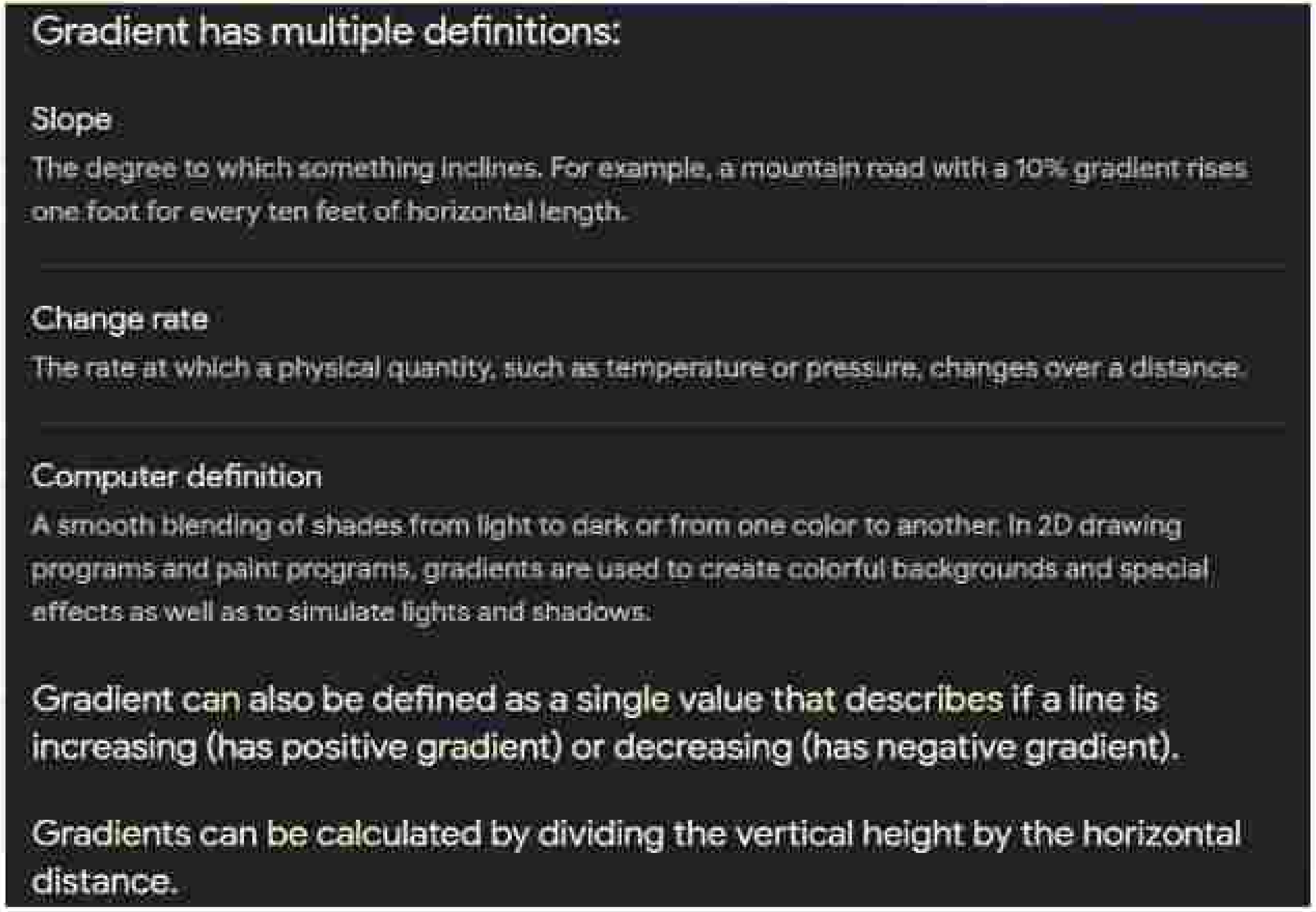
Deep Learning

Backpropagation

- Backpropagation is a supervised learning algorithm used to train artificial neural networks.
- In backpropagation the neural networks adjust weights and biases to minimize the error between predicted and actual outputs.
- Backpropagation aims to minimize the difference between predicted and actual outputs.
- It uses the gradient of the error with respect to weights to iteratively adjust weights for better predictions.
- During the forward pass, input data is fed through the neural network to produce predictions.(the process of passing input data through the neural network to get the prediction)
- The loss function quantifies the error between predicted and actual outputs.
- In backpropagation we try to minimize the loss to improve the model's accuracy.
- Iterate through forward and backward passes until the model converges and Set criteria for stopping iterations.

Deep Learning

Gradient



Gradient has multiple definitions:

Slope

The degree to which something inclines. For example, a mountain road with a 10% gradient rises one foot for every ten feet of horizontal length.

Change rate

The rate at which a physical quantity, such as temperature or pressure, changes over a distance.

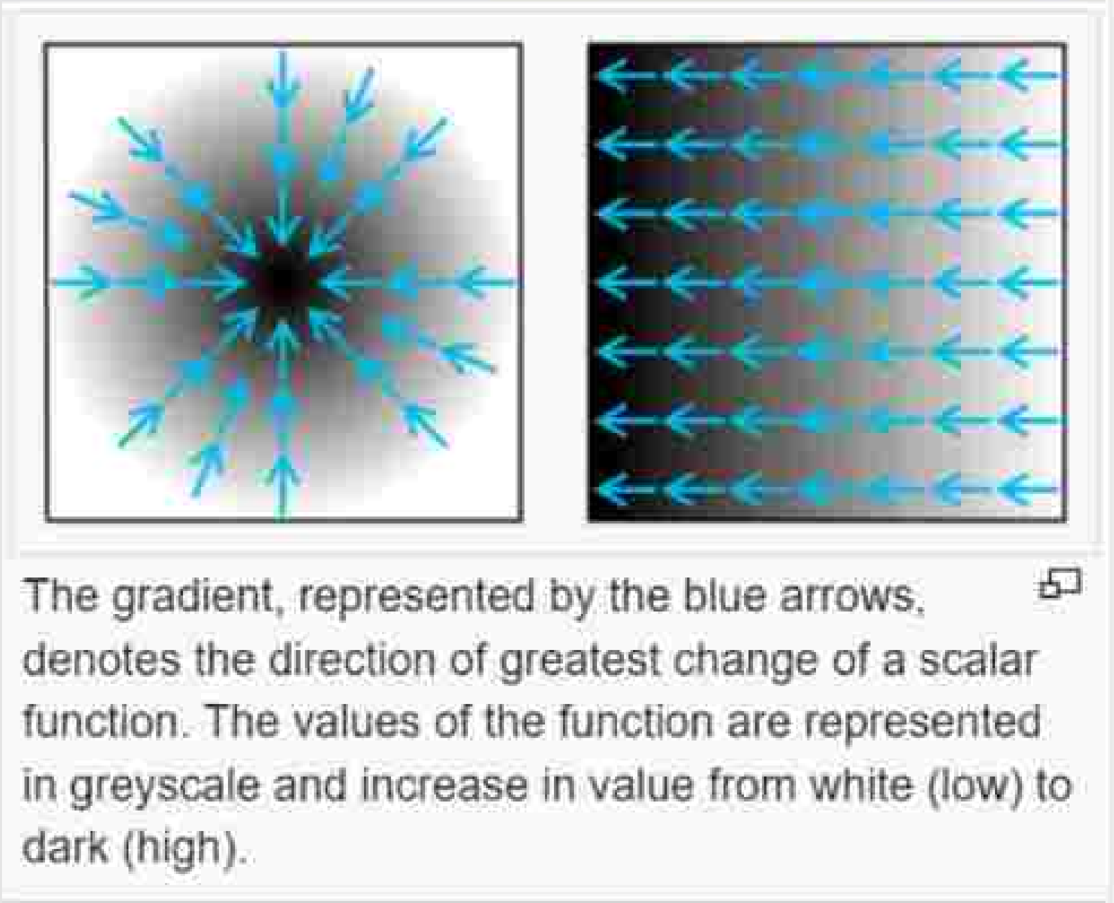
Computer definition

A smooth blending of shades from light to dark or from one color to another. In 2D drawing programs and paint programs, gradients are used to create colorful backgrounds and special effects as well as to simulate lights and shadows.

Gradient can also be defined as a single value that describes if a line is increasing (has positive gradient) or decreasing (has negative gradient).

Gradients can be calculated by dividing the vertical height by the horizontal distance.

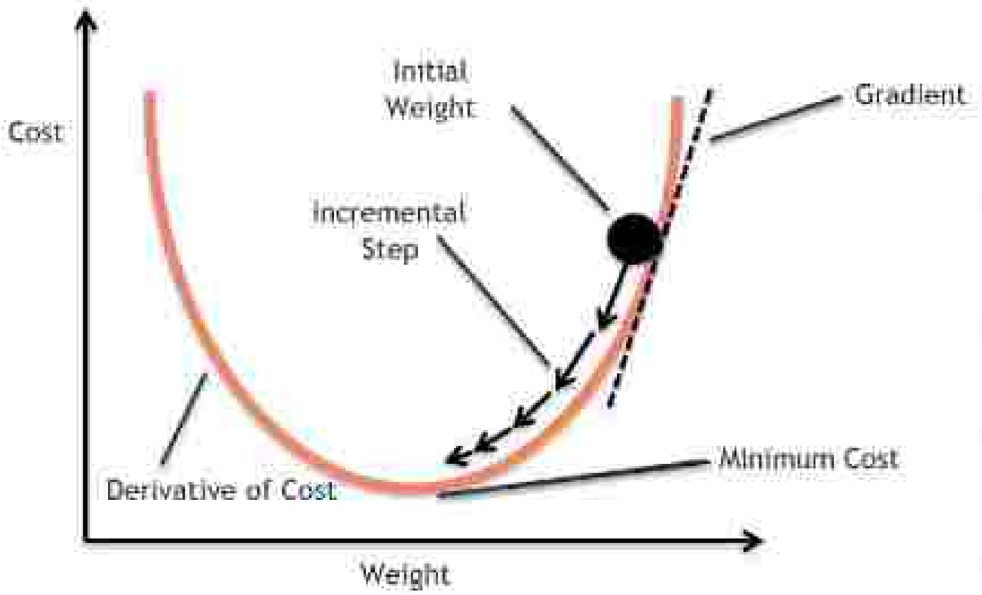
Gradient



The gradient, represented by the blue arrows, denotes the direction of greatest change of a scalar function. The values of the function are represented in greyscale and increase in value from white (low) to dark (high).

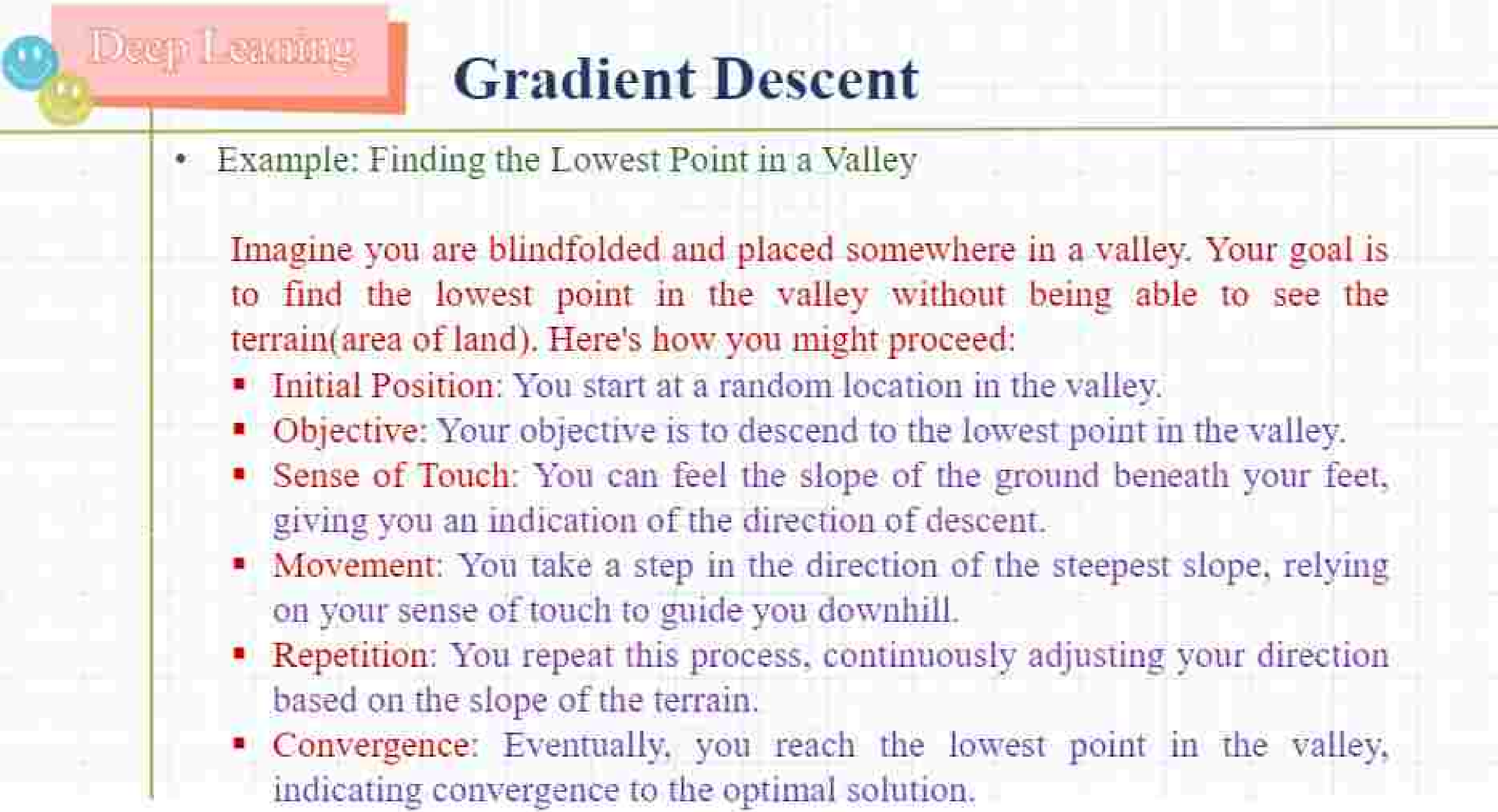
Gradient Descent

- Gradient Descent is an optimization algorithm used in machine learning and deep learning for training models/Neural networks and finding the optimal parameters that minimize a cost function.



