

Gaussian generative models for handwritten digit classification

Recall that the 1-NN classifier yielded a 3.09% test error rate on the MNIST data set of handwritten digits. We will now see that a Gaussian generative model does almost as well, while being significantly faster and more compact.

1. Set up notebook and load in data

As usual, we start by importing the required packages and data. For this notebook we will be using the *entire* MNIST dataset. The code below defines some helper functions that will load MNIST onto your computer.

In [1]:

```
%matplotlib inline
import matplotlib.pyplot as plt
import gzip, os, sys
import numpy as np
from scipy.stats import multivariate_normal

if sys.version_info[0] == 2:
    from urllib import urlretrieve
else:
    from urllib.request import urlretrieve
```

In [2]:

```
# Function that downloads a specified MNIST data file from Yann Le Cun's website
def download(filename, source='http://yann.lecun.com/exdb/mnist/'):
    print("Downloading %s" % filename)
    urlretrieve(source + filename, filename)

# Invokes download() if necessary, then reads in images
def load_mnist_images(filename):
    if not os.path.exists(filename):
        download(filename)
    with gzip.open(filename, 'rb') as f:
        data = np.frombuffer(f.read(), np.uint8, offset=16)
    data = data.reshape(-1,784)
    return data

def load_mnist_labels(filename):
    if not os.path.exists(filename):
        download(filename)
    with gzip.open(filename, 'rb') as f:
        data = np.frombuffer(f.read(), np.uint8, offset=8)
    return data
```

Now load in the training set and test set

```
In [3]: ## Load the training set
train_data = load_mnist_images('train-images-idx3-ubyte.gz')
train_labels = load_mnist_labels('train-labels-idx1-ubyte.gz')

## Load the testing set
test_data = load_mnist_images('t10k-images-idx3-ubyte.gz')
test_labels = load_mnist_labels('t10k-labels-idx1-ubyte.gz')
```

Downloading train-images-idx3-ubyte.gz
 Downloading train-labels-idx1-ubyte.gz
 Downloading t10k-images-idx3-ubyte.gz
 Downloading t10k-labels-idx1-ubyte.gz

```
In [10]: # choose 50000 indexes
np.random.seed(123)
idx_tr = np.random.choice(60000, 50000, replace=False)
mask_tr = np.zeros(60000, dtype=bool)
# generate a bit mask for training data
mask_tr[idx_tr] = 1
# generate inverse bitmask for validation data
mask_va = ~mask_tr

tr_X, tr_y = train_data[mask_tr], train_labels[mask_tr]
va_X, va_y = train_data[mask_va], train_labels[mask_va]

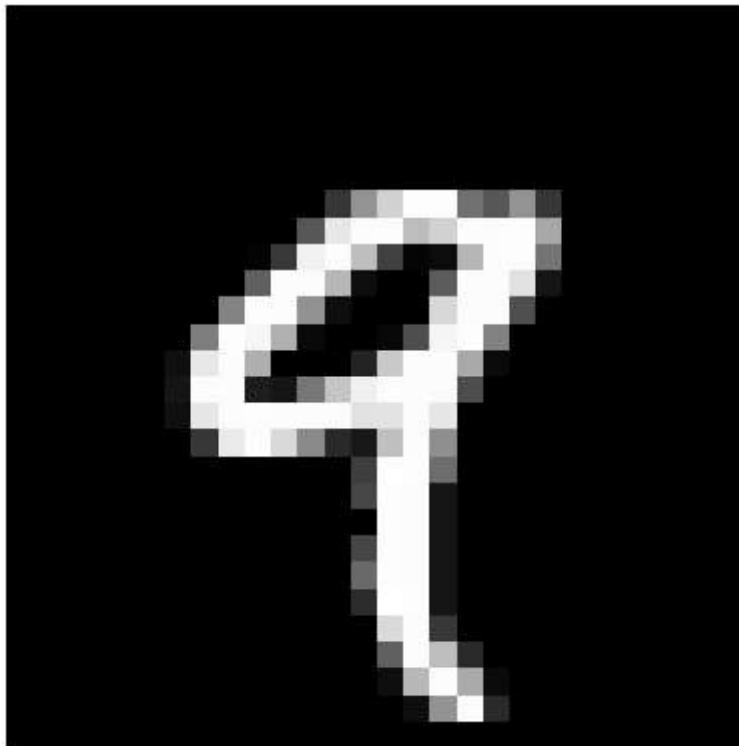
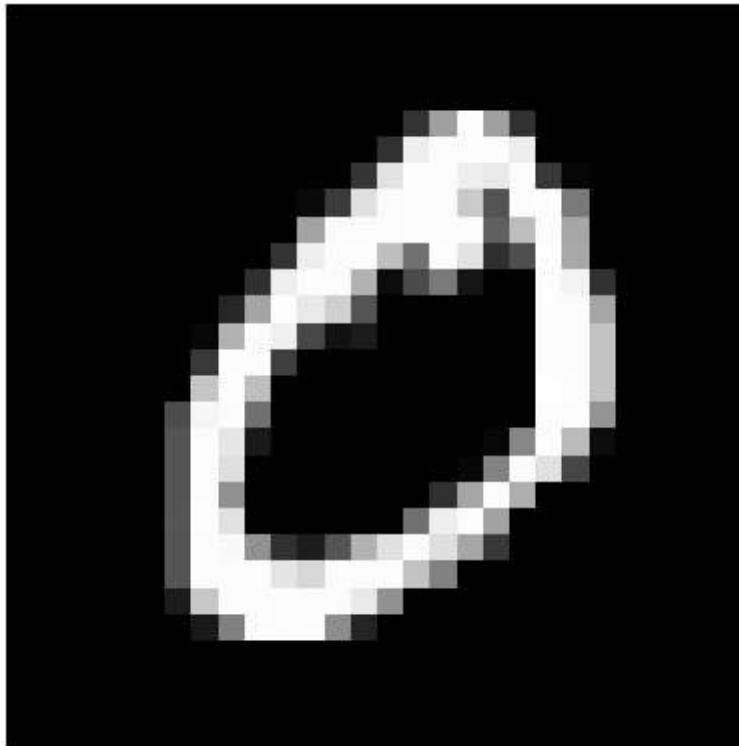
print(tr_X.shape, tr_y.shape, va_X.shape, va_y.shape)
```

