TP SVM

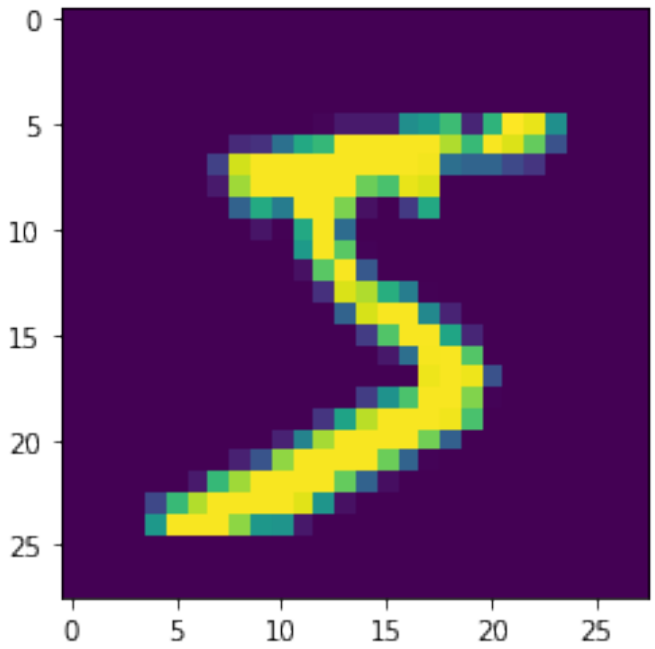
```
import jax
import jax.numpy as jnp
import numpy as np
import matplotlib.pyplot as plt
```

For this lab, we will use the MNIST dataset (~15Mo). It consists of 28x28 images (loaded as a 784 vector) and the associated label for training, validation and test sets.

The following code loads a subset of 1000 training samples and 500 validation samples.

```
# Load the dataset
data = np.load('mnist1k.npz')
X_train = data['X_train']
y_train = data['y_train']
X_val = data['X_val']
y_val = data['y_val']
X_train_bin = data['X_train_bin']
y_train_bin = data['y_train_bin']
X_val_bin = data['X_val_bin']
y_val_bin = data['y_val_bin']
N_train = len(y_train)
N_val = len(y_val)
plt.imshow(X_train[0,:].reshape(28,28))
print(y_train[0])
```

```
5
```



Next, we define the 0-1 loss that measures the error rate of a classifier.

```
def error_rate(y_hat, y):
    return (1.-(y_hat==y)).mean()
```

Implementing a binary Kernel SVM

Q1. Implement the code of the binary kernel SVM classifier in the following class using Stochastic Dual Coordinate Ascent (SDCA). It has to work for any kernel function like the provided linear kernel

```
'''
takes arguements
x1: m x d
x2: n x d
return the Gram matrix m x n
'''
def LinearKernel(x1, x2):
    return jnp.matmul(x1, x2.T)

def SDCAupdate(i,alpha,X,y,K,C):
  y_pred=jnp.dot(K,alpha)
  err=1-y[i]*y_pred[i]
  da=err/K[i,i]
  ai=y[i]*jnp.maximum(0,jnp.minimum(C,da+y[i]*alpha[i]))
  return ai

class KernelSVM():
    def __init__(self, X, y, kernel=LinearKernel, C=1.0):
        self.X = X
        self.y = y
        self.alpha = None
        self.kernel = kernel
        self.C = C

    '''
    x is a matrix nxd of n samples of dimension d
    returns a vector of size n containing the prediction of the class
    '''


    def train(self,key,epoch=10):
      self.alpha= np.zeros((len(self.X)))

      K=self.kernel(self.X,self.X)


      for e in range(epoch):
        key=jax.random.PRNGKey(40)
        key,skey=jax.random.split(key)
        r =jax.random.permutation(key,len(self.X))
```

```
        for i in r:


self.alpha[i]=SDCAupdate(i,self.alpha,self.X,self.y,K,self.C)
```

```
    def __call__(self, x):
        K=self.kernel(x,self.X)
        return jnp.sign(jnp.dot(K,self.alpha))
```

We first try a Linear on the training set reduce to digits 0 and 1 to check that our code works.

```
svm = KernelSVM(X_train_bin, 2*y_train_bin-1, LinearKernel)
key=jax.random.PRNGKey(40)
key,skey=jax.random.split(key)
svm.train(key)
y_hat = svm(X_val_bin)
print(y_hat)
err = error_rate(y_hat, 2*y_val_bin-1)
print(err)
```

```
WARNING:absl:No GPU/TPU found, falling back to CPU. (Set
TF_CPP_MIN_LOG_LEVEL=0 and rerun for more info.)
```

```
[ 1. -1.  1. -1. -1.  1. -1. -1.  1.  1.  1. -1. -1.  1.  1.  1. -1. -
1.
 -1.  1.  1. -1.  1.  1. -1. -1.  1.  1.  1.  1. -1. -1. -1.  1. -1.
1.
 -1.  1. -1. -1. -1.  1. -1.  1.  1.  1.  1. -1.  1.  1. -1.  1. -1. -
1.
  1.  1.  1. -1. -1.  1.  1. -1. -1. -1.  1. -1.  1. -1. -1.  1.  1. -
1.
  1. -1. -1.  1. -1.  1.  1. -1. -1. -1.  1.  1.  1.  1. -1. -1. -1. -
1.
 -1.  1. -1.  1.  1. -1.  1.  1. -1.  1. -1.  1. -1.  1. -1.  1.  1.]
0.009345794
```

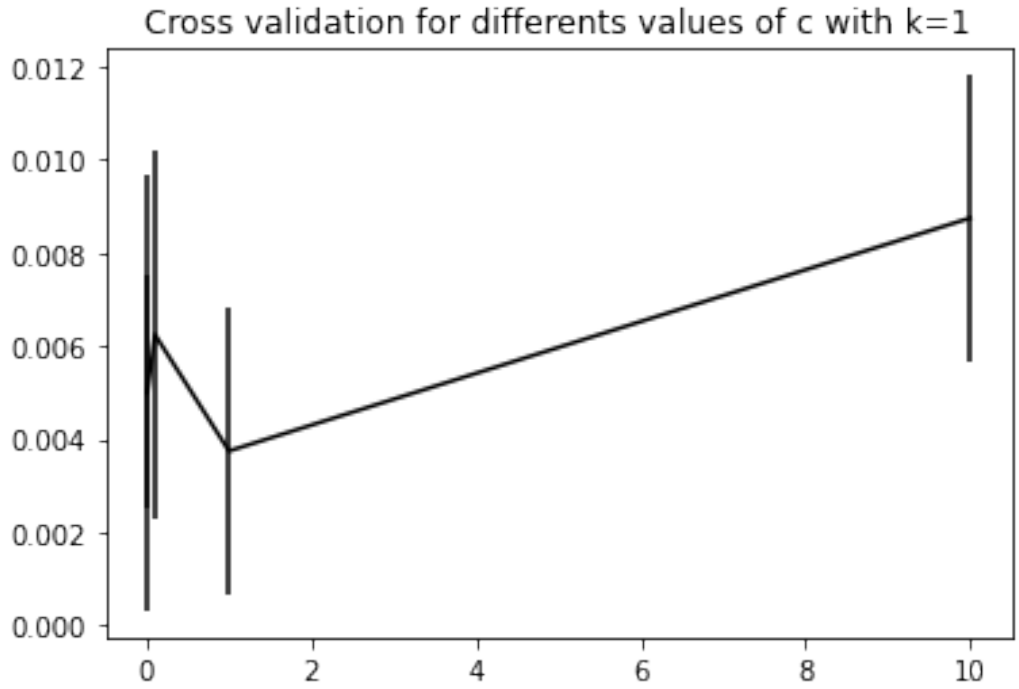
Q2. Use cross-validation to find the optimal number of epochs for

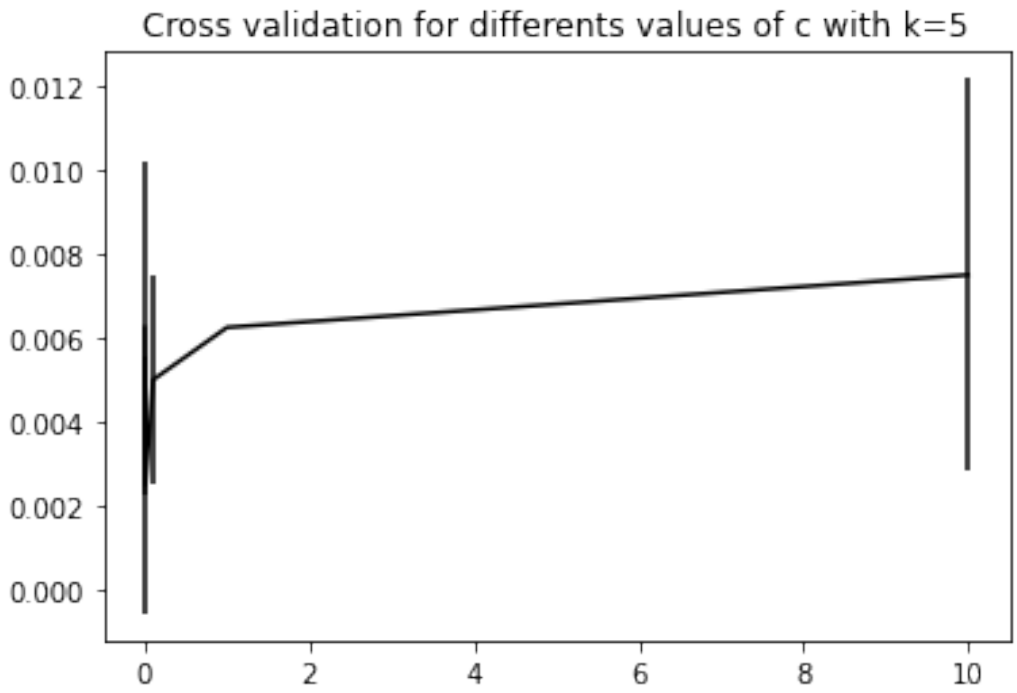
- Élément de liste
- Élément de liste

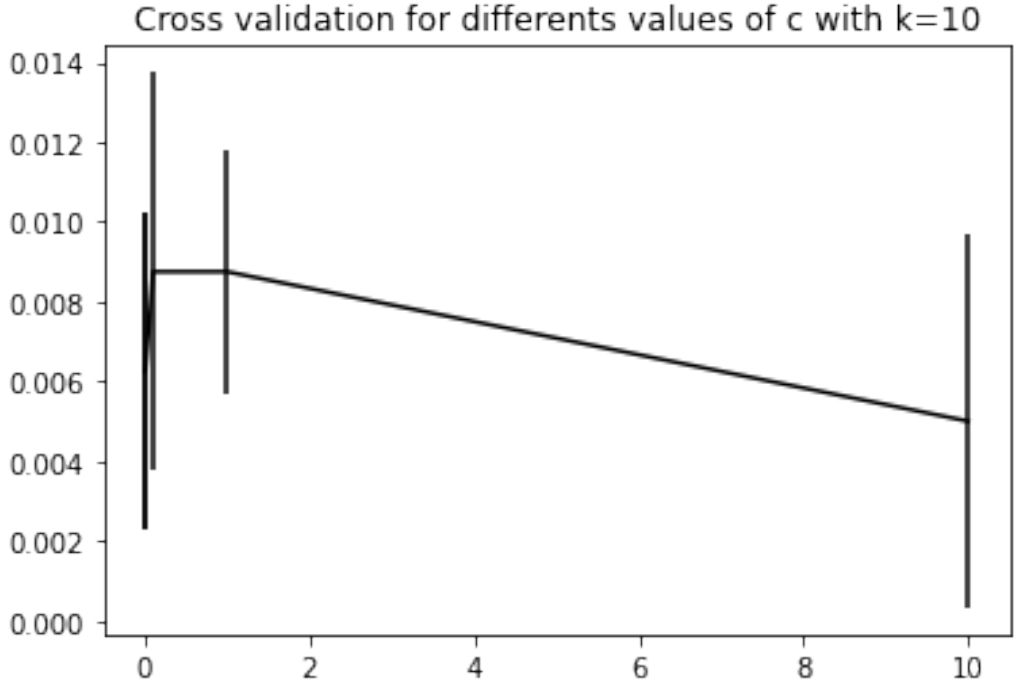
training, up to a maximum of 25, and the optimal value of C

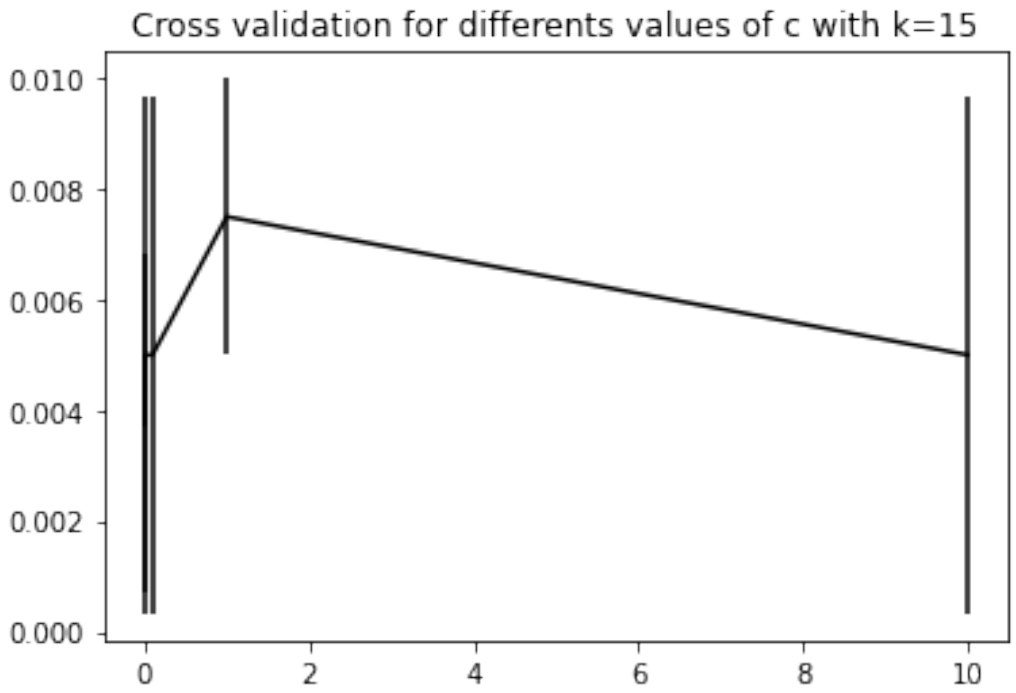
```
def randomSplit(key, X, y, train_part=0.5):
    n=X.shape[0]
    n_train = int(train_part * n); n_test = n - n_train
    p = jax.random.permutation(key,n)
```

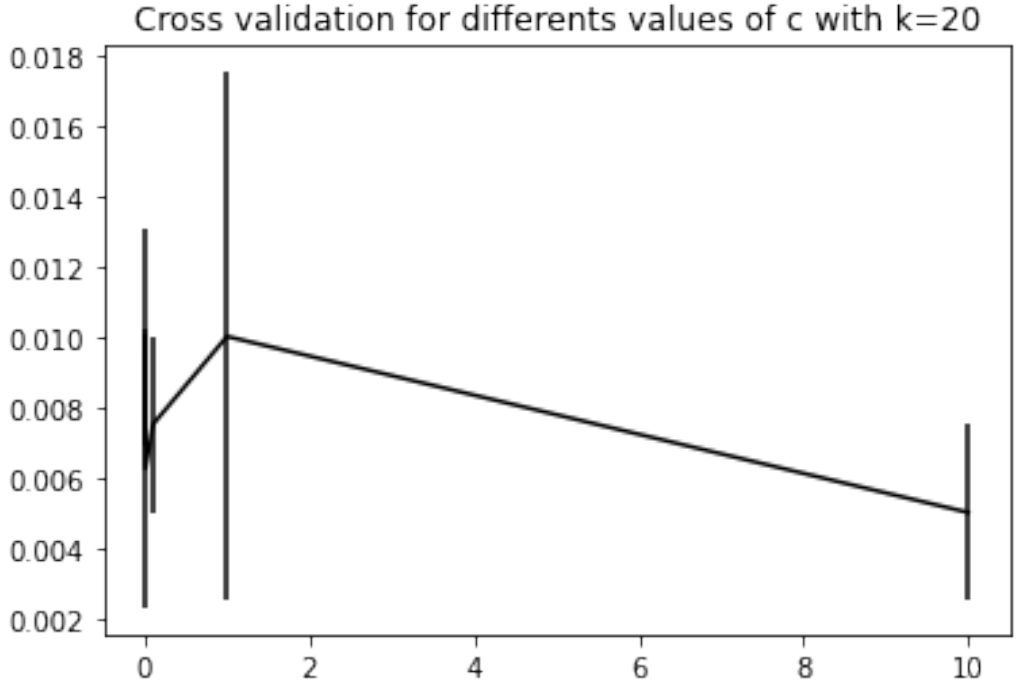
```
    X_train1 = X[p[0:n_train], :]; y_train1 = y [p[0:n_train]]
    X_val1 = X[p[n_train:],:] ; y_val1 = y[p[n_train:]]
    return X_train1, y_train1, X_val1, y_val1

key = jax.random.PRNGKey(42)
list_k = [1,5,10,15,20]
list_C = [0.001,0.01,0.1,1,10]
for k in list_k:
    l = []
    for c in list_C:
      lkc = []
      for s in range(5):
          key, skey = jax.random.split(key)
          X_train1, y_train1, X_val1, y_val1 = randomSplit(skey,
np.concatenate((X_train_bin,X_val_bin)),
np.concatenate((y_train_bin,y_val_bin)))
          svm = KernelSVM(X_train1, 2*y_train1-1, LinearKernel,C=c)
          svm.train(key,k)
          y_hat1 = svm(X_val1)
          lkc.append(error_rate(y_hat1, 2*y_val1-1))
      l.append(lkc)
    l = jnp.asarray(l)
    plt.errorbar(list_C, l.mean(axis=1), l.std(axis=1), fmt='-k')
    plt.title("Cross validation for differents values of c with k=" +
str(k))
    plt.figure()
plt.show()
```




Cross validation for differents values of c with k=5
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
2
4
6
8
10


Cross validation for differents values of c with k=10
0.014
0.012
0.010
0.008
0.006
0.004
0.002
0.000
0
2
4
6
8
10






```
<Figure size 432x288 with 0 Axes>
```

Les erreurs restent plutôt faibles, dans l'ensemble. Une augmentation de la valeur de C semble diminuer l'erreur, cependant on peut penser à un overfitting car les poids des alphas peuvent ainsi se mettre à un seulement sur ceux présent sur la marge. On peut donc prévoir une moins bonne généralisation pour ces valeurs. On remarque que le nombre

d'épochs n'importe que peu sur le résultat. Une valeur de k=15 et C=0.1 semble être pertinent.

Multi-class classification

Next, we want to perform a multiclass classification using our SVM. We will use the One-versus-All approach where we train a classifier for each class against all others. A test time, we select the class corresponding to the classifier that output the maximum score.

Q3. Code a Multi-class SVM using a One-versus-All approach and validate it on the validation set

```
class OnevsAllKSVM():
    def __init__(self, X, y,kernel=LinearKernel,C=1):
        self.X = X
        self.y = np.array([2*(y==i)-1 for i in range(10)])
        self.alpha=np.zeros((10,X.shape[0]))
        self.C=C
        self.kernel = kernel
        self.train(key)

    def train (self,key,epoch=10):
      key=jax.random.PRNGKey(40)
      K=self.kernel(self.X,self.X)
      self.alpha= np.zeros((10,len(self.X)))
      for e in range(epoch):

        key,skey=jax.random.split(key)
        r =jax.random.permutation(key,len(self.X))
        for i in r:
          xi=self.X[i]

          zi = self.y[:,i]*(jnp.dot((self.alpha)*(np.asarray(self.y)),
(self.kernel(xi,self.X)).T)).squeeze()

          self.alpha[:,i]+=(1-zi)/self.kernel(xi,xi)

self.alpha[:,i]=jnp.maximum(0,jnp.minimum(self.C,self.alpha[:,i]))




    def __call__(self, x):
         K=self.kernel(self.X,x)
         return jnp.argmax(jnp.dot(self.alpha*self.y,K),axis=0)

svm = OnevsAllKSVM(X_train, y_train, LinearKernel)
svm.train()

y_hat = svm(X_val,)
err = error_rate(y_hat, y_val)
print(err)
```

```
0.19000001
```

Le taux d'erreur pour le oneversusall est plus elevé comme attendu.L'entrainement est beaucoup plus long comparé à la classifaction binaire. Ce qui est cohérent car le nombre de frontières à déterminer est lui aussi plus grand.

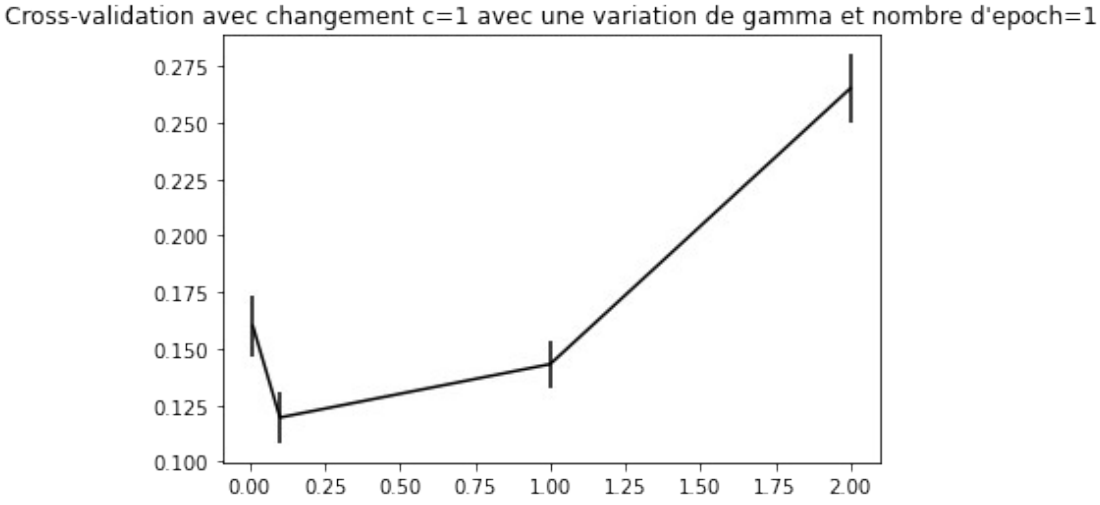
Kernels

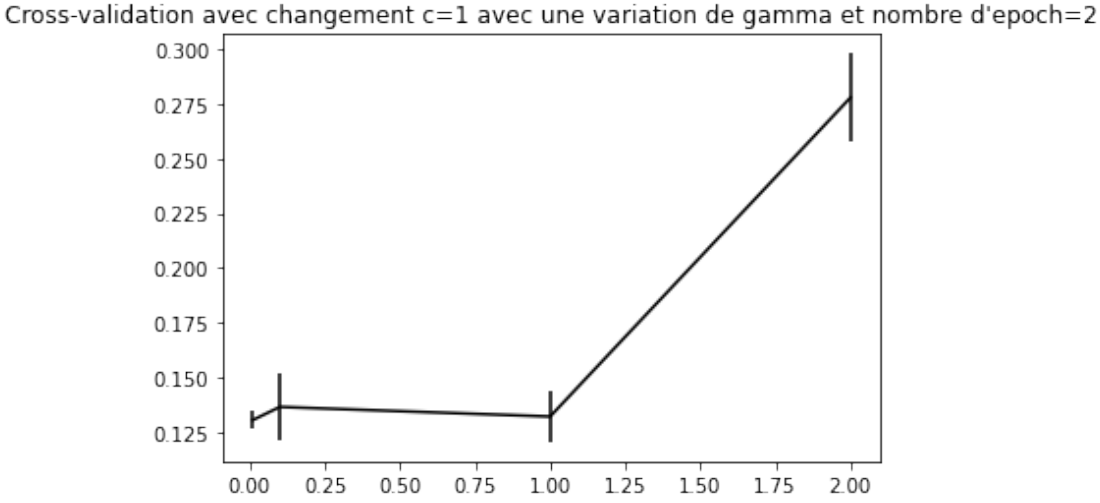
A linear classifier is unlikely to be able to classify correctly all classes, we will thus try several different kernels.

Q4. Code a class for the Gaussian kernel, the polynomial kernel and the inhomogeneous polynomial kernel, and perform cross-validation to select a kernel and its hyperparameters

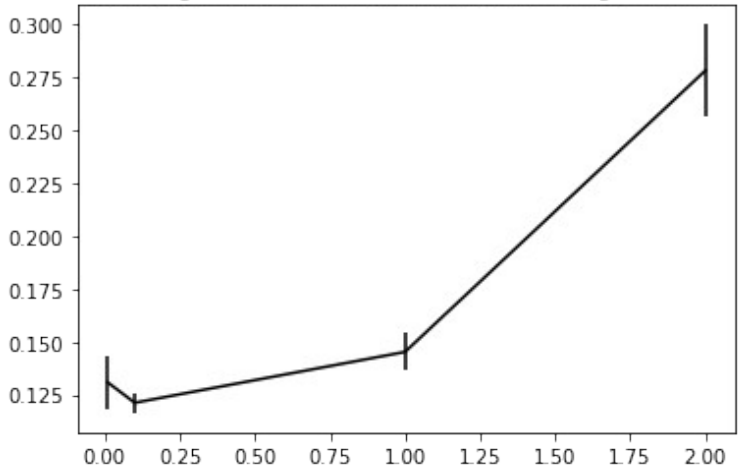
```
class GaussKernel():
    def __init__(self, gamma=1.0):
        self.gamma = gamma
    '''
    compute the Gram Matrix
    '''
    def __call__(self, x1, x2):
        return jnp.exp(-self.gamma * ( jnp.linalg.norm(x1, axis=-1,
keepdims=True)**2 + jnp.linalg.norm(x2, axis=-1, keepdims=False).T**2
- 2*jnp.dot(x1,x2.T) ))
class PolyKernel():
    def __init__(self, d=1.0, c=0.):
        self.d = d
        self.c = c
    '''
    compute the Gram Matrix
    '''
    def __call__(self, x1, x2):
        return jnp.power(jnp.matmul(x1, x2.T) + self.c, self.d)

gamma=[0.01,0.1,1,2]
K=[1,2,5,10,20]

for k in K:
  lp=[]
  for gam in gamma:
      lk=[]
      for s in range(5):

          key, skey = jax.random.split(key)

          X_train1, y_train1, X_val1, y_val1 = randomSplit(skey,
X_train, y_train)

          svm = OnevsAllKSVM(X_train1, y_train1,
GaussKernel(gamma=gam),C=0.1)
```

```
            svm.train(key,k)
            y_hat = svm(X_val1)
            lk.append(error_rate(y_hat, y_val1))
        lp.append(lk)
    lp = jnp.asarray(lp)
    plt.errorbar(gamma, lp.mean(axis=1), lp.std(axis=1), fmt='-k')
    plt.title("Cross-validation avec changement c=1 avec une variation
de gamma et nombre d'epoch="+str(k))
    plt.show()
```

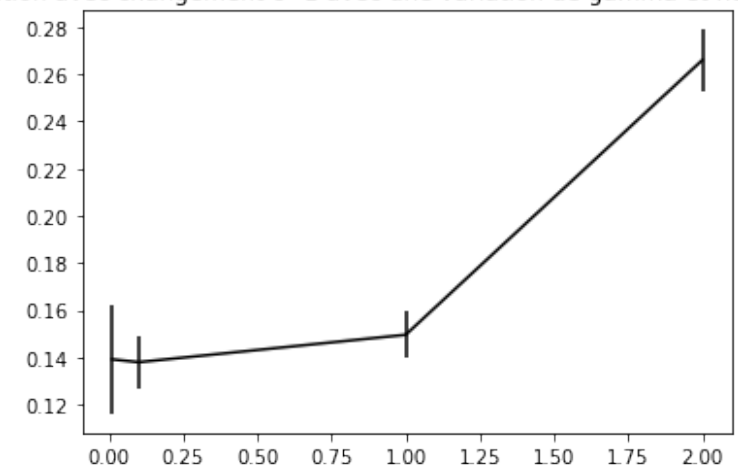




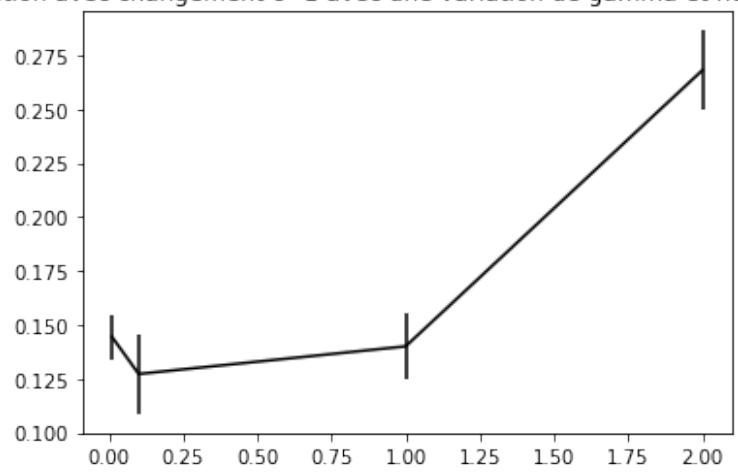
Cross-validation avec changement c=1 avec une variation de gamma et nombre d'epoch=5



Cross-validation avec changement c=1 avec une variation de gamma et nombre d'epoch=10

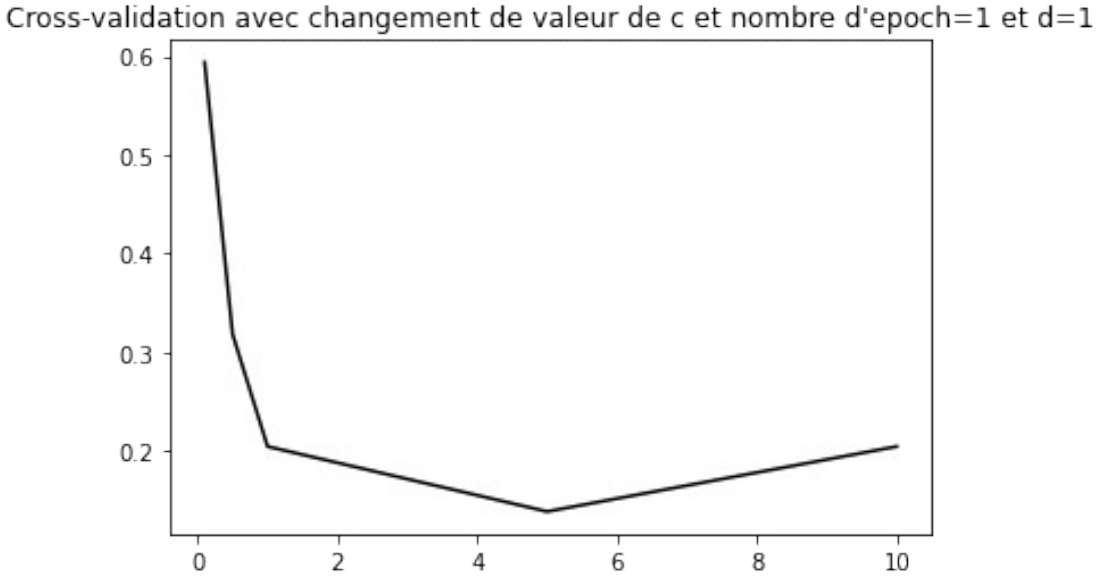


Cross-validation avec changement c=1 avec une variation de gamma et nombre d'epoch=20

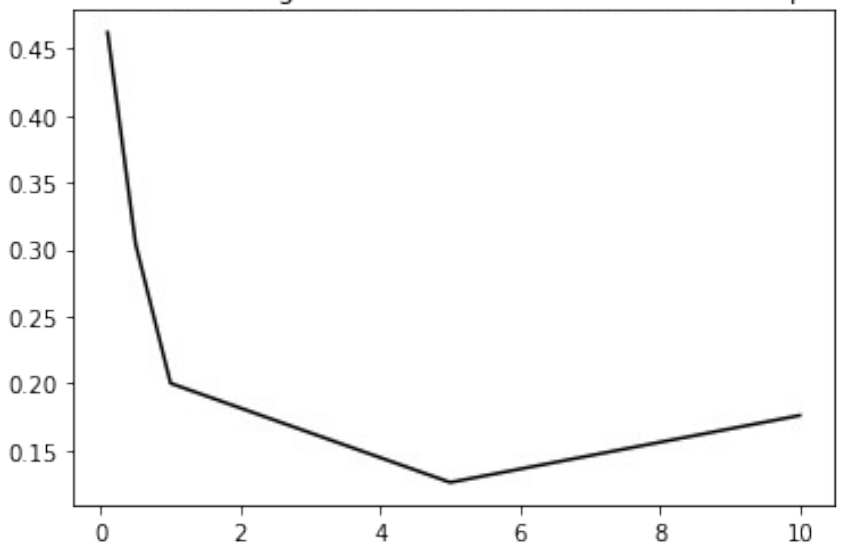


Un grand nombre d'époque semble être important pour obtenir une erreur minimale. Il s'agit donc de mettre en place une recherche de l'argument minimal. Il faut prendre une valeur de gamma petite pour avoir de bons résultats.

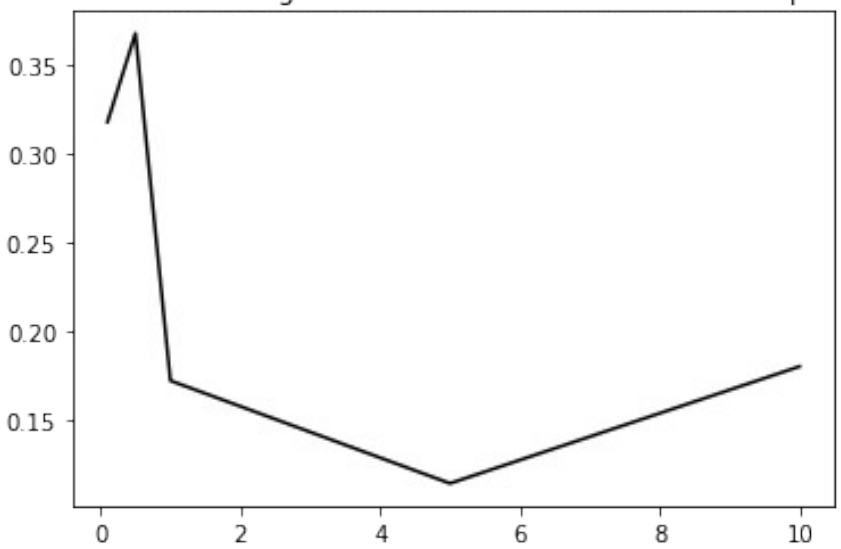
```
D=[d for d in range(1,3)]
K=[1,2,5,10,20]
C=[0.1,0.5,1,5,10]
for d in D:
  lp=[]
  for k in K:
      lk=[]
      for c in C:
          lc=[]
          for s in range(5):

              key, skey = jax.random.split(key)

              X_train1, y_train1, X_val1, y_val1 = randomSplit(skey,
X_train, y_train)
              svm = OnevsAllKSVM(X_train1, y_train1, PolyKernel(c,d))
              key=jax.random.PRNGKey(40)
              key,skey=jax.random.split(key)
              svm.train(key,k)
              y_hat = svm(X_val1)
              lc.append(error_rate(y_hat, y_val1))
          lk.append(lc)
      lk = jnp.asarray(lk)
      plt.errorbar(C, lk.mean(axis=1), lk.std(axis=1), fmt='-k')
      plt.title("Cross-validation avec changement de valeur de c et
nombre d'epoch="+str(k)+" et d="+str(d))
      plt.show()
```



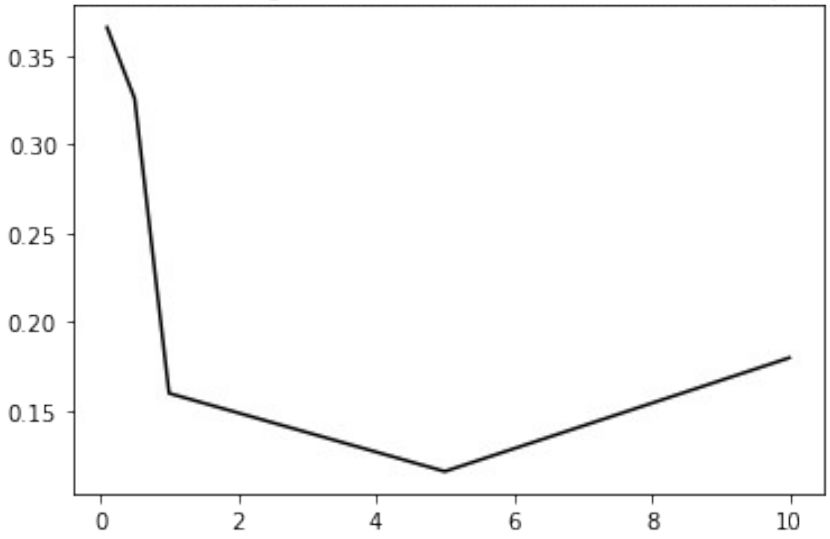
Cross-validation avec changement de valeur de c et nombre d'epoch=2 et d=1



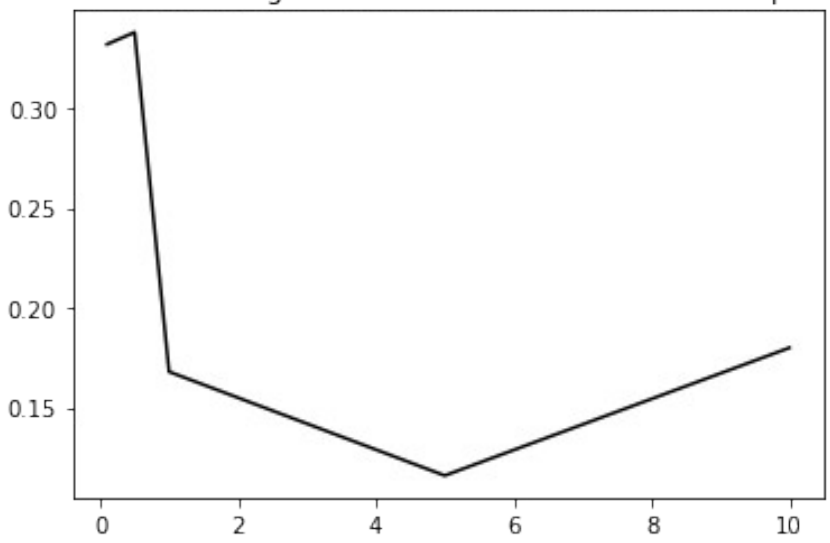
Cross-validation avec changement de valeur de c et nombre d'epoch=5 et d=1



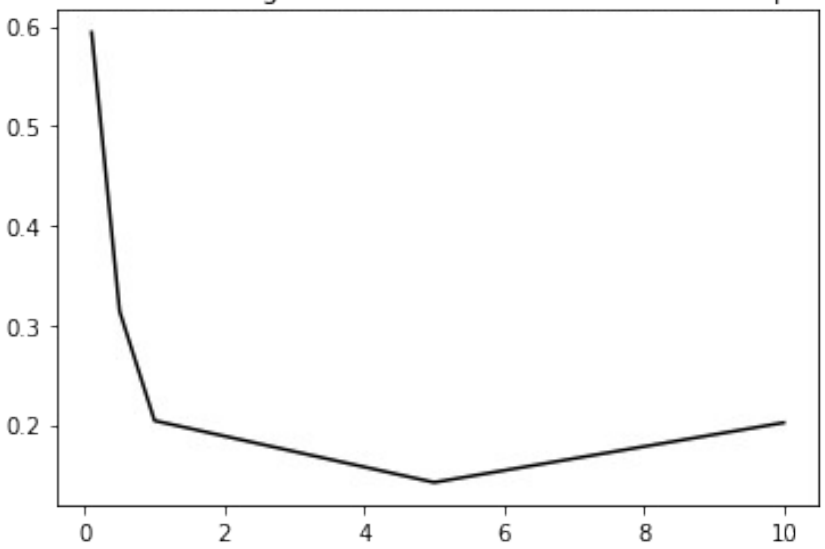
Cross-validation avec changement de valeur de c et nombre d'epoch=10 et d=1



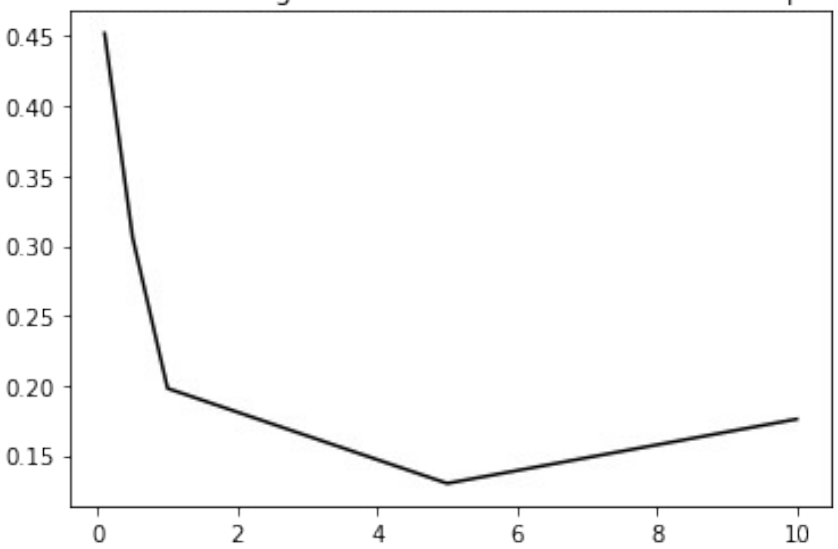
Cross-validation avec changement de valeur de c et nombre d'epoch=20 et d=1



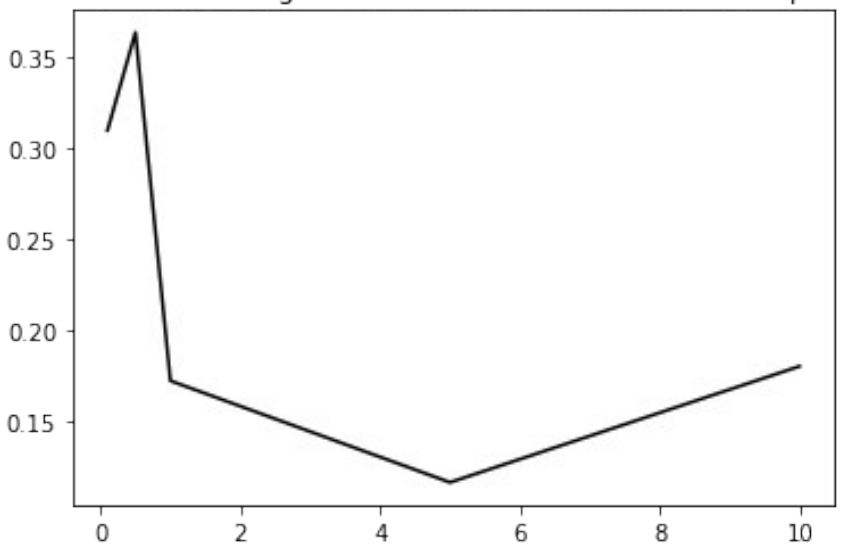
Cross-validation avec changement de valeur de c et nombre d'epoch=1 et d=2



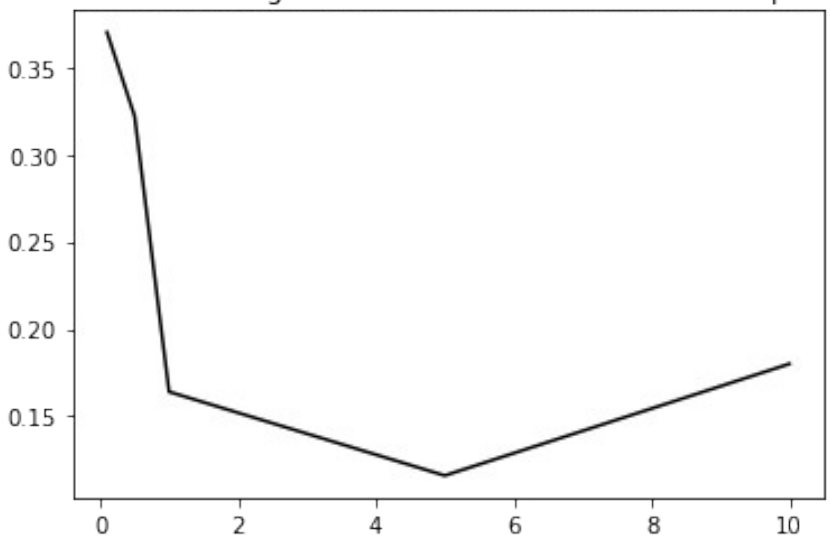
Cross-validation avec changement de valeur de c et nombre d'epoch=2 et d=2

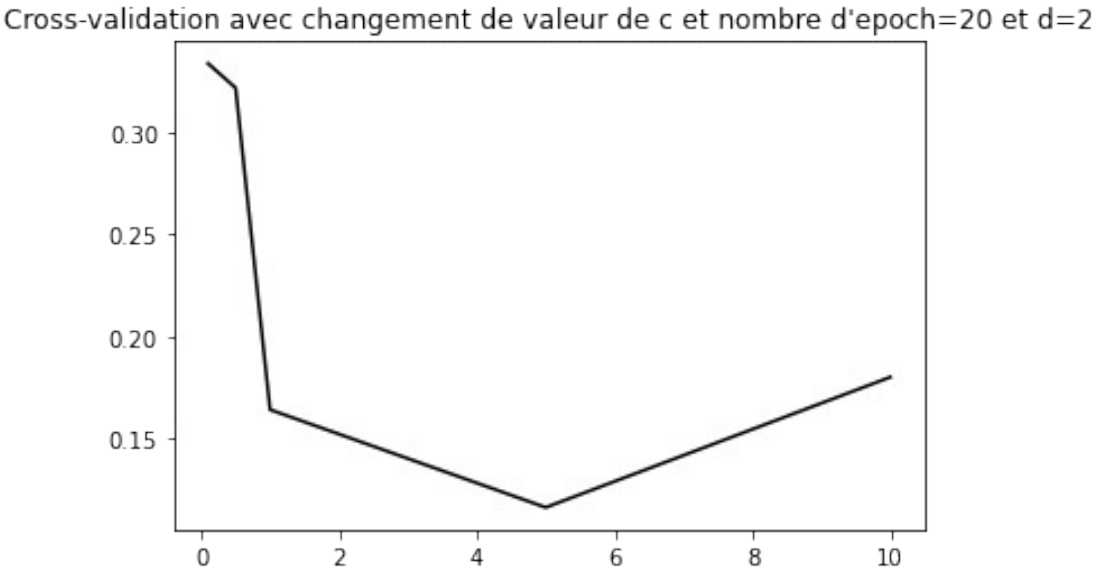


Cross-validation avec changement de valeur de c et nombre d'epoch=5 et d=2



Cross-validation avec changement de valeur de c et nombre d'epoch=10 et d=2

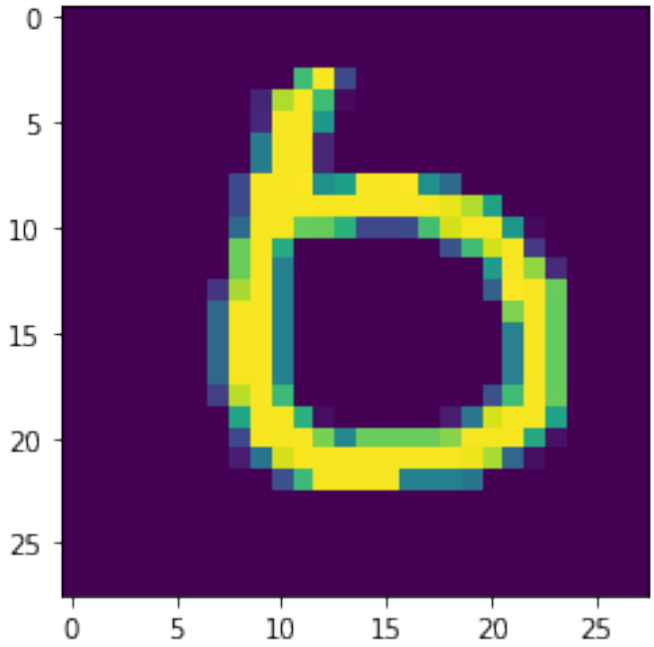


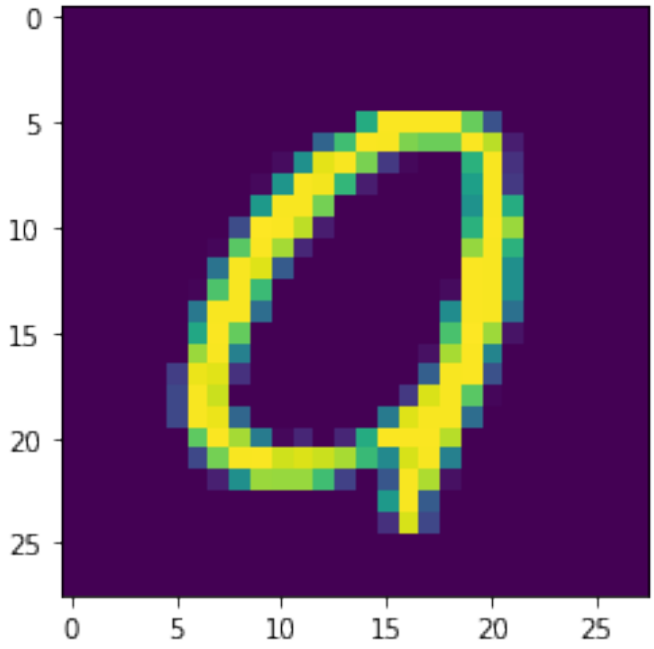


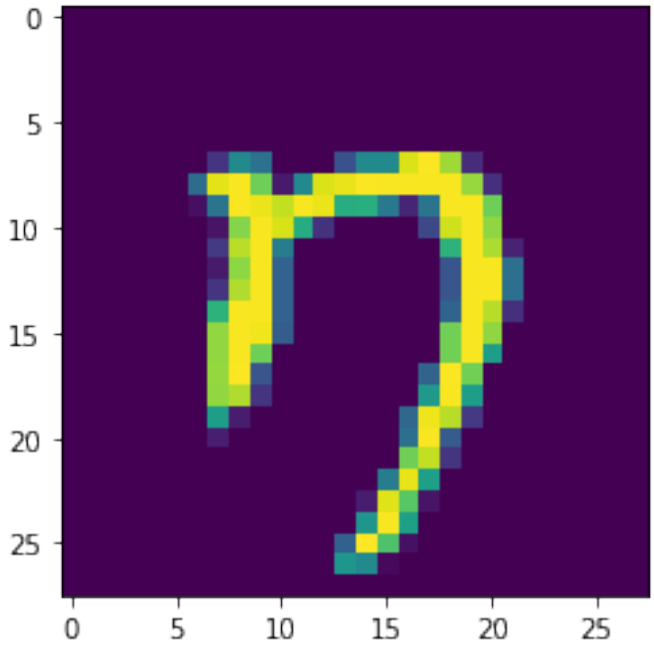
On voit donc qu'il ne faut pas prendre une valeur de c trop importante sous peine de voir l'erreur augmentée très rapidement. Le nombre d'epochs doit lui être au moins de 10 pour assurer une bonne valeur pour l'erreur.

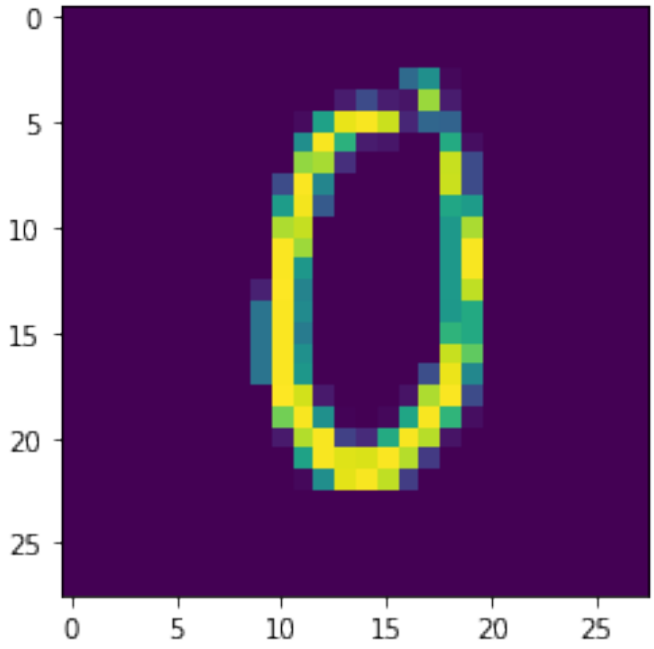
Q5. For the best performing kernel, visualize the support vectors (limit to the ones with largest weights)

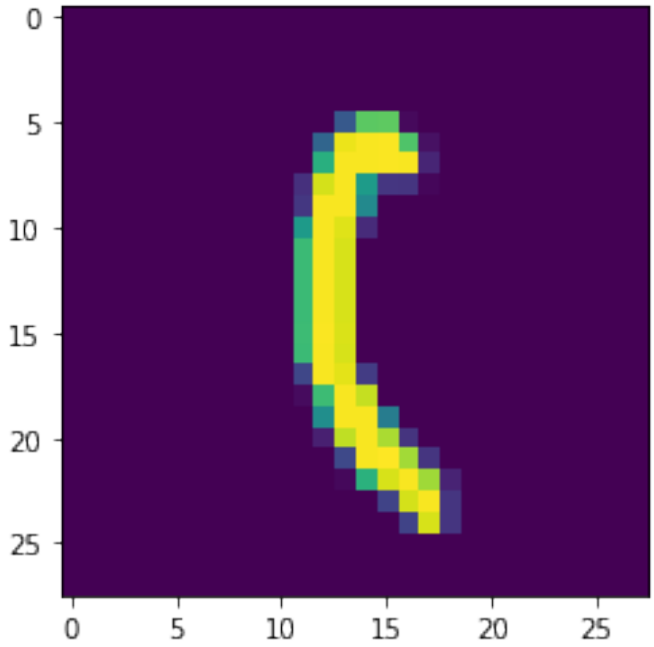
```
max_alpha=jnp.argsort(np.abs(svm.alpha),axis=1)
for k in range(10):
  for i in range(1,5):
    plt.imshow(X_train[max_alpha[k,len(max_alpha[0])-
i],:].reshape(28,28))
    plt.show()
```

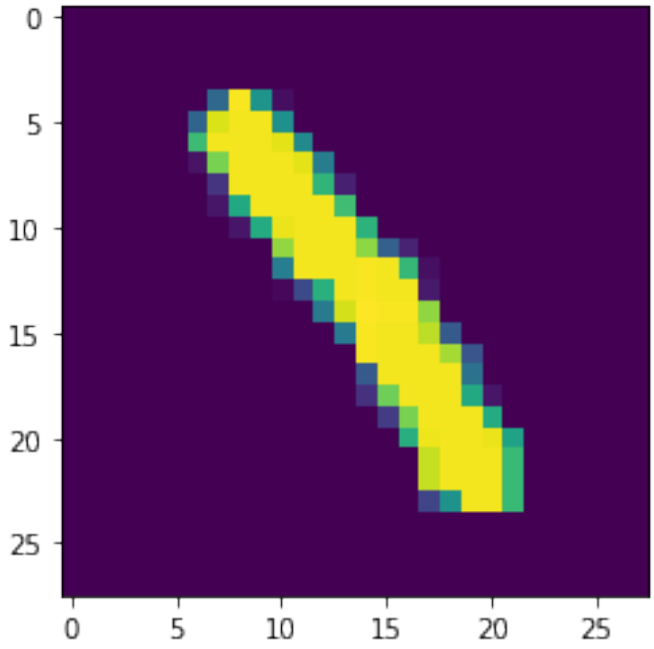

0
5
10
15
20
25
0
5
10
15
20
25

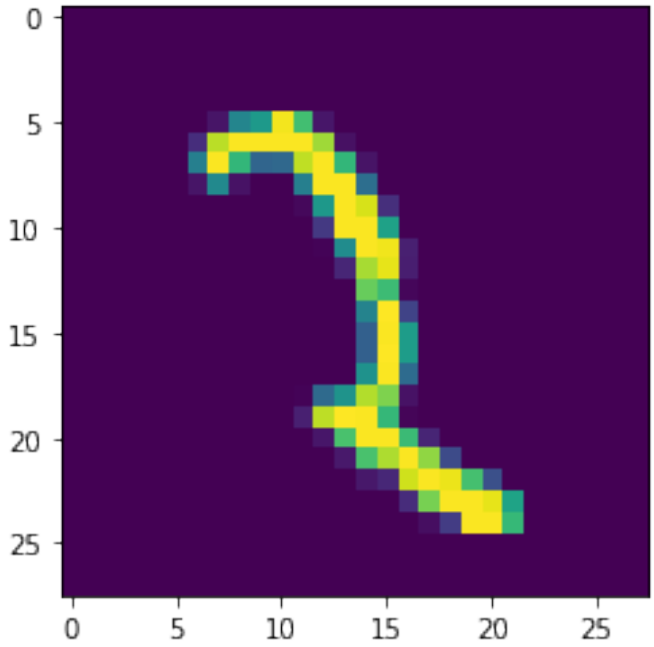

0
5
10
15
20
25
0
5
10
15
20
25

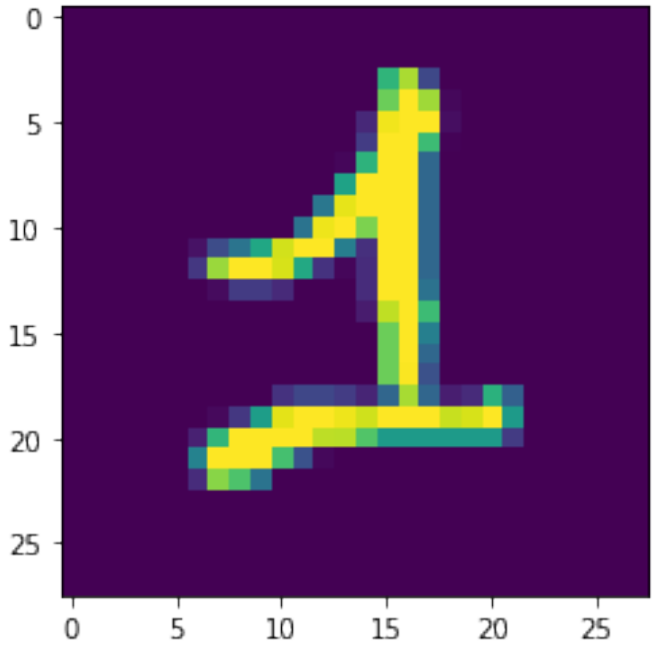

0
5
10
15
20
25
0
5
10
15
20
25

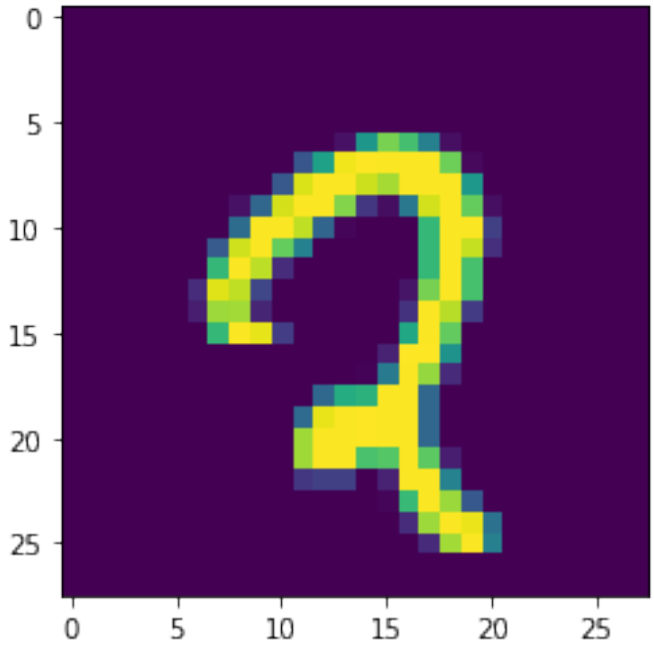

0
5
10
15
20
25
0
5
10
15
20
25

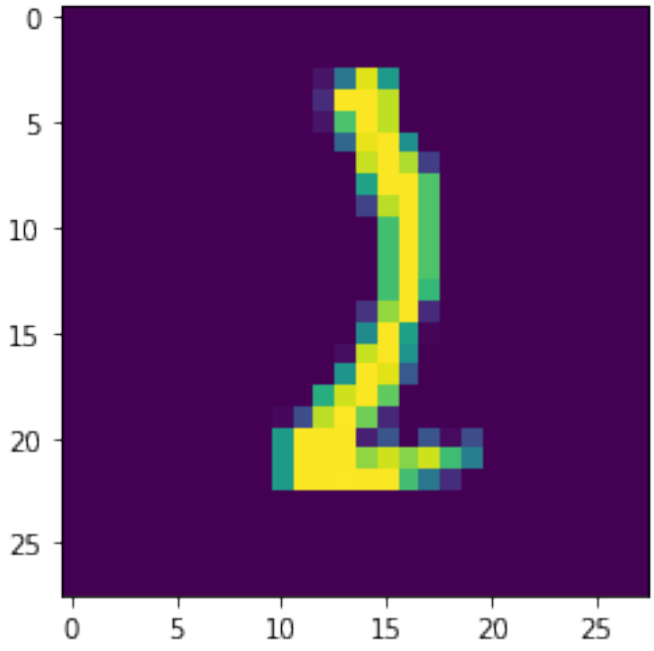

0
5
10
15
20
25
0
5
10
15
20
25

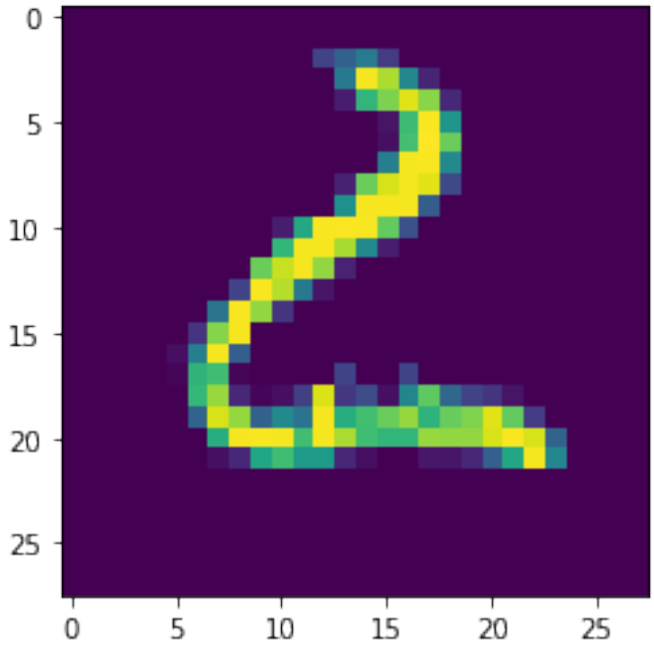

0
5
10
15
20
25
0
5
10
15
20
25

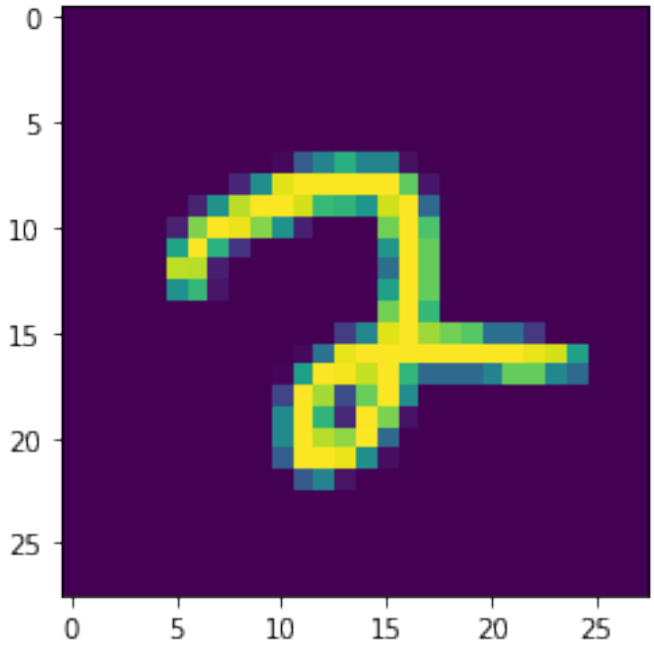

0
5
10
15
20
25
0
5
10
15
20
25

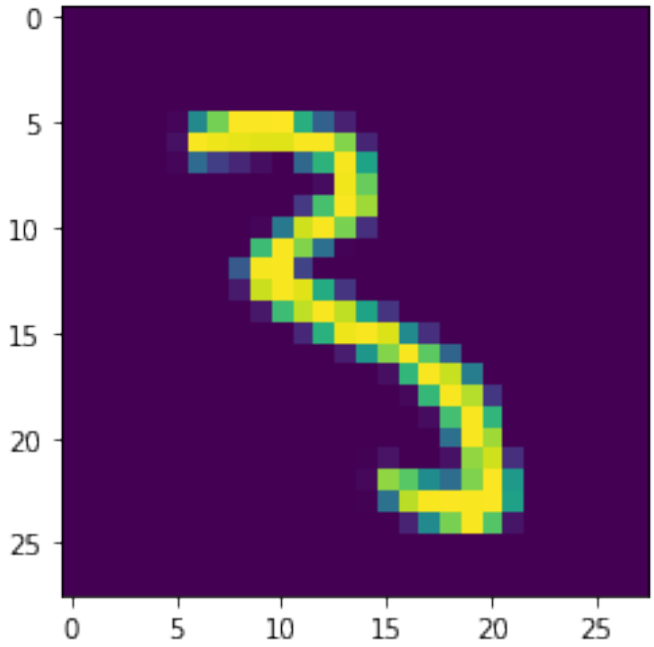

0
5
10
15
20
25
0
5
10
15
20
25

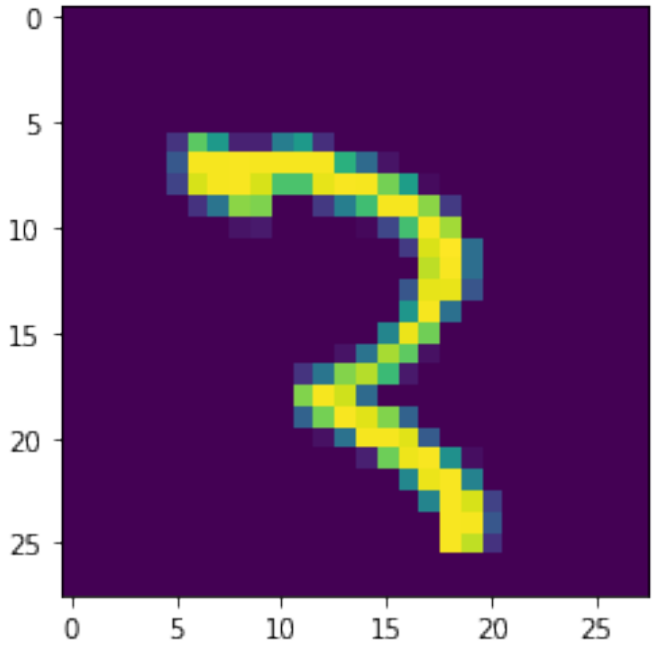

0
5
10
15
20
25
0
5
10
15
20
25

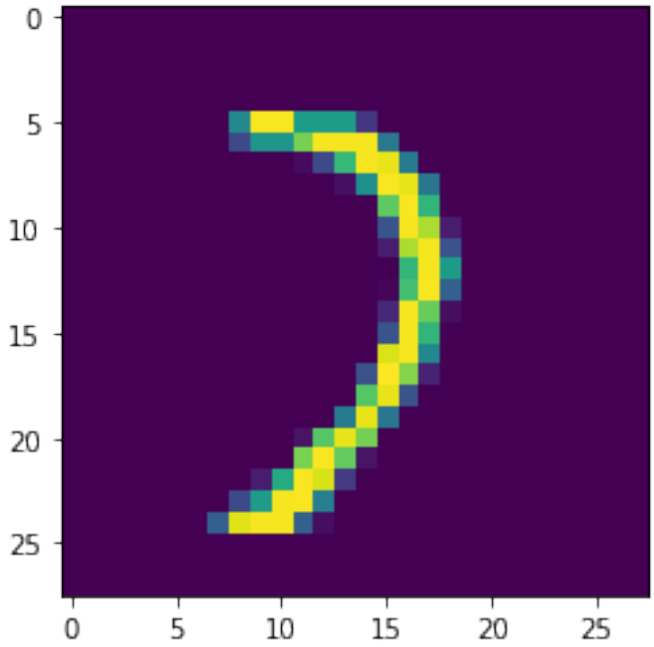

0
5
10
15
20
25
0
5
10
15
20
25

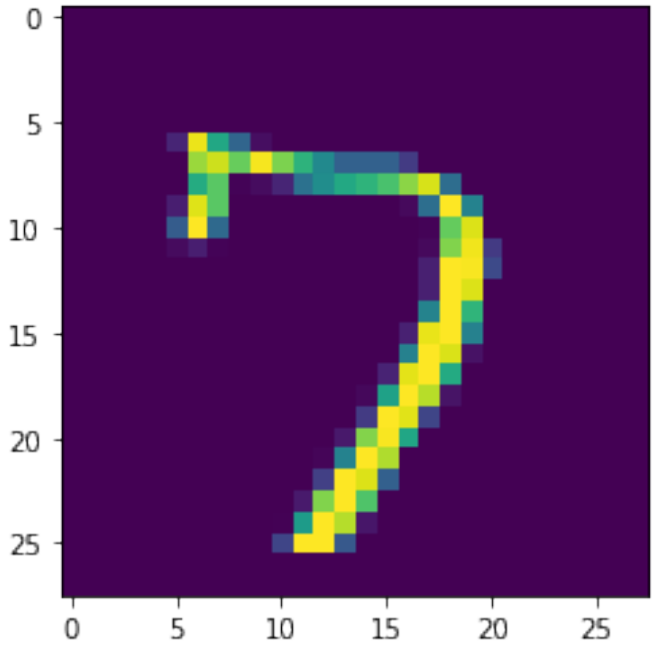

0
5
10
15
20
25
0
5
10
15
20
25

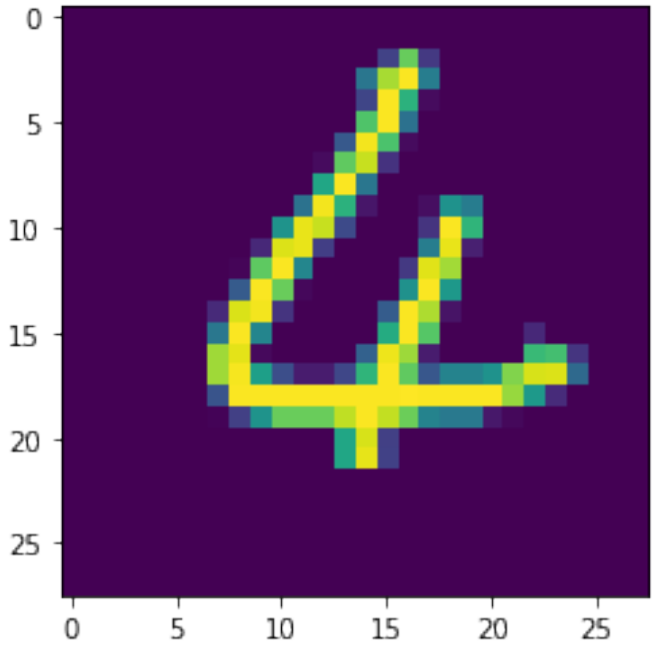

0
5
10
15
20
25
0
5
10
15
20
25

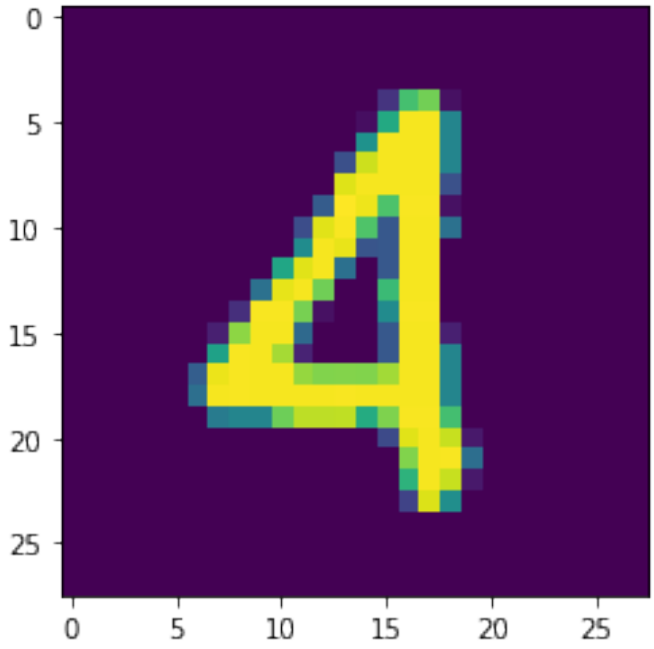

0
5
10
15
20
25
0
5
10
15
20
25

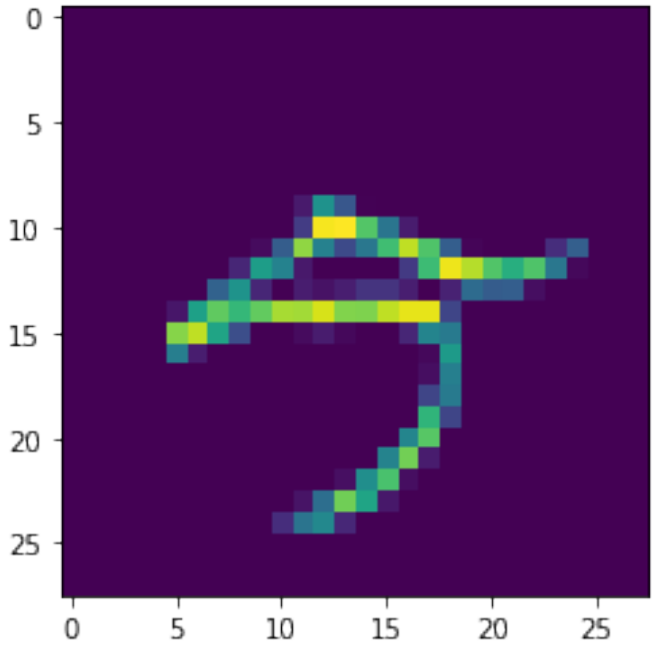

0
5
10
15
20
25
0
5
10
15
20
25

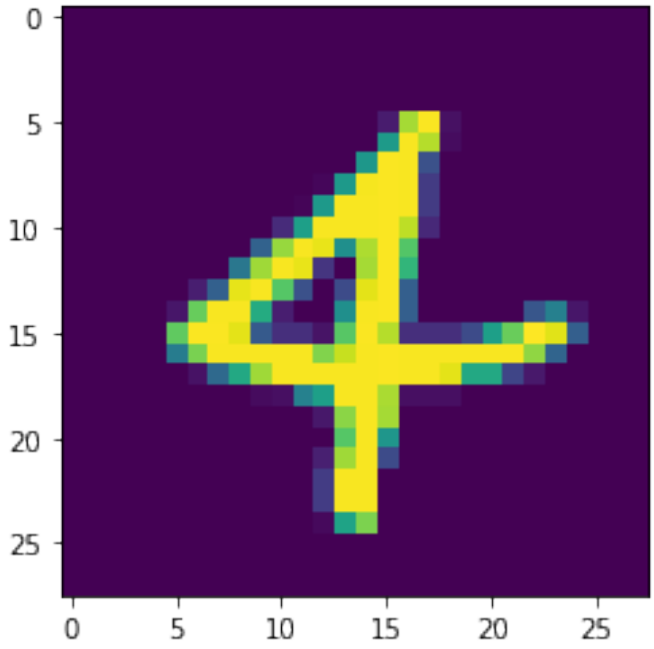

0
5
10
15
20
25
0
5
10
15
20
25

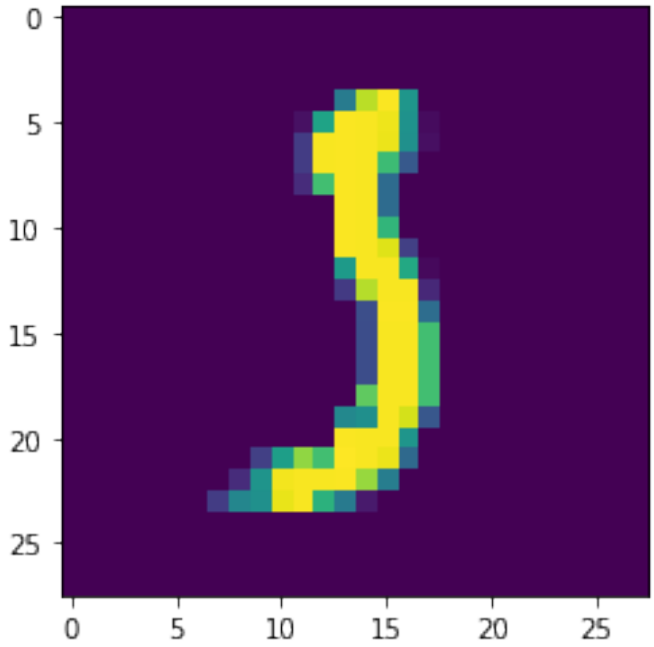

0
5
10
15
20
25
0
5
10
15
20
25

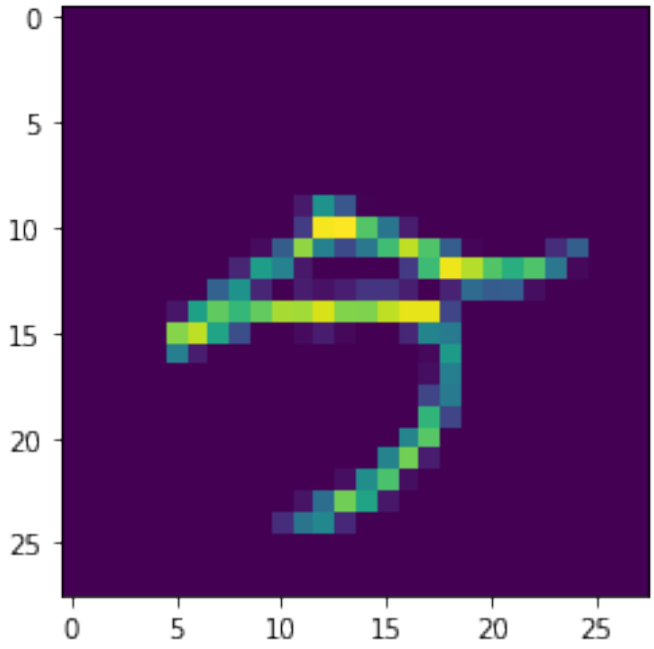

0
5
10
15
20
25
0
5
10
15
20
25

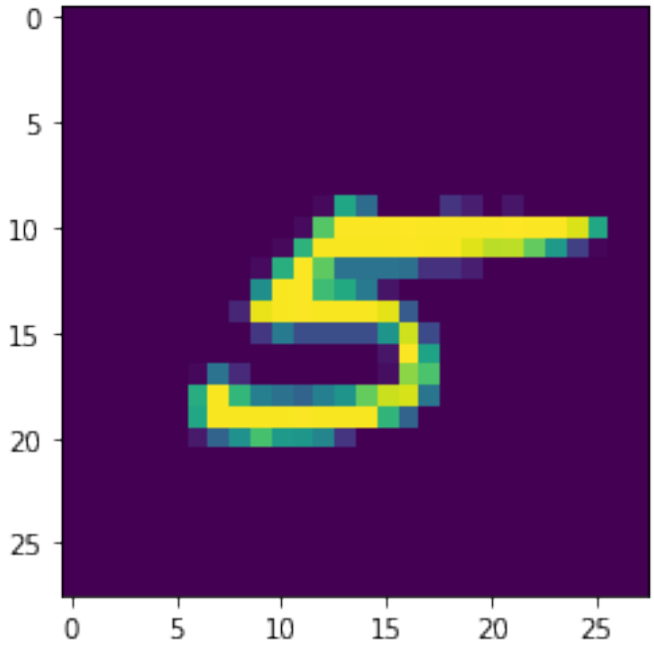

0
5
10
15
20
25
0
5
10
15
20
25

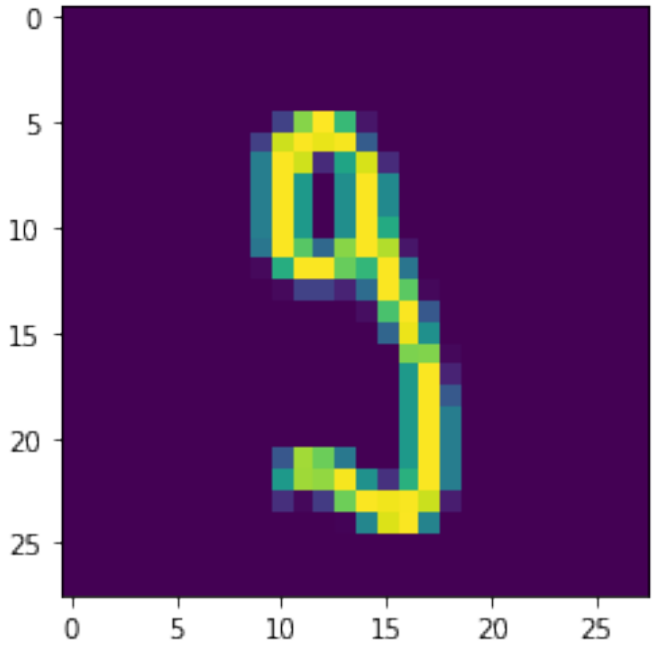

0
5
10
15
20
25
0
5
10
15
20
25

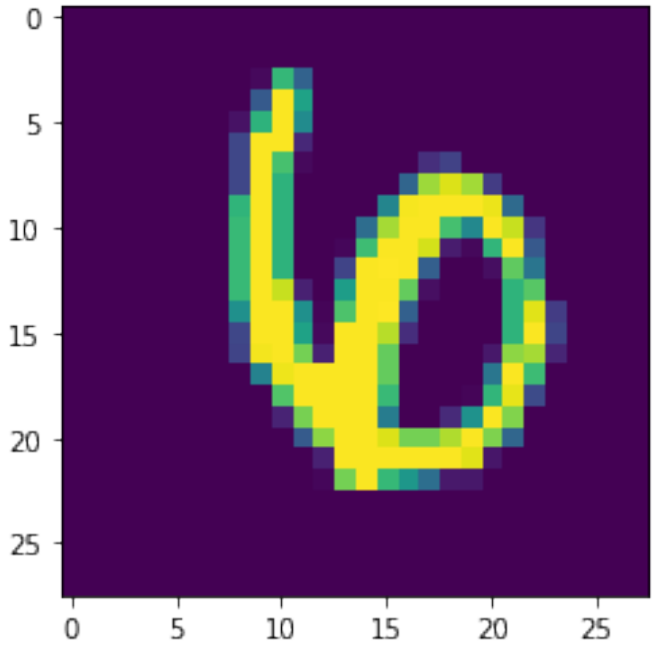

0
5
10
15
20
25
0
5
10
15
20
25

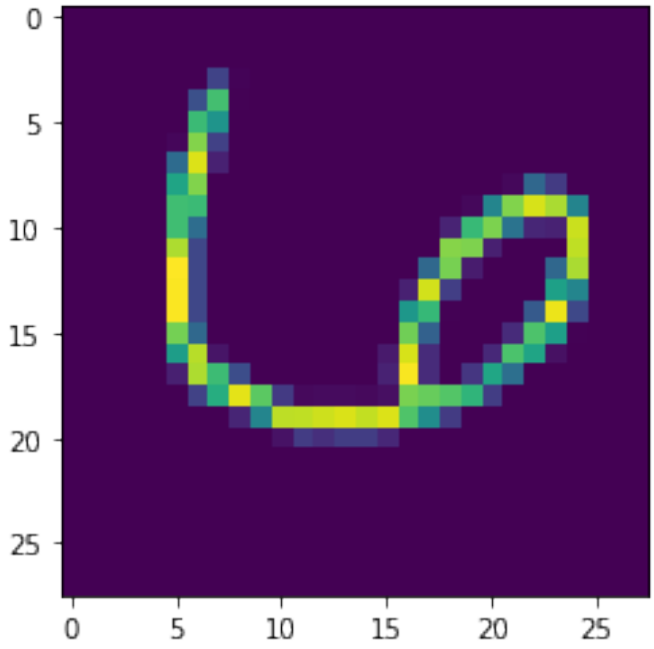

0
5
10
15
20
25
0
5
10
15
20
25

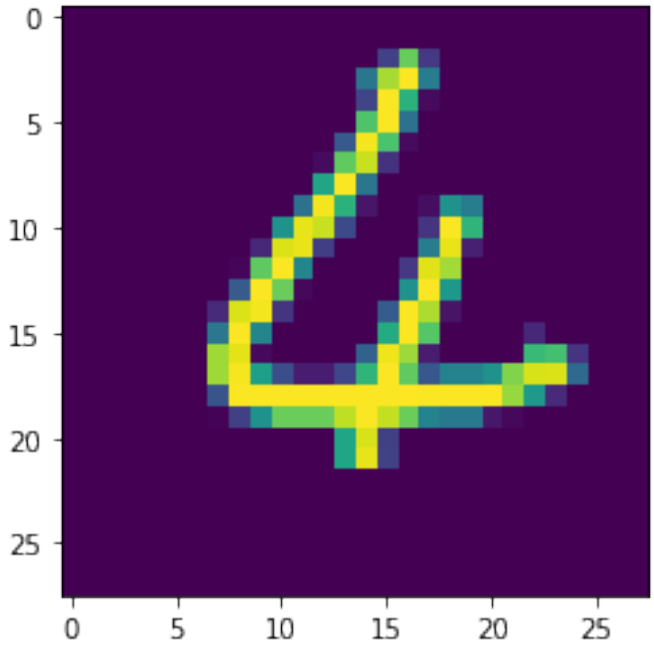

0
5
10
15
20
25
0
5
10
15
20
25

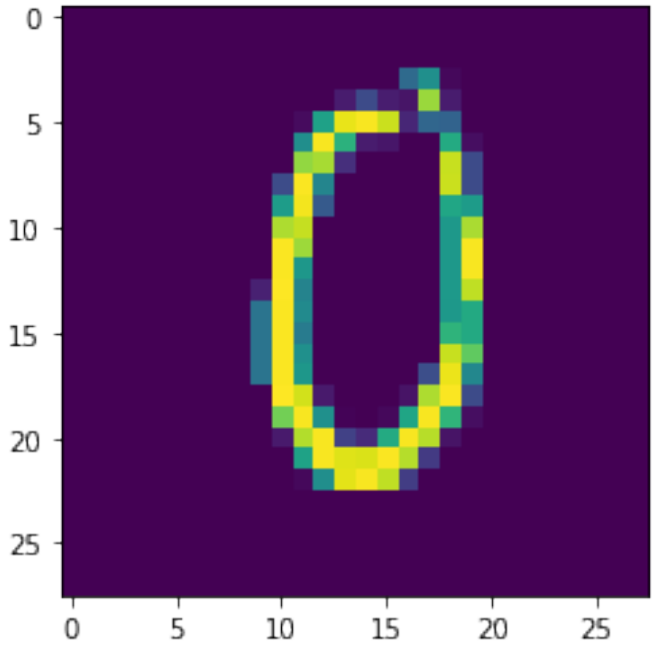

0
5
10
15
20
25
0
5
10
15
20
25

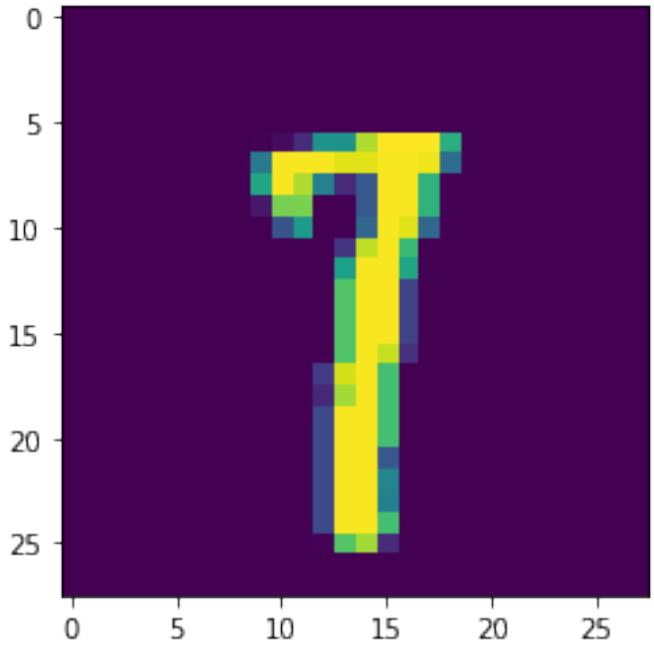

0
5
10
15
20
25
0
5
10
15
20
25

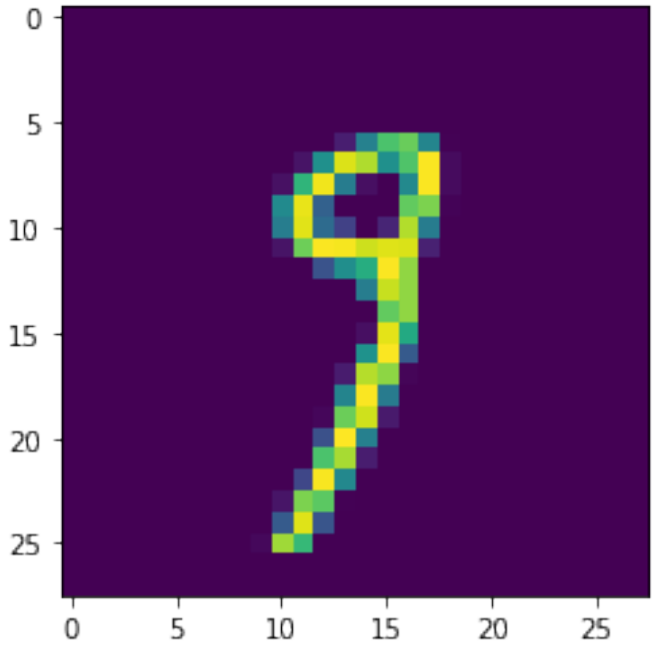

0
5
10
15
20
25
0
5
10
15
20
25

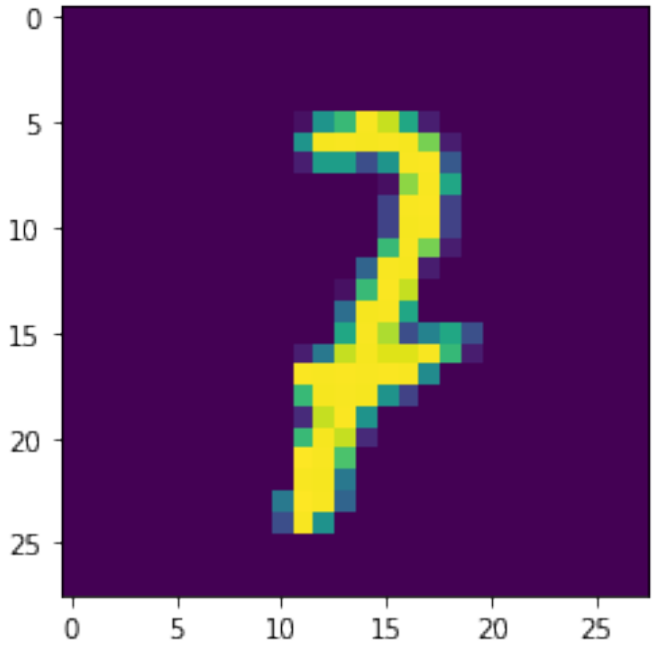

0
5
10
15
20
25
0
5
10
15
20
25

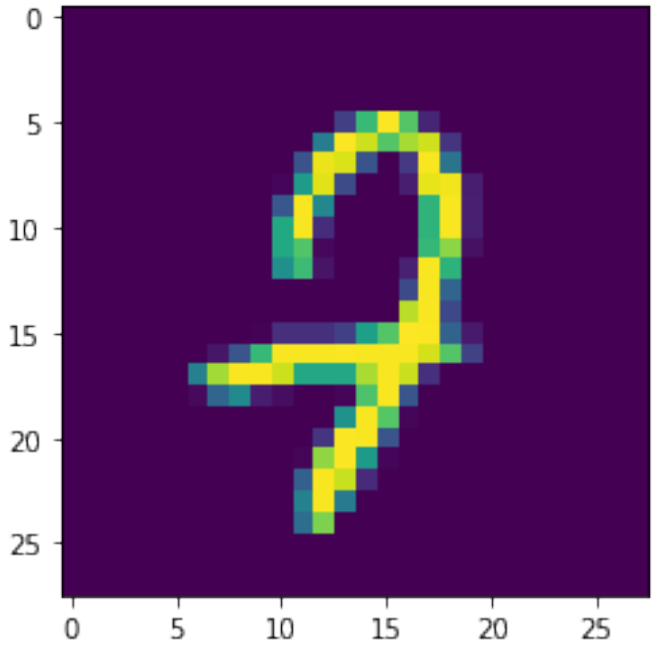

0
5
10
15
20
25
0
5
10
15
20
25

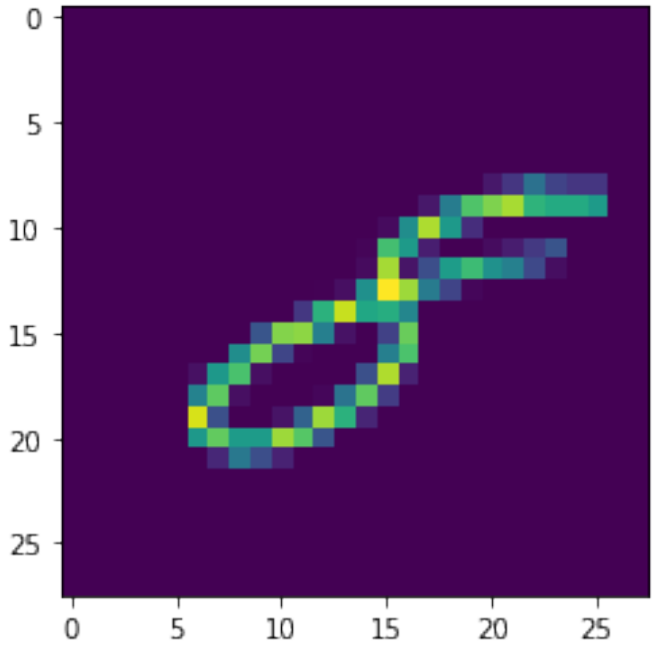

0
5
10
15
20
25
0
5
10
15
20
25

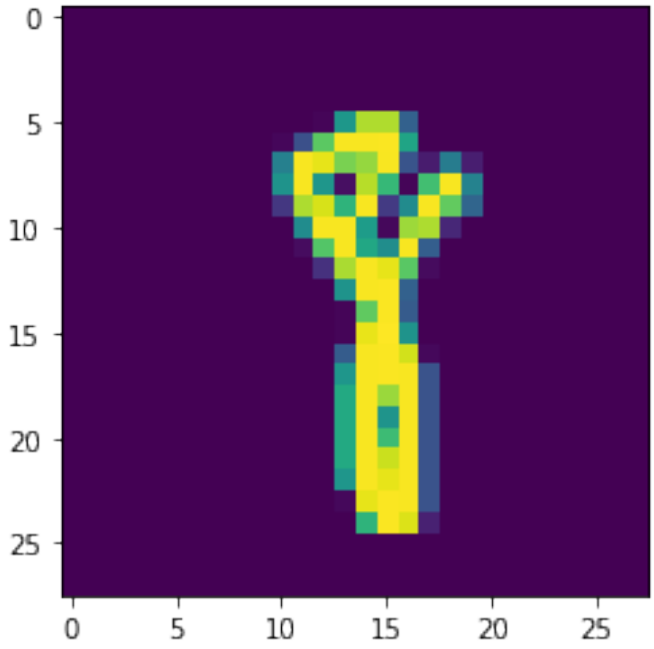

0
5
10
15
20
25
0
5
10
15
20
25

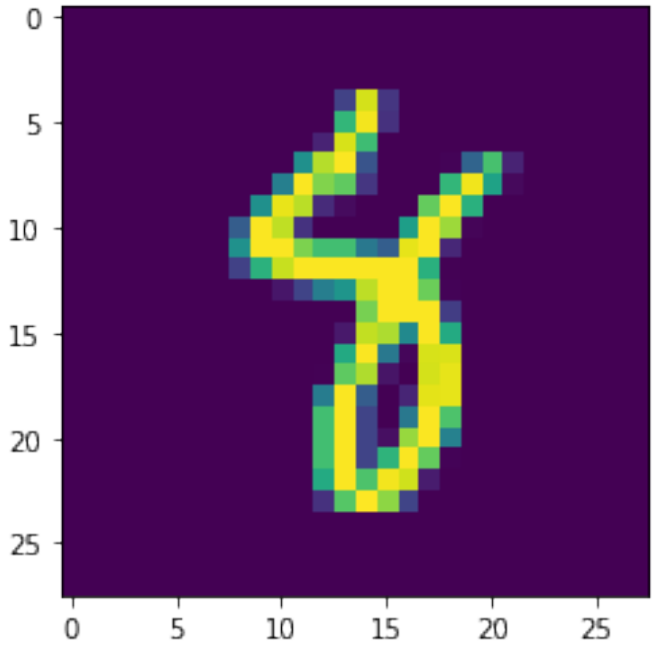

0
5
10
15
20
25
0
5
10
15
20
25

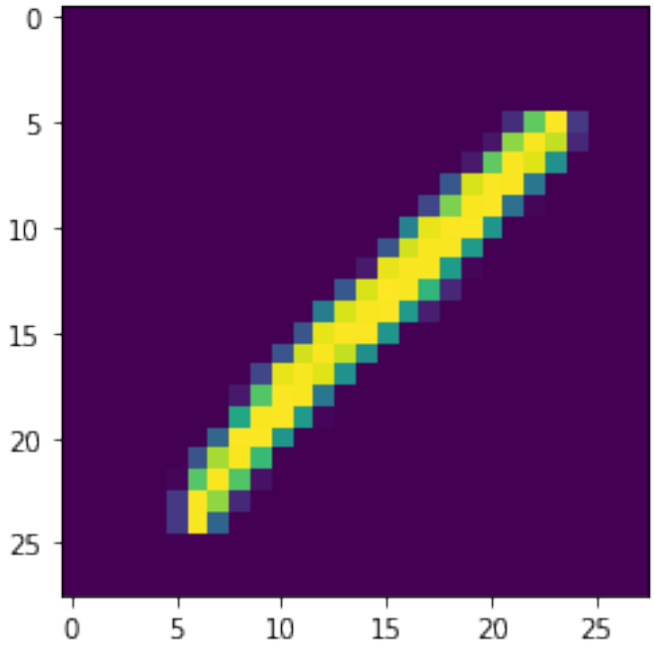

0
5
10
15
20
25
0
5
10
15
20
25

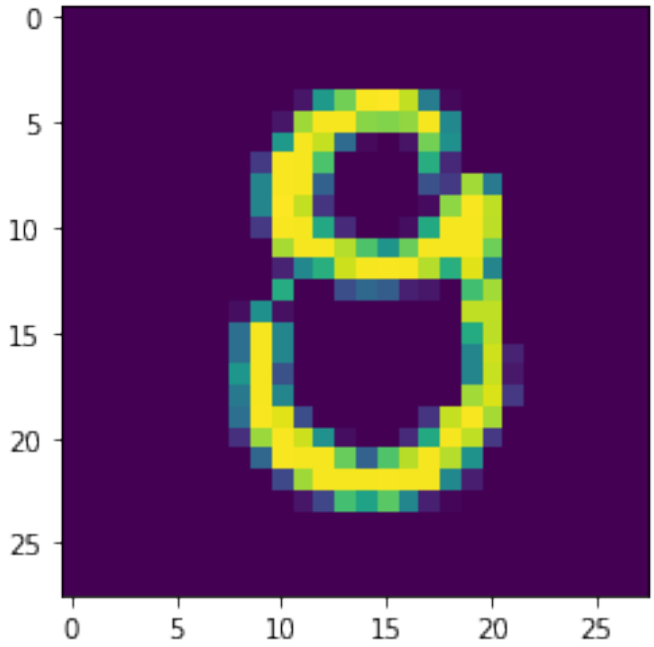

0
5
10
15
20
25
0
5
10
15
20
25

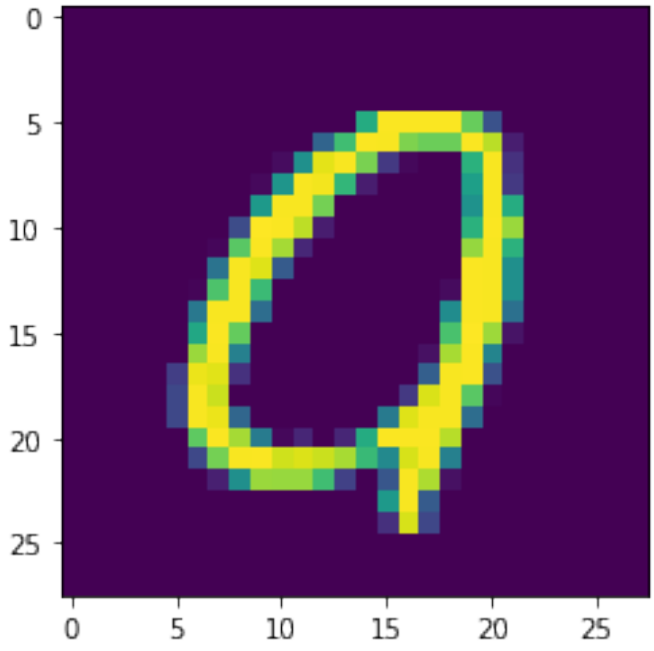

0
5
10
15
20
25
0
5
10
15
20
25

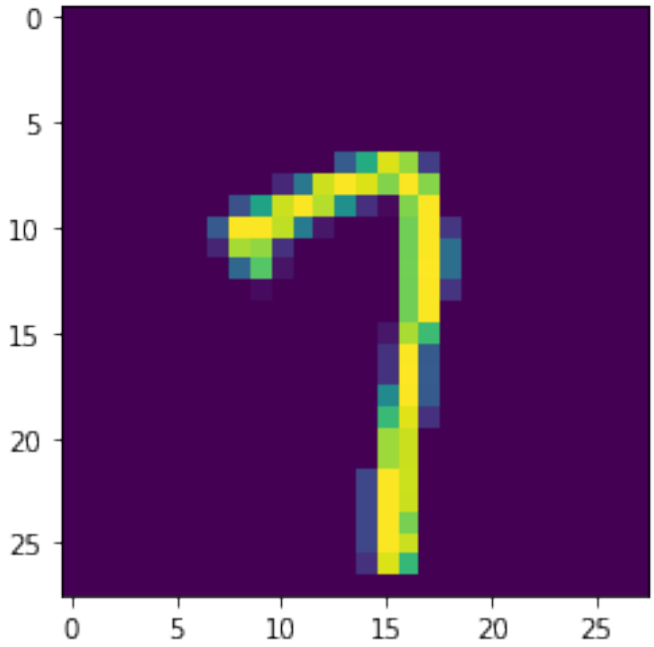

0
5
10
15
20
25
0
5
10
15
20
25

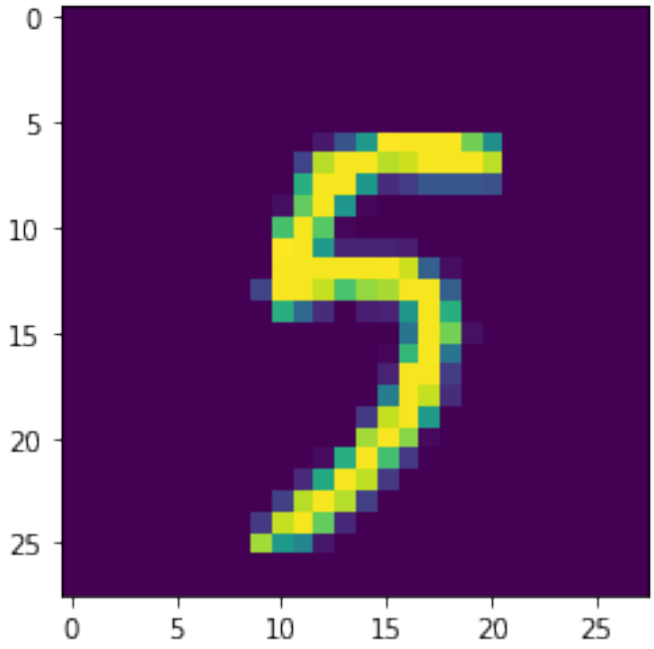

0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25


0
5
10
15
20
25
0
5
10
15
20
25





On retrouve ainsi tous les nombres qui sont écrits bizarre, le modèle a donc des difficultés à les classer. On se retrouve avec des alphas qui sont élevés pour des images difficiles à classer cela est cohérent avec le fait que ces images se trouvent proches de la frontière et ont donc des valeurs de alphas élevés.

2 dimensionnal visualization

In order to visualize the boundaries between the classes, we want to project all data into a 2 dimensional space using PCA, and then perform the classification there.

Q6. Implement a trainable kernel that performs a PCA projection followed by a Gaussian kernel and draw a scatter plot of the validation samples along with color coded region of each class (using pcolormesh or contourf)

```
from numpy import array
from numpy import mean
from numpy import cov
from numpy.linalg import eig
class PCAGaussKernel():
    def __init__(self, gamma=1.0, d=2):
        self.gamma = gamma
        self.d = d
        self.P = None
        self.mu = None



    '''
    train self.mu and self.P
    '''
    def fit(self,X):
      self.mu = mean(X, axis=0)
      X_centred = X - self.mu
      #calcule de la covariance pour les images centrées
      V = cov(X_centred.T)
      # calcule des valeurs et vecteurs propres
      values, vectors = eig(V)
      vectors=vectors.T

      idx=np.argsort(values)[::-1][:self.d]
      self.P = vectors[idx].T
      print(self.P.shape)


    '''
    return the Gram matrix of the projected samples under the Gaussian
kernel
    '''

    def __call__(self, x1, x2):


      new_x1=jnp.dot(x1-self.mu[None, :],self.P)
      new_x2=jnp.dot(x2- self.mu[None,:],self.P)
      new_x1=np.real(new_x1)
      new_x2=np.real(new_x2)

      return jnp.exp(-self.gamma * ( jnp.linalg.norm(new_x1, axis=-1,
keepdims=True)**2 + jnp.linalg.norm(new_x2, axis=-1,
keepdims=False).T**2 - 2*jnp.dot(new_x1,new_x2.T) ))
```

```
s=PCAGaussKernel(gamma=1,d=2)
s.fit(X_train)
```

```
(784, 2)
```

```
svm = OnevsAllKSVM(X_train, y_train, s)
```

```
svm.train()
y_hat = svm(X_val1)
print(error_rate(y_hat, y_val1))
```

```
0.46400002
```

```
lk
```

```
[DeviceArray(0.46400002, dtype=float32)]
```

On retrouve un taux d'erreur qui est correct pour ne garder seulement que deux dimensions. On reste bien meilleur que de l'aléatoire. Il faut maintenant tracer le contouur pour voir comment se répartisse les diférentes classes dans l'espace.

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-194-011112aa8510> in <module>()
----> 1 s(X_train,X_train)

TypeError: 'int' object is not callable
```

Q7. Modify your training procedure such that it optimizes all classifiers at each step, instead of training fully each classifier one after another and make a video of the evolution of the prediction boundary during training

J'ai déjà mis à jour tous les classifiers à chaque pas de temps dans ma précédente version de OneversusAll donc je ne le refais pas ici. Je ne finis pas les dernières questions car j'ai déjà passé beaucoup de temps sur ce tp.