In [1]:
```
%load_ext autoreload
%autoreload 2
import tme3
```

Apprentissage de paramètres par maximum de vraisemblance

Dans ce TME, l'objectif est d'apprendre grâce à l'estimateur de maximum de vraisemblance les paramètres de lois normales à partir d'un ensemble de données. Ces lois normales seront ensuite exploitées pour faire de la classification (comme nous l'avions vu en cours avec les images de désert, forêt, mer et paysages enneigés).

Ici, notre base de données d'apprentissage est la base USPS. Celle-ci contient les images réelles de chiffres provenant de codes postaux écrits manuellement et scannés par le service des postes américain. Ces données scannées ont été normalisées de manière à ce qu'elles soient toutes des images de 16x16 pixels en teintes de gris, cf. Le Cun et al., 1990:

Y. LeCun, O. Matan, B. Boser, J. S. Denker, et al. (1990) *Handwritten zip code recognition with multilayer networks*. In ICPR, volume II, pages 35–40.

Voici quelques exemples d'images de cette base :

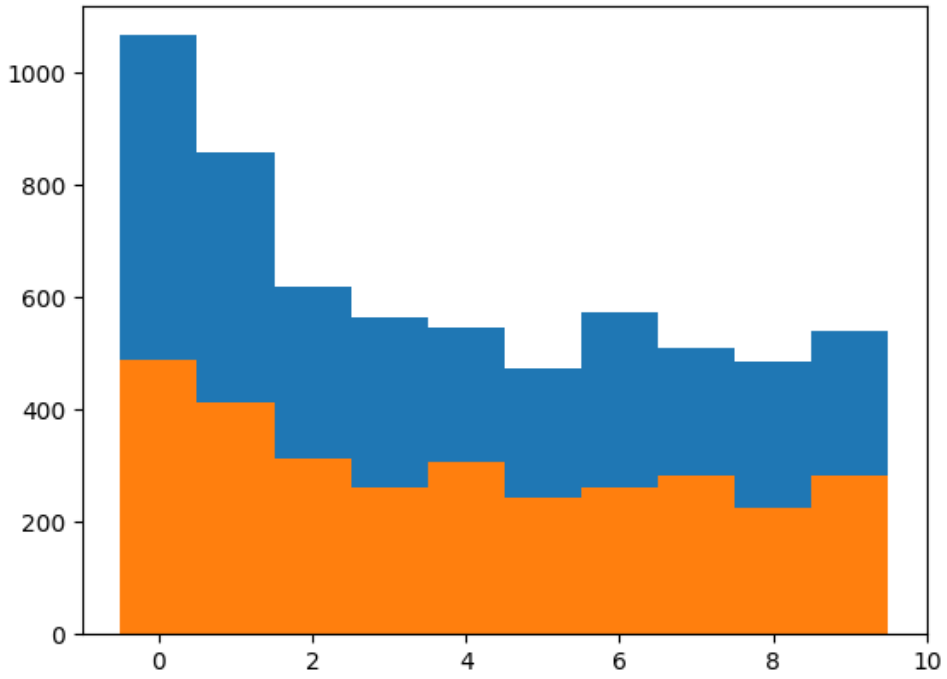
In [2]:
```
import numpy as np
import matplotlib.pyplot as plt
import pickle as pkl
```

Chargement des données et premières visualisations

Nous utiliserons la librairie pickle qui permet de sérialiser les objets en python (ie, les sauver et les charger très facilement). Une fois les données chargées, nous allons étudier très rapidement la distribution des classes, visualiser une imagette de chiffre et comprendre l'encodage de ces chiffres.

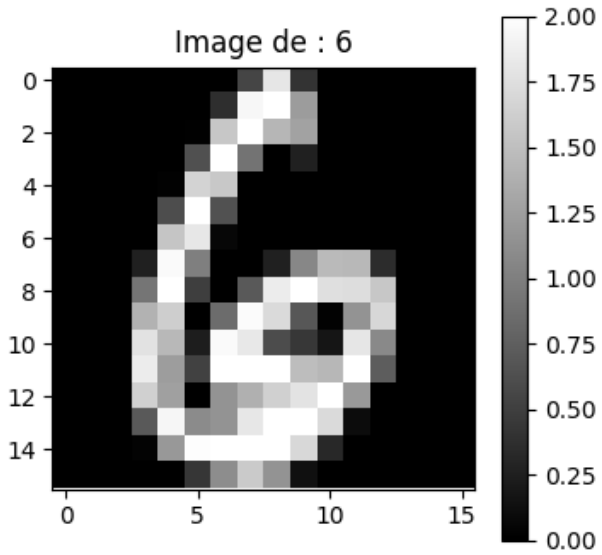
In [3]:
```
# Chargement des données
data = pkl.load(open("res/usps.pkl",'rb'))
# data est un dictionnaire contenant les champs explicites X_train, X_test, Y_train, Y_test
X_train = np.array(data["X_train"],dtype=float)
X_test = np.array(data["X_test"],dtype=float)
Y_train = data["Y_train"]
Y_test = data["Y_test"]

# visualisation de la distribution des étiquettes (dans les 10 classes de chiffres)
plt.figure()
plt.hist(Y_train, np.linspace(-0.5,9.5,11))
plt.hist(Y_test, np.linspace(-0.5,9.5,11))
#plt.savefig("distr_classes.png")
```

Out[3]:
```
(array([488., 412., 311., 260., 306., 244., 261., 282., 224., 281.]),
 array([-0.5,  0.5,  1.5,  2.5,  3.5,  4.5,  5.5,  6.5,  7.5,  8.5,  9.5]),
 <BarContainer object of 10 artists>)
```



In [4]:
```
# prise en main des matrices X, Y
print(X_train.shape)
# 6229 images composées de 256 pixels (image = 16x16)
print(X_test.shape,Y_train.shape, Y_test.shape)

# Affichage de l'image 18 de la base de données et récupération de l'étiquette associée:
# (1) remise en forme de la ligne de 256 pixels en 16x16
# (2) affichage avec imshow (en niveaux de gris)
# (3) récupération de l'étiquette dans Y_train

def show_digit(title,tab):
    plt.figure(figsize=(4,4))
    plt.imshow(tab.reshape(16,16),cmap="gray")
    plt.colorbar()
    plt.title(title)

show_digit("Image de : {}".format(Y_train[18]),X_train[18])
```

```
(6229, 256)
(3069, 256) (6229,) (3069,)
```



In [5]:
```
# analyse des valeurs min et max, recherche du nombre de niveaux de gris dans les images:
print(X_train.min(),X_train.max() )
print("niveaux de gris : ", len(np.unique(X_train)))
```

```
0.0 2.0
niveaux de gris :  2001
```

A. Apprentissage et évaluation d'un modèle gaussien naïf

A1- Maximum de vraisemblance

Nous allons étudier la distribution de probabilité des teintes de gris des images (en fait, nous allons étudier sa fonction de densité car on travaille sur des variables aléatoires continues) . Nous allons faire l'hypothèse (certes un peu forte mais tellement pratique) que, dans chaque classe, les teintes des pixels sont mutuellement indépendantes.

Autrement dit, si $X_i, i\in \{0,...,255\}$ représente la variable aléatoire "intensité de gris du ième pixel", alors $p(X_0,...,X_{255})$ représente la fonction de densité des teintes de gris des images de la classe et:

$$p(X_0,,...,X_{255})=\prod_{i=0}^{255} p(X_i)$$

Ainsi, en choisissant au hasard une image dans l'ensemble de toutes les images possibles de la classe, si celle-ci correspond au tableau `np.array([x_0,...,x_255])` , où les x_i sont des nombres réels compris entre 0 et 2, alors la valeur de la fonction de densité de l'image est égale à $p(x_0,...,x_{255}) = \prod^{255}_{i=0}p(x_i)$.

Nous allons de plus supposer que chaque X_i suit une distribution normale de paramètres $(μ_i,σ^2_i)$. Autrement dit, $$\forall i\in\{0,...,255\}, X_i \sim {\cal N}(μ_i,σ^2_i)$$

Par maximum de vraisemblance, estimez, pour une classe donnée, l'ensemble des paramètres $(μ_0,...,μ_{255})$ et $(σ^2_0,...,σ^2_{255})$ pour chaque classe (chiffre de 0 à 9). Pour cela, écrivez une fonction `learnML_parameters : float np.array x float np.array -> float np.array x float np.array` qui, étant donné le tableau d'images d'une classe tel que retourné par la fonction read_file (autrement dit un tableau de tableaux de 256 nombres réels), renvoie un couple de tableaux, le premier élément du couple correspondant à l'ensemble des $μ_i$ et le 2ème à l'ensemble des $σ^2_i$. C'est-à-dire que `learnML_class_parameters ( classe )` renverra deux matrices: $$ mu \in \mathbb R^{10 \times 256}, sig \in \mathbb R^{10 \times 256}$$

- mu contient les moyennes des 256 pixels pour les 10 classes
- std contient les écarts-types des 256 pixels pour les 10 classes

In [6]:
```
mu,sig = tme3.learnML_parameters ( X_train,Y_train )
print(mu.shape, sig.shape) # doit donner (10, 256) (10, 256)
```

```
(10, 256) (10, 256)
```

In [7]:
```
print(f"{mu[0][:10]=}\n{sig[0][:10]=}")
```

```
mu[0][:10]=array([0.00153774, 0.00446786, 0.01712161, 0.0631194 , 0.18406164,
       0.47139167, 0.89764099, 1.15019928, 1.020709  , 0.61678541])
sig[0][:10]=array([0.05015963, 0.07936951, 0.14648902, 0.26552234, 0.4423062 ,
       0.635148  , 0.7404621 , 0.74838703, 0.75203696, 0.67816278])
```

A2- Log-vraisemblance d'une image pour une classe

Écrivez une fonction `log_likelihood : float np.array x float np.array x float np.array -> float` qui, étant donné une image (donc un tableau de 256 nombres réels) et un couple de paramètres `( array ( [μ0,…,μ255] ), array ( [σ20,…,σ255] ) )` , renvoie la log-vraisemblance qu'aurait l'image selon cet ensemble de $μ_i$ et $σ_i$ (correspondant à une classe de chiffre). Rappelez-vous que (en mettant $-\frac{1}{2}$ en facteur) :

$$\log(p(x_0,\cdots,x_{255})=\sum_{i=0}^{255} \log p(x_i)=-\frac{1}{2}\sum_{i=0}^{255} \left[\log(2\pi

\sigma^2_i) + \frac{(x_i-\mu_i)^2}{\sigma_i^2} \right]$$

Notez que le module `np` contient une constante `np.pi` ainsi que toutes les fonctions mathématiques classiques directement applicables sur des vecteurs. Vous pouvez donc éventuellement coder la ligne précédente sans boucle, en une ligne.

Attention: dans la matrice `sig` calculée dans la question précédente, pour certains pixels de certaines classes, la valeur de $σ^2$ est égale à 0 (toutes les images de la base d'apprentissage avaient exactement la même valeur sur ce pixel).

- cette valeur pose problème dans le calcul précédent (division par 0)
- Réfléchir à différente manière de traiter ce problème:
- faible valeur par défaut de σ reflétant une variance très faible mais évitant la division par 0 (usage de `np.maximum` par exemple)
- vraisemblance de 1 pour le ou les pixels impactés

Attention (2) on utilisera dans le suite le paramètre $defeps$:

- $defeps>0$, il donne la valeur minimale d'écart type
- $degeps==-1$, il faut prendre une vraisemblance de 1 pour les pixels concernés

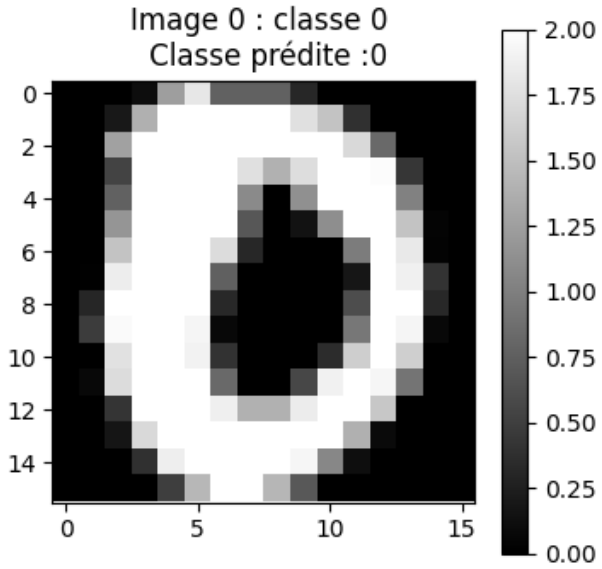
In [8]:
```
print("vraisemblance de l'image 0 selon les paramètres de la classe 0 avec defeps=1e-5")
print(tme3.log_likelihood(X_train[0], mu[0], sig[0],1e-5))

print("")

print("vraisemblance de l'image 0 pour toutes les classes avec defeps=-1")
print([tme3.log_likelihood(X_train[0], mu[i], sig[i],-1) for i in range(10)])

print("")

print("vraisemblance de l'image 0 pour toutes les classes avec defeps=1e-5")
print([tme3.log_likelihood(X_train[0], mu[i], sig[i],1e-5) for i in range(10)])
```

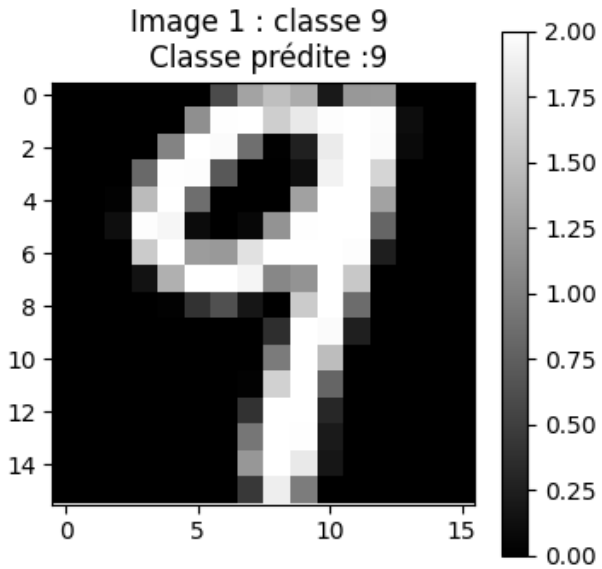
```
vraisemblance de l'image 0 selon les paramètres de la classe 0 avec defeps=1e-5
-90.69963035168726

vraisemblance de l'image 0 pour toutes les classes avec defeps=-1
[-111.88760421521835, -1716629080.989729, -364.83171019852006, -487.01085544875855, -544.910025540
4516, -387.7594698419803, -59747.8395637312, -581523.2639945432, -303.762503411686, -13497.8259109
16881]

vraisemblance de l'image 0 pour toutes les classes avec defeps=1e-5
[-90.69963035168726, -231211311074.5327, -364.8317101985202, -487.01085544875843, -513.12806474515
5, -387.75946984198, -59610.117733618186, -75567222244.77489, -271.980542616389, -857252055.477422
1]
```

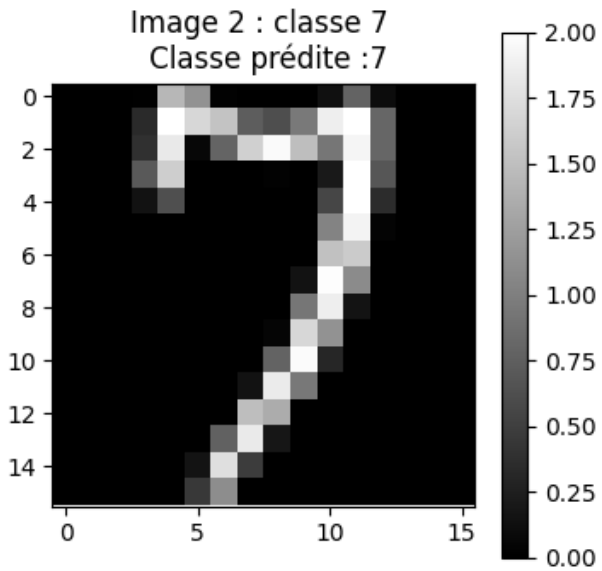
A3- Classification d'une image

Écrire une fonction `classify_image : float np.array x float np.array x float np.array -> int` qui, étant donnée une image et l'ensemble de paramètres déterminés dans les questions précédentes, renvoie la classe la plus probable de l'image, c'est-à-dire celle dont la log-vraisemblance est la plus grande.

In [9]:
```
for i in range(3):
    show_digit(f"Image {i} : classe {Y_train[i]} \n Classe prédite :{tme3.classify_image(X_train[
```







A4- Classification de toutes les images

Écrire une fonction `classify_all_images : float np.array x float np.array x float np.array -> float np.array` qui, étant donné un tableau X des images ($N \times 256$) et l'ensemble de paramètres déterminés dans les questions précédentes, renvoie un tableau $\hat Y$ qui donne la prédiction

de classe pour toutes les images

In [10]:
```
Y_train_hat = tme3.classify_all_images(X_train, mu, sig, -1)

print(Y_train_hat)
```

```
[0 9 7 ... 6 3 2]
```

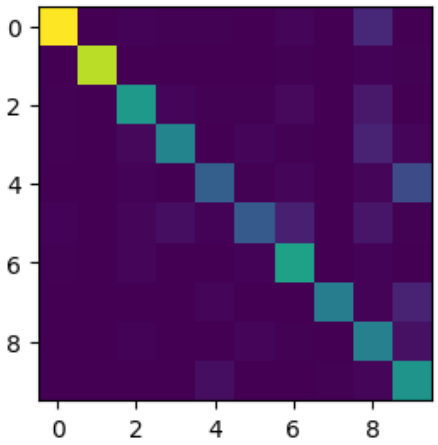
A5-Matrice de confusion et affichage du résultat des classifications

La matrice de confusion est de la forme $C \times C$ où C est le nombre de classe. Les lignes sont les vraies classes, les colonnes sont les classes prédites. Chaque case (i,j) contient le nombre d'images correspondant à la vraie classe i et à la prédiction j. Si votre classifieur est performant, vous devriez observer des pics sur la diagonale.

La fonction `matrice_confusion(Y, Y_hat)` prend en argument un vecteur d'étiquettes réelles et un vecteur de même taille d'étiquettes prédites et retourne la matrice de confusion.

Vous devriez obtenir une matrice de la forme:

In [11]:
```
# affichage de la matrice de confusion
m = tme3.matrice_confusion(Y_train, Y_train_hat)


plt.figure(figsize=(3,3))
plt.imshow(m);
```



Écrire une fonction `ClassificationRate(Y_train,Y_train_hat)` qui, à partir de ces 2 tableaux, calcule le taux de bonne prédiction

In [12]:
```
print(f"Taux de bonne classification: {tme3.classificationRate(Y_train,Y_train_hat)=}")
```

```
Taux de bonne classification: tme3.classificationRate(Y_train,Y_train_hat)=0.8099213356879114
```

A6- Ensemble d'apprentissage, ensemble de test

Dans la procédure que nous avons suivie jusqu'ici, nous avons triché. Les mêmes données servent à apprendre les paramètres et à évaluer le modèle. Evidemment, le modèle est parfaitement adapté et les performances sur-estimées.

Afin de réduire ce biais, nous allons maintenant évaluer les performances sur les données de test. Les performances devraient être plus basses... Mais plus réalistes.

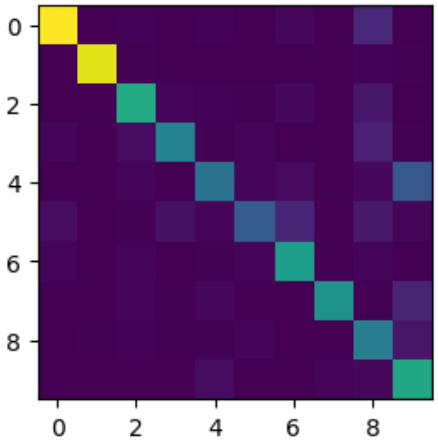
Effectuer ces calculs et afficher le taux de bonne classification et la matrice de confusion.

Attention: il faut donc utiliser les paramètres appris sur de nouvelles données sans réapprendre des paramètres spécifiques sinon ça ne marche pas

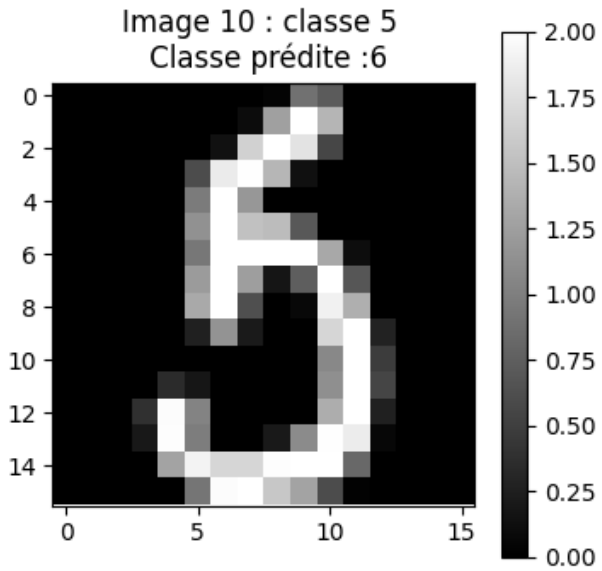
Afin de mieux comprendre les erreurs (et de vérifier vos connaissances sur numpy): la fonction retournera la liste des images mal classées. Afficher 3 images de chiffre mal classées, leurs étiquette prédite et leurs étiquette réelle. Normalement, vous devez retrouver automatiquement que le premier chiffre mal classé est l'image 10:

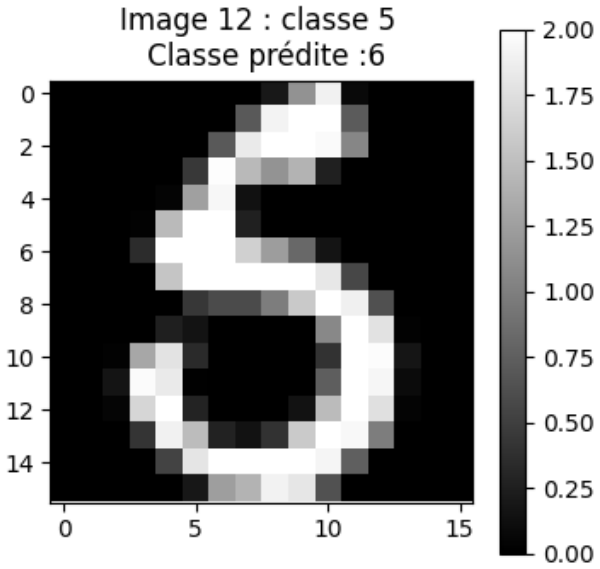
In [13]:
```
mal_classees=tme3.classifTest(X_test,Y_test,mu,sig,-1)
```

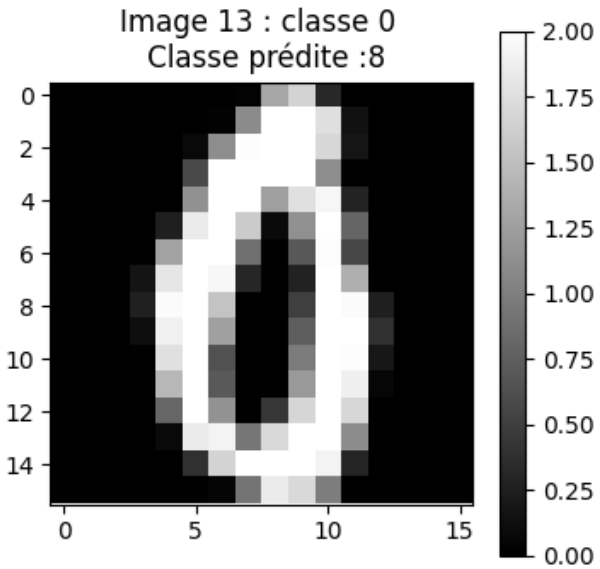
```
1- Classify all test images ...
2- Classification rate :  0.7934180514825676
3- Matrice de confusion :
```



In [14]:
```
for ind in mal_classees[0][:3]:
    show_digit(f"Image {ind} : classe {Y_test[ind]} \n Classe prédite :{tme3.classify_image(X_tes
```







Autres modélisations possibles pour les images

B. Modélisation par une loi de Bernoulli

Soit les indices i donnant les images et les indices j référant aux pixels dans l'image, nous cherchons à déterminer la probabilité d'illumination d'un pixel j pour une collection d'image (d'une seule classe, par exemple les 0).

Collection de 0: $$ X = \{\mathbf{x_i}\}_{i = 1,\ldots, N}, \qquad \mathbf{x_i} \in \{0,1\}^{256}$$

Modélisation de la variable de Bernoulli X_j, valeur du pixel j en écriture factorisée: $$ p(X_j = x_{ij}) = p_j^{x_{ij}} (1-p_j)^{(1-x_{ij})} = \left\{ \begin{array}{ccc} p_j & \mbox{ si } x_{ij} = 1 \\ 1-p_j & \mbox{ si } x_{ij} = 0 \\ \end{array} \right.$$

Expression de la vraisemblance

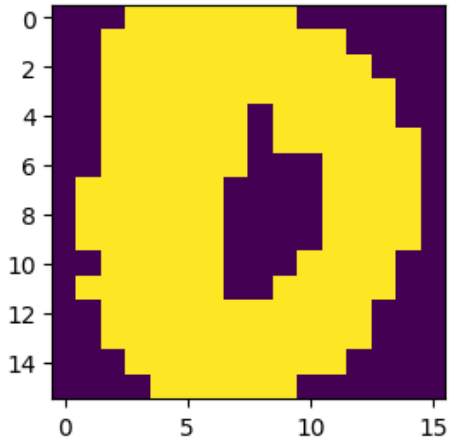
Maximisation de la vraisemblance $\Rightarrow$ $\nabla_{\theta} \mathcal L(X, \theta) = 0$:

$$p_j^\star = \frac{\sum_i x_{ij}}{N} $$

Intuitif: nombre de 1 pour le pixel j divisé par le nombre d'image = pourcentage d'illumination du pixel j

In [15]:
```
# binarisation des images pour coller avec Bernoulli:

Xb_train = tme3.binarisation(X_train)
Xb_test  = tme3.binarisation(X_test)

# affichage d'une image binaire:
plt.figure(figsize=(3,3))
plt.imshow(Xb_train[0].reshape(16,16))
```

Out[15]:
```
<matplotlib.image.AxesImage at 0x1cda46a55d0>
```



B-1: Ecrire la fonction d'apprentissage des paramètres qui retourne la matrice theta suivante:

$$ \theta^\star = \left[\begin{array}{ccc} [p_0^\star, \ldots, p_{255}^\star] & \mbox{Paramètres optimaux de la classe 0 au sens du max de vraisembalnce} \\ [p_0^\star, \ldots, p_{255}^\star] & \mbox{Paramètres optimaux de la classe 1 au sens du max de vraisembalnce} \\ \vdots & \\ [p_0^\star, \ldots, p_{255}^\star] & \mbox{Paramètres optimaux de la classe 9 au sens du max de vraisembalnce} \\ \end{array} \right] $$

Il faut ensuite calculer les : $$ \log p (\mathbf{x_i} | \theta^{(c)}) = \sum_j \log p(X_j = x_{ij}) = \sum_j {x_{ij}} \log p_j + {(1-x_{ij})}\log(1-p_j)$$

Faire passer les N images dans les C modèles donne un tableau de la forme : $$ \log p (X | \theta) = \left[\begin{array}{cccc} \log p (\mathbf{x_0} | \theta^{(0)}) & \log p (\mathbf{x_0} | \theta^{(1)}) & \ldots & \log p (\mathbf{x_0} | \theta^{(9)}) \\ & \vdots & \\ \log p (\mathbf{x_N} | \theta^{(0)}) & \log p (\mathbf{x_N} | \theta^{(1)}) & \ldots & \log p (\mathbf{x_N} | \theta^{(9)}) \\ \end{array} \right] $$

Chaque ligne donne pour une image sa probabilité d'appartenance à chaque classe c. Un argmax par ligne donne une estimation de la classe.

In [16]:
```
theta = tme3.learnBernoulli ( Xb_train,Y_train )
print(theta.shape)
print(theta)
```

```
(10, 256)
[[0.00093897 0.00657277 0.03192488 ... 0.02347418 0.00375587 0.        ]
 [0.         0.         0.         ... 0.00233372 0.         0.        ]
 [0.01941748 0.05987055 0.13430421 ... 0.27993528 0.20711974 0.11326861]
 ...
 [0.06666667 0.16078431 0.2745098  ... 0.         0.         0.        ]
 [0.01033058 0.05371901 0.1322314  ... 0.01446281 0.00206612 0.        ]
 [0.0037037  0.0037037  0.01111111 ... 0.00555556 0.00185185 0.        ]]
```

B-2: Écrire ensuite une fonction de calcul de la vraisemblance d'une image par rapport à ces paramètres

** Attention ** $log(0)$ n'est pas défini et $log(1-x)$ avec $x=1$ non plus ! La solution à ce problème est assez simple, il suffit de seuiller les probabilités d'illumination entre $\epsilon $ et $1-\epsilon$.

In [17]:
```
tme3.logpobsBernoulli(X_train[0], theta,epsilon=1e-4)
```

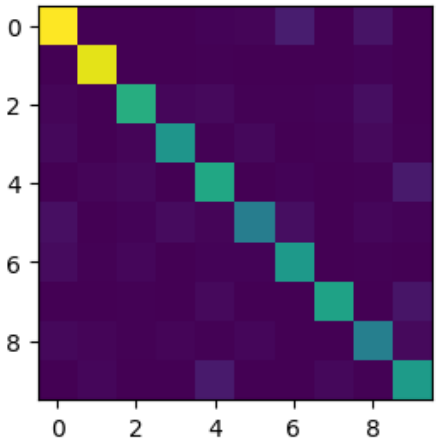
Out[17]:
```
array([  95.28940214, -913.86894309, -131.15364866, -104.77977757,
       -209.07303017,  -85.14159392, -122.04368898, -384.11935833,
        -71.06243118, -252.53913188])
```

Ce résultat vous parait-il normal? Qu'est ce qui peut expliquer cette valeur étonnante? (mettre votre réponse en commenatire à la suite de logpobsBernoulli dans `tme3.py` .

B-3: Évaluer ensuite vos performances avec les mêmes méthodes que précédemment

In [18]:
```
tme3.classifBernoulliTest(Xb_test,Y_test,theta)
```

```
1- Classify all test images ...
2- Classification rate :  0.8533724340175953
3- Matrice de confusion :
```



C. Modélisation des profils de chiffre

Comme expliquer dans le TD 2, il est possible de jouer avec les profils des images: chaque image est alors séparée en 16 lignes et pour chaque ligne, nous modélisons l'apparition du premier pixel allumé avec une loi géométrique. Pour plus de simplicité, nous vous donnons ci-dessous la fonction de transformation de la base d'image et son application.

In [19]:
```
################################################################################
# modelisation geometrique
def transfoProfil(X):
    x2 = []
    for x in X:
        ind = np.where(np.hstack((x.reshape(16, 16), np.ones((16,1))))>0.3)
        x2.append( [ind[1][np.where(ind[0] == i)][0] for i in range(16)])
    return np.array(x2)

Xg_train = transfoProfil(Xb_train)
Xg_test  = transfoProfil(Xb_test)
```

In [20]:
```
print(Xg_train[0]) # [3 2 2 2 2 2 2 1 1 1 2 1 2 2 3 4]
# une image est maintenant représentée par 16 entiers
```

```
[3 2 2 2 2 2 2 1 1 1 2 1 2 2 3 4]
```

C-123: Écrire les fonctions d'apprentissage des paramètres et de calcul de la vraisemblance avec cette modélisation

In [21]:
```
theta = tme3.learnGeom(Xg_train, Y_train,seuil=1e-4)

print(tme3.logpobsGeom(Xg_test[1], theta))
```

```
[ -8.59383056 -30.91829135 -21.49092763 -29.19427401 -25.74819885
 -24.9698666  -20.39913243 -32.67620668 -24.64316047 -28.58230496]
```

In [22]:
```
Y_train_hat = [tme3.classifyGeom(Xg_train[i], theta) for i in range (len(Xg_train))]
Y_test_hat  = [tme3.classifyGeom(Xg_test[i], theta) for i in range (len(Xg_test))]

m_train= tme3.matrice_confusion(Y_train, Y_train_hat)

plt.figure(figsize=(3,3))
plt.imshow(m_train)

m_test = tme3.matrice_confusion(Y_test, Y_test_hat)

plt.figure(figsize=(3,3))
plt.imshow(m_test)

print(f"Taux de bonne classification: {tme3.classificationRate(Y_train , Y_train_hat)=}")
print(f"Taux de bonne classification: {tme3.classificationRate(Y_test , Y_test_hat)=}")
```

```
Taux de bonne classification: tme3.classificationRate(Y_train , Y_train_hat)=0.6594959062449831
Taux de bonne classification: tme3.classificationRate(Y_test , Y_test_hat)=0.6448354512870642
```