Danmarks Tekniske Universitet

DTU

Perceptron and MLP Programming exercises

Bio-inspired control for robots - 31377

Authors

Filippo Bentivoglio - s210299
Ilknur Kasapoglu- s203030
Júlia Rey Vilches - s210194

January 9, 2022

Contents

1 Exercise 1.1 Perceptron

1. The *SignActivation* class is filled in the template. It contains the *forward* function, which in this case returns the output of the activation function. For the perceptron, the gradient is not defined.

```
1
2  class SignActivation(ActivationFunction):
3     """
4     Sign activation: `f(x) = 1 if x > 0, 0 if x <= 0`
5     """
6     def forward(self, x):
7        """
8        Function output.
9        """
10       if x > 0: return 1
11       return 0
12
13    def gradient(self, x):
14       """
15       Function derivative.
16       """
17       return 0
```

2. Next, we implement the initialization of a perceptron, defining the weights and activation function as seen in the following block of code.

```
1     def __init__(self, n_inputs, act_f):
2        """
3        Perceptron class initialization
4        """
5        if not isinstance(act_f,ActivationFunction) or not issubclass(type(
            act_f), ActivationFunction):
6           raise TypeError('act_f has to be a subclass of ActivationFunction (
               not a class instance).')
7
8        self.w = np.random.normal(size=(n_inputs+1,1)) # Define weights taking
             into account the bias w0
9        self.f = act_f # Define activation function
10
11       if self.f is not None and not isinstance(self.f, ActivationFunction):
12          raise TypeError("self.f should be a class instance.")
```

3. The *activation*, *output* and *predicted* functions are defined. As seen in the code, the *predic* function is implemented by combining the *activation* and *output* functions.

```
def activation(self, x):
    """
    Computes the activation `a` given input `x`
    """
    return self.w.T @ x


def output(self, a):
    """
    Computes the neuron output `y`, given the activation `a`
    """
    return self.f.forward(a)


def predict(self, x):
    """
    Computes the neuron output `y`, given the input `x`
    """
    out = self.output(self.activation(x))
    return out
```

To check that the *predict* function works, we tested it by computing the prediction on each of the data points given in x. At this point, the predictions are completely random.

4. The following code has been written to use the previous learning rule for the weight update.

```
mu = 0.1
perc = Perceptron(2,SignActivation())
notConverged = True

while notConverged:
    y_hat = []
    for i in range(len(data)):

        x_i = x[i]
        t_i = t[i]
        out = perc.predict(x_i)

        while out != t_i:
            perc.updateW(x_i,t_i,out,mu)
            out = perc.predict(x_i)

        notConverged = False
```

```
        y_hat.append(out)

    for i in range(len(data)):
        if perc.predict(x[i]) != t[i]:
            notConverged = True
            break

print("\nFinal weights :\n{} \n".format(perc.w))
```

In this case, 14 training epochs are required to reach convergence of the weights.

5. The final weight values are: [-0.125, -0.689, 0.354]. However, they do not reach the same final values for different initial values. This is because the initial values directly influence the output result, which is from where the final weights are updated.

6. Figure 1 presents a plot of the data points in two different colors, together with the line that separates them, as defined by the learned weights.

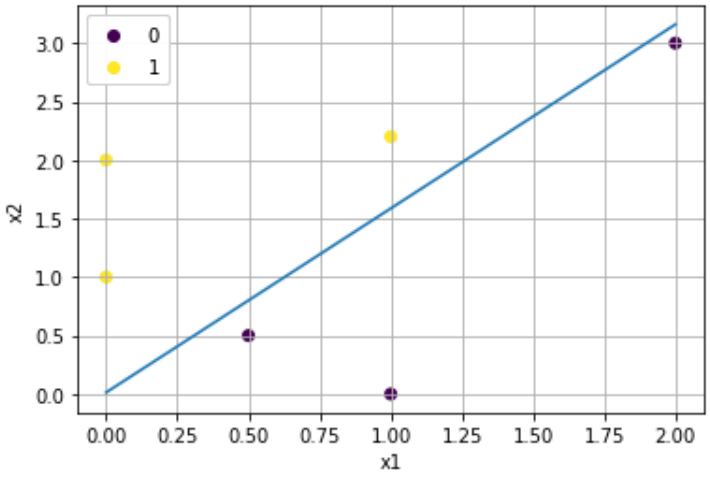

Figure 1: Classification of the data points using the perceptron model.

For this dataset, the line that separates the final classification outcome is defined by the following equation:

$$x_2 = 1.57x_1 + 0.0139 \tag{1}$$

2 Exercise 1.2 Multi-layer perceptron

1. First, the *Sigmoid* and *LinearActivation* classes are filled in accordingly.

```
class Sigmoid(ActivationFunction):
    """
```

```
    Sigmoid activation: `f(x) = 1/(1+e^(-x))`
    """
    def forward(self, x):
        """
        Activation function output.
        """
        return 1/(1+np.exp(-x))[0]

    def gradient(self, x):
        """
        Activation function derivative.
        """
        return self.forward(x) * (1 - self.forward(x)

class LinearActivation(ActivationFunction):
    """
    Linear activation: `f(x) = x`
    """
    def forward(self, x):
        """
        Activation function output.
        """
        return x

    def gradient(self, x):
        """
        Activation function derivative.
        """
        return 1
```

For this multi-layer perceptron, the sigmoid function will act as the activation function used for the hidden layer and the linear activation in the output layer.

To test the new activations function, the given dataset is being used

```
data = np.array( [ [0.5, 0.5, 0], [1.0, 0, 0], [2.0, 3.0, 0], [0, 1.0,
    1], [0, 2.0, 1], [1.0, 2.2, 1] ] )
x = data[:,:2]
act_f = Sigmoid()
perc = Perceptron(2,act_f)
for el in x:
    y = perc.predict(el)
    print(y)
```

The outcomes are showed in 2

```
Data point: [0.5 0.5]   -->     prediction: 0.5343
Data point: [1. 0.]     -->     prediction: 0.4586
Data point: [2. 3.]     -->     prediction: 0.7291
Data point: [0. 1.]     -->     prediction: 0.6084
Data point: [0. 2.]     -->     prediction: 0.7071
Data point: [1.  2.2]   -->     prediction: 0.6907
```

Figure 2: Outcomes given by a perceptron with a sigmoid function.

2. Inside the *Layer* class, we implement the initialization of each perceptron in the layer, in the *init* function.

```
class Layer:
   def __init__(self, n_inputs, n_units, act_f):
      """ Initialize the layer, creating `n_units` perceptrons with `
         num_inputs` inputs each. """
      self.n_inputs = n_inputs
      self.n_units = n_units

      self.ps = [Perceptron(self.n_inputs,act_f) for i in range(self.n_units
         )]

      ...
```

Code to initialize a 5 neurons layer and its respective outcomes, 3 when $[\pi, s1]$ was given as input.

```
l = Layer(2,5,Sigmoid())
out = l.predict(np.array((np.pi,1)).T)
print("\n--- Output of a layer of 5 neurons to the input [pi 1].T ---\n\n{}\
   n".format(out))
print("\n--- Weights of the whole layer ---\n\n{}\n".format(l.w[0]))
```

```
--- Output of a layer of 5 neurons to the input [pi 1].T ---

[0.47983167 0.02829935 0.04587734 0.09277917 0.36100712]


--- Weights of the whole layer ---

[[-0.16595599 -0.99977125 -0.70648822 -0.62747958 -0.20646505]
 [ 0.44064899 -0.39533485 -0.81532281 -0.30887855  0.07763347]]
```

Figure 3: Outcomes and respective weights of a layer of 5 neurons with input $[\pi, 1]$

3. Next, we define the MLP, in this exercise, we use a single hidden layer. As previously established, the hidden layer uses a sigmoid activation function, and the output layer a linear

activation function. The following code defines the hidden and output layers inside the *init* function. In this case, the output layer has 5 inputs.

```
def __init__(self, n_inputs, n_hidden_units, n_outputs, alpha=1e-3):
    self.n_inputs = n_inputs
    self.n_hidden_units = n_hidden_units
    self.n_outputs = n_outputs

    self.alpha = alpha

    self.l1 = Layer(n_inputs+1, n_hidden_units, Sigmoid())
    self.l2 = Layer(n_hidden_units+1, n_outputs, LinearActivation())
```

4. The *predict* functions is implemented using the previously defined functions for each layer. The network is initialized with 2 inputs and 1 output.

```
def predict(self, x):
    """
    Forward pass prediction given the input x
    """
    pred1 = self.l1.predict(x)
    pred1 = np.hstack((np.ones((1,1)),np.asarray(pred1).reshape(1,-1))).T
    pred2 = self.l2.predict(pred1)
    return pred2
```

To initialize a network with 2 inputs, 1 output and 3 units in the hidden layer:

```
mlp = MLP(2,3,1)
print("\nWeights from input to hidden layer:\n\n{}".format(mlp.
    export_weights()[0]))
print("\nWeights from hidden layer to output:\n\n{}\n".format(mlp.
    export_weights()[1]))
```

To check the correctness, in 4 the weights of the two layer are shown. From the input layer to the output layer, the first row represent the bias weights, one for each hidden units. The second and third row, instead, represent the weight for the first and second input respectively. Again one for each hidden units. Instead, from the hidden layer to the output one, there are 4 weights: one for the bias and three for the output of each unit in the hidden layer.

```
Weights from input to hidden layer:

[[[-0.16595599 -0.39533485 -0.62747958]
  [ 0.44064899 -0.70648822 -0.30887855]
  [-0.99977125 -0.81532281 -0.20646505]]]

Weights from hidden layer to output:

[[[ 0.07763347]
  [-0.16161097]
  [ 0.370439  ]
  [-0.5910955 ]]]
```

Figure 4: Respective weights of a network with 2 inputs, 1 output and 3 units in the hidden layer

5. The *calc prediction error* function is implemented to model to calculate the network predictions and return the MSE.

```
def calc_prediction_error(model, x, t):
    """ Calculate the MSE prediction error """
    x = np.hstack((np.ones((x.shape[0],1)),x))
    y_hat = np.asarray([model.predict(x[i]) for i in range(len(x))]).reshape
        (-1,1)[:,0]
    err = y_hat - t
    return np.sum(err**2) / (2*len(t))
```

To test the MES on the untrained network, the code below its used. The respective MSE is: 0.2760.

```
    data = np.array( [ [0.5, 0.5, 0], [1.0, 0, 0], [2.0, 3.0, 0], [0, 1.0,
        1], [0, 2.0, 1], [1.0, 2.2, 1] ] )
    x = data[:,:2]
    t = data[:,2]
    mlp = MLP(2,3,1)
    mse = calc_prediction_error(mlp,x,t)
    print("\n--- MSE for the first iteration: {} ---\n".format(mse))
```

6. Next, the function *MLP.train* is implemented. As a first step, the forward pass process is applied to each layer of the network in order to calculate activation. Hidden layer extracts the input data by computing a weighted sum of the inputs and running the result through the activation function. The output is transformed to the next hidden layer until reaching to the output layer. Next, the output layer error which corresponds to the difference between predicted form network and actual from training data is computed. In order to is calculate gradient, output error iteratively propagates backwards through the network from the output layer to the input layer. The weights are updated by using gradient descent for the weight optimization procedure that helps to have closer actual output to the target output.

```
    def train(self, x, t):
        """ Train the network """
        hidden_layer_act = []
        hidden_layer_out = []
        output_layer_act = []
        output_layer_out = []

        # Add column of 1 for bias
        x = np.hstack((np.ones((x.shape[0],1)),x))

        # Loop over training examples
        for j in range(len(x)):
            # Forward pass
            x_i = x[j]
            t_i = t[j]
            # Activation for each perceptron
            act_h = self.l1.activation(x_i)
            hidden_layer_act.append(act_h)
            # Output from each perceptron
            out_h = self.l1.output(act_h)
            hidden_layer_out.append(out_h)
            # Activation output-layer for each datapoint
            act_o = self.l2.activation(np.hstack((np.ones((1)),out_h)))
            output_layer_act.append(act_o)
            # Output from ouput-layer
            out_o = self.l2.output(act_o)
            output_layer_out.append(out_o)
        end for

        # # Convert to np array
        hidden_layer_act = np.hstack((np.ones((len(x), 1)),np.asarray(
            hidden_layer_act).reshape(len(x),self.n_hidden_units)))
        hidden_layer_out = np.hstack((np.ones((len(x), 1)),np.asarray(
            hidden_layer_out).reshape(len(x),self.n_hidden_units)))
        output_layer_act = np.asarray(output_layer_act)
        output_layer_out = np.asarray(output_layer_out).reshape(len(x),1)

        #Backpropagation:
        # Output error on predictions
        out_error = np.asarray(output_layer_out.T - t).T
        # Hidden layer error
        hidden_delta = (hidden_layer_out[:,1:] * (1 - hidden_layer_out
            [:,1:])) * np.dot(out_error, self.l2.w[0].T[:,1:])
```

```
        # Partial derivatives
        hidden_pd = x[:, :, np.newaxis] * hidden_delta[: , np.newaxis, :]
        output_pd = hidden_layer_out[:, :, np.newaxis] * out_error[:, np.
            newaxis, :]
        # Average for total gradients
        total_hidden_gradient = np.average(hidden_pd, axis=0)
        total_output_gradient = np.average(output_pd, axis=0)

        # Update weights
        self.l1.update_weights(-self.alpha*total_hidden_gradient)
        self.l2.update_weights(-self.alpha*total_output_gradient)
```

7. Finally, the network is trained to act like a XOR gate, by passing four training examples to train the function. It is tested with different learning rates, however it has been left at 0.1. The implementation of this training can be seen in the following block of code.

```
    """ Train as XOR gate """
    data = np.array( [ [0, 0, 0], [0, 1, 1], [1, 0, 1], [1, 1, 0] ] )
    x = data[:,:2]
    t = data[:,2]
    alpha = 0.1
    mlp = MLP(2,2,1,alpha)

    err = calc_prediction_error(mlp,x,t)
    it = 0
    while err > 0.001:
        mlp.train(x,t)
        err = calc_prediction_error(mlp,x,t)
        it +=1
    print("\n--- Converged after {} iterations with a learning rate of {}
        ---".format(it,alpha))
    print("\n--- Output after convergence: ---")
    x_bias = np.hstack((np.ones((x.shape[0],1)),x))
    for i in range(len(x)):
        print("\t",mlp.predict(x_bias[i])[0][0])
```

The results, 5, shows that using a learning rate of 0.1 and considering the network as converged when the MSE is lower than 0.001, 6733 iterations are needed. In addition, also the outputs, after the network has been trained, are showed.

```
--- Converged after 6733 iterations with a learning rate of 0.1 ---

--- Output after convergence: ---
        [0.01667909]
        [0.96168614]
        [0.96210172]
        [0.06929575]
```

Figure 5: Results coming from XOR network trained with a learning rate of 0.1

The two plot below represent how the MSE changes according to the number of epochs. They represent the same XOR network with identical hyper-parameters, the only difference is that on the right a logarithmic scale is used for the MSE.

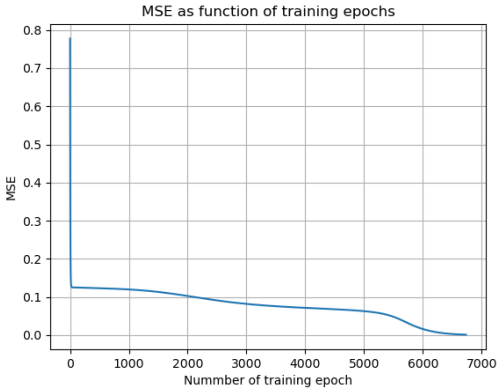

Figure 6: MSE on linear scale.

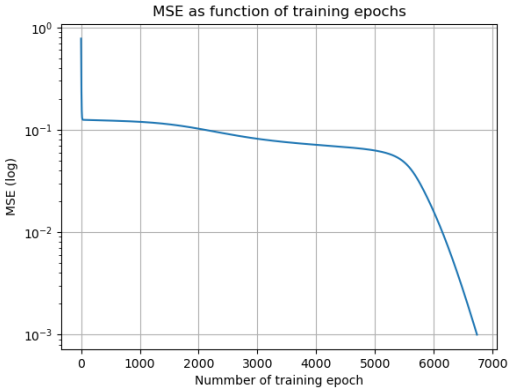

Figure 7: MSE on logarithmic scale.

3 Exercise 1.3 Fable exercise

In this exercise we tackle target reaching using the Fable robot and the previously defined neuron models. A webcam is required for this exercise. The objective is to move the robot's end-effector (green Lego piece) to a pixel location (x,y), using only joint commands.

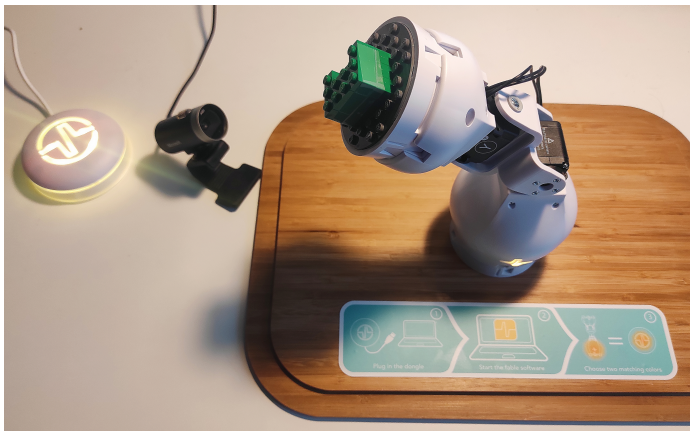

Figure 8: Material uses: Fable robot and webcam.

The robot is controlled through the FableAPI python library, therefore, we connect to the robot using the following given command:

```
from FableAPI.fable_init import api
api.setup(blocking=True)
moduleids = api.discoverModules()
print("Module IDs: ", moduleids)
```

The next important step for this exercise is to calibrate the camera colors. This must be done in order for the camera to detect and locate the green lego piece, that is, the Fable's end-effector.

The camera gives pixel coordinate systems, however, the command to make the robot move takes as inputs the X and Y motor angles. Therefore, we use a NN that takes as inputs the pixel coordinates, and as output the motor angles.

To train it, we make the robot move to specific points and gather a dataset, and it is trained accordingly until the error that we get is acceptable.

The program used to implement this taks is seen in the following block of code:

```
import camera_tools as ct
import cv2
import numpy as np

def callback(value):
    pass

# To test the camera and object detection algorithm:
def detectObj():
    cam = ct.prepare_camera()
    while True:
        img = ct.capture_image(cam)
        x, y = ct.locate(img)
        if x is not None:
```

```
            break
    print("Now the camera is done adjusting!")

    for _ in range(10):
        img = ct.capture_image(cam)
        x, y = ct.locate(img)
        print(x, y)

    cam.release()
```

The *getPos()* is used to get the training data. Since the end-effector, when detected, continuously slightly moves, the function capture 10 frames and return the average target position.

```
# Captures 10 images and locate the largest green object:
def getPos():
    cam = ct.prepare_camera()
    while True:
        img = ct.capture_image(cam)
        x, y = ct.locate(img)
        if x is not None:
            break
    # print("Now the camera is done adjusting!")
    X,Y = [],[]
    for _ in range(10):
        img = ct.capture_image(cam)
        x, y = ct.locate(img)
        # print(x, y)
        X.append(x)
        Y.append(y)
    cam.release()
    X = np.rint(np.mean(np.asarray(X))).astype(int)
    Y = np.rint(np.mean(np.asarray(Y))).astype(int)
    return (X,Y)

def showCamera():
    cam = ct.prepare_camera()
    ct.show_camera(cam)

def move(x,y, module = moduleID, getPixel = True):
    api.setPos(x, y, module)
    api.sleep(2)
    if getPixel:
        return getPos()

