TP 1 R420 Intelligence Artificielle

1 Régression linéaire

a) Ajout calcule des moindres carrés :

```
age=data[0]
salaire=data[1]

y_moyen = np.mean(salaire_tds)
x_moyen = np.mean(age_tds)

covariance = 0
variance = 0

for i in range(len(age_tds)):
    X = age_tds[i] - x_moyen
    Y = salaire_tds[i] - y_moyen
    covariance += X * Y
    variance += X ** 2

a1 = covariance / variance
a0 = y_moyen - a1 * x_moyen
```

Le résultat :

La courbe me semble bien respecter la loi des moindres carrés par le fait que la courbe passe à un écart minimisant la somme des carrés des écarts, entre les valeurs données pour les tests et les valeurs d'apprentissage.

b) Mesure de l'efficacité et commentaire :

Suppression de la graine de génération du nombre aléatoire, afin d'avoir une vraie randomisation du résultat. => `rng=np.random.default_rng()`

Étant donné que la valeur précédente était toujours la même l'apprentissage était très mauvais pour notre objectif, de plus le nombre d'entrées étant faible le coefficient détermination est très faible.

Résultat aprés randomisation de la graine :

c) Modifications du N pour avoir plus d'entrée pour l'apprentissage :

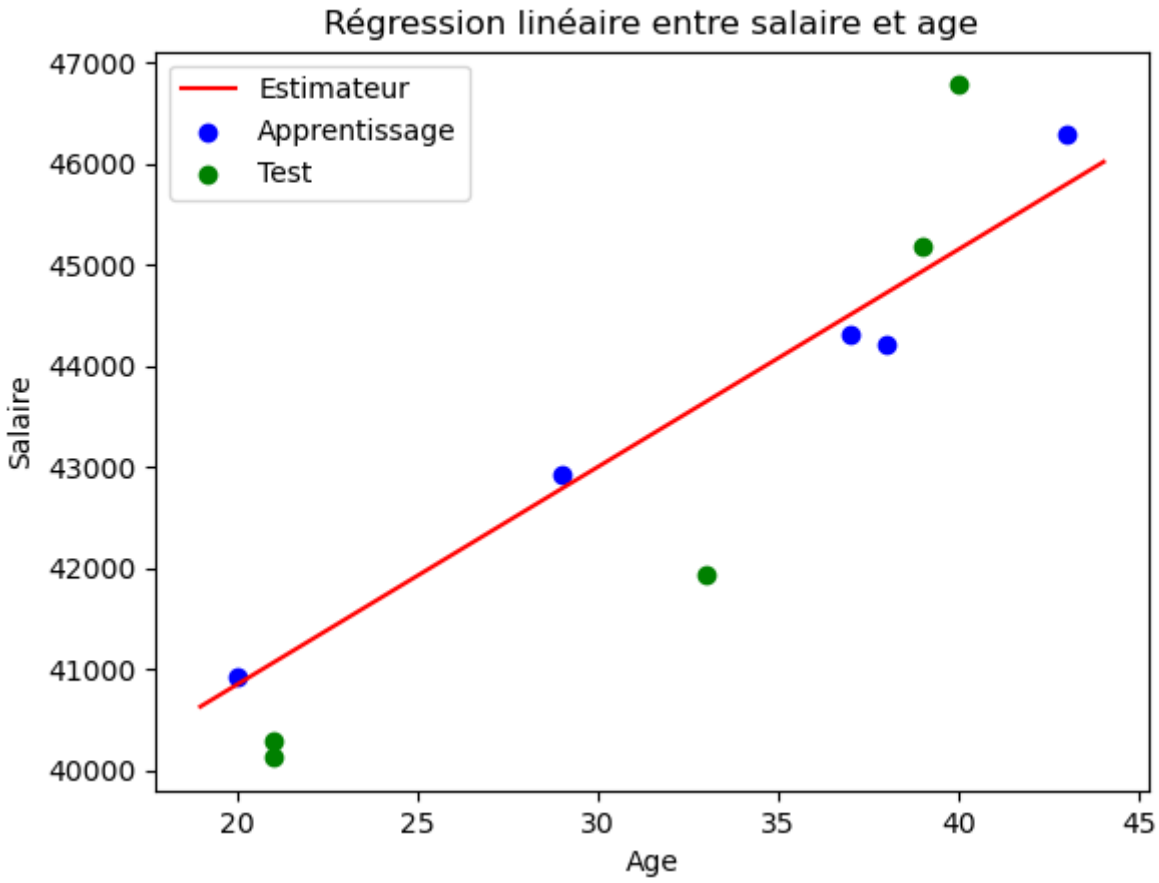
Augmentation du N dans le code correspondant au nombre de données généré aléatoirement à partir de tableau afin de vérifier une loi classique de l'IA :

```
age=rng.integers(low=20, high=50, size=N)
salaire = 35000 + 250*age + rng.normal(0, 1000, size=N)
```

C'est deux lignes génères un tableau de salaire et d'âge. Le tableau age de taille N avec un minimum de 20 ans et un maximum de 50 ans. Le tableau salaire à un minimum de 35000€ et est calculé en fonction de l'âge, j'y ajoute un bruit de variations des salaires.

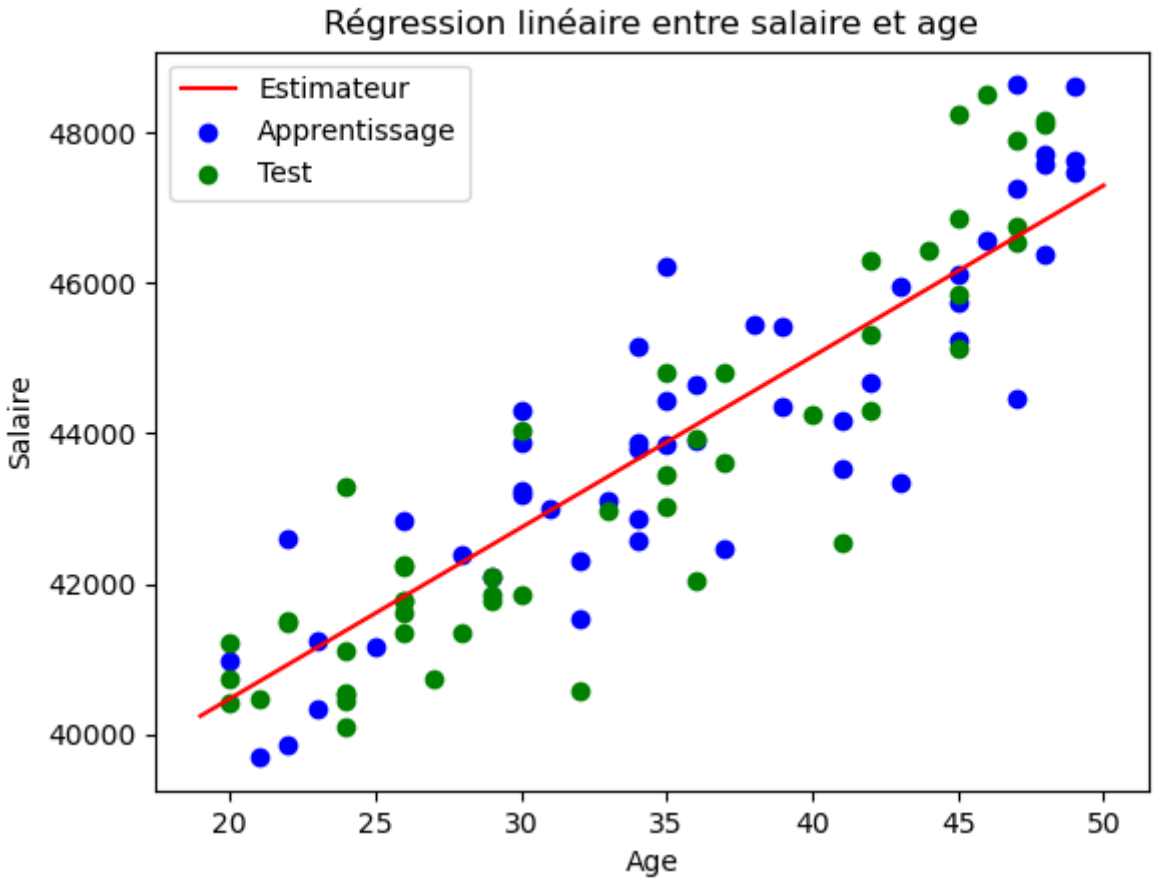
- N=10 : Le prédicteur du salaire est : salaire = 215.48580620021485 * age + 36535.10178859128 Erreur quadratique moyenne d'apprentissage 332.16219478521396 Erreur quadratique moyenne de

test 1189.653259220181 Coefficient de détermination (score R2) : 0.8015794244096321



- N=100 : Le prédicteur du salaire est : salaire = 227.90608123497316 * age + 35908.66679875896 Erreur quadratique moyenne d'apprentissage 1051.4462331693699 Erreur quadratique moyenne de

test 1054.9374367245575 Coefficient de détermination (score R2) : 0.8201031163084601



- N=1000 : Le prédicteur du salaire est : salaire = 246.55548164754921 * age + 35121.26278882333 Erreur quadratique moyenne d'apprentissage = 947.5140693283623 Erreur quadratique moyenne

de test = 1007.7837770619693 Coefficient de détermination (score R2) = 0.8149813896192919

Après ces trois essaie on voit bien que la loi se vérifie lorsque que le N est petit l'erreur est faible, mais l'efficacité est faible. Pour un N plus grand l'erreur augmente, mais l'efficacité est plus importante.

2 Régression polynomiale

a) Adaptation du programme pour faire de la régression polynomiale :

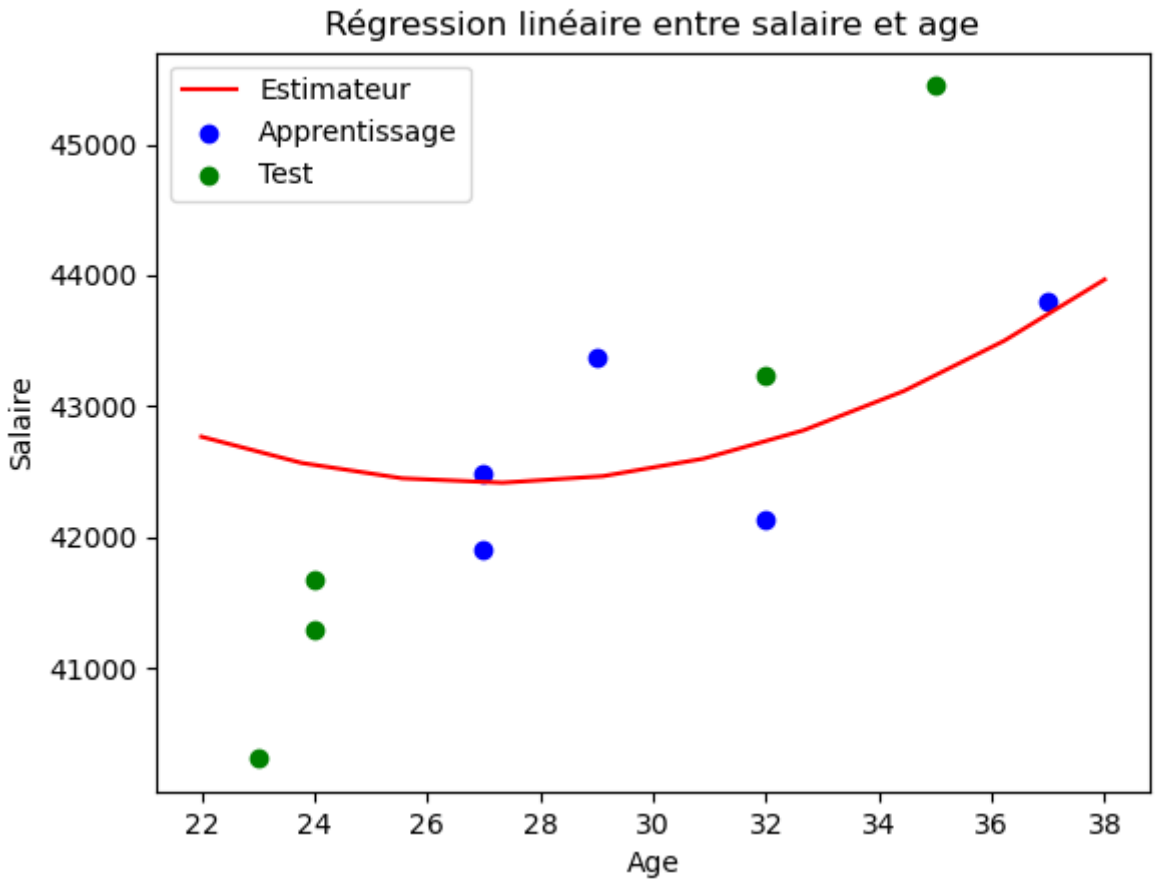
```
#Partie 2A
p = np.polyfit(age_tds, salaire_tds, deg=2)
a2, a1, a0 = p

# Affichage de la droite du prédicteur
x_values = np.linspace(age.min() - 1, age.max() + 1, num=N)
y_values = a0 + a1 * x_values + a2 * x_values ** 2
plt.plot(x_values, y_values, color='red', label="Estimateur")

# Calculer l'erreur d'apprentissage (erreur quadratique moyenne)
y_pred_tds  = a0 + a1 * age_tds + age_test + a2 *age_tds ** 2
rmse_tds = np.sqrt(np.mean((y_pred_tds - salaire_tds)**2))

# Calculer l'erreur de test (erreur quadratique moyenne)
y_pred_test = a0 + a1 * age_test + a2 * age_tds ** 2
rmse_test = np.sqrt(np.mean((y_pred_test - salaire_test)**2))
```

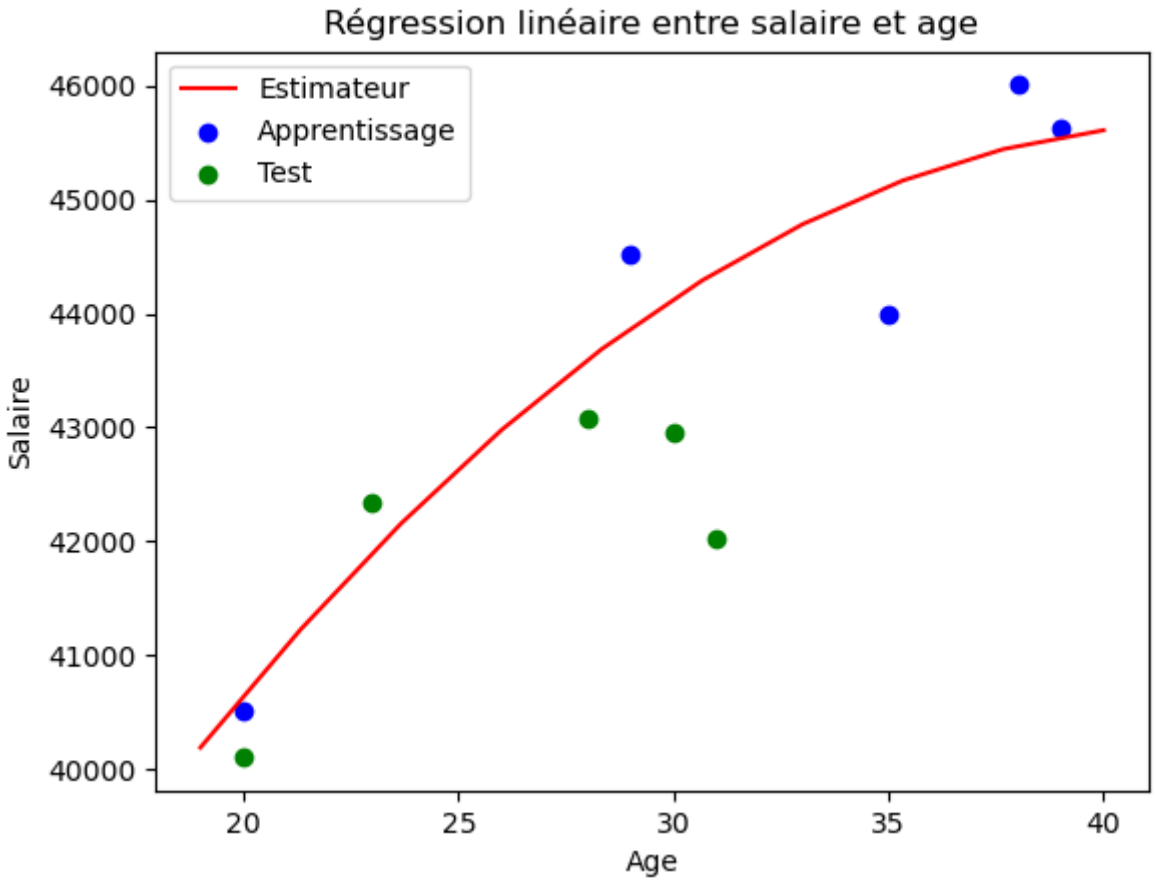
Résultat pour un N = 10 :



b) Vérification de la loi :

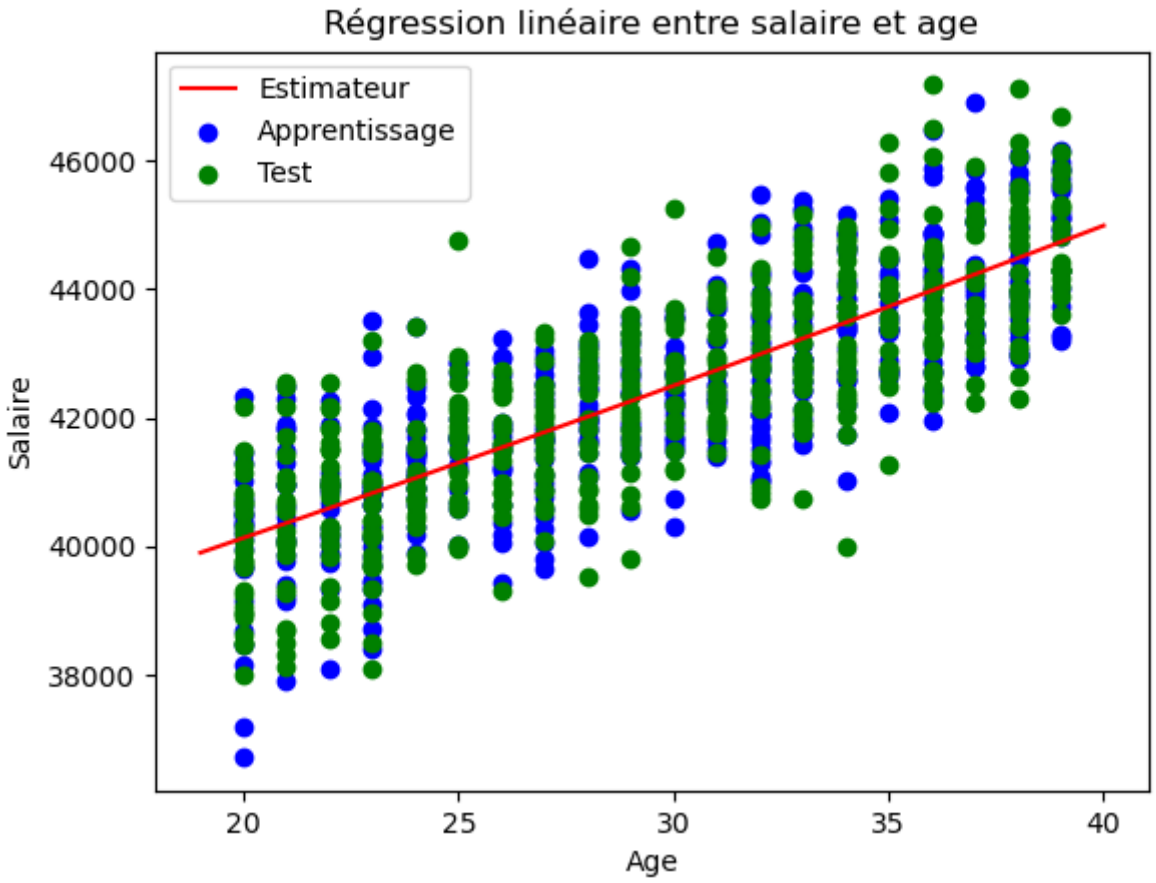
- N = 10 :Le prédicteur du salaire est : salaire = 848.003350226825 * age + 27691.306766548314 Erreur quadratique moyenne d'apprentissage = `630.2577432873293` Erreur quadratique moyenne de test

= 3531.2772190688647 Coefficient de détermination (score R2) = -9.901525389591455



- N = 1000 : Le prédicteur du salaire est : salaire = 205.61464372933247 * age + 35772.530290102615 Erreur quadratique moyenne d'apprentissage = 1014.0814767126684 Erreur quadratique moyenne

de test = 1097.6420756910752 Coefficient de détermination (score R2) = 0.6143304074118314



Après ces deux résultats, on s'aperçoit bien de la vérité de cette loi dans mon cas. Plus il y à de point(N=1000) moins la droite à une erreur importante, mais son efficacité est moindre, et vice versa pour N=10.

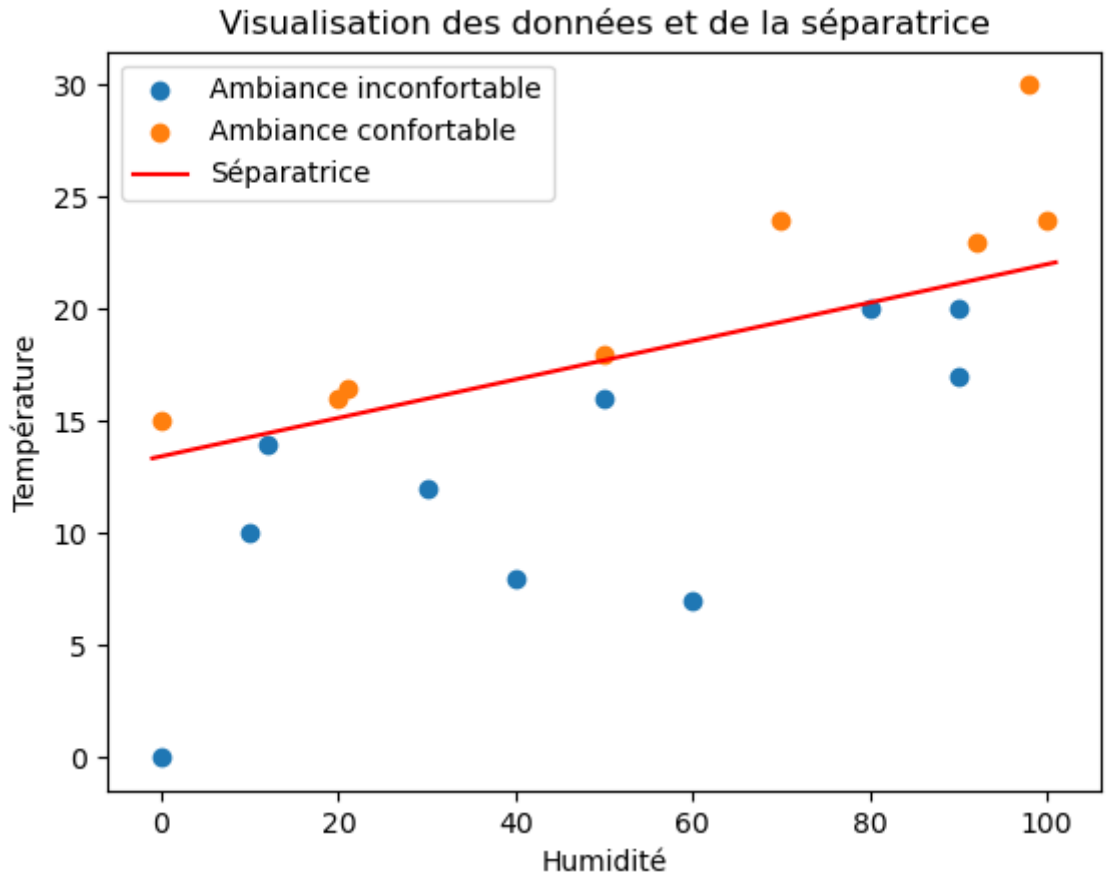
3 Classification par régression logistique

a) Vérifications de la convergence de l'algorithme :

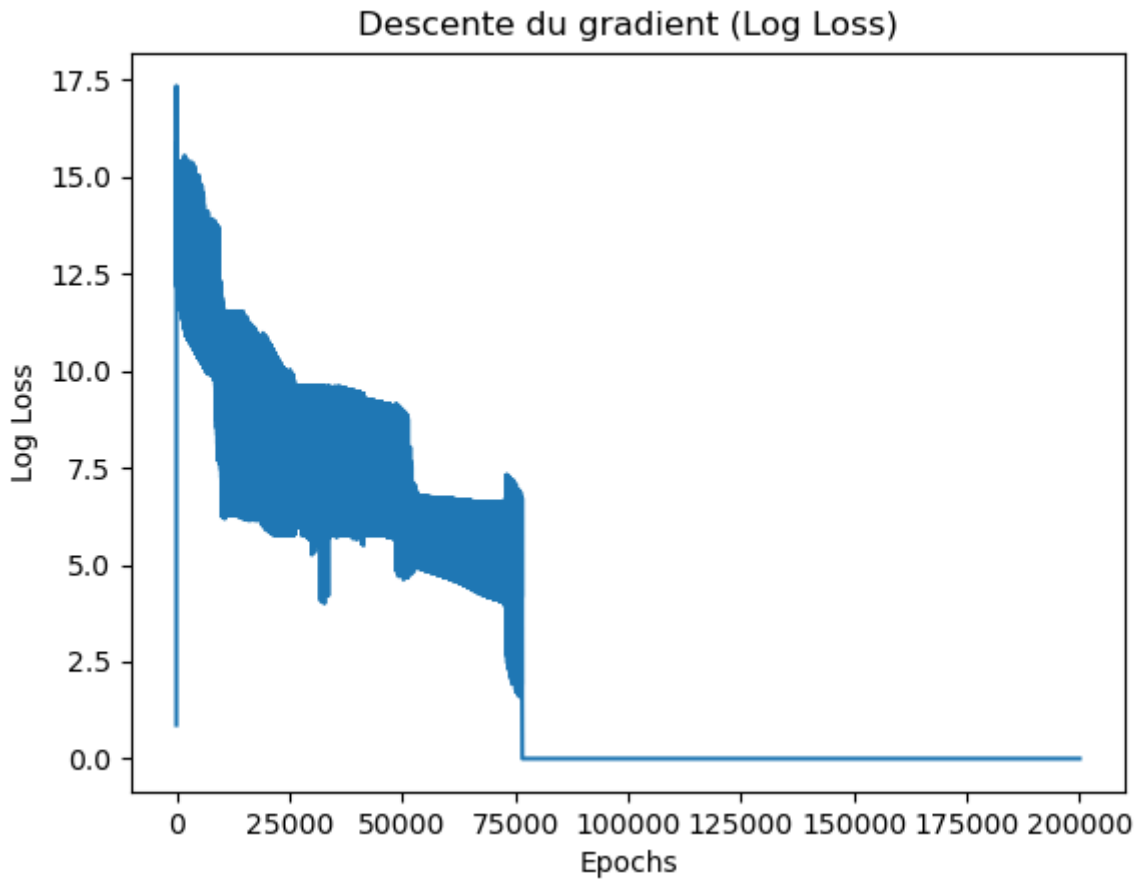
La convergence de l'algorithme est de mieux en mieux au fur et à mesure des itérations, il passe de 0.8 pour 10 000 itérations à 0.2 pour 200 000 itérations.

