# Parts C and D

Collaborators:

Talia Seada – 211551601, assigned to the morning group.

Yehudit Brickner – 328601018, assigned to the morning group.

Tavor Levine – 315208439, assigned to the evening group.

Noa Nussbaum – 206664278, assigned to the morning group.

Table of Contents:

# Part C

## Tables

Data set A results tables:

First:

**Accuracy score:** 99.8%

**Confusion matrix:**

| True Positive | False Positive | True Negative | False Negative |
|---|---|---|---|
| 51.0% | 0.20% | 48.80% | 0.0% |

**Precision:** 100%

**Recall:** 100%

**F1-score:** 100%

Second:

**Accuracy score:** 100%

**Confusion matrix:**

| True Positive | False Positive | True Negative | False Negative |
|---|---|---|---|
| 48.6% | 0.0% | 51.40% | 0.0% |

**Precision:** 100%

**Recall:** 100%

**F1-score:** 100%

Data set B results tables:

First:

**Accuracy score:** 98.2%

**Confusion matrix:**

| True Positive | False Positive | True Negative | False Negative |
|---|---|---|---|
| 86.6% | 1.40% | 11.60% | 0.4% |

**Precision:** 97%

**Recall:** 94%

**F1-score:** 96%

Second:

**Accuracy score:** 98.3%

**Confusion matrix:**

| True Positive | False Positive | True Negative | False Negative |
|---|---|---|---|
| 85.4% | 0.4% | 12.90% | 1.3% |

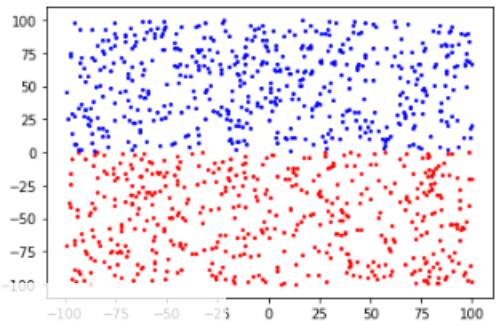
**Precision:** 95%

**Recall:** 98%

**F1-score:** 96%

# Discussions and Illustrations

plotting train



Data set A:

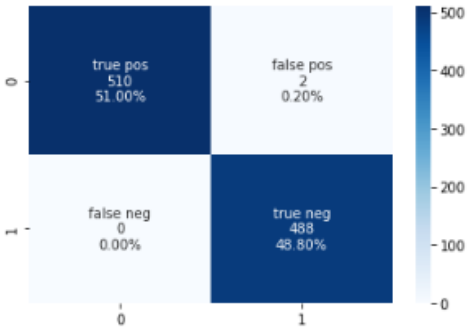
We tried running a few NN models and got the best result with a network with (4,4) as the hidden layer. Below are the results from the 2 tests we ran.

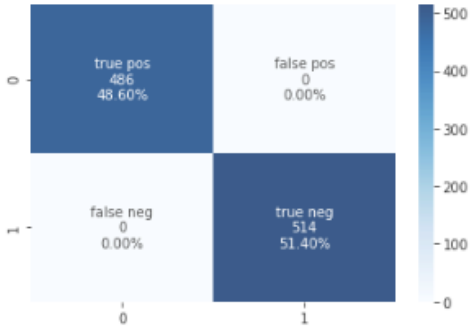
first test
0.998

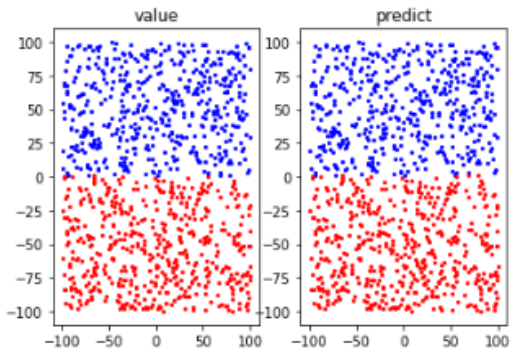
| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 1.00 | 1.00 | 1.00 | 510 |
| 1 | 1.00 | 1.00 | 1.00 | 490 |
| accuracy | | | 1.00 | 1000 |
| macro avg | 1.00 | 1.00 | 1.00 | 1000 |
| weighted avg | 1.00 | 1.00 | 1.00 | 1000 |

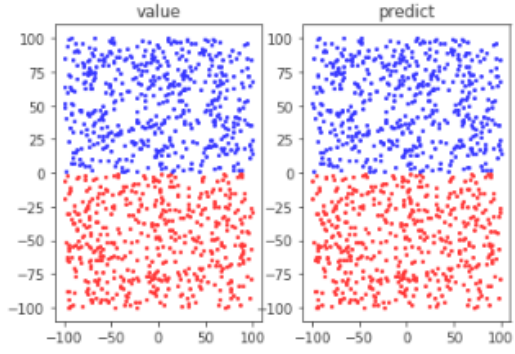
second test
1.0

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 1.00 | 1.00 | 1.00 | 486 |
| 1 | 1.00 | 1.00 | 1.00 | 514 |
| accuracy | | | 1.00 | 1000 |
| macro avg | 1.00 | 1.00 | 1.00 | 1000 |
| weighted avg | 1.00 | 1.00 | 1.00 | 1000 |

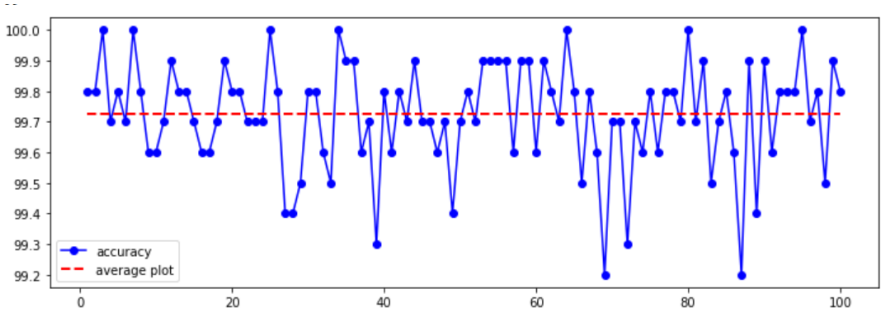



We ran the NN 100 times and here are our the results of the accuracy score:
min=99.2 max=99.72 avg=100.0

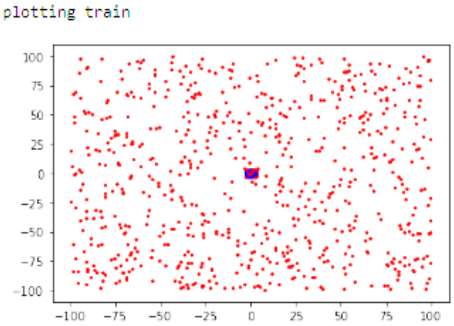


Data set B:

We decided to over sample our data in the area [-3,3] for the x and y axis. We did 70% of the data in the whole range and 30% of the data in the range [-3,3]

We did that so that we would have enough data so that the NN would work well and not over fit the data.

We tried running a few NN models and got the best result with a network with (5,10,10,4) as the hidden layer.

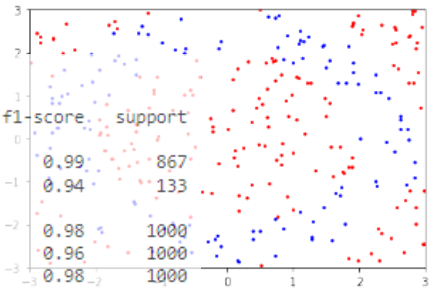


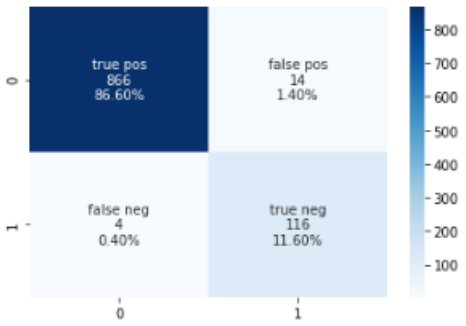

first test
0.982

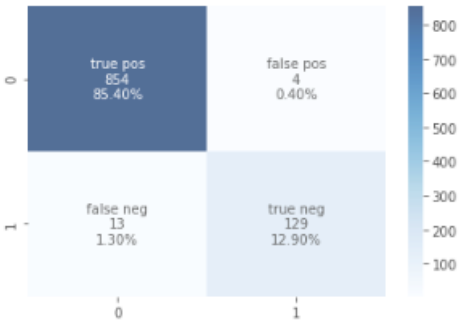
| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 0.98 | 1.00 | 0.99 | 870 |
| 1 | 0.97 | 0.89 | 0.93 | 130 |
| accuracy | | | 0.98 | 1000 |
| macro avg | 0.98 | 0.94 | 0.96 | 1000 |
| weighted avg | 0.98 | 0.98 | 0.98 | 1000 |

