simple-nn

March 16, 2025

0.1 Neural net experiments

[24]:
```
%matplotlib inline
import numpy as np
import torch
import matplotlib.pyplot as plt
```

0.1.1 1. Various helper functions

This function loads in a data set

[25]:
```
def load_data(datafile):
    data = np.loadtxt(datafile)
    n,p = data.shape
    rawx = data[:,0:2]
    rawy = data[:,2]
    x = torch.tensor(rawx, dtype=torch.float)
    y = torch.reshape(torch.tensor((rawy+1.0)/2.0, dtype=torch.float), [n,1])
    return x,y
```

This function plots the data set

[26]:
```
def plot_data(x,y):
    x_min = min(x[:,0]) - 1
    x_max = max(x[:,0]) + 1
    y_min = min(x[:,1]) - 1
    y_max = max(x[:,1]) + 1
    pos = (torch.squeeze(y) == 1)
    neg = (torch.squeeze(y) == 0)
    plt.plot(x[pos,0], x[pos,1], 'ro')
    plt.plot(x[neg,0], x[neg,1], 'k^')
    plt.xlim(x_min,x_max)
    plt.ylim(y_min,y_max)
    plt.show()
```

This function plots a decision boundary as well as the data points

[27]:
```
def plot_boundary(x,y,model):
```

```
    x_min = min(x[:,0]) - 1
    x_max = max(x[:,0]) + 1
    y_min = min(x[:,1]) - 1
    y_max = max(x[:,1]) + 1

    delta = 0.05
    xx, yy = np.meshgrid(np.arange(x_min, x_max, delta), np.arange(y_min,␣
↪y_max, delta))
    grid = np.c_[xx.ravel(), yy.ravel()]
    gn, gp = grid.shape
    Z = np.zeros(gn)
    for i in range(gn):
        pred = model(torch.tensor(grid[i,:], dtype=torch.float))
        Z[i] = int(pred > 0.5)

    # Put the result into a color plot
    Z = Z.reshape(xx.shape)
    plt.pcolormesh(xx, yy, Z, cmap=plt.cm.PRGn, vmin=-3, vmax=3)

    # Plot also the training points
    pos = (torch.squeeze(y) == 1)
    neg = (torch.squeeze(y) == 0)
    plt.plot(x[pos,0], x[pos,1], 'ro')
    plt.plot(x[neg,0], x[neg,1], 'k^')

    plt.xlim(x_min,x_max)
    plt.ylim(y_min,y_max)
    plt.show()
```

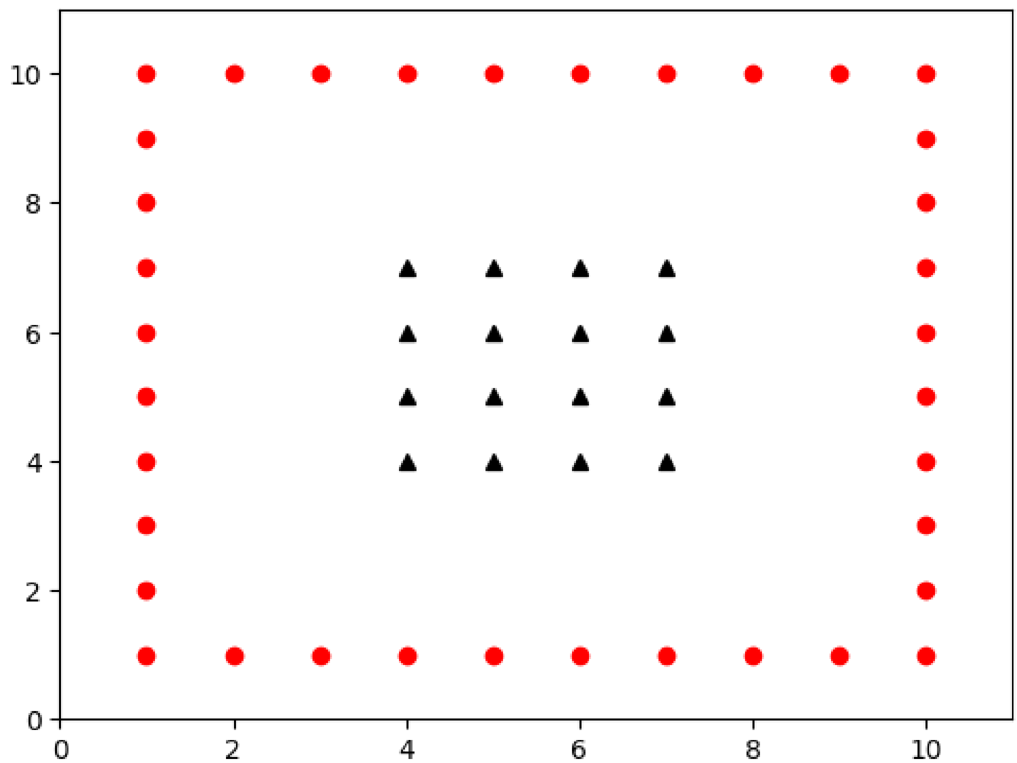
This function computes the error rate of the predicted labels `y1` given the true labels `y2`.

[28]:
```
def error_rate(y1, y2):
    sum = 0.0
    for i in range(0,y1.size()[0]):
        sum += ((y1[i]-0.5) * (y2[i]-0.5) <= 0.0)
    return int(sum)
```

0.1.2 2. Experiments with toy data

Let's load in one of the data sets and print it.

[42]:
```
x,y = load_data('data1.txt')
plot_data(x,y)
```



Next, we train a feedforward net on it. This takes many iterations of gradient descent (backpropagation). We'll print the status every 1000 iterations.

[ ]:
```
# Now train a neural net
#
# d is input dimension
# H is hidden dimension
d = 2
H = 4

# Use the nn package to define our model as a sequence of layers. nn.Sequential
# is a Module which contains other Modules, and applies them in sequence to
# produce its output. Each Linear Module computes output from input using a
# linear function, and holds internal Tensors for its weight and bias.
model = torch.nn.Sequential(
    torch.nn.Linear(d, H),
    torch.nn.ReLU(),
    torch.nn.Linear(H, 1),
    torch.nn.Sigmoid()
)

# The nn package also contains definitions of popular loss functions; in this
```

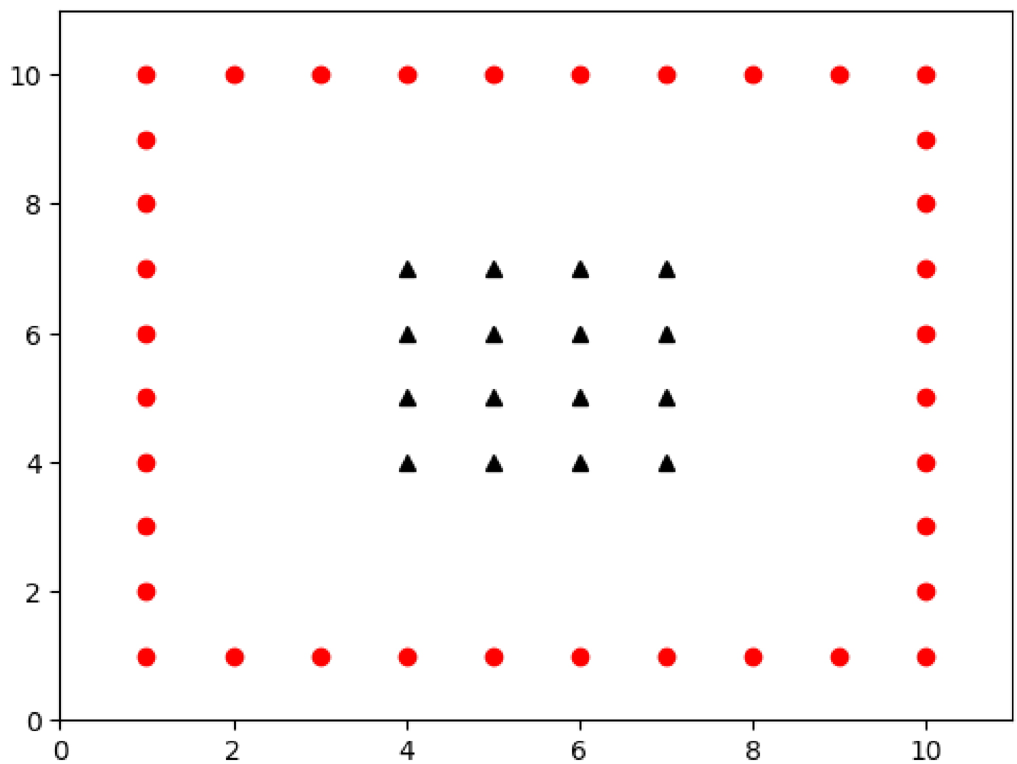
```
# case we will use binary cross entropy (BCE) as our loss function.
loss_fn = torch.nn.BCELoss()

prev_loss = 1.0
learning_rate = 0.25
done = False
t = 1
tol = 1e-4
while not(done):
    # Forward pass: compute predicted y by passing x to the model. Module␣
↪objects
    # override the __call__ operator so you can call them like functions. When
    # doing so you pass a Tensor of input data to the Module and it produces
    # a Tensor of output data.
    y_pred = model(x)
    t = t+1
    # Compute and print loss. We pass Tensors containing the predicted and true
    # values of y, and the loss function returns a Tensor containing the
    # loss.
    loss = loss_fn(y_pred, y)
    if t % 1000 == 0:
        print('Iteration %d: loss %0.5f errors %d' %
              (t, loss.item(), error_rate(y_pred, y)))
        if (prev_loss - loss.item() < tol):
            done = True
        prev_loss = loss.item()

    # Zero the gradients before running the backward pass.
    model.zero_grad()

    # Backward pass: compute gradient of the loss with respect to all the␣
↪learnable
    # parameters of the model. Internally, the parameters of each Module are␣
↪stored
    # in Tensors with requires_grad=True, so this call will compute gradients␣
↪for
    # all learnable parameters in the model.
    loss.backward()

    # Update the weights using gradient descent. Each parameter is a Tensor, so
    # we can access its gradients like we did before.
    with torch.no_grad():
        for param in model.parameters():
            param -= learning_rate * (1.0/np.sqrt(t)) * param.grad
print("Number of training errors:", error_rate(model(x), y))
plot_boundary(x,y,model)
```

[50]:
```python
def train_best_model(x, y, d, H, num_trials=5, tol=1e-4):
    best_error = float('inf')
    best_model = None
    best_iters = 0

    for trial in range(num_trials):
        model = torch.nn.Sequential(
            torch.nn.Linear(d, H),
            torch.nn.ReLU(),
            torch.nn.Linear(H, 1),
            torch.nn.Sigmoid()
        )

        loss_fn = torch.nn.BCELoss()
        learning_rate = 0.25
        prev_loss = 1.0
        t = 1
        done = False

        while not done:
            y_pred = model(x)
            t += 1
            loss = loss_fn(y_pred, y)

            if t % 1000 == 0:
                #print(f"Trial {trial+1}, Iteration {t}: loss={loss.item():.5f}␣
↪errors={error_rate(y_pred, y)}")
                if (prev_loss - loss.item() < tol):
                    done = True
                prev_loss = loss.item()

            model.zero_grad()
            loss.backward()

            with torch.no_grad():
                for param in model.parameters():
                    param -= learning_rate * (1.0 / np.sqrt(t)) * param.grad

        final_pred = model(x)
        errors = error_rate(final_pred, y)

        if errors < best_error:
            best_error = errors
            best_model = model
            best_iters = t

    # Plot the best model boundary
```

```
    plot_boundary(x, y, best_model)

    print(f"Best H: {H}")
    print(f"Number of iterations until convergence: {best_iters}")
    print(f"Training errors: {best_error}")

    return best_model, best_error, best_iters
```

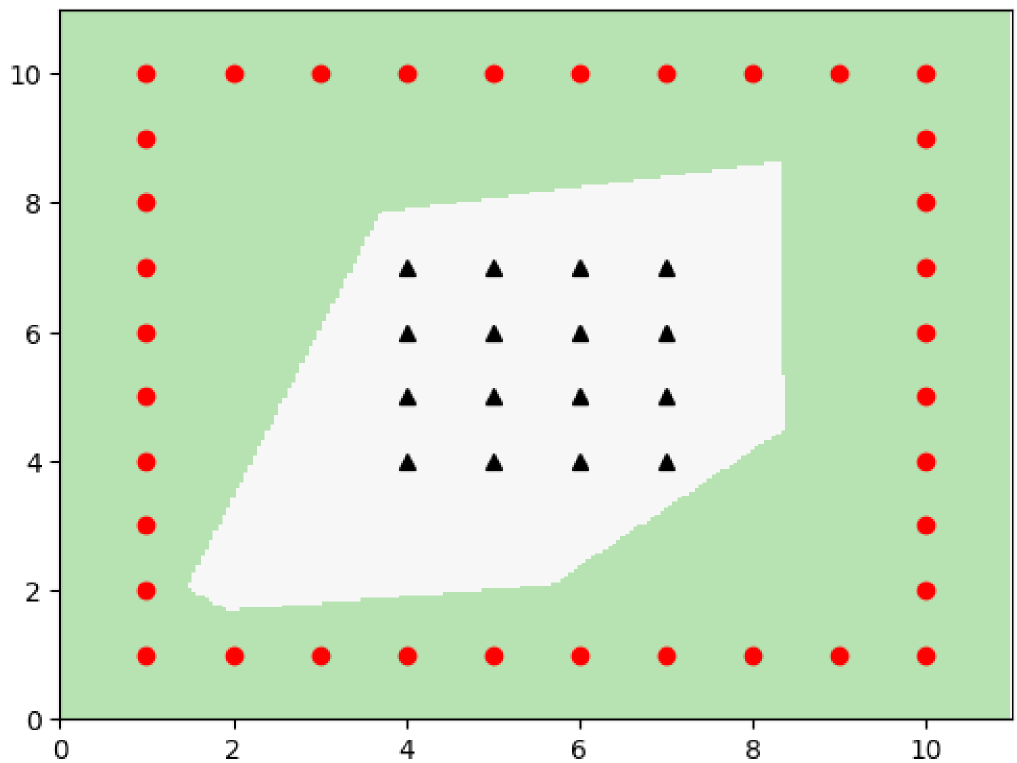
[51]:
```
for i in range(1, 6):
    print(f"\n--- Training on data{i}.txt ---")
    x, y = load_data(f"data{i}.txt")
    plot_data(x, y)

    for H_val in [4, 8]:
        print(f"\nTraining with H = {H_val}")
        train_best_model(x, y, d=2, H=H_val)
```

```
--- Training on data1.txt ---
```

