PyTorch

Python lists or tuples of numbers are collections of Python objects that are individually allocated in memory.

```
# first let's define a list and see the size it occupies
a = [1.0, 2.0, 3.0]
print(type(a))
print(a.__sizeof__())
```

```
<class 'list'>
64
```

Task 1

```
# define a list of integers and see the size it occupies
# print exactly like we did in previous cell
## code below
```

PyTorch tensors or NumPy arrays, on the other hand, are views over typically) contiguous memory blocks containing unboxed C numeric types rather than Python objects. Each element is 32 bit (4-byte for 32 bit and 8 byte for 64 bit in general) float in this case used in pytorch by default. This means storing a 1D tensor of 1000,000 float numbers will require exactly 4,000,000 contiguous bytes, plus a small overhead for the metadata (such as dimensions and numeric type).

```
import numpy as np
a = np.array([1.0, 2.0, 3.0])
print("a: {}, and it's dtype: {}".format(a, a.dtype))
print(a.nbytes)
```

```
a: [1. 2. 3.], and it's dtype: float64
24
```

```
import torch as t
a = t.ones(3)
print("a: {}, and it's dtpe: {}".format(a, a.dtype))
print("size: ", a.element_size()*a.nelement())
```

```
a: tensor([1., 1., 1.]), and it's dtpe: torch.float32
size:  12
```

```
# we can create a tensor of zeros and replace it's values
my_tensor = t.zeros((3, 2))
my_tensor
```

```
tensor([[0., 0.],
        [0., 0.],
        [0., 0.]])
```

```
# replace it's zeros with 1., 2., 3., 4., 5., 6.
my_tensor[0, 0] = 1.
my_tensor[0, 1] = 2.
my_tensor[1, 0] = 3.
my_tensor[1, 1] = 4.
my_tensor[2, 0] = 5.
my_tensor[2, 1] = 6.
```

```
# print my_tensor
my_tensor
```

```
tensor([[1., 2.],
        [3., 4.],
        [5., 6.]])
```

Task 2

```
# create a tensor of ones with size (5,5) and replace each element's values with any value you like
# then print the tensor
## code below
```

```
# we can pass lists of list to create a tensor
two_D_tensor = t.tensor([[4.0, 1.0], [5.0, 3.0], [2.0, 1.0]])
two_D_tensor
```

```
tensor([[4., 1.],
        [5., 3.],
        [2., 1.]])
```

Task 3

```
# create a list of lists and pass into t.tensor and print your tensor
## code below
```

```
# creating 3-d tensor
random_tensor = t.randn((3,4,3)) # here 3,4,3 ---> refers channels, rows, columns
random_tensor
```

```
tensor([[[-2.1811, -0.8786,  0.5294],
         [-1.5866,  0.4524, -0.9706],
         [-1.7160, -0.0160, -0.0269],
         [-1.7133,  0.4381,  0.0043]],

        [[-0.1985, -1.1853, -0.5703],
         [-1.7658,  0.6095,  0.5504],
         [-0.5733, -1.4506, -0.2268],
         [ 2.0636,  0.9851, -0.5847]],

        [[-0.2807, -0.8224,  0.0069],
         [ 0.0397, -0.9972, -0.8684],
         [ 0.9302, -0.6041,  0.7761],
         [-0.1464, -1.4105,  0.1765]]])
```

```
# indexing tensors
print("Printing 4th row last element of first (4,3) matrix: ", random_tensor[0][3][2])
print("Printing 1st row first element of second (4,3) matrix: ", random_tensor[1][0][0])
print("Printing 2nd row 2nd element of second (4,3) matrix: ", random_tensor[1][1][1])
print("Printing 4th row first element of last (4,3) matrix: ", random_tensor[2][3][0])
```

```
Printing 4th row last element of first (4,3) matrix:  tensor(0.0043)
Printing 1st row first element of second (4,3) matrix:  tensor(-0.1985)
Printing 2nd row 2nd element of second (4,3) matrix:  tensor(0.6095)
Printing 4th row first element of last (4,3) matrix:  tensor(-0.1464)
```

