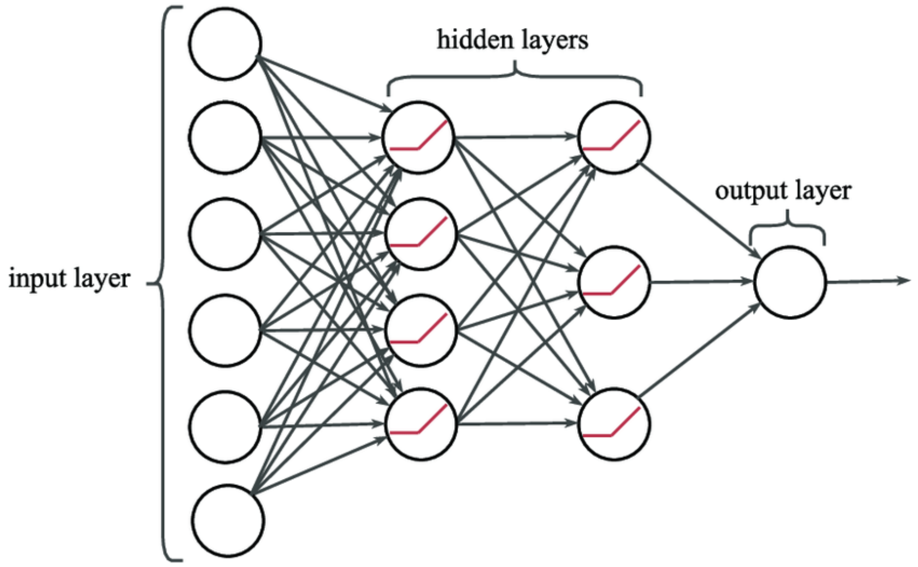
CI Course - EX8

Theory Overview

What is **Artificial Neural-Networks (ANN)** ?

Neural networks are a type of machine learning algorithm inspired by the structure and function of the human brain. They consist of interconnected nodes, or **"neurons"**, that process and transmit information. These networks can be trained on large amounts of data to learn patterns and relationships, and can be used for a variety of tasks such as image recognition, natural language processing, and predictive modeling.



In a vector form:

$$\left.\begin{aligned} y_1 &= f(\textstyle\sum_i^m w_{1i}x_1 + b_1) \\ y_n &= f(\textstyle\sum_i^m w_{ni}x_n + b_n) \end{aligned}\right\} \bar{y} = f(\sum_i^m \bar{w}_i\bar{x} + \bar{b})$$

Where:

w = weights

$$w = \begin{bmatrix} w_{11} & \dots & w_{1n} \\ \vdots & & \vdots \\ w_{n1} & \dots & w_{nn} \end{bmatrix}$$

b = bias

$$\bar{b} = \begin{bmatrix} b_1, \\ b_2, \\ \vdots \\ b_n \end{bmatrix}$$

Weights and bias are a learnable variables!

f = activation function as mention in the last lecture:

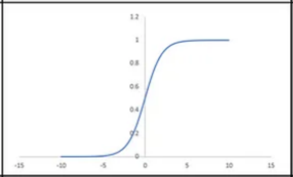
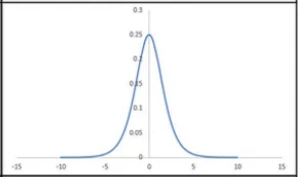
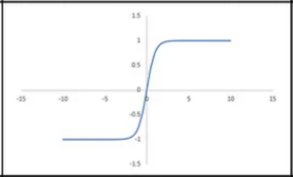
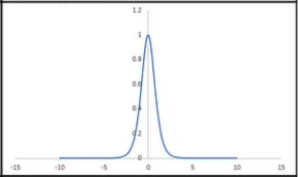
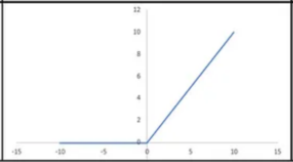
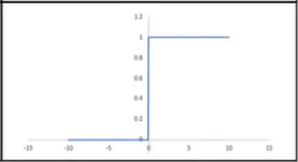
Activation function	f(x)	f(x)	$\frac{df(x)}{dx}$
sigmoid	$\frac{1}{1+e^{-x}}$		
tanh	$tanh(x)$		
RelU	max(0,x)		

Table 1: Activation functions (image by author)

Also x = inputs (or features) vector.

$$x = (x_1, x_2, ..., x_n)^T$$

And y = the output vector.

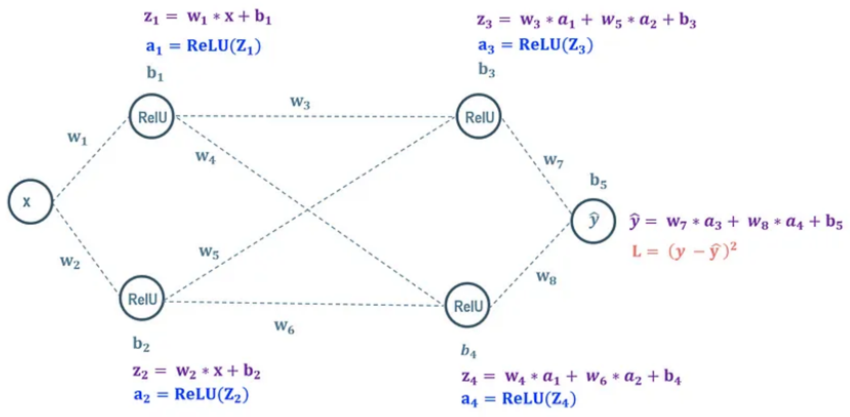
$$y = (y_1, y_2, ..., y_n)^T$$

The general process of ANN during training is **comparing the network's output $\bar{y}$ with the desired output y. The goal** is to **minimize the difference** between them, and updating the **weights and biases** respectively, allowing us to improve the network's performance.

The algorithm consists of two main steps: forward propagation and backward propagation.

Forward Propagation:

In an artificial neural network forward pass is the calculation process where values of the output layers are obtained from the input data. It involves traversing through all neurons from the first to the last layer.



Loss function:

A loss function measures how good a neural network model is in performing a certain task, which in most cases is regression or classification. The value of the loss function must be minimized during the backpropagation step in order to make the neural network better.

Mean Absolute Error (MAE): Also called L1 Loss, this loss function is used for regression problems. It measures the average magnitude of errors in a set of predictions, without considering their direction1.

$MAE = \frac{1}{n} \sum_{i=1}^{n} |y_i - \hat{y}_i|$

Mean Squared Error (MSE): Also called L2 Loss, this loss function is also used for regression problems. It measures the average squared difference between the predicted and actual values1.

$MSE = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$

Cross-Entropy Loss: Also known as Log Loss, this loss function is used for classification problems where the output can be either 0 or 1. It measures the performance of a classification model whose output is a probability value between 0 and 11.

$CrossEntropy = -\frac{1}{n}\sum_{i=1}^{n}[y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)]$

where n is the number of samples, y_i is the true value and $\hat{y}_i$ is the predicted value.

Back-Propagation

The main concept of back propagating is computing the gradients of the loss function with respect to the weights and biases, **starting from the output** layer and **moving towards the input** layer.

General equations of B.P:

Let's say we have a neural network with L layers, where $l \in \{1, ..., L\}$ denotes the layer index.The weights and biases of the network are represented by w_{jk}^{l} and b_j^l, respectively.The activation of the j^{th} neuron in the l^{th} layer is represented by a_j^l.The error in the output layer L is given by $\delta_j^L = \frac{\partial C}{\partial a_j^L}\sigma'(z_j^L)$, where C is the loss (=cost) function and $\sigma'(z)$ is the derivative of the activation function with respect to the weighted input z.The error in any other layer can be calculated recursively using the formula $\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1}\sigma'(z_j^l)$. The partial derivatives of the cost function with respect to the weights and biases can then be calculated as $\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l$ and $\frac{\partial C}{\partial b_j^l} = \delta_j^l$, respectively.

For a specific case:

For example the derivative of the Mean Squared Error (MSE) loss function with respect to the predicted value $\bar{y}$ is given by:

$$\frac{\partial}{\partial \hat{y}}MSE = \frac{\partial}{\partial \hat{y}}\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = -\frac{2}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)$$

where n is the number of samples and y_i is the true value.

In a neural network, the weights are updated during the backpropagation step using gradient descent. The gradient descent algorithm seeks to change the weights so that the next evaluation reduces the error. The weights are updated using the following formula:

$$w_{inew} = w_{iold} - \alpha\frac{\partial L}{\partial w_i}$$

where w_{new} and w_{old} are the new and old values of the weight, respectively, α is the learning rate, and $\frac{\partial L}{\partial w_i}$ is the partial derivative of the loss function with respect to the weight.

In the same way we will calculate the bias update

$$\bar{b}_{inew} = \bar{b}_{iold} - \alpha \frac{\partial L}{\partial \bar{b}_i}$$

where

$$\frac{\partial L}{\partial b_i} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial b_i} = \frac{\partial L}{\partial \hat{y}}$$

The derivative of the loss with respect to the weight depends on the specific architecture of the neural network and the chosen loss function. In general, the derivative of the loss with respect to the weight can be calculated using the chain rule. For example, let's consider a simple neural network with one input layer, one hidden layer, and one output layer. Let x be the input, w_1 be the weight connecting the input layer to the hidden layer, w_2 be the weight connecting the hidden layer to the output layer, and y be the output. Let L be the chosen loss function. Then, the derivative of the loss with respect to w_i

$$\frac{\partial L}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial w_i} = \frac{\partial L}{\partial \hat{y}} * X_i^T$$

The specific values of these partial derivatives depend on the chosen activation functions and loss function.

$$\frac{\partial L}{\partial X_i}$$

will be the:

$$\frac{\partial L}{\partial \hat{y}}$$

of the previus layer

$$\frac{\partial L}{\partial X_i} = \frac{\partial L}{\partial \hat{y}} * \frac{\partial \hat{y}}{\partial X_i} = \frac{\partial L}{\partial \hat{y}} * W_i^T$$

Exercise - Predicting fuel consumption

The purpose of the exercise is to train a neural network to predict fuel consumption (in km/l) based on various features of cars. The nueral network will be built from scratch!

Input and Output:

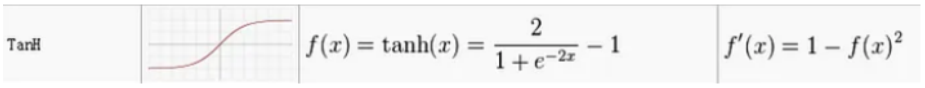
The **input** of the code is a dataset containing **399 data samples** of **4 features of cars**, such as displacement, number of cylinders, horsepower, and weight.

The **output** is the **fuel consumption** (km/l) corresponding to each sample (=**regression problem**).

To enhance the definition of the network architecture, we will employ object-oriented programming (OOP) techniques by implementing multiple classes.

Classes:

1. `FCLayer`: This class represents a fully connected layer in the neural network.
2. `ActivationLayer`: This class implements the activation function for the neural network. It uses the hyperbolic tangent (tanh) activation function and its derivative.



TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$

The derivative of the activation tanh is:

$$\frac{d}{dx}\tanh(x) = \frac{d}{dx}\left(\frac{e^x - e^{-x}}{e^x + e^{-x}}\right) = 1 - \tanh^2(x)$$

1. `Network`: This class represents the neural network itself.

Solution Steps:

1. Define and initialize the neural network classes
2. Load and preprocess the data
3. Creating the network
4. Training!
5. Evaluate

---

0. Import libraries

```
import numpy as np
import pickle
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
%matplotlib qt
from sklearn.model_selection import train_test_split
```

1. Define and initialize the neural network classes

FCLayer:

- Purpose: This class represents a fully connected layer in the neural network, responsible for forward and backward propagation.
- Actions: Initialization of the layer with random weights and biases, forward propagation calculation, and backward propagation updates.

```
class FCLayer():
    # input_size = number of input neurons
    # output_size = number of output neurons
    def __init__(self, input_size, output_size):
        self.input = None
        self.output = None
        self.input_size = input_size
        self.output_size = output_size
        self.weights = np.random.rand(output_size, input_size) - 0.5
        self.bias = np.random.rand(1, output_size,) - 0.5

    # returns output for a given input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = np.dot(self.weights, self.input) + self.bias
        return self.output.reshape(-1,)

    # computes dE/dW, dE/dB for a given output_error=dE/dY. Returns input_error=dE/dX.
    #### NOTE: for the code we will use dE as dL from the explanetion
    def backward_propagation(self, output_error, learning_rate):
        input_error = np.dot(self.weights.T, output_error) # dE/dx
        weights_error = np.dot(output_error.reshape(-1,1), self.input.reshape(1,-1)) #dE/dw

        # update parameters
        self.weights -= learning_rate * weights_error
        self.bias -= learning_rate * output_error # as we saw in the explanation output ero
        return input_error
```

ActivationLayer:

- Purpose: This class implements the activation function for the neural network.
- Actions: Initialization of the layer, including the tanh activation function and its derivative, forward propagation calculation, and backward propagation calculation.

```
class ActivationLayer():
    def __init__(self):
        self.input = None
        self.output = None
        self.activation = lambda x: np.tanh(x)
        self.activation_prime = lambda x: 1-np.tanh(x)**2

    # returns the activated input
    def forward_propagation(self, input_data):
        self.input = input_data
        self.output = self.activation(self.input)
        return self.output

    # Returns input_error=dE/dX for a given output_error=dE/dY.
    # No "learnable" parameters ->  no learning rate
    def backward_propagation(self, output_error, learning_rate):
        return self.activation_prime(self.input) * output_error
```

Network:

- Purpose: This class represents the neural network and provides methods for adding layers, defining the loss function, predicting output, and training the network.
- Actions: Initialization of the network, addition of layers, definition of the **mean squared error loss** function, prediction of output, and training the network using backpropagation.

```
class Network:
    def __init__(self):
        self.layers = []
        self.loss = None
        self.loss_prime = None

    # add layer to network
    def add(self, layer):
        self.layers.append(layer)

    # Loss function
    def mse(self, y_true, y_pred):
        return np.mean(np.power(y_true-y_pred, 2)) # np.power is faster

    # Loss function and its derivative
    def mse_prime(self, y_true, y_pred):
        return 2*(y_pred-y_true)/y_true.size

    # predict output for given input
    def predict(self, input_data):
        # sample dimension first
        samples = len(input_data)
        result = []

        # run network over all samples
        for i in range(samples):
            # forward propagation
            output = input_data[i]
            for layer in self.layers:
                output = layer.forward_propagation(output)
            result.append(output)
        return result

    # train the network
    def fit(self, x_train, y_train, epochs, learning_rate):
        # sample dimension first
        samples = len(x_train)

        # training loop
        E = []
        for i in range(epochs):
            err = 0
            for j in range(samples):
                # forward propagation
                output = x_train[j]
                for layer in self.layers:
                    output = layer.forward_propagation(output)

                # compute loss (for display purpose only)
                err += self.mse(y_train[j], output)

                # backward propagation
                error_grad = self.mse_prime(y_train[j], output)
                for layer in reversed(self.layers):
                    error_grad = layer.backward_propagation(error_grad, learning_rate)

            # calculate average error on all samples
            err /= samples
            print('epoch %d/%d   error=%f' % (i+1, epochs, err))
            E.append(err)

            plt.figure(0)
            plt.cla()
            plt.plot(range(len(E)), E)
            plt.xlabel('Epochs')
            plt.ylabel('Error')
            plt.pause(0.000001)
```

2. Load and preprocess the data

```
# training data
with open('auto_kml.pkl', 'rb') as H:
    data = pickle.load(H)

# Input: Features of various cars
# 0. displacement
# 1. Number of cylinders
# 2. horsepower
# 3. weight
# Output: Fuel consumption (km/l)

X = data['features']
Y = data['kml'].reshape(-1,1)

print(f"example of X[0]: {X[0]} | example of Y[0]: {Y[0]}")
print(f"X shape is: {X.shape} | Y shape is: {Y.shape}")
```

```
example of X[0]: [   8.   307.   130. 3504.] | example of Y[0]: [3.401152]
X shape is: (399, 4) | Y shape is: (399, 1)
```

```
D = np.concatenate((X,Y), axis=1)

## normalizing for better learning process of the net
scaler = StandardScaler()
scaler.fit(D)
D = scaler.transform(D)

## splitting the data for test and train

X_train, X_test, y_train, y_test = train_test_split(D[:,:-1], D[:,-1], test_size=0.15, rando
```

3. Creating the network Our network architecture will consist of **4 connected layers**, with the **tanh activation function applied after each layer**.

```
# Create network
net = Network()
#layer1 + act1
net.add(FCLayer(X_train.shape[1], 3))
net.add(ActivationLayer())
#layer2 + act2
net.add(FCLayer(3, 3))
net.add(ActivationLayer())
# #layer3 + act3
net.add(FCLayer(3, 3))
net.add(ActivationLayer())
# #layer4 + act4
net.add(FCLayer(3, 1))
net.add(ActivationLayer())
```

```
## Hyperparameters
epochs = 80
learning_rate = 0.2
```

4. Training

```
net.fit(X_train, y_train, epochs=epochs, learning_rate=learning_rate)
print()
predictions = net.predict(X_train)
err = 0
for y_p, y in zip(predictions, y_train):
    err += net.mse(y, y_p)
print('Train loss: ', err/len(y_train))
```

```
epoch 1/80   error=0.239973
epoch 2/80   error=0.086333
epoch 3/80   error=0.076210
epoch 4/80   error=0.073244
epoch 5/80   error=0.071816
epoch 6/80   error=0.070984
epoch 7/80   error=0.070445
epoch 8/80   error=0.070068
epoch 9/80   error=0.069790
epoch 10/80   error=0.069574
epoch 11/80   error=0.069402
epoch 12/80   error=0.069260
epoch 13/80   error=0.069140
epoch 14/80   error=0.069037
epoch 15/80   error=0.068948
epoch 16/80   error=0.068869
epoch 17/80   error=0.068798
epoch 18/80   error=0.068734
epoch 19/80   error=0.068676
epoch 20/80   error=0.068623
epoch 21/80   error=0.068574
epoch 22/80   error=0.068529
epoch 23/80   error=0.068487
epoch 24/80   error=0.068448
epoch 25/80   error=0.068412
epoch 26/80   error=0.068377
epoch 27/80   error=0.068345
epoch 28/80   error=0.068314
epoch 29/80   error=0.068284
epoch 30/80   error=0.068256
epoch 31/80   error=0.068230
epoch 32/80   error=0.068204
epoch 33/80   error=0.068179
epoch 34/80   error=0.068155
epoch 35/80   error=0.068132
epoch 36/80   error=0.068110
epoch 37/80   error=0.068088
epoch 38/80   error=0.068067
epoch 39/80   error=0.068046
epoch 40/80   error=0.068026
epoch 41/80   error=0.068007
epoch 42/80   error=0.067988
epoch 43/80   error=0.067969
epoch 44/80   error=0.067951
epoch 45/80   error=0.067933
epoch 46/80   error=0.067916
epoch 47/80   error=0.067899
epoch 48/80   error=0.067882
epoch 49/80   error=0.067865
epoch 50/80   error=0.067849
epoch 51/80   error=0.067834
epoch 52/80   error=0.067818
epoch 53/80   error=0.067803
epoch 54/80   error=0.067788
epoch 55/80   error=0.067774
epoch 56/80   error=0.067760
epoch 57/80   error=0.067746
epoch 58/80   error=0.067732
epoch 59/80   error=0.067719
epoch 60/80   error=0.067705
epoch 61/80   error=0.067693
epoch 62/80   error=0.067680
epoch 63/80   error=0.067668
epoch 64/80   error=0.067656
epoch 65/80   error=0.067644
epoch 66/80   error=0.067633
epoch 67/80   error=0.067621
epoch 68/80   error=0.067610
epoch 69/80   error=0.067600
epoch 70/80   error=0.067589
epoch 71/80   error=0.067579
epoch 72/80   error=0.067569
epoch 73/80   error=0.067559
epoch 74/80   error=0.067550
epoch 75/80   error=0.067540
epoch 76/80   error=0.067531
epoch 77/80   error=0.067522
epoch 78/80   error=0.067513
epoch 79/80   error=0.067505
epoch 80/80   error=0.067496

Train loss:  0.06747583678422203
```

5. Evaluate

```
predictions = net.predict(X_test)
print(f"Given input X_test[0]: {X_test[0]}")
print(f"Prediction is: {predictions[0][0]} | Real is: {y_test[0]}")

print("______")

err = 0
for y_p, y in zip(predictions, y_test):
    err += net.mse(y, y_p)
print('Total Test loss: ', err/len(y_test))
```

```
Given input X_test[0]: [-0.85446201 -0.99101366 -0.88884502 -1.21835308]
Prediction is: -0.8547638371622026 | Real is: -0.8544620067201791
______
Total Test loss:  0.06805227332219639
```

Summary

In this class we covered:

1. ANN structure and main concepts.
2. Learning process of ANN.
3. Build an ANN from scratch and use it to solve regression problem of car fuel consumption predicting based on its features.

Helpful and extra links

1. Neural Network In 5 Minutes
2. Neural Networks: Inside the Black Box
3. Neural Networks: Forward pass and Backpropagation
4. Back-Propagation algorithm: A step by step demonstration
