Deep Learning — Assignment 1

First assignment for the 2023 Deep Learning course (NWI-IMC070) of the Radboud University.

Names:

Group:

Instructions:

- Fill in your names and the name of your group.
- Answer the questions and complete the code where necessary.
- Keep your answers brief, one or two sentences is usually enough.
- Re-run the whole notebook before you submit your work.
- Save the notebook as a PDF and submit that in Brightspace together with the `.ipynb` notebook file.
- The easiest way to make a PDF of your notebook is via File > Print Preview and then use your browser's print option to print to PDF.

Objectives

In this assignment you will

1. Experiment with gradient descent optimization;
2. Derive and implement gradients for binary cross-entropy loss, the sigmoid function and a linear layer;
3. Test your gradient implementations with the finite difference method;
4. Use these components to implement and train a simple neural network.

In [ ]:
```
%matplotlib inline
import numpy as np
import scipy.optimize
import sklearn.datasets
import matplotlib.pyplot as plt

np.set_printoptions(suppress=True, precision=6, linewidth=200)
plt.style.use('ggplot')
```

1.1 Gradient descent optimization (6 points)

Consider the following function with two parameters and its derivatives:

$$f(x, y) = x^2 + y^2 + x(y + 2) + \cos(3x) \tag{1}$$

$$\frac{\partial f}{\partial x} = 2x - 3\sin(3x) + y + 2 \tag{2}$$

$$\frac{\partial f}{\partial y} = x + 2y \tag{3}$$

In [ ]:
```
def f(x, y):
    return x ** 2 + y ** 2 + x * (y + 2) + np.cos(3 * x)
def grad_x_f(x, y):
    return 2 * x - 3 * np.sin(3 * x) + y + 2
def grad_y_f(x, y):
    return x + 2 * y
```

A plot of the function shows that it has multiple local minima:

In [ ]:
```
def plot_f_contours():
    xx, yy = np.meshgrid(np.linspace(-5, 5), np.linspace(-5, 5))
    zz = f(xx, yy)
    plt.contourf(xx, yy, zz, 50)
    plt.contour(xx, yy, zz, 50, alpha=0.2, colors='black', linestyles='solid
    plt.xlabel('x')
    plt.ylabel('y')

plt.figure(figsize=(10, 7))
plot_f_contours()
```

Implement gradient descent

We would like to find the minimum of this function using gradient descent.

(a) Implement the gradient descent updates for x and y in the function below: (1 point)

In [ ]:
```
def optimize_f(x, y, step_size, steps):
    # keep track of the parameters we tried so far
    x_hist, y_hist = [x], [y]

    # run gradient descent for the number of steps
    for step in range(steps):
        # compute the gradients at the current point
        dx = grad_x_f(x, y)
        dy = grad_y_f(x, y)

        # apply the gradient descent updates to x and y
        x = x  # TODO: compute the update
        y = y  # TODO: compute the update

        # store the new parameters
        x_hist.append(x)
        y_hist.append(y)
```

```
    return x, y, f(x, y), x_hist, y_hist
```

In [ ]:
```
# The following assert statements check that your implementation behaves ser
# Use it to get a hint only if you are stuck.
assert optimize_f(3, 2, 0.1, 1)[0] != 3, "Hint: you are not changing `x`"
assert optimize_f(3, 2, 0.1, 1)[2] < f(3, 2), "Hint: the function value is i
assert abs(optimize_f(3, 2, 0.1, 1)[0] - 3) < 1, "Hint: you are probably tak
```

Tune the parameters

We will now try if our optimization method works.

Use this helper function to plot the results:

In [ ]:
```
# helper function that plots the results of the gradient descent optimizatio
def plot_gradient_descent_results(x, y, val, x_hist, y_hist):
    # plot the path on the contour plot
    plt.figure(figsize=(20, 7))
    plt.subplot(1, 2, 1)
    plot_f_contours()
    plt.plot(x_hist, y_hist, '.-')

    # plot the learning curve
    plt.subplot(1, 2, 2)
    plt.plot(f(np.array(x_hist), np.array(y_hist)), '.r-')
    plt.title('Minimum value: %f' % f(x_hist[-1], y_hist[-1]))
```

(b) Run the gradient descent optimization with the following initial settings:

`x=3, y=2, step_size=0.1, steps=10`

In [ ]:
```
results = optimize_f(x=3, y=2, step_size=0.1, steps=10)
plot_gradient_descent_results(*results)
```

(c) Does it find the minimum of the function? What happens? **(1 point)**

TODO: Your answer here.

