UE C0 - Apprentissage automatique

# TP 2 - Multilayer Perceptron

Thomas Boulanger Jules Farnault

19 novembre 2024

# Table des matières

# Introduction

Le but de ce TP est de résoudre des problèmes de régression. Pour cela, on commencera par mettre en place des régressions polynomiale puis on utilisera un réseau de neurones de types MLP. Afin de faciliter la lecture des codes, ils sont mis à disposition sur Github.

# 1 | Modèles polynomiaux

## 1.1 - Implémentation

On complète le squelette du code proposé :

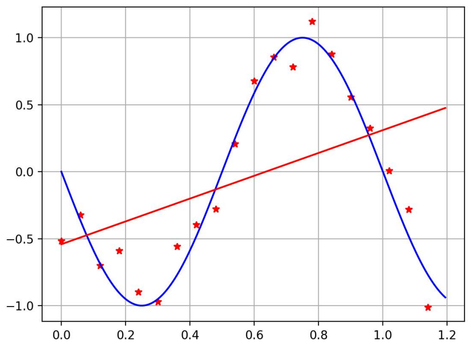
```
from sklearn.preprocessing import PolynomialFeatures
from sklearn.linear_model import LinearRegression
import matplotlib.pyplot as plt
import numpy as np


def genere_exemple_dim1(xmin, xmax, NbEx, sigma):
    x = np.arange(xmin, xmax, (xmax - xmin) / NbEx)
    y = np.sin(-np.pi + 2 * x * np.pi) + np.random.normal(
        loc=0, scale=sigma, size=x.size
    )
    return x.reshape(-1, 1), y


def getMSE(x, y, reg):
    return sum(pow(reg.predict(x) - y, 2)) / x.shape[0]


def plot_model(Xa, Ya, Xt, Yt, reg, nameFig):
    Ypred = reg.predict(Xt)
    plt.plot(Xa[:, 1], Ya, "*r")
    plt.plot(Xt[:, 1], Yt, "-b")
    plt.plot(Xt[:, 1], Ypred, "-r")
    plt.grid()
    plt.savefig(nameFig + ".jpg", dpi=200)
    plt.close()


def plot_error_profile(L_error_app, L_error_test, nameFig):
    plt.plot(range(1, len(L_error_app) + 1), L_error_app, "-r")
    plt.plot(range(1, len(L_error_test) + 1), L_error_test, "-b")
    plt.grid()
```

```
    plt.savefig(nameFig + ".jpg", dpi=200)
    plt.close()


def plot_confusion(Xt, Yt, reg, nameFig):
    plt.plot(Yt, reg.predict(Xt), ".b")
    plt.plot(Yt, Yt, "-r")
    plt.savefig(nameFig + ".jpg", dpi=200)
    plt.close()


def main(degreMax=12, NbEx=20, sigma=0.2):
    xmin = 0
    xmax = 1.2

    xapp, yapp = genere_exemple_dim1(xmin, xmax, NbEx, sigma)
    xtest, ytest = genere_exemple_dim1(xmin, xmax, 200, 0)

    L_error_app = []
    L_error_test = []

    for i in range(1, degreMax + 1):
        print("Degre = ", i)

        # Transformation de données d'entrée des bases d'app et de test
        poly = PolynomialFeatures(degree=i)
        Xa = poly.fit_transform(xapp)
        Xt = poly.transform(xtest)

        # Création du modèle linéaire
        reg = LinearRegression().fit(Xa, yapp)

        # Estimation des erreurs d'apprentissage et de test
        L_error_app.append(getMSE(Xa, yapp, reg))
        L_error_test.append(getMSE(Xt, ytest, reg))

        # plot du model de degré i
        plot_model(Xa, yapp, Xt, ytest, reg, "Model_%02d" % i)

    # Déterminer le degré opimal
    best = np.argmin(L_error_test) + 1
    print("Meilleur model -> degre =", best)
    plot_error_profile(L_error_app, L_error_test, "Profil_Err_App_Test")

    # Creation du modèle final optimal
    poly = PolynomialFeatures(degree=best)
    Xa = poly.fit_transform(xapp)
    Xt = poly.transform(xtest)
```