(50000, 784) (50000,) (10000, 784) (10000,)

The function **displaychar** shows a single MNIST digit. To do this, it first has to reshape the 784-dimensional vector into a 28x28 image.

```
In [11]: def displaychar(image):
    plt.imshow(np.reshape(image, (28,28)), cmap=plt.cm.gray)
    plt.axis('off')
    plt.show()
```

```
In [13]: displaychar(tr_X[0])
displaychar(va_X[1])
```



2. Fit a Gaussian generative model to the training data

For you to do: Define a function, **fit_generative_model**, that takes as input a training set (data `x` and labels `y`) and fits a Gaussian generative model to it. It should return the parameters of this generative model; for each label `j = 0, 1, ..., 9`, we have:

- `pi[j]` : the frequency of that label
- `mu[j]` : the 784-dimensional mean vector
- `sigma[j]` : the 784x784 covariance matrix

This means that `pi` is 10x1, `mu` is 10x784, and `sigma` is 10x784x784.

We have already seen how to fit a Gaussian generative model in the Winery example, but now there is an added ingredient. The empirical covariances are very likely to be singular (or close to singular), which means that we won't be able to do calculations with them. Thus it is important to **regularize** these matrices. The standard way of doing this is to add `cI` to them, where `c` is some constant and `I` is the 784-dimensional identity matrix. (To put it another way, we compute the empirical covariances and then increase their diagonal entries by some constant `c`.)

This modification is guaranteed to yield covariance matrices that are non-singular, for any `c > 0`, no matter how small. But this doesn't mean that we should make `c` as small as possible. Indeed, `c` is now a parameter, and by setting it appropriately, we can improve the performance of the model. We will study **regularization** in greater detail over the coming weeks.

Your routine needs to choose a good setting of `c`. Crucially, this needs to be done using the training set alone. So you might try setting aside part of the training set as a validation set, or using some kind of cross-validation.

```
In [14]: def fit_generative_model(x, y, c):
    k = 10 # Labels 0,1,...,k-1
    d = (x.shape)[1] # number of features
    mu = np.zeros((k,d))
    sigma = np.zeros((k,d,d))
    pi = np.zeros(k)
    #####
    ### Your code goes here
    #####
    for num in range(k):
        idxs = np.nonzero(y == num)[0]
        vals = x[idxs]
        mu[num,:] = np.mean(vals, axis=0)
        sigma[num,:,:] = np.cov(vals.T) + c * np.eye(784)
        pi[num] = len(idxs) / len(x)

    # Halt and return parameters
    return mu, sigma, pi
```