(d) Try a few different values for the `step_size` and the number of `steps` to get close to the optimal solution:

In [ ]:
```
# TODO: tune the parameters to find a better optimum
results = optimize_f(x=3, y=2, step_size=0.1, steps=10)
plot_gradient_descent_results(*results)
```

(e) What happens if you set the step size too small? And what if it is too large? **(1 point)**

(f) Were you able to find a step size that reached the global optimum? If not, why not? **(1 point)**

TODO: Your answer here.

Implement a decreasing step size

You might get better results if you use a step size that is large at the beginning, but slowly decreases during the optimization.

Try the following scheme to compute the step size η_t in step t, given a decay parameter d:

$$\eta_t = \eta_0 d^t \tag{4}$$

(g) Update your optimization function to use this step size schedule: (1 point)

In [ ]:
```
def optimize_f(x, y, step_size, steps, decay=1.0):
    # keep track of the parameters we tried so far
    x_hist, y_hist = [x], [y]

    # run gradient descent for the number of steps
    for step in range(steps):
        # compute the gradients at this point
        dx = grad_x_f(x, y)
        dy = grad_y_f(x, y)

        # apply the gradient descent updates to x and y
        x = x  # TODO: compute the update including step size decay
        y = y  # TODO: compute the update including step size decay

        # store the new parameters
        x_hist.append(x)
        y_hist.append(y)

    return x, y, f(x, y), x_hist, y_hist
```

In [ ]:
```
# The following assert statement checks that your implementation behaves ser
_trace = optimize_f(0.123, 0.456, 0.01, 2, 0.1)[3]
assert abs(_trace[1] - _trace[0]) > 5 * abs(_trace[2] - _trace[1]), "Hint: s
del _trace
```

(h) Tune the `step_sizes`, `steps` and `decay` parameters to get closer to the global minimum: (1 point)

In [ ]:
```
# TODO: tune the parameters to find the global optimum
results = optimize_f(x=3, y=2, step_size=0.1, steps=10, decay=1)
```

In [ ]:
```
assert results[2] < -2, "Hint: get closer to the optimum"
```

We will now look at some more complex functions that we can try to optimize.

1.2 Neural network components (16 points)

In this assignment, we will implement a simple neural network from scratch. We need four components:

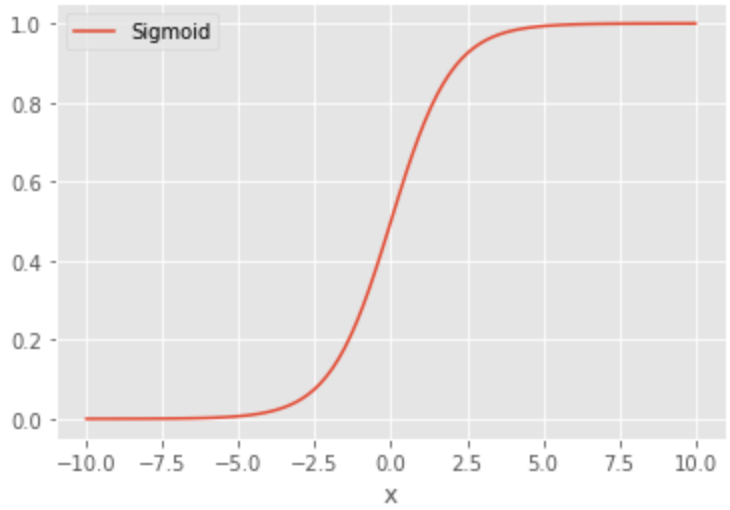
1. A sigmoid activation function,
2. A ReLU activation function,
3. A binary cross-entropy loss function,
4. A linear layer.

For each component, we will implement the forward pass, the backward pass, and the gradient descent update.

Sigmoid non-linearity

The sigmoid function is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{5}$$



(a) Give the derivative of the sigmoid function: **(1 point)**

$$\frac{\partial \sigma(x)}{\partial x} = \text{TODO Your answer here.} \tag{6}$$

(b) Implement the sigmoid and its gradient in the functions `sigmoid(x)` and `sigmoid_grad(x)` : **(2 points)**

In [ ]:
```python
def sigmoid(x):
    # TODO: implement the sigmoid function
    raise NotImplementedError

def sigmoid_grad(x):
    # TODO: implement the gradient of the sigmoid function
    raise NotImplementedError
```

```
# try with a random input
rng = np.random.default_rng(12345)
x = rng.uniform(-10, 10, size=5)
print('x:', x)
print('sigmoid(x):', sigmoid(x))
print('sigmoid_grad(x):', sigmoid_grad(x))
```

