In [ ]:
```python
# imports
from machine_learning_functions import *
```

In [ ]:
```python
(X := np.array([0.5, 1.0, 10.0, -5.0, -0.1, 0.0, -1.0, -0.3, 0.6, 0.4]))
```

Out[ ]:
```
array([ 0.5,  1. , 10. , -5. , -0.1,  0. , -1. , -0.3,  0.6,  0.4])
```

In [ ]:
```python
sigmoid = Sigmoid()
```

In [ ]:
```python
sigmoid(X)
```

Out[ ]:
```
array([ 0.04843825,  0.03301454,  0.00490444, -0.01170297,  0.11024104,
        0.09090909, -0.12063188,  0.19182462,  0.04429912,  0.05343058])
```

In [ ]:
```python
sigmoid.dX(X)
```

Out[ ]:
```
array([[ 0.04609199,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.03192458,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.00488039,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        , -0.01183993,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.09808796,
         0.        ,  0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.08264463,  0.        ,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        , -0.13518393,  0.        ,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.15502793,  0.        ,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.04233671,  0.        ],
       [ 0.        ,  0.        ,  0.        ,  0.        ,  0.        ,
         0.        ,  0.        ,  0.        ,  0.        ,  0.05057576]])
```

In [ ]:
```python
sigmoid.dX(X).shape
```

Out[ ]:
```
(10, 10)
```

In [ ]:
```python
# activation = RELU()
activation = Leaky_RELU(0.2)
```

In [ ]:
```python
activation(X)
```

Out[ ]:
```
array([ 0.5 ,  1.  , 10.  , -1.  , -0.02,  0.  , -0.2 , -0.06,  0.6 ,
        0.4 ])
```

In [ ]:
```python
activation.vectorised_derivative(X)
```

Out[ ]:
```
array([1. , 1. , 1. , 0.2, 0.2, 1. , 0.2, 0.2, 1. , 1. ])
```

In [ ]:
```python
activation.dX(X)
```

Out[ ]:
```
array([[1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 1. , 0. , 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0.2, 0. , 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0.2, 0. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 1. , 0. , 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0.2, 0. , 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0.2, 0. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. , 0. ],
       [0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 0. , 1. ]])
```

Test the layer class

In [ ]:
```
layer = Layer_Transformation(4, 2, RELU())
```

In [ ]:
```
layer.weights
```

Out[ ]:
```
array([[-0.09064785, -0.79155874],
       [-0.44657293, -0.2488978 ],
       [-0.51476828,  0.60746217],
       [ 0.5473846 , -0.99528016]])
```

In [ ]:
```
layer.bias
```

Out[ ]:
```
array([0.5, 0.5, 0.5, 0.5])
```

In [ ]:
```
layer.get_activations(
    np.array([0,0])
)
```

Out[ ]:
```
array([0.5, 0.5, 0.5, 0.5])
```

In [ ]:
```
layer.get_activations(
    np.array([1,1])
)
```

Out[ ]:
```
array([-0.        , -0.        ,  0.59269389,  0.05210445])
```

In [ ]:
```
layer.get_activations(
    np.array([0.34,0.81])
)
```

Out[ ]:
```
array([-0.        ,  0.14655799,  0.81702314, -0.        ])
```

In [ ]:
```
layer.weighted_sums
```

Out[ ]:
```
array([-0.17198285,  0.14655799,  0.81702314, -0.12006616])
```

In [ ]:
```
layer.activations
```

Out[ ]:
```
array([-0.        ,  0.14655799,  0.81702314, -0.        ])
```

In [ ]:
```
(
    layer.dAdZ(),
    layer.dZdW(),
    layer.dZdAp()
)
```

Out[ ]:
```
(array([[0., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 0.]]),
 array([0.34, 0.81]),
 array([[-0.09064785, -0.79155874],
        [-0.44657293, -0.2488978 ],
        [-0.51476828,  0.60746217],
        [ 0.5473846 , -0.99528016]]))
```

In [ ]:
```
cost_func = MSE()
cost_func(
    layer.activations,
    np.array([0, 0, 1, 0])
)
```

Out[ ]:
```
0.013739943763856552
```

In [ ]:
```
(
    (dAdZ := layer.dAdZ()),
    (dcdA := cost_func.dP()),
    (dcdZ := dAdZ @ dcdA)
)
```

Out[ ]:
```
(array([[0., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [0., 0., 0., 0.]]),
 array([-0.        ,  0.07327899, -0.09148843, -0.        ]),
 array([ 0.        ,  0.07327899, -0.09148843,  0.        ]))
```

In [ ]:
```
# here I will use outer product rather than transverses
(
    dcdZ,
    (dZdW := layer.dZdW()),
    (dcdW := np.outer(dcdZ, dZdW))
)
```

Out[ ]:
```
(array([ 0.        ,  0.07327899, -0.09148843,  0.        ]),
 array([0.34, 0.81]),
 array([[ 0.        ,  0.        ],
        [ 0.02491486,  0.05935599],
        [-0.03110607, -0.07410563],
        [ 0.        ,  0.        ]]))
```

