DTU ELECTRO
34752 Course Bio-inspired control for robots
Programming exercises

Teacher Assoc. Prof. Silvia Tolu, *stolu@dtu.dk*, b.326, r.106, NRT-Lab
Teaching Assistants (TAs): Anders Eiersted Molzen, *s194024@student.dtu.dk*, b.326, NRT-Lab
Júlia Rey Vilches, *jrevi@dtu.dk*, b.326, r.110, NRT-Lab

August 2025, **Part I Perceptron and MLP**

Contents

1 Neuron models

Exercise 1.1 Perceptron

The first two exercises (1.1 and 1.2) may be finished either by implementing everything from scratch or by filling out a provided template. In case of the first option, you are required to structure your code in a similar way as in the templates.

To find a template for the first exercise, check folder 1.1. In this first exercise, we will program the Perceptron model in Python. We will use the perceptron template in file `perceptron.py`. The parts to be implemented are labeled with **TODO** followed by a text that describes what to implement.

The perceptron has a number of weights and an activation function. Firstly, we will define the activation function. We will define them as classes so that we can define the value and the gradient in one Python object. Here we will use the sign activation function such that the output will be 0 or 1, and we will use the perceptron to act as a linear separator.

Recall that the activation of the perceptron is given as a linear combination of the inputs:

$$a = \sum_{i=0}^{n} w_i x_i = \mathbf{w}^T \mathbf{x}, \tag{1}$$

with $x_0 = 1$, and the output is then

$$y = f(a) \tag{2}$$

1. Fill in the `SignActivation` class in the template. The `forward` function should return the output of the activation function, and the `gradient` function should return the gradient. For the sign activation, you do not have to define the gradient. Test your function to ensure that your implementation works.
   Hint: instantiate your SignActivation class and call the function.

   Solution (code and outcome):

2. Implement the initialization of a perceptron in `__init__`, by defining the weights and activation function. The weights should be randomly distributed, to do so you can use `np.random.normal()`. Select initial parameters for the distribution, these can be adjusted later. Remember the bias w_0! Test your implementation.
   Hint: define an instance of the `Perceptron` class, passing the number of inputs and the activation functions.

   Solution (code and outcome):

3. Implement the `activation`, `output` and `predict` functions. The predict function can be implemented by combining the `activation` and `output` functions. Test that the `predict` function works, by computing the prediction on each of the data points given in `x`. The predictions should be completely random at this point.
   Hint: On previously initialized perceptron call the newly implemented functions with inputs

   Solution (code and outcome):

Now we will implement the learning algorithm, written as:

$$w_i(k+1) = w_i(k) + \eta(t_j - y_j)x_{ji}, \tag{3}$$

given the training example $\mathbf{x}_j$ with target t_j, the perceptron prediction y_j and learning rate η.

4. Write the code to use the previous learning rule for the weight update. Try out different values for the learning rate and weight initialization. How many training epochs are required to reach convergence?
   Hint: Try very low learning rate to increase number of epochs 10e-3

   Solution (code, outcome and your considerations):

5. Print the final weight values. Do they reach the same final values for different initial values? Why/why not?

   Solution (outcome and your considerations):

6. Make a plot of the data points in two different colors, together with the line that separates them, as defined by the learned weights.
   Hint: The color is determined by the target of data set. Use equation for straight line to plot the separation line.

   Solution (plot and your considerations):

Exercise 1.2 Multi-layer perceptron

The MultiLayer Perceptron (MLP) is a type of NN consisting of at least an input layer, a hidden layer, and an output layer. The MLP layers are fully connected, meaning that a neuron receives input from all neurons in the previous layer. An example is shown in Figure 1.

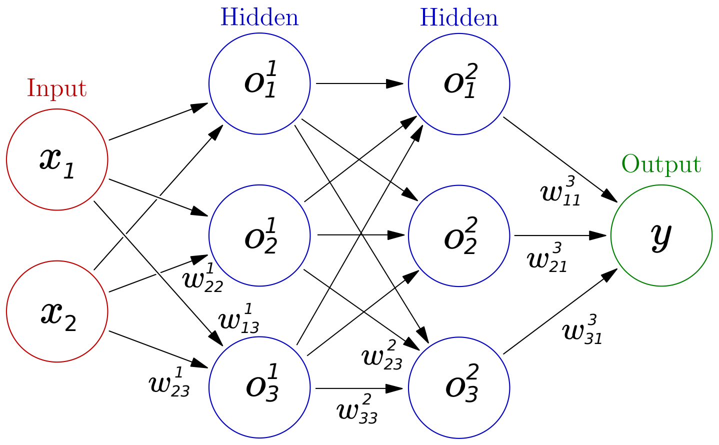

Figure 1: Illustration of an MLP with 4 layers: the input layer, two hidden layers and the output layer.

The mathematical definitions are explained in Table 1. The neurons in the MLP are the perceptrons that you have already implemented, but the learning algorithm is more complex. For completion, we write the equations for the prediction here. The activation of the neuron j in layer k can be written as

$$a_j^k = \left(\mathbf{w}_j^k\right)^T \mathbf{o}^{k-1}, \tag{4}$$

where $\mathbf{w}_j^k$ is a vector of the weights for the neuron j in layer k. The output of that neuron is then

$$o_j^k = f_k(a_j^k), \tag{5}$$

where f_k is the activation function of layer k.

Table 1: Definitions for the MLP.

Symbol	Description
$\boldsymbol{x}$	Input vector in input layer
$\boldsymbol{y}$	Output vector in output layer
a_j^k	Activation of neuron j in layer k
o_j^k	Output of neuron j in layer k
δ_j^k	Error for neuron j in layer k
w_{ij}^k	Weight for input i in neuron j in layer k

For this exercise (check folder 1.2), we will use the MLP to give a real-valued output. Initially, assume that the network has a 1-dimensional output y. Firstly, we will define the activation functions that are needed for the network in this case.

1. Open the template, `mlp.py`. Fill in the `Sigmoid` and `LinearActivation` classes. The sigmoid function will be used in hidden layers and the linear activation in the output layer. Test your functions with a range of inputs.
   Hint: Same as in 1.1

Solution (code and outcome):

2. Now we will create a `Layer` class which contains a number of perceptrons. Most of the functions for this have been created for you, as they are utility functions using your perceptron class. For example, the `activation` method calls the activation function in each of the perceptrons in the layer and returns all results in a vector (numpy array).
   Hint: Your perceptron from exercise 1 must work correctly
   Implement the initialization of each perceptron in the layer, in the `__init__` function. Test your `Layer` class by printing the output of a layer of 5 neurons to the input $\begin{bmatrix}\pi & 1\end{bmatrix}^T$. Also, print the weights of the whole layer - does it have the dimensions you would expect?
   Hint: Instantiate your layer class and print the weights of it

   Solution (code and outcome):

3. Now we will define the MLP. Here we will use a single hidden layer. Define the hidden layer and the output layer in `MLP.__init__`. The hidden layer should use a sigmoid, and the output layer a linear activation function. How many inputs does the output layer have?
   Hint: We will only use 2 layers, no need to create loops for extra layers

   Answer:

4. Implement the `predict function` using the functions defined for each layer. Initialize a network with 2 inputs and 1 output and select the number of units in the hidden layer. Check that the network layers have the expected number of perceptrons, each with the expected number of inputs.
   Solution (code and outcome):

To evaluate the network, we use the mean square error (MSE):

$$E = \frac{1}{N}\sum_{i=1}^{N}(y_i - t_i)^2 \tag{6}$$

where $(\boldsymbol{x}_i, t_i)$ is the input-output pair (training example) and y_i is the output of the network to the input $\boldsymbol{x}_i$.