To check that the gradient implementation is correct, we can compute the numerical derivative using the finite difference method. From Chapter 11.5 of the Deep Learning book:

> Because

$$f'(x) = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}, \tag{7}$$

we can approximate the derivative by using a small, finite ϵ:

$$f'(x) \approx \frac{f(x+\epsilon) - f(x)}{\epsilon}. \tag{8}$$

We can improve the accuracy of the approximation by using the centered difference:

$$f'(x) \approx \frac{f(x+\frac{1}{2}\epsilon) - f(x-\frac{1}{2}\epsilon)}{\epsilon}. \tag{9}$$

The perturbation size ϵ must be large enough to ensure that the perturbation is not rounded down too much by finite-precision numerical computations.

(c) Use the central difference method to check your implementation of the sigmoid gradient. Compute the numerical gradient and check that it is close to the symbolic gradient computed by your implementation: (1 point)

In [ ]:
```
# start with some random inputs
rng = np.random.default_rng(12345)
x = rng.uniform(-2, 2, size=5)

# compute the symbolic gradient
print('Symbolic ', sigmoid_grad(x))

# TODO: compute the numerical gradient
#PLACEHOLDER print('Numerical', TODO)
```

(d) Is the gradient computed with finite differences exactly the same as the analytic answer? Why (not)? (1 point)

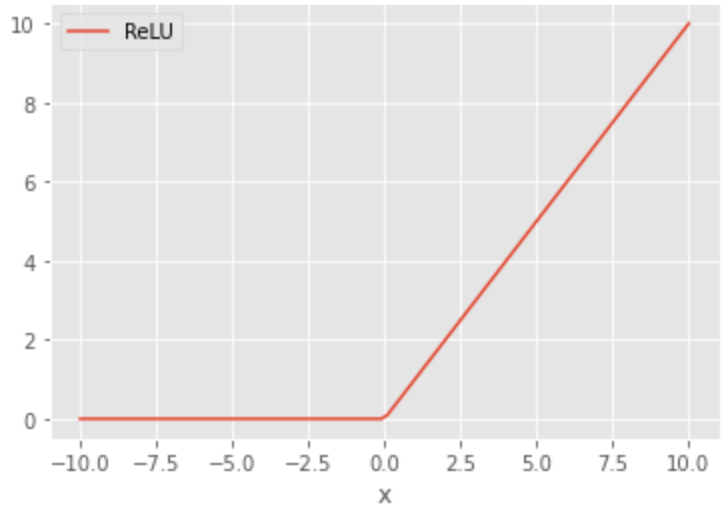
TODO: Your answer here.

If there is a big difference between the two gradients, please try to make this as small as possible before you continue.

Rectified linear units (ReLU)

The rectified linear unit is defined as:

$$f(x) = \max(0, x) \tag{10}$$



(e) Give the derivative of the ReLU function: **(1 point)**

Note: this gradient is not well-defined everywhere, but make a sensible choice for all values of x.

$$\frac{\partial f(x)}{\partial x} = \text{TODO Your answer here.} \tag{11}$$

(f) Implement the ReLU function and its gradient in the functions `relu(x)` and `relu_grad(x)`. Use the finite difference method to check that the gradient is correct: **(2 points)**

In [ ]:
```python
def relu(x):
    # TODO: implement the relu function
    raise NotImplementedError

def relu_grad(x):
    raise NotImplementedError

# try with a random input
rng = np.random.default_rng(12345)
x = rng.uniform(-10, 10, size=5)
print('x:', x)
print('relu(x):', relu(x))
print('relu_grad(x):', relu_grad(x))
print()

# TODO: compute and compare the symbolic and numerical gradients
```

Comparing sigmoid and ReLU

The sigmoid and ReLU activation functions have slightly different characteristics.

(g) Run the code below to plot the sigmoid and ReLU activation functions and their gradients:

In [ ]:
```
x = np.linspace(-10, 10, 100)

plt.figure(figsize=(15, 8))

plt.subplot(2, 2, 1)
plt.plot(x, sigmoid(x), label='Sigmoid')
plt.xlabel('x')
plt.legend(loc='upper left')

plt.subplot(2, 2, 2)
plt.plot(x, relu(x), label='ReLU')
plt.xlabel('x')
plt.legend(loc='upper left')

plt.subplot(2, 2, 3)
plt.plot(x, sigmoid_grad(x), label='Sigmoid gradient')
plt.xlabel('x')
plt.legend(loc='upper left')

plt.subplot(2, 2, 4)
plt.plot(x, relu_grad(x), label='ReLU gradient')
plt.xlabel('x')
plt.legend(loc='upper left');
```

(h) Which activation function would you recommend for a network that outputs probabilities, i.e., outputs $\in [0, 1]$? Why? **(1 point)**

TODO: Your answer here.

