Made by

Lucas Kristiansson 980320-5971

Rikard Radovac 010826-8376

Carolina Rönnewall 980322-7900

In [ ]:
```
import pandas as pd
import numpy as np
import torch
```

Task 1

Loading the synthetic dataset.

In [ ]:
```
# You may need to edit the path, depending on where you put the files.
data = pd.read_csv("data/a4_synthetic.csv")

X = data.drop(columns="y").to_numpy()
Y = data.y.to_numpy()
```

Training a linear regression model for this synthetic dataset.

In [ ]:
```
np.random.seed(1)

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

# We just declare the parameter tensors. Do not use nn.Linear.
w = torch.tensor(w_init, requires_grad=True)
b = torch.tensor(b_init, requires_grad=True)

eta = 1e-2
opt = torch.optim.SGD([w, b], lr=eta)

for i in range(10):
    sum_err = 0

    for row in range(X.shape[0]):
        x = torch.tensor(X[[row], :])
        y = torch.tensor(Y[[row]])

        # Forward pass
        opt.zero_grad()
        y_pred = x @ w + b
        err = (y_pred - y) ** 2
```

```
        # Backward and update.
        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 = 0.7999661130823178
Epoch 2: MSE = 0.017392390107906875
Epoch 3: MSE = 0.009377418010839892
Epoch 4: MSE = 0.009355326971438456
Epoch 5: MSE = 0.009365440968904256
Epoch 6: MSE = 0.009366989180952533
Epoch 7: MSE = 0.009367207398577986
Epoch 8: MSE = 0.009367238983974489
Epoch 9: MSE = 0.009367243704122532
Epoch 10: MSE = 0.009367244427185763
```

Task 2

In [ ]:

```
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()

    # For Task 2:

    # Operator +
    def __add__(self, right):
        # performs add operation
        new_data = self.data + right.data
        grad_fn = AdditionNode(self, right)
```

```
        if self.requires_grad or right.requires_grad:
            return Tensor(new_data, grad_fn=grad_fn, requires_grad=True)
        return Tensor(new_data, grad_fn=grad_fn)

    # Operator -
    def __sub__(self, right):
        # performs subtraction operation
        new_data = self.data - right.data
        grad_fn = SubtractionNode(self, right)
        if self.requires_grad or right.requires_grad:
            return Tensor(new_data, grad_fn=grad_fn, requires_grad=True)
        return Tensor(new_data, grad_fn=grad_fn)

    # Operator @
    def __matmul__(self, right):
        # performs matrix multiplication operation
        new_data = self.data @ right.data
        grad_fn = MatMulNode(self, right)
        if self.requires_grad or right.requires_grad:
            return Tensor(new_data, grad_fn=grad_fn, requires_grad=True)
        return Tensor(new_data, grad_fn=grad_fn)

    # Operator **
    def __pow__(self, right):
        # performs power operation
        # 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")
        grad_fn = PowNode(self, right)
        new_data = self.data**right
        if self.requires_grad:
            return Tensor(new_data, grad_fn=grad_fn, requires_grad=True)
        return Tensor(new_data, grad_fn=grad_fn)

    def tanh(self):
        # performs tanh operation
        new_data = np.tanh(self.data)
        grad_fn = TanhNode(self)
        if self.requires_grad:
            return Tensor(new_data, grad_fn=grad_fn, requires_grad=True)
        return Tensor(new_data, grad_fn=grad_fn)

    def sigmoid(self):
        # performs sigmoid operation
        new_data = sigmoid(self.data)
        grad_fn = SigmoidNode(self)
        if self.requires_grad:
            return Tensor(new_data, grad_fn=grad_fn, requires_grad=True)
        return Tensor(new_data, grad_fn=grad_fn)

    # Backward computations. Will be implemented in Task 4.
    def backward(self, grad_output=None):
        # We first check if this tensor has a grad_fn: that is, one of the
        # nodes that you defined in Taskprint("called backward")
```

```python
        if self.grad_fn is not None:
            # If grad_fn is defined, we have computed this tensor using some
            if grad_output is None:
                # This is the starting point of the backward computation.
                # This will typically be the tensor storing the output of
                # the loss function, on which we have called .backward()
                # in the training loop.

                # We always have a gradient of 1.0 with respect to the outpu
                self.grad_fn.backward(1)

            else:
                # This is an intermediate node in the computational graph.
                # This corresponds to any intermediate computation, such as

                # Here we simply pass the current gradient for this tensor a
                self.grad_fn.backward(self.grad)

        else:
            # If grad_fn is not defined, this is an endpoint in the computat
            # graph: learnable model parameters or input data.

            if self.requires_grad:
                # This tensor *requires* a gradient to be computed. This wil
                # typically be a tensor that holds learnable parameters.

                # The resulting gradient is simply the output gradient

                self.grad = grad_output
            else:
                # This tensor *does not require* a gradient to be computed.
                # will typically be a tensor holding input data.

                self.grad = None


# A small utility where we simply create a Tensor object. We use this to
# mimic torch.tensor.
def tensor(data, requires_grad=False):
    return Tensor(data, requires_grad)


def sigmoid(x):
    """Helper function to compute the sigmoid."""
    return 1 / (1 + np.exp(-x))
```