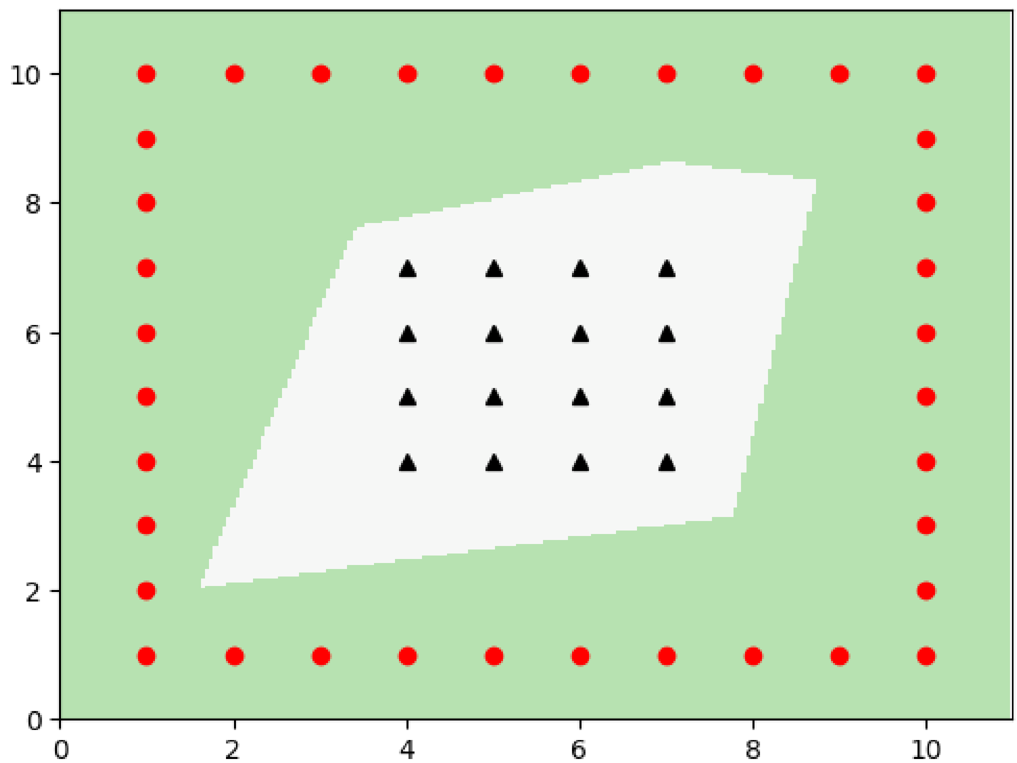



```
Training with H = 4
```

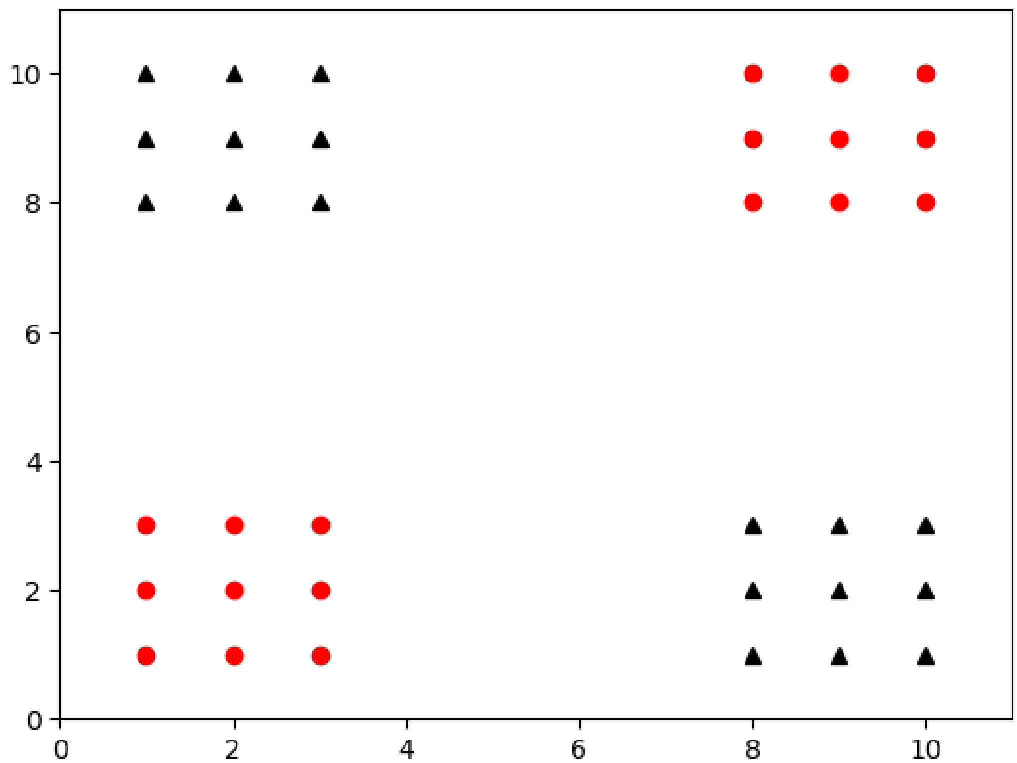
```
/var/folders/4h/z3_372tn7g5gxfj_rm1sbdc00000gn/T/ipykernel_52915/3617984656.py:9
: DeprecationWarning: __array__ implementation doesn't accept a copy keyword, so
passing copy=False failed. __array__ must implement 'dtype' and 'copy' keyword
arguments.
  xx, yy = np.meshgrid(np.arange(x_min, x_max, delta), np.arange(y_min, y_max,
delta))
```



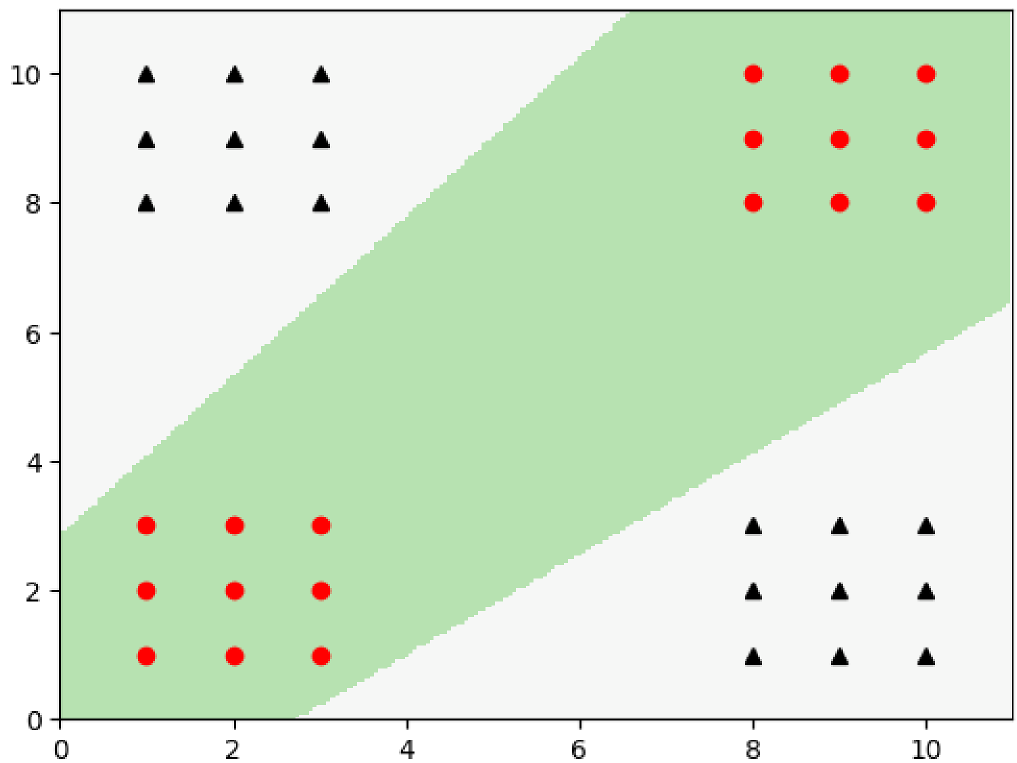

```
Best H: 4
Number of iterations until convergence: 156000
Training errors: 0

Training with H = 8
```