(i) Compare the gradients for sigmoid and ReLU. What are the advantages and disadvantages of each activation function in terms of their gradient? **(1 point)**

TODO: Your answer here.

Binary cross-entropy loss

We will use the binary cross-entropy loss to train our network. This loss function is useful for binary classification.

The binary cross-entropy (BCE) is a function of the ground truth label $y \in \{0, 1\}$ and the predicted label $\hat{y} \in [0, 1]$:

$$\mathcal{L} = -(y \log \hat{y} + (1 - y) \log(1 - \hat{y})) \tag{12}$$

To minimize the BCE loss with gradient descent, we need to compute the gradient with respect to the prediction $\hat{y}$.

(j) Derive the gradient for the BCE loss: **(1 point)**

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = \text{TODO: Your answer here.} \tag{13}$$

(k) Implement `bce_loss(y, y_hat)` and `bce_loss_grad(y, y_hat)` and use the finite difference method to check that the gradient is correct: **(3 points)**

In [ ]:
```python
def bce_loss(y, y_hat):
    # TODO: implement the BCE loss
    raise NotImplementedError

def bce_loss_grad(y, y_hat):
    # TODO: implement the gradient of the BCE loss
    raise NotImplementedError

# try with some random inputs
rng = np.random.default_rng(12345)
y = rng.integers(2, size=5)
y_hat = rng.uniform(0, 1, size=5)
print('y:', y)
print('y_hat:', y_hat)
print('bceloss(y, y_hat):', bce_loss(y, y_hat))
print()

# TODO: compute and compare the symbolic and numerical gradients
```

Linear layer

Finally, we need to compute the gradients for the linear layer in our network.

Define a linear model $\mathrm{y} = \mathrm{x}\mathrm{W} + \mathrm{b}$, where

- x is an input vector of shape N,
- W is a weight matrix of shape $N \times M$,
- b is a bias vector of shape M,
- y is the output vector of shape M.

(l) Derive the gradients for y with respect to the input x and the parameters W and b: **(1 point)**

Hint: If you have trouble computing this in matrix notation directly, try to do the computation with scalars, writing the linear model as

$$y_j = \sum_{i=1}^{N} x_i W_{ij} + b_j \tag{14}$$

where j ranges from 1 to M.

TODO: Your answer here.

$$\frac{\partial y_j}{\partial x_i} = TODO \qquad \frac{\partial y_j}{\partial W_{ik}} = TODO \qquad \frac{\partial y_j}{\partial b_k} = TODO \tag{15}$$

or

$$\frac{\partial \mathrm{y}}{\partial \mathrm{x}} = TODO \qquad \frac{\partial \mathrm{y}}{\partial \mathrm{W}} = TODO \qquad \frac{\partial \mathrm{y}}{\partial \mathrm{b}} = TODO \tag{16}$$

(keep only one)

(m) Given the gradient $\nabla_{\mathrm{y}}\mathcal{L}$ for the loss w.r.t. y, use the chain rule to derive the gradients for the loss w.r.t. x, W and b: **(1 point)**

TODO: Your answer here.

$$\nabla_{\mathrm{x}}\mathcal{L} = \tag{17}$$
$$\nabla_{\mathrm{W}}\mathcal{L} = \tag{18}$$
$$\nabla_{\mathrm{b}}\mathcal{L} = \tag{19}$$

1.3 Implement a one-layer model (2 points)

We can now implement a simple one-layer model with a sigmoid activation:

1. Given an input vector x, weight vector w and bias b, compute the output $\hat{y}$:

$$h = \mathrm{x}\mathrm{w}^T + b \tag{20}$$
$$\hat{y} = \sigma(h) \tag{21}$$

2. Compute the BCE loss comparing the prediction $\hat{y}$ with the ground-truth label y.
3. Compute the gradient for the BCE loss and back-propagate this to get $\nabla_{\mathrm{x}}\mathcal{L}$, the gradient of $\mathcal{L}$ w.r.t. x.

Hint: in numpy inner product and matrix multiplication is denoted as `np.dot(A, B)` or as `A @ B`.

