Міністерство освіти і науки України
Харківський національний университет 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

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 Хід роботи

1.1 Оптимізатор GDM

```
class OptimizerGDM(OptimizerAbstract):
    def __init__(self, network, learning_rate=0.001, decay=0.,
     ↪  momentum=0.):
        super(OptimizerGDM, self).__init__(network, 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)

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

        layer.weights -= weight_updates
```

```
22
23      def post_update_params(self):
24          self.iterations += 1
```

1.1.1 Результати роботи GDM

```
epoch: 0, loss: 1.271 lr: 0.1
epoch: 100, loss: 0.898 lr: 0.09990109791306606
epoch: 200, loss: 0.890 lr: 0.09980139522350523
epoch: 300, loss: 0.838 lr: 0.09970189134487882
epoch: 400, loss: 0.294 lr: 0.09960258568312436
epoch: 500, loss: 0.125 lr: 0.09950347764654374
epoch: 600, loss: 0.125 lr: 0.09940456664579173
epoch: 700, loss: 0.124 lr: 0.09930585209386389
epoch: 800, loss: 0.124 lr: 0.0992073334060854
epoch: 900, loss: 0.124 lr: 0.09910901000009911
epoch: 1000, loss: 0.124 lr: 0.09901088129585443
epoch: 1100, loss: 0.124 lr: 0.0989129467155956
epoch: 1200, loss: 0.124 lr: 0.09881520568385065
epoch: 1300, loss: 0.123 lr: 0.09871765762741981
epoch: 1400, loss: 0.123 lr: 0.09862030197536466
epoch: 1500, loss: 0.123 lr: 0.09852313815899665
epoch: 1600, loss: 0.123 lr: 0.09842616561186628
epoch: 1700, loss: 0.123 lr: 0.09832938376975192
epoch: 1800, loss: 0.123 lr: 0.09823279207064904
epoch: 1900, loss: 0.123 lr: 0.09813638995475912
epoch: 2000, loss: 0.123 lr: 0.09804017686447908
epoch: 2100, loss: 0.123 lr: 0.09794415224439025
epoch: 2200, loss: 0.123 lr: 0.09784831554124797
epoch: 2300, loss: 0.123 lr: 0.09775266620397072
epoch: 2400, loss: 0.123 lr: 0.09765720368362973
epoch: 2500, loss: 0.123 lr: 0.09756192743343838
epoch: 2600, loss: 0.123 lr: 0.0974668369087418
epoch: 2700, loss: 0.123 lr: 0.09737193156700649
epoch: 2800, loss: 0.123 lr: 0.09727721086781001
```

```
epoch: 2900, loss: 0.122 lr: 0.09718267427283064
epoch: 3000, loss: 0.122 lr: 0.09708832124583734
epoch: 3100, loss: 0.122 lr: 0.09699415125267946
epoch: 3200, loss: 0.122 lr: 0.09690016376127676
epoch: 3300, loss: 0.122 lr: 0.09680635824160931
epoch: 3400, loss: 0.122 lr: 0.09671273416570761
epoch: 3500, loss: 0.122 lr: 0.09661929100764259
epoch: 3600, loss: 0.122 lr: 0.09652602824351587
epoch: 3700, loss: 0.122 lr: 0.09643294535144986
epoch: 3800, loss: 0.122 lr: 0.09634004181157815
epoch: 3900, loss: 0.122 lr: 0.09624731710603567
epoch: 4000, loss: 0.122 lr: 0.09615477071894923
epoch: 4100, loss: 0.122 lr: 0.09606240213642782
epoch: 4200, loss: 0.122 lr: 0.09597021084655323
epoch: 4300, loss: 0.122 lr: 0.09587819633937046
epoch: 4400, loss: 0.122 lr: 0.09578635810687842
epoch: 4500, loss: 0.122 lr: 0.09569469564302051
epoch: 4600, loss: 0.122 lr: 0.09560320844367537
epoch: 4700, loss: 0.122 lr: 0.09551189600664763
epoch: 4800, loss: 0.122 lr: 0.09542075783165871
epoch: 4900, loss: 0.122 lr: 0.09532979342033765
--- 12.41968297958374 seconds ---
```

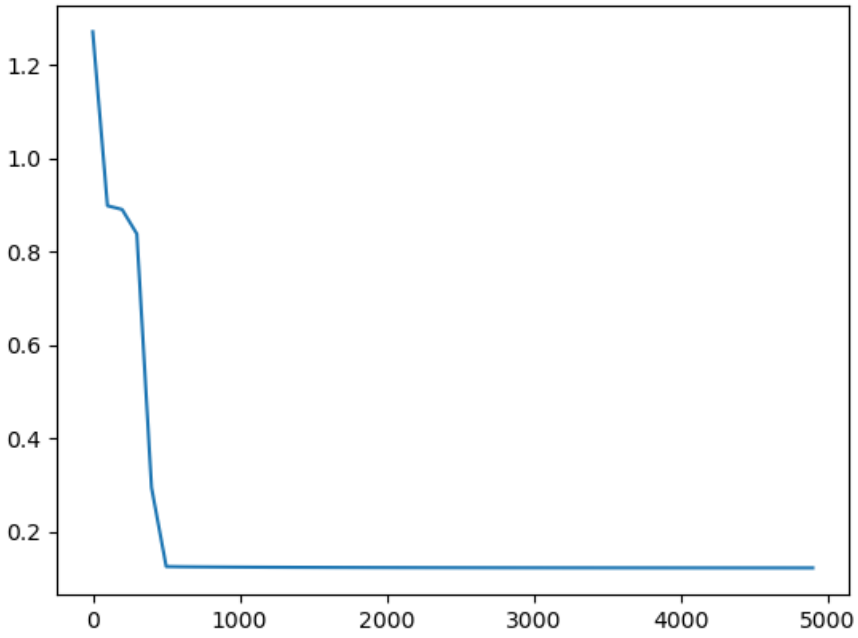

Рис. 1: Графік помилки GDM

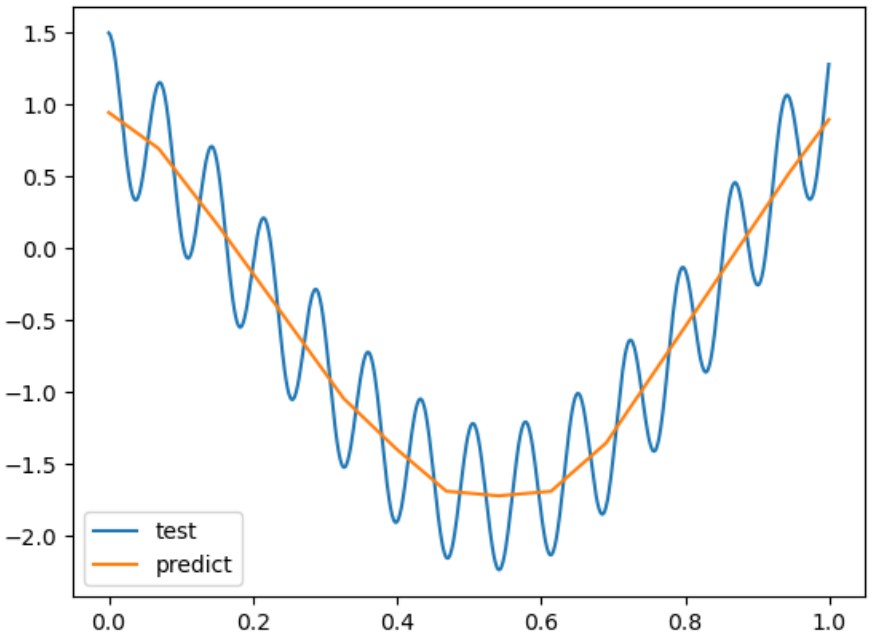

Рис. 2: Графік апроксимації функції оптимізатором GDM

1.2 Оптимізатор CGF

```
class OptimizerCGF(OptimizerAbstract):
    def __init__(self, network, learning_rate=0.001, decay=0.,
     ↪  epsilon=1e-7, max_update=10):
        super(OptimizerCGF, self).__init__(network, 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.weight_p = np.zeros_like(-layer.dweights)

        weight_update = self.current_learning_rate * layer.weight_p

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

```

```
        layer.weight_p = -layer.dweights + beta_weights *
        ↪ layer.weight_p

        layer.prev_dweights = layer.dweights.copy()

        layer.weights += weight_update

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

1.2.1 Результати роботи CGF

```
epoch: 0, loss: 1.261 lr: 0.05
epoch: 100, loss: 0.903 lr: 0.04549590536851684
epoch: 200, loss: 0.855 lr: 0.041701417848206836
epoch: 300, loss: 0.254 lr: 0.03849114703618168
epoch: 400, loss: 0.151 lr: 0.035739814152966405
epoch: 500, loss: 0.161 lr: 0.0333555703802535
epoch: 600, loss: 0.145 lr: 0.03126954346466542
epoch: 700, loss: 0.142 lr: 0.029429075927015894
epoch: 800, loss: 0.129 lr: 0.027793218454697056
epoch: 900, loss: 0.128 lr: 0.02632964718272775
epoch: 1000, loss: 0.127 lr: 0.02501250625312656
epoch: 1100, loss: 0.124 lr: 0.023820867079561697
epoch: 1200, loss: 0.120 lr: 0.02273760800363802
epoch: 1300, loss: 0.120 lr: 0.02174858634188778
epoch: 1400, loss: 0.121 lr: 0.020842017507294707
epoch: 1500, loss: 0.120 lr: 0.020008003201280513
epoch: 1600, loss: 0.119 lr: 0.019238168526356292
epoch: 1700, loss: 0.119 lr: 0.018525379770285292
epoch: 1800, loss: 0.119 lr: 0.017863522686673815
epoch: 1900, loss: 0.119 lr: 0.017247326664367024
epoch: 2000, loss: 0.119 lr: 0.016672224074691565
epoch: 2100, loss: 0.119 lr: 0.016134236850596968
epoch: 2200, loss: 0.119 lr: 0.015629884338855895
```

```
epoch: 2300, loss: 0.119 lr: 0.015156107911488331
epoch: 2400, loss: 0.119 lr: 0.014710208884966167
epoch: 2500, loss: 0.119 lr: 0.014289797084881395
epoch: 2600, loss: 0.119 lr: 0.01389274798555154
epoch: 2700, loss: 0.119 lr: 0.013517166801838336
epoch: 2800, loss: 0.119 lr: 0.013161358252171624
epoch: 2900, loss: 0.119 lr: 0.012823800974608874
epoch: 3000, loss: 0.119 lr: 0.012503125781445364
epoch: 3100, loss: 0.119 lr: 0.012198097096852892
epoch: 3200, loss: 0.119 lr: 0.011907597046915934
epoch: 3300, loss: 0.119 lr: 0.011630611770179114
epoch: 3400, loss: 0.119 lr: 0.011366219595362582
epoch: 3500, loss: 0.119 lr: 0.011113580795732384
epoch: 3600, loss: 0.119 lr: 0.010871928680147858
epoch: 3700, loss: 0.119 lr: 0.010640561821664184
epoch: 3800, loss: 0.119 lr: 0.010418837257762034
epoch: 3900, loss: 0.119 lr: 0.010206164523372118
epoch: 4000, loss: 0.119 lr: 0.010002000400080015
epoch: 4100, loss: 0.119 lr: 0.009805844283192783
epoch: 4200, loss: 0.119 lr: 0.009617234083477593
epoch: 4300, loss: 0.119 lr: 0.009435742592942064
epoch: 4400, loss: 0.119 lr: 0.009260974254491572
epoch: 4500, loss: 0.119 lr: 0.009092562284051647
epoch: 4600, loss: 0.119 lr: 0.00893016610108948
epoch: 4700, loss: 0.119 lr: 0.008773469029654325
epoch: 4800, loss: 0.119 lr: 0.00862217623728229
epoch: 4900, loss: 0.119 lr: 0.008476012883539583
--- 29.128910303115845 seconds ---
```

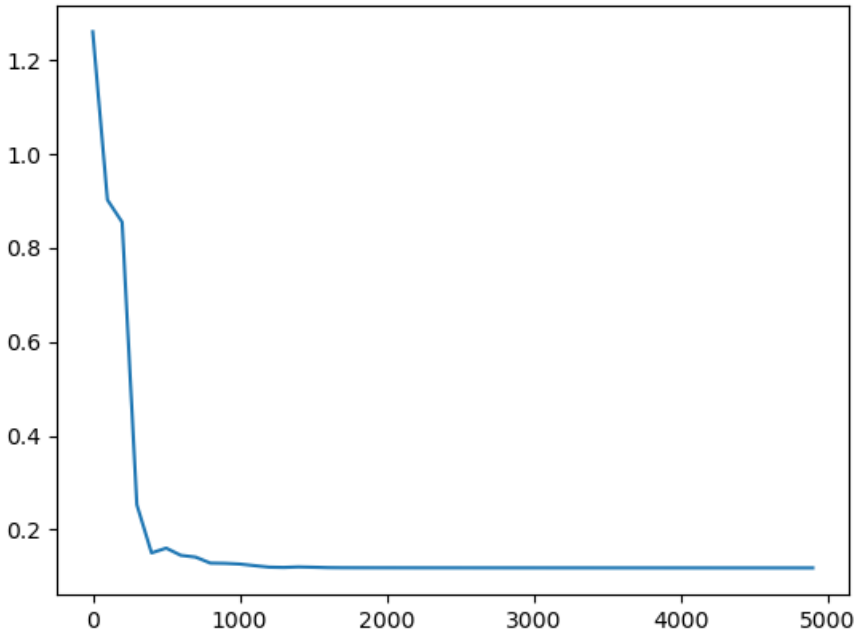

Рис. 3: Графік помилки CGF

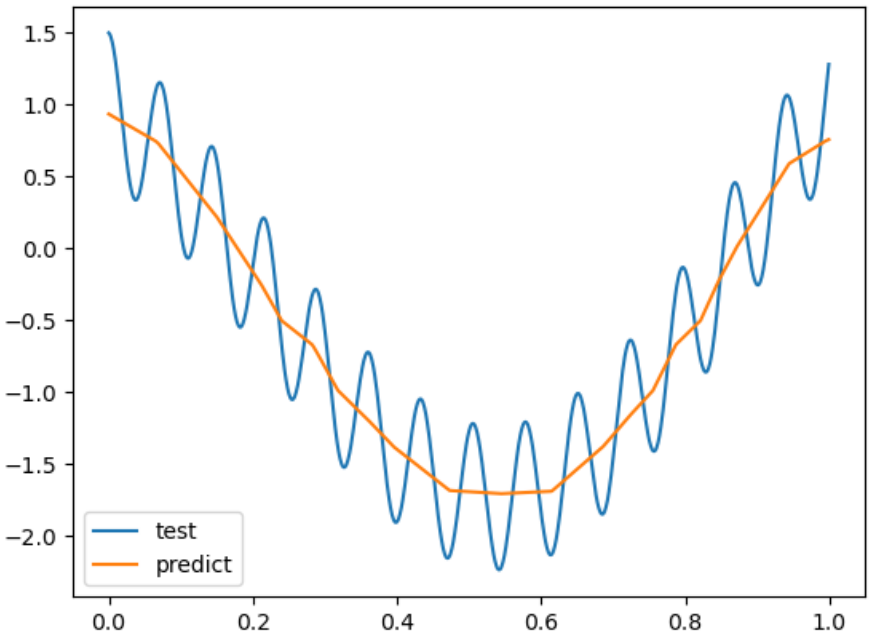

Рис. 4: Графік апроксимації функції оптимізатором CGF

1.3 Оптимізатор BFGS

```
class OptimizerBFGS(OptimizerAbstract):
    def __init__(self, network, learning_rate=0.001, decay=0.,
     ↪  epsilon=1e-7):
        super(OptimizerBFGS, self).__init__(network, learning_rate)
        self.decay = decay
        self.iterations = 0
        self.epsilon = epsilon
        self.loss_func = None

    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):
        flat_weights = layer.weights.flatten()
        flat_dweights = layer.dweights.flatten()

        I = np.eye(flat_weights.shape[0])
        if not hasattr(layer, f'prev_weights'):
            layer.H = I
            layer.prev_weights = np.zeros_like(flat_weights)
            layer.prev_dweights = np.zeros_like(flat_dweights)
            layer.Bs =
             ↪  layer.prev_weights.dot(-self.current_learning_rate)

        sk = (flat_weights -
         ↪  layer.prev_weights).reshape((flat_weights.shape[0], 1))
        yk = (flat_dweights -
         ↪  layer.prev_dweights).reshape((flat_dweights.shape[0], 1))

        eps = 1e-1
```

```
        ys = yk.T.dot(sk)
        sBs = sk.T.dot(layer.Bs)

        # powell damping
        if ys < eps * sBs:
            theta = ((1 - eps) * sBs) / (sBs - ys)
            yk = (theta * yk.flatten() + (1 - theta) * layer.Bs).T

        rho_inv = sk.flatten() @ yk.flatten()
        if abs(rho_inv) < 0.00001:
            rho = 1000
        else:
            rho = 1 / rho_inv

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

        direction = -layer.H @ flat_dweights

        alpha, fail = weak_wolfe(layer, self.loss_func, direction,
         ↪  flat_dweights, self.current_learning_rate)
        if fail:
            pass
        else:
            pass

        layer.Bs = layer.prev_weights.dot(-alpha)

        weight_update = alpha * direction

```

```
60          layer.weights +=
            ↪  weight_update.reshape((layer.weights.shape[0],
            ↪  layer.weights.shape[1]))
61
62          layer.prev_weights = layer.weights.flatten()
63          layer.prev_dweights = layer.dweights.flatten()
64
65      def fit(self, x, y, epochs=10000, loss_function=None):
66          self._validate_fit(loss_function)
67
68          for epoch in range(epochs):
69              def loss_func():
70                  predictions = self.network.forward(x)
71                  loss =
                    ↪  self.network.loss_function.calculate(predictions,
                    ↪  y)
72                  self.network.backward(predictions, y)
73                  return loss, predictions
74
75              self.loss_func = loss_func
76              loss, predictions = loss_func()
77
78              if (epoch % 100) == 0:
79                  print(f'epoch: {epoch}, ' +
80                        f'loss: {loss:.3f} ' +
81                        f'lr: {self.current_learning_rate}')
82
83                  self.losses[epoch] = loss
84
85              self.update_weights()
86
87          return self
88
89      def post_update_params(self):
```

```
90            self.iterations += 1
```

1.3.1 Результати роботи BFGS

```
epoch: 0, loss: 1.295 lr: 1
epoch: 100, loss: 0.142 lr: 0.5025125628140703
epoch: 200, loss: 0.130 lr: 0.33444816053511706
epoch: 300, loss: 0.128 lr: 0.2506265664160401
epoch: 400, loss: 0.127 lr: 0.2004008016032064
epoch: 500, loss: 0.126 lr: 0.1669449081803005
epoch: 600, loss: 0.126 lr: 0.14306151645207438
epoch: 700, loss: 0.126 lr: 0.1251564455569462
epoch: 800, loss: 0.126 lr: 0.11123470522803114
epoch: 900, loss: 0.126 lr: 0.10010010010010009
--- 2660.8307535648346 seconds ---
```

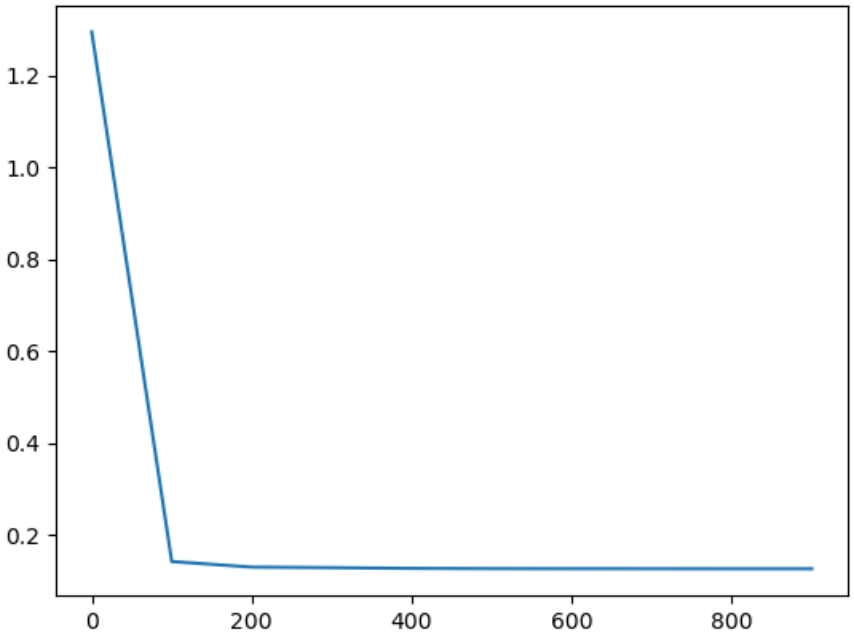

Рис. 5: Графік помилки BFGS

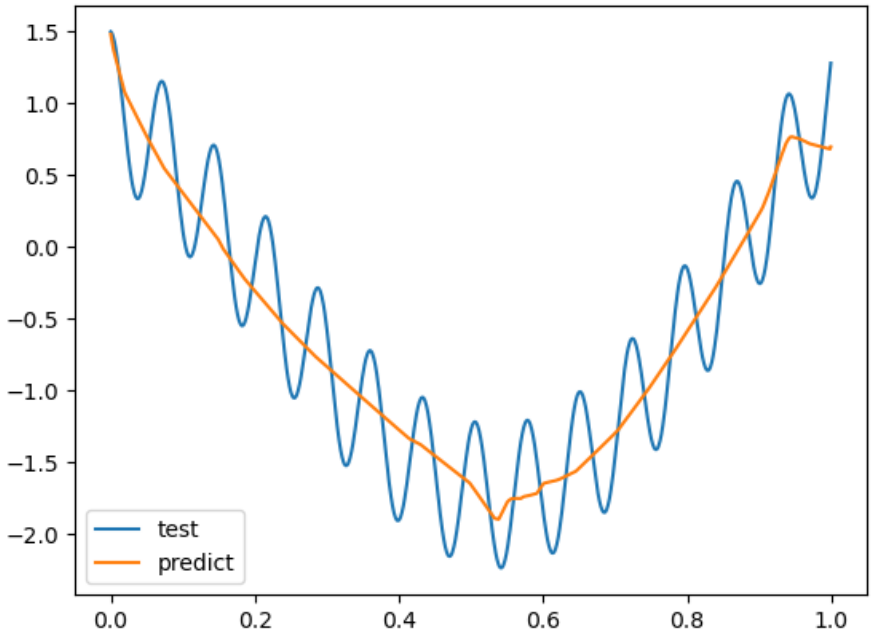

Рис. 6: Графік апроксимації функції оптимізатором BFGS

Висновок

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. BFGS є найповільнішим алгоритмом, але сходиться за найменьшу кількість епох, проте складність реалізації та досягнення такої самої точності, що і простіші алгоритми робить його поганим алгоритмом для цієї задачі.