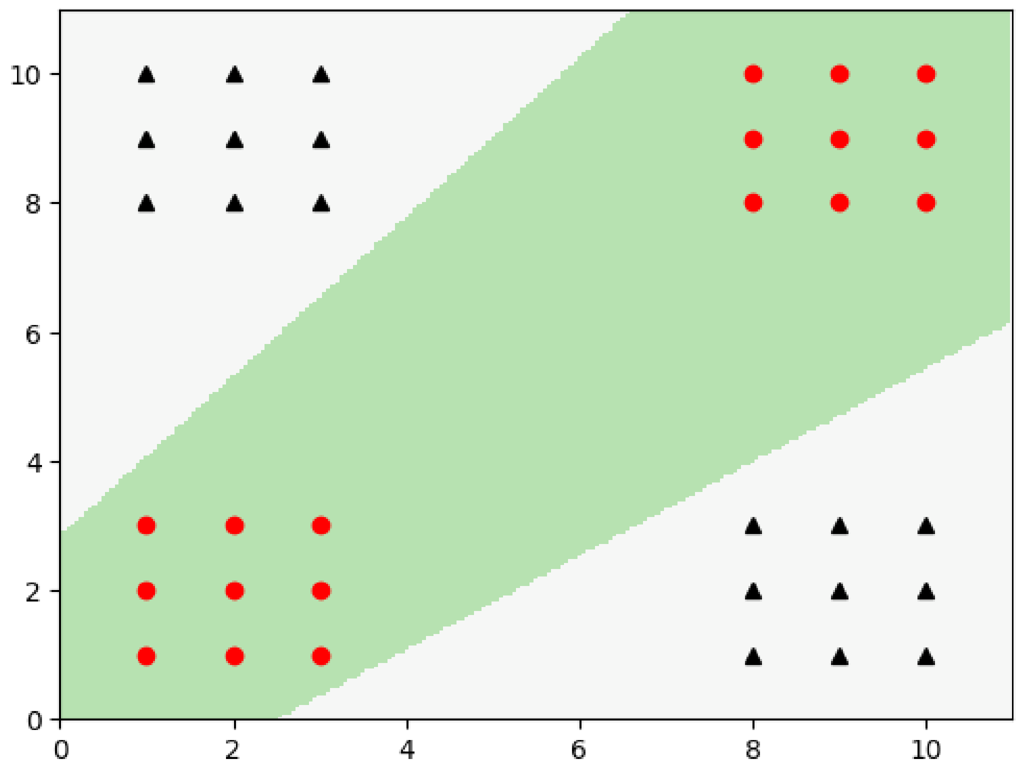

```
Best H: 8
Number of iterations until convergence: 187000
Training errors: 0

--- Training on data2.txt ---
```