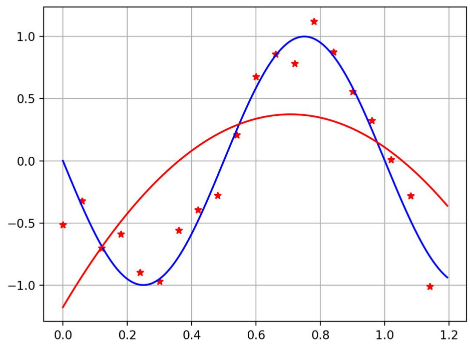
```python

    reg = LinearRegression().fit(Xa, yapp)

    plot_confusion(Xt, ytest, reg, "Confusion")


if __name__ == "__main__":
    main()
```

## 1.2 - Résultats

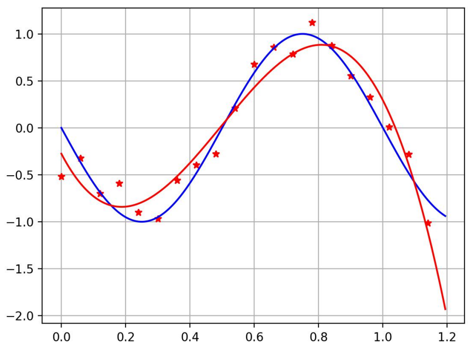
On lance alors le script et on enregistre les images. On peut alors observer les modèles de différentes complexité :