Gradient Descent

- Step-by-step explanation of how Gradient Descent works:
  - **Step:1 Initialization:** It begins by initializing the model parameters randomly or with some predetermined values. These parameters could be the weights and biases of a neural network, for example.
  - **Step:2 Compute Gradient:** At each iteration, the algorithm computes the gradient of the cost function with respect to each parameter. The gradient represents the direction of the steepest ascent of the function.
  - **Step:3 Update Parameters:** Once the gradient is computed, the algorithm updates the parameters by moving them in the opposite direction of the gradient. This is done to minimize the cost function. The update rule typically involves multiplying the gradient by a learning rate parameter and subtracting the result from the current parameter values.
  - **Step:4 Iterate:** Steps 2 and 3 are repeated iteratively until a stopping criterion is met. This could be a predefined number of iterations or until the change in the cost function falls below a certain threshold.



Gradient Descent

- Example: Finding the Lowest Point in a Valley

Imagine you are blindfolded and placed somewhere in a valley. Your goal is to find the lowest point in the valley without being able to see the terrain(area of land). Here's how you might proceed:

- Initial Position: You start at a random location in the valley.
- Objective: Your objective is to descend to the lowest point in the valley.
- Sense of Touch: You can feel the slope of the ground beneath your feet, giving you an indication of the direction of descent.
- Movement: You take a step in the direction of the steepest slope, relying on your sense of touch to guide you downhill.
- Repetition: You repeat this process, continuously adjusting your direction based on the slope of the terrain.
- Convergence: Eventually, you reach the lowest point in the valley, indicating convergence to the optimal solution.

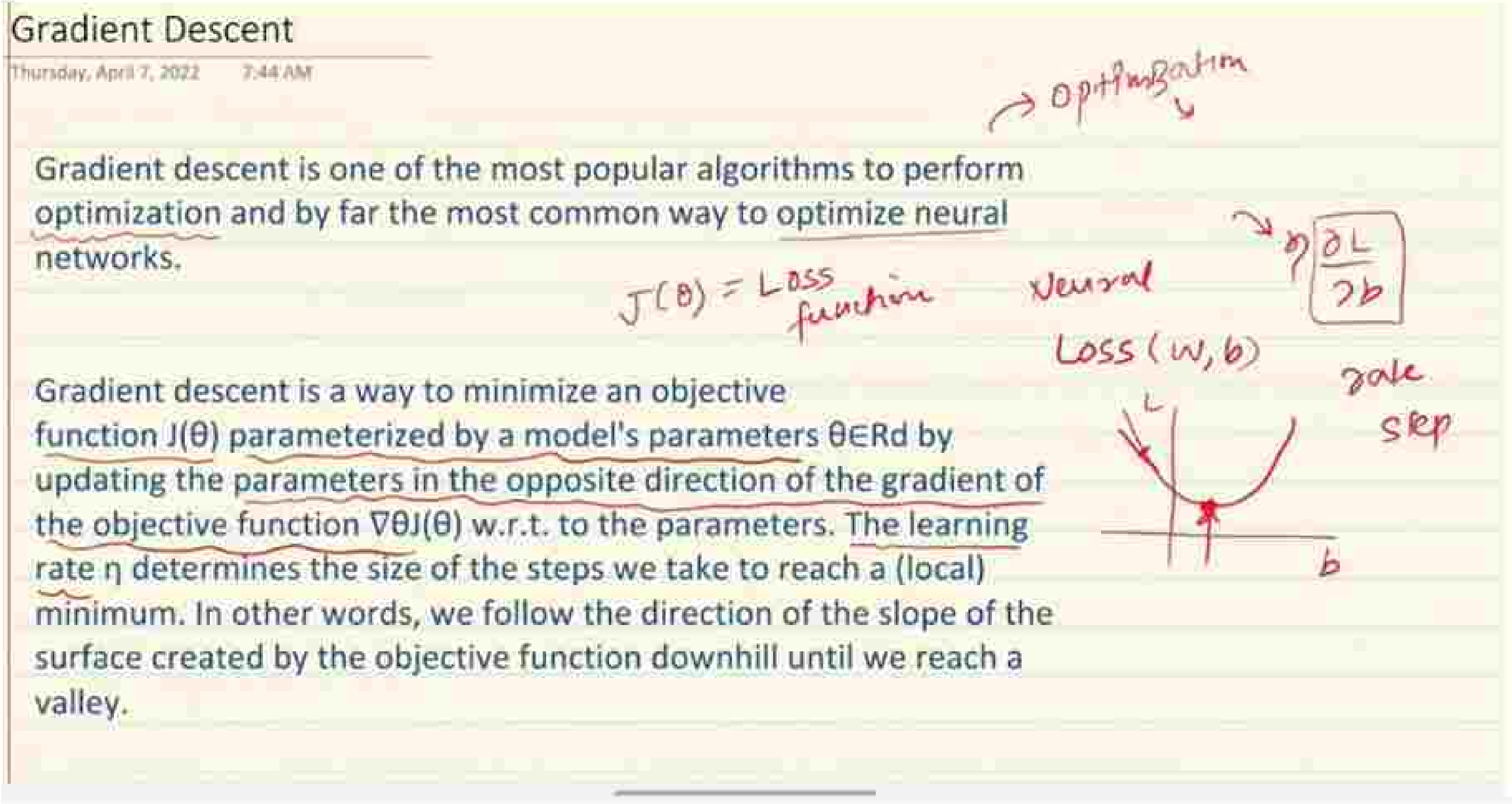
Gradient Descent

- In this analogy:
  - The blindfolded person represents the optimization algorithm.
  - The sense of touch represents the gradient, providing information about the direction of descent.
  - Moving downhill corresponds to updating the parameters in the direction that minimizes the objective function.
  - This example illustrates how Gradient Descent works by iteratively adjusting parameters to minimize a cost function, much like finding the lowest point in a valley by descending along the steepest slope.

Gradient Descent

There are three different variants of Gradient Descent

1. Batch Gradient Descent
2. Stochastic Gradient Descent
3. Mini-batch Gradient Descent



Gradient Descent

Thursday, April 7, 2022 7:44 AM

→ optimization

Gradient descent is one of the most popular algorithms to perform optimization and by far the most common way to optimize neural networks.

$J(\theta) = $ Loss function

Neural

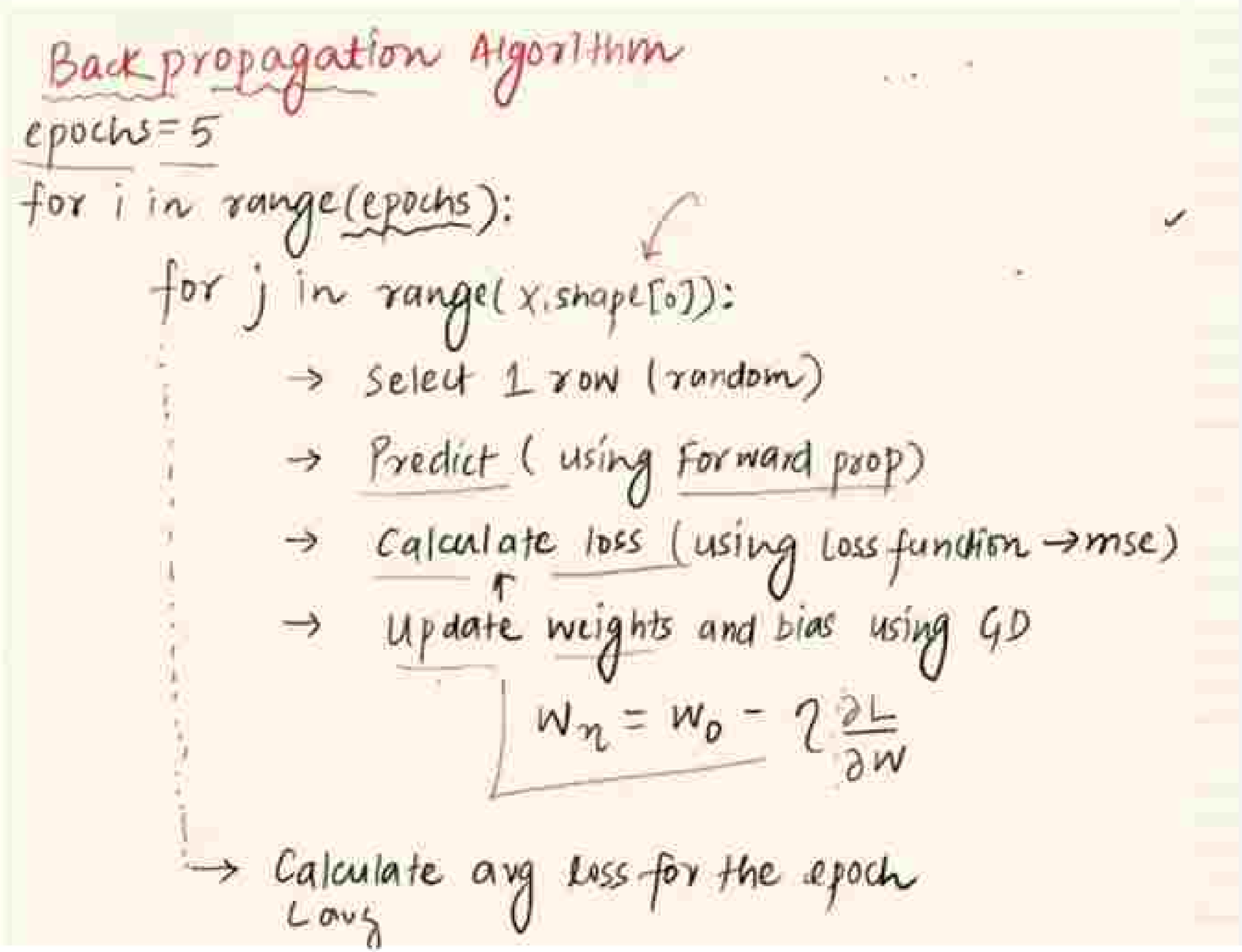
Loss(w, b)