Okay, let's try out your function. In particular, we will use **displaychar** to visualize the means of the Gaussians for the first three digits. You can try the other digits on your own.

```
In [18]: # Fit `c` parameter
cs = [2**i for i in range(-2, 18)]
def run_vals(cs):
```

```
# This is basically copied from the starter code, changing around the datasets
k = 10
for c in cs:
    mu, sigma, pi = fit_generative_model(tr_X, tr_y, c = c)
    score = np.zeros((len(va_y), k))
    for label in range(0, k):
        rv = multivariate_normal(mean=mu[label], cov=sigma[label])
        for i in range(0, len(va_y)):
            score[i, label] = np.log(pi[label]) + rv.logpdf(va_X[i, :])
    predictions = np.argmax(score, axis=1)
# Finally, tally up score
errors = np.sum(predictions != va_y)
print(f"c={c}, Err:{str(errors)}/10000, {errors/10000}")
# Get Performance metrics
run_vals(cs)
```

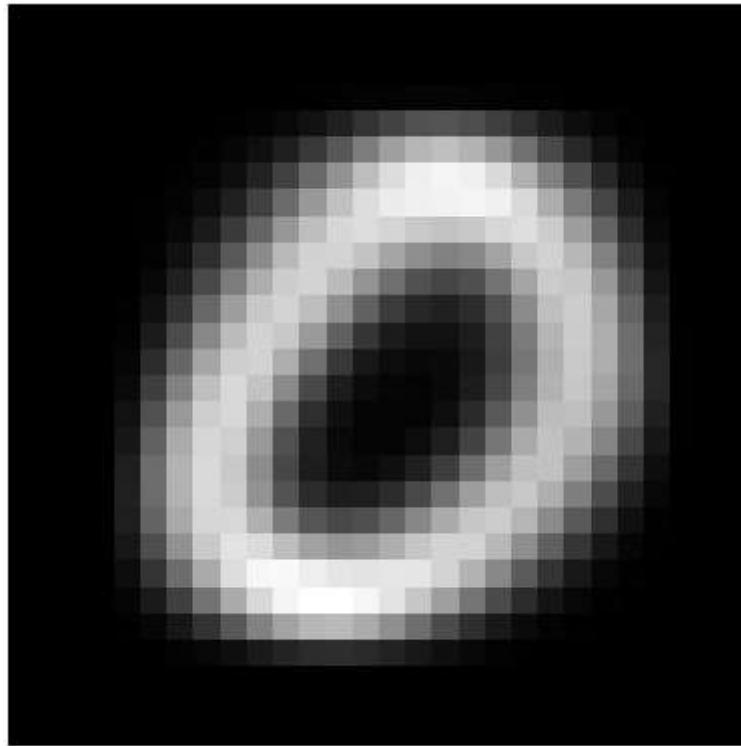
c=0.25, Err:1810/10000, 0.181
 c=0.5, Err:1732/10000, 0.1732
 c=1, Err:1657/10000, 0.1657
 c=2, Err:1571/10000, 0.1571
 c=4, Err:1471/10000, 0.1471
 c=8, Err:1365/10000, 0.1365
 c=16, Err:1235/10000, 0.1235
 c=32, Err:1110/10000, 0.111
 c=64, Err:963/10000, 0.0963
 c=128, Err:838/10000, 0.0838
 c=256, Err:715/10000, 0.0715
 c=512, Err:619/10000, 0.0619
 c=1024, Err:551/10000, 0.0551
 c=2048, Err:494/10000, 0.0494
 c=4096, Err:480/10000, 0.048
 c=8192, Err:524/10000, 0.0524
 c=16384, Err:651/10000, 0.0651
 c=32768, Err:871/10000, 0.0871
 c=65536, Err:1228/10000, 0.1228
 c=131072, Err:1751/10000, 0.1751

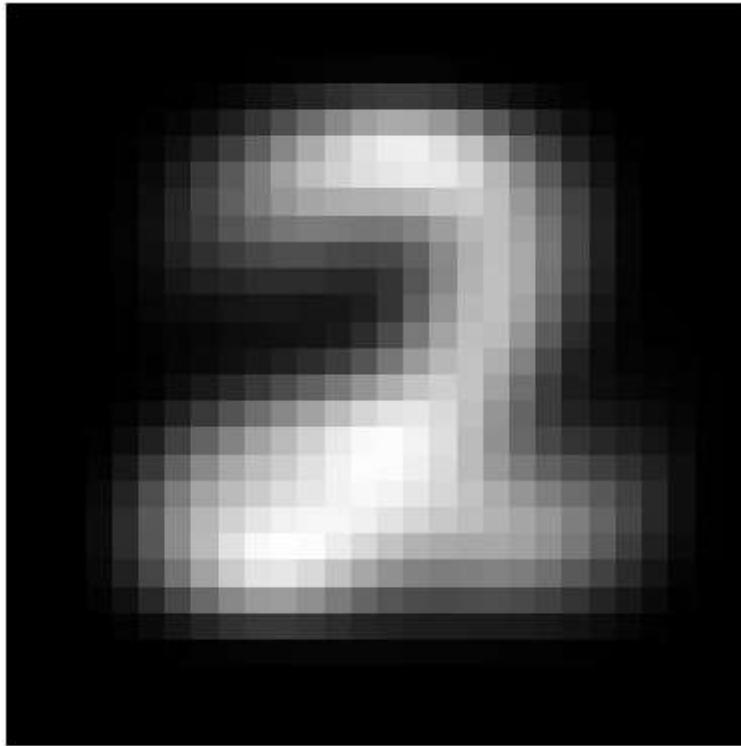
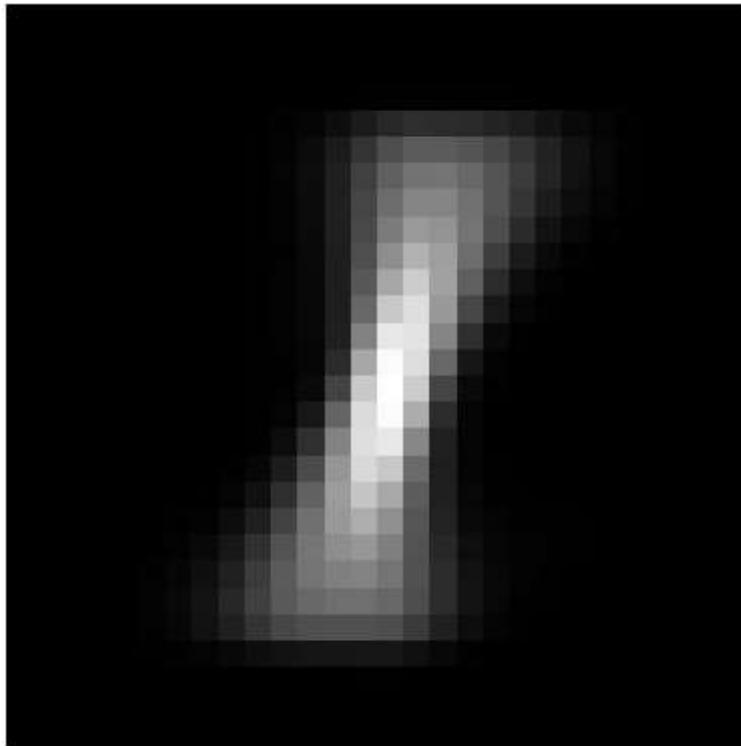
In [19]: cs = np.linspace(1024, 8192, 20).astype(np.int32)
 run_vals(cs)

```
c=1024, Err:551/10000, 0.0551
c=1401, Err:520/10000, 0.052
c=1778, Err:501/10000, 0.0501
c=2155, Err:492/10000, 0.0492
c=2533, Err:484/10000, 0.0484
c=2910, Err:485/10000, 0.0485
c=3287, Err:478/10000, 0.0478
c=3664, Err:472/10000, 0.0472
c=4042, Err:480/10000, 0.048
c=4419, Err:476/10000, 0.0476
c=4796, Err:477/10000, 0.0477
c=5173, Err:481/10000, 0.0481
c=5551, Err:488/10000, 0.0488
c=5928, Err:496/10000, 0.0496
c=6305, Err:495/10000, 0.0495
c=6682, Err:501/10000, 0.0501
c=7060, Err:510/10000, 0.051
c=7437, Err:519/10000, 0.0519
c=7814, Err:518/10000, 0.0518
c=8192, Err:524/10000, 0.0524
```

In [16]:

```
c = 0.1
mu, sigma, pi = fit_generative_model(tr_X, tr_y, c = 0.1)
displaychar(mu[0])
displaychar(mu[1])
displaychar(mu[2])
```





3. Make predictions on test data

Now let's see how many errors your model makes on the test set.

```
In [20]: # Compute Log Pr(Label|image) for each [test image, label] pair.  
c = 3664  
k = 10
```

```

mu, sigma, pi = fit_generative_model(tr_X, tr_y, c = c)
score = np.zeros((len(test_labels),k))
for label in range(0,k):
    rv = multivariate_normal(mean=mu[label], cov=sigma[label])
    for i in range(0,len(test_labels)):
        score[i,label] = np.log(pi[label]) + rv.logpdf(test_data[i,:])
predictions = np.argmax(score, axis=1)
# Finally, tally up score
errors = np.sum(predictions != test_labels)
print(f"c={c}, Err:{str(errors)}/10000, {errors/10000}")
# print("Your model makes " + str(errors) + " errors out of 10000")

```

c=3664, Err:438/10000, 0.0438

Best Results:

c=3664 = 438/10000, 0.0438 Error Rate

4. Quick exercises

You will need to answer variants of these questions as part of this week's assignment.

Exercise 1: What happens if you do not regularize the covariance matrices?

Answer: An error is thrown saying that the covariance matrix is not positive semidefinite

```

In [21]: mu, sigma, pi = fit_generative_model(train_data, train_labels, c = 0)
score = np.zeros((len(test_labels),k))
for label in range(0,k):
    rv = multivariate_normal(mean=mu[label], cov=sigma[label])
    for i in range(0,len(test_labels)):
        score[i,label] = np.log(pi[label]) + rv.logpdf(test_data[i,:])
predictions = np.argmax(score, axis=1)
# Finally, tally up score
errors = np.sum(predictions != test_labels)
print(f"c={c}, Err:{str(errors)}/10000, {errors/10000}")

```

```

-----
LinAlgError                                                 Traceback (most recent call last)
Cell In[21], line 4
    2 score = np.zeros((len(test_labels),k))
    3 for label in range(0,k):
----> 4     rv = multivariate_normal(mean=mu[label], cov=sigma[label])
      5     for i in range(0,len(test_labels)):
      6         score[i,label] = np.log(pi[label]) + rv.logpdf(test_data[i,:])

File c:\Users\James\anaconda3\envs\mlenv\lib\site-packages\scipy\stats\_multivariate.py:393, in multivariate_normal_gen.__call__(self, mean, cov, allow_singular, seed)
    388 def __call__(self, mean=None, cov=1, allow_singular=False, seed=None):
    389     """Create a frozen multivariate normal distribution.
    390
    391     See `multivariate_normal_frozen` for more information.
    392     """
--> 393     return multivariate_normal_frozen(mean, cov,
    394                                     allow_singular=allow_singular,
    395                                     seed=seed)

File c:\Users\James\anaconda3\envs\mlenv\lib\site-packages\scipy\stats\_multivariate.py:834, in multivariate_normal_frozen.__init__(self, mean, cov, allow_singular, seed, maxpts, abseps, releps)
    791     """Create a frozen multivariate normal distribution.
    792
    793 Parameters
    (...)

    830
    831     """
    832 self._dist = multivariate_normal_gen(seed)
    833 self.dim, self.mean, self.cov_object = (
--> 834     self._dist._process_parameters(mean, cov, allow_singular))
    835 self.allow_singular = allow_singular or self.cov_object._allow_singular
    836 if not maxpts:

File c:\Users\James\anaconda3\envs\mlenv\lib\site-packages\scipy\stats\_multivariate.py:417, in multivariate_normal_gen._process_parameters(self, mean, cov, allow_singular)
    410 dim, mean, cov = self._process_parameters_psd(None, mean, cov)
    411 # After input validation, some methods then processed the arrays
    412 # with a `_PSD` object and used that to perform computation.
    413 # To avoid branching statements in each method depending on whether
    414 # `cov` is an array or `Covariance` object, we always process the
    415 # array with `_PSD`, and then use wrapper that satisfies the
    416 # `Covariance` interface, `CovViaPSD`.
--> 417 psd = _PSD(cov, allow_singular=allow_singular)
    418 cov_object = _covariance.CovViaPSD(psd)
    419 return dim, mean, cov_object

File c:\Users\James\anaconda3\envs\mlenv\lib\site-packages\scipy\stats\_multivariate.py:172, in _PSD.__init__(self, M, cond, rcond, lower, check_finite, allow_singular)
    169 if len(d) < len(s) and not allow_singular:
    170     msg = ("When `allow_singular` is False, the input matrix must be "
    171           "symmetric positive definite.")

```

```
--> 172      raise np.linalg.LinAlgError(msg)
173 s_pinv = _pinv_1d(s, eps)
174 U = np.multiply(u, np.sqrt(s_pinv))
```

LinAlgError: When `allow_singular` is False, the input matrix must be symmetric positive definite.

Exercise 2: What happens if you set the value of `c` too high, for instance to one billion? Do you understand why this happens?

The error skyrockets to around 0.9. This means that the variances are all so insanely high and (relatively) equal that the classifier is just guessing at this point (10% to be right, $1 - 0.1 = 0.9$ Error)

```
In [22]: c = 1E9
mu, sigma, pi = fit_generative_model(train_data, train_labels, c = 1E9)
score = np.zeros((len(test_labels),k))
for label in range(0,k):
    rv = multivariate_normal(mean=mu[label], cov=sigma[label])
    for i in range(0,len(test_labels)):
        score[i,label] = np.log(pi[label]) + rv.logpdf(test_data[i,:])
predictions = np.argmax(score, axis=1)
# Finally, tally up score
errors = np.sum(predictions != test_labels)
print(f"c={c}, Err:{str(errors)}/10000, {errors/10000}")

c=1000000000.0, Err:8865/10000, 0.8865
```

Exercise 3: What value of `c` did you end up using? How many errors did your model make on the training set?

Best Results:

`c=3664 = 438/10000, 0.0438` Error Rate

If you have the time: We have talked about using the same regularization constant `c` for all ten classes. What about using a different value of `c` for each class? How would you go about choosing these? Can you get better performance in this way?

Q15 part d

```
In [23]: c=3664
mu, sigma, pi = fit_generative_model(train_data, train_labels, c = c)
score = np.zeros((len(test_labels),k))
for label in range(0,k):
    rv = multivariate_normal(mean=mu[label], cov=sigma[label])
    for i in range(0,len(test_labels)):
        score[i,label] = np.log(pi[label]) + rv.logpdf(test_data[i,:])
predictions = np.argmax(score, axis=1)
```

```
In [24]: idx_wrong = np.nonzero(predictions != test_labels)[0]
```

```

np.random.seed(123)
# Sample 5 randomly out of those with differing predictions from test_labels
samp = np.random.choice(idx_wrong, 5, replace=False)

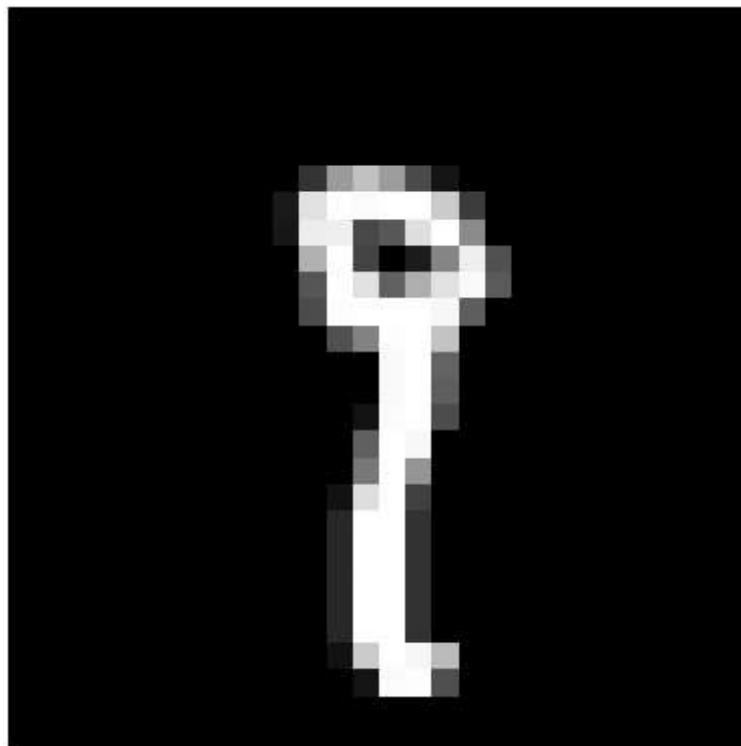
for idx in samp:
    log_prob = np.log(pi[label]) + rv.logpdf(test_data[i,:])
    print(f"--- Test Sample {idx}, True label {test_labels[idx]}, Predicted Label {pr
    score_row = score[idx]
    #  $P(x) = \sum_y P(x,y)$ 
    #  $\text{Log}(P(x)) = \text{Log}(\sum_y P(x,y))$ 
    #  $P(x,y) = \text{score}[x_i, y]$ 
    log_px = -np.inf
    for s in score_row:
        log_px = np.logaddexp(log_px, s)
    for i in range(10):
        # The score[idx,i] is Log(P(x/y)P(y)), Log(P(x,y))
        #  $P(y/x) = P(x,y)/P(x)$ ,  $\text{Log}(P(y/x)) = \text{Log}(P(x,y)) - \text{Log}(P(x))$ 
        print(f"\t[{i}], log(P({i}|x)) = {score_row[i] - log_px:10.4f}, P({i}|x) = {np.

displaychar(test_data[idx])
print("== END Test Sample {idx}")

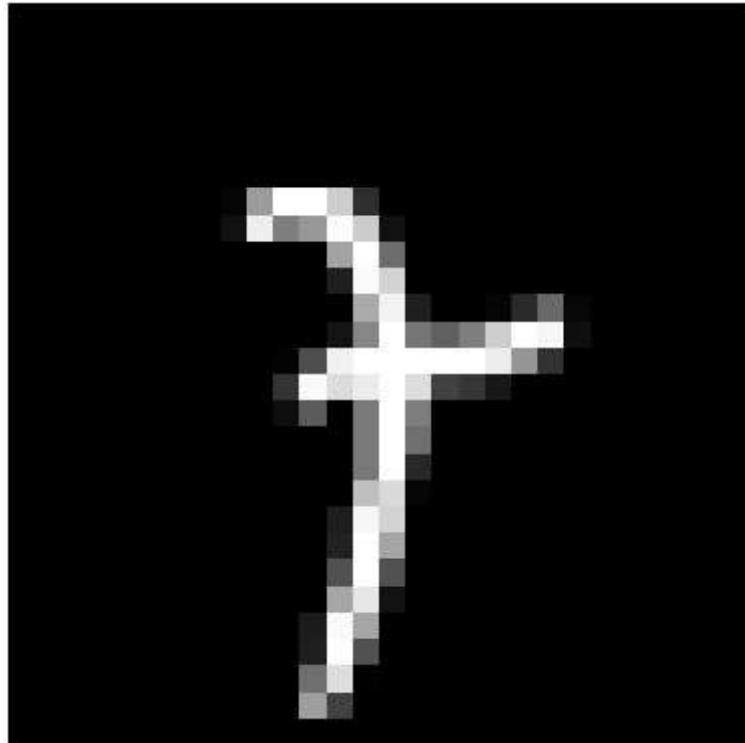
```

== Test Sample 320, True label 9, Predicted Label 1

[0], log(P(0 x)) =	-134.0400,	P(0 x) = 0.000000
[1], log(P(1 x)) =	-0.0000,	P(1 x) = 0.999985
[2], log(P(2 x)) =	-86.0809,	P(2 x) = 0.000000
[3], log(P(3 x)) =	-73.4812,	P(3 x) = 0.000000
[4], log(P(4 x)) =	-44.5374,	P(4 x) = 0.000000
[5], log(P(5 x)) =	-79.1180,	P(5 x) = 0.000000
[6], log(P(6 x)) =	-121.4409,	P(6 x) = 0.000000
[7], log(P(7 x)) =	-11.1198,	P(7 x) = 0.000015
[8], log(P(8 x)) =	-30.0164,	P(8 x) = 0.000000
[9], log(P(9 x)) =	-22.9886,	P(9 x) = 0.000000



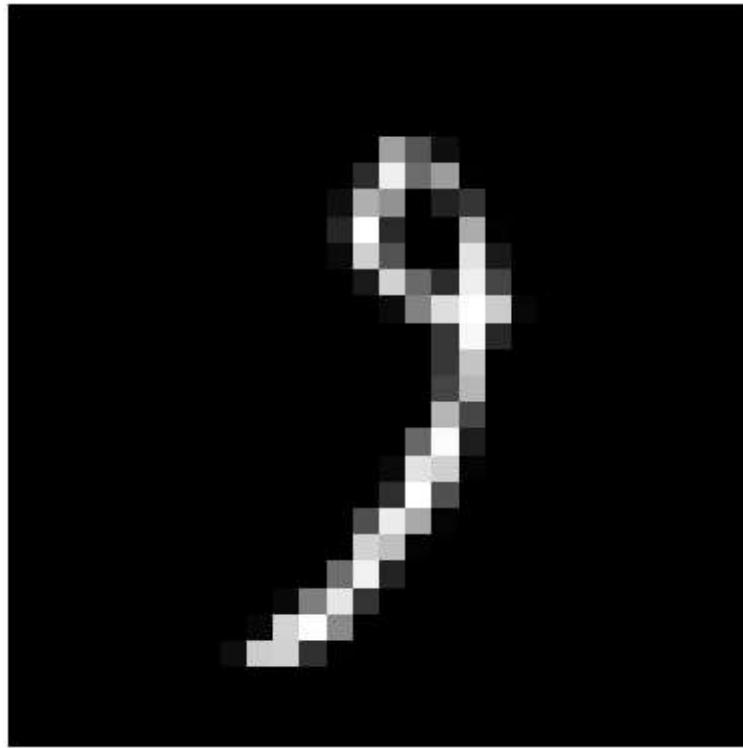
```
==== END Test Sample 320
==== Test Sample 4966, True label 7, Predicted Label 9
[0], log(P(0|x)) = -92.3110, P(0|x) = 0.000000
[1], log(P(1|x)) = -6.8519, P(1|x) = 0.001057
[2], log(P(2|x)) = -49.8557, P(2|x) = 0.000000
[3], log(P(3|x)) = -39.9352, P(3|x) = 0.000000
[4], log(P(4|x)) = -12.7763, P(4|x) = 0.000003
[5], log(P(5|x)) = -41.4150, P(5|x) = 0.000000
[6], log(P(6|x)) = -49.6453, P(6|x) = 0.000000
[7], log(P(7|x)) = -1.7935, P(7|x) = 0.166378
[8], log(P(8|x)) = -13.0910, P(8|x) = 0.000002
[9], log(P(9|x)) = -0.1833, P(9|x) = 0.832559
```



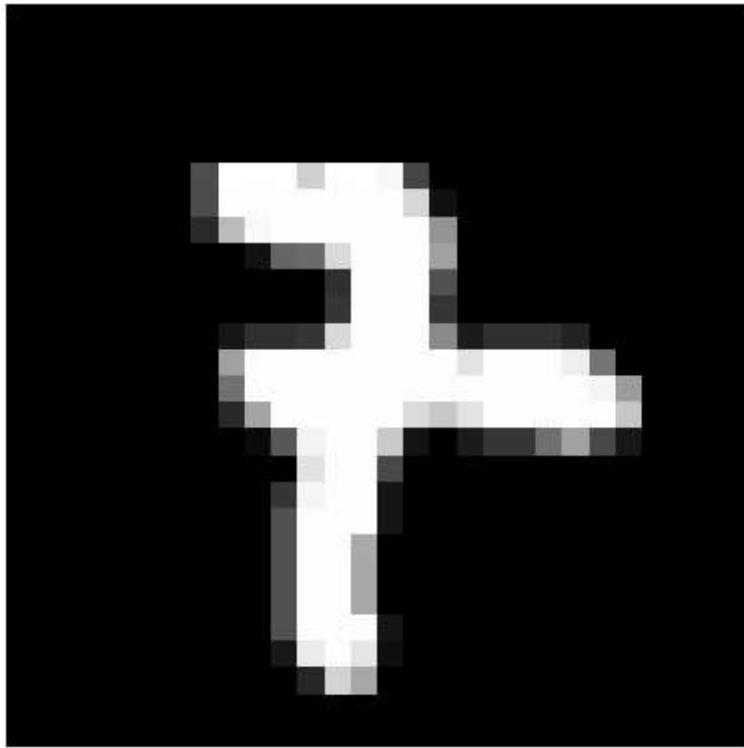
```
==== END Test Sample 4966
==== Test Sample 2129, True label 9, Predicted Label 2
[0], log(P(0|x)) = -58.5577, P(0|x) = 0.000000
[1], log(P(1|x)) = -277.8698, P(1|x) = 0.000000
[2], log(P(2|x)) = 0.0000, P(2|x) = 1.000000
[3], log(P(3|x)) = -34.4194, P(3|x) = 0.000000
[4], log(P(4|x)) = -74.5441, P(4|x) = 0.000000
[5], log(P(5|x)) = -64.0238, P(5|x) = 0.000000
[6], log(P(6|x)) = -114.5576, P(6|x) = 0.000000
[7], log(P(7|x)) = -89.2343, P(7|x) = 0.000000
[8], log(P(8|x)) = -30.1542, P(8|x) = 0.000000
[9], log(P(9|x)) = -59.2533, P(9|x) = 0.000000
```



```
==> END Test Sample 2129
==> Test Sample 3060, True label 9, Predicted Label 1
    [0], log(P(0|x)) = -35.9062, P(0|x) = 0.000000
    [1], log(P(1|x)) = -0.0563, P(1|x) = 0.945272
    [2], log(P(2|x)) = -38.8763, P(2|x) = 0.000000
    [3], log(P(3|x)) = -32.3519, P(3|x) = 0.000000
    [4], log(P(4|x)) = -18.7199, P(4|x) = 0.000000
    [5], log(P(5|x)) = -42.4339, P(5|x) = 0.000000
    [6], log(P(6|x)) = -56.9203, P(6|x) = 0.000000
    [7], log(P(7|x)) = -2.9569, P(7|x) = 0.051981
    [8], log(P(8|x)) = -35.9937, P(8|x) = 0.000000
    [9], log(P(9|x)) = -5.8975, P(9|x) = 0.002746
```



```
==> END Test Sample 3060
==> Test Sample 4498, True label 7, Predicted Label 4
    [0], log(P(0|x)) = -119.1755, P(0|x) = 0.000000
    [1], log(P(1|x)) = -162.8433, P(1|x) = 0.000000
    [2], log(P(2|x)) = -33.1990, P(2|x) = 0.000000
    [3], log(P(3|x)) = -33.0528, P(3|x) = 0.000000
    [4], log(P(4|x)) = -0.5702, P(4|x) = 0.565393
    [5], log(P(5|x)) = -56.1085, P(5|x) = 0.000000
    [6], log(P(6|x)) = -93.6327, P(6|x) = 0.000000
    [7], log(P(7|x)) = -0.8333, P(7|x) = 0.434607
    [8], log(P(8|x)) = -17.6050, P(8|x) = 0.000000
    [9], log(P(9|x)) = -46.9964, P(9|x) = 0.000000
```



==== END Test Sample 4498