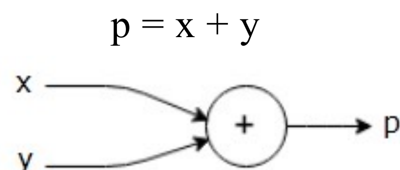
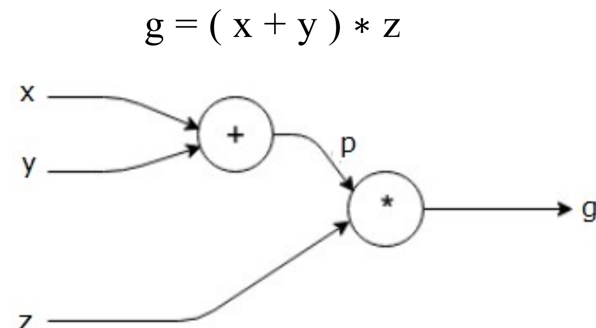
PyTorch Autograd

Computation graph:



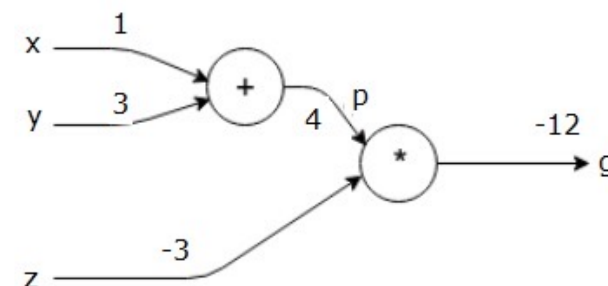


Forward Pass:

The forward pass refers to the process of **passing input data through a neural network to obtain output predictions**.

During the forward pass, input data is transformed through the network's layers, resulting in the model's predicted outputs. The forward pass is the inference process of a neural network and does not involve parameter updates.

p = 4 and z = -3 to get g = -12



Reference:
https://www.tutorialspoint.com/python_deep_learning/python_deep_learning_computational_graphs.htm

Backward pass (backpropagation, 反向傳播)

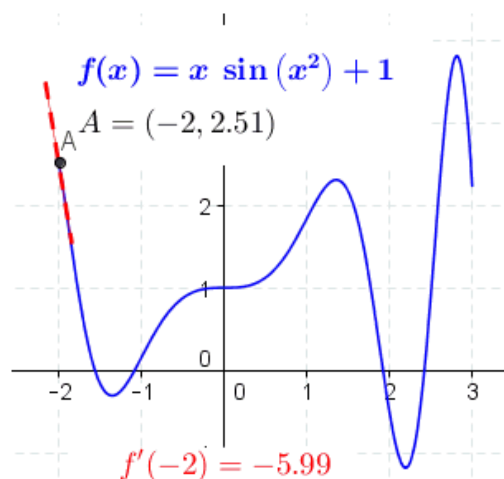
Backpropagation is an algorithm used to **compute the gradients of the loss function** with respect to the parameters of a neural network.

By using backpropagation, we can calculate how the loss changes with respect to each parameter, enabling us to update the parameters using optimization algorithms like gradient descent to minimize the loss.

Gradient:

A gradient is the **slope or directional derivative of a function at a certain point (函數在某一點的斜率/導數)**, indicating the rate of change of the function at that point.

In deep learning, the gradient typically refers to the **partial derivatives of the loss function (損失函數對於模型參數的偏導數)** with respect to the model parameters. It tells us which direction in parameter space leads to an increase or decrease in the loss, helping us update the parameters to minimize the loss.



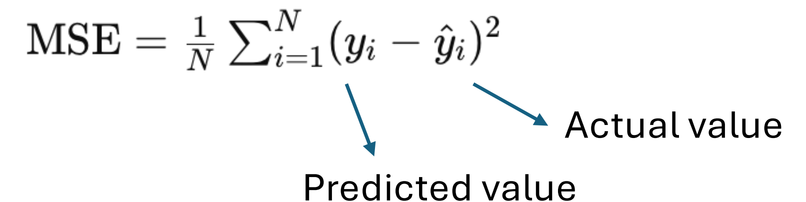
https://en.wikipedia.org/wiki/Slope

differentiate	$y = x\sin(x^2) + 1$

$$y'(x) = \sin(x^2) + 2x^2\cos(x^2)$$

Loss function:

- The loss function is a function that measures the difference or error between the predicted output of a model and the actual target value.
- In supervised learning, where we have both input features and corresponding target labels, the **loss function quantifies how well the model's predictions match the true targets**.
- The goal during training is to **minimize this loss function**, as a lower value indicates that the model is making more accurate predictions.



e.g. Mean Squared Error, MSE (最小化均方誤差)

Example of loss function (1/2)

Predicting housing prices:

Suppose we have a dataset with features like the **size of the house** (in square feet, sqft) and the **number of bedrooms**, and our goal is to predict the price of the house.

$$y = w_1 \times \text{size} + w_2 \times \text{number of bedrooms}$$

where w_1, and w_2 are the weights of the model.

House	Size (sqft)	Number of bedrooms	Actual price	Predicted price A w_1=100 w_2=10,000	Predicted price B w_1=200 w_2=12,000
House 1	1500	3	300,000	100*1500+10,000*3 =180,000	200*1500+12,000*3 =336,000
House 2	2000	4	400,000	100*2000+10,000*4 =240,000	200*2000+12,000*4 =448,000
House 3	1200	2	250,000	100*1200+10,000*2 =140,000	200*1200+12,000*2 =264,000

$$MSE_A = \frac{(180{,}000 - 300{,}000)^2 + (240{,}000 - 400{,}000)^2 + (140{,}000 - 250{,}000)^2}{3} = 17{,}366{,}666{,}666$$

$$MSE_B = \frac{(336{,}000 - 300{,}000)^2 + (448{,}000 - 400{,}000)^2 + (264{,}000 - 250{,}000)^2}{3} = 1{,}265{,}333{,}333$$

Example of loss function (2/2)

- To find the optimal weights (w_1, w_2) for our linear regression model using **gradient descent (梯度下降)**,
- we need to iteratively update the weights to minimize the Mean Squared Error (MSE) loss function.
- Here's how the gradient descent algorithm works:

```
1.Initialize the weights (w1 , w2 ) to some random values or zeros.
2.Calculate the gradient of the loss function with respect to each weight.
3.Update the weights in the opposite direction of the gradient to minimize the loss.
4.Repeat steps 2 and 3 until convergence (收斂) (until the change in the loss function becomes
very small or after a fixed number of iterations).
```

The update rule for each weight (w_i) at each iteration of gradient descent is given by:

gradient

$$w_i = w_i - \alpha \frac{\partial}{\partial w_i} \mathrm{MSE}$$

where α is the learning rate, a **hyperparameter** that controls the size of the steps we take during optimization.

gradientDescent.py

```
1 import torch
2
3 # 定義訓練資料
4 sizes = torch.tensor([1500, 2000, 1200], dtype=torch.float32)
5 bedrooms = torch.tensor([3, 4, 2], dtype=torch.float32)
6 prices = torch.tensor([300000, 400000, 250000], dtype=torch.float32)
7
8 # 初始化權重
9 w1 = torch.randn(1, requires_grad=True)
10 w2 = torch.randn(1, requires_grad=True)
11
12 # 定義訓練迴圈
13 learning_rate = 0.0000001
14 epochs = 10000
15
16 for epoch in range(epochs):
17     # 預測房價
18     predictions = w1 * sizes + w2 * bedrooms
19
20     # 計算損失
21     loss = torch.mean((predictions - prices) ** 2)
22
23     # 使用反向傳播計算梯度
24     loss.backward()
25
26     # 更新權重
27     with torch.no_grad():
28         w1 -= learning_rate * w1.grad
29         w2 -= learning_rate * w2.grad
30
31         # 將梯度歸零
32         w1.grad.zero_()
33         w2.grad.zero_()
34
35     if epoch % 1000 == 0:
36         print(f'Epoch {epoch+1}, Loss: {loss.item()}')
37
38 # Weighting
39 print(f'weighting：w1 = {w1.item()}, w2 = {w2.item()}')
40
```

create a tensor with a shape of (1,), meaning a one-dimensional tensor containing a single element. This element is sampled randomly from a standard normal distribution. (從常態分佈（mean=0, variance=1）中隨機取樣)

```
Epoch 1, Loss: 104347557888.0
Epoch 1001, Loss: 27090582.0
Epoch 2001, Loss: 27090094.0
Epoch 3001, Loss: 27089592.0
Epoch 4001, Loss: 27089102.0
Epoch 5001, Loss: 27088650.0
Epoch 6001, Loss: 27088178.0
Epoch 7001, Loss: 27087710.0
Epoch 8001, Loss: 27087170.0
Epoch 9001, Loss: 27086784.0
weighting：w1 = 201.56507873535156, w2 = -2.382746219635098
```

$y = w_1 \times$ size + $w_2 \times$ number of bedrooms
where w_1, and w_2 are the weights of the model.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$$

Actual value

Predicted value

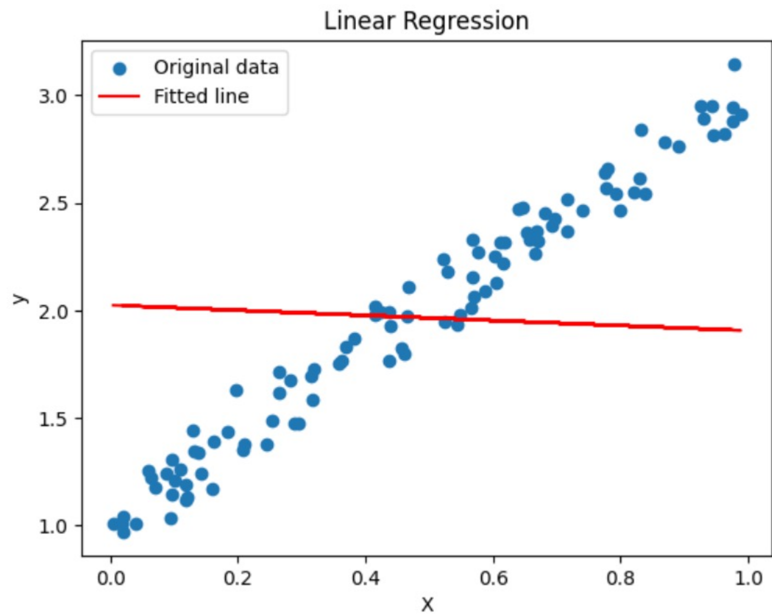
$$w_i = w_i - \alpha\frac{\partial}{\partial w_i}\text{MSE}$$

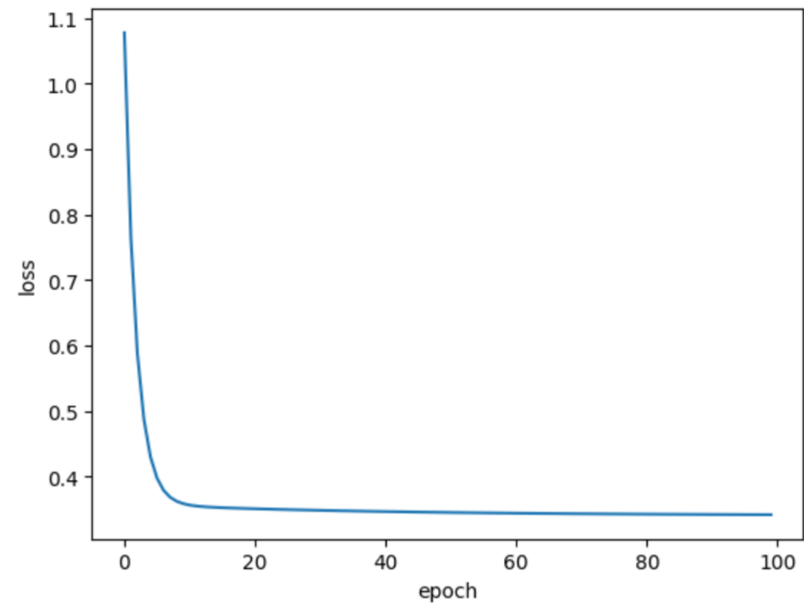
torch.no_grad() is utilized within the training loop when updating the weight parameters. We only require the updated values of the weights and not their gradients for this operation, using torch.no_grad() helps improve training efficiency and reduces memory consumption.

linearRegression.py

```
import torch
import numpy as np
import matplotlib.pyplot as plt



# Generate simulated data
np.random.seed(0)
X = np.random.rand(100, 1)
y = 2 * X + 1 + np.random.randn(100, 1) * 0.1

# Convert data to PyTorch tensors
X_tensor = torch.tensor(X, dtype=torch.float32)
y_tensor = torch.tensor(y, dtype=torch.float32)


# Initialize weights
w = torch.randn(1, requires_grad=True)
b = torch.randn(1, requires_grad=True)


# Set learning rate and number of epochs
learning_rate = 0.1
epochs = 100
losses = []

# Training loop
for epoch in range(epochs):
    # Predict
    y_pred = torch.matmul(X_tensor, w) + b

    # Loss
    loss = torch.mean((y_pred - y_tensor)**2)

    # gradients using autograd, 使用自動微分計算梯度
    loss.backward()
    losses.append(loss.item())

    # Update weights
    with torch.no_grad():
        w -= learning_rate * w.grad
        b -= learning_rate * b.grad

        # Zero gradients, 將梯度歸零
        w.grad.zero_()
        b.grad.zero_()

    if epoch % 10 == 0:
        print(f'Epoch {epoch+1}, Loss: {loss.item()}')

# Print trained weights
print(f'Trained weights：w = {w.item()}, b = {b.item()}')

# Plot data and fitted line
plt.scatter(X, y, label='Original data')
plt.plot(X, w.detach().numpy() * X + b.detach().numpy(), color='red', label='Fitted line')
plt.xlabel('X')
plt.ylabel('y')
plt.title('Linear Regression')
plt.legend()
plt.show()

plt.plot(losses)
plt.xlabel("epoch")
plt.ylabel("loss")
plt.show()
```

```
Epoch 1, Loss: 1.077553629875183
Epoch 11, Loss: 0.3561798334121704
Epoch 21, Loss: 0.35058942437171936
Epoch 31, Loss: 0.3481489419937134
Epoch 41, Loss: 0.34629207849502563
Epoch 51, Loss: 0.34487465023994446
Epoch 61, Loss: 0.3437926769256592
Epoch 71, Loss: 0.3429667055606842
Epoch 81, Loss: 0.34233614802360535
Epoch 91, Loss: 0.34185487031936646
Trained weights：w = -0.11859240382909775, b = 2.024920701980591
```





Useful functions: `torch.randn()`

```
torch.randn(2, 3, dtype=torch.float64)
torch.randn(2, 3, device='cuda')

tensorA = torch.empty(2, 3)
torch.randn(2, 3, out=tensorA)
```

```
1 import torch
2 print(torch.randn(2,3))
```

```
tensor([[ 0.1058,  0.4598,  0.7672],
        [-0.7206, -0.8602, -0.2089]])
```

```
1 import torch
2 print(torch.randn(4,6))
```

```
tensor([[ 0.2374,  0.7092, -0.4115, -0.0434, -1.1513,  1.5934],
        [ 0.8153, -1.9557, -0.5553, -0.0678, -1.7807,  2.1019],
        [-0.0523,  0.2248, -0.7518, -0.6969,  2.5506, -0.8744],
        [ 0.7808, -0.0385, -0.9007,  0.1005, -1.1714, -0.3219]])
```

```
1 import torch
2 #print(torch.randn(4,6))
3 print(torch.arange(1,20, step=2))
4 print(torch.arange(1,20, step=3))
5 print(torch.arange(5,20, step=5))
```

```
tensor([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19])
tensor([ 1,  4,  7, 10, 13, 16, 19])
tensor([ 5, 10, 15])
```

```
1 import torch
2 print(torch.logspace(1,20, steps=2))
3 print(torch.logspace(5,20, steps=2))
4 print(torch.logspace(5,20, steps=5))
```

```
tensor([1.0000e+01, 1.0000e+20])
tensor([1.0000e+05, 1.0000e+20])
tensor([1.0000e+05, 5.6234e+08, 3.1623e+12, 1.7783e+16, 1.0000e+20])
```

Useful functions:

torch.bernoulli()	Generates binary random variables (0 or 1) with a Bernoulli distribution.	
torch.cauchy()	Generates random numbers from a Cauchy distribution.	
torch.exponential()	Generates random numbers from an exponential distribution.	
torch.geometric()	Generates random numbers from a geometric distribution.	
torch.log_normal()	Generates random numbers from a log-normal distribution.	
torch.normal()	Generates random numbers from a normal distribution.	
torch. randint()	Generates random integers from a discrete uniform distribution.	torch.randint(low=0, high=10, size=(2, 3)) 生成一個形狀為(2, 3)的張量，數值是0到9之間的隨機整數。
torch.uniform()	Generates random numbers from a uniform distribution.	

torch.reshape()

```python
import torch

# Input tensor.
input_tensor = torch.tensor([[1, 2], [3, 4]])

# The new shape to reshape the input tensor.
output_tensor = torch.reshape(input_tensor, (4, 1))

print("Input Tensor:")
print(input_tensor)
print("Reshaped Tensor:")
print(output_tensor)
```

```
Input Tensor:
tensor([[1, 2],
        [3, 4]])
Reshaped Tensor:
tensor([[1],
        [2],
        [3],
        [4]])
```

torch.transpose()

```python
import torch

# Input tensor.
input_tensor = torch.tensor([[1, 2], [3, 4]])

# 將輸入張量的第一個維度和第二個維度交換
output_tensor = torch.transpose(input_tensor, 0, 1)

print("Input Tensor:")
print(input_tensor)
print("Transposed Tensor:")
print(output_tensor)
```

```
Input Tensor:
tensor([[1, 2],
        [3, 4]])
Transposed Tensor:
tensor([[1, 3],
        [2, 4]])
```

torch.cat()

```python
import torch

# Concatenates the given sequence of tensors in the
specified dimension.
# 將指定維度上的一系列張量連接在一起
tensor1 = torch.tensor([[1, 2], [3, 4]])
tensor2 = torch.tensor([[5, 6]])

#
result = torch.cat((tensor1, tensor2), dim=0)

print("Concatenated Tensor:")
print(result)
```

```
Concatenated Tensor:
tensor([[1, 2],
        [3, 4],
        [5, 6]])
```

torch.chunk()

```python
import torch

# Splits a tensor into a specific number of chunks
along a given dimension.
# 將張量沿著指定維度分成特定數量的塊
tensor = torch.tensor([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])


# 將張量分成兩個塊
chunks = torch.chunk(tensor, 2, dim=0)

print("Chunks:")
for chunk in chunks:
print(chunk)
```

```
Chunks:
tensor([1, 2, 3, 4, 5])
tensor([ 6,  7,  8,  9, 10])
```

torch.split()

```python
import torch

# 創建一個形狀為(3, 6)的張量
tensor = torch.tensor([[1, 2, 3, 4, 5, 6],[7, 8, 9, 10, 11, 12],[13, 14, 15, 16, 17, 18]])

# 將張量沿著列方向（維度0）拆分成兩個子張量
sub_tensors = torch.split(tensor,split_size_or_sections=2, dim=0)

#
for sub_tensor in sub_tensors:
    print(sub_tensor)
```

```
tensor([[ 1,  2,  3,  4,  5,  6],
        [ 7,  8,  9, 10, 11, 12]])
tensor([[13, 14, 15, 16, 17, 18]])
```

torch.gather()

```python
import torch

# 創建一個形狀為(2, 3)的輸入張量
input_tensor = torch.tensor([[1, 2, 3],[4, 5, 6]])

# 指定要收集的索引
index = torch.tensor([[0, 2],[1, 0]])

#
output_tensor = torch.gather(input_tensor, dim=1, index=index)

#
print(output_tensor)
```

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