Task 4

```
# Print 3rd row last element of first (4,3) matrix
# Print 2nd row first element of second (4,3) matrix
# Print 1st row 2nd element of first (4,3) matrix
# Print 1st row first element of last (4,3) matrix
```

Task 5

```
# create a 3d tensor of channels=5, rows=6, columns=3
## print your tensor and try to access each element with indexing like we did above
## code below
```

```
# Create a tensor with a fixed value for every element
fixed_tensor = t.full((3, 3), 100)
fixed_tensor
```

```
tensor([[100, 100, 100],
        [100, 100, 100],
        [100, 100, 100]])
```

Task 6

```
# create a tensor of shape of your choice and fill with value 50 like we did above
# print your tensor
```

```
# Concatenate two tensors with compatible shapes
concat_tensor = t.cat((fixed_tensor, t.full((3,3), 50)))
concat_tensor
```

```
tensor([[100, 100, 100],
        [100, 100, 100],
        [100, 100, 100],
        [ 50,  50,  50],
        [ 50,  50,  50],
        [ 50,  50,  50]])
```

Task 7

```
# create two tensors of your favorite shape and concatenate them using t.cat as we did above
# print both tensor before concatenation and also print concatenated tensor
```

```
# Change the shape of a tensor
reshaped = concat_tensor.reshape(3, 3, 2)
reshaped
```

```
tensor([[[100, 100],
         [100, 100],
         [100, 100]],

        [[100, 100],
         [100,  50],
         [ 50,  50]],

        [[ 50,  50],
         [ 50,  50],
         [ 50,  50]]])
```

Task 8

```
# create a tensor of shape (5,4,2) and then reshape it into some other dimension of your choice
```

```
# creating tensor from numpy array
x = np.array([[1, 2], [3, 4.]])
array_to_tensor = t.from_numpy(x)
array_to_tensor
```

```
tensor([[1., 2.],
        [3., 4.]], dtype=torch.float64)
```

```
# Convert a torch tensor to a numpy array
tensor_to_array = array_to_tensor.numpy()
tensor_to_array
```

```
array([[1., 2.],
       [3., 4.]])
```

Task 9

```
# create a 2-D numpy array of your choice and then convert into tensor using from_numpy and convert tensor back
```

```
# multiplying tensors
a = t.tensor([[1,2,3], [4,5,6]]) # observe the shape (2,3)
b = t.tensor([[0,1], [4,5], [1,2]]) # observe the shape (3,2)
a.matmul(b)
```

```
tensor([[11, 17],
        [26, 41]])
```

Task 10

```
# create two tensors like we did above and multiply them using matmul
```

```
# check it's documentation
? t.linalg.matrix_power
```

```
# To compute nth power of a square matrix
sq_matrix = t.ones((2,2))
print("sq matrix: ", sq_matrix)
t.linalg.matrix_power(sq_matrix, 3)
```

```
sq matrix:  tensor([[1., 1.],
        [1., 1.]])
tensor([[4., 4.],
        [4., 4.]])
```

Tensor Operations and How to do Gradients

```
# Create tensors.
x = t.tensor(7.)
w = t.tensor(4., requires_grad=True)
b = t.tensor(3., requires_grad=True)
x, w, b
```

```
(tensor(7.), tensor(4., requires_grad=True), tensor(3., requires_grad=True))
```

```
# Making the equation by arithmetic operation
y = w * x + b
y
```

```
tensor(31., grad_fn=<AddBackward0>)
```

```
# Computing derivatives
y.backward()
```

```
# Display gradients
print('dy/dx:', x.grad)
print('dy/dw:', w.grad)
print('dy/db:', b.grad)
```

```
dy/dx: None
dy/dw: tensor(7.)
dy/db: tensor(1.)
```

As expected, `dy/dw` has the same value as `x`, i.e., `7`, and `dy/db` has the value `1`. Note that `x.grad` is `None` because `x` doesn't have `requires_grad` set to `True`.

The "grad" in `w.grad` is short for *gradient*, which is another term for derivative. The term *gradient* is primarily used while dealing with vectors and matrices.

Task 11

```
# create 3 tensors x,w,b of your choice and then compute y=w*x + b. After defining y, compute gradients like we
# observe all the steps of tensor operations and gradients. Look into the documentation of tensor if needed.
```

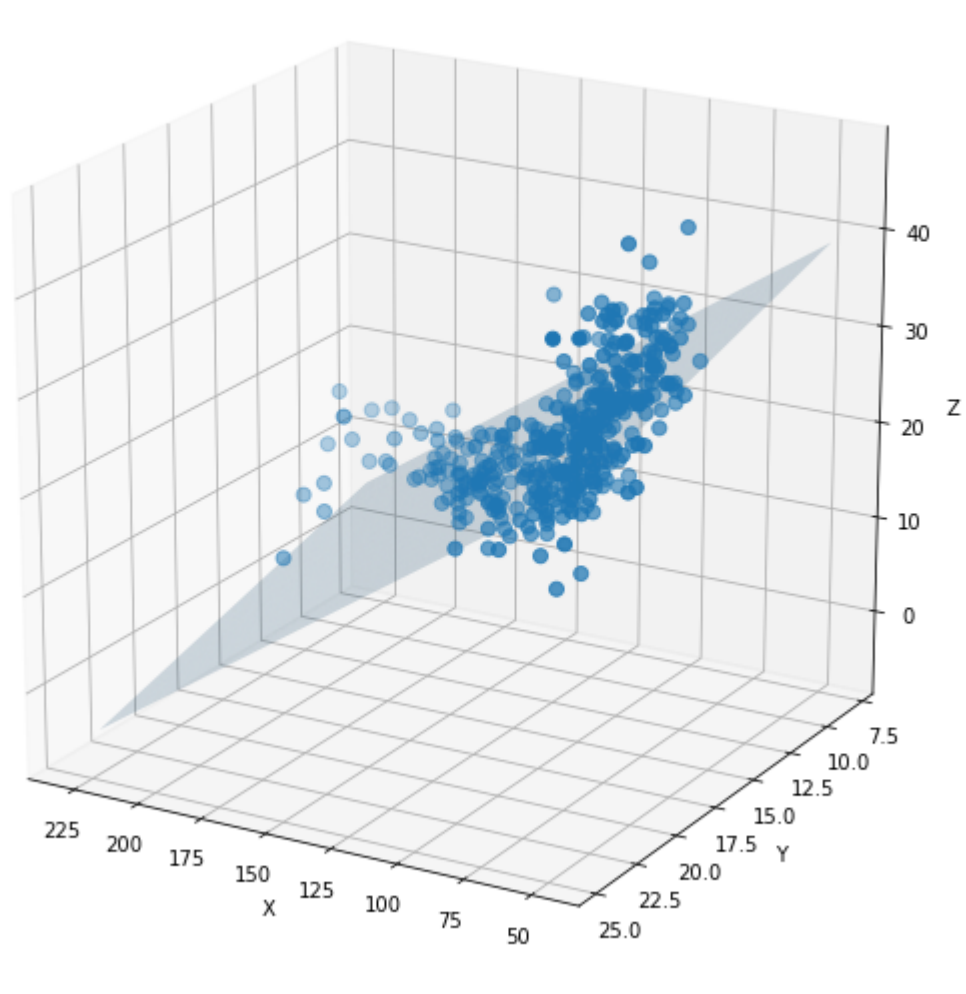
Task 12

Understand the whole Linear Regression From Scratch in next section. Then implement the same by taking input as area of the house and output as price of the house. You can take some random inputs of your own choice just like we did.

Linear Regression From Scratch

In a linear regression model, each target variable is estimated to be a weighted sum of the input variables, offset by some constant, known as a bias :

```
percent  = w11 * maths + w12 * physics + w13 * chemistry + b1
```




```
# Here i'm taking marks of maths, physics, chemistry as input and i'm creating an array of it
marks = np.array([[60, 70, 75],
                  [70, 75, 80],
                  [70, 72, 77],
                  [90, 91, 95],
                  [95, 95, 99],
                  [50, 60, 65]], dtype='float32')

# target is percentage which a person will get
percent = np.array([68.3, 75.0, 76.3, 92.0, 96.3, 58.3], dtype='float32')
```

```
# converting array into tensor
marks = t.from_numpy(marks)
percent = t.from_numpy(percent)

# printing marks and percentage both
print(marks)
print(percent)
```

```
tensor([[60., 70., 75.],
        [70., 75., 80.],
        [70., 72., 77.],
        [90., 91., 95.],
        [95., 95., 99.],
        [50., 60., 65.]])
tensor([68.3000, 75.0000, 76.3000, 92.0000, 96.3000, 58.3000])
```

The weights and biases (w11, w12,... w23, b1 & b2) can also be represented as matrices, initialized as random values. The first row of w and the first element of b are used to predict the first target variable which is percent.

Notice, our input is having 3 features, and output is 1, so our weight will be (1,3) shape and bias will be 1.

```
# creating weights and biases using randn function
marks_wt = t.randn(1, 3, requires_grad=True)
marks_bias = t.randn(1, requires_grad=True)
print(marks_bias)
print(marks_wt)
```

```
tensor([-0.0187], requires_grad=True)
tensor([[ 2.8057, -0.3628, -1.2635]], requires_grad=True)
```

```
# creating the model
#  @ represents matrix multiplication in PyTorch, and the .t method returns the transpose of a tensor.
def marks_model(x):
  return x @ marks_wt.t() + marks_bias
```

```
# Generate predictions
percent_pred = marks_model(marks)
print(percent_pred)
```

```
tensor([[ 48.1662],
        [ 68.0918],
        [ 72.9706],
        [ 99.4489],
        [106.9723],
        [ 36.3720]], grad_fn=<AddBackward0>)
```

```
# Compare with percent
print(percent)
```

```
tensor([68.3000, 75.0000, 76.3000, 92.0000, 96.3000, 58.3000])
```

You can see a big difference between our model's predictions and the actual targets because we've initialized our model with random weights and biases. Obviously, we can't expect a randomly initialized model to just work.

Loss function

How to define a loss function ?

- Calculate the difference between the two matrices (`marks` and `percent`).
- Square all elements of the difference matrix to remove negative values.
- Calculate the average of the elements in the resulting matrix.

You will get in the end, **mean squared error** (MSE).

```
# t.sum returns the sum of all the elements in a tensor.
# The .numel method of a tensor returns the number of elements in a tensor.
# MSE loss
def mse(t1, t2):
    diff = t1 - t2
    return t.sum(diff * diff) / diff.numel()
```

```
# Let's check the loss
percent_loss = mse(percent_pred, percent)
print(percent_loss)
```

```
tensor(841.5394, grad_fn=<DivBackward0>)
```

Here's how we can interpret the result: On average, each element in the prediction differs from the actual target by the square root of the loss.

```
# This is formatted as code
```

Compute gradients

With PyTorch, we can automatically compute the gradient or derivative of the loss w.r.t. to the weights and biases because they have requires_grad set to True. We'll see how this is useful in just a moment.

```
# Compute gradients
percent_loss.backward()
```

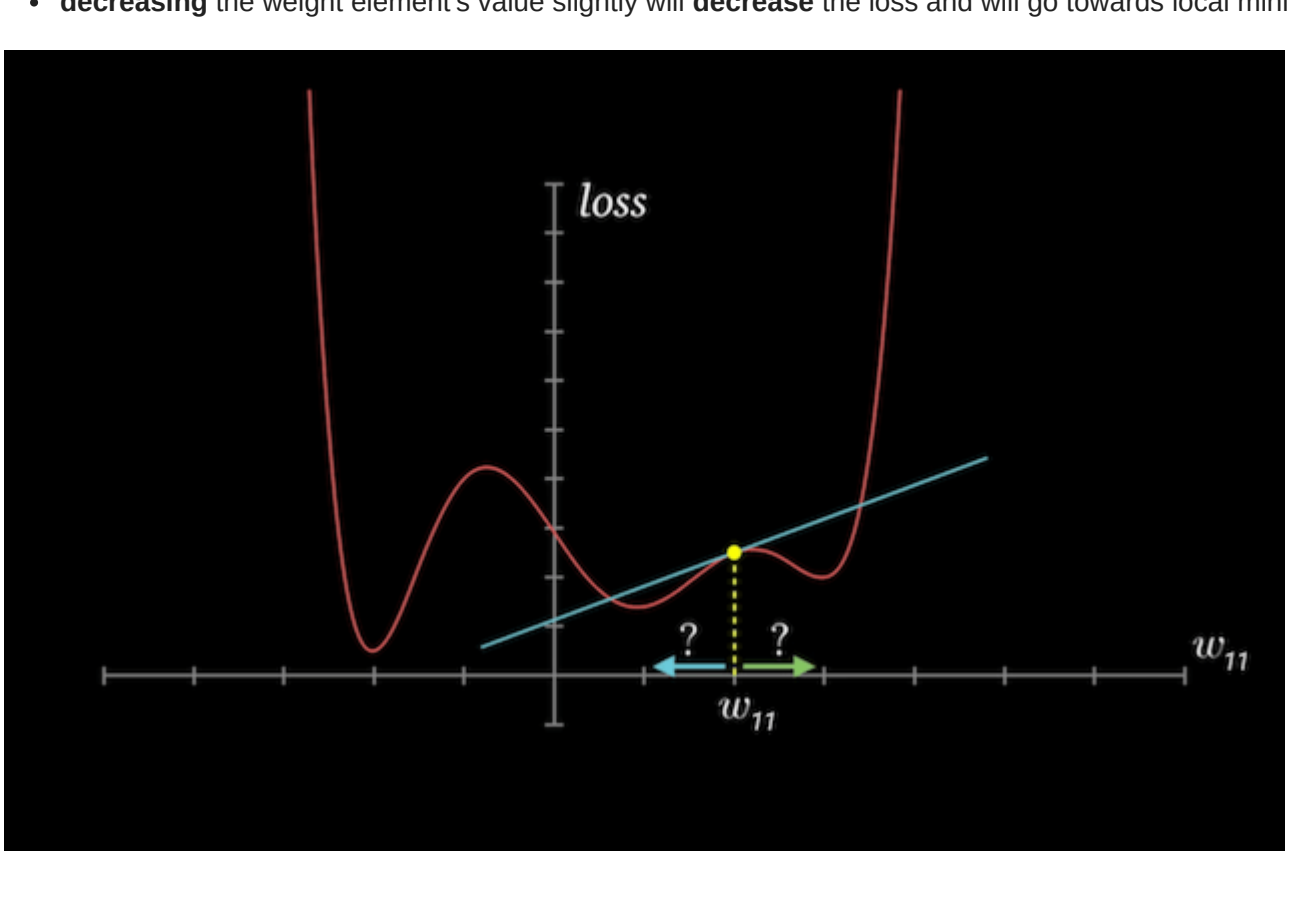
```
# check weights and it's gradient
print(marks_wt)
print(marks_wt.grad)
```

```
tensor([[ 2.8057, -0.3628, -1.2635]], requires_grad=True)
tensor([[ -34.6004, -282.7236, -356.6945]])
```

How to adjust weights and biases to reduce the loss?