```

```
# select a pixel in the camera frame, and move the arm accordingly.
def selectPoint():
    cam = ct.prepare_camera()

    class TestClass:
        def __init__(self):
            self.i = 0

        @ct.camera_loop(cam,low_green, high_green, wait_time=5.0) # only
           runs the function every 0.5 second.
        def go(self, x, y, clickpoint):
            if clickpoint is not None:
                print(clickpoint)
                xy = torch.tensor(clickpoint).float()
                xy = model(xy)
                print(xy[0],xy[1])
                print("\n")
                move(xy[0],xy[1],getPixel = False)
            if self.i > 1000:
                return True
            self.i += 1
            return False

    test = TestClass()
    test.go()

import torch
# # Define neural network
class Net(torch.nn.Module):
    def __init__(self, n_feature, n_hidden, n_output):
        super(Net, self).__init__()
        self.hidden = torch.nn.Linear(n_feature, n_hidden)
        self.predict = torch.nn.Linear(n_hidden, n_output)

    def forward(self, x):
        x = torch.sigmoid(self.hidden(x))
        x = self.predict(x)
        return x
```

```
# calibrate the camera, set threshold camera
low_green, high_green = ct.colorpicker()

# Create training data moving the end-effector to predefined position and
   saving its respective motor angles
```

```
stepX = 30
stepY = 30
angleY = 0
odd = True
x,y = [],[]

def appendL(x,list):
    for el in list:
        x.append(el)
    return x

while angleY >= -90:
    if odd:
        x1 = range(-90,90+stepX,stepX)
        y1 = [angleY]*len(x1)
        odd = False
    else:
        x1 = range(90,-90-stepX,-stepX)
        y1 = [angleY]*len(x1)
        odd = True
    angleY -= stepY
    x = appendL(x,x1)
    y = appendL(y,y1)
```

```

X = torch.tensor(knownPos).float()
t = torch.tensor(list(zip(x,y))).float()

model = Net(n_feature=2, n_hidden=50, n_output=2)
loss_func = torch.nn.MSELoss()
# optimizer = torch.optim.SGD(model.parameters(), lr=0.1)
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)

# loop for train
num_epochs = 100000
for _ in range(num_epochs):
    prediction = model(X) # Forward pass prediction. Saves intermediary
        values required for backwards pass
    loss = loss_func(prediction, t) # Computes the loss for each example,
        using the loss function defined above
    optimizer.zero_grad() # Clears gradients from previous iteration
    loss.backward() # Backpropagation of errors through the network
    optimizer.step() # Updating weights

```

```
19      #print("Prediction: ", prediction.detach().numpy())
20      print("Loss: ", loss.detach().numpy())
21
22  # Once the model is trained, move the end-effector to the position selected
       in the camera frame
23  selectPoint()
```

After implementing this code, the robot is able to move accordingly when clicking at a point on the camera frame.

Questions:

- What happens if you change the relative positioning of the camera and robot after starting your code?

  The training of the data is done at a fixed camera position, therefore, although a different position of the camera would be able to detect the green lego piece, it would not predict accurately the motor angles of the robot. That is, it would not go to the desired coordinate location in the new camera frame. For this reason, changing the relative positioning of the camera would imply having to restart the code to get the right prediction from the neural network.

- In your solution, is learning active all the time?

  Our solution does not involve continuous active learning. The training data is set at the beginning of our code and is directly fed to the neural network to make predictions. There is not an additional option to label new data points.

- If not, could you imagine a way to change your solution to have "active" (online) learning? Would it work?

  A possible solution would be to fed the already trained neural network with new labeled data points. However, this would be a bit of a hazardous task, as it would require to know the motor angles for each pixel coordinates in the camera frame, which turns out to be almost infeasible.