In [ ]:
```
# (
#     (dZdB := layer.dZdB()),
#     dcdZ,
#     (dcdB := dZdB @ dcdZ)
# )

(
    dcdZ,
    (dcdB := dcdZ)
)
```

Out[ ]:
```
(array([ 0.        ,  0.07327899, -0.09148843,  0.        ]),
 array([ 0.        ,  0.07327899, -0.09148843,  0.        ]))
```

In [ ]:
```
(
    (dZdAp := layer.dZdAp()),
    dcdZ,
    (dcdAp := dZdAp.T @ dcdZ)
)
```

Out[ ]:
```
(array([[-0.09064785, -0.79155874],
        [-0.44657293, -0.2488978 ],
        [-0.51476828,  0.60746217],
        [ 0.5473846 , -0.99528016]]),
 array([ 0.        ,  0.07327899, -0.09148843,  0.        ]),
 array([ 0.01437093, -0.07381474]))
```

In [ ]:
```
model = FFN(
    [2,4,2],
    [RELU() for _ in range(2)],
    MSE()
)
```

In [ ]:
```
(
    model.tranformation_layers[0].weights,
    model.tranformation_layers[0].bias,
    model.tranformation_layers[1].weights,
    model.tranformation_layers[1].bias
)
```

Out[ ]:
```
(array([[-0.55816015,  0.2858176 ],
        [-0.21456866, -0.57383174],
        [-0.43776862, -0.84339484],
        [ 0.80917179,  0.88996469]]),
 array([0.5, 0.5, 0.5, 0.5]),
 array([[-0.35003769,  0.9576626 , -0.8890226 ,  0.16348804],
        [-0.20132945,  0.35412975,  0.50686099,  0.80956816]]),
 array([0.5, 0.5]))
```

In [ ]:
```
model.foreward_propagate(
    np.array([0, 0]),
    np.array([1, 1])
)
```

Out[ ]:
```
(array([0.44104518, 1.23461473]), 0.18373728297811864)
```

In [ ]:
```
model.back_propogate(
    np.array([0, 0]),
    np.array([1, 1])
)
```

Out[ ]:
```
{'W2': array([[-0.27947741, -0.27947741, -0.27947741, -0.27947741],
        [ 0.11730736,  0.11730736,  0.11730736,  0.11730736]]),
 'B2': array([-0.55895482,  0.23461473]),
 'W1': array([[ 0.,  0.],
        [-0., -0.],
        [ 0.,  0.],
        [ 0.,  0.]]),
 'B1': array([ 0.1484204 , -0.45220608,  0.61584053,  0.09855418])}
```

In [ ]:
```
from random import uniform
```

In [ ]:
```
# # create random function to learn
# degree = 1
# coefficients = [uniform(-1, 1) for _ in range(degree+1)]

# def polynomial_function(x):
#     return sum(coefficients[i] * x**i for i in range(degree+1))

def linear_function(x): return 2*x+5
```

In [ ]:
```
num_data_items = 1000
X_data = [uniform(-100, 100) for _ in range(num_data_items)]
Y_data = [linear_function(X_data[i]) for i in range(num_data_items)]

X_data, Y_data = np.array(X_data), np.array(Y_data)
```

In [ ]:
```
function_emulator_model = FFN(
    [1, 1],
    [Linear_Activation_Funcion()],
    MSE()
)
```

In [ ]:
```
(
    function_emulator_model.tranformation_layers[0].weights,
    function_emulator_model.tranformation_layers[0].bias,
)
```

Out[ ]:
```
(array([[0.51662746]]), array([0.5]))
```

In [ ]:
```
# function_emulator_model.transformation_layers[0].weights = np.array([[2.0]])
# function_emulator_model.transformation_layers[0].bias = np.array([5.0])
```

In [ ]:
```
# function_emulator_model.transformation_layers[0].weights = np.array([[1.2]])
# function_emulator_model.transformation_layers[0].bias = np.array([3.0])
```

In [ ]:
```
function_emulator_model.foreward_propagate(2, 2*2+5)
```

Out[ ]:
```
(array([1.53325493]), 55.75228194202673)
```

In [ ]:
```
function_emulator_model.back_propogate(2, 2*2+5)
```

Out[ ]:
```
{'W1': array([[-29.86698028]]), 'B1': array([-14.93349014])}
```

In [ ]:
```
function_emulator_model.foreward_propagate(-3, (-3)*2+5)
```

Out[ ]:
```
(array([-1.04988239]), 0.0024882533143272354)
```

In [ ]:
```
function_emulator_model.back_propogate(-3, (-3)*2+5)
```

Out[ ]:
```
{'W1': array([[0.29929437]]), 'B1': array([-0.09976479])}
```

In [ ]:
```
training_manager = Model(
    FFN=function_emulator_model,
    data_set=(X_data, Y_data)
)
```

In [ ]:
```
training_manager.train_and_evaluate(
    learning_rate=0.0001,
    epochs=10,
    batch_size=50
)
```

Out[ ]:
```
14.35824670325447
```

In [ ]:
```
(
    function_emulator_model.tranformation_layers[0].weights,
    function_emulator_model.tranformation_layers[0].bias,
)
```

Out[ ]:
```
(array([[1.99942258]]), array([1.21192484]))
```

In [ ]:
```
for _ in range(10):
    x = uniform(-1, 1)
    y = linear_function(x)
    p, c = training_manager.FFN.foreward_propagate(
        np.array([x,]), np.array([y,])
    )
    print(f"For the input {x:.4f} the network predicted {p[0]:.4f} and actual va
```

```
For the input 0.8926 the network predicted 2.9967 and actual value was 6.7853 giv
ing a cost of 14.3534
For the input -0.8759 the network predicted -0.5393 and actual value was 3.2482 g
iving a cost of 14.3457
For the input -0.0286 the network predicted 1.1548 and actual value was 4.9428 gi
ving a cost of 14.3494
For the input -0.8426 the network predicted -0.4728 and actual value was 3.3147 g
iving a cost of 14.3458
For the input 0.8849 the network predicted 2.9812 and actual value was 6.7698 giv
ing a cost of 14.3534
For the input -0.6873 the network predicted -0.1622 and actual value was 3.6255 g
iving a cost of 14.3465
For the input 0.1227 the network predicted 1.4573 and actual value was 5.2455 giv
ing a cost of 14.3501
For the input 0.0584 the network predicted 1.3287 and actual value was 5.1168 giv
ing a cost of 14.3498
For the input 0.0623 the network predicted 1.3364 and actual value was 5.1245 giv
ing a cost of 14.3498
For the input -0.9207 the network predicted -0.6290 and actual value was 3.1585 g
iving a cost of 14.3455
```

In [ ]:
```
# create random function to learn
degree = 1
coefficients = [uniform(-1, 1) for _ in range(degree+1)]

def polynomial_function(x):
    return sum(coefficients[i] * x**i for i in range(degree+1))
```

In [ ]:
```
coefficients
```

Out[ ]:
```
[0.35074834378407993, -0.6476307797195384]
```

In [ ]:
```
num_data_items = 10000
X_data = [uniform(-100, 100) for _ in range(num_data_items)]
Y_data = [linear_function(X_data[i]) for i in range(num_data_items)]
```

```
X_data, Y_data = np.array(X_data), np.array(Y_data)
```

In [ ]:
```
function_emulator_model = FFN(
    [1, 1, 1],
    [Leaky_RELU(0.01), Linear_Activation_Funcion()],
    MSE()
)

# function_emulator_model = FFN(
#     [1, 1],
#     [Linear_Activation_Funcion()],
#     MSE()
# )
```

In [ ]:
```
(
    function_emulator_model.tranformation_layers[0].weights,
    function_emulator_model.tranformation_layers[0].bias,
)
```

Out[ ]:
```
(array([[-0.33452972]]), array([0.5]))
```

In [ ]:
```
training_manager = Model(
    FFN=function_emulator_model,
    data_set=(X_data, Y_data)
)
```

In [ ]:
```
training_manager.train_and_evaluate(
    learning_rate=0.0001,
    epochs=10,
    batch_size=50
)
```

Out[ ]:
```
2849.3752604979236
```

In [ ]:
```
(
    function_emulator_model.tranformation_layers[0].weights,
    function_emulator_model.tranformation_layers[0].bias,
    function_emulator_model.tranformation_layers[1].weights,
    function_emulator_model.tranformation_layers[1].bias,
)
```

Out[ ]:
```
(array([[1.29304125]]),
 array([11.45175091]),
 array([[2.14321746]]),
 array([-51.47822236]))
```

In [ ]:
```
for _ in range(10):
    x = uniform(-1, 1)
    y = linear_function(x)
    p, c = training_manager.FFN.foreward_propagate(
        np.array([x,]), np.array([y,])
    )
    print(f"For the input {x:.4f} the network predicted {p[0]:.4f} and actual va
```

```
For the input -0.8135 the network predicted -29.1891 and actual value was 3.3730
giving a cost of 1060.2882
For the input -0.7973 the network predicted -29.1441 and actual value was 3.4054
giving a cost of 1059.4733
For the input -0.1669 the network predicted -27.3971 and actual value was 4.6663
giving a cost of 1028.0572
For the input 0.2831 the network predicted -26.1502 and actual value was 5.5661 g
iving a cost of 1005.9251
For the input 0.7605 the network predicted -24.8272 and actual value was 6.5209 g
iving a cost of 982.7036
For the input -0.5839 the network predicted -28.5528 and actual value was 3.8322
giving a cost of 1048.7863
For the input -0.4445 the network predicted -28.1665 and actual value was 4.1110
giving a cost of 1041.8343
For the input -0.1829 the network predicted -27.4416 and actual value was 4.6341
giving a cost of 1028.8516
For the input 0.9026 the network predicted -24.4332 and actual value was 6.8053 g
iving a cost of 975.8413
For the input 0.2418 the network predicted -26.2646 and actual value was 5.4835 g
iving a cost of 1007.9462
```

In [ ]: