Homework 4

CS 559 - Neural Networks - Fall 2019

Matteo Corain 650088272

November 4, 2019

1 Question 1

1.1 Training set generation

According to the specifications of the exercise, the training set for the considered neural network has been generated as a set of 300 unidimensional samples $x_i, i = 1, \ldots, 300$, drawn at random in $[0, 1]$; this was accomplished by means of the `uniform()` function provided by the NumPy library and the obtained samples were stored in the `x` array. A set of noise samples ν_i was also created, by drawing at random some values in $\left[-\frac{1}{10}, \frac{1}{10}\right]$ and storing them in array `n`. Starting from these two arrays, the expected outputs have been generated by means of the function that the considered network needs to approximate, expressed using the functions provided by the NumPy library and stored in the array `d`:

$$d_i = \sin(20x_i) + 3x_i + \nu_i$$

1.2 Feed-forward procedure

The backpropagation training algorithm has been implemented using the matricial form presented in class. Specifically, let us consider a network with $L \geq 1$ layers, indicating as m_i the number of neurons in the i-th layer. Each layer i of the network receives an input y_{i-1} and produces an output y_i by applying its activation function $\phi_i(v_i)$ to its induced field $v_i = W_i \begin{bmatrix} 1 & y_{i-1} \end{bmatrix}^T$, where W_i is the weight matrix of the i-th layer.

The feedforward phase of the algorithm may therefore be implemented by repeating L times (one for each layer) the following operations:

- Compute the local induced field:
$$v_i = W_i \begin{bmatrix} 1 \\ y_{i-1} \end{bmatrix}$$
- Compute the layer output:
$$y_i = \phi_i(v_i)$$

For the first layer of the network, the input y_0 is the input x of the network. The feedforward procedure may be implemented using algorithm 1.

Algorithm 1 Feed-forward procedure

function FEED-FORWARD(x_i, W, ϕ)

▷ Initialization

$v \leftarrow$ EMPTY-LIST
$y \leftarrow$ EMPTY-LIST
$y[0] \leftarrow x_i$

▷ Computation of v and y values

for $i = 1, \ldots, L$ **do**
$v[i] \leftarrow W[i] \begin{bmatrix} 1 & y[i] \end{bmatrix}^T$
$y[i+1] \leftarrow \phi[i](v[i])$
end for
return v, y
end function

1.3 Backpropagation procedure

The partial results coming from the v and y vectors can then be used to compute the update terms for the gradient descent algorithm (to be precise, to estimate the value of the derivatives $\frac{\partial E}{\partial W_i}, \forall i$). Let us define the quantities $\delta_i, i = 1, \ldots, L$ as:

$$\delta_i = \begin{cases} (d - y_L) \cdot \phi'(v_L) & \text{if } i = L \\ \underline{W_{i+1}^T} \delta_{i+1} \cdot \phi'(v_i) & \text{if } i = 1, \ldots, L-1 \end{cases}$$

Where the $\cdot$ symbol denotes a dot product operation and $\underline{W_i}$ denotes the W_i matrix to which the first row has been removed (the one related to the bias contributions). The theoretical results of the backpropagation algorithm prove that, once the δ_i quantities are defined, it is possible to compute the partial derivatives of E as:

$$\frac{\partial E}{\partial W_i} = -\delta_i \begin{bmatrix} 1 \\ y_{i-1} \end{bmatrix}$$

This suggests the procedure shown in algorithm 2 to implement the backpropagation mechanism.

1.4 Error procedure

The error procedure is used to compute the value of the error function, given the current weights, the activation functions, the training samples and the associated desired outputs. Let us consider the error function for this exercise to be defined as the mean squared error of the classification task:

$$E = \frac{1}{n} \sum_{i=1}^{n} ||d_i - y_{i,L}||^2$$

Where n is the number of training samples ($n = 300$, in this case). Algorithm 3 shows a possible implementation of the error procedure.

