive learning rate optimization algorithm designed to address the vanishing learning rate problem that arises in Adagrad. RMSprop adjusts the learning rate by calculating an exponentially decaying average of squared gradients rather than accumulating the squared gradients over time like Adagrad.

RMSProp

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

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

The result is an adaptive learning rate for each parameter based on the recent history of gradients, which prevents the lr from becoming too small or too large.

By using an exponentially decaying average of squared gradients, RMSprop keeps learning steady over time, making it suitable for deep learning tasks. Unlike Adagrad, RMSprop avoids the problem of a rapidly shrinking learning rate, which allows the algorithm to continue updating parameters effectively over time.

Wait for Adam!

ADAM

Adam (Adaptive Moment Estimation) is an optimization algorithm that combines the advantages of two popular optimizers which are Momentum and RMSprop. It's widely used in deep learning because it adapts the learning rate for each parameter and incorporates an adaptive momentum, making the learning process both faster and more stable. Adam adjusts the learning rate for each parameter individually, based on both the first moment (mean) and the second moment (uncentered variance) of the gradient.

The main steps in Adam include computing these moments, bias-correcting them, and then updating the parameters.

Adam

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

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

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

Adam leverages Momentum to smooth the update direction, making learning faster and more stable. The adaptive learning rate from RMSprop enables Adam to handle noisy, sparse, or non-stationary data efficiently.

LET'S LIGHT ON IT

To create an SGD with Momentum optimizer, we have to specify the momentum attribute over there.

```
def train_with_sgd_momentum(model, X, y, epochs=1000, lr=0.1, momentum=0.9):
    criterion = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=lr, momentum=momentum)
    losses = []
    for epoch in range(epochs):
        outputs = model(X)
        loss = criterion(outputs, y)
        losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return model, losses
```

If it is Nesterov GD, we have set the nesterov parameter to be True.

```
def train_with_nesterov(model, X, y, epochs=1000, lr=0.1, momentum=0.9):
    criterion = nn.BCELoss()
    optimizer = optim.SGD(model.parameters(), lr=lr, momentum=momentum, nesterov=True)
    losses = []
    for epoch in range(epochs):
        outputs = model(X)
        loss = criterion(outputs, y)
        losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return model, losses
```

For AdaGrad,

```
def train_with_adagrad(model, X, y, epochs=1000, lr=0.1):
    criterion = nn.BCELoss()
    optimizer = optim.Adagrad(model.parameters(), lr=lr)
    losses = []
    for epoch in range(epochs):
        outputs = model(X)
        loss = criterion(outputs, y)
        losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return model, losses
```

For RMSProp,

```
def train_with_rmsprop(model, X, y, epochs=1000, lr=0.01):
    criterion = nn.BCELoss()
    optimizer = optim.RMSprop(model.parameters(), lr=lr)
    losses = []
    for epoch in range(epochs):
        outputs = model(X)
        loss = criterion(outputs, y)
        losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return model, losses
```

And this is for Adam...

```python
def train_with_adam(model, X, y, epochs=1000, lr=0.01):
    criterion = nn.BCELoss()
    optimizer = optim.Adam(model.parameters(), lr=lr)
    losses = []
    for epoch in range(epochs):
        outputs = model(X)
        loss = criterion(outputs, y)
        losses.append(loss.item())
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return model, losses
```

MERCI