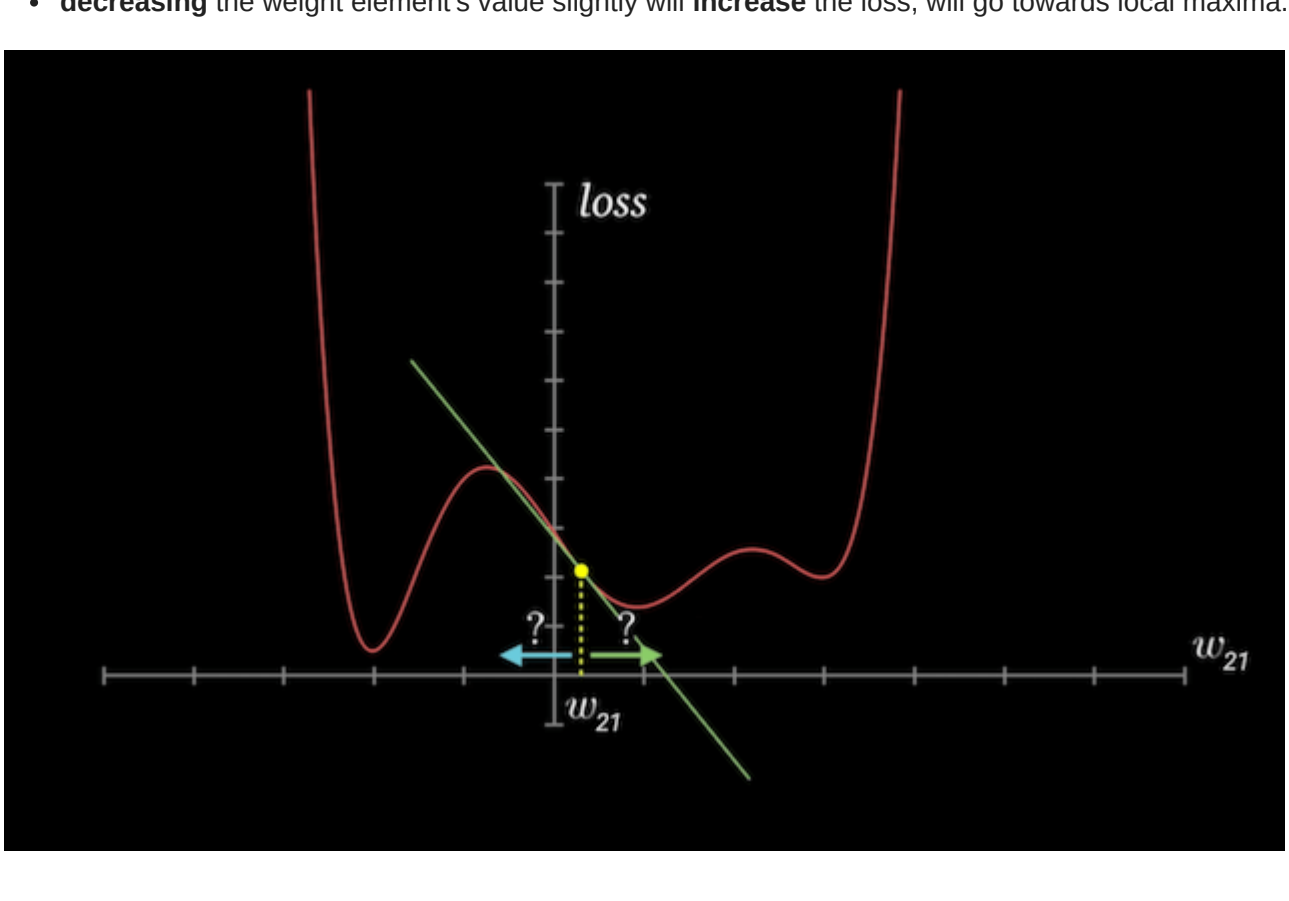
If a gradient element is **positive**:

- **increasing** the weight element's value slightly will **increase** the loss, and will go towards local maxima.
- **decreasing** the weight element's value slightly will **decrease** the loss and will go towards local minima.