5. Implement the function `calc_prediction_error`, which should take as input an instance of the `MLP` class and an array x containing an input vector in each row, and an array t with the corresponding targets in each row. The function should calculate the network predictions y_i and return the MSE. Test the function with your untrained network.
   Hint: Implement formula (6)

   Solution (code and outcome):

Backpropagation

To update the weights of all perceptrons, an error has to be associated with each. However, an error is only available for the output layer. To solve this issue, we will use what is known as backpropagation, i.e. backward propagation of errors.
The following equations can be derived using the chain rule of partial derivatives. The derivation can be found in the Appendix A.
The errors for the output layer are:

$$\delta_j^K = f_K'(a_j^K)(y - t), \tag{7}$$

where K denotes the last layer.
The backpropagation equation gives the errors δ_j^k of previous layers as a function of the errors in the layer $k+1$:

$$\delta_j^k = f_k'(a_j^k) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}, \tag{8}$$

where r^{k+1} is the number of neurons in layer $k+1$.
The weight change due to one training example is then:

$$\Delta w_{ij}^k = -\frac{\alpha}{N} \delta_j^k o_i^{k-1}, \tag{9}$$

where N is the number of training examples in the batch and α is the learning rate.
See Appendix A.

6. Now we will implement the function `MLP.train`. It should loop over all training examples for each:

   (a) Calculate the prediction (forward pass), saving all the intermediate steps (a_j^k and o_j^k). Do this without using your `MLP.predict` or the `Layer.predict` function, as these do not give you the activations.

   Hint: You need activation and output of each layer

   (b) From the prediction, calculate the error δ_j^2 for the output layer, using eq. (7) and the errors for the previous layer using eq. (8).
   Hint: δ is calculated only per layer

   (c) Calculate the weight change in each layer from eq. (9). Recall that o_0^{k-1} corresponds to the "bias input" $x_0 = 1$.

   The weight change contributions for all examples should be summed and applied at the end, not during the computation for each example.

   Solution:

7. Train your network to act as an XOR gate by passing four training examples to your `train` function. Two hidden units should be enough.

   (a) Try training with different learning rates. How many "epochs" do you have to train the network before it converges?

   (b) Try plotting the MSE as a function of the training epoch. What is the largest learning rate that still yields the lowest possible MSE? Use `plt.yscale('log')` to plot on a logarithmic y-axis. [Optional] If you have time left at this point, you can extend the code for multidimensional output.

Solution:

Exercise 1.3 Fable exercise

In this exercise, you will have to solve a problem involving the Fable robot by using neuron models. The problem is visual target reaching. For simplicity, the visual information comes from a single camera; thus, there will be no measure of depth in the image (see Fig. 2).
Web cam is required for this exercise as well as Fable. Recall to plug in the Fable dongle and select the same color in the robot and dongle before starting.

Problem: Given an (x,y) pixel location in the camera image, the end effector of the robot should move to that position in the image, using only joint commands.

Figure 2: Illustration of the exercise setup.

- *Get training data: Set the robot in known positions θ_x and θ_y, take the picture and get X Y, record a few positions.*
- *Train your network with training data.*
  Hint: You have 2 inputs X and Y, and 2 outputs θ_x and θ_y
- *After successful training, click on the picture to get X and Y, send it to your trained network to get output θ_x and θ_y for robot position*

Tools available:

- You can use any of your previous code from the course (neuron or neural network models).
- The robot is controlled through the FableAPI Python library, where you can give joint position, velocity, or torque commands. See `fable_example.ipynb`.
- The current pixel location of the end effector is available from the image with a camera library; see an example in `camera_example.ipynb`.

- If you need faster neural network training, you can use PyTorch. See `pytorch_example.ipynb` to get started.

  Hint: You can solve this exercise with PyTorch

Limitations:

- You cannot use a measuring tape.
- The complete solution should work if the relative positioning of the camera and the robot is changed after the code has been written.

Questions:

- What happens if you change the relative position of the camera and the robot after starting your code? Answer:
- In your solution, is learning active all the time? Answer:
- If not, could you imagine a way to change your solution to have "active" (online) learning? Would it work? Answer:

[Optional] After solving the problem with a bio-inspired solution, you are welcome to try and solve the problem with any other method you can think of. How do the methods compare?

Exercise 1.4 Spiking Neuron Model

The Leaky Integrate-and-Fire (LIF) model can be formulated as a differential equation.

$$\tau_m \frac{du_m}{dt} = u_{\text{rest}} - u_m(t) + R_m I(t) \tag{10}$$

where $\tau_m = R_m C_m$, and with the reset logic

$$\text{If } \ u_m(t) > u_{\text{thresh}} \Rightarrow u_m(t^+) = u_{\text{rest}}. \tag{11}$$

In this exercise, we will use $R_m = 10\,\text{M}\Omega$, $C_m = 1\,\text{nF}$, $u_{\text{thresh}} = -50\,\text{mV}$ and $u_{\text{rest}} = -65\,\text{mV}$. We will simulate the model using Euler integration.

1. Implement a leaky integrate-and-fire function in Python. It should take as input (1) an initial This can be done in the following way:
   (a) Initialize a membrane potential variable to $u_m(0)$ and a time variable to zero. Pick an internal simulation time step not larger than $\Delta t = 1 \times 10^{-5}\,\text{s}$.
   (b) Make a loop that terminates when the given end time has been reached. In every iteration:
      - Calculate the derivative $\frac{du}{dt}$ from Eq. (10).
      - Update the voltage by adding $\frac{du}{dt}\Delta t$ to it.
      - Evaluate the logic, Eq. (11).
      - Update the simulation time.

   (c) Return the final membrane potential. *Hint: Your LIF function should return the calculated membrane potential*

   Solution:

2. Use your function to simulate the LIF with $I = 1\,\text{nA}$ for 100 ms. Pick a suitable time step T_s for plotting the membrane potential. What is the minimum current required to produce a spike?

   Solution:

3. Write a function to calculate the inter-spike intervals (ISI) from the plotted values. Calculate the spiking frequency.

   Solution:

4. Run a series of experiments to plot the spiking frequency as a function of the constant input current for currents in the range 0-5 nA.

   Solution (code and outcome):

A Derivation of backpropagation

We want to find the gradient of the error with respect to each of the weights w_{ij}^k in each of the layers k. Using the chain rule, we can write the gradient as

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial a_j^k} \frac{\partial a_j^k}{\partial w_{ij}^k} \tag{12}$$

We identify the second partial derivative as the output of the previous layer $k-1$:

$$\frac{\partial E}{\partial w_{ij}^k} = \frac{\partial E}{\partial a_j^k} o_i^{k-1} \tag{13}$$

We define the "error" for each neuron as follows

$$\delta_j^k \equiv \frac{\partial E}{\partial a_j^k} \tag{14}$$

then

$$\frac{\partial E}{\partial w_{ij}^k} = \delta_j^k o_i^{k-1}. \tag{15}$$

The goal is then to determine the errors δ_j^k:

$$\delta_j^k = \sum_{l=1}^{r^{k+1}} \frac{\partial E}{\partial a_l^{k+1}} \frac{\partial a_l^{k+1}}{\partial a_j^k} = \sum_{l=1}^{r^{k+1}} \delta_l^{k+1} \frac{\partial a_l^{k+1}}{\partial a_j^k} \tag{16}$$

The activation of the neuron l in layer $k+1$ is

$$a_l^{k+1} = \sum_{j=0}^{r^k} w_{jl}^{k+1} o_j^k = w_{0l}^{k+1} + \sum_{j=1}^{r^k} w_{jl}^{k+1} f_k(a_j^k) \tag{17}$$

So, the derivative is

$$\frac{\partial a_l^{k+1}}{\partial a_j^k} = w_{jl}^{k+1} f_k'(a_j^k) \tag{18}$$

Putting this together, we get:

$$\delta_j^k = f_k'(a_j^k) \sum_{l=1}^{r^{k+1}} w_{jl}^{k+1} \delta_l^{k+1}. \tag{19}$$