b) Vérifiacation du taux de sensibilité à l'apprentissage :

En mettant un learning_rate de 0.2 (``def gradient_descent(X, y, learning_rate=0.2, epochs=200000)😃, la courbe sépare bien le milieux confort de l'inconfort et il n'y a aucun point oublié. Matrice de confusions : [[10 0][0 8]] :

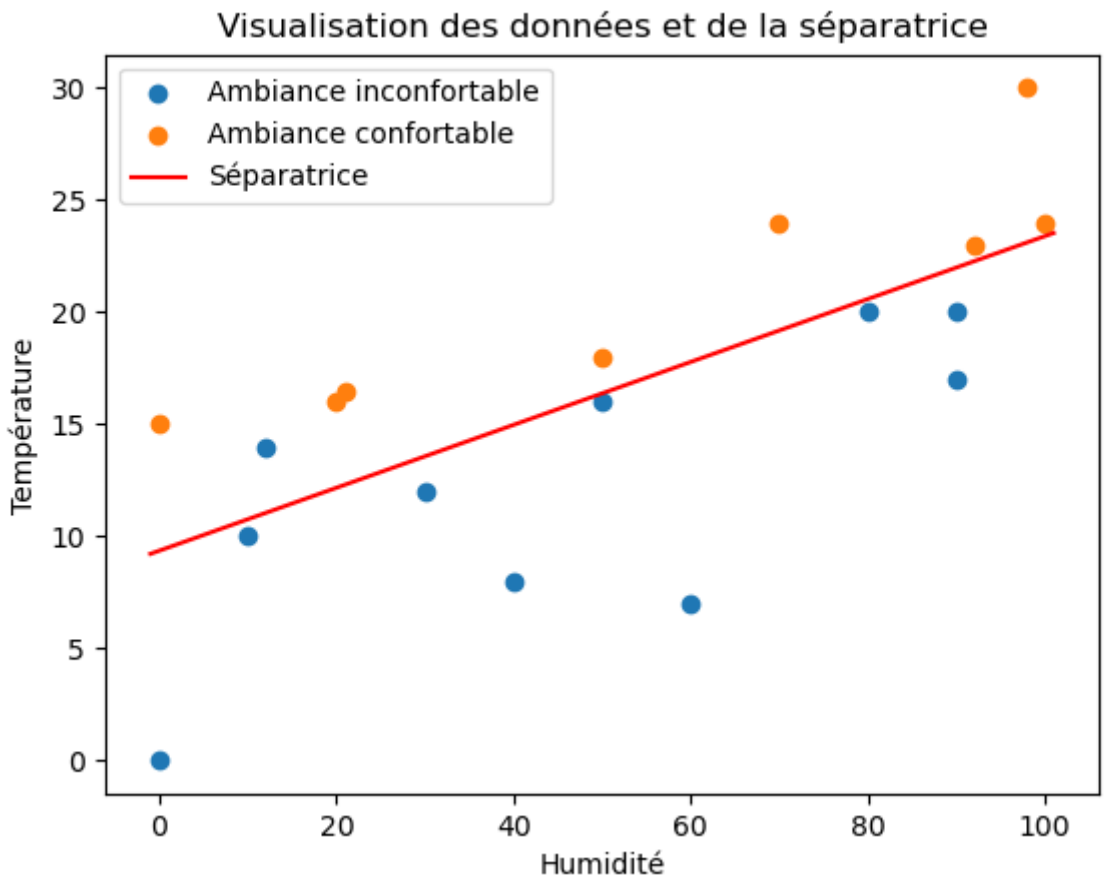


La courbe d'apprentissage est-elle bien plus chaotique que le 1er test et on peut voir que l'apprentissage est fini dés 75 000 itérations ce qui rapide..

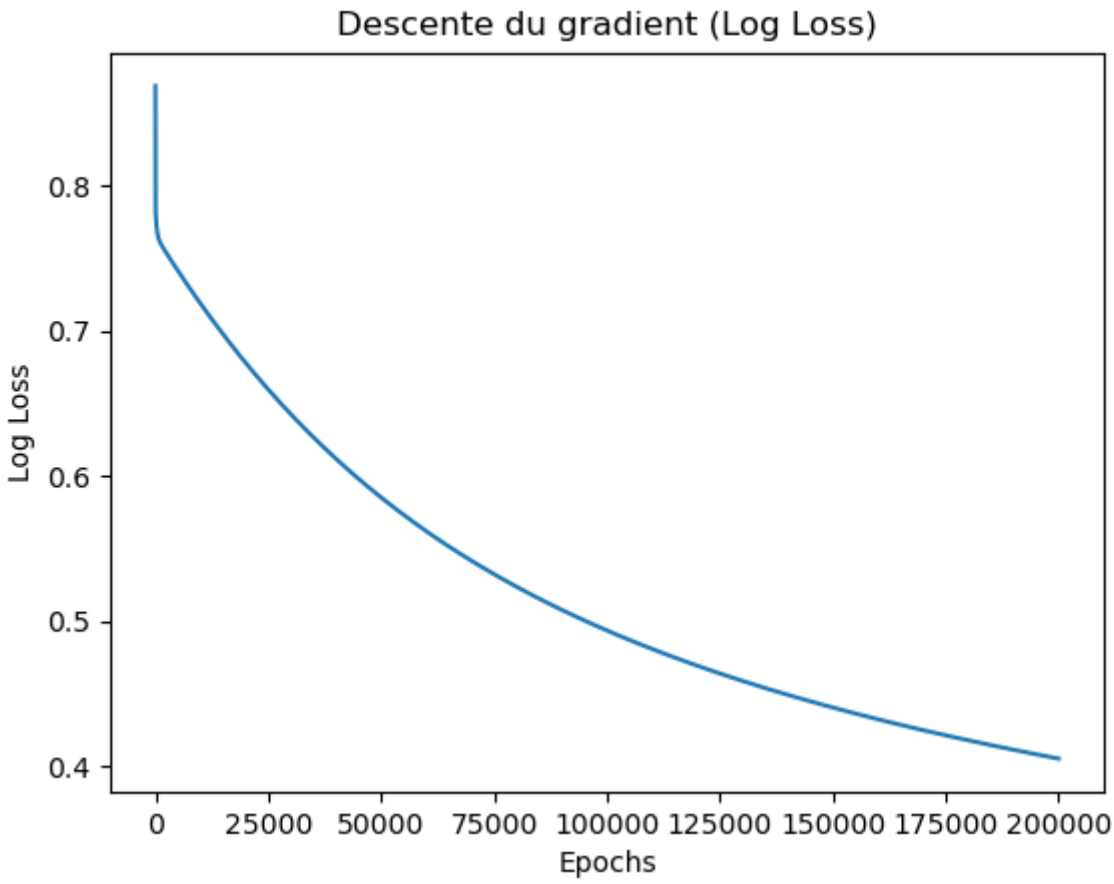


J'essaie maintenant avec un learning_rate de 0.0002 :

La courbe ressemble plus à celle du 1er essaie, avec un 1 confusion un point est mal placé. Matrice de confusion : [[9 1][0 8]] :



La courbe d'apprentissage est-elle bien plus linéaire qu'avant et on peut voir que l'apprentissage est efficace jusqu'à 200 000 itérations.



Pour conclure on voit bien la différence entre utilisé beaucoup de ressources de calcul pour avoir un résultat positif bien plus tôt, que si l'on prend notre temps avec un apprentissage long et petit à petit qui n'est pas nécessairement juste.

c) Modification du programme pour produire un dataset test et calculer la matrice de confusion sur ce dataset :

Ajout de, c'est quelque ligne de code aux bons endroits du code :

```
# Génére un nouveau dataset de test
ht_data_i_test = np.array([[2, 3], [15, 9], [8, 5], [12, 6], [4, 4], [7,
6], [11, 7], [9, 8], [6, 5], [10, 6]])
ht_data_c_test = np.array([[1, 14], [18, 15], [20, 17], [13, 19], [22, 23],
[16, 24], [24, 26], [21, 30], [17, 28], [19, 22]])

# Concaténer les données de test avec les données d'entraînement
X_test = np.concatenate((ht_data_c_test, ht_data_i_test))
y_test = np.concatenate((np.ones(len(ht_data_c_test)),
np.zeros(len(ht_data_i_test))))

# Ajouter une colonne de 1 pour le biais
X_test = np.hstack((np.ones((X_test.shape[0], 1)), X_test))

# Prédiction des étiquettes pour les données de test
y_pred_test = np.round(sigmoid(np.dot(X_test, theta)))
```

```
# Calcul de la matrice de confusion pour le dataset de test
confusion_test = confusion_matrix(y_test, y_pred_test)
print("Matrice test de confusion :\n", confusion_test)
```

À l'exécution du code la matrice de confusion pour les deux tableaux est bien différentes mais avec les données de test la matrice ne change pas : (Rappel ici le learning_rate est de 0.0002 et epochs=200000) :

Log loss après l'entraînement: 0.4054772779574016 Matrice de confusion : [[9 1] [0 8]] Matrice test de confusion : [[10 0] [0 10]]

Par curiosité j'ai essayé avec un learning_rate de 0.2 et epochs=1000, la matrice test de confusion est bonne :

Coefficients estimés : [0.97112763 -0.02666468 0.05104627] Log loss après l'entraînement : 0.7613784137993476 Matrice de confusion : [[4 6] [3 5]] Matrice test de confusion : [[0 10] [0 10]]

Le résultat ici est logique étant donné que le modèle n'est pas suffisamment entrainé lors de la mise en prod le résultat n'est pas du tout bon.

d) Vérification de la qualité de la classification dependant de la convergence du gradient :

Grâce au test précédent on s'aperçoit bien que plus la convergence est meilleur plus la classification sera précise et juste.

e) Taux de mesure mal classés :

Ajout de ce bout de code :

```
faux_positif = confusion[0, 1]
vrai_negatif = confusion[1, 1]
taux_faux_positifs = faux_positif / (faux_positif + vrai_negatif)

print("Taux de mesures classées inconfortables à tort :",
taux_faux_positifs)
```

En essayant avec un learning_rate=0.002 et epochs=200000 j'obtiens un 11% d'erreur, logique étant donné que 1 points a mal été classé sur les 10 à classer.

4 Classification par réseau de neurones

a) Modification pour visualiser la segmentation du plan entre les deux classes chatGPT m'a bien aidé pour cette partie :

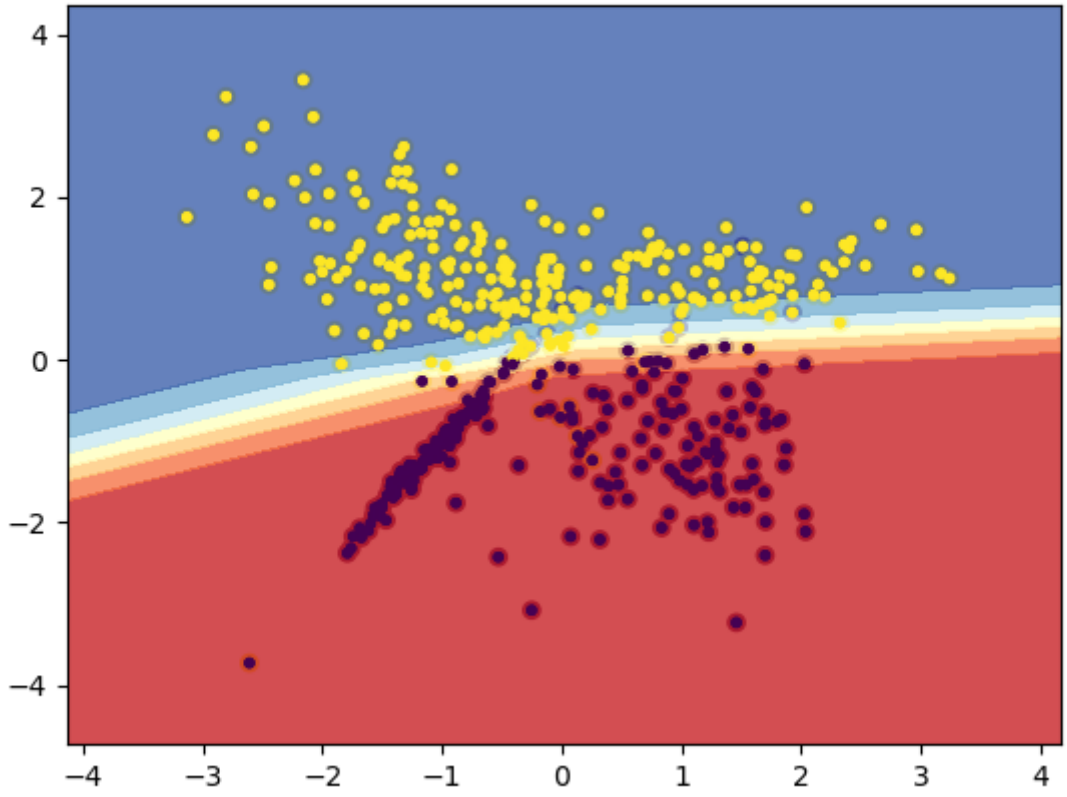
```
# Generate a grid of points to represent the entire plane
x_min, x_max = X.min() - 1, X.max() + 1
y_min, y_max = Y.min() - 1, Y.max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, 0.1),
                     np.arange(y_min, y_max, 0.1))
grid_data = np.c_[xx.ravel(), yy.ravel()]
```

```
# Predict the labels for the grid points
grid_predictions = model.predict(grid_data)

# Reshape the predictions and create a contour plot
grid_predictions = grid_predictions.reshape(xx.shape)
plt.contourf(xx, yy, grid_predictions, alpha=0.8, cmap=plt.cm.RdYlBu)

# Plot the test data points
plt.scatter(X_test, Y_test, marker=".", c=np.round(predictions))
```

Résultat de la commande :



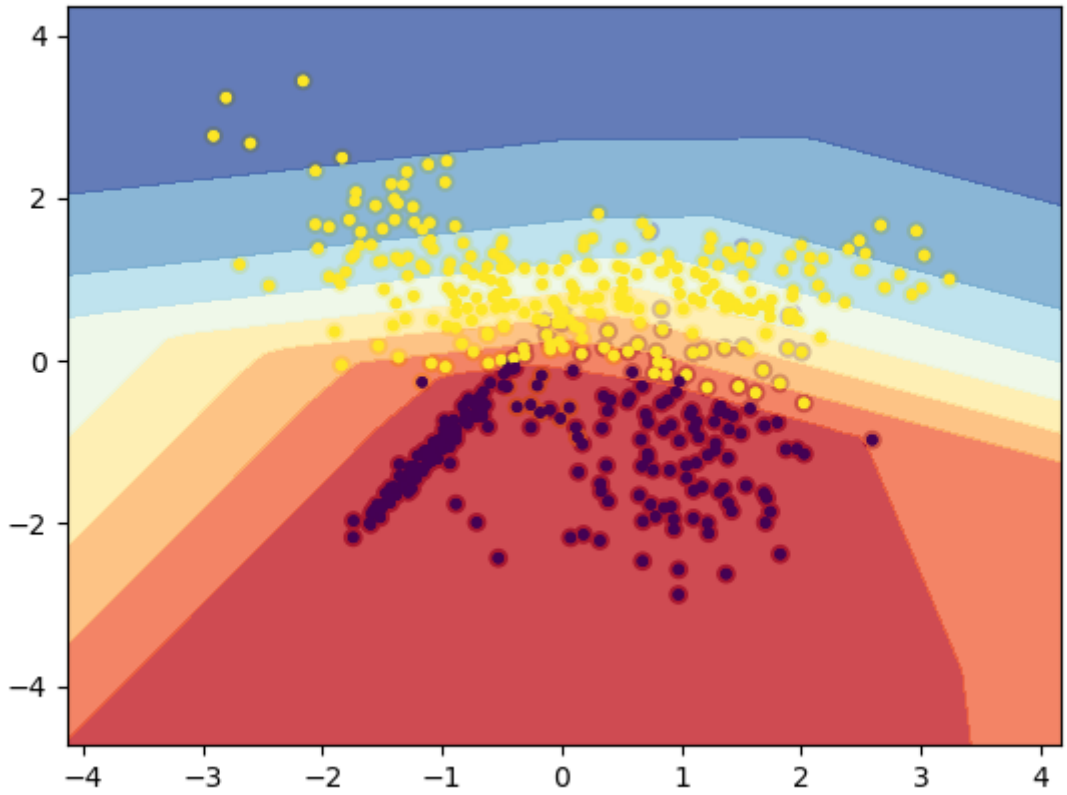
Les deux classes dont bien séparé en suivant un code couleurs :

Rouge et bleu, il n'y a uniquement qu'une classe. Les autres dérivées de couleurs signifient qu'il y à un potentiellement un point de l'autre classe.

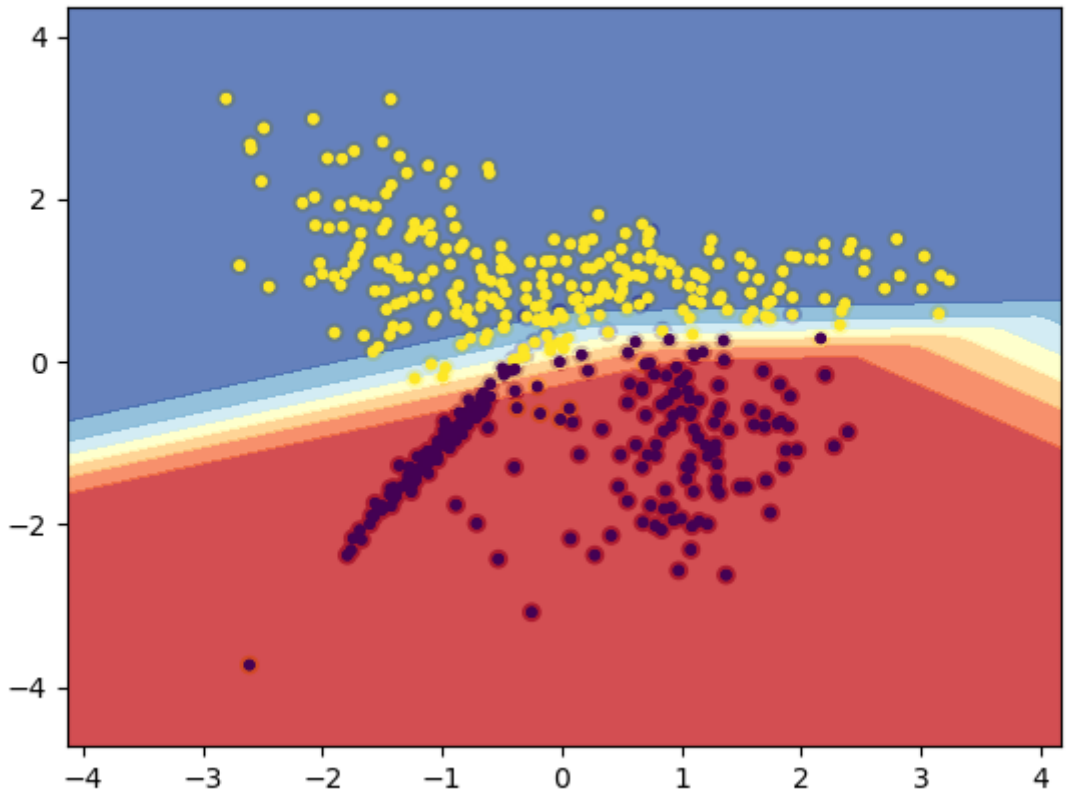
b) Pas réussie à avoir des résultats cohérents ou logique.

c) Modifiaction du nombre d'epoch :

30 : Test accuracy: 0.9020000100135803



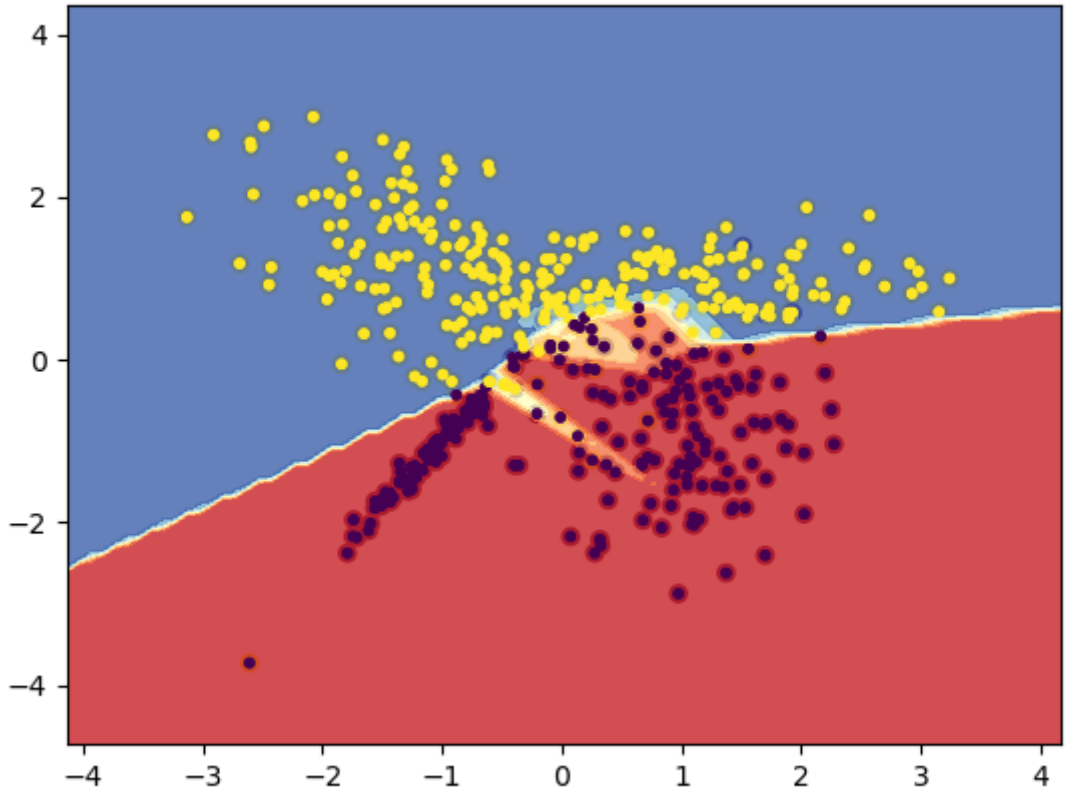
3000: Test accuracy: 0.9440000057220459



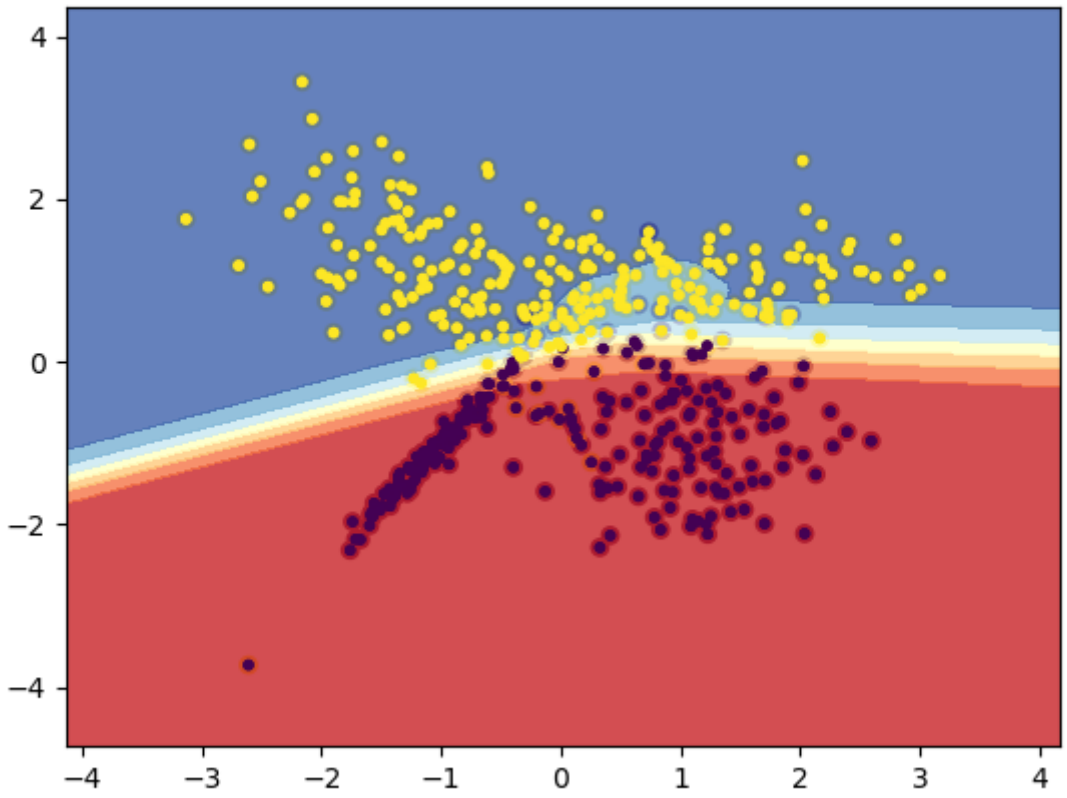
Entre les deux tests une différence est visible graphiquement, mais le "test accuracy" reste quand même très positif dans les deux cas.

d) En rajoutant des couches au modèle en rajoutant `tf.keras.layers.Dense(8, activation='relu')`, mais les résultats me semble peu différents :

Ici epochs = 3000 :



et ici epochs = 30 :



En comparant les résultats, on s'aperçoit rapidement que les couloirs de couleurs sont bien plus fine que les testes effectuer précédemment. J'en déduis donc qu'en augmentant les couches, on augmente la fiabilité et l'apprentissage du modèle au détriment de l'utilisation CPU.