Some sanity checks.

In [ ]:

```python
# 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
test_combination = (x1**2 - x2 @ w) ** 3

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.d


# Check that the results are as expected. Will crash if there is a miscalcul
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]]))
assert np.allclose(test_combination.data, np.array([[8.00000e-03, 1.40608e02
```

```
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]]
```

Tasks 3 and 4

In [ ]:

```
class Node:
    def __init__(self):
        pass

    def backward(self, grad_output):
        raise NotImplementedError("Unimplemented")

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


class AdditionNode(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right



    def backward(self, grad_output):
        # result = left + right
        # d_L/d_left = d_L/d_result * d_result/d_left = grad_output * 1
        # d_L/d_right = d_L/d_result * d_result/d_right = grad_output * 1

        self.left.grad = grad_output
        self.right.grad = grad_output
        self.right.backward(self.right.grad)
        self.left.backward(self.left.grad)
```

```python
class SubtractionNode(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def backward(self, grad_output):
        # result = left - right
        # d_L/d_left = d_L/d_result * d_result/d_left = grad_output * 1
        # d_L/d_right = d_L/d_result * d_result/d_right = grad_output * -1

        self.left.grad = grad_output
        self.right.grad = -grad_output
        self.right.backward(self.right.grad)
        self.left.backward(self.left.grad)


class MatMulNode(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def backward(self, grad_output):
        # result = left @ right
        # d_L/d_left = d_L/d_result * d_result/d_left = grad_output @ right.
        # d_L/d_right = d_L/d_result * d_result/d_right = left.T @ grad_outp


        self.left.grad =  grad_output @ self.right.data.T
        self.right.grad = self.left.data.T @ grad_output

        self.right.backward(self.right.grad)
        self.left.backward(self.left.grad)


class PowNode(Node):
    def __init__(self, left, right):
        self.left = left
        self.right = right

    def backward(self, grad_output):
        # result = left ** right
        # d_L/d_left = d_L/d_result * d_result/d_left = grad_output * right

        self.left.grad = (self.right * self.left.data ** (self.right - 1)) *
        self.left.backward(self.left.grad)


class TanhNode(Node):
    def __init__(self, left):
        self.left = left

    def backward(self, grad_output):
        # result = tanh(left)
        # d_L/d_left = d_L/d_result * d_result/d_left = grad_output * (1 - t
```

```
        self.left.grad = (1 - (np.tanh(self.left.data) ** 2)) * grad_output
        self.left.backward(self.left.grad)

class SigmoidNode(Node):
    def __init__(self, left):
        self.left = left

    def backward(self, grad_output):
        # result = sigmoid(left)
        # d_L/d_left = d_L/d_result * d_result/d_left = grad_output * (sigmo

        self.left.grad = grad_output * (sigmoid(self.left.data) * (1 - sigmo
        self.left.backward(self.left.grad)

class BCELossNode(Node):
    def __init__(self, y_pred, y_true):
        self.y_pred = y_pred
        self.y_true = y_true

    def forward(self):
        # forward for BCEloss
        self.output = -np.mean(self.y_true.data * np.log(self.y_pred.data) +
        return self.output

    def backward(self, grad_output):
        # result = -np.mean(y_true * np.log(y_pred) + (1 - y_true) * np.log(
        # d_L/d_y_pred = d_L/d_result * d_result/d_y_pred = grad_output * (-

        num_elements = np.prod(self.y_pred.data.shape)
        grad_y_pred = grad_output * (-self.y_true.data / self.y_pred.data +
        self.y_pred.grad = grad_y_pred
        self.y_pred.backward(self.y_pred.grad)
```

Sanity check for Task 3.

In [ ]:
```
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__.AdditionNod
e'>
```

Sanity check for Task 4.

In [ ]:

```python
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]]
```

An equivalent cell using PyTorch code. Your implementation should give the same result for `w.grad`.

In [ ]:

```python
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 de

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
```

Out[ ]:

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

Task 5

In [ ]:

```python
class Optimizer:
    def __init__(self, params):
        self.params = params

    def zero_grad(self):
        for p in self.params:
```

```python
            p.grad = np.zeros_like(p.data)

    def step(self):
        raise NotImplementedError("Unimplemented")


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

    def step(self):
        # Update the parameters in the negative gradient direction of the gr
        for p in self.params:
            if p.requires_grad:
                p.data -= self.lr * p.grad
```

In [ ]:

```python
# You may need to edit the path, depending on where you put the files.
np.random.seed(1)
data = pd.read_csv("data/a4_synthetic.csv")

X = data.drop(columns="y").to_numpy()
Y = data.y.to_numpy()


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

# We just declare the parameter tensors. Do not use nn.Linear.
w = tensor(w_init, requires_grad=True)
b = tensor(b_init, requires_grad=True)

eta = 1e-2
opt = SGD([w, b], lr=eta)

for i in range(10):
    sum_err = 0

    for row in range(X.shape[0]):
        x = tensor(X[[row], :])
        y = tensor(Y[[row]])

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

        # Backward and update.
        # TODO: compute gradients and then update the model.
        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 = 0.7999661130823178
Epoch 2: MSE = 0.017392390107906875
Epoch 3: MSE = 0.009377418010839892
Epoch 4: MSE = 0.009355326971438456
Epoch 5: MSE = 0.009365440968904256
Epoch 6: MSE = 0.009366989180952533
Epoch 7: MSE = 0.009367207398577986
Epoch 8: MSE = 0.009367238983974489
Epoch 9: MSE = 0.009367243704122532
Epoch 10: MSE = 0.009367244427185763
```

Task 6

In [ ]:

```
from sklearn.preprocessing import scale
from sklearn.model_selection import train_test_split

# You may need to edit the path, depending on where you put the files.
a4data = pd.read_csv("data/raisins.csv")

X = scale(a4data.drop(columns="Class"))
Y = 1.0 * (a4data.Class == "Besni").to_numpy()

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_s
```

In [ ]:

```
class Module:
    # Create a basic module to zero the gradients of all parameters.
    def zero_grad(self):
        for p in self.parameters():
            p.grad = np.zeros_like(p.data)

    def parameters(self):
        raise NotImplementedError("Method for getting parameters is not impl

class Layer(Module):
    # A simple dense layer with a tanh or sigmoid activation function., crea
    def __init__(self, in_features, out_features, activation: str = "tanh"):
        self.w = tensor(np.random.normal(size=(in_features, out_features)),
        self.b = tensor(np.random.normal(size=(1, out_features)), requires_g
        self.activation = activation

    def __call__(self, x):
        model_output = x @ self.w + self.b

        if self.activation == "tanh":
            return model_output.tanh()
        elif self.activation == "sigmoid":
```

```python
            return model_output.sigmoid()
        else:
            raise ValueError(f"Unknown activation function: {self.activation

    def parameters(self):
        return [self.w, self.b]

class MLP(Module):
    # A simple multi-layer perceptron with one hidden layer.
    def __init__(self, in_features, hidden_features, out_features):
        self.hidden = Layer(in_features, hidden_features)
        self.out = Layer(hidden_features, out_features, activation="sigmoid"

    def __call__(self, x):
        return self.out(self.hidden(x))

    def parameters(self):
        return [p for layer in [self.hidden, self.out] for p in layer.parame

    def disable_grad(self):
        for p in self.parameters():
            p.requires_grad = False

def bce_loss(y_pred, y_true):
    # Compute the binary cross-entropy loss.
    return Tensor(BCELossNode(y_pred, y_true).forward(), grad_fn=BCELossNode
```

In [ ]:

```python
def test_vs_torch():
    """Simple test to compare the implementation for sigmoid, tanh and bcelo
    bce = torch.nn.BCELoss()


    w_init = np.random.normal(size=(2, 1))
    x_init = np.random.normal(size=(1, 2))
    w_torch = torch.tensor(w_init, requires_grad=True)
    x_torch = torch.tensor(x_init, requires_grad=False)
    w_tensor = tensor(w_init, requires_grad=True)
    x_tensor = tensor(x_init, requires_grad=False)

    out_torch = x_torch @ w_torch
    out_torch = out_torch.sigmoid()
    err_torch = bce(out_torch, torch.tensor([[1.0]], dtype=torch.float64))
    err_torch.backward()

    out_tensor = x_tensor @ w_tensor
    out_tensor = out_tensor.sigmoid()
    err_tensor = bce_loss(out_tensor, tensor(np.array([[1.0]])))
    err_tensor.backward()


    assert np.allclose(w_tensor.grad, w_torch.grad.numpy()), "Gradients do n
    assert np.allclose(out_tensor.data, out_torch.data.numpy()), "Output doe


    w_torch = torch.tensor(w_init, requires_grad=True)
    x_torch = torch.tensor(x_init, requires_grad=False)
```

```
    w_tensor = tensor(w_init, requires_grad=True)
    x_tensor = tensor(x_init, requires_grad=False)

    out_torch = x_torch @ w_torch
    out_torch = out_torch.tanh()

    out_tensor = x_tensor @ w_tensor
    out_tensor = out_tensor.tanh()

    assert np.allclose(out_tensor.data, out_torch.data.numpy()), "Output doe


test_vs_torch()
```

In [ ]:
```
np.random.seed(1)
model = MLP(Xtrain.shape[1], 100, 1)



def train_model(model, X, Y, epochs=10, lr=0.01):
    opt = SGD(model.parameters(), lr=lr)
    for epoch in range(epochs):
        sum_err = 0
        for row in range(X.shape[0]):
            x = tensor(X[[row], :])
            y = tensor(Y[[row]])

            opt.zero_grad()
            y_pred = model(x)
            err = bce_loss(y_pred, y)

            err.backward()
            opt.step()
            sum_err += err.item()
        bce = sum_err / X.shape[0]
        print(f"Epoch {epoch+1}: BCE =", bce)



def evaluate_model(model, test_data, test_labels):
    # disable grad for more efficient evaluation (we do not need to store gr
    model.disable_grad()
    correct = 0
    total = 0

    for x, y in zip(test_data, test_labels):
        inputs = tensor(x)
        label = tensor(y)
        outputs = model(inputs)
        predicted = np.round(outputs.item())
        total += 1
        correct += (predicted == label.item())

    accuracy = correct / total
```

```python
    print(f"Accuracy on test set: {accuracy:.2%}")


train_model(model, Xtrain, Ytrain, epochs=10)

evaluate_model(model, Xtest, Ytest)
```

```
Epoch 1: BCE = 0.6234923395239745
Epoch 2: BCE = 0.4014429960012952
Epoch 3: BCE = 0.3636728892030264
Epoch 4: BCE = 0.33976173890169425
Epoch 5: BCE = 0.32221372001882365
Epoch 6: BCE = 0.3098793292138743
Epoch 7: BCE = 0.30177117733037573
Epoch 8: BCE = 0.29620560115160516
Epoch 9: BCE = 0.2921147371002634
Epoch 10: BCE = 0.2889261433814792
Accuracy on test set: 85.56%
```