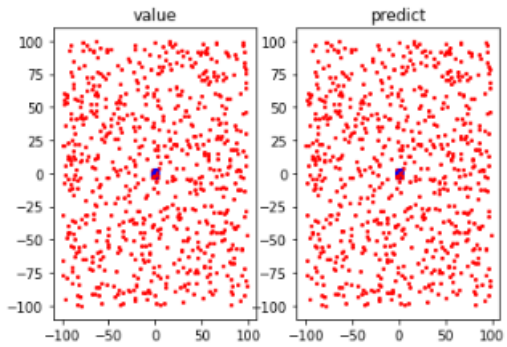
second test
0.983

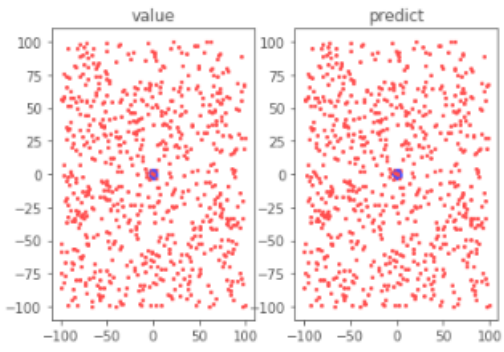
| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 1.00 | 0.99 | 0.99 | 867 |
| 1 | 0.91 | 0.97 | 0.94 | 133 |
| accuracy | | | 0.98 | 1000 |
| macro avg | 0.95 | 0.98 | 0.96 | 1000 |
| weighted avg | 0.98 | 0.98 | 0.98 | 1000 |



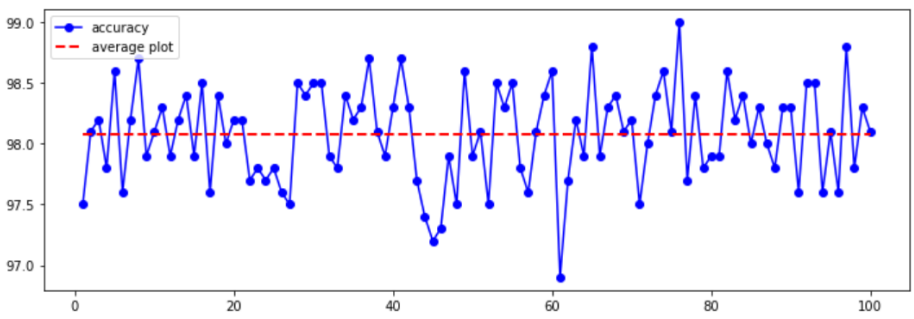


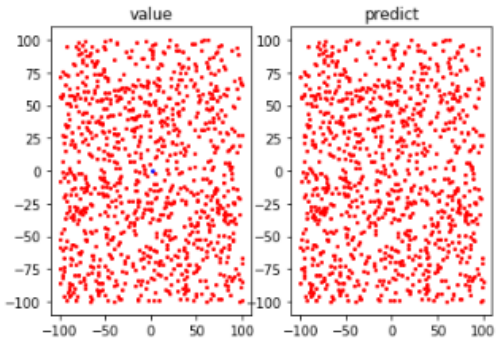




We ran the NN 100 times and here are our results:
min=96.9 max=99.9 avg=98.1

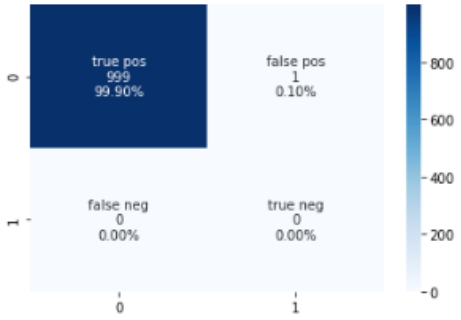


In the first part of the project we ran the adaline on the data in the given range without over sampling. Here are our results running a NN with (5,10,10,5) as hidden layers. You will see that the model is trained to find 1 point in the middle, so unless the point is very close to that point it will not find it and classify it wrong.

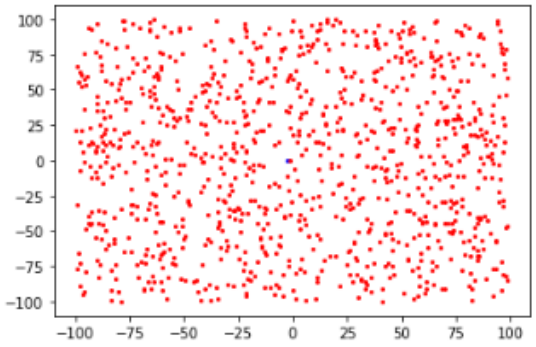
```
0.999
```




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 1.00 | 1.00 | 1.00 | 999 |
| 1 | 0.00 | 0.00 | 0.00 | 1 |
| accuracy | | | 1.00 | 1000 |
| macro avg | 0.50 | 0.50 | 0.50 | 1000 |
| weighted avg | 1.00 | 1.00 | 1.00 | 1000 |

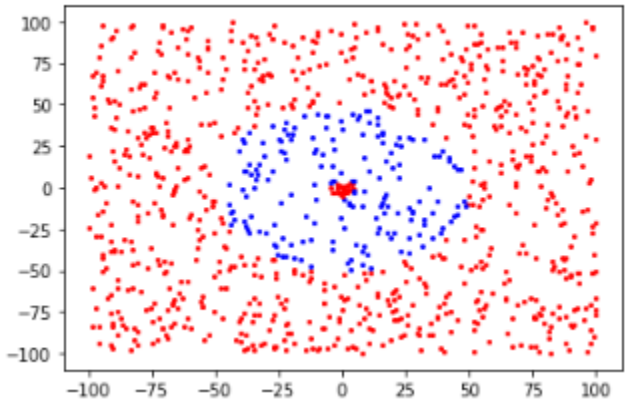



```
plotting train
```



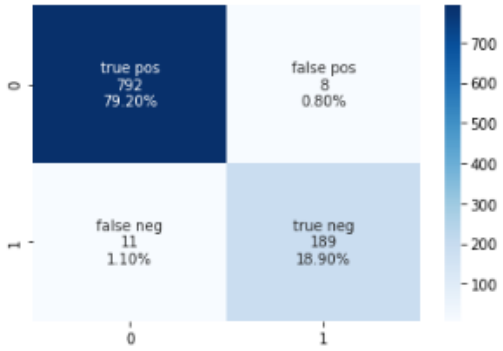
We also wanted to see how well we could get a NN to classify the data if the “bagels” were between radiuses 5 and 50 with a little over sampling in the middle [-5,5] for the x and y axis. We did 99.7% of the data regular and 0.03% of the data in the over sampled range.

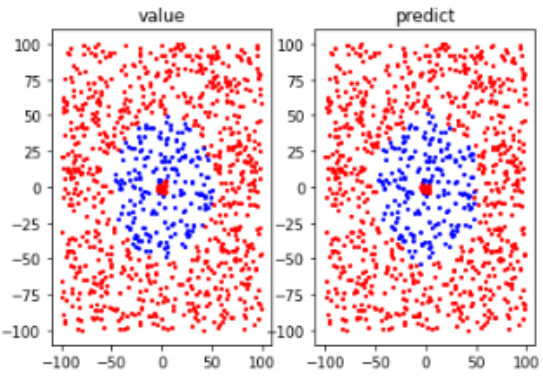
plotting train



first test
0.981

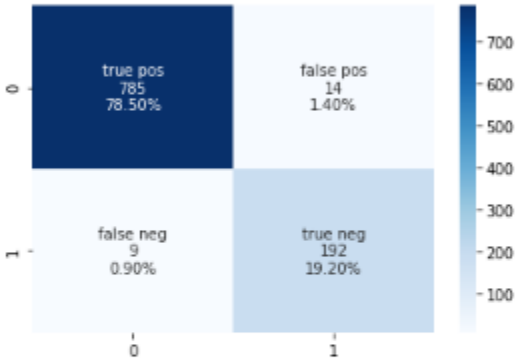
| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 0.99 | 0.99 | 0.99 | 803 |
| 1 | 0.94 | 0.96 | 0.95 | 197 |
| accuracy | | | 0.98 | 1000 |
| macro avg | 0.97 | 0.97 | 0.97 | 1000 |
| weighted avg | 0.98 | 0.98 | 0.98 | 1000 |

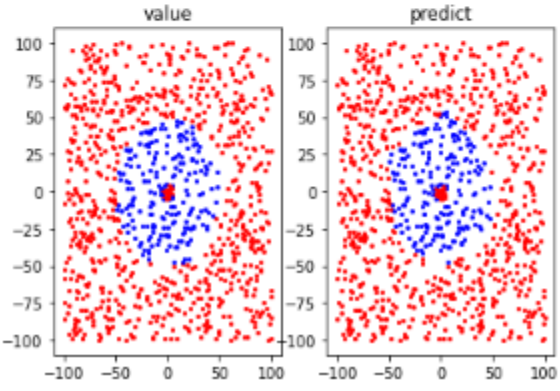



second test
0.977

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 0.98 | 0.99 | 0.99 | 794 |
| 1 | 0.96 | 0.93 | 0.94 | 206 |
| accuracy | | | 0.98 | 1000 |
| macro avg | 0.97 | 0.96 | 0.96 | 1000 |
| weighted avg | 0.98 | 0.98 | 0.98 | 1000 |



# Code

In [34]:
```
#libraries
import numpy as np
import pandas as pd
import math
import random

#preproccesing
from sklearn.metrics import classification_report,f1_score
from sklearn.metrics import confusion_matrix
from sklearn.neural_network import MLPClassifier

# visulization
import matplotlib.pyplot as plt
import seaborn as sns

import warnings
warnings.filterwarnings('ignore')
```