→ $\eta \boxed{\frac{\partial L}{\partial b}}$

rate

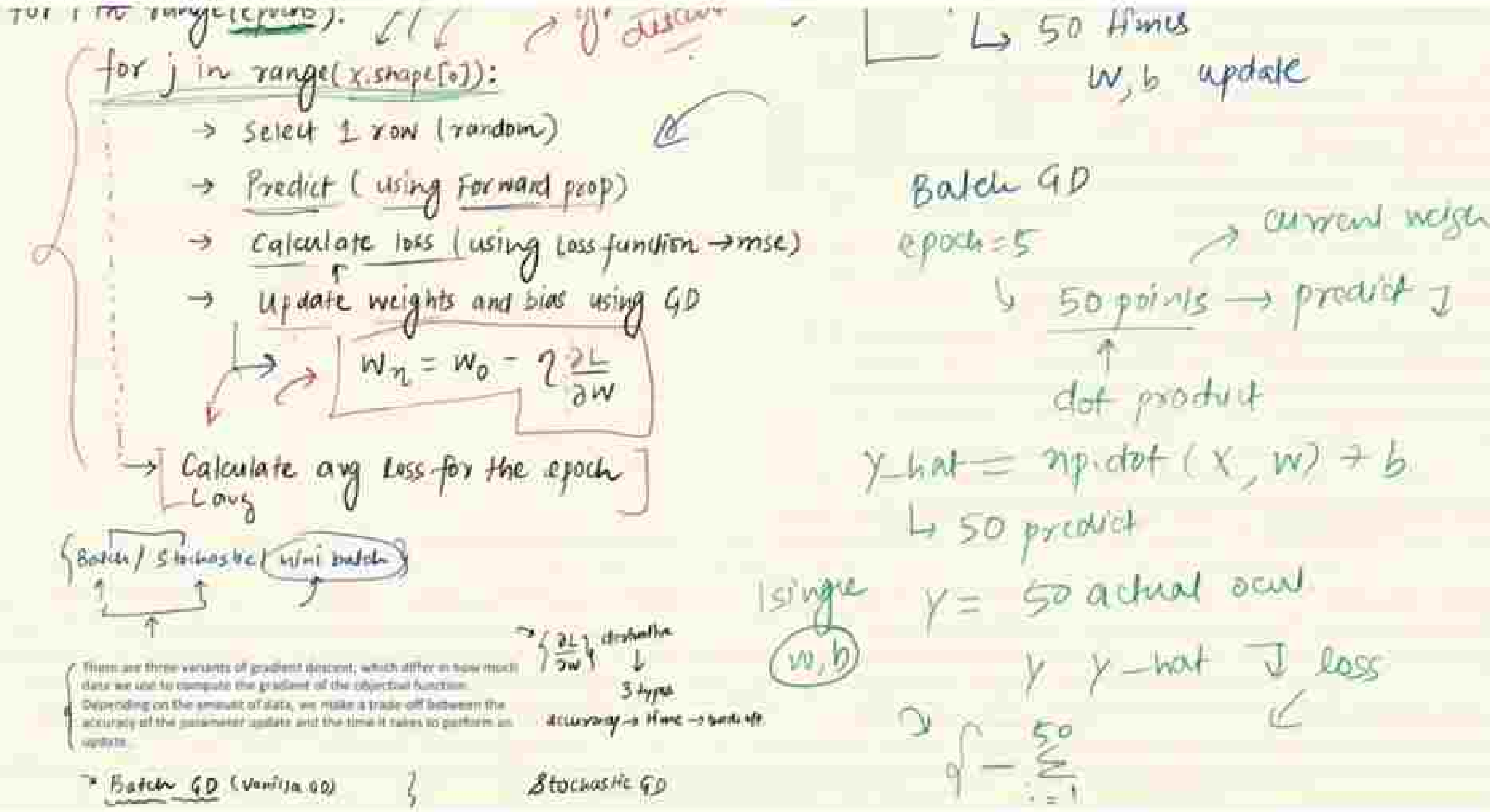
step

Gradient descent is a way to minimize an objective function J(θ) parameterized by a model's parameters θ∈Rd by updating the parameters in the opposite direction of the gradient of the objective function ∇θJ(θ) w.r.t. to the parameters. The learning rate η determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.



Back propagation Algorithm

```
epochs=5
for i in range(epochs):
    for j in range(x.shape[0]):
        -> Select 1 row (random)
        -> Predict ( using Forward prop)
        -> Calculate loss (using Loss function -> mse)
        -> Update weights and bias using GD
               $W_n = W_o - \eta \frac{\partial L}{\partial W}$
    -> Calculate avg loss for the epoch
       $L_{avg}$
```



```
for i in range(epochs):
    for j in range(X.shape[0]):
        → Select 1 row (random)
        → Predict (using Forward prop)
        → Calculate loss (using Loss function → mse)
        → Update weights and bias using GD
            → $w_n = w_0 - \eta \frac{\partial L}{\partial w}$
    → Calculate avg Loss for the epoch
      L avg
```

{Batch / Stochastic / Mini batch}

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 parameter update and the time it takes to perform an update.

→ {$\frac{\partial L}{\partial w}$} derivative
↓
3 types

accuracy → time → tradeoff

* Batch GD (vanilla GD)

Stochastic GD

↳ 50 times
W, b update

Batch GD

epoch = 5

↳ 50 points → predict J

→ current weigh

↑
dot product

y_hat = np.dot(X, W) + b

↳ 50 predict

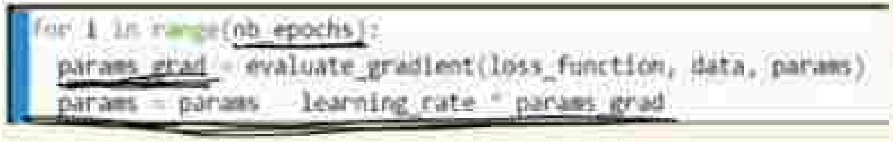
1 single (w, b)

y = 50 actual ocw

y y_hat J loss

$J = \sum_{i=1}^{50}$

↘ Batch GD (Vanilla GD)

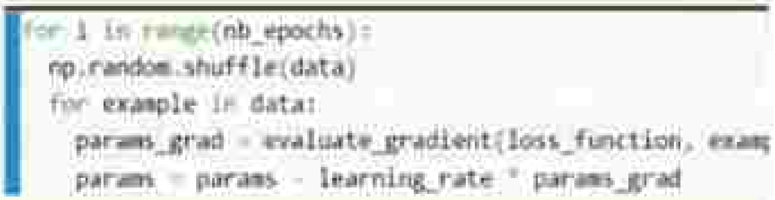


```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

entire dataset ↓ update

epochs = # of updates

Stochastic GD



```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for example in data:
    params_grad = evaluate_gradient(loss_function, examp
    params = params - learning_rate * params_grad
```

Batch GD

```
epoch = $\boxed{10}$
for in range(10):                         → total → 10 times
                                                    w,b update
        y_hat = np.dot(x, w) + b
     ↙
   50 values

        y = 50 values

          y_hat ,y → loss
         ↳ w,b update    $w_n = w_0 - \eta \frac{\partial L}{\partial w}$

    → loss
```

Batch Gradient Descent

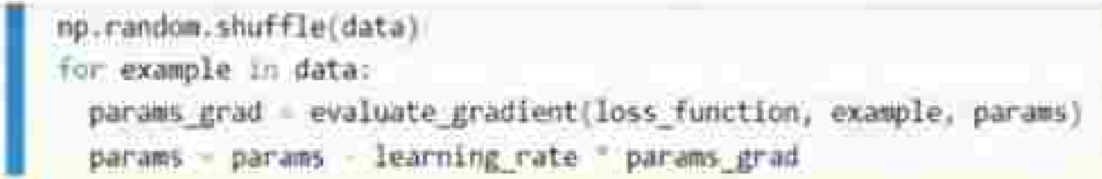
- Batch Gradient Descent (BGD) is a variant of the Gradient Descent optimization algorithm used to minimize a cost function in machine learning and deep learning models.
- It's called "batch" because it processes the entire training dataset in each iteration to update the model parameters.

  - Optimization Algorithm in Machine/Deep Learning
  - Variant of Gradient Descent
  - Minimize a cost function in ML/DL models
  - Processes entire training dataset in each iteration

Batch Gradient Descent

- **Working Principle**
  - Initialization: Start with initial guess for model parameters
  - Compute Gradient: Calculate gradient of cost function with respect to parameters using entire training dataset
  - Update Parameters: Adjust parameters using gradients and learning rate
  - Repeat until convergence criteria met

- **Key Aspects**
  - Uses entire dataset in each iteration so it is computationally expensive for large datasets
  - Accurate estimate of gradient
  - Often converges to global minimum in most of the problems



```
np.random.shuffle(data)
for example in data:
  params_grad = evaluate_gradient(loss_function, example, params)
  params = params - learning_rate * params_grad
```

$50 \times 10 = \underline{500}$

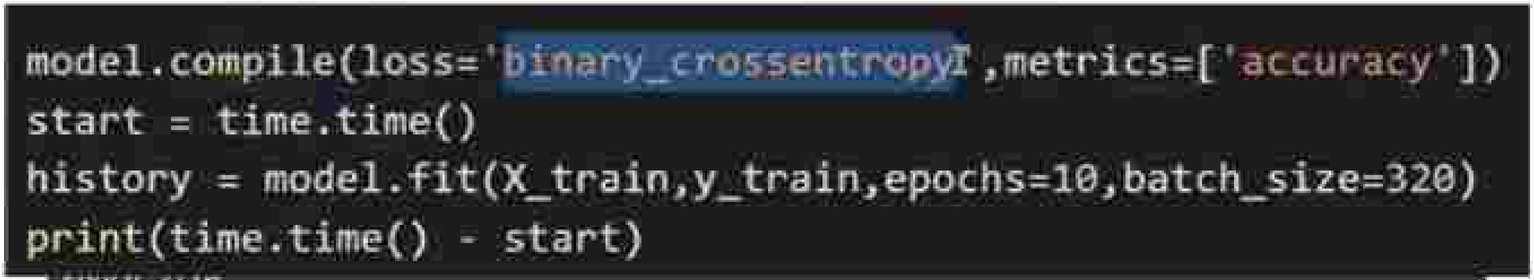
5

epoch → $\underline{10}$ (50 rows)

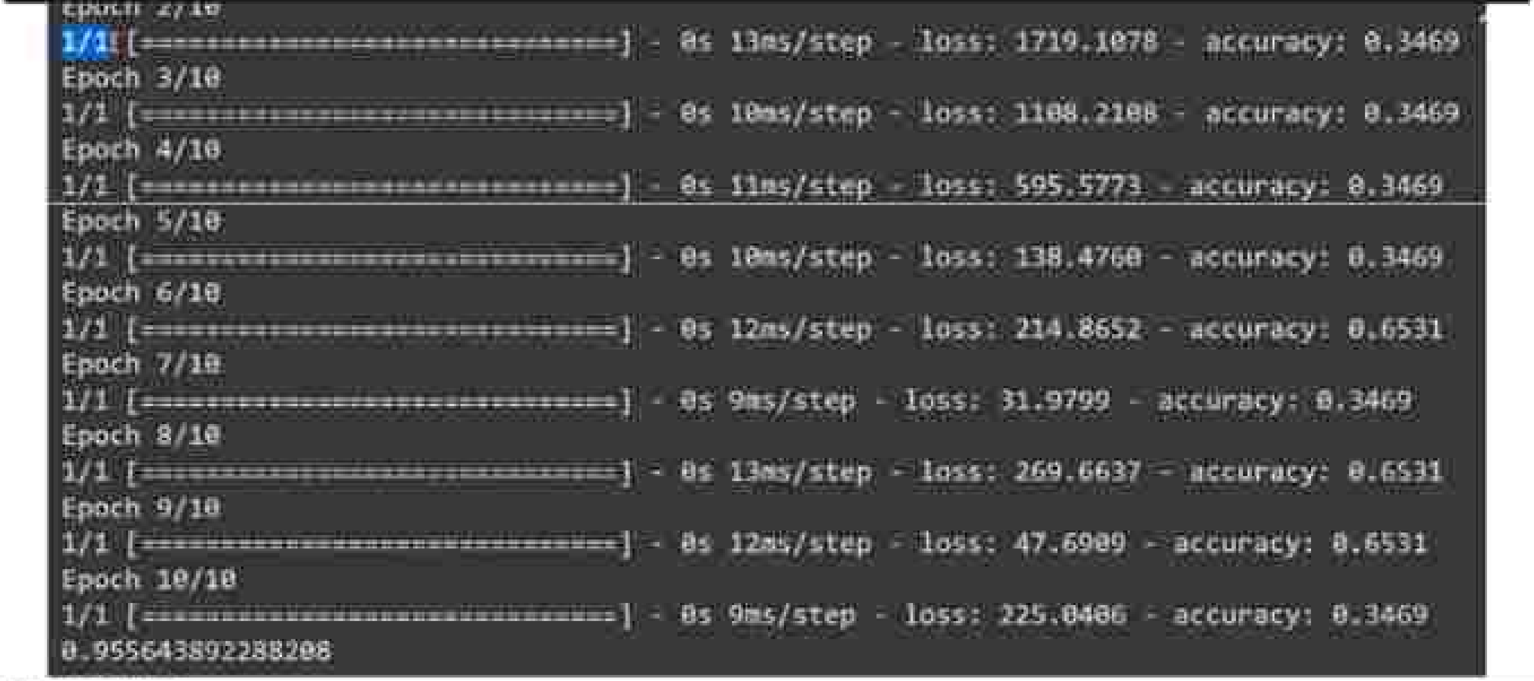
50 rows

```
for i in range(10):
    → shuffle
    for i in range( X.shape[0])
        ↳ 1 random point
        ↳ y_hat → forward
        ↳ loss
        ↳ w,b update → $w_n = w_o - \eta \frac{\partial L}{\partial w}$
avg loss print → for the epoch
```

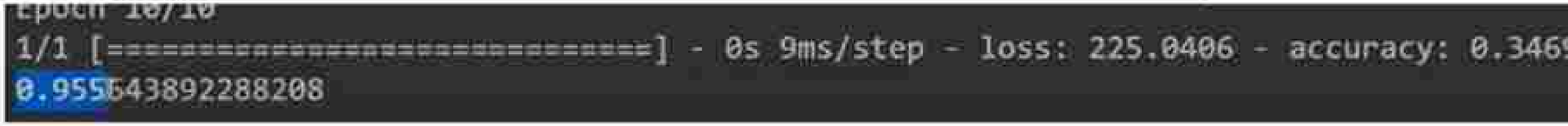
$10 \times 50 = 500$ update



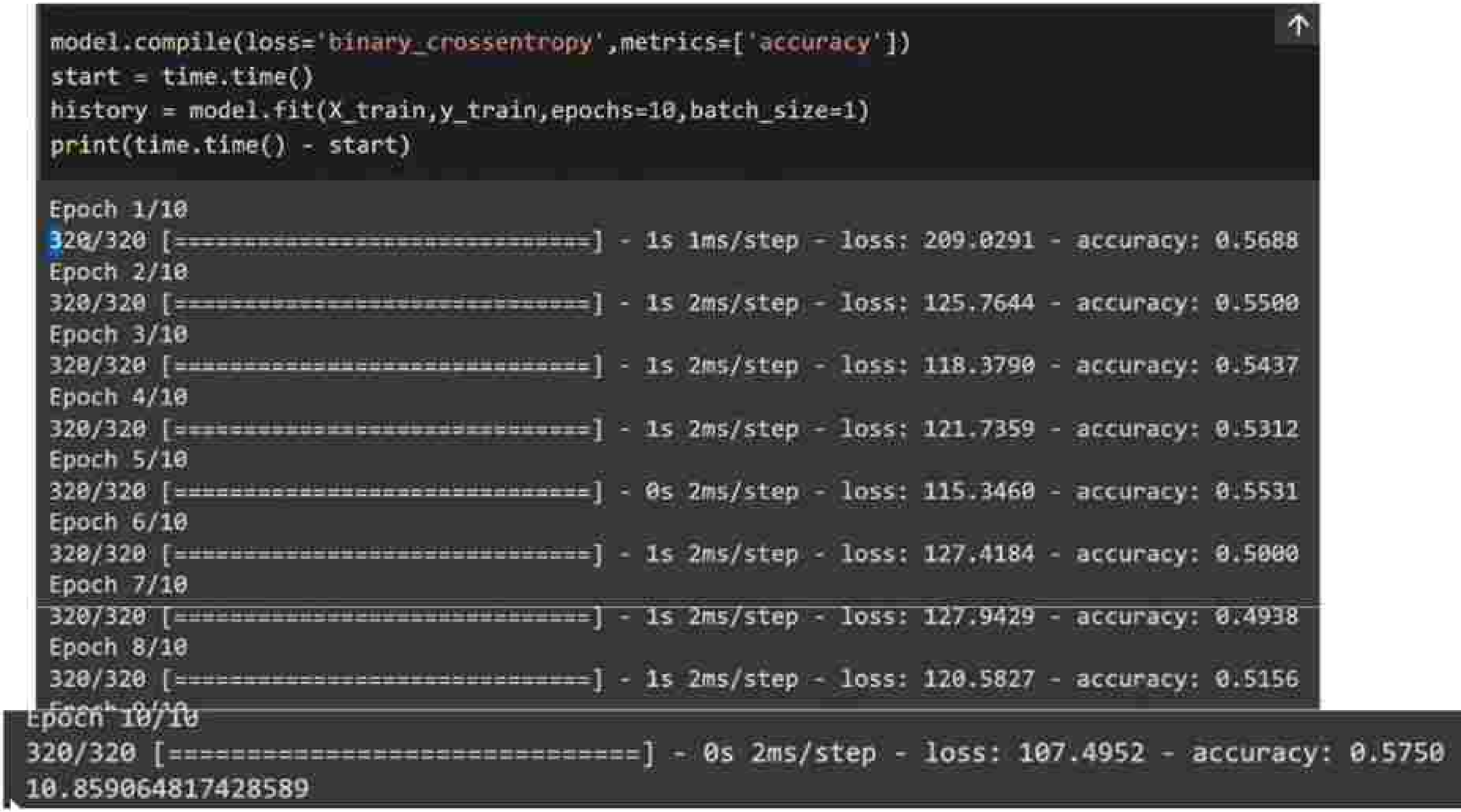
```
model.compile(loss='binary_crossentropy',metrics=['accuracy'])
start = time.time()
history = model.fit(X_train,y_train,epochs=10,batch_size=320)
print(time.time() - start)
```



```
Epoch 2/10
1/1 [==============================] - 0s 13ms/step - loss: 1719.1078 - accuracy: 0.3469
Epoch 3/10
1/1 [==============================] - 0s 10ms/step - loss: 1108.2108 - accuracy: 0.3469
Epoch 4/10
1/1 [==============================] - 0s 11ms/step - loss: 595.5773 - accuracy: 0.3469
Epoch 5/10
1/1 [==============================] - 0s 10ms/step - loss: 138.4760 - accuracy: 0.3469
Epoch 6/10
1/1 [==============================] - 0s 12ms/step - loss: 214.8652 - accuracy: 0.6531
Epoch 7/10
1/1 [==============================] - 0s 9ms/step - loss: 31.9799 - accuracy: 0.3469
Epoch 8/10
1/1 [==============================] - 0s 13ms/step - loss: 269.6637 - accuracy: 0.6531
Epoch 9/10
1/1 [==============================] - 0s 12ms/step - loss: 47.6909 - accuracy: 0.6531
Epoch 10/10
1/1 [==============================] - 0s 9ms/step - loss: 225.0406 - accuracy: 0.3469
0.955643892288208
```

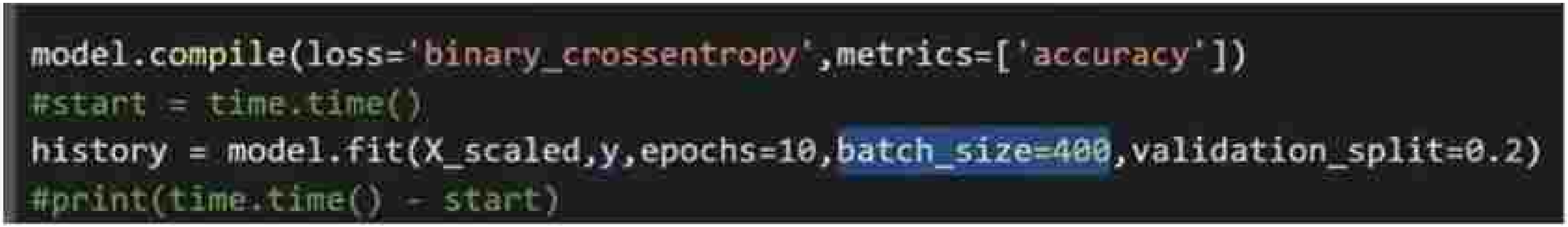


```
Epoch 10/10
1/1 [==============================] - 0s 9ms/step - loss: 225.0406 - accuracy: 0.346
0.955643892288208
```

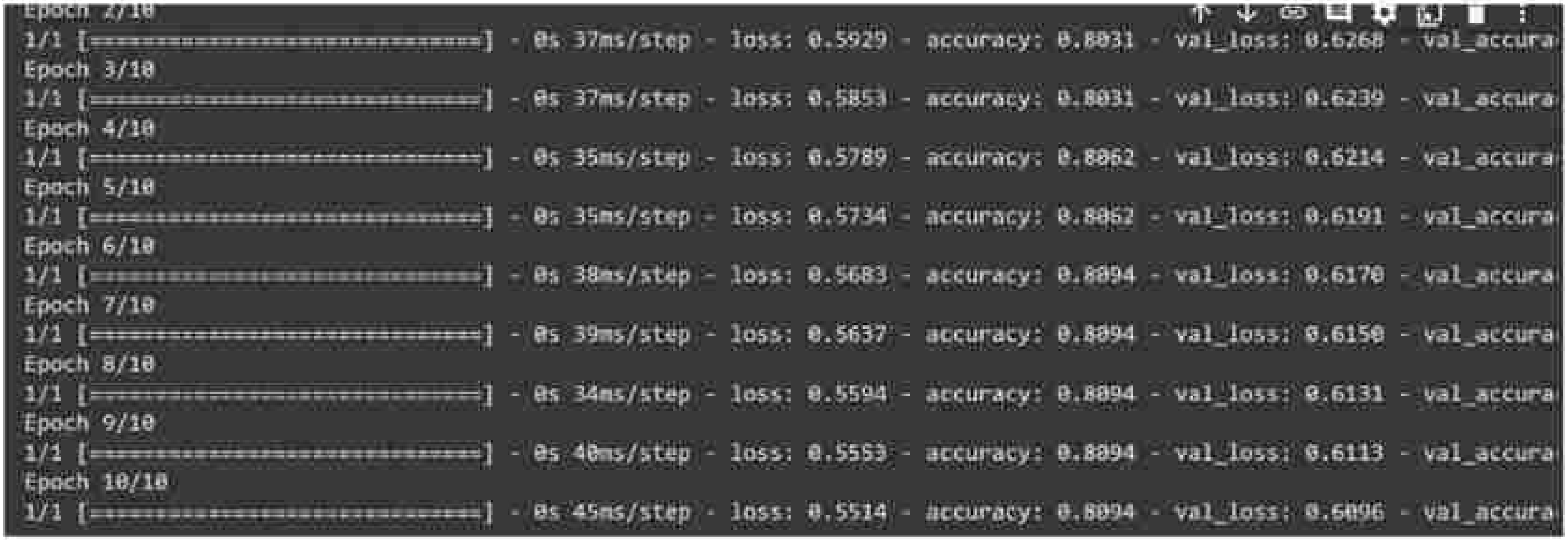


