Assignment4

February 25, 2024

1 Assignment 4

1.1 Group Members:

1.1.1 Nils Dunlop, e-mail: gusdunlni@student.gu.se

1.1.2 Francisco Alejandro Erazo Piza, e-mail: guserafr@student.gu.se

1.1.3 Chukwudumebi Ubogu, e-mail: gusuboch@student.gu.se

1.1.4 Task 1: A small linear regression example in PyTorch

---

[137]:
```
# Import necessary libraries
import pandas as pd
import numpy as np
import torch
from torch import optim

# Read data
data = pd.read_csv('a4_synthetic.csv')

# Convert input and output to numpy arrays
X = data.drop(columns='y').to_numpy()
Y = data.y.to_numpy()

(X, Y)
```

[137]:
```
(array([[ 1.76405235,  0.40015721],
        [ 0.97873798,  2.2408932 ],
        [ 1.86755799, -0.97727788],
        [ 0.95008842, -0.15135721],
        [-0.10321885,  0.4105985 ],
        [ 0.14404357,  1.45427351],
        [ 0.76103773,  0.12167502],
        [ 0.44386323,  0.33367433],
        [ 1.49407907, -0.20515826],
```

```
[ 0.3130677 , -0.85409574],
[-2.55298982,  0.6536186 ],
[ 0.8644362 , -0.74216502],
[ 2.26975462, -1.45436567],
[ 0.04575852, -0.18718385],
[ 1.53277921,  1.46935877],
[ 0.15494743,  0.37816252],
[-0.88778575, -1.98079647],
[-0.34791215,  0.15634897],
[ 1.23029068,  1.20237985],
[-0.38732682, -0.30230275],
[-1.04855297, -1.42001794],
[-1.70627019,  1.9507754 ],
[-0.50965218, -0.4380743 ],
[-1.25279536,  0.77749036],
[-1.61389785, -0.21274028],
[-0.89546656,  0.3869025 ],
[-0.51080514, -1.18063218],
[-0.02818223,  0.42833187],
[ 0.06651722,  0.3024719 ],
[-0.63432209, -0.36274117],
[-0.67246045, -0.35955316],
[-0.81314628, -1.7262826 ],
[ 0.17742614, -0.40178094],
[-1.63019835,  0.46278226],
[-0.90729836,  0.0519454 ],
[ 0.72909056,  0.12898291],
[ 1.13940068, -1.23482582],
[ 0.40234164, -0.68481009],
[-0.87079715, -0.57884966],
[-0.31155253,  0.05616534],
[-1.16514984,  0.90082649],
[ 0.46566244, -1.53624369],
[ 1.48825219,  1.89588918],
[ 1.17877957, -0.17992484],
[-1.07075262,  1.05445173],
[-0.40317695,  1.22244507],
[ 0.20827498,  0.97663904],
[ 0.3563664 ,  0.70657317],
[ 0.01050002,  1.78587049],
[ 0.12691209,  0.40198936],
[ 1.8831507 , -1.34775906],
[-1.270485  ,  0.96939671],
[-1.17312341,  1.94362119],
[-0.41361898, -0.74745481],
[ 1.92294203,  1.48051479],
[ 1.86755896,  0.90604466],
```

```
       [-0.86122569,  1.91006495],
       [-0.26800337,  0.8024564 ],
       [ 0.94725197, -0.15501009],
       [ 0.61407937,  0.92220667],
       [ 0.37642553, -1.09940079],
       [ 0.29823817,  1.3263859 ],
       [-0.69456786, -0.14963454],
       [-0.43515355,  1.84926373],
       [ 0.67229476,  0.40746184],
       [-0.76991607,  0.53924919],
       [-0.67433266,  0.03183056],
       [-0.63584608,  0.67643329],
       [ 0.57659082, -0.20829876],
       [ 0.39600671, -1.09306151],
       [-1.49125759,  0.4393917 ],
       [ 0.1666735 ,  0.63503144],
       [ 2.38314477,  0.94447949],
       [-0.91282223,  1.11701629],
       [-1.31590741, -0.4615846 ],
       [-0.06824161,  1.71334272],
       [-0.74475482, -0.82643854],
       [-0.09845252, -0.66347829],
       [ 1.12663592, -1.07993151],
       [-1.14746865, -0.43782004],
       [-0.49803245,  1.92953205],
       [ 0.94942081,  0.08755124],
       [-1.22543552,  0.84436298],
       [-1.00021535, -1.5447711 ],
       [ 1.18802979,  0.31694261],
       [ 0.92085882,  0.31872765],
       [ 0.85683061, -0.65102559],
       [-1.03424284,  0.68159452],
       [-0.80340966, -0.68954978],
       [-0.4555325 ,  0.01747916],
       [-0.35399391, -1.37495129],
       [-0.6436184 , -2.22340315],
       [ 0.62523145, -1.60205766],
       [-1.10438334,  0.05216508],
       [-0.739563  ,  1.5430146 ],
       [-1.29285691,  0.26705087],
       [-0.03928282, -1.1680935 ],
       [ 0.52327666, -0.17154633],
       [ 0.77179055,  0.82350415],
       [ 2.16323595,  1.33652795]]),
array([ 0.53895342,  0.21594929,  0.85495138,  0.5636907 ,  0.32130011,
        0.02282172,  0.43355955,  0.28494948,  0.62455331,  0.457357  ,
       -0.09488842,  0.58031413,  0.94447131,  0.30172726,  0.32449838,
```

```
 0.26373822,  0.38526199,  0.28157172,  0.62281853,  0.28318203,
 0.20235347, -0.2192494 ,  0.2351877 ,  0.069649  , -0.0850409 ,
 0.13153761,  0.37868332,  0.29131062,  0.23830441,  0.2282746 ,
 0.10282648,  0.32232423,  0.3400722 ,  0.03356524,  0.04442704,
 0.50787709,  0.79897741,  0.25672869,  0.27826382,  0.32854869,
-0.04091009,  0.52842224,  0.35116701,  0.51040517, -0.00891893,
-0.05060942,  0.36320961,  0.4108211 ,  0.04400282,  0.15071379,
 0.8404956 , -0.06019488, -0.07903399,  0.29997196,  0.55319739,
 0.60729503, -0.10907525,  0.04783283,  0.33979011,  0.3839887 ,
 0.3860405 ,  0.17411641,  0.15875186,  0.03734816,  0.19584133,
 0.14867472,  0.25425119,  0.14634071,  0.40984468,  0.51729792,
 0.05805776,  0.00122176,  0.8168331 ,  0.08095776,  0.08182919,
 0.08083589,  0.3336741 ,  0.36463529,  0.42725213,  0.37953982,
 0.01203189,  0.57378817, -0.05040895,  0.47352592,  0.51573575,
 0.50237034,  0.43274868,  0.19031785,  0.32825265,  0.37082139,
 0.34825692,  0.40705651,  0.83290736,  0.02051509,  0.01094772,
 0.18452787,  0.4515847 ,  0.4861225 ,  0.32067403,  0.57773763]))
```

[138]:
```python
# Random seed for reproducibility
np.random.seed(1)
torch.manual_seed(1)

# Initialize parameters
w_init = np.random.normal(size=(2, 1))
b_init = np.random.normal(size=(1, 1))

# Declare the parameter tensors
w = torch.tensor(w_init, dtype=torch.float32, requires_grad=True)
b = torch.tensor(b_init, dtype=torch.float32, requires_grad=True)

# Initialize the optimizer
eta = 1e-2
opt = optim.SGD([w, b], lr=eta)

for i in range(10):
    # Reset sum_err at the beginning of each epoch
    sum_err = 0

    for row in range(X.shape[0]):
        # Fix the input and output
        x = torch.tensor(X[row, :], dtype=torch.float32).view(1, -1)
        y = torch.tensor(Y[row], dtype=torch.float32).view(1, -1)

        # Forward pass.
        y_pred = x.mm(w) + b
        err = (y_pred - y).pow(2).sum()
```

```
        # Backward and update.
        opt.zero_grad()
        err.backward()
        opt.step()

        # For statistics.
        sum_err += err.item()

    mse = sum_err / X.shape[0]
    print(f'Epoch {i+1}: MSE =', mse)
```

```
Epoch 1: MSE = 0.7999662647869263
Epoch 2: MSE = 0.017392394159767264
Epoch 3: MSE = 0.009377418162580966
Epoch 4: MSE = 0.009355327616258364
Epoch 5: MSE = 0.009365440349979508
Epoch 6: MSE = 0.009366988411857164
Epoch 7: MSE = 0.009367207068114567
Epoch 8: MSE = 0.009367238481529512
Epoch 9: MSE = 0.009367244712136654
Epoch 10: MSE = 0.009367244620257224
```

Task 1 Discussion

Our model demonstrated substantial improvement over the training period, with the MSE reducing from an initial value of 0.7999 in epoch 1 to approximately 0.0094 by epoch 10. This progression indicates successful parameter optimization, aligning with the expected outcome of a correctly implemented linear regression model in PyTorch.

1.1.5 Task 2: Implementing tensor arithmetics for forward computations

---

We decided to centralize tasks 2-4 in the next cell because we faced several challenges when trying to replicate and adjust the code across multiple cells for each task. Initially, we copied the code from previous steps to incorporate these new requirements. But this strategy made it harder to keep track of changes and increased the likelihood of mistakes.

[139]:
```
class Tensor:

    # Constructor. Just store the input values.
    def __init__(self, data, requires_grad=False, grad_fn=None):
        self.data = data
        self.shape = data.shape
        self.grad_fn = grad_fn
        self.requires_grad = requires_grad
        self.grad = None
```

```
    # So that we can print the object or show it in a notebook cell.
    def __repr__(self):
        dstr = repr(self.data)
        if self.requires_grad:
            gstr = ', requires_grad=True'
        elif self.grad_fn is not None:
            gstr = f', grad_fn={self.grad_fn}'
        else:
            gstr = ''
        return f'Tensor({dstr}{gstr})'

    # Extract one numerical value from this tensor.
    def item(self):
        return self.data.item()

    # Operator +
    def __add__(self, right):
        if not isinstance(right, Tensor):
            right = tensor(right)
        return addition(self, right)

    # Operator -
    def __sub__(self, right):
        if not isinstance(right, Tensor):
            right = tensor(right)
        return subtraction(self, right)

    # Operator @
    def __matmul__(self, right):
        if not isinstance(right, Tensor):
            raise ValueError("Right operand must be a Tensor")
        return matrix_multi(self, right)

    # Operator **
    def __pow__(self, right):
        # NOTE! We are assuming that right is an integer here, not a Tensor!
        if not isinstance(right, int):
            raise Exception('only integers allowed')
        if right < 2:
            raise Exception('power must be >= 2')
        return power(self, right)

    # Operator .sum()
    def sum(self):
        sum_data = np.sum(self.data)
        grad_fn = SummationNode(self) if self.requires_grad else None
```

```
        return Tensor(sum_data, requires_grad=self.requires_grad,␣
↪grad_fn=grad_fn)

    # Backward computations. Implemented in Task 4.
    def backward(self, grad_output=None):
        # Skip gradient computation if not required
        if not self.requires_grad:
            return

        # Use provided gradient or default to ones if none is provided
        grad_output = np.ones_like(self.data) if grad_output is None else␣
↪grad_output

        # Initialize or accumulate gradient
        self.grad = grad_output if self.grad is None else self.grad +␣
↪grad_output

        # Propagate gradient to previous operation if it exists
        if self.grad_fn:
            self.grad_fn.backward(grad_output)

# A small utility where we simply create a Tensor object
def tensor(data, requires_grad=False):
    return Tensor(data, requires_grad)

# We define helper functions to implement the various arithmetic operations.
def addition(left, right):
    # Perform element-wise addition
    new_data = left.data + right.data

    # Determine if a gradient function is needed
    if left.requires_grad or right.requires_grad:
        grad_fn = AdditionNode(left, right)
    else:
        grad_fn = None

    # Return a new Tensor object
    return Tensor(new_data, requires_grad=left.requires_grad or right.
↪requires_grad, grad_fn=grad_fn)

def subtraction(left, right):
    # Perform element-wise subtraction
    new_data = left.data - right.data

    # Determine if a gradient function is needed
    if left.requires_grad or right.requires_grad:
        grad_fn = SubtractionNode(left, right)
```

```
    else:
        grad_fn = None

    # Return a new Tensor object
    return Tensor(new_data, requires_grad=left.requires_grad or right.
 ↪requires_grad, grad_fn=grad_fn)

def matrix_multi(left, right):
    # Check for shape compatibility for matrix multiplication
    if left.shape[1] != right.shape[0]:
        raise ValueError("Shapes are not compatible for matrix multiplication")

    # Perform matrix multiplication
    new_data = np.dot(left.data, right.data)

    # Determine if a gradient function is needed
    if left.requires_grad or right.requires_grad:
        grad_fn = MatMulNode(left, right)
    else:
        grad_fn = None

    # Return a new Tensor object
    return Tensor(new_data, requires_grad=left.requires_grad or right.
 ↪requires_grad, grad_fn=grad_fn)

def power(tensor, exponent):
    # Perform power operation
    new_data = tensor.data ** exponent

    # Determine if a gradient function is needed
    if tensor.requires_grad:
        grad_fn = PowerNode(tensor, exponent)
    else:
        grad_fn = None

    # Return a new Tensor object
    return Tensor(new_data, requires_grad=tensor.requires_grad, grad_fn=grad_fn)

class Node:
    def __init__(self):
        pass

    def backward(self, grad_output):
        raise NotImplementedError('Backward method not implemented.')

    def __repr__(self):
        return str(type(self))
```

```
class AdditionNode(Node):
    # Represents an addition operation in the graph.
    def __init__(self, left, right):
        super().__init__()
        self.left = left
        self.right = right

    def backward(self, grad_output):
        # Propagate gradients back to operands of the addition.
        if self.left.requires_grad:
            self.left.grad = grad_output if self.left.grad is None else self.
 ↪left.grad + grad_output
            if self.left.grad_fn:
                self.left.grad_fn.backward(grad_output)

        if self.right.requires_grad:
            self.right.grad = grad_output if self.right.grad is None else self.
 ↪right.grad + grad_output
            if self.right.grad_fn:
                self.right.grad_fn.backward(grad_output)

class SubtractionNode(Node):
    # Represents a subtraction operation in the graph.
    def __init__(self, left, right):
        super().__init__()
        self.left = left
        self.right = right

    def backward(self, grad_output):
        # Propagate gradients back to operands, adjusting for the operation's␣
 ↪nature.
        if self.left.requires_grad:
            self.left.grad = grad_output if self.left.grad is None else self.
 ↪left.grad + grad_output
            if self.left.grad_fn:
                self.left.grad_fn.backward(grad_output)

        if self.right.requires_grad:
            # Gradient for subtraction is negated for the right operand.
            neg_grad_output = -grad_output
            self.right.grad = neg_grad_output if self.right.grad is None else␣
 ↪self.right.grad + neg_grad_output
            if self.right.grad_fn:
                self.right.grad_fn.backward(neg_grad_output)

class MatMulNode(Node):
```

```
    # Represents a matrix multiplication operation in the graph.
    def __init__(self, left, right):
        super().__init__()
        self.left = left
        self.right = right

    def backward(self, grad_output):
        # Propagate gradients according to matrix multiplication rules.
        if self.left.requires_grad:
            # Gradient w.r.t. the left operand is computed by dotting␣
↪grad_output with right's transpose.
            grad_left = np.dot(grad_output, self.right.data.T)
            self.left.grad = grad_left if self.left.grad is None else self.left.
↪grad + grad_left
            if self.left.grad_fn:
                self.left.grad_fn.backward(grad_left)

        if self.right.requires_grad:
            # Gradient w.r.t. the right operand involves left's transpose and␣
↪grad_output.
            grad_right = np.dot(self.left.data.T, grad_output)
            self.right.grad = grad_right if self.right.grad is None else self.
↪right.grad + grad_right
            if self.right.grad_fn:
                self.right.grad_fn.backward(grad_right)

class PowerNode(Node):
    # Represents a power operation in the graph.
    def __init__(self, tensor, exponent):
        super().__init__()
        self.tensor = tensor
        self.exponent = exponent

    def backward(self, grad_output):
        # Propagate gradient based on the power rule.
        grad_tensor = self.exponent * (self.tensor.data ** (self.exponent - 1))
        self.tensor.grad = grad_output * grad_tensor if self.tensor.grad is␣
↪None else self.tensor.grad + (grad_output * grad_tensor)
        if self.tensor.grad_fn:
            self.tensor.grad_fn.backward(grad_output * grad_tensor)

class SummationNode(Node):
    # Represents a summation operation over all elements in the tensor.
    def __init__(self, tensor):
        super().__init__()
        self.tensor = tensor
```

```
    def backward(self, grad_output):
        # Propagate gradients uniformly to each element of the summed tensor.
        if self.tensor.requires_grad:
            self.tensor.grad = np.ones_like(self.tensor.data) * grad_output if␣
 ↪self.tensor.grad is None else self.tensor.grad + (np.ones_like(self.tensor.
 ↪data) * grad_output)
            if self.tensor.grad_fn:
                self.tensor.grad_fn.backward(grad_output)
```

Sanity Check for Task 2

[140]:
```
# Two tensors holding row vectors.
x1 = tensor(np.array([[2.0, 3.0]]))
x2 = tensor(np.array([[1.0, 4.0]]))
# A tensors holding a column vector.
w = tensor(np.array([[-1.0], [1.2]]))

# Test the arithmetic operations.
test_plus = x1 + x2
test_minus = x1 - x2
test_power = x2 ** 2
test_matmul = x1 @ w

print(f'Test of addition: {x1.data} + {x2.data} = {test_plus.data}')
print(f'Test of subtraction: {x1.data} - {x2.data} = {test_minus.data}')
print(f'Test of power: {x2.data} ** 2 = {test_power.data}')
print(f'Test of matrix multiplication: {x1.data} @ {w.data} = {test_matmul.
 ↪data}')

# Check that the results are as expected. Will crash if there is a␣
 ↪miscalculation.
assert(np.allclose(test_plus.data, np.array([[3.0, 7.0]])))
assert(np.allclose(test_minus.data, np.array([[1.0, -1.0]])))
assert(np.allclose(test_power.data, np.array([[1.0, 16.0]])))
assert(np.allclose(test_matmul.data, np.array([[1.6]])))
```

```
Test of addition: [[2. 3.]] + [[1. 4.]] = [[3. 7.]]
Test of subtraction: [[2. 3.]] - [[1. 4.]] = [[ 1. -1.]]
Test of power: [[1. 4.]] ** 2 = [[ 1. 16.]]
Test of matrix multiplication: [[2. 3.]] @ [[-1. ]
 [ 1.2]] = [[1.6]]
```

1.1.6 Task 3: Building the computational graph

Sanity Check for Task 3

[141]:
```
x = tensor(np.array([[2.0, 3.0]]))
w1 = tensor(np.array([[1.0, 4.0]]), requires_grad=True)
w2 = tensor(np.array([[3.0, -1.0]]), requires_grad=True)

test_graph = x + w1 + w2

print('Computational graph top node after x + w1 + w2:', test_graph.grad_fn)

assert(isinstance(test_graph.grad_fn, AdditionNode))
assert(test_graph.grad_fn.right is w2)
assert(test_graph.grad_fn.left.grad_fn.left is x)
assert(test_graph.grad_fn.left.grad_fn.right is w1)
```

```
Computational graph top node after x + w1 + w2: <class '__main__.AdditionNode'>
```

1.1.7 Task 4: Implementing the backward computations

Sanity Check for Task 4

[142]:
```
x = tensor(np.array([[2.0, 3.0]]))
w = tensor(np.array([[-1.0], [1.2]]), requires_grad=True)
y = tensor(np.array([[0.2]]))

# We could as well write simply loss = (x @ w - y)**2
# We break it down into steps here if you need to debug.

model_out = x @ w
diff = model_out - y
loss = diff ** 2

loss.backward()

print('Gradient of loss w.r.t. w =\n', w.grad)

assert(np.allclose(w.grad, np.array([[5.6], [8.4]])))
assert(x.grad is None)
assert(y.grad is None)
```

```
Gradient of loss w.r.t. w =
 [[5.6]
 [8.4]]
```

[143]:
```
# Pytorch Task 4 Sanity check
pt_x = torch.tensor(np.array([[2.0, 3.0]]))
pt_w = torch.tensor(np.array([[-1.0], [1.2]]), requires_grad=True)
pt_y = torch.tensor(np.array([[0.2]]))
```

```
pt_model_out = pt_x @ pt_w
pt_model_out.retain_grad() # Keep the gradient of intermediate nodes for␣
 ↪debugging.

pt_diff = pt_model_out - pt_y
pt_diff.retain_grad()

pt_loss = pt_diff ** 2
pt_loss.retain_grad()

pt_loss.backward()
pt_w.grad
```

[143]:
```
tensor([[5.6000],
        [8.4000]], dtype=torch.float64)
```

1.1.8 Task 5: Optimizers to update the model parameters

---

[144]:
```
class Optimizer:
    def __init__(self, params):
        self.params = params

    def zero_grad(self):
        for p in self.params:
            p.grad = np.zeros_like(p.data)

    def step(self):
        raise NotImplementedError('Step method not implemented.')

class SGD(Optimizer):
    def __init__(self, params, lr):
        super().__init__(params)
        self.lr = lr

    def step(self):
        # Update the parameters using the gradients
        for p in self.params:
            p.data -= self.lr * p.grad
```

Training Loop

[145]:
```
# Class to hold the parameters and gradients
class Parameter:
    def __init__(self, data):
        self.data = data
        self.grad = np.zeros_like(data)
```

[146]:
```
# Random seed for reproducibility
np.random.seed(1)

# Initialize parameters
w_init = np.random.normal(size=(2, 1))
b_init = np.random.normal(size=(1, 1))

# Create parameter objects
w = Parameter(w_init)
b = Parameter(b_init)

# Initialize the optimizer
params = [w, b]
eta = 1e-2
opt = SGD(params, lr=eta)

# Training loop
for i in range(10):
    sum_err = 0
    for row in range(X.shape[0]):
        # Reset gradients at the beginning of each step
        opt.zero_grad()

        # Fix the input and output
        x = X[row, :].reshape(1, -1)
        y = Y[row].reshape(1, -1)

        # Forward pass
        y_pred = x @ w.data + b.data
        err = ((y_pred - y) ** 2).sum()

        # Compute gradients
        err_grad = 2 * (y_pred - y)
        w.grad = x.T @ err_grad
        b.grad = np.sum(err_grad, axis=0, keepdims=True)

        # Update parameters
        opt.step()

        # Accumulate error
        sum_err += err

    mse = sum_err / X.shape[0]
    print(f'Epoch {i+1}: MSE =', mse)
```

```
Epoch 1: MSE = 0.7999661130823179
Epoch 2: MSE = 0.017392390107906875
Epoch 3: MSE = 0.009377418010839892
```

```
Epoch 4: MSE = 0.009355326971438458
Epoch 5: MSE = 0.009365440968904258
Epoch 6: MSE = 0.009366989180952535
Epoch 7: MSE = 0.009367207398577987
Epoch 8: MSE = 0.00936723898397449
Epoch 9: MSE = 0.009367243704122534
Epoch 10: MSE = 0.009367244427185761
```

Summary Table MSE Task 1 and Task 5

Epoch	MSE Task 1	MSE Task 5
1	0.7999662647869263	0.7999661130823179
2	0.017392394159767264	0.017392390107906875
3	0.009377418162580966	0.009377418010839892
4	0.009355327616258364	0.009355326971438458
5	0.009365440349979508	0.009365440968904258
6	0.009366988411857164	0.009366989180952535
7	0.009367207068114567	0.009367207398577987
8	0.009367238481529512	0.00936723898397449
9	0.009367244712136654	0.009367243704122534
10	0.009367244620257224	0.009367244427185761

The MSE values are almost identical, indicating that the optimizer is functioning as expected. The optimizer is updating the model parameters in a way that is consistent with the expected outcome of a correctly implemented linear regression model in PyTorch.

1.1.9 Task 6: Classifying raisins

---

[147]:
```python
from sklearn.preprocessing import scale
from sklearn.model_selection import train_test_split

# Load the raisins dataset
a4data = pd.read_csv('raisins.csv')

# Scale the input data
X = scale(a4data.drop(columns='Class'))
Y = 1.0*(a4data.Class == 'Besni').to_numpy()

# Split the data into training and test sets
np.random.seed(0)
shuffle = np.random.permutation(len(Y))
X = X[shuffle]
Y = Y[shuffle]
```

```
Xtrain, Xtest, Ytrain, Ytest = train_test_split(X, Y, random_state=0,␣
 ↪test_size=0.2)
```

[148]:
```
Xtrain.shape, Ytrain.shape, Xtest.shape, Ytest.shape
```

[148]:
```
((720, 7), (720,), (180, 7), (180,))
```

[149]:
```
Ytrain = Ytrain.reshape(-1, 1)
Ytest = Ytest.reshape(-1, 1)
```

[150]:
```
# Activation functions and their derivatives
def tanh(x):
    return np.tanh(x)

def tanh_derivative(x):
    return 1 - np.tanh(x) ** 2

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sigmoid_derivative(x):
    s = sigmoid(x)
    return s * (1 - s)

# Binary Cross-Entropy Loss and its derivative
def binary_cross_entropy(y_true, y_pred):
    # Applying epsilon and clipping to prevent log(0)
    epsilon = 1e-15
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)
    return -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(1 - y_pred))

def binary_cross_entropy_derivative(y_true, y_pred):
    epsilon = 1e-15  # to prevent division by zero
    y_pred = np.clip(y_pred, epsilon, 1 - epsilon)  # Clipping predictions to␣
 ↪avoid division by zero
    return -(y_true / y_pred) + ((1 - y_true) / (1 - y_pred))

# Test functions
print("Tanh test:", tanh(np.array([0, -2])))
print("Tanh derivative test:", tanh_derivative(np.array([0, -2])))
print("Sigmoid test:", sigmoid(np.array([0, -2])))
print("Sigmoid derivative test:", sigmoid_derivative(np.array([0, -2])))
print("Binary Cross-Entropy Loss test:", binary_cross_entropy(np.array([1, 0]),␣
 ↪np.array([0.9, 0.1])))
print("Binary Cross-Entropy Derivative test:",␣
 ↪binary_cross_entropy_derivative(np.array([1, 0]), np.array([0.9, 0.1])))
```

```
Tanh test: [ 0.         -0.96402758]
```

```
Tanh derivative test: [1.         0.07065082]
Sigmoid test: [0.5        0.11920292]
Sigmoid derivative test: [0.25       0.10499359]
Binary Cross-Entropy Loss test: 0.10536051565782628
Binary Cross-Entropy Derivative test: [-1.11111111  1.11111111]
```

[151]:
```python
class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, learning_rate=0.
↪01):
        # Initialize weights and biases for the first (input to hidden) layer
        self.W1 = np.random.randn(input_size, hidden_size)
        self.b1 = np.zeros(hidden_size)

        # Initialize weights and biases for the second (hidden to output) layer
        self.W2 = np.random.randn(hidden_size, output_size)
        self.b2 = np.zeros(output_size)

        # Set the learning rate for the training
        self.learning_rate = learning_rate

    def forward(self, X):
        # Forward pass through the first layer
        self.Z1 = np.dot(X, self.W1) + self.b1  # Linear step
        self.A1 = tanh(self.Z1)  # Activation step

        # Forward pass through the second layer
        self.Z2 = np.dot(self.A1, self.W2) + self.b2  # Linear step
        self.A2 = 1 / (1 + np.exp(-self.Z2))  # Sigmoid activation for binary␣
↪classification output

        return self.A2  # Return the final output

    def compute_loss(self, Y, Y_hat):
        # Compute the binary cross-entropy loss
        return binary_cross_entropy(Y, Y_hat)

    def backward(self, X, Y):
        # Backpropagation to update weights and biases

        # Compute derivative of loss w.r.t. output activation
        dA2 = binary_cross_entropy_derivative(Y, self.A2)

        # Derivative of the loss w.r.t. Z2, then update W2 and b2
        dZ2 = dA2 * (self.A2 * (1 - self.A2))  # Sigmoid derivative
        dW2 = np.dot(self.A1.T, dZ2)
        db2 = np.sum(dZ2, axis=0)
```

```
        # Derivative of the loss w.r.t. Z1, then update W1 and b1
        dA1 = np.dot(dZ2, self.W2.T)
        dZ1 = dA1 * tanh_derivative(self.Z1)  # tanh derivative
        dW1 = np.dot(X.T, dZ1)
        db1 = np.sum(dZ1, axis=0)

        # Update the weights and biases, applying the learning rate
        self.W1 -= self.learning_rate * dW1
        self.b1 -= self.learning_rate * db1
        self.W2 -= self.learning_rate * dW2
        self.b2 -= self.learning_rate * db2

    def train(self, X_train, Y_train, epochs=100):
        # Training loop
        for epoch in range(epochs):
            # Forward pass: compute predicted output
            Y_hat = self.forward(X_train)

            # Compute and print loss and accuracy every 100 epochs
            loss = self.compute_loss(Y_train, Y_hat)
            self.backward(X_train, Y_train)  # Backward pass to update weights

            if epoch % 100 == 0:
                predictions = self.forward(X_train) >= 0.5  # Threshold␣
↪predictions to binary outcomes
                accuracy = np.mean(predictions == Y_train)  # Calculate accuracy
                print(f'Epoch {epoch}, Loss: {loss:.4f}, Accuracy: {accuracy:.
↪4f}')
```

[152]:
```
# Hyperparameters
input_size = Xtrain.shape[1]
hidden_size = 10
output_size = 1  # Binary classification
learning_rate = 0.01
epochs = 10000

nn = NeuralNetwork(input_size, hidden_size, output_size, learning_rate)
nn.train(Xtrain, Ytrain, epochs)
```

```
Epoch 0, Loss: 0.8510, Accuracy: 0.8472
Epoch 100, Loss: 0.3498, Accuracy: 0.8708
Epoch 200, Loss: 0.3376, Accuracy: 0.8764
Epoch 300, Loss: 0.3312, Accuracy: 0.8778
Epoch 400, Loss: 0.3221, Accuracy: 0.8806
Epoch 500, Loss: 0.3157, Accuracy: 0.8847
Epoch 600, Loss: 0.3128, Accuracy: 0.8833
Epoch 700, Loss: 0.3095, Accuracy: 0.8833
Epoch 800, Loss: 0.3066, Accuracy: 0.8847
```

```
Epoch 900, Loss: 0.3048, Accuracy: 0.8833
Epoch 1000, Loss: 0.3015, Accuracy: 0.8792
Epoch 1100, Loss: 0.2969, Accuracy: 0.8806
Epoch 1200, Loss: 0.2921, Accuracy: 0.8806
Epoch 1300, Loss: 0.2876, Accuracy: 0.8847
Epoch 1400, Loss: 0.2836, Accuracy: 0.8861
Epoch 1500, Loss: 0.2801, Accuracy: 0.8861
Epoch 1600, Loss: 0.2769, Accuracy: 0.8903
Epoch 1700, Loss: 0.2739, Accuracy: 0.8903
Epoch 1800, Loss: 0.2712, Accuracy: 0.8903
Epoch 1900, Loss: 0.2690, Accuracy: 0.8903
Epoch 2000, Loss: 0.2670, Accuracy: 0.8917
Epoch 2100, Loss: 0.2653, Accuracy: 0.8903
Epoch 2200, Loss: 0.2638, Accuracy: 0.8889
Epoch 2300, Loss: 0.2625, Accuracy: 0.8931
Epoch 2400, Loss: 0.2613, Accuracy: 0.8944
Epoch 2500, Loss: 0.2602, Accuracy: 0.8972
Epoch 2600, Loss: 0.2592, Accuracy: 0.8958
Epoch 2700, Loss: 0.2583, Accuracy: 0.8958
Epoch 2800, Loss: 0.2575, Accuracy: 0.8958
Epoch 2900, Loss: 0.2567, Accuracy: 0.8958
Epoch 3000, Loss: 0.2560, Accuracy: 0.8958
Epoch 3100, Loss: 0.2554, Accuracy: 0.8958
Epoch 3200, Loss: 0.2547, Accuracy: 0.8972
Epoch 3300, Loss: 0.2542, Accuracy: 0.8958
Epoch 3400, Loss: 0.2536, Accuracy: 0.8958
Epoch 3500, Loss: 0.2531, Accuracy: 0.8958
Epoch 3600, Loss: 0.2526, Accuracy: 0.8958
Epoch 3700, Loss: 0.2521, Accuracy: 0.8958
Epoch 3800, Loss: 0.2517, Accuracy: 0.8958
Epoch 3900, Loss: 0.2512, Accuracy: 0.8958
Epoch 4000, Loss: 0.2508, Accuracy: 0.8972
Epoch 4100, Loss: 0.2504, Accuracy: 0.8972
Epoch 4200, Loss: 0.2500, Accuracy: 0.8972
Epoch 4300, Loss: 0.2497, Accuracy: 0.8986
Epoch 4400, Loss: 0.2493, Accuracy: 0.8986
Epoch 4500, Loss: 0.2490, Accuracy: 0.8986
Epoch 4600, Loss: 0.2486, Accuracy: 0.8986
Epoch 4700, Loss: 0.2483, Accuracy: 0.8986
Epoch 4800, Loss: 0.2480, Accuracy: 0.8986
Epoch 4900, Loss: 0.2477, Accuracy: 0.8986
Epoch 5000, Loss: 0.2474, Accuracy: 0.9000
Epoch 5100, Loss: 0.2471, Accuracy: 0.9000
Epoch 5200, Loss: 0.2469, Accuracy: 0.9000
Epoch 5300, Loss: 0.2466, Accuracy: 0.9000
Epoch 5400, Loss: 0.2463, Accuracy: 0.9000
Epoch 5500, Loss: 0.2461, Accuracy: 0.9000
Epoch 5600, Loss: 0.2459, Accuracy: 0.9000
```

```
Epoch 5700, Loss: 0.2456, Accuracy: 0.9000
Epoch 5800, Loss: 0.2454, Accuracy: 0.8986
Epoch 5900, Loss: 0.2452, Accuracy: 0.8986
Epoch 6000, Loss: 0.2450, Accuracy: 0.8986
Epoch 6100, Loss: 0.2448, Accuracy: 0.8986
Epoch 6200, Loss: 0.2446, Accuracy: 0.8986
Epoch 6300, Loss: 0.2444, Accuracy: 0.8986
Epoch 6400, Loss: 0.2442, Accuracy: 0.8986
Epoch 6500, Loss: 0.2440, Accuracy: 0.9014
Epoch 6600, Loss: 0.2438, Accuracy: 0.9014
Epoch 6700, Loss: 0.2436, Accuracy: 0.9028
Epoch 6800, Loss: 0.2435, Accuracy: 0.9042
Epoch 6900, Loss: 0.2434, Accuracy: 0.9042
Epoch 7000, Loss: 0.2433, Accuracy: 0.9042
Epoch 7100, Loss: 0.2432, Accuracy: 0.9042
Epoch 7200, Loss: 0.2431, Accuracy: 0.9042
Epoch 7300, Loss: 0.2429, Accuracy: 0.9056
Epoch 7400, Loss: 0.2427, Accuracy: 0.9056
Epoch 7500, Loss: 0.2425, Accuracy: 0.9056
Epoch 7600, Loss: 0.2423, Accuracy: 0.9069
Epoch 7700, Loss: 0.2420, Accuracy: 0.9069
Epoch 7800, Loss: 0.2417, Accuracy: 0.9069
Epoch 7900, Loss: 0.2414, Accuracy: 0.9069
Epoch 8000, Loss: 0.2411, Accuracy: 0.9069
Epoch 8100, Loss: 0.2408, Accuracy: 0.9069
Epoch 8200, Loss: 0.2405, Accuracy: 0.9069
Epoch 8300, Loss: 0.2402, Accuracy: 0.9069
Epoch 8400, Loss: 0.2399, Accuracy: 0.9069
Epoch 8500, Loss: 0.2396, Accuracy: 0.9069
Epoch 8600, Loss: 0.2394, Accuracy: 0.9069
Epoch 8700, Loss: 0.2391, Accuracy: 0.9069
Epoch 8800, Loss: 0.2388, Accuracy: 0.9069
Epoch 8900, Loss: 0.2385, Accuracy: 0.9069
Epoch 9000, Loss: 0.2383, Accuracy: 0.9069
Epoch 9100, Loss: 0.2380, Accuracy: 0.9069
Epoch 9200, Loss: 0.2377, Accuracy: 0.9069
Epoch 9300, Loss: 0.2375, Accuracy: 0.9069
Epoch 9400, Loss: 0.2372, Accuracy: 0.9069
Epoch 9500, Loss: 0.2370, Accuracy: 0.9069
Epoch 9600, Loss: 0.2368, Accuracy: 0.9069
Epoch 9700, Loss: 0.2365, Accuracy: 0.9069
Epoch 9800, Loss: 0.2363, Accuracy: 0.9069
Epoch 9900, Loss: 0.2361, Accuracy: 0.9069
```

The output of the model shows a solid a consistent improvement over the training period, both in terms of decreasing loss and increasing accuracy, a good indicator of a well-performing model. The neural network is effectively learning from the data and is becoming progressively better at classifying raisins.

The training starts with a relatively high accuracy and a moderate loss, which indicates that the initial random weights were effective for this task. The initial accuracy at epoch 0 suggest that even before any training, the structure of the model process the data in a meaninful way. The loss and accuracy show consistent improvement over the epochs. This gradual improvement is a sign of effective learning and optimization.

The model achieves an accuracy plateau of around 0.9069 after epoch 7600, with some minimal fluctuations from epoch 5000 to 7200. The stability of the accuracy may sugggests that the model has reached a point that is close ot its maximum potential given the current architecture. In order to see additional improvements, it might be required require changes in the architecture of the model or hyperparameter (e.g., learning rate, epochs, sizes of the hidden layer etc.).

[153]:
```
def predict(nn, X):
    """Generate predictions using the forward pass of the neural network."""
    predictions = nn.forward(X)
    return predictions >= 0.5  # Threshold predictions to get binary output

# Generate predictions for the test set
predictions = predict(nn, Xtest)

# Calculate accuracy
accuracy = np.mean(predictions == Ytest)

print(f'Accuracy on the test set: {accuracy:.4f}')
```

```
Accuracy on the test set: 0.8556
```

The model achieved an accuracy of 0.8556 on the test set. This indicates its strong ability to generalize to unseen data. It is common to see test accuracy lower than training accuracy. This happens because the model is optimized based on the training data. It may not perform as well on new data. However, as we can see by the high test accuracy, which highlights the model's effectiveness, it makes accurate predictions on new data, confirming its generalizability.