## Part C

neural network using sklearn

In [35]:
```
class NeuralNetwork:
    def __init__(self, learning_rate, train,hidden_layers):
        self.train=train
        self.clf=MLPClassifier(solver='lbfgs', alpha=learning_rate,hidden_layer_sizes=hidden_layers, random_state=1)

    def fit(self):
        train_x=self.train[["x","y"]]
        train_y=self.train[["value"]]
        self.clf.fit(train_x,train_y)

    def predict(self, test):
        test_x=test[["x","y"]]
        predicted=self.clf.predict(test_x)
        test['pred']=predicted
        return predicted

    def score(self, test, predicted):
        i=0
        count=0
        for index, row in test.iterrows():
            if row["value"]==predicted[i]:
                count +=1
            i+=1
        return round(count/i, 4)
    import math


    def lookin(self,data):
        listx=[]
        listy=[]
        for index, row in data.iterrows():
            listx.append(row["x"])
            listy.append(row["y"])
        l=[listx,listy]
        for i in range(len(self.clf.coefs_)):
            if i< len(self.clf.coefs_)-1:
                print("hidden layer ", i+1)
            else:
                print("output")
            datac=data.copy()
            l=(self.lookinlayer(datac,l,self.clf.coefs_[i],self.clf.intercepts_[i]))
```

```
    def lookinlayer(self,data,lastlayers,weight,bias):
        count = 0
        rlist=[]
        # for each row we use the activation formula with the weights and bias we returned
        # in the fit function to predict on the test data set
        for k in range(len(bias)):
            print("neuron ", k+1)
            pred1 = []
            predictionlist=[]
            for j in range(len(data)):
                prediction = bias[k]
                for i in range(len(lastlayers)):
                    prediction += (lastlayers[i][j] * weight[i][k])
#                 predictionlist.append(prediction)
                if prediction > 0:
                    predictionlist.append(prediction)
                    prediction = 1
                else:
                    predictionlist.append(0)
                    prediction = -1
                pred1.append(prediction)
        # now add the prediction list to the data set in order to make comparison
            rlist.append(predictionlist)
            data['pred']=pred1
            plotting_test(data)
        return rlist

    # same as lookinlayer but without the plotting
    def lookinlayer1(self,data,lastlayers,weight,bias):
        count = 0
        rlist=[]
        # for each row we use the activation formula with the weights and bias we returned
        # in the fit function to predict on the test data set
        for k in range(len(bias)):
            pred1 = []
            predictionlist=[]
            for j in range(len(data)):
                prediction=bias[k]
                for i in range(len(lastlayers)):
                    prediction += (lastlayers[i][j] * weight[i][k])
#                 predictionlist.append(prediction)
                if prediction > 0:
                    predictionlist.append(prediction)
                    prediction = 1
                else:
                    predictionlist.append(0)
                    prediction = -1
                pred1.append(prediction)
        # now add the prediction list to the data set in order to make comparison
            rlist.append(predictionlist)
            data['pred']=pred1
        return rlist
```

```
    def getLastHiddenLayerInfo(self,data):
            listx=[]
            listy=[]
            val=[]

#             for index, row in self.train.iterrows():
            for index, row in data.iterrows():
                listx.append(row["x"])
                listy.append(row["y"])
                val.append(row["value"])
            l=[listx,listy]
            for i in range(len(self.clf.coefs_)-1):
                data=self.train.copy()
                l=(self.lookinlayer1(data,l,self.clf.coefs_[i],self.clf.intercepts_[i]))
            l.append(val)
            return l
```

In [36]:

```
# this function builds the data set for part A of the assighnment
def build_data_partA(i):
    x = []
    y = []
    value = []
    random.seed(i)
    for i in range(1000):
        # generate two random numbers between -10000 to 10000
        randX = random.randint(-10000, 10000)
        randY = random.randint(-10000, 10000)
        x.append(randX / 100)
        y.append(randY / 100)
        # for part A if y > 1 then the value is 1
        if y[i] > 1:
            value.append(1)
        # else the value is -1
        else:
            value.append(-1)

    # make the data frame
    end = {'x': x, 'y': y, 'value': value}
    df = pd.DataFrame(data=end, columns=['x', 'y', 'value'])
    return df
```

In [37]:

```
# this function builds the data set for part B of the assighnment
def build_data_partB(i):
    x = []
    y = []
    value = []
    random.seed(i)
    for i in range(700):
        # generate two random numbers between -10000 to 10000
        randX = random.randint(-10000, 10000)
        randY = random.randint(-10000, 10000)
        x.append(randX / 100)
        y.append(randY / 100)
        # for part A if (4 <= y^2 + x^2 <= 9) then the value is 1
        if 4 <= (y[i] ** 2 + x[i] ** 2) <= 9:
#         if 25<=(y[i]**2+x[i]**2)>=2500:
            value.append(1)
        # else the value is -1
        else:
            value.append(-1)
    for i in range(700,1000):
        # generate two random numbers between -10000 to 10000
        randX = random.randint(-300, 300)
        randY = random.randint(-300, 300)
        x.append(randX / 100)
        y.append(randY / 100)
        # for part A if (4 <= y^2 + x^2 <= 9) then the value is 1
        if 4 <= (y[i] ** 2 + x[i] ** 2) <= 9:
#         if 25<=(y[i]**2+x[i]**2)>=2500:
            value.append(1)
        # else the value is -1
        else:
            value.append(-1)

    # make the data frame
    end = {'x': x, 'y': y, 'value': value}
    df = pd.DataFrame(data=end, columns=['x', 'y', 'value'])
    return df
```

In [38]:

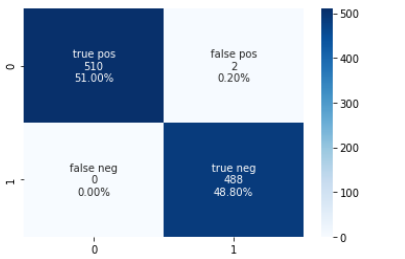
```
# this function plots the values of the actual values of the data compared to the prediction values we predicted
def plotting_test(test):
    f, ax = plt.subplots(1, 2)
    ax[0].set_title("value")
    ax[1].set_title("predict")

    for index, row in test.iterrows():
        if row['value'] == 1:
            ax[0].plot(row['x'], row['y'], markersize=2, marker="o", color="blue")
        else:
            ax[0].plot(row['x'], row['y'], markersize=2, marker="o", color="red")

        if row['pred'] == 1:
             ax[1].plot(row['x'], row['y'], markersize=2, marker="o", color="blue")
        else:
            ax[1].plot(row['x'], row['y'], markersize=2, marker="o", color="red")
    plt.show()

def plotting_train1(train):
    for index, row in train.iterrows():
        if row['value'] == 1:
            plt.plot(row['x'], row['y'], markersize=2, marker="o", color="blue")
        else:
            plt.plot(row['x'], row['y'], markersize=2, marker="o", color="red")
    plt.show()
    for index, row in train.iterrows():
        if row['value'] == 1:
            plt.plot(row['x'], row['y'], markersize=2, marker="o", color="blue")
        else:
            plt.plot(row['x'], row['y'], markersize=2, marker="o", color="red")
    plt.xlim([-3, 3])
    plt.ylim([-3, 3])
    plt.show()

def plotting_train2(train):
    for index, row in train.iterrows():
        if row['value'] == 1:
            plt.plot(row['x'], row['y'], markersize=2, marker="o", color="blue")
        else:
            plt.plot(row['x'], row['y'], markersize=2, marker="o", color="red")
    plt.show()
```

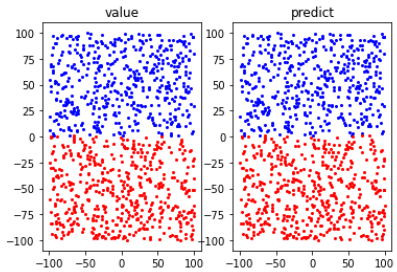
In [39]:
```
# this function plots the confussion matrix
def confussion_matrix (cf_matrix):
    group_names = ['true pos', 'false pos', 'false neg', 'true neg']
    group_counts = ["{0:0.0f}".format(value) for value in cf_matrix.flatten()]
    group_percentages = ["{0:.2%}".format(value) for value in cf_matrix.flatten()/np.sum(cf_matrix)]
    labels = [f"{v1}\n{v2}\n{v3}" for v1, v2, v3 in zip(group_names,group_counts,group_percentages)]
    labels = np.asarray(labels).reshape(2,2)
    sns.heatmap(cf_matrix, annot=labels, fmt='', cmap='Blues')
```

## Part C main

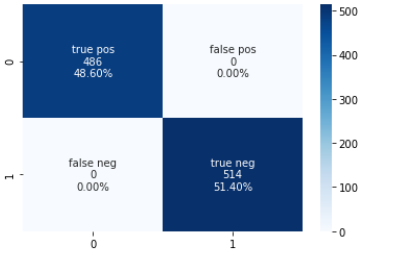
In [40]:
```
train = build_data_partA(1)
net1=NeuralNetwork(0.1,train,(4,4))
net1.fit()
print("plotting train")
plotting_train2(train)
# net1.lookin(train)

print("first test")
first_test = build_data_partA(9)
test_y=first_test[["value"]]
pred1=net1.predict(first_test)
print(net1.score(first_test,pred1))
con_mat1 = confusion_matrix(pred1,test_y)
confussion_matrix(con_mat1)
print(classification_report(test_y, pred1))
plotting_test(first_test)



print("second test")
second_test = build_data_partA(8)
test_y=second_test[["value"]]
pred2=net1.predict(second_test)
print(net1.score(second_test,pred2))
con_mat2 = confusion_matrix(pred2,test_y)
confussion_matrix(con_mat2)
print(classification_report(test_y, pred2))
plotting_test(second_test)
```

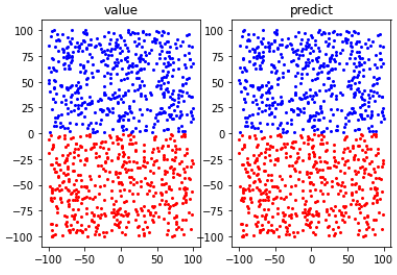
```
first test
0.998
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00       510
           1       1.00      1.00      1.00       490

    accuracy                           1.00      1000
   macro avg       1.00      1.00      1.00      1000
weighted avg       1.00      1.00      1.00      1000
```




```
second test
1.0
              precision    recall  f1-score   support

          -1       1.00      1.00      1.00       486
           1       1.00      1.00      1.00       514

    accuracy                           1.00      1000
   macro avg       1.00      1.00      1.00      1000
weighted avg       1.00      1.00      1.00      1000
```



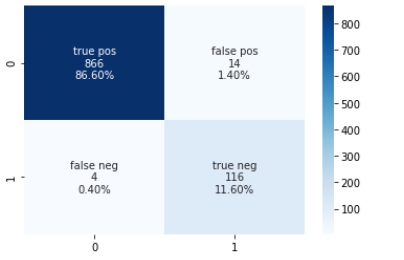



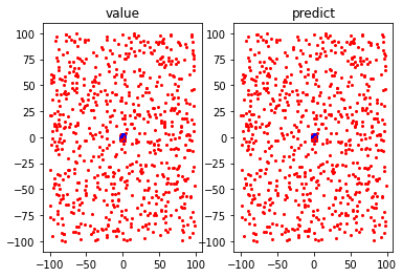
part b:

In [44]:
```
# using relu
# (5,2) ~0.935
# (5,10,10,5) ~ 0.985
# (5,10,10,10,5) ~0.975
# (10,20,10,5) ~0.975
# (10,5) ~0.98

# using sigmoid
# (5,2) ~0.935
# (10,5) ~0.98
```

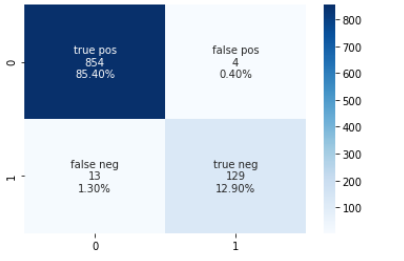
In [45]:
```
trainb = build_data_partB(9)
net2=NeuralNetwork(0.01,trainb,(5,10,10,4))
net2.fit()
print("plotting train")
plotting_train1(trainb)
# net2.lookin(trainb)

print("first test")
first_testb = build_data_partB(3)
test_yb=first_testb[["value"]]
pred1=net2.predict(first_testb)
print(net2.score(first_testb,pred1))
con_mat1 = confusion_matrix(pred1,test_yb)
confussion_matrix(con_mat1)
print(classification_report(test_yb, pred1))
plotting_test(first_testb)


print("second test")
second_testb = build_data_partB(7)
test_yb=second_testb[["value"]]
pred2=net2.predict(second_testb)
print(net2.score(second_testb,pred2))
con_mat2 = confusion_matrix(pred2,test_yb)
confussion_matrix(con_mat2)
print(classification_report(test_yb, pred2))
plotting_test(second_testb)
```

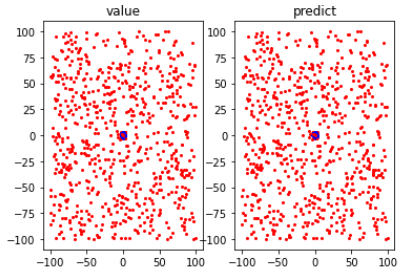
```
first test
0.982
              precision    recall  f1-score   support

          -1       0.98      1.00      0.99       870
           1       0.97      0.89      0.93       130

    accuracy                           0.98      1000
   macro avg       0.98      0.94      0.96      1000
weighted avg       0.98      0.98      0.98      1000
```




```
second test
0.983
              precision    recall  f1-score   support

          -1       1.00      0.99      0.99       867
           1       0.91      0.97      0.94       133

    accuracy                           0.98      1000
   macro avg       0.95      0.98      0.96      1000
weighted avg       0.98      0.98      0.98      1000
```

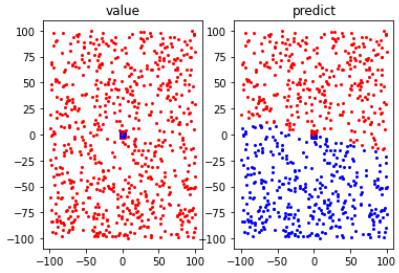


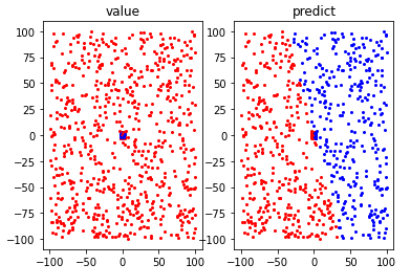
## Look in the hidden layers for each neuron:

```
In [46]: net2.lookin(trainb)
```

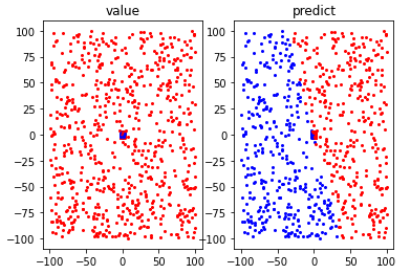
```
hidden layer  1
neuron  1
```



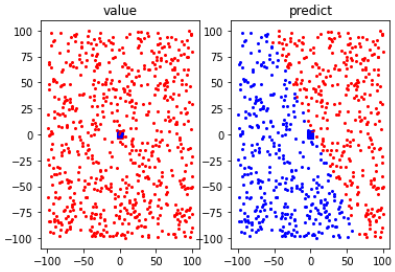

```
neuron  2
```



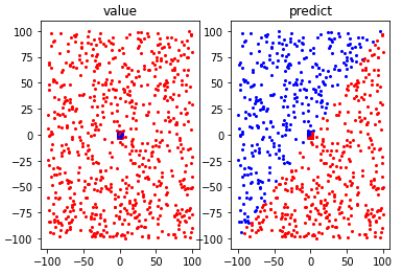

```
neuron  3
```




```
neuron  4
```

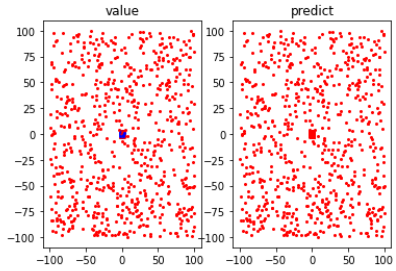



```
neuron  5
```

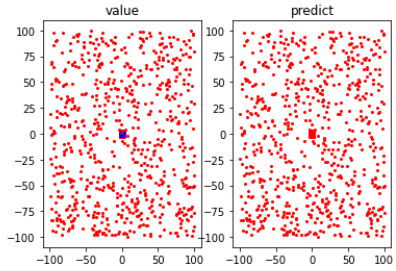


hidden layer 2

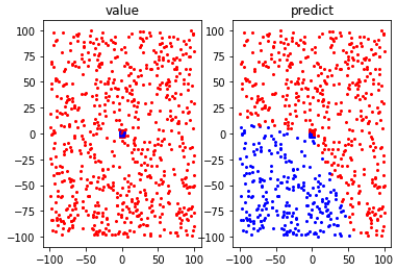
neuron 1



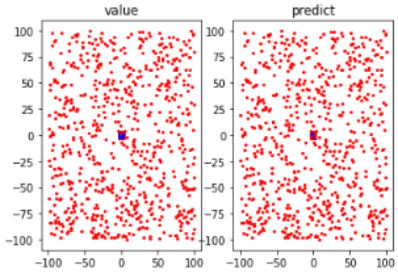

neuron 2



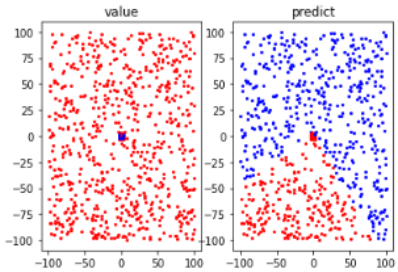

neuron 3



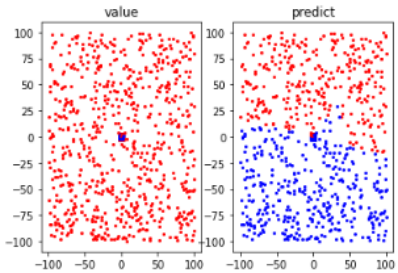

neuron 4



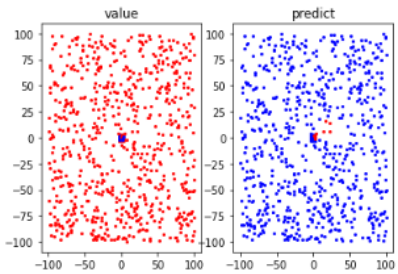

neuron 5



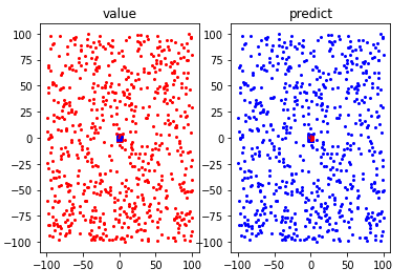

neuron 6



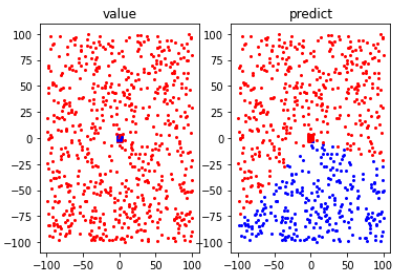

neuron 7



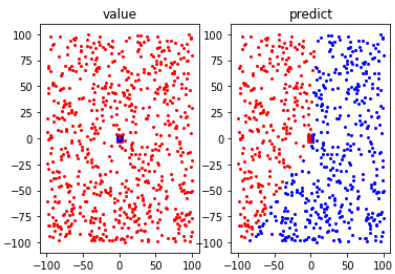

neuron 8




neuron 9

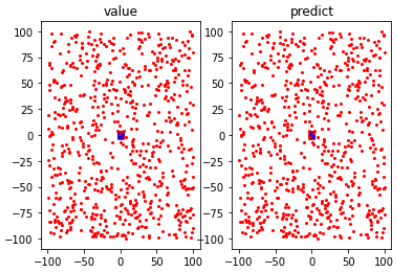



neuron 10

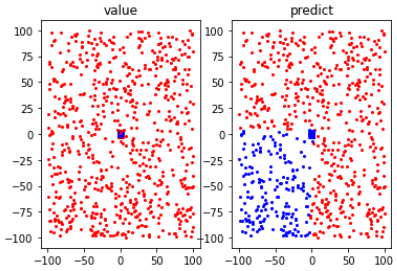


hidden layer 3

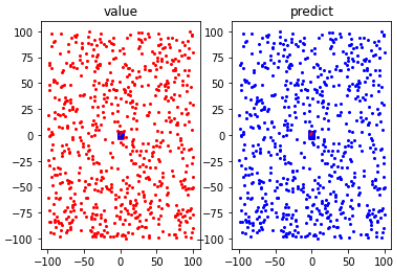
neuron 1



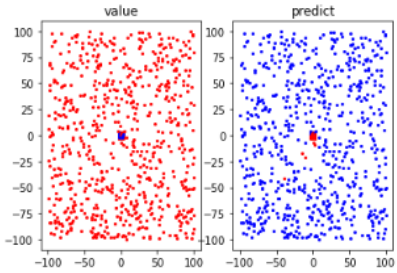

neuron 2



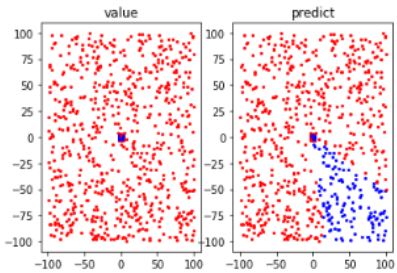

neuron 3



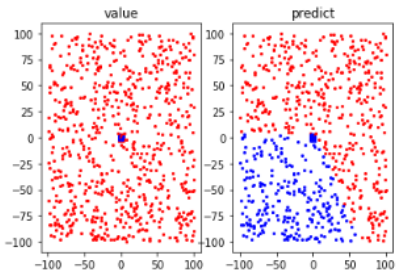

neuron 4



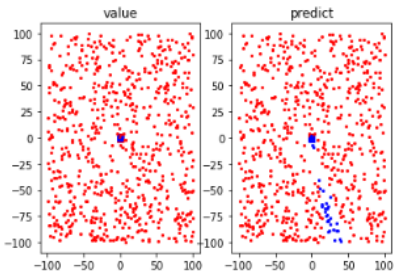

neuron 5



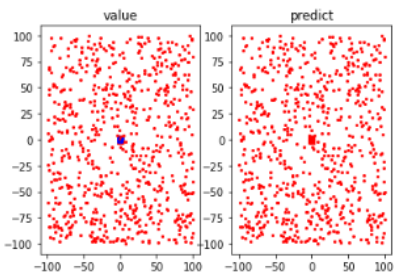

neuron 6



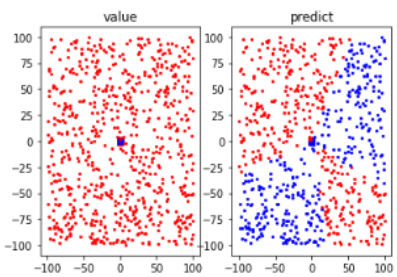

neuron 8



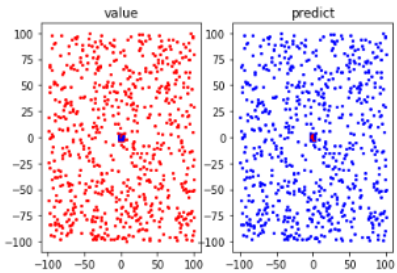

neuron 9



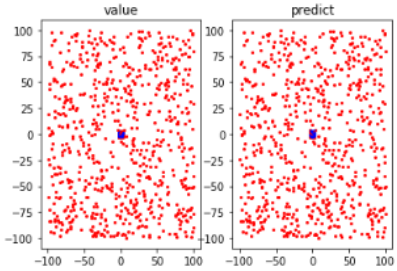

neuron 10



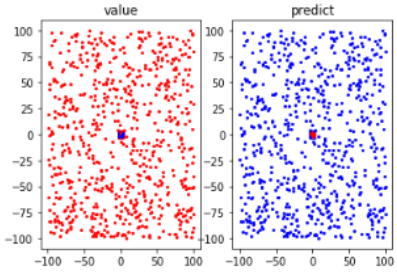

neuron 7



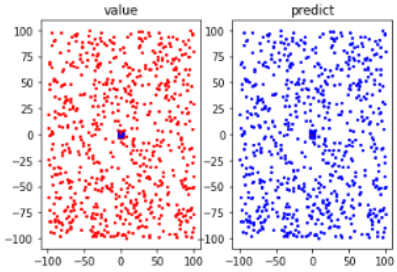
```
hidden layer  4
neuron  1
```



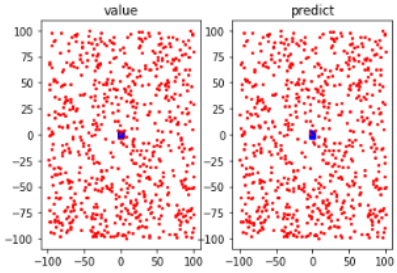

```
neuron  2
```



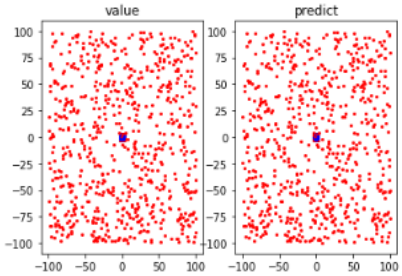

```
neuron  3
```




```
neuron  4
```




```
output
neuron  1
```



# Part D

## Table

Data set A results table:

First

**Accuracy score:** 94.4%

**Confusion matrix:**

| True Positive | False Positive | True Negative | False Negative |
|---|---|---|---|
| 51.0% | 0.60% | 48.40% | 0.0% |

**Precision:** 99%

**Recall:** 99%

**F1-score:** 99%

Second

**Accuracy score:** 98.3%

**Confusion matrix:**

| True Positive | False Positive | True Negative | False Negative |
|---|---|---|---|
| 48.60% | 0.60% | 50.80% | 0.0% |

**Precision:** 99%

**Recall:** 99%

**F1-score:** 99%

Data set B results tables:

First:

**Accuracy score:** 95.2%

**Confusion matrix:**

| True Positive | False Positive | True Negative | False Negative |
|---|---|---|---|
| 86.3% | 4.10% | 8.90% | 0.7% |

**Precision:** 97%

**Recall:** 94%

**F1-score:** 96%

Second:

**Accuracy score:** 95.2%

**Confusion matrix:**

| True Positive | False Positive | True Negative | False Negative |
|---|---|---|---|
| 85.5% | 3.6% | 9.7% | 1.2% |

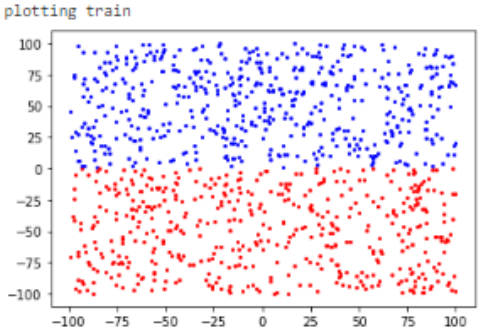
**Precision:** 96%

**Recall:** 99%

**F1-score:** 97%

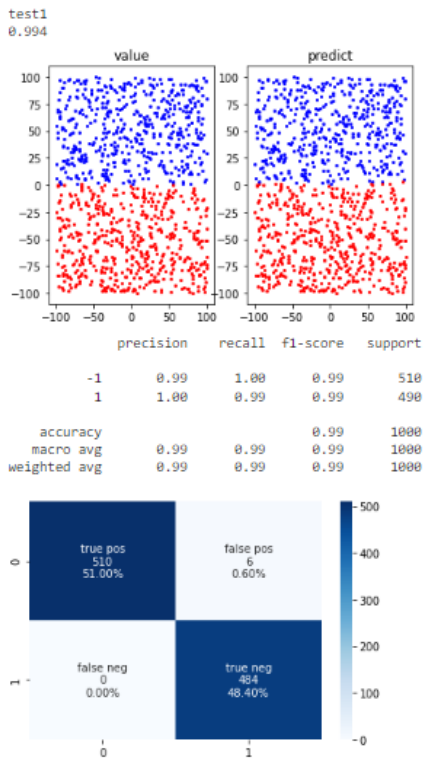
# Discussions and Illustrations

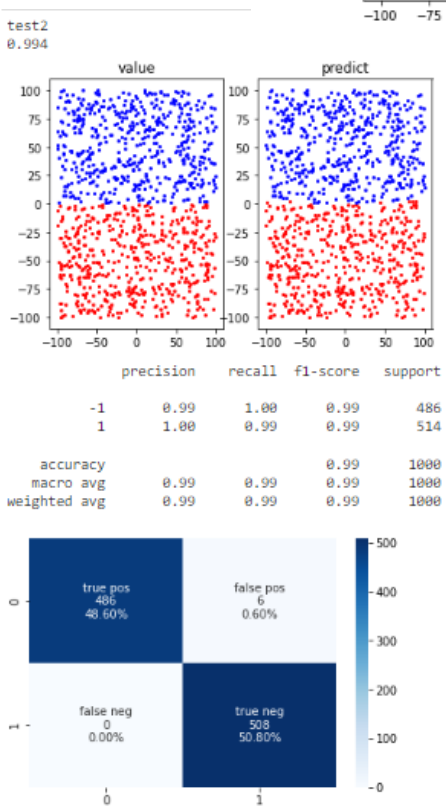
In this part of the project we had to take the last layer of the NN and use that data as input for the adaline. An adaline gives a linear separation of the data.
The reason we will get good results for both data sets is that we are giving the adaline data that has already been transformed so that when we linearly divide that data it can give us a non linear division of our original data.



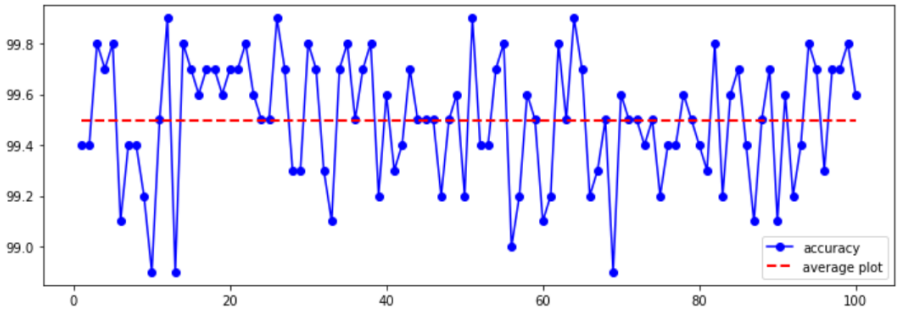

For both datasets we had to play with the learning rate, the amount of epochs and had to decide if we wanted at a certain point to lower the learning rate.

Our best results for dataset A was using learning rate 0.9, 50 epochs. And not lowering the learning rate.

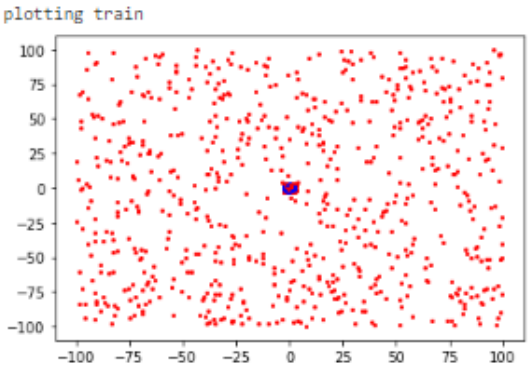


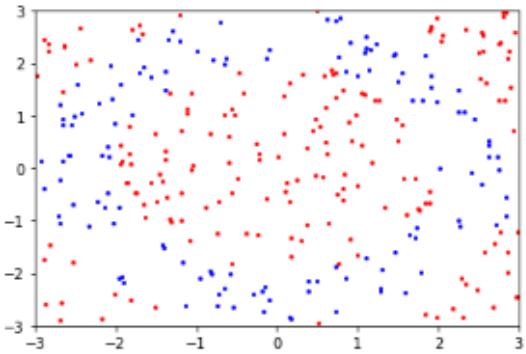




We ran the adaline 100 times and here are our results:
min=98.9  max=99.5  avg=99.9

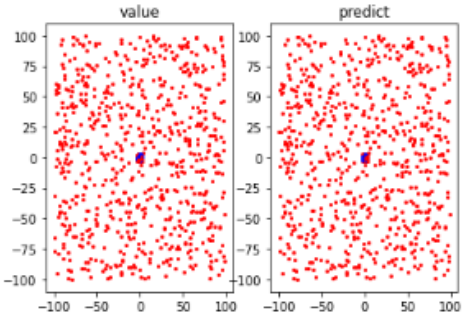


Our best results for dataset B was using learning rate 0.66, 225 epochs, and not lowering the learning rate

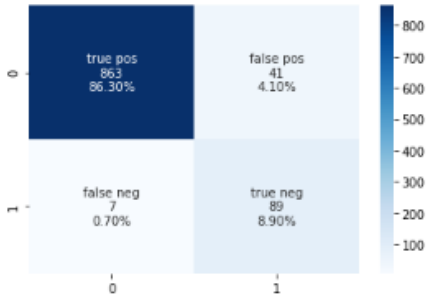


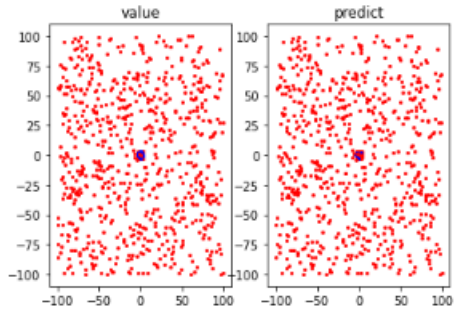
test1
0.952



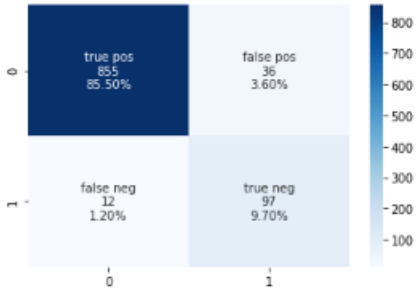

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 0.95 | 0.99 | 0.97 | 870 |
| 1 | 0.93 | 0.68 | 0.79 | 130 |
| accuracy | | | 0.95 | 1000 |
| macro avg | 0.94 | 0.84 | 0.88 | 1000 |
| weighted avg | 0.95 | 0.95 | 0.95 | 1000 |



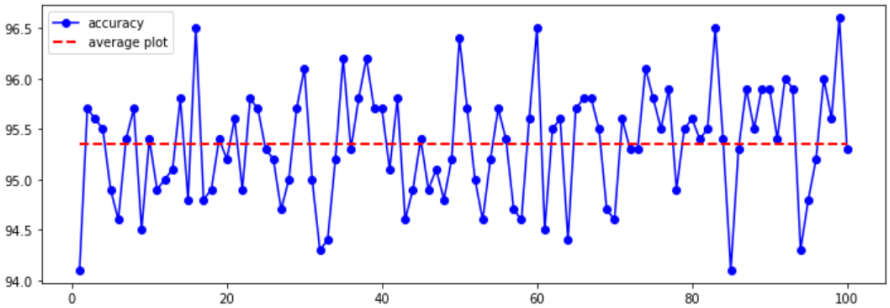

test2
0.952




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 0.96 | 0.99 | 0.97 | 867 |
| 1 | 0.89 | 0.73 | 0.80 | 133 |
| accuracy | | | 0.95 | 1000 |
| macro avg | 0.92 | 0.86 | 0.89 | 1000 |
| weighted avg | 0.95 | 0.95 | 0.95 | 1000 |

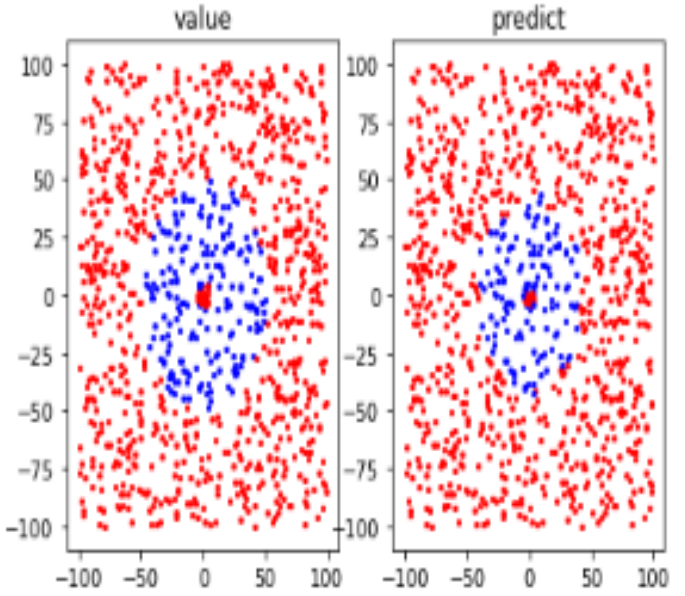



We ran the adaline 100 times and here are our results:
min=94.1  max=96.6  avg=95.35

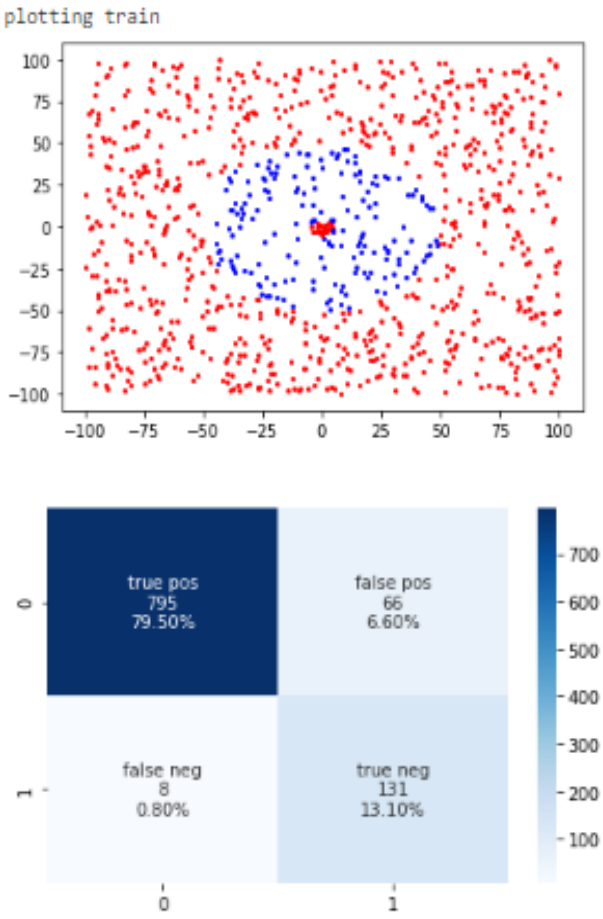


We did not make an adaline for the data that had barely 1 point in the wanted area because even if it were to classify wrong it still would  be 99% correct.

We did make an adaline for the bigger “bagel” like in part C.
Our best results were with learning rate 0.99, 250 epochs, and lowering the learning rate every 93 epochs. You can see that it is miss classifying the outer area of the bagel. We assume that with more tweaking of the adaline or the NN that is based on you could get better results, but we weren't able to figure out how to tweak it.




| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 0.92 | 0.99 | 0.96 | 803 |
| 1 | 0.94 | 0.66 | 0.78 | 197 |
| accuracy | | | 0.93 | 1000 |
| macro avg | 0.93 | 0.83 | 0.87 | 1000 |
| weighted avg | 0.93 | 0.93 | 0.92 | 1000 |



# Code

## part D

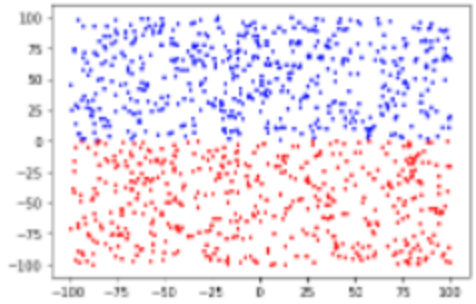
```
class Adaline:
    def __init__(self, learning_rate, train, num,epoch,change_lr):
        self.learning_rate = learning_rate
        self.train = train
        self.num=num # number of neurons in last layer
        self.epoch=epoch
        self.change_lr=change_lr

    # this function generates random small weights and bias for the Adaline algorithm
    def _weight_genarate(self,num):
        weight = []
        for i in range(num):
            random.seed(i)
            rand = random.uniform(-0.1, 0.1)
            rand = round(rand, 4)
            weight.append(rand)
        # now generate the bias
        random.seed(4)
        bias = random.uniform(0, 1)
        bias = round(bias, 4)
        return weight, bias

    # this function fits the adaline model on the training data
    def fit(self):
        ERR = []
        mse = []
        EPS = 0.0001
        # generate weights and bias
        weight, bias = self._weight_genarate(self.num)
        oldmse=1
        count=0
        while(count<self.epoch):
            if(count==self.change_lr):
                self.learning_rate=self.learning_rate/2
                self.change_lr+=self.change_lr
            ERR = []
            # for each row we fix the bias and wights in order to get the minimum error
            for index, row in self.train.iterrows():
                predicted=bias/100
                for i in range(self.num):
                    predicted += row[i]/1000 * weight[i]
                for k in range(len(weight)):
                    weight[k] = round((weight[k] + self.learning_rate * (row["value"]-predicted) * row[k]/1000),3)
                bias= round((bias + self.learning_rate * (row["value"]-predicted)),3)
                # error calculation
                error = (row["value"] - predicted) ** 2
                # if the error is small enough return
                ERR.append(error)
            mse.append(np.sum(ERR))
#             print(mse[-1])
            if len(mse) >= 2:
                # checking if the error is smaller then eps or if it hasnt changed
                if abs(mse[-1] - mse[-2]) < EPS  or abs(mse[-1] - mse[-2])==oldmse :
                    break
            # updating the old mse
            if len(mse)>=2:
                oldmse=abs(mse[-1] - mse[-2])
            count+=1
#         print(count)
        return weight, bias
```

```
#     # this function predicts on a test data and returns the number of correct predictions
    def predict(self, dftest, test, weight, bias):
        count = 0
        pred = []
        # for each row we use the activation formula with the weights and bias we returned
        # in the fit function to predict on the test data set
        for index, row in dftest.iterrows():
            predicted=bias/100
            for i in range(self.num):
                predicted += row[i]/1000 * weight[i]
#             print(predicted)
            if predicted > 0:
                predicted = 1
            else:
                predicted = -1
            pred.append(predicted)

            if predicted == row['value']:
                count += 1
        # now add the prediction list to the data set in order to make comparison
        test['pred'] = pred
        return count

    # this function caculates the acuuracy of the predictions
    def score(self, pred, dftest):
        acurr = pred / len(dftest)
        res = round(acurr, 4)
        return res
```

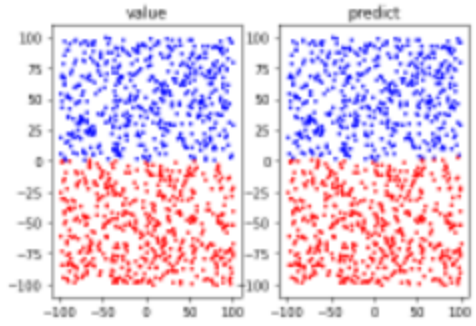
**MAIN**

```
# create train df
traina=build_data_partA(1)
listA=net1.getLastHiddenLayerInfo(traina)
dftrain=pd.DataFrame()
for i in range(len(listA)):
    if(i<len(listA)-1):
        dftrain[i]=listA[i]
    else:
        dftrain["value"]=listA[i]
print("plotting train")
plotting_train2(traina)



ada1=Adaline(0.9,dftrain,len(listA)-1,epoch=50,change_lr=-1)
weight, bias =ada1.fit()


# create test df
test = build_data_partA(9)
newtest = net1.getLastHiddenLayerInfo(test)
dftest = pd.DataFrame()
for i in range(len(newtest)):
    if(i<len(newtest)-1):
        dftest[i]=newtest[i]
    else:
        dftest["value"]=newtest[i]

print("test1")
count=ada1.predict(dftest, test, weight, bias)
print(ada1.score(count, dftest))
plotting_test(test)
print(classification_report(test["value"],test["pred"]))
con_mat1 = confusion_matrix(test["pred"],test["value"])
confussion_matrix(con_mat1)
```

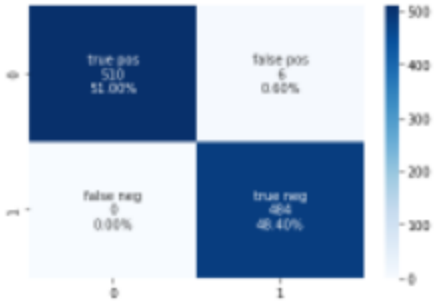
```
plotting train
```



```
test1
0.994
```

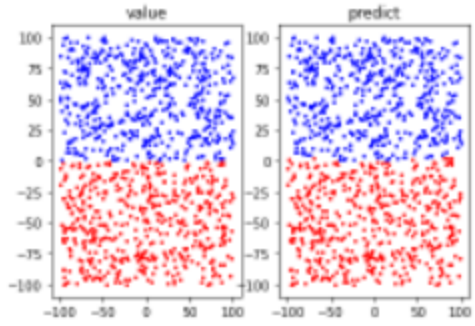



| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 0.99 | 1.00 | 0.99 | 510 |
| 1 | 1.00 | 0.99 | 0.99 | 490 |
| accuracy | | | 0.99 | 1000 |
| macro avg | 0.99 | 0.99 | 0.99 | 1000 |
| weighted avg | 0.99 | 0.99 | 0.99 | 1000 |

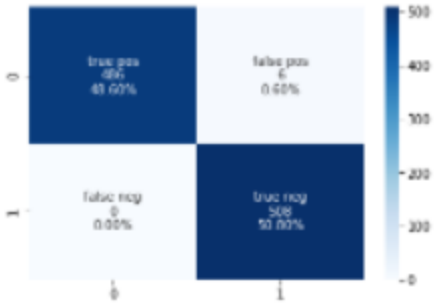


```
print("test2")
test = build_data_partA(8)
newtest = net1.getLastHiddenLayerInfo(test)
dftest = pd.DataFrame()
for i in range(len(newtest)):
    if(i<len(newtest)-1):
        dftest[i]=newtest[i]
    else:
        dftest["value"]=newtest[i]

count=ada1.predict(dftest, test, weight, bias)
print(ada1.score(count, dftest))
plotting_test(test)
print(classification_report(test["value"],test["pred"]))
con_mat1 = confusion_matrix(test["pred"],test["value"])
confussion_matrix(con_mat1)
```

```
test2
0.994
```

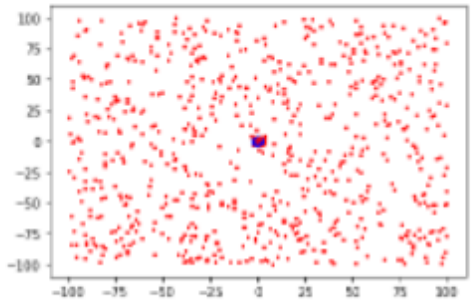


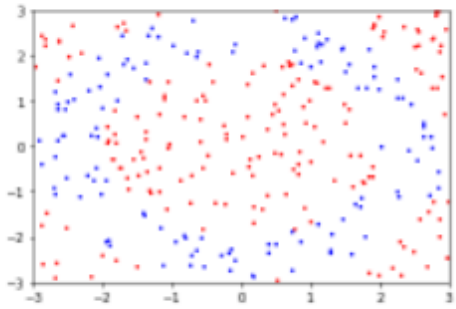

```
              precision    recall  f1-score   support

          -1       0.99      1.00      0.99       486
           1       1.00      0.99      0.99       514

    accuracy                           0.99      1000
   macro avg       0.99      0.99      0.99      1000
weighted avg       0.99      0.99      0.99      1000
```



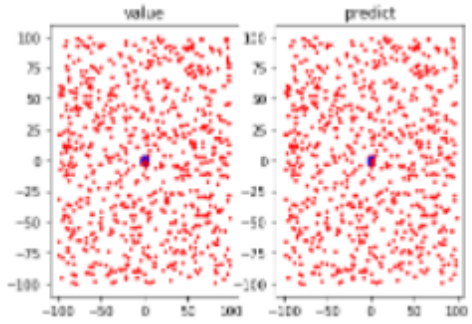
```
trainb=build_data_partB(9)
listA=net2.getLastHiddenLayerInfo(trainb)
dftrain=pd.DataFrame()
for i in range(len(listA)):
    if(i<len(listA)-1):
        dftrain[i]=listA[i]
    else:
        dftrain["value"]=listA[i]
print("plotting train")
plotting_train1(trainb)

adab=Adaline(0.66, dftrain, len(listA)-1,epoch=225,change_lr=-1)
weight,bias =adab.fit()

print("test1")
testb=build_data_partB(3)
listB=net2.getLastHiddenLayerInfo(testb)
dftest=pd.DataFrame()
for i in range(len(listB)):
    if(i<len(listB)-1):
        dftest[i]=listB[i]
    else:
        dftest["value"]=listB[i]

count=adab.predict(dftest,testb,weight, bias)
print(adab.score(count, dftest))
plotting_test(testb)
con_mat1 = confusion_matrix(testb["pred"],testb["value"])
confussion_matrix(con_mat1)
print(classification_report(testb["value"],testb["pred"]))
```

```
plotting train
```

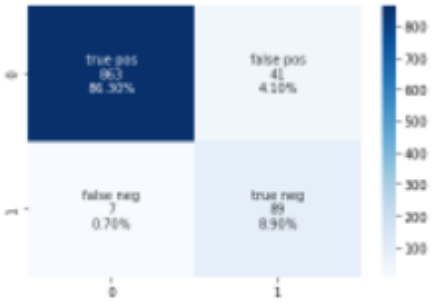




```
test1
0.952
```

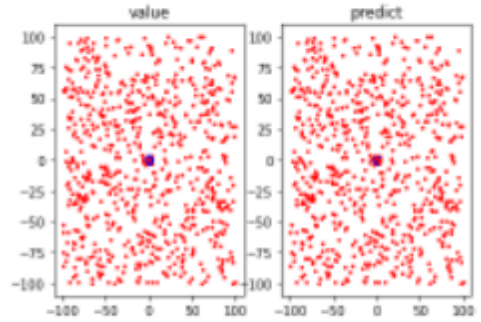



| | precision | recall | f1-score | support |
|---|---|---|---|---|
| -1 | 0.95 | 0.99 | 0.97 | 870 |
| 1 | 0.93 | 0.68 | 0.79 | 130 |
| accuracy | | | 0.95 | 1000 |
| macro avg | 0.94 | 0.84 | 0.88 | 1000 |
| weighted avg | 0.95 | 0.95 | 0.95 | 1000 |



```
print("test2")
testb=build_data_partB(7)
listB=net2.getLastHiddenLayerInfo(testb)
dftest=pd.DataFrame()
for i in range(len(listB)):
    if(i<len(listB)-1):
        dftest[i]=listB[i]
    else:
        dftest["value"]=listB[i]

count=adab.predict(dftest,testb,weight, bias)
print(adab.score(count, dftest))
plotting_test(testb)
con_mat1 = confusion_matrix(testb["pred"],testb["value"])
confussion_matrix(con_mat1)
print(classification_report(testb["value"],testb["pred"]))
```

```
test2
0.952
```



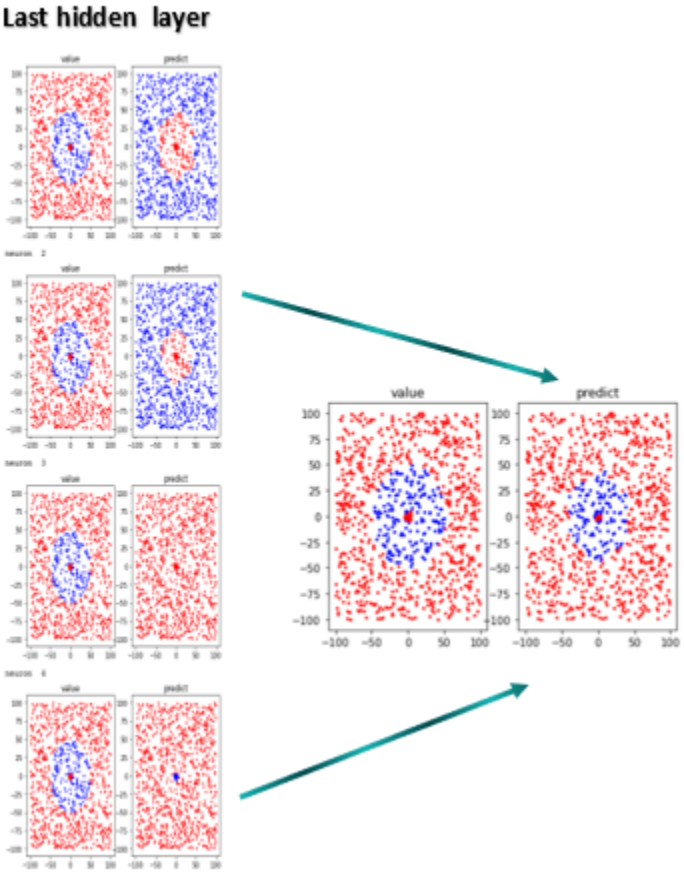

```
              precision    recall  f1-score   support

          -1       0.96      0.99      0.97       867
           1       0.89      0.73      0.80       133

    accuracy                           0.95      1000
   macro avg       0.92      0.86      0.89      1000
weighted avg       0.95      0.95      0.95      1000
```

## Summary and Discussion

In summary, we've learned that the neural network is more effective in classifying, no matter how the data is divided (whether linearly or non-linearly).

We've learned how to understand how each neuron classifies the data, and how to tweak the different parameters in order to improve our results.

We also learned that when taking the last layer of the neural network, and using that as training input for the adaline, one gets better results than when using adaline only (by looking at the results we've reached in our first assignment, and comparing them to part D, dataset A). While analyzing our results, we've noticed that the adaline classification is very similar to one of the neurons in the last hidden layer, but may classify the opposite, as shown below:



Additionally, we've learned how and when to update the learning rate in the algorithm parameters.

Furthermore, we noticed how when the neural network is greater and more complex, the better the results we are able to achieve (whilst avoiding overfitting, nonetheless).