```
model.compile(loss='binary_crossentropy',metrics=['accuracy'])
start = time.time()
history = model.fit(X_train,y_train,epochs=10,batch_size=1)
print(time.time() - start)
```

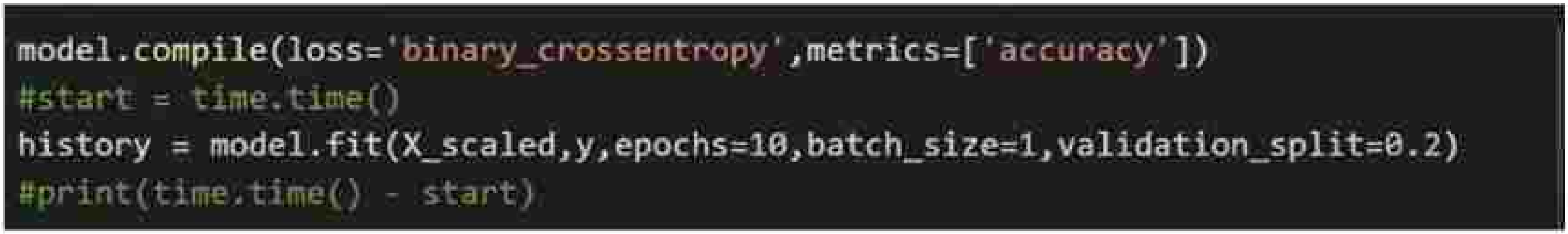
```
Epoch 1/10
320/320 [==============================] - 1s 1ms/step - loss: 209.0291 - accuracy: 0.5688
Epoch 2/10
320/320 [==============================] - 1s 2ms/step - loss: 125.7644 - accuracy: 0.5500
Epoch 3/10
320/320 [==============================] - 1s 2ms/step - loss: 118.3790 - accuracy: 0.5437
Epoch 4/10
320/320 [==============================] - 1s 2ms/step - loss: 121.7359 - accuracy: 0.5312
Epoch 5/10
320/320 [==============================] - 0s 2ms/step - loss: 115.3460 - accuracy: 0.5531
Epoch 6/10
320/320 [==============================] - 1s 2ms/step - loss: 127.4184 - accuracy: 0.5000
Epoch 7/10
320/320 [==============================] - 1s 2ms/step - loss: 127.9429 - accuracy: 0.4938
Epoch 8/10
320/320 [==============================] - 1s 2ms/step - loss: 120.5827 - accuracy: 0.5156
Epoch 9/10
Epoch 10/10
320/320 [==============================] - 0s 2ms/step - loss: 107.4952 - accuracy: 0.5750
10.859064817428589
```



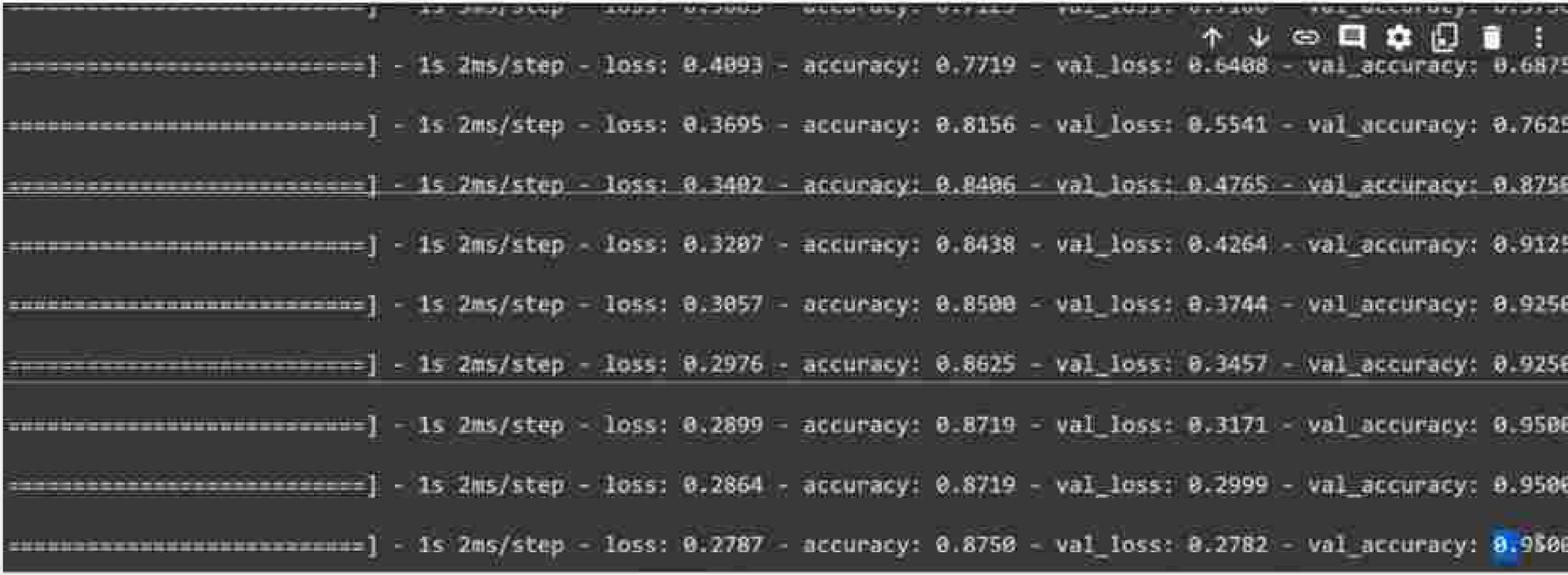
```
model.compile(loss='binary_crossentropy',metrics=['accuracy'])
#start = time.time()
history = model.fit(X_scaled,y,epochs=10,batch_size=400,validation_split=0.2)
#print(time.time() - start)
```



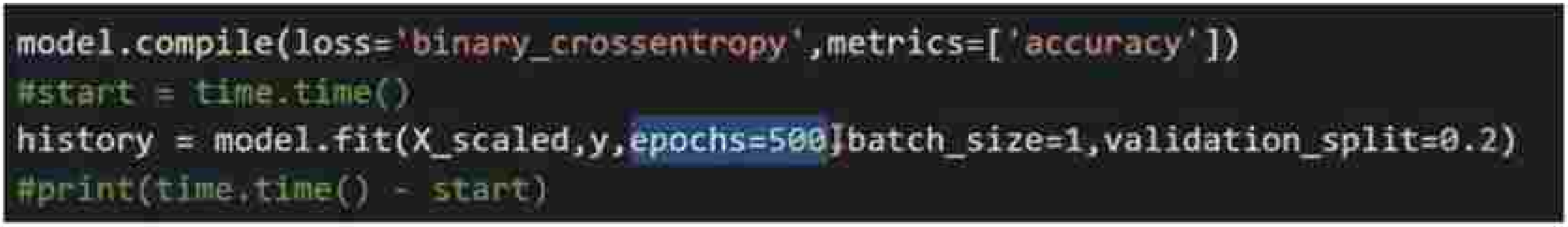
```
Epoch 2/10
1/1 [==============================] - 0s 37ms/step - loss: 0.5929 - accuracy: 0.8031 - val_loss: 0.6268 - val_accura
Epoch 3/10
1/1 [==============================] - 0s 37ms/step - loss: 0.5853 - accuracy: 0.8031 - val_loss: 0.6239 - val_accura
Epoch 4/10
1/1 [==============================] - 0s 35ms/step - loss: 0.5789 - accuracy: 0.8062 - val_loss: 0.6214 - val_accura
Epoch 5/10
1/1 [==============================] - 0s 35ms/step - loss: 0.5734 - accuracy: 0.8062 - val_loss: 0.6191 - val_accura
Epoch 6/10
1/1 [==============================] - 0s 38ms/step - loss: 0.5683 - accuracy: 0.8094 - val_loss: 0.6170 - val_accura
Epoch 7/10
1/1 [==============================] - 0s 39ms/step - loss: 0.5637 - accuracy: 0.8094 - val_loss: 0.6150 - val_accura
Epoch 8/10
1/1 [==============================] - 0s 34ms/step - loss: 0.5594 - accuracy: 0.8094 - val_loss: 0.6131 - val_accura
Epoch 9/10
1/1 [==============================] - 0s 40ms/step - loss: 0.5553 - accuracy: 0.8094 - val_loss: 0.6113 - val_accura
Epoch 10/10
1/1 [==============================] - 0s 45ms/step - loss: 0.5514 - accuracy: 0.8094 - val_loss: 0.6096 - val_accura
```



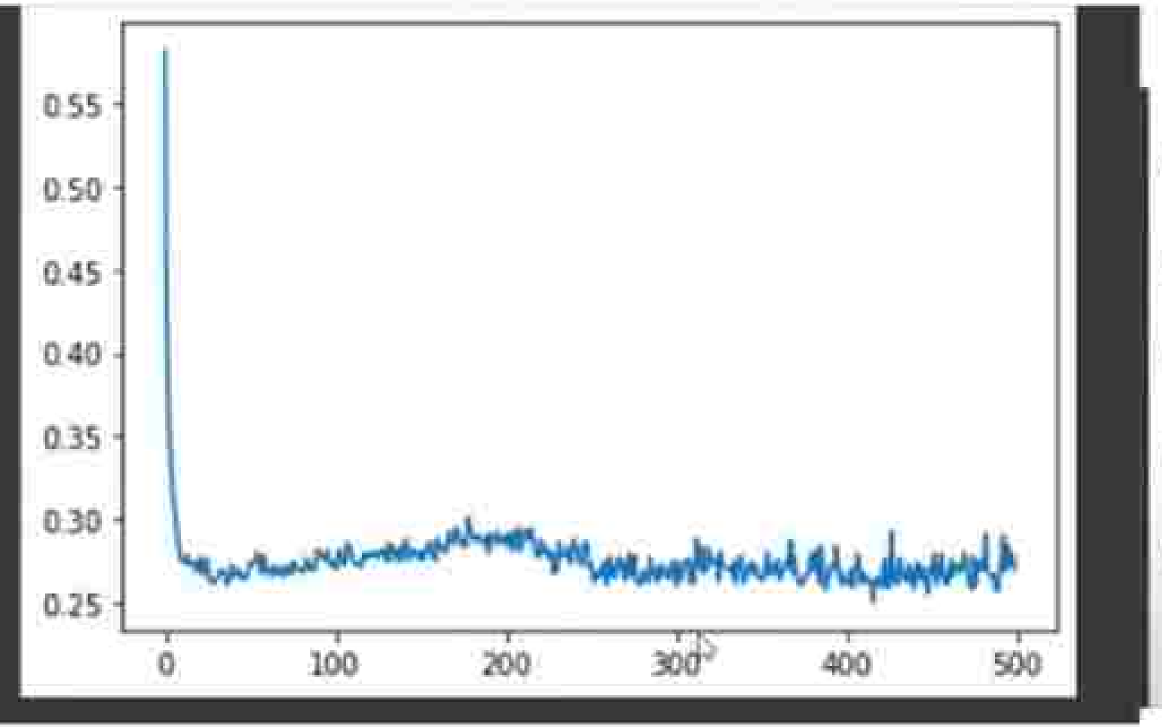
```
model.compile(loss='binary_crossentropy',metrics=['accuracy'])
#start = time.time()
history = model.fit(X_scaled,y,epochs=10,batch_size=1,validation_split=0.2)
#print(time.time() - start)
```

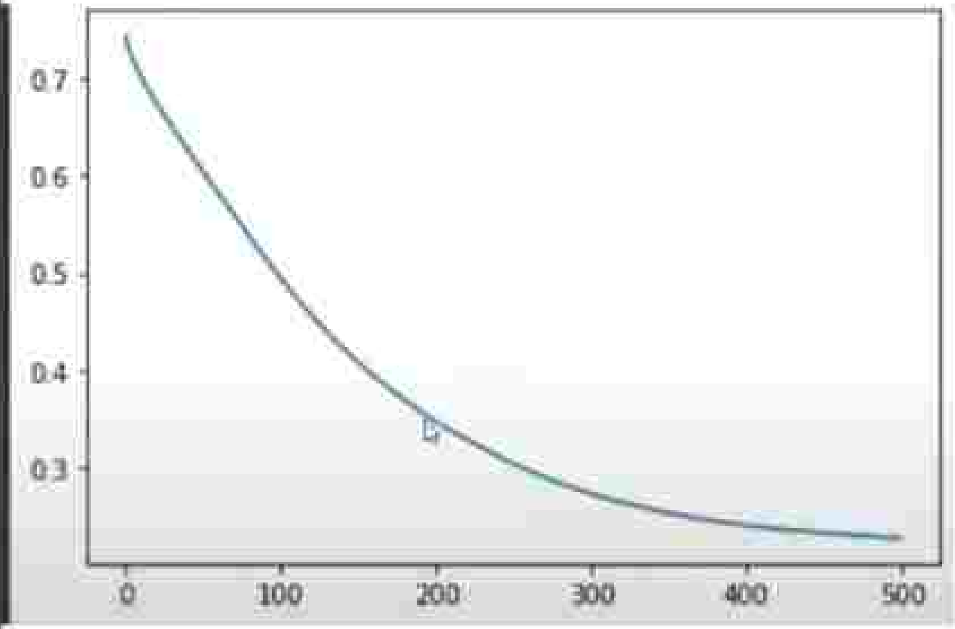


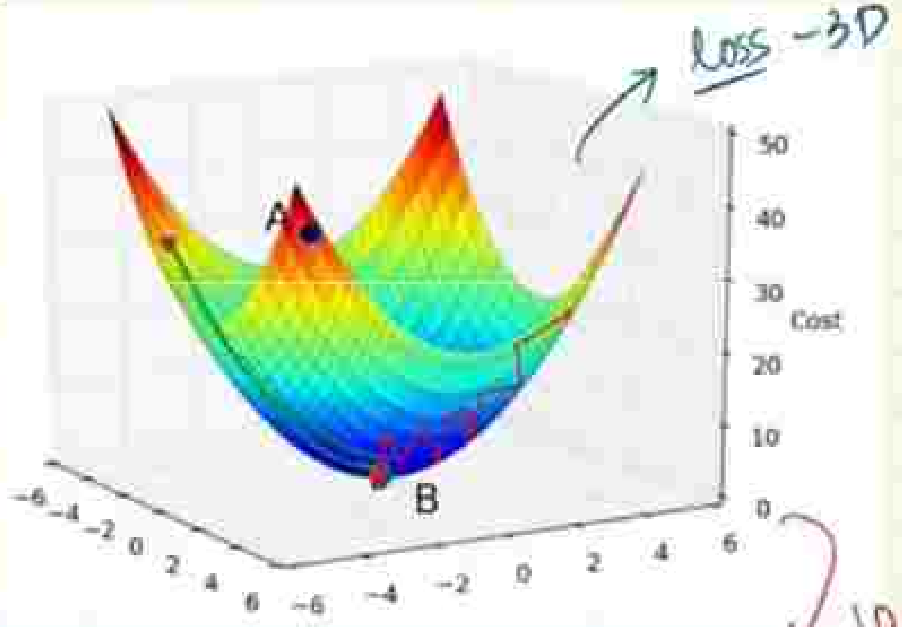
```
==============================] - 1s 2ms/step - loss: 0.4093 - accuracy: 0.7719 - val_loss: 0.6408 - val_accuracy: 0.6875
==============================] - 1s 2ms/step - loss: 0.3695 - accuracy: 0.8156 - val_loss: 0.5541 - val_accuracy: 0.7625
==============================] - 1s 2ms/step - loss: 0.3402 - accuracy: 0.8406 - val_loss: 0.4765 - val_accuracy: 0.8750
==============================] - 1s 2ms/step - loss: 0.3207 - accuracy: 0.8438 - val_loss: 0.4264 - val_accuracy: 0.9125
==============================] - 1s 2ms/step - loss: 0.3057 - accuracy: 0.8500 - val_loss: 0.3744 - val_accuracy: 0.9250
==============================] - 1s 2ms/step - loss: 0.2976 - accuracy: 0.8625 - val_loss: 0.3457 - val_accuracy: 0.9250
==============================] - 1s 2ms/step - loss: 0.2899 - accuracy: 0.8719 - val_loss: 0.3171 - val_accuracy: 0.9500
==============================] - 1s 2ms/step - loss: 0.2864 - accuracy: 0.8719 - val_loss: 0.2999 - val_accuracy: 0.9500
==============================] - 1s 2ms/step - loss: 0.2787 - accuracy: 0.8750 - val_loss: 0.2782 - val_accuracy: 0.9500
```

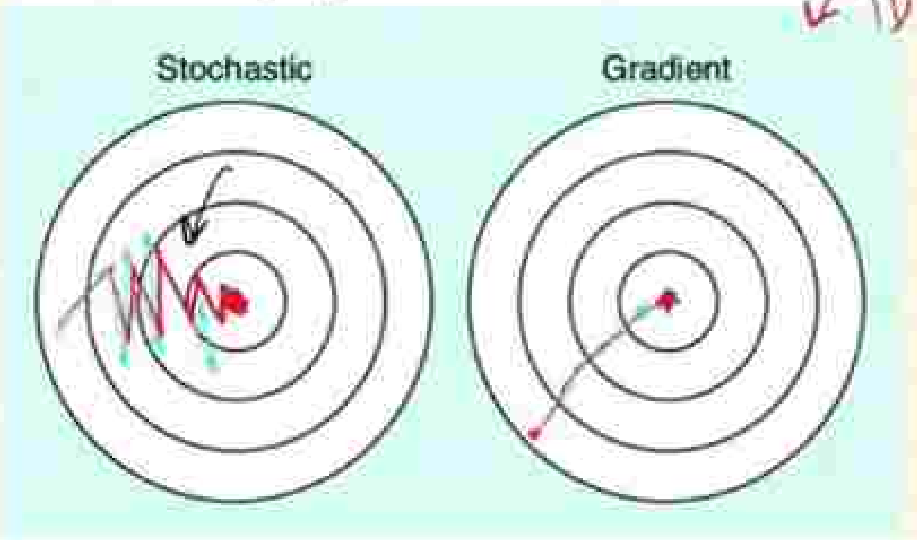


```
model.compile(loss='binary_crossentropy',metrics=['accuracy'])
#start = time.time()
history = model.fit(X_scaled,y,epochs=500,batch_size=1,validation_split=0.2)
#print(time.time() - start)
```









Stochastic

↳ random → 1

→ point → updates

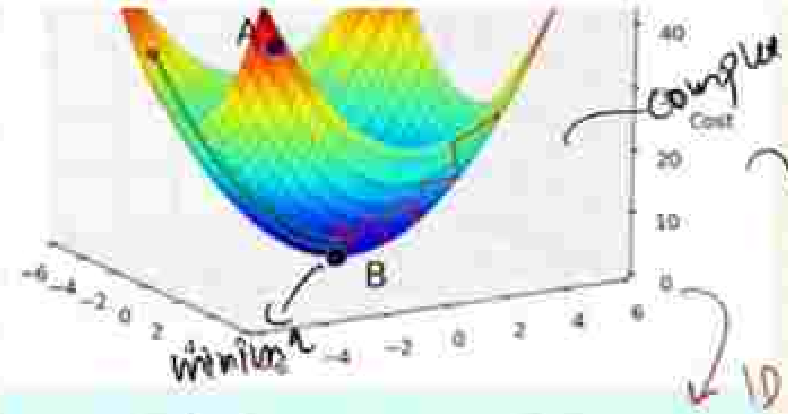
Batch → all point → 1 update

→ dataset

Spikey SGD useful

Yes

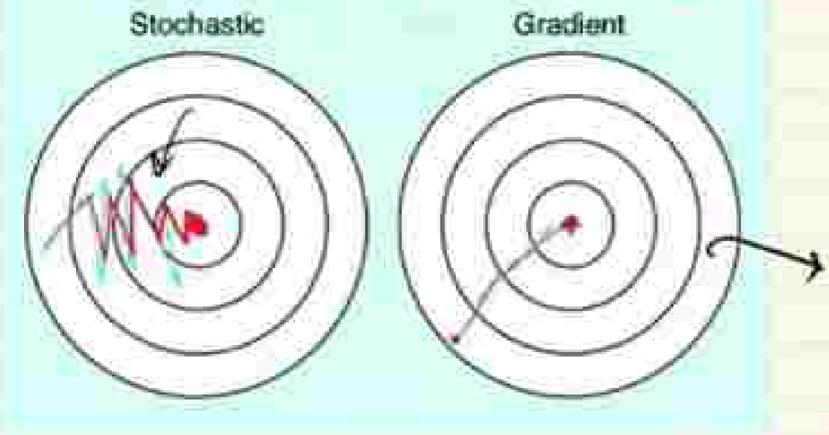
No

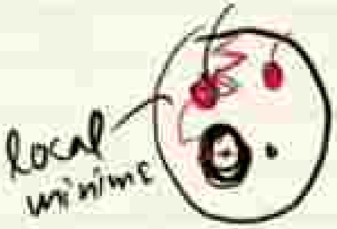


Batch → dataset
↓
all point
↓
1 update

Spikey SGD useful

Yes	No
	SGD BGD





SGD
↓
help the algo to move out of local minima

avg loss print → for the epoch

frequency of weight u

* Batch GD → 1 loop

rows

epoch = 10

→ epoch

for in range(10):

loop

→ total → 10 times

w,b update

y_hat = np.dot(x, w) + b

50 values

dot → smart replacement → loops

↳ vectorization ~ faster loop

Optimized !

y = 50 values

y_hat , y → loss

↳ w,b update $w_n = w_0 - \eta \frac{\partial L}{\partial w}$

→ loss

Vectorization → faster × loops

Vectorization → big dataset

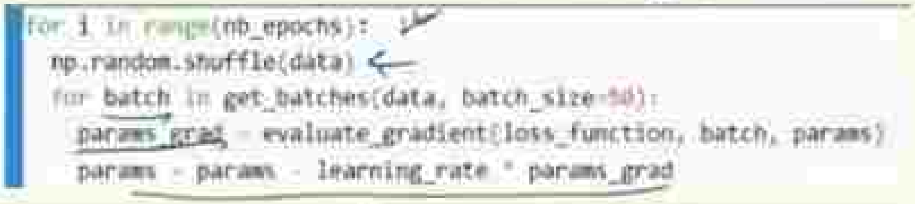
np.dot (x , w) + b

RAM

x → dataset of 10 crore

Mini Batch Gradient Descent

SGD → BGD



```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

$\frac{n}{x}$ = # of batches

↓

updates/epoch

Keras batch-size = x

bgd > mbgd > sgd

BGD ↔ SGD Best of both worlds

320 rows → batches ↓ 32

In every epoch

10 batches

10 update

```
for i in epochs →
    for j in num of batch
        1 batch
          ↳ y-pred ( vector
          ↳ loss
          ↳ update
```

np.dot

→ Why batch_size is provided in multiple of(2)?

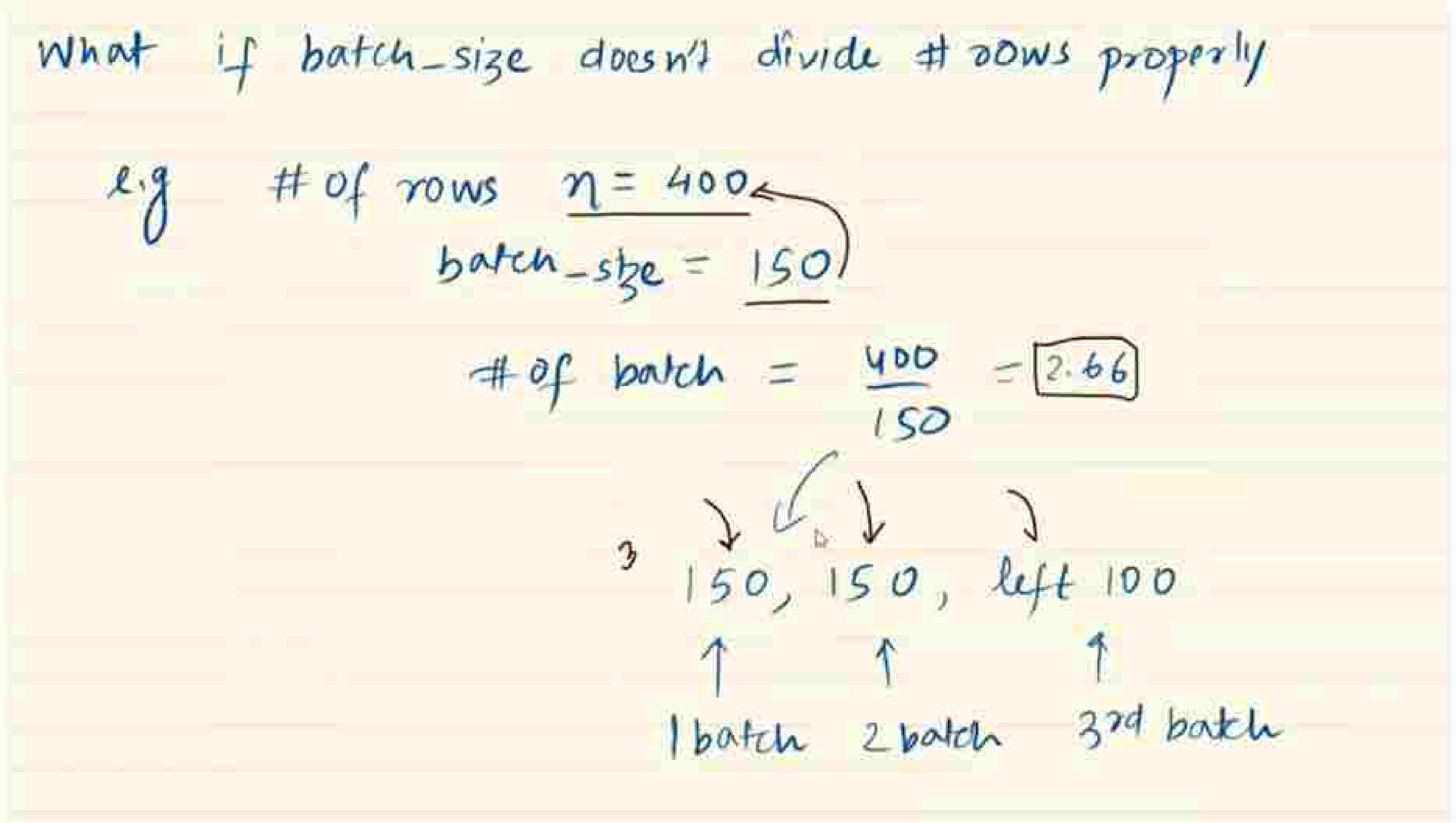
↑

2, 4, 8, 32, 64, 128, 256

RAM effective → binary

cores

RAM → optimization



What if batch_size doesn't divide # rows properly

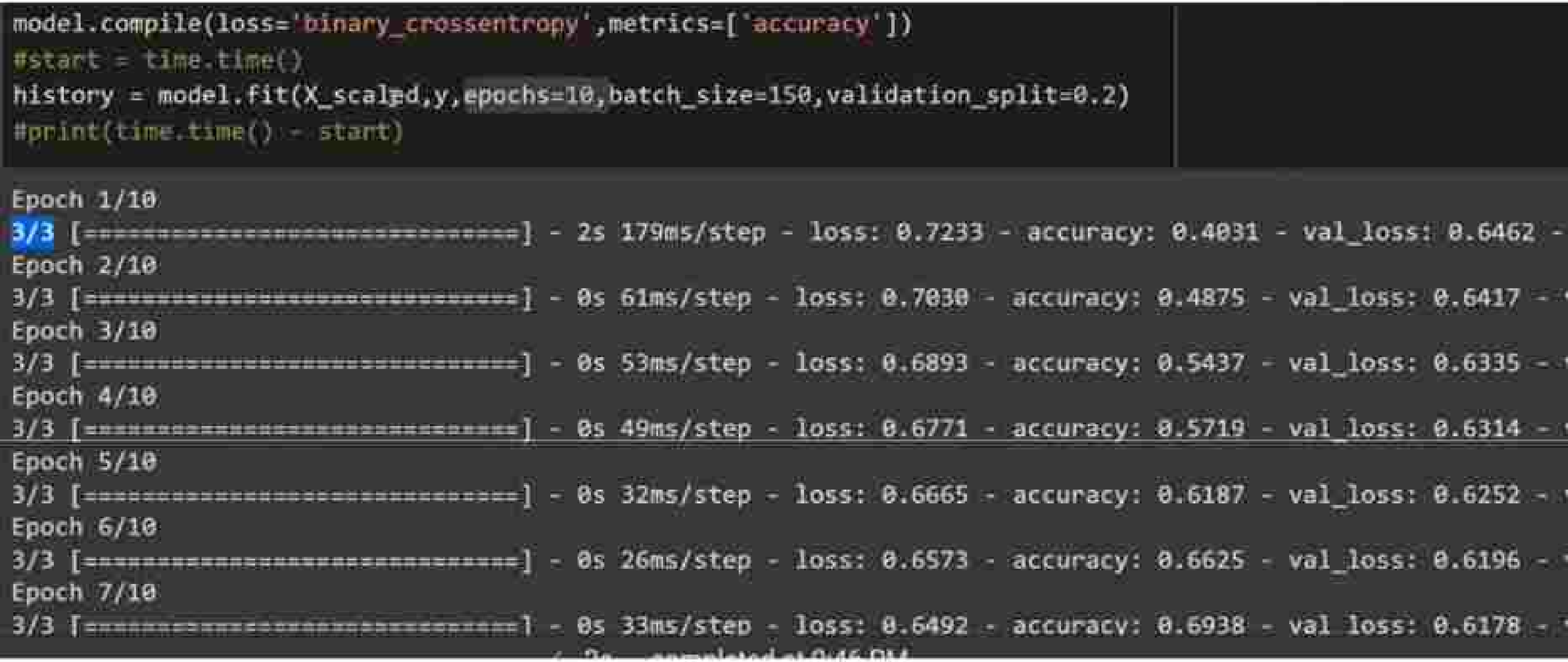
e.g # of rows $n = 400$

batch_size = 150

# of batch = $\frac{400}{150} = \boxed{2.66}$

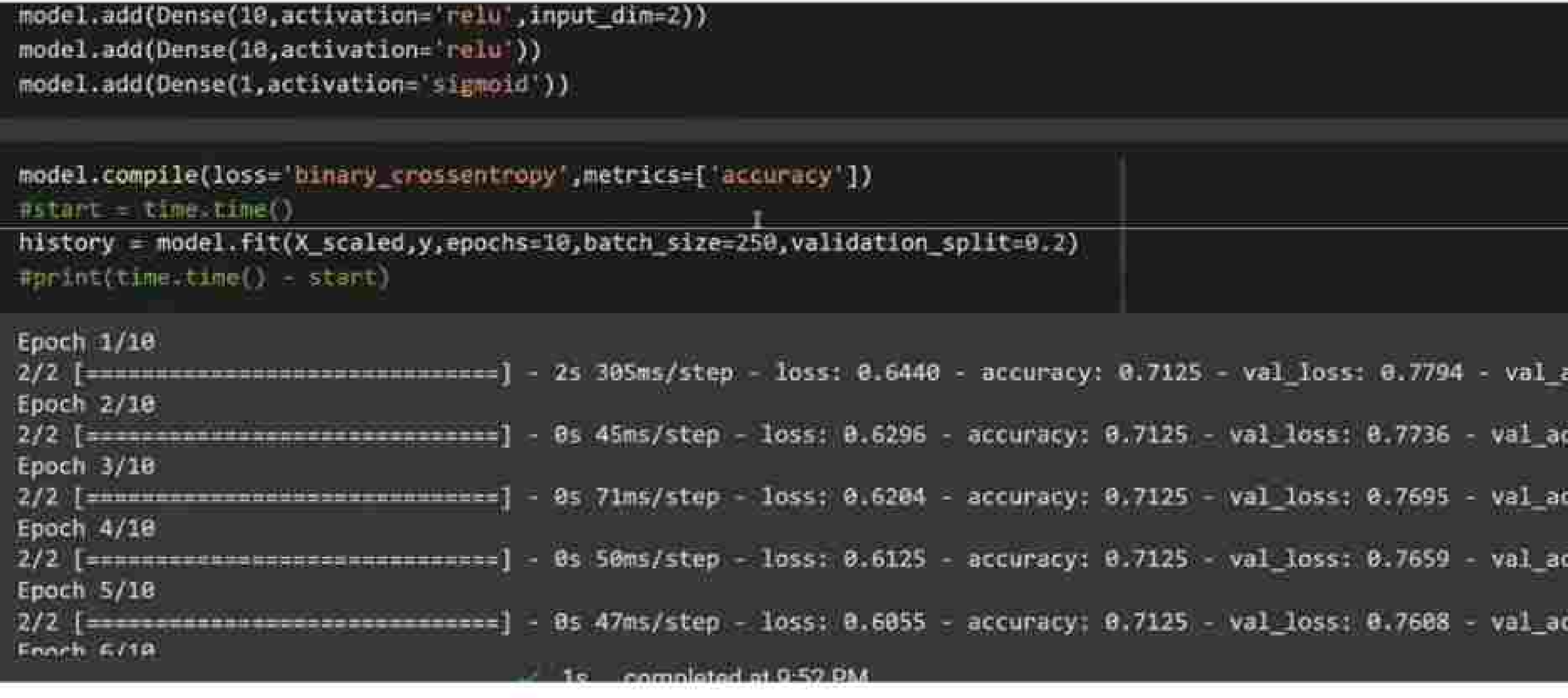
3 150, 150, left 100

1 batch 2 batch 3rd batch



```
model.compile(loss='binary_crossentropy',metrics=['accuracy'])
#start = time.time()
history = model.fit(X_scaled,y,epochs=10,batch_size=150,validation_split=0.2)
#print(time.time() - start)
```

```
Epoch 1/10
3/3 [==============================] - 2s 179ms/step - loss: 0.7233 - accuracy: 0.4031 - val_loss: 0.6462 -
Epoch 2/10
3/3 [==============================] - 0s 61ms/step - loss: 0.7030 - accuracy: 0.4875 - val_loss: 0.6417 -
Epoch 3/10
3/3 [==============================] - 0s 53ms/step - loss: 0.6893 - accuracy: 0.5437 - val_loss: 0.6335 -
Epoch 4/10
3/3 [==============================] - 0s 49ms/step - loss: 0.6771 - accuracy: 0.5719 - val_loss: 0.6314 -
Epoch 5/10
3/3 [==============================] - 0s 32ms/step - loss: 0.6665 - accuracy: 0.6187 - val_loss: 0.6252 -
Epoch 6/10
3/3 [==============================] - 0s 26ms/step - loss: 0.6573 - accuracy: 0.6625 - val_loss: 0.6196 -
Epoch 7/10
3/3 [==============================] - 0s 33ms/step - loss: 0.6492 - accuracy: 0.6938 - val_loss: 0.6178 -
```



```
model.add(Dense(10,activation='relu',input_dim=2))
model.add(Dense(10,activation='relu'))
model.add(Dense(1,activation='sigmoid'))
```

```
model.compile(loss='binary_crossentropy',metrics=['accuracy'])
#start = time.time()
history = model.fit(X_scaled,y,epochs=10,batch_size=250,validation_split=0.2)
#print(time.time() - start)
```

```
Epoch 1/10
2/2 [==============================] - 2s 305ms/step - loss: 0.6440 - accuracy: 0.7125 - val_loss: 0.7794 - val_a
Epoch 2/10
2/2 [==============================] - 0s 45ms/step - loss: 0.6296 - accuracy: 0.7125 - val_loss: 0.7736 - val_ac
Epoch 3/10
2/2 [==============================] - 0s 71ms/step - loss: 0.6204 - accuracy: 0.7125 - val_loss: 0.7695 - val_ac
Epoch 4/10
2/2 [==============================] - 0s 50ms/step - loss: 0.6125 - accuracy: 0.7125 - val_loss: 0.7659 - val_ac
Epoch 5/10
2/2 [==============================] - 0s 47ms/step - loss: 0.6055 - accuracy: 0.7125 - val_loss: 0.7608 - val_ac
Epoch 6/10
```

Stochastic Gradient Descent

- SGD is a variant of the gradient descent optimization algorithm widely used in machine learning and deep learning.
- Unlike batch gradient descent, which computes the gradient using the entire dataset, SGD updates the model parameters using a single training example at a time.

Stochastic Gradient Descent

Working Principle:

- Initialization: Start with initial guess for model parameters.
- For each epoch:
  - Shuffle the training dataset.
  - Iterate through each training example:
    - Compute the gradient of the cost function with respect to the current training example.
    - Update the model parameters using the computed gradient and a predefined learning rate.
- Repeat the process for a fixed number of epochs or until convergence criteria are met.

Mini-batch Gradient Descent

- MBGD strikes a balance between the efficiency of stochastic gradient descent (SGD) and the stability of batch gradient descent by updating the model parameters using a small subset of the training data at each iteration.
- Instead of using the entire training dataset (as in batch gradient descent) or just one example (as in SGD), MBGD divides the dataset into small batches and updates the parameters based on the average gradient computed over each batch.

Mini-batch Gradient Descent

Working Principle:

- Initialization: Start with an initial guess for the model parameters.
- Divide the training dataset into mini-batches of equal size (e.g., 32, 64, or 128 examples per batch).
- For each epoch:
  - Shuffle the training dataset to introduce randomness and prevent the model from getting stuck in local minima.
  - Iterate through each mini-batch:
    - Compute the gradient of the cost function with respect to the mini-batch.
    - Update the model parameters using the computed gradient and a predefined learning rate.
- Repeat the process for a fixed number of epochs or until convergence criteria are met.

Mini-batch Gradient Descent

Advantages:

- Offers a good compromise between the efficiency of SGD and the stability of batch gradient descent.
- Well-suited for training on large datasets that do not fit into memory.

Limitation:

- Requires tuning of hyperparameters such as the learning rate and batch size.

Advanced Optimizers

Optimizers :

- Optimizers adjust model parameters iteratively during training to minimize a loss function, enabling neural networks to learn from data.
- Choosing an appropriate optimizer for a deep learning model is important as it can greatly impact its performance. Optimization algorithms have different strengths and weaknesses and are better suited for certain problems and architectures.
- Some advanced optimizers used in neural networks:
  1. SGD with Momentum
  2. Nesterov Accelerated Gradient (NAG)
  3. AdaGrad (Adaptive Gradient)
  4. Gradient Descent with RMSprop(Root Mean Squared Propagation)
  5. Adam (Adaptive Moment Estimation)

SGD with Momentum

- Momentum optimization is a popular variant of the gradient descent optimization algorithm commonly used to train neural networks. It addresses some of the limitations of basic gradient descent, particularly slow convergence in the presence of flat or small gradients and oscillations in the optimization process.

- In momentum optimization, instead of updating the weights based solely on the current gradient, it also considers the accumulation of past gradients to determine the direction of the update.
- This is achieved by introducing a new parameter called the momentum parameter, denoted by β, which is typically set to a value between 0 and 1.

SGD with Momentum

The update rule for momentum optimization can be expressed as follows:

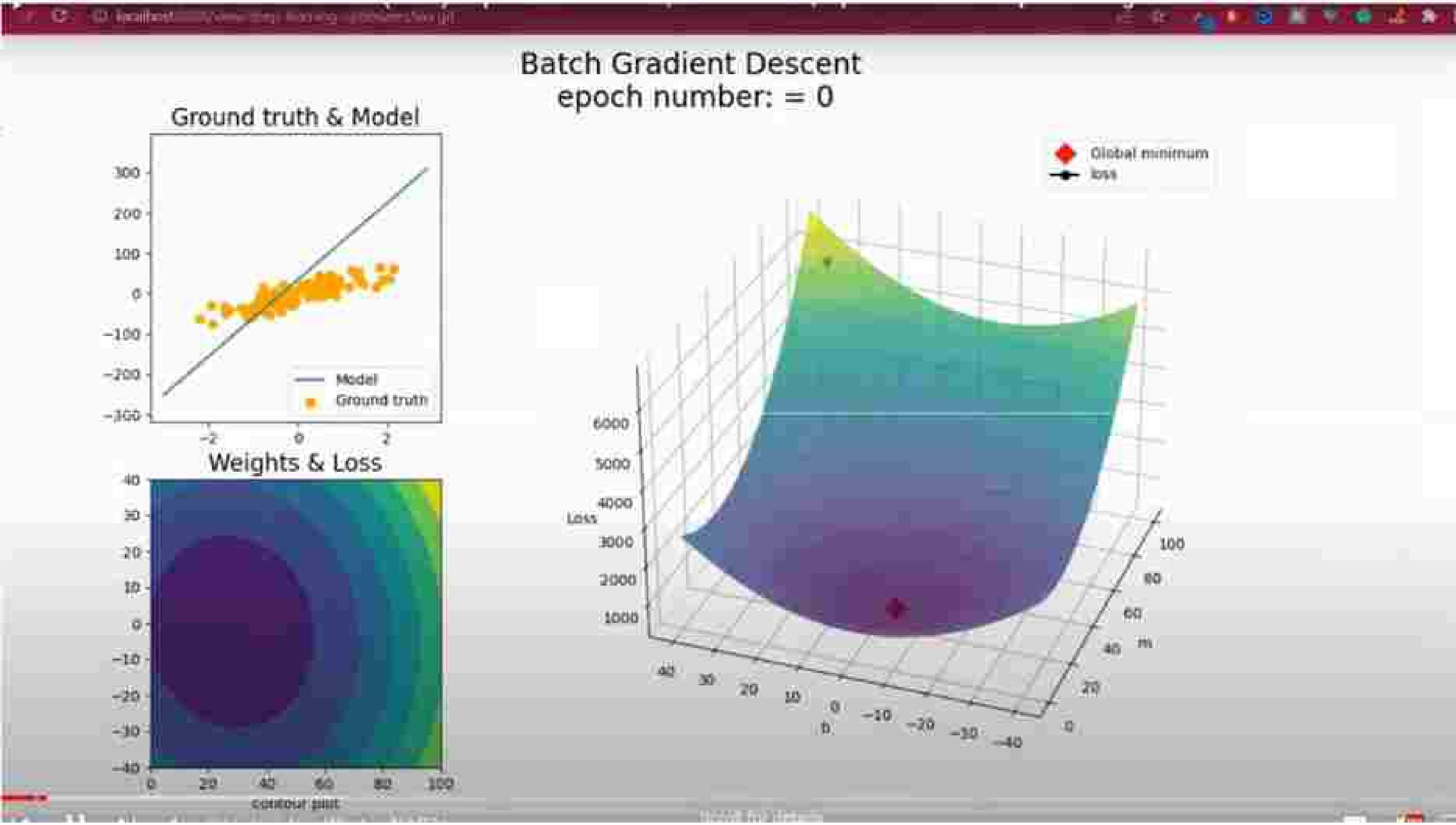
$$v_{t+1} = \beta \cdot v_t + \alpha \cdot \nabla J(\theta)$$
$$\theta_{t+1} = \theta_t - v_{t+1}$$

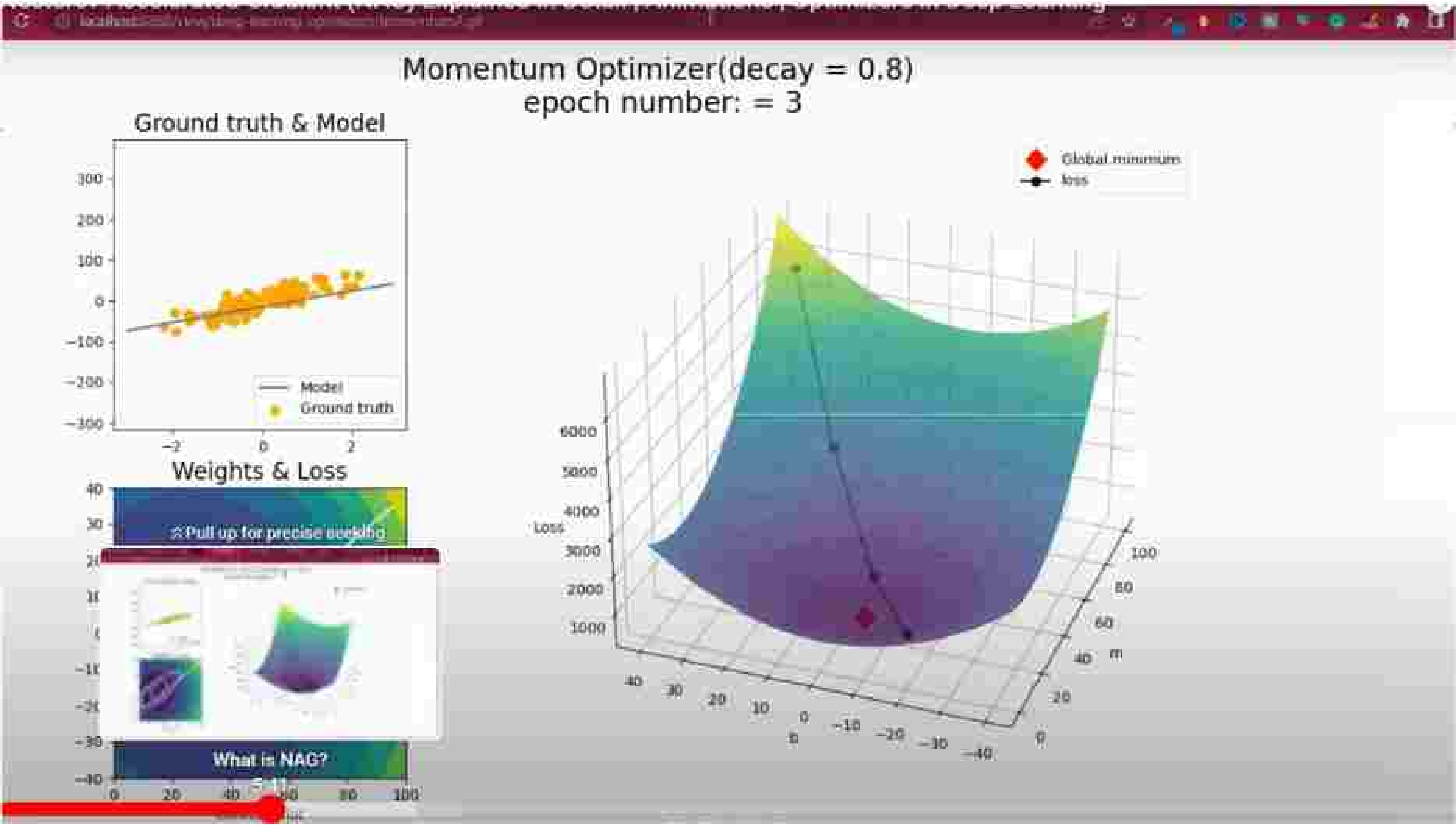
where:

- v_t is the momentum term at iteration t,
- α is the learning rate,
- $\nabla J(\theta)$ is the gradient of the loss function J with respect to the parameters θ, and
- θ_t and θ_{t+1} are the parameters at iterations t and $t+1$, respectively.

SGD with Momentum

- The momentum term Vt is an exponentially weighted moving average of past gradients(Exponentially Weighted Moving Average is a method for smoothing time-series data by assigning exponentially decreasing weights to older observations. It is widely used for trend analysis, noise reduction, and forecasting in various fields.). It accelerates the updates in directions where the gradients point consistently over time and dampens oscillations in directions where the gradients change direction frequently.
- By incorporating momentum, the optimizer gains inertia, enabling it to continue moving in the same direction for a longer time and traverse through regions of flat or small gradients more efficiently. This leads to faster convergence and reduced oscillations during training.
- In short momentum optimization accelerates gradient descent by introducing a momentum term that accumulates past gradients, helping the optimizer navigate through complex optimization landscapes more effectively. It is widely used in practice due to its ability to speed up training and improve convergence for deep learning models.


Batch Gradient Descent
epoch number: = 0
Ground truth & Model
300
200
100
0
−100
−200
−300
−2
0
2
Model
Ground truth
Weights & Loss
40
30
20
10
0
−10
−20
−30
−40
0
20
40
60
80
100
contour plot
Global minimum
loss
6000
5000
4000
3000
2000
1000
Loss
40
30
20
10
0
−10
−20
−30
−40
b
100
80
60
40
20
0
m



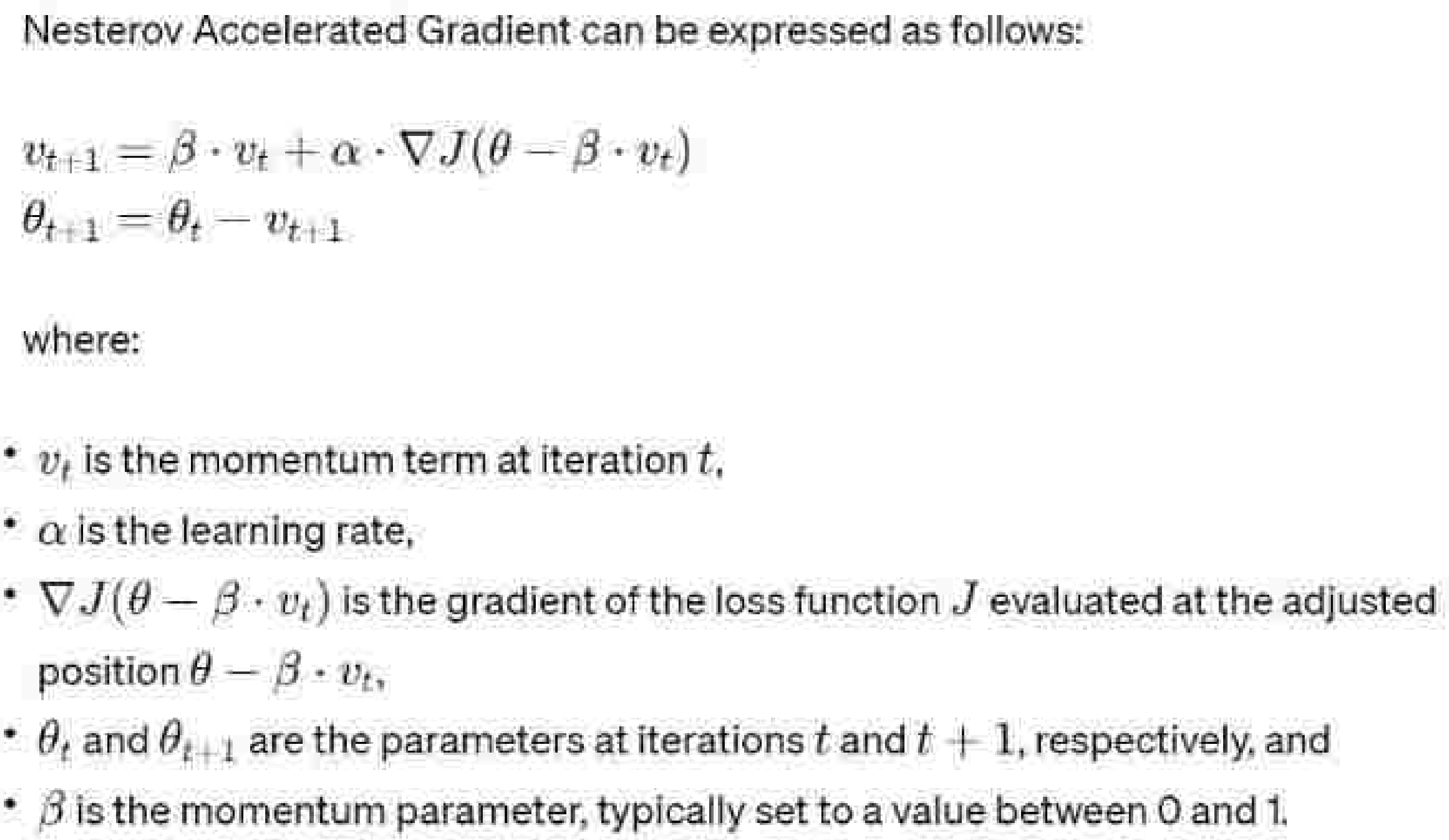
Momentum Optimizer(decay = 0.8)
epoch number: = 3

Nesterov Accelerated Gradient (NAG)

- Nesterov Accelerated Gradient (NAG) is an optimization algorithm that builds upon the momentum optimization method. It aims to improve upon the original momentum approach by addressing the issue of momentum overshooting, which can occur when the current gradient update is combined with the accumulated momentum term.

- In Nesterov Accelerated Gradient, instead of evaluating the gradient at the current position of the parameters, it evaluates the gradient at an adjusted position that takes into account the momentum term. This adjustment is made to anticipate the future position of the parameters based on the momentum.

Nesterov Accelerated Gradient (NAG)



Nesterov Accelerated Gradient can be expressed as follows:

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

where:

- v_t is the momentum term at iteration t,
- α is the learning rate,
- $\nabla J(\theta - \beta \cdot v_t)$ is the gradient of the loss function J evaluated at the adjusted position $\theta - \beta \cdot v_t$,
- θ_t and θ_{t+1} are the parameters at iterations t and $t + 1$, respectively, and
- β is the momentum parameter, typically set to a value between 0 and 1.

Nesterov Accelerated Gradient (NAG)

- The key difference between Nesterov Accelerated Gradient and traditional momentum optimization is that the gradient is evaluated at the "lookahead" position ($\theta - \beta \cdot v_t$) which anticipates the future position of the parameters before updating them with the momentum term. This allows NAG to correct the momentum overshooting problem by incorporating a more accurate gradient estimate.

- By incorporating Nesterov momentum, the optimizer can adjust the momentum term more effectively and reduce oscillations, leading to faster convergence and improved optimization performance compared to traditional momentum optimization.

Deep Learning