Міністерство освіти і науки України
Харківський національний університет радіоелектроніки

Кафедра штучного інтелекту

Дисципліна: “Штучні нейронні мережі: архітектура”

ЛАБОРАТОРНА РОБОТА 1

# “ОЗНАЙОМЛЕННЯ З ВІЗУАЛЬНИМ СЕРЕДОВИЩЕМ ІМІТАЦІЙНОГО МОДЕЛЮВАННЯ MATLAB. СТВОРЕННЯ НЕЙРОННОЇ МЕРЕЖІ З ПРЯМОЇ ПЕРЕДАЧІ ІНФОРМАЦІЇ. АЛГОРИТМИ НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ.”

Виконали ст. гр. ІТШІ-18-1:
Апраксін Антон Романович
Михно Євген Віталійович
Соколенко Дмитро Олександрович

Прийняла:
Чала О. С.
з оцінкою “_____”
“___”_____20___р

Харків 2021

# 1 ОЗНАЙОМЛЕННЯ З ВІЗУАЛЬНИМ СЕРЕДОВИЩЕМ ІМІТАЦІЙНОГО МОДЕЛЮВАННЯ MATLAB. СТВОРЕННЯ НЕЙРОННОЇ МЕРЕЖІ З ПРЯМОЇ ПЕРЕДАЧІ ІНФОРМАЦІЇ. АЛГОРИТМИ НАВЧАННЯ НЕЙРОННИХ МЕРЕЖ

## 1.1 Мета роботи:

Ознайомлення із візуальним середовищем імітаційного моделювання MATLAB (Matrix laboratory). Освоєння методики створення нейронної мережі із прямою передачею даних. Освоєння різноманітних алгоритмів навчання нейронних мереж та моделювання їх у середовищі MATLAB.

## 1.2 Лістинг коду:

### 1.2.1 Оптимізатор CGF

```
class OptimizerCGF(OptimizerAbstract):
    def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7,
max_update=10):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.max_update = max_update

    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * (1. / (1.
 + self.decay * self.iterations))

    # Update parameters
    def update_params(self, layer):
        def calc_beta(dweights, dweights_prev):
            assert dweights.shape[0] == dweights.size
            assert dweights_prev.shape[0] == dweights_prev.size
            return dweights.dot(dweights) / (dweights_prev.dot(
dweights_prev) + 1)

        if not hasattr(layer, 'prev_dweights'):
            layer.prev_dweights = layer.dweights.copy()
            layer.prev_dbiases = layer.dbiases.copy()
            layer.weight_p = np.zeros_like(-layer.dweights)
```

```
            layer.bias_p = np.zeros_like(-layer.dbiases)

        weight_update = self.current_learning_rate * layer.weight_p
        biases_update = self.current_learning_rate * layer.bias_p

        beta_weights = np.array([calc_beta(dweight, dweight_prev) for (
dweight, dweight_prev) in
                                 zip(layer.dweights.T, layer.
prev_dweights.T)])
        beta_biases = np.array([calc_beta(dweight, dweight_prev) for (
dweight, dweight_prev) in
                                zip(layer.dbiases.T, layer.prev_dbiases
.T)])

        layer.weight_p = -layer.dweights + beta_weights * layer.
weight_p
        layer.bias_p = -layer.dbiases + beta_biases * layer.bias_p

        layer.prev_dweights = layer.dweights.copy()
        layer.prev_dbiases = layer.dbiases.copy()

        layer.weights += weight_update  # np.clip(weight_update, -self.
max_update, self.max_update)
        layer.biases += biases_update  # np.clip(biases_update, -self.
max_update, self.max_update)

    def post_update_params(self):
        self.iterations += 1
```

### 1.2.2 Оптимізатор GDM

```
class OptimizerGDM(OptimizerAbstract):
def __init__(self, learning_rate=0.001, decay=0., momentum=0.):
        self.learning_rate = learning_rate
        self.current_learning_rate = learning_rate
        self.decay = decay
        self.iterations = 0
        self.momentum = momentum

    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate * \
                                         (1. / (1. + self.decay * self.
```

```
iterations))

 def update_params(self, layer):
     if not hasattr(layer, 'weight_momentums'):
         layer.weight_momentums = np.zeros_like(layer.weights)
         layer.bias_momentums = np.zeros_like(layer.biases)

     weight_updates = \
         self.momentum * layer.weight_momentums + \
         (1.0 - self.momentum) * self.current_learning_rate * layer.
dweights
     layer.weight_momentums = weight_updates

     bias_updates = \
         self.momentum * layer.bias_momentums + \
         (1.0 - self.momentum) * self.current_learning_rate * layer.
dbiases
     layer.bias_momentums = bias_updates

     layer.weights -= weight_updates
     layer.biases -= bias_updates

 def post_update_params(self):
     self.iterations += 1
```

### 1.2.3 Оптимізатор BFGS

```
 class OptimizerBFGS(OptimizerAbstract):
     def __init__(self, learning_rate=0.001, decay=0., epsilon=1e-7)
:
         self.learning_rate = learning_rate
         self.current_learning_rate = learning_rate
         self.decay = decay
         self.iterations = 0
         self.epsilon = epsilon

     def pre_update_params(self):
         if self.decay:
             self.current_learning_rate = self.learning_rate * (1. /
 (1. + self.decay * self.iterations))

     # Update parameters
     def update_params(self, layer):
```

```
            I = np.eye(layer.weights.shape[1])
            if not hasattr(layer, f'prev_weights'):
                layer.H = I
                layer.prev_weights = np.zeros_like(layer.weights)
                layer.prev_dweights = np.zeros_like(layer.dweights)

            sk = layer.weights - layer.prev_weights
            yk = layer.dweights - layer.prev_dweights

            rho_inv = yk.T @ sk
            rho = 1 / (rho_inv + 0.1)

            A1 = (I - rho * (yk.T @ sk))
            A2 = (I - rho * (sk.T @ yk))
            left = A1 @ (layer.H @ A2)
            layer.H = left + rho * (yk.T @ yk)

            weight_update = self.current_learning_rate * -(layer.H @
layer.dweights.T)

            layer.weights += weight_update.T

            layer.prev_weights = layer.weights.copy()
            layer.prev_dweights = layer.dweights.copy()

        def post_update_params(self):
            self.iterations += 1
```

## 1.3 Результати виконання:

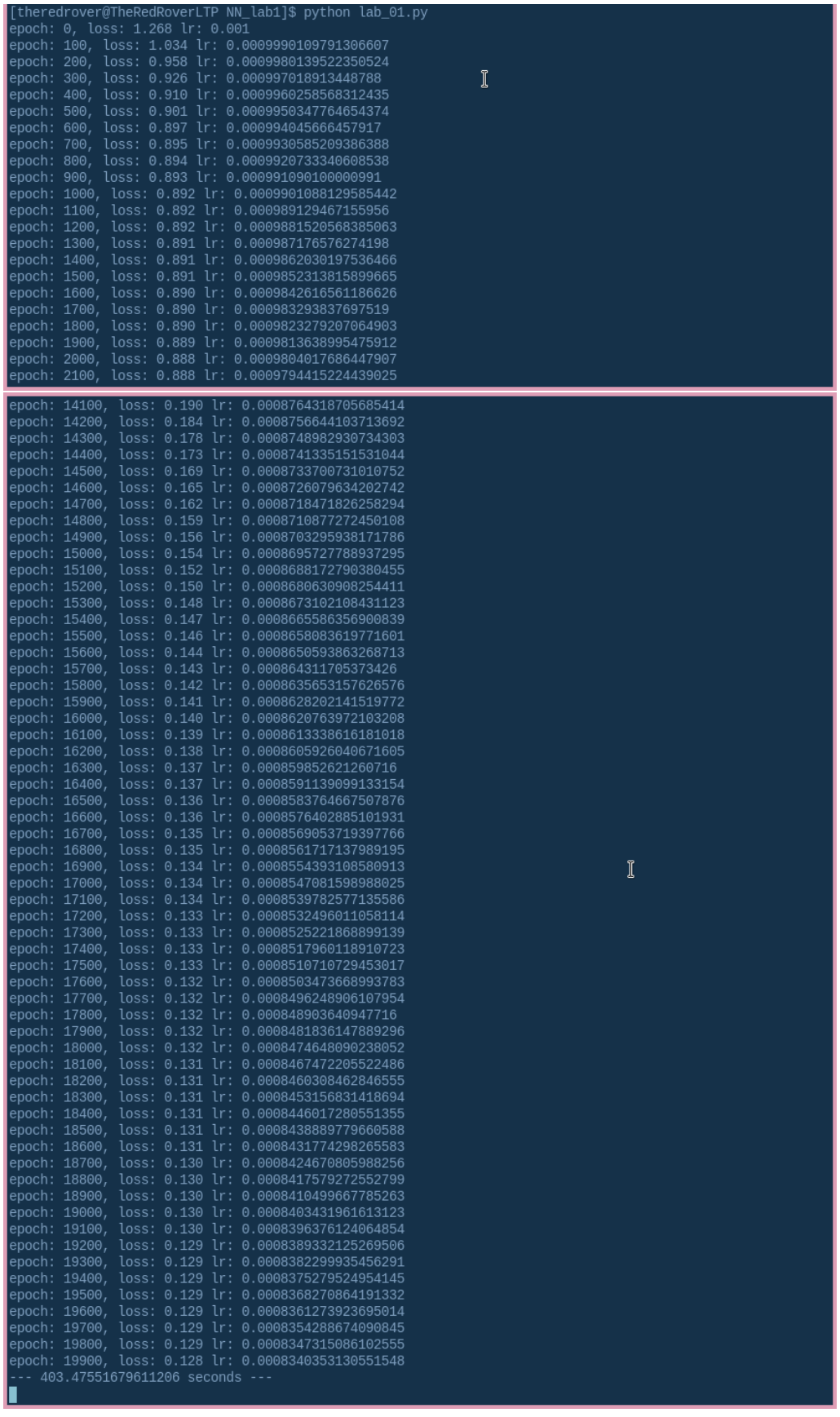


Рис. 1: Показники початкових та кінцевих епох з використанням CGF

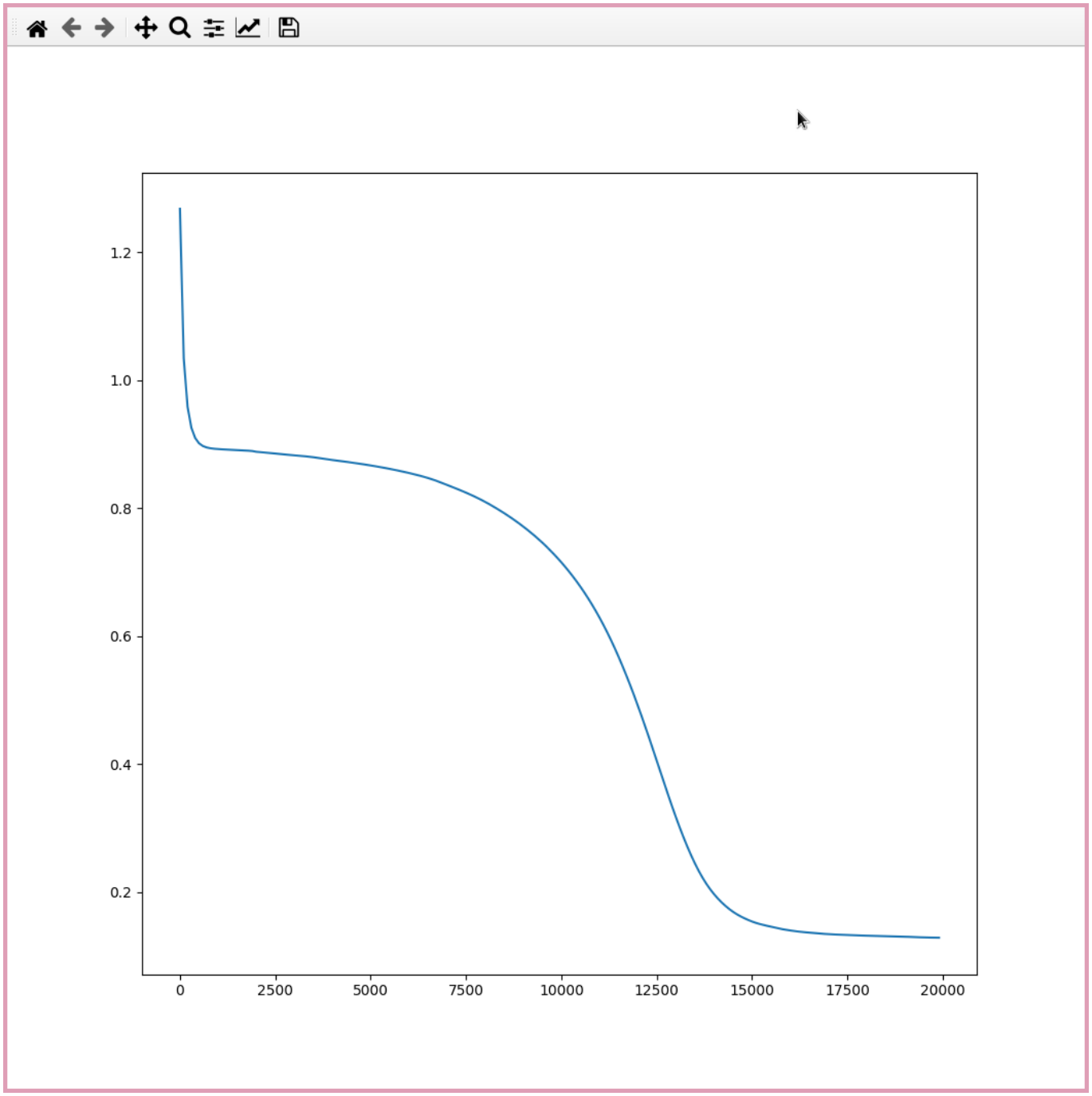


Рис. 2: Зниження помилки відносно епох з використанням CGF

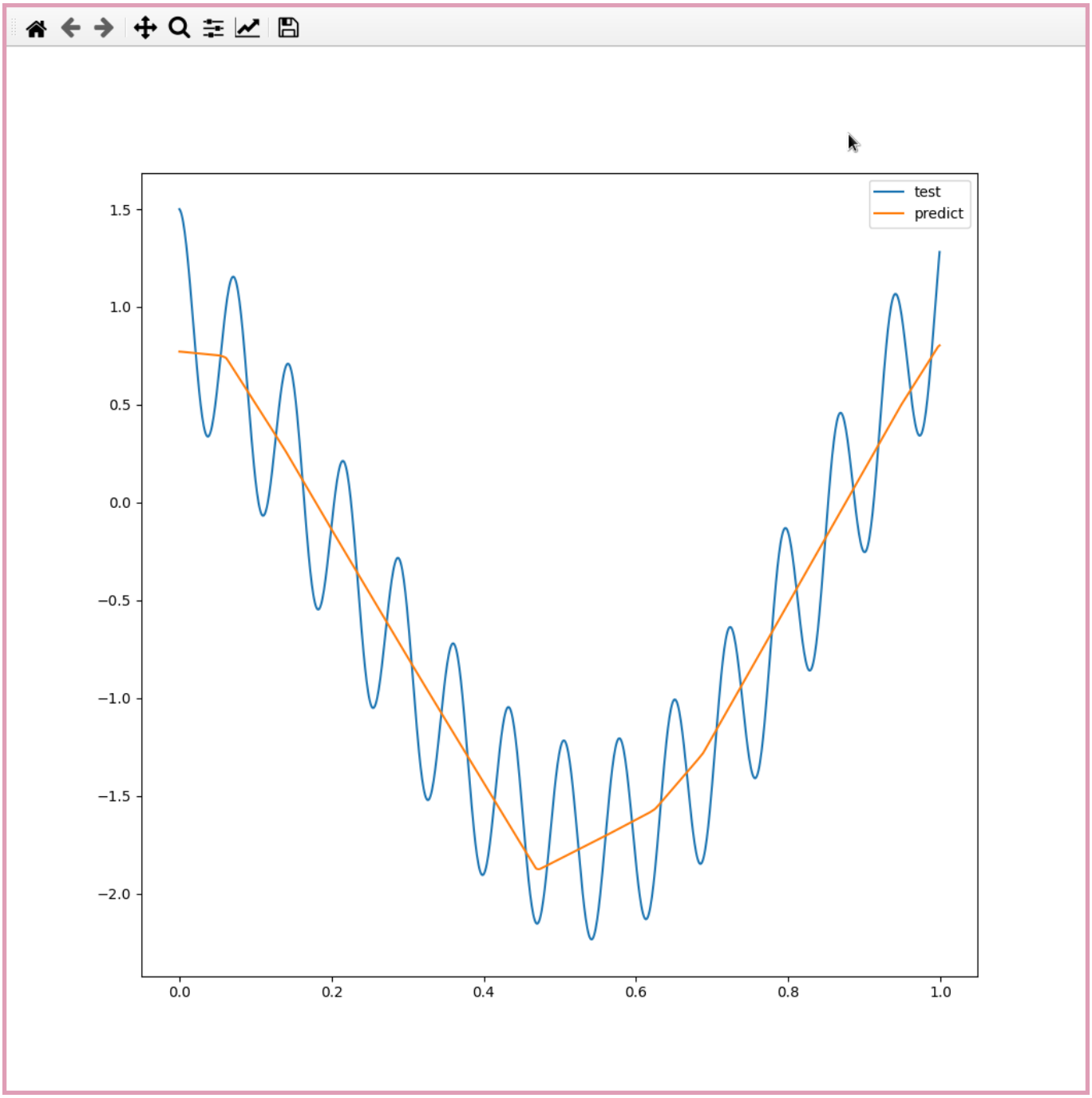


Рис. 3: Класифікація з використанням CGF

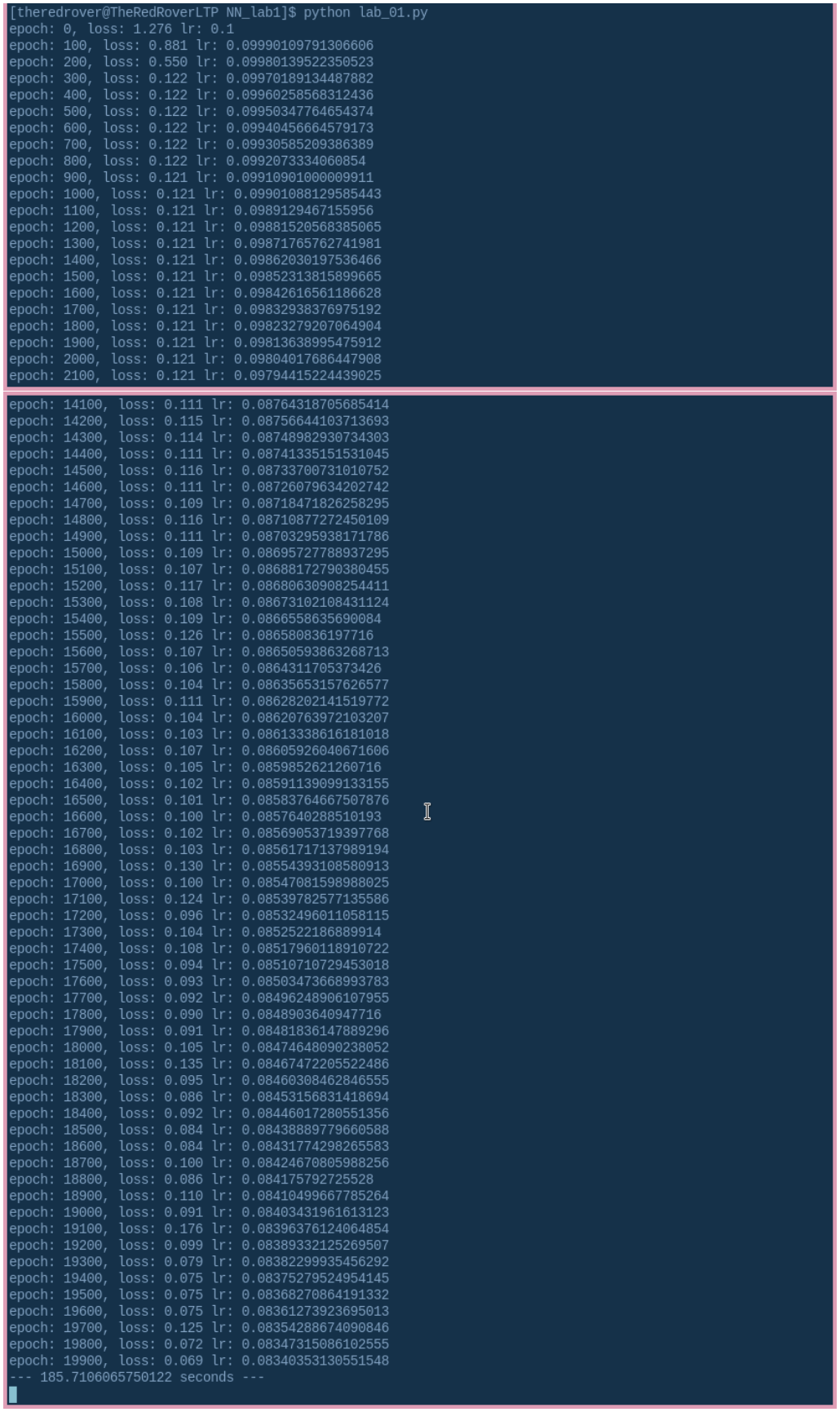


Рис. 4: Показники початкових та кінцевих епох з використанням GDM

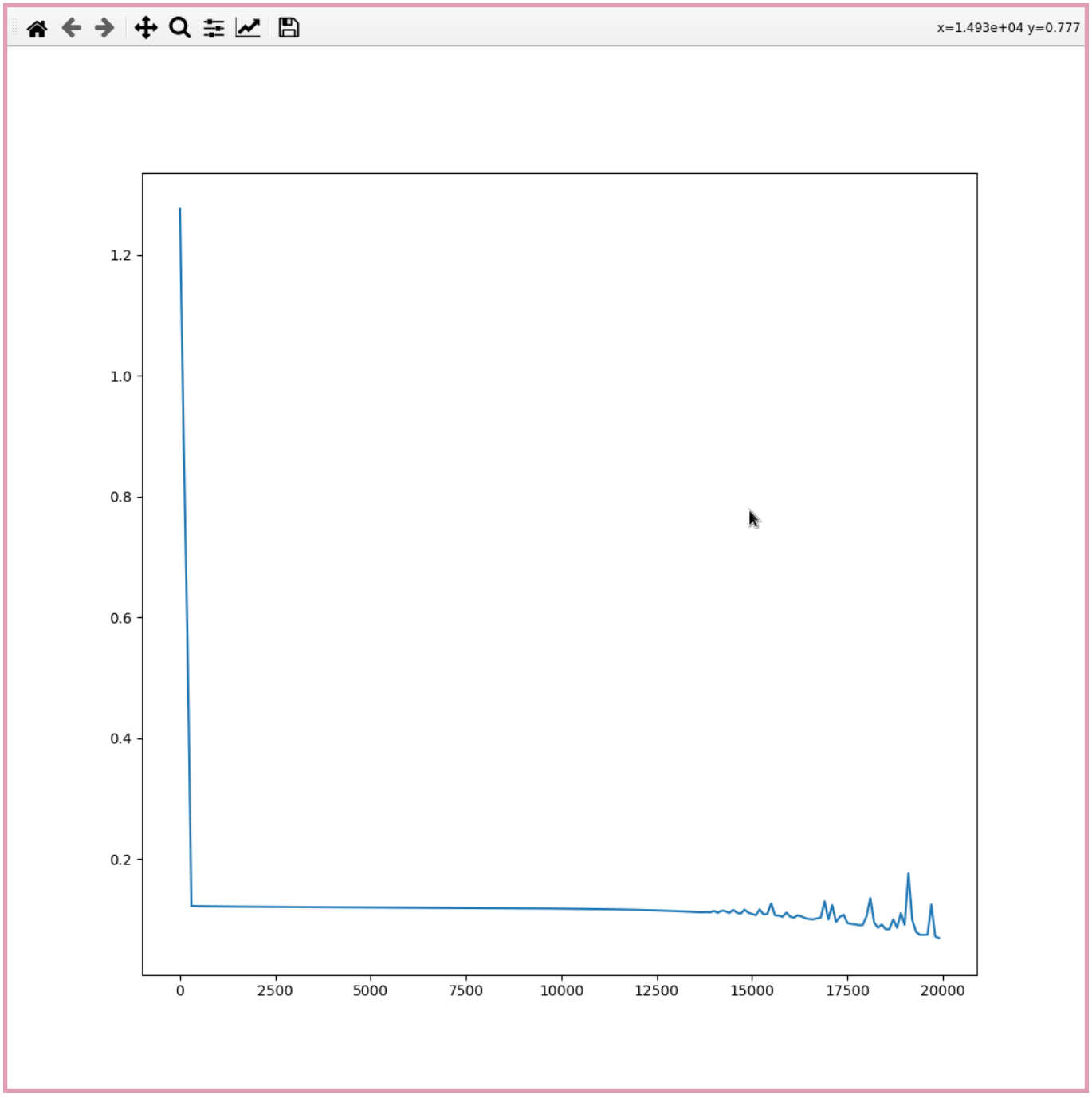


Рис. 5: Класифікація з використанням GDM

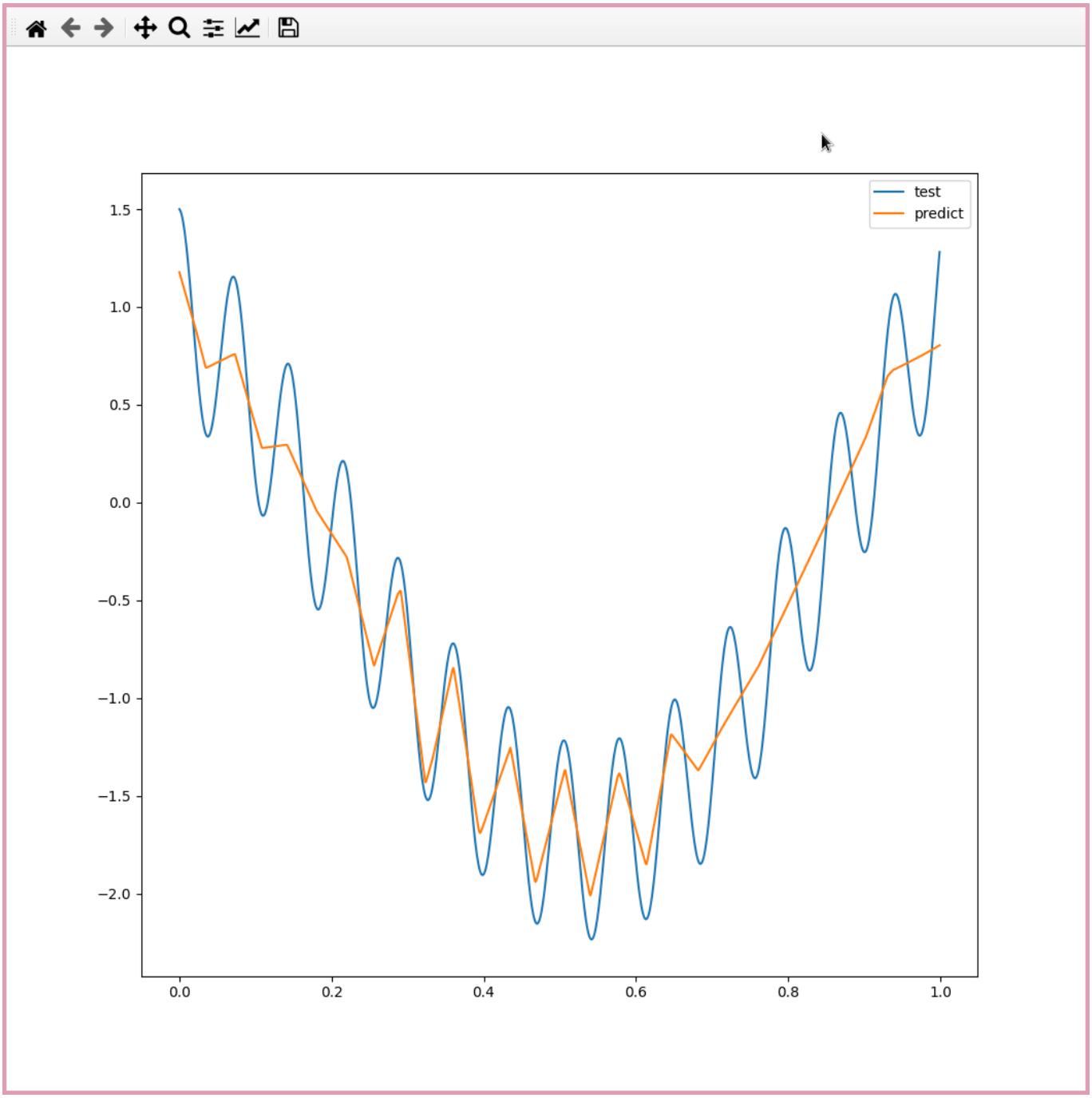


Рис. 6: Класифікація з використанням GDM

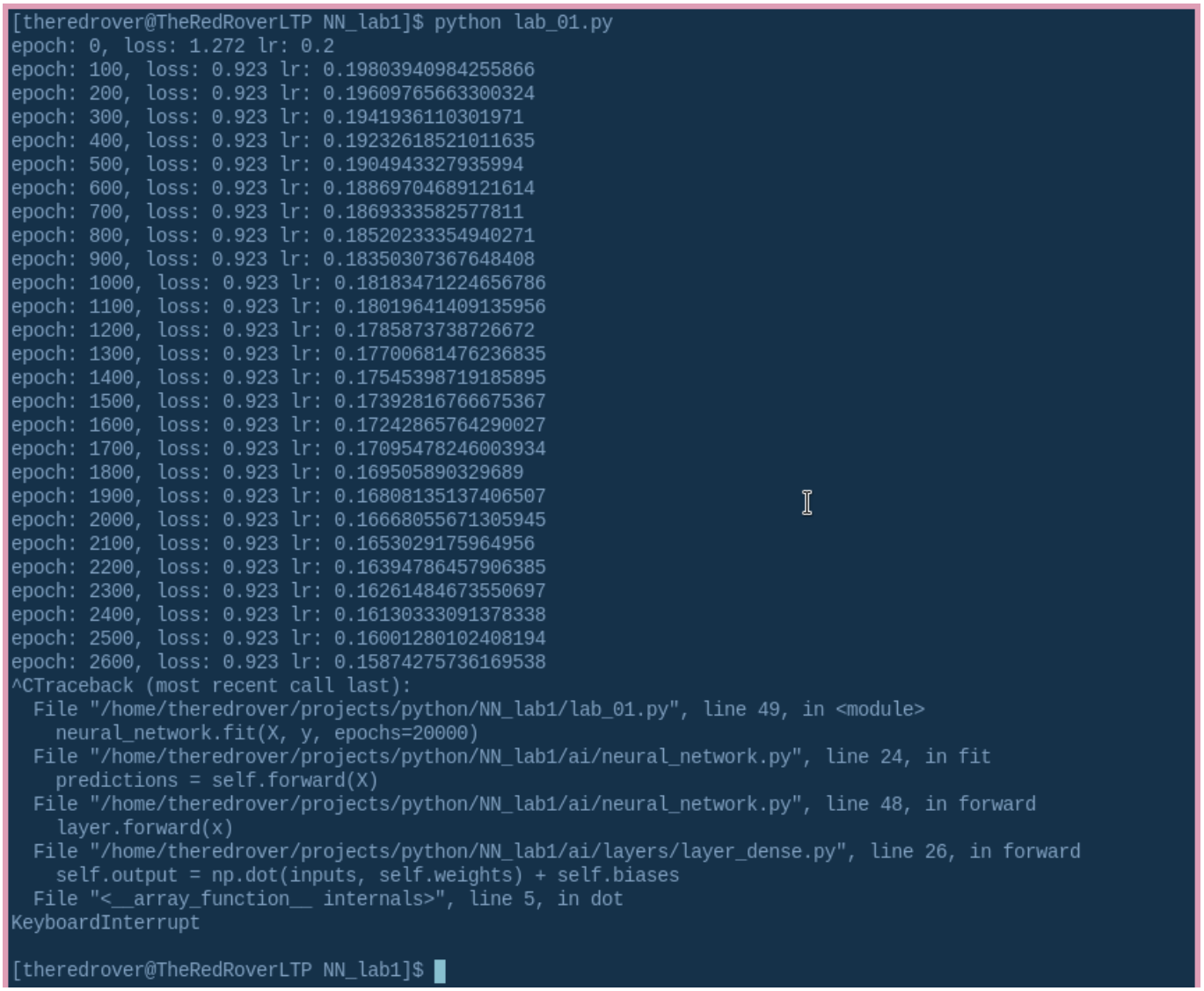


Рис. 7: Невдача при створені алгоритму BFGS

Висновки:

В ході виконання лабораторної роботи ми ознайомилися із принципами створення нейронної мережі з прямою передачею інформації. За основу був взят код із книжки The Neural Networks from Scratch. Нами були дописані необхідні алгоритми оптимізації, що використовуються всередені мереж.

Алгоритм GDM показав найкращі результати щодо швидкості та точності виконання, але, можливо, це є результатом того, що інші алгоритми вийшли неоптимізованими, та в нас є сумніви щодо правильності їх реалізації.