(a) Complete the implementation below: **(2 points)**

In [ ]:
```
# initialize parameters
rng = np.random.default_rng(12345)
w = rng.normal(size=5)
b = rng.normal()

# implement the model
```

```
def fn(x, y):
    # TODO: forward: compute h, y_hat, loss
    h = 0
    y_hat = 0
    loss = 0

    # TODO: backward: compute grad_y_hat, grad_h, grad_x
    grad_y_hat = 0
    grad_h = 0
    grad_x = 0

    return loss, grad_x

# test with a random input
x = rng.uniform(size=5)
y = 1

loss, grad_x = fn(x, y)
print("Loss", loss)
print("Gradient", grad_x)

assert np.isscalar(loss), "Loss should be scalar"
assert grad_x.shape == x.shape, "Gradient should have same shape as x"
```

(b) Use the finite-difference method to check the gradient $\nabla_x \mathcal{L}$:

In [ ]:

```
# start with some random inputs
rng = np.random.default_rng(12345)
x = rng.uniform(size=5)
y = 1

# set epsilon to a small value
eps = 0.00001

numerical_grad = np.zeros(x.shape)
# compute the gradient for each element of x separately
for i in range(len(x)):
    # compute inputs at -eps/2 and +eps/2
    x_a, x_b = x.copy(), x.copy()
    x_a[i] += eps / 2
    x_b[i] -= eps / 2

    # compute the gradient for this element
    loss_a, _ = fn(x_a, y)
    loss_b, _ = fn(x_b, y)
    numerical_grad[i] = (loss_a - loss_b) / eps

# compute the symbolic gradient
loss, symbolic_grad = fn(x, y)

print("Symbolic gradient")
print(symbolic_grad)
print("Numerical gradient")
print(numerical_grad)
```

1.4 Implement a linear layer and the sigmoid and ReLU activation functions (5 points)

We will now construct a simple neural network. We need to implement the following objects:

- `Linear` : a layer that computes `y = x*W + b` .
- `Sigmoid` : a layer that computes `y = sigmoid(x)` .
- `ReLU` : a layer that computes `y = relu(x)` .

For each layer class, we need to implement the following methods:

- `forward` : The forward pass that computes the output `y` given `x` .
- `backward` : The backward pass that receives the gradient for `y` and computes the gradients for the input `x` and the parameters of the layer.
- `step` : The update step that applies the gradient updates to the parameters of the layer, based on the gradient computed and stored by `backward` .

(a) Implement a class `Linear` that computes `y = x*W + b` : (3 points)

In [ ]:
```
# Computes y = x * w + b.
class Linear:
    def __init__(self, n_in, n_out, rng = np.random.default_rng(12345)):
        # initialize the weights randomly,
        # using the Xavier initialization rule for scale
        a = np.sqrt(6 / (n_in * n_out))
        self.W = rng.uniform(-a, a, size=(n_in, n_out))
        self.b = np.zeros((n_out,))

    def forward(self, x):
        # TODO: compute the forward pass
        y = 0  # TODO
        return y

    def backward(self, x, dy):
        # TODO: compute the backward pass,
        # given dy, compute the gradients for x, W and b
        dx = 0       # TODO
        self.dW = 0  # TODO
        self.db = 0  # TODO
        return dx

    def step(self, step_size):
        # TODO: apply a gradient descent update step
        self.W = self.W  # TODO
        self.b = self.b  # TODO

    def __str__(self):
        return 'Linear %dx%d' % self.W.shape


# Try the new class with some random values.
# Debugging tip: always choose a unique length for each dimension,
```

```
#  so you'll get an error if you mix them up.
rng = np.random.default_rng(12345)
x = rng.uniform(size=(3, 5))

layer = Linear(5, 7, rng=rng)
y = layer.forward(x)
dx = layer.backward(x, np.ones_like(y))
print('y:', y)
print('dx:', dx)

# Verify correctness
assert y.shape == (3,7)
assert dx.shape == x.shape
layer.W *= 2
layer.b = layer.b * 2 + 1
y2 = layer.forward(x)
dx2 = layer.backward(x, np.ones_like(y))
assert np.all(y2 == 2 * y + 1)
assert np.all(dx2 == 2 * dx)
```

(b) Implement a class `Sigmoid` that computes `y = 1 / (1 + exp(-x))` : (1 point)