Algorithm 2 Backpropagation procedure

function BACKPROPAGATE(x_i, d_i, W, ϕ, ϕ')

▷ Initialization

$v, y \leftarrow$ FEED-FORWARD(x_i, W, ϕ)
$\delta \leftarrow$ EMPTY-LIST
$\nabla E \leftarrow$ EMPTY-LIST

▷ Computation of δ_i values

$\delta[L] \leftarrow (d_i - y[L]) \cdot \phi'[L](v[L])$
for $i = L - 1, \ldots, 1$ **do**
$\delta[i] \leftarrow \left(\underline{W[i+1]}\right)^T \delta[i+1] \cdot \phi'[i](v[i])$
end for

▷ Computation of ∇E components

for $i = 1, \ldots, L$ **do**
$\nabla E[i] = -\delta[i] \begin{bmatrix} 1 & y_{i-1} \end{bmatrix}^T$
end for
return v, y
end function

Algorithm 3 Error procedure

function ERROR(x, d, n, W, ϕ)
$e \leftarrow 0$
for $i = 1, \ldots, n$ **do**
$v, y \leftarrow$ FEED-FORWARD($x[i], W, \phi$)
$e \leftarrow e + (d[i] - y[L])^2$
end for
return e/n
end function

1.5 Training procedure

The training procedure is the one that is effectively used to train the network, updating the weights starting from the results of the backpropagation procedure. It implements a standard gradient descent method, in which weights are updated for each sample as:

$$W_i \leftarrow W_i - \eta \frac{\partial E}{\partial W_i}, i = 1, \ldots, L$$

In which the derivative term is computed through the backpropagation procedure. This can be implemented as shown in algorithm 4, which uses as stopping criteria two conditions:

- The value of the energy function drops below some value ϵ;
- The maximum number of epochs λ is reached.

It has been chosen to stop the epochs when the value of the error function falls below a certain threshold instead of considering the decrease of energy from an epoch to the following because of

Algorithm 4 Training procedure

function TRAIN($x, d, W, \phi, \phi', n, \eta, \epsilon, \lambda$)

▷ Initialization

$epochs \leftarrow 0$
$errors \leftarrow$ EMPTY-LIST
$errors[0] \leftarrow$ ERROR(x, d, W, ϕ)

▷ Main weights update loop

while $epochs < \lambda \wedge errors[epochs] > \epsilon$ **do**
$epochs \leftarrow epochs + 1$

▷ Compute update terms and update weights

for $i = 1, \ldots, n$ **do**
$\nabla E \leftarrow$ BACKPROPAGATE($x[i], d[i], W, \phi, \phi'$)
for $j = 1, \ldots, L$ **do**
$W[j] \leftarrow W[j] - \eta \nabla E[j]$
end for
end for

▷ Error registration

$errors[epochs] \leftarrow$ ERROR(x, d, W, ϕ)

▷ Update of η

if $errors[epochs] > errors[epochs - 1]$ **then**
$\eta \leftarrow 0.9\eta$
end if
end while
return v, y
end function

the unpredictability of the values of the latter. There is no guarantee that a small delta would be obtained when the fit is already good, or vice versa: as it will be shown in the energy graph, in fact, in the considered running example the profile of the energy function presents multiple flat zones, in which the energy decrease is minimal (up to 10^{-8}, for some iterations), also for relatively high values of the energy function. This justifies the usage of the value of the error function to define the stopping condition of the algorithm rather than its relative difference, since otherwise there would be the possibility for the algorithm to stop in correspondence of one of the aforementioned high-energy plateaus.

This algorithm also implements the η update rule, which reduces the learning rate by a factor of 0.9 when it detects that the error increases from an epoch to the next.

1.6 Implementation details

The described algorithms were implemented using the Python programming language in conjunction with the NumPy scientific computing library. In particular, we have that:

- Procedure FEED-FORWARD is implemented in the `feedforward()` function;
- Procedure BACKPROPAGATE is implemented in the `backpropagate()` function;

- Procedure ERROR is implemented in the `error()` function;
- Procedure TRAIN is implemented in the `train()` function.

The initial weights matrices `W1` and `W2` for the network have been chosen according to the weight initialization heuristic, which prescribes to draw the starting values from a normal distribution, with mean $\mu = 0$ and variance $\sigma^2 = \frac{1}{m}$, where m is the number of inputs for the neurons in the considered layer (i.e. the number of neurons in the previous layer). In our case, we have $m_1 = 2$ and $m_2 = 25$, so reasonable values for the standard deviation are:

$$\sigma_1 = \sqrt{\frac{1}{2}} = 0.7071, \sigma_2 = \sqrt{\frac{1}{25}} = 0.2$$

As for the training parameters, the following values have been chosen:

$$\eta = 0.1, \epsilon = 0.01, \lambda = 5000$$

The training procedure has then been executed, passing it:

- The `x` and `d` vectors generated as explained before;
- The lists `W = [W1, W2]`, `phi = [phi1, phi2]` and `der_phi = [der_phi1, der_phi2]`, containing respectively the two weight matrices, the two activation functions and their two derivatives (passed as function pointers);
- The three training parameters `eta = 0.1`, `eps = 0.01` and `epoch_limit = 5000`.

1.7 Results

The training algorithm was able to converge to a good enough fit in 4085 epochs. Results have been plotted by making use of the `plot_fit()` and `plot_errors()` functions. The first is used to plot the function fitted from the generated dataset, obtained by evaluating the output of the network (through the `feedforward()` function) on a evenly-spaced array of x values (generated using the `linspace()` function of the NumPy library). The second simply plots the contents of the passed `errors` list for increasing number of epochs. The obtained fit is shown in figure 1; figure 2 shows instead the value of the error function for increasing epochs.

It is possible to notice that the fitting curve results to be quite close to the data points, without exhibiting any major overfitting-related problem (e.g. oscillations in the vicinity of data points). As expected, the most prominent improvements in terms of the reduction of the error function are shown in the first epochs, while the profile becomes almost flat when the number of epochs increases (when the reduction is limited also by the decreasing value of η).

1.8 Complete Python code

```
import numpy as np
import matplotlib.pyplot as plt

# Use TeX for text rendering
plt.rc('text', usetex=True)
```

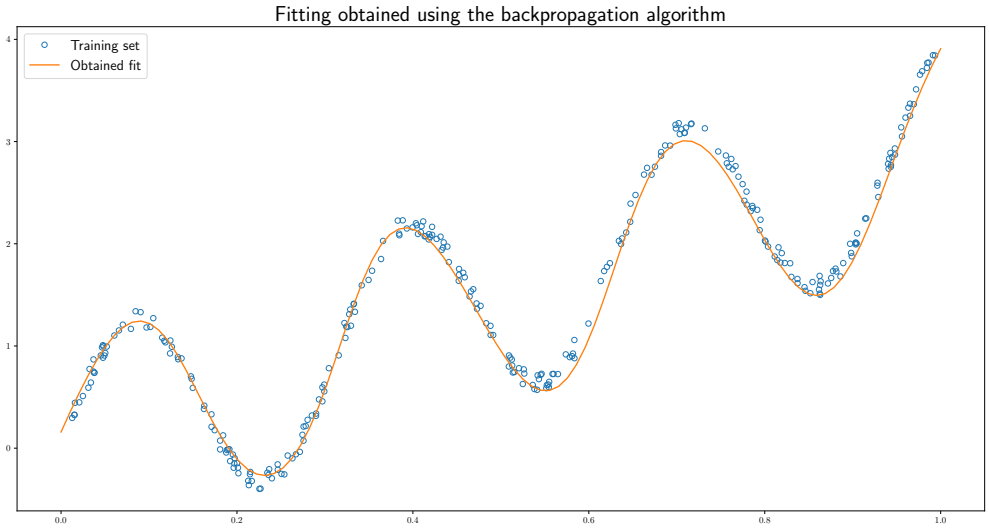

Figure 1: Fitting obtained using backpropagation and $\epsilon = 0.01$

```
# Set the random seed for reproducibility
np.random.seed(2019)

# Activation functions
phi1 = lambda v : np.tanh(v)
der_phi1 = lambda v : 1 - np.tanh(v) ** 2
phi2 = lambda v : v
der_phi2 = lambda v : 1

def feedforward(x, W, phi, i):
    v = []
    y = [np.array(x[i]).reshape(1,1)]

    # Compute local fields and outputs
    for j in range(len(W)):
        v.append(W[j] @ np.concatenate(([1], y[j]), axis=None).reshape(len(y[j]) + 1, 1))
        y.append(phi[j](v[j]).reshape(len(v[j]), 1))

    return v, y

def backpropagate(x, d, W, phi, der_phi, i):
    # Compute local fields and outputs
    v, y = feedforward(x, W, phi, i)
    delta = [0] * len(W)
    dEW = [0] * len(W)

    # Compute delta signals
    delta[len(W) - 1] = (d[i] - y[len(W)]) * der_phi[len(W) - 1](v[len(W) - 1])
    for j in reversed(range(len(W) - 1)):
        delta[j] = W[j + 1].transpose()[1:] @ delta[j + 1] * der_phi[j](v[j])

    # Compute the gradient
```

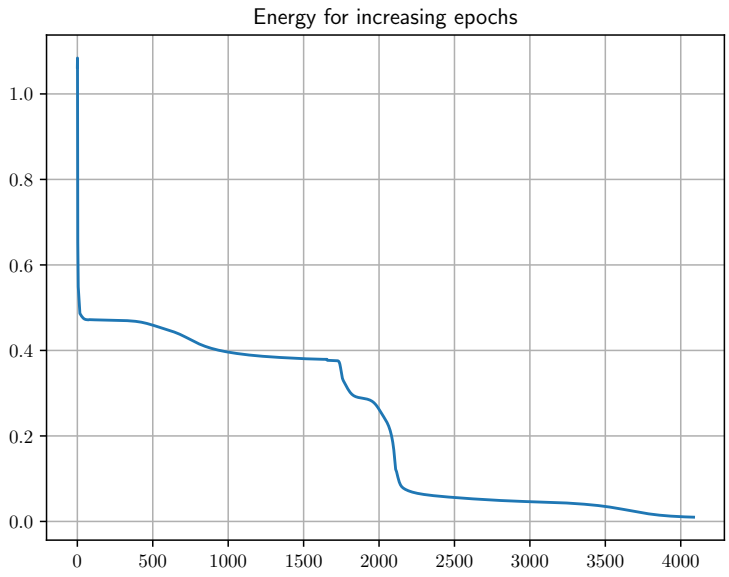

Figure 2: Error for increasing number of epochs ($\epsilon = 0.01$)

```
    for j in range(len(W)):
        dEW[j] = -delta[j] @ np.concatenate(([1], y[j]), axis=None).reshape(1, len(y[j]) + 1)

    return dEW

def error(x, d, W, phi):
    return sum([(d[i] - feedforward(x, W, phi, i)[1][-1].item()) ** 2 for i in range(len(x))]) / len(x)

def train(x, d, W, phi, der_phi, eta, eps, epoch_limit):
    # Run the gradient descent method
    epochs = 0
    errors = [error(x, d, W, phi)]

    while epochs < epoch_limit and errors[epochs] >= eps:
        # Increment epochs
        epochs = epochs + 1

        # Update weights
        for i in range(len(x)):
            dEW = backpropagate(x, d, W, phi, der_phi, i)
            for j in range(len(dEW)):
                W[j] -= eta * dEW[j]

        # Register the error
        errors.append(error(x, d, W, phi))
        print("Epoch {}: eta = {}, err = {}".format(epochs, eta, errors[epochs]))

        # Decrease eta if necessary
        if errors[epochs] > errors[epochs - 1]:
            eta = 0.9 * eta

    return epochs, errors
```

```
def plot_fit(x, d, W, phi):
    x_eval = np.linspace(0, 1, 100)
    y_eval = [feedforward(x_eval, W, phi, i)[1][-1].item() for i in range(len(x_eval))]

    plt.figure()
    plt.plot(x, d, linestyle="none", marker="o", fillstyle="none")
    plt.plot(x_eval, y_eval)
    plt.title("Fitting obtained using the backpropagation algorithm")
    plt.legend(["Training set", "Obtained fit"])

def plot_errors(errors):
    plt.figure()
    plt.plot(errors)
    plt.title("Errors for increasing epochs")
    plt.grid()

# Generate the training set
n_samples = 300
x0 = 0
xn = 1
x = np.random.uniform(x0, xn, size=n_samples)
n = np.random.uniform(-1/10, 1/10, size=n_samples)
d = np.sin(20 * x) + 3 * x + n

# Define network initial weights
n_hidden = 24
sigma1 = np.sqrt(1/2)
sigma2 = np.sqrt(1 / (n_hidden + 1))
W1 = np.random.normal(0, sigma1, size=(n_hidden, 2))
W2 = np.random.normal(0, sigma2, size=(1, n_hidden + 1))

# Run the training algorithm
eta = 0.1
eps = 0.01
epoch_limit = 5000

epochs, errors = train(x, d, [W1, W2], [phi1, phi2], [der_phi1, der_phi2], eta, eps, epoch_limit)

# Plot the fitting and the errors
plot_fit(x, d, [W1, W2], [phi1, phi2])
plot_errors(errors)

# Show plots
plt.show()
```