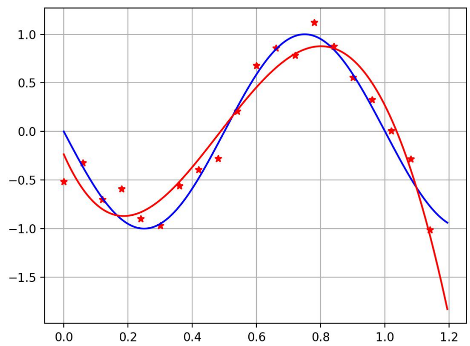

(a) Complexité = 1



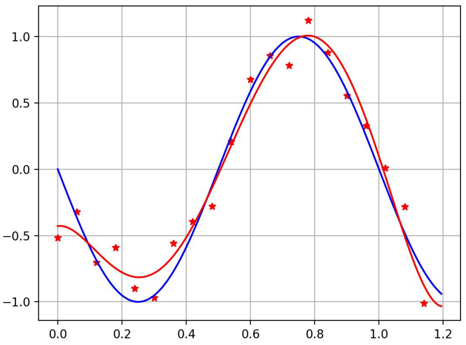

(b) Complexité = 2



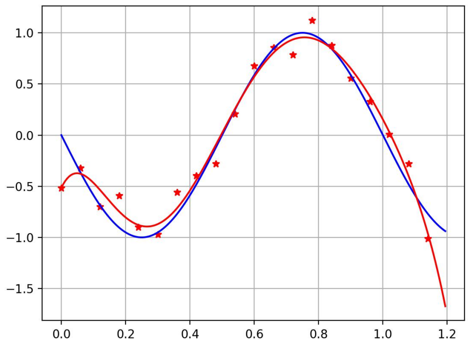

(c) Complexité = 3



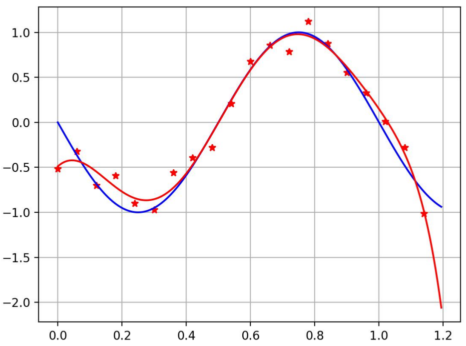

(d) Complexité = 4



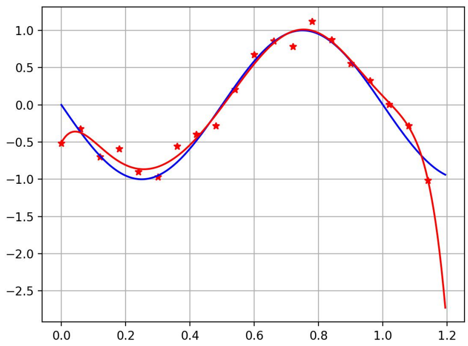

(a) Complexité = 5



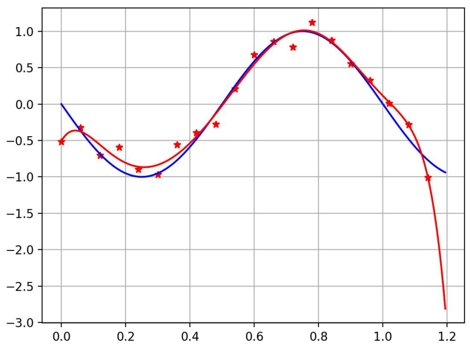

(b) Complexité = 6



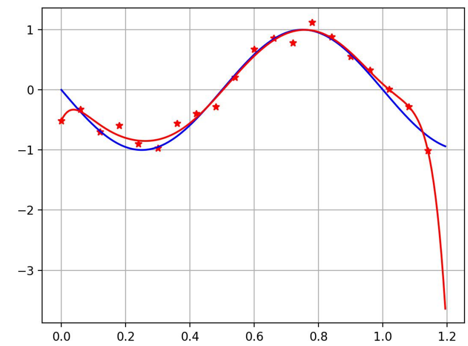

(c) Complexité = 7



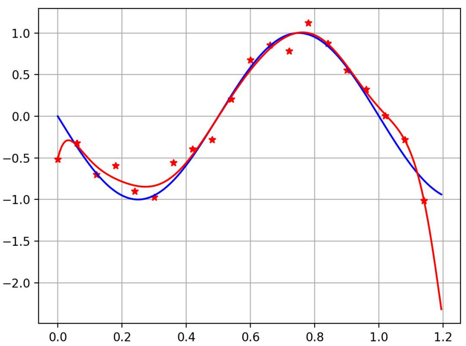

(d) Complexité = 8



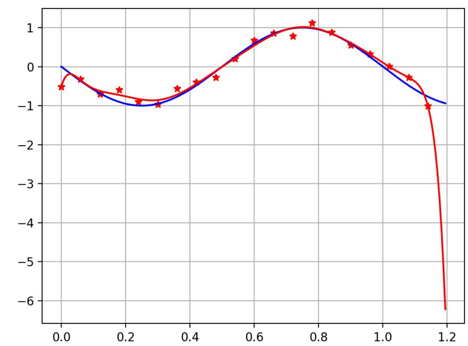

(e) Complexité = 9




(f) Complexité = 10




(g) Complexité = 11




(h) Complexité = 12

On peut alors regarder l'erreur entre le modèle, les données d'entrainement et les données de test

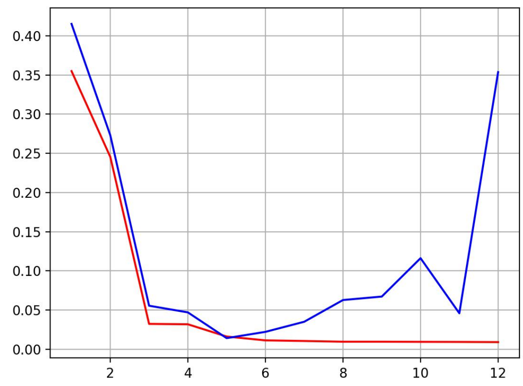



FIGURE 1.3 – En rouge : erreur sur les données de test, en bleu : erreur sur les données d'apprentissage

On peut remarquer que l'erreur d'apprentissage continue de diminuer alors que l'erreur de test diminue puis augmentant à partir d'une complexité de 5. Le taille de modèle optimale est donc 5.

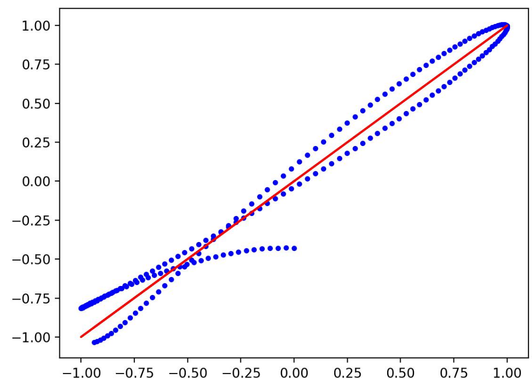
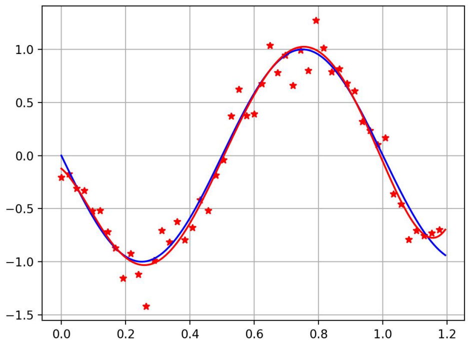


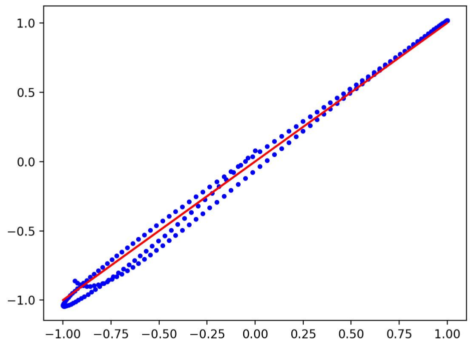

FIGURE 1.4 – Courbe de confusion pour la complexité 5, en bleu et la bissectrice, en rouge

On remarque que notre système n'est pas parfait, on a une approximation qui est proche mais qu'un erreur subsiste.

On change le nombre d'exemples de 20 à 50 et on refait le même entraînement. Le modèle optimale est alors de complexité 6, on remarque que ce modèle approxime mieux la courbe d'origine qu'avec 20 exemple :



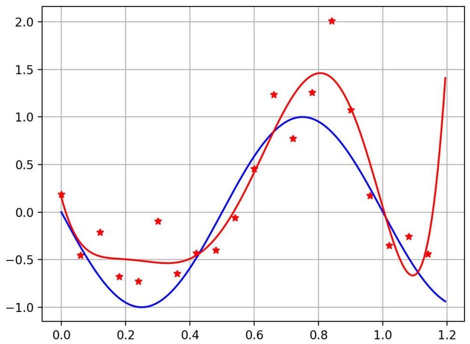

(a) 50 exemples, complexité 6



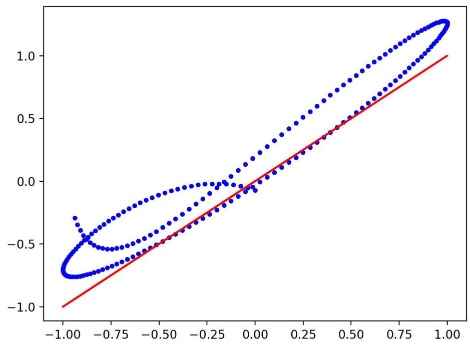

(b) Courbe de confusion avec 50 exemples

Un grand nombre de données d'entraînement permet donc d'augmenter la précision du modèle.

On augmente le bruit dans les mesures en repassant à 20 points.



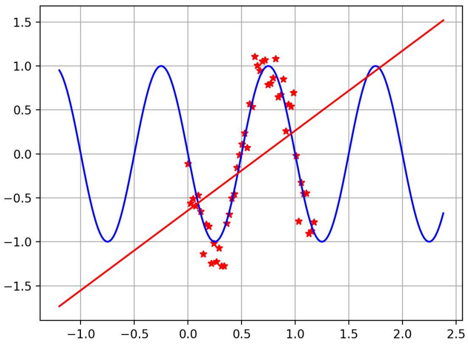

(a) Courbe optimale avec $\sigma = 0.5$ pour une complexité de 6



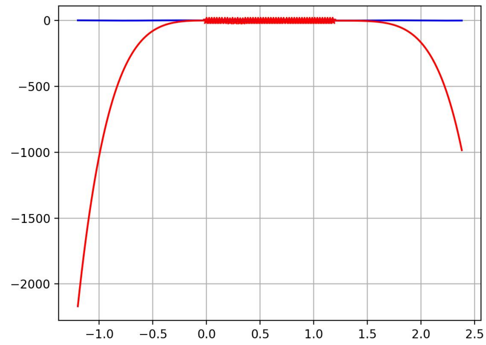

(b) Courbe de confusion avec $\sigma = 0.5$

On remarque que le bruit à un gros impact, la courbe optimale est très loin de correspondre à la courbe voulue car les points sont très éloignés. On retrouve cette erreur dans la courbe de confusion ou on est très loin d'être aligné avec la 1ère bissectrice.

On va maintenant agrandir la fenêtre des tests pour étudier comment cela se passe quand on limite les points sur un petit intervalle. On utilise une courbe de sinus dans l'intervalle -1.2 à 2.4 et on a que des points dans l'intervalle 0 à 1.2.




(a) Courbe optimale : complexité de 1




(b) Complexité 6

On remarque que les résultats sont aberrants. En effet, on a des effets de bords qui ne sont pas justes. Il est nécessaire, pour avoir une bonne régression, d'avoir des points un peu partout sur l'intervalle.

# 2 | Multilayer Perceptron

## 2.1 - Implémentation

On complète le squelette du code proposé et on ajoute quelques fonctions :

```
from sklearn.neural_network import MLPRegressor
import matplotlib.pyplot as plt
import numpy as np

from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D
from matplotlib.ticker import LinearLocator, FormatStrFormatter


def genere_exemple_dim1(xmin, xmax, NbEx, sigma):
    x = np.arange(xmin, xmax, (xmax - xmin) / NbEx)
    y = np.sin(-np.pi + 2 * x * np.pi) + np.random.normal(
        loc=0, scale=sigma, size=x.size
    )
    return x.reshape(-1, 1), y


def getMSE(x, y, reg):
    return sum(pow(reg.predict(x) - y, 2)) / x.shape[0]


def plot_model(Xa, Ya, Xt, Yt, reg, nameFig):
    Ypred = reg.predict(Xt)
    plt.plot(Xa[:, 1], Ya, "*r")
    plt.plot(Xt[:, 1], Yt, "-b")
    plt.plot(Xt[:, 1], Ypred, "-r")
    plt.grid()
    plt.savefig(nameFig + ".jpg", dpi=200)
    plt.close()


def plot_error_profile(L_error_app, L_error_test, nameFig):
```

```python
    plt.plot(range(1, len(L_error_app) + 1), L_error_app, "-r")
    plt.plot(range(1, len(L_error_test) + 1), L_error_test, "-b")
    plt.grid()
    plt.savefig(nameFig + ".jpg", dpi=200)
    plt.close()


def plot_confusion(Xt, Yt, reg, nameFig):
    plt.plot(Yt, reg.predict(Xt), ".b")
    plt.plot(Yt, Yt, "-r")
    plt.savefig(nameFig + ".jpg", dpi=200)
    plt.close()


def open_file(file_name) -> np.array:
    with open(file_name, "r") as file:
        file.readline()
        data = np.loadtxt(file, delimiter=";")
    return data


def sinus_cardinal(vect: np.array = None) -> float:
    A = np.array([[1, 1], [-2, 1]])
    b = np.array([0.2, -0.3])
    x = -np.pi + 2 * vect * np.pi
    z = A.dot(x + b)
    h = np.sqrt(np.transpose(z).dot(z))
    if np.abs(h) < 0.001:
        return 1
    else:
        return np.sin(h) / h


def plot_surf(figname, regr=None, show=False, save=False):
    step_v = 0.005

    x1v = np.arange(0, 1, step_v)
    x2v = np.arange(0, 1, step_v)
    Xv, Yv = np.meshgrid(x1v, x2v)

    R = np.zeros(Xv.shape)
    for i, x1 in enumerate(x1v):
        for j, x2 in enumerate(x2v):
            if not regr:
                R[i, j] = sinus_cardinal(np.array([x1, x2]))
            else:
                R[i, j] = regr.predict(np.array([[x1, x2]]))[0]

```

```python
    fig = plt.figure()
    ax = plt.axes(projection="3d")

    # Plot the surface
    if regr:
        label = "model"
    else:
        label = "sinc 2D"
    surf = ax.plot_surface(
        Xv, Yv, R, cmap=cm.coolwarm, linewidth=0, antialiased=False, label=label
    )

    ax.set_zlim(-1.01, 1.01)
    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
    plt.legend()
    fig.colorbar(surf, shrink=0.5, aspect=5)
    if save:
        plt.savefig(figname, dpi=300)
    if show:
        plt.show()
    plt.close()


def plot_app_test(Xapp, Yapp, Xtest, Ytest, show=False, save=False, figname=None):
    fig = plt.figure()
    ax = plt.axes(projection="3d")

    # Plot the surface

    ax.scatter(Xtest[:, 0], Xtest[:, 1], Ytest, color="b", label="Test")
    ax.scatter(Xapp[:, 0], Xapp[:, 1], Yapp + 0.001, color="r", label="Apprentissage")

    ax.set_zlim(-1.01, 1.01)
    ax.zaxis.set_major_locator(LinearLocator(10))
    ax.zaxis.set_major_formatter(FormatStrFormatter("%.02f"))
    plt.legend()
    if save:
        if figname == None:
            raise ValueError("No figname given")
        plt.savefig(figname + ".jpg", dpi=300)
    if show:
        plt.show()
    plt.close()

```

```python

def main():
    Xapp = open_file("sinc_dim2_input.csv")
    Yapp = np.array([sinus_cardinal(Xapp[i]) for i in range(np.shape(Xapp)[0])])
    plot_surf("Sin_card_plot", save=True)

    num_points = 40
    grid_line = np.linspace(0, 1, num_points)

    Xtest = np.zeros((num_points**2, 2))

    for i in range(num_points):
        for j in range(num_points):
            Xtest[i * num_points + j] = [grid_line[i], grid_line[j]]

    Ytest = np.array([sinus_cardinal(Xtest[i]) for i in range(np.shape(Xtest)[0])])

    plot_app_test(
        Xapp=Xapp,
        Yapp=Yapp,
        Xtest=Xtest,
        Ytest=Ytest,
        show=True,
        save=True,
        figname="app_test_MLP",
    )

    L_error_app = []
    L_error_test = []
    reg_list = []
    hidden_layer_max = 30

    for i in range(1, hidden_layer_max + 1):
        print("hidden_layer size = ", i)

        # Création du modèle linéaire
        reg = MLPRegressor(
            hidden_layer_sizes=(i,),
            max_iter=2000,
            activation="tanh",
            solver="lbfgs",
            tol=0.0001,
        ).fit(Xapp, Yapp)
        reg_list.append(reg)

        # Estimation des erreurs d'apprentissage et de test
```

```python
        L_error_app.append(getMSE(Xapp, Yapp, reg))
        L_error_test.append(getMSE(Xtest, Ytest, reg))

        # plot du model de degré i
        # plot_model(Xa, yapp, Xt, ytest, reg, "Model_%02d" % i)

    best = np.argmin(L_error_test) + 1
    best_reg = reg_list[best - 1]
    print("Meilleur model -> hidden_size =", best)
    plot_error_profile(L_error_app, L_error_test,
     ↪  "Profil_Err_App_Test_MLP")

    plot_surf("Meilleur model MLP", regr=best_reg, save=True)

    plot_confusion(Xtest, Ytest, best_reg, "Confusion_MLP")


if __name__ == "__main__":
    main()

```

Comme le problème est petit, on utilise une seule couche et on suppose qu'à partir de 2000 itération, on aura convergé, malgré le fait que notre critère est petit. Nous faisons les choix arbitraires d'utiliser tanh en fonction d'activation et le solveur lbfgs.

## 2.2 - Résultats

Dans un premier temps, on peut observer la fonction sinus cardinal généré par le script.

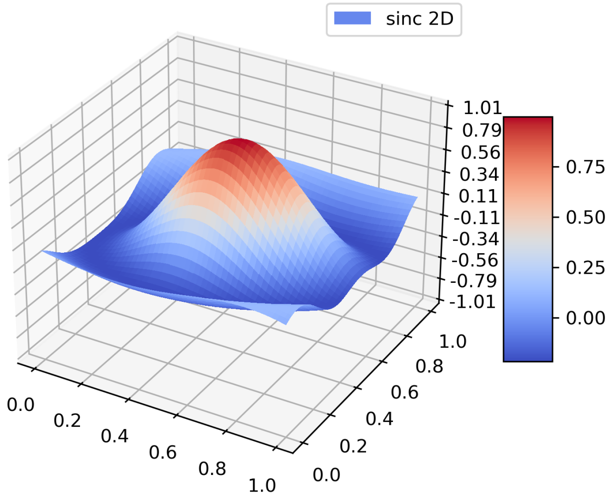
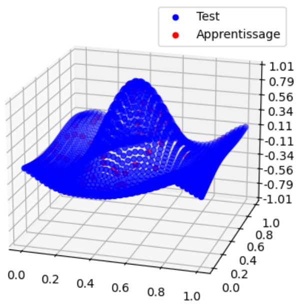
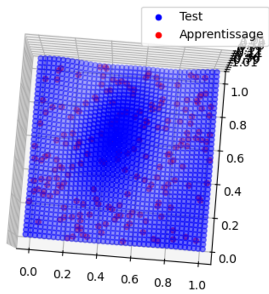



FIGURE 2.1 – Sinus cardinal 2D

On peut alors visualiser l'ensemble d'apprentissage et de test du réseau de neurone.




(a) Ensembles d'apprentissage et de test




(b) Vue du dessus

On entraîne ensuite un MLP avec une couche caché de taille variant entre 1 et 30 neurones, tanh comme fonction d'activation, 2000 itérations max pour converger, une tolérance de 0.0001 et lbfgs comme solveur. Le meilleur modèle est atteint pour une taille de couche caché de 26.

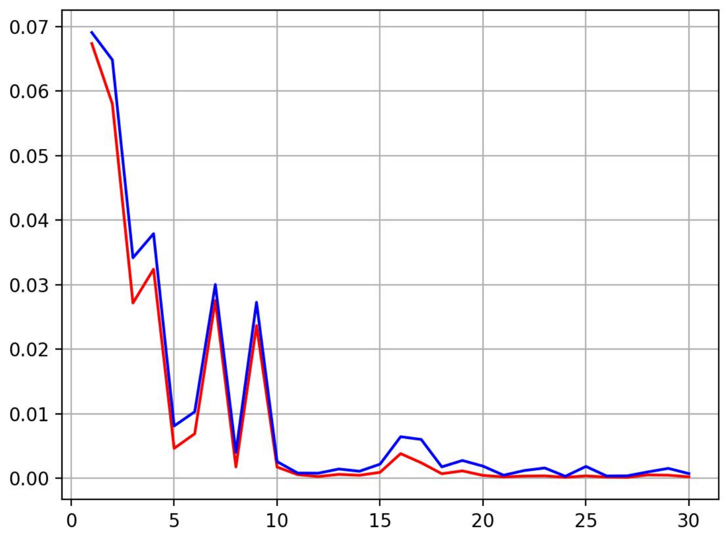
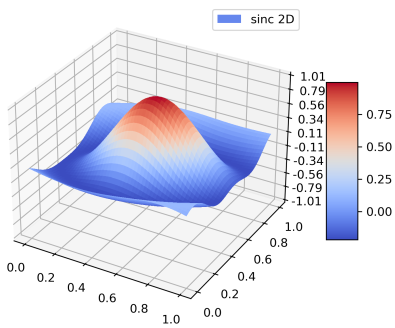
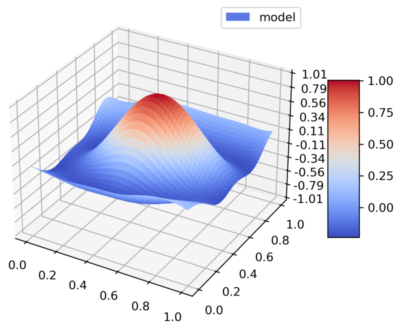



FIGURE 2.3 – Profil de l'erreur sur les ensembles d'apprentissage et de test en fonction de la taille de la couche caché

On peux alors regarder les surfaces générées par le test et la prédiction. A l'œil nue, il est difficile de distinguer une différence.




(a) Surface générée par Ytest




(b) Surface générée par Ypred (hidden_size = 26)

Enfin, on regarde la confusion du modèle généré.

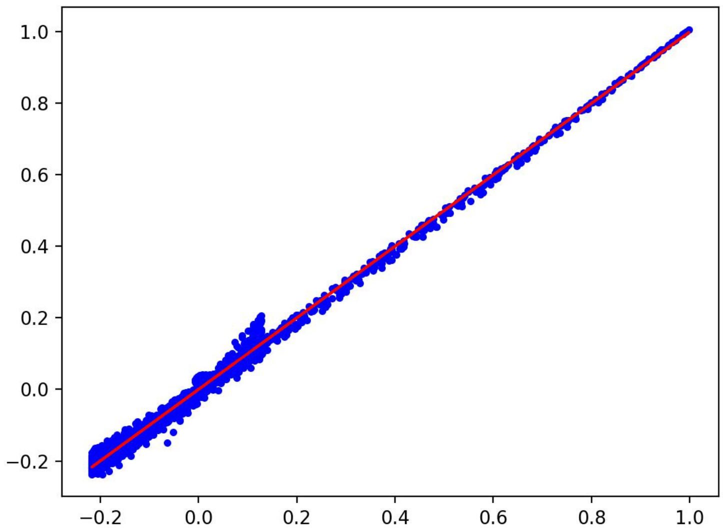



FIGURE 2.5 – Confusion du MLP avec une couche caché de taille 26

On remarque que la courbe de confusion est très proche de la première bissectrice. On a donc un modèle qui reproduit très bien notre sinus cardinal sur l'intervalle voulu.

# Conclusion

Lors de ce TP, nous avons pu étudier des modèles polynomiaux pour une fonction simple (sin) puis un estimateur non linéraire (multilayer perceptron) qui permet d'estimer une fonction plus complexe en 2D avec une couche cachée.