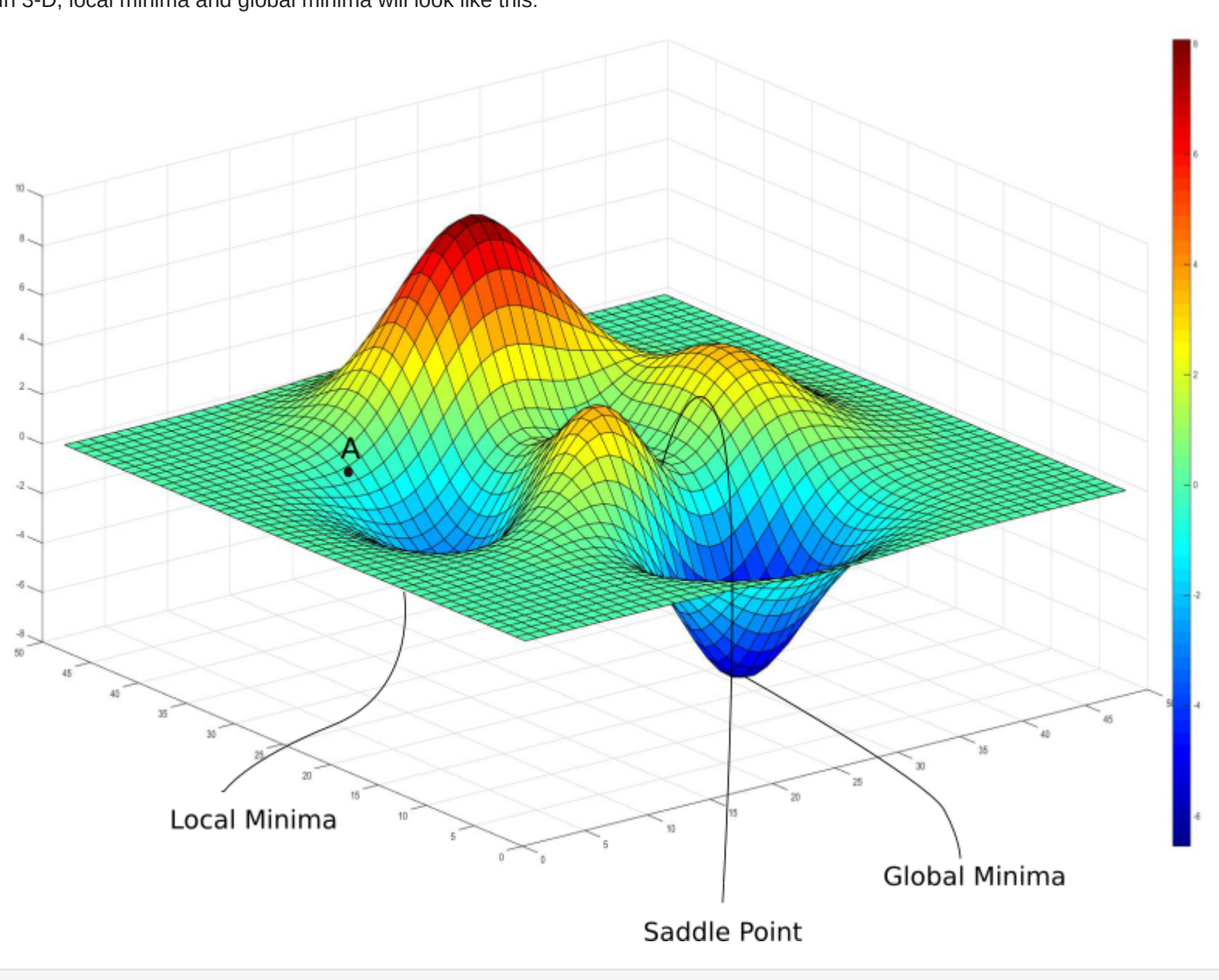


If a gradient element is **negative**:

- **increasing** the weight element's value slightly will **decrease** the loss, will go towards local minima.
- **decreasing** the weight element's value slightly will **increase** the loss, will go towards local maxima.



In 3-D, local minima and global minima will look like this:




```
# We can subtract from each weight element a small quantity proportional to the derivative of the loss w.r.t. t
with t.no_grad():

    marks_wt -= marks_wt.grad * 1e-5
    marks_bias -= marks_bias.grad * 1e-5
    marks_wt.grad.zero_() # note this step of resetting the gradient to zero
    marks_bias.grad.zero_() # We need to do this because PyTorch accumulates gradients, which may lead to unexp
```

Here 1e-5 is the learning rate. We multiply the gradients with a very small number (10^-5 in this case) to ensure that we don't modify the weights by a very large amount. We want to take a small step in the downhill direction of the gradient, not a giant leap. This number is called the learning rate of the algorithm.

We use torch.no_grad to indicate to PyTorch that we shouldn't track, calculate, or modify gradients while updating the weights and biases.

```
# Let's verify that the loss is actually lower
percent_pred = marks_model(marks)
percent_loss = mse(percent_pred, percent)
print(percent_loss)
```

```
tensor(839.7478, grad_fn=<DivBackward0>)
```

Before we proceed, we reset the gradients to zero by invoking the .zero_() method. We need to do this because PyTorch accumulates gradients. Otherwise, the next time we invoke .backward on the loss, the new gradient values are added to the existing gradients, which may lead to unexpected results.

```
# You can check if gradients are set to zero or not
print(marks_wt.grad)
print(marks_bias.grad)
```

```
tensor([[0., 0., 0.]])
tensor([0.])
```

Training the model with these steps:

1) Generating predictions

2) Calculating the loss

3) Compute gradients w.r.t the weights and biases

4) Adjust the weights by subtracting a small quantity proportional to the gradient

5) Reset the gradients to zero

```
# Generate predictions
percent_pred = marks_model(marks)
print(percent_pred)
```

```
tensor([[ 48.6525],
        [ 68.6135],
        [ 73.4732],
        [100.0763],
        [107.6270],
        [ 36.7909]], grad_fn=<AddBackward0>)
```

```
# Calculate the loss
percent_loss = mse(percent_pred, percent)
print(percent_loss)
```

```
tensor(839.7478, grad_fn=<DivBackward0>)
```

```
# Compute gradients and print gradients of weight and bias
percent_loss.backward()
print(marks_wt.grad)
print(marks_bias.grad)
```

```
tensor([[  45.5616, -198.1306, -267.1762]])
tensor([-10.3223])
```

Let's update the weights and biases using the gradients computed above.

```
# Adjust weights & reset gradients
with t.no_grad():
    marks_wt -= marks_wt.grad * 1e-5
    marks_bias -= marks_bias.grad * 1e-5
    marks_wt.grad.zero_()
    marks_bias.grad.zero_()
```

```
# looking at new weight and bias
print(marks_wt)
print(marks_bias)
```

```
tensor([[ 2.8056, -0.3580, -1.2573]], requires_grad=True)
tensor([-0.0185], requires_grad=True)
```

With the new weights and biases, the model should have a lower loss

```
# Calculate loss
preds = marks_model(marks)
loss = mse(preds, percent)
print(loss)
```

```
tensor(838.7366, grad_fn=<DivBackward0>)
```

We have achieved a little reduction in the loss merely by adjusting the weights and biases slightly using gradient descent.

To reduce the loss further, we can repeat the process of adjusting the weights and biases using the gradients multiple times. Each iteration is called an epoch. Let's train the model for 50 epochs.

```
# training for multiple epochs

for i in range(150):
  percent_pred = marks_model(marks)
  percent_loss = mse(percent_pred, percent)
  percent_loss.backward()
  with t.no_grad():
    marks_wt -= marks_wt.grad * 1e-5
    marks_bias -= marks_bias.grad * 1e-5
    marks_wt.grad.zero_()
    marks_bias.grad.zero_()
```

```
# calculate loss

percent_pred = marks_model(marks)
percent_loss = mse(percent_pred, percent)
print(percent_loss)
```

```
tensor(706.0983, grad_fn=<DivBackward0>)
```

The loss is now much lower than its initial value. Let's look at the model's predictions and compare them with the targets.

```
# result is good and we can make it better with more epochs
print(percent_pred)
print(percent)
```

```
tensor([[ 53.1078],
        [ 70.3147],
        [ 73.7257],
        [ 98.8381],
        [105.7360],
        [ 41.5860]], grad_fn=<AddBackward0>)
tensor([68.3000, 75.0000, 76.3000, 92.0000, 96.3000, 58.3000])
```

FEEDBACK FORM

Please help us to improve by filling this form.
https://forms.zohopublic.in/cloudyml/form/CloudyMLDeepLearningFeedbackForm/formperma/VCFbldnXAnbcgAIl0lWv2blgHdSldheO4RfktMdg