In [ ]:
```
# Computes y = 1 / (1 + exp(-x)).
class Sigmoid:
    def forward(self, x):
        # TODO: compute the forward pass
        raise NotImplementedError  # TODO

    def backward(self, x, dy):
        # TODO: compute the backward pass,
        # return the gradient for x given the gradient for y
        raise NotImplementedError  # TODO

    def step(self, step_size):
        raise NotImplementedError  # TODO

    def __str__(self):
        return 'Sigmoid'


# try the new class with some random values
rng = np.random.default_rng(12345)
x = rng.normal(size=(3, 5))

layer = Sigmoid()
y = layer.forward(x)
dx = layer.backward(x, np.ones_like(y))
print('y:', y)
print('dx:', dx)

assert y.shape == x.shape, "Output sigmoid should have the same shape as inp
assert dx.shape == x.shape, "Gradient sigmoid should have the same shape as
assert np.all(y > 0) and np.all(y < 1), "Output of sigmoid should be betweer
```

(c) Implement a class `ReLU` that computes `y = max(0, x)` : (1 point)

In [ ]:
```
# Computes y = max(0, x).
class ReLU:
    def forward(self, x):
        # TODO: compute the forward pass
        raise NotImplementedError  # TODO

    def backward(self, x, dy):
        # TODO: compute the backward pass,
        # return the gradient for x given dy
        raise NotImplementedError  # TODO

    def step(self, step_size):
        raise NotImplementedError  # TODO

    def __str__(self):
        return 'ReLU'


# try the new class with some random values
rng = np.random.default_rng(12345)
x = rng.uniform(-10, 10, size=(3, 5))

layer = ReLU()
y = layer.forward(x)
dx = layer.backward(x, np.ones_like(y))
print('y:', y)
print('dx:', dx)

assert y.shape == x.shape, "Output of ReLU should have the same shape as inp
assert dx.shape == x.shape, "Gradient of ReLU should have the same shape as
```

Verify the gradients

The code below will check your implementations using SciPy's finite difference implementation `check_grad`. This is similar to what we did manually before, but automates some of the work.

(d) Run the code and check that the error is not too large.

In [ ]:
```
## Verify gradient computations for Linear
# test for dx
rng = np.random.default_rng(12345)
layer = Linear(5, 7, rng)
def test_fn(x):
    x = x.reshape(3, 5)
    # multiply the output with a constant to check if
    # the gradient uses dy
    return 2 * np.sum(layer.forward(x))
def test_fn_grad(x):
    x = x.reshape(3, 5)
    # multiply the incoming dy gradient with a constant
    return layer.backward(x, 2 * np.ones((3, 7))).flatten()
```

```
err = scipy.optimize.check_grad(test_fn, test_fn_grad, rng.uniform(-10, 10,
print("err on dx:", err)
assert np.abs(err) < 1e-5, "Error on dx is too large, check your implementat

# test for dW
x = rng.uniform(size=(3, 5))
layer = Linear(5, 7, rng)
def test_fn(w):
    layer.W = w.reshape(5, 7)
    # multiply the output with a constant to check if
    # the gradient uses dy
    return 2 * np.sum(layer.forward(x))
def test_fn_grad(w):
    layer.W = w.reshape(5, 7)
    # multiply the incoming dy gradient with a constant
    layer.backward(x, 2 * np.ones((3, 7)))
    return layer.dW.flatten()

err = scipy.optimize.check_grad(test_fn, test_fn_grad, rng.uniform(-10, 10,
print("err on dW:", err)
assert np.abs(err) < 1e-5, "Error on dW is too large, check your implementat

# test for db
x = rng.uniform(size=(3, 5,))
layer = Linear(5, 7, rng)
def test_fn(b):
    layer.b = b
    # multiply the output with a constant to check if
    # the gradient uses dy
    return 2 * np.sum(layer.forward(x))
def test_fn_grad(b):
    layer.b = b
    # multiply the incoming dy gradient with a constant
    layer.backward(x, 2 * np.ones((x.shape[0], 7)))
    return layer.db

err = scipy.optimize.check_grad(test_fn, test_fn_grad, rng.uniform(-10, 10,
print("err on db:", err)
assert np.abs(err) < 1e-5, "Error on db is too large, check your implementat
```

In [ ]:

```
## Verify gradient computation for Sigmoid
# test for dx
layer = Sigmoid()
def test_fn(x):
    # multiply the output with a constant to check if
    # the gradient uses dy
    return np.sum(2 * layer.forward(x))
def test_fn_grad(x):
    # multiply the incoming dy gradient with a constant
    return layer.backward(x, 2 * np.ones(x.shape))

rng = np.random.default_rng(12345)
err = scipy.optimize.check_grad(test_fn, test_fn_grad, rng.uniform(-10, 10,
print("err on dx:", err)
assert np.abs(err) < 1e-5, "Error on dx is too large, check your implementat
```

In [ ]:
```
## Verify gradient computation for ReLU
# test for dx
layer = ReLU()
def test_fn(x):
    # multiply the output with a constant to check if
    # the gradient uses dy
    return 2 * np.sum(layer.forward(x))
def test_fn_grad(x):
    # multiply the incoming dy gradient with a constant
    return layer.backward(x, 2 * np.ones(x.shape))

rng = np.random.default_rng(12345)
err = scipy.optimize.check_grad(test_fn, test_fn_grad, rng.uniform(1, 10, si
print("err on dx:", err)
assert np.abs(err) < 1e-5, "Error on dx is too large, check your implementat
```

1.5 Construct a neural network with back-propagation

We will use the following container class to implement the network:

1. The `forward` pass computes the output of each layer. We store the intermediate inputs for the backward pass.
2. The `backward` pass computes the gradients for each layer, in reverse order, by using the original input `x` and the gradient `dy` from the previous layer.
3. The `step` function will ask each layer to apply the gradient descent updates to its weights.

(a) Read the code below:

In [ ]:
```
class Net:
    def __init__(self, layers):
        self.layers = layers

    def forward(self, x):
        # compute the forward pass for each layer
        trace = []
        for layer in self.layers:
            # compute the forward pass
            y = layer.forward(x)
            # store the original input for the backward pass
            trace.append((layer, x))
            x = y
        # return the final output and the history trace
        return y, trace

    def backward(self, trace, dy):
        # compute the backward pass for each layer
        for layer, x in trace[::-1]:
            # compute the backward pass using the original input x
            dy = layer.backward(x, dy)

    def step(self, learning_rate):
```

```
        # apply the gradient descent updates of each layer
        for layer in self.layers:
            layer.step(learning_rate)

    def __str__(self):
        return '\n'.join(str(l) for l in self.layers)
```

1.6 Training the network (10 points)

We load a simple dataset with 360 handwritten digits.

Each sample has 8×8 pixels, arranged as a 1D vector of 64 features.

We create a binary classification problem with the label 0 for the digits 0 to 4, and 1 for the digits 5 to 9.

In [ ]:
```
# load the first two classes of the digits dataset
dataset = sklearn.datasets.load_digits()
digits_x, digits_y = dataset['data'], dataset['target']

# create a binary classification problem
digits_y = (digits_y < 5).astype(float)

# plot some of the digits
plt.figure(figsize=(10, 2))
plt.imshow(np.hstack([digits_x[i].reshape(8, 8) for i in range(10)]), cmap='
plt.grid(False)
plt.tight_layout()
plt.axis('off')

# normalize the values to [0, 1]
digits_x -= np.mean(digits_x)
digits_x /= np.std(digits_x)

# print some statistics
print('digits_x.shape:', digits_x.shape)
print('digits_y.shape:', digits_y.shape)
print('min, max values:', np.min(digits_x), np.max(digits_x))
print('labels:', np.unique(digits_y))
```

We divide the dataset in a train and a test set.

In [ ]:
```
# make a 50%/50% train/test split
train_prop = 0.5
n_train = int(digits_x.shape[0] * train_prop)

# shuffle the images
rng = np.random.default_rng(12345)
idxs = rng.permutation(digits_x.shape[0])

# take a subset
x = {'train': digits_x[idxs[:n_train]],
     'test':  digits_x[idxs[n_train:]]}
```

```
y = {'train': digits_y[idxs[:n_train]],
     'test':  digits_y[idxs[n_train:]]}

print('Training samples:', x['train'].shape[0])
print('Test samples:', x['test'].shape[0])
```

We will now implement a function that trains the network. For each epoch, it loops over all minibatches in the training set and updates the network weights. It will then compute the loss and accuracy for the test samples. Finally, it will plot the learning curves.

(a) Read through the code below.

In [ ]:
```
def fit(net, x, y, epochs=25, learning_rate=0.001, mb_size=10):
    # initialize the loss and accuracy history
    loss_hist = {'train': [], 'test': []}
    accuracy_hist = {'train': [], 'test': []}

    for epoch in range(epochs):
        # initialize the loss and accuracy for this epoch
        loss = {'train': 0.0, 'test': 0.0}
        accuracy = {'train': 0.0, 'test': 0.0}

        # first train on training data, then evaluate on the test data
        for phase in ('train', 'test'):
            # compute the number of minibatches
            steps = x[phase].shape[0] // mb_size

            # loop over all minibatches
            for step in range(steps):
                # get the samples for the current minibatch
                x_mb = x[phase][(step * mb_size):((step + 1) * mb_size)]
                y_mb = y[phase][(step * mb_size):((step + 1) * mb_size), Non

                # compute the forward pass through the network
                pred_y, trace = net.forward(x_mb)

                # compute the current loss and accuracy
                loss[phase] += np.mean(bce_loss(y_mb, pred_y))
                accuracy[phase] += np.mean((y_mb > 0.5) == (pred_y > 0.5))

                # only update the network in the training phase
                if phase == 'train':
                    # compute the gradient for the loss
                    dy = bce_loss_grad(y_mb, pred_y)

                    # backpropagate the gradient through the network
                    net.backward(trace, dy)

                    # update the weights
                    net.step(learning_rate)

            # compute the mean loss and accuracy over all minibatches
            loss[phase] = loss[phase] / steps
            accuracy[phase] = accuracy[phase] / steps
```

```
            # add statistics to history
            loss_hist[phase].append(loss[phase])
            accuracy_hist[phase].append(accuracy[phase])

        print('Epoch %3d: loss[train]=%7.4f  accuracy[train]=%7.4f  loss[tes
              (epoch, loss['train'], accuracy['train'], loss['test'], accura

    # plot the learning curves
    plt.figure(figsize=(20, 5))

    plt.subplot(1, 2, 1)
    for phase in loss_hist:
        plt.plot(loss_hist[phase], label=phase)
    plt.title('BCE loss')
    plt.xlabel('Epoch')
    plt.legend()

    plt.subplot(1, 2, 2)
    for phase in accuracy_hist:
        plt.plot(accuracy_hist[phase], label=phase)
    plt.title('Accuracy')
    plt.xlabel('Epoch')
    plt.legend()
```

We will define a two-layer network:

- A linear layer that maps the 64 features of the input to 32 features.
- A ReLU activation function.
- A linear layer that maps the 32 features to the 1 output features.
- A sigmoid activation function that maps the output to [0, 1].

(b) Train the network and inspect the results. Tune the hyperparameters to get a good result. **(1 point)**

In [ ]:
```
# construct network
rng = np.random.default_rng(12345)
net = Net([
        Linear(64, 32, rng=rng),
        ReLU(),
        Linear(32, 1, rng=rng),
        Sigmoid()])

# TODO: tune the hyperparameters
fit(net, x, y,
    epochs = 25,
    learning_rate = 0,
    mb_size = 10)

# Note: add more cells below if you want to keep runs with different hyperpa
```

(c) How did each of the hyperparameters (number of epochs, learning rate, minibatch size) influence your results? How important is it to set each correctly? **(3 points)**

TODO: Your answer here.

(d) Create and train a network with one linear layer followed by a sigmoid activation: `net = Net([Linear(...), Sigmoid()]` **(1 point)**

In [ ]:
```
# TODO: Your code here.
```

(e) Discuss your results. Compare the results of this single-layer network with those of the network you trained before. **(1 point)**

TODO: Your answer here.

(f) Repeat the experiment with a network with two linear layers, followed by a sigmoid activation: `[Linear, Linear, Sigmoid]`. **(1 point)**

In [ ]:
```
# TODO: Your code here.
```

(g) How does the performance of this network compare with the previous networks. Can you explain this result? What is the influence of the activation functions in the network? **(1 point)**

TODO: Your answer here.

(h) One way to improve the performance of a neural network is by increasing the number of layers. Try a deeper network (e.g., a network with four linear layers) to see if this outperforms the previous networks. **(1 point)**

In [ ]:
```
# TODO: Your code here.
```

(i) Discuss your findings. Were you able to obtain a perfect classification? Explain the learning curves. **(1 point)**

TODO: Your answer here.

1.7 Final questions (6 points)

You now have some experience training neural networks. Time for a few final questions.

(a) What is the influence of the learning rate? What happens if the learning rate is too low or too high? **(2 points)**

TODO: Your answer here.

(b) What is the role of the minibatch size in SGD? Explain the downsides of a minibatch size that is too small or too high. **(2 points)**

TODO: Your answer here.

(c) In the linear layer, we initialized the weights w with random values, but we initialized the bias b with zeros. What would happen if the weights w were initialised as zeros? Why is this not a problem for the bias? (2 points)

TODO: Your answer here.

The end

Well done! Please double check the instructions at the top before you submit your results.

This assignment has 45 points.

Version 00f98aa / 2023-09-04