TP DT

Save the notebook as either PDF or HTML and make sure all the results are saved correctly (I won't run them and the original format does not save the results automatically), **and put your name in the filename**.

Questions are in green boxes. The maximum time you should spend on each question is given as indication only. If you take more time than that, then you should come see me.

Analyzes are in blue boxes. You should comment on your results in theses boxes (Is it good? Is it expected? Why do we get such result? Why is it different from the previous one? etc)

In [ ]:
```
%%shell
jupyter nbconvert --to html /PATH/TO/YOUR/NOTEBOOKFILE.ipynb
```

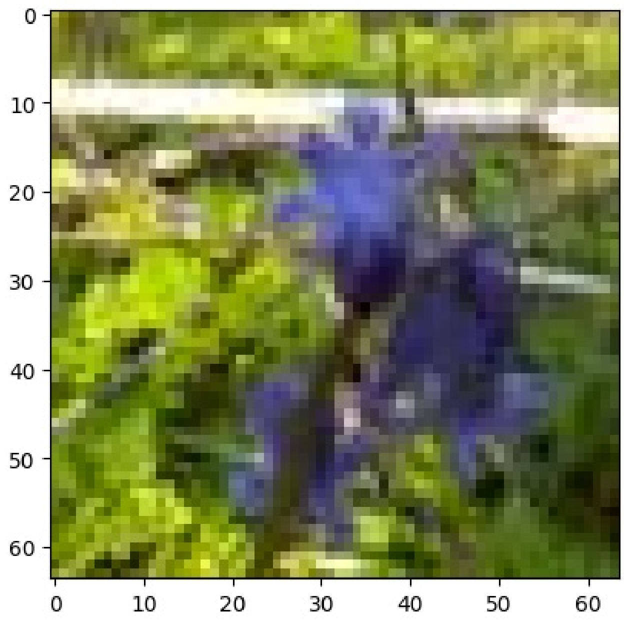
In [1]:
```
import jax
import jax.numpy as jnp
import numpy as np
import gzip
import pickle
import matplotlib.pyplot as plt
import time
```

For this lab, we will use the bluebell dataset. It consists of 64×64 color images, which we will have to flatten into $12k$ dimensional vectors. The code for the dataset comes with several train/val/test splits, but in this notebook, we will use the first split and do our own cross-validation routines.

In [ ]:
```
!unzip bluebell_64.zip
```

In [8]:
```
# Load the dataset
from bluebell import Bluebell
X_train_ds = Bluebell('bluebell_64', 'train', split=0)
X_val_ds = Bluebell('bluebell_64', 'val', split=0)
X_train = np.array([img.flatten()/127.5 - 1. for img, lab in X_train_ds])
y_train = np.array([lab for img, lab in X_train_ds])
X_val = np.array([img.flatten()/127.5 - 1. for img, lab in X_val_ds])
y_val = np.array([lab for img, lab in X_val_ds])
plt.imshow(X_train[0].reshape(64, 64, 3)/2+0.5)
print(y_train[0])
```

```
0
```



Next, we want to reduce the number of dimensions that will be search through with the decision trees. Since our images are $64 \times 64 \times 3$ values, this leads to a very high dimensional space that has to be searched at each step. However, dimensions where all images have the same value, or very close values, will never be selected in the tree because they do not provide a good gain.

We will thus select only the 2048 dimension with the highest variance to perform our analysis.

In [9]:
```
dim = jnp.argsort(X_train.std(axis=0), descending=True)[0:2048] #it's close the the idea of PCA ..maximizing th
X_train = jnp.array(X_train[:, dim])
X_val = jnp.array(X_val[:, dim])
```

```
print(X_train.shape, X_val.shape)
```

```
(1200, 2048) (300, 2048)
```

In [10]:
```
print(np.unique(y_train))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11]
```

Implementing a randomized Decision Tree

Q1. Implement the code of a function that finds an optimal threshold along a given dimension, using the $0-1$ loss with specified example weights and test it on the 150th dimension. To speed-up things, we will only consider 8 thresholds between the minimum and maximum value (use 'linspace'). Compare it to assigning a unique label to all samples. You should get a significant decrease of loss from ~0.92 (random 1/12 chance) to ~0.84. *(Indicative time: 10 minutes for a slow version, but take the extra 20 minutes to have a parallel version testing all thresholds at once that runs in under 1s, it is worth it for the next questions.)*

In [11]:
```
'''
takes arguements
y_pred: prediction
y_true: true labels
weights: weights for each example
'''
@jax.jit
def zeroOneLoss(y_pred, y_true, weights):
    return (weights * (y_pred != y_true)).sum(axis=0)/(1e-12+weights.sum(axis=0))
```

In [12]:
```
import jax.nn

'''
takes arguments
X: training samples
y: training labels
dim: dimension to use
w: weight associated to each example (can be True/False or 1/0 to remove some examples)
returns the gain and the threshold
# '''

@jax.jit
def findBestTh(X, y, dim, w):
    n = X.shape[0]
    theta = jnp.linspace((X[:, dim]).min(), (X[:, dim]).max(), 8)
    C_0 = zeroOneLoss(jnp.full((n, 12), jnp.arange(12)), y[:, None], w[:, None]).min()
    mask_left = X[:, dim, None] < theta[None, :]

    C_left = zeroOneLoss(jnp.full((n, 12), jnp.arange(12))[:, None, :], y[:, None, None], mask_left[:, :, None]
    C_right = zeroOneLoss(jnp.full((n, 12), jnp.arange(12))[:, None, :], y[:, None, None], ~mask_left[:, :, Non
    C = C_0 - (jnp.mean(mask_left, axis=0).T * (C_left - C_right) +  C_right)
    idx = C.argmax()
    return  C[idx] ,theta[idx]
```

In [ ]:
```
%%time

G, th = findBestTh(X_train, y_train, 150, jnp.ones(len(y_train)))
print(G, th)

w_below = jnp.ones(len(X_train)) * (X_train[:, 150] <= th)
w_above = jnp.ones(len(X_train)) * (X_train[:, 150] >= th)

print("Nonzero weights below or equal to threshold:", jnp.count_nonzero(w_below))
print("Nonzero weights above or equal to threshold:", jnp.count_nonzero(w_above))

G, th = findBestTh(X_train, y_train, 150, w_below)
print(G, th)
G, th = findBestTh(X_train, y_train, 150, w_above)
print(G, th)
```

```
0.07833338 -0.42857143
Nonzero weights below or equal to threshold: 573
Nonzero weights above or equal to threshold: 627
-0.012853384 -0.42857143
0.011746407 -0.42857143
CPU times: user 497 ms, sys: 7.7 ms, total: 505 ms
Wall time: 506 ms
```

Analyze your results in this box. Observing what we got above , there is a bug in the assigning zeros to weights after doing the split, because we get always the same th, which can't be true for all cases. Selecting 8 values for splits often fails to effectively divide the data, typically resulting in trivial divisions such as 0 versus X or its reverse, without genuinely informative splits. This choice needs refinement, particularly by incorporating weights to ensure that the chosen thresholds represent the underlying data distribution more accurately. Additionally, a repeated issue arises from not dynamically updating the selection filter; without changes, we consistently select the same dimensions and thresholds, which leads to redundant and ineffective splits. Addressing this by updating the filter criteria and using indexed masks for splitting ensures more logical results and avoids repetitive selection of the same dimension and threshold in subsequent iterations.

We can vectorize over the dimension by using vmap. The batched function can now operate on a array of dimensions.

In [13]:
```
batched_findBestTh = jax.vmap(findBestTh, in_axes=(None, None, 0, None), out_axes=0)
```

Q2. Wrap the batched function in a function that test all dimensions to find the best combination of component and threshold. Use blocks of 256 dimensions to process at a time, as we found it a good setup with respect to speed (You can change those values later to optimize for speed).Test it on the entire train set and make sure it obtains the lowest error.*(Indicative time: It could take you 15 minutes to an hour to code and should run in less than 5 seconds.)*

In [14]:
```
batched_findBestTh_again = jax.vmap(batched_findBestTh, in_axes=(None, None, 0, None), out_axes=0)
```

In [15]:
```
@jax.jit
def findBestDTh(X, y, w):
    dim_batches = jnp.array(jnp.split(np.arange(2048), 8))
    Gains, ths = batched_findBestTh_again(X, y, dim_batches, w)
    Gains = Gains.flatten()
    ths = ths.flatten() #to avoid the loops
    best_dim = jnp.argmax(Gains)
    best_gain = Gains[best_dim]
    best_th = ths[best_dim]

    return best_gain, best_dim, best_th
```

In [24]:
```
%%time
g, d, t = findBestDTh(X_train, y_train, jnp.ones(len(X_train)))
print("gain: {} dim: {} th: {}".format(g, d, t))

w_below = jnp.ones(len(X_train)) * (X_train[:, d] <= t)
w_above = jnp.ones(len(X_train)) * (X_train[:, d] >= t)

g, d, t = findBestDTh(X_train, y_train, w_below)
print("gain: {} dim: {} th: {}".format(g, d, t))

g, d, t = findBestDTh(X_train, y_train, w_above)
print("gain: {} dim: {} th: {}".format(g, d, t))
```

```
gain: 0.08250004053115845 dim: 30 th: -0.4285714328289032
gain: -0.008686721324920654 dim: 30 th: -0.4285714328289032
gain: 0.007938623428344727 dim: 30 th: -0.4285714328289032
CPU times: user 5.83 s, sys: 1.08 s, total: 6.91 s
Wall time: 5.16 s
```

This aligns with the observation we had before, so I decided to split the data using indexes, to solve that bug.

Q3. Implement the code of the Decision Tree class by using the previous functions. To achieve reasonable speed, loop only over dimensions that have variations (there is no threshold if all samples have the same value) in batches of 256 or more, inspired by the previous function. Do not split and slice the data but zero the associated weights instead, it is faster (all functions have the same size of arrays and are thus compiled and optimized only once). Debug it on only 256 dimensions and 256 samples, because using all dimensions/samples takes about 2 minutes. Test it on the full set with a maximum depth of 8 and a leaf size less than 10 to analyze and comment. *(Indicative time: It could take you 30 minutes to an hour to code and debug since it involves recursion.)*

We first try on the training set reduced to digits 0 and 1 with all dimensions to check that our code works.

In [17]:
```
#for debbuging :
x_train_01 = X_train
y_train_01 = y_train
x_val_01 = X_val
y_val_01 = y_val


x_train_01 = x_train_01[256:512, 0:256]
y_train_01 = y_train_01[0:256]
x_val_01 = x_val_01[0:100, 0:256]
y_val_01 = y_val_01[0:100]
```

In [18]:
```
class RandomizedDT():
    def __init__(self, percent_dimension=1.0, max_depth=8, max_size=20, verbose=False, space=0, pos=None , rng_
        self.max_depth = max_depth
        self.max_size = max_size
        self.percent_dimension = percent_dimension
        self.verbose = verbose
        self.space = space
        self.pos = pos
        self.left = None
        self.right = None
        self.label = None
```

```python
        self.rng_key =  jax.random.key(0) if rng_key is None else rng_key

    def fit(self, X, y, w=None):
        n, p = X.shape
        w = jnp.ones(n) if w is None else w
        label = jnp.argmax(jnp.sum(jax.nn.one_hot(y, num_classes=12) * w[:, None], axis=0))
        if self.max_depth == 0 or len(w) <= self.max_size :
            self.label = label
            if self.verbose:
                print(f'{" " * self.space}Leaf: Class {self.label}, Depth {self.max_depth}')
            return

        self.rng_key, skey = jax.random.split(self.rng_key)
        acceptable_dims = jnp.nonzero(jnp.std(X, axis=0))[0]
        dims = jax.random.permutation(skey, acceptable_dims)[:int(len(acceptable_dims) * self.percent_dimension
        n_batches = int(len(acceptable_dims) * self.percent_dimension) // 4
        dim_batches = jnp.array(jnp.split(dims, n_batches))
        Gains, ths = batched_findBestTh_again(X, y, dim_batches, w)
        Gains = Gains.flatten()
        ths = ths.flatten()
        idx= jnp.argmax(Gains)
        self.dim, self.th = dims[idx], ths[idx]

        if self.verbose:
            print(f'{" " * self.space}Node: Gain {Gains[idx]}, Dim {self.dim}, Threshold {self.th}, Depth {self

        left_mask = X[:, self.dim] <= self.th
        right_mask = ~left_mask
        self.left = RandomizedDT(self.percent_dimension, self.max_depth - 1, self.max_size, self.verbose, self.
        self.right = RandomizedDT(self.percent_dimension, self.max_depth - 1, self.max_size, self.verbose, self
        self.left.fit(X[left_mask], y[left_mask], w[left_mask])
        self.right.fit(X[right_mask], y[right_mask], w[right_mask])
        # self.left.fit(X, y, w*right_mask) #that doesn't work unfortunately
        # self.right.fit(X, y, w*left_mask)

    def predict(self, X):
        if self.label is not None :
            return jnp.array([self.label])
        return jnp.concatenate([self.left.predict([x]) if x[self.dim] < self.th else self.right.predict([x]) fo
```

In [19]:

```python
%%time
decision_tree = RandomizedDT(percent_dimension=1.0, max_depth= 8 ,  max_size= 10 ,verbose = True)
decision_tree.fit(x_train_01,y_train_01)
predictions  = decision_tree.predict(x_val_01)
```

```
Node: Gain 0.08203125, Dim 46, Threshold -0.4285714328289032, Depth 8
   Node: Gain 0.08461540937423706, Dim 240, Threshold -0.4464986026287079, Depth 7
      Node: Gain 0.07843142747879028, Dim 235, Threshold -0.7770308256149292, Depth 6
         Node: Gain 0.10294115543365479, Dim 67, Threshold -0.8117647171020508, Depth 5
            Node: Gain 0.11320751905441284, Dim 181, Threshold -0.9260504245758057, Depth 4
               Node: Gain 0.10344827175140381, Dim 57, Threshold -0.7355742454528809, Depth 3
                  Node: Gain 0.1304347813129425, Dim 198, Threshold -0.8745098114013672, Depth 2
                     Node: Gain 0.10526315867900848, Dim 88, Threshold -0.848739504814148, Depth 1
                        Leaf: Class 0, Depth 0
                        Leaf: Class 4, Depth 0
                     Leaf: Class 3, Depth 1
                  Leaf: Class 4, Depth 2
               Node: Gain 0.12499997019767761, Dim 231, Threshold -0.6459383964538574, Depth 3
                  Node: Gain 0.1499999761581421, Dim 46, Threshold -0.9305322170257568, Depth 2
                     Node: Gain 0.09090910106897354, Dim 233, Threshold -0.8610644340515137, Depth 1
                        Leaf: Class 3, Depth 0
                        Leaf: Class 4, Depth 0
                     Leaf: Class 0, Depth 1
                  Leaf: Class 4, Depth 2
            Node: Gain 0.20000001788139343, Dim 219, Threshold -0.8274509906768799, Depth 4
               Leaf: Class 4, Depth 3
               Leaf: Class 9, Depth 3
         Node: Gain 0.11764705181121826, Dim 24, Threshold -0.14285707473754883, Depth 5
            Node: Gain 0.10000002384185791, Dim 226, Threshold -0.6168067455291748, Depth 4
               Node: Gain 0.1428571343421936, Dim 158, Threshold -0.831932783126831, Depth 3
                  Node: Gain 0.142857164144516, Dim 237, Threshold -0.7624650001525879, Depth 2
                     Node: Gain 0.0, Dim 167, Threshold -1.0, Depth 1
                        Leaf: Class 9, Depth 0
                        Leaf: Class 9, Depth 0
                     Leaf: Class 4, Depth 1
                  Leaf: Class 3, Depth 2
               Leaf: Class 7, Depth 3
            Leaf: Class 11, Depth 4
      Node: Gain 0.142857164144516, Dim 140, Threshold 0.4285714626312256, Depth 6
         Node: Gain 0.1499999761581421, Dim 203, Threshold -0.7378151416778564, Depth 5
            Leaf: Class 6, Depth 4
            Node: Gain 0.08333335816860199, Dim 42, Threshold 0.7277312278747559, Depth 4
               Leaf: Class 1, Depth 3
               Leaf: Class 2, Depth 3
         Leaf: Class 2, Depth 5
   Node: Gain 0.08730155229568481, Dim 238, Threshold 0.4285714626312256, Depth 7
      Node: Gain 0.10344827175140381, Dim 159, Threshold 0.7142858505249023, Depth 6
         Node: Gain 0.11267608404159546, Dim 41, Threshold -0.19999992847442627, Depth 5
```

```
            Node: Gain 0.050000011920928955, Dim 196, Threshold -0.7277311086654663, Depth 4
               Leaf: Class 8, Depth 3
               Node: Gain 0.0555555522441864, Dim 131, Threshold 0.5596640110015869, Depth 3
                  Node: Gain 0.0588235259056091, Dim 19, Threshold 0.9372549057006836, Depth 2
                     Node: Gain 0.0588235259056091, Dim 64, Threshold 0.7098039388656616, Depth 1
                        Leaf: Class 2, Depth 0
                        Leaf: Class 0, Depth 0
                     Leaf: Class 0, Depth 1
                  Leaf: Class 11, Depth 2
            Node: Gain 0.11764711141586304, Dim 201, Threshold 0.4285714626312256, Depth 4
               Node: Gain 0.10810810327529907, Dim 151, Threshold -0.4285714328289032, Depth 3
                  Leaf: Class 10, Depth 2
                  Node: Gain 0.09677419066429138, Dim 118, Threshold 0.7647059559822083, Depth 2
                     Node: Gain 0.07142859697341919, Dim 121, Threshold -0.4756302535533905, Depth 1
                        Leaf: Class 11, Depth 0
                        Leaf: Class 8, Depth 0
                     Leaf: Class 7, Depth 1
               Node: Gain 0.142857164144516, Dim 208, Threshold -0.3949579894542694, Depth 3
                  Leaf: Class 6, Depth 2
                  Node: Gain 0.1666666567325592, Dim 7, Threshold 0.2179272174835205, Depth 2
                     Leaf: Class 1, Depth 1
                     Leaf: Class 5, Depth 1
         Node: Gain 0.1875, Dim 251, Threshold -0.8285714387893677, Depth 5
            Leaf: Class 11, Depth 4
            Node: Gain 0.15384617447853088, Dim 208, Threshold 1.0, Depth 4
               Node: Gain 0.15384617447853088, Dim 227, Threshold 0.7165267467498779, Depth 3
                  Leaf: Class 7, Depth 2
                  Leaf: Class 5, Depth 2
               Leaf: Class 0, Depth 3
      Node: Gain 0.12820512056350708, Dim 107, Threshold 0.4285714626312256, Depth 6
         Node: Gain 0.12000003457069397, Dim 107, Threshold -0.4016806185245514, Depth 5
            Leaf: Class 11, Depth 4
            Node: Gain 0.15000000596046448, Dim 237, Threshold -0.1932772397994995, Depth 4
               Leaf: Class 5, Depth 3
               Node: Gain 0.07692308723926544, Dim 115, Threshold -0.3624649941921234, Depth 3
                  Leaf: Class 1, Depth 2
                  Node: Gain 0.08333335816860199, Dim 95, Threshold 0.602241039276123, Depth 2
                     Leaf: Class 6, Depth 1
                     Leaf: Class 5, Depth 1
         Node: Gain 0.142857164144516, Dim 1, Threshold 0.22689075767993927, Depth 5
            Leaf: Class 6, Depth 4
            Node: Gain 0.1666666567325592, Dim 30, Threshold 0.4005602300167084, Depth 4
               Leaf: Class 5, Depth 3
               Leaf: Class 10, Depth 3
CPU times: user 21.9 s, sys: 523 ms, total: 22.5 s
Wall time: 23.2 s
```

In [20]:

```python
print(y_val_01)
print(predictions)
print("the error is : ", zeroOneLoss(predictions,  y_val_01 , jnp.ones_like(y_val_01)))
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11  0  1  2  3  4  5  6  7  8  9 10 11
  0  1  2  3  4  5  6  7  8  9 10 11  0  1  2  3  4  5  6  7  8  9 10 11
  0  1  2  3  4  5  6  7  8  9 10 11  0  1  2  3  4  5  6  7  8  9 10 11
  0  1  2  3  4  5  6  7  8  9 10 11  0  1  2  3  4  5  6  7  8  9 10 11
  0  1  2  3]
[ 8  9  5  5  3  6  7  3  0  8  9  6  6  3  6  0  7  2  2  0  2 10  7  5
  1  4  2  2  0  3  2  0  9  8 11  2  8  3  5  8  0  7  1  4  4  5  5  2
  1  3 10  4  0  2  1  0  3  5  2  8  8  4  6  2  4  2  4  0  9  5  6  7
 11  6  1  7  3  2  5  4  9  5  2  8  6  3  6  2  2  2  6  4  7  5  6  2
  2  0  1  4]
the error is :  0.97
```

In [21]:

```python
print("Accuracy on the 01 validation set:", jnp.mean(predictions == y_val_01)*100 , "%")
```

```
Accuracy on the 01 validation set: 3.0 %
```

In [22]:

```python
%%time
decision_tree = RandomizedDT(percent_dimension=1.0, max_depth= 8 ,  max_size= 20 ,verbose = True)
decision_tree.fit(X_train,y_train)
predictions =decision_tree.predict(X_val)

print("Accuracy on the bigger validation set:", jnp.mean(predictions == y_val)*100 , "%")
print("the error is : ", zeroOneLoss(predictions,  y_val , jnp.ones_like(y_val)))
```

```
Node: Gain 0.08250004053115845, Dim 30, Threshold -0.4285714328289032, Depth 8
   Node: Gain 0.0820244550704956, Dim 1233, Threshold 0.4554622173309326, Depth 7
      Node: Gain 0.07336956262588501, Dim 147, Threshold -0.7142857313156128, Depth 6
         Node: Gain 0.11016947031021118, Dim 1759, Threshold 0.4577031135559082, Depth 5
            Node: Gain 0.08000004291534424, Dim 1701, Threshold 0.46442580223083496, Depth 4
               Node: Gain 0.07377046346664429, Dim 2019, Threshold 0.4711484909057617, Depth 3
                  Node: Gain 0.04672896862030029, Dim 1680, Threshold 0.7422970533370972, Depth 2
                     Node: Gain 0.04166668653488159, Dim 197, Threshold -0.43977591395378113, Depth 1
                        Leaf: Class 8, Depth 0
                        Leaf: Class 6, Depth 0
                     Leaf: Class 4, Depth 1
                  Leaf: Class 1, Depth 2
               Node: Gain 0.1320754587650299, Dim 413, Threshold -0.7467787265777588, Depth 3
                  Node: Gain 0.09756097197532654, Dim 1011, Threshold -0.5630252361297607, Depth 2
```

```
                        Leaf: Class 8, Depth 1
                        Node: Gain 0.11428572237491608, Dim 849, Threshold -0.4151260256767273, Depth 1
                           Leaf: Class 7, Depth 0
                           Leaf: Class 4, Depth 0
                     Leaf: Class 1, Depth 2
               Node: Gain 0.16393443942070007, Dim 865, Threshold 0.7613446712493896, Depth 4
                  Node: Gain 0.13333332538604736, Dim 2034, Threshold 0.16302525997161865, Depth 3
                     Node: Gain 0.1363636553287506, Dim 660, Threshold -0.7154061794281006, Depth 2
                        Leaf: Class 7, Depth 1
                        Leaf: Class 4, Depth 1
                     Leaf: Class 1, Depth 2
                  Node: Gain 0.09677419066429138, Dim 232, Threshold -0.5025209784507751, Depth 3
                     Node: Gain 0.07142858952283859, Dim 791, Threshold -0.7310924530029297, Depth 2
                        Node: Gain 0.0, Dim 1688, Threshold -0.929411768913269, Depth 1
                           Leaf: Class 1, Depth 0
                           Leaf: Class 1, Depth 0
                        Leaf: Class 7, Depth 1
                     Leaf: Class 4, Depth 2
            Node: Gain 0.10606062412261963, Dim 1319, Threshold 0.20672273635864258, Depth 5
               Node: Gain 0.12121212482452393, Dim 1307, Threshold 0.43305325508117676, Depth 4
                  Node: Gain 0.13461539149284363, Dim 163, Threshold -0.20336127281188965, Depth 3
                     Node: Gain 0.10526314377784729, Dim 687, Threshold -0.8397759199142456, Depth 2
                        Leaf: Class 6, Depth 1
                        Node: Gain 0.09375, Dim 963, Threshold -0.33893558382987976, Depth 1
                           Leaf: Class 11, Depth 0
                           Leaf: Class 5, Depth 0
                     Leaf: Class 0, Depth 2
                  Leaf: Class 10, Depth 3
               Node: Gain 0.13636362552642822, Dim 457, Threshold 0.14285719394683838, Depth 4
                  Node: Gain 0.11363634467124939, Dim 153, Threshold -0.7142857313156128, Depth 3
                     Leaf: Class 1, Depth 2
                     Node: Gain 0.08823531866073608, Dim 1847, Threshold -0.3848739564418793, Depth 2
                        Leaf: Class 10, Depth 1
                        Node: Gain 0.08695653080940247, Dim 739, Threshold -0.4621848464012146, Depth 1
                           Leaf: Class 6, Depth 0
                           Leaf: Class 5, Depth 0
                  Node: Gain 0.18181821703910828, Dim 1933, Threshold 0.10588239133358002, Depth 3
                     Leaf: Class 2, Depth 2
                     Leaf: Class 11, Depth 2
         Node: Gain 0.09756100177764893, Dim 1180, Threshold 0.4285714626312256, Depth 6
            Node: Gain 0.12328767776489258, Dim 306, Threshold -0.4285714328289032, Depth 5
               Node: Gain 0.1702127754688263, Dim 788, Threshold 0.7747900485992432, Depth 4
                  Node: Gain 0.12903223931789398, Dim 815, Threshold -0.16974790394306183, Depth 3
                     Leaf: Class 8, Depth 2
                     Node: Gain 0.1111111119389534, Dim 1293, Threshold 0.7624651193618774, Depth 2
                        Node: Gain 0.045454561710357666, Dim 1032, Threshold -0.4420167803764343, Depth 1
                           Leaf: Class 7, Depth 0
                           Leaf: Class 4, Depth 0
                        Leaf: Class 1, Depth 1
                  Leaf: Class 1, Depth 3
               Node: Gain 0.19230768084526062, Dim 1963, Threshold -0.4577030837535858, Depth 4
                  Leaf: Class 6, Depth 3
                  Leaf: Class 11, Depth 3
            Node: Gain 0.09090909361839294, Dim 76, Threshold -0.7154061794281006, Depth 5
               Node: Gain 0.06306305527687073, Dim 918, Threshold -0.0666666105389595, Depth 4
                  Node: Gain 0.24000000953674316, Dim 1540, Threshold 0.7142858505249023, Depth 3
                     Leaf: Class 7, Depth 2
                     Leaf: Class 1, Depth 2
                  Node: Gain 0.04651162028312683, Dim 1642, Threshold -0.13165268301963806, Depth 3
                     Leaf: Class 1, Depth 2
                     Node: Gain 0.03896103799343109, Dim 1052, Threshold 0.16526611149311066, Depth 2
                        Leaf: Class 7, Depth 1
                        Node: Gain 0.027397260069847107, Dim 906, Threshold -0.12941177189350128, Depth 1
                           Leaf: Class 1, Depth 0
                           Leaf: Class 4, Depth 0
               Node: Gain 0.1428571343421936, Dim 1521, Threshold -0.03529411554336548, Depth 4
                  Leaf: Class 6, Depth 3
                  Leaf: Class 3, Depth 3
      Node: Gain 0.09888356924057007, Dim 1199, Threshold 0.4285714626312256, Depth 7
         Node: Gain 0.08222812414169312, Dim 631, Threshold -0.4285714328289032, Depth 6
            Node: Gain 0.08805030584335327, Dim 1572, Threshold 0.4442577362060547, Depth 5
               Node: Gain 0.07751935720443726, Dim 1025, Threshold 0.4285714626312256, Depth 4
                  Node: Gain 0.08571431040763855, Dim 2024, Threshold -0.14957976341247559, Depth 3
                     Node: Gain 0.06578949093818665, Dim 1005, Threshold -0.08123242855072021, Depth 2
                        Node: Gain 0.05882349610328674, Dim 1400, Threshold 0.4442577362060547, Depth 1
                           Leaf: Class 5, Depth 0
                           Leaf: Class 2, Depth 0
                        Leaf: Class 0, Depth 1
                     Node: Gain 0.13793101906776428, Dim 1928, Threshold -0.4935574233531952, Depth 2
                        Leaf: Class 5, Depth 1
                        Node: Gain 0.1363636553287506, Dim 612, Threshold 0.49915969371795654, Depth 1
                           Leaf: Class 0, Depth 0
                           Leaf: Class 9, Depth 0
                  Node: Gain 0.2499999701976776, Dim 139, Threshold -0.20336127281188965, Depth 3
                     Leaf: Class 6, Depth 2
                     Leaf: Class 11, Depth 2
               Node: Gain 0.1000000536441803, Dim 1081, Threshold 0.41736698150634766, Depth 4
                  Node: Gain 0.12499997019767761, Dim 141, Threshold -0.7154061794281006, Depth 3
                     Leaf: Class 6, Depth 2
```

```
                    Leaf: Class 10, Depth 2
                 Leaf: Class 11, Depth 3
           Node: Gain 0.055045902729034424, Dim 115, Threshold -0.14285707473754883, Depth 5
              Node: Gain 0.10606062412261963, Dim 1022, Threshold 0.4868347644805908, Depth 4
                 Node: Gain 0.125, Dim 1113, Threshold -0.32324931025505066, Depth 3
                    Leaf: Class 9, Depth 2
                    Node: Gain 0.1111111044883728, Dim 1316, Threshold -0.6302521228790283, Depth 2
                       Leaf: Class 5, Depth 1
                       Node: Gain 0.09375, Dim 1905, Threshold -0.5361344814300537, Depth 1
                          Leaf: Class 9, Depth 0
                          Leaf: Class 0, Depth 0
                 Leaf: Class 11, Depth 3
              Node: Gain 0.06578946113586426, Dim 1279, Threshold 0.40392160415649414, Depth 4
                 Node: Gain 0.06504064798355103, Dim 1400, Threshold -0.38599440455436707, Depth 3
                    Node: Gain 0.190476194024086, Dim 833, Threshold -0.7142857313156128, Depth 2
                       Leaf: Class 5, Depth 1
                       Leaf: Class 0, Depth 1
                    Node: Gain 0.04901963472366333, Dim 134, Threshold -0.29411765933036804, Depth 2
                       Leaf: Class 5, Depth 1
                       Node: Gain 0.04301074147224426, Dim 1557, Threshold -0.653781533241272, Depth 1
                          Leaf: Class 5, Depth 0
                          Leaf: Class 9, Depth 0
                 Node: Gain 0.2413792908191681, Dim 948, Threshold 0.6156864166259766, Depth 3
                    Leaf: Class 2, Depth 2
                    Leaf: Class 10, Depth 2
        Node: Gain 0.1119999885559082, Dim 1777, Threshold -0.7142857313156128, Depth 6
           Node: Gain 0.07563027739524841, Dim 807, Threshold -0.32773110270500183, Depth 5
              Node: Gain 0.17391303181648254, Dim 1175, Threshold 0.21344542503356934, Depth 4
                 Leaf: Class 2, Depth 3
                 Leaf: Class 10, Depth 3
              Node: Gain 0.06250002980232239, Dim 269, Threshold 0.6806724071502686, Depth 4
                 Node: Gain 0.06024095416069031, Dim 1103, Threshold 0.027451029047369957, Depth 3
                    Leaf: Class 6, Depth 2
                    Node: Gain 0.02816903591156006, Dim 1777, Threshold -0.7176470756530762, Depth 2
                       Node: Gain 0.02816903591156006, Dim 93, Threshold 1.0, Depth 1
                          Leaf: Class 3, Depth 0
                          Leaf: Class 0, Depth 0
                       Leaf: Class 0, Depth 1
                 Leaf: Class 10, Depth 3
           Node: Gain 0.12977105379104614, Dim 1790, Threshold -0.7142857313156128, Depth 5
              Node: Gain 0.19565215706825256, Dim 1214, Threshold 0.7736696004867554, Depth 4
                 Node: Gain 0.14705881476402283, Dim 868, Threshold 0.7971989512443542, Depth 3
                    Node: Gain 0.07692310214042664, Dim 1107, Threshold -0.18319320678710938, Depth 2
                       Node: Gain 0.08695653080940247, Dim 134, Threshold -0.299719899892807, Depth 1
                          Leaf: Class 5, Depth 0
                          Leaf: Class 2, Depth 0
                       Leaf: Class 10, Depth 1
                    Leaf: Class 3, Depth 2
                 Leaf: Class 10, Depth 3
              Node: Gain 0.08235293626785278, Dim 61, Threshold -0.4285714328289032, Depth 4
                 Leaf: Class 3, Depth 3
                 Node: Gain 0.057971030473709106, Dim 1671, Threshold -0.7859944105148315, Depth 3
                    Leaf: Class 10, Depth 2
                    Node: Gain 0.052631571888923645, Dim 35, Threshold -0.4285714328289032, Depth 2
                       Leaf: Class 6, Depth 1
                       Node: Gain 0.056603774428367615, Dim 1373, Threshold -0.26050421595573425, Depth 1
                          Leaf: Class 9, Depth 0
                          Leaf: Class 11, Depth 0
Accuracy on the bigger validation set: 36.0 %
the error is :  0.64
CPU times: user 1min, sys: 3.24 s, total: 1min 3s
Wall time: 60 s
```

Analyze your results in this box. Achieving 36% accuracy on the validation set is a promising start, and while it's a decent result for now, there's room for improvement. We'll aim to enhance this by implementing Random Forests and Boosting techniques. Personally, I am satisfied with this initial outcome. Additionally, using vmap has significantly sped up our computations, enabling us to reach these results in just 1 minute and 30 seconds, which is quite efficient.

Q4. Use cross-validation on the full digit dataset (0-9) to select a reasonnable depth between 2 and 8, using random splits of half the training set to save on training time. *(Indicative time: maximum 10 minutes to code, about 20 minutes to run)*

In [ ]:

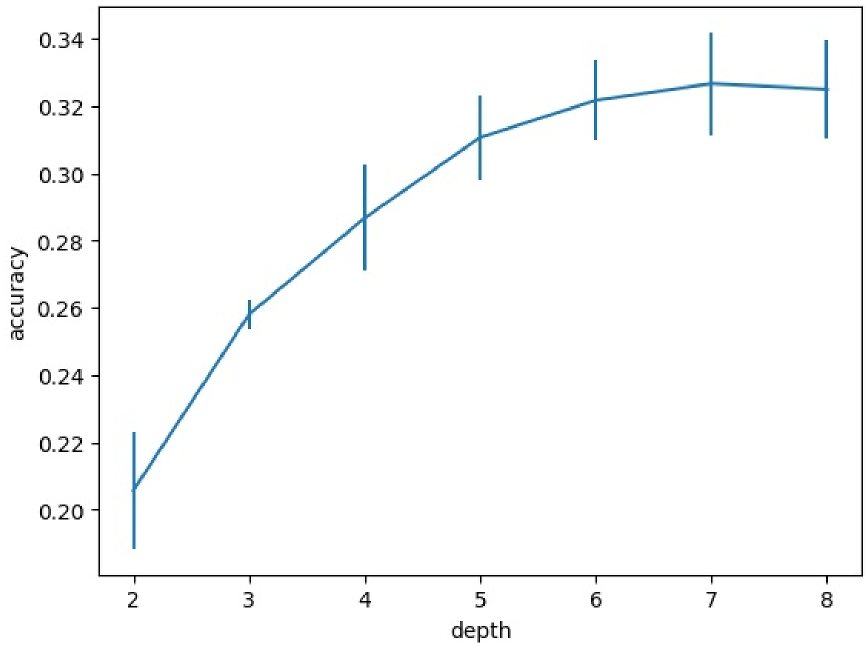
```
def cross_validation_tree_depths(X_train, y_train, depths, splits=5, subset_size=0.5):
    avg_accuracies = []
    deviations = []
    for current_depth in depths:
        fold_accuracies = []
        for i in range(splits):
            # Randomly shuffle indices for cross-validation
            key,skey = jax.random.split(jax.random.key(i))
            perm_indices = jax.random.permutation(skey, len(y_train))
            cut_off = int(len(y_train) * subset_size)

            indices_train = perm_indices[:cut_off]
```

```
            indices_val = perm_indices[cut_off:2*cut_off]

            X_sub_train = X_train[indices_train]
            y_sub_train = y_train[indices_train]
            X_sub_val = X_train[indices_val]
            y_sub_val = y_train[indices_val]

            # Initialize and train the decision tree at the current depth
            decision_tree = RandomizedDT(max_depth=current_depth, verbose=False)
            decision_tree.fit(X_sub_train, y_sub_train)

            # Evaluate the decision tree
            predicted = decision_tree.predict(X_sub_val)
            accuracy = jnp.mean(predicted == y_sub_val)
            fold_accuracies.append(accuracy)

        accuracies_np = jnp.array([fold_accuracies])
        avg_accuracies.append(jnp.mean(accuracies_np))
        deviations.append(jnp.std(accuracies_np))

    # Determine the optimal tree depth
    optimal_depth = depths[jnp.argmax(jnp.array(avg_accuracies))]

    return optimal_depth, avg_accuracies, deviations
```

In [ ]:
```
%%time

depths = range(2, 9)
optimal_depth, avg_accuracies, deviations = cross_validation_tree_depths(X_train, y_train, depths)
```

```
CPU times: user 6min 26s, sys: 1min 5s, total: 7min 31s
Wall time: 6min 4s
```

In [ ]:
```
plt.errorbar(depths, avg_accuracies, yerr= deviations)
plt.xlabel("depth")
plt.ylabel("accuracy")
plt.show()

print("Best depth:", optimal_depth)
```




```
Best depth: 7
```

In [ ]:
```
print(f"the Optimal Depth is : {optimal_depth}")
print()
```

```
the Optimal Depth is : 7
```

Analyze your results in this box. The accuracy curve as a function of tree depth initially increases, indicating improved learning, but then it begins to decrease, signaling overfitting with deeper trees. This overfitting occurs because the model becomes too complex, capturing noise in the training data rather than generalizing from it. Conversely, insufficient depth leads to underfitting, where the model is too simplistic to capture the underlying patterns. The optimal depth, in our case, has been determined to be 7, balancing complexity and generalization. Cross-validation completed faster than anticipated, and training on half of the data proved to be effective, providing a good balance between training time and model performance.

Random Forest

Next, we want to mitigate the tendancy of decision trees to overfit when the depth is too high and to underfit when the depth is too small by implementing random forests.

Q5. Code a Random Forest of decision trees, each trained on a subset of the training set. Perform a corase cross-validation to set a reasonnable number of trees (25, 50, 75), percent of training data used (0.5, 0.75), percent of dimensions used (0.5, 0.75) and depth 3. *(Indicative time: less than 20 minutes to code, takes more than 20 minutes to run)*

In [ ]:
```
class RandomForest():
    def __init__(self, nb_trees = 10 , percent_dataset= 0.75, percent_dimension= 0.5, max_depth=8):
        self.nb_trees = nb_trees
        self.percent_dataset = percent_dataset
        self.percent_dimension = percent_dimension
        self.max_depth = max_depth
        self.trees = []

    def fit(self, X, y):
        n_samples = X.shape[0]
        sample_size = int(n_samples * self.percent_dataset)
        k = int(X.shape[1] * self.percent_dimension)

        for i in range(self.nb_trees):
            # Randomly selecting data samples and features
            key, skey = jax.random.split(jax.random.key(i))
            idxs = jax.random.choice(skey, n_samples, shape=(sample_size,), replace=False)
            # Creating a new tree
            tree = RandomizedDT(percent_dimension= self.percent_dimension, max_depth=self.max_depth , rng_key
            tree.fit(X[idxs,:],y[idxs])
            self.trees.append(tree)

    def predict(self, X):
        y = []
        for dt in self.trees:
            y.append(dt.predict(X))
        y = jax.nn.one_hot(jnp.array(y), num_classes=12)
        return y.sum(axis=0).argmax(axis=1)
```

In [ ]:
```
def coarse_cross_validation(X, y,X_val , y_val , nb_trees_options, percent_dataset_options, percent_dimension_
    best_score = float('inf') # juste 2 is enough
    best_params = None
    for nb_trees in nb_trees_options:
        for percent_dataset in percent_dataset_options:
            for percent_dimension in percent_dimension_options:
                for depth in depths:
                    rf = RandomForest(nb_trees, percent_dataset, percent_dimension, depth)
                    rf.fit(X, y)
                    predictions = rf.predict(X_val)
                    score = zeroOneLoss(predictions,  y_val , jnp.ones_like(y_val))
                    if score < best_score:
                        best_score = score
                        best_params = (nb_trees, percent_dataset, percent_dimension, depth)
                    print(f"Trees: {nb_trees}, Data%: {percent_dataset}, Dim%: {percent_dimension}, Depth: {de

    return best_params
```

In [ ]:
```
%%time
best_params = coarse_cross_validation(X_train, y_train, X_val , y_val  , [25, 50, 75], [0.5, 0.75], [0.5, 0.75
```

```
Trees: 25, Data%: 0.5, Dim%: 0.5, Depth: 3, Error : 0.6466666460037231
Trees: 25, Data%: 0.5, Dim%: 0.75, Depth: 3, Error : 0.6499999761581421
Trees: 25, Data%: 0.75, Dim%: 0.5, Depth: 3, Error : 0.6366666555404663
Trees: 25, Data%: 0.75, Dim%: 0.75, Depth: 3, Error : 0.6399999856948853
Trees: 50, Data%: 0.5, Dim%: 0.5, Depth: 3, Error : 0.6000000238418579
Trees: 50, Data%: 0.5, Dim%: 0.75, Depth: 3, Error : 0.6066666841506958
Trees: 50, Data%: 0.75, Dim%: 0.5, Depth: 3, Error : 0.6433333158493042
Trees: 50, Data%: 0.75, Dim%: 0.75, Depth: 3, Error : 0.6333333253860474
Trees: 75, Data%: 0.5, Dim%: 0.5, Depth: 3, Error : 0.6033333539962769
Trees: 75, Data%: 0.5, Dim%: 0.75, Depth: 3, Error : 0.6266666650772095
Trees: 75, Data%: 0.75, Dim%: 0.5, Depth: 3, Error : 0.6366666555404663
Trees: 75, Data%: 0.75, Dim%: 0.75, Depth: 3, Error : 0.6433333158493042
CPU times: user 1h 6min 34s, sys: 8min 54s, total: 1h 15min 28s
Wall time: 1h 2min 26s
```

Analyzing the results with a small number of trees in our Random Forest, we observed a decrease in error compared to previous models, indicating that the method has partially achieved our initial goals. However, the error rate remains above 0.6, suggesting that this approach might not be sufficient on its own. I am hopeful that integrating boosting will further enhance the performance. The best hyperparametres we got using the cross-validation are : Trees: 50, Data%: 0.5, Dim%: 0.5, Depth: 3.
Which gave an error of 0.600.

Boosting

To have a more efficient training procedure, we will remove the independance between the trees by using boosting

Q6. Code the BoostingClassifier that obtains a combination of Randomized Trees using AdaBoost. Each tree is trained using the weighted $0-1$ loss. To allow the tree combination, convert the output of each tree to a one-hot encoded vector. The output of the boosted trees is then the weighted sum of these one-hot vectors and the predicted class is the argmax. Test with the same parameters as the best Random Forest. *(Indicative time: about 30 minutes to code, runs about as fast as a random forest)*

In [ ]:

```
class BoostedTrees():
    def __init__(self, nb_trees, percent_dataset=1., percent_dimension=1., max_depth=8):
        self.nb_trees = nb_trees
        self.percent_dataset = percent_dataset
        self.percent_dimension = percent_dimension
        self.max_depth = max_depth
        self.trees = []
        self.tree_weights = []

    def fit(self, X, y):
        n_samples = X.shape[0]
        sample_weights = jnp.ones(n_samples)
        num_classes = len(jnp.unique(y))

        for i in range(self.nb_trees):
            # Randomly select data samples based on weights
            key, skey = jax.random.split(jax.random.key(i))
            idxs = jax.random.choice(key, n_samples, shape=(n_samples,), p=sample_weights , replace =True)
            tree = RandomizedDT(percent_dimension= self.percent_dimension, max_depth=self.max_depth , rng_key
            tree.fit(X[idxs], y[idxs], sample_weights[idxs])
            self.trees.append(tree)

            # Evaluate errors and calculate tree weight
            pred = tree.predict(X)
            incorrect = jnp.not_equal(pred, y)#.astype(jnp.float32)
            error = jnp.sum(sample_weights * incorrect) / jnp.sum(sample_weights) #or just zeroOneLoss
            alpha = jnp.log((1 - error) / (error + 1e-10)) + jnp.log(num_classes - 1)
            self.tree_weights.append(alpha)

            # Update weights
            sample_weights *= jnp.exp(alpha * incorrect)
            sample_weights /= jnp.sum(sample_weights)


    def predict(self, X):
        tree_preds = jnp.array([tree.predict(X) for tree in self.trees])
        # Convert predictions to one-hot vectors
        tree_preds_onehot = jax.nn.one_hot(tree_preds, num_classes=12)
        final_preds = jnp.tensordot(tree_preds_onehot, jnp.array(self.tree_weights), axes=[0, 0])
        # Argmax over weighted sums for final prediction
        return jnp.argmax(final_preds, axis=-1)
```

In [ ]:

```
nb_trees, percent_dataset, percent_dimension, max_depth = best_params
best_trained_tree = BoostedTrees(nb_trees, percent_dataset, percent_dimension, max_depth )
best_trained_tree.fit(X_train, y_train)
predictions = best_trained_tree.predict(X_val)
```

In [ ]:

```
print("the error is : ", zeroOneLoss(predictions,  y_val , jnp.ones_like(y_val)))
print("Accuracy on the bigger validation set:", jnp.mean(predictions == y_val)*100 , "%")
```

```
the error is :  0.51666665
Accuracy on the bigger validation set: 48.333336 %
```

Analyze your results in this box. Boosting, with RF best hyperparams, helps ameliorating the performance significantly as we can see above.
We've now an error of just 51% instead of 60% found with Random Forest.

Visualization

In order to visualize the decision, we can produce an image the contains only the relevant information with respect to the decisions taken by a tree.

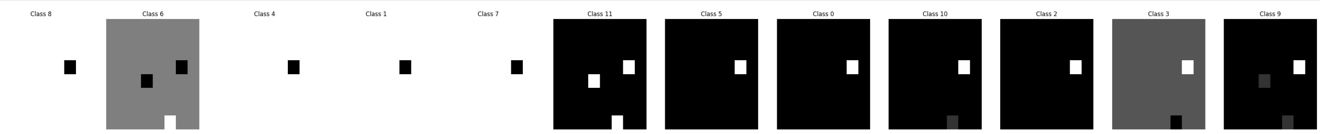
Q7. For a trained tree, select a leaf and build an image that has a value of 1 for each pixel in the decision path that should be above the threshold, 0 for each pixels in the decision path that should be below the threshold and 0.5 everywhere else. For all classes, show an average all such images for each leaf corresponding to that class. *(Indicative time: about one hour to code)*

In [ ]:

```
class DecisionVisualization():
```

```
In [ ]: class DecisionVisualization():
    def __init__(self, tree, num_features):
        self.tree = tree
        self.num_features = num_features  # Total number of features

    def trace_paths(self, node, current_path=[]):
        """Recursively collect paths to all leaves starting from the given node."""
        if node is None:
            return []
        if node.label is not None:  # It's a leaf node
            return [(int(node.label), current_path)]
        # Continue tracing down the tree
        left_path = self.trace_paths(node.left, current_path + [(node.dim, 'left', node.th)])
        right_path = self.trace_paths(node.right, current_path + [(node.dim, 'right', node.th)])
        return left_path + right_path

    def build_decision_images(self, paths):
        """Build decision images from traced paths."""
        decision_images = {label: [] for label, _ in paths}
        for label, path in paths:
            image = jnp.full(self.num_features, 0.5)  # Start with 0.5 for all features
            for dim, direction, th in path:
                # Update image based on direction and threshold
                if direction == 'left':
                    # Feature values for 'left' are below the threshold
                    image = image.at[dim].set(0)
                else:
                    # Feature values for 'right' are above the threshold
                    image = image.at[dim].set(1)
            decision_images[label].append(image)
        return decision_images

    def average_class_images(self, decision_images):
        """Average decision images for each class."""
        class_images = {}
        for label, images in decision_images.items():
            if images:
                class_images[label] = jnp.mean(jnp.stack(images), axis=0)
        return class_images

    def visualize_decisions(self):
        """Generate and average class images from decision paths."""
        paths = self.trace_paths(self.tree)
        decision_images = self.build_decision_images(paths)
        return self.average_class_images(decision_images)

    def plot_decision_images(self):
        """Plot the averaged decision images for each class."""
        class_images = self.visualize_decisions()
        num_classes = len(class_images)
        fig, axes = plt.subplots(1, num_classes, figsize=(num_classes * 4, 4))
        if num_classes == 1:
            axes = [axes]  # Make it iterable for consistent indexing
        for i, (label, image) in enumerate(class_images.items()):
            ax = axes[i]
            ax.imshow(image.reshape((int(self.num_features**0.5), -1)), cmap='gray', aspect='auto')
            ax.set_title(f'Class {label}')
            ax.axis('off')
        plt.show()
```

```
In [ ]: visualization = DecisionVisualization(decision_tree, num_features=64)
visualization.plot_decision_images()
```



Analyze your results in this box. Answer

Processing math: 100%