

# Rapport interactif

## Sommaire

### I. Implémentation sur notre dataset

### II. Cas de tests

## Rappel du sujet

Nous essayerons de répondre à la problématique:

- **Quel est ce monument parisien ?**

À partir de notre dataset.

## Dataset

Nous avons utilisés une extension Chrome qui récupère entre 400 et 500 images à partir d'une recherche, puis nous avons nettoyés certaines images incohérentes.

## Librairie

Nous avons réalisé **ML**, notre propre librairie de machine learning en C++. Elle dépend de la bibliothèque **Eigen**, qui nous sert à réaliser plus facilement des calculs matriciels.

## Application web

Nous avons décidé de réaliser notre application web en Python en utilisant le micro-framework **Flask** ainsi que **Bootstrap** et **JQuery**

## Étapes de développement

1. Implémentation du perceptron
2. Interopérabilité du perceptron
3. Implémentation du perceptron multi-couches
4. Interopérabilité du perceptron multi-couches
5. Validation sur les cas de tests et correction de la librairie
6. Rédaction de la première version du rapport pour l'étape n°2
7. Création du site web
8. Ajout des fonctionnalités save et load sur le perceptron
9. Interopérabilité du save et du load sur le perceptron
10. Ajout des fonctionnalités save et load sur le perceptron multi-couches
11. Interopérabilité du save et du load sur le perceptron multi-couches
12. Implémentation du Radial Basis Function Network
13. Interopérabilité du Radial Basis Function Network
14. Amélioration et finition du site web (*en cours*)
15. Rédaction du rapport interactif (*en cours*)

## Difficultés rencontrées

- Problème d'initialisation des valeurs aléatoires, on avait des modèles tous identiques.
- Tentative d'utiliser `std::random_device` pour obtenir des valeurs aléatoires plus uniformes. Cela fonctionnait mais on s'est rendus compte que `rand()` de `stdlib` suffisait pour notre besoin.
- Confusions due au biais (on a tenté de le supprimer plusieurs fois)
- Erreur d'allocation de mémoire entraînant des erreurs lors du passages des objets par l'interopérabilité
- Destruction d'une instance d'un perceptron multi-couches, mal paramétré entraînant des erreurs lors de l'utilisation de l'interoperabilité.

- Prise en main de NumPy.

## Partie 1. Implémentation sur notre dataset

Dans cette partie, nous montrerons l'étape de pré-traitement des données (*data preprocessing*), puis nous appliquerons notre modèle linéaire ainsi que notre perceptron multi-couches à notre dataset.

### a) Importer le dataset

In [1]:

```
import os
import numpy as np
from PIL import Image
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from IPython.display import clear_output
from tensorflow.keras.losses import CategoricalCrossentropy

from ml import *
from rbfn import *
```

Nous définissons les constantes dont nous aurons besoin, tout au long du rapport.

In [2]:

```
IMG_SIZE = (64, 64)
PATH = os.path.join("data_large/")
TRAIN = os.path.join(PATH, "train")
classes = os.listdir(TRAIN)
```

In [3]:

```
print(f"Les différentes classes possibles sont: {', '.join(classes)}")
```

Les différentes classes possibles sont: arc-de-triomphe, hotel-de-ville, jardin-des-tuilleries, moulin-rouge, musée-d-orsay, palais-de-l-elysee, place-de-la-concorde, pont-neuf

In [4]:

```
def random_rotation(image):
    """
    Random rotation of the image
    """
    rand_rot = np.random.uniform(-25, 25)
    return image.rotate(rand_rot)
```

In [5]:

```
def horizontal_flip(image):
    """
    Flip the image horizontally
    """
    return image.transpose(Image.FLIP_LEFT_RIGHT)
```

In [6]:

```
transformations = {
    "rotate": random_rotation,
    "flip": horizontal_flip,
}

num_transformations_to_apply = np.random.randint(1, len(transformations)+1)
print(num_transformations_to_apply)
```

1

In [7]:

```
def showImg(tensor, label, prediction):
    """
    Affiche une image avec sa prediction et son label
    """
    fig, ax = plt.subplots()
    ax.imshow(tensor.reshape((IMG_SIZE[0], IMG_SIZE[1], 3)))
    ax.set_title(f'Label: {classes[np.argmax(prediction)]}')
    ax.set_xlabel(f'Prediction: {np.argmax(prediction)} / Expected output: {np.argmax(label)}')
    ax.set_xticks([])
    ax.set_yticks([])
    return ax
```

In [8]:

```
def import_images_and_assign_labels(folder, label, X, Y, IMG_SIZE=IMG_SIZE, data_aug=False):
    """
    Convertit et redimensionne les images d'un dossier en NumPy Array.
    """
    for file in os.listdir(folder):
        image_path = os.path.join(folder, file)
        im = Image.open(image_path)
        im = im.resize(IMG_SIZE)
        if data_aug == True:
            for i in range(num_transformations_to_apply):
                k = np.random.choice(list(transformations))
                transformed_img = transformations[k](im)
                transformed_img = transformed_img.convert("RGB")
                transformed_arr = np.array(transformed_img)
                transformed_arr = np.reshape(transformed_arr, (IMG_SIZE[0]* IMG_SIZE[1] * 3,))
                X.append(transformed_arr)
                Y.append(label)
        im = im.convert("RGB")
        im_arr = np.array(im)
        im_arr = np.reshape(im_arr, (IMG_SIZE[0]* IMG_SIZE[1] * 3,))
        X.append(im_arr)
        Y.append(label)
```

In [9]:

```
for s in ["train", "valid", "test"]:
    if s == "test":
        print(f"Nombre d'images dans le {s} set:")
        print(f"{len(os.listdir(os.path.join(PATH, s)))} images.")
    else:
        print(f"Nombre d'images par classes dans le {s} set:")
        res = 0
        for cl in classes:
            print(f"- {cl}: {len(os.listdir(os.path.join(PATH, s, cl)))} images.")
            res+=len(os.listdir(os.path.join(PATH, s, cl)))
        print("Total :", res, "images.")
        print()
```

Nombre d'images par classes dans le train set:

- arc-de-triomphe: 471 images.
- hotel-de-ville: 359 images.
- jardin-des-tuileries: 393 images.
- moulin-rouge: 391 images.
- musee-d-orsay: 418 images.
- palais-de-l-elysee: 441 images.
- place-de-la-concorde: 457 images.
- pont-neuf: 487 images.

Total : 3417 images.

Nombre d'images par classes dans le valid set:

- arc-de-triomphe: 129 images.

```
- hotel-de-ville: 129 images.  
- jardin-des-tuilleries: 129 images.  
- moulin-rouge: 129 images.  
- musee-d-orsay: 129 images.  
- palais-de-l-elysee: 129 images.  
- place-de-la-concorde: 129 images.  
- pont-neuf: 129 images.  
Total : 1032 images.
```

Nombre d'images dans le test set:  
1040 images.

In [10]:

```
def import_dataset(IMG_SIZE=IMG_SIZE, data_aug=False):  
    """  
    Crée les datasets d'entraînement et de validation  
    """  
    X_train, y_train, X_valid, y_valid = [], [], [], []  
    labels = np.identity(len(os.listdir(TRAIN)))  
    for set_type in ["train", "valid"]:  
        for cl, lab in zip(classes, labels):  
            if set_type == "train":  
                X_set, y_set = X_train, y_train  
                import_images_and_assign_labels(  
                    os.path.join(PATH, set_type, cl),  
                    lab,  
                    X_set,  
                    y_set,  
                    IMG_SIZE,  
                    data_aug  
                )  
            else:  
                X_set, y_set = X_valid, y_valid  
                import_images_and_assign_labels(  
                    os.path.join(PATH, set_type, cl),  
                    lab,  
                    X_set,  
                    y_set,  
                    IMG_SIZE,  
                    data_aug  
                )  
  
    return (np.array(X_train) / 255.0, np.array(y_train)), \  
           (np.array(X_valid) / 255.0, np.array(y_valid))
```

In [11]:

```
(X_train, y_train), (X_valid, y_valid) = import_dataset(data_aug=True)
```

## Affichage des images du training set

In [12]:

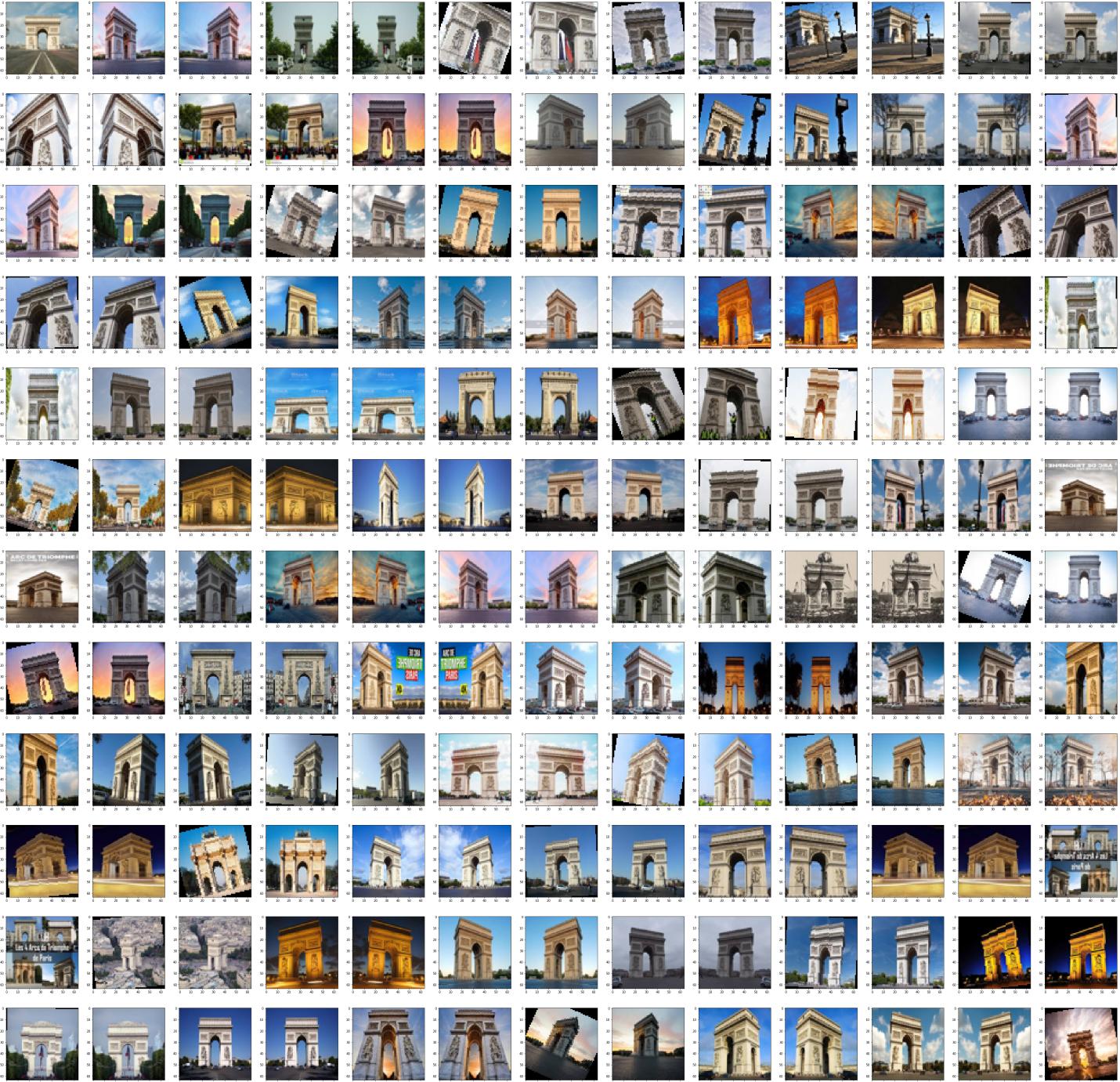
```
print(f"Nombre d'images après importation des données : {len(X_train)}")
```

Nombre d'images après importation des données : 6834

In [13]:

```
fig = plt.figure(figsize=(64, 64))  
columns = 13  
rows = 12  
for i in range(1, columns*rows +1):  
    img = X_train[i].reshape((IMG_SIZE[0], IMG_SIZE[1], 3))  
    fig.add_subplot(rows, columns, i)
```

```
plt.imshow(img)
plt.show()
```



In [14]:

```
for s in ["train", "valid", "test"]:
    if s == "test":
        print(f"Nombre d'images dans le {s} set:")
        print(f"{len(os.listdir(os.path.join(PATH, s)))} images.")
    else:
        print(f"Nombre d'images par classes dans le {s} set:")
        res = 0
        for cl in classes:
            print(f"- {cl}: {len(os.listdir(os.path.join(PATH, s, cl)))} images.")
            res+=len(os.listdir(os.path.join(PATH, s, cl)))
        print("Total : ", res, "images.")
    print()
```

Nombre d'images par classes dans le train set:

- arc-de-triomphe: 471 images.
- hotel-de-ville: 359 images.
- jardin-des-tuileries: 393 images.
- moulin-rouge: 201 images

moulin-rouge. 391 images.  
- musee-d-orsay: 418 images.  
- palais-de-l-elysee: 441 images.  
- place-de-la-concorde: 457 images.  
- pont-neuf: 487 images.  
Total : 3417 images.

Nombre d'images par classes dans le valid set:

- arc-de-triomphe: 129 images.  
- hotel-de-ville: 129 images.  
- jardin-des-tuileries: 129 images.  
- moulin-rouge: 129 images.  
- musee-d-orsay: 129 images.  
- palais-de-l-elysee: 129 images.  
- place-de-la-concorde: 129 images.  
- pont-neuf: 129 images.  
Total : 1032 images.

Nombre d'images dans le test set:

1040 images.

## Affichage d'une image test

In [15]:

```
picture_test = np.random.randint(0, len(X_valid)-1)
showImg(X_valid[picture_test], y_valid[picture_test], [0, 0, 0]);
```

Label: arc-de-triomphe



Prediction: 0 / Expected output: 3

## b) Appliquer le modèle linéaire au dataset

Cette section est incomplète.

In [16]:

```
input_dim = len(X_train[0])
```

In [17]:

```
p_model = create_linear_model(input_dim)
```

In [18]:

```
picture_test_linear = np.random.randint(0, len(X_train))
test_before = predict_linear_model_classif(p_model, input_dim, X_train[picture_test_linear])
print("Before training:", test_before)
```

Before training: -1.0

```
In [19]:
```

```
train_linear_classification_model(p_model, input_dim, X_train, y_train.flatten())
```

```
In [20]:
```

```
test_after = predict_linear_model_classif(p_model, input_dim, X_train[picture_test])
print("After training:", test_after)
```

```
After training: 1.0
```

```
In [21]:
```

```
destroy_linear_model(p_model)
```

## c) Appliquer le PMC au dataset

Nous allons créer un petit modèle contenant une seule couche cachée puis nous allons l'entraîner pour voir s'il sera capable de prédire correctement une image de la Place de la Concorde, notre image test.

```
In [22]:
```

```
NUM_CLASSES = len(classes)
picture_test = 411
input_dim = [len(X_train[0]), 32, NUM_CLASSES]
```

```
In [23]:
```

```
p_model, len_output_layer = create_mlp_model(input_dim)
```

```
In [24]:
```

```
def accuracy(model):
    """
    Evaluate notre modèle sur les données d'entraînement et de validation.
    """
    true_preds = 0
    total_preds = len(X_train)
    for x, y in zip(X_train, y_train):
        if np.argmax(predict_mlp_model_classification(model, x, NUM_CLASSES)) == np.argmax(y):
            true_preds += 1
    print(f"Accuracy training: {round((true_preds / total_preds) * 100, 2)}%")

    true_preds = 0
    total_preds = len(X_valid)
    for x, y in zip(X_valid, y_valid):
        if np.argmax(predict_mlp_model_classification(model, x, NUM_CLASSES)) == np.argmax(y):
            true_preds += 1
    print(f"Accuracy valid: {round((true_preds / total_preds) * 100, 2)}%)
```

Voyons si notre modèle non entraîné arrive à prédire correctement une image aléatoire du valid set.

```
In [25]:
```

```
test_before = predict_mlp_model_classification(p_model, X_valid[picture_test], input_dim[-1])

showImg(X_valid[picture_test], y_valid[picture_test], test_before);
print("Prediction:", test_before)
```

```
Prediction: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

Label: arc-de-triomphe



Prediction: 0 / Expected output: 3

**Nous pouvons voir que notre modèle prédit correctement le monument environ 1 fois sur 8 (soit 12.5%). Nous avons 8 classes de monuments, ce qui veut dire que nos résultats sont totalement normaux, car notre modèle n'est pas encore entraîné.**

In [26]:

```
accuracy(p_model)
```

```
Accuracy training: 13.78%
Accuracy valid: 12.5%
```

**Entraînons désormais notre modèle. Par soucis de temps, nous allons entraîner manuellement un petit modèle de 1000 époques et puis nous chargerons un gros modèle déjà entraîné par nos soins.**

In [27]:

```
train_classification_stochastic_gradient_backpropagation_mlp_model(p_model, X_train, y_train.flatten(), epochs=1000, alpha=0.01)
```

**Voyons si notre modèle entraîné arrive à prédire correctement une image aléatoire du valid set.**

In [28]:

```
test_after = predict_mlp_model_classification(p_model, X_valid[picture_test], input_dim[-1])

showImg(X_valid[picture_test], y_valid[picture_test], test_after);

print("Prediction:", test_after)
```

```
Prediction: [1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0, 1.0]
```

Label: arc-de-triomphe



Prediction: 0 / Expected output: 3

**Prédiction correcte : Etait-ce un coup de chance ou notre modèle réussit désormais à distinguer les différents monuments ?**

**Prédiction incorrecte :** Notre modèle n'as pas réussi à prédire notre image test, as-t-il réellement appris quelque chose durant sa phase d'entraînement ? Notre modèle est potentiellement entrain de sous-apprendre.

Pour en avoir le coeur net, voyons comment il s'en sort face à toutes les données du train et du valid set. Nous aurons un meilleur point de vue de son avancé.

In [29]:

```
accuracy(p_model)
destroy_mlp_model(p_model)
```

```
Accuracy training: 13.78%
Accuracy valid: 12.5%
```

**Prédiction correcte :** ^^ C'était un coup de chance, pas de bol. Le pourcentage est beaucoup trop faible, notre modèle prédit toujours l'équivalent d'une chance sur 8.

**Prédiction incorrecte :** C'est tout à fait normal, notre modèle n'a pas été assez entraîné.

Passons à la partie intéressante, nous allons charger un modèle pré-entraîné et nous allons effectuer le même test.

Pour commencer, nous allons devoir ré-importer une nouvelle fois les données mais avec une taille d'image différente, nous allons cette fois utiliser des images plus petites (8x8).

In [30]:

```
IMG_SIZE = (8, 8)
```

In [31]:

```
(X_train, y_train), (X_valid, y_valid) = import_dataset(IMG_SIZE)
```

Assurons-nous que les images soit à la bonne dimension.

In [32]:

```
print("Dimensions des images applatis : ", X_train[0].shape)
print("Dimensions souhaités : ", (IMG_SIZE[0]*IMG_SIZE[1]*3,))
```

```
Dimensions des images applatis : (192,)
Dimensions souhaités : (192,)
```

Le prochain modèle que nous allons chargé a été entraîné 10 millions d'époques et ne contient aucune couche cachée.

In [33]:

```
p_model2 = load_mlp_model("models/mlp/MLP_10000000_8x8_8_t_acc-65.5_v_acc-58.04.txt")
```

Est-ce que le modèle entraîné 10 millions d'époques sera capable de prédire notre image sélectionnée aléatoirement. La suite à la prochaine cellule!

In [34]:

```
test_load = predict_mlp_model_classification(p_model2, X_valid[picture_test], input_dim[-1])

showImg(X_valid[picture_test], y_valid[picture_test], test_load);

print("Prediction:", test_load)
```

```
Prediction: [0.23109108, 0.20234938, 0.43612346, -0.08851555, 0.11945536, -0.12421525, 0.2687144, 0.032881748]
```

Label: jardin-des-tuileries



Prediction: 2 / Expected output: 3

**Prédiction correcte : FIIIOUUU ! On a pas travaillé 3 mois pour rien ;)**

**Prédiction incorrecte : Voyons sur toutes les données comment notre modèle réagit. (3 mois à la poubelle ):-)**

In [35]:

```
accuracy(p_model2)
destroy_mlp_model(p_model2)
```

Accuracy training: 4.24%  
Accuracy valid: 5.62%

**On peut voir que notre modèle pré-entraîné à un peu plus de 1 chance sur 2 de prédire correctement le monument parisien sur des images qu'il n'a jamais vu.**

**On aurait de meilleures prédictions si nous avons un modèle soit plus gros, soit plus entraîné. Essayons.**

**Le prochain modèle que nous allons chargé à était entraîné 15 millions d'époques et ne contient 3 couches cachées (16, 8, 8).**

In [36]:

```
p_model3 = load_mlp_model("models/mlp/overfit/MLP_15000000_16x16_16_8_8_8_t_acc-80.6_v_ac
c-57.27.txt")
```

**Est-ce que le modèle entraîné 15 millions d'époques sera capable de prédire notre image sélectionnée aléatoirement. La suite à la prochaine cellule!**

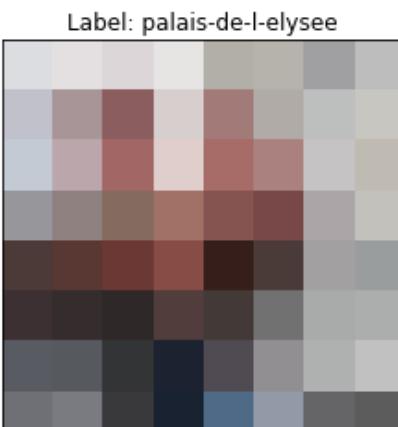
In [37]:

```
test_p3 = predict_mlp_model_classification(p_model3, X_valid[picture_test], input_dim[-1])

showImg(X_valid[picture_test], y_valid[picture_test], test_p3);

print("Prediction:", test_p3)
```

Prediction: [0.0050210035, 0.035364542, 0.036835983, 0.022681413, 0.0068340665, 0.7810085  
4, -0.10207913, 0.02421058]



**Réponse logique, oui. Maintenant, voyons sur toutes les données.**

In [38]:

```
accuracy(p_model3)
destroy_mlp_model(p_model3)
```

Accuracy training: 3.48%  
Accuracy valid: 6.4%

Aïe, notre modèle a sur-appris. :(

---

## d) Découverte de nouveaux modèles

Pour faciliter notre recherche d'architectures, nous avons décidé de créer un équivalent au **GridSearch** de Sklearn.

In [39]:

```
import random
```

On insère les hyperparamètres que l'on souhaite essayer sur notre modèle à la prochaine cellule.

In [40]:

```
number_of_hidden_layers = [3, 4, 5]
hidden_layers = [32, 64, 128]
size_img = [(8, 8), (16, 16)]
EPOCHS = [1000]
alphas = [1e-2, 1e-3]
```

In [41]:

```
def accuracy_for_grid_search(model, len_output, X_tr, y_tr, X_val, y_val):
    """
    Evaluate notre modèle sur les données d'entraînement et de validation.
    """
    train_total, valid_total = len(X_train), len(X_valid)
    true_preds = 0
    total_preds = len(X_tr)
    for x, y in zip(X_tr, y_tr):
        if np.argmax(predict_mlp_model_classification(model, x, len_output)) == np.argmax(x):
            true_preds += 1
    train_acc = round((true_preds / total_preds) * 100, 2)

    true_preds = 0
    total_preds = len(X_val)
    for x, y in zip(X_val, y_val):
        if np.argmax(predict_mlp_model_classification(model, x, len_output)) == np.argmax(x):
            true_preds += 1
    valid_acc = round((true_preds / total_preds) * 100, 2)

    return train_acc, valid_acc
```

Notre fonction `grid_search()` a pour but de trouver le modèle qui a les meilleures résultats de prédictions et de les enregistrer.

```

def grid_search(epochs=EPOCHS, size_img=size_img, hidden_layers=hidden_layers, number_of_
hidden_layers=number_of_hidden_layers):
    """
    Equivalent au Grid Search de Sklearn.
    """
    max_train_acc, max_val_acc = 0.0, 0.0

    for ep in EPOCHS:
        for s in size_img:
            IMG_SIZE = s
            (X_train, y_train), (X_valid, y_valid) = import_dataset(IMG_SIZE=IMG_SIZE, d
ata_aug=True)
            input_dim = [len(X_train[0])]
            for al in alphas:
                for n, num_h in enumerate(number_of_hidden_layers):
                    h = random.choices(hidden_layers, k=num_h)
                    if h:
                        input_dim.extend(random.choices(hidden_layers, k=num_h))
            input_dim.append(NUM_CLASSES)

            model, last_output_layer = create_mlp_model(input_dim)

            for _ in range(ep):
                train_classification_stochastic_gradient_backpropagation_mlp_mod
el(model, X_train, y_train.flatten(), epochs=len(X_train), alpha=al)

                train_acc, valid_acc = accuracy_for_grid_search(model, last_output_l
ayer, X_train, y_train, X_valid, y_valid)

                if input_dim[1:-1] == []:
                    filename = f"models/mlp/MLP_{ep}_{al}_{IMG_SIZE[0]}x{IMG_SIZE[1]
}_{input_dim[-1]}_t_acc-{train_acc}_v_acc-{valid_acc}.txt"
                else:
                    filename = f"models/mlp/MLP_{ep}_{al}_{IMG_SIZE[0]}x{IMG_SIZE[1]
}_{''.join(map(str, input_dim[1:]))}_t_acc-{train_acc}_v_acc-{valid_acc}.txt"

                print(f"Params ~ epochs={ep},alpha={al},img_size={IMG_SIZE},input_di
m={input_dim[1:]} ~ t_acc: {train_acc}% / v_acc: {valid_acc}%")

                if train_acc > max_train_acc:
                    max_train_acc = train_acc
                    save_mlp_model(model, filename)
                if valid_acc > max_val_acc:
                    max_val_acc = valid_acc
                    save_mlp_model(model, filename)

            destroy_mlp_model(model)

            input_dim = [len(X_train[0])]

    print(f"Max Train acc: {max_train_acc}% / Max Valid_acc: {max_val_acc}%")



```

C'est parti, on lance la recherche d'architecture.

In [43]:

```
# grid_search()
```

Cette fonction nous a permis d'essayer toutes les possibilités des **hyperparamètres** de manière automatisé, de pouvoir entraîner plusieurs **modèles** toute une nuit et d'obtenir une sorte de recap à la fin du grid search.

Nous avons essayé tout un panel de taille d'image allant de 6x6 à 64x64, un nombre de couches cachées allant de 0 à 4, des nombres de noeuds par couche entre 4 et 1024 et enfin un nombre d'époques de 1000 allant jusqu'à 20M.

Aujourd'hui, à notre grande surprise, nous avons découvert que pour notre problématique, les modèles sans couches cachées et avec une petite taille d'images prédisent mieux que de **grand/gros** modèles avec une grande taille d'images.

**En plus de leur taille, les grands modèles prennent plus de temps à s'entraîner.**

**Nos modèles qui généralisent le mieux ont un taux de réussite d'environ 65% de précision sur des données inconnues.**

## e) Les courbes d'apprentissage

**Grâce à la fonction ci-dessous, nous pouvons afficher une courbe d'apprentissage pour la partie MLP.**

In [44] :

```
def display_mlp_curves(size_img, input_dim, epochs, alpha):
    """
    Courbe d'apprentissage d'un modèle MLP.
    """
    losses, val_losses, accs, val_accs = [], [], [], []
    model, last_output_layer = create_mlp_model(input_dim)

    (X_train, y_train), (X_valid, y_valid) = import_dataset(IMG_SIZE=size_img, data_aug=True)

    for ep in range(epochs):
        train_classification_stochastic_gradient_backpropagation_mlp_model(model, X_train, y_train.flatten(), epochs=len(X_train))

        y_preds = []
        y_true = []
        y_true_1 = []
        y_preds_1 = []

        for x, y in zip(X_train, y_train):
            preds = predict_mlp_model_classification(model, x, input_dim[-1])
            y_preds_1.append(preds)
            y_preds.append(np.argmax(preds))
            y_true_1.append(y)
            y_true.append(np.argmax(y))
        loss = CategoricalCrossentropy()
        losses.append(loss(y_true_1, y_preds_1))
        accs.append(accuracy_score(y_true, y_preds))

        y_preds = []
        y_true = []
        y_true_1 = []
        y_preds_1 = []

        for x, y in zip(X_valid, y_valid):
            preds = predict_mlp_model_classification(model, x, input_dim[-1])
            y_preds_1.append(preds)
            y_preds.append(np.argmax(preds))
            y_true_1.append(y)
            y_true.append(np.argmax(y))
        val_losses.append(loss(y_true_1, y_preds_1))
        val_accs.append(accuracy_score(y_true, y_preds))

    clear_output(True)

    print(f"MLP - {input_dim} - {epochs} ep. - {alpha} lr.")

    plt.plot(losses)
    plt.plot(val_losses)
    plt.legend(['loss', 'val_loss'], loc='upper left')
    plt.title('Evolution of loss (CCE)')
    plt.xlabel('epochs')
    plt.ylabel(f'categorical cross-entropy (softmax loss)')
    plt.show()

    plt.plot(accs)
```

```

plt.plot(val_accs)
plt.legend(['acc', 'val_acc'], loc='upper left')
plt.title('Evolution of accuracy')
plt.xlabel('epochs')
plt.ylabel(f'Accuracy (%)')
plt.show()

```

In [47]:

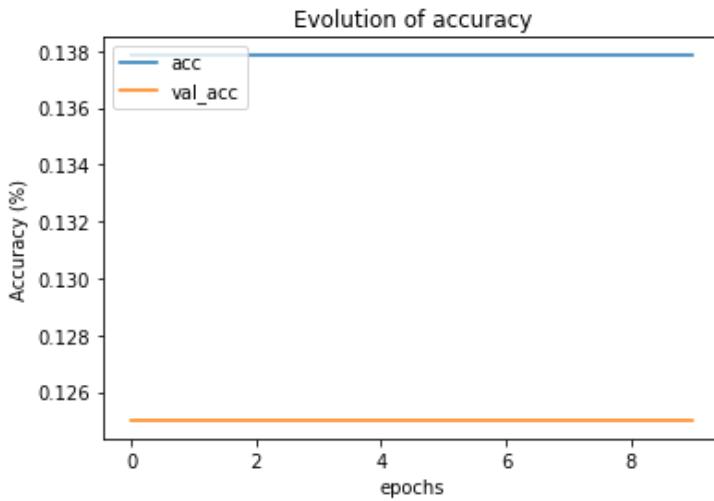
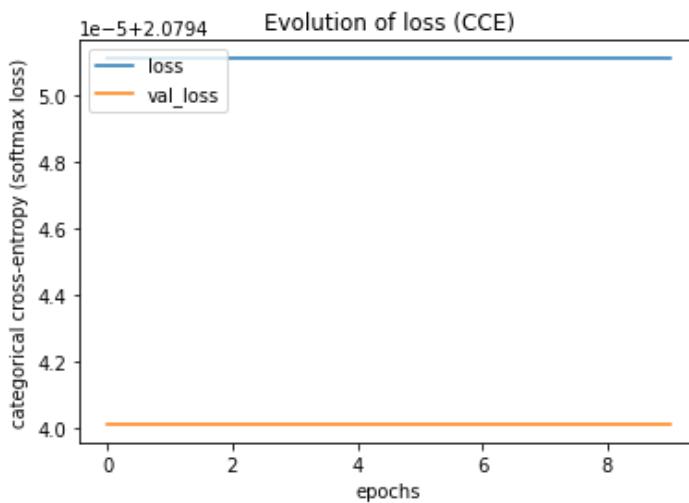
```

IMG_SIZE=(8, 8)

display_mlp_curves(size_img=IMG_SIZE, input_dim=[IMG_SIZE[0]*IMG_SIZE[1]*3, 32, 32, NUM_CLASSES], epochs=10, alpha=0.0001)

MLP - [192, 32, 32, 8] - 10 ep. - 0.0001 lr.

```



## Partie 2. Cas de tests

Avant de démarrer notre recherche d'architecture viable pour nos modèles, il a fallu s'assurer que les algorithmes de notre lib étaient corrects. Pour cela, nous avons utilisé des jeux de données où les résultats sont visuellement interprétables, ou bien dont les prédictions peuvent être calculées à la main sans y passer 4heures. L'idéal était donc de travailler sur des points en 2D ou 3D, avec une quantité limitée de point. Si nos modèles permettent de classifier nos points correctement, on pourra ensuite passer à de plus gros volumes de données, dans des dimensions impossible à interpréter pour l'Humain.

## Classification

## Linear Simple :

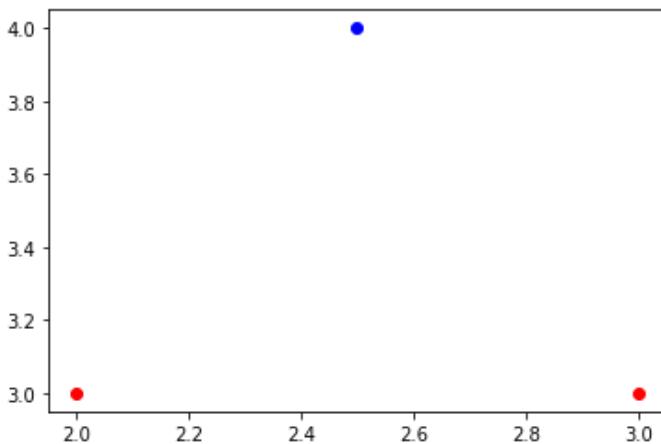
```
Linear Model : OK
MLP (2, 1)   : OK
```

In [48]:

```
X = np.array([
    [2.5, 4],
    [2, 3],
    [3, 3]
])
Y = np.array([
    1,
    -1,
    -1
])
```

In [49]:

```
plt.scatter(X[0, 0], X[0, 1], color='blue')
plt.scatter(X[1:3,0], X[1:3,1], color='red')
plt.show()
plt.clf()
```



<Figure size 432x288 with 0 Axes>

In [50]:

```
model_dim = len(X[0])
errors = 0

for _ in range(50):
    test_after = []
    p_model = create_linear_model(model_dim)

    train_linear_classification_model(p_model, model_dim, X, Y, alpha=0.001, epochs=10_000)

    for data, expected in zip(X, Y):
        out = predict_linear_model_classif(p_model, model_dim, data)
        test_after.append(out)
        if out != expected:
            errors += 1
    print(test_after)
    destroy_linear_model(p_model)

print(f"errors: {errors}")
```

```
[1.0, -1.0, -1.0]
[1.0, -1.0, -1.0]
[1.0, -1.0, -1.0]
[1.0, -1.0, -1.0]
[1.0, -1.0, -1.0]
[1.0, -1.0, -1.0]
```



In [56]:

```
model = create_linear_model(input_dim)
test_dataset = [[x1, x2] for x1 in np.arange(-1, 5, 0.5) for x2 in np.arange(-1, 5, 0.5)]
colors = ["blue" if output >= 0 else "red" for output in Y]

predicted_outputs = [predict_linear_model_classif(model, input_dim, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label == 1 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])

train_linear_classification_model(model, input_dim, flattened_dataset_inputs, Y, epochs=10000)

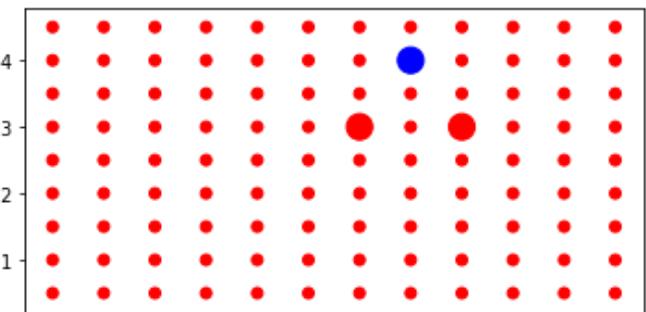
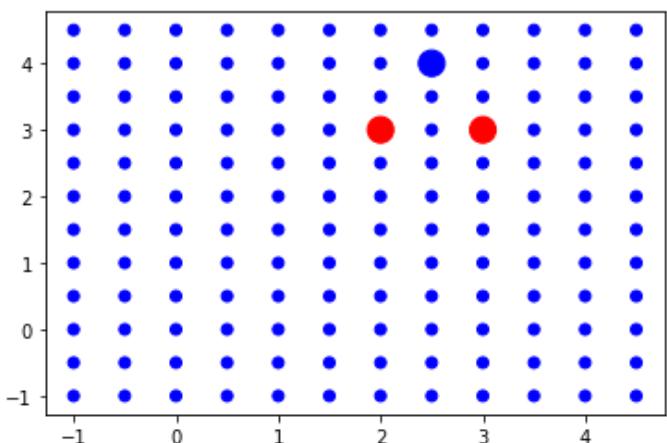
predicted_outputs = [predict_linear_model_classif(model, input_dim, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label == 1 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

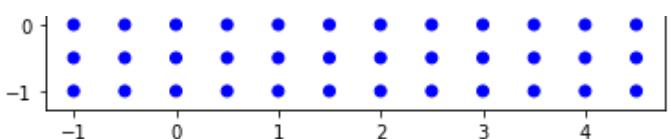
test_after = predict_linear_model_classif(p_model, input_dim, [1, 1])

print("Prediction after training of [1, 1], the prediction need to be equal to 1. \nPrediction:", test_after)

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])

destroy_linear_model(model)
```





Prediction after training of [1, 1], the prediction need to be equal to 1.  
Prediction: -1.0

## MLP (simple)

In [57]:

```
input_dim = [len(X[0]), 1]

p_model, _ = create_mlp_model(input_dim)
test_before = predict_mlp_model_classification(p_model, [1, 1])

print("Before training:", test_before)

train_classification_stochastic_gradient_backpropagation_mlp_model(p_model, X, Y)

test_after = predict_mlp_model_classification(p_model, [1, 1])

print("After training:", test_after)
```

Before training: [-0.9950548]

After training: [-0.9950548]

## MLP (advanced)

In [58]:

```
model, _ = create_mlp_model(input_dim)
test_dataset = [[x1, x2] for x1 in np.arange(-1, 5, 0.5) for x2 in np.arange(-1, 5, 0.5)]
colors = ["blue" if output >= 0 else "red" for output in Y]

predicted_outputs = [predict_mlp_model_classification(model, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label[0] >= 0 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])

train_classification_stochastic_gradient_backpropagation_mlp_model(model, X, Y)

predicted_outputs = [predict_mlp_model_classification(model, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label[0] >= 0 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

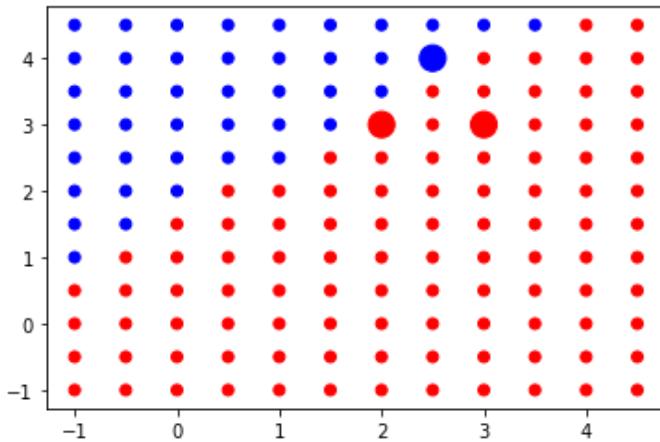
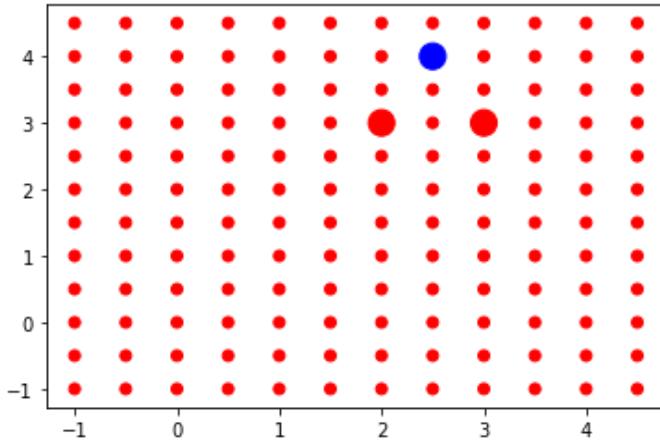
test_after = predict_mlp_model_classification(p_model, [1, 1])

print("Prediction after training of [1, 1], the result need to be equal to 1. Result:", test_after)

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
```

```
flattened_dataset_inputs.append(p[1])
```

```
destroy_mlp_model(model)
```



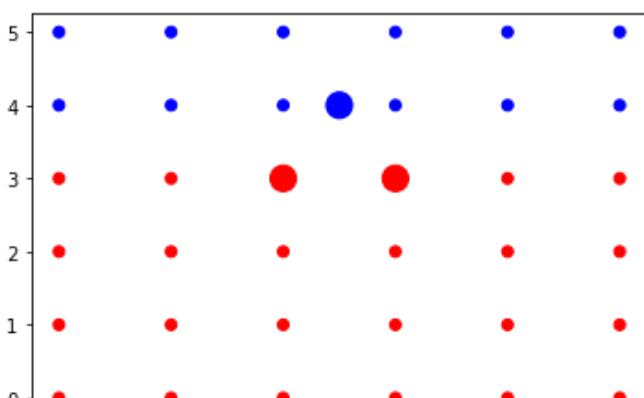
```
Prediction after training of [1, 1], the result need to be equal to 1. Result: [-0.995054  
8]
```

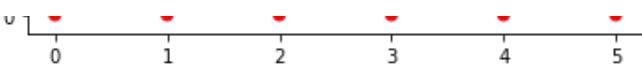
## RBFN

```
In [59]:
```

```
num_classes = 2
k = 2
input_dim = 2
expected_output = [[1,0] if label >= 0 else [0,1] for label in Y]
model = create_rbfn_model(input_dim, num_classes, k)
train_rbfn_model(model, X, expected_output)

colors = ['blue' if coord >= 0 else 'red' for coord in Y]
plot_input = [[x,y] for x in range(6) for y in range(6)]
plot_output_colors = ['blue' if predict[0] > predict[1] else 'red' for predict in [predict_rbfn(model, coord) for coord in plot_input]]
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.scatter([p[0] for p in plot_input], [p[1] for p in plot_input], c=plot_output_colors)
plt.show()
```



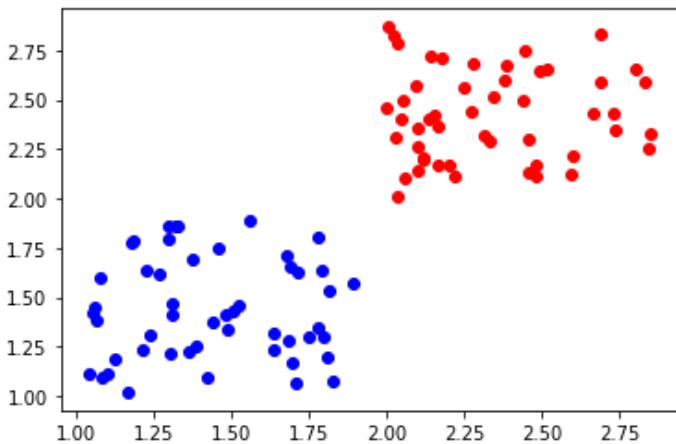


## Linear Multiple :

```
Linear Model : OK
MLP (2, 1)   : OK
```

In [60]:

```
X = np.concatenate([np.random.random((50,2)) * 0.9 + np.array([1, 1]), np.random.random((50,2)) * 0.9 + np.array([2, 2])])
Y = np.concatenate([np.ones((50, 1)), np.ones((50, 1)) * -1.0])
plt.scatter(X[0:50, 0], X[0:50, 1], color='blue')
plt.scatter(X[50:100,0], X[50:100,1], color='red')
plt.show()
plt.clf()
```



<Figure size 432x288 with 0 Axes>

## Linear model (simple)

In [61]:

```
input_dim = len(X[0])

p_model = create_linear_model(input_dim)
test_before = predict_linear_model_classif(p_model, input_dim, [1.25, 1.25])

print("Before training:", test_before)

train_linear_classification_model(p_model, input_dim, X, Y)

test_after = predict_linear_model_classif(p_model, input_dim, [1.25, 1.25])

print("After training:", test_after)
```

Before training: -1.0  
After training: -1.0

## Linear model (advanced)

In [62]:

```
model = create_linear_model(input_dim)
test_dataset = [[x1, x2] for x1 in np.arange(1, 3, 0.1) for x2 in np.arange(1, 3, 0.1)]
colors = ["blue" if output >= 0 else "red" for output in Y]

predicted_outputs = [predict_linear_model_classif(model, input_dim, p) for p in test_dataset]
```

```

predicted_outputs_colors = ['blue' if label == 1 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])
#print(flattened_dataset_inputs)

train_linear_classification_model(model, input_dim, X, Y, epochs=100000)

predicted_outputs = [predict_linear_model_classif(model, input_dim, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label == 1 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

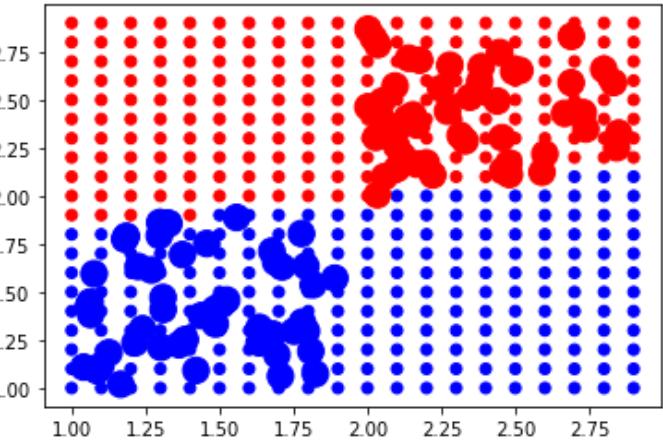
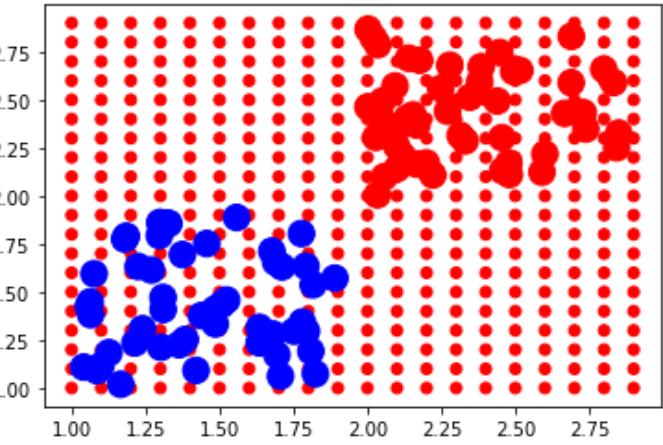
test_after = predict_linear_model_classif(p_model, input_dim, [2.5, 2.5])

print("Prediction:", test_after)

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])

destroy_linear_model(model)

```



Prediction: 1.0

## MLP (simple)

```
In [63]:
input_dim = [len(X[0]), 1]

p_model, _ = create_mlp_model(input_dim)
test_before = predict_mlp_model_classification(p_model, [1.25, 1.25])

print("Before training:", test_before)

train_classification_stochastic_gradient_backpropagation_mlp_model(p_model, X, Y)

test_after = predict_mlp_model_classification(p_model, [1.25, 1.25])

print("After training:", test_after)

Before training: [-0.84828365]
After training: [0.488269]
```

## MLP (advanced)

In [64]:

```
model, _ = create_mlp_model(input_dim)
test_dataset = [[x1, x2] for x1 in np.arange(1, 3, 0.1) for x2 in np.arange(1, 3, 0.1)]
colors = ["blue" if output >= 0 else "red" for output in Y]

predicted_outputs = [predict_mlp_model_classification(model, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label[0] >= 0 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])

train_classification_stochastic_gradient_backpropagation_mlp_model(model, X, Y, epochs=100000)

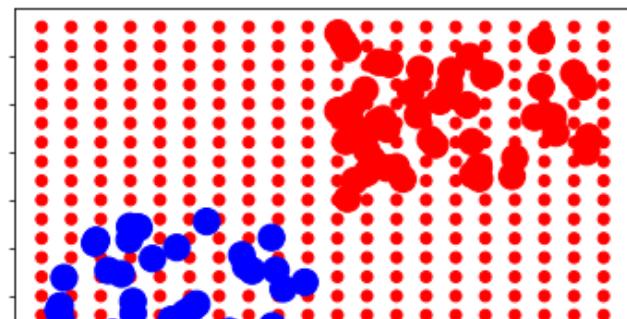
predicted_outputs = [predict_mlp_model_classification(model, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label[0] >= 0 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

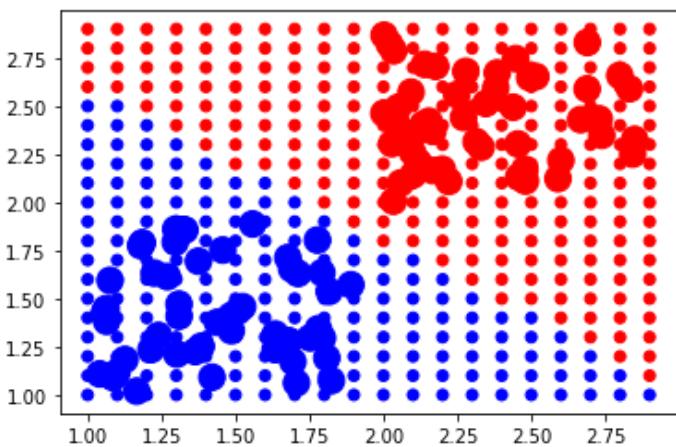
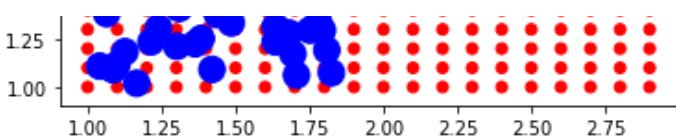
test_after = predict_mlp_model_classification(p_model, [1, 1])

print("Prediction:", test_after)

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])

destroy_mlp_model(model)
```





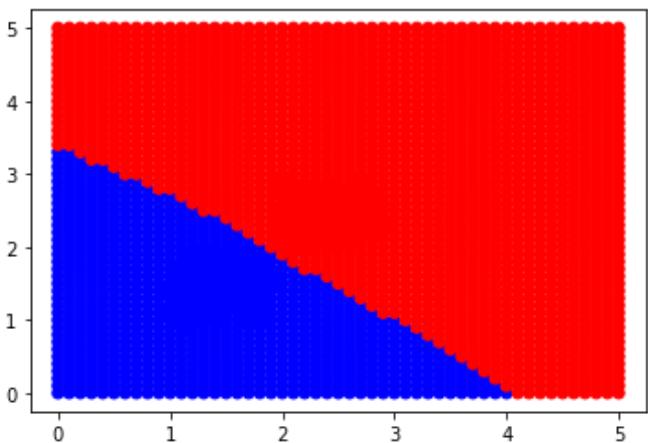
Prediction: [0.6327855]

## RBFN

In [65]:

```
num_classes = 2
k = 20
input_dim = 2
expected_output = [[1,0] if label >= 0 else [0,1] for label in Y]
model = create_rbfm_model(input_dim, num_classes, k)
train_rbfm_model(model, X, expected_output)

colors = ['blue' if coord >= 0 else 'red' for coord in Y]
plot_input = [[x/10,y/10] for x in range(51) for y in range(51)]
plot_output_colors = ['blue' if predict[0] > predict[1] else 'red' for predict in [predict_rbfm(model, coord) for coord in plot_input]]
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.scatter([p[0] for p in plot_input], [p[1] for p in plot_input], c=plot_output_colors)
plt.show()
```



## XOR :

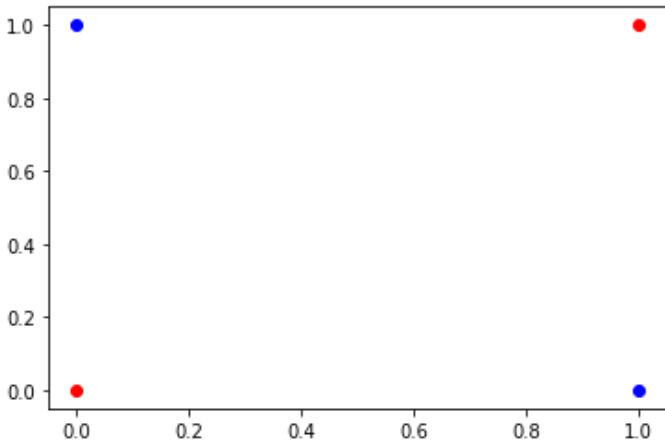
Linear Model	:	KO
MLP (2, 2, 1)	:	OK

In [66]:

```
X = np.array([[1, 0], [0, 1], [0, 0], [1, 1]])
Y = np.array([1, 1, -1, -1])
```

In [67]:

```
plt.scatter(X[0:2, 0], X[0:2, 1], color='blue')
plt.scatter(X[2:4,0], X[2:4,1], color='red')
plt.show()
plt.clf()
```



<Figure size 432x288 with 0 Axes>

## Linear model (simple)

On isole les 4 points par 2 sorties de modèle linéaire, chacune entraînée par 2 valeurs d'une classe et 1 valeur de l'autre classe. On simule en réalité le travail d'un MLP (2,2,1)

In [68]:

```
input_dim = len(X[0])

p_model = create_linear_model(input_dim)
test_before = predict_linear_model_classif(p_model, input_dim, [0, 0])

print("Before training:", test_before)

train_linear_classification_model(p_model, input_dim, X, Y)

test_after = predict_linear_model_classif(p_model, input_dim, [0, 0])

print("After training:", test_after)
```

Before training: 1.0  
After training: 1.0

## Linear model (advanced)

In [69]:

```
X = np.array([[0, 0], [1, 0], [0, 1], [1, 1]])
Y = np.array([-1, 1, 1, -1])
```

In [70]:

```
input_dim = len(X[0])

model_top = create_linear_model(input_dim)
model_bottom = create_linear_model(input_dim)

test_dataset = [[x1, x2] for x1 in np.arange(-1, 2, 0.1) for x2 in np.arange(-1, 2, 0.1)]
colors = ["blue" if output >= 0 else "red" for output in Y]

predicted_outputs_top = [predict_linear_model_classif(model_top, input_dim, p) for p in
```

```

n test_dataset]
predicted_outputs_bottom = [predict_linear_model_classif(model_bottom, input_dim, p) for p in test_dataset]

predicted_outputs_colors = ['blue' if top == 1 and bottom == 1 else 'red' for (top, bottom) in zip(predicted_outputs_top, predicted_outputs_bottom)]

plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

train_linear_classification_model(model_top, input_dim, X[:-1], Y[:-1], alpha=0.01, epochs=100000)
train_linear_classification_model(model_bottom, input_dim, X[1:], Y[1:], alpha=0.01, epochs=100000)

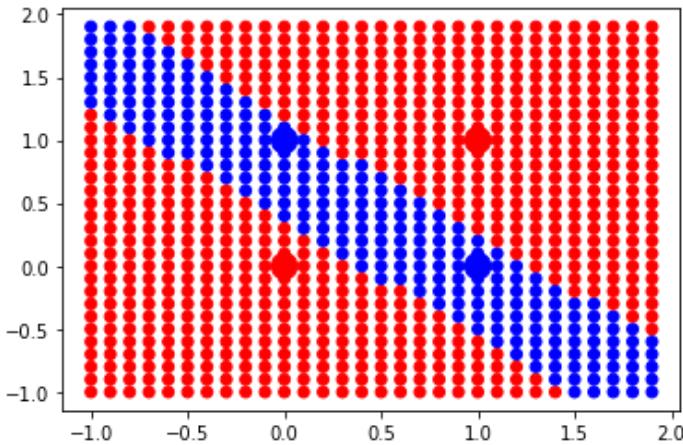
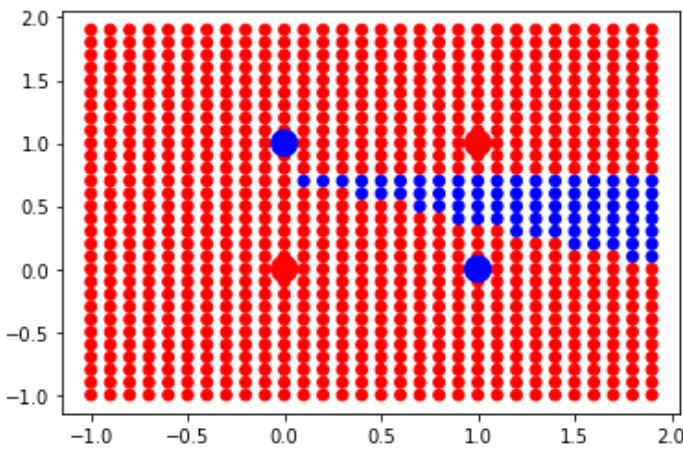
predicted_outputs_top = [predict_linear_model_classif(model_top, input_dim, p) for p in test_dataset]
predicted_outputs_bottom = [predict_linear_model_classif(model_bottom, input_dim, p) for p in test_dataset]

predicted_outputs_colors = ['blue' if top == 1 and bottom == 1 else 'red' for (top, bottom) in zip(predicted_outputs_top, predicted_outputs_bottom)]

plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

destroy_linear_model(model_top)
destroy_linear_model(model_bottom)

```



## MLP (simple)

In [71]:

```
input_dim = [len(X[0]), len(X[0]), 1]
```

```

p_model, _ = create_mlp_model(input_dim)
test_before = predict_mlp_model_classification(p_model, [1, 1])

print("Before training:", test_before)

train_classification_stochastic_gradient_backpropagation_mlp_model(p_model, X, Y)

test_after = predict_mlp_model_classification(p_model, [1, 1])

print("After training:", test_after)

```

Before training: [0.6203795]  
After training: [-0.10921684]

## MLP (advanced)

In [72]:

```

model, _ = create_mlp_model(input_dim)
test_dataset = [[x1, x2] for x1 in np.arange(-1, 2, 0.1) for x2 in np.arange(-1, 2, 0.1)]
colors = ["blue" if output >= 0 else "red" for output in Y]

predicted_outputs = [predict_mlp_model_classification(model, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label[0] >= 0 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])

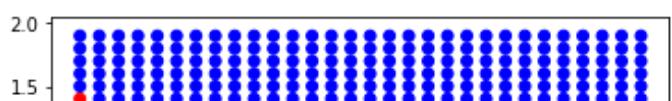
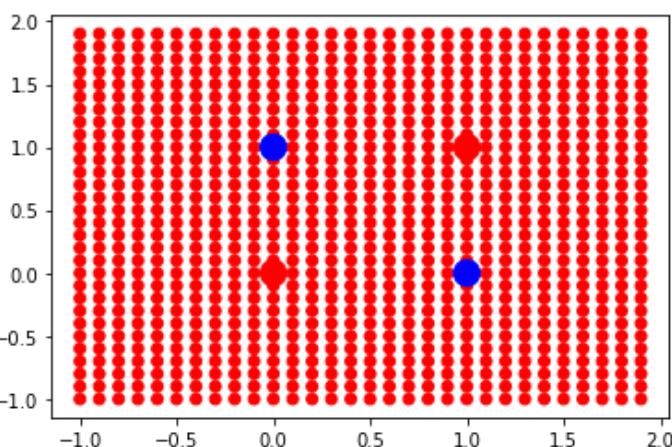
train_classification_stochastic_gradient_backpropagation_mlp_model(model, X, Y, epochs=100000)

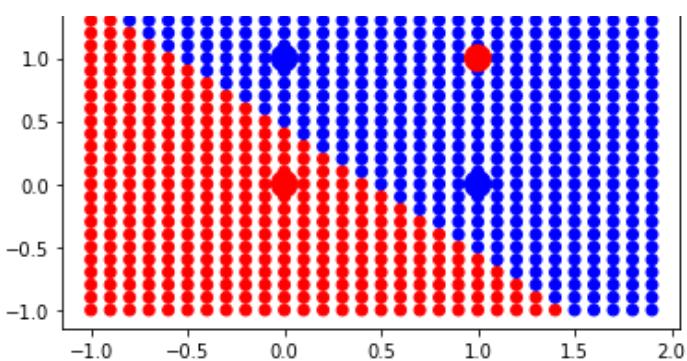
predicted_outputs = [predict_mlp_model_classification(model, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label[0] >= 0 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

test_after = predict_mlp_model_classification(p_model, [1, 1])

destroy_mlp_model(model)

```



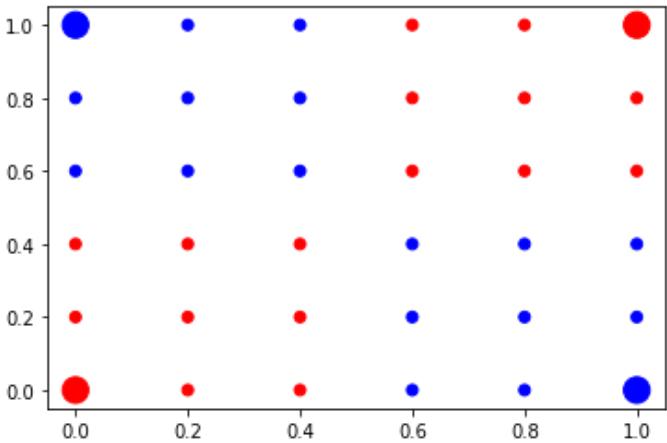


## RBFN

In [73]:

```
num_classes = 2
k = 4
input_dim = 2
expected_output = [[1,0] if label >= 0 else [0,1] for label in Y]
modif_input = [[coord[0],coord[1]] for coord in X]
model = create_rbfm_model(input_dim, num_classes, k)
train_rbfm_model(model, X, expected_output, naif=True)

colors = ['blue' if coord >= 0 else 'red' for coord in Y]
plot_input = [[x/5,y/5] for x in range(6) for y in range(6)]
plot_output_colors = ['blue' if predict[0] > predict[1] else 'red' for predict in [predict_rbfm(model, coord) for coord in plot_input]]
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.scatter([p[0] for p in plot_input], [p[1] for p in plot_input], c=plot_output_colors)
plt.show()
```



## Cross :

Linear Model	:	KO
MLP (2, 4, 1)	:	OK

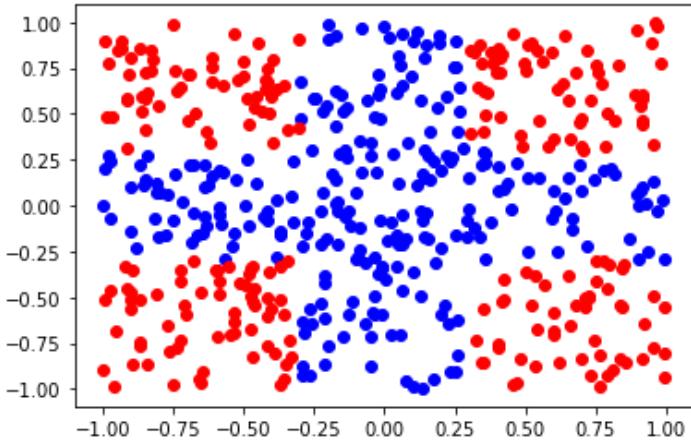
In [74]:

```
X = np.random.random((500, 2)) * 2.0 - 1.0
Y = np.array([1 if abs(p[0]) <= 0.3 or abs(p[1]) <= 0.3 else -1 for p in X])
```

In [75]:

```
plt.scatter(np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]] == 1, enumerate(X)))))[ :,0], np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]] == 1, enumerate(X)))))[ :,1], color='blue')
plt.scatter(np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]] == -1, enumerate(X)))))[ :,0], np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]] == -1, enumerate(X)))))[ :,1], color='red')
```

```
plt.show()  
plt.clf()
```



```
<Figure size 432x288 with 0 Axes>
```

## Linear model (simple)

On génère plusieurs modèles linéaires qui vont découper la croix en différentes parties pour isoler 1 à 1 les blocs rouges

```
In [76]:
```

```
input_dim = len(X[0])  
  
p_model = create_linear_model(input_dim)  
test_before = predict_linear_model_classif(p_model, input_dim, [0, 0])  
  
print("Before training:", test_before)  
  
train_linear_classification_model(p_model, input_dim, X, Y)  
  
test_after = predict_linear_model_classif(p_model, input_dim, [0, 0])  
  
print("After training:", test_after)
```

```
Before training: -1.0  
After training: -1.0
```

## Linear model (advanced)

```
In [77]:
```

```
input_dim = len(X[0])  
  
model_vl = create_linear_model(input_dim)  
model_vr = create_linear_model(input_dim)  
model_ht = create_linear_model(input_dim)  
model_hb = create_linear_model(input_dim)  
  
test_dataset = [[x1, x2] for x1 in np.arange(-2, 2, 0.1) for x2 in np.arange(-2, 2, 0.1)]  
colors = ["blue" if output >= 0 else "red" for output in Y]  
  
predicted_outputs_vl = [predict_linear_model_classif(model_vl, input_dim, p) for p in test_dataset]  
predicted_outputs_vr = [predict_linear_model_classif(model_vr, input_dim, p) for p in test_dataset]  
predicted_outputs_ht = [predict_linear_model_classif(model_ht, input_dim, p) for p in test_dataset]  
predicted_outputs_hb = [predict_linear_model_classif(model_hb, input_dim, p) for p in test_dataset]
```

```

predicted_outputs_colors = ['blue' if (vl == 1 and vr == 1) or (ht == 1 and hb == 1) else 'red' for (vl, vr, ht, hb) in zip(predicted_outputs_vl, predicted_outputs_vr, predicted_outputs_ht, predicted_outputs_hb)]

plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

```

```

ht = np.array([[x[0],x[1],n] for n,x in enumerate(X) if (x[1] > -0.3 and (x[0] > 0.3 or x[0] < -0.3) ) ])
c = ["blue" if Y[int(n)] == 1 else "red" for n in ht[:, 2]]
plt.scatter(ht[:, 0], ht[:, 1], c=c)
plt.axis([-1, 1, -1, 1])
plt.title("Horizontal top")
plt.show()

```

```

vr = np.array([[x[0],x[1], int(n)] for n,x in enumerate(X) if x[0] > -0.3 and (x[1] > 0.3 or x[1] < -0.3) ])
c = ["blue" if Y[int(n)] == 1 else "red" for n in vr[:, 2]]
plt.scatter(vr[:, 0], vr[:, 1], c=c)
plt.axis([-1, 1, -1, 1])
plt.title("Vertical right")
plt.show()

```

```

vl = np.array([[x[0],x[1],int(n)] for n,x in enumerate(X) if (x[0] < 0.3 and (x[1] < -0.3 or x[1] > 0.3))])
c = ["blue" if Y[int(n)] == 1 else "red" for n in vl[:, 2]]
plt.scatter(vl[:, 0], vl[:, 1], c=c)
plt.axis([-1, 1, -1, 1])
plt.title("Vertical left")
plt.show()

```

```

hb = np.array([[x[0],x[1],int(n)] for n,x in enumerate(X) if (x[1] < 0.3 and (x[0] < -0.3 or x[0] > 0.3))])
c = ["blue" if Y[int(n)] == 1 else "red" for n in hb[:, 2]]
plt.scatter(hb[:, 0], hb[:, 1], c=c)
plt.axis([-1, 1, -1, 1])
plt.title("Horizontal bottom")
plt.show()

```

```

# On était bloqué la wola
train_linear_classification_model(model_vl, input_dim, vl[:, :-1], Y[vl[:, -1].astype(int)], epochs=100000)
train_linear_classification_model(model_vr, input_dim, vr[:, :-1], Y[vr[:, -1].astype(int)], epochs=100000)
train_linear_classification_model(model_ht, input_dim, ht[:, :-1], Y[ht[:, -1].astype(int)], epochs=100000)
train_linear_classification_model(model_hb, input_dim, hb[:, :-1], Y[hb[:, -1].astype(int)], epochs=100000)

```

```

predicted_outputs_vl = [predict_linear_model_classif(model_vl, input_dim, p) for p in test_dataset]
predicted_outputs_vr = [predict_linear_model_classif(model_vr, input_dim, p) for p in test_dataset]
predicted_outputs_ht = [predict_linear_model_classif(model_ht, input_dim, p) for p in test_dataset]
predicted_outputs_hb = [predict_linear_model_classif(model_hb, input_dim, p) for p in test_dataset]

```

```

predicted_outputs_colors = ['blue' if (vl == 1 and vr == 1) or (ht == 1 and hb == 1) else 'red' for (vl, vr, ht, hb) in zip(predicted_outputs_vl, predicted_outputs_vr, predicted_outputs_ht, predicted_outputs_hb)]

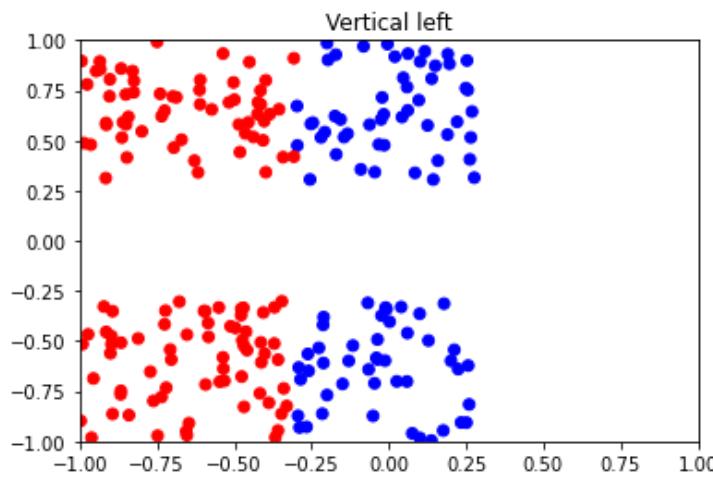
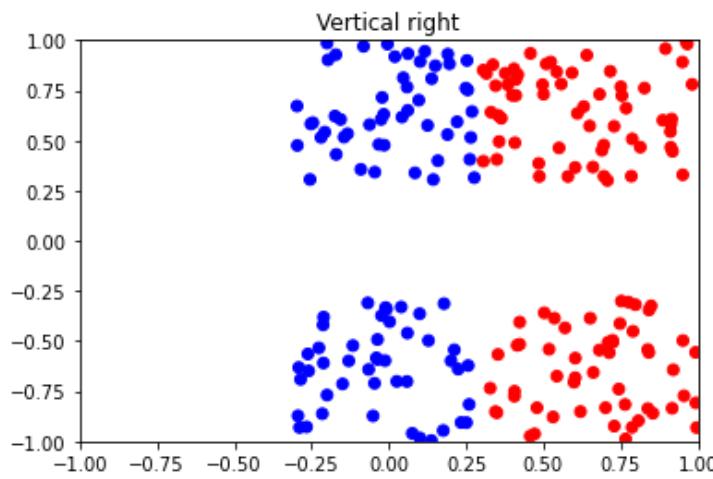
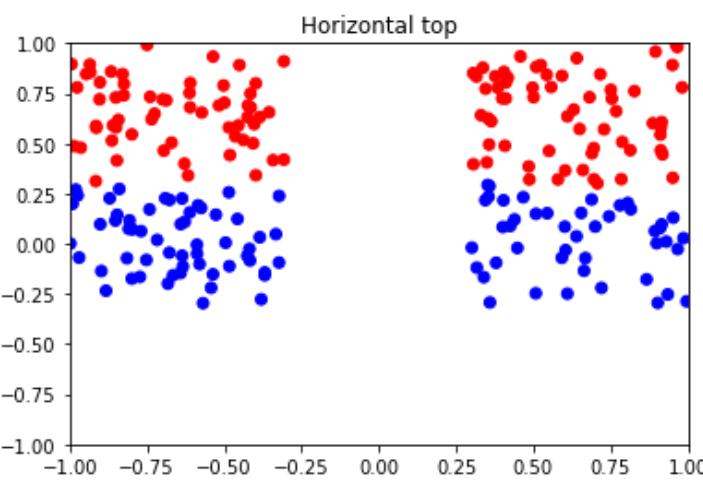
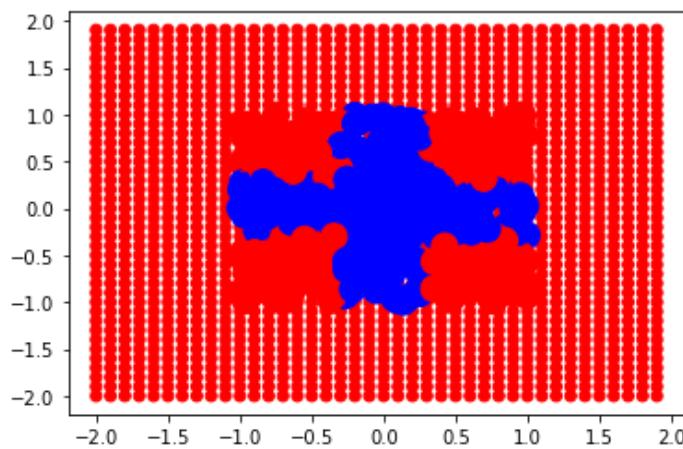
```

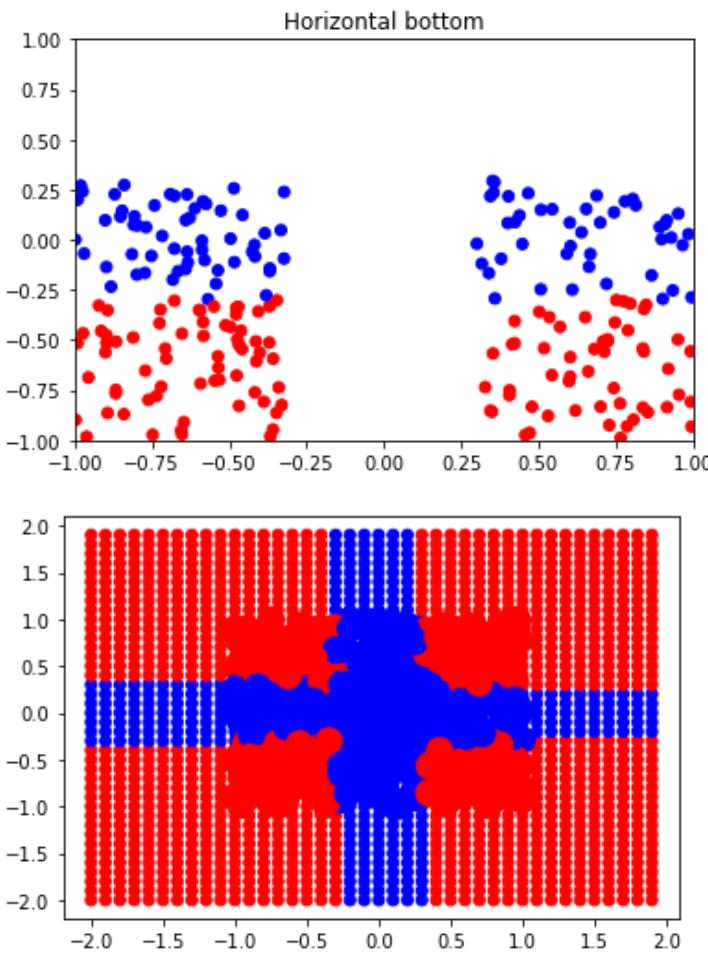
```

plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

```

```
destroy_linear_model(model_vl)
destroy_linear_model(model_vr)
destroy_linear_model(model_ht)
destroy_linear_model(model_hb)
```





## MLP (simple)

In [78]:

```
input_dim = [len(X[0]), len(X[0])*2, 1]

p_model, _ = create_mlp_model(input_dim)
test_before = predict_mlp_model_classification(p_model, [0, 0])

print("Before training:", test_before)

train_classification_stochastic_gradient_backpropagation_mlp_model(p_model, X, Y)

test_after = predict_mlp_model_classification(p_model, [0, 0])

print("After training:", test_after)
```

Before training: [-0.23398942]  
After training: [0.05618626]

## MLP (advanced)

In [79]:

```
model, _ = create_mlp_model(input_dim)
test_dataset = [[x1, x2] for x1 in np.arange(-2, 2, 0.1) for x2 in np.arange(-2, 2, 0.1)]
colors = ["blue" if output >= 0 else "red" for output in Y]

predicted_outputs = [predict_mlp_model_classification(model, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label[0] >= 0 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()
```

```

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])

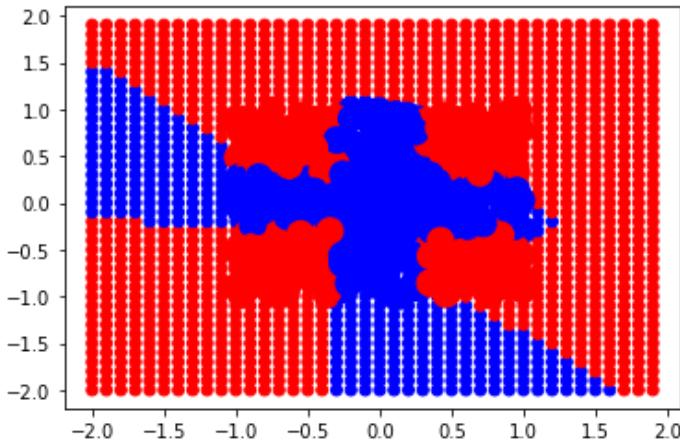
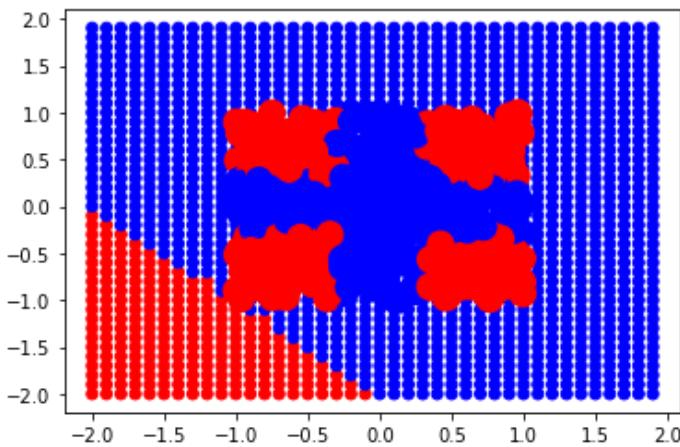
train_classification_stochastic_gradient_backpropagation_mlp_model(model, X, Y, alpha=0.01, epochs=200000)

predicted_outputs = [predict_mlp_model_classification(model, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label[0] >= 0 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

test_after = predict_mlp_model_classification(p_model, [1, 1])

destroy_mlp_model(model)

```



## RBFN

In [80]:

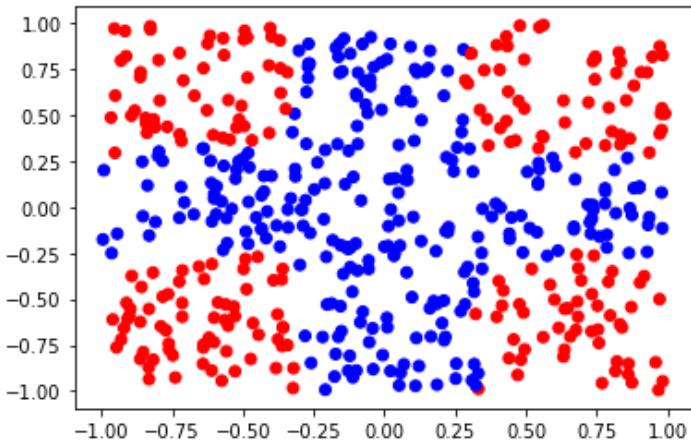
```

num_classes = 2
k = 30
input_dim = 2
expected_output = [[1, 0] if coord >= 0 else [0, 1] for coord in Y]
modif_input = [[coord[0], coord[1]] for coord in X]
model = create_rbfm_model(input_dim, num_classes, k)
train_rbfm_model(model, modif_input, expected_output)

colors = ['blue' if coord >= 0 else 'red' for coord in Y]
plot_input = np.random.random((500, 2)) * 2.0 - 1.0
plot_output_colors = ['blue' if predict[0] > predict[1] else 'red' for predict in [predict_rbfm(model, coord) for coord in plot_input]]
# plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.scatter([p[0] for p in plot_input], [p[1] for p in plot_input], c=plot_output_colors)

```

```
)  
plt.show()
```



## Multi Linear 3 classes :

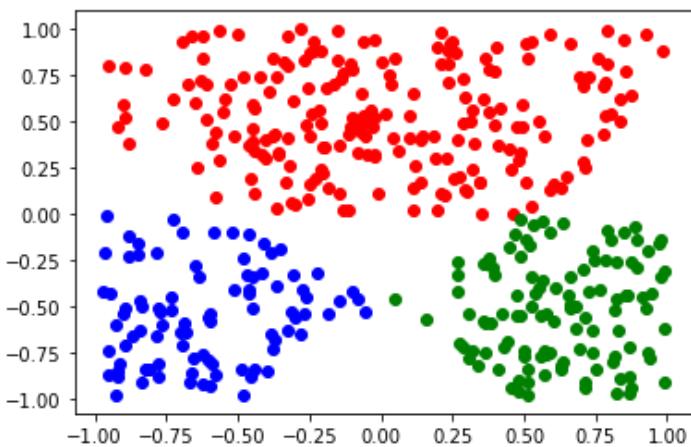
```
Linear Model x3 : OK  
MLP (2, 3) : OK
```

In [81]:

```
X = np.random.random((500, 2)) * 2.0 - 1.0  
Y = np.array([[1, 0, 0] if -p[0] - p[1] - 0.5 > 0 and p[1] < 0 and p[0] - p[1] - 0.5 < 0 else # bleu  
             [0, 1, 0] if -p[0] - p[1] - 0.5 < 0 and p[1] > 0 and p[0] - p[1] - 0.5 < 0 else # rouge  
             [0, 0, 1] if -p[0] - p[1] - 0.5 < 0 and p[1] < 0 and p[0] - p[1] - 0.5 > 0 else # vert  
             [0, 0, 0] for p in X])
```

In [82]:

```
plt.scatter(np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][0] == 1, enumerate(X))))[:,0], np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][0] == 1, enumerate(X))))[:,1], color='blue')  
plt.scatter(np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][1] == 1, enumerate(X))))[:,0], np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][1] == 1, enumerate(X))))[:,1], color='red')  
plt.scatter(np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][2] == 1, enumerate(X))))[:,0], np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][2] == 1, enumerate(X))))[:,1], color='green')  
plt.show()  
plt.clf()
```



<Figure size 432x288 with 0 Axes>

## Linear model (simple)

In [83]:

```
input_dim = 2

p_model = create_linear_model(input_dim)
test_before = predict_linear_model_classif(p_model, input_dim, [-0.75, -0.50])

print("Before training:", test_before)

flattened_Y = Y[-1].flatten()

train_linear_classification_model(p_model, input_dim, X, flattened_Y)

test_after = predict_linear_model_classif(p_model, input_dim, [-0.75, -0.50])

print("After training:", test_after)
```

Before training: 1.0

After training: -1.0

## Linear model (advanced)

In [84]:

```
input_dim = len(X[0])

model_dr = create_linear_model(input_dim)
model_dl = create_linear_model(input_dim)
model_h = create_linear_model(input_dim)

test_dataset = [[x1, x2] for x1 in np.arange(-1.5, 1.5, 0.1) for x2 in np.arange(-1.5, 1.5, 0.1)]
colors = ["blue" if output[0] == 1 else ("red" if output[1] == 1 else ("green" if output[2] == 1 else "black")) for output in Y]

dr = np.array([1 if y[2] == 1 else -1 for y in Y])
h = np.array([1 if y[1] == 1 else -1 for y in Y])
dl = np.array([1 if y[0] == 1 else -1 for y in Y])

predicted_outputs_dr = [predict_linear_model_classif(model_dr, input_dim, p) for p in test_dataset]
predicted_outputs_dl = [predict_linear_model_classif(model_dl, input_dim, p) for p in test_dataset]
predicted_outputs_h = [predict_linear_model_classif(model_h, input_dim, p) for p in test_dataset]

predicted_outputs_colors = ['green' if (dr == 1 and dl == -1 and h == -1) else ("blue" if (dl == 1 and dr == -1 and h == -1) else ("red" if (h == 1 and dr == -1 and dl == -1) else "black")) for (dr, dl, h) in zip(predicted_outputs_dr, predicted_outputs_dl, predicted_outputs_h)]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=40)
plt.show()

flattened_dataset_outputs = []
for p in Y:
    flattened_dataset_outputs.append(p[0])
    flattened_dataset_outputs.append(p[1])
    flattened_dataset_outputs.append(p[2])

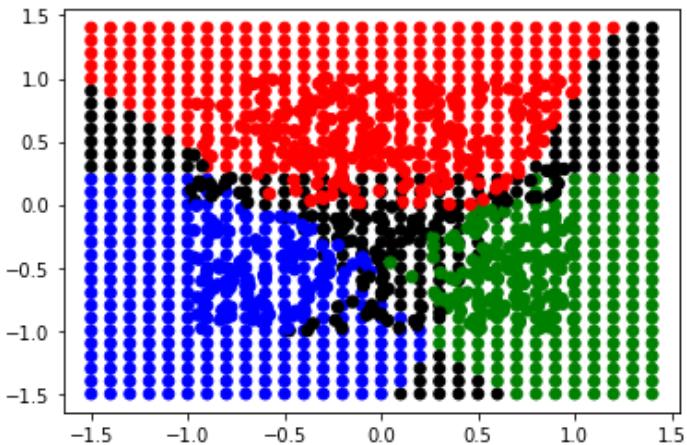
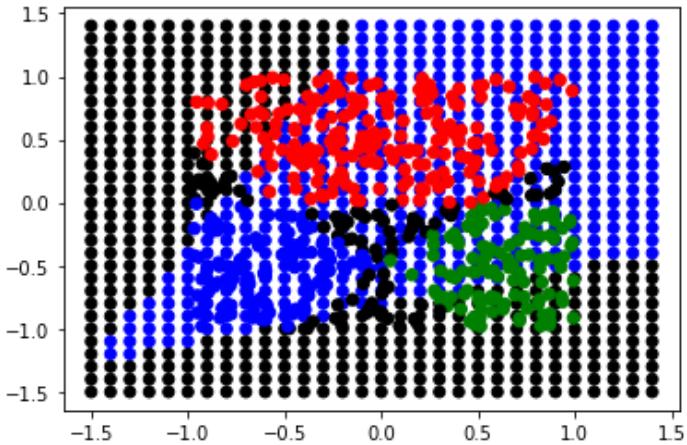
train_linear_classification_model(model_dl, input_dim, X, dl, alpha=0.01, epochs=100000)
train_linear_classification_model(model_dr, input_dim, X, dr, alpha=0.01, epochs=100000)
train_linear_classification_model(model_h, input_dim, X, h, alpha=0.01, epochs=100000)
```

```

predicted_outputs_dr = [predict_linear_model_classif(model_dr, input_dim, p) for p in test_dataset]
predicted_outputs_dl = [predict_linear_model_classif(model_dl, input_dim, p) for p in test_dataset]
predicted_outputs_h = [predict_linear_model_classif(model_h, input_dim, p) for p in test_dataset]

predicted_outputs_colors = ['green' if (dr == 1 and dl == -1 and h == -1) else ("blue" if (dl == 1 and dr == -1 and h == -1) else ("red" if (h == 1 and dr == -1 and dl == -1) else "black")) for (dr, dl, h) in zip(predicted_outputs_dr, predicted_outputs_dl, predicted_outputs_h)]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=40)
plt.show()

```



## MLP (simple)

In [85]:

```

input_dim = [len(X[0]), 3]

p_model, len_output_layer = create_mlp_model(input_dim)
test_before = predict_mlp_model_classification(p_model, np.array([-0.75, -0.50]), len_output_layer)

print("Before training:", test_before)

train_classification_stochastic_gradient_backpropagation_mlp_model(p_model, X, Y.flatten())

test_after = predict_mlp_model_classification(p_model, np.array([-0.75, -0.50]), len_output_layer)

print("After training:", test_after)

```

Before training: [0.46211717, 0.63514894, 0.63514894]  
After training: [0.5670651, 0.25488898, 0.10530939]

## MLP (advanced)

In [86]:

```
input_dim = [len(X[0]), 3]

model, len_output_layer = create_mlp_model(input_dim)
test_dataset = [[x1, x2] for x1 in np.arange(-1.5, 1.6, 0.2) for x2 in np.arange(-1.5, 1.6, 0.2)]
colors = ["blue" if np.argmax(output) == 0 else ("red" if np.argmax(output) == 1 else "green") for output in Y]

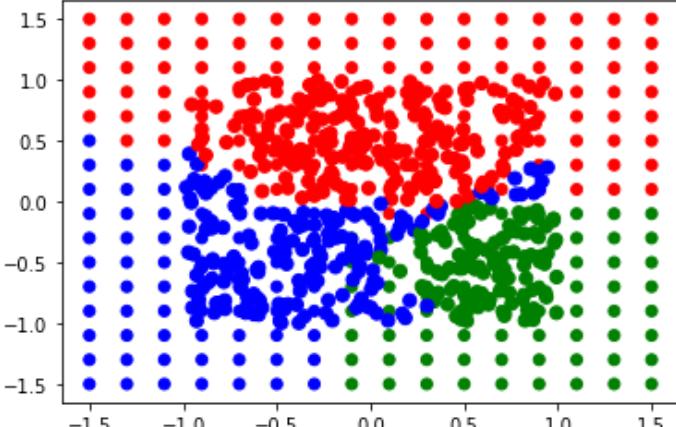
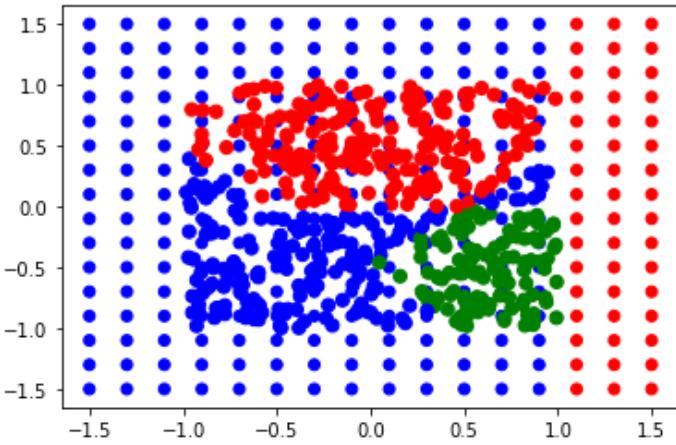
predicted_outputs = [predict_mlp_model_classification(model, p, len_output_layer) for p in test_dataset]
predicted_outputs_colors = ["blue" if np.argmax(output) == 0 else ("red" if np.argmax(output) == 1 else "green") for output in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=50)
plt.show()

train_classification_stochastic_gradient_backpropagation_mlp_model(model, X, Y.flatten())

predicted_outputs = [predict_mlp_model_classification(model, p, len_output_layer) for p in test_dataset]
predicted_outputs_colors = ["blue" if np.argmax(output) == 0 else ("red" if np.argmax(output) == 1 else "green") for output in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=50)
plt.show()

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])

destroy_mlp_model(model)
```

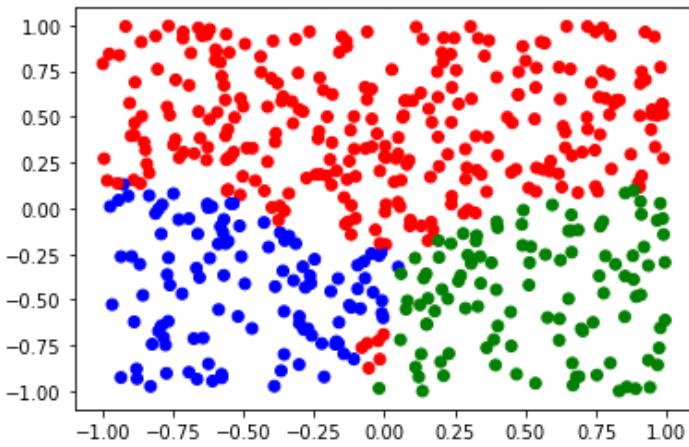


## RBF

In [87]:

```
num_classes = 3
k = 30
input_dim = 2
expected_output = Y
modif_input = [[coord[0], coord[1]] for coord in X]
model = create_rbfn_model(input_dim, num_classes, k)
train_rbfn_model(model, modif_input, expected_output)

#colors = ['blue' if coord >= 0 else 'red' for coord in Y]
plot_input = np.random.random((500, 2)) * 2.0 - 1.0
plot_output_colors = ['blue' if predict[0] > predict[1] and predict[0] > predict[2] else ('red' if predict[1] > predict[2] else 'green') for predict in [predict_rbfm(model, coord, 3) for coord in plot_input]]
#plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.scatter([p[0] for p in plot_input], [p[1] for p in plot_input], c=plot_output_colors)
plt.show()
```



## Multi Cross :

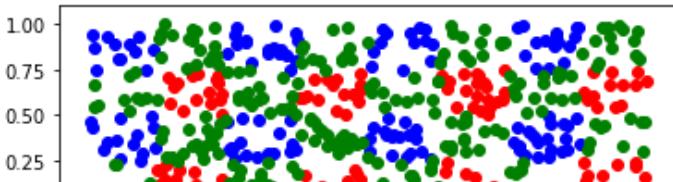
```
Linear Model x3 : KO
MLP (2, ?, ?, 3): OK
```

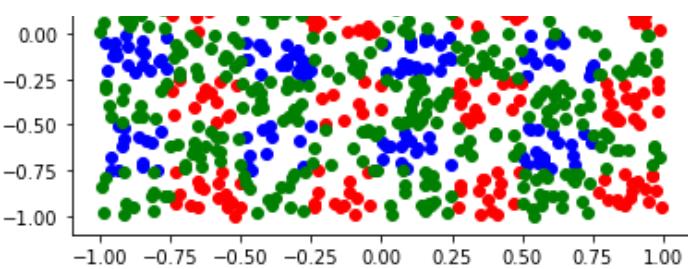
In [88]:

```
X = np.random.random((1000, 2)) * 2.0 - 1.0
Y = np.array([[1, 0, 0] if abs(p[0] % 0.5) <= 0.25 and abs(p[1] % 0.5) > 0.25 else [0, 1, 0] if abs(p[0] % 0.5) > 0.25 and abs(p[1] % 0.5) <= 0.25 else [0, 0, 1] for p in X])
```

In [89]:

```
plt.scatter(np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][0] == 1, enumerate(X))))[:,0], np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][0] == 1, enumerate(X))))[:,1], color='blue')
plt.scatter(np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][1] == 1, enumerate(X))))[:,0], np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][1] == 1, enumerate(X))))[:,1], color='red')
plt.scatter(np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][2] == 1, enumerate(X))))[:,0], np.array(list(map(lambda elt : elt[1], filter(lambda c: Y[c[0]][2] == 1, enumerate(X))))[:,1], color='green')
plt.show()
plt.clf()
```





<Figure size 432x288 with 0 Axes>

## Linear model (simple)

Impossible de classer tous ces groupes avec uniquement 3 “droites”, on a alors transformé les entrées de notre modèle pour n’avoir qu’une instance des quatre groupes qui se répétaient à l’infini sur l’ensemble d’origine.

In [90]:

```
"""
input_dim = len(X[0])

p_model = create_linear_model(input_dim)
test_before = predict_linear_model_classif(p_model, input_dim, [2, 2])

print("Before training:", test_before)

flattened_X = X.flatten()

train_linear_classification_model(p_model, input_dim, flattened_X, Y)

test_after = predict_linear_model_classif(p_model, input_dim, [2, 2])

print("After training:", test_after)
"""
```

Out[90]:

```
'\ninput_dim = len(X[0])\n\np_model = create_linear_model(input_dim)\ntest_before = predict_linear_model_classif(p_model, input_dim, [2, 2])\n\nprint("Before training:", test_before)\n\nflattened_X = X.flatten()\n\ntrain_linear_classification_model(p_model, input_dim, flattened_X, Y)\n\ntest_after = predict_linear_model_classif(p_model, input_dim, [2, 2])\n\nprint("After training:", test_after)\n'
```

## Linear model (advanced)

Ne fonctionne pas

In [91]:

```
"""
model = create_linear_model(input_dim)
test_dataset = [[x1, x2] for x1 in range(1, 2) for x2 in range(1, 2)]
colors = ["blue" if output >= 0 else "red" for output in Y]

predicted_outputs = [predict_linear_model_classif(model, input_dim, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label == 1 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])

train_linear_classification_model(model, input_dim, flattened_dataset_inputs, Y)
```

```

predicted_outputs = [predict_linear_model_classif(model, input_dim, p) for p in test_dataset]
predicted_outputs_colors = ['blue' if label == 1 else 'red' for label in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.show()

test_after = predict_linear_model_classif(p_model, input_dim, [1.25, 1.25])
print("Prediction:", test_after)

flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])
    flattened_dataset_inputs.append(p[1])

destroy_linear_model(model)
"""

```

Out[91]:

```

'model = create_linear_model(input_dim)\ntest_dataset = [[x1, x2] for x1 in range(1, 2)
for x2 in range(1, 2)]\ncolors = ["blue" if output >= 0 else "red" for output in Y]\n\npredicted_outputs = [predict_linear_model_classif(model, input_dim, p) for p in test_dataset]\npredicted_outputs_colors = ['blue' if label == 1 else 'red' for label in predicted_outputs]\nplt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)\nplt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)\nplt.show()\n\nflattened_dataset_inputs = []\nfor p in X:\n    flattened_dataset_inputs.append(p[0])\n    flattened_dataset_inputs.append(p[1])\n\ntrain_linear_classification_model(model, input_dim, flattened_dataset_inputs, Y)\n\npredicted_outputs = [predict_linear_model_classif(model, input_dim, p) for p in test_dataset]\npredicted_outputs_colors = ['blue' if label == 1 else 'red' for label in predicted_outputs]\nplt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)\nplt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)\nplt.show()\n\ntest_after = predict_linear_model_classif(p_model, input_dim, [1.25, 1.25])\n\nprint("Prediction:", test_after)\n\nflattened_dataset_inputs = []\nfor p in X:\n    flattened_dataset_inputs.append(p[0])\n    flattened_dataset_inputs.append(p[1])\n\ndestroy_linear_model(model)\n'

```

## MLP (simple)

**Nous avons cherché à augmenter le nombre de neurones par couche jusqu'à ne plus observer qu'une stagnation des performances de traitement contre un temps de traitement croissant. On a considéré qu'une forme de (2,64,64,3) était un bon compromis performances/rapidité. Il a cependant fallu travailler sur le nombre d'époques ainsi que sur la correction apportée à chaque époque pour favoriser l'entraînement**

In [92]:

```

input_dim = [len(X[0]), 2, 2, 3]

p_model, len_output_layer = create_mlp_model(input_dim)
test_before = predict_mlp_model_classification(p_model, [2.5, 2.5], len_output_layer)

print("Before training:", test_before)

train_classification_stochastic_gradient_backpropagation_mlp_model(p_model, X, Y.flatten(),
epochs=10000)

test_after = predict_mlp_model_classification(p_model, [2.5, 2.5], len_output_layer)

print("After training:", test_after)

```

Before training: [-0.7615942, 0.22981052, -0.38679492]  
After training: [0.2643014, 0.21673441, 0.45577648]

## MLP (advanced)

In [93]:

```
input_dim = [len(X[0]),26,26,3]

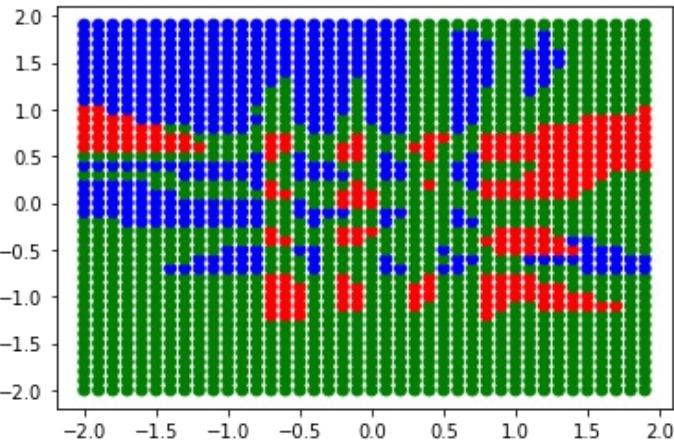
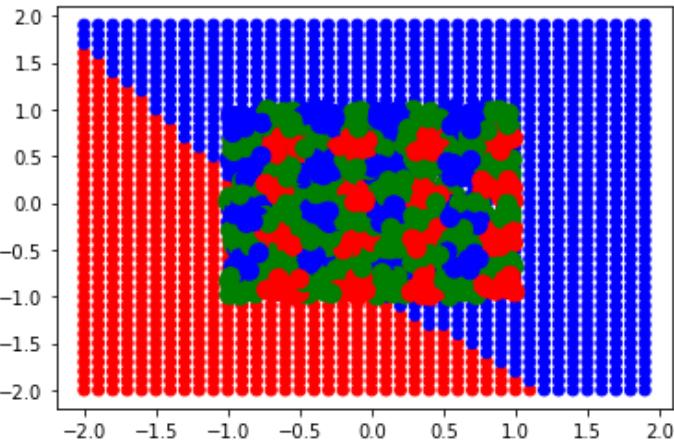
model, len_output_layer = create_mlp_model(input_dim)
test_dataset = [[x1, x2] for x1 in np.arange(-2, 2, 0.1) for x2 in np.arange(-2, 2, 0.1)]
colors = ["blue" if np.argmax(output) == 0 else ("red" if np.argmax(output) == 1 else "green") for output in Y]

predicted_outputs = [predict_mlp_model_classification(model, p, len_output_layer) for p in test_dataset]
predicted_outputs_colors = ["blue" if np.argmax(output) == 0 else ("red" if np.argmax(output) == 1 else "green") for output in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=100)
plt.show()

train_classification_stochastic_gradient_backpropagation_mlp_model(model, X, Y.flatten(), alpha=0.03, epochs=1000000)

predicted_outputs = [predict_mlp_model_classification(model, p, len_output_layer) for p in test_dataset]
predicted_outputs_colors = ["blue" if np.argmax(output) == 0 else ("red" if np.argmax(output) == 1 else "green") for output in predicted_outputs]
plt.scatter([p[0] for p in test_dataset], [p[1] for p in test_dataset], c=predicted_outputs_colors)
# plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=100)
plt.show()

destroy_mlp_model(model)
```



## RBF

In [94]:

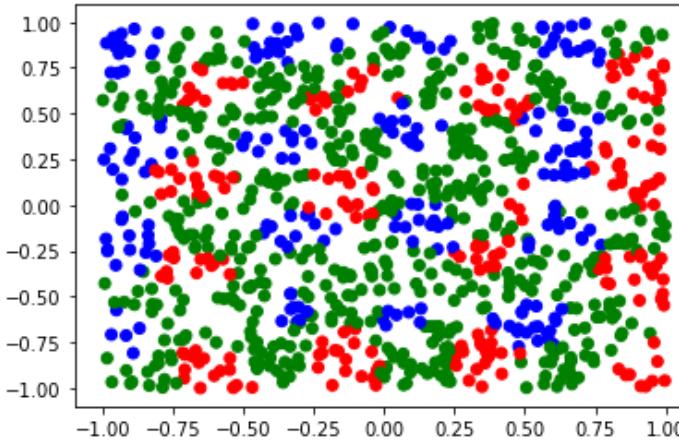
```
num_classes = 3
```

```

num_classes = 3
k = 100
input_dim = 2
expected_output = Y
modif_input = [[coord[0],coord[1]] for coord in X]
model = create_rbfn_model(input_dim, num_classes, k)
train_rbfn_model(model, modif_input, expected_output)

#colors = ['blue' if coord >= 0 else 'red' for coord in Y]
plot_input = np.random.random((1000, 2)) * 2.0 - 1.0
plot_output_colors = ['blue' if predict[0] > predict[1] and predict[0] > predict[2] else ('red' if predict[1] > predict[2] else 'green') for predict in [predict_rbfm(model, coord, 3) for coord in plot_input]]
#plt.scatter([p[0] for p in X], [p[1] for p in X], c=colors, s=200)
plt.scatter([p[0] for p in plot_input], [p[1] for p in plot_input], c=plot_output_colors)
plt.show()

```



## Régression

### Linear Simple 2D :

```

Linear Model : OK
MLP (1, 1)   : OK

```

In [95]:

```

X = np.array([
    [1],
    [2]
])
Y = np.array([
    2,
    3
])

```

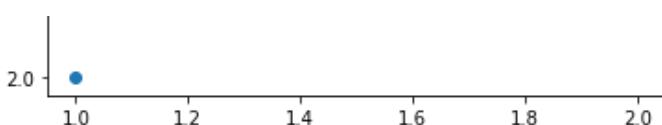
In [96]:

```

plt.scatter(X,Y)
plt.show()
plt.clf()

```





<Figure size 432x288 with 0 Axes>

## Linear model (simple)

In [97]:

```
input_dim = len(X[0])

model = create_linear_model(input_dim)

flattened_dataset_inputs = np.array(X).flatten()

test_before = predict_linear_model_regression(model, input_dim, [3])

train_linear_regression_model(model, input_dim, flattened_dataset_inputs, Y)

test_after = predict_linear_model_regression(model, input_dim, [3])

print("before:", test_before)
print("after:", test_after)

destroy_linear_model(model)

before: -2.8970000743865967
after: 4.0
```

## Linear model (advanced)

In [98]:

```
input_dim = len(X[0])

model = create_linear_model(input_dim)
flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])

test_dataset_inputs = [i for i in range(0, 6)]
predicted_outputs = [predict_linear_model_regression(model, input_dim, [p]) for p in test_dataset_inputs]

plt.plot(test_dataset_inputs, predicted_outputs)
plt.scatter([p[0] for p in X], Y, s=200)
plt.axis([0, 5, 0, 5])
plt.show()

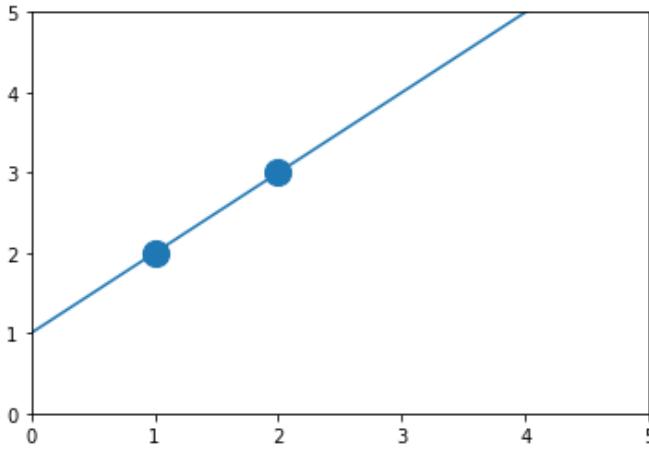
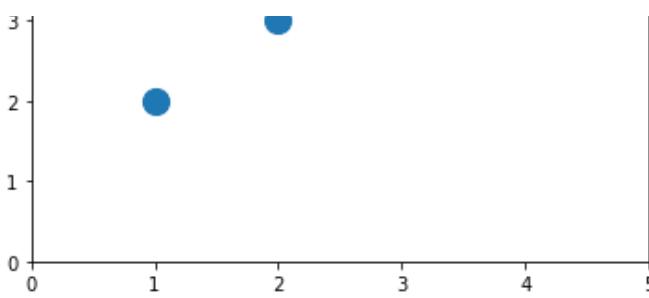
train_linear_regression_model(model, input_dim, flattened_dataset_inputs, Y)

test_dataset_inputs = [i for i in range(0, 6)]
predicted_outputs = [predict_linear_model_regression(model, input_dim, [p]) for p in test_dataset_inputs]

plt.plot(test_dataset_inputs, predicted_outputs)
plt.scatter([p[0] for p in X], Y, s=200)
plt.axis([0, 5, 0, 5])
plt.show()

destroy_linear_model(model)
```





## MLP (simple)

In [99]:

```
model, _ = create_mlp_model([1, 1])

test_before = predict_mlp_model_regression(model, [3])[0]

train_regression_stochastic_gradient_backpropagation_mlp_model(model,
                                                                X,
                                                                Y)

test_after = predict_mlp_model_regression(model, [3])[0]

print("before:", test_before)
print("after:", test_after)

destroy_mlp_model(model)
```

before: -1.0

after: 4.415881156921387

## MLP (advanced)

In [100]:

```
model, _ = create_mlp_model([1, 1])

test_dataset_inputs = [i for i in range(-10, 11)]
predicted_outputs = [predict_mlp_model_regression(model, [p])[0] for p in test_dataset_inputs]

plt.plot(test_dataset_inputs, predicted_outputs)
plt.scatter([p[0] for p in X], Y, s=200)
plt.axis([-10, 10, -10, 10])
plt.show()

train_regression_stochastic_gradient_backpropagation_mlp_model(model,
                                                                X,
                                                                Y)

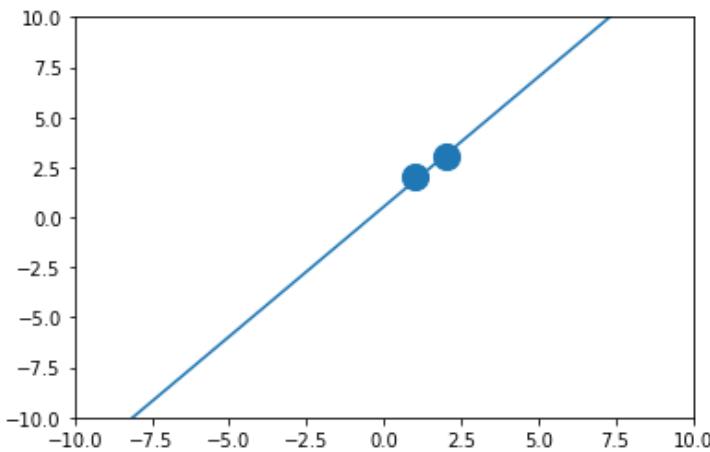
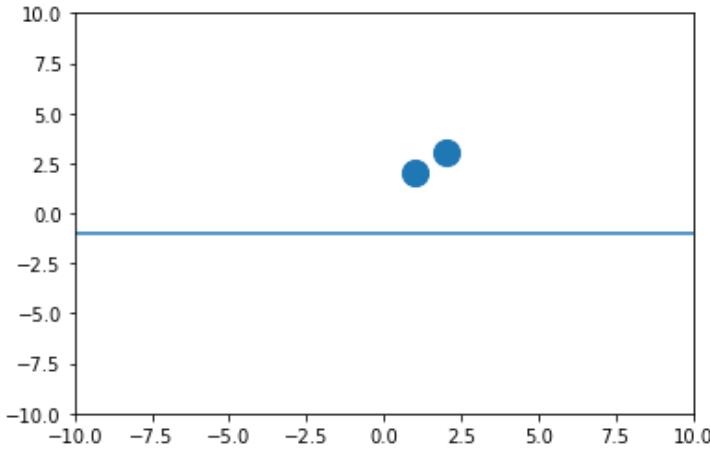
test_dataset_inputs = [i for i in range(-10, 11)]
predicted_outputs = [predict_mlp_model_regression(model, [p])[0] for p in test_dataset_inputs]
```

```

inputs]

plt.plot(test_dataset_inputs, predicted_outputs)
plt.scatter([p[0] for p in X], Y, s=200)
plt.axis([-10, 10, -10, 10])
plt.show()

```



## RBF

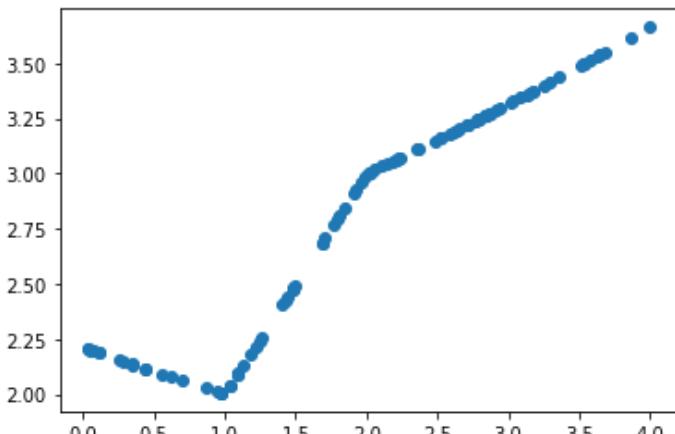
In [101]:

```

num_classes = 1
k = 2
input_dim = 1
expected_output = Y
modif_input = X
model = create_rbfm_model(input_dim, num_classes, k)
train_rbfm_model(model, modif_input, expected_output, True)

plot_input = np.random.random((100, 1)) * 4.0
plot_output_value = [predict[0] for predict in [predict_rbfm(model, coord, 1) for coord
in plot_input]]
plt.scatter(plot_input, plot_output_value)
plt.show()

```



## Non Linear Simple 2D :

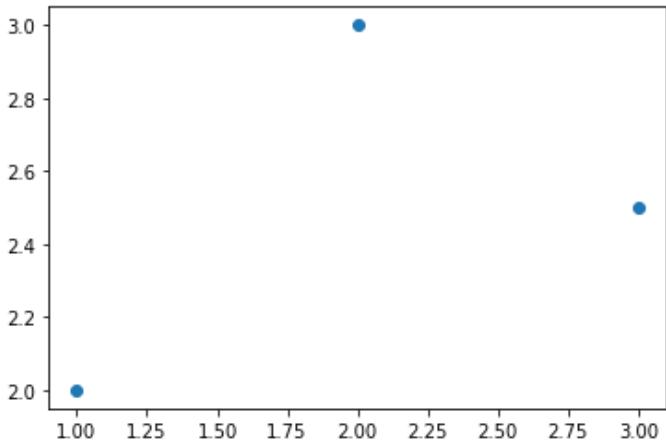
```
Linear Model      : OK
MLP (1, ?, 1)    : OK
```

In [102]:

```
X = np.array([
    [1],
    [2],
    [3]
])
Y = np.array([
    2,
    3,
    2.5
])
```

In [103]:

```
plt.scatter(X,Y)
plt.show()
plt.clf()
```



<Figure size 432x288 with 0 Axes>

## Linear model (simple)

In [104]:

```
input_dim = len(X[0])

model = create_linear_model(input_dim)

flattened_dataset_inputs = np.array(X).flatten()

test_before = predict_linear_model_regression(model, input_dim, [3])

train_linear_regression_model(model, input_dim, flattened_dataset_inputs, Y)

test_after = predict_linear_model_regression(model, input_dim, [3])

print("before:", test_before)
print("after:", test_after)

destroy_linear_model(model)
```

```
before: 1.5410001277923584
after: 2.75
```

## Linear model (advanced)

In [105]:

```
input_dim = len(X[0])

model = create_linear_model(input_dim)
flattened_dataset_inputs = []
for p in X:
    flattened_dataset_inputs.append(p[0])

test_dataset_inputs = [i for i in range(0, 6)]
predicted_outputs = [predict_linear_model_regression(model, input_dim, [p]) for p in test_dataset_inputs]

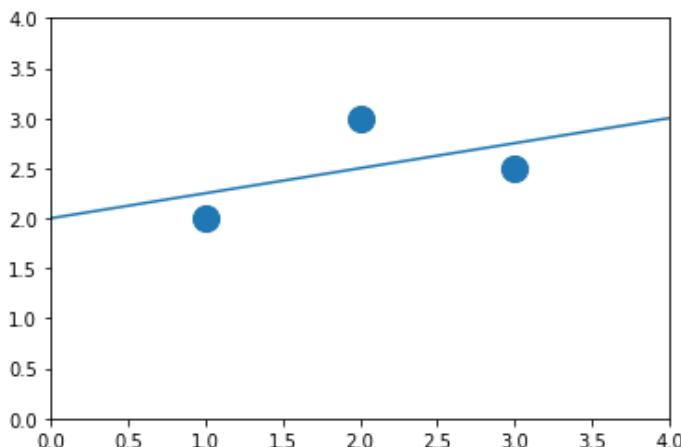
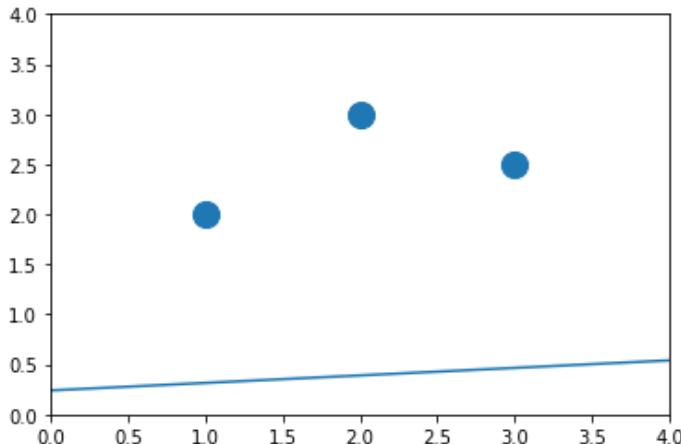
plt.plot(test_dataset_inputs, predicted_outputs)
plt.scatter([p[0] for p in X], Y, s=200)
plt.axis([0, 4, 0, 4])
plt.show()

train_linear_regression_model(model, input_dim, flattened_dataset_inputs, Y)

test_dataset_inputs = [i for i in range(0, 6)]
predicted_outputs = [predict_linear_model_regression(model, input_dim, [p]) for p in test_dataset_inputs]

plt.plot(test_dataset_inputs, predicted_outputs)
plt.scatter([p[0] for p in X], Y, s=200)
plt.axis([0, 4, 0, 4])
plt.show()

destroy_linear_model(model)
```



## MLP (simple)

On a décidé de mettre 1 couche cachée de 2 neurones.

In [106]:

```

model, _ = create_mlp_model([1, 1, 1])

test_before = predict_mlp_model_regression(model, [3])[0]

train_regression_stochastic_gradient_backpropagation_mlp_model(model,
                                                                X,
                                                                Y)

test_after = predict_mlp_model_regression(model, [3])[0]

print("before:", test_before)
print("after:", test_after)

destroy_mlp_model(model)

```

before: 0.0  
after: 2.513078451156616

## MLP (advanced)

In [107]:

```

model, _ = create_mlp_model([1, 2, 1])

test_dataset_inputs = [i for i in range(-10, 11)]
predicted_outputs = [predict_mlp_model_regression(model, [p])[0] for p in test_dataset_inputs]

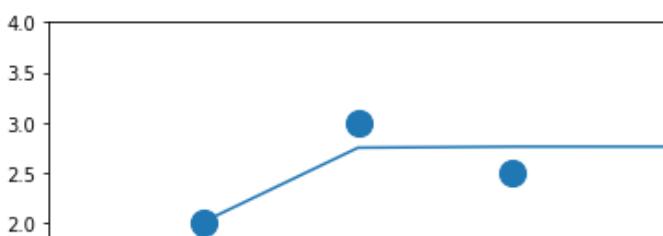
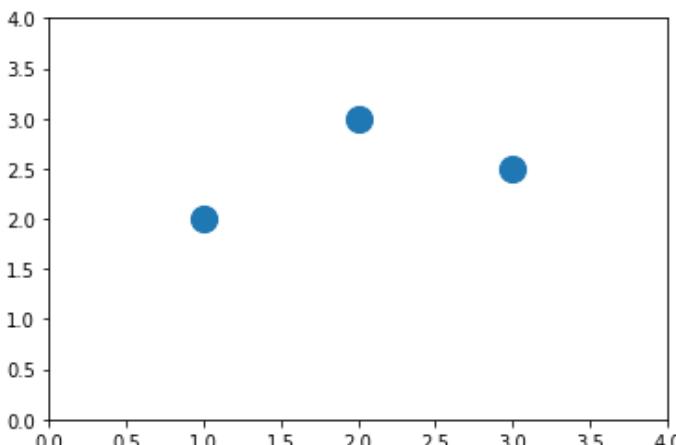
plt.plot(test_dataset_inputs, predicted_outputs)
plt.scatter([p[0] for p in X], Y, s=200)
plt.axis([0,4,0,4])
plt.show()

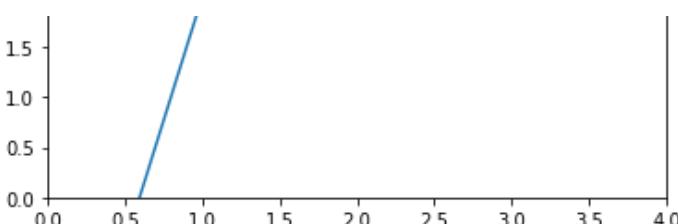
train_regression_stochastic_gradient_backpropagation_mlp_model(model,
                                                                X,
                                                                Y,
                                                                epochs=1000000)

test_dataset_inputs = [i for i in range(-10, 11)]
predicted_outputs = [predict_mlp_model_regression(model, [p])[0] for p in test_dataset_inputs]

plt.plot(test_dataset_inputs, predicted_outputs)
plt.scatter([p[0] for p in X], Y, s=200)
plt.axis([0,4,0,4])
plt.show()

```



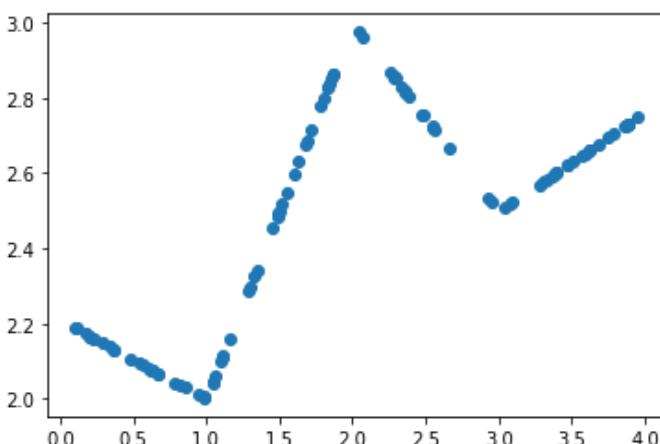


## RBF

In [108]:

```
num_classes = 1
k = 3
input_dim = 1
expected_output = Y
modif_input = X
model = create_rbfm_model(input_dim, num_classes, k)
train_rbfm_model(model, modif_input, expected_output, True)

plot_input = np.random.random((100, 1)) * 4.0
plot_output_value = [predict[0] for predict in predict_rbfm(model, coord, 1) for coord in plot_input]
plt.scatter(plot_input, plot_output_value)
plt.show()
```



## Linear Simple 3D :

Linear Model	: OK
MLP (2, 1)	: OK

In [134]:

```
X = np.array([
    [1, 1],
    [2, 2],
    [6, 1]
])
Y = np.array([
    2,
    3,
    5
])
```

In [135]:

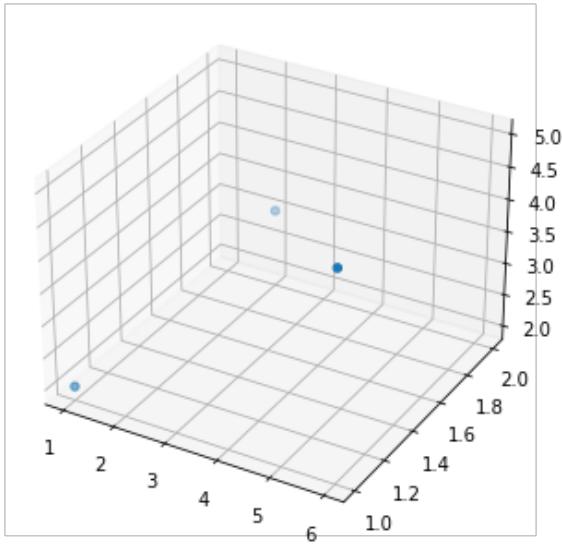
```
from mpl_toolkits.mplot3d import Axes3D
#pip install plotly
import plotly.express as px
import pandas as pd
import plotly.io as pio
```

```

pio.renderers.default='notebook'

fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(X[:,0],X[:,1],Y)
plt.show()
plt.clf()

```



<Figure size 432x288 with 0 Axes>

## Linear model (simple)

Sachant que notre matrice d'input était  $\{(1,2,3),(1,2,3)\}$ , les deux colonnes sont donc des combinaisons linéaire l'une de l'autre ( $C_1 = C_2 ; L_2 = 2L_1 \dots$ ) de ce fait lors de l'exécution plus avant de la librairie il nous est impossible de réaliser l'inverse cette matrice. Pour contrer ce problème nous avons découvert qu'il était possible de dupliquer une des lignes du dataset pour briser cette restriction sans créer de nouvelle datas. Une autre possibilité proposée mais moins sur car elle ajoutait de nouvelles données au dataset. Les nouveaux points étaient des jumeaux des points déjà existant avec des coordonnées décaler sur le repère à l'échelle de 0.00001 pour minimiser l'impact.

In [136]:

```

input_dim = len(X[0])

model = create_linear_model(input_dim)

flattened_dataset_inputs = np.array(X).flatten()

test_before = predict_linear_model_regression(model, input_dim, [1, 1])

train_linear_regression_model(model, input_dim, flattened_dataset_inputs, Y)

test_after = predict_linear_model_regression(model, input_dim, [1, 1])

print("before:", test_before)
print("after:", test_after)

destroy_linear_model(model)

before: -0.04399997740983963
after: 2.357142925262451

```

## Linear model (advanced)

In [137]:

```

input_dim = len(X[0])

model = create_linear_model(input_dim)

```

```

flattened_dataset_inputs = np.array(X).flatten()

test_dataset_inputs = np.array([[i, j] for i in np.arange(0, 6, 0.1) for j in np.arange(0, 6, 0.1)])
predicted_outputs = np.array([predict_linear_model_regression(model, input_dim, x) for x in test_dataset_inputs])

df = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_prev": predicted_outputs})

old = {"cat": ["B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_prev": Y}

df2 = pd.DataFrame(old)
d = pd.concat([df, df2], ignore_index = True)
d.reset_index()

fig = px.scatter_3d(d, x="x0", y="x1", z="z_prev", color="cat", size=[10 for _ in range(len(d))])
# fig.show("notebook")

train_linear_regression_model(model, input_dim, flattened_dataset_inputs, Y)

predicted_outputs = np.array([predict_linear_model_regression(model, input_dim, x) for x in test_dataset_inputs])

df_aft = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_aft": predicted_outputs})

old2 = {"cat": ["B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_aft": Y}

df2 = pd.DataFrame(old2)
d_aft = pd.concat([df_aft, df2], ignore_index = True)
d_aft.reset_index()

fig = px.scatter_3d(d_aft, x="x0", y="x1", z="z_aft", color="cat", size=[10 for _ in range(len(d_aft))])
# fig.show("notebook")

destroy_linear_model(model)

```

## MLP (simple)

In [138]:

```

input_dim = [len(X[0]), 1]

model, _ = create_mlp_model(input_dim)

test_before = predict_mlp_model_regression(model, [1, 1])[0]

train_regression_stochastic_gradient_backpropagation_mlp_model(model,
                                                                X,
                                                                Y,
                                                                epochs=100000)

test_after = predict_mlp_model_regression(model, [1, 1])[0]

print("before:", test_before)
print("after:", test_after)

destroy_mlp_model(model)

before: -1.0
after: 1.999664068222046

```

## MLP (advanced)

In [139]:

```
input_dim = [len(X[0]), 1]

model, _ = create_mlp_model(input_dim)

test_dataset_inputs = np.array([[i, j] for i in np.arange(0, 6, 0.1) for j in np.arange(0, 6, 0.1)])
predicted_outputs = np.array([predict_mlp_model_regression(model, x)[0] for x in test_dataset_inputs])

df = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_prev": predicted_outputs})

old = {"cat": ["B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_prev": Y}

df2 = pd.DataFrame(old)
d = pd.concat([df, df2], ignore_index = True)
d.reset_index()

fig = px.scatter_3d(d, x="x0", y="x1", z="z_prev", color="cat", size=[10 for _ in range(len(d))])
# fig.show("notebook")

train_regression_stochastic_gradient_backpropagation_mlp_model(model,
                                                                X,
                                                                Y,
                                                                epochs=100000)

predicted_outputs = np.array([predict_mlp_model_regression(model, x)[0] for x in test_dataset_inputs])

df_aft = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_aft": predicted_outputs})

old2 = {"cat": ["B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_aft": Y}

df2 = pd.DataFrame(old2)
d_aft = pd.concat([df_aft, df2], ignore_index = True)
d_aft.reset_index()

fig = px.scatter_3d(d_aft, x="x0", y="x1", z="z_aft", color="cat", size=[10 for _ in range(len(d_aft))])
# fig.show("notebook")

destroy_mlp_model(model)
```

## RBF

In [140]:

```
num_classes = 1
k = 3
input_dim = 2
expected_output = Y
modif_input = X
model = create_rbfm_model(input_dim, num_classes, k)
train_rbfm_model(model, modif_input, expected_output, True)

plot_input = np.array([[i, j] for i in np.arange(0, 6, 0.1) for j in np.arange(0, 6, 0.1)])
plot_output_value = [predict[0] for predict in [predict_rbfm(model, coord, num_classes) for coord in plot_input]]

df_aft = pd.DataFrame({"cat": ["A" for _ in range(len(plot_input))], "x0": plot_input[:, 0], "x1": plot_input[:, 1], "z_aft": plot_output_value})
```

```

old2 = {"cat": ["B", "B", "B"], "x0": X[:,0], "x1": X[:,1], "z_aft":Y}

df2 = pd.DataFrame(old2)
d_aft = pd.concat([df_aft, df2], ignore_index = True)
d_aft.reset_index()

fig = px.scatter_3d(d_aft, x="x0", y="x1", z="z_aft", color="cat", size=[10 for _ in range(len(d_aft))])
# fig.show("notebook")

```

## Linear Tricky 3D :

Linear Model	:	OK
MLP (2, 1)	:	OK

In [141]:

```

X = np.array([
    [1, 1],
    [2, 2],
    [3, 3]
])
Y = np.array([
    1,
    2,
    3
])

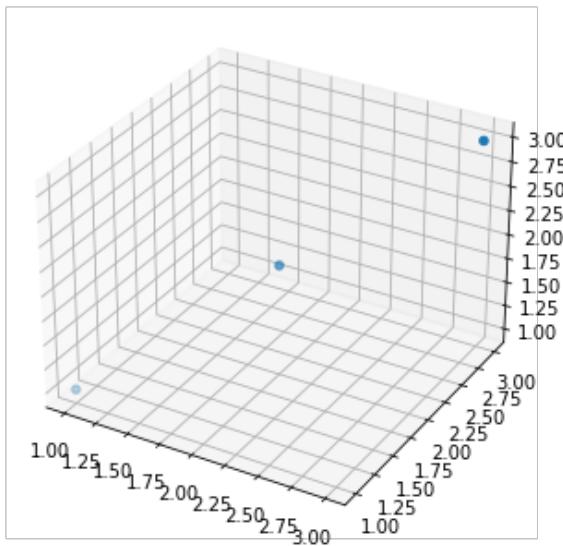
```

In [142]:

```

from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(X[:,0], X[:,1], Y)
plt.show()
plt.clf()

```



<Figure size 432x288 with 0 Axes>

## Linear model (simple)

In [143]:

```

input_dim = len(X[0])

model = create_linear_model(input_dim)

```

```

flattened_dataset_inputs = np.array(X).flatten()

test_before = predict_linear_model_regression(model, input_dim, [1, 1])

train_linear_regression_model(model, input_dim, flattened_dataset_inputs, Y)

test_after = predict_linear_model_regression(model, input_dim, [1, 1])

print("before:", test_before)
print("after:", test_after)

destroy_linear_model(model)

```

before: 0.27000004053115845  
after: 1.0

## Linear model (advanced)

In [144]:

```

input_dim = len(X[0]) # 2

model = create_linear_model(input_dim)

flattened_dataset_inputs = np.array(X).flatten()

test_dataset_inputs = np.array([[i, j] for i in np.arange(0, 6, 0.1) for j in np.arange(0, 6, 0.1)])
predicted_outputs = np.array([predict_linear_model_regression(model, input_dim, x) for x in test_dataset_inputs])

df = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_prev": predicted_outputs})

old = {"cat": ["B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_prev": Y}

df2 = pd.DataFrame(old)
d = pd.concat([df, df2], ignore_index = True)
d.reset_index()

fig = px.scatter_3d(d, x="x0", y="x1", z="z_prev", color="cat", size=[10 for _ in range(len(d))])
# fig.show("notebook")
#print(df.head())

train_linear_regression_model(model, input_dim, flattened_dataset_inputs, Y)

predicted_outputs = np.array([predict_linear_model_regression(model, input_dim, x) for x in test_dataset_inputs])

df_aft = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_aft": predicted_outputs})

old2 = {"cat": ["B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_aft": Y}

df2 = pd.DataFrame(old2)
d_aft = pd.concat([df_aft, df2], ignore_index = True)
d_aft.reset_index()

fig = px.scatter_3d(d_aft, x="x0", y="x1", z="z_aft", color="cat", size=[10 for _ in range(len(d_aft))])
# fig.show("notebook")

destroy_linear_model(model)

```

## MLP (simple)

In [145]:

```

input_dim = [len(X[0]), 1]

model, _ = create_mlp_model(input_dim)

test_before = predict_mlp_model_regression(model, [1, 1])[0]

train_regression_stochastic_gradient_backpropagation_mlp_model(model,
                                                                X,
                                                                Y,
                                                                epochs=100000)

test_after = predict_mlp_model_regression(model, [1, 1])[0]

print("before:", test_before)
print("after:", test_after)

destroy_mlp_model(model)

```

before: -2.0  
after: 0.9999992847442627

## MLP (advanced)

In [146]:

```

input_dim = [len(X[0]), 1]

model, _ = create_mlp_model(input_dim)

test_dataset_inputs = np.array([[i, j] for i in np.arange(0, 6, 0.1) for j in np.arange(0, 6, 0.1)])
predicted_outputs = np.array([predict_mlp_model_regression(model, x)[0] for x in test_dataset_inputs])

df = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_prev": predicted_outputs})

old = {"cat": ["B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_prev": Y}

df2 = pd.DataFrame(old)
d = pd.concat([df, df2], ignore_index = True)
d.reset_index()

fig = px.scatter_3d(d, x="x0", y="x1", z="z_prev", color="cat", size=[10 for _ in range(len(d))])
# fig.show("notebook")

train_regression_stochastic_gradient_backpropagation_mlp_model(model,
                                                                X,
                                                                Y,
                                                                epochs=100000)

predicted_outputs = np.array([predict_mlp_model_regression(model, x)[0] for x in test_dataset_inputs])

df_aft = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_aft": predicted_outputs})

old2 = {"cat": ["B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_aft": Y}

df2 = pd.DataFrame(old2)
d_aft = pd.concat([df_aft, df2], ignore_index = True)
d_aft.reset_index()

fig = px.scatter_3d(d_aft, x="x0", y="x1", z="z_aft", color="cat", size=[10 for _ in range(len(d_aft))])

```

```
# fig.show("notebook")
```

```
destroy_mlp_model(model)
```

## RBF

In [147]:

```
num_classes = 1
k = 3
input_dim = 2
expected_output = Y
modif_input = X
model = create_rbfm_model(input_dim, num_classes, k)
train_rbfm_model(model, modif_input, expected_output, True)

plot_input = np.array([[i, j] for i in np.arange(0, 6, 0.1) for j in np.arange(0, 6, 0.1)])
plot_output_value = [predict[0] for predict in [predict_rbfm(model, coord, num_classes) for coord in plot_input]]

df_aft = pd.DataFrame({"cat": ["A" for _ in range(len(plot_input))], "x0": plot_input[:, 0], "x1": plot_input[:, 1], "z_aft": plot_output_value})

old2 = {"cat": ["B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_aft": Y}

df2 = pd.DataFrame(old2)
d_aft = pd.concat([df_aft, df2], ignore_index = True)
d_aft.reset_index()

fig = px.scatter_3d(d_aft, x="x0", y="x1", z="z_aft", color="cat", size=[10 for _ in range(len(d_aft))])
# fig.show("notebook")
```

## Non Linear Simple 3D :

Linear Model	:	KO
MLP (2, 2, 1)	:	OK

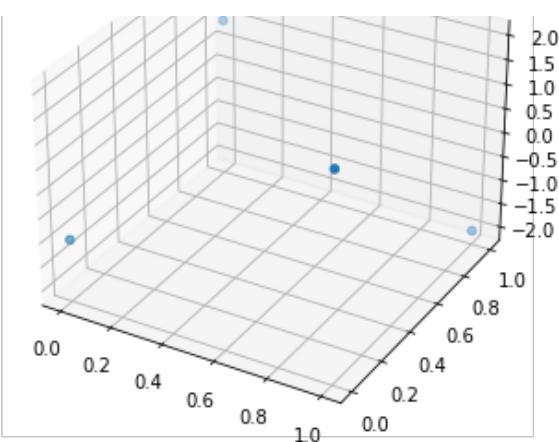
In [148]:

```
X = np.array([
    [1, 0],
    [0, 1],
    [1, 1],
    [0, 0],
])
Y = np.array([
    2,
    1,
    -2,
    -1
])
```

In [149]:

```
from mpl_toolkits.mplot3d import Axes3D
fig = plt.figure()
ax = Axes3D(fig)
ax.scatter(X[:, 0], X[:, 1], Y)
plt.show()
plt.clf()
```





<Figure size 432x288 with 0 Axes>

## Linear model (simple)

In [150]:

```
input_dim = len(X[0])

model = create_linear_model(input_dim)

flattened_dataset_inputs = np.array(X).flatten()

test_before = predict_linear_model_regression(model, input_dim, [1, 0])

train_linear_regression_model(model, input_dim, flattened_dataset_inputs, Y)

test_after = predict_linear_model_regression(model, input_dim, [1, 0])

print("before:", test_before)
print("after:", test_after)

destroy_linear_model(model)

before: 0.23399999737739563
after: -0.008695651777088642
```

## Linear model (advanced)

In [151]:

```
input_dim = len(X[0])

model = create_linear_model(input_dim)

flattened_dataset_inputs = np.array(X).flatten()

test_dataset_inputs = np.array([[i, j] for i in np.arange(0, 6, 0.1) for j in np.arange(0, 6, 0.1)])
predicted_outputs = np.array([predict_linear_model_regression(model, input_dim, x) for x in test_dataset_inputs])

df = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_prev": predicted_outputs})

old = {"cat": ["B", "B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_prev": Y}

df2 = pd.DataFrame(old)
d = pd.concat([df, df2], ignore_index = True)
d.reset_index()

fig = px.scatter_3d(d, x="x0", y="x1", z="z_prev", color="cat", size=[10 for _ in range(len(d))])
# fig.show("notebook")
```

```

train_linear_regression_model(model, input_dim, flattened_dataset_inputs, Y)

predicted_outputs = np.array([predict_linear_model_regression(model, input_dim, x) for x in test_dataset_inputs])

df_aft = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_aft": predicted_outputs})

old2 = {"cat": ["B", "B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_aft": Y}

df2 = pd.DataFrame(old2)
d_aft = pd.concat([df_aft, df2], ignore_index = True)
d_aft.reset_index()

fig = px.scatter_3d(d_aft, x="x0", y="x1", z="z_aft", color="cat", size=[10 for _ in range(len(d_aft))])
# fig.show("notebook")

destroy_linear_model(model)

```

## MLP (simple)

In [152]:

```

input_dim = [len(X[0]), len(X[0]), 1]

model, _ = create_mlp_model(input_dim)

test_before = predict_mlp_model_regression(model, [1, 0])[0]

train_regression_stochastic_gradient_backpropagation_mlp_model(model,
                                                                X,
                                                                Y,
                                                                epochs=100000)

```

```
test_after = predict_mlp_model_regression(model, [1, 0])[0]
```

```
print("before:", test_before)
print("after:", test_after)
```

```
destroy_mlp_model(model)
```

before: 0.0

after: 1.9998633861541748

## MLP (advanced)

In [153]:

```

input_dim = [len(X[0]), len(X[0]), 1]

model, _ = create_mlp_model(input_dim)

test_dataset_inputs = np.array([[i, j] for i in np.arange(0, 6, 0.1) for j in np.arange(0, 6, 0.1)])
predicted_outputs = np.array([predict_mlp_model_regression(model, x)[0] for x in test_dataset_inputs])

df = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_prev": predicted_outputs})

old = {"cat": ["B", "B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_prev": Y}

df2 = pd.DataFrame(old)
d = pd.concat([df, df2], ignore_index = True)
d.reset_index()

```

```

fig = px.scatter_3d(d, x="x0", y="x1", z="z_prev", color="cat", size=[10 for _ in range(len(d))])
# fig.show("notebook")

train_regression_stochastic_gradient_backpropagation_mlp_model(model,
                                                                X,
                                                                Y,
                                                                alpha=0.01,
                                                                epochs=100000)

predicted_outputs = np.array([predict_mlp_model_regression(model, x)[0] for x in test_dataset_inputs])

df_aft = pd.DataFrame({"cat": ["A" for _ in range(len(test_dataset_inputs))], "x0": test_dataset_inputs[:, 0], "x1": test_dataset_inputs[:, 1], "z_aft": predicted_outputs})

old2 = {"cat": ["B", "B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_aft": Y}

df2 = pd.DataFrame(old2)
d_aft = pd.concat([df_aft, df2], ignore_index = True)
d_aft.reset_index()

fig = px.scatter_3d(d_aft, x="x0", y="x1", z="z_aft", color="cat", size=[10 for _ in range(len(d_aft))])
# fig.show("notebook")

destroy_mlp_model(model)

```

## RBF

In [154]:

```

num_classes = 1
k = 4
input_dim = 2
expected_output = Y
modif_input = X
model = create_rbfm_model(input_dim, num_classes, k)
train_rbfm_model(model, modif_input, expected_output, True)

plot_input = np.array([[i, j] for i in np.arange(0, 2, 0.1) for j in np.arange(0, 2, 0.1)])
plot_output_value = [predict[0] for predict in [predict_rbfm(model, coord, num_classes) for coord in plot_input]]

df_aft = pd.DataFrame({"cat": ["A" for _ in range(len(plot_input))], "x0": plot_input[:, 0], "x1": plot_input[:, 1], "z_aft": plot_output_value})

old2 = {"cat": ["B", "B", "B"], "x0": X[:, 0], "x1": X[:, 1], "z_aft": Y}

df2 = pd.DataFrame(old2)
d_aft = pd.concat([df_aft, df2], ignore_index = True)
d_aft.reset_index()

fig = px.scatter_3d(d_aft, x="x0", y="x1", z="z_aft", color="cat", size=[10 for _ in range(len(d_aft))])
# fig.show("notebook")

```

In [ ]:

In [ ]: