In [1]:
```
# import numpy and pandas and matplotlib (as plt)
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

Purpose:

Classify MNIST using SVCs (SVM classifiers) and kernels. MNIST is a dataset of handwritten digits; the task is to classify the digits as 0-9. The images themselves are 28 x 28 pixels large.

Data Set-up

Import data and import train_test_split from sklearn.

In [2]:
```
from sklearn.model_selection import train_test_split
from sklearn.datasets import fetch_openml

# fetch "MNIST original"
data = fetch_openml('mnist_784', version=1)
```

In [3]:
```
# determine X and y
X = np.array(data.data.astype('float32'))
y = np.array(data.target.astype('int64'))

# print the shape of X and y

print("X shape: ", X.shape)
print("y shape: ", y.shape)


# Use train_test_split. Keep test at 25%.

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.25)


# The SVM algorithm runs in O(n^2) time, where n is the number of training point
# To prevent the algorithm from taking forever, take only the first 10000 data p
# from the training set and the first 2000 data points from the test set

X_train, y_train = X_train[:10000], y_train[:10000]
X_test, y_test = X_test[:2000], y_test[:2000]
```

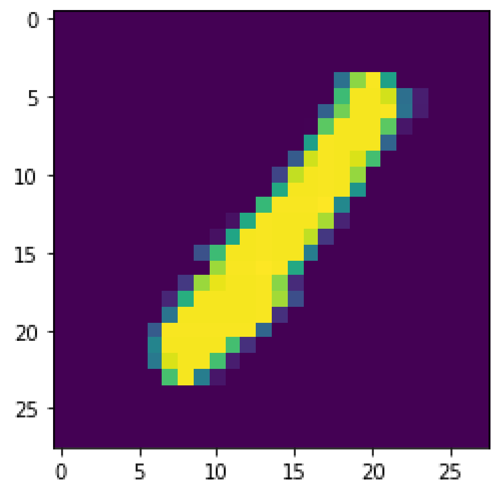
```
X shape:  (70000, 784)
y shape:  (70000,)
```

We will only use 1's and 7's for our classification problem. The following code block should filter out only the 1's and 7's:

In [4]:
```
# Make new test and train sets with only 1's and 7's

train_mask = np.array(((y_train == 1).astype(int) + (y_train == 7).astype(int)).
test_mask = np.array(((y_test == 1).astype(int) + (y_test == 7).astype(int)).ast

X_test = X_test[test_mask, :]
y_test = y_test[test_mask]
```

```python
X_train = X_train[train_mask, :]
y_train = y_train[train_mask]
```

In [5]:
```python
# Use this to visualize the dataset
# Feel free to change the index
plt.imshow(X_train[4].reshape(28,28))
plt.show()
```



Use PCA from sklearn

We will use Principal Component Analysis (PCA) to manipulate the data to make it more usable for SVC. The main idea of principal component analysis (PCA) is to reduce the dimensionality of a data set by projecting the data on to a space while still retaining as much variance in the data as possible.

In [6]:
```python
# import PCA
from sklearn.decomposition import PCA

# There are a total of 28 * 28 features (one per pixel)
# Let's project this down to 2 features using pca (2 features so we can plot out
pca = PCA(n_components=2)

# Use pca to transform X_train, X_test
X_train_pca = pca.fit_transform(X_train)
X_test_pca = pca.fit_transform(X_test)

#print the shape of X_train_pca
print(X_train_pca.shape)
```

```
(2158, 2)
```

What change do you notice between our old training data and our new one?

Answer: Our old data had 784 features, but the new data has 2

SVC and Kernels

Now we will experiment with support vector classifiers and kernels. We will need LinearSVC, SVC, and accuracy_score.

SVMs are really interesting because they have something called the dual formulation, in which the computation is expressed as training point inner products. This means that data can be lifted into higher dimensions easily with this "kernel trick". Data that is not linearly separable in a lower dimension can be linearly separable in a higher dimension - which is why we conduct the transform. Let us experiment.

A transformation that lifts the data into a higher-dimensional space is called a kernel. A polynomial kernel expands the feature space by computing all the polynomial cross terms to a specific degree.

In [7]:
```
# import SVC, LinearSVC, accuracy_score
from sklearn.svm import SVC, LinearSVC
from sklearn.metrics import accuracy_score

# fit the LinearSVC on X_train_pca and y_train and then print train accuracy and
lsvc = LinearSVC()
lsvc.fit(X_train_pca, y_train)

print('train acc: ', accuracy_score(y_train, lsvc.predict(X_train_pca)))
print('test acc: ', accuracy_score(y_test, lsvc.predict(X_test_pca)))

# use SVC with an RBF kernel. Fit this model on X_train_pca and y_train and prin
svc = SVC(kernel='rbf')
svc.fit(X_train_pca, y_train)
print('train acc: ', accuracy_score(y_train, svc.predict(X_train_pca)))
print('test acc: ', accuracy_score(y_test, svc.predict(X_test_pca)))
```

```
train acc:  0.9731232622798888
test acc:  0.9154589371980676
train acc:  0.9828544949026877
test acc:  0.9251207729468599
```

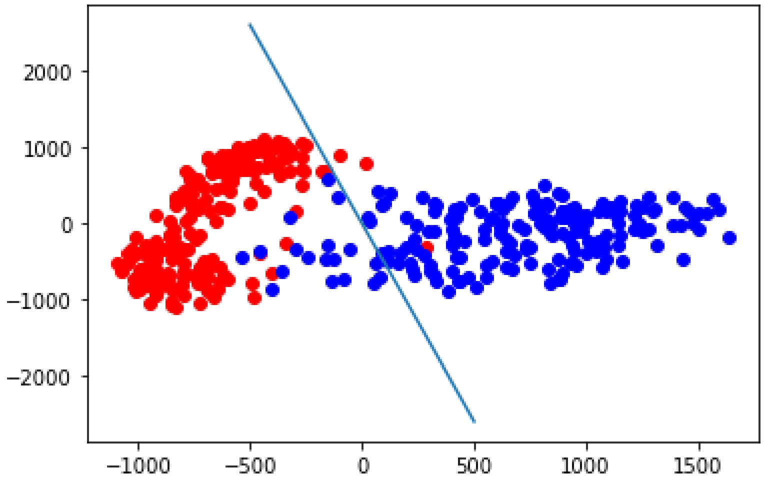
```
/opt/anaconda3/envs/hw2env/lib/python3.7/site-packages/sklearn/svm/_base.py:986:
ConvergenceWarning: Liblinear failed to converge, increase the number of iterati
ons.
  "the number of iterations.", ConvergenceWarning)
```

Visualize

Now plot out all the data points in the test set. Ones should be colored red and sevens should be colored blue. We have already provided the code to plot the decision boundary. The plot is a result of using PCA on a 784 dimensional data.

Hint: `plt.scatter`

In [8]:
```
ones = X_test_pca[y_test == 1,:]
sevens = X_test_pca[y_test == 7,:]
plt.scatter(ones[:, 0], ones[:, 1], c='red')
plt.scatter(sevens[:, 0], sevens[:, 1], c='blue')

# Code to plot the decision boundary of a linear svm

weights = lsvc.coef_[0]
x = np.linspace(-500,500,100)
y = x / weights[1] * -weights[0]
plt.plot(x,y)
plt.show()
```



Sort!

Now we're going to do a kind of hack. We've trained a linearSVM (SVC) on a binary classification problem. But what if we wanted something more regression-like? Say we wanted to score each datapoint on how "one-y" or how "seven-y" it looked. How would we do that? Check out the documentation.

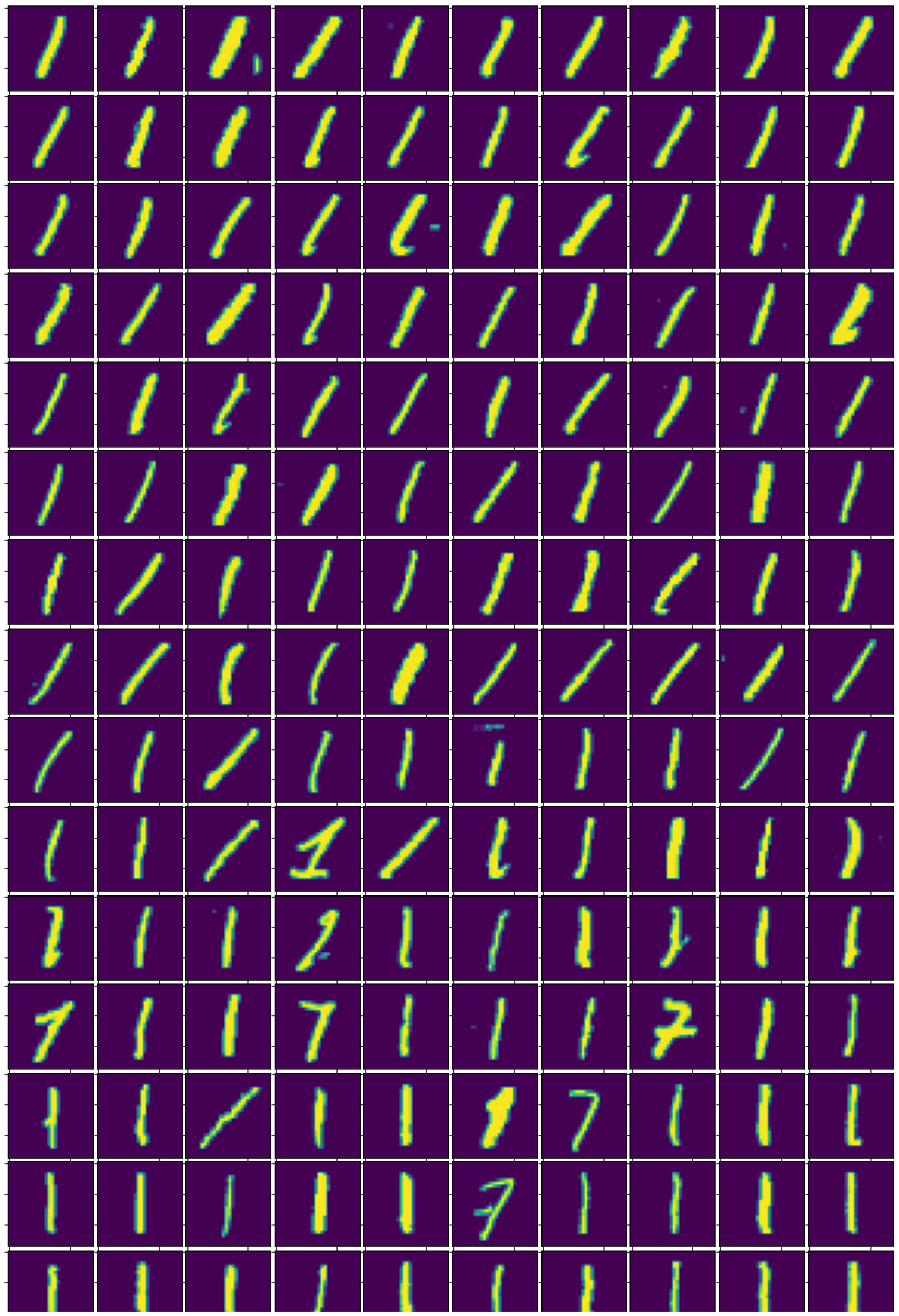
In the block below, create a list of scores for each datapoint in `X_test_pca`. Then sort `X_test` using the scores from `X_test_pca` (we're using `X_test` instead of `X_test_pca` because we want to plot the images). The block after contains code to plot out the sorted images. You should see 1's gradually turn in to 7's.

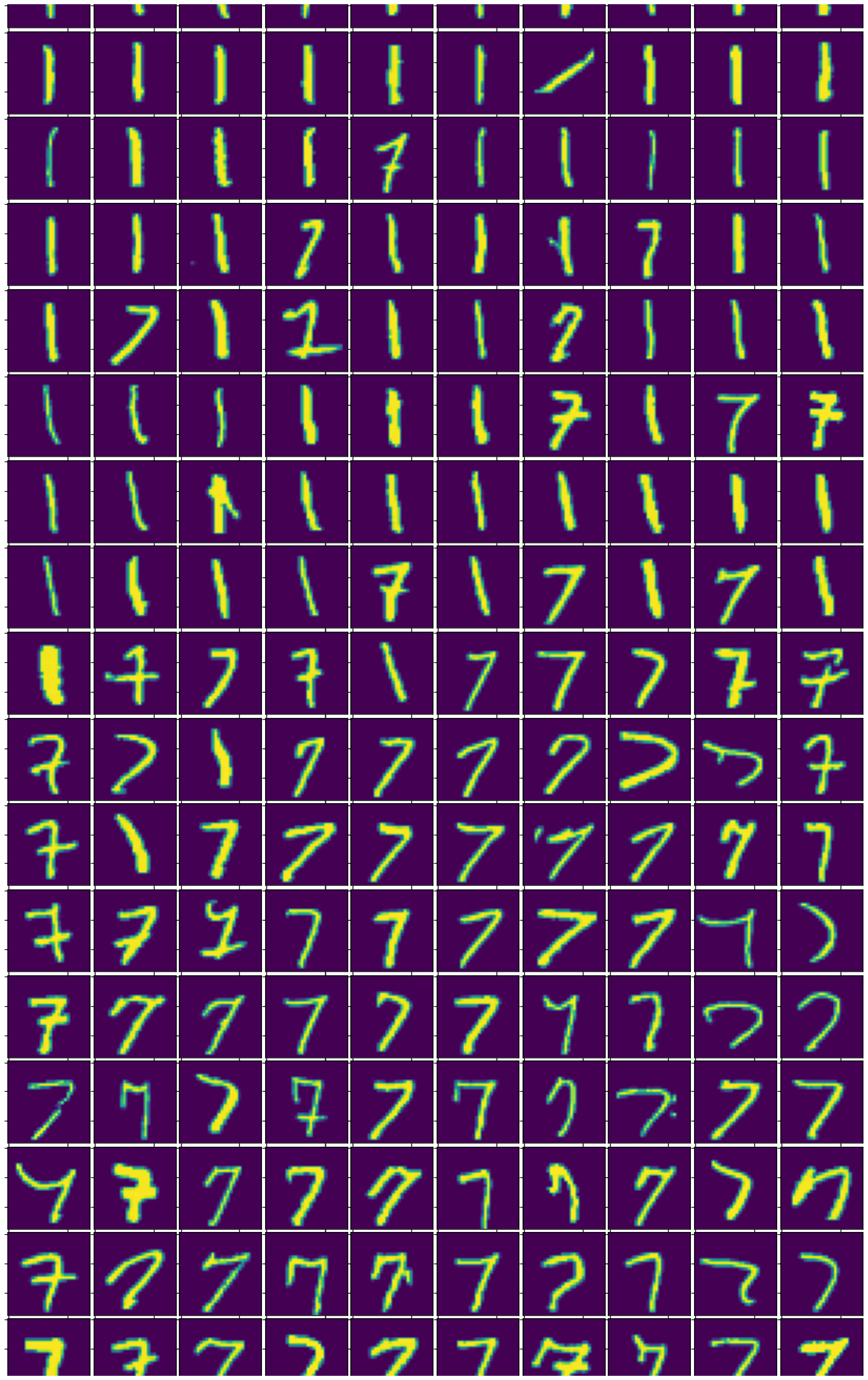
In [9]:
```
scores = lsvc.decision_function(X_test_pca)
zipped = zip(scores, X_test)
zipped = sorted(zipped)
sorted_X = [element for _, element in zipped]
```

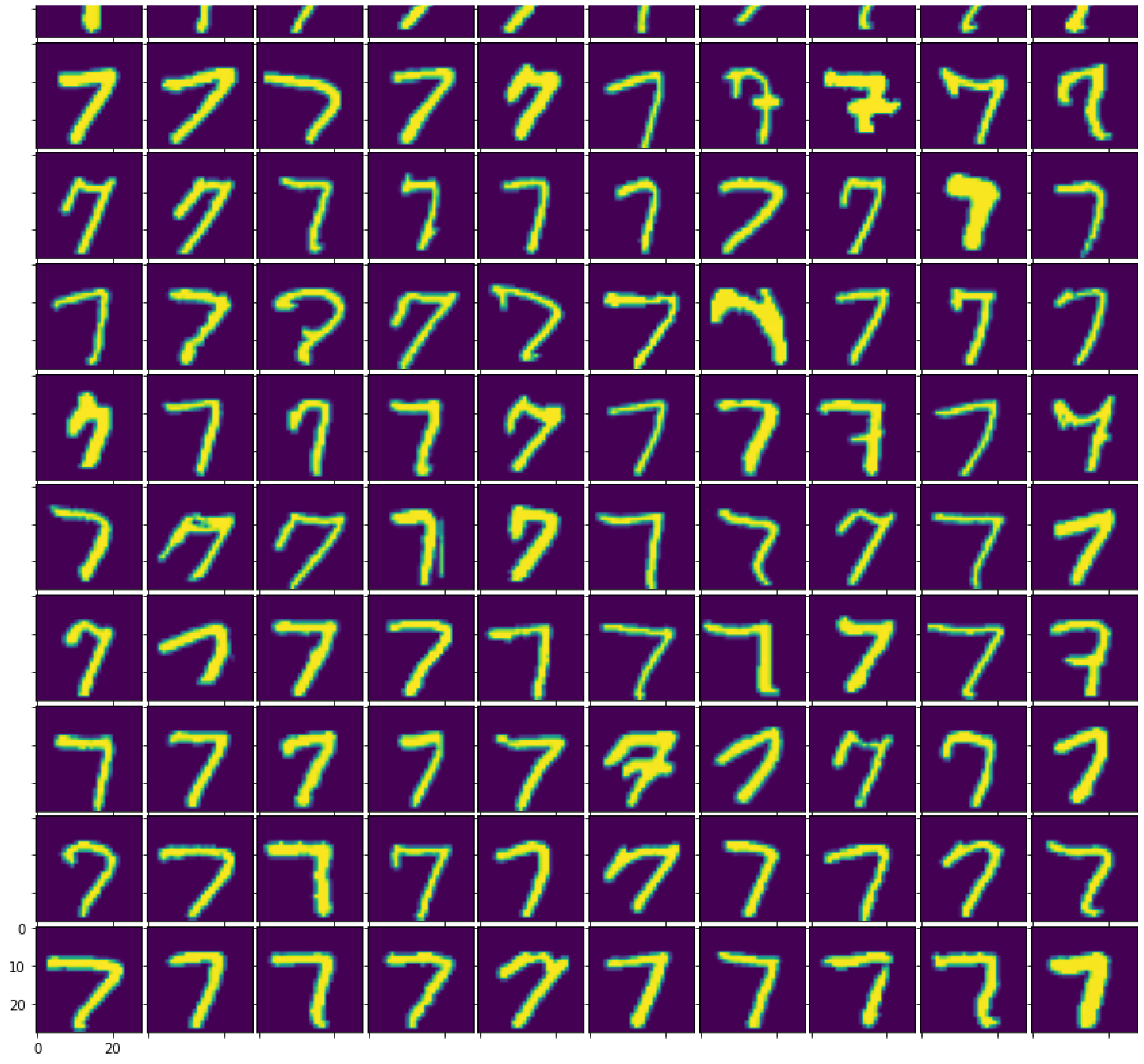
Code to plot the images (this may take some time)

In [10]:
```
from mpl_toolkits.axes_grid1 import AxesGrid

def plot(x):
    plt.imshow(x.reshape(28,28))
    plt.show()

def plot_dataset(X):
    fig = plt.figure(1, (60, 60))

    fig.subplots_adjust(left=0.05, right=0.95)

    grid = AxesGrid(fig, 141,  # similar to subplot(141)
                    nrows_ncols=(40, 10),
                    axes_pad=0.05,
                    label_mode="1",
                    )

    for i in range(400):
        grid[i].imshow(X[i].reshape(28,28))
```

```
# We're assuming sorted_X has ~200 datapoints in it
# This may take a long time to run
plot_dataset(sorted_X)
```







Conclusions

1) What is a kernel and why is it important?

A kernel is a function that maps non-linear points in one dimension to a higher dimension where they are linearalizable.

2) Can we kernelize all types of data? Why or why not?

Yes, since a kernel works on functions of data.

3) What are some pros/cons of kernels? (look into runtime)

Kernels can reduce the runtime of transferring between dimensions by a significant margin, although kernels are limited in the amount of functions they can accurately model.