Deep Learning — Assignment 1

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

Names: Luka Mucko, Luca Poli

Group: 46

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 [330…

```
%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:

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

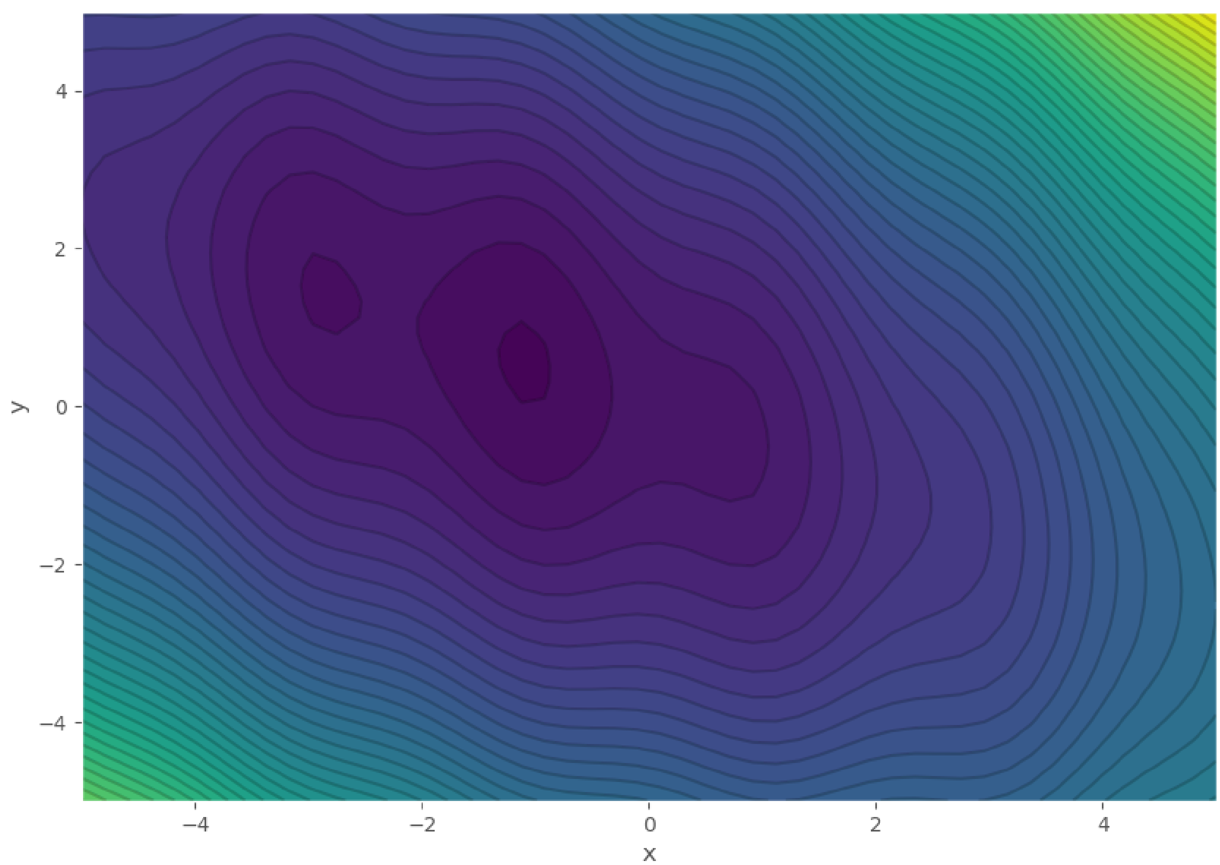
In [331…

```
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 [332…

```
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 [333…

```
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 - step_size*dx # TODO: compute the update
        y = y - step_size*dy # 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 [334…

```
# The following assert statements check that your implementation behaves sensibly
# 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 increasing,
assert abs(optimize_f(3, 2, 0.1, 1)[0] - 3) < 1, "Hint: you are probably taking steps
```

Tune the parameters

We will now try if our optimization method works.

Use this helper function to plot the results:

In [335…

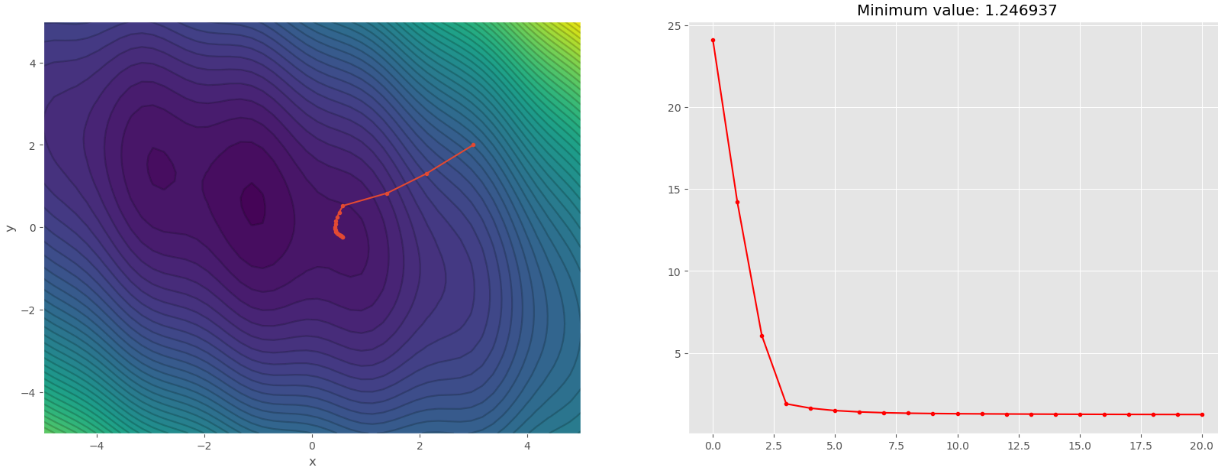
```
# helper function that plots the results of the gradient descent optimization
def plot_gradient_descent_results(x, y, val, x_hist, y_hist):
    # plot the path on the contour plot
    plt.figure(figsize=(20, 7))
    plt.subplot(1, 2, 1)
    plot_f_contours()
    plt.plot(x_hist, y_hist, '.-')

    # 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 [336…

```
results = optimize_f(x=3, y=2, step_size=0.1, steps=20)
plot_gradient_descent_results(*results)
```



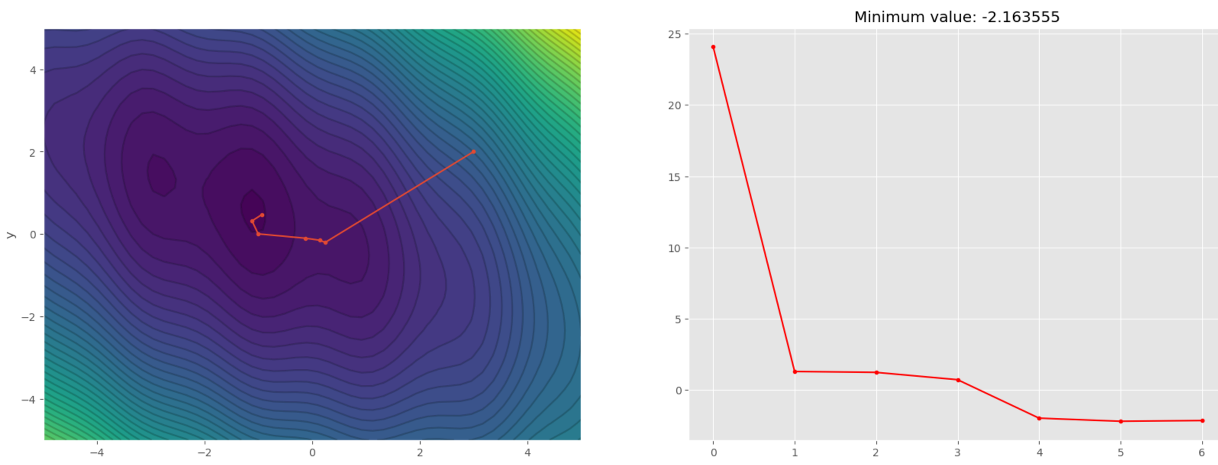
(c) Does it find the minimum of the function? What happens? (1 point)

The algorithm got stuck in the local minimum.

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

In [337…

```
# TODO: tune the parameters to find a better optimum
results = optimize_f(x=3, y=2, step_size=0.315, steps=6)
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)

If the step_size is too small the algorithm will most likely get stuck in a local minimum because the new point is too close to the previous (e.g. step_size=0.1, size=10 in the previous example).

If the step_size is too large the algorithm will "shoot off" to a point that could be higher and then oscilate around the minimum, local or global (e.g. $f(x) = x^2$ and $\eta = 1$ for x=1)

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

step_size=0.315 and steps=6 got pretty close to the global optima but not exactly. There is no guarrantee that with a fixed step_size we could reach the global (or local) optimum exactly.

Unless we are in the vicinity of the global minimum and we implement line search for learning rate, but that is in most cases hard or impossible to do.

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 [338…

```
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)

        eta=step_size * np.power(decay, step)
        # apply the gradient descent updates to x and y
        x = x  - eta*dx# TODO: compute the update including step size decay
        y = y  - eta*dy# 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
```

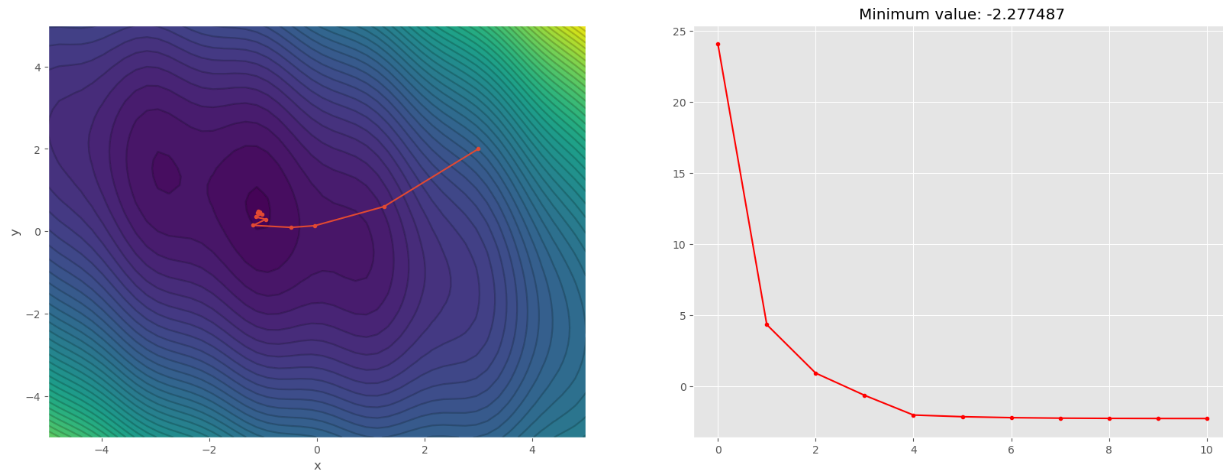
In [339…

```
# The following assert statement checks that your implementation behaves sensibly
_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: step size s
del _trace
```

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

In [340…

```
# TODO: tune the parameters to find the global optimum
results = optimize_f(x=3, y=2, step_size=0.2, steps=10, decay=0.95)
plot_gradient_descent_results(*results)
```



In [341…

```
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)**

$$\frac{\partial \sigma(x)}{\partial x} = \frac{e^{-x}}{(1 + e^{-x})^2} \tag{6}$$

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

In [342…

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

```python
def sigmoid_grad(x):
    return np.exp(-x) / (1 + np.exp(-x))**2

# 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))
```

```
x: [-5.45328  -3.664833  5.947309  3.525093 -2.177809]
sigmoid(x): [0.004264 0.024969 0.997394 0.971393 0.101761]
sigmoid_grad(x): [0.004246 0.024346 0.002599 0.027788 0.091406]
```

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 [343…

```python
# 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

h = 0.001
sigmoid_grad_cdm = (sigmoid(x + h/2) - sigmoid(x - h/2)) / h
print('Numerical', sigmoid_grad_cdm)
```

```
Symbolic  [0.188245 0.219215 0.178901 0.221338 0.238508]
Numerical [0.188245 0.219215 0.178901 0.221338 0.238508]
```

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

With the finite differences we get the same result up to a certain precision. This is because the finite differences are an approximation of the derivative.

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 .

$$\frac{\partial f(x)}{\partial x} = \text{if } x < 0 : 0 \text{ else } 1 \tag{11}$$

(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 [344…

```python
def relu(x):
    return np.maximum(0, x)


def relu_grad(x):
    return np.where(x < 0, 0, 1)

# 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))


h = 0.001
relu_grad_cdm = (relu(x + h/2) - relu(x - h/2)) / h
print('Numerical relu_grad(x):', relu_grad_cdm)
```

```
x: [-5.45328   -3.664833  5.947309  3.525093 -2.177809]
relu(x): [0.        0.        5.947309 3.525093 0.       ]
relu_grad(x): [0 0 1 1 0]
Numerical relu_grad(x): [0. 0. 1. 1. 0.]
```

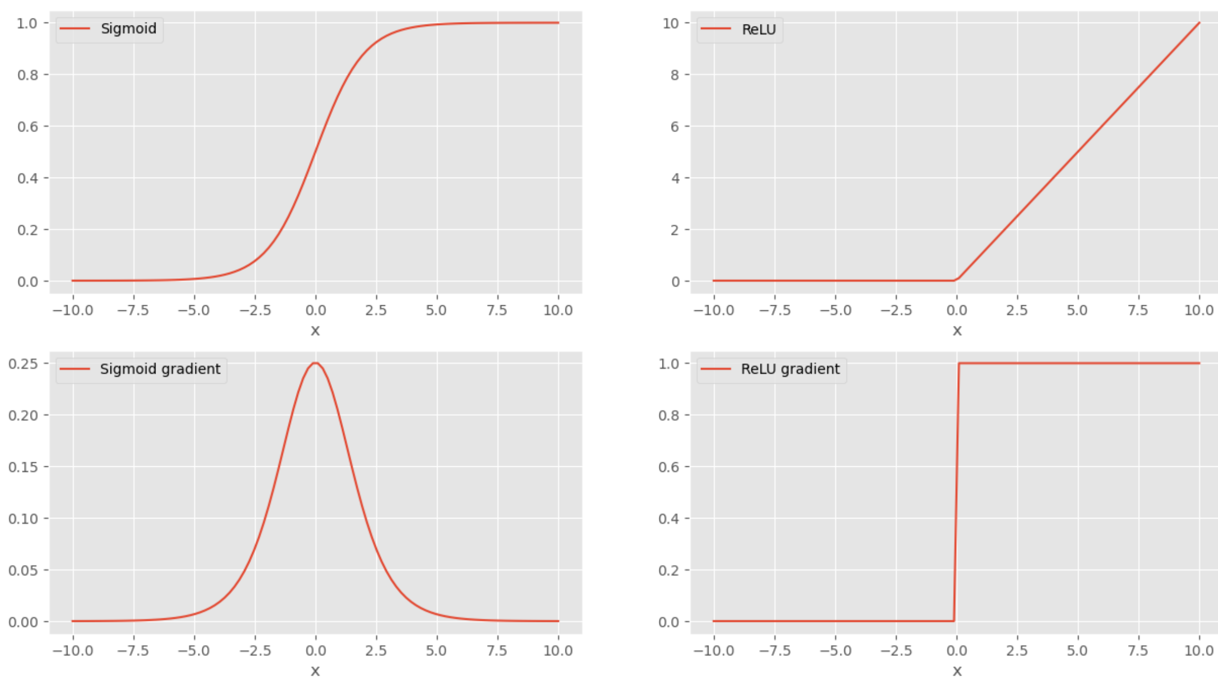
Comparing sigmoid and ReLU

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 [345…

```
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)**

We would recommend the sigmoid function for probabilities, because it is bounded between 0 and 1.

(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)

The ReLu gradient is either 0 or 1, which makes it easier to compute. The sigmoid gradient is more complex.

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)

$$\frac{\partial \mathcal{L}}{\partial \hat{y}} = -(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}) \tag{13}$$

(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 [346…

```
def bce_loss(y, y_hat):
    # TODO: implement the BCE loss
    return -(y*np.log(y_hat) + (1-y)*np.log(1-y_hat))
    raise NotImplementedError

def bce_loss_grad(y, y_hat):
    # TODO: implement the gradient of the BCE loss
    return -(y/y_hat - (1-y)/(1-y_hat))
    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()

def bce_loss_grad_numerical(y, y_hat):
    h=0.001
    return (bce_loss(y,y_hat+h) - bce_loss(y,y_hat-h))/(2*h)

print("bce_loss_grad", bce_loss_grad(y,y_hat))
print("bce_loss_grad_numerical", bce_loss_grad_numerical(y,y_hat))
```

```
print("difference:", np.abs(bce_loss_grad(y,y_hat) - bce_loss_grad_numerical(y,y_hat))
# TODO: compute and compare the symbolic and numerical gradient
```

```
y: [1 0 1 0 0]
y_hat: [0.676255 0.39111  0.332814 0.598309 0.186734]
bceloss(y, y_hat): [0.391186 0.496117 1.100172 0.912072 0.206697]

bce_loss_grad [-1.478733  1.642332 -3.004682  2.489474  1.22961 ]
bce_loss_grad_numerical [-1.478734  1.642333 -3.004691  2.489479  1.229611]
difference: [0.000001 0.000001 0.000009 0.000005 0.000001]
```

Linear layer

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

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

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

(l) Derive the gradients for $\mathbf{y}$ with respect to the input $\mathbf{x}$ and the parameters $\mathbf{W}$ and $\mathbf{b}$: (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.

δ is the Kronecker delta function

$$\frac{\partial y_j}{\partial x_i} = W_{ij} \qquad \frac{\partial y_j}{\partial W_{ik}} = \delta_{jk} x_i \qquad \frac{\partial y_j}{\partial b_k} = \delta_{jk} \tag{15}$$

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

$$\nabla_{\mathbf{x}}\mathcal{L} = \nabla_{\mathbf{y}}\mathcal{L}\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \nabla_{\mathbf{y}}\mathcal{L}^T W^T \tag{16}$$

$$\nabla_{\mathbf{W}}\mathcal{L} = \nabla_{\mathbf{y}}\mathcal{L}\frac{\partial \mathbf{y}}{\partial \mathbf{W}} = x^T \nabla_{\mathbf{y}}\mathcal{L} \tag{17}$$

$$\nabla_{\mathbf{b}}\mathcal{L} = \nabla_{\mathbf{y}}\mathcal{L}\frac{\partial \mathbf{y}}{\partial \mathbf{b}} = \mathbf{1}\nabla_{\mathbf{y}}\mathcal{L}^T = \nabla_{\mathbf{y}}\mathcal{L} \tag{18}$$

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 $\mathbf{x}$, weight vector $\mathbf{w}$ and bias b, compute the output $\hat{y}$:

$$h = \mathbf{x}\mathbf{w}^T + b \tag{19}$$
$$\hat{y} = \sigma(h) \tag{20}$$

1. Compute the BCE loss comparing the prediction $\hat{y}$ with the ground-truth label y.

2. Compute the gradient for the BCE loss and back-propagate this to get $\nabla_{\mathbf{x}}\mathcal{L}$, the gradient of $\mathcal{L}$ w.r.t. $\mathbf{x}$.

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 [347…

```
# 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 = x @ w.T + b
    y_hat = sigmoid(h)
    loss = bce_loss(y, y_hat)

    # TODO: backward: compute grad_y_hat, grad_h, grad_x
    grad_y_hat = bce_loss_grad(y, y_hat)
    grad_h = grad_y_hat * sigmoid_grad(h)
    grad_x = grad_h * w.T

    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"
```

```
Loss 2.3098802440910484
Gradient [ 1.282477 -1.138274  0.784228  0.233444  0.067864]
```

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

In [348…

```
# 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)
```

```
Symbolic gradient
[ 1.177245 -1.044874  0.719879  0.214289  0.062295]
Numerical gradient
[ 1.177245 -1.044874  0.719879  0.214289  0.062295]
```

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 [349…

```
# 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
```

```python
        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):
        y = x @ self.W + self.b
        return y

    def backward(self, x, dy):
        # given dy, compute the gradients for x, W and b
        dx = dy @ self.W.T
        self.dW = x.T @ dy
        self.db = np.sum(dy, axis=0)

        return dx

    def step(self, step_size):
        self.W -= step_size * self.dW
        self.b -= step_size * self.db

    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)
```

```
y: [[ 0.252427  0.428688 -0.081518  0.207314 -0.058535 -0.005779  0.059303]
 [ 0.382911  0.146397 -0.275544 -0.026378 -0.333927 -0.537221 -0.223564]
 [ 0.15955   0.155119 -0.222059  0.428698 -0.231045 -0.345936 -0.119919]]
dx: [[-0.326296 -0.992105  1.657474  0.165888 -0.622481]
 [-0.326296 -0.992105  1.657474  0.165888 -0.622481]
 [-0.326296 -0.992105  1.657474  0.165888 -0.622481]]
```

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

In [350…

```python
# Computes y = 1 / (1 + exp(-x)).
class Sigmoid:
    def forward(self, x):
        return sigmoid(x)
```

```
    def backward(self, x, dy):
        # return the gradient for x given the gradient for y
        dx = sigmoid_grad(x) * dy
        return dx

    def step(self, step_size):
        return

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

```
y: [[0.194063 0.779667 0.295117 0.435567 0.481173]
 [0.322811 0.202977 0.656761 0.589297 0.124242]
 [0.912728 0.72482  0.318779 0.711401 0.385338]]
dx: [[0.156402 0.171786 0.208023 0.245848 0.249646]
 [0.218604 0.161777 0.225426 0.242026 0.108806]
 [0.079656 0.199456 0.217159 0.20531  0.236853]]
```

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

In [351…

```
# Computes y = max(0, x).
class ReLU:
    def forward(self, x):
        return relu(x)

    def backward(self, x, dy):
        dx = relu_grad(x) * dy
        return dx

    def step(self, step_size):
        return

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

```
y: [[0.       0.       5.947309 3.525093 0.      ]
 [0.       1.966175 0.       3.455121 8.836057]
 [0.       8.977623 3.344749 0.       0.      ]]
dx: [[0. 0. 1. 1. 0.]
 [0. 1. 0. 1. 1.]
 [0. 1. 1. 0. 0.]]
```

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 [352…

```
## 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, size=3 * 5
print("err on dx:", err)
assert np.abs(err) < 1e-5, "Error on dx is too large, check your implementation of Lin

# 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, size=5 * 7
print("err on dW:", err)
assert np.abs(err) < 1e-5, "Error on dW is too large, check your implementation of Lin

# 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, size=7))
print("err on db:", err)
assert np.abs(err) < 1e-5, "Error on db is too large, check your implementation of Lir
```

```
err on dx: 8.877935601100038e-07
err on dW: 1.671517959170096e-06
err on db: 0.0
```

In [353…

```
## 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, size=5))
print("err on dx:", err)
assert np.abs(err) < 1e-5, "Error on dx is too large, check your implementation of Sig
```

```
err on dx: 4.823853650098719e-08
```

In [354…

```
## 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, size=5))
print("err on dx:", err)
assert np.abs(err) < 1e-5, "Error on dx is too large, check your implementation of Rel
```

```
err on dx: 0.0
```

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 [355…

```
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 [356…

```
# 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='gray')
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))
```

```
digits_x.shape: (1797, 64)
digits_y.shape: (1797,)
min, max values: -0.8117561971974786 1.847470154168513
labels: [0. 1.]
```

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

In [357…

```
# 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])
```

```
Training samples: 898
Test samples: 899
```

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 [358…

```
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), None]

            # 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[test]=%7.4f
          (epoch, loss['train'], accuracy['train'], loss['test'], accuracy['test']

# 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)

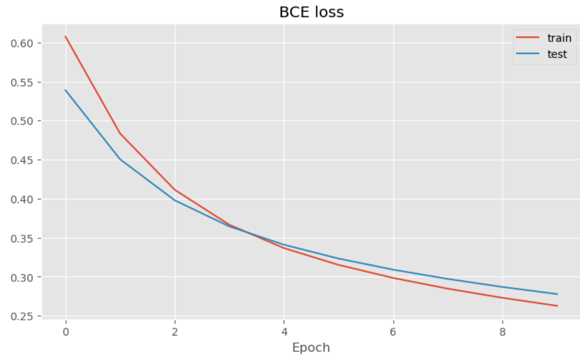
In [359…

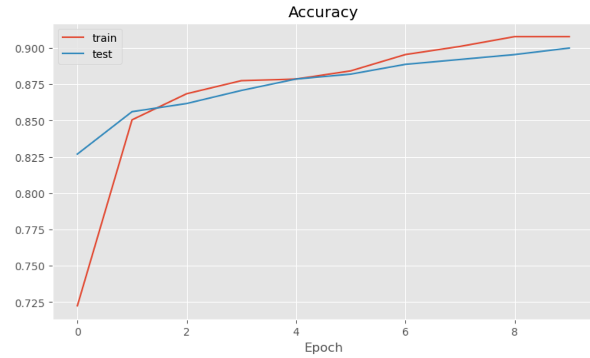
```
# construct network
rng = np.random.default_rng(12345)
net = Net([
        Linear(64, 32, rng=rng),
        ReLU(),
        Linear(32, 1, rng=rng),
        Sigmoid()])

print("less epochs net")
fit(net, x, y,
    epochs = 10,
    learning_rate = 0.001,
    mb_size = 10)

# Note: add more cells below if you want to keep runs with different hyperparameters.
rng = np.random.default_rng(12345)
net = Net([
        Linear(64, 32, rng=rng),
        ReLU(),
        Linear(32, 1, rng=rng),
        Sigmoid()])

print("\nhigher epochs net")
fit(net, x, y,
    epochs = 50,
    learning_rate = 0.001,
    mb_size = 10)
```

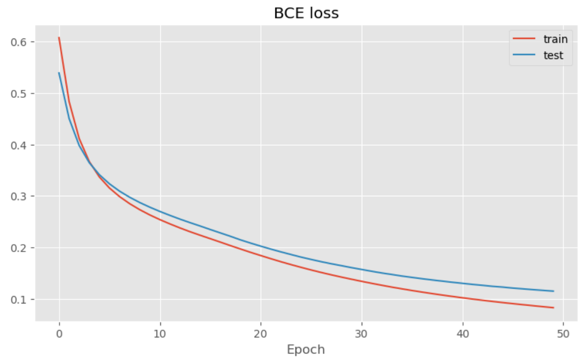
```
less epochs net
Epoch   0: loss[train]= 0.6074  accuracy[train]= 0.7225  loss[test]= 0.5388  accuracy
[test]= 0.8270
Epoch   1: loss[train]= 0.4834  accuracy[train]= 0.8506  loss[test]= 0.4504  accuracy
[test]= 0.8562
Epoch   2: loss[train]= 0.4114  accuracy[train]= 0.8685  loss[test]= 0.3979  accuracy
[test]= 0.8618
Epoch   3: loss[train]= 0.3666  accuracy[train]= 0.8775  loss[test]= 0.3644  accuracy
[test]= 0.8708
Epoch   4: loss[train]= 0.3367  accuracy[train]= 0.8787  loss[test]= 0.3410  accuracy
[test]= 0.8787
Epoch   5: loss[train]= 0.3152  accuracy[train]= 0.8843  loss[test]= 0.3233  accuracy
[test]= 0.8820
Epoch   6: loss[train]= 0.2985  accuracy[train]= 0.8955  loss[test]= 0.3091  accuracy
[test]= 0.8888
Epoch   7: loss[train]= 0.2847  accuracy[train]= 0.9011  loss[test]= 0.2972  accuracy
[test]= 0.8921
Epoch   8: loss[train]= 0.2730  accuracy[train]= 0.9079  loss[test]= 0.2869  accuracy
[test]= 0.8955
Epoch   9: loss[train]= 0.2628  accuracy[train]= 0.9079  loss[test]= 0.2779  accuracy
[test]= 0.9000

higher epochs net
Epoch   0: loss[train]= 0.6074  accuracy[train]= 0.7225  loss[test]= 0.5388  accuracy
[test]= 0.8270
Epoch   1: loss[train]= 0.4834  accuracy[train]= 0.8506  loss[test]= 0.4504  accuracy
[test]= 0.8562
Epoch   2: loss[train]= 0.4114  accuracy[train]= 0.8685  loss[test]= 0.3979  accuracy
[test]= 0.8618
Epoch   3: loss[train]= 0.3666  accuracy[train]= 0.8775  loss[test]= 0.3644  accuracy
[test]= 0.8708
Epoch   4: loss[train]= 0.3367  accuracy[train]= 0.8787  loss[test]= 0.3410  accuracy
[test]= 0.8787
Epoch   5: loss[train]= 0.3152  accuracy[train]= 0.8843  loss[test]= 0.3233  accuracy
[test]= 0.8820
Epoch   6: loss[train]= 0.2985  accuracy[train]= 0.8955  loss[test]= 0.3091  accuracy
[test]= 0.8888
Epoch   7: loss[train]= 0.2847  accuracy[train]= 0.9011  loss[test]= 0.2972  accuracy
[test]= 0.8921
Epoch   8: loss[train]= 0.2730  accuracy[train]= 0.9079  loss[test]= 0.2869  accuracy
[test]= 0.8955
Epoch   9: loss[train]= 0.2628  accuracy[train]= 0.9079  loss[test]= 0.2779  accuracy
[test]= 0.9000
Epoch  10: loss[train]= 0.2536  accuracy[train]= 0.9112  loss[test]= 0.2696  accuracy
[test]= 0.9022
Epoch  11: loss[train]= 0.2453  accuracy[train]= 0.9135  loss[test]= 0.2619  accuracy
[test]= 0.9056
Epoch  12: loss[train]= 0.2374  accuracy[train]= 0.9146  loss[test]= 0.2545  accuracy
[test]= 0.9124
Epoch  13: loss[train]= 0.2300  accuracy[train]= 0.9180  loss[test]= 0.2477  accuracy
[test]= 0.9146
Epoch  14: loss[train]= 0.2232  accuracy[train]= 0.9180  loss[test]= 0.2412  accuracy
[test]= 0.9169
Epoch  15: loss[train]= 0.2166  accuracy[train]= 0.9247  loss[test]= 0.2344  accuracy
[test]= 0.9180
Epoch  16: loss[train]= 0.2099  accuracy[train]= 0.9258  loss[test]= 0.2278  accuracy
[test]= 0.9202
Epoch  17: loss[train]= 0.2030  accuracy[train]= 0.9270  loss[test]= 0.2212  accuracy
[test]= 0.9247
Epoch  18: loss[train]= 0.1965  accuracy[train]= 0.9337  loss[test]= 0.2142  accuracy
```

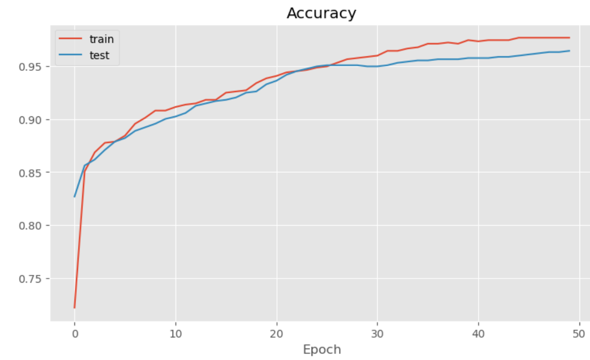
```
[test]= 0.9258
Epoch  19: loss[train]= 0.1899  accuracy[train]= 0.9382  loss[test]= 0.2080  accuracy
[test]= 0.9326
Epoch  20: loss[train]= 0.1838  accuracy[train]= 0.9404  loss[test]= 0.2020  accuracy
[test]= 0.9360
Epoch  21: loss[train]= 0.1776  accuracy[train]= 0.9438  loss[test]= 0.1963  accuracy
[test]= 0.9416
Epoch  22: loss[train]= 0.1717  accuracy[train]= 0.9449  loss[test]= 0.1909  accuracy
[test]= 0.9449
Epoch  23: loss[train]= 0.1662  accuracy[train]= 0.9461  loss[test]= 0.1858  accuracy
[test]= 0.9472
Epoch  24: loss[train]= 0.1608  accuracy[train]= 0.9483  loss[test]= 0.1807  accuracy
[test]= 0.9494
Epoch  25: loss[train]= 0.1557  accuracy[train]= 0.9494  loss[test]= 0.1762  accuracy
[test]= 0.9506
Epoch  26: loss[train]= 0.1508  accuracy[train]= 0.9528  loss[test]= 0.1718  accuracy
[test]= 0.9506
Epoch  27: loss[train]= 0.1462  accuracy[train]= 0.9562  loss[test]= 0.1678  accuracy
[test]= 0.9506
Epoch  28: loss[train]= 0.1418  accuracy[train]= 0.9573  loss[test]= 0.1641  accuracy
[test]= 0.9506
Epoch  29: loss[train]= 0.1377  accuracy[train]= 0.9584  loss[test]= 0.1602  accuracy
[test]= 0.9494
Epoch  30: loss[train]= 0.1336  accuracy[train]= 0.9596  loss[test]= 0.1567  accuracy
[test]= 0.9494
Epoch  31: loss[train]= 0.1297  accuracy[train]= 0.9640  loss[test]= 0.1532  accuracy
[test]= 0.9506
Epoch  32: loss[train]= 0.1260  accuracy[train]= 0.9640  loss[test]= 0.1498  accuracy
[test]= 0.9528
Epoch  33: loss[train]= 0.1223  accuracy[train]= 0.9663  loss[test]= 0.1469  accuracy
[test]= 0.9539
Epoch  34: loss[train]= 0.1190  accuracy[train]= 0.9674  loss[test]= 0.1439  accuracy
[test]= 0.9551
Epoch  35: loss[train]= 0.1157  accuracy[train]= 0.9708  loss[test]= 0.1412  accuracy
[test]= 0.9551
Epoch  36: loss[train]= 0.1127  accuracy[train]= 0.9708  loss[test]= 0.1387  accuracy
[test]= 0.9562
Epoch  37: loss[train]= 0.1097  accuracy[train]= 0.9719  loss[test]= 0.1363  accuracy
[test]= 0.9562
Epoch  38: loss[train]= 0.1069  accuracy[train]= 0.9708  loss[test]= 0.1340  accuracy
[test]= 0.9562
Epoch  39: loss[train]= 0.1042  accuracy[train]= 0.9742  loss[test]= 0.1318  accuracy
[test]= 0.9573
Epoch  40: loss[train]= 0.1016  accuracy[train]= 0.9730  loss[test]= 0.1297  accuracy
[test]= 0.9573
Epoch  41: loss[train]= 0.0991  accuracy[train]= 0.9742  loss[test]= 0.1276  accuracy
[test]= 0.9573
Epoch  42: loss[train]= 0.0967  accuracy[train]= 0.9742  loss[test]= 0.1258  accuracy
[test]= 0.9584
Epoch  43: loss[train]= 0.0945  accuracy[train]= 0.9742  loss[test]= 0.1238  accuracy
[test]= 0.9584
Epoch  44: loss[train]= 0.0923  accuracy[train]= 0.9764  loss[test]= 0.1224  accuracy
[test]= 0.9596
Epoch  45: loss[train]= 0.0902  accuracy[train]= 0.9764  loss[test]= 0.1204  accuracy
[test]= 0.9607
Epoch  46: loss[train]= 0.0881  accuracy[train]= 0.9764  loss[test]= 0.1190  accuracy
[test]= 0.9618
Epoch  47: loss[train]= 0.0862  accuracy[train]= 0.9764  loss[test]= 0.1174  accuracy
[test]= 0.9629
Epoch  48: loss[train]= 0.0842  accuracy[train]= 0.9764  loss[test]= 0.1160  accuracy
```

```
[test]= 0.9629
Epoch  49: loss[train]= 0.0824  accuracy[train]= 0.9764  loss[test]= 0.1145  accuracy
[test]= 0.9640
```









In [360…

```
net = Net([
        Linear(64, 32, rng=rng),
        ReLU(),
        Linear(32, 1, rng=rng),
        Sigmoid()])

print("lower learning rate net")
fit(net, x, y,
    epochs = 100,
    learning_rate = 0.01,
    mb_size = 10)

net = Net([
        Linear(64, 32, rng=rng),
        ReLU(),
        Linear(32, 1, rng=rng),
        Sigmoid()])

print("\nhigher learning rate net")
fit(net, x, y,
    epochs = 100,
    learning_rate = 0.001,
    mb_size = 10)
```

```
lower learning rate net
Epoch   0: loss[train]= 0.4336  accuracy[train]= 0.8034  loss[test]= 0.2922  accuracy
[test]= 0.8921
Epoch   1: loss[train]= 0.2477  accuracy[train]= 0.9146  loss[test]= 0.2061  accuracy
[test]= 0.9281
Epoch   2: loss[train]= 0.1800  accuracy[train]= 0.9292  loss[test]= 0.1540  accuracy
[test]= 0.9562
Epoch   3: loss[train]= 0.1376  accuracy[train]= 0.9461  loss[test]= 0.1250  accuracy
[test]= 0.9640
Epoch   4: loss[train]= 0.1090  accuracy[train]= 0.9618  loss[test]= 0.1109  accuracy
[test]= 0.9652
Epoch   5: loss[train]= 0.0890  accuracy[train]= 0.9730  loss[test]= 0.1012  accuracy
[test]= 0.9674
Epoch   6: loss[train]= 0.0720  accuracy[train]= 0.9787  loss[test]= 0.0944  accuracy
[test]= 0.9663
Epoch   7: loss[train]= 0.0605  accuracy[train]= 0.9865  loss[test]= 0.0902  accuracy
[test]= 0.9674
Epoch   8: loss[train]= 0.0514  accuracy[train]= 0.9888  loss[test]= 0.0865  accuracy
[test]= 0.9697
Epoch   9: loss[train]= 0.0447  accuracy[train]= 0.9899  loss[test]= 0.0836  accuracy
[test]= 0.9708
Epoch  10: loss[train]= 0.0385  accuracy[train]= 0.9921  loss[test]= 0.0804  accuracy
[test]= 0.9719
Epoch  11: loss[train]= 0.0336  accuracy[train]= 0.9933  loss[test]= 0.0779  accuracy
[test]= 0.9730
Epoch  12: loss[train]= 0.0292  accuracy[train]= 0.9933  loss[test]= 0.0760  accuracy
[test]= 0.9753
Epoch  13: loss[train]= 0.0251  accuracy[train]= 0.9944  loss[test]= 0.0739  accuracy
[test]= 0.9764
Epoch  14: loss[train]= 0.0221  accuracy[train]= 0.9955  loss[test]= 0.0741  accuracy
[test]= 0.9753
Epoch  15: loss[train]= 0.0194  accuracy[train]= 0.9978  loss[test]= 0.0722  accuracy
[test]= 0.9764
Epoch  16: loss[train]= 0.0171  accuracy[train]= 0.9978  loss[test]= 0.0718  accuracy
[test]= 0.9764
Epoch  17: loss[train]= 0.0150  accuracy[train]= 0.9989  loss[test]= 0.0713  accuracy
[test]= 0.9764
Epoch  18: loss[train]= 0.0135  accuracy[train]= 0.9989  loss[test]= 0.0702  accuracy
[test]= 0.9775
Epoch  19: loss[train]= 0.0120  accuracy[train]= 1.0000  loss[test]= 0.0707  accuracy
[test]= 0.9775
Epoch  20: loss[train]= 0.0109  accuracy[train]= 1.0000  loss[test]= 0.0690  accuracy
[test]= 0.9775
Epoch  21: loss[train]= 0.0099  accuracy[train]= 1.0000  loss[test]= 0.0689  accuracy
[test]= 0.9775
Epoch  22: loss[train]= 0.0090  accuracy[train]= 1.0000  loss[test]= 0.0690  accuracy
[test]= 0.9775
Epoch  23: loss[train]= 0.0083  accuracy[train]= 1.0000  loss[test]= 0.0685  accuracy
[test]= 0.9775
Epoch  24: loss[train]= 0.0076  accuracy[train]= 1.0000  loss[test]= 0.0683  accuracy
[test]= 0.9775
Epoch  25: loss[train]= 0.0071  accuracy[train]= 1.0000  loss[test]= 0.0684  accuracy
[test]= 0.9775
Epoch  26: loss[train]= 0.0066  accuracy[train]= 1.0000  loss[test]= 0.0677  accuracy
[test]= 0.9775
Epoch  27: loss[train]= 0.0061  accuracy[train]= 1.0000  loss[test]= 0.0678  accuracy
[test]= 0.9787
Epoch  28: loss[train]= 0.0058  accuracy[train]= 1.0000  loss[test]= 0.0674  accuracy
[test]= 0.9787
Epoch  29: loss[train]= 0.0054  accuracy[train]= 1.0000  loss[test]= 0.0674  accuracy
```

```
[test]= 0.9787
Epoch  30: loss[train]= 0.0051  accuracy[train]= 1.0000  loss[test]= 0.0672  accuracy
[test]= 0.9798
Epoch  31: loss[train]= 0.0048  accuracy[train]= 1.0000  loss[test]= 0.0671  accuracy
[test]= 0.9787
Epoch  32: loss[train]= 0.0045  accuracy[train]= 1.0000  loss[test]= 0.0673  accuracy
[test]= 0.9798
Epoch  33: loss[train]= 0.0043  accuracy[train]= 1.0000  loss[test]= 0.0671  accuracy
[test]= 0.9798
Epoch  34: loss[train]= 0.0041  accuracy[train]= 1.0000  loss[test]= 0.0672  accuracy
[test]= 0.9798
Epoch  35: loss[train]= 0.0039  accuracy[train]= 1.0000  loss[test]= 0.0675  accuracy
[test]= 0.9798
Epoch  36: loss[train]= 0.0037  accuracy[train]= 1.0000  loss[test]= 0.0672  accuracy
[test]= 0.9809
Epoch  37: loss[train]= 0.0036  accuracy[train]= 1.0000  loss[test]= 0.0676  accuracy
[test]= 0.9787
Epoch  38: loss[train]= 0.0034  accuracy[train]= 1.0000  loss[test]= 0.0675  accuracy
[test]= 0.9809
Epoch  39: loss[train]= 0.0033  accuracy[train]= 1.0000  loss[test]= 0.0675  accuracy
[test]= 0.9787
Epoch  40: loss[train]= 0.0032  accuracy[train]= 1.0000  loss[test]= 0.0678  accuracy
[test]= 0.9809
Epoch  41: loss[train]= 0.0031  accuracy[train]= 1.0000  loss[test]= 0.0679  accuracy
[test]= 0.9809
Epoch  42: loss[train]= 0.0029  accuracy[train]= 1.0000  loss[test]= 0.0681  accuracy
[test]= 0.9798
Epoch  43: loss[train]= 0.0028  accuracy[train]= 1.0000  loss[test]= 0.0680  accuracy
[test]= 0.9798
Epoch  44: loss[train]= 0.0027  accuracy[train]= 1.0000  loss[test]= 0.0680  accuracy
[test]= 0.9787
Epoch  45: loss[train]= 0.0026  accuracy[train]= 1.0000  loss[test]= 0.0684  accuracy
[test]= 0.9798
Epoch  46: loss[train]= 0.0026  accuracy[train]= 1.0000  loss[test]= 0.0684  accuracy
[test]= 0.9809
Epoch  47: loss[train]= 0.0025  accuracy[train]= 1.0000  loss[test]= 0.0685  accuracy
[test]= 0.9798
Epoch  48: loss[train]= 0.0024  accuracy[train]= 1.0000  loss[test]= 0.0687  accuracy
[test]= 0.9798
Epoch  49: loss[train]= 0.0023  accuracy[train]= 1.0000  loss[test]= 0.0688  accuracy
[test]= 0.9787
Epoch  50: loss[train]= 0.0023  accuracy[train]= 1.0000  loss[test]= 0.0689  accuracy
[test]= 0.9798
Epoch  51: loss[train]= 0.0022  accuracy[train]= 1.0000  loss[test]= 0.0690  accuracy
[test]= 0.9798
Epoch  52: loss[train]= 0.0021  accuracy[train]= 1.0000  loss[test]= 0.0691  accuracy
[test]= 0.9798
Epoch  53: loss[train]= 0.0021  accuracy[train]= 1.0000  loss[test]= 0.0691  accuracy
[test]= 0.9798
Epoch  54: loss[train]= 0.0020  accuracy[train]= 1.0000  loss[test]= 0.0692  accuracy
[test]= 0.9787
Epoch  55: loss[train]= 0.0020  accuracy[train]= 1.0000  loss[test]= 0.0695  accuracy
[test]= 0.9798
Epoch  56: loss[train]= 0.0019  accuracy[train]= 1.0000  loss[test]= 0.0694  accuracy
[test]= 0.9798
Epoch  57: loss[train]= 0.0019  accuracy[train]= 1.0000  loss[test]= 0.0697  accuracy
[test]= 0.9798
Epoch  58: loss[train]= 0.0018  accuracy[train]= 1.0000  loss[test]= 0.0697  accuracy
[test]= 0.9798
Epoch  59: loss[train]= 0.0018  accuracy[train]= 1.0000  loss[test]= 0.0700  accuracy
```

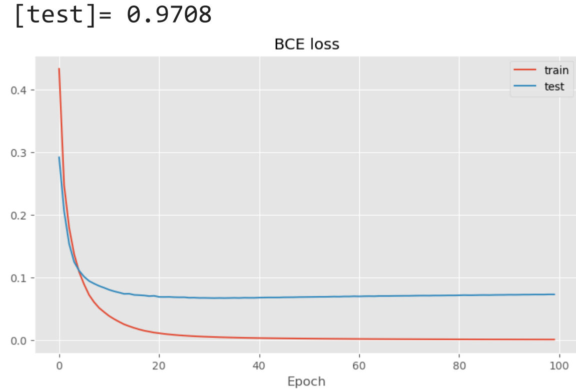
```
[test]= 0.9798
Epoch  60: loss[train]= 0.0017  accuracy[train]= 1.0000  loss[test]= 0.0698  accuracy
[test]= 0.9809
Epoch  61: loss[train]= 0.0017  accuracy[train]= 1.0000  loss[test]= 0.0701  accuracy
[test]= 0.9809
Epoch  62: loss[train]= 0.0017  accuracy[train]= 1.0000  loss[test]= 0.0702  accuracy
[test]= 0.9809
Epoch  63: loss[train]= 0.0016  accuracy[train]= 1.0000  loss[test]= 0.0701  accuracy
[test]= 0.9809
Epoch  64: loss[train]= 0.0016  accuracy[train]= 1.0000  loss[test]= 0.0704  accuracy
[test]= 0.9809
Epoch  65: loss[train]= 0.0016  accuracy[train]= 1.0000  loss[test]= 0.0704  accuracy
[test]= 0.9820
Epoch  66: loss[train]= 0.0015  accuracy[train]= 1.0000  loss[test]= 0.0704  accuracy
[test]= 0.9809
Epoch  67: loss[train]= 0.0015  accuracy[train]= 1.0000  loss[test]= 0.0705  accuracy
[test]= 0.9820
Epoch  68: loss[train]= 0.0015  accuracy[train]= 1.0000  loss[test]= 0.0706  accuracy
[test]= 0.9820
Epoch  69: loss[train]= 0.0014  accuracy[train]= 1.0000  loss[test]= 0.0707  accuracy
[test]= 0.9820
Epoch  70: loss[train]= 0.0014  accuracy[train]= 1.0000  loss[test]= 0.0707  accuracy
[test]= 0.9820
Epoch  71: loss[train]= 0.0014  accuracy[train]= 1.0000  loss[test]= 0.0709  accuracy
[test]= 0.9820
Epoch  72: loss[train]= 0.0013  accuracy[train]= 1.0000  loss[test]= 0.0710  accuracy
[test]= 0.9820
Epoch  73: loss[train]= 0.0013  accuracy[train]= 1.0000  loss[test]= 0.0711  accuracy
[test]= 0.9820
Epoch  74: loss[train]= 0.0013  accuracy[train]= 1.0000  loss[test]= 0.0710  accuracy
[test]= 0.9820
Epoch  75: loss[train]= 0.0013  accuracy[train]= 1.0000  loss[test]= 0.0712  accuracy
[test]= 0.9820
Epoch  76: loss[train]= 0.0013  accuracy[train]= 1.0000  loss[test]= 0.0712  accuracy
[test]= 0.9820
Epoch  77: loss[train]= 0.0012  accuracy[train]= 1.0000  loss[test]= 0.0713  accuracy
[test]= 0.9820
Epoch  78: loss[train]= 0.0012  accuracy[train]= 1.0000  loss[test]= 0.0715  accuracy
[test]= 0.9820
Epoch  79: loss[train]= 0.0012  accuracy[train]= 1.0000  loss[test]= 0.0714  accuracy
[test]= 0.9820
Epoch  80: loss[train]= 0.0012  accuracy[train]= 1.0000  loss[test]= 0.0716  accuracy
[test]= 0.9820
Epoch  81: loss[train]= 0.0012  accuracy[train]= 1.0000  loss[test]= 0.0719  accuracy
[test]= 0.9820
Epoch  82: loss[train]= 0.0011  accuracy[train]= 1.0000  loss[test]= 0.0717  accuracy
[test]= 0.9820
Epoch  83: loss[train]= 0.0011  accuracy[train]= 1.0000  loss[test]= 0.0718  accuracy
[test]= 0.9820
Epoch  84: loss[train]= 0.0011  accuracy[train]= 1.0000  loss[test]= 0.0719  accuracy
[test]= 0.9820
Epoch  85: loss[train]= 0.0011  accuracy[train]= 1.0000  loss[test]= 0.0720  accuracy
[test]= 0.9820
Epoch  86: loss[train]= 0.0011  accuracy[train]= 1.0000  loss[test]= 0.0719  accuracy
[test]= 0.9820
Epoch  87: loss[train]= 0.0010  accuracy[train]= 1.0000  loss[test]= 0.0721  accuracy
[test]= 0.9820
Epoch  88: loss[train]= 0.0010  accuracy[train]= 1.0000  loss[test]= 0.0722  accuracy
[test]= 0.9820
Epoch  89: loss[train]= 0.0010  accuracy[train]= 1.0000  loss[test]= 0.0723  accuracy
```

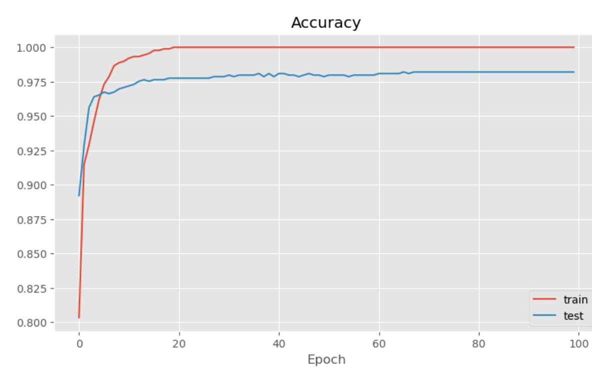
```
[test]= 0.9820
Epoch  90: loss[train]= 0.0010  accuracy[train]= 1.0000  loss[test]= 0.0724  accuracy
[test]= 0.9820
Epoch  91: loss[train]= 0.0010  accuracy[train]= 1.0000  loss[test]= 0.0724  accuracy
[test]= 0.9820
Epoch  92: loss[train]= 0.0010  accuracy[train]= 1.0000  loss[test]= 0.0725  accuracy
[test]= 0.9820
Epoch  93: loss[train]= 0.0010  accuracy[train]= 1.0000  loss[test]= 0.0726  accuracy
[test]= 0.9820
Epoch  94: loss[train]= 0.0009  accuracy[train]= 1.0000  loss[test]= 0.0726  accuracy
[test]= 0.9820
Epoch  95: loss[train]= 0.0009  accuracy[train]= 1.0000  loss[test]= 0.0728  accuracy
[test]= 0.9820
Epoch  96: loss[train]= 0.0009  accuracy[train]= 1.0000  loss[test]= 0.0728  accuracy
[test]= 0.9820
Epoch  97: loss[train]= 0.0009  accuracy[train]= 1.0000  loss[test]= 0.0728  accuracy
[test]= 0.9820
Epoch  98: loss[train]= 0.0009  accuracy[train]= 1.0000  loss[test]= 0.0730  accuracy
[test]= 0.9820
Epoch  99: loss[train]= 0.0009  accuracy[train]= 1.0000  loss[test]= 0.0729  accuracy
[test]= 0.9820

higher learning rate net
Epoch   0: loss[train]= 0.6324  accuracy[train]= 0.7225  loss[test]= 0.5750  accuracy
[test]= 0.8247
Epoch   1: loss[train]= 0.5246  accuracy[train]= 0.8438  loss[test]= 0.4848  accuracy
[test]= 0.8393
Epoch   2: loss[train]= 0.4441  accuracy[train]= 0.8674  loss[test]= 0.4218  accuracy
[test]= 0.8506
Epoch   3: loss[train]= 0.3886  accuracy[train]= 0.8809  loss[test]= 0.3791  accuracy
[test]= 0.8652
Epoch   4: loss[train]= 0.3505  accuracy[train]= 0.8921  loss[test]= 0.3489  accuracy
[test]= 0.8798
Epoch   5: loss[train]= 0.3230  accuracy[train]= 0.8989  loss[test]= 0.3264  accuracy
[test]= 0.8831
Epoch   6: loss[train]= 0.3021  accuracy[train]= 0.9022  loss[test]= 0.3090  accuracy
[test]= 0.8888
Epoch   7: loss[train]= 0.2854  accuracy[train]= 0.9067  loss[test]= 0.2946  accuracy
[test]= 0.8910
Epoch   8: loss[train]= 0.2716  accuracy[train]= 0.9135  loss[test]= 0.2825  accuracy
[test]= 0.8978
Epoch   9: loss[train]= 0.2596  accuracy[train]= 0.9180  loss[test]= 0.2718  accuracy
[test]= 0.9034
Epoch  10: loss[train]= 0.2491  accuracy[train]= 0.9236  loss[test]= 0.2623  accuracy
[test]= 0.9056
Epoch  11: loss[train]= 0.2397  accuracy[train]= 0.9247  loss[test]= 0.2537  accuracy
[test]= 0.9101
Epoch  12: loss[train]= 0.2310  accuracy[train]= 0.9258  loss[test]= 0.2456  accuracy
[test]= 0.9180
Epoch  13: loss[train]= 0.2230  accuracy[train]= 0.9270  loss[test]= 0.2383  accuracy
[test]= 0.9202
Epoch  14: loss[train]= 0.2155  accuracy[train]= 0.9315  loss[test]= 0.2314  accuracy
[test]= 0.9225
Epoch  15: loss[train]= 0.2084  accuracy[train]= 0.9303  loss[test]= 0.2247  accuracy
[test]= 0.9258
Epoch  16: loss[train]= 0.2017  accuracy[train]= 0.9315  loss[test]= 0.2187  accuracy
[test]= 0.9281
Epoch  17: loss[train]= 0.1952  accuracy[train]= 0.9360  loss[test]= 0.2130  accuracy
[test]= 0.9337
Epoch  18: loss[train]= 0.1891  accuracy[train]= 0.9416  loss[test]= 0.2074  accuracy
```

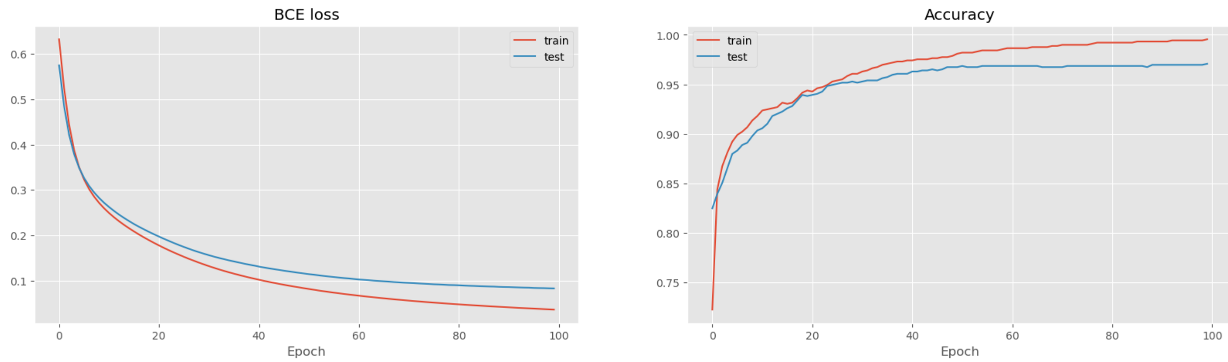
```
[test]= 0.9393
Epoch  19: loss[train]= 0.1832  accuracy[train]= 0.9438  loss[test]= 0.2022  accuracy
[test]= 0.9382
Epoch  20: loss[train]= 0.1775  accuracy[train]= 0.9427  loss[test]= 0.1971  accuracy
[test]= 0.9393
Epoch  21: loss[train]= 0.1720  accuracy[train]= 0.9461  loss[test]= 0.1922  accuracy
[test]= 0.9404
Epoch  22: loss[train]= 0.1669  accuracy[train]= 0.9472  loss[test]= 0.1875  accuracy
[test]= 0.9427
Epoch  23: loss[train]= 0.1619  accuracy[train]= 0.9494  loss[test]= 0.1828  accuracy
[test]= 0.9483
Epoch  24: loss[train]= 0.1571  accuracy[train]= 0.9528  loss[test]= 0.1784  accuracy
[test]= 0.9494
Epoch  25: loss[train]= 0.1524  accuracy[train]= 0.9539  loss[test]= 0.1742  accuracy
[test]= 0.9506
Epoch  26: loss[train]= 0.1479  accuracy[train]= 0.9551  loss[test]= 0.1701  accuracy
[test]= 0.9517
Epoch  27: loss[train]= 0.1437  accuracy[train]= 0.9584  loss[test]= 0.1662  accuracy
[test]= 0.9517
Epoch  28: loss[train]= 0.1396  accuracy[train]= 0.9607  loss[test]= 0.1627  accuracy
[test]= 0.9528
Epoch  29: loss[train]= 0.1356  accuracy[train]= 0.9607  loss[test]= 0.1592  accuracy
[test]= 0.9517
Epoch  30: loss[train]= 0.1318  accuracy[train]= 0.9629  loss[test]= 0.1560  accuracy
[test]= 0.9528
Epoch  31: loss[train]= 0.1282  accuracy[train]= 0.9640  loss[test]= 0.1529  accuracy
[test]= 0.9539
Epoch  32: loss[train]= 0.1247  accuracy[train]= 0.9663  loss[test]= 0.1500  accuracy
[test]= 0.9539
Epoch  33: loss[train]= 0.1214  accuracy[train]= 0.9674  loss[test]= 0.1474  accuracy
[test]= 0.9539
Epoch  34: loss[train]= 0.1183  accuracy[train]= 0.9697  loss[test]= 0.1446  accuracy
[test]= 0.9562
Epoch  35: loss[train]= 0.1153  accuracy[train]= 0.9708  loss[test]= 0.1423  accuracy
[test]= 0.9573
Epoch  36: loss[train]= 0.1124  accuracy[train]= 0.9719  loss[test]= 0.1399  accuracy
[test]= 0.9596
Epoch  37: loss[train]= 0.1096  accuracy[train]= 0.9730  loss[test]= 0.1375  accuracy
[test]= 0.9607
Epoch  38: loss[train]= 0.1069  accuracy[train]= 0.9730  loss[test]= 0.1352  accuracy
[test]= 0.9607
Epoch  39: loss[train]= 0.1044  accuracy[train]= 0.9742  loss[test]= 0.1333  accuracy
[test]= 0.9607
Epoch  40: loss[train]= 0.1020  accuracy[train]= 0.9742  loss[test]= 0.1309  accuracy
[test]= 0.9629
Epoch  41: loss[train]= 0.0995  accuracy[train]= 0.9753  loss[test]= 0.1289  accuracy
[test]= 0.9629
Epoch  42: loss[train]= 0.0972  accuracy[train]= 0.9753  loss[test]= 0.1271  accuracy
[test]= 0.9640
Epoch  43: loss[train]= 0.0950  accuracy[train]= 0.9753  loss[test]= 0.1253  accuracy
[test]= 0.9640
Epoch  44: loss[train]= 0.0929  accuracy[train]= 0.9764  loss[test]= 0.1236  accuracy
[test]= 0.9652
Epoch  45: loss[train]= 0.0908  accuracy[train]= 0.9764  loss[test]= 0.1219  accuracy
[test]= 0.9640
Epoch  46: loss[train]= 0.0888  accuracy[train]= 0.9775  loss[test]= 0.1203  accuracy
[test]= 0.9652
Epoch  47: loss[train]= 0.0869  accuracy[train]= 0.9775  loss[test]= 0.1188  accuracy
[test]= 0.9674
Epoch  48: loss[train]= 0.0851  accuracy[train]= 0.9787  loss[test]= 0.1172  accuracy
```

```
[test]= 0.9674
Epoch  49: loss[train]= 0.0832  accuracy[train]= 0.9809  loss[test]= 0.1157  accuracy
[test]= 0.9674
Epoch  50: loss[train]= 0.0815  accuracy[train]= 0.9820  loss[test]= 0.1143  accuracy
[test]= 0.9685
Epoch  51: loss[train]= 0.0798  accuracy[train]= 0.9820  loss[test]= 0.1130  accuracy
[test]= 0.9674
Epoch  52: loss[train]= 0.0781  accuracy[train]= 0.9820  loss[test]= 0.1117  accuracy
[test]= 0.9674
Epoch  53: loss[train]= 0.0765  accuracy[train]= 0.9831  loss[test]= 0.1102  accuracy
[test]= 0.9674
Epoch  54: loss[train]= 0.0750  accuracy[train]= 0.9843  loss[test]= 0.1090  accuracy
[test]= 0.9685
Epoch  55: loss[train]= 0.0735  accuracy[train]= 0.9843  loss[test]= 0.1079  accuracy
[test]= 0.9685
Epoch  56: loss[train]= 0.0720  accuracy[train]= 0.9843  loss[test]= 0.1067  accuracy
[test]= 0.9685
Epoch  57: loss[train]= 0.0706  accuracy[train]= 0.9843  loss[test]= 0.1056  accuracy
[test]= 0.9685
Epoch  58: loss[train]= 0.0693  accuracy[train]= 0.9854  loss[test]= 0.1048  accuracy
[test]= 0.9685
Epoch  59: loss[train]= 0.0680  accuracy[train]= 0.9865  loss[test]= 0.1036  accuracy
[test]= 0.9685
Epoch  60: loss[train]= 0.0667  accuracy[train]= 0.9865  loss[test]= 0.1025  accuracy
[test]= 0.9685
Epoch  61: loss[train]= 0.0655  accuracy[train]= 0.9865  loss[test]= 0.1018  accuracy
[test]= 0.9685
Epoch  62: loss[train]= 0.0643  accuracy[train]= 0.9865  loss[test]= 0.1009  accuracy
[test]= 0.9685
Epoch  63: loss[train]= 0.0632  accuracy[train]= 0.9865  loss[test]= 0.1000  accuracy
[test]= 0.9685
Epoch  64: loss[train]= 0.0620  accuracy[train]= 0.9876  loss[test]= 0.0992  accuracy
[test]= 0.9685
Epoch  65: loss[train]= 0.0609  accuracy[train]= 0.9876  loss[test]= 0.0984  accuracy
[test]= 0.9685
Epoch  66: loss[train]= 0.0598  accuracy[train]= 0.9876  loss[test]= 0.0977  accuracy
[test]= 0.9674
Epoch  67: loss[train]= 0.0588  accuracy[train]= 0.9876  loss[test]= 0.0969  accuracy
[test]= 0.9674
Epoch  68: loss[train]= 0.0578  accuracy[train]= 0.9888  loss[test]= 0.0963  accuracy
[test]= 0.9674
Epoch  69: loss[train]= 0.0568  accuracy[train]= 0.9888  loss[test]= 0.0955  accuracy
[test]= 0.9674
Epoch  70: loss[train]= 0.0559  accuracy[train]= 0.9899  loss[test]= 0.0949  accuracy
[test]= 0.9674
Epoch  71: loss[train]= 0.0549  accuracy[train]= 0.9899  loss[test]= 0.0944  accuracy
[test]= 0.9685
Epoch  72: loss[train]= 0.0540  accuracy[train]= 0.9899  loss[test]= 0.0937  accuracy
[test]= 0.9685
Epoch  73: loss[train]= 0.0531  accuracy[train]= 0.9899  loss[test]= 0.0932  accuracy
[test]= 0.9685
Epoch  74: loss[train]= 0.0523  accuracy[train]= 0.9899  loss[test]= 0.0926  accuracy
[test]= 0.9685
Epoch  75: loss[train]= 0.0515  accuracy[train]= 0.9899  loss[test]= 0.0919  accuracy
[test]= 0.9685
Epoch  76: loss[train]= 0.0507  accuracy[train]= 0.9910  loss[test]= 0.0915  accuracy
[test]= 0.9685
Epoch  77: loss[train]= 0.0499  accuracy[train]= 0.9921  loss[test]= 0.0909  accuracy
[test]= 0.9685
Epoch  78: loss[train]= 0.0491  accuracy[train]= 0.9921  loss[test]= 0.0903  accuracy
```

```
[test]= 0.9685
Epoch  79: loss[train]= 0.0483  accuracy[train]= 0.9921  loss[test]= 0.0901  accuracy
[test]= 0.9685
Epoch  80: loss[train]= 0.0476  accuracy[train]= 0.9921  loss[test]= 0.0897  accuracy
[test]= 0.9685
Epoch  81: loss[train]= 0.0469  accuracy[train]= 0.9921  loss[test]= 0.0891  accuracy
[test]= 0.9685
Epoch  82: loss[train]= 0.0462  accuracy[train]= 0.9921  loss[test]= 0.0886  accuracy
[test]= 0.9685
Epoch  83: loss[train]= 0.0455  accuracy[train]= 0.9921  loss[test]= 0.0883  accuracy
[test]= 0.9685
Epoch  84: loss[train]= 0.0449  accuracy[train]= 0.9921  loss[test]= 0.0878  accuracy
[test]= 0.9685
Epoch  85: loss[train]= 0.0442  accuracy[train]= 0.9933  loss[test]= 0.0874  accuracy
[test]= 0.9685
Epoch  86: loss[train]= 0.0435  accuracy[train]= 0.9933  loss[test]= 0.0871  accuracy
[test]= 0.9685
Epoch  87: loss[train]= 0.0429  accuracy[train]= 0.9933  loss[test]= 0.0868  accuracy
[test]= 0.9674
Epoch  88: loss[train]= 0.0423  accuracy[train]= 0.9933  loss[test]= 0.0863  accuracy
[test]= 0.9697
Epoch  89: loss[train]= 0.0417  accuracy[train]= 0.9933  loss[test]= 0.0859  accuracy
[test]= 0.9697
Epoch  90: loss[train]= 0.0411  accuracy[train]= 0.9933  loss[test]= 0.0858  accuracy
[test]= 0.9697
Epoch  91: loss[train]= 0.0405  accuracy[train]= 0.9933  loss[test]= 0.0854  accuracy
[test]= 0.9697
Epoch  92: loss[train]= 0.0399  accuracy[train]= 0.9944  loss[test]= 0.0850  accuracy
[test]= 0.9697
Epoch  93: loss[train]= 0.0393  accuracy[train]= 0.9944  loss[test]= 0.0846  accuracy
[test]= 0.9697
Epoch  94: loss[train]= 0.0388  accuracy[train]= 0.9944  loss[test]= 0.0844  accuracy
[test]= 0.9697
Epoch  95: loss[train]= 0.0382  accuracy[train]= 0.9944  loss[test]= 0.0839  accuracy
[test]= 0.9697
Epoch  96: loss[train]= 0.0377  accuracy[train]= 0.9944  loss[test]= 0.0836  accuracy
[test]= 0.9697
Epoch  97: loss[train]= 0.0372  accuracy[train]= 0.9944  loss[test]= 0.0834  accuracy
[test]= 0.9697
Epoch  98: loss[train]= 0.0367  accuracy[train]= 0.9944  loss[test]= 0.0830  accuracy
[test]= 0.9697
Epoch  99: loss[train]= 0.0362  accuracy[train]= 0.9955  loss[test]= 0.0828  accuracy
[test]= 0.9708
```





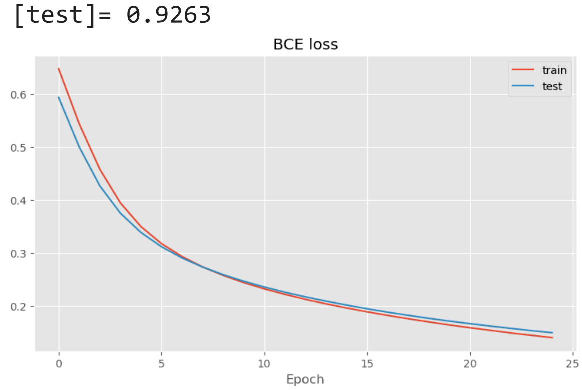


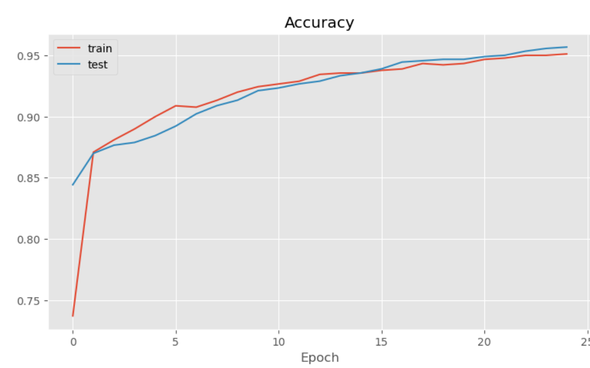
In [361…

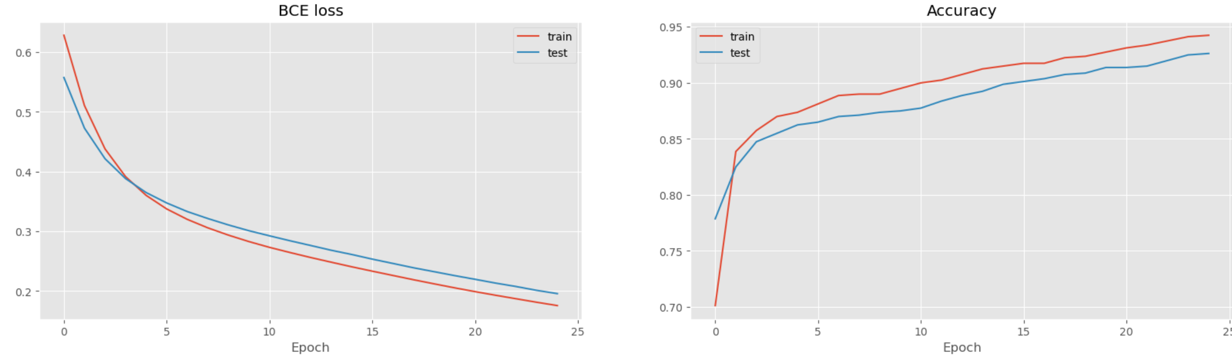
```
net = Net([
        Linear(64, 32, rng=rng),
        ReLU(),
        Linear(32, 1, rng=rng),
        Sigmoid()])

print("lower minibatch size net")
fit(net, x, y,
    epochs = 25,
    learning_rate = 0.001,
    mb_size = 1)

# Note: add more cells below if you want to keep runs with different hyperparameters.
rng = np.random.default_rng(12345)
net = Net([
        Linear(64, 32, rng=rng),
        ReLU(),
        Linear(32, 1, rng=rng),
        Sigmoid()])

print("\nhigher minibatch size net")
fit(net, x, y,
    epochs = 25,
    learning_rate = 0.001,
    mb_size = 100)
```

```
lower minibatch size net
Epoch   0: loss[train]= 0.6476  accuracy[train]= 0.7372  loss[test]= 0.5935  accuracy
[test]= 0.8443
Epoch   1: loss[train]= 0.5437  accuracy[train]= 0.8708  loss[test]= 0.5001  accuracy
[test]= 0.8699
Epoch   2: loss[train]= 0.4581  accuracy[train]= 0.8808  loss[test]= 0.4268  accuracy
[test]= 0.8765
Epoch   3: loss[train]= 0.3947  accuracy[train]= 0.8898  loss[test]= 0.3752  accuracy
[test]= 0.8788
Epoch   4: loss[train]= 0.3498  accuracy[train]= 0.8998  loss[test]= 0.3388  accuracy
[test]= 0.8843
Epoch   5: loss[train]= 0.3177  accuracy[train]= 0.9087  loss[test]= 0.3118  accuracy
[test]= 0.8921
Epoch   6: loss[train]= 0.2933  accuracy[train]= 0.9076  loss[test]= 0.2907  accuracy
[test]= 0.9021
Epoch   7: loss[train]= 0.2737  accuracy[train]= 0.9131  loss[test]= 0.2735  accuracy
[test]= 0.9088
Epoch   8: loss[train]= 0.2577  accuracy[train]= 0.9198  loss[test]= 0.2591  accuracy
[test]= 0.9132
Epoch   9: loss[train]= 0.2441  accuracy[train]= 0.9243  loss[test]= 0.2466  accuracy
[test]= 0.9210
Epoch  10: loss[train]= 0.2323  accuracy[train]= 0.9265  loss[test]= 0.2358  accuracy
[test]= 0.9232
Epoch  11: loss[train]= 0.2220  accuracy[train]= 0.9287  loss[test]= 0.2260  accuracy
[test]= 0.9266
Epoch  12: loss[train]= 0.2125  accuracy[train]= 0.9343  loss[test]= 0.2172  accuracy
[test]= 0.9288
Epoch  13: loss[train]= 0.2040  accuracy[train]= 0.9354  loss[test]= 0.2091  accuracy
[test]= 0.9333
Epoch  14: loss[train]= 0.1962  accuracy[train]= 0.9354  loss[test]= 0.2017  accuracy
[test]= 0.9355
Epoch  15: loss[train]= 0.1890  accuracy[train]= 0.9376  loss[test]= 0.1947  accuracy
[test]= 0.9388
Epoch  16: loss[train]= 0.1822  accuracy[train]= 0.9388  loss[test]= 0.1883  accuracy
[test]= 0.9444
Epoch  17: loss[train]= 0.1758  accuracy[train]= 0.9432  loss[test]= 0.1824  accuracy
[test]= 0.9455
Epoch  18: loss[train]= 0.1699  accuracy[train]= 0.9421  loss[test]= 0.1767  accuracy
[test]= 0.9466
Epoch  19: loss[train]= 0.1642  accuracy[train]= 0.9432  loss[test]= 0.1715  accuracy
[test]= 0.9466
Epoch  20: loss[train]= 0.1589  accuracy[train]= 0.9465  loss[test]= 0.1666  accuracy
[test]= 0.9488
Epoch  21: loss[train]= 0.1540  accuracy[train]= 0.9477  loss[test]= 0.1620  accuracy
[test]= 0.9499
Epoch  22: loss[train]= 0.1491  accuracy[train]= 0.9499  loss[test]= 0.1577  accuracy
[test]= 0.9533
Epoch  23: loss[train]= 0.1446  accuracy[train]= 0.9499  loss[test]= 0.1534  accuracy
[test]= 0.9555
Epoch  24: loss[train]= 0.1403  accuracy[train]= 0.9510  loss[test]= 0.1496  accuracy
[test]= 0.9566

higher minibatch size net
Epoch   0: loss[train]= 0.6283  accuracy[train]= 0.7013  loss[test]= 0.5575  accuracy
[test]= 0.7788
Epoch   1: loss[train]= 0.5105  accuracy[train]= 0.8387  loss[test]= 0.4728  accuracy
[test]= 0.8250
Epoch   2: loss[train]= 0.4380  accuracy[train]= 0.8575  loss[test]= 0.4216  accuracy
[test]= 0.8475
Epoch   3: loss[train]= 0.3915  accuracy[train]= 0.8700  loss[test]= 0.3883  accuracy
```

```
[test]= 0.8550
Epoch   4: loss[train]= 0.3602  accuracy[train]= 0.8738  loss[test]= 0.3650  accuracy
[test]= 0.8625
Epoch   5: loss[train]= 0.3375  accuracy[train]= 0.8812  loss[test]= 0.3475  accuracy
[test]= 0.8650
Epoch   6: loss[train]= 0.3201  accuracy[train]= 0.8888  loss[test]= 0.3331  accuracy
[test]= 0.8700
Epoch   7: loss[train]= 0.3059  accuracy[train]= 0.8900  loss[test]= 0.3215  accuracy
[test]= 0.8712
Epoch   8: loss[train]= 0.2937  accuracy[train]= 0.8900  loss[test]= 0.3107  accuracy
[test]= 0.8738
Epoch   9: loss[train]= 0.2829  accuracy[train]= 0.8950  loss[test]= 0.3010  accuracy
[test]= 0.8750
Epoch  10: loss[train]= 0.2732  accuracy[train]= 0.9000  loss[test]= 0.2925  accuracy
[test]= 0.8775
Epoch  11: loss[train]= 0.2645  accuracy[train]= 0.9025  loss[test]= 0.2842  accuracy
[test]= 0.8838
Epoch  12: loss[train]= 0.2562  accuracy[train]= 0.9075  loss[test]= 0.2763  accuracy
[test]= 0.8888
Epoch  13: loss[train]= 0.2484  accuracy[train]= 0.9125  loss[test]= 0.2683  accuracy
[test]= 0.8925
Epoch  14: loss[train]= 0.2406  accuracy[train]= 0.9150  loss[test]= 0.2613  accuracy
[test]= 0.8988
Epoch  15: loss[train]= 0.2333  accuracy[train]= 0.9175  loss[test]= 0.2535  accuracy
[test]= 0.9013
Epoch  16: loss[train]= 0.2261  accuracy[train]= 0.9175  loss[test]= 0.2463  accuracy
[test]= 0.9038
Epoch  17: loss[train]= 0.2190  accuracy[train]= 0.9225  loss[test]= 0.2390  accuracy
[test]= 0.9075
Epoch  18: loss[train]= 0.2121  accuracy[train]= 0.9237  loss[test]= 0.2325  accuracy
[test]= 0.9088
Epoch  19: loss[train]= 0.2054  accuracy[train]= 0.9275  loss[test]= 0.2259  accuracy
[test]= 0.9137
Epoch  20: loss[train]= 0.1990  accuracy[train]= 0.9313  loss[test]= 0.2197  accuracy
[test]= 0.9137
Epoch  21: loss[train]= 0.1929  accuracy[train]= 0.9338  loss[test]= 0.2133  accuracy
[test]= 0.9150
Epoch  22: loss[train]= 0.1871  accuracy[train]= 0.9375  loss[test]= 0.2076  accuracy
[test]= 0.9200
Epoch  23: loss[train]= 0.1812  accuracy[train]= 0.9413  loss[test]= 0.2012  accuracy
[test]= 0.9250
Epoch  24: loss[train]= 0.1757  accuracy[train]= 0.9425  loss[test]= 0.1957  accuracy
[test]= 0.9263
```







(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)

Setting the epochs to too low may result in underfitting where the network did not learn the characteristics of the images. Setting it too high might lead to overfitting where the network has almost perfect accuracy on train set but poor accuracy on test set.

Setting the learning rate too high leads to divergence from the minimum and poor generalization on both train and test sets. Setting it too low leads to a slow convergence to some minimum, increasing the learning rate might get us out of that minimum and converge to another one with lower loss and higher accuracy.

Higher minibatch size makes training faster but lower minibatch size generalizes slightly better (has a larger accuracy on test set).

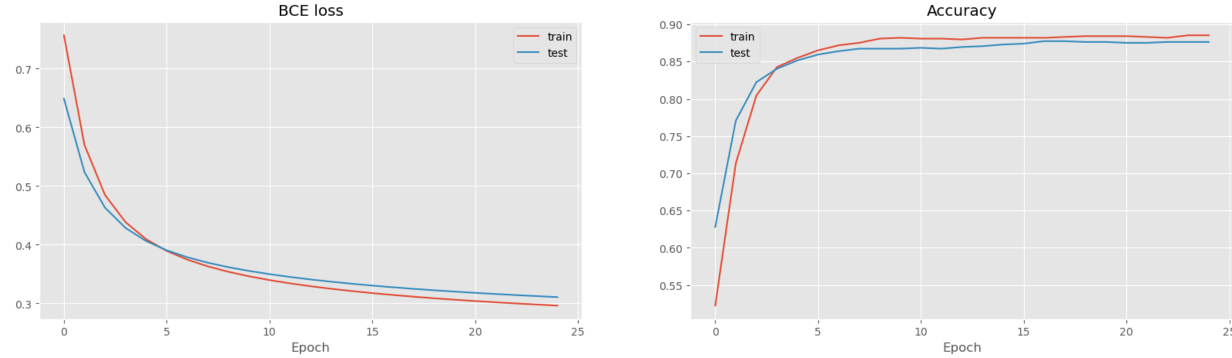
(d) Create and train a network with one linear layer followed by a sigmoid activation: (1 point)

`net = Net([Linear(...), Sigmoid()]`

In [362…

```
# TODO: Your code here.
rng = np.random.default_rng(12345)
net = Net([
        Linear(64, 1, rng=rng),
        Sigmoid()])

fit(net, x, y,
    epochs = 25,
    learning_rate = 0.001,
    mb_size = 10)
```

```
Epoch   0: loss[train]= 0.7569  accuracy[train]= 0.5225  loss[test]= 0.6488  accuracy
[test]= 0.6281
Epoch   1: loss[train]= 0.5701  accuracy[train]= 0.7135  loss[test]= 0.5236  accuracy
[test]= 0.7708
Epoch   2: loss[train]= 0.4846  accuracy[train]= 0.8045  loss[test]= 0.4630  accuracy
[test]= 0.8225
Epoch   3: loss[train]= 0.4384  accuracy[train]= 0.8427  loss[test]= 0.4285  accuracy
[test]= 0.8404
Epoch   4: loss[train]= 0.4094  accuracy[train]= 0.8551  loss[test]= 0.4062  accuracy
[test]= 0.8517
Epoch   5: loss[train]= 0.3894  accuracy[train]= 0.8652  loss[test]= 0.3905  accuracy
[test]= 0.8596
Epoch   6: loss[train]= 0.3746  accuracy[train]= 0.8719  loss[test]= 0.3787  accuracy
[test]= 0.8640
Epoch   7: loss[train]= 0.3631  accuracy[train]= 0.8753  loss[test]= 0.3693  accuracy
[test]= 0.8674
Epoch   8: loss[train]= 0.3539  accuracy[train]= 0.8809  loss[test]= 0.3617  accuracy
[test]= 0.8674
Epoch   9: loss[train]= 0.3462  accuracy[train]= 0.8820  loss[test]= 0.3553  accuracy
[test]= 0.8674
Epoch  10: loss[train]= 0.3397  accuracy[train]= 0.8809  loss[test]= 0.3498  accuracy
[test]= 0.8685
Epoch  11: loss[train]= 0.3341  accuracy[train]= 0.8809  loss[test]= 0.3450  accuracy
[test]= 0.8674
Epoch  12: loss[train]= 0.3292  accuracy[train]= 0.8798  loss[test]= 0.3408  accuracy
[test]= 0.8697
Epoch  13: loss[train]= 0.3249  accuracy[train]= 0.8820  loss[test]= 0.3370  accuracy
[test]= 0.8708
Epoch  14: loss[train]= 0.3210  accuracy[train]= 0.8820  loss[test]= 0.3335  accuracy
[test]= 0.8730
Epoch  15: loss[train]= 0.3175  accuracy[train]= 0.8820  loss[test]= 0.3304  accuracy
[test]= 0.8742
Epoch  16: loss[train]= 0.3143  accuracy[train]= 0.8820  loss[test]= 0.3275  accuracy
[test]= 0.8775
Epoch  17: loss[train]= 0.3114  accuracy[train]= 0.8831  loss[test]= 0.3248  accuracy
[test]= 0.8775
Epoch  18: loss[train]= 0.3088  accuracy[train]= 0.8843  loss[test]= 0.3224  accuracy
[test]= 0.8764
Epoch  19: loss[train]= 0.3063  accuracy[train]= 0.8843  loss[test]= 0.3201  accuracy
[test]= 0.8764
Epoch  20: loss[train]= 0.3040  accuracy[train]= 0.8843  loss[test]= 0.3180  accuracy
[test]= 0.8753
Epoch  21: loss[train]= 0.3019  accuracy[train]= 0.8831  loss[test]= 0.3160  accuracy
[test]= 0.8753
Epoch  22: loss[train]= 0.2999  accuracy[train]= 0.8820  loss[test]= 0.3141  accuracy
[test]= 0.8764
Epoch  23: loss[train]= 0.2980  accuracy[train]= 0.8854  loss[test]= 0.3124  accuracy
[test]= 0.8764
Epoch  24: loss[train]= 0.2963  accuracy[train]= 0.8854  loss[test]= 0.3107  accuracy
[test]= 0.8764
```



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

Multi-layer network generalizes much better than the single-layer network.

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

In [363…

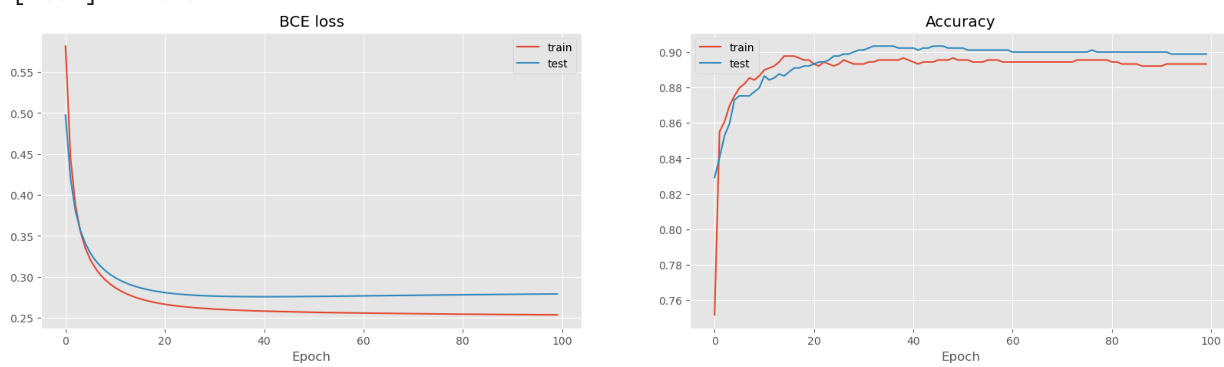
```
# TODO: Your code here.
rng = np.random.default_rng(12345)
net = Net([
        Linear(64, 64, rng=rng),
        Linear(64, 1, rng=rng),
        Sigmoid()])

fit(net, x, y,
    epochs = 100,
    learning_rate = 0.001,
    mb_size = 10)
```

```
Epoch   0: loss[train]= 0.5817  accuracy[train]= 0.7517  loss[test]= 0.4973  accuracy
[test]= 0.8292
Epoch   1: loss[train]= 0.4460  accuracy[train]= 0.8551  loss[test]= 0.4201  accuracy
[test]= 0.8404
Epoch   2: loss[train]= 0.3874  accuracy[train]= 0.8607  loss[test]= 0.3805  accuracy
[test]= 0.8528
Epoch   3: loss[train]= 0.3552  accuracy[train]= 0.8697  loss[test]= 0.3567  accuracy
[test]= 0.8596
Epoch   4: loss[train]= 0.3350  accuracy[train]= 0.8753  loss[test]= 0.3407  accuracy
[test]= 0.8730
Epoch   5: loss[train]= 0.3210  accuracy[train]= 0.8798  loss[test]= 0.3291  accuracy
[test]= 0.8753
Epoch   6: loss[train]= 0.3107  accuracy[train]= 0.8820  loss[test]= 0.3202  accuracy
[test]= 0.8753
Epoch   7: loss[train]= 0.3027  accuracy[train]= 0.8854  loss[test]= 0.3133  accuracy
[test]= 0.8753
Epoch   8: loss[train]= 0.2963  accuracy[train]= 0.8843  loss[test]= 0.3076  accuracy
[test]= 0.8775
Epoch   9: loss[train]= 0.2912  accuracy[train]= 0.8865  loss[test]= 0.3029  accuracy
[test]= 0.8798
Epoch  10: loss[train]= 0.2868  accuracy[train]= 0.8899  loss[test]= 0.2990  accuracy
[test]= 0.8865
Epoch  11: loss[train]= 0.2832  accuracy[train]= 0.8910  loss[test]= 0.2958  accuracy
[test]= 0.8843
Epoch  12: loss[train]= 0.2801  accuracy[train]= 0.8921  loss[test]= 0.2930  accuracy
[test]= 0.8854
Epoch  13: loss[train]= 0.2775  accuracy[train]= 0.8944  loss[test]= 0.2906  accuracy
[test]= 0.8876
Epoch  14: loss[train]= 0.2752  accuracy[train]= 0.8978  loss[test]= 0.2885  accuracy
[test]= 0.8865
Epoch  15: loss[train]= 0.2733  accuracy[train]= 0.8978  loss[test]= 0.2867  accuracy
[test]= 0.8888
Epoch  16: loss[train]= 0.2716  accuracy[train]= 0.8978  loss[test]= 0.2852  accuracy
[test]= 0.8910
Epoch  17: loss[train]= 0.2701  accuracy[train]= 0.8966  loss[test]= 0.2839  accuracy
[test]= 0.8910
Epoch  18: loss[train]= 0.2687  accuracy[train]= 0.8955  loss[test]= 0.2827  accuracy
[test]= 0.8921
Epoch  19: loss[train]= 0.2676  accuracy[train]= 0.8955  loss[test]= 0.2817  accuracy
[test]= 0.8921
Epoch  20: loss[train]= 0.2666  accuracy[train]= 0.8933  loss[test]= 0.2809  accuracy
[test]= 0.8933
Epoch  21: loss[train]= 0.2657  accuracy[train]= 0.8921  loss[test]= 0.2801  accuracy
[test]= 0.8944
Epoch  22: loss[train]= 0.2649  accuracy[train]= 0.8944  loss[test]= 0.2794  accuracy
[test]= 0.8944
Epoch  23: loss[train]= 0.2642  accuracy[train]= 0.8933  loss[test]= 0.2789  accuracy
[test]= 0.8955
Epoch  24: loss[train]= 0.2635  accuracy[train]= 0.8921  loss[test]= 0.2784  accuracy
[test]= 0.8978
Epoch  25: loss[train]= 0.2629  accuracy[train]= 0.8933  loss[test]= 0.2779  accuracy
[test]= 0.8978
Epoch  26: loss[train]= 0.2624  accuracy[train]= 0.8955  loss[test]= 0.2776  accuracy
[test]= 0.8989
Epoch  27: loss[train]= 0.2619  accuracy[train]= 0.8944  loss[test]= 0.2773  accuracy
[test]= 0.8989
Epoch  28: loss[train]= 0.2615  accuracy[train]= 0.8933  loss[test]= 0.2770  accuracy
[test]= 0.9000
Epoch  29: loss[train]= 0.2611  accuracy[train]= 0.8933  loss[test]= 0.2767  accuracy
[test]= 0.9011
```

```
Epoch  30: loss[train]= 0.2607  accuracy[train]= 0.8933  loss[test]= 0.2765  accuracy
[test]= 0.9011
Epoch  31: loss[train]= 0.2604  accuracy[train]= 0.8944  loss[test]= 0.2764  accuracy
[test]= 0.9022
Epoch  32: loss[train]= 0.2601  accuracy[train]= 0.8944  loss[test]= 0.2762  accuracy
[test]= 0.9034
Epoch  33: loss[train]= 0.2598  accuracy[train]= 0.8955  loss[test]= 0.2761  accuracy
[test]= 0.9034
Epoch  34: loss[train]= 0.2595  accuracy[train]= 0.8955  loss[test]= 0.2760  accuracy
[test]= 0.9034
Epoch  35: loss[train]= 0.2593  accuracy[train]= 0.8955  loss[test]= 0.2759  accuracy
[test]= 0.9034
Epoch  36: loss[train]= 0.2590  accuracy[train]= 0.8955  loss[test]= 0.2759  accuracy
[test]= 0.9034
Epoch  37: loss[train]= 0.2588  accuracy[train]= 0.8955  loss[test]= 0.2758  accuracy
[test]= 0.9022
Epoch  38: loss[train]= 0.2586  accuracy[train]= 0.8966  loss[test]= 0.2758  accuracy
[test]= 0.9022
Epoch  39: loss[train]= 0.2584  accuracy[train]= 0.8955  loss[test]= 0.2758  accuracy
[test]= 0.9022
Epoch  40: loss[train]= 0.2582  accuracy[train]= 0.8944  loss[test]= 0.2758  accuracy
[test]= 0.9022
Epoch  41: loss[train]= 0.2581  accuracy[train]= 0.8933  loss[test]= 0.2758  accuracy
[test]= 0.9011
Epoch  42: loss[train]= 0.2579  accuracy[train]= 0.8944  loss[test]= 0.2758  accuracy
[test]= 0.9022
Epoch  43: loss[train]= 0.2577  accuracy[train]= 0.8944  loss[test]= 0.2758  accuracy
[test]= 0.9022
Epoch  44: loss[train]= 0.2576  accuracy[train]= 0.8944  loss[test]= 0.2759  accuracy
[test]= 0.9034
Epoch  45: loss[train]= 0.2574  accuracy[train]= 0.8955  loss[test]= 0.2759  accuracy
[test]= 0.9034
Epoch  46: loss[train]= 0.2573  accuracy[train]= 0.8955  loss[test]= 0.2759  accuracy
[test]= 0.9034
Epoch  47: loss[train]= 0.2572  accuracy[train]= 0.8955  loss[test]= 0.2760  accuracy
[test]= 0.9022
Epoch  48: loss[train]= 0.2570  accuracy[train]= 0.8966  loss[test]= 0.2760  accuracy
[test]= 0.9022
Epoch  49: loss[train]= 0.2569  accuracy[train]= 0.8955  loss[test]= 0.2761  accuracy
[test]= 0.9022
Epoch  50: loss[train]= 0.2568  accuracy[train]= 0.8955  loss[test]= 0.2761  accuracy
[test]= 0.9022
Epoch  51: loss[train]= 0.2567  accuracy[train]= 0.8955  loss[test]= 0.2762  accuracy
[test]= 0.9011
Epoch  52: loss[train]= 0.2566  accuracy[train]= 0.8944  loss[test]= 0.2763  accuracy
[test]= 0.9011
Epoch  53: loss[train]= 0.2565  accuracy[train]= 0.8944  loss[test]= 0.2763  accuracy
[test]= 0.9011
Epoch  54: loss[train]= 0.2564  accuracy[train]= 0.8944  loss[test]= 0.2764  accuracy
[test]= 0.9011
Epoch  55: loss[train]= 0.2563  accuracy[train]= 0.8955  loss[test]= 0.2765  accuracy
[test]= 0.9011
Epoch  56: loss[train]= 0.2562  accuracy[train]= 0.8955  loss[test]= 0.2765  accuracy
[test]= 0.9011
Epoch  57: loss[train]= 0.2561  accuracy[train]= 0.8955  loss[test]= 0.2766  accuracy
[test]= 0.9011
Epoch  58: loss[train]= 0.2560  accuracy[train]= 0.8944  loss[test]= 0.2767  accuracy
[test]= 0.9011
Epoch  59: loss[train]= 0.2559  accuracy[train]= 0.8944  loss[test]= 0.2767  accuracy
[test]= 0.9011
```

```
Epoch  60: loss[train]= 0.2558  accuracy[train]= 0.8944  loss[test]= 0.2768  accuracy
[test]= 0.9000
Epoch  61: loss[train]= 0.2558  accuracy[train]= 0.8944  loss[test]= 0.2769  accuracy
[test]= 0.9000
Epoch  62: loss[train]= 0.2557  accuracy[train]= 0.8944  loss[test]= 0.2769  accuracy
[test]= 0.9000
Epoch  63: loss[train]= 0.2556  accuracy[train]= 0.8944  loss[test]= 0.2770  accuracy
[test]= 0.9000
Epoch  64: loss[train]= 0.2555  accuracy[train]= 0.8944  loss[test]= 0.2771  accuracy
[test]= 0.9000
Epoch  65: loss[train]= 0.2555  accuracy[train]= 0.8944  loss[test]= 0.2772  accuracy
[test]= 0.9000
Epoch  66: loss[train]= 0.2554  accuracy[train]= 0.8944  loss[test]= 0.2772  accuracy
[test]= 0.9000
Epoch  67: loss[train]= 0.2553  accuracy[train]= 0.8944  loss[test]= 0.2773  accuracy
[test]= 0.9000
Epoch  68: loss[train]= 0.2552  accuracy[train]= 0.8944  loss[test]= 0.2774  accuracy
[test]= 0.9000
Epoch  69: loss[train]= 0.2552  accuracy[train]= 0.8944  loss[test]= 0.2774  accuracy
[test]= 0.9000
Epoch  70: loss[train]= 0.2551  accuracy[train]= 0.8944  loss[test]= 0.2775  accuracy
[test]= 0.9000
Epoch  71: loss[train]= 0.2551  accuracy[train]= 0.8944  loss[test]= 0.2776  accuracy
[test]= 0.9000
Epoch  72: loss[train]= 0.2550  accuracy[train]= 0.8944  loss[test]= 0.2776  accuracy
[test]= 0.9000
Epoch  73: loss[train]= 0.2549  accuracy[train]= 0.8955  loss[test]= 0.2777  accuracy
[test]= 0.9000
Epoch  74: loss[train]= 0.2549  accuracy[train]= 0.8955  loss[test]= 0.2778  accuracy
[test]= 0.9000
Epoch  75: loss[train]= 0.2548  accuracy[train]= 0.8955  loss[test]= 0.2778  accuracy
[test]= 0.9000
Epoch  76: loss[train]= 0.2547  accuracy[train]= 0.8955  loss[test]= 0.2779  accuracy
[test]= 0.9011
Epoch  77: loss[train]= 0.2547  accuracy[train]= 0.8955  loss[test]= 0.2780  accuracy
[test]= 0.9000
Epoch  78: loss[train]= 0.2546  accuracy[train]= 0.8955  loss[test]= 0.2780  accuracy
[test]= 0.9000
Epoch  79: loss[train]= 0.2546  accuracy[train]= 0.8955  loss[test]= 0.2781  accuracy
[test]= 0.9000
Epoch  80: loss[train]= 0.2545  accuracy[train]= 0.8944  loss[test]= 0.2782  accuracy
[test]= 0.9000
Epoch  81: loss[train]= 0.2545  accuracy[train]= 0.8944  loss[test]= 0.2782  accuracy
[test]= 0.9000
Epoch  82: loss[train]= 0.2544  accuracy[train]= 0.8933  loss[test]= 0.2783  accuracy
[test]= 0.9000
Epoch  83: loss[train]= 0.2544  accuracy[train]= 0.8933  loss[test]= 0.2783  accuracy
[test]= 0.9000
Epoch  84: loss[train]= 0.2543  accuracy[train]= 0.8933  loss[test]= 0.2784  accuracy
[test]= 0.9000
Epoch  85: loss[train]= 0.2543  accuracy[train]= 0.8933  loss[test]= 0.2785  accuracy
[test]= 0.9000
Epoch  86: loss[train]= 0.2542  accuracy[train]= 0.8921  loss[test]= 0.2785  accuracy
[test]= 0.9000
Epoch  87: loss[train]= 0.2542  accuracy[train]= 0.8921  loss[test]= 0.2786  accuracy
[test]= 0.9000
Epoch  88: loss[train]= 0.2541  accuracy[train]= 0.8921  loss[test]= 0.2786  accuracy
[test]= 0.9000
Epoch  89: loss[train]= 0.2541  accuracy[train]= 0.8921  loss[test]= 0.2787  accuracy
[test]= 0.9000
```

```
Epoch  90: loss[train]= 0.2540  accuracy[train]= 0.8921  loss[test]= 0.2787  accuracy
[test]= 0.9000
Epoch  91: loss[train]= 0.2540  accuracy[train]= 0.8933  loss[test]= 0.2788  accuracy
[test]= 0.9000
Epoch  92: loss[train]= 0.2539  accuracy[train]= 0.8933  loss[test]= 0.2789  accuracy
[test]= 0.8989
Epoch  93: loss[train]= 0.2539  accuracy[train]= 0.8933  loss[test]= 0.2789  accuracy
[test]= 0.8989
Epoch  94: loss[train]= 0.2538  accuracy[train]= 0.8933  loss[test]= 0.2790  accuracy
[test]= 0.8989
Epoch  95: loss[train]= 0.2538  accuracy[train]= 0.8933  loss[test]= 0.2790  accuracy
[test]= 0.8989
Epoch  96: loss[train]= 0.2538  accuracy[train]= 0.8933  loss[test]= 0.2791  accuracy
[test]= 0.8989
Epoch  97: loss[train]= 0.2537  accuracy[train]= 0.8933  loss[test]= 0.2791  accuracy
[test]= 0.8989
Epoch  98: loss[train]= 0.2537  accuracy[train]= 0.8933  loss[test]= 0.2791  accuracy
[test]= 0.8989
Epoch  99: loss[train]= 0.2536  accuracy[train]= 0.8933  loss[test]= 0.2792  accuracy
[test]= 0.8989
```



(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)

Network with double linear layers converges faster to the minimum found by the optimization method. With large enough epoch size both network have negligible difference in accuracy on test set. Composition of linear function is a linear function so n linear layers is equivalent to 1 linear layer, activation functions add non-linearity to the composition giving the neural network the ability to fit to any function.

(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 [364…

```
# TODO: Your code here.
rng = np.random.default_rng(12345)
net = Net([
        Linear(64, 64, rng=rng),
        ReLU(),
        Linear(64, 32, rng=rng),
        ReLU(),
        Linear(32, 16, rng=rng),
        ReLU(),
        Linear(16, 1, rng=rng),
```

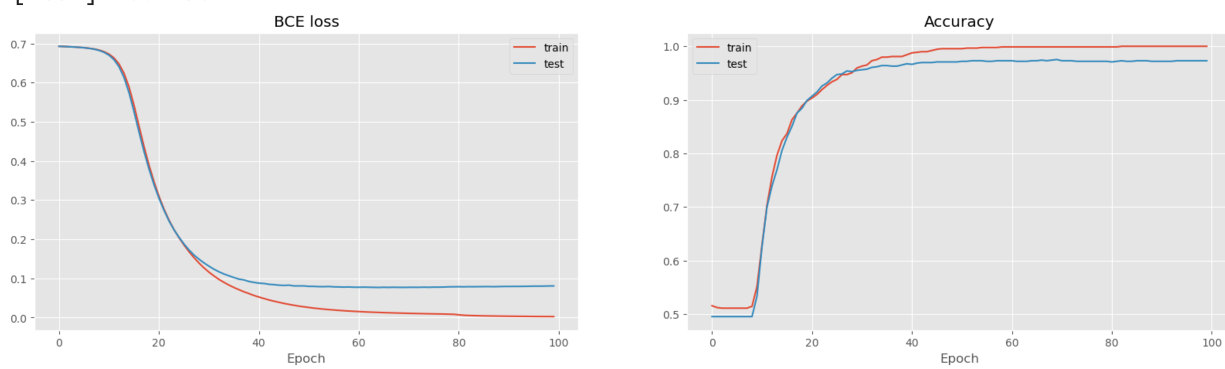
```
        Sigmoid()])

fit(net, x, y,
    epochs = 100,
    learning_rate = 0.001,
    mb_size = 10)
```

```
Epoch   0: loss[train]= 0.6927  accuracy[train]= 0.5157  loss[test]= 0.6926  accuracy
[test]= 0.4955
Epoch   1: loss[train]= 0.6921  accuracy[train]= 0.5124  loss[test]= 0.6921  accuracy
[test]= 0.4955
Epoch   2: loss[train]= 0.6916  accuracy[train]= 0.5112  loss[test]= 0.6916  accuracy
[test]= 0.4955
Epoch   3: loss[train]= 0.6909  accuracy[train]= 0.5112  loss[test]= 0.6909  accuracy
[test]= 0.4955
Epoch   4: loss[train]= 0.6901  accuracy[train]= 0.5112  loss[test]= 0.6901  accuracy
[test]= 0.4955
Epoch   5: loss[train]= 0.6891  accuracy[train]= 0.5112  loss[test]= 0.6890  accuracy
[test]= 0.4955
Epoch   6: loss[train]= 0.6878  accuracy[train]= 0.5112  loss[test]= 0.6875  accuracy
[test]= 0.4955
Epoch   7: loss[train]= 0.6859  accuracy[train]= 0.5112  loss[test]= 0.6853  accuracy
[test]= 0.4955
Epoch   8: loss[train]= 0.6832  accuracy[train]= 0.5146  loss[test]= 0.6821  accuracy
[test]= 0.4955
Epoch   9: loss[train]= 0.6790  accuracy[train]= 0.5517  loss[test]= 0.6773  accuracy
[test]= 0.5337
Epoch  10: loss[train]= 0.6727  accuracy[train]= 0.6292  loss[test]= 0.6699  accuracy
[test]= 0.6258
Epoch  11: loss[train]= 0.6629  accuracy[train]= 0.7034  loss[test]= 0.6585  accuracy
[test]= 0.7000
Epoch  12: loss[train]= 0.6476  accuracy[train]= 0.7562  loss[test]= 0.6405  accuracy
[test]= 0.7393
Epoch  13: loss[train]= 0.6239  accuracy[train]= 0.7966  loss[test]= 0.6132  accuracy
[test]= 0.7697
Epoch  14: loss[train]= 0.5889  accuracy[train]= 0.8236  loss[test]= 0.5743  accuracy
[test]= 0.8056
Epoch  15: loss[train]= 0.5423  accuracy[train]= 0.8371  loss[test]= 0.5260  accuracy
[test]= 0.8303
Epoch  16: loss[train]= 0.4894  accuracy[train]= 0.8629  loss[test]= 0.4740  accuracy
[test]= 0.8506
Epoch  17: loss[train]= 0.4371  accuracy[train]= 0.8753  loss[test]= 0.4243  accuracy
[test]= 0.8753
Epoch  18: loss[train]= 0.3895  accuracy[train]= 0.8888  loss[test]= 0.3795  accuracy
[test]= 0.8843
Epoch  19: loss[train]= 0.3471  accuracy[train]= 0.8978  loss[test]= 0.3394  accuracy
[test]= 0.8989
Epoch  20: loss[train]= 0.3096  accuracy[train]= 0.9034  loss[test]= 0.3045  accuracy
[test]= 0.9067
Epoch  21: loss[train]= 0.2772  accuracy[train]= 0.9101  loss[test]= 0.2739  accuracy
[test]= 0.9146
Epoch  22: loss[train]= 0.2490  accuracy[train]= 0.9191  loss[test]= 0.2468  accuracy
[test]= 0.9258
Epoch  23: loss[train]= 0.2248  accuracy[train]= 0.9270  loss[test]= 0.2243  accuracy
[test]= 0.9315
Epoch  24: loss[train]= 0.2040  accuracy[train]= 0.9337  loss[test]= 0.2051  accuracy
[test]= 0.9404
Epoch  25: loss[train]= 0.1853  accuracy[train]= 0.9382  loss[test]= 0.1879  accuracy
[test]= 0.9472
Epoch  26: loss[train]= 0.1683  accuracy[train]= 0.9472  loss[test]= 0.1727  accuracy
[test]= 0.9483
Epoch  27: loss[train]= 0.1531  accuracy[train]= 0.9472  loss[test]= 0.1596  accuracy
[test]= 0.9539
Epoch  28: loss[train]= 0.1393  accuracy[train]= 0.9506  loss[test]= 0.1491  accuracy
[test]= 0.9528
Epoch  29: loss[train]= 0.1269  accuracy[train]= 0.9596  loss[test]= 0.1396  accuracy
[test]= 0.9551
```

```
Epoch  30: loss[train]= 0.1159  accuracy[train]= 0.9629  loss[test]= 0.1312  accuracy
[test]= 0.9562
Epoch  31: loss[train]= 0.1063  accuracy[train]= 0.9652  loss[test]= 0.1234  accuracy
[test]= 0.9573
Epoch  32: loss[train]= 0.0974  accuracy[train]= 0.9730  loss[test]= 0.1169  accuracy
[test]= 0.9607
Epoch  33: loss[train]= 0.0899  accuracy[train]= 0.9753  loss[test]= 0.1112  accuracy
[test]= 0.9618
Epoch  34: loss[train]= 0.0829  accuracy[train]= 0.9798  loss[test]= 0.1066  accuracy
[test]= 0.9640
Epoch  35: loss[train]= 0.0767  accuracy[train]= 0.9798  loss[test]= 0.1021  accuracy
[test]= 0.9640
Epoch  36: loss[train]= 0.0711  accuracy[train]= 0.9809  loss[test]= 0.0981  accuracy
[test]= 0.9629
Epoch  37: loss[train]= 0.0658  accuracy[train]= 0.9809  loss[test]= 0.0960  accuracy
[test]= 0.9629
Epoch  38: loss[train]= 0.0612  accuracy[train]= 0.9809  loss[test]= 0.0921  accuracy
[test]= 0.9652
Epoch  39: loss[train]= 0.0564  accuracy[train]= 0.9843  loss[test]= 0.0897  accuracy
[test]= 0.9674
Epoch  40: loss[train]= 0.0523  accuracy[train]= 0.9876  loss[test]= 0.0878  accuracy
[test]= 0.9663
Epoch  41: loss[train]= 0.0486  accuracy[train]= 0.9888  loss[test]= 0.0870  accuracy
[test]= 0.9685
Epoch  42: loss[train]= 0.0449  accuracy[train]= 0.9899  loss[test]= 0.0849  accuracy
[test]= 0.9697
Epoch  43: loss[train]= 0.0418  accuracy[train]= 0.9899  loss[test]= 0.0840  accuracy
[test]= 0.9697
Epoch  44: loss[train]= 0.0389  accuracy[train]= 0.9921  loss[test]= 0.0827  accuracy
[test]= 0.9697
Epoch  45: loss[train]= 0.0361  accuracy[train]= 0.9944  loss[test]= 0.0820  accuracy
[test]= 0.9708
Epoch  46: loss[train]= 0.0336  accuracy[train]= 0.9955  loss[test]= 0.0827  accuracy
[test]= 0.9708
Epoch  47: loss[train]= 0.0314  accuracy[train]= 0.9955  loss[test]= 0.0807  accuracy
[test]= 0.9708
Epoch  48: loss[train]= 0.0292  accuracy[train]= 0.9955  loss[test]= 0.0807  accuracy
[test]= 0.9708
Epoch  49: loss[train]= 0.0274  accuracy[train]= 0.9955  loss[test]= 0.0808  accuracy
[test]= 0.9708
Epoch  50: loss[train]= 0.0258  accuracy[train]= 0.9955  loss[test]= 0.0797  accuracy
[test]= 0.9719
Epoch  51: loss[train]= 0.0241  accuracy[train]= 0.9966  loss[test]= 0.0796  accuracy
[test]= 0.9719
Epoch  52: loss[train]= 0.0227  accuracy[train]= 0.9966  loss[test]= 0.0789  accuracy
[test]= 0.9730
Epoch  53: loss[train]= 0.0214  accuracy[train]= 0.9966  loss[test]= 0.0789  accuracy
[test]= 0.9730
Epoch  54: loss[train]= 0.0203  accuracy[train]= 0.9978  loss[test]= 0.0793  accuracy
[test]= 0.9730
Epoch  55: loss[train]= 0.0193  accuracy[train]= 0.9978  loss[test]= 0.0783  accuracy
[test]= 0.9719
Epoch  56: loss[train]= 0.0183  accuracy[train]= 0.9978  loss[test]= 0.0781  accuracy
[test]= 0.9719
Epoch  57: loss[train]= 0.0174  accuracy[train]= 0.9978  loss[test]= 0.0777  accuracy
[test]= 0.9730
Epoch  58: loss[train]= 0.0166  accuracy[train]= 0.9989  loss[test]= 0.0780  accuracy
[test]= 0.9730
Epoch  59: loss[train]= 0.0159  accuracy[train]= 0.9989  loss[test]= 0.0774  accuracy
[test]= 0.9730
```

```
Epoch  60: loss[train]= 0.0152  accuracy[train]= 0.9989  loss[test]= 0.0774  accuracy
[test]= 0.9730
Epoch  61: loss[train]= 0.0146  accuracy[train]= 0.9989  loss[test]= 0.0777  accuracy
[test]= 0.9719
Epoch  62: loss[train]= 0.0140  accuracy[train]= 0.9989  loss[test]= 0.0773  accuracy
[test]= 0.9719
Epoch  63: loss[train]= 0.0135  accuracy[train]= 0.9989  loss[test]= 0.0771  accuracy
[test]= 0.9719
Epoch  64: loss[train]= 0.0130  accuracy[train]= 0.9989  loss[test]= 0.0769  accuracy
[test]= 0.9730
Epoch  65: loss[train]= 0.0126  accuracy[train]= 0.9989  loss[test]= 0.0773  accuracy
[test]= 0.9730
Epoch  66: loss[train]= 0.0122  accuracy[train]= 0.9989  loss[test]= 0.0770  accuracy
[test]= 0.9742
Epoch  67: loss[train]= 0.0118  accuracy[train]= 0.9989  loss[test]= 0.0773  accuracy
[test]= 0.9730
Epoch  68: loss[train]= 0.0114  accuracy[train]= 0.9989  loss[test]= 0.0771  accuracy
[test]= 0.9742
Epoch  69: loss[train]= 0.0111  accuracy[train]= 0.9989  loss[test]= 0.0770  accuracy
[test]= 0.9753
Epoch  70: loss[train]= 0.0107  accuracy[train]= 0.9989  loss[test]= 0.0772  accuracy
[test]= 0.9730
Epoch  71: loss[train]= 0.0104  accuracy[train]= 0.9989  loss[test]= 0.0772  accuracy
[test]= 0.9730
Epoch  72: loss[train]= 0.0102  accuracy[train]= 0.9989  loss[test]= 0.0772  accuracy
[test]= 0.9730
Epoch  73: loss[train]= 0.0099  accuracy[train]= 0.9989  loss[test]= 0.0776  accuracy
[test]= 0.9719
Epoch  74: loss[train]= 0.0096  accuracy[train]= 0.9989  loss[test]= 0.0772  accuracy
[test]= 0.9719
Epoch  75: loss[train]= 0.0094  accuracy[train]= 0.9989  loss[test]= 0.0776  accuracy
[test]= 0.9719
Epoch  76: loss[train]= 0.0091  accuracy[train]= 0.9989  loss[test]= 0.0776  accuracy
[test]= 0.9719
Epoch  77: loss[train]= 0.0089  accuracy[train]= 0.9989  loss[test]= 0.0779  accuracy
[test]= 0.9719
Epoch  78: loss[train]= 0.0085  accuracy[train]= 0.9989  loss[test]= 0.0784  accuracy
[test]= 0.9719
Epoch  79: loss[train]= 0.0082  accuracy[train]= 0.9989  loss[test]= 0.0785  accuracy
[test]= 0.9719
Epoch  80: loss[train]= 0.0070  accuracy[train]= 0.9989  loss[test]= 0.0787  accuracy
[test]= 0.9708
Epoch  81: loss[train]= 0.0059  accuracy[train]= 0.9989  loss[test]= 0.0784  accuracy
[test]= 0.9719
Epoch  82: loss[train]= 0.0052  accuracy[train]= 1.0000  loss[test]= 0.0788  accuracy
[test]= 0.9730
Epoch  83: loss[train]= 0.0049  accuracy[train]= 1.0000  loss[test]= 0.0786  accuracy
[test]= 0.9719
Epoch  84: loss[train]= 0.0045  accuracy[train]= 1.0000  loss[test]= 0.0788  accuracy
[test]= 0.9719
Epoch  85: loss[train]= 0.0043  accuracy[train]= 1.0000  loss[test]= 0.0790  accuracy
[test]= 0.9730
Epoch  86: loss[train]= 0.0041  accuracy[train]= 1.0000  loss[test]= 0.0791  accuracy
[test]= 0.9730
Epoch  87: loss[train]= 0.0039  accuracy[train]= 1.0000  loss[test]= 0.0788  accuracy
[test]= 0.9730
Epoch  88: loss[train]= 0.0037  accuracy[train]= 1.0000  loss[test]= 0.0791  accuracy
[test]= 0.9719
Epoch  89: loss[train]= 0.0036  accuracy[train]= 1.0000  loss[test]= 0.0794  accuracy
[test]= 0.9719
```

```
Epoch  90: loss[train]= 0.0034  accuracy[train]= 1.0000  loss[test]= 0.0794  accuracy
[test]= 0.9719
Epoch  91: loss[train]= 0.0033  accuracy[train]= 1.0000  loss[test]= 0.0795  accuracy
[test]= 0.9719
Epoch  92: loss[train]= 0.0031  accuracy[train]= 1.0000  loss[test]= 0.0795  accuracy
[test]= 0.9719
Epoch  93: loss[train]= 0.0030  accuracy[train]= 1.0000  loss[test]= 0.0797  accuracy
[test]= 0.9730
Epoch  94: loss[train]= 0.0029  accuracy[train]= 1.0000  loss[test]= 0.0799  accuracy
[test]= 0.9730
Epoch  95: loss[train]= 0.0028  accuracy[train]= 1.0000  loss[test]= 0.0801  accuracy
[test]= 0.9730
Epoch  96: loss[train]= 0.0027  accuracy[train]= 1.0000  loss[test]= 0.0801  accuracy
[test]= 0.9730
Epoch  97: loss[train]= 0.0026  accuracy[train]= 1.0000  loss[test]= 0.0802  accuracy
[test]= 0.9730
Epoch  98: loss[train]= 0.0025  accuracy[train]= 1.0000  loss[test]= 0.0806  accuracy
[test]= 0.9730
Epoch  99: loss[train]= 0.0025  accuracy[train]= 1.0000  loss[test]= 0.0808  accuracy
[test]= 0.9730
```



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

We we're not able to find a perfect classification on the test set. Given large epoch size the network will perfectly classify the train set but the loss on the test set will increase. One way to avoid this is using a validation set for which we do not update the parameters of the model and for which we can stop when we notice that the accuracy on validation set is increasing avoiding overfitting.

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)

The learning rate influences at each step how much the weights are updated. If the learning rate is too low the algorithm will most likely get stuck in a local minimum and won't be able to "get out" of it and will converge very slowly to it. If the learning rate is too high we might overshoot the minimum and get stuck in it by oscilating around it e.g $f(x) = x^2$ where the learning rate is x.

(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)

The minibatch size influence how many samples are used to compute the gradient at each step. If the minibatch size is too small, the gradient will be computed on a small number of samples and will be noisy. If the minibatch size is too high, the gradient will be computed on a large number of samples and will be less noisy, but the training will be slower.

(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)

If the weights w were initialised as zeros, the gradient would be zero and the weights would not be updated. This is not a problem for the bias because the bias is not multiplied by the input, so the gradient is not zero.

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