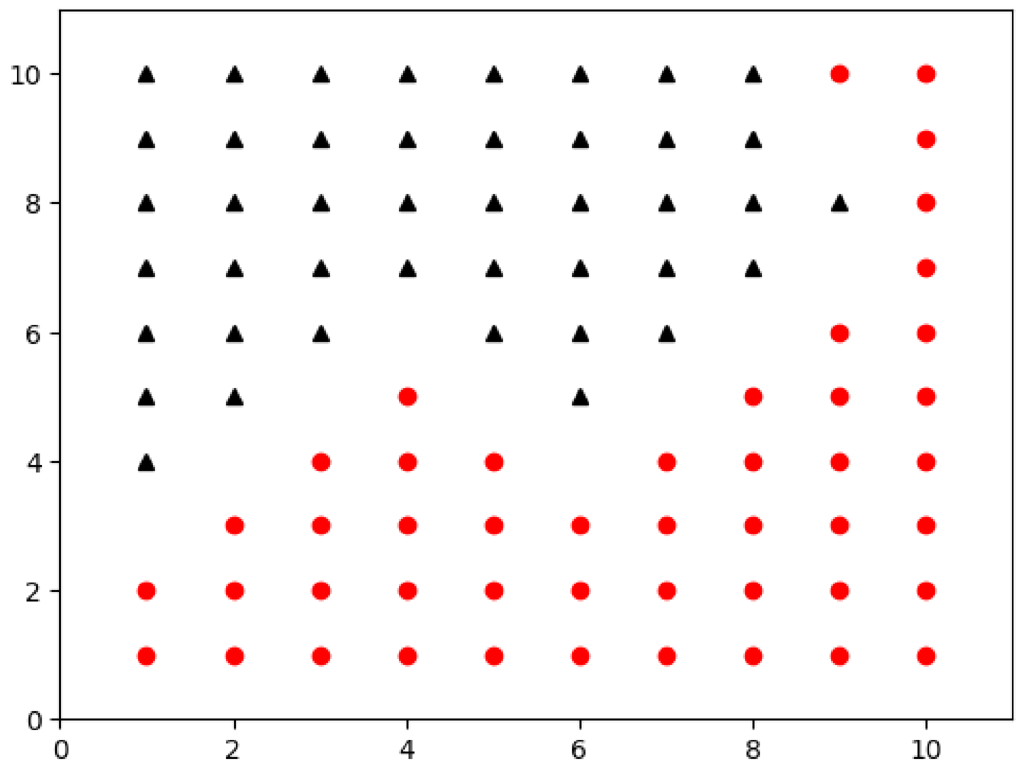
`Training with H = 4`



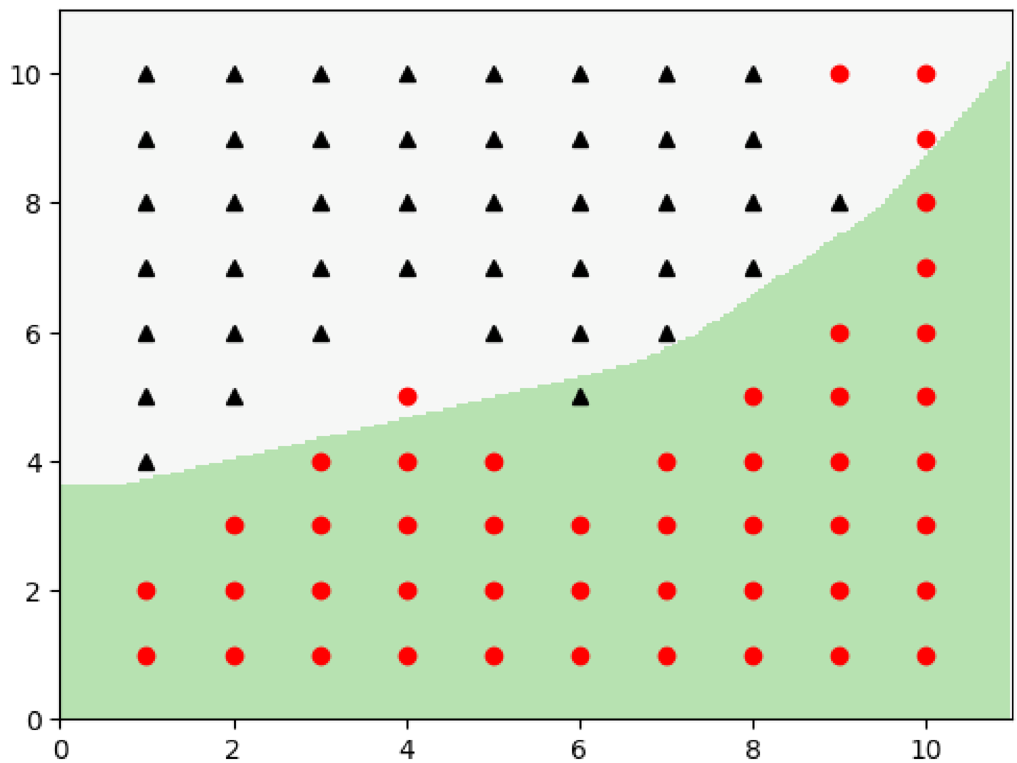

```
Best H: 4
Number of iterations until convergence: 59000
Training errors: 0

Training with H = 8
```



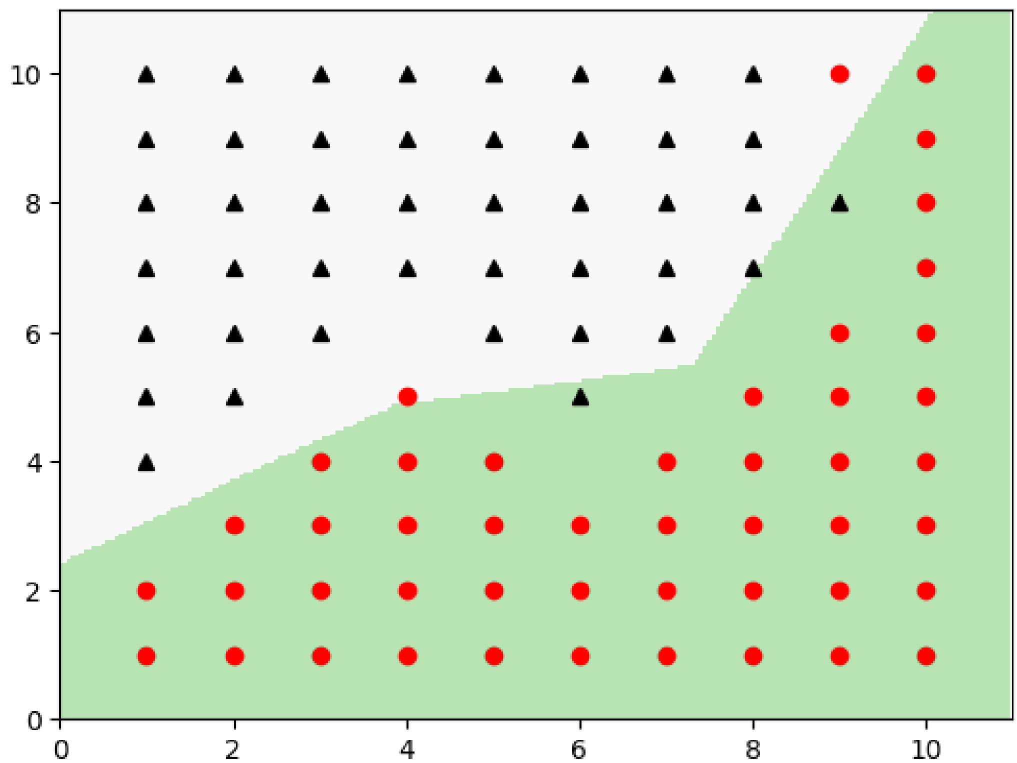

```
Best H: 8
Number of iterations until convergence: 64000
Training errors: 0

--- Training on data3.txt ---
```



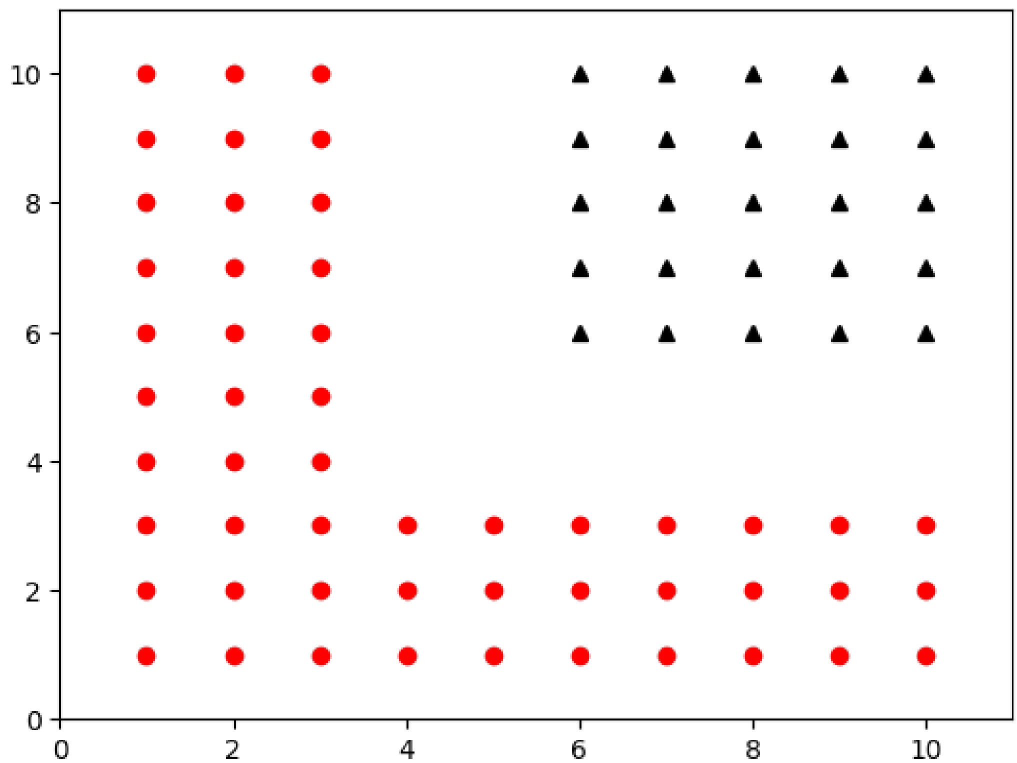
Training with H = 4



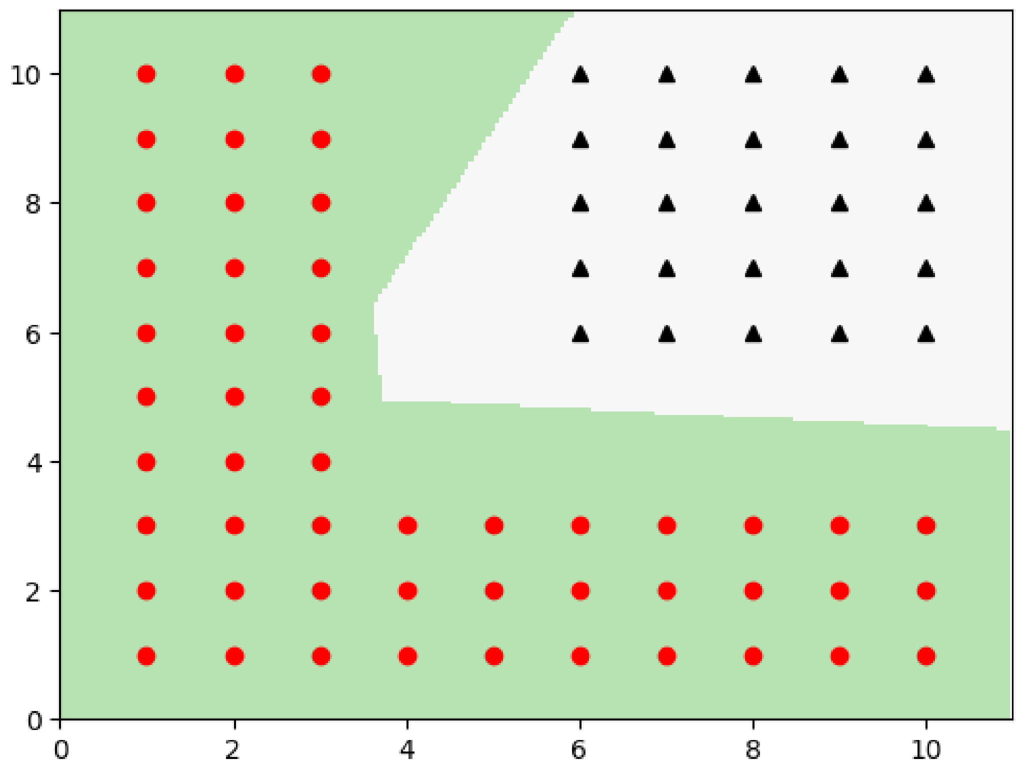

```
Best H: 4
Number of iterations until convergence: 185000
Training errors: 5

Training with H = 8
```



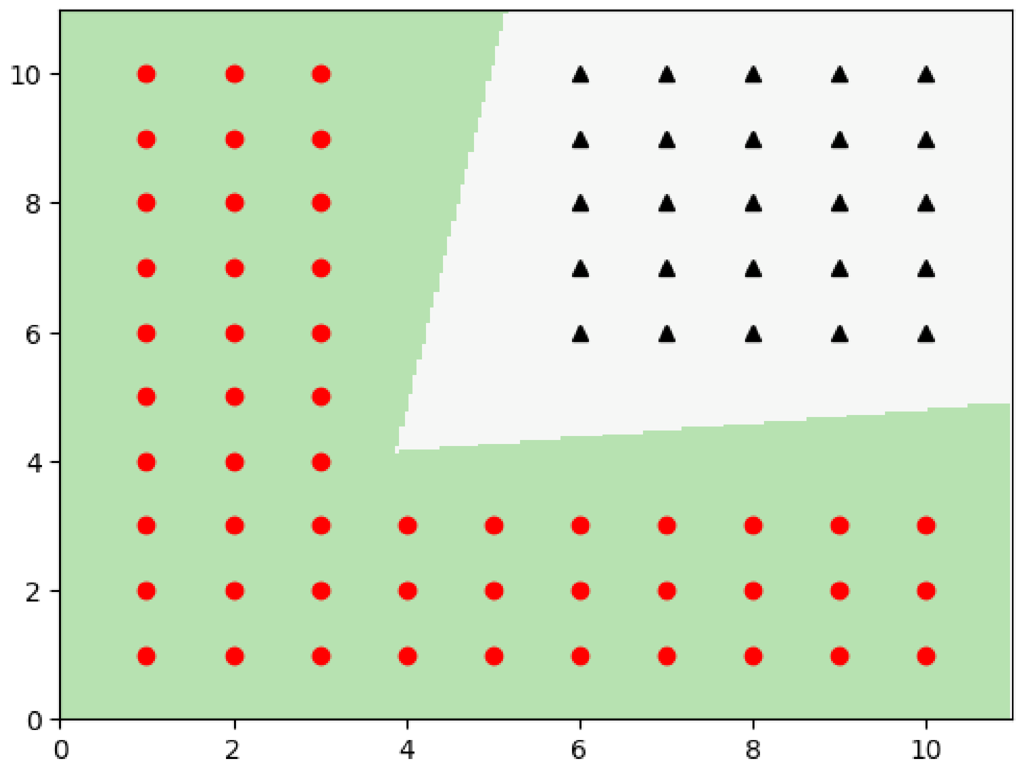

```
Best H: 8
Number of iterations until convergence: 486000
Training errors: 4

--- Training on data4.txt ---
```



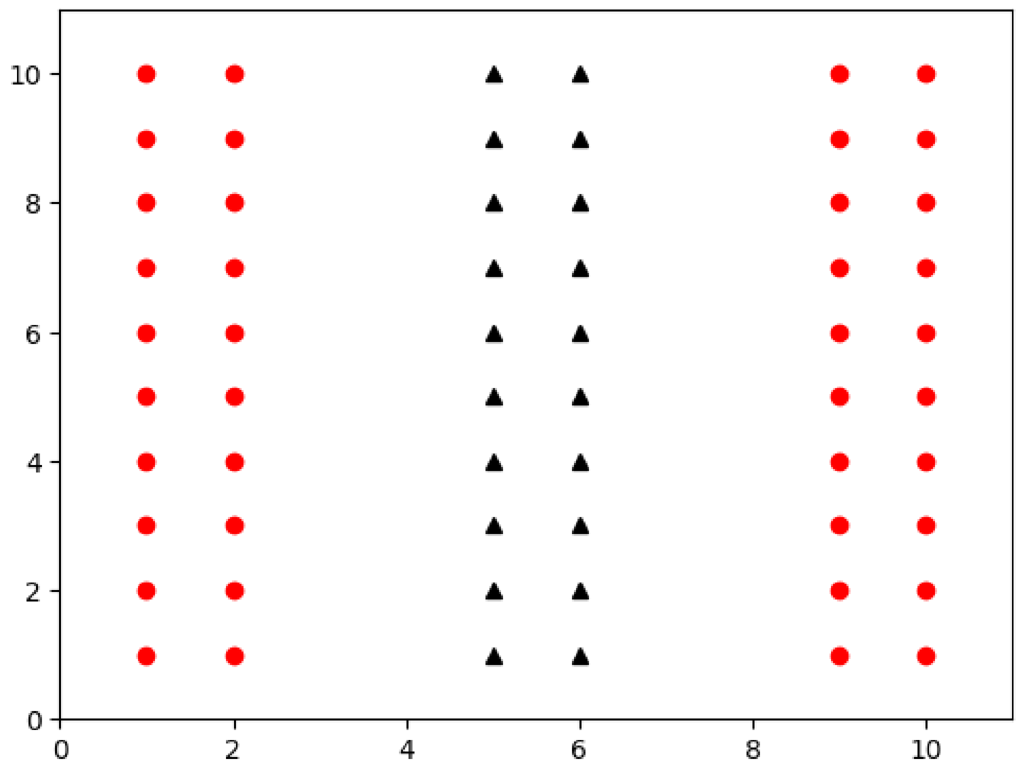
`Training with H = 4`



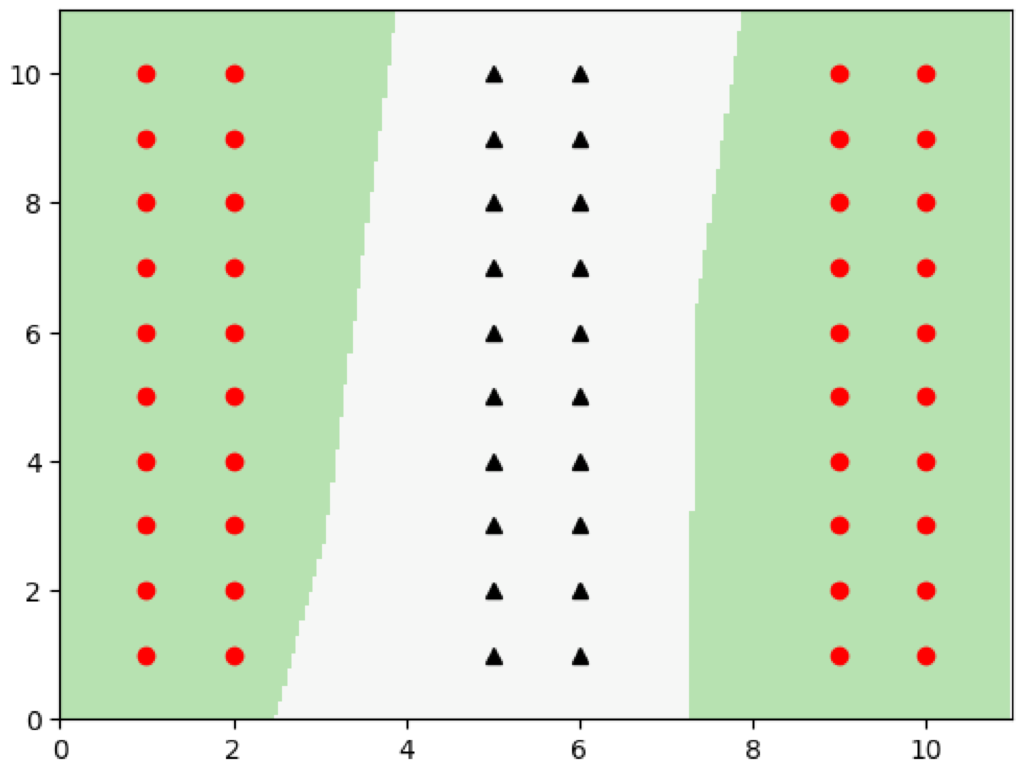

```
Best H: 4
Number of iterations until convergence: 141000
Training errors: 0

Training with H = 8
```



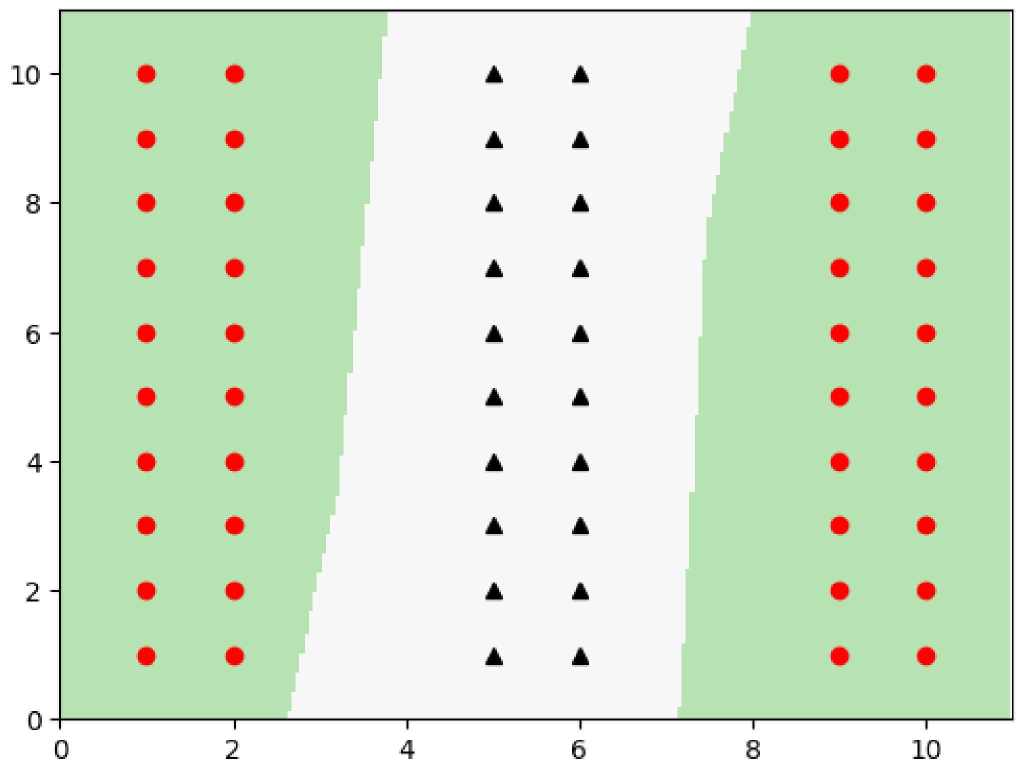

```
Best H: 8
Number of iterations until convergence: 50000
Training errors: 0

--- Training on data5.txt ---
```



Training with H = 4



```
Best H: 4
Number of iterations until convergence: 182000
Training errors: 0

Training with H = 8
```

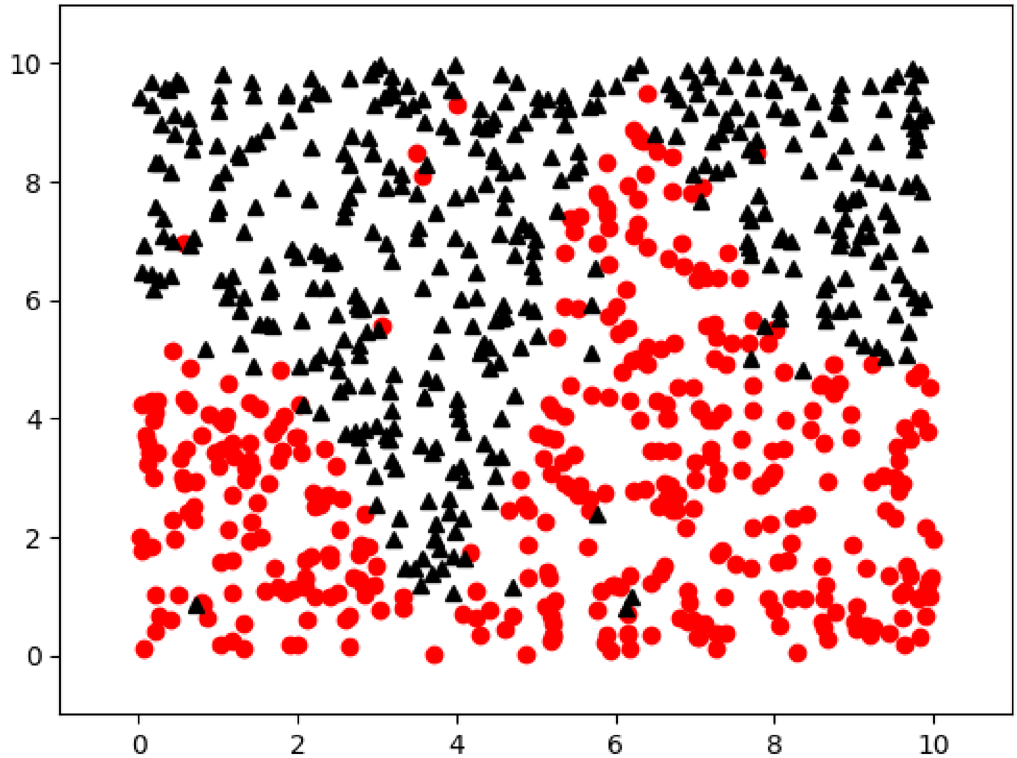



```
Best H: 8
Number of iterations until convergence: 114000
Training errors: 0
```

0.1.3 3. A different data set

The code in the next cell generates a data set of 800 points in which the labels are noisy.

[54]:
```
n = 800
np.random.seed(0)
X_train = np.random.rand(n,2)
x1 = X_train[:,0]
x2 = X_train[:,1]
y_train = ((np.exp(-((x1-0.5)*6)**2)*2*((x1-0.5)*6)+1)/2-x2)>0

idx = np.random.choice(range(n),size=(int(n*0.03),))
y_train[idx] = ~y_train[idx]
x = torch.tensor(X_train, dtype=torch.float) * 10
y = torch.reshape(torch.tensor(y_train, dtype=torch.float), [n,1])
plot_data(x,y)
```



Define a neural net with two hidden layers, each containing the same number of nodes. Hint: Start with the code above and just make a small tweak to it.

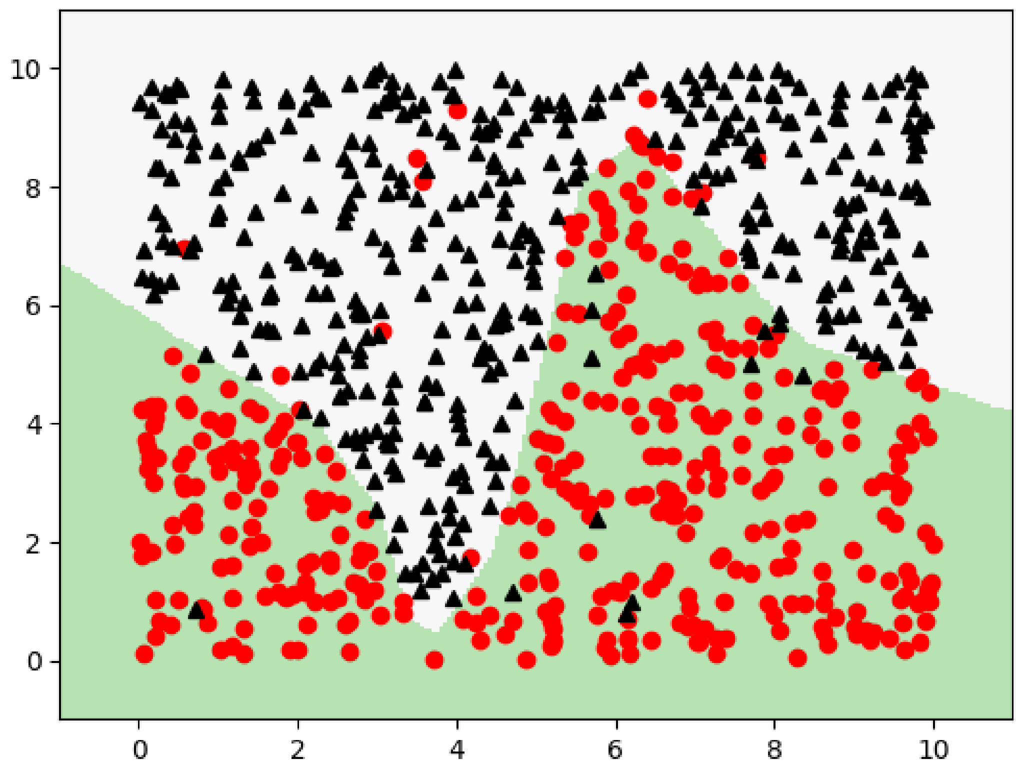
Train the net a few times, and print the decision boundary for the best (lowest-error) model that you find.

[55]:
```
def train_best_model_two_layers(x, y, d, H, num_trials=5, tol=1e-4):
    best_error = float('inf')
    best_model = None
    best_iters = 0

    for trial in range(num_trials):
        model = torch.nn.Sequential(
            torch.nn.Linear(d, H),
            torch.nn.ReLU(),
            torch.nn.Linear(H, H),
            torch.nn.ReLU(),
            torch.nn.Linear(H, 1),
            torch.nn.Sigmoid()
        )

        loss_fn = torch.nn.BCELoss()
        learning_rate = 0.25
```

```
        prev_loss = 1.0
        t = 1
        done = False

        while not done:
            y_pred = model(x)
            t += 1
            loss = loss_fn(y_pred, y)

            if t % 1000 == 0:
                #print(f"Trial {trial+1}, Iteration {t}: loss={loss.item():.5f}␣
↪errors={error_rate(y_pred, y)}")
                if (prev_loss - loss.item() < tol):
                    done = True
                prev_loss = loss.item()

            model.zero_grad()
            loss.backward()

            with torch.no_grad():
                for param in model.parameters():
                    param -= learning_rate * (1.0 / np.sqrt(t)) * param.grad

        final_pred = model(x)
        errors = error_rate(final_pred, y)

        if errors < best_error:
            best_error = errors
            best_model = model
            best_iters = t

    # Plot the best model boundary
    plot_boundary(x, y, best_model)

    print(f"Best H: {H}")
    print(f"Number of iterations until convergence: {best_iters}")
    print(f"Training errors: {best_error}")

    return best_model, best_error, best_iters
```

[57]:
```
train_best_model_two_layers(x, y, d = 2, H = 8, num_trials = 50)
```

```
/var/folders/4h/z3_372tn7g5gxfj_rm1sbdc00000gn/T/ipykernel_52915/3617984656.py:9
: DeprecationWarning: __array__ implementation doesn't accept a copy keyword, so
passing copy=False failed. __array__ must implement 'dtype' and 'copy' keyword
arguments.
  xx, yy = np.meshgrid(np.arange(x_min, x_max, delta), np.arange(y_min, y_max,
delta))
```




```
Best H: 8
Number of iterations until convergence: 510000
Training errors: 32
```

[57]:
```
(Sequential(
   (0): Linear(in_features=2, out_features=8, bias=True)
   (1): ReLU()
   (2): Linear(in_features=8, out_features=8, bias=True)
   (3): ReLU()
   (4): Linear(in_features=8, out_features=1, bias=True)
   (5): Sigmoid()
 ),
 32,
 510000)
```

[ ]: